Ethical Hacking
Die Netz Security

Veröffentlichung: 01/2005.

Defensive Programmierung

Fehler, die erst in der Betriebsphase einer Software auftreten, verursachen unverhältnismäßig hohe Wartungskosten. Mit einer besonders “misstrauischen” Einstellung können Entwickler die Fehlerrate ihrer Software deutlich senken und sich gleichzeitig gegen Angriffe auf die Daten ihrer Applikation schützen. Welche Effekte die Anwendung einer defensiven Programmierung auf Wartungskosten, Qualität und Sicherheit einer Software haben, zeigt dieser Artikel.

Manu Carus

Defensiv zu programmieren bedeutet grundsätzlich, auf allen Ebenen der Software-Architektur so viele Sicherheitsabfragen wie möglich zu implementieren. Auf der Data Tier einer mehrschichtigen Applikation werden in jeder Datenbankprozedur sämtliche Übergabeparameter validiert und alle Voraussetzungen, die für den Aufruf der Prozedur erfüllt sein müssen, explizit überprüft. Gleiches gilt für die implementierten .NET-Klassen in der Business Tier sowie für Eingaben der Benutzer auf der Presentation Tier. Das Ziel dieser Vorgehensweise besteht darin, die Software erheblich zu stabilisieren, damit jede nicht-erfüllte Voraussetzung zu einem Fehlerzustand führt, der in der Datenbank protokolliert wird und die weitere Verarbeitung innerhalb der Applikation verhindert.

Jede Entwicklung unterliegt bestimmten Voraussetzungen. Einige Beispiele aus der Implementierung einer Portalapplikation lauten:

- Data Tier:
  Die Datenbankprozedur *GetItemizedBill()* ermittelt die einzelnen Telefonverbindungen aus dem Einzelverbindungsnachweis einer Kundenrechnung und geht aufgrund der geltenden Datenbankrechte davon aus, dass der Aufrufer berechtigt ist, die Verbindungsdaten für diesen Kunden einzusehen.
- Business Tier:
  Die Methode *Customer.GetInvoiceAddress()* zur Ermittlung der aktuellen Rechnungsadresse eines Kunden liefert stets ein gültiges *Address*-Objekt zurück, niemals jedoch *null* bzw. *Nothing*.
- Presentation Tier:
  Die ASP.NET-Seite *changePaymentData.aspx* überprüft die Benutzereingaben “Kontonummer”, “Bankleitzahl” und “Kontoinhaber” mit Hilfe der ASP.NET-Validation-Controls *RequiredFieldValidator* und *RegularExpressionValidator*. Die nachfolgend aufgerufene ASP.NET-Seite *acknowledgeChangePayment.aspx* zur Bestätigung der geänderten Zahlungsdaten darf erst nach Aufruf der vorgenannten ASP.NET-Seite ausgeführt werden.

Geht man als Entwickler davon aus, dass solche Voraussetzungen stets erfüllt sind, so können sich im produktiven Betrieb der Applikation unerwünschte, ggf. sehr teure Effekte einstellen. Jede Software unterliegt Änderungszyklen. Klassen werden erweitert, Datenbankprozeduren angepaßt und bestehende Geschäftsregeln ausgebaut oder gar aufgehoben. D.h. die Voraussetzungen, von denen ursprünglich ausgegangen wurde, ändern sich innerhalb des Lebenszyklus der Software, und meist finden solche Änderungen ihren Weg nicht in die Entwicklung von Bugfixes oder neuen Programmversionen. Im Produktionsbetrieb der Software können sich dann bspw. folgende Situationen einstellen:

- Data Tier:
  In Version 2 der Portalapplikation sollen nicht nur die Kunden selbst, sondern auch die eigenen Sachbearbeiter des Unternehmens das Portal nutzen, bspw. um telefonische Änderungen des Kunden anzunehmen und über das Portal in die Kundendatenbank einzugeben. Aufgrund mangelnder Überprüfungen der Aufrufberechtigungen in den ASP.NET-Seiten ist es diesen Sachbearbeitern nun möglich, die Einzelverbindungen der Kunden einzusehen, was aus Datenschutzgründen besonders dann unerwünscht ist, wenn Mitarbeiter des Unternehmens selbst Kunden sind.
- Business Tier:
  Bei der Weiterentwicklung des Portals wird eine bestehende Geschäftsregel aufgeweicht: der Kunde muss bei der Registrierung nur noch eine Lieferanschrift eingeben; die Rechnungsadresse selbst soll nicht mehr explizit gespeichert werden, wenn sie mit der Lieferanschrift übereinstimmt. Die Methode *Customer.GetInvoiceAddress()* gibt in diesen Fällen nun *null* bzw. *Nothing* zurück. Wird diese Änderung nicht von allen Aufrufern dieser Methode berücksichtigt, entsteht zur Laufzeit eine *System.NullReferenceException* mit der Meldung "Object reference not set to an instance of an object.".
- Presentation Tier:
  Ein Hacker verschafft sich Zugang zu dem Web-Server, auf dem die Portalapplikation läuft und erhält unberechtigten Zugriff auf alle Assemblies im *\bin*-Verzeichnis der Web-Applikation. Ihm ist es nun möglich, alle verfügbaren .NET-Klassen der Business-Tier durch Disassemblierung einzusehen, ihre Funktionsweise zu verstehen und Skripte zur Nutzung dieser Klassen zu implementieren, und zwar unter Umgehung sämtlicher Validierungen, die ansonsten von den ASP.NET-Skripten in der Presentation Tier vorgenommen würden.

Jede dieser aufgeführten Situationen führt zu steigenden Wartungskosten, wenn die auftretenden fachlichen und technischen Fehler erst im Produktionsbetrieb entdeckt werden. Die Folge sind Bugfixes, wiederholte Tests und produktive Deployments. Je nach Fehlertyp müssen erheblich Zeit und Kosten in die Analyse und Behebung investiert werden. Im Falle des Sicherheitslecks besteht darüber hinaus die Gefahr unberechtigter Zugriffe auf sensible Daten, lesend wie schreibend.

Mit Hilfe einer defensiven Programmierung können die beschriebenen Situationen nicht grundsätzlich verhindert werden. Jedoch ist es möglich, die Eintrittswahrscheinlichkeit von Fehlersituationen zu reduzieren und die Aufwände, die Hacker betreiben müssen, um erfolgreich einbrechen zu können, unverhältnismäßig “hochzuschrauben”. Das Ergebnis ist eine zuverlässigere, stabilere Applikation.

Minas Tirith

Die einzige Konstante in jeder Projektarbeit ist die Tatsache, dass nichts so bleibt wie es ist. Daher lautet einer der wichtigsten Schlüssel zum Erfolg “Mißtrauen”. Entwickler, die sich möglichst defensiv aufstellen, alle Zusicherungen grundsätzlich in Frage stellen und bei jedem Verstoß gegen noch so triviale Voraussetzungen eine Exception auslösen, schaffen stabilere Software-Systeme. Die Wahrscheinlichkeit, dass Fehler ihren Weg bis in die Produktionsumgebung finden oder unerwünschte Seiteneffekte bei der Verarbeitung auftreten können, wird bei dieser Vorgehensweise deutlich reduziert. Produktionsfehler können aufgrund der zahlreichen Fehlerabfragen frühzeitiger gefunden werden.

Was bedeutet es, sich defensiv aufzustellen?

“Herr der Ringe”-Fans kennen aus dem dritten Teil der Trilogie die Stadt “Minas Tirith”, eine schier uneinnehmbare Festung, die im Grenzgebiet liegt und besonders häufig drohender Gefahr ausgeliefert ist. Minas Tirith wurde an einem Bergmassiv angesiedelt, so dass Angriffe nicht aus allen Himmelsrichtungen möglich sind (siehe Abbildung 1). Geschützt wird die Stadt nicht durch einen einzigen Burgwall, sondern durch sieben Befestigungswälle! In jeder Befestigungsmauer gibt es nur ein einziges Tor, das den Übergang in den nächsten Befestigungsring ermöglicht. Allerdings liegen diese sieben Tore nicht in einer Linie hintereinander, sondern wurden versetzt angelegt, so dass jeder Eindringling den Weg in die Stadt im Zickzack nehmen muß. Dabei muß er sich in jedem Befestigungsring bergauf bewegen, da die Stadt auf einem Hügel gebaut ist und der Weg sich mit jeder Ebene weiter hochschlängelt.

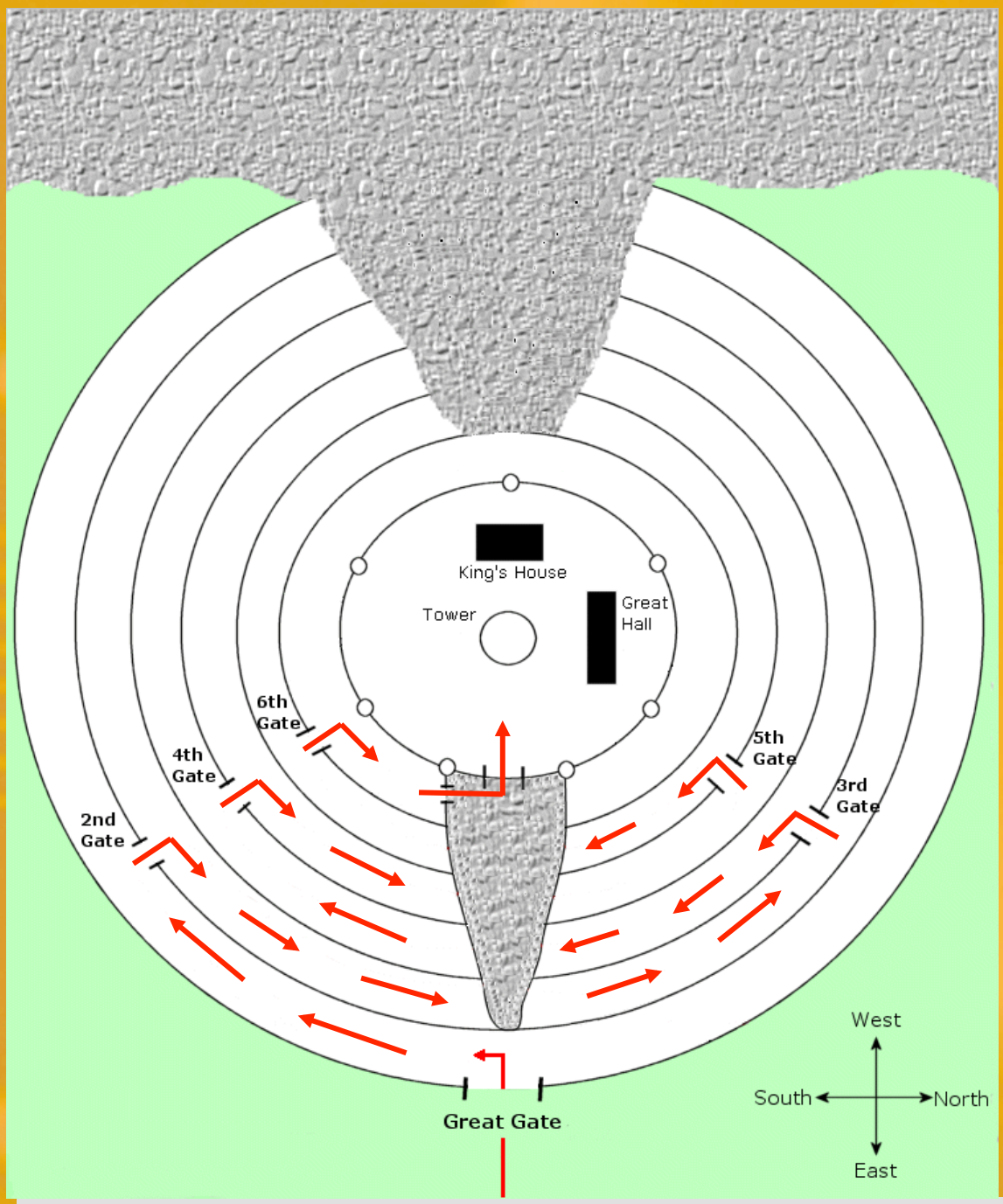

Abb. 1: Minas Tirith

Feinden wird der Zugang in die bewehrte Stadt quasi unmöglich gemacht.

Defensive Software-Architektur

Übertragen auf das Konzept der defensiven Programmierung stellt Minas Tirith das zu schützende Gut einer Software-Applikation dar: die Daten. Sollen diese Daten geändert werden, so ist nur ein einziger Weg von der Präsentationsschicht über die Business Tier bis in die Datenschicht möglich, auf dem die verschiedensten Validierungen vorgenommen werden. Somit wird sichergestellt, dass die Datenänderung ordnungsgemäß durchgeführt werden darf. Es ist kein direkter Zugriff auf die Datenbank möglich. Keine Ebene der Software-Architektur kann übersprungen werden. Alle “Schranken” müssen erfolgreich passiert werden.

Abbildung 2 zeigt eine defensive, mehrschichtige Software-Architektur.

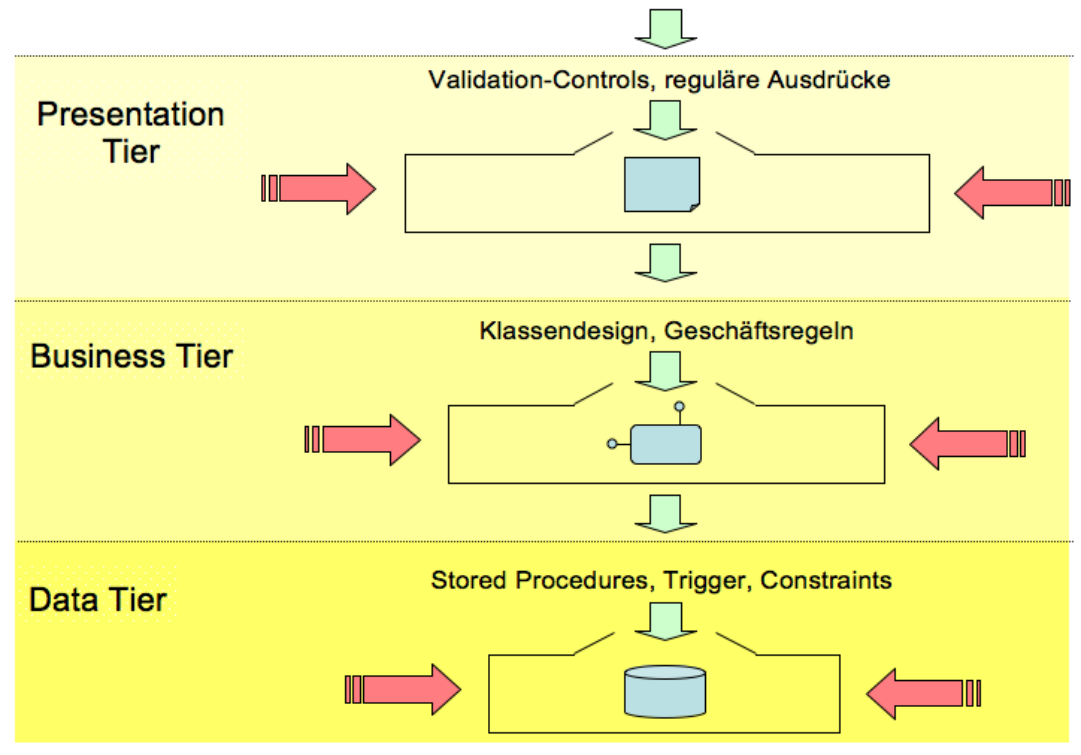

Abb. 2: Defensive Software-Architektur

In der Data Tier werden die Daten mit Hilfe von Constraints, Triggern und Stored Procedures geschützt. Die Constraints sorgen auf der unterst möglichen Ebene für die Einhaltung von Geschäftsregeln; bspw. darf das Login-Datum für einen Portalbenutzer nicht vor dem Erstelldatum seiner Benutzerkennung liegen. Trigger verhindern, dass Daten gelöscht werden können, und dass Daten nur in der vorgesehenen Weise geändert werden können; bspw. darf die Rolle eines Benutzers nachträglich nicht mehr geändert werden. Stored Procedures bilden ein Application Programming Interface (API), eine Schnittstelle, über die Daten gelesen und modifiziert werden können, allerdings nur unter vorheriger Überprüfung aller zugrunde liegenden fachlichen und technischen Geschäftsregeln. Bspw. wird eine Benutzerkennung nach drei mißlungenen Anmeldeversuchen automatisch für eine Stunde gesperrt.

Auf der Business Tier sorgt ein durchdachtes, sorgfältig konzipiertes Klassendesign dafür, dass bestimmte Voraussetzungen erfüllt sein müssen. Bspw. ist der Zugriff auf die Benutzerdaten, d.h. auf die Properties einer Klasse *User*, nur über die Klasse *Session* möglich: es muß zunächst eine benutzerdefinierte Session im System erzeugt und in der Datenbank eingetragen werden, bevor nur eine einzige Funktionalität der Portalapplikation aufgerufen werden kann. Die in der Business Tier implementierten .NET-Klassen überprüfen die ihnen übergebenen Informationen darüber hinaus mit Hilfe besonderer, anwendungsspezifischer Geschäftsregeln.

Die Presentation Tier nimmt letztlich die Benutzereingaben entgegen und überprüft diese möglichst restriktiv unter Einsatz von Validation-Controls und regulären Ausdrücken.

Wichtig in diesem Zusammenhang ist insbesonders, dass nicht nur die Presentation Tier zu verantworten hat, dass nur “saubere” Daten angenommen und an die nachgelagerten Ebenen weitergereicht werden dürfen. Vielmehr müssen auf jeder Software-Ebene, in jedem ASP.NET-Skript, in jeder .NET-Klasse und in jeder

Datenbankprozedur die gleichen Fehlervalidierungen durchgeführt werden. Somit wird verhindert, dass sich nachfolgende Komponenten auf die vorhergehenden Verarbeitungsschritte verlassen und zugrunde liegende Voraussetzungen ausgehebelt werden.

Die roten seitlichen Pfeile in der Abbildung 2 zeigen, dass kein direkter Zugriff auf der jeweiligen Software-Ebene möglich ist.

Möchte ein Administrator bspw. einige Grunddaten der Portalapplikation ändern, so ist diese Änderung nur durch Aufruf der für diesen Zweck vorgesehenen Datenbankprozedur möglich. Diese Prozedur hat neben der Datenänderung die Aufgabe, den Vorgang zu protokollieren, so dass später nachvollzogen werden kann, welcher Benutzer die Daten im System modifiziert hat. Ein direktes INSERT, UPDATE oder DELETE auf den Datenbanktabellen ist grundsätzlich nicht möglich, schon gar nicht unter Umgehung der Trigger und Constraints.

Ein SOAP-Web-Service zur Änderung von Kundendaten bspw. muss die vorgesehenen .NET-Klassen in der Business Tier nutzen und, exakt wie die Portalapplikation selbst, eine benutzerdefinierte Session erzeugen und in der Datenbank protokollieren, bevor auf die Kundendaten zugegriffen werden kann.

Die ASP.NET-Skripte in der Presentation Tier verlassen sich nicht auf die Client-seitige Validierung der Benutzereingaben, sondern überprüfen alle entgegen genommenen Werte server-seitig möglichst restriktiv mit Hilfe regulärer Ausdrücke. Somit werden einfache Angriffsversuche wie HTML-Injection, SQL-Injection und Cross-Site-Scripting verhindert.

Probleme bei der Implementierung einer einfachen Datenbankprozedur

Listing 1 zeigt die typische Implementierung einer einfachen T-SQL-Datenbankprozedur zur Änderung der E-Mail-Adresse eines Benutzers. Als Parameter werden Kennung und E-Mail-Adresse des Benutzers an die Prozedur übergeben. Ein einfaches UPDATE-Statement führt die Änderung durch und überprüft abschließend, ob dabei ein Fehler aufgetreten ist.

Das Problem bei dieser Implementierung ist, dass sich der Entwickler zu sehr darauf verlässt, dass die Datenbankprozedur in der vorgesehenen Weise aufgerufen wird. Wer führt die Datenänderung durch? Ist dieser Jemand überhaupt berechtigt, die Änderung durchzuführen? Ist sichergestellt, dass die Änderung später im System nachvollziehbar ist? Existiert der angegebene Benutzer? Ist dieser Benutzer inaktiv oder gesperrt?

Listing 1: Eine einfache Datenbankprozedur (T-SQL)

```
------------------------
---   SetUserEmail   ---
------------------------
CREATE PROCEDURE portal_owner.SetUserEmail @pUser    NVarChar(25),
                                           @pEmail   NVarChar(50)
WITH ENCRYPTION AS
BEGIN

  UPDATE portaldb.portal_owner.portal_user
  SET    email = @pEmail
  WHERE  login = @pUser

  IF @@ERROR <> 0
  BEGIN
    RAISERROR('-20999:unexpected error', 16, 1) WITH SETERROR, NOWAIT
    RETURN -1
  END

END
GO
```

Defensive Implementierung

Listing 2 zeigt, wie die gleiche Aufgabenstellung defensiv implementiert werden kann.

Listing 2: Eine defensiv implementierte Datenbankprozedur (T-SQL)

```
-----------------------
---   SetUserEmail   ---
-----------------------

CREATE PROCEDURE portal_owner.SetUserEmail @pUser    NVarChar(25),
                                           @pEmail   NVarChar(50),
                                           @pSession NVarChar(36)

WITH ENCRYPTION AS
BEGIN

  SET NOCOUNT ON

  -- check input parameters

  DECLARE @vReturnCode Int
  EXECUTE @vReturnCode = portal_owner.CheckMandatoryParameterEmpty
                                     @pParamName  = 'pUser',
                                     @pParamValue = @pUser,
                                     @pErrorCode  = '-20101'
  IF @vReturnCode = -1 RETURN -1

  DECLARE @vPortalUserId      Int
  DECLARE @vPortalUserStatus  TinyInt
  DECLARE @vPortalUserEmail   NVarChar(50)

  SELECT @vPortalUserId = id,
         @vPortalUserStatus = status,
         @vPortalUserEmail = email
  FROM   portaldb.portal_owner.portal_user
  WHERE  login = @pUser

  DECLARE @vCount Int
  SET @vCount = @@ROWCOUNT
  EXECUTE @vReturnCode = portal_owner.RejectCountZeroOrMultiple
                                     @pCount      = @vCount,
                                     @pEntityType = 'portal_user',
                                     @pEntityKey  = @pUser,
                                     @pErrorCode  = '-20102'
  IF @vReturnCode = -1 RETURN -1

  DECLARE @vErrMsg NVarChar(255)

  IF @vPortalUserStatus = 1
  BEGIN
    SET @vErrMsg = '-20103:user "' + @pUser + '" is not active anymore'
    RAISERROR(@vErrMsg, 16, 1) WITH SETERROR, NOWAIT
    RETURN -1
  END

  EXECUTE @vReturnCode = portal_owner.CheckMandatoryParameterEmpty
                                     @pParamName  = 'pEmail',
                                     @pParamValue = @pEmail,
                                     @pErrorCode  = '-20104'
  IF @vReturnCode = -1 RETURN -1

  IF @pEmail = @vPortalUserEmail
  BEGIN
    SET @vErrMsg = '-20105:email for user "' + @pUser + '" has not been changed'
    RAISERROR(@vErrMsg, 16, 1) WITH SETERROR, NOWAIT
    RETURN -1
  END

  EXECUTE @vReturnCode = portal_owner.CheckMandatoryParameterEmpty
                                     @pParamName  = 'pSession',
                                     @pParamValue = @pSession,
                                     @pErrorCode  = '-20106'
  IF @vReturnCode = -1 RETURN -1
```

```
  DECLARE @vSessionId           Int
  DECLARE @vSessionEndedAt      DateTime
  DECLARE @vSessionPortalUserId Int

  SELECT @vSessionId = id,
         @vSessionEndedAt = ended_at,
         @vSessionPortalUserId = portal_user_id
  FROM   portaldb.portal_owner.session
  WHERE  session_key = @pSession

  SET @vCount = @@ROWCOUNT
  EXECUTE @vReturnCode = portal_owner.RejectCountZeroOrMultiple
                                    @pCount      = @vCount,
                                    @pEntityType = 'session',
                                    @pEntityKey  = @pSession,
                                    @pErrorCode  = '-20107'
  IF @vReturnCode = -1 RETURN -1

  IF @vSessionEndedAt IS NOT NULL
  BEGIN
    SET @vErrMsg = '-20108:session "' + @pSession + '" is not active anymore ' +
                   '(ended at "' +
                   CONVERT(NVarChar(20), @vSessionEndedAt, 120) + '")'
    RAISERROR(@vErrMsg, 16, 1) WITH SETERROR, NOWAIT
    RETURN -1
  END

  IF @vSessionPortalUserId IS NULL
  BEGIN
    SET @vErrMsg = '-20109:session "' + @pSession + '" is not bound to a user'
    RAISERROR(@vErrMsg, 16, 1) WITH SETERROR, NOWAIT
    RETURN -1
  END

  IF @@ERROR <> 0
  BEGIN
    RAISERROR('-20999:unexpected error (parameter validation)', 16, 1) WITH SETERROR, NOWAIT
    RETURN -1
  END

  -- process logic

  UPDATE portaldb.portal_owner.portal_user
  SET    email = @pEmail
  WHERE  id = @vPortalUserId

  IF @@ERROR <> 0
  BEGIN
    SET @vErrMsg = '-20999:unexpected error (setting email for user "' +
                   @pUser + '")'
    RAISERROR(@vErrMsg, 16, 1) WITH SETERROR, NOWAIT
    RETURN -1
  END

  DECLARE @vReturn Int
  EXECUTE @vReturn = portal_owner.LogSessionActivity
                                @pSession        = @pSession,
                                @pActivityType   = 'change email',
                                @pAdditionalInfo = @pUser,
                                @pOldValue       = @vPortalUserEmail,
                                @pNewValue       = @pEmail
  IF (@vReturn = -1) RETURN -1

  RETURN 0

END
GO
```

Zunächst wird durch Aufruf der Hilfsprozedur *CheckMandatoryParameterEmpty()* sichergestellt, dass der Eingabeparameter *pUser* weder NULL ist noch einen leeren String oder bloß Leerzeichen enthält. Anschließend werden Status und bisherige E-Mail-Adresse des Benutzers in der Datenbank selektiert. Falls der Benutzer nicht bekannt ist oder den Status “inaktiv” hat, wird eine Exception mit einer benutzerdefinierten Fehlermeldung ausgelöst (z.B. “user '4711' is not active anymore”). Weitere Exceptions werden ausgelöst, wenn die übergebene E-Mail-Adresse leer ist oder mit der bisherigen E-Mail-Adresse des Benutzers übereinstimmt.

Zwecks Verifizierung einiger wichtiger Voraussetzungen wird ein zusätzlicher Übergabeparameter namens *pSession* eingeführt. Mit Hilfe des darin enthaltenen Session-Keys kann sichergestellt werden, dass ein Session-Objekt in der Datenbank erzeugt wurde. Diese Session enthält Informationen über den angemeldeten Benutzer, insbesondere über dessen Rolle und Berechtigungen. Ist die angegebene Session in der Datenbank nicht bekannt oder wurde diese bereits beendet, oder ist der Session mangels Authentifizierung noch kein authorisierter Benutzer zugeordnet, so wird eine Exception ausgelöst.

Erst wenn alle vorgenannten Voraussetzungen erfüllt sind, wird die neue E-Mail-Adresse des Benutzers in der Datenbank gespeichert. Durch Aufruf der Hilfsprozedur *LogSessionActivity()* wird innerhalb der gleichen Datenbanktransaktion protokolliert, welche Daten von welchem Benutzer geändert wurden und wie die alte und neue E-Mail-Adresse bei diesem Änderungsvorgang war. Auf diese Weise ist bei Commit der Datenänderung sichergestellt, dass auch das zugehörige Protokoll in der Datenbank festgeschrieben wird.

Vergleich der beiden Verfahrensweisen

Die defensiv implementierte Datenbankprozedur erfordert, dass der Aufrufer gewisse Vorleistungen erbringt:

1. Er muss in der Portaldatenbank zunächst eine Session erzeugen und somit die Voraussetzung für die nachgelagerte Protokollierung schaffen.
2. Er muss sich authentifizieren, damit das System überprüfen kann, ob der Aufrufer authorisiert ist.
3. Er muss sicherstellen, dass die durchzuführende Änderung sinnvoll ist und seinerseits sinnvolle Daten übergeben.

Durch die Art der Implementierung wird es Angreifern, die es schaffen, bis zu dieser Ebene der Software-Architektur durchzudringen, erheblich erschwert, Daten unberechtigt zu lesen, zu modifizieren oder zu löschen. Es wird zudem ausgeschlossen, dass Änderungen im System durchgeführt werden können, die später nicht mehr nachvollziehbar sind.

Tabelle 1 vergleicht einige Kennzahlen für die vorgestellten Verfahrensweisen. Die aufgeführten Werte haben natürlich keine Allgemeingültigkeit, zeigen aber sehr deutlich die Unterschiede auf. Die Anzahl der möglicherweise ausgelösten Exceptions ist bei der defensiven Implementierung sehr hoch. Dementsprechend umfangreich ist die Implementierung. Im Vergleich besteht die defensive Variante aus 7 Mal so vielen Zeilen Source-Code!

Kennzahl	Einfache Implementierung	Defensive Implementierung
Anzahl der Exceptions:	**1 Exception**	**11 Exceptions**
Lines of Code (LOC):	**16 Zeilen**	**122 Zeilen**

Tabelle 1: Einige Kennzahlen für die Datenbankprozedur *SetUserEmail()* im Vergleich

Wie viel Fehlerbehandlung ist erforderlich?

Je mehr Fehlersituationen behandelt werden, desto umfangreicher wird der Source-Code, und umso höher ist der in der Entwicklung entstehende Aufwand. Daher stellt sich die Frage, wie viel Fehlerbehandlung gemeinhin erforderlich ist. Die folgende Beobachtung führt zu einer allgemeingültigen Erkenntnis:

Jede Zeile Source-Code kann potentiell mind. einen Fehler verursachen.

Aus diesem Grund müssen mind. 50% des Source-Codes aus Fehlerbehandlung bestehen.

Diese Aussage gilt für jede Implementierung innerhalb der mehrschichtigen Architektur einer Software: für jede Datenbankprozedur, für jede .NET-Klasse und für jedes ASP.NET-Skript.

Auswertungen aus der Praxis zeigen, dass stabile und zuverlässige Software-Produkte sogar aus ca. 70% Fehlerbehandlung bestehen.

Fazit

Fehler, die erst in der Betriebsphase einer Software auftreten, führen zu unverhältnismäßig hohen Wartungskosten. Das Konzept der defensiven Programmierung bewirkt, dass die Eintrittswahrscheinlichkeit solcher Produktionsfehler sinkt und die Software insgesamt stabiler wird.

Natürlich gibt es keine fehlerfreie Software. Aber es ist möglich, zuverlässige Software mit einer geringen Restfehlerrate zu erstellen. Denn je restriktiver der Zugang zu den Applikationsdaten gehandhabt wird, desto kontrollierter wird der Verarbeitungsprozess. Seiteneffekte bei Code-Änderungen fallen aufgrund der gemeldeten Fehler früher auf. Zudem wird es Angreifern unverhältnismäßig erschwert, Daten zu modifizieren oder unberechtigt einzusehen.

Erreicht werden diese Effekte durch eine grundsätzlich misstrauische Einstellung des Entwicklers gegenüber allen Aufrufern des implementierten Codes. Bevor die eigentliche Verarbeitung erfolgreich durchgeführt werden kann, erfolgt eine sehr umfangreiche und restriktive Verifizierung aller Annahmen, Voraussetzungen und übergebenen Daten. Benutzereingaben werden im Zweifelsfall 3x validiert, 1x auf jeder einzelnen Software-Ebene innerhalb der Mehrschichtenarchitektur. Das Erzeugen eines benutzerdefinierten Session-Objekts im Datenmodell der Applikation ermöglicht zudem die Protokollierung aller durchgeführten Änderungen sowie eine Überprüfung, ob der Aufrufer authentifiziert und authorisiert ist, die Änderung durchzuführen.

Ausblick

Die Qualität einer Software wird maßgeblich durch die Vorgehensweise der Entwicklungsteams bestimmt. In der letzten Ausgabe wurde die Rekonstruktionsfähigkeit einer Anwendung vorgestellt und gezeigt, wie die Wartungskosten einer Software durch bestimmte Fehlerbehandlungs- und Protokollmechanismen gesenkt werden können.

In der nächsten Ausgabe wird vorgestellt, wie der erzeugte Source-Code einer Software mit Hilfe automatisierter Code-Reviews qualitätsgesichert und stabilisiert werden kann.

Ethical Hacking

Die Netz Security

Manu Carus ist unabhängiger Technologie-Berater und Software-Ingenieur der iCommit Integrationslösungen GmbH. Seine Schwerpunkte sind Software-Architekturen, technische Spezifikationen, Integration sowie Sicherheits-, Performance- und Skalierungsaspekte. Sie erreichen ihn unter manu.carus@ethical-hacking.de.