

Andreas Wiegenstein, Markus Schumacher,
Sebastian Schinzel, Frederik Weidemann

Sichere ABAP™-Programmierung




Galileo Press

Bonn • Boston

Auf einen Blick

1	Einleitung	27
---	------------------	----

Teil I Grundlagen

2	ABAP-Entwicklung aus Sicherheitssicht	53
3	Methoden und Werkzeuge zur Entwicklung sicherer Software	65

Teil II Anwendung und Praxis

4	Sichere Programmierung	89
5	Sichere Programmierung mit ABAP	135
6	Sichere Webprogrammierung mit ABAP	213
7	Sichere Programmierung in den ABAP-Technologien ...	285
8	Risiken in Business-Szenarien	319
9	Schlussfolgerungen und Ausblick	331

Teil III Anhang

A	Checklisten und Übersichten	339
B	Literatur- und Quellenverzeichnis	351
C	Die Autoren	355

Inhalt

Vorworte	17
----------------	----

1 Einleitung 27

1.1	Evolution von Sicherheitsrisiken	28
1.1.1	Angriffe auf die Applikation	28
1.1.2	Symptome versus Ursachen	30
1.1.3	Risiken durch ABAP-Code	32
1.2	(Un-)Sichere ABAP-Programme	34
1.2.1	Top Ten der falschen Annahmen	34
1.2.2	Reality Check	35
1.3	Ziel des Buches	40
1.3.1	Aufbau des Buches	42
1.3.2	Verwendete Konventionen	45
1.3.3	ABAP-Terminologie	45
1.3.4	Zielgruppen	46
1.4	Danksagung	50

Teil I Grundlagen

2 ABAP-Entwicklung aus Sicherheitssicht 53

2.1	Charakteristika von ABAP	54
2.1.1	Mächtigkeit von ABAP und ABAP-Entwicklern	55
2.1.2	Angriffsoberfläche von ABAP-Programmen	56
2.1.3	Datenaustausch bei SAP-Systemen	58
2.1.4	Sicherheitsdefekte in ABAP	61
2.2	ABAP-Entwicklungsprozesse	62
2.3	Besonderheiten bei externer Entwicklung	64

3 Methoden und Werkzeuge zur Entwicklung sicherer Software 65

3.1	Reifegrad von Sicherheitsprozessen	65
3.2	Spezifikation	68
3.2.1	De-facto-Standards	70
3.2.2	Grundschutzbaustein SAP	71
3.2.3	Individuelle Best Practices	72

3.2.4	Schulungen	73
3.2.5	Spezifisches Threat Modeling	73
3.3	Architektur und Design	74
3.3.1	Bewährte Designs	75
3.3.2	Dokumentierte Angriffsoberfläche	76
3.4	Implementierung und Programmierung	77
3.4.1	Training für Entwickler und QA-Teams	77
3.4.2	Code-Audit	78
3.4.3	Code Inspector	83
3.5	Test	84
3.6	Betrieb und Wartung	86

Teil II Anwendung und Praxis

4	Sichere Programmierung	89
4.1	Ursachen von Sicherheitsproblemen	93
4.1.1	Softwareentwicklung – Theorie vs. Praxis	93
4.1.2	Unwissen, Zeitmangel und technologischer Fortschritt	95
4.2	Organisatorische Maßnahmen	97
4.2.1	Erfassung des Ist-Zustands der Softwaresicherheit	98
4.2.2	Prozess für sichere Software etablieren	99
4.3	Sicherheitsprinzipien in der Softwareentwicklung	101
4.3.1	Prinzip #1 – Sicherheit als Priorität	102
4.3.2	Prinzip #2 – Risikobewusstsein	103
4.3.3	Prinzip #3 – Denken wie ein Angreifer	105
4.3.4	Prinzip #4 – Angreifer mit internem Wissen	106
4.3.5	Prinzip #5 – Prüfung aller Eingabewerte	106
4.3.6	Prinzip #6 – Reaktion auf alle Fehler	108
4.3.7	Prinzip #7 – Mehrschichtiges Schutzkonzept	110
4.3.8	Prinzip #8 – Möglichst kleine Angriffsoberfläche	111
4.3.9	Prinzip #9 – Überprüfung der Annahmen	112
4.3.10	Prinzip #10 – Handeln nach Standards	112
4.3.11	Prinzip #11 – Ständige Erweiterung des Wissens	114

4.4	Filterung und Validierung von Benutzereingaben	114
4.4.1	Repräsentation von Daten in Rechnersystemen	115
4.4.2	Validierung von Benutzereingaben	116
4.4.3	Behandlung unerwarteter Daten	119
4.4.4	Typische Fehler bei der Validierung mit Filtern	121
4.5	Encodierung von Ausgaben	123
4.5.1	Encodierungsprobleme	124
4.5.2	Encodierung von Daten	126
4.5.3	Typische Fehler bei der Encodierung	128
4.6	Indirektion	130
4.7	Checkliste für sichere Programmierung	132
5	Sichere Programmierung mit ABAP	135
5.1	Fehlende Berechtigungsprüfungen bei Transaktionen	138
5.1.1	Anatomie der Schwachstelle	138
5.1.2	Risiko	138
5.1.3	Maßnahmen	139
5.1.4	Selbsttest	143
5.2	Hintertüren – hart codierte Berechtigungen	143
5.2.1	Anatomie der Schwachstelle	144
5.2.2	Risiko	146
5.2.3	Maßnahmen	146
5.2.4	Selbsttest	146
5.3	Fehlende Berechtigungsprüfungen in RFC-fähigen Funktionen	148
5.3.1	Anatomie der Schwachstelle	149
5.3.2	Risiko	150
5.3.3	Maßnahmen	150
5.3.4	Selbsttest	151
5.4	Debug-Code in Assert Statements	151
5.4.1	Anatomie der Schwachstelle	153
5.4.2	Risiko	154
5.4.3	Maßnahmen	155
5.4.4	Selbsttest	155
5.5	Generischer und dynamischer ABAP-Code	156
5.5.1	Anatomie der Schwachstelle	157
5.5.2	Risiko	164
5.5.3	Maßnahmen	165

5.5.4	Selbsttest	165
5.6	Generische Funktionsaufrufe	166
5.6.1	Anatomie der Schwachstelle	167
5.6.2	Risiko	168
5.6.3	Maßnahmen	168
5.6.4	Selbsttest	168
5.7	Generische Reports (ABAP Command Injection)	169
5.7.1	Anatomie der Schwachstelle	170
5.7.2	Risiko	170
5.7.3	Maßnahmen	172
5.7.4	Selbsttest	172
5.8	SQL-Injection	173
5.8.1	Anatomie der Schwachstelle	176
5.8.2	Risiko	181
5.8.3	Maßnahmen	182
5.8.4	Selbsttest	184
5.9	Directory Traversal	185
5.9.1	Anatomie der Schwachstelle	187
5.9.2	Risiko	189
5.9.3	Maßnahmen	189
5.9.4	Selbsttest	190
5.10	Aufrufe in den Kernel	191
5.10.1	Anatomie der Schwachstelle	193
5.10.2	Risiko	196
5.10.3	Maßnahmen	197
5.10.4	Selbsttest	197
5.11	System Command Injection und System Command Execution	198
5.11.1	Anatomie der Schwachstelle	198
5.11.2	Risiko	206
5.11.3	Maßnahmen	207
5.11.4	Selbsttest	208
5.12	Checkliste für sichere ABAP-Programme	209

6 Sichere Webprogrammierung mit ABAP 213

6.1	Probleme von browserbasierten User Interfaces	215
6.1.1	Informationssicherheit	219
6.1.2	Berechtigungen	220
6.1.3	Integrität	221
6.1.4	Funktionsumfang	221

6.2	Sicherheitslücken in Web-Frontends	223
6.2.1	Günstiger Webshop	223
6.2.2	Verstecktes Passwort	225
6.2.3	Vermeintlicher Schutz vor Manipulation	226
6.3	Cross-Site Scripting	226
6.3.1	Anatomie der Schwachstelle	228
6.3.2	Risiko	234
6.3.3	Maßnahmen	236
6.3.4	Selbsttest	244
6.4	Cross-Site Request Forgery	246
6.4.1	Anatomie der Schwachstelle	248
6.4.2	Risiko	253
6.4.3	Maßnahmen	254
6.4.4	Selbsttest	259
6.5	Forceful Browsing	259
6.5.1	Anatomie der Schwachstelle	261
6.5.2	Risiko	265
6.5.3	Maßnahmen	267
6.5.4	Selbsttest	269
6.6	Phishing	270
6.6.1	Anatomie der Schwachstelle	271
6.6.2	Risiko	274
6.6.3	Maßnahmen	276
6.6.4	Selbsttest	277
6.7	HTTP Response Tampering	279
6.7.1	Anatomie der Schwachstelle	281
6.7.2	Risiko	282
6.7.3	Maßnahmen	282
6.7.4	Selbsttest	283
6.8	Checkliste für UI-Programmierung	283

7 Sichere Programmierung in den ABAP-Technologien 285

7.1	Verarbeitung von Dateien	286
7.1.1	Zugriff auf Dateien	287
7.1.2	Verarbeitung von Dateiinhalten	288
7.1.3	Dateiaustausch zwischen Client und Server	289
7.1.4	Zusammenfassung	290

7.2	Datenbankzugriffe	291
7.2.1	Datenbankabfragen mit Open SQL	292
7.2.2	Datenbankabfragen mit Native SQL	293
7.2.3	Validierung der Daten	293
7.2.4	Zusammenfassung	295
7.3	SAP GUI-Anwendungen	296
7.3.1	Ablauf der Interaktion zwischen SAP GUI und Backend	296
7.3.2	Varianten des SAP GUI	298
7.3.3	Maßnahmen im Backend	299
7.3.4	Zusammenfassung	300
7.4	SAP NetWeaver Application Server ABAP	301
7.4.1	Funktionsweise des AS ABAP	301
7.4.2	Hilfsmittel zur sicheren Entwicklung	302
7.4.3	Zusammenfassung	303
7.5	Business Server Pages	303
7.5.1	Entwicklung von BSP-Anwendungen	304
7.5.2	Absicherung der Interaktion	306
7.5.3	Zusammenfassung	308
7.6	Internet Transaction Server	309
7.6.1	Entwicklung von Webanwendungen mit dem ITS	309
7.6.2	Sichere Entwicklung mit dem ITS	310
7.6.3	Zusammenfassung	311
7.7	Web Dynpro ABAP	312
7.7.1	Entwicklung mit Web Dynpro	312
7.7.2	Sichere Entwicklung mit Web Dynpro	313
7.7.3	Zusammenfassung	314
7.8	Anbindung indirekter User Interfaces und externer Systeme	314
7.8.1	Sichere Anbindung externer User Interfaces	315
7.8.2	Sichere Anbindung externer Systeme	315
7.8.3	Zusammenfassung	316
7.9	Checkliste für SAP-Technologien	317

8 Risiken in Business-Szenarien 319

8.1	E-Recruitment	320
8.1.1	Angriffsmotive	321
8.1.2	Angriffsszenarien	322
8.1.3	Maßnahmen	324

8.2	Employee Self-Services	325
8.2.1	Angriffsmotive	326
8.2.2	Angriffsszenarien	326
8.2.3	Maßnahmen	327
8.3	Customer Relationship Management	328
8.3.1	Angriffsmotive	328
8.3.2	Angriffsszenarien	329
8.3.3	Maßnahmen	330
9	Schlussfolgerungen und Ausblick	331
9.1	Schlussfolgerungen	331
9.2	Ausblick	333
9.3	Was Sie mitnehmen sollten	335

Teil III Anhang

A	Checklisten und Übersichten	339
B	Literatur- und Quellenverzeichnis	351
C	Die Autoren	355
	Index	359

Keine Programmiersprache ist frei von Sicherheitsproblemen. ABAP bildet dabei keine Ausnahme. Besondere technische Merkmale einer Programmiersprache können spezifische Sicherheitsprobleme mit sich bringen, die in anderen Sprachen so nicht vorzufinden sind. In diesem Kapitel wird auf die typischen Sicherheitsprobleme von ABAP-Programmen eingegangen.

5 Sichere Programmierung mit ABAP

Mit diesem Kapitel tauchen wir in die technischen Tiefen von sicherer ABAP-Programmierung ein. Die Probleme, die hier erklärt werden, sind dabei grundsätzlicher Art, die beschriebenen unsicheren Programmier-techniken können in praktisch jeder Art von ABAP-Programm auftreten: in Transaktionen, in (RFC-fähigen) Funktionsbausteinen, in Business Server Pages, Webservices, Batch-Verarbeitungsprogrammen oder Web-Dynpro-ABAP-Applikationen. Dieses Kapitel sollten Sie daher sehr genau studieren, insbesondere wenn Sie Entwickler oder Sicherheitstester sind.

Ein charakteristisches Merkmal zahlreicher Sicherheitsprobleme ist die Kombination eines potenziell gefährlichen ABAP-Befehls mit externen Daten, die von diesem Befehl verarbeitet werden. Dabei ist völlig gleichgültig, woher die Eingaben kommen, ob aus Dateien, Datenbanktabellen, Eingabefeldern in Bildschirmmasken, Aufrufen über die RFC-Schnittstelle oder über Webservices: Schadhafte Daten können prinzipiell bestimmte ABAP-Befehle manipulieren und ihre Funktionalität beeinflussen. Ein Angreifer kann durch gezielt manipulierte Daten ein ABAP-Programm dazu bringen, Funktionen auszuführen, die der Entwickler gar nicht wissentlich vorgesehen hatte. Der Schaden ist damit im wahrsten Sinne des Wortes vorprogrammiert.

Alle Eingaben
können gefährliche
Daten enthalten

Andere Merkmale in diesem Kontext sind Sicherheitsfunktionen, die nicht oder falsch verwendet werden. Fehlende Berechtigungsprüfungen im Zusammenhang mit dem Aufruf von kritischer Funktionalität wären ein praktisches Beispiel. Diese Probleme resultieren häufig aus unvollständigen Spezifikationen und Anforderungen an das ABAP-Programm, aber auch aus mangelndem Wissen über die Konsequenzen. Auch hier entstanden im Laufe der Jahre einige interessante Anekdoten.

Sicherheits-
funktionen können
nicht oder falsch
verwendet werden

- 【★】 Beim einem Audit eines CRM-Add-ons wurden zwei ABAP-Funktionen untersucht. Die eine diente dazu, Kreditkartennummern zu verschlüsseln und die verschlüsselten Daten anzuzeigen, die andere wurde zur Entschlüsselung verwendet. Der Kunde wollte eine Überprüfung der Stärke des Verschlüsselungsalgorithmus. Bereits nach kurzer Zeit wurde deutlich, dass der Hersteller einen allgemein anerkannten und damit kryptografisch sicheren Algorithmus benutzt hatte. Somit war klar, dass weder ein Angriff durch Ausprobieren (Brute Force) noch eine mathematische Analyse Sinn hatte. Denn anerkannte Algorithmen sind aus Sicht des Entwicklers als sicher zu betrachten, bis ein Kryptoexperte eine gegenteilige Nachricht veröffentlicht.

Daraufhin wurden die beiden ABAP-Funktionen genauer betrachtet. Die Verschlüsselung der Kreditkartennummer konnte von allen Benutzern aufgerufen werden. Die Entschlüsselung erforderte allerdings eine entsprechende Berechtigung des angemeldeten Benutzers, die im Code geprüft wurde. Das Berechtigungswesen gehört zum SAP-Standard, somit bot sich auch hier keine grundsätzliche Angriffsmöglichkeit. Hätten Sie an dieser Stelle die Analyse mit den vorgegebenen Informationen abgebrochen und das Konzept als sicher erklärt?

Sicherheits-
probleme werden
oft übersehen

Wenn ja, hätten Sie ein schwerwiegendes Problem übersehen. Das Problem liegt darin, dass die Verschlüsselung eines Kreditkartendatums ohne Berechtigung aufgerufen werden konnte und dass gleichzeitig der verschlüsselte Text angezeigt wurde. Das sieht auf den ersten Blick nicht bedenklich aus. Denken Sie kurz darüber nach: Es kann höchstens 10^{16} Kreditkartennummern geben, da diese maximal aus 16 Ziffern bestehen. Die Menge der verfügbaren Kreditkartennummern wird allerdings algorithmisch noch weiter eingeschränkt, da die Nummern zusätzlich mit Prüfsummen codiert werden. Diese können mit dem sogenannten Luhn-Check verifiziert werden (siehe ISO, *Identification cards – Identification of issuers – Part 1: Numbering system*, 2006). Außerdem sind einige Nummernbereiche nicht vergeben.

Lassen Sie nun der Reihe nach alle möglichen Kreditkartennummern mit dem bekannten Algorithmus verschlüsseln, können Sie mit vertretbarem Aufwand eine sogenannte Rainbow Table erzeugen: Dabei handelt es sich um eine Tabelle, die alle Paare von Kreditkartennummer und zugehörigem Verschlüsselungstext enthält. Mit der Tabelle kann nun die Verschlüsselung leicht umgekehrt werden. Ein Angreifer braucht nur nach dem Verschlüsselungstext zu schauen und kann dann die zugehörige Kreditkartennummer in der Tabelle nachschlagen – ohne den Schlüssel zu

kennen. Hier wurde aufgrund eines Denkfehlers an einer kritischen Stelle die Berechtigungsprüfung vergessen. Mit fatalen Folgen.

Ein anderer Fall spielte sich im Jahr 2008 ab. Es gab den Auftrag, ein von Beratern entwickeltes SRM-Modul auf gängige Fehler hin zu überprüfen. Das Modul war als Webanwendung auf der Basis von Business Server Pages entwickelt worden. Die BSP-Anwendung rief aber im Hintergrund ABAP-Funktionen auf, die die Eingaben verarbeiteten und das Ergebnis an die Webanwendung zurücklieferten. Fast immer rufen moderne Frontends und Schnittstellen im Hintergrund bewährte alte Funktionsbausteine oder Reports auf, die die Daten verarbeiten. Jedenfalls hatte die Webanwendung eine Funktionalität, die dem angemeldeten Lieferanten seine Preiskonditionen bei dem Kunden anzeigte.

[*]

Diese Liste konnte für bestimmte Artikelgruppen über eine Auswahlliste in der BSP-Anwendung gefiltert werden. Die Auswahl wurde dann an die BSP-Anwendung geschickt. Letzten Endes wurde dieser Wert in einer Open-SQL-Anweisung verarbeitet, mit der die Preise aus der Datenbank ausgelesen wurden. Genauer gesagt, war dieser Wert Teil eines dynamisch erzeugten Auswahlfilters. Das heißt, wenn sich der Wert änderte, änderte sich das Ergebnis der Datenbankabfrage.

Nun konnte der Wert über das Frontend aber nicht nur auf eine andere Artikelgruppe, sondern auf jede beliebige Zeichenkette verändert werden. Dadurch konnte zum Nachweis der Sicherheitslücke der gesamte Filter des SQL-Statements verändert bzw. kontrolliert werden, und es war möglich, durch eine spezielle Sequenz von nur wenigen Zeichen den Filter so zu manipulieren, dass er gar nicht mehr filterte, sondern vielmehr alle Datensätze anzeigte, auch die der anderen Lieferanten.

Dieses unerwünschte Feature ist für Lieferanten, die unbedingt einen Auftrag haben wollen, sicher hoch interessant. Technisch handelt es sich hierbei um eine SQL-Injection-Attacke, die wir in Abschnitt 5.8 im Detail erklären werden.

Sicherheitslücken können zu ungewünschter Funktionalität führen

Bei den meisten Sicherheitsdefekten steckt der Teufel im (technischen) Detail. Software ist unglaublich komplex, und es sind während der Entwicklung nicht immer alle Anwendungsszenarien vorhersehbar. Daher ist es notwendig, die typischen Probleme zu verstehen und die eigenen Anwendungen so robust wie möglich zu bauen, sodass sie einem Angriff widerstehen können.

5.1 Fehlende Berechtigungsprüfungen bei Transaktionen

ABAP-Dialogprogramme können programmspezifisch mit Berechtigungen geschützt werden, sie können allerdings auch im SAP-Applikations-server durch Transaktionen gekapselt werden. Jede Transaktion hat einen eindeutigen Bezeichner, den sogenannten Transaktionscode. Durch die Eingabe dieses Bezeichners im SAP-Hauptmenü kann eine Transaktion durch den Benutzer direkt gestartet werden. Der Transaktionscode SE80 startet beispielsweise die ABAP Workbench, die verschiedene Werkzeuge für die ABAP-Entwicklung enthält. Jede Transaktion kann mit einem Berechtigungsobjekt versehen werden, sodass nur die berechtigten Benutzer die Transaktion starten und ausführen können.

Über Berechtigungsobjekte lässt sich auf diese Weise der Zugriff auf kritische Funktionalität in SAP-Systemen einschränken. Es gibt noch eine Vielzahl anderer Ressourcen, die durch Berechtigungsobjekte geschützt werden können. In diesem Abschnitt wird erklärt, warum das Pflegen eines Berechtigungsobjektes bei einer Transaktion allein noch nicht unbedingt dazu führt, dass die Berechtigung auch wirklich in allen Fällen geprüft wird. Als Beispiel einer kritischen Funktionalität soll das bereits erwähnte Starten von Transaktionen herangezogen werden.

5.1.1 Anatomie der Schwachstelle

Nicht nur der Benutzer, sondern auch ABAP-Programme können SAP-Transaktionen starten. Hierzu dienen die ABAP-Befehle `CALL TRANSACTION` und `LEAVE TO TRANSACTION`.

Berechtigungs-konzepte greifen nur dann, wenn die Berechtigung geprüft wird

Während `LEAVE TO TRANSACTION` eine implizite Berechtigungsprüfung (`AUTHORITY-CHECK`) für den angemeldeten Benutzer durchführt, geschieht dies bei `CALL TRANSACTION` nicht. Daher können Transaktionen von ABAP-Programmen aus gestartet werden, für die der angemeldete Benutzer eigentlich keine Berechtigung hat.

5.1.2 Risiko

- [★] Ein gutes Beispiel für diese Problematik ist eine Kundenentwicklung im CRM-Kontext. In einem Projekt wurde eine Anwendung in einem frühen Stadium der Entwicklung untersucht, die autorisierten Benutzern den Zugriff auf ihre Kreditkartendaten gewähren sollte. In dieser Anwendung gab es verschiedene Buttons, die jeweils eine Transaktion zum Anlegen,

Ansehen und Ändern von Kreditkartendaten starten konnten. Allerdings wurde hierbei im ABAP-Coding keine Berechtigungsprüfung durchgeführt.

Obwohl alle Berechtigungsobjekte korrekt gepflegt waren und die Benutzer der CRM-Anwendung diese drei Transaktionen nicht direkt starten konnten, hatten sie über die Buttons in der Anwendung dennoch Zugriff, da der Entwickler vergessen hatte, im Code die Berechtigungen zu prüfen, die im System gepflegt waren.

Manipulation ist trotz Berechtigungskonzept möglich

Sie können sich vorstellen, dass die Auftraggeber dieses Audits ein wenig überrascht waren, als ihnen eine Liste der Kreditkartendaten all ihrer Kunden vorgelegt werden konnte. Viel schlimmer ist es, wenn solche kritischen Daten durch eine derartige Sicherheitslücke in die falschen Hände geraten.

5.1.3 Maßnahmen

Entwickler sollten immer eine programmatische Berechtigungsprüfung im Code implementieren, wenn sie den Befehl `CALL TRANSACTION` verwenden. Ebenso sollte bei der Ausführung kritischer Programme und beim Zugriff auf wichtige Ressourcen eine explizite Berechtigungsprüfung im Code erfolgen. Ohne eine programmatische Berechtigungsprüfung könnte das ABAP-Programm unberechtigten Benutzern Zugriff auf eingeschränkte Transaktionen und Ressourcen geben. Solche Berechtigungsprobleme führen, wie die vorangegangene Anekdote zeigt, in vielen Fällen zu einer Compliance-Verletzung, denn durch solche Probleme ist die Nachvollziehbarkeit von Benutzeraktionen im SAP-System nicht mehr gewährleistet.

Verwenden Sie immer explizite Berechtigungsprüfungen

Der Befehl `LEAVE TO TRANSACTION` prüft automatisch das Berechtigungsobjekt `S_TCODE`, das heißt die Startberechtigungsprüfung für Transaktionen. Darüber hinaus ist es möglich, ein weiteres Berechtigungsobjekt in Transaktion SE93 zu pflegen. Ist dieses zweite Berechtigungsobjekt gepflegt, wird dementsprechend eine zusätzliche Berechtigungsprüfung durchgeführt, wenn die Transaktion durch `LEAVE TO TRANSACTION` gestartet wird.

Dieser Mechanismus funktioniert für `CALL TRANSACTION` jedoch nicht. Es gibt jedoch die Möglichkeit, automatische Berechtigungsprüfungen auch für `CALL TRANSACTION` zu erwirken, indem sogenannte Transaktionspaare über Transaktion SE93 gepflegt werden. Abbildung 5.1 zeigt die Liste der Prüfungen im Fall von Transaktion SE80 an. Springen Sie zum Beispiel von Transaktion SE80 zu Transaktion SE16, wird hier eine Berechtigungsprüfung vorgenommen.

Exce.	gerufene Transaktion	Transaktionstext	Prüfmerkz	Meldungstyp
000	SCPR2		JA	X
000	SCPR20	Aktivierung von BC-Sets	JA	X
000	SCPR3	Anzeige und Pflege von BC-Sets	JA	X
000	SCRP_DPCHCK			
000	SCRP_ORG			
000	SCU0	Customizing Cross-System Viewer		
000	SCU3	Tabellenhistorie	JA	X
000	SCWB	Korrekturworkbench	JA	X
000	SDCC	Service Data Control Center	JA	X
000	SDOCU	Pflege von Doku-Strukturen	JA	X
000	SE01	Transport Organizer (Erw. Sicht)	JA	X
000	SE03	Transport Organizer Tools	JA	X
000	SE07	Statusanzeige Transportwesen		
000	SE09	Transport Organizer	JA	X
000	SE09_WDCONV			
000	SE10	Transport Organizer	JA	X
000	SE11	ABAP Dictionary Pflege	JA	X
000	SE11_OLD	ABAP Dictionary Pflege	JA	X
000	SE12	ABAP Dictionary Anzeige	JA	X
000	SE12_OLD	ABAP Dictionary Anzeige		
000	SE13	Speicher-Param. für Tabellen pflegen	JA	X
000	SE14	Utilities für Dictionary-Tabellen	JA	X
000	SE15	Dictionary-Infosystem	JA	X
000	SE16	Data Browser	JA	X
000	SE17	Allgemeine Tabellenanzeige	JA	X
000	SE18	BAdI-Builder - Definitionen	JA	X
000	SE19	BAdI-Builder - Implementierungen	JA	X

Abbildung 5.1 Pflege von Transaktionspaaren in Transaktion SE93

Um sich die Liste anzeigen zu lassen, müssen Sie in Transaktion SE93 zunächst eine Transaktion auswählen, in diesem Fall Transaktion SE80. Anschließend wird über das Menü **HILFSMITTEL • BERECHTIGUNG FÜR GERUFENE TRANSAKTIONEN** die Liste angezeigt. Um eine Berechtigungsprüfung zu erwirken, muss ein Eintrag angelegt werden, der die aufrufende und die gerufene Transaktion definiert (ein Transaktionspaar). Diese Einträge werden in der Tabelle `TDCCOUPLES` gespeichert.

Wurde ein solcher Eintrag korrekt angelegt, wird eine Berechtigungsprüfung genauso durchgeführt, wie dies bei `LEAVE TO TRANSACTION` der Fall ist. Gibt es diesen Eintrag nicht, wird überhaupt keine Berechtigungsprüfung ausgeführt. Dieser Ansatz erfordert daher eine exakte und vollständige Konfiguration für jede Transaktion, die aufgerufen wird. Der Raum für Fehler ist dementsprechend groß.

Eine Grundregel ist, dass sicherer Code nicht von korrekter Konfiguration abhängen darf, wenn es programmatische Alternativen gibt. Auf dieses Defense-in-Depth-Prinzip wurde bereits in Abschnitt 4.3, »Sicherheitsprinzipien in der Softwareentwicklung«, eingegangen. Daher muss eine explizite programmatische Berechtigungsprüfung für jeden CALL TRANSACTION-Befehl erfolgen. **[+]**

Ein anderer wichtiger Aspekt ist, dass Transaktionen oft mit zusätzlichen Parametern aufgerufen werden. Diese Parameter können die Mussfelder im ersten Bildschirm der Transaktion ausfüllen. Auf diese Weise kann der erste Bildschirm programmatisch übersprungen werden, sodass die Transaktion in einem vom ABAP-Programm definierten Bildschirm startet. Abhängig von der Transaktion, kann dies zusätzliche Berechtigungsprüfungen mit feinerer Granularität notwendig machen. Aus diesem Grund erfordern Transaktionen, die mit zusätzlichen Parametern aufgerufen werden, unter Umständen mehr als ein Berechtigungsobjekt und müssen unbedingt programmatisch geschützt werden.

Listing 5.1 zeigt ein Beispiel für eine programmatische Berechtigungsprüfung, die sicherstellt, dass der angemeldete Anwender die erforderliche Berechtigung hat, um Transaktion SM30 zu starten.

```
AUTHORITY-CHECK OBJECT 'S_TCODE'
                  ID 'TCD'
                  FIELD 'SM30'.
IF sy-subrc = 0.
  CALL TRANSACTION 'SM30'.
ENDIF.
```

Listing 5.1 Beispiel für programmatische Berechtigungsprüfung

Beachten Sie, dass nach Aufruf des Befehls AUTHORITY-CHECK OBJECT insbesondere der Return-Code in SY-SUBRC geprüft werden muss. Dieser muss auf 0 gesetzt sein, nur dann ist ein Absprung erlaubt. **[+]**

Die zu bevorzugende, da umfassendere Variante, um eine programmatische Berechtigungsprüfung durchzuführen, ist jedoch die Verwendung des Funktionsbausteins AUTHORITY_CHECK_TCODE. Dieser Funktionsbaustein reagiert nicht nur beim Start des Programms auf eine fehlende Berechtigung, sondern er bietet auch die Möglichkeit, dass nur die in Transaktion SE97 gepflegten NO-CHECK-Indikatoren den externen Aufruf aus einem anderen Transaktionskontext heraus erlauben. Dies wird dabei vom Funktionsbaustein und nicht vom Entwickler bestimmt. Weitere Details finden Sie in der Dokumentation des Funktionsbausteins.

Listing 5.2 zeigt ein Beispiel für die Verwendung von `AUTHORITY_CHECK_TCODE`.

```
CALL FUNCTION 'AUTHORITY_CHECK_TCODE'
  EXPORTING
    TCODE = 'SM30'
  EXCEPTIONS
    OK      = 0
    NOT_OK  = 2
    OTHERS  = 3.
IF sy-subrc = 0.
  CALL TRANSACTION 'SM30'.
ENDIF.
```

Listing 5.2 Berechtigungsprüfung mit `AUTHORITY_CHECK_TCODE`

Vollständige
Berechtigungs-
prüfungen tragen
zur Compliance bei

Implementieren Sie explizite Berechtigungsprüfungen auf Codeebene immer dann, wenn Sie Transaktionen aus ABAP-Programmen starten oder auf kritische Funktionen bzw. Ressourcen zugreifen. Das ist die beste Verteidigungslinie, um Ihre Geschäftsanwendungen vor Missbrauch zu schützen, denn Berechtigungsprüfungen auf Codeebene können zwei Aspekte sicher gewährleisten:

- ▶ Unvollständige oder fehlerhafte Transaktionsstartberechtigungen führen zu Compliance-Verstößen.
- ▶ Komplexe Berechtigungsprüfungen können auch für die parametrisierte Verwendung von `CALL TRANSACTION` hinreichend durchgeführt werden.

Hinweis

Wir haben in diesem Abschnitt darüber gesprochen, dass anwendungsspezifische Berechtigungsprüfungen in der Regel nicht automatisch vom SAP-System ausgeführt werden. In diesen Fällen muss der ABAP-Code die Berechtigungsprüfung durch den Befehl `AUTHORITY-CHECK` programmatisch ausführen (dieser Hinweis gilt daher auch für Abschnitt 5.3. »Fehlende Berechtigungsprüfungen in RFC-fähigen Funktionen«).

Zur sicheren Programmierung gehören aber auch Systemeinstellungen, die direkten Einfluss auf die programmierten Berechtigungsprüfungen haben. Es gibt im SAP-System die Möglichkeit, Berechtigungsprüfungen zu deaktivieren, auch wenn in ABAP ein `AUTHORITY-CHECK` aufgerufen wird:

- ▶ Transaktion `AUTH_SWITCH_OBJECTS` erlaubt das gezielte Abschalten von Berechtigungen bestimmter Bereiche, wie zum Beispiel der Basisadministration oder bestimmter Funktionen wie zum Beispiel Aufrufe von C-Funktionen aus ABAP heraus. Die Berechtigungsprüfungen können mit dieser Transaktion jedoch nur abgeschaltet werden, wenn der Profilparameter `auth/object_disabling_active` auf Y gesetzt ist. Wir empfehlen daher dringend, den Profilparameter `auth/object_disabling_active` auf N zu setzen.

- ▶ Ferner erlaubt der Profilparameter `auth/rfc_authority_check` die Berechtigungsprüfungen für das Berechtigungsobjekt `S_RFC` zu deaktivieren. Bitte achten Sie darauf, dass dieser Profilparameter nie den Wert 0 hat.
- ▶ Ein weiterer kritischer Profilparameter ist `auth/system_access_check_off`, der beispielsweise Berechtigungsprüfungen für Dateizugriffe und für Aufrufe von Kernel-Funktionen deaktiviert. Bitte achten Sie darauf, dass dieser Parameter den Wert 0 hat.
- ▶ Abschließend wollen wir noch den Profilparameter `auth/no_check_in_some_cases` erwähnen. Er muss für die Verwendung von Rollen und damit des Profilgenerators, Transaktion `PFCG`, auf Y stehen. Dadurch ist es allerdings über Transaktion `SU24` möglich, die Prüfung bestimmter Berechtigungsobjekte für einzelne Transaktionen zu deaktivieren. Stellen Sie sicher, dass nur in besonderen Ausnahmefällen Berechtigungsobjekte für eine Transaktion deaktiviert werden dürfen.

5.1.4 Selbsttest

Um unzureichende Berechtigungsprüfungen im Code zu erkennen, müssen Sie zunächst alle Zugriffe auf kritische Funktionen und Daten im Code identifizieren. Für das verwendete Beispiel in diesem Kapitel müssen Sie beispielsweise alle Vorkommen von `CALL TRANSACTION` im ABAP-Code finden. Sie können hierfür zum Beispiel den Code Inspector nutzen (siehe Abschnitt 3.4.3).

Den nächsten Schritt müssen Sie anschließend manuell erledigen: Für jeden Zugriff auf eine kritische Funktion bzw. auf kritische Daten muss eine explizite Berechtigungsprüfung im Code stattfinden. Prüfen Sie daher, ob der Befehl `AUTHORITY-CHECK OBJECT` mit dem korrekten Berechtigungsobjekt aufgerufen wird. Kontrollieren Sie zusätzlich, ob auch der Rückgabewert in `SY-SUBRC` korrekt geprüft wird.

[+]

5.2 Hintertüren – hart codierte Berechtigungen

In ABAP-Programmen werden Berechtigungsobjekte geprüft, um die Zugriffe auf kritische Funktionen und Ressourcen einzuschränken. Dies könnten zum Beispiel administrative Funktionen sein, die nicht dem gesamten Benutzerkreis der Anwendung zur Verfügung stehen dürfen. Berechtigungen werden im Allgemeinen über sogenannte Authority Checks realisiert (siehe Abschnitt 5.1). Das Berechtigungswesen wird zentral im ABAP-Standard zur Verfügung gestellt, um Überprüfungen in SAP-Anwendungen standardisiert durchführen zu können. Entwickler und Administratoren profitieren so von einem soliden Konzept, vielfältigen Verwaltungstools und umfangreicher Dokumentation.

Entwickler neigen jedoch manchmal dazu, ihre eigenen Berechtigungskonzepte über die Programmiersprache zu realisieren. Hierzu wird in der Regel lediglich der Name des angemeldeten Benutzers überprüft, um bestimmte Aktionen zu erlauben oder zu verweigern. ABAP ermöglicht das Auslesen des Benutzernamens über die Variablen `SY-UNAME` oder `SYST-UNAME`. Viele Programmierer erstellen mittels dieser Variablen ein einfach gestricktes, paralleles Berechtigungskonzept in ihrem ABAP-Programm, das häufig zu Testzwecken verwendet wird. Der Entwickler stellt damit sicher, dass sein Testcode nur ausgeführt wird, wenn er ihn selbst aufruft; bei anderen Benutzern geschieht einfach nichts.

Eigentlich ist dieses Verhalten auf den ersten Blick nicht kritisch. Schließlich wird nur dem Entwickler eine Debugging-Information angezeigt oder eine Testfunktion zur Verfügung gestellt. Allerdings ist dieses Verhalten in der Regel nicht dokumentiert. Kritisch wird das spätestens, wenn der Code produktiv gesetzt wird und immer noch Testfunktionen enthält. In diesem Fall kann der Entwickler vermutlich interne Zustände sehen, die er eigentlich nicht sehen können sollte. Dies hängt natürlich immer von der Logik ab, die der Entwickler implementiert hat. Diese Zustände könnten zum Beispiel vertrauliche Geschäftsdaten sein, die in Tabellen gespeichert sind. Ein Auditor oder Revisor wird diese Berechtigung nie finden können, ohne sich den Sourcecode anzusehen.

Entwickler können
Hintertüren
einbauen

Noch schlimmer ist es, wenn Entwickler absichtlich eine Funktion im ABAP-Code platzieren, die nur ausgelöst wird, wenn ein bestimmter Benutzer den Code aufruft. In jedem Fall ist solcher Code als Hintertür (Backdoor) einzustufen, denn hier wird versteckte Funktionalität eingebaut, die gar nicht vorhanden sein darf, aber von bestimmten Benutzern ausgelöst werden kann.

5.2.1 Anatomie der Schwachstelle

Wenn Berechtigungen individuell von Benutzernamen abgeleitet werden, bezeichnen wir dies als *proprietäres Berechtigungskonzept*. Dieses umgeht das SAP-Berechtigungswesen und verstößt somit gegen Best Practices. Vor allem ist ein proprietäres Berechtigungskonzept in aller Regel nicht auditierbar, da es nicht dokumentiert wird – und selbst wenn es dokumentiert ist, wird jeder Auditor davon Bauchschmerzen bekommen.

Das Beispiel in Listing 5.3 zeigt den in der Einleitung beschriebenen Fall eines hart codierten Benutzernamens.

```
METHOD backdoor1 .
  IF syst-uname = 'JOHNDOE'.
    * do something evil
```

```

ELSE.
    * normal behavior
ENDIF.
ENDMETHOD.

```

Listing 5.3 Hart codierte Benutzernamen

Das Beispiel zeigt, wie sich der Entwickler mit dem Benutzernamen JOHNDOE zusätzliche Informationen über eine eigene Berechtigungsprüfung anzeigen lässt. Die Ablauflogik wird daher abhängig vom aktuell an SAP angemeldeten Benutzer bestimmt. Dabei ist es unerheblich, ob der Zugriff über einen Browser, das SAP GUI, RFC oder einen sonstigen Weg erfolgt.

Abfragen auf hart codierte Benutzernamen können die Ablaufslogik verändern

Charakteristisch für diesen Fall ist, dass eine Abfrage eingebaut wird, die überprüft, ob SYST-UNAME gleich einem explizit codierten Benutzernamen ist. Natürlich beschränkt sich das Problem nicht nur auf einen Benutzernamen. Wird alternativ nach fünf oder mehr verschiedenen Benutzernamen gefragt, ändert das nichts an der Problematik. Natürlich ist es für Entwickler ziemlich lästig, jedes Mal das Programm umzuschreiben, wenn sich etwas an der Liste der hart codierten Benutzernamen ändert. Entsprechend haben wir auch schon Code gesehen, der Tests auf bestimmte Benutzernamen dynamisch löst. Wäre das nicht sicherheitskritisch, könnten solche Lösungen fast schon als elegant betrachtet werden.

Anstatt die Prüfung nur auf fest einprogrammierte Benutzer zu beschränken, helfen sich kreative Entwickler mit einer dynamischen Benutzerverwaltung, die über eine Kundentabelle konfiguriert werden kann. Das Beispiel in Listing 5.4 zeigt dies exemplarisch.

```

METHOD backdoor2 .
    SELECT flag INTO lv_flag FROM zusers
        WHERE username = sy-username.
    ENDSELECT.
    IF sy-subrc = 0.
    * backdoor goes here
    ENDIF.
ENDMETHOD.

```

Listing 5.4 Verwaltung von hart codierten Benutzernamen in Tabellen

Alle Benutzer werden hierbei in einer Kundentabelle hinterlegt und können so bequem konfiguriert werden. Damit lässt sich leicht bestimmen, wer Zugriff auf den versteckten Code erhält. Im Unterschied zum ersten Beispiel wird jetzt nicht mehr nur auf einen konstanten String, sondern auf eine Tabelle hin geprüft. Es gibt sicherlich noch weitere Alternativen,

Manchmal werden hart codierte Benutzernamen in Tabellen verwaltet

das Berechtigungskonzept von SAP kreativ zu erweitern – und in jeder Alternative handelt es sich immer um ein Sicherheitsproblem.

5.2.2 Risiko

Das Risiko ist maßgeblich von der versteckten Funktionalität abhängig, die durch die proprietäre Benutzerprüfung ausgeführt werden kann. Das kann von minimalen Auswirkungen bis zum größten anzunehmenden Schadensfall reichen.

- [★]** Bei einem Code-Audit im Jahr 2009 haben wir bei einem Auftraggeber eine produktive BSP-Anwendung gefunden, die hart codierte Benutzernamen prüfte. Für alle normalen Benutzer zeigte die Webseite einen leeren Inhalt. Doch für vier privilegierte Entwickler war ein Online-Formular zugänglich, mit dessen Hilfe beliebige SQL-Anweisungen an die Datenbank abgesetzt werden konnten. Ferner wurden auch die Ergebnisse dieser Anfragen in der BSP-Anwendung dargestellt, somit konnten diese vier Entwickler *sämtliche* Daten aus der produktiven SAP-Datenbank auslesen; und zwei der Entwickler waren externe Berater. Noch einmal langsam: Zwei externe Berater waren theoretisch in der Lage, über das Internet *sämtliche* Geschäftsdaten aus dem SAP-Backend zu ziehen. Und diese (Test-)Funktionalität ist niemandem bei der Qualitätssicherung aufgefallen.

Proprietäre
Berechtigungs-
prüfungen verletzen
Compliance-
Anforderungen

Aber auf jeden Fall sorgen proprietäre, undokumentierte Berechtigungsprüfungen dafür, dass das gesamte SAP-System nicht mehr auditierbar ist. Der Kunde weiß nicht, welche zusätzlichen Business-Funktionen ausgeführt oder unterbunden werden. Die Nachvollziehbarkeit ist damit nicht mehr gewährleistet, ein äußerst kritischer Faktor für den Geschäftsbetrieb.

5.2.3 Maßnahmen

- [!]** Immer wenn Sie auf eine proprietäre Berechtigungsprüfung stoßen, sollte diese entfernt werden. Der einzig richtige Weg für Berechtigungsprüfungen ist der SAP-Standard, die adäquate programmatische Lösung sind Authority Checks (siehe Abschnitt 5.1). Wenn Sie trotz aller Warnungen dennoch eigene Berechtigungskonzepte entwickeln, sollten Sie gute Gründe dafür haben und diese auch dokumentieren.

5.2.4 Selbsttest

Für alle Entscheidungsträger, Administratoren und Auditoren gestaltet sich die Identifizierung abgeleiteter Berechtigungen als schwierig. Es gibt

keine effiziente Möglichkeit, solch ein Problem zu finden, ohne den Sourcecode zu analysieren. Dies erfordert zwingend ABAP-Kenntnisse, da die verdächtigen Codestellen manuell untersucht werden müssen. Der Aufwand hierfür kann sehr hoch sein. Betrachten Sie hierzu sämtliche Vorkommen von `SY-UNAME` und `SYST-UNAME`, und kontrollieren Sie dann die zugehörige Applikationslogik:

- ▶ Untersuchen Sie alle Vorkommen von `SY-UNAME` und `SYST-UNAME` im ABAP-Code:
 - ▶ Wird hierbei auf einen festen Benutzernamen hin geprüft? Wenn Sie `IF SY-UNAME = 'BENUTZERNAME'` finden, haben Sie eine hart codierte Berechtigung entdeckt.
 - ▶ Wird auf eine dynamische Liste von Benutzern hin geprüft? Wenn Sie eine Prüfung von `SY-UNAME` oder `SYST-UNAME` gegen einen Tabelleninhalt finden, haben Sie eine dynamisch codierte proprietäre Berechtigung entdeckt.
- ▶ Prüfen Sie für alle proprietären Berechtigungschecks, ob bestimmte Benutzer kritische Funktionen aufrufen können, die anderen nicht zugänglich sind. Alternativ sollten Sie auch überprüfen, ob wichtige Funktionen nur bei bestimmten Benutzern nicht durchgeführt werden, wie zum Beispiel die Ausführung eines Abbuchungsauftrags.

Beachten Sie, dass Sie mit diesen Tests nicht alle möglichen proprietären Berechtigungsprüfungen finden werden. Sie müssen immer individuell die Anwendung im Kontext betrachten. Im Rahmen einer Webanwendung könnte zum Beispiel auch ein Parameter oder ein Cookie genutzt werden, um die Applikationslogik zu beeinflussen. Sie werden staunen, aber ein Browser-Cookie mit dem Inhalt `ADMIN=TRUE` ist im Rahmen von Audits schon mehrfach vorgekommen. Die Anwendung lief dann im Administratormodus.

Es kann unterschiedliche proprietäre Berechtigungsprüfungen geben

Ist die proprietäre Berechtigungsprüfung wider Erwarten beabsichtigt, sollten Sie die zugehörige Dokumentation auf Vollständigkeit hin überprüfen und Auditoren über diesen Mechanismus informieren. In aller Regel werden Sie hier jedoch Code vorfinden, der vom Entwickler versehentlich nicht gelöscht oder der von bösartigen Benutzern als Hintertür eingebaut wurde.

[+]

Eine gute Alternative zur Erkennung von Berechtigungen, die aus Benutzernamen abgeleitet wurden, ist auch der Einsatz von statischen Codeanalysetools. Hiermit können Sie auch in großen Programmen schnell entdecken, ob die Variablen `SY-UNAME` und `SYST-UNAME` mit statischen Werten verglichen oder in `SELECT`-Anweisungen verwendet werden.

5.3 Fehlende Berechtigungsprüfungen in RFC-fähigen Funktionen

ABAP-Funktionen kapseln Business-Logik. Idealerweise kann dadurch derselbe Code von verschiedenen Programmen aufgerufen und somit mehrfach verwendet werden. Die (klassische) ABAP-Entwicklung erlaubt darüber hinaus, Funktionsbausteine (im Folgenden kurz: Funktionen) auch von anderen Systemen aus aufzurufen. Technisch wird dies durch die RFC-Schnittstelle (Remote Function Call) ermöglicht: Die betreffende Funktion kann per Konfiguration als RFC-fähig deklariert werden, dann ist sie prinzipiell von allen Systemen aus aufrufbar, zu denen eine Netzwerkverbindung existiert.

RFC-Aufrufe können
unterschiedlichen
Quellen
entstammen

Wie Abbildung 5.2 veranschaulicht, können RFC-fähige Funktionen von SAP-Systemen mit ABAP- oder Java-Stack aufgerufen werden. In diesem Beispiel greift sowohl das ABAP-System mit der System-ID A02 als auch das Java-System mit der System-ID J01 per RFC auf den Funktionsbaustein YYY im ABAP-System A01 zu. RFC-fähige Funktionen können darüber hinaus auch von einfachen, selbst geschriebenen Java- oder C-Programmen gerufen werden, die die RFC- oder Java-Connector-Schnittstelle (JCo) implementieren.

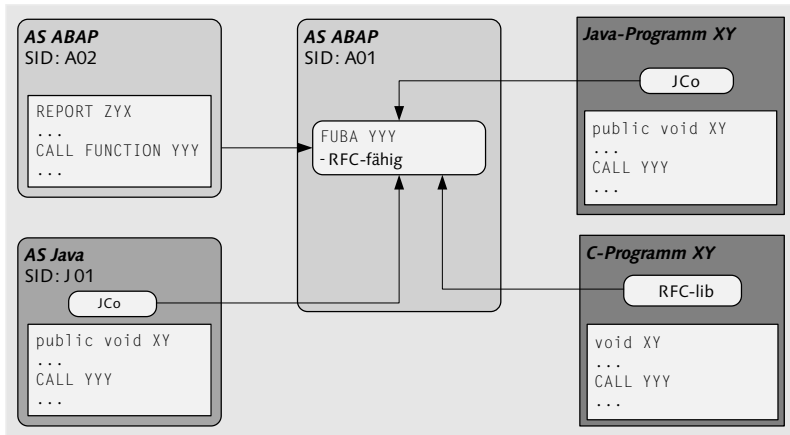


Abbildung 5.2 RFC-Szenario

Da die RFC- bzw. JCo-Schnittstelle über den SAP Service Marketplace (<http://service.sap.com>) verfügbar ist, kann prinzipiell jeder Entwickler ein Programm schreiben, das RFC-fähige ABAP-Funktionen aufruft und deren Rückgabewerte ausliest. Das aufrufende Programm benötigt dazu Benutzernamen und Passwort eines Benutzerkontos auf dem Host. Außerdem müssen IP-Adresse, SAP-System-ID und Mandant bekannt sein.

Das bedeutet, dass in einem Intranet prinzipiell jeder PC benutzt werden kann, um RFC-Aufrufe zu starten. Ebenso können SAP-Systeme von Partnern und Zulieferern RFC-Aufrufe an Ihr SAP-System absetzen. Sie müssen dazu allerdings eine Verbindung zum Server aufbauen können, was in einem B2B-Szenario in der Regel möglich ist.

Jeder kann RFC-Aufrufe starten

Natürlich überprüft ein SAP-System eingehende RFC-Aufrufe. Hierbei werden zunächst Benutzername und Passwort verifiziert. Dann wird geprüft, ob der angegebene Benutzer die Berechtigung hat, RFC-Aufrufe zu starten. Dies hängt von den entsprechenden Einträgen im Berechtigungsobjekt S_RFC ab. Sind alle Prüfungen positiv, wird der RFC-Aufruf zugelassen.

Allerdings gibt es noch eine Besonderheit: In einer sogenannten Trusted/Trusting-Verbindung zwischen zwei SAP-Systemen entfällt die Prüfung des Passwortes. Das Trusted System benötigt nur den Benutzernamen eines Benutzerkontos mit RFC-Berechtigungen, dann kann es auf dem Trusting System RFC-fähige Funktionen ausführen. Dieser Unterschied ist auch noch einmal optisch in Abbildung 5.3 dargestellt. Die nicht erfolgende Passwortüberprüfung im Falle des Trusted/Trusting-Szenarios ist hier eingekreist. Dies ist SAP-Standard und wird hier nur noch einmal zum generellen Verständnis erklärt.

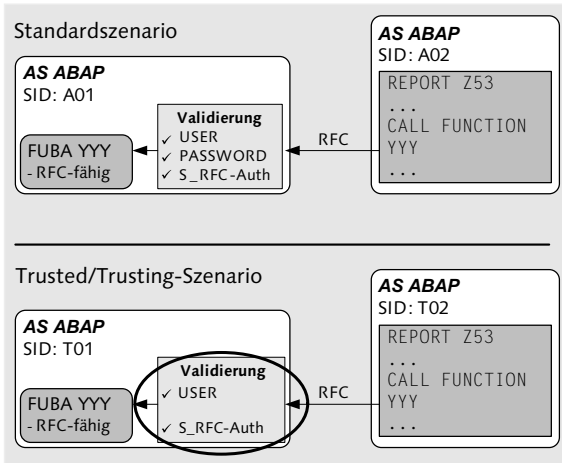


Abbildung 5.3 Sicherheitsprüfungen in RFC-Szenarien

5.3.1 Anatomie der Schwachstelle

Für die sichere ABAP-Programmierung ist in diesem Kontext wichtig, dass ein SAP-System zunächst nur prüft, ob ein bestimmtes Benutzerkonto RFC-fähige Funktionen aufrufen darf. Das SAP-System überprüft

Die RFC-Prüfung erfolgt nur auf Funktionsgruppen

jedoch nicht, welche Funktionen aufgerufen werden dürfen. Der einzige Schutz für Funktionen besteht darin, für die ganze Funktionsgruppe eine Berechtigungsprüfung einzurichten. Dadurch werden allerdings alle Funktionen der Funktionsgruppe auf die gleiche Berechtigungsstufe gestellt. Dies führt im praktischen Betrieb häufig zu Problemen.

5.3.2 Risiko

- [*]** Bei einem auditierten Add-on für SAP E-Recruitment enthielt eine Funktionsgruppe verschiedene Funktionen, von denen die meisten unkritisch waren. Daher war auch keine Berechtigungsprüfung für die Funktionsgruppe konfiguriert worden. Es wurde jedoch eine RFC-fähige Funktion gefunden, die Dateien auf dem SAP-Applikationsserver löschen konnte, und hierbei war der Name der zu löschenden Datei ein Funktionsparameter. Daher konnte jeder Benutzer mit S_RFC-Berechtigungen beliebige Dateien auf dem produktiven SAP-System löschen, für die der SAP-Server die Betriebssystemberechtigung hatte – und das sogar von jedem System im Netzwerk aus.

Die Berechtigung muss auch bei RFCs explizit geprüft werden

Für die ABAP-Entwicklung bedeutet das insbesondere, dass alle RFC-fähigen Funktionen die Berechtigung des aufrufenden Benutzers explizit prüfen müssen. Anderenfalls können sie beliebig von anderen Systemen aus aufgerufen werden.

Natürlich können Sie Berechtigungen für die Funktionsgruppe einrichten. Dadurch werden allerdings alle Funktionen in der Funktionsgruppe gleich stark oder schwach geschützt. Besteht die Funktionsgruppe aus Funktionen unterschiedlichen Schutzbedarfs, kann dieser Ansatz Probleme mit sich bringen: Entweder sind dann auch eigentlich harmlose Funktionen durch hohe Berechtigungen geschützt und daher für viele Benutzer nicht verfügbar, oder kritische Funktionen sind auf einer zu niedrigen Schutzstufe und können unberechtigt aufgerufen werden.

5.3.3 Maßnahmen

Dieses Problem kann auf zwei Arten gelöst werden:

- ▶ Beim Design einer Funktionsgruppe kann bereits darauf geachtet werden, dass nur Funktionen gleicher Schutzstufe enthalten sein dürfen. Dies muss jedoch gut dokumentiert und überwacht werden, anderenfalls kann solch ein Konzept mit der Zeit (versehentlich) unterwandert werden.
- ▶ Daher ist ein anderer Weg zu bevorzugen: Jede RFC-fähige Funktion sollte eine explizite programmatische Berechtigungsprüfung durch-

führen. Dadurch ist die Funktion nicht mehr von der Schutzstufe der Funktionsgruppe abhängig. Ein Entwickler kann so die zur Ausführung seiner Funktion nötigen Berechtigungen festlegen und in der Funktion prüfen. Die kritische Funktion bleibt somit auch geschützt, wenn die Berechtigungen für die Funktionsgruppe nachträglich reduziert werden.

Der Beispielcode in Listing 5.5 führt eine explizite programmatische Berechtigungsprüfung durch das ABAP-Kommando `AUTHORITY-CHECK` durch. Nur wenn der Rückgabewert von `AUTHORITY-CHECK` gleich 0 ist, hat der anfragende Benutzer die Berechtigung für die Aktion.

```
AUTHORITY-CHECK OBJECT 'Z_EXAMPLE'
  ID 'FILEGRP' FIELD 'FIN'
  ID 'ACTVT'   FIELD '23'.
IF sy-subrc <> 0.
  RAISE 'User not authorized'.
ENDIF.
DELETE DATASET filename.
```

Listing 5.5 Berechtigungsprüfung in RFC-fähigen Funktionen

5.3.4 Selbsttest

Um solch ein Problem zu entdecken, müssen Sie zunächst alle RFC-fähigen Funktionen identifizieren. Dann muss geprüft werden, ob für eine Funktion spezielle Berechtigungen erforderlich sind. Ist dies der Fall, sollte in der Funktion ein hinreichender `AUTHORITY-CHECK` im ABAP-Code implementiert sein.

5.4 Debug-Code in Assert Statements

Komplexe Softwareapplikationen bestehen grundsätzlich aus mehreren Komponenten, die miteinander verknüpft sind. Damit die Komponenten Informationen austauschen können, haben sie gewisse Anforderungen, zum Beispiel Datentypen oder Wertebereiche von Parametern. Im Paradigma *Programming by Contract* werden diese Anforderungen als ein Vertrag definiert, der zwischen Aufrufer und Aufrufendem besteht. Datentypen von Parametern können zum Beispiel in einer Schnittstelle als explizite Anforderung hinterlegt werden.

Das detaillierte Einhalten des Vertrags zwischen Aufrufer und Aufrufendem kann mit *Assertions* überprüft werden. ABAP erlaubt mit dem entsprechenden `ASSERT`-Kommando auch feingranulare Anforderungen explizit im Code zu dokumentieren, wie beispielsweise die Prüfung von

Mit Assertions können Anforderungen feingranular geprüft werden

Wertebereichen. ASSERT hilft Programmierern, Annahmen explizit im Code zu dokumentieren und in der Entwicklungs- und Testphase zu testen. Ein großer Vorteil gegenüber Prüfungen mit IF-Verzweigungen ist, dass Assertions im Produktivsystem abgeschaltet werden können und somit die Performance nicht beeinflussen.

ASSERT verarbeitet immer einen logischen Ausdruck:

- ▶ Ist der logische Ausdruck wahr, wird der Code hinter dem ASSERT-Befehl ausgeführt.
- ▶ Ist der logische Ausdruck falsch, bricht die Verarbeitung ab.

Es gibt allerdings zwei Varianten von Assertions in ABAP: aktive und konfigurierbare.

- ▶ Bei aktiven Assertions (ASSERT) wird der logische Ausdruck immer evaluiert.
- ▶ Bei konfigurierbaren Assertions (ASSERT ID) bietet das SAP-System mehrere mögliche Aktionen an, die in Transaktion SAAB in einer Checkpoint-Gruppe konfiguriert werden können. Diese Aktionen sind: INAKTIV, ANHALTEN, PROTOKOLLIEREN und ABBRECHEN.

Abbildung 5.4 zeigt die verschiedenen Optionen für die Konfiguration der Checkpoint-Gruppen von Assertions.



Abbildung 5.4 Konfiguration von Checkpoint-Gruppen für Assertions

INAKTIV bedeutet, dass ASSERT-Anweisungen vom SAP-System zur Laufzeit ignoriert werden. Diese Einstellung sollte auf Produktivsystemen gesetzt sein.

[+]

In Test- und Entwicklungssystemen helfen die übrigen Einstellungen dem Programmierer bei der Entwicklung und beim Testen des Codes. Diese drei Aktionen sind relevant, wenn der logische Ausdruck als falsch ausgewertet wird.

- ▶ ANHALTEN ermöglicht den Absprung in den Debugger.
- ▶ PROTOKOLLIEREN schreibt einen Eintrag in die Log-Datei von Transaktion SAAB.
- ▶ ABBRECHEN beendet das laufende Programm und führt zu einem Kurzdump. Dies geschieht übrigens auch bei allen aktiven Assertions, wenn der logische Ausdruck falsch war.

Das Codebeispiel in Listing 5.6 zeigt den Einsatz einer aktiven Assertion. Das Unterprogramm `division` dividiert die Variablen `a` und `b` und gibt das Ergebnis in Variable `c` zurück. Hier ist zu beachten, dass Divisionen durch 0 nicht definiert sind und zu einem Laufzeitfehler führen würden. Um sicherzustellen, dass `c` nicht 0 ist und um das Debugging in der Entwicklungs- und Testphase zu erleichtern, wird in Zeile 4 des Beispiels das Kommando `ASSERT` genutzt.

```
FORM division
  USING a b TYPE f
  CHANGING c TYPE f.
  ASSERT b <> 0.
  c = a / b.
ENDFORM.
```

Listing 5.6 Einsatz von ASSERT

5.4.1 Anatomie der Schwachstelle

Bei Assertions gibt es zwei Sicherheitsaspekte: Sowohl der Code hinter einem `ASSERT ID`-Befehl als auch der Code innerhalb eines `ASSERT ID`-Befehls werden abhängig von der Systemkonfiguration ausgeführt. Wie bereits angemerkt, können Prüfungen mit `ASSERT ID` in Produktivumgebungen ausgeschaltet werden – etwa aus Performance-Gründen.

Bei Assertions wird Code abhängig von der Konfiguration ausgeführt

Ist nun der logische Ausdruck in einer Assertion so konstruiert, dass er immer falsch ist, wird auf dem Testsystem der Code hinter dem Befehl `ASSERT ID` nie ausgeführt. Steht die Konfiguration dann noch auf `PROTOKOLLIEREN`, läuft der Code auch ohne Fehler durch. Gelangt der Code allerdings in das Produktivsystem, verhält er sich anders, denn in Produk-

tivsystemen sind Assertions per Grundeinstellung **INAKTIV**. Das heißt, der Code hinter dem `ASSERT ID`-Befehl wird plötzlich ausgeführt.

Der Code in Listing 5.7 zeigt äußerst böses ABAP, das auf den Testsystemen (je nach Konfiguration) nicht erkannt wird, aber im Produktivsystem auch hartgesottene Administratoren das Fürchten lehren würde.

```
FORM dont_call_me.
  ASSERT ID zvf_test CONDITION 1 = 2.
  EXEC SQL.
    drop table USR01
  ENDEXEC.
ENDFORM.
```

Listing 5.7 Unerwünschte Nebeneffekte im Produktivsystem durch Assertions

Das SAP-System wird im produktiven Betrieb jedoch nicht nur das Ergebnis der `ASSERT ID`-Prüfung ignorieren, sondern die Prüfung gar nicht erst ausführen. Das folgende Codebeispiel zeigt einen `ASSERT ID`-Aufruf, mit dem der Rückgabewert einer Methode geprüft wird. Der `ASSERT ID`-Befehl wird hierbei über die Checkpoint-Gruppe `ZVF_TEST` konfiguriert:

```
ASSERT ID zvf_test CONDITION verify( l_param )= 'X'.
```

Hier wird die Klassenmethode `verify` nur aufgerufen, wenn Assertions nicht als **INAKTIV** konfiguriert sind. Problematisch wird das, wenn `verify` zum Beispiel Autorisierungsprüfungen oder sicherheitsrelevante Werteprüfungen durchführt. Diese werden dann im Produktivsystem nicht ausgeführt, was zu kritischen Sicherheitslücken führen kann.

Assertions können zu unterschiedlichem Verhalten von Test- und Produktivsystem führen

Ein weiteres Problem kann durch Seiteneffekte der aufgerufenen Methode entstehen. Verändert die gerufene Methode den Datenzustand im laufenden System, können im Produktivsystem ebenfalls Sicherheitslücken auftauchen, die im Testsystem nicht vorhanden waren. Beispielsweise wenn `verify` die übergebenen Daten nicht nur prüft, sondern auch filtert, codiert oder anderweitig verändert.

5.4.2 Risiko

Das große Problem von Assertions ist, dass dasselbe ABAP-Programm sich unter Umständen im Produktivsystem anders verhält als im Testsystem. Das bedeutet, dass Sicherheitslücken im Produktivsystem meistens noch nicht einmal im Testsystem nachvollziehbar sind, und das obwohl der Code identisch ist.

- [*]** In einem Fall sollte eine CRM-Eigenentwicklung getestet werden. Eines der Module sollte monatliche Analysen der Umsätze eines neuen Online-Shops nach verschiedenen Kriterien berechnen. Während der Entwick-

lung dieses Programms hatten die Programmierer im Testsystem keine Daten, um ihre Berechnungen auszuprobieren und zu verifizieren. Daher haben sie sich eine Testfunktion gebaut, die die entsprechenden SAP-Tabellen initialisierte und mit über 100.000 Testbestellungen füllte. Dadurch waren Testdaten vorhanden, und die Entwickler konnten testen. Die Funktion selbst wurde nur wenig zu Testzwecken benutzt und war hinter einer Assertion versteckt. Die Assertion brach die Ausführung ab, da sie eine logische Bedingung hatte, die immer falsch war. Dadurch wurde der Code nicht ausgeführt, war aber einfach zu reaktivieren.

Allerdings hatten die Entwickler auch diese Assertion einer Checkpoint-Gruppe zugewiesen, was bedeutete, dass die Ausführung der Assertion von der Konfiguration des SAP-Systems abhing. Wäre dieser Code nicht bei einem Sicherheitsaudit entdeckt worden, wären beim ersten Aufruf der monatlichen Umsatzanalyse alle Bestellungen gelöscht und mit Testdaten überschrieben worden.

Assertions können schwerwiegende Sicherheitsprobleme auslösen

Dieses Beispiel zeigt deutlich, dass funktionale Tests in einem Testsystem überraschend ganz andere Ergebnisse liefern wie in einem Produkktivsystem.

5.4.3 Maßnahmen

Stellen Sie sicher, dass im logischen Ausdruck von Assertions keine Methoden gekapselt bzw. dass gekapselte Methoden nicht sicherheitskritisch sind. Sicherheitskritische Prüfungen müssen immer durchgeführt werden und dürfen daher nicht innerhalb von `ASSERT ID`-Ausdrücken realisiert werden. Assertions dürfen nur für Prüfungen verwendet werden, die funktionalen Charakter haben und die Performance im Produkktivsystem beeinflussen würden.

[+]

In jedem Fall müssen Sie jeglichen Code entfernen, der hinter einer Assertion steht, deren logischer Ausdruck immer falsch ist. Anderenfalls wird dieser Code ausgeführt, wenn die Assertion durch Konfiguration `INAKTIV` wird und somit nicht zur Ausführung kommt.

[+]

5.4.4 Selbsttest

Da Schwachstellen im Zusammengang mit Assertions abhängig von der Systemkonfiguration auftreten, ist ein Code-Audit die einzig verlässliche Methode für die Analyse. Dabei sollten alle Aufrufe von `ASSERT` bzw. `ASSERT ID` manuell im Code analysiert werden.

Ein besonderes Augenmerk sollten Sie auf `ASSERT ID`-Aufrufe legen, die innerhalb des logischen Ausdrucks Methoden aufrufen. Hier müssen Sie prüfen, ob die Methode sicherheitsrelevante Seiteneffekte hat (zum Beispiel Filtern oder Codieren von Daten). Dieser Code sollte entfernt werden.

[!] An dieser Stelle folgt bewusst kein sicheres Codebeispiel, da selbst einfache Beispiele in bestimmten Szenarien sicherheitskritisch sein können.

Bei Tests in laufenden Systemen gibt es praktisch keine Chance, Schwachstellen zu finden, die aus unsicheren Assertions resultieren. Ein kleiner Indikator für Probleme mit Assertions ist allerdings, wenn eine Schwachstelle im Produktivsystem auftritt, nicht jedoch im Testsystem.

Bei Audits lohnt es sich, die Log-Dateien von Transaktion SAAB zu analysieren. Wenn Sie die Checkpoint-Gruppe der Assertions kennen, können Sie diese über den Reiter **AKTIVIERUNG** auf den Wert **PROTOKOLLIEREN** setzen. Dann sehen Sie, wo `ASSERT`-Befehle verwendet werden. Abbildung 5.5 zeigt das **PROTOKOLL** der Checkpoint-Gruppe `ZVF_TEST`. Hieran lässt sich genau erkennen, wo die Assertion ausgelöst wurde: in diesem Beispiel in der Seite *encoding.htm* in Zeile 12 der BSP-Anwendung `ZVF_HTMLB`.

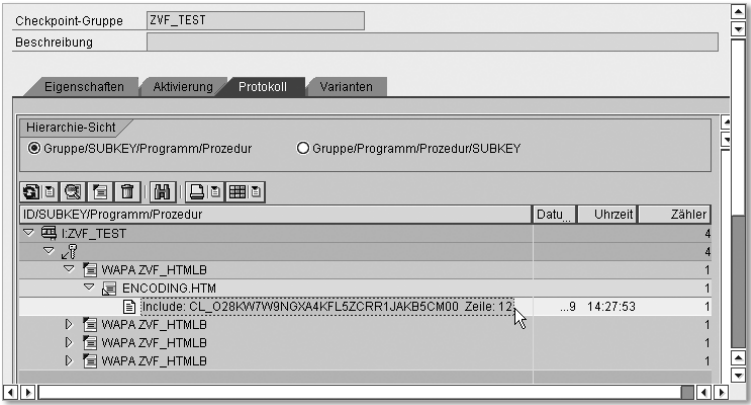


Abbildung 5.5 Protokoll-Eintrag einer Checkpoint-Gruppe

5.5 Generischer und dynamischer ABAP-Code

Bei Softwareprojekten ist die Wiederverwendbarkeit von Funktionalität ein wichtiger Kostenfaktor. Code, der so geschrieben ist, dass er von verschiedenen Teilen eines Programms verwendet werden kann, reduziert Entwicklungs- und vor allem Wartungskosten.

Wenn Code allgemeinere Probleme lösen kann als die im konkreten Fall erforderlichen Aufgaben, wird von *generischem Code* gesprochen. Generischer Code ermöglicht die Wiederverwendbarkeit von Funktionalität in verschiedenen Anwendungsszenarien. Daher versuchen viele Entwickler, ihren Code möglichst generisch zu entwickeln. Doch je flexibler ein Programm geschrieben ist, desto komplexer ist meist der Code. Das kann dazu führen, dass das Programm plötzlich Funktionalität beinhaltet, die der Entwickler eigentlich gar nicht absichtlich eingebaut hat.

Mit anderen Worten: Flexibilität führt zu mehr oder weniger erwünschten Nebeneffekten. Prinzipiell sind Nebeneffekte nicht schlimm, sofern sie erkannt werden. Dann kann der Entwickler Maßnahmen planen, die den potenziellen Schaden verhindern.

Flexibilität kann zu unerwünschten Nebeneffekten führen

5.5.1 Anatomie der Schwachstelle

Nebeneffekte werden in der Regel allerdings nicht erkannt, oder vielmehr wird das Schadenspotenzial der Nebeneffekte nicht erkannt. Das liegt vor allem daran, dass unerwünschte Nebeneffekte im normalen Betrieb einer Software nicht auftreten. Nur ganz gezieltes Benutzerverhalten löst die Nebeneffekte aus.

Schauen Sie sich den Code in Listing 5.8 an. Es handelt sich dabei um eine Methode, die Eingaben aus einem Bestellformular in eine interne Tabelle einliest. Können Sie erkennen, welches Sicherheitsproblem hier vorliegt?

```
METHOD handle_code.
  TYPES: BEGIN OF line,
          id TYPE string,
          num TYPE i,
        END OF line.
  TYPES arttab TYPE TABLE OF line.
  DATA: lv_numitems TYPE i,
        lv_num       TYPE n,
        request       TYPE REF TO if_http_request,
        lv_str        TYPE string,
        lv_line       TYPE line,
        lt_arttab     TYPE arttab.

  lv_numitems = request->get_form_field( 'numitems' ).
  REFRESH lt_arttab.
  DO lv_numitems TIMES.
    CLEAR lv_line.
    lv_num = sy-index.
    CONCATENATE 'article-' lv_num
      INTO lv_str.
    lv_line-id = request->get_form_field( lv_str ).
```



```

        CONCATENATE 'amount-' lv_num
        INTO lv_str.
        lv_line-num = request->get_form_field( lv_str ).
        APPEND lv_line TO lt_arttab.
    ENDDO.
ENDMETHOD.

```

Listing 5.8 Beispiel für generischen Code

- [★]** Das in Listing 5.8 dargestellte Beispiel für generischen Code ist einem Webshop nachempfunden, der in einem Sicherheits-Audit analysiert wurde. Er liest eine bestimmte Anzahl von Artikelnummern samt der jeweiligen Bestellmenge ein und speichert die Bestellung in einer internen Tabelle. Der Code wurde flexibel entwickelt, damit er kleine und große Bestellungen gleichermaßen behandeln kann. Dazu übergibt die sendende HTML-Seite einfach die Anzahl der Zeilen in der Bestellung (`numitems`) an den serverseitigen Handler. Dann wird genau die angegebene Menge an Zeilen eingelesen und verarbeitet.

Der Code funktioniert zweifelsohne, aber er hat Nebeneffekte. Diese Nebeneffekte werden allerdings im normalen Betrieb der Anwendung nicht auffallen, denn in der Regel werden Benutzer und Tester nur eine überschaubare Anzahl von Artikeln bestellen. Doch was geschieht, wenn ein Benutzer gezielt einen sehr hohen Wert für die Variable `numitems` an den Server schickt?

Die Methode wird versuchen, die angegebene Anzahl von Zeilen einzulesen und in die interne Tabelle zu speichern. Wenn für `numitems` aber 10.000.000 eingegeben wird, wird die `DO`-Schleife sehr lange laufen und damit ebenso viel CPU-Zeit verbrauchen. Zudem wird für jede Zeile ein (leerer) Eintrag in der internen Tabelle angelegt. Dies führt zusätzlich zu einem hohen Verbrauch an Arbeitsspeicher, denn auch ein leerer String und ein leerer Integer belegen Speicherplatz. Ein Angreifer könnte in diesem Fall mit wenigen Anfragen an den Server einen sogenannten *Denial of Service* provozieren. Das bedeutet, dass die Verfügbarkeit des SAP-Systems für andere Benutzer stark beeinträchtigt wird, da erhebliche Ressourcen blockiert werden.

Dieses einfache Beispiel verdeutlicht, dass auch harmlos aussehender Code kritische Nebeneffekte haben kann, und das, obwohl der Code eigentlich genau das macht, was er soll. Es gibt jedoch einen wichtigen Unterschied: Funktionierender Code tut, was er soll; sicherer Code tut, was er tun soll – *und nichts anderes*. Angelehnt an ein populäres juristisches Zitat müsste von Entwicklern verlangt werden:

»Schwören Sie, Code zu schreiben, der gemäß der Spezifikation funktioniert, gemäß der gesamten Spezifikation und nur gemäß der Spezifikation.«

Abbildung 5.6 verdeutlicht den hier wichtigen Unterschied: Es gibt Code, der die Anforderungen erfüllt, möglicherweise aber auch noch mehr – dies sind die erwähnten Nebeneffekte. Im Gegensatz dazu gibt es Code, der *nur* die Anforderungen erfüllt und sonst nichts. Jeder Nebeneffekt wird früher oder später von einem Angreifer missbraucht. Je spezifischer daher der Code geschrieben ist, desto geringer das Risiko, dass er unerwünschte Nebeneffekte hat.

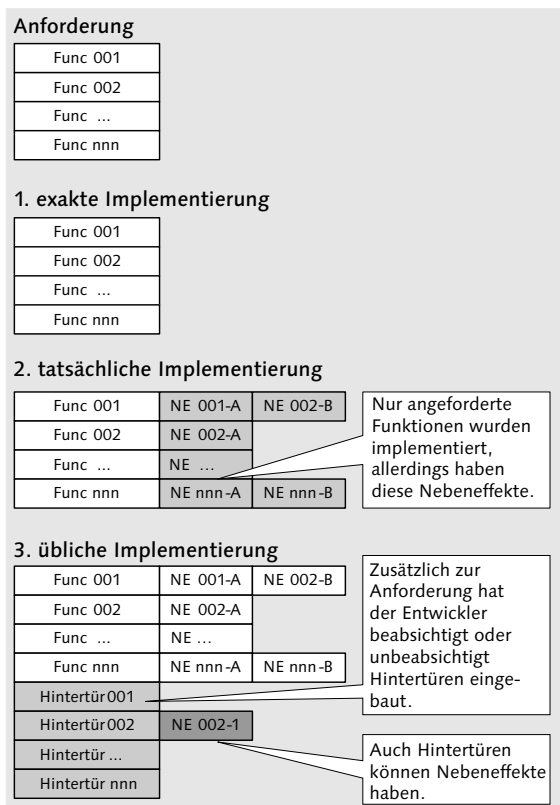


Abbildung 5.6 Logischer Missbrauch vs. Veränderung von Code

Hierbei sind zwei prinzipielle Ursachen von Nebeneffekten durch generischen Code zu unterscheiden: logischer Missbrauch und semantische Änderungen.

- Manche Nebeneffekte ergeben sich durch einen Designfehler in der Logik des Programms. Das bedeutet, dass das Programm für alle

Generischer Code kann zu logischem Missbrauch oder semantischen Änderungen führen

erwarteten Eingaben korrekt ausgeführt wird. Jedoch ergeben sich bei manchen Werten Nebeneffekte, die der Entwickler nicht vorhergesehen hat. Solch ein Problem wurde im Codebeispiel zum Webshop dargestellt: Der Entwickler hat nicht damit gerechnet, dass jemand einen sehr hohen Wert für die Anzahl der bestellten Artikel eingibt und so die Verarbeitungszeit der Anfrage stark beeinflusst. Solche Schwachstellen sind häufig in Kundenprojekten zu finden.

- Die zweite Variante von Nebeneffekten entsteht dadurch, dass ein ABAP-Befehl in unerwarteter Weise semantisch verändert wird, denn manche ABAP-Befehle werden erst zur Laufzeit erzeugt und ausgeführt; sie stehen daher zur Entwicklungszeit noch nicht (vollständig) fest. Erst durch Eingaben des Benutzers wird der Code vollständig ermittelt und ausgeführt. In diesem Fall wird von sogenanntem dynamischen ABAP gesprochen. Hier würde der Angreifer nicht die unzureichende Logik des Programms manipulieren, sondern versuchen, das Programm selbst zu ändern. Allerdings ist dies nur in gewissen Grenzen möglich.

Abbildung 5.7 illustriert noch einmal die beiden beschriebenen Ursachen von Nebeneffekten.

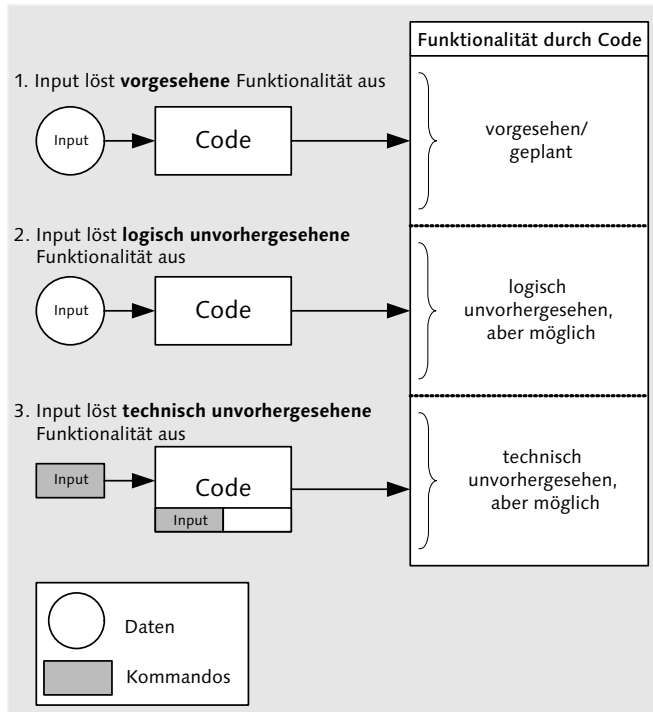


Abbildung 5.7 Ursachen von Nebeneffekten bei generischem Code

Hinweis

Wir können hier nur schwierig eine Anleitung für die generelle Vermeidung von logischen Denkfehlern in ABAP-Code liefern, da die Definition eines logischen Fehlers stark vom Geschäftsszenario abhängt. Ein logischer Fehler in Applikation A kann durchaus zum normalen Funktionsumfang von Applikation B gehören. Daher beschäftigt sich dieser Abschnitt mit semantischen Nebeneffekten von ABAP-Befehlen und konzentriert sich auf sogenannte dynamische ABAP-Befehle.

Bei einem dynamischen Befehl ist das endgültige Verhalten des Befehls erst zur Laufzeit erkennbar und wird nicht vom Compiler bestimmt, sondern vom Interpreter. Durch entsprechende Eingaben kann ein böartiger Benutzer damit den Befehl in unerwünschter Weise verändern. Dadurch ergeben sich abhängig vom ABAP-Befehl ganz unterschiedliche Sicherheitsrisiken. Manche dieser Sicherheitsrisiken sind so weit verbreitet, dass sie einen eigenen Namen haben, wie zum Beispiel die SQL-Injection (siehe Abschnitt 5.8).

Bei dynamischen Befehlen steht das Verhalten erst zur Laufzeit fest

Doch generell sind alle Befehle, die dynamischen Code ermöglichen, ein potenzielles Sicherheitsrisiko. Diese Befehle sind nicht grundsätzlich schlecht, aber Sie müssen sich ihrer Risiken bewusst sein. Nehmen Sie als Analogie die Lupe zur Verdeutlichung. Fast jeder, der eine Lupe benutzt, tut dies, um Texte besser lesen oder Bilder besser erkennen zu können, denn das ist die primäre Funktion der Lupe. Doch jeder sollte sich des Risikos bewusst sein, dass eine Lupe auch Feuer entfachen kann, wenn die Sonne in einem bestimmten Winkel hindurchscheint. Nur mit hinreichendem Wissen einer Technologie können daher ihre Nebeneffekte kontrolliert werden.

Hinweis

Im Folgenden werden einige ABAP-Befehle betrachtet, die dynamischen Code ermöglichen und denen kein spezieller Abschnitt in diesem Buch gewidmet ist. Beachten Sie hier jedoch: Nur weil eine Schwachstelle noch keinen allgemein üblichen Namen erhalten hat, ist sie dadurch nicht weniger gefährlich.

Kritische ABAP-Befehle, die dynamischen Code ermöglichen

Die Befehle `ASSIGN` und `WRITE` können unter anderem Daten von einer Variablen in eine andere kopieren. Alle Variablen sind explizit deklariert, und im Code ist genau zu erkennen, von wo nach wo die Daten kopiert werden. Listing 5.9 zeigt die normale Verwendung dieser Befehle. Hier werden Variablen kopiert, deren Inhalt in eine Tabelle geschrieben werden soll. Dieser Code hat aber lediglich Demonstrationscharakter.

```

* Explicit usage of ASSIGN
DATA: lv_str1      TYPE string,
      lv_str2      TYPE string,
      lv_dest(80) TYPE c,
      lv_ccinf     TYPE string.
FIELD-SYMBOLS <fs>.
ASSIGN lv_str1 TO <fs>.
* Explicit usage of WRITE
WRITE lv_str2 TO lv_dest.

```

Listing 5.9 Beispiel für explizite Programmierung (Kopieren)

Werden die Befehle `ASSIGN` und `WRITE` wie im Beispiel aus Listing 5.10 jedoch dynamisch verwendet, ist dem Code nicht mehr anzusehen, welche Variable zur Laufzeit ausgelesen wird. Je nach Eingabe des Benutzers können die Befehle nun die lokalen Variablen `lv_str1`, `lv_str2`, `lv_dest` und `lv_ccinf` auslesen. Die Eingabe muss dazu nur den Namen der Variablen beinhalten. Das bedeutet in diesem Fall, dass der Benutzer generischen Zugriff auf sämtliche lokalen Variablen des Programms hat. Dadurch kann er sämtliche Daten auslesen, die von diesem Programm gerade verarbeitet werden. In diesem Beispiel wäre es möglich, die Variable `lv_ccinf` auszulesen, die die Kreditkartendaten eines anderen Benutzers enthält, hier `U0002342`.

```

REPORT zassign.
PARAMETERS: cdynamic(80) TYPE c DEFAULT 'lv_str1'.
DATA: lv_str1      TYPE string,
      lv_str2      TYPE string,
      lv_dest(80) TYPE c,
      lv_ccinf     TYPE string.
FIELD-SYMBOLS <fs>.
CALL FUNCTION 'ZGETCCDATA'
  EXPORTING user = 'U0002342'
  IMPORTING cdata = lv_ccinf.
* Dynamic usage of ASSIGN
* (Input: SAP GUI input field)
ASSIGN (cdynamic) TO <fs>.
* Dynamic usage of WRITE
* (Input: SAP GUI input field)
WRITE (cdynamic) TO lv_dest.

```

Listing 5.10 Beispiel für die dynamische Programmierung (Kopieren)

Aber `WRITE` und `ASSIGN` können nicht nur lokale Variablen des eigenen Programms auslesen. Beide Befehle können auch die Variablen aller aufrufenden ABAP-Programme und die als `PUBLIC` markierten Klassenattribute der instanziierten Klassen auf dem SAP-Server auslesen. Alles, was dazu bekannt sein muss, sind die Namen des Programms bzw. der aufru-

fenden Klasse sowie die Namen der Variablen bzw. der Attribute. Dies kann ein Angreifer durch Kenntnis von SAP-Internas wissen oder aber durch Ausprobieren und Raten herausbekommen.

Listing 5.11 zeigt die verschiedenen Möglichkeiten, wie Variablen dynamisch aus Reports und Klassen ausgelesen werden können; sowohl `WRITE` als auch `ASSIGN` können hierfür verwendet werden. Der `ASSIGN`-Befehl besitzt noch weitere Alternativen, um Klassenattribute auszulesen, jedoch sind auch die Zugriffsmöglichkeiten von `WRITE` weitreichend. Besonders interessant ist hier, dass die Zugriffe auf Klassenattribute sogar mehrfach verschachtelt sein können, wie in Beispiel #3 aus Listing 5.11 gezeigt wird. Hier soll jedoch nicht auf alle Details eingegangen werden, sondern im Wesentlichen sollen die technischen Varianten zum besseren Verständnis aufgelistet werden. Insgesamt sind sieben Beispiele zusammengestellt.

```
REPORT Z_DYNAMIC_EXAMPLE.
  DATA: request TYPE REF TO if_http_request.
  DATA: lv_attr TYPE string.
  DATA: lv_class TYPE string.
  DATA: lc_bsprt TYPE REF TO cl_bsp_runtime.
  FIELD-SYMBOLS <fs> TYPE string.
  DATA: lv_dynamic(80) TYPE c,
        lv_dest(80) TYPE c.

* #1: Read variable "secret" from report "z_rep_abap_book"
  lv_dynamic = '(z_rep_abap_book)secret'.
* Dynamic ASSIGN
  ASSIGN (lv_dynamic) TO <fs>.
  WRITE: / 'ASSIGN:', <fs>.
* Dynamic WRITE
  WRITE (lv_dynamic) TO lv_dest.
  WRITE: / 'WRITE:', lv_dest.

* #2: Read public attribute "user" from class
* "z_cl_abap_book"
  lv_dynamic = 'z_cl_abap_book=>user'.
* Dynamic ASSIGN
  ASSIGN (lv_dynamic) TO <fs>.
  WRITE: / 'ASSIGN:', <fs>.
* Dynamic WRITE
  WRITE (lv_dynamic) TO lv_dest.
  WRITE: / 'WRITE:', lv_dest.

* #3: Read public attribute "session" from public class
* variable "request" of class variable "lc_bsprt" (nested
* access)
  lv_dynamic = 'lc_bsprt->request->session'.
```

```

* Dynamic ASSIGN
  ASSIGN (lv_dynamic) TO <fs>.
  WRITE: / 'ASSIGN:', <fs>.
* Dynamic WRITE
  WRITE (lv_dynamic) TO lv_dest.
  WRITE: / 'WRITE:', lv_dest.
* Generic memory read from classes through ASSIGN
PARAMETERS:
  c_attrb TYPE string,
  c_class TYPE string.

* #4: Access any attribute of an explicit class
  ASSIGN lc_bspst->(c_attrb) TO <fs>.
  WRITE: / <fs>.

* #5: Access any static attribute of an explicit class
  ASSIGN cl_bsp_runtime=>(c_attrb) TO <fs>.
  WRITE: / <fs>.

* #6: Access an explicit static attribute of any class
  ASSIGN (c_class)=>some_attrb TO <fs>.
  WRITE: / <fs>.

* #7: Access any static attribute of any class
  ASSIGN (c_class)=>(c_attrb) TO <fs>.
  WRITE: / <fs>.
  ASSIGN (c_class)=>(c_attrb) TO <fs>.
  WRITE: / <fs>.

```

Listing 5.11 Auslesen von Variablen aus Programmen und Klassen

5.5.2 Risiko

Dynamische `WRITE`- und `ASSIGN`-Befehle erlauben Lesezugriff auf nahezu beliebige Variablen im Speicher der aktiven ABAP-Anwendung. Basieren diese Zugriffe auf Benutzereingaben, kann es einem Angreifer gelingen, gezielt Daten auszuspähen. Datenbankzugriffe können prinzipiell durch Berechtigungen geschützt werden, Zugriffe auf Variablen jedoch nicht.

Dynamischer Code
kann zu Vertraulich-
keitsverlust führen

Eine solche Schwachstelle im ABAP-Code kann leicht dazu führen, dass vertrauliche Informationen preisgegeben werden, je nachdem, wie das Programm die gelesenen Daten weiterverarbeitet. Jedes Programm, das dynamische Variablenzugriffe erlaubt, gefährdet damit die Vertraulichkeit von produktiven Daten im SAP-System.

- [★]** Doch auch in anderer Hinsicht kann ein dynamisches `WRITE` bzw. `ASSIGN` zu Risiken führen. Unlängst gab es einen Fall, in dem ein Entwickler eine Hintertür in einen Report eingebaut hat. Er hat jedoch keinen direkten Vergleich von `SY-UNAME` mit seinem Benutzernamen durchgeführt, denn

dieser wäre leicht aufgefallen. Stattdessen hat er eine Variable `lv_name` definiert, die anfangs den Wert `'8Y0UN4M3'` hatte, dann hat er durch `TRANSLATE` diese Variable zur Laufzeit so geändert, dass sie den Wert `'SY-UNAME'` erhielt. Anschließend wurde mittels `ASSIGN lv_name TO <lv_var>` dynamisch der Wert von der Systemvariablen `SY-UNAME` in die Variable `<lv_var>` kopiert und dort ein Vergleich mit seinem Benutzernamen durchgeführt. Dies war eine hervorragende Tarnung seiner Hintertür.

Wie Sie sehen, können durch dynamischen Code auch schadhafte Funktionen verschleiert werden, da dynamischer Code viel schwieriger zu auditieren ist. Das Schadenspotenzial hängt dabei nur von der Kreativität des böswilligen Entwicklers ab.

Mit dynamischem Code kann Schadcode verschleiert werden

5.5.3 Maßnahmen

Sie sollten möglichst auf dynamischen ABAP-Code verzichten und Ihr Coding so konkret wie möglich an das zu lösende Problem anpassen. Generischer Code ist zwar wiederverwendbar, kann jedoch leicht zu kritischen Sicherheitslücken führen, wenn er zu komplex wird. Dynamischer Code ist kaum auditierbar und es kann daher zu Compliance-Problemen kommen.

[!]

5.5.4 Selbsttest

Um die hier dargestellten Probleme im Code zu finden, müssen Sie alle Vorkommen von `WRITE` und `ASSIGN` auf dynamische Verwendung hin prüfen. Beachten Sie aber bitte, dass es noch andere Befehle gibt, die dynamisch verwendet werden können. Dynamische Verwendung erkennen Sie daran, dass die Variablen in den `WRITE`- und `ASSIGN`-Zugriffen in Klammern stehen:

- ▶ dynamische `ASSIGN`-Befehle: `ASSIGN (var) TO <fs>.`
- ▶ dynamische `WRITE`-Befehle: `WRITE (var) TO dest.`

Dann muss geprüft werden, wo die Variable `var` gesetzt wird. Enthält sie Benutzereingaben, haben Sie ein Sicherheitsproblem identifiziert. In jedem Fall sollten Sie ganz genau prüfen, warum dynamisch auf Daten zugegriffen wird. Meistens lassen sich solche Zugriffe ohne Einbuße in statische Datenzugriffe umwandeln, die wesentlich weniger Angriffspotenzial haben als dynamische Zugriffe.

[+]

Vermeiden Sie in jedem Fall Code, der Benutzereingaben verwendet, um den dynamischen Teil einer Variablen zu befüllen. Anderenfalls kann ein böswilliger Benutzer gezielt den dynamischen Befehl zu Ihrem Nachteil missbrauchen.

Index

A

A2A 59

ABAP

ABAP Workbench 138

Besonderheit 42

dynamisches 170

Entwicklungsprozess 42

Funktionsbaustein 46

Mächtigkeit 42

Programm 46

Report 46

Sicherheitsdefekt 27

Sicherheitsproblem 34, 35

typisierte Programmiersprache 115

Unbreakable-Mythos 55

ABAP-Befehl

ASSERT 152

ASSERT ID 152

AUTHORITY-CHECK 151, 267

CALL 'SYSTEM' 204

CALL FUNCTION 166

CALL METHOD 166

CALL TRANSACTION 138

DELETE 179, 294

DELETE DATASET 189

GENERATE SUBROUTINE-POOL
172

INSERT 179, 294

INSERT REPORT 172

LEAVE TO TRANSACTION 138

MODIFY 179, 294

OPEN CURSOR 179

OPEN DATASET 186, 203

PERFORM 166

READ DATASET 189

REPLACE ALL OCCURRENCES 122

SELECT 179

SUBMIT 166

SYSTEM-CALL 195

TRANSFER 189

UPDATE 179, 294

Abbuchungsauftrag 147

Abnahmekriterium 64

absoluter Dateipfad 186

Action Message Format 279

Add N Edit, Cookie 245, 263, 348

Add-on

Browser 244

SAP 320

Ad-hoc-Aktion 65

Administrationsrecht 33

Administrator

Anwendername 109

Passwort 109

vertrauenswürdiger 213

Adobe Flash 272

AGate 309

agile Softwareentwicklung 82

Aktienkurs 235

Aktion

Ad-hoc 65

Plan 65

aktive Assertion 152

Algorithmus

Spidering 245

Verschlüsselung 136

allgemeine Geschäftsbedingungen 70

alphanumerisches Zeichen 120

AMF 279

Ampel, rote 98

Anforderung

Ergonomie 105

funktionale 38, 68

marktübliche 34

nicht funktionale 68

Qualität 68

sichere Software 34

testbare 70

veränderte 107

Angreifer

Administrationsrechte 33

internes Wissen 106

Rechenleistung 109

Angriff

Angriffsvektor 57

Intranet 335

Muster 29, 30

Schlagzeile 38

Simulation 38

- Angriff (Forts.)
 - zielgerichteter* 109
- Angriffsoberfläche 42, 56, 111, 195
 - E-Recruitment* 320
 - vergrößerte* 57
- Anmeldung
 - am SAP-System* 59
 - Daten* 236
- Annahme, falsche 35
- anonymer Anwender 246
- Anpassung 330
- Anti-Phishing-Funktionalität 277
- Anwender 109
 - anonymer* 246
 - Benutzerausgabe* 330
 - Benutzereingabe* 43, 244
 - Interaktion* 296
 - Kennung* 61
 - Konto sperren* 119
 - sicherheitsbewusster* 270
- Anwendername
 - Administrator* 109
 - hart codierter* 61
- Anwenderschnittstelle
 - Browser* 43
 - grafische* 114
- Anwendung
 - Drittanbieter* 334
 - eigenentwickelte* 334
- API 56
- Application Programming Interface 56
- Application-to-Application 59
- AppScan 350
- Arbeitshypothese, falsche 65
- Arbeitsvermittler 321
- Architektur 42
- ASP 314
- ASSERT 152
- ASSERT ID 152
- Assertion 151
 - aktive* 152
 - konfigurierbare* 152
- Asset 53, 89
- ASSIGN 161
- Attachment, Bewerber 34
- Auditor 48
- Aufrufkette 55
- Auftragshistorie 182
- Ausdruck, regulärer 123
- Ausnahmefehler 108, 109
- außenstehender Bewerber 54
- außenstehender Kunde 53
- Austauschprodukt 329
- Authentifizierung, RFC-Aufruf 38
- Authentisierungsverfahren 41
- Authentizität 89
- AUTHORITY_CHECK_TCODE 141
- AUTHORITY-CHECK 151, 267
- automatische Gegenmaßnahme 119
- automatisierte Fehlersuche 245
- automatisierte Mandantenprüfung 293
- Automobilhersteller 329
- Automobilzulieferer 329
- Awareness-Workshop 77

B

- B2B, Szenario 37
- Base64 279
- Basic Authentication 246
- Batch-Datei 108, 127
- Bedienelement 312
- Bedrohung 91
 - Modell* 73
- Behebungskosten 332
- Benutzer → Anwender
- Berechtigung 41
 - Funktionsgruppe* 150
- Berechtigungskonzept 37, 113
 - proprietäres* 144
- Berechtigungsobjekt 138
 - S_C_FUNCT* 204
 - S_DATASET* 204, 287
 - S_GUI* 289, 290
 - S_LOG_COM* 202
 - S_PATH* 287
 - S_RFC* 143, 149, 315
 - S_TCODE* 139
- Berechtigungsprüfung 139
 - Anwendername* 43
 - fehlende* 43, 261
 - RFC* 43
 - zur Laufzeit* 33
- Berechtigungsstufe, Funktionsgruppe 150
- Best Practices 72

Bestätigungsticket 111
 Bestellung 246
 Anforderung 41
 Nummer 132
 Seite 223
 Betrieb 42
 Betriebsblindheit 76
 Betriebssystemkommando 55, 203
 Bewerber
 außenstehender 54
 Daten 73
 Datensatz 323
 Bewerbung
 interne 326
 Prozess 323
 Bildschirmmaske 269, 297
 binäre Daten 115
 Bitfeld 115
 Blacklist 29, 119
 Bookmark 280
 böses Skript 36
 Broken Image 250
 Browser 221
 Add-on 244
 Ausführen von Schadcode 221
 Cache 220
 Firefox 218
 Schwachstelle 218
 Sperrmechanismus 261
 Standardfunktionalität 226
 Zertifikat 246
 Brute Force 136
 BSI-MM 99
 BSP, forceEncode 243
 Buffer Overflow 191, 285
 Building Security In Maturity Model 99
 Burp Suite 349
 Business Case 100
 Business Server Pages 243

C

C/C++ 191
 CALL 'SYSTEM' 204
 CALL FUNCTION 166
 CALL METHOD 166
 CALL TRANSACTION 138
 Cascading Style Sheets → CSS
 C-Call 195
 Change Request 64
 Checkpoint-Gruppe 156
 Checksumme 118
 Checkvariante 83
 CL_BSP_SERVER_SIDE_COOKIE 268, 307
 CL_GUI_FRONTEND_SERVICES 205, 288
 CL_HTTP_ENTITY 288
 CL_HTTP_EXT_ITS 309
 CL_HTTP_UTILITY 237, 302, 303
 CL_VSI 289
 Client
 falsche Annahme 214
 korrektes Verhalten 213
 clientseitige Prüfung 268
 Client-Server-Architektur 55
 Client-Side Framework 36
 Cloud-Computing 58, 335
 Code
 generischer 157, 167
 Review 99
 Scanner 81
 Code Inspector 143, 197
 Codeanalyse, statische 83
 Code-Audit, manuelles 78
 CodeProfiler 348
 Command Execution 202, 207
 Command Injection 203, 207
 Common Weakness Enumeration 344
 Compliance 33, 171
 Freiheitsstrafe 33
 Verstoß 142, 335
 CONCATENATE 186
 Cookie 147, 220, 260
 Manipulation 263
 Server-Side 268
 CRM 117
 Cross-Site Scripting → XSS
 CSF 36
 CSS 227
 Datei 252
 CSV
 Datei 127
 CWE/Sans Top 25 344
 Cyber-Terrorismus 39

D

- Darstellung, polymorphe 82
- Darstellungsfehler, Encodierung 129
- Data Dictionary 299
- Datei
 - Schnittstelle* 286
 - Verarbeitung* 185
 - Zugriff* 59
- Dateipfad
 - absoluter* 186
 - relativer* 186
- Daten
 - binäre* 115
 - Datenfluss* 80
 - Datenskandal* 34
 - Diebstahl* 74, 329
 - Encodierung* 126
 - Hygiene* 295
 - Indirektion* 189
 - Korrektheit* 58
 - nicht vertrauenswürdige Quellen* 59
 - personenbezogene* 235
 - schadhafte* 59
 - unerwartete* 119
 - Validierung* 116
 - Verlust* 108
 - von Mitarbeitern* 321, 322
- Datenaustausch 42
 - Definition des Formats* 58
 - interaktiver* 58
- Datenbank
 - Hersteller* 291
 - ID* 294
 - Plattform* 293
 - spezifisches SQL-Kommando* 292
 - Treiber* 184
 - Zugriff* 61
- Datenschutz
 - Anforderung* 325
 - Bestimmung* 70
 - Gesetz* 189
- Datenschutzgesetz 103
- Datenvalidierung, Stolperfalle 121
- DDIC-Objekt 83
- deaktivierte WAF 31
- Debugging, Anweisung 323
- Defacement 235
- De-facto-Standard 344
- Defense in Depth 110, 141
- DELETE 179, 294
- DELETE DATASET 189
- DELETE READ DATASET 189
- demilitarisierte Zone 230
- Denial of Service 92, 158, 282
- Design
 - Entscheidung* 75
 - Fehler* 223
 - Review* 76
- Design Pattern 75
- Designmuster, verbotenes 62
- DIAG
 - Nachricht* 300
 - Protokoll* 296
- Dienst, internetfähiger 316
- Dienstreise 325
- digitale Signatur 95, 301
- DMZ 230
- Document Object Model 233
- DOM 233
- DOM-basiertes XSS 233
- Domäne, Feld 299
- Drittanbieter 265
 - Anwendung* 334
- dynamische Funktionalität 261
- dynamische Klausel, Open SQL 175
- dynamischer Inhalt 303
- dynamischer Link 245
- dynamisches ABAP 170
- Dynpro 46, 296

E

- EarlyWatch Alert 286
- Effektivität, Maßnahme 101
- Eigenentwicklung
 - Anwendung* 334
 - kundenspezifische* 39
 - Verantwortung* 39
- Eingabe
 - manipulierte* 38
 - TAN* 111
- Eingabewert, Prüfung 107
- eingeschränkte Funktionalität 36
- Einkaufsabteilung 40
- Einschleusen, schadhafte Daten 59
- Einschränkung, gestalterische 312
- Einspielen, Support Package 193

Elevation of Privileges 92
 E-Mail
 manipulierte 233
 Spam 271
 Employee Self-Services 270, 325
 Encodierung
 Bibliothek 75
 Darstellungsfehler 129
 falsche 130
 korrekte 244
 Überoptimierung 129
 Entwickler
 Experte für Business 36
 Verantwortung 42
 Entwicklung
 Kosten 73, 101
 Paradigma 46
 Richtlinie 64
 Stil 331
 System 62
 Entwicklungsprojekt
 externes 42, 331
 unterfinanziertes 97
 Entwicklungsprozess 67, 99
 ABAP 42
 Feedback 66
 Entwurfsmuster 75
 Erfahrung 39
 Ergonomie, Anforderung 105
 Erkennungsrate 245
 Erpressung 322
 erschwerte Kommunikation 64
 Erwartungshaltung, falsche 66
 Erwartungswert, Schwachstelle 100
 Erweiterung 330
 Quellcode 53
 ESS 270, 325
 Event-Handler, OnInputProcessing 305
 Eventualfall 65
 Exit-URL 275, 302
 Exploit Me, Add-on 245
 externe News-Seite 254
 externer Verweis 330
 externes Entwicklungsprojekt 42, 331
 externes User Interface 315

F

Fachabteilungsleiter 49
 falsche Annahme 35
 falsche Arbeitshypothese 65
 falsche Encodierung 130
 falsche Erwartungshaltung 66
 Falscheingabe 105, 109
 Falschinformation 235
 Feedback, im Entwicklungsprozess 66
 fehlende Berechtigungsprüfung 43, 261
 Fehler
 Behandlung 56, 109
 Design 223
 falsche Encodierung 130
 Fehlermeldung 58, 109, 119
 fehleranfällige Implementierung 40
 Fehlerbehebung, im Produkktivsystem 68
 Fehlersuche, automatisierte 245
 Fehlverhalten, menschliches 321
 Feld, Domäne 299
 Feldwert 299
 Fernglassymbol 78
 Filterregel, WAF 36
 Finanzdaten 56, 57
 Finanztransaktion 246
 Nachvollziehbarkeit 92
 Firebug 348
 Firefox 218
 Firewall 28
 definierter Zugangspunkt 28
 IP-Adresse 28
 Port der Applikation 28
 Status des Anwendungsprotokolls 28
 Firmengeheimnis, Verlust 101
 Flash-Film 246
 Flüchtigkeitsfehler 96
 forceEncode 305
 Forceful Browsing 261
 Variante 261
 Format String
 Angriff 192
 Schwachstelle 191
 Format, Datenaustausch 58
 Formulardaten 250
 Foxyproxy 349

Frameset 272
 generisches 252, 272, 274
 Freemind 347
 Freiheitsstrafe 33
 Freitextfeld 327
 Fremdanwendung 72
 Frontend-Technologie 328
 Funktion
 sichere 41
 undokumentierte 194
 xypass 193
 funktionale Anforderung 38, 68
 Funktionalität
 dynamische 261
 eingeschränkte 36
 Funktionsaufruf, generischer 168
 Funktionsbaustein 46
 AUTHORITY_CHECK_TCODE 141
 GUI_UPLOAD 288
 SXPG_CALL_SYSTEM 202
 SXPG_COMMAND_EXECUTE 203,
 207
 WS_EXECUTE 205
 Funktionsgruppe, Berechtigung 150
 Fuzzing 85

G

ganzheitliche Sicherheit 34
 Garantieanspruch 328, 329
 Garantiebeschreibung 330
 Gegenmaßnahme, automatische 119
 Gehaltsdaten 74, 104
 Gehaltsinformation 235
 Gehaltszahlung 326
 Geheimschrift 112
 gemeinsames Zeitfenster 64
 GENERATE SUBROUTINE-POOL 172
 generische Sicherheitsanforderung 69
 generischer Code 157, 167
 generischer Funktionsaufruf 168
 generisches Frameset 252, 272, 274
 Gesamtkosten 31
 Geschäftsanwendung 55
 fehlerhafte Logik 38
 Geschäftsgeheimnis 73, 271
 Geschäftsleitung 49
 Geschäftsprozess 44
 automatisierter 93

Geschäftsprozess (Forts.)
 lückenlose Dokumentation 71
 gestalterische Einschränkung 312
 GET-Parameter 257
 Going-Live 103
 Governance, Risk and Compliance 33
 grafische Anwenderschnittstelle →
 GUI
 Graphical User Interface → GUI
 GRC 33
 Grundkenntnis, Sicherheitstraining
 226
 Grundschutz 29, 71, 72
 Handbuch 71
 SAP-System 53
 GUI 114, 289
 Download 289
 Upload 289
 Guided Activity 265
 Gutscheincode 223

H

Hacker 39
 Contest 77
 Hailstorm 350
 hart codierter Anwendername 61
 Hash-Wert 194
 Headhunter 321
 Hidden Field 220, 257, 264
 Hintertür 33, 56, 61, 72, 144, 147,
 172, 323
 HTML 216
 deaktiviertes UI-Element 265
 HTML Business 310
 Steuerzeichen 127, 230
 HTTP 29
 Body 280
 Content-Type 247
 Handler 301
 Header 258, 280
 POST 260
 Referrer 226, 258
 Request 216
 Response 216
 HttpFox 348
 HTTPS 219, 255
 Warnmeldung 219
 Httpwatch 349

I

IAC 309
 ICF 301
 ICM 301
 IDS 30, 224
 IF_HTTP_RESPONSE 283
 Implementierung, fehleranfällige 40
 Inbetriebnahme 86
 Index 132
 Indirektion 131
 Industriespionage 62, 109
 Industriestandard 48
 infizierte Webseite 236, 251
 Informatikhochschule 95
 Information Disclosure 92
 Infrastruktur 65, 235
 Inhalt, dynamischer 303
 Innentäter 34
 INSERT 179, 294
 INSERT REPORT 172
 Insider 316
 Inspektion 83
 Integrität 89
 interaktiver Datenaustausch 58
 interne Bewerbung 326
 interne Revision 65
 internes Wissen, Angreifer 106
 Internet Application Components 309
 Internet Communication Framework 301
 Internet Communication Manager 301
 Internet Sales 92
 internetfähiger Dienst 316
 Intranet 213
 Angriff 335
 Kompromittierung 32
 Intrusion Detection System 30, 224
 ISA-Server 54

J

Java Connector 148, 314
 Java-Stack 148
 JCo 148, 314
 Journalist 326

K

Kampagne 328
 Kanonisierung 117
 Kernel-Methode 195
 Kernursache 34
 Keylogging 236
 Klasse 46
 CL_BSP_SERVER_SIDE_COOKIE 268, 307
 CL_GUI_FRONTEND_SERVICES 205, 288
 CL_HTTP_ENTITY 288
 CL_HTTP_EXT ITS 309
 CL_HTTP_UTILITY 237, 302, 303
 CL_VSI 289
 Kommando, verbotenes 62
 Kommunikation
 erschwerte 64
 verschlüsselte 38
 Komplexität 40, 96
 Kompromiss 331
 Kompromittierung, Intranet 32
 Konfiguration 65
 Assertion 152
 Fehler 69, 108
 Konkurrent 329
 Kontodaten 248, 326
 Kontrollfluss 80
 korrekte Encodierungsfunktion 244
 Korrektheit, Daten 58
 Kosten
 Entwicklung 101
 Fehler im Produktivsystem 68
 operative 100
 senken 101
 Test 101
 Tool-Einsatz 100
 Kreativität 39
 Kreditkarten
 Daten 271
 Industrie 71
 Nummer 38, 136
 Transaktion 71
 Krimineller 333
 Kritikalität 98
 Kulturkreis, unterschiedlicher 64
 Kunde
 außenstehender 54

Kunde (Forts.)
 Kundenliste 73
 Kundennummer 329
kundenspezifische Eigenentwicklung
 39
Kündigung 62
Kurzdump 109, 185

L

Lagerverwaltung 41
Laufzeit, unerwünschter Nebeneffekt
 55
Least-Privilege 77
LEAVE TO TRANSACTION 138
Lebenslauf 322
Lebenszyklus 70
Lernmodus, WAF 29
Lesezeichen 280
Lieferant 53
Link, dynamischer 245
Location-Header 280
Log-Datei 110, 220
Login-Seite 280
Lösung, proprietäre 113
Luhn-Check 136

M

Mandantenprüfung, automatisierte
 293
manipulierte Eingabe 38
manipulierte E-Mail 233
manuelles Code-Audit 78
Marketing 328
marktübliche Anforderung 34
Maßnahme, Effektivität 101
Medikament 329
Mehrkosten 64, 332
menschliches Fehlverhalten 321
Methode 46
Metrik 66
Microsoft Silverlight 272
MIME 260
minimiertes Risiko 99
Missverständnis 64
Mitarbeiterdaten 321, 322
MODIFY 179, 294

Modus
 stateful 307
 stateless 306
MSSQL 291
Mythos, Unbreakable-Mythos 55

N

Nachvollziehbarkeit 146
 Finanztransaktion 92
Namenskonvention 72
Native SQL 173, 291
Nebeneffekt, unerwünschter 55
Netzwerkschnittstelle 114
Neuimplementierung 68
News-Seite, externe 254
nicht funktionale Anforderung 68
NoScript 349

O

objektorientierte Programmierung 54
obskure Sicherheit 106
öffentliches Portal 252
Offshoring 64
OK-Code 300
OnInputProcessing 305
Online-Banking 111, 270
Online-Dokumentation 41
Online-Formular 330
Online-Shop 223, 329
OPEN CURSOR 179
OPEN DATASET 186, 203
 FILTER 203
Open SQL 173, 291
 dynamische Klausel 175
 plattformunabhängiger Zugriff 55
Open Web Application Security
 Project 71, 253
operative Kosten 100
Outsourcing 64
OWASP 71, 253

P

PAI 298
Parsen 217
Partnerfirma 53

Passwort
 Administrator 109
 Feld 225
 schwaches 323
 Speichern im Klartext 225
 Patch, einspielen 53
 pauschales Risiko 71
 Payment Card Industry Data Security
 Standard 71
 PBO 297
 PCI-DSS 71
 PDF 247
 Datei 220
 Peer Review 78
 Penetrationstest 63, 69, 98
 Blackbox 86
 Grenze 66
 Konzeption 66
 Whitebox 85
 PERFORM 166
 persistiertes XSS 232
 Personalabteilung 322
 Personalwesen 325
 Personalwirtschaft 321
 personenbezogene Daten 57, 235,
 325
 Pflichtenheft 48
 Phishing, Seite 277
 PHP 314
 PI 59, 286, 315
 plattformunabhängiger Zugriff, Open
 SQL 55
 Plausibilitätsprüfung 117
 Plug-in 218
 polymorphe Darstellung 82
 Portal
 Komponente 256
 öffentliches 252
 Portscan 235
 Prepared Statement 174
 Presse, Datenskandal 34
 Problembewusstsein 40
 Process After Input 298
 Process Before Output 297
 Product Security Standard 78
 Produktivdaten 327
 Produktivsystem 61, 62, 324
 Profilergenerator 143

Profilparameter
 auth/object_disabling_active 142
 auth/rfc_authority_check 143
 auth/system_access_check_off 143
 Programm 46
 Programmierfehler 60
 Programmiersprache, typisierte 115
 Programmierung
 objektorientierte 54
 prozedurale 54
 Programming by Contract 151
 Projektleiter 49
 proprietäre Lösung 113
 proprietäres Berechtigungskonzept
 144
 Protokoll
 HTTP-basiert 29
 XML-basiert 28
 Prototyp 82
 Proxy-Cache 282
 prozedurale Programmierung 54
 Prozess
 Logik 323
 reaktiver 65
 Überwachung 66
 Prüftabelle 299
 Prüfung
 clientseitige 268
 Eingabewert 107

Q

QA-Prozess 31
 Qualität
 Anforderung 68
 Sicherung 244
 Quality Gate 63
 quantitatives Risiko 101
 Quellcode, Erweiterung 53

R

Rabatt 328
 Rache 325
 Rainbow Table 136, 194
 reaktiver Prozess 65
 Reaktivierung, UI-Element 269
 Rechenzeit 58
 reflektierendes XSS 232

- Regelbasis, Virensscanner 31
 - Regressanspruch 40
 - Regression 63
 - Test* 86
 - regulärer Ausdruck 123
 - regulatorische Vorgabe 32
 - Reifegrad 42, 62, 101
 - Programm* 62
 - Sicherheitsprozess* 65
 - Reisekosten 325
 - relativer Dateipfad 186
 - Remote Function Call → RFC
 - REPLACE ALL OCCURRENCES 122
 - Report 46
 - Repudiation 92
 - Reputation 34
 - Schaden* 323
 - Verlust* 322
 - Return on Investment 31
 - Reverse Engineering 298
 - Revision, interne 65
 - Revisor 48
 - RFC 148
 - Aufruf* 108
 - Authentifizierung* 38
 - Integration in ABAP* 55
 - Trusted System* 149
 - Trusting System* 149
 - Verschlüsselung* 38
 - Zugriff* 37
 - Rich Client 335
 - Risiko
 - Analyse* 90
 - Bewusstsein* 49
 - Minimierung* 99
 - pauschales* 71
 - quantitatives* 101
 - ROI 31
 - Rolle 41
 - rote Ampel 98
 - Rücksprungadresse 192, 252, 275
 - Rückwärtsnavigation 81
 - Rüstkosten 101
- S**
-
- Sales 328
 - Sans 196
 - SAP
 - Grundschutz* 53
 - Internet Sales* 328, 330
 - ISA-Server* 54
 - Kernel* 191
 - Product Security Standard* 78
 - Security Guides* 41
 - Security Notes* 234, 286
 - Support* 39
 - System-ID* 148
 - Theme* 313
 - SAP Cryptolib 347
 - SAP Customer Relationship Management 117
 - SAP Developer Network 84
 - SAP E-Recruitment 320
 - Attachment* 34
 - SAP GUI 213, 296
 - Reverse Engineering* 298
 - SAP Interactive Forms by Adobe 272
 - SAP NetWeaver Application Server ABAP 267
 - SAP NetWeaver Business Client 213
 - SAP NetWeaver Process Integration 59, 286, 315
 - SAP NetWeaver, Konfiguration 69
 - SAP Service Marketplace 148
 - SAP Virus Scan Interface 31
 - SAP-Standard
 - Anforderung an sichere Software* 39
 - Erweiterung* 27
 - Modifikation* 27
 - SAP-System, Anmeldung 59
 - Sarbanes-Oxley Act 71, 92
 - Schadcode 282
 - Schadenspotenzial 57, 76, 90, 98
 - XSS* 228
 - Schadenssumme 101
 - schadhafte Daten, einschleusen 59
 - Schlagzeile 38
 - Schutzbedarf 90, 333
 - falsche Priorität* 93
 - Klasse* 90
 - Schutzziel 43
 - CIA* 89
 - Schwachstelle
 - Browser* 218
 - polymorphe Darstellung* 82
 - Symptom* 30

- Schwachstelle (Forts.)
 - Ursache* 30
- Screen 296
 - Darstellung* 312
- SCRUM 67
- SDL 72
- SDN 84
- Secure Development Lifecycle 72
- Secure Network Communication 38, 297
- Secure Socket Layer 219
- Secure Store and Forward 301
- Security by Obscurity 106
- Security Engineering 42
- Security Guides 41
- Security Pattern 75
- Security Response 66, 100
- SELECT 179
- Service 328
 - Anwender* 59
- serviceorientierte Architektur 335
- Session, Cookie 248
- Shortdump 109, 185
- Showstopper 74
- sichere Funktion 41
- sichere Software
 - marktübliche Anforderung* 34
 - SAP-Standard* 39
- Sicherheit
 - Anforderung* 32, 69
 - Bewusstsein* 66, 73
 - Funktion* 334
 - ganzheitliche* 34
 - Infrastruktur* 65
 - Konferenz* 114
 - Konfiguration* 65
 - Mindestmaß* 41
 - obskure* 106
 - Patch* 333
 - Prinzip* 102
 - Schulung* 95, 114
 - Überprüfung* 30
 - Update* 53
 - Verständnis* 35
- sicherheitsbewusster Anwender 270
- Sicherheitsfehler, ABAP-Kontext 41
- Sicherheitslücke, Top Ten 71
- Sicherheitsorganisation
 - OWASP* 71
- Sicherheitsorganisation (Forts.)
 - Sans* 196
 - WASC* 196
- Sicherheitsproblem
 - ABAP* 34
 - Programmiersprache* 34
- Sicherheitsprozess, Reifegrad 65
- Sicherheitstest 105
 - Angriffssimulation* 38
- Sicherheitstraining, Grundkenntnis 226
- SID 148
- Signatur, digitale 95, 301
- Simulation, Angriff 38
- Single Sign-on 41, 220
- Skript, bössartiges 36
- Smart Client 213
- SNC 38, 297
- SOA 335
- SOAP 59, 315
- Softwareabnahme 72
- Softwareentwicklung, agile 82
- SOX 71, 92
- Spam-E-Mail 271
- Speicherbereich 192
- Sperre, Anwenderkonto 119
- Spezifikation 42, 68
 - Anforderungseigenschaft* 68
 - funktionale Anforderung* 68
 - nicht funktionale Anforderung* 68
 - unvollständiges Testen* 94
- Spidering-Algorithmus 245
- Spiralmodell 67
- Spoofing 92
- SQL 173, 291
 - datenbankspezifisches Kommando* 292
 - Dialekt* 292
 - Sprachumfang* 293
 - Standardbefehlssatz* 291
 - Statement manipulieren* 181
- SSF 301
- SSL 219
- SSO2-Ticket 220
- Stammdaten 325
 - Kunden* 328
- Standard
 - Abweichung* 99
 - Anpassung* 100

- Standard (Forts.)
 - sichere Software* 99
 - Startseite 313
 - Stateful-Modus 307
 - Stateless-Modus 306
 - statische Codeanalyse 83
 - Stellenausschreibung 321, 325
 - Steuerzeichen 107
 - HTML* 127
 - SQL-String* 183
 - Stolperfalle, Datenvalidierung 121
 - Streitigkeit 62
 - STRIDE 92
 - Structured Query Language → SQL
 - Stylesheet 246, 313
 - SUBMIT 166
 - Substitutionschiffre 112
 - Support Package 191, 193
 - Symptom, Schwachstelle 30
 - System
 - Absturz* 108
 - Härtung* 53
 - Kommando* 208
 - Name* 109
 - Stillstand* 33, 101
 - SYSTEM-CALL 195
 - SYST-UNAME 144
 - SY-SUBRC 141
 - SY-UNAME 144
- T**
-
- Tabbed Browsing 218
 - Tabelle
 - HTTP_WHITELIST* 302
 - Protokollierung* 292
 - SPTH* 287
 - TDCOUPLES* 140
 - Tamper Data 245, 348
 - TamperIE 349
 - Tampering 92
 - Tastatureingabe 236
 - technisches Versagen 108
 - Termindruck 96
 - Test 42
 - Anwender* 182
 - Betrieb* 244
 - Funktionalität* 33
 - Kosten* 101
 - Test (Forts.)
 - System* 62
 - testbare Anforderung 70
 - Theme 252
 - Threat Modeling
 - STRIDE* 91
 - Tool* 347
 - Tippfehler 108
 - Top Ten
 - OWASP* 71, 253
 - Sicherheitslücke* 71
 - Transaktion
 - AUTH_SWITCH_OBJECTS* 142
 - PFCG* 143
 - SAAB* 152, 156
 - SCI* 72, 83
 - SE11* 83
 - SE16* 139
 - SE24* 83
 - SE37* 83
 - SE38* 83
 - SE80* 78, 81, 138
 - SE93* 139
 - SE97* 141
 - SICF* 269, 303
 - SM30* 141
 - SM49* 198, 207
 - SM69* 198, 207
 - ST05* 174
 - SU24* 72, 143
 - Transaktionspaar 139
 - TRANSFER 189
 - Transportauftrag 83
 - Trusted System 149
 - Trusting System 149
 - typisierte Programmiersprache 115
- U**
-
- Überoptimierung, Encodierungsfunktion 129
 - Überwachung, Prozess 66
 - UI-Element
 - Look and Feel* 313
 - reaktiviertes* 269
 - Unbreakable-Mythos 55
 - undokumentierte Funktion 194
 - unerwartete Daten 119
 - unerwünschter Nebeneffekt 55

- Unified Rendering 312
- unterfinanziertes Entwicklungsprojekt 97
- Unternehmenskultur 32, 99
- Unternehmenswert 53, 89
- unterschiedlicher Kulturkreis 64
- Unwissen 95
- UPDATE 179, 294
- URL 216, 259
 - Encoding* 233
 - Filter* 256
 - Parameter* 313
- Urlaubsanfrage 246, 325, 326
- Ursache, Schwachstelle 30
- USB-Stick 54
- Use-Case 38
- User Interface, externes 315

V

- veränderte Anforderung 107
- Verantwortung, Entwickler 42
- Verarbeitung, Datei 185
- Verbindlichkeit 89
- Verbindung, verschlüsselte 270
- verbotenes Designmuster 62
- verbotenes Kommando 62
- verbotenes Zeichen 121
- Verfügbarkeit 89, 158, 189
- vergrößerte Angriffsoberfläche 57
- Verkaufsstatistik 328
- Verlust, Firmengeheimnis 101
- Versagen, technisches 108
- verschlüsselte Kommunikation 38
- verschlüsselte Verbindung 270
- Verschlüsselung 41, 112
 - Algorithmus* 136
 - Lösungen* 38
 - RFC-Aufruf* 38
 - Verbindung* 270
- verteidigungslinie 60
- vertrauenswürdiger Administrator 213
- Vertraulichkeit 89
- Verweis, externer 330
- Verwendungsnachweis 78
- Video
 - XSRF* 251
 - XSS* 228

- Vier-Augen-Prinzip 78
- Virens Scanner 31, 41, 322
 - explizite Programmierung* 288
 - Regelbasis* 31
- Virtual Private Network 325
- Virus Scan Interface 31
- Vorgabe, regulatorische 32
- Vorgehensmodell 42
- Vorwärts-Rückwärts-Navigation 78
- VPN 325
- VSI 31

W

- WAF 29, 36, 224
 - Best Practices* 29
 - deaktivierte* 31
 - Filterregel* 36
 - Grundschutz* 29
 - komplementäres Werkzeug* 30
 - Lernmodus* 29
- Wahrscheinlichkeit, Sicherheitsfehler 35
- Warenkorb 223, 264, 283
- Wartezeit 58
- Wartung 86
- WASC 196
- Wasserfallmodell 67
- Web 2.0 58, 335
- Web Application Firewall → WAF
- Web Developer 348
- Web Dynpro ABAP 312
- Web Reporting 309
- Web Vulnerability Scanner 349
- WebGUI 298, 309
- WebInspect 350
- WebRFC 309
- Web-Scarab 350
- Webschnittstelle 333
- Webseite, infizierte 236, 251
- Web service 37
- Webtransaktion 309
- Wertebereich 115, 117, 118
- Wettbewerber 54, 342
- WGate 309
- What you see is what you get 230
- Whitebox-Tool
 - Application Intelligence Platform* 348

Whitebox-Tool (Forts.)
 Code Inspector 347
 CodeProfiler 348
Whitelist 29, 119
Wiederverwendbarkeit 156, 167
Wireless Markup Language 312
Wireshark 350
Wirtschaftsprüfung 65
WML 312
Worst-Case 109
WRITE 161
WYSIWYG 230

X

XI → SAP NetWeaver Process
 Integration
XML, Protokoll 28
XSRF 236
 Video 251
XSS 232
 Cheat Sheet 234
 DOM-basiertes 233
 Einschleusen von Schadcode 234

XSS (Forts.)
 Kurzfilm 228
 persistiertes 232
 reflektierendes 232
 Schadspotenzial 228
 XSS Me 245
xxpass 193

Z

Zahlungsinformation 315
Zahlungslauf 41
Zeichen
 alphanumerisches 120
 verbotenes 121
Zeitdruck 82
Zeitfenster, gemeinsames 64
Zeitmangel 96
Zeitstempel 131
Zeitzone 64
zielgerichteter Angreifer 109
Zugriff, plattformunabhängiger 55
Zugriffsmöglichkeit 56
Zugriffsschutz 89
Zulieferer 40