

Xpert.press

Die Reihe **Xpert.press** vermittelt Professionals in den Bereichen Softwareentwicklung, Internettechnologie und IT-Management aktuell und kompetent relevantes Fachwissen über Technologien und Produkte zur Entwicklung und Anwendung moderner Informationstechnologien.

Walter Kriha · Roland Schmitz

Sichere Systeme

Konzepte, Architekturen und Frameworks

Prof. Dr. Walter Kriha
Hochschule der Medien
Nobelstraße 10
70569 Stuttgart
kriha@hdm-stuttgart.de

Prof. Dr. Roland Schmitz
Hochschule der Medien
Nobelstraße 10
70569 Stuttgart
schmitz@hdm-stuttgart.de

ISBN 978-3-540-78958-1

e-ISBN 978-3-540-78959-8

DOI 10.1007/978-3-540-78959-8

Xpert.press ISSN 1439-5428

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

© 2009 Springer-Verlag Berlin Heidelberg

Dieses Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdrucks, des Vortrags, der Entnahme von Abbildungen und Tabellen, der Funksendung, der Mikroverfilmung oder der Vervielfältigung auf anderen Wegen und der Speicherung in Datenverarbeitungsanlagen, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Eine Vervielfältigung dieses Werkes oder von Teilen dieses Werkes ist auch im Einzelfall nur in den Grenzen der gesetzlichen Bestimmungen des Urheberrechtsgesetzes der Bundesrepublik Deutschland vom 9. September 1965 in der jeweils geltenden Fassung zulässig. Sie ist grundsätzlich vergütungspflichtig. Zuwiderhandlungen unterliegen den Strafbestimmungen des Urheberrechtsgesetzes.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Einbandgestaltung: KünkelLopka, Heidelberg

Gedruckt auf säurefreiem Papier

9 8 7 6 5 4 3 2 1

springer.de



Inhaltsverzeichnis

1	Einführung und Motivation	1
	Literatur	9
2	Angriffe.....	11
2.1	Eine Übersicht der Attacken.....	12
2.1.1	Wie lassen sich die Angriffe klassifizieren?	12
2.1.2	Eine Übersicht der häufigsten Attacken auf Applikationen	13
2.1.3	Lokale Angriffsformen	15
2.1.4	Zeitliche Entwicklung der Attacken	15
2.2	Die Attacken im Einzelnen.....	18
2.2.1	Externe Attacken auf Applikationslevel	18
2.2.2	Lokale Attacken auf Applikationslevel.....	30
2.3	Grundlagen der Input-Validierung	38
2.3.1	Abwehr auf der Netzwerkschicht.....	39
2.3.2	Input-/Output-Validierung auf Applikationsebene	48
2.3.3	Forms-Validierung.....	50
2.4	Attacken auf Ebene der Semantik.....	61
2.4.1	Die Kunst der Täuschung.....	61
2.4.2	Phishing und Multi-Faktor-Authentisierung.....	63
2.4.3	Soziale Attacken	67
2.5	WEB2.0-Techniken und ihre Problematik.....	69
2.5.1	Attacken auf Web2.0	70
2.5.2	Die Techniken hinter Web2.0	72
2.5.3	Spezielle Aspekte der AJAX-Security	77
2.6	Zur Psychologie der Verteidigung.....	84
	Literatur	88

3	Grundprinzipien des Designs sicherer Systeme	91
3.1	Economy of Mechanism (KISS)	93
3.2	Fail-Safe-Defaults	94
3.3	Complete Mediation	94
3.4	Open Design: The Design Should Not Be Secret	97
3.5	Separation of Privilege	98
3.6	Principle of Least Privilege (POLP)	99
3.7	Least Common Mechanism	100
3.8	Psychological Acceptability	103
3.9	Zusammenfassung der Prinzipien von Saltzer und Schroeder	104
3.10	Principle of Least Authority – POLA	105
3.10.1	POLA vs. POLP	105
3.10.2	Access Control Lists und Sandboxes	108
3.10.3	Capabilities	111
3.10.4	Zusammenfassung	113
	Literatur	114
4	Platform Security	115
4.1	Platform Security heute	116
4.2	Die „Immutable Laws of Security“: Schadensbegrenzung unmöglich?	121
4.3	Sicherheit und Zuverlässigkeit von Betriebssystemen	126
4.3.1	Klassische Sicherheitsarchitektur in Betriebssystemen	127
4.3.2	Kernel-Architektur: Microkernel vs. monolithischer Kernel	130
4.3.3	Die Sicherheit von Plug-In-Architekturen	134
4.4	Kapselung durch virtuelle Maschinen	139
4.4.1	Sicherheitsumfeld	139
4.4.2	Paravirtualisierte VM-Architekturen	141
4.4.3	Hybride Systeme im Embedded-Control-Bereich	142
4.4.4	Virtualisierung unter Einschluss der Verfügbarkeit: Mainframes	143
4.4.5	Grenzen der Sicherheit durch Virtualisierung	144
4.5	Softwarebasierte Isolation am Beispiel Singularity	145
4.6	Das Least Authority (POLA) Prinzip beim Bau von Systemen	151
4.6.1	Additive und subtraktive Rechtezuweisung	152
4.6.2	Labeling und Tainting	153
4.6.3	Authority Reduction in Vista	154
4.6.4	Privilegienanpassung in Betriebssystemen	157
4.6.5	Privilegienanpassung in Applikationen	159

4.7	Die Sicherheit von Servern.....	160
4.7.1	Grundkonzepte der Serversicherheit.....	161
4.7.2	Sicherheitsrelevante Architekturmerkmale von Servern.....	165
4.7.3	Absicherung von Servern am Beispiel des OKWS-Web-Servers	169
4.8	Administration.....	174
4.8.1	Race Conditions.....	175
4.8.2	SU – verschiedene Implementationen.....	175
4.8.3	Installation	179
	Literatur	180
5	Java Sprach- und Plattformssicherheit.....	183
5.1	Die Evolution der Java-Sicherheit.....	184
5.2	Sprachbasierte Sicherheit	187
5.2.1	Memory Protection und Typsicherheit.....	187
5.2.2	Erweiterungen und Ableitungen von Klassen.....	193
5.3	Access Control Policies und ihre Implementation.....	195
5.3.1	Grundlegende Aspekte von Policies	195
5.3.2	Java Code Access Security	198
5.3.3	Codebasierte Policies.....	203
5.3.4	Userbasierte Policies mit JAAS	207
5.3.5	Alternative Methoden der Zugriffskontrolle.....	217
5.4	Designaspekte der Umstellung auf Java 2 Security.....	218
5.5	Code Security für Enterprise-Applikationen	223
	Literatur	226
6	Enterprise Security.....	229
6.1	Grundlagen von Enterprise Security.....	230
6.1.1	Vom Application Tower zur verteilten Enterprise-Applikation.....	230
6.1.2	Security-Infrastruktur.....	232
6.1.3	Serviceorientierte Architekturen (SOA)	234
6.1.4	Eventgetriebene Systeme und Busse.....	235
6.2	Problematik der Ende-zu-Ende-Sicherheit	239
6.2.1	Speichern von Passwörtern in Registries	240
6.2.2	Nicht-authentisierter Zugriff durch den Suchdienst.....	241
6.2.3	Zugriff durch Patch-Services	244
6.2.4	Propagation von Identität, Recht und Aufruf.....	245
6.3	JAVA EE und EJB	248
6.3.1	Grundprinzipien sicherer Software in JAVA EE und EJB.....	250
6.3.2	JAVA-EE-Applikationsinterfaces und ihre Verwendung.....	251
6.3.3	Rolle, Permission und Ressource.....	253

6.3.4	Delegation und Impersonation	256
6.3.5	Rollenbasierte, deklarative Access Control anhand eines Beispiels	262
6.3.6	Deklarative und programmatische Zugriffskontrolle	272
6.3.7	Annotationen für Sicherheit in Java EE	276
6.3.8	Sicherheitsdiskussion der JAVA-EE-Sicherheit	277
6.4	Struts	278
6.4.1	Authentisierung in Web-Applikationen	278
6.4.2	Spezielle Probleme der Front-End-Applikationen	288
6.4.3	Authentisierungsobjekte	289
6.4.4	Autorisierung	290
6.5	Light-weight Container Security: Das Spring Security (Acegi) Framework	295
6.5.1	Philosophie	295
6.5.2	Grundprinzipien	296
6.5.3	Authentisierung	297
6.5.4	Das Authentication-Objekt	298
6.5.5	Authorisierung	299
6.5.6	Diskussion der Spring Security Security	302
6.6	Grids	303
6.6.1	Grid Security versus Enterprise Security	303
6.6.2	Grid Security Standards	305
6.6.3	Qualifizierte Delegation durch Proxy-Zertifikate	306
6.7	Integration von Conventional-off-the-Shelf (COTS) Applikationen	309
	Literatur	310
7	Application Server Security	313
7.1	Vom Socket-Server zur Container-Architektur	314
7.1.1	Authentisierungs-Frameworks	315
7.1.2	Concurrency und Multithreading	316
7.1.3	Komponentenarchitektur	316
7.1.4	Vom Standalone-Server zum Teil eines verteilten Systems	319
7.1.5	Server-Cluster	319
7.1.6	Anforderungen an die Entwickler	320
7.2	Beispielarchitektur eines Applikationsservers	320
7.2.1	System Context und Security Context	321
7.2.2	Verwaltung von Credentials	322
7.2.3	Formen der Authentisierung	324
7.2.4	Der Classloader	325
7.3	Security-Interfaces zwischen Server und Applikationen	328
7.4	Repräsentation von Identität: Die Erstellung eines Subjects	331
7.4.1	Grundlegende Szenarien	331
7.4.2	Initiale Authentisierung	332

7.4.3	Bereits diesem Application-Server bekannter User	333
7.4.4	Bereits gegenüber einem Application-Server bekannter User	334
7.4.5	Identity Assertion durch vorgelagerte Authentisierung	337
7.4.6	Identity Assertion und Delegation	339
7.4.7	Standard-Interface für Authorization-Provider: JACC ...	341
7.4.8	Erzeugung von Subjects im Application-Server-Verbund	342
7.5	Probleme zwischen JAAS und JAVA EE.....	344
7.6	Absichern von Application-Servern („Hardening“)	347
7.6.1	Auslieferungszustand vs. Default-Is-Deny-Prinzip.....	348
7.6.2	Absicherung der Verbindungen	349
7.6.3	Absicherung der Ressourcen.....	350
7.6.4	Verwaltung des Runtime-Systems	350
7.6.5	Isolation	351
7.6.6	Cross-Cell-Trust.....	353
7.6.7	Code Access Control in Application-Servern	354
7.6.8	„Fully Trusted“ .NET.....	355
7.6.9	Isolierende VMs und Application-Server: SAP	356
7.6.10	User Identity Propagation zu Datenbanken.....	358
7.7	Zusammenfassung	361
	Literatur	362
8	Sichere Multi-Vendor Komponentensysteme.....	363
8.1	Dynamische Services und ihre Isolation in OSGI	364
8.1.1	Anwendungsbeispiele in Heimnetzwerken	365
8.1.2	Namespaces durch Classloader-Architektur	366
8.1.3	Trusted Code und Rechtezuweisung	369
8.2	Sichere Transaktionen mit FINREAD	370
8.2.1	Benutzung	371
8.2.2	Bedrohungsmodell	372
8.2.3	FINREAD-Architektur	374
8.2.4	Trust- und Key-Management	376
8.2.5	Softwareumgebung zur Isolation	380
	Literatur	383
9	Web-Services und objektbasierte Sicherheit.....	385
9.1	Modelle sicherer Kollaboration	386
9.1.1	Fallstudie: E-Voting.....	388
9.1.2	Grenzen der kanalbasierten Sicherheit.....	390
9.2	Objektbasierte Sicherheit.....	390
9.2.1	XML-Signaturen.....	391
9.2.2	XML-Verschlüsselung.....	392
9.2.3	Interoperabilität.....	394

9.3	SOAP Security.....	396
9.3.1	WS-Security.....	396
9.3.2	Security Token Issuer (STI).....	398
9.3.3	WS-Trust-Szenarien.....	401
9.3.4	Autoritäten und Aussagen.....	403
9.3.5	Bedrohungen und Risiken.....	406
9.3.6	Einbindung in Applikationen.....	407
9.3.7	Web-Services über REST	407
9.4	SOA-Security	411
	Literatur	413
10	Sichere Software: Mechanismen und Konstruktionsprinzipien	415
10.1	Rahmenbedingungen sicherer Software	417
10.1.1	Die Macht des Faktischen.....	417
10.1.2	Die Frage der Verantwortung	418
10.2	Grundlegende Mechanismen sicherer Software	419
10.2.1	Filter.....	419
10.2.2	Isolation	420
10.2.3	Hierarchische Modi.....	421
10.2.4	Identitäts- vs. autoritätsbasierte Zugriffskontrolle	423
10.2.5	Kanalbasierte Sicherheit vs. objektbasierte Sicherheit....	425
10.2.6	Capabilities	426
10.2.7	Schlüssel	427
10.2.8	Zugriffskontrolle in lokalen und verteilten Systemen.....	428
10.2.9	Infrastruktur vs. Application Level Security.....	429
10.2.10	Front-End vs. Back-End Security	430
10.2.11	Föderation	431
10.2.12	Basismechanismen der Zugriffskontrolle	431
10.3	Sicherheit und Softwarearchitektur	432
10.3.1	Perspektiven auf Software	432
10.3.2	Zur Bestimmung der Gefährlichkeit	433
10.3.3	Mikro-Architektur.....	434
10.3.4	Klassen, Objekte und Referenzen	436
10.3.5	Funktionaler Ansatz.....	440
10.3.6	Closures	442
10.3.7	Tainting.....	444
10.3.8	Object Capabilities.....	445
10.3.9	Sicherheitsbausteine.....	448
10.3.10	Makro-Architektur	453
10.3.11	Design Patterns für sichere Software	458
10.4	Concurrency	462
10.5	Gegenkräfte	473
10.5.1	Nachträgliches Eindämmen	474
10.5.2	Globale Navigation	474
10.5.3	Globale Namespaces.....	474

10.5.4	Alias-Definitionen	475
10.5.5	Degeneration von Architektur.....	475
10.5.6	Ambige Interfaces.....	476
10.5.7	Paarweise Methoden mit State.....	476
10.5.8	State halten.....	476
10.5.9	Überaggregierung	477
10.6	Sicherheit der Applikation.....	477
10.7	Sind Sicherheit und Nebenläufigkeit „Aspekte“?.....	481
	Literatur	483
11	Fehleranalyse.....	485
11.1	Fehleranalyse als didaktisches Prinzip	485
11.2	Fehleranalyse als theoretisches Prinzip	486
11.3	Datenbasis und Hypothesenbildung	487
11.4	Die Fehler im Einzelnen	489
11.4.1	MFSA 2005-44: Privilege escalation via non-DOM property overrides	489
11.4.2	MFSA 2005-43 „Wrapped“ JavaScript: URLs bypass security checks.....	492
11.4.3	MFSA 2005-33 JavaScript „lambda“ replace exposes memory contents.....	493
11.4.4	MFSA 2005-34 PLUGINSPPAGE privileged javascript execution	494
11.4.5	MFSA 2005-36: Cross-site scripting through global scope pollution.....	495
11.4.6	MFSA 2005-37: Code execution through JavaScript: Favicons	496
11.4.7	MFSA 2005-38 Search plug-in cross-site scripting	497
11.4.8	MFSA 2005-31/39 Arbitrary code execution from Firefox sidebar panel I + II.....	498
11.4.9	MFSA 2005-28: Unsafe /tmp/plugtmp directory exploitable to erase user's files	499
11.4.10	MFSA 2005-25: Image drag and drop executable spoofing	500
11.4.11	MFSA 2005-24: http auth prompt tab spoofing.....	501
11.5	Klassifikation der Schwachstellen im Mozilla-Firefox-Browser...	502
11.5.1	Übersicht.....	503
11.5.2	Fehlermuster	503
11.6	Browserarchitektur und Sicherheitskonzepte	504
11.6.1	Was ist ein Browser?	504
11.6.2	Gibt es ungefährlichen Input?	505
11.6.3	Symbolische Referenzen.....	507
11.6.4	Hobbles und Patches – JavaScript im Netscape-Browser.....	507
11.6.5	Sicherheitskonzept des Browsers.....	510

11.6.6	Data-Tainting.....	514
11.6.7	Eine alternative Architektur.....	516
	Literatur.....	517
12	Browsersicherheit durch Object Capabilities	519
12.1	Beschreibung des Experiments.....	520
12.2	Architektur, Design und Implementation der Lösung	522
12.2.1	Makro-Architektur	522
12.2.2	Komponenten und Applikation.....	523
12.2.3	Taming und Capability Architecture.....	526
12.2.4	HTML.....	527
12.3	Sicherheitsanalyse	528
	Literatur	531
13	Security und Usability	533
13.1	Zum Verhältnis von Usability und Security	534
13.2	Lokale vs. fremdvermittelte Sicherheit.....	536
13.2.1	Praktisches Key-Management.....	537
13.2.2	Namen und ihre eindeutige Verwendung.....	540
13.2.3	Rechte und Eigentum.....	543
13.2.4	Discretionary vs. Mandatory – über Eigentum und Verfügung	544
13.2.5	Bequemlichkeit vs. Sicherheit – der richtige Gegensatz?.....	545
13.2.6	Erkennen und Ausführen von Intentionen	547
13.2.7	Psychologische Voraussetzungen der User-Interaktion	548
13.2.8	Messen von Intention – Verständnisprüfung	554
13.2.9	Autorität aus Designation	556
13.2.10	Sichere User-Interfaces	557
13.2.11	Herstellen eines Trusted Path.....	558
13.2.12	Web2.0 und die Frage der Nutzerintention	560
	Literatur	561
14	Bestimmung der Sicherheit durch formale Ansätze	563
14.1	Zur Frage der Entscheidbarkeit von Sicherheit	564
14.1.1	Die Access Control Matrix	564
14.1.2	Capabilities modelliert durch Take/Grant-Systeme	568
14.1.3	Differenzierung des Take/Grant-Ansatzes	574
14.2	Der Scoll-Ansatz in der Sicherheitsanalyse (Fred Spiessens)	580
14.2.1	Das Confused-Deputy-Problem	580
14.2.2	Kernkonzepte der Sicherheitsanalyse	582
14.2.3	KBM: Konservative Modelle für Systeme interagierender Entitäten.....	586

14.2.4	Scoll: Eine Sprache, um Sicherheitsprobleme auszudrücken	592
14.2.5	Die Analyse des Confused Deputy mit Scollar	596
14.3	Prüfung Operationaler Umgebungen	602
14.3.1	Modellierung und Generierung	605
14.3.2	Model-Checking von Sicherheitsprotokollen	607
14.3.3	Modell-getriebenes Security-Engineering	607
14.3.4	Lernen und Testen	609
14.3.5	Beweisbare Plattformensicherheit	609
	Literatur	613
15	Schlussbetrachtungen	615
15.1	Security vs. Safety	615
15.2	Security vs. Usability	616
15.3	Mehr Sicherheit durch Einschränkung der Nutzer?	618
15.4	Prozessorientierte Security	618
15.5	Modellbasierte, integrierte Security	619
15.6	Zuverlässigkeit und Verfügbarkeit	620
	Literatur	621
	Index	623

Kapitel 1

Einführung und Motivation

Warum wird so viel unsichere Software entwickelt? In dem Buch „Internet-Security aus Software-Sicht – Grundlagen“ [KS] haben wir vor allem die Mängel in der Wahrnehmung von Sicherheitsproblemen betont und versucht, diese Wahrnehmung anhand von Fallbeispielen und Sicherheitsanalysen zu stärken. Gleichzeitig wurden dort die sicherheitstechnischen Grundbausteine, Protokolle und Dienste in einer solchen Form vorgestellt, dass sie für praktisch tätige Entwickler, Architekten und Infrastrukturspezialisten verständlich und nutzbar sind.

Mit der verbesserten Wahrnehmung von Sicherheitsproblemen und Grundkenntnissen in Kryptografie allein ist es jedoch nicht getan. Wie sichert man eigentlich konkret eine Server-Applikation für das Internet ab? Gemeint sind hier nicht mehr Infrastrukturmaßnahmen, welche die Angriffsfläche innerhalb der DMZ verkleinern, sondern die Serversoftware selbst und die Plattform, auf der sie läuft. Welche Strategien und Mechanismen müssen wir einsetzen, damit unsere Software die Businessmodelle erfüllt, sodass nur autorisierte Personen Zugriff erhalten? Das Wissen über die Strategien ist in den Köpfen erfahrener Systemarchitekten enthalten, oder in den Vorschriften zur Softwareentwicklung mancher Konzerne und nicht zuletzt in einzelnen wissenschaftlichen Papieren z. B. über den Bau sicherer Web-Server. Eine Aufgabe dieses Buches ist es, dieses Wissen zusammenzutragen, zu diskutieren, die jeweiligen Grenzen der Strategien aufzuzeigen und somit für die Entwickler nutzbar zu machen.

Heute entsteht Software meist auf der Basis von Frameworks, die bereits Sicherheitstechniken z. B. für die Autorisierung von Zugriffen eingebaut haben und deren Verständnis essenziell für den Bau sicherer Systeme ist. Dies betrifft Applikationsentwickler wie Systemingenieure. Die letzteren verwenden ebenfalls Frameworks, um Applikationen mit der jeweiligen Firmeninfrastruktur zu verbinden. Somit entstehen diffizile Abhängigkeiten zwischen Infrastruktur und Applikationen. Dieser Band hat daher als einen weiteren Schwerpunkt, die Absicherung von Enterprise-Software durch Frameworks wie z. B. EJB, JAAS etc. zu erklären.

Das Buch bleibt aber nicht bei der Betrachtung einzelner Teile von Sicherheitstechniken stehen, sondern versucht das Gesamtsystem zu betrachten. Ein Gesamt-

system besteht aus der Kombination von Betriebssystem, Programmiersprache, Applikationsarchitektur und nicht zuletzt den beteiligten Personen. Zwischen diesen Komponenten bestehen weitere Abhängigkeiten, und die Sicherheit der einzelnen Komponenten garantiert noch längst nicht ein sicheres Gesamtsystem. Leider sorgt jedoch umgekehrt eine einzige unsichere Komponente meist für die Unsicherheit des Gesamtsystems.

Der Einbezug der beteiligten Personen in das Sicherheitskonzept des Gesamtsystems mag manche überraschen, vor allem wenn deutlich wird, dass dies auch die beteiligten Entwickler, Administratoren etc. umfasst. Oft wird unter psychologischen Aspekten der Sicherheit lediglich auf die potenziellen Angreifer geschaut, und beim Anteil der Usability an einem sicheren System lediglich auf den Benutzer. Dabei haben Unverständnis, Druck sowie schlechte Werkzeuge auf Seiten der Entwickler und Administratoren ebenfalls einen entscheidenden Anteil an unsicheren Systemen. Nicht zu vergessen alte Programmiermythen wie die Mär vom Ausgleich technischer Defizite in Programmiersprachen durch „Disziplin“. Wir müssen begreifen, dass gewisse Defizite immer wieder zu Sicherheitsproblemen führen werden. Täglich bestätigen z. B. neue Buffer-Overflow-Attacken diese Erkenntnis.

Sehr interessant sind in diesem Zusammenhang auch die Untersuchungen zur Risikowahrnehmung und ihrem Einfluss auf die IT-Sicherheit, wie sie von Volker Scheidemann [Schei] durchgeführt wurden. Die ungleich bessere Akzeptanz von Virenscannern im Vergleich zu Verschlüsselungsprodukten kann demnach nicht nur auf Probleme der Usability bei letzteren zurückgeführt werden. Es scheint grundsätzliche, psychologisch motivierte Defizite bei der Risikoeinschätzung zu geben. Wir werden im Kapitel zu den Attacken sowie zu Usability und Security daher versuchen, den Ansatz von Scheidemann im Bereich der Softwareentwicklung anzuwenden. Sollte sich auch hier der Einfluss von falscher Risikowahrnehmung zeigen, wäre ein weiterer Grund für unsichere Software (bzw. deren lange Lebensdauer) gefunden und der Stärkung der Wahrnehmung innerhalb der Usability müsste noch mehr Gewicht eingeräumt werden.

Aber die verkürzte Sichtweise auf ein Gesamtkonzept von Sicherheit hat natürlich eine alte Tradition, die erst mühsam überwunden werden muss. So erweitert z. B. das Konzept der „multi-lateralen“ bzw. „mehrseitigen“ Sicherheit nach Ranneberg et al. [Rann] die Sicherheit einer Geschäftslösung, indem es nicht nur nach der Sicherheit der anbietenden Firma fragt, sondern auch nach der Sicherheit des Konsumenten. Dadurch entstehen ganz neue Forderungen z. B. nach dem Schutz der Privatsphäre oder dem Recht auf Anonymität.

Bei den bisher angesprochenen Problemen befinden wir uns im „klassischen“ Bereich der IT-Security, verstanden als Einsatz von Sicherheitstechnik, um die Einhaltung von Businessmodellen auf der Basis von Authentisierung, Autorisierung und Nachweisbarkeit zu gewährleisten, angereichert mit dem Wissen um die Problematik der Benutzbarkeit und die Wissensdefizite bei den Beteiligten. Das vorliegende Buch versucht jedoch noch einen Schritt weiter zu gehen, indem es einige unbequeme Fragen stellt:

Wieso versagt Software im Zusammenhang mit Sicherheitsproblemen häufig vollständig? Wieso scheint es bei Softwaresystemen keine Schadensbegrenzung zu geben, wie wir sie aus anderen Bereichen wie z. B. dem Automobilbau kennen? Dort bewirkt der Ausfall einer Komponente meist nicht mehr den völligen Verlust der Sicherheit. Mit diesen Fragen verlassen wir die typische Angriffsperspektive, aus der Sicherheit in erster Linie als Schutz vor Angriffen auf die Businessmodelle interpretiert wird. Nun geht es auch um grundsätzlichere Fragen der Zuverlässigkeit und Berechenbarkeit von Software unter dem Gesichtspunkt der „Safety“, also dem sicheren Funktionieren von Software. Eine zentrale These dieses Buches ist, dass es letztlich Defizite der „Safety“ – also grundlegende Mängel in der Art, wie wir unsere Software konstruieren – sind, die dann auch gravierende Sicherheitslöcher nach sich ziehen. Es sind also nach Ansicht der Autoren nicht nur Denkfehler bei der Verwendung von Security-Protokollen, die Sicherheitsprobleme verursachen. Diese gibt es zweifellos und wir werden versuchen, sie nach Kräften zu erklären und zu beseitigen. Aber dieser andere Bereich – hier „Safety“ genannt, wird mit der völligen Computerisierung unserer Lebenswelt noch wesentlich wichtiger. Hier stößt die bloße Wahrnehmung von Angriffsmöglichkeiten und ihrer Abwehrmaßnahmen als Werkzeug zur Entwicklung sicherer Software an ihre Grenze und ein Konzept von Schadensbegrenzung wird nötig. Damit verbunden sind auch Konzepte der Schadensabschätzung bzw. der Beweisbarkeit von Schadensgrenzen.

Abbildung 1.1 versucht, das Verhältnis der Dimensionen Security und Safety innerhalb von Softwaresystemen abzubilden:



Abb. 1.1 Sichere Systeme als Kombination der Aspekte Security und Safety

Deshalb arbeitet dieser Band anhand der diskutierten Software-Frameworks die jeweiligen Mechanismen der Zugriffskontrolle (Zugriff hier verstanden als kausale Möglichkeit, Effekte zu erzielen) sowie deren Defizite heraus. Wir sehen uns also die Konzepte und Techniken im Einsatz an und untersuchen anschließend ihre Eigenschaften. Als Leitprinzip werden wir an jeder Stelle, an der ein Zugriff kontrolliert wird, die Frage nach der tatsächlich vorhandenen Autorität stellen (existieren zu viele Rechte?). Und wir werden die Zugriffskontrolle mit der Intention der Beteiligten vergleichen: Bewirkt die Kontrolle tatsächlich die Sicherheit, die gemeint ist?

Warum scheinen bestimmte Sicherheitsmängel innerhalb von Applikationen oder Systemen immer wiederzukehren? Zur Beantwortung dieser Frage untersucht das Buch bestimmte Fehlertypen und führt sie auf architektonische Ursachen in Systemen, Sprachen oder Applikationen zurück. Zu einer ganzheitlichen Betrachtung von sicheren Systemen gehört aber zwangsläufig auch die Betonung von „Sicherheit, die funktioniert“ – ein Begriff, den Simson Garfinkel in seiner Studie zum Zusammenspiel von Usability und Security ([Garf]) geprägt hat: Die Anwendung von Sicherheitsprinzipien und ihre Implementation in realen Systemen mit realen Benutzern in realen Situationen. Wir werden im Laufe des Buches des Öfteren Sicherheitsmechanismen begegnen, die in der „wirklichen Welt“ einfach nicht funktionieren. Sei es, weil die User-Interfaces unverständlich oder mühsam zu bedienen sind, sei es, weil Implementationen beim Einschalten des „sicheren Modus“ leider unerträglich langsam werden. Aus diesem Grund laufen .NET-Server meist „fully trusted“ und bei Java Enterprise Edition (JAVA EE) basierten Servern wird ebenfalls meist empfohlen, die Code-Based-Security auszuschalten. Oder sei es, weil die Modellierung der Konfiguration von Sicherheitskriterien so schlecht ist, dass selbst die Entwickler Fehler machen.

Nun zum Aufbau des Buches: Wir beginnen mit einer Untersuchung gängiger Attacken auf Plattformen und Applikationen sowie der Abwehrmaßnahmen. Zentrale Aussagen dieses Teils sind die Notwendigkeit formaler Beschreibung von Input und Output von Applikationen sowie die Reduktion der Applikationen und Modulen zur Verfügung stehenden Autorität. Außerdem wird die semantische Attacke als zukünftig dominierende Form dargestellt. Gedanken zur Sicherheit von Web2.0-Applikationen beenden das Kapitel zu Attacken. Am Beispiel von Web Application Firewalls untersuchen wir, was beim Auftreten von Sicherheitslücken in Legacy-Applikationen schnell getan werden kann. Ziel der Darstellung der Attacken ist jedoch in erster Linie, auf Grundsatzfragen hinzuweisen, z. B. warum eine Schwäche oder Lücke eigentlich so fatal für die Sicherheit des Gesamtsystems ist. Dazu werden sowohl lokale Angriffe als auch solche über das Netzwerk vorgestellt.

Im Anschluss an die Behandlung von Angriffen werden die Grundkonzepte, die hinter sicherer Software stehen, anhand einer Diskussion der klassischen Thesen von Salzer und Schroeder [SaSch] eingeführt. Wir gehen bei unserer Diskussion davon aus, dass die Plattformensicherheit und mit ihr das sogenannte Host-Bedrohungsmodell in Zukunft stark an Bedeutung gewinnen wird. Zwei entscheidende Begriffe werden in diesem Kapitel eingeführt: Das „Principle Of Least

Authority (POLA) als zentrale Maßnahme der Schadensbegrenzung und das Gegenteil davon: „Ambient Authority“ (s. z. B. [Wiki]) als frei schwebende, jederzeit verfügbare Möglichkeit, Schaden anzurichten. Alle bekannten Konzepte und Techniken zur Absicherung von Zugriffen (Access Control Lists, Sandboxes, Isolation durch Modi etc.) werden wir daraufhin untersuchen, inwieweit sie POLA beachten bzw. wieviel Ambient Authority zugelassen wird. Dabei werden wir das Prinzip der „Autorität“ im Sinne einer kausalen Möglichkeit, einen Effekt zu erzielen, vom Prinzip des „Rechts“ abgrenzen, hier verstanden als grundsätzliche Erlaubnis, etwas zu bewirken, was jedoch nicht automatisch die Möglichkeit dazu einschließt.

In einem speziellen Kapitel wird anschließend die Sicherheit von Plattformen (Betriebssystemen) und ihre Technik als Voraussetzung für die Sicherheit von Softwaresystemen behandelt. Wie sicher sind unsere Plattformen? Welche Mittel der Zugriffskontrolle verwenden sie? Welche Hardware-/Softwaremechanismen bewirken Sicherheit?

Neben Betriebssystemen stellen Sprachen einen wichtigen Teil der Sicherheit einer Software dar. Im Rahmen der Plattformsicherheit wird das Singularity-Betriebssystem von Microsoft kurz vorgestellt, da es eine rein auf Software basierende Isolationsstrategie verfolgt. Die Mächtigkeit von kontrollierten Referenzen auf Objekte zeigt sich daran, dass keine Hardware zur Virtualisierung nötig ist und auf grobe Isolationskonzepte wie Modi verzichtet werden kann. Die richtige Form von Softwareinstallation wird in diesem Zusammenhang ebenfalls vorgestellt.

Die Absicherung von Server-Applikationen wie Web-Servern etc. ist nach wie vor ein großes Problem. Das entsprechende Kapitel zeigt gängige Isolationstechniken sowie die Einengung des Zugriffs auf Backend-Systeme und die Problematik mächtiger, funktionaler Zugriffsidentitäten (Accounts). Im Kapitel zu Application-Servern wird auf diese Erkenntnisse aufgebaut.

Im Anschluss an die Plattformsicherheit werden sicherheitsrelevante Aspekte von Programmiersprachen unter spezieller Berücksichtigung von Java untersucht. Auf die gängigen Frameworks wie JAAS wird dabei natürlich eingegangen, ebenso auf die Code Access Security, wie sie durch die Java 2 Security gegeben ist. Eine wesentliche Aussage dieses Kapitels ist, dass die Code Access Security keinesfalls so unabhängig von der Architektur der Applikation ist, wie es zunächst bei Ansicht der externen Konfigurationsmöglichkeiten aussieht. Es wird gezeigt, dass hier bei Beginn der Applikationsarchitektur verschiedene Sicherheitsebenen eingeführt werden müssen, um zu verhindern, dass Applikationen über Systeminterne Bescheid wissen müssen bzw. zu viel Autorität erhalten. Der Stack-Walk-Mechanismus wird ebenfalls vorgestellt und bewertet. Interessanterweise existieren neben der ACL-basierten Zugriffsmechanik in Java durchaus weitere Sicherheitsmechanismen wie Namespaces und Object-Capabilities. Diese spielen in später behandelten Frameworks dann eine zentrale Rolle.

Anschließend an die sprachbasierte Sicherheit tritt das Thema Enterprise-Security ganz in den Vordergrund. Konzepte und Techniken zur Absicherung ganzer Infrastrukturen und Applikationen werden vorgestellt und kritisch hinterfragt. Das Verhältnis von Infrastruktur und Applikation in Bezug auf Sicherheit

wird untersucht und am Beispiel von JAVA-EE-Mechanismen nachvollzogen. Entscheidende Erkenntnisse sind dabei, dass Sicherheitscode zwar erfolgreich aus den Applikationen verbannt werden konnte, dass dadurch aber die Rolle der Infrastruktur extrem wichtig wird und Anpassungen der Infrastruktur nun ähnlich kritisch zu sehen sind, wie Änderungen an den Applikationen, etwa aus Gründen der Reorganisation. Zudem stellt die Delegation von Requests über mehrere Stufen einer Infrastruktur hinweg nach wie vor ein großes Problem dar.

Das Problem der Delegation wird anhand der sogenannten Proxy-Zertifikate aus dem Grid-Computing-Bereich erneut aufgegriffen. Dabei diskutieren wir die Frage, ob durch vermehrten Einsatz von objektbasierten Sicherheitsmechanismen (Token, Zertifikate) die Infrastruktur wieder flexibler gestaltet werden könnte, indem eine semantische Trennung zwischen Business und Infrastruktur eingeführt wird.

Ein weiteres Ergebnis des Kapitels zur Enterprise-Security ist die Erkenntnis, welche zentrale Rolle der Applikation-Server im Gesamtkonzept sicherer Applikationen spielt. Dieser wird im Anschluss an die Enterprise-Security behandelt. Wir diskutieren unter anderem die wichtigen Interfaces zwischen Application-Server und Firmeninfrastruktur. Der Trend bezüglich sicherer Delegation von Aufrufen geht dabei klar in Richtung von tokenbasierten Mechanismen, mit denen Erzeuger und Transporteur von Requests klar unterschieden und authentisiert werden können. Auch auf spezielle Probleme der Application-Server wird dabei eingegangen, z. B. im Zusammenhang mit dem „Härten“ von Application-Servern.

Die Sicherheitsproblematik von Embedded-Control-Plattformen wird in Zukunft kaum zu überschätzen sein – getrieben durch die Ausbreitung der Rechner in sämtliche Lebenswelten. Der Zwang zur günstigen Produktion steht dabei oft in einem Spannungsverhältnis zur Isolation fremder Softwarekomponenten aus Sicherheitsgründen. Dies ist z. B. dann ein Thema, wenn die Anzahl der Steuergeräte in Fahrzeugen reduziert werden soll. In einem Kapitel zu Multi-Vendor-Komponentensystemen werden am Beispiel von OSGi und FINREAD zwei Architekturen zur Bereitstellung sicherer Services vorgestellt. Die dazu nötigen Isolationstechniken auf Basis virtueller Maschinen sowie getrennte Namespaces durch Class-Loader werden untersucht und bewertet. Das Konzept der signaturbasierten Softwaredistribution wird ebenfalls vorgestellt und kritisch hinterfragt.

Neben der klassischen End-to-End-Infrastruktur moderner Firmenapplikationen, die vom Browser des Kunden über die Midrange-Server bis zum Mainframe reicht, beinhaltet dieses Buch auch die Grundlagen föderativer Sicherheit auf Basis von Web-Services sowie die sicherheitstechnischen Grundlagen virtueller Organisationen am Beispiel von Grid Computing.

Web-Services dienen uns als Beispiele objektbasierter Sicherheit auf der Basis von signierten Nachrichten (Messages), und sie zeigen einen Weg aus der Abhängigkeit von der Infrastruktur. Da die Sicherheitsinfrastrukturen verschiedener Firmen ebenfalls sehr verschieden sein können, kann Sicherheit nicht mehr als reines Vertrauensverhältnis innerhalb einer Infrastruktur implementiert werden. Sicherheit muss daher auf höheren Ebenen – eben in den Nachrichten und Dokumenten – angesiedelt werden. In diesem Kontext wird auch die Notwendigkeit von

Sicherheitsabstraktionen für Applikationsentwickler gezeigt, die erst die Techniken objektbasierter Sicherheit einsetzbar machen. Grids sind von diesem Standpunkt aus lediglich als extreme Form von Föderation zu sehen. Das Konzept der virtuellen Organisation mit ihren Sicherheitsproblemen wird kurz dargestellt. Einige besonders mächtige Mechanismen aus der Grid-Security werden ebenfalls vorgestellt. Hier ist die zentrale These, dass Grids aufgrund ihrer extremen Sicherheitsproblematiken (wohl nur noch übertroffen von Peer-to-Peer-Netzwerken) zur Entwicklung neuartiger Sicherheitstechniken geradezu gezwungen sind.

Dieser Band behandelt eine Vielzahl von aktuellen Technologien im Zusammenhang mit sicherer Software. Im Vordergrund steht dabei sicher die Vermittlung von Kenntnissen zu aktuellen Frameworks und Techniken. Dennoch wird dabei auch versucht, die in den Plattformen, Sprachen und Frameworks eingesetzten grundlegenden Sicherheitsmechanismen und ihre Funktionsweisen herauszuarbeiten. Dabei stellt sich heraus, dass es nur eine kleine Zahl grundsätzlicher Techniken ist, auf denen die Sicherheit zumeist basiert, nämlich:

- Isolierte Räume und Vertrauenszonen,
- Access-Control-Listen und Referenzmonitore,
- Call-Verfolgung,
- Multi-Level-Security (Labelling, Tagging, Tainting),
- Namespace-basierte Isolation,
- Object Capabilities,
- kryptografische Methoden,
- modusbasierte Security und
- authentifizierte Sessions.

Interessanter als die bloßen Mechanismen sind die Dimensionen, in denen sie sich unterscheiden: So funktionieren einige unabhängig von der Softwarearchitektur, während andere fester Bestandteil der Architektur werden. Entsprechend lässt sich die Zugriffskontrolle bei einigen dynamisch und schnell ändern, während andere Änderungen an der Software bedingen. Einige Mechanismen stellen Rechte permanent zur Verfügung, andere lassen Rechte in Abhängigkeit vom Bedarf fließen. Einige basieren auf Erlaubnissen in Form von Regeln, andere arbeiten kausal auf der Basis von Zugriffsmöglichkeiten über Referenzen oder signierten Vollmachten. Einige hängen an den beteiligten Daten, während andere auf der Codebasis operieren. Schließlich funktionieren einige eher lokal innerhalb von Vertrauenszonen, während andere in verteilten Systemen mit sich gegenseitig misstrauenden Partnern eingesetzt werden können.

Im letzten Teil des Buchs steht die theoretische Analyse der Sicherheitsprobleme von Softwaresystemen im Blickpunkt. In einem Grundlagenkapitel zur Konstruktion sicherer Software wird eine sehr wichtige These vertreten: Sichere Software benötigt Softwarebausteine, die nachweisbar und korrekt bestimmte Sicherheitskonzepte implementieren und dadurch eine Reduktion von Autorität vornehmen. Sichere Software ist daher nicht nur eine Frage von korrekten Sicherheits-Policies in Form von ACLs, sondern eine Frage der Softwarearchitektur auf unterschiedlichsten Ebenen der Granularität (einzelne Statements, Module,

Architekturmuster). Sicherheit ist demnach kein „Aspekt“, der beliebig externalisiert werden kann, sondern eine Kerneigenschaft, die in der Architektur der Software angesiedelt ist. In diesem Teil des Buches kommt Sicherheit sozusagen „nach Hause“ zur Software und wird als Softwareproblem erkannt – jenseits aller Kryptografie oder Protokolle. Man kann diese Erkenntnis noch radikaler formulieren: Unsichere Software ist ein generelles Qualitätsproblem der Softwareentwicklung und wird nicht nur durch schwierig zu verstehende Kryptografie oder fehlerhaft implementierte Sicherheitsprotokolle verursacht.

Anschließend wird eine konkrete Sicherheitsanalyse – hier von Mozilla/Firefox – vorgestellt. Innerhalb dieser Analyse werden die gemeldeten Sicherheitsprobleme untersucht, kategorisiert und anschließend auf grundlegende Probleme der Plattform, der Sprache oder der Applikationsarchitektur zurückgeführt. Am Ende steht die Erkenntnis, dass die Sicherheit komplexer Applikationen wie z. B. Browsern momentan außer Kontrolle ist und nur durch grundsätzliche Änderungen der Architektur sowie der Plattformsicherheit verbessert werden kann. Das Kapitel hat zudem den didaktischen Zweck, die bisher aufgebaute Wahrnehmung von Sicherheitsproblemen und deren Ursachen zu testen und bietet darüber hinaus reichlich Hypothesen für empirische Untersuchungen von Sicherheitsfehlern.

Daran anschließend wird ein alternatives Browsermodell basierend auf dem Konzept von Object-Capabilities vorgestellt, der sogenannte DARPA-Browser. Dieser Browser wurde explizit unter Berücksichtigung von Least-Authority-Prinzipien und einer sicheren Sprache entwickelt und dient als Forschungsinstrument für die Möglichkeiten und Grenzen solcher Architekturen. Zentrale Erkenntnis ist hier, dass es tatsächlich möglich ist, eine Architektur zu entwickeln, bei der bösartige Komponenten nur minimalen Schaden anrichten können. Außerdem wird klar, dass es innerhalb von Applikationen weitere Bereiche der Isolation geben muss.

Aus den obigen Analysen ergeben sich fast automatisch auch Fragen nach der Usability: Ist es wirklich so, dass beide in einem fast unversöhnlichen Gegensatz stehen, dass also mehr Security zwangsläufig zu einer schlechteren Bedienbarkeit führt? Welche Rolle spielt die System- bzw. Applikationsarchitektur dabei? Gibt es eine Alternative zu der momentan üblichen Masse an Sicherheitsdialogen mit unverständlichen Meldungen? Die zentrale Aussage dieses Kapitels ist die Behauptung, dass ohne eine massive Reduktion von Autorität kein wirklicher Gewinn an Sicherheit durch Verbesserungen der Benutzbarkeit möglich ist.

Der dritte Teil schließt mit der Betrachtung neuerer formaler Ansätze in der Sicherheitsanalyse sowie bei der Entwicklung von Sicherheitskomponenten als Softwarebausteinen. Das Konzept der Erreichbarkeit als Basis für Sicherheitsanalyse und -design wird anhand von Capabilities und erweiterten Take/Grant-Systemen vertieft. Wir reflektieren hier die Ansätze von Miller und Shapiro (s. [MS]) sowie Spiessens (s. [Spie07]) und ihren Versuch, durch Object-Capabilities eine Schadensbegrenzung innerhalb der Software einzuführen. Zentrale These dieses Kapitels ist die Behauptung, dass ohne die Entwicklung von Softwarekomponenten/-bausteinen zur Reduktion von Autorität weder die Entwicklung sicherer Software noch deren Analyse möglich ist. Diese Bausteine verwirklichen prüfbare Pattern

für Kollaboration (s. [Spie07]). Das Kapitel stellt außerdem Ansätze zur Modellierung von Sicherheitsaspekten in Infrastrukturen vor. Am Beispiel des „confused deputy“ stellt Fred Spiessens die Sprache Scoll und den Ansatz der automatischen Überprüfung von Sicherheitsmodellen (modell checking) vor.

Dieser Band richtet sich verstärkt an Softwareentwickler, z. B. in Enterprise-Umgebungen. IT-Security-Spezialisten, die traditionell eher im Bereich der Netzwerksicherheit zu Hause sind, erfahren durch dieses Buch eine Vielzahl von sicherheitsrelevanten Fehlerquellen und Ursachen im Bereich der Softwareentwicklung und können daher ihre Abwehrmaßnahmen besser planen. Aber auch der mehr theoretisch interessierte Leser wird im Bereich der capabilitybasierten Zugriffskontrolle und ihrer Modellierung hoffentlich wichtige Anregungen finden. In seiner fundamentalen Ausrichtung schuldet der Band sehr viel der sicheren Programmiersprache „E“ [Erighs] sowie den Diskussionen um die Verwendung von Capabilities als Mittel der Schadensreduktion auf der Cap-talk-Mailing-Liste [Cap].

Wie für den ersten Band finden sich aktuelle Arbeiten, Vorlesungsunterlagen und Diskussionen sowie Fehlerkorrekturen auf <http://www.kriha.de/krihaorg/sicheresysteme.html>.

Literatur

- [Cap] Cap-talk mailing list, <http://www.eros-os.org/pipermail/cap-talk/>
- [Erighs] Erighs.org, Homepage der sicheren Sprache „E“
- [Garf] S. Garfinkel, Design Principles and Patterns for Computer Systems That Are Simultaneously Secure and Usable, Dissertation, Massachusetts Institute of Technology (MIT) 2005, available online at <http://www.simson.net/thesis/>
- [KS] W. Kriha, R. Schmitz, Internet-Security aus Software-Sicht – Grundlagen, Springer-Verlag 2008
- [MS] M.S. Miller, J.S. Shapiro, Paradigm Regained: Abstraction Mechanisms for Access Control, Proceedings of 8th Eighth Asian Computing Science Conference (ASIAN’03), edited by Vijay Saraswat, Springer-Verlag 2003 (available online at <http://www.erighs.org/talks/asian03/>)
- [Rann] K. Rannenberg, A. Pfitzmann, G. Müller, Sicherheit – insbesondere mehrseitige IT-Sicherheit, <http://www.wiwi.de/publikationen/Sicherheit/insbesonderemehrsei.pdf>
- [SaSch] J.H. Saltzer, M.D. Schroeder, The Protection of Information in Computer Systems, available online at <http://www.cs.virginia.edu/~evans/cs551/saltzer/>
- [Scheid] V. Scheidemann, It won’t happen to me – Aspekte der Risikowahrnehmung und ihr Einfluss auf die IT-Sicherheit, in: Innovationsmotor IT-Sicherheit, Tagungsband zum 10. Deutschen IT-Sicherheitskongress, BSI 2007
- [Spie07] F. Spiessens, Patterns of Safe Collaboration, Ph.D. dissertation, Department of Computing Science and Engineering, Université Catholique de Louvain, Feb. 2007, http://www.info.ucl.ac.be/~pvr/fsp_thesis.pdf
- [Wiki] Ambient Authority, Wikipedia http://en.wikipedia.org/wiki/Ambient_authority

Kapitel 2

Angriffe

Glücklicherweise gibt es seit einiger Zeit ausgezeichnete Hilfen gegen das komplexe Problem der web-basierten Attacken. So behandelt z.B. Sverre Huseby [Hus] das Thema Input-Validierung für Web-Applikationen ausführlich. Auch der Guide von OWASP.org zur Entwicklung sicherer Web-Applikationen deckt die möglichen Angriffe sehr gut ab. Deshalb wollen wir hier nur einige wesentliche Angriffsformen behandeln, im Übrigen aber mehr Wert auf die Einordnung der Attacken in ein generelles Sicherheitskonzept legen. Außerdem wollen wir einige theoretische Überlegungen zur grundsätzlichen Bekämpfung dieser Attacken anstellen.

Durch das Phänomen des „Ubiquitous Computing“, der vollständigen Durchdringung des Alltags und Lebensraums durch Computer, bekommt neuerdings auch die lokale Sicherheit der Geräte eine immer größere Bedeutung. So ist die Installation neuer Software zwar heutzutage meist über einen Netzwerkanschluss durchzuführen, aber die Frage, wie sich die Installation auf die lokale Sicherheit des Hosts¹ auswirkt, hat zwar nichts mehr mit „Web Security“ im engeren Sinne zu tun, ist aber für unsere Diskussion dennoch von Interesse, da es sich bei diesen Hosts um Kundenrechner, also Teile des Gesamtsystems, handelt. Wir wollen daher auch lokale Angriffe besprechen.

Erreichen wollen wir durch diese Diskussion zum einen natürlich eine weitere Sensibilisierung für Web-basierte Angriffsmöglichkeiten. Zum anderen möchten wir einen theoretischen Rahmen für die Gefährlichkeit der Attacken aufspannen und schließlich auch die möglichen Abwehrmaßnahmen aufzeigen, die vom Bereich Netzwerk bis hin zur Softwarearchitektur, je nach Angriffsform, ganz unterschiedliche Formen annehmen können.

¹ „Host“ wird synonym zu „Rechner“ und „Node“ verwendet.

2.1 Eine Übersicht der Attacken

2.1.1 *Wie lassen sich die Angriffe klassifizieren?*

Eine erste Möglichkeit zur Klassifikation der Angriffe ist die „Schicht“ oder „Ebene“, in der sie auftreten: Wir unterscheiden Angriffe auf das Netzwerk, Angriffe, die auf fehlende oder fehlerhafte Input-Validierung durch einen Web-Server abzielen oder aber konzeptionelle Angriffe, die die Kommunikationssession zwischen Client und Server angreifen.

Die Angriffe auf der Netzwerkebene sind häufig auf die Verfügbarkeit der Server-Applikation gegenüber dem Internet ausgerichtet, können aber auch die Verfügbarkeit des internen Netzwerks betreffen. Die Möglichkeiten, diese Angriffe durch reine Softwaretechniken abzuwehren, sind eher begrenzt. So sind Sites wie grc.com oder unlängst heise.de erfolgreich mit dem Effekt angegriffen worden, dass sie längere Zeit nicht zur Verfügung standen. Die Abwehrmaßnahmen beinhalten die Kontaktaufnahme zum ISP mit dem Ziel, die Netzwerkstränge, aus denen die Angriffe erfolgen, bereits vor dem Erreichen des eigenen Netzwerks zu blockieren. Dies hat zur Folge, dass legale User aus diesen Bereichen den Server ebenfalls nicht erreichen können. In diesem Fall hilft es einem ausgeschlossenen Kunden weiter, wenn er einen anonymisierenden Proxy verwendet, der sich in einem anderen IP-Adresssegment befindet.

Die weitaus größte Vielfalt an Angriffsmöglichkeiten nutzt hingegen Schwächen in der Validierung von Input. Darunter fallen Buffer-Overflow-Attacken, aber auch client- und serverseitige Cross-Site-Scripting-Attacken. Die Struktur dieser Angriffe ist eigentlich sehr einfach, ihre Vermeidung setzt jedoch ein klares Konzept in Softwaretechnik und Organisation der Entwicklung voraus. Diese Angriffsform kann extern über das Netzwerk, aber auch lokal erfolgen.

Die dritte Gruppe von Angriffen zeichnet sich dadurch aus, dass keinerlei netzwerktechnische oder softwaretechnische Fehler vorliegen. Das Gesamtsystem funktioniert wie erwartet, die Attacke läuft hingegen auf konzeptioneller Ebene: Der Benutzer irrt sich in Bezug auf den Adressaten oder den Inhalt einer Aktion. Programmierer haben hier serverseitig die Aufgabe, diese Attacken so weit es geht zu erschweren oder über eine Gesamtlösung auszuschließen. So lassen sich theoretisch Phishing-Attacken durch Signierung jeder einzelnen Transaktion unterbinden.

Eine andere Klassifizierung teilt die Angriffe in solche ein, deren Abwehr lediglich ein erweitertes Internetbedrohungsmodell voraussetzt, bei dem die Kommunikationskanäle, die verwendeten Protokolle und die Benutzer betrachtet werden und solche, bei denen ein hostbasiertes Bedrohungsmodell zum Tragen kommt (also zum Beispiel Angriffe durch kompromittierte Programme). Diese Unterscheidung ähnelt stark derjenigen zwischen „Netzattacke“ und „lokaler Attacke“. Ein weiteres Kriterium kann die Formation des Angriffs sein: Ist es ein direkter Angriff zwischen zwei Beteiligten oder spielt eine dritte Instanz eine Rolle? Im Web-Bereich kommt hier die Schwierigkeit hinzu, dass sich Angriffe meist

zwischen zwei direkt Beteiligten abspielen, Ausgang und Zielpunkt des Angriffs jedoch eine dritte Instanz ist.

2.1.2 Eine Übersicht der häufigsten Attacken auf Applikationen

Mittlerweile kennt hoffentlich jeder Softwareentwickler, der Web-Applikationen entwickelt, das Open Web Application Security Project (OWASP, www.owasp.org) und dessen exzellenten „Guide to building secure web applications“. Eine andere sehr gute Quelle von Erklärungen, wie Web-Attacken eigentlich funktionieren, findet man bei [Hus]. OWASP veröffentlicht regelmäßig eine Liste der Top-Ten-Angriffsformen auf Web-Applikationen, die wir kurz analysieren wollen:

- A1 Unvalidated Input,
- A2 Broken Access Control,
- A3 Broken Authentication and Session Management,
- A4 Cross Site Scripting,
- A5 Buffer Overflow,
- A6 Injection Flaws,
- A7 Improper Error Handling,
- A8 Insecure Storage,
- A9 Application Denial of Service,
- A10 Insecure Configuration Management.

Angeblich richten sich 70% der Attacken auf Applikationen und nur wenige auf Betriebssysteme oder Netzwerkkomponenten – ein klarer Hinweis auf die Schwächen der Applikationssoftware in Bezug auf Security. Sollten in dieser Liste nicht auch Phishing-Attacken auftauchen? Normalerweise sind dies semantische Attacken auf Client-Seite. Die Frage ist nur, wie weit serverseitig diese Attacken erleichtert werden. Dazu sehen wir weiter unten ein Beispiel, in dem Cross-Site-Scripting zu leichter Durchführbarkeit von Phishing-Attacken führt. Interessant sind auch „Meta-Attacken“ wie das sog. *War-Googling*, d. h. die Verwendung von Google zum Finden von ungeschützten URLs auf gefährliche Administrationsfunktionen in Application-Servern wie z. B. die JMX-Console (s. u.). Lassen Sie uns die zehn Angriffsformen nun grob in drei Kategorien aufteilen:

Input-/Output-bezogene Angriffe:

- A1 Unvalidated Input,
- A4 Cross Site Scripting,
- A5 Buffer Overflow,
- A6 Injection Flaws,
- A7 Improper Error Handling,
- A9 Application Denial of Service.

Infrastrukturbezogene Angriffe:

A8 Insecure Storage,

A9 Application Denial of Service,

A10 Insecure Configuration Management,

AAA (Authentication, Authorization, Accounting)-bezogene Angriffe,

A2 Broken Access Control,

A3 Broken Authentication and Session Management,

A9 Application Denial of Service.

Offensichtlich machen die Input-/Output-bezogenen Schwächen den größten Teil der Attacken aus. Man sollte also vermuten können, dass dieser Tatsache softwaretechnisch Rechnung getragen wird und jedes Web-Framework mit einem ausgezeichneten Sub-Framework für sämtliche Fälle von Input-/Output-Validierung geliefert wird. Außerdem sollten diese Validation-Frameworks auch separat als intelligente Proxies in der Infrastruktur einsetzbar sein, um eine Defense-in-Depth zu erreichen. Ein Beispiel dafür ist das `mod_security`-Modul für den Apache-Webserver. Trotzdem muss konstatiert werden, dass generell die Verfügbarkeit kompletter Frameworks zur Validierung noch sehr mangelhaft ist (siehe dazu auch die schöne Zusammenstellung gängiger Web-Frameworks in Bezug auf die unterstützten Validierungsmethoden in [Vela]). Die Autoren besprechen hier die einzelnen Schwächen und ihre Behebung durch das plattformübergreifende Tool HDIV).

Neben der großen Zahl von Input-/Output-Problemen fällt auch die Rolle der Authentisierung und Autorisierung von Requests ins Auge. Diese unterscheiden sich von den Input-/Output-Problemen dadurch, dass sie nicht nur am Eingang von Firmen oder Applikationen geprüft werden müssen, sondern auch im weiteren Verlauf von Requests durch ein Unternehmensnetzwerk – genauer gesagt, an jeder Trust Boundary im Unternehmen.

Aber reicht es tatsächlich aus, am Eingang des Unternehmens Input/Output zu prüfen und die weiter hinten gelagerten Komponenten auf das Ergebnis der Prüfung vertrauen zu lassen, z. B. die Businesslogik in EJBs? Streng genommen muss jede Komponente nicht nur Authentisierung und Autorisierung prüfen, sondern auch den Input bzw. den Output. Dass dies in vielen Fällen unterlassen wird, ist eine Frage von Performance und Aufwand für die Entwicklung. Anders gesagt: Wenn nur am Eingang kontrolliert wird, dann haben die Filterfunktionen die Aufgabe, den Input wirklich vollständig zu prüfen, also auch gefährliche Zeichen für weiter hinten kommende Komponenten herauszufiltern. Das ist im Grunde genommen ein gefährliches Antipattern, da es Vertrauensbeziehungen schafft, die jederzeit von beiden Seiten ausgehebelt werden können, z. B. durch Änderungen im Source Code der Businesslogik (oder dem Einsatz einer anderen Datenbank), von denen der Filter nichts mitbekommt.

Denial-of-Service-Angriffe auf Application Level stellen ebenfalls ein interessantes Problem dar, da sie in allen Gruppen vertreten sind. Sie werden weiter unten detaillierter besprochen.

2.1.3 Lokale Angriffsformen

Lokale Angriffsformen sind dadurch gekennzeichnet, dass der Angreifer direkten Zugriff auf die angegriffene Maschine bzw. Software bekommt. Dabei verwendete Angriffsformen sind:

- das Ausnutzen von Schwächen in der Erzeugung von Pseudo Random Numbers (PNRG),
- die hostinternen Attacken durch Symbolic Links,
- nicht-atomare Filesystemzugriffe,
- das Ausnutzen von GUI-Nachrichten unter Windows zur Kontrolle anderer Applikationen (Shatter-Attacks),
- Buffer Overflows und
- Installation von Trojanischen Pferden oder Viren.

Bei diesen als „lokale“ Angriffe bezeichneten Angriffen ist der Angreifer als Benutzer des lokalen Systems angemeldet. Die Einordnung der Viren oder Trojanischen Pferde unter die lokalen Angriffe mag überraschen, „fängt“ man sich doch umgangssprachlich diese Formen von Malware im Internet ein. Diese Angriffe werden also zunächst über das Netzwerk ermöglicht, sie korrumpieren jedoch in ihrer Auswirkung das lokale System und damit die hostbasierte Sicherheit. Diese Konfusion lässt sich auch marketingtechnisch gut ausnutzen. So versprechen manche Hersteller von Antivirensoftware die „Sicherheit des Internet“ verbessern zu wollen – gerade so, als ob es für das Internet ein Problem darstellen würde, wenn sich jemand einen Virus oder Trojaner herunter lädt oder auf eine Phishing-Attacke hereinfällt.

Gemeint ist häufig wohl stattdessen die Sicherheit *im* Internet, das heißt, während man online ist. Aber auch hier zeigt sich, dass die Angriffe, die sich speziell auf die Verbindung beziehen – also die Tatsache, dass ein User über das Internet mit einem Server verbunden ist und jemand die Daten mitlesen oder verändern kann – eher in der Minderzahl sind. So konnte ein populärer Dienstanbieter wie E-Bay noch bis vor kurzem sein Geschäft überwiegend ohne Kanalverschlüsselung und damit ohne Schutz der Integrität und Vertraulichkeit anbieten. Für die heute gängigen Attacken muss man daher schon vom Bedrohungsmodell der verteilten Systeme ausgehen, das bedeutet sich gegenseitig misstrauende Teilnehmer (Rechner und Benutzer).

2.1.4 Zeitliche Entwicklung der Attacken

Greift man nur die Attacken auf Authentisierung bzw. deren Credentials heraus und betrachtet deren zeitliche Entwicklung (siehe Abb. 2.1), so ergibt sich eine „Leiter“ aus eingesetzter Sicherheitstechnologie und den jeweiligen Attacken: Am Anfang stand das Multi-User-System, dessen Sicherheit meist durch verschiedene Passwörter der Benutzer erreicht wurde. Bereits damals gab es Attacken auf die

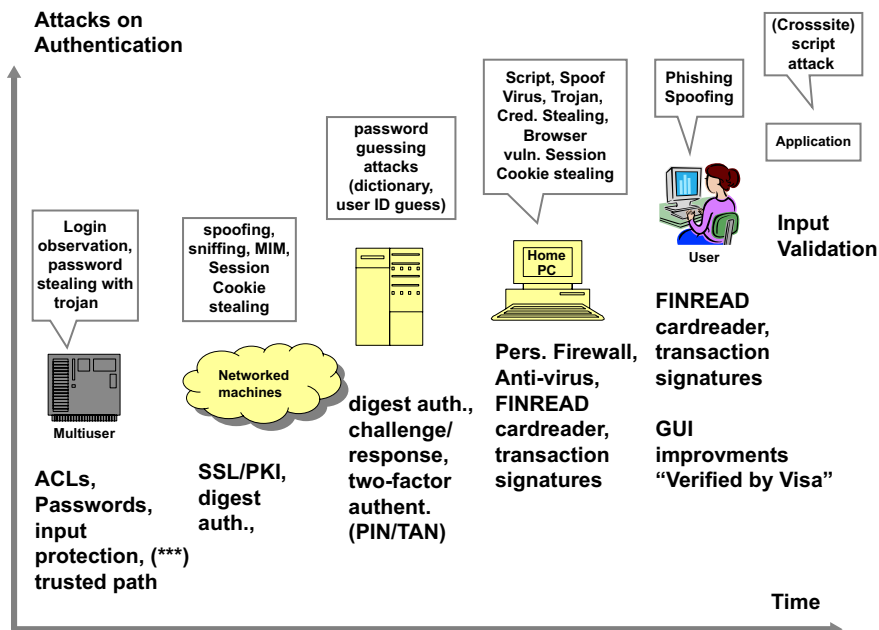


Abb. 2.1 Hierarchie der Authentisierungsmethoden und daraus resultierende Attacken

Credentials in Form von Trojanern oder dem schlichten Mitlesen über der Schulter des betroffenen Users – was aus den heute überall implementierten „*****“ als Echo der für das Passwort eingetippten Zeichen resultierte (eine Kritik daran sowie grundsätzliche Vorschläge zur Verbesserung der Passwordialoge finden sich bei Peter Gutmann, *Security Usability Fundamentals*, [Gutm] S. 111ff.). Die Vernetzung der Rechner brachte völlig neue Gefahrenpunkte. Auf das Ablauschen von Daten auf dem Transportweg war die Antwort SSL. Mögliche Man-in-the-middle (MITM) Attacken glaubte man durch das Konzept der Zertifikate sowie durch vertrauenswürdige Autoritäten zu deren Erstellung in den Griff zu bekommen. Mit der zunehmenden Absicherung der Transportwege geriet die Authentisierung auf Seiten der Server mehr in den Blickpunkt: Auf die Schwäche der Passwörter antwortete man mit Multi-Faktor-Authentisierungen mittels PIN/TAN oder Handy. Als die serverseitige Authentisierung sicherer wurde, begann der Angriff auf die Plattform der User: den PC. Trojanische Pferde, die Credentials wie PIN/TAN lokal ausforschen und weiterschicken, können mithilfe von Viruscheckern nur eingeschränkt bekämpft werden. Auf die semantischen Attacken in Form von Phishing waren die Antwort dynamische Challenge-Response-Verfahren mithilfe von kleinen Rechnern, die die Challenges zugesandt bekommen und Responses errechnen, unabhängig von der unsicheren PC-Plattform. Dynamische Man-in-the-Middle-Attacken sowie Trojanische Pferde, die Keyboard-Logging etc. durchführen können, erfordern jedoch eigentlich Transaktionssignierung durch sichere Smartcard-Systeme auf Seite der Kunden – was jedoch die meisten Firmen

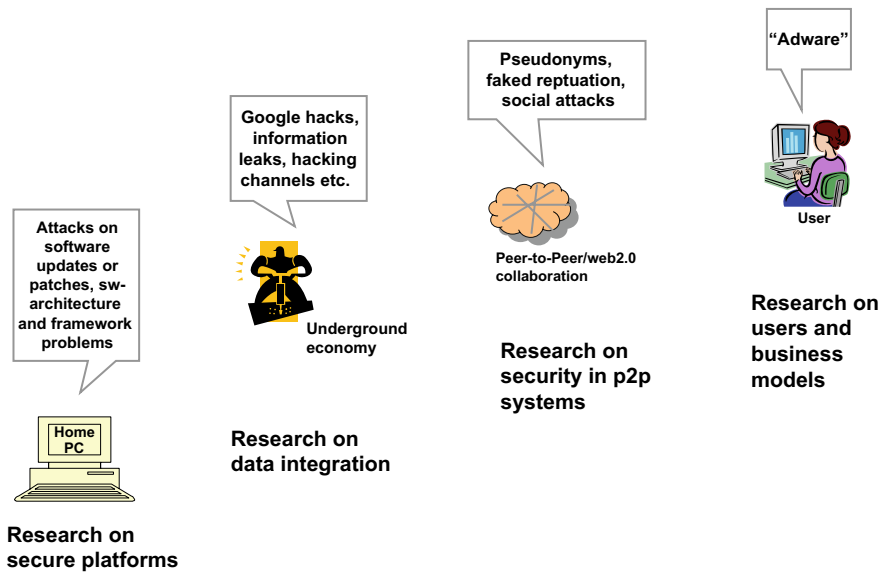


Abb. 2.2 Neue Angriffsformen und die Gegenmaßnahmen

aufgrund der Kosten sowie erschreckend schlechter Usability [Gutm] scheuen. Mit den Cross-Site-Scripting (XSS) Attacken geriet wieder der Server bzw. diesmal die Applikation auf dem Server in den Blickpunkt der Angreifer. Jetzt werden Schwächen der Input-Validierung der Applikation dazu benutzt, die Credentials eines Benutzers zu stehlen oder in dessen Namen Bestellungen zu tätigen oder generell Effekte zu erzielen (z. B. das Rücksetzen von Geräten auf den unsicheren Auslieferungszustand).

Aber auch außerhalb des Bereichs der Authentisierung gibt es genügend Beispiele für Paare aus Attacke und Abwehr. Abbildung 2.2 zeigt beispielsweise Buffer Overflows und als Antwort der Betriebssysteme Non-Executable-Stacks, die Randomisierung des Adressraums oder der Einsatz von Magic Fields (siehe dazu den Abschnitt zu Buffer Overflows weiter unten). Welche Angriffsformen (lokal oder über das Netzwerk) sind nun gefährlicher? Die Antwort darauf erweist sich als erstaunlich schwierig. Der momentane Trend geht jedoch eindeutig in Richtung semantischer bzw. sozialer Attacken, wobei auch die „klassischen“ Attacken, z. B. auf die Plattformensicherheit, sich weiter ausdehnen werden. Leider sind hierzu die Abwehrmaßnahmen noch weit weniger entwickelt als z. B. bei der Übertragung von Daten über das Internet.

Bei der Betrachtung der Angriffsformen fallen einige Dinge ins Auge: Die Angriffsziele wechseln und kehren auch wieder zurück, dann allerdings mit neuen Angriffstechniken. Die Angriffstechniken selber werden zunehmend kombiniert (Phishing mit XSS-Attacken). Die Abwehrmaßnahmen sind den Angriffen nie weit voraus und vor allem: Sie beseitigen häufig die Schwächen nicht wirklich. So ist die Gefahr von Man-In-The-Middle-Attacken längst nicht durch den Einsatz

von SSL gebannt, da viele User die Konzepte hinter Zertifikaten und CAs nicht verstehen. Manche Abwehrmaßnahmen können sogar neue Angriffsvektoren schaffen oder sie verwenden, z. B. problematische visuelle Marker, deren schlechte Usability letztlich nicht für mehr Sicherheit sorgt.

An den Attacken zeigt sich sehr deutlich, dass von Sicherheit nur noch im Zusammenhang des Gesamtsystems, bestehend aus den beteiligten Plattformen, den Netzwerken sowie den unterschiedlichen Akteuren, gesprochen werden kann. Es gibt nicht mehr den einzelnen Angriffspunkt oder die einzelne Angriffstechnik. Stattdessen muss sich eine IT-Lösung heute in dem genannten Systemzusammenhang bewähren, wenn sie sicher genannt werden will.

2.2 Die Attacken im Einzelnen

Hier können wir uns auf eine kurze Beschreibung der wesentlichen Attacken beschränken. Für Web-Applikationen sind ausgezeichnete Informationen im bereits erwähnten Web Application Security Guide unter www.owasp.org verfügbar. Detaillierte Beschreibungen finden sich ebenso im Buch zur Sicherheit von Web-Applikationen von [Hus]. Aktuelle Arbeiten zu Angriffen auf allen Ebenen finden sich auch auf der Webseite zum Buch.²

2.2.1 Externe Attacken auf Applikationslevel

Cross-Site-Scripting (XSS) Attacken

Cross-Site-Scripting-Attacken sind in mehrfacher Hinsicht interessant. Sie sind einmal ein Angriff über einen gutgläubigen Dritten, mit dem das Opfer eine Vertrauensbeziehung hat. Es gibt sie in persistenter Form (der eingeschleuste Code wird beim Server abgespeichert und später an andere ausgeliefert, z. B. als Profildaten eines Users) wie auch in nicht-persistenter Form (der Server prüft eine Eingabe, die Code enthält, nicht richtig, sondern sendet die Daten an das Opfer).

Cross-Site-Scripting-Attacken sind deshalb so gefährlich, weil sie das Vertrauensverhältnis, das ein Nutzer mit einem Server unterhält, ausnutzen und missbrauchen. Für den Intranetprogrammierer kommen diese Attacken häufig völlig überraschend, weil bössartige Nutzer im „User Conceptual Model“ eines Intranetentwicklers nicht vorkommen. Die Möglichkeit von Cross-Site-Scripting-Attacken machen somit noch einmal deutlich, dass im momentanen Sicherheitsmodell für das Web die Server größtenteils die Verantwortung für die Sicherheit des Clients übernehmen müssen. Abbildung 2.3 zeigt die prinzipielle Funktionsweise einer Cross-Site-Scripting-Attacke (gezeigt ist die nicht-persistente Form des Angriffs).

² www.kriha.de/krihaorg/sicheresysteme.html

Cross-Site Scripting (XSS)

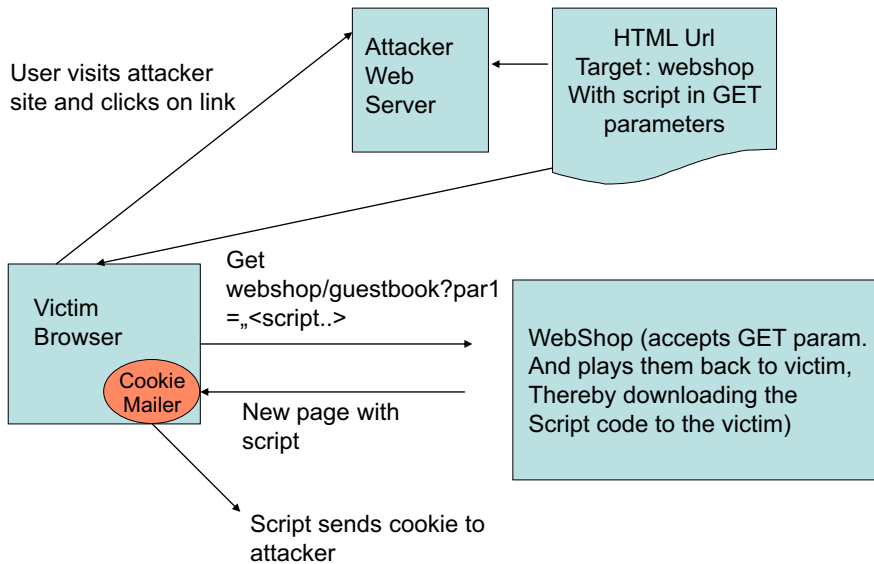


Abb. 2.3 Cross-Site-Scripting-Attacke (nicht-persistente Form)

Der Webmailer prüft hier nicht alle Felder einer empfangenen Form (die tatsächlich vom Angreifer stammt), sondern gibt die Felder ungeprüft an den Client zurück. So kann der Angreifer ein eigenes Script in einer Form platzieren, die für das Opfer vom Webmailer zu kommen scheint. Der Webmailer könnte allerdings neben der notwendigen Feldprüfung auch ein dynamisches Tagging der von ihm ausgelieferten Formulare vornehmen und dadurch erkennen, dass er nie diese Form ausgeliefert hat. Das fremde Script wird also über den Webmailer geladen und wird damit im Vertrauenskontext des Browsers mit dem Mailer ausgeführt. Damit könnte es z. B. das Cookie des Mailers an den Angreifer schicken.

Wie findet man solche Schwächen? Eine Möglichkeit ist das systematische Einfügen von Script in alle Felder von Formularen einer Applikation sowie in alle URLs, die mit Parametern arbeiten. So könnte etwa das folgende Script eingefügt werden:

```
<SCRIPT>alert('Vulnerable!')</SCRIPT>
```

Wenn auf dem Browser des Clients dieser Alarm ausgelöst wird, hat man eine XSS-Schwäche gefunden. Für systematische Tests verwendet man einen sogenannten „Fuzzer“. Das eingefügte Script sollte dabei Einträge in ein Log vornehmen. Abbildung 2.4 zeigt die persistente Version dieser Attacke.

Hier landet das vom Angreifer gepostete Script erst einmal in der Datenbank des Mailers. Jeder Benutzer dieses Mailers, der sich dann das Profil des Opfers anschaut, erhält damit automatisch ebenfalls das Script im Browser. Die weiteren

Injection Attack

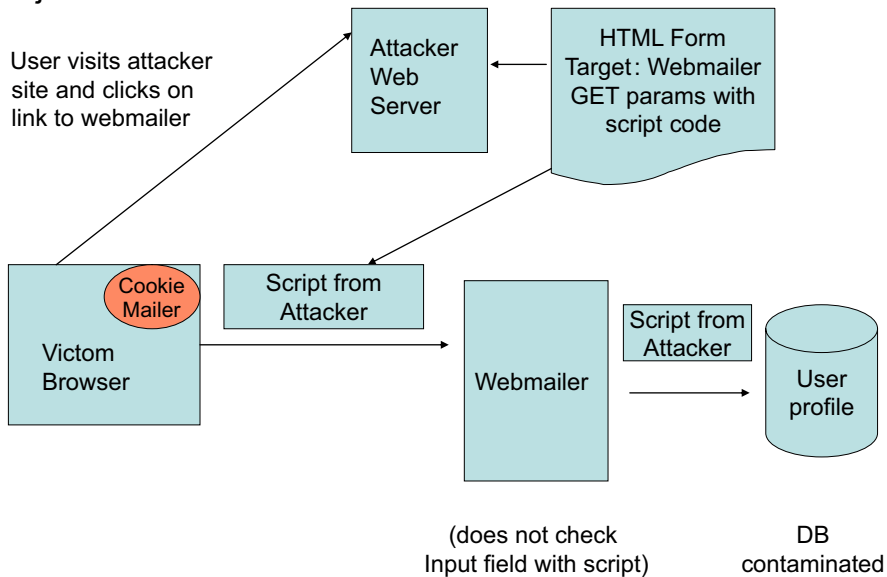


Abb. 2.4 Cross-Site-Scripting-Angriffe (persistente Form)

Angriffsmöglichkeiten können dann wie im oberen Fall auf die Cookies abzielen. Dieser Angriff ist der SQL-Injection sehr ähnlich, nur dass dort nicht der Browser, sondern der Mailserver selbst durch das eingeschleuste SQL (das z. B. Datenbankmanipulationen oder System Utilities ausführen kann) das Opfer wäre. Wir sparen uns an dieser Stelle die Beschreibung der SQL-Injection und verweisen auf das Security-Kapitel in [New04] für eine detaillierte Darstellung inklusive einer teilweisen Lösung des Problems über sogenannte „Prepared Statements“. Diese betten Input von Seiten des Clients fest in eine vorgeparste Struktur ein, d. h. der Input wird als strikt als Terminal-Symbol behandelt und keiner weiteren Interpretation (z. B. durch den SQL Parser) unterzogen. Dadurch kann in den User-Input eingebetteter SQL Code keinen Schaden anrichten. Mehr zur Frage des Parsens von Input im Kapitel zu formalen Methoden weiter unten.

Abwehrmaßnahmen für die Cross-Site-Scripting-Angriffe sind neben einer systematischen Input-Validierung auch eine systematische Output-Filterung auf Scripts. Mehrere Google-Artikel beschäftigen sich detailliert mit den Maßnahmen gegen XSS Attacken, siehe <http://code.google.com/p/doctype/wiki/ArticleXSS>.

Cross-Site-Request-Forgery (XSRF) Attacken

Wird eine Sessionsemantik vom Server angeboten, so sollte der Server grundsätzlich die empfangenen Forms darauf prüfen, ob sie tatsächlich von ihm ausgeliefert wurden (z. B. über eine Random ID, die serverseitig einige Zeit gespeichert wird).

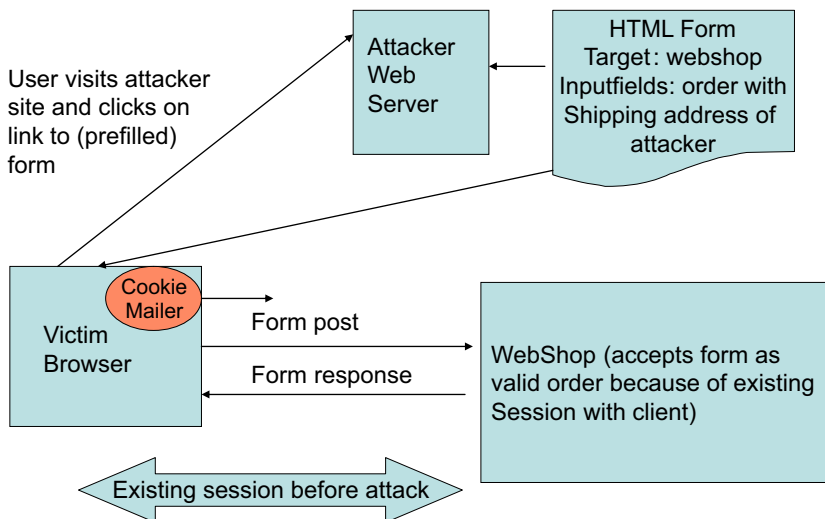


Abb. 2.5 Cross-Site-Request-Forgery

Wie man im Beispiel oben sehen kann, schleust ein Angreifer anderenfalls ausgefüllte Formulare in eine bestehende Session ein, ohne dass der Benutzer davon etwas merken muss. Nochmals: Die Sicherheit des Clients ist Sache des Servers!

Das Einschleusen von Requests in eine existierende Session wurde früher als „Web-Trojaner“ bezeichnet und heute als *Cross-Site-Request-Forgery* (s. Abb. 2.5).

Die Gefahr bei dieser Attacke liegt darin, dass zwischen einem Client und einem Server eine Session existiert. Wenn ein Link auf der Seite eines Angreifers auf den Server zeigt, und das Opfer klickt auf den Link, dann geht der Request an den Server. Gleichzeitig wird existierende Sessioninformation (z. B. das Session-Cookie) vom Browser an den Server übermittelt. Der Server erkennt damit den Request sofort als valide an. Ein Beispiel für so einen XSRF-Angriff ist das Zurücksetzen eines Home-Routers auf den Auslieferungszustand (mit evtl. bekanntem Root-Passwort) durch einen einzigen Link der im REST-Stil durch einen GET einen Reset des Routers auslösen kann: GET 192.168.1.1/ResetRouter [Schlott].

Hier eine kleine Liste von Eigenschaften der XSRF-Attacke nach Mario Heidrich [Heid], übersetzt und angereichert um eigene Anmerkungen:

- Während man in einer Applikation eingeloggt ist, kann jede andere Applikation einen Request an diese Applikation schicken, inklusive der eigenen Privilegien. Dies bedeutet, dass der Browser so implementiert ist, dass er jegliche Cookies der Applikation mitschicken wird (z. B. Session-Cookies) oder den Einstieg in eine existierende SSL-Session erlaubt.
- Je länger die Session dauert, desto höher ist das Risiko. Die längere Dauer erhöht schlicht die Wahrscheinlichkeit, in einem anderen Browserfenster auf bösartige Links zu stoßen.

- Es ist egal, ob die angreifende Applikation GET- oder POST-Requests verwendet. POST ist nur marginal schwieriger. GET-Aufrufe sollten ohnehin mit einer idempotenten Semantik verwendet werden.
- Cross-Site-Scripting ist der beste Freund von XSRF. Der Token-Ansatz zur Absicherung von Formularen durch den Server scheitert, wenn der Angreifer das Cookie stehlen kann, auf dem die Token-Generierung beruht.
- Soziales bookmarking ist der zweitbeste Freund von XSRF: Das Opfer muss ja auf einen Link des Angreifers gelockt werden. Auf einer Community Site sind Angreifer und Opfer sich bereits ganz nahe!
- Komplexe Formulare oder Transaktionen, die aus mehreren Schritten bestehen, sind kein Schutz gegen XSRF – man muss nur mehr als einen Request abfeuern. Die Erlaubnis, Javascript auszuführen, erleichtert mehrfache Requests natürlich erheblich bzw. macht sie erst möglich.

Eine interessante Frage in diesem Zusammenhang ist allerdings, ob nicht der Browser helfen könnte, zumindest das Einschleusen von Requests in existierende Sessions zu erkennen. Dazu müsste der Sessionmechanismus jedoch für den Browser erkennbar sein, denn die Bedeutung der Cookies bzw. deren Gültigkeit ist ihm verschlossen – mit Ausnahme der Basic-http-Authentisierung erfährt er nichts von der Session. Eine radikale Möglichkeit wäre indes, beim Vorliegen von Cookies eines Servers bzw. einer existierenden SSL-Verbindung zu einem Server die Annahme eines Links zu diesem Server von einer anderen Site zu verweigern. Die NoScript Firefox Extension verweigert z. B. das Ausführen von Cross-Domain-Form-Submits.

Weitere Möglichkeiten der Vermeidung von XSRF-Attacken beruhen darauf, dass Server oder WAF prüfen, ob eine Page bzw. eine URL kürzlich an den Client geschickt wurde. Ist dies nicht der Fall, wird der Request des Clients abgelehnt. Implementiert wird dies heute in vielen Frameworks durch die Erzeugung eines nicht-vorhersagbaren Token, das in die Form eingebettet wird und das in einem Zusammenhang mit einer SessionID des Clients steht. Wird allerdings dann durch eine Cross-Site-Scripting-Attacke diese SessionID des Clients entführt und an den Angreifer geschickt, kann dieser die Form in voller Gültigkeit konstruieren. Alternativ könnte man eine zusätzliche Authentisierung der Transaktion bei wichtigen Requests verlangen. Allerdings setzt dies die Verwendung einer PIN/TAN bzw. einer digitalen Signatur auf Client-Seite voraus, was normalerweise nicht gegeben ist. Man könnte sich aber eine leichtgewichtige Implementierung einer temporären PIN/TAN-Lösung folgendermaßen vorstellen: Aus dem Session-Cookie wird bei Beginn der Session über ein Script ein Array von 20 TANs durch wiederholtes hashing erstellt. Beginnend mit dem letzten Hash-Wert wird dann bei wichtigen Requests dieser Array, beginnend mit dem letzten Hash-Wert pro Request an der Server geschickt. Der Server hat sich den gleichen Array von Hash-Werten aus dem Session-Cookie generiert und kann nun sehr leicht prüfen, ob der Request des Clients vom korrekten Hash-Wert begleitet wird. Dieser Hash-Wert kann bei einer XSRF-Attacke vom Angreifer nicht mitgeliefert werden, da er ihm nicht bekannt ist.

Web2.0-Applikationen haben typischerweise einen erhöhten Bedarf an (legalen) Cross-Site-Requests, sodass die Frage der Validierung solcher Requests von großer Bedeutung ist. Das W3-Konsortium hat daher kürzlich einen ersten Draft einer Spezifikation für legale Cross Site Access Control vorgestellt (s. [W3]). Die Idee beruht auf einem serverseitigen Tagging in Form von neuen http-Headern, die ausdrücken, von welchen Sites ein Cross-Site-Zugriff auf bestimmte Gebiete des Servers erlaubt sein sollen. Diese Information kann auch per Meta-Elemente in XML-Files eingebettet werden. Es liegt dann in der Verantwortung des Browsers, derartige Anweisungen strikt zu befolgen.

Trotz der großen Bedeutung von XSRF-Attacken tauchen immer wieder Applikationen auf, die durch diesen Angriff verwundbar sind. Häufig stellt sich heraus, dass das verwendete Framework zwar sehr wohl in der Lage ist, sichere Token zu erzeugen, den Entwicklern jedoch die Notwendigkeit dieser Maßnahme nicht klar war und sie daher auch nicht eingeschaltet wurde.

Surf Jacking

Die von Sandro Gauci [Gauc] unlängst geschilderte Surf-Jacking-Attacke ist aus zwei Gründen interessant: Sie zeigt, dass dem Moduswechsel innerhalb einer Software besondere Aufmerksamkeit gewidmet werden muss, um Sicherheitsprobleme zu vermeiden. Außerdem stellt sich bei der Attacke erneut die Frage, ob sich die gängigen Browser tatsächlich richtig verhalten – technisch und in Bezug auf die sichere Benutzbarkeit.

Die Surf-Jacking-Attacke selbst ist recht einfach: Eine Firmenseite benutzt sowohl http also auch https. Wenn der Entwickler vergessen hat, die Option „Secure Cookie“ zu setzen, dann liefert der Browser das kritische Session-Cookie auch über die normale http-Verbindung aus. Falls es einem Angreifer gelingt, einen Man-in-the-Middle zu spielen, kann er dieses Session-Cookie abfangen. Der Angreifer kann darüber hinaus durch eigene Links versuchen, den Benutzer zum Anklicken eines unsicheren http-Links auf der Firmenseite zu bewegen.

In diesem Fall ist es also der Übergang von https zu normalem http, bei der die Vulnerability auftritt. Man erinnere sich: Ein ähnliches Problem hatte man schon in der umgekehrten Reihenfolge beim Übergang von http zu https bei der gleichen Seite. Viele Entwickler vergaßen hier, ein neues Cookie zu erstellen und benutzten das vorherige der normalen http Session – obwohl dieses Cookie während der Übertragung ja völlig ungesichert war. Jetzt wird beim Weg zurück zum normalen http das wichtige Session-Cookie per Default verraten. Stellt sich die Frage, wieso der Browser seine Kenntnis der Kommunikation vom Client zur Firmenseite nicht nutzt, um das Cookie zu schützen? Und wieso ist das sichere Verhalten nicht der Default, sondern muss vom Entwickler extra gewählt werden?

Die Probleme von Entwicklern beim Wechsel von Sicherheitsmodi sind bekannt (z. B. weist Eric Rescorla in seinem Buch zu SSL und TLS [Resc] auf genau diese Problematik hin). Surf Jacking ist ein schönes Beispiel dafür, wie verschie-

dene Arten von Defiziten im Verbund Entwickler-Applikationssoftware-Tools letztlich eine Schwäche im System ermöglichen.

XML-basierte Attacken

XML ist bekanntlich ein Dokumentformat, und Dokumente können ja wohl nicht bössartig sein, oder? Dass aber auch Textdokumente Buffer-Overflow-Code enthalten können, haben wir bereits diskutiert. Aber was kann an der Abarbeitung von Dokumenten – speziell XML-Dokumenten – gefährlich sein? Nehmen wir als Beispiel einen Web-Service, der XML-Dokumente annimmt, transformiert und zurückgibt. Abbildung 2.6 visualisiert aktuelle Diskussionen in der XML-Dev-Mailing-Liste.

Möglich sind sowohl Angriffe auf interne Ressourcen, wenn eine Entity Reference darauf zeigt, als auch der Missbrauch des Dienstes für einen DoS-Angriff auf andere Hosts. Der Grund für die Gefahren liegt tief im Design von XML bzw. HTML. Beide Techniken stellen sogenannte „Implicit Authority Demands“, d. h. sie nehmen für sich bestimmte Rechte automatisch an. So ist es für den Parser selbstverständlich, dass er Entity-Definitionen über den Entity-Manager auflösen lässt. Ein Browser berücksichtigt bei Zugriffen auf andere Sites das „same origin“-Prinzip, d. h. er erlaubt normalerweise keinen Datentransport zwischen Domains durch den Browser hindurch. Macht das Ihr XML-Processing-System auch? Die Ausnahmen und Workarounds werden wir unten betrachten, wenn wir Web2.0-Techniken diskutieren. Die gleiche Problematik gilt auch für Extension Functions, wie sie in XSLT vorkommen, und XPath Expressions: Im Grunde genommen ist alles gefährlich, was das Verhalten der eigenen Software steuert und zu Zugriffen auf eigene Ressourcen veranlasst. Dazu gehören auch DOS-Angriffe, wenn das XML-Processing-System externe Schemata oder Entities anfordert.

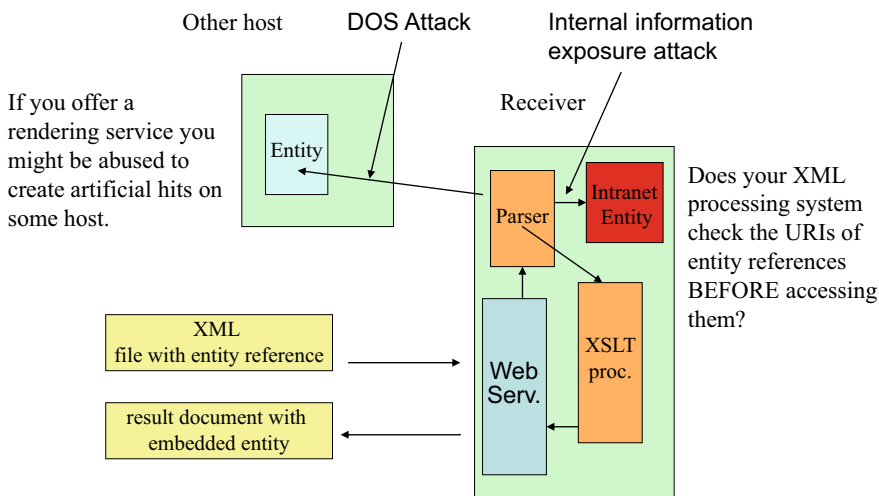


Abb. 2.6 Mögliche Gefahren beim Verarbeiten von XML-Dokumenten

XML besitzt einige eingebaute Schwächen bezüglich Zyklen bei der Verarbeitung von Entities, die zum Stillstand der Applikation führen können. Auch muss darauf geachtet werden, dass die Definition einer Schemadatei nicht durch eine andere ersetzt werden kann – anderenfalls können wichtige Syntaxprüfungen umgangen werden. Das Wichtigste ist jedoch, das Grundmuster der Angriffe zu erkennen: Externe Dokumente, die symbolische Referenzen enthalten, werden geladen. Dabei werden die Referenzen automatisch aufgelöst und in direkte Ressourcen verwandelt – typischerweise mit den Rechten des Verarbeitungssystems.

Zum Schluss noch ein praktisches Beispiel für Webservices mit XML: Amazon E-commerce Services (ECS) bietet einen Service an, der Amazon Wish Lists in beliebige Formate transformieren kann. Dies geschieht mithilfe von Stylesheets, die vom User angegeben werden. Dazu stellt der User einen REST-Request (XML über http) an Amazon, in dem das Stylesheet referenziert wird.

Dieses Vorgehen hat den großen Vorteil, dass die Daten von Amazon für beliebige Clients aufbereitet werden können, ohne dass notwendigerweise eine Installation eines Formatters oder Renderers beim Client erfolgen muss. Amazon sichert diesen Service unter anderem dadurch, dass er nicht in der normalen Amazon-Domain funktioniert, sondern in speziell dafür erstellten, wie z. B. `xml-de.amznxslt.com`. Das führt dazu, dass der Service auch die normalen SessionIDs nicht bezieht und – hoffentlich – gegen interne Zugriffe durch Links und Extensions, wie oben beschrieben, abgesichert ist. Die Möglichkeit einer indirekten DOS-Attacke bleibt allerdings weiterhin bestehen, denn der Client bringt den Amazon-Server in jedem Fall dazu, Referenzen auf andere Sites aufzulösen.

Zeichencode-Attacken (Alias-Problem und Visual Spoofing)

Bei den Attacken auf Zeichencodeebene stoßen wir auf ein altes Antipattern der sicheren Software: Die Verwendung von Alias-Definitionen. Wir haben bereits ein Beispiel für dieses Problem beim Filtern von Page Requests kennen gelernt: Pages (oder allgemeiner alle Arten von Web-Ressourcen) innerhalb des Web-Containers können im Deployment Descriptor durch Angabe der URL sowie der Zugriffsmethode und der nötigen Rollenberechtigung gefiltert werden. Aber Web-Container erlauben auch weitere Arten des Aufrufs einer Page, z. B. über den zugehörigen Servlet Class Name. In diesem Fall greift der definierte Filter dann nicht! Darüber hinaus birgt die Verwendung des Class Path zum Suchen der Klasse weitere Risiken, da er die falschen Servlets enthalten kann.

Das Antipattern im Bereich Zeichencode entsteht durch die mehrdeutige Abbildung von Codes auf Encodings und auf Glyphs (Font-Zeichen, s. Abb. 2.7): Zwar bietet Unicode eindeutige Codes für praktisch alle bestehenden Alphabete und Sprachen an, die Abbildung dieser Codes auf entsprechende Fonts und Zeichen ist jedoch nicht immer eindeutig.

Die erste Attacke besteht nun darin, durch die Verwendung ungewöhnlicher Encodings die Filtersoftware auszutricksen. Dies ist beim Alias-Antipattern immer dann möglich, wenn die Software nicht zuerst normalisiert und dann erst filtert. In

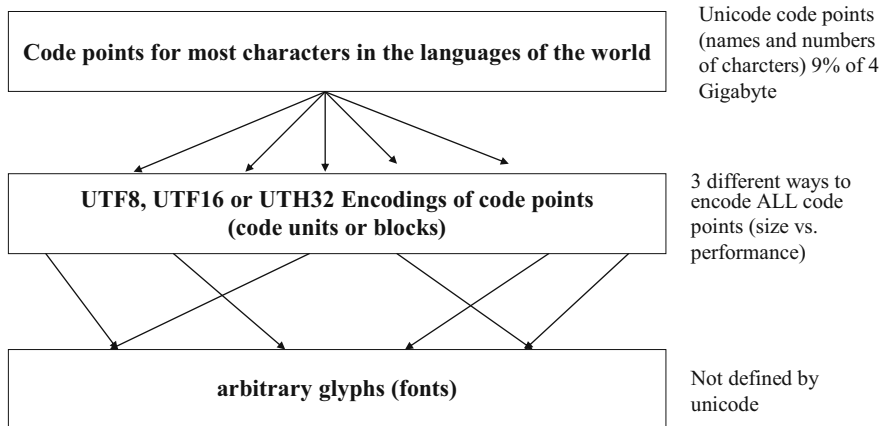


Abb. 2.7 Nicht eindeutige Abbildung von Unicode-Codepoints auf Fonts

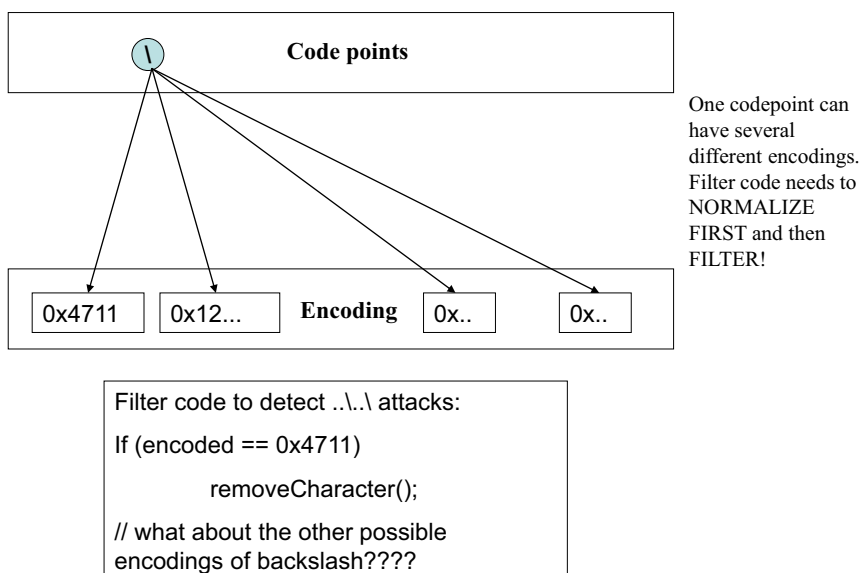


Abb. 2.8 Angriff auf Filter durch ungewöhnliches Encoding

diesem Fall werden nämlich die anderen möglichen Encodings nicht ausgefiltert und können später als normale Zeichen vom Code ausgewertet und interpretiert werden. Abbildung 2.8 zeigt genau diesen Fall.

Der Filter in diesem Beispiel versucht zwar, Backslashes zu entdecken und auszufiltern, reagiert aber nicht auf die anderen möglichen Encodings dieses Zeichens. So kann es passieren, dass dieses Zeichen als Befehl, das Verzeichnis zu wechseln, interpretiert wird.

Im zweiten Fall, dem sogenannten *Visual Spoofing* oder Homographen-Problem stellt der Mensch die entscheidende Schwachstelle dar. Grundlage des

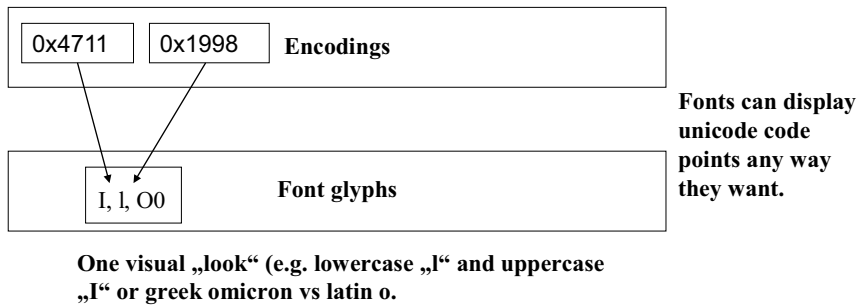


Abb. 2.9 Visual Spoofing durch gleich aussehende Codepoints

Angriffs ist die Abbildung von Codepoints auf Glyphs (Fonts), die nicht standardisiert ist (siehe Abb. 2.9). Ein Mensch kann daher bei ähnlich aussehenden Fontzeichen nicht wirklich sagen, welcher Codepoint dahintersteckt. Auf www.ebay.com kann das Zeichen „a“ ein ASCII-Zeichen sein oder aber auch das kyrillische „a“ mit dem Strich. Für die Maschine klar unterscheidbar – für den Menschen nicht. Damit kann ein Browsernutzer durch eine Phishing-Mail noch leichter auf fremde Sites gelockt werden, selbst wenn er den angezeigten Hostnamen visuell prüft.

Hier haben die Browserentwickler eine richtige Entscheidung getroffen: Sie schränken die Mächtigkeit von Unicode dadurch ein, dass sie bei Domain Names nicht den Glyph, sondern die Zahl des Encodings anzeigen, inklusive Escape-Zeichen.

Das Homographen-Problem ist Stellvertreter für eine ganze Klasse von Fällen, in denen die Maschine Unterschiede trivial erkennen kann, der Mensch jedoch nicht. Wärmebildkameras und Geigerzähler sind andere Beispiele dafür. Wenn nun aber die Maschine etwas erkennen kann, was der Mensch nicht kann – kann sie dann den Menschen warnen? Ein triviales, aber wichtiges Anwendungsfeld stellen Phishing-Mails dar, bei denen im Anchor Tag einer HTML-Mail die im HREF-Attribut angegebene URL nicht mit dem Inhalt des Anchors übereinstimmt, sondern der URL des Angreifers entspricht. Aber selbst wenn der Vergleich schwieriger wird, z. B. zwischen einer URL und einem symbolischen Namen, kann die Maschine den Menschen entlasten: Die Maschine kann sich einen einmal gelernten Zusammenhang merken und ab dann überwachen. Dies geht besonders gut bei Zuordnungen von Schlüsseln zu Namen, also bei der Frage, wer welchen Public Key besitzt. Dieses Verfahren nennt man *Key Continuity Management*.

Application-DOS-Attacken

Diese Form von Attacke taucht in allen unseren Rubriken auf, da sie durch unterschiedlichste Fehler in der Applikation erleichtert werden:

- Keine Begrenzung der Anzahl von Eingaben eines Users in ein System. Leider ist dies eine Prüfung, die States voraussetzt (also eine Kontrolle, wie viel der

User bereits eingegeben hat) und die deshalb schlecht an eine vorgelagerte Prüfkomponte ausgelagert werden kann. Hier kann nur die Applikation selber prüfen.

- Keine vorgelagerte Authentisierung. Somit können unauthentisierte Requests zu weit in die eigene Infrastruktur hinein gelangen.
- Keine Filtermöglichkeit beim eigenen ISP. Dadurch könnte im Notfall der Fluss von Upstream-Paketen gestoppt werden (indem z. B. beim ISP ganze Netzwerke blockiert werden).
- Unausgewogene Systemarchitekturen, die keine Verengung von Requests innerhalb der Infrastruktur durchführen.
- Kein Caching der User Requests. Dies ist vor allem im Falle von sehr teuren, d. h. die Back-Ends stark belastenden Requests, sehr gefährlich.
- Keine Prüfung auf menschliche Clients bei öffentlichen Requests, die States verändern.

Von einem System-Engineering-Gesichtspunkt aus gesehen, der die Web-Applikation als Reihe von Queues auffasst, muss deren Konfiguration die Anzahl der Requests in Richtung der Back-Ends verringern. Abbildung 2.10 (entnommen aus [WAS]) zeigt klar diesen Effekt.

Wenn gegen dieses Prinzip verstoßen wird, entstehen oft instabile Applikationen, die bei starken Lastschwankungen sogar abstürzen können. Die Untersuchung des Request-Verhaltens sowie eine genaue Beobachtung des States, der für Clients von der Applikation oder Infrastrukturkomponenten gehalten wird, hilft, die Ge-

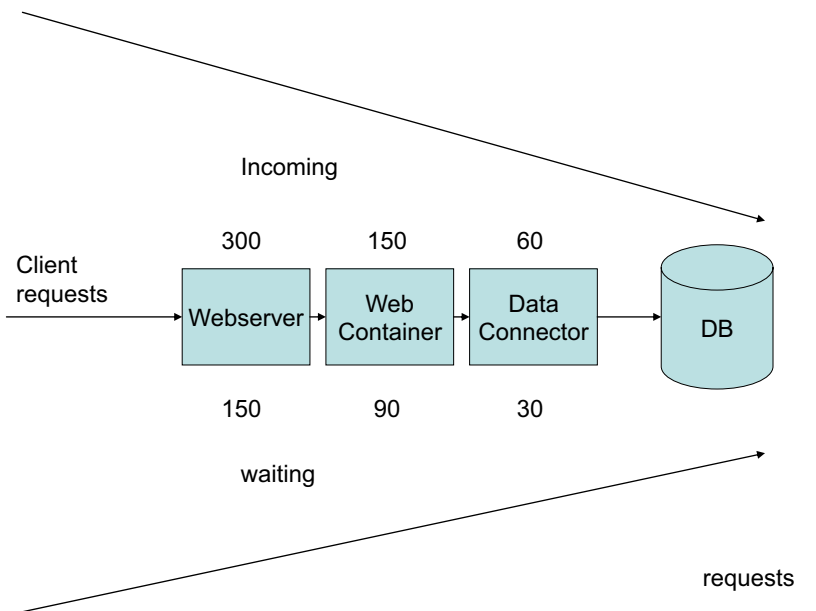


Abb. 2.10 Verringerung der Anzahl der Requests in Richtung des Back-Ends

fahr von Application-DOS-Attacken zu minimieren. Hinzu kommen Infrastrukturmaßnahmen (Load-Balancer, Cluster etc.) sowie organisatorische Maßnahmen (Filtern auf ISP-Level).

Letztlich ist jedoch gegen eine konzentrierte Distributed-Denial-of-Service-Attacke kaum ein Kraut gewachsen – eine normale Infrastruktur vorausgesetzt. Dies musste im vorletzten Jahr z. B. auch der Heise Verlag erfahren, als seine Homepage einen Tag lang nicht erreichbar war.

Meta-Attacken: War-Googling

Mittlerweile hat es sich herumgesprochen, dass Google ein ausgezeichnetes Instrument für das Entdecken von Sicherheitslücken ist. Das bezieht sich keineswegs nur auf offene Eingänge von Web-Applikationen oder Servern. Beim Suchen nach Namen von Entwicklern finden sich häufig Datenbanken, die auf öffentlichen Servern zum Zwecke des Transports nach Hause abgelegt werden. Meist beziehen sich die Queries jedoch auf bekannte Administrationseingänge, die häufig ungeschützt bleiben. Selbst wenn sich herausstellt, dass sie z. B. durch Passwörter geschützt sind, ist die Qualität der Passwörter oft sehr gering.

Dass dies keine geringe Gefahr darstellt, wurde unlängst in einer Diplomarbeit gezeigt: Bei der Entwicklung eines Frameworks kam zur Erleichterung der Administration die Java Management Extension JMX zum Einsatz. Die Konsole dieser Erweiterung ist per Default nach einer Installation aktiviert und am Web sichtbar. Abbildung 2.11 zeigt, wie mithilfe von Google Hunderte von Application-Servern lokalisiert werden konnten, deren Managementinterface komplett offen vom Internet aus zugänglich war (die Wirkung eines „Default Passwords“ kann getrost vernachlässigt werden).

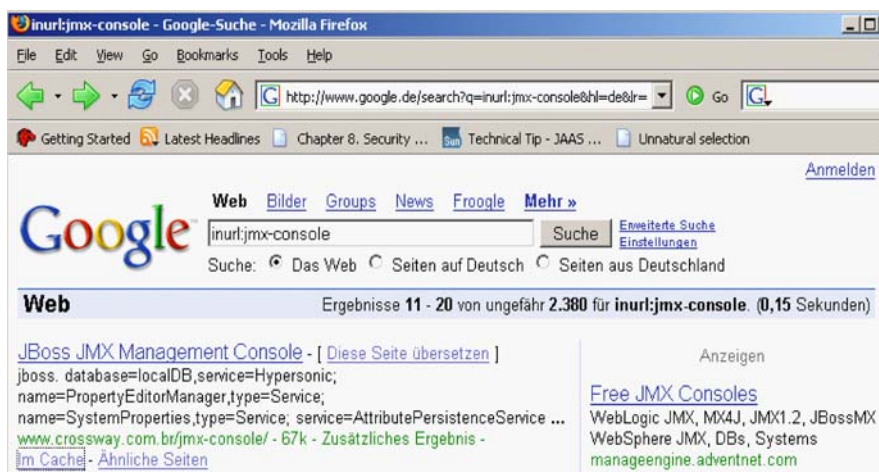


Abb. 2.11 Über Google gefundene offene JMX-Konsole

Heute sind viele dieser URLs entweder nicht mehr öffentlich oder gesichert. Ein Klick auf „Im Cache“ von Google zeigt jedoch oft noch, dass der Konsolenzugang einmal vorhanden und funktionsfähig war.

2.2.2 Lokale Attacken auf Applikationslevel

Buffer-Overflow-Attacken

Angeblich gehen 60 bis 80% aller von Angreifern übernommenen Maschinen auf das Konto von Buffer-Overflow-Attacken – mit zumindest nicht nachlassender Tendenz. Trotz dieser hohen praktischen Relevanz sind Buffer Overflows aus theoretischer Sicht kaum mehr interessant, da ihre wesentlichen Aspekte bereits mehrfach analysiert worden sind. Im Folgenden eine kleine Übersicht der Ergebnisse der Analysen:

- Buffer Overflows sind an Sprachen wie C oder C++ gebunden, die keine Memory Protection kennen, d. h. deren Semantik und Befehle durch die Verwendung von Primitiven wie direkten Pointerzugriffen auf Adressen unterlaufen werden können. Das POLA-Prinzip lässt sich auf solchen Systemen rein softwaretechnisch nicht implementieren.
- Die Gefährlichkeit von Buffer Overflows resultiert direkt aus der Sicherheitsstrategie des Betriebssystems: Stellt es viel „Ambient Authority“ zur Verfügung, kann selbst ein Solitärspiel ein gefährlicher Angriffspunkt sein. Bei Verwendung von Sandboxes oder anderen Isolationsmechanismen können sie hingegen völlig ungefährlich sein.
- Buffer Overflows können auch durch manuelle Codeinspektion nicht verhindert werden (z. B. wurde „bind“ mit dem Ziel analysiert, Overflows endgültig auszurotten – kaum drei Monate später war der nächste Angriff da).

Dennoch wollen wir Buffer-Overflow-Attacken an dieser Stelle kurz diskutieren, weil sie den Zusammenhang zwischen Zuverlässigkeit und Sicherheit von Software sehr schön untermauern. Das lässt sich daran demonstrieren, wie Angreifer die Einstiegspunkte für mögliche Buffer-Overflow-Attacken finden, nämlich über Abstürze von Software. Sehen wir uns dazu das folgende kleine Stück Code in Abb. 2.12 an, das einen Buffer Overflow produziert (ohne ihn jedoch sofort mit Schadcode auszunutzen).

Gibt man die Zeichen in der linken Spalte nacheinander ein, ergeben sich am Display die Ausgaben in der rechten Spalte. Bei Eingabe von aaaa erscheint die erste Auffälligkeit: Die Integer-Variable foo wird offensichtlich teilweise überschrieben. Bei noch mehr Eingaben des Zeichen „a“ stürzt das Programm irgendwann ab und es ergibt sich das in Abb. 2.13 gezeigte Bild der CPU-Register.

Entscheidend ist, dass unsere Zeichen es bis in das Instruktionsregister der CPU geschafft haben und dort als Adresse interpretiert werden, wo der nächste auszuführende Code geladen werden soll. Wir haben also unser Ziel erreicht, die CPU

```
#include <stdio.h>

int main(int argc, char** argv) {

    int foo=0xeeee;
    char myArray[4];
    gets(myArray);
    printf(" print integer first: %x ", foo);
    printf("%s ", myArray);

}
```

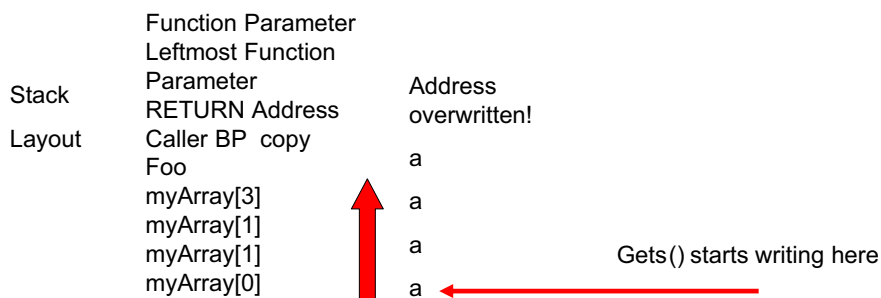
Keyboard Input (with return)	Display Output
a	Eeee a
aa	Eeee aa
aaa	Eeee aaa
aaaa	Ee00 aaaa
aaaaaaaaaaaaaaaa	Core dump with EIP = 6161616161616161 (Hex 61 == `a`)

Abb. 2.12 Beispielcode zur Erzeugung eines Buffer Overflows

Exception: STATUS_ACCESS_VIOLATION at **eip=61616161**
eax=00000012 ebx=00000004 ecx=610E3038 edx=00000000 esi=004010AE
edi=610E21A0
ebp=61616161 esp=0022EF08
program=D:\kriha\security\bufferoverflow\over.exe, pid 720, thread main
cs=001B ds=0023 es=0023 fs=003B gs=0000 ss=0023
Stack trace:
Frame Function Args
 90087 [main] over 720 handle_exceptions: Exception:
 STATUS_ACCESS_VIOLATION
 104452 [main] over 720 handle_exceptions: Error while dumping state
 (probably corrupted stack)

Abb. 2.13 Zeichen „aaaa“ sichtbar im Instruktionsregister der CPU

auf eine von uns definierte Stelle im Speicher zu lenken. Wenn wir an dieser Stelle nun Schadcode platzieren, wird ihn die CPU ausführen. Innerhalb dieses Codes haben wir die Adresse des Codeanfangs so platziert, dass sie im IP-Register der CPU landet. Wie das geschieht, ist ganz einfach: Beim Zurückkehren von einer Funktion lädt die CPU automatisch die vorher auf dem Stack gespeicherte Return-Adresse in dieses Register. Genau diesen Stack haben wir mit der gezeigten Methode überschrieben und an der Stelle der Return-Adresse unsere Anfangsadresse gespeichert (s. Abb. 2.14).



Keyboard Input (with return)	Stack layout
a	eeee a (first array element)
aa	eeee aa (first and second)
aaa	eeee aaa (first, second and third)
aaaa	ee00 aaaa (4 array elements + zero)
aaaaaaaaaaaaaaaa	aaaaaaaaaaaa (all local variables and the return address overwritten, crash on function return)

Abb. 2.14 Status des Stacks nach Buffer Overflow

Natürlich würde ein Angreifer nicht nur das Zeichen „a“ einschleusen. Das Beispiel demonstriert lediglich, wie durch das Überschreiben der Array-Grenzen zunächst die benachbarte Variable foo und anschließend auch die Return-Adresse überschrieben werden können. Der erste Überschreibvorgang tritt bereits beim Schreiben des vierten „a“ auf: Denn C beendet jeden String mit einer binären Null, für die jedoch im Array nun kein Platz mehr ist und die somit den ersten Teil der darüber liegenden Integer-Variablen foo zerstört.

Es gibt viele weitere Möglichkeiten, Schadcode einzuschleusen (s. auch [Kriha]). Dort finden sich auch Abwehrmaßnahmen auf den Ebenen des Betriebssystems, wie etwa Stack Pages nicht-ausführbar machen, zufällige Adressverschiebungen im Programm unter Anpassung der Referenzen und des Compilers, z. B. Platzierung von Array-Variablen am Beginn einer Funktion, Platzierung von sogenannten „Canaries“ – d. h. Überwachungsvariablen auf dem Stack und die Prüfung, ob sie überschrieben wurden).

Viel Zeit und Geld wird derzeit auch bei Microsoft in die Entwicklung von Analysetools gesteckt, die Buffer Overflows automatisch erkennen sollen. Längerfristig geht es jedoch auch dort eher in Richtung sicherer Programmiersprachen (siehe [Hunt]). Der Grund dafür wird im Vortrag sichtbar, den Alexander Sotirov (VMWare) und Mark Dowd (IBM) anlässlich der Black Hat Conference im August 2008 gehalten haben: Unter dem Titel „Bypassing Browser Memory Protections – setting back browser security by 10 years“ [SD] zeigen die beiden Autoren, wie die Sicherheitsmechanismen zu Verhinderung von Buffer Overflows in Windows Vista ausgehebelt werden können.

Es lohnt, auf das Paper näher einzugehen, denn es beschreibt nicht nur die Mechanismen einfach und verständlich, sondern macht die grundsätzliche Problematik der Absicherung sehr deutlich. Die Absicherung des Speichers in Vista geschieht grob in drei Bereichen:

- Sicherung der Werte auf dem Stack bzw. dem Heap. Das schließt sichere Listenverwaltung von memory chunks im Heap sowie die Platzierung von Cookies oder Canaries bzw. eine geänderte Reihenfolge von automatischen Variablen auf dem Stack ein.
- Verhinderung der Ausführung von Code in bestimmten Speicherbereichen (sog. Data Execution Protection (DEP), die teilweise Hardwareänderungen bei Intel Prozessoren benötigte)
- Randomisierung des Adressraumes von Programmen (ASLR) bzw. deren dynamisch geladene Bibliotheken.

Man muss ehrlicherweise zugeben, dass bei der Betrachtung der vielen Abwehrmechanismen, die bei Vista eingesetzt werden, es wirklich unglaublich erscheint, dass weiterhin Buffer Overflows möglich sein sollen. Wie erreichen die Autoren das?

Im ersten Schritt haben die Autoren die einzelnen Mechanismen untersucht und dabei ganz speziell auf die Ausnahmen geachtet. So stellte sich heraus, dass die Absicherung der Return-Adresse einer Funktion nur dann erfolgt, wenn eine Bibliothek mit der sog. GS-Option übersetzt wurde. Aber selbst dann erzeugt der Compiler den Prüfcode inklusive Einsatz des Canaries nur in wenigen Fällen, wenn nämlich bestimmte Formen von Stringarrays als lokale Variablen der Funktion auftreten. Beides stellt sich als Grundproblematik von Vista heraus: Optionale Features (Sicherheit durch „Opt-In“) hat sich in vielen Untersuchungen von User-Verhalten als völlig nutzlos herausgestellt. Wir scheinen einen starken „Status-Quo-Bias“ (siehe dazu Kap. 13) zu besitzen, und das gilt offensichtlich auch für Softwarefirmen). Unvollständige Sicherheitsfeatures lassen einfach zu viele Lücken. So muss ein Angreifer nicht notwendigerweise auf den Return der Funktion warten, um den Angriffscode wirksam werden zu lassen. Selbst einzelne Variablen auf dem Stack, z. B. wenn sie Funktionsadressen beinhalten, sind lohnende Ziele zum Überschreiben. Vista verwendet sog. sichere Exception Handler (SEH), deren Code nicht auf den Stack verzweigen darf. Viele Bibliotheken sind jedoch nicht mit den entsprechenden Optionen übersetzt.

Die vielen Ausnahmen erklären sich aus zwei Gründen: Kompatibilitätsprobleme und Performanceprobleme. Wenn sich herausstellt, dass eine vollständige Sicherung der Werte auf dem Stack 42% der Performance kostet, werden Hersteller sehr zögerlich beim Einsatz dieses Features sein. Für uns ist an dieser Stelle wichtig, eines festzustellen: Der Performanceaufwand der Absicherung einer unsicheren Sprache (sprich einer Sprache, die nicht memory-safe ist), ist bereits größer als der Performanceverlust beim Einsatz einer sicheren Sprache. Kompatibilitätsprobleme spielen anscheinend bei der zweiten Absicherungstechnik eine große Rolle: Data Execution Protection verhindert die Ausführung von Angriffscode z. B. auf dem Stack, leider aber auch die Ausführung von legitimen Programmen

vieler Hersteller. Leider sind sogar Microsoft-Basisbibliotheken, die von vielen Herstellern verwendet werden, nicht kompatibel mit diesem Feature.

Ausnahmen und Einschränkungen spielen auch beim dritten Mechanismus, der Randomisierung von Adressräumen, eine große Rolle. Zunächst stellen die Autoren fest, dass die Randomisierung meist nur einen kleinen Bereich betrifft, in dem der mögliche Anfang einer Bibliothek liegen kann. Dann zeigen sie, dass die virtuelle Adresse einer geladenen Bibliothek für alle Programme unter Vista gleich vergeben wird. Schließlich gibt es auch binaries, die gar nicht randomisiert werden dürfen. Größenprobleme etc. führen dazu, dass die möglichen Anfangsstellen von Code im Ram sehr eingegrenzt sind. Wir ersparen uns die Details, mit denen die Autoren mögliche Anfangsadressen von Code berechnen (s. [SD]) und erklären stattdessen kurz, wie Angreifer Code einschleusen: Eine beliebte Technik hierzu ist das sog. Heap Spraying, bei dem der Heap einer Applikation – nach Möglichkeit einer, der nicht als NX (not-executable) gekennzeichnet ist, wie z. B. der Heap der Java VM – mit Angriffscode aus dem Web vollgeladen wird. Dies kann ein Applet mithilfe von komprimierten Dateien, die heruntergeladen werden, bewerkstelligen. Die Kompression ist sehr wirkungsvoll, um sog. NOP Slides, also Nullinstruktionen, die letztlich erst zum wirklichen Anfang von Code hinführen, zu verkleinern. Dadurch müssen Angreifer nicht ganz genau die Anfangsadresse ihres Codes innerhalb der Opferapplikation anspringen.

Nachdem die Autoren im zweiten Schritt Techniken zum Aushebeln der Sicherheitsmechanismen erklären, erläutern sie im dritten Schritt die Angriffe. Fassen wir vorher noch kurz zusammen, worauf es ankommt: Es muss Code an einer Stelle platziert werden, die grundsätzlich ausführbar ist. Diese Stelle muss dem Angreifer bekannt sein, denn er muss die Opferapplikation dazu bringen, diese Stelle anzuspriegen. Dazu benötigt er meist die Kenntnis der Lage von Variablen und Funktionen, die er zum Verzweigen auf seinen Angriffscode verwenden muss. Dabei kann das Verzweigen über mehrere Indirektionen gehen (Trampoline Code), und notfalls müssen nützliche Bibliotheken dafür erst geladen und angesprungen werden.

Die Autoren erreichen diese Effekte, indem sie die Angriffsmöglichkeiten geschickt kombinieren. Interessant sind für sie vor allem Bibliotheken, die keine Absicherung des Stacks haben, die ausführbar sein müssen und die die Randomisierung des Adress Spaces negativ beeinflussen, sprich die möglichen Anfangsadressen von DLLs enger festlegen. Dies sind alles Bibliotheken, die entweder aus Kompatibilitätsgründen, aus Performancegründen oder schlicht aus Faulheit der Hersteller den Sicherheitsmechanismen von Vista nicht genügen und daher als Ausnahmen behandelt werden müssen (dem Softwareentwickler sei an dieser Stelle ein Blick auf die Algorithmen, die Vista zur Bestimmung der Ausnahmen verwendet, empfohlen. Sie sind im Paper von Sotirov und Dowd detailliert aufgeführt und von erschreckender Komplexität).

Wie erreichen die Autoren nun, dass die für die Angriffe nötigen Bibliotheken geladen werden? Sie verwenden dazu den Lademechanismus schlechthin: Den Internet Explorer. Heutige Browser sind aufgrund der Vielzahl von Protokollen und Services, die sie anbieten, auf die dynamische Erweiterung ihres Codes durch

Nachladen von Extensions (Java, javascript, flash, activeX, image rendering, videoplayer, URL Protokolle etc.) angewiesen. Angreifer können diese Extensions durch Angaben im HTML oder Javascript von Pages steuern und damit das Adresslayout des Browsers kontrollieren. Sind nun unsichere DLLs geladen, können mit Heap-Spraying Angriffsroutinen geladen werden. Dann genügt ein bekannter Fehler, um z. B. über das Auslösen einer Exception in den Angriffscode zu verzweigen.

„Two factors contribute to this problem: the degree to which the browser state is controlled by the attacker; and the extensible plugin architecture of modern browsers.

The internal state of the browser is determined to a large extent by the untrusted and potentially malicious data it processes. The complexity of HTML combined with the power of JavaScript and VBscript, DOM scripting, .NET, Java and Flash give the attacker an unprecedented degree of control over the browser process and its memory layout.

The second factor is the open architecture of the browser, which allows third-party extensions and plugins to execute in the same process and with the same level of privilege. This not only means that any vulnerability in Flash affects the security of the entire browser, but also that a missing protection mechanism in a third-party DLL can enable the exploitation of vulnerabilities in all other browser components. [SD], S. 46

Die Autoren verwenden keine neuartigen Angriffstechniken. Sie nutzen schlicht vorhandene Schwächen geschickt aus und kombinieren verschiedene Techniken. Das Problem für Microsoft ist, dass aufgrund der Probleme mit Kompatibilität, Performance, Faulheit oder Erweiterbarkeit eine schnelle Abhilfe nicht in Sicht ist.

Eine Applikation, die wie die Browser im Allgemeinen mit vollen Privilegien läuft und eine Erweiterungsarchitektur, die von außen kontrolliert werden kann – das passt offensichtlich nicht zusammen. Wir werden später aber untersuchen, ob es sogar möglich ist, eine Browsererweiterung zu laden, die komplett unter der Kontrolle des Angreifers steht, ohne dass die Sicherheit des Browsers selbst bzw. des Benutzers gefährdet ist. Als Beispiel wird uns dazu eine durchgängig POLA-konforme Softwarearchitektur dienen, welche Applikationen und ihre inneren Module in ihren Rechten massiv einschränkt.

Shatter Attack

Diese spezielle Attacke wurde hier aufgenommen, weil sie einen gewissen Aha-Effekt bei manchen Entwicklern auslöst: Ein Eingabefeld in einem GUI ist nichts anderes als ein Guckloch in den Speicher einer Applikation – eigentlich selbstverständlich, schließlich müssen die Eingabedaten ja irgendwo abgelegt werden.

Unter Umständen kann das Eingabefeld aber dazu genutzt werden, Schadcode einzufüllen. Dieser Code landet dann in einem bestimmten Datenbereich der Applikation. Mithilfe eines Debuggers lässt sich die Adresse des Speicherbereichs leicht ermitteln. Wenn es nun möglich wäre, die Applikation dazu zu bringen, diesen Code zu starten, dann wäre die Übernahme perfekt. Genau diese Möglichkeit ergibt sich durch einen Designfehler im Windows API, der es anderen GUI-

Shatter Attack

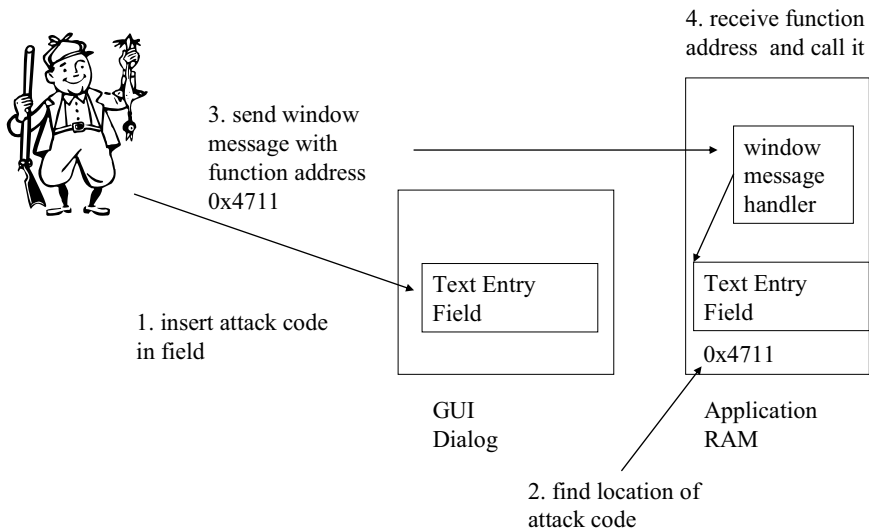


Abb. 2.15 Shatter Attack unter Windows

Applikationen gestattet, sogenannte Windows Events an beliebige andere GUI-Applikationen zu schicken. Darunter sind auch Events, die Callback-Adressen beinhalten, d. h. Verweise in den Speicherbereich der Applikation. Diese nimmt den Event entgegen und verzweigt an die angegebene Adresse (s. Abb. 2.15).

Microsoft hat bisher vergeblich versucht, diesen Fehler grundsätzlich zu beheben, da sich dadurch große Inkompatibilitäten für Windows-Applikationen ergeben würden. Es wurden lediglich Maßnahmen ergriffen, um z. B. zu prüfen, ob eine Callback-Adresse ursprünglich auch von der jeweiligen Applikation registriert wurde. Gänzlich gelang dies bisher jedoch nicht. Im Windows-Bereich gilt deshalb das grundsätzliche Verbot von GUIs für Services, die mit wichtigen Privilegien (meist System) laufen, um die Gefahr einer Übernahme gering zu halten. Insgesamt ist die Inter-Applikation-Kommunikation über Fenster in Windows ein großes Sicherheitsproblem, da es die Isolierung durch Prozesse über das GUI aufhebt.

UNIX Shellscript und Utility-Attacken

UNIX Shellscripts bieten eine große Zahl von Möglichkeiten, wie nicht privilegierte Benutzer durch das Ausnutzen von Symbolic-Link-Features oder durch Race-Conditions-Aktionen mit Root-Rechten ausführen können. Sie sind hier aufgeführt, weil sie zwei Basisprinzipien bzw. Gefahren für Software gut demonstrieren: Die Umwandlung von Referenzen auf Ressourcen und die Sicherheitslücke zwischen Prüfung einer Ressource und Zugriff (Time-Of-Check-To-

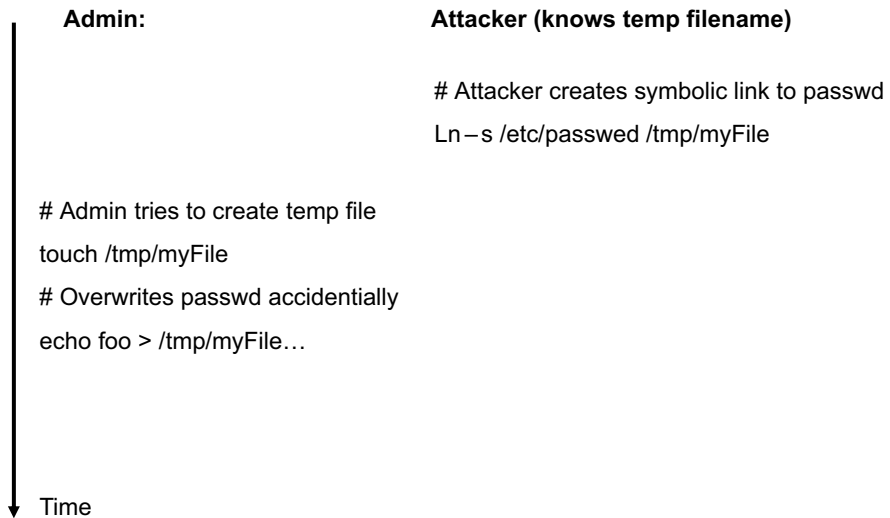


Abb. 2.16 Angriff auf Password-File durch UNIX Shellsript

Time-Of-Use, TOC2TOU). Abbildung 2.16 zeigt ein Beispiel eines solchen Angriffs, wie er so ähnlich von Dominik Vogt (s. [Vogt]) im Linux Magazin geschildert wurde.

Die Behandlung von Files durch Shellscripts ist im Allgemeinen nicht atomar. Das bedeutet, häufig wird eine Prüfung mittels `stat(Filename)` durchgeführt und anschließend wird das File geöffnet und geschrieben. Da diese Operationen nicht innerhalb einer Transaktion stattfinden, können Angreifer dazwischen die Files ändern. Gefährlich ist auch die Eigenschaft der Unix-Directory-Rechte, dass ein Schreibrecht auf diesem Metalevel das Recht beinhaltet, innerhalb der Datei Namensänderungen der Files durchzuführen, obwohl nur deren Eigentümer Lese- und Schreibrechte hätte (s. Abb. 2.17).

Vogt schlägt zur Absicherung dieser Probleme die Bibliothek Gate Guardian vor, die unter Anderem sichere temporäre Files anlegt und Filefunktionen auf Filedeskriptorbasis (wobei die richtigen Sicherheitsmodi gesetzt sind) verwendet. Erneut erweisen sich Capabilities (in diesem Fall in der Form von Filedeskriptoren) als sicherer Weg, Namen auf Ressourcen abzubilden.

Des Weiteren zeigt Vogt in wenig Code insgesamt sieben Sicherheitsprobleme in Unix auf, die sämtlich auf die beiden genannten Prinzipien zurückzuführen sind:

- Filerechte prüfen und erst anschließend öffnen und bearbeiten.
- Files öffnen ohne Prüfung, ob es sich um einen symbolischen Link handelt, der von jemand anderem auf ein wichtiges File gesetzt wurde.
- Temporäre Files öffnen, ohne obige Prüfung bzw. mit eigenem, konstanten Namen (ebenfalls Symlink-Problem).
- Files öffnen, ohne atomar gleichzeitig die Zugriffsrechte richtig zu setzen.

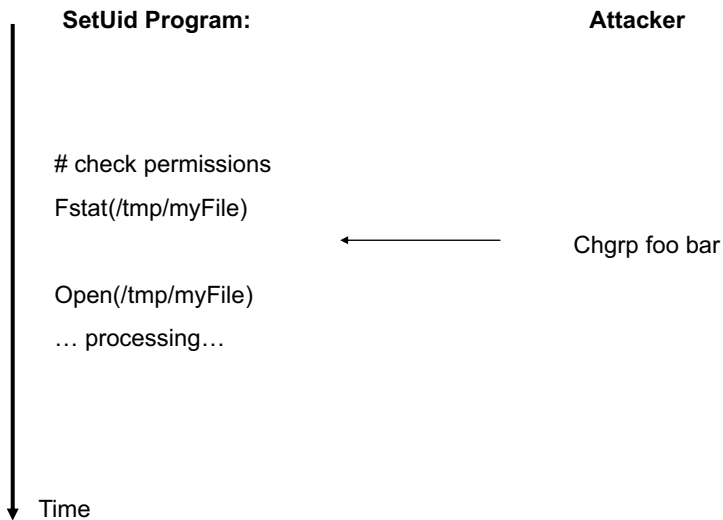


Abb. 2.17 Ausnutzen nicht-atomarer File-Zugriffe

- In unsicheren Verzeichnissen arbeiten, d. h. solche, bei deren Väterverzeichnis-
sen andere User Schreibrechte besitzen.
- Das Löschen von Dateien mit `unlink()` arbeitet auf Basis des Filenamens.
Hat ein Angreifer einen Hard-Link auf das File gesetzt (hierfür braucht er nur
Leserechte), so wird das File nicht gelöscht und wichtige Informationen bleiben
im System.
- Filenamen mit Leerzeichen werden von vielen Routinen in mehrere Filenamen
zerlegt (Shellvariablen z. B. müssen Hochkommata haben).

Shellscripts oder Utility-Programme sollen das Leben einfacher machen und
schnell arbeiten. Es verwundert daher nicht, dass in sehr vielen Scripts und Tools
diese Probleme zu finden sind. Die Sicherheitsmechanismen von Unix sind an
dieser Stelle undurchsichtig und gefährlich. Kleinste Fehler haben so fatale Kon-
sequenzen.

2.3 Grundlagen der Input-Validierung

Angesichts der immer noch dürftigen Lage im Bereich der Input-Validierung
wollen wir hier einige Konzepte und Werkzeuge vorstellen und auch Ideen für
eine grundsätzliche Verbesserung vorschlagen. Dieses Buch beschäftigt sich zwar
nicht mit Netzwerksicherheit und der Abwehr von Attacken auf diesem Level
durch Firewalls und Paketfilter, dennoch gibt es für Entwickler von Applikationen
einige Dinge in diesem Bereich, die Auswirkungen auf die Applikationsarchitek-
tur haben können. Deshalb sollten Entwickler Grundkenntnisse über das Filtern
von Requests sowie über gute und schlechte Eigenschaften von Applikationen in
Bezug auf die Filterung kennen.

2.3.1 Abwehr auf der Netzwerkschicht

Wie wir gesehen haben, laufen fast alle Attacks heutzutage auf der Anwendungsebene. Einer der Gründe dafür ist vielleicht die Qualität, die Firewalls und Paketfilter mittlerweile erreicht haben. Es lohnt sich daher, einmal einen kurzen Blick auf die Softwarearchitektur dieser Komponenten zu werfen. Wir tun dies am Beispiel des IPTables-Frameworks von Linux (s. auch [IPtables]). Anhand seiner Konstruktion und Arbeitsweise ergeben sich daraus auch Hinweise, wie eine gut zu filternde Applikation gebaut sein sollte.

Ein weiterer Grund für eine kurze Betrachtung von Firewalls sind die Auswirkungen auf die Softwarearchitektur, die manche Firewall oder VPN-Techniken mit sich bringen. Man denke nur an den Einsatz von Network Address Translation (NAT) oder Verschlüsselung von Datenpaketen. Wenn die Applikation auf die Idee kommt, Netzwerkinformationen wie IP-Adressen oder Ports in ihren Datenpaketen zu transportieren, dann ist sie schlagartig bei Einsatz der genannten Techniken nicht mehr Firewall-fähig. Und nicht zuletzt überschätzen viele Softwareentwickler die Möglichkeiten von Firewalls drastisch bzw. bauen Applikationsprotokolle, die sehr schlecht für die Überwachung in Firewalls geeignet sind. So schaffen es peer-to-peer-Protokolle wie z. B. Skype, eine direkte Kommunikation zwischen Partnern herzustellen, selbst wenn sich beide Partner hinter Firewalls befinden (s. Abb. 2.18).

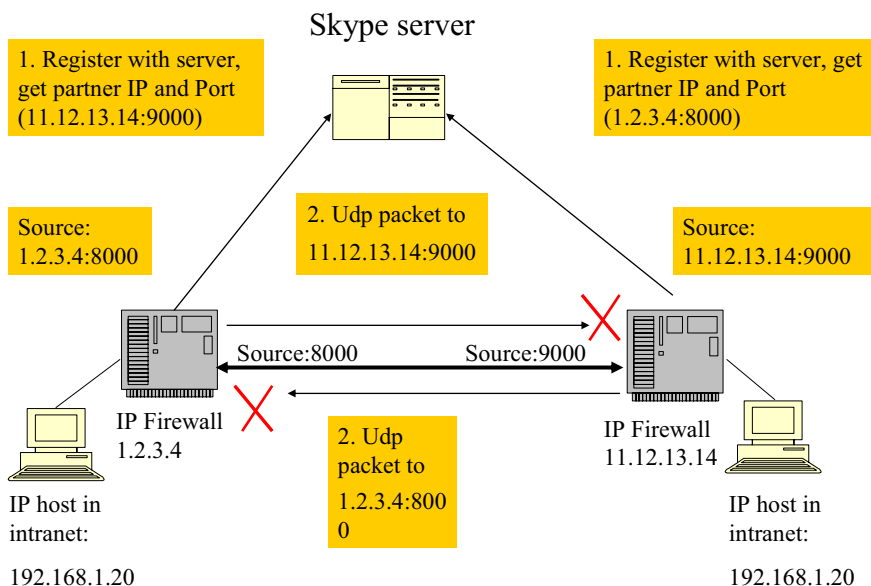


Abb. 2.18 Direkte Verbindung via Skype trotz Firewalls (nach [Schmidt])

Wir werden uns daher nun kurz mit den folgenden Themen beschäftigen:

- Was bedeutet eigentlich Filtern auf dem Netzwerklevel?
- Was sind schlechte Applikationsprotokolle im Bezug auf Filtering (Zuordnung Request/Response, Ports etc.)?
- Die Middleware-Problematik,
- verschlüsselte Inhalte und NAT-Fähigkeit,
- Content Filtering,
- Funktion der IPTables-Architektur als objektorientiertes Modell,
- Ende-zu-Ende-Sicherheit vs. Sicherheit auf der Netzwerkschicht und
- Korrektheit von IPTables Scripts.

Der eigentliche Filtervorgang lässt sich ganz grob so beschreiben (siehe Abb. 2.19): Es gibt ein Datenpaket, das gefiltert werden muss. In diesem Datenpaket finden sich sowohl Netzwerkparameter als auch Applikationsdaten. Weitere Netzwerkparameter existieren in der Umgebung des Filters. Zusätzlich hat der Filter ein Gedächtnis und kann sich z. B. vorausgegangene Datenpakete und deren Eigenschaften merken. Es gibt nun im Filter einen Regelsatz, der aus Parametern Muster bildet. Passt das jeweilige Muster einer Regel auf das aktuelle Datenpaket inklusive der Umgebung, so wird eine Aktion ausgelöst, die z. B. in der Annahme des Paketes resultieren kann. Andernfalls wird das Muster der nächsten Regel auf das Datenpaket angewendet. Passt kein Muster – d. h. hat keine Regel eine An-

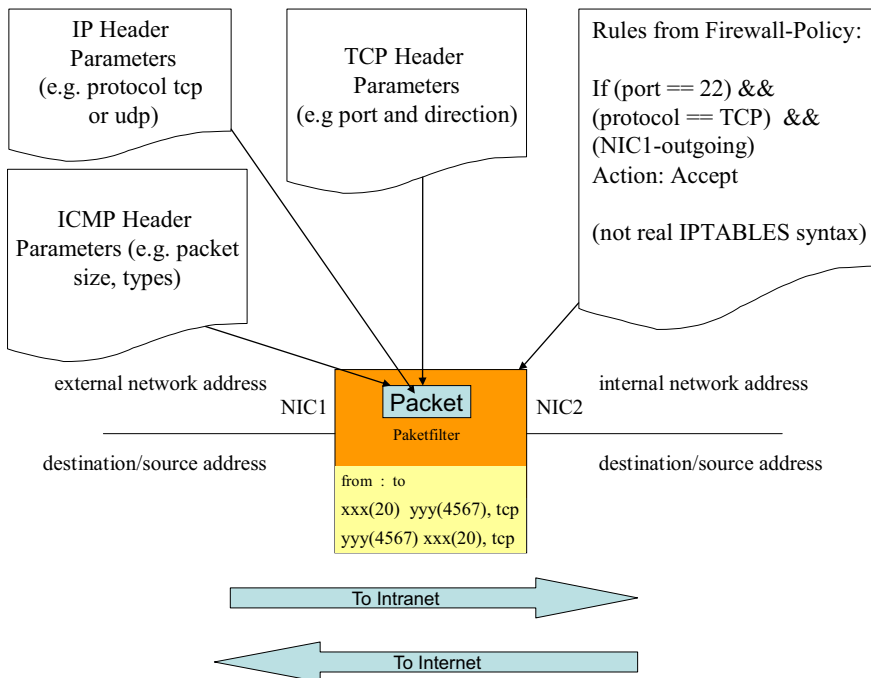


Abb. 2.19 Entscheidungsfindung eines Paketfilters

nahmeaktion ausgelöst, dann sollte das Paket aus Sicherheitsgründen verworfen werden (Default-is-deny-Policy). Es ist aber auch eine „Default-is-accept“-Policy denkbar, bei der alle Pakete, die nicht explizit durch eine Regel verboten sind, weiter geleitet werden.

Es sind nicht wirklich viele Paketinformationen, die von den Regeln zur Entscheidungsfindung verwendet werden können. Hier sind sie:

- Netzwerk Header (IP, TCP, ICMP mit Protokollen, Ports und Adressen),
- Netzwerkadressen,
- Richtungsinformation: Woher/Wohin?
- Parameter vorheriger Pakete (z. B. vorher ausgehend, jetzt reinkommend) und
- evtl. eine Inspektion des Nutzinhaltes des Paketes. Dieser ist jedoch Applikationsspezifisch und braucht eine Filtererweiterung, damit der Filter weiß, welche Daten wichtig sind bzw. was sie bedeuten. Oft wird die Aufgabe des Content-Filterns jedoch an spezielle Proxies ausgelagert, die als Prozesse auf dem Firewall laufen.

Wir wollen die Arbeitsweise und die Probleme des Filters kurz an einem Beispiel erklären. Nehmen wir an, der Firewall erhält am NIC, der mit dem Internet verbunden ist, ein Paket mit folgenden Eigenschaften: Protokoll UDP, Absenderport 4444, Zielport 2345. Soll dieses Paket den Firewall passieren dürfen? Diese Frage ist nicht leicht zu beantworten und erfordert einen Rückgriff auf die Firmenpolicy bezüglich solcher Pakete. Eventuell kann man aus den verwendeten Ports oder IP-Adressen auf eine bestimmte Applikation schliessen, die man erlauben möchte, oder man entschließt sich zu der fragwürdigen Haltung, alle UDP-Pakete zu akzeptieren.

Leichter tut sich der Firewall, wenn er ein Gedächtnis besitzt. Vielleicht ging dem Paket ein internes Paket voraus, das an die Zieladresse gerichtet war, von der jetzt das andere Paket kommt. Dann kann der Firewall es als Antwort ansehen und durchlassen. Besser ist es aber, wenn im Netzwerkprotokoll die Frage von Aufnahme der Verbindung und Antwort durch Header-Flags klar geregelt ist wie bei TCP. Daher ist auch die Bereitschaft, ein auf TCP beruhendes Protokoll durchzulassen, generell viel grösser als bei UDP. Hinzu kommt, dass TCP sogenannte *Sequence Numbers* zur Kennzeichnung der Pakete als Antwortpakete auf vorhergehende Pakete verwendet. Diese fehlen in UDP, was bestimmte Arten von Angriffen auf Netzwerkebene stark erleichtert.

Nehmen wir nun an, die Applikation ist so konstruiert, dass sie mehrere Ports und Protokolle verwendet, wie dies beispielsweise manche Multimedia-Streaming-Applikationen tun. Somit erhält der Firewall plötzlich Verbindungsanfragen aus dem Internet ins Intranet. Dies sehen die Sicherheitsverantwortlichen für das Intranet aus verständlichen Gründen gar nicht gerne und neigen dazu, solche Requests lediglich für ausgewählte eigene Server (z. B. den Firmen-Web-Server) zuzulassen. Die Problematik ist allerdings nicht ganz so schlimm, wenn die verwendeten Ports und Protokolle einer Applikation feststehen, z. B. ausgehend immer TCP-Port xx und hereinkommend immer UDP-Port yy. In dieser Situation könnten Regeln dafür sorgen, dass diese Ports immer offen sind und so das Funk-

tionieren der Applikation sicherstellen. Allerdings sollte klar sein, dass man sich mit solchen statischen Regeln immer auch Einfallstore in das eigene System öffnet: Niemand kann schließlich garantieren, dass die so geöffneten Ports nicht auch von anderen, bösartigen Anwendungen ausgenutzt werden.

Noch schwieriger wird es aber, wenn die Applikation mit mehreren Ports und Protokollen arbeitet, und zwar dynamisch, d. h. sie und ihre Partner einigen sich über Kommandos in den Nutzdaten, welche Ports und Protokolle sie momentan verwenden wollen. Jetzt hat der Firewall nur dann eine Chance, wenn er die Nutzdaten der Applikation verstehen kann – damit ist die nächste applikationsspezifische Erweiterung des Firewalls erforderlich. FTP ist ein Beispiel für ein Protokoll, das ein solches Modul benötigt.

Kennt der Firewall die Nutzdaten der Applikation nicht und schaltet NAT (Network Address Translation) ein, d. h. in alle ausgehenden Pakete wird die IP-Adresse/Port-Kombination des Firewalls eingetragen (wobei sich der Firewall den ursprünglichen Sender merkt), so erhalten die Empfänger der Pakete in den Paketen die Absenderadresse des Firewalls, in den Nutzdaten der Anwendung jedoch irgendwelche anderen IP-Adressen des Sendernetzwerkes – meist aus dem nicht gerouteten 192.168.x.y/-Bereich. Jeder Versuch, an diese Adressen etwas zurückzuschicken, scheitert natürlich, da sie der Firewall verwirft (er lässt bei NAT nur Pakete mit seiner Zieladresse durch, da er sie über seine interne Tabelle wieder einem internen Empfänger zuordnen kann).

Selbst wenn der Firewall über Zusatzmodule die internen Nutzdaten der Applikation verstehen und notfalls patchen kann – sobald die Applikation die Daten verschlüsselt, ist es auch damit vorbei und die Applikation scheitert am Firewall.

Nun haben wir bereits einige problematische Eigenschaften von Applikationen kennen gelernt:

- die Verwendung mehrerer Ports und Protokolle,
- dynamisches Aushandeln von Ports und Protokollen,
- Einbetten von Netzwerkdaten in Applikationsnutzdaten und
- Verschlüsseln von Netzwerkdaten in Applikationsnutzdaten.

Firewall-Architekten mögen jedoch auch noch andere Eigenschaften von Applikationen oder speziell Middleware sehr wenig. Dazu gehören:

- proprietäre Applikationsprotokolle (was macht diese Applikation mit dem Netzwerk?),
- überladene Applikationen, die gleichzeitig Client- und Server-Rollen übernehmen,
- Omnibus Middleware (die Bündelung von verschiedenen User-Sessions und Diensten über eine Verbindung erlaubt kein granulares Filtern),
- generell Middleware, da über die Definition von Remote Procedure Calls prinzipiell jegliche Funktionalität möglich ist (CORBA, DCOM, WebServices) und
- verschlüsselte Nutzdaten (keine Content-Filterung möglich).

Applikationsarchitekten sollten bei Web-Applikationen daran denken, dass innerhalb des Firewall-Komplexes clientseitige SSL-Verbindungen terminiert wer-

den, damit der Filter seine Aufgabe erfüllen kann. Dies könnte zu einer Sicherheitsproblematik in der Applikation führen, z. B. der Offenlegung von User Credentials.

Bei der Verwendung komplizierter dynamischer Port- und Protokollzuordnungen kann von den Firewall-Betreuern eine Proxy-Lösung verlangt werden, d. h. ein Teil der Applikation muss in die DMZ verlagert werden, um dort speziell abgesichert zu werden bzw. um von dort weniger Zugriff aufs Intranet zu erhalten. Existiert für diese Applikation kein solcher Proxy, dann können ihre Daten nicht durch einen Bastion-Host mit getrennten Netzwerkkarten durchgereicht werden.

Zum Schluss behandeln wir am Beispiel von IPTables die Architektur eines Filter-Frameworks. Die Syntax der Regelfiles ist stark an den Vorgänger IPChains angelehnt und beinhaltet die Netzwerkelemente, Kommandos für die Positionierung von Regeln bzw. die Bearbeitung von ganzen Regelketten sowie die Aktionen, die durchgeführt werden sollen, wenn ein Muster auf ein Datenpaket passt. Abbildung 2.20 zeigt Beispiele für die Syntax.

Wie können solche Regelfiles, die letztlich den Firewall ausmachen und die entsprechende Firmenpolitik repräsentieren, im Kernel implementiert werden?

Abbildung 2.21 zeigt den Fluss von Datenpaketen durch das Netfilter Framework. An den mit NF_IP_xxxx gekennzeichneten Stellen sind diese Datenpakete für eine eventuelle Bearbeitung (Mangle) oder eine Prüfung (Filtering) zugänglich. Der grobe Ablauf sieht so aus: Ein hereinkommendes Datenpaket landet in der Pre-routing Queue (NF_IP_PRE_ROUTING). Dort kann es bearbeitet werden, z. B. kann die Zieladresse via NAT geändert werden. Anschließend geht das Paket in eine Routing-Verarbeitung. Dort wird entschieden, welchen Verlauf das Paket im Firewall nehmen wird. Ist es an den Firewall selbst gerichtet (z. B. eine SSH-Verbindung zur Verwaltung des Firewalls selbst), so gelangt es in NF_IP_LOCAL_IN (Input Chain) und kann dort gefiltert, aber nicht mehr modifiziert werden.

iptables -t table -command [chain] [match] -j [target/jump]

Example:

- `iptables -T FILTER -A INPUT -i $IFACE -p tcp --sport 80 --state ESTABLISHED -j ACCEPT` (allow incoming web traffic if it belongs to a previous outgoing request)
- `iptables -A INPUT -i $IFACE -p tcp --sport 20 --state ESTABLISHED, RELATED -j ACCEPT` (allow incoming ACTIVE ftp traffic if it belongs to a previous outgoing request, even though the incoming request is for a new – but related - port)
- `iptables -A INPUT -i $IFACE -p udp -j LOG --log-prefix „UDP Incoming:“`
- `iptables -A INPUT -i $IFACE -p udp -j DROP` (log and drop all udp traffic)

Abb. 2.20 Beispielregeln für IPTables

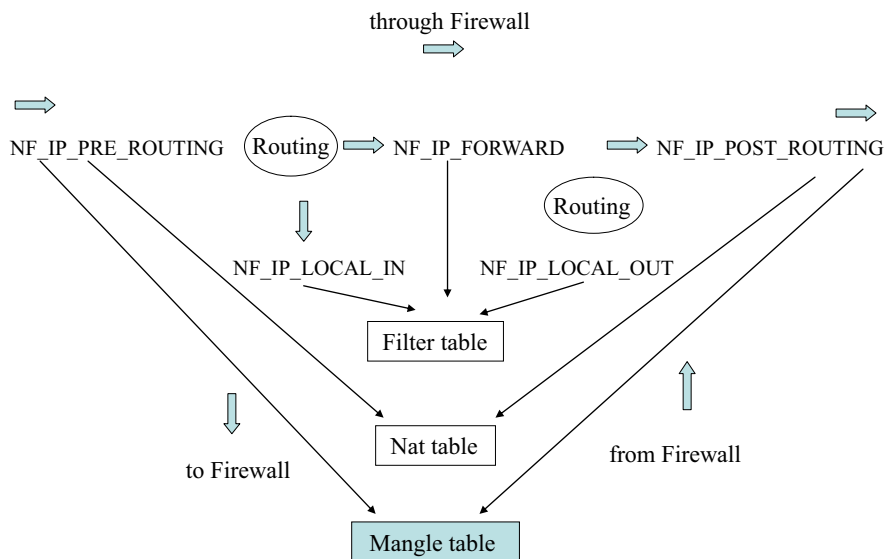


Abb. 2.21 Fluss von Paketen durch das Netfilter Framework

Ist das Paket jedoch nicht für den Firewall selbst bestimmt, sondern für das Internet oder das Intranet, so landet es in `NF_IP_Forward` (Forward Chain). Dort kann es wiederum nur gefiltert, aber nicht modifiziert werden. Anschließend landet es in `NF_IP_POST_ROUTING`. Dort kann es wieder modifiziert werden. Beispielsweise kann dort bei Einsatz von NAT die IP-Adresse des Firewalls anstatt der ursprünglichen Senderadresse im Intranet eingetragen werden.

Hier sieht man den entscheidenden Vorteil der klaren Trennung von Modifikation und Filterung: Die Entscheidung, ob ein Request aus dem Intranet (z. B. Host 192.168.1.24) auf das Internet hinaus darf, kann ja nur anhand der IP-Adresse erfolgen. Würde jetzt frühzeitig schon die IP-Adresse des Firewalls dort eingetragen, wäre die Filterregel, die es z. B. nur Hosts mit einer Adresse größer 100 erlaubt, in das Internet zu gehen, sinnlos geworden. Die richtige Reihenfolge von Modifikation und Filterung bestimmt also das Ergebnis.

Wie kann nun ein Datenpaket modifiziert oder ausgefiltert werden? Die Stellen mit `NF_IF_xxxx` stellen Callback-Schnittstellen dar, an denen sich interessierte Module registrieren können. Diese Module werden „Tables“ genannt. Grob gesagt gibt es drei verschiedene Funktionseinheiten: Filtern, Mangeln, NAT. Es gilt dabei die Regelung, dass nur Mangle und NAT Pakete ändern dürfen (denn nur sie registrieren sich dann auch für den Callback an der `PRE_ROUTING` bzw. `POST_ROUTING` Stelle).

Das Netfilter-Framework im Linux Kernel ist natürlich in C implementiert. Es lohnt sich dennoch, ein OO-Analysemodell dafür zu entwickeln, da sich so die verwendeten Patterns besser ausdrücken lassen. Vor allem das Template/Hook-Pattern als Basismuster von Framework-Architekturen und das Factory-Pattern für die dynamische Erzeugung von Objekten kommen hier zum Einsatz. Das Frame-

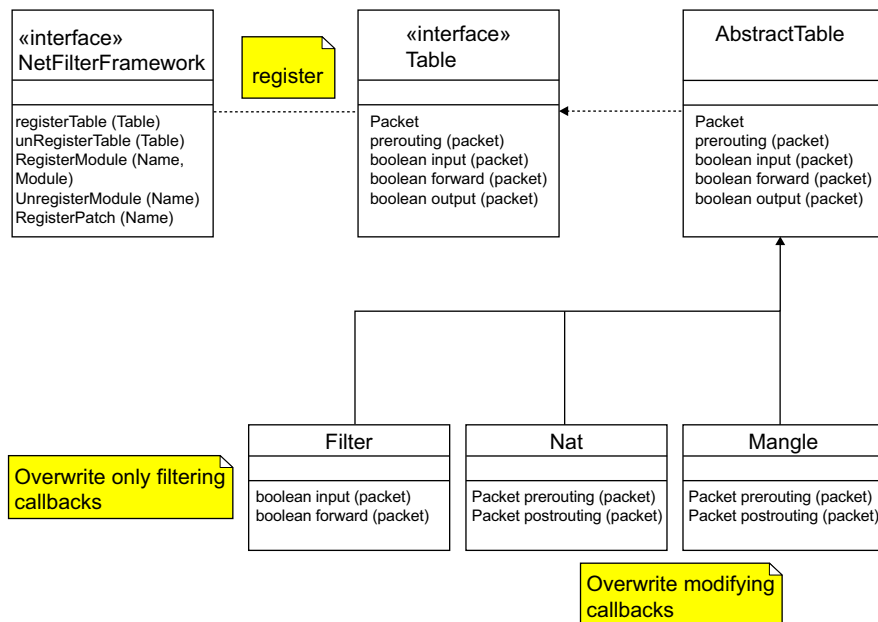


Abb 2.22 UML-Diagramm des NetFilter Frameworks

work selbst ist durch eine Reihe von Codemodulen zusätzlich erweiterbar. Diese Module schließen die Lücke zwischen der textuellen Beschreibung der Regeln und ihren Elementen (die Muster im Datenpaket) und der tatsächlichen Implementation von Mustererkennung im Code.

Im IPTables-Framework besteht, wie oben beschrieben, eine strikte Trennung zwischen modifizierenden und rein filternden Arbeitsschritten. Beide sind zeitlich genau definiert.

In Abb. 2.22 wird diese Trennung durch die unterschiedlichen Returnwerte der Callback-Funktionen ausgedrückt, die entweder nur eine Entscheidung mitteilen (verwerfen oder weitermachen), oder aber ein Datenpaket zurückgeben, das evtl. geänderte Daten beinhaltet. Die Modifikation ist nur in den Arbeitsschritten (Tables) Mangle und NAT erlaubt, Entscheidungen über Weitermachen oder Wegwerfen hingegen nur in der Filter-Table. Somit sorgt das Framework für klare Reihenfolgen von Modifikation und Filtern – ein Problem, das den Vorgänger IPChains dauernd belastet hat.

Entscheidend ist auch die Kombination einzelner Regeln in *Ketten* (Chains) sowie der Aufbau einzelner Regeln aus Kommandos, Aktionen (Targets) und sogenannten *Matches*. Matches sind spezielle Token, für die es eine Filtererweiterung im Framework selbst braucht. Diese kann z. B. mit `registerMatch (match)` registriert werden. Unter Linux gibt es ein Framework namens „Patch-O-Matic“, mit dem solche Matches hergestellt werden können. Sie sind der Code, der die Anweisungen in der textuellen Regel tatsächlich durchführt.

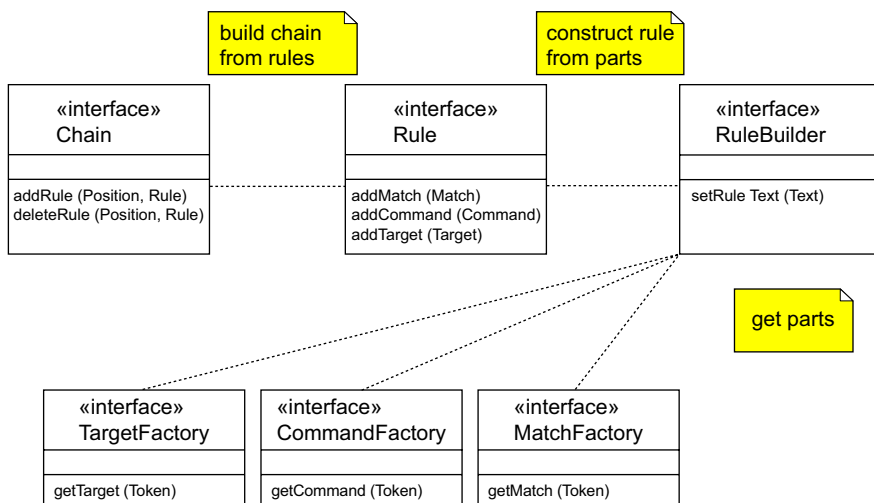


Abb. 2.23 UML-Diagramm für die Umsetzung von Filterregeln in Aktionen

So ist z. B. der Regelteil `...-m state state=established` eine Anweisung an das state-Modul zu prüfen, ob der Status der Verbindung „etabliert“ ist, d. h. der Request im Datenpaket muss eine Antwort auf eine vorherige Anfrage sein. Das Framework, das die Umsetzung der Regeltexte in Aktionen durchführt, könnte ungefähr wie in Abb. 2.23 dargestellt aussehen.

Abschließend lässt sich feststellen, dass das Netfilter-Framework und seine Steuerung durch IPTables sowohl die Performance als auch die Korrektheit des Filterns unterstützen und gleichzeitig extrem erweiterbar sind durch die Callback-Schnittstellen. Durch weitere Netzwerkkarten lassen sich damit sehr einfach Demilitarized Zones definieren, in denen dann Wireless Access Points für das Home-Netzwerk oder auch extern zugängliche Web-Server oder Mailserver positioniert werden können (s. Abb. 2.24).

Das obige Beispiel aus Rusty-Harolds-IPChains-Howto lässt sich in IPTables leicht durch sogenannte *user-defined Chains* implementieren. Der Datenverkehr wird in die sechs Grundrichtungen aufgeteilt (Intranet-Internet, Internet-Intranet, Intranet-DMZ, DMZ-Intranet, DMZ-Internet, Internet-DMZ) und anschließend die wenigen Regeln auf diese Filterketten verteilt.

Trotz aller Gliederung neigen aber Firewall-Skripts schnell zur Unübersichtlichkeit. Aufgrund ihrer Funktionsweise sind sie überdies stark von der Reihenfolge der Regeln abhängig. Kleinste Änderungen können hier zu dramatischen Effekten führen. Die Grundregel ist aber, dass am Anfang einer Filterkette die speziellen Regeln stehen müssen und am Ende die allgemeinen. Würde man diese Reihenfolge umkehren, so kämen die speziellen Regeln nie zum Einsatz, da die allgemeinen Regeln bereits alle Muster in den Datenpaketen abdecken würden. Insofern ist es in der Realität nicht so leicht herauszufinden, was ein Regelsatz nun tatsächlich bewirkt. Es gibt dazu Hilfsmittel in IPTables selbst (der `-C`-Parameter),

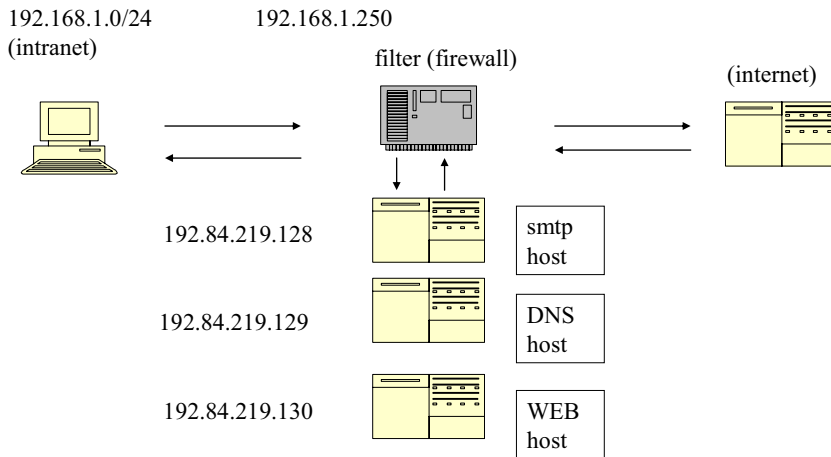


Abb. 2.24 Erzeugung einer DMZ mit smtp-, DNS- und Web-Server

mit denen Dummypakete, die gewissen Mustern einer Regel entsprechen, automatisch erzeugt und dem Filterframework vorgesetzt werden. Über den `-L`-Parameter können die gesamten erlaubten Verbindungen angezeigt werden.

Dennoch besteht ein tiefer semantischer Graben zwischen den allgemein formulierten Policies einer Firma in der Form „Web-Zugriffe werden erlaubt, Mail auch, SSH nur ausgehend“ etc. und dem konkreten, in IPTables-Syntax formulierten Regelsatz. Hier wäre es schön, eine domänenspezifische Sprache zu haben, mit der die High-Level-Policies formuliert werden könnten. Daraus ließen sich entweder sofort Regelsätze generieren oder aber eine Überprüfung der manuell erzeugten Regelsätze durchführen (modellbasiert im Stile eines Model-Checkers oder durch die Erzeugung von Testpaketen). Die Modellierung von Sicherheitsaspekten ist jedoch momentan noch nicht perfekt. Die folgenden Merkmale wären eine große Erleichterung für komplexe Firewall-Zonen:

- Beweis der Correctness – wird Input, der nicht den Requirements entspricht, geblockt?
- Beweis der Liveness – sind die Requirements in Bezug auf die Durchlässigkeit der Firewall für die nötigen Services erfüllt?
- Beweis von Correctness und Liveness über mehrere Firewall-Konfigurationen hinweg.
- Wie wirkt sich das Mixen von ACCEPT- und REJECT-Anweisungen für das Model-Checking aus?

Betrachtet man das Filtern von Requests aus einem etwas größeren Abstand, dann erscheint es als spezieller Fall von sog. Complex Event Processing (CPE). Der Begriff sowie die Methodik wurde ursprünglich von David Luckham an der Stanford University geprägt und zielt auf die Verarbeitung von Events durch Regeln, die in einer Event Processing Language formuliert wurden [Luck]. Solche Sprachen erlauben praktisch die beliebige Kombination von Events nach Zeit oder

Art des Auftretens. Auch Intrusion-Detection-Systeme fallen letztlich darunter. CEP-Systeme stellen im Grunde Datenbanken auf den Kopf: Es gibt fixe Queries, die abgespeichert werden (die Muster, nach denen im Input gesucht wird). Die Daten werden anschließend durch die gespeicherten Queries geschickt. Passt das Datenmuster zum Query-Muster, wird ein Event ausgelöst. Mehrere solcher Events können wiederum Events auf höheren Ebenen auslösen. Ein Beispiel für ein CEP-Framework auf Java Basis ist das frei verfügbare Esper ([Esp]).

2.3.2 Input-/Output-Validierung auf Applikationsebene

Die Bedeutung der Input-Validierung für Web-Applikationen steht im krassen Gegensatz zu den verfügbaren Software-Frameworks, die diesen Punkt systematisch abdecken würden. Im Folgenden besprechen wir zunächst einen Blacklist-basierten Ansatz und machen dann einen Vorschlag für ein generelles Input-Validation-Framework, das auf Whitelists beruht, genauer gesagt, auf Grammatiken für die Kommunikationsprotokolle. Als letzten Teil besprechen wir `mod_security` – ein Framework für die externe Validierung von Web-Services auf XML/SOAP-Basis, das einen neuen Typ der Validierung darstellt: die Web-Application Firewall (WAF). Eine WAF stellt u. U. die einzige Chance dar, schnell und sicher auf gefundene Schwächen vor allem in Legacy-Applikationen zu reagieren, und wir werden ihre Einsatzmöglichkeiten und Fähigkeiten untersuchen.

Die Output-Validierung ist eine zusätzliche Maßnahme, welche die „Defense-in-Depth“ verstärkt. Normalerweise dürften Datenfelder im Output der Applikation keine Sonderzeichen wie die HTML-Zeichen „<“, „>“ und „&“ beinhalten. Diese können durch einen Filter leicht durch ihre entsprechenden XHTML-Entity-Referenzen ersetzt werden. Es könnte der Gedanke aufkommen, dass es nur Output-Validierung braucht, um XSS-Attacken zu vermeiden. Dies würde nur dann gelten, wenn im Rahmen der Output-Validierung nicht nur das übliche Encoding von HTML-Zeichen als Entities stattfinden würde (aus „>“ wird `>` oder der entsprechende numerische Code beginnend mit einem `#`-Zeichen). Stattdessen müssten Inhalte von Variablen einer kompletten Typ- und Inhaltsprüfung unterzogen werden, denn es gibt durchaus gefährliche Inhalte, die keine spitzen Klammern benötigen. Beispiele für solche Attacken finden sich beispielsweise auf ha.ckers.org. Dort findet sich z. B. die Einbettung von bösartigen Statements in eigenes Javascript. Im Allgemeinen tut man jedoch gut daran, eine komplette Input-Validierung mit zusätzlichem Output-Encoding durchzuführen.

Gefahr durch Sanitizing und Filterketten

Bevor wir auf die verschiedenen Prinzipien des Filterns von Input eingehen, müssen zwei grundsätzliche Probleme beim Umgang mit Input-Daten erwähnt werden. Das erste betrifft die Frage, ob Sanitizing – also die Reparatur von Input, bei dem gefährliche Metadaten entdeckt wurden – überhaupt erlaubt sein soll.

Es lässt sich schnell zeigen, dass z. B. durch die automatische Entfernung von Metazeichen im Filter eine Attacke erst richtig „scharf“ gemacht werden kann. Das Prinzip dahinter wird als Double-Decoding bezeichnet und lässt sich am folgenden Inputstring leicht demonstrieren:

```
...//filename
```

Der Sanitizing-Prozess ändert den ersten Slash in einen Backslash und filtert anschließend die Kombination „..`\`“ aus dem Inputstring heraus. Übrig bleibt „..`/filename`“ – die Attacke zum Verlassen des Pfades ist erst aufgrund der Filterung und Änderung entstanden.

Deshalb ist es eine sinnvolle Regel, festzulegen, dass beim Auftauchen verdächtiger Zeichen beim Filtern der Inputstring nicht abgeändert („entschärft“) wird, sondern dass der gesamte Request abgelehnt wird.

Das zweite Grundproblem bei der Input-Validierung ist das Verhalten von Filterketten. Diese Filterketten entstehen dadurch, dass ein Request innerhalb einer Infrastruktur von verschiedenen Komponenten gefiltert werden kann. Dadurch entsteht die Möglichkeit des Double-Decodings, wie sie oben geschildert wurde, plötzlich nicht nur innerhalb eines Filters, sondern zwischen unabhängigen Komponenten. Ein Angreifer, der die Reihenfolge der Filter kennt, kann den Input so bauen, dass jede Komponente mithilft, den Angriff gegen eine weiter hinten gelagerte Komponente erst zu ermöglichen, indem sie Zeichen entfernt oder ändert.

Grundsätzlich kann eine Komponente nicht wissen, welche Validierungsbedürfnisse andere Komponenten haben – zumal sich diese im Laufe der Zeit auch ändern können, z. B. durch andere Back-End-Systeme oder Interpreter. Die Regel muss also lauten, dass jede Komponente für sich filtert, den Inputstring aber unverändert an weiter hinten befindliche Komponenten weitergibt (wenn der Request nicht gänzlich abgewiesen wird).

Blacklist-basiertes Filtern

Ted Neward beschreibt in [New04] ein kleines Framework für das Filtern von Input-Requests. Hier stellen wir eine etwas abgeänderte Version ohne Rücksicht auf Erweiterbarkeit, Optimierung oder Fehlerbehandlung vor (s. Abb. 2.25). Das Framework stellt einen Tainting-Mechanismus dar, der sicherstellt, dass der Wert des Strings nicht bezogen werden kann, ohne dass die Prüfungen abgelaufen sind. Unmittelbar bei Empfang eines Input-Wertes wird dieser in eine Variable vom Typ `TaintedXX` verpackt.

Wir werden im Zusammenhang mit der Browser Security die Technik des „Tainting“ („verderben“) nochmals kennen lernen. In einem vollen Ausbauzustand hätte das Framework Methoden, die nur `TaintedXX`-Typen als Parameter annehmen würden, und andere Methoden, die stattdessen Strings akzeptieren würden. Entscheidend ist dabei natürlich, dass die `TaintedXX`-Typen nicht von der Klasse `String` abgeleitet sind.

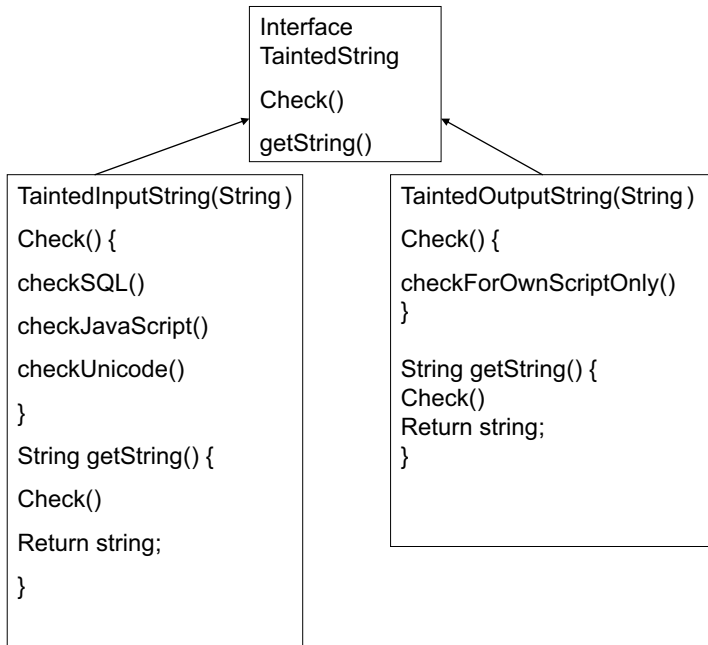


Abb. 2.25 Framework für Tainted Strings

Diese Architektur erlaubt sicher, mit noch unvalidierten Variablen umzugehen und sie sogar weiterzugeben etc. Dadurch, dass sie nur durch spezielle Methoden, die den Typ TaintedXX akzeptieren, bearbeitet werden können, ist aber sichergestellt, dass diese Variablen nicht an Interpreter weitergegeben werden. Dies ist eine sehr feingranulare und mächtige Methode der Zugriffskontrolle und basiert letztlich auf dem Labelling-Prinzip, das wir bereits im Zusammenhang mit Content Level Security in „Internet-Security aus Software-Sicht“ [KS] vorgestellt haben.

2.3.3 Forms-Validierung

HTML-Forms sind die Basis der Geschäftsprozesse des Webs. Selten zeigt sich der Zusammenhang zwischen Sicherheitsanforderungen und Anforderungen der Applikationslogik bzw. Geschäftslogik der Firmen so eng wie in der Behandlung von Forms. So beruht ein großer Teil des Kundenfeedbacks meist auf dem Ausfüllen von HTML-Forms – gleichzeitig bereiten diese Forms große Sicherheitsprobleme für die Web-Applikationen. Der Grund liegt meist in der geringen Strukturierung des Inputs (vielleicht einfache Textfelder mit beliebigem erlaubtem Input). Dabei müsste eigentlich allen Beteiligten klar sein, dass die Felder und die Struktur einer Form direkt mit einem möglicherweise automatisierbaren Prozess innerhalb der Firma zusammenhängen. Eine Feedbackform ist nicht „publishing“ allein.

Sie ist gleichzeitig Input für die zentrale Datenverwaltung der Firma und sollte daher genau definiert sein, am Besten durch eine detaillierte Schemadefinition, die Art und Inhalt der Felder und ihre Kombination genau festlegt.

Spontan entwickelte Forms neigen dazu, große Lücken für XSS-Attacken zu öffnen. Daher sollte ein wohldefiniertes Set von Feldern und deren Attributen sowie von definierten Filtern/Prüfroutinen dazu existieren. Wildwuchs von Forms bzw. deren Feldern in Firmen ist nicht nur Zeichen von mangelnder Sicherheit, sondern zeigt auch, dass Front-End-Systeme und zentrale Geschäftsprozesse nicht gekoppelt sind. Meist müssen dann Personen die Daten der Forms entnehmen und manuell bearbeiten sowie in die Schemata der Firmendaten umsetzen.

Bei der Validierung der Forms, z. B. in Servlet-Filtern, sollten die Metadaten der Form herangezogen werden. Illegale Kombinationen von Feldern, fehlende Felder etc. werden dann sofort und automatisch erkannt und abgewiesen. Der alleinige Blick auf die Struktur einzelner Felder ist zu wenig.

Natürlich spielt auch die Informationsarchitektur eine entscheidende Rolle. Sie definiert den Umfang der Forms, mögliche Felder und wie variabel oder detailliert der Inhalt jeweils definiert ist. Eine Explosion der zur Verfügung stehenden Forms ist wiederum ein Zeichen für mangelnde Definition des Kundenfeedbacks.

Im Idealfall verfügt die Firma über ein Repository zur Verfügung stehender Form- und Feldtypen. Mitarbeiter, die Forms einsetzen wollen, werden durch Tools bei der Auswahl bzw. der Anpassung existierender Forms und Felder unterstützt, wodurch der Wildwuchs vermieden wird. Gleichzeitig erhalten die Filterfunktionen der Applikationen die Metadaten der Forms zur weiteren Verarbeitung oder zum Filtern des Inputs. Somit zeigt sich, dass die effiziente Behandlung von Input von Kundenseite auch die sicherste ist, da sie automatisch das Whitelisting ermöglicht.

Whitelist-basiertes Filtern am Beispiel `mod_security`

Die oben vorgestellte Technik des Filterns oder Validierens mithilfe von tainted Variables basiert auf Blacklists potenziell gefährlicher Input-Strings. Dieses Prinzip widerspricht allerdings dem Sicherheitsprinzips des „Default-is-Deny“.

Auf der anderen Seite benötigt ein Whitelist-basiertes Filtern eine genaue Beschreibung dessen, was erlaubt sein soll, und nur dieses wird erkannt. Anderer Inhalt wird automatisch als fehlerhaft abgewiesen. Beispielsweise könnte eine Grammatik des Kommunikationsprotokolls der Applikation als Basis für whitelist-filtering dienen. Dies hätte den großen Vorteil, dass die Grammatik an intelligente Proxies in vorgelagerten Firewalls ausgeliefert werden könnte und dort zusätzliches Filtern stattfinden könnte. Außerdem zwingt es die Applikationsentwickler zu einer genauen Definition ihrer Interfaces nach außen. Das Whitelist Framework enthielte einen Parser der entsprechenden Grammatik und ein Parse-Error wäre gleichbedeutend mit der Verwerfung des Inhalts und der Erzeugung einer Fehlermeldung. Die Sorge, dass diese Form der Filterung zu einem endlosen Parsing-Prozess führen könnte, kann durch Stoppvariablen im Parser, wie sie noch von alten SGML-Parsern bekannt sind, gelindert werden.

Als Beispiel für das Whitelist-basierte Filtern betrachten wir nun den Filter `mod_security` für den Apache-Web-Server vor. Web-Services stellen ein besonderes Problem für Firewalls dar: Sie sind in ihrer Mächtigkeit äquivalent zu Middleware-Protokollen wie CORBA oder DCOM, werden jedoch meist über die ohnehin offenen Ports 80 oder 443 und das `http`-Protokoll transportiert. Dies hat die Folge, dass solche Requests von den meisten Firewalls nicht gefiltert werden (falls nicht ein spezieller XML-Firewall verwendet wird). `mod_security` ist ein Filter für den Apache-Web-Server, der vier der wichtigsten Angriffsmöglichkeiten auf Web-Services abdeckt. Hierbei handelt es sich nach [Rist] um:

- Falsche Input-Längen der Variablen,
- Variablen, die falsche Zeichen oder Meta-Zeichen enthalten,
- Variablen, die SQL-Kommandos enthalten und
- Antworten, die SOAP-Fehlercodes verraten.

Das `mod_security` Framework versteht SOAP-Requests und ist in der Lage, den Input durch reguläre Ausdrücke zu prüfen. Dafür sind einige weitere Funktionen des Filter-Frameworks nötig, speziell die Kanonisierung des Inputs vor der Anwendung von Regeln: In Abb. 2.26 wird eine Input-Variable `Number` darauf überprüft, ob sie eine bis neun numerische Stellen besitzt. `mod_security` erlaubt es, die Fehlermeldung bzw. ihre Nummer zu konfigurieren und so das Durchsickern von SOAP-Fehlercodes zu verhindern.

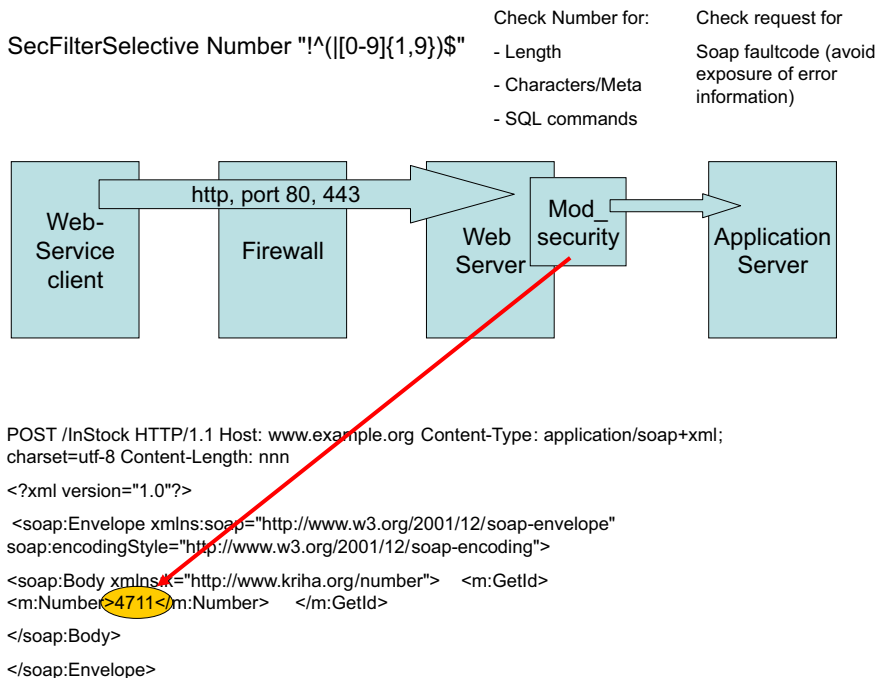


Abb. 2.26 Beispiel-Filter eines SOAP-Requests mit `mod_security`

Weitere Beispiele und Informationen zur Performance finden sich auf der Homepage von `mod_security`: <http://www.modsecurity.org/>. `mod_security` erlaubt es, eine Input-Validierung bereits vor der eigentlichen Applikation durchzuführen. Dadurch wird die Validierung innerhalb der Applikation natürlich keinesfalls ersetzt, es entsteht jedoch eine bessere Sicherheit durch „defense in depth“. Ein Nebeneffekt solcher vorgezogener Filter ist ein Zwang für die Entwickler, ihre Interfaces exakt zu beschreiben. Im Fall von Web-Services geschieht dies ohnehin durch WSDL. Deshalb bleibt zu überlegen, ob es nicht ausreichen sollte, wenn das Filter-Framework die WSDL-Interface-Beschreibung der Applikation einlesen könnte. Natürlich müssten die Variablen dann genauer in ihrer Art bestimmt werden, wie dies durch Pattern-Regeln z. B. in XML-Schemata möglich ist. Länge, erlaubte Zeichen des Vokabulars und sogar bestimmte Muster des Aufbaus können so leicht festgelegt werden. Schwieriger ist die Lage bei kritischen Wörtern wie „select“, die natürlich auch in Textfeldern vorkommen können und dann versehentlich gefiltert würden.

Nicht erfassbar durch `mod_security` sind natürlich die Gefahren des Protokolls selbst: Was wird eigentlich durch die jeweiligen Requests serverseitig ausgelöst bzw. welche Daten werden als Antworten zurückgegeben? Da Request und Response im Rahmen von Web-Services beliebig festgelegt werden können, bleibt hier nur die Inspektion der Interfaces bzw. der Applikation übrig.

Output Encoding am Beispiel Search Engine Security

Meist wird im Zusammenhang mit Validierung nur von Input-Validierung gesprochen. Wie bereits gezeigt, fällt diese am leichtesten, je genauer der mögliche Input für die Applikation spezifiziert werden kann. Daten, die jedoch aus den eigenen Datenbanken kommen, werden üblicherweise als „trusted“ angesehen, unter der Annahme, dass die Input-Validierung (falls die Daten von Benutzern kamen) richtig funktioniert hat. Diese Annahme entspricht bereits nicht dem Prinzip des „Defense-in-Depth“. Es gibt jedoch Applikationen, die mit der Einschränkung des möglichen Inputs ohnehin große Probleme haben und zwar nicht aufgrund schlechten Designs des User-Interfaces. Es handelt sich hier z. B. um Search Engines. Jeder Versuch, den möglichen Input einzuschränken, führt schnell zu Problemen bei der Benutzung. So sollten User in der Lage sein, nach „<“ oder „>“ in jedem beliebigen Kontext zu suchen. Die Suche nach „<script>“ bringt jedoch ganz andere Ergebnisse. Suchfelder in Applikationen haben daher eine hohe Fehlerrate in Bezug auf XSS-Schwächen. Dazu kommt, dass diese Suchfunktionen ja meist Teil der Applikationsdomain sind und daher Zugriff auf Session-Cookies etc. der Applikation selbst besitzen, die im Fehlerfall kompromittiert werden können.

Ganz besonders interessant wird die Frage der Validierung, wenn die Applikation selbst eine Search Engine ist. Abbildung 2.27 zeigt eine zweistufige Search Engine, bestehend aus einer Front-End-Applikation, die die Queries annimmt und einer Query/Response Engine als Teil des Back-Ends, das die Query letztlich

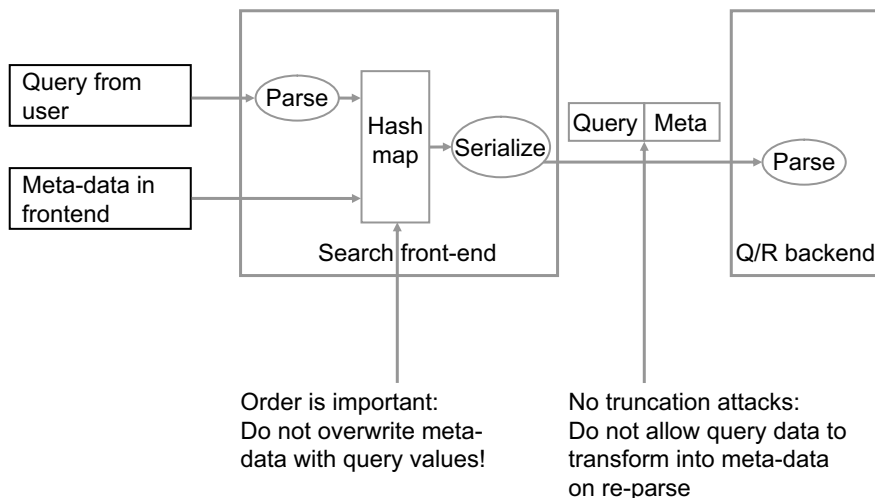


Abb. 2.27 Search Engine Input Processing

verarbeitet und aus dem Index eine Antwort erzeugt. Auffallend ist, dass die Query mehrfach geparkt wird – eine nicht ganz ungefährliche Aktion, wenn Attacken Fehler oder Missverständnisse bei der Erzeugung der Queries bzw. deren erneutem Parsen ausnützen können (siehe dazu auch den Abschnitt zur Applikationssicherheit am Beispiel von gmail im Kap. 4).

Abbildung 2.27 zeigt zwei mögliche Schwachstellen. Die erste besteht darin, dass die vom User kommenden Daten – der sog. Query-String – mit Meta-Daten der Search Engine gemischt werden. Eine mögliche Implementation davon ist eine simple Hashmap aus Namen und Werten. Der Query-String vom User wird beim Parsen aufgebrochen und in Name/Value-Paare getrennt, die anschließend in die Hashmap eingefügt werden. Daran anschließend wird mit den Meta-Daten des Front-Ends genauso verfahren. Das kann z. B. ein User-Token aus einer SSO-Authentisierung sein, das so an das Back-End weitergegeben wird.

Geschieht dies nicht in der richtigen Reihenfolge, dann besteht durch das gemeinsame Halten von Daten und Meta-Daten in einer Map die Gefahr, dass User-Daten die Meta-Daten überschreiben können. In dem Beispiel könnte dann der User durch einen geschickt gewählten Query-String eine UserID für das Back-End frei bestimmen.

Das mag zunächst nicht besonders wahrscheinlich klingen – wer würde denn nicht sofort erkennen, dass hier die Reihenfolge eine große Rolle für die Sicherheit spielt? Diese klare Erkenntnis mag bei den ersten Entwicklern der Applikation durchaus vorhanden sein, aber ist sie es auch bei neuen Teammitgliedern, die später dazu stoßen, ohne die anfänglichen Diskussionen um die Architektur und die Sicherheit mitgemacht zu haben?

Wir halten fest, dass das gemeinsame Halten von User-Daten und eigenen Meta-Daten in einer Datenstruktur mit anschließender Serialisierung und erneutem Par-

sen keine glückliche Konstruktion ist. Denn sie birgt noch eine zusätzliche Gefahr: Bei der Serialisierung der MAP z. B. in eine URL, um damit per http die Query plus Meta-Daten an den Query/Response-Server zu schicken, muss ein Trennzeichen verwendet werden, um die Daten von den Meta-Daten zu trennen. Wenn dies nicht konsequent und richtig gemacht wird, besteht die Möglichkeit einer sog. Truncation-Attacke: Der User fügt das Trennzeichen selbst in seine Daten ein – verkürzt sie sozusagen selbst – und fügt nach dem Trennzeichen eigene Meta-Daten ein. Wenn dann nicht sorgfältig geparst wird, werden aus User-Daten schnell eigene Meta-Daten. Bekannt ist diese Truncation-Attacke unter anderem durch das vorzeitige Beenden einer SQL-Anweisung durch Kommentarzeichen oder Abschlusszeichen. Auch im Bereich der Markup Languages können Tags früher geschlossen werden und anschließend weitere eigene Tags eingeführt werden. Die Gefahr einer SQL-Injection-Attacke besteht allerdings bei der Search Engine normalerweise nicht, denn der Index ist im Regelfall ein Set von Files und keine SQL-Datenbank.

Wir haben bereits gesagt, dass Input-Validierung bei Search Engines ein problematisches Vorgehen ist, denn es besteht die Gefahr, dass User-Input unzulässig eingeschränkt wird. Damit richtet sich der Fokus automatisch vor allem auf die Output-Validierung, auch *Encoding* genannt. Hier zeigen sich weitere Besonderheiten der Search Engine: Eine moderne Search Engine nimmt nicht einfach nur einen Query-String und vergleicht ihn mit einem Index. Stattdessen gibt es weitere Funktionen wie „Meinten Sie vielleicht ...“ oder die automatische Fehlerkorrektur von Eingaben. Spezialisten können bestimmte Default-Antworten auf bestimmte Queries festlegen, Dictionaries können vor und nach der Durchführung von Queries befragt werden etc. Sicherheitstechnisch hat dies folgende Konsequenzen:

- Output Encoding ist die wesentliche Sicherheitsmaßnahme gegen XSS-Attacken.
- Der Output der Search Engine, der dann durch das Front-End ausgeliefert wird, ist nicht vertrauenswürdig. Die Search Engine bezieht von verschiedensten Stellen ihren Input, da können durchaus auch Scripte dazu gehören.
- Die Hilfsfunktionen der Search Engine können einen User-Input erst gefährlich machen, indem sie die Attacke erst zusammenbauen: Zum Beispiel könnte der Query-Input „alert(\"foo\")“ durch das Vorschlagssystem umgewandelt werden in: Do you mean alert(\"foo\")? Eine Input-Validierung im Front-End müsste also sogar die Korrekturmöglichkeiten des Back-Ends mit einrechnen, was praktisch nicht möglich ist.
- Letztlich geht es aber noch extremer: Angriffscode muss nicht einmal im User-Input selbst vorhanden sein. Der Angreifer muss lediglich Input entdeckt haben, der bestimmte Scriptteile als Antwort der Search Engine verursacht. Das geht durch Zufall, systematisches Suchen oder noch einfacher durch einen Mitwisser, der innerhalb der Firma dafür sorgt, dass bestimmte Scripts von der Search Engine gespidert werden. Einfache Magic Numbers bei den Scripts lassen sie durch einfache Query-Strings finden.

Das Encoding des Outputs wird dadurch extrem wichtig. Im HTML, das an den User ausgeliefert wird, müssen alle Stellen, die Output der Search Engine enthalten, encodiert werden, also:

- Stellen mit Element-Content auf HTML-Metazeichen wie „<“ und „>“,
- Stellen mit Attributen auf „javascript:“ URLs oder „onopen“ etc. Anweisungen und
- eventuell weitere Formate wie XML und Javascript, je nach Konstruktion des Front-Ends.

Diese Regeln bieten zwei Schwierigkeiten. Erstens muss verhindert werden, dass durch Fehler in der Programmierung aus Versehen nicht-codierter Output im HTML landet. Hier gilt das gleiche Argument wie oben: Später dazu stoßende Teammitglieder oder sog. „schnelle Fixes“ sind bekannt als Ursachen von solchen Fehlern. Softwaretechnisch lässt sich das Problem durch sog. *Wrapper* in den Griff bekommen: Der originale Output der Search Engine wird durch eine Wrapper-Klasse so „eingewickelt“ dass der Zugriff auf die Originaldaten nicht mehr möglich ist. Wird dann nur der so gewrappte Output an die View-Klassen weitergegeben, dann besteht nicht die Gefahr des Zugriffs auf die Rohdaten (s. Abb. 2.28). Eine andere Möglichkeit wäre der Einsatz des Typsystems, in dem die originale Query bzw. encodierte Formen davon und die von der Antwort unterschiedliche Typen darstellen. Darstellungsfunktionen dürften dann gar keine nicht-encodierten Typen als Parameter annehmen. Dies stellt ein simples Tainting-Verfahren dar.

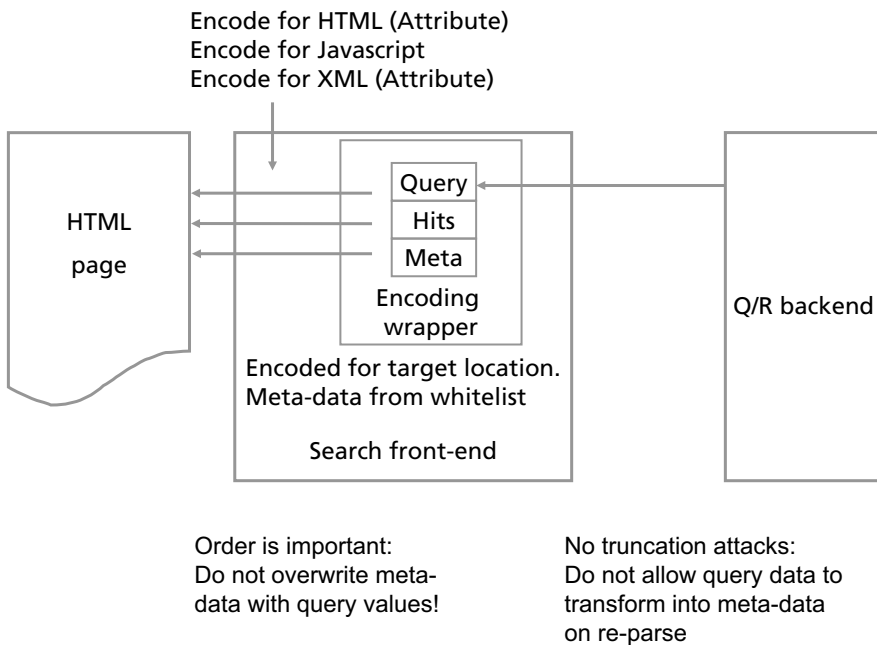


Abb. 2.28 Search Engine Output Processing

Die Implementation der Queries und Responses im Front-End sollte jedoch nicht einfach durch Strings und Methoden wie `getQuery()`, `getQueryEncoded()` etc. geschehen. Hier ist die Gefahr von Missverständnissen besonders groß. Und wieso ist die einfachste Methode „`getQuery()`“ nicht auch die sicherste, d. h. die encodierte Methode?

Die zweite Schwierigkeit bei der Encodierung des Outputs liegt im Gebiet der Zeichen und Zeichensätze: Welche Zeichen müssen wie encodiert werden? Dies hängt natürlich vom Ziel des Outputs ab, und außerdem von der Kompetenz derjenigen, die die Encodierung durchführen. Hier handelt es sich eindeutig nicht um etwas, das täglich von Applikationsentwicklern durchgeführt wird. Daher ist hier unbedingt der Einsatz einer professionellen Bibliothek anzuraten. Bei selbst geschriebenen Klassen zum Encodieren findet man XSS-Angriffsmöglichkeiten meist innerhalb der ersten Minute durch schlichtes Einfügen von `<script>alert („Hello“)</script>` an den unterschiedlichsten Stellen des Inputs (natürlich auch inmitten von Wörtern etc.). Wer das nicht glaubt oder sich über die unzähligen Möglichkeiten des Einfügens von Script informieren möchte, der wird bei den XSS Cheat sheets fündig: <http://ha.ckers.org/xss.html>.

Informationen zur Auswahl einer professionellen Open-Source-Bibliothek zur Encodierung finden sich bei Nick Coblenz. Er vergleicht mehrere Bibliotheken, unter anderem auch die neue ESAPI Library von OWASP [Cobl]. Besonders wertvoll neben der Bibliothek selbst ist die ESAPI Secure Coding Guideline (http://www.owasp.org/index.php/ESAPI_Secure_Coding_Guideline), die einen Fragebogen zu verschiedenen Sicherheitsthemen wie Session-Handling, Input Validation etc. darstellt, der je nach Applikation oder Projekt konkret mit den jeweils gewählten Lösungen ausgefüllt werden muss. Projekte haben dadurch die Sicherheit, nichts Wichtiges übersehen zu haben.

Zum Vorgehen beim Encoding ein Zitat:

“In general, both the ESAPI and Reform libraries encode any value other than a–z, A–Z, and 0–9 (there are some exceptions). This is a great approach to ensuring client-side input cannot be interpreted as HTML or JavaScript commands when it is redisplayed in the browser.” [Cobl]

Das API zum Encodieren ist meist recht einfach. Hier ein Beispiel aus der OWASP-Reform-Bibliothek. Man sieht deutlich, dass es das Ziel ist, das über das Encoding entscheidet:

- `HtmlEncode`,
- `HtmlAttributeEncode`,
- `XmlEncode`,
- `XmlAttributeEncode`,
- `JsString` und
- `VbsString`.

Search Engine Back-Ends erzeugen teilweise eigene Tags, um Resultate hervorzuheben oder zu unterscheiden. In diesen Fällen ist es jedoch meist möglich,

diese speziellen Tags nachträglich in einem Servlet-Filter wieder in wirksame HTML-Tags zurück zu verwandeln, nachdem sie durch das Encodieren zunächst unbrauchbar gemacht wurden. Im Weiteren gilt natürlich, dass in URLs, die vom Front-End erzeugt werden, lediglich eigene Meta-Daten der Applikation auftreten sollten bzw. der encodierte Query-String des Users. Keineswegs sollte jedoch eine Mischung aus beiden entstehen, bei dem User-Input an den Stellen der Meta-Daten auftaucht. Egal ob schädlich oder nicht: Das Front-End hat keine URLs mit fragwürdigem Inhalt zu erzeugen.

Möglichkeiten und Einsatzbedingungen einer Web Application Firewall (WAF)

Die Mächtigkeit einer WAF ergibt sich aus zwei Dingen: Der Kenntnis des Applikationsprotokolls und der Möglichkeit umfangreiche States, z. B. zwischen einer vom Server ausgelieferten Page und einer Antwort des Clients, zu halten. So kann die vom Client ausgefüllt zurückgesendete Form Page auf Änderungen an „hidden“ Feldern automatisch geprüft werden. Requests von Clients können dynamisch geprüft werden, ob sie überhaupt einmal vom Server in einer Page vorhanden waren, oder ob sie konstruiert wurden. Natürlich können Pages auch darauf geprüft werden, ob sie tatsächlich vom Server ausgeliefert wurden, um Cross Site Request Forgery zu entdecken. Hier zusammengefasst die Möglichkeiten einer WAF:

- URL-checking,
- unicode normalization,
- message canonicalization for filtering,
- stateful filtering of selected requests,
- stateful connection of input/output values und
- stateful link/request control (did the link come from the server?).

Teilweise können diese Filtermöglichkeiten sogar in Konflikt mit der Applikationslogik kommen, wenn diese z. B. die clientseitige Konstruktion von URLs vorsieht, die dann von der WAF zurückgewiesen werden, weil sie nie so in einer Page vom Server geliefert wurden.

Einen guten Überblick und methodischen Vergleich diverser WAF-Produkte liefert [Roth], der auch auf die negativen Aspekte von WAFs eingeht. Dazu gehört z. B. der enorme Konfigurationsaufwand, den diese Produkte benötigen. Zwar gibt es mittlerweile sog. „selbstlernende“ Produkte, die jedoch umfangreiche Tests voraussetzen und vor ähnlichen Problemen stehen wie die „selbstlernenden“ Intrusion-Detection-Systeme.

Lohnt sich überhaupt der Aufwand für die Konfiguration einer WAF als zusätzliche Verteidigungslinie? Die Antwort lautet eindeutig „ja“ und liegt in der Art und Weise begründet, wie auf eine entdeckte Schwachstelle in einer großen Ap-

plikation häufig reagiert wird: Mit panischen Änderungen am Source Code. Diese Änderungen führen in der Regel kurz nach dem erneuten Deployment bereits zu hässlichen Stellungnahmen in den Medien: Hat wieder nicht geklappt. Oft wird nämlich durch eine schnelle Codeänderung das entdeckte Problem gar nicht wirklich beseitigt. Teilweise werden neue Schwächen eingebaut oder eng verwandte Schwächen bleiben unentdeckt.

Die bange Frage vom Management: „Sind jetzt alle XSS-Vulnerabilities gefunden und gefixt?“ lässt sich leider typischerweise bei großen Applikationen so einfach nicht beantworten. Ältere Web-Applikationen wurden oft ohne ein Konzept von Input-Validierung entwickelt und die publizierte Schwäche erweist sich bei näherem Hinsehen als fundamentaler Architekturfehler. Oder es stellt sich heraus, dass neu hinzu gekommene Entwickler eine ganze Reihe von Funktionen eingebaut haben, ohne die vorhandene Validierungslogik zu nutzen. Dort finden sich dann zuhauf XSS-Attacken.

Leider ist häufig auch das simple Abschalten der gefährlichen Funktionen aus geschäftlichen Gründen gar nicht möglich (nebenbei bemerkt dauert selbst das Unterdrücken eines bestimmten http-Requests in der Regel viel länger, als man meinen sollte: Die Entwicklungsmannschaft wird unvorbereitet von der Publizierung einer XSS-Schwäche der eigenen Applikation überrascht und stellt fest, dass weder die Filter-Syntax vorgelagerter Proxies bekannt ist, noch wie viele Proxies mit Caches eventuell existieren, die alle abgeändert werden müssen. Wenn aber das bloße Abschalten eines Requests schon so lange dauert – wie hoch sind dann die Chancen, in einer alten Web-Applikation schnell und zuverlässig durch Änderungen im Source Code den Fehler beseitigen zu können?).

Aber zurück zum Geschäftsproblem: Die Funktion, bei der die XSS-Schwäche publiziert wurde, erweist sich als „mission critical“ für das Geschäft und kann nicht abgeschaltet oder unterdrückt werden. Jetzt erweist sich der Einsatz einer WAF als echter Retter: Mit der WAF lassen sich kontext-sensitive Änderungen an einzelnen Requests dynamisch durchführen, ohne dass die Applikation sofort geändert werden müsste. Der Request wird sozusagen entschärft. Weniger kritische Requests werden natürlich einfach unterdrückt. Durch den Einsatz einer WAF erkaufte sich das Entwicklungsteam die Zeit, publizierte Schwächen korrekt und vor allem vollständig ausmerzen zu können. Die Erfahrung hat gezeigt, dass dies den Aufwand für die Konfiguration einer WAF in jedem Falle wert ist.

Ein ähnlicher Effekt – Vermeiden von Codeänderungen in alten Applikationen – lässt sich übrigens auch durch die Verwendung von Servlet-Filtern erreichen. Diese sind relativ schnell anpassbar und der Performance Impact ist gering. Eine Besonderheit besteht darin, dass der Inhalt des Requests innerhalb eines solchen Filters nicht geändert werden kann. Im Lichte des oben zu Sanitizing Gesagten ist dies aber ein kleineres Problem. Notfalls besteht die Möglichkeit, durch ein Wrapper-Objekt eine modifizierbare Version des Client-Requests zu erstellen und weiter zu reichen. Im Beispiel zu Struts-Security weiter unten wird der Einsatz eines Servlet-Filters in Bezug auf die Möglichkeiten zur Access Control durchgespielt.

Testinstrumente: Fuzzer und (semi-)automatisierte Testtools

Nach H.D. Moore – Autor des Metasploit Frameworks (s. [Moore]) – war der Juli 2006 der Monat des Browser-Bugs. Der Monat endete mit der folgenden Zählung von gefundenen Schwachstellen:

- Internet Explorer: 25,
- Mozilla: 2,
- Safari: 2,
- Opera: 1 und
- Konqueror: 1.

Interessanter als die Tatsache, dass der übliche Verdächtige, der Internet Explorer, klar in dieser Statistik führt, ist die Methodik, mit der diese Bugs gefunden wurden: der Einsatz von sogenannten „Fuzzern“. Was ist das eigentlich? Moore verwendet selbst geschriebene Werkzeuge, die verschiedene Schwächen durch die Eingabe von beliebigen Werten entdecken können:

“As promised, I have released my ActiveX fuzzing tool, aptly named AxMan. This tool was used to discover and debug almost every single ActiveX flaw published during the Month of Browser Bugs. In addition to the MoBB issues, this tool discovered over 100 unique flaws on a Windows XP SP2 system with common third-party packages installed. I am releasing this tool without my blacklist.js file of discovered vulnerabilities; this should give the vendors some breathing room while they figure out how to address these problems. An online demonstration of AxMan is available, but the interface is not designed to work across a slow network and a locally installed version will run much faster. Enjoy and happy bug hunting!”

Diese Werkzeuge entdecken:

- Angriffspunkte in COM-Objekten, die im Internet Explorer zugänglich sind,
- DHTML-Implementierungsschwächen durch Eingabe „böser“ Werte für Methodenargumente und Properties,
- Eingabe von beliebigen Werten in CSS-Style-Values und
- DHTML-Implementierungsschwächen durch die Manipulation des DOM durch Hinzufügen oder Löschen beliebiger Elemente.

Im Grunde sind diese Fuzzer nicht viel anders als das bereits 20 Jahre alte Tool „crashme“, das für Tests von Betriebssystemen eingesetzt wurde. Ziel war es damals, Input-Validierungsschwächen im System-Call-Interface des jeweiligen Betriebssystems zu finden. Die Taktik war sehr einfach: Die crashme-Applikation konfigurierte in der Initialisierungsphase alle System-Exceptions (Signale) so, dass sie der Applikation selbst zugestellt wurden und nicht zum Abbruch des Programms führten. Anschließend erzeugte crashme Zufallszahlen in einem langen String und setzte den Instruktionspointer auf den Beginn des Strings, d. h. crashme fing an, die Zufallszahlen als Maschineninstruktionen auszuführen. Natürlich ergeben Zufallszahlen keine sinnvollen Maschinencodes, aber das heißt nicht, dass sie nicht ausgeführt werden würden. Manche erzeugten Fehler (divide by zero, memory access failure etc.), aber ab und zu entstanden auch Codes, die einen Einsprung in den Kernel über den Trap-Mechanismus repräsentierten. Diese Ein-

sprünge (Funktionen) erwarteten natürlich bestimmte Argumente, die ebenfalls Zufallszahlen waren. Eine gut abgesicherte Kernel-Funktion gibt dann einen Fehler an die Applikation zurück. Eine schlechte hingegen führt Kernel-Operationen mit unsinnigen Parametern durch und bringt den Kernel aller Wahrscheinlichkeit nach zum Absturz.

Auf sourceforge.net finden sich übrigens noch weitere Fuzzer-Projekte. Eine Einführung mit Vorstellung verschiedener Fuzzer findet sich in [Lehr]. Für die Suche nach Buffer Overflows eignet sich da besonders FileFuzz, der Dateien mit zufälligen Inhalten erzeugt und sie Programmen vorsetzt.

Es lohnt sich an dieser Stelle, das Vorgehen beim „fuzzen“ mit der „normalen“ Testmethodik zu vergleichen, wo über Regressionstests die gewünschte Funktionalität geprüft wird – aber natürlich nie die Programmfunktionen, die außerhalb der Spezifikation bzw. der Use Cases einer Applikation liegen. Dieser Affront gegen die normale Testmethodik wird nochmals verschärft durch den Angriff auf eine User-Applikation – nicht auf einen ohnehin gefährdeten öffentlichen Web-Server. Dabei ist doch bekannt, dass viele PC-Programme bereits „von selbst“ abstürzen und von den Benutzern sehr vorsichtig bedient werden müssen – gerade so, als ob die Instabilität der Software auf PCs ein Naturgesetz darstellen würde. Der enge Zusammenhang zwischen Softwarequalität und Sicherheit wird leider von vielen Softwarefirmen verneint.

Der manuelle Aufwand beim Testen von Web-Applikationen ist nach wie vor hoch. Automatisierte Testwerkzeuge leiden gerne unter zu vielen falschen Treffermeldungen bzw. übersehen wichtige Dinge. Ein interessantes Werkzeug, das diese Probleme vermeiden soll und besonders auf Web2.0-Applikationen zugeschnitten ist, wurde kürzlich von Google vorgestellt: Ratproxy, entwickelt vom bekannten Sicherheitsspezialisten Michael Zalewski. Ratproxy ist ein halbautomatisches, überwiegend passiv operierendes Werkzeug, das einige fortgeschrittene Analysetechniken beherrscht, wie z. B. dynamische Decompilierung von Action Script. Mehr zu diesem Tool bei [Goog].

Zum Abschluss sei noch ein Werkzeug erwähnt, das sich nicht defensiv auf die Absicherung von Infrastrukturen oder Software richtet, sondern das aktiv versucht, Angriffsversuche festzustellen. Gemeint ist das Werkzeug „Web Exploit Finder“, mit dem geprüft werden kann, ob der eigene Browser beim Besuch einer Website durch dortigen Code angegriffen und kompromittiert wurde. Eine genaue Beschreibung des Werkzeugs findet sich bei [Müller]. Die Feststellung des Angriffs verwendet virtuelle Maschinen und lässt sich auch beliebig mit spiders zur automatischen Durchsuchung des Webs kombinieren.

2.4 Attacken auf Ebene der Semantik

2.4.1 *Die Kunst der Täuschung*

Das Buch „Die Kunst der Täuschung“ (im Original „The art of deception“) von Kevin Mitnick [Mit] stellt eine ausgezeichnete Sammlung von Social-Engineer-

ing-Techniken dar. Wer es mit der Erwartung liest, technische Tipps für Angriffe zu erhalten, wird es enttäuscht weglegen: Praktisch alle Angriffe und Täuschungen, die im Buch geschildert werden, sind nicht-technischer Natur. Das wichtigste technische Hilfsmittel bei der Durchführung der Angriffe ist das Telefon. Wieso dem Autor nach Verbüßen seiner Haftstrafe die Benutzung des Internet verboten wurde, bleibt zunächst rätselhaft. Eher hätte man ihm das Telefonieren verbieten müssen – vielleicht haben die Richter aber auch die Möglichkeit gesehen, die Techniken der Täuschung mit dem Telefon aufs Internet zu übertragen: Bei beiden Medien fehlt z. B. der unmittelbare visuelle Eindruck, was dazu führt, dass andere Kanäle und Mitteilungsformen deutlicher wahrgenommen werden.

Der wichtigste Beitrag des Buches zur Sicherheitsdiskussion liegt aber darin, an vielen Beispielen zu zeigen, wie Täuschung auf sozialer Ebene funktioniert. Die Grundregeln hierfür muss man sich allerdings selbst anhand der Beispiele erarbeiten. Sie lauten:

- Eine gute Täuschung manipuliert die Technik nicht. Im Gegenteil, sie braucht ihr normales Funktionieren.
- Die Täuschung selbst beruht auf dem Teil, der dem Opfer nicht gesagt wird. Sie besteht nicht in der glatten Lüge, sondern in dem, was das Opfer scheinbar ganz automatisch aus dem Gesagten schließt. Der Trugschluss, der die Täuschung ausmacht, wird also vom Opfer selbst hergestellt.
- Die Täuschung nutzt normales, ansonsten hilfreiches Verhalten und dessen Motivation aus: Wir wollen Kollegen, die neu angefangen haben, helfen. Wir wollen einem Besucher unserer Firma helfen etc. Daraus lässt sich schließen, dass ein gewisses Maß an Täuschung in einer offenen und freien Gesellschaft praktisch unvermeidlich ist. Abwehrmaßnahmen, die generell das Misstrauen innerhalb der Mitarbeiterschaft vergrößern, sind für das normale Zusammenleben kontraproduktiv und rechnen sich letztlich auch nicht.
- Die Täuschung baut Vertrauen durch mehrere kleine Informationsbrocken auf, die für sich genommen unerheblich erscheinen, in der Summe aber beim Opfer letztlich den Schluss der Vertrauenswürdigkeit auslösen: „Aber er kannte doch die Abteilungsnamen und die Namen der Chefs!“
- Jeder kann Opfer einer Täuschung werden – es kommt nur darauf an, den richtigen Ansatz zu wählen: Wenn ich eine E-Mail von einer Bank bekomme, zu der ich keine Kundenbeziehung habe, dann lässt mich das relativ kalt. Kommt die Mail aber von einer Firma, zu der ich oder Familienangehörige gerade eine Beziehung aufgebaut haben, von der bekannt ist, dass sie ihre Rechnungen online verschickt und erwähnt die Mail eine hohe Rechnungssumme, dann ist die Chance groß, dass in der Aufregung der Anhang der Mail vorschnell geöffnet wird. Das kann gleichsam eine Lawine auslösen: Weitere Mails werden an Adressen aus dem Adressbuch meiner Maschine verschickt und die Absender werden ebenfalls meinem Adressbuch entnommen. Damit ist die Chance groß, dass Empfänger die Anhänge ebenfalls öffnen, denn Sender und Empfänger gehören zu meinem sozialen Netzwerk. Genau so hat der erste „soziale“ Virus Klez funktioniert.

Was jetzt dringend nötig wäre, ist eine Untersuchung und Klassifizierung der von Mitnick ausgenutzten Irrtümer der Betroffenen. Diese Irrtümer führen dann zu einem User Conceptual Model, das z. B. auf Browserarchitekturen angewendet werden könnte: Wie können Browser vor diesen Irrtümern schützen? Ein solcher Schutz würde bedeuten, dass der Nutzer nicht mehr mit Meldungen über Zertifikate und Warnungen über an das Internet gesendete Informationen verwirrt wird, sondern dass seine Aktionen und Zielvorstellungen auf einer höheren, „semantischen“ Ebene abgesichert werden.

2.4.2 Phishing und Multi-Faktor-Authentisierung

In den letzten Jahren – vielleicht unter dem Einfluss verbesserter Sicherheit in der Übertragung von Daten durch SSL – haben sich semantische Attacken immer mehr ausgebreitet. In ihrer häufigsten Form werden sie als *Phishing* (ein Kunstwort aus „Password“ und „Fishing“) bezeichnet. Die Funktionsweise des Phishing ist recht einfach: Es werden gefälschte Mails verschickt – scheinbar von Sites, bei denen die Benutzer sich registriert haben. Das können Banken, Shops, Bezahl-dienste etc. sein. Die Mails beinhalten einen Link und der Text fordert mit verschiedenen Begründungen und unter Ausübung von Druck („Registrieren Sie sich neu oder Ihr Konto wird gesperrt!“) dazu auf, diesen Link zu öffnen. Folgt der Nutzer dieser Anweisung, landet er auf einer gefälschten Site, die der originalen Seite des Dienstleisters zum Verwechseln ähnlich sieht. Dort wird der Benutzer zur Eingabe seiner Credentials (je nach Service meist UserID/Passwort aber auch PIN/TAN) aufgefordert. Im Anschluss daran missbraucht der Urheber der gefälschten Site diese Credentials für eigene Transaktionen im Namen des Opfers. Mittlerweile haben allerdings trojanische Pferde, die vom Angreifer auf dem Rechner des Opfers zum Zweck des Stehlens von Credentials platziert werden, die gefälschten Mails als häufigste Form des Phishing abgelöst.

Aus technischer Sicht läuft beim Phishing mit gefälschten Mails alles ordnungsgemäß ab – meist wird in der Mail die Eigenschaft von HTML ausgenutzt, dass der angezeigte Link nicht mit der tatsächlich referenzierten URL übereinstimmen muss. Aber davon abgesehen, ist der einzige technische Vorwurf, den man der Original-Site machen kann, dass sie auf kanalbasierte Sicherheit (d. h. SSL) statt auf nachrichtenbasierte Sicherheit im Sinne einer Signierung von Transaktionen gesetzt hat. Dies setzt jedoch wieder (teure) Infrastruktur wie etwa einen Smartcard Reader voraus. Als technische Alternative dazu wird von einigen Firmen die sogenannte Multi-Faktor-Authentisierung angesehen. Im Finanzbereich ist sie mittlerweile Standard: Neben einer UserID und einer PIN wird für jede einzelne Transaktion eine zusätzliche Authentisierung in Form einer sogenannten Transaktionsnummer (TAN) vom Nutzer verlangt. Diese Nummern wurden vorher auf dem Postwege an die Kunden verschickt. Zur Authentisierung seiner Transaktion muss der Kunde die nächste, noch nicht verwendete TAN eingeben und danach streichen (weshalb das Verfahren auch Streichliste genannt wird).

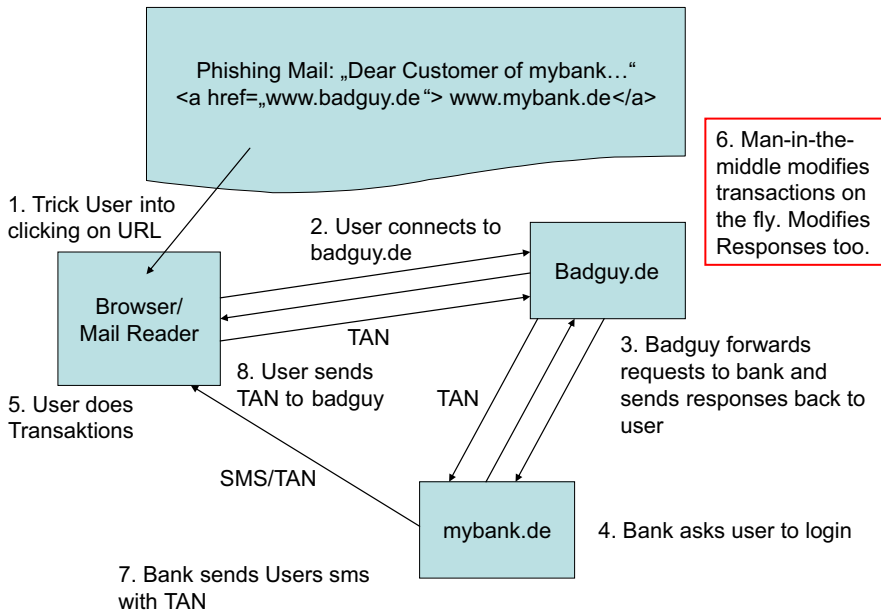


Abb. 2.29 Phishing-Angriff auf das MTAN-Verfahren

Beim sogenannten iTAN-Verfahren schreibt die Original-Site sogar vor, die wievielte Nummer auf der Liste eingegeben werden muss. Andere Systeme verschicken eine TAN von der Original-Site per SMS an den Kunden (MTAN-Verfahren, s. Abb. 2.29).

All dieses technische Brimborium ist jedoch völlig nutzlos, wenn der Benutzer bereits eine Verbindung zu einer gefälschten Site eröffnet hat: Dann wird er auch das zweite und dritte Geheimnis dem sog. Man-in-the-Middle (MITM) zum Missbrauch vorlegen. Der MITM spielt dabei die Rolle eines intelligenten Proxies zwischen dem Kunden und der Site und fängt die Credentials ab bzw. ändert die Transaktionsdaten zwischen Kunde und Bank in Real-Time. Der Angriff gestaltet sich dadurch lediglich etwas aufwändiger (s. Abb. 2.30, entnommen aus [A-I3]).

Voraussetzung für jede Phishing-Attacke ist, dass das Opfer auf einen gefälschten Link klickt und auf dem Server des Angreifers seine Geheimnisse preisgibt. Das entscheidende Kriterium zum Erfolg des Angriffs ist deshalb die Glaubwürdigkeit der Phishing-Mail sowie der gefälschten Site. Heutige Phishing-Mails kommen häufig in einer Sprache daher, der man ansieht, dass sie mithilfe eines Übersetzungsprogramms generiert wurden („Geehrte Kunden Postbank“), aber es gibt keine Garantie dafür, dass dies immer so bleibt.

Umso schöner ist es aber, wenn die Original-Site bei der Attacke noch mithilft. Im Screenshot (Abb. 2.31) ist klar zu sehen, dass die Titel der Page zwar von der LBBW kommen, innerhalb der Page jedoch etwas anders (in diesem Fall zu Demozwecken lediglich www.google.de, bei einem realen Angriff aber eine gefälschte Site der LBBW) dargestellt wird. Ebenfalls deutlich sichtbar in der Browser-URL ist der Server der LBBW.

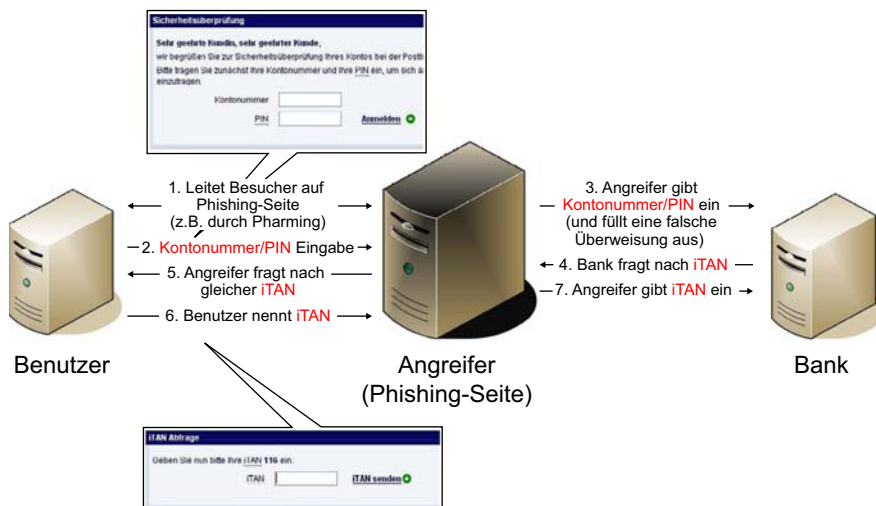


Abb. 2.30 Phishing-Angriff auf iTAN-Verfahren

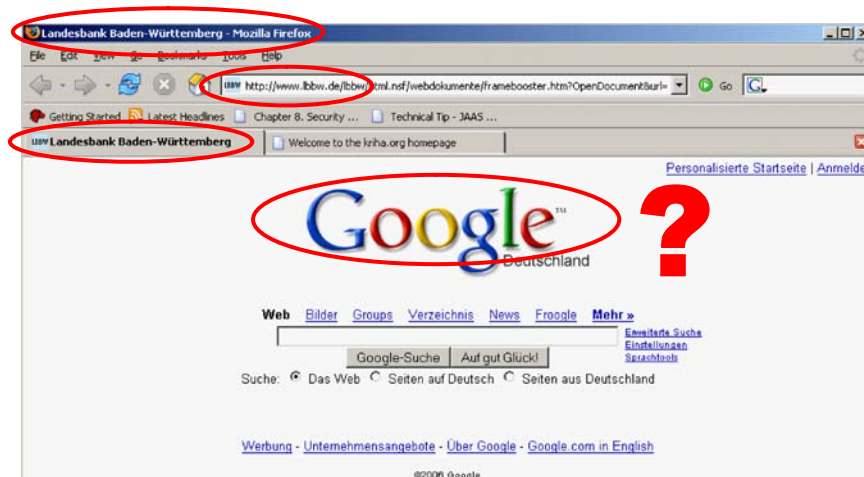


Abb. 2.31 Fremden Link einbetten

Die Input-Validierungsschwäche eines Bankenservers kann für Phishing-Angriffe ausgenutzt werden:

Der Server der Bank nimmt GET-Parameter an, ohne sie zu prüfen und öffnet mit ihnen eine neue Page (Abb. 2.32).

Die Antwort des Servers der Bank zeigt, wie es zu der Anzeige oben kommt. Es wird ein Frameset mit der URL, die dem Server übergeben wurde, geöffnet. Der Server erwartet natürlich eine URL einer LBBW Seite, prüft das jedoch nicht.

A "Phishing-Link" to LBBW Bank:

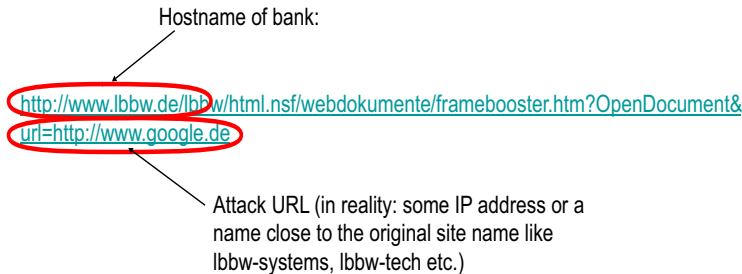


Abb. 2.32 Aufbau des „Phishing-Links“ zur LBBW

```
<html><head><title>Frame-Generator</title>
<SCRIPT LANGUAGE="JavaScript">
function createFrameset() {
    var Titel="Landesbank Baden-W&uuml;rtemberg";
    var s=location.search;
    var Url=s.substring(s.indexOf("url=")+4,s.length);
    document.clear();
    document.open("text/html");
    document.writeln('<HTML><HEAD><TITLE>'+Titel+'</TITLE><
/HEAD>');
    document.writeln('<FRAMESET rows="*,19" frameborder=0
border=0 framespacing=0><FRAME name="level_1" marginwidth=0
marginheight=0 src="'+Url+'" scrolling=no><FRAME
name="bottom" marginwidth=0 marginheight=0 src="bottom.htm"
scrolling=no></FRAMESET>');
    document.writeln('</HTML>');
    document.close();
}
</SCRIPT>
</head><body onload="createFrameset()"></body></html>
```

Der Entdecker dieser Angriffsmöglichkeit, Frank Falkenberg, hat dieses Problem an die IT-Abteilung der Bank gemeldet, doch auch mehr als eine Woche danach funktionierte die Attacke noch (siehe dazu die Anmerkungen zum Einsatz einer Web Application Firewall). Das Problem ist für einen normalen Anwender nur sehr schwer zu erkennen, da der Browser keine visuellen Hinweise auf den fremden Content gibt und die eingeschleuste URL noch dazu durch Maskierungen (IP-Adresse, Name ähnlich der LBBW) versteckt werden kann. Schutz davor gibt

es nur, wenn der Benutzer daran denkt, die üblichen Sicherheitsregeln beim E-Banking strikt zu befolgen:

- Alle Browserfenster schließen und ein neues für das E-Banking öffnen (das schließt existierende Sessions, um Cross-Session-Vulnerabilities zu vermeiden. Beispielsweise könnte eine Site per Script ein Popup-Window einer anderen abfangen).
- Die URL der Bank von Hand eintippen (keine Bookmarks, sie könnten mit Script versehen sein bzw. bereits die falsche URL enthalten).
- Keine Links aus Mails benutzen.

All diese Sicherheitsvorkehrungen gehen aber ins Leere, wenn der Angreifer sich im selben LAN wie das Opfer befindet und es ihm gelingt, die Antworten des DNS-Servers auf die Anfragen des Opfers zu fälschen (DNS-Poisoning). Dann landet ein User auch bei Eingabe der korrekten URL von Hand auf dem Server des Angreifers. Das einzige, was jetzt noch helfen kann, ist ein großes technisches Wissen des Users, der das ihm präsentierte SSL-Zertifikat des Angreifers sehr genau prüft, oder aber die Nutzung von elektronisch signierten Bank-Transaktionen wie im HBCI-Standard. Der Angreifer hat dann keine Chance, die Transaktion des Opfers in seinem Sinne zu verändern.

Mittlerweile gibt es auch mit TLS-PSK (PSK steht für „Pre-Shared Keys“) eine Erweiterung des SSL/TLS-Standards, der nach dem Aufbau eines sicheren Tunnels zwischen Client und Server den Server zusätzlich durch ein leichtgewichtiges Challenge-Response-Verfahren authentisiert, welches wiederum auf einem vorher ausgetauschten Geheimnis beruht (s. [RFC4279] und Kap. 13). Auch dieses Verfahren sollte in der Lage sein, die meisten Phishing-Attacken zu verhindern, wenngleich das Sicherheitsniveau nicht an das von HBCI heranreicht.

2.4.3 Soziale Attacken

Die „Kunst der Täuschung“ kann noch weit über semantische Attacken (also Irrtümer der Opfer) hinausgehen und weitere menschliche Schwächen wie blinde Gier, Neid, Bössartigkeit und Neugier mit einbeziehen. Einen kleinen Ausblick auf Dinge, die da kommen könnten, geben Mike Bond und George Danezis in „A pact with the Devil“ (s. [BoDa]). Dort beschreiben sie Malware, die die Nutzer bewusst installieren, weil sie teilweise davon profitieren. Solche Malware verbreitet sich durch Ausnutzen der menschlichen Schwächen und nicht durch technische Mängel oder Irrtümer. Dadurch, dass sie bewusst installiert wird, verschwimmt die Grenze zwischen Attacke und normaler Software. Das Prinzip ist nicht unähnlich dem mancher Peer-to-Peer-Software wie z. B. Kazaa, die neben dem Download von Musik auch zahlreiche Werbeeinblendungen mit sich bringt. Im hier zu besprechenden Fall resultiert jedoch der Vorteil des Einsatzes der Malware aus dem

Nachteil anderer User derselben Malware. Bond und Danezis geben folgendes Beispiel, das auf dem Prinzip von „Zuckerbrot und Peitsche“ beruht:

- Phase 1 (Zuckerbrot): Die Malware verspricht dem User Vorteile durch die Installation und hält dieses Versprechen auch ein. Im Hintergrund sammelt die Malware Daten von der Maschine des Users und speichert sie auf anderen Rechnern.
- Phase 2 (Peitsche): Die Malware warnt den User, dass beim Versuch, die Malware zu entfernen, die privaten Daten des Users an die Öffentlichkeit gelangen würden.

Als Durchführungsbeispiel geben die Autoren einen Virus namens „Satan“, der sich per E-Mail bewusst oder unbewusst verbreitet. Von einem befallenen Rechner wird an einen anderen Benutzer eine Mail mit der Aufforderung geschickt, das angefügte Programm zu installieren. Die Mail verspricht, dadurch Daten von Benutzern auf der befallenen Maschine dem User zur Verfügung zu stellen und beweist dies auch durch Ausschnitte der Daten, die den Absender der Mail betreffen. Der User entscheidet sich zur Installation und erhält Zugriff auf Daten der anderen Benutzer. Gleichzeitig fängt die Malware an, alle Aktionen des Users zu protokollieren und auf der ursprünglich befallenen Maschine in verschlüsselter Form zu speichern. Irgendwann hat die Malware genug von den Spionageaktionen des Users gegen die Ausgangsmaschine gesammelt und geht in die Phase der Erpressung über: Die Malware warnt den User vor dem Entfernen, denn das hätte das Aufdecken seiner Taten zur Folge. Die Malware installiert dazu eine regelmäßige Verbindung zwischen beiden Maschinen zur Überwachung (z. B. durch Leases: Die Daten bleiben versteckt, wenn alle x Stunden oder Tage eine Nachricht von Maschine y kommt. Andernfalls werden sie aufgedeckt.) Gleichzeitig fordert die Malware den User auf, sie an weitere User zu mailen und verspricht dadurch Zugriff auf deren Daten. Der Benutzer ist jetzt in einer schwierigen Lage. Er hat vielleicht den Verdacht, dass auch seine Daten anderen zur Verfügung gestellt werden, aber von der Protokollierung seiner Spionagetätigkeit gegen andere User wusste er nichts. Ist die Erpressungsmöglichkeit einmal vorhanden, kann die Malware beliebig zwischen Zuckerbrot und Peitsche wählen.

Aus diesem Beispiel entstand eine interessante Diskussion, inwieweit z. B. die Nutzung von Capabilities und ein sicherer Desktop, der den Willen des Benutzers respektiert (d. h. der keine Programme automatisch ohne ausdrückliche Einwilligung des Users ausführt), diese Art von Attacke abwehren können. Das Ergebnis ist nicht ganz klar: Einerseits können sie die Attacke nicht abwehren, denn der User führt ja die Malware freiwillig aufgrund der ihm versprochenen Vorteile aus. Er muss allerdings bewusst die dazu nötigen Ressourcen freigeben, denn in einem Capability-basierten System bekommt keine Software automatisch Zugriff auf angeforderte Ressourcen. In dieser bewussten Zuteilung von Ressourcen liegt ein hoher Sicherheitsfaktor. Das Ziel der Malware muss es also sein, den User zu diesen Freigaben zu motivieren. Die Frage ist also, wie man den Benutzern anzeigen könnte, was wirklich passiert, wenn sie die Malware benutzen und wie die Benutzer bestimmte Daten dennoch schützen können. Hier kann der defensive Mechanismus der Schadensbegrenzung durch Capabilities durchaus nützen, der

Rest ist eine Frage neuer Abstraktionen und neuer User-Interfaces, um sie auch sichtbar und benutzbar zu machen.

2.5 WEB2.0-Techniken und ihre Problematik³

Web2.0 und seine technische Grundlage AJAX (Asynchrones JavaScript und XML) haben in letzter Zeit großes Aufsehen erregt. Gartner erwähnt es als „Key Technology“ unter den „Emerging Technologies 2006“:

“Web 2.0 represents a broad collection of recent trends in Internet technologies and business models. Particular focus has been given to user-created content, lightweight technology, service-based access and shared revenue models. (technical base: AJAX, RubyOnRails etc)”. (s. [Gart])

Gartner nennt im gleichen Atemzug auch andere Techniken wie Collaborative Intelligence (die gemeinsame Erstellung von Content), Mashups (die Kombination externer Quellen in neue Inhalte), und Social Network Analysis (Dinge wie OpenBC oder LinkedIn und deren Möglichkeiten, soziale Netzwerke zu erstellen und auszuwerten). Für andere hingegen sind auch klassische Client/Server Anwendungen wie Wikis Teil von Web2.0, obwohl dort meist nicht über Domains hinweg aggregiert oder massiv mit JavaScript gearbeitet wird.

Oft wird der Begriff „Web2.0“ in Deutschland als Synonym für AJAX – also die asynchrone Kommunikation mithilfe von JavaScript und XML – verwendet. Die Technik AJAX macht jedoch nur einen kleinen Teil von Web2.0 aus. Bei Web2.0 geht es nicht so sehr um Technologien, sondern in erster Linie um die soziale Komponente des „neuen Web“ (s. [OR] und [EPR]). Beispiele für Web2.0-Applikationen finden sich bei Google (Maps, Suggests etc.), Yahoo etc. Ein schönes Beispiel für die Einbindung von Google Maps Services in eigene Seiten findet sich unter www.sk8mag.de: Hier werden geografische Informationen mit Daten und Bildern zu Skateparks verknüpft. Gute Beispiele für die Nutzung von AJAX im Rahmen von Web2.0-Applikationen sowie ausführliche Erläuterungen der Techniken finden sich bei Kai Jäger [Jäger].

In vielen der Web2.0-Applikationen sind der freie Datenaustausch und die freie Benutzung von Diensten anderer Kernbestandteile der Architektur. Dieser Austausch von Information und Funktion findet auf einer technischen Plattform statt, nämlich dem Web und seinen User-Agenten (Browsern), die ursprünglich eher im klassischen Sinne eine Client/Server-Architektur darstellte und keine kollaborative. Dies drückt sich z. B. im zentralen Sicherheitsgebot für den Einsatz von Scripts oder Applets aus: „Same Origin Principle“ – Verbindungsaufbau nur zurück zu der Site, von der der Code kam.

Im Folgenden geht es darum, die Abbildung der neuen sozialen und Geschäftsmodelle von Web2.0 über die Ajax-Technologie auf eben die vorhandene Brow-

³ Dieses Kapitel entstand gemeinsam mit Kai Jäger, dem Autor von „Ajax in der Praxis“.

serwelt in Bezug auf die Sicherheitsmodelle zu untersuchen: Werden die existierenden Modelle verletzt? Sind die Browser in der Lage, die neuen Funktionen sicher zur Verfügung zu stellen? Wo sind die Problempunkte? Sollten die Sicherheitsmodelle gelockert werden, um bessere Aggregation zu ermöglichen?

2.5.1 *Attacken auf Web2.0*

Sehr früh nach der Vorstellung von Web2.0 sind kritische Stimmen zur Sicherheit von Web2.0-Anwendungen aufgetaucht. So wurde die hohe Komplexität des Codes bemängelt, die Intransparenz der Seiten durch dynamische Inhalte, die Testprobleme dynamischer Seiten etc. Auch sind bereits erste Exploits aufgetaucht, bei denen JavaScript an Mails angehängt wurde und sich so verbreiten konnte (s. unten). Sind dies neue Attacken oder lediglich Nutzungen bekannter Angriffsvektoren wie z. B. Cross-Site-Scripting? Sind diese Probleme Implementationschwächen oder bedingt durch die Architektur von Web2.0? Diesen Fragen wollen wir uns jetzt zuwenden. Zunächst zum ersten bekannt gewordenen Exploit:

In einem Artikel der Financial Times zu Schwächen der Web2.0-Applikationen in Punkto Sicherheit [Nutt] erwähnt der Autor zwei Attacken. Die erste Attacke wurde als der „Yamanner Worm“ bekannt:

“Bad code” struck Yahoo’s web mail service in June when a virus writer sent out an e-mail with some JavaScript code embedded invisibly inside it. Yahoo Mail was vulnerable to what became known as the Yamanner worm as it allowed JavaScript to be executed. Anyone opening the e-mail triggered the script, which made requests for the user’s address book and sent the worm on to everyone listed. The aim was probably to collect addresses for spam lists. “The Yamanner worm couldn’t have happened without Ajax,” says Billy Hoffman, security researcher at SPI Dynamics. “This is the frightening thing – from Yahoo’s point of view, nothing dangerous happened, the user just composed and sent an e-mail. It’s the same for the browser – it saw some Java and it just ran it, and the end user couldn’t do anything about it either.”

Dieses – in der Tat beängstigende – Szenario beschreibt im Grunde genommen einen Typus von Cross-Site-Scripting-Attacke, bei dem durch angefügtes – und nicht auf der Server-Seite gefiltertes – JavaScript der Browser anderer Nutzer kontrolliert wird. Bemerkenswert daran ist der Kontrollverlust auf Seiten der Benutzer:

- Die Adressbuchdaten sind für automatischen Zugriff verfügbar.
- Dass das Lesen von Mail eine Authentisierung benötigt, spielt keine Rolle, denn der User ist bereits authentisiert.
- Die von dem Wurm gestarteten Mails sind für den Browser lediglich Form-Posts an die gleiche Internetadresse (Yahoo) – also liegt noch nicht einmal ein Cross-Domain-Zugriff vor.
- Das Versenden der Mails findet asynchron und unsichtbar statt.
- Die asynchronen Requests werden nicht als Message-Aktivität im Browser angezeigt.

Der Browser kann offensichtlich nicht mehr erkennen, dass es um die Versendung von Mail geht. Er sieht nur einen normalen Internet Post. Was hätte auf der anderen Seite der Web-Server tun können? Wenn z. B. für das Verschicken einer Mail grundsätzlich ein Captcha (ein Akronym für *Completely Automated Public Turing Test to tell Computers and Humans Apart*, das sind die bekannten zufällig erzeugten Codes, die zwar für das menschliche Auge, (noch) nicht aber für Maschinen lesbar sind und die von vielen großen Diensteanbietern bei der Registrierung von Nutzern verwendet werden) nötig wäre (vgl. [Ram]), wäre also interaktiver Input durch einen Nutzer erforderlich, dann wäre die Attacke gescheitert – allerdings zum Preis von deutlich weniger Bequemlichkeit für den Nutzer bis hin zum Ausschluß von Sehbehinderten. Das gleiche Angriffsszenario könnte man sich im Übrigen auch in einem Wiki vorstellen: JavaScript, das an eine neue Page angefügt wird, wird beim Besuch der Seite im Browser der Nutzer ausgeführt und modifiziert weitere Seiten, trackt die Navigation des Nutzers, verschickt Mails, etc.

Zur Abwehr von Cross-Site-Scripting-Attacken gehört nichts anderes als die Erkenntnis, dass der Server für den Client-Browser und dessen Sicherheit verantwortlich ist – zumindest im gegenwärtigen Modell. Der Fehler wäre durch richtiges serverseitiges Filtern vermeidbar gewesen. Eine bessere Strategie von Seiten Yahoos wäre es gewesen, bestimmtes Script – festgelegt durch eigene URLs – zuzulassen, aber auf jeden Fall zu verhindern, dass die Nutzer eigenes Script einschleusen können (E-Bay musste bereits die gleiche Erfahrung mit Script machen, das Verkäufer zur besseren Darstellung ihrer Sites verwendet hatten: Manche dieser Scripts „verbesserten“ die Bewertungen des Verkäufers im DOM der Kundenbrowser): Beispielsweise mittels `mod_security` könnten erlaubte Scripts zugelassen und andere herausgefiltert werden.

Der zweite Angriff betraf MySpace – eine bekannte Social-Networking-Site, die vor allem von Jugendlichen gerne genutzt wird. Hier gelang es einem Angreifer, sein Profil mit Script zu ergänzen, das beim Anschauen des Profils im Browser des Nutzers ausgeführt wurde. Das Script änderte die Profileinträge des Betrachters. Die nötige Zustimmung des Nutzers zu so einer Änderung erledigte das Script unsichtbar im Hintergrund.

Auch in diesem Fall ergibt die Analyse des Angriffs wenig grundsätzlich Neues. Die Tatsache, dass die Zustimmung der Nutzer zu den Profiländerungen zwar erforderlich war, aber automatisch erteilt wurde, führt wieder zurück auf das Problem des „Trusted Path“ – wie kann sichergestellt werden, dass tatsächlich ein Nutzer seine Zustimmung gegeben hat und nicht nur eine Maschine?

In beiden genannten Fällen hätte das Script übrigens auch jederzeit Zugriff auf die Credentials (in Form von Cookies) der Nutzer für die jeweiligen Sites. Für die Betreiber der Sites folgt aus den beiden Beispielen, dass ein Framework, mit dem jegliche Form von Javascript sicher erkannt und gefiltert werden kann, unabdingbar ist. Zum Abschluss dieser Diskussion noch ein Zitat von Kai Jäger:

„Das Web ist so rasant schnell zu einer Plattform für Applikationen geworden, dass die Sicherheitskonzepte noch weit hinterherhinken und sich wohl auch nicht so ohne weiteres nachträglich integrieren lassen. Es wäre allerdings schon ein großer Gewinn, wenn für Web-Applikationen, die mit sensiblen Daten hantieren, der gleiche Aufwand zur Absiche-

rung getrieben würde, wie mit Offline-Applikationen. Die „Beta“-Unsitte der Web 2.0 Welt zeigt doch, mit welcher Gleichgültigkeit Web-Anwendungen heute noch entwickelt werden. Jetzt wo abzusehen ist, dass sich ein nicht unwesentlicher Teil zukünftiger Entwicklungen im Bereich Software im Web abspielen wird, wird es Zeit, hier auch mit der angebrachten Sorgfalt zu Werke zu gehen.“

2.5.2 Die Techniken hinter Web2.0

Aus technischer Sicht wird oft die Einführung des XMLHttpRequest-Objekts für asynchrone Kommunikation als entscheidend für Web2.0 genannt. Allerdings lassen sich solche Effekte auch mit anderen Mitteln wie z. B. dynamischen Script Tags erzielen. Uns scheinen zwei andere Dinge zentral für die Technik hinter Web2.0 zu sein:

- JavaScript kann mittlerweile in großem Stil eingesetzt werden – über Browserinkompatibilitäten hinweg und ohne, dass die Browser dabei abstürzen, wie es noch vor wenigen Jahren unvermeidbar war. Mit Web2.0 macht das Web eindeutig einen Sprung in Richtung Code und Applikation – weg von den deskriptiven Methoden der Informationsgewinnung und -darstellung in HTML.
- Die Einbettung der Inhalte von anderen Seiten sowie die Ausführung von Aktionen durch Scripts bleiben dem Nutzer fast vollständig verborgen. Das bedeutet, der Nutzer gibt die Kontrolle über Aktionen mehr und mehr an Script ab, das von externen Seiten geladen wurde.

HTML selbst hat bereits Tags mit dieser Eigenschaft. Vor allem das `IMG`-Tag führt dazu, dass ein Browser ohne Nutzerinteraktion selbständig Requests auf andere Sites erzeugt und von dort Daten holt. Diese Bilddaten werden transparent eingebettet. Dieser Mechanismus ist asynchron und kann ebenfalls zur Übermittlung von Informationen an andere Seiten genutzt werden, was z. B. die Werbeindustrie für das Tracking des Surfverhaltens der Nutzer intensiv nutzt. Die AJAX-Techniken erweitern diesen Mechanismus lediglich in Bezug auf die Daten und Protokolle, die jetzt automatisch ohne User-Input angesteuert werden können – wie der Yamanner-Wurm deutlich gezeigt hat.

Bevor wir uns den verschiedenen AJAX-Techniken und ihren Sicherheitsimplikationen zuwenden, wollen wir kurz die Grundregeln für die Sicherheit der Kommunikation zwischen Browser und Web-Server erläutern, die wir unter folgenden Stichwörtern zusammengefasst haben:

- „Same Origin“-Prinzip,
- Übernahme existierender Sessions,
- Frames und IFrames,
- Javascript kann nachgeladen werden und
- `<submit>` schickt Daten an den Server.

Mit dem „Same Origin“-Prinzip wird erreicht, dass aktive Inhalte (sprich Code) wie Javascript oder Java Applets lediglich mit dem Server kommunizieren können, von dem sie gekommen sind. Das gleiche gilt für die Behandlung von Co-

kies: Sie werden nur zur Site zurückgeschickt von der sie gekommen sind. Noch wichtiger in diesem Zusammenhang: Fremdes Script darf nicht auf Cookies einer anderen Site zurückgreifen. Dies hätte zur Folge, dass die Credentials, die eine Session darstellen (also eine SessionID enthalten), oder auch SSO-Credentials, die für einen User stehen, an fremde Sites gelangen könnten.

Dieses Prinzip verhindert auch die Brückenbildung zwischen Internet und Intranet. Wenn es nicht gelten würde, könnte Code vom Internet auf Inhalte des Intranets zugreifen und diese nach „draußen“ schaffen.

Das fremde JavaScript kann zwar den DOM-Baum der Seite manipulieren, allerdings ist es, was Cookies betrifft, auf seine Domäne begrenzt. Außerdem ist zu sagen, dass Browser JavaScript-Cookies und HTTP-Cookies voneinander abschotten, d. h. JavaScript kann keine HTTP-Cookies lesen oder manipulieren, selbst wenn diese von derselben Domäne stammen. Wenn man also z. B. eine SessionID in einem HTTP-Cookie speichert, kann das JavaScript diese nicht auslesen. Wird die SessionID jedoch an die URL der Seite angehängt, kann das fremde JavaScript diese auslesen, an einen fremden Host übertragen und dort z. B. die Session kapern. Hier sind (HTTP-)Cookies also sicherer als andere Maßnahmen.

Das zweite Prinzip „Übernahme existierender Sessions“ lässt sich am Besten anhand eines Beispiels erläutern: Im Bereich des E-Banking gibt es eine Sicherheitsregel, die besagt, dass vor der Eröffnung einer E-Banking-Session alle Browserfenster geschlossen werden sollten. Der Grund hierfür liegt darin, dass der Benutzer – während er eine E-Banking-Session eröffnet hat – in einem zweiten Fenster andere Sites besuchen könnte. Sollte sich auf einer solchen Site ein Link auf eine Form befinden, der diese Form an den E-Banking-Server schickt, so wird dieser Request automatisch vom Browser der existierenden E-Banking-Session zugeordnet. Ohne eine weitere Transaktionssicherung durch eine TAN würde diese Form vom Server ausgeführt, ohne dass der User dies wirklich autorisiert hätte. Das ist die klassische Cross-Site-Request-Forgery oder Web-Trojaner-Attacke, wie wir sie oben beschrieben haben. Sie erweist sich auf Community Sites als besonders wirksam. Ihre Voraussetzung ist nämlich, dass das Opfer einen Link des Angreifers anklickt. Und bei Community Sites sind Opfer und Täter bereits auf einer Site zusammen, womit die Wahrscheinlichkeit eines böartigen Links deutlich steigt [Heid].

Frames und IFrames eröffnen einen neuen Namensraum innerhalb des Browser-DOMs. Sie können mit anderen URLs verknüpft werden, werden jedoch im Kontext der umgebenden Seite dargestellt. Diese Einbettung kann völlig transparent sein, wie das Beispiel der LBBW oben gezeigt hat. Sie ist vom Nutzer kaum bemerkbar, da alle URLs, die im Browser sichtbar sind, noch auf die originale Site zeigen.

Von Beginn an war bei diesen Einbettungen ein kritischer Punkt das Verhältnis vom umgebenden zum eingebetteten DOM sowie die Rechte von Scripts, auf die verschiedenen DOMs zuzugreifen. Selbst ohne Einbettung von Frames oder IFrames sind immer wieder Fälle aufgetaucht, wo z. B. Script einer Site ein neu eröffnetes Popup einer anderen Site (dessen Namen bekannt war) abfangen und manipulieren konnte. Die Sicherheitsverletzungen, die durch Frames und IFrames ermöglicht wurden, sind zahllos.

Außerhalb von Frames oder IFrames erzeugt ein Zugriff auf eine normale Site-URL eine neue Seite. Lediglich Bilder werden asynchron mithilfe des IMG-Tags nachgeladen, ohne dass ein Seitenwechsel erfolgt. Aber das Gleiche gilt für spezielle MIME-Types ebenso, speziell für Javascript-URLs. Wenn durch ein Script innerhalb eines DOMs dynamisch ein JavaScript-Tag eingefügt wird, dann holt der Browser per Default die darin enthaltene URL und lädt sie. Es handelt sich hierbei um Javascript oder serialisierte JavaScript-Objekte.

Ebenfalls durch die Semantik von HTML ist das Verhalten von „submit“ bei Forms bestimmt: Wenn der Nutzer den „submit“-Button klickt, wird der Inhalt der Form an den Server geschickt.

Kommen wir nun zu den bei Web2.0 eingesetzten Techniken. Diese lauten stichwortartig:

- Cross Domain Proxy,
- Javascript on Demand,
- JSON,
- dynamisch erstelltes Script kann mit `eval()` jederzeit ausgeführt werden,
- XMLHttpRequest,
- IMG-Tag und
- unsichtbare IFrames.

Aus den oben gesagten Regeln und Verboten geht hervor, dass die browserseitige Aggregation von Daten von verschiedenen Sites ein Problem darstellt: Lediglich das IMG-Tag kann auf verschiedene externe Sites zeigen – der Browser löst die Referenzen auf und ersetzt sie durch den Inhalt der Bilder. Mit normalem HTML- oder XML-Content geht das nicht. Der Xlink-Standard hätte dies durch erweiterte Link-Tags ermöglicht, hat sich aber im Browserbereich nicht durchgesetzt – vielleicht aus Sicherheitsgründen.

Der einfachste Ausweg aus diesem Problem ist die Verlagerung der Aggregation auf die Serverseite. Klassische Beispiele dafür sind die web-basierten RSS-Reader, die Serverseitig die Nachrichten von den verschiedenen Sites heranholen und sie dem Client als aggregierte Page anbieten. Serverseitig gibt es natürlich keine Begrenzungen in Bezug auf den gleichzeitigen Zugriff auf mehrere Sites.

Javascript on Demand besteht im dynamischen (scriptbasierten) Einfügen von Script-Tags in ein DOM und dem anschließenden Laden der URL. Der Server kann dabei entweder direkt Javascript zurückliefern oder – wenn es sich mehr um Daten handelt – diese Daten in serialisierte Javascript-Objekte (JSON – JavaScript Object Notation) verwandeln und so zurückliefern. Code und Daten können auf Seite des Clients sofort weiterverarbeitet bzw. ausgeführt werden. Dazu noch einmal Kai Jäger:

„Mit On-Demand JavaScript zu arbeiten ist eigentlich recht unproblematisch: Die Arbeit beim Client übernimmt der Browser, der Server muss lediglich seine Daten als interpretierbaren JavaScript-Code bereitstellen. Da Anfragen an den Server jedoch ausschließlich über ein HTTP-GET erfolgen können, ist die Menge an Daten, die mit einer Anfrage übertragen werden können, recht beschränkt und nicht für alle Anwendungen sinnvoll. Allerdings kennt On-Demand JavaScript keine „Barrieren“ und lässt sich problemlos über Domänengrenzen hinweg einsetzen.“

Die Nutzung von On-Demand-JavaScript bemerkt der User nur anhand eines Ladebalkens in der Statuszeile. Die fremde URI wird in manchen Browsern angezeigt (in der Statuszeile), meistens jedoch nicht. Außerdem lässt sich diese Form des Cross-Site-Scripting nicht separat deaktivieren. Weil heutzutage die meisten Werbebanner über eine Abwandlung von On-Demand-JavaScript funktionieren, hätte eine solche Option auch weitreichende Folgen.

Mächtig und gleichzeitig gefährlich im Zusammenhang mit Script ist die Fähigkeit der `eval()`-Funktion, heruntergeladene Texte jederzeit als JavaScript ausführen zu lassen. Damit in engem Zusammenhang steht JSON (Java Script Object Notation), ein Format für serialisierte JavaScript-Objekte für die Datenübertragung (s. auch www.json.org). Das Format zeichnet sich unter Anderem dadurch aus, dass es sehr schnell geparkt werden kann und nur winzige Parser benötigt – wichtig für die Verwendung in Pages. Um einen JSON-Text in ein JavaScript-Objekt zu konvertieren, kann man die `eval()`-Funktion nutzen, die wiederum den JavaScript-Compiler aufruft. Da JSON eine echte Teilmenge von JavaScript darstellt, wird der Compiler den JSON-Text ohne Probleme verarbeiten und ein Objekt erzeugen:

```
var myObject = eval('(' + myJSONtext + ')');
```

Da die `eval()`-Funktion jedes JavaScript-Programm kompilieren und ausführen kann, sollte sie nur eingesetzt werden, wenn die Quelle des Programms vertrauenswürdig ist, wie es üblicherweise bei Web-Applikationen der Fall ist, bei denen der Web-Server sowohl die Webseite als auch den JSON-Code liefert. In anderen Fällen, insbesondere wenn die Daten von Clients kommen, ist es besser, einen JSON-Parser zu verwenden. Da ein solcher Parser nur JSON-Text erkennt, ist er deutlich sicherer als die `eval()`-Funktion:

```
var myObject = myJSONtext.parseJSON();
```

Der wohl bekannteste Vertreter der AJAX-Techniken ist das `XMLHttpRequest`-Objekt. Dieses Objekt hat vielfach die `IFrames` verdrängt. Es erlaubt den asynchronen Aufruf von serverseitigen Funktionen. Die Daten vom Server können anschließend dynamisch die angezeigte Page verändern. Ein erneutes Laden der gesamten Page ist nicht nötig.

Das Diagramm in Abb. 2.33 zeigt, wie mittels der `send()`-Methode ein Aufruf an den Server erzeugt wird. Dieser antwortet mit einer Response im XML-Format, die von einem Callback in Empfang genommen wird. Die Callback-Funktion modifiziert daraufhin dynamisch das Document Object Model (DOM) der angezeigten Seite.

Die Anwendung des `XMLHttpRequest`-Objekts ist sehr einfach und kann auch mit JSON kombiniert werden, wie das folgende kommentierte Beispiel von Jim Ley [Ley] zeigt (in der Tat werden momentan wohl mehr AJAX-Daten über das JSON-Format ausgetauscht als über XML).

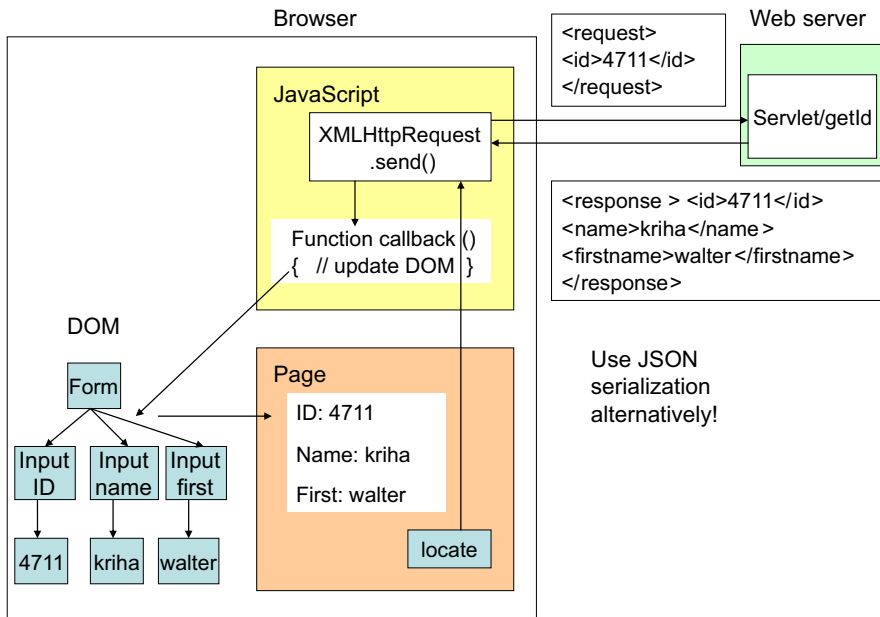


Abb. 2.33 Modifikation des DOM im Browser durch XMLHttpRequest.send()

```

Xmlhttp = new XMLHttpRequest(); // Browser specific
xmlhttp.open("GET", "/routeplanner/airport.1?LHR", true);
xmlhttp.onreadystatechange=function() { // called when re-
quest is done
    if (xmlhttp.readyState==4) {
        if (xmlhttp.status!=404) {
            // convert JSON result directly into function
            // Components can be accessed with dot or array nota-
tion
            var local=new Function("return "+xmlhttp.responseText)();
            alert("Code - Name\n"+local[0].id+' - '+local[0].name);
        } else {
            alert("Airport not found");
        }
    }
}
xmlhttp.send(null);

```

XMLHttpRequest kann also sowohl XML- als auch JSON-Objekte transportieren. Allerdings gilt auch für die Requests das „Same Origin“-Prinzip, sodass sie nur zurück zu der Site transportiert werden können, von der der Code geladen wurde.

Aber sogar über IMG-Tags lässt sich ein gewisses Maß an Asynchronität erreichen: Eine JavaScript-Anwendung kann dynamisch die URI eines Bildes ändern. Über den Query-String können dabei Daten an den Server gesendet werden. Dieser reagiert, indem er das entsprechende Bild zurückliefert.

Eine weitere Alternative, die nach wie vor zum Einsatz kommt, sind unsichtbare IFrames. Dabei erzeugt man entweder dynamisch per JavaScript oder statisch im HTML einen IFrame. Um eine Anfrage an den Server zu stellen, wird nun entweder ein unsichtbares Formular abgesendet, welches als Ziel den IFrame hat, oder man weist dem IFrame eine neue URI zu und übergibt so per GET seine Daten. Der Server reagiert darauf entweder, indem er Klartext, HTML oder XML zurückgibt, welches vom besitzenden Dokument ausgelesen werden kann (das wird durch Cross-Site-Scripting-Barrieren zunehmend schwieriger) oder es lädt eine JavaScript-Datei, welche z. B. im besitzenden Dokument eine Funktion aufruft. IFrames wurden eingesetzt, bevor XMLHttpRequest große Verbreitung fand. Sie werden heute zunehmend in sogenannten „Comet“-Anwendungen verwendet, bei der Daten vom Server quasi an den Client gestreamt werden und so der aufwändige Verbindungsauf- und Abbau eingespart werden kann.

Als Beispiel für die sich ergebenden Möglichkeiten sei hier Google Maps (vgl. auch die unter [GoMa] gesammelten Links) genannt: Google Maps verwendet XMLHttpRequest und den oben erwähnten „IMG-Tag-Trick“. Für in fremde Seiten eingebettete Karten wird Google jedoch auf XMLHttpRequest verzichten, und stattdessen On-Demand-JavaScript einsetzen. Letztlich ist das alles eine Frage der richtigen Abstraktion. Einige AJAX-Frameworks entscheiden auch automatisch, ob sie nun XMLHttpRequest, IFrames oder doch besser On-Demand-JavaScript einsetzen – für den Programmierer ist das vollständig transparent.

2.5.3 *Spezielle Aspekte der AJAX-Security*

Browsersicherheit

Einen großen Einfluss auf die Sicherheit von AJAX-Anwendungen haben zweifellos die Browser, die sich in ihren Sicherheitsmodellen jedoch teilweise deutlich unterscheiden. So ist das Sicherheitsmodell der Internet-Explorer-Linie zonenbasiert, während die Mozilla-Linie basierend auf Privilegiendefinitionen arbeitet (s. Abb. 2.34).

Dies hat zur Folge, dass mozillabasierte Browser jeden Request einzeln bewerten können, während die Erlaubnis im Internet Explorer an die jeweilige Zone geknüpft und damit persistent ist. Die komplette Freigabe der Privilegien durch Einsatz von ActiveX-Komponenten erlaubt beliebige Zugriffe im Hintergrund auf die lokalen Ressourcen. Für eine Diskussion der Browsersicherheit im Zusammenhang mit AJAX-Techniken siehe [Gent].

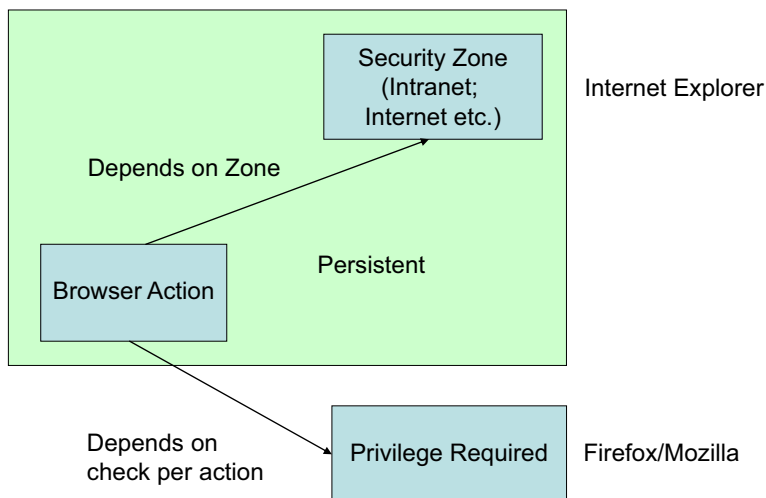


Abb. 2.34 Sicherheitsmodelle des Internet Explorer und Mozilla/Firefox

Sicherheitsprobleme von Community Sites

Community Sites sind ein großes Thema in Web2.0-Anwendungen und haben sich in letzter Zeit stark ausgebreitet. Sie stellen sicherheitstechnisch ganz neue Anforderungen, denn sie haben das Problem geänderter Vertragsverhältnisse: Statt viele User in einer $n : 1$ -Relation pro Web-Applikation haben wir jetzt $m : n$ Verhältnisse zwischen Usern, vermittelt durch den Web-Server. Die Pages auf dem Server sind alle in der gleichen Domain und zudem meist öffentlich für lesenden Zugriff. Dies ist eindeutig ein Bedrohungsszenario, das nicht durch das „Same Origin“-Prinzip entschärft werden kann. Gleichzeitig erlauben Community Sites oft eine große Zahl verschiedenster Multimedia-Formate und prüfen nicht sorgfältig genug auf eingebettete Scripts (s. Abb. 2.35). Schön wäre es vor diesem Hintergrund, wenn die Verwendung von Script nicht nur eine Benutzereinstellung im Browser wäre, sondern wenn auch der Server durch Tags klar kommunizieren könnte, dass von seiner Seite kein Script erlaubt sein sollte – selbst wenn die ausgelieferte Seite Script enthalten sollte. Das wäre dann ein „no script“-Tag, ähnlich wie die no-cache-Instruktion. Der Browser wüsste dann, dass kein Script vom Server erwünscht ist.

Die technischen Probleme von Community Sites sind aber wahrscheinlich gar nicht die wirklichen Sicherheitsprobleme dieser Sites. Ein kurzer Besuch auf einer Site wie www.mobile.de zeigt, dass die Gefahren hier eindeutig eher auf der semantischen Ebene zu suchen sind: So erscheinen beim Browsen der KFZ-Angebote auf dieser Site regelmäßig Hinweise an die Leser, dass kein Geld als Vorausleistung gezahlt werden sollte, dass bestimmte Überweisungstechniken unsicher sind, etc. Community Sites sind schlicht deshalb ein größeres Risiko für Besucher, weil sich natürlich auch sehr dumme oder gefährliche Ideen anderer

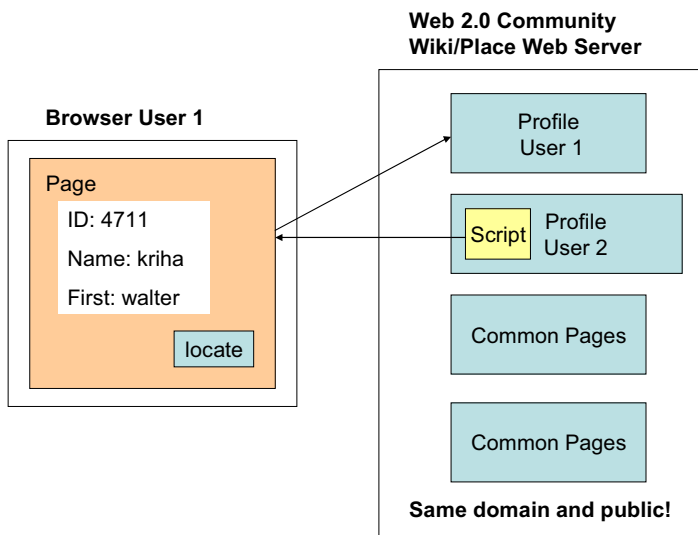


Abb. 2.35 Eingebettetes Script in Nutzerprofilen kann andere Nutzer bedrohen

hier finden lassen. Die am Anfang dieses Buches besprochene Verantwortung der Serverbetreiber für die Sicherheit der Kunden wird dadurch auch auf der semantischen Ebene wichtig und endet nicht bei der Konfiguration sicherer SSL-Parameter.

Genuine AJAX/Web2.0-Attacks

Gibt es überhaupt genuine AJAX/Web2.0-Attacks? Bisher haben wir die Meinung vertreten, dass es sich bei den meisten der bislang im Rahmen von AJAX-Anwendungen aufgetauchten Attacks nur um Spielarten bereits bekannter Attacks handelt:

- Browserfehler, die für Cross-Site-Scripting ausgenutzt werden können.
- Je mehr Sites JavaScript für Web2.0-Funktionen voraussetzen, desto mehr werden die User gezwungen, JavaScript zu erlauben.
- Forceful Browsing: Das Erraten von Dateien, die keine offiziellen Links besitzen, stellt kein auf AJAX beschränktes Problem dar.
- Web-Trojaner (Cross Site Request Forgery) sind auch ohne AJAX effektiv.
- Viele angebliche AJAX-Probleme sind wohl eher Usability-Probleme.

Fröbe schildert in [Frö] eine ganze Reihe von Angriffsmöglichkeiten, die sich vor allem auf die Verwendung von Javascript beziehen – und zwar entweder durch direkte Ausführung im Browser oder via Einbettung in Videoformate, PDF etc. (Abb. 2.36).

Er spricht insbesondere von einer Aushöhlung des „Same Origin“-Prinzips durch aktive Inhalte. Ein Punkt, der für die Existenz genuiner AJAX-Probleme

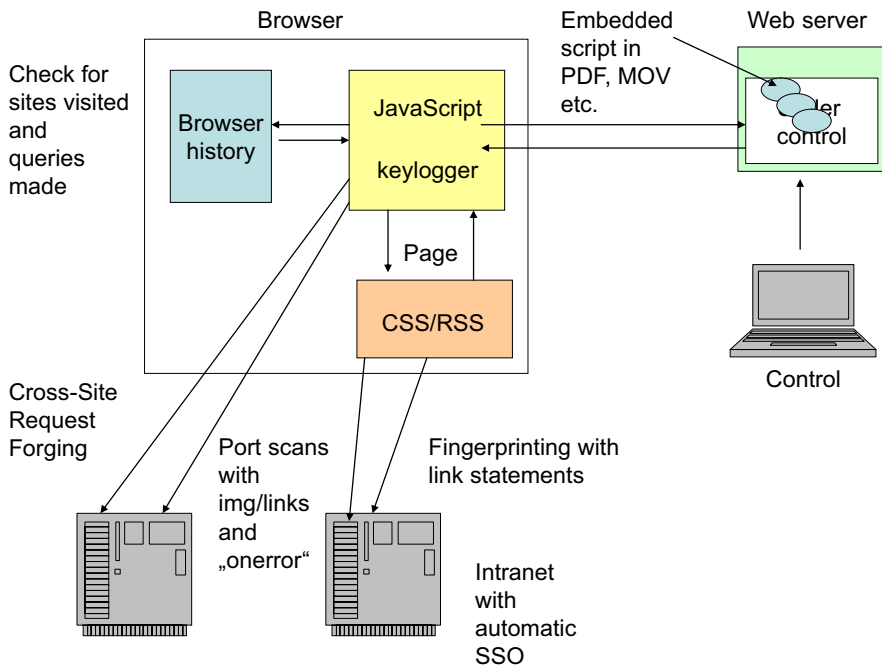


Abb. 2.36 Angriffsmöglichkeiten durch JavaScript nach Fröbe

sprechen könnte, ist die Tatsache, dass Fehler, die nur im Hintergrund auftreten, von den Entwicklern oft selbst nicht behandelt bzw. übersehen werden.

Noch bedenklicher ist jedoch ein kürzlich in [CTNW] unter dem Titel „JavaScript Hijacking“ geschilderter Angriff, der als eine der ersten genuinen Web2.0-Attacken bezeichnet wurde. Hierbei handelt es sich um das Auslesen von Daten einer fremden Site, die an den Browser des Angreifers geschickt werden. Normalerweise darf nach dem „Same Origin“-Prinzip Script einer Site nicht Daten einer anderen Site in Empfang nehmen. Es lohnt sich, diese Attacke genauer zu betrachten, da sie ein weiterer Beleg für die These ist, dass die Sicherheit browserseitig mittlerweile außer Kontrolle geraten ist. Im Sinne der sich durch dieses Buch ziehenden Diskussion von „Least Privilege“-Mechanismen in Software (POLA) ist die Attacke ebenfalls sehr lehrreich.

Zur Erinnerung hier noch einmal die bekannten Sicherheitsprobleme bei der Verwendung von JavaScript:

- Klartextübertragung von Script – besonders kritisch bei Man-in-the-Middle (MITM)-Attacken sowie beim Übergang von http auf https. Das Script kann eine Umleitung auf den MITM bewirken.
- Untypisierte Variablen können zur Laufzeit falsch gesetzt werden (Integer durch String ersetzt). Mögliche Abwehr durch Soft-Typing und Trademarking.

- Die Prototype-Funktion erlaubt die nachträgliche Änderung von bereits definierten Objekten (Hinzufügen von Funktionen, Interception von Funktionen durch eigene Erweiterungen).
- Ausführung von Strings als JavaScript-Code on-the-fly durch die `eval()`-Methode.

Mehrere dieser Eigenschaften werden durch den Angriff wie folgt ausgenutzt:

- Der User besucht mit dem Browser eine Site, die das Angriffsscript herunterlädt.
- Aus diesem Script ergibt sich ein Zugriff auf eine andere Site mittels `<script>`-Tag und einer URL, die auf einen Service zeigt, der JSON-formatierte Daten zurückliefert. Da der Zugriff vom gleichen Browser erfolgt, schadet auch eine SSL-Verbindung zwischen der angegriffenen Site und dem Browser nicht, da der Request in diese einfach eingefügt wird. Schlimmstenfalls wird der User aufgefordert, sich erneut gegenüber der angegriffenen Site zu authentisieren.
- Die zurückgelieferten Daten im JSON-Format wäre normalerweise dem Angriffsscript nicht zugänglich. Jedoch hat das Script vor dem Aufruf der angegriffenen Site zentrale Funktionen des JavaScript-Interpreters im Browser überladen. In diesem Fall ist es die Setter-Funktion für die Variable „E-Mail“. Diese ist im JSON-Format der angegriffenen Site ebenfalls enthalten. Nun wird beim Setzen der Variable der Callback, der vom Angriffsscript installiert wurde, aufgerufen.
- Der Callback sendet die abgefangenen Daten dann an den Angreifer.

Der Artikel von Chess et.al. nennt mehrere Umgehungsmöglichkeiten für diese Attacke. Unter anderem wird vorgeschlagen, dass kein sofort ausführbarer JSON-Datenstrom mehr geliefert werden soll, sondern ein Datenformat, das erst durch Client-Code in richtiges JSON verwandelt werden muss. Dies nimmt dem installierten Callback seine Wirkung. In dem Zusammenhang wird auch die Verwendung anonymer Funktionen in JavaScript empfohlen, da sie ohne Namen auch nicht einfach überschrieben werden können.

Diese Hinweise dürfen jedoch nicht darüber hinwegtäuschen, dass das ursprüngliche Problem eine globale Daten-/Funktionsstruktur innerhalb des Browsers darstellt, durch die unterschiedliche Sites gekoppelt sind. Letztlich gibt es keine wirkliche Isolation von Sites im Browser durch die Verwendung eines globalen Interpreters. Dies haben auch die Autoren erkannt und fordern in ihrem Paper eine neue Strategie für Cross-Domain-Kommunikation sowie ein „Sandboxing“ des JavaScript-Interpreters. Der letzte Punkt wäre dahingehend zu konkretisieren, dass es unterschiedliche Ausführungsumgebungen pro Site geben müsste und die Kommunikation zwischen Sites strikt dem Prinzip der Objekt-Capabilities folgen müsste. Die unsichere Behandlung von Daten und Script im Browser führt zwangsläufig zu den Empfehlungen, sich keine Authentisierung einer anderen Site aufzwingen zu lassen, wenn man eine Site besucht und vor der Verwendung wichtiger Dienste wie E-Banking den Browser komplett neu zu starten, um sozusagen böses, eingeschleustes Script zu beseitigen. Gerade der letzte Punkt sagt alles über den aktuellen Stand der Sicherheit bei Browserarchitekturen aus!

Server bzw. Application-Server die Login-Page mit einem Fehlercode taggen (im Meta-Element) und das `XMLHttpRequest`-Objekt könnte diesen überprüfen.

Insgesamt stellt dies ein schönes Beispiel dafür dar, dass ein nutzerzentrierter Modus eines Browsers doch deutlich unterschiedlich zu einem automatischen Modus wie bei AJAX ist.

Zusammenfassung

AJAX bzw. Web2.0 haben eine deutliche Lücke zwischen den mit Web2.0 beabsichtigten Möglichkeiten für Kollaboration und den Sicherheitsfähigkeiten aktueller Browser aufgezeigt. So ist es einerseits nötig, Cross-Site-Requests zu beschränken, schon um die Credentials der Sites (Cookies) zu schützen. Andererseits könnte man sich noch weitere Anwendungen der Aggregation von Daten und Services vorstellen. Ungelöst ist die Problematik der Mächtigkeit von JavaScript. Durch die Notwendigkeit permanenter Benutzerinteraktion („Wollen Sie das zulassen?“) im Namen höherer Sicherheit könnte der Vorteil moderner Web-Applikationen – die höhere „responsiveness“ – leicht wieder verloren gehen. Usability-Fragen tauchen im Zusammenhang mit Security nicht nur an dieser Stelle vermehrt auf, wie wir noch sehen werden.

Attacken wie die oben Vorgestellten haben in den letzten Jahren große Aufmerksamkeit in der Presse erzielt. Selbst den Normalbenutzern von PCs und Web-Applikationen ist heute die Gefahr eingeschleuster Malware in Form von Viren und Trojanern bekannt und der Umgang mit Virencheckern vertraut. Dies mag auch daher rühren, dass die Angriffe oft in der völligen Übernahme von Systemen oder Applikationen enden. Dennoch muss an dieser Stelle einmal nach der Rolle der Attacken innerhalb der gesamten Sicherheit eines Systems gefragt werden. Wieso sind diese Angriffe so fatal? Wieso kommt es zum völligen Zusammenbruch der Sicherheit des Systems?

Es ist nicht selbstverständlich, dass ein Fehler in der Input-Validierung von File-Pfaden automatisch zum Verlust wichtiger Dokumente führt: Selbst wenn eine „../“ Anweisung übersehen wird, muss der Zugriff auf Ressourcen z.B. außerhalb des Docroot-Verzeichnisses des Web-Servers noch lange nicht gelingen. Selbst wenn ein Editor ein mit bösartigen Makros verseuchtes Attachment einer Mail öffnet, ist dies kein Grund, dass die Makros mehr als dieses eine Attachment ruinieren können. Und selbst wenn ein Spiel wie Solitär durch einen Buffer Overflow komplett mit Schadcode verseucht wird, heißt dies nicht zwangsläufig, dass dieser Code tatsächlich auch Schaden anrichten kann.

Es muss also noch mehr dazu kommen, damit ein großer Schaden durch eine Attacke entstehen kann. Ganz allgemein und vorläufig lässt sich die Vermutung aufstellen, dass es den beteiligten Systemen an Mitteln der Schadensbegrenzung mangelt: Jede Attacke ist daher sofort fatal, wenn sie gelingt. Die folgenden Kapitel werden die Ursachen derartig mangelhafter Schadensbegrenzung auf den verschiedensten Ebenen untersuchen: Plattformen, Sprachen und Applikationsarchitekturen. Dabei werden Dinge wie schlechte Isolation, ungenügende Sprach-

eigenschaften, globale Erreichbarkeit, ambient vorhandene Autorität und andere eine wichtige Rolle spielen. Daneben werden wir Techniken vorstellen, mit denen sich innerhalb eines gesamten Systems von Plattformen und Applikationen eine Reduktion von Autorität und damit der Schadensmenge erreichen lässt. Den Anfang machen dabei Überlegungen zur Sicherheit unserer Softwareplattformen, sprich den Betriebssystemen und den darauf laufenden Servern.

2.6 Zur Psychologie der Verteidigung

Die in diesem Kapitel vorgestellten Attacken wie XSS und XSRF lassen sich technisch gut als Probleme der Input-Validierung erklären. Es gibt zahlreiche, wenn auch aufwändige Weisen, sie abzustellen. Wieso scheint Software jedoch trotzdem nicht sicherer zu werden? Nach wie vor gibt es unzählige Web-Applikationen, deren Validierungsprobleme sogar in Live-hacking-Sessions demonstriert werden können. Liegt es an der mangelnden Kenntnis der Entwickler über die Attacken? Auch darüber bestehen Zweifel, denn viele Firmen haben die Erfahrung gemacht, dass auch nach mehreren Kursen und Informationstagen für ihre Entwickler z. B. zu Problemen der Input-Validierung in der Folge weiterhin XSS-Schwächen in der Software aufgetaucht sind: Schwächen, die vielfach schon lange vorhanden waren und sogar solche, die erst nach den Kursen eingebaut wurden. Offensichtlich wurde durch die Informationstage und Kurse entweder die Wahrnehmung der Gefahr oder die Fähigkeit zur Abwehr bei den Entwicklern nicht verbessert. Beobachtet man das Verhalten nach den Kursen und Informationstagen, dann richtet sich der Verdacht eher auf die fehlende Wahrnehmung der Gefahr, denn oft kann man keine Aktivitäten zur Verbesserung der Situation im Softwareteam feststellen. Stattdessen werden die vorgestellten Attacken kurz diskutiert, der Aufwand und die Kosten kurz geschätzt, und dann wird zur Tagesordnung übergegangen.

Nun kann man dies auch zum Teil auf die Schwierigkeit schieben, vorhandene Schwächen in Legacy-Applikationen zu beheben. Wir haben dies oben diskutiert und Möglichkeiten vorgestellt, z. B. durch externe, vorgezogene Filterung. Aber ist das alles? Wieso werden dann sogar neue Schwächen eingebaut? Wieso werden zentrale Framework-Mechanismen z. B. zur Vermeidung von XSRF-Attacken durch Page-Authentisierung nicht genutzt?

Es waren solche ungeklärten Phänomene, die Volker Scheidemann dazu brachten, neben den klassischen technischen Erklärungen nach weiteren zu suchen. Basierend auf der sog. „Outrage“-Theorie wird die Risikowahrnehmung von Menschen stark durch drei Bereiche beeinflusst: Durch Eigenschaften der Quelle der Gefahr, durch den Kontext und durch persönliche Faktoren [Scheid]. Quellenbezogene Faktoren sind demnach Faktoren, die die Natur des Risikos ausmachen. Zu ihnen gehören z. B. die „Schrecklichkeit“ des möglichen Schadens, der Zeit und Ort des Eintritts, die Kontrollierbarkeit, die Identität der Opfer etc. Kontextbezogene Faktoren verändern die Wahrnehmung des Risikos dynamisch. Die Beurtei-

lungsperspektive, der persönliche Nutzen, Zeitdruck und die Freiwilligkeit des Risikos sind Beispiele dafür. Persönliche Faktoren sind Alter, Geschlecht, die Profession im Zusammenhang mit dem Risiko und nicht zuletzt die Selbsteinschätzung.

Scheidemann gelingt es, auf Basis dieser Einflüsse auf die Risikowahrnehmung zu zeigen, dass die unterschiedliche Akzeptanz von Viren-Scannern im Vergleich zur Verschlüsselung auffällig auch mit diesen Faktoren korreliert. Im Folgenden werden wir den Versuch unternehmen, den Ansatz auf die Entwickler von Web-Anwendungen anzuwenden. Spielt dort etwa auch die verzerrte Risikowahrnehmung eine Rolle?

Dämpfung der Wahrnehmung von Risiko	Einstellung zu XSS im Web-Applikation-Kontext
<i>Quellenbezogene Faktoren:</i>	
Verzögerter Eintritt des Schadens, diffuser Schadensort, entfernter Schadensort, bereits einige Zeit nichts passiert	Meist passiert ja längere Zeit gar nichts. Oft wird die Schwäche der Applikation von „white hats“ gemeldet. Der Schaden tritt konzeptuell entfernt auf (beim Opfer), die Schwächen sind „irgendwo“ im Applikationscode
Geringe „Schrecklichkeit“ des Risikos und der Risikofolgen	Meist wird ja nur die Angriffsmöglichkeit in den Medien berichtet. Über tatsächliche, noch dazu nur monetäre Schäden, erfährt man meist nichts
Hohe geglaubte Kontrollierbarkeit des Risikos oder der Folgen	Die Applikation ist vertraut. Man glaubt, Fehler schnell beheben zu können. Das steht im starken Widerspruch zur tatsächlichen Hilflosigkeit, falls eine echte Schwäche entdeckt wird. Die Behebung dauert tatsächlich viel länger als geglaubt
Geringe moralische Verwerflichkeit	Es handelt sich nur um Softwareschwächen, die finanzielle Angriffe ermöglichen
Man ist nicht selbst unter den Opfern	Es trifft unbekannte, für die Entwickler anonyme Opfer, nämlich die Kunden, und außerdem nur eine eng begrenzte Zahl von ihnen
Nur Erwachsene sind betroffen	Kunden von Shops und Finanzdienstleistern sind Erwachsene
Folgen sind wiedergutmachbar	Die Softwareschwäche kann behoben werden. Ein eventueller finanzieller Schaden wird ausgeglichen von der Firma, nicht von den Entwicklern
Kein vom Menschen verursachtes Risiko	Nein
Das Risiko ist fair verteilt	Es kann jeden der Kunden treffen, sogar die Entwickler, wenn sie als Kunden auftreten
<i>Kontextbezogene Faktoren:</i>	
Nur die Sicherung des Status Quo möglich, nur Auswahl zwischen mehreren Negativen möglich (Beurteilungsperspektive). Größere Risikoakzeptanz im Verlustbereich	Die Beseitigung der XSS-Schwächen ist sehr aufwändig und ist nicht nur durch Installation einer Applikation machbar. Der Aufwand steht in Konkurrenz mit dem Aufwand für weitere Businessfunktionen. Bestenfalls wird mit dem großen Aufwand erreicht, dass weiterhin nichts gemeldet wird bzw. nichts passiert

Dämpfung der Wahrnehmung von Risiko	Einstellung zu XSS im Web-Applikation-Kontext
Geringer persönlicher Nutzen durch die Risikobekämpfung	Entwickler haben keinen persönlichen Nutzen davon, wenn die grundsätzlichen XSS-Schwächen beseitigt werden
Voreingenommenheit	Hat nicht jede Software Fehler? Meine Software ist so komplex, dass ich mich kaum noch auskenne, wie soll da ein Angreifer leichte Angriffspunkte finden? (Diese Perspektive vergisst, dass der Angreifer ein grundsätzliches anderes – einfacheres – Bild der Applikation serviert bekommt)
Zeitdruck	Web-Developer stehen im Vergleich z. B. mit Entwicklern von Business-Logik meist unter einem hohen Zeitdruck, da neue Funktionen vom Business kurzfristig verlangt werden. Dies führt dazu, dass Risiken vernachlässigt werden
Risiko wird freiwillig auf sich genommen	Nein
Kein Vertrauen in warnende Institutionen	Institutionen wie BSI, heise.de etc. werden respektiert, aber eher als professionell Gleichgestellte, denn als Quelle von Autorität
<i>Persönliche Faktoren:</i>	
Männlich und/oder älter	Meist sind die Entwickler Männer, heutzutage auch meist älter
Professioneller Umgang mit dem Risiko	Als Entwickler von Software sind die Entwickler natürlich an einen professionellen Umgang mit Softwarefehlern gewohnt. Die hohe Zahl von Bugs mindert die Relevanz des einzelnen Fehlerfalles
Selbsteinschätzung mit unrealistischem Optimismus	„It won't happen to me“ ist weit verbreitet, zumal meist, wie gesagt, zunächst nur die XSS-Schwäche festgestellt wird. Selbst wenn der Entwickler nicht sagen kann, auf welche Weise Input-Validierung innerhalb der eigenen Applikation durchgeführt wird, glaubt er an „das framework macht das“ oder „die DMZ macht da bestimmt etwas“

Bis auf zwei Punkte passt die Lebenswelt der Web-Entwickler ganz gut mit den Kriterien für eine gedämpfte Wahrnehmung von XSS-Risiken zusammen. Man müsste jetzt natürlich genauer untersuchen, wie die Wahrnehmung von Risiko mit der Motivation, etwas dagegen zu unternehmen, zusammenhängt. Hier gehen wir momentan schlicht von der Annahme aus, dass aus einer geringeren Wahrnehmung auch eine geringere Motivation zur Bekämpfung des Risikos folgt.

Für das Softwaremanagement ließe sich mit diesen Ergebnissen eine Erklärung für die seltsame Apathie gegenüber Problemen der Input-Validierung in Web-Applikationen basteln. Die objektiven, technischen Probleme der Beseitigung werden verstärkt durch die geringe Risikowahrnehmung und -Einschätzung gegenüber XSS-Attacken.

Aber ist das eine unrealistische Einschätzung des Risikos? Oder ist die niedrige Einschätzung des Risikos und die damit verbundene geringe Motivation, es zu bekämpfen, nur für diejenigen problematisch, die gerne eine Verschiebung der

Einschätzung wünschen? Der Soziologe Michael Zwick betont in seinen Arbeiten zu Risiko und Sicherheit zwei Punkte: Risiko ist erstens eine gesellschaftliche Konstruktion. Es gehen darin zweifellos auch persönliche Faktoren ein, aber im Grunde gibt es eine gesellschaftliche Auseinandersetzung darüber, was und in welcher Größenordnung als Risiko anerkannt werden soll.

Der zweite Punkt, den Zwick beobachtet hat, ist der, dass die Risikoeinschätzung der Allgemeinheit meist keineswegs so „unrealistisch“ ist, wie dies häufig dargestellt wird. So wägt die Allgemeinheit generell auch die mit den Risiken verbundenen Vorteile ab (Beispiel Kernenergie). Und auch noch so viele Warnungen vor den Gefahren des Internets bringen die Leute nicht zur Aufgabe der Internetaktivitäten.

Und wie „realistisch“ ist nun die Einstellung der Entwickler gegenüber den XSS-Schwächen und anderen Problemen der Input-Validierung? Noch halten sich die echten Schadensfälle in Grenzen. Schlimmstenfalls steht die Firma einige Zeit am Pranger. Noch dürfte es für den Entwickler lohnender sein, neue Funktionen zu implementieren, statt ein aufwändiges Re-Design der Validationsarchitektur durchzuführen.

Aus den obigen Überlegungen kann das Softwaremanagement lediglich die Erkenntnis mitnehmen, dass es nicht die wahrgenommenen Risiken der XSS-Schwächen sind, die Entwickler zu einer anderen Haltung bringen. Wie kann diese Haltung aber geändert werden? Dazu bieten sich zwei Alternativen an: Einzelne Faktoren können manipuliert werden. So könnte die „Schrecklichkeit“ des möglichen Schadens dadurch erhöht werden, dass in der Presse genauestens über die finanziellen Schäden einer Attacke berichtet wird (was wohl kaum im Interesse der betroffenen Firmen wäre). Die Firma könnte vertraglich die Entwickler für derartige Schwächen, wenn sie gefunden werden, in finanziellen Regress nehmen – und gleichzeitig und öffentlich Penetration-Tester beauftragen. Damit kann die persönliche Betroffenheit gesteigert werden („Not in my backyard“) und damit die Risikowahrnehmung.

Alles dies sind relativ schwierige und unpopuläre Maßnahmen. Vielleicht wäre den Firmen auch anzuraten, nicht an den Risikofaktoren zu drehen und stattdessen zusätzliche Anreize in Form von Belohnungen bei gefundenen Schwächen zu schaffen. Dazu eignen sich interne Wettbewerbe, bei denen nur die Systeme der Kollegen angegriffen werden dürfen. Natürlich muss dazu extra Zeit reserviert werden, was hohe Kosten verursacht.

Manche Firmen lagern auch die Angriffsversuche schlicht an externe Firmen (sog. Pen-Tester) aus. Dies ist durchaus ein sinnvoller Weg, hat aber ebenfalls Grenzen. Aus Kostengründen können externe Pen-Tests nicht im Zyklus der Softwareänderungen durchgeführt werden, sondern seltener. Dadurch entstehen größere Perioden, in denen die Applikation auf ihre eigene Sicherheitsarchitektur sowie internes – meist funktionales – Testen angewiesen ist.

Außerdem „gewöhnt“ sich ein Softwareteam schnell an die externe Durchführung von Pen-Tests und fängt an, auf deren Ergebnisse zu vertrauen. „Die werden schon Schwächen finden, wenn welche da sind, und dann können wir sie fixen“ ist eine mögliche Reaktion der Entwickler, die die Verantwortung verschiebt und

damit die eigene Risikowahrnehmung noch weiter schwächt. Der Aufwand für eine grundsätzliche Lösung z. B. der XSS-Schwächen wird dadurch relativ zu den sonstigen Aufgaben noch höher und unattraktiver.

Welche weiteren Maßnahmen bieten sich an? Es hilft enorm, wenn Schwächen der Input-Validierung vom Team als Architekturschwächen und damit unprofessionelles Entwickeln wahrgenommen werden. Ein erster Schritt dorthin ist die Erstellung eines Architekturdokuments, das die Situation analysiert und darstellt, welche technischen Maßnahmen der Mitigation implementiert wurden.

Die Entwickler verschiedener Teams müssen miteinander ins Gespräch kommen. Dazu dienen praxisnahe Workshops, bei denen die Entwickler selbst ihre Probleme und Lösungen zur Diskussion stellen. Daraus ergibt sich oft die Möglichkeit des re-uses von Filtern etc. Security-Spezialisten spielen hier nur eine helfende Rolle. Auf weiteres drastisches Ausmalen der Risiken sollten sie nach den obigen Ergebnissen besser verzichten.

Der Austausch von Entwicklern zur Analyse der Lösungen anderer Teams ist sehr empfehlenswert. Die damit verbundene Änderung des Blickwinkels führt schnell zu einem klaren Blick auf die Fehler. Die Tatsache, dass der Analysierende selbst Entwickler ist, hilft der Aufnahme der gefundenen Probleme im anderen Team enorm.

Im Grunde genommen bestätigt sich hier wieder die alte Aussage von Bruce Schneier, dass Sicherheit ein Prozess ist. Und es wird klar, dass der Bau sicherer Systeme nicht nur hohe technische Ansprüche stellt, sondern ein Verständnis der Sichtweisen aller Beteiligten erfordert.

Literatur

- [A-I3] Homepage der Arbeitsgruppe Identitätsschutz im Internet an der Ruhr-Universität Bochum, <https://www.a-i3.org/>
- [BoDa] M. Bond, G. Danezis, A pact with the Devil, Technical Report, University of Cambridge, Computer Laboratory, June 2006, <http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-666.pdf>
- [Cobl] N. Coblenz, Presentation Layer Output Encoding <http://nickcoblenz.blogspot.com/2008/08/presentation-layer-output-encoding.html>.
- [CTNW] B. Chess, Y. Tsipenyuk O'Neil, J. West, Javascript Hijacking, <http://www.net-security.org/article.php?id=995>
- [EPR] F. Ebert, B. von Prollius, A. Retter, Web2.0, Studienarbeit an der Hochschule der Medien, Stuttgart, http://www.hdm-stuttgart.de/~bv007/web20/web20_ajax_ror.pdf
- [Esp] Esper, Complex Event Processing in Java
- [Frö] B. Fröbe, Welche neuen Gefahren kommen mit Web 2.0 auf uns zu? Der Netzwerk Insider, März 2007, Seite 18ff.
- [Gart] Gartner's 2006 Emerging Technologies, <http://www.gartner.com/it/page.jsp?id=495475>
- [Gauc] S. Gauci, Surf Jacking – HTTPS will not save you, Aug. 2008, <http://enablesecurity.com/2008/08/11/surf-jack-https-will-not-save-you/>
- [Gent] K. Genthe, Ajax in the Enterprise, Diplomarbeit an der HdM 2007, http://www.kriha.de/krihaorg/dload/ajax_in_the_enterprise.pdf

- [GoMa] Linksammlung zu Google Maps:
<http://www.google.com/apis/maps/documentation/>,
http://www.mapki.com/wiki/Main_Page,
<http://www.mapmap.org/googlemaps/>,
<http://www.pragmaticprogrammer.com/titles/sdgmapi/>
- [Goog] Google Ratproxy, Michael Zalewski,
<http://code.google.com/p/ratproxy/wiki/RatproxyDoc>
- [Guttm] Peter Gutmann, Security Usability Fundamentals, Draft 2008,
<http://www.cs.auckland.ac.nz/~pgut001/pubs/usability.pdf>
- [Heid] M. Heidrich, CSRF demystified, www.gnucitizen.org
- [Hunt] G. Hunt et al., An Overview of the Singularity Project,
<ftp://ftp.research.microsoft.com/pub/tr/TR-2005-135.pdf>
- [Hus] S. Huseby, Sicherheitsrisiko Web-Anwendung, dpunkt-Verlag 2004.
- [Jäger] K. Jäger, Ajax in der Praxis, Springer Verlag 2008
- [Kriha] W. Kriha, Buffer Overflows – How they work, <http://www.kriha.de/krihaorg/docs/lectures/security/bufferoverflow/bufferoverflow.pdf>
- [IPTables] IPTables Online Tutorial,
<http://www.boingworld.com/workshops/linux/iptables-tutorial/>
- [Lehr] T. Lehr, Fuzzing, in: Hacking und Hackerabwehr, Technical Report des Instituts für Telematik der Uni Karlsruhe, <http://doc.tn.uka.de/2007/TM-2007-1.pdf>
- [Ley] J. Ley, Using the XML http Request Object,
<http://jibbering.com/2002/4/httprequest.html>
- [Luck] D. Luckham, The Power of Events
- [Mit] K. Mitnick, W. Simon, Die Kunst der Täuschung, mitp-Verlag 2006
- [Moor] H.D. Moore, Metasploit Project, <http://metasploit.com/>
- [Müller] Th.Müller, Benjamin Mack, Web Exploit Finder,
http://www.xnos.org/fileadmin/labs/wef/Whitepaper_WEF_Automatic_Drive_By_Download_Detection_English.pdf
- [New04] Ted Neward, Effective Enterprise Java, Addison-Wesley 2004
- [Nutt] C. Nuttall, The hidden flaws in Web 2.0,
<http://www.ft.com/cms/s/e7bab6d6-2636-11db-afa1-0000779e2340.html>
- [OR] T. O'Reilly, What is Web 2.0 – Design Patterns and Business Models for the Next Generation of Software,
<http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html>.
- [Ram] A. Raman, Create an anonymous authentication module,
<http://www.javaworld.com/javaworld/jw-03-2005/jw-0307-captcha.html>
- [Resc] E. Rescorla, SSL and TLS: Designing and Building Secure Systems, Addison – Wesley 2000
- [RFC4279] P. Eronen, H. Tschofenig, Pre-Shared Key Ciphersuites for Transport Layer Security (TLS), <http://www.ietf.org/rfc/rfc4279.txt>
- [Rist] I. Ristic, Introducing mod_security,
http://www.onlamp.com/pub/a/apache/2003/11/26/mod_security.html
- [Roth] S. Roth, Evaluation von Web Application Firewalls, Diplomarbeit im Studiengang Medieninformatik der HDM Stuttgart,
<http://www.kriha.de/krihaorg/dload/university/security/roth.pdf>
- [Scheid] V. Scheidemann, It won't happen to me – Aspekte der Risikowahrnehmung und ihr Einfluss auf die IT-Sicherheit, in: Innovationsmotor IT-Sicherheit, Tagungsband zum 10. Deutschen IT-Sicherheitskongress, BSI 2007
- [Schlott] S. Schlott, Web Application Security, CCC Ulm
<http://www.cccs.de/wiki/pub/Main/VorTraege/cccs-web-security.pdf>
- [Schmidt] J.Schmidt, Wie Skype und Co. die Firewalls austricksen,
<http://www.heise-security.co.uk/articles/82481>

- [SD] A.Sotirov, M.Dowd, Bypassing Browser Memory Protections – Setting back browser security by 10 years, <http://taossa.com/archive/bh08sotirovdowd.pdf>
Dazu auch: <http://www.heise.de/security/Die-Rueckkehr-der-Pufferueberlaeufer--/news/meldung/114054>
- [Vela] R.Velasco, G.Vicente, Are Java Web Applications Secure? <http://www.theserverside.com/tt/articles/article.tss?l=AreJavaWebApplicationsSecure>
- [Vogt] D. Vogt, Wurzelbehandlung. Symlink-Fehler und Race-Conditions in Shell-scripts, Linux Magazin 04/05, S. 64ff.
- [WAS] WebSphere Application Server Information Center,
<http://publib.boulder.ibm.com/infocenter/wasinfo/v5r1/index.jsp>
- [W3] W3 Cross-Site Access Control Specification 2008, www.w3.org

Kapitel 3

Grundprinzipien des Designs sicherer Systeme

Die Hersteller von Rechnersystemen stehen heute vor der Problematik, mehrere Erfolgsfaktoren ihrer Software gleichberechtigt sicherzustellen. Dazu gehört natürlich die Sicherheit, aber auch die Flexibilität (d.h. Erweiterbarkeit und Korrigierbarkeit), die Performance und nicht zuletzt auch die Robustheit im Fehlerfall.

Sie sind unterschiedlich erfolgreich dabei. So erreicht kaum ein Hersteller eine wirkliche Robustheit des Systems im Falle von Softwarefehlern. Reboot und/oder Re-Installation sind hier die üblichen Designmittel. In Fällen, wo das schwierig bis unmöglich ist (z. B. dem Automobilbau), versucht man, die fehlende architektonische Robustheit durch exzessives Testen auszugleichen.

Eine hohe Performance wird gerne durch die Verwendung „schneller“ Programmiersprachen wie C und C++ erreicht – leider auf Kosten geringerer Robustheit und einer hochgradigen Anfälligkeit für sog. Buffer-Overflow-Attacken mit anschließender Übernahme des Systems durch den Angreifer.

Die Flexibilität wird durch die Verwendung von Plug-In-Konzepten wie bei Eclipse oder durch den nachträglichen Download von Code erreicht, sei es um Funktionalität zu erweitern oder Korrekturen (Patches) durchzuführen. Typischerweise erlangen Plug-Ins volle Kontrolle über das System – ihnen wird also absolut vertraut. Das gleiche gilt für Patches, die häufig durch die digitale Signatur des Herstellers abgesichert sind. Die Signatur von Software bindet zwar den Hersteller an die Identität der Software, sie stellt aber keinesfalls eine technische Begründung für das Vertrauen in diese Software dar. Genauso wenig erhöht eine Signatur die Sicherheit des Codes oder seine Qualität.

Kommt jetzt noch Sicherheit als Kriterium hinzu, dann kann es passieren, dass die Balance zwischen den Erfolgsfaktoren nicht mehr richtig gelingt: Manche Hersteller, gerade im Automobilbau, scheuen davor zurück, ihre Rechner für Erweiterungen durch andere Hersteller zu öffnen. Andererseits gibt es eine Menge von Komponenten, die beim Fahrzeug austauschbar sind oder ergänzt werden können, ohne dass sich jemand Sorge um die Sicherheit des Systems „Auto“ machen würde. Die Sorge vor gegenseitiger Beeinflussung von Steuerungssoftware führt

hingegen zu einer riesigen Zahl von Controllern in Fahrzeugen und damit zu hohen Kosten. Gerne würde man die Software wieder auf einem Zentralrechner konzentrieren – wenn dies die Anforderungen „Robustheit“ und „Sicherheit“ erlauben würden. Genauso attraktiv wären offene Entertainment-Systeme, wenn die Sicherheit und Funktionsfähigkeit bei Nutzung mobiler Software garantiert werden könnte.

Teilweise treffen die Hersteller in diesem Zusammenhang seltsame Entscheidungen. So konnten wir in den letzten Jahren feststellen, dass auf vielen Smartphones auf der Basis von Symbian OS und Java die Möglichkeiten der Ansteuerung von kritischen Komponenten (SIM-Karte, Bluetooth etc.) bei Java-Applikationen verhindert wurde. Der Download von in C geschriebener Software war jedoch ohne Probleme möglich.¹ Gerade C-Programme lassen sich jedoch auf solchen Plattformen so gut wie gar nicht in ihren potenziellen Rechten und Zugriffsmöglichkeiten einschränken, was die Einhaltung des POLA-Prinzips (s. unten) unmöglich macht und der Sicherheit dieser Systeme natürlich abträglich ist.

Dieses Kapitel soll dazu dienen, die Problematik sicherer Systeme besser zu verstehen und Softwareentwickler darauf vorzubereiten, dass hier in den nächsten Jahren massive Änderungen auf allen Ebenen zu erwarten sind. Dazu wenden wir uns nun einigen Grundprinzipien zu, die bereits vor überraschend langer Zeit formuliert wurden: Saltzer und Schroeder haben bereits 1975 in ihrem Paper zu „The Protection of Information in Computer Systems“ [SaSch] einen Grundstein für das Design sicherer Systeme gelegt. Heute ist das Paper Pflichtlektüre für jeden, der sich mit sicheren Systemen beschäftigt. Dieser Rückgriff auf „altes“ Wissen ist in der Wissenschaft ganz normal – lediglich die Informatik scheint sich damit schwer zu tun. In der letzten Zeit scheint sich das jedoch zu ändern: So greifen Vertreter der Capability-Bewegung (siehe www.erights.org) gerne auf ältere Schriften zurück und zeigen, wie einseitig die Entwicklung von Zugriffskontrollen ausschließlich auf Basis von Access Control Lists wirklich ist. Peter van Roy und Seif Haridi entdecken in ihrem grundlegenden Werk zu konzeptbasierter Programmierausbildung [RH] längst vergessene Konzepte wie deklarative Concurrency und funktionale Techniken neu und zwischen beiden Gruppen wird der Beitrag dieser Konzepte in Bezug auf sichere Systeme diskutiert.

Es könnte sein, dass diese scheinbar längst irrelevant gewordenen Paradigmen durchaus einen großen Beitrag zur Sicherheit heutiger Systeme leisten könnten – falls man sie denn noch kennen würde. Wir greifen diese Gedanken weiter unten in diesem Kapitel zu den Grundprinzipien sicherer Software wieder auf, und wollen hier nun eine eigene Interpretation und gegebenenfalls Erweiterung der Grundprinzipien von Saltzer und Schroeder im Hinblick auf gegenwärtige Probleme der Sicherheit in Systemen geben. Im Anschluss daran wollen wir die drei für uns zentralen Begriffe *Principle of Least Authority* (POLA), *Ambient Authority* und *Permissions* definieren, da sie für die folgenden Kapitel von fundamentaler Bedeutung sind.

¹ Symbian ist mittlerweile sehr restriktiv geworden und setzt eine zentrale Signatur für die Instruktion von C++-Code voraus.

3.1 Economy of Mechanism (KISS)

Das KISS (Keep the design as simple and small as possible) Prinzip ist in den Ingenieurwissenschaften eine selbstverständliche Forderung. Die Begründung dafür von Saltzer und Schroeder im Zusammenhang mit der Sicherheit von Softwaresystemen ist recht interessant: Ungewollte Zugriffspfade können nur durch zeilenweise Codeinspektion entdeckt werden und die wiederum setzt voraus, dass Designs einfach und klein sein sind.

Das könnte man heute schärfer formulieren: Designs müssen so beschaffen sein, dass sie abgeschlossene Bereiche enthalten, über die konkrete und sichere Aussagen über Zugriffsmöglichkeiten und Effekte getroffen werden können. „Zugriffsmöglichkeit“ ist dabei als grundsätzliche Erreichbarkeit von Ressourcen zu verstehen, auf denen Effekte erzielt werden können. Eine zeilenweise Analyse ist angesichts heutiger Codemengen nicht mehr möglich und hat wahrscheinlich noch nie funktioniert, wenn man die Geschichte der Buffer-Overflows betrachtet (wie oft wurde nach wiederholten Inspektionen der dns-bind-code als frei von Overflows erklärt?). An ihren Interfaces müssen Komponenten ihre grundsätzlichen Zugriffspfade sichtbar machen. Entscheidend ist dabei die *Granularität* der Isolation: Grobe Granularität bedeutet riesige Zonen gleichen Gefahrenpotenzials, z. B. die ganze Applikation. Damit ist eine Sicherheitsabschätzung nicht mehr möglich. Die Isolation von Komponenten oder Modulen innerhalb der Applikation und voreinander sorgt jedoch dafür, dass nicht alle Bereiche eines Systems in Bezug auf die Effekte eines Inputs untersucht werden müssen. Seiteneffekte hingegen erschweren die Analyse enorm. Ein Beispiel für solche „Ripple-Effekte“ ist die Speicherverwaltung in C oder C++:

```
char * mem = getSomething(); // foreign code
putSomething(mem) // foreign code
```

Der Entwickler steht hier vor dem Problem, dass er prüfen muss, ob er nach Aufruf von `getSomething()` verantwortlich für die Freigabe geworden ist. Und beim Aufruf von `putSomething()` stellt sich die Frage, ob die aufgerufene Funktion die Verantwortung übernimmt oder nicht. In jedem Fall schließt eine defensive Programmierung hier die Analyse zweier fremder Funktionen ein – es gibt keine Lokalität von Entscheidungen. Pointer-Zugriffe, globale Variablen oder Datenstrukturen zerstören ebenfalls Lokalität und erzeugen unklare Zugriffspfade.

Funktionen in funktionalen Sprachen oder die Verwaltung von virtuellem Speicher von Prozessen haben hingegen diese isolierende Eigenschaft. Anders gesagt: Hat ein Modul grundsätzlich einen Zugriffspfad (sei es durch globale Referenzen, Pointer, oder durch Systemcalls etc.), dann ist die Frage, ob es diesen Zugriffspfad während seiner Laufzeit verwenden wird, prinzipiell unentscheidbar, da sie auf das Halteproblem der theoretischen Informatik reduziert werden kann. Das aber bedeutet praktisch, dass eine genaue Analyse der Gefahr durch dieses Modul mit formalen Methoden nicht möglich ist. Somit können bei fehlender, feingranularer Isolation weder der Praktiker noch ein Analysewerkzeug (Modell-Checker) eine klare Aussage zum Gefahrenpotenzial machen.

Das erste Prinzip von Salzer und Schröder stellt somit die gegenseitige Abhängigkeit von Softwarearchitektur und Sicherheitsanalyse fest. Der Leser kann diese Abhängigkeit weiter unten im Kap. 12 zu „Browsersicherheit durch Object Capabilities“ exemplarisch nachvollziehen.

3.2 Fail-Safe-Defaults

Zugriffsentscheidungen sollten aufgrund von expliziter Erlaubnis (*permission*) und nicht aufgrund von Verboten gefällt werden. Dieses Prinzip ist heute auch als „Default-is-deny“ benannt. Es sollte z.B. bei der Installation von Software oder Firewalls, die jeweils in ihrem ursprünglichen Zustand Sicherheit garantieren müssen, angewendet werden. Aber auch White-Lists zum Filtern von erlaubten Requests in Web-Applikationen gehorchen diesem Prinzip.

Die Implementation dieses Prinzips ist hingegen nicht so selbstverständlich: Wieso muss man beim Paketfilter IPTables unter Linux die „Default-is-deny“-Option extra setzen? Dieser Text wird auf einem Laptop geschrieben, in dessen Betriebssystem ein Bildschirmschoner mit Passwortabfrage konfiguriert ist. Die Zeit bis zu seiner Aktivierung ist relativ kurz gewählt. Leider wird der Bildschirmschoner in Abhängigkeit von bestimmten Applikationen, Zuständen etc. regelmäßig nicht aktiv und das System bleibt damit ungeschützt. Im Zusammenhang mit Single-Sign-On-Authentisierungen kann dies ein fataler Fehler sein.

Auf Softwarearchitektur angewandt, erkennt man schnell viele Verstöße gegen das Prinzip. Global navigierbare Datenstrukturen wie File-Systeme, Verzeichnisse oder die Gesamtheit der Systemcalls, die einem Programm immer zunächst zur Verfügung stehen, implementieren keinesfalls eine „Default-is-Deny“-Logik. Auch hier muss man heute fordern, dass Zugriffsentscheidungen nicht nur auf der Basis von expliziten Permissions erlaubt sind, sondern die Permissions müssen zudem minimal sein, also angepasst an den jeweiligen Zweck.

3.3 Complete Mediation

Jeder Zugriff auf jedes Objekt muss vorab auf hinreichende Autorität geprüft werden. Dies ist wahrscheinlich eines der komplexesten Prinzipien von Saltzer und Schroeder. Es lohnt sich deshalb, hier die originalen Forderungen noch einmal kurz aufzuführen:

“This principle, when systematically applied, is the primary underpinning of the protection system. It forces a system-wide view of access control, which in addition to normal operation includes initialization, recovery, shutdown, and maintenance. It implies that a foolproof method of identifying the source of every request must be devised. It also requires that proposals to gain performance by remembering the result of an authority check be examined skeptically. If a change in authority occurs, such remembered results must be systematically updated.”

Der Zugriff auf ein Objekt muss also

- über die Zeitachse hinweg verfolgt werden,
- über Sicherheitszonen mit unterschiedlichem Vertrauen hinweg verfolgt werden,
- über die gesamte Kette der Zugreifenden hinweg verfolgt werden und
- eventuelle Änderungen in der Autorisierung bezüglich eines Objektes müssen unmittelbar allen Instanzen mitgeteilt werden, die sich bereits eine Entscheidung anhand der ursprünglichen Rechte gemerkt (sprich: in einem Cache gespeichert) haben.

„Über die Zeitachse hinweg verfolgen“ bedeutet, dass ein Objekt eine Sicherheitszone verlassen kann – z. B. auf einem optischen Medium für Backup-Zwecke gespeichert werden kann – und dennoch keine illegalen Zugriffe gestattet sein dürfen. Bei der Rückkehr des Objektes in die ursprüngliche Sicherheitszone sind möglicherweise erneute Prüfungen nötig.

In vielen Sprachen kann ein Objekt auch in serialisierter Form in eine Datei gespeichert werden. Damit verlässt es eine Sicherheitszone, kann danach auf einem anderen System manipuliert werden und anschließend wieder in das Ursprungssystem eingelesen werden. Diese Änderungen passieren außerhalb der Kontrolle des Ursprungssystems. Complete Mediation bedeutet, dass durch solche Änderungen die Sicherheit des Systems nicht beeinträchtigt werden darf. Dies ist beispielsweise der Grund, warum spezielle Rechte eines Files beim Lesen von einem Wechselmedium unter Unix ausgeblendet werden.

Die gesamte Kette von Aufrufern einer Aktion zu verfolgen, ist lokal innerhalb eines Prozesses nicht ganz so schwierig, wenn auch sehr aufwändig. Wir werden bei der Diskussion der Java-2-Security-Mechanismen sehen, wie ein `AccessControlContext`-Objekt als Liste aller aufrufenden Objekte inklusive ihrer Rechte erstellt werden kann. Java verwendet hier einen Stack-Walkback-Mechanismus, der den Aufruf bis zum ersten Aufrufer zurückverfolgt. Der Preis für diese komplette Kontrolle ist zum einen in geringerer Performance zu zahlen (und er ist recht hoch, ungeachtet aller Optimierungen). Zum anderen besteht der Preis in erhöhtem Design- und Wartungsaufwand, denn das Konzept bedingt, dass in unterschiedlichen Layern verschiedene Kontrollpunkte eingebaut werden müssen, die unterschiedliche Abstraktionen betreffen: Ein Client, der z. B. Mails verschicken möchte, müsste sonst Low-Level-Socket-Rechte erhalten, da diese in der Implementation nötig sind. Das Beispiel zeigt, dass Complete Mediation auch die Kapselung von Implementationen durch Interfaces betrifft: Der Aufrufer bräuchte plötzlich Rechte auf Ressourcen, die am Interface gar nicht sichtbar sind und die er dadurch gar nicht kennt. In diesem Falle wird in der Software gerne ein Mechanismus verwendet, der das Ausführen sog. „privilegierter“ Funktionen erlaubt. Wir werden die Gestaltung von Interfaces zu Softwaremodulen mit privilegierten Funktionen noch im Detail untersuchen. Hier nur als Heuristik zunächst die Anmerkung, dass derartige Funktionen nicht am Interface zum normalen Aufrufer sichtbar sein sollten, die Komponente diese Rechte aber durchaus erst zur Laufzeit erhalten sollte. Umgekehrt sollten die Funktionen, die dennoch mit den Privilegien

des Aufrufers aufgerufen werden, auch am Interface der Komponente sichtbar sein, sprich, der Komponente „geöffnet“ übergeben werden. Im Prinzip bedeutet das, dass selbst innerhalb einer Software Komponenten sich gegenseitig nicht vertrauen sollten.

Besonders interessant und schwierig wird das Prinzip der Complete Mediation aber im Falle verteilter Systeme. Erstens ist es hier noch viel offensichtlicher, dass unterschiedliche Sicherheitszonen, sprich Zonen unterschiedlichen Vertrauens, existieren. So passiert ein Web-Request während des Aufrufs die Zonen Web-Server, Servlet-Container und typischerweise auch Back-End in Form einer Datenbank oder eines EJB-Containers. An den Grenzen werden erneute Zugangsprüfungen nötig, die jedoch je nach verwendeter Technik sehr unterschiedlich ausfallen können. Zweitens ist es im lokalen Fall relativ leicht möglich, ein Zugriffskontrollsystem zu implementieren, das einen Aufruf komplett verfolgen kann, also quasi eine Draufsicht auf den Fluss des Requests durch die Komponenten (oder ähnlich wie ein Garbage Collector eine Sicht auf die allokierten Objekte) besitzt. Im verteilten Fall ist diese globale Draufsicht natürlich nicht möglich (so wenig wie es z. B. eine global gültige Zeit in verteilten Systemen gibt).

Wir werden am Beispiel von Enterprise Security weiter unten den lokalen Stack-Walkback-Mechanismus in eine verteilte Umgebung umsetzen. Sichere Delegation ist hier der entscheidende Begriff. Sichere Delegation bedingt ein komplettes Protokollieren aller Beteiligten eines Aufrufs sowie deren Aktionen in einem Sicherheitskontext, der vom Aufrufer bis zum endgültigen Ziel des Aufrufs führt. Dafür existieren sehr unterschiedliche Qualitätsstufen und bis heute existiert praktisch kein System, das Complete Mediation in der Praxis eines verteilten Systems vollständig durchsetzt. Zu groß sind die Probleme in Bezug auf Performance und Kompatibilität. Häufig wird daher in verteilten Systemen mit recht groben und großen Vertrauenszonen gearbeitet, innerhalb derer keine weiteren Zugriffskontrollen stattfinden.

Außerdem muss an dieser Stelle gefragt werden, ob Complete Mediation in dieser Form in manchen Fällen nicht sogar über das Ziel hinaus schießt oder vielleicht gar nicht der effizienteste Kontrollmechanismus ist: Betrachten wir ein Call-Target, das das komplette Protokoll aller Beteiligten und deren Rechte mitgeteilt bekommt – was kann es daraus schließen? Es kann den tatsächlichen Pfad mit den konfigurierten Pfaden vergleichen und feststellen, dass der Request des Aufrufers einen legalen Pfad gegangen ist und dass keine unbefugten Instanzen daran beteiligt waren. Dies bedeutet, den Pfad – der ja letztlich auch durch die Infrastruktur vorgegeben ist – als Teil des Kontrollmechanismus auszuwerten. Ändert sich der Pfad – und sei es nur aus Gründen der Optimierung – dann ändern sich die Rechte. Das zieht beträchtlichen Wartungs- und Konfigurationsaufwand nach sich und bindet die Infrastruktur zu sehr an die Sicherheitslogik. Wie kommt es dazu?

Die Sicht von Saltzer und Schroeder – wie auch die in der Java 2 Security – ist stark prozessorientiert: Die Autorisierung eines Zugriffs wird durch die beteiligten Komponenten und deren Reihenfolge bestimmt. Der eigentliche Sinn eines Aufrufs wird nicht mitgeteilt und bleibt nur implizit definiert durch die Reihenfolge der Komponenten. Kontrastieren wir das mit der Transaktionssignierung, in der

die Semantik einer Aktion fälschungssicher festgehalten wird: Ein Aufrufer will beispielsweise eine Kontotransaktion durchführen. Statt sich mittels PIN/TAN-Verfahren bei seiner Bank zu authentisieren und dann anschließend den Auftrag über einen sicheren Kanal an die Bank zu senden, signiert er den Auftrag und schickt ihn über eine unsichere Leitung an die Bank. Die Bank kann die Nachricht authentisieren, den Auftrag durchführen und hat gleichzeitig einen nicht-abstreitbaren Beleg für die Auftragserteilung. Welche Rolle spielen in diesem Fall sogenannte Intermediates, also gut- oder böswillige Dritte, durch deren Hände die Nachricht geht? Die Antwort ist einfach: keine. Im schlimmsten Fall kann es zu einer Unterdrückung der Nachricht kommen. Ein mehrfaches Versenden (*Replay*) einer authentischen Nachricht lässt sich mit einfachen Mitteln (Sequence Numbers) verhindern.

Es müsste also untersucht werden, wie die Forderung nach Complete Mediation bei der Verwendung von message- bzw. objektbasierter Sicherheit aussehen könnte. Zu vermuten ist, dass die Protokollierung dadurch wesentlich weniger aufwändig und damit die Performance besser wird. Zu vermuten ist aber auch, dass durch das Mehr an Semantik, welches der Call mit sich bringt, Intermediates und Targets wesentlich bessere Entscheidungen über Zugriffsrechte treffen können und eine größere Unabhängigkeit von der Infrastruktur zu erreichen ist. In jedem Fall ist diese Frage sowohl im lokalen als auch im verteilten System sehr interessant. Wir werden der Bedeutung von Complete Mediation auch im Zusammenhang mit objekt- bzw. messagebasierter Sicherheit im Kapitel zu Web-Services Security (Kap. 9) noch weiter nachgehen.

Die Semantik einer Aktion richtig auszudrücken, ist bei der Verwendung von Transaktionssignierung äußerst wichtig, denn die Empfänger der Nachricht werden ihre Autorisierung anhand der Semantik durchführen und sich nicht mehr so sehr am Überbringer orientieren. Deshalb ist die Semantik der Transaktionen letztlich auch das Wichtige an der neu entstehenden Web Service Security und nicht die Tatsache, dass sie XML verwendet und über SOAP transportiert werden kann. Das größte Problem besteht darin, eine geeignete Sprache zu finden, um die Semantik der Aktionen sowie die damit verbundenen Sicherheitsbedingungen (Policies) auszudrücken.

Abschließend bleibt zum Prinzip der Complete Mediation zu sagen, dass Zugriffsentscheidungen nicht nur anhand des „wer“, sondern auch anhand des „was/warum“ getroffen werden können. Ein Beispiel für die Verknüpfung beider Gesichtspunkte bilden die im Kapitel „Enterprise Security“ (Kap. 6) besprochenen Proxy-Zertifikate.

3.4 Open Design: The Design Should Not Be Secret

Anders ausgedrückt: Die Sicherheit einer Lösung darf nur in der Stärke ihrer Schlüssel liegen und nicht in der Geheimhaltung des Designs. Dieses Prinzip ist bei kryptografischen Algorithmen schon lange als Kerckhoff'sches Prinzip bekannt

und hat sich dort seit langem bewährt. Es auch auf das Design von Software auszu-dehnen, erscheint nur logisch. Gilt es aber auch für die konkrete Implementation?

Ja und nein. Zwar finden sich viele Sicherheitslücken in der konkreten Implementation von Systemen, Protokollen oder kryptografischen Verfahren und nicht im Design selber. Viele Attacken beginnen mit dem Sammeln von Informationen und der Suche nach einer Lücke (z.B. das sogenannte *Port Scanning*). Deshalb sollten Betriebssysteme nicht einfach Informationen über Interna preisgeben, genauso wenig wie die Netzwerkkonfiguration öffentliche Information ist. Entsprechend sollten Applikationen keine Systemdetails in Fehlermeldungen offenbaren. Sicherheitslücken in offen gelegten Implementationen können also deutlich schneller ausgenutzt werden.

Andererseits hilft aber eine öffentliche Implementation enorm, die Sicherheit zu steigern, indem Implementationsfehler durch die gutwillige „Community“ schneller gefunden werden. Dafür gibt es allerdings keinerlei Garantie, wie erneute Buffer-Overflow-Fehler in der bind-Software des DNS nach einer äußerst umfangreichen Codeinspektion wenige Monate zuvor zeigten.

Aus heutiger Sicht kann weiter ergänzt werden: Die größte Gefahr von unsicherer Software besteht bei neuen und speziellen Applikationen, z. B. gerade Web-Applikationen, die einen speziellen Firmenservice implementieren. Diese werden unter hohem Zeitdruck und meist für nur einen Auftraggeber erstellt und sind damit automatisch nicht-öffentlich. Beide Faktoren führen zu einer erhöhten Fehlerwahrscheinlichkeit.

3.5 Separation of Privilege

Wenn möglich, sollten Schutzmechanismen, die zwei verschiedene Schlüssel benötigen, eingesetzt werden, da sie robuster und flexibler sind als ein Mechanismus, bei dem ein einziger Schlüssel genügt. Dieses Prinzip ist in der Finanzwelt bekannt als „Vier-Augen-Prinzip“ (z.B. muss die Annahme eines Schecks ab einem bestimmten Betrag von zwei Personen geprüft werden). In diesem Kontext ist das Prinzip „Separation of Privilege“ aber auch in dem Sinn von „Defense in Depth“ zu verstehen.

Wichtiger aber ist ein anderer Aspekt: Bei getrennten Aufgaben sollten auch die damit verbundenen Privilegien getrennt werden, und nicht, wie häufig bei hierarchischen Rechtemodellen, die Vorgesetzten automatisch alle Rechte der Untergebenen zusätzlich zu ihren eigenen erhalten. Dies führt lediglich zur Akkumulation sinnloser Privilegien und damit zu Missbrauchsmöglichkeiten.

Das Prinzip ist gleichzeitig ein schönes Beispiel dafür, dass in der Implementation von Sicherheitsprinzipien fast nichts wirklich absolut gilt: Nehmen wir an, Sie haben in ihrer Bankensoftware zur Verarbeitung von Belegen das Vier-Augen-Prinzip erfolgreich implementiert. Am Montagmorgen in einer kleinen Filiale meldet sich eine der beiden Mitarbeiterinnen krank und nach kurzer Zeit klingelt das Telefon bei Ihnen: Wie kann man für begrenzte Zeit das Vier-Augen-Prinzip

ausschalten, damit die betroffenen Belege dennoch fristgerecht verarbeitet werden können?

Ein anderes Beispiel für Moduswechsel in Systemen sind die Probleme mit Stellvertretungen, z. B. wenn ein Mitarbeiter in Urlaub geht: Kann der Stellvertreter die Privilegien in kontrollierter Weise für eine begrenzte Zeit übernehmen?

Rollensysteme werden häufig so implementiert, dass Mitarbeiter zwar verschiedene Rollen besitzen, sie aber nicht gleichzeitig ausführen dürfen – also nicht gleichzeitig arbeiten und ihre Arbeit prüfen dürfen. Auch in diesen Fällen muss eine Implementation von Privilege Separation Rücksicht nehmen auf Krankheit, Jobwechsel, Stellvertreterregelungen etc.

Auf *Permissions* bezogen, also auf die explizite Erlaubnis für gewisse Aktionen oder Ressourcenzugriffe, ist das Prinzip von Saltzer und Schröder nichts Neues. Wichtig für das Design sicherer Systeme ist dabei vor allem die Forderung, dass *Permissions* sich nicht an einer Stelle ballen und dass sie sicher und nachvollziehbar übertragen werden können. Auf die Architektur der Softwarekomponenten übertragen, muss man ergänzen, dass Komponenten ebenfalls eine klare, sichtbare Funktion haben sollten. Sie sollten die dafür nötigen Rechte explizit und sichtbar erhalten und eine Ballung von Zugriffsmöglichkeiten innerhalb einer Komponente vermeiden.

3.6 Principle of Least Privilege (POLP)

Jedes Programm und jeder Nutzer eines sicheren Systems sollte mit der kleinsten Menge an Rechten arbeiten, die es oder er benötigt, um seine Aufgabe zu erfüllen. Aus Businesssicht geht es bei sicheren Systemen in erster Linie darum, Berechtigungen zu erteilen und zu verwalten. Je weniger Berechtigungen ein Mitarbeiter erhält, desto geringer ist die Gefahr des Missbrauchs durch den Mitarbeiter. Rollenbasierte Zugriffskontrollsysteme, die mit der sogenannten Role Based Access Control (RBAC) arbeiten, implementieren dieses Prinzip erfolgreich. Wir werden uns im Zusammenhang mit Enterprise Security mit Frameworks beschäftigen, die RBAC nutzen.

Das Prinzip POLP ist ein auf Schadensminimierung ausgerichtetes Prinzip, das zu den wichtigsten in der Architektur sicherer Systeme gehört. Umso erstaunlicher ist es, dass heutige Plattformen auf praktisch allen Stufen der technischen Realisierung, also: im Betriebssystemkern, in Applikationen, in Modulen und Objekten gegen dieses Prinzip massiv verstoßen. Durch die Verletzung dieses Prinzips entsteht das Antipattern „Ambient Authority“. Das sind Privilegien, die unnötig und frei verfügbar sind und, falls es zu einem Programmierfehler oder Missbrauch kommt, zu einem wesentlich größeren Schaden beitragen, als es eigentlich nötig wäre. Es ist wichtig, zu verstehen, dass es genau die „Ambient Authority“ in vielen modernen Betriebssystemen ist, die es Viren und Trojanischen Pferden erlaubt, großen Schaden anzurichten.

Aus heutiger Sicht muss der Ansatz von Saltzer und Schröder deshalb um das „Principle Of Least Authority“ (POLA, siehe [Mil]) erweitert werden, bei dem die Verteilung von Privilegien innerhalb technischer Principals und Komponenten betrachtet wird. Dieses Buch nimmt an vielen Stellen auf das POLA-Prinzip Bezug, und deshalb werden wir es im Anschluss an die Diskussion der Saltzer-Schröder-Prinzipien detailliert vorstellen.

3.7 Least Common Mechanism

Die Informationen und Mechanismen, die sich mehr als ein Nutzer teilen und von denen alle Nutzer abhängen, sollten auf ein Mindestmaß beschränkt bleiben. Jeder gemeinsame Mechanismus (insbesondere solche, die auch gemeinsame Variablen beinhalten), stellt einen potenziellen Informationspfad zwischen verschiedenen Nutzern dar und muss deshalb mit großer Sorgfalt entworfen werden, um die Sicherheit nicht unabsichtlich zu gefährden.

Aus heutiger Sicht versteckt sich hinter der Vermeidung gemeinsamer Mechanismen sehr viel mehr als nur die Sorge vor unerlaubten Informationsaustausch zwischen Komponenten. Gemeinsame Mechanismen haben einen negativen Einfluss auf die nicht-funktionalen Anforderungen an heutige Systeme wie z. B. Verfügbarkeit und Zuverlässigkeit. Dies wird deutlich, wenn man sich klar macht, dass der Code einer Applikation letztlich viel mehr bewirkt als durch formale Rechte oder Permissions ausgedrückt wird: Der Code nimmt Platz im Speicher ein und verbraucht evtl. dynamisch immer mehr vom RAM. Er bewirkt Netzwerkverkehr und kann somit zu Problemen mit der Verfügbarkeit und Erreichbarkeit des Systems führen. Die Ausführung des Codes erzeugt Wärme etc.

Das Prinzip, gemeinsame Mechanismen zu vermeiden, zielt letztendlich auf die maximale Isolation zwischen Komponenten und zwischen Nutzern. Dazu zählt auch eine Form der Isolation, die nicht in Form von Rechten ausdrückbar ist bzw. heute noch nicht ausgedrückt wird, wie eben die Wärmeerzeugung oder der Speicherplatzverbrauch. Zu den Problemfällen gehören:

- Gemeinsame Daten im Kernel
- Gemeinsame Daten bei Multithreading
- Application Server, die Requests verschiedenster Nutzer und Applikationen behandeln
- Globale Daten, statische Daten
- Tree-navigation
- Globale Resource Verzeichnisse
- sowie als nicht-Funktionale Aspekte:
- Speicherplatzbedarf
- CPU-Nutzung
- Wärmeerzeugung
- Netzwerkauslastung

Nicht zufällig sind dies auch Punkte, die die Zuverlässigkeit und Korrektheit von Programmen meist sehr negativ beeinflussen können, besonders, wenn Sprachen mit Seiteneffekten verwendet werden.

In den harmloseren Fällen einer Verletzung dieses Prinzips können Denial-Of-Service-Attacken einzelner Nutzer andere von der Benutzung ihrer Infrastruktur ausschließen: So z. B. bei einer Distributed-DOS-Attacke auf eine Web-Applikation, deren gemeinsamer Sicherheitsmechanismus – der Flaschenhals-Firewall – ebenfalls aus Sicherheitsgründen gewünscht ist.

Betriebssysteme haben Isolationstechniken seit Jahrzehnten eingebaut: Virtueller Speicher schützt verschiedene Prozesse voneinander (Abb. 3.1).

Quotas sind Verwaltungsalgorithmen, die verhindern, dass einzelne Nutzer oder Komponenten Systemressourcen monopolisieren können. Aber wie sieht es damit bei Applikationsservern aus? Dort können in einer Virtual Machine normalerweise die verschiedensten Applikationen von verschiedenen Nutzern gleichzeitig benutzt werden, ohne dass Techniken wie virtueller Speicher zum Einsatz kommen, um Requests sicher voneinander zu trennen. Dies stellt eine nicht unbeachtliche Gefahr dar, und wir werden weiter unten ein Design von SAP kennen lernen, bei dem versucht wird, die Requests besser zu isolieren.

Sun spricht bezüglich der Java-5-Neuerungen davon, dass in Zukunft auch die Ressourcenverwaltung unter Java besser isoliert und kontrolliert werden soll. Es werden somit weitere Techniken aus der Betriebssystemwelt eingebaut.

Aber nicht nur zwischen Prozessen und Nutzern gilt das Prinzip des Least Common Mechanism: Auch innerhalb großer Applikationen sollten einzelne Komponenten voneinander isoliert werden können. Dies geht hinunter bis in das feingranulare Objektdesign, z. B. die Unsitte, Referenzen statt Kopien von privaten Variablen zu veröffentlichen.

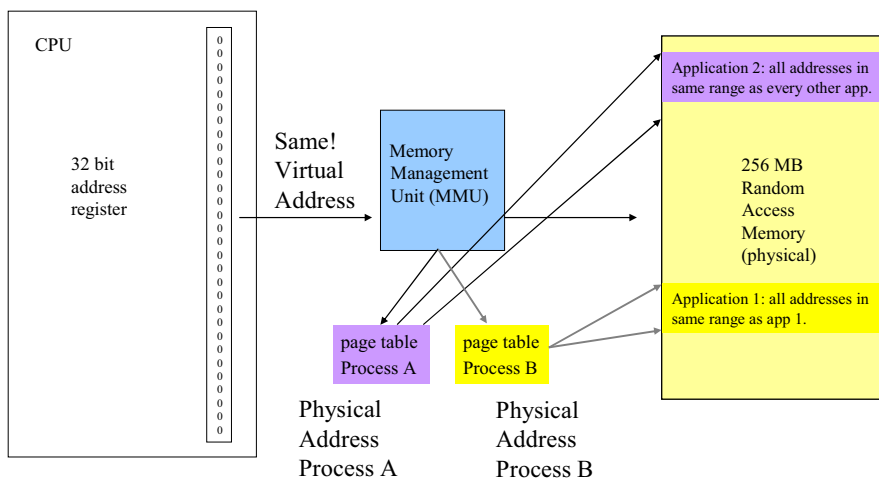


Abb. 3.1 Adressisolation bei einem Betriebssystem

Wenn ein Browser Fenster, Teilfenster, Sites und Programmcode alle gemeinsam in einer globalen Baumstruktur abbildet und dann versucht, innerhalb dieser globalen Datenstruktur Sicherheitszonen zu bilden (z. B. das „Same Origin“-Prinzip: Nur Code von der Site X darf Daten von der Site X bearbeiten) – was kann man dann erwarten? Ganz einfach: Endlose Sicherheitslücken, die daraus resultieren, dass mit immer neuen If-Clauses versucht wird, die Same-Origin-Policy auf einer globalen Datenstruktur des DOMs zu realisieren. Das DOM ist global navigierbar, ebenso wie File-Systeme und kann daher als globale Datenstruktur verstanden werden. Dazu mehr weiter unten.

Gemeinsame Daten sind bereits im Single-Threaded-Fall ein Problem, sowohl im Kernel als auch in Applikationen. Die Gefahren reichen von korrupten Daten durch falsche Behandlung (z. B. Verletzung der relationalen Integritätsregeln) bis hin zu korrupten Daten durch Fehler bei unterbrochenen Zugriffen oder Zuverlässigkeitsproblemen durch Deadlocks.

Noch kritischer wird die Nutzung gemeinsamer Daten im Falle von Multi-Threading. Hier gelingt es nur unter Einsatz von Locking-Techniken, Datenkorruption zu vermeiden – oft zum Preis von geringer Performance. Seltsamerweise ist gerade diese schwierige Form von Multi-Threading sehr weit verbreitet und wird z. B. in Java durch Monitore unterstützt.

Konkurrierende Verfahren, die auf dem Prinzip der Communicating Sequential Processes (CSP) beruhen, sind dagegen fast in Vergessenheit geraten. Dabei sind gerade sie es, die formal relativ einfach zu verifizieren sind.

Aus der gleichzeitigen Behandlung von gemeinsamen Daten resultieren aber auch massive Sicherheitsprobleme, wie z. B. Time-of-check To Time-of-Use (ToCtoU) Attacken: Hier wird erst ein Datum geprüft, das dann von einem anderen Thread nachträglich geändert wird. Die nachfolgende Nutzung des Datums prüft nicht erneut (ein Fehler, der häufiger beim Mozilla/Firefox-Browser auftrat). Anfällig für diese Attacken sind auch Unix-Administrations-Skripts, die Files manipulieren (s. [Vogt]).

CSP kann sehr gut als Pipeline-Pattern ausgedrückt werden, bei dem zwar parallele Ausführung von Code stattfindet, aber immer auf getrennten Daten. Beispiele sind Unix Pipelines oder die Shading-Architektur moderner Grafikkarten. Wie sehr die Plattformarchitektur hier die Sicherheit prägt, lässt sich schon an der Linux-Netfilter-Architektur demonstrieren. Die Vorgängerarchitektur IPChains hatte eine sehr unangenehme Eigenschaft: Die Zeitpunkte der Modifikation von Datenpaketen konnten mit den Zeitpunkten der Filterung von Daten bunt gemischt werden. Dadurch kam der Reihenfolge von Filterung bzw. Modifikation eine entscheidende Bedeutung für das Ergebnis des Filterprozesses zu.

Der Nachfolger IPTables, basierend auf dem neuen Netfilter Framework, hat dieses Sicherheitsproblem dadurch vermieden, dass er klar definierte Zonen der Modifikation mit klar definierten Zonen der Filterung erzeugt: Innerhalb von Filterzonen kann jetzt nicht mehr modifiziert werden.

3.8 Psychological Acceptability

Es ist von wesentlicher Bedeutung, dass das Nutzer-Interface auf einfache Bedienbarkeit hin entworfen ist, sodass die Nutzer die Schutzmechanismen routinemäßig und automatisch korrekt bedienen. In dem Maße, wie das Bild, das ein Nutzer sich von seinen Sicherheitszielen macht, mit den Mechanismen übereinstimmt, werden Fehler bei der Bedienung minimiert.

Saltzer und Schroeder hatten bereits 1975 verstanden, dass eine gute Usability ein integraler Bestandteil von Plattformensicherheit ist. Aber erst in den vergangenen Jahren hat man begonnen, diese Erkenntnis wieder zu entdecken. Die Gründe dafür haben wir bereits erwähnt: Das bislang vorherrschende Internetbedrohungsmodell wird in zunehmendem Maße als nicht umfassend genug erkannt: Viren und Trojanische Pferde attackieren die Endpunkte der Kommunikation und Phishing-Attacken greifen über semantische Mittel die Benutzer der Plattformen an. Die gesamte Sicherheit eines Geschäftsvorgangs hängt somit von mehreren Bedrohungsmodellen ab und die Komponente „Nutzer“ ist dabei von entscheidender Bedeutung.

Umso ernüchternder sind die bisherigen Ergebnisse der Forschungen zum Verhältnis von Security und Usability. So konnte Garfinkel in [Garf] nachweisen, dass die heutigen Systeme wichtige Sicherheitsprinzipien nicht zum Nutzer transportieren können, d. h. die Benutzer verstehen die Terminologie oder die Mechanismen nicht – Beispiele sind die Zertifikatswarnmeldungen gängiger Browser oder die kaum genutzte Möglichkeit der Verschlüsselung von Mail. Zudem wird der Nutzer irregeführt, da die Systeme häufig nicht das tun, was im GUI suggeriert wird.

Das Verhältnis von Sicherheit und Usability ist von so großer Bedeutung, dass es weiter unten in einem eigenen Kapitel behandelt wird. Dort wird auch auf den sehr wichtigen Begriff des *User Conceptual Models* eingegangen: Wie können wir das Verständnis von Nutzern erfassen und nicht zuletzt auch testen? Die Usability-Forschung hat hier bereits Grundlagen für die empirische Überprüfung geschaffen.

Das Prinzip der psychologischen Akzeptanz muss jedoch noch deutlich erweitert werden: Nicht nur die Endbenutzer sind als zentraler Teil der Sicherheit eines Systems zu verstehen, auch das konzeptuelle Modell der Entwickler bezüglich einer Applikation ist wichtig. So hat man langsam verstanden, dass Entwickler, die typischerweise Intranet-Applikationen entwickeln, ein User Conceptual Model mitbringen, das für Internet-Applikationen sehr gefährlich ist: So gehen Intranet-entwickler häufig viel zu naiv mit dem Input von Nutzern um oder verkennen die Gefahren von Denial-of-Service-Attacken.

Hier nur noch ein kurzes Beispiel für die Probleme, die auf das Systemdesign im Bereich Usability zukommen werden. In der oben erwähnten Rede zur RSA-Konferenz 2006 schlägt Bill Gates vor, den Browserbenutzern durch bestimmte Farben einen Hinweis auf die „Güte“ einer besuchten Site zu geben – quasi ein „trusted rating“ für Websites (wohl gegen entsprechende Gebühr). Egal wie das „Recht“, eine Farbe zu tragen, zustande kommt (vielleicht über eine vertrauenswürdige Certificate Authority): Es ist der Browser, der die Farbe letztlich anzeigen

muss. Der Benutzer wird bald anfangen, den Farben zu vertrauen und in seiner Wachsamkeit nachlassen. Umso fataler ist es nun, wenn es Malware gelingt, die Farben im Browser zu manipulieren. Bei sämtlichen Browsern war es bislang regelmäßig der Fall, dass z.B. über JavaScript wichtige Elemente des User-Interface wie die Anzeige von sicheren Verbindungen, Namen von Websites etc. gefälscht werden konnten. Ist das GUI selbst manipuliert, hat ein Nutzer keine Chance, die Gefahr zu erkennen.

Was fehlt, ist ein „Trusted Path“ – d.h. ein wirklich sicherer Weg, den Nutzer über gewisse Dinge zu informieren bzw. Input vom Nutzer zu sammeln. Der Dreitastengriff <ctrl-alt-del>, der zum Windows-Log-in führt, ist übrigens ein hart im Kern des Betriebssystems verdrahteter Trusted Path zum Log-in-Programm des Betriebssystems. Sobald der Benutzer diese drei Tasten gleichzeitig drückt, landet er sicher nicht bei einem Trojanischen Pferd. Ansonsten kann natürlich jedes Programm einen Dialog zur Abfrage der Benutzerkennung starten. Beliebt bei Malware-Autoren sind auch Bilder von Systemdialogen, die einen anderen Dialog überlagern, von dem nur der <OK>-Button sichtbar ist.

Die Einführung von Farbkennungen, die Sicherheit auf einer Plattform vorgaukeln, die bereits von der Architektur her nicht in der Lage ist, granulare, beschränkte Rechtezonen in Programmen zu schaffen bzw. Module voneinander zu isolieren, ist schlicht eine Einladung zum Desaster. Die zunehmende Popularität von Web2.0-Techniken wird diese Problematik noch verschärfen.

3.9 Zusammenfassung der Prinzipien von Saltzer und Schroeder

Fassen wir die wesentlichen Regeln des Designs sicherer Systeme noch einmal zusammen:

- Software muss aus kleinen, isolierten Einheiten aufgebaut werden, deren externe Beziehungen am Interface deutlich werden. Damit wird sowohl praktische Schadensreduzierung durch Isolation als auch eine schnelle und einfache Sicherheitsanalyse möglich.
- Zugriffsentscheidungen dürfen nur auf der Basis expliziter, minimaler und keinesfalls durch immer und global verfügbare Permissions fallen.
- Systeme müssen in möglichst klein-granulare Sicherheitszonen aufgeteilt werden. An den Grenzen der Sicherheitszonen sind erneute Zugriffskontrollen nötig. Rechte für privilegierte Funktionen sollen nicht außerhalb der privilegierten Komponente sichtbar werden. Umgekehrt soll eine privilegierte Komponente nicht Rechte des Aufrufers erhalten müssen, um mit Parametern des Aufrufers arbeiten zu können. Nach Möglichkeit sollte neben ablaufbasierter Kontrolle auch eine semantische Kontrolle des Requests durchgeführt werden. Der Zugriff auf sicherheitskritische Ressourcen muss über die Zeitachse und über Zonen hinweg kontrolliert werden. Änderungen an Rechten müssen unmittelbar an die Stellen der Zugriffskontrolle propagiert werden.

- Das Softwaredesign von Applikationen sollte öffentlich sein. Zumindest sollte ein interner Review-Prozess stattfinden, in dessen Verlauf eine Sicherheitsanalyse durch nicht an der Entwicklung Beteiligte erstellt wird.
- Softwarekomponenten sollten eine klare, eingeschränkte Sicherheitsfunktion besitzen und Ballungen von Autorität vermeiden. Sie sollten die benötigten Rechte explizit und sichtbar machen. Das Softwaredesign sollte ausschließlich die sichere und kontrollierte Übertragung von Rechten ermöglichen.
- Sichere Systeme dehnen das Prinzip der „least privileges“ auch auf Softwareagenten als Stellvertreter von Benutzern aus. Wenn die Rechte auch Zugriffsmöglichkeiten umfassen, die nicht explizit aus einer Businesssicht modelliert sind, dann spricht man von „least authority“. Sichere Systeme erlauben keine feste, statisch zugeordnete Autorität, sondern verteilen sie nur nach Bedarf und feingranular. Im Falle von Code-Fehlern oder böartigem Code (man beachte, dass dies dann keinen wirklichen Unterschied mehr macht) wird dadurch der Schaden minimiert. Eine detaillierte Behandlung von POLA erfolgt gleich im Anschluss.
- Gemeinsam genutzte Ressourcen wie globale Daten, Verzeichnisse oder Kernel-Strukturen sollten vom Softwaredesign vermieden werden. Zu vermeiden ist außerdem die gegenseitige Beeinflussung der Komponenten durch nicht-funktionale Aspekte wie CPU-Benutzung, Wärmeerzeugung, Platzbedarf und Netzwerkbelastung. Das Softwaredesign muss nicht nur illegalen Informationsaustausch unterbinden, sondern auch die Verfügbarkeit und Zuverlässigkeit der Lösungen sichern.
- Das sichere Systemdesign berücksichtigt die Benutzbarkeit der verwendeten Technologien bei allen Beteiligten. Dabei schließt es Entwickler und Administratoren genauso wie Mitarbeiter vom Business ein. Mechanismen, die in der Vergangenheit eine hohe Fehlerquote gezeigt haben, werden durch die Wahl besserer Sprachen und Architekturen ersetzt.

3.10 Principle of Least Authority – POLA

3.10.1 *POLA* vs. *POLP*

Der Begriff POLA stammt aus dem Umfeld der Sprache „E“ [Erights] und der Capability-Bewegung [Cap] und stellt eine Erweiterung von POLP, dem Principle Of Least Privilege, dar. Dass Privilegien minimal verteilt sein sollten, ist kein Geheimnis. Die Einschränkung der Benutzerrechte auf ihre fachlich nötigen Funktionen wird bereits vielfach praktiziert. Kern dieser Einschränkungen sind die als Privilegien oder auch als Permissions modellierten Rechte von Benutzern oder Softwareagenten. Typisch für POLP-Systeme ist die Trennung der Rechtedefinitionen von der Software, meist durch externe Konfigurationsdaten realisiert. Privilegien oder Permissions bezeichnen daher das, was innerhalb eines Systems erlaubt ist. Wieso also reicht das POLP-Prinzip nicht aus?

Das Schadenspotenzial einer Software hängt direkt von seiner *Autorität* ab, hier verstanden als Möglichkeit, Effekte zu erzielen. Besonders deutlich wird dies bei einem Betriebssystemkern, der eine sogenannte *Trusted Base* bildet – eine Menge von Code, dem sehr hohe Privilegien zugesprochen werden. Aber auch Serverprogramme oder simple Spiele wie Solitaire operieren meist mit zu viel Autorität. Wird die Möglichkeit, Effekte zu erzielen, nicht durch die Zugriffskontrolle eingeschränkt und kontrolliert? Benutzerberechtigungssysteme modellieren die Rechte der Nutzer in Abhängigkeit von ihrer Funktion. Applikationen prüfen zur Laufzeit, ob ein Nutzer das Recht hat, um einen Effekt zu erzielen. Somit dürfte kein Raum für Missbrauch gegeben sein.

Dies stimmt allerdings nur unter zwei Voraussetzungen: Die Rechte müssen richtig konfiguriert sein. Dies ist eine Frage der Organisation und des Aufwands, der betrieben wird. Die Granularität der Rechteaufteilung und die Menge von Rechten, Gruppen, Rollen etc. können hier auf Dauer zu Problemen führen. Die zweite Voraussetzung ist noch viel kritischer. Das Programm muss fehlerfrei sein: Es muss an den richtigen Stellen die richtigen Prüfungen durchführen. Auf keinen Fall darf es ein Problem bei der Input-Validierung besitzen (Buffer-Overflow, Fehlinterpretation von User-Werten etc.). Und wir müssen voraussetzen, dass der Programmierer weder selbst böartigen Code eingefügt noch unabsichtlich einen Fehler gemacht hat. Das ist eine sehr hohe Hürde und kann normalerweise nicht vorausgesetzt werden. Nur wenn beide Voraussetzungen gegeben sind, ist ein POLP-System wirklich „sicher“!

Der Begriff der „Autorität“ in POLA hingegen geht über die bloße Erlaubnis hinaus und bezeichnet die tatsächliche Möglichkeit, Effekte zu erzielen. Der Unterschied scheint zunächst gering zu sein – im Idealfall verschmelzen beide Prinzipien sogar. Allerdings tritt dieser Idealfall praktisch nicht ein: In den meisten Systemen hat Software mehr Autorität, als Rechte vergeben sind, weil nicht alle Möglichkeiten, Effekte zu erzielen, tatsächlich als Rechte modelliert sind. Schlimmer ist jedoch, dass meist die kompletten Rechte eines Users an ein Programm übergeben werden. Ist dieses fehlerhaft oder böartig, können die Rechte missbraucht werden. Die Argumentation für POLA setzt daher bereits früher an: Wieso hat ein Programm überhaupt die Möglichkeit, ein Recht zu verwenden, das es im Zusammenhang mit der momentanen Aufgabe gar nicht benötigt? Wieso hat ein Programm die Möglichkeit, etwas aufzurufen, für das es möglicherweise keine Rechte besitzt? Wieso werden bei der Zugriffskontrolle Rechte zur Prüfung herangezogen, die für die momentane Funktion des Programms keine Rolle spielen?

Im Grunde genommen stellt das POLA-Prinzip an jeder Stelle im Programmcode eine entscheidende Frage: Kann der Code potenziell an dieser Stelle auch Aktionen ausführen, die mit der eigentlichen Aufgabe in keinem Zusammenhang stehen – Aufgabe hier im Sinne des Benutzers der Applikation verstanden. Ist dies der Fall, dann liegt ein Verstoß gegen POLA vor und das Gefahrenpotenzial der Applikation ist größer als nötig. Hier wird deutlich, dass POLA aggressiv auf Schadensbegrenzung abzielt (der Unterschied zwischen fehlerhaftem und böartigem Code wird in der Folge auch unwichtig). Zählen wir kurz die Eigenschaften

eines auf POLA basierenden Systems auf und diskutieren sie anschließend an einem Beispiel.

Kennzeichnend für POLA-Systeme ist die Ansicht, dass Code alleine keinerlei Gefahr darstellt, denn er besitzt nach dem Laden keine Autorität. Autorität und damit die Möglichkeit, Effekte zu erzielen, bekommt der Code erst durch die Übertragung von sogenannten *Capabilities*. Diese Rechteübertragung ist demnach feingranular und dynamisch. Capabilities vereinen in sich das Recht zum Zugriff auf eine geschützte Ressource sowie die aktuelle Autorität, dies zu tun. Kryptografische Schlüssel können als solche Capabilities gesehen werden, aber auch Objektreferenzen in Programmiersprachen, die „memory safe“ sind. Grundlage der Schadensbegrenzung in POLA-Systemen ist letztlich die Forderung, dass Code-Module niemals mehr Autorität besitzen sollten, als sie für ihre Aufgabe tatsächlich benötigen. Existiert solche überflüssige Autorität, so sprechen wir von *Ambient Authority*. Aber was bedeutet „tatsächlich benötigen“ in diesem Zusammenhang? Sehen wir uns dazu das Beispiel eines Programms zum Kopieren von Dateien an.

Ein typisches Copy-Programm eines Betriebssystems nimmt zwei Parameter, den Namen der Quelldatei und den Namen der Zieldatei. Der Aufruf lautet also: `copy QuellDatei ZielDatei`. Das Copy-Programm liest die die Dateinamen bezeichnenden Zeichenketten ein, navigiert durch das Dateisystem an die entsprechenden Stellen und öffnet die Dateien. Dazu muss es die Rechte des Aufrufers verwenden. Passen diese Rechte zu den bei den Ressourcen vermerkten ACLs, dann werden die Zugriffe gestattet und der Kopiervorgang durchgeführt. Aufgrund der existierenden Ambient Authority könnte das Copy-Programm jederzeit und ohne Interaktion mit dem Nutzer beispielsweise einen Socket öffnen und die Quelle über das Internet an jemanden schicken. Der Nutzer besitzt im Allgemeinen dieses Recht und somit auch das Programm. Ebenso könnte das Copy-Programm alle Dateien des Nutzers manipulieren, nicht nur die beiden, deren Namen als Zeichenkette übergeben wurden.

Der Kopiervorgang ließe sich aber mit deutlich weniger Rechten durchführen: Wozu benötigt das Copy-Programm Navigationsrechte auf das Dateisystem? Wozu benötigt es die Rechte, um die Files lesen und schreiben zu können? Hier kommt die Verbindung zum Softwaredesign ins Spiel: Wenn das Copy-Programm statt Zeichenketten direkt geöffnete Filedeskriptoren (ähnlich wie Capabilities) als Argumente bekäme, bräuchte es kein Recht, auf Files des Users zuzugreifen. Wenn also das Copy-Programm durch eine Buffer-Overflow-Attacke übernommen würde, dann könnte es zwar Schaden anrichten – aber nur in Bezug auf das zum Schreiben geöffnete File und sonst auf nichts.

Die Fähigkeiten von POLA zur Schadensbegrenzung dürften damit deutlich geworden sein. Es besteht aber auch ein unmittelbarer Zusammenhang mit der Sicherheitsanalyse von Systemen: Wie groß ist die Bedrohung durch das obige Copy-Programm? Ohne POLA ist das Gefahrenpotenzial gleich dem Rechteset des Benutzers – und damit maximal. Mit POLA ermittelt eine Sicherheitsanalyse anhand der granular übergebenen Autorität das Gefahrenpotenzial und kann zu einer genauen Aussage kommen.

Auf Basis von POLA lassen sich Systeme realisieren, bei denen Sicherheit Teil der Softwarearchitektur ist und die weit über die Möglichkeiten von z. B. Access-Control-Listen hinausgehen. Selbst innerhalb von Applikationen können Zonen geringster Autorität hergestellt werden. Im Unterschied zu Sandboxes wird diese Autorität dynamisch weitergegeben, anstatt durch statische Konfiguration. Und nur durch die Weitergabe der Autorität z. B. in Form von Capabilities kann Autorität propagiert werden. Die Regeln für die Propagation werden wir weiter unten noch detailliert untersuchen.

3.10.2 Access Control Lists und Sandboxes

Wir werden jetzt zwei Mechanismen untersuchen, die für den Bau sicherer Systeme nur begrenzt tauglich sind, da sie POLA nicht wirklich unterstützen: Access Control Lists und Sandboxes. Beide Mechanismen werden heute jedoch in den meisten kommerziellen Systemen eingesetzt und eine Diskussion ihrer Defizite ist daher besonders wichtig. Besonderes Augenmerk werden wir dabei auf den Aspekt der ambienten Autorität bzw. der Propagation von Privilegien richten. Im letzten Abschnitt werden wir dann eine Form der Zugriffskontrolle kennen lernen, die Rechte nicht aus der Identität der Beteiligten (User oder Softwareagent) ableitet, sondern feingranular innerhalb der Software propagiert.

Die Zugriffskontrolle durch Access Control Lists (ACL) dient uns als erstes Beispiel. Bei einer Ressource sind Nutzeridentitäten und Rechte in einer ACL vermerkt. Typische Beispiele sind die File-Access-Rechte unter Unix oder Windows. Beim Zugriff von Programmen auf diese Ressource wird die Identität des Nutzers genutzt, um in der ACL nachzusehen, ob das entsprechende Recht für diesen Zugriff dort vermerkt ist: Darf der Nutzer lesen/schreiben etc.?

Wichtig für das Verständnis dieses Mechanismus sind die folgenden Punkte:

- Programme werden mit der Identität des Nutzers versehen, der sie startet.
- Programme erhalten *alle* Rechte dieses Nutzers im betreffenden System, ungeachtet dessen, was das Programm tatsächlich tun muss – es entsteht also Ambient Authority.
- Bei der Zugriffskontrolle (z. B. wenn das Programm einen Systemcall benutzt) wird das von der Ressource verlangte Recht gegen die gesamte Rechtemenge des Nutzers geprüft.
- Der bei der Prüfung verwendete Rechtevektor des Programms/Benutzers ist statisch vorhanden und nicht abhängig von der momentanen Programmfunktion.

Im Klartext bedeutet dies komplettes Vertrauen in ein Programm, das von einem vertrauenswürdigen Nutzer gestartet wurde. Ist das Programm aber durch einen Virus oder andere Malware kompromittiert worden, dann wird die Malware mit den gesamten Rechten des Nutzers ausgeführt – wie das korrekte Programm. Aber nicht nur bösartige Programme profitieren von diesem Mechanismus: Auch

ein gutartiges, aber fehlerhaftes Programm kann dadurch weitaus größeren Schaden anrichten als ohne Ambient Authority.

Letztlich liegt die Ursache für diese Schwäche darin, dass alle Rechte des Nutzers zur Kontrolle herangezogen werden, ohne Rücksicht auf den eigentlichen Zweck des Programms (muss es wirklich auf einen Socket zugreifen?) und ohne Rücksicht auf die Absicht des Nutzers (will er wirklich, dass eine bestimmte Ressource angefasst wird?). Abbildung 3.2 zeigt, dass die Auswertung der Rechte statisch und nicht auf die momentane Aktivität bezogen ist. Dadurch wird immer mit viel zu vielen Rechten gearbeitet.

Wenn wir diese beiden Fehler unterbinden wollen, dann stehen uns verschiedene Mittel zur Verfügung: Um die Aktionen eines Programms auf seinen wirklichen Zweck eingrenzen zu können, d. h. die Ressourcenzugriffe in Relation zum Programm zu begrenzen, verwenden wir eine sogenannte *Sandbox* – also einen Referenzmonitor. Er hält eine Relation zwischen dem jeweiligen Programm und den möglichen Ressourcen und überprüft bei jedem Zugriff, ob die Ressource in der Sandbox für dieses Programm konfiguriert ist, d. h., ob die Sandbox Policy den Zugriff auf eine Ressource für dieses Programm erlaubt (s. Abb. 3.3).

Der Effekt ist der, dass ein Nutzer zwar ein Programm starten und auf eine Ressource zugreifen lassen kann, deren Eigentümer er ist – er besitzt also alle Rechte an der Ressource. Da der konfigurierte Zweck des Programms diese Ressource aber nicht mit einschließt, wird die Sandbox den Zugriff ablehnen. Selbst wenn das

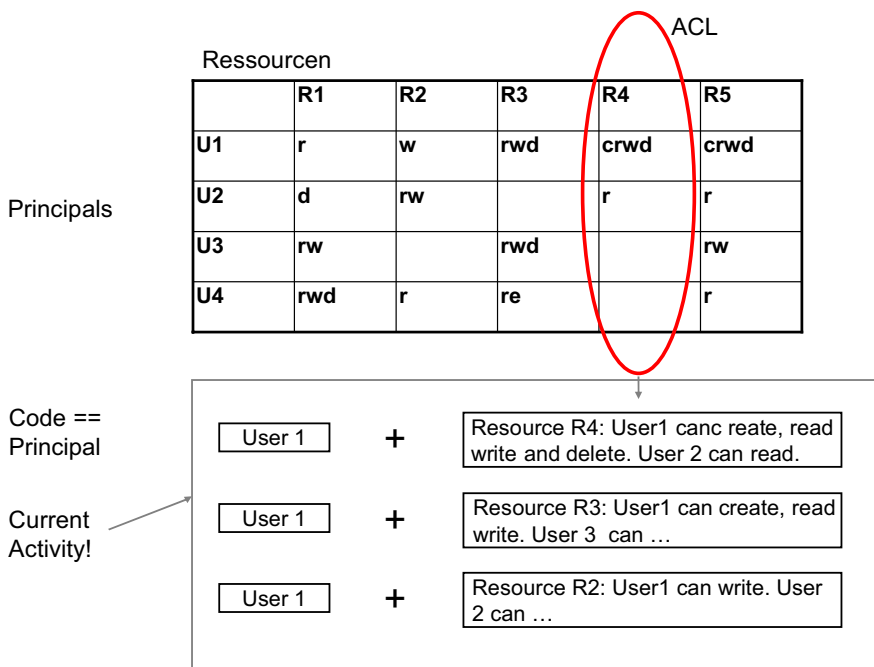


Abb. 3.2 Access Control List für fünf Ressourcen und vier Nutzer (Principals)

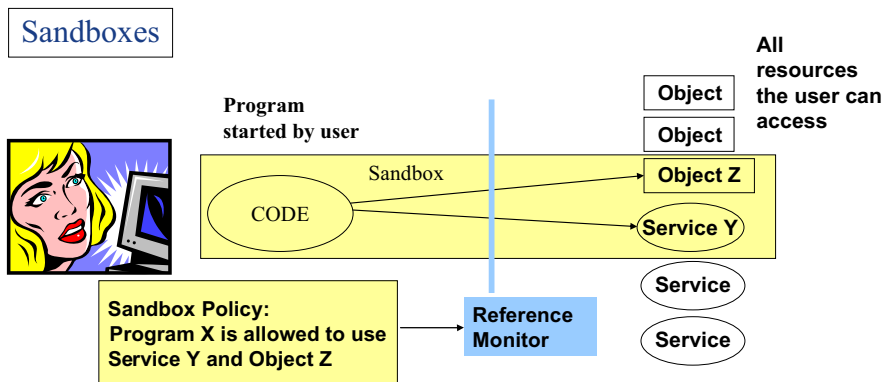


Abb. 3.3 Eine Sandbox wird realisiert durch einen Referenzmonitor

Programm durch Malware ersetzt wurde, ist die Ressource sicher. Die Sandbox schützt also Ressourcen vor Irrtümern des Nutzers sowie vor böartigem Code.

Eine weitere Möglichkeit, die vorhandenen Rechte einzuschränken, liegt in der Definition verschiedener Rollen des Benutzers, also Subsets von Rechten. Der Benutzer darf somit nur in jeweils einer Rolle tätig sein. Dies stellt den Benutzer selber und nicht das Programm in eine Sandbox.

Beide Mechanismen, die durch Rollen eingeschränkter ACLs eines Users oder die eingeschränkte Sandbox eines Programms, bewirken eine Reduktion der Autorität, jedoch findet diese Reduktion immer noch sehr grobgranular statt. Immer noch sind im Moment der Ausführung einer Funktion grundsätzlich mehr Rechte vorhanden, als nötig wären. Über den grundsätzlichen Sinn von granularen Einschränkungen gehen die Meinungen weit auseinander. Hier ein Zitat von Butler Lampson anlässlich der Keynote zur Usenix-Konferenz 2005 [Lamp] (aus einer Mail der cap-talk-Liste von Jed Donelly [Don]).

“I think, for example, that the Principle Of Least Privilege has done an enormous amount of damage to security because what it encourages you to do is to make everything fine grain and work out all the dependencies very carefully and it’s too complicated. You can’t keep track of it. You’re bound to mess it up. Even if you get it right today it will be wrong three months from now. Nobody will have the patience to ever look at it again because there’s just too much of it. So I say absolutely no to fine grain protection. Everything should be as coarse grain as possible because otherwise you won’t be able to administer it. That’s a very unpopular position with most people. I think there’s a lot of empirical evidence that tells us now that it’s right.”

Lampson steht mit dieser Ansicht nicht alleine. Ein ähnlicher Satz findet sich auch beim Autor von Qmail, Daniel Bernstein in [Bern].

Feingranulare Rechtedefinitionen auf Basis von ACLs und Sandbox-Konfigurationen – beide sind übrigens extern zum Source Code – haben tatsächlich diese negativen Eigenschaften bezüglich der Wartbarkeit und Administration. Letztlich liegt das Problem darin, dass die Rechte statisch an eine Identität (Benutzer oder

Softwareagent oder beides) gebunden sind und daher nie exakt zu den tatsächlich und aktuell benötigten Rechten passen. Verwaltbar sind sie nur auf Basis von Aggregaten wie Rollen und Gruppen, die ihrerseits nicht zu zahlreich werden dürfen, damit sie verständlich und übersichtlich bleiben. Es gibt noch eine weitere Möglichkeit, statische Rechtevergabe im Zusammenhang mit bestimmten Programmen oder Ressourcen einzuschränken: Der Benutzer muss diese Anpassung selber vornehmen, indem das System vor Anwendung eines Rechts einen Warn-dialog anzeigt und um eine Bestätigung bittet. Vielen Benutzern kommen hier gleich die Erfahrungen mit Personal Firewalls in den Sinn: Es werden unverständliche Fragen mit unklaren Konsequenzen gestellt. Am Ende leidet entweder die Usability oder die Sicherheit der Systeme beträchtlich. Der Zusammenhang zwischen Usability und der Art und Weise der Zugriffskontrolle sowie der Propagation von Rechten ist überaus eng und wir werden ihn im Laufe des Buches noch öfters diskutieren. Hier wollen wir vorab behaupten, dass Systeme, die ambiente Autorität verwenden und diese nur durch den User einschränken lassen wollen, notwendigerweise entweder die Benutzbarkeit oder die Sicherheit als Prinzip aufgeben müssen.

Lassen Sie uns hier jedoch einmal annehmen, dass die feine Granularität der Rechteverteilung nicht durch externe Konfiguration, sondern durch eine besondere Form der Softwarearchitektur erreicht wird. Die Einschränkung der Rechte wird somit Teil des Softwaredesigns (eine erste vorstellbare Konsequenz dieses Ansatzes könnte sein, dass es weniger Warndialoge benötigt, denn die Möglichkeiten sind von vorne herein eingeschränkt. Unklar ist aber, wie sich dies auf die Flexibilität des Systems auswirkt).

3.10.3 *Capabilities*

Wenn wir also den Zugriff auf eine Ressource von der tatsächlichen Absicht des Aufrufers abhängig machen wollen, dann greifen wir zu sogenannten *Capabilities*, oder noch besser zur objektorientierten Variante davon, den *Object Capabilities*. Wir werden später Capabilities noch anhand von praktischen Beispielen genauer untersuchen, deshalb hier vorab nur die Kerneigenschaft von Capabilities: Capabilities sind *granulare* und *delegierbare* Rechte, die einen Request begleiten. Ein Programm kann nur dann auf eine Ressource zugreifen, wenn es vom Nutzer direkt oder indirekt von aufrufenden Funktionen eine Capability dafür erhalten hat. Der Besitz einer Capability ist das Recht – und im Fall von Objekt Capabilities auch gleichzeitig das Mittel (nämlich eine Referenz auf das gewünschte Objekt), den Zugriff durchzuführen. Wie Abb. 3.4 zeigt, stellen Capabilities praktisch ein dynamisches Gegenstück zu ACLs dar: Für jede Aktivität des Nutzers wird dynamisch eine Teilmenge der Rechtemenge des Nutzers erzeugt und dem Code als Capability übergeben.

Es wird also bei einer Zugriffsentscheidung eben nicht der ganze Rechtevektor eines Users herangezogen (und potenziell missbraucht), sondern es stehen nur

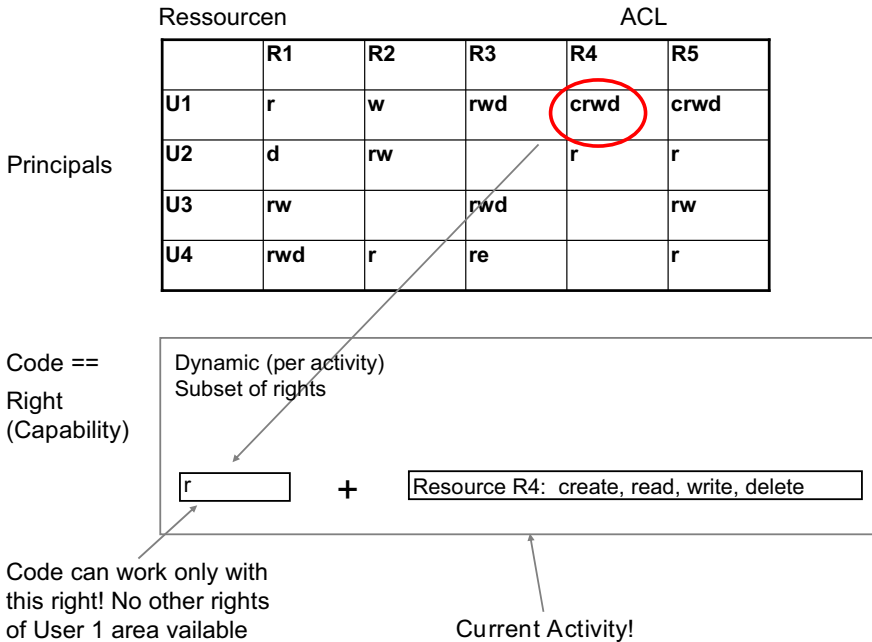


Abb. 3.4 Verhältnis von ACL und Capabilities

einzelne, für die jeweilige Aktion benötigte Rechte zur Verfügung. Auch diese könnten natürlich von Malware oder fehlerhafter Software missbraucht werden – POLA ist ein defensives Prinzip und kann Fehler nicht definitiv ausschließen – aber es werden eben nur diese Rechte missbraucht und nicht alle Rechte. Plastisch an einem Beispiel aus der wirklichen Welt ausgedrückt: Angenommen, Sie verwalten eine Reihe von Zimmern und besitzen zu jedem Zimmer einen Schlüssel. Nun kommt jemand, der Sie um Zutritt zu einem dieser Zimmer bittet (und dazu auch berechtigt ist): Geben Sie ihm dann Ihren kompletten Schlüsselbund (ACL) oder nur den einen Schlüssel, der für dieses Zimmer nötig ist (POLA)?

Die gängigen Betriebssysteme wie Unix und Windows funktionieren übrigens nach dem Antipattern der Ambient Authority und sind dementsprechend anfällig für Malware. Aber auch innerhalb von Programmen stellt sich die Frage, ob jedes Modul bzw. jedes Objekt mit den gleichen Rechten laufen muss wie das gesamte Programm – völlig unabhängig von seiner Funktion.

Machen wir uns nun auch das Verhältnis von Capabilities zu einer Sandbox klar. Die folgende Abb. 3.5 zeigt eine Sandbox Policy in Form einer ACL, bei der Applikationen als Principals in die Zeilen eingetragen sind. Der Referenzmonitor verfügt über eine statische Rechtezuteilung für den Zugriff eines Programms auf Ressourcen. Die Rechtemenge eines Programms kann entweder mit der Rechtemenge des Nutzers kombiniert werden (Schnittmenge oder Vereinigungsmenge) oder aber ohne Berücksichtigung der Nutzerrechte verwendet werden. In jedem Fall aber erfolgt die Zugriffskontrolle letztlich anhand von Identitäten, denen sta-

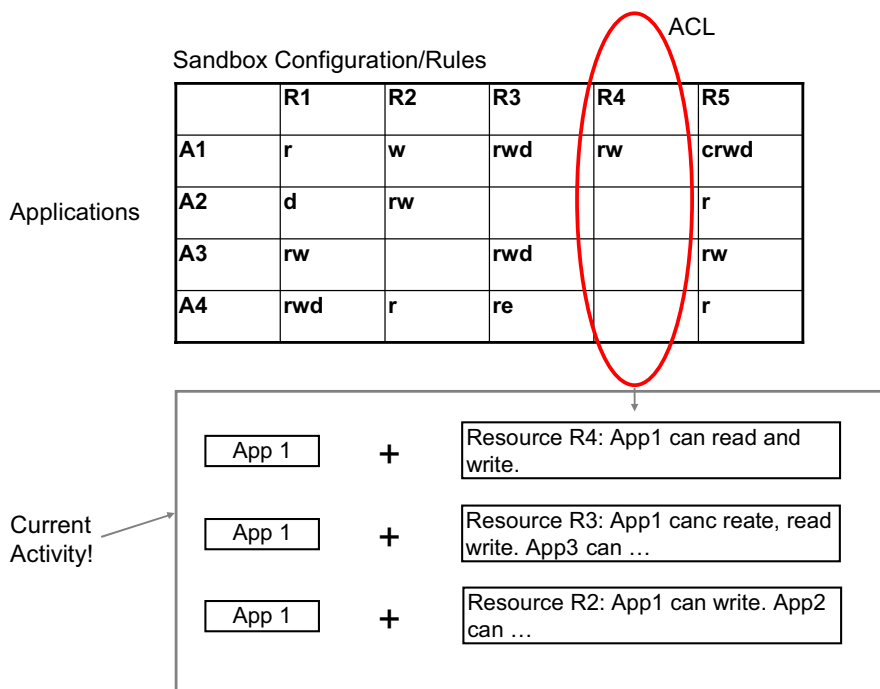


Abb. 3.5 Sandbox Policy in Form einer ACL

tisch Rechte zugeordnet sind. Auch wenn durch die Sandbox grundsätzlich weniger Rechte eines Users missbraucht werden können, ist immer noch eine beträchtliche Menge ambierter Autorität vorhanden. Die in der Literatur häufig zu findende Unterscheidung zwischen ACLs und Capabilities als spaltenweise bzw. zeilenweise Sicht auf die gesamte Access Control Matrix eines Systems ist daher falsch, da sie das *Fließen* (also die dynamische Anpassung der Rechte an die Aktivität) der Capabilities nicht berücksichtigt. Erst dadurch entsteht der die Autorität einschränkende Effekt.

POLA gilt es auf allen Ebenen des Systemdesigns zu berücksichtigen: Dazu gehört z. B. auch der Zugriff auf Datenbanken. Muss ein Functional User innerhalb eines Servers tatsächlich Eigentümer aller Tabellen sein? Im Kapitel zur Absicherung von Servern werden wir geeignete Strategien der Einschränkung von Rechten kennen lernen.

3.10.4 Zusammenfassung

Zugriffskontrollen, die Rechte auf Basis von Identität verteilen, sind notwendigerweise grob und leisten sich viel ambiente Autorität, die von Malware oder fehlerhaften Programmen ausgenutzt werden kann. Versuche, die Rechte noch

feiner zu unterteilen, führen schnell zu Administrations- bzw. Wartungsproblemen. Das Einschalten des Users als dynamische Filter der Rechte in einer konkreten Situation führt hingegen zu Usability-Problemen.

Alternative ergänzende Systeme verlagern die Propagation von Rechten in die Software. In sicheren Systemen auf Basis von Capabilities darf es dem Code nicht gelingen, von selbst Autorität zu generieren. Ausschließlich durch die dynamische Weitergabe darf Autorität propagiert werden. Dies führt zur Vorstellung von Code als harmloser Sammlung von Funktionen, die erst dann sicherheitstechnisch bedeutsam werden, sobald Autorität in Form von Zugriffsmöglichkeiten (z. B. durch Referenzen) zugeführt wird. Das Fehlen von Autorität wiederum ist die Unmöglichkeit, kausale Effekte zu erzielen. Wir werden im Kapitel zu Softwaremechanismen sowie im Kapitel zu formalen Methoden der Beschreibung und Analyse sicherer Systeme erneut auf den Begriff der Autorität zurückkommen. Dort werden wir auch den Unterschied zwischen Information und Autorität beleuchten. Der Unterschied zwischen Rechtevergabe auf Basis von Identität und durch Propagation, wie er hier ausgedrückt wird, ist sicher fundamental in Bezug auf das Design sicherer Systeme. Allerdings kann man bereits hier berechtigte Zweifel anmelden, dass es tatsächlich ein „entweder-oder“ in dieser Frage gibt: Letztlich muss z. B. auch in Capability-basierten Systemen ein auf Benutzer bezogenes Konzept von Eigentum an Ressourcen existieren, das durchaus statisch sein wird und daher auch effektiv in Form einer Access Control Matrix verwaltet werden kann. Dies wirft die interessante Frage auf, wie beide Ansätze kombiniert werden können.

Literatur

- [Bern] D.J. Bernstein, Some thoughts on security after ten years of qmail 1.0, available online at <http://cr.yp.to/qmail/qmailsec-20071101.pdf>
- [Cap] Cap-talk mailing list, <http://www.eros-os.org/pipermail/cap-talk/>
- [Don] J. Donelly, cap-talk mailing-list Beitrag, <http://www.eros-os.org/pipermail/cap-talk/2008-April/010761.html>
- [Erighs] Erighs.org, Homepage der sicheren Programmiersprache „E“
- [Garf] S. Garfinkel, Design Principles and Patterns for Computer Systems That Are Simultaneously Secure and Usable, Dissertation, Massachusetts Institute of Technology (MIT) 2005, available online at <http://www.simson.net/thesis/>
- [Lamp] B. Lampson, Keynote zur Usenix Konferenz 2005, <http://www.usenix.org/events/sec05/tech/>
- [Mil] M.S.Miller, Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control, Dissertation, Johns Hopkins University, Baltimore, available online at <http://www.erights.org/talks/thesis/index.html>
- [RH] P. van Roy, S. Haridi, Concepts, Techniques, and Models of Computer Programming, The MIT Press, 2004
- [SaSch] J.H. Saltzer, M.D. Schroeder, The Protection of Information in Computer Systems, available online at <http://www.cs.virginia.edu/~evans/cs551/saltzer/>
- [Vogt] D. Vogt, Wurzelbehandlung. Linux Magazin 04/05 Seite 64 ff.
- [Wag] David Wagner, Object Capabilities for Security Google techtalk, November 30, 2007. <http://www.youtube.com/watch?v=EGX2I31OhBE>

Kapitel 4

Platform Security

Die vielfältigen Angriffsmöglichkeiten in dem gigantischen, durch das Internet geschaffenen verteilten System haben viele Hersteller von Software wie auch deren Nutzer überrascht. Für das sogenannte *Internet Threat Model* sind sie exemplarisch im RFC zur sicheren Spezifikation von Internet-RFCs von Eric Rescorla festgehalten worden [RFC 3552]. Jede Applikation und jedes Protokoll, das für das Internet geschrieben wird, sollte die dort gelisteten Fragen beantworten können (dieser RFC eignet sich auch sehr gut zur Dokumentation eigener Projekte, wenn unklar ist, auf welche Sicherheitsprobleme eingegangen werden sollte).

Mittlerweile dürften die Grundregeln der sicheren Kommunikation in verteilten Systemen einigermaßen bekannt sein, und es existieren Software-Frameworks, mit denen sich die Integrität und Vertraulichkeit von Daten beim Transport über unsichere Kanäle schützen lassen (siehe JSSE im Java-Bereich). Darüber hinaus sind praktisch alle am Internet betriebenen Web-Server mittlerweile in eine DMZ eingebunden.

Mit dem zunehmenden Reifegrad der Netzwerksicherheit ist allerdings auch die Erkenntnis über deren Grenzen deutlich geworden: Es tauchen immer mehr semantische Attacken auf (z. B. Phishing) – also Attacken, bei denen die Netzwerksicherheit keine Rolle spielt (und auch nicht kompromittiert wird). Die Konzepte der *Perimeter Security* – d. h. das Schaffen sicherer Räume mit einem genau überwachten Zugang – scheitern mittlerweile häufig an der zunehmenden Mobilität der Arbeitswelt. Und nach wie vor scheint die Zahl der durch Buffer-Overflows übernommenen Systeme sehr hoch zu sein.

All dies dürfte dazu beigetragen haben, dass in der letzten Zeit die Sicherheit der Plattform wieder mehr in den Vordergrund gerückt ist – der eigentliche Grund dürfte aber ein anderer sein: „Ubiquitous Computing“ und die Philosophie des „Always Connected“ haben die Zahl der vernetzten Rechner stark ansteigen (und ihre Größe stark schrumpfen) lassen. Je nach Einsatzgebiet bergen viele dieser Rechner bei Verlust oder Kompromittierung hohe finanzielle Risiken in sich, die aufgrund der Mobilität der Rechner nicht durch Maßnahmen der Network Security allein abgefangen werden können.

Unter dem Begriff „Plattform“ verstehen wir in den folgenden Abschnitten in erster Linie Betriebssysteme und deren Werkzeuge. Plattformen spielen eine zentrale Rolle für die Sicherheit eines Gesamtsystems, bestehend aus der Kombination von Betriebssystem, Programmiersprache, Applikationsarchitektur und beteiligten Personen. Wenn die Sicherheit der Plattform bereits mangelhaft ist, dann können darauf aufbauende Komponenten meist nicht mehr viel ausrichten.

4.1 Platform Security heute

Lassen Sie uns zunächst einige typische Beispiele für besonders sensitive, dabei aber kleine und mobile Plattformen geben: Moderne Smartphones können bei Missbrauch enorme Kosten verursachen und werden heute sogar für die Authentisierung von Bankgeschäften verwendet. Auch PDAs tragen häufig sicherheitskritische Informationen mit sich. Sind sie mit Malware infiziert, können sie diese über die Synchronisation mit dem Desktop-PC ihres Besitzers auf ganze Firmennetzwerke übertragen. Laptops mobiler Mitarbeiter haben wichtige Kundendaten auf ihren Festplatten.

Die Kleinstrechner, die in Fahrzeugen eingesetzt werden, tragen eine hohe Verantwortung für die Sicherheit der Passagiere. Aber auch der Wiederverkaufswert hängt von einem Rechner ab: Der Kilometerstand des digitalen Tachometers darf nicht fälschbar sein – aber muss doch bei einem Ausfall durch einen neuen mit dem gleichen Kilometerstand ersetzt werden können.

In Zukunft werden die Rechner im Haus eine ebenso große Rolle spielen: Sie werden alles kontrollieren, von der Heizung bis zum Entertainment-System. Wer sie kompromittiert, kann Einbrüche sicher planen oder die Bewohner zum Wahnsinn treiben. Nicht zuletzt werden auch die Rechner auf Smartcards in Zukunft wichtige Daten, aber auch Applikationen tragen. Beispiele hierfür sind die Finread-Architektur für Finanzdienstleister oder die Gesundheitskarte.

Allen diesen Plattformen sind einige Dinge gemeinsam:

- Sie sind „mission-critical“, d. h. ein Ausfall oder eine Fehlfunktion hat weitreichende Konsequenzen.
- Sie müssen gelegentlich mit neuer Software versorgt werden.
- Sie operieren selbständig und vernetzt.
- Sie können von den verschiedensten Benutzern verwendet werden.
- Sie sind lokalen Angriffen wie auch Angriffen aus Netzwerken heraus ausgesetzt.
- Sie sind häufig Träger mehrerer Applikationen, teilweise sogar von verschiedenen Herstellern.

Demzufolge sollten diese Plattformen *robust* und *zuverlässig* sein. Das bedeutet, sie sollten auf Programmierfehler oder Attacken nicht nach dem „Alles oder Nichts“-Prinzip reagieren, sondern die Effekte eines Fehlers eingrenzen können.

Da sie mission-critical sind, müssen sie zudem eine sehr hohe Resistenz gegen die Übernahme durch Viren oder Trojaner aufweisen.

Werden heutige Plattformen diesen Anforderungen gerecht?

Malware und ihre Bekämpfung nimmt auf heutigen Sicherheitskonferenzen einen fast unglaublichen Raum ein (z. B. die BSI-Sicherheitskonferenz 2007). Und praktisch keine Plattform, ob Handy, PDA oder PC, ist bis jetzt davor sicher gewesen. Aber wie sehen die Abwehrmaßnahmen aus?

Sehen wir uns dazu aktuelle Ansätze an. Bill Gates hat sowohl 2005 als auch 2006 auf der RSA-Konferenz Vorträge gehalten und dabei zwei sehr verschiedene Ansätze zur Plattformsicherheit vertreten. 2005 – vielleicht unter dem Eindruck des sehr erfolgreichen Updates von Windows durch den Service Pack 2 – propagierte Gates den durch Data Center kontrollierten Home-PC (Abb. 4.1).

In diesem Szenario melden individuelle PCs sicherheitsrelevante Ereignisse an Data Center, wo sie ausgewertet werden. Sind Patches nötig, erfolgen sie automatisch.

Das stellt eine Übertragung des Sicherheitsmodells für Firmen auf die Privatanutzer dar. Aber noch wichtiger: Die Idee des sicheren Einzelrechners wird aufgegeben. Sicherheit ist nur noch als permanenter Patch-Service durch den Hersteller denkbar – natürlich gegen eine monatliche Gebühr. Dieses Modell ist für den Hersteller eines Betriebssystems durchaus attraktiv, schafft es doch eine permanente Einnahmequelle. Für die Nutzer ergeben sich durch die faktische Gleichschaltung mit Firmenrechnern (schon der Name „Personal Computer“ ist im Firmenkontext unsinnig geworden: Persönlich ist an diesen Rechnern lediglich noch die Bedienung, alles andere wie z. B. die Softwareausstattung ist streng kontrolliert) einige gravierende Nachteile.

Die Überwach- und Patchlogik verletzt die Privatsphäre der Nutzer, indem sie notwendigerweise viele Events an die Data Center melden muss, und noch schlimmer: Es handelt sich um eine nachträgliche Sicherheit, sie kann naturgemäß nicht vor sogenannten Zero-Day-Attacken schützen, also wenn ein Virus das erste Mal auftritt. In diesem Fall werden die Systeme kompromittiert und können erst später wieder in einen korrekten Zustand durch einen Patch gebracht werden. Der

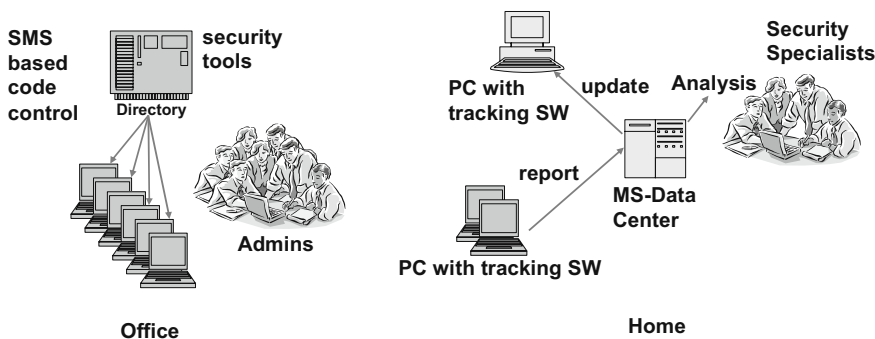


Abb. 4.1 Security für Windows-PCs durch Data Center

von Gates vorgestellte Virens Scanner als Grundausstattung ändert an dieser Problematik nichts, denn auch Virens Scanner können im Wesentlichen nur bereits bekannte Malware detektieren.

Das Problem wird durch Werkzeuge noch verschärft, die einen Patch sofort nach Bereitstellung auf der Download-Seite automatisch analysieren. Diese Werkzeuge vergleichen den Patch mit der Originalsoftware und entdecken dadurch z. B. geänderte Zahlen. Daraus werden dann automatische Exploits generiert und geprüft [Knop]. Die Automatisierung der Exploit-Erstellung bedeutet konkret, dass die Zeit zwischen Veröffentlichung des Patches und Ausnutzung durch Exploits fast auf Null schrumpft und damit die Zahl der noch offenen Rechner sehr hoch ist.

Interessanterweise brachte Gates im nächsten Jahr auf der RSA 2006 entscheidende Erweiterungen zu diesem Sicherheitsmodell vor. Zum ersten Mal war dort von der Notwendigkeit einer sicheren Plattform und sicheren Programmiersprachen zur Vermeidung von Buffer-Overflow-Attacken die Rede. Aber auch das Alles-oder-Nichts-Prinzip in Bezug auf die Rechte von Programmen wurde in Frage gestellt: Neuere Versionen des mitgelieferten Browsers sollen mit reduzierten Rechten laufen können und damit versehentlich ausgeführter Malware weniger Hilfsmittel zur Verfügung stellen.

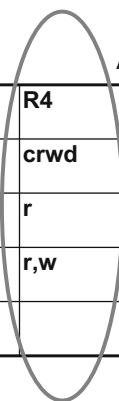
Der neue Internet Explorer von Windows Vista kann ebenfalls in einer Art Sandbox laufen, welche die Zugriffe auf das lokale Dateisystem auf spezielle temporäre Verzeichnisse begrenzt und darüber hinaus Sicherheitslevels einführt (siehe unten). Ob sich schließlich die Idee der Kennzeichnung des Sicherheitsrisikos von Sites durch Farben als hilfreich oder als Falle herausstellen wird, werden wir im Kap. 13 weiter diskutieren.

Wieso diese Kursänderung? Vielleicht ist es doch die Erkenntnis, dass das 2005 vorgeschlagene „Check-And-Patch“-Modell auf die oben erwähnten Rechner und Anwendungen nicht passt: Wenn das Auto bereits in der Hauswand steckt, ist die Meldung, dass ein Patch für die Steuerlogik herunter geladen wird, nicht befriedigend. In Bezug auf die Kosten von Softwareentwicklungen wäre es aber wünschenswert, wenn ein Modell für Sicherheit und seine Mechanismen grundsätzlich verschiedene Anwendungsbereiche abdecken könnte.

Vielleicht ist aber auch das Konzept der Rechtekontrolle bei der Ausführung von Programmen grundsätzlich als Problem erkannt worden. Die meisten der gegenwärtig verwendeten Plattformen verwenden dazu nämlich das Konzept der Access Control Lists (ACL). Wir haben die verschiedenen Techniken bereits im Kap. 3 genauer besprochen und wollen hier nur nochmals die problematischen Eigenschaften dieses Mechanismus vorstellen.

In der ACL ist für eine Ressource vermerkt, welches Recht ein Zugreifender besitzen muss, damit der Zugriff gestattet wird. So hat in Abb. 4.2 der User U1 die Rechte „create, read, write, delete“ auf der Ressource R4.

Entscheidend ist nun, dass beim Zugriff geprüft wird, ob der Zugreifende dieses Recht grundsätzlich besitzt, d. h. statisch und unabhängig von der momentan durchgeführten Aktion. Der Zugreifende wird bei diesem Verfahren übrigens mit der jeweiligen Software gleichgesetzt, die sämtliche Rechte des Benutzers erhält. Es ist diese Übertragung statischer Rechte auf Software, die letztlich zu einer



The diagram shows a table with 'Ressourcen' (Resources) as columns (R1-R5) and 'Principals' (Users) as rows (U1-U4). The 'ACL' label is positioned above the table. A circle is drawn around the R4 column, indicating the focus of the example.

	Ressourcen					ACL
	R1	R2	R3	R4	R5	
U1	r	w	rwd	crwd	crwd	
U2	d	rw		r	r	
U3	rw		rwd	r,w	rw	
U4	rwd	r	re		r	

Abb. 4.2 Beispiel einer ACL

hohen „Ambient Authority“ führt: Das Programm hat grundsätzlich Zugriff auf alle Rechte des Benutzers und – falls es von Malware übernommen wird – stehen diese Rechte auch der Malware komplett zur Verfügung.

In der Capability-Community wird dies gerne am Beispiel des Windows-Spiels Solitaire demonstriert: Wieso kann Solitaire die Platte formatieren? Weil der Benutzer als Administrator arbeitet, und falls Solitaire durch eine Buffer-Overflow-Attacke übernommen wurde, hat auch der Virus dessen Rechte. Die Antwort könnte dann lauten: Dann darf der Benutzer schlicht nicht als Administrator arbeiten. Das ist sehr richtig, vernachlässigt aber, dass dann immer noch alle Rechte des Benutzers zur Verfügung stehen und damit weit mehr als z. B. Solitaire wirklich benötigen würde. Das reicht z. B. aus, um alle Daten des Users zu vernichten/zu missbrauchen.

Zudem gibt es auf vielen Kleinstplattformen keine unterschiedlichen User, da sie nur einem Eigentümer gehören.

Auf der Basis der Zugriffskontrolle nur durch ACLs lässt sich demnach das Problem der Malware sicher nicht lösen. Besonders problematisch ist dabei, dass dieser Mechanismus zwangsläufig zu Alles-oder-Nichts-Entscheidungen der Benutzer führt: Lässt ein Benutzer die Ausführung einer bestimmten Software zu, dann kann er das nur ganz oder gar nicht tun. Keine wirkliche Basis für Entscheidungen von Usern und damit auch ein enormes Problem der Usability: Wie soll diese Gefahr kommuniziert werden? Wie das geschehen kann bzw. ob das überhaupt sinnvoll möglich ist, werden wir im Kap. 13 zu Usability und Security weiter untersuchen.

Im Jahr 2007 hat Microsoft dann mit Singularity auch sein eigenes Konzept eines sicheren Betriebssystems vorgestellt: Es zeichnet sich durch eine eigene, besonders sichere Sprache aus und es unterstützt die modulare Aufteilung von Software in einzelne, voneinander getrennte Bereiche besser als andere Betriebssysteme. So sollen beispielsweise Browser-Plug-Ins in Zukunft nicht mehr als Teil der Anwendung laufen, sondern in einem separaten Prozess, der über RPC-Mechanismen mit dem Hauptteil kommuniziert.

Dennoch ist in der Rede von Bill Gates deutlich geworden, dass der Hauptzweig der Windows-Software von solchen Mechanismen weit entfernt ist (mittlerweile ist Singularity Open Source geworden und man darf das wohl so interpretieren, dass es nicht die neue kommerzielle Plattform der Firma werden wird). Möglich erscheint zumindest, dass die teilweise sehr instabilen und unsicheren Device Driver in eine Zone geringerer Rechte ausgelagert werden könnten.

Das Problem der heutigen Plattformen liegt darin, dass sie zur Vermeidung ambienter Autorität andere Formen der Zugriffskontrolle benötigen würden. Diese wiederum würden radikale Einschnitte in die Architektur von Plattformen, Sprachen und Applikationen voraussetzen. Und nicht zuletzt haben sich die Benutzer bereits an die momentanen Mechanismen gewöhnt (oder sollen vermehrt durch Ausbildung daran gewöhnt werden). Aber der Druck wächst. Unter der Meldung „Windows kollabiert unter dem eigenen Gewicht“ findet sich folgendes Zitat der Gartner-Analysten Michael Silver und Neil MacDonald:

„Weil es versucht, Programme der letzten zwanzig Jahre zu unterstützen, kollabiert der Microsoft-Moloch Windows unter seinem eigenen Gewicht. Mittlerweile ist es der Hardware entwachsen und wird von Minute zu Minute komplexer. Die Kunden fragen sich, ob sie Vista überspringen und gleich auf Windows 7 warten sollen. Soll Windows das Maß der Dinge bleiben, braucht es radikale Änderungen.“ [ZDNet.de]

ZDNet hatte kurz davor Windows Vista auf den zweiten Platz seiner „Top 25 Flops“ gesetzt (hinter Software-Security). Der nächste Abschnitt wird jedoch zeigen, warum eine höhere Plattformsicherheit im Nachhinein so schwierig zu erreichen ist, und dass sich das Problem keinesfalls auf Windows alleine reduzieren lässt, sondern die meisten anderen Plattformen genauso betrifft.

Um noch einmal eine Analogie zur Automobilindustrie zu ziehen: Wir parken unsere Fahrzeuge ohne Sorge neben anderen Fahrzeugen, leihen sie Freunden und installieren Zubehör, ohne Alpträume zu bekommen. Darüber hinaus erwarten wir, dass das Fahrzeug beim Kauf in Bezug auf die Sicherheit vollständig ausgestattet ist und die Bremsen kein Nachrüstgut darstellen. Reparaturen oder Änderungen am Fahrzeug betreffen typischerweise eine eng eingegrenzte Funktionalität. So wird ein Austausch der Blinkerlampe den Motor nicht schädigen oder herausfallen lassen. Das Fahrzeug scheint dem Prinzip der Schadensbegrenzung zu genügen. Allerdings gibt es interessanterweise Grenzen der Modifizierbarkeit und Robustheit. Werden sie überschritten, erlischt die Betriebserlaubnis. Wir werden auf diesen Aspekt speziell im Kap. 13 zu Usability und Security noch eingehen.

Übertragen wir das Beispiel auf unseren Laptop, so treten die Unterschiede deutlich hervor: Wir schließen unseren Laptop ungern an fremde Netze an, wie wir umgekehrt auch ungern fremde Rechner in unsere Netze aufnehmen. Deshalb verriegeln wir unsere Wireless LANS nach Möglichkeit. Bei den heute üblichen Flatrates ist es jedoch nicht die Nutzungsgebühr der Netze, die uns dazu veranlasst, sondern die Angst vor Angriffen auf unsere Plattformen.

Die Angst treibt uns dazu, Malware-Scanner und Firewalls zu kaufen – Nachrüstaktionen, um die fehlende Basissicherheit der Plattformen wenigstens zu mildern. Jede Installation neuer Software ist eine potenzielle Gefahr und das Öffnen

von Attachments verlangt hellseherische Fähigkeiten. Dieses Verhalten ist uns zur zweiten Natur im Umgang mit Rechnern geworden. Es hat sich sogar tief in die Hirne der Entwickler eingegraben, wie die folgende Diskussion der „immutable laws“ in Security zeigt.

4.2 Die „Immutable Laws of Security“: Schadensbegrenzung unmöglich?

Wie tief sich die Ansicht von der Unmöglichkeit von Schadensbegrenzung in die Köpfe von Softwareentwicklern eingegraben hat, zeigt sich an der Liste von „Immutable Laws of Security“, wie sie von Microsoft auf seinen technet-Seiten formuliert wurde. Diese (scheinbaren) Gesetzmäßigkeiten sind deshalb so bedeutsam, weil sie den Rahmen des Möglichen abstecken, der beim sog. Threat-Modelling beachtet werden muss. Anders gesagt: Sie bestimmen, ab wann man sich bei der Entwicklung von Software keine Gedanken zur Sicherheit mehr machen muss.

Die originale Liste enthält zehn Gesetze [Tech]: Sie wurde von Larry Osterman verallgemeinert und auf Slashdot [Slash] heftig diskutiert. Die folgenden Seiten verwenden auch Anregungen aus Diskussionen der CapTalk-Mailing-Liste [Cap] zu diesem Thema. Die entscheidende Frage bei dieser Diskussion ist, ob es sich dabei wirklich um allgemeine Gesetzmäßigkeiten handelt oder ob ihre Gültigkeit schlicht an eine bestimmte Form von Softwarearchitektur gebunden ist. Diese Frage lässt sich durch ein einfaches Verfahren beantworten: Dazu wird die Liste der Gesetzmäßigkeiten durch alle Voraussetzungen ergänzt, die aussagenlogisch nötig sind, um die Gesetzmäßigkeiten in wahre Aussagen zu verwandeln. Stellt sich heraus, dass es zu diesen Voraussetzungen technische Alternativen gibt, dann handelt es sich bei den Gesetzmäßigkeiten eben nicht um solche allgemeiner Art (ein Verfahren nach der sog. strukturalen Textanalyse nach M. Titzmann). Ange wandt auf die angesprochenen „immutable laws“ ergibt sich dabei folgendes Bild:

Law 1: If a bad guy can persuade you to run his program on your computer, it's not your computer anymore.

Anders formuliert: Es gibt Gute und Schlechte. Der Besitzer eines Computers lässt typischerweise nicht Software anderer Leute auf seinem Computer laufen, sondern muss/kann dazu überredet werden, dies zu gestatten. Von einem der „Schlechten“ überredet zu werden, ist ein Fehler des Computerbesitzers. Ist der Besitzer der Software ein „Schlechter“, dann verliert der Computerbesitzer die Kontrolle über den Rechner. Software übernimmt durch den Prozess der Installation und des Startens die komplette Kontrolle über den Rechner.

Das Gesetz Nr. 1 lässt einige interessante logische Lücken: So wird nichts darüber gesagt, wie man die „Guten“ von den „Schlechten“ unterscheiden kann. Ebenso nichts darüber, was die Software beider unterscheidet, sprich, wieso z. B. ein Update von installierter Software in Ordnung ist. Es wird darüber hinaus im-

plizit gesagt, dass der Rechner sich nicht gegen die Installation von Software der „Schlechten“ wehren kann. Und es wird keine Erklärung gegeben, wieso eigentlich der Rechnerbesitzer die Kontrolle durch die Installation der Software verliert, bzw. warum der Rechner sich nicht verteidigen kann.

Angesichts der Voraussetzungen des Gesetzes Nr. 1 kann es kaum den Status eines „immutable laws of security“ beanspruchen. Stattdessen beschreibt es die Art und Weise, wie bestimmte Betriebssysteme funktionieren, als gesetzmäßig. Es existieren Alternativen, bei denen die Sicherheit eines Systems nicht vom psychischen Zustand eines Programmierers abhängt. Wir werden uns im Folgenden noch mit solchen Alternativen beschäftigen.

Law 2: If a bad guy can alter the operating system on your computer, it's not your computer anymore.

Betriebssysteme kontrollieren einen Rechner. „Bad Guys“ können das Betriebssystem abändern. Wenn das passiert, dann verliert der Computerbesitzer die Kontrolle über den Rechner. Es wird keine Erklärung gegeben, warum und wie die Änderung passiert. Bringt man aber Gesetz 1 und 2 zusammen, kann man daraus schließen, dass die Betriebssystemänderung durch Installation fremder Software geschehen könnte. Jegliche Änderung im Betriebssystem hat also den Effekt des kompletten Kontrollverlusts. Für diese drastische Konsequenz wird ebenfalls keine Erklärung gegeben – Betriebssysteme haben demnach einen Alles-oder-Nichts-Charakter. Diese Alles-oder-Nichts-Eigenschaft ist jedoch abhängig von bestimmten Charakteristiken der Implementation und daher kein allgemeines Gesetz, wie wir oben bei der KFZ-Analogie gesehen haben.

Law 3: If a bad guy has unrestricted physical access to your computer, it's not your computer anymore.

Uneingeschränkter physischer Zugriff auf einen Rechner durch einen Unbefugten bedeutet den Verlust der kompletten Kontrolle für den Besitzer. Eine Erklärung für dieses Alles-oder-Nichts-Statement wird nicht gegeben. Es gibt anscheinend keine Möglichkeit, böswillige Manipulationen zu unterbinden, sobald einem Angreifer physischer Zugriff gestattet wird. Aber ist das wirklich so? Zwar sind in der letzten Zeit neue Verfahren des physischen Eingriffs bekannt geworden, etwa eine starke Abkühlung von RAM-Bausteinen, um damit den Speicherinhalt bis zu einem Auslesen durch einen anderen Rechner zu bewahren. Es handelt sich dabei jedoch um sehr aufwändige und spezielle Techniken, die im Alltag nicht vorausgesetzt werden können. Für die Absicherung der Rechner im Alltag gibt es hingegen durchaus Maßnahmen wie Smartcards, Verschlüsselung der Harddisks und generell den Schutz privater Daten durch kryptografische Techniken. Eventuell kann einem Angreifer nicht verwehrt werden, eigene Daten auf den Rechner zu spielen. Somit handelt es sich auch bei Gesetz 3 um keine allgemeingültige Aussage, sondern um eine Folgerung aus Gesetz 1.

Law 4: If you allow a bad guy to upload programs to your website, it's not your website any more.

Offenbar ist dies nur eine Variation des ersten Gesetzes, wobei hier angenommen wird, dass die Plattform der Website den gleichen problematischen Umgang mit „ambienter Autorität“ wie der PC pflegt. Stattdessen ließe sich aber auch die Meinung vertreten, dass Code alleine nicht notwendigerweise eine große Gefahr darstellt, sondern dass es erst die automatische und blinde Übertragung von Rechten auf den Code ist, die ihn gefährlich macht. Die Analogie aus dem Fahrzeugbereich lautet hier: Ein KFZ verliert nicht automatisch durch den Transport gefährlicher Inhalte an Sicherheit.

Law 5: Weak passwords trump strong security.

Demnach gibt es neben schwachen auch starke, gute Passwörter. Es gibt starke Sicherheitsmechanismen in Systemen. Mit starken Passwörtern und starken Sicherheitsmechanismen ist die Sicherheit des Systems in Ordnung. Schwache Passwörter hingegen hebeln die starken Sicherheitsmechanismen aus.

Was aber macht Passwörter schwach? Das Gesetz gibt dazu keine Erklärung, aber wir wissen, dass Menschen dazu neigen, schwache Passwörter zu verwenden, weil sie sich starke Passwörter nicht merken können.

Wie im ersten Gesetz wird hier der User beschuldigt, die Schwachstelle der Security des Systems zu sein. Passwörter sind jedoch nie ein guter Sicherheitsmechanismus, denn sie werden bei jeder Benutzung verraten. Daher müssen bei der Nutzung von Passwörtern grundsätzlich alle Teilnehmer eines Systems als „Good Guys“ vorausgesetzt werden. Diese Annahme skaliert natürlich nicht mit der Zahl der Beteiligten. Darüber hinaus muss festgehalten werden, dass eine scheinbar „starke“ Sicherheitstechnik, die jedem vom Benutzer gestarteten Programm ohne Beachtung der Absichten des Benutzers oder der Ziele des Programms alle Rechte des Benutzers überträgt, eben keine starke Sicherheit eines Systems darstellt.

Law 6: A computer is only as secure as the administrator is trustworthy.

Dieses Gesetz macht zwei sehr interessante Annahmen: Zum einen haben alle Rechner einen allmächtigen Administrator. Zum anderen kann der Administrator seinerseits nur durch Vertrauen kontrolliert werden – das heißt: gar nicht. Es gibt also nichts in einem Rechner, das nicht durch einen Administrator geändert oder umgangen werden könnte.

Auch hier sprechen die Fakten eine andere Sprache: Es gibt viele Ansätze für „mandatory control“, also der Kontrolle von Ressourcen nicht nur durch Rechtebesitz. Dazu gehört z.B. ein abgesicherter Audit-Pfad. Kryptografisch lässt sich auch im Systembereich eine Non-Repudiation erreichen, d.h. der Administrator kann seine Handlungen im Nachhinein nicht abstreiten. Feingranulare Kontrolle durch Sandboxes oder Capabilities ist möglich, die einen allmächtigen Administrator unnötig macht. Das bekanntermaßen problematische Konzept des Administrators in Unix und Windows wird in Gesetz 6 als allgemeingültig angesehen. Die

weitere Forderung des Vertrauens in einen Administrator ergänzt logisch die Forderung, nur Software von „Good Guys“ zu installieren. Sie stellt damit moderne Sicherheitskonzepte schlicht auf den Kopf: Sichere Systeme haben rein gar nichts damit zu tun, dass man Personen vertraut. Im Gegenteil entsteht ein sicheres System dadurch, dass die Sicherheit nicht auf Vertrauen basieren muss. Allein die Tatsache, dass man jemand, der am Rechner als Administrator arbeitet, komplett vertrauen muss, ist bereits der Beweis eines unsicheren Systems und einer unsicheren Systemarchitektur.

Law 7: Encrypted data is only as secure as the decryption key.

Eine Selbstverständlichkeit: Mit Schlüsseln können verschlüsselte Daten wieder entschlüsselt werden, also stellen sie eine schützenswerte Ressource dar. Implizit wird aber auch gesagt, dass der Schutz von Schlüsseln auf heutigen Plattformen ein schwieriges Problem ist und dass es offensichtlich Möglichkeiten gibt, solche Schlüssel zu stehlen. Warum aber haben die heutigen Systeme Probleme damit? Liegt es nicht vielleicht daran, dass „ambient authority“ den Programmen im Überfluss zur Verfügung steht?

Law 8: An out of date virus scanner is only marginally better than no virus scanner at all.

Aktuelle Virens Scanner sind sehr wichtig. Neue Viren tauchen täglich auf und der Virens Scanner muss sie möglichst schnell kennen lernen. Das Mittel hierzu ist der häufige Update der Scannersoftware. Es gibt aber immer eine Zeitspanne zwischen dem Auftauchen eines Virus und dem Moment, in dem der Scanner die Signatur des Virus (sein Erkennungszeichen) lernt. In diesem Zeitraum ist das System hilflos der Malware ausgeliefert.

Es werden keine Gründe für die Bedeutung der Virens Scanner gegeben, sie werden schlicht als Notwendigkeit für die Sicherheit des Systems bezeichnet – was eindeutig eine falsche Aussage ist und lediglich auf den Eigenschaften bestimmter Plattformen bzw. bestimmter Softwarearchitekturen beruht, nämlich genau jenen mit großen Mengen ambienter Autorität. Interessant ist allerdings die Frage, wer eigentlich für das Veralten des Virusscanners verantwortlich ist? Auch hier ergibt sich aus dem Kontext die Antwort: Wieder der User, der seinen Virens Scanner nicht aktuell hält. Das Gesetz 8 hält somit die Auffassung aufrecht, dass die Sicherheit des Systems Aufgabe des Benutzers sein muss, der sich um regelmäßige Updates eines Scanners kümmern muss. Beides ist jedoch falsch: Wirklich sichere Systeme benötigen keine Virens Scanner und die Erzeugung von Sicherheit ist dort auch nicht Aufgabe des Benutzers – zumindest wenn es sich um hochtechnische Bereiche handelt.

Hier lohnt es sich durchaus noch einmal, die Analogie zum Fahrzeug zu ziehen: Dort hat der Fahrer durchaus Aufgaben im Bereich der Sicherheit. Diese Aufgaben beziehen sich aber im Wesentlichen auf den Bereich des Fahrens und nicht der Bremsprüfung bzw. dem Ausstatten des Fahrzeugs mit einer Bremse nach dem Kauf.

Law 9: Absolute anonymity isn't practical, in real life or on the Web.

Die Interpretation dieses Gesetzes hängt stark davon ab, in welchem Sinne „practical“ verstanden wird: Als „nicht möglich“ oder als „schwer zu erreichen“ oder als „nicht wirklich wichtig“. Die Kontextinformation lässt uns ergänzen, dass es wohl in Richtung „nicht wirklich wichtig“ geht. So hat Garfinkel in seiner Thesis zu Usability und Security nachgewiesen, dass die meisten Betriebssysteme keinerlei Anstrengungen machen, die Anonymität der Benutzer überhaupt zu sichern [Garf]. Wir werden in Kap. 13 darauf noch zurückkommen.

Law 10: Technology is not a panacea.

Es gibt eine Grenze zwischen der Verantwortung der Technik und derjenigen der Benutzer. Die Technik kann die Verantwortung der Benutzer nicht komplett übernehmen. Das Gesetz lässt jedoch die Frage offen, wo diese Grenze zu ziehen ist (das betrifft auch die begriffliche Ebene).

Heißt das nun, dass der Fahrzeughalter bei jedem Start die Bremsen, den Zylinderkopf und die Lichter am Fahrzeug prüfen muss? Oder kann er sich auf die Umwelt konzentrieren und auf die Technik des Fahrzeugs vertrauen? So pauschal formuliert klingt das Gesetz 10 eher wie eine Ausrede für die Sicherheitsmängel der heutigen Plattformen.

In der Zusammenschau stellt sich bei den zehn „immutable laws“ immer wieder der Benutzer als Schwachstelle in ansonsten sicheren Systemen heraus, sei es, dass Passwörter schwach sind, den „Schlechten“ die Installation von Software erlaubt wird (bzw. aus schlechten Quellen), dass den falschen Personen (Administratoren) vertraut wird oder dass schlicht der Malware-Scanner nicht up-to-date gehalten wird. Es muss sich um Benutzerfehler handeln, denn die Systeme gehorchen ja den „immutable laws“, können daher gar nicht falsch sein.

Es gibt jedoch etwas zu lernen in diesen „Gesetzen“: Jede Aussage zur Sicherheit ist vom Systemkontext abhängig. Dies gilt bis hin zum richtigen Verhalten des Benutzers bzw. seinen Pflichten. In einem System ohne Schadensbegrenzung werden die Pflichten des Benutzers zwangsläufig ganz andere sein als in einem System, das erfolgreich die Reduktion von Autorität und damit von Schadensmöglichkeit betreibt. Das Benutzerverhalten in Systemen reduzierter Autorität werden wir im Kap. 13 zu Usability und Security weiter untersuchen.

Zum Abschluss lässt sich noch sagen, dass der Ausdruck „immutable“ gleichzeitig völlig falsch und auf eine tragische Art dennoch richtig in Bezug auf die derzeit gängigsten Betriebssysteme ist: Solche Systeme beruhen auf der Absicherung der Zugriffe durch Access Control Lists (ACLs) und dem damit verbundenen hohen Maß an ambierter Autorität, das den Programmen zur Verfügung steht. Diese Systeme sind tatsächlich im Sinne einer Verbesserung der Sicherheit „immutable“. Sie können durch mehr und mehr Warndialoge angereichert werden, aber sie können keine in der Architektur der Systeme angelegte Schadensreduktion bewirken.

In seinem Artikel zu den Regeln des Threat-Modelling hat Larry Osterman in [Oster] die ursprüngliche Liste der „immutable laws“ ersetzt und dabei zusätzliche Aspekte wie die Überschreitung von Vertrauensgrenzen und die privilegierte Ausführung von Code verwendet. Hier die Liste ohne weitere Kommentierung:

Law 1: If the data hasn't crossed a trust boundary, you don't really care about it.

Law 2: If the threat requires that the attacker is already running code on the client at your privilege level, you don't really care about it.

Law 3: If your code runs with any elevated privileges (even if your code runs in a restricted svchost instance) you need to be concerned.

Law 4: If your code invalidates assumptions made by other entities, you need to be concerned.

Law 5: If your code listens on the network, you need to be concerned.

Law 6: If your code retrieves information from the internet, you need to be concerned.

Law 7: If your code deals with data that came from a file, you need to be concerned (these last two are the inverses of law 1).

Law 8: If your code is marked as safe for scripting or safe for initialization, you need to be *really* concerned.

4.3 Sicherheit und Zuverlässigkeit von Betriebssystemen

Ein wesentlicher Baustein beim Bau sicherer Systeme ist zweifellos das Betriebssystem (wir bezeichnen es auch allgemein als *Plattform*). Das Betriebssystem vermittelt zwischen „Innen“ (der Hard- und Software-Plattform selbst) und dem „Außen“ in Form von Input (Daten, Aktionen) durch Andere. In dieser Schnittstellenfunktion ist es an allen Aktionen beteiligt und hat zudem die Macht und die Aufgabe, die Ressourcen eines Systems fair und sicher zu verwalten. Dies tut es typischerweise mithilfe besonderer Privilegien wie dem uneingeschränkten Zugriff auf Hardware. Die Folge sind performante, aber kritische Aktionen.

In diesem Abschnitt wollen wir die Sicherheitsprinzipien von Betriebssystemen unter zwei Aspekten untersuchen: Einmal werden wir uns die Architektur des Betriebssystems und ihre Konsequenzen für die Sicherheit und Zuverlässigkeit des Systems selbst ansehen. Zum anderen werden wir untersuchen, wie Betriebssysteme Applikationen in Bezug auf Sicherheit unterstützen können, z. B. indem sie durch geeignete APIs und Architektur das POLA-Prinzip unterstützen. Dabei werden wir eine Reihe von Konzepten und Mechanismen zur Schadensreduktion kennen lernen, wie z. B. Isolation durch virtuelles Memory, Privilegienstufen in der Hardware und Software und Rechtekontrolle durch ACLs. Dabei werden wir jeweils untersuchen, wie effektiv die Ansätze sind und wie sie degenerieren. Beginnen wir mit der Sicherheit des Systems bzw. der Plattform selbst.

4.3.1 Klassische Sicherheitsarchitektur in Betriebssystemen

Die klassischen Prinzipien eines Betriebssystems zur Absicherung von Zugriffen bestehen im Wesentlichen aus Isolationstechniken wie unterschiedlichen Privilegien bzw. Modi während der Ausführung von Programmen sowie der Trennung der Adressräume von Prozessen durch Hardware.

Der Privilegienmodus eines Prozesses wird in den meisten Fällen recht simpel bestimmt: Ist ein bestimmtes Bit in der CPU gesetzt, hat der Code des Prozesses vollen Zugriff auf die Hardware wie etwa CPU und I/O-Geräte, ohne jede weitere Kontrolle. Man spricht in diesem Fall davon, dass der Prozess im sog. *Kernel Mode* läuft. In diesem Modus wird nur Code des Betriebssystems selbst (genauer: des Betriebssystemkerns) ausgeführt, dem komplett vertraut wird. Ist das Bit nicht gesetzt, läuft der Prozess im sog. *User Mode* ohne spezielle Privilegien. In diesem Modus hat er keinen Zugriff auf gefährliche Hardwarefunktionen und kann bei Fehlverhalten leicht vom Betriebssystem beendet werden. Das Bit, das über den Modus eines Prozesses entscheidet, kann natürlich auch von Anwendungen nicht ohne weiteres geändert werden, sondern ein Prozess wechselt vom User Mode in den Kernel Mode durch Auslösen eines sog. *Software-Interrupts*, der letztlich nichts anderes als einen kontrollierten Zugang zu den Kernel-Funktionen (s. Abb. 4.3) darstellt.

Der Übergang vom User-Mode in den Kernel-Mode wird gerne mit dem Wechsel der Ausführungsrechte eines Programms verwechselt, wenn es unter der Identität eines normalen Users bzw. mit der Identität des Superusers oder Administrators läuft. Beides sind völlig getrennte Dinge. Wird ein Programm von einem Administrator ausgeführt, so muss es genauso wie ein Programm, das unter einer regulären (einfachen) Identität eines Users läuft, für kritische Funktionen in den

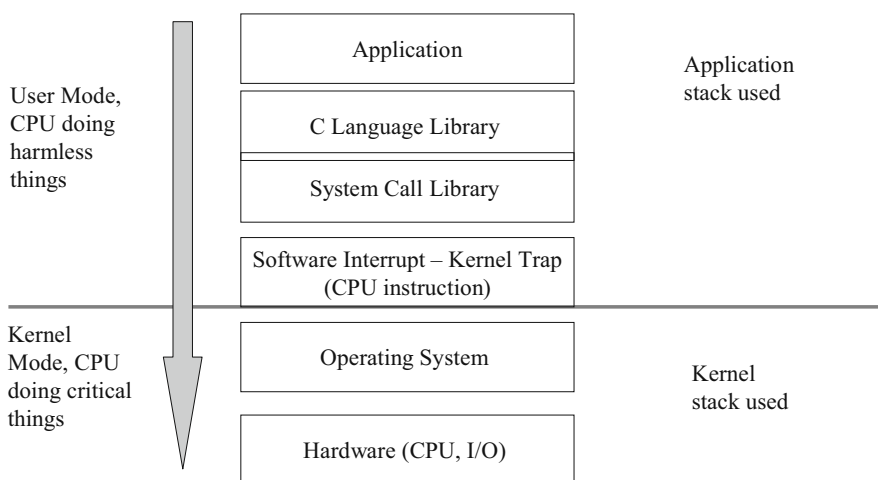


Abb. 4.3 Übergang vom User zum Kernel Mode

Kernel-Mode wechseln. Es stehen lediglich erweiterte Kernel-Funktionen zur Verfügung, wenn die Rechte des Administrators verwendet werden.

Der Wechsel vom User-Mode in den Kernel-Mode ist zudem recht aufwändig in Bezug auf die Performance und die Architektur tendiert im Laufe der Zeit gerne dazu, sehr viele Funktionen innerhalb des Kerns anzusiedeln und damit die Fehleranfälligkeit zu erhöhen. Versuche, mehrere Privilegienstufen einzuführen, waren bisher kaum erfolgreich, und es ist durchaus fraglich, ob zwei weitere Stufen tatsächlich eine bessere Aufteilung der Programme nach Gefahrenpotenzial bringen würden. Tatsächlich ist das Konzept unterschiedlicher Modi oder Privilegienstufen durchaus problematisch für die Softwarearchitektur, wie Abb. 4.4 zeigt.

Es gibt mehrere Gründe für die Probleme mit Privilegienmodi innerhalb der Softwarearchitektur. Typischerweise benötigen die Funktionen innerhalb eines Levels unterschiedliche Rechte aus den auf dem jeweiligen Level vorhandenen Rechten. Dies führt zunächst dazu, dass allgemein zuviel ambiente Autorität an die Funktionen gegeben wird. Außerdem stellt jede Funktion innerhalb eines Levels eine Gefahr für alle anderen Funktionen auf diesem Level dar, da eine Separierung nicht möglich ist. Dann muss die Softwarearchitektur sicherstellen, dass bei jedem Wechsel zurück aus einem privilegierten Modus der privilegierte Modus wieder abgeschaltet wird. Ein Umgehen dieser Funktion bewirkt, dass normaler Code plötzlich mit zusätzlichen Rechten läuft. Hierzu werden wir im Kap. 5 zur Java Sprach- und Plattformsicherheit Softwaremechanismen diskutieren, die dies garantieren sollen. Und letztlich entstehen am Übergang vom unprivilegierten zum privilegierten Modus zwei klassische Probleme der Zugriffskontrolle: Erstens muss geprüft werden, ob der Übergang grundsätzlich erlaubt ist, d. h. darf der unprivilegierte Code den privilegierten Code aufrufen? Und zweitens muss der privilegierte Code sicherstellen, dass die übergebenen Argumente mit den Rechten des Aufrufers in Ressourcen umgewandelt werden. Ansonsten entsteht das berüch-

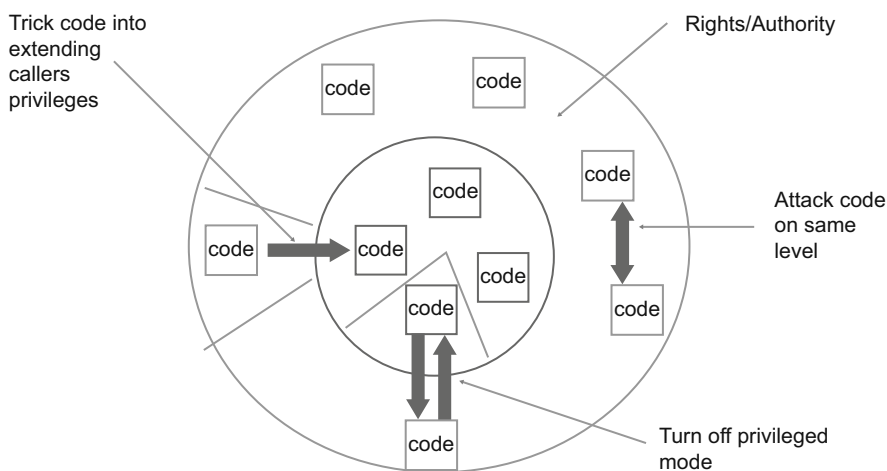


Abb. 4.4 Sicherheitsprobleme der Modus-Architektur

tigte „Confused Deputy“-Problem, bei dem der privilegierte Code fälschlicherweise seine erweiterten Rechte verwendet und dem Aufrufer dadurch eine ungewollte Privilegienerweiterung verschafft.

Das zweite klassische Mittel der Isolation in Betriebssystemen – ebenfalls kritisch für die Performance des Systems – ist die virtuelle Speichervergabe pro Prozess, d. h. die Trennung der Adressräume von Prozessen während der Ausführung. Performance-kritisch ist diese Maßnahme weniger wegen der Kosten der damit verbundenen Adressumsetzung – dies erledigt die sog. Memory Management Unit (MMU) dank assoziativem Cache und der Integration in die CPU (s. Abb. 4.5) recht effektiv. Die Trennung bedeutet jedoch auch, dass die Kommunikation zwischen Prozessen bzw. der Transport von Daten zwischen Kernel und Prozessen Kopieraktionen zwischen unterschiedlichen Adressräumen sowie meist zusätzliche Context Switches der beteiligten Prozesse erfordert.

Dies ist die häufigste Form der Isolation von Prozessen in den gängigen Betriebssystemen. Es hat jedoch Versuche gegeben, auf der Basis von getagtem Memory (z. B. in Capability-Systemen, siehe [Levy]) solche Trennungen auf andere Weise zu erreichen. In den letzten Jahren sind sogar softwarebasierte Methoden der Trennung von Code immer beliebter geworden. Das Microsoft-Betriebssystem Singularity ist ein solcher Kandidat und wir werden es weiter unten vorstellen. Dort existieren weder unterschiedliche Privilegienlevel noch eine hardwarebasierte Isolation durch virtuelles Memory. Stattdessen kontrolliert das System durch Compiler und Laufzeitumgebung sehr genau die Propagation von Referenzen und erlaubt dem Code nicht, beliebige eigene Referenzen zu erzeugen.

Isolation durch getrennte Adressräume bzw. Modi ist demnach das wesentliche Element der Zugriffskontrolle von Plattformen. Natürlich muss dieses Konzept der Kontrolle von Autorität mit der Kontrolle von Rechten ergänzt werden. Dies geschieht durch die bereits eingangs erwähnten Access Control Lists (ACLs), die je-

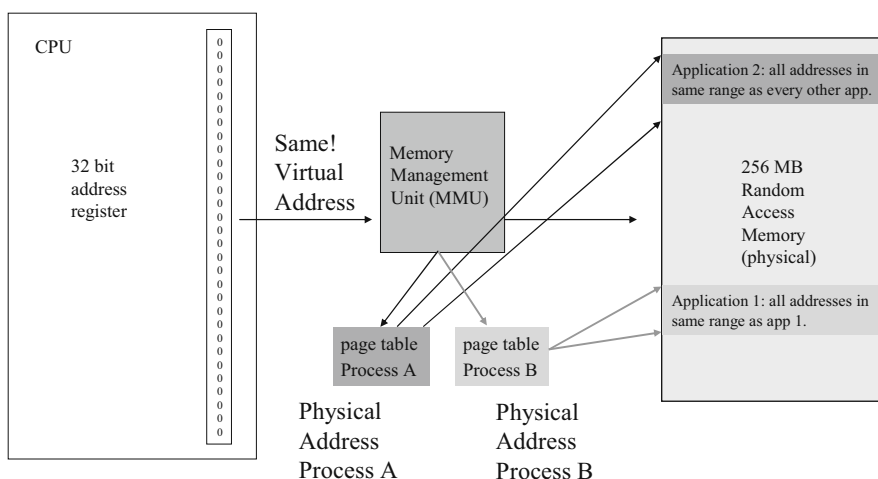


Abb. 4.5 Isolation durch Trennung der Adressräume

dem Benutzer statisch gewisse Rechte auf Ressourcen zubilligen. Auch hier findet im Grunde genommen eine Isolation statt: Benutzer werden anhand ihrer Rechte voneinander isoliert. Innerhalb des Kontextes eines Benutzers jedoch gibt es keine Isolation: Alle vom Benutzer gestarteten Programme erhalten die gleichen Rechte. Im Falle des Superusers oder Administrators sind das meist alle Rechte.

4.3.2 Kernel-Architektur: Microkernel vs. monolithischer Kernel

Andrew Tanenbaum hat in [Tan] jüngst die Diskussion um die richtige Architektur von Betriebssystemen erneut angefasst. Jahre nach dem öffentlichen Streit um die richtige Architektur für Linux – es standen der Microkernel mit Tanenbaum als Vertreter sowie der klassische monolithische Kernel mit Thorvalds als Vertreter zur Diskussion – stehen wir wieder vor der gleichen Fragestellung. Damals, d. h. vor mehr als zehn Jahren, haben vor allem zwei Argumente den Streit entschieden: Performance und Bekanntheit der Technologie. Es wussten weit mehr Leute, wie ein monolithischer Kernel gebaut wird, als es Experten für moderne Microkernel gab. Durch die notwendige Kommunikation der Komponenten des Microkernels untereinander war zudem die Performance zunächst schlechter.

Heute führt Tanenbaum zwei neue Argumente für den Microkernel ein, die an Bedeutung durch die Ausbreitung der Rechner in alle – darunter auch lebenswichtigen – Gebiete dramatisch gewonnen haben: Sicherheit und Zuverlässigkeit (denken Sie nur an die in Autos oder Flugzeugen verwendeten Plattformen). Tanenbaum ist der Ansicht, dass der Microkernel beides weit besser garantieren kann als ein monolithischer Kernel. Im Grunde spricht er dabei von der besseren Isolierung der Komponenten voneinander durch Kapselung, also Objektorientierung. Was besser für die Zuverlässigkeit von Software ist, ist jedoch auch besser für die Sicherheit: Die Kapselung führt dazu, dass POLA wesentlich besser durchgesetzt werden kann.

Der Zusammenhang zwischen Zuverlässigkeit und Sicherheit ist extrem eng. Wir haben bereits am Beispiel von Buffer-Overflow-Attacken gesehen, wie eine Schwäche in der Zuverlässigkeit von Software sofort für einen Angriff auf die Sicherheit ausgenutzt werden kann. Nach Tanenbaum hat jedes System Fehler, und zwar meist umso mehr, je mehr Code vorhanden ist. Dies lässt sich reduzieren, aber nicht vermeiden. Daraus folgt aber, dass die Softwarearchitektur auf die Schadensreduzierung ausgerichtet sein muss: Komponenten müssen abstürzen können, ohne dass das gesamte System zusammenbricht. Komponenten müssen zu einem Restart fähig sein. Und Komponenten müssen voneinander isoliert sein – was sich nicht kennt, kann sich auch nicht direkt negativ beeinflussen. Auf Sicherheit angewendet, bedeutet dies: Softwarekomponenten müssen dem Prinzip POLA folgen.

Tanenbaum argumentiert außerdem stark gegen shared data im Kernel und plädiert für eine bessere Kapselung von Daten und Funktionen. Sehen wir uns dazu zunächst die unterschiedlichen Architekturen an.

Der monolithische Kernel fasst alle wesentlichen Funktionen des Betriebssystems (Sicherheit, File-System, Prozessverwaltung, Devices) in einem Image zusammen. Dabei greifen die unterschiedlichen funktionalen Blöcke auf globale Datenstrukturen und Funktionen zu. Der Vorteil dieses Ansatzes liegt in der globalen gegenseitigen Verfügbarkeit der Funktionen und Daten, die mit der maximalen Geschwindigkeit von überall her aufgerufen und manipuliert werden können. Manche Funktionen, speziell Treibermodule, können über ein Modulkonzept dynamisch geladen werden (s. Abb. 4.6). Dabei ist eine Auflösung der Referenzen auf Kernel-Daten oder Funktionen zur Laufzeit nötig.

Der große Vorteil des monolithischen Kernels – die globale Verfügbarkeit – äußert sich aber gleichzeitig auch als eine große Schwäche: Die Isolation von Komponenten – eines der zentralen Kriterien für Qualität und Sicherheit – ist nicht vorhanden: Ein Fehler in einer Komponente reißt das ganze System mit sich. Die fehlende Isolation bedeutet auch, dass alle Kernel-Komponenten – also auch die weniger zuverlässigen – mit höchsten Privilegien arbeiten. Diese Privilegien können verstreut über den ganzen Code aufgerufen werden, z. B. indem kritische CPU-Befehle wie HLT oder I/O Control verwendet werden.

Die Manipulation globaler Datenstrukturen durch unzuverlässige Treibersoftware ist ein weiterer Grund für dieses schlechte Verhalten hinsichtlich Zuverlässigkeit und Sicherheit.

Code im Kernel-Mode erhält maximale Privilegien, und je mehr Kernel-Code vorhanden ist, desto mehr Code läuft mit diesen Privilegien – oft, ohne sie wirklich zu benötigen: Ein klarer Verstoß gegen POLA und ein Zeichen für viel zu viel „ambient authority“.

Mikrokernel (auch Multi-Server OS genannt) funktionieren hingegen nach anderen Prinzipien. Hier laufen im Kernel-Mode mit höchsten Privilegien nur dieje-

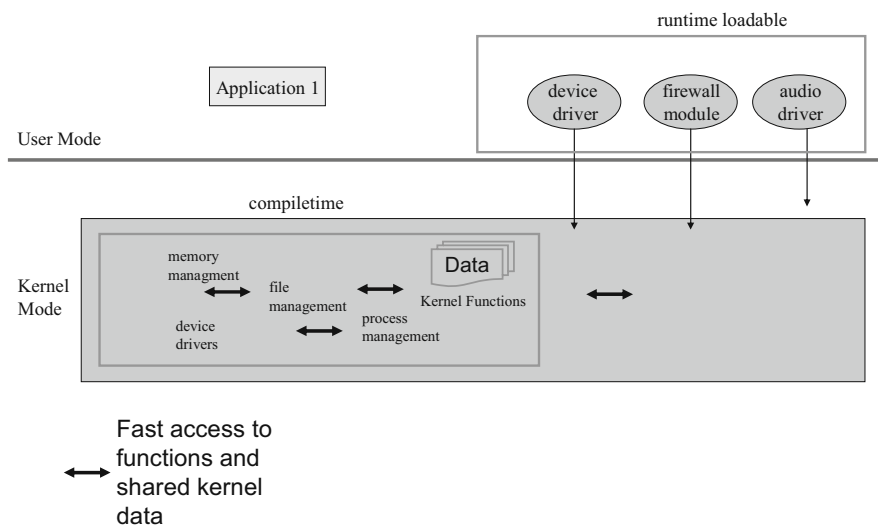


Abb. 4.6 Monolithischer Kernel

nigen Funktionen, die diese unbedingt benötigen. Der Rest der Betriebssystemfunktionen ist ausgelagert in Serverprozesse, die im User-Mode laufen (Achtung: Das heißt nicht, dass diese Prozesse mit den Privilegien eines normalen Nutzers laufen. Sie laufen lediglich gegenüber der CPU in einem nicht-privilegierten Modus, können also nicht direkt kritische CPU-Befehle ausführen. Zur Verdeutlichung: Der allmächtige Administrator eines Systems muss zum Abschalten der CPU ebenfalls Funktionen des Betriebssystems verwenden, die zur Ausführung in den Kernel-Mode wechseln müssen, da nur dort der HLT-Befehl ausgeführt werden kann. Die Privilegien des Administrators bestehen also darin, dass er die Funktionen zum Wechsel in den Kernel-Modus aufrufen darf).

Prozesse sind durch virtuelles Speichermanagement voneinander isoliert und können sich gegenseitig nur indirekt beeinflussen. Stürzt eine solche User-Mode-Komponente ab, dann kann dies von einem Monitor im System festgestellt werden, der anschließend die Komponente wieder startet: Ein wichtiger Schritt in die von Tanenbaum geforderte Richtung hin zu Computern „ohne Reset-Knopf“.

Abbildung 4.7 zeigt die bessere Isolierung der Komponenten voneinander beim Microkernel und ihre Verbindung durch Kommunikationsprotokolle. Der Preis dieser Architektur liegt in eben dieser Kommunikation: Sie ist aufwändiger als im Falle des monolithischen Kernels, da sie häufige Wechsel zwischen User- und Kernel-Mode bedingt.

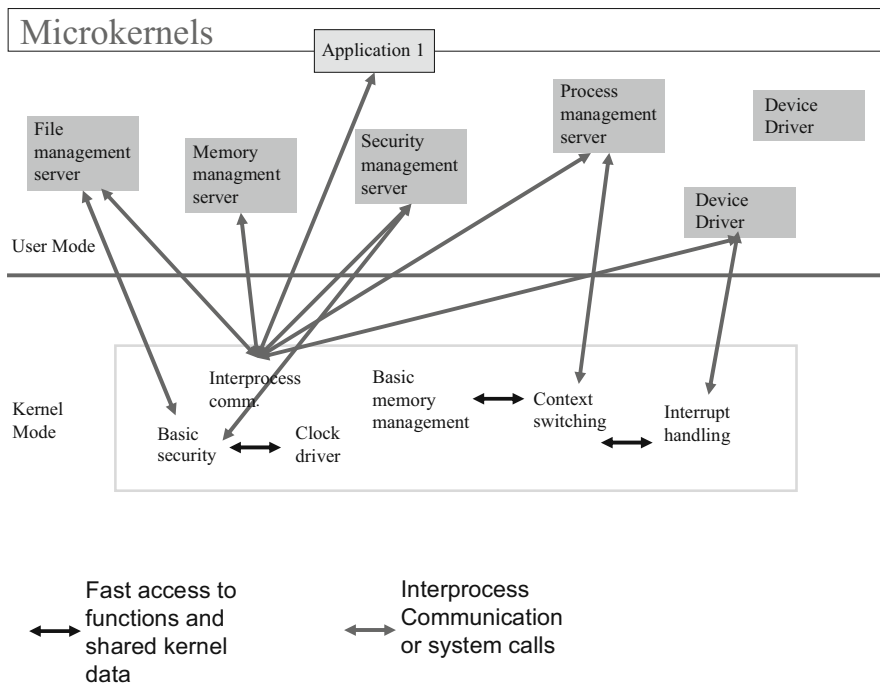


Abb. 4.7 Kommunikation der Komponenten beim Microkernel

Ein besonderer Vorteil des Mikrokernels ist die Auslagerung wesentlicher Teile der Gerätetreiber in den User Space. Lediglich da, wo sie direkt privilegierte CPU-Befehle verwenden müssen (interrupt handling etc.), laufen Routinen im Kernel Mode.

Sicherheitstechnisch liegt der entscheidende Vorteil eines Microkernels in seiner sehr schmalen *Trusted Computing Base* (TCB). Dadurch ist der Anteil des Codes, der auf der höchsten Privilegienstufe läuft, sehr gering – nach wie vor eines der wichtigsten Kriterien bei der Beurteilung der Sicherheit von Systemen.

Darüber hinaus bietet der Microkernel noch weitere Vorteile, die dann – verglichen mit Lösungen auf Basis von Virtualisierung alleine (siehe unten) – sich auch positiv für die Sicherheit auswirken. So zählt Gernot Heiser in [Heis08] als Vorteile speziell im Bereich Embedded Control auf:

- Geringere Kosten für die Interprozesskommunikation,
- integriertes Scheduling,
- komplettes, wenn auch minimales API,
- Mechanismus-/Policy-Trennung und
- nur geringfügig größer als ein reiner Hypervisor.

Embedded-Control-Systeme benötigen häufig Interaktionen zwischen Services, die durch eine totale Isolation in Form mehrerer virtueller Maschinen sehr teuer werden. Außerdem benötigen diese Systeme oft eine globale Ressourcenbetrachtung (Deadlines für Scheduling, Stromverbrauch etc.), die ebenfalls einen gemeinsamen Ressource-Manager voraussetzen. Der Einsatz von mehreren virtuellen Maschinen, auf denen dann komplette Gastbetriebssysteme laufen, scheitert meist an den zur Verfügung stehenden Hardwareressourcen, vergrößert jedoch auch die TCB enorm und ist daher unsicher.

Dennoch steht auch ein Microkernel vor schwierigen Sicherheitsproblemen, speziell was den Verbrauch von Ressourcen wie dem Speicher angeht. Bei der impliziten Allokation von Kernel-Memory durch Applikationen entsteht die Gefahr, dass dem Kernel selbst der Speicher ausgeht bzw. dass DOS-Attacken möglich werden. Bei dem eingangs erwähnten Trend, immer mehr Services und Applikationen in Fahrzeugen etc. auf einem Rechner zusammenzufassen, ist dies eine beträchtliche Gefahr für die Sicherheit des Systems. Laut Heiser funktionieren die klassischen Hilfsmittel gegen Kernel-DOS-Attacken, wie z. B. Quotas, im Bereich Embedded Control nicht besonders gut.

Im Falle des seL4-Kerns (secure embedded L4) wurde diese Problematik dadurch gelöst, dass die Ressourcenallokation explizit gemacht und an die Applikationen in Form von Capabilities auf Speicherbereiche delegiert wurde (siehe [HEKKP], S. 4, Kernel API: seL4 sowie für einen schnellen Überblick über die Techniken im L4 Umfeld auch [Zais]). Im Umfeld dieses Microkernels und seiner kommerziellen Variante OKL4 von den Open Kernel Labs findet sich eine ganze Reihe sehr interessanter Technologien im Bereich Sicherheit. So wird dort die beweisbare Sicherheit des Kernels aufgrund formaler Verfahren angestrebt. Wir werden im Abschnitt zu Virtualisierung und formale Methoden darauf zurückkommen.

4.3.3 Die Sicherheit von Plug-In-Architekturen

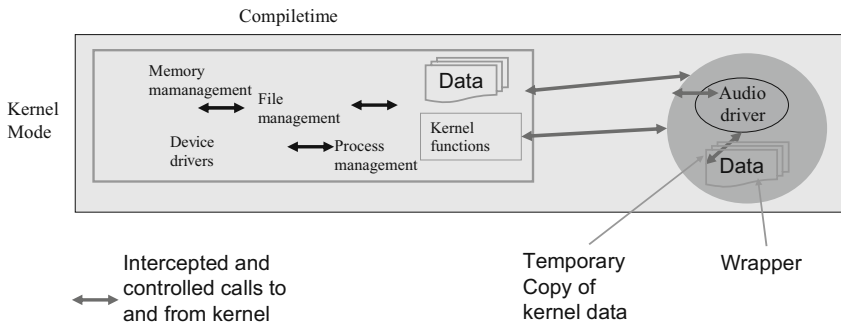
Wenden wir uns jetzt einem Problem zu, das zunächst aussieht, als wäre es nur für Betriebssysteme relevant: Gerätetreiber. Allerdings kommt heute kaum eine Basissoftware mehr ohne ein Erweiterungskonzept aus. Beispiele dafür sind Browser-APIs, Office-Pakete und Entwicklungsumgebungen wie Eclipse. Jüngst hat auch OpenOffice.org angekündigt, ebenfalls ein Erweiterungskonzept zu integrieren, basierend auf der Mozilla-Technik. Wir werden die Sicherheitsproblematiken dieser Erweiterungen am Beispiel von Browsern noch im Detail diskutieren. Hier sei nur vorausgeschickt, dass es im Allgemeinen üblich ist, dass solche Erweiterungen (auch *Plug-Ins* genannt) Zugriff auf sämtliche Funktionen des Basissystems erhalten und es damit komplett übernehmen oder kompromittieren können. Leider äußert sich diese Architekturschwäche in einer großen Zahl von Sicherheitsproblemen, verursacht durch Erweiterungen.

Bei aktuellen Betriebssystemen machen die Gerätetreiber ca. 70 Prozent des Codes aus und variieren dabei stark in der Qualität. Trotzdem haben sie alle vollen Zugriff auf Kernel-Daten und -Funktionen. Microsoft hat in der Vergangenheit bestätigt, dass die Treiber das Hauptproblem für die Zuverlässigkeit von Windows XP darstellen.

Gleichzeitig sind Gerätetreiber aber auch das entscheidende Softwaremittel für die Erweiterbarkeit von Systemen: Plug-In Patterns (oder Framework Patterns) haben wir mittlerweile auch im Bereich der Applikationen (Firefox und IE Extensions und Plug-ins) und Tools (Eclipse Services und Plug-Ins) kennen gelernt. Deshalb ist es wichtig, das Prinzip der Erweiterbarkeit auf seine Probleme der Zuverlässigkeit und Sicherheit hin zu untersuchen (Gerüchten zu Folge sind mittlerweile auch große Teile des Firefox-Browsers durch Extensions und Plug-ins geprägt und sie führen anscheinend auch dort, wie auch beim Internet Explorer, zu großen Problemen bezüglich Stabilität und Sicherheit). Microsoft erwähnt im Zusammenhang mit seinem Forschungsbetriebssystem Singularity, dass in Zukunft Plug-Ins für den Internet Explorer in einem eigenen Adressraum laufen sollen, um die Übernahme des Basissystems zu vermeiden. Das Singularity-Betriebssystem verwendet darüber hinaus eine „Sealed Process Architecture“ [Hunt], bei der eine nachträgliche Erweiterung von Prozessen durch dynamisch ladbaren Code untersagt ist. Weiter unten werden wir uns mit der sicheren Architektur von Browsern auseinandersetzen und am Beispiel des DARPA-Browsers untersuchen, wie die Kritikalität von Plug-Ins herabgesetzt werden kann.

Tanenbaum zeigt einige Ansätze auf, die Schwächen traditioneller monolithischer Kernel in Bezug auf Treibersicherheit auszumerzen. Ein Beispiel ist das NOOKS-Projekt, bei dem die Treiber in sog. Wrapper verpackt werden (s. Abb. 4.8). Jeder Zugriff von oder zum Treiber wird durch die Funktionen des Wrappers abgefangen und kontrolliert. Beim Aufruf einer Treiberfunktion werden die Speicherbereiche des Kernels auf „read-only“ gesetzt: Der Treiber kann also nicht unkontrolliert Kernel-Daten oder Code modifizieren. Muss der Treiber Datenstrukturen des Kernel manipulieren, werden sie in den Speicherbereich des

User Mode

**Abb. 4.8** Abgesicherter monolithischer Kernel

Treibers kopiert, dort manipuliert (und anschließend kontrolliert) und zuletzt in atomarer Form zurückkopiert. Ein schlechter Treiber kann also keine wichtigen Datenstrukturen beschädigen.

Zur Verdeutlichung des letzten Punktes: Ohne das kontrollierte und atomare Zurückkopieren der Änderungen geschieht mit den Kernel-Datenstrukturen das gleiche wie mit einer Datenbank, die ohne relationale Integritätsregeln von mehreren Applikationen benutzt wird: Irgendwann wird eine Applikation aufgrund von Softwarefehlern unvollständige Records zurücklassen.

Zum Einsatz kommen hier also die Prinzipien Isolation und Interception (Interposition), um POLA zu erreichen. Die Grenzen des Ansatzes sind bedingt durch seinen hohen manuellen Aufwand bei der Erstellung der Sandboxes (denn nichts anderes sind diese Wrapper), und der Tatsache, dass der Treiber gegenüber der CPU nach wie vor im höchstprivilegierten Modus läuft und z. B. die CPU anhalten könnte.

Eine noch stärkere Isolierung gegenüber Kernel und Hardware lässt sich durch den Einsatz von Virtual Machines und der Verlagerung der Treiber in den User Space erreichen. In Abb. 4.9 läuft der Kern des Betriebssystems auf einer VM und die meisten Device Driver für diesen Kern ebenfalls unter einer eigenen VM. Den Treibern ist es nicht mehr möglich, direkt im privilegierten Modus auf die CPU zuzugreifen. Interprozesskommunikation ist allerdings nötig, um die Daten und Befehle vom Kernel zu den Treibern und zurück zu schaffen. Ein schlechter Treiber kann hier das System nur noch wenig beeinflussen.

Dies waren Beispiele für die Isolierung durch getrennte Speicherbereiche und ansatzweise auch durch Interception. Wenden wir uns im nächsten Schritt den rein softwarebasierten Methoden zur Isolation von Plug-Ins zu. Plug-Ins sind softwaretechnisch gesehen sog. *Inversion-of-Control* (IOC) Architekturen, bei denen ein Basissystem die Funktionen von Plug-Ins aufruft. In diesem Sinne kann man auch z. B. Enterprise Java Beans als Plug-Ins ansehen, da der Container, in dem diese Plug-Ins laufen, zu bestimmten Zeiten festgelegte Methoden dieser Beans aufrufen wird.

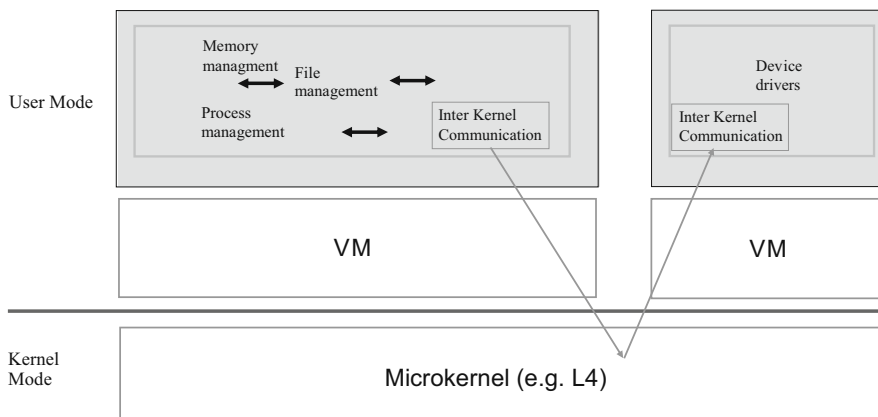


Abb. 4.9 Durch Einsatz von VMs gesicherter Kernel

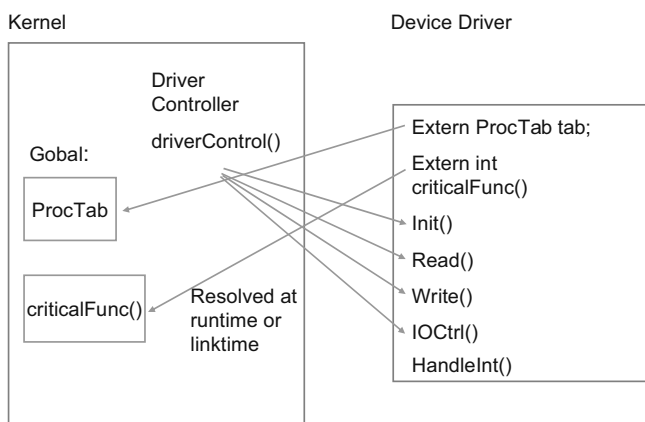


Abb. 4.10 Treiberschema unter C

Entscheidend für die Isolation der Komponenten voneinander ist nun, was sie jeweils voneinander sehen können. Schauen wir uns zuerst an, warum das normale Treiberschema in C oder C++ hier ungenügend ist (Abb. 4.10).

Durch die Verwendung globaler Daten und Funktionen ist es jedem Treiber erlaubt, diese anzusprechen. Der Compiletime oder Runtime Linker wird hier die Referenzen auflösen und den Zugriff dadurch ermöglichen. Der Kernel hat keine Möglichkeit, dies zu verhindern. Da in C oder C++ ohnehin keine Sicherheit in Bezug auf den Speicher erreichbar ist (ein Pointer kann jederzeit jede beliebige Stelle abändern), muss man bei der Verwendung dieser Sprachen zwangsläufig auf die Isolation von Speicherbereichen setzen. Anders sieht es aus, wenn typsichere Sprachen wie Java oder C# eingesetzt werden können, die die Memory Protection garantieren können (Abb. 4.11).

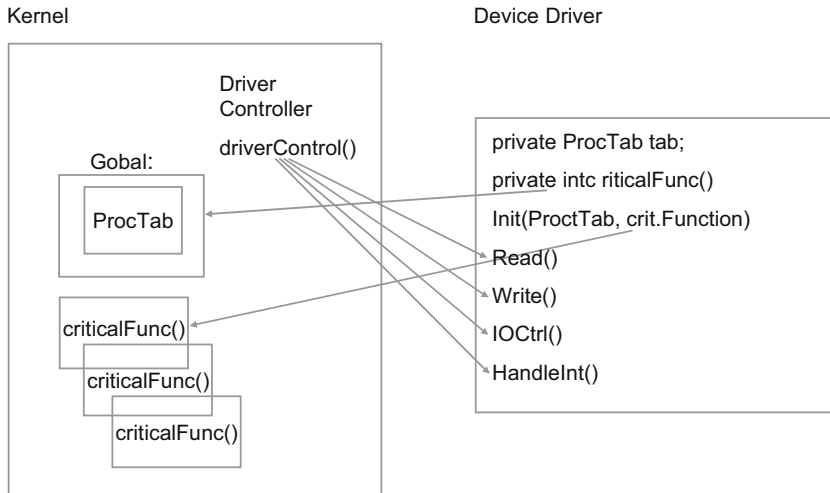


Abb. 4.11 Treiberschema unter Java oder C#

Der Schnittstelle zwischen Kernel und Treibern kommt jetzt die entscheidende Bedeutung zu, speziell den Argumenten. Es gelten folgende Regeln:

- Der Driver kann ausschließlich über die Argumente und deren Methoden auf Kernel-Daten oder Funktionen zugreifen. Es gibt keine andere Möglichkeit, auch nicht durch die Navigation von globalen Verzeichnissen oder durch sog. *Composite Objects* (Design Pattern), die über ihre Tree-Navigation Zugriffe auf viele andere Objekte gestatten.
- Der Treiber macht eigene Abhängigkeiten und Bedürfnisse im Interface der `init`-Methode sichtbar: Dort steht, was der Treiber vom Kernel benötigt, um zu funktionieren.
- Der Kernel ist in der Lage, Proxy-Objekte oder speziell für den Treiber zugeschnittene Closures (das sind initialisierte higher-order Functions oder deren objektorientiertes Äquivalent bzw. deren Simulationen durch Command Patterns) herzustellen und an den Treiber zu übergeben. Damit kontrolliert der Kernel komplett die Ausführungsumgebung des Drivers.

Diese Prinzipien können leicht abgewandelt werden: So könnten z. B. die Abhängigkeiten bzw. Bedürfnisse des Treibers auch durch Meta-Daten statt im Interface der `init`-Funktion festgehalten werden. Wir werden dafür später ein Beispiel im Rahmen der Besprechung der Open Services Gateway Initiative (OSGI) kennen lernen – ein dynamisches Framework zur Installation und Kombination von Services (also Plug-Ins), bei dem die Isolation der Services voneinander vor allem durch isolierte Name Spaces erzeugt wird.

Die strikte Durchführung des Prinzips: *Zugriff nur über erhaltene Referenzen (Argumente)* führt uns direkt ins Gebiet der Objekt-Capabilities – einem weiteren System zur Isolation von Komponenten, diesmal jedoch durch während der Lauf-

zeit übergebene Referenzen (Keys). Wir werden dieses Prinzip bei der Besprechung der sicheren DARPA-Browserarchitektur später im Detail kennen lernen. Dieses Prinzip ist eine Verschärfung des Inversion- of-Control-Prinzips und des *Dependency-Injection*-Prinzips, bei dem die Abhängigkeiten einer Komponente durch Konstruktoren, Methoden etc. aufgelöst werden. Abbildung 4.12 zeigt das Verhältnis dieser Prinzipien zueinander.

Beide Prinzipien – getrennte Namespaces und Object Capabilities – sind geeignet für reine Isolation in Software und lassen sich natürlich auch miteinander kombinieren.

Abschließend noch eine Anmerkung zur Wartbarkeit solcher Software: Dadurch, dass Abhängigkeiten a) strikt begrenzt und b) deutlich sichtbar gemacht werden, steigt die Wartbarkeit solcher Software enorm. Zudem lassen sich die Konsequenzen von Änderungen gut durchschauen: Eine Komponente, die eine Referenz nicht erhält, kann sie auch nicht nutzen oder missbrauchen.

Gerade die Deklaration von internen Abhängigkeiten durch eine Komponente würde typische Sicherheitsentscheidungen sehr viel einfacher machen: Darf eine Komponente überhaupt installiert werden? Dies hängt in erster Linie davon ab, was sie benötigt, um ihre Arbeit zu tun. Momentan – und wir werden uns damit noch bei der Absicherung von Applikationen und Servern beschäftigen – muss in einem Reverse-Engineering-Prozess versucht werden, alle Abhängigkeiten einer Software zu entdecken, damit z. B. die Regeln für eine Sandbox wie SE-Linux oder Systrace aufgestellt werden können.

Es darf jedoch nicht verschwiegen werden, dass der Einsatz solcher Isolationsmethoden nicht ohne Änderungen im Design der Software vor sich gehen wird: Jede Methode und jede Referenz muss darauf geprüft werden, ob sie zuviel von einer Komponente nach außen offen legt.

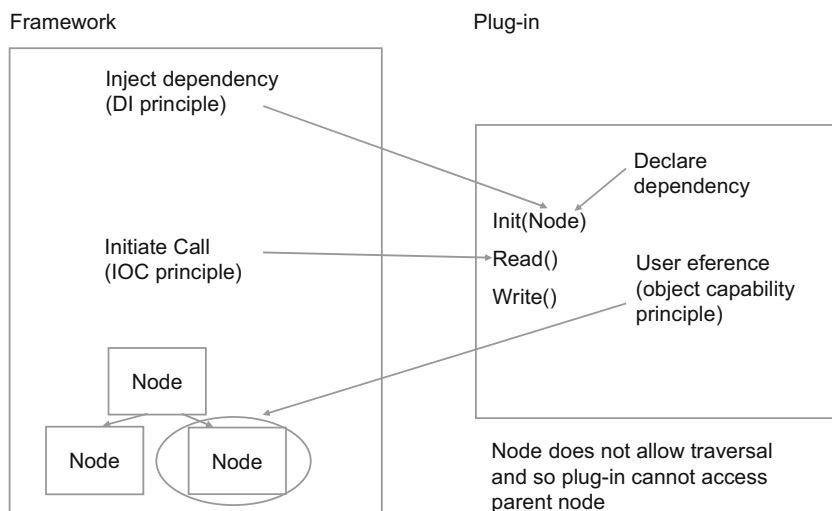


Abb. 4.12 Capabilities, Inversion-of-Control und Dependency-Injection-Prinzip

4.4 Kapselung durch virtuelle Maschinen

Zum „Schweizer Taschenmesser“ der IT-Security gehören neben SSL und Firewalls seit einiger Zeit auch Virtual Machines. Auf ihnen lässt man kritische Applikationen oder Server mit dem Anspruch laufen, dass sie dann „sicher“ oder „ungefährlich“ sind.

Wie immer sind Teile dieser Erwartungen richtig, andere leider falsch, und deshalb wollen wir hier Virtual Machines unter dem Gesichtspunkt des tatsächlichen Nutzens für die Sicherheit betrachten, und zwar für die Sicherheit aller Beteiligten: Systeme, Umgebung, Applikation selbst etc. Wo ergibt der Einsatz von VMs ein tatsächliches Plus an Sicherheit und welche Probleme sind davon unabhängig?

Wir gehen so vor, dass wir uns zunächst die Sicherheit des Umfelds einer Virtual Machine vor Augen führen. Anschließend schauen wir uns einige VM-Techniken an und überlegen anschließend, welche Sicherheitseffekte diese nach sich ziehen. Die Diskussion über die damit tatsächlich erreichte Sicherheit werden wir im Kapitel zu Application-Servern fortsetzen.

4.4.1 Sicherheitsumfeld

Abbildung 4.13 zeigt Schwachstellen herkömmlicher Betriebssysteme – sog. *Legacy-Systeme* wie MS-Windows und Linux – wie sie seit vielen Jahren im Einsatz sind und deren Architektur sich im Vergleich zur Hardware kaum weiterentwickelt hat.

Das Diagramm zeigt Angriffspunkte auf fast allen Ebenen. Klaus Mehnert schließt daraus in seiner Dissertation zum Thema Virtuelle Maschinen: „Monolithische Betriebssysteme (Legacy) sind schlicht nicht vertrauenswürdig“ [Mehn]. Das monolithische Betriebssystem als Trusted Computing Base ist schlicht zu groß, als dass es noch zuverlässig sein, geschweige denn validiert werden könnte. Allein die Angriffsmöglichkeiten durch versteckten Code in unzähligen Treibern können recht subtil sein und auch Open-Source-Betriebssysteme sind schon so attackiert worden. Wenn der unten abgebildete Code unentdeckt im Kernel landet, kann dieser durch einen unscheinbaren Treiberaufruf übernommen werden:

```
Driver.c:
    if (uid = 0){
        // do something harmless
    }
    //run as root now!
```

Andere, etwas diffizilere Attacken sind die Angriffe auf die Zufallszahlengeneratoren aktueller und älterer Betriebssysteme, wie sie Zalewski in [Zal] schildert.

Same runtime environment
for all applications

A common,
navigable
filesystem with
ambient authority

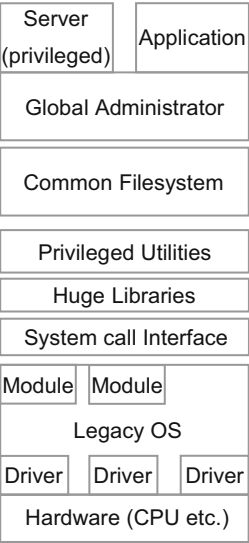
Tons of unsafe
but privileged
scripts and
utilities
(setUid)

>300 complex
system calls

Countless
dynamically
loadable
modules

>100.000
drivers for
windows

Network scans (IP seq.)



Huge TCB, 2 modes only

Cycle stealing
applications create
a problem for near-
realtime multimedia
applications

Dangerous accounts, single
audit and log features

Lots of unverified
system libraries
with memory leaks
etc.

Incomplete quota
administration

Attacks on random
number generation

Unsafe kernel
languages (memory)

Covered channels
(cache, bios, CPU)

Abb. 4.13 Schwachstellen herkömmlicher Betriebssysteme

Diese Angriffe können z. B. im rapiden Auslesen von /dev/random bestehen, so dass dem System die Zufallszahlen ausgehen. Kontrolliert dann der Angreifer noch die zur Verfügung stehende Entropie (Keyboard Input, Network Traffic), dann kann er Random Numbers erraten. Da diese für die Herstellung von Schlüsseln unbedingt nötig sind, kann der Angreifer so die Schlüsselgüte kompromittieren. Für unsere Zwecke hier sind solche Attacken zu speziell und für die überwiegenden Fälle von Sicherheitsproblemen in Software auch nicht relevant. Ihre Abwehr profitiert jedoch meist vom Einsatz virtueller Maschinen.

Sehen wir uns nun einige weitere Angriffsmöglichkeiten an und fragen uns nach den Konsequenzen.

Wenn z. B. eine Server-Applikation durch Buffer Overflows übernommen wird, dann erhält der Angreifer oft Kontrolle über ein privilegiertes Programm, je nachdem, unter welcher Identität der Server läuft. Dadurch ist es leicht, weitere Komponenten der Maschine anzugreifen, z. B. andere Applikationen. Andererseits – der Angriff muss nicht anderen Komponenten gelten, sondern kann nur dazu dienen, die Applikationsdaten zu korrumpieren bzw. einen Service zu erschleichen.

Der Missbrauch von Administrationspasswörtern lässt sich leichter kontrollieren, wenn verschiedene Maschinen verschiedene Passwörter besitzen.

Je weniger Principals auf einer Application-Server-Maschine tätig sind, umso besser, angesichts der Race Conditions in vielen Admin Scripts und der großen

Zahl von Utilities mit Superuser-Rechten. Aber das gilt grundsätzlich für alle Server Maschinen.

Performancekritische Applikationen mit leichten Realtime-Anforderungen profitieren auf jeden Fall von einer eigenen Maschine, jedoch muss dies keine VM sein. Erst wenn ein VM-Monitor mehrere VMs mit unterschiedlichen Performanceanforderungen gleichzeitig überwachen kann, macht eine VM Sinn.

Letztlich kann man sagen, dass alle Schwächen einer Komponente (Applikation oder OS), die Angriffe auf andere Komponenten ermöglichen, von einer VM profitieren, dass damit aber die Sicherheit der Komponenten intern nicht gesteigert werden kann. Eine SQL-Injection-Attacke auf eine Web-Applikation funktioniert innerhalb einer VM genauso gut. Schlechte kryptografische Schlüssel zur Absicherung von Admin-Menüs in Server-Applikationen werden ebenfalls durch den Einsatz von VMs nicht besser.

Es macht also Sinn, kritische Applikationen getrennt voneinander mit jeweils eigener Betriebssysteminstanz laufen zu lassen, sodass bei einem Angriff nicht alle Applikationen oder Server betroffen sind.

4.4.2 Paravirtualisierte VM-Architekturen

Als Beispiel einer „paravirtualisierten“ Lösung sei hier kurz das bei Mehnert (s. [Mehn]) beschriebene NIZZA-System vorgestellt. Es besteht aus einem Microkernel und paravirtualisierten „Linux-Gästen“, die für die Laufzeitumgebung des Microkernels speziell angepasst und mit den Services des Microkernels über inter-

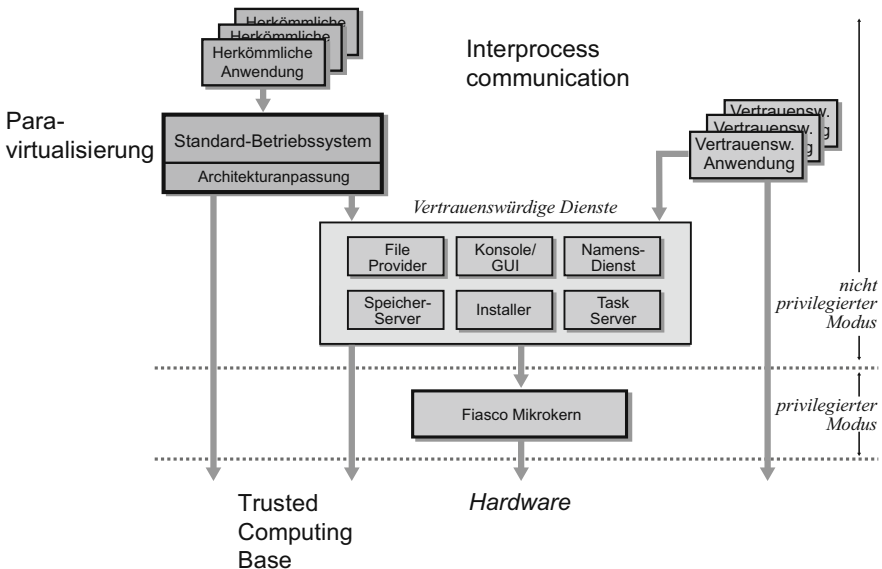


Abb. 4.14 Architektur einer paravirtualisierten Lösung nach Mehnert [Mehn]

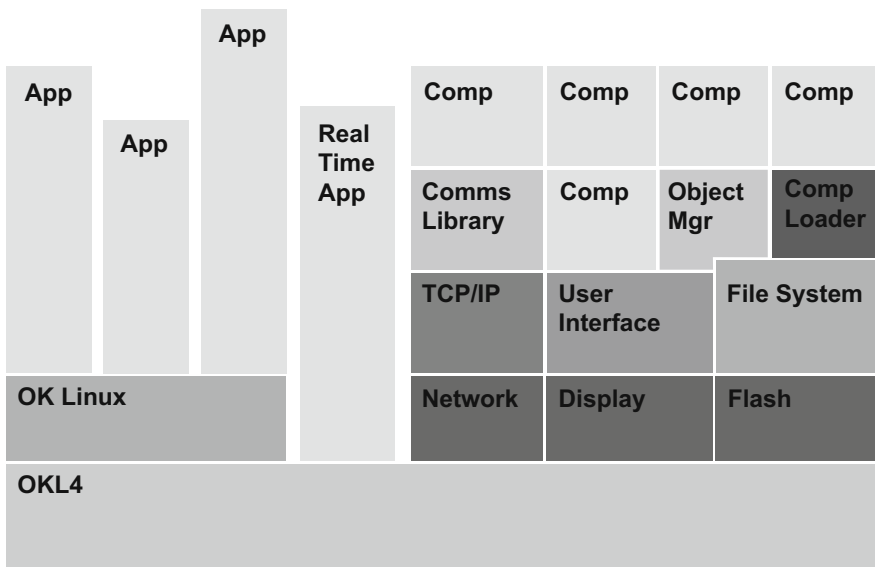
process communication (IPC) kommunizieren. Trusted Applications sowie in eingeschränktem Maß auch die Gäste dürfen sogar direkt Hardware ansprechen (s. Abb. 4.14).

Nach Mehnert ist die Paravirtualisierung unter anderem deshalb ein guter Ansatz, weil einige Architekturen wie etwa Intel 80×86 Schwächen im CPU-Design besitzen und z. B. ein Aufruf privilegierter Funktionen im nicht-privilegierten Modus keine Schutzverletzung auslöst. Das bedeutet, dass z. B. ein nicht-privilegierter Gast-Kernel, der eine Interruptsperr setzen möchte, schlicht ignoriert wird – und damit ungeschützte Regionen im Gast-Kernel entstehen, da der VM-Monitor von diesem Wunsch nichts erfährt (POPF Flag).

Interessant im NIZZA-Design ist die Möglichkeit, nebeneinander Realtime-Applikationen und normale Applikationen zu betreiben. Ein weiterer Vorteil paravirtualisierter, auf Microkernen beruhender Virtual Machines ist die deutlich kleinere Trusted Code Base, die dafür nötig ist.

4.4.3 Hybride Systeme im Embedded-Control-Bereich

In „Virtualization for Embedded Systems“ beschreibt Gernot Heiser [Heis07] mögliche Szenarien für den Einsatz von Virtualisierung. Da Isolation der Kernpunkt von Virtualisierung ist, stellt sich die Frage, wie dies für eine Vielzahl von Services und Applikationen auf einer Plattform erreicht werden kann. Die erste



From: G.Heiser, Virtualization of Embedded Systems

Abb. 4.15 Virtualisierung auf Basis des seL4 Microkernels

Idee, nämlich jeder Applikation ein eigenes Gast-OS zu geben, scheitert schnell an der Kostenfrage. Eher interessant scheinen Kombinationen von Hypervisor und Microkernel zu sein, die zusammen ein hybrides System, bestehend aus verschiedenen isolierten Komponenten, formen können. Basis der Isolation ist dabei natürlich immer noch das virtuelle Memory zwischen oder innerhalb von Gastbetriebssystemen und Komponenten.

Abbildung 4.15 aus [Heis07] zeigt ein hybrides System.

Wichtig ist, dass auf Basis einer hybriden Architektur eine Isolation auf kleinerer Granularität erreicht werden kann, als durch den bloßen Einsatz von Virtualisierung. Hier können sogar einzelne Systemkomponenten voneinander getrennt werden, während gleichzeitig die Kommunikationsfähigkeit sowie die Verwaltung des Systems als Ganzes (Power, Scheduling) gewährt bleibt.

4.4.4 Virtualisierung unter Einschluss der Verfügbarkeit: Mainframes

Die Darstellung der granularen Isolation von Komponenten, Netzwerk-Stacks etc. voreinander (Abb. 4.15) ähnelt bereits verdächtig Konzepten, wie sie im Bereich der Mainframes seit vielen Jahren im Einsatz sind (Abb. 4.16).

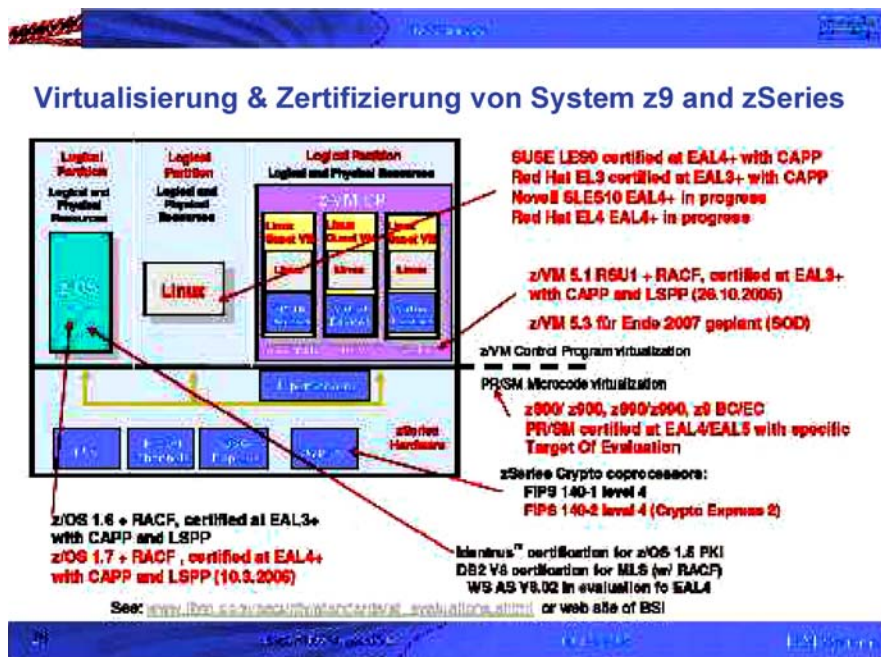


Abb. 4.16 Virtualisierungsoptionen der IBM Mainframes nach [Gnir]

Man beachte, dass hier zwei verschiedenen Formen von Virtualisierung eingesetzt werden. Auf Hardwareebene isolieren sog. Logical Partitions (LPARs) die Systeme und Komponenten voneinander. Zusätzlich kann eine Virtualisierung auf Softwarebasis oberhalb der LPAR-Schicht laufen und erst darauf z. B. ein Gastbetriebssystem oder eine Komponente.

Die LPAR-Schicht erlaubt etwas, mit dem kleinere Systeme nach wie vor kämpfen: Die Isolation der Systeme in Bezug auf die Verfügbarkeit. Während Virtualisierungslösungen für kleine Systeme oft noch mit Dingen wie der fehlenden Unterstützung für virtualisierte DMA-Zugriffe etc. kämpfen, gelingt es den IBM Mainframes, die Anwendungen komplett so zu trennen, dass jeder Anwendung oder Komponente ein genau spezifizierter Anteil der CPU zugeordnet wird. Das heißt, Performanceversprechen einer Applikation können nicht durch Fehlverhalten anderer, gleichzeitig laufender Applikationen gebrochen werden.

Im Endeffekt bedeutet dies, dass das Virtualisierungskonzept der Mainframes nicht nur die logische Isolierung im Sinne von Korrektheit umfasst, sondern auch die physische Isolierung im Sinne von Konsistenz oder Verfügbarkeit, spricht den Liveness-Aspekt.

4.4.5 Grenzen der Sicherheit durch Virtualisierung

Das folgende Argument zeigt die Grenzen der Virtualisierung in Bezug auf Security deutlich auf: Die Größe der Trusted Computing Base wächst mit dem Grad der Virtualisierung!

“Virtualization does not support minimizing the TCB. Compared to running a service on top of a native OS, running it in a virtual machine requires a hypervisor and a guest OS, both part of the TCB. Compared to a native OS, virtualization increases the TCB.” (aus [Heis07], S.16)

Dass dies keine nur theoretische Gefahr darstellt, zeigte sich, als die momentan herrschende Überzeugung, dass viele Gefahren der Plattformsicherheit durch den Einsatz von Virtuellen Maschinen vermieden werden können, durch empirische Studien erschüttert wurde. So stellte Tavis Ormandy kürzlich fest, dass praktisch alle VM auf bekannten Plattformen durch verschiedene Attacks übernommen werden können [Orm].

Die VMs können daher die verlangte „Root Security“ – d. h. kein Code innerhalb des Gastbetriebssystems kann die VM übernehmen – nicht sicherstellen. Es muss ja nicht gleich die komplette Übernahme der Maschine sein. Bereits scheinbar harmlose Schwächen der Virtualisierung, wie etwa die, dass das Gastsystem bzw. dessen Applikationen die Tatsache der Virtualisierung erkennen können, können in einem entsprechenden Kontext bereits ausgenutzt werden. Viele Firmen führen z. B. Softwaretests auf Compute-Servern unter Verwendung von VMs durch. Attack-Code, der erkennen kann, wann er unter einer VM läuft, kann sich in diesem Fall entsprechend harmlos verhalten und wird daher bei den Tests nicht erkannt.

Das Vorgehen beim Test der VMs ist sehr interessant. Der Autor verwendet zwei bekannte „Fuzzer“, nämlich CRASHME und IOFUZZ, um Random-Input sowohl für das System Call API wie auch die IO-Schnittstellen zu erzeugen. Nach Ormandy sind diese beiden Eingänge als die Hauptschwächen der VMs erkannt worden. Für die genauen Ergebnisse pro VM (es wurden sowohl open source als auch kommerzielle VMs getestet) sei auf das Paper von Ormandy verwiesen. Hier seien jedoch noch seine abschließenden Tipps für den Einsatz von VMs zur Isolation von Gästen und Applikationen kurz erwähnt, da sie generelle Bedeutung für die Plattformensicherheit besitzen:

- VMs (die gewöhnlich unter privilegierten Accounts arbeiten) sollen wie andere kompromittierbare Dienste behandelt werden, d. h. am Besten unter Verwendung von Isolationstechniken wie chroot, systrace, ACLs sowie POLA-Regeln.
- Die virtuellen Maschinen wie die Gastbetriebssysteme sollten so schmal wie möglich konfiguriert sein. Ungenutzte Hardwaretreiber oder Services sind auch beim Gast zu entfernen.
- Die Integrität der Gastsysteme ist genauso wie die der Trägermaschinen durch regelmäßige Updates, Privilegieneinschränkungen der Gastapplikationen etc. zu sichern.
- Systeme, die mehrere unterschiedliche Privilegienzonen ermöglichen, sind vorteilhaft (XEN). Sicherheitslevel von Plattformen (z. B. Berkeley Security Levels) sind zu nutzen [Orm].

4.5 Softwarebasierte Isolation am Beispiel Singularity

Oben haben wir zwei zentrale Mechanismen der Isolation durch Kapselung kennen gelernt: Unterschiedliche Modi, die unterschiedlichen Privilegien entsprechen, und getrennte Adressräume. Zugang zu beiden ist nur durch genau definierte und überwachte Interfaces gestattet. In einem Fall ist dies das System-Call-Interface, im anderen die Interprozesskommunikation. Beide Mechanismen sind grobgranular und außerdem beim Wechsel von Modus oder Adressraum sehr teuer (mehrere Tausend Zyklen).

Es hat jedoch schon früh in der Geschichte der Betriebssysteme Verfahren gegeben, die deutlich granularer waren und keine privilegierten Modi gekannt haben. Es handelt sich dabei um Deskriptor- bzw. Capability-Systeme. Diese Systeme zeigten sehr schnell auch eine Affinität zur objektorientierten Programmierung, und so entstand die Idee, Objekte zur Isolation zu verwenden.

Objekte ersetzen den Gedanken der Zugriffsprüfung durch die Frage der Erreichbarkeit: Kann ein Stück Code ein Objekt erreichen, das eine bestimmte Fähigkeit besitzt? Dann kann der Code diese Fähigkeit verwenden, ansonsten nicht. Grundvoraussetzung dabei ist, dass der Code die Kapselung eines Objektes nicht brechen kann. Dies kann durch Hardware gesichert werden: Sogenannte *C-Lists* – Listen von Capabilities – können objektprivat sein und sind damit für den Zugriff von außen verborgen. Oder aber die Kontrolle des Zugriffs auf kritische Objekte

wird dem Compiler überlassen: Er darf keinen Code akzeptieren bzw. generieren, der die Kapselung durchbricht.

Das Betriebssystem *Singularity* (s. [Hunt]) von Microsoft Research greift den Gedanken der moduslosen Sicherheit wieder auf und versucht, sie durch Compilertechniken zu realisieren. Voraussetzung dafür ist eine sichere Sprache – in diesem Fall Sing#, die keine Pointerkonzepte wie in C oder C++ kennt und damit Sicherheit auf der Ebene der Mikroarchitektur garantieren kann.

Diese Eigenschaft allein reicht natürlich nicht für ein Betriebssystem (dann wären VMs in Java und .NET bereits Betriebssysteme) aus, und so enthält die Makroarchitektur von Singularity einige Besonderheiten (s. Abb. 4.17):

- Die Kapselung beruht auf leichtgewichtigen Prozessen.
- Kernel, Applikationen, Treiber und Systemserver sind allesamt Prozesse.
- Ein Prozess ist ein geschlossener Object-Space, der keine Referenzen in andere Prozesse hinein besitzt.
- IPC ist durch ein Kanalkonzept realisiert, das sich durch streng typisierte Kommunikation, definiert durch State-Machines, auszeichnet.
- Objekte können an andere Prozesse übertragen werden, wechseln dann jedoch den Besitzer, sodass keine prozessübergreifenden Synchronisationsprobleme oder eine zentrale Garbage Collection nötig sind.
- Ein kleiner Teil Code ist unverified und trusted, der große Rest von Mikrokernel und Applikationen etc. ist verifiably trusted, d. h. er wurde vom sicheren Compiler erzeugt.

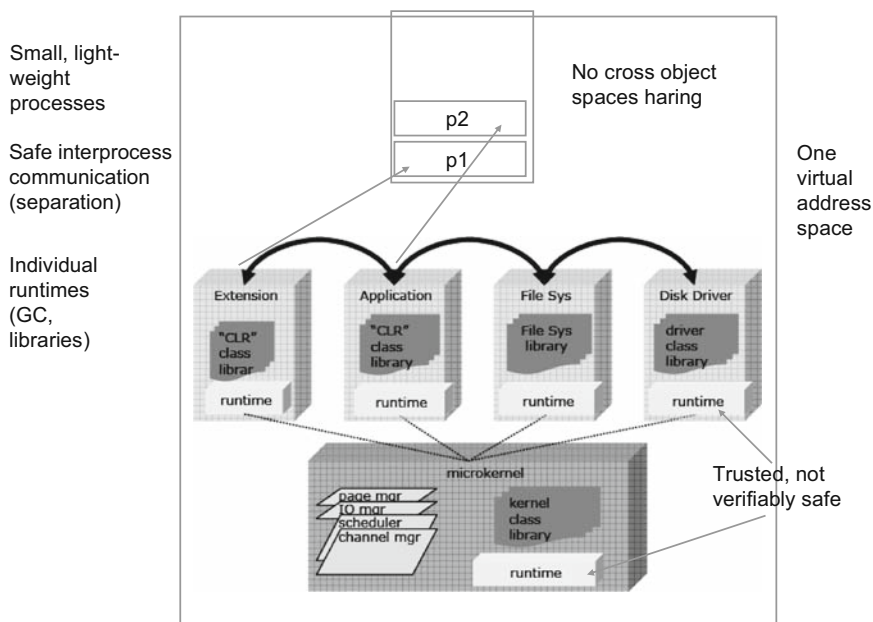


Abb. 4.17 Die Architektur von Singularity

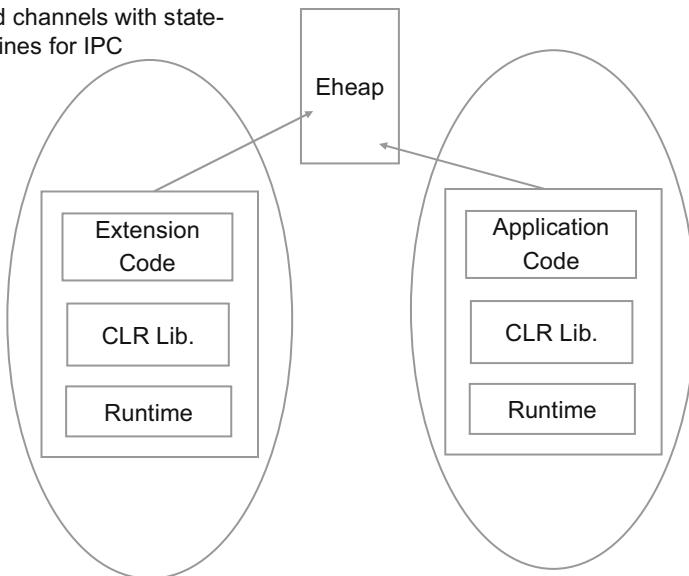
- Es sind keine dynamischen Erweiterungen von Prozessen (Code laden) erlaubt. Jede Erweiterung muss ein eigener Prozess sein.
- Jede Softwarekomponente ist durch Meta-Daten umfangreich beschrieben und wird ohne eigenen Installer installiert.

Trusted Code läuft hier nicht in irgendeinem privilegierten Modus. Entscheidend ist der Zugriff auf Objekte, die Privilegien kapseln. Damit kommt den Referenzen auf Objekte eine entscheidende Bedeutung zu.

Die Runtime-Komponenten gelten als vertrauenswürdig, der Rest wird im Wesentlichen verifiziert. Die Runtimes (inklusive Garbage-Kollektoren) können ausgetauscht und damit an spezielle Anforderungen der Applikationen oder Treiber angepasst werden. Der einzige große virtuelle Adressraum sorgt für effiziente Interprozesskommunikation. Selbst wichtige Kernelfunktionen sind nur einen Methodenaufruf plus Kanalaroutinen entfernt.

Für die Übertragung von Objekten oder den Transport von Daten dient der *Exchange Heap* (s. Abb. 4.18). Dort kann ein Prozess, z. B. ein Ethernet Driver, Daten platzieren und dann ein Objekt zum Zugriff darauf über einen Kanal zu einem Applikationsprozess transportieren. Entscheidend ist wiederum, dass die Eigentümerschaft an den Daten damit an den anderen Prozess übergeht, was enorme Verwaltungs- und Verantwortungsprobleme vermeidet (Synchronisation von gleichzeitigen Zugriffen, Garbage Collection).

Typed channels with state-machines for IPC



No common, shared variables
between both object spaces

Abb. 4.18 Transport von Objekten in Singularity

Die Verwaltung und das Tracking von Datenstrukturen bzw. Objekten spielt eine große Rolle in Singularity, und zwar nicht nur zwischen Prozessen. Für die Verwaltung von Referenzen stellt Sing# – das erweiterte C# – den Typ `TRef`

```
class TRef<T> where T:ITracked {
    public TRef([Claims] T i_obj);
    public T Acquire();
    public void Release([Claims] T newObj);
}
```

When creating a `TRef<T>`, the constructor requires an object of type `T` as an argument. The caller must have ownership of the object at the construction site. After the construction, ownership has been passed to the newly allocated `TRef`. The `Acquire` method is used to obtain the contents of a `TRef`. If the `TRef` is full, it returns its contents and transfers ownership to the caller of `Acquire`. Afterwards, the `TRef` is said to be empty. `Release` transfers ownership of a `T` object from the caller to the `TRef`. Afterwards, the `TRef` is full. `TRefs` are thread-safe and `Acquire` operations block until the `TRef` is full.

Abb. 4.19 Methoden des Typs `TRef` in Sing#

```
public contract NamespaceContract : ServiceContract {
    in message Bind(char[] in path, ServiceContract.Exp:Start exp);
    out message AckBind();
    out message NakBind(ServiceContract.Exp:Start exp);
    in message Notify(char[] in pathSpec, NotifyContract.Imp:Start imp);
    out message AckNotify();
    out message NakNotify(NotifyContract.Imp:Start imp);
    in message Find(char[] in pathSpec);
    out message AckFind(FindResponse[] in results);
    out message NakFind();
    out message Success();
    override state Start: one {
        Success! -> Ready;
    }
    state Ready: one{
        Bind? -> ( AckBind! or NakBind! ) -> Ready;
        Find? -> ( AckFind! or NakFind! ) -> Ready;
        Notify? -> ( AckNotify! or NakNotify! ) -> Ready;
    }
}
```

Abb. 4.20 NameSpace-Kontrakt in Singularity

bereit, der eine gezielte Übertragung des Eigentums an einer Referenz vornehmen kann (Abb. 4.19).

In Zeiten ständiger Sicherheitslöcher, die durch Input-Validierungsschwächen verursacht werden, ist es schön, einen relativ sicheren Kommunikationsmechanismus zu sehen. Kanäle in Singularity dürfen nur getypte Parameter verwenden und der Ablauf der Kommunikation muss durch State-Machines (einen sogenannten *Kontrakt*, s. Abb. 4.20) festgelegt werden.

Als eingebettete Klassen eines Kanals werden jeweils ein *Exporter* bzw. *Importer* definiert, die die entsprechenden Messages des Kontrakts zugeordnet bekommen. Will ein Prozess einen Kanal eröffnen, dann erzeugt er Instanzen von Exporter und Importer und schickt die Exporter-Instanz über einen existierenden Kanal an den Empfänger. Danach können beide miteinander kommunizieren.

Interessant ist die Frage, wie das System mit Erweiterungen umgeht. Wir haben oben gelernt, dass es oft die Erweiterungen (Plug-Ins, Treiber etc.) sind, die die Sicherheit von Systemen untergraben, und wir werden weitere Beispiele dafür kennen lernen. Singularity erlaubt keine dynamischen Erweiterungen zur Laufzeit (kennt daher auch keine Runtime Reflection) und zwingt Erweiterungen dazu, als separater Prozess zu laufen. Dieses Konzept könnte auf traditionellen Plattformen aufgrund der für Context Switch und IPC nötigen Zeiten zu großen Performanceproblemen führen. Hier werden – aufgrund des gemeinsamen Adressraums – IPC-Kommandos in wenigen hundert Zyklen abgearbeitet.

Ein System, das nicht auf Privilege Modi und grober Trennung beruht, muss auch die Installation von Software anders behandeln: Software muss mitteilen, welche Rechte (sprich Objekte) sie benötigt, um korrekt zu funktionieren, und nur diese werden zugeteilt. Die Tools von Singularity erzeugen sogenannte *Manifests*, d. h. Meta-Daten zur jeweiligen Software (s. Abb. 4.21), die in den Code eingebettet sind (ein Annotationskonzept wie neuerdings auch bei Java). Ein Singularity Installer liest diese Annotationen und erzeugt die richtigen Objektverbindungen.

Man sieht, dass in dem Manifest auch Versionsnummern und Dependencies spezifiziert werden. Singularity prüft bei der Installation diese Angaben und verweigert das Laden, wenn es Inkonsistenzen gibt, bzw. die verlangten Pakete nicht vorhanden sind. Dieses Konzept ist übrigens sehr verwandt mit OSGI – der Open Services Gateway Initiative –, die wir weiter unten genauer kennen lernen werden. Dort ist der Anspruch allerdings nicht ganz so hoch wie bei Singularity: OSGI erlaubt es, in Java geschriebene Services dynamisch zu laden und zu entladen und prüft dabei Dependencies. Zentraler Mechanismus dort sind nicht-hierarchische ClassLoader. Während Singularity also sog. *Object Spaces* als Prozesse definiert, könnte man bei OSGI eher von *Type Spaces* reden. Dort ist die Isolation auch nicht so stark wie bei Singularity.

Sicherheitstechnisch entscheidend ist jedoch, dass hier vom Prinzip der „setup“-Methode abgerückt wird. An keiner Stelle wird zur Installation eines Applikations-Codes dieser Applikation aufgerufen, um dann – natürlich mit Administrationsrechten – die Installation durchzuführen. Systeme, die dies zulassen, brauchen sich nicht wundern, wenn die Artefakte von Applikationen wild innerhalb des Systems verteilt werden, eine geregelte De-Installation nicht möglich ist

```

<manifest>
<application identity="S3Trio64" />
<assemblies>
<assembly filename="S3Trio64.exe" />
<assembly filename="Namespace.Contracts.dll" version="1.0.0.2299" />
<assembly filename="Io.Contracts.dll" version="1.0.0.2299" />
....
<assembly filename="ILHelpers.dll" version="1.0.0.2299" />
<assembly filename="Singularity.V1.ill" version="1.0.0.2299" />
</assemblies>
<driverCategory>
<device signature="/pci/03/00/5333/8811" />
<ioMemoryRange baseAddress="0xb8000" rangeLength="0x8000" fixed="True" />
<ioPortRange baseAddress="0x4ae8" rangeLength="0x2" fixed="True" />
....
<extension startStateId="3" contractName="Microsoft.Singularity-.Extending.ExtensionContract"
endpointEnd="Exp,a ssembly="Namespace.Contracts" />
<serviceProviders tartStateId="3" contractName="Microsoft-.Singularity.Io.VideoDeviceContract"
endpointEnd="Exp"
assembly="Io.Contracts" />
</driverCategory> </manifest>

```

Abb. 4.21 Manifest in Singularity

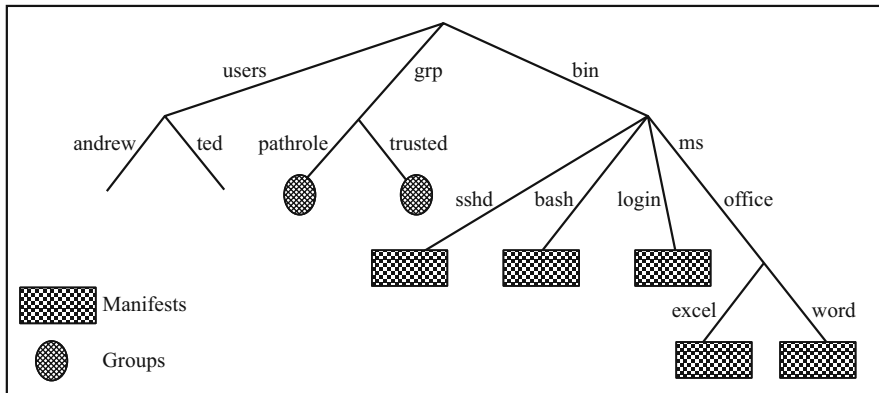
und zudem völlig unklar ist, welche Modifikationen während der Installation durch die verfügbaren Administrationsrechte am System selbst erfolgt sind.

Isolation durch Objekte allein reicht nicht für vollständige Sicherheit auf Betriebssystemebene. Manchmal müssen Zugriffe auch durch Prüfung des Principals autorisiert werden. Hier entscheidet sich Singularity für eine Erweiterung des Konzeptes von Java und .NET, das auf dem Verfolgen der Aufrufgeschichte sowie den Rechten des jeweiligen Principals beruht (siehe Kap. 5, Java Sprach- und Plattformsicherheit). Singularity fügt dem noch die Möglichkeit hinzu, die Güte der jeweiligen Authentisierung zu berücksichtigen. Der betreffende Mechanismus nennt sich *Compound Principal* und stellt eine Erweiterung der Access Control Lists dar (s. [ABW]).

Sowohl Nutzer als auch Applikationen sind hier Principals, und Ressourcen wie Files können in ihren Access Control Lists Pattern angeben, die dann mit den Compound Principals (siehe Abb. 4.22) verglichen werden. So könnte ein File verlangen, dass es von einem User in der Rolle *X* über eine beliebige Reihe von Applikationsaufrufen verwendet werden kann, wenn die letzte Applikation *Y* ist. Das Programm, das den Login durchgeführt hat, dient dabei zur Festlegung der Güte der Authentisierung.

```
/bin/sshd @ /users/ted (+ {/grp/pathrole})* + /bin/ms/office/word
```

Abb. 4.22 Compound Principal mit Authentication History



An example naming tree

Abb. 4.23 Globaler Namespace mit Ressourcen

Ob sich dieses Konzept aber wirklich durchsetzt, scheint fraglich: Bindet es doch Front-End-Mechanismen wie Authentisierung eng an Backend-Mechanismen wie ACLs, und der daraus resultierende Verwaltungsaufwand bei Änderungen ist hoch. Aber das ist nicht das Schlimmste: Wenn unterschiedliche Gütegrade von Authentisierungen protokolliert werden, dann müssen ab diesem Zeitpunkt alle Applikationen und Ressourcen dem Rechnung tragen. Das kann leicht zu groben Sicherheitslöchern führen, wenn die Berücksichtigung nicht durch spezielle Infrastrukturschichten erzwungen werden kann.

Ein Grund dafür, warum der zusätzliche Access-Control-Mechanismus nötig ist, ist die Tatsache, dass in Singularity alle Services und Systementitäten innerhalb eines hierarchischen Namespaces abgebildet werden, der von Clients durchsucht werden kann. Damit könnten Prozesse an illegale Ressourcen gelangen: Das File-System ist ein Teilbaum innerhalb des globalen Namespaces (s. Abb. 4.23, aus [ABW]).

Sollte sich das Konzept der „hardwarelosen“ Isolation durchsetzen, dann könnte das auch auf das Design von Prozessoren Einfluss haben. In Kap. 8 werden wir anhand von OSGI untersuchen, wie das dynamische Laden von Services unterschiedlicher Hersteller sicher gestaltet werden kann.

4.6 Das Least Authority (POLA) Prinzip beim Bau von Systemen

Die folgenden Ausführungen stützen sich stark auf Untersuchungen im Rahmen des Asbestos Betriebssystems bzw. der Entwicklung sicherer Web-Services im

OKWS-Projekt (s. [Krohn]; [KEFK]). Wir werden untersuchen, wie heutige Betriebssysteme Applikationen bei der Verwirklichung des POLA-Prinzips unterstützen oder auch behindern und wie die Applikationsarchitektur selbst POLA bewerkstelligen kann. Nicht die Sicherheit des Betriebssystems selbst, sondern die Sicherheit der darauf gebauten Applikationen steht dabei im Vordergrund.

4.6.1 *Additive und subtraktive Rechtezuweisung*

Beginnen wir mit Beispielen für die Reduktion von Privilegien, um POLA zu erreichen. Als erstes sind hier das „chroot“-Verfahren oder die Einrichtung sogenannter „Jails“ zu nennen. In beiden Fällen werden Prozesse in einem bestimmten Bereich des File-Systems „eingesperrt“ – im Falle von Jails sogar mit Einschränkungen in Bezug auf die Netzwerk- oder Interprozesskommunikation. Verwendet wird hier ein subtraktives Modell, bei dem vorhandene Privilegien bewusst eingeschränkt werden. Dies geschieht durch die Verwendung von System Calls, die die Eigentümerschaft, die Identität und die Laufzeitumgebung ändern.

Dieses Verfahren wird gerne zur Absicherung von Servern eingesetzt, ist jedoch relativ schwerfällig, komplex und hat einen hohen administrativen Aufwand.

Das „Jail“ oder die „chroot“-Umgebung stellt letztlich eine Isolation der dort eingeschlossenen Prozesse dar. Auf der anderen Seite zeichnen sich additive Methoden durch eine Anreicherung von Privilegien nach Bedarf aus (s. Abb. 4.24). Das additive Verfahren ist insgesamt sicherer, da es das Prinzip des „default is deny“ bzw. „least Privilege“ besser realisiert. Hier läuft ein Prozess mit zunächst wenigen Privilegien und erhält Zugriff auf weitere Ressourcen durch Capabilities. Ein Unix Filedescriptor ist ein klassisches Beispiel für eine solche Capability.

Heutige Betriebssysteme, aber auch Sprachen wie Java, verwenden an verschiedenen Stellen Capabilities – dies jedoch nicht konsistent und durchgängig schlüssig. Das zentrale Prinzip der Capabilities – kein Zugriffsrecht, ohne eine Capability dynamisch zu empfangen – wird an vielen Stellen durchbrochen, z. B. durch fest codierte Rechte wie `getCurrentWorkingDirectory()` oder `getFileSystemRoot()`: Funktionen, die also einem Prozess einfach Ressourcenzugriffe erlauben, ohne explizite Erteilung einer Capability (s. [KEFK]).

Der Vorteil von Capabilities zeigt sich an folgendem Beispiel: Verschiedene Prozesse sollen Ereignisse in einem Logfile protokollieren. Sie können dazu entweder direkten Zugriff auf das File erhalten – mit der Konsequenz, dass ein böser Prozess das Logfile löschen kann – oder sie können ihre Ereignisse per Interprozesskommunikation an einen Log-Prozess senden, der sie seinerseits protokolliert. Der Log-Prozess stellt also eine Kapselung des Files dar und garantiert die Sicherheit.

Alternativ könnten die Prozesse eine Capability auf das File erhalten: Diese Capability würde ein *Append-Only*-Recht darstellen (vorstellbar als Filedeskriptor, bei dem bestimmte Funktionen abgeschaltet wurden). Dann könnten diese ebenfalls Prozesse protokollieren, ohne jedoch das File löschen zu können. Ein separa-

Privilege Restriction and Separation with Rights Reduction vs Capabilities

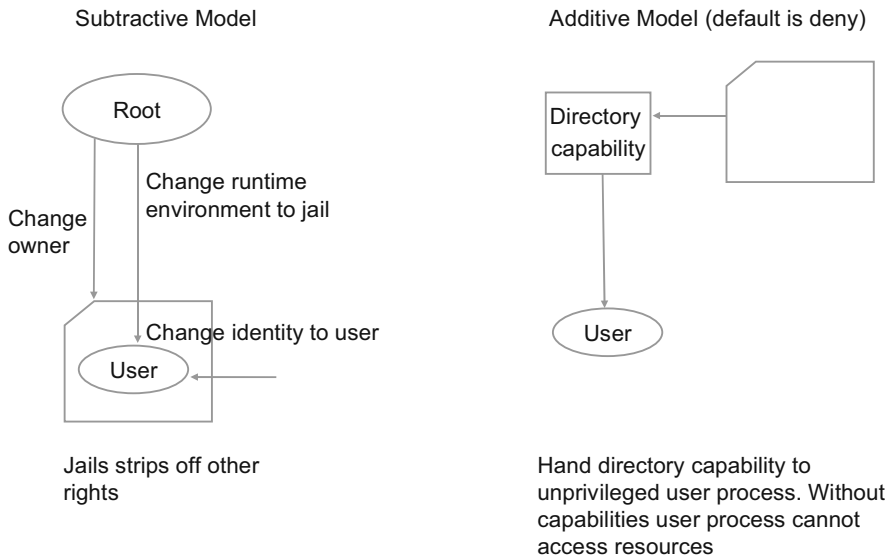


Abb. 4.24 Subtraktiver und additiver und Ansatz zur Rechtevergabe

ter Prozess zum Schutz des Files wäre unnötig. Diese Technik könnte auch für die Absicherung von Servern, die wir in Kap. 4.7 behandeln, zum Einsatz kommen.

4.6.2 Labeling und Tainting

Capabilities erlauben bereits sehr granulare Zugriffskontrollen. Manche Applikationen verwalten jedoch innerhalb eines Servers mehrere Nutzer, und es stellt sich die Frage, wie die Daten der Nutzer voreinander geschützt werden können. Sogenannte *Labeling*-Systeme könnten hier in Zukunft eine große Rolle spielen: Daten erhalten durch einen privilegierten Prozess bestimmte Tags (*Labels*) zugeteilt, und in der Folge überwacht das System die Verwendung der gekennzeichneten Daten. Das Betriebssystem Asbestos verwendet dieses Verfahren unter dem Namen „Decentralized Mandatory Access Control“, aber auch im Datenbankbereich wird es unter den Namen „Multi-Level-Security“ oder LBAC (Label Based Access Control) langsam populär.

Labeling-Verfahren scheinen eine gewisse Ähnlichkeit mit den sog. *Tainted Variables* zu besitzen: Mit einer Variablen werden Meta-Daten mitgeführt, die den Einsatz der Variablen kontrollieren. Die Programmiersprache PERL z. B. erlaubt es, auf diese Weise Variablen zu kennzeichnen, die potenziell gefährlichen Inhalt tragen – z. B. Daten, die vom Nutzer eingegeben wurden. Das System kann dann

verhindern, dass diese Variablen direkt in Datenbanken eingegeben werden (Verhindern von SQL-Injection-Attacken) oder dem User ohne Prüfung zurückgespielt werden (Verhindern von Cross-Site-Scripting-Attacken). Interessanterweise sollte es eine Version von JavaScript in Netscape 4.x Browsern geben, die auf dem Einsatz von Tainted Variables beruhte. Das Projekt wurde jedoch gestoppt und die Entwickler entschieden sich für ein Modell analog dem Java 2 Security Modell, basierend auf Permissions, die zur Laufzeit geprüft werden (siehe Kap. 5, Java Sprach- und Plattformsicherheit).

Möglicherweise wurde das Prinzip des „Tainting“ oder „Labeling“ zu früh verworfen. Wir werden es bei der Diskussion der Sicherheitsarchitektur von Browsern nochmals aufnehmen und einige mögliche Anwendungen bei bekannten Browserschwächen durchspielen. Es hat den entscheidenden Vorteil, dass die Sicherheit von Daten konstant garantiert werden kann, ohne dass man auf sporadische und zufällige Aufrufe von Prüfungen durch Entwickler angewiesen wäre – eine häufige Fehlerursache, denn leicht wird eine wichtige Prüfung einfach vergessen. Aber auch die Sicherheit von Funktionen kann dadurch garantiert werden. Zum Beispiel können unerwartete Call-Pfade schnell den Aufruf einer gefährlichen Basisfunktion aus unsicherem Code heraus ermöglichen. Nicht so jedoch, wenn auch die jeweiligen Funktionen einen „Taint“ besitzen und ein kleiner Mechanismus darüber wacht, dass getaintete und ungetaintete Funktionen nicht in einem Aufrufpfad vorkommen dürfen.

Das Prinzip des Labeling stellt neben Zugriffskontrollen über Permissions (ACLs), Capabilities und Isolation durch Namespaces die vierte große Methode der Zugriffskontrolle dar. Interessanterweise findet das Prinzip im neuen Vista-Betriebssystem von Microsoft Verwendung. Dieses wird in Abschn. 4.6.3 kurz vorgestellt und diskutiert.

4.6.3 Authority Reduction in Vista

Zu Beginn dieses Kapitels wurde gefragt, ob der PC zu Hause sicherheitstechnisch aufgegeben wurde, d. h., ob eine sichere Plattform nur durch die Einbindung in zentrale Administration und deren Updates bzw. Richtliniendurchsetzung zu erreichen ist.

Im neuen Windows Vista sind jedoch zwei Mechanismen enthalten, die diese Aussage in Frage stellen. Streng genommen ist es nur ein Mechanismus, denn die sog. User Access Control (UAC) baut direkt auf Integrity Levels (IL) auf. Es geht hierbei darum, zu verhindern, dass User permanent mit dem Administrator Account arbeiten – wie dies bisher meist der Fall war. Versuche, normale User unter XP oder Windows 2000 auf Nicht-Admin-Accounts zu setzen, gingen mangels Usability völlig daneben und führten zu unbrauchbaren Systemen. Typische Fälle waren User, die zwar verstanden, dass sie für bestimmte Aktionen Administrator werden mussten, dann jedoch vergaßen, zurückzugehen zum ursprünglichen, weniger mächtigen Account und stattdessen als Administrator weiterarbeiteten. Na-

türlich konnten sie dann am nächsten Tag – wieder eingeloggt mit ihrem normalen Account – bestimmte Daten nicht mehr lesen oder finden, da sie unter Admin-Privilegien angelegt wurden. Der Wechsel der Identität bringt zwar mehr Privilegien, aber auch eine neue Umgebung.

Vista teilt nun die installierten Daten und Programme in verschiedene Integritätslevel (IL) ein. Default-mäßig gibt es vier davon: Low, medium, high, system. Sie werden wie ACL-Einträge pro Ressource verwaltet. Hier handelt es sich also um ein Label-based-Access-Control-System, wie es in Form der sog. *-Properties schon oft beschrieben wurde. Im Bereich strenggeheimer und wichtiger Daten werden solche Labels verwendet, um sog. Write-Down bzw. Read-Up zu verhindern: Daten einer hohen Sicherheitsstufe sollen nicht Applikationen oder Geräten mit geringerer Einstufung übergeben werden können (Write-Down) und sie sollen auch nicht durch solche gelesen werden können (Read-Up). Das Schreiben von Daten an Ressourcen höherer Einstufung ist hingegen problemlos möglich – auch wenn der Schreiber sie anschließend nicht mehr lesen kann, falls er eine niedrigere Einstufung besitzt.

Vista dreht nun diese Logik um und implementiert No-Write-Up, aber Read-Up, also genau umgekehrte Regeln im Vergleich mit dem geschilderten Ansatz der *-Properties. Joanna Rutko erklärt in [Rut] diese Policy durch das Interesse des Herstellers, seine High-Level-Systemsoftware vor Angriffen durch Malware, die auf einem niedrigeren Level läuft, zu schützen (s. Abb. 4.25).

Das hierarchisch gegliederte Labeling-System als Zugriffskontrolle hat fundamentale Probleme: Es schützt nur die oberen Teile vor Angriffen von unten, denn

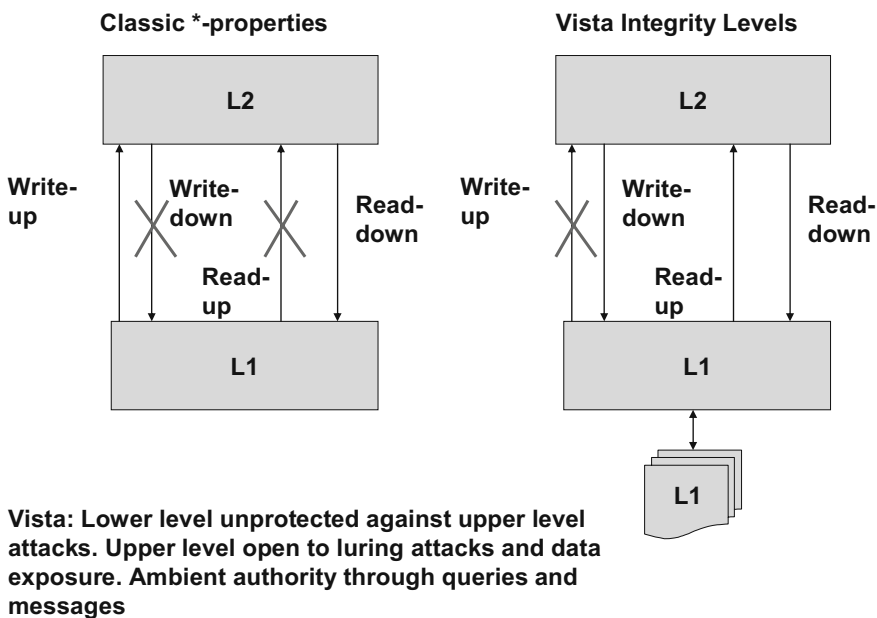


Abb. 4.25 *-Properties vs. Vistas labelbasierten Ansatz

es schränkt die Autorität der hohen Levels nicht ein. Es erlaubt, dass Prozesse niedrigeren Levels beliebige Daten von Prozessen höheren Levels auslesen können und ist durch die vier Levels auch sehr grobgranular. Wenn ein Prozess eines niedrigen Levels – z. B. ein Browser – kompromittiert wurde, dann kann er alle Daten des Users sowie der Maschine lesen und verschicken.

Interessant sind einige damit verbundene Implementationsprobleme: Das Windows-Messaging-System ist ja durch seine Shatter-Attacken berühmt geworden. Einige dieser Nachrichten scheinen jetzt unterdrückt zu werden, wenn ein Prozess niedriger Einstufung einem höheren Prozess eine solche Nachricht schickt. Andererseits kommen immer noch Nachrichten an eine geöffnete hochprivilegierte Shell durch, die zum Ausführen beliebiger Kommandos verwendet werden kann. Möglicherweise sind dafür Hooks, die für Accessibility verwendet werden (CBT-Hooks) verantwortlich, da sie nicht abgeschaltet werden können, ohne die Accessibility-Hilfen zu verlieren. Niedrige Prozesse können sogar durch Queries erfahren, wann eine privilegierte Shell geöffnet wird.

Kindprozesse scheinen den Sicherheitslevel des Vaterprozesses zu erben und können dadurch von ihm kontrolliert werden. So erhält ein Thunderbird Mail-Client Zugriff auf die Keys des PGP-Prozesses für das Enigma-PGP-Plug-In.

Diese Beispiele zeigen, wie schwierig es ist, auf der Basis von üppiger Ambient Authority Rechte einzugrenzen und wie schnell ein Level- oder Modus-System sich als unpraktisch erweisen kann, bzw. doch zu viele Rechte vergibt.

Die User Account Control (UAC) verwendet nun ebenfalls die definierten Integrity Levels, indem jeder Prozess entsprechend seines Accounts mit einem bestimmten Level gestartet wird. Dabei gibt es einige Ungereimtheiten (defaultmäßig wird mit dem „medium“-Level gestartet, Installationsprogramme starten immer mit Administratorrechten nach Abfrage des Admin-Passwortes). Benutzer können nun je nach Arbeitszweck verschieden privilegierte Accounts anlegen. Sie entsprechen eigentlich Rollen, denn das System sollte dabei berücksichtigen, dass es sich um ein und dieselbe Person handelt und z. B. nicht immer neue iTunes-Bibliotheken anlegen. Die Verwendung von Identitäten zur Reduktion von Autorität ist nur ein extrem grobgranulares System, das semantische Probleme für den User mit sich bringt.

In diesem Zusammenhang sind noch zwei fundamentale Fehler der Implementation zu erwähnen: Beim Wechsel der Accounts zum Zwecke der Privilegierungserhöhung ist es unsinnig, das Passwort des höheren Accounts abzufragen. Es geht hier nicht um die Identifikation der Person des Administrators, sondern darum, dass ein aktueller User über einen Trusted Path nachweisen kann, dass er – und nicht ein Programm – den Account wechseln möchte. Das Recht dazu muss vorher durch einen Administrator-Account vergeben worden sein. Das Passwort für diesen abzufragen, stellt im Prinzip eine Sicherheitslücke dar, denn der Benutzer müsste es nicht wissen. Hier liegt wieder die Verwirrung zwischen den Begriffen „Identität“ und „Rolle“ zu Grunde. Man könnte argumentieren, dass das Passwort des Administrators sozusagen zu einer Multi-Faktor-Authentisierung in Form eines zweiten Geheimnisses führt. Das ist durchaus korrekt, denn die gewünschte Aktion ist sicherheitskritisch und gegen eine zweite Authentisierung wäre im

Prinzip an dieser Stelle nichts einzuwenden. Für eine Authentifikation müsste jedoch der Faktor etwas sein, das zum Benutzer und nicht zum Administrator gehört. Das Administratorpasswort dafür zu verwenden, ist ein grober Missbrauch, da er das Geheimnis des Administrators unnötigerweise verbreitet.

Die Ausführung fremden Codes zum Zwecke der Installation von Software, und dies mit Admin-Privilegien, ist schlicht eine sicherheitstechnische Katastrophe und müsste durch einen rein deskriptiven Prozess ersetzt werden.

Um auf die eingangs gestellte Frage zurückzukommen: Sichert Vista nun die Home-PC-Plattform gegen Malware etc.? Tatsächlich sichert Vista aufgrund der „No-Write-Up“-Regel die Microsoft-Teile des Betriebssystems besser als XP. Die User-Daten hingegen bleiben angreifbar.

Interessant ist, wie im Beispiel von Rutko eine Hierarchie aus Applikationen mit unterschiedlichen Sicherheitsleveln (die Autorin zeigt auch die nötigen Tricks, um ein Programm tatsächlich mit dem eingetragenen Level zu starten) erstellt wird. Ganz unten in der Hierarchie finden sich Applikationen wie Browser oder Mail, bei denen mit Malware-Attacken zu rechnen ist. Sie laufen mit geringen Privilegien und Dateizugriffe auf bestimmte Ressourcen sind durch eigene ACLs abgesichert. Damit wird verhindert, dass z.B. der Internet Explorer beliebig im Dateisystem schreiben kann. Auf höheren Leveln laufen Office-Programme. Der Grund dafür ist eigenartig: Diese Programme sind natürlich nicht wirklich sicherer, sondern sollen durch den höheren Level vor Attacken „von unten“ geschützt werden. Sie laufen also hochprivilegiert und können dadurch natürlich auch große Schäden anrichten.

Es gibt weitere Verbesserungen in Vista, die im Wesentlichen der Verhinderung von Buffer-Overflow-Attacken dienen, indem Pages als „non-executable“ gekennzeichnet werden und eine Randomisierung der Codeplatzierung im Speicher durchgeführt wird. Rutko bemerkt allerdings zurecht, dass die Gefahr durch fehlerhafte oder bösartige Treiber weiterhin besteht und keine Mechanismen zur nachträglichen Entdeckung von Kompromittierungen eingeführt wurden.

4.6.4 Privilegienanpassung in Betriebssystemen

Privilegien den jeweiligen Notwendigkeiten anzupassen, kann auf verschiedene Weisen vom Betriebssystem unterstützt werden. Additive und subtraktive Verfahren lassen sich auch bei der Zugriffskontrolle über Permissions (identitäts- oder ressourcenbezogen) anwenden, wie Abb. 4.26 zeigt: Die linke Seite stellt ein subtraktives Verfahren dar. Das Utility „DecreaseMyRights“ reduziert für die Ausführung eines Programms die statisch zur Verfügung stehenden Rechte eines Nutzers – und reduziert so die Mächtigkeit des Programms und damit auch den potenziellen Schaden durch einen Missbrauch. Unsön daran ist allerdings, dass vor dem Aufruf von „DecreaseMyRights“ der Nutzer mit vollen, also meist zu hohen Privilegien arbeitet – bereits ein Verstoß gegen POLA.

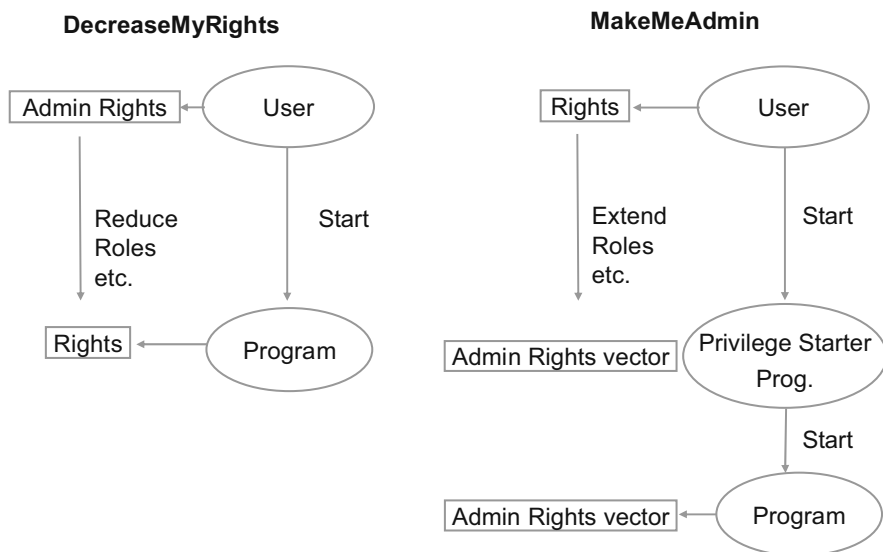


Abb. 4.26 Additive und subtraktive Verfahren bei der Zugriffskontrolle über Permissions

Die rechte Seite realisiert ein additives Verfahren. Hier arbeitet ein Nutzer mit geringen Rechten, benötigt aber plötzlich erweiterte Rechte zur Ausführung eines bestimmten Programms. Er verwendet jetzt „MakeMeAdmin“ – ein Werkzeug, das weitere Programme mit vollen Admin-Privilegien starten kann. Nur diese Programme laufen jetzt mit statischen hohen Privilegien.

An diesem Beispiel lässt sich gut erkennen, dass die Reduzierung von Privilegien auch auf Basis statischer Permissions möglich ist, dass aber die Anpassung der Privilegien an die tatsächlich benötigten Privilegien nur ungenügend möglich ist: Es bleibt immer noch zu viel Ambient Authority übrig.

Abschließend seien noch Beispiele für Zugriffskontrollen auf verschiedenen Ebenen genannt: Systrace im Berkeley Unix erlaubt die feingranulare Überwachung von System Calls von Programmen. Es ist sehr gut geeignet, nachträglich Server oder gefährliche Applikationen abzusichern. Das verwendete Prinzip ist ein Referenzmonitor (Sandbox) auf System-Call-Ebene (s. Abb. 4.27).

Systrace lässt sich auf einzelne Server oder Applikationen anwenden. Der Preis dafür liegt darin, herauszufinden, was das jeweilige Programm wirklich tun will/kann/muss, und dann die Regeln für den Referenzmonitor zu bestimmen.

Ungleich größer und umfassender ist der Mandatory-Access-Control-Ansatz von SELinux (Security Enhanced Linux, vgl. [McCa]), einer sicheren Linux-Variante, die durch die amerikanische NSA gefördert wurde und heute als Open Source in den meisten Linux-Distributionen verfügbar ist: Hier kann das ganze Betriebssystem inklusive Applikationen in eine Sandbox aus Regeln eingesperrt werden. Die Regeln sind nicht so granular wie System Calls, sondern beziehen sich nur darauf, welche Programme welche Ressourcen anfassen dürfen, welche Programme welches andere Programm aufrufen dürfen und welche Nutzer bzw.

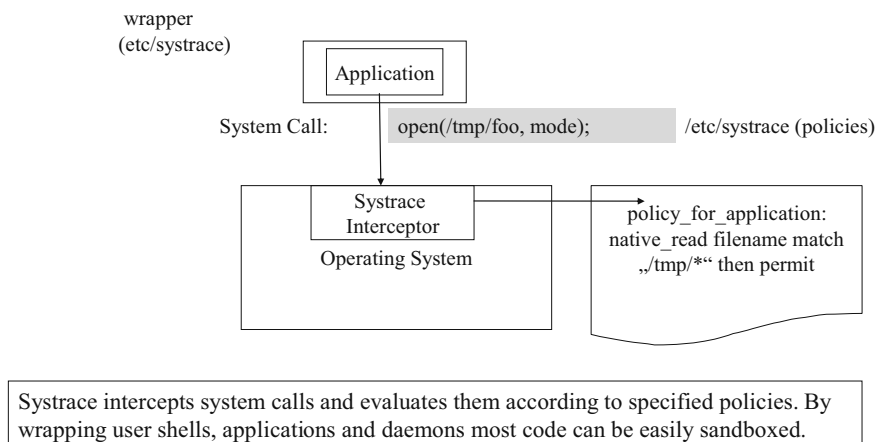


Abb. 4.27 Systrace in OpenBSD

Rollen welche Programme starten dürfen bzw. welche Rollenwechsel vorgenommen werden können.

Leider ließ sich SELinux ursprünglich nicht für einzelne Programme einschalten, sondern nur global für das gesamte System. Der Konfigurationsaufwand war dementsprechend hoch, wird heute aber zunehmend von Betriebssystemherstellern wie Redhat geleistet. Ernüchternd ist aber die Tatsache, dass viele der nötigen Regeln und Erlaubnisse durch Reverse-Engineering der Programme gewonnen werden müssen – ein Zeichen für den primitiven Stand der Softwaretechnik in Bezug auf die explizite Kennzeichnung von Abhängigkeiten, Ressourcenbedarf etc.

Ein grundsätzliches Problem existiert bei allen diesen Referenzmonitoren, die auf statischen Informationen über Rechte beruhen. Sehr generischen Programmen wie Editoren oder Shells können vorab kaum fixe Rechte auf bestimmte Files zugeordnet werden. Die Konfiguration solcher Programme besteht dann auch häufig aus sog. Wildcards, d. h. es werden Regular Expressions statt exakter Definitionen eingesetzt und im Allgemeinen sind solche Wildcards dann von der Form „*.*“, sprich der Zugriff auf alle Ressourcen ist erlaubt. Was dem Ansatz auf Basis von Referenzmonitoren fehlt, ist die Möglichkeit, dynamisch Rechte in Form von Capabilities wie z. B. geöffneten Filedeskriptoren zwischen Applikationen zu transportieren und somit sehr granular, aber dynamisch die Zugriffsrechte zu beschränken.

4.6.5 Privilegienanpassung in Applikationen

Ein schönes Beispiel für Reduktion von Privilegien durch Identitätsänderungen ist die Polaris-Umgebung der Firma Combex (www.combex.com). Polaris löst das Problem der Mailanhänge, die mit Viren versehen sein können, dadurch, dass es einen unprivilegierten Nutzer erzeugt und die Office-Programme zur Bearbeitung

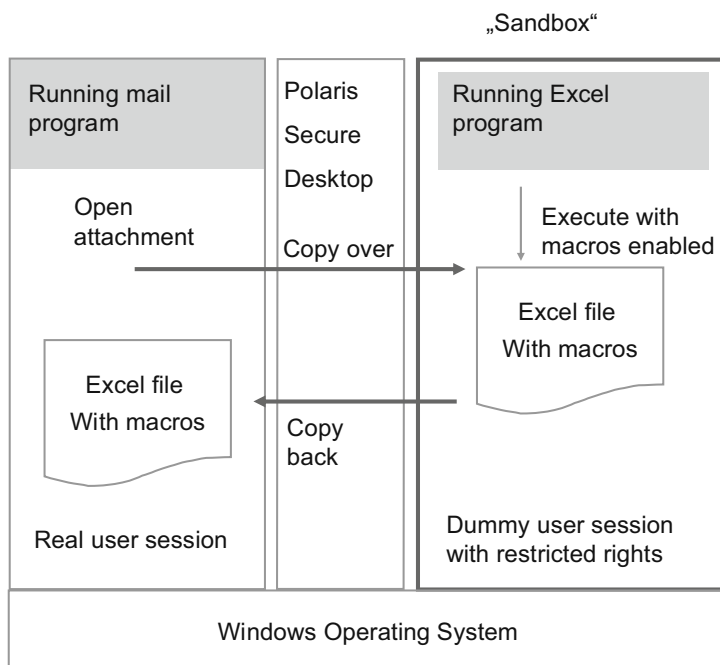


Abb. 4.28 Unprivilegierter User fungiert in Polaris als Sandbox

des Anhangs unter der Identität dieses Nutzers ausführt. Der Anhang wird zunächst in den Bereich des unprivilegierten Nutzers kopiert, dort mit dessen geringen Rechten bearbeitet und anschließend wieder zum ursprünglichen Empfänger der Mail zurückkopiert. Abbildung 4.28 zeigt die Ausführung der eigentlichen Applikation als unprivilegierter User.

In dem Anhang evtl. enthaltene Malware läuft somit nur mit den geringen Rechten des unprivilegierten Nutzers. Sie könnte zwar das File (also die Mail) beschädigen, aber nicht auf andere Files übergreifen und sich dadurch auch nicht vermehren. Die Gefahr von böartigen Makros in Anhängen wird hier also nicht durch Filterung vermindert, sondern dadurch, dass der böartige Code keine Rechte erhält, um Schaden anzurichten.

4.7 Die Sicherheit von Servern

Informationen über die Kunst, einen Server abzusichern (gemeint ist hier natürlich Serversoftware, nicht eine spezielle Maschine), sind leider teilweise schwer zu lokalisieren. Noch schwieriger wird es, wenn man nach sicherem Serverdesign fragt, also wie weit bereits bei der Architektur einer Serverlösung Sicherheit berücksichtigt werden kann. Dieser Abschnitt soll die diesbezüglichen Grundprinzi-

pien deutlich machen. Auf Application-Server und ihre Sicherheit gehen wir weiter unten in Kap. 7 detailliert ein.

4.7.1 *Grundkonzepte der Serversicherheit*

Wieso raten die Installationshandbücher von Betriebssystemen dazu, alle nicht benötigten Server (manchmal auch Services genannt) abzuschalten? Die Antwort lautet meist: Weil sie potenziell gefährlich sind. Die Begründung dafür fällt allerdings schon etwas schwerer.

Grundsätzlich ist ein nach außen offener Server schlicht ein weiterer Eingang in ein System. Ein Betriebssystem hat diverse Eingänge, z. B.:

- Desktop Login,
- Remote Login,
- USB-Anschlüsse.
- Harddisk,
- Netzwerk und
- Fileserver.

Bei den meisten von diesen Eingängen wartet ein Serverprozess auf Daten von außerhalb. Bei manchen von ihnen wird eine Gegenseite authentisiert, bei manchen nicht. Meldet sich ein Benutzer an einem System an, um auf ihm – mit seinen Rechten – zu arbeiten, dann wird er zwangsweise vom Betriebssystem authentisiert und danach impersonifiziert: Ab dem Moment der Impersonifizierung läuft jedes Stück Software, das der Nutzer startet, mit den Rechten dieses Nutzers, und nicht mit den Rechten eines Administrators. Der Prozess der Authentisierung selbst hingegen läuft meist mit Administratorrechten.

Diese zwangsweise Authentisierung gilt jedoch nicht für alle Server. Das Erzeugen eines Servers ist softwaretechnisch mittlerweile sehr einfach geworden: Nur ein paar Zeilen Java Code, und ein Dummy-Server ist fertig (Abb. 4.29, nach [Flan]).

So oder mit den direkten Socket API Calls in C/C++ fangen die meisten Entwickler an, Server zu programmieren und profitieren dabei zunächst davon, dass die Netzwerk-Stacks keine Authentisierung oder Autorisierung erzwingen. Der Code oben stellt einen rudimentären Server dar, der jegliches Kommando vom Netzwerk liest, ausführt und das Resultat zurückgibt, bzw. einen Dateinamen erhält, die Datei lokal liest und sie zurückschickt.

Das englische Verb „to serve“ heißt auch „ausführen für“ und der obige Server tut genau das: Er benutzt beim File-Zugriff bzw. beim Ausführen der Kommandos die Rechte der Identität, mit der er gestartet wurde. Starten wir obigen Code als Administrator, ist also der Zugriff auf alle Ressourcen des Systems möglich.

Die nächsten Schritte des Entwicklers sind dann die Trennung von Server-Code und Service-Code. Damit entsteht die Möglichkeit, dass ein einzelner Server mehrere verschiedene Services ausführen kann. Danach wird eine einfache Authenti-

```

ServerSocket ss = new ServerSocket(port);
for(;;) {
    Socket client = ss.accept();
    BufferedReader in = new BufferedReader(
        new InputStreamReader(client.getInputStream()));
    PrintWriter out = new PrintWriter(client.getOutputStream());
    while((line= in.readLine()) != null) {
        // read command (or filename) from network
        // execute command
    }
    out.print(result);
}

```

Abb. 4.29 Beispielhafter Java-Server-Code

sierung eingebaut. Für die Authentifikation und auch die Frage der Rechte des Requests wird der Zugriff auf eine Autorität nötig, die über Identität und Rechte Auskunft geben kann. Diese Autorität – auch *Realm* oder *Domain* genannt – ist meist repräsentiert durch eine *Registry*: Ein Verzeichnis von Identitäten, Credentials (z. B. Passwörter) und Rechte (z. B. Rollen).

Die Frage, welche Registry der Server zur Validierung eines Requests verwendet, hat riesige Konsequenzen: Sie bestimmt, in welchem Bereich diese Validierung gilt, bzw. wer diese Validierung anerkennen wird.

Abbildung 4.30 zeigt die verschiedenen Möglichkeiten für die Realms. Zunächst könnte der Request beispielsweise gegen die lokale Betriebssystem-Registry (OS-Realm) validiert werden. Dazu muss die Identität, die den Request gestellt hat, dort ein eingetragener Nutzer sein. Nehmen wir an, das ist der Fall. Der Server empfängt also Identität und Credentials des Aufrufers und validiert sie gegen die OS-Registry.

Ab jetzt hat der Server die Möglichkeit, weiter unter seiner eigenen Identität zu arbeiten. Dies hat aber als Konsequenz, dass der Server jeden Zugriff auf eine Ressource, die er im Laufe der Abarbeitung eines Requests benötigt, prüfen muss, nämlich daraufhin, ob der validierte Aufrufer das Recht hat, die entsprechende Ressource zu nutzen. Der Zugriff erfolgt dann zwar mit den Rechten des Servers, geprüft werden aber die Rechte des Aufrufers. In manchen Fällen wird sich der Server entscheiden, dennoch mit der eigenen Identität auf eine Ressource zuzugreifen – dies ist dann eine sog. *privilegierte Operation*, da sie mit anderen Rechten als denen des Aufrufers ausgeführt wird.

Der Server hat jedoch noch eine weitere Möglichkeit: Er könnte die Identität wechseln und den Aufrufer impersonifizieren, d. h. gegenüber dem System, das die Zugriffe prüft, die Identität des Aufrufers annehmen. Der Effekt im Code ist

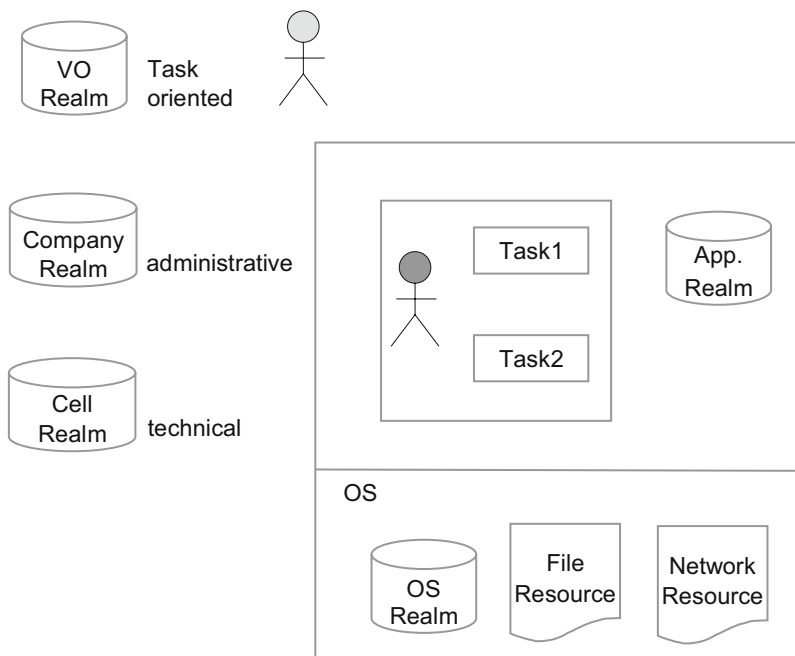


Abb. 4.30 Unterschiedliche Realms bei der Authentisierung eines Requests

gewaltig: Jetzt muss der Server nicht mehr bei jedem Ressourcenzugriff vorher die Rechte des Nutzers prüfen. Das System übernimmt das für ihn, und zwar das System, dessen Realm verwendet wurde, um den Nutzer zu authentifizieren. Falls der Nutzer über das Betriebssystem authentisiert wurde und der Server danach die Identität des Nutzers angenommen hat, wird bei einem Dateizugriff automatisch vom Betriebssystem geprüft, ob der Nutzer das Zugriffsrecht bezüglich dieser Datei besitzt. Selbst wenn der Server einen Codierfehler hat und auf eine Datei zugreift, die nur Administratoren lesen dürfen, wird das Betriebssystem den Zugriff ablehnen.

Zu beachten ist hier jedoch, dass der Server vor der Impersonifikation mit den eigenen Rechten lief. Sind diese sehr groß, stellt sich die Frage, wie kritisch die Aktionen des Servers waren, die unter diesen mächtigen Rechten durchgeführt wurden, und ob sie von außen beeinflusst werden könnten.

Auch mithilfe der Java 2 Security können solche automatischen Prüfungen genutzt werden, allerdings sind die Benutzer dort meist nicht über eine Registry des Betriebssystems autorisiert, sondern über ein LDAP oder eine andere Datei, die nichts mit dem Betriebssystem zu tun hat. Entsprechend kennt das Betriebssystem diese Identitäten nicht und eine Impersonifizierung findet nicht statt. Stattdessen prüft das Java-Runtime-System die Zugriffe automatisch (siehe dazu Kap. 5, Java Sprach- und Plattformsicherheit). In beiden Fällen – Authentisierung über Betriebssystem oder Java 2 Security – sind aber besondere Privilegien für das Wechseln der Identität nötig.

Ist die Impersonifikation also sicherer als das „Handeln im Auftrag“? Das kommt schlicht auf die Rechte des Nutzers im System an: Was wäre, wenn die authentifizierte Identität sich als Systemverwalter herausstellt? Dann greift der Server nach der Impersonifizierung mit den vollen Root-Rechten zu. Die Frage ist, ob das gewollt ist oder eher ein Sicherheitsrisiko darstellt. Selbst wenn ein normaler Nutzer impersonifiziert wird, kann dies positive und negative Effekte haben:

- Der Server kann im Namen des Nutzers Dateien verwalten, auf die nur dieser Nutzer Zugriff hat (ein positiver Effekt, da er POLA entspricht).
- Der Server kann auf weitere Dateien zugreifen, die dem Nutzer gehören, aber mit dem Server selbst und seiner Aufgabe nichts zu tun haben (ein negativer Effekt, da er Rechteausweitung statt -einschränkung beinhaltet).

Das bedeutet, dass bei einer Impersonifizierung, die Zugriff auf weitere Ressourcen einer Domain erlauben würde, die Schnittmenge der dadurch erworbenen Rechte (des Nutzers) mit den Rechten, die für den Server selbst definiert sind, gebildet werden sollte. Der Server sollte dann in einer Sandbox laufen, die seine Rechte unabhängig vom jeweiligen Aufrufer definiert und einschränkt.

Bis heute haben die meisten Betriebssysteme oder Applikationen Probleme mit dem „Handeln im Auftrag“. Die Implementation von sinnvollen Stellvertreterlogiken in Applikationen ist ein großes Softwareproblem, wir werden darauf im Zusammenhang mit Enterprise-Applikationen noch zurückkommen.

Was passiert nun, wenn sich der Server in der Verwendung der Identitäten irrt, er also entweder mit seinen Privilegien zugreift, wo er die des Nutzers hätte nehmen sollen, oder umgekehrt? Sehr schnell sind in diesem Fall illegale Zugriffe und damit Sicherheitslöcher möglich. So könnte der Server in unserem Socket-Beispiel übergebene Dateinamen nicht richtig prüfen (z. B. auf ../.-Kombinationen) und dadurch auf Dateien außerhalb des eigentlichen Serverbereichs zugreifen. Verstärkt wird die Gefahr noch durch die Tatsache, dass auf vielen Systemen viele Ressourcen für alle Nutzer lesbar sind, obwohl dies nicht nötig wäre (Ambient Authority).

Bevor wir uns mit der konkreten Absicherung von Server-Applikationen befassen, stellen wir abschließend die Gründe für die Gefährlichkeit von Server-Applikationen noch einmal kurz zusammen:

- Server sind einfach zu erstellen, bieten aber immer potenzielle Einfallstore für Angreifer.
- Selbstgebaute Authentisierungen können gefährlich sein.
- Hohe Rechte für den Server sind im Fehlerfall gefährlich.
- In den Systemen ist meist viel Ambient Authority verfügbar.
- Server irren sich bei den Zugriffsrechten (Confused-Deputy-Problem).
- Server machen Fehler bei der Umsetzung von Eingabe-Strings in Ressourcen.
- Manche Server implementieren sehr verschiedene Funktionen, die es schwierig machen, das gesamte Gefahrenpotenzial abzuschätzen.
- Server müssen teilweise privilegierte Operationen ausführen und brauchen daher spezielle Rechte, zumindest für gewisse Zeit. Dies ergibt Angriffsmöglichkeiten.

- Falls der Aufrufer nicht gegen die Betriebssystem-Registry autorisiert wurde, haben die Betriebssysteme keine Möglichkeit, die Aktionen des Servers intern zu kontrollieren.
- Die Rechte der Aufrufer können weit mehr umfassen, als die Server-Applikation eigentlich benötigen würde. Ohne sandboxartige Einschränkung der Server-Applikationen existiert auch dadurch Ambient Authority.

Noch nicht berücksichtigt haben wir bisher die Frage, wie die Sicherheit des Servers selber zu bewerkstelligen ist und wo die Grenzen des Betriebssystems hier liegen. Dem wenden wir uns jetzt zu.

4.7.2 Sicherheitsrelevante Architekturmerkmale von Servern

Das, was im klassischen Sinn als „Server“ oder „Server-Applikation“ bezeichnet wird, setzt sich zusammen aus einer Ausführungsumgebung (meist ein Prozess), dem Applikations- oder Service-Code, sowie dem User State, der während des Requests bearbeitet wird (s. Abb. 4.31).

Zum Server gehört aus logischer Sicht auch der Zustand aller nutzerbezogenen Daten in der Datenbank. Die Datenbank befindet sich bei größeren Applikationen

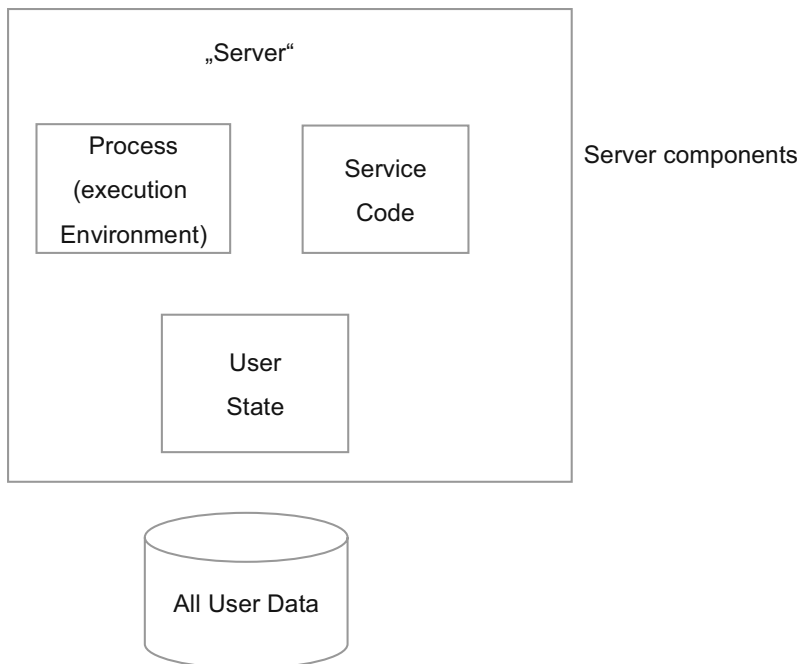
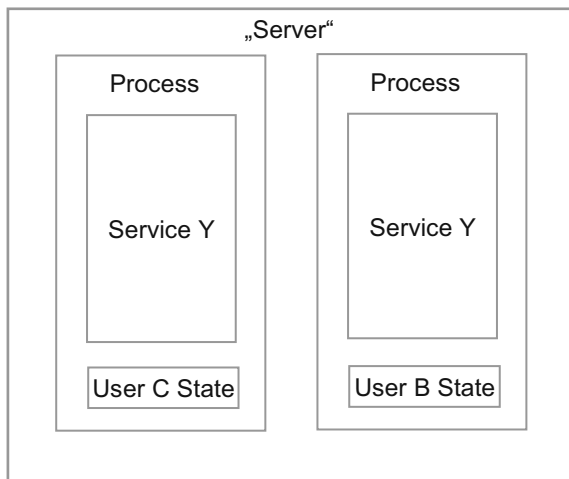


Abb. 4.31 Komponenten eines Servers

Abb. 4.32 Ein Prozess pro Nutzer



meist auf einer separaten Maschine, sowohl aus Performance- wie auch aus Sicherheitsgründen.

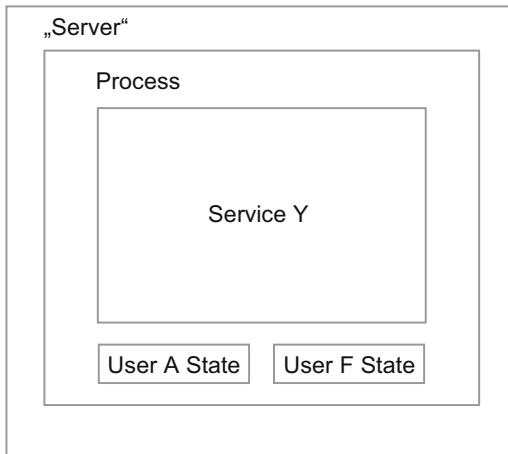
Die Komponenten Prozess, Applikationscode und User State verschmelzen bei kleineren, dedizierten Server-Applikationen zu einer Einheit im Code. Größere Server, vor allem solche, die unterschiedliche Services unterstützen, trennen die Komponenten jedoch voneinander. Application Server sind dafür ein typisches Beispiel. Zur Laufzeit müssen die Komponenten natürlich wieder zusammengeführt werden. Dies kann auf unterschiedliche Weisen mit jeweils verschiedenen Konsequenzen für die Sicherheit geschehen.

Im ersten Fall (s. Abb. 4.32) wird für jeden Nutzer ein eigener Prozess gestartet, der dann einen bestimmten Service ausführt. Hier besteht eine starke Trennung der Nutzer und Services untereinander, die Kompromittierung eines Prozesses betrifft immer nur einen User und – bei richtiger Organisation des Zugriffs auf die Datenbank – die Daten dieses Nutzers in Bezug auf diesen Service, also nicht alle Daten des Nutzers und schon gar nicht die Daten aller Nutzer.

Der einzige wirkliche Nachteil dieser Architektur ist, dass sie sehr viele Prozesse benötigt, nämlich so viele, wie sie Benutzer hat. Hat die Applikation darüber hinaus noch Sessioncharakter, d. h. hält sie den User State über einen einzelnen Request hinaus, dann müssen Prozessressourcen im schlimmsten Fall bis zum Session-Timeout gehalten werden. Diese Architektur, naiv implementiert, skaliert leider nicht, obwohl sie die beste Isolation der Komponenten leistet und damit im Angriffs- oder Fehlerfall den geringsten Schaden erfährt.

Eine andere Kombinationsmöglichkeit besteht darin, einen Prozess pro Service zu starten, diesen Prozess aber für alle Nutzer dieses Services zu verwenden (Abb. 4.33).

Die Anzahl der Prozesse ist hier gleich der Anzahl der Services. Innerhalb des Prozesses kann entweder single-threaded oder multi-threaded gearbeitet werden. Im single-threaded-Fall sollte die Architektur zudem *eventgetrieben* arbeiten, d. h. der Thread blockiert nicht beim Zugriff auf Ressourcen, die nicht fertig sind, son-



Process is either single-threaded, event-driven or multithreaded.

One Process per Service.

Abb. 4.33 Ein Prozess pro Service

dem wechselt sofort die Aufgabe. Der Vorteil der single-threaded, eventgetriebenen Architektur ist, dass keine Synchronisationsprimitive beim Zugriff auf Daten nötig sind, da es nur einen Thread gibt, der nicht durch andere Threads im gleichen Service unterbrochen werden kann.

Wie wichtig die Frage der Concurrency-Architektur ist, zeigt das Symbian Betriebssystem, das vor allem im Bereich von PDAs und Handys verwendet wird [GR]. Dort verzichtet man bei den allermeisten Application-Servern (auch Kleinstrechner profitieren in der Architektur von Serverprozessen) auf multithreading und arbeitet stattdessen mit sogenannten „Active Objects“. Diese Objekte definieren Methoden, die nicht blockieren dürfen und in kurzer Zeit beendet sein müssen. Die Methoden werden dann in Abhängigkeit von Events von einer Scheduler-Komponente aufgerufen.

Das Design der Active Objects erzwingt zwar eine unterschiedliche Architektur der Services (sie müssen in kurze, nicht-blockierende Funktionen aufgeteilt werden). Der Lohn ist aber, dass es keine Fehler durch Synchronisationsprobleme im Code gibt. Solche Fehler eröffnen sehr häufig auch Angriffsflächen.

Ein Nachteil der eventgetriebenen Architektur liegt in der Notwendigkeit, die Request-Daten selbständig verwalten zu müssen. Diese liegen ansonsten während der ganzen Laufzeit des Requests einfach auf dem Stack des Threads. Im Multithreaded Fall sollte ebenfalls eventgetrieben gearbeitet werden. Hier sind sehr hohe Request-Zahlen möglich. Die Java Runtimes litten vor der Umstellung auf die sog. NIO (New I/O), die eventgetrieben arbeitet, unter Performanceproblemen, da für jeden Request ein eigener Thread nötig war.

In beiden Fällen muss jeweils zu Beginn einer Verarbeitung eines Requests der User State mit dem jeweiligen Bearbeitungs-Thread verbunden werden

(s. Abb. 4.34). Dazu gehört auch der sog. Sicherheitskontext, d. h. die Identität und die Rechte/Credentials des Nutzers, der den momentanen Request gestellt hat.

Das Prinzip „ein Prozess pro Service“ hat sehr gute Isolationseigenschaften. Wenn ein Service korumptiert ist, führt dies allerdings dazu, dass potenziell alle Nutzerdaten dieses Services (richtige Datenbankzugriffsarchitektur vorausgesetzt) betroffen sein können.

Die nächste Kombination (Abb. 4.35) zeigt die Architektur eines heutigen Application Servers.

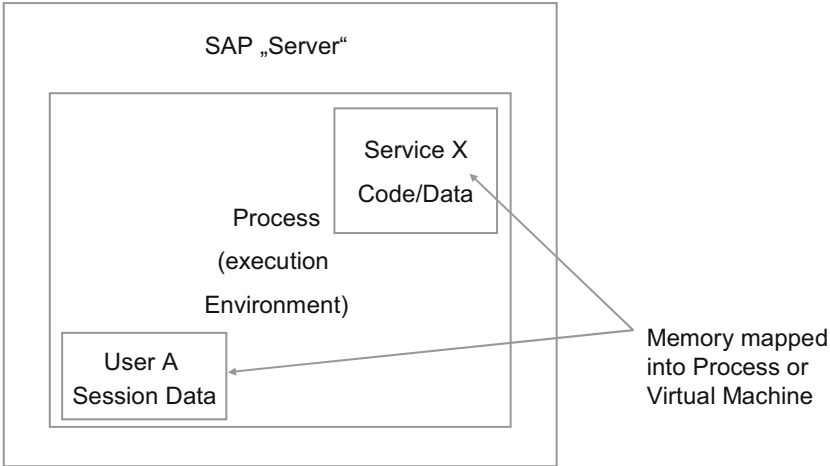


Abb. 4.34 Verbinden der Nutzerdaten mit dem Service

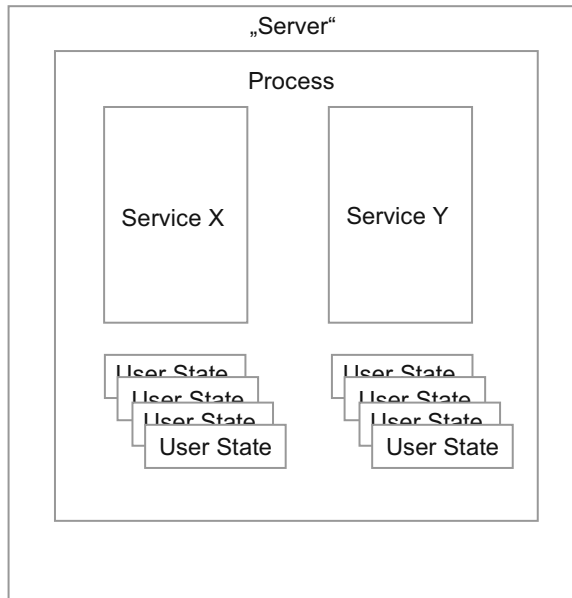


Abb. 4.35 Ein Prozess für viele Nutzer und Services

Hier verarbeitet ein Prozess (oft mithilfe einer Virtual Machine) mehrere Requests unterschiedlicher Nutzer, die verschiedene Services verwenden. Die Verarbeitung ist meist multi-threaded, d. h. es gibt einen Thread pro Request.

Wenn dieser Prozess kompromittiert ist, dann sind potenziell alle Daten der Nutzer über alle Services hinweg betroffen – es sei denn, die im Innern des Prozesses laufende Virtual Machine kann die Services und Nutzer auch bei der Kompromittierung eines Services erfolgreich voneinander trennen. Auf die Möglichkeiten, dies auf Basis von Java zu tun, gehen wir im Kap. 7 später genauer ein. Dort untersuchen wir die Isolierung bezüglich Administration, Datenbankzugriff und Applikationen.

Die Komplexität eines solchen Applikation-Servers ist durch multi-threading und verschiedenste Services sehr hoch. Die Tatsache, dass zudem gleichzeitig die Sicherheitskontexte unterschiedlicher Nutzer im Prozess verfügbar sind (mit jeweils unterschiedlichen Rechten), macht die korrekte Isolierung der Komponenten voneinander noch wichtiger. Allerdings ist dies schwierig, da ja alle Services und Requests innerhalb eines Prozesses laufen. In Abschn. 4.7.3 wollen wir an einem konkreten Beispiel untersuchen, wie sich eine bessere Isolierung softwaretechnisch erreichen lässt.

4.7.3 Absicherung von Servern am Beispiel des OKWS-Web-Servers

OKWS ist der Versuch, einen Webserver zu bauen, der sowohl sicher als auch performant ist. Die oben dargestellten Prinzipien der Schadensbegrenzung durch Isolation sollten also hier verwirklicht werden. Die OKWS-Architektur folgt dabei dem oben geschilderten subtraktiven Modell der Privilegieneinschränkung, indem ein mächtiger Launcher-Prozess weniger privilegierte Arbeits- und Helferprozesse erzeugt und sie in Jails einschließt.

Interessant aus softwaretechnischer Sicht ist dabei die Erkenntnis, dass die Architektur der Software selbst einen großen Beitrag zur Sicherheit leisten kann und nicht nur kryptografische Mittel oder Rechteprüfungen. Dabei werden wir Sicherheitsprinzipien mit softwarebasierten Lösungsmustern zu Bausteinen sicherer Applikationen verknüpfen. Ein Beispiel dafür wird die Regelung des Zugriffs auf Datenbanken sein, wie sie in Server-Applikationen ja permanent nötig ist.

Maxwell Krohn beginnt seine Beschreibung von OKWS in [Krohn] mit folgenden Sicherheitsprinzipien für den Bau sicherer Server:

- Isolierung der Serverprozesse durch chroot oder jails,
- Serverprozesse laufen als unprivilegierte Nutzer,
- der Zugriff auf Datenbank sollte minimale Rechte besitzen und
- verschiedene Funktionalitäten sollten auf unterschiedliche Prozesse aufgeteilt werden.

Basierend auf einer der oben diskutierten Serverarchitekturen, nämlich dem Konzept „Ein Prozess, ein Service, mehrere User“, werden diese Prinzipien in der in Abb. 4.36 dargestellten Architektur umgesetzt.

Zentrales Prinzip dieser Architektur sind unabhängige Prozesse, die – eingesperrt in Jails (d. h. spezielle Verzeichnisse) – miteinander durch Remote Procedure Calls kommunizieren. Direkter Zugriff auf Filterressourcen ist nur innerhalb des jeweiligen Verzeichnisses pro Service möglich.

Logisch gesehen wird dadurch eine Objektkapselung erzeugt. Der Log-Service ist der einzige, der auf das Logfile zugreifen kann. Wird ein Service kompromittiert, kann er zwar das Logfile mit sinnlosen Einträgen aufblasen, er kann aber keine Log-Einträge löschen. Analog ist der Konfigurationsserver der einzige, der auf Konfigurations-Files Zugriff besitzt.

Betrachten wir nun die Art und Weise, wie das gesamte System initialisiert wird: Ein privilegierter Loader-Prozess startet die anderen Prozesse als Kindprozesse. Diese Kindprozesse erben offene Kommunikationskanäle untereinander. Der Loader ist der einzige privilegierte Prozess in diesem Design, da er privilegierte Sockets öffnen muss (d. h. er kann Ports mit Nummer kleiner 1024 öffnen). Besonders wichtig ist, dass die eigentliche Applikationslogik – oft die schwächste Stelle in Bezug auf Sicherheit – in unprivilegierten Prozessen läuft, die zudem nur auf ihr jeweils eigenes Verzeichnis zugreifen dürfen. Diese Prozesse erhalten

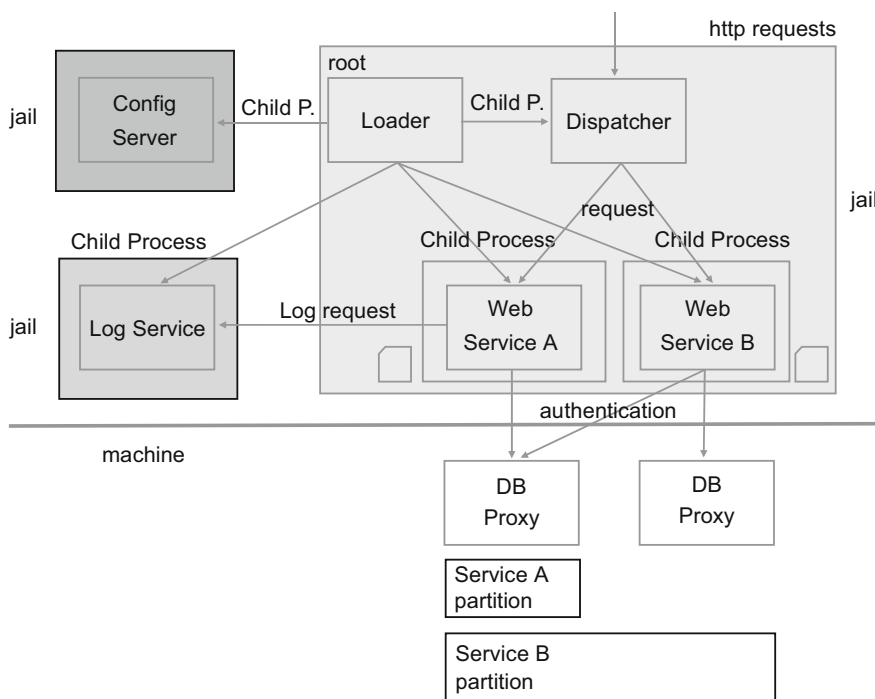


Abb. 4.36 Architektur des OKWS-Web-Servers

dynamisch verschiedene Nutzer- bzw. Gruppenidentitäten zugewiesen und damit verschiedene Rechte.

Der Dispatcher-Prozess empfängt http-Requests von außen, interpretiert sie und leitet sie an die zuständigen Serviceprozesse weiter. Es ist wichtig, dass der Dispatcher klein und überschaubar bleibt, da er sicherheitskritische Interpretationen von symbolischen Informationen vornimmt.

Neben der Prozessaufteilung ist der Zugriff auf die Datenbank von besonderem Interesse. Hier gilt es, den potenziellen Schaden durch einen kompromittierten Service zu begrenzen, indem der einzelne Service am besten nur Zugriff auf die Daten eines Users erhält, die er für den Service tatsächlich benötigt. OKWS implementiert dieses Prinzip durch eine Anzahl von Proxy-Servern, die – auf dem Datenbankserver laufend – eine breite Schnittstelle zur Datenbank besitzen, dem einzelnen Service jedoch nur eine auf ihn zugeschnittene, schmale Schnittstelle anbieten. Zudem authentisieren sie den rufenden Service. Sollte jetzt tatsächlich ein Service kompromittiert werden (oder einen Programmierfehler aufweisen), dann schränkt der Proxy die Zugriffsmöglichkeiten auf Daten drastisch ein. Dies minimiert die kompromittierten Daten in der Datenbank (s. Abb. 4.37–40).

Sehen wir uns dazu die verschiedenen Auswirkungen auf die User-Daten an. Nehmen wir dazu an, dass in einer Datenbanktabelle Daten verschiedener User gehalten werden. Außerdem greifen verschiedene Services über ihre Prozesse auf diese Tabelle zu. Meist wird diesen Prozessen eine Identität gegeben, die globale Rechte für die Tabelle enthält. Somit kann ein kompromittierter Serverprozess Daten aller User und Services missbrauchen (Abb. 4.37).

Data in Database		
User A data		
User F data	Compromised Process which Serves Service C	
Service C partition		

Abb. 4.37 Größtmöglicher Schaden durch Zugriff von Servern auf komplette Tabelle

Wenn ein Service C nun lediglich Zugriff auf bestimmte Teile der Tabelle benötigt, dann kann in der oben geschilderten Architektur auf Basis von DB-Proxies

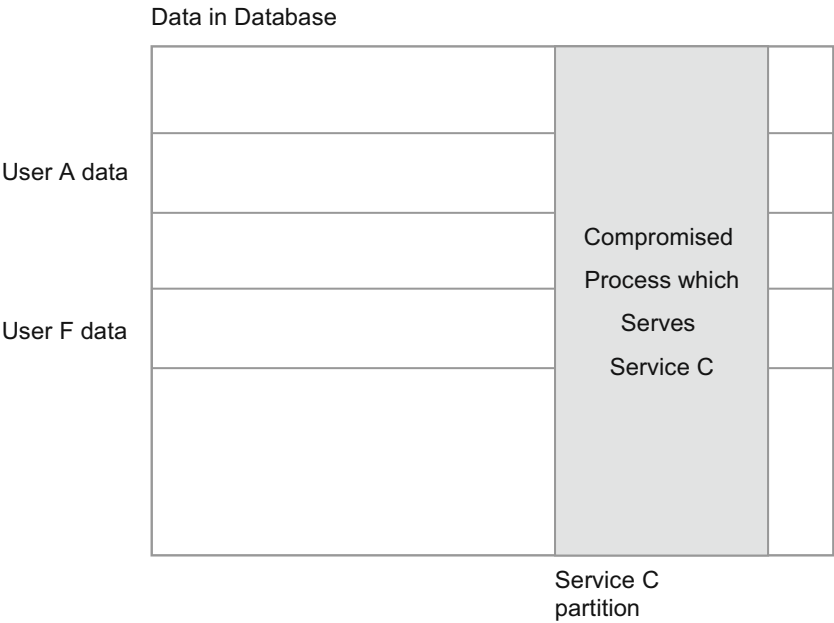


Abb. 4.38 Durch Proxy eingeschränkter Zugriff auf Daten eines Services

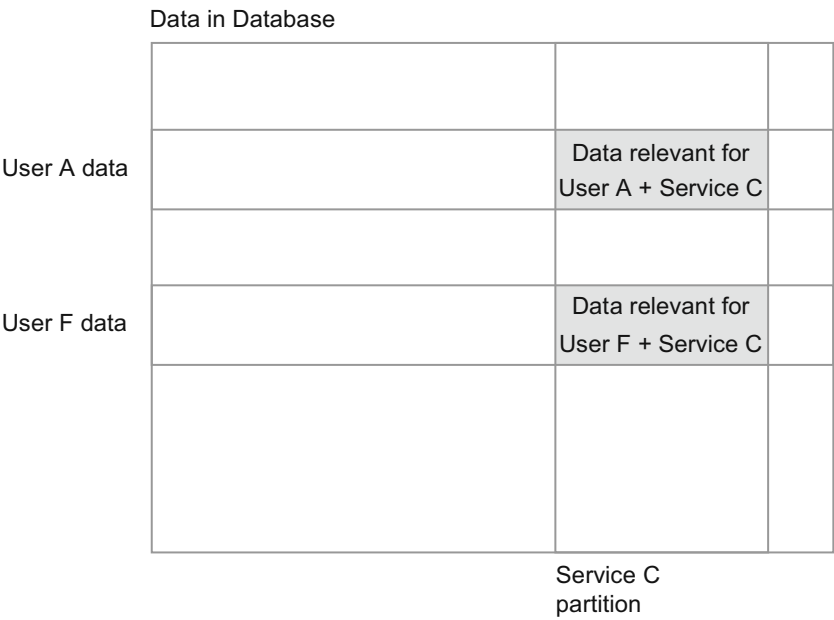


Abb. 4.39 Eingeschränkter Zugriff auf Daten eines Services und seiner User

diese Einschränkung durch Filterung der Requests von Service C erreicht werden (Abb. 4.38). Die Einschränkung des Zugriffs durch die Serverprozesse kann jedoch noch weiter gehen.

Die Reduktion von Autorität durch Softwarebausteine ist ein Pattern, das wir in Kap. 10 zu den Softwaremechanismen für sichere Systeme weiter untersuchen werden. Im Idealfall lässt sich damit der Missbrauch auf die Daten weniger User beschränken (Abb. 4.40).

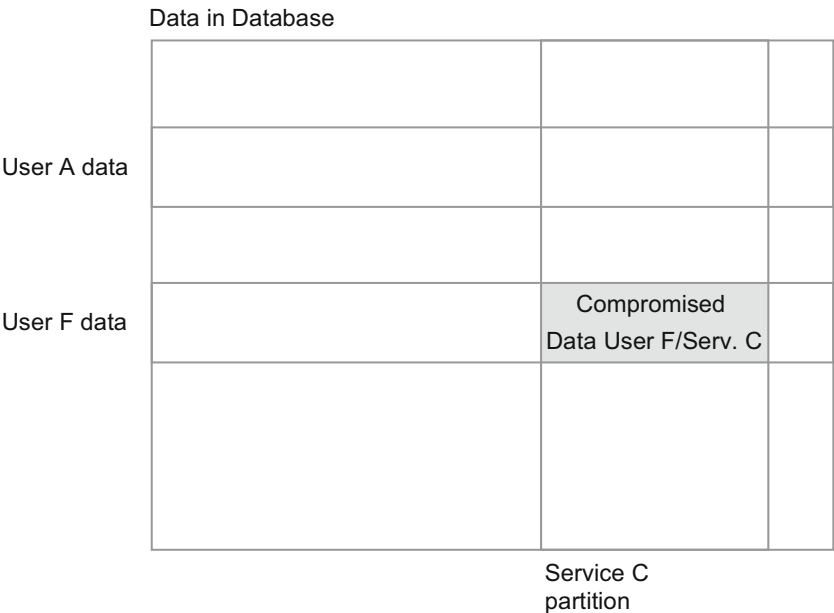


Abb. 4.40 Einschränkung kompromittierter auf eine Client-Identität und Service

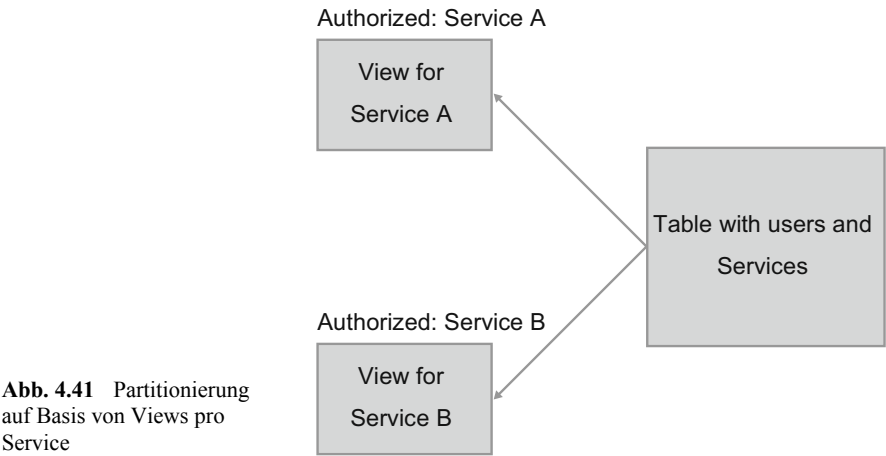


Abb. 4.41 Partitionierung auf Basis von Views pro Service

Eine häufig verwendete Alternative zu Softwarebausteinen sind sog. Datenbank-Views, die Ausschnitte von Tabellen repräsentieren und bei denen der Zugriff durch Server von der Datenbank kontrolliert werden kann. Hier wird die Selektion der Daten bei der Erstellung der Views durchgeführt. Jedem View können unterschiedliche Zugriffsrechte zugeordnet werden und gefährliche Tabellen – solche, die allgemeine Informationen mit sehr kritischen innerhalb einer Zeile mischen – können bequem separiert werden (Abb. 4.41).

4.8 Administration

Wie sollte ein Serversystem administriert werden? Über einen angemeldeten lokalen Benutzer oder über eine Netzwerkverbindung? In welchem Fall ist die Gefahr für den Server größer?

Die Betriebssysteme Windows und Linux geben darauf eine sehr unterschiedliche Antworten (s. [Petr]). So entscheiden sich die Linux-Administratoren normalerweise für eine Remote-Administration über eine sichere Verbindung zum Server (zum Beispiel über einen SSH-Tunnel). Zur Begründung wird gesagt, dass eine Remote-Administration das Serversystem weniger gefährdet. Der Zugang sowie die Rechte des Benutzers können eingeschränkt werden, und die Gefahren durch komplexe GUI-Systeme wie etwa XWindows ist geringer. Auch ist hier die Gefahr geringer, dass Viren und Trojanische Pferde durch Browser, Mail oder andere Dienste auf den Server gelangen.

Im Windows 2003 Server Administration Guide hingegen wird die lokale Administration, unter Ausnutzung aller Möglichkeiten grafischer Interfaces, empfohlen. Allerdings wird davor gewarnt, verschiedene Funktionen als lokal angemeldeter Administrator zu verwenden. Hier wird argumentiert, dass die größere Bequemlichkeit der Administration auch zu größerer Sicherheit führt.

Verwenden wir unser Konzept der Angriffsfläche in diesem Fall, dann erkennen wir, dass die Linux-Administratoren beim lokal angemeldeten Administrator eine größere Angriffsfläche für das System sehen. Technisch gesehen ist diese Auffassung richtig, speziell bei Betriebssystemarchitekturen wie Linux oder Windows. Im Windows-Fall wird eher der Bequemlichkeit Rechnung getragen und es bleibt im Ergebnis für den lokalen Fall eine größere Angriffsfläche übrig.

Woher kommt dann das Gefühl, dass die Verbindung zum Internet die größte Gefahr für einen Rechner darstellt? Offensichtlich kann die durch die Trennung von Rechnern theoretisch größere Sicherheit nicht in eine praktisch größere Sicherheit umgesetzt werden (ganz analog zur angeblich größeren Zuverlässigkeit verteilter Systeme, die jedoch mangels Redundanzen fast nie realisiert wird).

Es sind wohl in erster Linie zwei Faktoren, die dafür verantwortlich sind:

- Eine große Zahl von Schwächen in der Input-Validierung bei der beteiligten Software (OS, Middleware, Applikationen).
- Grundsätzliche Probleme, die Identität der Teilnehmer betreffend, und Verständnisprobleme beim Nutzer bezüglich der verwendeten Mechanismen: Mit wem kommuniziere ich eigentlich im Internet?

4.8.1 Race Conditions

In Kap. 2 haben wir bereits Beispiele für das sog. TOC2TOU-Problem gesehen: Dies bezeichnet ein Validierungsproblem, das durch parallele Zugriffe (Nebenläufigkeit) entsteht. Bei einer sogenannten *Race Condition* hängt das Ergebnis eines Prozesses vom zeitlichen Ablauf der Einzeloperationen ab. Charakteristisch dafür ist eine Reihe von Befehlen, die vom Sinn her atomar sein sollten (also als ein Befehl wirken sollten), jedoch als Einzelbefehle implementiert wurden. Dadurch entstehen zeitliche Lücken in der Ausführung der Befehle, die ein parallel laufender Prozess nutzen kann, um in der Zwischenzeit Änderungen durchzuführen (s. Abb. 4.42). Der nächste Befehl, der in der Reihe der Befehle ausgeführt wird, vertraut jedoch darauf, dass sich nach der ersten Prüfung nichts mehr geändert hat.

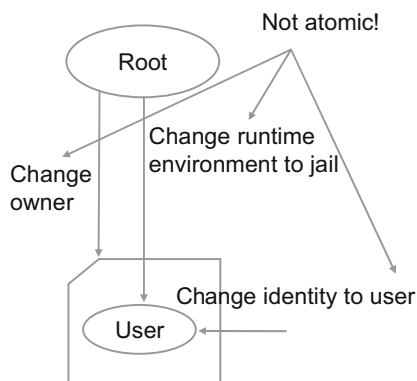


Abb. 4.42 Administration und Race Conditions

Jails strips off other rights

4.8.2 SU – verschiedene Implementationen

Die Utility „su“ kennt praktisch jeder Unix- bzw. Linux-Nutzer: Eingeloggt als normaler Benutzer, erlaubt sie es dem Benutzer, seine Rechte gewaltig zu erhöhen, indem er eine zusätzliche Identität – die „effective user ID“ des Superusers Root bekommt. Auf klassischen Unix-Systemen war die Voraussetzung für die Nutzung von „su“ auf Seiten des Benutzers, dass er die Credentials von Root (hier das Passwort) kannte. Somit fand durch das Ausführen von „su“ eigentlich nichts statt, was der Benutzer nicht ohnehin gekonnt hätte, nämlich sich als Superuser einzuloggen.

Nicht zu verwechseln ist „su“ mit sogenannten *setuid*-Programmen. Das sind Programme, deren Rechtevektor das *setuid*-Bit beinhaltet, und die bei Aufruf mit den Rechten des Programmeigentümers ausgeführt werden und nicht mit den Rechten des Aufrufers. Hier findet eine echte Privilegienerhöhung statt, wenn

auch nur temporär. Setuid-Programme werden uns im Zusammenhang mit Remote Services noch beschäftigen. Dort erlauben sie Servern bei geschickter Anwendung, mit minimaler Autorität zu laufen und trotzdem bei Bedarf privilegierte Kommandos auszuführen.

Das Abfragen der Credentials von Root beim Aufruf von „su“ bezweckt zwei Dinge. Erstens beweist der Benutzer durch die Eingabe des Superuser-Passworts seine Berechtigung, als solcher zu handeln. Ab diesem Moment wird bei der Ausführung eines Programms und dem Zugriff auf Ressourcen jeweils der Rechtevektor des Superusers zur Prüfung herangezogen. Zu Logging-Zwecken wird aber trotzdem immer die „real user ID“ des Ausführenden mitprotokolliert – oder sollte zumindest. Damit kann das System die echte Identität des als Superuser handelnden Benutzers feststellen – was bei einem direkten Login als Root nicht möglich ist. Denn Root ist nur ein sogenannter *Functional User* und entspricht keiner Identität im wirklichen Leben.

Der zweite Zweck der Abfrage der Credentials besteht darin, dem Benutzer diesen Wechsel in den Berechtigungen auch klar mitzuteilen. Mit anderen Worten: Es soll keinen automatischen Übergang von einer normalen in eine funktionale Identität geben.

Soweit zur bekannten Funktion von „su“. SELinux zeigt jedoch plötzlich davon abweichende Verfahren. So gestattet Fedora Linux einem Benutzer, der die Rolle „staff“ besitzt, die automatische Transition in die Rolle des Administrators durch Aufruf von „su“, ohne die Credentials von Root zu prüfen.

Dazu muss man wissen, dass SELinux für Programme auf einem Unix/Linux-System eine Art Sandbox herstellt. Nicht mehr die Rechte eines Aufrufers alleine entscheiden darüber, welche Aktionen ein Programm ausführen darf, sondern eine Datenbank mit Rechtezuordnungen an Domänen bestimmt die letztendlichen Rechte (s. Abb. 4.43).

SE-Linux: Type Enforcement and File Context Definitions

File contexts define the security contexts of resources belonging to a domain. A program running within a domain (sandbox) is restricted to use those resources and the resources are at the same time protected from the program (or other programs)

```
ping.fc: /bin/ping/* -- system_u:object_r:ping_exec_t
```

this line allows only ping_exec_t domain programs access to the files in the ping directory. Calling it in a system admin role does not change this!

```
ping.te: role sysadm_r typesp ing_t; allowp ing_te tc_t:file {getattr read}
```

This line allows programs in ping_t domain to access file objects in the etc_t domain through the methods getattr and read. [Mcca]

Abb. 4.43 Definition von Typen und Objekten in SE-Linux

So kann z. B. für Systemprogramme festgelegt werden, welche Files sie überhaupt lesen bzw. schreiben können. Dies wird zur Laufzeit vom erweiterten Kern des Betriebssystems geprüft.

Schnell fällt auf, dass die Funktion von Rollen – sonst das wichtigste Mittel, um feingranulare Zugriffsrechte zu modellieren – hier wesentlich weniger wichtig ist. Abbildung 4.44 zeigt den Einfluss der Domänendefinition sowie der Transitionsregeln auf die Zugriffsentscheidung. Die Konfiguration der Rechte in der SELinux-Konfigurationsdatenbank entscheidet letztlich über Zugriff und Ausführung, nicht der Rechtevektor eines Nutzers oder einer Rolle. Auch ein Superuser kann also mittels eines Programms nicht mehr tun, als es die Rechte des Programms erlauben.

Vor diesem Hintergrund ist jetzt zu prüfen, welche der drei alternativen Implementation von „su“ auf einem SELinux-System richtig ist, d. h. bei welcher Alternative der Schaden bei Ausführung von bösartiger Software möglichst gering bleibt:

- die klassische Form mit Abfrage des Root Credentials,
- die neue Form in Fedora Linux mit automatischer Transition von Rolle „staff“ in die Rolle „sysadmin“ oder
- die klassische Form, jedoch mit Abfrage des Benutzer-Credentials.

Zur ersten Alternative: Einer der oben erwähnten Gründe für das Abfragen des Root Credentials fällt offensichtlich sofort weg: Es muss nicht mehr geprüft wer-

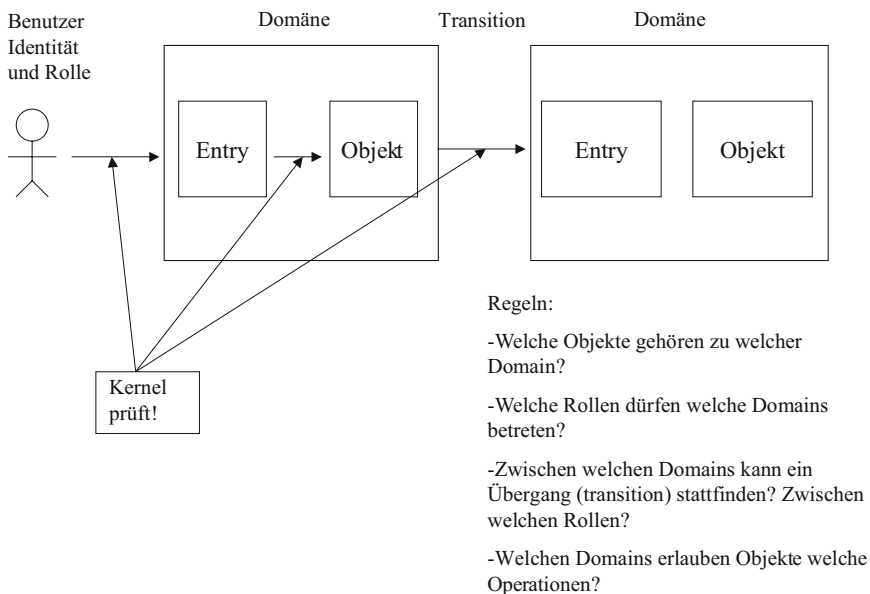


Abb. 4.44 Festlegung von Rechten in SELinux

den, ob Benutzer *X* das Recht hat, als „sysadmin“ zu arbeiten. Dieses Recht ist in Form einer Transformationsregel bereits in der SELinux-Rechtedatenbank festgelegt und kann dort vom Kernel nachgelesen werden. Aber wie steht es mit dem zweiten Grund, nämlich der Prüfung der „Liveness“ der Anforderung? Hier sind folgende Punkte zu klären:

- Lässt sich verhindern, dass Programme ohne Wissen des Benutzers eine solche Transition vollziehen? Das ließe sich zumindest teilweise durch SELinux-Transitionsregeln zwischen Domänen regeln.
- Lässt sich verhindern, dass in Abwesenheit eines Nutzers, der in der Rolle „staff“ eingeloggt ist, ein anderer Nutzer dies ausnutzt und über „su“ in die Rolle des Administrators schlüpfen kann? In der klassischen Form von „su“ lautet die Antwort: ja. Natürlich ist es trotzdem ein Sicherheitsproblem, wenn ein eingeloggter Nutzer seinen Arbeitsplatz verlässt, aber eine mögliche automatische Transition zu Root würde die Problematik noch verschärfen.

Bei der zweiten Alternative, der neuen (stillen) Form von „su“ unter Fedora Linux, haben sich die Entwickler mit Sicherheit daran orientiert, dass der erste Grund für die Abfrage der Root Credentials, wie bereits oben erklärt, weggefallen ist. Selbst wenn jedoch die Anzahl der Programme, von denen aus eine Transition zu „su“ möglich ist, durch Regeln drastisch eingeschränkt wird, bleibt als Sicherheitsrisiko die Möglichkeit des unbewachten Desktops. In diesem Sinne scheint die Fedora-Lösung etwas großzügig zu sein.

Nun zur dritten Alternative, dem Abfragen des benutzereigenen Kennung: Für diese Lösung lässt sich als Argument anführen, dass es keinen Sinn macht, zur bloßen Prüfung, ob der (ohnehin berechnete) Benutzer auch wirklich vor dem Bildschirm sitzt, das Superuser-Passwort abzufragen. Im Gegenteil, die Nutzung der SELinux-Regeln macht es gerade unnötig, dass der Benutzer dieses Passwort kennt. Die Abfrage des *eigenen* Passworts (oder noch besser eines zeitlich begrenzten Tokens wie bei Kerberos) ist hingegen als Mittel der direkten Autorisierung des Übergangs zu Root-Rechten zu verstehen und in diesem Sinne auch absolut nötig, wie unsere Diskussion der „Liveness“-Problematik zeigt. Ferner muss die grundsätzliche Erlaubnis, eine Aktion auszuführen, von dem konkreten Willen dazu zu einem bestimmten Zeitpunkt unterschieden werden. Man könnte also mit einiger Berechtigung sagen, dass die klassische „su“-Implementation mit Abfrage der Nutzer-Credentials unter SELinux die sicherste und bequemste ist.

Man könnte jetzt noch einwenden, dass die zusätzliche Abfrage des Root-Passwortes als Mittel der „Defense-in-depth“ verstanden werden kann: einfach als weiteres Merkmal in einer Multi-Faktor-Authentisierung. Der Wunsch nach einer Verstärkung der Authentisierung gegenüber einem bloßen User-Passwort ist durchaus verständlich, nur müsste es dann ein zusätzliches Merkmal des speziellen Users sein und nicht das Root-Passwort mit seinen weit reichenden Konsequenzen. Durch die Verwendung des Root-Passworts wird hier schlicht die Authentisierung mit der Autorisierung verwechselt.

4.8.3 *Installation*

Die Installation neuer Software bzw. von Updates stellt für jede Plattform ein hohes Sicherheitsrisiko dar. Entscheidend für das Risiko sowie die Wartbarkeit der Plattform ist jedoch die Art und Weise, wie die neue Software in das System gelangt. Viele Betriebssysteme verwenden hierfür sogenannte *Installer* – das sind Programme, die mit der jeweiligen Software mitgeliefert werden und die diese Software dann im System platzieren, Einträge in Verzeichnisse und Konfigurations-Files vornehmen, etc. Häufig sind diese Ressourcen jedoch durch das Betriebssystem geschützt, sodass der Installer mit Administratorrechten gestartet werden muss. Dies hat zwei entscheidende Nachteile zur Folge:

- Die Art und Weise, wie der Installer die Software installiert und konfiguriert, kann vom Betriebssystem nicht kontrolliert werden. Dies kann zu Konflikten führen und Probleme bei der Deinstallation verursachen.
- Der fremde Installer wird mit Administratorrechten ausgestattet und kann dadurch praktisch unkontrolliert Aktionen im System veranlassen. Dies setzt absolutes Vertrauen in die zu installierende Software bzw. den Installer voraus.

Verschiedene Plattformen (Windows/Symbian/Java) unterstützen deshalb die Signierung von Software durch Hersteller. Bei genauer Betrachtung erkennt man jedoch schnell, dass diese Art der Vertrauenssicherung keineswegs Software daran hindert, Schäden auf einer Plattform anzurichten: Durch die Signatur kann lediglich die Integrität der Software verifiziert und der Urheber der Software bestimmt werden – ob dieser auch vertrauenswürdig ist, ist eine ganz andere Frage.

Ein besserer Ansatz wird im Singularity-Betriebssystem von Microsoft oder der Open-Services-Gateway-Initiative (OSGI) verfolgt. Dort ist der Installer bereits Teil der Plattform und zu installierende Software muss durch Meta-Daten anzeigen, wie sie installiert werden möchte. Dies erlaubt eine genaue Prüfung durch das Betriebssystem und verhindert, dass fremde Software Administratorrechte erhalten kann. Auch kann die Plattform dadurch eine genaue Verwaltung der Artefakte, die zur Applikation gehören (Files, Libraries, Konfiguration), sicherstellen, was z. B. eine sichere Deinstallation erlaubt.

Ein weiteres großes Problem bei der Installation von Software sind Konflikte mit bereits installierten Komponenten. Darunter leiden nicht nur PCs im Heimumfeld, sondern auch professionell gewartete Terminal-Server-Installationen in großen Firmen. Natürlich besteht die Möglichkeit, solche Konflikte grundsätzlich dadurch zu vermeiden, dass jede Applikation eine eigene Instanz des Betriebssystems erhält, z. B. durch Virtualisierung oder durch separate Rechner (Blade-Ansatz). Dies ist jedoch eine sehr grobgranulare Lösung mit hohem Aufwand in Bezug auf Hardware und Verwaltung. Einfacher wäre es, wenn nur der Verwaltungsaspekt des Betriebssystems pro Applikation zur Verfügung gestellt werden könnte, jede Applikation also eigene Instanzen von Konfigurationsdateien und DLLs erhält. Eine solche Lösung vertreibt die Firma Softricity unter dem Stichwort „Applikationsvirtualisierung“: Eine Zwischenschicht zwischen Applikation

und Betriebssystem virtualisiert hier alle Dienste und Dateien, die sonst gemeinsam von Applikationen manipuliert würden. So erhält jede Applikation eine eigene Registry (vgl. [Dom]).

Literatur

- [ABW] M. Abadi, A. Birrell, T. Wobber, Access Control in a World of Software Diversity, Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS '05) Santa Fe, NM, June 2005
- [Cap] Capability Mailing Liste, <http://www.erights.org>
- [Dom] S. Dommel, Terminal Services: Funktionalität, Einsatzgebiete, Alternativen, Trends und der praktische Einsatz in einer webbasierten Umgebung, Diplomarbeit Hochschule der Medien 2006
- [Flan] D. Flanagan, Java Examples in a Nutshell, O'Reilly, 2004
- [Garf] S. Garfinkel, Design Principles and Patterns for Computer Systems That Are Simultaneously Secure and Usable, Dissertation, Massachusetts Institute of Technology (MIT) 2005, available online at <http://www.simson.net/thesis/>
- [Gnir] M. Gnirss, Mainframe Security, Vortrag anlässlich des 2.Security Day, Hochschule der Medien Stuttgart, 19. Oktober 2007
- [GR] A. Gerlicher, S. Rupp, Symbian OS. Eine Einführung in die Anwendungsentwicklung, dpunkt-Verlag 2004
- [Heis08] G. Heiser, The Role of Virtualization in Embedded Systems, 1st Workshop on Isolation and Integration in Embedded Systems, Glasgow, UK, April, 2008, http://www.ertos.nicta.com.au/publications/papers/Heiser_08.pdf
- [Heis07] G. Heiser, Virtualization for Embedded Systems, http://www.cs.rutgers.edu/news/colloquia/?action=getpdf&colloquium_id=2038
- [HEKKP] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, S. Petters, Towards Trustworthy Computing Systems: Taking Microkernels to the Next Level, ACM Operating Systems Review, 41(4), 3–11, (July, 2007), http://www.ertos.nicta.com.au/publications/papers/Heiser_EKKP_07.pdf
- [Hunt] G. Hunt et al., An Overview of the Singularity Project, <ftp://ftp.research.microsoft.com/pub/tr/TR-2005-135.pdf>
- [KEFK] M. Krohn, P. Efstathopoulos, C. Frey, F. Kaashoek et al., Make Least Privilege a Right (not a Privilege). Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS '05) Santa Fe, NM, June 2005
- [Knop] Dirk Knop, Automatische Exploits durch Patch-Analyse, Heise-News, <http://www.heise.de/newsticker/Automatische-Exploits-durch-Patch-Analyse--meldung/106976>
- [Krohn] M. Krohn, Building Secure High-Performance Web Services with OKWS, Master Thesis, Massachusetts Institute of Technology 2005, available online at <http://pdos.csail.mit.edu/~max/docs/okws.pdf>
- [Levy] H.M. Levy, Capability-based computer systems, available online at <http://www.cs.washington.edu/homes/levy/capabook/>
- [McCa] B. McCarty, SELinux: NSA's Open Source Security Enhanced Linux, O'Reilly 2004
- [Mehn] F. Mehnert, Kapselung von Standard-Betriebssystemen, Dissertation TU Dresden 2005, http://os.inf.tu-dresden.de/papers_ps/mehnert-phd.pdf
- [Orm] Tavis Ormandy, An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments. <http://tavis0.decsystem.org/virtsec.pdf>

- [Oster] Larry Osterman, Threat Modelling Rules of Thumb, <http://blogs.msdn.com/larryosterman/archive/2007/09/21/threat-modeling-again-threat-modeling-rules-of-thumb.aspx>
- [Petr] N. Petreley, Security Report Windows vs. Linux, http://www.theregister.co.uk/security/security_report_windows_vs_linux/
- [RFC 3552] E. Rescorla, B. Korver, Guidelines for Writing RFC Text on Security Considerations, www.ietf.org/rfc/rfc3552.txt
- [Slash] Slashdot Diskussion der “immutable laws of security” von Microsoft, <http://it.slashdot.org/article.pl?sid=07/10/01/1556258>
- [Rut] J. Rutko, Running Vista Every Day, <http://blogs.zdnet.com/security/?p=29&tag=nl.e589>, <http://theinvisiblethings.blogspot.com/2007/02/running-vista-every-day.html>
- [Tan] A. Tanenbaum et al., Can we make Operating Systems reliable and secure? IEEE Computer Magazine, May 2006, available online at www.computer.org
- [Tech] Microsoft Technet Artikel zu „Immutable Laws of Security“, <http://www.microsoft.com/technet/archive/community/columns/security/essays/10imlaws.mspx>
- [Zais] B. Zaiser, Sel4 – ein sicherer Microkernel. HDM 2008
- [Zal] M. Zalewski, Silence on the Wire: A Field Guide to Passive Reconnaissance and Indirect Attacks, No Starch Press 2005
- [ZDNet] M. Hülskötter, R. Joerges, Windows kollabiert unter seinem eigenen Gewicht, ZDNet.de, <http://cgi.zdnet.de/itmanager/?p=146>

Kapitel 5

Java Sprach- und Plattformsicherheit

In diesem und den folgenden Kapiteln werden wir drei verschiedene Aspekte von Sicherheit im Zusammenhang mit der Sprache Java beleuchten:

- Sprache und Plattform
- Enterprise Security
- Application Server Security

Beim ersten Punkt wird die Absicherung von Java-Code auf einzelnen Maschinen untersucht. Welche Architekturmerkmale sichern die Sprache selbst? Wie funktioniert Code Access Control? Welche Rolle spielen Policies, Code und User? Und nicht zuletzt: Wie funktioniert der Kontrollmechanismus beim Aufruf von Methoden? Ein konzeptionelles Framework für die Definition und Durchsetzung von Policies wird vorgestellt und anhand der Java-Mechanismen vertieft.

Der zweite Punkt, Enterprise Security, besteht aus zwei großen Themenbereichen: Zum einen die Verschiebung von Security-Code aus den Applikationen in die Infrastruktur hinein, um Applikationen portabel zu machen. Dies wird am Beispiel JAVA EE und der dortigen Separation von Concerns und Context diskutiert. Das zweite Thema ist die End-to-End-Security: Wie wird ein Request von außerhalb der Firma in Empfang genommen, abgesichert und durch die Infrastruktur zu den Backend-Systemen geleitet? Welche Probleme tauchen hierbei auf und wie sehen die Lösungsmöglichkeiten z. B. für Delegation im JAVA-EE-Umfeld aus?

Der dritte Punkt schließlich, Application Server Security, geht über die im Kap. 4 zur Plattformsicherheit besprochene Absicherung von Servern hinaus. Zwei Themen dominieren auch hier: Die Integration der Application Server Security mit vorhandenen Sicherheitssystemen durch neue APIs sowie Implementationsmöglichkeiten der Delegation von Requests über mehrere Maschinen hinweg. Die Propagation von Subject-Information generell sowie speziell zu Backend-Systemen wird erläutert und Angriffsmöglichkeiten auf Zellen von Applikationsservern werden diskutiert.

Viele der Themen werden anhand praktischer Beispiele vertieft, z. B. wann, wie und wo im Struts-Framework Zugriffe abgesichert werden sollten, welche Feh-

ermöglichkeiten bestehen etc. Exkurse greifen spezielle Themen wie die instanzbasierte Zugriffskontrolle im Spring-Framework auf, oder wie durch Speicherfehler das Typsystem einer Sprache korumpiert werden kann.

In diesem Kapitel diskutieren wir also Sicherheitskonzepte der Java-Plattform, und zwar aus der Sicht der Programmiersprache selbst, aus Sicht einer Enterprise-Architektur, wie sie Enterprise Java Beans und generell JAVA EE bietet, und zuletzt aus der Sicht von Application-Servern, die die Verbindung zur Sicherheitsinfrastruktur herstellen. Wir beginnen mit einer Übersicht der bisherigen Entwicklung der Sicherheitskonzepte im Java-Umfeld.

5.1 Die Evolution der Java-Sicherheit

Die Sprache Java war ursprünglich für embedded-control-Anwendungen, unter Anderem in der Haustechnik, konzipiert. Im Gegensatz zu den meisten anderen Sprachen konnte (und musste) Java-Code also als mobil und daher auch als potenziell fremd und gefährlich eingestuft werden. Die Plattform selbst und ihr Eigentümer mussten vor böartigem Code geschützt werden. Sicherheit war deshalb von Beginn an ein wichtiges Thema bei der Entwicklung von Java.

Das Kernkonzept der Java-Security umfasst Typsicherheit (sie wird durch Prüfungen beim Laden von Code gesichert), als Folge davon auch Speichersicherheit (Typenregeln können nicht durch Low-Level-Zugriffe wie Pointer ausgehebelt werden), und eine relativ allgemeingültige Definition einer abstrakten Maschine für Bytecode – die theoretisch auch andere Sprachen ausführen könnte (wobei sich bei einzelnen Sprachen wie z.B. Smalltalk Probleme auftun). Wichtig für die Sicherheitskonzepte ist der letzte Punkt, denn er schließt aus, dass die Virtual Machine die javaspezifische Sicherheit hart implementiert (alternativ könnte sich eine VM natürlich auch an die jeweilige Sprache dynamisch anpassen). So sind die zentralen Java-Sicherheitsmechanismen in der Java-Runtime-Umgebung untergebracht. Diese Runtime-Umgebung hatte zwar keine Vorstellung (bzw. Repräsentation) von Nutzern, dafür aber ein Konzept von Mandatory Access Control (MAC). Mandatory deshalb, weil Code in sog. *Sandboxes* eingesperrt werden konnte, die genau bestimmten, welche Aktionen der Code innerhalb einer solchen Sandbox ausführen durfte.¹

Waren die ersten dieser Sandboxes als verschiedene Versionen von *Security-Manager*-Klassen programmiert und daher starr in ihrem Verhalten – z.B. die *Applet-Sandbox* – so entstand bald der Wunsch, dieses Verhalten flexibel anpassen zu können, und zwar möglichst ohne Codeänderungen. Schnell wurden die Grenzen eines starren *Securitymanagers* erkannt [WF] und ein granulares Autorisierungssystem für Code eingeführt. Das Autorisierungssystem bestimmte die Rechte von Code anhand seiner Herkunft (die über Signaturen verifiziert werden

¹ MAC bedeutet für uns lediglich, dass bei der Zugriffskontrolle nicht allein Rechte des Benutzers herangezogen werden.

konnte). Das führte zum Begriff der *Security-Policy*, die Erlaubnis (Permissions) und Klassen (Codesource) miteinander kombinierte und so eine Sicherheitsphilosophie ausdrücken konnte. Dadurch war es auch möglich, die Mächtigkeit lokal vorhandener Klassen einzugrenzen, die ansonsten immer als voll privilegiert angesehen wurden. Damit war die Java 2 Security geboren.

Aus dem Firmenumfeld hingegen wurde Druck in Richtung auf weitere Verbesserungen aufgrund einer komplett anderen Sichtweise auf Risiken ausgeübt: Hier sind es die Ressourcen der Firma, die vor den Nutzern geschützt werden müssen. Unterschiedliche Nutzer besitzen unterschiedliche Rechte (Arbeitsvertrag, Position) und entsprechend wichtig ist die Identität eines Nutzers bei der Ausführung von Programmen. Die eingesetzten Programme hingegen werden im Allgemeinen als „gutartig“ angesehen – sind sie doch selbst geschrieben oder sog. COTS (Common Off The Shelf) Produkte.

Beide Sichtweisen – gefährlicher Code vs. gefährliche User – sind auch heute noch sehr weit voneinander entfernt. So sehen wenige Entwickler von Firmensoftware ein, dass es vorteilhaft sein kann, wenn Code an sich in seinen Rechten eingeschränkt wird – unabhängig von den Rechten eines Nutzers. So laufen die meisten Application-Server oder Applikationen *ohne* die codebasierte Java-Security. Dabei wird leider nicht bedacht, dass Code nicht nur bösartig, sondern schlicht auch fehlerhaft sein kann, was bei mehreren Applikationen innerhalb einer VM und potenziell hunderten von parallelen User-Requests schnell zu falschen Ergebnissen führen kann (vgl. das Beispiel in Abb. 5.25).

Das Konzept des gefährlichen, weil mobilen Codes prägte die Security der ersten Java-Versionen. Wenn kein mobiler/gefährlicher Code geladen wurde, musste auch kein Securitymanager geladen werden und sämtlicher Code lief mit den vollen Rechten der lokalen Installation. Dieses Konzept des „Security-Schalters“ erweist sich heute als problematisch, da viele Großprojekte wie Eclipse heute nur mit vollen Privilegien der lokalen Software laufen können.

Die Autorisierung von Aufrufen auf Basis der Herkunft des Codes wurde bald auf identitätsbasierte Autorisierung (Nutzeridentität bzw. Rollen) durch den Java Authentication und Authorization Service (JAAS) ausgedehnt. JAAS hatte zwei große Aufgaben. Zum einen war es eine Java-Version des bekannten Pluggable-Authentication-Mechanism (PAM) aus dem Unix-Bereich. Damit wurden konkrete Authentisierungsmechanismen z. B. über Passwörter oder Zertifikate) aus den Java-Applikationen entfernt und in eine Middleware (eben JAAS) verbannt. Dadurch mussten Applikationen bei Änderung der Authentisierungsmechanismen in einer Firma nicht mehr neu programmiert werden. Zum anderen wurde durch JAAS ein Konzept der Nutzer eingeführt. Der Nutzer – „Subject“ genannt – musste Eigenschaften und Rechte bekommen, die Grundlage für die Zugriffskontrolle waren. So wie die Quelle von Code die Rechte eines Aufrufs bestimmt, so sollte auch die Identität des Nutzers über seine Rechte bestimmen. Während der Zugriffskontrolle mussten beide Rechtebereiche, Code und Nutzer, kombiniert werden.

Als Programmiersprache für das Internet beworben, musste Java auch in Bezug auf die Sicherung von Vertraulichkeit und Integrität erweitert werden: Bibliotheken für SSL-Verbindungen (JSSE) sowie ein Framework für Anbieter kryptografi-

scher Algorithmen wurden integriert. Ständig wurden weitere Sicherheitsmechanismen und APIs eingegliedert, wie z. B. Unterstützung für Kerberos, SASL, GSS-API etc., die sich vor allem auch mit dem Problem der Delegation von Credentials über Rechnergrenzen hinweg beschäftigen. Diese haben wir bereits im ersten Teil des Buches behandelt. Im Laufe der Zeit wanderten viele Komponenten, die zunächst eine Java-Extension darstellten, in den Kernbereich der Java-Plattform.

Die größte Erweiterung der Java-Plattform geschah jedoch parallel dazu mit der Entwicklung eines Enterprise-Komponentenmodells in Form von JAVA EE. Es wurden verschiedene Komponententypen definiert (Clients, Web, Entity Beans) und dafür Laufzeitumgebungen geschaffen, sogenannte Container. Leitidee war vor allem, Code, der von der firmeninternen Infrastruktur abhängig war, aus den Applikationen zu verbannen. Seine Funktionalität wurde von den Containern übernommen. Neben Transaktionen und Persistenz von Objekten war die Sicherheit der dritte Aspekt, der so zum Teil aus den Applikationen verschwand. Erreicht wurde die Abgrenzung der Applikation von der Infrastruktur vor allem durch die Betonung deklarativer anstelle programmatischer Sicherheit. Für die Delegation von Requests über mehrere Komponenten und Server hinweg wurde mit CSiv2 eine Technologie aus dem CORBA-Bereich in JAVA EE übernommen.

Die Application-Server erweisen sich für die Unternehmenssoftware mittlerweile als extrem wichtige Laufzeitumgebungen. Sie sind durch eine ganze Reihe von Interfaces mit Ressource-Managern verbunden und dazu gehören auch solche, die die Authentisierung bzw. Autorisierung unternehmensweit regeln. Je mehr Firmen ihre Services über das Internet anbieten, desto stärker steigt die Belastung der Applikation-Server. Größere Applikationen laufen daher häufig auf Clustern von Application-Servern, deren Verwaltung und Integration mittels Single-Sign-On und Delegation einen Schwerpunkt der Sicherheitsproblematik in solchen Umgebungen darstellt. Abbildung 5.1 zeigt einige der Sicherheitsanforderungen,

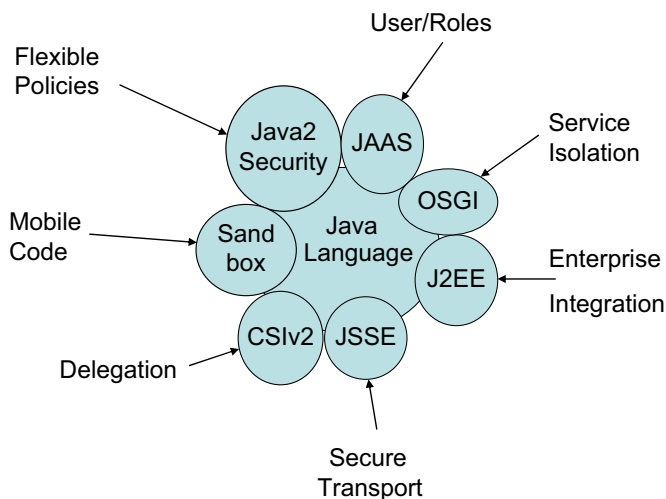


Abb. 5.1 Übersicht der Security-Merkmale von Java

denen sich die Java-Plattform ausgesetzt sah, sowie deren Lösungen in Form von zusätzlichen Frameworks.

Wo also steht die Java-Security heute?

Die mangelnde Isolation von Applikationen innerhalb eines Application-Servers bzw. einer VM wurde als Problem erkannt. Momentan wird daher auf Basis von Classloader-Technologien versucht, Codeteile verschiedener Herkunft oder Kritikalität (z. B. Systemapplikationen) weiter zu isolieren und z. B. die Implementation von Application-Servern vor internen Angriffen zu schützen. Beispiele sind das Servicekonzept der Open-Services-Gateway-Initiative (OSGI) oder die ebenfalls auf OSGI basierende Klassenstruktur von Websphere. Der entscheidende Mechanismus zur Zugriffskontrolle ist hier die Aufteilung der Klassen in unterschiedliche Namespaces und die strikte Überwachung, welche Software auf welche Namespaces zugreifen darf.

Andere Ansätze zur Lösung des Isolationsproblems basieren auf sog. *Object-Capabilities*, d. h. der strikten Kontrolle, welche Komponenten welche Referenzen zur Laufzeit erhalten können. Wir werden beide Ansätze im Detail kennen lernen.

Ein großes Problem ist ein im Allgemeinen unterschätzter Teil von sicherer Software: Die Garantie der Verfügbarkeit oder „Liveness“, das heißt eine Zusicherung, dass Programme sicheren Fortschritt während der Laufzeit machen. Dazu gehört das Management von Ressourcen insgesamt (Speicher, Disk), z. B. durch Quota-Regelungen. Diese Funktionalität ist für zukünftige Java-Versionen geplant. Dadurch rückt man immer näher an die Funktionalität von Betriebssystemen. Schwierig ist das Problem der Ressourcenverwaltung unter Anderem deshalb, weil einfache qualitative Isolationstechniken (*X* darf auf *Y* zugreifen oder nicht) hier nicht mehr ausreichen. Stattdessen muss die Allokation von Ressourcen quantitativ genau überwacht werden.

Es gehört nicht viel Phantasie dazu, um zu erkennen, dass der Aspekt der Verfügbarkeit und Zuverlässigkeit in Zukunft viel stärker als notwendiger Bestandteil eines sicheren Systems wahrgenommen werden wird.

5.2 Sprachbasierte Sicherheit

5.2.1 *Memory Protection und Typsicherheit*

Der Schutz des Speichers, d. h. die Verhinderung eines direkten Zugriffs der Software auf Speicheradressen, stellt einen der Eckpfeiler der Java-Security dar und ist auch einer der Hauptunterschiede zwischen Java und ihren Vorgängersprachen C/C++. Wieso aber ist die Memory Protection eigentlich so wichtig und welchen Zusammenhang hat sie mit Typsicherheit? Und wieso ist das Thema überhaupt für die Entwicklung sicherer Software interessant?

Prozessgrenzen alleine reichen nicht aus, um das Verhalten einer Software festlegen zu können. Prozesse bestimmen, welche Ressourcen eine Software generell verwenden kann, sie bestimmen jedoch nicht, wie innerhalb der Software sicherheitsbezogene Entscheidungen gefällt werden. Das Problem ist also die Zugriffsregelung innerhalb von Software. Dies kann durch externe Policies und die Verwendung eines Reference-Monitors erreicht werden, aber auch durch interne architektonische Maßnahmen wie Objekt-Capabilities oder Namespaces. Auch Mischformen wie z. B. innerhalb der Java-Plattform sind möglich. Allen diesen Mechanismen ist jedoch eines gemeinsam:

Damit innerhalb eines Programms überhaupt Regeln für Zugriffe bzw. deren Berechtigung aufgestellt werden können, braucht es die Gewissheit, dass diese Regeln nicht durch direkten Zugriff auf Speicher umgangen werden können. Memory Protection ist damit die Grundlage für Isolation – einem Kernmuster von sicherer Software. Ohne Isolation keine Abgrenzung von Effekten und damit auch keine Abschätzungen von Sicherheitsrisiken. Der letzte Punkt kann gar nicht stark genug betont werden: Es ist ohne sichere Mechanismen der Isolation nicht möglich, überhaupt Effekte innerhalb einer Software zu bestimmen. Wenn das nicht möglich ist, erübrigt sich auch eine Sicherheitsanalyse. Dieser Punkt wird später theoretisch noch untermauert und am Beispiel einer Browserimplementation dargestellt.

Entweder sorgt die Hardware durch weitere Speicherschlüssel innerhalb von Prozessgrenzen für eine Abgrenzung von Komponenten der Software gegeneinander, oder diese Abgrenzung muss durch Software erreicht werden. Voraussetzung dafür ist jedoch eine *typsichere* Sprache. Sie sorgt dafür, dass die Verhaltensgarantien eines Datentyps nicht umgangen werden können, z. B. durch Pointer-Zugriffe auf den Speicher.

Eine typsichere Sprache braucht ein Kontrollsystem, das die Typsicherheit überwacht. Im Falle von Java geschieht das durch die Virtual Machine, und zwar einmal beim Laden einer Klasse (Abb. 5.2).

Hier sorgt der Bytecode Verifier dafür, dass keine illegalen Instruktionen (Stack-Overflow etc.) im Code vorhanden sind. Zum Zweiten kontrolliert die VM

Java Source to Execution Model

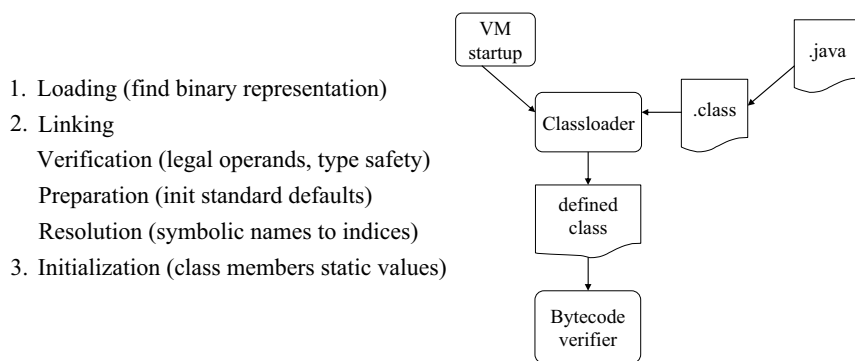


Abb. 5.2 Diagramm des Classloading-Vorgangs

während der Laufzeit die Typkonversionen (Casts) auf Gültigkeit. Im Allgemeinen erlaubt sind Casts von Subtypen hinauf in Richtung des allgemeinen Typs „Objekt“. Sogenannte „Downcasts“, also Konversionen eines Objekts in einen Subtyp, sind nur dann legal, wenn der statische Typ des Objektes tatsächlich ein Subtyp des Konversionsziels oder der gleiche Typ wie das Konversionsziel ist. Eine Möglichkeit, dies zu überprüfen, ist, das zu castende Objekt nach seinem Typ (Klasse) zu fragen. Entspricht sie dem Konversionsziel, ist der Cast in Ordnung, sonst wird eine `ClassCastException` geworfen. Es könnte aber auch sein, dass das zu castende Objekt ein Subtyp des Konversionszieles ist. Deshalb wird vor Abbruch das zu castende Objekt auch nach seiner Superklasse gefragt und auch diese wird wieder mit dem Konversionsziel verglichen, und danach wird nach der nächsten Superklasse gefragt. Entweder entspricht dann eine der Superklassen des zu castenden Objekts dem Konversionsziel, oder es gibt keine weitere Superklasse mehr, wodurch der Downcast als fehlerhaft verweigert wird. Für das Verständnis der Attacke, die wir unten beschreiben, ist wichtig, dass der Cast dynamisch zur Laufzeit geprüft wird, während Zugriffe zur Laufzeit keine Prüfungen mehr beinhalten (die Richtigkeit der Zugriffe wurde ja anhand der Typen beim Laden von der VM geprüft). Die Identität einer Klasse wird wesentlich dadurch vom Classloader bestimmt, dass er bei der geladenen Klasse vermerkt wird. Wird ein und dieselbe Klasse von zwei verschiedenen Classloadern geladen, so handelt es sich für das Java-Runtime um zwei verschiedene Klassen. Erzeugt eine der Klassen ein Objekt und versucht man dieses auf eine Instanz der anderen Klasse zu casten, entsteht eine `ClassCastException` (siehe auch Kap. 5.3.2, Java Code Access Security).

Wir geben nun ein Beispiel für die Sicherheitsprobleme, die entstehen, wenn die Typsicherheit gebrochen wird (vgl. [GA]): Es zeigt, wie sich durch Änderung eines einzigen Bits auf einer Smartcard ein Zustand erzeugen lässt, bei dem zwei Referenzen unterschiedlichen Typs auf eine einzige Speicheradresse zeigen. Damit lässt sich in diese Adresse über einen entsprechenden Zugriff ein Wert schreiben, der dann über den zweiten Zugriff als Methodenadresse interpretiert und ausgeführt werden kann. Die erste Referenz entsteht korrekt nach dem Laden des Programms. Die zweite Referenz entsteht als Cast des Objektes auf einen anderen Typ. Wieso gelingt der Cast? Weil sich durch einen physikalischen Speicherfehler der Wert der zweiten Referenz tatsächlich geändert hat und jetzt auf einen echten, aber unterschiedlichen Typ von Objekt zeigt. Damit kann unter Ausnutzung des oben geschilderten dynamischen Cast-Mechanismus die Referenz auf den tatsächlichen Typ gecastet werden.

Die Attacke beginnt mit einem geschickten Layout von Klassen und deren Verteilung im Speicher einer Smartcard (Abb. 5.3).

Auffallend ist die hohe Zahl von B-Instanzen. Die Hoffnung dahinter ist, dass bei Änderung eines Bits innerhalb einer B-Instanz eine Referenz entsteht, die eigentlich vom Typ A ist, jedoch durch den Speicherfehler tatsächlich auf eine B-Instanz zeigt. Wenn der Bitfehler auftritt, tritt folgende Situation ein: Wir haben eine Referenz vom Typ A, die tatsächlich auf ein Objekt vom Typ B zeigt (Abb. 5.4).

Mit Zugriffen vom Typ A (wir nutzen das Attribut `i` vom Typ `int`) können wir nun Adressen manipulieren. Anschließend casten wir die Referenz auf den

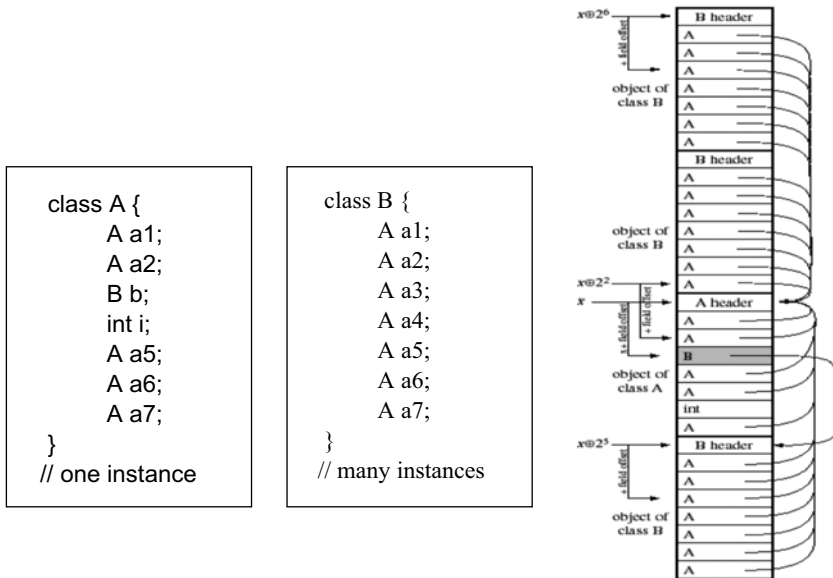


Abb. 5.3 Klassendefinition des Beispielangriffs

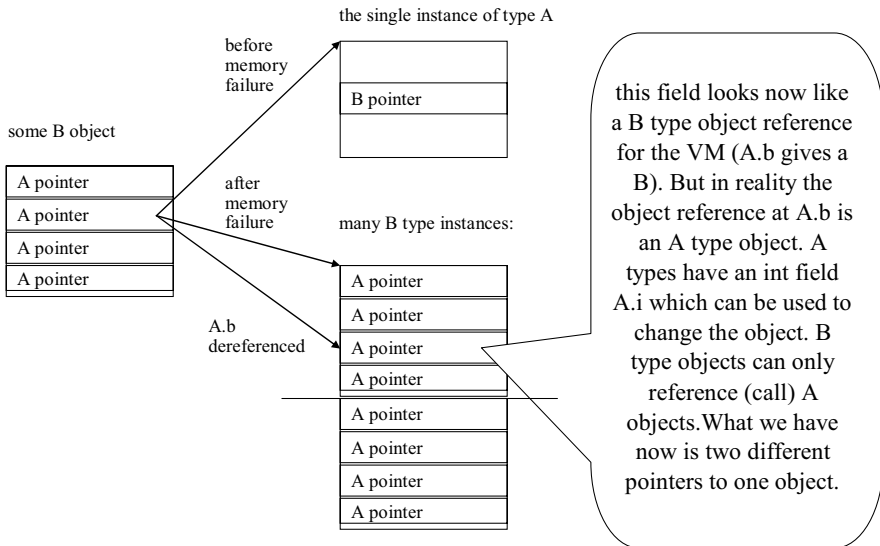
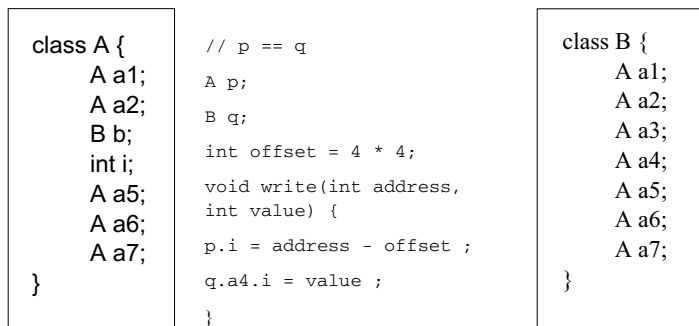


Abb. 5.4 Nach dem Speicherfehler: Referenz vom Typ A zeigt auf Objekt vom Typ B

Typ B (dynamisch wird von der VM geprüft: Bist du ein B?) und greifen dann über B-Methoden zu. Der Code zum Ausnutzen des Speicherfehlers ist in Abb. 5.5 dargestellt.



p.i sets the address of an object of type A. When this A reference is used with a4.i the real access would be i-base of A. So this offset is subtracted first.
(from Govindavajhala...)

Abb. 5.5 Zuweisung einer Speicheradresse an ein Objekt vom Typ A

Ist die Java-Typsicherheit absolut oder ist dieses Merkmal durch Policies anpassbar? Auf den ersten Blick scheint die Antwort ein klares Ja zu sein: Die Typsicherheit ist in der Architektur der Sprache angesiedelt und damit nicht verhandelbar. So gehört z. B. auch die Verwaltung des Stacks nicht in den Bereich von Applikationen. Bei genauerem Hinsehen gibt es jedoch manche Funktionalität in der Java-Umgebung, z. B. die Zugriffskontrolle, die intensiv mit dem Stack arbeitet. Ein weiterer Grenzfall ist die Verwendung sogenannter nativer Methoden: Diese sind typischerweise in C oder C++ implementiert und gestatten daher Zugriff auf den rohen Speicher.

Aufgabe der Access Control in Java ist es dann, den Zugriff auf diese gefährlichen Methoden streng zu begrenzen. Damit haben wir ein interessantes Architekturmuster gefunden: Wenn eine Sicherheitsfunktionalität innerhalb einer Architektur fest verankert ist, kann es aus Gründen der Systemverwaltung bzw. Implementation nötig sein, Teile daraus in Policies auf einer höheren Ebene zu sichern. Dies verringert zwar die Sicherheit des Basissystems, bringt aber mehr Flexibilität.

Hinter dem Muster steckt letztlich die Frage, welche Sicherheitsversprechen auf welcher Ebene der Architektur eingehalten werden sollen. Sollte z. B. die sichere Übertragung von Daten auf der Netzwerkschicht (über IPSec) oder auf der Transportschicht (über SSL) stattfinden? Sollten Dokumente über gesicherte Transportkanäle (über HTTPS) oder durch Signaturen und Verschlüsselung der Dokumente selbst (Web-Services Security) übertragen werden? Wir werden in den weiteren Kapiteln noch mehrfach auf diese Problematik stoßen. Diese Frage existiert statt in der hier besprochenen vertikalen Dimension auch noch in einer horizontalen Dimension: Das berühmte End-to-End-Argument in verteilten Systemen behandelt das Problem, wie ein über mehrere Nodes laufender Request abgesichert werden soll: Soll der Request nur an den Endpunkten der Übertragung geprüft und

gesichert werden? Oder sollen auch die Knoten, die unterwegs beteiligt sind, jede Teilübertragung prüfen und notfalls neu anfordern?

Eine weitere interessante Frage in diesem Zusammenhang ist: Was bedeutet die Typsicherheit von Java in Bezug auf privat deklarierte Felder einer Klasse? Das Laufzeitsystem muss verhindern, dass auf solche Felder von außerhalb der Klasse zugegriffen werden kann. Andererseits ermöglicht es die automatische Serialisierung von Objekten, dass Objekte automatisch gelesen und gespeichert werden können. Also muss doch ein Zugriff möglich sein. Der Zugriff geschieht über Reflektion und ist durch eine sogenannte *Permission* geregelt, geschieht also auf Basis einer Security Policy. Code, der Reflektion für das Auslesen privater Variablen verwenden möchte, muss deshalb eine spezielle Autorisierung erhalten. Ein zusätzlicher Schutz gegen ungewollte Serialisierung besteht darin, dass private Felder als *transient* gekennzeichnet werden können und dann nicht serialisiert werden.

Dennoch – die Sprachsicherheit von Java und die Notwendigkeit von automatischer Serialisierung durch den Einsatz von Reflektion stehen hier im Konflikt miteinander und es zeigt sich, dass Sicherheit von Software nicht allein auf einer Ebene – hier der Sprachebene – entschieden werden kann. Es gibt zu viele unterschiedliche Interessen beim Umgang mit Ressourcen, als dass eine einzige, fundamentale Policy, ausgedrückt in einer Architektur, dem gerecht werden könnte. Eine ähnliche Problematik taucht häufig in Bezug auf Administrative- und Businessrollen auf: Es gibt schlicht gefährliche, aber notwendige Funktionen, die nicht völlig isoliert werden können, da sie von bestimmten Rollen benutzt werden müssen.

Die Absicherung von reflektiven Methoden geschieht also über die Access-Control-Mechanismen von Java und ist demnach im Bereich der Policies angesiedelt. Allerdings erlaubt Java zurzeit nicht alle Möglichkeiten von Meta-Objekt-Protokollen (wie Reflektion) und schränkt daher die Gefahren für die Language Security etwas ein. Einen Überblick über die Mächtigkeit von Meta-Objekt-Protokollen findet man bei [Kicz], der auch die damit verbundene Sicherheitsproblematik diskutiert.

Durch den Begriff der „Policies“ (siehe auch Kap. 5.3) ist ein weiteres Architekturmuster aufgetaucht: Das Verhältnis zwischen Sicherheit, die in einer Architektur ausgedrückt ist, und Sicherheit, die in externen Regelsätzen oder Policies festgehalten ist. Je nach Gewichtung kann mehr oder weniger Funktionalität in die flexiblen Policies verlagert werden – teilweise auf Kosten der Sicherheit in der Basisarchitektur, die dann mächtige generische Mechanismen aufweisen muss. Es stellt sich jedoch die Frage: Wer verwaltet die Konsistenz der externen Policies? Wer öfters Konfigurationsfiles bearbeiten durfte, mit denen die Funktionsweise von Software in nennenswertem Umfang gesteuert werden kann, kennt dieses Problem sehr genau.

Zum Abschluss unserer Diskussion der Typsicherheit in Java noch ein Vergleich mit C++: Dort gibt es ebenfalls das Keyword `private`, jedoch ist dieses Keyword nur für den Compiler relevant. Dieser prüft, ob falsche Zugriffe stattfinden: Nehmen wir z.B. an, dass eine Bibliothek eine Klasse enthält, die ein als `private` deklariertes Feld beinhaltet, in dem ein sicherheitskritischer Status ge-

halten wird. Dieser Status wird durch öffentliche `get()`- und `set()`-Methoden per Zugriffskontrolle geschützt. Die Kontrolle lässt sich aber leicht aushebeln, indem während der Kompilierung durch ein Header File `private` durch `public` ersetzt wird. Da der kompilierte Code ohne weitere Prüfung ausgeführt wird, kann der Client jetzt durch eine andere Klassendefinition direkt auf die Objekte der Klasse der Bibliothek zugreifen. Software in C++ geht daher oft auch einen anderen Weg als die feingranulare Zugriffskontrolle in Java: Die Abgrenzungen sind dort häufig durch Prozessgrenzen gesetzt, oder es kommen Vertrauensmechanismen ins Spiel, die z. B. die Aufrufe von signierten Bibliotheken anhand von Tags überprüfen. Die neue Zugriffskontrolle im Symbian-OS ist ein Beispiel für dieses Vorgehen.

Sicherer, mobiler Code ist natürlich auf so einer Basis nur schwer zu erreichen. Im Kap. 2 über Attacken wurden die Buffer Overflows angesprochen. Auch sie – und sie machen immerhin nach mehreren Schätzungen zwischen 50 und 80% aller momentanen Sicherheitslücken aus – sind die unmittelbare Konsequenz von fehlender Memory Protection. Das Interesse an softwarebasierter Memory Protection und Isolation wird in Zukunft noch steigen. Kleincomputer wie Smartphones und PDAs profitieren massiv von der größeren Geschwindigkeit und den geringeren Hardwarekosten dieser Form der Absicherung.

5.2.2 Erweiterungen und Ableitungen von Klassen

Mit zum Thema Language Security gehört die Sicherheit der gesamten Laufzeitumgebung auch und gerade im Falle von Erweiterungen und Ableitungen. Java ist eine objektorientierte Sprache und gestattet die Definition von neuen Typen durch Ableitung vorhandener. Diese Schnittstelle war seit Einführung des Merkmals „Vererbung“ ein wunder Punkt: Abgeleitete Klassen können in vielen Programmiersprachen direkt auf Felder der Basisklassen zugreifen. Das äußert sich dann häufig als Problem für die Weiterentwicklung von Frameworks oder Bibliotheken, denn Änderungen der Basisklassen machen plötzlich abgeleitete Klassen inkompatibel. Verschärft wird dieses Problem noch, wenn Basisklasse und Ableitung von verschiedenen Firmen erstellt werden. Ganz besonders kritisch wird es im Falle von mobilem Code, denn dann taucht plötzlich ein Sicherheitsproblem auf: Die abgeleitete Klasse wird nachgeladen und kann jetzt auf Interna des installierten Frameworks zugreifen.

Es stellt sich also die Frage: Was ist ein legales API eines Frameworks oder einer Bibliothek? Es gibt zwei entscheidende Forderungen an ein solches API:

- Clients sollten durch Änderungen des Basisframeworks nicht gebrochen werden.
- Clients dürfen durch Ableitungen keinen Zugriff auf sicherheitsrelevante Datenstrukturen oder Methoden erhalten.

Die letzte Forderung ist z. B. im Eclipse Projekt erst relativ spät als wichtig erkannt worden (siehe: [KPH]). Beide Forderungen sind ein schönes Beispiel dafür,

wie eng die Prinzipien guter und sicherer Software zusammenhängen. Wie geschieht nun die Absicherung eines Software-Kerns vor illegalen Zugriffen von Seiten der Clients? Hierfür gibt es Maßnahmen auf unterschiedlichen Ebenen:

Maßnahmen auf der Ebene der Programmiersprache:

- Keine sicherheitskritischen Strukturen und Funktionen außerhalb von Packages zugreifbar machen (Felder als `private` deklarieren, Klassen als `final` deklarieren, Package als `protected` deklarieren).

Kryptografische Maßnahmen:

- Packages durch Signaturen vor nachträglichem Einfügen oder Abändern schützen

Maßnahmen, die einen Zugriffskontrollmechanismus voraussetzen:

- Packages vor Aufruf schützen.
- Klassen nicht nachträglich zu Packages hinzufügen lassen (Sealing).
- Die Sichtbarkeit des Softwarekerns einschränken (ClassLoader-Design zum Schutz vor Reflektion).

Wie viel Vertrauen genießen die in der Architektur der Sprache Java angesiedelten Sicherheitsmechanismen bei den Herstellern? Offensichtlich gibt es gerade im Embedded-Control-Umfeld zwei Lager: Das eine fährt eine reine Java-Politik, z. B. auf Basis von OSGI, das andere hat eher Angst vor der Flexibilität von Java, etwa dass z. B. über native Funktionen Sicherheitslücken entstehen könnten. So sperren viele Handyhersteller gewisse Funktionen ihrer Geräte für den Zugriff aus Java heraus. In C oder C++ programmiert, ist jedoch beispielsweise der Zugriff auf die SIM-Karte etc. kein Problem. Aber ist das wirklich eine sinnvolle Strategie? Letztlich benötigt man eigentlich nur eine Socket-Schnittstelle, um aus der Sprachumgebung ausbrechen zu können. Über ein RPC-Protokoll über Sockets kann dann ein C oder C++-Proxy angesprochen werden, der dann native Aufrufe ans Betriebssystem machen kann. Das Sperren jeglicher Socket-Funktionen hingegen ist wenig sinnvoll, wird aber teilweise insofern gemacht, als die Sockets auf Http-Socket-Typen begrenzt werden. Vielleicht ist es eher die Sorge, dass eine Öffnung der eigenen Plattform (z. B. Multimediaanlagen in Fahrzeugen) den Benutzern erlauben würde, fremde Erweiterungen zu laden – mit unklaren Folgen für die Sicherheit, aber in erster Linie wohl folgenreich für die eigenen Geschäftsmodelle.

Die Sicherheitsfragen im Embedded-Control-Umfeld werden wir später an zwei Beispielen untersuchen: Der Multi-Vendor-Serviceplattform OSGI und den multifunktionalen Class 4 Cardreadern der FINREAD-Architektur. Anhand des letzteren Beispiels werden wir auch kurz die Einschränkungen der Java-Umgebung erläutern, die aus Sicherheitsgründen gerne gemacht werden: Sie bestehen im Wesentlichen aus dem Entfernen von weiteren Paketen aus dem Java-Runtime für Embedded Systems.

5.3 Access Control Policies und ihre Implementation

Der Bereich der Zugriffskontrolle, der außerhalb des Codes flexibel entschieden werden kann, wird im Allgemeinen als *Policy* bezeichnet. Eine Policy verknüpft grob gesagt Subjekte mit Rechten auf Objekte und bildet somit eine Access-Control-Matrix ab. Subjekte können dabei entweder Code oder Nutzer sein, oder auch Kombinationen von beiden, wobei dann entsprechende Prioritäten zu setzen sind (stellen die Rechte des Codes eine Obergrenze des Erlaubten dar oder kann ein Nutzer diese Grenze aufgrund seiner persönlichen Rechte überschreiten?). Mit „Code“ bzw. „User“ als mögliche Besitzer von Rechten sind auch schon die beiden Formen von Access Control Policies in der Java-Plattform genannt.

5.3.1 Grundlegende Aspekte von Policies

Policies müssen zunächst formuliert und anschließend durchgesetzt werden. Im Bereich der Zugriffskontrolle wird dafür häufig das Prinzip der *Interception* eingesetzt: Vor dem Zugriff auf geschützte Ressourcen wird ein Wächter eingefügt, der die Zugangsberechtigung prüft, wie z. B. der Reference Monitor in Abb. 5.6. Wenn die Aktionen von Software durch die Menge der eigenen Rechte begrenzt werden, dann konstituiert dieses Prinzip eine sogenannte *Sandbox*, d. h. einen Sicherheitsbereich, in dem Software eingeschlossen ist und die dort nur die vom Referenzmonitor genehmigten Aktionen ausführen kann.

Dieses Diagramm ist auch gültig für Sandboxes auf Ebene eines Betriebssystems wie z. B. Systrace oder SELinux. Die Unterschiede bestehen hier lediglich in der Granularität der Zugriffskontrolle. Man kann an diesem Bild leicht erkennen, dass der Referenzmonitor grob aus zwei Bereichen besteht: Einem Definitionsbereich (oder auch Entscheidungsbereich) und einem Bereich, der die tatsächliche Kontrolle erzwingt. Diese Einteilung entspricht der Unterscheidung zwischen Autorisierung und Access Control. Autorisierung ist im Wesentlichen die Defini-

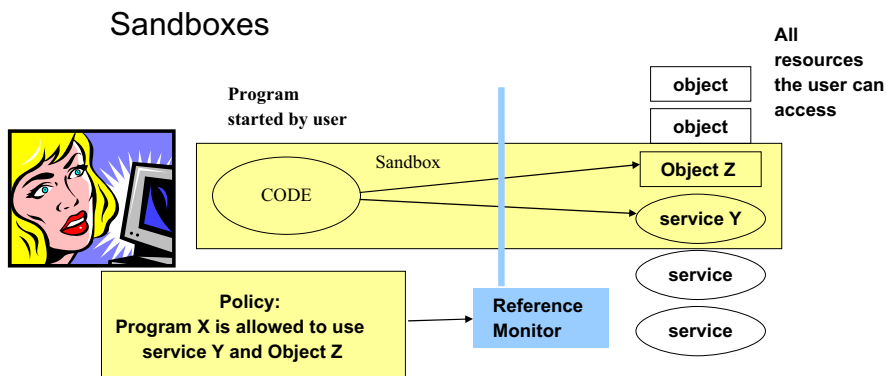


Abb. 5.6 Policies und Referenzmonitor für Sandbox

tion von Rechten und Regeln, die Access Control erzwingt dann die Zugangskontrolle anhand dieser Definitionen.

Die Definitionen können statisch sein – ein fester Satz von Rechten, Rollen etc. sowie Regeln für deren Kombination (z.B. durch Vererbung von Rechten). Ein Compiler kann aus diesen Definitionen dann eine Rechtetabelle pro Subjekt erzeugen. Die Access-Control-Logik prüft dann vor einem Zugriff, ob das von der Ressource verlangte Recht auch in der Rechtetabelle vorhanden ist.

Die Definitionen und Regeln könnten aber auch erst dynamisch in dem Moment ausgewertet werden, wo eine Entscheidung tatsächlich benötigt wird. Beispielsweise könnte eine Business Rule Engine eine Zugriffsentscheidung erstellen. Alternativ – wie im Falle des IBM-Portal-Servers² – könnte eine Logik Vererbungsregeln auf Rollen anwenden und die Entscheidung cachen. Je mehr dynamische Entscheidungen getroffen werden, desto flexibler kann das System neuen Anforderungen der Geschäftslogik angepasst werden, allerdings nimmt die Komplexität der Verwaltung und die Logik der Implementation deutlich zu.

Es gibt noch eine weitere Unterscheidung in Bezug auf die Eigenschaften der Definitionen: Sie können lokal oder auf entfernten Rechnern gehalten werden. Die Access-Control-Logik sollte davon nicht betroffen sein – zumindest nicht in Bezug auf ihre Interfaces. Was sich natürlich nicht verhindern lässt, ist die Tatsache, dass Definitionen auf entfernten Rechnern (z. B. in einem LDAP) Laufzeitprobleme mit sich bringen können, die ihrerseits ein aufwändiges Caching benötigen. Das Treffen der Zugriffsentscheidungen in Real-Time durch eine Rule Engine oder aufwändige Vererbungsalgorithmen ist ebenfalls sehr performanceintensiv. Wir werden auf diese Probleme weiter unten im Kapitel zu JAVA EE und EJB noch eingehen und Lösungsmöglichkeiten vorschlagen (wie z. B. die Propagation von Rechten über signierte Tokens zwischen Rechnern).

Neben der Definition der Regeln, der Entscheidung über den Zugriff und der Durchführung der tatsächlichen Kontrolle, kann ein weiterer Aspekt von Bedeutung sein: Informationen über die Rechte selbst, z. B. ob ein Subjekt (oder Akteur) ein bestimmtes Recht besitzt. Solche Informationen sind beispielsweise nötig, wenn ein User-Interface nur solche Aktionen anzeigen soll, die für einen Akteur tatsächlich zulässig sind. In diesem Fall muss es ein Query-Interface zum Entscheidungssystem geben, um die möglichen Rechte abzufragen. Aktionen, für die ein Akteur keine Rechte besitzt, werden ausgegraut oder – wie im Fall von Portalen – werden nur die Portlets gezeigt, für die auch eine Berechtigung vorliegt. Dieses Query-Interface ist für die Personalisierung von Applikationen unerlässlich. Enterprise Java Beans hat an dieser Stelle einen Designmangel, der zu nicht-portablen Umwegen führt.

Der Access-Control-Punkt kann ebenfalls zwei unterschiedliche Formen annehmen: Er kann programmatisch sein (das heißt, es wird Code eingefügt, der Funktionen zur Zugriffskontrolle aufruft) oder er kann deklarativ (entweder als sog. Annotations im Code selbst oder rein extern als XML-Daten) sein. In diesem

² Siehe Kapitel „Content Management Security“ in Kriha/Schmitz, Internet-Security aus Software-Sicht, Springer Verlag 2008

Fall steuern also Meta-Daten die Zugriffskontrolle. Wir werden die erste Form gleich bei der Diskussion der Java 2 Security kennen lernen und die zweite im Bereich Enterprise Security mit JAVA EE und EJB weiter unten.

Das Konzept von Zugriffskontrolle und Entscheidung bzw. Information über Zugriffsrechte ist so fundamental, dass unter der Führung von OASIS eine Spezifikation dazu entstanden ist, die extensible Access Control Markup Language [XACML]. Sie enthält eine Grammatik zur Beschreibung von Policies und ihren Bestandteilen wie z. B. Regeln. Darüber hinaus definiert sie eine Architektur der Zugriffskontrolle und ihrer Verwaltung. Obwohl sie aus dem Bereich der Web-Services und XML kommt, hat sich die Begrifflichkeit hinter XACML, wie sie in Abb. 5.7 dargestellt ist, mittlerweile auch in anderen Bereichen als praktisch erweisen. Die Beschreibung der Java-Zugriffskontrollen unten folgt der dort verwendeten Terminologie.

Die Spezifikation definiert im Wesentlichen vier Punkte (Authorities) der Zugriffskontrolle:

- Policy Enforcement Point (PEP): Die Stelle, an der eine Zugriffskontrolle erzwungen wird.
- Policy Decision Point (PDP): Die Stelle, wo die Entscheidung über den Zugriff getroffen wird.
- Policy Information Point (PIP): Die Stelle, wo Informationen zu Rechten und Subjekten vorhanden sind.
- Policy Administration Point (PAP): Die Stelle, wo Policies verwaltet werden.

Alle diese Authorities können auch auf entfernten Rechnern sein. Beispielimplementationen findet man bei [OXACML].

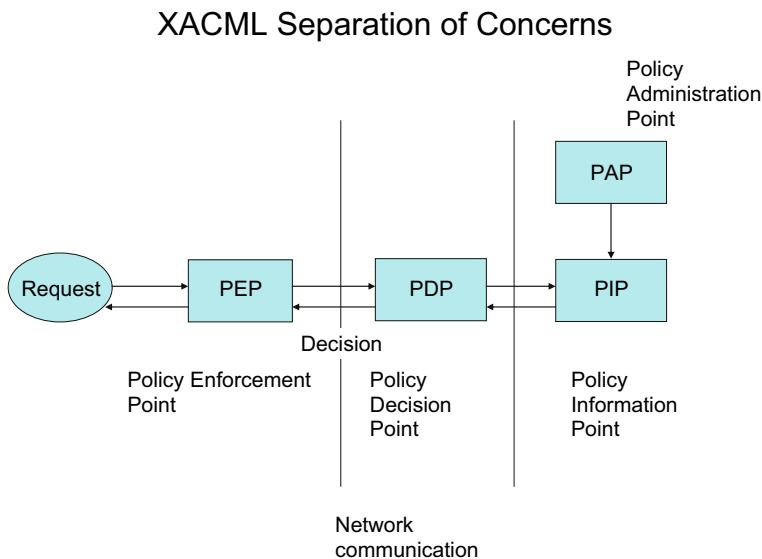


Abb. 5.7 Autoritäten der Zugriffskontrolle

5.3.2 Java Code Access Security

Wir geben hier zunächst einen kurzen Überblick über die Architektur der Java Code Access Security. Im Anschluss folgt eine getrennte Betrachtung der codebasierten wie der userbasierten Zugriffsdefinition und Kontrolle. Dazu beginnen wir mit dem Definitionsbereich der Security-Policies (PIP), gehen über zum Entscheidungsbereich (PDP) und enden mit dem Durchsetzungsbereich (PEP).

Abbildung 5.8 gibt dazu eine Übersicht in grafischer Form.

In Abb. 5.8 sieht man das zentrale Prinzip des Definitionsbereiches: Er besteht aus Identitäten (hier verstanden als *geprüfte* Identität) und Rechten (Permissions). Beide können durch eine Policy kombiniert werden und das Ergebnis ist eine *Protection Domain*: Ein Entscheider, der bestimmen kann, ob eine gewisse Identität ein Recht besitzt.

Die Default-Policy-Implementation in Java geschieht durch ein Textfile, das Definitionen wie diese enthält:

```
grant codeBase "file:${java.ext.dirs}/*" {
    permission java.security.AllPermission;
};
```

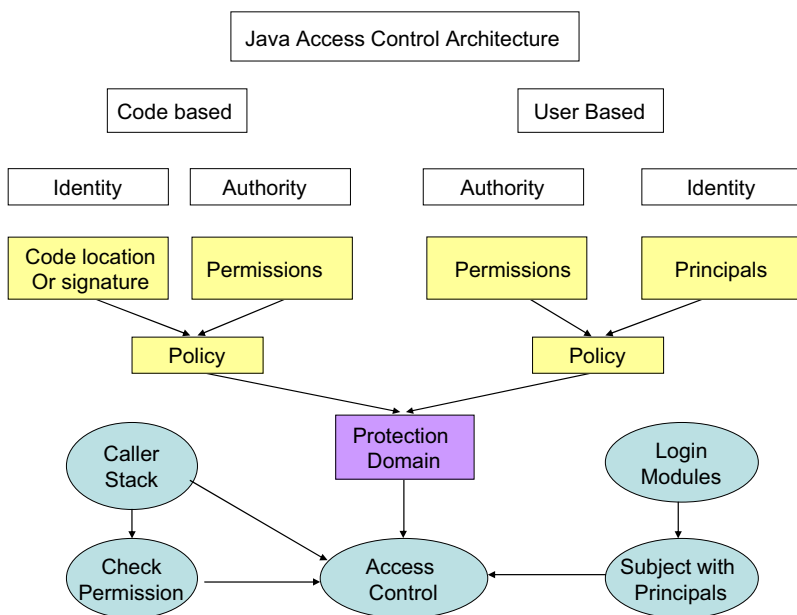


Abb. 5.8 Java Access Control Architektur

Einem Stück Code, das durch seinen Ort identifiziert wird, wird eine Instanz eines Permission-Typs (hier: `AllPermission`) zugeordnet. Permissions können noch genauer qualifiziert werden, wie im folgenden Beispiel:

```
grant {
    permission java.io.FilePermission    "/tmp/*",    "read";
};
```

Jetzt tritt allerdings die Problematik auf, dass geklärt werden muss, ob ein Zugriff auf `/tmp/foo.html` jetzt von dieser Regel abgedeckt ist. Dazu muss die Entscheidungsebene eine „`implies`“-Funktion anbieten, die diese Frage beantwortet. Es ist jederzeit möglich und bei größeren Projekten auch sinnvoll, eine datenbankgestützte Implementation einer Policy zu verwenden. Ted Neward hat in [New01] die Entwicklung einer eigenen Policy-Implementation gezeigt.

Die Struktur der Protection Domain ist in Abb. 5.9 dargestellt.

Es liegt in der Verantwortung eines Secure Classloaders, die Protection Domain korrekt zu initialisieren. Von der korrekten Funktionsweise hängt logischerweise die gesamte Zugriffskontrolle ab. Zudem wird die Identität der Klassen auch durch den verwendeten Classloader bestimmt, der übrigens auch als Classloader pro Klasse vermerkt wird.

Nach der Definition von Rechten und ihren Trägern muss nun der Mechanismus zur Einhaltung der Regeln geklärt werden. Wodurch wird überhaupt die Prüfung eines Zugriffs veranlasst? Getriggert wird die Prüfung über drei Schritte:

- Generelles Anschalten der Java 2 Security durch Definition einer Policy-Instanz:
`java ... -Djava.security.manager -Djava.security.policy = myPolicy`

Protection Domain (one per CodeSource)

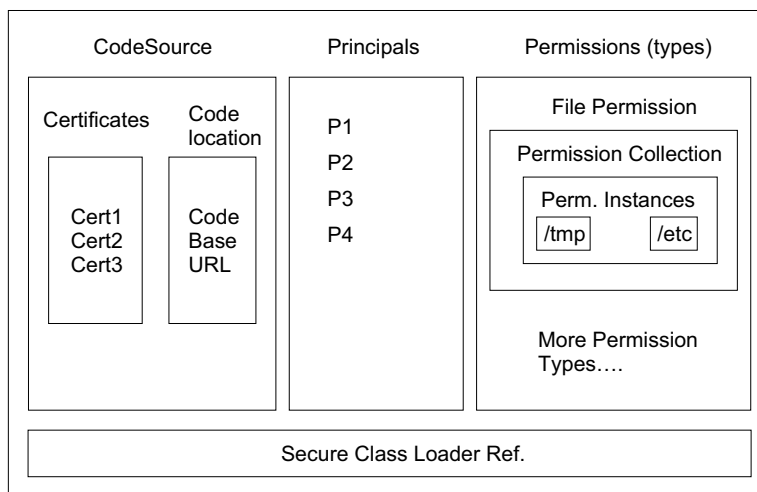


Abb. 5.9 Struktur der Protection Domain

- Einfügen einer Permission-Prüfung im Code der Applikation
- Aufruf des Objektes, das eine Permission-Prüfung verlangt.

Bereits der erste Schritt, das Anschalten der Java 2 Security, kann Überraschungen bringen. Viele Applikationen werden nicht mehr funktionsfähig sein und häufig Security-Exceptions aufwerfen. Der Grund ist, dass im Java-Kern nun sehr viele Zugriffsüberprüfungen aktiviert sind, die ein korrektes Policy-File mit entsprechenden Rechtezuordnungen verlangen. Standardmäßig sind die folgenden Bereiche geschützt (eine genaue Liste der Permissions des Java-Kerns findet sich in der Java-Security-Dokumentation [Gong]):

Files: `canRead()`, `FileInputStream()`, `RandomAccessFile()`, `isDirectory()`, `isFile()`, `length()`, `canWrite()`, `FileOutputStream()`, `mkdir()`, `renameTo()`, `createTempFile()`, `delete()`, `deleteOnExit()`.

Network: `send()`, `receive()`, `getLocalAddress()`, `getHostName()`, `getLocalHost()`, `getAllByName()`.

Virtual Machine: `ClassLoader()`, `loadLibrary()`, `checkPermission()`, `checkLink()`, `checkExit()`.

Threads: `stop()`, `resume()`, `suspend()`, `interrupt()`, `setPriority()`, `setName()`, `setDaemon()`.

System: `getPrintJob()`, `setProperty()`, `getProperty()`, `setDefault()`, `getFont()`, `getEventQueue()`.

Security: `getFields()`, `getMethods()`, `getConstructors()`, `setPublicKey()`, `addCertificate()`.

Der zweite Schritt, das Einfügen von Permission-Überprüfungen, geschieht programmatisch in der Java 2 Security: Wenn ein Objekt einen Zugriff auf die nötige Berechtigung abprüfen will, wird folgender Code eingefügt:

```
fileAccess(String fileName) {
    Object context = null;
    SecurityManager sm = System.getSecurityManager();
    if (sm != null){
        context = sm.getSecurityContext();
    }
    FilePermission p = new FilePermission(fileName, "read");
    sm.checkPermission(p, context);
}
```

Der Code fordert also eine Prüfung programmatisch an, indem er eine Instanz einer Permission erzeugt und zur Prüfung an den `SecurityManager` delegiert. Das Argument `context` ermöglicht es, bei der Prüfung einen anderen als den Sicherheitskontext des momentanen Threads zu verwenden. Dies ist vor allem bei Server-Applikationen nötig, um den Code im Sicherheitskontext eines anderen Benutzers ausführen zu können und nicht im mächtigen Kontext des Servers

selbst. Der dazu verwendete Mechanismus wird im Kap. 5.3.4, userbasierte Policies, genauer geschildert.

Es ist möglich, eigene Permission-Typen zu schreiben. Speziell der Code zur Überprüfung von Input-Variablen wie hier im Beispiel der Name des Files und der gewünschte Zugriffsmodus lässt jedoch viel Raum für Fehler bei der Implementation (Stichwort Input-Validation). Weiter unten wird eine Möglichkeit vorgestellt, auf der Basis von eigenen Permission-Typen zu einer instanzbasierten Zugriffskontrolle zu gelangen.

Im dritten Schritt wird das abgesicherte Objekt aufgerufen. Es bleibt noch der konkrete Mechanismus zu klären, wie eine verlangte Prüfung durchgesetzt wird, also der Policy Enforcement Point. Der PEP in der Java 2 Security wird aus dem Zusammenspiel einer Reihe von Klassen gebildet: `Permission`, `SecurityManager`, `AccessController`, `AccessControlContext` sowie Hilfsklassen zur Kombination von Protection Domains.

Während in den ersten Versionen von Java dieser Zusammenhang als Methode in der Klasse `SecurityManager` jeweils programmiert werden musste (was das kritisierte Konzept der starren Sandbox verursachte), wurde in Java 2 der Zugriff als Hierarchie von typisierten Permission-Klassen modelliert. Die ausufernden Methoden der Form `checkYYY(...)` in der Klasse `SecurityManager` konnten dann durch eine Methode `checkPermission(Permission)` ersetzt werden. Seitdem delegiert der `SecurityManager` lediglich noch den `checkPermission()`-Aufruf an die `AccessController`-Klasse.

Der `AccessController` erstellt eine Momentaufnahme des Call-Stacks, d. h. aller Objekte, die sich während des zu prüfenden Aufrufs auf dem Stack des momentanen Threads befinden (siehe Abb. 5.10) und somit alle zum Aufruf der Prüfung beigetragen haben.

Abbildung 5.11 zeigt den sogenannten *Stack-Walk-Mechanismus* von Java. Jedes dieser Objekte hat über seine Klasse und den Classloader, der diese Klasse ge-

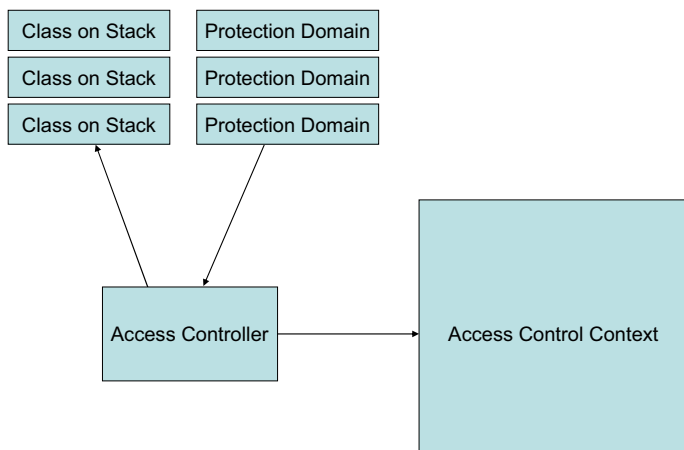


Abb. 5.10 Klassen und Protection Domains

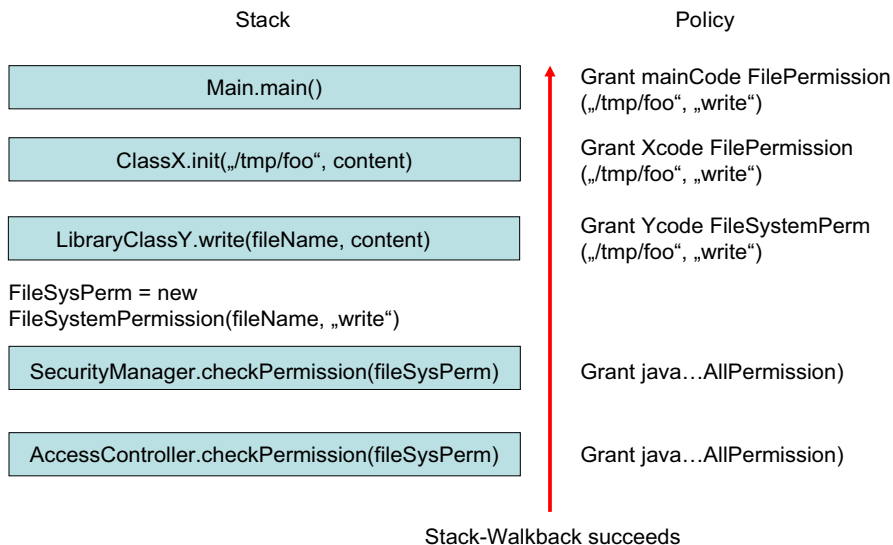


Abb. 5.11 Stack Walk Inspection in Java

laden hat, eine Verbindung zu seiner Protection Domain – das ist eine Definition, die Eigenschaften des Codes wie Ursprung und Signierer mit den dafür definierten Permissions verknüpft – eine Rechtssphäre des Codes sozusagen (s. Abb. 5.9).

Einen ganz ähnlichen Mechanismus, allerdings nicht auf der Granularitätsstufe von Klassen, sondern von ganzen Libraries (DLLs), implementiert die neue Sicherheitsarchitektur des Symbian-OS, nennt ihn jedoch fälschlicherweise „Capabilities“.

Während des Aufbaus des `AccessControlContext` werden die Protection Domains nach den definierten Permissions befragt. Diese Permissions wiederum besitzen eine `implies(Permission)`-Methode, mit der sie prüfen, ob die angeforderte Permission in ihnen enthalten ist. Die gleiche Methode steht auch für Collections von Permissions zur Verfügung. Um auch dynamische Permissions zu ermöglichen – also Änderungen während der Laufzeit – befragt eine Protection Domain zusätzlich eine Instanz eines Policy-Objektes nach geänderten Permissions.

Die Regel für den Zugriffsschutz ist danach sehr einfach: Die verlangte Permission muss im Rechteset sämtlicher auf dem Stack befindlichen Protection Domains vorhanden sein. Damit ist sichergestellt, dass alle beteiligten Objekte die verlangte Permission besitzen. Trifft dies auch nur für eine Klasse nicht zu, wird mit einer `SecurityException` abgebrochen. Abbildung 5.12 zeigt diesen Fall, bei dem nicht alle Beteiligten das gleiche Recht für den Zugriff besitzen. Auf diese Weise wird verhindert, dass ein Objekt beliebige Methoden anderer Objekte aufrufen darf. Gleichzeitig findet jedoch auch eine konzeptionelle „Verflachung“ des Rechtesphäre statt: Alle beteiligten Objekte müssen das gleiche Recht besitzen, auch wenn sie auf sehr verschiedenen Abstraktionsebenen angesiedelt sind.

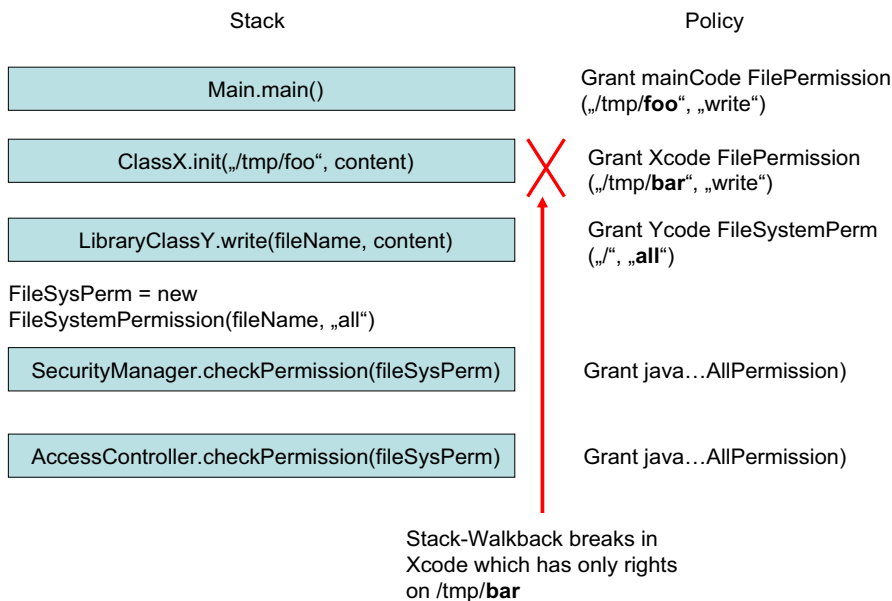


Abb. 5.12 Abbruch des Stack Walk mit Fehler

Dies hat langfristige Auswirkungen auf das Design von Software, die Java 2 Security verwenden möchte. Wir werden dazu gleich Beispiele diskutieren.

Natürlich werden beim Aufbau des `AccessControlContext` diverse Optimierungen durchgeführt, so wird z. B. Buch darüber geführt, ob Klassen aus gleichen Protection Domains auf dem Stack liegen. Diese haben notgedrungen die gleichen Rechte. Dennoch ist der Mechanismus zur Laufzeit performanceintensiv.

5.3.3 Codebasierte Policies

Auch für codebasierte Sicherheits-Policies spielt das Konzept der *Identität* eine große Rolle. Zentrales Element von Java in Bezug auf die Identität von Code ist die Klasse `CodeSource`. Sie vereinigt den Ort des Codes (seine Codebase, normalerweise eine URL) mit einer möglichen Signatur des Codes, z. B. in Form eines signiertes Jar-File. Die Möglichkeiten der Signierung von Jars werden in der Literatur (s. z. B. [Krü]) ausführlich besprochen, sodass es uns hier eher auf die Verwendung dieses Mechanismus ankommt: Wozu dient die Signatur eines Codes? Zunächst einmal lässt sich damit eine Identität erstellen: Man kann also wie bei den sog. „Strongnames“ bei .NET ein Stück Code eindeutig identifizieren. Identität durch Signaturen schließt auch Herkunft mit ein – falls der Signierer bekannt ist. Auf der Herkunft lässt sich wiederum Vertrauen aufbauen. Zusätzlich zur Herkunft des Codes lässt sich auch seine Integrität verifizieren, d. h. der Code wurde seit seiner Signatur nicht mehr verändert. Dies ist natürlich wichtig, da sich

das Vertrauen in den Code aufgrund seiner Herkunft ansonsten nicht rechtfertigen lassen würde: Wurde der Code nämlich nach der Signierung durch eine vertrauenswürdige Quelle noch verändert, dann stammt er eben nicht wirklich aus dieser Quelle.

Basierend auf der Identität lassen sich dann Regeln bezüglich der Verwendung des Codes erstellen, beispielsweise, welcher Code darf welche Packages aufrufen, bzw. welcher Code darf Ergänzungen zu existierenden Packages erstellen. Damit lässt sich eine sichere Basis von Code erzeugen – eine *Trusted Code Base*. Besonders wichtig ist die Identität von Code in dem Moment, wenn sichergestellt werden muss, dass ein kritischer Aufruf einer Systemfunktionalität tatsächlich durch ein bestimmtes Stück Code durchgeführt wurde. Als Beispiel kann der Log-in-Code dienen: Dieser führt für das Betriebssystem die Authentisierung der Nutzer durch und darf auf keinen Fall durch gefälschten Code ersetzt werden. Ein weiteres Beispiel wäre Code, der für eine kritische Aktion unbedingt eine Nutzereingabe benötigt, um sicherzustellen, dass der Code tatsächlich von einem Benutzer und nicht automatisch durch ein Programm ausgeführt wird. Dieser Code wird nach einer Sicherheitsüberprüfung signiert und die kritische Systemfunktion prüft beim Aufruf die Identität des Codes anhand seiner Signatur.

Die Identität im Sinne der Herkunft von Code wird auch bei der Installation von Programmen genutzt, wie etwa im Sicherheitskonzept von Symbian. Das Installationsprogramm trägt nur dann bestimmte Permissions für die neue Applikation ein, wenn die Signatur des Codes geprüft werden kann und es sich beim Signierer um eine vertrauenswürdige Instanz handelt (z. B. weil das Zertifikat des Signierers bereits installiert ist). Wir behandeln die dabei verwendeten Techniken wie Zertifikatsketten ausführlich, wenn wir den FINREAD Cardreader diskutieren. Selbstverständlich wird hierbei nicht die Sicherheit des Code im technischen Sinne geprüft (d. h. der Code wird nicht auf seine Gefährlichkeit hin untersucht), sondern lediglich um eine Sicherheitsannahme, die auf dem Vertrauen in den Signierer des Code beruht.

Bevor wir uns der Frage zuwenden, wie eine Software auf die Schutzmechanismen von codebasierten Policies umgestellt werden kann, wollen wir den von Java für die Ausführung privilegierten Codes zur Verfügung gestellten Mechanismus vorstellen: An der Schnittstelle zwischen Applikationen und System selbst tritt oft der Fall ein, dass der Systemcode auf eine Ressource zugreifen muss, die jedoch dem Applikationscode nicht direkt zugänglich sein soll. Da der Applikationscode daher die entsprechende Permission nicht besitzt, kann ein Aufruf von der Applikation ins System nicht gelingen (s. Abb. 5.13).

Es muss also eine Möglichkeit geben, Systemcode mit Permissions auszustatten, aber die Prüfung an der Grenze zur Applikation zu beenden – eine Form kontrollierter Privilege Elevation also, wie wir sie auch vom „setuid“-Konzept aus Unix kennen.

Genau diese Möglichkeit gestattet die `doPrivileged()`-Methode in Zusammenarbeit mit einer Instanz der Klasse `PrivilegedAction`. Diese Klasse ist mithilfe einer Inner Class implementiert, (siehe „`new PrivilegedAction.`“ in der Abb. 5.14) die den privilegiert auszuführenden Code übernimmt und kapselt.

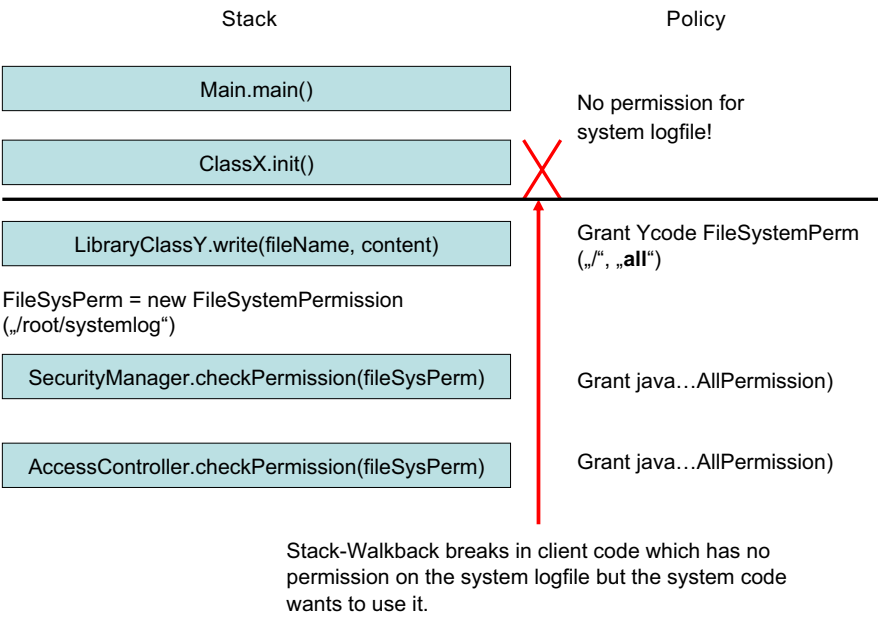
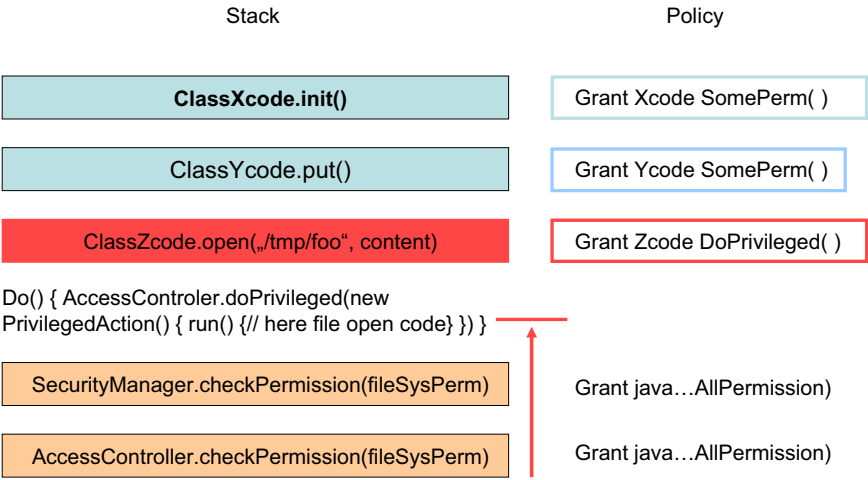


Abb. 5.13 System Code will System Logfile manipulieren, ohne dass Client-Code dazu autorisiert wäre



Stack-Walkback stops at doPrivileged()

Abb. 5.14 Beendigung des Stack Walk durch Privilegierung

Zu Beginn des Codes erfolgt dann automatisch ein Aufruf von `enablePrivilege()`, der mithilfe einer Markierung auf dem Stack die Prüfung der Permissions beendet.

Am Ende des Aufrufs wird automatisch die Privilegierung aufgehoben, wenn der Aufruf den Stack rückwärts wieder abbaut. Hier ein Codebeispiel dazu:

```
libraryMethod() {
// this class does have the right to load a library!
    final String lib = "awt";
/* must be final because java does not support real closures
with persistent init. Environment */

/* anon inner class of type PrivilegedAction stops stack-
walkback without explicit privilege modes in the code */
    AccessController.doPrivileged(
        new PrivilegedAction() {
            public Object run() {
                // will cause a checkPermission which callers
                // of randommethod do NOT have in their policy
                System.loadLibrary(lib);
                return null;
            }
        });
}
```

Derselbe Mechanismus wird ebenfalls für userbasierte Policies angewendet, um Impersonation zu erreichen, also das Ausführen von Code mit den Rechten eines Aufrufers statt z. B. mit den Rechten eines Servers.

Da der Stack-Walk-Access-Control-Mechanismus auch im Falle von Callbacks aus einem Systemlevel zurück in eine Applikation funktioniert, tritt ein weiterer problematischer Effekt auf: Alle Objekte, die am Callback beteiligt sind (also auch die Systemklassen), müssen die entsprechende Permission aufweisen (s. Abb. 5.15). Aber woher sollte der Systemcode wissen, welche Permissions von einem Callback-Objekt verwendet werden sollen? Typischerweise wird daher dieser Code die Allpermission im Security Policy File zugesprochen bekommen.

Der bei der Stack Inspection verwendete Algorithmus nimmt den Durchschnitt der in allen Protection Domains vorhandenen Rechte und verwendet sie zur Prüfung des Zugriffs. Das heißt, auch wenn ein Bibliothekscode, der das Recht zu `doPrivileged()` besitzt, einen Callback in normalen Applikationscode macht, erhält dadurch der Applikationscode keineswegs erweiterte Rechte.

Der `doPrivileged()`-Mechanismus ist aber noch aus einem anderen Grund sehr interessant: Duale Methoden, die globalen State verändern, sind bekanntermaßen sehr gefährlich, da sie immer paarweise auftreten müssen, was aber sehr schwierig zu erreichen ist, da häufig der zweite Aufruf, der den ursprünglichen Zustand wieder herstellen soll, vergessen wird. Ob beim Setzen von Interrupt-Sperren mittels `get/setInt`-Funktionen oder bei der Allokation von Speicher mittels `malloc/free` oder der Allokation von JDBC Statements und Results: Wenn der zweite Aufruf nicht erfolgt, entstehen Lücken im System. Noch kriti-

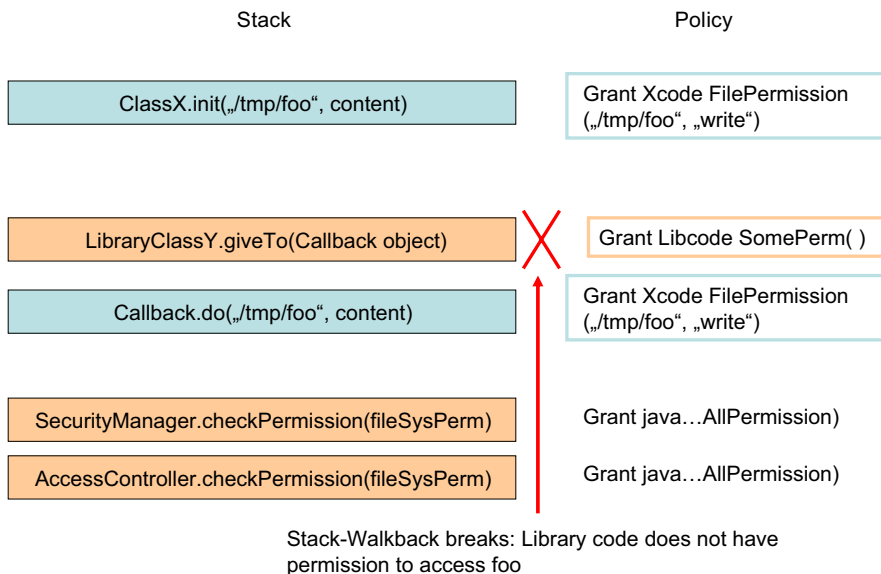


Abb. 5.15 Unterbrechung des Stack Walk im Fall eines Callback

schon ist natürlich das Setzen globaler Privilegienmodi. Wenn hier nicht garantiert wird, dass am Ende einer Funktion bzw. einer Methode der ursprüngliche Zustand wieder hergestellt ist, dann entstehen große Sicherheitslücken im Code.

Die Java Access Control erreicht dies durch ein Sprachkonstrukt, das den sogenannten *Closures* in funktionalen Sprachen sehr nahe kommt (das Prinzip der Closures wird später noch detailliert behandelt). Java-Entwickler, die selbst Konzepte wie Higher Order Functions oder Closures ausprobieren wollen, seien auf die Apache-Commons-Bibliotheken verwiesen, die entsprechende Pakete enthalten.

5.3.4 Userbasierte Policies mit JAAS

Wer bereits einmal mit codebasierter Access Control gearbeitet hat, der hat erfahren, dass die Bedeutung der Identität des Ausführenden dadurch meist verringert wird: Wenn Code durch eine feingranulare Sandbox eingeschränkt ist, dann kann er meist auch durch verschiedene Benutzer ausgeführt werden.

Allerdings gibt es sehr viele Fälle, in denen die Identität eines Ausführenden oder Aufrufers für die Businesslogik der Applikation eine zentrale Bedeutung besitzt. So ist es bei Banktransaktionen natürlich wichtig zu wissen, wer sie veranlasst hat, damit die richtigen Konten geändert werden. Bei diesem Beispiel Konten wird aber auch deutlich, dass es zwei Formen von Identität gibt: Eine passive Form (das wäre etwa der Empfänger einer Geldüberweisung) und eine aktive, also der momentan tätige Überweiser, der die Transaktion veranlasst. Diese aktive

Identität wird meist *Principal* genannt und der Prozess ihrer Erzeugung (also die Verifikation der Identität) *Authentisierung*. Im Anschluss an die Erzeugung eines Principals werden typischerweise gewisse Rechte mit dieser Identität verknüpft – ein Vorgang, der *Authorisierung* genannt wird. Diese Rechte werden dann beim Prozess der Access Control ausgewertet.

Auf Probleme wie z. B. die Frage, ob Code eine Kontomanipulation auch ohne eine angemeldete Identität (also ohne einen Principal) durchführen darf, werden wir nach der Vorstellung der Java-Authentisierungs- und Authorisierungsservices (JAAS) eingehen. Vorab eine wichtige Unterscheidung: Wir betrachten hier Authentisierung und Authorisierung eher aus der Perspektive einer Standalone-Applikation, also im Stil von Java2SE. In Kap. 6, das sich mit Enterprise Java beschäftigt, betrachten wir Authentisierung und Authorisierung aus der Perspektive der Integration von Authentisierung und Authorisierung in Enterprise-Sicherheitsarchitekturen und die dazugehörigen Mechanismen wie deklarative Sicherheit, Delegation von Identität durch Interceptoren, etc. Dort gilt als generelle Regel, dass Applikationen keine initiale Authentisierung oder Authorisierung vornehmen sollen, da dies von der Enterprise-Infrastruktur übernommen wird.

Die Erstellung einer geprüften Identität (Principal)

Die Verifikation einer Identität kann auf die verschiedensten Weisen erfolgen – durch Passwörter, Zertifikate, biometrische Merkmale, etc. Damit ist bereits das erste Softwareproblem genannt: Wenn Applikationen die Authentisierung selber übernehmen, dann müssen sie bei jeder Änderung der Authentisierungsmechanismen angepasst werden. Um dies zu vermeiden, hat SUN für das Solaris-Betriebssystem die sogenannten Pluggable Authentication Modules (PAM) eingeführt – ein Framework, das die verschiedenen Authentisierungsmechanismen kapselt (s. [SL]).

Für Java wurde dieses Framework unter dem Namen JAAS konzeptuell übernommen und angepasst. Wir werden hier nur auf die groben Konzepte eingehen, da die genauen Mechanismen unter [JAAS] detailliert beschrieben sind. Das Grundkonzept eines Authentisierungsframeworks muss die folgenden Ansprüche befriedigen:

- Neue Mechanismen zur Authentisierung müssen ohne Änderung des Applikationscodes eingeführt werden können.
- Es müssen verschiedene Mechanismen gemeinsam für eine Prüfung der Identität herangezogen werden können (z. B. PIN und TAN).
- Es müssen die verschiedensten Datenquellen zur Prüfung der Identität herangezogen werden können (Datenbanken, LDAP, Files).
- Es müssen logische Bedingungen für das Zustandekommen einer korrekten Authentisierung angegeben werden können (z. B. ein Passwort reicht nur aus, wenn zusätzlich eine TAN gegeben wird, ein Fingerabdruck hingegen reicht allein).
- Die Qualität der Authentisierung sollte im Ergebnis festgehalten werden können.

- Es sollte die Möglichkeit geben, weitere Attribute der Authentisierung wie etwa den Ort im Ergebnis festzuhalten.
- Erst nach Befragung aller beteiligten Mechanismen darf eine Authentisierungsentscheidung getroffen werden. Erst dann dürfen weitere Credentials (d. h. Attribute, die zu einem Principal gehören) eingetragen werden.

JAAS leistet die meisten dieser Anforderungen durch ein Framework, bestehend aus `LoginContext`, `LoginModule(s)`, `Callbacks` und `Callbackhandler` sowie `Subject`-, `Principal`- und `Credential`-Klassen (s. Abb. 5.16). Gesteuert wird die Authentisierung durch eine Klasse `Configuration`, über die die nötigen Authentisierungsmechanismen als `LoginModules` konfiguriert werden.

Neben den Loginmodulen im Login Config File müssen im JAAS Policy File auch die Principals sowie deren Permissions definiert werden: Man sieht, dass die Syntax des originalen `grant` lediglich mit dem Keyword `principal`, dem jeweiligen Typ des Principal (z. B. `KerberosPrincipal`) und einer Instanzvariablen (z. B. `admin` oder `user`) erweitert wurde. Hier ein Code-Beispiel für eine principalbasierte Permission:

```
// if the code comes from "foo.com" and is running as Duke",
// grant it read/write to all files in /tmp.
grant codeBase "foo.com", principal foo.com.Principal"User" {
permission java.io.FilePermission "/tmp/*", "read,write";
};
```

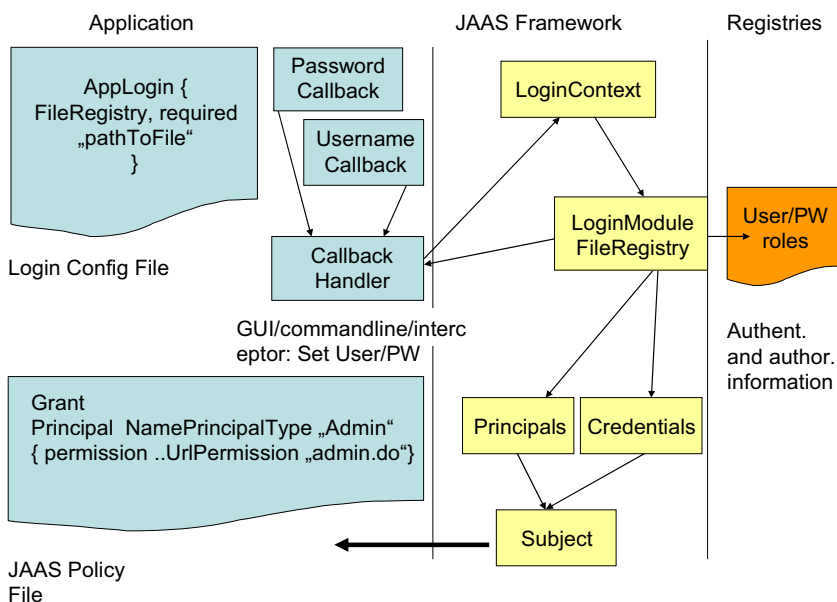
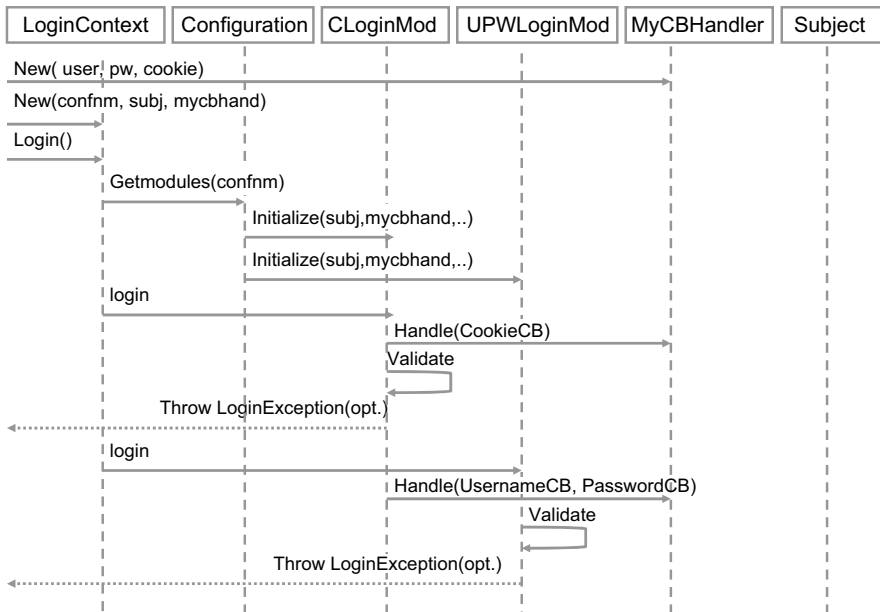


Abb. 5.16 Zusammenspiel von Applikation und JAAS-Framework bei der Authentisierung



User information is processed and validated: module combinations for voting are: *required*, *requisite*, *sufficient* and *optional*

Abb. 5.17 JAAS-Initialisierungs- und Verarbeitungsphase bei der Authentisierung eines Users

Die initiale Authentisierung erfolgt durch einen JAAS-Log-in (s. auch Abb. 5.19), der sich im Sequence-Diagramm in Abb. 5.17 darstellt.

Zunächst wird ein `LoginContext`-Objekt instanziiert und mit einem Konfigurationsfile sowie einem sog. `CallbackHandler`-Objekt versorgt. Das Config-File beschreibt die benötigten Loginmodule (also die Authentisierungsmechanismen und deren Backends) sowie die Logik, mit der die Entscheidungen der Module verknüpft werden sollen.

Anschließend ruft die Applikation die `login()`-Methode vom `LoginContext` auf.

Zum Füllen des `Subject` ruft der `LoginContext` die `login()`-Methode aller Loginmodule der Reihe nach auf. Über den `Callbackhandler`, der den Modulen vorher bereits durch eine `init()`-Methode übergeben wurde, können die Module die Authentisierungscredentials der externen Identität (z. B. `UserID` und `Passwort`) erhalten.

Wenn alle Module ihre `login()`-Methode erfolgreich beendet haben, ruft der `LoginContext` die `commit()`-Methode aller Module auf – anderenfalls die `abort()`-Methode. Beim `commit()` trägt jedes Modul seinen `Principal` sowie die zugehörigen `Credentials` ein, bei `abort()` werden diese entfernt. Entfernt werden sie auch bei `logout()`. Das Sequenzdiagramm der Abb. 5.18 zeigt die entsprechende Abstimmungsmechanik. Dieses Vorgehen stellt sicher, dass die „Meinungen“ aller Module berücksichtigt werden. Dieses Muster taucht auch in anderen Authentisierungsframeworks wie z. B. in `Spring's Acegi` auf, dort unter dem Namen „`Providerpattern`“. Mit `getSubject()` erhält die Applikation –

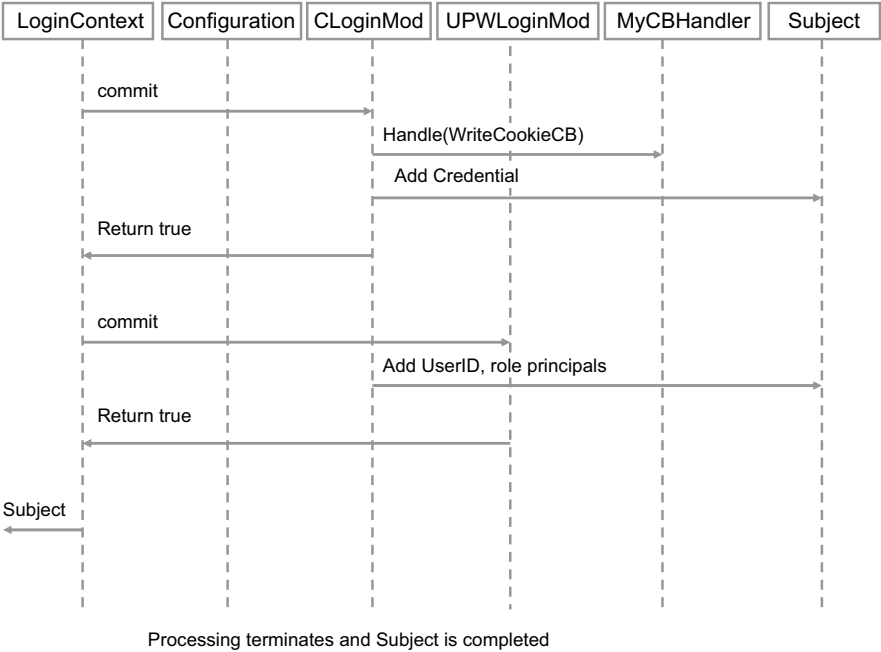


Abb. 5.18 JAAS-Abstimmungsphase bei der Authentisierung eines Users

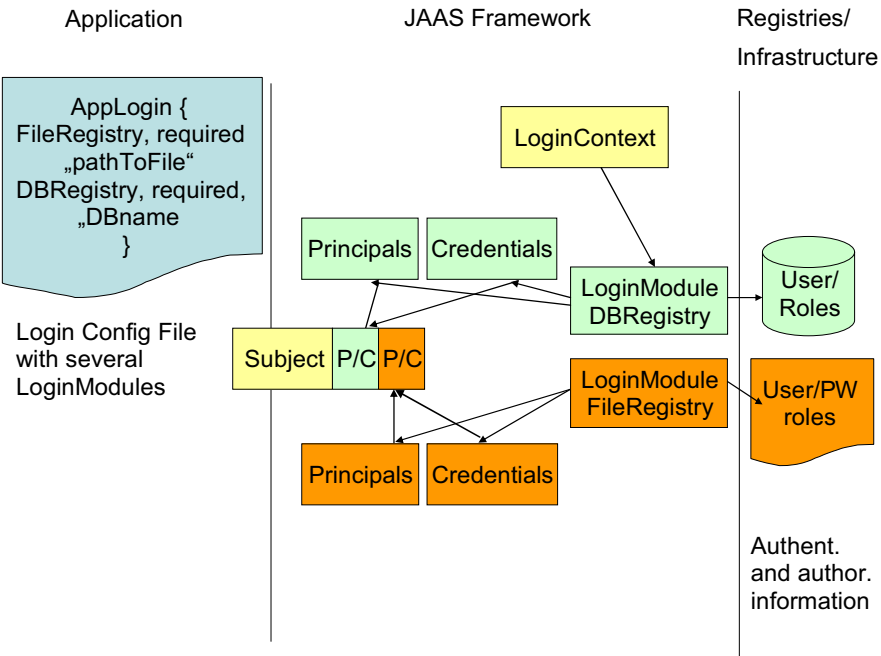


Abb. 5.19 Initiale Authentisierung via JAAS-Log-in

falls die Authentisierung erfolgreich war – ein Subject-Objekt mit eingetragenen Principals und Credentials einer externen Identität zurück.

Der folgende Code zeigt die nötigen Schritte, um eine externe Identität zu authentisieren:

```

CallbackHandler handler = new MyHandler();
LoginContext lc = new LoginContext("some-config", handler);
try {
    lc.login();
    Subject subject = lc.getSubject();
} catch(LoginException e) {
    System.out.println("authentication failure");
    e.printStackTrace();
}

// Perform work as authenticated Subject
// ...

// Scope of work complete, logout to remove authentication info
try {
    lc.logout();
} catch(LoginException e) {
    System.out.println("logout failed");
    e.printStackTrace();
}

// A sample MyHandler class
class MyHandler
    implements CallbackHandler
{
    public void handle(Callback[] callbacks) throws
        IOException, UnsupportedCallbackException
    {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof NameCallback) {
                NameCallback nc = (NameCallback)callbacks[i];
                nc.setName(username);
            } else if (callbacks[i] instanceof PasswordCallback) {
                PasswordCallback pc = (PasswordCallback)callbacks[i];
                pc.setPassword(password);
            } else {
                throw new UnsupportedCallbackException(callbacks[i],
                    "Unrecognized Callback");
            }
        }
    }
}

```

Das Ergebnis der Authentisierung ist im positiven Fall eine Instanz der Klasse `Subject`. Dabei handelt es sich um einen Container, der die verschiedenen Principals, die durch die Anwendung mehrerer `LoginModules` entstanden sind, beinhaltet. Natürlich sind die Einträge nachträglich nicht mehr vom Applikationscode zu ändern, sonst wären beliebige Manipulationen möglich. Wichtig ist, dass die Principals häufig Rollen sind und dass ihnen beliebige Credentials zugeordnet sein können (Passwörter, Token, etc.). Diese Credentials entsprechen im Falle der Rollen Authorisierungen der Identität zum Zugriff auf bestimmte Ressourcen.

Zugriffskontrolle auf Basis eines Subjects

Wozu ist diese `Subject`-Instanz verwendbar? Im Wesentlichen kann damit auf zwei Weisen eine Zugriffskontrolle durchgeführt werden: programmatisch oder impersonifizierend.

Bei der programmatischen Zugriffskontrolle kann im Applikationscode an beliebigen Stellen auf Attribute des `Subject` zugegriffen und daraus eine Zugriffsentscheidung erzeugt werden, z. B.: Hat ein `Subject` eine bestimmte Rolle, dann erlaube eine bestimmte Aktion. Das `Subject` kann auch für Logging und Auditing verwendet werden, um beispielsweise nutzerspezifische Abrechnungen zu erstellen. Dabei stellt sich die Frage, welchen Principal der Code zur Zugriffskontrolle verwenden soll? Da es für jeden Principal eine spezielle Klasse gibt, kann dies bei der Abfrage der Principals des `Subjects` angegeben werden: Der Befehl

```
set principals =  
subject.getPublicPrincipals(UsernamePrincipal.class)
```

gibt den `UsernamePrincipal` zurück, der vom entsprechenden `LoginModule` eingetragen wurde. Leider wurde bei der Integration von JAAS mit JAVA EE versäumt, hier eine klare Standardisierung in Bezug auf die Principals vorzunehmen, was wiederum zu herstellerspezifischen Maßnahmen geführt hat (siehe auch Kap. 7 zur Application Server Security).

Diese Verwendung des `Subject` unterliegt dem Goodwill der Entwickler, die diese Abfragen programmieren müssen (und dies eben auch vergessen können). Das entscheidende Kriterium ist hier: Der Code, der diese Abfragen durchführt, läuft nicht mit den Rechten des `Subjects`, das durch JAAS authentisiert wurde (im Serverfall entspricht das dem Client, der sich angemeldet hat), sondern mit einer anderen Identität und damit anderen Rechten. Es ist deshalb wesentlich besser, den Applikationscode zur Laufzeit mit dem Zugriffskontext des authentisierten und autorisierten Nutzers auszuführen. Dazu bietet JAAS die zwei Methoden `Subject.doAs()` und `Subject.doAsPrivileged()` an. Beide

nehmen als Argument eine Instanz der bereits bekannten `PrivilegedAction`-Klasse:

```
/* performs permission checks both on the rights of mySubject
as well as the rights of the Subject calling the doAs() me-
thod.*/

Subject.doAs(mySubject, action);

// performs permission only on the rights of mySubject

Subject.doAsPrivileged(mySubject, action, null);

/* performs permission only on the rights of some access
control context */

Subject.doAsPrivileged(mySubject, action,
accessControlContext);
```

Bei `Subject.doAs()` wird ein Schnappschuss des existierenden `AccessControlContexts` (also vor der Ausführung der `PrivilegedAction`) genommen und dann ein neuer `AccessControlContext` erzeugt, der die speziellen Rechte des neuen `Subjects` beinhaltet. Wenn jetzt in Folge der Ausführung der Aktion, also des `Applicationscodes`, der für den Client ausgeführt werden soll, eine `checkPermission()` verlangt wird, dann wird beim `Stack.Walkback` der `AccessControlContext` des `Subjects` und der vorher existierende herangezogen. Der Effekt ist ähnlich bei der codebasierten Zugriffskontrolle, wenn kein `doPrivileged()` innerhalb einer Bibliothek verwendet wird: Es werden die `Permissions` der Bibliothek und auch die des Aufrufers geprüft. Damit die Aktion durchführbar ist, müssen die `Permissions` des Client `Subjects` eine Teilmenge der vorher gültigen `Permissions` sein.

Analog zur Verwendung von `doPrivileged()`, kann mit `Subject.doAsPrivileged()` bei der Zugriffsentscheidung nur der `AccessControlContext` des neuen `Subjects` herangezogen werden, um die Prüfung des Aufrufers zu umgehen. In diesem Fall spricht man von *Impersonifizierung*: Der Applikationscode wird tatsächlich mit den Rechten des `Subjects` aufgeführt und nicht mit den Rechten z.B. einer Serveridentität. Abbildung 5.20 zeigt die beiden Methoden `doAs()` bzw. `doAsPrivileged()` und ihre Auswirkungen auf die Prüfung der `Permissions` noch einmal im Vergleich.

Die Integration der JAAS-Authorisierung in die Java-Standalone und Enterprise-Plattformen verlief nicht ganz ohne Probleme. Beispielsweise wurde bei der Einführung des Mechanismus zur Impersonifikation die Anpassung der bereits vorhandenen `AccessController.doPrivileged()`-Methode vergessen. Sie hätte das `Subject` als weiteres Argument benötigt, das jetzt leider abgeschnitten

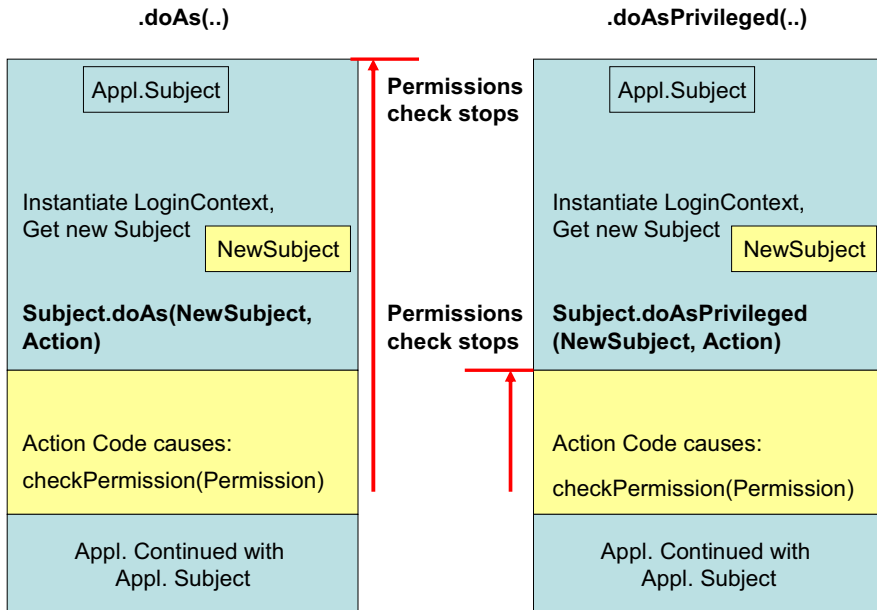


Abb. 5.20 `doAs()` versus `doAsPrivileged()`

wird, sobald in einer `PrivilegedAction` ein weiterer `doPrivileged (Action)`-Call erfolgt:

```
Subject.doAs(subject, new PrivilegedAction()
{
    public Object run()
    {
        // Subject is associated with the current thread
        java.security.AccessController.doPrivileged(new
PrivilegedAction())
        {
            // NOTE: no Subject in interface!!
            public Object run()
            {
                // Subject cut off from the current thread
                return null;
            }
        });
        // Subject again associated with the current thread
        return null;
    }
});
```

Die Hersteller von Application-Servern behelfen sich mit proprietären Hilfsklassen für `Subject`, die in der Lage sind, das `Subject` bzw. dessen Credentials

über verschiedene `doPrivileged()`-Aufrufe hinweg fließen zu lassen. Die Integrationsprobleme mit der Enterprise-Plattform werden in Kap. 7 angesprochen.

Jetzt stellen sich einige wichtige Fragen:

- Können die Rechte des neuen Subject die Rechte eines vorherigen übersteigen? Das heißt, kann eine Rechteerweiterung gegenüber einer Serveridentität stattfinden?
- Können die Rechte des neuen Subjects die Rechte der Betriebssystemidentität eines Servers übersteigen? Kann jetzt also beispielsweise auf Files zugegriffen werden, die der laufende Server sonst nicht manipulieren könnte?
- Sind die Rechte des neuen Subjects durch die Rechte des Codes bei Verwendung von Code Access Security (Sandbox) begrenzt, oder können die Rechte des Codes sogar durch die Rechte des Subjects erweitert werden?
- Macht eine evtl. Erweiterung der Rechte über die Code-Rechte hinaus überhaupt Sinn?
- Welche Rechte sind für `Subject.doAsPrivileged()` nötig?

Die Virtual Machine, die eine Java-Applikation bzw. einen Java-Server ausführt, läuft unter einer gewissen Betriebssystemidentität, die Zugriffsrechte z. B. auf lokale Files dieser Identität beinhaltet. Diese Identität beinhaltet nicht notwendigerweise – und oft sogar ganz bewusst nicht – die Möglichkeit, auf eine entfernte Datenbank zuzugreifen. Die dafür nötigen Credentials sind der Betriebssystemidentität nicht bekannt (im einfachsten Fall existiert eine UserID/Passwort-Kombination für diese Datenbank in einem Konfigurationsfile der Java-Applikation).

Abbildung 5.21 zeigt auch, dass die Rechte des Betriebssystemnutzers zusätzlich durch Java-2-Security-Maßnahmen eingeschränkt werden können. „Server-

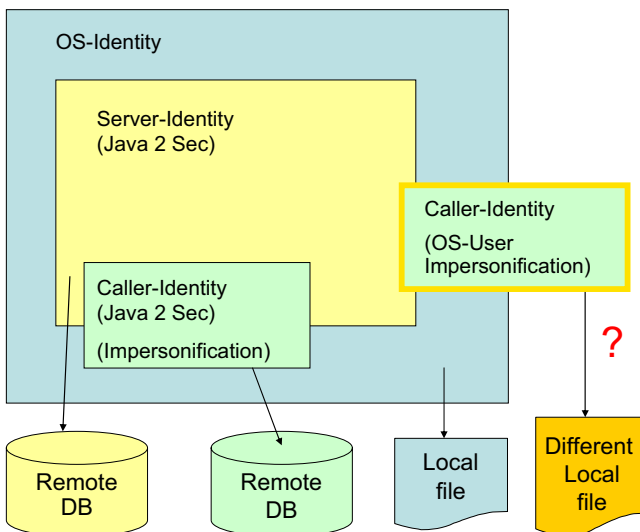


Abb. 5.21 Mögliche Zugriffsrechte bei der Impersonifikation

Identity“ und „Caller-Identity“ sind durch Grant-Statements in Policy-Files der VM in ihren Rechten bestimmt. Streng genommen findet dabei die erste Impersonifikation schon mit der Server-Identity statt: Der Server-Code läuft statt der originalen Betriebssystemidentität mit der der Serveridentität.

Offensichtlich kann also durch `Grant Principal`-Statements die Rechtenmenge vergrößert werden, d. h. eine bestimmte Nutzeridentität kann die Rechte von Code vergrößern.

Jetzt zur entscheidenden Frage: Kann eine der Java-2-Identitäten auf ein File eines anderen Betriebssystemnutzers zugreifen, d. h. außerhalb des Rechtebereichs der Virtual Machine agieren? Ein typischer Fall wäre, wenn die Caller-Identität grundsätzlich auch dem Betriebssystem als Betriebssystemidentität bekannt wäre. Dann wäre nur noch ein Mechanismus nötig, um die Caller-Identität in eine entsprechende Betriebssystemidentität zu verwandeln. Während .NET auf der Basis von Windows Security diese Funktionalität erlaubt, ist sie im Java-Standard nicht enthalten. Proprietäre Erweiterungen in Application-Servern erlauben dies jedoch häufig.

Der Zugriff von Seiten der Server- bzw. der Caller-Identität kann natürlich durch Java 2 Security kontrolliert werden, d. h. die Erlaubnis zum Aufruf selbst steht unter der Kontrolle der Java 2 Security. Die dafür nötigen Credentials zur Anmeldung an der Datenbank könnten wiederum unter den Principals des Subjects enthalten sein und von dort zur Etablierung einer Verbindung zur Datenbank entnommen werden. In Kap. 6 werden wir untersuchen, wie weit JAVA EE hier eine Standardisierung vornimmt.

5.3.5 *Alternative Methoden der Zugriffskontrolle*

Java verwendet hauptsächlich klassische Formen der Zugriffskontrolle durch Policies, die auf Access-Control-Matrizes abgebildet werden. Daneben existieren jedoch auch andere Formen der Zugriffskontrolle innerhalb der Java-Plattform, z. B. die Isolation von Komponenten durch Namespaces oder die Kontrolle von Zugriffen durch die Verteilung von Referenzen als Capabilities. Beide Mechanismen werden wir in späteren Kapiteln vertiefen. Das Namespace-Konzept dominiert die Sicherheitsstrategie von OSGI und erlaubt die Isolation fremder Services voneinander bei gleichzeitig möglicher Kommunikation über Service-Interfaces. Zunehmend nutzen auch Application-Server das Namespace-Konzept zur Isolation von Applikationen voneinander und vom Application Server Code. Eine hierarchische Form des Classloaders wird in Kap. 7.1.10 der Application-Server-Security dargestellt, das Peer-To-Peer-Classloading-Konzept im Kap. 8.1.2 der sicheren Multi-Vendor-Komponentensysteme. Auch die Mächtigkeit der Object Capabilities wird später im Kap. 12 am praktischen Beispiel gezeigt und anschließend theoretisch weiter vertieft.

5.4 Designaspekte der Umstellung auf Java 2 Security

Koved et al. haben kürzlich ein Konzept zur Umstellung von Eclipse auf die Verwendung von Java-2-Security-Mechanismen erstellt [KHP]. Die folgende Darstellung orientiert sich an ihren Beispielen und Diagrammen. Eine solche Umstellung bedeutet keineswegs lediglich die Erstellung eines Policy Files mit entsprechenden Permissions (obwohl auch dies einen nicht zu unterschätzenden Aufwand darstellt). Gleichzeitig muss der Code so strukturiert werden, dass er entsprechende Privilegien unterer Schichten oder Bibliotheken kapselt und vor dem Client-Code schützt. Dazu behandeln wir hier ein (kommentiertes) Beispiel von Koved et al. Abbildung 5.22 zeigt das zugrunde liegende Szenario.

Der Client-Code versucht hier, einen Socket zu erzeugen. Dies ist durch eine `SecurityManager.checkPermission(Permission p)`-geschützte Methode, deshalb benötigt der Client die entsprechende Permission `p`. Der Bibliothekscode will jedoch alle solchen Aktivitäten protokollieren und benötigt deshalb ein File-Objekt. Da dies ebenfalls eine geschützte Operation ist, bricht der StackWalk-Mechanismus an dieser Stelle ab, da der Client-Code keine entsprechende Permission besitzt.

Dazu nun das kommentierte und leicht veränderte Code-Beispiel von Koved et al.:

```
import java....*;  
  
public class LibraryCode2 {  
    private static final String logFileName = "audit.txt"; //  
    internal  
  
    // va-  
    riabile!!
```

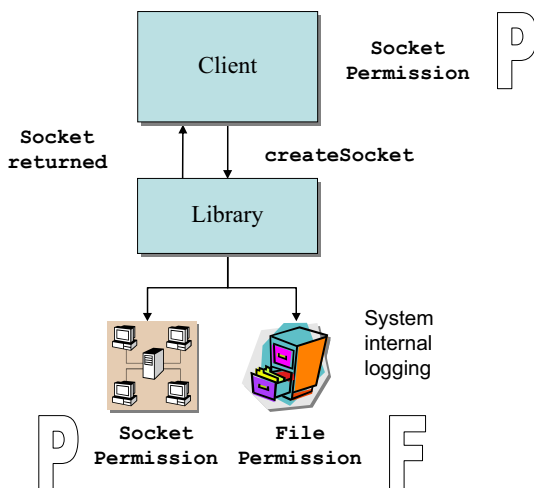


Abb. 5.22 Beispielszenario nach [KHP]

```

public static Socket createSocket(String host, int port)
throws unknownHostException, IOException,
PrivilegedActionException {

    Socket socket = new Socket(host, port);
    File f = new File(logFileName);
    PrivWriteOp op = new PrivWriteOp(f);
// privileged internal operation
    FileOutputStream fos =
        (FileOutputStream) AccessController.doPrivileged(op);
    BufferedOutputStream bos = new BufferedOutputStream(fos);
    PrintStream ps = new PrintStream(bos, true);
    ps.print("Socket " + host + ":" + port);
    return socket;
}

class PrivWriteOp implements PrivilegedExceptionAction {
    private File f;
    PrivWriteOp (File f) {
        this.f = f;
    }
    public Object run() throws IOException {
        return new FileOutputStream(f); // returns file
        capability }
}

```

An diesem Beispiel lassen sich die Konsequenzen der Code Access Security gut demonstrieren. Eine Möglichkeit, den Code wieder ablauffähig zu machen, wäre natürlich, dem Client-Code die Permission F für den File-Zugriff durch einen entsprechenden Eintrag im Policy-File zu geben.

Es ist nicht einfach, solche Stellen im Code zu entdecken und manuell zu bereinigen. Werkzeuge können dabei helfen, die Stellen zu finden (s. z. B. SWORD4J von [PKH]). Wichtiger ist jedoch die Frage, ob ein bloßes Ergänzen von Permissions ausreicht. Zwischen Client und Bibliothekscode liegt eine Abstraktionsgrenze: Der Client soll gar nicht über Implementationsdetails Bescheid wissen. Also ist es richtig, an dieser Stelle in der Bibliothek `doPrivileged()` einzusetzen und das File ohne entsprechende Client-Permission zu erzeugen. Anschließend erhält der Bibliothekscode eine File-Permission für das Logfile sowie die Erlaubnis, `doPrivileged()` zu verwenden.

Aber wie sieht es in Bezug auf die Socket-Permission aus? Was wäre, wenn die Bibliothek verschiedenste Verbindungsarten anbieten könnte, oder ein eventuell darüber liegender Layer zwischen mehreren Verbindungsarten transparent wechseln könnte? Macht es dann immer noch Sinn, vom Client eine sehr konkrete Socket-Permission zu verlangen? Sollte der Client evtl. eine allgemeine Netzwerk-

oder Mail-Permission erhalten und darunter liegende Bibliotheken diese mit `doPrivileged()` in konkrete Ressourcen verwandeln?

Schnell zeigt sich bei diesen Fragestellungen, dass auch die Permissions auf sehr verschiedenen Abstraktionsschichten funktionieren und die Software entsprechend diesen Schichten mit `doPrivileged()`-Aufrufen strukturiert werden muss. Code Access Security greift damit plötzlich sehr tief in die Systemarchitektur ein. Dies wird bei Java im Vergleich zu .NET noch dadurch verschärft, dass die Zugriffskontrolle programmatisch erfolgt, d. h. Aufrufe von `checkPermission()` sind fest im Code verdrahtet. .NET erlaubt hier deklarative CAS. Wir werden das Problem der unterschiedlichen Ebenen von Rechten anhand der Enterprise Security weiter unten noch vertiefen.

Das Beispiel von Koved et al. enthält noch weitere interessante Details zur Java Code Access Security, z. B. die Rolle des mit `doPrivileged()` erzeugten Logfiles:

```
class PrivWriteOp implements PrivilegedExceptionAction {
    private File f;
    PrivWriteOp (File f) {
        this.f = f;
    }
    public Object run() throws IOException {
        return new FileOutputStream(f); // returns file
        capability }
}
```

Wie im Code deutlich sichtbar, wird das File im Status „privileged“ geöffnet bzw. erzeugt, in einen `FileOutputStream` gewrappt und dann nach dem Security Check dem Aufrufer ausgehändigt. Dieser kann anschließend nach Belieben damit verfahren – ohne weitere Sicherheitschecks. Dies ist ein Resultat der Entscheidung die Zugriffskontrolle bei einem File-Objekt in der Phase des Erzeugens bzw. Öffnens, jedoch nicht mehr beim Lesen und Schreiben durchzuführen (ganz ähnlich dem Verhalten eines Unix Filedescriptors). Aus Performancegründen erscheint dieses Verhalten sehr sinnvoll, es ermöglicht aber auch den Missbrauch des Objektes nach dem Sicherheitscheck – ein sogenannter *Time-Of-Check-To-Time-Of-Use* (TOC2TOU)-Fehler.

```
Class X{
    X(){
        permission= new XPermission();
        SM.checkPermission(Permission);
    }
    readX() { // no check}
    writeX() { // no check}
}
```

Wie gefährlich ist nun die `doPrivileged()`-Operation an dieser Stelle wirklich? Die Gefährlichkeit hängt im Wesentlichen von zwei Fragen ab: Woher kamen die Argumente der privilegierten Operation und was passiert mit dem erzeugten Objekt? In diesem Fall sieht man, dass der File-Name innerhalb der Bibliothek erzeugt wird und nicht durch den Client herein gegeben wird. Das erzeugte Objekt selbst bleibt unter der Kontrolle der erzeugenden Klasse und ist damit nicht gefährdet, obwohl keine Sicherheitschecks mehr beim Lesen und Schreiben stattfinden werden.

Würde auf der anderen Seite aber der Client ein zusätzliches File-Argument als Parameter mitgeben (dies müsste überhaupt nichts mit dem Logfile der Bibliothek zu tun haben, es genügt bereits, wenn der Parameter Teil einer privilegierten Operation wird), dann bestünde bereits die Möglichkeit einer Attacke: Der Client könnte z. B. den Namen des Logfiles verwenden und die Bibliothek würde dann – falls sie nicht explizit prüft, ob ein als Argument gegebener Name zufälligerweise dem Namen des Logfiles gleicht – mit ihren eigenen Privilegien in einer `doPrivileged()`-Aktion das Logfile überschreiben bzw. auslesen. Norm Hardy beschreibt unter dem Titel „Confused Deputy“ eine solche Attacke auf einem Großrechner (s. [Har]). Sie dient heute als beliebtes Beispiel für die Mächtigkeit von Capabilities und dafür, wie gefährlich es ist, Namen in Objekte zu transformieren.

Nachdem sich in diesem Beispiel die `doPrivileged()`-Aktion als ungefährlich herausgestellt hat (das muss nicht so bleiben: Ein beunruhigender Gedanke ist, dass nach unserer Prüfung jemand auf die Idee kommen könnte, die Bibliotheksmethode wie beschrieben mit einem File-Argument zu erweitern und damit nachträglich eine gefährliche Confused-Deputy-Attacke zu ermöglichen!), müssen wir noch die Gefahr bezüglich des an den Client zurückgegebenen Objektes vom Socket prüfen. Wie gesagt, der Sicherheitscheck ist vorbei und der Client erhält vollen und uneingeschränkten Zugriff auf das Objekt durch dessen Referenz. Die Referenz fungiert hier als sogenannte *Object Capability* – sie ist Recht und technische Zugriffsmöglichkeit in einem.

Aber ist das nicht gefährlich? Man kann nachweisen, dass mit solchen einfachen Capabilities bestimmte Sicherheitsmodelle nicht implementiert werden können. Dazu gehören vor allem die sog. *-Properties aus dem Geheimdienstbereich und das Problem der späteren Rücknahme eines Rechts auf Zugriff. Miller und Shapiro haben jedoch in einem ausgezeichneten Artikel ([MS]) nachgewiesen, dass durch geeignete Softwaretechniken diese Defizite leicht ausgeglichen werden können. In diesem Fall könnte unter Verwendung des Decorator Patterns ein Wrapper bzw. ein Facet um das Socket-Objekt geschaffen werden, der feingranular jeden Zugriff auf das Objekt einer zusätzlichen Zugriffskontrolle unterzieht (s. Abb. 5.23).

Java bietet mittlerweile in Form des sog. Guard Objects diese Funktionalität als Ergänzung zum stackwalkbasierten Zugriffsschutz an (weitere Beispiele finden sich bei [New04]). An dem Decorator/Guard-Beispiel lässt sich die Flexibilität von Capabilities in Verbindung mit Softwaretechniken erkennen. So kann der Decorator sowohl statische als auch dynamische Regeln für eine Zugriffsentscheidung verwenden.

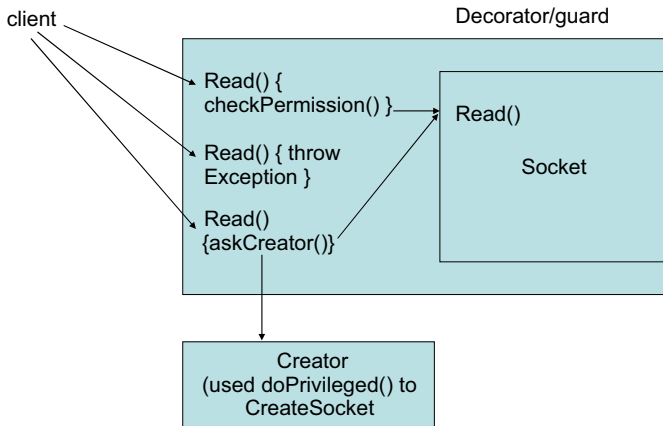


Abb. 5.23 Wrapper um das Socket-Objekt des Beispiels

Fassen wir kurz abschließend einige Tipps für die Verwendung von `doPrivileged()` zusammen:

- `doPrivileged()` führt neben dem Prinzip der Access-Control-Matrix ein weiteres Sicherheitsprinzip ein: Sicherheitsmodi mit unterschiedlichen Regeln. An den Schnittstellen der Modi treten bekannte Sicherheitsprobleme wie etwa der „Confused Deputy“ auf.
- Nur intern verwendete Objekte sollten nicht nach außen hin sichtbar werden.
- Wenn möglich, keine Argumente von Clients für `doPrivileged()`-Sektionen verwenden.
- Falls dies unumgänglich ist, müssen diese Argumente nicht nur auf ihre Syntax geprüft werden, sondern auch auf ihre Semantik im Zusammenhang mit der gesamten privilegierten Funktion: Überschreibt möglicherweise ein File-Argument Systemfiles?
- Mehrere `doPrivileged()`-Aufrufe innerhalb einer Protectiondomain haben die gleichen Permissions. Das ist eine Chance für eine Confused-Deputy-Attacke, falls die Argumente nicht semantisch geprüft werden.
- Wenn Zugriffskontrolle nur während der Erzeugung eines Objektes verwendet wurde, aber das Objekt dem Aufrufer zurückgegeben wird, dann kann durch einen Decorator/Guard eine beliebig feingranulare zusätzliche Zugriffskontrolle implementiert werden.

Bei der nachträglichen Absicherung von Eclipse durch Java 2 Security tauchen im Grunde die gleichen fundamentalen Probleme wie weiter oben im Falle von SELinux beschrieben auf:

- Durch Trial-und-Error-Verfahren müssen die nötigen Privilegien herausgefunden werden, und zwar durch Ausführen des Code bzw. durch Code-Inspektion.
- Eine komplette Abdeckung ist meist unmöglich, und so tauchen später zur Laufzeit Security-Exceptions auf.

- Im Notfall werden sehr großzügige Permissions vergeben (unter Java: `All-Permission`), damit die Applikationen laufen.
- Die Verlagerung von Code in ohnehin privilegierte Funktionen erlaubt geringere Client Permissions.
- Eine durchgängige Implementation von Least-Privilege-Prinzipien (POLA) erfordert oft architektonische Änderungen im Code.

Besonders der erste Punkt weist auf einen wichtigen Zusammenhang hin: Security unterläuft die so sehr geschätzte Trennung von Interface und Implementation zur Laufzeit. Security zeigt sich somit als typische Querschnittsaufgabe, in diesem Fall sogar durch zentrale Abstraktionen. Das Tool SWORD4J von Koved et al. erlaubt z. B. eine Analyse aller Permissions, die direkt oder indirekt durch privilegierte Aktionen dem Client-Code zur Verfügung stehen.

Halten wir abschließend noch einmal die Grundprobleme einer nachträglichen Einführung von Code Access Security fest: Es besteht die Notwendigkeit einer Art Reverse-Engineering des Codes bei der Umstellung einer Applikation auf CAS: Mühsam muss im Trial-and-Error-Verfahren jede Stelle identifiziert werden, die zur Laufzeit eine Security-Exception auslöst. Danach müssen die Permissions modifiziert werden. Es sind teilweise architektonische Änderungen nötig, damit es nicht bei der Definition einer großen und viel zu groben Protection Domain bleibt.

5.5 Code Security für Enterprise-Applikationen

„Turn on code security“ fordert Ted Neward in [New04] auch für Enterprise-Applikationen – also solche, die im Allgemeinen gut kontrolliert sind in Bezug auf Source-Code-Management und die sichere Konfiguration von Betriebssystemen. Er begründet das mit der gesteigerten Sicherheit auch im Falle von Programmierfehlern. Attacken, die mangelnde Input-Validierung ausnutzen, können Code zum Zugriff auf falsche Ressourcen veranlassen. Ein gutes Beispiel von Neward ist hier die bekannte „`..\\.`“-Pfadattacke, bei der es durch einen Fehler bei der Prüfung eines File-Namens oder Pfades zum Zugriff auf eine Ressource kommt, für die normalerweise gar kein Zugriff aus der Server-Applikation heraus vorgesehen ist. Findet der Zugriff jedoch mit den Rechten des Servers statt, dann gelingt er häufig trotzdem und der Client erlangt illegalen Zugriff.

Erfolgt der Zugriff jedoch geschützt durch Code Access Security, dann verweigert der Server den Zugriff automatisch, da die entsprechende File-Permission nicht vorliegt (s. Abb. 5.24). Die Schritte, die nötig sind, um eine eigene Policy zur Verwendung unter Java 2 Security zu entwickeln, sind gut in [Wund] beschrieben.

Die Java 2 Security kann in diesen Fällen schlimmeres Verhüten, indem das Prinzip der „Full Mediation“ von Saltzer und Schröder erreicht wird. Der oben geschilderte Stack-Inspection-Mechanismus führt dazu, dass die effektiven Zugriffsrechte der Applikation verwendet werden und diese schließen keinen Zugriff

auf `/etc/passwd` ein. Abbildung 5.25 zeigt den Abbruch des Zugriffs trotz vorhandenen Softwarefehlers. Somit bleibt auch ein ansonsten fataler Input-Validierungsfehler eher harmlos. (Zu Recht weist der Autor von `qmail`, D. Bernstein, jedoch

CAS für Server Applications

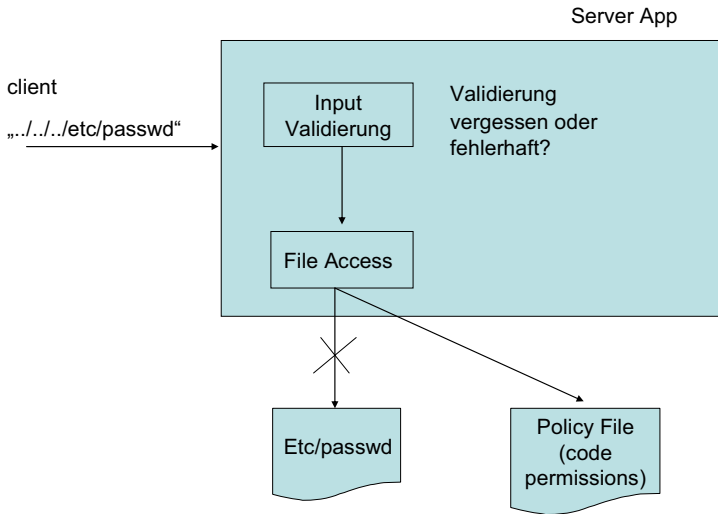


Abb. 5.24 Abwehr eines Angriffs auf die Input-Validierung einer Server-Applikation durch Code Access Security

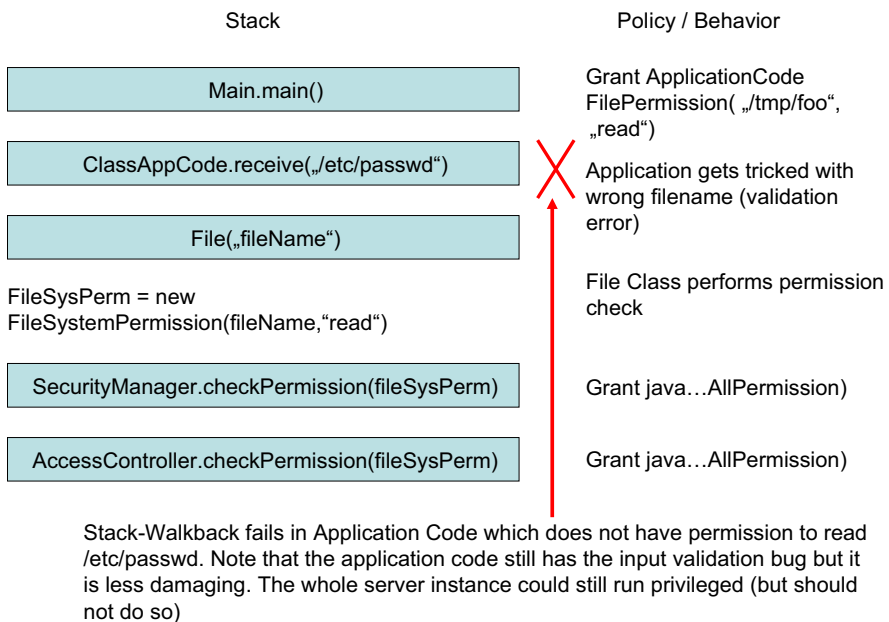


Abb. 5.25 Stack Inspection verhindert Zugriff auf kritische Ressource

darauf hin, dass diese Rechteeinschränkung den Softwarefehler, der zweifellos vorliegt, nicht wirklich behebt.)

Besonders interessant wird Java 2 Security dann, wenn die Permissions auf Rollendefinitionen ausgeweitet werden, d. h. bei der Zugriffsprüfung wird auch die Rolle eines Aufrufers und nicht nur das Recht des Codes beachtet.

Wieso gibt es dann so wenige Applikationen, die Java 2 CAS einsetzen? Einige Gründe wurden schon erwähnt, beispielsweise die Einstellung von Entwicklern im Unternehmensumfeld, dass CAS nur etwas für Applikationen mit mobilem Code wäre – also die falsche Einschätzung, dass CAS nur gegen eingeschleusten Code hilft. Es ist hier das User Conceptual Model, das zu schlechter Sicherheit führt.

Schwerer wiegt allerdings das Performanceargument: Hersteller von Application-Servern geben an, dass das Einschalten von Java CAS ca. 40% CPU-Performance kostet. Sie empfehlen deshalb, dass CAS nur zu Testzwecken eingeschaltet werden sollte, zu Laufzeit aber deaktiviert werden sollte (d. h. keine SecurityManager/Policy-Kombination instanziiert werden soll). Abbildung 5.26 zeigt einen

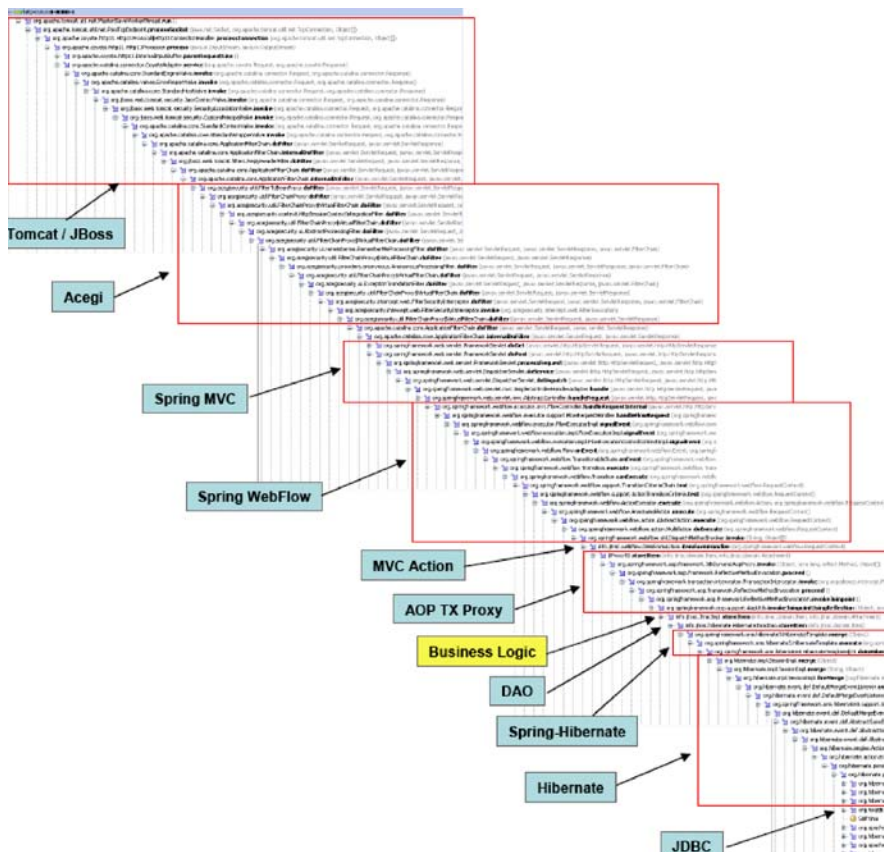


Abb. 5.26 Java-Call-Stack von http bis jdbc (entnommen aus [Thom])

typischen Callstack, der im Fall des Stack-Walkback durchlaufen wird und verdeutlicht, woher dieser Performanceverlust kommt.

Deutlich sichtbar sind in Abb. 5.26 die verschiedenen Protection Domains und der winzige Anteil der Business Logic am Stack, der natürlich in erster Linie dem Prinzip des Separation of Concerns geschuldet ist, wie es in Kap. 6 zu Enterprise Security noch genauer behandelt wird.

Ein weiterer Grund für die fehlende Anwendung der Code Security unter Java ist die mangelnde psychologische Akzeptanz (Salzer und Schröder) dieses sehr feingranularen Kontrollmechanismus: Es ist sehr mühsam, entsprechende Policies aufzustellen, d. h. Protection Domains mit entsprechend granularen Rechten für eine Applikation zu erstellen. Offensichtlich sind entsprechende Software-Tools nötig, um die richtig zugeschnittenen Sandboxes automatisch zu erstellen und zu testen (vgl. [JSR]).

Stellt also Java in Bezug auf das zögerliche Anwenden der Code Access Security einen Sonderfall dar? Leider nicht. In Kap. 7 zur Application Server Security werden wir kurz anhand der Erkenntnisse von Dinis Cruz die Probleme im .NET-Umfeld mit „Fully Trusted“-Servern zusammenfassen.

Literatur

- [GA] S. Govindavajhala, Andrew W. Appel, Using Memory Errors to Attack a Virtual Machine, 2003 IEEE Symposium on Security and Privacy, pp. 154–165, 2003, <http://www.cs.princeton.edu/~appel/papers/memerr.pdf>
- [Gong] L. Gong, Java™ 2 Platform Security Architecture, <http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc.html>. Permissions findet man unter <http://java.sun.com/j2se/1.4.2/docs/guide/security/permissions.html>
- [Har] N. Hardy, The Confused Deputy (or why capabilities might have been invented), ACM SIGOPS Operating Systems Review Vol. 22 (4) (October 1988), pp. 36–38
- [JAAS] JAAS Authentication Tutorial, <http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/tutorials/GeneralAcnOnly.html>
- [JSR] IBM Java Security Research Homepage, http://domino.research.ibm.com/comm/research_projects.nsf/pages/javasec.index.html
- [KHP] L. Koved, T. Habeck, M. Pistoia, Enabling Java 2 Runtime Security with Eclipse Plug-ins, OSGi Developer Forum and World Congress. Paris, France, October 2005
- [Kicz] G. Kiczales, A. Paepcke, Open Implementations and Meta-Object Protocols, Tutorial, <http://www2.parc.com/csl/groups/sda/publications/papers/Kiczales-TUT95/for-web.pdf>
- [Krü] G. Krüger, Handbuch der Java-Programmierung, Addison-Wesley, 2002
- [MS] M.S. Miller, J. Shapiro, Paradigm Regained: Abstraction Mechanisms for Access Control, <http://www.eros-os.org/papers/paradigm-final.pdf>
- [New04] T. Neward, Effective Enterprise Java, Addison-Wesley 2004
- [New01] T. Neward, java.security.Policy: When “java.Policy” Just Isn’t Good Enough, <http://www.javageeks.com/Papers/JavaPolicy/JavaPolicy.pdf>
- [OXACML] Open eXtensible Access Control Markup Language, www.openxacml.org
- [PKH] M. Pistoia, L. Koved, T. Habeck, SWORD4J: Secure WORKbench Development environment For Java. ECOOP 2005 – 19th European Conference on Object-Oriented Programming, ACM, July 2005

- [KPH] M. Pistoia, L. Koved, T. Habeck,, Adding Security to Eclipse,
<http://www.eclipse.org/equinox/incubator/security/documents/addingsecurity/Adding%20Security%20to%20Eclipse.pdf>
- [SL] V. Samar, C. Lai, Making Login Services Independent of Authentication Technologies, <http://java.sun.com/security/jaas/doc/pam.html>
- [Thom] P. Thomas, Homepage, <http://ptrthomas.wordpress.com>
- [WF] D. Wallach, E. Felten, et al., Extensible Security Architectures for Java
- [Wund] H. Wunderlich et al., WebSphere Application Server for z/OS V5 and J2EE 1.3 Security Handbook, IBM RedBook,
<http://www.redbooks.ibm.com/abstracts/sg246086.html?Open>
- [XACML] eXtensible Access Control Markup Language (XACML) Version 2.0,
www.oasis.org

Kapitel 6

Enterprise Security

Das Thema „Enterprise Security“ gilt unter Entwicklern als nicht besonders einfach. Das erkennt man schon daran, dass immer noch eine große Zahl von Stand-alone-Applikationen auch in großen Firmen entwickelt wird. Häufig sind dies Applikationen, die den klassischen Ansatz des „Application Tower“, ohne Rücksicht auf die Integration in bestehende Software- und Sicherheitsstrukturen implementieren.

Multi-Tier-Applikationen auf Basis von JAVA EE oder .Net gelten als schwierig zu entwickeln und als teuer zu installieren und zu unterhalten. Die Sicherheitskonzepte dieser Frameworks gelten ebenfalls als schwierig. Nicht zuletzt sind es die komplexen Application-Server, die für Probleme bei der Sicherheit sorgen. Sie stellen Sicherheitsfunktionen zur Verfügung und benötigen ihrerseits eine Integration in bestehende Sicherheitsinfrastrukturen der Firmen.

Dieses Kapitel behandelt zwei große Themenbereiche:

End-to-end-Security:

Wie wird ein Request von außerhalb der Firma in Empfang genommen, abgesichert und durch die Infrastruktur zu den Backend-Systemen geleitet? Welche Probleme tauchen hierbei auf und wie sehen die Lösungsmöglichkeiten z. B. für Delegation aus?

Das Verschieben von Security-Code aus den Applikationen in die Infrastruktur:

Dies geschieht, um Applikationen portabel und wartbar zu machen (Separation of Concerns and Context).

Beide Themenfelder werden zunächst allgemein und danach im Umfeld von JAVA-EE-Technologien vorgestellt. Ziel ist es dabei, Entwicklern die Grundproblematiken aufzuzeigen und Lösungsmöglichkeiten zu entwickeln, ohne dabei jedoch im Detail stecken zu bleiben.

6.1 Grundlagen von Enterprise Security

Enterprise Security lässt sich anhand dreier Dimensionen diskutieren: Verteilung (Topologie), Infrastruktur und Applikationsarchitektur. Alle drei beeinflussen sich gegenseitig. So haben unterschiedliche Konzepte von Codeverteilung Einfluss auf die Infrastruktur, die zur Absicherung nötig ist. Andererseits erzwingt die Infrastruktur z. B. durch Zonenkonzepte oder Netzwerkstruktur bestimmte Softwarekonzepte. Im Folgenden wollen wir diese Abhängigkeiten genauer beleuchten, und zwar beginnend mit der Architektur von Applikationen.

6.1.1 *Vom Application Tower zur verteilten Enterprise-Applikation*

Der klassische Application Tower entspricht in etwa den Konzepten von Java2SE bzw. dem klassischen Konzept von „Applikation“ als einer Einheit von Code, die bestimmte Geschäftsvorgänge durchführen kann. „Application Tower“ ist als Begriff heute negativ besetzt, da derartig gebaute Applikationen meist die nötige Funktionalität selbst programmieren, statt dafür existierende Komponenten oder Services einzusetzen. Gleichzeitig errichten sie meist eine eigene Verwaltung von sicherheitsrelevanten Informationen wie Usern oder Rollen und Rechten (s. Abb. 6.1). Diese Eigenschaften führen dazu, dass bei Änderungen der Organisation, der Geschäftsvorgänge oder des Personals der Firma für jede Applikation ein großer Änderungsaufwand entsteht und ein einheitlicher Datenbestand nicht möglich ist.

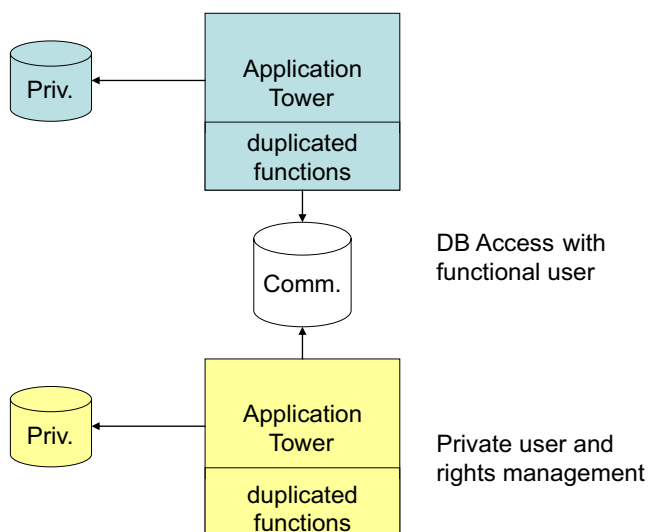


Abb. 6.1 Application Tower mit Duplizierung sicherheitsrelevanter Funktionen

Die Access Control innerhalb einer solchen Applikation geschieht wie bei fast allen Anwendungen innerhalb von Firmen auf der Basis von Principals (authentierte Nutzer) und deren Rollen bzw. Rechten. Die Nutzer und ihre Rollen und Rechte werden häufig von der Applikation selbst verwaltet, meist in der gleichen Datenbank, die auch für die Geschäftsprozesse der Applikation benötigt wird. Diese Art von Applikation kann sowohl als sogenannter *Fat Client* oder als Server-Applikation ausgelegt sein. Sicherheitstechnisch sind zwei Dinge sehr interessant: Die Kontrolle des Ablaufs innerhalb der Applikation sowie der Zugriff auf die Datenbank.

Ohne Java 2 Security und ohne JAAS-gestützte Impersonation basiert die gesamte Sicherheit der Geschäftsvorgänge innerhalb der Applikation auf programmierten Sicherheitsabfragen. Ohne den Einsatz einer Sandbox auf Code- oder Identitätsbasis gibt es also keine zusätzliche Kontrolle. Der Applikationscode bildet eine einheitliche Vertrauenszone.

Der Zugriff auf das Backend erfolgt entweder über die UserID – falls die Benutzer auf der Datenbank angemeldet sind, oder mithilfe einer sogenannten *Functional UserID* – das bedeutet, einem eingetragenen Datenbankbenutzer, der Zugriffsrechte auf alle für die Applikation relevanten Daten innerhalb der DB besitzt. Dieser Functional User sollte

- kein interaktiver Benutzer auf dem System sein, auf dem die Applikation läuft,
- keine echte Nutzeridentität darstellen, die zu diesem Zweck missbraucht wird und
- nicht im Code hart programmiert sein.

Das Konzept des Functional Users ist aus der Sicht der Systemsicherheit nicht unkritisch: So ist der bekannte Unix Account „root“ oder der Windows-User „Administrator“ ein Functional User, der keinen Rückschluss auf eine echte Person erlaubt und deshalb Probleme mit der Nachvollziehbarkeit und Verantwortlichkeit für bestimmte Aktionen aufwirft. Andererseits sollte grundsätzlich vermieden werden, für Maschinen oder Softwareagenten Principals von echten Personen einzusetzen: Dazu wären nämlich die Credentials dieser Personen im Zusammenhang mit der Applikation festzuhalten. Daraus ergeben sich Missbrauchsmöglichkeiten der persönlichen Accounts der Mitarbeiter.

Weitere sicherheitskritische Aspekte dieser Architektur sind:

- Direkter Zugriff auf die Datenbank, was vor allem bei Verwendung von Dynamic SQL die Datenbank lahm legen kann.
- Falls serverseitig betrieben, wird meist ein sog. Connection Pool verwendet, d. h. es werden mehrere Verbindungen zur Datenbank gleichzeitig gehalten. Alle diese Verbindungen werden typischerweise mit dem gleichen Functional User eröffnet, sodass der Zugriff auf die Datenbank unabhängig von einem echten User geschieht. Applikationsfehler in Bezug auf Userrechte können so nicht vom Backend entdeckt werden.
- Der Zugriff über den Functional User auf die Datenbank ist sehr breit, da er ja auf die Daten aller User zugreifen muss.

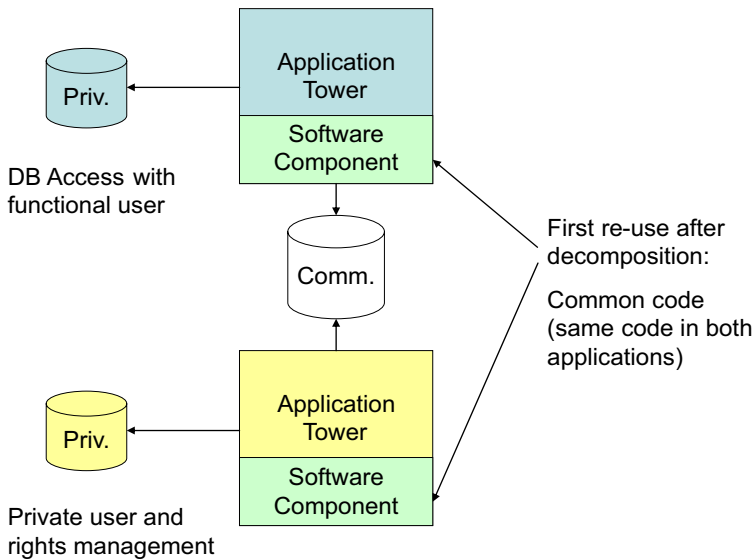


Abb. 6.2 Application Tower mit wieder verwendbarer SW-Komponente

Der nächste Schritt in der Evolution der Applikationsarchitektur bestand in der Einführung von Softwarekomponenten bzw. deren Wiederverwendung (s. Abb. 6.2). Diese Komponenten waren noch fester Teil der Applikation, insofern trat kein Problem bezüglich Authentisierung oder Autorisierung an der Grenze zwischen Applikation und Komponente auf. Dagegen machten sich andere Probleme bemerkbar: Was tun die Komponenten intern? Kann eine Applikation diesen Komponenten so unbedingt vertrauen wie den selbst entwickelten Teilen? Es zeigte sich schnell, dass viele dieser Komponenten nur eine schlechte Anpassung an die Infrastruktur besaßen. Typisch für Enterprise-Umgebungen ist die Vielfalt der verwendeten Services und Backend-Systeme unterschiedlichster Hersteller.

Außerdem wurde erkannt, dass die Komponenten leichter wieder verwendbar waren, wenn sie zur Laufzeit dynamisch aufgerufen werden konnten, statt nur als Softwarebibliothek beim Bau der Applikation zu existieren. So wurden aus den Softwarekomponenten verteilte, allgemein verfügbare Komponenten, die jetzt natürlich jeden Aufruf auf entsprechende Autorisierung des Aufrufers prüfen mussten. Probleme wie Delegation eines Aufrufs über mehrere Knoten hinweg traten auf und der Sicherheitscode wurde immer komplexer.

6.1.2 Security-Infrastruktur

Im Bereich der CORBA-Infrastruktur begann man dann sehr schnell damit, Sicherheitsaspekte einem dedizierten Service zu übertragen und im Übrigen so weit

wie möglich aus den Applikationen in die Infrastruktur, also die Middleware, zu verschieben.

Kennzeichnend für diese Phase waren die Einführung eines Transaktions- bzw. Sicherheitskontextes, der jeden Aufruf für die Applikation unsichtbar begleitete und von Infrastrukturschichten verarbeitet wurde. Sehr wichtig war ferner die Erkenntnis, dass sich viele Applikationsprobleme nicht durch ein einfaches Layer-Pattern oder durch Aufteilung in separate Komponenten lösen ließen. Sicherheitsfragen sind typische Querschnittsthemen (sogenannter Cross-Cutting Concern), d. h. Code zu ihrer Behandlung tritt an den unterschiedlichsten Stellen der Applikation auf. So konnten Applikationsentwickler leicht den Aufruf von Sicherheitsprüfungen vergessen und damit die ganze Sicherheit untergraben. In Bezug auf die Wartbarkeit stellte sich heraus, dass selbst der programmierte Aufruf von Services einen gewissen „cold spot“ – also eine ungewollte harte Codierung – in einer Applikation darstellte.

Wollte man sich also nicht komplett auf die korrekte Programmierung der Sicherheitsabfragen als einzigen Sicherheitsmechanismus innerhalb einer Applikation verlassen, war der logische nächste Schritt die sogenannte „Separation of Concerns and Context“: Die Delegation von non-functional Requirements wie der Security an ein Framework bzw. dessen Runtime-Stellvertreter, den Container. Diese Architektur, die heutige Multi-Tier-Applikationen auf Basis von JAVA EE oder .NET beschreibt, ist in Abb. 6.3 wiedergegeben.

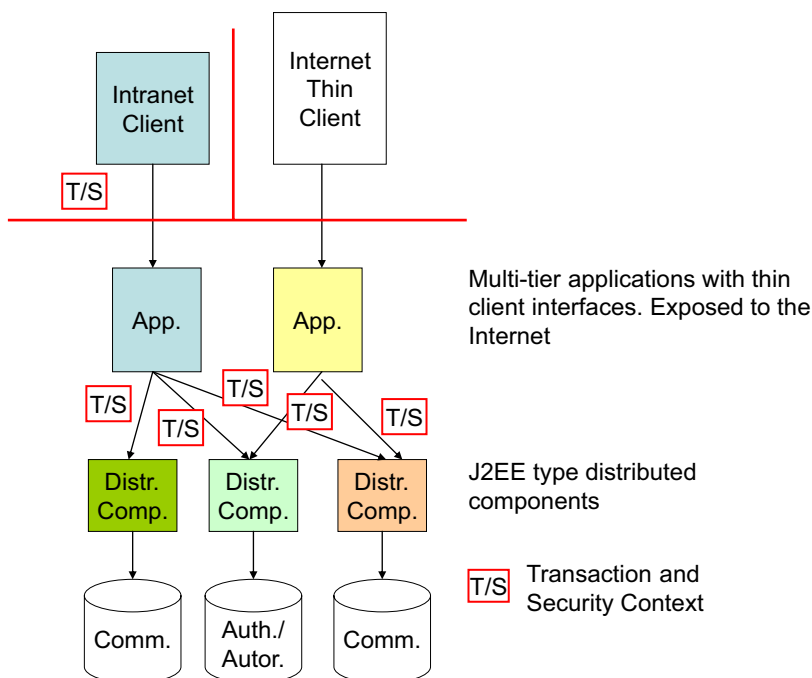


Abb. 6.3 Multi-Tier-Applikation mit verteilten Komponenten auf JAVA-EE-Basis

Typisch für diese Phase war auch die Ausdehnung der Firmensoftware auf Internetkunden. Plötzlich gab es die verschiedensten Eingänge in die Applikationen von verschiedenen Sicherheitszonen aus. Die verteilten Komponenten konnten in den meisten Fällen gar keine initiale Authentisierung mehr durchführen, da sie keinen Zugriff auf die vom Client verwendeten Credentials hatten (und auch keinen haben sollten). Trotzdem musste verhindert werden, dass sich Aufrufe so kreuzten, dass z. B. nicht-authentisierte Kunden Zugriff auf kritische Komponenten oder Applikationen erhielten.

Somit stellten sich einige drängende Fragen:

- Wo sollen die Requests authentisiert werden und durch wen?
- Wo sollen die User autorisiert werden?
- Wie kann die Tatsache einer erfolgten Authentisierung auf sichere Weise von Komponente zu Komponente weitergegeben werden?
- Wie steht es mit Meta-Daten wie der Qualität der Authentisierung und ihrem Ort?

Es gab in der Vergangenheit genügend Beispiele für eine schlechte Sicherheitsarchitektur, z. B. das INPOL-Projekt der Polizeien der Bundesländer. Dort wurden Requests erst in der Datenbank über sogenannte „stored procedures“ authentisiert und autorisiert, mit fatalen Folgen für die Performance und die Wartbarkeit. Statt dieses Ansatzes hat sich in Bezug auf das Spannungsfeld von Sicherheit wie Performance das Trichterkonzept bewährt: Eine frühzeitige, schrittweise Filterung der Requests führt zu einer besseren Sicherheit des Gesamtsystems, genauso wie eine rechtzeitige Verengung der Eingänge, die Applikationen und Services vor Überlastung schützt.

6.1.3 Serviceorientierte Architekturen (SOA)

Der nächste Schritt – und dies ist momentan im Gange – besteht in der Umstellung der Applikationen auf die sogenannten serviceorientierten Architekturen (SOA): Unabhängige Services werden durch Business Services in bestimmte Businessprozesse zusammengestellt (s. Abb. 6.4). Services können auch extern sein, was enorme Probleme für die Interoperabilität von Sicherheitsmechanismen sowie die gegenseitigen Vertrauensverhältnisse bringt. In Kap. 9 werden die dafür nötigen Prinzipien und Techniken diskutiert, z. B. die Betonung von objektbasierter oder messagebasierter Sicherheit durch Signaturen und Verschlüsselung.

Nicht zuletzt ergeben sich in solchen Architekturen mit Beteiligung externer Partner auch große Probleme für die Datensicherheit im Zusammenhang mit der Überlassung, dem Test, der Integration und dem Betrieb externer Services. Aber selbst ohne externe Partner stellt allein die Notwendigkeit von Data Caching in weltweiten Servicearchitekturen ein Sicherheitsproblem dar: Wie wird garantiert, dass Daten im Cache nur nach korrekter Autorisierung herausgegeben werden?

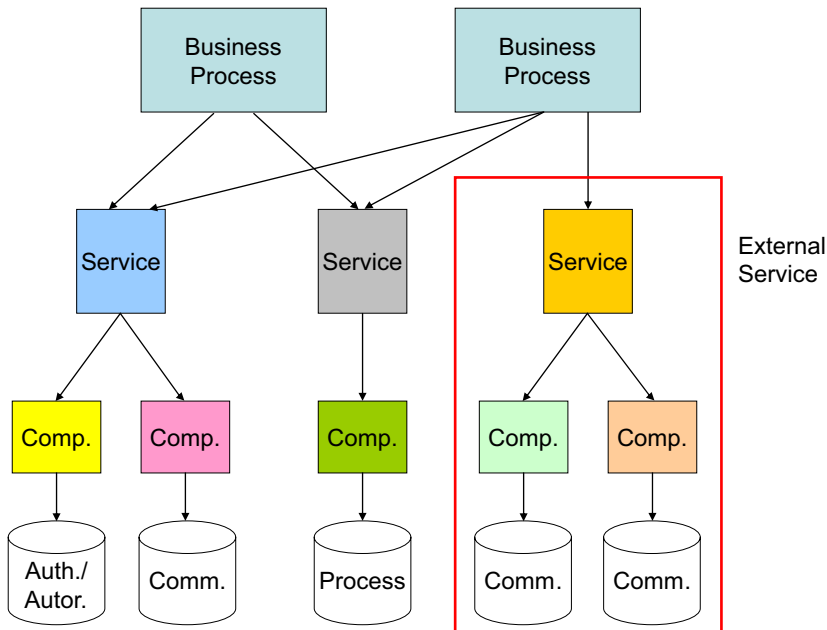


Abb. 6.4 Serviceorientierte Architektur

Durch das Caching sind die Daten ja aus dem Schutzkontext ihrer Objekte herausgerissen worden. Auch hier gibt es die Möglichkeit, die Constraints bei den Daten festzuhalten und beim Zugriff zu prüfen.

Eines der Hauptargumente für die SOA-Architektur ist die sog. „lose Kopplung“ zwischen den Services und den Komponenten, die dadurch angeblich erreicht werden kann. Ein kurzer Blick auf die Architektur zeigt jedoch, dass es immer noch Aufrufpfade gibt, bei denen der Aufrufer den Aufgerufenen kennen muss. Nicht notwendigerweise den Ort des Aufgerufenen, aber der Aufruf selbst sowie die Antwort darauf wird fester Bestandteil der Architektur des Aufrufers. Dadurch entsteht eine beträchtliche Kopplung. Diese Kopplung lässt sich noch weiter durch den Übergang zu Event-Driven Systems bzw. der Verwendung von Publish-Subscribe-Architekturen wie den Enterprise-Service-Bus reduzieren.

6.1.4 Eventgetriebene Systeme und Busse

Die momentan letzte Phase in der Entwicklung der Applikationsarchitektur besteht in einer noch weiteren Entkopplung der Applikationen von den Services durch Publish/Subscribe-Pattern im sog. Enterprise Service Bus (s. Abb. 6.5).

Gregor Hohpe beschreibt die Steigerung des Grades an Entkopplung in [Hoh] sehr gut in seinem Beispiel der Verarbeitung von Bestellungen. Hier wird nur

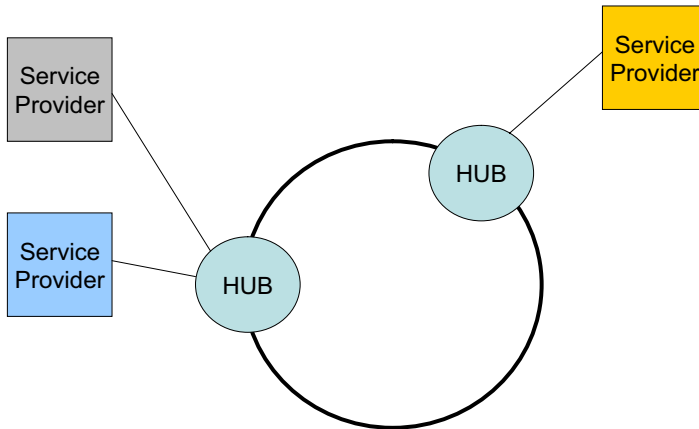


Abb. 6.5 Enterprise Service Bus

noch der Eingang einer Bestellung durch eine Applikation auf einem Bus gepostet. Subscriber-Services haben sich für solche Nachrichten registriert und posten ihrerseits asynchron dazu weitere wichtige Daten, z. B. die Zustelladresse des Bestellers. Entscheidend ist das asynchrone Reagieren auf Events durch Verarbeitung und anschließendes Posten von Ergebnissen. Streng genommen existiert auch hier noch eine Kopplung der Beteiligten in einem solchen System. Allerdings besteht diese Kopplung darin, dass eine vernünftige Kombination von Beteiligten vorhanden ist, damit eine „Applikation“ – eigentlich eher eine Verarbeitung einer Aufgabe – entsteht.

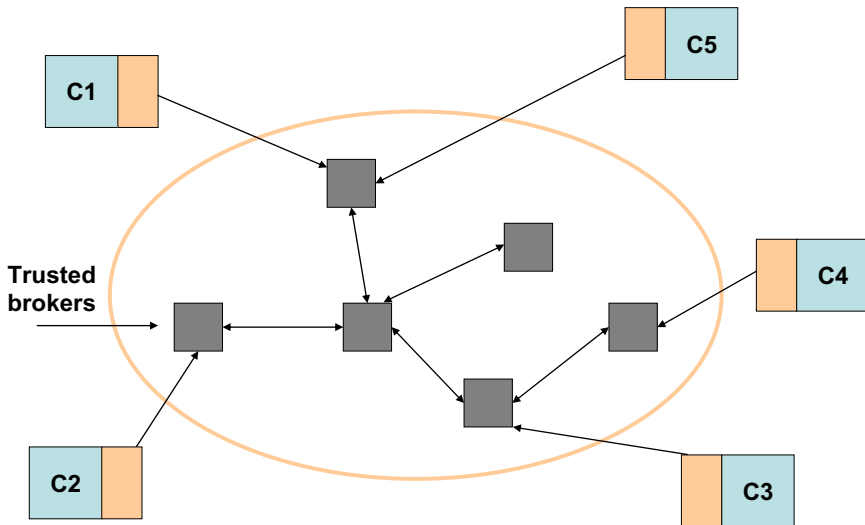
Für die Sicherheit ergeben sich aus diesem Ansatz ganz neue Probleme: Wenn Sender und Empfänger bewusst entkoppelt sind, wie kann dann ein Empfänger eine Nachricht authentisieren? Wie verschlüsseln? Notwendigerweise muss der Nachweis des Senders die Nachricht in irgendeiner Form begleiten. Wie soll dies geschehen: Transparent oder explizit? Zurzeit werden in Firmen proprietäre Wege gegangen, um z. B. die Senderidentität einem Empfänger zu übermitteln. Dies kann durch „Envelope-Techniken“ geschehen, bei der der Sendername einfach in die Nachricht eingebettet wird. Dies kann aber auch dadurch geschehen, dass Authentisierungs-Token wie das LTPA-Token von Websphere (siehe Kap. 7) in serialisierter Form der Nachricht angefügt und am Empfängerort zur Impersonation des Senders innerhalb einer Message Bean verwendet werden (s. [Reith]).

Am Grundproblem der Sicherheit in Event-Driven-Systemen ändert dies jedoch wenig: Vertraulichkeit durch Verschlüsselung setzt Schlüssel und damit Kenntnis von Schlüsselträgern voraus. Integrität durch Signaturen setzt voraus, dass jemand den Autor einer Nachricht kennt – genau das soll aber eigentlich nicht der Fall sein, um die lose Kopplung nicht zu zerstören. Dass Senderinformationen in eine Nachricht technisch eingebettet werden können, ist daher eher als Problem zu sehen.

Sicherheit in eventgetriebenen Systemen steckt daher noch in den Kinderschuhen. G. Mühl schlägt dazu ein System vor, das wie ein Overlay über dem verteil-

Communication-Architecture 4: distributed system with local brokers

Rebecca distributed notification middleware through overlay network

**See: Mühl et.al. Pg. 21**

ten System funktioniert und z. B. Schlüssel für die sichere Übertragung von Daten verteilt und die Rechtekontrolle für das Senden und Empfangen der Beteiligten implementiert (s. Abb. 6.6). Somit können die Beteiligten sichere Nachrichten senden und empfangen, ohne notwendigerweise die Empfänger oder Sender kennen zu müssen. Der Nachteil ist die Einführung eines globalen Admin-Overlays, aber dieser ist meist bereits aus Gründen der Systemverwaltung nötig [MFP]. Grundsätzlich ist aber auch die Frage zu stellen, ob Abläufe, bei denen z. B. die Vertraulichkeit der Nachrichten extrem wichtig ist, sinnvollerweise in einer eventgetriebenen Architektur behandelt werden sollten.

Eine genauere Betrachtung der Architektur zeigt jedoch die Grenzen der Sicherheit deutlich auf. Es existieren drei zentrale Stellen der Kontrolle:

- Zugangskontrolle der Clients zu den Brokern,
- Autorisierung der Broker, gewisse Eventtypen empfangen zu dürfen und
- schlüsselbasierte Sicherung des Inhalts von Events anhand von Schlüsselklassen. Dies schließt die Erlaubnis für „publish“ und „subscribe“ dieser Inhalte ein.

Ideal wäre, wenn Publisher *A* eine Nachricht an Subscriber *B* schicken könnte, die nur der Subscriber lesen könnte. Dazu müsste *A* aber *B* kennen, was dem ganzen Konzept der eventgetriebenen, entkoppelten Verarbeitung widerspricht – auch wenn der nötige öffentliche Schlüssel von *B* über eine zentrale Schlüsselautorität leicht erhältlich wäre. Wenn eine End-To-End-Sicherung von Kommunikationspaaren nicht möglich ist, dann ist es logisch, Klassen von Bearbeitern von Nachrichten zu bilden. Das unten dargestellte Konzept verwendet dazu zwei Mecha-

Broker, die keine Schlüssel für diese Attribute besitzen, können dennoch die Eventtypen routen und sogar content-basierte Filterungen der nicht-verschlüsselten Attribute vornehmen.

Es gibt einige Dinge in dieser Sicherheitsarchitektur, die möglicherweise verbesserungsfähig wären. So sollten externe Autoritäten für Rollen- und Schlüsselverwaltung ebenfalls eventbasiert im gleichen Broker-Netz tätig sein. Es gibt hier nicht unbedingt einen Grund, separate Mechanismen einzuführen. Die Einführung neuer Eventtypen, deren Berechtigungen und Schlüssel könnte eventgetrieben geschehen, indem die Autorität die Events mitliest und gegebenenfalls automatisch Informationen, Schlüssel etc. postet. Theoretisch müsste dafür die a-priori-Ausstattung aller Teilnehmer mit einem Private Key sowie dem Public Key der Autorität genügen. Auch die Verwendung eines rollenbasierten ACL-Systems für Eventtypen und eines rollenbasierten Schlüsselsystems könnte sich als redundant erweisen. Auf Basis der Schlüsselklassen ließen sich auch die Rechte bezüglich der Events verwalten.

6.2 Problematik der Ende-zu-Ende-Sicherheit

Wir haben gesehen, dass wir es im Enterprise-Bereich heute mit komplexen, hochgradig verteilten Architekturen zu tun haben. Komplexe, verteilte Infrastrukturen laufen aber Gefahr, dass im Verlauf der Bearbeitung eines Kunden-Requests verschiedene Sicherheitszonen berührt und verschiedene Rechte eingesetzt werden. Kritische Punkte sind zum Beispiel der Wechsel einer Identität während des Aufrufs, denn damit sind meist veränderte Rechtevektoren verbunden.

In diesem Abschnitt soll am Beispiel einer Sicherheitslücke im Cisco-Portal die „Ende-zu-Ende“-Problematik verdeutlicht werden, also die Frage, wie ein Request von außen auf sichere Weise in die Infrastruktur gelangen und dort von unterschiedlichen Stellen authentisiert und autorisiert werden kann.

Im August 2005 musste die Firma Cisco der Öffentlichkeit mitteilen, dass durch einen Fehler (also nicht durch einen Angriff) im Portal Kundenpasswörter mithilfe der Suchfunktionalität des Portals ermittelt werden konnten. Die Passwörter waren nötig, um beispielsweise Patches für Cisco-Produkte herunterzuladen. Daraufhin musste Cisco alle Accounts sperren und allen Kunden neue Passwörter zusenden (vgl. [heise08_05]). Dieser Vorfall besitzt zwei interessante Aspekte:

- Die Passwörter waren offenbar unverschlüsselt bzw. nicht als Hash-Wert abgelegt.
- Die Suchfunktionalität ist für nicht-authentisierte Kunden zugänglich, der Download von Patches hingegen nicht.

Behandeln wir diese Aspekte nun der Reihe nach.

6.2.1 *Speichern von Passwörtern in Registries*

Wie sollten Passwörter überhaupt in Registries gespeichert werden? Wir gehen davon aus, dass es sich beim Speichern letztlich um das Ablegen von Daten in einer Datenbanktabelle handelt.

Zur Auswahl stehen die Alternativen Klartext, in gehashter oder in verschlüsselter Form. Welche Gründe gibt es für unverschlüsseltes Speichern? Gar keine, die Frage ist schlicht falsch gestellt und müsste lauten: Wieso benötigt ein Server überhaupt den Klartext eines Passwortes? Hierfür gibt es im Wesentlichen zwei Gründe: Einmal möchte man den Kunden im Falle eines vergessenen Passwortes das Originalpasswort zusenden können. Darüber hinaus gibt es, wie wir bereits gesehen haben, Authentisierungsverfahren in Form eines Challenge-Response-Protokolls, bei denen das unverschlüsselte Passwort in die Berechnung der Response eingeht. Das Zusenden eines vergessenen Originalpasswortes wird relativ häufig von Firmen angewandt, obwohl es ein sehr unsicheres Verfahren darstellt, nicht zuletzt aufgrund der Zusendung über unverschlüsselte Mail. Das Passwort kann dadurch bekannt werden, ohne dass der Besitzer dies ahnt. Wenn der Besitzer das gleiche Passwort mehrfach anwendet – was die meisten Benutzer tun – dann ist jetzt eine größere Zahl von Sites in Gefahr. Besser wäre es, ein neues, zeitlich begrenztes Passwort serverseitig zu generieren, anstatt das Originalpasswort zu versenden.

Besser ist die Begründung im Falle eines Challenge-Response-Verfahrens, wie es z. B. im relativ neuen TLS-PSK-Standard verwendet wird. Hier schickt der Benutzer eine Challenge, die mit seinem Passwort erstellt wurde an den Server, und der Server muss die Kenntnis des Passwortes beweisen. Der große Vorteil dieses Verfahrens liegt darin, dass ein Irrtum des Benutzers bezüglich seines Ziels (Phishing etc.) nicht dazu führt, dass sein Passwort verraten wird.

Das Modul im Server, welches die Response erzeugt, benötigt dabei das unverschlüsselte Passwort und stellt damit auch ein Sicherheitsrisiko dar. Allerdings muss die Response keineswegs auf vorgelagerten Servern erzeugt werden, sondern auch auf hochsicheren Backend-Systemen erzeugt und dann an den vorgelagerten Server weitergeleitet werden.

Ebenso bedeutet der Einsatz von Digest-Verfahren nicht, dass die Passwörter in jedem Fall unverschlüsselt in einer Registry liegen müssen. Stattdessen können die Passwörter von der Registry auf Anfrage dynamisch entschlüsselt werden. So speichert zum Beispiel die Sicherheits-Registry RACF auf IBM Mainframes-Passwörter nur verschlüsselt.

Wie sieht es mit dem Hashing von Passwörtern aus? Hier gilt wie immer beim Einsatz von Hash-Algorithmen: Es handelt sich um keine Verschlüsselung. Das bedeutet, wie bereits angesprochen: Wird die Datei mit den gehashten Passwörtern gestohlen, so kann der Angreifer eine Dictionary-Attacke durchführen. Dies ist bei korrekt verschlüsselten Passwörtern nicht so leicht möglich, da der Angreifer den verwendeten Key nicht kennt. Daher gehört beim Hashing von Passwörtern unbedingt ein „Salt“-Wert dazu. Dieser Wert liefert zusätzliche Entropie und wird zu-

meist im Klartext mit den Hash-Werten zusammen abgespeichert (für jedes Passwort ein anderes). Trotzdem wird damit eine Dictionary-Attacke erschwert, weil

- a) der Angreifer nicht auf vorher berechnete Hash-Werte zu den Dictionary-Einträgen zurück greifen kann, sondern selber rechnen muss.
- b) der Angriff für jedes Passwort erneut durchgeführt werden muss.

Bei der Verschlüsselung der Passwörter liegt die ganze Sicherheit in der Sicherheit des Keys, mit dem die Verschlüsselung durchgeführt wird. Dieser Schlüssel muss unbedingt geschützt werden, denn sein Verlust würde sämtliche Passwörter auf einen Schlag kompromittieren. Server werden diesen Schlüssel typischerweise in einer Smartcard geschützt verwalten.

In diesem Fall ist aber die Wahrscheinlichkeit, gehashte Passwörter zusammen mit dem Salt zu erbeuten, um ein Vielfaches größer als die Wahrscheinlichkeit, eine verschlüsselte Datenbank zusammen mit dem Key zu erbeuten. Ist aber der Schlüssel nicht bekannt, ist es so gut wie unmöglich, eine Dictionary-Attacke auf die Passwörter zu fahren (es sei denn, der Schlüssel selbst ist wiederum aus dem Passwort abgeleitet).

Bleibt festzuhalten dass das Hashing der Passwörter eine bessere Performance bringt und das Schlüsselpfand vermeiden. Die dabei entstehende Sicherheit ist im Wesentlichen eine Sicherheit für den Serveranbieter. Zieht man auch die Sicherheit des Benutzers in Betracht, erscheint die Verschlüsselung besser. da sie das versehentliche Verraten des Passworts durch Senden an den falschen Server verhindern kann, z. B. durch den Einsatz von TLS-PSK. Hier zeigt sich erneut, wie unterschiedlich Sicherheit beurteilt werden kann, je nachdem, ob sie unilateral oder multilateral gesehen wird [Rann] (ein Dankeschön an dieser Stelle an R. Schneider für die Diskussion zum Unterschied der beiden Verfahren).

Die Speicherung von Passwörtern im Klartext ist dagegen eindeutig minderwertige Sicherheit. Dass diese Sorge berechtigt ist, zeigen immer wieder Fälle, in denen dadurch erbeutete Passwörter für Angriffe auf weitere Systeme ausgenutzt werden – mit dem Wissen, dass viele Benutzer das gleiche Passwort für viele Systeme verwenden (Passwortsynchronisierung). Ein aktuelles Beispiel ist der Verlust einer Onlinebewerberdatenbank bei der Unternehmensberatung PriceWaterhouseCoopers, die sofort für Angriffe auf MoneyBookers und Click&Buy verwendet wurden. Laut dem Heise Newseintrag sollen ca. 50% der Benutzer darüber hinaus ihr PwC-Passwort auch für ihren Mailaccount verwendet haben [Heise09_08].

6.2.2 Nicht-authentisierter Zugriff durch den Suchdienst

Um zu klären, wie es dazu kommen konnte, dass Passwörter über das Such-Interface auf dem Portal öffentlich gemacht wurden, müssen zuerst einmal mögliche Architekturen des Systems untersucht werden. Eine mögliche Search-Architektur zeigt Abb. 6.7.

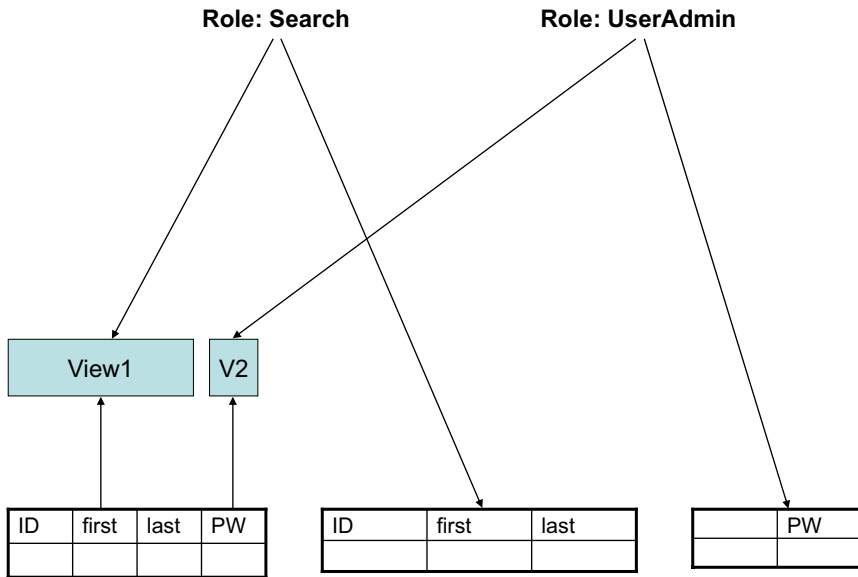


Abb. 6.8 Mögliche Tabellenorganisation und ihre Auswirkungen auf die Passwörter

Zusammenfassend bleiben folgende potenzielle Sicherheitsprobleme festzuhalten, die letztlich zu der fatalen Sicherheitslücke geführt haben können:

- Die Passwörter sind unverschlüsselt abgelegt.
- Die Passwörter sind Teil einer umfangreicheren Tabelle, die für mehrere Zwecke benötigt wird.
- Die Passwörter sind für die Datenbank nur einfache Daten, im Gegensatz zu den Passwörtern der Services, die als Datenbanknutzer registriert sind. Es gibt innerhalb der Datenbank keinen Hinweis auf die Besonderheit dieser Daten.
- Indizierservices erhalten Zugriff auf die Tabelle durch Zuweisung einer Rolle an einen Functional User (DB User). Dadurch ist nicht sofort sichtbar, welche Rechte auf welche Ressourcen dadurch tatsächlich zugestanden werden.
- Die Indizierservices filtern die Passwörter nicht. Falls doch, wäre das Wissen um die Besonderheit der Daten im Code der Services versteckt.
- Einmal ausgelesen, sind die Daten einfache Files und können von ihren Eigentümern (den Indizierservices) irgendwo abgelegt werden.
- Damit die Index-Files gelesen werden können, müssen Rechte (z. B. read-all) vergeben werden.
- Search Frontend Services greifen auf die abgelegten Daten zu. Im öffentlichen Fall hat sich ein Benutzer nicht authentisiert, und deshalb läuft der Service mit einer eigenen Functional UserID für nicht-authentisierte Services oder eine speziellen Rolle für solche Zwecke.
- Die Functional UserID der Frontend Services hat das Recht, auf die Index-Files zuzugreifen.

Im Kapitel „Enterprise Search Engine Security“ in Kriha/Schmitz, Internet-Security aus Software-Sicht [KS] findet sich eine Beschreibung der Sicherheitsanforderungen einer Search Eingabe sowie deren Architektur.

6.2.3 Zugriff durch Patch-Services

Der Registrierungs- bzw. Patch-Service benötigt ebenfalls Zugriff auf die Passworttabelle, falls die eingegebenen Kundenpasswörter gegen die gespeicherten geprüft werden müssen. Abbildung 6.9 zeigt die grundlegende Situation. Auch hier stellt sich die Frage: Wer erhält Zugriffsrechte und wie granular können die Rechte vergeben werden?

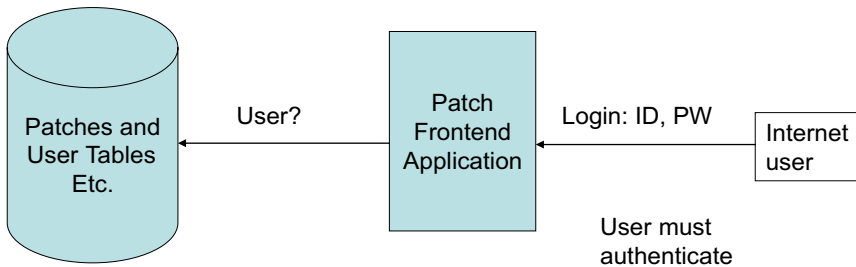


Abb. 6.9 Propagation des Zugriffs durch authentifizierte User

Anders als im Falle des Search-Interfaces oben haben wir im Falle des Patch-Services einen bereits authentifzierten User. Aber bedeutet das, dass die Zugriffe auf die Tabellen mit den Patch-Daten mit der Identität und den Rechten des jeweiligen Users erfolgen? Das würde voraussetzen, dass die User als Datenbank-User auf der Datenbank registriert sind. Nur so kann die Datenbank mit der Identität der User etwas anfangen.

Aller Wahrscheinlichkeit nach wird dies nicht der Fall sein, sodass der Patch-Service ebenfalls unter einer Functional UserID auf die Datenbank zugreifen wird. Wird das ein anderer User sein, als der für den Search-Service verwendete? Dies hängt nicht zuletzt von der Komponentenarchitektur des Gesamtsystems und seines Connection Pools für die Datenbankverbindung ab. Es ist extrem teuer, eigene Connections für jeden Nutzer zu erzeugen, sodass im Allgemeinen lediglich Functional User verwendet werden.

Der Registrierungs- bzw. Anmeldeservice erfolgt ebenfalls unter einer Functional UserID, da zu diesem Zeitpunkt noch kein authentifzierter User vorliegt.

Als Sicherheitsprobleme bleiben festzuhalten:

- Authentisierung (Vergleich des Passworts) sowie Auslieferung der Patch-Daten erfolgen unter künstlichen UserIDs.
- Die Services haben unter ihrer ID Zugriff auf die Tabellen.
- Zumindest bei der Auslieferung kann in weiten Teilen die Identität des jeweiligen Users protokolliert werden.
- Die Identität des Users wird auf Datenbankebene nicht ausgewertet.
- Die Kontrolle der Rechte des jeweiligen Users findet im Service statt.
- Der Service benötigt Wissen über die Rechte des jeweiligen Users.

6.2.4 Propagation von Identität, Recht und Aufruf

Der gerade beschriebene Vorfall stellt ein klassisches Beispiel für den GAU in einer Infrastruktur dar: Nicht-authentisierte User können Services aufrufen, für die sie nicht die nötigen Zugriffsrechte besitzen – und zwar *nach* einer durchgeführten Zugriffskontrolle (s. Abb. 6.10). Deutlich wird hier sichtbar, dass eine rein perimeterbasierte Sicherheitstechnik (also Eingangskontrolle in einen vertrauenswürdigen Bereich) alleine nicht ausreicht. Die Aufgabe der Ende-zu-Ende-Sicherheit ist es, solche Überschreitungen von Rechten zu verhindern. Die Mittel dazu sind:

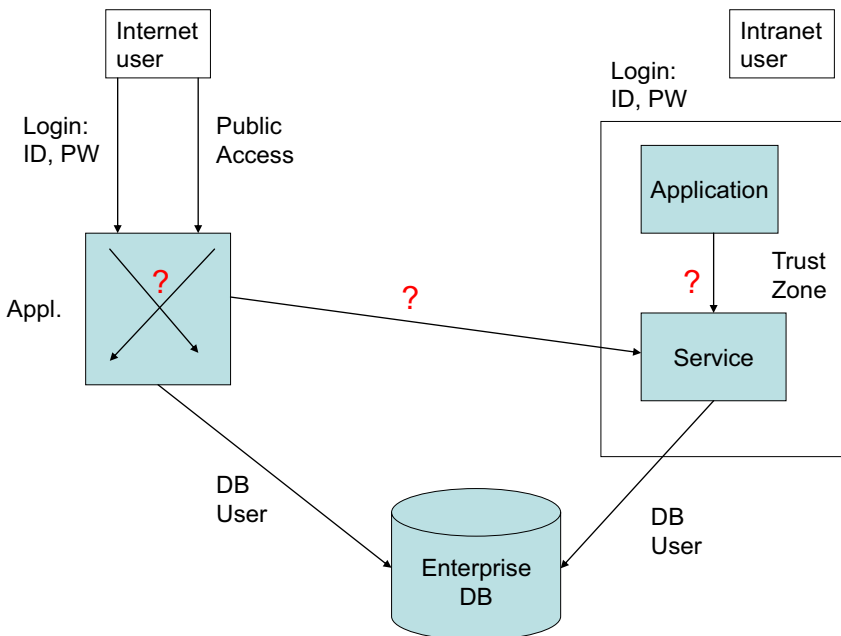


Abb. 6.10 Gefährliches Kreuzen von Requests

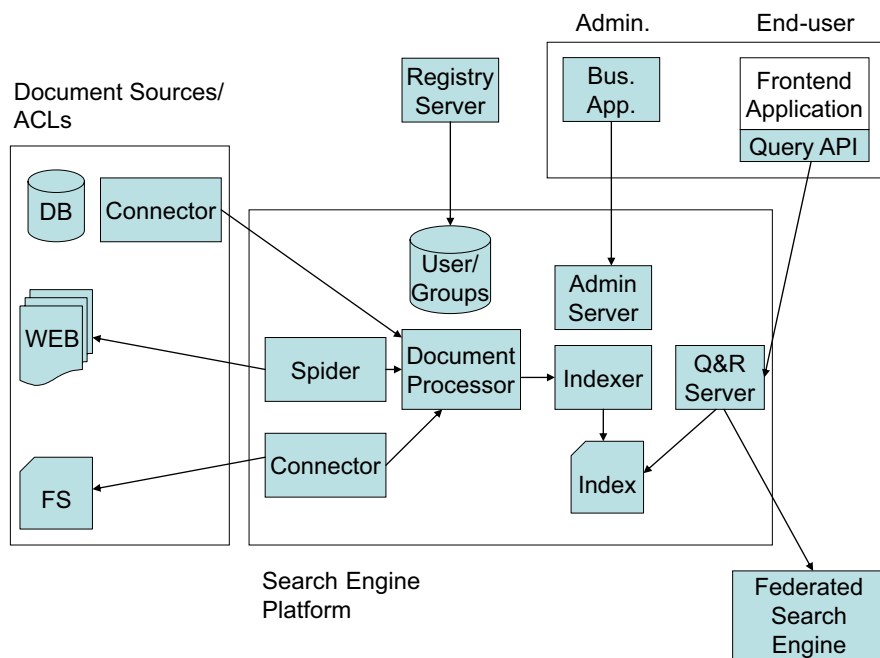


Abb. 6.11 Kanonische Enterprise-Search-Plattform

gegenseitige Authentisierung, Weitergabe von Identitäten und den dazu gehörigen Rechten sowie die Protokollierung des gesamten Ablaufs eines Requests.

In den folgenden Abschnitten werden wir nach technischen Lösungen für die Verwirklichung echter Ende-zu-Ende-Sicherheit suchen, und zwar auf Basis häufig eingesetzter Frameworks.

Um zu zeigen, wie subtil Sicherheitslücken entstehen können und welche Bedeutung vor allem auch die Zugriffskontrolle möglichst weit hinten bei den Backend-Systemen besitzt, noch ein Beispiel aus dem Bereich Enterprise Search. Abbildung 6.11 stellt eine typische Enterprise-Search-Plattform dar, wie sie Firmen wie Autonomy oder FAST herstellen.

Um zu verhindern, dass Search-User Zugriff auf Suchergebnisse erhalten, die sie normalerweise nicht sehen dürften, weil sie gar keine Zugriffsrechte für die Dokumente selbst besitzen, werden beim Extrahieren der Dokumente gleichzeitig die zugehörigen ACLs extrahiert und gemeinsam mit den Daten im Index der Search Engine gespeichert. Bei einem Search Request eines Users in einer Frontend-Applikation wird die Identität des Users auch an die Search Engine übermittelt – genauer gesagt an den für Queries zuständigen Server. Dieser greift nun auf den Index zu und führt einen Match durch zwischen den Anfragewörtern des Users und den Indexdaten. Gleichzeitig sorgt jedoch ein Sicherheitsmodul dafür, dass die Rollen des Users aus einem Repository extrahiert und die Anfragewörter des Users um diese Rollen erweitert werden. Das heißt, Indexein-

träge mit ACLs, die sich nicht auch als Rollen beim User finden, werden beim Abgleich ignoriert.

Neben dem eben beschriebenen Eingang zur Search Engine für User gibt es noch einen administrativen Eingang. Dieser erlaubt es, z. B. neue Collections – sprich Datenquellen – den einzelnen Frontend-Applikationen zuzuordnen sowie durch Boosting und Blocking die Resultate von Queries zu beeinflussen. Dieser Eingang ist nur für Administratorrollen zugänglich.

Man könnte nun auf die Idee kommen, die Zugriffskontrolle für User-Queries statt innerhalb der Search Engine gleich vorne in den Frontend-Applikationen zu machen. Das heißt, in den Frontend-Applikationen existiert Wissen darüber, welche Rollen welche Dokumente sehen dürfen. Der Mechanismus, der zu den Dokumentendaten auch die ACLs aus den Backend-Systemen extrahiert, ist dann nicht nötig.

Eine scheinbar kleine Optimierung bzw. Bequemlichkeit für die Entwicklung. Die Konsequenzen sind jedoch gewaltig. Passiert nun ein Fehler in der Administration, d. h. wird fälschlicherweise eine Datenquelle eines Backend-Systems einer Frontend-Applikation zugeordnet, deren User eigentlich keinen Zugriff haben dürften, so gibt es keinen Schutz dieser Daten mehr: Die Frontend-Applikation weiß ja nicht, dass nun zusätzliche Daten in der Search-Engine zur Verfügung stehen und kann daher den Zugriff nicht kontrollieren. Die Search Engine selbst jedoch gleicht die Dokumenten-ACLs nicht mehr mit den Rollen des Users ab und gibt dadurch die Daten frei.

Beim zuerst geschilderten Fall prüft hingegen die Search Engine selbst beim Zugriff auf den Index die ACLs gegen die Rollen des Aufrufers. Eine Fehlkonfiguration der Datenquellen durch Administratoren ist dadurch recht harmlos, denn die Rollen des Users passen nicht zu den ACLs der fälschlicherweise freigegebenen Datenquellen und der Request wird daher für diese Daten ignoriert.

Das Beispiel zeigt nochmals einige zentrale Erkenntnisse der Enterprise Security auf:

- Werden Daten der Sicherheit der Backend-Systeme entrissen, müssen vorgelagerte Instanzen die Sicherheit garantieren.
- Search Engines sind sehr problematische Architekturen, da sie die Daten grundsätzlich den Backends entreißen. Sie propagieren im Normalfall auch nicht die Identität des Users an die Backends, da die Datenextraktion, d. h. das Indexieren der Daten, ohne Beteiligung von Usern außerhalb der Laufzeit von Suchanfragen stattfindet. Normalerweise werden Functional UserIDs für den Lesezugriff der Search Engine auf die Backends verwendet.
- Die Verlagerung von Zugriffskontrolle „nach vorne“ macht teilweise Sinn, z. B. aus Performancegründen, oder um unsinnige Requests früh blocken zu können. Im Allgemeinen ist es jedoch wünschenswert, dass

auch die nachgelagerten Systeme, speziell die Datenbesitzer, eine Zugriffsprüfung anhand der propagierten Identität des echten Aufrufers durchführen können. Sobald diese User-Identität durch funktionale Identität, z. B. der eines Servers, ersetzt wird, fallen Kontrollmöglichkeiten auch gegen fehlerhafte Konfigurationen weg und der Zugriff auf die Daten wird zu breit.

6.3 JAVA EE und EJB

Die Enterprise-Java-Architektur überlässt vieles im Bereich Security den Herstellern von Containern und Application-Servern. Dabei folgt sie dem Grundprinzip, dass Security etwas ist, das am Besten von Applikationen ferngehalten wird – und zwar aus Gründen der Wartbarkeit, aber auch um sicherzustellen, dass sicherheitsrelevante Prüfungen nicht vergessen werden können. Bezogen auf die Security in Enterprise-Java-Applikationen findet man daher die folgenden Ausprägungen:

Keine/selbstgebaute Security: Die Güte der Authentisierung von Authentisierung oder Delegation bleibt den Application-Servern und der System-Security überlassen. Das Gleiche gilt für das Principal Management.

Deklarative Security: Annotationen im Code oder Deployment-Descriptoren regeln Rollendefinitionen, Permissions, Transportsicherheit und die Art der Authentisierung. Die Ausführung und Überwachung bleibt wieder dem Systemmanagement überlassen.

Programmatische Security: Authentisierung und Autorisierung basieren auf dem Zugriff auf Principal-Attribute (Name, Credentials) sowie dem Abfragen von Rolleneigenschaften (was wiederum nur eine Form von Credentials darstellt).

Das ursprüngliche Modell von Sicherheit in Enterprise-Applikationen baut also ganz stark auf einer System-Level-Security auf, wie sie z. B. in CORBA definiert wurde. Transportsicherheit, Weitergabe von Credentials durch Delegation und ein rollenbasiertes Sicherheitskonzept waren dabei die Grundpfeiler. Im Grunde genommen handelt es sich um die Definition eines Systemkontextes, der Aufrufe begleitet und der die Sicherheit der Aktionen bestimmt. Große Bedeutung kommt in so einem Konzept natürlich der Authentisierung des Aufrufers sowie der Etablierung und Delegation dieses Kontextes zu. Brüche in der Delegation oder im Sessionkontext stellen Sicherheitslücken dar.

Von einer höheren Warte aus gesehen, entspricht dieses System den Versuchen, Sicherheit einzuführen, indem man Sicherheitsmechanismen auf niedrigeren Protokollschichten in einem Netzwerk einführt: IPSec bzw. IPv6 koppeln Systeme durch eine gemeinsame Sicherheitsschicht unterhalb der Transportschicht, SSL

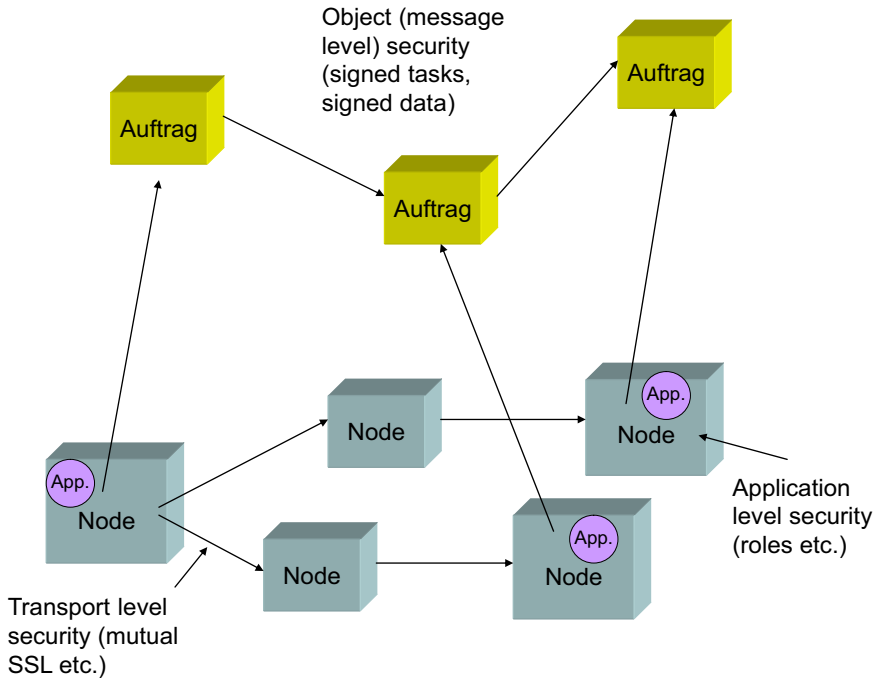


Abb. 6.12 Verhältnis von kanalspezifischer, objektspezifischer und programmatischer Security

verbindet Applikationen direkt durch eine abgesicherte Session. Beide Sicherheitsprotokolle können jedoch nicht die Sicherheit der Daten außerhalb der Übertragung garantieren. Analog kann auch die sichere (und aufwändige) Delegation durch Weitergabe von Credentials nur dafür sorgen, dass die Aufrufpfade kontrollierbar sind, nicht jedoch für die Sicherheit der dabei manipulierten Daten im Sinne eines Auftrags garantieren.

Mittlerweile ist jedoch mit der sog. objekt- oder messagebasierten Sicherheit eine neue Form der Security eingeführt worden, entstanden aus der Unzufriedenheit mit kanalspezifischen bzw. rein infrastrukturbasierten Sicherheitsformen. Auch wenn die Applikationen selbst keinerlei Sicherheitscode beinhalten, ketten diese Formen die Applikationen dennoch indirekt an die Infrastruktur (s. Abb. 6.12). Die Unflexibilität entsteht dadurch, dass gewisse Aufrufwege als „sicher“ definiert sind – aber was geschieht, wenn neue Partner in kurzer Zeit auftauchen? Das Entfernen von Sicherheitscode aus Applikationen bedeutet also nicht, dass sie damit im Geschäftssinn flexibler geworden sind.

Zu dieser Problematik eine kleine Analogie: Die sicherheitstechnische Bindung an die Infrastruktur entspricht in etwa der Situation, dass ein bestimmter Rechner aus Sicherheitsgründen nur an einer einzigen Netzwerkdose betrieben werden kann, d.h. der Rechner kann nirgendwo sonst an das Netzwerk angeschlossen werden. Zwar mag dies in Einzelfällen auch heute noch sinnvoll sein, schränkt aber die Möglichkeiten mobilen Arbeitens drastisch ein. Die Sicherheit ist hier

einfach auf einem zu tiefen Netzwerklayer realisiert worden. Generell gilt in verteilten Systemen: Je tiefer die Schicht liegt, auf der ein Mechanismus implementiert wird, umso bequemer und sicherer lässt er sich gestalten, aber gleichzeitig schränkt er auch die Möglichkeiten darüber liegender Schichten oder Applikationen drastisch ein. Objektbasierte Sicherheit geht den genau umgekehrten Weg: Mit objekt- bzw. messagebasierter Sicherheit hängt die Sicherheit durch Signaturen und Verschlüsselung direkt am Objekt. Applikationen können so granulare Geschäftsmodelle nutzen, ohne auf die darunter liegende Sicherheitsinfrastruktur Rücksicht zu nehmen. Der Preis dafür ist, dass die Sicherheit der Daten wieder in den Bereich der Applikation und damit der Applikationsprogrammierer zurückkehrt und dort für mehr Komplexität sorgt.

Natürlich kann die objektbasierte Sicherheit auch einfach als interoperabler Ersatz für bestehende Implementationen genutzt werden, bringt dann jedoch ihre Vorteile auf semantischer Ebene nicht zum Ausdruck.

6.3.1 Grundprinzipien sicherer Software in JAVA EE und EJB

Eine detaillierte Darstellung der JAVA-EE- und EJB-Sicherheitskonzepte findet sich für verschiedene Hersteller von Application-Servern in den Büchern zu Websphere ([KNNP]), BEA Weblogic ([BEA]), JBOSS ([JBOSS]) sowie natürlich im JAVA EE Tutorial von Sun ([J2EE], Chapt. 32).

Da die JAVA-EE-Spezifikation bezüglich Security relativ viele Aufgaben an die Hersteller von Application Servern bzw. Containern delegiert, beschränkt sich die Aufgabe der Entwickler von Applikationen oder Komponenten meist auf die deklarative Angabe von Sicherheitsanforderungen.

Hier sind, kurz zusammen gefasst, die Prinzipien der JAVA EE Security:

- Die Zugriffskontrolle auf Ressourcen erfolgt über die Rolle(n) des Aufrufers.
- Identitäten werden über Komponenten hinweg delegiert, um jederzeit Informationen über den Aufrufer auswerten zu können. Bevorzugt wird die deklarative Spezifikation von Sicherheitsanforderung in Applikationen.
- Der programmatische Zugriff auf Identität und Rollen aus dem Code heraus muss möglich sein.
- Sicherheitsanforderungen müssen sich auf den jeweiligen Firmenkontext einfach anpassen lassen, ohne Änderungen am Applikationscode vornehmen zu müssen (Prinzip der „Separation of Concerns und Context“).
- Der Anschluss von Legacy-Applikationen oder Datenbanken muss möglich sein.
- Es wird im Allgemeinen ein dreischichtiges Modell, bestehend aus Web-Komponente (Login), EJB-Komponenten (Business Logik) und Backend-Systemen (CICS, Datenbanken) unterstellt.

Wir werden im Folgenden die Konsequenzen dieser Grundprinzipien auf das Design sicherer JAVA-EE-Applikationen untersuchen.

6.3.2 JAVA-EE-Applikationsinterfaces und ihre Verwendung

Die meisten Applikationen sind nicht wirklich an Sicherheitsfunktionen interessiert, sondern nur an den Ergebnissen, die diese liefern, also an Begriffen wie Subject, Credentials, oder Principals.

Abbildung 6.13 zeigt ganz grob die Funktionen einer Server-Applikation. Die gewünschten Interfaces lassen sich auf die drei Bereiche Eingangsprüfung, Ausführung und Transport (Forwarding, Delegation, Weiterleitung) gliedern. Die folgende Tabelle zeigt, wie sich die von JAVA EE angebotenen Funktionen auf diese Bereiche verteilen.

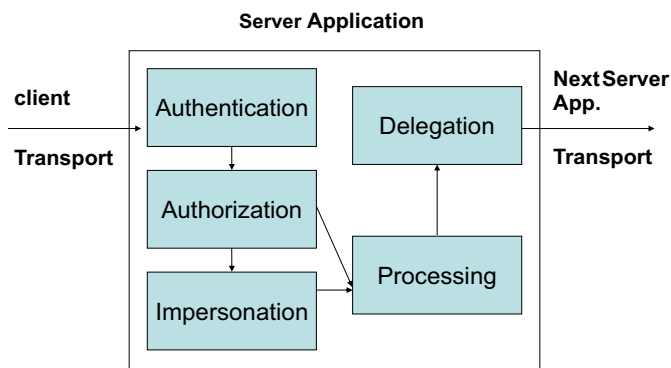


Abb. 6.13 Sicherheitsfunktionen bei Server-Applikationen

Processing Location	JAVA EE Interface	Purpose
Transport	web.xml	Stellt sicher, dass Daten entsprechend ihrer Daten-Klassifikation übertragen werden
Entry	web.xml, ejb.xml config Files: Definition von Container-Objekten als Eingang	Festlegung, durch welche Eingänge ein Request die Applikation erreichen kann. Gilt für Web-Komponenten wie Businesslogikkomponenten (EJB)
Entry	web.xml: deskriptive Bestimmung der nötigen Authentisierung	Erlaubt der Applikation, die Qualität der Authentisierung zu bestimmen. Nur im Web-Container.
Entry	web.xml, ejb.xml: applikationsspezifische Rollen, deklarativ	Applikation legt intern fest, welche Rolle ein Request besitzen muss, um auf eine Ressource (Web-URL oder EJB-Komponente) zugreifen zu können
Entry	??.xml: enterprise-spezifische Rollen, deklarativ. Syntax zwischen Web und EJB unterschiedlich	Erlaubt während der Installation die Anpassung applikationsinterner Rollen an vorhandene firmenspezifische Rollen

Processing Location	JAVA EE Interface	Purpose
Verarbeitung	Programmatisch: Principal/Subject können abgefragt werden. Syntax zwischen Web und EJB unterschiedlich	Der Applikationscode trifft Entscheidungen anhand von User-Identität und Rolle. Klassisch nötig zur Implementation instanzspezifischer Sicherheit, zum Beispiel soll nur der Eigentümer eines Kontos Transaktionen vornehmen können.
Verarbeitung	Programmatisch: Subject.doAs, doAsPrivileged Interface zur Impersonation	Applikation kann bestimmte Codepassagen unter der Identität des Aufrufers oder einer anderen ausführen lassen. Dazu wird ein anderes Subject Object mit anderen Zugriffsrechten (Access-ControlContext) an einen Thread gebunden, der dann den Code ausführt.
Weitergabe/ Delegation	Deskriptive Bestimmung der Identität, unter der die Applikation Aufrufe an andere Dienste oder Komponenten durchgeführt haben will: Run-as-Element	Der Container benutzt diese Identität zum Aufbau von Verbindungen zu Backends oder Komponenten. Es gibt eine spezielle Version für JCA-Ressource-Manager.
Weitergabe	Programmatische Bestimmung der Identität, unter der die Applikation Aufrufe an andere Dienste oder Komponenten durchführen will. Benötigt Credentials für die zu verwendende Rolle	Die Applikation authentisiert sich selber gegenüber anderen Ressourcen-Managern. Bei EJBs??

Es wird empfohlen, dass Applikationen in erster Linie die deklarativen Möglichkeiten von JAVA EE nutzen sollten. Allerdings ist damit zum Beispiel keine instanzbasierte Sicherheit möglich. Dazu gibt es zwar einige Vorschläge, wie die Ableitung eines eigenen Permission-Objektes, das auch Instanzwerte berücksichtigt, die dann in Policy-Files oder Repositories eingetragen werden müssen. Es existiert jedoch noch kein Standard in diesem Bereich.

Weitere, teilweise hässliche Probleme mit der jetzigen Version von JAVA EE bestehen in der Tatsache, dass die Rollen als Principals definiert sind (.NET leidet im Web-Bereich ebenfalls unter Mechanismen, die die Eintragung aller User verlangen) und dass die definierten Rollen innerhalb einer Applikation nicht durch ein Standard-Interface programmatisch abgefragt werden können. Ein weiteres Problem ist die Kombination von applikations- und enterprise-spezifischen Rollendefinitionen in den Deployment-Deskriptoren der Applikationen. Bei der Erstinstallation kann die Enterprise-Infrastruktur (z. B. ein LDAP-Server) auf die Rollen angepasst werden – aber was passiert, wenn die Applikation ein Update erhält und nochmals installiert werden soll? Wenn innerhalb der Unternehmung keine Änderungen bezüglich der Rollen und Rechte durchgeführt wurden, ist das Update kein Problem. Ansonsten stellt sich die Frage, ob die Definitionen innerhalb der

Applikation diejenigen im Enterprise-Verzeichnis überschreiben sollen oder nicht? Noch schwieriger wird es, wenn sich sowohl Applikation wie auch Umgebung in Bezug auf die Sicherheitsdefinitionen geändert haben.

Applikationen, die die oben beschriebenen deklarativen und programmatischen Interfaces benutzen, sind vor Änderungen der Sicherheitslogik im Business sowie bei den Sicherheitsmechanismen innerhalb eines Unternehmens zwar nicht gänzlich geschützt, aber der Anpassungsaufwand ist eingegrenzt und recht klar erkennbar.

6.3.3 Rolle, Permission und Ressource

Das Verhältnis von Rolle, Permission und Ressource, in Abb. 6.14 am Beispiel eines EJB-Modules dargestellt, zeigt, dass die eigentliche Sicherheits-Policy für den Zugriff erst auf Modulebene konfiguriert wird. Die einzelne Bean braucht bei der Erstellung durch den Bean Developer keine Rollen, es sei denn, dass sie intern Rollen durch die Verwendung der `isUserInRole(role)`-Methode nachfragt. Dann deklariert der Entwickler diese interne Verwendung und macht sie dadurch sichtbar für den Applikationsentwickler.

Der Applikationsentwickler entwirft auf Modulebene globale Sicherheitsrollen. Jetzt kann es natürlich sein, dass während der Entwicklung der einzelnen Beans diese Rollen noch gar nicht bekannt waren. Der Bean-Entwickler ist jedoch frei,

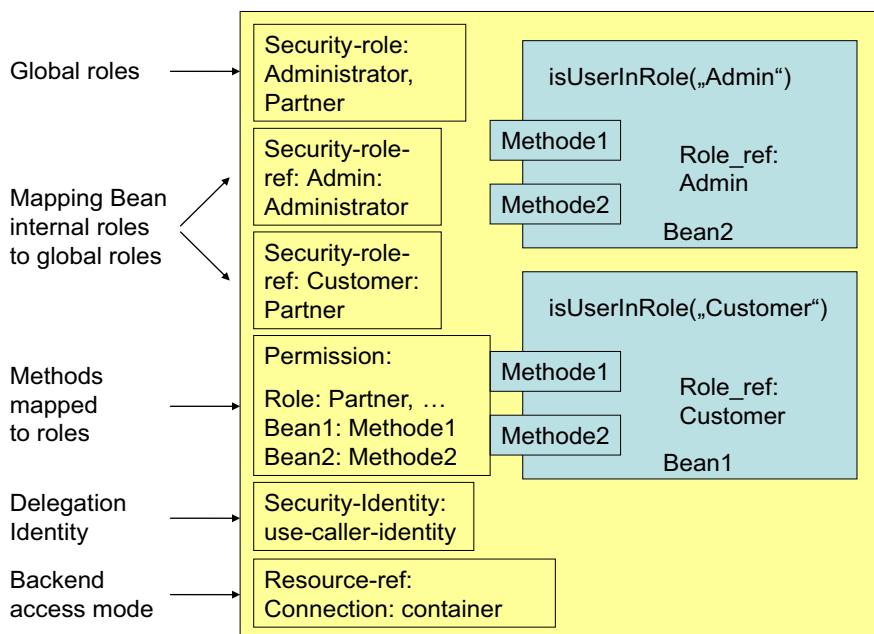


Abb. 6.14 Identitäten und Rollen in einem EJB-Modul

eigene Rollen zu definieren. Diese werden dann vom Applikationsentwickler auf einer der globalen Rollen abgebildet.

Jetzt muss nur noch im EJB-Modul die jeweilige globale Rolle mit den entsprechenden Permissions für Methodenzugriffe versehen werden. Jede Rolle erhält dabei eine Liste von Permissions, die wiederum Methodenzugriffe auf Beans erlauben. In [KNNP] wird diese Liste als *JAVA EE:Method-Permission-Table* bezeichnet.

Wichtig ist, dass die sogenannte *Security-View* – also ein Gesamtkonzept von Zugriffssicherheit – erst auf der Ebene des Moduls entsteht, denn dort werden die globalen Rollen definiert und mit Methodenzugriffen verknüpft.

Das Element *Resource-ref* definiert, wie auf Backend-Systeme mit eigener Authentisierung zugegriffen werden soll. Datenbanken und Applikationen auf Mainframes besitzen meist eine eigene Authentisierung und Autorisierung, und so nützt die Identität des ursprünglichen Aufrufers hier nichts. Entweder kann ein Functional User im Connector definiert werden und die Aufgabe der Authentisierung gegenüber dem Backend dem Container überlassen werden (die empfohlene Methode), oder die Applikation kann mithilfe eines eigenen Mappings die Identität des Aufrufers in eine auf dem Backend bekannte Identität umwandeln.

Die Verwendung eines Functional Users führt allerdings oft dazu, dass ein sehr breiter Zugriff auf Backend-Daten gestattet wird und die Sicherheit dieser Daten ausschließlich von der Applikation gewährleistet werden kann. Wird hingegen für

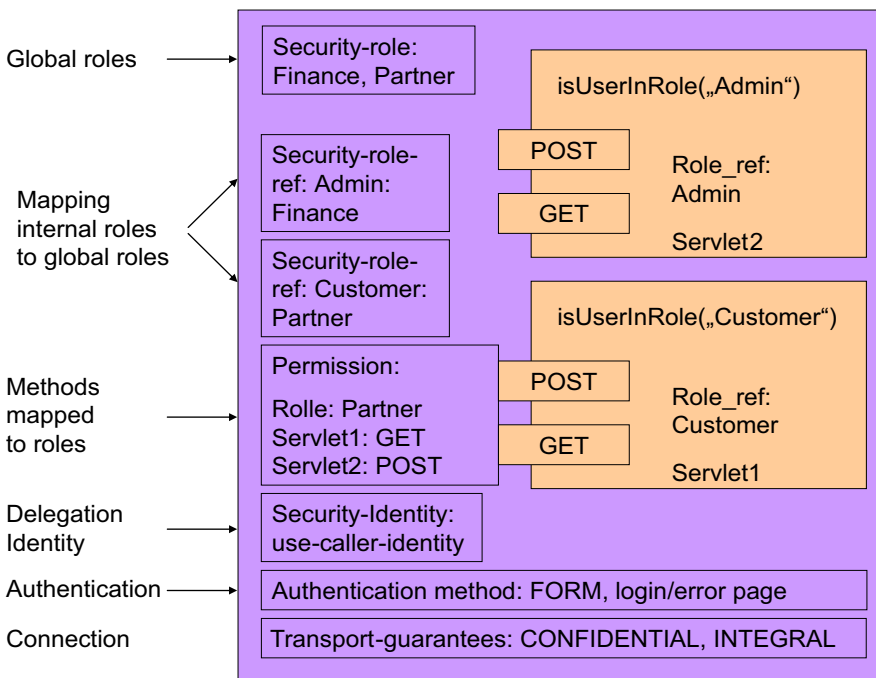


Abb. 6.15 Authentisierung, Identitäten und Rollen in einem Web-Modul

jede Verbindung ein spezieller User verwendet, dann entstehen Performanceprobleme, da diese Verbindungen dann nicht effizient in einem Pool verwaltet werden können. Sie müssen ja für jeden User neu und speziell aufgebaut werden. Abhilfe können hier nur Protokolle wie CSIV2 (oder SASL) schaffen, sofern sie auf den Backendsystemen unterstützt werden.

Die Sicherheitsdefinitionen innerhalb eines Web-Moduls sind bis auf Unterschiede in der Namensgebung fast identisch – mit Ausnahme der Bereiche Log-in und Transport von Daten (s. Abb. 6.15). Hier gibt es für Web-Module zusätzliche Angaben, da diese im Allgemeinen in einer Multi-Tier-Architektur die Authentisierung übernehmen.

Wichtig ist die Tatsache, dass das Web-Modul deklarativ die Art und Weise der Authentisierung des Aufrufers bestimmen kann. JAVA EE bietet hierfür verschiedene Verfahren an, wobei die Methoden BASIC (Username/Passwort) und FORMS die am meisten verwendeten sind. Zusätzlich ist auch eine zertifikatsbasierte Authentisierung möglich. Im Endeffekt bedeutet dies, dass innerhalb eines Servlets z.B. die Applikation Informationen einfach über `request.getUserPrincipal()` erhalten kann. Die eigentliche Authentisierung findet durch den Container bzw. Application-Server mithilfe von JAAS-Log-in-Modulen statt, wie sie oben beschrieben wurden.

Die bisherige Konfiguration der Applikation beschränkte sich auf die Definition von logischen Rollen und Rechten, hat aber noch keine Verbindung zu echten Identitäten (Principals). Diese Verbindung herzustellen, ist Aufgabe des Deployments und geschieht deklarativ auf der Ebene der gesamten Applikation (d. h. im EAR File).

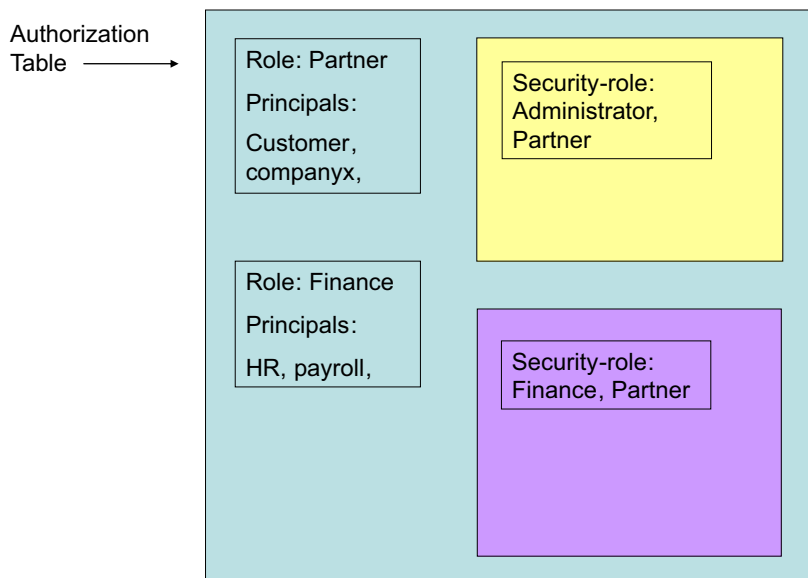


Abb. 6.16 Abbildung von Rollen auf Identitäten in der Applikation

In dieser Liste (sog. Authorization Table) werden die logischen Rollen auf konkrete, in der Firma existierende Identitäten abgebildet (s. Abb 6.16). Diese Identitäten können auch Gruppen von Nutzern sein. Damit kann der Container nun bei einem Aufruf eine identitätsbasierte Zugriffskontrolle durchführen, indem die Rolle(n) des Principals mit der verlangten Rolle des Servlets oder Beans verglichen wird.

6.3.4 Delegation und Impersonation

Delegation bedeutet die Weiterleitung eines Aufrufs an weitere Intermediates oder Targets, sodass der Empfänger die Tatsache der Weiterleitung feststellen und die Zugriffskontrolle an der Identität des ursprünglichen Aufrufers ausrichten kann. Die Enterprise Java Beans Specification ist an dieser Stelle jedoch sehr unpräzise: Sie überlässt die Details der Delegation den Herstellern des Containers bzw. des Application-Servers.

Der Aufruf von `getCallerPrincipal()` wird im Allgemeinen so verstanden, dass es sich hier um die Identität des ursprünglichen Aufrufers und nicht um die eines Intermediates handelt. Dafür gibt es jedoch keinerlei Garantie. Es könnte auch die Identität des direkten Aufrufers des Beans (Intermediate) zurückgegeben werden. Es lohnt sich deshalb, die Abläufe bei der Authentisierung bzw. Autorisierung sowie bei der Verarbeitung eines Aufrufs innerhalb einer EJB einmal genauer anzusehen (Abb. 6.17).

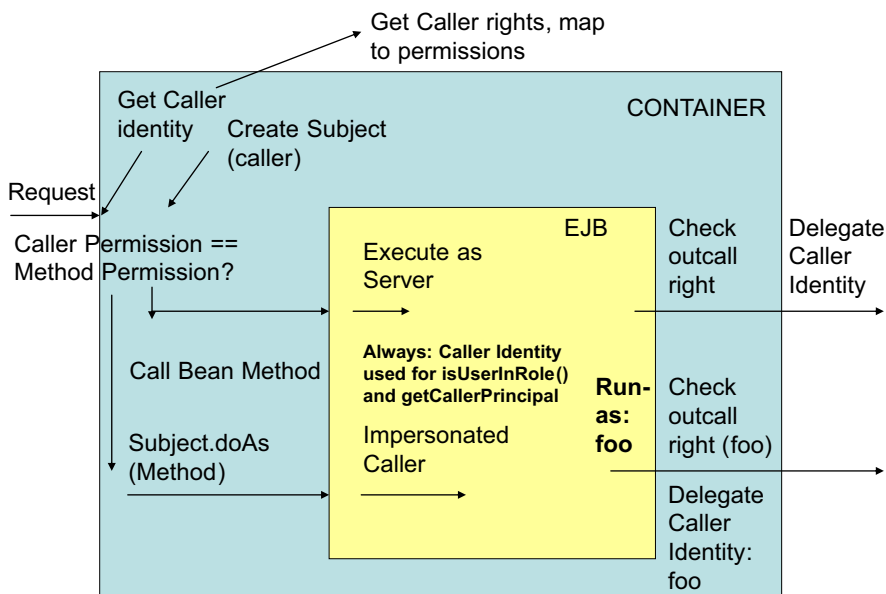


Abb. 6.17 Fluss der Identität eines Aufrufers durch einen Container

Wenn der Container einen Aufruf für eine EJB empfängt, dann wird er zunächst versuchen, die Identität des Callers festzustellen, d. h. die Identität, deren Rechte für die Autorisierung des Zugriffs auf ein Bean verwendet werden soll. Die EJB Specification verlangt, dass diese *Autorisierungsidentität* über verschiedene Maschinen weiterfließen kann.

Die *Authentisierungsidentität* kann dagegen die des letzten Intermediates sein. In diesem Fall übermittelt der Intermediate lediglich die Autorisierungsidentität an den nächsten Empfänger. Oder beides fällt zusammen und es wird eine neue Authentisierung und Autorisierung einer neuen Identität durchgeführt. Insgesamt ergeben sich folgende Identitäten:

- Authentisierungsidentität (letzter Aufrufer),
- Autorisierungsidentität (letzter Aufrufer oder „erster“ Aufrufer),
- Prüferidentität (z. B. laufender Server bzw. Container),
- Ausführungsidentität (entspricht entweder der Autorisierungsidentität, was Impersonation bedeutet, oder ist eine fremde Identität, abhängig vom Container) und
- Weitergabeidentität (eine konfigurierte Identität).

Wie diese Identität festgestellt wird, hängt vom Implementationslevel des CSIv2-Protokolls ab. Die EJB Specification kennt zwei Einstellungen zur Delegation: Use-Caller-Identity (Default) oder run-as sowie zwei grundsätzlich verschiedene Ziele eines ausgehenden Aufrufs: Das nächste EJB oder ein EIS/DB als Target (s. Abb. 6.18).

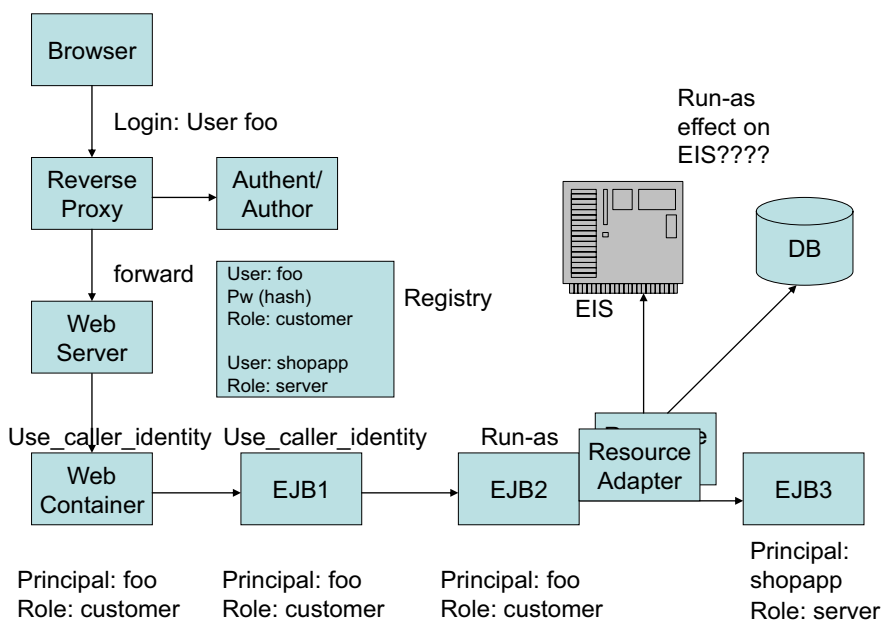


Abb. 6.18 Identitätspropagation in EJB

Zur Bedeutung und Wirkung der `run-as`-Einstellung in EJB finden sich verschiedene und widersprüchliche Darstellungen in der Literatur. So wird teilweise behauptet, dass ein mit diesem Tag ausgezeichnetes Bean mit den unter `run-as` angegebenen Rechten ausgeführt wird. Dies wäre dann als Impersonation zu verstehen, wie oben im Rahmen der Java-Plattform diskutiert. Dies ist jedoch falsch: Der Code des Beans wird nicht mit der `run-as`-Identität (also der Rolle) ausgeführt. Lediglich beim Aufruf anderer Beans wird zur Prüfung, ob der Aufruf autorisiert ist, der `run-as`-Wert genommen und nicht die Rolle des Aufrufers. Außerdem erhält das aufgerufene Bean die Identität des mit der `run-as`-Einstellung gekoppelten Principals mitgeteilt. Dort wird diese Identität dann als `Caller-Identity` zur Autorisierung verwendet. Das Ergebnis ist ein Identitäts- und Rechtewechsel ähnlich dem `AccessController.doPrivileged()`-Call im Rahmen der Java 2 Security. Auch ändert sich die Identität des Aufrufers im aufrufenden Bean nicht. Sie bleibt weiterhin voll erhalten und bei einer Abfrage der Identität wird die des Aufrufers zurückgegeben. Bei der Frage `isCallerInRole` (Rolle) wird die `run-as`-Identität nicht geprüft. Technisch gesehen bedeutet das `run-as`-Tag lediglich, dass der Call Stack des aktiven Threads eine weitere Rolle erhält, die für spezielle Aufrufautorisierungen verwendet wird.

JBoss verwendet die `run-as`-Identität auch, um die Rechte des Codes zu verringern: So kann man dort interne Beans, die vor Zugriffen geschützt werden sollen, durch eine spezielle Rolle schützen. Wenn jetzt während der Verarbeitung eines Aufrufs ein solches internes Bean verwendet werden soll, kann es nur mit der unter `run-as` angegebenen Rolle aufgerufen werden. [JBoss]. Der Aufruf einer weiteren EJB (downstream call) sollte also normalerweise mit der empfangenen `Caller-Identity` ausgeführt werden.

Anders sieht es meist beim Aufruf eines Backend-Systems aus. Das liegt daran, dass viele Backend-Systeme (meist Applikationen oder Datenbanken auf Mainframes) eine andere Domain bzw. Realm bilden – im Sinne von JAVA EE also eine andere Authentisierungs- und Autorisierungsinstanz bilden. Dies bedingt meist ein Mapping von Identitäten aus der verteilten JAVA-EE-Welt auf die dort im Backend bekannten Identitäten.

Der Aufruf eines über einen Java Connector angeschlossenen Ressourcen-Managers (EIS oder DB) kann unter EJB ebenfalls deklarativ oder programmatisch vom Bean-Entwickler gesteuert werden. Das entscheidende Element der Konfiguration ist hier `res-auth`: Mit dem dortigen Wert (Container oder Applikation) legt man fest, wer für die Authentisierung gegenüber dem Backend-System verantwortlich ist: Der Container oder die Applikation. Im Falle der Applikation wird sie dann bei der Eröffnung einer Verbindung zum Backend-System Credentials für die Authentisierung mitgeben müssen, eventuell unter Verwendung eines Credential-Mapping-Systems (vgl. Abb. 6.19).

Beim Setting „Container“ ist es den Systemmanagern überlassen, mithilfe der Werkzeuge des Containers und des Ressourc adaptors die entsprechenden Credentials zu erstellen, die eine Authentisierung gegenüber dem Backend erlauben.

Die applikationsgesteuerte Authentisierung gegenüber dem Backend hat jedoch gravierende Nachteile:

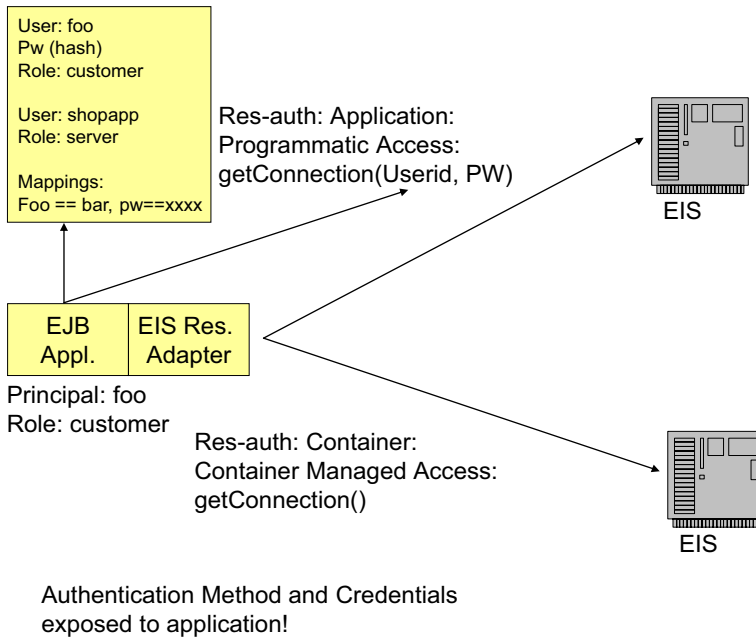


Abb. 6.19 Unterschiedliche Werte für `res-auth` bei Aufruf eines Ressourcen-Managers

- Kenntnis der Applikation von den Authentisierungsmechanismen des Backends.
- Kenntnis der Applikation von Credentials sowie deren Verwaltung.
- Probleme beim Deployment in Data-Center-Umgebungen, da die Credentials nicht von den Entwicklern verwaltet werden dürfen. Die Credentials dürfen also nicht im Code gehalten werden (was bei Security-Policies mit Passwort-änderungen ohnehin Unsinn ist).

Das Mapping einer Caller-Identität kann entweder auf eine andere Identität des gleichen Nutzers erfolgen (JohnB ist BJohn auf dem Mainframe) oder aber – und das ist wesentlich häufiger der Fall – auf einen künstlichen Nutzer, der auf dem Mainframe Rechte an Daten und Applikationen besitzt (s. Abb. 6.20). Er kann dabei OS-User sein oder auch nur für Datenbanken oder Applikationen berechtigt sein.

Wichtig für das Design von JAVA-EE/EJB-Applikationen ist an dieser Stelle, dass die Backend-Sicherheit das gesamte Design dominiert: Je nachdem, welche Nutzer dort bekannt sind, müssen die entsprechenden Mappings auf den Application-Servern der vorgeschalteten Systeme vorgenommen werden. Es macht daher Sinn, bei Projekten das Systemdesign, speziell das des operationalen Modells, vom Backend her zu entwerfen.

Abbildung 6.21 beschreibt drei mögliche Identitäts-Mappings zwischen der JAVA-EE-Domäne und der Legacy-Domäne in einem EIS. Der unterste Fall zeigt ein Mapping vieler Nutzer in der verteilten Domain auf einen funktionalen User in der Enterprise Domain. Dies passiert sehr häufig, vor allem wenn die User selber nicht als Identitäten innerhalb der Enterprise Domain registriert sind. Mit diesem

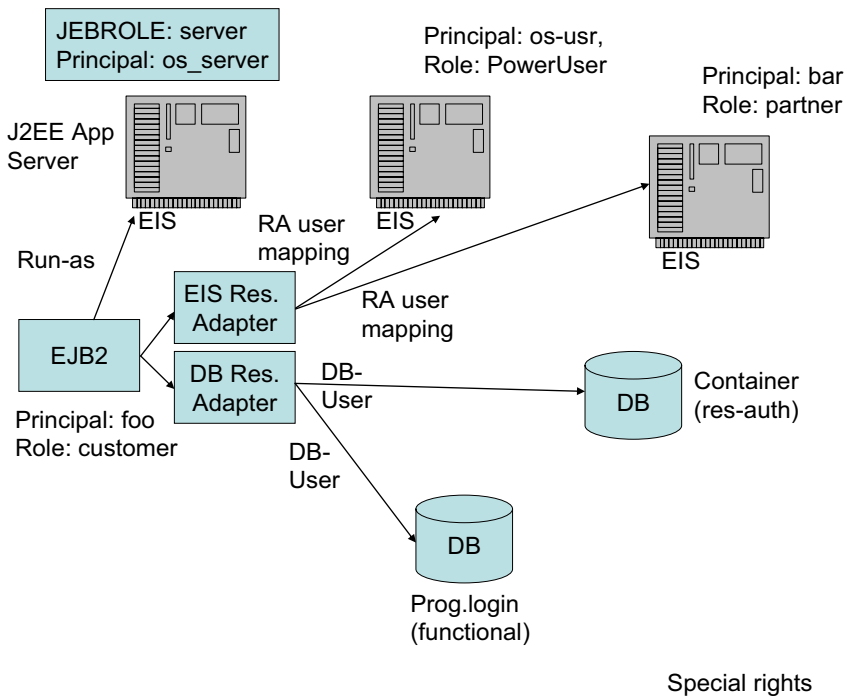


Abb. 6.20 Mapping auf Identitäten im Backend

Ansatz sind jedoch auch gravierende Nachteile verbunden: Das Enterprise-System verliert die Möglichkeit der eigenen Autorisierungsprüfung und muss dem funktionalen User sehr breite Zugriffsrechte einräumen, denn dieser handelt ja für alle User der verteilten Domain. Gleichzeitig leidet die Nachvollziehbarkeit stark: Das Enterprise-System erfährt nicht, für welchen User der funktionale User in Wirklichkeit handelt. Das Logging und Auditing kritischer Aktionen wird also behindert.

Die mittlere Lösung im Diagramm zeigt ein 1:1-Mapping jedes Nutzers der verteilten Domain auf einen User der Enterprise Domain. Solche Mappings können auf der Client-Seite passieren, d. h. der Ressource-Adapter bzw. Container führt das Mapping durch. Dazu benötigt er die Credentials der jeweiligen User, die er über einen Mapping-Service erhalten kann. Alternativ können diese Mappings auch auf der Seite des EIS vollzogen werden: Der Ressource-Adapter öffnet eine Verbindung zum EIS und authentisiert sich. Danach transportiert er die delegierte User-Identität zum EIS, welches dann ein Mapping auf eigene entsprechende Identitäten durchführt. Je nach Mechanismus der Delegation, der verwendet wird, basiert dieses Mapping rein auf Vertrauen in den Ressource-Adapter oder auf nachprüfbaren Credentials (z. B. einem SSO-Token, das den Request begleitet).

Die oberste in Abb. 6.21 skizzierte Lösung ist der Idealfall: Verteilte und EIS-Domäne beruhen auf der gleichen Registry (in diesem Fall meist eine auf dem Mainframe basierende, z. B. RACF). In diesem Fall erübrigen sich die Mappings,

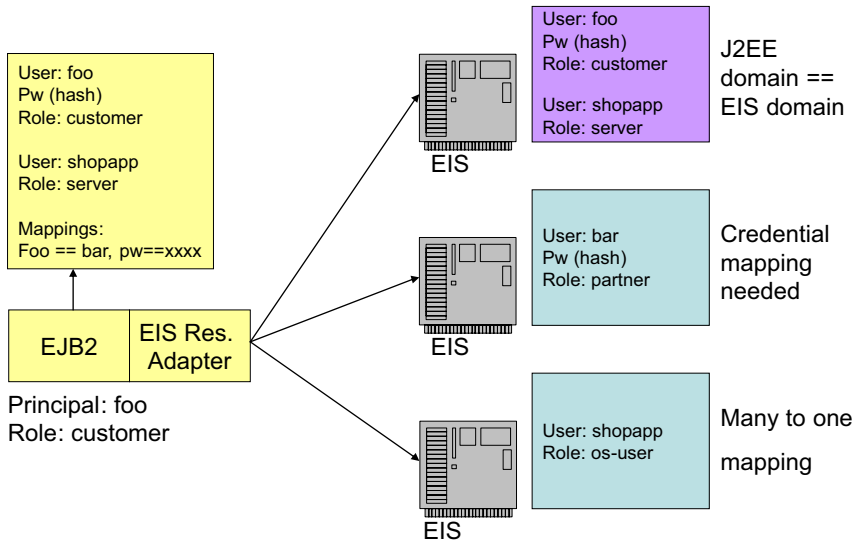


Abb. 6.21 Möglichkeiten des Identitäts-Mappings

denn beim Beginn eines Requests, also wenn der Client bzw. Aufrufer sich beim SSO-Service authentisiert, werden die Credentials des Clients mithilfe der EIS-Registry validiert. Dort wird ein SSO-Token erzeugt und an den Reverse Proxy geliefert. Dieses Token begleitet dann den Request des Clients über alle Intermediates bis hin zum Target. Auf dem EIS/Mainframe-System kann dann dieses Token zur Autorisierung verwendet werden, oder bei Bedarf sogar dazu, den User dort zu impersonifizieren, d. h. zum Betriebssystem-User zu machen. Diese Möglichkeit beruht darauf, dass das Token (also die Authentisierung) ja ursprünglich auf diesem System erstellt wurde. Voraussetzung ist natürlich, dass die User als Betriebssystemnutzer in der Domain eingetragen sind.

Neben den Clients erhalten auch alle Intermediates Token vom SSO-System, mit denen sie sich ausweisen können. So entsteht eine komplette Kette zur End-to-End-Validierung eines Requests (s. Abb. 6.22).

Die beim Request weitergereichten Informationen können auch Daten über die Natur des Requests sowie dazu gehörige Rechte beinhalten. Die Vorteile eines solchen Systems liegen auf der Hand:

- Es müssen keine Authentisierungsgeheimnisse weitergegeben werden.
- Zwischen zwei kommunizierenden Knoten findet jeweils eine gegenseitige Authentisierung statt.
- Die Tatsache des „Arbeitens im Auftrag“ wird ganz klar in Form von Meta-Daten eines Requests kommuniziert.
- Eine Trusted-Third-Party (hier die EIS-Registry) ermöglicht durch signierte Tokens die Authentisierung.

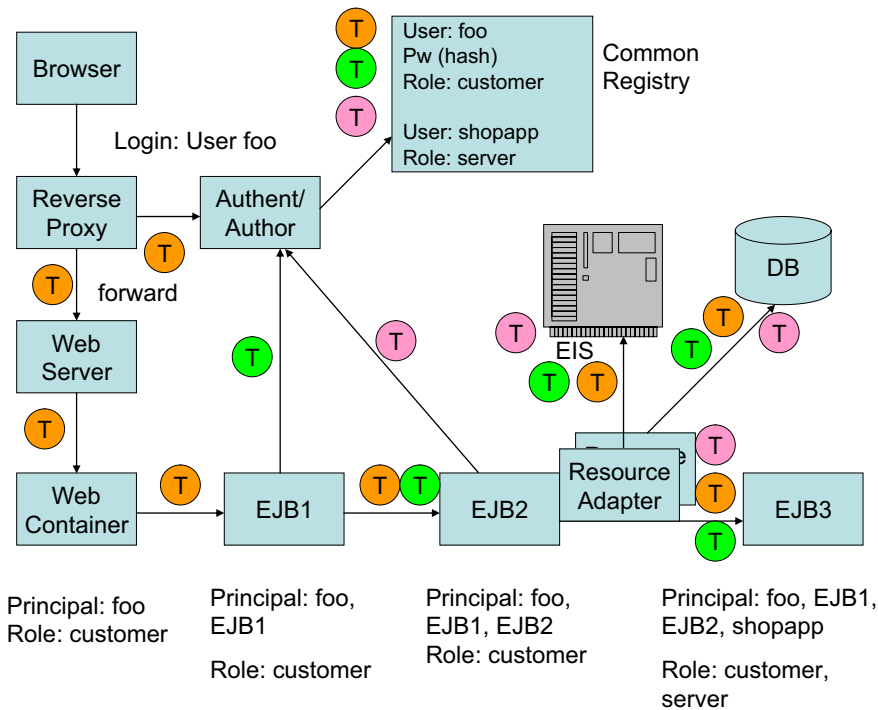


Abb. 6.22 Fluss von SSO-Tokens zur End-to-End-Validierung eines Requests

Momentan sind weder die JAVA-EE-Implementierungen noch die beteiligten Legacy Systeme in der Lage, eine solche vollständige Delegation durchzuführen, jedoch geht die Entwicklung der Application-Server immer mehr in Richtung tokenbasierter Requests (siehe auch Kap. 7 zur Application Server Security).

6.3.5 Rollenbasierte, deklarative Access Control anhand eines Beispiels

In diesem Abschnitt wollen wir einige Besonderheiten der JAVA-EE-Sicherheit anhand eines Beispiels diskutieren. Insbesondere wollen wir auf die rollenbasierte Zugriffskontrolle in JAVA EE eingehen.

Zuvor ist aber vielleicht eine kurze Diskussion darüber angebracht, was Application Level Security überhaupt beinhaltet. Sicherheitsrelevante Entscheidungen haben nicht nur mit der Übertragung von Daten oder der Prüfung gültiger statischer Rollen zu tun: Fragen wie die, ob ein Kunde einen Sonderpreis erhält, sein Auftrag einer besonderen Prüfung bedarf etc. sind Teile der Businesslogik der Applikation, die sich heutzutage täglich ändern kann. Es ist also eminent wichtig, solche Entscheidungen flexibel treffen zu können, ohne dass die Applikation dau-

ernd angepasst werden muss. Dies spricht grundsätzlich für ein Sicherheitsdesign, bei dem der Ort der Kontrolle von den Verfahren, die die Entscheidung herbeiführen, getrennt ist, also wieder für die Trennung von Policy Enforcement Point (PEP) und Policy Decision Point (PDP), wie sie im XACML-Standard definiert ist. Dieses Vorhaben ist jedoch gerade im Bereich instanzbasierter Sicherheit – also bei allen Entscheidungen, die nicht mit statischen Rollen zu tun haben, sehr problematisch. Hier zeigt sich eine klare Einschränkung von Java EE: Die deklarative Sicherheit dort ist ausschließlich auf Klassen (d.h. Rollen) bezogen, nicht instanzbasiert.

Wir verdeutlichen nun diese Problematik anhand eines Beispiels aus dem Java EE Tutorial [J2EE]. Es demonstriert die deklarative Rollendefinition durch Deployment-Deskriptoren. Hervorgehoben sind einige interessante Aussagen bezüglich Security, die sich im Kommentar der Konfiguration befinden. In dem Beispiel geht es um das Lesen und Schreiben von Mitarbeiterinformationen. Offensichtlich sind Mitarbeiter in der Rolle „employee“ berechtigt, ihre eigenen Informationen zu lesen. Bereits an dieser Stelle wird deutlich, dass die deklarative Sicherheit von Java EE an dieser Stelle nicht wirklich weiterhelfen kann: Die Frage, ob die Informationen jemandem gehören, ist eindeutig auch von der Instanz der Information abhängig und nicht nur von der Rolle.

```
<assembly-descriptor>
<security-role>
<description>
    This role includes the employees of the enterprise who are allowed to
    access the employee self-service application. This role is allowed only to
    access his/her own information.
</description>
    <role-name>employee</role-name>
</security-role>
<security-role>
<description>
    This role includes the employees of the human resources department.
    The role is allowed to view and update all employee records.
</description>
    <role-name>hr-department</role-name>
</security-role>
```

Die zweite Rollendefinition „hr-department“ bezieht sich auf die Zugriffsrechte der Personalabteilung. Diese Rolle darf laut Kommentar alle Mitarbeiterinformationen lesen und abändern.

Die Definition der Rolle „payroll-department“ macht jedoch deutlich, dass mit „allen“ Mitarbeiterinformationen wohl kaum wirklich alle gemeint sein können, denn es wird eine spezielle Rolle für den Zugriff der Gehaltsabteilung definiert, die die Gehaltsdaten aller Mitarbeiter lesen und schreiben darf. Somit stellt sich die Frage, ob dies ein ausschließliches Recht ist, oder ob z.B. in der Rolle „hr-department“ zumindest das Lesen des Gehalts erlaubt ist. Auch die Frage von

Stellvertreterregelungen bzw. Ausnahmefällen (Krankheit etc.) ist noch offen. Soll ein Stellvertreter eines Inhabers der Rolle „payroll-department“ einfach die nötigen Rechte erhalten, oder soll sich die Tatsache der Stellvertretung an irgendeiner Stelle niederschlagen? Gravierender ist die Frage, ob sich die Rechte des Stellvertreters an irgendwelchen Stellen nicht doch von den Originalrechten unterscheiden sollen, und wie sich diese Unterscheidung im Code der Applikation auswirkt.

```

<security-role>
<description>
    This role includes the employees of the payroll department. The role is
    allowed to view and update the payroll entry for any employee.
</description>
<role-name>payroll-department</role-name>
</security-role>
<security-role>
<description>
    This role should be assigned to the personnel authorized to perform
    administrative functions for the employee self-service application.
    This role does not have direct access to sensitive employee and payroll
    information.
</description>
<role-name>admin</role-name>
</security-role>
...
</assembly-descriptor>

```

Sehr interessant, weil im Extremfall völlig orthogonal zu den bisher betrachteten Businessrollen ist die nächste Definition: Es wird eine „admin“-Rolle definiert, die innerhalb der betrachteten Applikation (employee self-service application) nicht näher beschriebene administrative Funktionen erledigen soll, ohne jedoch direkten Zugriff auf sensitive Mitarbeiter- oder Gehaltsdaten zu haben. Sensitive Mitarbeiterdaten können z. B. private Telefonnummern, Firmenkredite, Bonuszahlungen oder Abmahnungen sein. Gehören Backups zu diesen administrativen Aufgaben? Mit der Entscheidung, administrative Funktionen in die Applikation explizit aufzunehmen, ist eine wichtige Weichenstellung erfolgt. Das bedeutet, dass versucht wird, auch administrative Maßnahmen durch das Rechtssystem der Applikation selbst zu erledigen (statt durch externe Maßnahmen wie z. B. SQL-Direktzugriff auf die Datenbank) und sozusagen den Sicherheitskontext der Applikation auch für Administration nicht zu verlassen.

Dieses Vorgehen bei der Administration hat entscheidende Vorteile:

- Die Audit-, Log- und Trace-Fähigkeiten der Applikation können auch für die Administration genutzt werden.
- Meta-Rollen, wie das Recht, Applikationsrollen zu erzeugen, können so eingebaut werden. Hier ist das Ziel die sogenannte *Management Service Delegation*, d. h. Zweigstellen kann die eigene Verwaltung auch in Bezug auf Rollen und

Personen übertragen werden. Auch mandantenfähige Systeme (im Allgemeinen fallen alle Application Service Provider darunter) benötigen diese Fähigkeit.

- Administrative und geschäftliche Nutzung können über dasselbe GUI erfolgen (dies bedeutet aber nicht, dass beide Funktionen gleichermaßen z. B. im Internet verfügbar sein sollten).

Aber es tauchen auch sehr große Probleme bei diesem Vorgehen auf:

- Wie verhindert man beim Administrationszugriff, der oft in einem privilegierten Modus erfolgt, dass Administratoren Businessdaten einsehen können?
- Für besonders kritische Operationen könnte eine bessere Qualität der Authentisierung verlangt werden. Das Verbot des Zugangs vom Internet aus ist nur ein Beispiel dafür. Es könnte auch eine bessere Passwortqualität oder ein zusätzliches Geheimnis, bis hin zur Authentisierung mit Chipkarte und Zertifikat, verlangt werden. Dies ist für die Applikation im momentanen Java EE nicht beeinflussbar.

Im Folgenden wollen wir einige weitere Probleme und deren Lösungsmöglichkeiten vorstellen und dabei einige Grundprinzipien der Sicherheit in Enterprise-Umgebungen herausarbeiten. Am besten beginnen wir mit einem Blick auf das Informationsmodell der Applikation, z. B. auf die Mitarbeiterinformationen (Abb. 6.23).

Die Mitarbeiterinformationen lassen sich grob in vier Bereiche aufteilen und auch auf entsprechende Rollen verteilen. Es zeigt sich jedoch schnell, dass es große Überlappungen bei den Rollen in Bezug auf die Daten gibt. An dieser Stelle gehen wir für den Moment von einer Repräsentation der Daten in Form einer

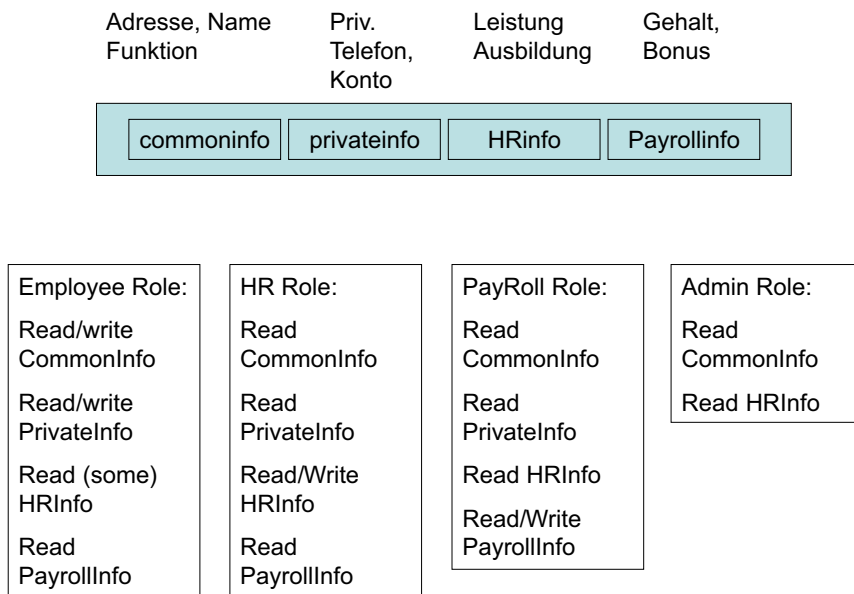


Abb. 6.23 Informationsmodell der Beispielapplikation

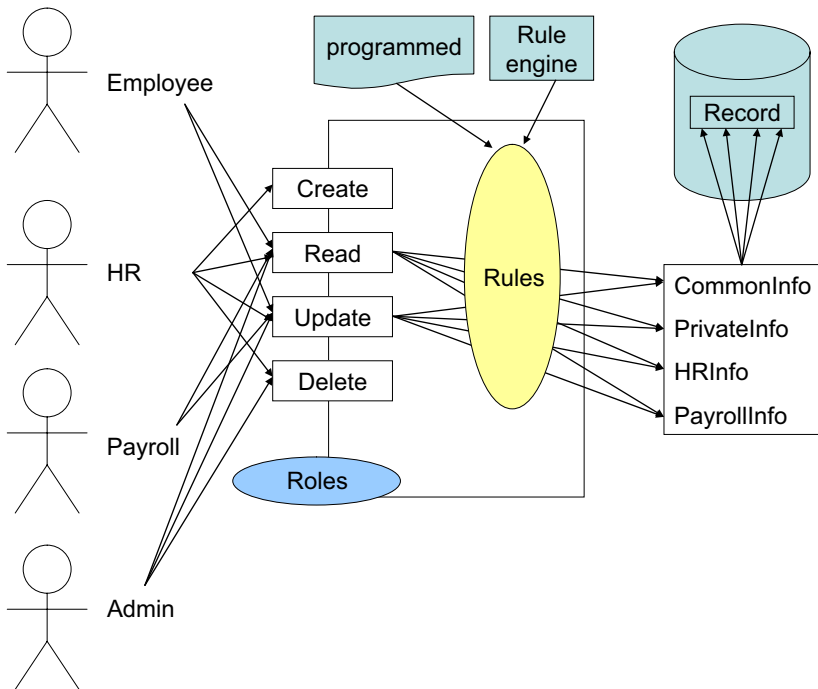


Abb. 6.24 Deklaratives Modell der Zugriffskontrolle in der Beispielapplikation

Tabelle aus. Ein mögliches deklaratives Modell der Zugriffskontrolle durch Bean-Methoden könnte dann wie in Abb. 6.24 aussehen.

Ein Entity Bean kapselt die Mitarbeitertabelle und bietet die typischen generischen CRUD (Create, Read, Update, Write) Aktionen an. Eine direkte Abbildung der Aktionen auf eine Sicherheitssicht, d.h. ein Firmenmodell von Rollen und Verantwortungen, fällt auf dieser Ebene schwer. Wenn man überhaupt von einer Gruppierung von Aktionen durch generische Rollen sprechen kann, dann bilden sich in kommerziellen Applikationen häufig folgende Gruppen:

- Read Access Only (Kunden, Personal),
- Read/Write Access (Sachbearbeiter) und
- Create/Delete (Administrator, Functional User).

In unserem Beispiel zeigt sich schnell, dass die Rollenzuteilung auf die Methoden nur sehr wenig Sicherheit bringt: Viele Methoden müssen für alle Rollen zugänglich sein und die unterschiedlichen Regeln bezüglich der Datenzugriffe müssen im Inneren der Methode erfolgen. Zumindest könnte man jedoch durch deklarative Sicherheit die Mitarbeiter vom Erzeugen bzw. Löschen von Datensätzen ausschließen.

Das generische Design der Methoden führt bei überlappenden Datenzugriffen zwangsläufig zu einer hohen Komplexität in der Businesslogik innerhalb der Komponente. Sie kann programmatisch durch `context.isUserInRole`

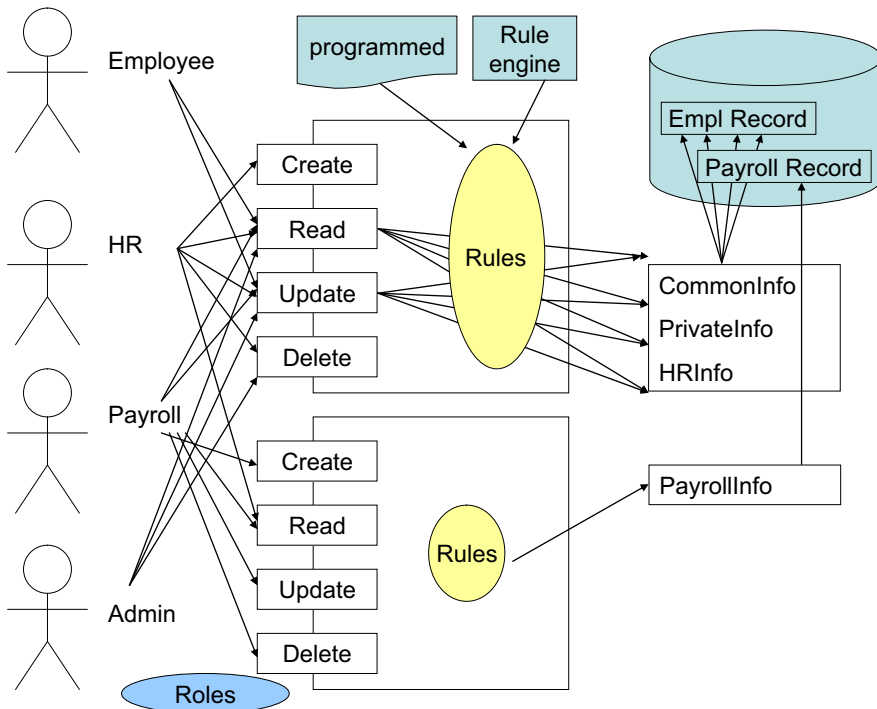


Abb. 6.25 Abtrennung der Gehaltsinformation im Informationsmodell

(„Employee“) etc. erstellt werden, oder durch Delegation des Requests an eine Rule Engine, deren Ergebnis z. B. in der Auswahl verschiedener Objekte (je nach Rolle) besteht.

Wie der Zugriff auf die Datenbank gesichert ist, spielt hier zunächst keine Rolle, da keinerlei Wissen über die Rollen und Principals in der Datenbank gespeichert ist. Wie werden aber auf diesen kritischen Punkt im Laufe der Diskussion noch öfters stoßen und im Kapitel zur Sicherheit von Applikation-Server-Mechanismen zur Propagation von Identität an Datenbanken kennen lernen. Die Java-EE-Spezifikation gibt uns jedoch keinerlei Garantien darüber und überlässt diese Mechanismen den Herstellern der Container und Server. Als Konsequenz kann die Applikation, wenn sie solche Mechanismen voraussetzt, nicht auf allen Containern korrekt laufen.

Eine weitere Designmöglichkeit bestünde darin, die besonders kritischen Gehaltsinformationen von den sonstigen Mitarbeiterinformationen bereits im Informationsmodell zu trennen und diese Trennung auch in der Datenbank zu verfolgen (Abb. 6.25).

Damit wurden die überlappenden Rollenzugriffe deutlich verringert und die Komplexität der Sicherheitslogik in den Komponenten wird ebenfalls geringer. Wenn die Rollen bzw. Identitäten auch bis zur Datenbank propagiert werden, kann diese sicherstellen, dass weder die Employee- noch die Admin-Rolle auf Gehalts-

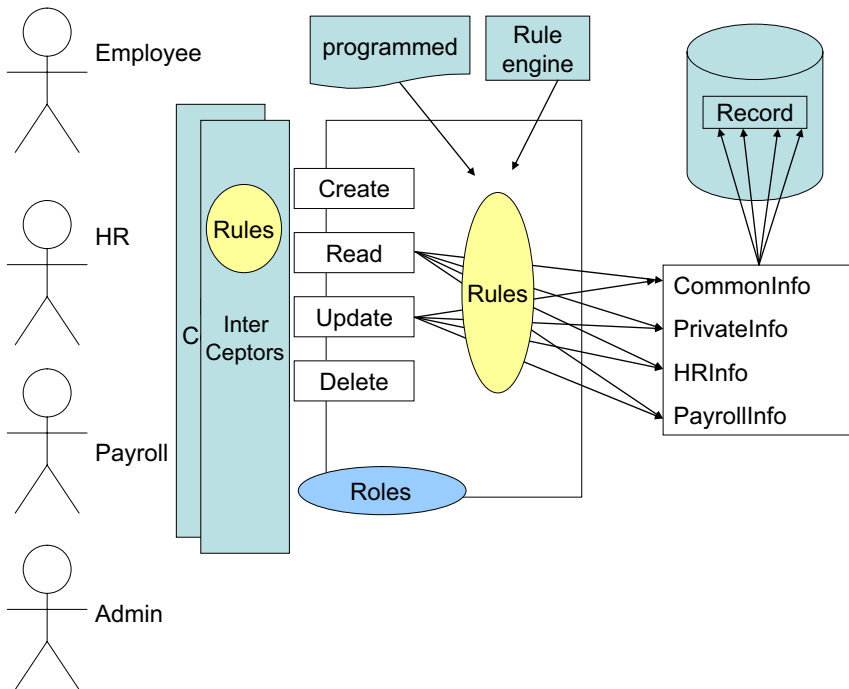


Abb. 6.26 Zugriffskontrolle via Interceptors

informationen zugreifen könnten. Der Preis dafür besteht darin, dass wir für die Darstellung der gesamten Mitarbeiterinformationen nun Zugriff auf mehrere Beans bzw. Tabellen brauchen. Das Design könnte sich aber für Fälle mit besonders kritischen Informationen, wie z. B. Passwörtern innerhalb der Datensätze, anbieten.

Wer nun die Daten nicht trennen und keine Rule Engine einsetzen möchte und auch die programmatische Sicherheit nicht mag, dem bleibt als Möglichkeit, die Zugriffsregeln auch in sogenannten *Interceptors* unterzubringen, welche die Requests bereits vor der Rule Engine filtern (Abb. 6.26).

Dieser Ansatz sorgt für größere Flexibilität und lässt den Zugriffscode sauber durch eine Reihe von Filtern programmieren. Datenbanktechnisch ist nichts Besonderes gefordert, da Rollen und Identitäten auf Seiten der Datenbank nicht für Kontrollen verwendet werden.

Die nächste Lösungsmöglichkeit stellt nur eine relativ kleine Abänderung unserer ersten, generischen Lösung dar (Abb. 6.27).

Statt weniger, generischer Methoden sind die Rollenzugriffe nun auf mehrere Methoden verteilt worden. Dadurch lassen sich deklarativ die Rollen besser zuordnen, es finden weniger Mehrfachbesetzungen statt und die Businesslogik entflechtet sich spürbar. Hier zeigt sich eine Grunderfahrung mit deklarativer Sicherheit: Wenn man feststellt, dass für eine Methode oder eine Klasse mehrere Rollen definiert werden müssen, sollte man dann evtl. die Methoden entflechten. Der Preis für die Entflechtung liegt unter Umständen in einer höheren Quote duplizierten

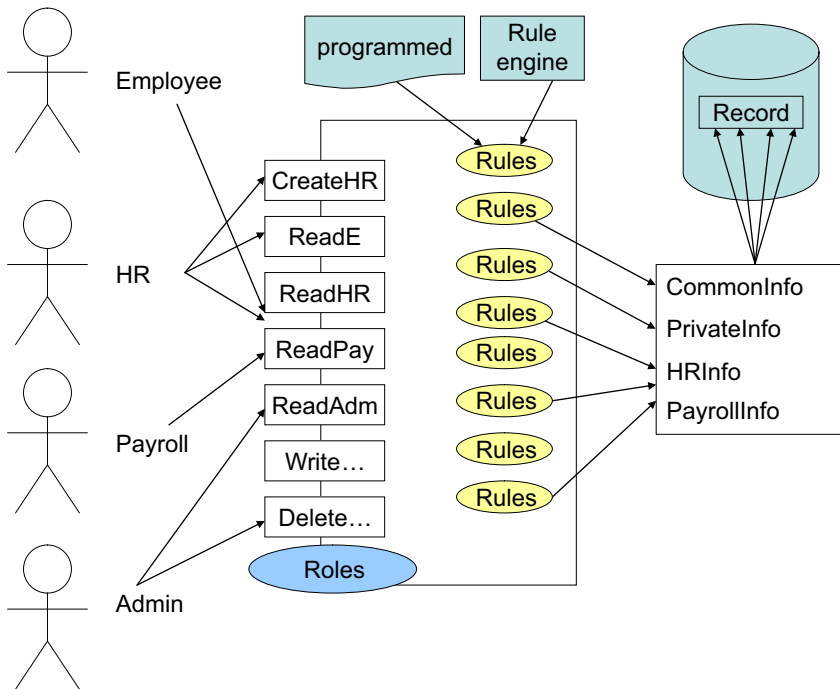


Abb. 6.27 Definition verschiedener Methoden für die verschiedenen Rollenzugriffe

Codes. Hier kann man durch Vererbung oder Delegation von Basisfunktionalität jedoch wieder Ordnung herstellen.

Das folgende Design (s. Abb. 6.28) kann für die gesamte Applikation sehr kritische Folgen haben. Es wurde z. B. im bereits erwähnten (gescheiterten) Projekt INPOL eingesetzt und war dort angeblich für minutenlange Queries verantwortlich.

Normalerweise sieht man bei Enterprise-Applikationen – genauso wie beim Design einer Firewall-Komponente – das „Trichterprinzip“ im Einsatz: Beginnend mit einem recht breiten Zugang (z. B. über das Internet) wird schnell eine massive Verengung des Zugangs durch Authentisierung und Autorisierung erreicht. Die Prüfung der Zugangskontrolle findet also bereits sehr nahe des Eingangs statt, und je granularer dies erfolgt, desto besser.

Das hier gezeigte Design widerspricht diesem Prinzip jedoch völlig: Es lässt durch öffentliche, generische Methoden jeden Request bis auf die Datenbank gelangen. Dort ruht nun die gesamte Sicherheitslogik – wahrscheinlich in Form von Stored Procedures – und muss nun herausfinden, welcher Request welchen Zugriff auf welche Teile des Records besitzt. Gerüchten zufolge hat sich dieses Design im oben genannten INPOL-Projekt als fatal erwiesen, sowohl in Bezug auf die Performance als auch in Bezug auf die Flexibilität der Sicherheitslogik. Zu beachten ist ferner, dass eine Änderung der Sicherheitslogik in der Datenbank große Konsequenzen auf vorgelagerten Code besitzen kann. Um Missverständnissen vorzubeugen: Dies soll

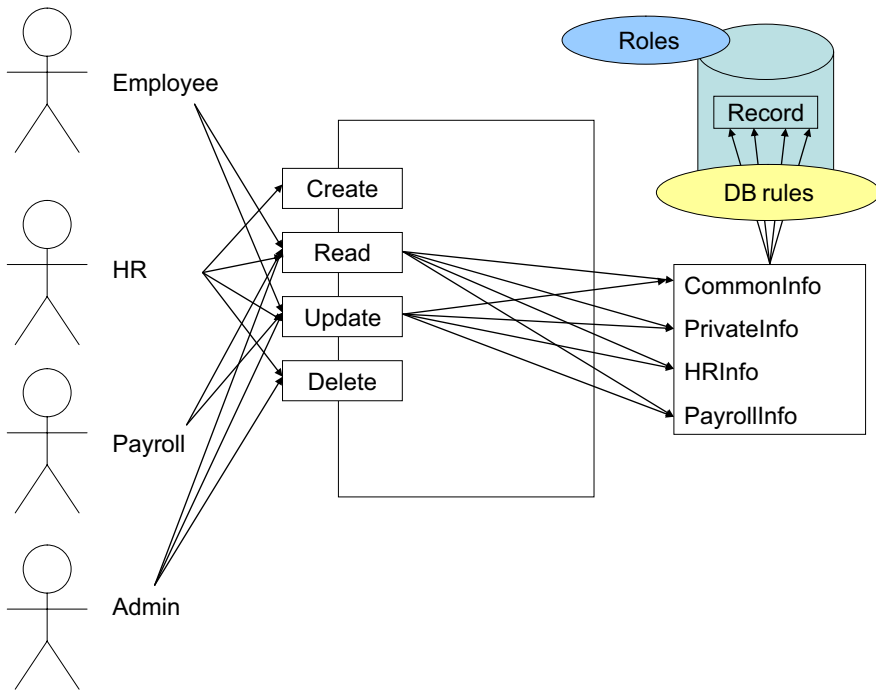


Abb. 6.28 Zugriffskontrolle in der Datenbank

kein Plädoyer gegen eine Sicherheitslogik und Prüfungen von Requests auf dem Backend sein. Im Gegenteil: Solche Prüfungen sind dort als zusätzlicher Schutz sehr wichtig, etwa gegen verwechselte Pfade oder schlichte Programmierfehler. Backend Security ist ein wichtiger Teil des Prinzips „Defense in Depth“, und ein zu breiter Zugang auf Daten im Backend ist immer problematisch.

Unser nächstes Design (Abb. 6.29) bricht die Informationen bereits auf der Ebene der Datenbank in einzelne „Views“ auf. Diesen Views können Rollen und Identitäten zugeordnet werden und damit – Propagation der Identitäten vorausgesetzt – eine hohe Sicherheit erreicht werden.

Ein ähnliches Prinzip verwendet unser letztes Design (Abb. 6.30): Es greift zwar generisch auf das Backend zu, hält die Businesslogik jedoch in verschiedenen „Fassaden“ vor dem Backend getrennt voneinander.

Die Sicherheitslogik wird durch dieses Prinzip einfacher und überschaubarer. Der generische Zugriff auf das Backend muss allerdings alle Rollen erlauben. Durch die Fassaden mit Access Control entstehen dann unterschiedliche Sicherheitszonen: Am Eingang wird die Identität geprüft, weiter hinten bleibt es der Applikation überlassen, für die Sicherheit der Requests zu sorgen. Das ist aus Performancegründen ein sehr beliebter Ansatz, verstößt aber im Grunde gegen das Prinzip der „Complete Mediation“, welches besagt, dass möglichst jeder Zugriff autorisiert werden sollte.

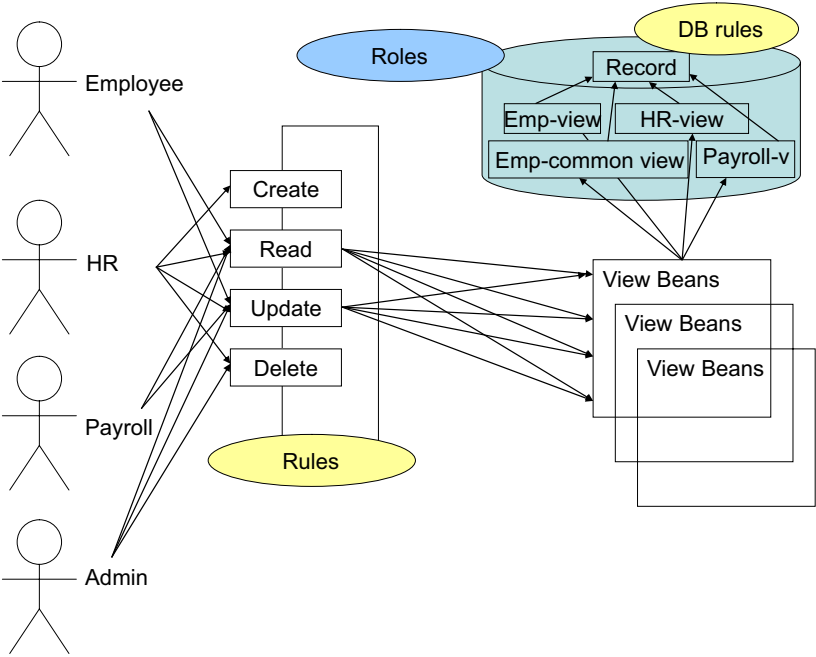


Abb. 6.29 Zugriffskontrolle über unterschiedliche Views auf die Datensätze

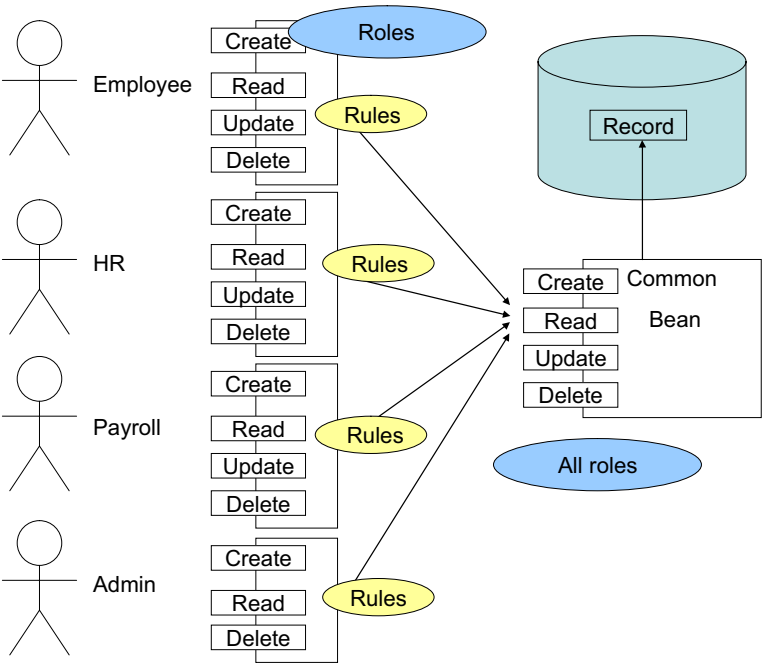


Abb. 6.30 Zugriffskontrolle über Fassaden vor dem Backend

Das Prinzip der Sicherheitszonen lässt sich aber durchaus mit Backend-Prüfungen koppeln (Propagation der Identitäten vorausgesetzt). Besonders effektiv kann dies werden, wenn die Backend-Prüfung auf Datenbankseite mit den neueren Konzepten des Label Based Access Control (LBAC) geschieht. Die Integration solcher Verfahren, die letztlich auf dem Prinzip des Data Tagging in rollenbasierte Konzepte von JAVA EE beruhen, muss allerdings noch genauer untersucht werden. Die Granularität der Zugriffskontrolle wird dadurch jedoch enorm gesteigert.

6.3.6 Deklarative und programmatische Zugriffskontrolle

Wir behandeln nun einige Beispiele für die deklarative und programmatische Zugriffskontrolle von Java EE und diskutieren einige Spezialfälle. Wir beginnen mit der Zuordnung von mehreren Rollen zu einer Methode: Was bedeutet es, wenn eine Methode mehreren Rollen zugeordnet wird? Bedeutet es, dass das Bean intern die verschiedenen Rollen prüft? Nicht notwendigerweise, aber was macht es für einen Sinn, wenn es nicht so ist?

Java-EE-Rollen besitzen keine hierarchische Struktur – es sind im Grunde genommen reine Rechte-Token für eine Gruppe von Methoden oder Klassen. Ob ein Recht ein anderes Recht mit einschließt, muss an anderer Stelle entschieden werden, nämlich da, wo die Credentials eines Principals aus einem LDAP-Verzeichnis oder anderen Quellen gefüllt werden. Also bleibt eigentlich nur die Erklärung, dass bei mehreren Rollen mit programmatischer Security gearbeitet wird (also mit `getPrincipal()` und `isUserInRole()`).

Sehen wir uns auch dazu wieder ein leicht erweitertes Beispiel aus dem Java EE Tutorial an:

```
@RolesAllowed("payroll") // needed??
@RolesAllowed("employee")
@DeclareRoles("payroll")

@Stateless public class PayrollBean implements Payroll {
    @Resource SessionContext ctx;
    public void updateEmployeeInfo(EmplInfo info) {
        oldInfo = ... read from database;

        // The salary field can be changed only by callers with role "payroll"
        if (info.salary != oldInfo.salary && !ctx.isCallerInRole("payroll")) {
            throw new SecurityException(...);
        }

        ...
    }
    ...
}
```

Ist das ein guter Ansatz? Könnte man die Unterschiede zwischen den Rollen „payroll“ und „employee“ nicht auch in zwei verschiedene Methoden auslagern

und damit wieder auf deklarative Weise realisieren? Muss an dieser Stelle unbedingt programmatisch gearbeitet werden?

An dieser Stelle können wir auf diverse Lösungsmöglichkeiten, die wir in Abschn. 6.3.5 behandelt haben, verweisen: Die Methoden könnten getrennt werden. Es könnten die Mitarbeiterinformationen in unterschiedliche Klassen/Tabellen aufgeteilt werden. Es könnte über vorgelagerte Fassaden bereits ein Vorkontrolle vorgenommen worden sein, etc. Die konkrete Lösung ist dann Resultat eines Bündels aus Überlegungen zu Wartbarkeit, Performance, Sicherheit etc.

Nun eine kleine Verschärfung in Bezug auf die Sicherheitsanforderungen: Wir wollen natürlich nicht, dass Mitarbeiter der Lohnstelle ihre eigenen Gehaltsdaten verändern können. In diesem Fall stimmt zwar die Rolle, aber die Instanzdaten sollen einen Zugriff nicht zulassen:

```
@RolesAllowed("payroll") // needed??
@RolesAllowed("employee")
@DeclareRoles("payroll")

@Stateless public class PayrollBean implements Payroll {
    @Resource SessionContext ctx;
    public void updateEmployeeInfo(EmpInfo info) {
        oldInfo = ... read from database;

        // The salary field can be changed only by callers with role "payroll"
        if (info.salary != oldInfo.salary && !ctx.isCallerInRole("payroll")) {
            throw new SecurityException(...);
        }

        if (info.salary != oldInfo.salary && ctx.isCallerInRole("payroll")) {
            callerName = ctx.getCallerPrincipal().getName();

            if (callerName.equals(oldInfo.name()) //
                throw new SecurityException(..); payroll employee == employee!!!
            }
        }
    }
    ...
}
...
}
```

Es ist ein Attribut der Instanz, das hier den Unterschied ausmacht. Leider sind längst nicht alle instanzabhängigen Prüfungen so stabil wie die oben genannte: Der Mitarbeiter der Gehaltsstelle darf zwar sein eigenes Gehalt nicht bestimmen. Aber was passiert, wenn ein Notfall eintritt und keine zweite Person vor Ort ist? In einem solchen Fall muss auch eine solche Regel auf sichere Weise für einige Zeit aufgehoben werden können. Wenn sie hart codiert ist, ist das aber nicht mehr möglich.

Nochmals ein kleines Beispiel für instanzbasierte Access Control: Oft bestimmt die Höhe eines Betrags das Erfordernis einer bestimmten Rolle (z. B. Vorgesetzter muss Bestellung unterschreiben) oder einen speziellen Vorgang (z.B. Vier-

Augen-Prinzip). Aber auch die Kundensegmentierung wechselt mittlerweile fast täglich, also die Einordnung der Kunden in bestimmte Kategorien, die wiederum mit Aktionen, Rabatten, Privilegien und Angeboten verbunden sind. Waren diese früher relativ statisch, so braucht man sich heute nur die Aktionen im Bereich der Mobilfunkanbieter, DSL-Anbieter oder im Modemarkt anschauen, um zu wissen, dass die Angebote und Klassifizierungen fast täglich wechseln. Die harte Codierung solcher Regeln fällt da als Lösung weg. In Bezug auf die Zugriffskontrolle bedeutet das, dass an der Stelle der Kontrolle nichts mehr berechnet werden darf, sondern dort darf nur ein Ergebnis abgefragt werden. Jede Berechnung führt automatisch zur Verlagerung des Businesskalküls in die Zugriffskontrolle und verhindert die flexible Änderung der Businessregeln bzw. erzwingt dauernde Anpassungen der Zugriffskontrolle.

An dieser Stelle drängt sich eine Analogie auf: In der Präsentationsschicht einer Applikation wollen wir im Allgemeinen ebenfalls jede Form von Businesslogik vermeiden. Anderenfalls muss die Darstellung permanent an die Änderungen der Businesslogik angepasst werden. Eventuell lassen sich erfolgreiche Mechanismen bei der Trennung von Präsentation und Businesslogik auf die Zugriffskontrolle übertragen.

Sehen wir uns dazu das Template auf der linken Seite von Abb. 6.31 an: Es soll die Kundendaten aus der rechten Komponente bearbeiten.

Abbildung 6.31 zeigt jedoch einen falschen Ansatz: Der Code des Templates holt sich Attribute des Modells und stellt darüber Berechnungen an, die jedoch nicht dem Layout dienen, sondern die Businesslogik darstellen: Wann ist ein Kunde ein Goldkunde? Wie viel Rabatt bekommt ein Goldkunde? Diese Berechnungen können am nächsten Tag bereits überholt sein und zwingen zur Anpassung des GUIs.

Abbildung 6.32 zeigt hingegen die richtige Lösung: Das Template bezieht lediglich Attribute aus dem Modell und unterlässt eigene Berechnungen der Businesslogik. Im Notfall sind dazu die Schnittstellen des Modells zu erweitern.

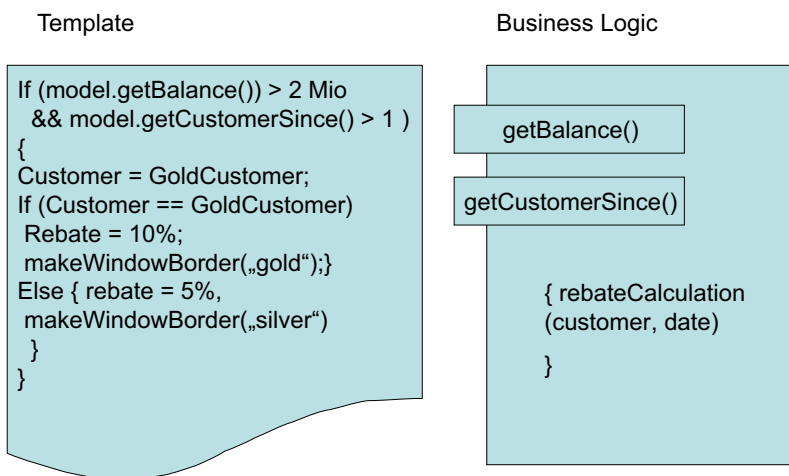


Abb. 6.31 Hart codierte Berechnungen aus der Businesslogik im Template

Wohlgermerkt, das Template verheimlicht nicht, dass es um unterschiedliche Renderings für verschiedene Kundenklassifizierungen weiß, aber es weiß nicht, wie diese berechnet werden.

Wir verallgemeinern unsere Überlegungen bezüglich der Businesslogik zu einer generellen Applikationsarchitektur, bei dem die Applikation bei der Businesslogik nur Attribute der Instanzen nachfragt, diese aber nicht selbst berechnet (s. Abb. 6.33). So ist die Applikation von häufigen Wechslen in der Businesslogik unabhängig.

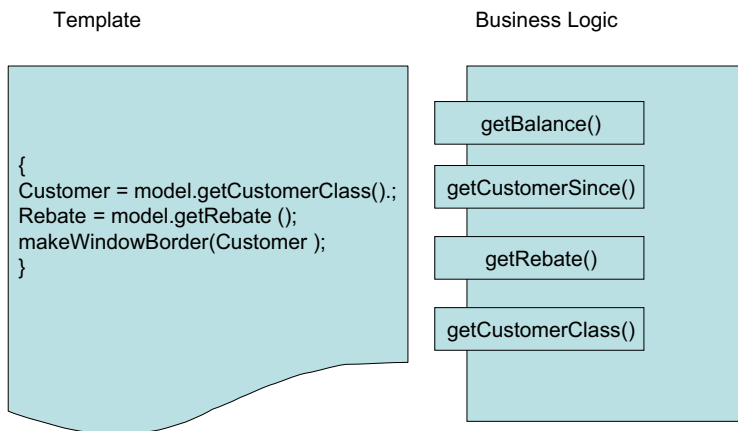


Abb. 6.32 Berechnungen sind in der Businesslogik gekapselt

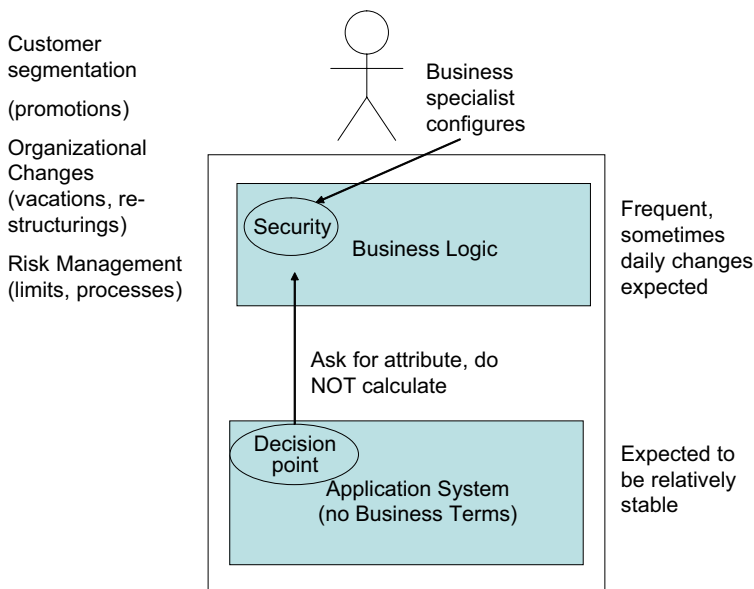


Abb. 6.33 Generische Applikationsarchitektur mit ausgelagerter Businesslogik

Es lohnt sich auch bei dedizierten Applikationen (nicht nur bei ganzen Frameworks oder Production Lines), eine Aufteilung in System und Businesslogik durchzuführen, orthogonal zu den üblichen Ebenen (Client-Ebene, Middleware, Backend). Im Systemteil der jeweiligen Ebene finden sich stabile Routinen, die – wenn sie Entscheidungen treffen müssen – diese per Attributabfrage an die höhere Businesslogik Schicht erledigen. Sie sind nur ausführende Organe, die Politik wird oben bestimmt.

Sehen wir uns zum Abschluss noch einen anderen Ansatz zur instanzbasierten Zugriffskontrolle an: Ein sehr interessanter Ansatz im Java-Bereich benutzt dazu erweiterte Permission-Klassen. Grundproblem beim Durchführen von Prüfungen, die Instanzdaten beinhalten, ist der Zugriff auf diese Daten, falls sich der Prüfcode außerhalb des Objektes befindet (z. B. in einem Interceptor), oder eben in einem Permission-Objekt. Deshalb muss zunächst ein generisches Interface definiert werden, das die zu prüfenden Objekte implementieren müssen. Zu diesem Interface könnte z. B. die Methode `isOwner(Name)` gehören. Es wird außerdem ein neuer Typ von Permissions definiert, die dieses Interface verstehen und auswerten können. Jetzt kann in einem Interceptor eine Permission instanziiert und mit `SystemManager.checkPermission()` geprüft werden. Entscheidend ist natürlich das Interface der zu prüfenden Komponente. Für verschiedene Komponenten können allerdings auch verschiedene Permission-Typen angelegt werden.

6.3.7 *Annotations für Sicherheit in Java EE*

Der Vollständigkeit halber geben wir hier die Liste der sicherheitsrelevanten Annotationen in Java EE an:

`@DeclareRoles`
Anzeigen, dass eine Rolle programmatisch verwendet wird
`@RolesAllowed`
Gestatte der Rolle den Zugriff auf eine Methode oder Klasse
`@PermitAll`
Erlaube allen Rollen Zugriff
`@DenyAll`
Unterbinde den Zugriff auf eine Methode für alle Rollen
`@RunAs`
Wechsele die Delegationsidentität (nur für downstream-calls)

Die neu hinzugekommenen Annotationen duplizieren lediglich eine Funktionalität, die bereits mit den Deployment-Deskriptoren vorhanden war. Sie erleichtern die Konfiguration allerdings dadurch, dass separate Deskriptoren vermieden werden können. In größeren Firmen wird es jedoch häufig der Fall sein, dass Annotationen durch Deskriptoren überschrieben werden müssen.

Aus Sicht der Usability für Entwickler bringen die Annotationen, die übrigens als Meta-Daten ins Laufzeitsystem gelangen können und dort über Reflektion abgefragt werden können, den Vorteil, dass der Code damit auch die Meta-Daten enthält. Die sicherheitsrelevanten Annotationen sind übrigens keine generativen Anweisungen, sondern tatsächlich reine Attributierungen des Verhaltens.

6.3.8 Sicherheitsdiskussion der JAVA-EE-Sicherheit

Für eine kurze Einordnung der Sicherheitsmechanismen der JAVA-EE-Plattform ist es wichtig, die Absichten dieser Architektur nochmals kurz zusammenzufassen. Im Blickpunkt stand dabei von Beginn an der Versuch, Komponenten zu entwickeln, die von ihrer Umgebung unabhängig waren. Dies bedeutete die Abtrennung der Sicherheitsanforderungen von der Applikation, selbst denn die in den Firmen üblicherweise vorhandene Sicherheitsinfrastruktur unterscheidet sich von Firma zu Firma drastisch.

Im gleichen Zug wollte man die Applikationsentwickler nicht mehr mit der Komplexität der Sicherheitsmechanismen belasten und schwenkte auf deklarative Sicherheit um. Und last but not least sollte die Prüfung der Sicherheitsanforderungen automatisiert werden. Somit wurde die Prüfung an die Container bzw. die Laufzeitumgebungen delegiert.

Die Folge dieser Entscheidungen war, dass die Laufzeitumgebung die Verantwortung für die Sicherheit der Applikationen übernommen hat. Von welchen Servern bzw. über welche Proxies Requests kommen, ist für die Sicherheitsbewertung enorm wichtig. Damit steigt erneut die Bedeutung der Infrastruktur und Änderungen darin werden sehr aufwändig.

Die von der Architektur unterstützte Delegation ist rudimentär. Empfänger können letztlich nicht feststellen, ob ein bestimmter Request tatsächlich mit den Intentionen des Clients übereinstimmt. Lediglich das Vorliegen einer authentisierten Identität kann geprüft werden.

Weitere kritisierte Eigenschaften der JAVA-EE-Sicherheitsarchitektur sind:

- starke Abhängigkeit von Mechanismen der Application-Server,
- Separation of Concerns eröffnet Chancen für Missverständnisse,
- mächtiger Infrastruktur Security Layer Voraussetzung,
- Caller Identity unklar,
- QOS der Delegation nicht bestimmbar,
- fehlende APIs zur Abfrage von Security-Werten,
- Impersonation durch Verwendung der Caller Credentials ist schlecht,
- Wechsel der Identität nicht mehr tracebar,
- keine hierarchischen Rollen: aber das ist implementationsbedingt und
- undefinierte Verbindung von JAAS und EJB-Principals.

Wir werden in Kap. 7 einen Blick unter die Haube wagen und untersuchen, wie Mechanismen der Delegation, Propagation etc implementiert sind und welche Schnittstellen Application-Server für Applikationen und Infrastruktur anbieten. Davor wollen wir noch die Benutzung von JAVA-EE-Sicherheitsmechanismen im Rahmen des sehr populären Model2 Frameworks Struts zeigen.

6.4 Struts

Struts, das bekannte GUI-Framework für Web-Applikationen, dient uns im Folgenden lediglich als Beispiel für eine Web-Applikation, die auf einem Software-Framework basiert. Es geht uns hier nicht um Attacken auf Web-Applikationen oder das komplexe Thema Input-Validierung – diese werden an anderer Stelle behandelt – sondern um die Gesamtarchitektur der Sicherheit und deren Einbettung in umgebende Komponenten wie Application-Server und Sicherheitsinfrastruktur. Es ergeben sich erstaunlich viele und unterschiedliche Möglichkeiten der Authentisierung bzw. Autorisierung innerhalb einer Web-Applikation – mit teils dramatischen Unterschieden in der Sicherheit und Wartbarkeit bzw. Kompatibilität mit dem Rest der Firma.

Hier noch ein Tipp für diejenigen, die einige technische Grundlagen der Web-Applikationen auf JAVA-EE-Basis nachholen möchten: Auf der Struts-Page von apache.org ([Apa]) findet sich eine Liste der Basistechnologien sowie weiterführende Links. Man beachte, dass diese nicht gerade kleine Liste noch nichts zur Security enthält! Zusammen mit dem OWASP-Guide zur Entwicklung sicherer Web-Applikationen (s. [McRee]) bietet diese Seite dennoch gute Grundlagen für Technik und Sicherheit von Web-Anwendungen.

6.4.1 *Authentisierung in Web-Applikationen*

Zur Problematik der Authentisierung und Autorisierung bei Web-Applikationen und zum Framework Struts hier zunächst ein kurzer Ausschnitt aus dem Artikel von Dirk Heringhaus zu den Sicherheitslücken in der T-COM-Applikation OBSOC – der zentralen Kundenverwaltungsapplikation der Telekom [Her]:

„Es fing damit an, dass ich einen Webhosting-Vertrag im OBSOC administriert habe. Der Link, der mich in meinen Vertrag führte, produzierte folgende URL in meinem Browser:

<http://www.t-mart-web-service.de/contractview/frameset.asp?ConPK=xyz>

wobei xyz für die Vertragsnummer des Vertrages steht, der gerade administriert wurde. Neugierig wie ich war, änderte ich willkürlich die Vertragsnummer und bestätigte mit Enter die nun von mir geänderte URL. Ergebnis: Ich bekam einen anderen Vertrag eines anderen Kunden zum administrieren dargestellt. Exakt so, als ob ich meinen eigenen Vertrag

verwalte. Ich brauchte also nur mein Schließfach aufschließen, eine neue Nummer darauf schreiben und schon hatte ich Zugriff auf das Schließfach, dessen Nummer ich auf mein Schließfach geschrieben hatte. Wie im Märchen. **Die Telekom brauchte einen Tag, um den Fehler zu beheben.**

Wenn man einen solchen Fehler findet, fängt man doch an, sich Sorgen um seine Daten zu machen. Ein aufmerksames Studieren der Seite, die mich in die Vertragsverwaltung führte, brachte unter dem Link ZUKAUF folgenden Text zu Tage:

```
http://www.t-mart-web-service.de/SalesFrontend/  
initializeSalesfrontend.asp?intPurchaseType=2&intConPK=xyz
```

Da nach dem anklicken des Links sofort der sogenannte Konfigurator gestartet wurde, und damit die Adresse nicht mehr in der Ziel-URL auftauchte, kopierte ich den Link von der Eigenschaftenseite in die Zwischenablage. Von dort aus fügte ich den Link in die Adresszeile meines Browsers ein, änderte die `intConPK` und bestätigte die Eingabe. Was sollte Anderes passieren, als das ich den entsprechenden fremden Vertrag nun im Zukaufsmodul dargestellt bekam.

Ich brauchte also nur in meinem Vertrag die Vertragsnummer ändern, mit dem Vertrag zum Schließfachvermieter gehen, und sagen, dass ich für diesen Vertrag etwas dazukaufen möchte. Nicht nur, dass ich das ohne weitere Überprüfung durfte: Durch ein noch später erklärtes Sicherheitsloch, durfte ich auch noch direkt den Inhalt des Schließfachs mitnehmen. **Auch hier war der Fehler von der Telekom schnell behoben.**“

Die Hervorhebungen stammen von den Autoren und sollen das Muster hinter den Schwächen hervorheben: Es existieren viele einzelne Zugänge in die Applikation, sodass schwere Fehler in der Behandlung des Inputs – und Heringhaus kann später auch noch gravierende Mängel bei der Authentisierung und Autorisierung nachweisen – in allen Ressourcen einzeln lokalisiert und beseitigt werden müssen. Die Ressourcen einer Web-Applikation, die nach außen sichtbar sind, sind ihre URLs. Obwohl diese durch sog. Deep Linking keinen gemeinsamen Eingang – wie eine Fat-Client-Applikation – besitzen, muss der Zugriff dennoch geprüft, authentisiert und autorisiert werden. Gelingt es nicht, dafür eine zentrale Stelle zu schaffen, durch die jeder Zugriff hindurch muss, entsteht das oben geschilderte Fehlermuster. Dieses Muster hat leider auch noch die Eigenschaft, dass es nicht leicht zu beseitigen ist, da es sich letztlich um einen schweren Mangel der Architektur handelt: Es muss nachträglich eine zentrale Instanz (*ChokePoint*) eingebaut werden, durch die alle Requests geleitet werden.

Das Struts-Framework hat versucht, diesen Fehler von Beginn an zu vermeiden enthält deshalb eine solche zentrale Stelle, das sogenannte *ActionServlet*, früher auch Controller genannt. Abbildung 6.34 zeigt die grobe Architektur von Struts und die an der Bearbeitung eines Requests hauptsächlich beteiligten Klassen. In Bezug auf die Anbindung von Backends legt Struts kaum etwas fest: Es ist ganz klar ein Frontend-Framework.

In der Abb. 6.34 sind Möglichkeiten eingezeichnet, in diese Architektur Authentisierung bzw. Autorisierung einzubringen – mit unterschiedlichen Folgen. Der Rest des Kapitels beschreibt diese Möglichkeiten und ihre Konsequenzen für die Sicherheit und Wartbarkeit einer Applikation auf der Basis von Struts.

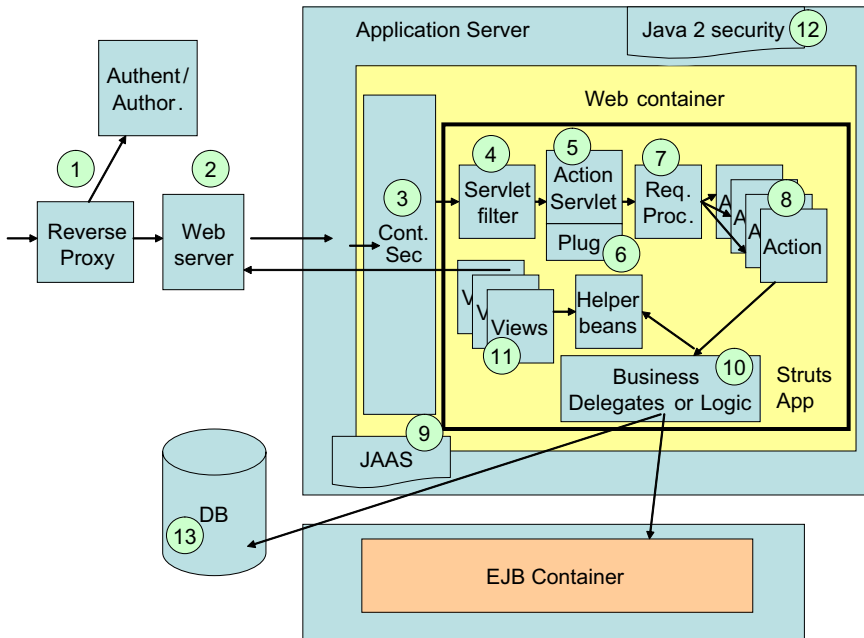


Abb. 6.34 Aufbau einer Struts-Anwendung

Die Nummern in Abb. 6.34 kennzeichnen Punkte, an denen Authentisierung und/oder Autorisierung eingebaut werden kann. Sie werden weiter unten verwendet. Hier eine kurze Beschreibung der Punkte:

- 1 Reverse Proxy: Er kann zur vorgelagerten, externen Authentisierung und Autorisierung verwendet werden. Seine Nutzung wirft interessante Fragen in Bezug auf die Weitergabe dieser Entscheidungen auf und wie innerhalb der Applikation darauf zugegriffen werden kann.
- 2 Web-Server: Plug-Ins zur Kontrolle von Authentisierung und Autorisierung gibt es z. B. für den Apache-Web-Server genügend. Da der Reverse Proxy konzeptionell nicht viel anders ist als ein Web-Server, kann dieser auch im Wesentlichen die gleichen Funktionalitäten übernehmen.
- 3 Container Managed Security: Das bedeutet die Benutzung der deklarativen Sicherheit durch eine Konfiguration des JAVA-EE-Web-Containers. Sie setzt voraus, dass der Application-Server in die Firmeninfrastruktur integriert ist oder integriert werden kann, z. B. durch LoginModule von JAAS.
- 4 Das Interceptor Framework des Servlet Containers.
- 5 Das zentrale Servlet des Struts-Frameworks.
- 6 Der Plug-In-Mechanismus zur Erweiterung des ActionServlets.
- 7 Der RequestProcessor vor Ausführung einer Action.
- 8 Die Actions selbst.

- 9 JAAS: Erweiterbare Authentisierung und Autorisierung, entweder einzeln oder integriert in den Application-Server zu verwenden.
- 10 Business Delegates – die Schnittstelle zur Businesslogik. Soll hier der Request nochmals geprüft werden? Wenn EJBs verwendet werden, findet die Prüfung am Eingang der Komponenten zwangsläufig statt.
- 11 View-Klassen (JSPs). Auch sie könnten nochmals – oder erstmals – die Identität bzw. Rolle prüfen. Wenn sie Sicherheitsaufgaben erhalten, dann sollte ein Mechanismus existieren, der alle View-Objekte auf diese Weise sichert.
- 12 Java 2 Security: Wollen wir einen Request mit den Rechten des Aufrufers bezüglich des Codes laufen lassen? Dann benötigen wir ein Policy File mit entsprechenden Definitionen der Permissions. Wollen wir generell verschiedene Actions voneinander durch Code Access Security abgrenzen?
- 13 Datenbank: Sollen unsere Nutzer dort einzeln bekannt bzw. als Datenbankbenutzer registriert sein? Oder sind sie dort nur – bei eigener Authentisierung oder Autorisierung – als Daten präsent?

Ein Request auf eine Struts-URL läuft nun ungefähr in folgender Weise ab:

Durch die vorgelagerte, optionale Web-Infrastruktur gelangt der Request an den Web-Container bzw. die Servlet Engine. Dort kann der Request zunächst durch sogenannte *Servlet Filter* bearbeitet werden. Es handelt sich hier um ein im Servlet-Standard definiertes Interceptoren-Framework.

Im Anschluss daran gelangt der Request in das Action Servlet – die zentrale Anlaufstelle für alle Requests. Dort wird die Abbildung des Requests auf eine ausführende Klasse vom Typ Action an den RequestProcessor delegiert.

Nach erfolgter Abbildung wird die `process()`-Methode der entsprechenden Aktion ausgeführt und validierte Request-Parameter übergeben.

Der Aufruf der Businesslogik erfolgt über sogenannte *Business Delegates* und erzeugt Model-Helper-Objekte, die die Ergebnisse beinhalten.

Der Request wird – gesteuert durch eine Konfigurationsdatei, die `struts-config` – an die entsprechende View (ein JSP) zum Rendern weitergeleitet.

Das View-Objekt nutzt die Helper-Objekte als Datencontainer und rendert deren Inhalt in eine HTML Page, die dann zum Aufrufer geschickt wird.

Da es sich bei Struts um ein erweiterbares Framework handelt, ist es wichtig, die bereits vorhandenen Erweiterungspunkte zu verstehen:

Plug-Ins: Das zentrale Servlet kann durch eine Applikationslogik erweitert werden, vor allem, wenn etwas spezielles beim Start oder Shutdown nötig ist.

RequestProcessor: Durch ihn gelangen alle Requests einmal. Dort kann beispielsweise geprüft werden, ob der Nutzer authentisiert ist oder ob er die nötigen

Rechte besitzt, eine Aktion auszuführen. Die Klasse besitzt einen Hook für das Einhängen von Applikationscode, jedoch wird dieser nur einmal aufgerufen.

ActionServlet: Ursprünglich oft für Erweiterungen durch Applikationen genutzt, ist die Ableitung eine neuen zentralen Servlet-Klasse für Struts eher die Ausnahme geworden. Für die Kontrolle der Autorisierung ist sie ohnehin wenig geeignet, wie wir gleich sehen werden.

Action-Klasse: Für jeden Request-Typ wird eine Instanz dieser Klasse benötigt. In früheren Versionen von Struts immer ein Singleton, kann dies jetzt über die Konfiguration gesteuert werden.

View-Klasse: Zur Erzeugung der Präsentation wird der Request an eine Instanz dieser Klasse – hier JSPs – geleitet.

Die Eigenheiten des Frameworks müssen verstanden werden, andernfalls können, wie wir noch sehen werden, größere Sicherheitslücken entstehen.

Bezüglich der Authentisierung gibt es einige Entscheidungen zu treffen:

- Wird die Struts-Applikation die Authentisierung selbst durchführen oder durchführen lassen? Dafür kämen dann Reverse Proxy, Web-Server oder Container in Frage.
- Wenn die Authentisierung extern durchgeführt wird, wie wird der Principal (der authentifizierte Nutzer) in die Struts-Applikation gebracht?
- Sind die Benutzer der Applikation eine eigene Domain/Realm oder Teil einer existierenden Domain/Realm? Entsprechend muss die Applikation selber eine Domain pflegen oder kann das anderen überlassen. Davon hängen dann Funktionen zur Nutzerverwaltung wie Anlegen, Löschen und Ändern ab.
- Sind unterschiedliche Gütegrade der Authentisierung erforderlich? Falls ja, wie und wo wird das Ergebnis festgehalten?

Die Frage der eigenen Authentisierung hat große Konsequenzen für die Wartbarkeit, aber auch für die Sicherheit der Applikation. Um wartbar zu sein, sollte die Applikation verschiedene Formen von Authentisierung unterstützen können und bei neueren Formen nicht sofort größere Änderungen im Code benötigen. Eigene Authentisierung bedeutet unter Umständen auch, Schnittstellen zu existierenden Verzeichnissen herzustellen, z. B. über LDAP APIs (hierbei bitte die Prüfung durch einen `bind(UserId, Passwort)`-Aufruf durchführen und *nicht* mit Administratorrechten die Einträge des Verzeichnisses selber auswerten – dieses Vorgehen besitzt viel zu viel „Ambient Authority“ und nimmt außerdem der Registry die Möglichkeit, das Passwort auszulagern).

Der Idealtyp einer völlig selbständigen Applikation auf Struts-Basis würde Nutzerinformationen, insbesondere Credentials, in eigenen Tabellen der Applikationsdatenbank ablegen. Die Applikation würde nun Log-in, Log-out und Log-in-Error-Seiten erstellen und diese für den unauthentisierten Zugang freigeben. Anschließend müsste man, um einen Choke-Point für sämtliche anderen Res-

sourcen zu finden, an einer zentralen Stelle eine Authentisierungsprüfung veranlassen. Für diese zentrale Stelle bieten sich bei einem Blick auf Abb. 6.35 mehrere Stellen an.

Servlet Filter und Request Processor scheinen besonders gut für die Authentisierung geeignet zu sein. Die Requests durchlaufen beide Knoten mit Sicherheit, bevor Aktionen durchgeführt werden. Sicherheitstechnisch ideal ist natürlich ein besonders frühes Abfangen unauthentisierter Requests, damit solche gar nicht erst in das Innere der Applikation gelangen können. Hier hat der Servlet-Filter einen gewissen Vorteil gegenüber dem Request Processor. Dadurch müssen login/logout/loginError gar nicht als Actions codiert werden – mit den entsprechenden Ausnahmen in Request Processor zum Durchlassen dieser speziellen nicht-authentisierten Requests.

Besser ist es allerdings, die nicht-authentisierten Requests bereits im Servlet-Filter abzufangen und auf eine Login-JSP umzulenken. Der Servlet-Filter prüft die Credentials der Nutzer durch Datenbankzugriffe. Nach erfolgreicher Authentisierung werden die User Credentials in einem neu erzeugten User- oder Principal-Objekt geschrieben und in der neu erzeugten Session gespeichert.

So oder ähnlich wird in vielen Web-Applikationen die Authentisierung gehandhabt mit den Zielen,

- einen gemeinsamen Einstieg zu definieren und
- eine harte und genaue Grenze zwischen authentisierten und nicht-authentisierten Zugriffen zu schaffen.

Unterschiedliche Formen oder Qualitätsstufen der Authentisierung könnten ebenfalls vom Servlet-Filter erzeugt und im `user`-Objekt festgehalten werden.

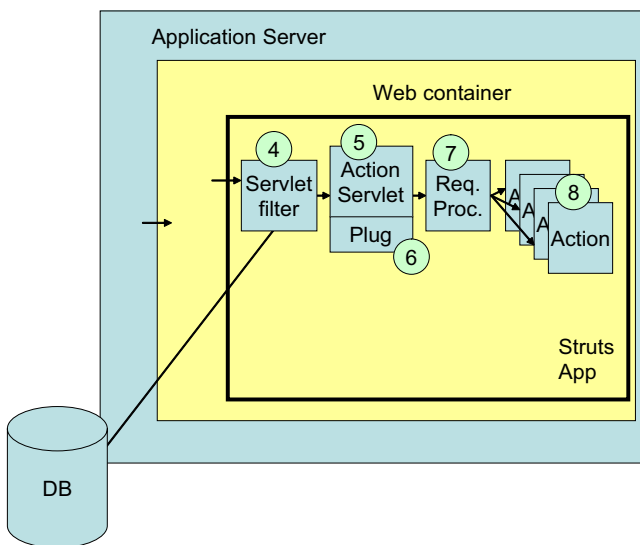


Abb. 6.35 Architektur einer Applikation auf Basis von Struts

Hier entsteht eher das Problem, wie der Rest der Applikation damit umgeht. Dies ist jedoch eine Frage der Autorisierung. Nachteile dieser Lösung sind:

- Die Authentifikation ist eine Insellösung, losgelöst von der Infrastruktur der Firma.
- Änderungen im Authentisierungsmechanismus erfordern Änderungen der Applikation.
- Die Applikation ist verpflichtet, ihre Nutzer selber zu verwalten, also bei Personalwechsel etc. die neuen Informationen einzupflegen und Funktionen zur Verwaltung zu implementieren.
- Unauthentisierte Requests erreichen die Maschine, auf der die Struts-Applikation läuft. Dies ist im Sinne einer sicheren Infrastruktur wenig wünschenswert.

Eine bessere Isolierung von Authentisierungsmechanismen und eine bessere potenzielle Einbindung in vorhandene Infrastruktur bietet die Verwendung von JAAS – dem bereits beschriebenen Java-Framework zur Authentisierung und Autorisierung. Für eine professionelle Applikation wird man wahrscheinlich ein Modul verwenden, das gegen eine Datenbank oder ein LDAP authentisieren kann. Dieses Modul – oder evtl. Module – wird in eine Login-Config-Datei aufgenommen und entweder über die Kommandozeile oder über einen Eintrag in das Java Security File der Installation des Application-Servers gestartet.

Die Integration in Struts erfolgt – etwas anders als in [Moo] vorgeschlagen – hier bereits im Servlet-Filter, der die üblichen Schritte der Authentisierung vornimmt:

```
try {  
    lc = new LoginContext("DBLogin", new  
        MyCallbackHandler(username,password));  
    lc.login();  
} catch (LoginException le) {...}  
return lc.getSubject();
```

Das Login-Modul füllt das Subject mit verschiedenen Principals und Credentials aus der Datenbank. Dazu benötigt das Login-Modul einen Zugang zur Datenbank mit den Authentisierungsdaten. Sollten die Login-Module bereits in die Sicherheitsinfrastruktur eines Application-Servers integriert sein, so benötigen manche Zugriffe bereits entsprechende Principals oder Credentials. Eine elegante Möglichkeit, diese den Modulen zur Verfügung zu stellen, wird beim JBOSS-Application-Server beschrieben: Sie können durch ein spezielles Login-Modul ins Subject bei `login()` eingetragen werden und stehen diesen damit schon beim Aufruf der weiteren Login-Module zur Verfügung.

Das so authentifizierte und gefüllte Subject-Objekt wird anschließend im Session Cache gespeichert. Damit ist die Basis geschaffen, auf beliebige eingetragene

Principals später in der Applikation über `checkPermission(...)` eine Prüfung zu veranlassen.

Noch eleganter als die eben beschriebene Integration mittels JAAS ist natürlich die Verwendung der Container-Managed-Authentisierung des Java EE. Dazu müssen den zu schützenden Web-Ressourcen Methoden und Rollen zugeordnet werden. Dies geschieht im File `web-app.xml`:

```
<login-config>
<auth-method>FORM</auth-method>
<form-login-config>
<form-login-page>/logon.jsp</form-login-page>
<form-error-page>/logonError.jsp</form-errorpage>
</form-login-config>
</login-config>
```

Die Konfiguration der entsprechenden Login-Module ist abhängig vom Application-Server. Wenn der Web-Container nicht bereits korrekt über die entsprechenden JAAS-Module in die firmeneigene Infrastruktur eingebettet ist, kann dies die Applikation selber veranlassen. Bezüglich der Integration in die Struts-Applikation gibt es jetzt jedoch Unterschiede: Die Applikation muss nicht mehr selbst die unauthentisierten Zugriffe abfangen, sondern spezifiziert lediglich die entsprechenden Login- und Logout-Pages, die vom Application-Server für das Log-in verwendet werden sollen (jedenfalls bei der heute fast ausschließlich verwendeten Forms Based Login). Dafür kann die Applikation über die von Java EE definierten Methoden an Principal-Informationen herankommen bzw. programmatisch abfragen, ob ein Nutzer eine bestimmte Rolle besitzt. Die Frage, wo und wie die Auto-risierungsinformation genutzt wird, klären wir weiter unten in Abschn. 6.4.4.

Diese Lösung ist sicherheitstechnisch besser als die vorigen – wenn die Konfiguration der Applikation stimmt! Ansonsten entstehen große Sicherheitslöcher, da die Authentisierung nicht mehr zwangsläufig im Code, sondern außerhalb der Applikation stattfindet. Ein weiterer Nachteil besteht darin, dass unauthentisierte Zugriffe immer noch den Application-Server erreichen und eine Einbindung in eine evtl. vorhandene Single-Sign-On-Infrastruktur nicht vorhanden ist.

Wie würde so eine Integration in eine SSO-Infrastruktur aussehen? Mit dieser Frage verlassen wir den Zweig der eigenen Authentisierung durch Applikation oder Container und gehen über zur vorgelagerten, externen Authentisierung über einen Reverse Proxy (RP): Auf einem Authentisierungsserver, der mit dem Reverse Proxy gekoppelt ist, werden Forms Pages zum Zwecke der Authentisierung deponiert. Kommt ein nicht-authentisierter Request von einem Client am RP an (dieser erkennt das z. B. an einem fehlenden Session-Cookie oder daran, dass eine SSL-Session gerade erst aufgebaut wurde), dann leitet der RP den Request auf eine Login Page um. Diese befindet sich am Besten außerhalb der eigentlichen Applikation, um, wie oben erwähnt, unauthentisierte Requests überhaupt nicht auf den Application-Server gelangen zu lassen. Der Client füllt nun die Login Page

aus und der Authentisierungsserver prüft die Credentials des Clients gegen ein LDAP oder eine Datenbank.

Falls der Client authentisiert werden konnte, muss diese Tatsache den folgenden Applikationen auf irgendeine Weise mitgeteilt werden. Dies kann durch die Erstellung eines CSIV2-Tokens, durch private Tokens oder auch schlicht nur durch das Einfügen des Client-Namens in den Remote User Header des http-Requests passieren. Die möglichen Mechanismen sind teils von den Authentisierungsservern bzw. den verwendeten Application-Servern abhängig und werden im Kapitel zur Sicherheit der Application-Server noch genauer beschrieben. Abhängig von der gewünschten Absicherung und den Performanceanforderungen können zur Sicherung des Tokens symmetrische Schlüssel oder auch digitale Signaturen verwendet werden. Handelt es sich um Clusterlösungen, so müssen die nötigen Schlüssel bzw. Zertifikate auf alle beteiligten Maschinen gebracht und gegebenenfalls auch erneuert werden.

Egal ob über Token oder einen Header-Eintrag – die Authentisierungsinformation muss den ursprünglichen Request des Clients zur Applikation begleiten. Natürlich stellt ein Token im Prinzip eine unterschriebene Erklärung des Authentisierungsservers dar, die besagt, dass der Client erfolgreich authentisiert wurde – und ist damit so gut wie eine Eintrittskarte für weitere Applikationen. Der – ungeschützte – Eintrag des Client-Namens in den http-Header ist natürlich noch kritischer zu sehen. Abhängig von der Struktur des verwendeten Netzwerks ist für die Verbindung vom Authentisierungsserver über den Reverse Proxy zum Application-Server die Verwendung von SSL mit gegenseitiger Authentisierung zwingend nötig, damit die Tokens nicht abgefangen bzw. gefälschte Header eingeschleust werden können.

Auf Seiten der Applikation stellt sich jetzt die gleiche Frage wie bei der nicht-externen Authentisierung oben: Soll die Übernahme der externen Authentisierung durch die Applikation selbst vorgenommen werden – und wenn ja, an welcher Stelle und durch welchen Mechanismus? Im Falle der Container Managed Authentication muss der Application-Server so konfiguriert werden, dass er die Information vom Authentisierungsserver dem Request entnehmen und ein Subject erzeugen kann, das diese Informationen dann enthält.

Besonders interessant ist in diesem Zusammenhang die Möglichkeit, bei der Ausstellung des Tokens weitere Daten über die Authentisierung dort zu hinterlegen: Beispielsweise, bei welchem Reverse Proxy der Client sich angemeldet hat, um wieviel Uhr, wie lange die Authentisierung gültig sein soll, bis hin zu Rechten oder Organisationszugehörigkeiten. Diese Metadaten können dann einen Request besser qualifizieren und sogar sein Routing steuern. Das Hauptproblem dabei ist jedoch zu garantieren, dass

- diese Metadaten tatsächlich im Verlauf eines Requests überall hin propagiert werden,
- die Metadaten über einen Standardmechanismus für Zugriffskontrolle genutzt werden können (z. B. Permission Checks) und
- eine systemweite Sicherheitsschicht dafür sorgt, dass die Metadaten tatsächlich auch berücksichtigt werden.

Der letzte Punkt ist besonders im Zusammenhang mit unterschiedlichen Qualitätsstufen der Authentisierung wichtig: Es muss vermieden werden, dass Services, die eine starke Authentisierung benötigen, in der Folge von Requests aufgerufen werden, die selbst nur schwach oder gar nicht authentisiert wurden. Falls das nicht durch Systemsoftware garantiert werden kann (d. h. programmatisch von den Applikationen geprüft werden muss), ist es sicherer, nur zwei Formen der Authentisierung (stark und gar nicht) zuzulassen und die nicht-authentisierten Requests in einer besonderen Zone der Firewall abzufangen (dies können z. B. Zugriffe auf statische Seiten eines Web-Servers, der die Homepage der Firma trägt, sein).

Für den Fall, dass die Applikation keine Container Managed Authentication will, bietet sich der gleiche Mechanismus wie bereits oben geschildert an: Ein Servlet-Filter könnte die Authentisierungsdaten dem http-Request entnehmen und über den üblichen JAAS-Mechanismus ein Subject erzeugen.

Im Falle der Container Managed Authentication muss der Application-Server so konfiguriert werden, dass die Authentisierungsdaten dem Request durch einen Interceptor entnommen und über JAAS verarbeitet werden. Alle drei großen Application-Server (Weblogic, Websphere und JBoss) bieten solche Interceptoren an. Daneben bieten die meisten Application-Server noch proprietäre Token-Formate an.

Der Effekt ist in beiden Fällen ziemlich gleich: Die Applikation erhält Zugriff auf ein Subject – entweder durch eigene Methoden oder durch das bekannte `getPrincipal()` aus Java EE. Damit wären die Alternativen bezüglich Authentisierung ausgereizt. Langfristig besser, weil SSO-fähig, ist zweifellos die Container Managed Authentication mit vorgelagerter Authentisierung durch einen Reverse Proxy. Hier ist die Applikation am Besten vor unauthentisierten Requests, geänderten Formen der Validierung oder generellen Änderungen der Infrastruktur geschützt. Außerdem kann sie über bekannte und standardisierte Methoden auf Principal- und Rolleninformationen zugreifen.

Die Sicherheitsspezialisten werden diese Lösung ohnehin befürworten, denn die Applikation erhält dabei keinen Zugriff mehr auf das Passwort der Benutzer und kann es daher auch nicht mehr missbrauchen (also ungewollt protokollieren und zum Log-in in andere Systeme verwenden – sei es gewollt oder als Angriff). Damit werden die Angriffsmöglichkeiten auf das Geheimnis der Benutzer wesentlich geringer.

Ist die Infrastruktur bereits zum Single-Sign-On-System ausgebaut, kann die Applikation das Authentisierungstoken für weitere Requests verwenden. Für den Fall, dass die Applikation dennoch ein Passwort eines Benutzers für weitere Anmeldungen an anderen Systemen benötigt, besitzen die meisten Application-Server mittlerweile das Konzept eines *Tresors* (Vault), in dem die Geheimnisse der Benutzer gespeichert sind. Falls die Applikation die entsprechenden Zugriffsrechte besitzt, kann sie ein Geheimnis aus dem Vault auslesen und verwenden. Hierbei besteht der Vorteil, dass diese Aktion genau protokolliert werden kann. Ein automatischer Passwortwechsel in regelmäßigen und kurzen Abständen ist in solchen Fällen dennoch angebracht, um die Gefahren durch das Speichern von Geheimnissen zu verringern.

6.4.2 *Spezielle Probleme der Front-End-Applikationen*

In diesem Abschnitt besprechen wir einige spezielle Sicherheitsprobleme der Front-Applikationen, nämlich:

- Die Frage der administrativen Zugänge: Wie sichert man die Applikation in Bezug auf die mächtigen Administrationsfunktionen? Bei Front-Applikationen betrifft dies ganz speziell die Frage des Internetzugangs sowie die Qualität der Authentisierung.
- Was passiert mit Test-Log-ins, Test- und Wartungs-URLs? Diese bleiben häufig in Installationen erhalten, ohne dass dies den Betreibern klar ist.

Beide Punkte waren übrigens ebenfalls Schwachstellen in der OBSOC-Application der T-COM, wie sie eingangs geschildert wurde. Dort war das Einloggen unter Administrations-Accounts aus dem Internet möglich. Es galten dabei die gleichen Regeln für Administratorpasswörter wie für die normaler Kunden (nämlich keine außer der automatischen Generierung von Usernamen, die sehr leicht durchschaubar war). Eine stärkere Authentisierung, wie es für Admin-Funktionen eigentlich angebracht wäre, fand nicht statt. Dies führte dazu, dass ein Angreifer sehr leicht Namen und Passwörter von Administratoren erraten konnte und damit die Applikation massiv hätte manipulieren können. Hier zeigt sich die Stärke einer externen Authentisierung, am Besten vorgelagert über Reverse Proxies: Die Requests aus dem Internet bzw. dem Intranet benutzen dabei zwei verschiedene Eingänge und erzeugen unterschiedliche Metadaten (d. h. sie halten den Ort des Log-in fest), die anschließend zur groben Autorisierung von Requests genutzt werden können. Dies kann bereits im Reverse Proxy passieren oder aber auch erst in einem Servlet-Filter der Application. Für eine bessere defense-in-depth kann man ruhig beides vorsehen. Bei Container Managed Authentication sollten zudem beide Instanzen der Applikation verschieden konfiguriert werden. Die Trennung der Authentisierung über verschiedene Reverse Proxies hat zudem den Vorteil, dass der Log-in der Mitarbeiter mit deren eigenem Account und nicht unter einem Functional User erfolgen kann, da dann leicht gegen eine andere Domain/Realm/Registry authentisiert werden kann.

Der zweite Punkt – übrig gebliebene Test- oder Wartungszugänge – kann nach dem gleichen Muster gesichert werden: Die dafür nötigen Log-ins sind gar nicht aus dem Internet verfügbar. Um auch Attacken aus dem Intranet vorzubeugen, werden auch diese Funktionen in das Konzept allgemeiner Autorisierung eingebunden, wie wir es unten in Abschn. 6.4.4 gleich schildern werden.

6.4.3 Authentisierungsobjekte

Bevor wir auf die Autorisierungsmöglichkeiten eingehen, wollen wir die Bedeutung der Authentisierungsobjekte zeigen und diskutieren, warum die Handhabung dieser Objekte und ihre sichere Delegation so kritisch ist. Wir tun dies an einem Beispiel aus der Praxis: Eine Web-Applikation auf Basis von Struts.

Bei dieser Applikation trat zur allgemeinen Überraschung folgender Effekt auf: Wenn zwei Benutzer sich zur gleichen Zeit einloggten und die gleiche Aktion per URL auslösten, so erhielten entweder ein Benutzer oder beide Fehlermeldungen. Das System selbst vermerkte seltsame Exceptions bei der Ansprache der Datenbank. Beunruhigender war jedoch die Tatsache, dass in vielen Fällen einer der beiden Benutzer scheinbar die Identität wechselte und mit dem Namen des anderen Benutzers begrüßt wurde.

Die Applikation übernahm durch eigene Programmierung die ausgelagerte Authentisierung durch einen Reverse Proxy. Die Prüfung der Rechte des Benutzers wurde durch eigenen Code bewerkstelligt, wobei die Rechte selbst durch einen Batch-Prozess täglich aus einem zentralen Benutzerberechtigungssystem geholt und in die Datenbank der Applikation importiert wurden (ein gutes Mittel, wenn aus Performance- oder API-Gründen keine Onlineprüfung möglich ist und außerdem eine Lösung, die sehr gut in verteilten Umgebungen skaliert. Sie ist den Versuchen, ein Single-Sign-On-System durch Replikation der Benutzerdaten in verschiedene lokale Repositories zu erreichen, nicht unähnlich).

Nach umfangreichen Debugging-Sessions fanden wir folgendes heraus: In der Architektur der Applikation gab es ein Missverständnis bezüglich der sog. Action-Klassen im Struts-Framework. Diese Klassen repräsentieren Aktionen bezüglich der Businesslogik und werden im GUI durch URLs repräsentiert. Sie bilden diese URLs auf entsprechendes Verhalten ab, das zur Anzeige einer neuen Page führt.

Diese Klassen sind in den älteren Struts-Versionen immer Singletons (neuerdings kann man dies durch Konfigurationsparameter steuern – die Auswirkungen dieser „Flexibilität“ auf die Sicherheit mag nach diesem Beispiel jeder selber beurteilen).

Singleton-Objekte gibt es für jede Art von Aktion, also genau einmal. Da eine Aktion durch verschiedene oder den gleichen Benutzer gleichzeitig mehrfach aufgerufen werden kann, muss dieses Objekt zustandslos programmiert werden. In anderen Worten: Es darf sich nichts in Member-Variablen merken, was nicht für alle Requests der gleichen Art gelten würde. Es darf auch nicht durch einzelne Requests modifiziert werden. Muss pro Request ein State gehalten werden, so darf dies nur über den Request-Parameter bzw. über den Session Kontext eines Requests geschehen.

Die Entwickler hatten dies missverstanden und speicherten innerhalb der Action Objekte für jeden Request in die Database Connection. Führt nun zwei Benutzer die gleiche Aktion annähernd gleichzeitig aus, wurde die Database Connection des ersten durch die des zweiten überschrieben. Hatte der zweite Benutzer den Request schon fast beendet, war die Connection wahrscheinlich bereits wieder

freigegeben. Versuche, den Request des ersten Benutzers fortzuführen, endeten in Exceptions über illegale States in der Datenbankbenutzung.

Im Objekt, das in der Action-Klasse gespeichert wurde, waren aber auch die Daten des aktuellen Benutzers enthalten – dies erklärte, warum die Benutzer plötzlich die Identität wechselten. Zugriffskontrollen vor der Benutzung einer Aktion verwendeten genau dieses Objekt.

Gemessen an seiner Schwere, war dieser Fehler relativ lange im Code versteckt. Nun handelte es sich glücklicherweise um eine wenig sicherheitskritische Applikation, aber der Gedanke, den gleichen Effekt in einer E-Business-Applikation zu sehen, ist doch ziemlich beunruhigend. Das Beispiel ist aus mehreren Gründen interessant:

- Es zeigt, wie notwendig ein genaues Verständnis der heutigen Framework-Architekturen ist.
- Es erklärt auch gut das Misstrauen der Sicherheitsfachleute gegenüber den Softwareentwicklern: Es passiert einfach zu häufig, dass fatale Fehler im Bereich der Authentisierung oder Autorisierung im Code entdeckt werden.
- Es bestätigt die Richtigkeit der Strategie, Basistechniken der Sicherheit wie Authentisierung und Autorisierung aus der Applikation fernzuhalten.
- Es zeigt, dass die Erlaubnis, Objekte zu manipulieren, die ein Ergebnis von Authentisierung sind (also z.B. das JAAS Subject), extrem gefährlich und mächtig ist: Wer ein Subject im gegenwärtigen Thread austauschen kann, beeinträchtigt auch die Sicherheit nachfolgender Komponenten (wenn sie den Vorgängern Vertrauen entgegenbringen und nicht auf vollständiger Delegation bestehen).
- Es zeigt die Gefahr selbst gebauter Autorisierungen, bei denen Subjects beliebig manipuliert werden können.

Entdeckt die Applikation, dass sie Rechte (Principals/Credentials) auf einfache Weise selber austauschen kann, dann können auf diese Weise schnell Stellvertretersysteme wie oben geschildert implementiert werden – meist mit recht tragischem Ausgang, da eben solche komplexen Funktionalitäten nie durch den einfachen Austausch von Rechten ohne Berücksichtigung der Semantik erfolgen sollten.

6.4.4 Autorisierung

Bezüglich der Implementation von Autorisierung stellen sich eigentlich die gleichen Fragen wie oben bei der Authentisierung:

- Eigene oder fremde Verwaltung der Daten?
- Externe (Reverse Proxy), vorgelagerte (Container) oder selbst implementierte Zugriffskontrolle (wo und wie)?

Die fremde Verwaltung der Autorisierungsdaten bedeutet meist nicht nur das Ablegen in einem externen Repository, sondern auch, dass die dort vergebenen Rechte und Principals durch einen externen Validierungsprozess gelaufen sind. Die meisten Applikationen verfahren in diesen Fällen so, dass sie die groben Autorisierungsrechte aus einer externen Quelle beziehen und diese dann auf applikationsinterne, feingranulare Rechte abbilden. In einigen Fällen enthalten die externen Registries jedoch nicht nur Rollen, sondern auch Instanzberechtigungen, z. B. auf einzelne Dokumente. Dies entspricht im Ergebnis dann eher einer Access Control List (ACL).

Applikationen sollten heute unbedingt in der Lage sein, externe Autorisierungsdaten zu beziehen. Für die eigene Verwaltung von Autorisierungsdaten genügt meist ein simples Datenbankschema mit Tabellen für Nutzer, Gruppen, Rollen und Rechte (Permissions). Um zur Laufzeit umfangreiche dynamische Kalkulationen von Rechten für einen User zu vermeiden, ist es geschickt, durch Views und Stored Procedures die endgültigen Rechte eines Users vorher auf der Datenbankebene bestimmen zu lassen. Neue Rechte für Rollen sollten einfach eingefügt und an alle betroffenen User automatisch propagiert werden können.

Eine heikle Frage im Zusammenhang mit Rechteänderungen ist immer die Frage, wie schnell und sicher alle gecachten Instanzen erneuert werden können. Viele Systeme cachen Rechte daher nur eine bestimmte Zeit und veranlassen danach ein neues Lesen der Rechte. Besonders in Umgebungen mit replizierten Applikationen können dadurch jedoch schnell Lücken in der Gültigkeit von Rechte-Updates entstehen.

Darf ein Recht auch einzelnen Usern gegeben werden, d. h. ohne Rollendefinition? Dies ist letztlich eine Frage der Granularität der Rechteverwaltung und kann im Extremfall sehr schnell zur völligen Unübersichtlichkeit der Rechteverteilung führen – wie auch bei ACLs von File-Systemen mit Einzelrechten. Letztlich muss die Rechtedefinition auch die Möglichkeit bieten, Rechte dynamisch zu erweitern. Java erlaubt dies z. B. dadurch, dass dynamische Policies zusätzlich zu statischen Definitionen beim `checkPermission()`-Call herangezogen werden können.

Kommen wir nun zur Frage, wo die Autorisierung am Besten stattfindet. Diese Frage ließ sich für die Authentisierung klar beantworten: So weit vorne wie möglich. Im Grunde gilt das auch für die Autorisierung: So können keine illegalen Requests ins Innere der Infrastruktur bzw. der Applikation gelangen. Allerdings gibt es einen zentralen Unterschied zur Authentisierung: Die Frage „wie oft muss eigentlich autorisiert werden?“. Das Prinzip der „Complete Mediation“ verlangt, dass jeder Zugriff autorisiert wird. Die Frage ist nur, was ist denn die Granularität eines „Zugriffs“? Ist ein Zugriff alles ab dem Beginn des Aufrufs durch den Client? Oder gibt es innerhalb eines Gesamtzugriffs bzw. einer Transaktion mehrere Autorisierungsstufen?

Fangen wir diesmal mit der externen, vorgelagerten Autorisierung an. Es gibt eine ganze Anzahl von Produkten, die dies ermöglichen, wie z. B. Netegrity Site-minder, IBM WebSeal etc. Diese Produkte verfügen über eine Konfiguration, die URLs auf Principals oder Rollen abbildet. Ein authentisierter Request auf eine URL wird dann zusätzlich grob durch einen Vergleich mit der Konfiguration auto-

riert. Nur wenn für diesen User die URL gestattet ist, wird der Request weitergeleitet.

Interessant ist jetzt das Verhalten der Applikation. Sie kann zunächst darauf vertrauen, dass eine Prüfung des Requests bezüglich des Users stattgefunden hat. Damit kann die zugehörige Action-Klasse im Fall von Struts ohne weitere Autorisierung aufgerufen werden. Aber jetzt beginnen die Probleme: Nehmen wir an, dass das Entwicklungsteam etwas größer geworden ist. Plötzlich fangen Entwickler an, einzelne Action-Klassen von anderen Action-Klassen aus aufzurufen. Diese Requests wurden jedoch nie von der vorgelagerten Autorisierung geprüft und es entsteht eine Lücke in der Mediation des Requests. Es ist diese Möglichkeit der Verzweigung von Requests im Code, die wiederholte Autorisierungen nötig machen.

Der nächste Schritt (nach der externen Autorisierung) wäre die Container Managed Authorisation mithilfe von Security Constraints. Hier ein kleines Beispiel:

```
<security-constraint>
<display-name>ExampleSecurityConstraint</display-name>

<web-resource-collection>
<web-resource-name>ExampleStrutsCollection</webresource-name>
<url-pattern>/struts/example</url-pattern>
<http-method>POST</http-method>
<http-method>GET</http-method>
</web-resource-collection>

<auth-constraint>
<role-name>exampleRole</role-name>
</auth-constraint>

<user-data-constraint>
<transport-guarantee>CONFIDENTIAL</transportguarantee>
</user-data-constraint>
</security-constraint>
```

Security Constraints arbeiten jedoch nur auf der ursprünglich angeforderten URI, nicht aber bei Requests, die über einen RequestDispatcher (z. B. `<jsp:include>` oder `<jsp:forward>`) laufen. Innerhalb der Applikation wird angenommen, dass die Applikation selber kompletten Zugriff auf alle Ressourcen besitzt und keine User Requests weiterleitet, sofern sie nicht entschieden hat, dass auch der Requester die nötigen Rechte besitzt.

Bei der Container Managed Authorization entsteht ein wichtiger Vorteil für die Applikation: Sie kann ab jetzt durch Standard-Interfaces auf ein Subject-Objekt zugreifen und es für weitere Autorisierungen verwenden. Somit sind wir im Inneren der Applikation angekommen und müssen die Fragen klären, wo und wie die Autorisierungen durchzuführen sind.

Dazu sind die folgenden Schritte nötig:

- Definition der zu schützenden Ressourcen bzw. Aktionen.
- Definition von Permissions, die den Zugang kontrollieren.
- Einfügen von Kontrollaufrufen vor dem Zugriff auf die Ressourcen, möglichst so, dass sie von keinem Entwickler vergessen werden können.
- Eventuell den gesamten Request durch Impersonation mit den Rechten des Callers ausführen lassen.

Fangen wir mit der Definition der Ressourcen an: Aus Abb. 6.34 wird schnell deutlich, dass die Action-Klassen direkt für verschiedene Requests stehen und damit eine wichtige Ressource darstellen. In Hinblick auf die Weiterleitung der Requests kommt jedoch auch der Schicht der Business-Delegates eine wichtige Funktion zu: Sie stehen für die Businesslogik der Applikation und sollten ebenfalls geschützt werden.

Die Definition der Permissions kann entweder etwas granularer sein als die Definition von Principals und Rollen oder aber diese genau reflektieren.

Der Ort von Kontrollaufrufen ist ebenfalls schnell gefunden. Da Aktionen untereinander aufgerufen werden können, müssen die Kontrollen unmittelbar beim Aufruf einer Aktion stattfinden. Damit ist eine vorgelagerte Implementation in Servlet-Filter, Action Servlet oder Request Prozessor ungeeignet.

Für die Impersonation müssen wir einen Access Control Context schaffen, der für den Caller steht und bei jeder Zugriffsentscheidung ausgewertet wird.

Bei der Frage des „Wie“ müssen wir unterscheiden, ob die Applikation die Autorisierung selber implementiert oder aber Container-Management dafür verwendet. Eine selbstgeschriebene Autorisierung für eine Struts-Applikation nutzt meist das File `struts-config.xml`, um die Aktionen mit einem Attribut für das Zugriffsrecht bzw. den Principal zu erweitern.

In der Nutzerverwaltung der Datenbank werden dann pro Nutzer entsprechende Rechte eingetragen. Spannend ist dann nur noch die Frage, wie sichergestellt werden kann, dass auch wirklich jede Aktion automatisch einer Zugriffsprüfung unterliegt.

Hierfür bietet sich das Template/Hook Pattern an: Es wird eine Basisklasse der Applikation von der Action-Basisklasse abgeleitet, die `process`-Methode überschrieben und als `final` deklariert. Diese `process`-Methode wird nun zur Template-Methode und ruft zuerst eine `pre-process`-Methode auf, in der die Zugriffsprüfung stattfindet. Anschließend ruft sie eine neu hinzugekommene applikationsspezifische `main-process`-Methode auf, die die Funktion der alten `process`-Methode übernimmt und den requestspezifischen Code aufnimmt. Abschließend kann noch eine Hook-Methode für post-processing aufgerufen werden.

Die Entwickler werden verpflichtet, diese neue Basisklasse als Basis für ihre Action-Klassen zu verwenden. Verletzungen dieser Regel lassen sich sehr einfach

in der Entwicklungsumgebung prüfen. Analog können wir beim Aufruf von Business Delegates verfahren.

Eine komplette Impersonation ist schwerer selbst zu entwickeln, da sie sich in die Java-Zugriffsmechanismen einklinken müsste. Daher ist es einfacher, das JAAS Framework dafür zu nutzen. Wir erweitern unsere obige Implementation der Authorisierung über JAAS im Servlet-Filter, indem wir die weiteren Zugriffs-Principals bzw. Rollen im Subject speichern. Sollten diese Daten in einer anderen Registry stehen, ist ein neues Login-Modul fällig.

Jetzt müssen wir für jeden zu schützenden Ressource-Typ eine Permission-Klasse definieren, die den Zugriff dann validieren kann. Dies kann durch die Ableitung einer BasePermission-Klasse geschehen.

In unserer Pre-Process-Hook-Methode instanziiieren wir die Permission und rufen checkPermission auf dem Security-Manager auf. Bevor wir das tun können, müssen wir in einem Policy File die Principals und ihre Permissions spezifizieren, damit das System die Zuordnung der Rollen zu den Rechten kennt. Die Permission prüft dann lediglich, ob beim Aufruf von checkPermission ein Eintrag für die entsprechende Rolle im Subject des Users vorhanden ist. Hier ein Beispiel eines Policy Files mit den Rollen „user“ und „admin“:

```
grant Principal * * {
    permission com.xor.auth.perm.URLPermission \
        "/struts-example/logoff.do";
};

grant Principal com.tagish.auth.TypedPrincipal "user" {
    permission com.xor.auth.perm.URLPermission \
        "/struts-example/editRegistration.do";
};

grant Principal com.tagish.auth.TypedPrincipal "admin" {
    permission com.xor.auth.perm.URLPermission \
        "/struts-example/editRegistration.do";
    permission com.xor.auth.perm.URLPermission \
        "/struts-example/adminMenu.do";
};
```

Wichtig ist, dass diese Prüfung tatsächlich im Kontext des richtigen Subjects – also des Aufrufers – vorgenommen wird. Die Subject.doPrivileged()-Methode des User Subjects sorgt dafür.

Ein Nachteil der Container-Managed-Autorisierung besteht allerdings darin, dass Änderungen der Rollendefinition bzw. der Rechtedefinition ein Redeployment der Applikation benötigen, während die selbst verwaltete Applikation dies dynamisch über die Datenbank regeln kann.

6.5 Light-weight Container Security: Das Spring Security (Acegi) Framework

Die Konzepte, mit denen im EJB- und JAVA-EE-Umfeld die speziellen Eigenschaften der Ausführungsumgebung (Kontext) sowie die Infrastrukturdienste für Transaktionen, Persistenz und Sicherheit (Concerns) aus dem Businesscode herausgehalten werden, sind nicht unumstritten. Viele halten die so entstandene Lösung für zu schwerfällig und umständlich und noch dazu nicht zwischen verschiedenen Containerherstellern portabel.

Mit dem Open-Source -basierten Security Framework Acegi [ACEGI] für den light-weight Container Spring (mittlerweile voll integriert als „Spring Security [SPRINGSEC]) existiert eine interessante Alternative zur Container-Managed-Security von JAVA EE. Im Folgenden wollen wir die zentralen Eigenschaften dieses Frameworks beleuchten und mit JAVA-EE-Konzepten vergleichen.

6.5.1 Philosophie

Spring Security definiert die Aufgaben von Enterprise Security ähnlich wie in JAVA EE, jedoch mit einem zentralen Unterschied: Spring Security folgt nicht dem Prinzip der Least Privilege Execution (POLA), d. h. Code selbst wird nicht wie bei der Java2 Security in seinen Rechten beschnitten. Die Designer von Spring Security sehen dafür im Enterprise-Umfeld keinen Grund: Dort gilt Software im Allgemeinen als vertrauenswürdig. Damit dürften sie die Einstellung der meisten Entwickler teilen, obwohl wir weiter oben Beispiele gesehen haben, wo Code Access Security auch bei Enterprise-Applikationen sinnvoll wäre, z. B. um File-Zugriffe zu beschränken.

Stattdessen hebt Spring Security die Bedeutung von instanzbasierter Zugriffskontrolle hervor und bietet ein Framework dafür an, das nicht nur Rollen, sondern auch Access Control Lists für individuelle Objekte verwalten kann. Hier stützt sich Spring Security auf die Interceptor- und Injection-Fähigkeiten, die Spring aus der aspektorientierten Programmierung übernommen hat.

Die Schnittstelle zur Businesslogik wird Service Layer Security genannt und kann entweder durch Rollendefinitionen oder eben instanzbasierte ACLs gesichert werden.

Das Filtern von Web Requests ist von JAVA-EE-Web-Containern bzw. den Servlet APIs bekannt. Spring Security geht hier über die rein rollenbasierte Definition von Zugriffskontrollen hinaus und kann auch Werte von Parametern bei der Entscheidung berücksichtigen.

Darüber hinaus bietet Spring Security die üblichen Fähigkeiten eines Sicherheitsframeworks wie z. B. die Durchführung einer Authentisierung, die Integration mit JAVA-EE-Container-Managed-Security, die Unterstützung von externer, vorgelagerter Authentisierung z. B. durch das Siteminder-Produkt von Netegrity/Com

puter Associates (siehe dazu <http://static.springframework.org/spring-security/site/index.html>).

6.5.2 Grundprinzipien

Sicherheit wird in Spring Security als ein „Aspekt“ gesehen, der im Wesentlichen aus der Businesslogik herauszuhalten ist. Zugriffskontrollen werden konfiguriert anstatt programmiert. Dies ist in einem klaren Kontrast zur Implementation der Java 2 Security bei der ja im Code durch Aufrufe von `SecurityManager.checkPermission(Permission)` die Kontrolle aktiviert wird. Der Ansatz deckt sich jedoch ganz gut mit dem Prinzip von EJB/JAVA EE, die Sicherheit ebenfalls aus dem Business-Code herauszuhalten.

In Spring Security kann somit jedes Objekt als sog. Secure-Object konfiguriert werden. Praktisch bedeutet dies, dass konfigurativ für die Methoden des Objekts ein Interceptor bestimmt wird, der dann die Zugriffskontrolle übernimmt (s. Abb. 6.36). Dies setzt natürlich die Fähigkeit des Grundframeworks (hier Spring) voraus, an beliebigen Stellen durch Proxies bzw. durch Techniken der aspektorientierten Programmierung Interceptor-Code einzufügen zu können.

Ja nach verwendeter AOP-Technologie werden unterschiedliche Interceptoren konfiguriert (z. B. AspectJ Interceptor, AOP Alliance Interceptor oder Filter Interceptor für Web-Requests). Alle Interceptoren operieren auf Konfigurationsobjekten, die die zu sichernden Methoden erwähnen [SPRINGSEC, S. 88ff.].

Erweiterbarkeit hat einen hohen Stellenwert in Spring. Sie wird durch die konsequente Verwendung von Interfaces anstelle von konkreten Klassen realisiert. Das ermöglicht verschiedene Implementationen, z. B. für verschiedene Formen des Datenhaltens (LDAP, Datenbank). Die wichtige Frage, welche Implementationen zur Laufzeit dann ausgewählt werden, löst Spring Security durch eine konsequente Verwendung des Provider-Patterns: Eine Manager-Klasse verwaltet eine

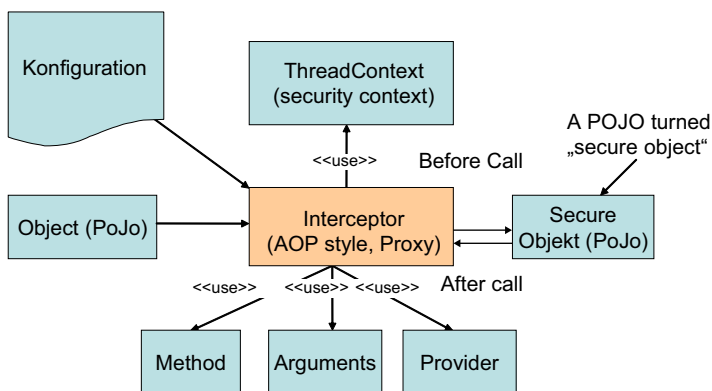


Abb. 6.36 Rolle des Interceptors in Spring Security

Reihe von Implementationen (Providern – z. B. für verschiedene Formen der Authentisierung). Wir werden verschiedene Formen der Verwendung dieses Patterns weiter unten kennen lernen.

Neben der Interceptor-Technologie ist es die einfache Konfiguration von Objekten zur Laufzeit, die als zweites Grundprinzip der Spring Security gesehen werden kann. Seit der Version 2.0 ist die früher aufwändige Konfiguration der Sicherheitsklassen und ihre Erzeugung und Verflechtung mit dem Rest des Spring Frameworks (das sog. „Wiring“) durch das Prinzip des Security Namespaces erheblich vereinfacht worden. Der Security Namespace ist ein eigenes Schema mit mächtigen Tags, die komplexe Sicherheitsanforderungen wie z. B. die Anbindung eines LDAP-Servers für Authentisierung durch ganz einfache Anweisungen im zentralen Spring Application Context File ermöglichen. [SPRINGSEC, S. 11ff.]. Die Funktionen des Security Namespaces umfassen die üblichen Funktionen der Sicherheit: Authentisierungsfunktionen und Optionen, Zugriffskontrolle der Businessmethoden, Absicherungen der Web-Requests, Absicherung der Business Requests sowie User-Details. Wir verzichten im Folgenden auf eine Darstellung der konkreten Optionen zur Absicherung von Kanälen sowie die unterschiedlichsten Formen der Authentisierung bzw. der Integration mit Containern und werfen stattdessen einen Blick auf die Architektur.

6.5.3 *Authentisierung*

Ziel der Authentisierung ist die Erstellung eines Objektes, das eine erfolgreich geprüfte Identität innerhalb der jeweiligen Domain bzw. Realm repräsentiert und das mit Credentials und Rechten gefüllt ist. Dieses Objekt kann dann in der Folge für Sicherheitsentscheidungen verwendet werden.

Spring Security und JAAS unterscheiden sich im Prinzip an dieser Stelle bis auf die üblichen Unterschiede in der Terminologie nur wenig. Was bei JAAS das „Subject“ darstellt, wird in Spring Security als „Authentication“ bezeichnet und ist ein Container für Sicherheitsinformationen. Dieser Container wird von dem konfigurierten Authentisierungs-Mechanismus (also basic authentication, forms-basierte Authentisierung etc.) mit den Daten des Users gefüllt und einem Authentication-Manager zur weiteren Bearbeitung übergeben. Dieser ist ähnlich dem JAAS-Login-Context für die Auswahl der richtigen Provider sowie deren Synchronisation zuständig. Ein Provider entspricht grob einem JAAS-Login-Modul und kann Daten, die vom Authentisierungs-Mechanismus gesammelt wurden, gegen eine Registry validieren (z. B.: LDAP oder Datenbank). In Abb. 6.37 ist ein DAO-Provider gezeigt, der über eine JDBC-Anbindung Details zum jeweiligen Nutzer abfragen kann.

Das Authentisierungs-Objekt fließt dabei zwischen Mechanismus, Manager und Provider und wird dabei mit Rechten (s. Abb. 6.37) aufgefüllt. Ein fertiges Authentication-Objekt ist die Grundlage aller weiteren Sicherheitsentscheidungen.

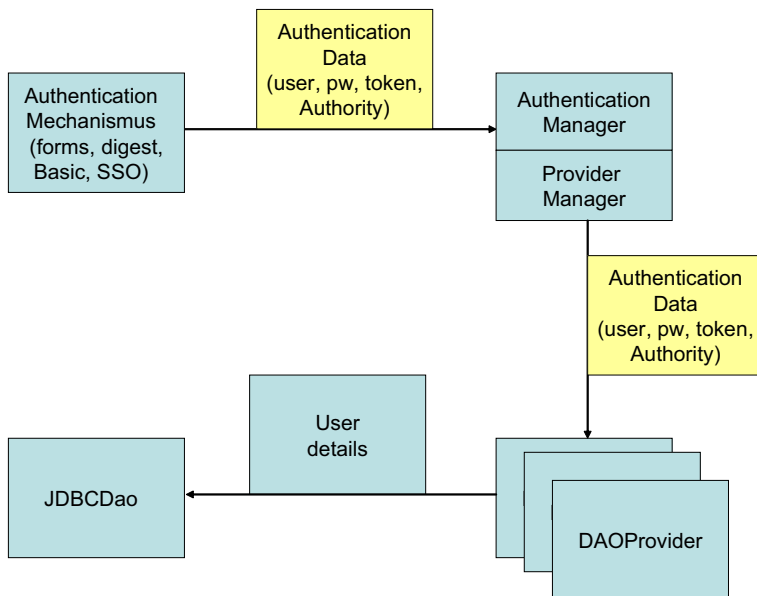


Abb. 6.37 Fluss eines Authentisierungsobjekts

6.5.4 Das Authentication-Objekt

Egal, ob Zugriffskontrollen fest programmiert sind oder ob sie durch AOP dynamisch in den Code eingespeist werden können – sie benötigen Zugriff auf die Authentisierungsdaten des Principals und seine Rechte, die typischerweise in einem Rechtecontainer gespeichert werden.

Wie kommt der Code der Zugriffskontrollen an dieses Objekt und wo wird es gespeichert? Speicherung ist wichtig, wenn z. B. eine Interaktion mit einem User aus mehreren Requests besteht. Und da beliebige Pojos gesichert werden können, muss ein Zugriff auf das Authentication-Objekt jederzeit möglich sein.

Vergleichen wir dies kurz mit JAVA EE: Dort existieren Methoden im Web- und EJB-Container, die den Zugriff auf Principal-Objekte erlauben (`Request` bzw. `Context.getPrincipal()`). Spring Security speichert während eines Requests das Authentication Object innerhalb eines `SecureContext`-Objektes und speichert dieses wiederum in einem `ThreadLocal`-Objekt namens `ContextHolder`. Damit stehen die Nutzerdaten während des gesamten Requests zur Verfügung. Bei JAAS hingegen werden Code und Subject durch eine statische Methode im Subject zusammengebracht: `Subject.doPrivileged(Subject, Action)`.

Für längeres Speichern der Authentifikations-Daten erlaubt Spring Security verschiedene Implementationen, z. B. in der User-Session.

Ein wichtiger Punkt beim Umgang mit Authentisierungsobjekten ist die Frage, wer sie manipulieren darf. Spring Security erlaubt mithilfe von sog. Runas-

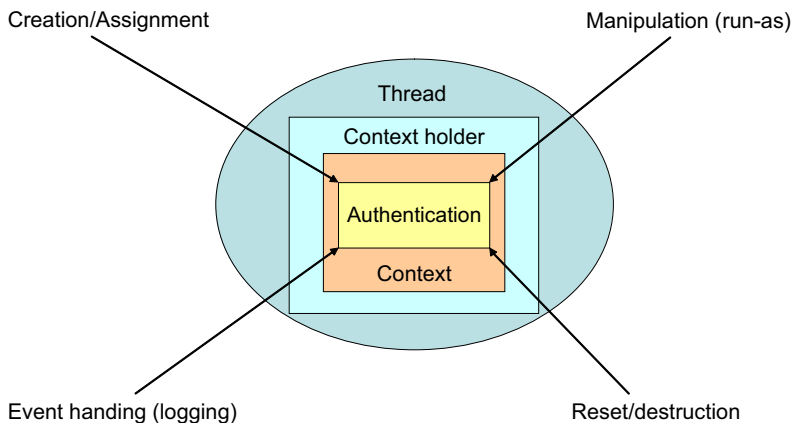


Abb. 6.38 Manipulation von Authentisierungsobjekten

Managern das Austauschen von Authentication-Objekten im momentanen Thread (s. Abb. 6.38). Dies erlaubt einer Applikation, unter einer anderen Identität bzw. mit anderen Rechten zu arbeiten und diese Identität auch noch an andere Rechner zu delegieren. Auch hier geht Spring Security davon aus, dass der Umgang mit diesem Mittel in Enterprise-Applikationen getestet und abgesegnet ist. Ansonsten stellt der Mechanismus natürlich auch eine erhebliche Missbrauchsgefahr da. Der Umgang mit Authentisierungsobjekten – sie stellen ja eine erfolgreiche Authentisierung dar – ist z. B. im Windows-Bereich an Systemprivilegien gebunden.

Wichtig in Bezug auf die Bindung der Nutzerdaten an einen Thread ist natürlich auch noch, dass am Ende des Requests die Referenz im Thread auf das Authentication-Objekt wieder auf Null gesetzt werden muss – sonst kann ein neuer Request mit der falschen Identität abgearbeitet werden!

Ähnlich den Principal-Objekten in JAAS, die häufig Rollen darstellen, funktionieren Granted-Authority-Objekte in Spring Security. Diese Rechteobjekte werden ebenfalls in Authentication-Objekten verwaltet.

Über verschiedene Wrapper-Klassen erlaubt Spring Security auch die Einbindung von Software, die für JAVA-EE-Container-Managed-Security geschrieben wurde. Dazu müssen in diesen Adaptionern die Subject/Principal-Strukturen in die entsprechenden Spring-Security-Objekte transformiert werden.

6.5.5 *Autorisierung*

Bezüglich der Autorisierung beschränken wir uns auf eine kurze Beschreibung des Grund-Frameworks sowie seiner Anwendung für instanzbasierte Zugriffskontrolle. Für das Filtern von Web-Requests verweisen wir auf die Referenzdokumentation zu Spring Security [SPRINGSEC].

Das erste Basispattern für die Autorisierung in Spring Security ist das Interceptor Pattern (Abb. 6.39). Durch die Definition von `<filter>`-Anweisungen in

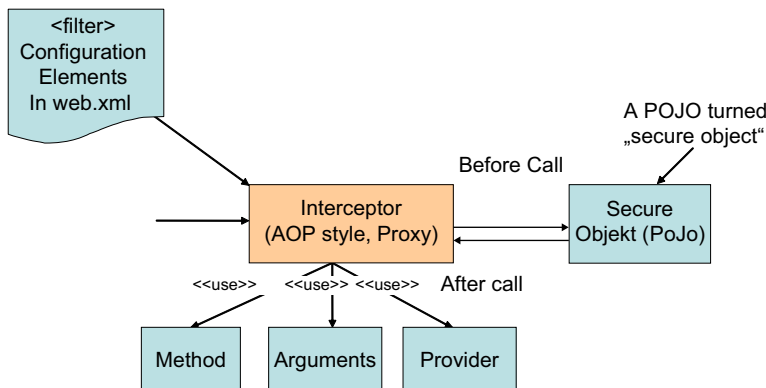


Abb. 6.39 Interceptor Pattern zur Autorisierung

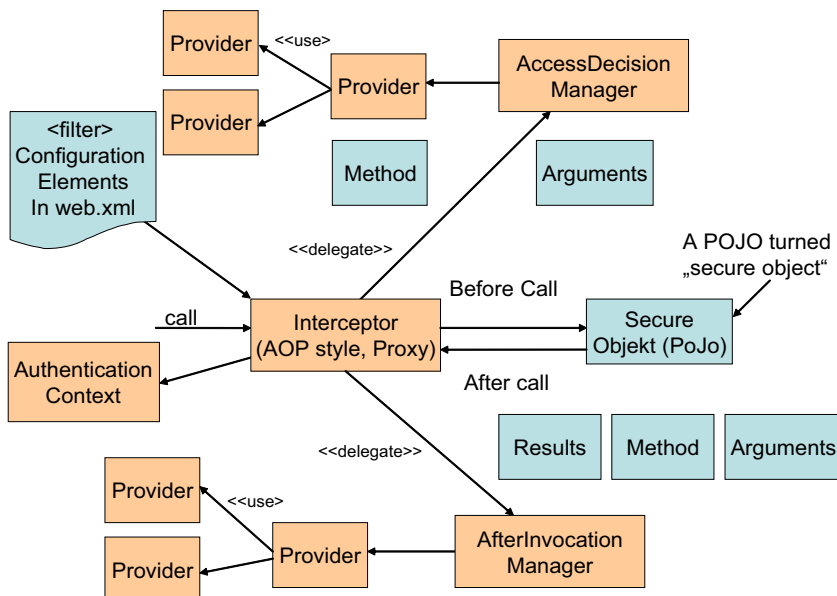


Abb. 6.40 Access-Decision und After-Invocation-Manager

web.xml werden Interceptoren beim Aufruf von Secure Objects getriggert. Konfigurationsparameter steuern dann bestimmte Filterungen.

Das zweite Basispattern ist das Provider-Pattern. Je nach verwendeter Implementation des Access-Decision-Managers werden bei der Zugriffskontrolle die verschiedenen Voter-Implementationen angefragt und das Ergebnis dann im Decision-Manager kombiniert. Im Grunde handelt es sich hier um eine Abwandlung des Chain-of-Responsibility-Patterns, wie es auch bei JAAS im Einsatz ist. Es erlaubt verschiedenen Implementationen, gemeinsam über etwas abzustimmen.

Vom Access-Decision-Manager gibt es eine Variante, die speziell für die Zugriffskontrolle nach dem Aufruf des Objektes zuständig ist: Der After-Invocation-Manager erhält die gleichen Objekte zur Entscheidungsfindung wie der Access-Decision-Manager, jedoch zusätzlich noch beim Aufruf des Secure Objects zurückgegebene Objekte (s. Abb. 6.40). Aufrufe des After-Invocation-Managers können dazu führen, dass die zurückgegebenen Objekte modifiziert werden, z. B. aus einer Collection bestimmte Objekte entfernt werden. Auch direkte Objektmanipulationen wären möglich (Felder zurücksetzen etc.).

Auf Basis dieses Interceptor-Frameworks bietet Spring Security nun die Möglichkeit, sowohl komplexe Sicherheitsregeln durchzusetzen, bei denen die momentanen Werte von Objekten direkt in die Zugriffskontrolle eingehen, als auch instanzbasierte Zugriffskontrolle durchzuführen. In diesem Fall sind die speziellen Rechte eines Principals für ein bestimmtes Objekt in einer Access Control List (ACL) festgehalten. Die Rechte sind also individuell pro Objekt und nicht pro Klasse des Objekts bestimmt.

Wie auch bei der Authentisierung in Spring Security kommt dabei erneut das Provider-Pattern zum Einsatz. Eine ACL-Manager-Implementation (konkret ein `AclAfterInvocationProvider`) kann nach einem Match zwischen den gültigen ACLs für ein Objekt und eines entsprechenden Principal gefragt werden. Mögliche Implementationen sind z. B. ACLs in einer Datenbank, die über einen Object Key (dazu muss das Domain Object transformiert werden) selektiert werden können [SPRINGSEC, S. 84ff].

Die Absicherung solcher Domain Objects, die eine eigene Auflistung von erlaubten Principals besitzen, kann auf verschiedene Arten erfolgen. Die Spring Security Referenzdokumentation listet verschiedene Möglichkeiten und ihre Vor- bzw. Nachteile auf.

1. Business-Code im Domain-Objekt-Code selbst vergleicht die Grant-Authority-Objekte aus dem Authentication-Objekt des Aufrufers mit den ACLs aus dem Domain-Objekt. Dies bindet Sicherheitscode unmittelbar an Business-Code und ist in der Folge schwer wartbar.
2. Vor dem Aufruf des Businessobjekts (bzw. dessen Konstruktion) wird der Sicherheitskontext des Aufrufers mit Grant-Authorities für ALLE Businessobjekte, auf die der User Zugriff haben soll, ausgestattet. Wenn der User für eine große Zahl von Domain-Objekten berechtigt ist, entsteht hier ein großer Overhead.
3. Eine Instanz eines Access-Decision-Voters prüft VOR der Instanziierung des Domain-Objekts, ob der User die nötigen Rechte besitzt. Nur bei Erfolg wird das Domain-Objekt instanziiert. Dies bedeutet ZWEI Zugriffe pro Objektinstanziierung auf die Datenbank.

Spring Security erlaubt jedoch auch den Einsatz des oben erwähnten `Acl-After-Invocation-Provider`, der ohne zusätzlichen Datenbankzugriff und ohne den Code des Domain-Objekts mit Sicherheitsabfragen zu belasten, die Bedingungen für den Zugriff prüfen kann. Allerdings ist es dafür nötig, dass das Auslesen der Datenbank für die Erzeugung des Domain-Objekts mit Administrationsrechten geschieht – die Einschränkung auf die User-Rechte erfolgt erst im Nachhinein durch den

Acl-After-Invocation-Provider. Die Lösung ist zwar performant, in der Sicherheit aber nicht gleichbedeutend mit einer Delegation des Users an die Datenbank im Sinne der oben diskutierten Ende-zu-Ende-Sicherheit, denn die Entscheidung wird im Code und nicht im Backend selbst getroffen.

6.5.6 Diskussion der Spring Security Security

Dank der durchgängigen Verwendung von Interfaces lassen sich praktisch alle Beteiligten dieses Frameworks – Manager, Provider, Items – durch andere Implementationen ersetzen. Konfigurationsattribute steuern bzw. triggern Interceptoren und Provider (s. Abb. 6.41).

Da Java kein Ersetzen von Methoden in Objekten zur Laufzeit gestattet, muss Spring Security ein Interceptor-Konzept zum Einfügen des Security-Codes verwenden. Der Interceptor-Code kann dabei generativ über Bytecode-Manipulationen oder dynamisch in Form von Proxy/Wrappern durch den Classloader zusammengestellt werden.

Unter der Prämisse, dass die Software keine bössartigen Teile enthält und außerdem fehlerfrei ist, kann Spring Security auf Least-Privilege-Konzepte verzichten. Das Framework selbst ist elegant. Ob sich die ACLs als instanzbasierte Kontrolle in großem Stil verwenden lassen, hängt sehr von der effizienten Implementation der Zugriffe ab. Spring Security bietet hier natürlich Caches und andere

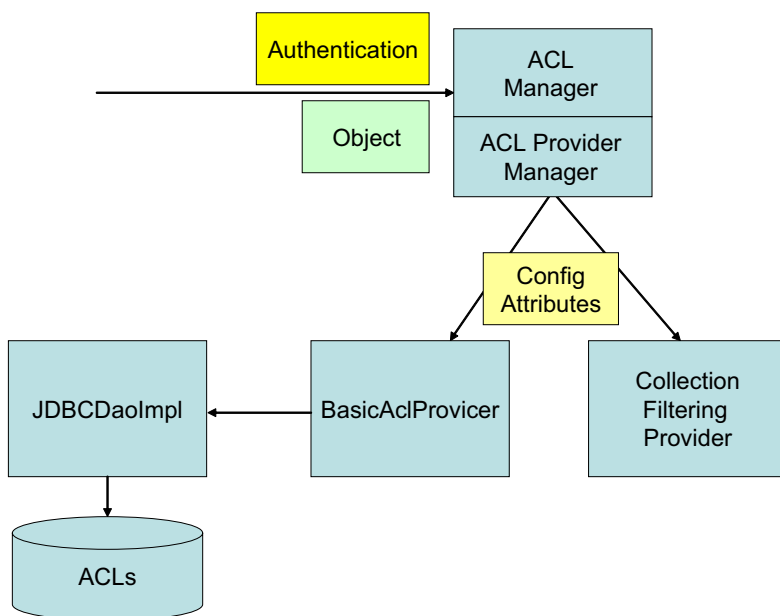


Abb. 6.41 Providersteuerung durch Konfigurationsattribute

Optimierungen. Alternativ dazu sollte jedoch auch der Einsatz von Rule Engines geprüft werden, die über den Interceptor-Ansatz ebenfalls eingebunden werden können. Es hängt letztlich stark vom Design der Businesslogik ab, ob sich ACL- bzw. regelbasierte Zugriffskontrolle eher lohnt.

Bei der Aktivierung der jeweiligen Provider/Voter-Implementationen findet erneut ein Chain-of-Responsibility-Pattern Anwendung. Die Provider werden nur tätig, wenn sie entsprechende Konfigurationsattribute in den XML-Konfigurations-Files erkennen, für die sie zuständig sind.

Mittlerweile geht der Trend gerade in Sachen Frameworks für Web-Applikationen noch weiter in Richtung leichtgewichtiger Ansätze. Neben Ruby-on-Rails beispielsweise erscheint selbst Spring bereits wieder als schwerfällig, und es bleibt abzuwarten, wie diese neuen Frameworks mit den Security-Fragen umgehen.

Spannend ist diese Frage nicht zuletzt deshalb, weil z. B. Ruby eher an Smalltalk erinnert und daher auch nicht statisch typisiert ist. Wie wird sich die dynamische Typisierung auf die Sicherheit von Applikationen auswirken? Ist das Fehlen statischer Typen ein Problem für die Input-Validierung? Die dabei auftretenden Probleme würden den Rahmen dieses Buches sprengen, deshalb sei an dieser Stelle nur auf die Dissertation von M. S. Miller ([Mil]) verwiesen, der Techniken wie Soft-Typing bzw. Trademarking behandelt, mit denen auch in dynamischen Sprachen sichere Objekte erzeugt werden können.

6.6 Grids

In einem *Grid* (dt. „Gitter“) wirken viele Rechner in einem organisationsübergreifenden Netz zusammen, um an einer gemeinsamen Aufgabe zu arbeiten. Um die verteilten Hard- und Softwareressourcen in einem Grid transparent nutzen zu können, werden sie – ähnlich wie in CORBA – der nutzenden Anwendung über eine Abstraktionsschicht zugänglich gemacht. Um die unterschiedlichen Komponenten des Grid untereinander interoperabel zu machen, ist eine starke Standardisierung unumgänglich. Alle Sicherheitsaspekte von Grids würden alleine bereits ein Buch füllen (ein solches ist kürzlich erschienen, s. [Chak]), und so wollen wir hier nur einige Aspekte der Grid Security herausgreifen, die auch im Bereich von Enterprise Security in Zukunft eine Rolle spielen könnten.

Die Notwendigkeit von Sicherheit in einer fast total verteilten Umgebung, wie sie ein Grid darstellt, stellt deutlich härtere Anforderungen an die Software, als eine gut kontrollierte und verwaltete Firmenumgebung.

6.6.1 *Grid Security versus Enterprise Security*

Grid Security unterscheidet sich von der normalen Enterprise Security deutlich, vor allem durch den Mangel an implizitem Vertrauen (s. Abb. 6.42).

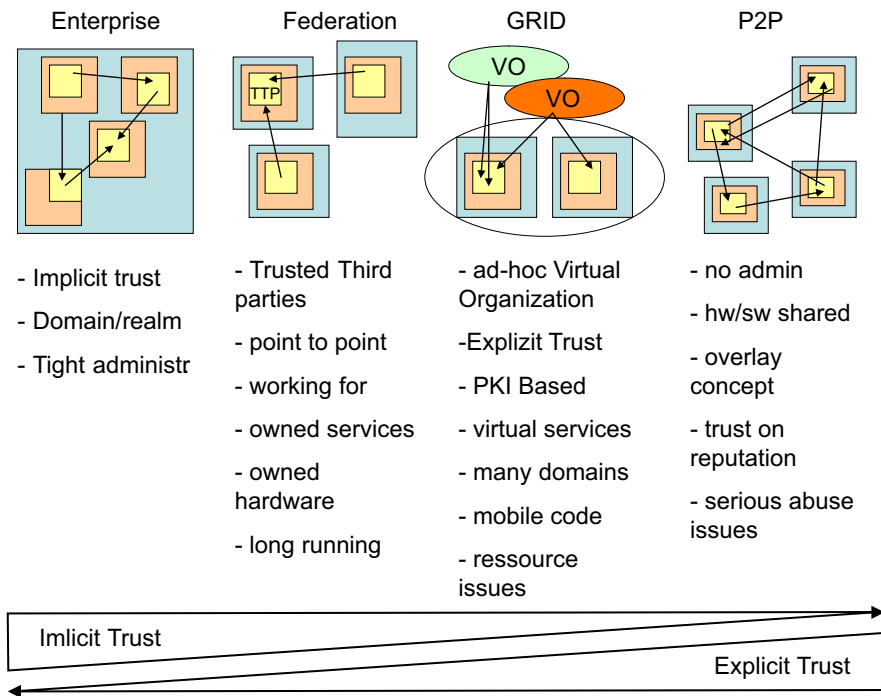


Abb. 6.42 Implizites vs. explizites Vertrauen in Strukturen mit zunehmendem Verteilungsgrad

Implizites Vertrauen bedeutet Vertrauen, das aufgrund von Verwaltung und Kontrolle von Rechnern und Software, die unter eigener Kontrolle sind, entsteht. So funktioniert Delegation in Enterprise-Netzwerken oft so, dass das Target einen Request eines Intermediate-Servers schlicht und einfach deshalb akzeptiert, weil er von einem internen Rechner kommt. Bisweilen werden sogar Log-in-Credentials von Nutzern auf diese Weise weitergegeben. Die Identität eines Aufrufers wird oft lediglich vom Intermediate behauptet, es erfolgt jedoch keinerlei Beweis dafür, dass der Client tatsächlich an diesem Request beteiligt ist. Genauso wenig hat ein Client die Möglichkeit, zu bestimmen, was mit seinem Request geschieht. Eine Nicht-Abstreitbarkeit von Requests ist ebenfalls kaum vorhanden und wird durch striktes Logging ersetzt.

Im Grid hingegen ist der Intermediate potenziell eine fremde Maschine und das unmittelbare – implizite – Vertrauen daher nicht angebracht. Es müssen also Mechanismen gefunden werden, die es dem Client wie auch allen anderen an einem Request beteiligten Parteien erlauben, den Request zu authentisieren. Die Weitergabe von Credentials oder die Verwendung von symmetrischen Keys, wie sie in Enterprise Clustern z. B. für die Validierung von SSO-Tokens verwendet werden, sind im Grid völlig ausgeschlossen.

Auch was die Nutzer betrifft, unterscheiden sich Enterprise und Grid klar: Der Requestor im Enterprise-Fall ist meist ein Kunde und damit Teil der Enterprise

Domain, die im Idealfall von einer Registry verwaltet wird. Im Grid-Fall ist der Requestor Teil einer virtuellen Organisation – d. h. einer ad-hoc gebildeten Organisation, deren Mitglieder keine Beziehungen zu den konkreten Providern der Maschinen des Grids zu haben brauchen. Sie sind dementsprechend in keiner Domain eines Grid-Providers enthalten. Dies bedeutet, es müssen Mapping-Verfahren eingesetzt werden, um den Requestor auf lokale Identitäten abzubilden.

Aus der Sicht der Grid-Provider stellt sich das Problem, dass fremde Identitäten Funktionen auf den eigenen Maschinen nutzen sowie eigene Ressourcen verwenden. Dafür will die Firma natürlich Gegenleistungen – sei es in Form von Geld oder CPU-Cycles – aber noch wichtiger als die Abrechnung ist natürlich, dass diese fremden Nutzer keinen Schaden an der eigenen Maschine anrichten können. Dies gilt auch umgekehrt für die Nutzer des Grid: Sie erhalten Ergebnisse bzw. Software von zunächst fremden Maschinen zurück, die im Prinzip der eigenen Infrastruktur schaden können. Auch dieses symmetrische Misstrauen ist in normalen Internet-Applikationen nicht vorhanden.

Wer sich mit dem Konzept des Compute Grids noch ein wenig schwer tut, kann sich auch über das Konzept des Internet-Service-Providers langsam an die Problematik herantasten: Die Infrastruktur des ISP fungiert als eine Grid-Komponente und hostet eine Reihe von Kunden des ISP. Diese Kunden können aus Sicht des ISP als simple virtuelle Organisationen verstanden werden. Sie besitzen wiederum eigene Kunden, die jedoch nicht Kunden des ISP sind. Trotzdem brauchen auch sie Zugang zu den Services des ISP.

Die Verwaltung der Infrastruktur wirft riesige Probleme auf, wie wir schon beim Application-Server diskutiert haben: Die Security sollte in der Lage sein, Verwaltungsoperationen an die jeweiligen Kunden zu delegieren, die sie unter Umständen wiederum – eventuell in reduzierter Form – an ihre Kunden weitergeben können. Die entscheidende Fähigkeit in diesem Umfeld ist die *Mandantenfähigkeit* – Features wie etwa ein globales Administratorpasswort für den ganzen Cluster würden diese unmöglich machen.

6.6.2 Grid Security Standards

Bevor wir auf die Themen Delegation, Identitäts-Mapping und Servicekontrolle genauer eingehen, ein Wort zu den wesentlichen Standards für das Grid Computing: Die Open Grid Services Architektur (OGSA) beschreibt die zentralen Teile der Grid-Architektur ([OGF]). Die Sicherheit selbst ist in der Grid Security Infrastructure ([GSI], [Wel]) spezifiziert und auch der Globus Programming Guide enthält ein sehr gutes Kapitel zu Grid Security ([Glob]).

Die Grid Computing Security setzt stark auf Web-Services-Security und die entsprechenden Standards. Es ist kein Wunder, dass sich die Entwickler der Grid Security für die Techniken der Web-Service-Security als Träger entschieden haben. Beide setzen auf objektbasierte Sicherheit und erlauben damit letztlich eine viel größere Unabhängigkeit von der Infrastruktur und eine Konzentration auf

Businessaspekte – und werden dadurch auch wieder sehr interessant für Enterprise-Infrastrukturen, die unter der mangelnden Flexibilität ihrer Systeme leiden. So kann man mittels objektbasierter Security leicht Knoten, die für die Businesslogik relevante Funktionen oder Services ausführen, von solchen, die lediglich Infrastrukturdienste wie z. B. Load-Balancing durchführen, unterscheiden.

Die großen Schwierigkeiten, ein komplettes Tracing über kanalbasierte Sicherheit herzustellen, führt zu innovativen Konzepten, die vor allem auf objektbasierter Sicherheit (Verschlüsselung und Signatur von Nachrichten) beruhen: Wenn den Überbringern von Nachrichten nicht komplett vertraut werden kann, dann muss die Sicherheit eben in der Nachricht selbst liegen.

6.6.3 *Qualifizierte Delegation durch Proxy-Zertifikate*

Damit kommen wir zu einem Kernkonzept der Sicherheit in Grids – die sichere Delegation von Aufträgen über sogenannte *Proxy-Zertifikate* ([RFC3820], siehe auch [WFK]).

Manche Entwickler sind überrascht, nachdem sie eine Grid-Umgebung wie z. B. Globus (siehe www.globus.org) herunter geladen haben, wenn sie im allerersten Schritt der Installation sofort aufgefordert werden, eine Certificate Authority zu erzeugen sowie verschiedene Zertifikate zu generieren. Dies macht deutlich, wie sehr die Sicherheit einer hochgradig verteilten Umgebung von der Nutzung von PKI-Techniken abhängig ist. Auf die Erzeugung der Zertifikate folgt sofort die nächste Überraschung: Mit diesen Zertifikaten werden nicht wie üblich Authentisierungen durchgeführt – das klassische Namenszertifikat bindet ja einen Namen an einen Public Key – sondern diese Zertifikate werden für die Erzeugung weiterer Zertifikate verwendet, eben Proxy-Zertifikate. An ihnen werden wir zeigen, wie mächtig das Konzept der PKI sein kann, wenn es die Schlüssel und nicht etwa die Namen in den Vordergrund stellt. Leider dominiert im Moment das klassische Namenszertifikat das Denken der Entwickler so stark, dass andere Möglichkeiten gar nicht in Betracht gezogen werden, z. B. die Möglichkeit, gewisse Attribute oder auch Autorität an Keys zu binden. Die Möglichkeit einer solchen Bindung besteht ja auch bereits im klassischen Namenszertifikat, aber dort hat es sich als problematisch herausgestellt, Attribute von einer CA signieren zu lassen – was ist, wenn sich das Attribut ändert? In diesem Fall müsste mit hohem Aufwand (erneute Identitätsprüfung) ein neues Zertifikat ausgestellt werden.

Im Falle der Proxy-Zertifikate ist dies gar kein Problem: Der Besitzer eines normalen (sogenanntes *End-Certificate*)-Zertifikats, das von einer CA signiert wurde, stellt nämlich ein Proxy-Zertifikat selbst aus. In diesem werden bestimmte Attribute und Rechte an den Besitz eines Private Key gebunden. Gleichzeitig wird die Gültigkeit des Proxy-Zertifikats, verglichen mit einem End-Certificate, stark eingegrenzt, sodass sich ändernde Attribute oder Rechte keine große Rolle spielen.

Was kann man mit ein Proxy-Zertifikat erreichen? Proxy-Zertifikate sorgen für eine sichere und feingranulare Delegation: Wir wollen jemandem (dem Besitzer

eines Private Key) ein Recht geben, in unserem Namen etwas zu tun. Wenn wir also z. B. einen Service für eine Urlaubsbuchung benutzen, dann kann es sein, dass dieser Service Teilaufträge an andere Services weitergeben will. Für uns besteht daher die Notwendigkeit, das Recht, in unserem Namen einen Auftrag zu vergeben, and den Service zu delegieren. Wie kann der Service diesen weiteren Services (Targets) beweisen, dass er in unserem Auftrag handelt? Die Antwort ist: Er präsentiert ein von uns ausgestelltes Proxy-Zertifikat.

Im Grunde tun wir mit dem Ausstellen eines Proxy-Zertifikats nichts anderes, als eine Vollmacht für den Überbringer auszustellen. Eine solche Vollmacht muss mindestens folgende Anforderungen erfüllen:

- Dem Aussteller erlauben, den Auftraggeber und den Auftrag zu benennen.
- Dem letztendlichen Empfänger (Target) erlauben, die Vollmacht zu verifizieren.
- Dem Target erlauben, den Überbringer zu authentisieren.
- Optional: Dem Aussteller erlauben, die Vollmacht inhaltlich und zeitlich zu begrenzen.

Diese Anforderungen lassen sich durch folgende Vorgehensweise realisieren:

- Der beabsichtigte Träger der Vollmacht erzeugt ein asymmetrisches Schlüssel-paar und schickt den öffentlichen Schlüssel an den Aussteller der Vollmacht.
- Der Aussteller der Vollmacht verwendet sein Endzertifikat, um eine Vollmacht zu signieren. Die Vollmacht enthält den öffentlichen Schlüssel des Trägers, den Namen des Ausstellers sowie Einschränkungen bezüglich Gültigkeit und Umfang der Vollmacht. Diese signierte Vollmacht geht an den Träger.
- Der Träger stellt nun Requests an ein Target im Auftrag des Erstellers der Vollmacht. Er legt dazu dem Target die Vollmacht vor.
- Das Target kann die Echtheit der Vollmacht anhand der Signatur des Ausstellers verifizieren (dazu ist natürlich das Endzertifikat des Ausstellers nötig).
- Das Target authentisiert den Überbringer der Vollmacht als legalen Träger der Vollmacht. Dazu wird auf Basis des Public Keys, der in der Vollmacht als der öffentliche Schlüssel des legalen Trägers genannt ist, ein Challenge-Response-Verfahren mit dem Überbringer durchgeführt. Dabei muss der Überbringer nachweisen, dass er tatsächlich über den zum Public Key gehörigen Private Key verfügt.
- Das Target prüft den Auftrag und die Einschränkungen in der Vollmacht und entscheidet dann über die Auftragsannahme.

Dieses Verfahren erfüllt die oben genannten Anforderungen an eine Vollmacht. Trotzdem bleiben noch einige Fragen offen:

- Was passiert, wenn das Target nun seinerseits einen weiteren Service aufrufen muss, um den Auftrag zu erfüllen?
- Wie können Aufträge und Einschränkungen in interoperabler Weise formuliert werden?

Das erste Problem taucht dann auf, wenn ein Target die Vollmacht weitergeben will – also selbst als Träger der Vollmacht agiert. Das nächste Target kann die Vollmacht anhand des Zertifikats des Ausstellers authentisieren, aber das anschließende Challenge/Response-Verfahren zur Authentifikation des Trägers schlägt fehl, da der in der Vollmacht erwähnte öffentliche Schlüssel dem ursprünglichen Überbringer gehört, aber nicht dem Target, welches jetzt die Vollmacht vorgelegt hat.

Dieses Problem lässt sich dadurch umgehen, dass der Aussteller der Vollmacht in der Vollmacht verfügt, dass der legale Träger weitere Vollmachten (mit Einschränkungen) ausstellen darf. In diesen Zweitvollmachten ist die Ursprungsvollmacht enthalten und somit kann der Empfänger die ganze Kette der Vollmachten prüfen. Dieses System ist sehr leichtgewichtig und flexibel und erlaubt eine sichere Delegation an Dritte. Die Targets können dann je nach verwendetem Protokoll auch den Public Key des Auftraggebers verwenden, um gewisse Daten zu verschlüsseln und so für Intermediates unlesbar zu machen.

Das zweite Problem ist schwieriger zu lösen: Wie können wir Aussagen sowohl interoperabel als auch flexibel gestalten? Hier bietet sich wieder SAML als Möglichkeit an, um solche sicherheitsrelevanten Aussagen zu formulieren.

Die Erlaubnis, weitere Vollmachten auszustellen, hat übrigens einen wichtigen Nebeneffekt: Der Benutzer von Grid Services stellt nämlich als allererstes sich selber ein Proxy-Zertifikat aus, das inhaltlich unbegrenzt ist, aber zeitlich etwa auf acht Stunden begrenzt bleibt. Dieses Proxy-Zertifikat verwendet der Benutzer nun, um eventuelle weitere Vollmachten auszustellen. Das hat den großen Vorteil, dass der hochkritische Private Key, der zum Endzertifikat des Benutzers gehört, nicht die ganze Zeit im Zugriff sein muss, sondern nur einmal am Anfang eines Arbeitstages. Das eigene Proxy-Zertifikat kann dann auch leicht im File-System abgelegt werden – nur geschützt durch die Zugriffsschutzmechanismen des Betriebssystems. Das wiederum erleichtert automatische Services, die nicht durch User-Interaktion gestartet werden: Ein Schadensfall wird auf jeden Fall durch das zeitliche Limit der Vollmacht begrenzt.

Dieses Prinzip ist auch von den Ticket Granting Tickets (TGT) bei Kerberos bekannt. Allerdings ist die Delegation dort durch die bloße Weitergabe des Tickets implementiert. Im Gegensatz zum Proxy-Zertifikat ist eine inhaltliche Einschränkung der Vollmacht nicht möglich.

Zum Abschluss unserer Diskussion des Einsatzes von Proxy-Zertifikaten in Grids noch ein Hinweis auf das Key-Management. Die Schlüssel der beteiligten Parteien besitzen natürlich innerhalb der Grid Security eine sehr große Bedeutung: Sie identifizieren und autorisieren die Benutzer. Der Desktop der Nutzer ist also ein denkbar schlechter Aufbewahrungsort für diese Schlüssel. Aus diesem Grund wurde eine zentrale, auf gesicherten Servern basierende Schlüsselverwaltung für Grids namens *Purse* entwickelt. Die Private Keys verbleiben dabei immer auf dem Purse-Rechner und werden bei Bedarf durch einfachere PINS bzw. Passwörter freigeschaltet. Diese können problemlos in regelmäßigen Abständen geändert werden, ohne dass sich am Private Key selbst etwas ändert.

Für den Entwickler sollte die wichtigste Erkenntnis aus diesem Abschnitt die sein, dass Zertifikate viele andere Zwecke haben können, als nur einen Namen mit einem Key zu verbinden.

6.7 Integration von Conventional-off-the-Shelf (COTS) Applikationen

Zum Abschluss unserer Diskussion der Enterprise Level Security wenden wir uns der Frage zu, welche Kriterien „fertige“ Applikationen erfüllen müssen, damit sie in eine bestehende Infrastruktur integrierbar sind. Dies hängt, wie nicht anders zu erwarten, im Wesentlichen von der bereits bestehenden Infrastruktur ab. Wir unterscheiden dabei drei Ausbaustufen der Authentisierungs-Infrastruktur und nennen die dazu gehörigen Kriterien für die Integration einer COTS-Applikation:

Stufe 1: Keine Single-Sign-On-Infrastruktur

- Die Applikation führt eigene Authentisierung und Authorisierung im Code durch.
- Die Applikation kann ausgelagerte Authentisierungsinformationen (d. h. ein externes Repository, meist LDAP) zur Authentisierung nutzen.
- Die Applikation kann ausgelagerte Authorisierungs-Informationen nutzen bzw. diese Informationen werden in eine interne Datenbank der Applikation importiert.
- Es existieren konfigurierbare Credentials für das Ansprechen anderer Backend-Systeme oder Services.

Stufe 2: Existierende SSO-Infrastruktur

- Die Applikation führt keine eigene Authentisierung mehr durch.
 - Die Applikation ist in der Lage, von einer Trusted Third Party ausgestellte Authentisierungs-Tokens zu verstehen und zu validieren.
 - Die Applikation ist in der Lage, aus dem Authentisierungs-Token eine eigene Repräsentation des Users zu erstellen.
 - Die Applikation legt die Schnittstellen ihrer Authentisierung und Authorisierungs-Module offen und ist damit an verschiedene Verfahren zur Authentisierung und Authorisierung anpassbar.
- Die Applikation ist in der Lage, alle Verbindungen unter Schutz von Integrität und Vertraulichkeit herzustellen bzw. zu akzeptieren (meist via SSL).

In dieser Stufe sind unter Umständen umfangreiche Integrationsarbeiten nötig, die jedoch durch die offenen Interfaces der Applikation erleichtert werden. Eine Weitergabe erhaltener Credentials ist jedoch nicht möglich. Muss die Applikation Backends ansprechen, so geschieht das unter einer funktionalen UserID.

Stufe 3: Komplette Integration in Enterprise-Infrastruktur

Alle unter 2 genannten Punkte, zusätzlich:

- Die Applikation verwendet die Standardrepräsentationen für Principals und Credentials auf der jeweiligen Enterprise-Plattform (z. B. "Subject" in JAVA EE).
- Die Applikation verwendet die Standardservices für Impersonation und Delegation auf der jeweiligen Enterprise-Plattform (z. B. Subject.doAs() in JAVA EE) und kann damit die Caller-Identität für Sicherheitsentscheidungen verwenden und an die Backends die Identität glaubwürdig weitergeben.
- Die Applikation führt Authorisierungen durch die Standard-Interfaces (neuer JSR) durch.
- Die Applikation definiert Rollen und Rechte deskriptiv.

Literatur

- [ACEGI] B. Alex, Acegi Reference Documentation 1.0,
<http://www.acegisecurity.org/docbook/acegi.pdf>
- [SPRINGSEC] Spring Security Reference Documentation 2.0x, 2007, <http://static.springframework.org/spring-security/site/reference/pdf/springsecurity.pdf>
- [Apa] Apache Software Foundation, Struts User Guide,
<http://struts.apache.org/1.x/userGuide/preface.html>
- [BEA] BEA WebLogic, Introduction to WebLogic Security,
<http://e-docs.bea.com/wls/docs81/secintro/index.html>
- [Chak] A. Chakrabati, Grid Computing Security, Springer-Verlag 2007
- [Glob] Globus Toolkit – Core: Developer's Guide,
<http://www.globus.org/toolkit/docs/3.2/core/developer/javaprogramming.html>
- [GSI] Globus Toolkit Version 4 Grid Security Infrastructure: A Standards Perspective,
<http://www.globus.org/toolkit/docs/4.0/security/GT4-GSI-Overview.pdf>
- [heise08_05] Gesucht und gefunden: Cisco-Passwörter,
<http://www.heise.de/security/news/meldung/62491>
- [heise09_08] Gestohlene PwC-Datensätze für Missbrauch von Click&Buy benutzt [Update]
<http://www.heise.de/newsticker/Gestohlene-PwC-Datensaeetze-fuer-Missbrauch-von-Click-Buy-benutzt-Update--/meldung/115621>
- [Her] D. Heringhaus, No More Secrets, Bericht zu den Sicherheitslücken in OBSOC, der T-COM Web-Applikation zur Kundenverwaltung,
<http://www.ccc.de/t-hack/stn/inhlt/drartkl.html>

- [Hoh] G. Hohpe, Event-Driven Architectures, <http://www.eaipatterns.com/docs/EDA.pdf>
- [J2EE] E. Armstrong et al., The J2EE 1.4 Tutorial, <http://java.sun.com/j2ee/1.4/docs/tutorial/doc/>
- [JBoss] The JBoss 4 Application Server Guide, <http://docs.jboss.org/jbossas/jboss4guide/r1/html/index.html>
- [KNNP] L. Koved, A. Nadalin, N. Nagarathnam, M. Pistoia, Enterprise Java Security: Building Secure J2EE Applications, Addison-Wesley 2004
- [McRee] R. McRee, SELinux, Apache, and Tomcat – A Securely Implemented Web Application Server, http://www.owasp.org/images/0/01/Secure_Web_App_Server_McRee_OWASP.pdf
- [MFP] G. Mühl, L. Fiege, P. Pietzuch, Distributed Event-Based Systems, Springer Verlag 2006
- [Mil] M. S. Miller, Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control, Dissertation, Johns Hopkins University, Baltimore, available online at <http://www.erights.org/talks/thesis/index.html>
- [Moo] D. Moore, Use JAAS for Struts, <http://www.mooreds.com/jaas.html>
- [OGF] Open Grid Forum, The Open Grid Services Architecture, Version 1.0, <http://www.ggf.org/documents/GFD.30.pdf>
- [Rann] K. Rannenbergh, A. Pfitzmann, G. Müller, Sicherheit – insbesondere mehrseitige IT-Sicherheit, <http://www.wiiw.de/publikationen/Sicherheitinsbesonderemehrsei.pdf>
- [Reith] L. Reith, Concept of data distribution for worldwide distributed services in service-oriented architectures, Diplomarbeit an der HdM 2007
- [RFC3820] S. Tuecke et al., Internet X.509 Public Key Infrastructure (PKI) – Proxy Certificate Profile, <http://www.ietf.org/rfc/rfc3820.txt>
- [Wel] V. Welch, Grid Security Infrastructure Message Specification, <http://www.ggf.org/documents/GFD.78.pdf>
- [WFK] V. Welch, I. Foster, C. Kesselman et al., X.509 Proxy Certificates for Dynamic Delegation, <http://www.globus.org/alliance/publications/papers/pki04-welch-proxy-cert-final.pdf>

Kapitel 7

Application Server Security

Moderne Applikationsserver bieten eine Fülle von sicherheitsrelevanten Bibliotheken, APIs und konfigurierbaren Funktionen an. Sie spielen auch eine zentrale Rolle in Enterprise-Infrastrukturen. Ohne ein Verständnis ihrer Funktionen ist heute keine Applikationsentwicklung mehr möglich.

Applikationsserver sind ursprünglich angetreten, das Leben von Applikationsentwicklern zu erleichtern – ein frommer Wunsch, an den so mancher Entwickler angesichts der Dokumentation zu JBOSS, Websphere oder Weblogic nicht mehr so recht glauben kann. „Das Leben erleichtern“ bedeutet in der Softwareentwicklung meist, etwas „transparent“ zu machen, das heißt unsichtbar für den jeweiligen Verwender. Das bedeutet, dass bestimmte Funktionen nicht mehr programmiert werden müssen, sondern bereits in Bibliotheken zur Verfügung stehen. Aber darüber hinaus wird auch der Aufruf der Funktion, also die Entscheidung zu ihrem Einsatz, aus dem Code der Applikation in einen Konfigurationsbereich verlagert. Die spezielle, transparente Eigenschaft der Applikation wird beschreibend (deskriptiv) festgelegt und kann daher auch nachträglich – der Situation in einer Firma entsprechend – geändert und angepasst werden. Damit haben wir die beiden entscheidenden Funktionen eines modernen Application-Servers bereits genannt: „Separation of Concerns“ (die Applikation muss sich um bestimmte Funktionen nicht mehr kümmern) und „Separation of Context“ (die Applikation kann je nach Umgebung angepasst werden, ohne den Code ändern zu müssen).

Wir wollen uns in diesem Kapitel in erster Linie mit den Funktionen, die ein Application-Server den jeweiligen Applikationen anbietet, beschäftigen. Unter die von uns behandelten Sicherheitsaspekte von Application-Servern fallen die Identität von Aufrufern, deren Rechte und die Organisation der Rechte in Form von Rollen, Vertrauen zwischen Komponenten, die Sicherheit der übertragenen Daten und nicht zuletzt die Frage, welche Prüfungen wo stattzufinden haben. Die eben genannte Transparenz der technischen Lösung – sie muss nicht mehr selbst programmiert werden – ist nämlich aus Security-Sicht keineswegs unproblematisch: Bei falscher Bedienung können enorme Lücken entstehen. Der zweite wichtige Punkt ist die Rolle des Application-Servers innerhalb einer Enterprise-Architektur.

Mit welchen weiteren Komponenten und Services muss er kommunizieren? Wie können neue Dienste integriert werden? Wie kann er an eine Firma angepasst werden? Die Sicherheit des Applikationsservers selbst werden wir dagegen im Abschn. 7.6 untersuchen.

Eine detaillierte Darstellung aller Interfaces und Mechanismen bezüglich Security in einem Application-Server ist dabei keineswegs unser Ziel. Stattdessen wollen wir ein Verständnis für das Zusammenspiel der Techniken wecken und einzelne, besonders kritische und schwierige Bereiche wie Delegation besonders herausgreifen. Deshalb erscheint es uns richtig, zuerst die Entwicklungsphasen zum Application-Server-Konzept zu beschreiben, da sich daraus jeweils neue Sicherheitsprobleme ergeben haben. So ist heute z. B. der Thread zur Laufzeit der Träger der Sicherheitsinformationen eines Calls (der sogenannte Security Context) und nicht mehr wie früher der Prozess.

7.1 Vom Socket-Server zur Container-Architektur

Ganz am Anfang aller Server stand die Abtrennung des Servercodes vom Kern des Betriebssystems und des Serverprozesses vom Kernel-Prozessraum. Der ursprüngliche Server ist ein im User-Mode laufender Prozess – funktional vielleicht Teil des Betriebssystems, aber nicht in dessen privilegiertem Modus laufend, sondern unter einer User-Identität. Die Vorteile der Trennung waren eine größere Systemstabilität – der Server als Prozess konnte abstürzen, ohne dass das Betriebssystem mit hinuntergerissen wurde – und kürzere Entwicklungszeiten. Beispielhaft dafür war die Entwicklung von XWindows als verteilter Netzwerkgrafik, deren Server komplett außerhalb des Kernels lief (eine Architekturentscheidung, die auch heute noch in einem krassen Gegensatz zur Microsoft-Philosophie der maximalen Integration von Services in den Betriebssystemkern steht).

Sicherheitstechnisch bedeutete dies zwar eine Entmachtung des Servers, aber eine Verbesserung der Systemsicherheit, da er für die Benutzung wichtiger Dienste auf das geschützte System-Call-Interface angewiesen war und dies nur im Rahmen seiner Rechte benutzen konnte.

Dank des Socket-Designs als simpler Endpunkt eines Kommunikationskanals konnte ein Server sehr einfach Daten oder Kommandos empfangen und senden. Auffallend ist, dass der Empfangsmechanismus nicht fest an eine Authentisierung mit nachfolgender Authorisierung gebunden war, wie das sonst bei der Nutzung von Betriebssystemressourcen vom System selber erzwungen wurde: Das Prinzip „Keine Shell ohne vorherigen Log-in“ galt hier nicht. Stattdessen implementierten diese Server eine eigene Authentisierung oder authentisierten Clients gegenüber der Registry des Betriebssystems. Damit war grundsätzlich die Möglichkeit verschiedener Rechtsbereiche innerhalb eines Hosts gegeben. Implementierte der Server seine eigenen Prüfungen, konnte ihn das Betriebssystem dabei nicht unterstützen. Vergaß er die Prüfungen, konnte das System ebenfalls nichts dagegen unternehmen: Der Server hatte ja seine eigenen Rechte durch eine eigene Identität.

Bald standen weitere Trennungen an: Die Server mussten ihre aktuelle Adresse, unter der Clients sie erreichen konnten, nicht mehr selber verwalten. Dies übernahm beispielsweise ein Portmapper-Dienst. Server wurden in der Folge auch automatisch gestartet, falls entsprechende Requests ankamen, dafür sorgte der INET-Daemon. Dennoch stand am Beginn der Entwicklung von Internetservern eine Softwarearchitektur, die Serverfunktionalität (Annehmen von Requests, Protokollierung, Authentisierung und Zugriffskontrolle) und die jeweilige Businesslogik in sich vereinte. Klassische Beispiele hierfür sind FTP-Server, Mail-Server, usw. Die drei wichtigsten Trennungen standen jedoch noch bevor, und sie spielten (und spielen) vor allem im Bereich der Web-Server und Applikationsserver eine große Rolle.

7.1.1 *Authentisierungs-Frameworks*

Die erste Trennung bestand in der Abtrennung der Authentisierung von der Businesslogik und letztlich vom gesamten Server durch die Einführung eines Authentisierungs-Frameworks, welches Pluggable Authentication Modules (PAM) genannt wurde. Was war der Grund für diese Trennung? Er lag schlicht in der permanenten Änderung bzw. Erweiterung der möglichen Methoden der Authentisierung, die die Server-Applikationen zur dauernden Anpassung zwangen: Passwörter, biometrische Verfahren, Kerberos-Tickets, Zertifikate etc. erfordern alle unterschiedliche Formen der Authentisierung – und zwar in jedem Server!

Allein die Tatsache, dass die Authentisierung eine Funktionalität ist, die jeder Server braucht, lässt es sicherheitstechnisch ratsam erscheinen, dass dies von einem – gut getesteten – Framework erledigt wird und nicht individuell pro Applikation von Applikationsprogrammierern erledigt werden muss. Dies beruht auf dem alten Prinzip der Softwareentwicklung, dass Applikationsprogrammierer keinen Systemcode erstellen sollten, da die Fehlerquote dabei zu hoch ist. Hier wird das Architektur-Pattern des „Separation of Concerns“ durch die Auslagerung von Sicherheitsfunktionalität in Bibliotheken erreicht. Die Applikationsserver sollten dieses Prinzip später noch wesentlich erweitern und selbst die Aufrufe der Bibliotheken den Applikationen wegnehmen.

Im J2SE- wie im JAVA-EE-Umfeld wird diese Separation of Concerns durch das JAAS-Framework erreicht – eine javabasierte Implementation der unter Unix als PAM bekannten modularen Authentisierung. Wir werden ihre Grundkonzepte weiter unten noch behandeln. Anzumerken ist noch, dass JAAS eine späte Entdeckung im Java-Umfeld darstellte, da sich das ursprüngliche Java-Konzept auf Umgebungen beschränkte, die nur einen Nutzer kannten (z.B. Desktop oder Browser). In Serverumgebungen konnte das natürlich nicht ausreichen, da hier für jeden User ein sogenannter *Security Context* nötig war – also schlicht Informationen über den jeweiligen Request und den, der ihn gestellt hatte.

7.1.2 Concurrency und Multithreading

Die zweite Trennung wurde gerade auch im Web-Umfeld sehr populär: Sie ersetzte das Prinzip des „ein Serverprozess – eine Applikation“ durch das Prinzip „ein Serverprozess – mehrere Applikationen“ und bedeutete den Übergang vom Prozessmodell zum multithreaded Server, der gleichzeitig mehrere Applikationslogiken ausführen kann, und dies natürlich auch für unterschiedliche User. Der Grund lag in der Systemlast, die ein Prozess pro Server-Applikation mit sich brachte, sobald mehrere gleichzeitige Requests anlagen: Bis zu diesem Zeitpunkt stellte der Prozess die entscheidende Strukturierung für komplexe Abläufe dar, isoliert von anderen Aufgaben durch die Prozessgrenzen. Jetzt rückte der Thread als Repräsentant eines Requests in den Vordergrund. An ihm anstatt am Prozess musste jetzt der Security Context eines Requests – also alle Informationen, die für die sichere Abarbeitung des Requests nötig sind, wie zum Beispiel Informationen über den User (Principal, Credentials) – festgemacht werden. Darin (neben Transaktionskontexten) liegt auch der Grund, dass viele Application-Server heutzutage der Applikation selbst das Anlegen eigener Threads verbieten.

Gleichzeitig hat sich damit – zumindest am Anfang – eine sehr schwierige Form der Concurrency durchgesetzt: Das „Shared-State“-Konzept beim Multithreading, das ausgefeilte Locks und Semaphorlogik benötigt, um geteilte Daten vor der Korruption durch mehrere gleichzeitig laufende Threads zu schützen. Die Auswirkungen dieser Form der Concurrency auf die Sicherheit von Servern sind subtil. Sie besteht unter anderem in der Möglichkeit sogenannter Time-Of-Check-To-Time-Of-Use (ToC2ToU)-Fehlern: Wenn ein Request erst geprüft und dann verwendet wird, kann es passieren, dass in der Zwischenzeit ein anderer Thread die Prüfbedingungen invalidiert hat. In Kap. 10 wird speziell im Abschn. 10.4 zu Concurrency auf die damit verbundenen Gefahren eingegangen werden.

Wichtig bei dieser Trennung ist weiterhin, dass zwischen den Applikationen keine Prozessgrenze und damit auch keine Speichersicherheit mehr besteht – es sei denn, es werden innerhalb des Application-Servers besondere Maßnahmen getroffen, sei es durch Sprachkonstrukte wie typsichere Sprachen, Systemkonstrukte wie isolierende Classloader oder andere Maßnahmen aus dem Bereich des Virtual Memory Management.

Außerdem musste jetzt zum ersten Mal ein Interface zwischen der Serverbasis und den jeweiligen Applikationen definiert werden, also ein Plug-in-Konzept, damit die Applikationen dynamisch ladbar wurden. Diese Art von Interfaces – auch „Verträge“ genannt – sind für heutige Applikationsserver typisch.

7.1.3 Komponentenarchitektur

Die dritte und wahrscheinlich wichtigste Trennung bestand darin, dass die Funktionalität des „Serving“, also alles, was aus rein technischer Sicht zu einem Server

gehört, jedoch nicht ursprünglich applikationsspezifisch ist (zum Beispiel Logging oder Security), wirklich als separate Anforderung begriffen wurde. Damit verloren Applikationen ihren Charakter als unabhängige Einheiten, die selbständig ausgeführt werden konnten, wie das klassische *.exe-Programm, und verwandelten sich in Code, der zu seiner Ausführung die Laufzeitumgebung eines Application-Servers benötigte, welcher dann die nötigen Dienste zur Verfügung stellt.

Ursprünglich mag dabei der Gedanke der Performanceverbesserung durch einige Prozesse, die viele Server-Applikationen laufen lassen können, möglicherweise sogar dynamisch nach Bedarf den Speicher füllen oder freigeben können, im Vordergrund gestanden haben. Des Weiteren vereinfacht sich natürlich der Applikationscode, wenn wichtige Dienste über Standard-Interfaces aufgerufen werden können. Bald aber trat noch ein anderer Gedanke hinzu: Die Applikation wird zu einer Komponente, die unabhängig von der Umgebung ist und deshalb in vielen verschiedenen Umgebungen eingesetzt werden kann. Das von der Sprache Java her bekannte Prinzip des „Code Once – Run Everywhere“ hatte sich nämlich sehr schnell als Illusion herausgestellt: Einzelne Programme haben viel zu viele Abhängigkeiten in Bezug auf ihre Umgebung, als dass sie überall laufen könnten. Dies hat mit der verwendeten Sprache nichts zu tun, sondern damit, dass viele Services umweltabhängige Teile beinhalten, wie etwa verschiedene Datenbanken, verschiedene Rollennamen, verschiedene Repositories für Benutzer oder Kunden sowie verschiedene Protokollierungsdienste. Gerade auch die Sicherheitsdefinitionen und Mechanismen variierten stark von Firma zu Firma und erschwerten die Portabilität von Applikationen enorm.

Damit war der Weg frei für eine Komponentenarchitektur auf Basis von sog. *Containern* – also Ausführungsumgebungen für Applikationen, wie es früher das Betriebssystem selbst war. Zum Beispiel separieren Enterprise Java Beans die Funktionalitäten Security, Transactions und Persistence von den Applikationen und stellen sie über den EJB-Container bereit. Da aber über die drei genannten Abhängigkeiten hinaus noch weitere bestehen, entstand die JAVA-EE-Architektur, die eine weitere Separation of Concerns einführte, nämlich das Finden von Ressourcen über Datasources, Java Naming und Directory Interfaces sowie die deklarative Definition von Sicherheitswünschen von Seiten der Applikationen. Daneben wurde auch die „Separation of Context“ eingeführt, also das Auslagern von spezifischen Informationen in deklarativer Form, die dann anschließend während der Installation ohne Source-Code-Änderungen an die jeweilige Umgebung angepasst werden können. Wir haben dazu deklarative Deployment-Deskriptoren aus dem Rollenbereich und der Delegation von Requests im Kap. 6 gesehen.

Nach dieser dritten Umwälzung war die Welt für die Applikationen scheinbar in Ordnung. Die sicherheitsrelevanten Interfaces, die für sie zur Verfügung stehen, werden weiter unten in diesem Kapitel im Abschnitt zu Application-Interfaces genauer beschrieben. Bei genauer Betrachtung jedoch stellt man fest, dass man das Problem der Abhängigkeit von Umgebungen jetzt nur an eine andere Stelle verschoben hat, nämlich auf den Application-Server bzw. dessen Hersteller. Um nämlich zum Beispiel die Applikation von der Authentisierung zu entlasten, muss der Application-Server sie selber durchführen, und dies bedeutet je nach Umge-

bung, dass die User-Daten in einer Datenbank, einem File, einem LDAP-Verzeichnis oder in einem Custom Repository vorliegen können – in verschiedensten Strukturen. Somit hat jetzt der Application-Server das gleiche Problem wie vorher die Applikation: Wie soll er sich gegen verschiedenste Umgebungen isolieren? Zur Lösung dieses Problems wird das gleiche Mittel wie im Falle der Applikation eingesetzt: Eine Serie von Interfaces, die den Server von der Umgebung isolieren. Meist geschieht das über die Definition von Interfaces zu sogenannten *Providern* oder *Ressource-Managern*. Um beispielsweise zu verhindern, dass jeder Application-Server sich an unzählige Datenbanken anpassen muss, wurde ein standardisiertes Interface zu einem Ressource-Manager spezifiziert – inklusive verschiedener Möglichkeiten, sich gegenüber solchen zu authentisieren – und das Ganze wurde dann die Java-Containerarchitektur: Ein Vertrag zwischen Application-Server/Container auf der einen Seite und Providern von externen Services auf der anderen. Das sich daraus ergebende Bild ist in Abb. 7.1 dargestellt.

Abbildung 7.1 (sowie die folgenden zur Application-Server-Architektur) orientiert sich grob an der Architektur des Websphere-Application-Server (WAS) in der Version 6.1, wie sie von Carla Sadtler et al. in [SAFK] beschrieben wurde. Es zeigt den heutigen Application-Server als Basis für Plug-ins zur Anbindung von externen Services. Ebenfalls deutlich wird die zentrale Rolle des CSiv2-Protokolls zur Weitergabe von Authentisierungs- und Autorisierungsinformation zwischen verschiedenen Application-Servern.

Leider wurde gerade im Bereich der Security hier nur sehr spät oder gar nicht standardisiert, sodass heutige Application-Server immer noch zu speziellen Lö-

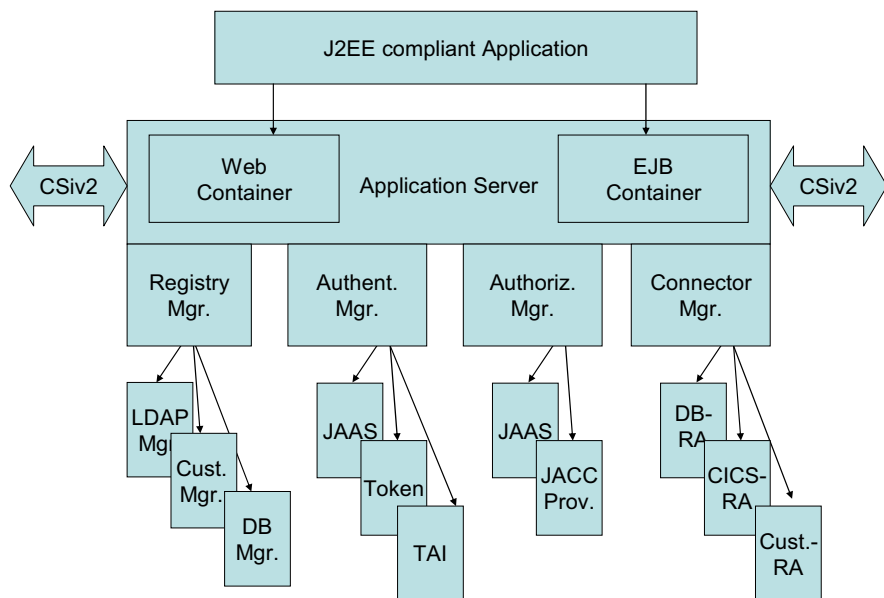


Abb. 7.1 Komponentenarchitektur eines Application-Servers mit Ressource-Managern

sungen gezwungen sind. Wir werden weiter unten ein Beispiel betrachten, wie sich auf Basis von Frameworks und Plug-in-Technologie Application-Server anpassbar machen.

7.1.4 Vom Standalone-Server zum Teil eines verteilten Systems

Eine letzte Trennung bleibt noch nachzutragen: Die Trennung von der Idee des Application-Servers als für sich allein stehende Plattform zur Ausführung von Applikationen und dem Anbieten von Services. Heutzutage ist der Application-Server Teil eines verteilten Systems, das mitunter die Firmengrenzen deutlich überschreitet und externe Partner über das Internet mit einbezieht. Dazu war es nötig, sich vom Konzept des Standalone-Servers, der zum Beispiel eine initiale Authentisierung selbst durchführt, zu verabschieden. Weder Web-Container noch EJB-Container werden in Zukunft initiale Authentisierungen durchführen müssen. Sie erhalten vielmehr von Single-Sign-On-Servern (sogenannte *Security Token Issuer*) bereits authentifizierte User. Nur deren Berechtigungen sind noch durch den Server zu prüfen. Das Stichwort hier heißt *Identity Assertion* und meint die Behauptung eines Client, im Auftrag einer anderen Identität mit deren Rechten zu agieren. Diese Behauptung kann durch geeignete Token gestützt werden.

Genauso wichtig ist es für Application-Server in Zukunft, diese Token und deren Informationen mit den Requests weiter fließen lassen zu können, nämlich an andere Application-Server innerhalb und außerhalb der Firma. Dazu müssen die Application-Server Methoden wie Web-Services einsetzen können und auf die Übertragung von Sicherheitskontexten mit eingebetteten Sicherheitsinformationen vorbereitet sein. Wir werden uns die heutigen Möglichkeiten und Grenzen bei der Weitergabe von Token ansehen und selbst ein Beispiel einer externen Weiterleitung unter Verwendung von Web Services Security durchspielen.

7.1.5 Server-Cluster

Was hat die physische Architektur einer Applikationslösung oder einer Firmeninfrastruktur mit Security zu tun? Wenn der Server-Cluster für transparentes Failover sorgen soll, das heißt, dass Requests von Clients im Falle eines Problems auf andere Rechner umgeleitet werden, dann müssen diese entweder die Clients neu authentisieren oder die bestehenden Subjekte weiternutzen oder sie neu erzeugen. Für die neuerliche Erzeugung der Subjects aus Sicherheits-Tokens benötigen sie aber die Kenntnis der Schlüssel, die für Signaturen oder Verschlüsselung verwendet wurden. Somit bilden Server-Cluster oder Teile von Clustern auch Vertrauenszonen. Das gleiche gilt für Credentials, die in einem gemeinsamen Cache gehalten werden und von verschiedenen Servern gemeinsam genutzt werden.

Weiter unten werden wir Beispiele für die Propagation von authentisierten Subjekten und ihren Rechten kennen lernen – ein Mechanismus, der im Zusammenhang mit Clustering von Application-Servern und Caching nicht ganz ohne Probleme ist.

7.1.6 Anforderungen an die Entwickler

Schließen wir diese Einführung mit den Anforderungen an einen Entwickler, der auf Basis der bislang angesprochenen Technologien eine Sicherheitslösung zu entwickeln hat: Das Wissen, wie man in Java oder C# eine kryptografische Bibliothek benutzt, oder wie man ein Rollenkonzept implementiert, reicht heute nicht mehr aus: Hinzu muss das Wissen darüber kommen, wo man diese Konzepte bereits implementiert in den Application-Servern findet und das Wissen über deren Verwendung muss vorhanden sein. Das bedeutet: Wenig Code, aber viel Abstraktion und Wissen um die Laufzeitplattform.

Es zeigt sich, dass die Application-Server dem Entwickler zwar sehr viel Codierungsarbeit abnehmen können, dass die Schwierigkeiten für die Entwickler sich aber jetzt verlagert haben. Teilweise bestehen die Probleme jetzt genau darin, eine Sicherheitslösung in ein Konzept eines Application-Servers einzupassen und dazu seine Mittel und Architektur zu verwenden.

Somit erscheinen Application-Server mit ihren Containern und Providern heute als hochkomplexe Softwarearchitekturen. Die Entwicklung dorthin lässt sich sehr gut an Source-Code-Beispielen nachvollziehen, etwa David Flanagans Beispielen aus dem .NET und RMI-Package ([Flan]). Dort findet man einfache, socket-basierte Beispiele für Client und Server. So findet sich im RMI-Package bereits eine Banking-Applikation mit ladbaren Applikationsmodulen und einer Multithreaded-Server-Plattform. Man kann sich hier gut vorstellen, wie sich diese Plattform durch weitere Funktionalitäten wie Security, Persistenz, etc. erweitern lässt.

Als nächsten Schritt könnte ein Entwickler aus Ted Newards „Server Side Java Programming“ ([New]) lernen, wie ein moderner Application-Server entwickelt wird, insbesondere was Isolierung der Applikationen voneinander durch selbst geschriebene Classloader bedeutet. Schließlich steht mit JBOSS ein professioneller Application-Server zur Verfügung, an dem sich beispielsweise JNDI studieren lässt.

7.2 Beispielarchitektur eines Applikationsservers

Ziel dieses Kapitels ist es, die Architektur von Application-Servern unter dem Blickwinkel potenzieller Sicherheitsprobleme zu beleuchten. Am Anfang unserer Analyse stehen die Interaktionen mit der Umwelt, aus denen Angriffsmöglichkeiten, aber auch Ansprüche an die Zuverlässigkeit abgeleitet werden können.

7.2.1 System Context and Security Context

Das System-Context-Diagramm (Abb. 7.2) für unseren beispielhaften Application-Server zeigt eine hohe Zahl von Kommunikationspartnern, teilweise mehrere Stufen tief. Diverse Services und Protokolle sind dabei im Einsatz. Zweck des Diagramms ist es, sicherzustellen, dass keine Verbindungen bzw. potenzielle Abhängigkeiten vergessen werden. Deshalb ist auch die Ausführungsplattform – sprich: das Betriebssystem – modelliert.

Aus Abb. 7.2 ergeben sich weitere Fragen in Bezug auf die Art der Authentisierung zwischen den Kommunikationspartnern (keine Authentisierung, nur serverseitig, oder gegenseitig?), die Kommunikationskanäle (ungeschützt, integritätsgeschützt, vertraulich?), und die Art der Protokolle (Standardprotokoll oder proprietär?). Für jeden Partner werden Funktion und Verbindungsrichtung der Kommunikation (d.h.: wer baut sie auf?) festgehalten. Auch Einschränkungen bezüglich der transportierten Datenmengen sollten festgehalten werden (Sessiongröße).

Die Zusammenfassung der Antworten auf diese Fragen mündet in das Security-Context-Diagramm. Dieses Diagramm sowie weitere assoziierte Tabellen klären für jede Protokollart die Form der Authentisierung sowie die Sicherheit des Transportes. Darüber hinaus beinhaltet das Security-Context-Diagramm die Topologie als zusätzliches Sicherheitselement: Firewalls trennen Kommunikationspartner und erzwingen Protokollwechsel. Die Komponente „Application-Server“ selbst

System Context

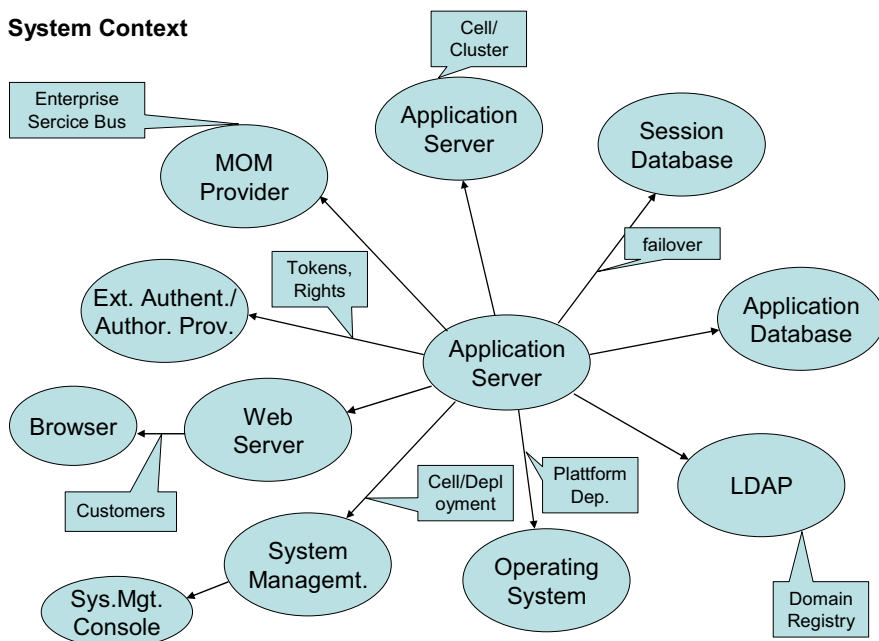
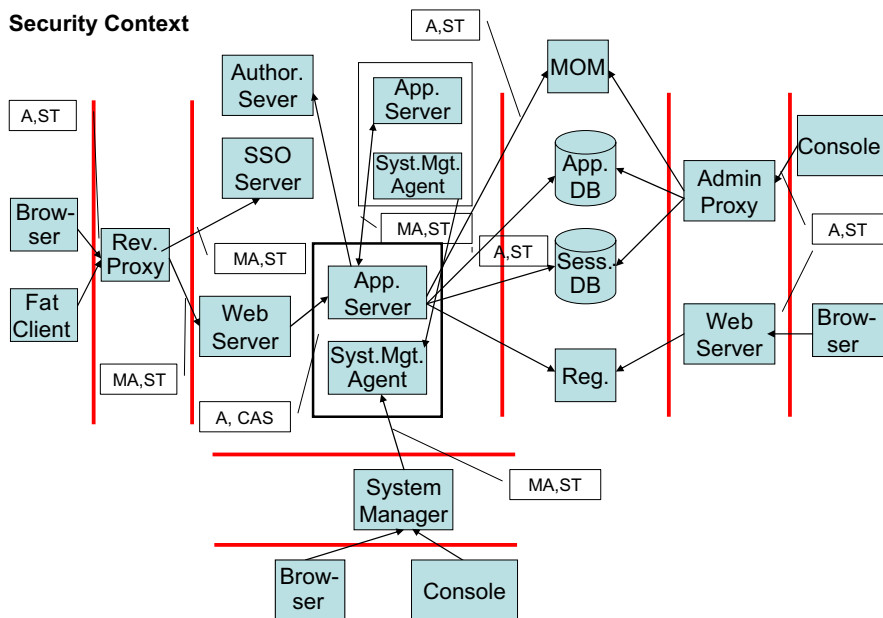


Abb. 7.2 Möglicher System-Context eines Application-Servers

Security Context**Abb. 7.3** Security Context des Beispiel-Application-Servers¹

wird ebenfalls aufgebrochen und besteht nun aus dem Server selbst sowie einem Verwaltungsagenten. Die Verwendung von Code Access Security ist ebenfalls vermerkt (s. Abb. 7.3). Auch Identität, Delegation und Impersonifikation können eingetragen werden.

Das Security-Context-Diagramm für einen Application-Server zeigt auch besonders kritische Verbindungen, wie z. B. die zwischen Reverse Proxy und SSO-Server oder zum Systemmanagement. Diese werden einerseits durch besonders sichere Transportprotokolle und andererseits durch Einbettung in eine DMZ vor Angriffen aus dem Intranet gesichert.

7.2.2 Verwaltung von Credentials

Eine zwangsläufige Folge der hohen Zahl von Kommunikationspartnern ist eine große Zahl an Credentials, die verwaltet werden müssen. Außerdem muss ein Application-Server eine große Zahl verschiedener Authentisierungsmöglichkeiten beherrschen. Das Credential-Diagramm (s. Abb. 7.4) hält die Art und Position der beteiligten Credentials fest. Relevant für Application-Server sind dabei vor allem die folgenden Arten von Credentials:

¹ MA = Mutual Authentication, A = Serverside Authentication, ST = Secure Transport, CAS = Code Access Security

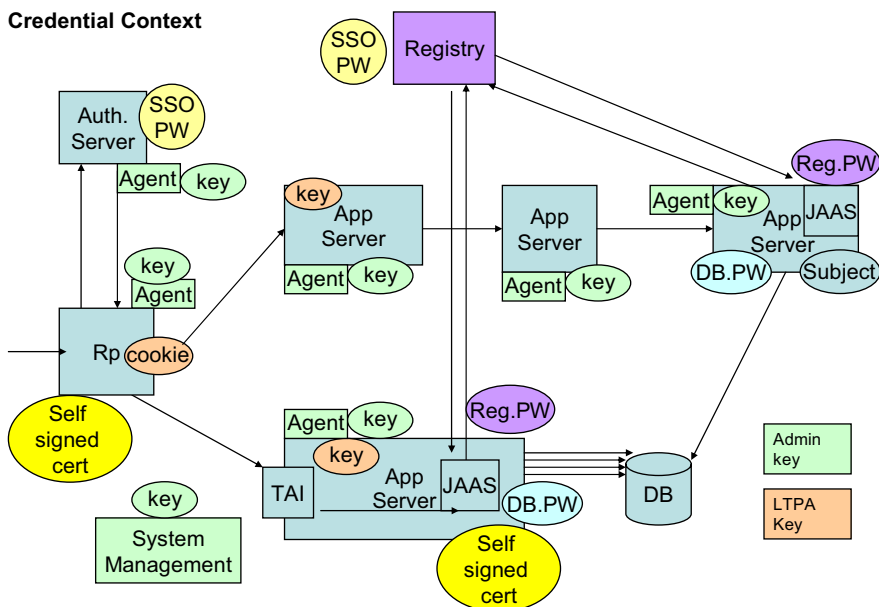
Credential Context

Abb. 7.4 Credential-Diagramm mit Art und relevantem Netzknoten pro Credential

- Datenbank-Credentials,
- Registry-Credentials (LDAP etc.),
- Credentials vom Single-Sign-On-Server zur Validierung von Identity Assertions,
- Credentials für das Systemmanagement (z. B. Admin-Passwort),
- Credentials zur Erstellung sicherer Cookies, die ein authentifizierte Subject repräsentieren und
- Zertifikate für alle beteiligten Services zur gegenseitigen Authentisierung.

Für jedes Credential sollten darüber hinaus die folgenden Informationen in assoziierten Tabellen festgehalten werden:

- Symmetrisches oder asymmetrisches Credential,
- Position (Speicherort und evtl. Transportwege),
- Absicherung und
- Einsatzzweck.

Welche Security-Implicationen ergeben sich nun aus dem obigen Credential-Diagramm des Application-Servers? Zunächst fällt auf, dass das Administrator Credential (der Agent Key), das zu beliebigen Modifikationen des Servers wie etwa Installation von Applikationen oder dem Zugriff auf Ressourcen berechtigt, auf allen Knoten, die einen Management Agent laufen lassen, vorhanden ist. Dies bedeutet im Klartext, dass die Kompromittierung dieses Credentials zum Verlust einer ganzen Zelle von verbundenen Application-Servern führen kann.

Der LTPA-Key, der die Identität von Subjects zwischen Application-Servern durch verschlüsselte Cookies propagiert, erlaubt die Erzeugung neuer Identitäten,

wenn er missbraucht wird. Das Passwort des SSO-Servers, das die Ausstellung neuer Identity Assertions erlauben würde, fließt zu allen Front-End-Application-Servern, und ist dort sowie auf dem Transportweg besonders zu schützen.

Ergänzt werden diese Credentials durch Passwörter für den Zugriff auf Registries und Datenbanken, wobei die Kontrolle hier dem Container oder der Applikation unterliegen kann.

Die selbst signierten Zertifikate erlauben eine gegenseitige Authentisierung aller Kommunikationspartner. Hier ist darauf zu achten, dass mit der Authentisierung nicht automatisch eine Autorisierung verbunden ist.

Eine weitere Folgerung aus dem Credential-Context-Diagramm sollte ein Plan zur regelmäßigen Erneuerung der Credentials im Rahmen von Passwort-Aging-Policies bzw. der Erneuerung von Zertifikaten sein.

7.2.3 Formen der Authentisierung

Wesentliches Element der Sicherheitsarchitektur eines Application-Servers ist die Authentisierung der Kommunikationspartner. Während ursprünglich der Schwerpunkt auf der initialen Authentisierung von z. B. Kunden über das Internet lag – der Application-Server also als Standalone-Server – ist mittlerweile die Übernahme von Autorisierungsaussagen (Identity Assertion) von SSO-Servern oder anderen Application-Servern im Zuge der Delegation von Requests die dominierende Anforderung.

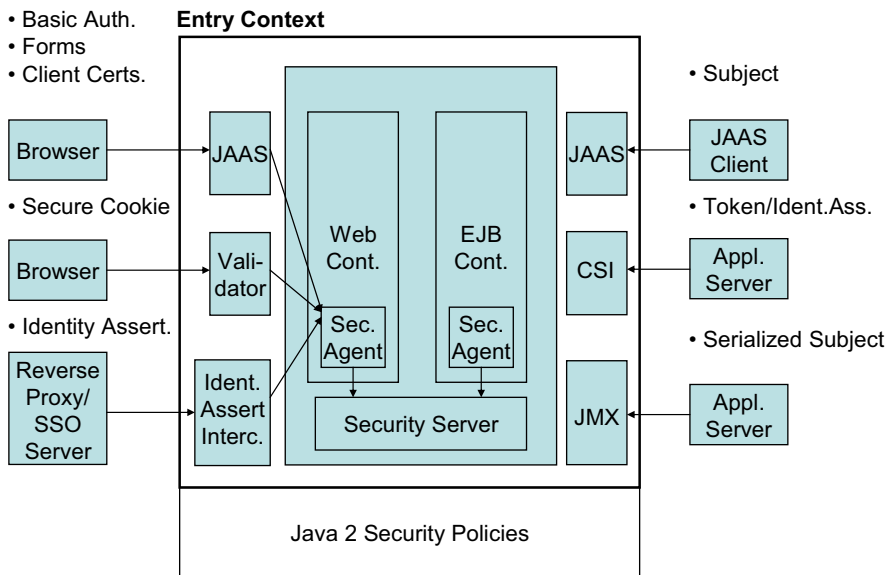


Abb. 7.5 Entry-Context-Diagramm

Das *Entry-Context-Diagramm* (Abb. 7.5) zeigt die verschiedenen Formen der Authentisierung und ihre Abbildung auf die Architektur des Servers. Dies schließt die direkte Propagation von authentisierten Subjects inklusive ihrer Autorisierungs-Credentials ein. Sie wird vor allem unter eng gekoppelten Application-Servern innerhalb von Clustern verwendet.

Das Entry-Context-Diagramm erwähnt nicht zufällig die Java 2 Security: Code-Komponenten wie die Validatoren von Identity Tokens sind extrem kritisch für die Sicherheit einer ganzen Zelle (diese Token werden dann ja zu weiteren Servern propagiert), und sollten daher auch durch Code Access Security abgesichert werden. Das bedeutet z. B., dass eben nur die unbedingt nötigen Komponenten das Recht zur Erstellung eines Subjects besitzen.

Wer sich entschließt, Java 2 Security konsequent im Application-Server einzusetzen, der sollte unbedingt auch ein Policy-Context-Diagramm erstellen, um die Abhängigkeiten zwischen den verschiedenen Policy Files und den Komponenten einer Zelle zu modellieren. Das Vorgehen schildert [New].

7.2.4 *Der Classloader*

Als letztes sicherheitskritisches Element der Architektur soll die Verwendung des Classloader-Mechanismus dargestellt werden. Der Mechanismus ist deshalb so wichtig, weil er ein chronisches Problem der Application-Server, nämlich die schlechte Isolation der Applikationen voneinander und vom Server selbst, lösen kann.

In Kap. 4 zur Plattformsicherheit wurde bereits gezeigt, dass innerhalb eines Prozesskontextes unterschiedlicher Code für verschiedene User ausgeführt wird. Die Isolation von Applikationen auf einem Application-Server untereinander ist natürlich generell durch die Verwendung einer eigenen Virtual Machine zu verbessern, und zwar am besten durch Trennung in unterschiedliche Installationen des Application-Servers, damit keine Abhängigkeiten über Konfigurationsfiles (z. B. Policy-Bestimmungen) zwischen Applikationen entstehen. Dieses Vorgehen wird auch von Ted Neward (s. [New]) empfohlen. Allerdings gibt es noch weitergehende Abhängigkeiten, die später in Abschn. 7.6 (Application-Server Hardening) besprochen werden. Außerdem spricht der gestiegene Aufwand im Systemmanagement dagegen, jeden Application-Server als Insel zu betreiben. Hier muss jede Firma zwischen der Notwendigkeit von Isolation und dem dazugehörigen Verwaltungsaufwand selbst abwägen.

Im Kap. 5 (Java Sprach- und Plattformsicherheit) wurde bereits erwähnt, dass der Classloader in Java unter anderem die Identität einer Klasse bestimmt. Das bedeutet, zwei verschiedene Versionen einer Klasse können über zwei Classloader geladen werden. Classloader können aber auch generell den Zugriff auf Klassen überhaupt erst ermöglichen oder unterdrücken. Durch sie kann letztlich kontrolliert werden, welche Klassen einer Applikation überhaupt zur Verfügung stehen.

Dahinter verbirgt sich ein Isolationsmechanismus, der auf Namespaces beruht und überdies auf Typebene arbeitet und nicht wie z. B. Object-Capabilities auf

Instanzebene. Classloader können unterschiedliche Implementationen von Klassen an bestimmte Applikationen verteilen und dadurch unterschiedliche Policies implementieren. Der Mechanismus selber, d. h., wer welche Typen von Implementationen erhält, beruht auf Konfigurationsinformationen, also Policies. In diesem Punkt ähnelt der Mechanismus der Access-Control-Matrix. Im Unterschied dazu werden die Ergebnisse der Sicherheits-Policy jedoch Teil der Softwarearchitektur zur Laufzeit und können danach nur durch explizites Entladen rückgängig gemacht werden – was wiederum voraussetzt, dass keine Referenzen auf diese Klassen gehalten werden.

Eine Zugriffskontrolle über Klassen ist nicht ganz unproblematisch. Durch Serialisierung können beispielsweise Objekte zwischen unterschiedlichen Klassenversionen transportiert werden (s. [New]). Selbst wenn abgeleitete Klassen durch Classloader gesperrt sind, können Objekte über Basisklassen manipuliert werden. Nicht zuletzt ist auch der Verwaltungsaufwand für Klassen-Policies beträchtlich. Im Kap. 8 wird gezeigt, wie sich durch spezielle Organisation von Classloadern die Services voneinander in Bezug auf Policies isolieren lassen. Die Einschränkung muss deshalb gemacht werden, weil die logische Trennung von Services, d. h. ihre gegenseitige Unabhängigkeit, zwar in Bezug auf die Korrektheit schnell erreichbar ist, in Bezug auf die „Liveness“-Problematik – also die Verfügbarkeit – im jetzigen Stadium der Java-Runtimes aber nicht.

An dieser Stelle soll nur der „klassische Mechanismus“ der Verwendung von Classloadern in Application-Servern dargestellt werden. Die neuesten Versionen von Eclipse oder Websphere verwenden bereits den in Kap. 8 besprochenen Peer-to-Peer-Classloader-Mechanismus.

Der „klassische“ Mechanismus folgt dem aus Java bekannten hierarchischen Prinzip: Eine Hierarchie von Classloadern lädt zunächst Java-Basisklassen und Extensions, anschließend Application-Server-Klassen und -Bibliotheken und zuletzt Applikationen und deren Module (s. Abb. 7.6). Auf Wunsch können sogar die einzelnen Module durch separate Classloader geladen werden.

Die hierarchische Anordnung wirft aus Sicht der Security mehrere grundsätzliche Probleme auf:

Die Trennung von Klassen des Application-Servers und Klassen, die die Applikationen mitbringen, ist ungenügend. Es ist somit schwierig, unterschiedliche Versionen von Bibliotheken auseinander zu halten. Dies stellt ein Problem für namespace-basierte Isolation dar.

Die Default-Lookup-Policy delegiert Anfragen nach Klassen zunächst an die in der Hierarchie darüber liegenden Loader. Alle Klassen, die durch höhere Loader geladen werden, sind auf den unteren Ebenen sichtbar (s. Abb. 7.6). Selbst wenn Applikationen keine Referenz auf Objekte solcher Klassen erhalten können, so sind statische Methoden dennoch erreichbar.

Die Klassen des Application-Servers befinden sich in der Hierarchie höher als die Applikationsklassen. Entwickler können dadurch ungewollt Abhängigkeiten zu Application-Server-spezifischen Klassen einbauen und so Wartungsprobleme verursachen.

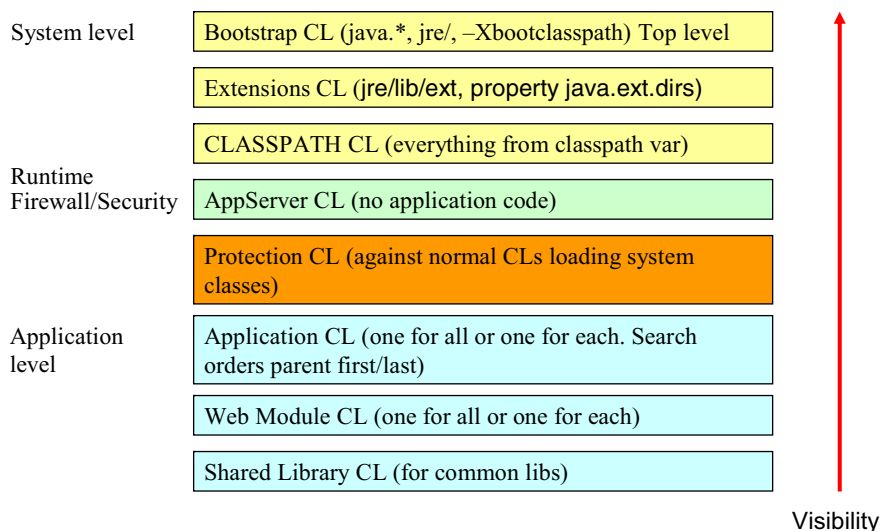


Abb. 7.6 Classloader-Hierarchie für Application-Server

Das Ändern der Default-Lookup-Policy – beispielsweise eine Delegation von Suchanfragen zuerst an weiter unten liegende Loader statt nach oben – ergibt bestenfalls schwierig zu diagnostizierende `ClassCast-Exceptions`, oder schlimmstenfalls auch Sicherheitslücken, wenn nämlich unterschiedliche Versionen von Klassen unterschiedliche Sicherheitspolicies ausdrücken. Abgesehen davon tritt durch die großzügige Verwendung statischer Methoden im Java-Code das grundsätzliche Problem der „Ambient Authority“ auf.

Probleme mit `ClassNotFoundExceptions` werden gerne dadurch gelöst, dass Bibliotheken einem Loader übergeben werden, der sehr hoch in der Hierarchie steht. Damit stehen die Klassen zwar mit großer Wahrscheinlichkeit der Applikation, die sie benötigt, zur Verfügung, aber eben auch allen anderen.

Die Aktion des Kopierens einer Bibliothek in eine Stelle im File-System kann je nach Konfiguration der Classloader sehr unterschiedliche Konsequenzen haben: Entweder werden Klassen nicht oder zu oft gefunden, oder aber die falschen Versionen zuerst gefunden, etc. Das Problematische daran ist, dass hier ein Grundprinzip sicherer Software verletzt wird: Die Security Policy kann auf verschiedenen semantischen Ebenen manipuliert werden, ohne dass ein Bewusstsein über die Konsequenzen existiert. Klassisches Beispiel dafür ist die Administration von Web-Servern, wo durch eine simple Änderung eines Zugriffsrechts auf ein Verzeichnis komplexe Security Policies ausgehebelt werden können. Ein anderes Beispiel bildet die viel beschworene „DLL-Hell“ im Windows-Betriebssystem. Diese Problematiken werden wir später in unserer Diskussion grundsätzlicher Sicherheitsmechanismen wieder aufnehmen.

Grundsätzlich ist aber eine auf Namespaces basierende Isolation von Code machbar und theoretisch nicht weniger mächtig als andere Formen der Isolation

wie ACLs oder Capabilities, wie bereits Wallach et al. in ihrer Untersuchung der Java-Sicherheit gezeigt haben (s. [WF]). Die geschilderte klassische Art der Verwendung wirft aber die Frage auf, ob es sinnvoll ist, einen komplexen Mechanismus, der bereits genügend Frustration über schlecht durchschaubare Cast-Exceptions und Class-Not-Found-Exceptions verbreitet, auch noch mit der Durchführung von Sicherheits-Policies zu beauftragen.

Die Komplexität steigt noch weiter an, wenn auch das dynamische Laden und Entladen von Code beabsichtigt wird. OSGI löst die damit verbundenen Probleme über den massiven Einsatz von Constraints (siehe Kap. 8).

7.3 Security-Interfaces zwischen Server und Applikationen

Bevor wir auf die Schnittstelle zwischen Application-Servern und Infrastruktur eingehen, lohnt es sich, noch einmal kurz die Anforderungen der Applikationen an den Application-Server zu betrachten. Welche Sicherheitsdienste benötigt eine Applikation eigentlich? Besser gesagt: Welche Sicherheitsfunktionen und Eigenschaften benötigt sie? Denn das Wort „Dienste“ klingt doch sehr nach einer „Pflicht“ der Applikation, durch den Aufruf von Funktionen oder Methoden Sicherheitsfunktionen auszulösen. Der Aufruf eines Service bindet den Applikationscode örtlich und zeitlich an den Service. Die deklarative Festlegung, dass ein gewisses Verhalten erwünscht ist, entkoppelt hingegen Applikation und Dienst.

Wir müssen also zwischen dem Code einer Applikation und ihrem Ausführungskontext unterscheiden. Ein Beispiel: Eine Applikation verarbeitet Lohndaten und schickt sie an ein anderes System. Die Daten sind als vertraulich klassifiziert und dürfen daher nur unter Beachtung von Vertraulichkeit und Integrität übertragen werden. Das bedeutet jedoch weder, dass die Applikation nun notwendigerweise Code enthalten muss, um die Vertraulichkeit zu implementieren, noch dass die Applikation Aufrufe an Systembibliotheken tätigen muss, um beispielsweise eine SSL-Session zu erzeugen. Stattdessen könnte in den Konfigurationsdaten der Applikation stehen, dass alle Transportkanäle vertraulich und integer sein müssen. Ist diese Applikation damit nun „security aware“? Das war lange Zeit im CORBA-Bereich eine Kernunterscheidung, nach der der ORB einer Applikation spezielle Interfaces zur Kontrolle der Sicherheit bzw. ihrer Implementation anbieten musste (fortgeschrittener Modus) oder selbst für die Sicherheit einstehen musste (einfacher Modus). Im Geschäftssinn ist die obige Applikation sicherlich „security aware“, denn ohne die Beachtung der richtigen Transportkanäle entstehen Businessfehler. Aus der Sicht der Entwickler jedoch würde man diese Applikation wohl eher als „security unaware“ bezeichnen.

Wahrscheinlich gibt es in der Realität kaum Applikationen, die „security unaware“ sind, denn die meisten wollen doch zumindest authentifizierte User haben. Der Begriff der Security Awareness stellt eine gewisse Verkürzung auf die softwaretechnische Sicht dar und lässt dabei Businessvorgaben ziemlich außer Acht. Leider wird in den meisten Fällen die Applikation nicht anhand von Business

Requirements modelliert, oder die UML-Modelle beinhalten nicht die Sicherheitsanforderungen aus Businesssicht.

Was ist aber in der verkürzten Sichtweise mit „security unaware“ gemeint? Am ehesten wohl der Umstand, dass eine Applikation weder im Code noch in ihrer Konfiguration angepasst werden muss, wenn sich die Sicherheitsmechanismen in der Infrastruktur einer Firma ändern. Das wäre etwa der Fall, wenn eine Firma von einem symmetrischen Key-Distribution-System auf ein asymmetrisches umsteigt, oder wenn sich die Zugangsprotokolle und nötigen Credentials zum Zugriff auf Backend-Systeme oder Datenbanken ändern. Im ersten Fall bedeutet jedoch der Umstieg auf eine PKI meistens umfangreiche Anpassungen und es ist unwahrscheinlich, dass solche Änderungen tatsächlich ganz transparent vonstatten gehen können. Im zweiten Fall jedoch stehen die Chancen recht gut, dass die Applikation unverändert bleiben kann – wenn sie die entsprechenden Interfaces zu ihrer Umgebung eingehalten und sich nicht von konkreten Mechanismen abhängig gemacht hat.

Welche Änderungen könnten in der Umgebung passieren, die Auswirkungen auf die Applikation besitzen? Dazu eine kleine Liste von Beispielen:

- Authentisierungsmethoden werden geändert: Zum Beispiel der Einsatz biometrischer Merkmale anstelle von UserID und Passwort. Dieses Problem haben wir bereits behandelt und festgestellt, dass die konkreten Methoden der Authentisierung innerhalb der Applikation nichts zu suchen haben.
- Das Verzeichnis der Rechte eines Users ist von einem File in einen LDAP-Server gewandert. In diesem Fall müsste der Code der Applikation geändert werden.
- Die Rollen von Usern haben sich geändert und heißen nach einer Reorganisation anders. Müssen die Rollennamen innerhalb der Applikation jetzt angepasst werden?
- Die Verbindung zu einer Datenbank auf dem Host hat sich verändert. Müssen nun die UserID/Passwortkombinationen, die die Applikation bisher verwendet hat, geändert werden?

Aus diesen Beispielen können wir einen kleinen Anforderungskatalog bezüglich nötiger Interfaces ableiten:

- Kein Authentisierungscode in der Applikation. Wenn die Applikation wissen möchte, wer der User ist, geht das nur über Standard-Interfaces (Objekte), die vom Application-Server bereitgestellt werden. Diese können durch die Applikation parametrisiert oder sogar ersetzt werden, aber dies geschieht in gesonderten Modulen (zum Beispiel JAAS).
- Kein Authorisierungscode in der Applikation. Wenn die Applikation verhindern will, dass User ohne bestimmte Rechte auf Ressourcen der Applikation zugreifen, dann kann sie das über die Konfiguration nötiger Rollen pro Resource oder über Abfragen an Standard-Interfaces tun.
- Keine Credentials für den Zugriff auf Backends im Code. Stattdessen sollte ein Konfigurations-Interface benutzt werden.

- Kein Aufbau eigener Verbindungen zu Backends. Stattdessen sollte man die Verbindung über ein Container-Interface aufbauen lassen.
- Kanalsicherheit sollte nicht programmiert werden müssen, d. h. Vertraulichkeit und Integrität sollten von einem sicheren Kanal zur Verfügung gestellt werden.
- Keine Ausführungskontexte selber erstellen bzw. an einen Thread binden. Dies sollte nur über ein Container-Interface (Konfiguration) oder ein Standard-Interface (hier eine Methode des Subject) geschehen.

Allerdings muss eine Applikation noch einige andere Dinge beachten, zum Beispiel, dass durch JAVA EE der Zugriff auf Betriebssystemressourcen von Seiten der Applikation nicht geregelt ist. Hier kommt es also auf die Rechte des Applikationsservers an. .NET erlaubt z. B. die Ausführung solcher Zugriffe mit der Identität eines Windows-Users.

Zum Abschluss unserer Diskussion der Application-Interfaces noch eine interessante Fragestellung: Was passiert, wenn die Applikation mehr Dienste braucht, als durch die Standard-Interfaces des Servers zur Verfügung gestellt wird? Ein Beispiel dafür wäre die Erweiterung des Aufrufkontextes mit speziellen Sicherheitsinformationen oder der Wunsch nach einem bestimmten, automatischen Logging von Requests. In diesem Fall treten zwei Probleme auf, nämlich: wie kann die Applikation den Code des Servers erweitern, und wie kann dies auf portable Weise zwischen Servern verschiedener Hersteller geschehen?

Einer der Hauptkritikpunkte am Design von Enterprise Java Beans war das Fehlen eines Interceptor-Konzeptes, das es den Beans erlaubt hätte, den Request selbst zu manipulieren. Im CORBA-Umfeld besteht hingegen seit langem das Konzept des Portable Interceptors. Hierbei werden die Requests beim Senden und Empfangen an speziellen Interception Points gefiltert und die ORBs stellen entsprechende Funktionen für die Registrierung der Filterfunktionen bereit. Die gewünschten Informationen werden dann zusammen mit den eigentlichen Requests „huckepack“ übertragen.

Als Muster ist das Portable Interceptor Pattern sehr interessant, weil es eine minimalintrusive Schnittstelle zwischen zwei getrennten Komponenten (Applikation und Server) auf portable Weise anbietet. Interessante Anwendungsgebiete wie die Übertragung impliziter Kontexte oder die Ergänzung von Verarbeitungsaspekten finden sich bei [Oser].

Über die bereits besprochenen Applikationen hinaus gibt es auch solche Applikationen, die ganz besondere Sicherheitsanforderungen haben, beispielsweise selbständig sichere Kanäle nach außen errichten zu wollen oder die auf ganz besondere Backend-Systeme Rücksicht nehmen müssen. Weitere Möglichkeiten sind die Unterstützung einer speziellen Form der Authentisierung oder die Teilnahme an einer serviceorientierten Architektur.

Hier bieten die Application-Server typischerweise Bibliotheken und Frameworks mit den entsprechenden Funktionen an. Die vier wichtigsten im Bereich JAVA EE sind:

- JSSE für kanalbasierte Sicherheit,
- GSS-API für tokenbasierte Kommunikation,

- WebServices Security für XML/SOAP-basierte Kommunikation im Rahmen einer serviceorientierten Architektur (SOA) und
- JAAS/ACC für die verschiedenen Mechanismen der Authentisierung und Autorisierung.

Dies sind keine automatischen Containerfunktionen (obwohl sie natürlich zur Implementation der oben erwähnten deklarativen Sicherheit ebenfalls von Containern verwendet werden können). Hier ist jedoch gemeint, dass diese Bibliotheken explizit von den Applikationen selbst angesprochen werden. Entsprechend höher ist der Anteil an Code innerhalb der Applikation, der über Sicherheitsdinge bis hin zu Mechanismen und Protokollen Bescheid weiß.

7.4 Repräsentation von Identität: Die Erstellung eines Subjects

In diesem Abschnitt wird die Realisierung von Authentisierung und Autorisierung durch Application-Server näher betrachtet. Wir haben zwar oben gesehen, dass für die Applikationen letztlich nur ein erstelltes Subject und seine Credentials interessant sind – woher das Subject kommt, wer es wie erstellt hat etc. ist zwar ein wichtiges Qualitätsmerkmal der Sicherheit, aber momentan von der Applikation selbst nicht bestimmbar. Wenn solche wichtigen Dinge jedoch dem Application-Server bzw. Container überlassen werden, dann lohnt es sich schon einmal, genauer zu betrachten, wie es zu der Repräsentation einer externen Identität in Form eines Subject-Objektes kommt und wie damit weiter umgegangen wird.

7.4.1 Grundlegende Szenarien

Bevor wir uns damit beschäftigen, wie die Application-Server mithilfe von JAAS oder anderen Mechanismen ein Subject erstellen, sollen zunächst einmal die unterschiedlichen Szenarien gezeigt werden, unter denen die Erstellung oder Verwendung eines Subjects erfolgt (Abb. 7.7).

Abhängig von der Rolle des Application-Servers als Standalone-Server oder im Verbund mit anderen, in einem SSO-Kontext etc., werden unterschiedliche Eingänge für die Requests genutzt. Die Implementationen unterscheiden sich dabei deutlich zwischen den verschiedenen Herstellern von Application-Servern. Die folgenden Diagramme orientieren sich grob am IBM Websphere Application-Server, dürften jedoch vom Prinzip her gut übertragbar sein (für eine Beschreibung der BEA-Weblogic-Mechanismen siehe [Chugh]. Erstaunlich, wie unterschiedlich die Terminologie der Hersteller selbst in Bezug auf Basiskonzepte wie „Rolle“ oder „User“ sein kann).

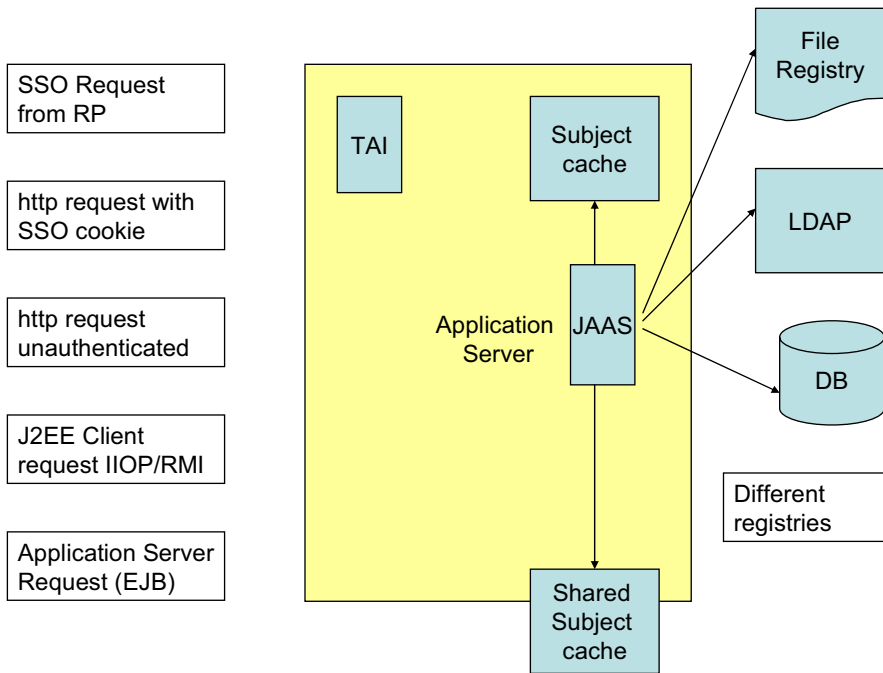


Abb. 7.7 Mögliche Request-Typen und Registries bei der Erstellung eines Subjects

7.4.2 Initiale Authentisierung

Als *initiale Authentisierung* bezeichnen wir die Situation, wenn ein User sich gegenüber einem Application-Server durch Angabe von Credentials (das heißt abhängig vom Authentisierungsmechanismus UserID/Passwort oder Zertifikat etc.) authentisiert. Der Application-Server überprüft diese Angaben selbst durch Abgleich mit einem Verzeichnis (zum Beispiel, indem ein LDAP-bind (UserID, Passwort) durchgeführt wird) und erstellt ein Containerobjekt vom Typ „Subject“. Anschließend fragt der Application-Server die Rechte des Users im Verzeichnis ab (das kann das gleiche Verzeichnis oder ein anderes sein) und füllt die Rechte in den Subject-Container (s. Abb. 7.8). Dadurch entsteht eine Repräsentation des Users innerhalb dieses Application-Servers. Aber Vorsicht: Diese Repräsentation ist zunächst ausschließlich innerhalb dieses Application Servers und nur in Bezug auf die beiden Verzeichnisse gültig, die zur Authentisierung und Authorisierung verwendet wurden. Diese beiden Verzeichnisse definieren die Domäne, in der die Repräsentation gültig ist.

Das Subject Object selbst ist innerhalb des Application-Servers ohne kryptografische Absicherung – es befindet sich quasi innerhalb einer Trusted Computing Base. Es kann mit einer Unix User- oder Group-ID auf einem Rechner verglichen werden.

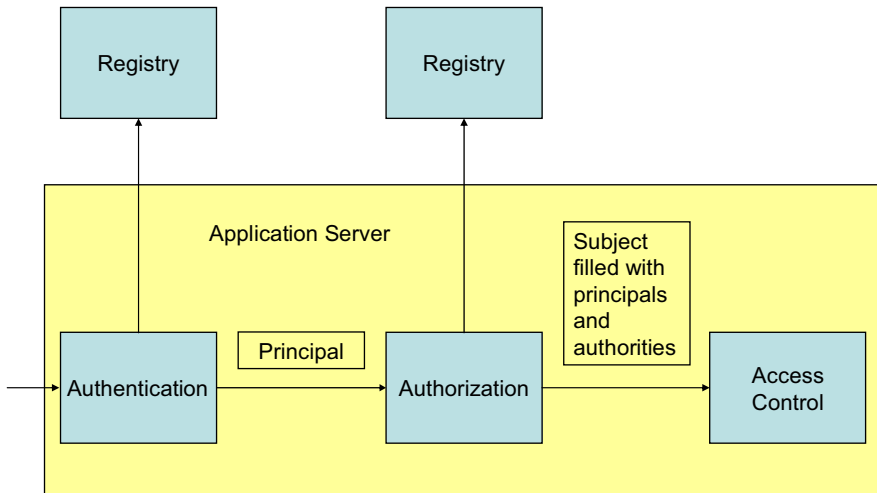


Abb. 7.8 Vorgang der Erstellung eines Subjects im Application-Server

Der Request des Users könnte jetzt unter Verwendung dieses Subject Objects kontrolliert abgearbeitet werden. Der Code der Applikation kann in Abhängigkeit von den Inhalten spezifische Entscheidungen treffen oder den User sogar durch `Subject.doPrivileged(...)` impersonifizieren, wie wir bei der Diskussion von Java EE gesehen haben.

Würde man es dabei bewenden lassen, würde sich beim nächsten Request des Users das gleiche Szenario wiederholen – er würde zur Authentisierung aufgefordert, die Credentials würden geprüft usw. Um dies zu vermeiden, muss das Ergebnis der Authentisierung – das erstellte Subject Object – gespeichert und seine Verbindung zum jeweiligen User festgehalten werden. Stellt der User dann den nächsten Request, kann der Application-Server diesen Request sofort der gespeicherten Repräsentation des Users (Subject) zuordnen und braucht nicht ein weiteres Mal zu authentisieren. Wir werden im Anschluss an die Szenarien die verschiedenen Möglichkeiten, einen Request einem User zuzuordnen, untersuchen. Für den Moment beschränken wir uns auf den einfachsten Mechanismus: Das Setzen eines Cookies mit einer Nummer im Browser des Users, bzw. auf Serverseite das Abspeichern des Subject Objects unter dieser Nummer (weiter unten beschäftigen wir uns mit der Problematik der Absicherung dieses Tokens).

Ein Request von einem bereits authentisierten User stellt eine zweite Form des Eingangs in den Application-Server dar.

7.4.3 Bereits diesem Application-Server bekannter User

Beim nächsten Request nach der initialen Authentisierung begleitet das Token (Cookie), das die Verbindung zum gespeicherten Subject ermöglicht, den Request.

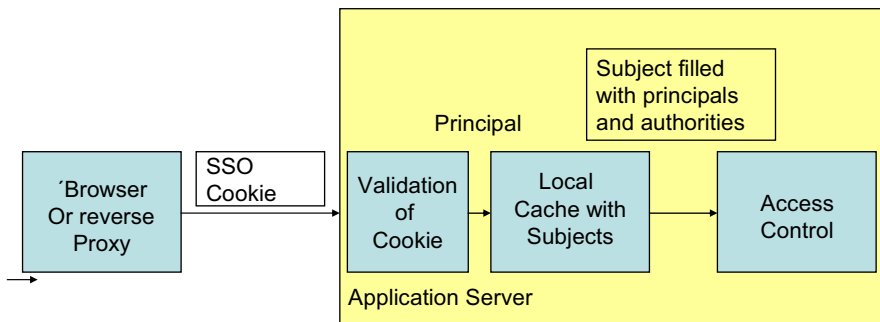


Abb. 7.9 Auffinden des Subject-Objekts bei bereits authentisiertem User

Der Application-Server erkennt das Identitäts-Token, verwendet es, um das richtige Subject zu finden und kann dann der Applikation dieses Subject zur Verfügung stellen. Somit ist keine Authentisierung und keine Authorisierung mehr nötig – die entsprechenden Repositories werden nicht angefasst, wie Abb. 7.9 zeigt.

Das Subject ist dabei lokal bereits vorhanden – es ist ja vom Application-Server selber erzeugt worden. Beim lokalen Cache erübrigen sich weitere Überprüfungen – vorausgesetzt, der Code zum Zugriff ist durch Code Access Security gesichert. Liegt das Subject in einem shared Cache, sollte eine Prüfung auf Authentizität des Subjects erfolgen, z. B. indem das Subject verschlüsselt abgelegt wird.

Der gerade geschilderte Fall und alle weiteren, die jetzt noch folgen, unterscheidet sich vom ersten Fall dadurch, dass der Application-Server keine initiale Authentisierung mehr vornimmt – das heißt, der User (oder „Holder-of-Key“) benutzt seine initialen Credentials nicht, sondern nur das vorher vom Server übermittelte Token. Man spricht in diesem speziellen Fall jedoch nicht von einer *Identity Assertion*: Dort behauptet der Requestor eine Identität, und der Empfänger akzeptiert diese Behauptung und führt keine Prüfung mehr auf Basis der initialen Credentials dieser Identität durch. Allerdings versucht der Empfänger (Server) natürlich, die Gültigkeit der Aussage (Token) zu prüfen bzw. vertraut dem Sender.

Deutlicher wird das Prinzip der Identity Assertion unter der Annahme, dass Identitätsprüfer/Aussteller von Tokens und Empfänger unterschiedliche Instanzen sind. Dies ist bei einem vorgelagerten Single-Sign-On, oder wie im folgenden Beispiel zweier Application-Server, die einen SSO-Verbund bilden, der Fall.

7.4.4 Bereits gegenüber einem Application-Server bekannter User

Nehmen wir also an, dass ein Request eines Users bereits von einem Application-Server authentisiert wurde. Dieser Server hat beim Browser oder Reverse Proxy ein Cookie hinterlassen, das die Identität des Users repräsentiert – natürlich kryptografisch mit einem Schlüssel gesichert. Im Websphere-Umfeld wird dieses Co-

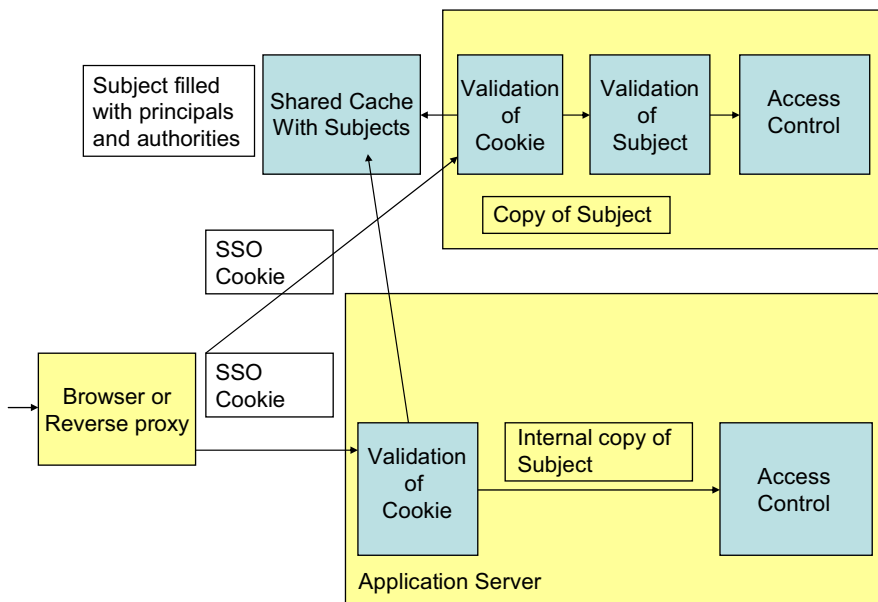


Abb. 7.10 Subject-Erstellung mithilfe eines LTPA-Token (SSO-Cookie)

kie als *Lightweight Third Party Authentication* (LTPA)-Token bezeichnet. Es darf nicht mit dem Token eines vorgelagerten SSO-Servers verwechselt werden, sondern es repräsentiert eine Vertrauensbeziehung direkt zwischen verschiedenen Application-Servern: Diese müssen beide den Schlüssel zum Erstellen bzw. Lesen des Cookies kennen.

Der nächste Request dieses Users kann nun auf einem anderen Application-Server landen, begleitet von dem erwähnten Cookie. Der Server prüft das Cookie mit dem gemeinsamen Schlüssel und entnimmt ihm die Identität des Aufrufers. Der Server sucht unter dieser Identität im gemeinsamen Cache nach einem passenden Subject, prüft dieses ebenfalls anhand eines Schlüssels auf Authentizität und kopiert es zu sich (s. Abb. 7.10). Es erfolgt keine weitere Autorisierung (die nötigen Informationen sind ja bereits im Cookie enthalten). Es muss keine Registry mehr kontaktiert werden, da kein neues Subject erzeugt wird.

Ein Subject Object ist eine javaspezifische Datenstruktur, die zur Laufzeit innerhalb eines Application-Servers Informationen zur Identität und zu den Rechten von Principals sowie deren Credentials (öffentliche und geheime) enthalten kann. Der Gedanke, Subject-Strukturen direkt zwischen Application-Servern auszutauschen, ist demnach zunächst nicht gerade selbstverständlich: Es setzt absolutes Vertrauen zwischen diesen Servern voraus, denn dadurch findet beim empfangenden Server weder eine Authentisierung noch eine Autorisierung des Users statt. Dieses Vertrauen wird durch einen gemeinsamen symmetrischen Schlüssel zur Sicherung des Cookies ausgedrückt. Auch für die Absicherung der Subject-Information im Cache wird ein (anderes) gemeinsames Geheimnis benötigt. Der

Besitz dieses Schlüssels ist gleichbedeutend mit der Möglichkeit, beliebige Subjekte zu erstellen und auf den anderen Application-Servern unter dieser Identität Aufrufe auszuführen.

Es kann sein, dass ein Applikation-Server ein SSO-Cookie mit einem Request erhält, aber das entsprechende Subject-Objekt nicht lokalisieren kann, da es nicht in seinem lokalen Cache vorhanden ist (d. h. der Server hat für den User noch kein Subject erzeugt). Jetzt hat der Application-Server mehrere Möglichkeiten:

- Es findet sich kein passendes Subject im gemeinsamen Cache. Das SSO-Cookie enthält jedoch einen Hinweis auf den ersten Aussteller dieses SSO-Cookies, der demzufolge auch ein Subject erstellt haben muss. Nun kann der Application-Server den erstausstellenden Server kontaktieren und eine Kopie des Subjects erhalten. Auch hier werden symmetrische Schlüssel zur Absicherung der Authentizität des Cookies eingesetzt.
- Der Application-Server nutzt den Principal aus dem SSO-Cookie, erfragt in einer Registry die dazugehörenden Authorisierungen und füllt mit ihnen ein neu erzeugtes Subject auf. In der Folge verwendet er dieses Subject.
- Der Application-Server sieht an dem SSO-Cookie, dass bei der initialen Authentisierung besondere Informationen zur Verfügung standen (z. B. Ort der Authentisierung, Qualität etc.), die jedoch im SSO-Cookie selbst nicht gespeichert sind. Kann der Application-Server an das originale Subject nicht herankommen, indem er den Hersteller des Subject direkt kontaktiert oder es einem Cache entnimmt, muss er in diesem Fall darauf verzichten, selbst ein neues Subject zu erstellen und eine eigene Autorisierung durchzuführen. Er muss stattdessen den Request mit einem Fehler abrechnen, da er ohne die Zusatzinformationen zur Authentisierung kein äquivalentes Subject herstellen könnte.

Abbildung 7.11 zeigt den Fall, dass die Identität des Aufrufers dem Cookie entnommen werden kann, aber kein Subject zur Verfügung steht (weder im Cache noch über JMX-Aufrufe). Jetzt muss unter Verwendung eines Rechteverzeichnisses ein neues Subject erstellt werden, jedoch ohne erneute Authentisierung.

Das letzte der obigen drei Szenarien ist von großer Wichtigkeit, wenn beispielsweise Step-up-Authentisierungen eingesetzt werden, d. h. die Qualität der Authentisierung eine große Rolle für die Zugangsberechtigungen spielt. Der Verlust solcher Informationen im Subject könnte große Sicherheitslöcher eröffnen, wie wir jetzt zeigen werden.

Bei der Erstellung eines Subjects im Application-Server werden spezielle Daten im Objekt gespeichert, zum Beispiel die IP-Adresse des Rechners, über den der Aufruf das System betreten hat. Diese Information könnte in einem SSO-Token eines vorgelagerten Servers protokolliert worden sein. Der nächste Application-Server liest diese Information aus und speichert sie im Subject-Objekt. Wichtig dabei: Diese Information ist dynamisch und sie ist nicht in einem Repository vorhanden – wie es etwa statische Rollen des Users sind. Die gängigen Application-Server bieten zur Verarbeitung dieser Information spezielle Frameworks an, meist Erweiterungen auf Basis von JAAS wie im Kap. 5 beschrieben.

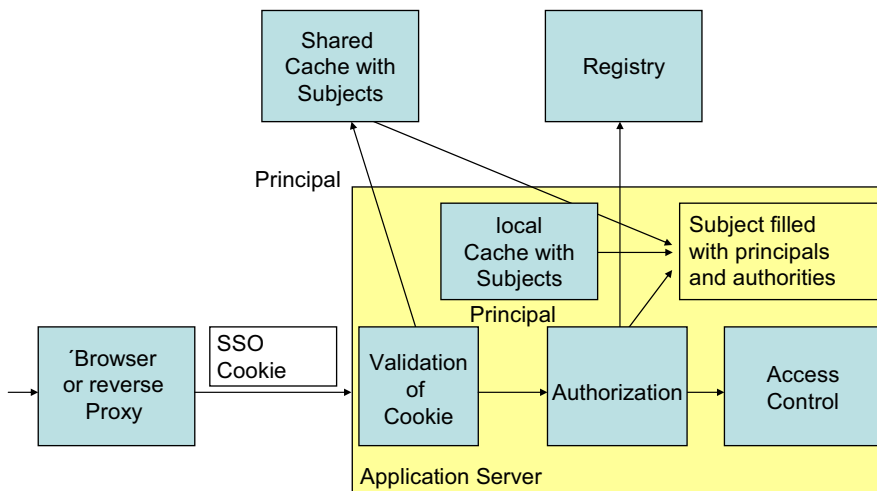


Abb. 7.11 Neuerstellung eines Subjects mithilfe einer Rechte-Registry

Das Problem mit diesem Verfahren tritt erst auf, wenn durch einen Server-Crash die Information dieses Subjects verloren gegangen ist. In diesem Fall verfügt der Client beispielsweise noch über ein SSO-Token mit der Identität des Users, der Application-Server, der das erweiterte Subject gehalten hat, ist jedoch abgestürzt. Jetzt leitet das Load-Balancing-Subsystem den Request des Clients an einen anderen Application-Server weiter, der jedoch die erweiterte Subject-Information nicht besitzt. Kann der Server nicht an die erweiterte Information herankommen, muss er entweder den Request ablehnen oder aber er führt eine neue Authorisierung anhand des überlieferten SSO-Tokens durch – das heißt, er akzeptiert die Identitätsbehauptung, erzeugt ein neues Subject und füllt dessen Credentials durch Abfrage eines Repositories, das natürlich die erweiterte, dynamische Information nicht enthält. Dadurch ist ein subtiler Wechsel in den Credentials und damit den Rechten des Clients eingetreten, der je nach Bedeutung der verlorenen Information gefährlich sein kann.

7.4.5 Identity Assertion durch vorgelagerte Authentisierung

Der Begriff *Identity Assertion* bezeichnet die Angabe einer (anderen) Identität als Basis für weitere Aufrufe. Im Gegensatz zu einer Authentifikation verifiziert die empfangende Instanz diese Behauptung nicht selbst, sondern vertraut dabei auf die Zusicherung einer vertrauenswürdigen dritten Instanz. Im Bereich der EJB-Containersicherheit beruht Identity Assertion für weitere Aufrufe auf dem CSIv2-Protokoll. Die Anwendung und Konfiguration dieses Protokolls zwischen Application-Servern ist in Kap. 18 des IBM Redbook zum Z/OS Websphere Application-Server [z/OS] sehr ausführlich beschrieben. Die Validierung einer solchen

Übernahme kann ganz unterschiedliche Qualität besitzen, von einem bloßen String, der die Tatsache der Identity Assertion lediglich behauptet, bis hin zu einem von zentraler Stelle signierten Token als Beweis einer nachgewiesenen Identität.

Das Szenario der vorgelagerten Authentisierung haben wir bereits im Buch „Internet-Security aus Software-Sicht“ [KS] einschließlich der Verwendung von Reverse Proxy und Single-Sign-On-Konzepten ausführlich behandelt. Deshalb konzentrieren wir uns hier nur noch auf die Auswirkungen der vorgelagerten Authentisierung auf die Erstellung eines Subjects im Application-Server. Abbildung 7.12 zeigt ein grundlegendes Szenario.

Der Request des Users wird von einem SSO-Token begleitet, in dem der Aussteller (der SSO-Server) die Identität des Users festgehalten hat. Dies kann ein einfaches Name/Value-Paar im http-Header sein, oder aber ein kryptografisch gesichertes Cookie mit User-Namen und weiteren Daten. Der Application-Server erkennt und prüft das Token und erzeugt daraufhin ein Subject Object mit der Identität des Users. Im Anschluss daran fragt der Application-Server das Autorisierungsverzeichnis nach den Rechten des Subjects und füllt damit das Subject.

Die Application-Server bieten hier sehr unterschiedliche Methoden an, eine Identity Assertion zu prüfen und zu übernehmen. Websphere hat für diesen Fall das sog. TAI (Trust Association Interceptor) Konzept, bei dem der Request abgefangen und geprüft wird. Im Anschluss daran erstellt er ein SSO-Token, das er an den Browser des Users zurückgibt. Damit kann der nächste Request des Users wie bei einem bereits bekannten User behandelt werden. Das ursprüngliche SSO-Token wird durch ein eigenes Cookie ersetzt, das auf ein gecachtes Subject zeigt.

Es lohnt sich an dieser Stelle, die Konsequenzen für die Sicherheit des Application-Servers bzw. der ganzen Infrastruktur zu beleuchten. Identity Assertion ist ein auf Vertrauen basierendes Verfahren, bei dem der empfangende Application-

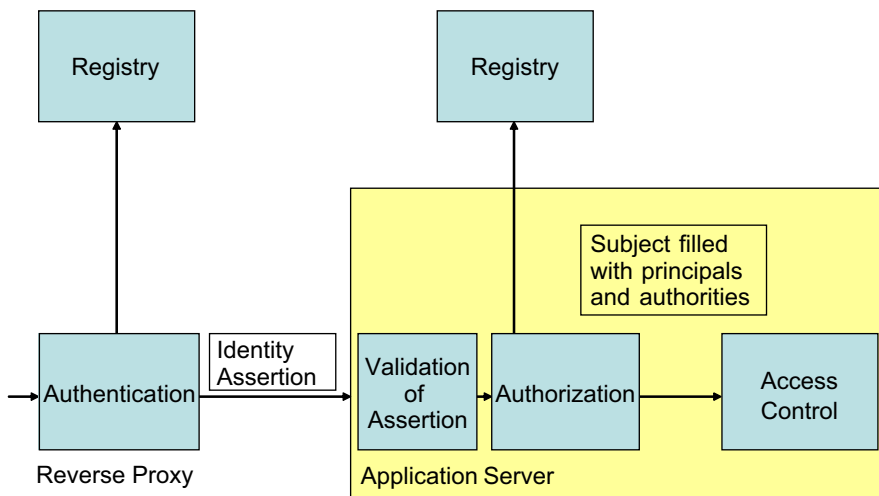


Abb. 7.12 Vorgelagerte Authentisierung durch SSO-Server

Server absolut auf die Korrektheit der Information angewiesen ist. Die Korrektheit kann auf zwei Arten gewährleistet werden:

- Die Information über die Authentisierung kommt über einen dafür vorgesehenen sicheren Kanal. Zusätzlich ist sichergestellt, dass der Sender der Information ein dazu berechtigter Server ist (kanalbasierte Sicherheit).
- Der Erzeuger der Information signiert sie digital, sodass die Authentizität vom Empfänger direkt verifiziert werden kann. Ein sicherer Kanal für den Transport ist nicht unbedingt nötig (objektbasierte Sicherheit).

Das Interceptor-Konzept ist meist flexibel genug, eine Signaturlösung zu unterstützen – vorausgesetzt, der Erzeuger der Information (z. B. CA-Siteminder, CAS, WebSeal) unterstützt die Signatur ebenfalls. Performanceprobleme dürften ebenfalls keine große Rolle spielen, da die Erzeugung und Prüfung der Signatur ja nur einmal pro Session stattfindet.

Momentan dominiert jedoch noch die kanalbasierte Absicherung. Bei dieser ergibt sich jedoch zumindest bei Websphere folgendes in [Botz05b] geschilderte Problem: Setzt man zur Absicherung der Verbindung gegenseitige Authentisierung mit SSL ein, so genügt das nicht. Es reicht nicht, den Client (Reverse Proxy oder Web-Server) zu authentisieren – zusätzlich muss auch eine Autorisierungsprüfung durchgeführt werden, dass dieser Rechner das Recht besitzt, Identity Assertions zu schicken. Je mehr Zertifikate oder gar Root-Zertifikate sich im Trust-Store eines Servers befinden, desto mehr Maschinen können sich authentisieren und desto eher können auch Namenskonflikte auftreten (z. B. könnten verschiedene CAs den gleichen Rechnernamen zertifizieren). Wir wollen außerdem vermeiden, dass alle anderen Application-Server, mit denen unser Server selbstsignierte Zertifikate ausgetauscht hat, uns wie der Reverse Proxy Identity Assertions schicken.

Hier zeigt sich wieder eine Schwäche des Zertifikatskonzeptes in Bezug auf Namen: Letztlich meinen wir doch eigentlich einen privaten Schlüssel, wenn wir sicherstellen wollen, dass wir mit der richtigen Autorität zur Ausstellung von Identity Assertions reden. Außerdem scheint auch der Application-Server hier eine wichtige Indirektion zu vermissen, nämlich den Einsatz von Autorisierungslogik nach der Authentisierung von Maschinen. Wenn es einem Angreifer gelingt, den vorgelagerten Reverse Proxy zu knacken, dann ist er in der Lage, beliebige User zu impersonifizieren und damit die gesamte Infrastruktur zu kompromittieren.

7.4.6 Identity Assertion und Delegation

Nun zur Delegation von Requests zwischen Application-Servern. Momentan wird hierfür vor allem CSIV2 als Protokoll und Datenstruktur eingesetzt, und zwar meist in der einfachsten Form: Der Name des Principals wird dem nächsten Application-Server mitgeteilt. Auch hier ist der empfangende Server auf Gedeih und Verderb dem Sender ausgeliefert, da dieser im Prinzip ja jeden Principal nennen kann.

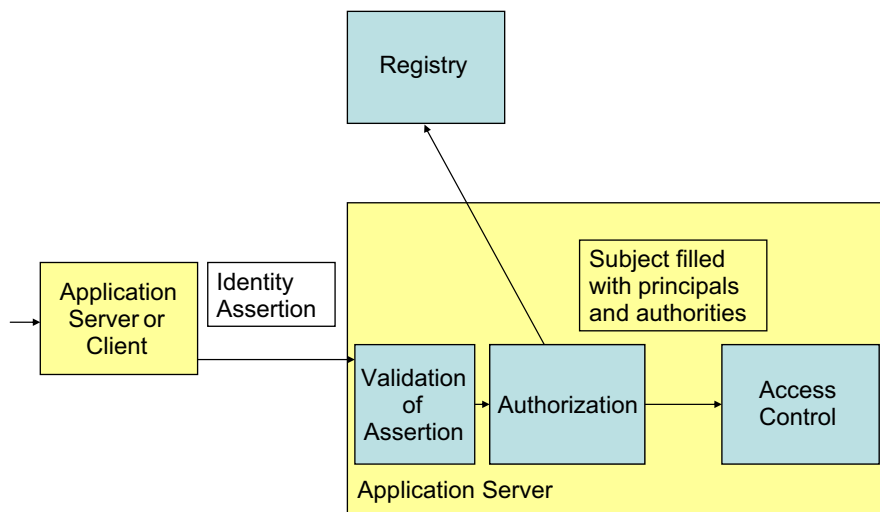


Abb. 7.13 Delegation eines Requests von Server zu Server

Abbildung 7.13 zeigt die betreffende Situation: Ein Application-Server delegiert den Request eines authentisierten Clients an einen anderen Server. Der empfangende Server muss nun ein neues Subject erstellen und dazu bei einer Registry für Autorisierungsinformationen anfragen. Höhere Protokollversionen von CSIV2 erlauben jedoch auch die Übertragung von signierten Token, mit denen z.B. ein vorgelagerter Authentisierungsserver die Identität (mit Gültigkeitsdauer) bestätigen kann. Außerdem können bereits im Token Autorisierungsdaten mit übertragen werden.

Was bedeutet dies? Es bedeutet zum einen, dass nicht nur die Identität des Aufrufers übertragen wird, sondern auch die dazugehörige Autorisierungs-Informationen (Rollen des Aufrufers, Privilegien etc.). Der empfangende Server prüft dieses Token und erstellt daraus ein neues internes Subject-Objekt – ohne neue Authentisierung oder Autorisierung, was einen entsprechenden Performancegewinn bei weiteren Requests und Zugriffen auf Verzeichnisse nach sich zieht. Zum anderen bedeutet ein Subject Object in Token-Form, dass zwischen Aufrufer und Aufgerufenem entweder ein Vertrauensverhältnis besteht oder aber, dass der Empfänger die Gültigkeit der Tokens kryptografisch nachprüfen kann.

Auffällig an diesem Mechanismus ist, dass nicht einfach ein serialisiertes Subject übermittelt wird, sondern ein Satz von prüfbaren Token. Dieser Mechanismus kommt dann auch zwischen verschiedenen Application-Servern zum Tragen – vorausgesetzt, die Server gehören zur gleichen Vertrauensdomain. Voraussetzung ist aber auch hier zunächst eine gegenseitige Authentisierung, entweder über Zertifikate oder eine serverseitige SSL-Verbindung, bei der sich der Client (Sender) mit einem Passwort authentisiert. Wie im vorigen Kapitel ist auch hier wieder die Frage, wie der empfangende Server die Authentisierung der Clients von der Autorisierung für bestimmte Requests trennen kann.

7.4.7 *Standard-Interface für Authorization-Provider: JACC*

JAVA EE definiert zwei verschiedene Interfaces zur Applikation und zur Infrastruktur. Die Interfaces zur Applikation wurden in Bezug auf Sicherheit bereits im Kap. 6 zur Enterprise Security beschrieben. Es handelt sich dabei um Funktionen wie `isCallerInRole(roleName)`, eingebettet im Code der Applikation oder deklarative Sicherheit in Form von Definitionen im Deployment Descriptor bzw. in den Annotations des Codes.

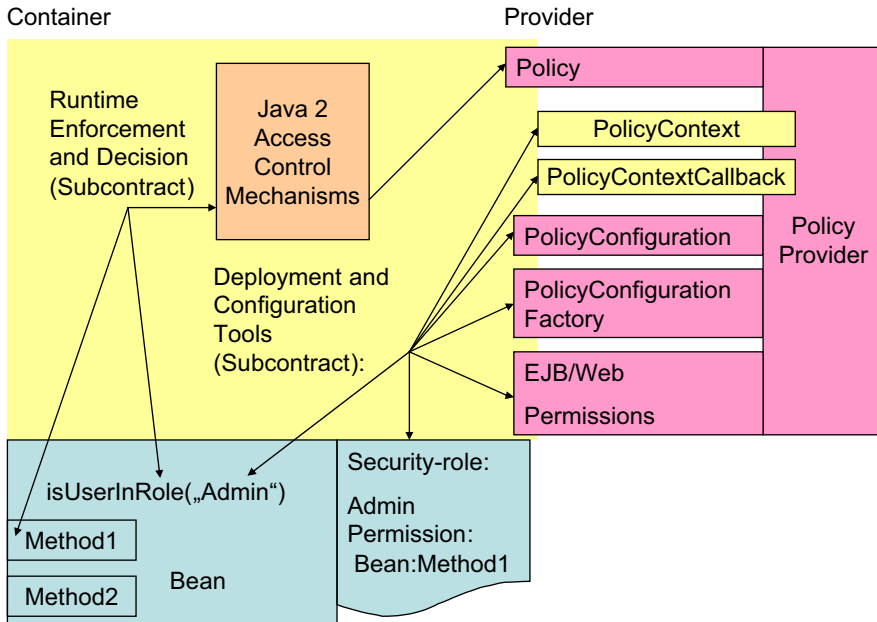
Diese Interfaces stellen einen Vertrag zwischen Container und Applikation dar. Um diesen Vertrag erfüllen zu können, muss der Container auf die Infrastruktur zugreifen, z. B. um die Rollen von Principals zu erfahren. Dies kann proprietär durch die Hersteller der Application-Server geschehen, oder indem ein Standard-Interface zur Infrastruktur definiert wird. Das bekannteste solche Interface ist wohl die JAVA EE Container Architecture, die generell die Einbindung von Backend-Systemen und Datenbanken über Adapter beschreibt. Hier wird ein Vertrag zwischen Container/Application-Server einerseits und Ressource Manager (z. B. Datenbank) andererseits geschlossen.

Im Bereich der Sicherheitsfunktionen war JAAS ein wesentliches Interface, allerdings vor allem auch für Applikationen, um deren Authentisierungswünsche zu erfüllen. Die Frage der Authorisierung blieb dagegen offen, wurde also proprietär gelöst. Der Java Authentication Authorization Contract for Containers (s. [JACC]) soll hier Abhilfe schaffen und eine definierte Schnittstelle zwischen Containern und Herstellern von Systemen zur Konfiguration und Entscheidung von Sicherheitsanfragen (sog. *Policy Providern*) bieten.

Dieser Standard soll es ermöglichen, dass z. B. containereigene Werkzeuge Permissions in einem fremden Policy-Provider-System konfigurieren können. Zur Laufzeit, wenn der Container Sicherheitsentscheidungen benötigt (z. B. weil eine Bean-Methode aufgerufen wird, für die eine Rolle konfiguriert wurde), kann er über die allgemeinen Java-2-Zugriffsmechanismen eine Entscheidung herbeiführen. Dazu legt der Container wie gewohnt ein bestimmtes Permission-Objekt an (JACC definiert hier die nötigen Objekte, um die EJB und Servlet Permissions aus den Deployment-Deskriptoren abzudecken). Anschließend kann er `checkPermission(Permission)` aufrufen. Die nötigen Permission Collections werden dann durch den Provider zur Auswertung bereitgestellt.

Interessant ist, dass durch Callbacks in den Container der Policy Provider zur Entscheidung zusätzliche Informationen erhalten kann, z. B. die Argumente einer Bean-Methode, die aufgerufen werden soll. Sowohl Container als auch Provider registrieren demnach gegenseitig Funktionen.

Abbildung 7.14 zeigt die beiden Hauptverträge des JACC-Standards: Policy Configuration sowie Policy Decision/Enforcement. Im Rahmen der Konfiguration können die Rollen und Permissions, die entweder im Code direkt oder in Deployment-Deskriptoren definiert sind, mittels bereits gestellter Permission-Klassen im Policy-Configuration-Objekt des Providers installiert werden (nicht in Abb. 7.14 ist hingegen gezeigt, dass es pro Instanz eines Applikationsmodules eine spezielle Policy-Konfiguration, bestimmt durch ein PolicyContext Label, geben kann).



Application Module with Deployment Descriptor

Abb. 7.14 Zusammenspiel von Container, Applikation und Policy Provider unter JACC

Zur Laufzeit wird dann der Provider nach Entscheidungen gefragt. Zu den Aufgaben des Policy Providers gehören die folgenden Abbildungen:

- Permissions auf Rollen (d. h. Rollen werden als Bündel von Permissions in JAVA EE aufgefasst).
- Principals auf Rollen (mit der Unterscheidung, ob es sich um Principals im Sinne von Autoritäten oder um Rollen-Principals handelt).
- Rollenreferenzen im Code der Applikationen auf Rollennamen.

Die Integration der vom Provider bereitgestellten Klassen erfolgt konzeptionell über Interfaces, deren Implementation der Provider liefert. Gefunden und geladen werden die Implementationsklassen dann über Einträge in Java Property Files sowie über Reflection-Mechanismen.

7.4.8 Erzeugung von Subjects im Application-Server-Verbund

In diesem Abschnitt wollen wir die eben kennen gelernten Konzepte der Authentisierung und Autorisierung an kompletten Infrastrukturen durchspielen und dabei speziell auf die Absicherung der Komponenten eingehen.

Das Ausgangsszenario ist ein komplexer Verbund aus mehreren Application-Servern, einem Authentisierungsserver und einem Autorisierungsservice. Wir

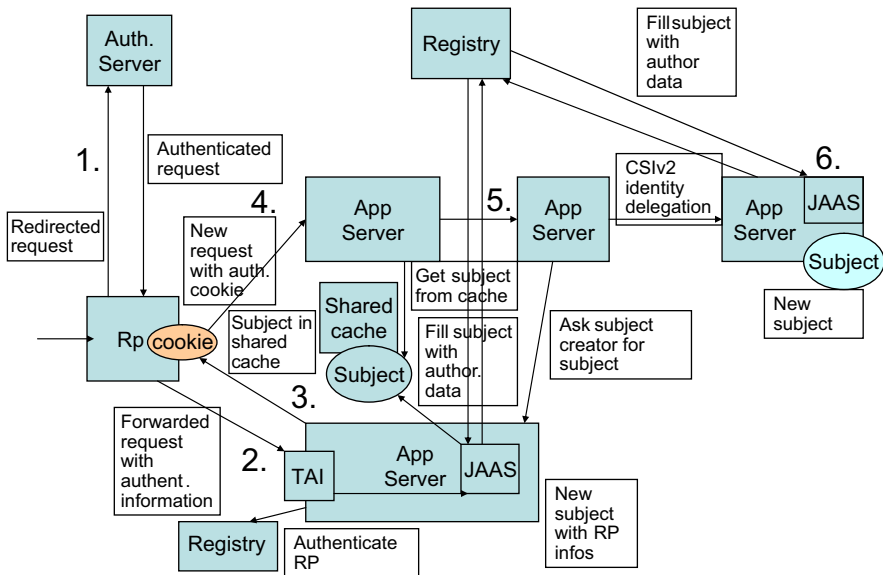


Abb. 7.15 Erzeugung von Subjects im Application-Server-Verbund

verfolgen den Weg eines Requests, der im Verbund mehrfach weiter delegiert wird (Abb. 7.15).

Es laufen dabei die folgenden Schritte nacheinander ab:

- Die initiale Authentisierung wird durch einen externen Authentication-Server vorgenommen. Der Request gelangt zunächst zum Reverse Proxy, wird zum Authentication-Server umgeleitet, dort authentisiert und, mit den Authentisierungsinformation versehen, an den Reverse Proxy zurückgegeben.
- Der Reverse Proxy leitet den Request mit den Authentisierungsinformationen im Namen des Callers an einen ersten Application-Server weiter (Identity Assertion mit vorgelagerter Authentisierung). Die Authentizität der Identity Assertion wird über einen Trust Association Interceptor (TAI) geprüft. Zusätzlich wird der Reverse Proxy authentisiert, also geprüft, ob die Identity Assertion vom RP gesendet wurde.
- Der Application-Server erzeugt ein Subject unter Zuhilfenahme des Rechteverzeichnisses (Autorisierung). Er erzeugt ein SSO-Cookie und gibt dies an den RP zwecks Behandlung weiterer Requests zurück. Das erzeugte Subject wird im shared Cache des Serververbunds abgelegt.
- Es erfolgt ein neuer Request an einen weiteren Application-Server. Der RP übergibt das SSO-Cookie an den Server. Der Server validiert das SSO-Cookie und fordert ein dazu passendes Subject im shared Cache an. Das Subject wird zur Ausführung des Requests bzw. zur Delegation an den nächsten Server (falls „Use-Caller-Identity“ gesetzt ist) verwendet.
- Der Request wird an einen weiteren Application-Server mit Identity Assertion delegiert (z. B. durch Weitergabe des SSO-Tokens). Der empfangende Server

hat keinen Zugriff auf den shared Cache und fordert deshalb eine Kopie des Subjects beim ursprünglichen Ersteller an (der im SSO-Token steht). Das übertragene Subject wird wieder normal ohne weitere Zugriffe auf Verzeichnisdienste verwendet.

- Delegation des Requests an einen weiteren Application-Server per CSIV2. Dabei wird lediglich der Name des Callers übertragen. Der Server hat keinen Zugriff auf den shared Cache und kennt den Erstausssteller nicht. Er glaubt die Identität des Callers aufgrund der Vertrauensbeziehung zum vorderen Application-Server (beide haben sich gegenseitig authentisiert). Der Server erzeugt nun ein neues Subject und füllt es mit Informationen aus dem Autorisierungsverzeichnis. Anschließend wird das Subject normal verwendet.

Die bei der Delegation von Requests zwischen den Servern verwendeten Credentials wurden bereits im Zusammenhang mit der Architektur von Application-Servern dargestellt. Der Trend geht dabei eindeutig zu immer mehr Tokens, die Zugriffe autorisieren oder sogar einen Request während seiner Laufzeit begleiten und Daten zur Bearbeitung sammeln. Momentan sind bereits sechs verschiedene Token-Typen beim Websphere-Application-Server im Einsatz.

Kryptografisch gesicherte, d. h. im Allgemeinen signierte Token, haben in verteilten Systemen große Vorteile: Sie ersparen viele Aufrufe von Verzeichnisdiensten. Gleichzeitig verhindern sie die Verbreitung von Geheimnissen wie Login-Credentials innerhalb der Infrastruktur.

Allerdings werden statt digitalen Signaturen momentan – wahrscheinlich aus Performancegründen – hauptsächlich symmetrische Schlüssel zur Sicherung der Token eingesetzt. Dadurch entsteht ein Angriffsvektor auf die gesamte Infrastruktur, denn da diese Schlüssel notwendigerweise allen beteiligten Servern bekannt sein müssen, genügt eine einzige kompromittierte Maschine, um die ganze Infrastruktur zu übernehmen.

7.5 Probleme zwischen JAAS und JAVA EE

Wer die sich Klasse `javax.security.auth.Subject` einmal genauer angesehen hat und außerdem im JAVA-EE-Umfeld mit `getPrincipal()` gearbeitet hat, mag sich gefragt haben, welcher der vielen möglichen Principals eines Subjects jetzt wohl der JAVA EE Principal ist. Denis Pilipchuk hat die dabei auftretenden Probleme zwischen JAAS und JAVA EE kurz beschrieben (s. [Pil]):

- Unklare Feststellung der JAVA-EE-Identität aus Login-Modulen (API-Defizit).
- Inkompatible Mechanismen bei der Delegation (Subject Propagation vs. Re-Creation).
- Abschneiden des Delegations-Principals (API-Defizit).

Diese Probleme haben ihre Ursache in fehlender Standardisierung und haben als Konsequenz, dass jeder Hersteller von Application-Servern hier proprietäre Lösungen verwendet (s. Abb. 7.16). Dadurch fallen die Inkompatibilitäten den meisten Entwicklern nicht auf.

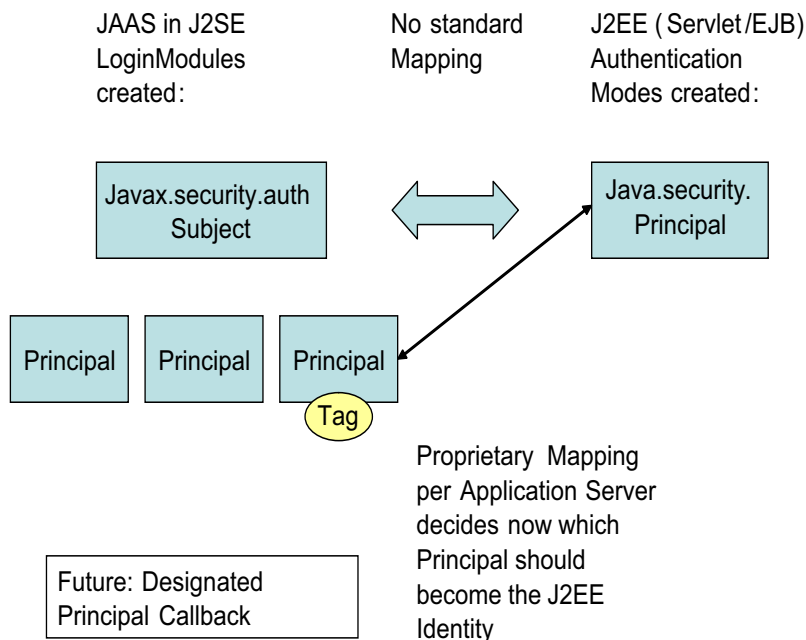


Abb. 7.16 Proprietäre Auswahl der JAVA-EE-Identität

Die unklare Frage der Identität beruht auf einem Defizit im API, das den designierten JAVA EE Principal nicht anzeigen kann. JAVA-EE-Konstrukte wie Servlets und EJBs haben keine Bindung an die JAAS-Authentisierungsmodi.

Applikationsserver verwenden z. B. spezielle Tags, mit denen sie den designierten Principal auszeichnen. Als Folge müssen bei der Analyse von Sicherheits-Policies die jeweiligen serverspezifischen Mechanismen berücksichtigt werden.

Der nächste Punkt – verschiedene Subjects für einen User – nimmt seinen Ursprung in der Verwendung von Identity-Assertion-Mechanismen über Token. Das kann ein Single-Sign-On-Token eines vorgelagerten SSO-Servers sein, das zu einer User-Identität mehrere Principals enthält (abgesehen von weiteren Informationen wie z. B. zur Qualität der Authentisierung). Application-Server können über Interceptor-Mechanismen aus diesen Token ein Subject erzeugen, das die nötigen Principals (die dann im Falle von Java 2 Security direkt in Permissions umgesetzt werden) enthält.

Das Problem taucht in dem Moment auf, wo nicht mehr dieses Subject an andere Server weitergegeben wird, sondern nur noch der Principal (Name). Jetzt muss der empfangende Server versuchen, daraus wieder ein Subject zu erzeugen. Wie im Falle der Subject Propagation von Websphere bereits angesprochen, kann dies durch Propagation einer serialisierten Form des Subjects oder durch eine erneute Erzeugung erfolgen.

In Abb. 7.17 wird ein neues – und diesmal leider unterschiedliches Subject – erzeugt.

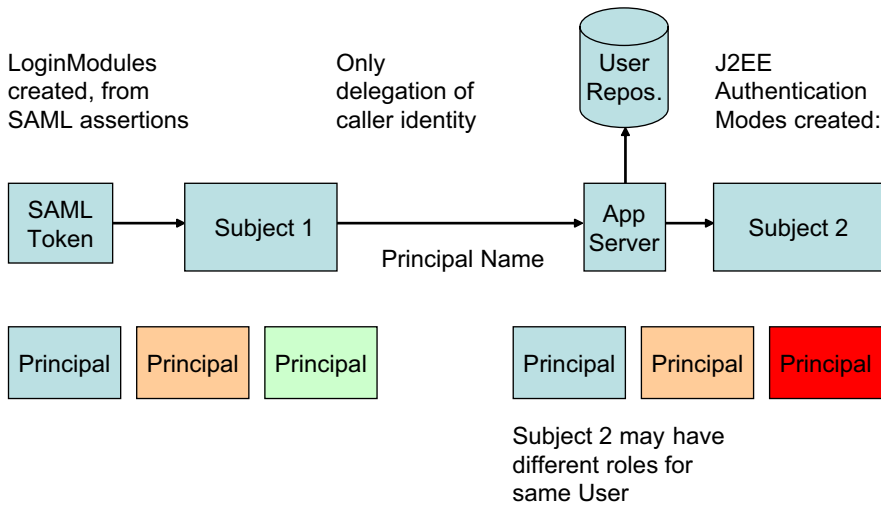


Abb. 7.17 Delegation mit Neuerzeugung des Subjects

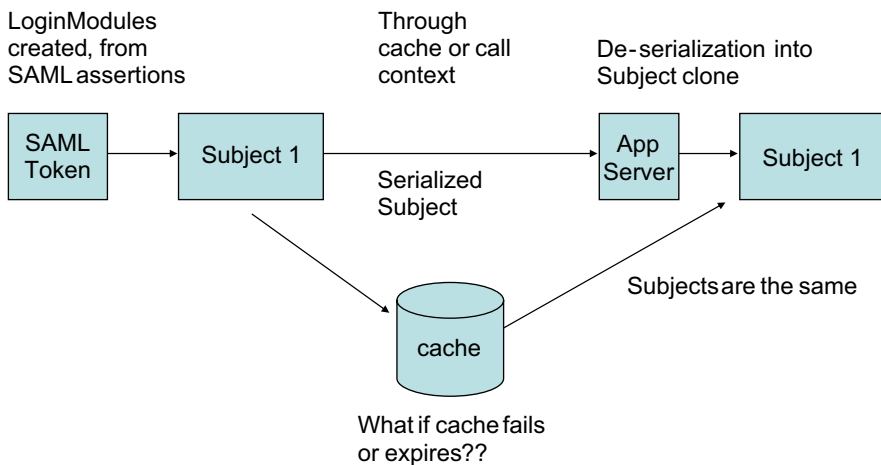


Abb. 7.18 Delegation mit Propagation des Subjects

Aber auch die andere Möglichkeit, ein serialisierter Klon des Subjects wie in Abb. 7.18 dargestellt, ist nicht ohne Probleme: Die Protokolle dafür sind ebenfalls hochgradig proprietär und im Falle der Verwendung von Caches kann es durchaus sein, dass ein Subject nicht mehr vorhanden ist. Dann tritt der gleiche Fall wie weiter oben geschildert ein: Der Request muss abgewiesen werden, wenn der Empfängerserver erkennen kann, dass er das Subject nicht im Originalzustand wieder herstellen kann.

Das letzte Problem ist wiederum ein API-Problem: Ältere Methoden der Access-Controller-Klasse sind nicht in der Lage, ein Subject zusätzlich zur Action-

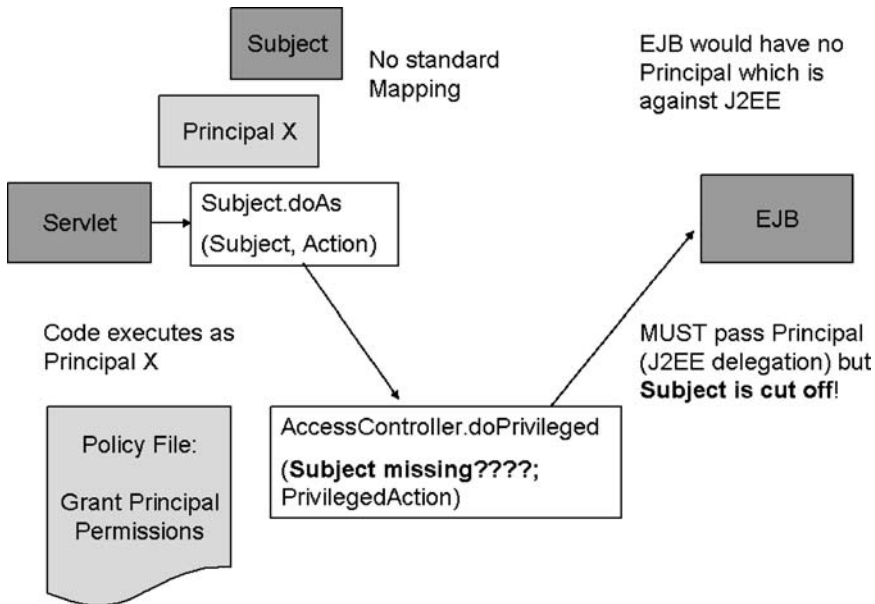


Abb. 7.19 Abschneiden des Delegations-Principals

Klasse aufzunehmen. Damit wird die Laufzeitidentität des Aufrufers abgeschnitten und kann insbesondere nicht an weitere JAVA-EE-Komponenten weitergereicht werden (s. Abb. 7.19). JAVA EE fordert die Übergabe des Principals jedoch im Standard.

Auch hier greifen wieder herstellerspezifische Mechanismen, wie besondere Formen von Subject (WSSubject im Websphere-Fall) oder doppelte Call-Stacks bei Weblogic.

7.6 Absichern von Application-Servern („Hardening“)

In diesem Abschnitt wollen wir einige Sicherheitsprobleme von Application-Servern ansprechen und wie man sich dagegen schützen kann. Je nach Application-Server fallen Probleme wie Hilfen natürlich unterschiedlich aus, und so ist der Fokus hier nicht auf der konkreten Konfiguration der Application-Server – dafür ist die herstellerepezifische Dokumentation unerlässlich – sondern wir wollen allgemeine Regeln in Bezug auf Topologie, Ressourcen, Verbindungen, Isolation etc. und ihrer Absicherung aufstellen. Daraus können für ein Projekt Checklisten für Architektur und Design der Applikation während der Entwicklung erstellt werden sowie für die Sicherheitsprüfungen des Ergebnisses vor dem Deployment.

Es zeigt sich dabei an vielen Stellen, dass die Application-Server noch längst nicht die Sicherheit und den Isolationslevel von Betriebssystemen erreicht haben. Im Folgenden haben wir vor allem die Frage der Konsistenz und Korrektheit von

Applikationen auf Application-Servern im Blickpunkt. Die Antworten würden wahrscheinlich noch kritischer ausfallen, wenn man den „Liveness“-Aspekt mit hinzunehmen würde: Ist sichergestellt, dass eine Applikation nicht nur korrekt arbeitet, sondern in ihrer Arbeit auch Fortschritte macht, also tatsächlich verfügbar und performant ist und Anforderungen von Clients erledigt? Diese Frage zielt auf das Problem der Verwaltung gemeinsamer Ressourcen: Kann eine Applikation im Fehlerfall andere Applikationen im gleichen Application-Server behindern? Gemeint ist hier nicht die bösartige Attacke, sondern schlicht der Umgang mit gemeinsamen Ressourcen wie CPU, File-System, Netzwerk etc. Die Java-Laufzeitumgebungen sind noch längst nicht in der Lage, hier eine faire Gleichbehandlung der Applikationen sicherzustellen.

7.6.1 *Auslieferungszustand vs. Default-Is-Deny-Prinzip*

Die Installation und Konfiguration von Application-Servern beinhaltet eine Vielzahl von Files, Prozessen und Konfigurationsmaßnahmen. Eine sichere Konfiguration steht dabei klar im Konflikt mit dem Wunsch der Hersteller nach „download and run“ ihrer Software. Nach dem Motto, dass der erste Eindruck der Wichtigste ist, sollen Entwickler nicht sofort mit der zusätzlichen Komplexität von Sicherheitsmaßnahmen und ihrem unter Umständen negativen Einfluss auf die einfache Benutzbarkeit konfrontiert werden. In der Konsequenz wird die Software entgegen dem Prinzip „default-is-deny“ die Software grundsätzlich in einem sehr offenen Zustand ausgeliefert. Ein „Härtungsschritt“ ist daher in einer ernsthaften Entwicklungsumgebung unumgänglich (ja, auch die Entwicklungsumgebung sollte die reale Situation in der Produktion weitestgehend wiedergeben).

Die folgenden Anmerkungen stützen sich im Wesentlichen auf die von Keys Botzum in [Botz06] genannten Sicherheitsprobleme:

- ungeschützter File-Transfer für die Installation und Wartung von Applikationen und Servern,
- ungeschützte Ressourcen wie Queues, Directories und Agents,
- Verwendung unsicherer Protokolle,
- Sicherheit generell ausgeschaltet,
- fehlende Zertifikate und
- ungeschützte Schlüssel.

Ein wichtiger Teilaspekt der Konfiguration ist die Kenntnis aller Artefakte des Application-Servers. Dazu gehören statische Artefakte wie Konfigurations-Files, Credentials, Bibliotheken und Verzeichnisse wie auch dynamische Artefakte in Form von Prozessen, Agenten, Services, Backend-Anbindungen etc. Zu diesen Artefakten sind die folgenden Regeln wichtig:

- Alle dynamischen Artefakte (Prozesse, Agenten) müssen bekannt sein. Welche Rolle innerhalb des Applikation-Servers haben sie? Wo laufen sie? Welche Rechte besitzen sie?

- Alle statischen Artefakte (Konfigurationsdateien, Keys) müssen über Betriebssystemmittel geschützt werden.
- Demo-Code ist vor Auslieferung zu entfernen.
- Default Keys und Passwörter etc. müssen vor Auslieferung angepasst werden.

Von besonderer Bedeutung ist natürlich der Umgang mit Credentials. Für jedes Credential (Passwort, symmetrischer/asymmetrischer Key, Rolle) im Gesamtsystem (Zelle mit Application-Server, Web-Server, Plug-Ins, Datenbanken und Verzeichnissen) müssen als Minimum die folgenden Fragen positiv beantwortet werden können:

- Sind die Zertifikate im Trust Store gültig?
- Sind nur die minimal nötigen Zertifikate im Trust Store? Besonders kritisch sind Root-Zertifikate verschiedener Signierer, die z. B. die Ausstellung von verschiedenen Zertifikaten pro Name möglich machen.
- Gibt es für jedes Credential einen Prozess der Erneuerung bzw. des Rückrufes?
- Wird bei Servern die Übereinstimmung zwischen Namen im Zertifikat und dem Servernamen geprüft?
- Wie sind die Schlüssel gespeichert bzw. abgesichert?
- Gibt es einen sicheren Mechanismus, regelmäßig symmetrische Schlüssel zu verteilen?
- Wurden alle Default-Credentials durch neue ersetzt?
- Wird das Ablaufen von Credentials administrativ überwacht?

Die Erneuerung von abgelaufenen Passwörtern oder Zertifikaten sowie deren Verteilung ist eine wichtige administrative Aufgabe, die von den Application-Server-Herstellern nur sehr unvollkommen unterstützt wird. So gestaltet sich allein der Austausch von Passwörtern in Firmen mit Password Aging Policies als sehr schwierig. Eventuell kann es Phasen während des Update der Credentials geben, in denen besser keines der beteiligten Systeme abstürzen sollte, da nach dem Restart die alten Credentials vielleicht schon deaktiviert, die neuen aber noch nicht aktiviert sind.

Durch geeignete Konfiguration können viele dieser Probleme beseitigt werden. Manche sind jedoch auch tief in der Architektur der Application-Server angesiedelt.

7.6.2 Absicherung der Verbindungen

In [Botz05a] wird eine ganze Reihe von teils sehr überraschenden Sicherheitsproblemen bei der Absicherung von Zugriffen auf Services erwähnt. So ist der modifizierende Zugriff auf globale Verzeichnisse ohne Zugangskontrolle möglich. Teilweise sind Services global für alle authentisierten User offen. Aus den dort geschilderten Schwächen lässt sich die folgende allgemeine Regel aufstellen:

Für jede Verbindung im gesamten System (Cell- oder Application-Server mit Backends, Web-Server, Directories, SSO-Server etc.) sollte gelten:

- Die Transportqualität (Vertraulichkeit und Integrität) des Protokolls muss festgelegt werden.
- Es muss geprüft werden, ob auf Seiten des Services Authentisierung eingeschaltet ist.
- Es muss geprüft werden, ob eine weitere Autorisierung des Requests durchgeführt wird, oder ob die bloße Authentisierung das Recht zum Zugriff impliziert. Dieser Punkt ist besonders im Zusammenhang mit Zertifikaten in Trust-Stores zu prüfen.
- Die Verbindungen sind auf die existierenden Sicherheitszonen abzubilden und der Ort für die Services ist zu bestimmen.

7.6.3 Absicherung der Ressourcen

Wem „gehören“ eigentlich Ressourcen wie DataStores oder Queues in JAVA-EE-Architekturen? Solche Ressourcen werden von Application-Servern anhand von Deployment-Deskriptoren erstellt und über Remote Objects in globalen Verzeichnissen zur Verfügung gestellt. Das bedeutet, dass Applikationen innerhalb eines Servers auf die Ressourcen anderer Applikationen zugreifen können, wenn keine weitere Autorisierung mehr stattfindet. Je nach Hersteller sind die Methoden der Absicherung ganz unterschiedlich. Entscheidend ist, dass die Ressourcen so konfiguriert sind, dass nur berechnete Applikationen zugreifen können. Globale Sichtbarkeit darf nicht globale, uneingeschränkte Verwendung bedeuten.

7.6.4 Verwaltung des Runtime-Systems

Die Komplexität der Laufzeitumgebung von Application-Servern mit ihren vielen Services und Anbindungen an andere Systeme macht eine zentrale Verwaltung zu einem wertvollen Produktivitätstool. Installation und Konfiguration von Applikationen, Anbindung an Backend-Systeme, Integration mit vorgelagerten Web-Servern und Proxies, globale Verzeichnisse und Caches müssen alle durch Systemmanagementwerkzeuge erledigt werden.

Als Folge besitzen die Application-Server spezielle APIs zur Administration ganzer Verbände von Servern, im Websphere-Jargon „Cells“ genannt. Sicherheitstechnisch haben diese Cells sowohl positive als auch kritische Eigenschaften:

- Das zentrale Deployment von Applikationen und Schlüsseln sichert Konsistenz und Korrektheit und vereinfacht die regelmäßige Erneuerung von Schlüsselmaterial.
- Agenten auf jedem System erlauben zentrales Tracking, Logging und Auditing der gesamten Infrastruktur.

- Systemmanagement APIs und Bibliotheken sind auf allen Servern vorhanden und können von Applikationen benutzt werden.
- Die Systemmanagementschlüssel zur Authentisierung und Autorisierung sind auf allen Maschinen vorhanden.
- Die Kompromittierung einer Maschine der Cell genügt, um die gesamte Konfiguration der Cell ändern zu können.

Die Konsequenz dieser Eigenschaften ist, dass eine Cell als Zone absoluten Vertrauens aufgefasst werden muss. Eine Applikation auf einem beliebigen Application-Server könnte – wenn sie im Besitz des Schlüssels ist – jede beliebige Konfiguration innerhalb der Cell ändern.

7.6.5 Isolation

Das Isolieren von Komponenten voneinander gehört zu den wichtigsten Entwurfsmustern sicherer Software. Mehrere Eigenschaften der modernen Application-Server machen dies jedoch schwierig. Abbildung 7.20 zeigt zwei Application-Server in einer Zelle, zusammen mit ihren sicherheitsrelevanten Ressourcen und Schnittstellen.

Ein Blick auf Abb. 7.20 zeigt, dass die Erreichbarkeit der sicherheitsrelevanten Ressourcen innerhalb einer Zelle sehr hoch ist: Applikationen haben Zugriff auf

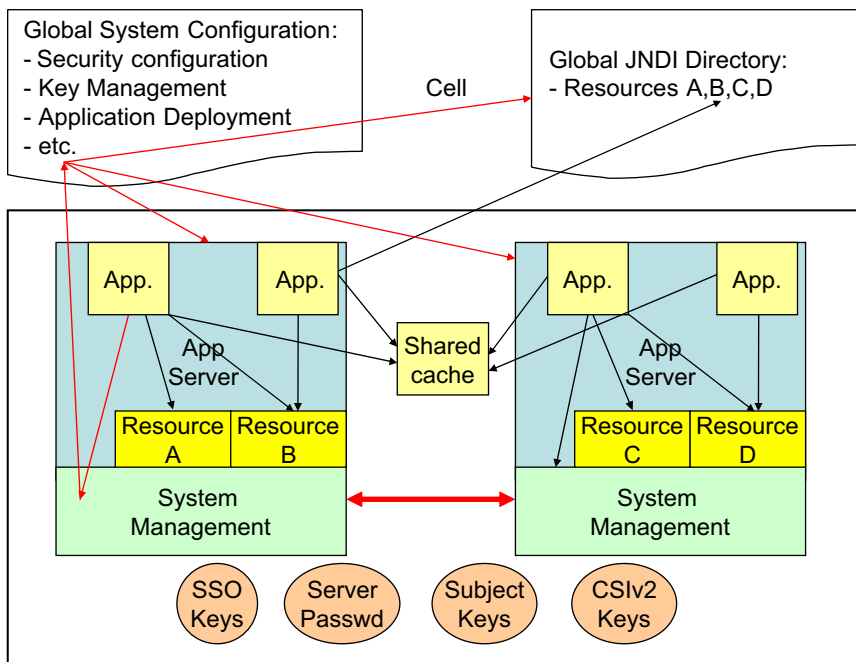


Abb. 7.20 Sicherheitsrelevante Daten zweier Application-Server einer Zelle

Ressourcen anderer Applikationen und globale Verzeichnisse bieten ebenfalls die Möglichkeit, nach Ressourcen zu suchen. Selbst wenn gewisse Zugriffe durch geeignete Konfiguration unterbunden werden, bleibt immer noch die Möglichkeit, durch Änderungen über das Systemmanagement diese Einschränkungen wieder aufzuheben. Dazu gehört auch die Änderung von Java-2-Security-Konfigurationen für Applikationen. Damit stehen dann zentrale Verwaltungs-APIs zur Verfügung.

Ein weiteres Problem für Applikationsarchitekten ist die Vielzahl von Ressourcen und Services, die den Applikationen zur Verfügung stehen. Welches Vertrauen sollte z. B. ein Cache Service erhalten? Wird er geteilt mit anderen Applikationen? Wo überall können Applikationen miteinander interagieren – und ist dies beabsichtigt oder unbeabsichtigt? Besonders problematisch ist die Zeitachse bei solchen Abschätzungen von Gefahren: Kann vielleicht durch nachträgliche Installation von weiteren Applikationen oder durch Änderungen der Systemkonfiguration aus einer sicheren Applikation eine unsichere werden?

Diese Frage ist besonders für Architekten interessant, die eine globale Hosting-Plattform für JAVA-EE-Applikationen auf Basis eines Clusters von Application-Servern zur Verfügung stellen wollen: Soll der ganze Cluster eine Cell darstellen oder sollen besonders kritische Applikationen in eigenen Cells untergebracht werden? Botzum erwähnt auch die Möglichkeit, eine eigene Cell zur Isolation von Applikationen zu erzeugen:

- ein Application-Server installiert unter einer eigenen Betriebssystemidentität
- eigene Systemverwaltungspasswörter und
- eigene Schlüssel für Verschlüsselung (benutzt für Validierung von Identity Assertions).

Betrachtet man diese Anweisungen zur Isolation, merkt man, dass sie hauptsächlich auf die Selbstverwaltung des Application-Servers zielen: Das Betriebssystem kennt weder die Passwörter der Systemverwaltung einer Cell oder eines Application-Servers, noch ist es mit den Schlüsseln befasst, die der Server für die Feststellung von Trust benötigt. Einzig die Betriebssystemidentität wird genutzt, um zu verhindern, dass andere Cells per File-Zugriff die Konfiguration ändern können. Alle anderen Zugriffe über Netzwerkverbindungen müssen ohnehin innerhalb der Cell authentisiert werden, wobei das Betriebssystem nicht beteiligt ist.

Ein Application-Server bzw. eine Cell, bestehend aus mehreren Servern, bildet demnach eine logische Hülle über physische Maschinen und Betriebssysteme. Die Möglichkeiten der internen Isolation von Cells durch physische Trennung von Hosts, Prozessgrenzen oder unterschiedliche Betriebssystemidentitäten sind somit stark eingeschränkt.

Konkret bedeutet das, dass die Sicherheit einer Applikation oder eines Application-Servers stärker von den anderen Servern und Applikationen einer Cell abhängt als von der direkten Laufzeitumgebung. Das wird schnell verständlich, wenn man erkennt, dass die meisten Verbindungen der Applikationen und Server untereinander über Netzwerkverbindungen funktionieren, deren Absicherung über Schlüssel und Rechte geschieht, die wiederum innerhalb dieser Cell verwaltet wer-

den. Somit sind die Credentials in Form von Schlüsseln, Passwörtern und Rechten entscheidend für die Sicherheit einer Cell.

7.6.6 *Cross-Cell-Trust*

Wir betrachten nun die Security-Aspekte der logischen Verbindung von Zellen, die sich gegenseitig vertrauen. Wir haben die entscheidenden Kriterien für die Isolation von Cells bereits erörtert: Schlüssel für SSO, Subject Propagation und Delegation über CSIv2. Es ergeben sich jedoch einige überraschende Konsequenzen aus der Tatsache, dass sich zwei Cells in gewissem Maße vertrauen, die letztlich dazu führen können, dass Applikationen der einen Cell Ressourcen der anderen missbrauchen bzw. sogar die Systemkonfiguration ändern können.

- **Verwendung gleicher SSO-Schlüssel:**
Um einem SSO-Verbund anzugehören, müssen beide Cells den gleichen Schlüssel für die SSO-Information (im Websphere-Jargon LTPA genannt) verwenden. Interessant daran ist nur die Konsequenz: Beide Cells können mit diesem Schlüssel anschließend authentifizierte Identitäten erzeugen und damit Services der anderen Cell aufrufen bzw. den Caller an diese Services delegieren. Ein Sicherheitskompromiss in der einen Cell kann sich so direkt auf die andere Cell auswirken.
- **Subject Propagation:**
Eine Cell verwendet aufgefüllte Subject Objects einer anderen Cell. Dazu wird nach der Übertragung der Identitätsinformation ein Rückruf des empfangenden Servers an den Sender in der anderen Cell mit der Bitte um Übertragung der Subject Information ausgelöst. Dazu muss der anfragende Server sich jedoch mit seinem Administrationspasswort authentisieren. Gleichzeitig wird er auf der Empfängerseite als zusätzlicher Cell-Administrator eingetragen, um Zugriff auf die Subject-Daten zu erhalten. Vom großen Vertrauen in die Subject-Daten einmal abgesehen, das in der empfangenden Cell herrschen muss, sind jetzt beide Cells auch administrativ völlig gekoppelt und könnten gegenseitig beliebige Änderungen der Konfiguration erstellen.
- **Cross-Cell-RMI/IIOP über CSIv2-Absicherung**
Kennzeichen der CSIv2-Absicherung ist die gegenseitige Authentisierung der beteiligten Server. Wenn ein Sender hierfür sein (Admin-)Passwort verwendet, wird er von allen Empfängern abhängig: Sie könnten z. B. seine User-ID und sein Passwort gegenüber Dritten missbrauchen. In diesem Fall sollte unbedingt eine gegenseitige Authentisierung über Zertifikate bzw. digitale Signaturen durchgeführt werden.

Abgesehen von den nur zum Teil überraschenden Abhängigkeiten zwischen Cells zeigt das Thema Cross-Cell-Trust nur erneut die Problematik von gemeinsamen symmetrischen Schlüsseln in verteilten Umgebungen auf.

7.6.7 Code Access Control in Application-Servern

Dieser und der nächste Abschnitt beschäftigen sich mit der feingranularen Zugriffskontrolle auf Code. Im JAVA-EE-Bereich wird dies realisiert durch die Java 2 Security (in Form von Policy Files, die Permissions und Codebereiche verbinden) und im .NET-Bereich durch ein ausgefeiltes Permission-System. Der Java-Mechanismus wurde bereits behandelt, daher betrachten wir in Abschn. 7.6.8 nur noch die Konsequenzen bei .NET-basierten Servern, die „fully trusted“ – also ohne Einschränkung – laufen.

Wir haben gerade festgestellt, dass durch die Verwendung gemeinsamer Credentials und eines gemeinsamen, zentralen Systemmanagements das Vertrauen in alle Komponenten einer Cell absolut sein muss. Abgesehen von der dadurch entstehenden grundsätzlichen Gefahr kann aber durch Einschränkung der Rechte von Applikationscode die Isolation verstärkt, bzw. gefährlicher Code in seiner Wirkung eingeschränkt werden. Botzum erwähnt das Beispiel einer Applikation, die MBeans, also Verwaltungsobjekte, benötigt. Diese müssen mit erhöhten Privilegien ausgeführt werden. Diese Managementobjekte können ihre Rechte auf unterschiedliche Weise erhalten:

- Als Teil der gesamten Applikation. Dadurch erhält die Applikation insgesamt höhere Rechte.
- Als separates Modul mit eigenen Permissions ausgestattet, getrennt von der Applikation (dies ist bei der Architektur der Applikation zu berücksichtigen).
- Als Bibliothek, die von einem hohen Classloader im System geladen wird (z. B. dem Extension Classloader), der die geladenen Module mit zusätzlichen Permissions versieht. Das Kopieren von Bibliotheken in den ClassPath von System-Classloadern ist zwar ein beliebtes Mittel, um sowohl die Erreichbarkeit der Klassen als auch ihre Privilegien sicherzustellen, nur leider schießt der Mechanismus meist über das Ziel hinaus: Zu viele Applikationen bekommen Zugriff auf diese Klassen und ihre Rechte sind oft zu mächtig.

Die erste Lösung verstößt eindeutig gegen das POLA-Prinzip, indem sie die Applikation insgesamt mit zu vielen Rechten ausstattet. Die zweite Lösung ist zwar feingranular, muss aber frühzeitig in der Applikationsarchitektur sowie im Deployment berücksichtigt werden.

Die dritte Lösung ist ebenfalls ein Verstoß gegen POLA, macht aber ein grundsätzliches Problem deutlich: Je höher Code in der Hierarchie der Classloader angesiedelt wird, umso sichtbarer und erreichbarer ist er von unteren Schichten der Hierarchie aus gesehen, also von Applikationen. Nun ist oft gerade auch der Code des Application-Servers durch höhere Classloader geladen. Oft sollen Applikationen jedoch diese Klassen nicht verwenden, und zwar:

- Aus Sicherheitsgründen, wenn keine Java 2 Security eingesetzt wird, und um zu verhindern, dass interne Application-Server APIs von Applikationen z. B. zum Systemmanagement verwendet werden.
- Aus Wartungsgründen: Änderungen der Serverklassen brechen sonst die Applikationen der Kunden.

Hier bietet die Verwendung des Servicemodells der Opens-Services-Gateway-Initiative (OSGI) neuerdings die Möglichkeit, über getrennte Namespaces Applikationen voneinander und auch von den Serverklassen der Infrastruktur zu trennen. Das genaue Prinzip der Peer-Classloader wird in Kap. 8 behandelt. Auch das neueste Websphere Release bedient sich der OSGI-Classloader, um Applikationen und Serverklassen voneinander zu trennen.

Als Regel für sichere Applikationen kann man daher festhalten:

- Die Positionierung von Bibliotheken darf nicht nur unter dem Gesichtspunkt der einfachen Erreichbarkeit möglichst hoch im Class-Path des Systems durchgeführt werden.
- Beim Deployment mehrerer Applikationen auf einem Application-Server ist jede Applikation in Bezug auf ihre Packages zu überwachen.
- Kritische Funktionen müssen in eigene Packages ausgelagert werden, damit sie separate Rechte erhalten können.
- Manipulationen der Applikationen am Classloader-Verhalten (Suchstrategie) sind zu kontrollieren.

7.6.8 „Fully Trusted“ .NET

Wir haben bereits grundlegende Serverarchitekturen und ihre Probleme kennen gelernt. Leider ist der normale Application-Server die Architekturform, die im Falle eines Angriffs oder Fehlers auch die weitestreichenden Konsequenzen nach sich zieht. Das ergibt sich aus der Eigenschaft des Application-Servers, sowohl verschiedene Applikationen als auch verschiedene User und ihre Requests innerhalb eines Prozesskontexts zu bearbeiten (s. Abb. 7.21).

Entsprechend kritisch ist die Frage nach den Rechten des Applikation-Servers. Im Java-Lager ist die Antwort darauf meist: Mit Root-Rechten, aber ohne Java 2 Security (wegen der Performance). Im .NET-Lager heißt es ebenso häufig: Der Server läuft „fully trusted“.

Dinis Cruz hat in der Vergangenheit die Konsequenzen daraus im Detail untersucht (s. [Cruz]): Am Beispiel von Internet-Service-Providern (ISP) spielt Cruz die Möglichkeiten von .NET-Servern durch, wenn sie im Modus „Fully Trusted“ laufen. ISP-Umgebungen sind auf maximale Isolation der Kunden von einander angewiesen. Diese Kunden besitzen ja unter Umständen bereits selber wieder Kunden, an die sie Services weitergeben. Die Kunden dürfen daher keinesfalls Zugriff auf globale Administrationsfunktionen erhalten.

Cruz nennt unter anderem folgende Angriffsmöglichkeiten, die aus dem „Fully-Trusted“-Modus resultieren: `RevertToSelf` ist eine spezielle .NET-Funktion, mit der ein Stück Code, das per Impersonifizierung gestartet wurde (also mit den Rechten eines bestimmten Users ausgeführt werden soll), wieder in die ursprünglichen Rechte des Servers zurückkehren kann. So ein Stück Code kann über diese Funktion also ihre eigentlichen Rechte erheblich ausdehnen. Darüber hinaus stehen mächtige Meta-Daten-Funktionen ebenso zur Verfügung, wie die Möglichkeit

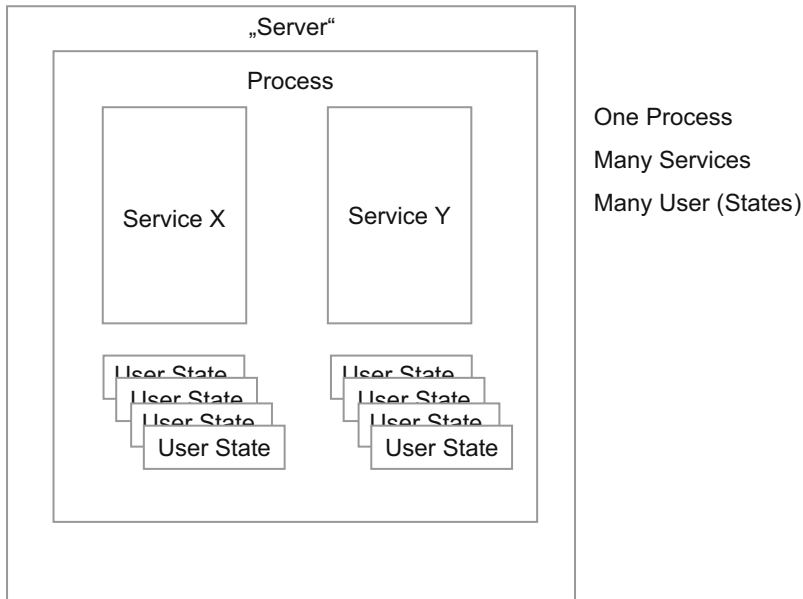


Abb. 7.21 Ein Prozesskontext beherbergt mehrere Dienste und mehrere Nutzer

ungesicherten (unmanaged) Code auszuführen, per Reflection in Packages hineinzugreifen, sowie Security Tokens beliebig manipulieren zu können. Dadurch können z. B. auf dem Call Stack des Threads beliebige Identitäten abgelegt werden, unter denen der Code dann laufen soll. Die gesamte Code Access Security kann ebenfalls umgangen werden.

Bleibt die Frage zu klären, warum diese Server alle „Fully Trusted“ laufen? Die Antwort liegt vor allem darin, dass im eingeschränkten Modus etliche wichtige Funktionen fehlen, wie etwa der Aufruf von ungesichertem Code oder das Erzeugen von COM-Objekten. Die Vorträge von Cruz anlässlich der OWASP-Konferenzen sind eine Fundgrube für jeden .NET-Entwickler der im Serverbereich tätig ist.

7.6.9 Isolierende VMs und Application-Server: SAP

Von der Firma SAP kommt ein interessantes Design für eine Prozessarchitektur, die versucht, mithilfe von Memory-Management-Techniken größere Isolation zu erreichen (s. [Kuck]): Process Attachable Virtual Machines (PAVM) sowie Trennung des User-Kontexts von VM-Daten durch Shared-Memory-Techniken. Beide Techniken sind auch in [Blüh] beschrieben. Das Ziel ist es, sowohl User bzw. Requests voreinander zu schützen, als auch die VM als Ganzes vor Requests zu schützen, die im Extremfall zum Absturz der ganzen VM führen können. Damit

wären alle User und ihre Daten betroffen. Der Idealfall – jeder User erhält eine eigene Virtual Machine – ist leider aus Ressourcengründen nicht durchführbar.

Als Kompromiss bleibt die Möglichkeit, dass zu jedem Zeitpunkt innerhalb einer Virtual Machine lediglich ein User durch genau einen Service bearbeitet wird (s. Abb. 7.22). Damit gleicht die Architektur eigentlich unserem Eingangsbeispiel im Kap. 4: Ein Prozess pro User. Dort hatten wir gesagt, dass dies aufgrund der vielen und lange laufenden Prozesse nicht skaliert. Die SAP-Architektur

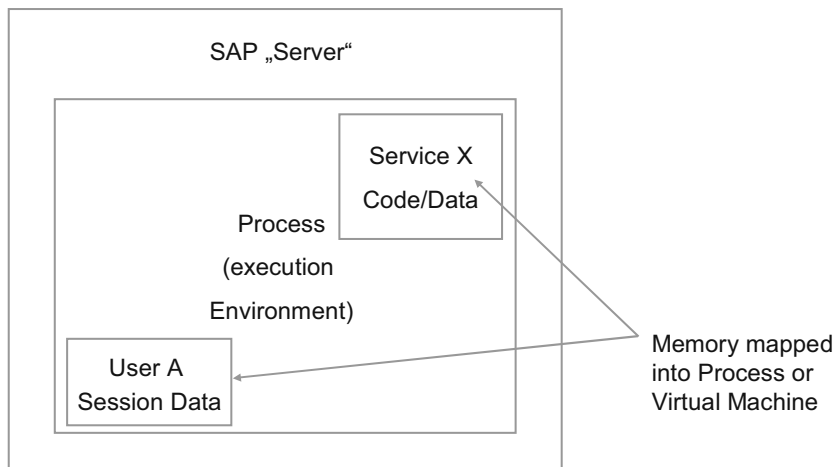


Abb. 7.22 Ein Service pro User in einer Virtual Machine

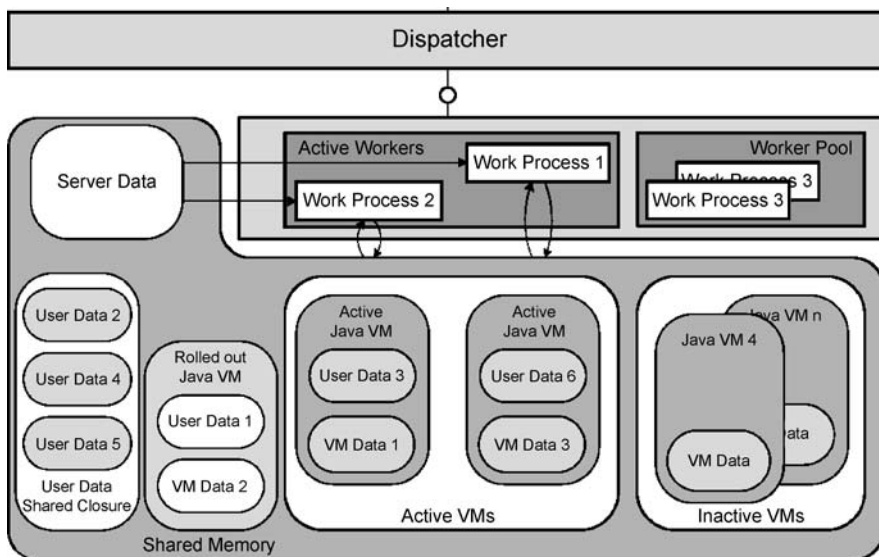


Abb. 7.23 Trennung von VM und Nutzerdaten in der SAP-Architektur

kommt jedoch mit einer weit geringeren Zahl von Prozessen aus, da der spezifische Kontext eines Requests, also Application Code und User State, jeweils bei Beginn einer Verarbeitung auf Anforderung in den Prozess abgebildet wird – ähnlich der manuellen Behandlung des Request-States in eventgetriebenen Architekturen.

Die folgende Abb. 7.23 (aus [Blüh]) zeigt die daraus resultierende Trennung von VM und User-Daten sowie deren Mapping über Shared Memory. Der SAP-Ansatz verbindet somit Skalierfähigkeit mit maximaler Isolierung der Komponenten.

Der Ansatz von SAP mag im ersten Moment übertrieben erscheinen, jedoch werden immer wieder Fälle gemeldet, in denen User die Daten anderer User lesen oder sogar kompromittieren konnten, siehe dazu auch [heise76736].

7.6.10 *User Identity Propagation zu Datenbanken*

Sichere Software folgt dem Prinzip von POLA und gestattet nur die minimal nötigen Rechte pro User. An der Schnittstelle zu Datenbanken wird dieses Prinzip jedoch von vielen Application-Servern verletzt. Die Ursache dafür ist der sogenannte *Connection Pool*: Eine Sammlung von Datenbank-Connections zur dynamischen und schnellen Verwendung durch einen Request. Eine einzelne nutzerspezifische Connection hat die folgenden Eigenschaften:

- Sie ist teuer und langsam zu erzeugen.
- Die User-Credentials müssen bei der Erzeugung angegeben werden.
- Autorisierung und Auditing werden auf Seiten der Datenbank mit dem angegebenen User-Principal durchgeführt.

In vielen Fällen erzeugt der Application-Server den Connection Pool mithilfe einer konfigurierten Identität (z. B. dem `DBAccessPrincipal`) und gespeicherten Credentials. Das hat die Konsequenz, dass alle Datenbankzugriffe unter dieser künstlichen Identität stattfinden und die granulare Access Control der Datenbank die Zugriffe nicht mehr für den Endnutzer individuell kontrollieren kann (s. Abb. 7.24).

Unter der Voraussetzung, dass die Datenbank bzw. der Ressource-Adapter (ein Plug-In des Datenbankherstellers für Application-Server zum Anschluss der Datenbank) die sogenannte „connection pool:Re-Authentication“ erlauben, kann ein Application-Server aber auch den echten Caller-Principal zur Datenbank propagieren. Eine wesentliche zweite Voraussetzung dabei ist allerdings, dass das Austauschen des User-Principals für die Connection wesentlich schneller vonstatten geht als das Anlegen der Connection. Abbildung 7.25 zeigt diesen Vorgang.

Der Application-Server muss dazu über datenbankspezifische Methoden den Principal der Datenbank austauschen können. Wie das geschieht, ist abhängig von der Datenbank. So erlauben manche den Wechsel des Principals ohne weitere Credentials (z. B. ein Passwort), während andere solche immer voraussetzen.

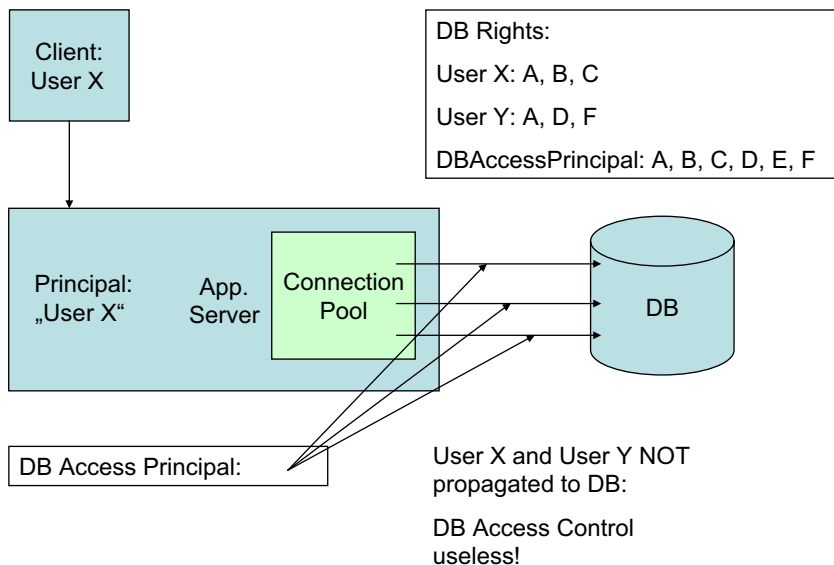


Abb. 7.24 Datenbankzugriff durch den Server unter einer künstlichen Identität

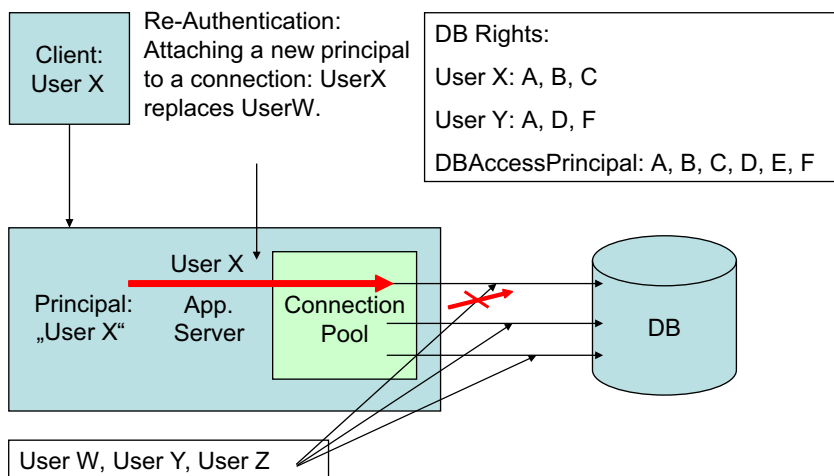


Abb. 7.25 Austauschen des Principals für eine bestehenden Verbindung

Der Wechsel des Principals einer Connection kann übrigens auch durch das generelle Mapping auf eine applikationsspezifische Identität geschehen: Es wird dann nicht auf den individuellen Principal des Aufrufers gewechselt, sondern auf eine bestimmte Datenbankidentität speziell für eine Applikation. Alle Connections, die diese Applikation verwendet, werden dann unter dieser Identität ausgeführt.

Application-Server-Hersteller sind natürlich bemüht, solche Mechanismen auf eine allgemeine Weise zur Verfügung zu stellen. Dies geschieht, wie üblich, durch das Konzept von Interceptoren, die als Plug-Ins der Applikation erlauben, das Verhalten des Servers abzuändern, ohne dass dafür der Code des Application-Servers oder der Applikation selbst geändert werden müsste. Keys Botzum et. al beschreiben eine entsprechende Lösung für den Websphere-Application-Server in [Botz05a] (s. Abb. 7.26).

Die Lösung verwendet zwei applikationsspezifische Erweiterungen: Eine JAAS-LoginModule-Konfiguration mit einem speziellen Modul sowie einem DataStoreHelper-Objekt, das die tatsächliche, datenbankspezifische Re-Authentisierung durchführt. In einem ersten Schritt wird die J2C-Engine ein Subject erzeugen. Das spezielle LoginModule legt darin ein Principal-Information-Objekt an, über das die UserIdentität an das spätere DataStoreHelper-Objekt transportiert wird. Woher die User-Identität vom LoginModule genommen wird, bleibt der Applikation überlassen und ist anpassbar. Danach prüft die J2C-Engine, ob für den gewünschten User bereits eine Connection im Pool angelegt wurde. Falls nicht (und der Pool kann noch wachsen), wird eine neue Connection mit den Credentials des Users angelegt. Falls der Pool seine maximale Größe erreicht hat, muss eine existierende Connection re-authentisiert werden, d.h. sie muss den Principal wechseln. Dafür ruft die J2C-Engine den Ressource-Adapter auf und übergibt Subject- sowie Connection-Information. Der Ressource-Adapter ruft das DataStoreHelper-Objekt und dieses führt die tatsächliche Re-Authentisierung auf der Connection durch.

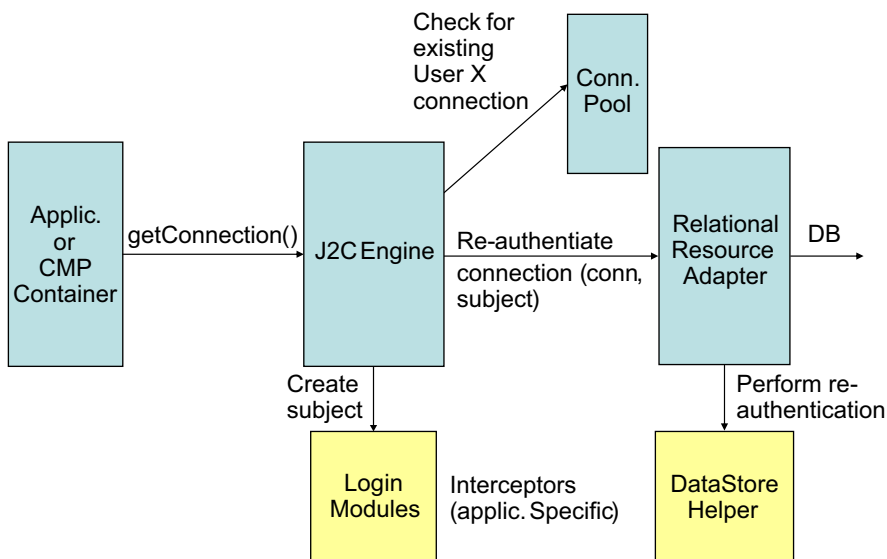


Abb. 7.26 Interceptor-Lösung für den IBM-Websphere-Application-Server

Bei dieser Lösung ergeben sich einige spezielle Probleme:

- Beim Anlegen einer neuen Connection geht die Lösung davon aus, dass der Thread der Caller-Thread ist, d. h., dass der Caller Principal verfügbar ist.
- Manchmal muss ein Passwort des Callers zur Re-Authentisierung verfügbar sein. Dies bricht eine saubere Delegation des Aufrufs und erzwingt Zugriff auf Passwort-Stores.
- Wenn kein Authentisierungs-Credential beim Wechsel des Principals verlangt wird, dann muss die Infrastruktur verhindern, dass die Datenbank über andere Verbindungen erreicht werden kann – sonst könnten andere Applikationen diese Sicherheitslücke ausnutzen.

Der Lösungsvorschlag ist zweifellos realisierbar, birgt aber, wie gesehen, auch einige Sicherheitsprobleme in sich, da plötzlich wieder User-Passwörter nötig sind bzw. Identitäten ohne dazu gehörige Credentials gewechselt werden können. Auf Seiten der Datenbank wären deshalb bessere Interfaces nötig: Die Datenbank sollte bessere Formen der Delegation unterstützen, z. B. die tokenbasierte Identity-Assertion. Auch die Angabe mehrere Principals sollte im Sinne von Traced-Delegation möglich sein.

Abgesehen von dieser speziellen Lösung des Problems existieren im Prinzip auch noch andere Möglichkeiten: Es sollte z. B. möglich sein, die Verbindungen zunächst mithilfe eines Functional Users zu erzeugen und dann anschließend die Identität auf den Principal eines Callers zu wechseln. Das Recht zum Wechsel könnte die Datenbank an die Connection binden sowie zusätzlich an den Principal, der die Connection ursprünglich erzeugt hat. Der „DBAccessPrincipal“ hätte dann das Recht, Connections zur Datenbank zu erzeugen, die dann die Identität des Principals austauschen dürfen. Andere Connections hätten dieses Recht nicht. Auf Datenbankseite könnte diese Funktion durch das Verhalten des Connection-Stubs sichergestellt werden. Der Stub verhielte sich hier wie eine Capability zum Wechsel der Identität. Das Rechtssystem der DB selbst wäre nur beim Anlegen der Connection zu konsultieren. In Kap. 10 werden wir diesen Ansatz der Erzeugung von Sicherheit durch sogenannte Sicherheitsbausteine weiter verfolgen.

7.7 Zusammenfassung

Die Menge der nötigen Sicherheitsdetails und die Komplexität der Application-Server-Architekturen sind erschreckend. Ohne intime Kenntnisse des jeweiligen Produktes und seines Umgangs mit Schlüsseln etc. ist eine Absicherung nicht möglich.

Isolierte Zonen mit bestimmten Sicherheitsqualitäten lassen sich kaum bilden, da durch die Verwendung globaler Verzeichnisse und die Kopplung über System Management Services oder Application-Server APIs fast alle Ressourcen für Applikationen offen stehen.

In eng kontrollierten Entwicklungsumgebungen ist dies kaum ein Problem. Allerdings trifft der Vorschlag von Botzum, den Code kritischer Applikationen durch

Peer-Review Zeile für Zeile prüfen zu lassen, nicht mehr die Realität heutiger Entwicklungsteams: Dort gibt es meist keine zwei Entwickler mehr, die die gleichen Kenntnisse für das jeweilige Stück Source-Code mitbringen würden: Es gibt in den meisten Projekten keinen Raum mehr für Doppelbesetzungen oder Backup-Personal.

Literatur

- [Blüh] A. Blüher, SAP VM Container, Seminar System Modellierung 2005, Hasso-Plattner-Institute for Software Systems Engineering, http://wendtstud1.hpi.uni-potsdam.de/sysmod-seminar/SS2005/elaborations/04-SAP_VM_Container.pdf
- [Botz05a] K. Botzum, V. Jessani, S. Barghouti, IBM WebSphere Developer Technical Journal: Database identity propagation in WebSphere Application Server V6, http://www.ibm.com/developerworks/websphere/techjournal/0506_barghouthi/0506_barghouthi.html
- [Botz05b] K. Botzum, B. Hines, P. Ilichko et al., Advanced authentication in WebSphere Application Server, http://www-128.ibm.com/developerworks/websphere/techjournal/0508_benantar/0508_benantar.html
- [Botz06] K. Botzum, WebSphere Application Server V6 advanced security hardening – Part 1 and 2, http://www-128.ibm.com/developerworks/websphere/techjournal/0512_botzum/0512_botzum1.html
- [Chugh] A. Chugh, J. Mountjoy, Weblogic. The definitive Guide. Kapitel 17 “Java and Security” über Authentisierung in Weblogic erhältlich unter http://www.onjava.com/pub/a/onjava/excerpt/weblogic_chap17/index.html
- [Cruz] D. Cruz, .NET Full Trust, http://www.owasp.org/index.php/.Net_Full_Trust
- [Flan] D. Flanagan, Java Examples in a Nutshell, Source Code unter <http://examples.oreilly.com/jenut3/>
- [heise76736] Sicherheitslücke in SquirrelMail, <http://www.heise.de/newsticker/meldung/76736>
- [JACC] R. Monzillo, Java Authorization Contract for Containers, JSR-115, SUN Microsystems, <http://jcp.org/en/jsr/detail?id=115>
- [KS] W. Kriha, R. Schmitz, Internet-Security aus Software-Sicht, Springer 2008
- [Kuck] N. Kuck et al., SAP VM Container: Using Process Attachable Virtual Machines to Provide Isolation and Scalability for Large Servers, <http://bitser.net/isolate-interest/papers/PAVM.pdf>
- [New] T. Neward, Server Based Java Programming, Manning Publications, 2000
- [Oser] P.H. Oser, ORB Interceptors: A Crucial Missing Feature in the Enterprise JavaBeans Model, http://www.elca.ch/resources/InvokersNeedToBeAddedToJ2EE5_2.pdf
- [Pil] D. Pilipchuk, Using JAAS in Java EE and SOA Environments, <http://today.java.net/pub/a/today/2006/09/14/using-jaas-in-ee-and-soa.html>
- [SAFK] C. Sadtler, F. Albertoni, B. Fagalde, T. Kleinubing, WebSphere Application Server V6.1: Planning and Design, IBM Redbook, 2006
- [WF] D. Wallach, E. Felten et al., Extensible Security Architectures for Java, ACM SOSP’97, <http://www.cs.princeton.edu/sip/pub/sosp97.pdf>
- [z/OS] WebSphere Application Server for z/OS V5 and J2EE 1.3 Security Handbook (IBM Redbook), <http://www.redbooks.ibm.com/abstracts/sg246086.html>

Kapitel 8

Sichere Multi-Vendor Komponentensysteme

Multi-Vendor-Systeme sind eine wirtschaftliche Notwendigkeit, z. B. um die Anzahl der Rechner in Fahrzeugen zu reduzieren. Sie sind aber auch aus Gründen der Usability notwendig, da wir nicht gern zwanzig verschiedene Kartenlesegeräte für unsere verschiedenen Smartcards (EC-Karten, Kreditkarten, Badges) mit uns herumtragen wollen. Multi-Vendor-Systeme müssen also von verschiedensten Interessenten erweitert werden können. Sie müssen gleichzeitig verschiedene Applikationen betreiben können, ohne dass dabei Konflikte, etwa hinsichtlich der Ressourcenvergabe, auftreten dürfen.

Eine offene, erweiterbare Architektur ist demnach Voraussetzung für solche Geräte, was schnell zu Sicherheitsproblemen führen kann: Wie werden Installation und Updates abgesichert? In den meisten Fällen besteht die Antwort aus der Nutzung signierter Applikationen. Dieses Verfahren wird mittlerweile auch bei Betriebssystemen wie Windows Vista eingesetzt, um den Code von Treibern abzusichern. Symbian OS, FINREAD und andere haben ebenfalls solche Konzepte entwickelt. Wir werden in diesem Kapitel untersuchen, wie diese Konzepte durch Nutzung von Zertifikaten realisiert werden, zunächst am Beispiel OSGI: Dieses Framework ist als Plattform im Automotive- und Home-Bereich sehr populär geworden, aber auch als Basis von erweiterbaren Entwicklungsplattformen wie Eclipse. Wir untersuchen die Isolationseigenschaften der Plattform als Beispiel für sichere Namespaces.

Von großer Bedeutung ist auch eine klare Kenntnis der Bedrohungsmodelle solcher Plattformen. Bruce Schneier hat z. B. in einem Artikel [Schn99] die wesentlichen Szenarien für Smartcards zusammengestellt. Das Modell ist recht komplex aufgrund der Tatsache, dass es potenziell sehr viele Beteiligte an Smartcard-Lösungen gibt: Vom Hersteller über Firmen, die Applikationen entwickeln, bis hin zum Eigentümer der Karte, der jedoch nicht unbedingt die Rechte an den Daten der Karte innehat – zumindest nicht alle Rechte. Mögliche Manipulationen an der Hardware verkomplizieren das Modell zusätzlich. Wir werden speziell bei der Behandlung der FINREAD-Smartcard-Reader-Architektur darauf eingehen.

8.1 Dynamische Services und ihre Isolation in OSGI

Sogenannte „Hot-Swap“- und „Hot-Plug“-Techniken sind inzwischen Standard in vielen Bereichen der Hardware. Software hingegen tut sich mit dem Konzept von Erweiterungen noch schwer, ganz zu schweigen vom späteren Entfernen von Erweiterungen, vielleicht sogar zur Laufzeit des Systems. So erlaubte der ursprüngliche Plug-In-Mechanismus von Eclipse zwar das Laden von Extensions, allerdings mussten diese beim Start bereits vorhanden sein, und ein späteres dynamisches Entladen war unmöglich.

Dieses Problems hat sich die Open-Services-Gateway-Initiative [OSGI] angenommen und einen Framework-Aufsatz für Java entwickelt, der auf dem Konzept von dynamisch ladbaren Services beruht. Diese können auch dynamisch entladen werden. Die Services werden in sogenannten *Bundles* geliefert. Natürlich spielen Abhängigkeiten der Services und ihrer Komponenten (z. B. Packages, die sie benötigen oder anbieten) eine zentrale Rolle, und das Framework muss Funktionen anbieten, um diese Abhängigkeiten auszudrücken und zu kontrollieren.

Was ist daran aus der Sicht von sicherer Software so interessant? Abhängigkeiten sind lediglich eine andere Sichtweise auf die Erreichbarkeit von Code. Hinter der Kontrolle von Abhängigkeiten steckt also auch die Kontrolle von Zugriffen. Ein Servicekonzept will alle Zugriffe durch das Service-Interface zwingen und muss daher vermeiden, dass Zugriffsmöglichkeiten an dem Interface vorbei entstehen. Das ist aber nicht der einzige Grund. Die Services selber sollen maximale Unabhängigkeit voneinander besitzen: Diese sollen sich

- a) zur Laufzeit nicht stören oder manipulieren können (außer durch die Interfaces) und sie sollen
- b) frei variierbar sein, also eigene Versionen von Bibliotheken benutzen können, ohne eine gemeinsame Version nutzen zu müssen. Gleichzeitig sollen sie aber auch
- c) interoperabel sein, d. h. sie müssen von allen anderen Services potenziell nutzbar sein. Dies ist besonders ein Problem für Klassen des Service-Interfaces, wie noch zu zeigen sein wird.

Die Konzepte der Versionierung, Abhängigkeiten der Services und die Frage ihrer Granularität, Security und Isolation spielen bei einem dynamischen Servicekonzept ineinander und werden in den nächsten Abschnitten beschrieben. Zuvor jedoch wollen wir kurz einige Beispiele aus Projekten mit OSGI zeigen. Das OSGI-Framework ist übrigens auch seit Eclipse Version 3.0 die Basis der Plug-In-Technologie von Eclipse, jedoch sind noch längst nicht alle Eclipse-Erweiterungen auf dieser dynamischen Basis „angekommen“. So wurde z. B. ein großes neues Eclipse-Release nötig, bei dem eine große Zahl von Extensions mit ausgeliefert wurde – quasi miteinander synchronisiert und garantiert interoperabel. Das ist aber streng genommen ein Widerspruch zur Philosophie des dynamischen Ladens, das solche Mammut-Releases eigentlich unnötig machen sollte, wenn die Abhängigkeiten der Plug-Ins unter Kontrolle sind. Anders ausgedrückt: Die Not-

wendigkeit eines Mammut-Releases ist ein Zeichen dafür, dass die Serviceunabhängigkeit noch nicht wirklich überall realisiert wurde.

8.1.1 Anwendungsbeispiele in Heimnetzwerken

OSGI spielt nicht nur im Automotive-Bereich eine wichtige Rolle, sondern kann dies in Zukunft auch in Heimnetzwerken tun (vgl. [Kut]). Die Konfiguration solcher Netzwerke ist besonders kritisch, da sie das technische Vermögen der meisten Anwender übersteigt. Ron Kutschke entwickelte hierzu eine Konfigurationssprache, die es Service-Providern erlaubt, die unterschiedlichsten Geräte über einen Service zu konfigurieren. OSGI-Services im Residential Gateway übernehmen dann die durch Meta-Daten getriebene Anpassung an konkrete Hardware und Protokolle (s. Abb. 8.1).

Eine weitere wichtige Frage bei Heimnetzwerken ist das Finden von nötigen Updates der Maschinen. Markus Block (s. [Block]) entwickelte hierzu eine OSGI-JXTA-Kopplung, die das Finden der Updates durch Peer-to-Peer-Mechanismen ermöglicht. Die Updates selbst sind semantisch durch eine Ontologie beschrieben und können somit klar ausgewählt werden. Da auf den einzelnen Geräten unter-

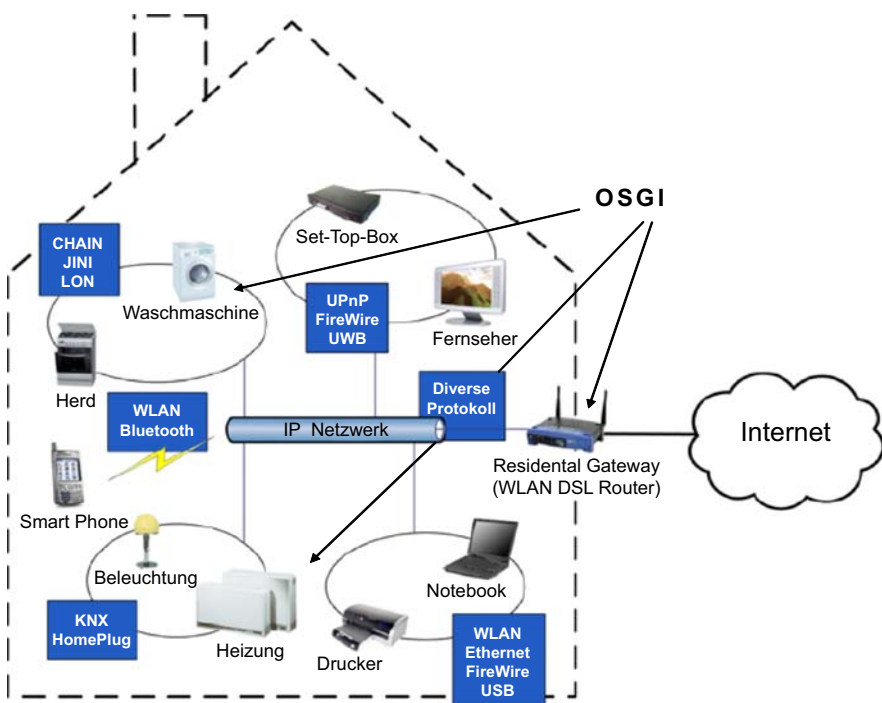


Abb. 8.1 OSGI-Services in einem Heimnetzwerk [Kut]

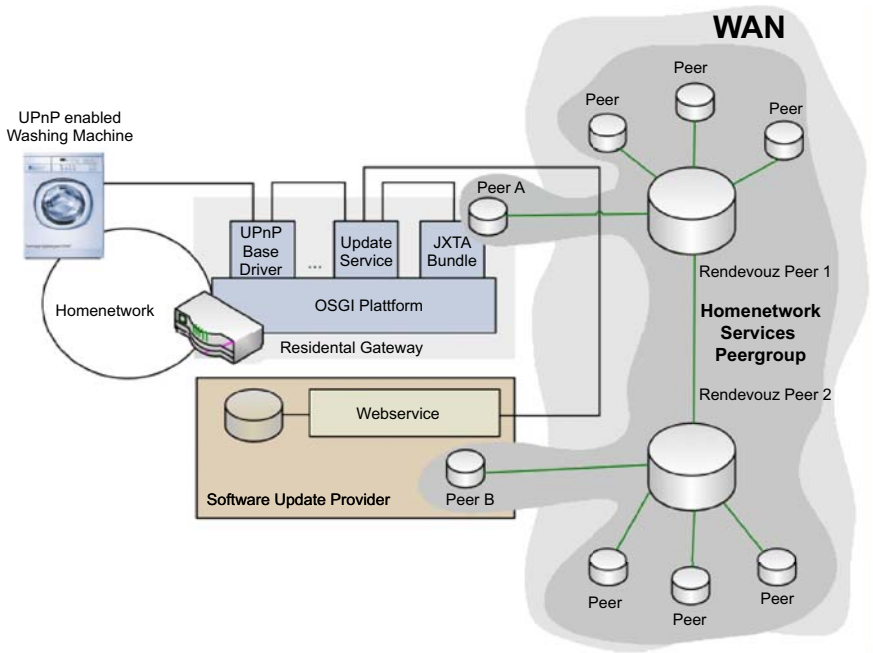


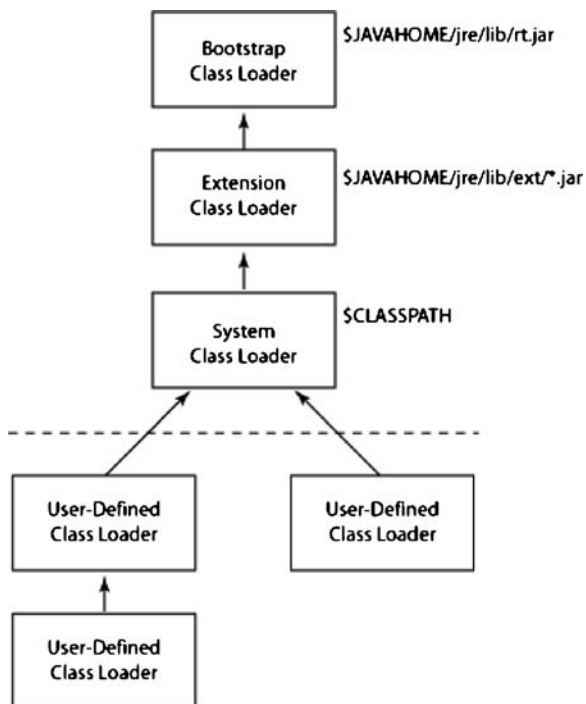
Abb. 8.2 Updating für Heimnetzwerke via OSGI [Kut]

schiedliche Services unterschiedlicher Hersteller ausgeführt werden, spielt die Isolation dieser Services eine große Rolle – nicht zuletzt aus Haftungsgründen, da durch die angeschlossenen Geräte und Maschinen Schäden verursacht werden können.

Abbildung 8.2 zeigt, wie auf der Basis des OSGI-Frameworks die verschiedenen Bundles (UPnP, Jxta etc.) kooperieren können. OSGI geht dabei im Allgemeinen von einer Multi-Application-Architektur pro Virtual Machine aus. Die schlechtere Isolation der Services bzw. Applikationen voreinander wird hier durch die bessere Performance und den geringeren Ressourcenverbrauch begründet. OSGI selbst erlaubt dabei jedoch auch Lösungen, die jedem Bundle eine eigene VM zuteilen. Durch die dann nötige Interprozesskommunikation wird die Performance jedoch teilweise um Faktoren schlechter.

8.1.2 Namespaces durch Classloader-Architektur

Das normale Konzept zum Laden von Klassen in Java verwendet eine Hierarchie von Classloader-Objekten, beginnend mit dem Boot Classloader (s. Abb. 8.3).

Abb. 8.3 Classloader-Hierarchie

Dazu gehört eine Auflösungsregel, die Classloader im Allgemeinen strikt beachten, hier dargestellt an einem Stück Beispielcode aus [SB]:

```

public Class loadClass(String name) throws
ClassNotFoundException
{
    Class c = findLoadedClass(name) //checks this CLs cache
    if (c == null)
    {
        try{
            c = getParent().loadClass(name); //delegate to parent CL
        }

        catch (ClassNotFoundException e) {...}

        if (c==null)

            c = findClass(name); //load Class
    }

    return c;
}

```

Eine Möglichkeit der Isolation besteht darin, getrennte Classloader nebeneinander zu verwenden. Klassen – selbst Klassen gleichen Namens und mit gleichem `.class`-File – werden von der Java Runtime als unterschiedliche Typen behandelt, wenn sie von verschiedenen Classloadern geladen wurden. Damit kann z. B. eine Klasse nur dann Parameter in ihren Methoden annehmen, wenn die Klassen der Parameter nicht selbst bereits durch den Classloader der Klasse geladen wurden und der Aufrufende die Parameter durch den gleichen Classloader geladen hat. Konkret bedeutet dies, dass Services oder Komponenten, die miteinander interagieren, die in den Interfaces verwendeten Klassen durch einen gemeinsamen Classloader laden lassen müssen, sonst kommt es zu `ClassCastException`s oder `ClassConstraintExceptions` zur Laufzeit. Das führt entweder dazu, dass alle Klassen der kommunizierenden Services oder Komponenten in der Hierarchie der Classloader bei einem gemeinsamen Vater deponiert werden (z. B. dem System-Classloader), damit sie konstant von einem einzigen Classloader geladen werden, zu dem Preis, dass die Isolation der Services aufgehoben wird und ihre Klassen dann sogar ganz anderen Diensten zur Verfügung stehen – oder aber es wird das Prinzip der hierarchischen Classloader aufgegeben und man geht zu einem Peer-to-Peer-Classloader über: Hier können Classloader auch an ihre Peers delegieren und nicht nur an den Parent (s. Abb. 8.4).

Dadurch können getrennte Namespaces innerhalb einer VM erzeugt werden. Das erlaubt es, unterschiedliche Versionen einer Klasse innerhalb der VM oder auch unterschiedliche Klassen gleichen Namens zu haben. Dieses Verfahren wird bei OSGI eingesetzt. Aber auch ohne verschiedene Versionen einer Klasse hat das Verfahren seine Vorteile: Es verhindert, dass auf Kleinstgeräten unnötig oft die gleichen Klassen geladen werden.

Wenn ein Service eine Klasse innerhalb eines Packages zum Export anbietet, dann werden die Classloader der anderen Services, wenn sie diese Klasse ebenfalls benötigen, diese nicht selbst laden, sondern sich an den Peer Loader wenden,

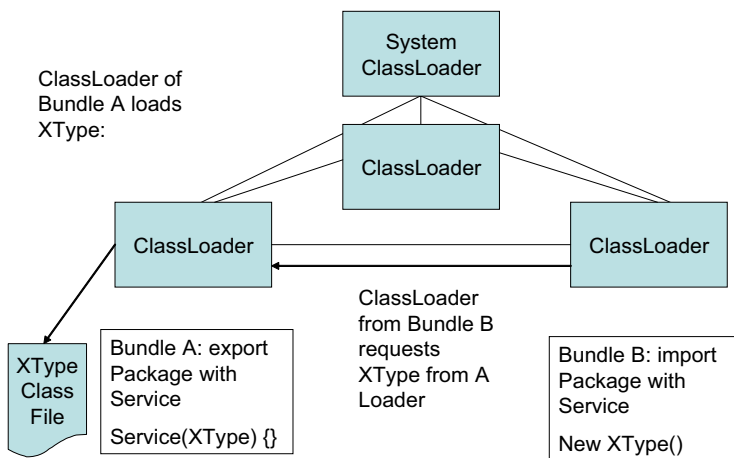


Abb. 8.4 Peer-to-Peer-Classloader

der die Klasse als erstes geladen hat. Dazu müssen Services deklarieren, welche Klassen sie exportieren bzw. importieren wollen. Diese Meta-Daten werden von den Classloadern dann zum Laden verwendet. OSGI gestattet es auch, die Versionen von Packages und Klassen festzulegen. Im Allgemeinen wird das Framework versuchen, die aktuellste Version einer Klasse zu laden (vgl. [Block]).

OSGI gestattet es auch, das Verhalten bei der Anforderung von Klassen per Reflection festzulegen bzw. einzuschränken. Ein weiterer Effekt dieses Verfahrens ist, dass es über Java-2-Permissions gesteuert werden kann. So hat OSGI spezielle Administrations-Permissions definiert, die den Export bzw. Import von Packages steuern. Auf diese Weise wird verhindert, dass Services indirekt an Packages kommen und dadurch ihre Fähigkeiten bzw. ihre Autorität erweitern können. Wir werden die Gefahren beim Laden im Kap. 10 unter dem Stichwort „Loader Isolation“ [Mil] noch genauer untersuchen.

Application-Server verwenden das Peer-to-Peer-Classloader-Modell, um ihre eigenen Klassen vor der Verwendung durch Applikationen zu schützen und zwar sowohl aus Sicherheits- als auch Wartungsgründen (Applikationen sollen nicht gegen Interfaces programmieren, die nicht stabil gehalten werden können). Wir haben dazu bereits Beispiele im Bereich Websphere gesehen.

Das Peer-to-Peer-Modell der Classloader isoliert also Softwareservices voneinander. Gefährlich sind jedoch gemeinsame Klassen, die noch über die verbliebene Hierarchie erreichbar sind. Auch können Klassen, die noch verwendet werden, bei der Deinstallation von Services nicht einfach entfernt werden.

8.1.3 *Trusted Code und Rechtezuweisung*

OSGI unterstützt die Signatur von Bundles, auch durch mehrere unabhängige Signierer. Dazu werden in einem sogenannten *Manifest File* zunächst die Hash-Werte aller in dem Bundle enthaltenen Ressourcen gebildet. Die Signatur erfolgt nun nicht direkt über das Manifest, sondern um die Signatur durch mehrere Signierer zu ermöglichen, wird für jeden Signierer ein separates *Signature File* gebildet, das wiederum einen Hash-Wert des Manifests enthält. Im *Signature Block File* ist schließlich für jeden Signierer die digitale Unterschrift über dem Signature File zusammen mit dem dazu gehörigen Public Key enthalten (s. auch Abb. 8.5). Das Manifest, das Signature File und das Signature Block File bilden zusammen die Meta-Daten eines Bundles und werden gemeinsam mit diesem im Bundle Archive gespeichert.

Über die Signatur kann der Urheber des Codes also sicher verifiziert werden. Die Rechteverteilung erfolgt dann über die Autorisierungsmechanismen der Java 2 Security: Eine Vorkonfiguration der Rechte kann durch den Hersteller der Services erfolgen, die Endkonfiguration nimmt dann der Administrator vor, der Software von gewissen Herstellern global bestimmte Rechte zuteilen kann.

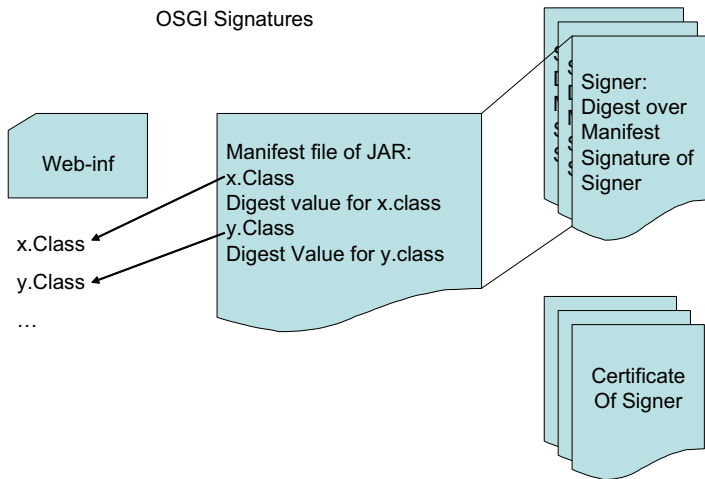


Abb. 8.5 OSGI Bundle Signature

8.2 Sichere Transaktionen mit FINREAD

Im Kapitel zu Enterprise Security haben wir das Thema Ende-zu-Ende-Sicherheit bereits aus Firmensicht behandelt. Dort ist damit häufig die sichere Bearbeitung eines Kunden-Requests vom Moment des Eintritts in die Firmeninfrastruktur (meist DMZ) bis hin zu den Backend-Systemen auf Mainframes gemeint.

Nicht zuletzt die Vielzahl von Phishing-Attacken zeigt jedoch, dass bei dem Thema Ende-zu-Ende-Sicherheit auch der Benutzer oder Kunde auf der anderen Seite einbezogen werden muss – inklusive dessen IT-Ausstattung wie z. B. der PC oder das Handy. Finanzinstitute beispielsweise mussten lernen, dass der PC des Kunden kein geeignetes Mittel ist, um Geheimnisse wie PINs/TANs oder Schlüssel aufzubewahren. Im Einzelnen sind die Probleme auf der Kundenplattform die folgenden:

- Attacken durch Malware auf gespeicherte Geheimnisse (ohne Beteiligung des Users).
- Semantische Attacken auf Geheimnisse durch direkte Abfrage beim User unter Vorspiegelung falscher Informationen.
- Korruption von Transaktionssignierung, indem dem Kunden ein anderer Inhalt zum Signieren angezeigt wird, als dann im Hintergrund tatsächlich durch den Private Key des Kunden signiert wird.

Es ist besonders der letzte Punkt, der weh tut, denn im Grunde ist die Transaktionssignierung als objektbasierte Sicherheit sehr gut geeignet, um die Kommunikation eines Kunden mit seinem Finanzinstitut abzusichern, da dann Man-in-the-

Middle-Attacken nicht mehr möglich sind (der Angreifer kann den vom Kunden signierten Auftrag nicht abändern). Wenn natürlich dem Kunden bereits beim Signaturvorgang falsche Daten angezeigt werden, kann auch die Transaktionssignierung nicht mehr helfen.

Ein wesentlicher Grund dafür, dass der PC als Instrument für sichere Signaturen nicht taugt, liegt in den Sicherheitsproblemen des GUIs: Es bietet keinen „Trusted Path“ für den Benutzer von den angezeigten Daten zum Signaturmechanismus. Die Systemsoftware müsste eigentlich verhindern, dass Inhalte, die angezeigt werden, andere sind als Inhalte, die dann signiert werden, hilft dem User aber – zumindest im Moment – nicht weiter.

Der FINREAD-Ansatz erklärt deshalb den PC zur „untrusted plattform“ und versucht, für den User eine geschützte und sichere Umgebung für die Signierung von Transaktionen zu schaffen. Dazu muss der Benutzer allerdings eine spezielle Hard- und Software verwenden: Den FINREAD Card Reader. Ein solcher Card Reader erhöht nicht die Sicherheit der beteiligten Protokolle, sondern gestattet lediglich einen sicheren Zugriff auf das in Smartcards enthaltene Schlüsselmaterial. Er stellt also lediglich ein sicheres Interface zur Smartcard dar. Klar ist, dass nur Card Reader mit eigenen Displays in der Lage sind, den Nutzer vor Täuschungen zu bewahren.

Auf den folgenden Seiten untersuchen wir die folgenden Aspekte des FINREAD-Konzeptes:

- Benutzung und Modi,
- Architektur,
- Bedrohungsmodell,
- Schlüssel- und Zertifikatsmanagement und
- Software Environment zur Isolation (VM, Packages, Threading, Core-Functions, Class-Loader).

Die Sicherheit von Embedded-Control-Plattformen wird in den nächsten Jahren eine immer größere Rolle spielen. Der FINREAD-Ansatz ist ein exzellentes Beispiel für die Durchführung einer Sicherheitsanalyse, basierend auf begründeten Annahmen, Anforderungen, Risikoabschätzungen und der Definition von Abwehrmaßnahmen. Die FINREAD-Sicherheitsanalyse [FR_Sec] kann durchaus als beispielhaft für Projekte gelten, die eine sichere Interaktion zwischen Personen und Schlüsseln garantieren müssen. Die geplante Gesundheitskarte, Jobkarte und evtl. auch Wahlkarten könnten alle in diese Kategorie fallen.

8.2.1 Benutzung

Eine FINREAD-Transaktionslösung basiert auf mehreren Komponenten: Einem als „untrusted“ angenommenen PC sowie einem als „trusted“ angenommenen Card Reader der Klasse 4 (d. h. ein Kartenleser mit eigenem Display und eigener Tastatur), der auf der FINREAD-Architektur beruht. Wenn ein Benutzer diesen

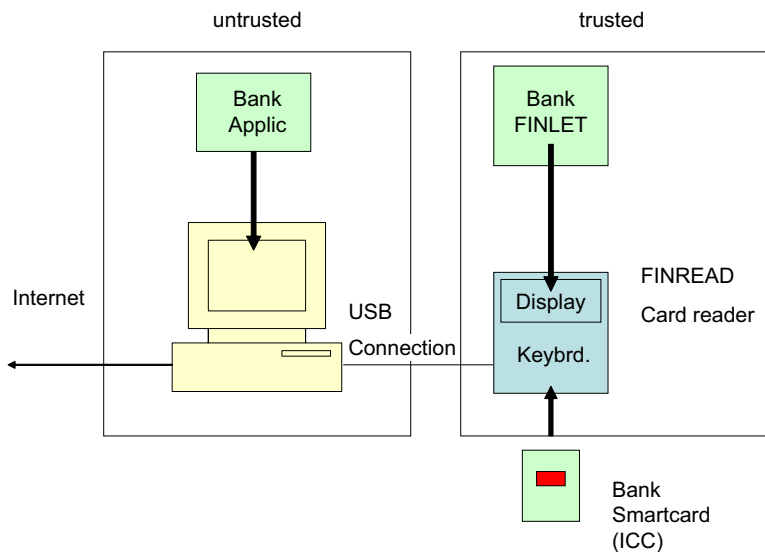


Abb. 8.6 Basisszenario mit Nutzer-PC und FINREAD Card Reader

Card Reader an den PC anschließt und eine Smartcard in den Card Reader steckt, dann geschieht grob das Folgende: Eine spezielle Form eines Applets (genannt *Stiplet*) erkennt die Smartcard als zu ihrer Firma gehörend und der Card Reader wechselt in den sogenannten sicheren Modus. In diesem Modus gibt der Benutzer die PIN über das Tastaturfeld des Readers ein und das Stiplet erhält Zugriff auf die Smartcard. Gleichzeitig kommuniziert eine Applikation auf dem PC über das Stiplet mit der Karte.

Soll der Benutzer eine Transaktion bestätigen, fordert dies die Applikation auf dem PC an (s. Abb. 8.6). Die zu signierenden Daten erscheinen im Display des Card Readers und das Stiplet veranlasst die Signierung der Daten durch die Smartcard. Auf diese Weise wird verhindert, dass eine manipulierte Applikation auf dem PC falsche Daten signieren kann. Der Benutzer muss allerdings die Korrektheit der Daten auf dem Display des Card Readers prüfen.

8.2.2 Bedrohungsmodell

Schneier und Shostack haben in [Schn99] die Bedrohungsszenarien bei Smartcards auf zwei entscheidende Faktoren zurückgeführt:

Die Trennung des Rechners auf der Smartcard von wesentlichen Komponenten wie Display, Keypad und I/O, die alle von einem Terminal bereitgestellt werden müssen.

Die große Zahl der am Gesamtsystem Beteiligten: Hersteller, Benutzer, Eigentümer etc. und die Tatsache, dass z. B. Dateneigentümer und Kartenbesitzer unterschiedlich sein können (vgl. auch Abb. 8.7).

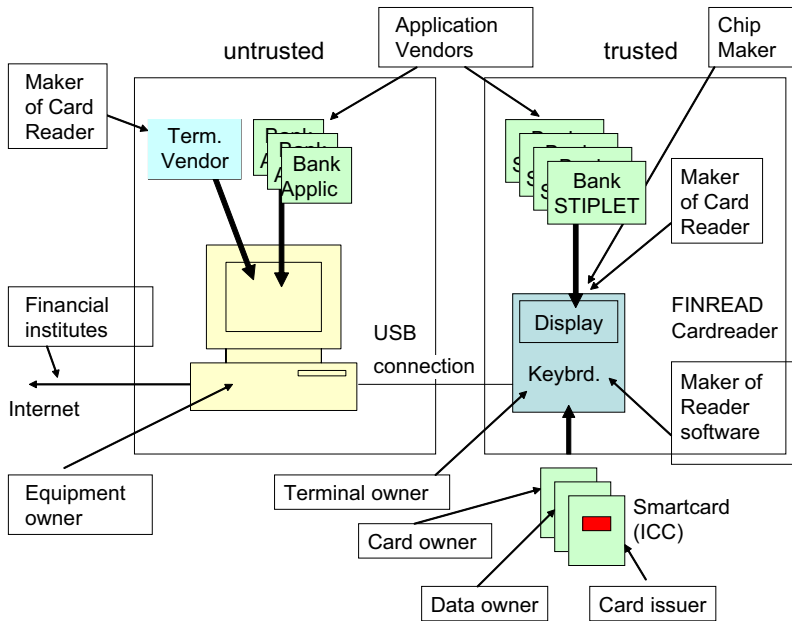


Abb. 8.7 Beteiligte Parteien im Smart-Card-Szenario

Sicherheitsannahmen, die die Gefahr dieser technischen Trennung dadurch verringern wollen, dass Eigentümer der Daten und Besitzer der Karte als identisch angenommen werden, werden im Fall eines Diebstahls ad absurdum geführt. Das Gleiche gilt für den Besitz des Terminals.

Die ideale Smartcard hätte Display und Tastatur „an Bord“ und hätte Kontrolle über die eigene I/O inklusive Stromversorgung. Da dies unter anderem an den momentanen Hardwarekosten scheitert, stellt die Bereitstellung eines sicheren Terminals zurzeit einen Kompromiss dar.

Das Terminal muss einige wesentliche Sicherheitsleistungen erbringen:

- Sicherstellen, dass Applikation und Karte zusammenpassen.
- Identifikation des Terminals gegenüber Instituten.
- Sicheres Speichern von Schlüsseln.
- Durch den Benutzer kontrollierte, sichere Freigabe von Schlüsseln bzw. deren Verwendung.
- Anbieten eines Trusted Path für den Nutzer z. B. zur Eingabe des PINs.
- Genaue Information des Benutzers über den momentanen Modus und dessen Gefahren.

Das ideale Terminal würde nur eine einzige Karte bearbeiten können, aber auch das ist weder finanziell noch von der Benutzungsfreundlichkeit her gesehen eine gute Lösung. Die FINREAD-Spezifikation folgt an dieser Stelle den Erwartungen des Marktes und bietet einen transparenten Modus an, in dem der Kartenleser wie ein „dummer“ Kartenleser funktioniert, d.h. ohne die Intelligenz der im Leser

laufenden Stiplets. Applikationen auf dem PC können dann mit Smartcards arbeiten, ohne dass die Stiplets die Kommunikation zwischen einer Untrusted Application auf dem PC und der Smartcard überwachen könnten.

Ein FINREAD-Kartenleserterminal lässt dabei per Default nur ungefährliche Kommandos vom PC an die Smartcard zu. Bei gefährlichen Kommandos wird der Nutzer darauf hingewiesen, dass sich der Leser im transparenten Modus befindet und wird außerdem zur Eingabe der PIN am Kartenleser aufgefordert. Aus Security-Sicht ist dies natürlich ein kritischer Punkt, denn der Benutzer kann hier durch falsches Verhalten einen Missbrauch der Smartcard zulassen. Softwareseitig lässt sich der transparente Modus jedoch für eine Karte abschalten.

Eine vollständige Liste der Bedrohungsmodelle würde die Beteiligten in einer Tabelle gegenüberstellen und potenzielle Angriffsmöglichkeiten auflisten. Wichtig dabei ist: Die Beteiligten werden nicht als Personen verstanden, sondern als Rollen. Dies ist von großer Bedeutung, wenn z. B. ehemals getrennte Rollen plötzlich in einer Person vereint werden und die Sicherheit auf der Verteilung der Rollen auf verschiedene Identitäten beruhte. Hier zu Illustrationszwecken ein kleiner Auszug aus der vollständigen Liste:

Rolle des Angreifers	Angriffsziel	Angriffsart
Datenbesitzer	Terminal	Falsche App. Zuordnen
Datenbesitzer	Stiplets im Terminal	Denial of Service
Kartenbesitzer	Terminal	PIN Ablauschen
Stiplets	Kartenbesitzer	PIN Ablauschen
Institute	Terminalbesitzer	Terminal fälschen
PC-Software	Kartenbesitzer	Impersonation, PIN

Eine umfangreiche Liste von Bedrohungen ist auch Teil der FINREAD-Spezifikation, ebenso die dazu getroffenen Abwehrmaßnahmen. Eine ganze Reihe von Bedrohungen, die Schneier bei fest eingebauten Terminals definiert, fallen beim persönlichen FINREAD-Kartenleser weg, z. B. Angriffe des Kartenbesitzers bzw. Datenbesitzers auf das Terminal, um z. B. Telefongebühren zu umgehen oder Waren zu erhalten, ohne dass ein Abbuchungsvorgang durchgeführt werden kann. Genauer untersucht werden muss hingegen der Fall, was passiert, wenn das Terminal bzw. die Karten verloren gehen oder gestohlen werden.

8.2.3 *FINREAD-Architektur*

Wesentliches Sicherheitsmerkmal des Readers ist ein geschützter Bereich, in dem ein privater Schlüssel gehalten werden kann. Er dient der Authentisierung des Readers gegenüber der Chipkarte bzw. dem PC. Dieser Bereich muss „tamper-proof“ sein, er darf also nicht manipuliert werden können, bzw. solche Manipulationen müssen erkannt werden können (er kann aber natürlich zerstört werden).

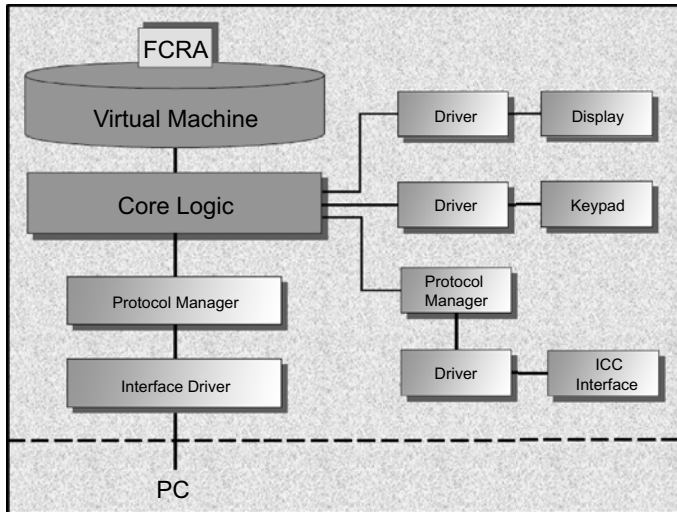


Abb. 8.8 Softwarearchitektur des FINREAD-Card Readers

Die Software eines FINREAD-Readers gliedert sich in zwei große Bereiche: Core Logic und Applikationen, die von unterschiedlichen Herstellern stammen können. Die Core Logic kontrolliert I/O (Driver, Events, Communication) und enthält auch eine Java Virtual Machine sowie ein zugeschnittenes J2ME-Runtime (s. Abb. 8.8). Die Architektur lässt nur die Ausführung einer einzigen Applikation zu jedem Zeitpunkt zu. Die Applikationen teilen sich grundsätzlich in solche, die von einem angeschlossenen PC aus angestartet werden und solche, die bereits beim Einstecken einer Karte aktiviert werden. Dies ist eine Sicherheitsmaßnahme einiger Applikationshersteller, die verhindern wollen, dass ihre Smartcards ohne dazwischen geschaltete Stiplets angesprochen und manipuliert werden können. Es ist Aufgabe der Core Logic, hier eine korrekte Verbindung zwischen Karte und Kartensoftware im Reader herzustellen.

Ein naives „Chain-of-Responsibility“-Pattern, bei dem die Core-Logik jedes Stiplet fragt, ob es für eine bestimmte Karte zuständig ist, birgt Sicherheitsrisiken in sich: Ein Stiplet könnte sich fälschlicherweise für zuständig erklären und die Smartcard-Signatur missbrauchen. Ein deklarativer Ansatz, bei dem Software und Kartentyp durch die Systemsoftware des Readers selbst aufeinander abgebildet werden, ist hier deswegen sicherer: Der Zusammenhang von Reader-Stiplet und der jeweiligen Karte könnte z. B. durch eine Signatur des Herstellers des Readers bestätigt werden. Sollte jemand versuchen, eine Applikation für einen bereits registrierten Kartentyp zu installieren, dessen Software ebenfalls bereits installiert ist, dann müsste der Reader den Konflikt melden und die Installation verweigern. Das folgende Sequenzdiagramm aus [FR_Arc] in Abb. 8.9 verdeutlicht das Vorgehen.

Die Core Logic interpretiert alle Kommandos vom PC und kontrolliert, ob sie zum jeweiligen Betriebsmodus der Karte und des Readers passen. Sie stellt APIs zur Verfügung, mit deren Hilfe die Stiplets Daten von der Tastatur des Readers

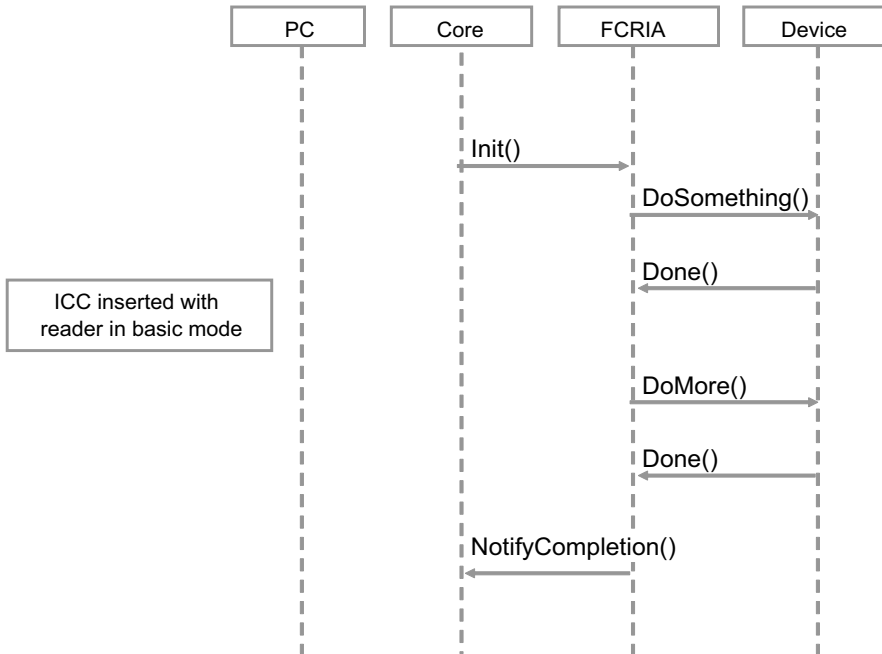


Abb. 8.9 Kommunikation der Core Logic mit den Devices auf dem Reader (Tatstatus, Display etc.) [FR_Arc]

erhalten und Informationen auf das Display des Readers ausgeben können. Direkte Zugriffe sind nicht möglich. Auf diese Weise muss ein Stiplet den Wert einer eingegebenen PIN gar nicht erfahren. Aufgabe des Readers hingegen ist es, sicherzustellen, dass der Nutzer nur durch die Core-Software die PIN eingeben kann und nicht durch einen gefälschten Dialog des Stiplets.

8.2.4 Trust- und Key-Management

Die FINREAD-Architektur definiert eine Reihe von öffentlichen und privaten Schlüsseln für den Kartenleser selbst, den Hersteller des Kartenlesers sowie den Hersteller von Applikationen für den Leser (s. auch Abb. 8.10):

- FCR Private/Public Authentication Key plus Certificate,
- Application SW Public Root Key,
- Core SW Public Root key und
- Public Master Key for Root Key Replacement.

Wie man sieht, ist lediglich ein einziger privater Schlüssel auf dem Kartenleser vorhanden. Er dient zur Authentisierung des Lesers gegenüber Karte und/oder PC und wird bei der Personalisierung des Lesers erzeugt und im Sicherheitsmodul

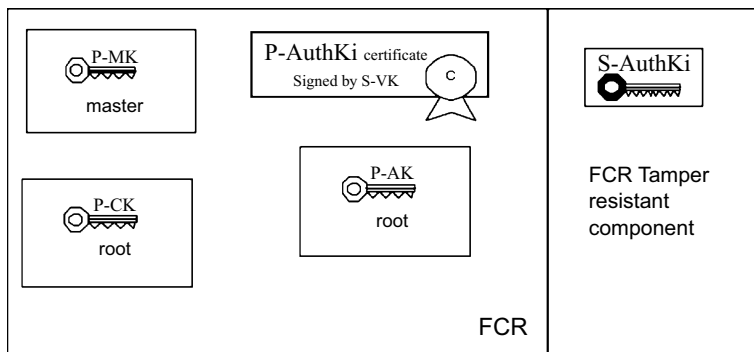


Abb. 8.10 Im Card Reader vorinstallierte Schlüssel [FR_Sec]

gespeichert. Zu diesem Schlüssel existiert im Leser auch ein vom Hersteller unterzeichnetes Zertifikat des zugehörigen Public Keys.

Alle anderen Schlüssel sind öffentliche Schlüssel und dienen nur der Verifizierung von Signaturen von Software im Falle der Erstinstallation oder eines Updates. Zu unterscheiden sind dabei drei Fälle:

- Eine neue Applikation wird installiert oder ein Update erfolgt. Dieser Vorgang wird mit dem öffentlichen Schlüssel für Applikationssoftware geprüft.
- Die Core Logic selbst muss ausgetauscht werden. Dieser Vorgang wird mit dem öffentlichen Schlüssel für Core-Software geprüft, die der Hersteller der Core-Software mit seinem privaten Schlüssel signiert hat.
- Die Public Keys für Core und Applikationssoftware selbst müssen erneuert werden. Das herunter geladene Schlüsselpaket wird mit dem öffentlichen Master-Schlüssel des Leserherstellers verifiziert.

Das folgende Diagramm aus der FINREAD-Spezifikation (Abb. 8.11) zeigt den Vorgang von Downloads mit zugehörigen Komponenten und Prüfungen. Die Core-Software – und hier speziell der Download-Manager – prüft den Softwaretypus und selektiert den entsprechenden Schlüssel. Dabei gelten folgende sicherheitstechnisch relevante Bedingungen:

- Erst wenn die Validierung positiv ausfällt, wird die Software aus einem temporären Bereich in den Block für Applikationen oder Core Logic geladen.
- Bei einer Erneuerung der Core Logic werden alle Applikationen gelöscht.
- Beim Löschen einer Applikation stellt die Core Logic sicher, dass alle zugehörigen Daten ebenfalls vernichtet werden.
- An keiner Stelle übergibt die Core Logic die Kontrolle einem „Installer“ aus dem herunter geladenen Paket, sondern verteilt die Software entsprechend ihres Typs.

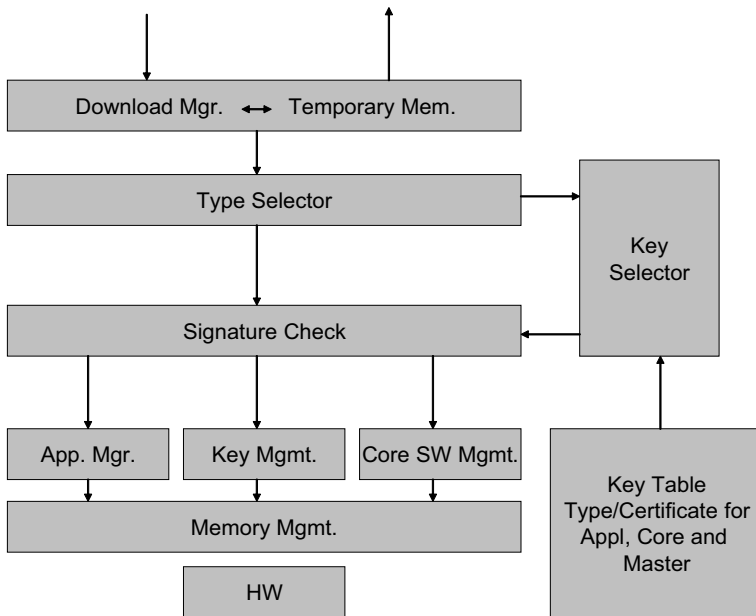


Abb. 8.11 Vorgehensweise beim Download von Applikationen [FR_Sec]

Anders als OSGI arbeitet FINREAD nicht mit Java-2-Permissions. Wenn die Signatur einer Applikation validiert werden kann, gilt sie als „trusted“. Die Kontrolle erfolgt über die ausgewählten APIs der Kernsoftware sowie über ein eingebautes Sandbox-Konzept der Virtual Machine (siehe Abschn. 8.2.5).

Die Sicherung von Software allein durch digitale Signaturen der Hersteller wurde bereits mehrfach diskutiert. Es handelt sich dabei um eine rein vertrauensbasierte Sicherung, die die effektive Qualität der Software nicht prüfen oder einschränken kann. Der Vorgang einer solchen Sicherung ist in der FINREAD-Spezifikation sehr klar und gut beschrieben und dient uns an dieser Stelle als Vehikel für eine kurze Beschreibung. FINREAD erlaubt übrigens sowohl horizontale als auch hierarchische Zertifizierungen über mehrere Stufen hinweg. Die Organisation solcher Vertrauensketten beinhaltet einige finanzielle und politische Sprengkraft, denn sie schafft Abhängigkeiten unter den Herstellern (wer darf z. B. Treiber unter Windows signieren? Symbian-OS-Software beispielsweise darf lediglich von der Symbian-Organisation oder von ihr beauftragten Testcentern signiert werden).

Hersteller von Applikationen für den Card Reader besitzen ein selbstsigniertes Zertifikat für den öffentlichen Schlüssel des Herstellers des Lesers. Dieser hat mit dem zugehörigen privaten Schlüssel den öffentlichen Schlüssel des Lesers signiert. Wenn der Leser sich authentisieren will, schickt er das Zertifikat mit seinem öffentlichen Schlüssel an die Gegenseite. Diese kann das Zertifikat mit dem öffentlichen Schlüssel des Herstellers prüfen. Anschließend kann ein Challenge/Response-Verfahren durchgeführt werden, in dessen Verlauf der Kartenleser

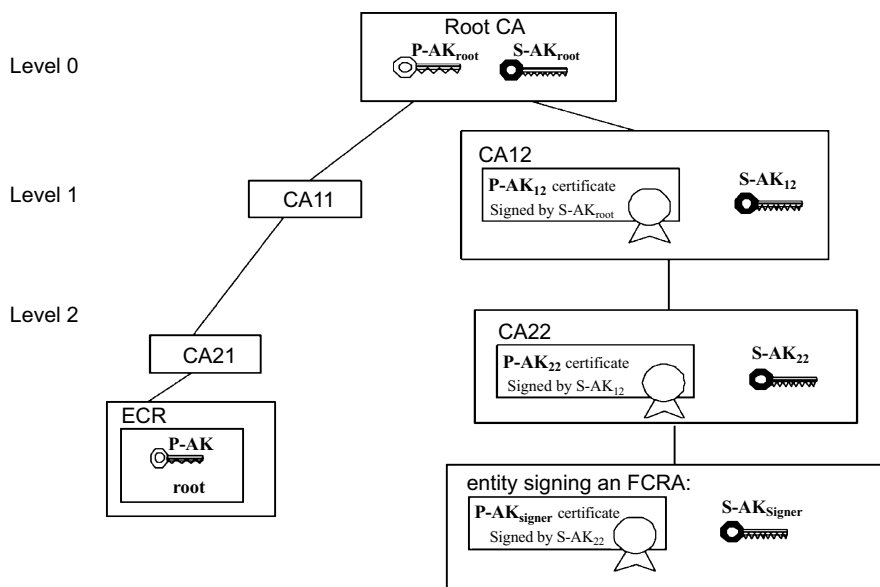


Abb. 8.12 Hierarchische Verifikation eines Software-Downloads mithilfe von Zertifikatsketten [FR_Sec]

den Besitz des zugehörigen privaten Schlüssels durch seine Anwendung auf die Challenge beweist. Die Gegenstelle wiederum prüft mit dem öffentlichen Schlüssel des Lesers die korrekte Beantwortung der Challenge.

Abschließend sei noch der Vorgang des Downloads einer Applikation auf den Leser gezeigt, kontrolliert durch eine hierarchische Kette von Zertifikaten (s. Abb. 8.12). Zum Einsatz kommt hierbei im Leser nur das selbst signierte Root-Zertifikat des Herstellers des Lesers. Mithilfe des öffentlichen Schlüssels darin kann der Leser letztlich die Vertrauenskette prüfen. Im Paket der Software werden dazu alle Zertifikate der dazwischen beteiligten Hersteller mitgeliefert. Die Core Logic des Readers kann dann die Kette der Zertifikate prüfen.

Hier die Schritte im Einzelnen: Das Softwarepaket für den Reader ist mit dem privaten Schlüssel des Herstellers der Software signiert. Der öffentliche Schlüssel des Herstellers, mit dem die Signatur geprüft werden kann, begleitet die Software in Form eines Zertifikates. Dieses ist lediglich die Unterschrift einer zweiten Firma unter den Namen und den öffentlichen Keys des Herstellers. Auch der öffentliche Schlüssel der zweiten Firma begleitet die Software in Form eines Zertifikats, signiert durch eine dritte Firma und so weiter.

Der Download-Manager prüft nun die Kette der Signaturen, beginnend mit der signierten Software selbst durch deren öffentlichen Schlüssel. Danach prüft er das Zertifikat des öffentlichen Schlüssels durch den Schlüssel im Zertifikat der übergeordneten Firma. Stößt der Download-Manager irgendwann auf ein Zertifikat, das mit dem geheimen Schlüssel des Herstellers des Readers signiert wurde, dann kann er dieses mit dem dazugehörigen öffentlichen Schlüssel des Herstellers des

Readers validieren, den er bereits besitzt. Somit ist die Kette der Zertifikate bei der Wurzel – dem Hersteller des Lesers – angelangt und die Verifikation ist abgeschlossen.

Was bedeutet dies nun für die Sicherheit der geladenen Software? Das Verfahren prüft, wie gesagt, nicht die Gefährlichkeit der Software. Das Verfahren hat außerdem das Problem, dass die Vertrauensweitergabe mit jeder weiteren Ebene von Zertifikaten kritischer wird: Hierarchische Vertrauensketten erlauben jeder weiteren Ebene die weitere Unterzeichnung von Schlüsseln, die wiederum ihrerseits weitere Schlüssel unterzeichnen dürfen. Die eigentliche Gefahr dabei ist, dass die Prüfverfahren, die die Grundlage der Unterzeichnung eines Zertifikats sein sollten, mit der Anzahl der Beteiligten immer kritischer werden: Prüft die CA21 auch noch so korrekt wie die Root-CA? Das gesamte Verfahren beruht auf transitivem Vertrauen, mit der Konsequenz, dass übergeordnete CAs die Weitergabe dieses Vertrauens nicht kontrollieren können.

Ist nun die Symbian-Methode, Softwarezertifizierungen entweder nur selber zu erstellen oder an wenige, geprüfte Testcenter zu delegieren, sicherer? Zumindest ist es in so einem Kontext wohl einfacher, die strikte Einhaltung bestimmter Testverfahren zu garantieren. Andererseits besteht nach wie vor die Frage, was dort eigentlich geprüft werden kann? Der mathematisch strenge Nachweis der Ungefährlichkeit einer Software lässt sich schließlich grundsätzlich nicht führen.

Der Ansatz der Signierung und Zertifizierung von Software zum Download ist aber prinzipiell durchaus sinnvoll: Es gibt eine Reihe von Attacks auf semantischer Ebene, die durch keine noch so gute Zugriffskontrolle auf Plattformen nicht ausgeschaltet werden können, etwa weil der Benutzer fälschlicherweise die Zustimmung zu bestimmten Aktionen erteilt. Letztlich benötigt man aber Sicherheitsmaßnahmen auch auf der Plattform selbst, um Erweiterungen der Softwareplattform sicher zu gestalten. Der nächste Abschnitt beschäftigt sich deshalb mit der Architektur der Plattformsoftware und ihrem Einfluss auf die Sicherheit von Transaktionen.

8.2.5 Softwareumgebung zur Isolation

Die Kernstücke sicherer Software auf FINREAD-Lesern sind eine Laufzeitumgebung für Applikationen auf Basis einer Virtual Machine (und damit verbunden die Isolation der Applikationen von der Hardware), sowie die Architektur der Applikationen selbst in Form des eventgetriebenen STIP-Frameworks.

Bei der FINREAD Virtual Machine handelt es sich um eine Java-VM, die speziell für den Einsatz in Embedded-Control-Umgebungen konzipiert wurde. Die gesamte Laufzeitumgebung von FINREAD folgt in Grundzügen der Architektur von J2ME und dessen Connected Limited Device Configuration (CLDC) Profile, das die Bibliotheken und VM-Funktionen für solche Plattformen definiert. Uns interessiert an dieser Stelle nur, welche Einschränkungen die FINREAD-Architek-

tur aus Sicherheitsgründen trifft. Dies dokumentiert sich in erster Linie an den von FINREAD verbotenen Bibliotheken und Mechanismen, nämlich:

- Threading,
- File und buffered I/O,
- Reflection,
- Security Manager,
- Custom Classloader und
- Native Methods.

FINREAD verzichtet aus Sicherheitsgründen auf multi-threading und weil die Natur der Applikationen es nicht erfordert. Es darf ohnehin nur jeweils eine Applikation zu jeder Zeit auf der VM laufen. Dies ist ein deutlicher Unterschied z. B. zu OSGI mit den verschiedensten Bundles und Services, die unterschiedliche Applikationen darstellen können.

Passend dazu verbietet FINREAD auch den Einsatz eigener Classloader, um die verwendeten Typen genau kontrollieren zu können. Standardklassen dürfen von Applikationen nicht ersetzt werden.

Der Zugriff auf Files und Streams wird unterbunden, da kein File-System vorhanden ist. Die Speicherung von Daten für Applikationen erfolgt über spezielle Core-Logic-APIs. Ohne die Verwendung von Java 2 Security und seinen Permissions ist die Einschränkung von Zugriffen auf ein File-System ohnehin problematisch, da die File-Klassen in Java-Zugriffsmethoden auf Verzeichnisse erlauben. Dadurch wäre die Gefahr gegeben, dass Applikationen die Daten anderer sehen oder ändern könnten.

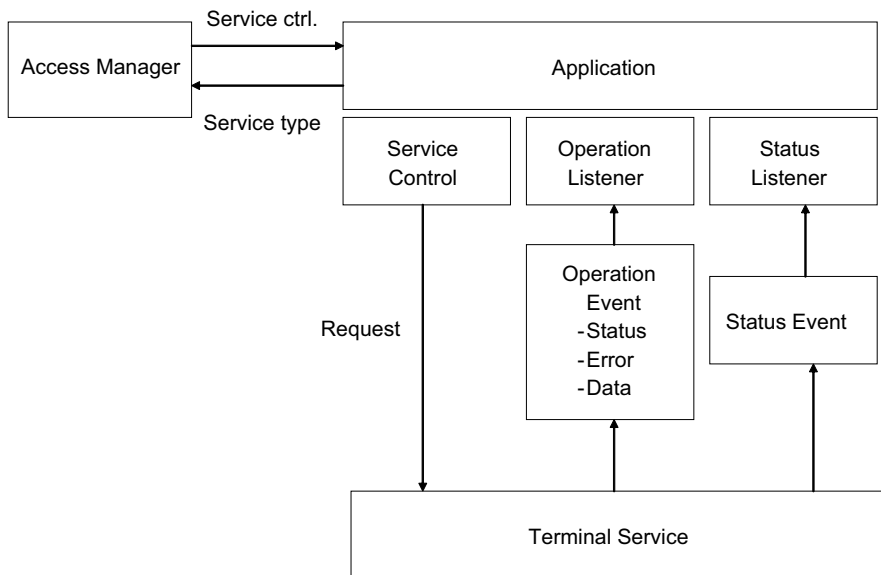
Der Reflection-Mechanismus wird von FINREAD verboten, da es bei den Applikationen keine Notwendigkeit dafür gibt. Damit ist auch die Gefahr geringer, dass Applikationen ungewollten Zugriff auf Klassen des Laufzeitsystems erhalten.

Die Security-Manager-Implementation von Java 2 wird von FINREAD als unnötig und zu groß angesehen. Ein in die VM eingebauter Security-Manager verwirklicht eine einfache Sandbox für die jeweils laufende Applikation.

Auch Custom Classloader werden nicht unterstützt. Damit können Applikationen keine Typen unkontrolliert laden.

Nicht zuletzt ist die Erweiterung der Laufzeitumgebung durch native Methoden nicht möglich. Dadurch könnten sich Applikationen keinen direkten Zugriff auf Devices wie Keypad, Display oder Kommunikationskanäle verschaffen.

Nun zur Applikationsarchitektur: Sie folgt der Spezifikation von STIP ([STIP]). Die FINREAD-Architektur teilt die Skepsis vieler Plattformen, was die Verwendung von Shared State Multithreading betrifft: Der dann nötige Einsatz von Monitoren und Semaphoren erschwert die Applikationsentwicklung deutlich und führt zu Qualitäts- und Sicherheitsproblemen. Ähnlich dem Symbian-OS und seinen Active Objects definiert FINREAD deshalb eine eventgesteuerte Architektur für Applikationen. Einer Applikation steht nur ein einziger Thread zur Verfügung. Dafür muss die Applikationslogik in nicht-blockierende, kurze Funktionen aufgeteilt werden. Abbildung 8.13 zeigt die Verwendung von Service-Control-Objekten sowie den dazu gehörenden Listnern, die die asynchron mitgeteilten Resultate in Empfang nehmen.



Asynchronous request handling in FINREAD

Abb. 8.13 Verwendung von Service-Control-Objekten und Listnern [FR_Arc]

Eine Applikation, die mit einem Gerät kommunizieren will (oder einen anderen Core-Service benötigt), tut das, indem sie ein Service Control Object dafür vom Access-Manager anfordert. Der Aufruf von Methoden des Service Control Objects fordert die Durchführung einer Aktion beim Terminal an, wartet jedoch nicht auf das Resultat. Ist die Applikation am Resultat interessiert (z. B. Keypad Input), muss sie einen Listener dafür registrieren. Diesem Listener Object teilt dem Terminal das Ergebnis in Form eines Operation Events mit. Ist eine Applikation grundsätzlich am Status des Terminals interessiert, dann kann sie auch dafür einen Listener registrieren. Bei Änderung des Status (z. B. Karte entfernt) erfährt die Applikation davon durch ein Operation Event Object.

Diese Architektur erlaubt es, die Applikationen in Form von State-Machines zu implementieren.

Voraussetzung für eine solche eventgetriebene Architektur ist jedoch, dass die Aktionen kurz und nicht-blockierend sind, da die einzelnen Aktionen kooperativ sein müssen: Es gibt keine automatische Preemption, da nur ein Thread und eine Applikation vorhanden sind.

Abschließend bleibt zur Applikationsarchitektur noch zu sagen, dass FINREAD sogar ein eigenes Auslieferungsformat definiert: JEFF statt JARs. Die Gründe dafür bestehen darin, dass JAR-Files nicht unmittelbar ausführbar sind und Plattformen mit wenig Speicher deshalb doppelten Speicher zum Auspacken und Installieren benötigen.

Literatur

- [Block] M. Block, Konzeption und Implementierung eines JXTA OSGI Bundles für ein Residential Gateway, Diplomarbeit HdM 2005
- [FR_Arc] Financial Transactional IC card reader (FINREAD) – Part 4: Architectural Overview, erhältlich unter http://www.finread.com/pages/about/specifications/01_specifications.html
- [FR_Sec] Financial Transactional IC card reader (FINREAD) – Part 3: Security Requirements, erhältlich unter http://www.finread.com/pages/about/specifications/01_specifications.html
- [Kut] R. Kutschke, Konfigurationsmanagement in Heimnetzen: Erstellung eines Konzepts mit Hilfe semantischer Modelle (OWL-S) und der Verwendung der Spezifikationen TR-069 und OMA DM, Diplomarbeit HdM 2006
- [Mil] M.S. Miller, Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control, Dissertation, Johns Hopkins University, Baltimore, online unter <http://www.erights.org/talks/thesis/index.html>
- [OSGI] OSGI Technical Specification R4, www.osgi.org
- [Schn99] B. Schneier, A. Shostack, Breaking Up Is Hard To Do: Modeling Security, Threats for Smart Cards, USENIX Workshop on Smart Card Technology, USENIX Press, 1999, pp. 175–185, online unter <http://www.schneier.com/paper-smart-card-threats.html>
- [SB] L. Shankar, S. Burns, Demystifying class loading problems, Part 1–4, <http://www-128.ibm.com/developerworks/java/library/j-dclp1/index.html>
- [STIP] Small Terminals Interoperability Platform, www.stip.org

Kapitel 9

Web-Services und objektbasierte Sicherheit

Wir haben nun die objektbasierte Sicherheit bereits mehrfach gestreift: Das Beispiel zu den Proxy-Zertifikaten im Rahmen der Grid-Security zeigte, wie sich Delegation explizit und nachprüfbar gestalten lässt. Im Bereich des föderativen Identitätsmanagements spielen signierte Token und die Secure Association Markup Language (SAML) [SAML] eine große Rolle. Und nicht zuletzt das E-Voting-Beispiel zeigt, wie mächtig objektbasierte Sicherheit sein kann und wie sie prozedurale Sicherheit weniger wichtig werden lässt. Aber auch im Bereich Enterprise Security leisten Token zunehmend Dienste als Träger von Sicherheitsaussagen.

Dennoch begeben wir uns mit objektbasierter – oder auch message-basierter – Sicherheit auf einen Pfad, der entgegengesetzt zur softwaretechnischen Behandlung von Sicherheit in den vergangenen Jahren verläuft. Statt Sicherheit nämlich in der Infrastruktur zu verstecken, heben wir sie jetzt wieder auf den Level der Applikation: Objektbasierte Sicherheit spielt sich auf der Anwendungsebene ab. Ihre Objekte sind dort bekannt und Teil des Business.

Grob gesagt bedeutet objektbasierte Sicherheit die Anwendung von Signatur und Verschlüsselung auf Texte und Nachrichten, sodass sie an mehrere mögliche Empfänger geschickt und dort authentisiert werden können (wohlgemerkt – die Nachricht wird authentisiert, nicht der Sender).

Objektbasierte Sicherheit ist in hohem Maße von darunter liegenden Infrastrukturen unabhängig und erlaubt daher neue und flexiblere Modelle der sicheren Collaboration. Es sind diese Modelle, die das Ganze treiben, und daher macht es Sinn, mit ihnen als Grundlage anzufangen, anstatt mit dem Grab an XML-xxx-, WS-xxxx- und OASIS-xxxx-Standards, das sich darum unter dem Einfluss von Gremien und Organisationen wie IBM/Microsoft, W3C, Oasis, Liberty Alliance etc. entwickelt hat.

Wir werden uns auch hier wieder auf den größeren Zusammenhang beschränken und einige besonders wichtige Prinzipien hervorheben. Für die Details verweisen wir auf [Wee], speziell die Teile 6 und 7, die sich mit Security beschäftigen. Dort beschreiben Web-Servicearchitekten sämtliche wichtigen Standards unter besonderer Berücksichtigung der Konzepte – offensichtlich war auch bei den

Machern der Web-Servicesarchitektur ein Unbehagen über die Vielzahl der WS-xxxx-Standards entstanden, die den Blick auf die Grundkonzepte verstellen. Eine spezielle Diskussion der Sicherheitsaspekte von Web-Services findet sich bei O'Neill im Buch „Web Services Security“ [ONeill03].

9.1 Modelle sicherer Kollaboration

Kontrastieren wir zu Beginn kurz einen Kaufvorgang am Internet mit verschiedenen Geschäftsvorgängen im Alltag und vergleichen die Sicherheitstechniken und Ergebnisse:

Der normale Kaufvorgang im Internet beginnt mit einem Klick auf einen Link, der zu einem Web-Shop führt. Dort muss ein Account für den Kunden eingerichtet werden. Danach können Waren angeklickt werden und damit eine Bestellliste (Warenkorb) erstellt werden. Anschließend hinterlegt man Bank- oder Kreditkarteninformationen und drückt auf einen Knopf mit der Aufschrift „Bestellen“. Man kann dann die Warenliste ausdrucken und erhält einige Zeit später die Ware.

Im realen Leben würden wir uns kaum mit der geringen Sicherheit dieses Vorgangs zufrieden geben – weder als Käufer noch als Verkäufer: Keiner der Beteiligten hat non-repudiation (Nicht-Abstreitbarkeit) erreicht. Der Käufer kann behaupten, dass er nichts gekauft habe. Der Verkäufer hat als Nachweis der Bestellung nichts als ein paar Log-Einträge in der Hand, die keinen rechtlichen Beweis darstellen (schließlich kann er diese Daten auch selbst erzeugt haben). Selbst wenn er die Authentisierung, die der Kunde durchgeführt hat, im Nachhinein nachweisen könnte (indem sie z. B. durch einen vertrauenswürdigen Dritten erfolgt), so hätte er immer noch keine Möglichkeit, die Bestellung zu beweisen.

Umgekehrt hat aber auch der Kunde nichts in der Hand (außer einem selbst ausgedruckten Zettel), das einen Beweis für die Bestellung darstellen würde. Der Verkäufer kann immer sagen, dass er keine Bestellliste empfangen hat.

Im realen Leben hätte der ganze Vorgang Unterschriften benötigt, und zwar unter einer klaren Aussage bezüglich Ware und Preis, Datum und Ort etc: Egal, ob der Kunde nun vergesslich oder der Händler schlampig ist: Die Sicherheit des Auftrags und der durchgeführte Kauf sind im Objekt des Kaufvertrages festgehalten.

Aber die Unterschiede gehen noch weiter: Wenn der Händler weitere Zulieferer benötigt, kann er ihnen den unterschriebenen Vertrag als Garantie zeigen – in der elektronischen Version kann er die Authentisierung des Kunden nicht an andere weitergeben, ohne dass massives Vertrauen auf Seiten der Empfänger nötig wäre.

Alternativ könnte der Kunde im realen Leben dem Händler explizite Vollmachten erteilen – eingeschränkt auf den Auftrag und unterschrieben durch den Kunden. Implizit sind darin Angaben zur Identität des Kunden enthalten (Ort, Name etc.). Eine elektronische Form davon haben wir bereits in Form des Proxy-Zertifikats kennen gelernt. Hier nun eine Liste weiterer interessanter sicherheitsrelevanter Aktionen aus dem täglichen Leben, die normalerweise reibungslos funktionieren:

- Bei schwierigen Transaktionen benötigen wir eventuell einen Treuhänder, der Dinge in Empfang nimmt und bestätigt. Wieder sind hierfür Aussagen und Unterschriften nötig.
- Wir versenden Berechtigungen an Personen (z. B. PIN/TAN-Listen etc.). Ein Teil der Nachricht ist für den Briefträger, aber auch für Bedienstete im eigenen Unternehmen lesbar. Ein anderer Teil ist nur für den Empfänger bestimmt und verdeckt. Teile verstecken zu können, ist eine wichtige Eigenschaft von Nachrichten im realen Leben.
- Wir verkaufen/erwerben/verwenden/anerkennen Gutscheine (Tickets, Geschenke etc.). Wir liefern Waren oder Dienstleistungen gegen signierte Bescheinigungen aus, ohne deren Überbringer zu authentisieren (wenn wir nicht im rechtsfreien Fußballuniversum der FIFA operieren).
- Wir bitten Dritte um eine Bestätigung einer Aussage oder eines Sachverhalts und erhalten eine signierte Bescheinigung, ausgestellt durch eine Autorität.
- Wir makeln zwischen Käufer und Verkäufer und geben dabei nur Stück für Stück die benötigten Informationen frei – beispielsweise nennt ein Immobilienmakler einem potenziellen Käufer erst dann die genaue Adresse des Objekts, wenn er auch eine Adresse des potenziellen Käufers hat.
- Wir tauchen bei einem Händler auf, der uns noch nie gesehen hat, und erwerben mit unserer Kreditkarte eine Ware – ohne dafür einen Account bei ihm eröffnen zu müssen.
- Wir beschreiben die Art und Weise, wie wir mit anderen zusammenarbeiten wollen, in „Allgemeinen Geschäftsbedingungen“. In ihnen ist rechtsverbindlich z. B. die Art und Weise der Sicherheit der Transaktionen (Vorkasse, Treuhänder etc.) festgehalten.

Diese Liste ließe sich durch Vorgänge, wie sie im Handelsrecht, im BGB, im Erbrecht etc. beschrieben sind, beliebig erweitern. Die Beispiele zeigen, dass geschäftliches wie privates Handeln in der Realität

- kollaborativ ist, d. h. oft mehr als zwei kooperierende Beteiligte einschließt.
- häufig unter den Beteiligten eine Autorität hat, die von allen anderen anerkannt wird.
- auf Unterschriften beruht, die uns Aussagen akzeptieren lassen, die weitergegeben werden können und die vor allem für Nachweisbarkeit sorgen.
- auf Bescheinigungen beruhen, die auch ohne die Anwesenheit des Ausstellers oder des Begünstigten wirksam sind.
- auf der Tatsache beruht, dass alle Beteiligten die Aussagen im unterschriebenen Dokument verstehen.
- Teile von unterschriebenen Aussagen produziert, die nicht allen Beteiligten zugänglich sein sollen (oder nur zeitlich verzögert zugänglich sind).

Welche Bedeutung haben diese flexiblen, kollaborativen Sicherheitsmechanismen im realen Leben? Es gibt Untersuchungen darüber, dass die Absicherung des Risikos bei Geschäften durch Gesetze, Prozeduren etc. einen wesentlichen Vorteil entwickelter Länder ausmachen. Ihr Fehlen behindert Gesellschaften in wirtschaftlicher Hinsicht enorm. Die Vertreter der objektbasierten Sicherheit – speziell die Web-Servicesanhänger – glauben, dass unsere momentanen Sicherheitsmechanismen im Internet, die überwiegend auf kanalbasierter Sicherheit (oder Transportsicherheit) beruhen, uns ebenfalls in den geschäftlichen und privaten Transaktionen behindern.

Tatsächlich lassen sich die oben genannten Beispiele mittels Transportsicherheit allein (z. B. durch SSL) nicht durchführen. In der Praxis wird eine notwendige Delegation zu einem nachgeordneten Server oft so realisiert: „Glaube mir, ich arbeite für Kunde XXXX“ oder „Hier hast du UserID und Passwort des Kunden“.

Das entspricht im wirklichen Leben dem bereits diskutierten Beispiel, dass jemand seinen Nachbarn beauftragt, Geld von der Bank mitzubringen, und dieser dafür mit EC-Karte und PIN des Eigentümers ausgestattet wird.

Im folgenden Beispiel wollen wir einige wichtige Merkmale der objektbasierten Sicherheit herausarbeiten. Dabei geht es uns darum, die Änderungen in der Bedeutung der Infrastruktur durch objektbasierte Sicherheit zu untersuchen. Wir sprechen über das sogenannte *E-Voting*, also über die Möglichkeit, Wahlen per Rechner automatisiert durchzuführen und auszuwerten. Da nach eigener Erfahrung viele Entwickler sehr stark an der prozessorientierten Sicht von Autorisierung und Kontrolle hängen („wenn alle Module getestet sind und in der richtigen Reihenfolge aufgerufen werden, dann ist auch das Ergebnis korrekt“), macht dieses Beispiel auf drastische Weise den Gegensatz von prozessorientierter Sicherheit und Ende-zu-Ende-Sicherheit deutlich.

9.1.1 Fallstudie: *E-Voting*

Wir haben im Kap. 4 zur Plattformensicherheit das Prinzip der „complete mediation“, d. h. der Zugriffskontrolle bei allen Schritten, kennen gelernt. Unklar war jedoch, inwieweit dieses Prinzip auch Gültigkeit im Falle einer Ende-zu-Ende-Sicherheit auf Basis von Signaturen und Verschlüsselung besitzt. Wir wollen diesen Fall anhand der heiß diskutierten Frage der Sicherheit von Wahlmaschinen für das sog. *E-Voting* untersuchen.

Die Firma Diebold hat in den USA mit ihren äußerst unsicheren Wahlmaschinen traurige Berühmtheit erlangt. Gestützt auf 4,5 Milliarden Dollar aus dem Staatshaushalt wurden die Kommunen aufgefordert, diese Maschinen zu kaufen. Trotz der Ausstattung mit Smartcards erwies sich die gesamte Sicherheit der Systeme als sehr brüchig [KSRW].

Die Firma berief sich angesichts vernichtender Sicherheitsanalysen auf den Prozess der Erstellung und Abnahme der Maschinen als Garant für deren Korrektheit (dass die Firma nicht einmal dies korrekt durchführte und z. B. noch kurz vor Wahlen Firmware austauschte, sei nur am Rande erwähnt). Diebold lehnte vehe-

ment einen sog. Paper-Trail – also einen gedruckten Beleg eines Wahlvorgangs – mit der Begründung ab, dass Drucker in Wahllokalen viel zu anfällig wären und die gesamten Wahlen gefährden würden – dabei können Sie in jedem Supermarkt selbst für eine einzelne Banane eine gedruckte Quittung erhalten, und das scheint die Supermärkte weder finanziell noch wartungstechnisch zu überfordern. Würden diese Paper-Trails im Wahllokal gesammelt, könnte man am Ende des Auszählvorgangs stichprobenartig verschiedene Wahllokale auswählen, deren Paper-Trails manuell auszählen und mit dem elektronischen Ergebnis vergleichen. Auch dabei wären zwar gewisse Toleranzen möglich, da die manuelle Auszählung ebenfalls fehlerträchtig ist, aber dadurch hätte man zumindest eine gewisse Kontrollmöglichkeit. Anderenfalls beruht das Vertrauen in die Richtigkeit des Wahlergebnisses (eine wesentliche Voraussetzung für das Funktionieren von Demokratie) vollständig auf dem Vertrauen in den Prozess der Erstellung der Software und des Gesamtsystems. Peinlicherweise wurde jedoch die Software durch ein Versehen öffentlich zugänglich und prompt zeigten sich gravierende Mängel. So erwiesen sich die Smartcards als funktionslos und nur der Beruhigung dienend.

Diebold hatte natürlich recht, jeden manuellen Überprüfungsversuch abzulehnen: In einem System, dessen Vertrauenswürdigkeit auf dem Glauben an die Richtigkeit des Prozesses beruht, ist jeder Verdachtsmoment natürlich tödlich – denn er beseitigt gerade das Vertrauen, auf das das System letztlich baut (auch die EC-Kartensysteme der Banken beruhen auf dem Vertrauen in die Richtigkeit und Zuverlässigkeit der Prozesse innerhalb der Bank und der beteiligten Maschinen. Diese werden natürlich geprüft und überwacht, aber letztlich steckt das Vertrauen im Prozess, und der könnte fehlerhaft sein.). Somit gehören Unternehmen (meist Unternehmensberatungen), die Zertifizierungen solcher Prozesse durchführen, als weiterer Bestandteil des Glaubenssystems einfach dazu.

Wie sichert man also nun Wahlmaschinen ab? Die Antwort könnte lauten: Fast gar nicht. Letztlich muss jede Stimme richtig gezählt und ein korrektes Gesamtergebnis erzeugt werden. Ein System, das auf kryptografischen Verfahren beruht, könnte ungefähr folgendermaßen funktionieren:

Der Wähler verschlüsselt seine Stimmabgabe mit zwei öffentlichen Schlüsseln: Der eine gehört dem Wähler, der andere der Wahlleitung (die Zertifikate für die Wähler könnte ebenfalls die Wahlleitung ausstellen, analog zu den in Deutschland versendeten Wahlbenachrichtigungen). Der Wähler erhält nach der Stimmabgabe einen gedruckten Beleg, der seine verschlüsselte Stimme enthält. Alle mit dem öffentlichen Schlüssel der Wahlleitung verschlüsselten Stimmen werden elektronisch zur Wahlleitung übermittelt und auf einer Website dargestellt. Nach der Wahl kann die Wahlleitung mit ihrem Schlüssel die Stimmen entschlüsseln und die Stimmen der Kandidaten zählen. Auch Privatpersonen können ihre persönliche Stimme mittels ihres privaten Schlüssels prüfen. Somit kann gewährleistet werden, dass alle Stimmen korrekt gezählt wurden.

Natürlich stellen sich bei der Realisierung dieses Systems weitere Probleme; so muss z. B. verhindert werden, dass die Wahlleitung künstliche Stimmen erzeugt. Auf der anderen Seite ist keine Wahl – weder elektronisch noch in der wirklichen Welt – ohne ein Mindestmaß an Vertrauen in die Wahlleitung durchführbar. Auch

müssen geeignete Vorkehrungen dafür getroffen werden, dass die Stimmabgabe weiterhin anonym erfolgen kann. Das größte Problem ist jedoch die Verhinderung des Stimmenkaufs, da nun die Stimmabgabe durch einen Wähler vom Käufer der Stimmer nachgeprüft werden kann. Das gleiche gilt allerdings auch für die Briefwahl in einem papierbasierten System. Ein elektronisches Wahlsystem müsste hier garantieren, dass eine massenhafte Durchführung von Stimmenkäufen auf elektronischem Wege am Aufwand scheitert.

Mit dieser „Ende-zu-Ende“-Kontrollmöglichkeit der Stimmabgabe stellt sich die Frage nach der Sicherheit der dabei beteiligten Plattformen wesentlich entspannter: Es geht gar nicht mehr so sehr um die Fälschungsgefahr, sondern um die größtmögliche Verfügbarkeit für die Wähler. Eine „Complete Mediation“ im Sinne von kleinstgranularen Zugriffskontrollen ist dadurch gar nicht mehr nötig.

9.1.2 Grenzen der kanalbasierten Sicherheit

Bevor wir mit der objektbasierten Sicherheit ein ganz anderes Verfahren einführen, lassen Sie uns noch einmal kurz die wesentlichen Sicherheitsmerkmale kanalbasierter Sicherheit zusammenfassen (wer noch mehr zu kanalbasierter Sicherheit erfahren möchte, der sei auf den Band „Internet-Security aus Software-Sicht“ [KS] verwiesen. Dort haben wir in unseren beiden Fallstudien zu ebay.com und bahn.de Geschäftsmodelle vorgestellt, die weitgehend kanalbasierte Sicherheit verwenden, d. h. üblicherweise einen SSL-Tunnel zwischen Web-Server und Kundenbrowser herstellen). Dabei haben wir auch festgestellt, dass sich damit keine Nicht-Abstreitbarkeit der getätigten Transaktionen erreichen lässt. Weder der Verkäufer noch der Käufer erhält eine – rechtssichere – Bestätigung der Bestellung. Aber selbst wenn man meint, auf dieses Sicherheitsmerkmal verzichten zu können, stößt die kanalbasierte Sicherheit schnell an Grenzen, wenn weitere Partner in die Transaktion mit einbezogen werden sollen: Wie kann die Tatsache einer erfolgten Authentifikation eines Kunden oder dessen aktueller Status als Kunde auf sichere Art und Weise von einer Firma an die nächste transferiert werden, wenn nicht vorher schon eine Vertrauensbeziehung zwischen diesen bestand? Warum sollte Firma *A* der Firma *B* vertrauen, was die Durchführung der Authentisierung betrifft? Auch wenn die Weitergabe des Sicherheitskontexts gelingt, können Unterschiede in der Infrastruktur der Unternehmen dafür sorgen, dass die darin enthaltenen Informationen unterschiedlich interpretiert werden. Die Komplexität steigt immer weiter an, wenn weitere Partner ins Spiel kommen, und lässt sich eigentlich nur dadurch auflösen, dass alle Beteiligten einer zentralen Autorität vertrauen.

9.2 Objektbasierte Sicherheit

Was bedeutet nun „Object Security“ im Gegensatz zu kanalbasierter Sicherheit? Letztlich bedeutet Object Security die Möglichkeit, sichere Aussagen über ein

Dokument machen zu können – und zwar angesichts des Dokuments allein, ohne eine Form von sicherem Kanal oder einer authentisierten Sitzung, die zum Transport des Dokuments verwendet wurden. Unter diese sicheren Aussagen fallen:

- Wer ist der Autor des Dokuments?
- Ist die Information integer, d. h. ist sie nur vom Autor manipuliert worden?
- Ist die Information vertraulich, d. h. kann sie nur von einem bestimmten Empfänger gelesen werden?

Die entscheidenden Sicherheitstechniken, um die Object Security zu erreichen, sind die elektronische Signatur des Dokuments sowie deren Verschlüsselung. Hier noch einmal zur Erinnerung die entscheidenden Bestandteile einer elektronischen Signatur:

- die genaue Bestimmung der signierten Information,
- der verwendete Hash-Algorithmus,
- das Public-Key-Verfahren, mit dem der Hash-Wert verschlüsselt wurde und
- die beteiligten Schlüssel des Signierers.

Der private Schlüssel dient zum Unterschreiben und ist nur dem Signierer bekannt, der öffentliche Schlüssel dient auf Seiten des Empfängers zum Entschlüsseln des Hash-Werts und so zum Überprüfen der Autorschaft.

Eine digital signierte Bestätigung einer vertrauenswürdigen Instanz (CA), die die Zugehörigkeit des öffentlichen Schlüssels zur Identität des Signierers bescheinigt.

9.2.1 XML-Signaturen

Ein großer Teil der Informationen im Internet liegt heute in Form von XML-Dokumenten vor. Deshalb war es wichtig, einen Standard für das Signieren von XML-Dokumenten zu finden, der Signaturen über XML-Dokumente interoperabel und universell verifizierbar macht. Dieser Standard ist der XML Digital Signature Standard des W3C, im Folgenden als XML-DSIG abgekürzt. Er besteht im Wesentlichen aus einer Umsetzung der obigen Signaturbestandteile in Elemente und Attribute in XML [XML-Sig]. Er erlaubt es, nicht nur ein Dokument als ganzes zu signieren (*Enveloping Signature*), sondern auch nur gewisse Teile des Dokuments (*Enveloped Signature*), oder auch, ein Dokument zu signieren, für das lediglich eine eindeutige URI angegeben wird (*Detached Signature*). Abbildung 9.1 zeigt die dabei verwendeten XML-Elemente.

Der Standard definiert also keine neuen Techniken, um Signaturen zu erstellen, sondern ermöglicht nur die Anwendung von Signaturen auf XML-Dokumenten. Damit können zwei der Forderungen für Object Security abgedeckt werden: Über die Signatur ist die Autorenschaft sowie die Unversehrtheit des Dokumentes prüfbar. Um eine möglichst große Interoperabilität gewährleisten zu können, werden die XML-Dokumente vor der eigentlichen Signatur noch in eine sogenannte *kano-nische Form* gebracht (Abb. 9.2).

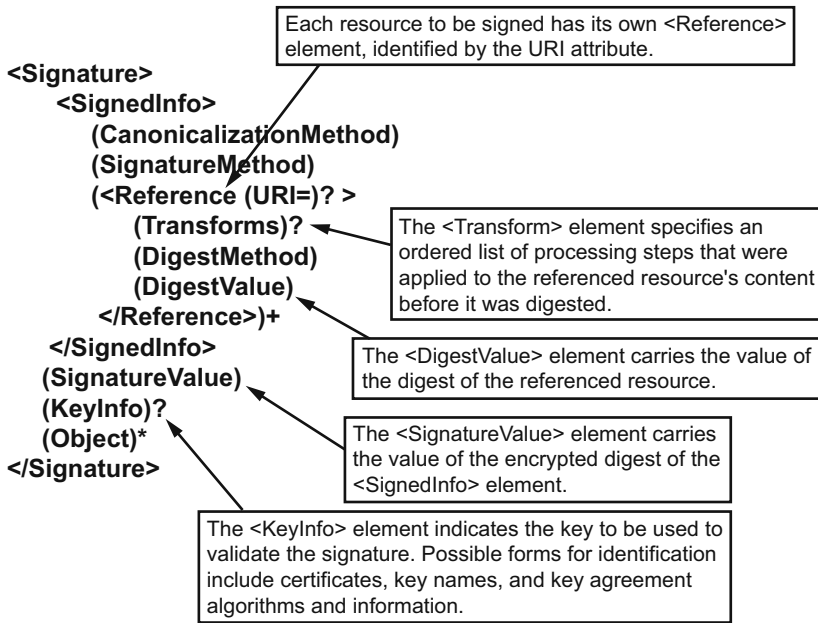


Abb. 9.1 XML-Elemente für das Signieren von Dokumenten

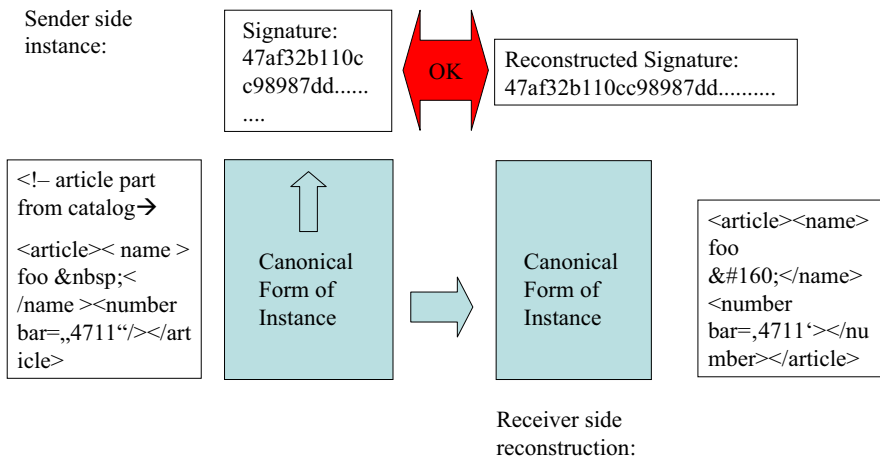


Abb. 9.2 Kanonische Form von XML zur signierten Übertragung

9.2.2 XML-Verschlüsselung

Nach dem gleichen Muster wie die Anwendung digitaler Signaturen erfolgt auch die Verschlüsselung von XML-Dokumenten. XML-Dokumente sollen ganz oder

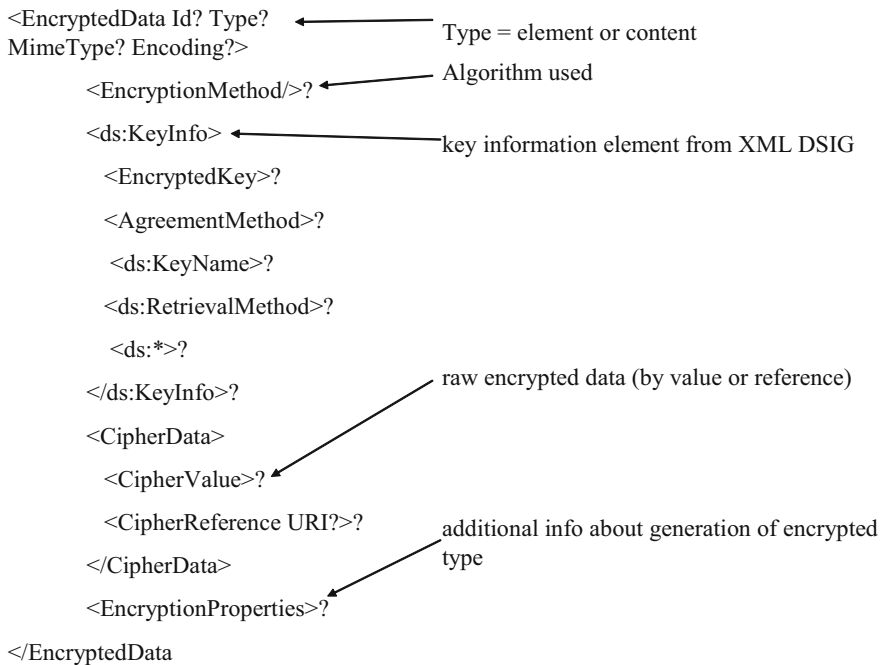


Abb. 9.3 XML-Verschlüsselung und ihre Elemente

teilweise verschlüsselt werden können, und die verschlüsselte Information sowie die Informationen, die zum Entschlüsseln nötig sind, sollen Bestandteil des Dokumentes werden. Dafür sind nötig:

- Informationen zum verwendeten Verschlüsselungsalgorithmus,
- Informationen zum verwendeten Schlüssel und
- die verschlüsselten Daten selbst und deren genaue Bestimmung.

Umgesetzt in XML-Elemente und -Attribute ergibt dies das in Abb. 9.3 dargestellte Schema.

Der zugehörige Standard heißt XML-Encryption [XML-Enc] und bezieht sich in Teilen wieder auf [XML-Sig], um doppelte Definitionen von Elementen wie dem „Key Information Element“ zu vermeiden. Zentraler Mechanismus des Teilens von Definitionen sind dabei die bekannten XML-Namespaces-Definitionen, die hier nicht nur zur Identifikation eines Schemas und seiner Elemente, sondern auch der Definition einzelner Werte oder Konstanten dienen.

Mit diesen Bausteinen lassen sich bereits Non-Repudiation-Eigenschaften, z. B. bei Warenbestellungen, aufbauen: Der Kunde schickt einen signierten Auftrag an den Händler, dieser antwortet mit einer signierten Auftragsbestätigung, deren Erhalt wiederum vom Kunden (einschließlich einer Referenz auf den ursprünglichen signierten Auftrag) signiert und zurückgeschickt wird.

9.2.3 Interoperabilität

Solange wir in unserer eigenen Umgebung – sprich bei unseren eigenen Werkzeugen – bleiben, tauchen keinerlei Interoperabilitätsprobleme bei unseren XML-Dokumenten auf. Aber sobald wir anfangen, signierte und verschlüsselte XML-Dokumente zu versenden, tauchen einige unangenehme Probleme auf.

- Mangelnde Interoperabilität (bezogen auf Format, verwendete Algorithmen und Empfängerpräferenzen) und
- ungelöstes Key-Management.

Beginnen wir mit der mangelnden Interoperabilität und dem Rückgriff auf ein altes und an anderer Stelle bereits gelöstes Problem: Dem Signieren von Datenstrukturen, die für den Transport über Netzwerke bestimmt sind. Die Abstract Syntax Notion One (ASN.1), eine Sprache, die ähnlich wie XML strukturierte Informationen zur Verwendung in Kommunikationsprotokollen beschreibt, musste frühzeitig feststellen, dass die lockere Beschreibung von Datenstrukturen in den Basic Encoding Rules (BER) dazu führte, dass auf der Empfängerseite das übertragene und signierte Dokument anders repräsentiert wurde (obgleich kompatibel zu BER). Daraufhin schlug natürlich die Prüfung der Signatur – die auf binären Daten und nicht auf der logischen Struktur des Dokuments arbeitet – fehl. In ASN.1 wird das Problem durch spezielle Encodier-Regeln, den Distinguished Encoding Rules (DER) gelöst, welche die Struktur der Daten bitweise vorschreiben.

XML besitzt die gleiche Problematik der unterschiedlichen Abbildung einer logischen Struktur auf verschiedene physische Strukturen. Da aber nur die physischen Strukturen signiert werden, sind die Signaturen bei gleichem logischen Inhalt des Dokuments leider häufig verschieden. Betrachtet man den DOM-Tree eines Dokuments als „Abstract Syntax Tree“ (AST), dann erlaubt XML verschiedene Grammatiken zur Serialisierung dieses AST (Abb. 9.4).

Die Lösung im XML-Bereich ist die gleiche wie damals bei ASN.1, nämlich die Definition eines kanonischen Formats für die Datenübertragung bei angewandter Signierung: XML-Canonicalization [XML-Can], ein weiterer Standard des W3C. Softwareentwickler dürften in den meisten Fällen glücklicherweise vom Umgang damit verschont bleiben, da die Erstellung einer kanonischen Form in XML-Frameworks bereits eingebaut sein müsste – zumindest sollte dies bei Java oder bei .NET der Fall sein. Im Bereich Embedded Control, wo meistens mit C oder C++ gearbeitet wird, kann die Lage hier insgesamt schlechter aussehen.

Nachdem das Formatproblem gelöst ist – und der Standard XML-Sig um ein weiteres Element – die verwendete Methode der Kanonisierung – erweitert ist, taucht ein weiteres Problem auf: Es könnte sein, dass wir beim Signieren und Verschlüsseln Verfahren eingesetzt haben, die der Empfänger (vielleicht ein schwächerer PDA oder ein winziges Handy) nicht beherrscht. Damit kann unser Dokument dort nicht geprüft oder entschlüsselt werden.

Wir brauchen also über die Sicherheit des Objektes selbst hinaus weitere Regeln und Techniken, wenn wir unsere Dokumente interoperabel zwischen ver-

DTD/Schema:

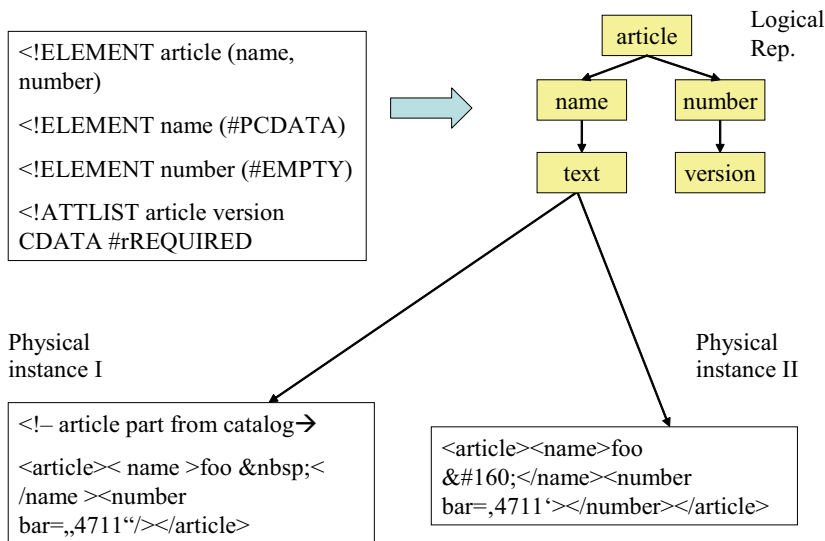


Abb. 9.4 Verschiedene physische Repräsentationen eines logischen Modells

schiedenen Teilnehmern versenden wollen. Dazu müssen wir von den Fähigkeiten und Präferenzen unserer Mitspieler in Bezug auf Security erfahren. Diesen Problemen wenden wir uns in den nächsten Abschnitten zur messagebasierten Sicherheit, speziell Web-Services, zu.

Zuvor müssen jedoch noch weitere Probleme im Zusammenhang mit unseren Schlüsseln gelöst werden. Wir sind bislang der Frage aus dem Weg gegangen, wie und woher die Schlüssel kommen (vielleicht aus einem Java Keystore?), wie sie kommuniziert werden (über Zertifikate?) und wie sie validiert werden. Das W3C hat auch dazu einen Standard [XML-KMS] entwickelt. Im Zusammenhang mit Web-Services werden wir diese Problematik noch vertiefen, denn dort sind ebenfalls Services und Standards dazu entstanden.

Diskutieren wir am Ende noch kurz mögliche Fehler von Entwicklern im Zusammenhang mit der Signierung und Verschlüsselung von Dokumenten. Generell sollten Dokumente erst signiert und dann verschlüsselt werden (analog einem Brief in der wirklichen Welt, den man erst unterschreibt und dann in ein Kuvert gibt). Dies hat weniger sicherheitstechnische als juristische Gründe: Bei einer Unterschrift, die verbindlichen Charakter haben soll, sollte der Inhalt dessen, was unterschrieben wird, auch tatsächlich im Klartext für den Signierer zu sehen sein. Weitere mögliche Fehler betreffen die Teile des Dokuments und seiner Meta-Daten, die signiert werden. Hier ist große Vorsicht geboten: Alle Daten, die unsigniert bleiben, können von einem Angreifer unbemerkt manipuliert werden. Besonders übel kann sich auch die Vermengung objektbasierter und kanalbasierter Si-

cherheitsmaßnahmen auswirken: Der Autor des Dokuments geht von einer durch SSL gesicherten und verschlüsselten Übertragung der Informationen aus und belässt die Geheimnisse deshalb im Klartext. Der Sender des Dokuments wiederum nimmt objektbasierte Sicherheit an und verzichtet deshalb auf den sicheren Kanal.

Schließlich und endlich besteht natürlich auch die Gefahr korumpierter (öffentlicher) Schlüssel, wenn sie nicht durch eine Nachfrage bei der ausstellenden CA validiert werden.

9.3 SOAP Security

Wir haben im vorherigen Abschnitt erkannt, dass zwischen Objektsicherheit und der sicheren Übertragung des Objekts als Nachricht ein Unterschied besteht. Neben Interoperabilitätsproblemen mit unterschiedlichen Formaten und Algorithmen taucht jetzt auch das Problem auf, dass ein Dokument typischerweise mit Metadaten für den Versand versehen wird. Auch übertragen wir nicht unbedingt immer Dokumente, sondern häufig auch signierte Minitexte, die sicherheitsrelevante Aussagen beinhalten, sogenannte *Token*. Ein Beispiel für solche Token sind SSO-Token, Zertifikate, oder UserID/Passwort-Kombinationen.

Ein gebräuchliches Format für das Senden von Messages zwischen Anwendungen über das Internet ist SOAP, das Basisformat der Web-Services, das wir im Buch „Internet-Security aus Software-Sicht“ [KS] in Verbindung mit anderen Middleware-Protokollen wie CsiV2 von CORBA beschrieben haben. SOAP folgt dem aus HTML geläufigen Header/Body-Muster und bettet beides in einen Umschlag (Envelope) ein. Für uns ist besonders das Security-Element interessant, das Teil des Headers einer SOAP-Nachricht ist und mit dem verschiedene sicherheitsrelevante Informationen wie X.509-Zertifikate, UserIDs, verschlüsselte Passwörter usw. im Rahmen eines XML-Dokuments transportiert werden können.

9.3.1 WS-Security

Wie also bewältigt man nun die Übertragung von signierten und/oder verschlüsselten Informationen und Token über SOAP? Die Antwort darauf lautet WS-Security – ein ursprünglich von IBM, Microsoft und Verisign definierter Standard, der mittlerweile von OASIS weiterentwickelt und verwaltet wird. WS-Security ermöglicht *secure messages* – also die Authentifizierung von Nachrichten und deren Token, jedoch nicht die Authentisierung einer Session bzw. eines Senders.

Im Detail beschreibt WS-Security, wie die oben beschriebenen Elemente zur Signatur und Verschlüsselung von XML-Dokumenten in den Header einer SOAP-Nachricht eingebettet werden können sowie die Übertragung von binären Token. Token spielen in den Geschäftsmodellen der Web-Services eine zentrale Rolle

(wie auch im normalen Leben) und deshalb schließt WS-Security ein Standardformat für deren Übertragung ein, das *UserName Token*:

```
<Security>
<UsernameToken ID="...">
<Username>...</Username>
<Password
type="...">...</Password>
</UsernameToken>
...
```

Wie man sieht, wird das im Token enthaltene Passwort hier im Klartext übertragen. WS-Security empfiehlt daher, dass dieses Token nur in Verbindung mit einem sicheren Kanal verwendet werden sollte. Andere Token besitzen den Typ *BinarySecurityToken* und enthalten z. B. Zertifikate oder Kerberos-Tickets. Typischerweise werden dort auch Elemente vom Typ `<encrypted>` verwendet. Die Frage, woher diese (signierten) Token kommen, beantwortet weiter unten der Standard-WS-Trust mit Methoden zum Anfordern von Bestätigungen (also wiederum Token) sowie deren Prüfung.

Interessant sind die Schritte zur Authentisierung einer WS-Security-Nachricht auf Seiten des Empfängers: Am Anfang steht die sogenannte *Document-Key-Validation*, bei der der Hash-Wert über den signierten Teil des Dokuments mit dem verglichen wird, was man erhält, wenn man den öffentlichen Schlüssel des Signierers auf die Signatur im Dokument anwendet. Als zweiter Schritt folgt die *Key-Identity-Validation*: Wer ist der Autor des Dokuments, d. h. wer wird im Zertifikat als Inhaber des Signaturschlüssels genannt? Schließlich muss noch geprüft werden, ob das Zertifikat von einer vertrauenswürdigen CA stammt und ob es noch gültig ist (*Document-Key-Check*). Offen bleiben hingegen die Fragen, wer der Sender des Dokuments ist (dies muss nicht notwendigerweise der Autor sein) und wann die Signatur erfolgte.

Authentisiert werden kann die Nachricht also nur in Bezug auf den Autor und die Unverfälschtheit bzw. Vertraulichkeit selbst, aber nicht in Bezug auf die Identität des Senders. Wenn sich der Empfänger aus Sicherheitsgründen von der Übereinstimmung von Sender und Autor überzeugen muss, kann er dies durch ein Challenge-Response-Protokoll tun. Es bietet sich an, dabei als Challenge eine mit dem den öffentlichen Schlüssel des Autors verschlüsselte Zufallszahl zu verwenden.

WS-Security löst einen Großteil der oben erwähnten Probleme mit objektbasierter Sicherheit (Schlüsselmanagement, Aushandeln von Algorithmen) nicht. Zu diesen Fragen sind aber weitere WS-Standards entweder bereits definiert oder in Vorbereitung. Wir werden in diesem wie im Abschn. 9.3.3 einige davon (WS-Trust, SAML) näher kennen lernen.

Wenn wir nun nach den möglichen Angriffsformen fragen, dann muss sicher an erster Stelle die Replay-Attacke genannt werden: Ein Angreifer fängt eine signierte Nachricht ab und legt sie dem Empfänger nochmals vor. Dagegen helfen die bereits bekannten Mittel: Einführung von Zählern und/oder Zeitstempel (Time-

stamps). Wichtig ist aber die Erkenntnis, dass Replay-Attacken grundsätzlich zu den Gefahren objektbasierter bzw. nachrichtenbasierter Kommunikation gehören, da der Sender durch eine Signatur nicht zeitgleich authentisiert wird. Mindestens ebenso wichtig ist jedoch die bereits erwähnte Gefahr der Konfusion zwischen objektbasierter und kanalbasierter Sicherheit – jeder verlässt sich dabei auf den anderen und in der Folge werden Geheimnisse im Klartext oder wichtige Daten übertragen, ohne durch eine Signatur geschützt zu sein. Grundsätzlich kann aber bei richtiger und konsequenter Anwendung von objektbasierter Sicherheit auf die zu übertragenden Token auf eine kanalbasierte Sicherheit ganz verzichtet werden.

Auch SOAP-Fehlercodes können unerwünschte Informationen an potenzielle Angreifer beinhalten und sollten eventuell gefiltert werden. Damit stellt sich die Frage nach den Möglichkeiten einer vorgelagerten Authentisierung und Filterung von WS-Security-Messages. Das Framework `mod_security` für den Apache-Webserver leistet solche Dienste, wie auch eine Reihe kommerzieller sogenannter XML-Appliances, die eigentlich nichts anderes als intelligente Proxies darstellen, die XML und WS-Security-Standards verstehen und prüfen können. Um dies leisten zu können, müssen diese Proxies also Zugang zu den relevanten Schlüsseln zur Validierung der Nachrichten besitzen.

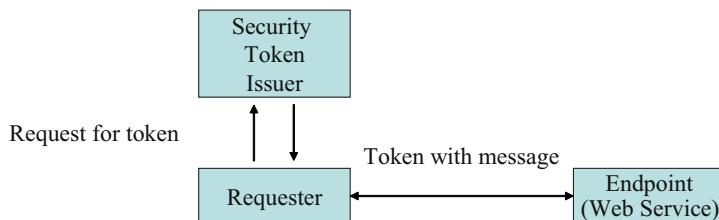
9.3.2 *Security Token Issuer (STI)*

Nun kommen wir zu den Problemen, die von WS-Security ungeklärt bleiben. Dazu gehört zum Beispiel die Frage, woher eigentlich die wichtigen Token kommen. Selbstverständlich können diese teilweise selber von einem Requester hergestellt werden, wie z. B. im Fall des Username Tokens, das ein Passwort enthält.

Web-Services favorisieren jedoch meist ein Modell, bei dem Token von einer Trusted Third Party – dem sogenannten *Security Token Issuer* ausgestellt werden.

Abbildung 9.5 zeigt das Basisszenario.

Dieses Modell beruht darauf, dass alle an Transaktionen Beteiligten entweder direkt oder indirekt (über Föderation von TTPs) den Ausstellern der Token vertrauen. Dies kann sogar sicherer sein, als eine direkte gegenseitige Authentisierung, z. B. von Mitarbeitern einer Firma bei Partnerfirmen. Dazu ein Hinweis: Wer denkt: „Was könnte sicherer sein, als eine direkte, möglicherweise starke Authentisierung eines Mitarbeiters einer Fremdfirma gegenüber dem eigenen System?“, der braucht nur an den Fall zu denken, dass der Mitarbeiter die Fremdfirma verlässt. Dann handelt er nicht mehr im Auftrag dieser Firma, kann aber mit seinen Authentisierungscredentials eventuell immer noch bei dem eigenen System Bestellungen durchführen, falls z. B. die Fremdfirma den Mitarbeiter nicht bei den anderen Firmen „abgemeldet“ hat, oder diese vergessen, den fremden Account zu löschen. Beispiele föderativen Identity-Managements finden sich u. a. im Band „Internet-Security aus Software Sicht“ [KS], der die Probleme der Identifizierung, Authentisierung und Authorisierung detailliert bespricht.



- no user authentication credentials in DB
- no changes when authentication mechanisms change
- technically and administratively easier
- perhaps safer

Abb. 9.5 Vorteile der Security-Token-Issuer-Architektur

Die Vorteile des Modells beinhalten:

- Wesentlich geringere Infrastrukturkosten beim Service Provider.
- Geringerer Aufwand für die Verwaltung von Credentials beim Requester.
- Geringere Verbreitung von Geheimnissen des Requesters (nur zur TTP).
- Die Möglichkeit, nicht nur Identitätsnachweise, sondern noch andere Bestätigungen auszustellen.
- Einen einfacheren, zentralisierten Wechsel auf andere Authentisierungsmethoden.

Bei den Nachteilen sind, wie immer bei tokenbasierten Mechanismen, Replay-Attacken und die Frage nach der Gültigkeitsprüfung der beteiligten Schlüssel und Bescheinigungen zu nennen.

Was uns nun noch fehlt, sind geeignete Methoden, um solche Token zu beantragen. Der Standard-WS-Trust löst dieses Problem, indem er Methoden zur Beantragung, zur Rücksendung und zur Validierung von Token spezifiziert – natürlich als Web-Services in der Sprache WSDL (Web Service Definition Language, [WSDL]) beschrieben. Abbildung 9.6 zeigt das grundsätzliche Szenario.

Ein Antragsteller (Requester) weiß aus der WSDL-Beschreibung eines gewünschten Web-Services, dass zu seiner Nutzung ein bestimmtes Security Token erforderlich ist. Er kontaktiert den Security Token Issuer, authentisiert sich (hierfür sind verschiedene Methoden wie Challenge/Response-Protokolle etc. möglich) und erhält im Gegenzug das gewünschte Token, mit dem er Zugang zum gewünschten Web-Service erhält. Mit dieser Vorgehensweise sind etliche Vorteile verbunden: Die Authentisierungscredentials des Requesters müssen nur an einem zentralen Ort gespeichert sein, während der Anbieter des Web-Services nicht wissen muss, wie die Authentisierung und Authorisierung beim Security Token Issuer abläuft.

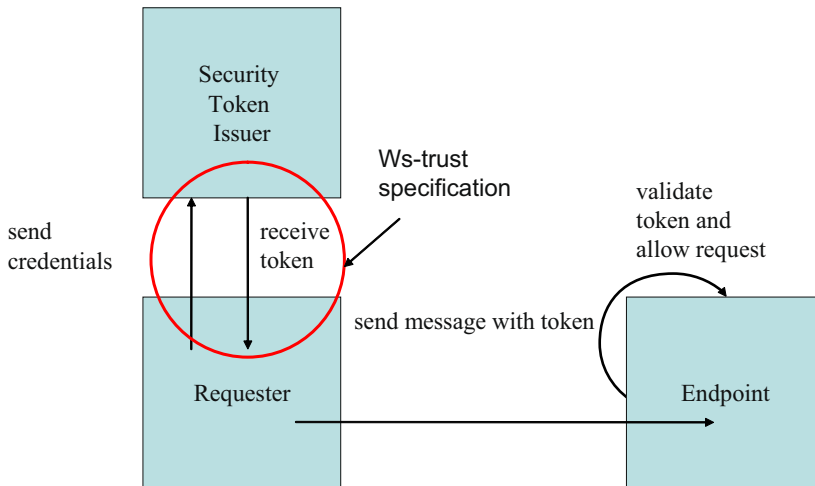


Abb. 9.6 Anfordern und Prüfen von Token durch WS-Trust

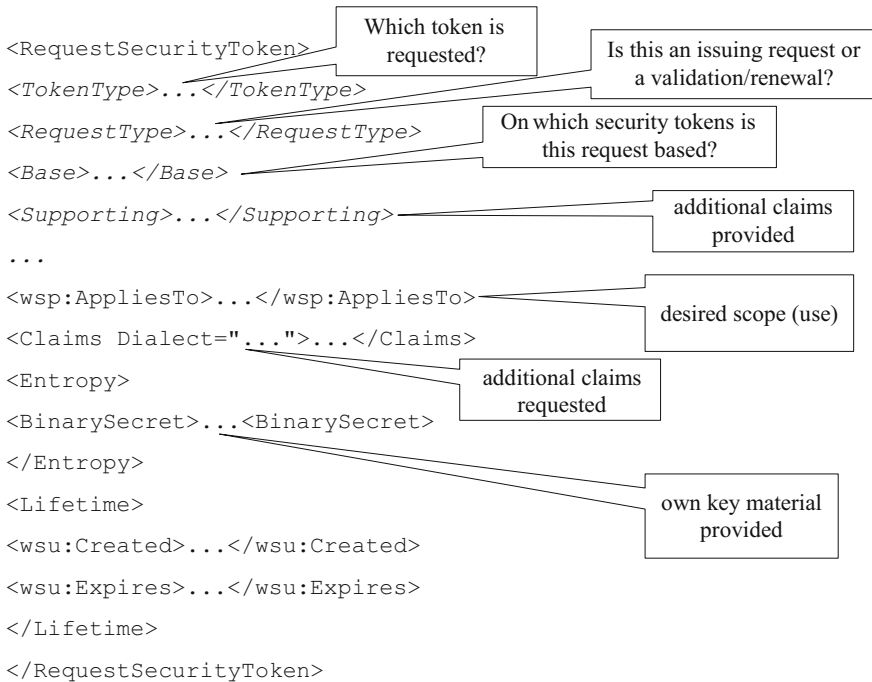


Abb. 9.7 Security Token Request

Dazu definiert WS-Trust im Wesentlichen zwei XML-Datenstrukturen, die einen Request und eine Response beinhalten. Typischerweise wünscht sich der Antragsteller ein gewisses Token und bietet dem Aussteller dafür gewisse Sicherheiten an. Zu diesen können eigene Zertifikate oder andere Authentisierungstoken

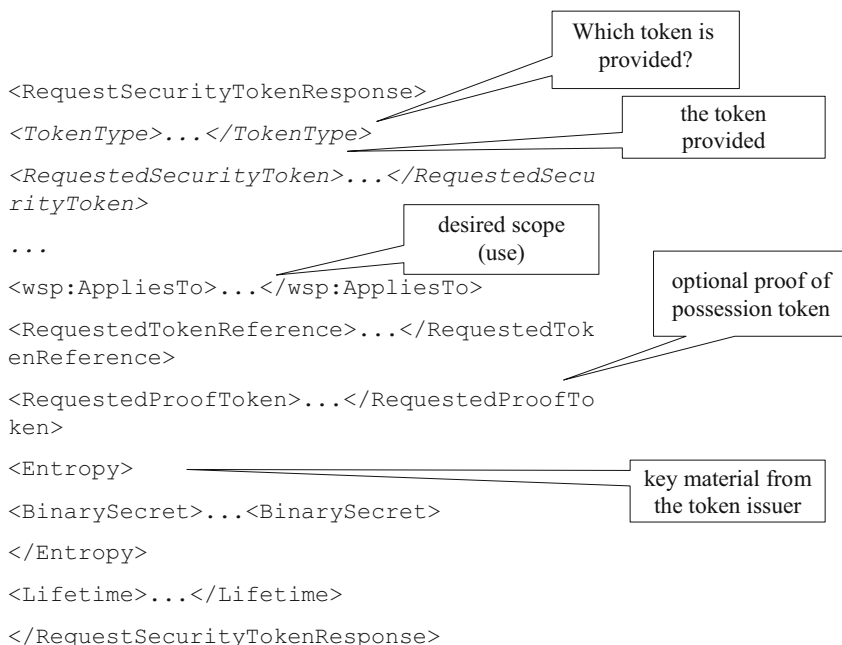


Abb. 9.8 Security Token Response

gehören. Aber auch Zufallszahlen zur Erhöhung der Sicherheit bei der Schlüsselgenerierung und Übertragung werden vom Requester geliefert. Abbildung 9.7 zeigt das Security-Token-Request-Element im Detail.

Man beachte, dass Schlüsselmaterial, welches eventuell im Rahmen des `<BinarySecret>`-Tags transportiert wird, durch einen vertraulichen Kanal abgesichert sein muss. Dieser kann entweder durch ein Protokoll zur Transportsicherung wie SSL geliefert werden, oder dadurch, dass die ganze SOAP-Nachricht, die das Security-Token-Request-Element enthält, verschlüsselt wird.

In der Antwort (Abb. 9.8) – die mit den gelieferten Schlüsseln und Credentials gesichert werden kann – liefert der Security Token Issuer das verlangte Token sowie optional auch sogenannte *proof-of-possession*-Tokens, die für eine interaktive Bestätigung des Besitzes des in einem Token enthaltenen Schlüssels gegenüber einem Service Provider benutzt werden können.

Beide, Request- und Response-Elemente, sind Teil des SOAP-Message-Body. Sie können aber auch Security-Elemente im Header referenzieren (z. B. Schlüsselinformationen zur Entschlüsselung von Schlüsselmaterial).

9.3.3 WS-Trust-Szenarien

WS-Trust ist sicherlich eine Kernspezifikation der Web-Services in Bezug auf Security. In diesem Abschnitt behandeln wir einige weitere Szenarien, die damit

ermöglicht werden. Besonders interessant ist vielleicht die Möglichkeit der Ausstellung von Sitzungsschlüsseln (Session Keys) für eine sichere kanalbasierte Kommunikation mit den Service Providern. Es hat sich nämlich schnell herausgestellt, dass die Abwicklung jeglicher Kommunikation über signierte und verschlüsselte Objekte bzw. Nachrichten recht aufwändig wird, vor allem, wenn die Kommunikation den Charakter einer länger dauernden Sitzung besitzt. Somit wird auf Basis der objektbasierten Sicherheit eine kanalbasierte Sicherheit ermöglicht.

Szenario 1: Ein Requester fordert ein Token für einen Service an, welches einen mit dem Public Key einer dritten Partei und dem Public Key des Service verschlüsselten Sitzungsschlüssel enthält. Das Token wird zunächst an die dritte Partei und von dieser an den Service Provider weiter gereicht. Mithilfe des in dem Token enthaltenen Sitzungsschlüssels kann die dritte Partei im Auftrag des Requesters einen sicheren Kanal zum Service aufbauen und diesen nutzen.

Szenario 2: Ein Requester lässt sich ein vom STI ein Token erzeugen, welches gewisse Attribute an den Public Key des Requesters knüpft. Beim Vorweisen dieses Token beim gewünschten Service muss der Requester noch nachweisen, dass er tatsächlich über den zum Public Key gehörigen Private Key verfügt (Proof-of-Possession).

Szenario 3: Ein Web-Service-Provider kann vom Security Token Issuer verlangen, dass er für ihn ein bestimmtes Token validiert.

Szenario 4: Um zu garantieren, dass ein Token tatsächlich aktuell ist und nicht von einem Angreifer per Replay-Attack wieder eingespielt wird, kann ein Token-Empfänger ein zusätzliches Challenge/Response-Protokoll initiieren.

Üblicherweise können die Token von den Requestern als „Black Box“ behandelt werden, ohne sie interpretieren zu müssen. Die Token liegen entweder im Klartext oder in verschlüsselter Form vor.

Die Kopplung verschiedener Security Token Issuer, die auf jeweils unterschiedlichen Techniken beruhen, erlaubt über diese Szenarien hinaus eine Cross-Domain-Interoperabilität durch Umwandlung von Token in andere Token. Dies kann im Vordergrund vor einem Request an einen Service Provider geschehen (für den Requester sichtbar) oder im Hintergrund transparent vom Service Provider angefordert werden.

Wer nun meint, damit das Ende der Web-Services-Standards in Bezug auf Security erreicht zu haben, irrt leider gewaltig. Im obigen Abschnitt haben wir bereits drei Probleme der Web-Services-Security erwähnt, die bisher noch nicht besprochen wurden:

- Es ist unklar, woher ein Requester wissen soll, welche Token und Bestätigungen er für die Kommunikation mit einem Service Provider (WS-Policies) beantragen soll.
- Es ist unklar, wie ein sicherer Kanal zu einem Service Provider mithilfe des Session Token aufgebaut werden soll (WS-SecureConversation).
- Es ist unklar, wie Security Token Issuer sich untereinander und den Service Providern Token anerkennen bzw. umwandeln sollen (WS-Federation).

Mit diesen Standards oder Vorschlägen für Standards (den genauen Status kann man den Seiten von OASIS entnehmen [OASIS-OPEN], die meisten sollten bis Mitte 2007 von Oasis verabschiedet werden), betreten wir sehr unsicheres Gelände bezüglich Interoperabilität. Überhaupt lässt selbst die Kompatibilität verschiedener Implementationen von Web-Services-Security-Standards bereits auf den niedrigeren Ebenen noch sehr zu wünschen übrig. Große Firmen mit vielen Außenkontakten berichten leider immer wieder, dass Anbindungen über Web-Services speziell an den jeweiligen Partner angepasst werden müssen.

9.3.4 *Autoritäten und Aussagen*

Kommen wir nun zum größten bisher noch fehlenden Teil der Businessmodelle für Web-Services: Aussagen über sicherheitsrelevante Dinge. Wir haben eingangs geschildert, wie vielfältig im Alltag mit unterschriebenen Bestätigungen, Vollmachten, Gutscheinen, Kreditkarten etc. umgegangen wird. Entscheidend ist dabei immer ein Element: Alle Beteiligten müssen die gemachten Aussagen in ihrer Bedeutung für die Sicherheit und den Geschäftsvorgang verstehen. Was im Alltag kein großes Problem darstellt, erweist sich in der Kommunikation von Maschinen über Web-Services als ein schwerwiegendes Problem: Flexible, mächtige Kollaborationsformen zwischen mehreren Beteiligten hängen entscheidend davon ab, dass die Semantik der Zusammenarbeit klar ausgedrückt werden kann. Die Voraussetzungen, der Prozess, die Prüfung des Geschäftsvorganges müssen deshalb so beschrieben werden, dass sie maschinenlesbar sind und vor allem auch von Maschinen verstanden werden können. Dieses Problemfeld erinnert sehr an Versprechungen, wie sie vor einiger Zeit bezüglich UDDI (Universal Description, Discovery and Integration) gemacht wurden: Intelligente Services sollten sich anhand von Servicebeschreibungen, die in einem universellen Repository hinterlegt sind (eben UDDI) ihre Partnerservices selbst aussuchen. Das Ergebnis war ein völliges Desaster und auch acht Jahre später existieren solche intelligenten Services noch nicht. Allerdings gibt es Ansätze, in eng begrenzten Businessprozessen zu automatisierten Businessprozessen wie z. B. Bestellungen zu kommen. Die Lösungen basieren stark auf Beschreibungen der Semantik durch OWL und andere Standards.

Im Bereich der Security ist jedoch eine semantisch mächtige und dennoch klar definierte Sprache vonnöten, sonst lassen sich die Bedingungen der Kollaboration einfach nicht ausdrücken. OASIS hat hier mit der *Security Assertion Markup Language* (SAML [SAML]) eine XML-basierte Beschreibungssprache für Eigenschaften und Policies vorgestellt. Sie arbeitet eng mit einer weiteren Spezifikation zur Zugriffskontrolle (extensible *Access Control Markup Language*, XACML) zusammen.

Ein signiertes Token wie z. B. ein Session-Token eines Kerberos-Systems hat im Rahmen einer Sicherheitsinfrastruktur oft eine rein technische Bedeutung. Diese Token lassen sich meist mit wenigen Attributen sicher beschreiben. Mit

SAMLWS-Security

Assertion

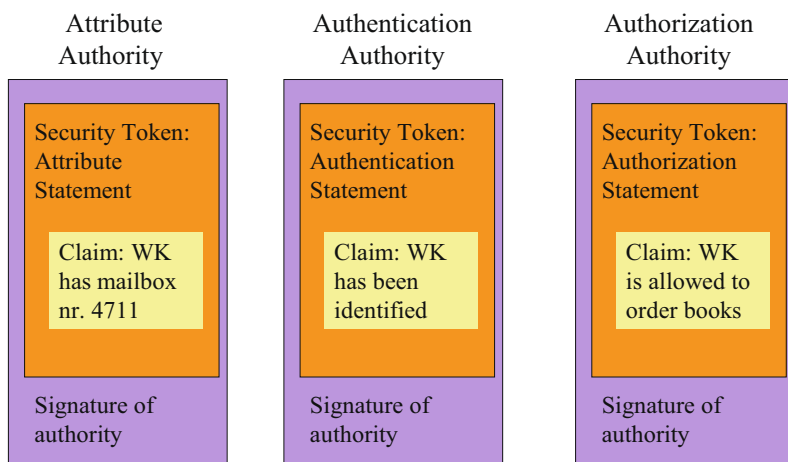
Claim

Credential

Endorsed (signed) security token

Authority

Security Token Issuer

Abb. 9.9 Unterschiedliche Terminologie zu Security-Aussagen**Abb. 9.10** Verschiedene Formen von Autoritäten und Zertifikaten

dem Begriff der *Aussage* aber bewegen wir uns in den Bereich der Geschäftsvorgänge und der Kollaboration hinein. Das wird schon am Begriff selbst deutlich, der leider von verschiedenen Gremien ganz unterschiedlich geprägt wurde: SAML bezeichnet ihn mit „Assertion“, WS-Security mit „Claim“. Auch andere zentrale Begriffe werden von den beiden Gremien unterschiedlich bezeichnet (Abb. 9.9).

Wichtiger als der sprachliche Wirrwarr ist jedoch die Rolle der Autoritäten in diesem Kontext. SAML unterscheidet je nach Aufgabe drei Arten von Autoritäten (bzw. Security Token Issuer), die in WS-Security unter dem Begriff des Security Token Issuer zusammengefasst werden (Abb. 9.10).

Hier einige typische SAML-Assertions:

Assertion Element: Enthält Meta-Daten über die Assertion selbst (Aussteller, Signatur, etc.).

Conditions: Valid until, valid not before, etc.

Advice: Zusätzliche Informationen vom Aussteller der Assertion.

Secure Association Markup Language (SAML)

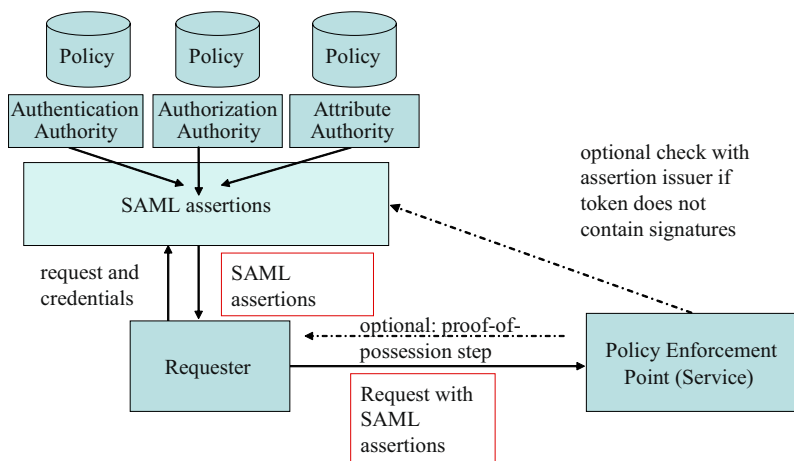


Abb. 9.11 SAML-Architektur [SAML]

Subject Statement: Informationen über den Inhaber der Assertion (Name, Identifikationsmöglichkeiten, etc.).

Authentication Statement: Bescheinigung darüber, dass der Aussteller das Subject zu einem bestimmten Zeitpunkt authentisiert hat. Kann auch Informationen über die Authentifikationsmethode, den Ort etc. beinhalten.

Authorization Decision Statement: Ein Request des Subjects ist bewilligt oder abgelehnt worden. Das Statement enthält Informationen über die Art des Requests, die angeforderten Ressourcen und die Credentials, die das Subject vorgelegt hat.

Attribute Statement: Der Herausgeber bestätigt, dass das Subject die in dem Token genannten Attribute besitzt.

SAML-Assertions können in den Header von SOAP-Nachrichten eingebettet werden (als Teil eines WS-Security-Headers). Das Zugriffsmodell XACML kann SAML-Assertions auswerten und auf dieser Basis Zugriffsentscheidungen treffen. Der grundlegende Mechanismus hierfür sieht (in Verbindung mit einer SAML-Runtime) wie in Abb. 9.11 abgebildet aus.

Mithilfe von SAML können alle Policies und Mechanismen, die sich auf Authentifikation, Authorisierung und Attributzuweisung beziehen, an den Security Token Issuer bzw. die drei Typen von Authority, die SAML kennt, ausgelagert werden. Der Service Provider muss nur die SAML-Aussagen überprüfen, bevor er dem Requester den Zugriff gewährt. Durch SAML werden alle diese sicherheitsrelevanten Aussagen zwischen verschiedenen Services interoperabel, da das Format der Aussagen durch den Standard festgelegt ist.

9.3.5 Bedrohungen und Risiken

Welche typischen Bedrohungen und Risiken ergeben sich nun aus WS-Security, wenn man davon ausgeht, dass die Algorithmen für Verschlüsselung und Integritätsschutz richtig implementiert und verwendet wurden und auch die Zufallszahlenerzeugung für Schlüsselmaterial und Challenges den üblichen Anforderungen genügt? Abbildung 9.12 fasst die möglichen Bedrohungen in einer Übersicht zusammen.

Wir haben schon mehrfach die Gefahr von Replay-Attacken bei digitalen Signaturen oder allgemeiner bei tokenbasierten Authentifikationsmechanismen erwähnt. Dieser Gefahr begegnet man am Besten, indem die sich digitale Signatur auch auf Elemente erstreckt, die es erlauben, ein Replay durch einen Angreifer zu entdecken. Dies können Zeitstempel oder Sequence Numbers sein. [WS-SEC] empfiehlt, Zeitstempel, die älter als fünf Minuten sind, in interaktiven Szenarien zurückzuweisen.

Ebenfalls bereits erwähnt haben wir die Gefahr einer „Secure Channel Confusion“: WS-Security lässt die Möglichkeit zu, sensitives Schlüsselmaterial im Klartext zu übertragen. Dies ist natürlich nur dann sicher, wenn hierfür ein sicherer Transportkanal zur Verfügung steht – und ist dies wirklich für alle Intermediaries der Fall? Bevor man sich hier auf vage Aussagen und Annahmen verlässt, sollte man lieber die Vorteile der objektbasierten Sicherheit auch im Hinblick auf den sicheren Transport von Credentials nutzen.

Auch die Kombination von XML-Encryption und XML-Signatur in SOAP-Headern kann Gefahren in sich bergen: Bildet man beispielsweise eine digitale

Attacks against WS-Security

XML Level	signatures over plain text, encrypted plain text. Guessing attacks
Message level	Signature misses security tokens used. Token replacement attacks
Key Exchange level (ws-trust)	Same keys used for authentication and message encryption allows key attacks.
Secure Message level	Replay attacks. Message (or message parts) copied and sent again by attacker
Secure Channel confusion	Message uses unencrypted binary key material (password etc.) under the assumption of a secure transport/channel. Channel not secure.
Sender/author confusion	Application or trust engine forgets to check liveness of token. Author is not sender.
Certificate level	Trust engine does not verify validity of certificates

Abb. 9.12 Attacken auf WS-Security

Signatur über einen Klartext und verschlüsselt den Klartext danach, so sollte die Signatur nicht unverschlüsselt bleiben – anderenfalls verfügt ein Angreifer durch Anwendung des Public Key zu der Signatur über den Hash-Wert des Klartexts. Sind dem Angreifer darüber hinaus gewisse Teile des Klartexts bekannt, etwa weil es sich um standardisierte Header handelt, sind damit Dictionary Attacks auf den Klartext möglich.

Als letztes möchten wir auf die Gefahr eines „Token Exchange“ hinweisen: Angenommen, jemand besitzt verschiedene Zertifikate für denselben Public Key, die diesen Public Key aber mit unterschiedlichen Attributen ausstatten. Wird nun eine digitale Signatur mithilfe einer Referenz auf ein Token validiert, das wiederum eines dieser Zertifikate enthält, wäre es möglich, dieses Token gegen ein anderes mit einem anderen Zertifikat (für den gleichen Public Key) auszutauschen. Der einfachste Weg, einen solchen Angriff abzuwehren, ist natürlich, das Token bzw. einen eindeutige TokenID in die Signatur mit einzuschließen. Generell gilt beim Einsatz digitaler Signaturen, dass möglichst alle sicherheitsrelevanten Informationen (wie IDs oder Zeitstempel) mit signiert werden sollten, um einen Austausch dieser Informationen durch einen Angreifer zu verhindern.

Hintergrundannahmen hierzu sind, dass traditionelle Attacken gegen Vertraulichkeit und Integrität durch Signaturen und Verschlüsselung verhindert werden. Man-In-The-Middle-Attacken werden durch gegenseitige Authentisierung verhindert. Außerdem gehen wir von einer korrekten Erzeugung von Zufallszahlen aus. Applikationen und Benutzern ist das Identitätsproblem der Namen in Zertifikaten bewusst (mehrere CAs, Namensduplikate). Weitere Bedrohungsmodelle finden sich in den einschlägigen WS-Security-Standards.

9.3.6 Einbindung in Applikationen

In [Ant] findet sich ein Beispiel für die Vorgehensweise beim Einsatz von Authentisierung, Signierung und Verschlüsselung mit dem Web Services Security for Java Framework (WSS4J). Das Activity-Diagramm in Abb. 9.13 beschreibt die nötigen Schritte.

9.3.7 Web-Services über REST

REST ist die Abkürzung für Representational State Transfer Architecture und beschreibt die Architektur des WWW, wie sie durch Roy Fielding beschrieben wurde [Fielding]. Kennzeichnend für REST ist, dass jedes Dokument bzw. jeder Prozess, der durch einen Service realisiert wird, als URI repräsentiert wird, d. h. als Web-Ressource (Abb. 9.14).

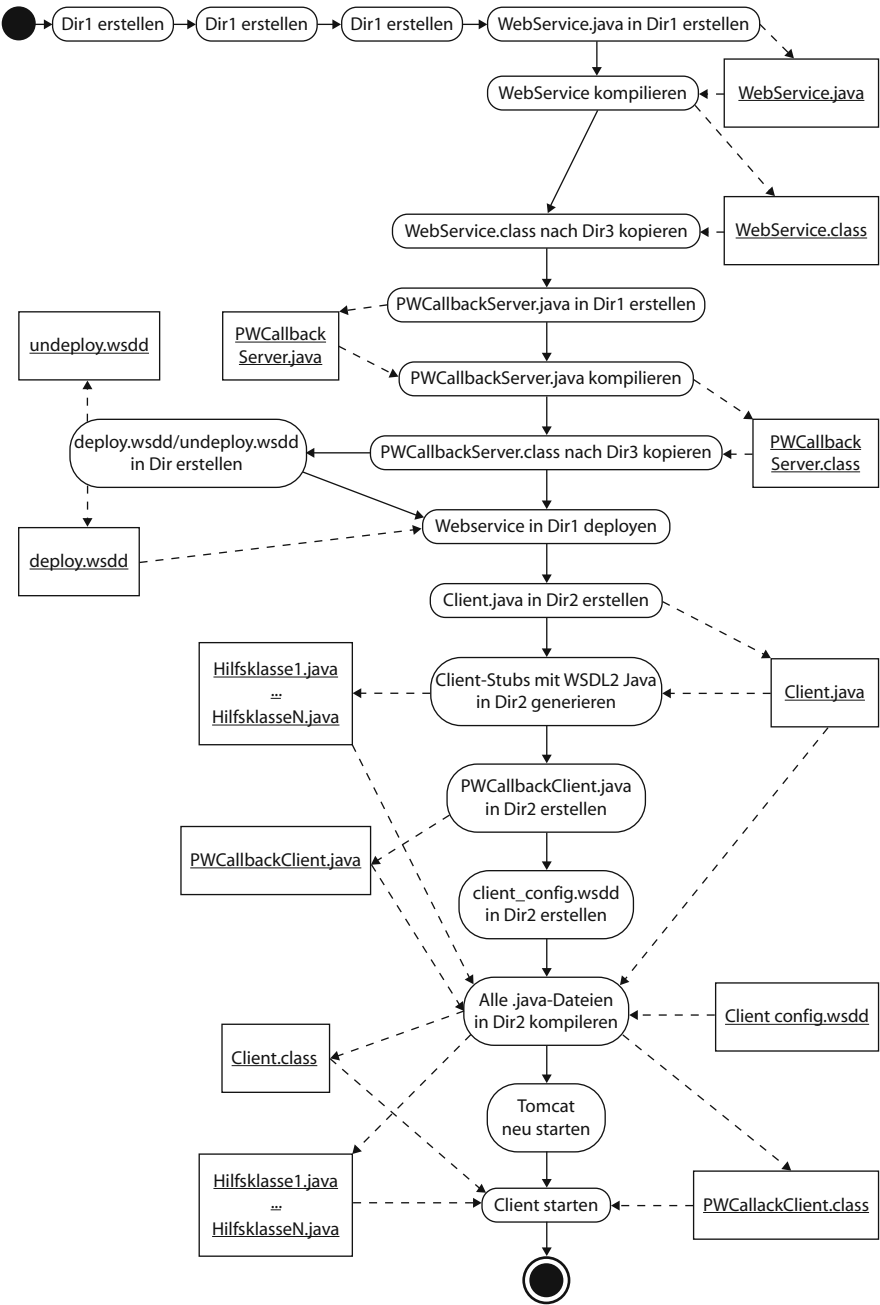


Abb. 9.13 Activity-Diagramm für den Einsatz von WSS4J

Representational State Transfer Architecture (REST)

- All Resources (processes and documents) have an URI
- All Ressourcen are accessed through HTTP primitive operations (get, post, put, delete)
- GET is supposed to be idempotent

Sample Request Url (Yahoo Traffic Web Services) GET Method:

<http://api.local.yahoo.com/MapsService/V1/trafficData?appid=YahooDemo&street=701+First+Street&city=Sunnyvale&state=CA>

Abb. 9.14 REST-Prinzipien und Beispiel

Eine REST-Applikation muss ein Modell ihrer Ressourcen und deren Hierarchie erstellen. Anschließend werden die Ressourcen und ihre Eigenschaften als URL's sichtbar gemacht. Es unterscheidet sich dann ein deutsches von einem englischen Dokument an einer Sprachmarkierung in der URL. Nichts soll dabei impliziert bleiben (z. B. versteckt im Session State).

Diese Ressourcen bzw. deren State/Repräsentation werden durch ganz wenige Methoden manipuliert: GET, PUT, POST, DELETE. Dies steht in starkem Kontrast zur ursprünglichen SOAP-Philosophie, die klar aus dem Bereich der Remote Procedure Calls stammt. Bei REST liegt der Schwerpunkt auf den Ressourcen – sprich Substantiven – während SOAP eher die Verben der Remote Procedure Calls verwendet.

Viele Web-Services werden sowohl mit einem SOAP als auch mit einem REST API angeboten. Abbildung 9.15 zeigt das Interface zu einem Publish/Subscribe-Service, bei dem man sich für den Empfang von Nachrichten registrieren kann. Eine Einführung in das Thema Web-Services über REST findet sich bei [Bayer].

REST-Services gelten als sehr leichtgewichtig und einfach zu verwenden. Authentisierung bzw. Authorisierung findet durch einfache, vom Service generierte Token statt, die den Request begleiten. Abbildung 9.16 zeigt die Verwendung von Produkt- oder Authentisierungstoken bei dem Newsgator-Service.

Amazon.com verwendet für seine kommerziell genutzten Services zwei verschiedene Token [ONeill06].

- Access Key ID: ein nicht-öffentliches Token, das lediglich den User bezeichnet.
- Secret Access Key: von Clients verwendet, um einen HMAC über einen Request zu erzeugen und den Request damit zu authentifizieren.

Mark O'Neill hat anlässlich der RSA-Konferenz 2006 auf einige Sicherheitsproblematiken mit REST-Services hingewiesen. Darüber hinaus ist sein Vortrag eine ausgezeichnete Einführung in die REST-Philosophie.

Subscribe to pubsub.com:

Post to: <http://www.pubsub.com/REST/subscribe.php>

```
<pubsub-rpc version="1.0"> <subscribe> <email-address>foo@example.com
</email-address> <delivery-method>REST</delivery-method> <delivery-
address> http://www.yourserver.com/your/script </delivery-address>
<title>Bar</title> <query-string>Bar</query-string> </subscribe>
</pubsub-rpc>
```

Result:

```
<xml> <validation-result>true</validation-result> <subscription-
token>12345abcdef</subscription-token> <note> Your subscription was
created and confirmed. </note> </xml>
```

Cancel subscription:

Post to: <http://www.pubsub.com/REST/cancel-subscription.php>

```
<subscription-token>12345abcdef</subscription-token>
```

Abb. 9.15 REST-Beispiel für einen Pub/Sub-Service

<http://www.newsgator.com/ngs/api/NewsGatorRESTAPI.pdf>

API Product Key

Developers wishing to use the NewsGator API should first obtain an API Product Key from the NewsGator API Signup Page. This product key should accompany each HTTP request as a X-NGAPIToken HTTP header.

Authentication

Unless otherwise noted, all HTTP requests are required to be authenticated using HTTP Digest (preferred) or Basic using NewsGator Online credentials. Credentials submitted using Basic should be submitted using the HTTPS protocol.

Sample Request

```
GET /ngws/svc/Location.aspx HTTP/1.1 Authorization: Basic
dXNlcm5hbWU6cGFzc3dvcmQ= User-Agent: curl/7.12.2 (i386-pc-
win32) libcurl/7.12.2 OpenSSL/0.9.7e zlib/1.2.1 Host: 192.168.10.204
Pragma: no-cache Accept: */* X-
NGAPIToken:A8A8A9B5DC8240a4B1065C8CE4771074
```

Abb. 9.16 Newsgator-REST-Service

- Einfache Token für Produkte und User IDs verwendet.
- Shared Keys nur selten für die Authentisierung von Nachrichten verwendet.
- Parameter werden oft über GET Methoden übertragen. Oft sind Credentials enthalten, die in Server Logs sichtbar werden.
- Die Semantik der REST APIs ist nicht immer richtig: So gibt es GET-Methoden, die serverseitig den State ändern, d. h. nicht nur „Read-only“ sind.
- Viele Web-Services bieten sowohl ein REST als auch ein SOAP API an, vergessen jedoch, das REST API im Firewall abzusichern.
- Firewalls sollten REST-Methoden genauso wie SOAP-Methoden filtern können.
- Und zuletzt: wenn ein Web-Service beide Interfaces anbietet, wird meist das REST-Interface sehr viel mehr verwendet als das SOAP API.

9.4 SOA-Security

Das Konzept der serviceorientierten Architektur (SOA) könnte letztlich die Rettung für Web-Services darstellen. Letztere waren drauf und dran, als kompletter Re-Write von CORBA-Services in Form von Interfaces, beschrieben in XML, in die Geschichte einzugehen. Darüber hinausgehende Konzepte wie automatisches Matchen von Anbietern und Nachfragern via UDDI waren zunächst sensationelle Fehlschläge. SOA brachte zunächst einen klareren Businessbezug für die IT im Sinne von Outsourcing von Services, und brachte mehr Klarheit über die Abstraktionsstufen der Service-Interfaces sowie deren Komposition in Prozesse. Bekannt geworden ist das Diagramm der vernetzten Provider von Services, die ihrerseits wieder Services auslagern. Abbildung 9.17 zeigt die Vernetzung der Services über verschiedene Firmen hinweg.

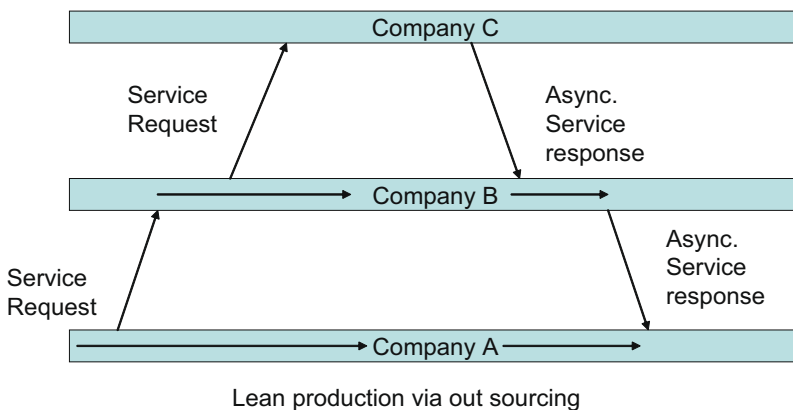


Abb. 9.17 Servicedelegation in SOA

Wie der Name sagt, ist das Konzept SOA auf den Aufruf von Services bezogen, die von anderen Providern bereitgestellt werden. So weit ist ein Service eine logische Entität ohne reale Einschränkungen und aus der Sicht der IT gesehen. Harte Bindungen zu Providern werden durch den dynamischen Bindemechanismus via WSDL vermieden.

Hinter dem Aufruf eines Services eines Providers steht jedoch im Businesssinne sehr viel mehr: Welche Garantien für Verfügbarkeit und Zuverlässigkeit werden gegeben? Schließlich hängen zentrale eigene Prozesse davon ab.

Welche Performance kann man erwarten? Kann der global tätige Konzern seine Services beliebig weltweit verteilen? Solange Services logisch gesehen werden, ist das gut möglich.

Wie sieht die Sicherheit von Serviceaufrufen aus? Was kann der Provider erwarten? Was kann der Aufrufer erwarten? Wir haben diese Problematik im Bereich Federation untersucht, dabei aber eines ganz außer Acht gelassen: Die Daten, die dabei ausgetauscht werden.

Was passiert mit Daten, die im Rahmen eines Serviceaufrufs fließen? SOA bedeutet streng genommen eine starke Kapselung der Daten. Ideal sind Stateless Interfaces, die Befehle durchführen lassen. Aber es wird Services geben, die umfangreiche Firmendaten benötigen – wie wird deren Sicherheit garantiert?

Schnell stellt sich bei diesen Überlegungen heraus, dass SOA sehr viel mehr Fragen aufwirft, als nur die Zugriffskontrolle auf die Services bzw. die Sicherheit der Aufrufe während des Transports. Welche Daten dürfen überhaupt das Unternehmen verlassen – z. B. in Richtung Indien? Welche Daten stehen dann für Tests externer Services zur Verfügung? Wie komplex die dahinter liegenden Probleme sind, werden im Report zur Globalisierung der Softwareentwicklung [ACM] im Zusammenhang mit Outsourcing und Offshoring nach Indien und andere Länder erörtert.

Dass SOA auch für die IT Probleme mit sich bringt, lässt sich erneut am Problem der Daten zeigen: Weltweite Services benötigen aus Gründen der Performance Daten aus Caches bzw. replizierte Daten. Wie sieht die Sicherheit von gecachten Daten in SOA aus? Caching reißt Daten generell aus dem originalen Sicherheitskontext. Wie wird der Zugriff auf Caches abgesichert? Wie fließt Authentisierungsinformation über messageorientierte Middleware-Schnittstellen (Verwendung von Activity Envelope zum Einbetten von Credentials in Messages, Verwendung von LTPA-Token, die von Message-Adaptern in Runtime Identities umgesetzt werden vor Aufruf von Enterprise Beans)? Siehe dazu auch eine aktuelle Untersuchung bei DaimlerChrysler [Reith].

Für die Ausführung eines Services bei einem externen Provider gilt im übrigen die gleiche Problematik wie für GRID-Computing: Wie kann der Auftraggeber sicher sein, dass beim Provider richtig gerechnet wird und nur die autorisierten Instanzen mit den Daten des Aufrufers in Kontakt kommen?

Ein Design einer SOA-Architektur sollte daher die Probleme der Vertraulichkeit von Daten im ersten Schritt klären.

Literatur

- [ACM] Globalization Report der ACM, www.acm.org/globalizationreport.pdf
- [Ant] E. Anton, Web Services in realen Business Applikationen, Diplomarbeit Hochschule der Medien, 2006
- [Bayer] T. Bayer, REST Web Services – eine Einführung, 2002, <http://www.oio.de/public/xml/rest-webservices.pdf>
- [Fielding] R.T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, Dissertation University of California, Irvine, 2000, http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation_2up.pdf
- [KS] W. Kriha, R. Schmitz, Internet-Security aus Software-Sicht, Springer 2008
- [KSRW] T. Kohno, A. Stubblefield, A. Rubin, D. Wallach, Analysis of an Electronic Voting System, <http://avirubin.com/vote.pdf>
- [Reith] L. Reith, Concept of data distribution for worldwide distributed services in service -oriented architectures, HDM/DaimlerChrysler 2007
- [OASIS-OPEN] Organization for the Advancement of Structured Information Standards, Specification status page, <http://www.oasis-open.org/specs/>
- [ONeill03] M. O'Neill, Web Services Security, 2003
- [ONeill06] M. O'Neill, Applying Security for REST, http://www.vordel.com/downloads/rsa_conf_2006.pdf
- [SAML] Security Assertion Markup Language, <http://www.oasis-open.org/specs/index.php#samlv2.0>
- [Wee] S. Weerawarana et. al, Web Services Platform Architecture, Prentice Hall 2005
- [WSDL] Web Services Description Language (WSDL) 1.1, <http://www.w3.org/TR/wsdl>
- [WS-SEC] Web Service Security: SOAP Message Security, <http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf>
- [XML-Can] J. Boyer, Canonical XML, <http://www.w3.org/TR/xml-c14n>
- [XML-Enc] D. Eastlake, J. Reagle: XML Encryption Syntax and Processing, <http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/>
- [XML-KMS] S. Farrell, S. Mysore et al., XML Key Management Specification, <http://www.w3.org/TR/xkms2/>
- [XML-Sig] D. Eastlake, J. Reagle, D. Solo: XML-Signature Syntax and Processing, <http://www.ietf.org/rfc/rfc3275.txt>

Kapitel 10

Sichere Software: Mechanismen und Konstruktionsprinzipien

In diesem Kapitel wollen wir die Grundbedingungen zur Produktion sicherer Software beleuchten. Zum Teil haben wir das an Beispielen bereits in den vorherigen Kapiteln getan – speziell die Anforderungen an Software, die sich aus besonderen Bedrohungsmodellen ergeben. Dazu gehören die Plattformicherheit, die Sicherheit in vernetzten Firmenstrukturen oder föderativen Strukturen, sowie die Sicherheit in lose gekoppelten Systemen bis hin zu Web-Infrastrukturen. Im Vordergrund stand dabei die korrekte Identifizierung und Autorisierung von Benutzern sowie das Verhindern von illegalen Erweiterungen von Privilegien. Die Software selbst (Web Application Server, Web Application) wird dabei meist als korrekt angesehen.

Ein entscheidendes Kriterium für sichere Software ist in diesem Kontext die Kenntnis der spezifischen Bedrohungen, wie sie z. B. für Web-Applikationen ausgezeichnet auf www.owasp.org zusammengefasst und erklärt werden. Dazu gehört natürlich auch die Kenntnis der Angriffsmöglichkeiten auf Plattformen, Applikationen oder die Vorstellungen bei Nutzern, also die jeweiligen Bedrohungsmodelle. Software muss sich trotz Bedrohungen korrekt verhalten und große Teile des Sicherheitscodes in Enterprise-Architekturen beschäftigen sich genau mit diesem Abwehrgedanken. Authentisierungsmethoden, Delegationsmodelle und rollenbasierte Zugriffskontrolle sind Beispiele dafür.

Aber die Kenntnis von Bedrohungen und Abwehrmaßnahmen ist nur eine Teilbedingung für sichere Software. Es ist die Art und Weise, wie Software heute konstruiert wird, die über Zuverlässigkeit und Sicherheit – auch aber nicht nur im Hinblick auf mögliche Angriffe – wesentlich mitentscheidet. Dies geschieht auf den Ebenen der Mikro-Architektur, durch Sicherheitsbausteine und durch die jeweilige Makro-Architektur einer Lösung, also längst bevor das Konzept von Angriffen überhaupt ins Spiel kommt. Unter der Annahme, dass sichere Lösungen auf einer unsicheren Basis nicht möglich sind, werden wir einen Blick auf die Konstruktion von Software insgesamt werfen und den Zusammenhang von Zuverlässigkeit, Sicherheit und Architektur tiefer untersuchen. Dabei wird die Frage nach den Ana-

lysemöglichkeiten von Software in Bezug auf die Sicherheit eine wesentliche Rolle spielen: Lassen unsere Architekturen eine Analyse überhaupt zu?

Mit diesem Kapitel bewegen wir uns weg von den klassischen Fragen zur Sicherheit von Systemen und hin zu Kernfragen der Softwareentwicklung und ihrer Relevanz für Sicherheit und Zuverlässigkeit. Der Leitgedanke ist dabei, dass es nicht nur der „Security“-Anteil eines Systems oder einer Software ist, der Probleme bereitet, sondern dass es Mängel im „Safety“-Anteil sind, die Angriffe häufig so erfolgreich machen. Getreu der „outrage“-Theorie für die Einschätzung von Risiken (siehe Kap. 2) kann man vermuten, dass der Safety-Aspekt wesentlich weniger öffentlichkeitswirksam als der Security-Aspekt ist: Angreifer bzw. drastische Fehler in der Zugangskontrolle kommen in die Medien. Die Diskussion über den Einfluss von Seiteneffekten im Kontext von Concurrency auf die Stabilität einer Software eher weniger. Und doch werden hier Fehler eingebaut, die sich anschließend auch als Security-Probleme darstellen können – ihre Ursache liegt jedoch wesentlich tiefer in der Konstruktion der Software selbst, z. B. in einem fundamentalen Mangel an Schadensbegrenzung oder an einer falsch verstandenen Flexibilität.

So werden wir beispielsweise untersuchen, wie sich Autorität durch Software eingrenzen lässt. Wir tun dies an zwei Beispielen: Zum einen an einer kurzen Analyse der Sicherheitsmängel, die in verschiedenen Versionen des Mozilla-Browsers aufgetreten sind und zum anderen durch den Entwurf eines sicheren Browsers, den sog. DARPA-Browser. Dabei wird sich zeigen, dass in Systemen, die mit reduzierter Autorität arbeiten, der Benutzer eine wichtige Rolle als Quelle von Autorität spielen muss, und dass frei verfügbare Autorität (sogenannte *Ambient Authority*) eine ständige Quelle für Sicherheitsprobleme darstellt.

Der Benutzer als entscheidende Instanz bei Fragen der Sicherheit und als Quelle von Autorität – das kann nicht sein, werden alle sagen, die schon Erfahrungen mit den unseligen Sicherheitsdialogen einer Personal Firewall oder der meisten Web-Browser erlebt haben. Offenbar hat sichere Software auch sehr viel mit guter Usability zu tun. Diesem Problem werden wir im Kap. 13 nachgehen. Darin wollen wir zeigen, dass sichere und bequeme Nutzer-Interfaces nur auf Basis sicherer, autoritätsreduzierter Architekturen möglich sind. Indem die Rechte des Users nicht mehr automatisch komplett auf Softwareagenten übertragen werden, kommt der gezielten Ermächtigung durch den User eine sehr große Bedeutung zu. Die Kommunikation zwischen System und User muss daher semantisch eindeutig und unfälschbar ablaufen – und zugleich noch effektiv sein. In diesem Dilemma kann aber die Erkenntnis helfen, dass die Interaktion zwischen User und System in autoritätsreduzierten Systemen grundsätzlich ganz anders ablaufen kann als bisher gewohnt. Usability wird damit zur zentralen Herausforderung und gleichzeitig zur echten Neuerung in solchen Systemen. Da die Entwicklung sicherer Systeme in dem genannten Sinn einige Brüche mit bisherigen Verfahren erfordert wollen wir zunächst jedoch in einem kleinen Exkurs die wirtschaftlichen und historischen Rahmenbedingungen der Entwicklung sicherer Software heutzutage diskutieren.

10.1 Rahmenbedingungen sicherer Software

10.1.1 *Die Macht des Faktischen*

„Um fast 15 Prozent auf rund 7,4 Milliarden US-Dollar stiegen die Umsätze mit Sicherheitssoftware Gartner zufolge im vergangenen Jahr. Maßgeblich angetrieben wurden die Verkäufe durch die große Nachfrage nach Anti-Viren-Programmen, die inzwischen fast 55 Prozent aller Umsätze mit Sicherheitssoftware ausmachen.“

Zu Beginn unserer Diskussion ein Zitat aus dem heise-Newsticker vom September 2006 [heise9_06].

Die Forderungen nach dem Einsatz von Firewalls, Virenschannern etc. sind aus den Listen der Empfehlungen von IT-Security-Fachleuten nicht wegzudenken. Es scheint, das Geschäft mit IT-Sicherheit ist fest in der Hand des Zubehörmektes. So verwundert es nicht, dass alle technischen Maßnahmen, die die Sicherheitsarchitektur betreffen, schnell auf Kritik dieser Industrie stoßen werden. So wurde die Firma Symantec mit der Kritik an neuen Sicherheitsmekanismen in Windows Vista bei der EU vorstellig, die angeblich die Konkurrenz benachteiligen. Auch dazu ein Zitat aus der Presse [CW]:

Kernelschutz sperrt Security-Software aus

Der Kernelschutz Patchguard wird kritisiert, weil er die Möglichkeiten von Sicherheits-Software einschränkt. Durch die Sperrung des Systemkerns für den Zugriff durch Drittanbieter, sei auch ein Schutz der Sicherheits-Software nicht mehr möglich, hieß es. Symantec verwendet bei seinen eigenen Produkten Techniken, die mithilfe des Zugriffs auf den Kernel verhindern, dass die Software verändert werden kann. Genau diese Funktionen wären unter Windows Vista nicht mehr einsetzbar. Microsoft sieht den Kernelschutz hingegen als einzige Möglichkeit, den Trend zur Ausnutzung von Fehlern zu stoppen. Man will eine zusätzliche Hürde für Angreifer schaffen.

Die Sicherheitsmängel der existierenden Plattformen haben offenbar eine Industrie geschaffen, die den Mangel zementieren will, um auch in Zukunft Firewalls und Virenschanner verkaufen zu können. Aber sind Firewalls und Virenschanner tatsächlich nötig? Im Grunde bedeutet die Nutzung einer Firewall gleichzeitig das Eingeständnis, dass es um die Sicherheit unserer Plattformen extrem schlecht bestellt ist. Hier zeigt sich das abgrundtiefe Misstrauen der für die IT-Security eines Systems Verantwortlichen gegenüber den Benutzern des Systems sowie gegenüber der Sicherheit der Betriebssysteme und Applikationen. Diese Verantwortlichen haben gesehen, wie Entwickler die Grundelemente von Bedrohungsmodellen ignoriert haben (oder sie meist gar nicht kannten). Sie haben gesehen, dass Betriebssysteme wie Applikationen eine „alles oder nichts“-Eigenschaft bezüglich der Sicherheit an den Tag legen: Wenn ein Sicherheitsproblem auftritt, dann führt es meist zu einer kompletten Übernahme von Applikation oder System, das in der Folge sofort für weitere Angriffe verwendet wird.

Firewalls und Virenschanner können aber immer nur „after the fact“-Sicherheitsmekanismen sein, d. h. sie schützen nur vor bereits bekannten Attacken und wä-

ren bei einer besseren Softwarearchitektur gar nicht – oder zumindest nicht so sehr – wichtig. Dieser Erkenntnis stellt sich jedoch die Macht des Faktischen entgegen: „Man“ setzt Firewalls und Virens Scanner ein, weil sie eben nötig sind und weil es jeder tut.

Die Frage, warum sie nötig sind, führt schnell zurück in die Vergangenheit der Informatik, d. h. in die sechziger und siebziger Jahre des vergangenen Jahrhunderts, in denen zentrale Architekturentscheidungen getroffen wurden, die auch für die heutige Zeit noch bedeutsam sind – allerdings auf der Basis damaliger Hard- und Softwarekenntnisse. Die Grundlagen der kommerziellen Systeme von heute sind tatsächlich bereits so alt – sie wurden zu einem Zeitpunkt entworfen, als bezüglich der Sicherheit noch völlig andere Regeln galten. So existierte kein Markt für „Shrink-Wrapped“-Applikationen, sondern die meisten Anwendungsprogramme waren von den Nutzern selbst geschrieben bzw. selbst übersetzt. So erklärt sich die ungeprüfte Annahme gängiger Betriebssysteme, dass Applikationen grundsätzlich gut sind und den Willen ihrer Nutzer vertreten. Deshalb übergeben die Systeme den Applikationen beim Start alle Rechte ihrer Nutzer – ohne Einschränkung.

Die Dominanz der heute gängigen Systeme, Sprachen und Techniken ist so groß, dass alternative Konzepte völlig untergegangen sind (z. B. alternative Konzepte für Concurrency, die weniger gefährlich als das heutige Shared-State-Multi-Threading sind). So können heute längst nicht mehr bei allen Informatikern Kenntnisse der Konzepte funktionaler Sprachen vorausgesetzt werden. Anders gesagt: wer mit Access Control Lists und Sprachen wie C/C++ aufgewachsen ist, für den sind Virens Scanner und Firewalls eine architektonische Selbstverständlichkeit – wie auch sinnlose GUI-Dialoge, die die Nutzer zu Entscheidungen nötigen, die sie gar nicht treffen können.

So führt uns die Frage nach den Gründen, warum wir Firewalls und Virens Scanner brauchen, bis hinunter in die kleinsten Einheiten von Software: Funktionen und Objekte sowie deren Sicherheitseigenschaften. Im zweiten Teil dieses Kapitels (Abschn. 10.3.3) werden wir uns unter dem Stichpunkt „Mikro-Architektur“ speziell damit auseinandersetzen und dabei teilweise auf Literatur zurückgreifen, die Praktiker angesichts ihres Alters wohl schnell als archaisch oder irrelevant bezeichnen werden. Aber es ist eine Tatsache, dass die Informatik mittlerweile selbst eine Geschichte hat, die es lohnt zu studieren. Das haben beispielsweise auch die Entwickler von Microsoft Research getan, als sie für ihr Singularity-Betriebssystem Capability-Konzepte aus den 1970er-Jahren weiterentwickelt und an die heutigen Kenntnisse angepasst haben.

10.1.2 Die Frage der Verantwortung

Lohnt es sich überhaupt, sichere Software zu entwickeln? Sollte z. B. ein Browser wie Firefox wieder mehr Wert auf Sicherheit statt auf extreme Flexibilität und Ausbaubarkeit legen? Aus ökonomischer Sicht ist die Antwort darauf ein klares

Nein – vor allem nicht, wenn es um Software für den Endverbraucher geht. Diese sind schließlich in Bezug auf Software rechtlich in der gleichen hilflosen Lage wie gegenüber dem medizinischen Apparat oder der Lebensmittelindustrie. Das Risiko einer Fehlfunktion trägt in allen Fällen letztlich der Endkunde. Die Hersteller werden erst dann ein wirkliches (d. h. wirtschaftliches) Interesse an der Herstellung sicherer Software entwickeln, wenn ihnen auch die mit der Nutzung der Software verbundenen Risiken aufgebürdet werden.

Es ist anzunehmen, dass im Rahmen der rechtlichen und gesellschaftlichen Klärung der Verantwortung für Systeme und Software auch die Eigenschaft des Rechners als „universelle Maschine“ zur Sprache kommen wird – z. B. im Zusammenhang mit Digital Rights Management und TCP (Trusted Computing Platform). Was darf der Eigentümer eines Rechners oder von Software? Welche Rechte behält der Hersteller? Die Vorstellung, dass auch ein legitimer Eigentümer vielleicht nicht über universelle Rechte am erworbenen System verfügt, mag ungewohnt erscheinen, ist uns aber in anderen Lebensbereichen längst vertraut: Auch ein Auto ist keine „universelle Fahrmaschine“, sondern ein System, bei dem die gegenseitigen Verantwortungen von Hersteller, Werkstatt und Käufer genau definiert wurden. Verändert z. B. ein Kunde die interne Software zur Steuerung der Elektronik in seinem Auto, erlischt wahrscheinlich seine allgemeine Betriebserlaubnis und der Hersteller des Autos kann im Schadensfall – anders bei einem Unfall, der durch einen Ausfall der Elektronik bei unveränderter interner Software passiert – nicht haftbar gemacht werden. Die aktuelle Situation in der Softwareindustrie im Bezug auf Haftung sieht freilich anders aus, wie Firefox-Mitentwickler Blake Ross in [Ross] sehr deutlich macht:

“Perhaps the most compelling impetus to use dialogs in a product is liability. Dialogs force the user to decide so that the company doesn’t have to, a sleight of hand that the in-house lawyers appreciate later when it turns out that the user’s decision was wrong.”

10.2 Grundlegende Mechanismen sicherer Software

Wir haben in den vergangenen Kapiteln eine große Anzahl von Sicherheitsmechanismen und -prinzipien kennen gelernt. Jetzt ist es vielleicht an der Zeit, die Wichtigsten nochmals kurz zu diskutieren und sie vor allem zueinander in Beziehung zu setzen: Welche Einsatzgebiete erfordern bzw. fördern welche Mechanismen? Welche allgemeinen Regeln für den Einsatz von bestimmten Mechanismen gibt es?

10.2.1 *Filter*

Der Filter schränkt eine grundsätzliche Erreichbarkeit – die Möglichkeit, Ressourcen zu nutzen – ein. Er bedient sich dabei eines vorgegebenen Regelwerks. Da die Erreichbarkeit grundsätzlich nach wie vor besteht, liegt es an der Sinnhaftigkeit

und Konsistenz des Regelwerkes, die Sicherheit zu garantieren. Dies ist eine große Schwachstelle des Filters, die sich besonders bei anwachsenden Regelsätzen zeigt. Es gibt momentan keine formalen Verfahren, die Sicherheit von Firewall-Regeln zu prüfen.

Eine andere potenzielle Schwachstelle des Filters liegt im sogenannten *Bypass* (topologisch durch eine neue Route oder semantisch durch Definition eines anderen Namens für eine Ressource), der den Filterregeln verborgen bleibt.

Blacklists sind schlechte Filtertechniken, da sie neue gefährliche Typen von Requests nicht erfassen. Whitelists hingegen gehen grundsätzlich in die richtige Richtung, nämlich die Kommunikation zweier Agenten als Sprache zu formalisieren und damit automatisch prüfen zu können.

Ein Vorteil der Filtertechnologie besteht darin, dass sie *externalisierbar* ist, d. h. auch außerhalb der Applikation in vorgeschalteter Form funktionieren kann. Der Nachteil dieses Ansatzes ist allerdings, dass beispielsweise eine Input-Validierung nun an zwei Orten (Firewall und Applikation) erfolgen muss, da sonst eine zu große Abhängigkeit der Applikation von der Infrastruktur entstehen würde.

Bei komplexen Kommunikationsprotokollen zwischen Applikationen muss ein Teil der Applikationslogik in den Filter kopiert werden. So entsteht ein intelligenter Applikations-Proxy, der wie die Applikation selbst und mit ihr zusammen gewartet werden muss. Intelligente Proxies können auch Modifikationen an den Applikationsdaten durchführen.

10.2.2 Isolation

Bei der Isolation von Ressourcen oder Objekten handelt es sich nach wie vor um eine der wichtigsten Sicherheitstechniken in Software oder Infrastruktur. Dabei sind zwei Funktionsweisen zu unterscheiden: Isolation kann durch *Interception* hergestellt werden – eine Extremform eines Filters. Isolation kann aber auch durch grundsätzliche Nicht-Erreichbarkeit von Dingen hergestellt werden. Virtual Memory Management ist ein Beispiel für Interception-basierte Isolation, wobei die Regeln hier sehr einfach sind: Existiert ein Mapping der virtuellen Adresse auf eine physikalische Adresse, gelingt der Zugriff, ansonsten wird er unterbunden. Ein gutes Beispiel für eine grundsätzliche Nicht-Erreichbarkeit ist Fehlen von Referenzen auf Objekte. Die Trennung von Code durch verschiedene Versionen von Klassen über Classloader-Techniken stellt ebenfalls eine Form der Isolation durch Nicht-Erreichbarkeit dar und wird in OSGI verwendet.

Der zentrale Vorteil von Isolation durch Nicht-Erreichbarkeit ist, dass die Sicherheitsanalyse dadurch stark vereinfacht wird. Es brauchen keine Regeln für die Interception geprüft werden, da der Zugriff auf die Ressource schlicht nicht möglich ist.

Die meisten Systeme oder Applikationen kommen allerdings nicht mit Isolation aus, sondern müssen Kommunikation zwischen isolierten Komponenten gestatten, z. B. durch Interprozesskommunikation lokal oder über das Netzwerk. Damit öff-

nen sie neue Angriffsvektoren durch schlechte Input-Validation, schlechte kryptografische Schlüssel oder schlechtes Key-Management. Web-Applikationen oder Application-Server sind ein gutes Beispiel dafür.

10.2.3 *Hierarchische Modi*

Die Realisation von Sicherheitszonen durch unterschiedliche, meist hierarchisch angeordnete Modi ist eine weit verbreitete Technik. Paradebeispiel dafür ist die Unterscheidung zwischen dem „Privileged“- und „Non-Privileged“-Modus in CPUs. Lange galt dieses Zwei-Modi-Konzept als einfach und gleichzeitig ausreichend, um Betriebssysteme zu implementieren. Mittlerweile hat man aber erkannt, dass diese Alles-oder-nichts-Logik nur dazu führt, dass im privilegierten Modus sehr viele und mächtige Operationen ohne Unterschied möglich sind. Es ist jedoch sehr unwahrscheinlich, dass ein Modul wie z. B. ein Treiber alle Systemressourcen benötigt. So würde man bestimmte Privilegien wie den Hardwarezugriff für Ein/Ausgabe gerne an grundsätzlich weniger privilegierten Code auslagern. Leider zeigt sich beim Versuch, die Privilegien in verschiedene Modi zu unterteilen (in CPUs meist als Protection Rings implementiert), ein grundsätzliches Problem des modalen Ansatzes: Die Auswahl der Privilegien pro Level oder Ring tendiert dazu, zu Konflikten zu führen, da unterschiedlicher Code unterschiedliche Kombinationen von Privilegien benötigt, diese jedoch fix auf einige wenige Modi verteilt sind.

Letztlich führt die Verwendung nur weniger Modi zu einer sehr groben Vergabe der Privilegien, während viele Modi schnell unbeherrschbar werden. Ein Ansatz, der feingranular genau bestimmte Rechte z. B. durch Referenzen auf Objekte verteilt, ist hier deutlich besser. Das Microsoft-Forschungsprojekt Singularity [Hunt] wie auch sichere Sprachen wie „E“ verfolgen diesen Weg.

Daneben gibt es im modalen Ansatz ein weiteres Problem: Der Wechsel zwischen den Modi, wenn er als paarweiser Funktionsaufruf implementiert ist. Was passiert, wenn der Aufruf der Rückföhrfunktion vergessen wird? Klassische Beispiele für dieses Problem sind die Sperre der Interrupts in Kernels oder das Ausführen von Code im Superuser-Modus durch SetUID-Programme.

Zu den Problemen der mangelnden Rechtegranularität und der Problematik der Rückkehr in den alten Modus gesellen sich zwei weitere Sicherheitsprobleme: Der Missbrauch des privilegierten Codes durch den Aufrufer („Confused Deputy“ [Hardy]) und der Missbrauch von Ressourcen durch implizite Allokation.

In Abb. 10.1 betrachten wir dazu kurz die Interaktionsmöglichkeiten an der Schnittstelle zweier Modi oder Privilegienlevel.

Das Confused-Deputy-Problem lässt sich schnell zeigen: Der Aufrufer besitzt die Ressourcen B und D und möchte privilegierten Code aufrufen, um diese beiden Ressourcen auf bestimmte Weise zu bearbeiten. Statt den Namen von B und D verwendet er jedoch die Namen A und B. Die Ressource A ist jedoch im Besitz des privilegierten Modus oder Levels. Prüft nun der privilegierte Code die Umsetzung des Namens in die Ressource nicht genau (z. B. den Open System Call

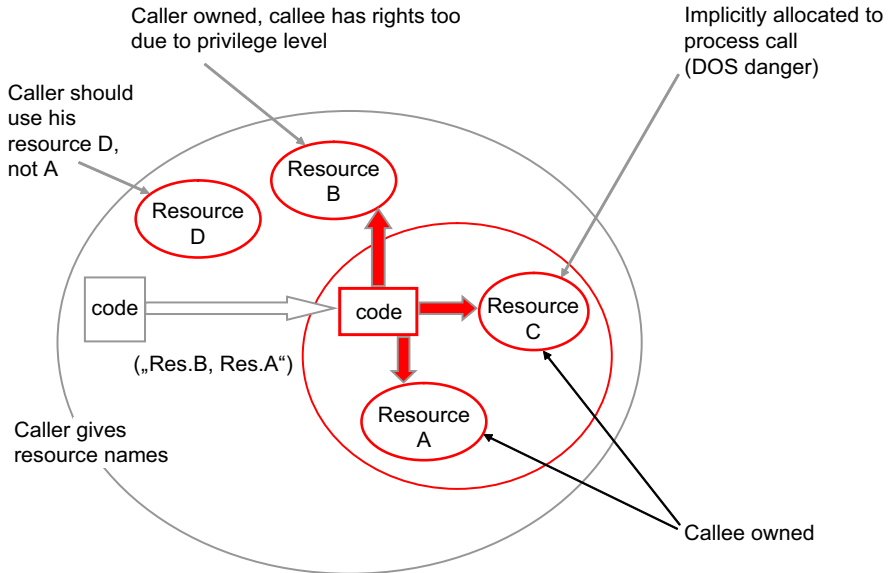


Abb. 10.1 Interaktion und Autorität

open(„B“, „A“)), dann wird statt der Caller-Ressource D die privilegierte Ressource A manipuliert – auf die der Aufrufer selbst keine Zugriffsrechte hätte. Der privilegierte Modus ist jetzt „confused“.

Schnell stellt sich auch die Frage wieso, der privilegierte Code ebenfalls die Rechte auf die Ressourcen des Aufrufers (B, D) besitzt? Er sollte statt den Namen dieser Ressourcen im Zuge des Aufrufs Capabilities erhalten, die den Zugriff gestatten. Dies würde die Ressourcen des Aufrufers vor dem privilegierten Level oder Code schützen. Dies war ein Kritikpunkt am Design der Java Language Security, bei der alle Beteiligten an einem Call gleiche, statische Zugriffsrechte auf die Ressourcen benötigten. Es ist jedoch möglich, dass der Aufrufer durch `doPrivileged()` und der Manipulation des Access Control Contexts die ursprünglich zur Verfügung stehenden Rechte deutlich verringert (siehe Kap. 14 zu den formalen Methoden).

Noch unangenehmer ist jedoch der Fall der Ressource C. Sie ist dem Aufrufer ohnehin nicht direkt zugänglich und er kann auch nicht durch Namensmanipulation den Code verwirren. Der privilegierte Code allokiert diese Ressource jedoch implizit, um den Call des Aufrufers zu verarbeiten. Dies ermöglicht Denial-of-Service-Attacken und damit einen Angriff auf die Liveness des Systems. Es geht dabei nicht um die Frage der grundsätzlichen Freigabe der Ressourcen. Nehmen wir an, dass bei Beendigung des Prozesses des Aufrufers der privilegierte Code die Chance erhält, die Ressourcen wieder freizugeben. Es geht vielmehr darum, dass der Aufrufer über die implizite Allokation das Verhalten des Systems massiv beeinflussen und andere Applikationen stören kann.

Im Kap. 4 zur Plattformensicherheit hatten wir kurz den Microkernel `seL4` angesprochen. Dieser Kernel verwendet zwei Techniken, um das Problem der impliziti-

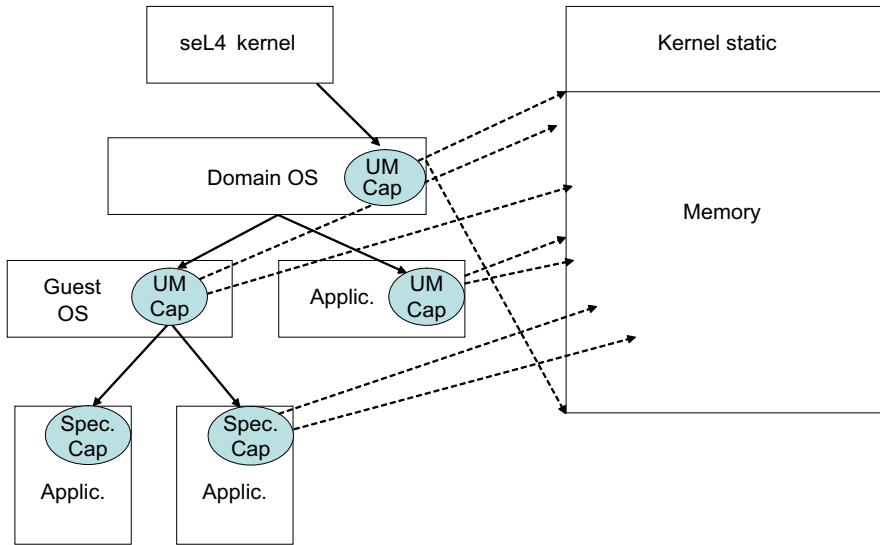


Abb. 10.2 Memory Capabilities, erweitertes Diagramm aus [EDE]

ten Allokation von Speicher grundsätzlich zu vermeiden: Er macht die Allokation explizit für den Caller (er verlagert die Allokation genau genommen sogar auf ihn) und er verwendet Capabilities, um dem Caller Zugriff auf einen genau abgesteckten und kontrollierten Bereich des Speichers zur eigenen Verwaltung zu übergeben [EDE]. Abbildung 10.2 zeigt die abgestufte, granulare Weitergabe von Speicherzugriffsrechten.

Bei den Capabilities auf Speicher handelt es sich um nicht-fälschbare Objekte, die granularen Zugriff auf Speicher erlauben. Besonders günstig daran ist, dass es den Empfängern solcher Capabilities möglich ist, diverse verschiedene Policies zur Speicherverwaltung zu implementieren. So ist es wahrscheinlich, dass das Gast-OS eine andere Policy fährt als spezielle Applikationen, die direkt auf dem Domain-OS laufen.

Die eben gezeigte Architektur stellt ein Grundmuster für capability-basierte Verwaltung von Ressourcen dar. Die Besitzer der Basis-Capabilities haben natürlich auch das Recht, die vergebenen Unterrechte zurückzunehmen. Interessant ist weiterhin, dass die Verwaltung der Capabilities (in Form von Bäumen) durch den Kernel im Speicherbereich der Capabilities selbst erfolgt, also auch da keine dynamische Allokation von Speicher im Kernel benötigt.

10.2.4 Identitäts- vs. autoritätsbasierte Zugriffskontrolle

Während kleine Plattformen wie Mobiltelefone oder PDAs gänzlich ohne ein Konzept von Identität auszukommen scheinen – ihr Besitz und die Kenntnis einer PIN ist ausreichend zur Nutzerauthentifikation – spielt das Konzept einer Identität im

Firmenumfeld eine zentrale Rolle, etwa für Abrechnungen etc. Die Frage ist, ob diese Business Sicht auch die richtige für den Aufbau sicherer Software ist. Sehen wir uns kurz die übliche Vorgehensweise einer Firmenapplikation an: Erst Authentisierung (also eine abgesicherte Identifizierung) und dann Autorisierung durch statische Zuteilung globaler Rollen und Rechte (d. h. ohne Rücksicht auf die geplante Aktion). Anschließend wird die Software ausgeführt – hoffentlich im Sinne des identifizierten Kunden und der Firma. Wie steht es um die Garantien dahinter?

Genauer Nachdenken zeigt, dass es zwischen der Tatsache der Verifikation einer Identität und der nachfolgenden Ausführung einer Aktion keinen wirklich gesicherten Zusammenhang gibt. Der erste an einer Transaktion auf Empfängerseite beteiligte Rechner mag noch an so einen Zusammenhang glauben – die Verbreitung etwa von Phishing-Techniken strafen diese Annahme jedoch Lügen.

Delegationstechniken, die alle Beteiligten protokollieren können, können die Situation verbessern. Ein gutes Beispiel dafür ist die CORBA-Delegation im „fully traced“-Modus. Der endgültige Empfänger einer Aktion kann so die Namen aller Beteiligten und deren Rechte verifizieren – einen Willensbeweis des Auftraggebers besitzt er jedoch nicht. Eine Transaktionssignierung durch den Kunden stellt einen solchen Willensbeweis dar. Sie reduziert die Menge des Vertrauens, die firmenseitig der Transaktion entgegengebracht werden muss, enorm und entlastet dadurch auch die Infrastruktur. Die durch die Signatur erreichte Nicht-Abstreitbarkeit wird üblicherweise als Sicherheit des Empfängers verstanden – in dem Sinne, dass ein Auftraggeber den Auftrag im Nachhinein nicht abstreiten kann. Die Nicht-Abstreitbarkeit erzeugt aber auch entscheidend mehr Sicherheit auf der Seite des Auftraggebers.

Die Transaktionssignierung stößt in der Praxis jedoch auf zwei grundlegende Probleme: Zum einen die sichere clientseitige Durchführung der Signatur auf einer potenziell unsicheren Rechnerplattform, zum anderen besteht innerhalb der Firmen-IT häufig ein grundsätzliches Unbehagen gegenüber kryptografischen Lösungen anstelle von Lösungen durch Prozesse und Infrastruktur.

Stimmt es tatsächlich, dass die neue Generation der kleinen, mobilen Devices ohne ein Konzept von Identität und daran gebundene Rechte auskommt? Das Java-2-Security-Beispiel zeigt klar, dass auch Code Access Control in Form einer Sandbox identitätsbasierte Zugriffskontrolle verwenden kann: Hier ist es die Identität des Codes, die die jeweiligen Rechte bestimmt – erneut ohne Berücksichtigung des tatsächlichen Kontexts, was schnell zur Vergabe von zu viel Autorität führt.

Miller et al. zeigen die grundsätzlichen Schwächen der identitätsbasierten Zugriffskontrolle am Beispiel der Entwicklung der Browser-Security bzw. von Office-Paketen auf. Die Ausführung von Scripts oder Macros kann um den Preis gefährlicher, ambienter Autorität gestattet werden, die dem Code automatisch zur Verfügung steht. Oder Scripts und Macros können abgeschaltet werden. In dem Fall ist der Preis die nicht mehr funktionierende Applikation. Somit „thrashed“ das System zwischen diesen beiden Polen (Abb 10.3), wenn weitere identitätsbasierte Mechanismen hinzugefügt oder abgeschaltet werden [MSLAS, S. 2ff.].

Und das ist *kein* Problem der Usability bzw. mangelnder User-Interfaces: Das System selbst kann Autorität nicht granular und sicher weitergeben. Es kann nicht

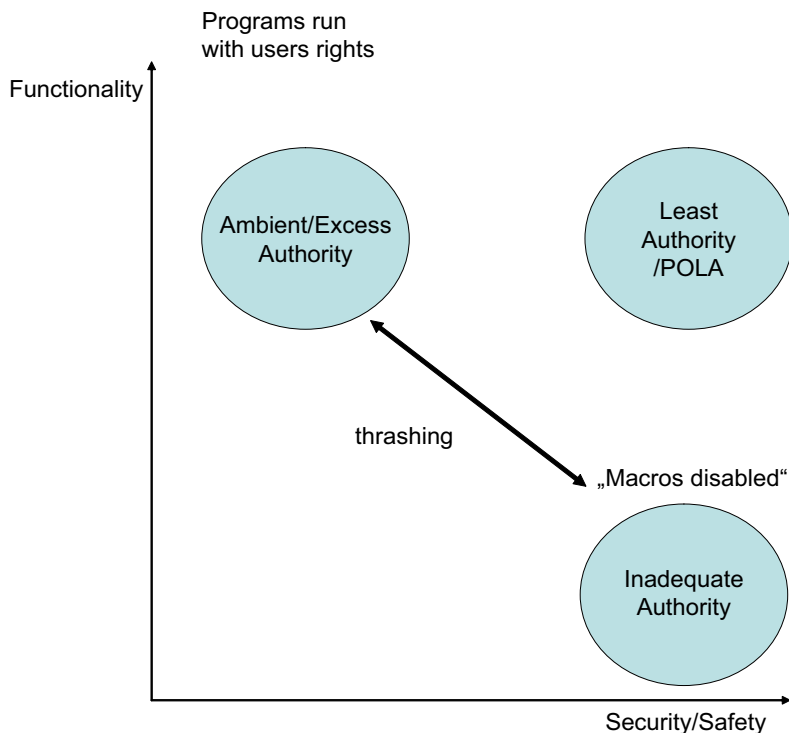


Abb. 10.3 Diagramm in Anlehnung an [MSLAS, S. 3]

die Ausführung von Macros auf dieses eine Dokument beschränken – das Anschalten von Macros gefährdet automatisch *alle* Dokumente des Benutzers. Man erinnere sich an dieser Stelle auch an die Funktionsweise von Referenzmonitoren, die auf statischen Regel-Files basieren: Auch dort wird zwangsläufig mit Wildcards gearbeitet, die Programmen dann Zugriffe auf große Mengen von Files gestatten.

10.2.5 Kanalbasierte Sicherheit vs. objektbasierte Sicherheit

Die obigen Überlegungen zur Transaktionssignierung spiegeln sich in der Frage nach den Grenzen kanalbasierter Sicherheit wieder. Es gibt dabei empfängerseitig nur eine sehr schwache Verbindung von Identität und Aktion. Selbst die Feststellung der Identität gilt nur für sehr kurze Zeit und es stellen sich schwierige Fragen für die Software:

- Wie lange soll die festgestellte Identität Gültigkeit haben?
- Wo soll die festgestellte Identität Gültigkeit haben (nur innerhalb des Systems, das die Authentisierung vollzogen hat, oder auch darüber hinaus)?

- Wie soll die verifizierte Identität softwaretechnisch gehalten werden?
- Wie soll sie an andere Server weitergegeben werden?
- Wie aussagekräftig ist die Feststellung einer Identität überhaupt?

Der letzte Punkt kann vielleicht unter Zuhilfenahme eines Beispiels aus der „wirklichen Welt“ besser erklärt werden: Im Zusammenhang mit der Sicherheitsüberprüfung von Passagieren auf Flugplätzen taucht öfters die Forderung nach spezieller Behandlung von Passagieren auf, die bereits in der Vergangenheit einer Sicherheitskontrolle unterzogen worden sind und dies durch einen Ausweis oder ähnliches belegen können. Soll dieser Personenkreis ohne weitere Prüfung das Flugzeug betreten können oder vor jedem Flug neu überprüft werden?

Die spontane Antwort vieler wird hier sicherlich lauten: Personen mit entsprechendem Ausweis sollten kein weiteres Mal kontrolliert werden müssen. Der Aufwand, solche Ausweise fälschungssicher zu erstellen, weiterzuleiten und zu prüfen, wäre jedoch enorm. Für die Erstellung müsste eine – wahrscheinlich erst neu zu schaffende – allgemein anerkannte Behörde mit verlässlichen Sicherheitsprozeduren verantwortlich sein. Wie Bruce Schneier in [Schn06] völlig zurecht hervorhebt, steckt hinter der spontanen Antwort jedoch noch ein anderer Irrtum: Auch der größte Aufwand bei der Vorab-Authentisierung kann nie die Sicherheit erreichen, die durch eine konkrete Personenprüfung vor Ort auf Waffen, Sprengstoff etc. erreicht werden kann. Der schnelle Schluss von der bekannten Identität auf die Ungefährlichkeit ist leider falsch.

Zurück in der IT-Welt drängt sich der Verdacht auf, dass die Menge der Infrastruktur, die serverseitig aufgefahren wird und die Menge der Sicherheitsstandards und Technologien, die bei der Authentifikation verwendet werden, letztlich nur über die relative magere Form von Garantien, die dabei entstehen, hinwegtäuschen soll.

10.2.6 Capabilities

Im obigen Abschnitt ist das Misstrauen gegenüber der Ableitung von Autorität aus einer bloßen Feststellung einer Identität bzw. dem Eigentum an Ressourcen deutlich geworden, da so ein zu breiter Zugriff gestattet wird und das POLA (*Principle of Least Authority*) Prinzip verletzt wird.

POLA lässt sich durch Capabilities auf eine sehr granulare Weise erreichen. Wir werden in Abschn. 10.1.22) Beispiele sehen, wie Objektreferenzen als Schlüssel wirken können, und zwar sowohl lokal als auch verteilt, wie Abb. 10.4 belegt. Objektbasierte Sicherheit kann wie granulare Capabilities wirken und ist leicht zu delegieren.

Neben den bisher genannten Objekt-Capabilities gibt es eine ganze Reihe von anderen Formen, die z. B. auf dem Prinzip der Nicht-Erratbarkeit beruhen. Populär geworden sind sog. Web-Capabilities, die in Form von Schlüsseln an URL angehängt werden und dem Besitzer dieser Web-Capabilities Zugriff auf die Res-

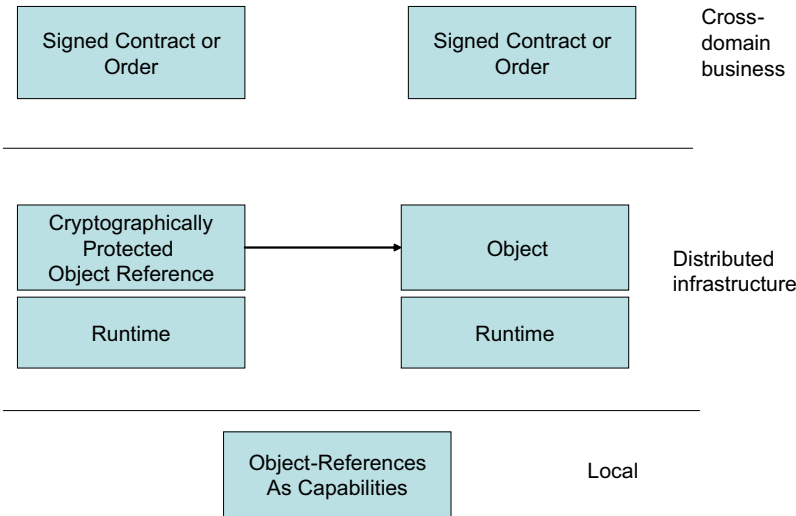


Abb. 10.4 Formen von Capabilities

sourcen verschaffen. Ein einfaches Beispiel für derartige Capabilities ist der Service „Doodle“ (www.doodle.ch), der mit URLs wie den folgenden arbeitet: <http://www.doodle.ch/wt9bmuy8c6hhphm6>. Nutzer von Doodle können einen geplanten Termin dort eingeben und eine Abstimmung einleiten. Alle Beteiligten erhalten die URL per Mail und können nun – ohne Authentisierung oder weitere Autorisierung – direkt ihre Stimme abgeben. Filesharing durch Capabilities wird z. B. durch die Website <http://drop.io/> angeboten, die ebenfalls die einzelne Resource über URL-Schlüssel zugreifbar macht. Waterken-URLs erlauben eine Capability-basierte Kommunikation zwischen Web-Clients und Web-Applikationen, siehe [Close] Waterken-Server <http://waterken.sourceforge.net/>.

Zusammen mit den oben dargestellten Problemen mit der Authentisierung in verteilten Systemen, ergibt dies einen weiteren Grund für die Nutzung von Capability-Mechanismen. Ihre Granularität und Unabhängigkeit von konkreter Infrastruktur werden Capabilities zu einem der Sicherheitsmechanismen der Zukunft machen.

10.2.7 Schlüssel

In einem System, das auf objektbasierter Sicherheit beruht, kommt den Schlüsseln, die zur Erzeugung von Signaturen bzw. MACs genutzt werden, eine zentrale Bedeutung zu. Schlüssel dienen hier vor allem zur Erzeugung neuer Schlüssel, wie am Beispiel von Proxy-Zertifikaten deutlich wird. Das Problem der Behandlung von Schlüsseln auf unsicheren Desktops haben wir bereits im Zusammenhang mit Grid-Computing und der Finread-Infrastruktur behandelt. Im Kap. 13 zu Usability

werden wir ein weiteres großes Problem im Zusammenhang mit Key-Management besprechen, nämlich das automatische Key-Management am Beispiel von sicherer E-Mail wie von Garfinkel in [Garf] vorgeschlagen. Die Gefahr symmetrischer Schlüssel hat sich am Beispiel von Application Server Cells nochmals deutlich gezeigt: Nur ein kompromittierter Node und die ganze kryptografische Absicherung einer Zelle ist hinfällig.

10.2.8 Zugriffskontrolle in lokalen und verteilten Systemen

Funktioniert eigentlich die Zugriffskontrolle der Java 2 Security, basierend auf einem Stack-Walkback auch verteilt? Und was wäre damit gewonnen? Natürlich gibt es zwischen verteilten Systemen keinen direkten gemeinsamen Stack. Aber bevor wir uns überlegen, wie dieser simuliert werden kann, lassen Sie uns nochmals einen Blick auf die Java 2 Security in Bezug auf die Access Control werfen. Die geforderte „Complete Mediation“-Eigenschaft der Kontrolle wird dort durch folgende Dinge erzeugt:

- Alle beteiligten Klassen sind bekannt (evtl. sogar durch Signaturen ihrer Hersteller).
- Die Reihenfolge der Aufrufe ist bekannt, d. h. der Aufrufspfad zwischen Sendern und Empfängern von Methodenaufrufen ist jederzeit zur Laufzeit nachprüfbar.
- Ein Pfad ist dann legal, wenn er einer Permission gehorcht – alle beteiligten Klassen besitzen diese Permission. Eine Permission kann auch eine Rolle eines Subjects sein.
- Der Kontrollpunkt der Zugriffskontrolle wird von Programmierern durch Aufruf von `checkPermission()` festgelegt.
- Bei einem Aufruf von `doPrivileged()` verschwindet der bisherige Call-Stack für die Zugriffskontrolle, d. h. eine Kombination von Permissions findet nicht statt.
- Die Prüfung eines Aufrufpfades kann lokal und mit akzeptabler Performance durchgeführt werden.

Bei der Besprechung der Java 2 Security stellte sich heraus, dass das Einsetzen der Kontrollpunkte (z. B. nur bei der Erzeugung oder auch bei der Verwendung von Objekten) sowie die gezielte Verwendung von privilegierten Aufrufen die Kernprobleme der Architektur darstellen. Eine Schichtenarchitektur vorausgesetzt, macht es für die Aufrufer von Methoden auf höheren Schichten meist keinen Sinn, direkte Permissions für Klassen unterer Schichten zu besitzen.

Vergleichen wir nun die Mechanismen lokaler Kontrolle mit den Möglichkeiten eines Servers, den Aufruf einer seiner Funktionen zu überwachen. Der Server kennt zunächst weder den Code des Aufrufers noch dessen Identität. Eine Authentisierung gibt dem Server die Möglichkeit, die Rechte des Aufrufers zu erfahren. Der Code selbst muss als „trusted“ angenommen werden. Interessant ist die Frage

nach dem Aufrufspfad in verteilten Systemen: Ohne spezielle Maßnahmen erfährt ein Server höchstens von seinem direkten Aufrufer, und er kann diesen Aufrufer seinerseits auch nicht an weitere Server weitergeben.

Delegation stellt nun Mechanismen zur Verfügung, den lokalen Stack im verteilten System zu simulieren, indem jeder beteiligte Server seine Aktionen protokolliert und dieses Protokoll weitergereicht wird. Ohne kryptografische Absicherung ist dies lediglich eine Information, die komplettes Vertrauen in die Kette der Server voraussetzt. Mit Signaturen im Protokoll hingegen kann ein kompletter Trace des Aufrufs erstellt werden (sogenannte *traced delegation*), die äquivalent zur Kontrolle im lokalen System verwendet werden kann. Ein Server kann anhand des Traces den kompletten Verlauf des Requests verfolgen und die Rechte der beteiligten Server prüfen. Kann nun ein Intermediate-Server einen nicht-verfolgbaren Call auf einen anderen Server durchführen? Wenn dieses System ein Rogue-Server ist, d. h. mit bössartiger Absicht arbeitet und z. B. Kundendaten protokolliert, dann ist das auch durch Traced Delegation nicht verhinderbar. Handelt es sich um einen „legalen“ Server innerhalb der Infrastruktur, dann kann dieser einen Trace des Aufrufs verlangen. Wenn der aufrufende Server nicht privilegiert arbeiten darf (d. h. ohne Call Trace seiner Aufrufer), dann wird der Server den Request ablehnen. Programmierfehler können auf diese Weise ebenfalls entdeckt werden. Über die nötigen Permissions der beteiligten Server, die zur Kontrolle in einem gemeinsamen Repository verwaltet werden müssen, bleibt dieser Mechanismus an eine Domain gekoppelt. Weder im lokalen noch im verteilten Fall wird die ursprüngliche Intention des Aufrufers transportiert.

Eine Entkoppelung von konkreter Infrastruktur muss genau dies erreichen, also die Absicherung der Intention des Aufrufers durch kryptografische Maßnahmen wie Signaturen. So ergibt sich aus der Abhängigkeit des Kontrollmechanismus von der Infrastruktur die Notwendigkeit objektbasierter Sicherheit in Cross-Domain-Applikationen, wie sie z. B. Web-Services vorsehen.

10.2.9 Infrastruktur vs. Application Level Security

Das eben Gesagte spielt auch eine große Rolle für das Verhältnis zwischen Infrastruktur und Applikation in Bezug auf Security. Traditionell tendieren verteilte Systeme dazu, maximale Transparenz anzustreben. Eine Applikation, die auf Java 2 Security beruht, bemerkt von der Zugriffskontrolle wenig, deshalb sollen auch verteilte Applikationen von der tatsächlichen Verteiltheit des Aufrufpfades nichts bemerken. JAVA EE verbannt Security in die Infrastruktur der Application-Server.

Mit der Entstehung der Web-Services und dem Drang nach Cross-Domain-Geschäftsvorgängen entsteht ein Gegentrend: Requests beinhalten signierte Intentionen der Auftraggeber, die für die Applikationen Aufträge, Tickets, Vollmachten etc. darstellen.

Natürlich können die beteiligten Signaturen oder Keys von Infrastruktur geprüft werden (und sollten auch geprüft werden, z. B. in Firewalls). Dennoch wandert wieder mehr Security in die Applikation, allerdings auf einem höheren semantischen Niveau. Gleichzeitig wird dadurch das Vertrauen in die Infrastruktur weniger kritisch. Paradebeispiel dafür ist die Bedeutungslosigkeit von Man-in-the-Middle-Attacken, wenn eine Transaktionssignierung durch Signaturen durchgeführt wird.

So ist in Zukunft mit mehr Applikationen zu rechnen, die mit Tokens, Vollmachten, Aufträgen etc. als Geschäftsobjekten direkt umgehen, statt auf sichere Kanäle zu vertrauen. Diese Applikationen werden diese Geschäftsobjekte viel leichter weitergeben können, als bisher über sichere Kanäle und Sessions.

10.2.10 Front-End vs. Back-End Security

In der Architektur von Infrastrukturen, Systemen und Applikationen gehen wir oft vom Architekturmuster des Trichters aus. Der Trichter soll Angriffsmöglichkeiten im Verlaufe von Requests minimieren, aber auch die möglichen Rechte und Autoritäten im Fall eines Angriffs minimieren, um Schaden zu begrenzen.

Stimmt dieser Ansatz auch für das Verhältnis von Front-End und Back-End in verteilten Multi-Tier-Applikationen? Tatsächlich findet im Laufe eines Requests eines Internet-Clients auch eine teilweise Ausweitung von Rechten und Zugriffsmöglichkeiten statt. Am Eingang einer Firmeninfrastruktur wird eine strenge und frühe Authentisierung des Clients empfohlen. Anschließend wird der Request des Client am Besten durch Impersonation seiner Identität durch die beteiligten Server durchgeführt, um sicherzustellen, dass bei jeder Zugriffskontrolle die tatsächlichen Rechte des Aufrufers verwendet werden und nicht die des Servers. Wenn der Aufruf in Richtung der großen Datenbanken auf Mainframes transportiert wird, stellt man jedoch sehr häufig fest, dass die vorgelagerten Server mit ihren eigenen Rechten auf die Back-Ends zugreifen, sozusagen „im Auftrag von Client X“ arbeiten anstatt „als Client X“.

Dies ist durchaus normal und findet ja auch in der lokalen Java-2-Zugriffskontrolle im Falle von `doPrivileged()` statt. Es ist auch nicht einfach, auf Back-Ends die Ressourcen eines Servers von denen eines Kunden zu trennen. Wenn jedoch der Zugriff des Servers auf Back-Ends zu breit wird, empfiehlt es sich, durch einen zugriffsfilternden Softwarebaustein eine Reduzierung des Zugriffs zu erreichen. Dies ist allerdings eine Reduktion durch Verhalten des Codes und nicht durch Reduktion der Rechte zu erreichen. Wir werden die Reduktion von Autorität durch Verhalten im Zusammenhang mit Object Capabilities noch genauer diskutieren und Softwarebausteine dafür vorstellen. Der RPC-Layer zur Eingrenzung von Datenbankzugriffen im Kapitel zur Absicherung von Servern ist ein Beispiel für diese Technik.

Ideal im Sinne von POLA wäre es, wenn ein Intermediate-Server dynamisch – d. h. durch Referenzen – Capabilities zum Zugriff auf Client-Daten erhalten würde

und ohne diese Capabilities kein Zugriff möglich wäre. Beim Zugriff auf Back-Ends wird erneut auch die Problematik statischer Eigentumsrechte deutlich: Sie sind unabhängig vom Zweck des Zugriffs vorhanden und erlauben viel zu breiten Zugriff.

10.2.11 Föderation

Die zentralen Erkenntnisse zu Föderation als Sicherheitsmechanismus waren:

- Eine direkte Authentisierung sollte nur durchgeführt werden, wenn die damit verbundenen Pflichten (Überwachung der Gültigkeit des Accounts) erfüllt werden können (z. B. Bestellvorgänge von Firmen bei Lieferanten).
- Personenbezogene Daten und Geheimnisse sollten einen Identity Provider entweder gar nicht verlassen, oder kontrolliert und evtl. anonymisiert (Weitergabe einer authentisierten Sitzung).
- Den Token (also signierte Aussagen zur Sicherheit) kommt in der Zukunft eine zentrale Rolle zu, aber:
- Das Problem der Semantik von Aussagen zur Sicherheit muss gelöst werden.

10.2.12 Basismechanismen der Zugriffskontrolle

Dieser Abschnitt soll noch einmal die Basismechanismen der Zugriffskontrolle Revue passieren lassen. Sie sind bei den jeweiligen Techniken bereits erwähnt worden, aber es lohnt sich, sie einmal einander gegenüberzustellen.

Die Mechanismen der Zugriffskontrolle auf Ressourcen sind die folgenden:

- Access Control Lists (ACLs) (statisch, instanzbasiert, Rechtevergabe pro Resource, ambiente Autorität, Verwaltungsproblem, Performanceproblem, [Bish]),
- Role Based Access Control (RBAC, Skalierungsproblem, ambiente Autorität),
- Sandbox durch Stack-Walking (Konfigurationsproblem),
- Namespaces (statisch, komplex, unvollkommene Isolation?),
- Object Capabilities (dynamische, im Code platzierte Autoritätsbehandlung),
- Tagging/Labeling/Tainting (komplexe Architektur vorab nötig) und
- Signaturen, objektbasierte Sicherheit.

Die wohl beste Erklärung von Access Control Lists leistet Michael Bishop in [Bish]. Ausgehend von der globalen Protection-Matrix eines Systems, bei der Benutzer und Ressourcen über zulässige Aktionen verknüpft werden, zeigt Bishop

ACLs als Technologie, mit der die nur lückenhaft gefüllte Protection-Matrix (manche Benutzer haben gar keine Rechte bei bestimmten Ressourcen und diese Zellen bleiben dann leer) effizient bei den Ressourcen gespeichert werden können. ACLs sind discretionary, d. h. sie vergeben Zugriffsrechte über Eigentum an Ressourcen und erlauben die Weitergabe solcher Rechte durch die Eigentümer. Bei der Role Based Access Control hingegen werden die Rechte für einzelne Rollen aufgrund einer zentralen Policy vergeben.

ACLs und RBAC stellen Mechanismen für die rein identitätsbasierte Autorisierung dar. Alle anderen Möglichkeiten vermischen die Rechtezuweisung aufgrund von Nutzeridentitäten und aufgrund von Codeherkunft.

10.3 Sicherheit und Softwarearchitektur

Vieles in diesem und den nächsten Kapiteln orientiert sich an den Forschungen zu sicheren Sprachen und Betriebssystemen, wie sie zurzeit im Umfeld der Sprache „E“ durchgeführt werden. Mark S. Miller hat dies unlängst in seiner Ph. D. Thesis [Mil] zusammengefasst. Die folgenden Seiten verwenden Beispiele und Konzepte daraus.

10.3.1 Perspektiven auf Software

Sicherheit ist nur sehr selten eine gleichberechtigte Sichtweise auf Software. Unter den „Architectural Views“ auf Software dominieren im Normalfall ganz andere Aspekte, wie Flexibilität und Erweiterbarkeit sowie Kompatibilität über verschiedene Versionen hinweg. Wir haben dieses Thema im Zusammenhang mit der Evolution von Eclipse APIs bereits einmal kurz angesprochen. Dort war der Fokus klar auf dem Problem der kompatiblen Erweiterung des Frameworks, ohne existierende Plug-Ins zu brechen. Deshalb wurden Konventionen festgelegt, welche Teile der Software nur durch das Framework verwendet werden dürfen und welche Extensions zur Verfügung stehen. Wichtig ist in diesem Zusammenhang die Erkenntnis, dass die Java-Sprachmittel wie `private`, `protected`, `public` sowie Interfaces und Abstract Classes es nicht erlauben, diese Trennung zwischen Kern und Erweiterung so zu definieren, dass eine bruchlose Erweiterung des Frameworks möglich wäre. Das Eclipse-Team musste eigene Regeln definieren sowie ein zusätzliches Attribut für Methoden einführen: `isAPI` – d. h. die Methode kann von Extensions verwendet werden.

Betrachtet man die gängigen Design Patterns, z. B. die aus dem bekannten Buch der „Gang of Four“ [GHJV], dann fällt auf, dass die meisten dieser Pattern sich mit dem Problem mangelnder Flexibilität gerade von statisch getypten Programmiersprachen wie C++ beschäftigen. Die Lösungen bestehen im Allgemeinen aus dem Einführen einer weiteren Indirektion sowie der genauen Kontrolle der

Instanziierung von Klassen, d.h. der Operator `new` wird als gefährlich für die Erweiterbarkeit erkannt und durch Factories gekapselt. Warum aber bedroht `new` die Erweiterbarkeit und Flexibilität einer Frameworks oder einer Applikation? Schlicht aufgrund der Tatsache, dass an dieser Stelle im Programm keine andere Implementation verwendet werden kann (eine evtl. Spezialisierung für einen Kunden), sondern dass genau die eine hinter `new` angegebene Klasse verwendet werden muss.

Was bedeutet nun `new` für die Sicherheit einer Applikation? Mit dem Erzeugen eines neuen Objektes erweitert eine Applikation ihre Zugriffsmöglichkeiten auf den Code. Das neue Objekt besitzt Methoden, die aufgerufen werden können. Erweitert die Applikation dadurch ihre Rechte? Besitzt sie jetzt eine größere Autorität, mithin größere Fähigkeiten? Die Kernfrage hinter diesen Fragen ist, wie durch das Laden von Code die Möglichkeiten einer Applikation, gewisse Effekte zu erzielen, verändert werden.

Welche Regeln sollten gelten, wenn man nicht die Erweiterbarkeit, sondern die Sicherheit einer Applikation in den Vordergrund stellt? Diese Sicherheit soll auch angesichts von Erweiterungen gelten. Dieser Frage werden wir in den nächsten Abschnitten nachgehen, und zwar auf drei verschiedenen Ebenen der Softwarearchitektur: Der Mikro-Architektur, den Bausteinen sicherer Software sowie der Makro-Architektur, bestehend aus der Komposition größerer Teile.

10.3.2 Zur Bestimmung der Gefährlichkeit

Welches Gefahrenpotenzial besitzt eine bestimmte Software? Da die Bestimmung der Sicherheit immer auch eine Abwägung von Risiken und Vorteilen bedingt, ist die Frage nach der Gefährlichkeit zunächst subjektiv. Wir erreichen mehr, wenn wir statt nach der Gefährlichkeit nach den Effekten fragen, die eine Software erzielen kann – ohne sie bezüglich Gefährlichkeit bewerten zu wollen. Aber selbst diese Vereinfachung führt schnell in weitere Probleme: Welche Effekte eine Software erzielen kann, hängt zunächst einmal vom enthaltenen Code ab. Versucht man durch eine Codeanalyse die Effekte zu bestimmen, stellt man schnell fest, dass die Effekte auch von äußeren Einflüssen abhängen, z.B. von der Zugriffskontrolle durch das Betriebssystem, ausgedrückt z.B. durch ACLs der Ressourcen oder durch ein Policy-File wie bei Java, bei dem interne Zugriffe innerhalb der Software geprüft werden. In beiden Fällen ist jedoch die Eingrenzung der Effekte ein von außen eingefügtes Moment, das bei einer Inspektion des Codes allein nicht sichtbar ist. Das ist scheinbar nur ein kleines Ärgernis, hat jedoch für die Sicherheitsanalyse von Software dramatische Konsequenzen, wie noch zu sehen sein wird.

Mark S. Miller verwendet in [Mil] die Begriffe „Permission“ und „Authority“, um die Effekte von Software besser beschreiben zu können. Eine „Permission“ stellt eine explizite Erlaubnis dar und kann technisch z.B. durch die Konfiguration eines Referenzmonitors ausgedrückt werden. „Authority“ hingegen wird als Kausalität verstanden, mit der Effekte erzielt werden – egal, ob erlaubt oder nicht.

Technisch lässt sich dies gut mit dem Begriff der „Erreichbarkeit“ erklären: Wenn eine Software durch Verwendung von Aufrufen Effekte erzielen kann, dann müssen diese Aufrufe natürlich erreichbar sein. Erreichbarkeit ist somit das Gegenstück zur Isolation. Erklärt man also den Begriff der Erreichbarkeit als Möglichkeit zum Erzielen eines Effekts, so zeigt sich: Erreichbarkeit drückt ein Potenzial aus. Eine Software erhält eine neue Bibliothek zum Aufruf externer Funktionen. Eine externe Zugriffskontrolle unterdrückt jedoch diese Aufrufe. Hat sich nun die Erreichbarkeit bzw. Authority der Software durch die neue Bibliothek geändert? Nicht im Sinne des kausalen Erzielens von Effekten, jedoch im Sinne des erweiterten Potenzials zum Erzielen solcher Effekte. Wir möchten deshalb neben den Begriffen „Authority“ und „Permission“, wie sie von Miller gebraucht werden, noch den Begriff des „Potenzials“ hinzunehmen, um die Gefährlichkeit von Software besser analysieren zu können.

Was bestimmt nun die Erreichbarkeit von Ressourcen, um Effekte zu erzielen? Man kann einem Programm z. B. Ressourcen beim Aufruf übergeben, im Sinne geöffneter File-Deskriptoren, wie im Falle von Unix. Dies ist unbedingt von dem Fall zu unterscheiden, in dem einem Programm nur Strings als Argumente übergeben werden und das Programm anschließend z. B. das File-System benutzt, um ein File zu öffnen. Hier besitzt das Programm Rechte, die *nicht* beim Aufruf übergeben werden. Miller bezeichnet dies als „Endowment“, d. h. eingebaute bzw. bei der Initialisierung entstandene Rechte.

Aber auch innerhalb eines Programms spielt die Erreichbarkeit von Code oder Variablen eine große Rolle. Sie bestimmt, welche Codestücke als potenzielle Autorität gelten können. Je mehr Teile einer Software untereinander erreichbar sind, desto schwieriger wird die Wartung, aber auch die Sicherheitsanalyse. Dies fängt bei der Gestaltung von Objekt-Interfaces an und geht bis zur Verwendung von Verzeichnissen. Es wird noch zu zeigen sein, dass die Frage der Erreichbarkeit alle Ebenen einer Softwarearchitektur betrifft: Mikro-Architektur, Bausteine und Makro-Architektur.

10.3.3 Mikro-Architektur

Die Mikro-Architektur einer Software besteht im Wesentlichen aus folgenden Aspekten:

- Laden der Software,
- Laden von Erweiterungen und
- Beziehungen der Objekte untereinander.

Wenden wir uns zunächst dem Themenkomplex „Laden und Linken“ zu und vergleichen wir zwei verschiedene Lademethoden: Statisches Linken und Laden sowie dynamisches Laden von Code. Wir unterscheiden dabei zwischen Permission und Authority im obigen Sinne, also zwischen der expliziten Erlaubnis, etwas zu tun, und der praktischen Möglichkeit, einen Effekt zu erzielen.

Beim statischen Linken, wie wir es bereits am Beispiel der Kernel Extensions kennen gelernt haben, werden zur Compile-Zeit externe Referenzen aufgelöst und Zugriffsmöglichkeiten geschaffen. Dies geschieht automatisch. Anschließend kann das Programm geladen und ausgeführt werden. Betrifft das statische Linken nun Permissions oder Authority?

Um diese Frage beantworten zu können, müssen wir uns das Verhältnis von Programm und Umgebung ansehen. Durch den Linkvorgang ändert sich die innere Struktur des Programms, Module oder Komponenten werden miteinander verbunden und auf Variable kann zugegriffen werden. Dies bedeutet zunächst eine Ausdehnung der Autorität innerhalb der Applikation, d. h. eine Reduktion der Isolation. Nach außen hin, d. h. zur Umgebung des Programms (z. B. zum Betriebssystem), kann das Programm nun ebenfalls mehr Autorität besitzen, beispielsweise mehr System-Calls aufrufen. Es hat sich also die Möglichkeit, Effekte zu erzielen verändert. Dies bedeutet jedoch noch nicht, dass sich die tatsächliche Autorität verändert hat. Eventuell scheitert ein Aufruf an der fehlenden Erlaubnis.

Wir halten an dieser Stelle fest, dass eine einfache Sichtweise von Autorität als einem kausalen Zusammenhang zwischen Software und durch sie erzielte Effekte und Permissions als Untermenge davon, nicht ausreicht. Wir müssen auch die Möglichkeit, Effekte zu erzielen, einbeziehen. Außerdem müssen wir uns fragen, ob die technische Trennung von Permissions und Authority überhaupt Sinn macht.

Betrachten wir zunächst wieder eine Applikation, die statisch mit Modulen und Bibliotheken verlinkt wird. Permissions werden in Form von ACLs gesetzt, die beim Zugriff auf Ressourcen des Betriebssystems geprüft werden. In diesem Fall ändert sich durch das Hinzulinken weiterer Bibliotheken nichts an den Permissions. Es steigern sich lediglich die Möglichkeiten, Effekte zu erzielen (potenzielle Autorität), und soweit diese mit den Permissions verträglich sind, wird auch die tatsächliche Autorität des Programms gesteigert.

Für die Analyse von Sicherheit in einem Programm müssen wir also feststellen, dass wir nur schwer entscheiden können, ob der Code eines Programms gefährlich ist, denn dazu müssen wir auch die Permissions mit einbeziehen, welche außerhalb des eigentlichen Programms angesiedelt sind.

Wie ist nun die Situation, wenn nicht alleine das Betriebssystem Zugriffe prüft, sondern ein interner Referenzmonitor Aufrufe innerhalb des Programms prüft (z. B. anhand einer Konfigurationsdatei, wie im Falle einer Java Policy oder einer Systrace-Konfiguration)? Hier ist entscheidend, wie granular der Referenzmonitor arbeitet. In den meisten Fällen wird er keine Zugriffe auf globale Variablen prüfen können, sondern lediglich Funktions- oder Methodenaufrufe. Auch hier stellen wir fest, dass neu hinzugelinkter Code die potenzielle oder tatsächliche Autorität steigern kann. Beunruhigend ist, dass der Code durch globale Zugriffe Autorität erlangen kann – vorbei am Referenzmonitor, wenn dessen Granularität zu hoch ist.

Wie sollten sich Erreichbarkeit und Laden von Code generell verhalten? Diese Frage führt uns schnell zurück auf die Frage nach dem Ursprung von Autorität bzw. Erreichbarkeit: Wo kommen eigentlich die Referenzen her? Miller spricht in diesem Zusammenhang von „Loader Isolation“ und argumentiert dafür, dass geladener Code generell keine Autorität haben, sondern alle wichtigen Referenzen

anschließend als Parameter bekommen sollte. Dies entspricht dem Prinzip der Inversion-of-Control-Architekturen.

Inwieweit entspricht nun z. B. Java dem Prinzip des gefahrlosen Ladens? Hier müssen zwei verschiedene Dinge betrachtet werden: Zum einen der Zugriff auf Klassen, der in Java ja von den sogenannten *Classloadern* reguliert wird, zum anderen die Autorität, die durch den Besitz eines Klassenobjektes allein erreichbar ist. Hier sind es vor allem die statischen Variablen und Funktionen, die ohne den Besitz eines Objektes erreichbar sind.

Im Kap. 5 haben wir gesehen, dass das klassische Verfahren zum Laden von Code bei Java in hierarchisch angeordneten Classloader-Instanzen besteht. Ein Request für eine Klasse wird u. U. bis an die Spitze der Hierarchie weitergeleitet und von dort befriedigt. Dies bedeutet im Allgemeinen, dass auf eine große Menge von Klassen potenziell von Applikationen und deren Objekte zugegriffen werden kann. Die Betrachtung von OSGI und modernen Application-Servern hat uns aber gezeigt, dass es vorteilhaft sein kann, von der hierarchischen Anordnung der Classloader abzuweichen und zumindest einzelne Bibliotheken durch horizontale Anordnung vor den Applikationen zu verstecken. OSGI zwingt nun allen Code dazu, seine Abhängigkeiten von anderen Bibliotheken etc. genau zu spezifizieren, ebenso die von ihm exportierten Klassen. Dadurch können Classloader nicht nur die Kompatibilität von Klassen prüfen und sicherstellen, sondern auch verhindern, dass Code ohne Erlaubnis andere Klassen verwendet. Wartung und Sicherheit von Software werden so deutlich verbessert.

Weniger sicher hingegen ist die Verwendung von statischen Variablen oder Funktionen. Wir haben im Zusammenhang mit der Wartbarkeit von Software genügend schlechte Erfahrungen damit gemacht: Oft stellt sich im Nachhinein heraus, dass die in statischen Variablen oder Funktionen gefasste Funktionalität doch über Objekte spezialisiert werden müsste. Aus Sicht der Sicherheit sind diese Konstrukte noch viel fataler: Der bloße Besitz eines Klassenobjektes erlaubt deren Benutzung. Es ist nicht einmal der Besitz eines Objektes nötig. Die Erreichbarkeit der Klassen innerhalb eines Programms wird dadurch enorm gesteigert. Mit anderen Worten: Die Isolation zwischen Modulen lässt nach. Wer konstruiert schon Bibliotheken unter dem Gesichtspunkt der Sicherheitsproblematik darin enthaltener Klassen? Meist wird sehr großzügig externer Code importiert. Änderbare statische Variablen erschweren die Analyse von Effekten innerhalb eines Programms gewaltig, da praktisch von überall her Seiteneffekte möglich sind.

10.3.4 Klassen, Objekte und Referenzen

Bleiben wir noch ein wenig auf der Ebene der Mikro-Architektur und fragen danach, wie das Basisprinzip der Objektorientierung – die Kapselung – mit sicherer Software zusammenhängt. Tatsächlich ist es das Prinzip der Kapselung und nicht die Vererbung oder Polymorphismus, das unter den OO-Konzepten die größte Bedeutung für die Sicherheit besitzt. Meist wird Kapselung mit besserer Wartbar-

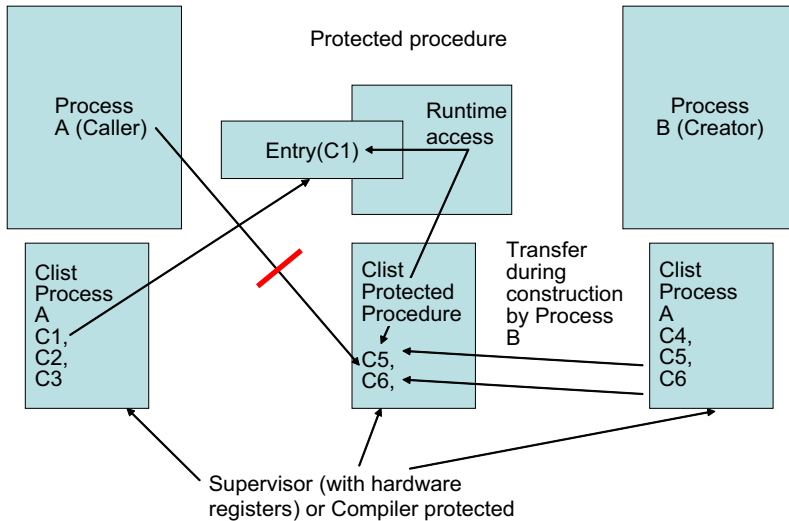


Abb. 10.5 Erweiterung des Supervisors

keit begründet. Unter dem Gesichtspunkt der Sicherheit ist die Bedeutung der Kapselung jedoch noch viel größer: Sie erlaubt einen kontrollierten Zugriff auf Autorität und entspricht damit im Prinzip der Erweiterung eines Supervisors durch Custom Code [Levy]. Ein Beispiel aus dem Bereich der Capability-Systeme soll dies verdeutlichen (Abb. 10.5).

In diesem Beispiel ist es die Hardware, die den Zugriff auf die Capabilities reguliert, aber das Beispiel des Singularity-Betriebssystems hat gezeigt, dass eine typsichere Sprache dazu ebenfalls fähig ist. Was steht im Kern dieses Beispiels? Es zeigt, dass Kapselung es erlaubt, Funktionen einem Aufrufer zur Verfügung zu stellen, die ihrerseits Zugriff auf Referenzen benötigen, ohne dass jedoch diese Referenzen direkt dem Zugriff des Aufrufers ausgesetzt wären. Da eine Referenz die Voraussetzung von Erreichbarkeit ist und Erreichbarkeit mit (potenzieller) Autorität einhergeht, erlaubt die Kapselung eine Verringerung der Autorität durch die Objektmethoden. Entscheidend an diesem Beispiel ist darüber hinaus, dass Konstrukteur und Aufrufer des Objektes verschiedenen Sicherheitszonen oder Kontexten angehören können. Die Typsicherheit der Sprache verhindert dann, dass unerlaubt auf die eingebetteten Referenzen zugegriffen werden kann. Oft wird in OO-Programmen davon ausgegangen, dass Konstrukteur und Aufrufer eines Objektes dieselbe Instanz sind – ausgedrückt durch einen „public constructor“. Das setzt aber voraus, dass der Konstrukteur auch alle Referenzen, die das Objekt benötigt, besitzt und weitergeben kann – oder aber die Klasse besitzt Möglichkeiten, diese Referenzen durch Zugriffe auf Directories oder statische Methoden anderer Klassen zu erhalten. Beides sind aber kritische Eigenschaften in Bezug auf die Minimierung von Erreichbarkeit. Durch Kapselung kann beliebiger Code im User-Modus privilegierte Sektionen erzeugen und sie vor Zugriffen schützen, ohne dass ein Supervisor benötigt würde – eine Basisvoraussetzung für

Abb. 10.6 Bruch der Kapselung
beim OO-Design

```
Public Class Foo {
    Private Bar X;
    Private Directory Z;
    Public Foo (..) {}
    Public Bar getBar() { return X;}
    Public void setBar(Bar y) { X = y;}
    Public doX() {};
    Public doY() {};
    Private void getEnvironment() {
        Z = Context.getEnv();
        // traverse directory....
    }
}
```

modulare Sicherheitszonen innerhalb von Software, die ohne Referenzmonitor funktionieren.

Wir haben in diesem Beispiel Begriffe wie „Referenz“ oder „Interface“ verwendet, ohne sie in ihrer Bedeutung für sichere Software genauer zu untersuchen. Dies holen wir jetzt nach und werden dabei einige Denkweisen bei der Programmierung und beim Design von Software hinterfragen. Wenn Kapselung das wichtigste Prinzip der Objektorientierung darstellt – wieso fangen dann sehr viele Einführungen in das Konzept wie in Abb. 10.6 an?

`get`- und `set`-Methoden scheinen unabdingbar mit dem Konzept von Objekten verbunden zu sein. Dahinter steckt schlicht die immer noch weit verbreitete Denkweise, dass Objekte erweiterte Strukturen sind – d. h. im Endeffekt Datencontainer. Diese Sichtweise wird durch eine OO-Methodik gestützt, die den Fokus auf Strukturentwicklung (Klassen) hat und erst langsam das Verhalten als wichtig erkennt (z. B. UML 2.0).

Ein Blick auf die Sicherheitseigenschaften dieser Klasse offenbart große Probleme: Die Klasse vergibt über `get`- und `set`-Methoden direkten Zugriff auf Referenzen und damit Autorität. Der Aufrufer erhöht die Erreichbarkeit von Ressourcen oder Funktionen, indem er die `get`-Methode verwendet. Sehr viele OO-Programmierer sehen weiterhin kein Problem hinter der Rückgabereferenz auf das interne Objekt statt einer Referenz auf eine Kopie des Objektes. Die Per-Value bzw. Per-Reference-Semantik ist in Bezug auf die Sicherheit von Software von dramatischer Bedeutung. Durch Rückgabe der Referenz wird es dem Objekt unmöglich, die Manipulation des internen Objektes zu kontrollieren, da in den meisten Fällen nicht zwischen Referenz (Erreichbarkeit) und Modifizierbarkeit unterschieden wird. Beides sind jedoch in Wirklichkeit völlig orthogonale Konzepte, die nur durch die sehr eingeschränkten Möglichkeiten der gängigen Programmiersprachen in Bezug auf getrennte Environments vermischt werden.

Das Klassendesign offenbart weitere Probleme: Über eine statische Variable erhält das Objekt Zugriff auf eine potenziell große Anzahl von weiteren Objekten und kann seine Erweiterbarkeit dadurch erheblich steigern. Es wäre übrigens auch keine Verbesserung, die Referenz auf das Directory per Konstruktor zu übergeben – es sei denn, es wäre ein auf dieses Objekt zugeschnittenes, verringertes Directory. Andernfalls erhält das Objekt erhebliche potenzielle Autorität.

Das alles mag für Enterprise-Entwickler recht spitzfindig klingen – der Code ist ja als „gut“ angenommen. Für die Durchführung von Sicherheitsanalysen stellt sich durch solches Design jedoch das Problem, dass keine sicheren Zonen innerhalb der Software festgestellt werden können – die Erreichbarkeit von Code ist potenziell sehr groß. Damit muss theoretisch jede Zeile des Codes untersucht werden, was jedoch bei der Größe der heutigen Programme schlicht unmöglich ist. Auch Werkzeuge zur Programmanalyse tun sich bei der Feststellung der potenziellen Autorität eines Programms sehr schwer, wenn Referenzen einfach weitergegeben werden.

Wovon hängt die Korrektheit der Beispielklasse ab? Normalerweise sollte sie nur von den Komponenten abhängen, die diese Klasse ihrerseits verwendet. Miller nennt dies „defensive correctness“:

„A program P is defensively correct, if it continues to provide correct behavior to well behaved clients despite arbitrary behavior on the part of its other clients“. [Mil, S. 32]

Miller bezieht sich an dieser Stelle zwar vor allem auf das Problem des Multi-Threaded-Zugriffs auf Objekte, jedoch gilt das Gleich für die Verwendung interner Referenzen: Zwei Aufrufer können eine Referenz unterschiedlich manipulieren und sich damit gegenseitig beeinflussen. Miller weist darauf hin, dass den Ressourcen eines Nutzers innerhalb eines der populären Betriebssysteme die gleiche Rolle wie geteilten Referenzen zukommt: Programme eines Nutzers können sich gegenseitig durch Veränderung von Files des Nutzers (auf die sie alle Zugriff haben) manipulieren.

Das oben geschilderte Beispiel besitzt noch eine Schwäche, die sich jedoch diesmal direkt aus dem Konzept der Klasse ergibt: Viele Klassen haben ein recht „fettes“ Interface, d. h. Methoden der unterschiedlichsten Art sind zusammengefasst und zugreifbar durch den Besitz einer Referenz auf ein entsprechendes Objekt. Wir haben aber im Kapitel zu JAVA EE und Enterprise-Applikationen (Kap. 6) gesehen, dass die Mischung von Methoden unterschiedlicher Autorität problematisch ist. JAVA EE erlaubt deshalb die Sicherheitsqualifizierung eines EJB pro Methode, durchgeführt durch einen externen Referenzmonitor, in diesem Fall dem Container. Ohne die Annahme eines Referenzmonitors ist die Mischung von z. B. Read-Only-Funktionen und mutierenden Funktionen problematisch. Wichtig wäre es, ganz leicht und unproblematisch sogenannte *Facets* (unterschiedliche Sichtweisen) auf Objekte erzeugen zu können, die z. B. verhindern, dass bestimmte Methoden zugreifbar werden.

Der Zugriff auf Objekte über Referenzen könnte demnach granularer sein. Wir werden dies gleich im Abschnitt zu funktionalen Prinzipien anhand der Closures

und Higher-Order-Functions nochmals aufnehmen. Etwas später werden wir auch nochmals auf Designfragen sicherer Software eingehen. Zuvor soll jedoch noch gezeigt werden, dass einige der eben besprochenen Prinzipien sehr eng mit den Eigenschaften funktionaler Sprachen verbunden sind.

10.3.5 Funktionaler Ansatz

Wer sich mit sicherer Software beschäftigt, stellt schnell fest, dass funktionale Sprachen einige sehr gute Eigenschaften besitzen, was das sichere Design von Lösungen, aber auch vor allem die Unterstützung von Sicherheitsanalysen betrifft. Allerdings sind in der Ausbildung vieler Softwareentwickler diese Prinzipien gar nicht mehr enthalten und ihre Diskussion scheint zunächst von rein akademischen Interesse – was könnte z. B. an Seiteneffekten falsch sein, wenn doch Sprachen wie C++ und Java darauf beruhen?

Im Einzelnen sind die für sichere Software relevanten Eigenschaften der funktionalen Sprachen die folgenden:

- Keine „Dirty Assignments“ – d. h. Variablen werden einmal initialisiert und dann nicht mehr geändert. Nicht zuletzt im Zusammenhang mit Multi-Threading stellt dies einen großen Vorteil zur Vermeidung von Inkonsistenz und ungewollten Seiteneffekten dar.
- Die Parameterübergabe per Value und nicht per Referenz schafft leichtere Kontrolle der Erreichbarkeit und trennt sie von der Modifizierbarkeit.
- Keine globalen Variablen, damit lässt sich die Erreichbarkeit besser kontrollieren.
- Higher-Order-Functions, um granulare Zugriffsmöglichkeiten durch neue Funktionen zu schaffen.
- Closures, um granulare Zugriffsmöglichkeiten auf eingebettete Autorität zu schaffen sowie um für eine gleich bleibende Semantik der Funktion zu sorgen.

Das Verbot der mehrmaligen Wertezuweisung bedeutet einmal, dass Seiteneffekte vermieden werden können. Zum zweiten hilft es aber auch, sogenannte TOC2TOU (Time-Of-Check To Time-Of-Use) Fehler zu vermeiden, die gerade, wenn Multi-Threading verwendet wird, gerne auftreten (die Browseranalyse von Mozilla ergab mehrere Sicherheitslücken, die daraus resultierten, dass zwar eine Variable geprüft wurde, sie aber nachträglich durch einen anderen Thread wieder verändert wurde und damit die Prüfung hinfällig war). Statische Variablen, die geändert werden können, sind im Multi-

Threading-Fall eine häufige Fehlerquelle. Ross Anderson beschreibt mehrere dieser TOC2TOU-Fehler sowie deren systematische Entdeckung [And, S. 116ff.].

Globale Variablen (dazu gehören auch statische Variablen, auch wenn sie read-only sind) erlauben den Zugriff auf weitere Objekte von beliebigen Stellen aus und stellen somit einen Paradefall für „Ambient Authority“ dar. Globale Variablen verstecken sich gerne hinter statischen Methoden. Globale Variablen eignen sich

auch hervorragend für „Covered Channel Attacks“: Eigentlich isolierte Module tauschen Informationen über eine gemeinsame Kontaktstelle aus.

An dieser Stelle empfiehlt sich ein kleines Gedankenexperiment. Nehmen Sie eine beliebige Stelle im Source Code einer Applikation (z. B. innerhalb einer Methode einer Klasse) und versuchen Sie, möglichst viel Zugriff auf andere Objekte und Ressourcen zu erhalten – auch wenn sie mit dem Zweck ihrer Methode momentan gar nichts zu tun haben. Wenn bösartiger Code in ihr System eingeschleust wurde (sei es durch einen Buffer Overflow oder sei es nur durch ein Plug-In, eine Extension etc.), dann haben Sie genau dieses Szenario. Das gleiche gilt im Fehlerfall ihres Programms. Dadurch bestimmt sich das Gefahrenpotenzial an dieser Stelle des Codes.

Wie kommt man an „Autorität“ bzw. an Rechte? Das System-Call-Interface des Betriebssystems ist die erste Wahl, falls es in Ihrer Sprache direkt zugänglich ist. Buffer-Overflow-Attacken nützen die Tatsache, dass sich mit wenigen Maschinenbefehlen beliebige Betriebssystemfunktionen ergattern lassen (vorausgesetzt, der Nutzer des Programms hat die entsprechenden Rechte), aus. In typischeren Sprachen gilt es, nach globalen, statischen Variablen zu suchen. Viele Sprachen bieten einen sog. „Global Scope“, der Zugriff auf wichtige Ressourcen wie Logging bietet. Der Global Scope ist überall verfügbar. Ganz besonders praktisch sind Zugriffe auf Registries mit Objekten: RMI-Registry, JNDI-Trees etc. bieten eine große Zahl von Objekten für den Zugriff.

An nächster Stelle kommt die Untersuchung der momentan verfügbaren Referenzen: An welche weiteren Referenzen kann man über sie gelangen? Die „besten“ Beispiele sind hier Tree Interfaces, die es erlauben, von einem File ausgehend durch das ganze File-System zu navigieren, und Document Object Model (DOM) Instanzen, die über ihre Parent/Child-Interfaces global navigierbar sind.

Wenn Zugriffe auf solche mächtigen Quellen von Referenzen möglich sind, dann wird die Sicherheitsanalyse eines Moduls schlicht unmöglich. Es lässt sich dann nicht mehr in ausreichend kurzer Zeit feststellen, welche potenziellen Nebeneffekte dieses Modul haben kann, da es theoretisch Zugriff auf alle Ressourcen hat. Wir können dadurch im Design keine Isolation und in der Sicherheitsanalyse kein Endergebnis erreichen.

Ein wichtiger Punkt ist die granulare Zugriffskontrolle durch Funktionen, hier speziell sogenannte *Higher-Order-Functions*. Eine Sprache besitzt Higher-Order-Functions, wenn Funktionen andere Funktionen erzeugen und auch zurückgeben können. Abbildung 10.7 zeigt ein Beispiel einer solchen Higher-Order-Function.

Der Konstruktor `makeMultiplier()` liefert eine Funktion zurück (ist also selbst eine Higher-Order-Function), in diesem Fall eine Funktion, die ihr Eingabeargument mit zwei multipliziert. In der OO-Sprache zeigt eine Higher-Order-Function gewisse Ähnlichkeit mit dem Command Pattern: Sie stellt dem Aufrufer eine Methode zur Verfügung, die ihrerseits aufgerufen werden kann. Allerdings ist z. B. die Verwendung des Command Patterns in Java bei weitem nicht so elegant wie eine Higher-Order-Function etwa in Scheme.

Die Mächtigkeit des Konzepts der Higher-Order-Functions zeigt sich u. a. im Map/Reduce/Merge-Pattern, das Google sehr erfolgreich zur Auswertung seines Datenbestands einsetzt [DG].

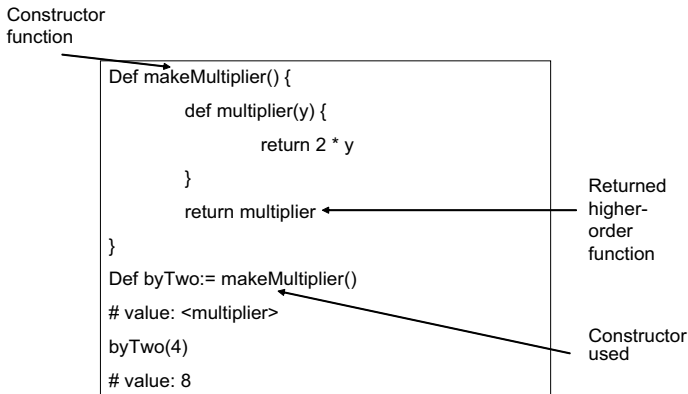


Abb. 10.7 Beispiel für den Einsatz von Higher-Order-Functions

10.3.6 Closures

Sicherheitstechnisch besonders interessant ist die Kombination von Higher-Order-Functions mit einem Zustand, genannt *Closure*. Was ist eine Closure?

Eine Closure assoziiert einen Codeblock innerhalb einer Methode bei der Initialisierung mit seiner momentanen Umgebung zur Zeit der Initialisierung. Wird der Codeblock nun zu einem späteren Zeitpunkt ausgeführt, so greifen alle Referenzen von innerhalb des Codeblocks auf lokale Variablen der umgebenden Methode auf die Instanzen der lokalen Variablen in der assoziierten Umgebung zu. Dadurch erhält der Codeblock die Qualität des mathematischen Funktionsbegriffes (d. h. er ist beliebig oft mit immer dem gleichen Ergebnis aufrufbar), und wir können zu beliebigen Zeiten bestimmte Effekte durch Aufruf einer Funktion erzielen, ohne dabei Angst vor Seiteneffekten haben zu müssen. Der Code wird damit in seiner Wirkung abschätzbar.

In Abb. 10.8 wird eine Funktion `byFour` mit dem Parameter 4 initialisiert. Dieser Parameter gehört zu der Umgebung, innerhalb derer die Funktion bei späteren Aufrufen immer wieder ausgeführt wird, d. h. man wird immer wieder einer Funktion erhalten, die den Eingabeparameter mit 4 multipliziert.

Nicht alle Sprachen verfügen über Closures. Java z. B. kommt mit den Inner Classes dem Prinzip der Closure relativ nahe, ohne jedoch beliebige Environments bei der Initialisierung einer Inner Class mitgeben zu können. Alle Initialisierungsvariablen müssen in Java „final“ deklariert sein (es kann sich ja um Variablen auf dem Stack handeln, die dann beim Aufruf der Methode der Inner Class längst nicht mehr vorhanden wären). Closures und Higher-Order-Functions lassen sich jedoch in Java simulieren, und es besteht die Möglichkeit, dass Closures in Java Version 1.7 enthalten sein werden. Das Beispiel der privilegierten Funktionen in JAAS hat gezeigt, wie wichtig das Konzept einer kopierten Ausführungsumgebung letztlich ist: Der „privileged“-Marker gilt nur innerhalb der Funktion und das Lexical Scoping sorgt dafür, dass alle Referenzen auf so einen Marker inner-

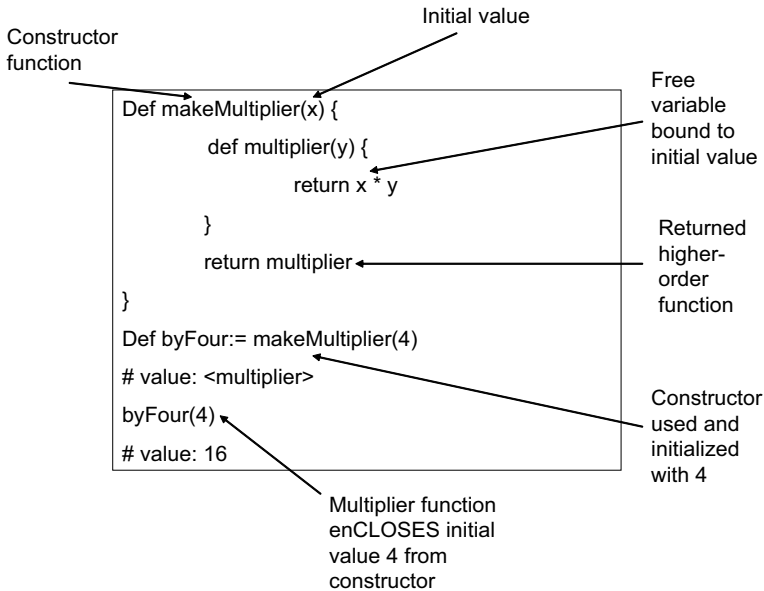


Abb. 10.8 Beispiel für eine Closure

halb der Funktion aufgelöst werden. Am Ende der Funktion ist auch der Marker ungültig.

Wir werden bei den Bausteinen sicherer Software einige Anwendungen von Closures und Higher-Order-Functions kennen lernen. Sicher lassen sich einige dieser Anwendungen in OO-Sprachen auch ohne diese Mittel simulieren, jedoch bei weitem nicht so elegant und vor allem nicht so leicht benutzbar – und die Usability von Sicherheitsmaßnahmen ist auch für die Programmierung entscheidend.

Der Schritt von Closures zu Objekten ist sehr einfach und besteht im Wesentlichen aus der Implementation einer Dispatch-Funktion, die den übergebenen Methodennamen prüft und an interne Funktionen verzweigt (Abb. 10.9). Die folgenden Beispiele sind [Kisel] entnommen.

Die „Object“-Charakteristik bringt lediglich einen normalerweise versteckten Dispatch über die Namen der eingebetteten Funktionen dazu. Dieser Dispatch ist hier sichtbar gemacht, wird aber natürlich sonst im Rahmen der Runtime-Umgebung einer Sprache zur Verfügung gestellt. Der Aufruf „object.foo(arg)“ ist also nur syntaktischer Zucker über den wirklichen Aufruf `object.dispatch („foo“, arg)`.

Ohne die in der Closure eingebetteten Werte würde es sich bei Make-Point-2D schlicht um eine Higher-Order-Funktion mit eingebetteten Funktionen handeln (die auch an unterschiedliche Empfänger verteilt werden können). So jedoch greifen diese Funktionen (und nur sie!) beim Aufruf auf die ursprünglich zum Zeitpunkt der Konstruktion übergebenen Parameter zu (Abb. 10.10). Wir werden diese Eigenschaft in den Bausteinen sicherer Software (Abschn. 10.3.9) ausnutzen.

Abb 10.9 Implementation eines Objekt-Dispatch-Mechanismus

Implementation of a simple dispatcher function (selector) which checks function name and forwards request to the proper function via “apply”

```
(define (make-point-2D x y)
  (define (get-x) x)
  (define (get-y) y)
  (define (set-x! new-x) (set! x new-x))
  (define (set-y! new-y) (set! y new-y))
  (lambda (selector . args) ; a dispatcher
    (case selector
      ((get-x) (apply get-x args))
      ((get-y) (apply get-y args))
      ((set-x!) (apply set-x! args))
      ((set-y!) (apply set-y! args))
      (else (error "don't understand" selector)))))
```

Taken from Oleg Kiselyov [Kisel]

Abb. 10.10 Beim Aufruf sind die Werte zum Zeitpunkt der Konstruktion erhalten geblieben

```
(define a-point-2D (make-point-2D 5 6))
(a-point-2D 'get-x) => 5
(a-point-2D 'set-y! 10)
(a-point-2D 'get-y) => 10
(define b-point-2D (make-point-2D 5 10))
```

; a-point-2D and b-point-2D currently have the same state

```
(equal? (list (a-point-2D 'get-x) (a-point-2D 'get-y))
  (list (b-point-2D 'get-x) (b-point-2D 'get-y))) ==> #t
```

```
; but they are different objects
(eq? a-point-2D b-point-2D) ==> #f
```

Taken from Oleg Kiselyov [Kisel]

10.3.7 Tainting

Beispiele für den Einsatz von Tainting wurden bereits im Kapitel zu den Attacks erwähnt. So kann mithilfe der Markierung von Objekten oder Daten erreicht werden, dass sie nicht aus Versehen an ungeeignete Funktionen gegeben werden. So sollte z. B. kein nicht-encodierter String als Output einer HTML-Page erscheinen. Und nicht-validierte Inputvariablen sollten nicht naiven Interpretationsfunktionen übergeben werden.

Aber nicht nur Daten sind für das Tainting geeignet. R. Schneider, der Autor der bekannten Mobmap Extension (www.mobmap.de) für das Multi-Player Online Game World of Warcraft (WoW), hat Tainting im Zusammenhang mit der Sicherheitsarchitektur von WoW für clientseitige Erweiterungen untersucht [Schn].

Die Grundproblematik für Blizzard, den Hersteller und Betreiber von WoW, liegt laut Schneider darin, dass einerseits die Erweiterbarkeit des Spiels durch

User eine Kerneigenschaft ist, andererseits die Erweiterungen zu einer Automatisierung des „User-Interfaces“ führen können, die wiederum den Spielspaß bzw. sogar die Spielgestaltung durch Blizzard negativ beeinflussen. Außerdem geschieht die Interaktion zwischen Add-On – so werden bei WoW Extensions genannt – und dem Basissystem des Clients über API-Aufrufe vom Client zum System sowie zurück über Callback-Events. Dies macht es schwierig, den Callgraphen eindeutig abzusichern und zu verhindern, dass unsichere Aufrufervariablen in sicheren Funktionen landen.

Hier zeigt sich genau die Stärke von Tainting: Da Variablen und Funktionen des aufrufenden Clients stets als „getainted“ betrachtet werden und diese Eigenschaft nicht verlieren, kann das Basissystem jederzeit feststellen, ob es sich um eine externe Variable oder Funktion handelt.

Schneider beschreibt jedoch noch eine weitere Besonderheit: Ähnlich der Java 2 Security wird der gesamte Callgraph durch einen Stack-Walk bei einer Überprüfung untersucht. Das Runtime-System kann daher überprüfen, ob jemals getaintete Variablen oder Funktionen letztlich an einem Aufruf beteiligt waren. Ähnlich wie in Java gibt es dann auch Möglichkeiten für das Runtime-System, sich in privilegierten Funktionen über das Tainting hinweg zu setzen bzw. in Form von Proxies den Client-Add-Ons nur eingeschränkten Zugriff auf privilegierte (meist automatisierbare) Funktionen zu gestatten [Schn].

Wir werden auf die Vorschläge von Schneider zur Verwendung von Tainting noch im Rahmen der Browser-Security zurückkommen. Auch dort gibt es das Problem, fremde Daten und Funktionen innerhalb eines Runtime-Systems auszuführen und zu kontrollieren. In der Vergangenheit hat das z. B. bei Firefox regelmäßig zu Irrtümern über die Herkunft einer Javascript-Funktion geführt, die anschließend fälschlicherweise mit den erweiterten Privilegien des Chrome-Runtime-Systems ausgeführt wurde.

Wir können an dieser Stelle festhalten, dass das Tainting immer dann günstig ist, wenn zwischen zwei unterschiedlichen Vertrauenszonen Dinge ausgetauscht werden und vom Runtime der jeweils anderen Zone ausgeführt bzw. bearbeitet werden sollen. Die Tatsache der unterschiedlichen Herkunft darf dabei einerseits nicht verloren gehen und andererseits muss die Überprüfung systematisch – egal durch welchen Callgraph herbeigeführt – durchgeführt werden können. Das heißt, eine Umgehung der Prüfung muss ausgeschlossen sein. Und ganz streng genommen stellt bereits eine starke Typisierung eine Form des Tainting dar, denn Interfaces dürfen dann nur die jeweils definierten Typen annehmen.

10.3.8 Object Capabilities

Wir kommen nun zu einem weiteren zentralen Konzept für sichere Software, den *Object Capabilities*. Wir haben bereits einige Grundkonzepte der Zugriffskontrolle kennen gelernt, wie z. B. Referenzmonitore, Isolation durch Memory Management, Namespaces oder Tainting. Im Gegensatz zu den meisten der genannten haben Object Capabilities jedoch die Eigenschaft, dass sie keine externe Entschei-

dung über oder Erlaubnis oder Ablehnung des Zugriffs mehr benötigen: Der Besitz einer Object Capability entspricht dem Recht, sie zu verwenden. Wir werden sehen, wie diese Eigenschaft gerade auch die Sicherheitsanalyse einfacher macht, da im Code bereits ablesbar ist, welche Ressourcen oder Funktionen überhaupt erreichbar sind.

Object Capabilities sind nicht fälschbare Referenzen auf Objekte. In typischen Sprachen entspricht eine lokale Object Capability einer Referenz. Entscheidend für eine auf Object Capabilities beruhende Architektur ist, dass es keine Möglichkeit für Code gibt, außerhalb einer Object Capability an Rechte, bzw. besser gesagt Zugriffsmöglichkeiten, zu gelangen. Das Gegenstück heißt, wie bereits mehrfach erwähnt, Ambient Authority – also frei verfügbare Rechte, die Code zu jeder Zeit verwenden kann. Durch die Kontrolle der Weitergabe von Capabilities kann die Erreichbarkeit von Ressourcen feingranular kontrolliert werden.

Object Capabilities lassen sich sehr schön mithilfe sogenannter Granovetter-Diagramme zeigen: Wenn Bob keine Referenz (also keine Object Capability) auf Carol bekommt, hat er keine Möglichkeit, auf Carol in irgendeiner Weise zuzugreifen (Abb. 10.11).

Miller et al. drücken dieses fundamentale Misstrauen der beteiligten Objekte so aus:

“Attempts to formalize the semantics of computation, including secure computation, have failed to capture the core intuitions of computer security practitioners. Fundamental to these intuitions is the notion of mutually suspicious, separately interested agents, interacting within a framework of rules, under constraints of partial knowledge, each in order to pursue their own interests. The formal tools for capturing such intuitions are to be found in non-zero-sum, partial-information game theory. The Granovetter Diagram expresses the core game-rule governing secure capability-based distributed multi-agent computation, viewed as a vast multi-player game.” [MMF, S. 2ff.]

Object Capabilities sind also feingranulare Rechte auf beliebige Objekte – z. B. Teile eines Browserfensters oder Teile eines Dokuments. Sie können jederzeit

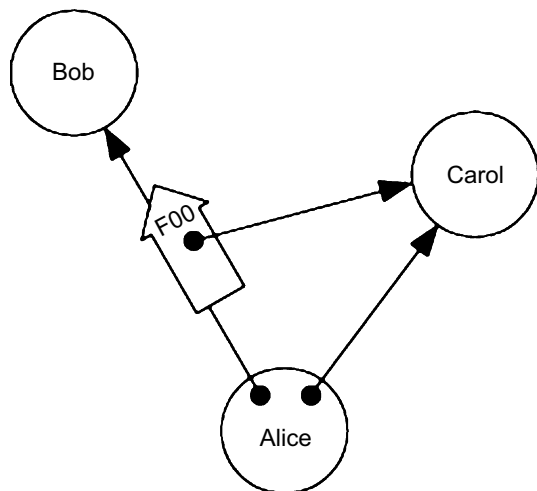


Abb. 10.11 Granovetter-Diagramm, angewendet auf Objektkollaboration

delegiert, d. h. weitergegeben werden, aber auch – unter Zuhilfenahme von Sicherheitsbausteinen – zurückgenommen werden. Sie können auch verweigert werden, was dem Verbot von Zugriffen gleichkommt. Sie sind damit eindeutig dynamisch in ihren Möglichkeiten, Rechte zu propagieren bzw. wieder zurückzunehmen. Dieses Fließen von Zugriffsmöglichkeiten in Verbindung mit der Instanzbezogenheit von Capabilities ist somit auch der größte Unterschied im Vergleich zu statischen ACL-Systemen die – da statisch definiert – meist Zugriffe auf ganze Gruppen von Objekten gestatten müssen (z. B. durch Wildcards), anstatt je nach Notwendigkeit nur den Zugriff auf einzelne Objekte zu gestatten.

Mithilfe von Object Capabilities lässt sich das POLA-Prinzip sehr gut implementieren: Ein Softwaremodul fängt quasi rechtlos an, wenn es instanziiert wird. Ohne Object Capabilities hat es keine Zugriffsrechte auf Ressourcen. Das Prinzip der Isolation (Confinement) ist damit hervorragend durchgesetzt. Jetzt kann dieses Modul bestimmte Object Capabilities erhalten – z. B. durch Aktionen eines Nutzers (oder aber auch durch eine entsprechende vorherige Konfiguration, sog. *Endowments*). Jetzt verfügt das Modul über genau die Rechte, die es für sein Funktionieren benötigt. Weitere Rechte kann es zur Laufzeit erhalten.

Das Objekt-Capability-Prinzip entspricht in vielerlei Hinsicht guter Praxis in der OO-Entwicklung, nämlich der Kapselung von Daten und Funktionen. Allerdings operiert es eher auf dem Level einer Komponente. Die Schwierigkeit beim Design auf Basis von Object Capabilities liegt darin, darauf zu achten, ob durch Referenzen Rechte vermittelt werden oder nicht. Gefährlich sind Objekte oder Komponenten mit sogenannten *Convenience Interfaces*: Das sind Methoden, die aus Bequemlichkeitsgründen bestimmte Referenzen zurückgeben, die mit dem eigentlichen Zweck des Objekts nichts zu tun haben. Genauso gefährlich sind Navigations-Interfaces, die es erlauben, z. B. über Tree-Walk-Methoden an weitere Referenzen, sprich Rechte, zu gelangen.

Exceptions stellen ein interessantes Problem im Kontext von Referenzen dar, die als Capabilities verstanden werden: Sie können eingebettete Referenzen vom Ort des Problems potenziell bis zum obersten Level der Applikation transportieren und dadurch Autorität vermitteln. Beim Design von Exceptions in Object-Capability-Systemen muss deshalb sichergestellt sein, dass keine unerwünschten Referenzen „entkommen“ können (für alle Entwickler, die mittlerweile im Minenfeld der unterschiedlichen Exception-Typen unterzugehen drohen, hier eine Erinnerung an das ursprüngliche Konzept von Exceptions als eine Realisierung des Prinzips „positiver“ Programmierung: Vom „Gut-Fall“ auszugehen und nur im Fehlerfall reagieren zu müssen, ist gleichzeitig performant und sicher).

Von entscheidender Bedeutung für Object Capability-basierte Architekturen ist die Frage, woher Rechte, sprich Autorität, kommen kann. Die möglichen Quellen von Autorität sind:

- Bei der Konstruktion eines Objektes an das Objekt übergeben zu werden.
- Durch Interaktion, d. h. Aufruf einer Methode mit Übergabe einer Referenz an ein Objekt.
- Durch Verwendung eines Konstruktors, der als Rückgabewert die Referenz auf das erzeugte Objekt hat.

Der letzte Fall ist speziell: Eigentlich ist der Besitz einer Konstruktor-Funktion gleichbedeutend mit der Autorität, ein solches Objekt zu erzeugen. Die Rückgabe der Referenz an den Aufrufer ist somit selbstverständlich. Weniger klar ist, dass öffentliche Konstruktoren an sich bereits ein Problem darstellen können, vor allem, wenn sie aus beliebigen anderen Modulen aufgerufen werden können. Gleiches gilt natürlich für Referenzen in statischen, globalen Variablen. Konsequenterweise, stellt sich dann auch die Frage, ob nicht z. B. auch die Verwendung von „synchronized“, also von einem Zugriffsmonitor, bereits der Zugriffskontrolle unterliegen sollte. Weiterhin stellt sich die Frage, welche Rechte ein Erzeuger eines Objektes potenziell an die erzeugten Objekte weitergeben kann oder soll. Hier wird deutlich, dass das Verhalten von Objekten bei der Propagation eine entscheidende Bedeutung für die Sicherheit von Modulen besitzt. Ohne die Möglichkeit, dieses Verhalten genauer zu spezifizieren, müsste man von der maximalen Weitergabe von Rechten ausgehen. Der Abschnitt weiter unten zu Sicherheitsbausteinen (Abschn. 10.1.22) greift diese Frage wieder auf, die anschließend in Kap. 14 unter dem Gesichtspunkt der Propagation von Autorität weiter vertieft wird.

Die Aufgabe des Supervisors ist hingegen einfach: Er muss verhindern, dass Objekte Rechte bzw. Autorität außerhalb der genannten Möglichkeiten erlangen können. Diese Aufgabe ist natürlich relativ einfach.

Wir werden das Spannungsfeld von objektorientierten Techniken und Object Capabilities genauer untersuchen, wenn wir die Umwandlung von Java-Bibliotheken in sichere Komponenten (das sogenannte *Taming*) besprechen. David Wagner diskutiert die dabei auftretenden Probleme sehr eindrücklich in einem Google-Tech-Talk [Wagn]. Einen sehr guten Überblick über den Stand der Entwicklung und Forschung zu Capabilities sowie speziell Objekt-Capabilities verschafft Toby Murray in „Analysing Object-Capability Security“ [Murr]. Nun aber wollen wir aus den obigen Konzepten Bausteine entwickeln, die durch ihr Verhalten gewisse Sicherheitseigenschaften bewirken.

10.3.9 Sicherheitsbausteine

“When practicing secure programming, it is important to understand the restrictive influence programmed entities have on the propagation of authority in a program.” [SPIESS]

Programmierte Softwarebausteine mit definierter Sicherheitsfunktion kann man sich wie materialisierte Security Patterns vorstellen. Die Idee dieser Bausteine ist aus mehreren Gründen fundamental. Zunächst gelang es Miller et al. dadurch nachzuweisen, dass bestimmte Defizite von Capabilities bei der Einschränkung von Rechten – speziell die Kontrolle des Widerrufs von Rechten – dadurch ausgeglichen werden konnten. Der zweite Grund liegt darin, dass durch diese Bausteine Sicherheit direkt in der Softwarearchitektur und Implementation verankert wird, anstatt nur durch externe Konfiguration von Rechten. Und drittens stellen diese Sicherheitsbausteine ideale Punkte zur Modellierung der Propagation von Rechten

dar, d. h. die Sicherheitsanalyse einer Software wird durch sie besser möglich. Auf die Möglichkeiten der Modellierung gehen wir im Kapitel zu formalen Methoden (Kap. 14) noch genauer ein.

Nehmen wir an, an Ihrem Rechner befindet sich eine Webcam. Können Sie sicher sein, dass diese Kamera nichts aufnimmt und überträgt, wenn Sie es nicht wollen? Eine Möglichkeit, dies zu entscheiden, wäre, die Kamera durch die Basis-einstellungen Ihres Computers zu kontrollieren, also über Permissions. Eine andere Möglichkeit wäre es, die Kamera abzunehmen und vom Rechner zu trennen. Dies ist eine Kontrolle der Autorität. Beide Möglichkeiten kontrollieren die Kamera über die Topologie (der Permissions oder der Autorität).

Aber wie wäre es, die Kontrolle der Kamera einer Software und deren Verhalten zu überlassen? Die Software realisiert einen Schalter, über den die Kamera ausgeschaltet werden kann. Physisch und in Bezug auf die Permissions des Basis-systems ist die Kamera aktivierbar. Das bedeutet, die Software reguliert die Kamera über ihr Verhalten. Eine Sicherheitsanalyse über die Permissions bzw. über die physische Verbindung der Kamera ist zwar möglich, bringt aber nur konservative Resultate. Wenn wir das Verhalten der steuernden Software kennen, können wir genauere Auskünfte geben. Eine solche Software, die uns eine Sicherheitsgarantie gibt (hier die Garantie, dass die Kamera nur dann arbeitet, wenn wir es wollen), nennen wir einen *Sicherheitsbaustein*.

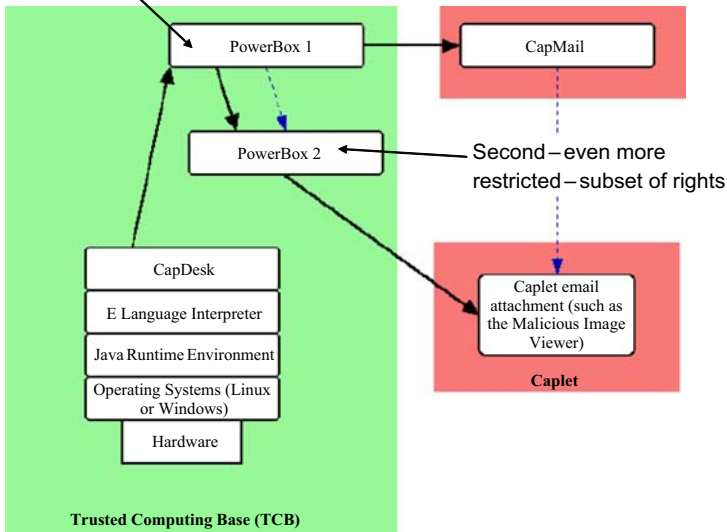
Sicherheitsverantwortlichen in der IT ist beim Gedanken an Sicherheitsbausteine oft unwohl. So beruht z. B. eine DMZ auf dem Prinzip der Trennung durch Topologie. Allerdings kontrollieren auch dort manche Elemente den Zugang eher durch ihr Verhalten, z. B. Application-Proxy-Software. Der Gedanke, Software-bausteine zu bauen, deren Verhalten Sicherheitsgarantien gibt, entstand unter anderem aus dem Bedürfnis, das Prinzip der Capabilities in seinen Anwendungen zu erweitern. Die Gültigkeit von Capabilities kann nicht ohne Software eingeschränkt werden, die die Gültigkeit über ihr Verhalten steuert. Bei Verwendung von Verhalten zur Kontrolle entsteht nun die Möglichkeit einer sehr feingranularen Kontrolle von Autorität, d. h. POLA kann effizient implementiert werden.

Typische Beispiele für solche Bausteine sind (neben einigen anderen) die *Powerbox* und der *Caretaker*, die wir nun in ihrer Funktionalität und Implementation besprechen wollen.

Die Powerbox (Abb. 10.12) ist ein rekursiv einsetzbarer Sicherheitsbaustein, der Capabilities in Form von Referenzen enthält. Software-Module können, wenn sie weitere Ressourcen benötigen, von einer übergeordneten Software (z. B. der Applikation) eine genau auf sie zugeschnittene Powerbox erhalten, die nur Rechte für das jeweilige Modul enthält. Module können ihrerseits neue Powerboxes konstruieren, diese mit Rechten aus ihrer eigenen Powerbox versehen und so untergeordnete Software (d. h. Objekte) mit Autorität versorgen. Diese Organisation der Autoritätsverteilung wird im Abschnitt „Autoritätstrichter“ (Abschn. 10.1.23) genauer untersucht.

Das Granovetter-Diagramm in Abb. 10.13 zeigt eine spezielle Version eines sogenannten Caretakers, ein Baustein, der es ermöglicht, ein eingebettetes Objekt c1 sicherheitstechnisch zu kontrollieren.

First subset of rights



Each application or module gets only as much authority as it needs for the job. This allows safe plug-ins like the renderer for a mail type. From the DARPA browser study (www.combex.org)

Abb. 10.12 Beispiel für die Anwendung von Softwarebausteinen

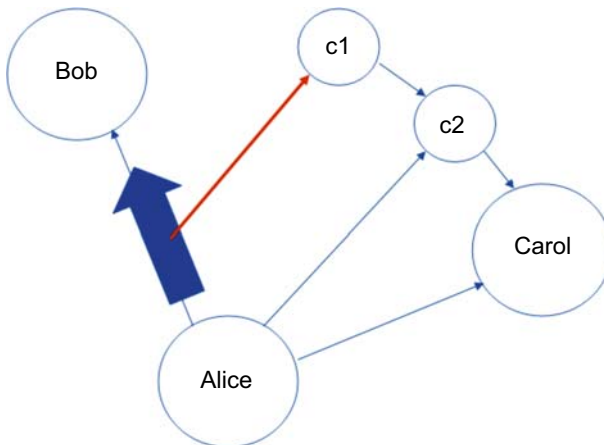


Abb. 10.13 Einsatz eines Caretaker-Sicherheitsbausteins als Proxy

Der folgende Code in der Sprache „E“ ist ein leicht erweitertes Beispiel aus [Mil] und zeigt die Verwendung der oben diskutierten funktionalen Konzepte zur Realisierung eines Caretakers.

Die drei Higher-Order-Functions, die vom Konstruktor erzeugt werden (Abb. 10.14), gehen an unterschiedliche Eigentümer. Der Eigentümer des einge-

```
def makeRevokableAndFilteringForwarderTriple(obj) :any {
    var innerObj := obj

    def forwarder {match [verb, args] {E call(innerObj, verb, args)}}

    def revoker {to revoke() {innerObj := null}}

    def somefunc {to somefunc() {innerObj.somefunc()}}

    [revoker, forwarder, somefunc]
}
```

Abb. 10.14 Beispielcode einer widerrufbaren Capability in E

```
class TRef<T> where T:ITracked {
    public TRef([Claims] T i_obj);
    public T Acquire();
    public void Release([Claims] T newObj);
}
```

Abb. 10.15 TRef-Typ, der Besitz an Referenzen regelt

betteten Objektes behält sich die `revoke`-Funktion selbst vor und kann damit jederzeit den Aufruf dieses Objektes durch andere unterbinden.

Natürlich lässt sich die Sicherheitseigenschaft eines solchen Bausteins auch durch OO-Patterns erreichen, z. B. durch Facets oder Decorators. Singularity verwendet übrigens ebenfalls Sicherheitsbausteine ähnlich dem oben erwähnten Caretaker Pattern, um eine ganz entscheidende Eigenschaft von Objekten zu realisieren: In Singularity gehört ein Objekt zu einem bestimmten Zeitpunkt immer nur einem Prozess. Dies macht Synchronisierung unnötig und erhöht die Performance und die Zuverlässigkeit des Codes ganz enorm. Es setzt jedoch voraus, dass es einen Mechanismus gibt, der Referenzen transportieren und invalidieren kann. Dieser Effekt wird durch den Baustein in Sing# (Abb. 10.15) erzeugt.

Object Capabilities sind wohl die Technik für sichere Software mit dem größten Einfluss auf die gesamte Architektur – dies gilt im guten wie im schlechten Sinne, da sie eventuell ein Re-Factoring erfordern. In [MS] führen Miller und Shapiro den Nachweis, dass Capabilities darüber hinaus die Einbettung in Software-Pattern benötigen, um zentrale Anforderungen wie die Widerrufbarkeit von Rechten erfüllen zu können.

Der Konstruktor in der obigen Abbildung erzeugt drei Funktionen, die an unterschiedliche Clients gegeben werden. Der Empfänger der `forwarder`-Funktion erhält einen Proxy, der jeden Request an das Zielobjekt weiterleitet. Er erhält jedoch keinen Zugriff auf dieses Objekt. Anders gesagt, er benötigt nicht die Autorität für dieses Objekt. Der Erzeuger der Closure wird wahrscheinlich die `revoker`-Funktion selbst behalten und damit auch das Recht auf jederzeitigen Widerruf des Rechts des `forwarders`. Der Empfänger der `somefunc`-Funktion erhält lediglich Zugriff auf einen speziellen Aspekt des Zielobjekts.

Ein anderer interessanter Baustein bewirkt, dass Referenzen in einem ähnlichen Sinn wie physische Dinge übergeben werden können, d. h. eine Referenz, die

weitergegeben wurde, ist danach nicht mehr für den ursprünglichen Inhaber verfügbar. Diese eigentlich sehr einfache Semantik ist in vielen OO-Sprachen nicht direkt realisierbar, sondern muss ebenfalls über verschiedene Pattern erzeugt werden. Wieso ist ein solches Verhalten überhaupt wünschenswert? Durch eine solcherart gestaltete Übergabe von Referenzen kann das Erzielen von Effekten genauer kontrolliert und analysiert werden, da es zu jedem Zeitpunkt nur einen Besitzer der Referenz gibt. Dies vermeidet auch Probleme im Zusammenhang mit Multi-Threading, denn es kann garantiert werden, dass dann immer nur ein Thread auf einem Objekt tätig ist, ohne dass hierfür zu Monitoren oder Semaphoren gegriffen werden muss. Dieses Prinzip der eindeutigen Eigentümerschaft über Referenzen ist auch in Microsofts neuem Betriebssystem Singularity ein Kernelement.

Typisch ist auch die Verwendung einer inkrementellen Initialisierung einer Funktion, bei der weitere Empfänger zwar Parameter setzen können, jedoch keinen Zugriff auf das ursprüngliche Zielobjekt erhalten. So können verschieden mächtige Autoritäten kombiniert und voreinander geschützt werden. Der letzte Empfänger in der Kette kann die Funktion verwenden, indem er sie mit einem Argument versieht, hat jedoch nie das Recht auf die eingebetteten Objekte. Technisch liegt diesem Prinzip das sogenannte *Currying* (benannt nach dem Logiker Haskell Curry) zu Grunde, das aus den funktionalen Sprachen bekannt ist: Dabei wird eine Funktion mit mehreren Argumenten schrittweise in eine Funktion verwandelt, die nur noch ein Argument nimmt. Im einfachsten Fall wird aus einer Funktion $f: X \times Y \rightarrow Z$ mit zwei Argumenten durch Zuweisung eines festen Wertes $x \in X$ eine neue Funktion $f_x: X \times Y \rightarrow Z$, die nur noch ein Argument benötigt (vgl. dazu auch unsere Diskussion der Higher-Order-Functions oben). Im Rahmen des Lambda-Kalküls, in dem jede Funktion immer nur ein Argument besitzen kann, ist das Currying zwingend erforderlich, um auch Funktionen mit mehreren Argumenten darstellen zu können. Für uns ist hier die Tatsache entscheidend, dass die Verwender der „Zwischenfunktionen“ verschiedenen Sicherheitskontexten angehören können. In Sprachen ohne Closures und Higher-Order-Functions ist diese Funktionalität durch Inner Classes oder durch das Proxy Pattern in Verbindung mit dem Command Pattern zu simulieren, was relativ mühsam ist.

Aus den gerade diskutierten Sicherheitsbausteinen ergibt sich jedoch ein Problem für die Sicherheitsanalyse: Sie muss erkennen können, dass eine Referenz durch einen Wrapper geschützt ist. Je nach Sprache und Implementierung kann der Wrapper vom gleichen Typ wie die Referenz sein, und damit ist nicht sofort sichtbar, dass die originale Referenz eingeschränkt wird. Damit erhebt man Kombinationen aus Software-Patterns und Sicherheitsprinzipien in den Status von Sicherheitsbausteinen, die bei der Analyse einer Software in Bezug auf ihre Sicherheit eine zentrale Rolle spielen. Das ist sicher ein ungewohnter Gedanke – sowohl für die Entwickler als auch für die Verantwortlichen der IT-Security. Andererseits stellen auch Proxies in Firewalls nichts anderes als Sicherheitsbausteine in Softwareform dar.

Eine ausgezeichnete Beschreibung mehrerer Sicherheitsbausteine findet sich in [Scheff] sowie im Wiki zur Programmiersprache E [Ewiki]). Dort werden diese Patterns im Bereich von Virtual Worlds eingesetzt, um z. B. Zugriffsentscheidun-

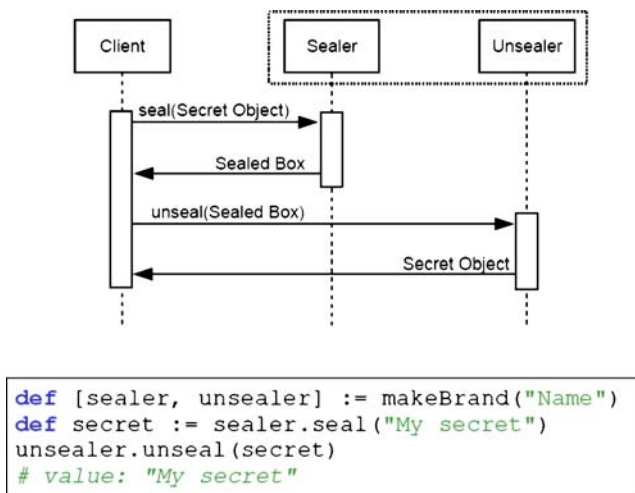


Abb. 10.16 Funktion des Sealer/Unsealer-Sicherheitsbausteins

gen aufgrund spatialer Eigenschaften wie Nähe oder Containment treffen zu können. Außerdem wird die feine Granularität der Delegation von Capability-basierten Rechten ausgenutzt. Ein Beispiel dafür ist das unten dargestellte Sealer/Unsealer Pattern. Es gestattet die Weitergabe des eingeschlossenen Inhalts an beliebige, nicht-vertrauenswürdige Instanzen. Entpacken kann den Inhalt jedoch nur derjenige, der die Unsealer-Funktion erhalten hat. Für andere bleibt der Inhalt schlicht unerreichbar.

Die Autoren zeigen ein weiteres Beispiel, Rights Amplification genannt, bei dem sie mithilfe des Sealer/Unsealer Patterns spezielle, eingeschränkte Fassaden von Objekten unterschiedlichen Gruppen zur Verfügung stellen können. In Kombination mit dem Revoker Pattern lässt sich die dergestalt übergebene Autorität jederzeit wieder zurücknehmen. Abbildung 10.16 zeigt den Ablauf des Ver- und Entsiegelns.

Im nächsten Abschnitt wollen wir in Bezug auf die Granularität der Softwarekonzepte einen Schritt hin zu größeren Architekturen wie z. B. der einer ganzen Applikation oder einer ganzen Maschine machen.

10.3.10 Makro-Architektur

Wie der Name schon sagt, stellt das Principle Of Least Authority (POLA) den Versuch dar, Schaden durch Begrenzung der zur Verfügung stehenden Autorität auf das unbedingt Notwendige zu minimieren. Dazu gehört auch, dass sich Code nicht einfach selber mehr Autorität verschaffen kann, sondern auf das Fließen von Autorität in Form von Referenzen angewiesen ist. Jetzt bleibt das Problem zu lösen, woher denn letztlich Autorität kommen soll und wie die Softwarearchitektur

damit umgeht. Wir fragen also nach den Quellen von Autorität und ihrer Verteilungsstruktur.

Die bekannteste Quelle von Autorität bei einem Rechner ist natürlich der Nutzer. In den meisten Systemen hängt an der Identität des Nutzers das Recht, auf bestimmte Ressourcen zugreifen zu können. Diese Rechte sind statisch und stehen in keinem Zusammenhang mit konkreten Aktionen des Nutzers bzw. mit Applikationen, die der Nutzer startet. Aber auch Programme können über ihre Konfiguration Autorität erhalten und speichern. Beispielsweise liegt es in der Natur von Schlüsseln, dass sie exzellent weitergegeben werden können und somit auch zur Übertragung von Autorität dienen können.

Im Kapitel zur Absicherung von Infrastruktur diene ein „Trichter“ zur Verringerung der Angriffsoberfläche als zentrales Element der Sicherheitsarchitektur. Eine ähnliche Architektur ist möglich, um Autorität in Form von Referenzen sehr feingranular an die Stellen zu transportieren, wo sie benötigt werden. Um die Granularität zu gewährleisten, muss beim Transport eine Reduktion von Autorität stattfinden, sonst bleiben wir beim Konzept der Ambient Authority, die allen Teilen einer Software die gleichen Nutzerrechte zur Verfügung stellt.

In einem ersten Schritt müssen dazu die globalen Rechte eines Nutzers auf die Rechte reduziert werden, die eine vom Nutzer gestartete Applikation für die momentanen Aktionen tatsächlich benötigt. Hier zeigt sich, dass die meisten Applikationen tatsächlich sehr viel weniger Rechte benötigen, als ihnen zur Verfügung gestellt werden, wie z. B. im Kap. 4 am Beispiel eines Copy-Programms gezeigt wurde.

Entscheidend für die Reduktion ist, dass der Nutzer der Applikation nur genau die benötigten Rechte zuteilt, und dies am Besten in Form von Referenzen und nicht in Form von Namen. Die Umsetzung von Namen in Capabilities bzw. Referenzen ist eine sehr kritische Operation und viele Attacken z. B. auf Browser nutzen hier Möglichkeiten, den Benutzer zu verwirren oder die Software zu falschen Zugriffen zu veranlassen.

Daher empfiehlt es sich, diese kritische Operation aus den Applikationen zu entfernen und in eine Komponente zu verlagern, deren Sicherheit prüfbar ist. Eine solche Komponente kann ein sicherer Desktop sein, wie er z. B. von der Firma Combex entwickelt wurde. Mehr dazu im folgenden Abschnitt zu Designation und Autorität.

Wenn eine Applikation die nötigen Capabilities erhalten hat, so stellt sich die Frage, ob allen Modulen innerhalb der Applikation alle diese Rechte zur Verfügung stehen sollen. Dies wäre wiederum ein Verstoß gegen POLA, denn nicht alle Module brauchen Zugriff auf alle Rechte. Hier empfiehlt sich die Anwendung der Inversion-of-Control-Architektur, um den Ausführungskontext der Module zu begrenzen. Die Module wiederum geben an einzelne Objekte nur die benötigten Capabilities weiter.

Es bleibt nun noch die Frage zu beantworten, wie eine Applikation weitere benötigte Capabilities erhält. Sie kann dazu entweder über den sicheren Desktop den Nutzer auffordern, eine Ressource zur Verfügung zu stellen (beispielsweise per Drag and Drop) oder die Applikation kann einen Sicherheitsbaustein verwenden –

Autorität

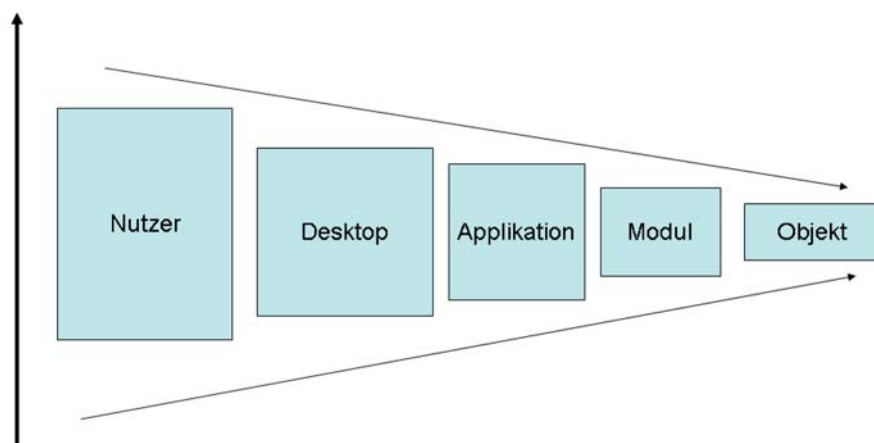


Abb. 10.17 Architektur als Autoritätstrichter

die Powerbox – um von dort für sie gespeicherte Rechte zu erhalten. Damit wird verhindert, dass Benutzer unnötig oft nach Rechten gefragt werden. Abbildung 10.17 zeigt, wie die Autorität in dieser Architektur vom Nutzer bis hin zum Objekt immer kleiner wird.

Unsere spätere Diskussion des DARPA-Browsers demonstriert eine Anwendung dieser Architektur und zeigt, wie dadurch auch Extensions von Software gefahrlos integriert werden können.

Designation und Autorität

Objekt Capabilities vereinen Designation und Autorität in der Referenz. Wer sie besitzt, kann die Ressource verwenden. Wieso führt dieses Konzept zu einer wesentlichen Verringerung von ambierter Autorität? Der Grund liegt in der Umsetzung von Namen zu Referenzen. Wenn eine Applikation File-Namen in Form von Strings erhält, muss sie in der Lage sein, diese in File-Objekte zu verwandeln. Da der Namensraum der File-Namen potenziell unbegrenzt ist, muss die Applikation für diese Umsetzung über alle Rechte des Aufrufers verfügen – selbst wenn nur ein einziger File-Name als String übergeben wird.

Zu der grundsätzlichen Gefahr, dass die Applikation die Rechte missbrauchen könnte, um andere Files des Nutzers zu manipulieren, kommt eine weitere Gefahr hinzu: Das „Confused Deputy“-Symptom, das auftritt, wenn die Applikation über größere Rechte als der Nutzer selbst verfügt (z. B. SetUID) und beim Zugriff auf Ressourcen fälschlicherweise die privilegierten Rechte verwendet (z. B. wenn der Benutzer als File-Namen „/etc/passwd“ oder „kernel32.dll“ übergibt). Nicht zuletzt kann der Nutzer durch besonders geformte Strings zu falschen Berechtigungen verleitet werden, wie dies bei Browser-URLs häufig der Fall ist. Es macht also

Sinn, diesen gefährlichen Prozess der Umformung von Name in Recht einer „trusted component“ zu übertragen. Der DARPA-Browser (Kap. 12) zeigt diese Übertragung im Detail sowie die entstehenden Gefahren, wenn beliebige Applikationsteile solche Umformungen selber durchführen dürfen.

Virtualisierung

Oft wird Virtualisierung fälschlicherweise nur als Architekturmittel von Betriebssystemen gesehen. Untersucht man aber die Fähigkeit zu korrekter Virtualisierung, stößt man schnell auf ein vielfach verwendbares Architekturmuster. Die folgenden Kriterien sind Millers Diskussion der Grundlagen der Virtualisierung von Popek und Goldberg entnommen [Mil, S. 79].

Die Übersicht zu den Grundkonzepten der Virtualisierung (Abb. 10.18) korreliert den CPU-Modus mit der Art der Instruktion und gibt das Verhalten der Instruktion an. Schlecht für die Virtualisierung ist es, wenn Instruktionen in beiden Privilegien-Modi unterschiedlich funktionieren. Ebenso schlecht ist es, wenn unkritische Instruktionen nicht exakt das gleiche Verhalten in beiden Privilegien-Modi zeigen, denn dann kann der Gast die Virtualisierung erkennen.

Die Eigenschaft „Behavior Sensitive“ bedeutet, dass durch die Ausführung der Instruktion Änderungen im Umfeld des Gastes von der Gastsoftware erkannt werden können, d. h. äußere Einflüsse werden spürbar. „Control Sensitive“ bedeutet, dass der Gast Effekte außerhalb seiner selbst veranlassen kann (z. B. die CPU anhalten).

Auf welche Softwareprobleme lässt sich dieses Architekturmuster übertragen? Beispielsweise auf mandantenfähige Software: Viele Applikationen müssen ihren Kunden voneinander isolierte Umgebungen zur Verfügung stellen, z. B. weil die Kunden ihrerseits die Applikationsdienste weiter verkaufen. Das Problem der Mandantenfähigkeit gilt im Allgemeinen als recht komplex, lässt sich aber auf der Basis dieses Architekturmusters besser verstehen.

Wie lässt sich nun diese Form der Virtualisierung implementieren? Zum einen besteht die Möglichkeit, dies auf der Ebene des Typsystems zu tun, nämlich durch Laden unterschiedlicher Klassen gleichen Namens. Das wäre eine Lösung, basierend auf Namespaces. Meist unterscheiden sich die Mandanten jedoch weniger im Verhalten der Klassen als in unterschiedlichen Instanzen von Klassen.

Da sich Virtualisierung an der Schnittstelle von Applikation und Umgebung abspielt, wäre es alternativ denkbar, der Applikation spezialisierte Objekte zu

Instructions	Privileged	Non-privileged
Behavior sensitive	Works	traps
Control sensitive	Works	traps
Innocuous	Works the same way in both modes	

Abb. 10.18 Grundkonzept der Virtualisierung

übergeben und sie nur durch diese auf die Umwelt reagieren bzw. einwirken zu lassen. Dies wäre z. B. durch die Anwendung einer Inversion-of-Control-Architektur möglich, bei der die Applikation ihre Abhängigkeiten am äußeren Interface deutlich macht.

An dieser Stelle wird nochmals deutlich, wie wichtig das Konzept der „Loader Isolation“ wirklich ist: Wenn es der Applikation gelingt, Objekte mit Control- oder Behavior-sensitivem Verhalten auf andere Weise zu erhalten, bricht sie die Virtualisierung. Wir werden im Abschn. 10.5 über „Gegenkräfte“ darauf genauer eingehen.

Software Extensions

Wir haben das Problem der sicheren Erweiterung von Software bereits mehrfach angesprochen. Im Bereich von Betriebssystemen haben wir z. B. das Device-Driver-Interface untersucht und festgestellt, dass es keine Isolation von Kernel und Extensions zulässt. Neben Ansätzen, den Zugriff von Extensions auf die Basissoftware durch Memory-Management-Techniken zu regulieren, haben wir auch softwarebasierte Ansätze untersucht und dabei festgestellt, dass die Durchsetzung strikter Inversion of Control mit der Übergabe von eingeschränkten Argumenten an die Erweiterung eine Form von Isolierung darstellt.

Wenn wir dieses Prinzip mit dem gerade diskutierten Ansatz der Virtualisierung vergleichen, dann stellt sich heraus, dass sichere Erweiterungen letztlich nur durch eine Form von Virtualisierung der von der Extension benötigten Ressourcen erreicht werden können. Ob die Virtualisierung durch Compiler-Techniken wie im Fall von Singularity oder durch Local-Procedure-Call-Techniken erzwungen wird (also durch getrennte Prozessräume), ist letztlich nur eine Frage von Performance und Implementation.

Reflection und Sicherheit

Ein Problem für auf Object Capabilities basierende Systeme stellt zunächst die Verwendung von Reflection dar, da sie Zugriff auf Referenzen außerhalb der öffentlichen Methoden erlauben würde. So verbietet z. B. Singularity den Einsatz von Reflection zur Laufzeit. Auch Embedded-Control-Plattformen wie die des Finread Card Reader, die den Einsatz von Multi-Vendor-Software auf Java Virtual Machines erlauben, haben das Reflection Package aus der Java-Runtime entfernt.

Bedeutet das nun, dass dann die automatische Persistenz von Objekten durch einen Serialisierer (wie er z. B. bei der Container-managed Persistence in JAVA EE verwendet wird) nicht möglich ist? Java löst dieses Problem momentan durch eine Permission, die der Code benötigt, wenn er per Reflection zugreifen will. Dem Sinn von Object Capabilities näher als ein solches globales, grobgranulares Recht wäre es jedoch, wenn der Serialisierer eine Referenz auf eine spezielle Sichtweise eines Objects erhalten könnte – nämlich eine, die ihm den Zugriff auf

Daten erlauben würde. Dieses Konzept versucht, die Autorität des Zugriffs durch ein spezielles Objekt zu verifizieren. Es könnten durchaus mehrere, verschiedene Sichtweisen existieren. Der Vorteil bei diesem Vorgehen wäre, dass ein Zugriff auf Objektinterna granular an bestimmte Objekte gegeben werden kann.

Generell scheint einfachere und besser verständliche Software durch die Repräsentation von Sicherheitsaspekten durch verschiedene Objekte zu entstehen, anstatt durch externe Kontrolle oder verschiedene Modi eines Objektes.

10.3.11 Design Patterns für sichere Software

Es gibt bereits eine Reihe von Büchern zu speziellen Security Patterns, sodass an dieser Stelle nur einige wenige zentrale Lösungsmuster erwähnt werden sollen. Da sichere Software ohnehin sehr viel mit qualitativ guter und robuster Software gemein hat, gelten viele Patterns ohnehin für den gesamten Bereich der Software.

Entscheidend sind jedoch nicht nur die Patterns, sondern vor allem die Einfachheit und Bequemlichkeit, mit der sie innerhalb der Programmiersprachen verwirklicht werden können. Dies kann unter Umständen dazu führen, dass neue Anforderungen an Programmiersprachen gestellt werden müssen (z. B. die Forderung nach Higher-order-Functions und Closures in Java).

Basispatterns

Interception ist ohne Frage die häufigste Technik zur Absicherung von Ressourcenzugriffen. In ihrer härtesten Form erzeugt sie Isolation von Komponenten durch fehlende Erreichbarkeit. Im Normalfall werden Requests gefiltert und/oder modifiziert. Die Basispatterns dafür sind *Proxy*, *Wrapper* und *Decorator*, deren Kenntnis hier vorausgesetzt wird. Aus ihnen lassen sich größere Architekturmuster wie *Reverse Proxy* zusammenbauen [Schu]. An dieser Stelle seien deshalb nur zwei sicherheitstechnisch interessante Spezialisierungen von Basispatterns kurz dargestellt:

Das *Portable Interceptor Pattern* (PI) wurde für die verteilte CORBA-Architektur entwickelt, findet sich aber in abgewandelter Form in JAAS, Servlet-Filtern, EJB Interceptors oder auch im IPTables-Bereich wieder. Ziel des Portable Interceptors ist es, zusätzliche Funktionalität anzubieten, ohne dafür Applikationen oder Middleware ändern zu müssen. Abbildung 10.19 zeigt die Basisarchitektur des Patterns.

Middleware und spezialisierte Interceptoren kommunizieren über ein Standard-API (s. auch [MVB]). Es sind mehrere PIs zulässig, die Reihenfolge kann beliebig sein oder auch administrativ festgelegt werden. Im Wesentlichen prüft oder manipuliert ein PI die Daten eines Requests und kann gegebenenfalls auch Exceptions werfen. Wichtig ist, dass ein PI den Request um zusätzliche Informationen (grün in Abb. 10.20) anreichern kann. Der Server-PI erhält dann Zugriff auf diese Informationen über eine API-Funktion.

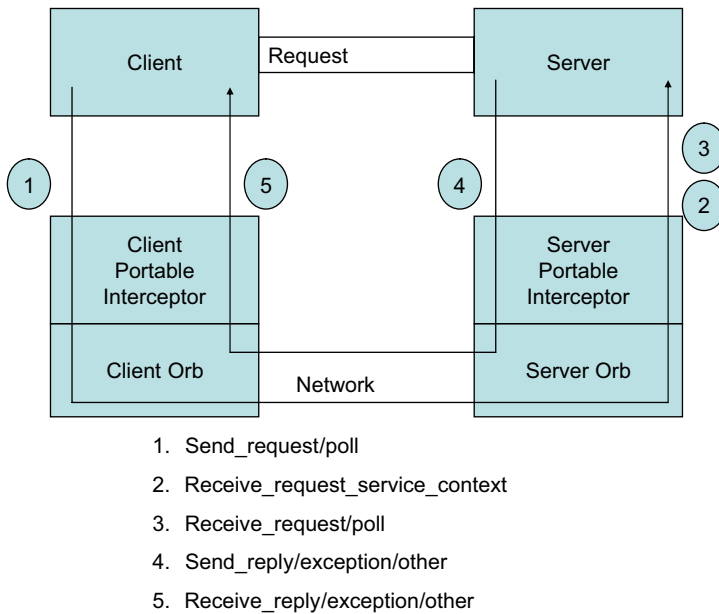
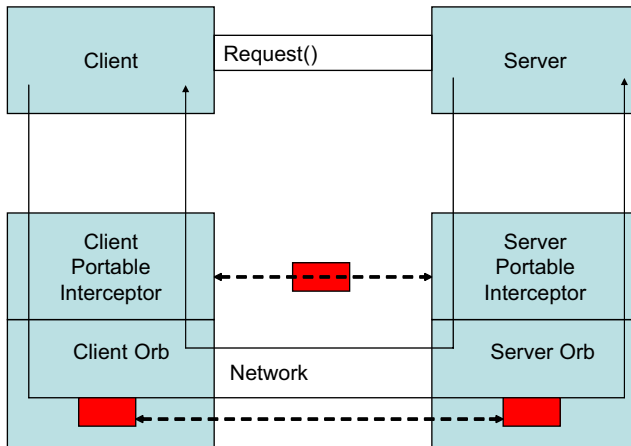


Abb. 10.19 CORBA-Portable-Interceptor-Architektur und API



Context Information Flow (orb-ORB and PI-PI)

Abb. 10.20 Orb-ORB bzw. PI-PI-Metakommunikation

Die Anwendungsgebiete der Interceptoren sind:

- Übertragung von Sicherheitsinformationen zwischen Interceptoren.
- Request-Redirection auf andere Server (Load-Balancing, Fault-Tolerance durch Anbindung an Group Communication).
- Logging und Performance-Daten sammeln.

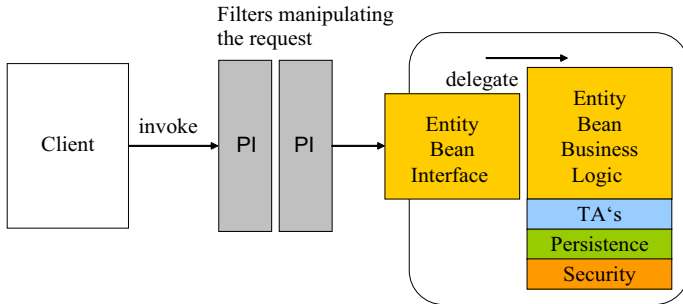


Abb. 10.21 Portable-Interceptor-Konzept, auf EJB-Container angewendet

Einen besonders interessanten Anwendungsfall schildert P. Oser in seinem Vorschlag zur Aufnahme von Interceptoren in die EJB-Architektur [Oser], wie in Abb. 10.21 dargestellt: Er besteht in der impliziten Übergabe von Kontextinformation, wie sie oft für Security und Transaktionsservices nötig ist. Der Interceptor bewirkt also etwas ähnliches wie ein zusätzlicher Parameter in der Signatur des Requests. Dies würde jedoch eine Änderung der Applikation bedeuten. Durch implizite Kontextweitergabe können Interceptoren miteinander kommunizieren und beispielsweise Sicherheitsinformationen weiterleiten. Auch ist es möglich, auf der Client-Seite Requests transparent zu verschlüsseln und im Server-PI wieder zu entschlüsseln, und unnötige Requests zu unterdrücken bzw. anders zu routen.

Bei allen Implementationen des Interceptor Patterns ergeben sich einige aus architektonischer Sicht kritische Punkte:

- Eine deutliche Unterscheidung von Modifikation und Filterung.
- Die Notwendigkeit eines Protokolls für die Entscheidungsfindung, wenn mehrere Interceptoren einen Request filtern.
- Eine notwendige Entscheidung zur Bedeutung der Reihenfolge von Interceptoren.
- Die Kommunikation von Interceptors untereinander, d. h. sowohl zwischen Client- und Server- Interceptoren und innerhalb von Interceptor-Ketten.
- Die Weitergabe von Interceptor-Daten über mehrere Server hinweg.
- Die Erzeugung eigener Antworten durch einen Interceptor.

Letztlich lässt sich durch Interceptoren eine Schicht unterhalb der Applikationen einführen, die Aufgaben der Infrastruktur übernimmt bzw. prüft. So können Interceptoren prüfen, ob ein Request die nötige Qualität besitzt, um von einem Server angenommen zu werden. Dies kann bei Verwendung von verschiedenen Qualitäten der Authentisierung sehr wichtig werden. Allerdings darf nicht verschwiegen werden, dass die Übertragung derartig wichtiger Funktionen auf Interceptoren eine lückenlose Administration erzwingt: Interceptoren sollten deshalb zentral verwaltet werden können und auch zur Laufzeit abgefragt bzw. konfiguriert werden können. Nur so kann die Systemadministration sicher sein, dass die Sicherheit von Requests gegeben ist.

Facet sample /** * <param> target is the underlying powerful object * <param> allowedMethods is a map. The key is a method name, the value * is the set of allowed numbers of arguments. So ["receive" =>[0, 2].asSet()] * would allow the receive method, with either 0 or 2 arguments, to be forwarded. */

```
def makeFacet(target, allowedMethods) {
    def filteringFacet {
        match [verb, args]
        { if (allowedMethods.maps(verb) &&
            allowedMethods[verb].contains(args.size()))
            { return E.call(target, verb, args) } } }
    return filtering Facet}
}
```

```
def chatController def chatFacet:= makeFacet(chatController,
["receive"=>[1].asSet(), "receiveFriend"=>[1].asSet()])
```

Abb. 10.22 Facet-Implementierung aus:
http://wiki.erights.org/wiki/Walnut/Complete#Capability_Patterns

Eine Weiterentwicklung des Interceptor Patterns ist die Definition von Token als Kontextinformation, durch die Interceptoren miteinander kommunizieren. Diese Token können über mehrere Server hinweg transportiert werden und jeder PI kann weitere Einträge hinzufügen. Die Einträge können ihrerseits kryptografisch abgesichert werden. Auf diese Weise entstehen Request-Logs, die mit dem Request mitfließen und die alle Beteiligten protokollieren können. Der Websphere Application Server verwendet sechs solcher Token für Delegation, Logging etc.

Das zweite Basispattern, das wir hier besprechen wollen, ist die mehrstufige Initialisierung von Objekten. Dieses Pattern trennt die Erzeugung eines Objekts von seiner Benutzung. Es wird seit langem beispielsweise durch Factory Methods realisiert, meist jedoch unter dem Gesichtspunkt der Flexibilität und Erweiterbarkeit von Software. Die mehrstufige Initialisierung erlaubt jedoch auch die feingranulare Vergabe von Autorität, kontrolliert durch entsprechende Methoden.

Der Code in Abb. 10.22 ist ein Beispiel für eine parametrisierbare Filtering-Facet-Methode .

Spezielle Security Patterns

Neben Kombinationen von allgemeinen Software-Patterns zu Architekturmustern gibt es auch reine Security Patterns, z. B. der sogenannte Honeypot: Eine Falle, bestehend aus einem scheinbar ungeschützten Rechner, die potenzielle Angreifer anlocken soll, sodass damit ihre Handlungen unter Laborbedingungen analysierbar sind. Während der Honeypot unter die Infrastrukturmuster fällt, gibt es auch einen Bedarf an reinen Softwaremustern, die spezielle Sicherheitsaspekte repräsentieren: Es sind vor allem föderative und kollaborative Pattern, wie sie zurzeit im Umfeld der Web Services Security populärer werden. Um message-basierte Security zwischen Applikationen realisieren zu können, braucht es Abstraktionen jenseits von

Signatur und Hash-Werten. Diese Abstraktionen müssen analog zu den Sicherheitsprinzipien des Alltags modelliert werden, z. B.:

- Vertrag,
- Vollmacht,
- Erlaubnis,
- Ticket und
- Notar (Trusted Third Party).

Aus diesen sozialen Mustern müssen Sicherheitsbausteine durch Frameworks zur Verfügung gestellt werden, die es Entwicklern erlauben, auf der Ebene der Semantik zu arbeiten, während die konkrete Absicherung der Bausteine vom Framework übernommen wird.

Diese Muster sind übrigens nicht zufällig in ihrer Wirkung ähnlich den Capabilities. Sie können wie Schlüssel wirken, indem sie z. B. weitergegeben werden können. Die steigende Verflechtung der Geschäftsmodelle, wie sie im Bereich der serviceorientierten Architektur (SOA) bzw. der Web-Services gepredigt werden, wird einen Trend in Richtung von Sicherheitsbausteinen auslösen, die wie Capabilities funktionieren. Nur so lassen sich die granularen Berechtigungen über verschiedenste Infrastrukturen hinweg erreichen.

10.4 Concurrency

“A folk definition of insanity is to do the same thing over and over again and to expect the results to be different. By this definition, we in fact require that programmers of multi-threaded systems be insane. Were they sane, they could not understand their programs.”

[Lee, S. 5]

Wieso braucht es ein Kapitel zu Nebenläufigkeit in einem Buch zu sicheren Systemen und sicherer Software? Auf den ersten Blick haben Techniken der parallelen Verarbeitung von Aufgaben kaum etwas mit Sicherheit zu tun. Allerdings haben wir in den letzten Abschnitten mehrfach gesehen, dass selbst Basismechanismen wie Seiteneffekte oder Architekturtypen wie IOC einen enormen Einfluss auf die Sicherheit eines Systems haben können. Wir müssen also anders fragen: Welchen – möglicherweise negativen – Einfluss hat Nebenläufigkeit auf die Sicherheit? Genau diesen Einfluss werden wir in diesem Kapitel untersuchen und am Ende feststellen, dass das Thema Nebenläufigkeit erstaunliche Parallelen zum Thema Sicherheit innerhalb einer Systemarchitektur aufweist: Fragen der Vorgehensweise, der Usability, nach Bausteinen für das Design und nicht zuletzt die Fragen nach Modellierung und Beweisbarkeit sind fast dieselben.

Beginnen wir gleich mit dem Kern des Problems zwischen Nebenläufigkeit und Sicherheit. Ein wesentlicher Bestandteil eines sicheren Systems ist die Ausführung von Aufgaben „nach Plan“. Pläne sind üblicherweise Sequenzen von

Voraussetzungen, Aktionen und Folgen, die sich wiederholen. Spätere Aktionen bauen auf früheren Prüfungen auf und so weiter. Nebenläufigkeit kann genau diese Planbarkeit durch Aktionen anderer Akteure durcheinander bringen (siehe dazu M. Millers „Interleaving Hazards, S. 97ff. in seiner Thesis zu „Robust Composition“ [Mil]). Das ist das sog. „Interleaving“ und es tritt sowohl innerhalb eines Prozesses auf, z. B. beim gemeinsamen Zugriff auf Datenstrukturen, wie auch in Form von I/O-Interleaving zwischen Prozessen in verteilten Systemen (siehe dazu den ausgezeichneten Google Tech-Talk von S. Srinivasan – einem der Väter des BEA-Weblogic-Application-Server [Srin08]). Die Folge sind Race-Conditions, also Ergebnisse, die von der zufälligen Folge dieser Interleavings bestimmt sind. Das System verhält sich nicht-deterministisch und das betrifft nicht nur die allgemeine Verarbeitung von Daten oder Ereignissen, sondern ganz besonders auch die Pläne, die für die Sicherheit des Systems selbst gemacht wurden. Ein klassisches Beispiel ist der Plan, dass zunächst per `stat()` ein File geprüft wird, bevor es in einer zweiten Aktion manipuliert wird. Gelingt es einem Angreifer durch Nebenläufigkeit, das File in der Zwischenzeit zu manipulieren – z. B. einen Symbolic Link abzuändern – dann schlägt der Sicherheitsplan fehl. Der Grund dafür ist der, dass die meisten Pläne atomar gedacht sind, also als untrennbare Einheit von Aktionen verstanden werden müssen. Und genau das ist es, was Nebenläufigkeit oft zerstört.

Das Gefahrenpotenzial der Nebenläufigkeit hängt jedoch stark davon ab, wie Multi-Threading eingesetzt wird. Von den drei gebräuchlichen Formen des Multi-Threading, also Deklarativ, Message/Event-basiert und Shared State Multi-Threading, ist es leider die gefährlichste Variante – das Shared State Multi-Threading –, die momentan am populärsten ist. Soweit die Probleme nicht „embarrassingly parallel“ sind – also sich perfekt in getrennte Datenbereiche partitionieren lassen – sind Lösungen, die auf dieser Variante basieren, leider nicht formal verifizierbar. Und sie leiden häufig an Qualitätsproblemen – die ja oft auch Sicherheitsprobleme sind.

“Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism.

Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that nondeterminism.” [Lee, S. 1]

Das kanonische Beispiel für die Probleme von Shared State Multi-Threading ist die Implementation des Observer Patterns, wie in Abb. 10.23 dargestellt. [Lee] und [Mil] zeigen beide unterschiedliche Formen von Nicht-Determinismen bei der Implementation des scheinbar einfachen Patterns auf.

Die Problematik der Multi-Threaded-Implementation erkennt man am Besten, wenn man nach den Garantien dieses Interfaces fragt. Was würde man sinnvollerweise von einem Mechanismus zur Benachrichtigung bei Änderungen eines beobachteten Objekts erwarten?

```

class Object {
  Int value;
  List observers;
  Synchronized addObserver(object) {
    observers.add(object);
  }
  Synchronized notifyObservers() {
    for (Object o : observers) o.update(this, value)
  }
  Synchronized changeVal(int val) {
    value = val;
    notifyObservers();
  }
  Void update(Object o, int v) {
    // receive an update from observed object
  }
}

```

Possibility of deadlock or concurrent modification exception. Unclear timing effect when added object becomes effective

Possibility of deadlock or reversed notification order even when single-threaded

Possibility of deadlock, livelock or incomplete notification cycle due to abused notification thread

Abb. 10.23 Nicht-Determinismen bei der Implementation des Observer Patterns

- Es werden die Benachrichtigungen in der richtigen Reihenfolge verschickt.
- Der Empfänger kann sich darauf verlassen, dass der letzte eintreffende Event den letzten Zustand des beobachteten Objekts repräsentiert.
- Der Beobachter kann während der Benachrichtigung den Status des beobachteten Objekts ändern, ohne dass sich die Reihenfolge der Benachrichtigungen verfälscht.
- Die Beobachter des Objekts können unabhängig von der Benutzung des Interfaces des beobachteten Objekts weitere Interfaces von anderen Objekten in beliebiger Reihenfolge nutzen.
- Es werden bei einer Änderung des beobachteten Objekts zuverlässig alle Beobachter informiert.
- Der Zeitraum für das Senden von Update-Nachrichten ist annähernd konstant für alle Beobachter.
- Die Implementation des Mechanismus ist gegenüber dem Verhalten der Beteiligten robust.

M. Miller nennt die letzte Bedingung auch das „defensive consistency“-Prinzip. Es ist unter den Bedingungen des Shared State Multi-Threading praktisch nicht einzuhalten und würde bedeuten, dass das beobachtete Objekt alle anderen Objekte an der Erzeugung von Threads hindern müsste, um z. B. Deadlock zu vermeiden [Mil, S. 113 ff.].

Die obige Implementation wird je nach Einsatz des Monitors (synchronized) bzw. dem single- oder Multi-Threaded-Betrieb unterschiedliche Fehler zeigen.

Von simplen Concurrent Modification Exceptions durch Manipulation der Beobachterliste während des Sendens von Updates über die Umkehrung der Benachrichtigung, wenn der Beobachtete während des Updates eine erneute Änderung verursacht, bis hin zu undurchsichtigen Deadlock- und Livelock-Situationen, wenn Beobachter weitere Objekte in umgekehrter Reihenfolge verwenden.

Selbst kopieren der Liste vor dem Senden der Updates ist keine Lösung, sondern ermöglicht Änderungen in der Reihenfolge der Benachrichtigungen, je nach Interleaving der Threads.

Das Deadlock-Problem dürfte dabei insgesamt das Schwierigste sein. Zur Verdeutlichung kurz nochmals das Szenario: Thread X hat Objekt B gelockt (durch Aufruf einer synchronisierten Methode) und will nun einen Beobachter bei Objekt C eintragen. Thread Y schickt gerade eine Update-Nachricht an Objekt D und möchte innerhalb dieser Update-Nachricht eine synchronisierte Methode von Objekt B aufrufen. Thread X kann somit keinen Beobachter beim Objekt C registrieren (C ist während des Aussendens der Updates gesperrt) und Thread Y kann die Benachrichtigung nicht zu Ende führen, weil er nicht auf das von Thread X gesperrte Objekt B zugreifen kann. Beide können ihre zuerst allokierten Objekte nicht mehr freigeben.

Lee spricht an dieser Stelle davon, das große Maß an Nicht-Determinismus durch Shared State Multi-Threading zu beschneiden, z. B. durch genaue Absprachen über die fixe Reihenfolge von Locks auf Objekten – was der genauen Definition von Aufrufpfaden gleichkommt [Lee, S. 8].

Hier kommt eine wichtige Parallele zu anderen Mechanismen (un-)sicherer Software ins Spiel: Die mangelnde Usability des Mechanismus, die sich letztlich als fehlende Zuverlässigkeit auswirkt. Wir haben dies bereits bei der Verwendung von paarweisen Funktionen mit modalem Charakter, oder bei der Verwendung von Sprachen, die nicht memory-safe sind, kennen gelernt. Theoretisch könnte man sich vorstellen, dass sich auch mit diesen Mechanismen sicher arbeiten lässt – aber eben leider nicht in der Praxis. Und es gibt für diese Mechanismen bis heute keine Möglichkeit, die Richtigkeit von Lösungen auf Basis dieser Mechanismen zu beweisen.

Wie problematisch dies letztlich für sichere Systeme ist, belegt Lee mit einer Anekdote aus einem mehrjährigen Forschungsprojekt. Die dort eingesetzte nebenläufige Software auf Basis von Java-Monitoren sollte gleichzeitige Modifikationen über das GUI zuverlässig durchführen können, ohne dass Inkonsistenzen zwischen den Threads auftreten durften. Die Software wurde einem rigorosen Modellierungsdesign und Testprozess unterworfen. Nach vier Jahren trat völlig unerwartet ein Deadlock ein.

Lee zieht daraus einige sehr interessante Schlussfolgerungen. Statt zu versuchen, die durch Shared State Multi-Threading verursachten Nicht-Determinismen mühsam zu begrenzen, sollte grundsätzlich zunächst mit deterministischen Komponenten begonnen werden und die Nicht-Determinismen nur dort, wo unbedingt notwendig, eingeführt werden. Diese Forderung ist ganz ähnlich der Forderung nach Reduzierung von ambierter Autorität in Systemen, wie sie von Anhängern

des POLA-Prinzips bzw. der Object Capabilities aufgestellt wird. Zur Erinnerung: Ambiente Autorität ist die grundsätzliche Möglichkeit, Effekte zu erzielen, die dann anschließend mehr oder weniger erfolgreich durch Regeln, Rechte etc. eingedämmt werden kann.

Aber die Ähnlichkeiten zwischen Problemen der Nebenläufigkeit und Problemen der Zugriffskontrolle (letztlich bedeutet ambiente Autorität ja nichts anderes, als grundsätzliche Erreichbarkeit) gehen noch weiter. Die Vertreter der Object-Capability-Idee sehen in Softwarebausteinen zur Reduktion von Effekten ein zentrales Mittel für die kontrollierte Propagation von Rechten bzw. Autorität (in Kap. 14 zeigen wir dann, dass es tatsächlich ohne solche Bausteine keine Möglichkeit gibt, bestimmte Schwächen von Capabilities, wie z.B. die Rücknahme von Autorität, zu umgehen).

Lee argumentiert in Bezug auf die Kontrolle von Nebenläufigkeit ganz ähnlich und erwähnt z.B. das Map Reduce Pattern als Beispiel. Map und Reduce sind Softwarebausteine, die jeweils eine Funktion annehmen und auf eine Kollektion von Dokumenten anwenden. Dieser Vorgang ist durch die Verteilung der Iteration über die Kollektion auf mehrere Rechner extrem parallelisierbar. Das Ganze beruht auf der Vermeidung von Seiteneffekten durch die Einhaltung funktionaler Prinzipien [Joel]. Lee geht jedoch über den bloßen Einsatz von Bausteinen zur Kontrolle der Nicht-Determinismen hinaus. Er sieht im Einsatz von Coordination Languages als Ergänzung allgemeiner Programmiersprachen die Lösung des Problems. Eine Coordination Language würde dann Prinzipien wie Map/Reduce zur Verfügung stellen und gleichzeitig die Einhaltung der damit verbundenen Constraints überwachen. Hier trifft sich Lee wiederum mit Ansätzen, sichere Kollaboration durch den Einsatz von logischen Ausdrücken zu formulieren und in Model Checkern prüfen zu lassen. Siehe dazu die Thesis von F. Spiessens in Kap. 14 [Spiess].

Es würde sich daher sehr lohnen, die anderen Prinzipien genauer zu untersuchen. Dies geht allerdings weit über den Rahmen dieses Buches hinaus – der interessierte Leser sei auf van Roy und Haridi [RH] verwiesen, die hervorragende Grundlagen für eine solche Analyse bieten. Wichtige theoretische Grundlagen sicheren Multi-Threadings hat Hoare in Communicating Sequential Processes (CSP) geschaffen [Hoare]. Dass auch Java-Entwickler nicht notwendigerweise auf Shared State Multi-Threading beschränkt sind, zeigt Belapurkar anhand der in Java entwickelten JCSP Bibliothek [Bela]. Anhand seines bekannten “Wot, no chickens?”-Beispiels zeigt P. Welch die Möglichkeit des Aushungerns von Java Threads trotz permanent neu erzeugter Ressourcen [Welch96]. Entwickler, die CSP einsetzen wollen, seien darüber hinaus auf die CSP-Homepage Wotug verwiesen. Hier finden sich eine mathematische Grundlage des CSP Ansatzes – über den z.B. der Shared-State-Ansatz nicht verfügt – sowie Ansätze und Werkzeuge zum Thema CSP [Wotug]. Hier deshalb nur noch einige kurze Hinweise auf die Qualitäts- und damit auch Sicherheitsproblematik hinter Multi-Threading, wie es klassisch als „Shared State“ Multi-Threading in Abb. 10.24 dargestellt ist. Und im Anschluss daran einige Alternativen, wie z.B. Futures/Promises.

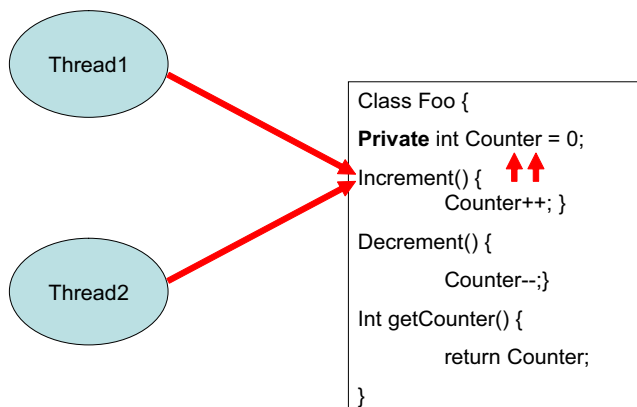


Abb. 10.24 Shared State Multi-Threading

Kennzeichnend für Shared State Multi-Threading ist, dass mehrere Threads auf gemeinsame Daten zugreifen können und deshalb durch spezielle Mechanismen für die Konsistenz der Daten gesorgt werden muss. Wir sehen dabei bereits das Grundübel: Grundsätzlich sind alle Daten für jeden Thread innerhalb eines Prozesses erreichbar.

Die Kritikpunkte am Shared State Multi-Threading sind die folgenden:

- Nur durch den Einsatz von Mutual-Exklusion-Techniken kann eine Inkonsistenz der Daten vermieden werden.
- Der Einsatz solcher Techniken (Locks, Semaphore, Monitore) ist für Entwickler schwierig zu durchschauen.
- Interfaces ändern sich beim Übergang von Single-Treaded zu Multi-Threaded.
- Die „Liveness“-Bedingung von Programmen wird neben der Konsistenzbedingung ebenfalls verletzt: Deadlocks oder Lifelocks verhindern den Fortschritt des Programms.

Bedenklich ist, dass diese Phänomene zwar automatisch entdeckt werden können, es jedoch der Kooperation und Absprache der Entwickler bedarf, um sie zu Vermeiden. Die Strategie zur Vermeidung besteht in der Absprache über die Reihenfolge von Ressourcenzugriffen. In verteilten, großen Umgebungen ist dies schnell ein Problem.

Aber sind Objekte und Concurrency nicht orthogonale Aspekte, d. h. unabhängig voneinander? Das folgende kleine Stack-Beispiel (Abb. 10.25) zeigt, dass dies nicht so ist: Ein Stack-Interface kann und muss sich bei der `pop()`-Methode ganz anders verhalten, wenn es weiß, dass es Multi-Threaded verwendet wird. Ist nichts auf dem Stack, kann die Methode den Aufruf blockieren, statt eine Exception zu erzeugen. Dieses Verhalten wäre im Single-Threaded-Fall unsinnig.

Es ist also nicht möglich, Code einfach zu migrieren bzw. zu verwenden, ohne ihn auf die Fähigkeit für Shared State Multi-Threading zu prüfen. Ansonsten riskiert man Zuverlässigkeit und Sicherheit.

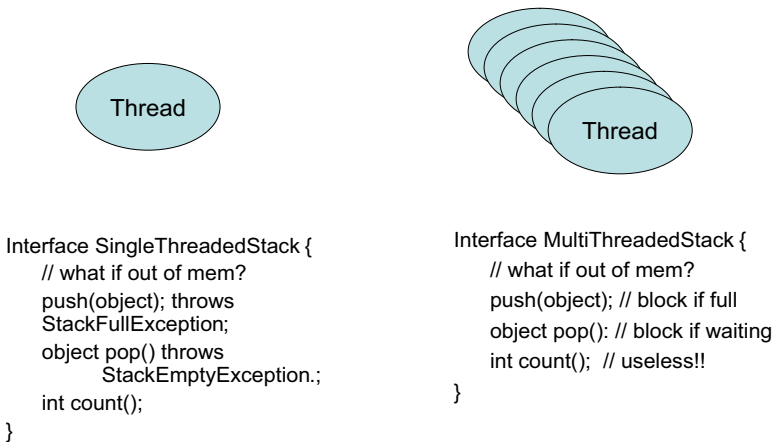


Abb. 10.25 Nicht-orthogonale Aspekte der Nebenläufigkeit

Wolken von Nicht-Determinismus

Woher kommen die Sicherheitsproblematiken beim Einsatz von Shared State Multi-Threading? In Kap. 4 wurden bereits Fälle von Administrationsscripts gezeigt, die zunächst einen Sicherheitstest durchführten und anschließend zugriffen. In der Zwischenzeit hatte jedoch ein anderer Prozess die Ressource abgeändert, z. B. den Symbolic Link auf eine andere Datei gerichtet oder einen neuen Symbolic Link erstellt. Der Zugriff des Admin-Scripts erfolgte dann unter falschen Annahmen.

Durch Shared State Multi-Threading entstehen „Wolken von Nicht-Determinismen“ in Programmen, die dann zu diesen Time-of-Check to Time-of-Use-Fehlern führen oder generell Verfügbarkeit und Zuverlässigkeit beeinträchtigen.

Einfache Fehler zeigen sich gerne im Anlegen von Singleton-Objekten: Statt nur ein Objekt zu erzeugen, werden mehrere gebaut. Dies wird oft durch eine statische, synchronisierte `getInstance()`-Methode vermieden. Diese Methode prüft, ob das Objekt schon existiert und legt es andernfalls an.

Schnell erkannten Entwickler, dass durch diese Technik jeder Zugriff auf das Singleton synchronisiert wurde, und um diese Performanceeinbuße zu umgehen, wurde das Double Checked Locking Pattern verwendet.

Leider gestattet das Java-Memory-Modell diese Technik nicht: Die Referenz eines Objektes darf außerhalb eines Synchronized-Blocks sichtbar werden, bevor das Objekt effektiv konstruiert wurde (Abb. 10.26). Obskure Fehler beim Booten von Applikationen waren die Folge, und Applikationsentwickler durften sich mit den Details des Java-Memory-Modells im Falle mehrerer Threads auseinandersetzen.

Noch immer findet sich viel Java-Code, der lediglich Write-Zugriffe auf Member-Variablen durch „synchronized“ schützt, die Read-Zugriffe jedoch frei lässt.

Zu diesem Zeitpunkt erkennen dann viele Entwickler, dass Objektorientierung und Shared State Multi-Threading teilweise orthogonale Konzepte sind und dass

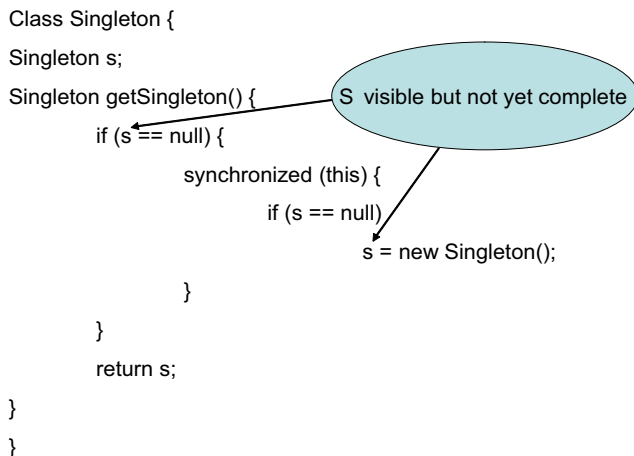


Abb. 10.26 Probleme des Java-Memory-Modells bei Double Checked Locking

„private“ nicht vor gleichzeitigen Zugriffen schützt. Die Verwendung statischer Variablen fällt unter das gleiche Problem und führt gerne bei parallelen Sessions zu Konfusion.

Während diese Techniken noch relativ leicht nachzuvollziehen sind, steigt die Schwierigkeit, sobald Performanceüberlegungen beim Zugriff auf Ressourcen auftreten: Üblicherweise wandern zu diesem Zeitpunkt die Synchronized Statements von den Signaturen der Methoden in den Code der Methoden selbst, um die Zugriffe auf einzelne Ressourcen granularer zu sperren; z. B. will man natürlich nicht beim Einzahlen von Geld auf ein Konto die ganze Einzahlungsmethode – und damit *alle* Konten – sperren, sondern nur das betroffene Konto. Jetzt hat die Entwicklerin die Aufgabe, genau zu unterscheiden, wo die Zugriffe auf einen ganzen Typ und seine Extension (also auf alle Objekte einer Klasse) gesperrt werden müssen (Update von Verwaltungsstrukturen, die z. B. alle Objekte beinhalten) und wo eine instanzbasierte Sperrung nötig ist.

Irrtümer hierbei führen zu langen Strecken, die unter Aussperrung anderer Threads stattfinden und machen aus der Applikation eine Single-Threaded-Version, oder die Datenkonsistenz beginnt zu leiden.

Die Effekte, die dabei für Entwickler auftreten, sind ähnlich wie bei der Entwicklung von Firewall-Regeln: Am Anfang sind die Regeln relativ leicht zu durchschauen, doch bereits ab einer geringen Zahl wächst die Komplexität sprunghaft an. Gleichzeitig gibt es bisher nur ungenügende Möglichkeiten formaler Verifikation.

In diesem Stadium tauchen dann fragwürdige Konstrukte auf: Ein Objekt wird als Monitor für einen Synchronized Block verwendet. An anderen Stellen wird dieses Objekt jedoch durch andere Instanzen ausgetauscht und somit das Monitorobjekt gewechselt. Plötzlich warten einige Threads auf einem Monitor, andere auf einem anderen für die *gleichen* Ressourcen. Bereits jetzt sind die entstehenden Effekte nicht mehr genau nachvollziehbar.

Andere Ansätze versuchen dann maximale Parallelität ohne Synchronisierung zu erreichen. Das ist z. B. bei der Verarbeitung von Web-Requests durch Servlet-Engines wichtig. Um einen maximalen Request-Durchsatz zu erreichen, dürfen hier natürlich keine Methoden auf höheren Schichten gelockt werden.

Dies führt in der Praxis dazu, dass Entwickler bewusst Inkonsistenzen in Kauf nehmen, wenn sie vermuten, dass die Effekte zwar etwas Performance kosten, jedoch keine dauernde Inkonsistenz von Daten oder Applikation erzeugen.

Ein Beispiel: Ein Content Management System verwendet einen Cache, um Seiten zu speichern. Dieser Cache besteht im Wesentlichen aus einer Hashtable. Key-Objekte halten den Seiteninhalt und außerdem Zeitstempel für die Gültigkeit der Seiten (ein Update-Mechanismus des Application-Servers erwies sich leider als fehlerhaft und konnte nicht verwendet werden).

Da Requests parallel eintreffen können, kann es vorkommen, dass mehrere Requests kurz hintereinander die gleiche Seite verlangen. Der erste Thread liest das Key-Objekt und prüft den Zeitstempel. Ist er abgelaufen, wird er künstlich um einige Sekunden verlängert, um weitere Requests nicht zu blockieren. Der Grund liegt darin dass die Erneuerung der Seite eine sehr teure Datenbankoperation einschließt, bei der zunächst eine neue CacheID erzeugt und anschließend der Seiteninhalt aus mehreren Fragmenten zusammengestellt wird.

Es stellte sich heraus, dass die Implementation die falschen Methoden synchronisierte bzw. an wichtigen Stellen nicht synchronisierte. Die Folge war, dass bei fast zeitgleichen Requests auf die gleichen Ressourcen mehrfache Aufrufe zur Erzeugung der Seite an die Datenbank geschickt wurden. Da dies eine teure Operation war, verschob sich die Laufzeit der gesamten Requests nach hinten und eröffnete noch mehr Möglichkeiten für Requests auf die gleiche Ressource, was die Situation weiter verschlimmerte.

Die Entwickler nahmen zwar die Tatsache der überflüssigen mehrfachen Datenbankabfragen in Kauf, übersahen jedoch den dadurch entstehenden Effekt auf die Zeitdauer, innerhalb derer es zu solchen Requests kommen konnte. Kreuzten sich Threads, konnte es außerdem passieren, dass der Zeitstempel eines früheren Threads den eines späteren Threads überschrieb – das klassische „Lost-Update“-Phänomen. In der Folge sprang der Zeitstempel der Objekte teilweise zurück in die Vergangenheit.

Welche Möglichkeiten der Sicherheitsanalyse gibt es in solchen Systemen? Die Antwort muss pessimistisch ausfallen; da der genaue Ablauf massive Nicht-Determinismen beinhaltet, ist praktisch keine Aussage über das Systemverhalten bei plötzlichen Flash-Crowds durch Marketing-Maßnahmen oder Missbrauch durch Spider und Crawler möglich. Auch zeitabhängige Sicherheitsprüfungen sowie Annahmen über die Gültigkeit von Informationen in Caches werden unmöglich.

Beispiele von Multi-Threading-Problemen im Zusammenhang mit Benachrichtigungen und Updates (verschachtelte Updates, Deadlock) finden sich bei [Mil].

Symbian OS verwendet das Konzept von Active Objects, um bei den meisten Servern mit einem einzigen Thread auskommen zu können. Auch die FINREAD-Architektur verwendet ein einfaches Event-Loop-Modell, um mit einem Thread pro Applet auszukommen.

Die Abb. 10.27 nach [REICH] zeigt den Event-Loop-Mechanismus von „E“. Innerhalb eines Containers („Vat“) läuft nur ein einziger Thread. Concurrency lässt sich durch mehrere Container ausnutzen. Auch JAVA EE Enterprise Java Beans werden durch ihren Container serialisiert. Eindrücklich stellt das Diagramm von Peter H. Welch [Welch] (Abb. 10.28) die Zerstörung der OO-Kapselung durch Threads dar.

Dass Browser durch gleichzeitige Events zum Absturz gebracht werden können, wurde mittlerweile für Firefox und Internet Explorer nachgewiesen [Zalew]. Diese Schwachstellen können wahrscheinlich auch zur Übernahme des Browsers durch fremden Code verwendet werden.

Der im übernächsten Kapitel behandelte sichere Browser („DARPA-Browser“) nutzt übrigens ein eventgetriebenes Modell, bei dem für den Entwickler die Se-

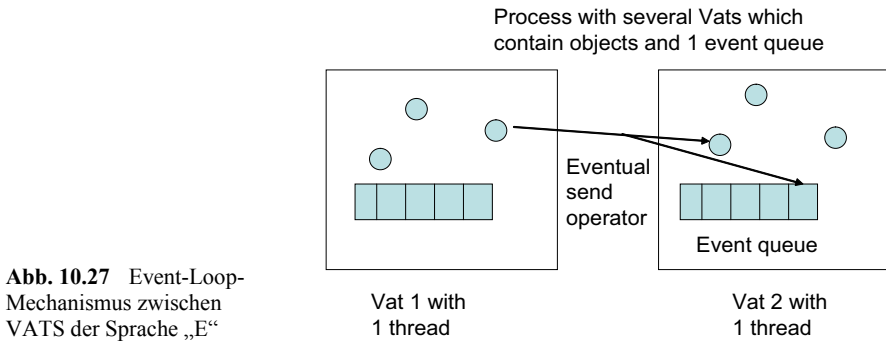
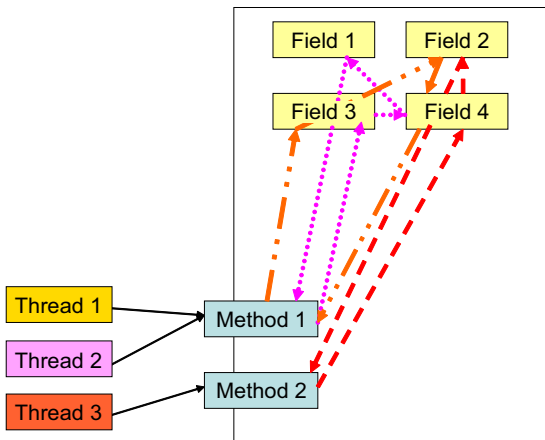


Abb. 10.27 Event-Loop-Mechanismus zwischen VATS der Sprache „E“



Event private fields within an object are unprotected from threads
From: Peter Welch, a CSP library for Java threads

Abb. 10.28 Die Orthogonalität von Kapselung und Concurrency nach P. Welch

mantik eines einzelnen Threads gewährleistet ist. Damit sind keine Attacks durch Race Conditions möglich, wie sie M. Zalewski für den Internet Explorer schilderte [Zalew].

Zum Abschluss des Themas Concurrency und alternative Verfahren soll noch kurz Kilim, ein neues Framework im Java-Bereich, vorgestellt werden. Es wird von Sriram Srinivasan entwickelt, einem der Väter des BEA-Weblogic-Application-Servers und ausgewiesenen Spezialisten für verteilte Systeme. Das Kilim-Framework bietet den Entwicklern die Möglichkeit, buchstäblich hunderttausende von Threads einsetzen zu können, zusammen mit einem auf Message Passing basierenden, Shared-Nothing-Kommunikationsprinzip. Actors bzw. Active Objects sind der dabei verwendete Grundgedanke [Srin08]. Realisiert wird das Framework durch Continuation Passing, statt dem üblichen Co-Routinen-Ansatz als Mittel zur Hauptschleife zurückzukehren. Das Programmiermodell stellt eine Erweiterung des Java-Codes durch Annotationen dar, die anschließend von einem Code-Generator ausgewertet und in Code für das speichern bzw. die Wiederaufnahme von Calls umgesetzt werden. Details zu dieser für Java eher ungewöhnlichen Architektur (Continuation Passing in einer strikt Stack-zentrischen Welt) siehe [Srin].

Abbildung 10.29 zeigt, dass Entwickler explizit die Methoden kennzeichnen müssen, die unterbrochen werden können. Anhand der Annotationen kann dann geprüft werden, ob alle teilnehmenden Funktionen ebenfalls „pausable“ sind (ähnlich den Java Checked Exceptions). Srinivasan streicht besonders die bessere Usability und Zuverlässigkeit von Kilim heraus, die zu wesentlich weniger Fehlern im Bereich Concurrency führt. Durch die Isolation (Shared-Nothing-Ansatz) wird das Testen erst ermöglicht.

Abb. 10.29 Kilim Task mit Kennzeichnung der unterbrechbaren Methoden

Kilim Task

```
class HttpConn extends Task {
    @pausable
    public void execute() {
        while (true) {
            HttpMsg m =
readReq();
            processMsg(m);
        }
    }
    @pausable
    public HttpMsg readReq() {
        ...
    }
}
new HttpConn(mbox).start();
```

Die Zukunft der Concurrency: Multi-Core-CPU's

Shared State Concurrency wie oben beschrieben, hat zwei fundamentale Probleme: Korrektheit und Lebendigkeit – sprich die Vermeidung von Deadlocks. Während dies auf den verfügbaren Plattformen bereits schwierig genug ist, zeichnet sich eine dramatische Verschlimmerung der Situation ab: In Zukunft dürfte eine Performancesteigerung der CPUs weniger über die Erhöhung der Taktrate, sondern mehr über den Einbau zusätzlicher Cores stattfinden. K. Jäger beschreibt in seiner Thesis zu „Finding Parallelism“ die dramatischen Auswirkungen für die Softwareentwickler [Jaeg]. Die Vielzahl an Cores zwingt einzelne Applikationen zu einem Multi-Threaded-Design, das auf der Basis von Shared State Multi-Threading zu großen Stabilitätsproblemen führen wird. Jäger beschreibt die Grundproblematik der Concurrency und weist auf zwei mögliche Alternativen hin: Software Transactional Memory (STM) bzw. Message-Passing-Concurrency. Während Message-Passing-Concurrency in Sprachen wie Erlang sehr erfolgreich eingesetzt wird ist, kennt man das Grundprinzip von STM bisher nur aus Transaktionen mit Optimistic Locking: Statt einzelne Datenbankeinträge zu sperren und so die Gefahr von Deadlocks heraufzubeschwören, werden Daten grundsätzlich kopiert und am Ende einer atomaren Task auf Korrektheit geprüft. Dies geschieht „Lock-Free“, auf Basis eines atomaren Test-and-Swap-Verfahrens auf Basis einer CPU-Instruktion. STM leidet, wie Optimistic Locking generell, unter lang dauernden Transaktionen, die häufig zu einem Rollback führen. Der Vorteil für den Entwickler ist jedoch, dass das Verfahren für ihn transparent ist.

Noch ist unklar, welche Verfahren bei den Multi-Cores letztlich zum Einsatz kommen werden.

„Finding Parallelismus“ ist eine überaus verständlich geschriebene Einführung in die Problematik hochgradiger Concurrency. Ein besonderes Detail darin sind die Antworten bekannter Autoren von Programmiersprachen auf die Frage, welchen Beitrag die Programmiersprachen zum Problem der Concurrency leisten werden. So sind in der Thesis sehr interessante Aussagen von Guido van Rossum, dem Vater von Python, von Bjarne Stroustrup (C++), Joe Armstrong (Erlang), Simon Peyton-Jones (Haskell), Larry Wall (Perl) Yukihiro Matsumoto (Ruby) und Roberto Jerusalem (Lua) zur Lösung dieses Problems enthalten. Die Thesis ist unter [Jaeg] frei erhältlich.

10.5 Gegenkräfte

In diesem Abschnitt sollen einige Techniken, die entweder die Qualität von Software bedrohen oder die Analyse ihrer Sicherheit erschweren, zusammengefasst werden. Die Bezeichnung „Antipattern“ für diese Techniken liegt zwar nahe, jedoch handelt es sich teilweise auch um Idiome oder größere Architekturen, und außerdem werden wir sie nicht in Pattern-Form präsentieren.

10.5.1 Nachträgliches Eindämmen

Dabei handelt es sich eigentlich um ein Meta-Antipattern, da es in den verschiedensten Formen konkreter Antipattern auftritt: Egal, ob eine Applikation mit dem Starten über alle Systemcalls verfügen kann, ob Programmiersprachen beliebigen Zugriff auf den ganzen Speicher einer Applikation erlauben oder ob mit Threads ein Programm zunächst nicht-deterministisch wird: Es liegt immer das gleiche Muster vor. Das System gestattet zunächst viel zu viele Möglichkeiten, Effekte zu erzielen, und der Programmierer versucht anschließend über Rechtedefinitionen, Sandboxes, Semaphore etc. diese Freiheit wieder einzuschränken.

Richtig wäre der andere Ansatz: Die gezielte und planbare Erweiterung von Möglichkeiten, Effekte zu erzielen, z. B. durch die Einführung von Bausteinen, die Nicht-Determinismus gezielt behandeln, oder von Capabilities, die gezielt Rechte propagieren können. Die nächsten Abschnitte zeigen weitere Probleme des nachträglichen Eindämmens auf.

Wie so oft helfen bei der Suche nach Schwachstellen in Systemen Fragen nach der Autorität eines Stücks Code (was kann dieser Code und steht das in direktem Zusammenhang mit dem „Plan“ des Codes?) bzw. im Falle von Nebenläufigkeit die Frage, ob der Code auch Multi-Threaded laufen könnte bzw. was passieren würde, wenn an bestimmten Stellen eine Unterbrechung auftreten würde.

10.5.2 Globale Navigation

Die Abbildung von Bäumen in Software erlaubt durch bidirektionale Referenzen ein problemloses Navigieren durch den Baum. Einfache Methoden erlauben das Hinzufügen oder Löschen von Knoten. Aus Sicht der Erweiterbarkeit ist dies eine sehr flexible Datenstruktur. Aus Sicht der Sicherheit birgt eine Referenz auf einen Knoten im Baum jedoch viel zu viel ambiente Autorität durch die Navigationsmöglichkeit.

Software, die mit Baumstrukturen arbeitet, sollte Sicherheitsbausteine beinhalten, die die beliebige Navigation durch den Verwender ausschließen, indem sie die Referenzen auf den Elternknoten unterdrücken. Damit ist es möglich, Capabilities auf eingeschränkte Sub-Trees herauszugeben. Andere Sicherheitsbausteine (Wrapper) sollten z. B. die Modifizierbarkeit von Knoten verhindern.

Sprachen sollten in Zukunft auch daran gemessen werden, wie gut und bequem sie solche Einschränkungen der Autorität ermöglichen, z. B. durch die Unterstützung von Higher-Order Functions und Closures.

10.5.3 Globale Namespaces

Öffentliche Verzeichnisse, die Objekte beinhalten, erlauben es der Software, relativ einfach Namen auf Capabilities abzubilden. Das ist wiederum aus Sicht der

Flexibilität sehr wünschenswert, denn auf diese Weise lassen sich z. B. Interfaces und Implementationen sehr einfach voneinander trennen und Clients können dennoch Objekte lokalisieren. Auch das dynamische Erweitern von Applikationen durch neue Klassen ist damit einfach möglich. Sicherheitstechnisch ist dieses Vorgehen aber problematisch, wie nicht zuletzt das Websphere-Beispiel des global schreibbaren JNDI-Verzeichnisses zeigt. Solche Verzeichnisse bringen zwei Probleme mit sich: Zum einen das gerade erwähnte Problem der globalen Navigation, das eine große Menge von Objekten zugreifbar macht – unabhängig davon, ob diese Objekte dann noch eine weitere Zugriffsprüfung vornehmen. Das größere Problem besteht jedoch darin, dass der Zugriff auf die Objekte über ein „Pull-Verfahren“ vor sich geht: Die Applikationssoftware besitzt eine Referenz auf ein globales Verzeichnis und holt sich daraus Objekte heraus, statt dass sie sie übergeben bekommt. Dies bricht Konzepte wie Virtualisierung der Umgebung, es sei denn, das ganze Verzeichnis stellt eine virtualisierte Instanz für diese Applikationsinstanz dar, was meist nicht der Fall ist.

10.5.4 Alias-Definitionen

Wir haben bereits Beispiele dafür gesehen, wie durch Alias-Definitionen Zugriffskontrollen umgangen werden können. Dies ist ein weiterer Beleg für die Gefahr bei der Umsetzung von Namen in Ressourcen und Rechte. Hier hilft nur die Normalisierung aller Zugriffe, am besten durch ein gut geprüftes Trusted Module. Mehrfache Codierungsmöglichkeiten wie im Falle von Unicode spielen die gleiche Rolle wie Alias-Definitionen und tauchen regelmäßig als Fehlerquellen auf.

Eine besonders gefährliche Form von Alias-Verwendungen besteht in mehrfachen Zugriffsmöglichkeiten auf Datenbanken. Wenn sowohl eine methodenbasierte Zugriffskontrolle als auch rohes SQL zum Zugriff auf die Daten verwendet werden, besteht die Gefahr, dass die Kontrolle durch Methoden umgangen werden kann. Zwei verschiedene Zugriffsmechanismen haben den Nachteil, dass die Regeln der Zugriffskontrolle nun zweimal definiert werden müssen – in jeweils unterschiedlichen Techniken. Der Preis für schnellere Listenzugriffe über direktes SQL kann dadurch recht hoch werden.

10.5.5 Degeneration von Architektur

Generell kann eine Architektur durch zu starkes Wachstum sicherheitstechnisch degenerieren. Zu viele Komponenten (DMZ mit zu vielen Servern, Betriebssystem mit zu vielen Utilities und Programmen), die miteinander Wechselwirkungen eingehen können (und sich gegenseitig attackieren können) führen dazu, dass sich Risiken multiplizieren.

So führt das Clustering von eigentlich unkritischen Applikationen auf einem zentralen System dazu, dass das Cluster plötzlich ein größeres Ausfallrisiko darstellt, als es bei einzelnen Applikationen der Fall wäre.

10.5.6 *Ambige Interfaces*

Eine Klasse sollte in ihrem Interface nur Typen sichtbar machen, die zu ihrer Aufgabe gehören. Eine `getSystem()`-Methode in einer GUI-Klasse, die Zugriff auf das Java-Runtime-System gibt, ist eindeutig ein Designfehler.

Unterschiedliche Benutzungsmodi von Ressourcen (Admin, HR, Nutzer) sollten besser durch unterschiedliche Objekte (Capabilities) dargestellt werden, als durch unterschiedliche Permissions. Dann drückt sich der Sicherheitsaspekt *innerhalb* der Software aus und wird auch dort sichtbar!

10.5.7 *Paarweise Methoden mit State*

Egal, ob es um das Wechseln zwischen verschiedenen Privilegienebenen oder das Allokieren von Ressourcen geht: Der Einsatz paarweiser Aufrufe zum Setzen und Rücksetzen scheitert an der Vergesslichkeit von Entwicklern sowie den unzähligen Ausstiegsmöglichkeiten aus Funktionen oder Methoden.

Sicherheit und Zuverlässigkeit bringt hier nur die Verwendung einer „Lexical Closure“, innerhalb derer der Zustand X gilt und außerhalb derer er nicht gilt.

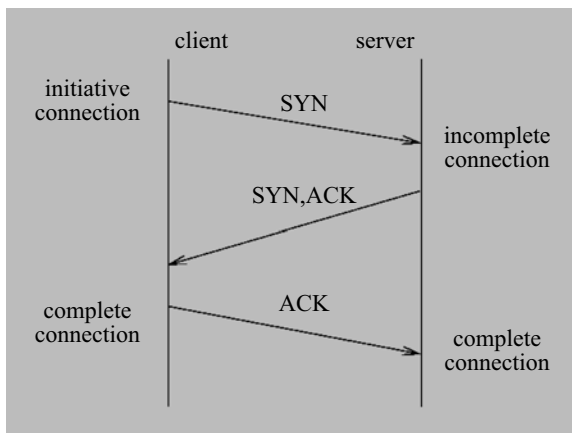
10.5.8 *State halten*

Objekte, die States für Clients halten, können letztlich durch diese Clients in ihrer Arbeitsweise behindert werden, z. B. indem Clients dafür sorgen, dass der Speicher des Prozesses überläuft. Hier handelt es sich nicht um Konsistenzfehler, sondern um einen Fehler in der „Liveness“ des Prozesses, der damit keinen Fortschritt mehr macht. Abhängigkeiten dieser Art sind sehr schwer zu beseitigen, wie das noch immer ungelöste Problem der Denial-of-Service-Attacken zeigt. Software sollte jedoch auf Probleme in der Liveness der Umgebung so reagieren, dass sie ihre Funktion notfalls einstellt, jedoch keine inkonsistenten Zustände oder falsche Aktionen zulässt. Für die Infrastruktur gilt die Regel der frühen Authentisierung von Requests, um die potenzielle Menge der States, die gehalten werden müssen, zu begrenzen.

Ein einfaches Beispiel für die Problematik von State stellen die SYN_FLOOD-Attacken dar, denen TCP-Stacks ausgesetzt sind. Diese nutzen die Tatsache, dass zum vollständigen Aufbau einer TCP-Verbindung ein sogenannter Three-Way-Handshake gehört (Abb. 10.30).

Der Angreifer weiß, dass ein Request zur Verbindungsaufnahme (SYN-Flag gesetzt) vom Netzwerk-Stack gespeichert wird. Statt der Vollendung der Verbindungsaufnahme im dritten Schritt des Handshakes durch den Client, stellt dieser jedoch erneut einen Request zur Verbindungsaufnahme und kann durch entsprechende Wiederholung die Tabellen des Netzwerk-Stacks zum Überlaufen bringen.

Abb. 10.30 State-
Attacke im Netzwerk.
Quelle: www.usenix.org



Manchen Entwicklern fällt es schwer, die Problematik hinter States zu erkennen, da es sich dabei eben nicht um logische Dinge bzw. funktionale Fragen handelt, sondern um die Erschöpfung globaler Ressourcen des Systems.

10.5.9 Überaggrierung

Der OSGI-Evangelist Peter Kriens argumentiert vehement gegen eine Verwaltung von Abhängigkeiten auf Basis von JAR-Files (Bundles im Sinne von OSGI) und beleuchtet damit ein Grundproblem des Softwaredesigns, nämlich den Umgang mit Abhängigkeiten. Generell neigen Entwickler dazu, im Eifer des Gefechts viel zu große Codeeinheiten zu importieren. Es ist schlicht bequemer, ein ganzes Package anstatt einer einzelnen Klasse zu importieren.

Die Nachteile für die Softwarearchitektur, aber auch für die Sicherheit sind enorm: Die Architektur leidet unter potenziell falschen Abhängigkeiten zwischen Klassen, die es in der Realität nicht gibt. Die Sicherheit leidet, weil Clients potenziell Zugriff auf zu viele Klassen erhalten und damit unter Umständen ambiente Autorität erhalten. Noch schlechter als die Versionierung auf Package-Level ist aber mit Sicherheit die Versionierung auf der Ebene von JAR-Files, die potenziell viele Packages beinhalten können. Damit werden generelle Abhängigkeiten erklärt, die dann das granulare Ändern einzelner Teile sehr erschweren. Dieses Problem zeigt sich spätestens dann, wenn einzelne Services ausgetauscht werden sollen und Unklarheit darüber besteht, welche anderen Services davon betroffen sind.

10.6 Sicherheit der Applikation

Daniel Bernstein, der geistige Vater von gmail, hat kürzlich die Erfahrungen bezüglich Sicherheit aus zehn Jahren gmail-Entwicklung in einem interessanten

Paper zusammengefasst [Bern]. Lassen Sie uns diese Erfahrungen mit den obigen Mechanismen sicherer Software vergleichen.

Ausgangspunkt für Bernsteins Analyse ist die Tatsache, dass qmail im Vergleich mit z. B. Sendmail eine bei weitem geringere Zahl an sicherheitskritischen Fehlern aufwies. Der Autor führt dies u. a. auf die Codegröße zurück, jedoch scheint uns das angesichts der vorgelegten Zahlen (Sendmail ca. 230.000 Wörter, qmail ca. 130.000 Wörter im Code). Zunächst seltsam. Allerdings führt Bernstein an, dass die Funktionalität beider Lösungen grob identisch sei und somit ein beträchtlicher Unterschied in einer problematischen weil unnötigen Komplexität des Sendmail-Codes liegt. Der Autor von qmail wurde übrigens u. a. durch die berühmte „qmail-Sicherheitsgarantie“ bekannt, die jedem 1.000 Dollar verspricht, der einen sicherheitsrelevanten Fehler in qmail findet, der Benutzern erlaubt, die Mail anderer Benutzer zu lesen oder zu korrumpieren.

Bernstein teilt die Prinzipien sicherer Applikationen in drei Aufgaben ein:

- Eliminieren von Fehlern,
- Eliminieren von Code und
- Eliminieren von Trusted Code.

Ad 1. Eliminieren von Fehlern

„Hidden Data Flow“ erwies sich auch für qmail als eine häufige Fehlerquelle und Angriffsmöglichkeit für Buffer-Overflow-Attacken. Gemeint sind unerwartete Seiteneffekte durch fehlerhaften Array-Zugriff oder die Verwendung von globalen Variablen jeglicher Art. qmail versucht, dies durch Mechanismen sowie Architektur zu vermeiden. So gibt es spezielle Arrays, die automatisch wachsen und beim Lesen von nicht-initialisierten Bereichen automatisch Null einfügen. Architektonisch wurde sehr viel Wert auf Partitionierung in kleinere Einheiten gelegt. Viele qmail-Funktionen werden in separaten, durch Pipelines verbundene Prozesse, erledigt [Bern, Kap. 3.1ff.].

Im Kontext der Integer-Semantik bemängelt Bernstein die schlechte Usability z. B. durch den automatischen Überlauf nach der größten möglichen Integer – einer der wenigen Fälle, bei denen die Gefahr eines Buffer Overflows in qmail bestanden hatte.

“Most programming environments are meta-engineered to make typical software easier to write. They should instead be meta-engineered to make incorrect software harder to write. An operation that is not exactly what I normally want should take more work to express than an operation that is exactly what I normally want. There are occasions when I really do want arithmetic modulo 232 (or 264), but I am happy to do extra work on those occasions.” [Bern, Kap. 3.2]

Beides, versteckte Datenflüsse durch Seiteneffekte wie auch Probleme mit der Semantik von Integer-Operationen, werden wohl nicht oft als Ursache von Sicher-

heitsproblemen in Software genannt. Sie scheinen zu „grundlegend“ zu sein. Umso überraschender ist es, wenn der Autor eines sehr erfolgreichen Softwarepaketes, das noch dazu eine recht komplexe Schnittstelle bedient, diese Faktoren betont.

Das Parsen von Input – speziell der Unterschied von Meta-Zeichen und „normalen“ Zeichen ist laut Bernstein ein permanenter Anlass für Fehler. Sein Vorschlag ist, die Unterscheidung so weit wie möglich bereits im Design zu vermeiden und ansonsten für jeden Fall eines Meta-Zeichens einen eigenen Test-Case zu schreiben.

Konsequenter Einsatz von Techniken zum flexiblen Testen von Routinen in unterschiedlichen Umgebungen ist der nächste Punkt. Code, der in verschiedenen Umgebungen funktionieren muss (z. B. lokal oder im Netzwerk), sollte so geschrieben werden, dass die Unterschiede in einzelne Komponenten separiert werden und nach Bedarf aktiviert werden können. Im Grunde beschreibt Bernstein hier einen konsequent an Unit-Tests und Mock-Ups orientierte Architektur im Sinne von Dependency Injection.

Ad 2. Eliminieren von Code

Funktionale Ansätze treten bei der Eliminierung von Code noch stärker zu Tage: Die Identifizierung von wiederholten Codestücken und deren consequentes Refactoring in eine gemeinsame Funktion führen zu einer wesentlich kleineren Code-Base und damit zu weniger Fehlern. Dieser Gedanke führt dann logischerweise weiter zur Verwendung von Higher-Order-Funktions als Spezialisierung von Abläufen. Dazu gehört auch das Abfangen temporärer Fehler in den unterschiedlichsten Situationen (z. B. Out-of-Memory-Fälle bei Funktionen, die intern Memory für eine Transformation allokalieren müssen).

Die qmail-Architektur vermeidet Code darüber hinaus durch eine konsequente Verwendung von ohnehin vorhandenen Mitteln des Betriebssystems wie INETD, Mechanismen der Zugriffskontrolle oder des File-Systems. So verzichtet qmail auf eine komplizierte Trennung in Vordergrund- und Hintergrundverarbeitung und empfängt alle externen Nachrichten in einer gemeinsamen Queue durch den INETD-Service. Sendmail enthält stattdessen funktional weitgehend identischen selbst geschriebenen Code mit ungleich höherer Fehlerzahl.

Ad 3. Eliminieren von Trusted Code

Die Größe und Kompaktheit des Codes ist oft ein Kernthema für sichere Software. Speziell die Größe der Trusted Computing Base (TCB), also des Codes, dem schlicht vertraut werden muss, scheint besonders kritisch zu sein. Bernstein spricht jedoch nicht davon, die TCB besonders klein zu halten. Er spricht davon, sie zu eliminieren, und das aus gutem Grund. Selbst kleinste TCBs enthalten meist Code,

der für viele Funktionen nötig ist und der z. B. im Sinne der Confused-Deputy-Attacken missbraucht werden kann.

“‘Secure’ operating systems and ‘virtual machines’ say that their security is enforced by a small base of trusted code. Unfortunately, a closer look shows that this ‘security’ becomes meaningless as soon as one program handles data from more than one source. The operating system does nothing to stop one message inside a mail-reading program from stealing and corrupting other messages handled by the same program, for example, or to stop one web page inside a browser from stealing and corrupting other web pages handled by the same browser. The mail-reading code and browsing code are, in fact, part of the trusted code base: bugs in that code are in a position to violate the user’s security requirements.”

Selbst kleine TCBs gefährden daher die Sicherheit von Applikationen und ihren Benutzern. Aus diesem Grund lehnt Bernstein auch das Principle of Least Privilege (POLP) ab. Statt zu versuchen, mühsam über Rechteeinschränkungen das Schadensausmaß zu verringern, sollte die Notwendigkeit, einer Software „zu trauen“, grundsätzlich beseitigt werden.

Ein probates Mittel dazu ist die Isolation von Funktionen in separate Prozesse, sprich, die Partitionierung von Funktionen der Applikation in verschiedene, getrennte Komponenten. Ansonsten hilft nur, wie Bernstein ausdrücklich hervorhebt, die Software fehlerlos zu schreiben.

Als Beispiel für die Eliminierung von TCBs zeigt Bernstein die Möglichkeit auf, sog. Single-Source-Transformationen – Umwandlungen eines Bildes in ein anderes Format – in eine separate Komponente auszulagern und nur die Quelle sowie das Ziel zu übergeben. Damit sind wir bei einer Architekturform, wie wir sie im Kap. 4 bereits einige Male gesehen haben (Absicherung von Servern bzw. der sichere Desktop Polaris) und wie wir sie in den folgenden Kapiteln zurowsersicherheit durch Objekt-Capabilities noch mehrfach erleben werden: Isolierte Komponenten kleinster Funktionalität, deren Möglichkeiten, Effekte zu erzielen, durch ihre Umgebung streng kontrolliert wird.

Das folgende Beispiel zeigt die Konstruktion einer solchen simplen „Sandbox“.

- Prohibit new files, new sockets, etc., by setting the current and maximum RLIMIT_NOFILE limits to 0.
- Prohibit filesystem access: chdir and chroot to an empty directory.
- Choose a uid dedicated to this process ID. This can be as simple as adding the process ID to a base uid, as long as other system-administration tools stay away from the same uid range.
- Ensure that nothing is running under the uid: fork a child to run setuid(targetuid), kill(-1,SIGKILL), and _exit(0), and then check that the child exited normally.

- Prohibit kill(), ptrace(), etc., by setting gid and uid to the target uid.
- Prohibit fork(), by setting the current and maximum RLIMIT_NPROC limits to 0.
- Set the desired limits on memory allocation and other resource allocation.
- Run the rest of the program.
(aus: [Bern, Kap. 5.2])

Je komplexer der Input für eine Applikation letztlich ist, umso wichtiger ist es, dass einzelne gefährliche Funktionen in abgesicherte Komponenten ausgelagert werden.

Zusammenfassend lässt sich sagen, dass sich die Erfahrungen aus einem lang-jährigen Projekt bezüglich der Mechanismen und Architekturprinzipien von sicheren Applikationen sehr gut mit den von uns vorgeschlagenen Mechanismen und Prinzipien decken. Die Bedeutung von POLA, funktionalen Ansätzen und einer guten Benutzbarkeit für sichere Systeme ist damit auch durch die Praxis bestätigt.

10.7 Sind Sicherheit und Nebenläufigkeit „Aspekte“?

Zum Abschluss noch einige kurze Gedanken über die Art und Weise, wie sich Sicherheitskonzepte bzw. Concurrency innerhalb einer Architektur verhalten. Es geht dabei um die Frage, wie groß der Einfluss auf die Architektur letztlich ist. Oder anders gesagt: Ist es möglich, Konzepte wie die Einschränkung von Erreichbarkeit oder von Nicht-Determinismen nachträglich einer Architektur hinzuzufügen? Die Frage ist letztlich auch für die Modellierung der Architektur von großer Bedeutung: Separierbare Aspekte lassen sich getrennt modellieren, prüfen und durch Injektionstechniken wie AOP nachträglich sogar implementieren. Das wäre in jedem Fall ein großer Vorteil für das Design wie auch die Analyse von sicheren Systemen.

Im Falle der Nebenläufigkeit durch Shared State Multi-Threading scheint die Sache relativ einfach zu sein. Code, der nicht dafür geschrieben wurde – das beginnt mit den Interfaces und schließt die Verwendung von Monitoren, Semaphoren etc. bis hin zur expliziten Verwaltung von Threads über Pools ein – ist normalerweise nicht Multi-Threaded zu betreiben. Das Observer-Beispiel hat dies klar gezeigt. Aber selbst einfachste Interfaces wie die eines generischen Stacks unterscheiden sich deutlich im Falle mehrerer Threads (so macht eine Methode zum Abfragen der Stack-Belegung keinen Sinn. Dafür könnte die Methode zur Rückgabe des obersten Elements den Thread suspendieren, falls der Stack leer ist – in der Hoffnung, dass ein anderer Thread etwas auf den Stack legen wird. Im Single-Threaded-Fall ist hier eine Exception nötig). Die Tatsache, dass der Code nicht anzeigt, ob er nebenläufig verwendet werden kann, ist bereits in sich ein großes Problem, denn er zwingt Entwickler dazu, den Source Code eines Systems im Detail auf diese Frage hin zu prüfen – bis in externe Bibliotheken hinein.

Andere Architekturen, die Nebenläufigkeit verwenden, scheinen dagegen weniger betroffen zu sein: Data-Flow über Pipelines wie im Falle von Grafikprozessoren oder hinter Promises/Futures versteckte Threads, Message-Passing Systeme etc. Sie haben jedoch alle die Eigenschaft, bereits für die parallele Verarbeitung entwickelt worden zu sein. Somit erscheint es mehr als fraglich, dass Nebenläufigkeit ein separierbarer Aspekt ist.

Die Frage, ob Sicherheit einen Aspekt darstellt, ist hingegen nicht ganz so einfach zu beantworten.

Wir haben im Verlauf des Buches eine ganze Reihe von Ansätzen gesehen, um Sicherheit und Software zu kombinieren: Von der strikten Auslagerung der Sicherheit in vorgelagerte, externe Komponenten über die Auslagerung von Sicherheitscode in untere Schichten bis hin zur granularen Überwachung von Methodenaufrufen durch Referenzmonitore bei Sandboxes oder durch JAVA-EE-Container. Alle diese Architekturen beruhen auf dem Prinzip der Interception, d. h. sie arbeiten mit der Einführung von mehr Indirektion. Die Auslagerung von Sicherheitsaspekten aus der Businesslogik der Applikation lässt sich auch gut durch ihre Komplexität begründen, oder dass dadurch Prüfungen erzwungen werden können, sollten diese von der Applikation vergessen worden sein. Wir haben aber auch gesehen, wie manche Aspekte der Sicherheit zur Applikation selbst gehören. Deutlich wird dies vor allem im Bereich der Web Services Security.

Nun stellt sich die Frage, ob Sicherheit auch im technischen Sinne einen „Aspekt“ darstellt. Aspektorientierte Programmierung (AOP) ist in den letzten Jahren populärer geworden. Ihre Grundannahme ist, dass Architekturelemente bzw. Sichtweisen – hier *Aspekte* genannt – unabhängig voneinander entwickelt werden und anschließend „verwoben“ werden können. Diese Kombination erfolgt dann typischerweise durch Techniken der Interception, wobei AOP aber nicht nur darauf reduziert werden sollte. Da die Auslagerung von Sicherheit, wie oben geschildert, ebenfalls meist auf dem Prinzip von Interception beruht, scheinen beide Techniken sehr gut zusammenzupassen. Lediglich die Techniken und Konzepte in diesem Kapitel – Object Capabilities – scheinen hier ein Problem zu sein: Sie verlagern Sicherheit ganz klar in die Architektur einer Software auf allen Ebenen – von der Mikro-Architektur über Bausteine bis hin zum Autoritätstrichter, IOC oder dem Virtualisierungskonzept auf der Ebene der Makro-Architektur.

Eine genauere Analyse der Prinzipien der Object Capabilities zeigt, woher die Probleme kommen: In diesen Systemen werden grundsätzlich nicht Referenzen beliebig verteilt, um sie nachher durch Abfangen kontrollieren zu müssen, sondern es wird vom Beginn an darauf geachtet, dass eine Minimierung der Referenzen und damit der Erreichbarkeit von Ressourcen sichergestellt ist. Es lässt sich also vermuten, dass für eine Kombination von AOP und Object Capabilities generell eine Capability-fähige Grundarchitektur vorliegen muss, da die Capability-Fähigkeit nicht im Nachhinein durch Interception hergestellt werden kann.

Für eine weiterführende Behandlung dieses Punktes siehe [MTS05] und [Spiess].

Literatur

- [And] R. Anderson, Security Engineering, A Guide to Building Secure Distributed Systems, Wiley 2001
- [Bern] D. Bernstein, Some thoughts on security after 10 years of qmail 1.0, 2007, <http://cr.yp.to/qmail/qmailsec-20071101.pdf>
- [Bish] M. Bishop, Computer Security – Art and Science, Addison-Wesley 2002
- [CW] <http://www.computerwelt.at/detailArticle.asp?a=107028&n=6>
- [Close] T. Close, Waterken YURL: Trust Management for Humans, 2004, <http://www.waterken.com/dev/YURL/Name/>
- [EDE] D. Elkadowe, P. Derrin, K. Elphinstone, A memory allocation model for an embedded microkernel, In Proceedings of the 1st International Workshop on Microkernels for Embedded Systems, pages 28–34, Sydney Australia, 2007
- [Ewiki] Wiki zur Programmiersprache E, http://wiki.erights.org/wiki/Walnut/Complete#Capability_Patterns
- [Garf] S. Garfinkel, Design Principles and Patterns for Computer Systems That Are Simultaneously Secure and Usable, Dissertation, Massachusetts Institute of Technology (MIT) 2005, available online at <http://www.simson.net/thesis/>
- [GHJV] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley 1995
- [heise9_06] Gartner: Markt für Sicherheitssoftware wächst weiter zweistellig, Meldung vom 16.9.2006, www.heise.de/security
- [Hunt] G. Hunt et al., An Overview of the Singularity Project, <ftp://ftp.research.microsoft.com/pub/tr/TR-2005-135.pdf>
- [Hardy] N. Hardy, The Confused Deputy Problem, <http://cap-lore.com/CapTheory/ConfusedDeputy.html>
- [Jaeg] K. Jäger, Finding Parallelism – How to survive in a multi-core world. Thesis an der HDM 2008, http://pub.kaijaeger.com/bachelor_thesis.pdf
- [Joel] I. Spolsky, Can your Programming language do this?, <http://www.joelonsoftware.com>
- [DG] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing, on Large Clusters, Google inc.
- [Lee] E.A. Lee, The Problem with Threads, Electrical Engineering and Computer Sciences, University of Berkeley, Technical Report No. UCB/EECS-2006-1, <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html>, Jan. 2006
- [Levy] H.M. Levy, Capability-based computer systems, available online at <http://www.cs.washington.edu/homes/levy/capabook/>
- [Mil] M.S. Miller, Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control, Dissertation, Johns Hopkins University, Baltimore, available online at <http://www.erights.org/talks/thesis/index.html>
- [MMF] M.S. Miller, C. Morningstar, B. Frantz, Capability-based financial Instruments, Proceedings of Financial Cryptography Feb. 2000, <http://www.erights.org/elib/capability/ode/index.html>
- [MS] M.S. Miller, J.S. Shapiro, Paradigm Regained: Abstraction Mechanisms for Access Control, Proceedings of 8th Eighth Asian Computing Science Conference (ASIAN'03), edited by Vijay Saraswat, Springer-Verlag 2003 (available online at <http://www.erights.org/talks/asian03/>)
- [MSLAS] M.S. Miller, M. Samuel, B. Laurie, I. Awad, M. Stay, Caja – Safe active content in sanitized Javascript, October 2007
- [MTS05] M.S. Miller, B. Tullloh, J.S. Shapiro, The Structure of Authority: why security is not a separable concern. In Multiparadigm Programming in Mozart/Oz: Proceedings of MOZ 2004, volume 3389 of Lecture Notes in Computer Science, Springer Verlag 2005

- [Murr] T. Murray, Analysing Object-Capability Security, Oxford University Computing Laboratory, Proceedings of the Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS'08) 2008
<http://web.comlab.ox.ac.uk/people/toby.murray/papers/AOCS.pdf>
- [MVB] R. Baldoni, C. Marchetti, L. Verde, CORBA request portable interceptors: analysis and applications, *Concurrency and Computation: Practice and Experience* 15(6): 551–579 (2003)
- [Oser] P.H. Oser, ORB Interceptors: A Crucial Missing Feature in the Enterprise JavaBeans Model,
http://www.elca.ch/resources/InvokersNeedToBeAddedToJ2EE5_2.pdf
- [REICH] Stefan Reich, Escape from Multi-threaded Hell,
<http://www.drjava.de/e-presentation/html-english/img0.html>
- [RH] P. van Roy, S. Haridi, Concepts, Techniques, and Models of Computer Programming, The MIT Press, 2004
- [Ross] B. Ross, Firefox and the Worry-Free Web, Kapitel 28 in L. F. Cranor, S. Garfinkel (Ed.): *Security and Usability – Designing Secure Systems that People Can Use*, O'Reilly, 2005
- [Scheff] M. Scheffler, J. P. Springer, B. Froehlich, Object-Capability Security in Virtual Environments, Bauhaus-Universität Weimar, To appear in the Proceedings of IEEE VR 2008.
- [Schn] R. Schneider, Das Konzept des Tainting in der Browser Security, HDM 2008,
www.kriha.de/krihaorg/dload/uni/BrowserTainting.pdf
- [Schn06] B. Schneier, Screening People with Clearances, CRYPTO-GRAM, October 2006
- [Schu] M. Schumacher et al., *Security Patterns: Integrating Security and Systems Engineering*, Wiley 2006
- [Spiess] A. Spiessens, Patterns of Safe Collaboration, Phd Thesis 2007, Département d'Ingénierie Informatique, Université catholique de Louvain
- [Srin06] S. Srinivasan, A Thread of One's Own, University of Cambridge, Computer Lab., New Horizons in Compilers Workshop, HIPC, Bangalore 2006
- [Srin08] S. Srinivasan, Kilim: Isolation Type Actors for Java, Google Tech-Talk June 2008
- [Zalew] M. Zalewski in full-disclosure: Concurrency strikes MSIE (potentially exploitable msxml3 flaws)
<http://lists.grok.org.uk/pipermail/full-disclosure/2007-January/051620.html>
- [Wagn] D. Wagner, Object Capabilities for Security, Google Tech Talk Nov. 2007,
<http://www.youtube.com/watch?v=EGX2I31OhBE>
- [Welch] P. Welch, J.M.R. Martin, A CSP Model for Java Threads (and vice versa),
www.cs.kent.ac.uk/projects/ofa/jcsp/csp-java-model.pdf
- [Bela] Abhijit Belapurkar, CSP for Java programmers,
<http://www-128.ibm.com/developerworks/java/library/j-csp1.html>
- [Hoare] T. Hoare, Communicating Sequential Processes, <http://www.usingcsp.com/> (online PDF Version)
- [Wotug] The place for communicating processes, <http://www.wotug.org/index.php>
- [Welch96], P.H. Welch, Wot – no chickens?,
<http://www.cs.ukc.ac.uk/projects/ofa/java-threads/0.html>

Kapitel 11

Fehleranalyse

In diesem Kapitel sollen reale Sicherheitsprobleme innerhalb einer Software analysiert werden. Als Datenbasis dazu dient die Vulnerability-Base des Mozilla-Firefox-Projektes. Neben der Tatsache, dass die Sicherheitsprobleme dank der Open-Source-Strategie öffentlich und damit für eine Analyse leicht zugänglich sind, stellt ein Browser als wichtigstes Interface zwischen einer Web-Anwendung und den Clients eine wichtige Komponente im heutigen Web dar, die aufgrund ihrer Komplexität und vielfachen Funktionalität häufig Sicherheitsprobleme aufwirft.

Auch der Mozilla-Firefox-Browser – lange gerühmt für seine höhere Sicherheit im Vergleich zum Internet Explorer – ist in den letzten beiden Jahren häufig wegen Sicherheitsproblemen in den Medien erwähnt worden (s. etwa [Schm]). Das wirft grundsätzliche Fragen bezüglich seiner Sicherheitsarchitektur auf.

Das Ziel der Analyse ist zunächst ein didaktisches: Das Verständnis für die abstrakten Sicherheitsprinzipien und komplexen Sicherheitsmechanismen soll durch tatsächliche, real aufgetretene Fehler verstärkt werden. Das zweite Ziel der Analyse ist jedoch ein theoretisches: Wie funktionieren Sicherheitsprobleme in großen Softwareprojekten? Kann man für das Auftreten solcher Fehler Regeln definieren? Sind die Fehler systematisch oder beliebig? Wie sicher sind die Fehlerbehebungen? Und nicht zuletzt: Welche Mechanismen werden eingesetzt und zu welchem Zweck?

11.1 Fehleranalyse als didaktisches Prinzip

Softwareentwickler lernen zwar häufig die Grundprinzipien und Mechanismen von sicherer Software, sind aber in der Praxis oft nicht in der Lage diese umzusetzen. Die Grundprinzipien für den Bau sicherer Systeme sind dabei seit der bahnbrechenden Arbeit von Saltzer und Schroeder aus dem Jahr 1975 [SaSch] wohlbekannt und sind von uns auch bereits in Kap. 3 beschrieben worden.

Die Anwendung dieser Prinzipien auf die Softwareentwicklung steht jedoch vor einem großen Problem: Sie sind zu abstrakt, um eine direkte Hilfe oder Hand-

lungsanleitung für Entwickler zu sein. Wie kommen wir also zu solchen Anweisungen und praktischen Richtlinien? Indem wir das „Top-Down“-Vorgehen, welches ausgehend von abstrakten Prinzipien sichere Software entwerfen will, durch ein „Bottom-Up“-Vorgehen ergänzen: Wie sehen echte Softwarefehler im Bereich Security aus? Wie werden sie entdeckt? Welche Prinzipien oder Mechanismen verletzen sie? Lassen sich dabei Muster entdecken und wie werden sie behoben?

Auf diese Weise werden wir in diesem Kapitel versuchen, die abstrakten Prinzipien in direkte Hilfestellungen zu überführen.

11.2 Fehleranalyse als theoretisches Prinzip

Wirkliche Analysen von Softwarefehlern sind recht selten in der realen Softwareentwicklung. Meist wird ein Fehler behoben, ohne dass grundsätzliche Überlegungen zum Auftreten des Fehlers angestellt werden. Schlimmstenfalls wird ein ganzes Projekt eingestellt – also als Fehler angesehen – und dann im Anschluss natürlich nicht weiter auf die Ursachen der Probleme untersucht. Die scheinbar ewige Wiederkehr von Security-Fehlern lässt sich dadurch wenigstens plausibel erklären.

Wir untersuchen daher in diesem Kapitel in einem zweiten Schritt die Ursachen der Fehler: Welche Typen von Fehlern treten auf? Wie schwerwiegend sind sie? Als besonders schwerwiegend dabei werden Fehler angesehen, die systematischer Natur sind und die durch die verwendete Technologie nicht fundamental zu beheben sind (z. B. Sicherheitsverletzungen durch zu viel Autorität). Typische Fragestellungen bei der Klassifikation der Fehler sind die folgenden:

- Lässt sich menschliches und technisches Versagen unterscheiden?
- Beheben die Bugfixes den Fehler, die Fehler oder ganze Fehlerklassen?
- Wie hoch ist die Wahrscheinlichkeit, dass weitere Fehler einer bestimmten Art auftreten?

Die letzte Fragestellung führt unmittelbar über in die Abbildung der gefundenen Probleme auf Basistechnologien, Entwicklungsprinzipien und nicht zuletzt auf die Architektur der Software. Dadurch können wir sozusagen nach einer Vermeidungsstrategie für solche Fehler suchen. Diese Strategien können auf Betriebssystemlevel, Sprachlevel oder Applikationslevel zu finden sein. Dazu wollen wir ganz konkret am Beispiel realer Software die Art und Weise der im Sicherheitsbereich aufgetretenen Fehler analysieren. Als Versuchskaninchen dient uns dazu die ausgezeichnete Datenbank des Mozilla-Firefox-Projektes zum Bereich Security-Vulnerabilities [Moz].

Eine komplette qualitative und quantitative Untersuchung der Fehler überschreitet jedoch bei Weitem den hier verfügbaren Rahmen. Die Analyse beschränkt sich daher auf die ausschnittsweise Untersuchung sicherheitsrelevanter Fehler. An verschiedenen Stellen werden Vorschläge für eine komplette Untersuchung unterbreitet.

Nach der Vorstellung der Datenbasis sowie einigen Überlegungen zur Hypothesenbildung werden in einem ersten Schritt sicherheitsrelevante Fehler schnittsweise vorgestellt. Die Fehler werden auf ihre Funktion untersucht und nach Typen eingeteilt. Bei jedem Fehler wird versucht, ihn auf die Verletzung eines Grundprinzips zurückzuführen. Außerdem wird bei jedem Fehler die Frage gestellt, ob es sich um einen systematischen, architekturbedingten Fehler handelt, oder ob er als singuläres Implementationsproblem anzusehen ist. „Architekturbedingt“ kann hier die verschiedensten Ebenen der Software vom Betriebssystem über die verwendete Sprache bis hin zur Applikationsarchitektur bedeuten.

Im Anschluss an die Vorstellung der Fehler werden Grundprinzipien der Sicherheit von Browsern diskutiert, und wie sich die Sicherheitskonzepte im Laufe der Zeit gewandelt haben.

11.3 Datenbasis und Hypothesenbildung

Die Vulnerability-Liste des Mozilla-Projektes – im Folgenden auch Datenbasis genannt – enthält sämtliche öffentlich gewordene Sicherheitsfehler seit Beginn des Projekts, einschließlich der Mozilla-Suite, Thunderbird und Firefox [Moz]. Abbildung 11.1 zeigt einen kleinen Ausschnitt daraus.

Damit existiert eine komplette Übersicht aller gefundenen Sicherheitsprobleme über die Jahre der Entwicklung eines komplexen Stücks Software hinweg – ein echter Glücksfall für die Analyse von Sicherheitsproblemen, da zu jedem gemeldeten Fehler dank Open-Source auch die Korrektur im Source-Code verfügbar und nachvollziehbar ist.

Ein einfacher Farbcode zeigt die Kritikalität des Fehlers an (critical, high, low). Wir werden hier allerdings eher danach fragen, was genau eigentlich im System einen Fehler zu einem kritischen Fehler macht. Die Einschätzung der Kritikalität in der Datenbasis beruht dagegen auf den möglichen Effekten des Fehlers. Leider wurden die Sicherheits-Fehler über die Jahre hinweg in unterschiedlichen Formaten und mit Inkonsistenzen im Bereich der Namensgebung abgespeichert (es gibt aber ein Mozilla-Projekt, das die Fehlermeldungen standardisieren soll).

Eine qualitative Analyse der Fehler ist auf der Datenbasis sicher möglich und wird hier auch zum Teil vorgenommen. Wie weit eine quantitative Analyse möglich ist, wird sich noch zeigen, sie würde jedoch einige Arbeit im Bereich der Datenaufbereitung voraussetzen.

Quantitativ zu überprüfende Fragestellungen wären z. B.:

- Ändert sich die Art der Fehler im Laufe des Projektes?
- Zeigen Mozilla und Firefox das gleiche Fehlerverhalten?
- Wie verteilen sich Fehler bezüglich ihres Ortes (Kern, Extensions, etc.)?
- Wie verändert sich die Schwere der Fehler im Laufe des Projektes?
- Wie verändern Architekturänderungen bzw. -entscheidungen das Fehlerbild?
- Konvergieren die Sicherheitsfehler?

Fixed in Firefox 1.0.8

MFSa 2006-27	Table Rebuilding Code Execution Vulnerability
MFSa 2006-25	Privilege escalation through Print Preview
MFSa 2006-24	Privilege escalation using crypto.generateCRMFRequest
MFSa 2006-23	File stealing by changing input type
MFSa 2006-22	CSS Letter-Spacing Heap Overflow Vulnerability
MFSa 2006-19	Cross-site scripting using .valueOf.call()
MFSa 2006-18	Mozilla Firefox Tag Order Vulnerability
MFSa 2006-17	cross-site scripting through window.controllers
MFSa 2006-16	Accessing XBL compilation scope via valueOf.call()
MFSa 2006-15	Privilege escalation using a JavaScript function's cloned parent
MFSa 2006-14	Privilege escalation via XBL.method.eval
MFSa 2006-13	Downloading executables with "Save Image As..."
MFSa 2006-12	Secure-site spoof (requires security warning dialog)
MFSa 2006-11	Crashes with evidence of memory corruption (rv:1.8)
MFSa 2006-10	JavaScript garbage-collection hazard audit
MFSa 2006-09	Cross-site JavaScript injection using event handlers
MFSa 2006-05	Localstore.rdf XML injection through XULDocument.persist()
MFSa 2006-03	Long document title causes startup denial of Service
MFSa 2006-01	JavaScript garbage-collection hazards

Fixed in Firefox 1.0.7

MFSa 2005-59	Command-line handling on Linux allows shell execution
MFSa 2005-58	Firefox 1.0.7 / Mozilla Suite 1.7.12 Vulnerability Fixes
MFSa 2005-57	IDN heap overrun using soft-hyphens

Fixed in Firefox 1.0.5/1.0.6

MFSa 2005-56	Code execution through shared function objects
MFSa 2005-55	XHTML node spoofing
MFSa 2005-54	Javascript prompt origin spoofing
MFSa 2005-53	Standalone applications can run arbitrary code through the browser

Abb. 11.1 Vulnerability-Liste des Mozilla-Projekts (Ausschnitt)

- Tauchen bestimmte Fehler an anderer Stelle erneut auf, nachdem sie behoben wurden?
- Tauchen bestimmte Fehler regelmäßig immer wieder auf?
- Gibt es Softwarekomponenten, die besonders anfällig für Sicherheitsprobleme sind?
- Bringen neue Features auch neue Sicherheitsprobleme?
- Wie verteilen sich die Sicherheitsprobleme auf die Komponenten?
- Wie schnell werden welche Fehlertypen behoben?
- Wie einfach ist es, bestimmte Sicherheitsfehler zu machen?
- Wie oft besteht der „Workaround“ einfach aus dem Abschalten von JavaScript?

Auf diese Art ließen sich auch Behauptungen wie in dem oben erwähnten Artikel zu Sicherheitsmängeln bei Firefox empirisch nachprüfen, z. B. ob das Extension-Konzept die Sicherheit tatsächlich verschlechtert hat.

An dieser Stelle beschränkt sich die Analyse jedoch auf qualitative Fragestellungen:

- Gibt es bestimmte Fehlertypen, und wenn ja, welche?
- Wo verschärft bzw. entschärft die Gesamt-Architektur die Kritikalität eines Fehlers?

- In welchen Komponenten tauchen Fehler auf?
- Gibt es systematisch auftretende Fehler, die einen Zusammenhang mit der Architektur aufweisen?
- Welche Sicherheitsmechanismen werden tatsächlich eingesetzt, welche könnten eingesetzt werden?
- Welche Sicherheitskonzepte sind anscheinend besonders kritisch?

Um diese Fragen zu beantworten, verwenden wir folgendes Vorgehensmodell:

- Aus den gemeldeten Sicherheitsproblemen werden einige ausgewählt, je nachdem, wie interessant bzw. unterschiedlich sie erscheinen. Für dieses Buch wurden die Fehler MFSA 2005-01 bis 2005-44 ausgewählt. Die neue Architektur von Firefox 2.0 wurde noch nicht berücksichtigt.
- Eine Liste von Fehlertypen wird erstellt.
- Jeder Fehlertyp wird in Bezug auf seine Funktion und seine Behebung untersucht.
- Fehlermuster werden ermittelt.
- Die Schwere des Fehlers in Bezug auf die Architektur wird abgeschätzt (das ist letztlich die Frage, wie gründlich der Fehler tatsächlich beseitigt werden kann).
- Lösungsmöglichkeiten werden diskutiert und erstellt.

Um es nochmals ganz deutlich zu sagen: Ziel dieser Fehleranalyse ist es, aus den Fehlern anderer zu lernen und die Erkenntnisse daraus auf die eigenen Projekte zu übertragen, nicht jedoch, eine Software oder ihre Entwickler anzuschwärzen.

11.4 Die Fehler im Einzelnen

11.4.1 MFSA 2005-44: Privilege escalation via non-DOM property overrides

Hier wird offensichtlich Code (JavaScript) von einer Autorität (hier heruntergeladener Page Content) erzeugt, aber von einer anderen (hier das privilegierte User-Interface, genannt Chrome) ausgeführt.

Die Fehlermeldung erklärt, dass der Code fälschlicherweise mit den Rechten des Aufrufers statt mit den Rechten des Code-Erzeugers ausgeführt wird. Zu dem Fehler werden zwei Reperaturmöglichkeiten genannt: Die erste Möglichkeit, hier Fix 1 genannt (vgl. Abb. 11.2), verhindert durch einen Automatismus, dass privilegierter Code überhaupt die geänderten anstatt der originalen Page-Inhalte aufrufen kann.

Die zweite Möglichkeit (Fix 2) besteht darin, dass beim Aufruf des platzierten Codes nicht die hohen Privilegien des GUI verwendet werden, sondern dass ein automatisches Rückführen auf die Privilegien des Contents stattfindet. Erreicht wird dies durch Erweiterungen im Security-Manager. Nach Lektüre von Kap. 5 über Java-Sicherheit kann man sich den Mechanismus dafür gut vorstellen: Beim

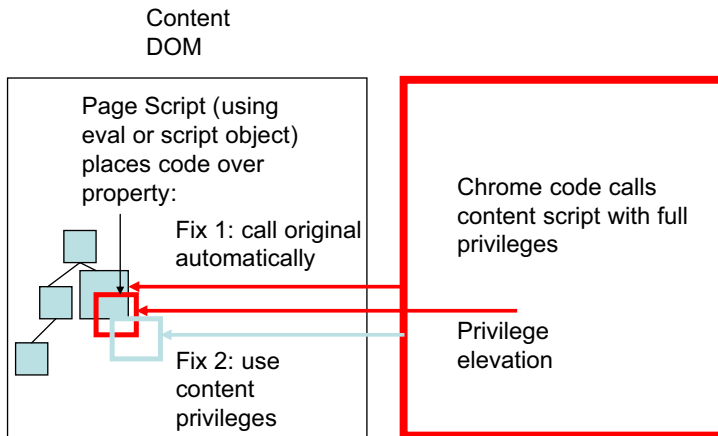


Abb. 11.2 Funktionsweise und Fixes zu MFSA 2005-44

Aufruf von Nicht-Chrome-Code aus dem GUI heraus wird durch den Access-Controller bzw. Security-Manager ein anderer Access-Control-Kontext dem Aufruf dynamisch mitgegeben, so wie auch ein Java-Server durch Änderung des Sicherheitskontextes einen User impersonifizieren kann, also den Server-Code mit den Privilegien des Users ausführen lassen kann.

Bleiben wir noch einen Moment bei den beiden vorgestellten Lösungsmöglichkeiten. Fix 1 benötigt eine Form von Gedächtnis: Wo kam der Code her? Dies erinnert an den Mechanismus „Tainted Variables“ bzw. Multi-Level-Security-Mechanismen, bei denen Daten mit Sicherheitstags ausgezeichnet werden. Interessanterweise besaß der originale Netscape-Browser einmal ein auf Tainting beruhendes Sicherheitskonzept, allerdings ging es nie in die Produktion. Dazu im Abschn. 11.6.6 mehr, wir halten jedoch hier bereits die Nützlichkeit einer Historie für Daten bzw. Code fest.

Implementieren ließe sich Fix 1 z. B. durch Wrapper-Klassen, die beim Einfügen neuer Werte diese automatisch einschließen und einen Aufruf je nach Quelle auf die Originalwerte umleiten. Schlecht wäre eine Implementation, die Entwickler von GUI-Routinen zwingt, selbständig daran zu denken, ihre Aufrufe eventuell umzuleiten – d. h. die Umleitung auf Originalwerte muss transparent erfolgen.

Fix 2 benötigt ebenfalls eine Form von Gedächtnis bezüglich der neu platzierten Werte. Der Automatismus kann dadurch die Umschaltung in einen geringeren Sicherheitskontext ohne Beteiligung der Entwickler bewerkstelligen.

Weitere Erklärungen zu diesem Fehler finden sich interessanterweise bei MFSA-2005-41. Dieser Fehler ist gewissermaßen der Urvater dieses Fehlertyps „Privilege Escalation“. Der Fehler 2005-44 stellt lediglich eine weitere Variante davon dar. Deutlich wird, dass weitere Security-Checks eingefügt werden mussten, da einige Missbrauchsmöglichkeiten bei der ersten Reparatur nicht erkannt wurden.

Welche Rolle spielt nun jeweils die Aktion des Nutzers (z. B. das Anklicken eines Links) in Bezug auf die Rechte des ausführenden Codes? Die Fehlermeldung

hat diesen Fehler als besonders kritisch eingestuft, da der User lediglich harmlose bzw. alltägliche Aktionen (z. B. Klicken auf einen Link) durchführen muss, um die Attacke zu ermöglichen. In Bezug auf die möglichen Folgen des Fehlers ist diese Einschätzung sicher korrekt. Uns interessiert jedoch vor allem, welche Architekturmerkmale diesen Fehler so kritisch werden lassen.

Fassen wir also zusammen, was den Fehler aus Sicht der Architektur so kritisch werden lässt:

- Es gibt privilegierte und nicht-privilegierte Principals im System.
- Die Kommunikation zwischen verschiedenen Principals findet über eine gemeinsam navigierbare, zunächst öffentliche Datenstruktur (Domain Object Model, DOM) statt.
- Code wird platziert und mit den Rechten des Aufrufers ausgeführt – also Discretionary Access Control.
- Die Herkunft von Code wird manuell – sprich von Entwicklern – verwaltet (kein Tainting).
- Der GUI-Code läuft immer privilegiert und ist nicht von expliziten Ermächtigungen von Seiten des Nutzers abhängig.

Dieses Architekturmuster bildet die Basis für eine Reihe bekannter Attacken, wie sie auch aus Betriebssystemen bekannt sind. Wieso darf z. B. kein Punkt für das „Current Directory“ im Pfad des Root/Administrators sein? Die Antwort ist einfach: Damit der Root/Administrator beim Navigieren des Dateisystems nicht aus Versehen Code der User aufruft und diesen mit vollen Privilegien ausführt. Ohne diese einfache Maßnahme werden sog. „Luring-Attacken“ möglich: Code wird absichtlich und öffentlich platziert, in der Hoffnung, dass er von einem privilegierten User bzw. in einem privilegierten Kontext ausgeführt wird.

Diese Art von Attacke ist seit Langem bekannt. Ersetzt man in dem geschilderten Szenario nun „Dateisystem“ durch „DOM“, so hat man das gerade diskutierte Mozilla-Fehlerbild. Verblüffend ist dabei, dass es offensichtlich nur kleiner Namensänderungen bedarf, um die Architektur des Fehlers unsichtbar zu machen. Die Mozilla-Programmierer mussten somit etwas erneut lernen, was in anderen Softwaregebieten schon längst bekannt war. Noch einmal: Um diesen Fehler möglich zu machen, müssen folgende Komponenten zusammenspielen:

- Code muss seine Rechte je nach Aufrufer erhalten (Identitätsbasierte Rechte bzw. discretionary access control).
- Der Ausführende ist sich nicht im Klaren über die Bedeutung des Codes oder über den Zeitpunkt des Aufrufs (siehe Problem des „confused deputy“ weiter unten).
- Es muss eine Möglichkeit bestehen, Code bereitzustellen, damit er von falschen Identitäten ausgeführt wird („Luring-Attacke“).

Muss es übrigens unbedingt Code sein, der platziert wurde? Keineswegs: Ersetzt man das eingeschleuste JavaScript durch File-Namen, erhält man die klassi-

sche „Confused-Deputy“-Angriffe, wie sie von Norm Hardy geschildert wurde: Der privilegierte Code wird mit falschen Eingaben gefüttert und greift auf die falschen Ressourcen mit vollen Privilegien zu. Womit wir bereits bei der nächsten Frage wären: Wie grundlegend wurde der Fehler behoben? Taucht er an anderen Stellen erneut auf? Schon die prinzipielle Möglichkeit der „Confused-Deputy“-Angriffe lässt auf weitere ähnliche Fehler schließen. Auch scheint es unwahrscheinlich, dass durch die zwei Fixes alle Möglichkeiten, Code fälschlicherweise privilegiert auszuführen, beseitigt wurden.

Es gibt noch weitere bedenkliche Architekturmerkmale: Eine gemeinsame, beliebig navigierbare Datenstruktur erlaubt außerdem Angriffen basierend auf Concurrent-Access-Problemen (Time-of-Check-to-Time-of-Use, TOCTOU) und verhindert die Isolation von bestimmten Gebieten: Die gesamte Datenstruktur wird öffentlich und ist damit nur mit hohem Aufwand zu schützen. Häufig – und der Fehlertext impliziert dies, indem er naiven Umgang von Chrome mit potenziell gefährlichen Werten erwähnt – beruht der Schutz auf der Einführung von IF-Abfragen, die den Zugriff auf bestimmte Zweige im DOM-Tree bzw. Dateisystem verweigern sollen. Dies ist jedoch kein Automatismus, sondern muss vom Entwickler jeweils neu bedacht werden (und ist damit fehlerträchtig). Wir werden weitere Fehlermeldungen sehen, bei denen genau diese mangelnde Abgrenzung z. B. auch zwischen Informationen aus verschiedenen Sites zu Sicherheitsproblemen führte. Fügen wir also zu den Voraussetzungen des Fehlers in der Architektur noch einen Punkt hinzu, nämlich die versuchte Isolation von Bereichen durch programmierte IF-Anweisungen.

11.4.2 MFSA 2005-43 „Wrapped“ JavaScript: URLs bypass security checks

Bei diesem Fehler könnte ein Angreifer Code einschleusen („script injection“), in der Hoffnung dass der Code später durch den JavaScript-Interpreter ausgeführt wird. Im Grunde genommen ist diese wieder eine Form der „Luring-Angriffe“. Eine Möglichkeit der Tarnung des Codes besteht im Verstecken in anderen Formaten. Dadurch werden die Security-Checks, die Code filtern sollen, umgangen, d. h. sie entdecken den Code nicht. Wenn in der Folge der User des Browsers den Code ausführt (z. B. indem er auf einen Link klickt) dann wird der Code mit den vollen Rechten des Users ausgeführt.

Dieser Fehler repräsentiert eine ganze Familie von Fehlern, die alle unter der Bezeichnung „script injection“ zusammengefasst werden können. Dabei werden JavaScript-URLs in Pseudo-Protokollen wie „view source:“ oder „jar:“ versteckt, oder es werden Icon-URLs im Software-Download-Dialog als JavaScript-URLs missbraucht (s. MFSA 2005-42). JavaScript-URLs tauchen auch in den Headern der „Tabs“ auf, als gebookmarkte Live-Feeds, als Attribute vieler HTML-Elemente oder als Inhalt privilegierter Seiten wie „about:“.

Ein Grund dafür, warum dieser Fehlertyp so schwer zu vermeiden ist, mag darin liegen, dass hier extrem flexible Mechanismen wie der URL-Protokollmechanismus verwendet werden. Spezielle URLs werden dabei schlicht dem richtigen Interpreter zur weiteren Verarbeitung übergeben. Das genügt aber keinesfalls dem „Default-is-Deny“-Prinzip, sondern ist ein völlig offener Mechanismus, der anschließend mit vielen Sicherheitschecks wieder notdürftig gebändigt werden muss. Ein weiterer Grund besteht in der Vielzahl von Stellen, an denen Code eingefügt werden kann. Nicht zuletzt ist die Grenze zwischen Daten und Code fließend: Interpretiert wird also, was sich interpretieren lässt, auch wenn dies gar nicht beabsichtigt ist.

Wie gefährlich ist nun dieser Fehler? Das hängt schlicht davon ab, mit welchen Privilegien der ausführende Interpreter läuft. Häufig scheint das der „privileged Mode“ bzw. Chrome-Mode zu sein, und damit erhält der eingeschleuste Code alle Rechte des Users.

Wie könnte der Fehler nun behoben werden? Der Text der Fehlermeldung lässt darauf schließen, dass das Einschleusen von Code durch das Einfügen von weiteren Checks unterbunden werden soll. Gibt es aber eine Garantie dafür, dass der verbesserte Filter den nächsten Fall unterbindet? Kaum, wenn wir den offenen Mechanismus zur Verwendung der JavaScript-URLs anschauen: Er stellt im Grunde genommen das Code-Äquivalent zur frei navigierbaren Datenstrukturen wie einem Dateisystem oder DOM dar: Alles ist grundsätzlich offen und muss im Nachhinein durch programmatische Checks gebändigt werden. Eine kritische Rolle scheint dabei auch die `eval()` Funktion in JavaScript zu spielen, mit deren Hilfe beliebiger Text dynamisch als Code interpretiert und ausgeführt werden kann.

11.4.3 MFSA 2005-33 JavaScript „lambda“ replace exposes memory contents

Diese Lücke wird durch einen Fehler im JavaScript-Interpreter selbst verursacht. Der Fehler ist deshalb so wichtig, weil er die Bedeutung des Type-Systems für die Sicherheit klar herausstellt. Er repräsentiert eine Fehlerfamilie, die Buffer-Overflows bzw. Heap-Overflows im C++-Code genauso umfasst wie Fehler bei der Implementation von JavaScript bzw. dessen nativen Objekten zum Zugriff auf Systemressourcen (siehe auch MFSA 2005-40, MFSA 2005-30).

Die JavaScript-Funktionalität, die es erlaubt, anonyme Funktionen als Returnwert zu verwenden, erlaubt durch einen Fehler auch den Zugriff auf den Speicher des Browsers selbst. Dort können Informationen anderer Sites gespeichert sein. Diese Informationen können ausgelesen und an fremde Sites gesendet werden. Voraussetzung für die Attacke ist ein Bruch des Daten-Confinements durch die Programmiersprache selbst, d. h. JavaScript erlaubt am Typesystem vorbei Zugriff auf den Speicher anderer Sicherheitszonen oder Bereiche.

Es ist wichtig festzuhalten, dass der Fehler bei korrekter Implementation von JavaScript nicht auftritt. Die Bereinigung des Fehlers garantiert allerdings nicht, dass

in Zukunft diese Attacken nicht mehr möglich sind, sie setzen aber weitere Fehler im Interpreter voraus. Die Bedeutung dieses Fehlers ist damit aus Sicht der Architektur des Browsers weitaus geringer, denn JavaScript schützt den Speicher automatisch gegen fremde, illegale Zugriffe. Die Sprache selbst bietet also einen Schutz und nicht der Programmierer, der mithilfe von IF-Abfragen versuchen müsste, Confinement zu erreichen. Der Fehler liegt also in der Implementation der Sprache. Inwieweit ist er nun ein Architekturfehler des Sprachlevels? Für C/C++ lässt sich dies klar beantworten: Letztlich haben C/C++ basierte Implementationen immer das Problem der Typumgehung und stellen deshalb ein Architekturproblem dar.

In der Folge werden wir keine Sicherheitsfehler durch Speicher-Korruption mehr besprechen, sondern verweisen auf [heise77956] und deren Erwähnung von KlokWerk. Angeblich wurden durch KlokWerks automatische Analysetools mehr als 655 Fehler in der Behandlung des Speichers nachgewiesen.

11.4.4 MFSA 2005-34 PLUGINSPAGE privileged javascript execution

Hier zeigt sich die Problematik von nicht-standardisierten Fehlermeldungen: Dieser Fehler ist fast völlig identisch mit dem erstgenannten Fehler des Non-DOM-Property-Overrides. Im Endeffekt ruft der GUI-Level (d. h. Chrome) ein heruntergeladenes Script in seinem Privileged-Mode auf. Hier sind es direkte DOM-Werte (Attribute wie PLUGINSPAGE), die JavaScript-URLs beinhalten. Beim Klick auf „Install Manually“ wird das Script geladen und ausgeführt. Eine Attacke hat folgende Voraussetzungen:

- einen Interpreter, der anscheinend immer zur Verfügung steht, sobald ein Teil eines Scripts gefunden wird (ambient authority). Dieser allgemeine Mechanismus verletzt hier u. a. das Prinzip, dass eine Installation von Software nie durch Code durchgeführt werden sollte, der mit dem Paket kommt. Installationen haben rein deskriptiv zu funktionieren (siehe dazu das Prinzip von MS-Singularity oder OSGI);
- eine „default-is-allow“ Logik, die grundsätzlich immer den Einsatz von Script erlaubt und programmatisch gebremst werden muss;
- ein User-Interface, das dem User keine Chance lässt, das Script zu erkennen und die Ausführung zu unterbinden;
- einen Interpreter, der im Privileged Mode läuft.

Es stellt sich die Frage, wie viele Instanzen dieses immer gleichen Fehlers noch entstehen bzw. gefunden werden. Wie könnte eine systematische Abwehrmaßnahme aussehen, die z. B. dafür sorgt, dass es auch wenn ein Script ausgeführt wird keine Rechte besitzt, weil es diese explizit übertragen bekommen müsste? Wäre es möglich, einen Modus zu erzeugen, in dem kein Interpreter zur Verfügung steht (weil keine Referenz darauf verfügbar ist)? Sollte es verschiedene Interpreter mit verschiedenen Privilegien geben, die auf gemeinsamen oder auch getrennten Daten arbeiten?

Grundsätzlich müsste bei der Lösung des Problems die Frage beantwortet werden, ob das Script selbst das Sicherheitsproblem darstellt oder eher der Privilegienlevel des ausführenden Interpreters. Im ersten Fall stellen sich weitere Fragen wie: Wann ist Script ungefährlich und wann nicht? Diese Frage wird sich niemals in voller Allgemeinheit durch einen automatischen Mechanismus beantworten lassen. Im zweiten Fall wird man nach Modellen suchen müssen, die Interpreter mit weniger Rechten ausstatten.

11.4.5 MFSA 2005-36: Cross-site scripting through global scope pollution

Bei diesem Fehler entsteht beim Browsen über verschiedene Sites der Effekt, dass Code einer zuerst besuchten Site sich auf Elemente später besuchter Sites auswirken kann. Eine Page einer neuen Site sollte aber natürlich keinerlei Beeinflussung durch andere Sites ausgesetzt sein.

Zunächst scheint hier ein Cross-Site-Scripting-Problem vorzuliegen, allerdings eines, das direkt im Client stattfindet und nicht über einen Server, der seinen Input nicht richtig validiert. Technisch gesehen ist das korrekt: Fremdes Script gelangt in einen anderen Page-Kontext. Cross-Site-Scripting wird normalerweise jedoch meist als eine Form fehlerhafter Input-Validierung verstanden, als deren Folge fremder Input weiteren Teilnehmern serviert wird. SQL-Injection ist ebenfalls ein Fehler der Input-Validierung, der sich jedoch gegen die eigene Datenbank richtet.

Hier findet jedoch gar keine fehlerhafte bzw. unterlassene Validierung statt: Stattdessen liegt das Problem im Konzept des „Global Scope“ begraben: Offensichtlich kennt das Dokumentenmodell globale Stellen, die – wahrscheinlich hierarchisch oberhalb der Pages angesiedelt – für mehrere Pages potenziell gültig sind. Beim Besuch einer neuen Page muss daher programmatisch eine Bereinigung kritischer Werte oder Funktionen stattfinden.

Entscheidend an diesem Fehler ist die „Verschmutzung“ (pollution) des Global Scope, als dessen Folge Input einer fremden Site in die Repräsentation einer anderen gelangen kann. Es handelt sich also um einen Fehler im „Confinement“ – in der Absperrung eigentlich getrennter Zonen. Auffällig ist die Verwendung des Begriffs Global Scope in diesem Zusammenhang – wie will man Bereiche voneinander trennen, wenn es einen Global Scope gibt? Dies erinnert an globale Variablen und deren Sicherheitsprobleme: Sie sind entweder Read-Only-Konstanten oder sie stellen ein gravierendes Sicherheitsproblem dar, da Kontrolle, Validierung und Zuständigkeit unklar sind.

Damit scheint es sich hier um ein Architekturproblem zu handeln, das sich zwar kurzfristig programmatisch durch das Einbauen eines weiteren IF lösen lässt, aber keine fundamentale Trennung der Bereiche zulässt. Zur Attacke sind also folgende Voraussetzungen nötig:

- ein globaler, von verschiedenen Seiten zugänglicher Bereich,
- das Fehlen von Confinement,
- ein globaler Interpreter.

Das Unbehagen angesichts des identifizierten Architekturproblems lässt keinen rechten Glauben an die grundlegende Wirksamkeit der Fixes aufkommen. Und so ist es auch kein Wunder, dass der Fehler wiederum eine ganze Klasse von Fehlern repräsentiert: Mangelnde Isolation zwischen Site-Contexten im Browser. Das könnte man auch als „Intra-DOM-Beinflussung“ bezeichnen im Gegensatz zum ersten Fehlermuster, bei dem Content das Browser-GUI beeinflusst.

Ein weiteres Beispiel für diesen Fehlertyp stellt MFSA 2005-13 dar: Das Script einer vorherigen Page kennt eine wichtige Variable einer bekannten Site und registriert zu dieser Variablen Events, Setter etc. So können z. B. Popup-Windows einer neuen Site vom Code einer vorherigen übernommen werden. Interessant ist in diesem Zusammenhang auch der vorgeschlagene Workaround: Für wichtige Sites soll ein neues Fenster geöffnet werden (eine Standard-Empfehlung im Bereich der E-Banking-Lösungen). Offensichtlich bietet das Konzept „Separates Fenster“ eine bessere Form der Isolierung von Sites (confinement) als dies innerhalb eines Fensters möglich ist. Es stellt sich also die Frage, wie die Konzepte „Window“ und „Site“ miteinander gekoppelt sind. Der in Kap. 12 vorgestellte DARPA-Browser setzt hier ganz harte Trennungen durch.

11.4.6 MFSA 2005-37: Code execution through JavaScript: Favicons

Dieser Fehler entspricht fast genau einer bereits behandelten Fehlerklasse, nämlich: DOM mit JavaScript-URL wird mit falschen Privilegien ausgeführt (siehe MFSA 2005-34). Wenn ein sogenanntes *Favicon* (eine kleine Bilddatei, die links neben der URL in der Adresszeile des Browsers angezeigt wird) programmatisch dem Document-Object-Model (DOM) hinzugefügt wird und der Link zeigt auf JavaScript, dann wird das Script im Kontext des Users und nicht dem Kontext der Site ausgeführt.

Es scheint, als handele es sich hier um einen typischen TOC2TOU-Fehler. Zur Erinnerung hier noch einmal die Schritte, wie es zu einem solchen Fehler kommen kann:

- Ein Objekt X wird zum Zeitpunkt T auf Brüche der Verletzungen geprüft und als harmlos erkannt.
- X wird geändert zu Y.
- Y wird zu einem Zeitpunkt T+1. Damit ist der ursprüngliche Check natürlich sinnlos geworden.

Dem hier vorliegenden TOC2TOU-Fehler liegt aber eigentlich noch etwas Fundamentaleres zugrunde, nämlich sogenannte *Dirty Assignments*, d. h. die nachträgliche Veränderung von Daten durch Zuweisung. In diesem Fall geht es nicht nur um Daten, sondern sogar um Code.

Es ist wichtig, an dieser Stelle klar herauszuarbeiten, dass es sich hier nicht um einen Validierungsfehler handelt: Selbst wenn der Browser alle Seiten, die geladen

werden, genau überprüft, bevor ein enthaltenes Script ausgeführt (oder unterdrückt) wird, kann er gar nicht alle Möglichkeiten entdecken, wenn das Script später dynamisch (also durch ein anderes Script) hinzugefügt werden kann. Das ist aber heute im Rahmen von AJAX-Lösungen der Normalfall.

Die Behebung dieses Fehlers muss also ganz klar auf der Reduktion der Privilegien des Interpreters beruhen. Zur Attacke bedarf es damit einer privilegierten Interpreter-Instanz, die nur programmatisch in ihren Effekten eingeschränkt wird.

11.4.7 MFSA 2005-38 Search plug-in cross-site scripting

Mit diesem Sicherheitsproblem gelangen wir in den Bereich der Extensions von Mozilla-Firefox-Software – ein sehr populäres wie auch gefährliches Feature heutiger Applikationen. Der Fehler besteht aus zwei unabhängigen Teilen, die beide behandelt werden sollen:

- Ein installiertes Search-Plug-In erhält beliebigen Zugriff auf Site-Daten und Funktionen.
- Search-Plug-Ins können darüber hinaus bei der Installation andere Search-Engines verdrängen (überschreiben), indem der Save-Dialog den echten File-Namen wegen Überlänge des Pfades nicht mehr anzeigt. Dann verwendet der User ein modifiziertes Search-Plug-In als „trojanisches Pferd“, ohne es zu merken.

Beim ersten Punkt liegt wieder ein Confinement-Problem vor: Offensichtlich haben Plug-Ins vollen Zugriff auf das DOM des Browsers und können damit Script injizieren. Dieses wird dann anschließend im Privilege-Level der jeweiligen Page ausgeführt und kann z. B. dazu benutzt werden, Cookies dieser Site zu stehlen. Das Confinement-Problem ist jedoch nicht innerhalb des DOM angesiedelt, sondern zwischen Extensions und DOM. Eine Möglichkeit wäre, das DOM einer Extension nicht direkt anzubieten, sondern nur über eingeschränkte Capabilities (d. h. man erstellt einen Wrapper um das echte DOM bzw. dessen Elemente). Diese könnten via Inversion-of-Control-Architektur an das Plug-In übergeben werden.

Der richtige Weg wäre grundsätzlich, Basissoftware und Extensions in unterschiedliche Adressbereiche auszulagern (wie z. B. für zukünftige Versionen des Internet-Explorer vorgesehen), oder eine strenge Objekt-Capability-Politik wie im Falle von MS-Singularity durchzusetzen. In beiden Fällen wäre der Zugriff von Extensions auf das DOM sehr gut innerhalb der Softwarearchitektur kontrollierbar. Das Lösungsmuster ähnelt dem für die Separierung von Betriebssystem-Kernel und Treibern.

Es handelt sich hier also um ein wirkliches Architekturproblem im Browser. Das wird auch sichtbar an der vorgeschlagenen, radikalen Lösung: JavaScript-URLs sind nicht mehr als Search-Aktionen erlaubt. Eine Lösung durch Reduktion der Page-Privilegien ist nicht machbar, da es sich nicht wirklich um ein Privilegienproblem der Page handelt. Es soll ja lediglich durch eine Extension kein Code eingeschleust mehr werden können. Angesichts der Architekturproblematik dürfte der gleiche Fehler auch noch bei vielen anderen Extensions auftreten können.

Der zweite Punkt steht stellvertretend für ein ganzes Bündel von Verletzungen, bei denen Betriebssystemressourcen wie Files, Directories oder Symbolische Links missbraucht werden. Die Angriffe laufen alle nach einem ähnlichen Muster ab:

- Privilegierter Code erhält einen geschickt präparierten Text von einer externen Site (Spoofing).
- Der Text wird dem User schlecht verstehbar präsentiert.
- Der User wird zu einer Bestätigungs-Aktion aufgefordert.
- Der Privilegierte Code setzt den Text in Objekte um (öffnet ein File gleichen Namens) und führt eine falsche Handlung durch.
- Gerne sind globale Verzeichnisse wie /tmp dabei beteiligt, um weitere User attackieren zu können.

Der Fehler besteht letztlich aus zwei entscheidenden Teilen: Einmal der Verwirrung des Users durch das GUI bzw. den Input, und zum anderen die Art und Weise, wie die Applikation mit OS-Ressourcen umgeht.

Beginnen wir mit der Verwirrungstaktik (Spoofing). Sie beruht darauf, potenziell kritischen Text so zu verpacken, dass der User die kritischen Teile nicht bemerkt. Gerne werden dazu in URLs lange Zeichenketten vor dem eigentlichen Hostnamen eingefügt (z. B. user1234454635635673@host). Oder, wie im Falle des Search-Plugin-Überschreibens, es wird ein sehr langer Hostname vor dem eigentlichen File-Namen gesetzt, sodass dieser in der Dialogbox gar nicht mehr sichtbar ist. (siehe auch MFSA 2005-17).

Offensichtlich weiß die Software aber, welches File überschrieben werden soll, denn sie tut es ja anschließend. Sie vergisst nur leider, diese Tatsache dem User – aufgeteilt nach Host-Name und File-Name – mitzuteilen. Das bloße Anzeigen der kompletten URL ist ein klarer Designfehler des GUI. Aber ist das der einzige Fehler des GUI in Bezug auf die Security? Die im GUI vorgenommene Umsetzung eines Namens in eine Ressource ist ein extrem kritischer Vorgang, der gerne für Attacken vom „Confused-Deputy“-Typ verwendet wird.

Kernproblem der GUI-Problematik ist hier, dass eine Separation von Designation (Path) und Authority (die Fähigkeit, ein File zu schreiben) vorliegt. Wie wäre es, wenn der User durch eine Geste das zu überschreibende Search-Plug-In selbst auswählen und zuordnen müsste, oder einen Platz für ein neues Plug-In selbst wählen müsste? Dann würde ihm das beabsichtigte Überschreiben eines existierenden Plug-Ins sofort klar. Lösungsarchitekturen für dieses Problem werden in Kap. 12 und 13 näher beschrieben.

Der zweite Teil der Fehlerproblematik, also der Umgang mit Ressourcen des Betriebssystems wie C:\tmp wird im Abschn. 11.6.3 näher betrachtet.

11.4.8 MFSA 2005-31/39 Arbitrary code execution from Firefox sidebar panel I + II

Hier wurde ein neues GUI-Feature (sidebar panel) eingeführt und erlaubte anschließend unprivilegiertem Page-Script das Öffnen privilegierter Pages (z. B.

about:config) sowie das Einfügen von Code, der dann natürlich ebenfalls privilegiert ausgeführt wird. Offensichtlich fehlt hier eine Prüfung auf eine User-Interaktion (eine Erlaubnis), um eine privilegierte Page zu laden. Auch scheint es Unklarheiten zu geben, wann und wie unprivilegierter Code privilegierten Code aufrufen kann.

Das Bedenklichste daran ist jedoch, dass diese Probleme – die doch sicherlich für bisherige Versionen der Software zumindest größtenteils gelöst sein müssten – durch den Einbau eines neuen GUI-Elementes wieder aufgetreten sind. Das wirft Fragen bezüglich der Sicherheits-Architektur auf. Bezeichnenderweise bringt ein weiteres neues Feature wie „Tabbed Browsing“ erneut die Gefahr der Ausführung von gefährlichem Code im privilegierten Modus (MFSA 2005-26)

So stellt sich die grundsätzliche Frage, wie resistent die verwendeten Sicherheitsmuster angesichts von funktionalen Änderungen in der Applikation sind.

11.4.9 MFSA 2005-28: Unsafe /tmp/plugtmp directory exploitable to erase user's files

Dieser Fehler fällt in die Fehlergruppe „Unsichere Verwendung von globalen Files, Verzeichnissen und Links“. In diesem Fall erzeugt Mozilla ein temporäres Verzeichnis für Plug-Ins, das gelöscht wird, wenn Mozilla geschlossen wird. Da der Name des Verzeichnisses bekannt ist, kann ein Angreifer einen symbolischen Link bereits vorher anlegen und dabei auf ein Verzeichnis eines bestimmten Users zeigen. Wenn dieser User Mozilla schließt, wird sein Verzeichnis gelöscht.

Ähnlich gelagert ist der Fehler MFSA 2005-21: Overwrite arbitrary files downloading .lnk twice: Schafft es eine Site, dass ein User ein .link File zweimalig herunterlädt, dann kann die Referenz im ersten File dazu benutzt werden, ein beliebiges File mit zu überschreiben. Denn beim zweiten Abspeichern des .link File wird nicht der Link überschrieben, sondern das zweite File wird dorthin geschrieben, wohin der erste symbolische Link gezeigt hat. Dieses Problem ist keineswegs Browser-spezifisch – es stellt sich daher die Frage, ob ein System überhaupt den Download von Symbolic-Link-Files erlauben sollte. Damit die Attacke funktionieren kann, müssen folgende Punkte gegeben sein:

- Der Browser muss die Rechte des Users während der Ausführung erhalten (z. B. via ACLs).
- Der Browser versucht, aus einem String (Namen) ein Objekt zu erzeugen. Dabei prüft er aber nicht, ob das Objekt schon existiert.
- Ein böartiger User kann das gleiche Objekt vorher global verfügbar anlegen. Über die Abbildung des Namens auf das Objekt (Designation vs. Authority) erhält der Browser dennoch Zugriff.

Die Gefährlichkeit dieser Attacke wird letztlich nur durch den ersten Punkt bestimmt. Erhält der Browser nämlich nicht alle Rechte des Users, dann wird der symbolische Link zwar ebenfalls ausgeführt, der Browser kann aber das Zielver-

zeichnis nicht löschen, da er keine Rechte dafür besitzt. Ansonsten ist die Attacke eine klassische Form der Symlink-Attacken auf Unix-Systemen, die ausnutzten, dass viele Programme nicht zwischen dem Erzeugen und dem Verwenden von Ressourcen unterscheiden.

An dieser Stelle wird deutlich, dass es meist mehrere Ursachen für eine Sicherheitsverletzung gibt. Hier sind es global verfügbare Verzeichnisse und Applikationen, die Ressourcen nicht prüfen bzw. ohne Prüfung Strings in Ressourcen umsetzen. Entscheidend ist jedoch, dass letztlich die Ressourcenrechte das Ausmaß des Schadens bestimmen. Ähnliche Ergebnisse sehen wir in Kap. 12 zum DARPA-Browser: Die dort entdeckten Fehler hatten oft keine oder nur eine sehr eingeschränkte Wirkung durch die fehlenden Rechte des Codes. POLA ist also tatsächlich auch schadensbegrenzend wirksam, und trägt nicht nur zur Schadensvermeidung bei.

Grundsätzlich lassen sich diese Fehler beim Ressourcenzugriff nur auf der Basis von POLA-Systemen lösen. Übergangsweise sollten alle Funktionen, die mit externen Ressourcen umgehen, zentral gehalten und nicht von Extensions einzeln implementiert werden. Gut wäre es, wenn die Extensions lediglich geöffnete Streams im Sinne von Inversion-of-Control-Architekturen erhalten würden und das eigentliche Low-Level-Interface zum Anlegen/Öffnen/Löschen von Ressourcen entweder zentral in der Applikationsbasis oder noch besser in einem externen Modul verwaltet wird.

11.4.10 MFSA 2005-25: Image drag and drop executable spoofing

In diesem Fall werden Bilder per „Drag and Drop“ vom Web auf dem Desktop gespeichert. Unter Windows ist es jedoch möglich, Kombinationen aus Bildern und ausführbarem Code zu erzeugen. Bei einer .exe Endung werden diese korrekt, d. h. mit einem Icon für Executables auf dem Desktop angezeigt. Da der User jedoch im Glauben ist, ein Bild abgespeichert zu haben, ist die Gefahr groß, dass er trotz dieser Endung auf das vermeintliche Bild doppelklickt und den Code ausführt.

Offenbar ist dies nicht nur ein Mozilla-Firefox-Fehler, sondern ein problematisches Verhalten des Betriebssystems insgesamt, und zwar in mehrfacher Hinsicht. Damit diese Attacke funktioniert, muss nämlich folgendes gegeben sein:

Eine einfache Geste des Users (Doppelklick) besitzt kontextabhängige Bedeutung – Öffnen von Daten mittels einer Applikation bzw. Starten eines Files als Applikation.

Der Wille des Users, ein Bild anzusehen, wird vom Betriebssystem ignoriert. Die eigentliche Absicht des Users „Öffne das Bild zur Ansicht“, der eine gewisse Read-Only-Semantik besitzt, wird stillschweigend vom Betriebssystem als Befehl „Starte Applikation“ umgesetzt. Das gleiche Problem tritt auch bei Mail-Anhän-

gen auf und stellt einen klaren Fehler im GUI-Design dar, der sich an dieser Stelle sicherheitskritisch auswirkt.

Beim Starten der Applikation werden sämtliche Rechte des Users automatisch an die Applikation übertragen – eine Folge der Access-Control-Listen, mit denen Ressourcen gesichert werden.

Verschärft wird die Situation durch die defaultmäßige Unterdrückung von Standard-File-Endungen unter Windows.

Offensichtlich ist eine Fehlerbehebung schwierig. Mozilla könnte die Semantik des Web („Das ist ein Bild“) zwangsweise umsetzen in eine Bild-Endung für das File. Das bedeutet eine Änderung des Dateinamens. Oder es bleibt bei einer Warnung für den User, keine unbekannten Files zu laden sowie keine Endungen zu unterdrücken, und Mozilla führt weiterhin den eingebauten Cast von Web-Image auf Dos-Executable durch.

Schnell stellt sich heraus, dass es der dritte Punkt – die Übertragung der Userrechte an die Applikation – ist, der die wirkliche Gefahr darstellt. Ohne die Verfügbarkeit der Autorität des Users für das gestartete Programm (Stichwort Ambient Authority) hätte die Malware wenige Möglichkeiten, Schaden anzurichten.

Für die Betriebssystemsicherheit wären einige weitere sicherheitskritische Fragen zu klären: Sind Files nur unstrukturierte Container von Daten oder stellen sie für sämtliche Systemfunktionen bekannte Typen dar? Funktionieren sie im Sinne von Objekten mit einem typegebundenen Interface oder allgemein als Daten? Und wie wird der Typ vermerkt (innerhalb des Containers oder durch die Endung)? Eine interessante Lösungsmöglichkeit besteht darin, ein File nur dann als Applikation und damit als startbar zu betrachten, wenn es in einem bestimmten, vom System dafür vorgesehenen Bereich im File-System liegt. Der Transfer in diesen Bereich kann nur mit Einverständnis (und damit Kenntnis) des Users erfolgen. Alle anderen Doppelklicks bewirken die Semantik „Öffnen zum Anschauen mit Applikation“ (siehe auch Kap. 13).

11.4.11 MFSA 2005-24: http auth prompt tab spoofing

Der Browser erlaubt über das GUI-Feature der „tabs“, mehrere Seiten parallel anzuzeigen. Öffnet eine der Seiten ein Authorisierungsfenster, fällt es dem User eventuell schwer, das Fenster der richtigen Seite zuzuordnen. Der Hinweis zur Vermeidung des Problems besteht darin, dass dem User empfohlen wird, keine vertrauenswürdigen bzw. nicht vertrauenswürdigen Sites in einer Browser-Session gleichzeitig zu verwenden.

Diese Empfehlung ist übrigens die gleiche wie im Falle der Cross-Site-Übergriffe durch einen globalen Bereich im DOM weiter oben und deutet wieder auf Isolationsprobleme hin, diesmal jedoch zwischen Pages, Sites und GUI. Der Fehler steht für die letzte große Fehlerkategorie, die wir hier behandeln wollen: Das Fehlen eines „Trusted-Path“ zwischen GUI und Pages. Dies führt dazu, dass User

bewusst in die Irre geführt werden können. Weitere Fehler dieser Kategorie sind bei Mozilla wie folgt aufgetaucht: MFSA 2005-16 (Verstecken von hoch-privilegierten Download- bzw. Security-Dialogen durch überlappende Fenster mit harmlosem Inhalt) und MFSA 2005-14 (Der SSL-Sicherheitsindikator für „sichere Site“ wird gefälscht). Von der letzteren Form existieren erneut mehrere Varianten. Besonders kritisch wird es, wenn der Benutzer komplett durch ein Script ersetzt werden kann: MFSA 2005-07 zeigt, dass script-generierte Events nicht von echten User-Klicks unterschieden werden konnten. Damit kann ein Script einen Download automatisch und ohne User-Intervention starten. Des Weiteren erwähnt MFSA 2005-05 die Möglichkeit, dass Pages, die „getabbed“ sind, aber im Hintergrund liegen, dennoch durch das Stehlen des Input-Focus oder durch das Starten von Dialogen, die der User anderen Pages zuordnet, Daten stehlen können.

Der extremste Fall von Spoofing wäre wohl der, eine Dummy-Version des echten Browsers zu starten, die der User als solche nicht erkennen kann – mit fatalen Folgen für die Sicherheit von Credentials, die der User z. B. beim Besuch von E-Banking-Sites in den gespoofen Browser eingibt.

Natürlich könnte man alle diese Fälle schlicht unter dem Begriff „Spoofing“ einordnen, es gibt jedoch einen fundamentalen Unterschied zu Fällen wie MFSA 2005-23 (long and confusing download string) und MFSA 2005-22 (Download file with hidden dangerous extension): In diesen Fällen funktioniert der Mechanismus des GUI bzw. seines Inhalts korrekt und der User lässt sich auf semantischem Wege täuschen. Bei den hier vorliegenden „Trusted-Path“-Fehlern betrügt das GUI den User hingegen, z. B. indem es den Authentisierungsdialog einer falschen Page zuordnet oder das SSL-Schloss ohne Grund anzeigt.

Für die Frage der Behandlung von Kommunikationswegen durch den Browser hat der DARPA-Browser (vgl. Kap. 12) übrigens eine interessante Antwort: Dort sind sämtliche Funktionen für das Öffnen/Schließen/Anzeigen von Kommunikationsverbindungen aus dem Bereich des Renderers in einen separaten Bereich des Browsers verbannt worden. Der Renderer dort ist in seinen Möglichkeiten, das GUI zu beeinflussen, extrem eingeschränkt und erhält zudem keine URLs, sondern lediglich geöffnete http-Verbindungen. Dies verringert die Gefahr von Spoofing enorm. Übrigens spielt die Erfassung „echten“ User-Inputs als Beweis für eine reale Person am Rechner eine große Rolle innerhalb der Trusted Computing Group, die Sicherheitsmodule und Software für PCs definiert.

11.5 Klassifikation der Schwachstellen im Mozilla-Firefox-Browser

In diesem Abschnitt wollen wir eine Klassifikation der diskutierten Schwachstellen versuchen. Zunächst erstellen wir eine Übersicht, die Schwachstellen auf bekannte Angriffstypen abbildet, dann betrachten wir konkret die Fehler in der Softwareentwicklung, die zu diesen Fehlern geführt haben.

11.5.1 Übersicht

Nr.	Sicherheitsproblem	Level	Script-Beteiligung
2005-24	Spoofing	Application / User-Interface	ja
2005-25	Spoofing	User / OS	nein
2005-28	Unauthorized Access	Application	nein
2005-31	Privilege Escalation	Application	ja
2005-33	Unauthorized Access	Language / JavaScript	ja
2005-34	Privilege Escalation	Application / JavaScript-Interpreter	ja
2005-36	Unauthorized Access	Application	ja
2005-37	Privilege Escalation	Application / JavaScript-Interpreter	ja
2005-38	Unauthorized Access	Application	ja
2004-43	Script Injection	Application / JavaScript-Interpreter	ja
2005-44	Privilege Escalation	Application / User-Interface	ja

Unsere Übersicht zeigt, dass die beobachteten Schwachstellen auf ganz wenige Grundtypen zurückführbar sind, jedoch auf allen Ebenen – vom Betriebssystem bis hin zum User – auftreten. Woher kommen die Schwachstellen? Welche Fehler-typen gibt es? Dieser Frage wenden wir uns jetzt zu.

11.5.2 Fehlermuster

Die von uns untersuchten Sicherheitsprobleme zeigen auf ganz wenige Fehler-muster:

- Isolationsfehler (Globale Bereiche/Navigation, IntraDOM, Non-DOM, Extensions),
- Privilegienfehler (DOM vs. GUI, Non-DOM vs. DOM, Extensions vs. GUI, etc.),
- Confused-Deputy-Problem (String in Ressourcen verwandeln oder Injected-Script ausführen),
- Trusted-Path-Probleme (GUI-Connection-Mismatch, Page-Dialog-Mismatch),
- Memory-Corruption und Language-Implementation (Buffer/Heap-Overflows, JavaScript-Interpreter-Fehler, native Objekte falsch implementiert),
- Ambient Authority (Keine Einschränkung von Rechten durch OS, Sprache oder Applikationsarchitektur),
- Keine Anwendung des Default-is-Deny-Prinzips (Flexibilität steht über Sicherheit) bzw. subtraktive Sicherheit.

Eine erstaunlich große Anzahl der Fehler kann durch den Patch „Abschalten von JavaScript“ behoben werden, wie unsere obige Übersicht zeigt. Offensichtlich ist die Integration von einem globalen DOM, Site-Security, flexiblem GUI und einem dynamischen Interpreter doch recht schwierig.

Obwohl es sich in jedem der Fälle um ein Sicherheitsproblem handelt, entsteht die Gefährlichkeit in vielen Fällen erst aus der Tatsache, dass während der Ausführung des fremden Codes „ambient authority“ herrscht, d.h. alle Rechte des Users uneingeschränkt dem Code zur Verfügung stehen. Die meisten, wenn nicht alle dieser Sicherheitsprobleme sind zudem auf dem Architekturlevel angesiedelt: Sie verschwinden nicht durch einen Fix. So lässt sich auch die bange Frage des Managements: „Haben wir jetzt alle Stellen mit diesem Fehler gefunden?“ leider nur mit einem „keine Ahnung“ beantworten.

Einer der Gründe für die Unsicherheiten im Framework ist wohl die Betonung der Flexibilität und Mächtigkeit der Features und Mechanismen im Mozilla-Umfeld – marketingtechnisch durchaus richtig. So wundert es nicht, dass viele neue Features teils alte und bekannte Sicherheitsprobleme wieder auftreten lassen (z.B. Privilegienprobleme). Wie unsere Fehleranalyse zeigt, scheint das Grundkonzept von Sicherheit eher subtraktiv als additiv zu sein: Man mache zunächst einen Mechanismus extrem flexibel und mächtig und versuche ihn dann durch Einschränkungen wieder zu zähmen.

Es zeigt sich auch deutlich, dass die Ebenen Betriebssystem, Sprache und Applikationsarchitektur sehr abhängig voneinander sind und sich gegenseitig beeinflussen. Oft entscheidet sich die Gefährlichkeit eines Fehlers erst durch das Umfeld von Betriebssystem oder Sprache. Im letzten Teil der Browseranalyse sollen deshalb verschiedene Sicherheitskonzepte und ihre Entwicklung im Zusammenhang mit der Browserarchitektur untersucht werden.

11.6 Browserarchitektur und Sicherheitskonzepte

Das Verständnis für die sicherheitsrelevanten Teile eines Web-Browsers ist nur langsam entstanden und lässt sich z. B. in Kap. 20 von [Flan] sehr schön nachlesen.

Die älteren Editionen des JavaScript Guides behandeln im Wesentlichen die aus heutiger Sicht bereits sehr alten Versionen von Netscape 2.0. Überraschenderweise findet man darin jedoch sehr viele Fehler erwähnt, die selbst nach umfangreichen Sicherheitsmaßnahmen auch in den neuesten Browserversionen immer wieder aufzutreten scheinen. Wenden wir uns zunächst aber einigen einfachen Fragen mit vielleicht überraschenden Antworten zu:

- Was ist eigentlich ein Browser?
- Wie funktioniert JavaScript im Browser?
- Welche Konzepte für Browser-Security gibt es?
- Wie könnte ein alternatives Konzept aussehen?

11.6.1 Was ist ein Browser?

Eine naive Antwort auf diese scheinbar harmlose Frage könnte lauten: „Ein Browser ist etwas, mit dem im Web Inhalte angeschaut und zwischen Inhalten navigiert

werden kann – also eine Art Viewer für Web-Inhalte. Das war zweifellos richtig zu Beginn der Web-Browser-Entwicklung (zu einer kurzen Übersicht über die historische Entwicklung von Web-Browsern siehe z. B. [MS]).

Sehr schnell wurden jedoch aus den rein passiven „Viewern“ generische Werkzeuge zum Download von beliebigen Datenstrukturen. Die Browser bekamen eine Erweiterungsarchitektur (Plug-Ins), mit denen nachträglich beliebige Interpreter aufgerüstet werden konnten. Gleichzeitig wurde die Hauptsprache HTML durch Script- und Objekt-Tags erweitert und durch die Einführung der Sprache JavaScript wurde der Browser auch zum Interpreter für heruntergeladenen Code. Häufig dient dieser Code dazu, lokale Dokumente auf Client-Seite zu manipulieren, weiterzuverarbeiten und das Ergebnis im Browser darzustellen. Die dazu verwendete Programmierschnittstelle ist das DOM. Das konzeptionelle Modell, das Softwareentwickler von ihrem Gegenstand haben, scheint da oft nicht mitzuwachsen, obwohl die Software doch selbst durch die neuen Fähigkeiten erweitert wird. Welche Rolle das konzeptionelle Modell der Entwickler spielen kann, möchten wir an einem realen Beispiel demonstrieren:

Eine Firma entscheidet sich dazu, den Browser im Intranet auch als Werkzeug zur Systemverwaltung zu benutzen. Es wird deshalb ein spezielles Plug-In geschrieben sowie eine Sprache definiert, mit deren Hilfe Software auf Arbeitsstationen konfiguriert bzw. erneuert werden kann. Das Ganze wird im Intranet auf die nötige Sicherheit geprüft, ein neuer MIME-Type für die speziellen Inhalte wird definiert und die Lösung auf die Arbeitsstationen verteilt. Das Ganze stellt eine große Erleichterung für die Systemverwaltung da, der Browser ohnehin auf allen Stationen vorhanden ist und somit Updates einfach ausgerollt werden können. Hier wird eine zentrale (und potenziell sicherheitskritische) Eigenschaft des Browsers gezielt ausgenutzt: Er kann, wie gerade gesehen, lokale Ressourcen manipulieren.

Leider wurde aber vergessen, die hausinternen Firewallverwalter von dem neuen MIME-Type zu unterrichten, denn der muss jetzt natürlich geblockt werden, sollte er von außen aus dem Internet kommen – ansonsten kann jemand vom Internet aus administrative Funktionen durchführen, falls ein User auf die jeweilige Site gelockt würde. Da ist er wieder: Der Browser als „harmloser Viewer“ auf das Web.

Im Laufe der Zeit erkennt man die Gefahr hinter den konzeptionellen Modellen der Entwickler etwas besser. Ein weiteres klassisches Beispiel ist die Fehleinschätzung der Homepage als „Einstieg“ in eine Web-Applikation. Der Gedanke ist in einer fat-client Applikation durchaus richtig, denn dort gelangt man in die Untermenüs nur durch die darüber gelagerten GUI-Schichten. Nicht so allerdings bei der Web-Applikation, die in der Regel über eine große Zahl von URLs direkt von draußen adressierbar ist (die ja auch als Bookmarks gerne genutzt werden). Hier muss jeder Zugriff auf eine beliebige Seite geschützt werden.

11.6.2 Gibt es ungefährlichen Input?

Bald ergab sich das Problem, dass durch bestimmten Input der Browser zu Aktionen veranlasst werden konnte, die z. B. Daten einer Site an andere lieferten oder

lokale Ressourcen des Users kompromittierten. Vor allem der vermehrte Einsatz von JavaScript führte zu immer neuen Schwachstellen – das zeigt nicht zuletzt unsere kurze Analyse der Fehlermeldungen zu Mozilla und Firefox oben. Natürlich stellte sich in diesem Zusammenhang auch die Frage, ob es ungefährlichen Input überhaupt gibt bzw. wie man gefährlichen Input erkennen und unschädlich machen kann.

Das lässt sich gut an zwei Beispielen diskutieren: Zum einen an der Filterung von Skript-Code und zum anderen am Umgang mit multimedialen Inhalten wie Bildern oder Filmen. Im ersten Fall – dem Skript – ist es ziemlich klar, dass heruntergeladene Inhalte nicht einfach ohne vorher gehende Filterung ausgeführt werden sollten (egal welche Form der Zugriffskontrolle auf Seite des Browsers implementiert ist). Es stellt sich also die Frage, wie und wo dieses Filtern geschehen soll. Es gab und gibt mehrere Versuche, z. B. auf Seiten von Anonymisierungs-Proxies, auf Serverseite bereits gefährliche Inhalte herauszufiltern, aber dennoch insgesamt weiterhin Skript zuzulassen. Es handelt sich also um einen weitaus schwierigeren Fall als beim Unterbinden einer gewöhnlichen Cross-Site-Scripting-Attacke, denn hier genügt es ja, das Script gänzlich zu unterdrücken. Derartige Versuche sind bislang gescheitert. Ohnehin ist aus theoretischer Sicht klar, dass sich die Korrektheit bzw. Gefährlichkeit von Code nicht mit algorithmischen Methoden feststellen lässt.

Einfacher scheint die Frage im Fall von Bildern und Filmen zu sein: Das ist schließlich kein Code. Bevor wir zeigen, wie gefährlich selbst „Nicht-Code“ sein kann, müssen wir erst noch klären, was denn überhaupt „Code“ ist: Code ist alles, was auf der Empfängerseite als Befehl verstanden werden kann. Damit ist die Definition von Code abhängig vom Browser und dessen Ausstattung mit Plug-Ins und liegt somit außerhalb der Kontrolle von Servern. Damit können wir auch eine Obergrenze für die Verhinderungsmöglichkeiten von Cross-Site-Scripting nennen: Serverseitiges Filtern setzt immer eine Theorie über die am Browser vorhandenen Sprachen voraus und ist damit immer nur für eine gewisse Zeit gültig.

Aber kommen wir von dieser eher theoretischen Frage zurück zur Frage nach der Sicherheit von Bildern und Filmen. Hier lautet die traurige Antwort, dass auf Browserplattformen, die auf nicht-typsicheren Sprachen wie C/C++ beruhen, selbst Bilder oder Filme Buffer-Overflows auslösen können. Dann kann dort versteckter Code zur Ausführung kommen und es hängt nur noch von den verfügbaren Privilegien ab, welche Schäden angerichtet werden können.

Aber wie sieht es dann mit Textdateien aus? Da in der Regel die Browser über Mechanismen verfügen, je nach heruntergeladenem Dateityp eine entsprechende Applikation aufzurufen und die Datei zu übergeben, ist das Problem dort genau das gleiche. Selbst Textdateien können also Code einschleusen.

Was sind die Konsequenzen daraus? Ganz einfach – der gesamte Rechner mit installierten Programmen muss jetzt gegen Overflows sicher sein, nicht mehr nur der Browser. Die Angriffsfläche, die nach außen durch den Browser greifbar wird, hat sich auf den ganzen Rechner ausgedehnt.

Wenn also die potenzielle Angriffsfläche schon so groß ist, gibt es dann wenigstens Möglichkeiten, den Schaden durch eingeschleusten Schadcode zu verrin-

gern? Ganz langsam setzt sich die Erkenntnis durch, dass die Gefährlichkeit von Input wie JavaScript nicht durch dessen Syntax oder Variablen bestimmt ist, sondern durch die Ressourcen, die dem Interpreter bei der Verarbeitung zur Verfügung stehen – also seinen Privilegien. Wenn diese im Sinne von POLA begrenzt sind, dann kann auch das gefährlichste Script keinen großen Schaden mehr anrichten. Interessanterweise scheint auch im Bereich des Internet Explorer ein Umdenken in diese Richtung stattzufinden: Neuere Versionen des IE sollen nicht mehr mit den vollen Privilegien des Users laufen.

Sehen wir uns jetzt den Weg zu dieser Erkenntnis an. Er führte über eine Menge „Hobbles“ hinweg.

11.6.3 Symbolische Referenzen

Symbolische Referenzen (symbolic links) sind ein klassisches Mittel, um eine „Confused-Deputy“-Angriffe durchzuführen. Das Prinzip beruht darauf, dem Opfer einen Zugriff zu gewähren, der jedoch letztlich auf eine ganz andere Ressource umgeleitet wird. Hält das Opfer den Link für ein normales File, übersieht es die Umleitung, denn die Umleitung selbst erfolgt automatisch. Es ist außerdem geradezu ein Ziel des Betriebssystemdesigns, die symbolische Referenz so transparent wie möglich erscheinen zu lassen, d. h. der Benutzer soll gar nichts bemerken.

Ebenfalls wenige Chancen hat das Opfer im Falle von grundsätzlich automatisch ausgeführten De-Referenzierungen wie Image Links in HTML, Entity-Referenzen in XML und SGML oder „include“- und „import“-Anweisungen von XSL.

Zur Gefährlichkeit der Referenzen tragen drei Faktoren bei:

- das Verstecken des eigentlichen Ziels (Spoofing),
- die unauthorisierte, automatische De-Referenzierung,
- die automatische Delegation der Rechte des Ausführenden an den Code.

11.6.4 Hobbles und Patches – JavaScript im Netscape-Browser

Das Wort „Patch“ bedeutet wörtlich übersetzt soviel wie „Flicken“ oder „Pflaster“, während mit „to hobble“ in diesem Fall „fesseln“ oder „einschränken“ gemeint ist. Die Begriffe werden von David Flanagan in [Flan] folgendermaßen eingeführt:

„The approach to JavaScript security in Navigator 2.0 and 3.0 has been to first identify security holes through which private information could be exported, and then to plug those holes. Typically, security holes are plugged by implementing a ‘security hobble’--i.e., by restricting or ‘hobbling’ the capabilities of JavaScript so that the hole cannot be exploited. [...] The problem with an identify-and-patch approach is that it can be difficult to identify security holes, and that there is no way of knowing when you’ve found all possible holes. A web browser is a complex thing, and JavaScript is a powerful scripting language.“

Hat sich daran bis heute wirklich so viel geändert? Für viele Entwickler deckt sich das „Hobbling“ mit dem Konzept „programmatische Security“, d. h. dem Einfügen von immer mehr IF-Abfragen zum Stopfen von Sicherheitslöchern (siehe dazu auch [Schm]). Die Autoren erwähnen in der Folge, wie z. B. durch diese „Hobbles“ verhindert wurde, dass Script Zugriff auf private Daten bekommen oder auf das lokale File-System zugreifen konnte. Ein paar Beispiele für die damaligen Attacken im Navigator 2.0:

- Script öffnet heimlich ein verstecktes neues Fenster und führt dort Code mit Privilegien des Users aus.
- Script verwendet spezielle Protokolle (file, about), um Zugriff auf lokale Dateien zu erhalten.
- Script sendet Inhalte heimlich nach außen und verrät dadurch geheime Nutzerdaten.
- Script gelingt es, durch parallele Mechanismen den File-Namen in Upload-Dialogen zu setzen und damit Files nach draußen zu schicken. Der Hauptmechanismus war bereits durch einen „Hobble“ gesichert worden.
- Script liest Daten aus Pages anderer Sites und schickt sie nach außen (als Reaktion darauf wurde das „Same-Origin-Prinzip“ eingeführt, d. h. von einem Webserver geladenes JavaScript hat nur Zugriff auf Eigenschaften derjenigen Objekte, die vom selben Ort wie das JavaScript stammen).

Hier einige Löcher im Navigator 3.0:

- Zugriffe auf das History-Objekt erlauben Script, private Daten zu versenden.
- Der Wert von Passwort-Feldern ist für Script zugänglich. Script gelingt es nach einem Patch, über andere http Methoden weiterhin heimlich Mail zu versenden.
- Der Wert des File-Upload-Dialogs war weiterhin manipulierbar, trotz des Hobbles in 2.0.
- Denial-of-Service-Attacken durch endlose Schleifen in Script oder Schließen von Browserfenstern bis hin zur Beendigung der Applikation. Script liest trotz Same-Origin-Prinzip weiterhin Daten aus Pages anderer Sites und schickt sie nach außen.

Verglichen mit den in Abschn. 11.4 beschriebenen Schwachstellen im Mozilla-Firefox-Browser scheint sich nicht viel geändert zu haben: Nach wie vor haben wir Isolationsprobleme, Privilegienprobleme und mächtige, generische Mechanismen, die fortwährend missbraucht werden und immer mehr „Hobbles“ benötigen, ohne dass klar ist ob der jeweilige „Hobble“ ausreichend ist.

Interessant ist ferner, dass die Autoren einige Probleme als schlichte „Bugs“ abtun:

„Not all security holes in Navigator 2.0 and 2.0.1 were the result of unforeseen interactions between the features of JavaScript and the rest of Navigator. Some were just bugs, plain and simple. For example, one bug allowed a web page to leave JavaScript code behind after it was unloaded. This code could do anything it wanted to.“ [Flan]

Ist das wirklich nur ein „Bug“ oder doch eine Architekturschwäche? Anscheinend ist die Sicherheit der Anwendung an dieser Stelle vom „Aufräumen“ abhängig – wieso überlässt man es dann den Entwicklern, in der Hoffnung, dass diese schon daran denken werden? Eine Ressourcenverwaltung, die im Einzelfall auf die Applikationsprogrammierer abgewälzt wird, scheitert meist. Aus dieser Perspektive ist es also keineswegs ein „Bug“, sondern eine Schwäche der Architektur, die kein automatisches Abräumen von Scriptinstanzen vorsieht. Ganz ähnlich liegt der Fall, wenn z. B. ein API in Form paarweiser Methoden entworfen wird:

- `getSpl()` und `setSpl(level)` zum Sperren von Interrupts,
- `alloc()` und `free()` für das Memory-Management,
- `setPrivileged()` und `resetPrivileged()` zum Wechsel zwischen Rechedomänen.

Allen diesen Mechanismen ist gemeinsam, dass ihr Erfolg davon abhängt, dass die Entwickler sie richtig einsetzen und insbesondere den zweiten Call nicht vergessen. Dies ist schwieriger als es aussieht, da der zweite Call häufig weit entfernt vom ersten auftritt, bzw. es zwischen den beiden Calls viele Möglichkeiten im Code gibt, einen anderen Verlauf zu nehmen – und damit gar nicht am zweiten Call vorbeizukommen.

In Anbetracht der bisher mit solchen Techniken gemachten schlechten Erfahrungen muss man hier von Architekturproblemen und nicht mehr von „Bugs“ reden. Es ist jedoch für die ganze Diskussion von sicherer Software sehr aufschlussreich zu beobachten, wer etwas für einen „Bug“ hält und wer für ein Architekturproblem – der Absturz eines Programms durch wilde Eingaben ist eben nicht nur ein „Bug“, sondern eröffnet die Möglichkeit einer Buffer-Overflow-Attacke. Wenn dieser Absturz durch die Verwendung einer Programmiersprache geschieht, die keine automatische Überwachung von Datentypen wie Arrays erlaubt, dann handelt es sich eben doch um ein Architekturproblem und nicht um einen Bug. Spätestens seit sich der Einsatz von Software immer mehr auf hochkritische Bereiche wie Fahrzeuge und Haustechnik verlagert, sollte der Unterschied ernst genommen werden.

Wie sehen mögliche Lösungsansätze aus? Closures sind Programmierkonstrukte, die Methoden und Ausführungsumgebung kapseln und dafür sorgen, dass außerhalb der Closure im Innern gesetzte Privilegien nach dem Aufruf der Funktion nicht mehr verfügbar sind. Ähnlich elegant ist die Verwendung von Stackvariablen, die beim Return von Funktionen oder Methoden automatisch beseitigt werden. Wir haben dieses Prinzip bereits in simulierter Form bei den `JavaDoPrivileged()`-Funktionen gesehen. Dort lebt der privilegierte Modus genau so lange, wie die Ausführung der Funktion dauert, und wird dann ohne Mithilfe des Programmierers zurückgesetzt.

Auch im Falle der Denial-of-Service-Attacken durch Endlosschleifen ist ein direkter Bezug zur Architektur da: Wenn Browser und fremdes Script im gleichen Prozess laufen, dann fallen Quota-Mechanismen des Betriebssystems weg (so wie durch Multithreading innerhalb eines Prozesses die Schutzmechanismen des Virtual-Memory-Managements nicht wirksam sind). Damit sind DOS-Attacken

schwieriger abzuwehren. Hier stellt sich wieder die Frage nach der grundsätzlichen Browserarchitektur.

Aber die Frage „Bug oder Architekturschwäche?“ ist selbst im Entwicklerlager sehr umstritten, wie der Rückzug des PHP-Sicherheitsexperten Stefan Esser aus der PHP-Entwicklung vor einiger Zeit gezeigt hat. Einer Meldung von Heise News zufolge war das Verhältnis des Sicherheitsexperten zu den PHP-Kernentwicklern aufgrund der von Esser wiederholt gemeldeten Sicherheitslücken bereits angespannt. Zum Bruch kam es, als die Entwickler Funktionen, die nach Ansicht von Esser immanent unsicher und gefährlich waren, wie z.B. `allow_url_fopen` oder `allow_url_include`, nicht ändern wollten. Mit ihnen kann PHP-Code von beliebigen externen Stellen geladen werden. Zur Begründung meinten die Entwickler, dass nur unerfahrene Programmierer damit Probleme hätten. Zu allem Übel sind diese Funktionen offenbar auch defaultmäßig freigeschaltet [Esser].

Die hier zutage tretende Denkweise der PHP-Kernentwickler ist noch in einer anderen Weise typisch: Die Forderung, dass eine Lösung auch für Entwickler eine bestimmte Usability haben sollte, damit sie sicher eingesetzt werden kann, tritt oft hinter den Glauben an die eigene Professionalität zurück. Der Zusammenhang zwischen Usability und Security gilt jedoch auch für Entwickler von Software.

11.6.5 Sicherheitskonzept des Browsers

Die Geschichte von JavaScript im Netscape-Browser zeigt sehr schön, dass in diesem Fall das Sicherheitskonzept sozusagen aus den Fehlern heraus langsam entwickelt wurde. So entstand das „Same-Origin“-Konzept aus der Gefahr heraus, dass fremdes Script auf Inhalte anderer Sites zugreifen konnte. Bezeichnenderweise handelte man sich beim Einbau dieses „Hobbles“ schnell ein weiteres Problem ein: Script von Unterdomains konnte nicht mehr auf Inhalte benachbarter Unterdomains derselben Domain zugreifen. Dieses Problem musste durch die Einführung einer speziellen Property (Domain) gelöst werden. Ein durchgängiges Sicherheitskonzept für den Browser ist hingegen nicht zu erkennen. Zu einem solchen Konzept würden die folgenden Punkte gehören:

- Ein Modell von Sicherheit, das den Systemkontext des Browsers sowie seine hauptsächlichen Abstraktionen mit einbezieht.
- Eine korrekte softwaretechnische Umsetzung des Modells in der Implementation – möglichst ohne einen Irrgarten von IF-Anweisungen, sondern durch klare Isolation von Komponenten und Abläufen.
- Das Vermeiden von sicherheitsrelevanten Änderungen unterhalb der Modell- und Konzeptebene durch Seiteneffekte oder Hintertüren aufgrund von Änderungen darunterliegender Schichten (Sprache, Betriebssystem).

Beginnen wir mit dem letzten der drei Punkte: Sicherheitsmodell und Implementation klaffen gerne weit auseinander. Ein gutes Beispiel dafür ist die Tatsa-

che, dass in einem Web-Server die Sicherheit des Servers, aber auch des gesamten Systems durch eine Änderung an den Zugriffsrechten auf Dateien oder Directories leicht ausgehebelt werden kann. Administratoren, die den Apache-Web-Server oder auch Servlet Engines in Applikation-Servern konfiguriert haben, können ein Lied davon singen. Das Sicherheitsproblem besteht darin, dass die technische Änderung der Zugriffsrechte auf einer Ebene weit unterhalb des Sicherheitskonzeptes durchgeführt wird. Die Konsequenzen solcher Änderungen sind daher nicht unmittelbar sichtbar, und es braucht extrem viel Erfahrung, um hier Fehler zu vermeiden. Entwickler und Systemverwalter müssen das Sicherheitskonzept sozusagen verinnerlicht haben, um die Verbindung von Konzeptebene und Dateisystem herstellen zu können.

Hier ein weiterer Beleg für unsere These, dass unterschiedliche Sicherheitsdomänen und Layers die Gesamtsicherheit eines Systems gefährden, wie Heise [heise85105] berichtet:

„Die Entwickler des Open-Source-Wiki-Systems Twiki melden mal wieder eine Sicherheitslücke. Sie betrifft die Versionen 4.0.x, 4.1.0 und ältere mit der Erweiterung SessionPlugin. Die Lücke führt dazu, dass lokale Nutzer Code einschleusen und unter den Rechten des Webservers ausführen lassen können. Das ist besonders auf Shared-Hosting-Plattformen fatal, wo fremde Admins auf dem selben Server die Lücke nutzen können. Oder sie lassen unsichere Webapplikationen laufen, durch die dann auch der Twiki-Server verwundbar wird.

Der Fehler betrifft das CGI-Session-Verzeichnis, in dem fatalerweise alle Benutzer Schreibrechte haben. Es reicht nicht, CGI Sessions zu verbieten, da es einen Session Cleanup Code gibt, den der Server grundsätzlich ausführt. Die Entwickler empfehlen ein Upgrade auf Version 4.1.1. Als Hotfix bietet es sich an, ein Verzeichnis mit korrekt gesetzten Zugriffsrechten einzurichten.“

Hier finden wir auch ein Beispiel für den engen Zusammenhang zwischen Usability und Security, wie ihn Simson Garfinkel formuliert hat: Ein gleichermaßen sicheres und benutzbares System sollte dem „Principle of least surprise“ genügen [Garf]. Was aber hat ein Verzeichnisrecht mit der Verwaltung von Diensten zu tun?

Nun zu dem Punkt „korrekte softwaretechnische Implementation“: Das bedeutet, dass die Implementation durch die verwendete Architektur in der Lage ist, das Modellkonzept wahrheitsgetreu zu verwirklichen. Im obigen Beispiel der paarweisen Funktionen ist dies z. B. nicht der Fall. Das Modellkonzept eines geschlossenen Sicherheitsbereiches aus Code und Daten wird verletzt, wenn es nicht durch ein Softwarekonstrukt so umgesetzt wird, dass gar keine andere Möglichkeit bleibt, als vollständig aufzuräumen.

Wir stoßen auf diese Problematik noch öfters, etwa bei der Trennung von Sites und ihrem dazugehörigen Script oder bei der Behandlung globaler Bereiche. Wenn im Sicherheitsmodell z. B. eine Isolation von Sites voreinander gefordert wird, macht dann ein globales DOM mit einem einzigen globalen Interpreter als Architektur der Implementation wirklich Sinn? Ein anderes Beispiel: Der Session-Mechanismus, der durch Cookies realisiert wird, ist Teil einer Vertrauensbeziehung zwischen Benutzer und Site – wieso kann dann Script innerhalb einer Page

auf dieses Cookie zugreifen und es per Image-Request an einen Attacker senden? Natürlich tritt dies nur ein, wenn die Site eine Cross-Site-Scripting-Vulnerability aufweist, über die fremdes Script eingeschleust werden kann. Trotzdem bleibt die Frage, warum auf dieses kritische Credential zugegriffen werden kann. Man könnte sich unterschiedliche Typen von Cookies überlegen, die durch sogenanntes *Tainting* (Tagging mit Sicherheitseigenschaften) geschützt werden könnten – dazu mehr im Abschn. 11.6.6.

Kommen wir nun zu unserem ersten Punkt, dem Sicherheitsmodell: Wie Flanagan oben (Beginn Abschn. 11.6.4) so schön beschrieben hat, scheint es im Falle der Web-Browser ursprünglich gar kein echtes Sicherheitsmodell gegeben zu haben. Aufgrund des eingeschränkten Zugriffs von JavaScript auf lokale Ressourcen wähten sich die Netscape-Entwickler sehr sicher. Die ersten Probleme, die auftauchten, bestanden im Zugriff von Script einer Seite auf allgemeine Browser-Daten wie z. B. die Browser-History. Durch den generischen Mechanismus der URL-Protokolle wie `file://`, `mailto:` etc. konnte Script jedoch plötzlich sehr wohl wichtige lokale Ressourcen lesen oder schreiben und sogar Netzwerkverbindungen aufbauen. Gepaart mit einem vor allem auf Flexibilität abzielenden Design von User-Interface-Elementen im HTML (unsichtbare Forms, unsichtbare Frames, Zero-Size-Bilder etc.) war dem Missbrauch Tür und Tor geöffnet. Man darf an dieser Stelle ruhig die Frage aufwerfen, was unsichtbare oder funktionsunfähige Elemente innerhalb eines User-Interface-Designs überhaupt sollen? Es kommt der Verdacht auf, dass etliche GUI-Elemente in HTML lediglich als Ersatz für ansonsten fehlende Funktionen missbraucht wurden (z. B. die Zero-Size-Bilder zur Kommunikation einer Page mit anderen Sites ohne Wissen des Nutzers, was vor allem im Bereich der Werbung massiv ausgenutzt wird, aber auch fester Bestandteil von AJAX-Techniken ist). Jeder erkannte Missbrauch von JavaScript wurde durch „Hobbling“, eine programmatische Einschränkung von JavaScript, gelöst. Als die Verletzungen der Privatheit von Usern durch Export ihrer Daten zu anderen Sites immer öfter vorkamen, entstand ein neues Sicherheitskonzept: Das „Same-Origin“-Prinzip: Script eines bestimmten Ursprungs durfte nur auf Daten zugreifen, die zu diesem Ursprung gehörten. Damit mussten die JavaScript-Interpreter das Herkunftsprinzip lernen.

Schnell stellt man allerdings fest, dass für ein echtes Sicherheitsmodell weitere Begriffe geklärt werden müssen, wie etwa:

- Browser-Session und Credentials,
- Windows, Frames und andere GUI-Konstrukte,
- Sites und Domains,
- Plug-Ins und Extensions, Download und Installation,
- Script in Pages und im Browser, dynamisches Script,
- User und Permissions, Abweichungen von same origin,
- Regeln für Intranet vs. Internet,
- Privileged Content vs. viele unprivilegierte, aber separate Site-Contexts,
- DOM und NON-DOM Elemente, Isolation.

In der Übersicht sieht also die Entwicklung des Sicherheitskonzeptes der Netscape-Mozilla-Browserlinie ungefähr wie in Abb. 11.3 aus.

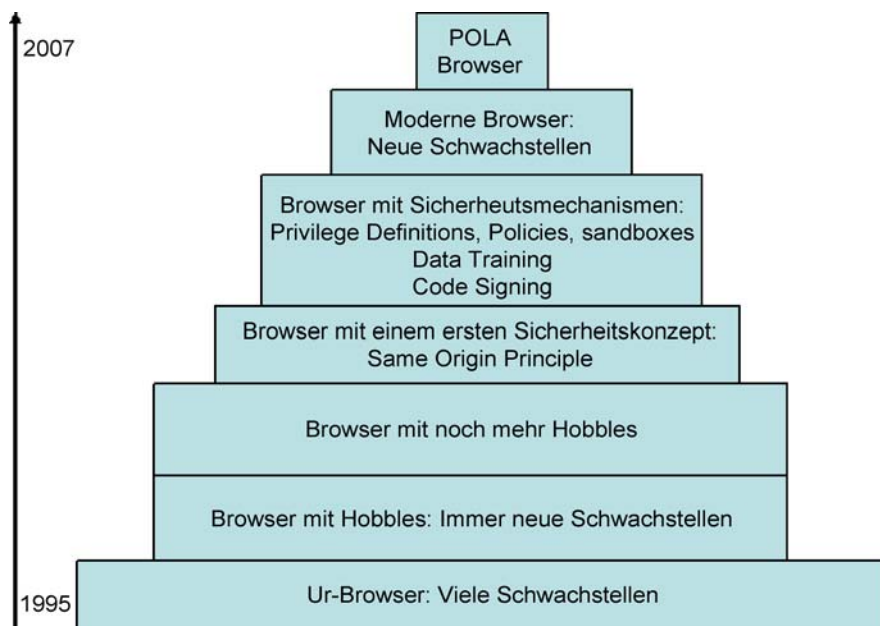


Abb. 11.3 Zeitliche Entwicklung der Browser-Security

Auffällig sind daran die wiederkehrenden „Hobbles“, die immer wieder nötig zu sein scheinen (bis hin zum „Über-Hobble“, dem generellen Abschalten von JavaScript im Browser).

Ein weiterer interessanter Punkt sind die verschiedenen Ansätze in Bezug auf die Sicherheitstechniken, mit denen das Sicherheitskonzept umgesetzt werden sollte. Bevor sich die Sicherheitstechnik basierend auf Permissions und Policies wie im Java-Bereich durchsetzen konnte, gab es einen Versuch, eine Technik einzusetzen, die auf der Herkunft von Daten oder Funktionen basiert. Durch die Ähnlichkeit zum Same-Origin-Konzept lag dies auch nahe. Dieses „Tainting“ genannte Konzept untersuchen wir gleich in einem eigenen Abschnitt (vgl. Abschn. 11.6.6), da es doch eine eher selten eingesetzte Technik darstellt.

Data-Tainting wurde jedoch bald als Sicherheitskonzept aufgegeben und durch eine feingranulare Rechtevergabe auf der Basis von signiertem Code oder User-Aktionen ersetzt. Damit folgt das JavaScript-Sicherheitsmodell in Mozilla-Firefox im Wesentlichen der Entwicklung der Security Policies in Java. Auch die Entwicklung beim Internet Explorer verlief ähnlich, nur wurden dort die Policies „Zonen“ genannt.

Beiden Konzepten – Policies wie Zonen – ist etwas gemeinsam, das wir bereits bei der Diskussion der Java2-Security kennegelernt haben: Sie werden kaum verwendet. Die Konfiguration der Policies erfolgt entweder manuell oder über grobe Templates, die sich ganz eindeutig an eine bestimmte Klientel richten, nämlich die Administratoren großer Intranets. Diese verfügen über die nötigen Kenntnisse, um die Policies entsprechend den Sicherheitsbedürfnissen einer Firma einzurichten. Im

Privatbereich ist die Verwendung dieser Sicherheitsinstrumente so gut wie unbekannt. Die Zonen bzw. Policies bedeuten im Allgemeinen, dass für kritische Operationen wie z. B. dem Download von Files oder dem Zugriff auf lokale Ressourcen eine Einstufung einer Site in eine vertrauenswürdige Zone nötig ist. Möchte der User also die entsprechende Funktionalität tatsächlich haben, bleibt nicht viel anderes übrig, als im entsprechenden Dialog „Wollen Sie wirklich ...“ mit „Ja“ zu antworten. Diese Entscheidung ist jedoch eine Alles-oder-nichts-Entscheidung, bei der der User die Kontrolle über den Browser (bzw. die Maschine, falls er Administrator ist), völlig abgibt. Das Gleiche gilt auch für die Erweiterung des Browsers durch Extensions oder Plug-Ins, die erfahrungsgemäß weitere Sicherheitsprobleme mit sich bringen.

Neben der Durchsetzung von Policies kennen beide Konzepte noch einen weiteren wichtigen Baustein: Signierten Code. So können JavaScript oder ActiveX durch ihre Erzeuger signiert werden. Die User können dann entsprechend der durch die Signatur bestätigten Herkunft des Codes Permissions vergeben. Auch dies ist wieder ein Feature, das hauptsächlich im Intranet Sinn macht, wie Powell und Schneider feststellen [PS].

Denn dem einzelnen User nutzt es wenig, wenn er erfahren kann, wer ein Stück Script erzeugt hat, falls zum Autor kein persönliches Vertrauensverhältnis besteht. Ansonsten gelten für diese Technik die gleichen Sicherheitsprobleme wie sie üblicherweise in Systemen mit zu viel ambienter Autorität auftreten.

11.6.6 Data-Tainting

Ausgangspunkt der Überlegungen zum Data-Tainting war das Prinzip des Same-Origin: Daten sollten nur an Sites geschickt werden, von denen sie ursprünglich herkamen. Aber es gab auch Daten, die die Maschine gar nicht verlassen sollten. Gleichzeitig sollte Script jedoch auf einer Maschine in der Lage sein, beliebige Operationen vorzunehmen, ohne dabei die Privatheit des Users zu verletzen. Flanagan nennt in diesem Zusammenhang das Beispiel eines JavaScript-Debuggers, der auf einer Site angeboten wird. Durch das Same-Origin-Gebot kann der Debugger in einem Fenster des Browsers nicht auf Script in anderen Fenstern zugreifen – der User muss den Code also lokal abspeichern. Lokal gespeicherter Code hat bei Browsern traditionell die Privilegien der Applikation (auch Java und .NET verfahren hier defaultmäßig nicht anders).

Code, der von Site X herkam, sollte also praktisch eine Auszeichnung (ein Tag) erhalten, damit später bei der Ausführung das Same-Origin-Prinzip eingehalten werden konnte. Das Gleiche galt für Daten, je nachdem, von wo sie stammten. Die Tags mussten natürlich bei Daten oder Code verbleiben, selbst wenn sie durch andere Funktionen manipuliert wurden. Das Tainting wurde also auch auf abgeleitete Daten oder Funktionen angewendet, was einen recht tiefgehenden Eingriff in JavaScript bedeutete.

Vom Prinzip her ähnelt das Verfahren stark der Multi-Level-Security, bei der Daten ebenfalls eine Auszeichnung in Levels und Kategorien erhalten (siehe auch

das Kapitel zu Content-Management-Security). Die Sicherheit dieses Systems besteht nun darin, dass es nicht mehr in der alleinigen Verantwortung eines Programms liegt, ob Daten manipuliert werden können. Programm und Daten müssen gleiche oder verträgliche Klassifikationen besitzen.

Interessant ist die Begründung von Flanagan, warum dieses Konzept letztlich im Netscape-Browser nicht mehr zum Einsatz kam: Existierendes JavaScript löste solche Mengen von Sicherheitsdialogen mit Bitte um Bestätigung aus, dass dies den Benutzern nicht mehr zugemutet werden konnte. Die Arbeit an alternativen User-Interface-Designs ging jedoch nicht schnell genug voran, sodass letztlich der Umstieg auf das bekannte Java-Sicherheitsmodell erfolgte.

Die Begründung klingt ganz ähnlich wie bei der generellen Ablehnung von fein-granularer Sicherheit. Aber ist die Belästigung der User eine zwangsläufige Folge granularer Sicherheit oder ein Zeichen, dass GUI und Sicherheitskonzept an dieser Stelle noch nicht aneinander angepasst sind? Die Frage der Zustimmung durch den User – im Data-Tainting-Fall bedeutet dies evtl. eine Konvertierung von Labels für Daten oder Code – kann bei fein-granularer Sicherheit grundsätzlich ganz anders erfolgen. Die Techniken dafür werden in den nächsten beiden Kapiteln zu den Themen Objekt-Capabilities und Usability vorgestellt (vgl. Kap. 13 und 14).

Welche Probleme könnte Tainting noch lösen? Was passiert z. B. mit Code, der – vom Internet kommend – lokal auf der Platte gespeichert wird? Meist erlangt er dadurch hohe Privilegien. Tainting könnte beispielsweise dafür sorgen, dass die Historie des Codes nicht vergessen wird. Sichere Abgrenzungen z. B. für Sites können dadurch erstellt werden. Generell kann das Konzept überall da verwendet werden, wo die Sicherheitspolicy die Herkunft von Daten oder Code für Trust-Zwecke verwendet, oder wo geschlossene Zonen aus Daten und Kommunikationspartnern erzeugt werden müssen.

Im Kap. 10 zu den Mechanismen sicherer Software wurde Tainting bereits einmal kurz vorgestellt. Hier wollen wir nochmals auf das Paper von R. Schneider [Schn] zurückkommen und einige der Ideen zum Einsatz von Tainting für die Browser Security kurz vorstellen. Diese Ideen sind nicht zuletzt durch die große Zunahme von Web2.0-Applikationen mit ihrem Konzept der Mashups und Cross-Site-Requests von großer Bedeutung.

Schneider nennt im Wesentlichen vier Anwendungsfälle:

1. Sichere Unterscheidung von Site-JavaScripts und Chrome JavaScript
2. Isolation von Websites gegeneinander im Browser
3. Sicheres Entfernen von Site-JavaScript beim Verlassen einer Website
4. Vermeiden oder gezieltes Erlauben von Cross-Site-Zugriffen auf Daten oder Funktionen

Der erste Punkt wurde oben anhand der Mozilla-Fehlerliste bereits ausführlich besprochen. Bedenklich daran ist, dass ohne zuverlässige und permanente Kennzeichnung fremden Codes die beschriebenen Irrtümer bei der Ausführung immer wieder auftreten werden.

Der zweite Punkt erlaubt es, Daten und Code einer Site untrennbar an diese Site zu binden. Bei der Erzeugung eines neuen Datums wird automatisch der Taint der Site, von der der Code zur Erzeugung stammt, auf das Datum propagiert.

Der dritte Punkt hilft Fehler zu vermeiden, die daraus resultieren, dass eine Site Code in eine andere Site injizieren kann. Beim Wechsel der Site wird dann trotzdem der Code der alten Site innerhalb der neuen Site aktiv. Der Internet Explorer zeigte kürzlich solche Sicherheitslücken, die jedoch innerhalb des Mozilla-Browsers ebenfalls schon vor längerer Zeit aufgetreten waren. So konnte eine Site den Code bestimmen, der beim Öffnen eines Pop-Ups einer anderen Site ausgeführt werden sollte.

Der letzte Punkt betrifft die generelle Notwendigkeit von Cross-Site-Requests bzw. dem Austausch über Sites hinweg. Durch Tainting kann dieser Austausch entweder sicher verboten oder gezielt erlaubt werden.

Der geeignete Leser sei bezüglich der Details an [Schn] verwiesen.

Ein drastisches Beispiel für das Fehlen von Tainting erlebte kürzlich die Polizei in Hessen: Eine größere Anzahl von Untersuchungsberichten mit persönlichen Daten und Erkenntnissen zu den untersuchten Personen erschien statt im Intranet im Internet. Die Untersuchung im Anschluss an den Skandal erbrachte als Resultat, dass der Beamte versehentlich den falschen Publizierungsvorgang ausgelöst hatte. Er hatte Publikationsrechte sowohl für das Intranet als auch das Internet erteilt, und da die Daten nicht sagen konnten, wo sie hindürfen, erscheinen sie schließlich an der falschen Stelle.

Einige offene Fragen bleiben noch anzusprechen: Wären Tainting-Konzepte verträglich mit anderen Sicherheitstechniken? Könnten sie in Browsern bzw. deren Skript-Sprachen nachgerüstet werden? Wie verträgt sich Tainting mit Privilegien bzw. Modi?

11.6.7 Eine alternative Architektur

Die Hauptproblematik der momentanen Browserarchitekturen lässt sich mit wenigen Stichpunkten beschreiben:

- Die Grundarchitektur der Sicherheit ist subtraktiv: Mächtige generische Mechanismen in der Implementation werden mühsam nachträglich gebündelt durch Policies.
- Sicherheitsmodell und Implementation sind strukturverschieden.
- Sicherheitstechniken wie Privilegienmodelle oder Zonen unterschiedlicher Privilegien bescheren permanente Fehler im Bereich Privilege Escalation.

Langsam scheint sich die Erkenntnis durchzusetzen, dass es unmöglich ist, einen wirklich sicheren Browser auf dieser Basis zu bauen. Sowohl für den Internet Explorer als auch den neuen Google-Browser Chrome werden neue Sicherheitsmechanismen angedacht. Unserer Meinung nach bleibt letztlich nur die Möglichkeit der Schadensbegrenzung, z. B. durch die Verwendung von POLA-

Mechanismen, die wiederum eine generelle Reduktion von Autorität mit sich bringen. Damit steigt man um auf ein additives Verfahren: Der Browser läuft grundsätzlich nicht mehr mit vollen Privilegien. Seine Komponenten werden ebenfalls mit minimalen Privilegien ausgestattet und erhalten bei Bedarf erweiterte Rechte in Form von Objekt-Capabilities.

Das Konzept der „Verengung“ von Zugriff durch eingeschränkte Capabilities löst auch den scheinbaren Konflikt zwischen dem Drang nach generischen Lösungen einerseits und der Forderung von granularem Zugriff aus Gründen der Sicherheit andererseits.

Kapitel 12 zeigt eine solche Architektur: Der DARPA-Browser war ein Versuch, einen sicheren Browser zu bauen, bei dem ein Renderer beliebig bösartig sein konnte. Schaden wird in diesem System dadurch verhindert, dass der Renderer keinen Zugriff auf Ressourcen erhält, die missbraucht werden könnten. Pages werden streng in der Implementation voneinander getrennt durch Memory-Management-Techniken oder durch einen separaten Interpreter/Renderer pro Page. Unabhängige Komponenten verwalten URL-Auflösungen, der Renderer erhält lediglich geöffnete Streams (analog zu geöffneten File-Deskriptoren).

Literatur

- [Esser] S. Esser, Blogeintrag,
<http://blog.php-security.org/archives/61-Retired-from-securityphp.net.html>
- [Flan] D. Flanagan, Javascript, The definitive Guide, O' Reilly 2006
- [Garf] S. Garfinkel, Design Principles and Patterns for Computer Systems That Are Simultaneously Secure and Usable, Ph. D. Thesis, MIT 2005,
<http://www.simson.net/thesis>
- [heise77956] Heise News: 655 Mängel im Firefox-Quelltext – theoretisch,
<http://www.heise.de/newsticker/meldung/77956>
- [heise85105] Heise News: Sicherheitslücke in Twiki,
<http://www.heise.de/newsticker/meldung/85105>
- [Moz] Mozilla Vulnerability List,
www.mozilla.org/products/security/vulnerabilities.html
- [MS] C. Meinel, H. Sack, WWW, Springer-Verlag 2004
- [PS] T. Powell, F. Schneider, Javascript – The complete Reference, McGraw-Hill Professional, 2004
- [SaSch] J. H. Saltzer, M.D. Schroeder, The Protection of Information in Computer Systems, online verfügbar unter <http://www.cs.virginia.edu/~evans/cs551/saltzer/>
- [Schm] J. Schmidt, Chrome-Plated Holes, <http://www.heise.de/security/artikel/61652>
- [Schn] R. Schneider, Das Konzept des Tainting in der Browser Security, HDM 2008, www.kriha.de/krihaorg/dload/uni/BrowserTainting.pdf

Kapitel 12

Browsersicherheit durch Object Capabilities

Capabilities wurden bereits mehrfach angesprochen, z. B. im Rahmen der Plattformensicherheit bei der Absicherung von Servern. Neben Namespaces, Access Control Lists und Labeling- bzw. Taming-Ansätzen stellen sie eine weitere Technik für sichere Software dar. Dieses Kapitel untersucht anhand eines real durchgeführten Experimentes, wie weit sich Capabilities – hier speziell Object Capabilities – für die Entwicklung sicherer Subsysteme eignen.

Von Bedeutung sind dabei beide Teile des Experiments: Die Architektur eines sicheren Browsers, aber auch die Möglichkeiten der Sicherheitsanalyse in Capability-basierten Architekturen. Der entscheidende Unterschied zu „normalen“ Softwarearchitekturen besteht darin, dass hier versucht wird, die Sicherheit des Subsystems – hier eines Renderers für einen Browser – in der Architektur selbst zu verankern. Entsprechend deutlich sind auch die Auswirkungen auf die Softwarearchitektur, die schnell zu Problemen bei der Integration bestehender Bibliotheken führen kann. Das sichere Einbetten solcher Bibliotheken – das sogenannte „Taming“, ist Bestandteil der Untersuchung.

In Bezug auf Java 2 Security wird häufig kritisiert, dass die Permissions viel zu granular sind, als dass sie in der Praxis verwendet würden – sprich, dass Softwareentwickler davor zurückscheuen. Die gleich beschriebene Lösung auf Basis von Capabilities erlaubt jedoch noch granularere Rechtevergaben. Dies wirft natürlich Fragen bezüglich der Usability dieses Ansatzes auf, insbesondere, weil im Zusammenhang mit Browserrechten auch die Benutzer mit der Konfiguration und Vergabe von Rechten konfrontiert sind und nicht nur die Entwickler.

Die Fähigkeiten des hier beschriebenen sicheren Browsers sind im Vergleich zu einem vollen Browser zweifellos gering. Interessant daran ist jedoch, wie schwierig es ist, auch nur diese minimale Funktionalität sicher anzubieten. Im Anschluss an das Experiment werden wir versuchen, die neuen Techniken aus dem Web-2.0-Umfeld in die vorgestellte Architektur einzubetten. Vielleicht stellt sich dabei heraus, dass die Object Capabilities durch weitere Techniken ergänzt werden müssen.

12.1 Beschreibung des Experiments

Im vorigen Kap. 11 zur Browseranalyse haben wir viele Sicherheitslücken gesehen, bei denen Script oder Plug-Ins Zugriffe durchführten, die die Sicherheit des Benutzers und seiner Daten gefährdeten. Im Jahr 2003 führten Mark Miller und Mark Stiegler von der Firma Combex ein Experiment zur Entwicklung eines sicheren Browsers durch [MS02]. Untersucht wurde, wie weit sich Object Capabilities als Grundlage für so einen Browser eignen würden. Im Anschluss an die Implementation wurde eine Sicherheitsanalyse von unabhängigen Spezialisten durchgeführt. Sie ist Gegenstand des Abschn. 12.3 zur Sicherheitsanalyse.

Dieser sogenannte DARPA-Browser (seine Entwicklung wurde durch die DARPA gefördert) hatte als Ziel, eine Runtime-Umgebung zu schaffen, bei der ein böartiger Renderer keinen Schaden anrichten konnte. Wir haben bei der Fehleranalyse im Kap. 11 gesehen, dass Browser Plug-ins und Extensions typischerweise die volle Kontrolle über einen Browser übernehmen können. Böartiger Code ist somit in keiner Weise kontrollierbar. Der DARPA-Browser hingegen sollte immun gegen böartigen Renderer-Code sein. Ziel der Entwicklung waren die folgenden Garantien, die die Benutzer von diesem sicheren Browser erwarten konnten:

- Keine Möglichkeit, Daten auf der Festplatte des Rechners der Nutzer zu verändern.
- Keine Möglichkeit, dasjenige Feld im Browser zu verändern, welches die zuletzt besuchte URL anzeigt.
- Keine Möglichkeit, das Browser-Icon in der linken oberen Ecke des Fensters zu verändern.
- Keine Möglichkeit, die Titelleiste dieses Fensters zu verändern.
- Keine Möglichkeit, Informationen von einer URL zu erhalten, die nicht die aktuell besuchte URL ist.
- Keine Möglichkeit, per Hyperlink zu einer anderen URL zu wechseln, ohne dass dies der Browser in seinem Adressfeld anzeigt.
- Keine Möglichkeit, Informationen an irgendeine URL zu senden.

In Zeiten von Web2.0 sind das keine besonders aufregenden Garantien. Im Gegenteil, manches davon ist eindeutig eine zu große Einschränkung für eine Informationsaggregation, wie sie heute z. B. durch Google Maps möglich ist. Deshalb hier nochmals der Hinweis: Es geht in diesem Experiment nicht darum, einen mächtigen Browser zu implementieren, sondern die Sicherheit bestimmter Funktionen durch die *Architektur* des Browsers zu garantieren, und eben nicht durch eine Vielzahl von IF-Abfragen im Code oder durch externe Konfiguration von Permissions.

Trotz allem: Wenn man diese Liste von Garantien mit den Fehlern in der Vulnerability-Liste von Mozilla und Firefox vergleicht, wird man feststellen, dass alle diese Requirements in den Browsern bereits gebrochen wurden, und dass es keine Garantie gibt, dass nicht in der nächsten Version alte, bereits behobene Fehler erneut auftauchen.



Abb. 12.1 Korruptierbare Elemente eines Browsers

Entscheidend für das Verständnis dieses Experiments ist, die Verfügbarkeit von Autorität zu beobachten: Was darf ein Browser automatisch? Wozu braucht er eine Einverständniserklärung (Autorisierung) durch den Nutzer? Was macht HTML implizit? Wie erhält ein Browser bzw. ein Renderer eine Autorisierung durch den User?

Ein bössartiger Renderer kann praktisch jede Funktionalität des Browsers korumpieren (Abb. 12.1).

Was darf denn dann unser sicherer Browser noch? Hier eine Liste der Fähigkeiten des Renderers:

- Der Renderer kann einen Tracing-Dienst aufrufen, um Debugging-Nachrichten auszugeben.
- Der Renderer kann eine Referenz (d.h. eine Capability) auf das `Javax.swing.JScrollPane`-Objekt bekommen, das den Teil des Browserfensters repräsentiert, das zum Renderer gehört (man beachte, dass der Renderer keine Capability für das URL-Feld oder das übergeordnete Fenster bekommt, sehr wohl aber eine Capability für den Scrollbalken).
- Der Renderer kann den Browser zum Wechsel zu einer neuen URL veranlassen. Dies geschieht durch Aufruf von `RenderPGrantor.gotoURL()` und Übergabe einer neuen URL als String. Dies veranlasst den CapBrowser zu einigen Aktionen, um das URL-Feld aktuell zu halten und die Anfrage mit der gewünschten Security Policy abzugleichen. Der Leser beachte dabei, dass es

der potenziell bösartige Renderer ist, der hier den Wechsel durch die Angabe eines Strings (URL) bestimmt. Dieses Vorgehen ist nicht optimal im Sinne der Capability-Theorie.

- Der Renderer kann den Browser bitten, ein Image für ihn zu holen.

Wie meist, wenn man einzelne Module untersucht, stellt sich heraus, dass die tatsächlich benötigten Rechte in keinem Verhältnis zu den üblicherweise durch ACLs gewährten Rechten stehen. Eigentlich sollte es bei so einem minimalen Set an Rechten unmöglich sein, einen Missbrauch zu begehen. Dennoch zeigte die Sicherheitsanalyse des DARPA-Browsers einige überraschende Sicherheitslücken (vgl. Abb. 12.4 und 12.5). Wichtig ist noch anzumerken, dass der Renderer diese Rechte nicht einfach besitzt, sondern sie vom Browser zugeteilt bekommt. Eigentlich besitzt der Renderer keinerlei Rechte und vermeidet somit jegliche Form der „Ambient Authority“.

12.2 Architektur, Design und Implementation der Lösung

12.2.1 Makro-Architektur

Die Makro-Architektur des sicheren Browsers folgt dem Modell des sich verengenden Trichters: Rechte werden von größeren Modulen an kleinere Module nur teilweise weitergegeben. Damit wird erreicht, dass jedes Modul nur die Rechte erhält, die es tatsächlich braucht.

Grundlage der Sicherheit ist eine Trusted Computing Base, die aus einem Interpreter der sicheren Sprache E besteht. Dieser wiederum läuft auf einem Java-Runtime-System, das wiederum auf einem entschärften Linux-System beruht. Zusammen bilden diese Komponenten die E Language Machine (ELM) (Abb. 12.2). Ein Desktop, der Rechte über Drag-und-Drop-Aktionen des Nutzers weitergeben kann, sowie ein Capability-fähiges Window Toolkit können Applikationen (sogenannte *Caplets*) starten. Diese Applikationen erhalten granular gerade die Rechte, die sie für ihre Ausführung benötigen und können diese Rechte wiederum in Teilen an Subsysteme wie z. B. den Renderer weitergeben.

Die sichere Programmiersprache E zeichnet sich durch eine funktionale Denkweise aus (allerdings sind auch Dirty Assignments erlaubt, wenn die Variablen eigens dafür gekennzeichnet sind). Basierend auf dem Lambda Calculus bietet die Sprache Closures, Higher-Order-Functions sowie umfangreiche Funktionen für verteilte Systeme auf der Basis von Capabilities.

Diese Architektur erlaubt es auch, aus einer Applikation heraus weitere Applikationen in einem sehr eingeschränkten Sicherheitskontext zu starten, z. B. aus einem Mail-Client heraus eine Applikation zur Bearbeitung eines bestimmten Dateiformats. Was auf anderen Plattformen eine hochriskante Aktion ist, hat hier kaum Sicherheitskonsequenzen – und benötigt daher auch keine speziellen Warnungen der Form „Wollen sie wirklich ...?“.

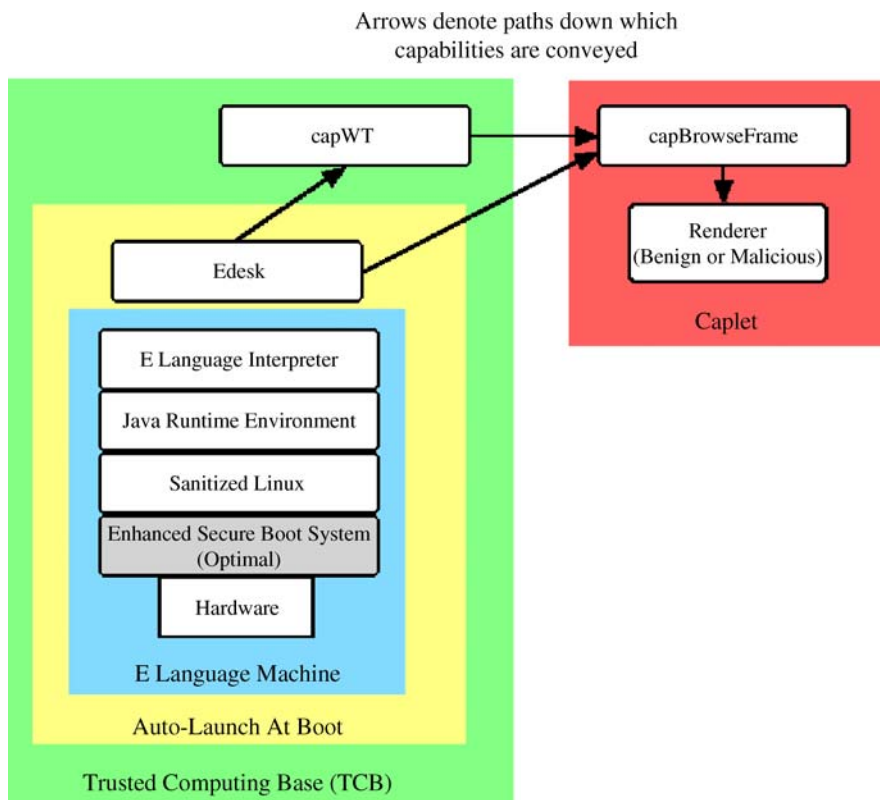


Abb. 12.2 Makro-Architektur des DARPA-Browsers

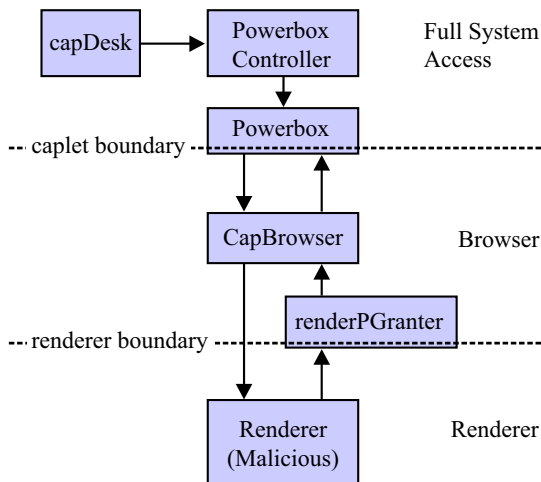
12.2.2 Komponenten und Applikation

Wie werden nun die benötigten Rechte an die Komponenten weitergegeben? Eine zentrale Rolle spielt dabei der sogenannte *Cap-Desk*, ein sicherer Desktop, der Anwenderaktionen in Rechte für Applikationen umsetzt. Der User ist in diesem Konzept die entscheidende Autorität. Der Cap Desk kann diese Rechte aber in sogenannte *Granter* oder Powerboxen verpacken. In Abb. 12.3 sieht man unten, dass z. B. der Cap-Browser keinen direkten Zugriff auf Komponenten mit vollem Systemzugriff besitzt. Er kann lediglich auf eine für ihn zugeschnittene Powerbox zugreifen und über diese weitere Rechte bzw. Referenzen erhalten.

Dieses Muster wiederholt sich zwischen Renderer und Browser. Damit werden direkte Attacken auf höhere Komponenten verhindert und das granulare Bereitstellen von Rechten für untergeordnete Komponenten erlaubt.

Zwischen den Komponenten existieren Trust-Grenzen: Im DARPA-Browser darf es z. B. nicht möglich sein, dass der Renderer eine andere Seite zeigt, als der Browser im URL-Fenster anzeigt – dies wird durch die Garantien des Browsers

Abb. 12.3 Rechtevergabe durch den Cap-Desk [MS02]



explizit ausgeschlossen. Wenn also der Renderer eine neue Seite benötigt, muss er den Browser über den Granter darum bitten. In einem ersten Designentwurf konnte der Renderer einen String mit der betreffenden URL schicken und erhielt eine Capability auf eine http-Connection zurück (Abb. 12.4). Dies stellte sich schnell als fehlerträchtig heraus, da es dem Renderer die Möglichkeit gab, durch spezielle URLs (../..) illegale Zugriffe zu tätigen. In einem zweiten Entwurf entschieden sich die Entwickler deshalb, diesen Verstoß gegen den Capability-Grundsatz, dass Designation gleich Authority ist, zu beseitigen: Der Renderer erhält nicht mehr den Text einer Seite zum rendern, sondern einen bereits geparsen Ast, in dem URLs bereits durch ihre entsprechenden Capabilities ersetzt wurden (Abb. 12.5). Über diese kann der Renderer nun die gewünschten Seiten anfordern.

Die neue Version gestattet dem Renderer nicht mehr die Kontrolle über die angezeigte URL und nimmt ihm die Autorität, selbst http-Connections zu erzeugen. Eine zusätzliche Verschärfung des Designs erreichten die Entwickler durch die Entwicklung eines speicherlosen Renderers, d. h. für jede Seite wurde ein neuer Renderer gestartet. Das Design gleicht im zweiten Fall mehr einer Inversion-of-Control (IOC) Architektur, in der der Renderer nicht mehr eigenständig Verbindungsobjekte erzeugen kann, sondern sie hereingereicht bekommt.

An diesem Beispiel wird deutlich, dass eine Capability-basierte Architektur in der bewussten und eingeschränkten Übergabe von Referenzen besteht – wiederum ganz typisch für IOC-Architekturen. Deren Sicherheitspotenzial ist momentan noch kaum erkannt worden. Meist werden die Vorteile von IOC in Bezug auf Entkopplung und Wartbarkeit diskutiert.

Ein weiterer Vorteil der zweiten Version ist die Konzentration einer der gefährlichsten Operationen überhaupt – zumindest in der Denkweise der Capability-Vertreter – nämlich der Umwandlung einer Designation in eine Autorität (Referenz) innerhalb einer Komponente. Da diese Operation sehr anfällig für das „Confused Deputy“-Syndrom ist (d. h. Zugriff auf falsche Ressourcen mit voller Autorität,

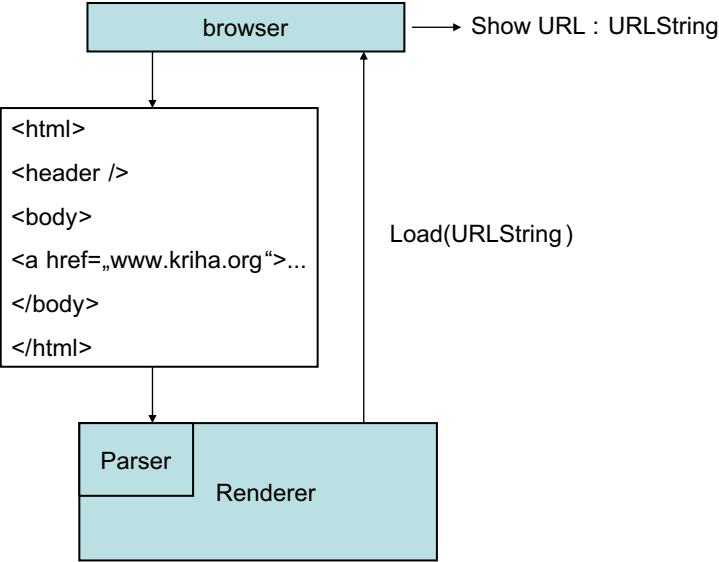


Abb. 12.4 Anfordern einer URL durch Übergabe der URL an den Browser als String

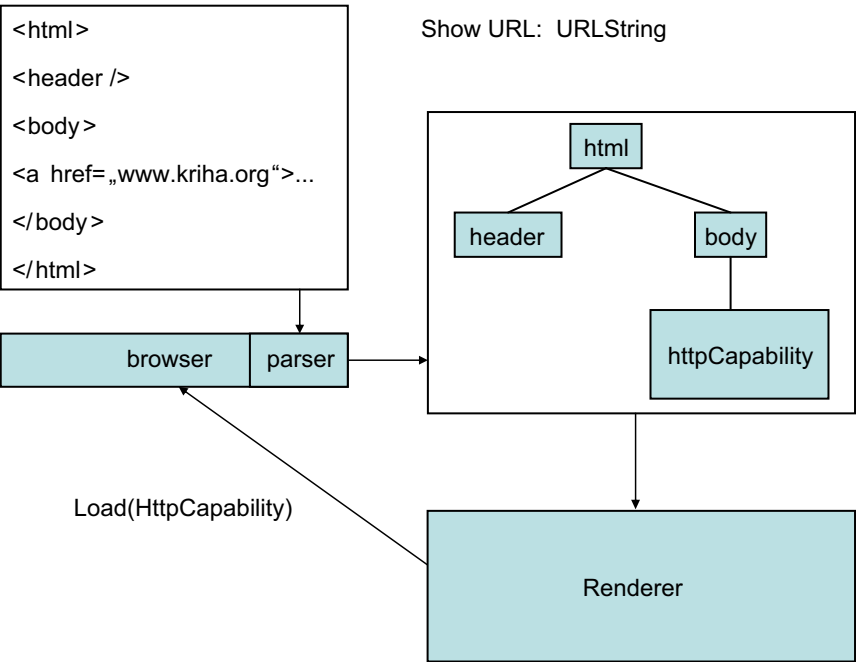


Abb. 12.5 Übergabe eines geparsen ASTs mit http-Capability an den Renderer

siehe [Hard]), sollte sie, wenn überhaupt, nicht wild über Module verstreut werden. Der Cap-Desk ist so eine hochkritische Komponente, die z. B. File-Namen in Referenzen umwandeln können muss. Nachgeordnete Module sollten deshalb bereits mit Capabilities arbeiten können und keine solche Umwandlung mehr benötigen. Kernproblem der Umwandlung ist ja, dass die Komponente, die sie bewerkstelligen muss, Zugriff auf viele Ressourcen benötigt, da ja unklar ist, welcher String als nächster umgewandelt werden muss.

12.2.3 Taming und Capability Architecture

Die Entwickler des DARPA-Browsers entschieden sich für die Verwendung von SWING als Toolkit für die grafische Benutzeroberfläche. Schnell entdeckten sie jedoch Probleme im Design des Toolkits, die dazu führten, dass beim Aufruf von Methoden mächtige, aber unerwünschte Referenzen für den Caller zurückgegeben wurden. Der Caller konnte dadurch seine Rechte erheblich erweitern.

Ein Beispiel für dieses Verhalten ist die Navigation von einem Panel zurück zum Parent Frame (und damit die Möglichkeit, dort Dinge zu ändern und zu fälschen). Um dennoch die Bibliothek verwenden zu können, wurde das Konzept des „Taming“ entwickelt:

Taming ist das Verpacken einer Bibliothek, die nicht im Sinne von Object Capabilities entwickelt wurde, in eine Schicht, die die überflüssige und gefährliche Autorität ausblenden kann. Leitlinie beim Entschärfen von Methodenzugriffen ist dabei die Frage: Bekommt der Aufrufer weitere Rechte durch einen Methodenauf-ruf? Und wenn ja, sind diese Rechte implizit bereits dadurch gedeckt, dass der

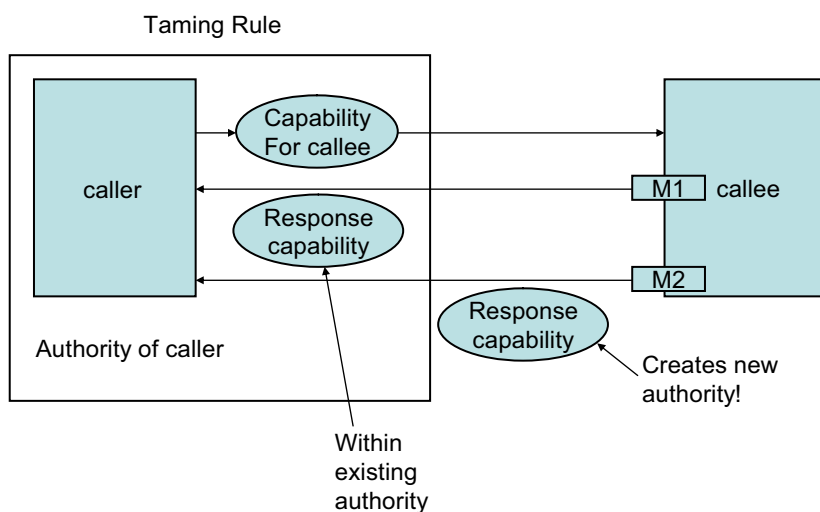


Abb. 12.6 Prinzip des Taming: Durch aufgerufene Objekte erzeugte Capabilities müssen schon vorher beim Caller vorhanden sein

Aufrufer das Recht zur Verwendung des Objektes besitzt, oder entstehen durch den Aufruf neue Rechte, wie in Abb. 12.6 die von dem gerufenen Objekt erzeugte Response Capability?

Häufig entdeckt man bei dieser Vorgehensweise auch grobe Designfehler in OO-Bibliotheken: Sollte man von einem File-Objekt tatsächlich zum Parent Directory und von dort in das gesamte File-System gelangen können? Um solche Probleme zu vermeiden, entstanden eine Reihe von Filtertechniken, wie sie auch in Werkzeugen wie Systrace verwendet werden: Einzelne Methoden wurden beispielsweise als verboten markiert. Bei anderen Methoden muss das Ergebnis in einen Sicherheitsbaustein (Facet) verpackt werden, der gefährliche Aufrufe ändert oder blockiert.

Die Gefahr beim Taming besteht darin, dass gefährliche Methoden übersehen werden können. Ein weiteres Problem ist: Wie soll man mit Global-Scope-Zugriffen verfahren? So ist z. B. `system.out` ein für alle immer offener Kanal – ein klarer Verstoß gegen POLA, denn viele Module werden diesen Kanal nicht benötigen. In der Sicherheitsanalyse traten dann auch gerade im Bereich der gezähmten Java-Bibliotheken größere Sicherheitslöcher auf, die jedoch schnell behoben werden konnten.

12.2.4 HTML

Ein weiterer sehr interessanter Aspekt des Browserexperiments ist die Kompatibilität von HTML mit einem Capability-Ansatz, der keine Zugriffe ohne Autorität erlaubt. Bestimmte HTML-Elemente wie das IMG-Tag zum asynchronen Laden von Bildern werden jedoch automatisch vom Browser ausgeführt – ohne Zustimmung des Benutzers.

Diese Eigenschaft des Browsers kann sowohl gebraucht (z. B. in Google Maps), als auch missbraucht werden, z. B. für Ein-Pixel-Bilder (sogenannte Web-Bugs) in E-Mails, die durch das Laden dem Sender verraten, dass die E-Mail angekommen und geöffnet wurde, also eine Verifizierung der E-Mail-Adresse. Bezeichnenderweise musste in E-Mail-Reader erst nachträglich eine Option zum Unterdrücken des Default-Verhaltens eingebaut werden, als sich herausstellte, dass HTML-Mail eben doch kein reines Textformat ist, sondern ambiente Autorität des Renderers ausnutzt.

Das FORM-Tag stellt eine Verbindung zu einem Server dar. Manche Browser geben eine Warnung aus, wenn Informationen per `submit` an diesen Server geschickt werden sollen, damit dem Benutzer der Akt der Veröffentlichung bewusst wird. Wie wir weiter oben bei der Besprechung der Attacks jedoch gesehen haben, ist dies in Zeiten von AJAX und der zeichenweisen, unmittelbaren Übertragung des Inhalts eine Warnung, die schon längst überholt ist.

Beliebige URLs können über dynamische Skripten ohne Zutun des Benutzers manipuliert werden. Der XLINK-Standard würde dies zwar ohne Scripting ermöglichen, würde aber an der grundsätzlichen Problematik nichts ändern: Es finden Zugriffe auf Ressourcen ohne Zutun des Benutzers statt.

Der DARPA-Browser unterbindet zwar solche Zugriffe, ist damit allerdings aber kein konkurrenzfähiger Browser. Das Problem, das es hier zu lösen gilt, ist: Wie sind solche automatischen Zugriffe generell so sicher zu gestalten, dass sich weitere Rückfragen erübrigen? Diese Problematik ist analog zu der des Taming von Source Code: Welche Methoden sind gefährlich, weil sie neue Autorität schaffen, und welche Methoden geben zwar neue Referenzen zurück, die jedoch keine Autorität tragen, die über die bereits vorhandene wesentlich hinausginge?

12.3 Sicherheitsanalyse

Angesichts der minimalen Rechte des Renderer-Subsystems müsste es trivial ohne Sicherheitsprobleme implementierbar sein. Dennoch traten im Laufe der Sicherheitsanalyse erstaunliche Angriffsmöglichkeiten auf, die meist auf unzulängliches Taming der Java-Plattform zurückzuführen waren.

Ein schönes Beispiel dafür ist die Erkenntnis, dass selbst die Eigenschaft eines Subsystems, ein Drop-Target zu sein, in Wirklichkeit eine wichtige Erlaubnis darstellt: Wenn der Renderer sich nämlich selbständig zum Drop-Target machen kann, kann er den Nutzer dazu verlocken, ein Dokument per Drag-und-Drop an den Renderer zu übergeben. Damit hätte der Renderer eine Möglichkeit, ohne den Browser Dokumente zu öffnen und zu rendern, und die vom Browser angezeigt URL wäre falsch.

Ein relativ schwieriges Beispiel ist hingegen die Untersuchung von Exceptions. Diese können Referenzen transportieren und eventuell Komponenten zur Verfügung stellen, die die entsprechenden Rechte normalerweise nicht besitzen. Das zeigt, wie sehr sich die Architektur von Objekt-Capability-basierten Systemen an manchen Stellen von der „normalen“ Denkweise der Entwickler unterscheidet. Allerdings ist das Problem der Exceptions mittlerweile auch bei Web-Applikationen erkannt worden. Dort unterdrückt man mittlerweile ebenfalls interne Informationen, die ansonsten über die Exception kommuniziert werden, um Angreifern keine Einstiegspunkte zu liefern.

Wir wollen hier aus den umfangreichen Sicherheitsanalysen zum DRAPA-Browser lediglich zwei Punkte herausgreifen: Die Methodik zur Erkennung von Autoritätsproblemen sowie die zugrunde liegenden Bedrohungsmodelle für den Renderer. Die Methodik, die Tribble und Wagner in [TW] verwenden, stützt sich in erster Linie auf eine statische Untersuchung von Erreichbarkeit unter der Annahme, dass nur die Autorität gefährlich ist, die auch erreichbar ist. In der Praxis bedeutet dies, die Call Chains der Aufrufe zu verfolgen und zu untersuchen, ob irgendwo eine Referenz zugänglich ist, die zu viel Autorität verschafft.

Das hört sich ähnlich aufwändig wie ein Durchsuchen des Source Codes Zeile für Zeile an. Die Mikro- und Makro-Architektur des Browsers sorgen jedoch dafür, dass es abgegrenzte Trust-Boundaries im gesamten System gibt: Wenn ein Modul bestimmte Referenzen bei der Initialisierung bzw. durch Parameter wäh-

rend der Laufzeit nicht erhält, dann kann es auf die Sicherheit dieser Objekte keinen Einfluss haben – es kann sie schlicht nicht erreichen. Durch das Vermeiden von globalen bzw. statischen Variablen, Registries etc. sowie durch die Proxy-Funktion der Power-Granter-Objekte entstehen isolierte, kalkulierbare Subsysteme, die den Aufwand für die Sicherheitsanalyse gering halten. Das bedeutet, dass in Zukunft Softwaredesign nicht nur Wartbarkeit und Erweiterbarkeit, sondern auch die Eignung für eine auch ökonomisch effiziente Sicherheitsanalyse garantieren muss. Dazu bilden garantierte Trust-Boundaries, wie sie durch Capabilities geschaffen werden, eine gute Möglichkeit.

Es gab im Wesentlichen zwei Bedrohungsmodelle für den Renderer: Ein isolierter, böartiger Renderer und ein Renderer, der mit einem böartigen Server zusammenarbeitet. Der isolierte Renderer könnte z. B. Informationen unterdrücken oder fälschen, beliebige Seiten besuchen etc. Natürlich könnte er auch DOS-Attacken fahren und das System über den Ressourcenverbrauch angreifen. Diese letztere Problematik wollen wir hier aber nicht weiter diskutieren, da wir das Quota-Problem sowie die momentanen Grenzen von Virtual Machines bezüglich Ressourcenschutz bereits in Kap. 4 angesprochen haben.

Weitaus interessanter sind natürlich kollaborative Attacken, bei denen der Renderer mit einer oder mehreren Sites zusammenarbeitet. Hier wird der DARPA-Browser durch mehrere Dinge beschränkt: Als ferngesteuerter Client eignet sich der Renderer sehr schlecht, da er kaum Zugriff auf Systemressourcen besitzt. An die Inhalte anderer Seiten kommt er ebenfalls kaum heran, da für jede Page ein neuer Renderer gestartet wird und somit keine Kommunikation der Pages untereinander stattfindet. Er kann auch nicht selbst URLs erfinden, und festcodierte URLs im Code kann er nicht verwenden, da sie nicht durch den Browser geparkt wurden. Er kann allerdings URLs in Seiten falsch darstellen. HTML lädt dazu durch das Design des Anchor Tags ein, wie dieses einfache Beispiel zeigt, das ja auch von Phishing-Mails gerne verwendet wird:

```
<a href=www.badguy.com> Deutsche Bank </a>
```

Die Sicherheitsanalyse ergibt unseres Erachtens ein klar positives Ergebnis für die Systemsicherheit. In Bezug auf semantische Attacken dürften Capabilities aber lediglich zur Schadensbegrenzung dienen können. Die semantische Lücke zwischen den durch Capabilities geschützten Ressourcen (Netzwerkverbindungen, Sites etc.) und dem konzeptuellen Modell des Nutzers in Bezug auf diese Ressourcen ist einfach zu groß. Eine Möglichkeit wäre jedoch, dieses konzeptuelle Modell des Nutzers systemseitig besser als bisher durch Objekte zu repräsentieren – dann könnte auch mehr mit Capabilities erreicht werden.

Was wäre in Zukunft nötig? Interessant wäre es, die verschiedenen bestehenden Ansätze zur Isolierung (Namespaces, Access Control, Capabilities) in Bezug auf ihren Beitrag zur Sicherheitsanalyse von Systemen zu untersuchen. Wo ist die Analyse ökonomischer und sicherer? Die Werkzeuge für die Erreichbarkeitsanalyse müssten verbessert werden, ähnlich denen, die zurzeit im Rahmen der Java 2

Security entstehen. Diese Werkzeuge müssten auch mit Sicherheitsbausteinen wie Facades umgehen können. Das Prinzip der „Capability Safety“ müsste klarer formuliert werden: Wann genau ist eine Komponente oder ein Objekt „capability safe“?

Schließlich muss die Weitergabe von granularen Rechten in Sprachen ohne Higher-Order-Functions bzw. Closures erleichtert werden. Inner Classes sind für diesen Zweck zu mühsam. Dynamische Proxies mit einem generellen `match (function, args)`-Mechanismus, wie in der Sprache E, wären hier besser geeignet [MS02].

Einen sehr Interessanten Ansatz verfolgt das Google-Caja-Projekt eines sicheren, auf Object-Capability-Grundlagen basierenden Javascript Subsets, das gefahrlos innerhalb von Seiten und im Zusammenhang mit Web2.0-Applikationen eingesetzt werden kann. Mark S. Miller et al. beschneiden Javascript, um spezielle Funktionen wie `eval()` und schränken die Verwendung von Konstruktoren oder Methoden ein [MSLAS].

Javascript hat in der Vergangenheit auch durch herstellerspezifische Erweiterungen sehr viel an Sicherheit verloren, siehe z. B. die Möglichkeit, eingekapselte Variablen über Erweiterungen der `eval()`-Funktion zu erreichen [Mich].

Gelingt das-Caja Projekt, dann steht neben dem Abschalten des Interpreters bzw. dem Neustarten bei jeder Page, wie es der DARPA-Browser macht, eine wirklich benutzbare Alternative zur Verfügung. Granular können dann dem Code von Web-Applikationen lokale Ressourcen des Users, aber auch auf Servern gehaltene Ressourcen, zur Verfügung gestellt werden. Das Paper von Miller et al. zeigt sehr schön auch die Grenzen der Identitätsbasierten Zugriffskontrolle im Falle von Browsern und Web-Applikationen auf.

Der kürzlich vorgestellte Google-Browser Chrome verwendet übrigens getrennte Javascript Interpreter pro Website – ohne dass Performanceproblem sichtbar würden. Dies deutet darauf hin, dass es weniger technische Gründe, sondern eher die Grenzen der Vorstellung der Entwickler waren, die die heutigen Browser in ihren Sicherheitsmechanismen so stark eingegrenzt haben. (<http://www.heise.de/newsticker/Google-Chrome-ueberholt-die-Konkurrenz--/meldung/115410>)

Die bisher genannten Punkte betreffen streng genommen nur die Schadensbegrenzung innerhalb von Browsern. Wenn diese Techniken eingesetzt werden, dann ist der nächste notwendige Schritt die Entwicklung von neuen User-Interfaces. Diese User-Interfaces müssen die in Zukunft immer häufiger und gefährlicher werdenden semantischen Attacken eindämmen. In Kap. 13 finden sich Beispiele für den Umgang mit Namen, Schlüsseln und Sicherheitstechniken, die konzeptuell auch für den Benutzer semantisch sehr wichtig sind.

Literatur

- [MS02] M. Miller, M. Stiegler, A Capability Based Client: The DarpaBrowser, Combex Technical Report,
<http://www.combex.com/papers/darpa-report/darpaBrowserFinalReport.pdf>
- [Mich] P. Michaux, Module Pattern provides no privacy, at least not in javascript™,
<http://peter.michaux.ca/article/8069>
- [MSLAS] M. S. Miller, M. Samuel, B.Laurie, I. Awad, M. Stay, Caja – Safe active content in sanitized Javascript, October 2007
- [Hard] N. Hardy, The Confused Deputy, or why capabilities might have been invented, *Operating Systems Review*, pp. 36:38, Oct., 1988,
<http://cap-lore.com/CapTheory/ConfusedDeputy.html>
- [TW] D. Wagner, D. Tribble, A Security Analysis of the Combex DarpaBrowser Architecture, <http://www.combex.com/papers/darpa-review/>

Kapitel 13

Security und Usability

“User interface design is every bit as important to security as the strength of the keys.”
(Mark Stiegler)

Die Benutzbarkeit von Techniken und Mechanismen in der Softwareentwicklung war bereits mehrfach ein Thema in diesem Buch. Sie spielt für die Sicherheit von Systemen bereits während ihrer Entstehung eine entscheidende Rolle. Schwierige Interfaces, unklare Verfahren und problematische Eigenschaften von Programmiersprachen haben fatale Folgen. Der Mensch mit seinen Defiziten muss daher in all seinen möglichen Rollen im Umgang mit Systemen (Entwickler, Administrator, Sachbearbeiter, Kunde etc.) als relevant für die Sicherheit wahrgenommen werden. Das Thema dieses Kapitels ist daher der Umgang mit Mechanismen der Sicherheit wie Schüsseln, Namen, Rechten und Absichten unter Berücksichtigung der Eigenheiten verschiedener Teilnehmer. Die Leitidee der vergangenen Jahre heißt dabei ganz einfach: „Nur Sicherheit, die funktioniert, ist Sicherheit“ und sie wurde von Garfinkel und anderen geprägt. Der Sicherheitsmechanismus, der zwar in der Theorie als sicher bewiesen ist, muss erst seine Tauglichkeit in der Anwendung innerhalb des Gesamtsystems Mensch/Maschine erweisen, bevor er akzeptiert werden kann. Und diese Tauglichkeit lässt sich empirisch prüfen. So ist z. B. das Key-Management eine der Ursachen „Why Johnny can’t encrypt“ (vgl. [WT]), und wir werden anhand der empirischen Untersuchungen von Simson Garfinkel und Peter Gutmann alternative Ansätze kennen lernen, die die Sicherheit der Beteiligten effektiv vergrößern.

Die Benutzung von Namen ist zwar einerseits unumgänglich, andererseits aber der Ansatzpunkt für semantische Attacken wie Phishing. Wir untersuchen deshalb die gängigen Namenskonzepte und zeigen Möglichkeiten auf, über private Namensräume sichere Verbindungen zu erstellen.

Rechte resultieren oft aus Eigentum. Wir werden das Verhältnis von Rechteaustübung (Discretionary Access) und dem Schutz des Systems sowie des Benutzers bei der Ausübung seiner Rechte untersuchen. Dabei werden wir feststellen, dass Viren und Trojanische Pferde die Sicherheitsmatrix des Systems nicht verletzen.

Im Anschluss untersuchen wir Techniken der Reduktion und fragen nach den Grenzen von Nutzerschnittstellen auf den momentanen Plattformen. In diesem Kontext erörtern wir auch die Rolle von Warnungen bei Benutzeraktionen: Wann sind sie sinnvoll, wann stellen sie nur ein digitales Analogon zu den in den USA verwendeten Farbcodes als Indikatoren für „tägliche Terrorgefahr“ dar?

Sichere Software muss die Absichten der Benutzer korrekt erkennen und sie auch korrekt ausführen. Bei der Entwicklung sicherer Software ist es deshalb nützlich, die konzeptuellen Modelle der Beteiligten zu kennen. Wir fragen nach Mechanismen, wie Benutzer ihre Absichten sicher kommunizieren können, z. B. durch klare Designation von Objekten. Aber selbst wenn Benutzer ihre Absichten klar kommunizieren, stellt sich die Frage, ob das System diese Absichten auch korrekt ausführt.

Eingestreut in diese Themen sind die Grundlagen eines sicheren User-Interface-Designs sowie Hinweise auf empirische Verfahren zur Prüfung von Verständnissen bezüglich Software und Aktionen.

13.1 Zum Verhältnis von Usability und Security

Über das Verhältnis von Usability und Security ist in der Vergangenheit viel Falsches gesagt worden. Dieser Abschnitt soll einige dieser Fehler korrigieren. Der erste Fehler wurde bereits in der Einleitung erwähnt: Usability ist für *alle* Beteiligten eines Systems von größter Bedeutung, und Defizite in den Techniken, Mechanismen, Programmiersprachen etc. wirken sich unweigerlich als Sicherheitsprobleme des Systems aus. Also muss nicht nur der Endbenutzer oder Kunde einbezogen werden, sondern auch die Entwickler, Administratoren und Verwalter von Lösungen. Usability hat viel mit konzeptionellen Modellen zu tun. Leider wurden bisher meist nur die Endbenutzer und eventuell noch die potenziellen Angreifer untersucht. In Kap. 2 hatten wir uns deshalb auch ganz bewusst mit der Wahrnehmung von Risiken durch Entwickler beschäftigt.

Der zweite Fehler besteht darin, Usability und Security als strikte Gegensätze aufzufassen. Wenn sich zwischen den beiden ein Graben auftut, dann liegt das eher daran, dass die Risikowahrnehmung nicht stimmt, z. B. weil die Darbietung des Risikos nicht verständlich ist. Oder die Warnungen sind in Wirklichkeit Alibifunktionen der Softwarehersteller, die den Benutzern gar keine Wahl ermöglichen. Wir werden dies am Beispiel von Windows untersuchen.

Der dritte Fehler ist, die Möglichkeiten der Usability zu überschätzen. Usability kann aus einem inhärent unsicheren System – wir können auch sagen, ein System, dessen Architektur keine Schadensbegrenzung kennt – kein sicheres System machen. Da helfen auch keine Orgien von Warndialogen – die Entscheidung des Users ist schlicht in jedem Fall gefährlich und die Folgen sind letztlich nicht von ihm abhängig. Ein Beispiel: Ein Kernkraftwerk benötigt eingebaute Sicherheit, die automatisch funktioniert. Die Isolation der Wirkungen voneinander ist es, die es dem User-Interface gestattet, gezielte Manipulationen vorzunehmen, deren Kon-

sequenzen klar definiert sind. Die Alternative wären Kraftwerksräume, die so voll mit Warnschildern wären, dass die Betreiber nicht mehr eintreten könnten. Der Zusammenhang zwischen Usability und Sicherheit eines Systems ist jedoch viel diffiziler. Ein schadensreduziertes System hat Autorität aus seinen Modulen entfernt –und zwar nicht nur überflüssige, ambiente Autorität, sondern im Idealfall jegliche automatisch vergebene Autorität. Eine geladene Software ist in so einem System zunächst machtlos und kann auch keinen Schaden anrichten. Es braucht also zugeführte Autorität, um Effekte erzielen zu können. Eine wichtige Quelle dieser Autorität ist der Benutzer.

Der vierte Fehler setzt die Sprache des User-Interface-Layers mit der Sprache des Systems gleich. Es zeigt sich, dass zwischen beiden im Allgemeinen eine enorme semantische Kluft existiert, die man so ausdrücken könnte: Konzepte, die dem System wichtig sind, finden keine verständliche Repräsentation innerhalb des User-Interfaces. Umgekehrt finden Konzepte der Benutzer keine Entsprechung innerhalb der Maschine. Ein Beispiel dafür sind E-Mailadressen: Für den Benutzer stellen sie wichtige semantische Objekte dar, das System hingegen behandelt sie als bloße Strings. Als weiteres Beispiel möge das automatische Update von Applikationssoftware dienen: Dies ist sicherlich ein sinnvoller Mechanismus, der jedoch von den meisten Systemen mit dem Recht auf einen TCP-Kanalzugang gleichgesetzt wird. Deshalb fragen sie den Nutzer: „Wollen Sie xyz den Zugang zum Netzwerk gestatten?“. Dabei sollte es doch um „Updates“ gehen, nicht um unbeschränkten Netzzugang – dies sind zwei grundverschiedene Dinge. Jeder Benutzer hat ein grobes Konzept von „Update“ (wir könnten es auch „Plan“ nennen) und versteht darunter wahrscheinlich, dass die Applikation vom Hersteller mit neuen Teilen der Software ausgestattet werden soll. Die auf dieses Konzept passende Frage „Wollen Sie die Applikation updaten?“ ist sehr wohl verständlich und auch vom Benutzer zu beantworten. Das Problem liegt nun auf der Seite des Software-systems: Es sollte diese Erlaubnis so umsetzen, dass es der Erwartungshaltung des Benutzers, der die Erlaubnis zugrunde liegt, entspricht. Es sollte also:

- Keine Nutzungsdaten oder Benutzerdaten an den Hersteller der Applikation übertragen.
- Den Update-Request, zusammen mit einer Übersicht der aktuellen Softwaremodule der Applikation, an die in der Installation angegebene Stelle schicken.
- Ebenfalls aus deskriptiven Installationstexten keine beliebige Kommunikation zwischen Applikation und Hersteller gestatten.
- Der neu geladenen Software keine Rechte automatisch zuordnen.
- Den Installationsprozess anhand von deskriptiven Anweisungen durchführen.

Ein solches Update-Verfahren entspricht der Erwartungshaltung des Benutzers und könnte – da harmlos – von diesem durchaus auch als permanentes Recht an die Applikation delegiert werden. Dinge, die vorher ungeheuer kritisch für die Systemsicherheit und mit entsprechenden Warnungsdialogen versehen waren, sind nun wesentlich unkritischer und können dem Benutzer klar kommuniziert werden. Dieser Effekt tritt generell in Systemen auf, die eine Reduktion von Autorität betreiben, wie beispielsweise SELinux, und er hat beträchtlichen Einfluss auf die

Usability dieser Systeme. Das bedeutet, dass das Systemdesign ganz andere Möglichkeiten der Interaktion erlauben kann – wenn es in der Lage ist, die Pläne des Users zu isolieren und während der Ausführung einzugrenzen.

Somit ist der fünfte Fehler beim Zusammenhang von Usability und Security darin zu sehen, die Möglichkeiten des User-Interfaces an den mangelnden Fähigkeiten der existierenden kommerziellen Systeme zu messen. Sichere Systeme brauchen ein anderes User-Interface und erlauben ein anderes User-Interface. Die Benutzbarkeit eines Sicherheitsmechanismus ist genau deshalb ein Problem, *weil* er als Sicherheitsmechanismus sichtbar geworden ist. Ein Gegenbeispiel bringt Peter Gutmann in [Gutm]: Man kann versuchen, die Passwörter eines Users möglichst gut gegen Phishing-Attacken zu schützen, z. B. durch Kontrollfragen bezüglich der Identität eines Servers („Wollen Sie wirklich an Server X Ihr Passwort schicken?“) – oder man kann einen Mechanismus zur Server-Authentisierung verwenden, wie TLS-PSK [RFC4279], der dem Server das Passwort des Users nicht verrät, sondern den Server mit einem Challenge-Response-Protokoll auf Basis des Passworts authentisiert – das Ganze im Rahmen des Aufbaus der TLS-Session erstellt. Der mögliche Fehler des Users bei der Identifizierung des Servers bleibt also folgenlos. Zusätzlich ist das Ganze transparent in ein Basisprotokoll eingebettet.

13.2 Lokale vs. fremdvermittelte Sicherheit

Das Problem der Herstellung initialen Vertrauens zwischen kommunizierenden Partnern ist sicher eines der größten im Bereich Security und soll aus technischer Sicht durch Zertifikate und Zertifikate ausstellende Stellen, die Certificate Authorities, erleichtert werden. Diese definieren Namen und binden sie an öffentliche Schlüssel. Leider sind Zertifikate, trotz ihrer großen Bedeutung für ihre Sicherheit, für die meisten Benutzer schwer zu verstehen. Zertifikatsbezogene Warnmeldungen, etwa von Browsern, werden deshalb häufig ignoriert.

Zertifikate stellen ein Beispiel für „fremdvermittelte“ Sicherheit dar: Sie wird von einer übergeordneten Autorität vermittelt, der Nutzer kann dabei nur in sehr geringem Maße Einfluss nehmen. Die meisten der in diesem Kapitel vorgestellten Techniken zum Schlüssel- oder Namensmanagement gehen im Gegensatz dazu von einer lokalen Sichtweise aus: Die Bedeutung eines Namens ist Benutzerspezifisch und *nur in Bezug auf einen Benutzer* eindeutig. Vertrauen wird normalerweise schrittweise aufgebaut und basiert auf der Historie von Interaktionen zwischen Partnern. Das kann beim Schlüsselmanagement ausgenutzt werden. Personen können sich lokale Namen und ihre Bedeutung gut merken – Rechner können sehr gut Schlüssel und Namen zusammen verwalten und wieder erkennen. Eine häufige Art, neue Namen zu lernen, besteht darin, dass die neue Person durch einen Bekannten vorgestellt wird („Darf ich vorstellen ...?“). Diese Einführungen sind lokal und verbinden Namen mit Bedeutung und Vertrauen – ganz ohne Certificate Authority.

Aufgrund der steigenden Zahl von Betrugsattacken auf Web-Nutzer mehren sich die Vorschläge für bessere externe Kennzeichnung von Sites, z. B. durch

Lokale Sicherheit

- Eigene Namensgebung für Kommunikationspartner (Petnames)
- Eigene Namensgebung für Objekte auf dem Rechner (Window-Namen)
- Die Historie einer Kommunikation mit anderen Partnern
- Die Historie eines Schlüssels
- Die Bestimmung eigener Orte für Objekte auf dem Rechner (Ortscodierung)
- Eigene Handlung zur Bestimmung einer Aktion
- Least Privilege Prinzip bei der Ausführung von Kommandos
- Bekannt werden über Freunde

Fremdenvermittelte Sicherheit

- Zertifikate die Schlüssel mit (nicht-eindeutigen) Namen verbinden
- Certificate Authorities deren Root Zertifikate riesige Mengen von Vertrauensbeziehungen herstellen
- Fremdbeurteilung von Sites durch andere Autoritäten mit Zuordnung von Farbcodes
- Signierter Code zum Download von Software
- Programme die das User Interface und seine Namensgebung bestimmen (Window-Title etc.)

Abb. 13.1 Lokale vs. fremdvermittelte Sicherheit

verschiedene farbliche Kennzeichnungen wie im Fall von Vista. Benutzer sollen dadurch einfache Indikatoren für die Sicherheit und Vertrauenswürdigkeit von Sites erhalten. Auch Oasis hat eine Arbeitsgruppe installiert, die Vorschläge zur besseren Darstellung des Security Contexts einer Aktion für den Nutzer erarbeiten soll. Diese Ansätze sind jedoch nicht ohne Gefahr: Was passiert, wenn solche „sicheren“ Indikatoren auf einer unsicheren Plattform verwendet werden, auf der sie manipuliert werden können? Die Manipulationsgefahr ist nicht das einzige Problem. Generell ist an diesen Ansätzen schlecht, dass das Vertrauen nicht lokal entstanden ist, sondern wieder durch externe Autoritäten vermittelt wird: Der Benutzer hat nicht über eine Reihe von Kommunikationen ein Vertrauen in eine Site aufgebaut, sondern das Vertrauen wird durch externe Maßnahmen hergestellt. Dadurch wird der Benutzer von Entscheidungen befreit – gerade an einer Stelle, wo eine bewusste Entscheidung nötig wäre.

Es ist somit interessant zu sehen, dass die Dichotomie zwischen lokaler Plattformensicherheit und Sicherheit durch Überwachung von außen (vgl. hierzu das Kap. 4) sich auch im Bereich der Usability wieder findet (Abb. 13.1).

13.2.1 Praktisches Key-Management

Seit Jahren beschäftigt die Frage nach der fehlenden Akzeptanz abgesicherter Nachrichten die Fachleute. Unter dem Titel „Why Johnny can’t encrypt“ gingen Whitten und Tygar [WT] in ihrer Studie der Usability von PGP dieser Frage nach. S. Garfinkel und andere haben kürzlich die Frage erneut aufgeworfen und empirische Untersuchungen zum Verschlüsseln von E-Mail durchgeführt [GSNM1, GSNM2, GM]. Die Ergebnisse waren ernüchternd: Die momentan verfügbaren

Verfahren werden von den Benutzern nicht verstanden und deshalb nicht eingesetzt. Als entscheidende Faktoren für diese Nicht-Akzeptanz fanden die Autoren die folgenden:

- Manuelles Schlüsselmanagement (Erzeugen und Verwalten von Schlüsseln) überfordert die meisten Nutzer.
- Zertifikate sind für die meisten Anwendungen nicht brauchbar und belasten die Benutzer nur unnötig.
- Sicherheit wird von den Entwicklern im Sinne „totaler“ Sicherheit verstanden. Da dies oft nur unter riesigem Aufwand realisierbar ist, bleibt man bei der scheinbar einzigen Alternative, keine Sicherheit anzubieten. Dabei lässt sich mit vertretbarem Aufwand der Grad der Sicherheit erheblich steigern.
- Die Historie von Kontakten bleibt gänzlich unberücksichtigt bei der Frage, ob einem Schlüssel vertraut werden kann.

In Garfinkels Beispiel wurde ein E-Mailsystem auf automatisches Schlüsselmanagement umgestellt. Dies führte zur Entlastung der Benutzer und sorgte gleichzeitig für die weitere Verbreitung von Schlüsseln. Das Stichwort dazu lautet *Key Continuity Management*: Die Entscheidung über die Akzeptanz eines fremden Schlüssels wird dabei zunächst dem Benutzer überlassen, die Verwaltung und Beobachtung des Schlüssels jedoch ab diesem Moment von der Maschine übernommen. Sollte plötzlich im Laufe der Zeit der Name einer bekannten Person, kombiniert mit einem anderen Schlüssel, auftauchen, erfolgt eine deutliche Warnung an den Benutzer, dass hier jemand wahrscheinlich eine Attacke versucht. Abbildung 13.2 (entnommen aus [Garf05]) fasst die Empfehlungen von Garfinkel zusammen.

- **Create Keys When Needed:** *Ensure that cryptographic protocols that can use keys will have access to keys, even if those keys were not signed by the private key of a well-known CertificateAuthority.*
- **Key Continuity Management:** *Use digital certificates that are self-signed or signed by unknown CAs for some purpose that furthers secure usability, rather than ignoring them entirely. This, in turns, makes possible the use of automatically created self-signed certificates created by individuals or organizations that are unable or unwilling to obtain certificates from well-known Certification Authorities.*
- **Track Received Keys:** *Make it possible for the user to know if this is the first time that a key has been received, if the key has been used just a few times, or if it is used frequently.*
- **Track Recipients:** *Ensure that cryptographically protected email can be appropriately processed by the intended recipient.*
- **Migrate and Backup Keys:** *Prevent users from losing their valuable secret keys.*

Abb. 13.2 Usability Guidelines für gute Security

Implementiert werden kann das Key Continuity Management im Falle von E-Mail durch vorgelagerte Proxies, die beispielsweise beim Senden automatisch den verwendeten Public Key in die ausgehende Mail einfügen sowie beim Empfang einen eventuell mitgelieferten Public Key des Senders registrieren (Abb. 13.3).

Interessant ist auch die Untersuchung der Rolle von Certificate Authorities, die die Autoren in diesem Zusammenhang durchführen. Ihre Hauptkritikpunkte an CAs sind die folgenden:

- Die Frage weltweit eindeutiger Inhabernamen in Zertifikaten ist ungelöst.
- Unklare Prozesse beim Ablauf und der Erneuerung von Zertifikaten über größere Zeiträume hinweg.
- Komplizierte Dialoge im Falle von fehlenden Root-Zertifikaten.

Auch andere Forscher haben die Frage aufgeworfen, ob die Sicherheit, die durch Root-Zertifikate tatsächlich erreichbar ist, wirklich den Aufwand an Warnungen und Dialogen rechtfertigt, mit denen der Benutzer konfrontiert wird, wenn ein Root-Zertifikat fehlt [Sti1].

Mittlerweile scheint sich auch bei der Absicherung von Infrastrukturen (Application-Server zu Application-Server) sowie bei den Entwicklungswerkzeugen (Eclipse Security Plug-Ins) die Erkenntnis durchzusetzen, dass das Schlüsselmanagement automatisch durchgeführt werden muss. So bietet der neueste Websphere-Application-Server die Möglichkeit, Schlüssel automatisch an Mitglieder der Cell zu verteilen, bzw. Public Keys bei der ersten Kontaktaufnahme zwischen Servern automatisch zu übernehmen [Botz06].

Auch im Bereich der Verwaltung von Infrastrukturen setzt sich mehr und mehr die Erkenntnis durch, dass SSL/TLS nur dann von Applikationsentwicklern per Default eingesetzt werden sollte, wenn die Schlüsselerzeugung und Verwaltung innerhalb einer Firma klar und einfach geregelt ist. Ein schönes Beispiel für eine

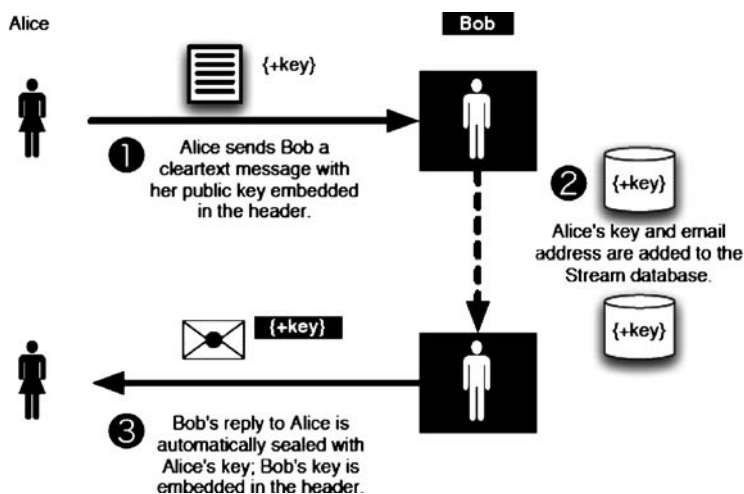


Abb 13.3 Automatisches Key Handling durch Proxies (aus [Garf05])

„schmerzlose“ Schlüsselerzeugung findet sich in [BDG] im Zusammenhang mit der Registrierung von neuen Laptops in einem drahtlosen LAN. Xerox Parc hat dazu eine Lösung („Network in a Box“) entwickelt, bei der der neue Mitarbeiter mit seinem Laptop einen geschützten Raum betritt. Dort wird per CD-ROM eine Software auf den Laptop geladen, mit deren Hilfe der Laptop sich an einem Access Point anmeldet. Von dort werden initiale Schlüssel auf den Laptop übertragen. Damit ist die Anmeldung und Konfiguration des drahtlosen Zugangs erledigt, ohne dass die Benutzer sich mit der für sie schwierigen Frage der Schlüsselgenerierung auseinandersetzen müssten – “the user is managing a small PKI without even realizing it”.

Schlüssel haben neben der Verständnisproblematik noch ein weiteres Problem: Sie sind generell für Menschen nicht merkbar. Die Benutzer gehen deshalb nicht direkt mit ihnen um, sondern verwenden Namen, mit denen sie die Schlüssel assoziieren. Aber Namen können unklar sein, doppeldeutig und fälschbar. Daraus entstehen Möglichkeiten für Manipulationen, wie das Phänomen Phishing als semantische Attacke klar beweist. Deshalb beschäftigt sich der folgende Abschn. 13.2.2 mit sicheren Namen.

13.2.2 *Namen und ihre eindeutige Verwendung*

Anscheinend ist es nicht möglich, Namen zu bilden, die sowohl global gültig, eindeutig, und trotzdem merkbar sind. B. Wilcox-O’Hearn hat für diese drei Dimensionen des „Namensraums“ das in Abb. 13.4 abgebildete Dreieck (entnommen aus [Sti1]) entworfen (erweitert und mit Overlays von Mark Miller).

Aus Abb. 13.4 lassen sich einige wichtige Erkenntnisse ableiten:

Nur Schlüssel können nicht-fälschbar sein. Ein Name hingegen kann jederzeit gefälscht werden. Dies wird beispielsweise an den gefälschten Absendernamen in jeder Spam-Mail deutlich.

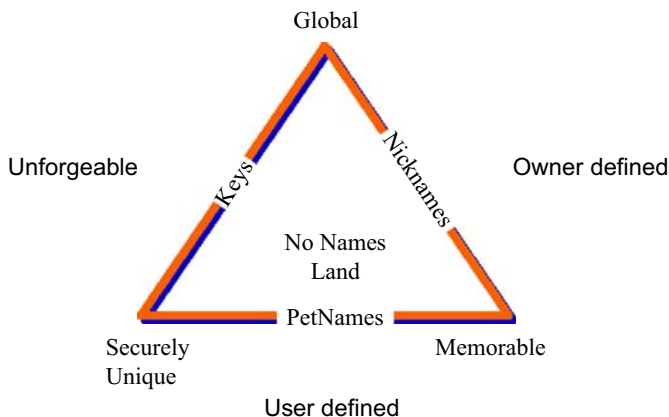


Abb 13.4 Dimensionen des Namensraums und ihre Beziehung zueinander

Namen können auch nachgeahmt werden, d. h. es können ähnlich klingende Namen erzeugt werden. Bei Hostnamen sind dies z. B. `www.bankx.com` bzw. `www.bankx-systeme.com`. Die Benutzer stehen ratlos vor die Frage, ob beide Domain-Namen etwas miteinander zu tun haben oder nicht (und ob sie dem Zertifikat, das `www.bankx-systeme.com` gehört, vertrauen sollen).

Aufgrund der möglichen globalen Vieldeutigkeit von Namen sind Namen und Schlüssel nur schwer miteinander zu kombinieren. Namen können zwar eindeutig sein – aber nur lokal, d. h. im Namensraum eines Benutzers (der Name „Müller“ in meinem Adressbuch ist für mich eindeutig.). Wenn Namen in einem lokalen Namensraum doppelt auftauchen, kann man sie durch private Namen (*Petnames*) auseinander halten. Nicknames hingegen werden nur von den Inhabern eines Namens bzw. Schlüssels vergeben.

Diese Erkenntnisse lassen sich nun in sogenannten Petname-Systemen ausnutzen. Dort wird ein Benutzer beim Eingang eines fremden Schlüssels und Namens aufgefordert, dem Schlüssel einen eigenen Namen zu geben – einen Petname. Dieser Name hat für den Benutzer eine eindeutige Bedeutung. Das Petname-System hat nun eine entscheidende Eigenschaft: Weder der Schlüssel selbst noch sein sogenannter *Nickname* – also der Name, den jemand seinem eigenen Schlüssel gibt, bzw. der Name, der öfters für eine bestimmte Entität (Firma, Person, Organisation) steht, werden dem Benutzer jemals präsentiert. Wenn eine signierte Mail empfangen wird, zeigt das Petname-System den Petname des Senders an, und damit, dass der Sender bereits bekannt ist. Fremde Schlüssel werden deutlich sichtbar als „unbekannt“ angezeigt. Der fremde Nickname spielt dabei keine Rolle bzw. verursacht eine Warnung, dass eine fremde Identität einen Nickname mit einem neuen Schlüssel versehen hat. Wichtig ist nur, dass der Benutzer vom Petname zum Schlüssel und zurück gelangen kann.

Der Begriff des „Nickname“, zu Deutsch „Spitzname“, ist im ganzen Petname-System wohl der am meisten verwirrende Begriff: Ein Nickname könnte z. B. `www.DieBankX.de` sein. Dies ist jedoch kein Spitzname, sondern eben ein offizieller Name. Das Entscheidende am Nickname ist aber, dass er vom Eigentümer des Schlüssels selbst und nicht vom Empfänger vergeben wird. Die Bedeutung des Namens (worauf er zeigt) ist also dem Eigentümer völlig klar – er hat ihn ja selbst vergeben. Diese Klarheit in der Bedeutung fehlt aber dem Empfänger des Namens unter Umständen, es sei denn, er definiert einen eigenen Namen dafür – eben den Petname. Das kann ruhig der gleiche Name wie der „offizielle“ Nickname sein, aber er ist in seiner Bedeutung für den Empfänger jetzt eindeutig.

Aus Sicht der Usability hat das System der nutzerdefinierten Namen auch seine Schwächen: Als Toolbar in Browsern oder Mail-Readern integriert, zeigt es an, wenn ein Kontakt (d. h. ein Schlüssel) bereits bekannt ist. Aber wie zeigt es an, dass ein Kontakt nicht bekannt ist (z. B. die Adresse eines Phishing-Servers)? Psychologisch besteht hier das Problem, dass wir das *Fehlen* einer Information weniger deutlich bemerken als ihr Vorhandensein.

Das folgende Beispiel (Abb. 13.5) demonstriert dies: Um sicherzustellen, dass ein Dialog, der den Kunden zur Eingabe von Kreditkarteninformationen auffordert, tatsächlich von VISA kommt, wurde das Konzept „Verified by VISA“

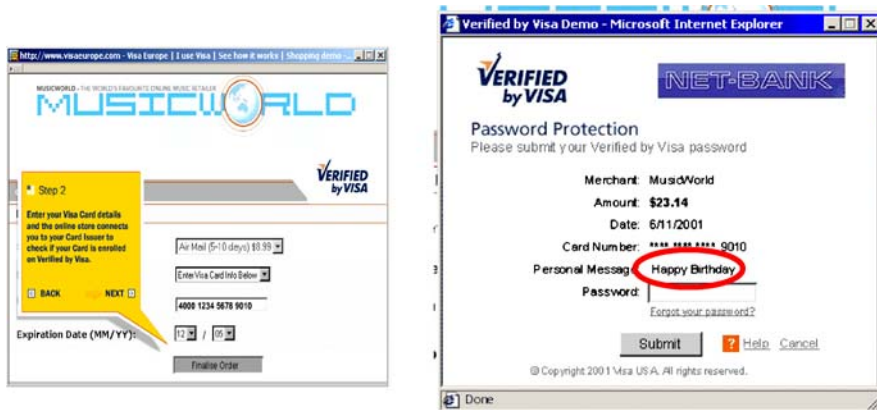


Abb. 13.5 Verified by Visa Demo (Screenshots)

[VISA] erfunden. Die Kunden können dabei eine nur ihnen bekannte Phrase bei VISA hinterlegen. Bei jedem authentischen Eingabedialog erscheint diese Phrase. Der Benutzer kann daraus schließen, dass bei der Erstellung des Eingabedialogs tatsächlich eine Verbindung zu VISA bestand.

Von den Möglichkeiten einer dynamischen Man-in-the-Middle-Attacke einmal abgesehen, hat dieses Verfahren noch eine weitere Schwäche: Der Benutzer muss im Falle eines Angriffs das *Fehlen* der Meldung bemerken und die richtigen Schlüsse daraus ziehen. Dies ist leider alles andere als selbstverständlich. In der Praxis werden die wenigsten Benutzer auf das Fehlen der Nachricht mit dem Abbruch der Transaktion reagieren, sondern bestenfalls an einen Fehler im System glauben.

Ka Ping Yee verwendet für die Erklärung des Phänomens das „Simon says“-Beispiel: Bei diesem in Amerika verbreiteten Kinderspiel muss ein Satz immer auf eine bestimmte Art beantwortet werden, es sei denn, im Satz fehlt ein bestimmtes Wort. Meistens plappern die Kinder jedoch auch dann die gewohnte Antwort, ohne das Fehlen des Schlüsselwortes zu bemerken.

Ein weiteres Problem der Petname-Systeme liegt darin begründet, dass das System dafür sorgen muss, dass vom Nutzer keine zu ähnlichen Petnames vergeben werden, die den Benutzer anschließend selbst verwirren könnten. Ab wann aber sind Namen ähnlich?

Die im Kap. 2 erwähnten Manipulationen von Domain Names durch Unicode-Buchstaben (Verwendung eines kyrillischen „a“ in E-Bay) werden durch Petname-Systeme bemerkt und könnten bei Feststellung der grundsätzlichen Ähnlichkeit zweier Domain Names für Warnungen verwendet werden.

Aus Sicht der Sicherheit bleibt das Problem bestehen, dass die Bindung eines Petnames an einen Schlüssel an die Entscheidung des Benutzers gebunden ist. Sie ist nicht durch Zertifikate gestützt und damit potenziell eine Sicherheitslücke. Untersuchungen haben jedoch gezeigt, dass es besser sein kann, diese grundsätzli-

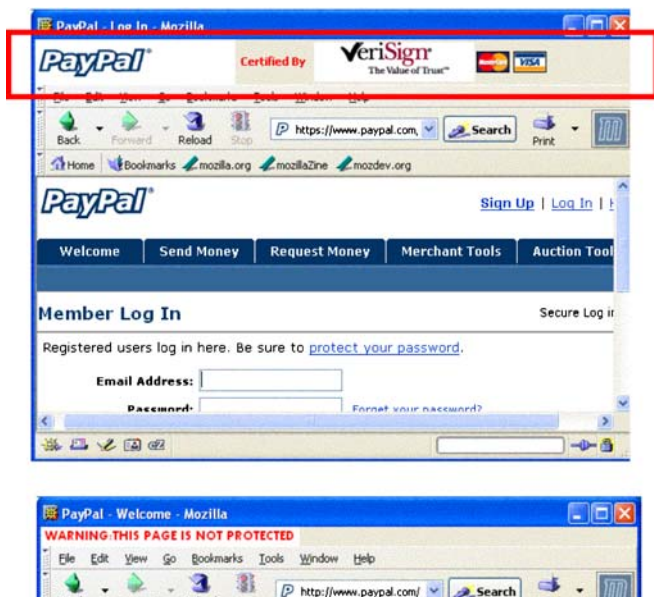


Abb. 13.6 Mozilla erkennt Paypal als „Trusted Site“ bei Verwendung eines Zertifikats (Screenshots)

che Entscheidung dem Benutzer bewusst zu überlassen und dafür die zusätzliche Komplexität von Zertifikaten zu vermeiden.

Zum Abschluss unserer Diskussion ein Beispiel eines Toolbar-basierten Petname-Systems, das zusätzlich versucht, das Konzept der Certificate Authority zu integrieren (Abb. 13.6) [Moz].

Die Mozilla Trustbar verwendet den offiziellen Namen auch als Petname. Die Einführung von CAs in das System erlaubt jedoch neue semantische Attacken, etwa durch gefälschte Zertifikats-Update-Meldungen.

13.2.3 Rechte und Eigentum

Die meisten Entwickler und auch Benutzer lassen sich recht schnell davon überzeugen, dass Programme oft mit zu vielen Rechten ausgeführt werden, also dass das POLA-Prinzip massiv verletzt wird.

Sobald jedoch von granularer Rechtevergabe die Rede ist, tauchen sofort große Bedenken aller (Entwickler wie Benutzer) über die Usability der Software auf. Neben der Frage der bequemen Benutzung – auf die wir gleich eingehen werden – schwingt hier möglicherweise noch eine andere Dimension mit, nämlich die Frage, wie Recht und Eigentum zusammenhängen.

13.2.4 *Discretionary vs. Mandatory – über Eigentum und Verfügung*

Die Frage, ob Discretionary-Access-Control-Mechanismen durch Mandatory Access Control ersetzt werden sollen, stellt sich für viele Entwickler zunächst nicht. Zu „normal“ erscheint es, die Tatsache des Eigentums an einer Ressource mit der kompletten Kontrolle darüber zu verbinden. Nicht umsonst heißt der Rechner heute meist „PC“, und hinter dem Wort „Personal“ steht die Vorstellung der kompletten Verfügungsgewalt über die Maschine. Hinzu kommt, dass der PC vom Design her generisch angelegt ist, also eine universelle digitale Rechenmaschine darstellt. Das impliziert die Fähigkeit, auch alles auf dieser Maschine tun zu können, d. h. im Prinzip muss jede beliebige Form von Software auf der Maschine laufen können.

Beide Konzepte – Eigentum und generische Maschine – führen zusammen zu dem Problem, wie der Benutzer vor der eigenen Macht geschützt werden kann. Kein leichtes Problem, aber auch längst nicht so weit hergeholt, wie es zunächst erscheinen mag. Momentan allerdings wird der Benutzer von der Software mit diesem Problem allein gelassen.

Wir haben bereits weiter oben angesprochen, dass in anderen Bereichen des täglichen Lebens die Begriffe „Eigentum“ und „(unbegrenzte) Verfügungsgewalt“ durchaus nicht immer identisch sind. Auf dem PC hingegen sind sie es meist – dort kann ein Doppelklick auf einen Mail-Anhang bewirken, dass eine fremde Software installiert wird – ein Beispiel für ein Auseinanderklaffen von *Intention* des Nutzers („Anhang öffnen“) und bewirkter *Aktion* („installiere Programm“), wie wir es weiter unten (Abschn. 13.2.6) noch besprechen werden. Gleichzeitig haben wir damit auch ein Beispiel dafür, wie ein kritischer Vorgang auf zu einfache Weise implementiert wurde.

Sehen wir uns an, wie die Trennung von Eigentum und Verfügung am Beispiel eines Fahrzeugs vor sich gegangen ist. Entscheidend ist dabei das Verstecken von Funktionen durch die Hersteller. Die Inhaber müssen (bzw. können) dadurch viel weniger schrauben und ein Fahrer braucht nicht mehr gleichzeitig ein Mechaniker sein – ganz anders als in den Frühzeiten des Automobils. Damit ist ohne Zweifel auch eine Entwertung von Wissen über Technik verbunden – dieses Phänomen sehen wir auch auf dem Gebiet der PC-Nutzung (siehe dazu [Step], der die Entwertung von Wissen über Computer durch moderne GUI-Techniken zeigt). Der heutige Fahrzeugmotor lässt keine „Schraubereien“ mehr zu, da er stark gekapselt ist. Auch in lizenzierten Autowerkstätten erfolgt die Reparatur eines Fahrzeugs häufig durch ein Software-Update oder das Austauschen eines ganzen Moduls. Das hat weitere Konsequenzen: Durch die zunehmende Digitalisierung der Produkte drängen die Hersteller nicht markengebundene Werkstätten aus dem Markt. Gleichzeitig ist aber die Bedienung von Fahrzeugen leichter und sicherer geworden.

Es scheint, als hätten wir im Falle des PC oder anderer Rechnerplattformen die richtige Trennlinie zwischen Eigentum und Verfügung noch nicht gefunden. Digital Rights Management (DRM) ist sicher ein Schritt in diese Richtung, glänzt aber

momentan durch massive Defizite in der Usability – ein Fahrzeug, dessen Sicherheitskonzept verhindert, dass die Ehefrau oder ein Freund damit fahren kann, würde ebenfalls nicht akzeptiert werden.

Welche Konsequenz hat die Trennung von Eigentum und Verfügung nun für den Bau von Softwaresystemen? Wenn man POLA als entscheidendes Grundprinzip der Architektur annimmt, eigentlich keine. POLA beschäftigt sich mit der Schadensbegrenzung bei der Ausführung von Code – trifft aber keine Entscheidungen über die Rechte von Personen. Die jeweiligen Rechte einer Person werden durch POLA lediglich entsprechend der gewollten Aktion eingegrenzt. POLA und rechtebasierte Zugriffskontrolle können also als unabhängige Prinzipien angesehen werden.

13.2.5 Bequemlichkeit vs. Sicherheit – der richtige Gegensatz?

Zu Beginn dieses Abschnitts ein Zitat:

“At the heart of the apparent conflict between security and usability is the idea that security makes operations harder, yet usability makes operations easier. Although this is usually true it’s imprecise. Security isn’t about making all operations difficult: it’s about restricting access to operations with undesirable effects. Usability isn’t about making all operations easy, either: it’s about improving access to operations with desirable effects. Tension between the two arises to the extent that a system is unable to determine whether a particular result is desirable. Security and usability come into harmony when a system correctly interprets the user’s desire.” [Yee02a, S. 245]

Erstaunlicherweise tritt der Effekt, dass eine feingranulare Rechtevergabe mit dem Argument einer daraus resultierenden schlechten Usability abgelehnt wird, sogar unmittelbar nach einer recht anschaulichen Darstellung der Gefahren durch Viren und Trojanische Pferde und ihrer Ursachen auf – so zu sehen in einem Video über das SkyNet-Virus von Mark Stiegler [Sti2]. Dort wird sehr klar gezeigt, dass es die Verstöße gegen POLA sind, die Angriffe durch Viren erst ermöglichen.

Der Grund für diese Ablehnung feingranularer Rechte dürfte darin zu suchen sein, dass die Benutzer bereits einem Übermaß an sinnlosen Warnungen und Aufforderungen für Bestätigungen ausgesetzt waren, deren Sinn und Zweck ihnen unverständlich blieb bzw. deren Inhalt ihnen nichts nutzte. Beispiele hierfür dürfte jeder PC-Nutzer zuhauf kennen: Erhält man eine Mail mit einem Office-Dokument im Anhang von einem Vorgesetzten – was nützt dann der Hinweis auf die mit dem Öffnen des Anhangs verbundenen Gefahren? Was nützen Hinweise des Browsers auf fehlende Root-Zertifikate oder unbekannte Zertifizierungsstellen, wenn der Sinn vom Nutzer nicht verstanden wird? Welche wirklichen Alternativen zum OK-Button gibt es? Beim Benutzer bleibt die Frustration über unsinnige, von ihm erwartete Aktionen zurück, die doch nichts anderes bewirken, als im Schadensfall die Schuld dem Nutzer zuschieben zu können. Hier wird vom Nutzer verlangt, eine globale Autorität zu vergeben, die im Fehlerfall fatal wird.

Ein anderes Beispiel für problematische Sicherheitswarnungen bilden Personal Firewalls mit ihren Warnungen in Bezug auf abgehende Requests: Die meisten Applikationen verwenden heutzutage das Internet, entweder für Software-Updates oder zum Nachladen von Daten. Den Sinn eines konkreten Applikations-Requests auf der Ebene des Netzwerkes wieder zu rekonstruieren, ist aber extrem schwierig. Hier gilt es in Zukunft anders zu denken:

- Ist die Erlaubnis einer Internetkommunikation an sich schon ein Sicherheitsproblem oder nicht eher das Fehlen wirksamer Kontrollen auf anderen Ebenen (laden von DLLs etc.)?
- Was bedeutet eigentlich ein OK-Button? Wie können wir sicherstellen, dass sowohl der Nachweis eines echten Nutzers (Liveness) erfolgt, als auch die richtige Autorisierung zum Ressource-Zugriff ausgesprochen wird? Das Prinzip des Power-Granters bzw. des Cap-Desk verlangt, dass Programme, die Ressourcen benötigen, sie nur von diesen über entsprechende Requests erhalten können (z. B., indem sich eine Dropzone öffnet und der Nutzer das gewünschte Objekt dort hineinzieht; von dort kann es als Capability an das Programm gehen).
- Die Programme selbst müssen Requests für Capabilities stellen. Das bedeutet, sie müssen eine formulierte Anfrage an den Capability-Verwalter stellen, der dann den Nutzer benachrichtigen kann. Dazu muss eine Begründung angegeben werden. Das Programm kann bei der Begründung natürlich lügen – allerdings entstehen durch solche Techniken schnell rechtliche Probleme für die Softwarefirmen, zumal die Anfragen und der nachfolgende Datenverkehr geloggt werden können.

Bei der Kommunikation eines Programms mit dem Nutzer muss klar unterschieden werden, ob echte Autorität verlangt wird oder ob nur eine „gefährliche Aktion“ ausgeführt werden soll. Gefährlich ist die Aktion jedoch im Allgemeinen nur aufgrund des schlechten Designs der Software und nicht inhärent aufgrund der Aktion.

Ein klassisches (positives) Beispiel stellt das im Kap. 4 erwähnte Polaris dar: Wird ein Office-Dokument in den Bereich eines fast rechtlosen Nutzers kopiert und dort bearbeitet, erübrigen sich allfällige Sicherheitswarnungen: Aufgrund des POLA-basierten Designs vom Polaris ist der potenzielle Schaden minimal. Die Interaktion der Software mit dem Nutzer steht hier also in direktem Zusammenhang mit der Sicherheitsarchitektur der Software selbst.

Wir können folgern, dass das Spannungsfeld von Usability und Security auf POLA und Nicht-POLA-Systemen zu ganz verschiedenen Ergebnissen und Techniken führt. Da die große Mehrheit der Softwaresysteme heute nicht auf POLA basiert, ist klar, dass die Usability-Forschung auf POLA-Systemen noch in den Kinderschuhen stecken muss. Leider übertragen Nutzer wie Entwickler ihre negativen Erfahrungen mit Nicht-POLA-Systemen auch auf jede andere Form von Software.

13.2.6 Erkennen und Ausführen von Intentionen

Fassen wir noch einmal die Erkenntnisse aus dem obigen Abschnitt zusammen:

- Auf unsicheren Plattformen funktioniert Malware entsprechend dem Design des Systems – Malware ist also kein Systemfehler.
- Der Schluss, dass Malware deshalb ein Problem des Benutzers ist, und die Gefährdung des Nutzers über das GUI abgefangen bzw. verringert werden muss, ist dennoch falsch: Die Plattformen haben einen massiven Designfehler, nämlich die fehlende Unterstützung von POLA. Das ist eben kein Benutzerproblem.
- Eine unsichere Plattform führt zu unsinnigen Warnungen, die mit ihrem Übermaß nur abstupfen.
- Diese Warnungen verschlechtern die Usability grundsätzlich und führen zu einer Ablehnung von granularer Rechtevergabe durch Benutzer.
- Ein POLA-basiertes System macht viele Warnungen überflüssig, weil es die Gefahren von Aktionen automatisch einschränkt.
- Erst in einem POLA-basierten System sind die Aktionen eines Nutzers als Akte ernst zu nehmen, mit denen wirklich Autorität ausgeübt wird. Deshalb ist ein POLA-basiertes System in besonderem Maße auf ein exzellentes User-Interface angewiesen, denn nur dadurch kann der Nutzer eines POLA-basierten Systems granulare Autorität an Programme und Komponenten vergeben, die per Default über keine Rechte verfügen.

Also spielt erst in einem POLA-basierten System die Usability tatsächlich eine entscheidende Rolle für die Sicherheit – so wichtig wie die verwendeten kryptografischen Schlüssel, wie Stiegler richtig anmerkt.

Abbildung 13.7 zeigt im Systemdiagramm den Zusammenhang von Intention, Programm und Effekt.

Nach [Yee02a] zerfällt die Ausführung einer Aktion durch den Benutzer in zwei Teile: Kommunikation zwischen Nutzer und Programm sowie Übertragung von Autorität zum Erzielen von Effekten. Der Kommunikationsteil beschäftigt sich mit der Semantik von User-Interfaces, ihren Begriffen und den angebotenen Aktionen. Sind die Begriffe verständlich? Sind die Aktionen ausreichend unterschiedlich, um alle nötigen Fälle abzudecken? Lässt das User-Interface dem Nutzer die Möglichkeit, seine Intentionen auszudrücken? Und nicht zuletzt: Ist die Kommunikation zwischen Benutzer und Programm authentisch, oder versucht jemand, dem Benutzer falsche Programme unterzuschieben, deren Aktionen er autorisieren soll? Der Betrug, der in der Kommunikation zwischen Benutzer und Programm auftreten kann, ist jedoch nicht der einzig mögliche. Selbst wenn der Nutzer seine Intentionen korrekt an das Programm weitergeben hat, ist noch lange nicht sichergestellt, wie diese Intentionen vom Programm in Effekte umgesetzt werden. Anders gefragt: Macht das Programm tatsächlich das, was im Nutzer-Interface versprochen wird?

intentions

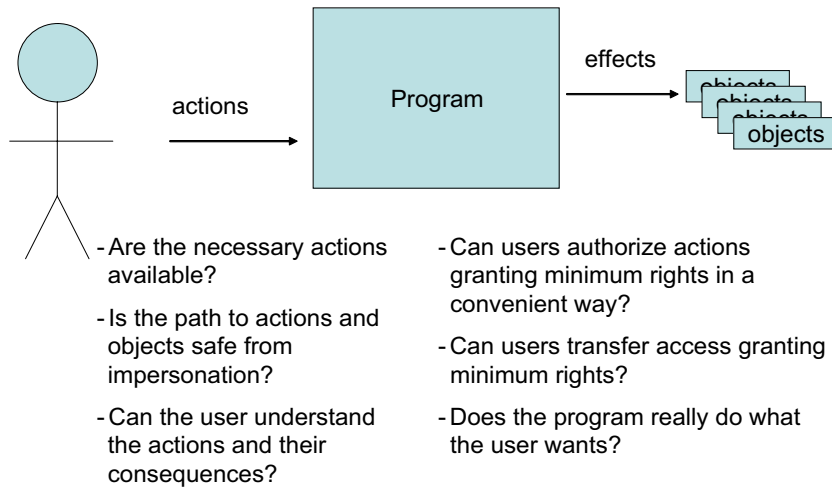


Abb 13.7 Zusammenhang von Intention, Programm und Effekt

Viren und Trojanische Pferde sind hier typische Beispiele für diese Art von Betrug am Benutzer. Garfinkel [Garf05] deckte jedoch noch eine ganze Reihe viel subtilerer Differenzen zwischen Intention und Effekt auf: Bei der Untersuchung von 250 gebrauchten Festplatten stellte er fest, dass die meisten noch wertvolle und kritische Daten ihrer vorherigen Eigentümer enthielten: Daten, die sowohl „Delete“-Befehle als auch „Format“-Anweisungen für die Festplatte unbeschadet überstanden hatten. Offensichtlich wird also die Anweisung „löschen“ von Seiten eines Benutzers auf den heutigen Systemen nur scheinbar und nicht im Sinne der Intention des Nutzers befolgt. Daten bleiben in temporären Verzeichnissen, Caches, Logs und Traces etc. zurück und werden selbst beim Formatieren einer Festplatte nicht gelöscht. Benutzer in Internetcafés erleiden dadurch einen beträchtlichen Verlust an Privatsphäre und Sicherheit.

Eine weitere Untersuchung der Datenspeicherung auf Windows-Maschinen im Zusammenhang mit der Frage nach der Sicherstellung der Privatsphäre wurde von M. Caloyannides [Cal] durchgeführt. Das Ergebnis war, dass es auf Windows-Maschinen kaum eine Möglichkeit gibt, einmal gespeicherte Daten spurlos wieder zu entfernen. Der Autor listet unzählige Speicherplätze innerhalb von Applikationen und Systemen auf, in denen sich scheinbar entfernte Daten und Datenspuren wieder finden.

13.2.7 Psychologische Voraussetzungen der User-Interaktion

In Kap. 2 wurden bereits einmal psychologische Theorien zur Erklärung komplexer Phänomene im Bereich der Risikoeinschätzung von Entwicklern verwendet.



Figure 2: The user has clicked on a button, we'd better pop up a warning dialog



Figure 3: What the previous dialog is really saying

Abb. 13.8 Entwicklersicht vs. User-Sicht nach [Gutm, S. 21ff.]

Konkret wurde die Outrage- sowie die Prospekt-Theorie eingesetzt, um das Verhalten gegenüber XSS-Schwächen in der Software zu verstehen.

Peter Gutmann hat Konzepte der Psychologie, die wesentlich den Umgang von Benutzern mit Sicherheitsmechanismen prägen, in seinem Buch „Security Usability Fundamentals“ zusammengetragen [Gutm]. Softwareentwicklern ist dieses Buch (bisher nur als Draft verfügbar) sehr ans Herz zu legen. Wir wollen hier nur einige der wichtigsten Grundkonzepte des Verhaltens erwähnen und verweisen ansonsten auf [Gutm]. Der Autor verwendet zur Darstellung der unterschiedlichen Sichtweisen von Entwicklern/Security-Experten und normalen Benutzern gerne folgende Methodik: Ein Sicherheitsdialog wird einmal wie von den Entwicklern entworfen dargestellt, und anschließend aus der Sichtweise der Benutzer betrachtet (Abb. 13.8). Dadurch wird z. B. deutlich, dass, während der Dialog für den Entwickler auf ein Problem außerhalb des GUI hinweist (Privacy), für den Benutzer der Dialog selbst das Problem ist. Wenn Gutmann dann die Sichtweise des Benutzers durch empirische Daten belegt und so z. B. fast 100% der Sicherheitsdialoge zum Einsatz von PKI als sinnloses Kauderwelsch für die meisten Benutzer entlarvt, dann bleibt dem Leser das Lachen im Halse stecken.

Gutmanns Ansatz wird deutlicher, wenn man ihn mit einem anderen Problem im Grenzbereich von Informatik und Psychologie vergleicht: Objektorientierung und menschliche Kategorisierung. Objektorientierung war lange Zeit einfach als Übung in „korrekter“ Kategorisierung gesehen worden, und Kategorisierung selbst als Übung in Logik und Sprache (Wittgenstein: Was definiert ein Spiel?). Erst als

sich empirische Psychologen wie Rösch und Lakoff [Lako] daran machten, empirisch zu untersuchen, wie Menschen tatsächlich kategorisieren, tauchten die riesigen Unterschiede auf. Heute sehen wir die Kategorisierung in der Objektorientierung weniger unter dem Aspekt genereller Richtigkeit, sondern beurteilen eine konkrete Hierarchie eher nach Kriterien wie Anwendbarkeit und Nützlichkeit, und sind uns ihrer Unvollkommenheit und Gebundenheit an die jeweilige Perspektive und den Verwendungszweck besser bewusst.

Im gleichen Sinn fragt Gutmann nicht danach, wie Menschen z. B. Sicherheitsentscheidungen treffen sollten, sondern danach, wie sie es konkret tun. Dieser Unterschied zwischen sollen und tun entpuppt sich schließlich auch als Unterschied zwischen denen, die die Interfaces bauen, und jenen, die sie verwenden müssen.

Das Buch von Gutmann geht in einem wesentlichen Aspekt über Studien zur Usability und Security, wie denen von Garfinkel und Yee, hinaus: Gutmann ist in der Lage, nicht nur die Fehler festzustellen, sondern er kann sie, zumindest teilweise, auf psychologische Basisbedingungen menschlichen Handelns zurückführen. Zu diesen Basisbedingungen gehören z. B. kognitive Tendenzen (Cognitive Biases), wie:

- *Confirmation Bias*: Wir suchen nach bestätigenden Hinweisen, nicht nach Dingen, die unsere Annahmen widerlegen. Dadurch übersehen wir Warnungen oder Ungereimtheiten leicht.
- *Blind Spot Bias*: Wir tendieren dazu, unsere eigenen Tendenzen und Vorurteile zu ignorieren.
- *Projection Bias*: Die Annahme, dass jeder so denkt wie wir. Der Projection Bias führt regelmäßig zu großen Überraschungen, wenn Benutzer oder Angreifer unsere Systeme ganz anders verwenden, als wir es gedacht hatten. Dieser Bias ist auch eine typische Schwäche der Internetprogrammierer, wenn sie das erste Mal eine Applikation für das Web bauen.
- *Personality Type Bias*: Nur ca. 7% der Bevölkerung denkt lt. Gutmann wie „Geeks“, d. h. eher technisch-mathematisch. Entwickler gehören jedoch häufig genau zu dieser Gruppe und sie bauen User-Interfaces für den Rest der Bevölkerung. Es gibt jedoch einen Trend, der das „Geektum“ langfristig begrenzt: Durch die Computerisierung und generelle Technisierung der Lebenswelt werden selbst Geeks angesichts von immer mehr und komplizierten Geräten zum Normalbenutzer ...
- *Pattern Recognition Bias*: Menschen sind in erster Linie darauf ausgerichtet, Muster wiederzuerkennen, weniger darauf, komplizierte Analysen durchzuführen. Das Erkennen bekannter Muster ist schlicht bequemer und möglicherweise unter Zeitdruck auch effektiver.
- *Subjective Validation Bias*: Menschen beziehen gerne generelle Aussagen auf sich. So glaubt ein Hypochonder nach dem Lesen eines medizinischen Artikels, sämtliche Symptome einer neuen Krankheit bei sich zu entdecken.
- *Rationalisation*: Wir erfinden gerne scheinbar rationale Erklärungen für Unbekanntes. Das erschwert es den Entwicklern von Sicherheitsmechanismen, auf Probleme hinzuweisen. Ungewöhnliches Verhalten wird wegrationalisiert.

- *Emotional Bias*: Emotionen wie Depression oder Optimismus bestimmen häufig den Ausgang von Entscheidungen und sind in der Lage, logische Überlegungen zu überstimmen.
- *Inattentional Blindness*: Das Übersehen von Dingen außerhalb unserer momentanen Aufmerksamkeit. Jeder Motorradfahrer kann diese Blindheit der Autofahrer gegenüber Nicht-Autofahrern leicht bestätigen. Leider erschwert sie auch die Wahrnehmung von sicherheitstechnisch bedeutungsvollen Dingen wie Warnungen.
- *Zero-Risk-Bias* bei Sicherheitsexperten: Der Trend bei Sicherheitsexperten, nur eine hundertprozentige Sicherheit zu akzeptieren. Da dies oft nicht erreicht werden kann, wird in der Folge gänzlich auf Sicherheitsmaßnahmen verzichtet. So sammelt ein Rechnersystem z. B. Fakten zum Verhalten der Benutzer und Kontaktpersonen. Aus dieser Historie der Kommunikation lassen sich bezüglich der Identifikation von Partnern Hinweise ableiten, ohne dass dies eine hundertprozentige Sicherheit darstellen würde. Sicherheitsexperten haben diese „minderwertige“ Sicherheit lange ignoriert. Garfinkel hat dies in seiner Thesis zu Usability und Security eindrucksvoll nachgewiesen und dabei das Konzept von „Sicherheit, die funktioniert, muss nicht absolut sein“ geprägt.

Zu diesen kognitiven Trends kommen weitere soziale, wie etwa Höflichkeit oder der Drang zum Teilen im Sinne eines sozialen Zements (Passwörter). Gutmann bringt hier das Beispiel des Offenhaltens von Türen für den nächsten bzw. den Tausch von Passwörtern oder Smartcards unter Kollegen. Nehmen wir z. B. die Urlaubszeit: Häufig geben Kollegen vor ihrem Urlaub ihre Credentials an andere Kollegen weiter, um ihnen die Chance zu geben, auf bestimmte Notfälle während ihrer Abwesenheit zu reagieren. Dieses Verhalten mag sicherheitstechnisch problematisch sein, es stärkt jedoch den Zusammenhalt in Teams, indem es das Vertrauen untereinander deutlich macht. Firmen, die den Austausch von Credentials auf keinen Fall akzeptieren wollen, müssen sicherstellen, dass die Arbeit als Stellvertreter softwaretechnisch sauber und ohne Austausch von Passwörtern möglich ist. Gleichzeitig dürfen Mitarbeiter bei der Verweigerung der Weitergabe nicht als unsozial angesehen werden. Wie schwierig das ist, zeigen die immer populärer werdenden Vereinzelungsanlagen. Meist haben diese Anlagen nach einem gescheiterten Zutrittsversuch eine Zeitbegrenzung. Wenn ein Mitarbeiter seine Ausweiskarte zwar durchgezogen, aber nicht schnell genug (oder zu schnell) den Mechanismus der Drehtüre betätigt hat, dann ist dieser Ausweis für einige Minuten gesperrt. Wenn dies einem Mitarbeiter einer Gruppe, die gemeinsam zum Mittagessen ging, passiert, wird im Allgemeinen ein anderes Mitglied seinen Ausweis durch einen Schlitz an den gescheiterten Kollegen geben, um ihm Zutritt zu gestatten. Rechtliche Drohungen helfen hier wenig: Ein anderes Verhalten des Kollegen wäre extrem unsozial, auch wenn es sicherheitstechnisch korrekt wäre. Sozial verträglich wäre es hingegen, wenn die Drehtüre grundsätzlich entweder auch den Austritt registrieren würde oder auch nach einem erfolgreichen Eintritt eine Zeitsperre erzwingen würde.

Diese „Biases“ beeinflussen unter anderem die Art und Weise, wie Menschen Entscheidungen treffen. Sicherheitsentscheidungen dürften wohl ohne Zweifel zu den wichtigsten dieser Entscheidungen gehören – zumindest nach Ansicht der Entwickler. Gutmann untersucht die Annahmen bezüglich menschlicher Entscheidungsfindung, die hinter den GUIs der gängigen Sicherheitsmechanismen stehen. Wenn Benutzern ein Dialog aus dem Bereich der PKI präsentiert wird, dann erwarten die Entwickler des Dialogs eine rationale, auf der Abwägung von Nutzen und Kosten beruhende Entscheidungsstrategie des Benutzers nach der „Subjective-Expected-Utility“-Theorie. Wie im Fall der oben erwähnten Kategorisierungsproblematik stellt diese aus der Ökonomie kommende Theorie jedoch eher ein „sollte“ dar denn ein „ist“. Gutmann stellt dieser Theorie empirische Erfahrungen ganz anderer Art gegenüber und zeigt, dass die menschliche Entscheidungsfindung durch ganz andere Dinge beeinflusst wird:

“The game-theoretic/economic approach to decision making is particularly problematic for security decisions because it treats a decision as a gamble, with some equivalent of a coin toss or die roll followed by immediate feedback on the value of the decision. Unfortunately security decisions don’t work this way: there’s no immediate feedback, no obvious feedback, and (since security failures are usually silent) there may never be any feedback at all, or at least not until it’s far too late (a phantom withdrawal from your bank account a year later) to take any corrective action.

This gambling/game-theoretic model does however perfectly model one aspect of user behaviour, the portion of the decision-making process that leads to dismissing warning dialogs. In this case the user gets immediate, strongly positive feedback on their decision: they can go ahead with their task. Other possible outcomes are unknown, and may never be known — was the phantom withdrawal the result of clicking ‘Cancel’ on a dialog, or because a credit card processor lost a backup tape, or because ...?” [Guttm, S. 41]

Zu den wichtigsten psychologischen Basismechanismen in Bezug auf Security-Entscheidungen der User gehören nach Gutmann:

- Satisficing: Das sprichwörtliche wilde Durchklicken durch Warndialoge.
- Conditioning: Dauerndes Eingeben von Passwörtern ohne Prüfung des Empfängers, Anwendung wahlloser biometrischer Authentisierung.
- Inattentional Blindness: Die Blindheit gegenüber etwas, das außerhalb der momentanen Aufmerksamkeit liegt.
- Watermark Fallacy: Der Glaube an das, was offiziell aussieht – im digitalen Zeitalter sind Markenzeichen etc. beliebig fälschbar.
- Not-My-Problem-Fallacy: Typisches Verhalten in einer Menge: Jemand wird schon etwas tun. Dieser Effekt ist im Internet besonders stark.
- Recognition Primed Decision Making: Der Benutzer nimmt unter Zeitdruck die Option, die er bereits kennt, d. h. Alternativen werden praktisch ignoriert.
- Superstition Based Decision Making: In komplexen Systemen entdeckt der Benutzer scheinbare Regeln.
- Framing: Die Entscheidung fällt aufgrund der schnellen Einordnung des Problems in einen Vorstellungsrahmen, der nicht der Richtige sein muss.

- **Overattention:** Bei genügend Zeit und Angst findet eine nicht mehr endende Evaluierung von Fakten statt, auch bekannt als Analysis/Paralysis Pattern.
- **Automatic Processes:** Der Benutzer entscheidet aufgrund eingefahrener Verhaltensweisen, ohne tatsächlich dem Problem seine Aufmerksamkeit zu widmen.
- **Nicht-Determinismus:** Menschen zeigen öfters nicht-erklärbare Verhaltensweisen in ihren Entscheidungen.

Wichtig für die Entwickler von Systemen ist bei dieser Aufzählung vor allem eines: Sie müssen begreifen, dass ihre Theorien zur Entscheidungsfindung der Benutzer höchstwahrscheinlich völlig falsch sind. Auf Basis von Tendenzen und Entscheidungsmodellen entstehen letztlich beim User mentale Modelle bezüglich der Sicherheit. Gut wäre es, wenn wir – ähnlich, wie es die Modeindustrie mit ihren Schneiderpuppen macht, die den verschiedenen Körpermaßen und -formen der Bevölkerung entsprechen – Modellpersonen hätten, die jeweils bestimmte Einstellungen und Fähigkeiten und damit Klassen von Usern bezüglich Sicherheit repräsentieren. User-Interface-Spezialisten behelfen sich damit, sich solche Modell-User während eines Designs vorzustellen, d. h. von verschiedenen „Personas“ auszugehen und das Design auf die Verwendbarkeit mit diesen zu prüfen. Häufig dominiert bei der Softwareerstellung aber noch der „Einheits-User“ (no sex, no age, no education, ...).

Kaum ein Softwareentwickler wird mentale Modelle in Bezug auf Sicherheit erstellen können. Was wir jedoch sicher tun können, ist, unsere Annahmen bezüglich der sicheren Benutzbarkeit unserer Systeme sowie speziell der Sicherheitsmechanismen während des Designs an gedachten Personas zu testen, sowie im Anschluss ihre Wirksamkeit einer empirischen Validierung zu unterziehen. Im Anschluss an die Resultate der psychologischen Grundlagen werden wir diesen Ansatz in Abschn. 13.2.8 darstellen.

Einige wichtige weitere Resultate der Untersuchungen von Gutmann:

- Erziehung von Usern funktioniert nicht. Benutzer wollen und können im Allgemeinen nicht zu Sicherheitsexperten erzogen werden. Sicherheitsüberlegungen sind ihnen nicht nur technisch fremd, sondern interessieren sie meist auch gar nicht.
- Typische Annahmen über User-Verhalten stimmen nicht. Ohne empirische Überprüfung der Verwendbarkeit und Akzeptanz kann kein Sicherheitsmechanismus als wirksam oder gültig angesehen werden.
- Warnungen konditionieren und verlieren ihre Wirksamkeit. Das gängige System der Warnhinweise und Dialoge ist praktisch völlig wirkungslos. Schlimmer, es konditioniert die Benutzer auf die generelle Missachtung von Warndialogen und schafft gefährliche Handlungsautomatismen.
- Auf Wiedererkennung basierende Verfahren sind gefährlich. Warenzeichenidentitäten funktionieren schlecht in der digitalen Welt, in der Symbole beliebig kopierbar und nachbaubar sind. Einfache Codierungen z. B. von Seiten nach Farben werden entweder nicht wahrgenommen oder schaffen gefährliche Automatismen, sobald sie von Angreifern nachgemacht werden.

- Verfahren, die darauf beruhen, dass das *Fehlen* von Informationen bemerkt wird, funktionieren schlecht.
- Die Repräsentation technischer Sicherheitskonzepte (Zertifikate, etc.) wird nicht verstanden. Das Verstehen ist jedoch eine der Basisvoraussetzungen für die Benutzung von Sicherheitsmechanismen, allerdings bei weitem nicht hinreichend. Dennoch werden die meisten in Sicherheitsmechanismen verwendeten Begriffe schlicht nicht verstanden und daher ignoriert (satisficing).
- Aktionsintegrierte Sicherheit funktioniert: Im Gegensatz zu Sicherheitsdialogen, die die Benutzer aus dem Kontext ihrer Arbeit reißen und gegen einige der genannten kognitiven Tendenzen arbeiten, wird ein im Kontext der Arbeit integrierter Sicherheitsmechanismus nicht als störend empfunden. Gleichzeitig kann dadurch eine bewusstere Auswahl von Alternativen erzeugt werden.

Die Studien von Garfinkel, Gutmann u. a. erklären mittlerweile recht gut, welche Sicherheitsmechanismen auch von Benutzern verstanden und verwendet werden können. Es stellt sich allerdings heraus, dass die funktionierenden Mechanismen eher selten sind. Wenn jedoch im Allgemeinen ein grobes Missverständnis auf Seiten der Benutzer existiert, das in vielen Fällen (PKI) auch durch noch so ausgeklügelte User-Interfaces nicht beseitigt werden kann, dann muss die grundsätzliche Frage nach dem Verhältnis von Usability und Security neu gestellt werden. Wenn Benutzer durch Sicherheitsdialoge in der Arbeit behindert werden, wenn sie durch sie konditioniert werden, dann liegt dies möglicherweise auch daran, dass die Sicherheitsentscheidungen grundsätzlich falsch entworfen sind. Vielleicht sind längst nicht alle Entscheidungen vom User zu treffen? So ist nur schwer verständlich, warum mit dem Öffnen eines Attachments einer Mail eine Sicherheitsentscheidung verbunden sein soll. Interessanterweise finden sich auch bei Gutmann Beispiele für die Verlagerung von Sicherheitsentscheidungen weg vom User und hin zur Maschine. Das beste Beispiel dafür ist die Verwendung von TLS-PSK, bei dem über ein Challenge/Response-Verfahren zwei Dinge erreicht werden: Eine grobe Authentisierung des Servers nach dem Aufbau des sicheren TLS-Tunnels, sowie die Authentisierung des Users gegenüber dem Server, ohne dass das Passwort des Users dabei verraten wird. Die Authentisierung des Servers gegenüber dem User besteht darin, dass auch er die Kenntnis des Passworts beweisen muss. „Falsche Verbindungen“, also der Aufbau von sicheren Tunneln zum falschen Endpunkt, werden so gleich beim Aufbau verhindert.

13.2.8 Messen von *Intention* – *Verständnisprüfung*

Es sollte heutzutage eine Selbstverständlichkeit sein, User-Interfaces auf ihre Benutzbarkeit hin zu prüfen. Zentraler Teil einer solchen Prüfung ist der Test des Verständnisses der angebotenen Informationen sowie die Bearbeitung typischer Aufgaben durch Testnutzer. Beides ist ebenfalls Teil der „User-Centered-Design“-

Methodik aus dem Bereich der Usability-Forschung. Nach dem Prinzip der Hypothesentests aus den Sozialwissenschaften werden dabei die Designelemente auf Verständnis und Anwendbarkeit geprüft. Dazu wird Testmaterial entwickelt, mit dem sich die Annahmen bezüglich Verstehen und Bearbeiten prüfen lassen. Anschließend wird das Testmaterial von einer Gruppe von Versuchspersonen bearbeitet. Die Ergebnisse werden analysiert und fließen wieder in das Design der Benutzerschnittstelle ein.

Untersucht man gängige User-Interfaces in dieser Art und Weise, zeigt sich bei den meisten Meldungen und Aktionen, die mit Sicherheit zu tun haben, dass die Sprache der Entwickler – die sich in den Dialogen niederschlägt – nichts mit dem Verständnis der Benutzer zu tun hat. Spezialausdrücke werden auf die Benutzer abgewälzt, die keinerlei Verständnis dafür mitbringen und sie auch nicht mit ihren Intentionen in Beziehung setzen können. Die Reaktion der Benutzer besteht dann üblicherweise im blinden Betätigen der OK-Buttons solcher Dialoge.

Ein Beispiel, das sich an dieser Stelle aufdrängt, ist das Verhalten des neuen Firefox-3.0-Browsers im Falle eines invaliden oder selbstsignierten Zertifikates. Firefox zwingt in diesem Fall die User zur expliziten Definition einer Ausnahme für eine solche Site und lässt sie nicht mehr einfach per Mausklick die Warnungen wegdrücken (Abb. 13.9). Zur Geschichte dieses Features gibt es übrigens sehr interessante Blog-Einträge, etwa <http://blog.johnath.com/index.php/2007/10/11/todo-break-internet/> oder <http://blog.ivanristic.com/2008/04/firefox-3-ssl-i.html>.

Es ist keine Frage, dass dieser Dialog den User zu mehr Aufmerksamkeit zwingt. Aber was wäre wohl das Ergebnis, wenn wir einen empirischen Test der Usability wie oben dargestellt durchführen würden? Wären die Begriffe verständ-



Abb. 13.9 Design des Dialogs für Probleme mit Serverzertifikaten in Firefox 3.0

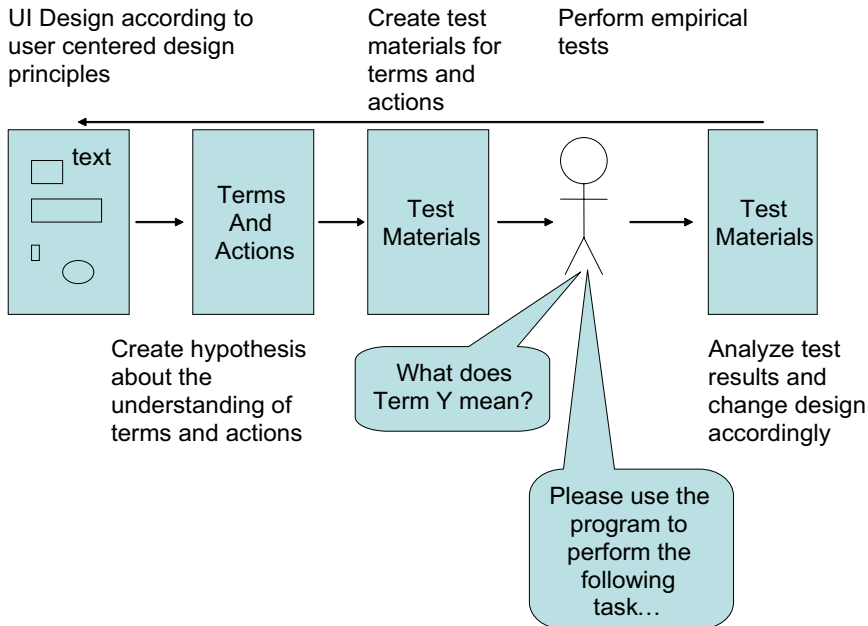


Abb. 13.10 Empirisches Testen der Benutzbarkeit anhand von Testmaterial zu Begriffen und Aktionen

lich? Könnte der User einen Plan entwickeln, wie er in Zukunft mit solchen Meldungen umgeht? Die Abbildung zeigt übrigens nur einen kleinen Teil des gesamten Dialogs, der durchaus noch mehr technischen Jargon für den Normal-User parat hat.

Bei sicherheitskritischen Applikationen wäre als nächstes der zweite Teil von Abb. 13.10 zu bearbeiten, das heißt, die Terme und Aktionen, deren Verständnis und Anwendbarkeit empirisch geprüft wurden, müssen jetzt in Bezug auf ihre wahrheitsgetreue Durchführung getestet werden. Im obigen Firefox-Dialog wäre das die Prüfung, ob bei einer Ablehnung durch den User tatsächlich keine Verbindung hergestellt wird und umgekehrt die Definition der Ausnahme funktioniert. Zu überlegen wäre weiterhin, ob es dabei eine zeitliche Komponente geben sollte – wie lange beispielsweise soll eine Ausnahme gelten? Wir müssen hier also genau prüfen, ob das konzeptuelle Modell der Benutzer bezüglich der Applikation mit der Wirklichkeit übereinstimmt und ob es sinnvoll ist.

13.2.9 *Autorität aus Designation*

Im Idealfall sind das Auswählen eines Objektes sowie die Bestimmung einer Aktion und die Vergabe der nötigen Berechtigung dafür (und nur dafür) ein einziger

Intention:

work on file „X“

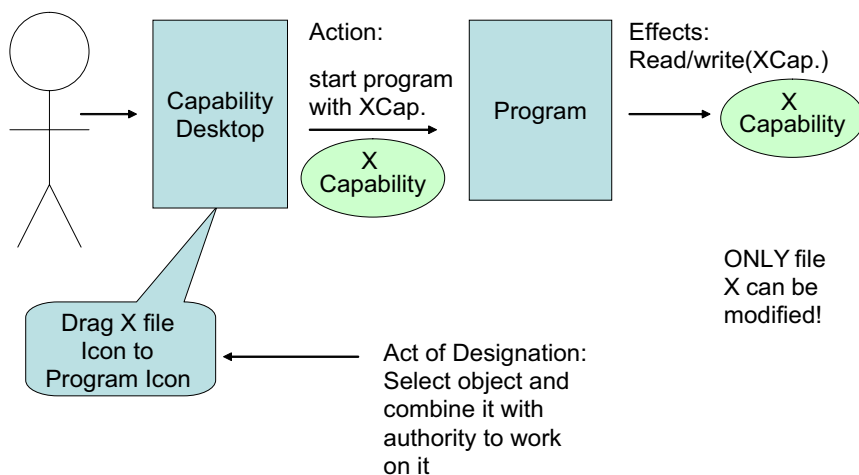


Abb. 13.11 User als Quelle der Autorität, granulare Weitergabe

Vorgang, *Designation* genannt. Die Benutzer machen dadurch deutlich, was sie erreichen wollen. Beispiele für Designation sind die Übergabe von Objekten an Programme durch Drag-and-Drop-Mechanismen (Abb. 13.11). Entscheidend aus Security-Sicht ist dabei, dass Programme Rechte nur auf die übergebenen Objekte erhalten, und zwar auf geöffnete „Handles“ oder „Capabilities“ der Objekte.

Ziel ist es, die Vergabe von Rechten bruchlos mit der Arbeitsweise des Benutzers zu integrieren. Was aber, wenn der nötige Mechanismus für die granulare Übergabe fehlt? Dann muss auf die interaktive Methode der Bestimmung von Objekten zurückgegriffen werden: Das Programm öffnet eine Dialogbox und fragt nach einem Namen eines Mail-Empfängers. Alternativ kann ein Abstraktionsmechanismus genutzt werden, der Mail-Adressen wie Filehandles behandelt.

Aus technischer Sicht müsste eine Re-Identifizierung von Objekten auf Systemebene (statt nur innerhalb von Applikationen) stattfinden. Momentan findet die Umsetzung eines Namens in ein Objekt erst innerhalb der Applikation statt. Betriebssysteme müssten deshalb zentrale Nutzerkonzepte wie Mail-Adressen als „First-Class-Objects“ realisieren.

13.2.10 Sichere User-Interfaces

Ka-Ping Yee hat in „Secure Interface Design“ [Yee02a] zehn Grundregeln für den Entwurf sicherer User-Interfaces formuliert. Die Spoofing-Attacken im Bereich der Browser-Interfaces haben gezeigt, wie weit die Applikationen momentan da-

von entfernt sind. Wir zitieren hier seine Regeln, die im wesentlichen für sich sprechen:

- Path of Least Resistance: Match the most comfortable way to do tasks with the least granting of authority.
- Active Authorization: Grant authority to others in accordance with user actions indicating consent.
- Revocability: Offer the user ways to reduce others' authority to access the user's resources.
- Visibility: Maintain accurate awareness of others' authority as relevant to user decisions.
- Self-Awareness: Maintain accurate awareness of the user's own authority to access resources.
- Trusted Path: Protect the user's channels to agents that manipulate authority on the user's behalf.
- Expressiveness: Enable the user to express safe security policies in terms that fit the user's task.
- Relevant Boundaries: Draw distinctions among objects and actions along boundaries relevant to the task.
- Identifiability: Present objects and actions using distinguishable, truthful appearances.
- Foresight: Indicate clearly the consequences of decisions that the user is expected to make.

13.2.11 Herstellen eines Trusted Path

Wir wollen hier nur eine der zehn Regeln von Yee herausgreifen und etwas genauer diskutieren: Die Herstellung eines Trusted Path zu Programmen und Objekten sowie deren wahrheitsgemäße Darstellung. „Trusted Path“ bedeutet, dass der



Abb. 13.12 Wirklich ein Systemdialog?

Benutzer sicher sein kann, mit dem von ihm gewollten Programm zu interagieren und nicht mit einem gefälschten. Das Fehlen eines Trusted Path kann bedeuten, dass Applikationen beliebige Fenster manipulieren und damit selbstverständlich auch Systemdialoge fälschen oder durch eigene Dialoge überlagern können (Abb. 13.12).

Was aber ist letztlich ein Click eines Benutzers wert, wenn der Inhalt, der dadurch bestätigt wird, beliebig manipulierbar ist? Abbildung 13.13 zeigt ein Beispiel für die Überlagerungstechnik bei Dialogen.

Bei Dialogen mit Benutzern tritt zudem der „Gehorsamkeitseffekt“ auf – die Benutzer tendieren dazu, den Befehlen des GUIs zu gehorchen. Dazu ein Zitat von Ka-Ping Yee:

“Phishing attacks combine the Obedience to Authority problem with the Simon Says problem. In order for a user to successfully evade a phishing attack, using current browsers or even with many of the proposed anti-phishing tools, a very weak stimulus (the lack of something) has to override a strong impulse (the compulsion to obey an authority). The odds are stacked against the user.” [Yee05]

Derzeit wird im Bereich der User-Interfaces mit verschiedenen Formen von Trusted Path experimentiert: Spezielle Farben oder auch spezielle Orte am Bildschirm sollen einen vertrauenswürdigen Dialog mit dem Betriebssystem signalisieren.

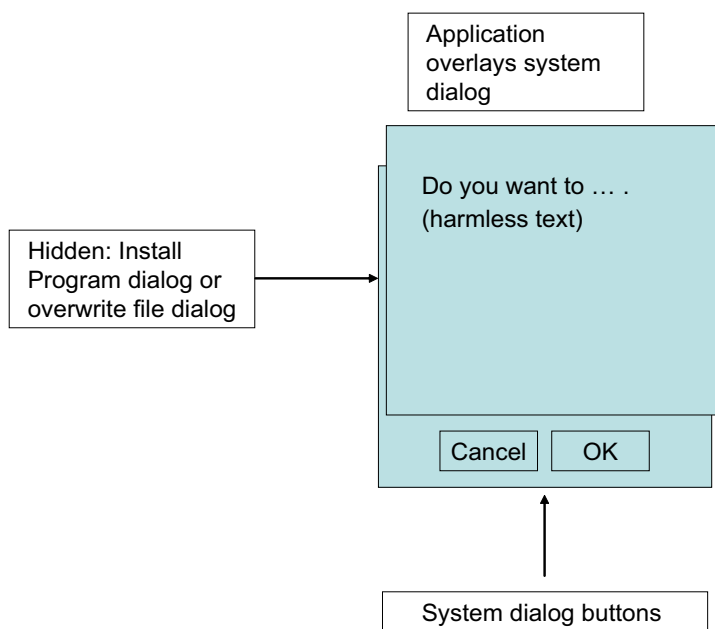


Abb. 13.13 Beispiel einer Überlagerung von Dialogen

13.2.12 *Web2.0 und die Frage der Nutzerintention*

Gerade haben wir die Frage aufgeworfen, was ein Mouseclick eines Benutzers angesichts der Manipulationsmöglichkeiten im GUI letztlich noch bedeuten kann. Die AJAX-Technologien machen sogar diese Überlegungen überflüssig: Der herunter geladene Code erzeugt seine eigenen Bestätigungen im Hintergrund, ohne dass der Benutzer etwas davon erfährt. Offenbar ist dies ein Sicherheitsproblem. Seine schnelle Beseitigung würde aber viele der hoch gelobten Vereinfachungen durch AJAX-Techniken wieder rückgängig machen.

Aus dem statisch installierten Renderer des DARPA-Browser-Experiments wird heute durch AJAX-Technologien etwas ganz anderes: Der Renderer ist nur noch zum Teil eine Browser-Komponente, der andere Teil des Renderers wird dynamisch in Form von JavaScript herunter geladen bzw. nachgeladen. Der Renderer ist somit zum Remote-Control-Client für den Server geworden. Was bedeutet das für die Intentionen des Benutzers und ihre sinngemäße Umsetzung durch den Browser? Offensichtlich sind die Intentionen des Benutzers hier gar nicht so leicht greifbar. Wie viel Integration und wie viel Aggregation will der Nutzer? Ab wo sieht er seine Privatsphäre verletzt?

Unsere obigen Restriktionen für einen sicheren Browser machen in diesem Zusammenhang zunächst keinen Sinn mehr: Wenn der Renderer keine neuen URLs definieren kann, kann er nicht dynamisch Daten an den Server schicken. Der Browser kann seinerseits eine Page nicht mehr so einfach parsen und Capabilities erstellen, wenn die Designation dazu im herunter geladenen Code steckt. Schließlich würde ein Renderer ohne Speichermöglichkeit jede Form von Datenaggregation verhindern, wenn für jede URL ein neuer Renderer gestartet werden muss.

Das Problem aus Capability-Sicht besteht also in einer Menge impliziter oder kodierter Autorität, die ohne Zutun des Nutzers ausgeführt wird. Aber um welche Autorität handelt es sich dabei genau? Die folgende Liste enthält abschließend einige Fragen und Ideen, die helfen können, sichere Interfaces bei maximal möglicher Interaktion zu realisieren:

- Was ist eigentlich das Bedrohungsmodell für Web2.0? Welche Autorität ist für die Arbeit in diesem Modell nötig?
- Wie können unterschiedliche Identitäten der beteiligten Server sichtbar gemacht werden, ohne dabei die Effekte der Aggregation zu verspielen?
- Welche Rolle spielt Transaktionssignierung? Wie ist sie vom Sessionmanagement zu trennen?
- Wie kann ein Trusted Path realisiert werden?
- Welche Form von Kontrolle über das User-Interface dürfen AJAX-Module nicht erhalten?
- Wie verhindert man ungewollte Brücken zwischen Intranet und Internet?
- Wie verhindern wir die zeichenweise, ungewollte Weitergabe von Eingabedaten?
- Welche AJAX-effekte sollten deklarativ statt programmatisch sein?

- Was muss mit den Credentials des Nutzers geschehen?
- Wie müssen Übergänge zwischen Vertrauenszonen visualisiert werden? Welche Rolle könnten Labelling-Systeme dabei spielen?
- Wie kann Identität und Vertrauen dem Nutzer sichtbar gemacht werden?

Literatur

- [BDG] D. Balfanz, G. Durfee, R. E. Grinter, D. K. Smetters, P. Stewart, Network in a Box: How to set up a secure wireless network in under a minute, Proc. 13th USENIX Security Symposium (2004), p. 204–221, available online at <http://www.parc.com/research/publications/files/5180.pdf>
- [Botz06] K. Botzum, WebSphere Application Server V6.1: What's new in security?, http://www-128.ibm.com/developerworks/websphere/library/techarticles/0606_botzum/0606_botzum.html
- [Cal] M. A. Caloyannides, Privacy Protection and Computer Forensics (Second Edition), Artech House, 2004
- [Garf05] S. Garfinkel, Sanitization and Usability, Kapitel 15 in L. F. Cranor, S. Garfinkel (Eds.): Security and Usability – Designing Secure Systems that People Can Use, O'Reilly, 2005
- [GSNM1] S. Garfinkel, J. Schiller, E. Nordlander, D. Margrave, R. Miller, How To Make Secure Email Easier To Use, Proc. CHI 2005: Technology, Safety, Community, Portland, Oregon, April 2–7, 2005, online at <http://www.simson.net/cv/pubs.php>
- [GSNM2] S. Garfinkel, J. Schiller, E. Nordlander, D. Margrave, R. Miller, Views, Reactions and Impact of Digitally-Signed Mail in e-Commerce, in: A. S. Patrick, M. Yung (Eds.), Proc. Ninth International Conference on Financial Cryptography and Data Security, Springer-Verlag 2005, online at <http://www.simson.net/cv/pubs.php>
- [GM] S. Garfinkel, R. Miller, Johnny 2: A User Test of Key Continuity Management with S/MIME and Outlook Express, presented at the Symposium on Usable Privacy and Security (SOUPS 2005), July 6–8, 2005, Pittsburgh, PA, online at <http://www.simson.net/cv/pubs.php>
- [Guttm] P. Guttmann, Security Usability Fundamentals, Draft 2008, <http://www.cs.auckland.ac.nz/~pgut001/pubs/usability.pdf>
- [KS] W. Kriha, R. Schmitz, Usability und Security, in KES 03/2007, Zeitschrift des Bundesamtes für Sicherheit in der Informationstechnik, <http://www.bsi.bund.de/literat/forumkes/kes0307.pdf>
- [Lako] G. Lakoff, Women, Fire and Dangerous Things – what categories reveal about the mind. University of Chicago Press 1987
- [Moz] Mozilla Trustbar Screenshots, http://trustbar.mozdev.org/screenshots.html#not_secured
- [RFC4279] P. Eronen, H. Tschofenig, Pre-Shared Key Ciphersuites for Transport Layer Security (TLS), <http://www.ietf.org/rfc/rfc4279.txt>
- [Step] N. Stephenson, Die Diktatur des schönen Scheins – Wie grafische Oberflächen Computernutzer entmündigen, Goldmann 2002
- [Sti1] M. Stiegler, An Introduction to Petname Systems, <http://www.skyhunter.com/marcs/petnames/IntroPetNames.html>
- [Sti2] M. Stiegler, The Skynet Virus: Why it is unstoppable; How to stop it (Video), <http://www.skyhunter.com/marcs/skynet.wmv>
- [VISA] Verified by Visa Konzept, siehe http://usa.visa.com/personal/security/visa_security_program/vbv/how_it_works.html

- [WT] A. Whitten, J. D. Tygar, Why Jonny Can't Encrypt, Kapitel 34 in L. F. Cranor, S. Garfinkel (Eds.): Security and Usability – Designing Secure Systems that People Can Use, O'Reilly, 2005 (ursprünglich in Proc. 8th USENIX Security Symposium, Washington, D.C., 1999)
- [Yee02a] K. P. Yee, User Interaction Design for Secure Systems, Berkeley University Tech Report CSD-02-1184, 2002. Online at <http://www.sims.berkeley.edu/~ping/sid/uidss-may-28.pdf>
- [Yee05] K. P. Yee, Guidelines and Strategies for Secure Interaction Design, Kapitel 13 in L. F. Cranor, S. Garfinkel (Eds.): Security and Usability – Designing Secure Systems that People Can Use, O'Reilly, 2005

Kapitel 14

Bestimmung der Sicherheit durch formale Ansätze

Bisher wurden von uns Fragen der Software-Sicherheit in erster Linie unter dem Gesichtspunkt der Entwicklung bzw. der Systemarchitektur behandelt. Mit dem Ziel, die Wahrnehmung der Entwickler für grundsätzliche Sicherheitsproblematiken in Software zu schärfen, haben wir dabei bewusst auf eine formale Darstellung weitestgehend verzichtet. Wir haben verschiedene Mechanismen der Isolation oder Kontrolle von Zugriffen kennengelernt und diskutiert und sind dabei auch auf Begriffe wie „identitätsbasierte Zugriffskontrolle“, „Sandboxes“, „Filter“, „Sicherheitsbausteine“ und nicht zuletzt „Erreichbarkeit“ gestoßen: Um einen Effekt zu erzielen (egal ob positiv oder negativ) muss ein Subjekt ein Objekt erreichen können. Diese Konzepte und Mechanismen operieren auf ganz unterschiedlichen Ebenen der Sicherheit von Systemen und werden dementsprechend unterschiedlich modelliert.

In diesem Kapitel soll nun anhand neuerer Forschungsarbeiten im Bereich der Modellierung von Autorität und Rechten eine Brücke zu den theoretischen Grundlagen der Access Control geschlagen werden. Dabei werden wir auch schwächere Kriterien als die Erreichbarkeit kennenlernen in Form von indirekter Erreichbarkeit oder Beeinflussung. Wir werden die theoretischen Modelle hinter Access Control Matrices sowie Take/Grant-Modellen kurz darstellen und untersuchen, wie sie sich z. B. auf Objekt Capabilities anwenden lassen. Die Erreichbarkeit von Objekten kann nun entweder über Regeln kontrolliert werden – z. B. in Form einer Access Control Matrix (ACM), die Subjekte und Objekte über deren Rechte verbindet. Durch Eintragen neuer Rechte ändert sich die Erreichbarkeit von Objekten – ohne dass jedoch dadurch auch sichergestellt ist, dass die Konfiguration der Software selbst eine solche Erreichbarkeit ermöglicht: Matrix und Software unterscheiden sich nach „dürfen“ und „können“. Die Erreichbarkeit kann jedoch auch durch das Fließen von Zugriffsrechten in Form von Capabilities erreicht oder eben auch verhindert werden, im Fall nämlich, dass keine Capabilities fließen. Hier muss durch das Fließen von Rechten ein Pfad zwischen den Subjekten entstehen, dann ist die Erreichbarkeit gegeben.

Es geht also im Fall der ACM darum zu prüfen, welche Manipulationen der Rechte möglich sind und ob die Folgen des Einfügens eines bestimmten Rechts berechenbar sind. Im zweiten Fall geht es darum, in einer Anfangskonfiguration verbundener Subjekte und deren Verhaltensregeln einen Pfad zu einem bestimmten Subjekt zu finden.

Im Weiteren werden wir untersuchen, wie Sicherheitsbausteine modelliert werden können und verschiedene Ansätze von constraint-basierten Modell-Checkern vorstellen, so z. B. Ansätze zur automatischen Prüfung von Sicherheitsprotokollen bzw. der Modellierung von Sicherheitsforderungen in Systemen oder Applikationen. Auch ein mögliches generatives Herangehen an Sicherheitsprobleme in Infrastrukturen werden wir besprechen, um uns am Ende formalen Aspekten der Input Validation zuzuwenden. Wir beginnen mit der Frage nach der Entscheidbarkeit von Sicherheit. Wie sich zeigt, ist dieses Frage algorithmisch unentscheidbar. Trotzdem werden wir uns sehen, dass man aus praktischer Sicht in dieser Frage einiges bewirken kann.

14.1 Zur Frage der Entscheidbarkeit von Sicherheit

Wenn wir die Frage nach der Sicherheit eines lokalen oder verteilten Systems stellen, dann tun wir das aus verschiedenen Gründen: Zum einen wollen wir für ein gegebenes System klären, ob es den Sicherheitsanforderungen genügt, d. h. ob die Implementation der jeweiligen Security-Policy entspricht. Wir können auch danach fragen, ob eine bestimmte Policy selbst konsistent und sicher ist, d. h. wir möchten das Modell selbst prüfen (z. B. ob es in sich widerspruchsfrei ist).

Zum anderen können wir aber alternativ auch versuchen, ein neues sicheres System zu konstruieren und dabei Methoden verwenden, die ein solches sicheres System erzeugen. Hier lautet die Frage dann: Wenn wir eine bestimmte Policy durchsetzen wollen, welche Konfiguration (Klassen, Objekte, Konfigurations-Files etc.) einer Software ist dazu nötig? Anders gefragt: Wie muss eine bestimmte Software erweitert werden, um eine neue Sicherheitsanforderung zu realisieren?

In beiden Fällen – Analyse oder Konstruktion eines Systems – treten einige Probleme auf: Die Policies – also die Vorgaben zur Sicherheit – sind meist in textueller Form formuliert. Sie werden dann von den Entwicklern in Konfigurationsdaten und Code umgesetzt. Am Beispiel der Access Control Matrix von Harrison, Ulmann und Ruzzo [HRU] lassen sich die Probleme dabei gut demonstrieren. Eine ausgezeichnete Diskussion der Access Control Matrix im Zusammenhang mit Berechenbarkeit findet sich bei [Spie].

14.1.1 Die Access Control Matrix

Ein Sicherheitssystem nach Harrison, Ulmann und Ruzzo besteht im Wesentlichen aus einer Anfangskonfiguration, bestehend aus Subjekten, Objekten und einer Ma-

trix, die Rechte zwischen beiden enthält. Hinzu kommt ein Satz von Kommandos, mit denen die Matrix gelesen, aber auch manipuliert werden kann (Abb. 14.1).

ACMs beschreiben die Rechte, die beim Zugriff eines Subjektes (eine aktive Instanz im System, z. B. ein Prozess) auf ein Objekt zur Verfügung stehen. Es ist dabei ohne Bedeutung, ob die Software-Architektur diese Zugriffe überhaupt ermöglichen würde. Die Rechte drücken daher eine Policy aus, die unabhängig von der Software-Konfiguration und damit allgemeingültig ist. ACMs können jedoch nicht nur Beziehungen zwischen Subjekten und Objekten beschreiben, sie können durch einen kleinen Trick auch Eigenschaften der Subjekte selbst beinhalten, indem diese selbst wieder als Objekte in der Matrix auftauchen (Abb. 14.2).

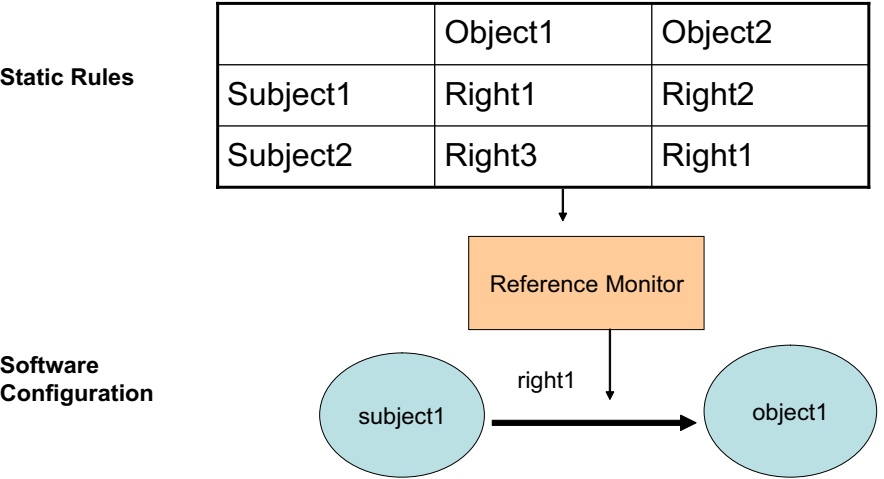


Abb. 14.1 Access Control Matrix und ihre Anwendung für die Zugriffskontrolle

Objects including subjects

	O1 (s1)	O2 (s2)	O3	O4	
subjects	S1	T1	T2	R1	R2
	S2	T2	T3	R3	R1
	S3	T1	T3	R2	R5

Abb. 14.2 Subjekteigenschaften modelliert durch Subjekte als Objekt

Beispielsweise kann ein Prozess P1 sowohl Subjekt als auch Objekt in einem System sein (P1 wird von User S1 gestartet und greift auf Datei O2 zu.) Bis zu diesem Punkt sind die Rechte in der ACM statisch und die Korrektheit der Konfiguration (die Übereinstimmung mit der Policy) ist durch einen Algorithmus prüfbar.

Interessant wird es, sobald dynamische Rechtemanipulationen auftreten, z. B. wenn neue Subjekte dynamisch angelegt werden. Harrison, Ulmann und Ruzzo haben diese Dynamik in Form von sogenannten *primitive commands* der Form $pc(S1...Sx, O1...Ox)$ abgebildet. Der Body eines Kommandos greift auf die Felder der Matrix zu und kalkuliert aus den dort vorgefundenen Rechten die neuen Rechte. Primitive commands können sein (vgl. [Bis] für eine vollständige Darstellung):

- `createSubject(s)` (erzeugt ein neues Subjekt s, d. h. in die ACM wird eine neue Zeile eingefügt)
- `createObject(o)` (erzeugt ein neues Objekt o, d. h. in die ACM wird eine neue Spalte eingefügt)
- `enterRight(r, a[s,o])` (in die Zelle in Zeile s und Spalte o wird ein Recht r eingefügt)
- `deleteRight(r, a[s,o])` (aus der Zelle in Zeile s und Spalte o wird ein Recht r entfernt)
- `destroySubject(s)` (entferne Subjekt s aus der Matrix, d. h. eine ganze Zeile wird entfernt)
- `destroyObject(o)` (entferne Objekt o aus der Matrix, d. h. eine ganze Spalte wird entfernt)

Die dynamische Rechtemanipulation bewirkt also eine Änderung von Zellen der Matrix und es entsteht eine neue Sicherheitskonfiguration. Somit taucht die Frage auf, ob auch die neue Konfiguration korrekt ist. Einige Änderungen der Rechte-Einträge sind ja in einem sich ändernden System unvermeidbar bzw. nötig. Harrison, Ulmann und Ruzzo reduzieren die Frage der Korrektheit auf die Frage nach der Entscheidbarkeit einer Änderung: Gibt es einen Algorithmus, der aus einer Anfangskonfiguration und dem nachfolgenden primitive commands sicher feststellen kann, ob ein bestimmtes Feld der Matrix ein (neues) Recht erhalten wird? In diesem Fall spricht man davon, dass das Recht entweicht („leaked“). Harrison, Ulmann und Ruzzo haben den Beweis geführt, dass bei komplexeren Kommandos und sich ändernden Subjekten die Frage, ob ein Recht „leaked“, nicht berechenbar ist. Neuere Studien haben zwar einige Fehler in den Beweisen gefunden, jedoch die Grundaussagen bestätigen können [TL].

Welche Systeme leiden nun unter dieser Unberechenbarkeit? Beispielsweise kombinieren heutige Benutzerberechtigungssysteme statische Rechtebeschreibungen (z. B. in Datenbanken oder LDAP) mit Applikationen oder Rule-Engines, die dynamisch neue Rechte ermitteln können. Ihre Korrektheit ist aufgrund der Komplexität nur sehr schwer zu überprüfen. Da sich keine formalen Beweise durchführen lassen, bleibt nur empirisches Testen.

Eine wichtige Eigenschaft von ACM-basierter Zugriffskontrolle ist, dass sie die Prüfung eines Zugriffs ohne Rückgriff auf den Kontext des Aufrufs durchführt. Dadurch erhält Malware die gleichen Rechte wie normale Programme. Lässt sich nun beispielsweise die Java2-Security dennoch teilweise auf eine ACM abbilden? Wir können dies in zwei Schritten erreichen. Im ersten Schritt nehmen wir nur einen Ausschnitt der ACM als aktuelle ACM – d. h. wir bilden quasi einen groben Kontext aus beteiligten Subjekten und Objekten. Dadurch entsteht eine Sandbox für Code, die die zur Verfügung stehenden Rechte grundsätzlich auf bestimmte Subjekte und Objekte eingrenzt. Im zweiten Schritt untersuchen wir bei der Kontrolle eines Zugriffs, ob alle an der Aufrufkette beteiligten Subjekte das entsprechende gleiche Recht besitzen. Nur wenn alle Beteiligten das verlangte Recht auch besitzen, darf der Zugriff gelingen – oder das letzte Subjekt, das die Prüfung durchführt, muss das „Privileged“-Recht besitzen. Java verwendet zum Tracing des Aufrufs die Analyse des Call-Stacks [Wall].

Zwei weitere Eigenschaften von Rechten in ACM-Systemen müssen noch diskutiert werden: Das „Owner“-Recht und das Löschen von Rechten. Das „Owner“-Recht wird typischerweise beim Ersteller eines neuen Subjekts eingetragen: Der Erzeuger ist der Eigentümer des neuen Objektes und erhält dadurch die Möglichkeit, diejenigen Rechte zu erhalten, die den Lebenszyklus eines Objektes bestimmen: z. B. „delete“ und „Ownership Change“. Dieses Verhalten wird oft mit der Eigenschaft „simple security“ begründet: Man kann nur solche Rechte vergeben, die man selber besitzt. Das muss aber nicht notwendigerweise der Fall sein. So macht es in bestimmten Umfeldern durchaus Sinn, die Rechte der Eigentümer eines Objekts deutlich einzuschränken bis hin zum Extremfall, dass z. B. eine Rolle ein Recht vergeben kann, das ihr selbst nicht zusteht. Beispielsweise kann ein Subjekt in der Rolle „Organisationsadministrator“ Subjekten innerhalb dieser Organisation das Recht zum Bearbeiten von Kundeneinträgen erteilen. Selbst darf die Rolle des Organisationsadministrators die Einträge jedoch nicht bearbeiten. Auch ist es dieser Rolle nicht gestattet, sich selbst an andere zu vergeben, also andere Subjekte in die Rolle des Organisationsadministrators zu bringen.

Was passiert beim *Löschen* von Rechten bezüglich der Sicherheit einer Konfiguration? An mancher Stelle findet man in der Literatur das Argument, dass das Löschen von Rechten grundsätzlich weniger problematisch sei als das Einfügen von neuen Rechten in die Matrix. Begründet wird dies dadurch, dass durch das Fehlen eines Rechts keine Sicherheitslücke entstehen kann. Dieses Argument setzt jedoch eine enge Interpretation von Sicherheit als bloßer Schutz gegen Missbrauch voraus. Immer mehr setzt sich jedoch die Auffassung durch, dass Sicherheit auch die grundsätzliche Verfügbarkeit und Gebrauchsfähigkeit von Systemen beinhaltet. In Bezug auf diese Kriterien kann das Löschen von Rechten fatal für die Sicherheit des Systems sein. Gerade große Firmen mit umfangreichen Benutzerberechtigungssystemen, die hunderte von firmenspezifischen Rollen mit tausenden von Rechten für Applikationen oder Daten beinhalten, sind anfänglich für Fehler durch Löschen von Rechten. Im schlimmsten Fall können so wichtige Businessprozesse von den Mitarbeitern nicht mehr durchgeführt werden. Leider lässt sich die Gefahr der Löschung genauso wenig wie das Eintragen eines neuen Rechts formal ent-

scheiden. Sogenannte „Liveness“-Probleme scheinen sogar noch schlimmer zu sein als bloße Rechtfragen: Attacken auf die Liveness einer Server-Applikation sind nur sehr schwer abzuwehren, wie Distributed Denial-of-Service (DDoS) Attacken täglich demonstrieren.

Das Löschen von Rechten wirft noch andere interessante Fragen auf: Wenn durch Kommandos ein Recht in eine ACM neu eingetragen wird, anschließend gelöscht und erneut eingetragen wird – ist dies das Gleiche, wie wenn das Recht sich nicht geändert hätte [Spie] ? In anderen Worten: Spielt auch die *Historie* von Rechten eine Rolle? In der ACM ist die Historie von Rechten in einer Zelle nur in Form zeitlich aufeinander folgender Matrizen festgehalten. Vielleicht wäre es aber auch wichtig, zu wissen, ob ein Subjekt einmal ein Zugriffsrecht auf ein Objekt besaß? „Chinese-Wall“-Zugriffsmodelle stellen genau diese Frage und wollen dadurch verhindern, dass z. B. externe Berater Informationen aus einer Firma in einem Auftrag für eine andere Firma verwenden.

Wichtiger als die Zugriffsrechte selbst (sie können ja gelöscht werden) ist die Frage, ob ein Subjekt durch ein einmal vorhandenes Zugriffsrecht beeinflusst wurde oder Einfluss genommen hat. Dieser Einfluss bleibt ja nach dem Entfernen des Rechts weiter erhalten und kann zukünftige Entscheidungen beeinflussen. Wir werden in Abschn. 14.1.2 eine Trennung von Zugriffsrechten in solche, die Rechte transportieren und solche, die nur Einfluss ermöglichen, vornehmen. Hier bleibt als Idee die Möglichkeit, einmal bestehende Rechte bzw. Einfluss in der Matrix als Eigenschaft des Subjektes bezüglich eines Objektes festzuhalten, auch wenn das ursprüngliche Recht gelöscht wurde.

Welche Rollen können ACMs dann innerhalb einer Sicherheitsarchitektur spielen? Es scheint, dass sie statische Rechtebeziehungen von Subjekten an Ressourcen recht effizient modellieren können. Sie können daher bei der Erzeugung von Capabilities helfen. Delegation von Rechten hingegen ist nicht ihre Stärke. Eine bedeutende Einschränkung bei der Zugriffskontrolle auf Basis von ACMs besteht darin, dass eine Ablehnung des Zugriffs zwar ein sicheres Ergebnis darstellt, die Genehmigung des Zugriffs hingegen nicht unbedingt: Das Recht auf eine Ressource X mag zwar für die Identität Y gegeben sein, es ist jedoch noch längst nicht klar, ob der Zugriff auch im Einklang mit den Intentionen von Y steht – d. h. ob der Softwareagent, der Y vertritt, zu Recht den Zugriff auf die Ressource Y verlangt. Diese Schwäche bei der Delegation von Rechten über ACMs (alle Rechte sind verfügbar) erklärt, warum Spiessens die Delegation von Rechten über Capabilities als „just in time least authority“ bezeichnet [Spie, S. 144ff.].

14.1.2 Capabilities modelliert durch Take/Grant-Systeme

Im Unterschied zu ACMs wird durch Capabilities der Fluss von Autorität und Rechten bzw. Information innerhalb eines Systems betrachtet, z. B. durch den Austausch von Capabilities. Dies stellt einen fundamentalen Unterschied zu ACM-Systemen dar, die ja, wie oben gezeigt, kontextfrei operieren, d. h. von der

realen Softwarearchitektur unabhängig sind. Durch Capabilities – speziell in Form der Objekt-Capabilities wie sie vor allem durch Mitglieder der erights.org Community [MiSt] bekannt gemacht wurden– wandert die Security-Policy, die bei Verwendung von ACMs in einem externen Modul realisiert ist, direkt in die Software-Architektur hinein (Abb. 14.3). Dadurch tritt eine automatische Restriktion in Kraft: „Etwas dürfen“ und „etwas können“ innerhalb eines Software-Systems sind nun enger aneinander gekoppelt als bei ACMs.

Was sind nun eigentlich die sicherheitsrelevanten Eigenschaften von Capabilities? Wesentliches Element der Sicherheitsstrategie in Systemen, die Capabilities verwenden, ist der Zugriff (Access) auf Objekte, der durch den Besitz von Capabilities ermöglicht wird. Wer eine Capability besitzt, kann sie zum Aufruf des dahinterstehenden Objekts verwenden. Wie entstehen Capabilities? Durch

- die Erzeugung eines Objektes (der Erzeuger erhält eine Capability auf das erzeugte Objekt),
- durch Übergabe von Capabilities an das erzeugte Objekt (Endowment),
- durch Übergabe von Capabilities mittels Methodenaufruf.

Auch hier zeigt sich die besondere Rolle der Erzeugung bei der Rechtevergabe, ähnlich dem „Owned“-Recht bei ACMs. Wie in einem objektbasierten System üblich, wird zur Startzeit der Applikation durch Erzeugung von Objekten und deren Verknüpfung durch Referenzen das Objektnetz erzeugt, das dann die Ausführung des Programms steuert. Eine Security-Policy sorgt bei der Erzeugung dieses Objektnetzes dafür, dass die richtigen Capabilities von den entsprechenden Objekten gehalten werden.

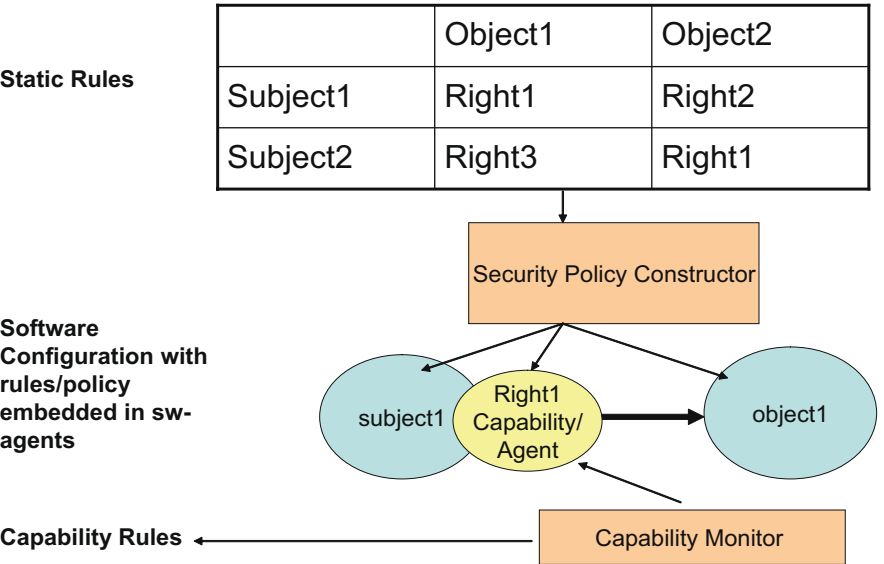


Abb. 14.3 Fluss von Rechten durch Capabilities

Dabei tritt sofort die Frage nach der Veränderung der Konfiguration z. B. durch die Weitergabe von Capabilities über Objekte auf. Die Erzeugung neuer Objekte kann zwar noch so geregelt werden, dass das erzeugte Objekt maximal alle Capabilities des erzeugenden Objekts erhält. Aber das Verhalten der Weitergabe von Capabilities – und damit von Rechten für Zugriffe – ist nirgends geregelt. Im Klartext bedeutet dies, dass für die Frage der Weitergabe von Rechten von einem Worst-Case-Szenario auszugehen ist: Jedes Subjekt wird – wenn es Zugriff auf ein Objekt durch eine Capability erhält – diesem Objekt alle seine Rechte (Capabilities) aushändigen (*grant*) und gleichzeitig alle Rechte von diesem Objekt übernehmen (*take*). Somit erhält man eine maximale Ausbreitung der Rechte auf alle Subjekte im System. Damit wird es aber unmöglich, isolierte Bereiche zu erzeugen, also Bereiche, die nicht von beliebigen anderen Stellen erreichbar bzw. manipulierbar sind. Alle Subjekte eines solchen Systems hängen rechtemäßig eng zusammen.

Somit lassen sich bekannte Security-Policies wie die sog. *-properties (also kein Schreiben von Informationen an Subjekte mit niedrigerer Sicherheitsklassifikation sowie kein Lesen von Subjekten mit höherer Sicherheitsklassifikation) nicht realisieren. Eine Eingrenzung von Autorität – auch *Confinement* genannt – kann nicht stattfinden. Eine abgestufte Security-Policy ist nicht möglich. Zu *Confinement* siehe auch Butler Lampsons bekanntes Paper „A Note on the Confinement Problem“ [Lamp]. Jonathan Shapiro konnte jedoch zeigen, dass sich durch die Einführung von Sicherheitsbausteinen sowie deren Verhaltensqualifizierung in Bezug auf die Weitergabe von Rechten das Confinement-Problem auch durch Capabilities lösen lässt [Shap].

Formal modelliert wurden capability-basierte Systeme schon früh anhand von sogenannten *Take/Grant-Modellen*, nicht zuletzt, da der Versuch einer formalen Verifizierung von Sicherheitsmodellen in Form von ACMs scheiterte. In einem Take/Grant-System gelten einfache Regeln: Das Basisrecht des Zugriffs auf ein Subjekt/Objekt über eine Capability wird in vier unterschiedliche Rechte aufgespalten. Sie lauten: *Take*, *Grant*, *Read*, *Write* und beschreiben jeweils eine gerichtete Beziehung zwischen zwei Subjekten, die beide stillschweigend mit dieser Beziehung einverstanden sind.

Abbildung 14.4 zeigt einige Basispatterns eines Take/Grant-Modells. Im oberen Fall erhält Bob durch das *Take-Right* an Carol automatisch alle Rechte von Carol, z. B. auch das an Alice. Grundvoraussetzung ist jeweils ein genereller Zugriff (Access). Im unteren Fall werden lediglich Daten von Alice zu Bob transportiert, aber keine Zugriffsrechte. Dadurch ändern sich die Rechtsverhältnisse im System nicht, jedoch möglicherweise die Kausalität zukünftiger Aktionen (Bob beschließt eine Aktion aufgrund der Daten von Alice).

Für die Propagation von Rechten sind also Take/Grant-Rechte nötig, für die Beeinflussung von Subjekten über Informationsaustausch hingegen Read/Write-Rechte. Der Besitz eines Zugriffsrechts über eine Capability ist dabei Voraussetzung. Die Bezeichnungen „de jure“ und „de facto“ sind vielleicht nicht ganz glücklich, da sie in Bezug auf die indirekten Zugriffsmöglichkeiten missverstanden werden können. De-Jure-Rechte sind auch alle Rechte, die indirekt bezogen

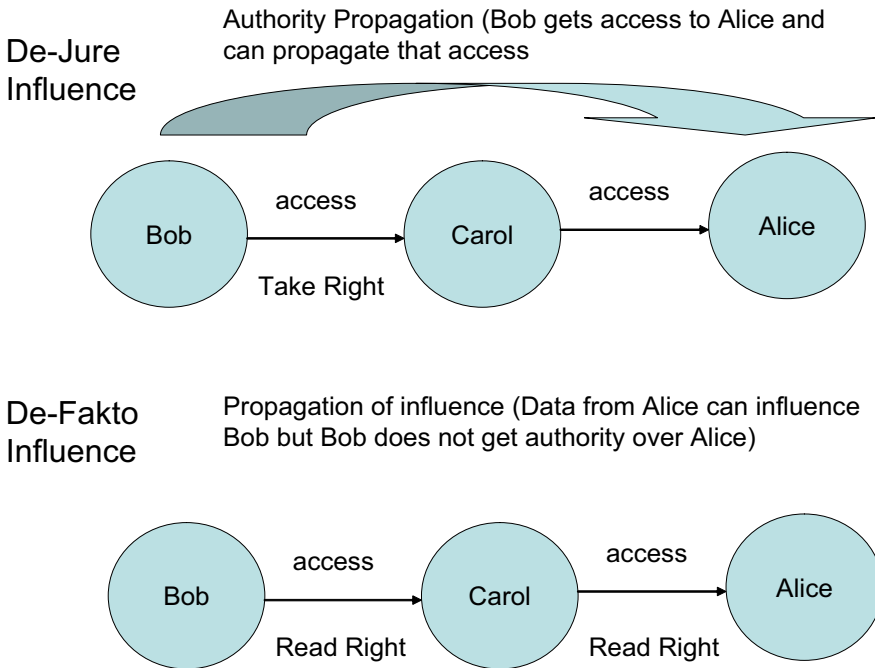


Abb. 14.4 Unterschied von Rechte- und Informationspropagation

werden können, solange es sich um echte Rechte, sprich Autorität bzw. Capabilities handelt. Dies schließt also nicht nur die Capabilities ein, die ein Subjekt bereits besitzt, sondern auch alle Capabilities, die das Subjekt über Kollaboration über diese Capabilities bezieht. Das Gleiche gilt entsprechend für De-Fakto-Rechte, die nicht Rechte/Capabilities propagieren, sondern nur Information.

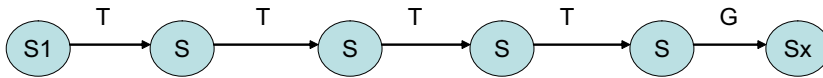
Ein Take/Grant-System lässt sich als eine Folge von Graphen darstellen, die durch Austausch von Capabilities mittels Take/Grant-Rechten in neue Graphen überführt werden. An die jeweiligen Graphen lassen sich Fragen stellen wie etwa:

- Ist es in diesem Graphen möglich, dass ein Subjekt X Zugriff auf ein Subjekt Y erhalten kann?
- Ist es in diesem Graphen möglich, dass eine Beeinflussung von Subjekt A durch Subjekt B stattfinden kann?

Spiessens nennt eine Reihe von Beeinflussungen wie Verbreiten, Spionieren, Finden etc. Take/Grant-Systeme kennen eine Reihe von Verbindungsmustern, die bestimmte Eigenschaften haben. Abbildung 14.5 zeigt eine sog. *Anfangs-* bzw. *Endspanne*, mit deren Hilfe Rechte transportiert werden können. Insbesondere gilt, dass jeder Graph, der aus Subjekten besteht, die durch Take oder Grant miteinander verbunden sind, eine Zone bildet, in der sämtliche Rechte aller Subjekte geteilt werden und in der somit keine Trennung innerhalb der Zone stattfindet.

Die entsprechenden Graphen lassen sich mit endlichem Aufwand traversieren und die Fragen somit entscheiden. Dies ist ein deutlicher Vorteil gegenüber der

Can S1 give Sx some right?



Can S1 get some right from Sx?

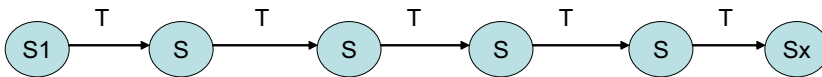


Abb. 14.5 Rechtepropagation in einfachen Take/Grant-Modellen

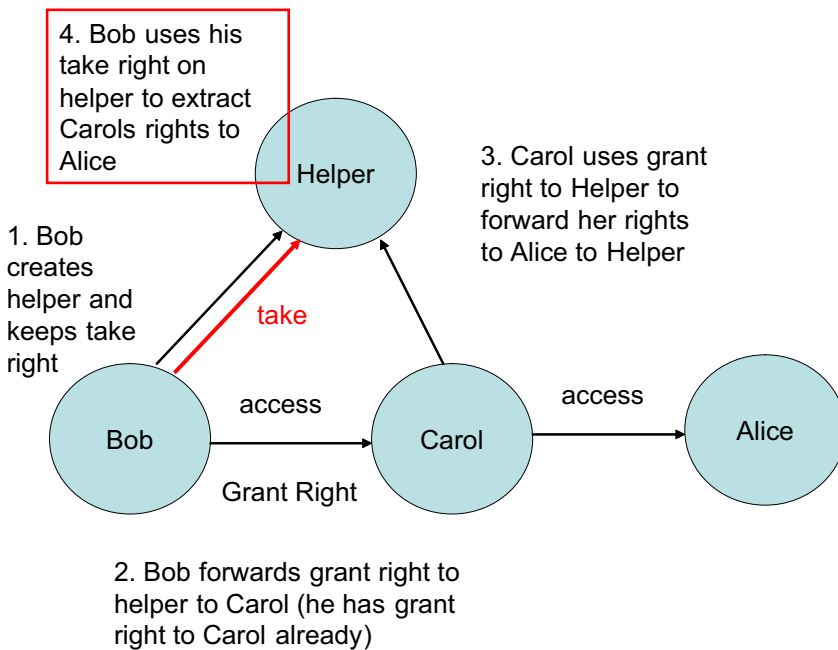


Abb. 14.6 Auswirkungen totaler Kooperation in einfachen Take/Grant-Modellen

Unentscheidbarkeit von ACMs. Leider ist der Vorteil nur ein relativer. Schnell wurde nämlich deutlich, dass diese ursprünglichen Formen von Modellierung mittels simpler *Take/Grant*- bzw. *Read/Write*- Rechte keine wirkliche Reduktion von Autorität bewirken können, wie die problemlose Rechtepropagation in Abb. 14.5 aufzeigt. Ein Beispiel bildet die Umkehrung einer Grant-Beziehung in eine Take-Beziehung mittels neu erzeugter Hilfsobjekte: Einfache Take/Grant-Systeme gehen von einem völlig kooperativen Verhalten der Beteiligten aus. Die Auswirkungen dieser totalen Kooperation zeigt Abb. 14.6: Ein Subjekt (Bob)

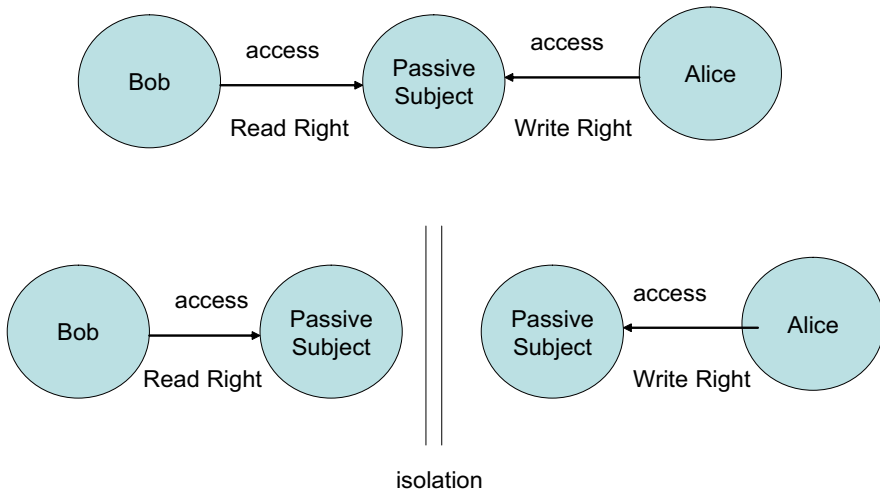


Abb. 14.7 Isolation von Autoritätsfluss durch zwei passive Subjekte

besitzt lediglich *Grant*-Rechte an einem anderen Subjekt (Carol), möchte aber von ihm Rechte erhalten. Dazu erzeugt Bob ein Hilfsobjekt, an dem er das *Take*-Recht und das *Grant*-Recht besitzt. Das *Grant*-Recht an dem Hilfsobjekt überträgt Bob an Carol, diese überträgt ihr Zugriffsrecht an Alice an das Hilfsobjekt und Bob kann schließlich per *Take*-Recht das Zugriffsrecht an Alice vom Hilfsobjekt bekommen.

Selbst im Falle von sog. *passiven Subjekten* (d.h. Objekten im Sinne von ACMs) wird davon ausgegangen, dass ein passives Subjekt jede Information oder Capability annehmen und in Falle eines *Take*- oder *Read*-Zugriffs durch andere Subjekte auch wieder herausgeben wird. Dies ermöglicht die Weitergabe von Rechten über passive Subjekte hinweg, wie Abb. 14.7 zeigt.

Zwei aufeinander folgende passive Subjekte hingegen trennen die Zugriffsgraphen voneinander. Es gibt eine ganze Reihe von Mustern innerhalb von *Take/Grant*-Modellen, die klären, ob z. B. ein Subjekt über eine Kette von anderen Subjekten einem anderen Subjekt ein Recht zuordnen bzw. sich von ihm holen kann. In seiner PhD-Thesis „Patterns of Safe Collaboration“ zeigt Spiessens, dass globale Variablen sich wie passive Subjekte modellieren lassen. Über globale Variablen wird Autorität unkontrolliert weitergegeben [Spie, S. 147].

Die Frage nach der Korrektheit von Capability-Systemen lässt sich demnach zwar algorithmisch in endlicher Zeit entscheiden (wenn auch teuer durch Traversierung des gesamten Graphen), die Antwort lautet aber leider immer, dass alle Rechte an alle Beteiligten gelangen können – und somit keinerlei Einschränkung von Autorität möglich ist. Das ist natürlich kein befriedigendes Resultat, und die Folge davon war eine Diskreditierung von capability-basierten Systemen, wie sie im bekannten Paper von Boebert über die Unfähigkeit von derartigen Systemen, die *-Properties zu erzwingen, zum Ausdruck kam [Boeb].

14.1.3 Differenzierung des Take/Grant-Ansatzes

Auf die beschriebene Problematik der unkontrollierten Delegation von Autorität bei Take/Grant-Modellen wurde in der Folge mit vier unterschiedlichen Ansätzen geantwortet:

- Einführung neuer Subjekte, die in Bezug auf Take/Grant- bzw. Read/Write-Rechte sowohl aktiv (invoke) als auch passiv (respond) sein können [VS],
- Einführung neuer, differenzierterer Rechte [Shap, VS],
- Aufsplittung von Subjekten in Composite Subjects [SMVS],
- Komplexe Verhaltensmodellierung der Kollaboration von Subjekten durch logische Constraints und Propagatoren [VS, Spie].

Der erste Ansatz beseitigt die übertrieben kollaborativen Annahmen des ursprünglichen Take/Grant-Modells, nämlich dass der bloße Zugriff über Take oder Grant immer impliziert, dass das Ziel des Zugriffs völlig damit einverstanden ist, dass alle Capabilities von ihm entweder kopiert werden oder ihm neue Capabilities beliebig übergeben werden können.

Durch die Einführung der neuen Eigenschaften *invoke* bzw. *respond* in Bezug auf Rechte ergibt sich ein dreidimensionales Modell (Abb. 14.8) für Zugriffe auf Subjekte anhand der folgenden Dimensionen:

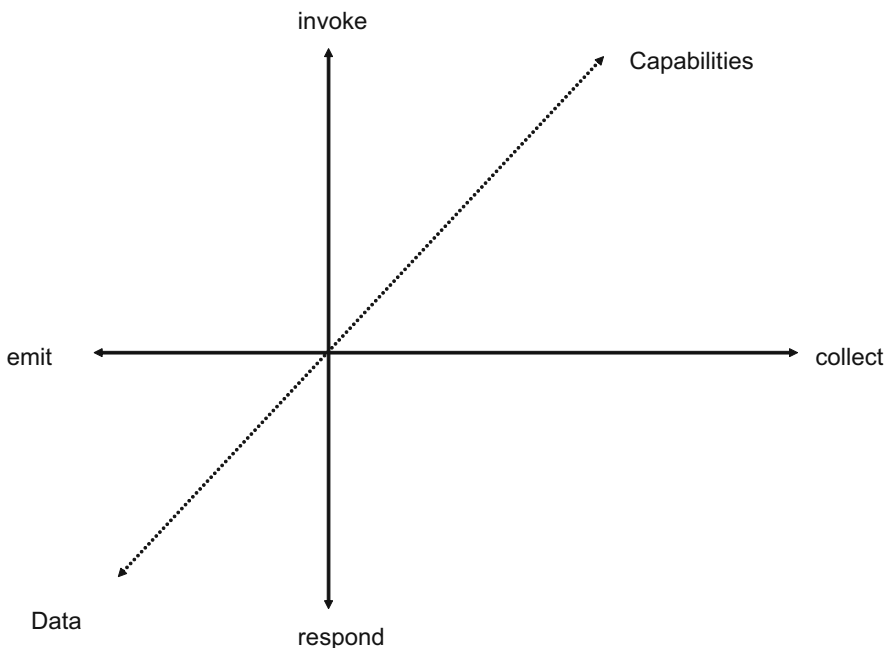
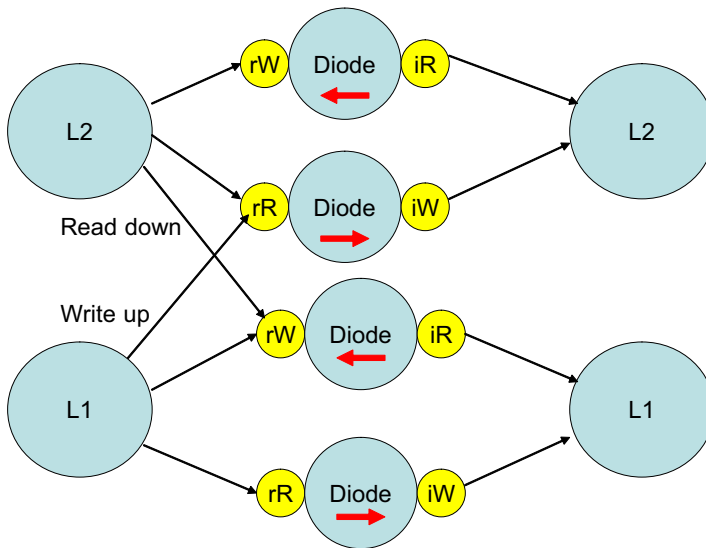


Abb. 14.8 Erweiterungsdimensionen von Take/Grant-Modellen

- *Invoke/Respond* (das entspricht dem Unterschied zwischen aktiven und passiven Subjekten),
- *Take/Grant* (der Übergabe von Rechten in Form von Capabilities, d.h. der Autoritätsgraph wird verändert),
- *Read/Write* (der Übergabe von Informationen ohne Änderung des Autoritätsgraphen, d.h. es werden keine neuen Rechte erzeugt jedoch Einflüsse vorgenommen).

Mit diesem Modell lassen sich acht verschiedene Kombinationen von Rechten als Eigenschaften von Subjekten beschreiben. Um Rechte von einem Subjekt zu übernehmen, braucht das übernehmende Subjekt die *Invoke/Take*-Eigenschaft. Gleichzeitig muss das Subjekt, dessen Rechte übernommen werden sollen, die *Respond/Take*-Eigenschaft besitzen. Analoges gilt für die anderen Kombinationen. Als Beispiel geben wir in Abb. 14.9 die Modellierung der *-Properties in einem capability-basierten System.

Van Roy und Spiessens verwenden diesen Ansatz, um Sicherheitsbausteine zu modellieren wie die „Datendiode“ (vgl. Abb. 14.9), die Informationsflüsse in die falsche Richtung unterbinden [VS]. In Capability-Systemen wird, wie eingangs erwähnt, das Sicherheitsmodul über initiale Subjekte und deren Rechte als Teil der Architektur in die Software integriert. So kann man über Softwarebausteine Mo-



Read-Down Diode: rW/iR ($L1 < L2$):
 Write-Up Diode: rR/iW ($L2 > L1$)
 Only data, no authority (capability),
 no assumptions about L1/L2 behavior

Abb. 14.9 Implementation der *-Properties durch Capabilities

dule einer Applikation voneinander isolieren oder nur einen kontrollierten Austausch von Informationen statt z. B. Rechten gestatten.

Aber auch mit diesem erweiterten Modell zeigt sich schnell, dass bestimmte komplexere Verhaltensweisen von Sicherheitsbausteinen – wie der sogenannte Caretaker – auch damit nicht modelliert werden können: Beispielsweise sind bei Sicherheitsbausteinen, die Proxy-Funktionalität besitzen, die bei der Konstruktion übergebenen Rechte (Capabilities) problematisch. Diese dürfen vom Proxy zwar verwendet werden, um übergebene Daten oder Rechte weiterzuleiten, sie dürfen aber nicht selbst weitergeleitet werden. Damit sind Take/Grant-Rechte beim Einsatz von Proxies verboten.

In Abb. 14.10 dürfte z. B. Bob weder Take- noch Grant-Rechte am Proxy-Objekt erhalten, denn sonst könnte er entweder direkt (take) oder durch das oben gezeigte Konstrukt eines zusätzlichen Hilfs-Subjektes die Zugriffsrechte des Proxies auf Carol übernehmen. Der Proxy RP hingegen ist auf diese Rechte an Carol angewiesen, um als Proxy arbeiten zu können, d. h. um Rechte von Alice weiterreichen zu können.

Eine mögliche Lösung wäre, den Proxy lediglich auf den Austausch von Daten zu reduzieren, d. h. tatsächlich Take- und Grant-Rechte zu verweigern. Damit wäre auch keine Weitergabe von Take- oder Grant-Rechten durch den Proxy hinweg möglich (Abb. 14.11).

Der zweite Ansatz zur Differenzierung des ursprünglichen Take/Grant-Modells konzentriert sich auf die Erzeugung neuer Typen von Rechten. Van Roy und Spiessens erfanden dazu zwei Rechte, die die Weitergabe von initialen Rechten

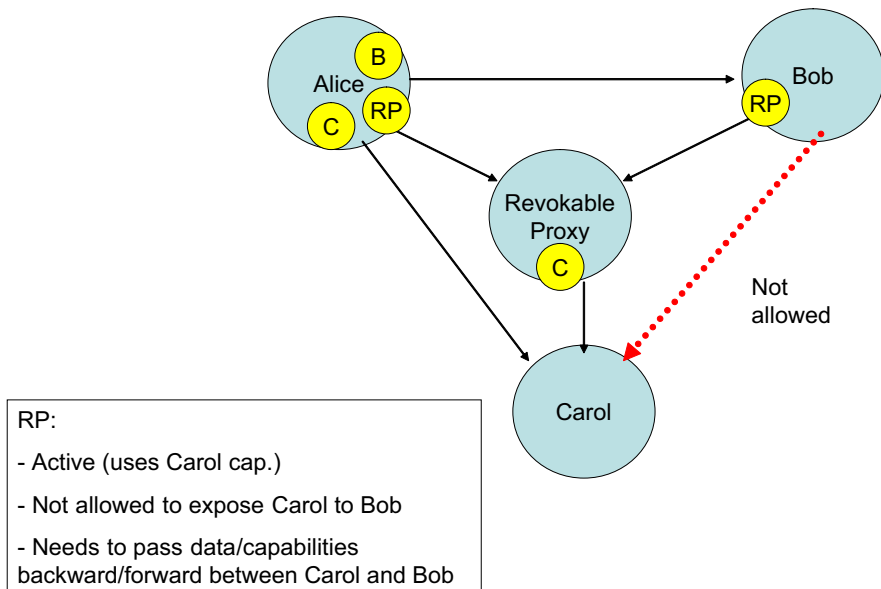


Abb. 14.10 Caretaker Sicherheitsbaustein

entweder erlauben oder verbieten [VS]. Initiale Rechte sind solche, die das Objekt bei seiner Konstruktion vom Erzeuger mitgeteilt bekommt (ähnlich den Objekten, die in OO-Sprachen durch den Konstruktor bei der Erzeugung eines Objektes mitgegeben werden). Dadurch steigt die Zahl der möglichen Rechtekombinationen, und weitere sinnvolle Sicherheitsbausteine sind damit modellierbar. So macht der revokable Proxy nun keine Probleme mehr (Abb. 14.12).

Aber auch damit ist das Autoritätsmodell von manchen Bausteinen noch nicht beschreibbar. Der dritte Ansatz führt keine weiteren Rechte ein, sondern zerlegt einen Sicherheitsbaustein in verschiedene Teilbausteine, die mit unterschiedlichen Rechten ausgestattet werden [SMVS]. Damit lässt sich erreichen, dass z. B. ein Proxy unterschiedliches Verhalten gegenüber zwei Subjekten zeigen kann. Der große Nachteil dieser Methode besteht darin, dass bislang einfache Sicherheitsbausteine plötzlich eine große interne Komplexität erreichen. Somit ist diese Methode für Entwickler nicht unbedingt leicht einsetzbar.

Der vierte und letzte Ansatz konzentriert sich noch stärker auf die Verhaltensbeschreibung der Subjekte und führt eine auf Logik und Constraints basierende Beschreibung dieses Verhaltens ein. Van Roy und Spiessens verwenden dazu eine interne Knowledgebase für Subjekte, von der ihr externes Verhalten abgeleitet werden kann [VS]. Dieses externe Verhalten führt zu Kollaboration mit anderen Subjekten, die wiederum auf das ursprüngliche Subjekt Auswirkungen und dessen interne Knowledgebase abändert – was wiederum die externen Kollaborationsregeln des Subjekts ändert. Ziel dieses Ansatzes ist die komplette Modellierung einer Software-Konfiguration und die algorithmische Prüfung abgeleiteter Konfi-

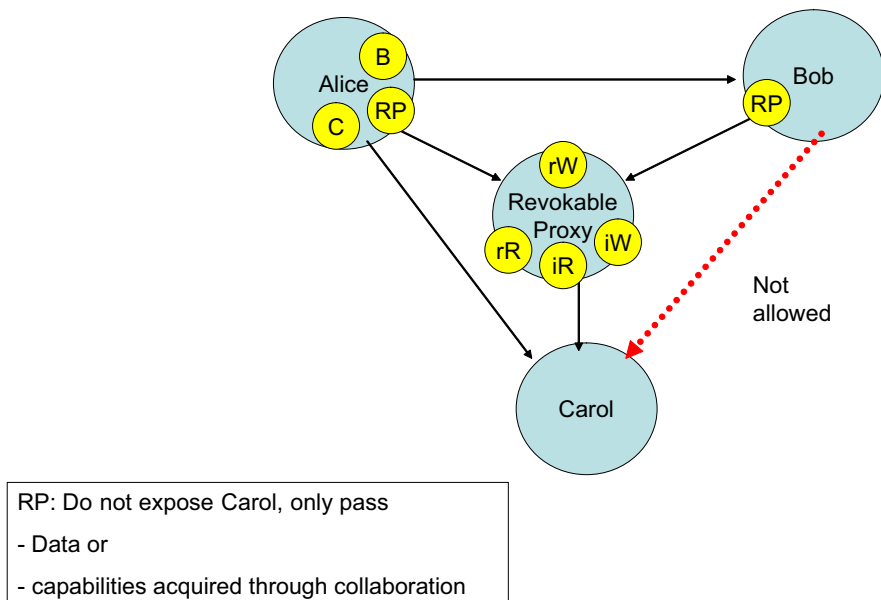


Abb. 14.11 Ausschluss der Propagation von Autorität erlaubt

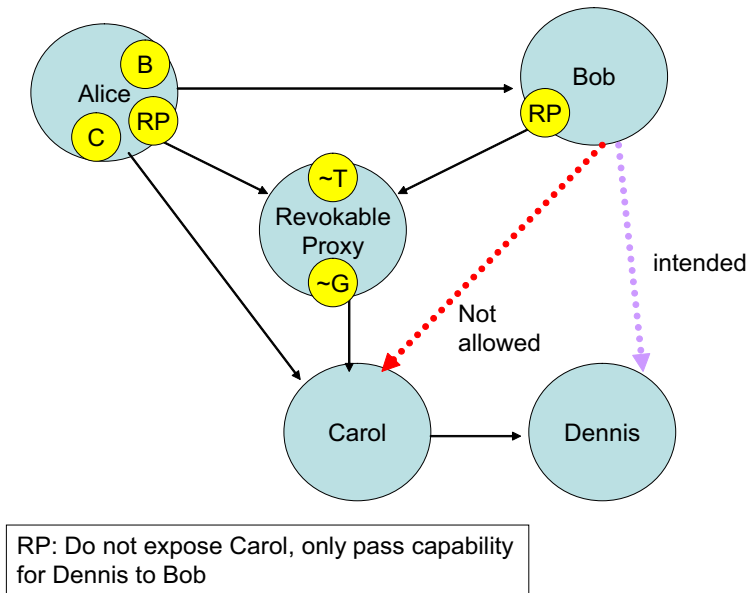


Abb. 14.12 Verbot der Weitergabe initialer Autorität

gurationen. Van Roy und Spiessens verwenden dazu Techniken wie Concurrent Constraint Programming, um Tools zum Modell-Checking zu erzeugen (siehe dazu Abschn. 14.2).

Allen diskutierten Ansätzen ist gemeinsam, dass sie die Wirkungen eines bloßen Zugriffsrechts (Access) deutlich einschränken und dadurch das Take/Grant-Modell wesentlich geeigneter für die Modellierung komplexen Sicherheitsverhaltens z. B. in Bausteinen machen. Spiessens spricht deshalb auch treffend von „Delegation: Just in Time Least Authority“ [Spie, S. 188ff.]. Es ist vorstellbar, dass sich diese Methoden noch erweitern lassen. So könnte man überlegen, ob Typinformationen über Capabilities bzw. deren Objekte in Objekt-Capability-Systemen ebenfalls in die Verhaltensregeln eingehen könnten.

In Abschn. 14.2 zeigt unser Gastautor Dr. Fred Spiessens die Mächtigkeit formaler Ansätze am Beispiel des Confused-Deputy-Problems auf. Dieses Problem haben wir bereits informell in Kap. 10 diskutiert. Die nötigen Rechte teilen sich auf in vier Bereiche: Rechte, die nur der Client (der Caller) benötigt, Rechte des Clients, die der privilegierte Code (der Callee) benötigt, Rechte des privilegierten Codes, die während des Clientrequests nötig sind und Rechte des privilegierten Codes, die unabhängig vom Clientrequest sind (Abb. 14.13).

Das Ergebnis unserer Diskussion in Kap. 10 war, dass durch ACLs oder Sandboxes entweder zu viele Rechte vergeben werden (der privilegierte Code erhält Zugriff auf die Files *aller* Clients), oder zuwenige (der privilegierte Code darf nur seine eigenen Files bearbeiten). Wenn man vom potenziellen Denial-of-Service-Problem durch die Allokation der Resource C oben absieht, dann besteht das Prob-

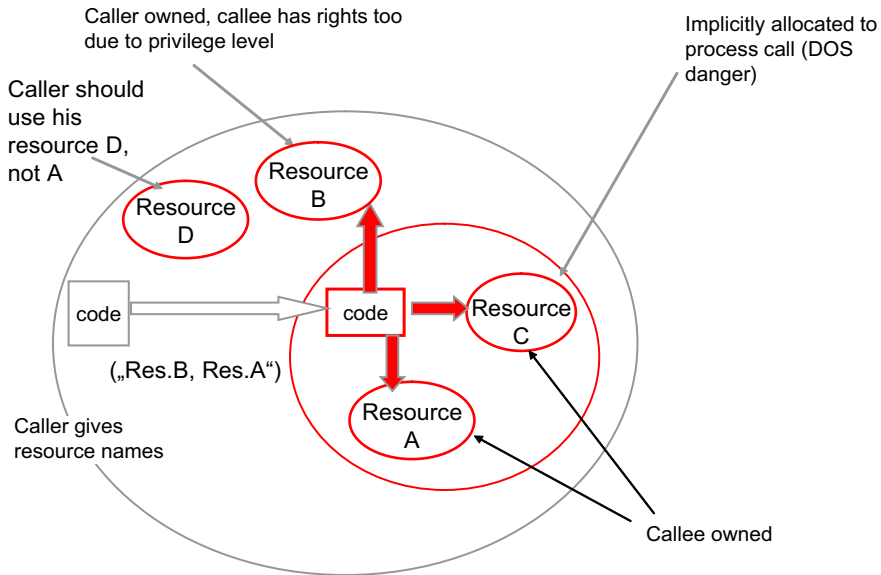


Abb. 14.13 Autorität, Interaktion und benötigte Rechte

lem darin, dass der Client die nötigen Files jeweils granular an den privilegierten Code übergeben kann, aber zu keiner Zeit Zugriff auf die Resource C erhalten darf.

Der privilegierte Code könnte alternativ in zwei Teile gespalten werden. Der eine Teil beschäftigt sich nur mit den Client-Ressourcen und nimmt daher zur Laufzeit die Identität des Clients an (Impersonation). Dadurch würde verhindert, dass der Client den Namen der Resource C erfolgreich in ein Objekt umwandeln kann, denn der privilegierte Code würde mit den Rechten des Clients auf Resource C zugreifen und scheitern. Allerdings würde die Impersonation des Clients durch einen Teil des privilegierten Codes diesem das Recht des Zugriffs auf alle Files des Clients gestatten.

Letztlich braucht es also einen dynamischen Mechanismus für die Delegation von Rechten – sei es durch die Manipulation des Call-Stacks, wie es in Java möglich ist [Spie, S. 190f.], oder durch die Verwendung von Objekt-Capabilities, sodass der privilegierte Code gar keine statischen Rechte auf Files des Clients benötigt.

Der Mechanismus allein kann jedoch die Richtigkeit des Codes nicht bewerkstelligen. Er muss auch korrekt eingesetzt werden. Confused-Deputy-Szenarien sind notorisch schwierig zu behandeln in der Programmierung, und es entstehen schnell subtile Sicherheitsprobleme. Gibt es vielleicht eine Möglichkeit, die minimal nötigen Rechte der Beteiligten Objekte in so einem Szenario automatisch zu ermitteln, und zwar so, dass einerseits die Funktionsfähigkeit des Codes garantiert ist und andererseits nur erlaubte Zugriffsmöglichkeiten zugelassen werden? Fred Spiessens beschreibt in Abschn. 14.2 das Konzept von Knowledge Behavior Models und ihre Implementation in Form einer Sprache bzw. eines Werkzeugs zur formalen Analyse der Sicherheit in Confused-Deputy-Szenarien.

14.2 Der Scoll-Ansatz in der Sicherheitsanalyse (Fred Spiessens)¹

Die formale Sicherheitsanalyse verwendet formale Modelle, um die Sicherheit oder Unsicherheit eines Systems zu analysieren und zu beweisen. Einige Ansätze konzentrieren sich auf das Finden beweisbarer Schwächen, während andere versuchen, bestimmte Sicherheitsaspekte existierender Systeme zu beweisen. Das erste Problem kann dadurch gelöst werden, dass ein einzelner Ausführungspfad im System gefunden wird, der die Schwäche demonstriert. Das zweite Problem muss alle möglichen Ausführungspfade in Betracht ziehen. In diesem Abschnitt zeigen wir einen formalen Ansatz, der dabei helfen soll, Schwächen bereits im Design zu vermeiden. Dieser Ansatz besteht aus den folgenden Teilen:

- Knowledge Behavior Models (KBM): Ein Formalismus, um Systeme interagierender Entitäten zu modellieren,
- Safe Collaboration Language (Scoll): Eine deklarative Sprache, um Sicherheitsprobleme auszudrücken,
- Scollar: Ein Werkzeug, das Lösungen für die in Scoll beschriebenen Sicherheitsprobleme berechnet.

Der Scoll-Ansatz wird im Detail erklärt in „Patterns of Safe Collaboration“ [Spie]. Das Scollar-Tool ist Open-Source und kann heruntergeladen werden von <http://www.scoll.evoluware.eu>. Bevor wir uns auf die formalen Details konzentrieren, wollen wir ein weiteres Mal das Confused-Deputy-Problem diskutieren. Wir werden es als Beispiel in den folgenden Abschnitten verwenden.

14.2.1 Das Confused-Deputy-Problem

Software mit gutem Design enthält normalerweise Teile (Komponenten, Objekte, Funktionen), die einen Service für andere Teile erbringen (z. B. Berechnungen durchführen, eine Datenbank verändern, die Präferenzen des Users hervorheben etc.). Lassen Sie uns diese Teile vorerst „Subjekt“ nennen. In einem sicheren System können Subjekte nicht alles tun: Sie brauchen dazu die richtige Permission (Erlaubnis). Das bedeutet, dass Service-Subjekte häufig die Permissions ihrer Clients benötigen, um ihre Aufgabe erledigen zu können, zum Beispiel eine Permission, um eine Eingabedatei zu lesen, in der die Formel steht, die berechnet werden soll und eine andere Permission, um die errechneten Resultate in eine Ausgabedatei zu schreiben. Ein Subjekt wird „Deputy“ genannt, wenn es Dinge im Auftrag eines Clients erledigt, der dazu seine Permissions an das Subjekt delegiert hat.

¹ Die Autoren danken Dr. Spiessens, Senior Researcher in der Forschungsgruppe „Security of Embedded Systems“, die von Prof. S. Etalle an der TU Eindhoven geführt wird, sehr herzlich für diesen Beitrag, der aus dem Englischen übersetzt wurde.

Das Problem

Ein Deputy wird „confused“ (verwirrt) genannt, wenn er aus Versehen seine eigenen Permissions verwendet, obwohl er stattdessen (nur) die Permissions des Clients hätte verwenden sollen. Zum Beispiel könnte einer der Clients den Deputy damit beauftragen, die Ergebnisse der Berechnungen in eine Datei zu schreiben, in die nur der Deputy schreiben darf. Der Deputy hatte wohl die Absicht, diese Datei zu benutzen, aber nur um daraus zu lesen oder Zwischenresultate früherer Berechnungen anzuhängen, um zukünftige Berechnungen zu beschleunigen. Wie kann sich der Deputy dagegen schützen?

Die Ursache

Wenn ein Deputy keine Möglichkeit hat auszuwählen, welche der delegierten Permissions seiner Clients für verschiedene Teile seines Auftrags verwendet werden sollen, dann kann er „verwirrt“ werden. Dies gilt umso mehr, wenn er nicht einmal unterscheiden kann zwischen seinen eigenen Permissions und denen, die von seinen Clients an ihn delegiert für bestimmte Aufgaben wurden. Diese Situation tritt in vielen Protection Systems auf, die einen unabhängigen Mechanismus (üblicherweise „Reference Monitor“ genannt) verwenden, um die Permissions eines Subjekts zur Laufzeit auszuwählen. Der Reference Monitor verwendet eine Access Control List (ACL), um die Permissions eines Subjekts zu prüfen. Er kann nicht wissen, für welchen Teil der Aufgabe der Deputy eine Permission verwenden möchte.

Die Verantwortlichkeit zwischen Software und System auf diese Weise aufzuteilen, wird „separation of concerns“ genannt. Es handelt sich dabei um ein gutes Designprinzip, denn es erlaubt Programmierern, sich darauf zu konzentrieren, dass das Programm das Richtige tut, während es Systemadministratoren erlaubt, sich auf die Verhinderung von falschen Aktionen zu konzentrieren.

In dem geschilderten Fall ist jedoch diese Trennung der Zuständigkeit nicht angebracht, denn die Schreiberlaubnis des Deputies sollte davon abhängen, was er gerade tun will: Z. B. Zwischenresultate für spätere Verwendung anhängen oder die Datei mit dem Report der endgültigen Ergebnisse überschreiben. Der Deputy sollte die Erlaubnis haben, seine eigene Datei zu jeder Zeit und für seine eigenen Zwecke schreiben und lesen zu können – aber nicht für die Zwecke seines Clients – außer natürlich dieser Client hat ebenfalls die Erlaubnis, diese Datei zu lesen und zu schreiben.

Capabilities als Rettung

Der Client muss dem Deputy eine Bezeichnung seiner Dateien zur Verfügung stellen, typischerweise ein Pfadname oder eine URL, die als Argument in einer Message oder einem Funktionsaufruf übergeben werden. Capabilities sind Bezeichnungen, die nicht gefälscht werden können und die außerdem eine nicht

abtrennbare Permission enthalten, sodass der Reference Monitor nicht nach Permissions suchen muss. In einem Capability-System kann der Deputy verlangen, dass seine Clients ihm eine Lese-Capability auf die Inputdatei und eine Schreib-Capability auf die Outputdatei zur Verfügung stellen. Auf diese Weise kann der Deputy seine eigenen Capabilities für die Zwischenresultate und die Capabilities seiner Clients für die Endresultate verwenden. Niemand muss dabei irgendeine Form von Buchführung machen oder die Access Control Lists befragen. Der Client kann keine Capability übergeben, die die private Datei des Deputies betrifft – außer er hätte sie auf eine zulässige Weise erhalten (also nicht durch Fälschung). Der Deputy kann also sicher sein, dass die richtige Erlaubnis verwendet wird, wenn er die Capabilities der Clients verwendet.

Diese Überlegung zeigt, dass man um sichere Software zu erzeugen nicht nur Systemadministratoren, sondern auch die Designer und Entwickler von Programmen einbeziehen muss: Nur die Entwickler kennen die detaillierten Verwendungszwecke der Objekte, Funktionen und Komponenten, die sie entwickeln. Der Capability Ansatz ist einfach und braucht keinen zustandsbehafteten Referenz-Monitor (einen mit einer Liste). Das geschilderte Problem des Confused Deputy und seine auf Capabilities basierende Lösung wurde zuerst von Norman Hardy publiziert [Hard].

Eine interessante Alternative – genannt „stack walking“ – erlaubt dem Programmierer beim Senden einer Message (oder dem Aufruf einer Funktion) Laufzeit-Tags auf dem Stack hinzuzufügen, um anzuzeigen, welche Permissions des momentanen Subjekts an das aufgerufene Subjekt delegiert werden sollten. Dieser Ansatz ist jedoch nur bei stackbasierten Systemen anwendbar und schließt Interprozessaufrufe sowie Inter-Thread-Aufrufe in einem parallelen oder verteilten Kontext aus. Ein weiterer Nachteil besteht darin, dass der Stack dadurch effektiv zu einer les- und schreibbaren Variable mit globalem und dynamischem Scope wird und zur offenen Datenkommunikation zwischen ansonsten eingeschränkten Subjekten verwendet werden kann.

Es ist offensichtlich, dass die Delegation von Permissions, sei es als Capabilities oder durch andere Mechanismen, sehr sorgfältig überwacht werden muss, um ihre Weitergabe an nicht-vertrauenswürdige Komponenten, die sie missbrauchen könnten, zu verhindern. In den nächsten Abschnitten werden wir eine capability-basierte Lösung des Confused-Deputy-Problems modellieren und analysieren und dabei entdecken, wie wir Deputies entwerfen können, die beweisbar nicht „verwirrt“ werden können.

14.2.2 Kernkonzepte der Sicherheitsanalyse

Bevor wir das formale Modell beschreiben, in dem wir praktische Sicherheitsprobleme ausdrücken und analysieren können, müssen wir einige grundlegende Konzepte diskutieren, die den Kern der Sicherheitsanalyse betreffen.

Permission versus Authority

Eine *Permission* erlaubt einer Entität eine Aktion durchzuführen, z. B. den Bildschirm zu manipulieren oder User-Input zu erhalten. Auf der anderen Seite beschreiben Miller und Shapiro in [MiSh] *Authority* als generellen Effekt, den eine Entität in einem System haben kann. Der direkte Effekt einer erlaubten Aktion ist eine einfache und direkte Form von Authority, z. B. ist der Effekt der Aktion „manipuliere den Bildschirm“ – falls erlaubt – die Änderung von Informationen, die auf dem Bildschirm sichtbar sind. Aber es gibt auch andere, indirekte und potenziell komplizierte Formen von Authority, die nicht einer einzelnen Aktion, kontrolliert durch eine Permission, entsprechen.

In Abb. 14.14 kann die Entität Alice die Anweisungen von Bob lesen und interpretieren, um den Bildschirm entsprechend zu manipulieren. Die resultierende Authority ist die gleiche, als hätte Bob die Permission zur Manipulation des Bildschirms selber gehabt. In der Abbildung werden Knoten, die nicht-vertrauenswürdige (oder unbekannte) Subjekte repräsentieren, stark umrandet dargestellt. Durchgezogene Pfeile repräsentieren Permissions, während gestrichelte Pfeile (indirekte) Authority anzeigen.

Das Verhältnis zwischen Permission und Authority ist kausal: Letztlich wird alle Autorität durch die Benutzung von Permissions erreicht. Aber diese Relation ist komplex und hat viele Aspekte, die bei der Modellierung eines sogenannten *Protection Systems* wichtig sind.

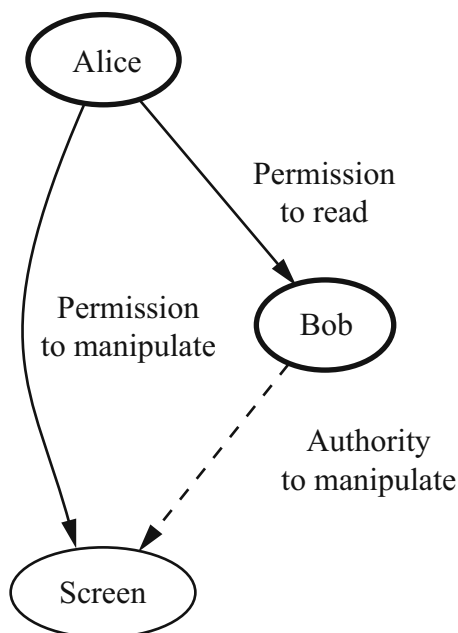


Abb. 14.14 Indirect Authority: Bob kann den Bildschirm indirekt manipulieren

Protection Systems

Wir verwenden den Begriff „Protection System“ für einen Mechanismus, der Permissions prüft, erzwingt und verwaltet, um die Entwicklung eines Systems aus einem gegebenen initialen Zustand zu kontrollieren. Üblicherweise ändert sich der Zustand eines Systems durch Aktionen bestimmter Entitäten im System. Wir nennen diese aktiven Entitäten „Subjekte“. Der Status eines Protection Systems wird „Protection State“ genannt und ist meist definiert als die Menge von Permissions, die einem Subjekt in einer bestimmten Phase in der Entwicklung eines Systems zur Verfügung stehen. Er wird repräsentiert als Access Control Matrix. Der Protection State kann sich ändern, z. B. wenn Entitäten ihre Permissions an andere Entitäten übergeben (delegieren, kopieren).

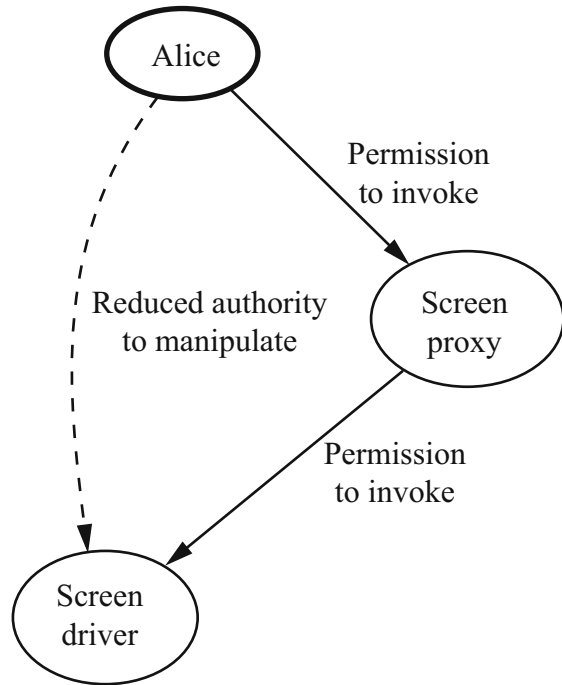
Permissions sind immer *positiv*, d. h.: Wenn wir zwei Protection States A und B eines Protection Systems betrachten, sodass A eine Teilmenge von B ist, dann sind in B nicht weniger Aktionen erlaubt als in A. Wir könnten eine Access Control Matrix verwenden, um Rechte als binäre Prädikate zwischen Subjekten und Objekten in unserem formalen Modell auszudrücken, z. B. `read(alice,bob)`. Es ist jedoch nicht besonders praktisch, komplexere Autoritätsbeziehungen in einer ACM zu modellieren. Um komplexe Sicherheitsprobleme zu lösen, muss unser Protection State alles einschließen, was direkte und indirekte Autorität beeinflusst. Das schließt Relationen und Prädikate beliebiger Stelligkeit ein, die indirekte Autorität beeinflussen (was kann passieren), das Verhalten der Subjekte (die Intentionen von Subjekten, das System zu beeinflussen), sowie das Wissen von Subjekten (der Teil eines Systems, der ein Subjekt und sein Verhalten beeinflussen kann).

Das Principle of Least Authority (POLA)

Das Prinzip der geringstmöglichen Autorität bestimmt, dass jede Entität nicht mehr Autorität besitzen sollte als für die Funktionalität, die es bereitstellen soll, nötig ist. Es ist eine extreme Interpretation von Saltzer und Schroeders „Principle of Least Privilege“ [SaSch], das oft in einer schwächeren Form als „Principle of Least Permissions“ verstanden wird (siehe auch Abschn. 3.10). Angesichts der vielen und komplexen Weisen, in denen Autorität vorkommen kann, ist POLA eher als Richtschnur zur Entwicklung sicherer Software denn als strikte Regel zu verstehen.

Wenn eine Software-Komponente Alice ein bestimmtes Gebiet auf einem Bildschirm manipulieren muss, dann ist in Übereinstimmung mit dem Prinzip der geringstmöglichen Permissions Alice die Erlaubnis zur Manipulation des Bildschirms zu geben. Aber dies wäre nicht streng in Übereinstimmung mit dem Prinzip der geringstmöglichen Autorität, denn der ganze Bildschirm stellt in diesem Fall nicht genau das dar, was Alice zur Erfüllung der Aufgabe benötigt, nämlich die Erlaubnis, nur einen Teil des Bildschirms zu manipulieren.

Abb. 14.15 Indirekte Autorität: Vom Bildschirm Proxy wird erwartet, dass er die Autorität von Alice, den Bildschirm zu manipulieren, reduziert



In einem Objekt-Capability-System können wir es besser machen, indem wir Alice eine Capability auf einen programmierten Bildschirm-Proxy geben, bei dem man darauf vertrauen kann, dass er Anweisungen zur Manipulation des Bildschirms so durchführt, dass ihre Effekte auf einen bestimmten Teilbereich des Bildschirms eingeschränkt sind. Abbildung 14.15 illustriert diese Situation.

Das Erzwingen von Sicherheit durch Capabilities beruht stark auf programmierten Entitäten (siehe auch Kap. 10), um die Autorität zu reduzieren, die sie anderen, mit denen sie interagieren, zur Verfügung stellen.

Als direkte Konsequenz aus POLA muss alle Autorität, die „ambient“ verfügbar ist (also ohne einen expliziten Transfer der Autorität durch eine andere Entität) vermieden werden. Ambiente Autorität wird in Sprachumgebungen wie der Sprache E und ihrer virtuellen Maschine [E] verhindert, indem das Ladeprogramm sicherstellt, dass alle Software ohne Zugriff auf andere Laufzeit-Entitäten, die Autorität anbieten könnten, geladen wird.

Verhaltensbasierte Sicherheitsanalyse und Sichere Approximation

Aufgrund der vielen komplexen Weisen, in denen Autorität vorkommen kann, ist eine vorgängige Worst-Case-Analyse der Sicherheitsanforderungen nötig. Der Scoll-Ansatz zur Sicherheitsanalyse basiert auf einem expliziten Modell der Software, der folgende Aspekte der Einhaltung von Sicherheit modelliert:

- auf welche Weise das Protection System die Propagation von Permissions einschränkt,
- unter welchen Umständen die vertrauenswürdigen Entitäten ihre Permissions (nicht) nutzen.

Den zweiten Aspekt benennen wir als Verhalten. Verhaltensbasierte Sicherheitsanalyse erlaubt uns, eine sichere Approximation der erreichbaren Autorität zu berechnen. Diese Näherungslösung kann genauer gemacht werden, indem das Verhalten detaillierter modelliert wird.

Um die Weitergabe der Autorität in einer Softwareimplementation (oder Softwareanalyse) sicher approximieren zu können, werden wir eine monotone Approximation des Verhaltens der Software-Entitäten modellieren. Das bedeutet, dass wir nicht in der Lage sein werden, direkt im Modell den Fakt auszudrücken, dass eine Softwarekomponente eine Permission nicht mehr benutzen wird – unter den gleichen Umständen, unter denen die Komponente vorher ihre Permission benutzt hatte (das verhaltensbasierte Äquivalent zu einem Widerruf einer Permission). Das ist eine Einschränkung im momentanen Scoll-Ansatz, die in zukünftigen Versionen aufgehoben werden soll.

Der monotone Ansatz hat jedoch den praktischen Vorteil, dass sichere Approximationen des Verhaltens von Software-Entitäten zur Laufzeit einfach aus dem Programmcode erschlossen werden können.

14.2.3 KBM: Konservative Modelle für Systeme interagierender Entitäten

Knowledge Behavior Models (KBM) wurden entworfen um:

- nicht nur die potenziellen Effekte der Verteilung von Permissions zu modellieren, sondern alle potenziellen Effekte, die für die Sicherheit relevant sind, eingeschlossen indirekte potenzielle Effekte wie die indirekte Autorität in Abb. 14.14;
- nicht nur die einschränkende Effekte von Permissions zu modellieren, sondern auch die eingeschränkte Benutzung von Permissions durch vertrauenswürdige Teile der Software;
- flexible Ausdruckskraft zur Modellierung verschiedener Teile einer Software im jeweils besten Detaillierungsgrad anzubieten;
- die inkrementelle Verfeinerung existierender Modelle zu unterstützen.

Vor einer formalen Definition eines KBM erklären wir die Teile, aus denen es besteht.

Prädikate und Fakten

Eine essenzielle Rolle spielen in KBMs die Prädikate, mit denen alle Permissions, Autorität und Verhaltensmerkmale ausgedrückt werden. Prädikate können beliebige Stelligkeit besitzen und stellen technisch die Kombination eines Labels p mit einer Funktion dar, die n -Tupel von Subjekten auf die Werte wahr oder falsch abbildet. Werden in die n -stelligsten Tupel konkrete Subjekte eingesetzt, so bezeichnet man diese Prädikate als *Fakten* (facts). Das erste Subjekt im Tupel heißt „base subject“, und das Prädikat bezeichnet seine Beziehung zu den anderen aufgeführten Subjekten. In alternativen Notationen kann das erste Subjekt auch aus dem Tupel herausgezogen werden. Eine ausführlichere Darstellung der Verwendung von Prädikaten zum Ausdruck von Sicherheitseigenschaften bietet Spiessens [Spie, S. 83ff].

Wir verwenden unterschiedliche Arten von Prädikaten, um drei verschiedene Aspekte des Protection States zu beschreiben:

- **State-Prädikat:** Üblicherweise binär, repräsentieren sie den Teil eines Protection State, der für die Subjekte nicht direkt sichtbar ist. Zum Beispiel könnte das Prädikat `access(A, B)` dazu verwendet werden, die Permission von Subjekt A zum Zugriff auf Subjekt B auszudrücken. State-Prädikate wurden ursprünglich *Permission-Predikate* in [Spie] genannt.
- **Verhaltensprädikate** (Behavior predicates): Von beliebiger Stelligkeit, repräsentieren sie den positiven Einfluss, den ein Subjekt auf die tatsächliche Benutzung einer Permission und/oder auf den Effekt der Benutzung der Permission hat. Zum Beispiel könnte das Prädikat `sendTo(A, B, X)` verwendet werden, um auszudrücken, dass das Subjekt A willens ist, das Subjekt B mit dem Argument X aufzurufen.
- Um zu unterstreichen, dass es sich dabei um das Verhalten von A handelt, verwenden wir eine alternative Notation, die das erste Argument vor die Prädikatsbezeichnung und getrennt durch einen Doppelpunkt stellt: `A:may.sendTo(B, X)`. Per Konvention stellen wir dem Bezeichner eines Prädikates oft ein „may“ voran, um anzuzeigen, dass wir ein *mögliches* Verhalten modellieren. Es handelt sich dabei um Verhalten, das wir aufgrund unseres begrenzten Wissens über die Einschränkungen im Verhalten der Entität nicht sicher ausschließen können, oder das zur sicheren Approximation des Verhaltens der Entität innerhalb des KBM dient.
- **Wissensprädikate** (Knowledge predicates): Von beliebiger Stelligkeit, repräsentieren sie potenzielle Effekte, die für ein Subjekt sichtbar sind. Wie Verhaltensprädikate besitzen auch sie eine alternative Notation, um zu unterstreichen, welche Entität das Wissen besitzt: `A:did.sendTo(B, X)` zeigt an, dass A den betreffenden Fakt kennt. Per Konvention fügen wir

dem Bezeichner ein „did“ voran, um anzuzeigen, dass wir Wissen über eine erfolgreiche Interaktion modellieren, auf das sich ein Subjekt mit Sicherheit verlassen kann. Die Unsicherheit besteht lediglich in unserer Approximation des Verhaltens, nicht in unserem Modell der Effekte von erfolgtem Verhalten. Sobald ein KBM definiert wurde, ist es relativ einfach, das Modell zu verfeinern, indem die Prädikate verfeinert werden. Der Hauptgrund für die Verfeinerung des Verhaltens einer vertrauenswürdigen Entität liegt darin, ihre eingeschränkte Benutzung von Permissions genauer ausdrücken zu können.

Statt zum Beispiel das Prädikat `B:may.return(Y)` zu benutzen, um auszudrücken, dass B möglicherweise Willens ist, Y beim Aufruf zurückzugeben, brauchen wir vielleicht ein verfeinertes Prädikat, um auszudrücken, dass B nur dann Y zurückgibt, wenn es X im selben Aufruf erhalten hat. Das könnte im Prädikat `B:may.exchange(X, Y)` ausgedrückt werden. KBMs stellen ein generelles Framework dar, das Softwareentwicklern erlaubt, inkrementell verfeinerte Verhaltens- und Wissenprädikate hinzuzufügen, ohne dass das ganze KBM neu modelliert werden muss.

Eine statische Menge von Subjekten pro endlichem Protection State

Um Sicherheit beweisen und sichere abstrakte Muster finden zu können, ist es essenziell, dass die Berechnung der eventuellen Autorität (potenzielle Effekte) im Modell handhabbar bleibt. Das ist der Hauptgrund, warum KBMs eine statische, endliche Menge von Subjekten besitzen.

Damit ein KBM repräsentativ ist, müssen alle Software-Entitäten in so eine feste Menge von Subjekten modelliert werden. Subjekte, die mehr als eine Entität modellieren, werden „aggregierte“ Subjekte genannt. Damit Aggregation sicher ist, muss jedes Prädikat in der KBM, das ein aggregiertes Subjekt betrifft, potenziell für mindestens eine der Entitäten, die im Subjekt aggregiert sind, gelten können. Anders gesagt, muss das Prädikat wahr sein für das Subjekt im Modell, sobald es für eine der Entitäten in der Software *möglich* ist.

Zum Beispiel könnte das Subjekt Bob verwendet werden, um eine bestimmte Software-Entität zu modellieren, zusammen mit all den Entitäten, die Bob möglicherweise zur Laufzeit erzeugt. Dann zeigt das Prädikat `access(Bob, Alice)` die Möglichkeit an, dass eine der in Bob aggregierten Entitäten Zugriff auf eine der in Alice aggregierten Entitäten hat. Aggregation ist in vielen Situationen nützlich, und verschiedene Strategien können dabei benutzt werden. Der Beweis, dass Aggregation tatsächlich die Sicherheit der Approximation erhält, findet sich in [Spie].

Regeln zur Modellierung der Übergänge im Protection State

Ein KBM enthält Regeln, die ausdrücken, welche Aktionen möglich sind und welche potenziellen Effekte jede Aktion auf den Status des Protection Systems hat. In anderen Worten, die Regeln definieren mögliche Übergänge im Protection State.

In traditionellen Protection Systems sind die Permissions die (einzigen) Voraussetzungen der Regeln und die möglichen Effekte sind alle modelliert als Änderungen in der Permission Matrix. Zum Beispiel würde die Regel

$$\text{grant}(A, B) \wedge \text{read}(A, X) \Rightarrow \text{read}(B, X)$$

anzeigen, dass wenn A Grant-Permission auf B und Lese-Permission auf X hat, der mögliche Effekt der ist, dass B ebenfalls Lese-Permission auf X erhält, weil sie von A übergeben wurde.

Im Vergleich zu solchen auf Permissions fokussierten Modellen machen Take/Grant-Modelle zwei wichtige Schritte hin zu ausdrucksstärkeren Übergängen im Protection State:

1. Die einschränkende Wirkung von Verhalten wird mit einbezogen. Diese äußert sich z.B. darin, dass die Take/Grant-Regel „grant“, welche De-Jure-Rechte propagiert, dies nur tut im Fall aktiver Grant-Subjekte:

$$\text{grant}(A, B) \wedge \text{read}(A, X) \wedge A:\text{active}() \Rightarrow \text{read}(B, X).$$

2. Die potenziellen Effekte der Regeln können alle Arten von Autorität repräsentieren, nicht nur Permissions. So kann z.B. die Take/Grant-Regel „spy“, die De-Facto-Lese-Autorität propagiert, repräsentiert werden als:

$$\text{read}(A, B) \wedge \text{read}(B, C) \wedge A:\text{active}() \wedge B:\text{active}() \Rightarrow \text{deFactoRead}(A, C).$$

Diese beiden Erweiterungen bringen Take/Grant-Modelle sehr nahe an KBMs. Zwei weitere sind nötig:

3. Das Verhalten der vertrauenswürdigen Subjekte ist nicht länger auf „aktiv“ oder „passiv“ beschränkt, sondern kann mit beliebiger Präzision erfasst werden. So könnte die folgende Regel die Propagation von Zugriffsrechten in einem System mit aufrufbasierter Interaktion modellieren:

$$\begin{aligned} &\text{access}(A, B) \wedge \text{access}(A, X) \wedge A:\text{may.sendTo}(B, X) \wedge \\ &B:\text{may.accept}() \Rightarrow \text{access}(B, X) \wedge A:\text{did.sendTo}(B, X) \wedge \\ &B:\text{did.accept}(X). \end{aligned}$$

4. Das Verhalten eines Subjekts ist nicht länger statisch, sondern kann angepasst werden als Antwort auf das, was das Subjekt über seine Umgebung lernt. Zum Beispiel kann das Subjekt Alice programmiert werden, die Subjekte aufzurufen, die es als Übergabeargumente selbst in einem Aufruf empfangen hat. Das könnte in einem KBM als folgende Regel – nur angewandt auf das Verhalten von Alice – ausgedrückt werden:

$$\text{alice:did.accept}(X) \Rightarrow \text{alice:may.sendTo}(X,).$$

In KBMs werden die Regeln, die potenzielle Effekte von möglichen Aktionen ableiten, „Systemregeln“ genannt. Die Regeln, die die Reaktion eines Subjekts auf seine Umgebung ausdrücken, werden „Verhaltensregeln“ (behavior rules) genannt. Behavior rules können nur dynamisches Verhalten modellieren, das monoton anwächst, wenn das Subjekt mehr Wissen erhält.

Anmerkung: Die De-Jure-Regeln in Take/Grant-Systemen werden als ein Weg gesehen, den Datenfluss zwischen Entitäten zu analysieren. Da jedoch Datenfluss auch über versteckte Kanäle (covert channels) passieren kann, ist diese Analyse nicht sicher. Die Wissensprädikate in KBMs werden nicht zur Analyse des Datenflusses verwendet. Sie repräsentieren nicht beliebige Fakten, sondern das Wissen über ein Faktum, von dem das Subjekt auch weiß, dass es durch eine legale Interaktion erworben wurde. Das Fehlen solchen Wissens hält nicht-vertrauenswürdige Subjekte innerhalb der KBM nicht davon ab, ihr volles mögliches Verhalten zu zeigen, d. h. es stellt keine Einschränkung dar.

Knowledge Behavior Models

Wir kommen nun zur formalen Definition eines KBM:

Definition 1 KBM

Ein KBM ist ein Tupel $(S; P_s; P_k; P_b; Sys; Beh)$, in dem gilt:

- S ist eine endliche Menge von Subjekten.
- P_s ist eine Menge von State-Prädikat-Symbolen mit ihrer Stelligkeit.
- P_k ist eine Menge von Wissensprädikat-Symbolen mit ihrer Stelligkeit.
- P_b ist eine Menge von Verhaltensprädikat-Symbolen mit ihrer Stelligkeit
- Sys ist eine Menge von system rules.
- Beh ist eine Menge von behavior rules.

Da keine Negation in den Regeln einer KBM erlaubt ist, ist die logische Semantik einfach und äquivalent zu datalog [Gall]. Der Abschnitt zur Sprache Scoll weiter unten enthält Beispiele für System- und Verhaltensregeln.

Jedes KBM spezifiziert eine logische Ableitung von einer initialen Konfiguration (einer endlichen Menge von Fakten) zu der kompletten Menge von Fakten, die im Modell erreichbar ist. Wenn ein KBM eine sichere Approximation der Permissions, des Verhaltens und der Propagation der Autorität in einem Programm modelliert, dann korrespondiert die Autorität, die im Modell nicht abgeleitet werden kann, mit der Autorität, die innerhalb der Software nicht erreichbar ist. Daher erlauben uns KBMs, Sicherheitsprobleme in Software auszudrücken und zu berechnen. Um Sicherheitsprobleme auszudrücken, definieren wir drei weitere Konzepte:

Definition 2 Konfiguration

Eine *Konfiguration* in einem KBM ist eine Menge von Fakten (grounded predicates) von Prädikaten aus $P_s \cup P_k$ über Subjekte in S .

Definition 3 Sicherheitsanforderungen

Die *Sicherheitsanforderungen* eines KBM sind eine Menge von Fakten von Prädikaten in $P_s \cup P_k \cup P_B$, die nicht aus einer gegebenen Konfiguration durch die Regeln im KBM abgeleitet werden können.

Definition 4 Liveness-Anforderung

Die *Liveness-Anforderungen* eines KBM sind eine Menge von Fakten von Prädikaten in $P_s \cup P_k \cup P_B$, die nicht aus der initialen Konfiguration durch die Regeln im KBM abgeleitet werden können. Im Gegensatz zu den Sicherheitsanforderungen liefern Liveness-Anforderungen im KBM keinen Beweis über die modellierte Software. Sie werden nur dazu benutzt, die Suche nach sicherem Verhalten der Subjekte und sicheren Konfigurationen auf solche Lösungen zu beschränken, in denen die Einschränkungen nicht die Liveness-Anforderungen ausschließen.

Die folgenden Typen von Sicherheitsproblemen können mit einem KBM ausgedrückt werden:

Einfaches Sicherheitsproblem (Simple-Safety-Problem):

Gegeben seien ein KBM, eine initiale Konfiguration für das KBM und eine Menge von Sicherheitsanforderungen. Es stellt sich das Problem: Werden die Sicherheitsanforderungen erfüllt? In anderen Worten: Ist es unmöglich, eine der Sicherheitsanforderungen aus dem KBM und der initialen Konfiguration abzuleiten?

Praktisches Sicherheitsproblem (Practical-Safety-Problem):

Gegeben seien ein KBM, eine initiale Konfiguration für das KBM und eine Menge von Sicherheitsanforderungen und Liveness-Anforderungen. Es stellt sich das Problem: Werden sowohl die Anforderungen bezüglich Sicherheit als auch diejenigen bezüglich Liveness erfüllt? In anderen Worten: Ist es möglich, aus dem KBM und der initialen Konfiguration alle Liveness-Anforderungen und gleichzeitig keine der Sicherheitsanforderungen abzuleiten?

Problem der Verhaltensmaximierung (Behavior-Maximization-Problem):

Gegeben seien ein KBM, eine initiale Konfiguration für das KBM und eine Menge von Sicherheitsanforderungen und Liveness-Anforderungen und eine Menge von Subjekten deren Verhalten maximiert werden soll. Es stellt sich das Problem: Welche Möglichkeiten gibt es, das Verhalten der gegebenen Subjekte zu maximieren unter der Voraussetzung, dass Sicherheits- und Liveness-Anforderungen erfüllt bleiben müssen?

Verhaltensmaximierung sucht nach Lösungen, die das kooperative Verhalten einer Menge von vertrauenswürdigen Subjekten maximieren. Es kann viele mögliche Lösungen dafür geben.

Problem der Wissensmaximierung (Configuration-Maximization-Problem):

Gegeben seien ein KBM, eine minimale initiale Konfiguration für das KBM_{min}, eine maximale initiale Konfiguration für das KBM_{max} und eine Menge von Sicherheitsanforderungen und Liveness-Anforderungen. Es stellt sich das Problem: Was sind die Möglichkeiten zur Maximierung der initialen Konfiguration zwischen min und max unter der Voraussetzung, dass Sicherheits- und Liveness-Anforderungen erfüllt bleiben müssen?

Die Wissensmaximierung sucht nach Lösungen, die die initiale Konfiguration maximieren. Es kann auch dabei viele Lösungen geben.

Generalisiertes Maximierungsproblem (Generalized-Maximization-Problem): Gegeben seien ein KBM, eine minimale initiale Konfiguration für das $\text{KBM}_{\min}$, eine maximale initiale Konfiguration für das $\text{KBM}_{\max}$, eine Menge von Subjekten, deren Verhalten maximiert werden soll und eine Menge von Sicherheitsanforderungen und Liveness-Anforderungen. Es stellt sich das Problem: Was sind die Möglichkeiten zur Maximierung der initialen Konfiguration zwischen min und max und des Verhaltens der Subjekte unter der Voraussetzung, dass Sicherheits- und Liveness-Anforderungen erfüllt bleiben müssen?

Ein generalisiertes Maximierungsproblem kombiniert die Maximierung des Verhaltens mit der Maximierung des Wissens und es kann mehrere Lösungen besitzen.

14.2.4 *Scoll: Eine Sprache, um Sicherheitsprobleme auszudrücken*

Scoll ist eine Sprache, in der man abstrakte Patterns interagierender Subjekte beschreiben und dabei die Sicherheitsprobleme beschreiben kann, die man angesichts dieser Patterns lösen möchte. Scoll ist eine erweiterbare Sprache, basierend auf einer Kernsprache, deren logische Semantik durch KBMs ausgedrückt wird. Wir werden die Sprache anhand eines Beispiels vorstellen, nämlich dem Confused Deputy Pattern von oben.

Beispiel: Ein einfaches Confused Deputy Pattern

Abbildung 14.16 zeigt das Problem, das wir in Scoll ausdrücken wollen. Nur das Verhalten des Deputies soll vertrauenswürdig sein, die anderen Knoten sind stark

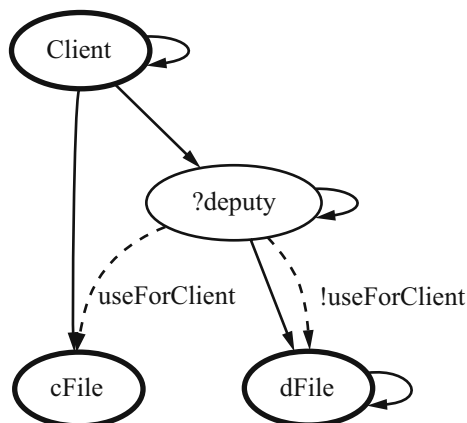


Abb. 14.16 Das Confused Deputy Pattern

umrandet. Durchgezogene Pfeile stellen die einfache Zugriffsberechtigung dar – der einzige Typ von Permission in Objekt-Capability-Systemen, wie sie in Kap. 10 bereits vorgestellt wurden. Zugriff bedeutet das Recht, das Objekt, auf das der Pfeil zeigt, zum Senden von Nachrichten bzw. als Argument in Nachrichten zu verwenden. Der Client hat Zugriff auf den Deputy und zur Datei des Clients „cFile“. Der Deputy hat Zugriff auf die Deputy-Datei „dFile“. Um sicher zu sein, dass unsere Approximation konservativ ist, geben wir jedem Subjekt Zugriff auf sich selbst.

Beachten Sie, dass dem Namen des Deputies in der Abbildung ein Fragezeichen vorangestellt ist. Scoll verwendet dieses Fragezeichen, um anzuzeigen, dass wir das maximal erlaubte Verhalten des Deputies finden wollen. Der gestrichelte Pfeil vom dFile des Deputies repräsentiert eine Sicherheitsanforderung: Die Lösungen, die wir für das maximale Verhalten des Deputies finden, sollen alle vermeiden, dass er sein eigenes File für Zwecke des Clients verwendet. Scoll verwendet Ausrufezeichen um Sicherheitsanforderungen anzuzeigen („!“ muss als „nicht“ interpretiert werden).

Der gestrichelte Pfeil vom Deputy zu cFile repräsentiert eine Liveness-Anforderung: Die Einschränkungen, die wir für die Lösungen des maximalen Verhaltens des Deputies finden, müssen immer noch erlauben, dass der Deputy das cFile für Zwecke des Clients benutzen kann.

Abbildung 14.17 zeigt ein zum Confused Deputy Pattern korrespondierendes Scoll-Programm. Prädikatsbezeichner und Subjekt-Konstanten beginnen mit Kleinbuchstaben. Variablen rangieren über allen Subjekten und beginnen mit Groß-

```
declare
  state: access/2
  behavior: may.sendTo/3 may.getFrom/2 may.return/2 may.accept/1
  knowledge: did.sendTo/3 did.getFrom/3 did.return/2 did.accept/2

system
  access(A,B) access(A,X) A:may.sendTo(B,X) B:may.accept()
  => access(B,X) A:did.sendTo(B,X) B:did.accept(X);
  access(A,B) access(B,X) A:may.getFrom(B) B:may.return(X)
  => access(A,X) A:did.getFrom(B,X) B:did.return(X);

behavior
  UNKNOWN f => may.sendTo( , ) may.getFrom( ) may.return( ) may.accept();g DEPUTY
  fdid.accept(F) => useForClient(F);g subject
  client: UNKNOWN
  cFile: UNKNOWN
  ?deputy: DEPUTY
  dFile: UNKNOWN

config
  access(client,client) access(cFile,cFile)
  access(deputy,deputy) access(dFile,dFile)
  access(client,cFile) access(client,deputy)
  access(deputy,dFile)

goal
  useForClient(deputy,cFile)
  !useForClient(deputy,dFile)
```

Abb. 14.17 Das Deputy Pattern in Scoll

buchstaben. Ein Unterstrich kann verwendet werden, um Variablen zu bezeichnen, deren Namen keine Bedeutung haben. Bezeichner von Prädikaten können das Punktzeichen enthalten, um die Lesbarkeit zu fördern. Verhaltenstypen werden nur mit Großbuchstaben geschrieben.

Scoll-Programme (auch Scoll-Patterns genannt) bestehen aus den folgenden sechs Teilen, deren Schlüsselwörter fett geschrieben sind:

Declare

Der erste Teil deklariert Bezeichner und Stelligkeit der Prädikate über die Subjekte im Programm. Scoll kennt drei Arten von Prädikaten:

- **State**-Prädikate modellieren den Sicherheits-State. In diesem Fall ist die Zugriffserlaubnis das einzige State-Prädikat.
- **Behavior**-Prädikate modellieren die Intentionen, die ein Subjekt haben kann:
 - `A:may.sendTo(B,X)` bedeutet, dass A eine Message an B schicken kann mit X als Argument.
 - `A:may.getFrom(B)` bedeutet dass A eine Message an B schicken kann und Zugriff auf die Rückgabeargumente akzeptiert.
 - `B:may.return(X)` bedeutet, dass B beim Empfang einer Message X zurückgeben kann.
 - `B:may.accept()` bedeutet, dass B Zugriff auf das Argument in einer erhaltenen Message akzeptieren kann.
- **Knowledge**-Prädikate modellieren den inneren Zustand eines Subjekts: Sie beschreiben, was ein Subjekt „wissen“ oder „lernen“ kann über das System und andere Subjekte. In unserem Beispiel stammt das Wissen aus erfolgreichen Interaktionen mit anderen Subjekten.
 - `A:did.sendTo(B,X)` bedeutet, dass A weiß, dass es an B eine Message mit X als Argument geschickt hat.
 - `A:did.getFrom(B,X)` bedeutet, dass A weiß, dass es an B eine Message geschickt hat und dass A das Rückgabeargument X von B akzeptiert hat.
 - `B:did.return(X)` bedeutet, dass B weiß, dass es X zurückgegeben hat.
 - `B:did.accept(X)` bedeutet, dass B weiß, dass es Zugriff auf ein Argument X akzeptiert hat.

Beachten Sie, dass das Scoll-Programm in Abb. 14.17 ein zusätzliches Prädikat `userForClient()` benutzt, das in diesem Teil nicht deklariert ist. Es handelt sich dabei um ein Prädikat von lokalem Wissen, das nur für ein Subjekt relevant ist (hier der Deputy) und das automatisch in Scoll bei der ersten Benutzung im Verhalten des Deputies deklariert wird. Wir benutzen es, um die Absicht des Deputies zu modellieren, eine Datei für die Zwecke des Clients zu benutzen.

System

Dieser Teil beinhaltet die Systemregeln: Das sind DataLog-Regeln, die konservativ alle Mechanismen für Subjekte beschreiben, um zu (inter-)agieren, und alle möglichen Effekte dieser Aktionen auf den Security-State und auf das Wissen der

Subjekte modellieren. Die linke Seite einer Regel (vor dem $\Rightarrow$ Zeichen) listet alle Bedingungen auf, die erfüllt sein müssen, damit die Regel “feuert”. Die rechte Seite listet die Effekte. Scoll-Regeln verwenden nur Variablen in den Prädikaten. Da Capability-Systeme Zugriffsrechte für alle Interaktionen benötigen, haben alle Regeln Prädikate für Zugriffsrechte in ihrem Bedingungsteil.

Die erste Systemregel in Abb. 14.17 drückt einen ersten Mechanismus zur Interaktion in Objekt-Capability-Systemen aus. Wenn A Zugriff auf B und X hat und A ist willens, X an B zu senden und B ist willens, Messages anzunehmen (mit mindestens einem Argument), dann wird B Zugriff auf X erhalten, A wird wissen, dass es B Zugriff auf X verschafft hat und B wird wissen, dass es Zugriff auf X durch die Annahme als Argument einer Message bekam.

Die zweite Systemregel in Abb. 14.17 drückt einen zweiten Mechanismus zur Interaktion in Objekt-Capability-Systemen aus: Wenn A Zugriff auf B und auf X hat, und A ist willens eine Message an B zu schicken und von dieser Nachricht das Rückgabeargument zu akzeptieren, und B ist darüber hinaus willens, X zurückzugeben, wenn es eine Message geschickt bekommt, dann wird A Zugriff auf X erhalten und A wird wissen, dass es X als Returnwert von B erhalten hat, und B wird wissen, dass es X an einen unbekannten Sender zurückgegeben hat.

Behavior

Verhaltensregeln sind ähnlich den Systemregeln, drücken jedoch aus, welches Wissen ein Subjekt benötigt, bevor es an einer Interaktion teilnehmen kann. Ein Subjekt kann daher sein ganzes Verhalten auf die Subjekte ausrichten, über die es ein gewisses Maß an Wissen erhalten hat. Wir modellieren die Einschränkungen im Verhalten einer Entität als notwendige Bedingungen in den Verhaltensregeln des Subjektes.

Es ist wichtig dabei zu beachten, dass das erste Argument eines jeden Prädikates in einer Verhaltensregel immer weggelassen wird (der Teil vor der Prädikatsbezeichnung). Das liegt darin begründet, dass sich das Argument implizit immer auf das Subjekt bezieht, dessen Verhalten beschrieben werden soll. Verhaltensregeln definieren einen Typus von Verhalten, ähnlich wie eine Klasse in objektorientierten Sprachen. Das obige Beispiel zeigt zwei Verhaltenstypen: UNKNOWN und DEPUTY.

Der UNKNOWN-Typ wird für die konservative Modellierung des Verhaltens von Entitäten verwendet, denen wir nicht trauen. Wir wollen nicht von ihren Verhaltenseinschränkungen abhängig sein, um die Ableitung der Sicherheitsanforderungen zu vermeiden und müssen daher annehmen, dass sie jede Chance zur Weitergabe von Zugriffsrechten nutzen. Das ist ausgedrückt durch eine einzige Verhaltensregel mit einer leeren linken Seite, die ohne weitere Bedingungen feuert:

```
=> may.sendTo(_,_) may.getFrom(_) may.return(_) may.accept();
```

Auf diese Weise modellieren wir nicht-vertrauenswürdige und unbekannte Entitäten konservativ, d. h. als Subjekte, die immer versuchen, jede mögliche Aktion durchzuführen. Zu beachten ist die Verwendung anonymer Variablen, angezeigt durch den Unterstrich.

Der DEPUTY-Typ ist gedacht zur Modellierung des minimalen Verhaltens des Deputies. Erinnern Sie sich, dass der Deputy das Subjekt ist, dessen maximal erlaubtes Verhalten wir berechnen wollen. Die Regel `did.accept(F) => useForClient(F)`; drückt aus, dass der Deputy gewillt ist, jedes File, das ihm geschickt wird, für die Zwecke des Clients zu verwenden, z. B. um die Ergebnisse der Kalkulation dort zu speichern.

Subject

In diesem Teil listen wir die Subjekte und ihr Verhalten, und wir fügen bei den Subjekten, deren maximal erlaubtes Verhalten wir wissen wollen, ein Fragezeichen als Prefix an. Subjekte, die auf diese Weise gekennzeichnet sind, werden als „query subjects“ bezeichnet. In unserem Beispiel ist der Deputy das einzige query subject.

Config

In diesem Teil sind alle Fakten (Prädikate mit Subjekten anstatt Variablen) aufgelistet, die am Anfang der Konfiguration wahr sind. In unserem Fall zählen wir dabei alle Zugriffsrechte auf, die im Graphen der Abb. 14.16 enthalten sind. Fakten der Konfiguration können dabei ebenso ein vorgestelltes Fragezeichen besitzen. Genau wie Verhaltensfakten eines query subjects werden diese optionalen Konfigurationsfakten dann maximiert. Unser Beispiel hat jedoch keine optionalen Konfigurationsfakten.

Goal

Dieser Teil enthält die Sicherheits- und Liveness-Anforderungen. `deputy: useForClient(cFile)` ist eine Liveness-Anforderung, die anzeigt, dass das maximale Verhalten, das wir für den Deputy berechnen, ihn nicht davon abhalten darf, `cFile` für die Zweck seiner Clients zu verwenden, nämlich die Ergebnisse dort abzuspeichern.

`!deputy:useForClient(dFile)` ist eine Sicherheitsanforderung, die anzeigt, dass das maximal berechnete Verhalten des Deputies ihm nicht erlauben darf, `dFile` für diesen Zweck zu verwenden.

14.2.5 Die Analyse des Confused Deputy mit Scollar

Scollar ist ein Werkzeug, um die Sicherheit in Mustern kollaborierender Subjekte zu analysieren. Es nimmt ein Scoll-Programm als Eingabe und berechnet die folgenden Outputs:

- **Fixpoints:**

Alle State-, Verhaltens- und Wissensfakten, die im Pattern erreichbar sind, werden berechnet. Der „minimale Fixpoint“ berechnet, welche Fakten erreichbar sind ohne die Zugabe von optionalen Fakten. Der „maximale Fixpoint“ berechnet, welche Fakten erreichbar sind, wenn alle optionalen Fakten ebenfalls hinzugefügt werden. Das Resultat für beide Fixpoints wird in einer einzigen

Tabelle zusammengefasst. Wie bereits vorher angemerkt, werden optionale Fakten in Scoll durch Fragezeichen gekennzeichnet. Im Falle des Fehlens von Fragezeichen sind beide Fixpoints identisch. Da ein Scoll-Programm eine endliche Menge von Subjekten besitzt, sind die Fixpoint-Berechnungen einfach und schnell. Durch die monotone Logik sind die im minimalen Fixpoint erreichbaren Fakten immer eine Untermenge der Fakten im maximalen Fixpoint. Wenn es unmöglich ist, den Sicherheitsanforderungen zu genügen, wird dies im minimalen Fixpoint angezeigt: Es wird dann ein Fakt als erreichbar angezeigt, der als Sicherheitsanforderung definiert ist. Wenn das Erreichen der Liveness-Anforderungen nicht möglich ist, zeigt sich das im maximalen Fixpoint: Dort wird es einen Fakt geben, der nicht erreichbar ist, obwohl er ein Liveness-Requirement darstellt. Wenn sowohl Sicherheits- als auch Liveness-Anforderungen im maximalen Fixpoint zutreffen, dann können alle optionalen Fakten gefahrlos erlaubt werden.

Was passiert im Fall, dass die Sicherheitsanforderungen im minimalen, aber nicht im maximalen Fixpoint erreicht werden? Und was, wenn die Liveness-Anforderungen umgekehrt nur im maximalen Fixpoint erreicht werden? In diesen Fällen kann Scollar angewiesen werden, maximale (irgendwelche oder alle) Kombinationen optionaler Fakten zu finden, die dazu führen, dass alle Liveness-Fakten, aber keine Sicherheits-Fakten erreichbar werden.

- **Lösungen:**

Scollar kann alle Weisen berechnen, in denen die Interaktionen zwischen Subjekten eingeschränkt werden müssen, damit:

- keine Sicherheitsanforderungen verletzt werden,
- keine Liveness-Anforderungen verhindert werden,
- jede Lösung eine Menge optional erlaubbarer Fakten darstellt, die maximal im Sinne der Sicherheit sind: Die Entfernung eines optionalen Faktums von der Lösung ist nicht nötig für die Sicherheit, aber das Hinzufügen eines Faktums bricht wenigstens eine Sicherheitsanforderung. Lösungen werden auch als minimale Mengen von Einschränkungen betrachtet: Das Hinzufügen einer Einschränkung ist nicht nötig für Sicherheit und das Entfernen einer Einschränkung bricht mindestens ein Sicherheitsziel.

Durch das Setzen optionaler Fakten im Scoll-Pattern entscheidet der Benutzer, welche Art von Einschränkungen aufgestellt werden können. Typischerweise werden die optionalen Fakten das Verhalten der Subjekte betreffen, auf die sich der Benutzer verlassen können möchte und/oder Teil des initialen Protection States sind, den der Benutzer kontrollieren kann. Die Benutzung von Scollar auf diese Weise ist am Besten, wenn es um das Design bzw. die Programmierung sicherer Muster geht, die Interaktionen zwischen vertrauenswürdigen (trusted, relied-upon) und nicht-vertrauenswürdigen (unknown, untrusted) Entitäten geht.

Wir haben Scollar verwendet, um nach allen maximalen Mengen von Verhaltensprädikaten für den Deputy zu suchen. Das Ziel war, zu entdecken, welche Einschränkungen nötig und hinreichend sind im Verhalten des Deputies, um zu

garantieren, dass er nicht seine eigenen Dateien falsch (als wären es Dateien des Clients) verwendet, und gleichzeitig ihn nicht daran hindern, die Client-Dateien für deren Zwecke zu benutzen.

Aus den Resultaten der Analyse können wir lernen, welches Verhalten wir vermeiden sollten bei der Implementation einer Deputy-Entität, die nicht verwirrt werden soll. Wenn wir die Details der Lösung betrachten, finden wir außerdem wichtige Hinweise darüber, wie gut wir das Problem selbst in Scoll beschrieben haben.

Verhaltensbeschränkungen für den Deputy

Scollar präsentiert die Lösungen zu dem Interaktionspattern als Zugriffsgraph (Abb. 14.18) und eine Liste der Lösungen für das Verhalten des Deputies in Tabelle 14.1, die in diesem Fall eine einzige Lösung zeigt.

Der Graph stellt den Zugriff zwischen Subjekten in allen Lösungen dar. Im Fall, dass mehrere Lösungen existieren, wird der Zugriff, der in wenigstens einer, aber nicht allen Lösungen erreichbar ist, als gestrichelter Pfeil dargestellt. Neuere Versionen von Scoll erlauben auch die Visualisierung anderer Relationen auf ähnliche Weise. Aus dem Graph des erreichbaren Zugriffs können wir unmittelbar erkennen, dass niemand direkten Zugriff auf `dFile` erhält (außer dem Deputy), obwohl dies nicht als Sicherheitsanforderung spezifiziert wurde. Die anderen drei Subjekte können gefahrlos einander zugänglich (introduced) gemacht werden.

Pro Lösung gibt es in der Tabelle eine Spalte. Im Beispiel gibt es nur eine Lösungsspalte. Die optionalen Fakten, die in der Lösung nicht erlaubt sind, werden mit einer 0 in den Feldern der Spalte angezeigt. In unserem Beispiel gibt es nur Verhaltensbeschränkungen für den Deputy. Lassen Sie sie uns eine nach der anderen betrachten:

deputy:may.return(dFile)

Der Deputy sollte nicht seine eigene Datei als Teil seines Dienstes an seine Clients zurückgeben. Eine erwartbare Einschränkung.

deputy:may.sendTo(client,dFile)

Natürlich sollte der Deputy nicht einfach seine Datei an einen Client schicken. Ansonsten könnte der Client diese Datei im Anschluss an den Deputy zum Überschreiben geben. Wie bekommt der Deputy übrigens Zugriff auf den Client? Der Client hat Zugriff auf sich selbst und könnte sich selbst an den Deputy übergeben haben, statt einer Datei. Das ist völlig legal und realistisch, da der Client selbst das Dateiinterface implementieren und das Überschreiben durch den Deputy behandeln könnte.

deputy:may.sendTo(cFile,dFile)

Der Deputy sollte auch nicht seine eigene Datei an die Datei des Clients schicken. Diese Einschränkung ist eine nützliche Warnung, betreffend die Datei des Clients: Da wir nicht abhängig sein wollen vom Verhalten dieser Datei, hat Scollar richtigerweise angenommen, dass es sich dabei auch um ein feindliches Subjekt handeln könnte, clever versteckt hinter einem Dateiinterface.

Abb. 14.18 Der aus der Analyse resultierende Zugriffsgraph

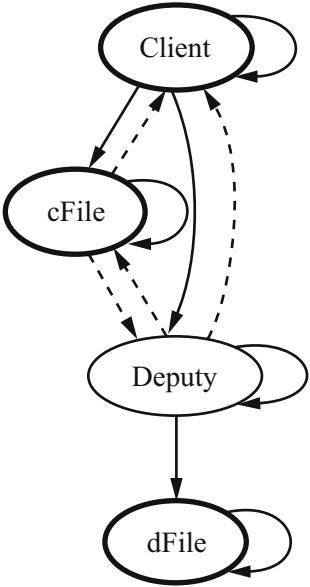


Tabelle 14.1 Verhaltenseinschränkungen für den Deputy (0 nicht erlaubt)

Deputy	Client	cFile	Deputy	dFile
Knowledge				
access(deputy,. . .)	1	1	1	1
deputy:did.receive(. . .)	1	1	1	–
deputy:did.return(. . .)	–	–	–	–
deputy:did.getFrom(client, . . .)	1	1	–	–
deputy:did.getFrom(cFile, . . .)	1	1	–	–
deputy:did.getFrom(deputy, . . .)	1	1	–	–
deputy:did.getFrom(dFile, . . .)	–	–	–	–
deputy:did.sendTo(client, . . .)	0	–	–	0
deputy:did.sendTo(cFile, . . .)	1	–	1	0
deputy:did.sendTo(deputy, . . .)	1	1	1	0
1 0				
deputy:did.sendTo(dFile, . . .)	0	0	0	–
deputy:useForClient(. . .)	1	1	1	0
Behavior:				
deputy:may.receive()	1	1	1	1
deputy:may.getFrom()	1	1	1	1
deputy:may.return()	1	1	1	0
deputy:may.sendTo(client,)	0	–	–	0
deputy:may.sendTo(cFile,)	1	–	1	0
deputy:may.sendTo(deputy,)	1	1	1	0
deputy:may.sendTo(dFile,)	0	0	0	–

deputy:may.sendTo(deputy,dFile)

Diese Einschränkung braucht etwas Erklärung. Auf den ersten Blick macht sie keinen Sinn: Warum sollte dem Deputy nicht erlaubt sein, seine eigene Datei an sich selber zu schicken? Der Grund ist, dass es ihn selber verwirren könnte: Als antwortender Dienst wird er annehmen, dass ein Client die Datei gesendet hat und er wird die Datei zum Schreiben des Resultates verwenden statt für sich selber. Wenn unser Deputy wirklich die eigene Datei an sich selber schicken muss, dann sollte er den Unterschied zwischen seiner eigenen Datei und der eines Clients erkennen können. Sein Verhalten in unserem Pattern müsste angepasst werden, um die Art und Weise des Unterschieds zu reflektieren. Im gegenwärtigen Detaillierungsgrad jedoch warnt Scollar vor der Gefahr, dass sich der Deputy selbst verwirrt.

deputy:may.sendTo(dFile,client)

Dies besagt, dass der Deputy nicht den Client an dFile schicken sollte. Das macht Sinn, falls dFile nicht eine einfache Datei ist – eine durchaus realistische Annahme. Statt einer einfachen Datei kann der Deputy dafür selbst einen zweiten Deputy verwenden. Die Einschränkung bringt die Gefahr ans Licht, dass dieser Deputy zweiten Grades nicht eingeschränkt genug ist. Wenn zum Beispiel ein derartiges dFile Deputy Zugriff auf den Client bekommt, dann könnte dFile selbst an den Client gelangen, der es selber überschreibt oder den ersten Deputy dazu benutzt, es zu überschreiben.

deputy:may.sendTo(dFile,cFile)

Aus den gleichen Gründen sollte der Deputy nicht die Datei des Clients an dFile schicken: dFile könnte dadurch dem Client bekannt werden.

deputy:may.sendTo(dFile,deputy)

Der Deputy sollte sich nicht selbst an dFile senden, denn dFile könnte dann selbst Client spielen und den Deputy dazu veranlassen, sich selber zu überschreiben. Wenn dies eine beabsichtigte Attacke von dFile wäre, dann wäre die Verhaltenseinschränkung des Deputy nutzlos, denn dFile könnte genauso gut ihn selber überschreiben. Aber falls „unbekanntes Verhalten“ wirklich meint, was es sagt und nicht notwendigerweise „feindseliges Verhalten“ bedeutet, dann macht die Einschränkung durchaus Sinn.

Weitere Folgerungen aus der Analyse

Wenn wir auf den Button in der ersten Reihe der Lösungsspalte klicken, werden die Fakten zum Verhalten und Wissen aller Subjekte des Patterns in einer eigenen Tabelle gezeigt. Tabelle 14.2 ist ein Ausschnitt aus dieser Sicht und zeigt die Wissensfakten des Deputies (oberer Teil) und seine Verhaltensfakten (unterer Teil) in Matrixform. Die Spalten in der Matrix zeigen das Subjekt im letzten Argument des Faktums.

Die Zielpredikate sind Teil des Wissens des Deputies. Die Sicherheitsanforderung ist angezeigt durch **0**, die Liveness-Anforderung mit **1**. Sie zeigen, dass die Lösung sowohl sicher als auch funktional ist.

Tabelle 14.2 Wissen und Verhalten des Deputies (1)

Deputy	Client	cFile	Deputy	dFile
Knowledge				
access(deputy, . . .)	1	1	1	1
deputy:did.receive(. . .)	1	1	1	–
deputy:did.return(. . .)	–	–	–	–
deputy:did.getFrom(client, . . .)	1	1	–	–
deputy:did.getFrom(cFile, . . .)	1	1	–	–
deputy:did.getFrom(deputy, . . .)	1	1	–	–
deputy:did.getFrom(dFile, . . .)	–	–	–	–
deputy:did.sendTo(client, . . .)	0	–	–	0
deputy:did.sendTo(cFile, . . .)	1	–	1	0
deputy:did.sendTo(deputy, . . .)	1	1	1	0
0				
deputy:did.sendTo(dFile, . . .)	0	0	0	–
deputy:useForClient(. . .)	1	1	1	0
Behavior:				
deputy:may.receive()	1	1	1	1
deputy:may.getFrom()	1	1	1	1
deputy:may.return()	1	1	1	0
deputy:may.sendTo(client,)	–	–	–	0
deputy:may.sendTo(cFile,)	1	–	1	0
deputy:may.sendTo(deputy,)	1	1	1	0
deputy:may.sendTo(dFile,)	0	0	0	–

Es ist oft nützlich, sich diese Details anzusehen, um interessante Dinge über das Pattern selbst zu lernen und wie es in Scoll ausgedrückt wird. Sehen wir uns zum Beispiel die deputy-internen Wissenfakten zu useForClient an. Wir haben bereits zwei von ihnen diskutiert: deputy:useForClient(cFile) ist, was wir erreichen wollten (liveness) und deputy:useForClient(dFile) ist, was wir als Verwirrung betrachten und vermeiden wollen.

Ein drittes Faktum, deputy:useForClient(client), richtet unsere Aufmerksamkeit erneut auf die Möglichkeit, dass der Client sich selbst als Datei präsentiert und dass daran auch nichts falsch ist. Der interessierte Leser kann versuchen, ob dieses Verhalten trotzdem verhindert werden kann, indem dieses Prädikat als Sicherheitsanforderung zum Zielteil (goal) des Patterns hinzugefügt wird.

Was ist mit dem letzten Fakt dieses Prädikats: deputy:useForClient (deputy)? Es besagt wörtlich, dass der Deputy sich selbst behandeln will, als ob er eine Datei wäre und für Zwecke des Clients benutzt werden sollte. Das war aber nicht unsere Absicht, oder? Natürlich kann der Client immer den Deputy selbst als Argument an den Deputy übergeben – der Client hat ja und braucht auch den Zugriff auf den Deputy. Das bedeutet, der Deputy kann immer „verführt“ werden, sich selbst als Client-Datei zu behandeln. Der einfachste Weg dies zu vermeiden ist sicherzustellen, dass Schreibkommandos an den Deputy keinen Effekt haben,

z. B. indem der Deputy kein File Interface anbietet oder indem die Schreibmethode nichts tut. Der interessierte Leser ist hier aufgefordert, dieses verfeinerte Verhalten in Scoll selbst zu modellieren.

Verfeinerte und Alternative Scoll Patterns

Das gezeigte Pattern kann noch auf viele verschiedene Weisen analysiert werden, jede mit ihren eigenen Vorteilen und Anwendungsbereichen:

- Die Verhaltensprädikate können verfeinert werden, z. B. um uns zu erlauben, das Verhalten des Deputies als eine Datei zu spezifizieren.
- Die Datei des Deputies könnte als „trusted“ angenommen werden im Sinne eines normalen File Interfaces oder ohne jedes Verhalten.
- Die Datei des Deputies könnte ein query subject werden, um weitere Lösungen zu finden, die dann von den Einschränkungen des Verhaltens dieser Datei abhängen.

14.3 Prüfung Operationaler Umgebungen

Während bei der Sicherheitsanalyse einer Software durch Code-Inspektion der Fokus hauptsächlich auf der Einhaltung von Confinement-Regeln – also der maximalen Isolation der internen Module und Komponenten voneinander – liegt, beschäftigt sich die Analyse operationaler Umgebungen stärker mit der Prüfung von legalen und notwendigen Verbindungen zwischen Infrastrukturteilen. Beispiele für solche Prüfungen sind:

- die Verbindungen durch einen Firewall-Komplex, bestehend aus Paketfiltern, Reverse Proxies, Bastion Hosts, Applications-Servern etc.;
- die Sicherheit einer Zelle, bestehend aus Applikations-Servern, Web-Servern etc. in Bezug auf die Konfiguration der Mechanismen zur Authentisierung, Autorisierung und Delegation von Aufrufen;
- die Input/Output Validierung einer Web-Applikation;
- die Korrektheit eines Benutzerberechtigungssystems, das die fachlichen Rollen und deren Rechte verwaltet;
- der Nachweis, dass trotz Delegation von Requests keine nicht-authentisierten Requests von Komponenten, die nur authentifizierte Requests erwarten, akzeptiert werden.

Die erste Antwort auf das Testproblem bei solchen Umgebungen besteht meist in der Einführung von Scripts bzw. von Testprogrammen. So wurde der Websphere-Application-Server nachträglich um eine Scripting Engine erweitert, die das Aufsetzen von Konfigurationen für ganze Zellen erleichtert. Die Sicherheit von Java-EE-Infrastrukturen kann mit speziellen Testprogrammen geprüft werden, die den jeweiligen Principal beim Ausführen von Code in Web-Containern, EJB-Containern etc. prüfen und festhalten. Die Ergebnisse werden dann mit den ent-

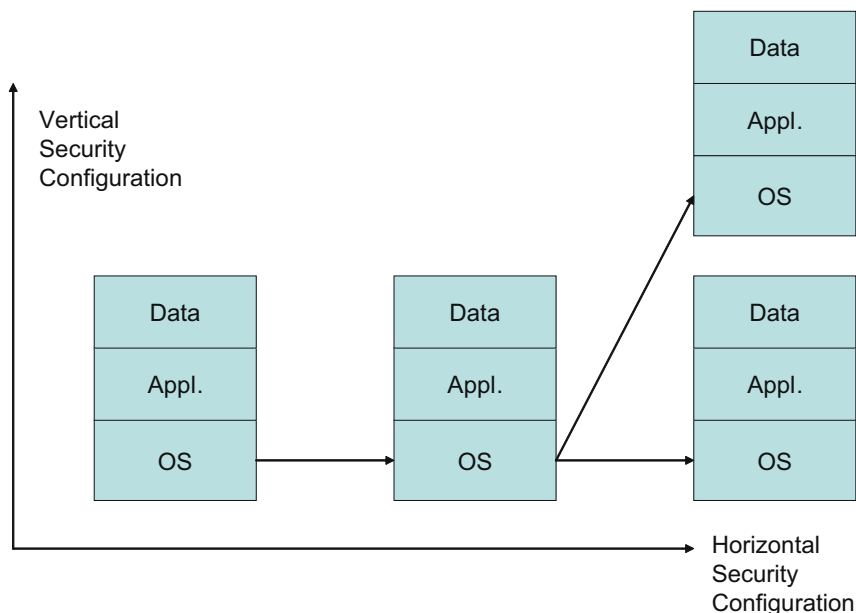


Abb. 14.19 Vertikale vs. Horizontale Sicherheitskonfiguration

sprechenden Rollendefinitionen verglichen. Zu beachten ist, dass hier Daten verschiedener Maschinen zusammengefasst werden müssen, da Requests typischerweise eine ganze Reihe von Applikations-Servern durchlaufen.

Das Problem dieser Verfahren liegt letztlich darin, dass die Vorgaben der Prüfung – die Gültigkeitskriterien – lediglich in Dokumenten enthalten sind und dann direkt in Code umgesetzt werden. Änderungen der Umgebungen ziehen somit größere Änderungen der Scripts bzw. Testprogramme nach sich. Ein formales Modell des zu prüfenden Systems sowie der Bedingungen der Gültigkeit von Sicherheit ist nicht vorhanden.

Eine weitere Schwierigkeit liegt darin begründet, dass die Sicherheitskonfigurationen einer Maschine nicht isoliert für sich betrachtet werden können, sondern nur eingebunden in ein Gesamtkonzept, das den Fluss von Requests durch die Infrastruktur definiert. Dadurch ergibt sich sowohl eine vertikale (durch die unterschiedlichen Software-Layer auf einer Maschine) als auch eine horizontale Dimension (durch den Datenfluss durch die Infrastruktur), wie in Abb. 14.19 dargestellt.

Die Korrektheit der Konfiguration einer Maschine kann daher nur im Verbund beurteilt werden, denn auch ein verneinter Zugriff kann ein Liveness-Problem in der Infrastruktur auslösen.

Die vertikale Sicherheitsdimension auf einer Maschine impliziert auch unterschiedliche Sicherheits-Domains pro Maschine, die voneinander teils abhängig sind, deren Abhängigkeiten jedoch auf der jeweils unteren Ebene nicht modelliert sind bzw. nicht erkannt werden (Abb. 14.20).

Vertical Security Configuration

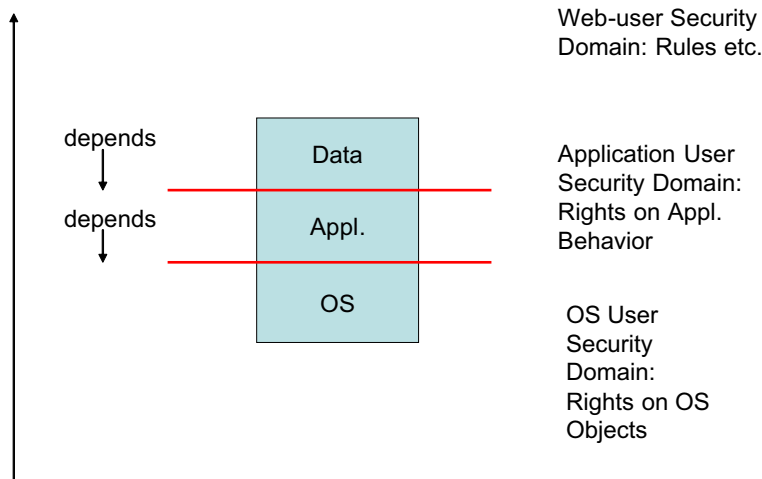


Abb. 14.20 Vertikale Sicherheitskonfiguration

Klassische Beispiele für diese Problematik bilden etwa die Modellierung von Nutzern einmal als Daten in einer Datenbank, dann als Benutzer der Datenbank (von ihr selbst verwaltet) und letztlich als Betriebssystemnutzer. Je nach Perspektive wird dieser Nutzer nur als Datum oder als Principal angesehen. Darunter fällt auch das generelle Problem, dass bestimmte Systemweite Entitäten wie E-Mail-Adressen für das Betriebssystem lediglich Daten in einem Applikations-File sind und keine Objekte, die das Betriebssystem selbst sichern und verwalten könnte.

Als Konsequenz dieser vertikalen Sicherheitsschichten kann eine Änderung auf untergeordneten Ebenen (z. B. an den Dateirechten im Betriebssystem) die Sicherheit einer ganzen Applikation aushebeln. Besser wäre ein Top-Down-Ansatz, bei dem von oben nach unten die korrekte Konfiguration durchgesetzt würde.

Der Versuch, operationale Umgebungen wie Firewall-Zonen oder komplexe Web-Applikationen auf ihre Sicherheit zu testen, ist aufwändig und teuer. Der Penetration-Test einer Applikation kann ein Team von Spezialisten zwei Wochen beschäftigen. Das Hauptproblem liegt im erforderlichen hohen manuellen Anteil beim Testen, der im Allgemeinen ca. 75% beträgt. Als Konsequenz können solche Tests nur selten durchgeführt werden. Dadurch entstehen lange Lücken im Lebenszyklus einer Applikation, während der die durchgeführten Änderungen nicht auf ihre Sicherheit getestet werden. Schnell sorgt die Erfüllung eines Businesswunsches dann für das Entstehen einer Sicherheitslücke, etwa durch Cross-Site-Scripting, z. B. durch Umgehung der Standard-Validierung des Inputs.

Im Folgenden wollen wir daher einige Ansätze zeigen, wie das Problem des Testens operationaler Umgebungen gelöst werden könnte. Dabei werden wir auch versuchen, die oben geschilderten Ideen zur Reduktion von Autorität auf die Infrastruktur zu übertragen. Die meisten Ansätze zur automatischen Prüfung von Sicherheit sind jedoch noch in einem frühen Stadium. So verkraften Testwerkzeuge oft nur kleine Mengen von Prüfobjekten.

14.3.1 Modellierung und Generierung

Der Sicherheitsaspekt eines Systems (hier verstanden als Infrastrukturen wie Firewall-Zonen sowie den dazugehörigen Applikationen und Servern bis hin zu den Backend-Systemen) kann als ein Teilaspekt des operationalen Aspektes aufgefasst werden. Die operationale Welt umfasst also die korrekte Installation von Knoten (Rechnern), Zonen (z. B. VLANs) und Applikationen sowie Servern und der dazugehörigen Netzwerke, Überwachungslogiken, Verzeichnisdiensten etc.

Im Java EE wie im .NET-Umfeld tauchen hier große Probleme bei der Wartbarkeit und Korrektheit von Konfigurationen auf. Aber auch die Planung zukünftiger Releases von Applikationen oder Middleware ist abhängig von der korrekten Erfassung der jeweiligen Auslieferungsstände und deren Abhängigkeiten. Die informelle Beschreibung dieser Entitäten und Abhängigkeiten erlaubt keine Validierung der Installationen. Stattdessen werden die Installationen durch komplexe Scripts durchgeführt, deren Korrektheit unklar ist.

Im großen Infrastrukturprojekt PAI des Daimler-Chrysler-Konzerns wurde versucht, eine Modellierung der operationalen Umgebung durchzuführen und daraus dann die korrekten Installationen zu generieren. Der Schwerpunkt lag dabei auf der „Liveness“ der Installation, d. h. ihrer korrekten Funktionsfähigkeit, nicht aber in der Ausmodellierung von Sicherheitsaspekten. Dennoch könnte man sich leicht vorstellen, wie der Ansatz mit weiteren Sicherheitsaspekten angereichert werden könnte.

Im PAI-Projekt wurde auf der Basis der Meta-Object-Facility (MOF) der OMG ein eigenes Metamodell für die Modellierung operationaler Aspekte entwickelt [Bleyh]. Die Entscheidung gegen ein UML-Profil wurde aufgrund mangelnder Unterstützung operationaler Aspekte in UML generell getroffen – das Deployment-Diagramm ist zu wenig ausdrucksstark, um die komplexen Abhängigkeiten und Entitäten zu repräsentieren. Ob das verstärkte Aufkommen von SYSml als Modellierungsprofil für Systeme und deren Architekturen daran etwas ändert, bleibt abzuwarten. Momentan scheint SYSml eher auf die Modellierung der Systemarchitektur abzielen statt auf operationale Aspekte.

Das Diagramm in Abb. 14.21 zeigt einen Ausschnitt des physikalischen Modells der operationalen Umgebung. Neben der physikalischen Sichtweise existieren auch ein Conceptual Level und ein Specification Level, die alle zusammen auf gemeinsamen Metamodell-Elementen beruhen.

Über Constraints wird der Zusammenhang zwischen den Sichtweisen gesichert. Hier ein Beispiel für die Notation von Constraints bezüglich der Betriebssysteme:

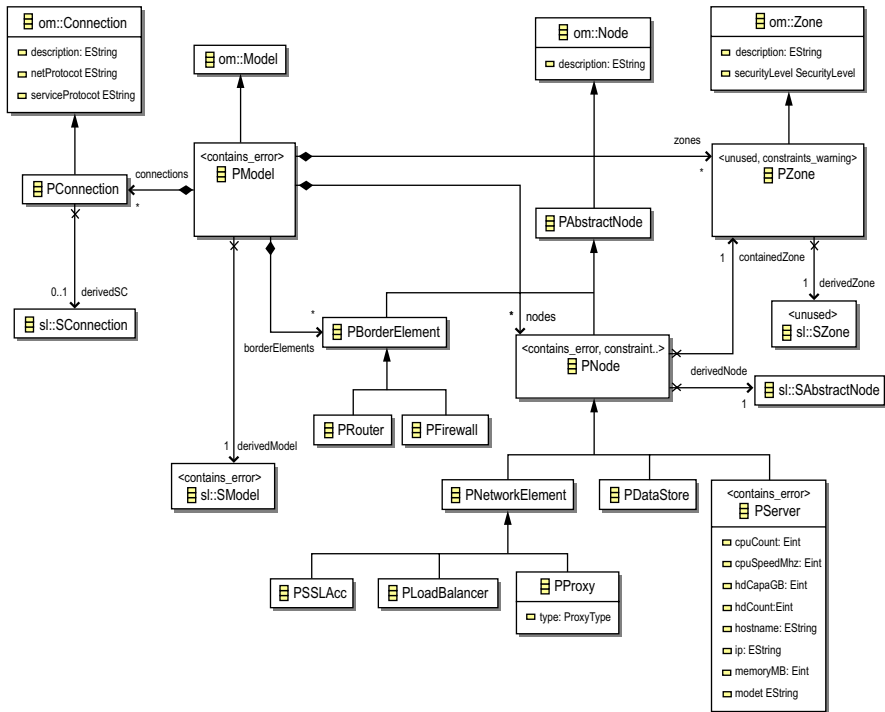


Abb. 14.21 Teil des Physical Models der Operationalen Umgebung [Bleyleh]

context sl:: SNode

inv:

```

self.deploymentUnits -> select(du | not du.supportedOSs
-> exists(os | os.ID = self.operatingSystem.ID) )
-> union(
-> self.derivedDUs -> select(du | not du.supportedOSs
-> exists(os | os.ID = self.operatingSystem.ID) )
)

```

Bleyleh schildert die Entwicklung einer domänenspezifischen Sprache (DSL) auf Basis der MOF und verwendet dazu allgemein verfügbare Werkzeuge aus dem Umfeld von Eclipse, wie z. B. das Eclipse Modelling Framework (EMF).

Für die konkrete Modellierung sowie die dabei verwendeten Techniken und Werkzeuge sei auf [Bleyleh] verwiesen, da eine genauere Behandlung den Rahmen dieses Buches völlig sprengen würde. Festzuhalten bleibt jedoch, dass damit ein Beginn der Modellierung von Sicherheitsaspekten in komplexen Infrastrukturen gemacht wurde.

Die Modellierung und anschließende Generierung von Konfigurationen (wie in objektorientierter Software ist auch im operationalen Bereich die Korrektheit der

Anfangskonfigurationen von entscheidender Bedeutung) ist jedoch nicht die einzige Möglichkeit, ein Modell von Sicherheitsaspekten zu verwenden.

14.3.2 Model-Checking von Sicherheitsprotokollen

An der Technischen Universität München ist in Zusammenarbeit mit BMW SECUml entstanden – ein UML-Profil zur Modellierung von Security [Jür]. Anders als der Ansatz von Bleyh zielt SECUml weniger auf die Generierung von sicherem Code für bestimmte Plattformen, sondern auf eine modellimmanente Prüfung der Sicherheit von z. B. Protokollen wie SSL.

Dieser Ansatz der Modellprüfung ist aktuell zweifellos der aufwändigste in Bezug auf die Modellierung und die Darstellung von Constraints. In eine ähnliche Richtung geht der Ansatz „executable UML“. Beiden ist gemeinsam, dass sie relativ große Mengen Code in Form von Pseudo-Code in die Modelle einbeziehen müssen und auch der Logik-Anteil in Form von Constraints beträchtlich ist.

Eine spezielle Form des Modell-Checkings von Software entwickelt sich im Umfeld der Programmiersprache OZ/Mozart, die von Peter van Roy und Saif Haridi [VH] entwickelt wurde. Die Sprache besitzt starke Logik-Elemente und ist geeignet für Concurrent Constraint Programming. Auf dieser Basis werden Modelle von Software-Agenten (Proxies z. B.) um interne und externe Constraints und Eigenschaften angereichert. Darauf wird dann versucht, eine Erreichbarkeitsanalyse durchzuführen. Die Modelle beziehen auch zukünftige Kollaborationen zwischen den Agenten mit ein, die zu Änderungen der Sicherheitseigenschaften führen können.

Die permanente, dynamische Änderung der Sicherheitskonfiguration ist jedoch im operationalen Umfeld (mit Ausnahme von Benutzerberechtigungssystemen) eher selten zu finden. Hier geht es darum, die Korrektheit einer Konfiguration nachzuweisen. Konkret stellt sich z. B. in einem Firewall-Komplex die Frage, ob bestimmte Systeme erreichbar sind und über welche Pfade. Hier bietet sich ebenfalls die constraint-basierte „Graph-Reachability“ Analyse von van Roy und Spiessens [VS] an. Es existieren bereits erste Produkte, die solche Techniken im Bereich von Infrastrukturen einsetzen [Str].

14.3.3 Modell-getriebenes Security-Engineering

HEALTH@NET – die Verteilte Elektronische Gesundheitsakte – verwendet einen modell-getriebenen Generierungsansatz für die Sicherheit der Infrastruktur sowie der Services und Applikationen [HMBAUS].

Die fachliche Anforderung bestand im Aufbau eines verteilten Repositories aus Gesundheitsdaten von Patienten unter Berücksichtigung der vollständigen Kontrolle der Patienten über ihre Daten. Dazu wurde ein zweistufiges Sicherheitskon-

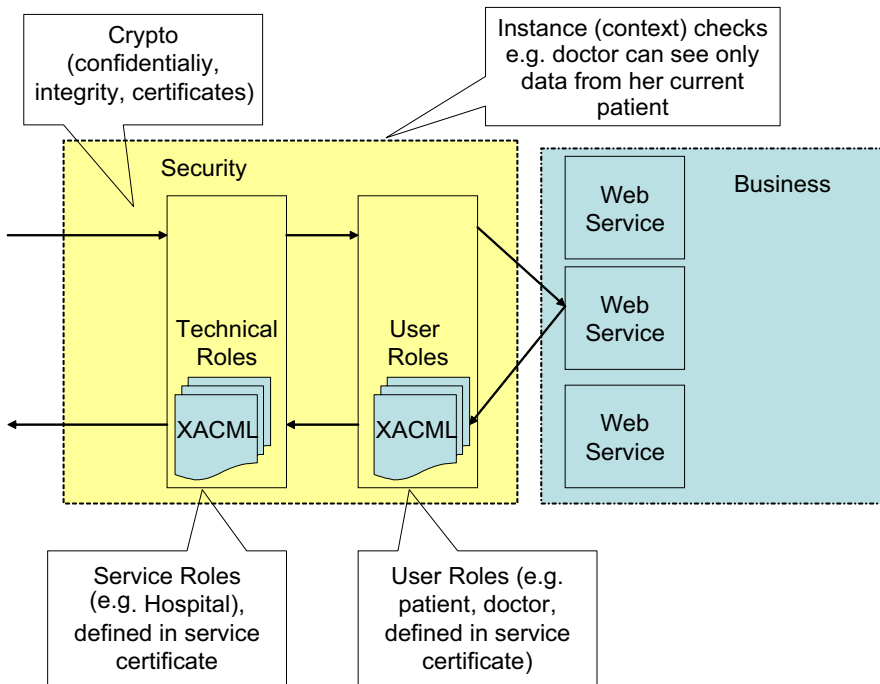


Abb. 14.22 Sicherheitsarchitektur von HEALTH@NET

zept unter Trennung von servicebezogenen und benutzerbezogenen Prüfungen entwickelt (Abb. 14.22). In der ersten Stufe werden die Rollen von Service-Stellen geprüft (Hospitäler, Apotheken etc.) Eingesetzt wird dabei – wie auch in der zweiten Stufe – ein hierarchisches Role-Based-Access-Control (RBAC) Konzept.

In der zweiten Stufe wird dann der konkrete Benutzer, ebenfalls anhand seines Zertifikates, geprüft und nur im Falle der Genehmigung wird der Request an den jeweiligen Web-Service weitergeleitet. Kontextabhängige, instanzbasierte Sicherheit kann hier ebenfalls eingesetzt werden, damit z. B. ein Patient Zugriffe zeitlich limitieren kann, oder damit ein Apotheker nur die Rezepte des momentanen Kunden einsehen kann.

Realisiert wird die Zugriffskontrolle durch Policy-Files auf Basis von XACML (Oasis). Diese Files werden aus dem Modell der Anwendung generiert.

Modelliert werden die Sicherheitsanforderungen als Teil von UML2.0-Modellen. In dieser fachlichen Modellierung gibt es auch nicht die in Abb. 14.22 gezeigte Trennung in einen Sicherheits- und einen Businessenteil. Die Sicherheitsanforderungen sind dort Teil der fachlichen Modellierung [HMBAUS, S. 143]. Konkrete Sicherheitsanforderungen sind beispielsweise Vertraulichkeit, Integrität, Non-Repudiation, aber auch komplexe Forderungen wie der Einsatz des Vier-Augen-Prinzips, Widerrufbarkeit von Erlaubnissen etc. Diese werden z. B. in

Activity-Diagrammen oder Klassen-Diagrammen festgehalten, und aus ihnen werden dann die entsprechenden Policy-Files für die Infrastruktur generiert.

Der in HEALTH@NET verwendete Ansatz ist typisch für model-driven Development: Artefakte, die die Infrastruktur benötigt, werden aus den Annotationen des Modells automatisch erzeugt. Somit kann sichergestellt werden, dass zentrale Anforderungen des Sicherheitskonzeptes sich auch in der Infrastruktur wiederfinden. Zusätzlich könnte man natürlich in der Zukunft auch Testkonzepte der Infrastruktur aus der Modellinformation generieren.

14.3.4 Lernen und Testen

Testen bedeutet, ein Verhalten gegen eine Spezifikation zu testen. Häufig existiert aber keine bzw. keine maschinenlesbare Spezifikation eines Verhaltens, und das Testen wird zur Glückssache.

Einen interessanten Ansatz verfolgen hier lernfähige Web-Application-Firewalls (WAF, vgl. [Str]), die innerhalb einer Testumgebung trainiert werden. Sie lernen quasi die korrekten Parameter von Applikationen und können bei Abweichungen einen Alarm auslösen. Dies ähnelt stark dem Vorgehen bei verhaltensgetriebenen Host-Intrusion-Detection-Systemen (zu deren Eigenschaften und Limitationen siehe [Schm]). Auch hier ist es günstig, die Fähigkeiten und Grenzen solcher Werkzeuge zu kennen und eventuell das Design der eigenen Applikationen bereits daran anzupassen. Dieses Vorgehen kommt natürlich all den Applikationen entgegen, die über kein ausformuliertes Modell ihrer Kommunikation mit der Außenwelt verfügen.

In jedem Fall besser ist jedoch das Testen anhand klarer Spezifikationen. Hier zeichnet sich ein Trend in Richtung besserer Modellierung von Tests und ein Weggehen vom script-basierten Testen ab. Ein Beispiel dafür ist das auf UML-Activity-Diagrammen basierende Framework *expecco* der Firma *eXept* ag. Dieses Framework erlaubt auch „Nicht-Entwicklern“ die Komposition komplexer Testfälle auf der Basis von Activity-Diagrammen. Stubs in verschiedenen Programmiersprachen erlauben den Anschluss an beliebige Systeme unter Test.

Bisher wurden die meisten Testwerkzeuge eher für funktionale Tests eingesetzt, sodass für den Test von Sicherheitsaspekten noch weitere Erfahrungen nötig sind. Ganz konkret stellt sich z. B. die Frage, wie das Testen von serviceorientierten Architekturen (SOA) aussehen soll. Durch deren hohe Entkopplungsgrade wird ein durchgängiges Testen der Services sehr schwer.

14.3.5 Beweisbare Plattformsicherheit

Ein sehr interessantes, und wenn erfolgreich auch bahnbrechendes, Projekt im Kernel-Bereich von Betriebssystemen läuft seit 2004 beim Australischen Institut

für Informations- und Kommunikationstechnologie (NICTA), siehe [Heis08]. Es geht dabei um nichts weniger als die Abschaffung der Trusted Computing Base, genauer des „Trusted“-Teils davon. Bisher war es ein erklärtes Ziel von sicheren Systemen, den Teil des Systems, der die höchsten Privilegien besitzt, so klein wie möglich zu halten. Der Grund lag darin, dass diesem Teil völlig vertraut werden muss. Im Umfeld des bereits erwähnten seL4 Projektes wurde jedoch der Versuch gestartet, den ca. 7000–10000 LOC großen Microkernel bzw. dessen API formal durch einen Theorem Prover beweisen zu lassen. Dazu muss das API als mathematisches Modell vorliegen.

Leider würde eine ausführliche Darstellung den Rahmen dieses Buches sprengen, deshalb wollen wir nur einige wesentliche Merkmale des Vorgehens dabei vorstellen. Verwendet wird ein Refinement-Prozess, der aus drei Ebenen besteht. Die oberste Ebene stellt die formale Spezifikation des Systems dar, die niedrigste der C- und Assemblercode der Implementation. Dazwischen findet sich eine Systembeschreibung in Form einer funktionalen Sprache (haskell), die als executable specification dient [Heis, S. 2f.]. Dies ist eine schöne Bestätigung der Bedeutung funktionaler Prinzipien, wie sie auch im Kap. 10 in diesem Buch vertreten werden.

Weitere interessante Projekte im Umfeld von seL4 sind die temporale Analyse von Kernel-Verhalten (z. B. Cache-Management) sowie eine flexible aber sichere Komponentenarchitektur auf der Basis explizit definierter Connectors [HEKKP].

Input-Validierung

Wir haben bereits angemerkt, dass Applikationsentwickler im Bereich der Input-Validierung bislang kaum Unterstützung durch Frameworks erhalten. Viele Firmen entscheiden sich daher zurzeit für die Einführung vorgelagerter Web-Applikation-Firewalls (WAF), die umfangreiche Filterfunktionen anbieten und teilweise, wie oben bereits erwähnt, sogar lernfähig sind, d. h. im Applikationstest trainiert werden auf den zu erwartenden Input der Applikation. Diese Vorverlagerung von Prüfungen wirft mehrere Fragen auf:

- Welche Instanz prüft welchen Input, und wird dabei eventuell der Inputstrom modifiziert?
- Besteht die Möglichkeit, dass Input mehrfach geprüft wird? Welche Konsequenzen ergeben sich daraus (Stichwort Doppel-Dekodierung)?
- Mit welchen Mitteln wird geprüft und welche Sicherheit lässt sich damit erreichen?
- Welche Rolle spielt der Kontext bei der Validierung eines Inputstrings? Anders gefragt: Lassen sich auch Teilstücke des Inputs validieren bzw. unschädlich machen?

Wir sind nicht die ersten, die bemerken, dass das Problem der Input-Validierung bisher kaum theoretisch angegangen wurde. Hansen und Patterson haben auf der Blackhat Konferenz 2005 einen sehr interessanten formalen Ansatz in

Form einiger Theoreme, bei denen auf die Terminologie der theoretischen Informatik zurück gegriffen wird, vorgestellt [HP]:

- Die Validierung eines Inputs muss durch einen Mechanismus der gleichen Stärke erreicht werden wie zur Erzeugung des Inputs nötig war.
- Für die Validierung eines Inputs sollte kein Mechanismus verwendet werden, der stärker ist als der Mechanismus, mit dem der Input erzeugt wurde.
- Jeder Input I muss innerhalb seines syntaktischen Kontexts validiert werden. Der syntaktische Kontext ist der Teil des Gesamtstrings S , der an die Applikation übergeben wird, der vom Input I in seiner syntaktischen Struktur beeinflusst werden kann.
- Es ist möglich zu einer unsicheren Sprache L eine sichere Teilsprache (d. h. eine Teilmenge) L' zu erzeugen, die nur sichere Strings enthält. Die Teilsprache wird erzeugt durch Einschränkung der in L verwendeten Terminalsymbole, Variablen und Ableitungsregeln in der erzeugenden Grammatik.

Diese theoretischen Aussagen zur Stärke bzw. zum Umfang der Validierung haben große praktische Konsequenzen. So lässt sich aus dem ersten Punkt mit Mitteln der theoretischen Informatik folgern, dass der beliebte Einsatz von regulären Ausdrücken zur Input-Validierung zwangsläufig nicht allen schädlichen Inhalt erkennen kann, da der dazugehörige Input durch einen mächtigeren Mechanismus (kontextfreie Grammatiken) erzeugt wurde. Diese Aussage hat einige Brisanz, weil viele in Firewalls eingesetzte Filter (z. B. auch das bereits beschriebene `mod_security`-Paket zur Input-Validierung bei Web-Services) auf regulären Ausdrücken beruhen.

Das von Hansen und Patterson zur Illustration gewählte Beispiel sind die bekannten Angriffe vom Typ „SQL Injection“. SQL ist wie die meisten Sprachen kontextfrei entworfen, d. h. mit einem Keller-Automaten leicht zu parsen. Die bekannten Attacken gegen Web-Applikationen beruhen nun darauf, einen Input so zu konstruieren, dass durch das Parsen ein syntaktischer Unterschied zum eigentlich gedachten Konstrukt entsteht:

Der SQL-Ausdruck

```
select rights from table where username = I1 and password = I2
```

wird durch die Eingaben

```
I1 = "root" sowie I2 = "` or '1' = '1"
```

in folgende Form gebracht:

```
select rights from table where username = root and password
='` or '1'='1'
```

Gedacht war natürlich, dass I_1 und I_2 beide Terminal-Strings sind, also nicht weiter geparkt werden sollten. Durch das Parsen wird der Boole'sche Ausdruck immer als TRUE ausgewertet und dem Angreifer werden Root-Rechte gewährt.

Das Beispiel zeigt die wichtige Rolle des Kontexts bei der Validierung des Inputs: Der Input allein ergibt kein gültiges SQL, eingebettet in den gesamten Kontext jedoch entsteht ein gültiger SQL-Satz, dessen syntaktische Struktur von der eigentlich beabsichtigten abweicht. Es ist jedoch unklar, wie weit sich diese Beeinflussung ausdehnt. Die Autoren nehmen daher einen Ausweg und fordern, dass in Fällen, wo keine Schranke für den syntaktischen Einfluss eines Input-Strings gefunden werden kann, der Input gegenüber der gesamten Sprache L geparkt werden muss. Halten wir daher vorerst die folgenden Punkte fest:

- Die Validierung eines Inputs ist vom zeitlichen Kontext abhängig (wie oft wird geparkt?).
- Die Validierung eines Inputs ist vom räumlichen Kontext abhängig (in welche Umgebungen wird der Input eingebettet?)

Was bringt die Einbindung in den gesamten Kontext der Sprache L ? Durch die Attacke ist ja ein syntaktisch gültiger Satz der Sprache L entstanden, der jedoch aus Security-Sicht ungültig ist. Wir führen deshalb eine sichere Sub-Sprache L_S ein, die nur noch „sichere“ Strings enthalten soll. Hier bedeutet das, dass an der Stelle von I_1 und I_2 lediglich Terminalstrings erlaubt sind. Ein entsprechender Filter (Kellerautomat) kann nun feststellen, dass eine Verletzung dieser Regel vorliegt und den Input zurückweisen.

Was lässt sich aus dieser theoretischen Erkenntnis ableiten? Die Gefährlichkeit eines Stücks Input ist keine Eigenschaft des Inputs selbst, sondern absolut kontextabhängig: Damit entscheiden die jeweiligen Interpreter, die verwendet werden, und die Einbettung in andere Kontexte über die Gefährlichkeit auch in der Zukunft. Eine Applikation sollte versuchen, eine minimale Sprache L_S zu definieren, die Terminalsymbole als Input besitzt. Am besten sind natürlich Token geeignet, die direkte Terminals darstellen und möglichst keine Sonderzeichen enthalten sollten. Nur müssen diese Terminals auch als solche von der Applikation behandelt werden, und es darf auf keinen Fall eine spätere erneute Validierung durch Parsen stattfinden.

Jetzt erkennen wir, dass die sog. Prepared Statements, die häufig als Mittel gegen SQL-Injection angeführt werden, genau diese Funktion erfüllen: Sie behandeln eine Variable als Terminal. Beunruhigend ist jedoch, dass der Verarbeitungskontext verloren gehen kann, da der Input in der Datenbank landet. Wer weiß schon, wie er später verarbeitet wird? Es ist dieser zeitliche Aspekt der Kontext-Abhängigkeit von Validierung, der am meisten Sorgen bereitet: Ein schönes Beispiel hierfür bildet das Durchtunneln von Angriffs-Code mittels doppelter Dekodierung, z. B. durch Verwendung von %% in einer URL [TI].

Der Trick beruht darauf, dass durch doppelte Dekodierung von Escape-Zeichen erst im zweiten Dekodierungs-Schritt der gefährliche Code entsteht, der Filter im ersten Schritt ihn jedoch nicht bemerkt. Microsofts Internet-Information-Server ist

mehrfach Opfer dieses Angriffs geworden. Leider kann der Fehler nicht nur innerhalb einer Applikation auftreten, sondern natürlich auch über verschiedene Teile einer Infrastruktur hinweg: Reverse Proxy, Web-Server, Applikations-Server, Applikation. Wer filtert bzw. modifiziert hier was? Einige einfache Fragen in dieser Richtung lösen oft betroffenes Schweigen bei den Beteiligten aus. Was kann nun eine Applikation tun? Die Grundlage für Validierung wird im Design der sicheren Sprache L_S gelegt. Der Input muss möglichst auf Terminalsymbole reduziert werden und diese müssen auch so behandelt werden. Die Grammatik der sicheren Sprache L_S muss für einen Parser zur Verfügung stehen. Der Aufwand für das Parsen steigt mit den Freiheitsgraden der Benutzer. Hier kann durch geschicktes Design der Nutzer-Interaktion viel erreicht werden.

Der hier beschriebene, zunächst überraschende zeitliche Aspekt von Input-Validierung kommt auch sehr gut in den Attacken auf Host-Intrusion-Detection-Systemen (HIDS) zum Ausdruck: Durch gezieltes Escaping kann dort Angriffs-Code am HIDS vorbeigeschleust werden. Kleinste Unterschiede in der Validierung zwischen Infrastrukturkomponenten lassen sich dazu bereits ausnutzen (siehe dazu auch hier wieder die Arbeit von M. Schmidt [Schm]).

Literatur

- [Bleyh] M. Bleyh, Modellierung operationaler Aspekte von Systemarchitekturen, Diplomarbeit an der Hochschule der Medien 2006, Präsentation downloadbar unter <http://www.kriha.de/krihaorg/dload/bleyh.pdf>
- [Bis] M. Bishop, Computer Security – Art and Science, MacMillan Technical Publishing 2002
- [Boeb] W.E. Boebert, On the Inability of an Unmodified Capability Machine to Enforce the *-Property, 7th DOD/NBS Computer Security Conference, 1984, <http://zesty.ca/capmyths/boebert.html>
- [Gall] H. Gallaire, J. Minker (eds), Logic and Data Bases. Perseus Publishing, 1978, 0.1.3
- [Hard] N. Hardy, The confused deputy: (or why capabilities might have been invented), SIGOPS Oper. Syst. Rev. 22(4), (1988), 36–38
- [Heis08] G. Heiser, The Role of Virtualization in Embedded Systems, 1st Workshop on Isolation and Integration in Embedded Systems, Glasgow, UK, April, 2008, http://www.ertos.nicta.com.au/publications/papers/Heiser_08.pdf
- [HEKKP] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, S.M. Petters, Towards Trustworthy Computing Systems: Taking Microkernels to the Next Level. ACM Operating Systems Review, 41(4), (July, 2007), 3–11, http://ertos.nicta.com.au/publications/papers/Heiser_EKKP_07.pdf
- [HMBAUS] M. Hafner, R. Mair, R. Brey, B. Agreiter, S. Unterthiner, T. Schabetsberger, HEALTH@NET – Die Verteilte Elektronische Gesundheitsakte. Eine Fallstudie in Modell-Getriebenem Security Engineering. In: Innovationsmotor IT-Sicherheit, Tagungsband zum 10. Deutschen IT-Sicherheitskongress, BSI Bonn, 2007
- [HP] R.J. Hansen, M.L. Patterson: Guns and Butter: Towards Formal Axioms of Input Validation, http://www.blackhat.com/presentations/bh-usa-05/BH_US_05-Hansen-Patterson/HP2005.pdf
- [HRU] M. Harrison, W. Ruzzo, J. Ullman, Protection in Operating Systems, Communications of the ACM 19 (8), (1976), 461–471

- [Jür] J. Jürjens, *Secure Systems Development with UML*, Springer 2005
- [Lamp] B.W.Lampson: A note on the confinement problem. *Commun. ACM* 16(10) (1973) 613–615
- [MiSh] M.S. Miller, J. Shapiro, Paradigm Regained: Abstraction Mechanisms for Access Control. In: 8th Asian Computing Science Conference (ASIAN03), (December 2003) 224–242
- [MiSt] M.S. Miller, M. Stiegler, T. Close, B. Frantz, K.P. Yee, C. Morningstar, J. Shapiro, N. Hardy, Tribble, E.D., Barnes, D., Bornstien, D., Wilcox-O’Hearn, B., Stanley, T., Reid, K., Bacon, D.: E: Open source distributed capabilities (2001) Available at <http://www.erights.org>.
- [SaSch] J.H. Saltzer, M.D. Schroeder, The Protection of Information in Computer Systems, 1973, online verfügbar unter <http://www.cs.virginia.edu/~evans/cs551/saltzer/>
- [Schm] M. Schmidt, Host Intrusion Detection Systeme, Diplomarbeit an der Hochschule der Medien, 2007, Kapitel 4: Ausnutzen mehrfachen Encodings
- [Shap] J. Shapiro, The Practical Application of a Decidable Access Model. SRL Technical Report SRL 2000-01, Johns Hopkins University, <http://srl.cs.jhu.edu/pubs/SRL2003-04.pdf>
- [SMVS] F. Spiessens, M. Miller, P. Van Roy, J. Shapiro, Authority Reduction in Protection Systems. <http://www.info.ucl.ac.be/~fsp/ARS.pdf>, 2004
- [Spie] F. Spiessens, Patterns of Safe Collaboration. PhD thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium (February 2007)
- [Str] S. Strobel, Applikationssicherheit – mehr als Härten und Patchen von Systemen, <http://www.cirosec.de/deutsch/presse/veroeffentlichungen/Applikationssicherheit.html>
- [TI] <http://www.technicalinfo.net/papers/CSS.html>
- [TL] M. V. Tripunitara, N. Li, The Foundational Work of Harrison-Ruzzo-Ullmann Revisited, Dept. of Computer Science, Purdue University
- [VS] P. van Roy, F. Spiessens, A Practical Formal Model for Safety Analysis in Capability Based Systems, In: TGC 2005, Volume 3705 of Lecture Notes in Computer Science, Springer-Verlag 2005, 248–278
- [VH] P. van Roy, S. Haridi, *Concepts, Techniques and Models of Computer Programming*, 2004, MIT Press
- [Wall] D.S.Wallach, D. Balfanz, D.Dean, E.W. Felten.: Extensible security architectures for java. In: Proceedings of the sixteenth ACM symposium on Operating systems principles, ACM Press (1997) 116–128

Kapitel 15

Schlussbetrachtungen

Nach dieser Tour-de-Force durch die unterschiedlichen Techniken und Methoden sicherer Software und Systeme wollen wir zum Abschluss nun einige, in unseren Augen auch gesellschaftlich besonders kritische, Aspekte und Trends aus den bisher behandelten herausgreifen und einen Blick in die Zukunft wagen.

15.1 Security vs. Safety

Wir haben dieses Buch mit der Frage begonnen, warum überhaupt unsichere Software existiert. Jetzt am Ende wollen wir noch einmal darauf zurückkommen und das Problem mehr von einer ökonomischen Perspektive aus betrachten. Bruce Schneier hat sich kürzlich dazu geäußert und bringt die beunruhigende Wahrheit auf den Punkt:

“The primary reason the IT security industry exists is because IT products and services aren’t naturally secure. If computers were already secure against viruses, there wouldn’t be any need for antivirus products. If bad network traffic couldn’t be used to attack computers, no one would bother buying a firewall. If there were no more buffer overflows, no one would have to buy products to protect against their effects. If the IT products we purchased were secure out of the box, we wouldn’t have to spend billions every year making them secure.” [Schn07]

Die Sicherheit unserer Systeme und IT-Lösungen ist also bereits bei Fertigstellung schlecht. Security-Produkte wie Virens Scanner und Firewalls bekämpfen Symptome und nicht die Ursachen schlechter Software. Schneier nennt auch das Grundproblem hinter unsicheren Systemen: Es ist den Herstellern gelungen, das Risiko der Systeme auf die Endkunden (Firmen oder Privatleute) abzuwälzen. Für diejenigen, die effektiv etwas zur Sicherheit der Systeme beitragen könnten, ist es damit ökonomisch unsinnig, diese Kosten aufzubringen: Der Endkunde kann sich nicht von unsicherer Software unterscheiden und wird sich im Zweifelsfall für die kostengünstigere und damit potenziell unsicherere Lösung entscheiden.

Warum aber ist den Softwareherstellern – anders als etwa der Automobilindustrie – das Abschieben des Risikos auf den Endkunden gelungen? Ein Grund dafür mag sein, dass die Softwareindustrie die beiden Teilaspekte von „Sicherheit“ – im Englischen als „Safety“ und „Security“ bezeichnet – in der Wahrnehmung ihrer Produkte trennt und das eine (Safety) durch das andere (Security) zu verdecken sucht. „Sichere“ Software impliziert aber zwei Dinge: Sie muss sicher sein im Sinne von „Security“, d. h. sie bietet Schutz gegen gezielte, bösartige Handlungen. Anders gesagt: Sie muss im Sinne von Geschäftsvorgängen oder rechtlichen Vorgängen sicher sein. Darunter fällt z. B. die Pflicht, die Verwendung der Software auf eine entsprechende Autorisierung der Personen hin zu prüfen. Bereits hier zeigen viele Entwickler Schwächen in der Wahrnehmung der Gefahren, aber auch in der Kenntnis der entsprechenden Frameworks und Protokolle, mit denen solche Gefahren eingedämmt werden können. Dieses Buch versucht, sowohl die Wahrnehmung von Gefahren als auch die nötigen Kenntnisse in diesem klassischen Feld der IT-Security zu vermitteln.

Aber sichere Software umfasst eben auch den Aspekt der „Safety“ – dem Verhalten von Geräten und Software bei ordnungsgemäßer Benutzung bzw. zufälligen Bedienfehlern. Wir erwarten, dass ein Gerät unseren Erwartungen entsprechend dient und dabei versucht, Schäden zu vermeiden oder zu minimieren. Nehmen wir als Beispiel das Programm Solitär – es wird mittlerweile zum kanonischen Beispiel problematischer Architektur – und betrachten sein Verhalten. Wenn es einem Angreifer gelingt, in dieses Programm einen Virus einzuschleusen, z. B. durch eine Buffer-Overflow-Attacke, dann besteht die Möglichkeit, dass alle Daten und Dienste, auf die der aktuelle Nutzer Zugriff hat, missbraucht oder gelöscht werden. Was aber, wenn das Programm Solitär nicht mit einem Virus infiziert ist, sondern einen Programmierfehler enthält? Auch in diesem Fall können Daten und Dienste beschädigt werden, und zwar aus dem gleichen Grund: Dem Fehlen von geeigneten Mechanismen zur Schadensbegrenzung. Also liegt nicht in erster Linie ein Security-Problem vor, sondern ein Safety-Problem, das sich eben auch im bösartigen Sinn ausnutzen lässt. Die Wurzeln dieses Problems liegen auf allen Ebenen der Software: Im Betriebssystem, in der verwendeten Sprache und im Applikationsdesign. Wir haben es also in erster Linie mit einem Problem der Softwarequalität zu tun und erst in zweiter Linie mit einem Security-Problem. Gelingt es uns, die Qualität der Software im Allgemeinen anzuheben, werden wir auch „sicherere“ Software erhalten.

15.2 Security vs. Usability

Ein Auto spricht zu seinem Fahrer mithilfe verständlicher Abstraktionen, nämlich den Anzeigeinstrumenten. Warnungen gibt es wenige und sie sind meist unmittelbar verständlich. Anders bei kommerzieller Software: Benutzer und auch Entwickler werden mit Warnungen bombardiert, die weitgehend unverständlich sind. Seiteneffekte sind gang und gäbe und stellen sowohl für die Zuverlässigkeit als auch

die Sicherheit von Software ein großes Problem dar. Usability wird daher in Zukunft ein großes Thema beim Bau sicherer Systeme sein. Aber sie ist natürlich von der darunter liegenden Technik abhängig. Eine Technik, die keine Einschränkung von Gefahren kennt (außer der, die Verantwortung auf die User abzuwälzen), wird eine Orgie von Warnungen produzieren. Eine Technik, die Safety automatisch erzeugt, braucht nicht viele Warnungen, oder wie oft sind Sie schon von Ihrem ABS-System im Auto gewarnt worden? Der Warnungsdialog „Wollen Sie wirklich das File x überschreiben?“ ist nur dann nötig, wenn ihr System keine automatische Versionierung kennt.

Im Zuge der Verbesserung der Usability wird auch die Psychologie einen starken Einzug in die IT-Security halten: Die Firmen werden erkennen müssen, dass das Angebot von kollaborativen Diensten noch mehr Verantwortung für die Sicherheit der Benutzer mit sich bringt als bisher. Das Verhalten der Benutzer untereinander muss einkalkuliert werden und vor schwarzen Schafen und gefährlichem Verhalten muss gewarnt werden. Dazu gehört auch die Erstellung unterschiedlicher konzeptueller Modelle der Nutzer, das bedeutet im Klartext: Eigenschaften der Nutzer wie Alter, Geschlecht, Beruf oder kultureller Hintergrund, die bislang keine Rolle in der Softwareentwicklung spielten, müssen in Zukunft klar erkannt und berücksichtigt werden. Zu den Nutzern gehören auch die Entwickler als Benutzer von Frameworks und Entwicklungswerkzeugen: Auch sie leiden massiv unter nur schwierig benutzbaren Sicherheitsmechanismen. Hinzu kommt ein subtiler Druck durch die Schwerfälligkeit existierender Infrastrukturen: Änderungen an Applikationen sind sehr aufwändig geworden. Mehrstufige Testzyklen und Deployment-Mechanismen machen kurzfristige Anpassungen fast unmöglich. Die Folge ist, dass Entwickler häufig dazu neigen, von Beginn an mit maximalen Zugriffsrechten zu arbeiten: Ein Datenbankschema, das lt. Businessspezifikation eigentlich nur Lesezugriffe erlauben müsste, wird gleich auch für Updates eingerichtet – man weiß ja nie, welche Anforderungen noch kommen. Die Einschränkung von Autorität in der Software wird also zusätzlich durch unflexible Anpassungsmöglichkeiten behindert.

Das bringt uns zum nächsten Punkt: Die Bedeutung der Semantik beim Bau sicherer Systeme wird dramatisch zunehmen. Objektbasierte Sicherheit in Form von signierten Vollmachten, Tickets, Erlaubnissen etc. bietet sehr viel mehr Flexibilität, als durch die bisherige, kanalbasierte Sicherheit auf der Ebene der Infrastruktur möglich war – allerdings unter der Voraussetzung, dass die Kommunikationspartner ihre Sicherheitsanforderungen in interoperabler Form angeben können. Aber die Semantik wird auch im Sinne der angebotenen Abstraktionen eine größere Rolle spielen: Dinge, die für Benutzer eines Computers wichtig sind, wie z. B. Adressen von Kommunikationspartnern, werden vom System auf einer entsprechenden hohen Ebene der Abstraktion repräsentiert werden – und nicht nur als String.

Das Einbeziehen von Benutzern, Entwicklern und Anwendern in Sicherheitslösungen kennzeichnet auch das endgültige Ende von eindimensionalen Bedrohungsmodellen. Zukünftige Sicherheitslösungen müssen „End-to-End“ arbeiten und praktisch sämtliche Bedrohungsmodelle einbeziehen.

15.3 Mehr Sicherheit durch Einschränkung der Nutzer?

Zum oben erwähnten Safety-Aspekt gehört auch die Zuverlässigkeit. Sie wird in Zukunft als Sicherheitsproblem verstärkt wahrgenommen werden, und zwar in dem Maße, wie wir von unseren Systemen abhängig werden. Diese Abhängigkeit von unserer computerisierten Umwelt fordert mehr Berücksichtigung von Sicherheitsaspekten und stellt dadurch gleichzeitig eine Gefahr dar: Sicherheit wird oft auch durch Einschränkung von Rechten der Nutzer und Funktionalitäten der erworbenen Systeme erreicht. Diese „Entmachtung“ kann aber schnell auch zu einer Entmündigung führen. Gute Beispiele hierfür sind wieder die Automobilindustrie, aber auch die Einführung von modernen GUIs [Step]: Wer kann heute noch etwas an seinem Fahrzeug reparieren? Oder anders gefragt: Wer will das noch? Hier hat ein technischer Entmündigungsprozess stattgefunden, der tatsächlich mehr Sicherheit gebracht hat und der wohl von der Mehrzahl der Nutzer positiv gesehen wird. In der Frage von Trusted-Plattform-Modulen und Digital-Rights-Management mag die Antwort ganz anders ausfallen. Hier besteht die Möglichkeit, dass z. B. durch „Remote Attestation“ die freie Wahl von Software massiv eingeschränkt wird [Diet]. Die daraus resultierende Einschränkung von Rechten des Besitzers einer solchen Plattform muss sehr sorgfältig gegen die Wirtschaftsinteressen abgewogen werden. Das Bundesamt für Sicherheit in der Informationstechnik (BSI) sagt dazu:

„Erst auf Grund dieses Mangels an vollständiger Kontrolle durch den Besitzer kann die Vertrauenswürdigkeit der Plattform auch gegenüber Dritten garantiert werden. Somit stellt der Kontrollverlust in gewisser Weise den Hauptnutzen eines TPM dar.“ ([BSI], zitiert nach [Diet, S. 61])

Dieser Gedanke lässt sich auch auf das gegenwärtige Dilemma in der Browserarchitektur übertragen: Dem Wunsch nach mehr Aggregation fremder Dienste und Daten steht die Gefahr des Missbrauchs von Credentials sowie von Cookies gegenüber. Der Grund ist schlicht darin zu sehen, dass im Moment Programme als Stellvertreter des Benutzers agieren und in ihren Aktionen nicht wirklich eingeschränkt werden können. Somit fehlt eine Basis für zuverlässige Willenserklärungen von Seiten der Benutzer.

15.4 Prozessorientierte Security

In heutigen, teilweise sehr sensiblen Anwendungsbereichen wird häufig versucht, Sicherheit durch einen sicheren Verarbeitungsprozess zu erreichen. Ob EC-Karte, E-Banking oder Wahlmaschine, der Bürger muss darauf vertrauen, dass der Prozess der Implementation und Durchführung der gewünschten Aktion fehlerfrei ist und keine Unbefugten Zugang zu seinen Daten erhalten, da die konkrete Willens-

äußerung des Bürgers nicht kryptografisch abgesichert wird. Hier tun sich durch die Verwendung von Signaturkarten und Applikationen auf Basis von Web-Services in Zukunft ganz neue Möglichkeiten auf, z. B. eine Erlaubnis granular und klar eingeschränkt zu erteilen. Die Vorteile objektbasierter Sicherheit zeigen sich jedoch auch klar im Fall von Infrastrukturen. So verwendet die Implementation der deutschen Gesundheitskarte reichlich Protokolle aus dem Bereich der Web-Services, da diese zum einen interoperabel sind und zum anderen granulare Willenserklärungen und Schutzmaßnahmen erlauben. Die Zugriffskontrolle wird also zunehmend semantisch mächtiger und damit zur „Intentional Access Control“, bei der jegliche Aktion darauf geprüft werden kann, ob sie tatsächlich der Intention des Urhebers entspricht. Unter anderem könnte sich das Phishing-Problem so lösen lassen.

Das große ungelöste Problem der objektbasierten Sicherheit ist jedoch bisher die mangelnde Benutzbarkeit, u. a. durch die schlechte Integration in die vorhandenen Systeme. Kein Wunder, dass Firmen solche Lösungen nicht annehmen wollen, wenn die Verwendung eines Class3-Readers zusammen mit einer Smartcard mehrere Anrufe beim Service benötigt und die Applikationen sich auch anschließend nur von Administratoren ohne Problem betreiben lassen. Die Usability objektbasierter Sicherheit für die Entwickler ist gleichfalls immer noch mangelhaft.

15.5 Modellbasierte, integrierte Security

Testwerkzeuge und Web-Application-Firewalls (WAF) haben eines gemeinsam: Sie sind den Applikationen nachgelagert und müssen oft mühsam konfiguriert oder programmiert werden, um die Applikation zu testen bzw. zu prüfen. Selbst bei „lernfähigen“ WAFs stellt sich letztlich doch die Frage, wieso sie etwas mühsam lernen müssen, was doch bei der Entwicklung der Applikation bekannt gewesen sein müsste – nämlich die Sprache, durch die mit der Applikation kommuniziert werden kann. Wer den Servlet-Filter-Mechanismus verwenden will, um eine Applikation weiter abzusichern, steht ebenso vor dem Problem, die Input/Output-Seite quasi nachbilden zu müssen, damit korrektes Filtern stattfinden kann.

Die Lösung für dieses Problem kann nur darin liegen, dass die Modelle der Applikation, die zu ihrer Konstruktion verwendet wurden (und die heute oft bereits für die Generierung von Datenbank- oder GUI-Schnittstellen verwendet werden), auch außerhalb der Applikation die Test- und Prüfwerkzeuge treiben können. Dieser statische Export von Modellinformation kann dann noch durch dynamische Meta-Daten ergänzt werden, sodass z. B. Testwerkzeuge sich automatisch an geänderte GUIs anpassen können. „Gute“ Softwareentwicklung wird in der Zukunft automatisch die Entwicklung der Test- und Prüffähigkeit von Applikationen als Teil der Applikationsarchitektur einschließen und damit eine gestaffelte Sicherheitstechnik ermöglichen.

15.6 Zuverlässigkeit und Verfügbarkeit

Trotz ihres Umfangs besitzen dieses Buch und der Grundlagenband wesentliche Lücken: So konnte z. B. die Sicherheit mobiler Systeme nur kurz angerissen werden. Auch die Sicherheit in Peer-to-Peer-Systemen hätte den Rahmen schlicht gesprengt, und so bleibt nur der Verweis auf eine aktuelle Arbeit von Marc Seeger zum Thema Anonymität in P2P-Filesharing-Systemen [Seeg], in der die verschiedenen Ansätze und Schwächen vorgestellt werden. Das Thema „Sicherheit trotz Erweiterbarkeit“ scheint langsam bei den Entwicklern angekommen zu sein: In aktuellen Arbeiten zu sicheren Betriebssystemen wie seL4 [seL4] oder Genode [Geno] wie auch in den neuesten Browserentwicklungen spielt es eine immer größere Rolle.

Aus theoretischer Sicht ist jedoch ein Mangel am meisten zu bedauern: Das Buch konzentriert sich zumeist auf den Aspekt der Korrektheit von Software oder Systemen. Zwar wird vielfach erklärt, dass auch Zuverlässigkeit und Verfügbarkeit untrennbar zur Sicherheit gehören, allerdings werden diese Aspekte in der Praxis meist kaum beachtet (eine Ausnahme bildet evtl. das sel4-Umfeld). Es ist jedoch zu erwarten, dass durch die zunehmende Ausbreitung von Rechnern bis hin zur kompletten Durchdringung der Gesellschaft mit Informationsverarbeitenden (und -sendenden) Geräten die Frage der Zuverlässigkeit und Verfügbarkeit von Software sehr viel wichtiger werden wird.

Leider ist dieser Aspekt alles andere als leicht zu fassen und zu berücksichtigen. Die Probleme, die unsere Applikationen zurzeit mit Denial-Of-Service-Attacken aufweisen, sind ein deutlicher Hinweis darauf. Im Hintergrund des Problems steht das Konzept der „Liveness“ einer Applikation und wie diese zu sichern ist. Eine Applikation muss eben nicht nur korrekt sein, sondern auch Fortschritte machen. Die Garantie, dass eine Applikation in der Lage ist, fortzuschreiten, ist jedoch in der Praxis kaum zu geben. Wir scheitern meist ja schon am Nachweis der korrekten Funktionsweise einer Applikation. Im Sinne garantierter „Liveness“ kommen jedoch noch sehr wichtige andere Faktoren dazu:

- Die korrekte Benutzung der Schnittstellen darf nicht zum „Aushungern“ einer Applikation führen, d. h. die Ressourcen des Servers dürfen durch Clients nicht beliebig verbraucht werden können.
- Änderungen an Konfigurationen und Policies (z. B. das Wegnehmen von Rechten) dürfen nicht dazu führen, dass die Funktion der Applikation plötzlich nicht mehr gegeben ist. Die meisten formalen Prüfungsverfahren gehen jedoch meist von einem monotonen Zuwachs an Rechten aus.
- Für komplexe Infrastrukturen muss ein formaler Nachweis geführt werden können, dass die gewünschte Konnektivität tatsächlich besteht.

Hier stößt man an die Grenzen der theoretischen Berechenbarkeit. Bessere Verfahren der automatischen Prüfung von Sicherheitsmodellen sowie der vermehrte Einsatz starker Virtualisierung (ähnlich den LPAR-Konzepten bei Mainframes) könnten hier in Zukunft helfen. Das Konzept der Objekt-Capabilities, wie er in

diesem Buch geschildert wird, dient ebenfalls der verbesserten Isolation und könnte damit ebenfalls zur Lösung des Liveness-Problems beitragen.

Die alltägliche tausendfache Nutzung von EC-Karten, die Einführung von Reisepässen mit biometrischen Merkmalen auf RFID-Chips oder auch die geplante Gesundheitskarte zeigen, wie sehr die Informatik speziell beim Thema Sicherheit schon heute unsere Gesellschaft durchdringt und beeinflusst, und umgekehrt auch von ihr beeinflusst wird. Die Frage nach sicherer Software und sicheren Systemen ist deshalb von großer gesellschaftlicher Bedeutung. Die Wahrnehmung von Sicherheitsproblemen muss in Zukunft über die Technik des einzelnen Systems hinausgehen und den gesellschaftlichen und ökonomischen Kontext berücksichtigen.

Literatur

- [BSI] Bundesamt für Sicherheit in der Informationstechnik, www.bsi.de/sichere_plattformen/trustcomp/infos/tpm_report/tpm_grundlagen.html
- [Geno] Genode Operating System Framework, <http://genode.org/>
- [Diet] P. Dietrich, DRM und Trusted Computing – Herr der eigenen Hardware?, Master Thesis, Hochschule der Medien Stuttgart 2006
- [Schn07] B. Schneier, Do We Really Need a Security Industry? Wired Magazine, May 2007, <http://tinyurl.com/23b3av>
- [Seeg] M. Seeger, The current state of anonymous filesharing, BSc. Thesis, Hochschule der Medien Stuttgart 2008, <http://www.marc-seeger.de/wp-content/uploads/2008/07/thesis-the-current-state-of-anonymous-filesharing.pdf>
- [seL4] Secure Microkernel Project, <http://ertos.nicta.com.au/research/sel4/>
- [Step] N. Stephenson, Die Diktatur des schönen Scheins – Wie grafische Oberflächen die Computernutzer entmündigen, Goldmann-Verlag 2002

Index

A

- Abfangen temporärer Fehler, 479
- abgesicherter monolithischer Kernel, 135
- ablaufbasierte Kontrolle, 104
- Abschalten von Javascript, 503
- Absichern von Application-Servern, 347
- Absicherung der Ressourcen, 350
- Absicherung der Verbindungen, 349
- Absicherung von Servern, 169
- Abwehr auf der Netzwerkschicht, 39
- Access Control Context, 95, 203, 214
- Access Control Lists, 108, 118
- Access Control Markup Language, 197
- Access Control Matrix, 564
- Access Control Policies, 195
- Acegi, 295
- ACL, 108, 118, 291
 - Beispiel, 119
 - Malware, 119
 - Solitaire, 119
- Active Object, 470
- Add-Ons, 445
- Administration, 174
- Administrationsfunktion, 288
- „after the fact“-Sicherheitsmechanismen, 417
- AJAX, 69
 - Java Script Object Notation, 75
 - On-Demand JavaScript, 74
 - Techniken, 72
 - XMLHttpRequest, 72
- aktionsintegrierte Sicherheit, 554
- Alias-Antipattern, 25
- Alias-Definition, 475
- „Alles oder Nichts“-Prinzip, 116
- allow_url_fopen, 510
- allow_url_include, 510
- Amazon.com, 409
- Ambient Authority, 99, 107, 158, 416, 446
- Ambige Interfaces, 476
- Analysis/Paralysis Pattern, 553
- Anfangs- bzw. Endspanne, 571
- Angriffe
 - Klassifikation, 12
- Angriffsformen
 - lokale, 15
- Anreicherung von Privilegien, 152
- Antipattern, 99, 473
- API in Form paarweiser Methoden, 509
- Append-Only-Recht, 152
- Application Integration in
 - SSO-Infrastruktur, 285
- Application Level Security, 262
- Application Server
 - LTPA, 335
- Application Server Security, 313
- Application Tower, 230
- Application-Server, 330
 - Architektur, 318
 - Authorisierungscode in der
 - Applikation, 329
 - Beispielarchitektur, 320
 - ClassLoader, 325
 - Concurrency, 316
 - Credentials für den Zugriff auf
 - Backends, 329
 - CSlv2, 318
 - Delegation mit Neuerzeugung
 - des Subjects, 346
 - Entry-Context-Diagramm, 325
 - Formen der Authentisierung, 324

- Identity Assertion, 334
- Identity Token, 334
- initiale Authentisierung, 332
- JAAS-Framework, 315
- keine eigenen Threads erzeugen, 316
- Komponentenarchitektur, 316
- Multithreading, 316
- Neuerstellung eines Subjects, 337
- POLA, 354
- Probleme mit ClassNotFoundException-Exceptions, 327
- Reverse Proxy, 343
- Security-Context-Diagramm, 321
- Security-Interfaces, 328
- Separation of Concerns, 313
- Separation of Context, 313
- Sicherheits-Framework, 330
- Single-Sign-On, 319
- Subject Propagation, 345
- Thread und Requestcontext, 316
- Verwaltung von Credentials, 322
- Zertifikate, 339
- Application-Server-Cluster, 319
- Application-Server-Verbund
 - Subject-Erzeugung, 342
- Applikationsarchitektur
 - Aufruf von Sicherheitsprüfungen vergessen, 233
 - Aufteilung in System und Businesslogik, 276
 - Businesslogik, 275
 - Eliminieren von Trusted Code, 479
 - Eliminieren von Code, 479
 - Eliminieren von Fehlern, 478
 - Multi-Tier, 233
 - sicherheitskritische Aspekte, 231
 - Templates, 274
 - Wartbarkeit der Sicherheit, 233
- Arbeiten im Auftrag, 261
- Architektur
 - des OKWS-Web-Servers, 170
 - einer paravirtualisierten Lösung, 141
 - eines Application-Servers, 318
- Architekturschwäche, 509
- Argumentation für POLA, 106
- ASLR, 33
- Aspekte, 481
- Assertion, 404
- Attacken, 70
 - Alias-Problem, 25
 - Angriff auf Password-File, 37
 - Application-DOS, 27
 - auf Ebene der Semantik, 61
 - auf REST Architekturen, 21
 - auf Web2.0, 70
 - Buffer Overflow, 30
 - Cross-Site-Request-Forgery (XSRF), 20
 - Cross-Site-Scripting (XSS), 18
 - durch Race Condition, 472
 - externe, 18
 - Genuine AJAX/Web2.0-Attacken, 79
 - Lokale auf Applikationslevel, 30
 - neue Angriffsformen, 17
 - Shatter Attack, 35
 - soziale, 67
 - Surf Jacking, 23
 - Übersicht, 12
 - UNIX Shellscript und Utility-Attacken, 36
 - Visual Spoofing, 25
 - War-Googling, 29
 - XML-basiert, 24
 - Zeichencode, 25
 - zeitliche Entwicklung, 15
- Attacken im Navigator, 508
- Aufgabe des Supervisors, 448
- Aufmerksamkeit, 552
- Ausführungsidentität, 257
- Auslieferungszustand, 348
- Aussagen, 403
- Authentisierung in Web-Applikationen, 278
- Authentisierungscode in der Applikation, 329
- Authentisierungs-Framework, 315
- Authentisierungsidentität, 257
- Authentisierungsmethoden
 - Hierarchie, 16
- Authentisierungsobjekt, 289
- Authentisierungs-Token, 236
- Authorisierung, 186, 208
- Authority, 433
- Authority Reduction in Vista, 154
- Authorization-Provider, 341
- Automatic Processes, 553
- automatisches Update, 535
- automatisierte Testtools, 60
- Automobilbau, 91
- Automobilindustrie, 120
- Automotive-Bereich
 - OSGI, 365
- Autorisierung, 290
- Autorisierungsidentität, 257
- Autorität, 106
- Autorität aus Designation, 556
- Autorität und Interaktion, 579
- Autoritäten, 403

Autoritäten der Zugriffskontrolle, 197
Autoritätstrichter, 449, 455

B

Backend Security

Defense in depth, 270
unterschiedliche Views auf die
Datensätze, 271

Bedeutung der Identität des Ausführenden,
207

Bedrohungsmodelle für den Renderer, 529

Behavior Rules, 590

Behavior Sensitive, 456

Beispiel für eine Closure, 443

Beispiel OKWS-Web-Server, 169

Beispielarchitektur eines

Applikationsservers, 320

Benutzbarkeit von Techniken und
Mechanismen, 533

Bequemlichkeit vs. Sicherheit, 545

Bestimmung der Gefährlichkeit, 433

Betriebssysteme

Capabilities, 152

Confused Deputy, 129

durch VMs gesicherter Kernel, 136

Isolation durch Trennung der

Adressräume, 129

Kernel-Datenstrukturen, 135

Komponenten beim Microkernel, 132

Mittel der Isolation, 129

monolithischer Kernel, 131

Sicherheitsarchitektur, 127

Treiberschema unter C, 136

Beweisbare Plattformensicherheit, 610

Bibliotheken

Classloader-Positionierung, 355

BinarySecurityToken, 397

bind(UserId, Passwort)-Aufruf, 282

Blacklist, 420

Blind Spot Bias, 550

Blindheit, 552

breiter Zugang auf Daten, 270

Browser, 504

Same Origin“-Prinzip, 78

Sicherheitsmodelle Internet Explorer,
Mozilla/Firefox, 78

Browser Memory Protections, 32

Browser Sicherheit

bösartiger Renderer, 521

Browserarchitektur, 504

Buffer Overflows, 30

Beispielcode zur Erzeugung, 31

Bug oder Architekturschwäche, 510

Bundle, 366

Business Rule Engine, 196

Businesslogik, 274

Bypass, 420

Bytecode Verifier, 188

C

Caching, 412

Caja-Projekt, 530

Caller-Identity, 217

Callgraph, 445

Canaries, 32, 33

Capabilities, 107, 111, 138, 426, 581

Formen, 427

Objekt Capabilities, 111

Capability

Filedeskriptor, 152

widerrufbar, 451

Capability Architecture, 526

Capability Safe, 530

Capability-basierte Architektur,
519, 524

Capability-System, 437

Cap-Browser, 523

Cap-Desk, 523

Captcha, 71

Card Reader

vorinstallierte Schlüssel, 377

Caretaker, 449

Modellierung, 576

Caretaker Sicherheitsbaustein, 576

CAS, 225

Casts, 189

Cell, 352

Chain-of-Responsibility“-Pattern, 375

ChokePoint, 279

Chroot, 152

Claim, 404

Classloader, 199, 217

Beispielcode, 367

Default-Lookup-Policy, 326

Classloader-Hierarchie für Application-
Server, 327

Classloader-Vorgang, 188

ClassNotFoundExceptions, 327

CLDC, 380

C-Lists, 145

Closures, 207, 440, 442

Code Access Control in Application-
Servern, 354

Code Access Security, 224

Code Security für Enterprise-
Applikationen, 223

- Codeanalyse, 433
- codebasierte Policies, 203
- Codepoint, 27
 - Abbildung auf Fonts, 26
 - Abbildung auf Glyphs, 27
- cognitive Biases, 550
- „Comet“-Anwendungen, 77
- Community Sites, 78
 - Sicherheitsprobleme, 78
- Complete Mediation, 291, 388, 428
- Concurrency, 462
- Concurrency und alternative Verfahren, 472
- Concurrent Constraint Programming, 578
- Conditioning, 552
- Confinement, 447, 495, 570
- Confirmation Bias, 550
- Confused Deputy, 226, 421, 524
- Confused-Deputy-Problem, 421, 580
- Connected Limited Device Configuration (CLDC), 380
- Connection Pool, 231
 - Austauschen des Principals, 359
 - Identitätspropagation, 358
- Constraints, 606
- Container
 - Ausführungskontexte selber erstellen, 330
- Container Managed Authentication, 288
- Container Managed Security, 280
- Container-Architektur, 314, 317
- Container-Managed-Authentisierung, 285
- content-basierte Filterungen, 239
- Continuation Passing, 472
- Control Sensitive, 456
- Convenience Interface, 447
- Coordination Language, 466
- Copy-Programm, 107
- CORBA-Infrastruktur, 232
- crashme, 60, 145
- Credential, 248, 551
 - Absicherung, 349
 - SSO-Schlüssel, 353
 - Verwaltung, 322
- Credential-Diagramm, 323
- Cross Domain Proxy, 74
- Cross Site Access Control, 23
- Cross-Cell-Trust, 353
- Cross-Cutting Concern, 233
- Cross-Site-Request-Forgery, 21
- Cross-Site-Scripting
 - Attacken, 18
 - Merkmale, 21

- CSlv2, 255
 - Delegation, 339
 - Token, 286
- Currying, 452

D

- Data Execution Protection, 33
- Datenbank-Views, 174
- Datenbankzugriff durch den Server, 359
- DBAccessPrincipal, 358
- DB-Proxies, 172
- Deadlock, 465
- Deadlock-Problem, 465
- Decentralized Mandatory Access Control, 153
- Decorator, 451
- Decorator Pattern, 221
- Deep Linking, 279
- De-Fakto, 571
- „default-is-allow“-Logik, 494
- Default-is-deny, 94, 348
- Default-Policy-Implementation, 198
- defensive consistency, 464
- defensive correctness, 439
- Degeneration von Architektur, 475
- De-Jure, 570
- deklarative Rollendefinition, 263
- deklarative Security, 248
- deklarative Sicherheit
 - Designfragen, 268
- deklaratives Modell der Zugriffskontrolle, 266
- Delegation
 - durch Weitergabe von Credentials, 249
 - implizites Vertrauen, 304
- Delegation mit Propagation des Subjects, 346
- DEP, 33
- Dependency-Injection-Prinzip, 138
- Deputy Pattern, 593
- Design der Benutzerschnittstelle, 555
- Design sicherer Systeme
 - Grundprinzipien, 91
 - Regeln, 104
- Designation, 557
- Designation und Autorität, 455
- Deskriptorsysteme, 145
- Diagramm, 449
- Die Kunst der Täuschung, 61
- Diebold, 388
- Differenzierung des Take/Grant – Ansatzes, 574
- Digital Rights Management (DRM), 544

- direkte Authentisierung, 431
- Dirty Assignments, 440, 496
- Discretionary vs. Mandatory, 544
- Diskreditierung von Capability-basierten Systemen, 573
- Diskussion der Spring Security Security, 302
- Dispatch-Funktion, 443
- DLL-Hell, 327
- Domain, 258
- Dominanz der heute gängigen Systeme, 418
- Double Checked Locking Pattern, 468
- Double-Decoding, 49
- Download von Applikationen, 378
- Drop-Target, 528
- Duplizierung sicherheitsrelevanter Funktionen, 230
- Dynamic SQL, 231
- dynamische Rechtemanipulationen, 566
- dynamische Services, 364

E

- E, 451
- Eigentum und Verfügung, 544
- Eigentümer, 419
- einfaches Sicherheitsproblem, 591
- Eingabe des PINs, 373
- eingefahrene Verhaltensweise, 553
- Einsatz eines Caretaker-Sicherheitsbausteins, 450
- EJB, 248
 - Architektur, 460
 - Authorization Table, 256
 - breiter Zugriff auf Backend-Daten, 254
 - Functional User, 254
 - Identitäten und Rollen in einem EJB-Modul, 253, 254
 - Interceptor-Konzept, 330
 - Resource-ref, 254
 - Security-View, 254
- embarrassingly parallel, 463
- Embedded Control
 - Hybride Systeme, 142
 - Plattformen, 371
- Emotional Bias, 551
- Encodings, 25
- Ende-zu-Ende-Sicherheit, 239
 - Bruch, 242
 - fehlerhafte Konfigurationen, 248
 - Identität, 257
 - Kreuzen von Requests, 245
 - Propagation des Zugriffs durch authentifizierte User, 244
 - Propagation von Identität, Recht und Aufruf, 245
 - Requestvalidierung, 261
 - Sicherheitsprobleme Beispiel Search, 243
- Endowment, 447
- End-to-End-Argument, 191
- End-to-End-Security, 229
- Enterprise Security, 229
 - Access Control, 231
 - drängende Fragen, 234
 - Grundlagen, 230
 - Grundprinzipien, 247
 - Impersonation, 236
 - Search Request, 246
 - Vertrauen in Intermediate, 304
- Enterprise Service Bus, 235
- Enterprise-Applikationen
 - Code Security, 223
- Enterprise-Search-Plattform, 246
- Entropie, 140
- Entry-Context-Diagramm, 324, 325
- Entscheidbarkeit von Sicherheit, 564, 580
- Entscheidungsfindung, 552
- Entwicklersicht vs. User-Sicht, 549
- Envelope-Techniken, 236
- Erlaubnis, 462
- Erreichbarkeit, 438
 - von Code, 434
- Erstellung einer geprüften Identität, 208
- Erstellung eines Subjects, 331
- Erweiterung eines Supervisors, 437
- Erweiterungen, 434
- Erweiterungsarchitektur, 35
- Erweiterungsdimensionen von Take/Grant Modellen, 574
- Erzeugung von Subjects, 342
- Erziehung von Usern, 553
- Event-Driven-System, 236
- eventgesteuerte Architektur, 381
- eventgetriebene Systeme, 235
- Event-Loop-Modell, 470
- Evolution der Java-Sicherheit, 184
- E-Voting, 388
- Exceptions, 447
- executable spoofing, 500
- expecco, 610
- explizite Allokation, 423
- extensible Access Control Markup Language, XACML, 403
- externalisierbar, 420

F

Facets, 439, 451
 Fakten, 587
 Fassade-Pattern, 270
 Fat Client, 231
 Fehlen statischer Typen, 303
 Fehleranalyse
 als didaktisches Prinzip, 485
 als theoretisches Prinzip, 486
 Hypothesenbildung, 487
 qualitative Analyse der Fehler, 487
 Fehlermuster, 503
 Fehlertoleranz, 238
 Festlegung von Rechten in SELinux, 177
 Filedeskriptor, 107, 434
 Filesystemzugriffe
 Ausnutzen nicht-atomarer File-
 Zugriffe, 38
 nicht-atomare, 15
 Filter, 419
 Filterketten
 Gefahr, 48
 Filtern
 Blacklists, 49
 Whitelists, 51
 Filtertechnologie, 420
 FINREAD, 371
 Architektur, 374, 381
 Card Reader, 371
 Kartenleserterminal, 374
 Konzept, 371
 Service Control Object, 382
 Softwarearchitektur des Card Readers,
 375
 Stiplet, 372
 Trust- und Key-Management, 376
 verbotene Bibliotheken und
 Mechanismen, 381
 Virtual Machine, 380
 Firefox-SSL-Dialog, 556
 Firewall
 besondere Zone, 287
 Firewalls, 417
 Direkte Verbindung trotz Firewalls, 39
 Entscheidungsfindung eines
 Paketfilters, 40
 Firewall-Scripts, 46
 problematische Eigenschaften von
 Applikationen, 42
 Web-Application Firewall (WAF), 48
 XML-Firewall, 52
 Fixpoints, 596
 Fließen von Zugriffsmöglichkeiten, 447

Föderation, 431
 FORM-Tag, 527
 Frames, 73
 Framing, 552
 Front-End vs. Back-End Security, 430
 Fully Trusted
 Angriffsmöglichkeiten, 355
 Fully Trusted Server, 226
 Fully Trusted“.NET, 355
 Functional User, 231
 Sicherheitsmerkmale, 231
 Functional UserID, 231
 Funktion des Sealer/Unsealer-
 Sicherheitsbausteins, 453
 funktionaler Ansatz, 440
 Funktionen, 93
 Fuzzer, 60

G

Game Theory, 552
 ganzheitliche Betrachtung von sicheren
 Systemen, 4
 Gate Guardian, 37
 GAU in der Infrastruktur, 245
 Geeks, 550
 Geektum, 550
 Gefahrenpotenzial, 433, 441
 gefährliche Funktion, 481
 Gefährlichkeit von Server-Applikationen,
 164
 gefahrloses Laden, 436
 Gehorsamkeitseffekt, 559
 generalisiertes Maximierungsproblem,
 592
 geprüfte Identität, 198
 Geschäft mit IT-Sicherheit, 417
 get- und set-Methoden, 438
 Global navigierbare Datenstrukturen, 94
 Global Scope, 441
 global scope pollution, 495
 globale Namespaces, 474
 globale Navigation, 474
 globale Variablen, 440
 globaler Namespace mit Ressourcen,
 151
 globales Administratorpasswort, 305
 globales DOM, 511
 Global-Scope-Zugriffe, 527
 Globus, 306
 Granovetter-Diagramm, 446
 Granter, 523
 granulare Zugriffsmöglichkeiten, 440
 Granularität, 93

Grenzen der kanalbasierten Sicherheit, 390
Grid Security, 303
 Delegation durch Proxy-Zertifikate, 306
 End-Certificate, 306
 Key-Management, 308
 Standards, 305
Grid Security Infrastructure, 305
Grids, 303
grobe Designfehler in OO-Bibliotheken, 527
größte Gefahr, 98
Grundkonzept der Virtualisierung, 456
Grundlegende Mechanismen sicherer Software, 419
Grundprinzipien
 Complete Mediation, 94
 Economy of Mechanism, 93
 Fail-Safe-Defaults, 94
 Least Common Mechanism, 100
 Open Design, 97
 Principle of Least Privilege, 99
 Psychological Acceptability, 103
 Separation of Privilege, 98
 sicherer Software in JAVA EE und EJB, 250
Guard Object, 221
Gültigkeit von Capabilities, 449

H

Handeln im Auftrag, 164
Hardening, 347
Harrison-Ulmann-Ruzzo, 564
Hashing von Passwörtern, 240
Hauptproblematik der momentanen Browserarchitekturen, 516
HEALTH@NET, 608
Heap Spraying, 34
Heimnetzwerk, 365
Hidden Data Flow, 478
hidden fields, 58
hierarchische Modi, 421
hierarchische Rechte Modelle, 98
hierarchische Verifikation eines Software-Downloads, 379
Higher-Order-Function, 207, 440, 441, 442
Historie von Rechten, 568
Hobbles und Patches, 507
hobbling, 507
Holder-of-Key, 334
Homographen-Problem, 26
Honeybot, 461

Hot-Plug, 364
Hot-Swap, 364
HTML Sicherheitsproblematik, 527
http
 Remote User Header, 286

I

Identität
 initiale Authentisierung, 332
 Registries und Subjects, 332
 von Code, 203
Identitätsmapping
 applikationsgesteuerte Authentisierung, Nachteile, 258
 Credential-Mapping-System, 258
 Mapping einer Caller-Identität, 259
 Möglichkeiten, 261
 run-as-Identität, 258
Identitätspropagation in EJB, 257
Identity
 Propagation zu Datenbanken, 358
Identity Assertion, 319, 334, 337
Identity Assertion und Delegation, 339
IFrames, 73
IMG-Tag, 72, 527
Immutable Laws of Security, 121
Impersonation, 579
Impersonation mit JAAS, 294
Impersonifikation, 164
Impersonifizierung
 Vor- und Nachteile, 164
Implementation des Observer Patterns, 463
Implementationen des Interceptor Patterns, 460
Implicit Authority Demands, 24
implizite Allokation, 133, 421
implizites Vertrauen, 304
Inattentional Blindness, 551, 552
Identitätsmapping
 Backend, 259
Infrastruktur, 229
 COTS-Applikation, 309
Infrastruktur vs. Application Level Security, 429
initiale Authentisierung, 234
initiale Authentisierung via JAAS-Log-in, 211
INPOL, 269
INPOL-Projekt, 234
Input, 505
Input-/Output-Validierung
 auf Applikationsebene, 48

- Input-Validierung
 - Bankenserver Fehler, 65
 - Grundlagen, 38
 - Installation, 179
 - Administratorrechte, 179
 - Singularity, 179
 - Installer, 179
 - instanzbasierte Access Control, 273
 - Integration der JAAS-Authorisierung, 214
 - Integration von Applikationen, 309
 - Intention, 547
 - Intention des Nutzers, 544
 - Intention, Programm und Effekt, 547
 - Interaktion und Autorität, 422
 - Interaktionsmöglichkeit, 421
 - Interception, 195, 420
 - Interceptor, 459
 - Interceptor-Framework, 301
 - Interleaving, 463
 - Intermediate, 257, 308
 - Intermediate-Server, 429
 - Internet Threat Model, 115
 - Interpreter, 494
 - Inversion-of-Control, 135
 - Inversion-of-Control-Architektur, 436, 457
 - Inversion-of-Control-Prinzip, 138
 - IOC, 135
 - IOFUZZ, 145
 - IPSec, 248
 - IPTables, 43
 - Isolation, 100, 420, 447
 - durch Nicht-Erreichbarkeit, 420
 - von Autoritätsfluss, 573
 - von Code, 327
 - Isolationstechniken, 127
 - Isolieren von Komponenten, 351
 - isolierende VMs, 356
- J**
- JAAS, 185
 - Credential, 209
 - doAs, 213
 - doAsPrivileged, 213
 - doPrivileged()-Operation, 221
 - Integration in Struts, 284
 - Logincontext, 209
 - Loginmodule, 209
 - Log-in-Sequenz, 210
 - mit Struts, 284
 - Principal, 209
 - Providerpattern, 210
 - Subject, 209, 213, 215, 290
 - Subject.doAsPrivileged, 216
 - JACC, 341
 - Container, Applikation und Policy Provider, 342
 - Jail, 152
 - Java 2 Security, 185, 217
 - Java Authentication Authorization
 - Contract for Container, 341
 - Java Authentication und Authorization Service, 185
 - Java Code Access Security, 198
 - Java Connector, 258
 - JAVA EE, 186, 248
 - Annotationen, 276
 - Applikationsinterfaces, 251
 - Authentisierung, Identitäten und Rollen in einem Web-Modul, 254
 - Container Architecture, 341
 - Delegation, 256
 - Deployment-Deskriptoren, 252, 263
 - Fluss der Identität eines Aufrufers, 256
 - getCallerPrincipal, 256
 - getUserPrincipal, 255
 - Impersonation, 256
 - isUserInRole, 253, 267
 - Permission, 253
 - Ressource, 253
 - Rolle, 253
 - run-as, 257
 - Use-Caller-Identity, 257
 - web-app.xml, 285
 - Java Security
 - checkPermission, 200
 - doAs() versus doAsPrivileged(), 215
 - Java Sprach- und Plattformsicherheit, 183
 - Java-Call-Stack, 225
 - Java-Memory-Modell, 468
 - Javascript, 72
 - Angriffsmöglichkeiten durch JavaScript, 80
 - eingebettetes Script in Nutzerprofilen, 79
 - eval()-Funktion, 75
 - Hijacking, 80
 - Sicherheitsprobleme, 80
 - Subset, 530
 - Javascript
 - lambda, 493
 - Java-Security
 - Accesscontrolcontext, 201
 - Accesscontroller, 201
 - checkPermission, 220
 - Erweiterungen und Ableitungen von Klassen, 193
 - Grant Principal, 217

- heute, 187
- Impersonifizierung, 214
- Java 2 Security anschalten, 200
- Kernkonzept der Java-Security, 184
- Klassen und Protection Domains, 201
- Liste der Permissions, 200
- native Funktionen, 194
- Package als protected, 194
- Packages, 194
- Packages durch Signaturen schützen, 194
- Packages vor Aufruf schützen, 194
- Permission, 201
- Permissions als Rollendefinitionen, 225
- Permission-Überprüfungen, 200
- Protection Domain, 198
- Prüfung eines Zugriffs, 199
- Securityexception, 202
- Securitymanager, 201
- Stack.Walkback, 214
- Struktur der Protection Domain, 199
- Übersicht, 186
- Java-Server-Code, 162
- JMX-Konsole, 29
- JNDI-Verzeichnis, 475
- JSON, 75
- just in time least authority, 568

K

- kanalbasierte Sicherheit, 425
- kanonische Form von XML, 392
- Kapselung, 437
- Kapselung durch virtuelle Maschinen, 139
- Kategorisierung, 549
- KBM, 586
- Kerckhoff'sches Prinzip, 97
- Kernel-DOS-Attacken, 133
- Key Continuity Management, 27, 538
- Key-Management, 537
- Kilim, 472
- Klassifikation der Schwachstellen Im Mozilla/Firefox Browser, 502
- Kleinstrechner, 116
- Knowledge Behavior Models, 580, 586
 - Definition, 590
- kognitive Tendenzen, 550
- kollaborative Sicherheitsmechanismen, 388
- komplexe Systeme, 552
- komplizierte Dialoge, 539
- Komponenten eines Servers, 165
- Komponenten und Applikation, 523

- Konditionieren, 553
- Konstruktion einer solchen simplen „Sandbox, 480
- Kontrolle der Autorität, 449
- Kontrolle der Instanziierung, 433
- kontrollierte Privilege Elevation, 204
- Konzepte der Psychologie, 549

L

- Label Based Access Control, 272
- Label-based-Access-Control-System, 155
- Labeling, 153
- Lademechanismus, 34
- Laden der Software, 434
- Laden und Linken, 434
- Lambda-Kalkül, 452
- Layer-Pattern, 233
- LBAC, 153, 272
- LDAP API, 282
- Lernen und Testen, 610
- Lexical Scoping, 442
- Light-weight Container Security, 295
- Lightweight-Third-Party Authentication (LTPA), 335
- Livelock, 465
- Liveness, 187, 422, 568
 - Aspekt, 348
 - Anforderung, 591
- Loader Isolation, 457
- Login-Config-Datei, 284
- lokale vs. fremdvermittelte Sicherheit, 536
- Löschen von Rechten, 568
- Lost-Update, 470
- LTPA-Key, 323

M

- Macht des Faktischen, 417
- Macros, 424
- Mailadressen als „First-Class-Objects“, 557
- Makro-Architektur, 453
- Makroarchitektur des DARPA-Browsers, 523
- Management Service Delegation, 264
- Managementobjekte
 - Rechte, 354
- mandantenfähige Software, 456
- Mandantenfähigkeit, 305
- Manifest File, 369
- Man-in-the-Middle (MITM), 64
- Map Reduce Pattern, 466
- MBeans, 354

Mechanismen, 415
 Mechanismen der Zugriffskontrolle, 431
 Mechanismus für granulare Übergabe, 557
 mehrstufige Initialisierung von Objekten, 461
 Memory Capabilities, 423
 Memory Protection, 187
 memory-safe, 33
 mentales Modell, 553
 Metamodell für die Modellierung operationaler Aspekte, 605
 Meta-Object-Facility (MOF), 605
 Metasploit Frameworks, 60
 Method-Permission-Table, 254
 Microkernel, 130
 Sicherheitsprobleme, 133
 Mikro-Architektur, 434
 „minderwertige“ Sicherheit, 551
 Mobmap, 444
 mod_security, 51
 Model-Checking, 607
 modellbasierte, integrierte Security, 619
 Modelle sicherer Kollaboration, 386
 modell-getriebenes Security-Engineering, 608
 Modellierung und Generierung, 605
 Modellierung von Nutzern, 604
 Modifizierbarkeit, 438
 modulare Sicherheitszone, 438
 Möglichkeit, Effekte zu erzielen, 435
 Monitor, 464
 monolithischer Kernel, 130
 Sicherheitsprobleme, 134
 monotone Approximation, 586
 Multi-Application-Architektur, 366
 Multi-Faktor-Authentisierung, 63
 multi-laterale Sicherheit, 2
 Multi-Level-Security, 153
 Multimedia-Streaming-Applikationen, 41
 multithreaded Server, 316
 Multi-Threaded-Zugriff, 439
 Multi-Threading, 102
 Multi-Tier-Applikation, 233
 Multi-Tier-Applikationen, 430
 Multi-Vendor Komponentensysteme, 363
 MySpace, 71

N

nachträgliches Eindämmen, 474
 Namen und ihre eindeutige Verwendung, 540

Namespace, 138
 durch Classloader Architektur, 366
 innerhalb einer VM, 368
 Kontrakt, 148
 Konzept, 217
 NAT, 42
 Navigations-Interface, 447
 Navigationsrechte, 107
 Nebenläufigkeit, 462
 und Sicherheit, 462
 Netfilter Framework, 44
 UML-Diagramm, 45
 Network Address Translation, 42
 Newsgator REST-Service, 410
 nicht fälschbare Referenz, 446
 Nicht-Determinismen bei der Implementation, 463
 Nicht-Determinismus, 468, 553
 nicht-deterministisch, 463
 nicht-orthogonale Aspekte der Nebenläufigkeit, 468
 Nickname, 541
 NIZZA, 141
 non-Functional-Requirements
 Delegation, 233
 Non-Privileged, 421
 NOP Slides, 34
 NoScript Firefox Extension, 22
 Notar, 462
 not-executable, 34
 Not-My-Problem-Fallacy, 552
 nur Sicherheit, die funktioniert, ist Sicherheit, 533
 Nutzerintention, 560
 nutzerspezifische Connection
 Eigenschaften, 358
 NX, 34

O

Object Capability, 221, 445
 Object Security, 391
 Object Spaces als Prozesse, 149
 objektbasierte Sicherheit, 97, 390, 425
 Merkmale, 388
 Sicherheitstechniken, 391
 Token, 396
 Objektreferenzen, 107
 OBSOC, 278
 OKL4, 133
 Open-Services-Gateway-Initiative [OSGI], 364
 Operator new, 433
 Opt-In, 33

- orthogonale Aspekte, 467
- Orthogonalität von Kapselung und
 - Concurrency, 471
- OSGI, 364
 - Bundle Signature, 370
 - Bundles, 364
 - JXTA-Kopplung, 365
 - Servicekonzept, 364
 - Services in einem Heimnetzwerk, 365
 - Versionierung, 364
- Out of Context Problem, 552
- „Outrage“-Theorie, 84
- Outsourcing, 412
- Overattention, 553
- OWASP, 13

P

- paarweise Methoden mit State, 476
- paarweiser Funktionsaufruf, 421
- Packages
 - kritische Funktionen, 355
- PAI Projekt, 605
- PAM, 208, 315
- Paper-Trail, 389
- Parallelität ohne Synchronisierung, 470
- Parameterübergabe, 440
- paravirtualisierte VM-Architekturen, 141
- Parsen von Input, 479
- Partitionierung von Funktionen, 480
- passive Subjekte, 573
- Passwort-Aging-Policies, 324
- Passwortsynchronisierung, 241
- Passwortwechsel, 287
- Patches, 117
 - Automatisierung der Exploit-
 - Erstellung, 118
 - Patchlogik, 117
 - Patch-Service, 117
- Pattern Recognition Bias, 550
- Peer-to-Peer-Classloader, 368
- Peer-to-Peer-Classloader-Modell, 369
- Performanceverlust, 226
- Perimeter Security, 115
- Permission, 99, 192, 198, 433, 435
 - versus Authority, 583
- Per-Reference-Semantik, 438
- Personal Firewalls, 111
- Personality Type Bias, 550
- Personas, 553
- Petnames, 541
- Pfadattacke, 223
- Phishing, 63
- Phishing-Angriff auf iTAN-Verfahren, 65
- PHP, 510
- PIN/TAN-Lösung
 - leichtgewichtig, 22
- Plan, 462
- Platform Security, 115
- Plazierung von Cookies, 33
- Pluggable Authentication Modules, 315
- Pluggable-Authentication-Mechanism,
 - 185
- Plug-Ins, 35, 134
 - Architekturen, 134, 135
- POLA, 100, 105, 151, 584
 - durch Capabilities, 426
 - Intermediate-Server, 430
 - Logfile Beispiel, 152
 - Prinzip, 447
 - vs. POLP, 105
 - zur Schadensbegrenzung, 5
- Polaris, 159, 546
- Policy Administration Point, 197
- Policy Decision Point (PDP), 197
- Policy Enforcement Point, 197
- Policy Information Point (PIP), 197
- Policy Providern, 341
- POLP, 99, 480
- Port Scanning, 98
- Portable Interceptor Pattern, 330, 458
- Powerbox, 449
- Prädikate, 587
- primitive commands, 566
- Principal, 208, 209, 210, 213
- Principle
 - of least Authority, 100, 105, 584
 - of least Privilege, 110, 480
- Principle of least surprise, 511
- Prinzip des Same-Origin, 514
- Privatnutzer, 117
- Privilege Escalation, 489
- privileged, 421
- privileged javascript execution, 494
- Privilegienanpassung
 - DecreaseMyRights, 157
 - in Applikationen, 159
 - in Betriebssystemen, 157
 - MakeMeAdmin, 158
 - subtraktives Verfahren, 157
- Privilegienmodi, 128
- „privilegierte“ Funktionen, 95
- problematische Sicherheitswarnung, 546
- Process Attachable Virtual Machines (PAVM), 356
- programmatische Security, 248
- programmierte Sicherheitsabfragen, 231
- Projection Bias, 550

prompt tab spoofing, 501
 Propagation einer serialisierten Form des
 Subjects, 345
 Propagation of Authority, 448
 Propagation von Identität, 267
 Propagation von Privilegien, 108
 Propagation von Rechten, 448
 Properties
 Implementation durch Capabilities,
 575
 und Capabilities, 573
 -Properties;Properties, 155
 Protection Rings, 421
 Protection State, 584
 Protection Systems, 584
 Proxy-Zertifikat, 306
 Vollmacht, 307
 Proxy-Zertifikat ([RFC3820]., 306
 prozessorientierte Security, 618
 Prüferidentität, 257
 Prüfung Operationaler Umgebungen, 602
 Pseudo Random Numbers (PNRG)., 15
 Psychologie der Verteidigung, 84
 psychologische Akzeptanz, 226
 psychologische Basisbedingungen
 menschlichen Handelns, 550
 psychologische Basismechanismen, 552
 Psychologische Voraussetzungen, 548
 public constructor, 437
 Publish/Subscribe-Pattern, 235
 Pull-Verfahren, 475
 Purse, 308

Q

gmail, 477
 gmail Architektur, 479
 Quellen von Autorität, 447, 454
 Query-Interface, 196
 Quotas, 101

R

Race Conditions, 175
 Race-Condition, 36, 463
 Rahmenbedingungen sicherer Software,
 417
 Randomisierung des Adressraumes, 33
 Rationalisation, 550
 Ratproxy, 61
 Read-Up, 155
 Realm, 258
 Re-Authentication, 358
 Rechte des Codes verringern, 258

Rechte und Eigentum, 543
 Rechte- und Informationspropagation, 571
 Rechteaübung, 533
 Rechtedefinitionen
 feingranular, 110
 Fließen, 113
 Propagation von Rechten, 114
 Rechtemenge, 112
 statische Rechtevergabe, 111
 Rechtekontrolle bei der Ausführung von
 Programmen, 118
 Reduktereduktion, 430
 Rechtezuweisung
 additiv und subtraktiv, 152
 Recognition Primed Decision Making,
 552
 Reduktion der Autorität, 110
 Reduktion von Autorität durch
 Softwarebausteine, 173
 Refactoring, 479
 Referenzen, 436
 Referenzen als Capabilities, 217
 Referenzmonitor, 109, 435
 Referenzmonitore
 Wildcard Problem, 159
 Refinement Prozess, 611
 Reflection und Sicherheit, 457
 Reflection-Mechanismus, 381
 reflektive Methode, 192
 Registries mit Objekten, 441
 Re-Identifizierung von Objekten auf
 Systemebene, 557
 Remote Admi, 174
 Replay, 97
 Replay-Attacken, 406
 Repräsentation von Identität, 331
 Representational State Transfer
 Architecture, 407
 RequestDispatcher, 292
 Reset des Routers Attacke, 21
 Residential Gateway, 365
 Ressourcenverwaltung, 101
 REST, 407
 Beispiel, 410
 Philosophie, 409
 Prinzipien, 409
 Services, 409
 Reverse Proxy, 280, 458
 RevertToSelf, 355
 revoker-Funktion, 451
 Rights Amplification, 453
 Risikowahrnehmung, 85
 Rule Engine, 268
 run-as, 258

S

- Safe Collaboration Language, 580
- Safety, 3
- Safety vs. Security, 416
- Saltzer und Schroeder, Grundprinzipien
- Same-Origin-Prinzip, 508
- SAML, 403
 - Advice, 405
 - Architektur, 405
 - Assertion, 404
 - Assertion Beispiele, 404
 - Assertion Element, 404
 - Attribute Statement, 405
 - Authentication Statement, 405
 - Authorization Decision, 405
 - Conditions, 405
 - Subject Statement, 405
- Sandbox, 108, 109, 118, 184, 195
- Sanitizing, 48
 - Gefahr, 48
- SAP-Architektur
 - VM, 357
- Satisficing, 552
- Schadensbegrenzung, 121
- Schadensminimierung, 99
- Schadenspotenzial
 - Autorität, 106
 - einer Software, 106
- Schlüssel, 427
- Schwachstellen herkömmlicher
 - Betriebssysteme, 140
- Scoll, 592
- Scoll Programme, 594
- Scoll-Ansatz, 580
- Scollar, 580, 596
- Sealed Process Architecture, 134
- Sealer/Unsealer Pattern, 453
- SECUml, 607
- Secure Channel Confusion, 406
- Secure Classloader, 199
- Secure Cookie, 23
- secure embedded L4, 133
- Security Assertion Markup Language (SAML), 403
- Security Constraint, 292
- Security Context
 - Application-Server, 322
 - Diagramm, 321
 - eines Requests, 316
- Security Patt, 458
- Security Policy, 192
- Security Token Issuer (STI), 319, 398
 - Architektur, 399
- Security und Usability, 103, 533
- Security Usability Fundamentals, 549
- Security vs. Safety, 615
- Security vs. Usability, 616
- Security-Infrastruktur, 232
- Security-Policy, 185
- SEH, 33
- seL4, 133, 142, 611
- SELinux, 158
- Semantik, 431
- Semantik einer Aktion, 97
- Semantik von Integer-Operation, 478
- semantische Kluft, 535
- semantische Kontrolle, 104
- separation of Concerns, 581
- Sequence Numbers, 41
- Serialisierung von Feldern
 - Schutz durch transient, 192
- Server zu Server Requests
 - Delegation, 340
- Server-Identity, 217
- Serversicherheit
 - Architektur der Software, 169
 - Authentisierung eines Requests, 163
 - Datenbanktabelle, 171
 - Durch Proxy eingeschränkter Zugriff auf Daten eines Services, 172
 - Ein Prozess pro Service, 167
 - Filterressourcen, 170
 - Grundkonzepte, 161
 - Identität, 161
 - multi-threaded Server, 166
 - Objektkapselung, 170
 - privilegierte Operation, 162
 - Registry, Domain, Realm, 162
 - Schadensbegrenzung durch Isolation, 169
 - sicherheitsrelevante
 - Architekturmerkmale, 165
 - System initialisieren, 170
- Servicedelegation in SOA, 411
- serviceorientierte Architekturen, 234
- setuid, 175
- „Shared-State“-Konzept, 316
- Shared State Multi-Threading, 463, 467
 - Kritikpunkte, 467
- sichere Delegation an Dritte, 308
- sichere Erweiterung von Software, 457
- sichere Exception Handler, 33
- sichere Listenverwaltung, 33
- sichere Transaktionen mit FINREAD, 370
- sichere User-Interfaces, 557
- sicherer Desktop, 523

- Sicherheit der Applikation, 477
- Sicherheit und Zuverlässigkeit, 126
- Sicherheit von Servern, 160
- Sicherheitsanalyse, 94, 441
 - formale, 580
 - Kernkonzepte, 582
 - Verhaltensbasiert, 585
- Sicherheitsanalyse einer Software, 449
- Sicherheitsarchitektur bei loser Kopplung, 238
- Sicherheitsarchitektur von
 - HEALTH@NET, 609
- Sicherheitsbaustein, 449
- Sicherheitsbausteine, 448
 - Modellierung, 575
- Sicherheitsdialog, 549, 554
- Sicherheitsdomänen und Layers, 511
- Sicherheitsentscheidung, 550
- Sicherheitsexperte, 551
- Sicherheitsfunktionen bei
 - Server-Applikationen, 251
- Sicherheitsindustrie, 417
- Sicherheitskonzept des Browsers, 510
- Sicherheitskonzepte
 - Browserarchitektur, 504
 - Netscape/Mozilla Browserlinie, 512
- Sicherheitslücke im Cisco-Portal, 239
- Sicherheitsmodell für Firmen, 117
- Sicherheitsplan, 463
- Sicherheits-Policy, 253
- Sicherheitsprobleme der Modus-
 - Architektur, 128
- sicherheitsrelevante Annotationen, 276
- Sicherheitsumfeld, 139
- Sicherheitszone, 95, 437
- Sicherheitszonen, 234
- Signature Block File, 369
- Signature File, 369
- signierte Token, 196
- Signierung von Jars, 203
- Single-Source-Transformationen, 480
- Singularity, 134, 451
 - Architektur, 146
 - Compound Principal, 150
 - Eigenschaften, 146
 - Exchange Heap, 147
 - Kontrakt, 149
 - Manifest, 149
 - Runtime, 147
 - Transport von Objekten, 147
 - Typ TRef in Sing#, 148
- Siteminder, 339
- SkyNet-Virus, 545
- Smartcard, 116, 189
- Bedrohungsmodell, 372
- Parteienszenario, 373
- Terminal, 373
- Smartphones, 92
- SOA, 234
- SOA Security, 411
- SOAP Security, 396
 - Envelope, 396
- SOAP-Fehlercodes, 52, 398
- Social-Engineering-Techniken, 62
- Socket API Calls, 161
- Socket-Design, 314
- Socket-Server, 314
- Soft-Typing, 303
- Software Extension, 457
- Softwarearchitektur und Sicherheit, 432
- softwarebasierte Isolation, 145
- Software-Interrupt, 127
- Softwarekomponenten, 232
- Solitaire, 119
- sozialer Trend, 551
- Speicherfehler, 190
- Speichern von Passwörtern, 240
- Speichersicherheit, memory safety
- spezielle Probleme der Front-End-
 - Applikationen, 288
- Sprachbasierte Sicherheit, 187
- Sprache E, 450, 522
- Spring Security, 295
 - Authentisierungs-Objekte, 298
 - Authorisierung, 299
 - Interceptor Pattern, 299
 - Manipulation von Authentisierungs-Objekten, 299
 - Philosophie, 295
 - Provider Pattern, 300
- SQL-Injection, 20
- SSO-Token, 338
 - durch symmetrische Keys, 304
 - End-to-End-Fluss, 262
- Stack-Walk, 206, 445, 582
- Stack-Walk-Mechanismus von Java, 201
- State halten, 476
- State-Attacke im Netzwerk, 477
- State-Machines, 382
- Status-Quo-Bias, 33
- Stellvertretungen, 99
- STI, 398
- Strongnames, 203
- Struts, 278
 - ActionServlet, 279
 - Aufbau einer Struts-Applikation, 280
 - Singletons, 289

SU, 175
 Credentials, 176
 SELinux, 176
 Subject, 286
 Subject Object in Token-Form, 340
 Subject-Erstellung mithilfe eines
 LTPA-Token, 335
 Subjective Validation Bias, 550
 Subjective-Expected-Utility-Theorie,
 552
 Suche nach Schwachstellen, 474
 Superstition Based, 552
 Surf Jacking, 23
 SWORD4J, 223
 Symbian-OS
 DLL-Labels, 202
 signierte Software, 378
 Symbolic-Link, 36
 Symbolische Referenzen, 507
 synchronized, 448
 System Context, 321
 System-Call-Interface, 441
 Systemdialoge, 559
 Systrace, 159

T

Tabellenorganisation, 243
 Tags, 153
 Tainted Strings, 50
 Tainting, 153, 444, 512, 514
 Tainting-Mechanismus, 49
 Take/Grant Systeme, 568
 Take/Grant-Modelle, 570, 589
 Taming, 448, 526
 Tausch von Passwörtern, 551
 technischer Entmündigungsprozess, 618
 Template/Hook Pattern, 293
 Testinstrumente, 60
 Theorien zur Entscheidungsfindung, 553
 Threat-Modelling, 126
 Ticket, 462
 Ticket Granting Tickets, 308
 Time-of-Check-to-Time-of-Use, 37, 102,
 220
 TLS-PSK, 536, 554
 TOC2TOU, 102, 175
 Fehler, 496
 Mozilla, 440
 Permission Checks, 220
 Threads, 316
 Token
 als Kontextinformation, 461
 Zusatzinformation, 286

totale Kooperation in einfachen
 Take/Grant Modellen, 572
 Trademarking, 303
 Trampoline Code, 34
 Transaktionssignierung, 370
 Transport- und Security-Aspekte, 238
 Transportschicht, 248
 Tree Interface, 441
 Trichterprinzip, 269
 Trust Association Interceptor (TAI), 338
 Trust-Boundary, 528
 Trusted Base, 106
 Trusted Code, 369
 Trusted Code Base, 204
 Trusted Computing Base, 479, 610
 Trusted Path, 104, 558
 Trusted Third Party, 261, 262, 398
 Trust-Store eines Servers, 339
 Twiki Sicherheitslücke, 511
 Type Spaces, 149
 Typen und Objekte in SE-Linux, 176
 Typsicherheit, 184, 187
 gebrochen, 189

U

Überaggregation, 477
 Übergang vom User zum Kernel Mode,
 127
 Überlagerung von Dialogen, 559
 Übersicht der häufigsten Attacken, 13
 Übertragung der Userrechte an die
 Applikation, 501
 Überwachungsvariablen auf dem Stack,
 32
 Ubiquitous Computing, 115
 UDDI, 403
 Umstellung auf Java 2 Security, 218
 Umwandlung einer Designation in eine
 Autorität, 524
 Unicode-Codepoints, 26
 universelle Maschine, 419
 unklare Prozesse, 539
 Unsichere Verwendung von globalen Files,
 499
 URLs bypass security checks, 492
 Usability
 Active Authorization, 558
 Expressiveness, 558
 Foresight, 558
 Identifiability, 558
 Relevant Boundaries, 558
 Revocability, 558
 Self-Awareness, 558

- Trusted Path:, 558
- Visibility, 558
- Usability, 111
 - Fehlen einer Information, 541
 - ist für alle Beteiligten, 534
 - lokale Namen und ihre Bedeutung, 536
 - Path of Least Resistance, 558
 - Serverzertifikate in Firefox 3.0, 555
 - Verständnisprüfung, 554
 - von Sicherheitsmaßnahmen, 443
- Usability und Security als strikte Gegensätze, 534
- User Conceptual Models, 103
- userbasierte Policies mit JAAS, 207
- User-Centered-Design, 554
- User-Interaktion, 548

V

- Vat, 471
- Verantwortung, 418
- Verantwortung für Systeme, 419
- Vereinzelungsanlagen, 551
- Verfügbarkeit, 143
- Vergleich mit C++, 192
- Verhalten in einer Menge, 552
- Verhaltensmaximierung, 591
- Verhaltensprädikate, 587
- Verhaltensregeln, 590
- verified by VISA, 541
- Verlagerung von Zugriffskontrolle „nach vorne“, 247
- Vermeidung von Seiteneffekten, 466
- Verringerung der Anzahl der Requests, 28
- Verstehen, 554
- verteilte Enterprise-Applikation, 230
- vertikale vs. horizontale Sicherheitskonfiguration, 603
- Vertrag, 462
- Vertrauenszone, 231
- Verwaltung des Runtime-Systems, 350
- Verwaltungsobjekte, 354
- Vier-Augen-Prinzip, 98
- Virens Scanner, 417
- Virtual World, 452
- Virtualisierung, 456
 - auf Basis des seL4 Microkernels, 142
 - Grenzen der Sicherheit, 144
 - Logical Partitions (LPARs), 144
 - Tips für den Einsatz von VMs, 145
 - Verfügbarkeit, 144
- Virtualization for embedded Systems, 142

- virtuelle Organisation, 305
- Vista
 - CBT-Hooks, 156
 - Integritätslevel, 155
 - UAC, 154
 - User Access Control, 154
- Visual Spoofing, 26
- Vollmacht, 462
- vorgelagerte Authentisierung, 337
 - durch SSO-Server, 338
 - Signaturlösung, 339

W

- WAF, 58
 - als zusätzliche Verteidigungslinie, 58
 - Vergleich der Produkte, 58
- WAS, 318
- Watermark Fallacy, 552
- Web Application Firewall (WAF)
 - Einsatzbedingungen, 58
- Web Server, 280
- Web2.0, 560
 - Mashups, 69
 - Techniken, 69
- Web-Applikationen
 - Authentisierung und/Autorisierung, 280
- WebSeal, 339
- Web-Services, 385
 - Bedrohungen und Risiken, 406
 - Document-Key-Check, 397
 - Document-Key-Validation, 397
 - Key-Identity-Validation, 397
 - non-repudiation, 386
 - Replay-Attacke, 397
 - über REST, 407
- Websphere-Application-Server, 318
- Web-Trojaner, 21
- Wechsel von Sicherheitsmodi, 23
- Wechsel zwischen den Modi, 421
- Weitergabekennung, 257
- weltweit eindeutige Inhabernamen, 539
- Whitelist, 420
- Willensbeweis, 424
- Wissensmaximierung, 591
- Wissensprädikate, 587
- World of Warcraft, 444
- WoW, 444
- Wrapper, 134, 474
- Write-Down, 155
- WSDL-Interface-Beschreibung, 53
- WSS4J, 408

WS-SecureConversation, 402
WS-Security, 396, 397
 Attacken, 406
 Einbindung in Applikationen, 407
 Java Framework (WSS4J), 407
WS-Trust, 399
 Anfordern und Prüfen von Token, 400
 Security Token Request, 400, 401
 Security Token Response, 401
 Szenarien, 401

X

XACML, 197, 263, 403
XML
 Appliances, 398
 kanonische Form, 391
 Signaturen, 391
 Signieren von Dokumenten, 392
 verschiedene physische
 Repräsentationen, 395
 Verschlüsselung, 392
XMLHttpRequest, 74, 75, 76
 Einsatz von vorgelagerter
 Authentisierung, 82
XSS Cheat sheets, 57

Z

Zeitdruck, 552
Zero-Day-Attacken, 117
Zero-Risk-Bias, 551

Zerstörung der OO-Kapselung durch
 Threads, 471
Zertifikate
 Usability, 536
Zertifikatsketten, 379
Zonen innerhalb der Software, 439
Zufallszahlengeneratoren, 139
Zugriff auf die SIM-Karte, 194
Zugriff auf Objekte, 95
Zugriff nur über erhaltene Referenzen,
 137
Zugriffskontrolle, 423, 431
 alternative Methoden, 217
 auf Basis eines Subjects, 213
 auf Basis von ACMs, 568
 Autoritäten, 197
 Basismechanismen, 431
 deklarativ und programmatisch, 272
 Gefahr selbst gebauter Autorisierung,
 290
 Granularität eines „Zugriffs“, 291
 in der Datenbank, 270
 in lokalen und verteilten Systemen,
 428
 instanzbasiert, Beispiel, 273
 Rechtemenge vergrößern, 217
 rollenbasiert, deklarativ – Beispiel,
 262
 über Fassaden, 271
 via Interceptor, 268
Zugriffsrechte bei Impersonifikation, 216
Zuverlässigkeit und Verfügbarkeit, 620