Andreas Wiegenstein, Markus Schumacher,
Sebastian Schinzel, Frederik Weidemann

Sichere ABAP™-Programmierung

Galileo Press

Bonn • Boston

Auf einen Blick

Inhalt

Teil II Anwendung und Praxis

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 Programmiertechniken 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.

Alle Eingaben können gefährliche Daten enthalten

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.

Sicherheitsfunktionen können nicht oder falsch verwendet werden

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.

[*] 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?

Sicherheitsprobleme 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.

Sicherheitslücken können zu ungewünschter Funktionalität führen

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.

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-Applikationsserver 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`.

Berechtigungskonzepte 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.

Manipulation ist trotz Berechtigungskonzept möglich

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.

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

Verwenden Sie immer explizite Berechtigungsprüfungen

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.

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.

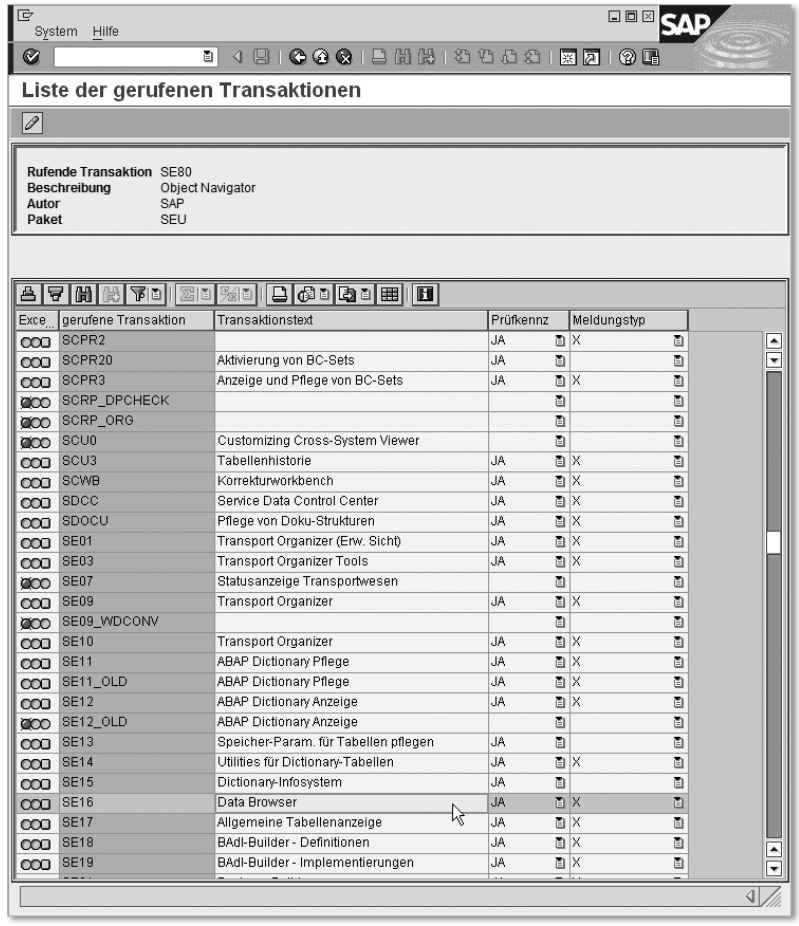

Abbildung 5.1 Pflege von Transaktionspaaren in Transaktion SE93

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 `TCDCOUPLES` 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 Berechtigungsprüfungen 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 müssten 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

Abfragen auf hart codierte Benutzernamen können die Ablaufslogik verändern

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.

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 usname = sy-uname.
  ENDSELECT.
  IF sy-subrc = 0.
*   backdoor goes here
  ENDIF.
ENDMETHOD.
```

Listing 5.4 Verwaltung von hart codierten Benutzernamen in Tabellen

Manchmal werden hart codierte Benutzernamen in Tabellen verwaltet

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,

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 Berechtigungsprüfungen 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.

Es kann unterschiedliche proprietäre Berechtigungsprüfungen geben

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.

[+]

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.

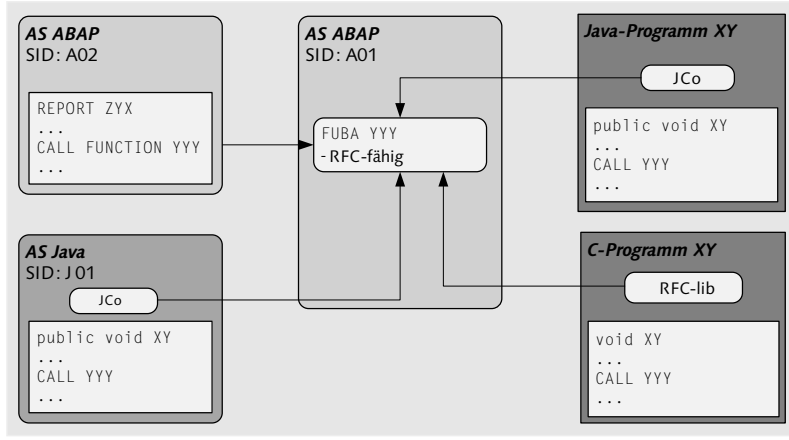

Abbildung 5.2 RFC-Szenario

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.

Jeder kann RFC-Aufrufe starten

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.

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.

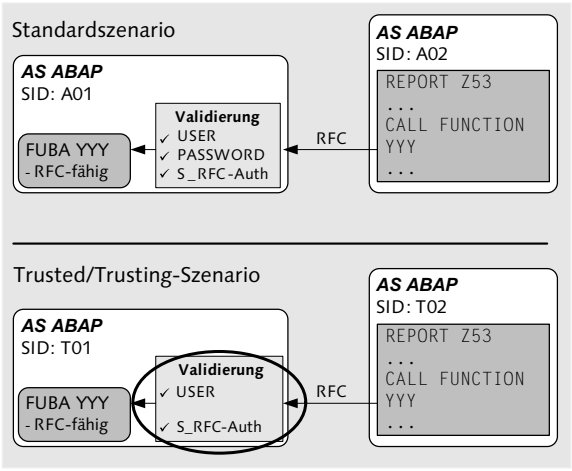

Abbildung 5.3 Sicherheitsprüfungen in RFC-Szenarien

5.3.1 Anatomie der Schwachstelle

Die RFC-Prüfung erfolgt nur auf Funktionsgruppen

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

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 Aufrundendem besteht. Datentypen von Parametern können zum Beispiel in einer Schnittstelle als explizite Anforderung hinterlegt werden.

Mit Assertions können Anforderungen feingranular geprüft 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

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.

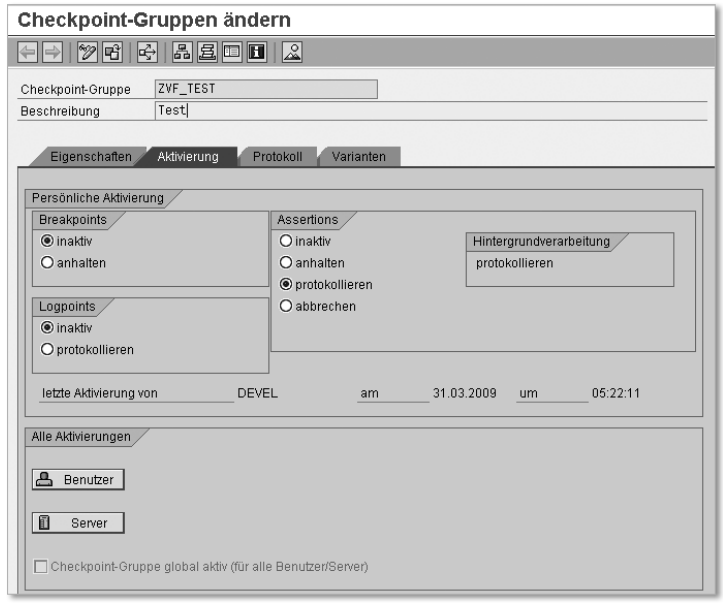

Abbildung 5.4 Konfiguration von Checkpoint-Gruppen für Assertions

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 wird Code abhängig von der Konfiguration ausgeführt

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.

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.

Assertions können schwerwiegende Sicherheitsprobleme auslösen

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.

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

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 Produktivsystem 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`.

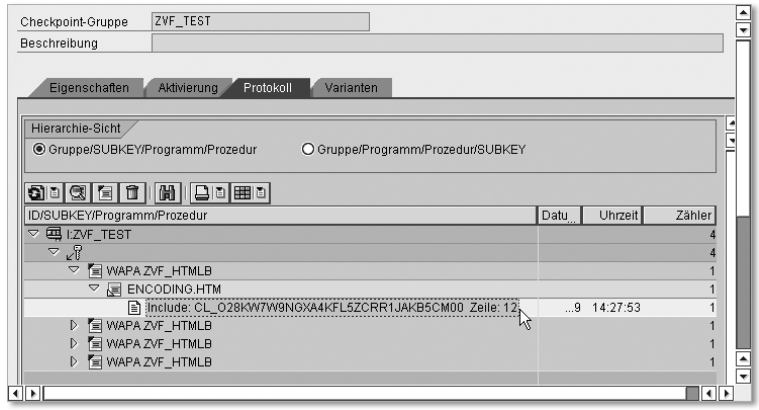

Abbildung 5.5 Protokoll-Eintrag einer Checkpoint-Gruppe

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.

Flexibilität kann zu unerwünschten Nebeneffekten führen

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.

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.

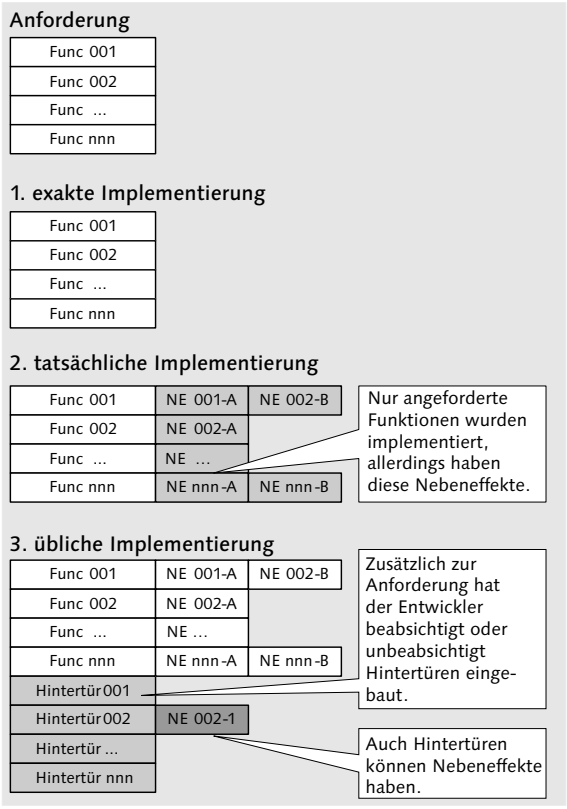

Abbildung 5.6 Logischer Missbrauch vs. Veränderung von Code

Generischer Code kann zu logischem Missbrauch oder semantischen Änderungen führen

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

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.

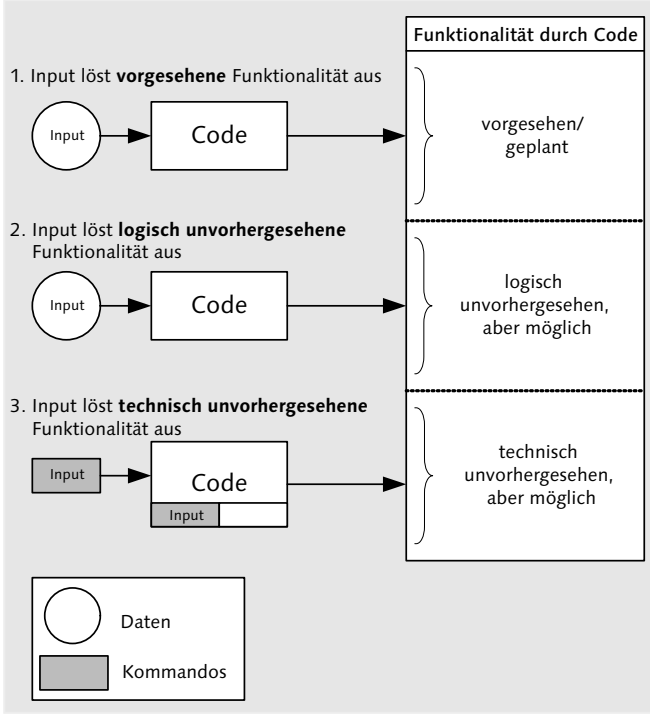

Abbildung 5.7 Ursachen von Nebeneffekten bei generischem Code

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 dynamischen Befehlen steht das Verhalten erst zur Laufzeit fest

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 bösartiger 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).

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 ccdata = 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 instanzierten 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-Interna 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_attrib 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_attrib TYPE string,
  c_class  TYPE string.

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

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

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

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

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z