Frank Simon · Jürgen Grossmann
Christian Alexander Graf · Jürgen Mottok · Martin A. Schneider

Basiswissen Sicherheitstests

Aus- und Weiterbildung zum
ISTQB® Advanced Level Specialist
Certified Security Tester

dpunkt.verlag

Inhalt

4 Sicherheitstesten im Softwarelebenszyklus

»Das Tragische an jeder Erfahrung ist, dass man sie erst macht, nachdem man sie gebraucht hätte.«

Nietzsche

Dieses Kapitel erläutert, warum sich Sicherheit am besten entlang eines Lebenszyklusprozesses realisieren und aufrechterhalten lässt. Zu diesem Zweck werden sicherheitsbezogene Aktivitäten für einen gegebenen Softwarelebenszyklus systematisch vorgestellt und erläutert. Hierzu zählen u.a. die Analyse von Anforderungen aus der Perspektive der IT-Sicherheit, die sicherheitsbezogene Prüfung von Architektur- und Entwurfsdokumenten, das Verständnis von implementierungsbegleitenden Sicherheitstestaktivitäten sowie der Entwurf, die Durchführung und Auswertung von Sicherheitstests im Rahmen des Komponenten-, Integrations- sowie des System- und Abnahmetests. Abschließend wird erläutert, welche Rolle der Sicherheitstest bei der Wartung und Pflege, d.h. bei der Aufrechterhaltung der Sicherheit im Betrieb der Software, spielt.

4.1 Die Rolle der Sicherheit im Softwarelebenszyklus

Sicherheit ist kein permanenter Zustand.

Die Sicherheit von Software ist kein permanenter Zustand. Sie ist flüchtig und muss im Rahmen von definierten Prozessen über den gesamten Lebenszeitraum einer Softwareanwendung immer wieder neu realisiert werden. Dies hat Konsequenzen sowohl für die Softwareentwicklung wie auch für den Betrieb der Software. So kann Sicherheit beispielsweise nur schwer nachträglich in ein Softwaresystem hinzugefügt bzw. eingepflegt werden. Die entsprechenden Änderungen in Architektur, Protokollen und Schnittstellen sind in der Regel so hoch, dass sie dann nur mit sehr hohen Kosten und sehr zeitaufwendig umgesetzt werden können. Ähnlich verhält es sich mit dem Testen der Sicherheit. Erste Sicherheitstests sollten bereits in den frühen Phasen der Softwareentwicklung

geplant, konzipiert und umgesetzt werden. Nur so kann sichergestellt werden, dass Schwachstellen sowohl in der Software als auch im Prozess frühzeitig erkannt und beseitigt werden können.

Grundsätzlich gilt: Sicherheit ist, bis auf sehr wenige Ausnahmen, das Ergebnis einer systematischen, sicherheitsorientierten Entwicklung, einer sorgfältigen Qualitätssicherung sowie einer sicherheitsorientierten Wartung der Software über die gesamte Betriebsphase. Wie beim Testen von Software im Allgemeinen ist auch das Testen der Sicherheit ein Prozess, der systematisch mit den Lebenszyklusaktivitäten einer Software abgestimmt und in diese integriert werden muss.

4.1.1 Der Softwarelebenszyklus und Lebenszyklusmodelle

Softwarelebenszyklusmodelle strukturieren die Erstellung und den Betrieb von Software.

Ein Softwarelebenszyklusprozess bietet einen definierten Rahmen für die Durchführung aller notwendigen Aktivitäten im Lebenszyklus einer Software, d.h. von der initialen Planung der Software über ihre Entwicklung bis hin zur Beendigung der Softwarenutzung. Ziel ist es, alle Aktivitäten zeitlich und inhaltlich aufeinander abzustimmen. So müssen beispielsweise im Entwicklungsprozess die Anforderungen der Benutzer ermittelt werden, bevor die eigentliche Entwicklung einer Softwareanwendung beginnen kann.

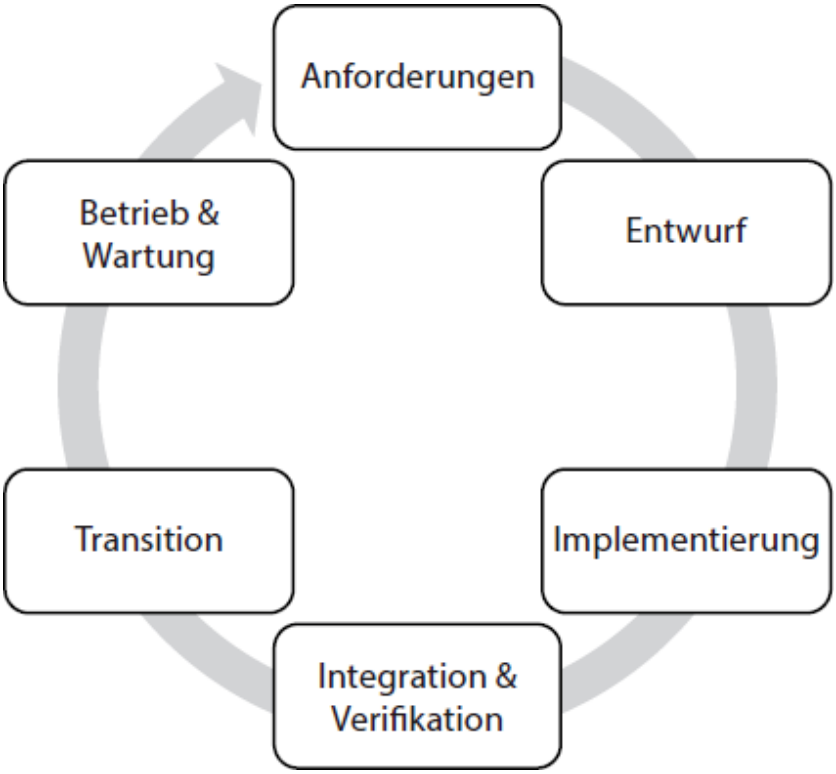

Abb. 4–1 *Vereinfachtes Lebenszyklusmodell angelehnt an ISO/IEC/IEEE 12207 [ISO 12207]*

In der Praxis existiert eine Vielzahl unterschiedlicher Lebenszyklusmodelle, die wiederum durch konkrete Vorgehensmodelle für die Softwareentwicklung ergänzt werden. Während ein Lebenszyklusmodell den kompletten Lebenszyklus der Software bzw. des Systems abdeckt und in der Regel Aktivitäten und ihre Abhängigkeiten auf einem hohen Abstraktionsniveau beschreibt, sind Vorgehensmodelle deutlich konkreter. Sie beschreiben das konkrete Vorgehen in einem konkreten Lebenszyklusabschnitt bzw. auch für den gesamten Lebenszyklus einer Software. Die einzelnen Aktivitäten, die Reihenfolge ihrer Durchführung und ihre Abhängigkeiten werden üblicherweise sehr konkret beschrieben, sodass sich auf der Ebene der Vorgehensmodelle beispielsweise zwischen iterativen und Wasserfallmodellen unterscheiden lässt (vgl. Kap. 3).

Standards zum Software- bzw. Systemlebenszyklus wie ISO/IEC/ IEEE 12207 [ISO 12207] oder ISO/IEC/IEEE 15288 [ISO 15288] definieren die grundlegenden Aktivitäten sowie ihre Abhängigkeiten im Software- bzw. Systemlebenszyklus. Die Entscheidung für einen konkreten Softwarelebenszyklus hängt von der Art der Organisation, vom konkreten Projekt und von weiteren Faktoren ab. Im Rahmen dieses Buches verwenden wir ein vereinfachtes Lebenszyklusmodell, das aus den oben genannten Normen abgeleitet ist (vgl. Abb. 4–1).

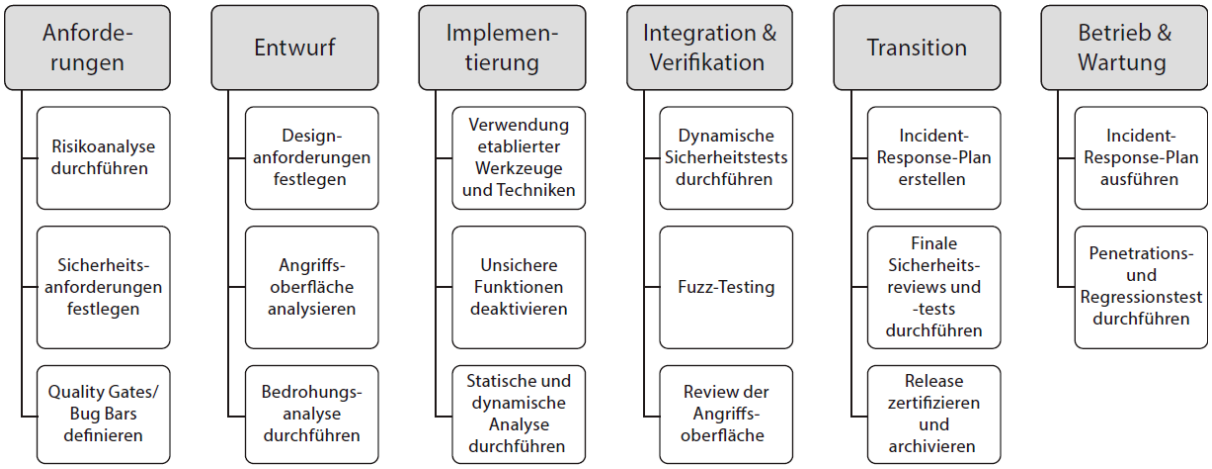

Abb. 4–2 *Zuordnung von sicherheitsbezogenen Aktivitäten zu einzelnen Phasen eines Softwarelebenszyklus (nach Microsoft SDL [Microsoft 10])*

Vorgehensmodelle für sichere, industrielle Softwareentwicklung

In der Industrie hat sich eine Reihe von Standards, Vorgehensmodellen und Best Practices etabliert, die den Aspekt der sicheren Softwareentwicklung sowie die Absicherung der Software im Betrieb in den Mittelpunkt stellen. Hierzu zählt neben Standards wie der ISO 27034 [ISO 27034] u.a. auch der Microsoft Security Development Lifecycle (Microsoft SDL) [Microsoft 10]. Microsofts Prinzip ist die prozessuale Realisierung von »Sicherheit durch Design« durch die systematische Integration von Softwaresicherheit als explizite Anforderung in den Entwicklungsprozess. Der Microsoft SDL beschreibt Absicherungsmaßnahmen und Best Practices, die den traditionellen Softwareentwicklungsprozess ergänzen und dadurch sicherstellen, dass die Realisierung von Sicherheit in dem Maße berücksichtigt und integriert wird, wie es für die jeweilige Anwendung notwendig ist. Abbildung 4–2 zeigt den Microsoft SDL mit leichten Anpassungen bezüglich der in diesem Buch verwendeten Termini für die einzelnen Phasen des Softwarelebenszyklus. Zu den Absicherungsmaßnahmen und Best Practices zählen neben Schulungsmaßnahmen, einer Bedrohungsanalyse und der Umsetzung eines sicherheitsorientierten Anforderungsmanagements und Designs auch Aktivitäten des Sicherheitstestens wie statische Analyse, dynamische Sicherheitstests, Fuzz-Testing und Sicherheitsreviews. Für die agile Softwareentwicklung existiert eine Zuordnung der grundlegenden Aktivitäten des Microsoft-SDL auf verschiedene Phasen eines agilen Entwicklungsprozesses bzw. auf den DevOps-Zyklus (vgl. [Microsoft 19]).

Exkurs: Sichere Softwareentwicklung in agile Prozessen

Zentrale Paradigmen der agilen Softwareentwicklung sind Dynamik, Kommunikation und selbstverantwortliches Handeln der agil arbeitenden Teams. So sind Beteiligte, ihr Know-how und ihre

Zusammenarbeit wichtiger als Prozesse und Werkzeuge. Die Bereitstellung lauffähiger Software hat Vorrang vor umfassender Dokumentation und Modellierung, und Kundenwünsche und daraus resultierende Anforderungsänderungen sind wichtiger als die sture Verfolgung eines Entwicklungsplans [Beck et al. 01]. Bezogen auf die Entwicklung sicherer Software besteht eine zentrale Herausforderung darin, die notwendige Sicherheitsexpertise und Sicherheitstestexpertise in den agilen Teams und Akteuren zu verankern. Es ist sicherzustellen, dass sowohl in den Entwicklungsteams wie auch in den Teams bzw. Gremien, die Architekturentscheidungen treffen, die notwendige Sicherheitsexpertise vorhanden ist, um die Umsetzung von Sicherheitsanforderungen realisieren und prüfen zu können. Sicherheit ist grundsätzlich eine nichtfunktionale Eigenschaft, sodass sich Sicherheitsanforderungen nur teilweise funktional ausdrücken lassen und sich so auf unterschiedlichen Ebenen in agilen Dokumentationsschemata wiederfinden.

Sichere Softwareentwicklung muss systematisch in agilen Prozessen verankert werden.

Sicherheitsanforderungen können funktionalen Charakter haben, wenn es um die Umsetzung konkreter Sicherheitsfunktionalität geht. Hierzu zählt beispielsweise die Realisierung konkreter Prüfalgorithmen für Passwortrichtlinien. Ein solcher Algorithmus, wie z.B. die automatisierte Prüfung der Passwortlänge und Zusammensetzung, kann als agile »Security User Story« definiert und im Rahmen eines Sprints umgesetzt und geprüft werden. Andere Sicherheitsanforderungen werden auf einer allgemeinen Ebene definiert (z.B. alle Kommunikation sollte TLS verwenden) und als eine globale Zusicherung realisiert, die als Abnahmekriterium in eine größere Zahl von User Stories integriert werden muss. Im Unterschied zu einer funktionalen Security User Story ist eine solche Zusicherung nicht abschließend abarbeitbar, sondern muss beim Aufsetzen neuer User Stories immer wieder auf ihre Relevanz und die damit einhergehende Notwendigkeit ihrer Integration als Abnahmekriterium geprüft werden. Die sorgfältige Definition geeigneter Abnahmekriterien sowie ihre systematische Überprüfung ist eine der entscheidenden Herausforderungen im agilen Vorgehen.

Grundlegend für den Erfolg eines agilen Projekts ist die Definition klarer Verantwortlichkeiten und Rollen für die Durchführung sicherheitsrelevanter Entwicklungsaktivitäten. Abbildung 4–3 zeigt diese Zuordnung der Aktivitäten zu den Phasen eines iterativen Vorgehens durch entsprechende Markierungen der Aktivitäten.

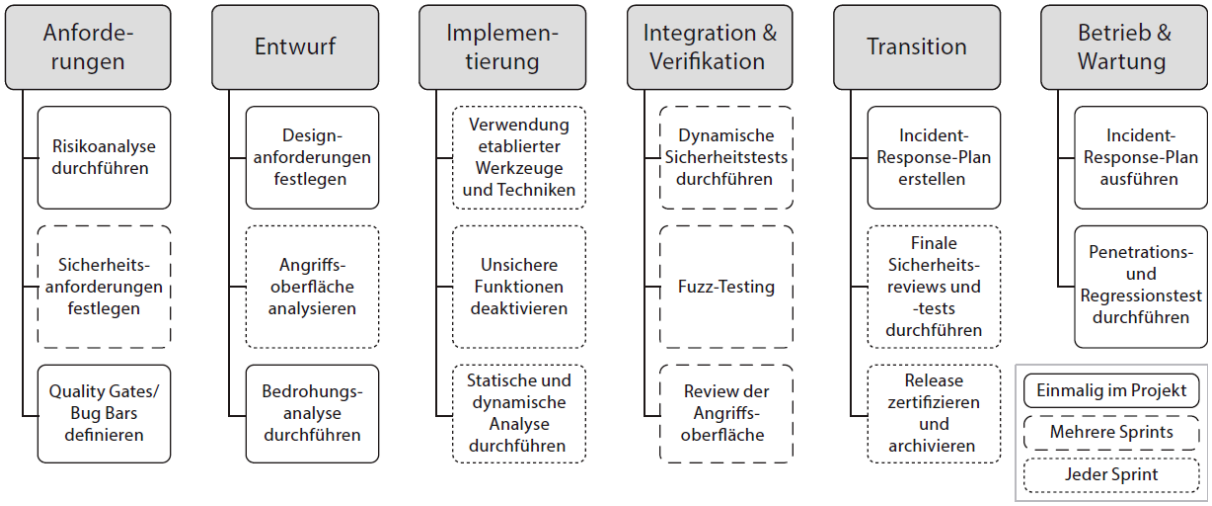

Abb. 4–3 *Zuordnung von sicherheitsbezogenen Aktivitäten zu einzelnen Phasen eines agilen Vorgehensmodells (nach Microsoft SDL [Microsoft 19])*

Microsoft hat die einzelnen Aktivitäten des Microsoft SDL auf die relevanten Phasen eines iterativen Vorgehens abgebildet und unterscheidet zwischen Aktivitäten, die einmalig im Projektverlauf ausgeführt werden sollten, Aktivitäten, die über mehrere Sprints hinweg iteriert werden sollten, und Aktivitäten, die für jeden Sprint durchgeführt werden sollten. Einmalige Aktivitäten bzw. Aktivitäten, die in großen Iterationen durchgeführt werden, sind solche Aktivitäten, die eher das gesamte Produkt bzw. Projekt berücksichtigen (vgl. Abb. 4–3, Kästen mit durchgehender Linie). Hierzu gehören Risiko- und Bedrohungsanalysen, die Definition der Quality Gates und Designanforderungen sowie die Spezifikation des Incident-Response-Prozesses. Das Ableiten und Dokumentieren von Sicherheitsanforderungen sowie die Durchführung aufwendiger Testaufgaben wie z.B. dynamischer Sicherheitstests werden regelmäßig in kürzeren Iterationen für mehrere Sprints gemeinsam durchgeführt (vgl. Abb. 4–3, Kästen mit gestrichelter Linie). Aktivitäten hingegen, die einen direkten Bezug zur Implementierung haben, wie beispielsweise die Durchführung statischer Tests und Codeanalysen, bzw. Prüfaktivitäten, die ein konkretes Release adressieren, wie z.B. geeignete Abnahmereviews und Abnahmetests, sollten Teil eines jeden Sprints sein (vgl. Abb. 4–3, Kästen mit gepunkteter Linie).

4.1.2 Sicherheit in den Phasen des Softwarelebenszyklus

Wie bereits beschrieben, ist die systematische Verankerung von Aktivitäten zur Realisierung und Pflege der Sicherheit im Softwarelebenszyklus die grundlegende Voraussetzung für die Entwicklung und den Betrieb sicherer Software. Im Folgenden werden Aktivitäten mit Bezug zur Sicherheit entlang der einzelnen Phasen des Softwarelebenszyklus aus Abbildung 4–1 erläutert.

4.1.3 Die Ermittlung von Sicherheitsanforderungen

Sicherheitsanforderungen als fester Bestandteil der Anforderungsermittlung

Für viele Unternehmen ist es bereits eine Herausforderung, elementare Benutzeranforderungen zu formulieren, die aussagekräftig, unmissverständlich, widerspruchsfrei, vollständig, richtig und testbar sind. Zur Ermittlung der Sicherheitsanforderungen sollten darüber hinaus alle sicherheitsrelevanten Aspekte berücksichtigt werden, die sich aus den Sicherheitszielen, Risiken und Regulierungen der Kunden bzw. Nutzer ableiten, die die Software einsetzen sollen. Insbesondere sollte beachtet werden, dass Anforderungen nicht allein durch die direkten Stakeholder (Kunden, Nutzer) definiert werden, sondern auch regulatorische, technische und geschäftliche Belange widerspiegeln müssen, die sich u.a. aus Standards, der Gesetzgebung bzw. den Geschäftszielen ableiten lassen (vgl. auch Kap. 9).

Hinweis: Rückgriff auf etablierte Methoden der Anforderungsermittlung

Zur Identifikation von Anforderungen existiert eine Vielzahl unterschiedlicher Methoden, die helfen, den Kreis der Stakeholder systematisch zu vervollständigen sowie den Prozess der Anforderungserhebung insgesamt zu systematisieren. Zu den einfachen Methoden zählen Recherche sowie einfache Interviews bzw. Workshops mit den Stakeholdern. Ein systematisches Vorgehen integriert Prozesse wie Risikoanalysen sowie fundierte Experteninterviews, sodass der Erfassungsvorgang systematisiert und der Kreis der Stakeholder systematisch vervollständigt werden kann. Einen guten Überblick über Strukturvorgaben und Qualitätsmerkmale in der Anforderungsspezifikation gibt der Internationale Standard ISO/IEC 29148 [ISO 29148]. Der Standard beschreibt die Spezifikation von Software in Form von Anforderungsspezifikationen und enthält Beispiele und Arbeitshilfen für eine praktische Umsetzung.

Risikobasierte Ansätze zur Gewichtung der Sicherheitsanforderungen.

Bereits während der Anforderungsermittlung sollten die grundlegenden Sicherheitsziele, die mit den Zielen assoziierten Risiken sowie abstrakte Gegenmaßnahmen verstanden sein, sodass sich insbesondere Letztere als Anforderungen an die Software formulieren lassen. Risikobasierte Softwareentwicklungsprozesse machen diese Abhängigkeit zwischen Sicherheitszielen, Risiken und Sicherheitsanforderungen explizit, indem sie einen direkten Bezug zwischen den Aktivitäten der Bedrohungs- und Risikoanalyse und der Anforderungsermittlung vorsehen.

Hinweis: Risikobasierte Ermittlung von Sicherheitsanforderungen

Um Sicherheitsanforderungen systematisch herleiten und angemessen gewichten zu können, ist es sinnvoll, zuvor Assets, Schutzziele sowie eine Liste möglicher Bedrohungen bzw. Risiken zu identifizieren.

Sicherheitsanforderungen, wie alle anderen Anforderungen auch, müssen auf detaillierte, eindeutige und rückverfolgbare Weise dokumentiert werden. Dadurch ist es möglich, dass Implementierungen und Tests auf die Anforderungen zurückzuführen sind und sich Implementierung und Anforderungen später verifizieren und validieren lassen.

Hinweis: Iteratives Vorgehen

Ein iteratives Vorgehen stellt sicher, dass Sicherheitsanforderungen auch in späteren Entwicklungsphasen an geänderte Benutzeranforderungen angepasst und entsprechend ergänzt, verfeinert und hinzugefügt werden können.

Sicherheit ist in der Praxis ein Kompromiss.

Schlussendlich ist zu beachten, dass 100%ige Sicherheit, ähnlich wie 100%ige Fehlerfreiheit, nicht zu erreichen ist. Die Realisierung von Sicherheit geht immer mit Kompromissen einher. Die Realisierung und Prüfung von Sicherheitsmechanismen sind arbeitsintensiv und teuer und Ressourcen sind in der Praxis immer beschränkt. Sicherheitsmechanismen benötigen Rechen- und Übertragungskapazität und können so u.a. die Performance einer Software negativ beeinflussen. Darüber hinaus sind sie häufig nicht vollständig transparent, d.h., Benutzer müssen sich mit den Sicherheitsmechanismen auseinandersetzen und manchmal von ihren intuitiven Interaktionsmustern abweichen. Das kann im schlimmsten Fall dazu führen, dass eine Software nicht genutzt bzw. Sicherheitsmechanismen deaktiviert werden.

Hinweis: Kompromisse zwischen Sicherheit und anderen Benutzeranforderungen

Kompromisse zwischen Sicherheit und anderen Benutzeranforderungen wie Performanz und Benutzbarkeit müssen sorgsam gegeneinander abgewogen werden und das Ergebnis systematischer Entscheidungsfindung sein. So sind Sicherheitsmechanismen, die nicht genutzt und vom Benutzer systematisch umgangen werden, nicht nur wirkungslos, sondern auch gefährlich, da ihre Wirkungslosigkeit bzw. ihr Fehlen in der Regel schwer erkenn- und prüfbar ist.

Beispiel: Kompromisse zwischen Sicherheit und Benutzbarkeit am Beispiel von Richtlinien zur Änderung des Passworts

Beispiel: Kompromisse zwischen Sicherheit und Benutzbarkeit am Beispiel von Richtlinien zur Änderung des Passworts

Das amerikanische National Institute of Standards and Technology (NIST) hat seine Empfehlungen für sichere Passwörter dahingehend angepasst, dass eine in vielen Systemen und Organisationen fest etablierte regelmäßige Änderung des Passwortes nicht mehr empfohlen wird. Die Begründung dafür ist, dass der Sicherheitsgewinn durch das regelmäßige Wechseln der Passwörter letztendlich zu gering war, um das Risiko einzugehen, dass Benutzer aus Überforderung möglichst einfache und kurze Passwortschemata nutzen oder sich die Passwörter in unsicherer Art und Weise notieren. Hierzu passt die ungeschriebene Weisheit von Sicherheitsauditoren, dass der Ort, wo man die meisten Passwörter bzw. Passworthinweise finden kann, die Unterseite der Tastatur am Arbeitsplatz eines Benutzers ist. Die NIST empfiehlt

inzwischen möglichst lange Passwörter, die sich gut merken lassen. Das heißt u.a. weniger Passwortänderungen und weniger Sonderzeichen [NIST SP 800-63a 17].

4.1.4 Der Entwurf sicherer Software

Ziel der Entwurfsphase ist es, auf der Basis der in der Anforderungsermittlung identifizierten Anforderungen für ein System bzw. eine Anwendung einen funktionierenden und sicheren Lösungsansatz zu entwerfen. Die Entwurfsphase beginnt mit der Analyse der dokumentierten Anforderungen, setzt sich fort mit der Auswahl der praktikabelsten Herangehensweise, um ein System bzw. eine Anwendung auf sichere Art zu entwickeln, und endet mit dem Dokumentieren des Entwurfs und der Architektur mittels geeigneter Techniken sowie im Einklang mit dem Softwarelebenszyklus.

Hinweis: Dokumentation in agilen Projekten

Aufgrund des Primats des laufenden Codes wird in agilen Softwareprojekten die Erstellung formaler Dokumentationen und Spezifikationen nicht im selben Umfang wie in der klassischen Softwareentwicklung betrieben. Die Herausforderung in agilen Prozessen besteht darin, die Dokumentationspflichten dort einzufordern, wo sie notwendig sind, und trotzdem eine rigide Qualitätssicherung auf Basis der weniger umfangreichen formalen Artefakte realisieren zu können.

Ziel beim Entwurf sicherer Systeme ist die Erstellung eines System- oder Anwendungsentwurfs, der den angegebenen Sicherheitsanforderungen genügt und die Umsetzung der Sicherheitsanforderungen unterstützt.

Hierzu werden ausgehend von den Benutzeranforderungen bzw. den Anforderungen aus der Regulierung konkrete Sicherheitsanforderungen an den Entwurf bzw. die Architektur abgeleitet und diese umgesetzt. Begleitend kann eine entwurfsspezifische Analyse der Angriffsoberfläche sowie der damit einhergehenden Bedrohungen und Risiken erfolgen.

Rückgriff auf existierende Prinzipien für einen sicheren Softwareentwurf

Für den Entwurf sicherer Software findet sich in der Praxis eine Reihe von Empfehlungen und Prinzipien, die als Grundlage für einen sicheren Softwareentwurf verwendet werden können. Als Beispiel seien hier die Empfehlungen zur Vermeidung der 10 typischsten Designfehler [IEEE 14] des IEEE Center for Secure Design genannt. Tabelle 4–1 benennt die 10 typischen Fehler, die im Dokument des IEEE Center dann genauer erläutert werden.

Vertrauen kann verdient oder entgegengebracht, aber nie vorausgesetzt werden.

Es ist ein Authentisierungsmechanismus zu nutzen, der sich nicht umgehen oder manipulieren lässt.
Erst authentisieren, dann autorisieren.
Daten und Steuerbefehle sind streng zu trennen. Steuerbefehle aus nicht vertrauenswürdigen Quellen dürfen nie ausgeführt werden.
Es ist ein Vorgehen zu definieren, sodass gewährleistet ist, dass alle Daten explizit validiert werden.
Verschlüsselung muss richtig angewendet werden.
Es sind sensible Daten zu identifizieren und deren (sichere) Handhabung zu definieren.
Es ist stets der Benutzer zu berücksichtigen (…). Die Sicherheit eines Softwaresystems ist untrennbar damit verbunden, wie es durch den Benutzer verwendet wird.
Es muss verstanden werden, auf welche Weise die Integration externer Komponenten die Angriffsfläche eines Systems verändert.
Sei flexibel, wenn es um zukünftigen Änderungen an Objekten und Akteuren geht (…). Die Sicherheit von Software muss auf Veränderungen ausgerichtet sein (…).

Tab. 4–1 *Empfehlungen des IEEE Center for Secure Design [IEEE 14]*

Die Empfehlungen entstammen der Praxis renommierter Sicherheitsexperten und beschreiben Sicherheitsprobleme, die schon seit Jahren bekannt sind und sich dennoch immer wieder auch in aktueller Software finden lassen. Gerade deshalb ist es wichtig, die Einhaltung solcher Empfehlungen, so selbstverständlich sie auch klingen mögen, im Softwareentwurf systematisch zu berücksichtigen und ihre Umsetzung nachhaltig zu prüfen.

4.1.5 Die Implementierung sicherer Software

Beispiel: Vermeiden von SQL-Injection- und Pufferüberlaufschwachsteller

In der Implementierungsphase werden die Anforderungen und der Softwareentwurf in ausführbaren Programmcode überführt. Sichere

Software stellt in diesem Zusammenhang besondere Anforderungen an die Implementierungsphase. Einerseits muss gewährleistet werden, dass die in den Sicherheitsanforderungen vorgesehene Sicherheitsfunktionalität implementiert und integriert wird. Hierzu zählt u.a. Software für die Realisierung bzw. Integration von Authentisierung und Autorisierung, Zugriffsschutz, Verschlüsselung etc. Darüber hinaus sollte sichergestellt werden, dass insgesamt sicherer Programmcode erstellt wird, d.h. Programmcode, der unabhängig davon, ob er explizite Sicherheitsfunktionalität realisiert, keine Schwachstellen und sicherheitsrelevanten Fehler enthält.

Hinweis:

Viele klassische Softwarefehler können auch Auswirkungen auf die Sicherheit der Software haben. So produzieren Softwarefehler per se unvorhergesehenes Verhalten und bieten einem Angreifer die Möglichkeit, dieses für seine Zwecke auszunutzen. Darüber hinaus zeigen viele bekannte Sicherheitslücken die gleichen Fehlerwirkungen, die auch bei klassischen Softwarefehlern auffallen. So lassen sich Pufferüberläufe oder andere Speicherfehler häufig durch Systemabstürze und anderes undefiniertes Verhalten erkennen. Sichere Software zeichnet sich immer auch durch geringe Fehlerraten aus.

Beispiel: Vermeiden von SQL-Injection- und Pufferüberlaufschwachstellen

Sichere Software durch Vermeiden von Implementierungsfehlern

Während der Implementierung haben Entwickler die beste Möglichkeit, sichere Codierungsverfahren anzuwenden, um Schwachstellen zu vermeiden, die direkt auf unsichere Programmkonstrukte zurückzuführen sind. Zu diesen Schwachstellen gehören SQL-Injection- und Pufferüberlaufschwachstellen (vgl. Kap. 2). Eine Überprüfung des Codes während der Implementierung zielt darauf ab, das Vorhandensein von unsicheren Programmierartefakten zu erkennen, um diese möglichst umgehend beseitigen zu können. Schwachstellen solcher Art in späteren Phasen eines Softwareentwicklungsprojekts zu finden, wäre schwierig und kostspielig, da in späteren Phasen häufig kein direkter Zugriff auf den Programmcode mehr zur Verfügung steht und für den systematischen Test oftmals statische Verfahren zuverlässiger und umfassender als dynamische Verfahren sind.

Beispiel: Erkennen von unsicheren C/C++-Codefragmenten, die potenziell Pufferüberläufe erlauben können

Beispiel: Erkennen von unsicheren C/C++-Codefragmenten, die potenziell Pufferüberläufe erlauben können

Unsicherer Code:

```
char buffer[BUF_SIZE];

gets(buffer);
```

Sichere Alternative:

```
char buffer[BUF_SIZE];

cin >> (buffer);
```

Beispiel: Erkennen von unsicheren Java-Codefragmenten, die potenziell SQL-Injection erlauben können

Unsicherer Code:

Beispiel: Erkennen von unsicheren Java-Codefragmenten, die potenziell SQL-Injection erlauben können

```
createQuery(

  “select * from USER where id =
  ’”+Id+”’”

);
```

Sichere Alternative durch Verwendung des org.owasp.esapi.Encoder:

```
Codec c = new MySQLCodec(MySQLCodec.Mode.ANSI);

Encoder encoder = ESAPI.encoder();

…

createQuery(

      “select * from USER where id = ’”+

      encoder.encodeForSQL(c, Id)+ ”’”

);
```

In der Praxis haben sich die folgenden »Best Practices« für die Implementierung sicherer Software etabliert.

- Erstellung von Programmcode mit etablierten Werkzeugen und Verfahren

- Verwendung von Programmierrichtlinien für sichere Softwareerstellung (siehe beispielsweise [OWASP 10; SANS 2; CMU 1])
- Durchführung von Komponentenreviews zur Inspektion der Richtigkeit, Wirksamkeit und Sicherheit der Implementierung
- Durchführung von Komponententests zur Überprüfung von Richtigkeit, Wirksamkeit und Sicherheit der Implementierung
- Sichere Verwahrung und Speicherung des Programmcodes, sodass die Integrität des Codes gewahrt bleibt.

4.1.6 Die Integration und Verifikation sicherer Software

Die dynamische Prüfung der Sicherheitsanforderungen in de Integration

In der Integrations- und Validierungsphase wird ein Softwaresystem aus seinen einzeln realisierten Komponenten zusammengesetzt und im Zuge des Zusammensetzens systematisch gegen seine Anforderungen geprüft. Komplexere Funktionalität, wie beispielsweise Autorisierungsfunktionalität oder Zugriffsschutz, steht häufig während eines solchen Integrationsprozesses zum ersten Mal in Form von Software zur Verfügung. Da Sicherheitsfunktionalität oftmals nicht selber erstellt wird, sondern in Form von externen Bibliotheken und Komponenten in ein Softwaresystem integriert wird, ist gesondert darauf zu achten, dass eine solche Integration umfassend und in sicherer Form stattfindet. Insgesamt werden in der Integrationsphase umfangreiche dynamische Sicherheits- und Robustheitstests (wie z.B. Fuzz-Testing) durchgeführt, sodass sichergestellt werden kann, dass die Anforderungen des Softwaresystems auch durch das integrierte System umgesetzt werden.

4.1.7 Die Transition sicherer Software

Die Prüfung der Sicherheit im Hinblick auf die Einsatzumgebung gewährleisten

In der Transitionsphase wird das Softwaresystem auf den Betrieb in einer konkreten Zielumgebung zugeschnitten sowie in diese transferiert und für den Betrieb konfiguriert. Bei diesem Vorgang ist sicherzustellen, dass der Transfer und Installationsprozess dahingehend abgesichert ist, dass die Software nicht vor der Installation manipuliert werden kann und durch die Konfiguration keine neuen Schwachstellen im Zielsystem erzeugt werden können. Als Absicherungsmaßnahmen werden kryptografische Verfahren zum Schutz der Software sowie Sicherheitstests zur Prüfung der konfigurierbaren Sicherheitseigenschaften und Verfahren empfohlen.

Hinweis: Absicherung des Softwaretransfers durch kryptografische Hashverfahren

Bei Software, die zum Download über das Internet angeboten wird, hat sich beispielsweise etabliert, diese durch kryptografische Hashverfahren gegen Manipulation abzusichern. Dieses wird immer wichtiger, wenn die Prozesse automatisiert werden und sich zunehmend ohne menschliche Kontrollen vollziehen, und entspricht u.a. den Vorgaben des BSI IT-Grundschutz für die Sicherstellung der Integrität und Authentizität von Softwarepaketen [BSI M 4.177].

Zur Transitionsphase gehört zudem die Erstellung einer Releasehistorie, sodass bekannt ist, welche Software in welchem Zielsystem installiert ist sowie das Aufsetzen eines Incident-Response-Plans zum Umgang mit Sicherheitsvorfällen während des Betriebs (vgl. Abb. 4–2).

4.1.8 Die Aufrechterhaltung der Sicherheit während des Betriebs

Sicherheit muss gewartet werden.

Auch nach Auslieferung und Inbetriebnahme eines Softwaresystems muss dieses gewartet werden, um die gewünschten Systemeigenschaften auch im laufenden Betrieb abzusichern. Gerade im Hinblick auf die Sicherheit eines Softwaresystems ist eine systematische und sorgfältige Wartung die Grundlage dafür, einmal erzielte Sicherheitseigenschaften über einen längeren Betriebszeitraum aufrechtzuerhalten.

Zur Wartung gehört neben der Aktualisierung von Teilsystemen und Komponenten, wie beispielsweise dem Betriebssystem oder einzelnen Bibliotheken, auch die Änderung des Programmcodes des Softwaresystems sowie die Migration des Softwaresystems auf neue Hard- und Softwareplattformen und in neue Kommunikationsinfrastrukturen. Wartungsaktivitäten können dabei verschiedene Ziele verfolgen. Eine korrektive Wartung zielt darauf ab, Fehler in bereits freigegebener Software zu beseitigen und die Software dadurch funktional und sicher zu halten. Bekannt sein sollten die umfangreichen Sicherheitsupdates, die Firmen wie Microsoft, Apple oder Adobe turnusmäßig anbieten, um ihre Produkte sicherer und attraktiver zu machen. Mittels adaptiver Wartung wird ein Softwaresystem an sich ändernde Rahmenbedingungen und Anforderungen angepasst. Hierzu würde beispielsweise die Anpassung einer Software an neue Authentisierungsverfahren zählen, damit die Software in einem neuen Firmenoder Organisationskontext betrieben werden kann. Die verbessernde Wartung hingegen adressiert die Optimierung und Erweiterung bestehender Systemeigenschaften, beispielsweise die Nachrüstung einer Software mit neuen kryptografischen Verfahren, die effizienter arbeiten und eine höheres Maß an Sicherheit bieten.

Sicherheitsbezogene Wartung besteht grundsätzlich in der Bereitstellung effektiver Prozesse und Infrastrukturen für die Softwarepflege (z.B. Change-Management-Prozesse, Update-Management-Prozesse) sowie für den Umgang mit

Sicherheitsvorfällen (Incident-Response-Prozesse). Darüber hinaus sollten regelmäßig Penetrationstests sowie ein Review der Sicherheitskonfiguration durchgeführt werden.

Beispiel: Veraltete Software im Bundesweiten Amtliche Anwaltsverzeichnis

Beispiel: Veraltete Software im Bundesweiten Amtliche Anwaltsverzeichnis

Im März 2018 musste das Bundesweite Amtliche Anwaltsverzeichnis (BRAV) vom Netz gehen, nachdem bekannt geworden war, dass zu jenem Zeitpunkt das BRAV eine veraltete Version der Open-Source-Bibliothek PrimeFaces in Verwendung hatte. Das BRAV ist ein zentrales Element zur Kommunikation der Rechtspflege in Deutschland und verwaltet die Stammdaten von Anwälten, d.h. ihre Namen, Kontaktadressen, Kommunikationsdaten sowie die Safe-ID für das besondere Anwaltspostfach (beA) [heise online 18]. Die Schwachstelle in PrimeFaces existiert in alten PrimeFaces-Versionen und basiert auf Sicherheitslücken, die es nicht authentifizierten Benutzern erlauben, böswilligen Code in den PrimeFaces-Parser einzuschleusen. Seit 2016 existieren Versionen von PrimeFaces, die diese Schwachstelle nicht mehr enthalten. Eine systematische Softwarepflege mit der regelmäßigen Aktualisierung zentraler Bibliotheken hätte also ausgereicht, den Betrieb des BRAV über Jahre hinweg sicherer zu gestalten und eine temporäre Abschaltung des Dienstes zu vermeiden.

4.1.9 Sicherheitstesten im Softwarelebenszyklus

Testen zur Prüfung von Qualitätseigenschaften und zur Identifikation von Abweichungen und Mängeln

Die Aufgabe der Qualitätssicherung bzw. des Testens ist es, sowohl die konkrete Umsetzung der Prozesse wie auch die Realisierung der Artefakte im Rahmen eines Lebenszyklusprozesses auf ihre Qualitätseigenschaften zu prüfen und Mängel sowie Abweichungen zu kommunizieren. Konkrete Qualitätssicherungsaktivitäten, wie das Sicherheitstesten oder sicherheitsbezogene Reviews, sollten grundsätzlich so in den Lebenszyklusprozess einer Softwareanwendung integriert werden, dass sie ihre Wirkung so effektiv wie möglich entfalten können. Zwischenartefakte an den Prozessschnittstellen sollten bereits qualitätsgesichert sein, sodass alle nachfolgenden Aktivitäten, die auf diesen Artefakten aufsetzen, reibungsloser vonstattengehen können. So ist es beispielsweise sinnvoll, noch vor der Definition der Sicherheitsanforderungen die grundlegenden Assets und Schutzziele für eine Anwendung zu definieren und zu validieren (vgl. Kap. 1). Anschließend können die Sicherheitsanforderungen auf Basis abgestimmter und qualitätsgesicherter Assets und Schutzziele systematisch abgeleitet werden.

Exkurs: ETSI Security Testing Activities im Softwarelebenszyklus

Die ETSI [ETSI TR 101 583 15] unterteilt das dynamische Sicherheitstesten in drei Hauptaktivitäten. Beim *Test der Sicherheitsfunktionen und -eigenschaften* wird das System aus der

Perspektive der Systemfunktionalität bzw. der Funktionsanforderungen betrachtet. Im Unterschied zum rein funktionalen Testen wird aber neben der legitimen Nutzung des Systems auch die Möglichkeit von vorsätzlichen Angriffen, d.h. der mutwilligen Überschreitung der intendierten Systeminteraktion, berücksichtigt. *Robustheitstests* sind eine Form der Prüfung, bei der die Systemeingaben zufällig mutiert oder systematisch modifiziert werden, um sicherheitsrelevante Fehler wie Abstürze, Verzögerungsschleifen oder Speicherlecks zu finden. *Security-bezogene Last- und Performanztests* gehen darüber hinaus. Sie sind motiviert durch die hohe Zahl von verteilten Denial-of-Service-Angriffen (DDoS) im Internet *und* zielen darauf ab, das System durch das Aufspielen hochfrequenter, sequenzieller oder paralleler Lastszenarien bis an seine Leistungsgrenze zu bringen.

Abbildung 4–4 zeigt die Verortung der genannten Testaktivitäten im Lebenszyklus eines Softwaresystems und in Relation zu den Aktivitäten der Sicherheitsrisikobeurteilung, des Penetrationstests und des Regressionstests. Es ist zu beachten, dass sich das von der ETSI verwendete Lebenszyklusmodell sowie die Zuordnung der Testaktivitäten von den in diesem Buch verwendeten Definitionen leicht unterscheidet.

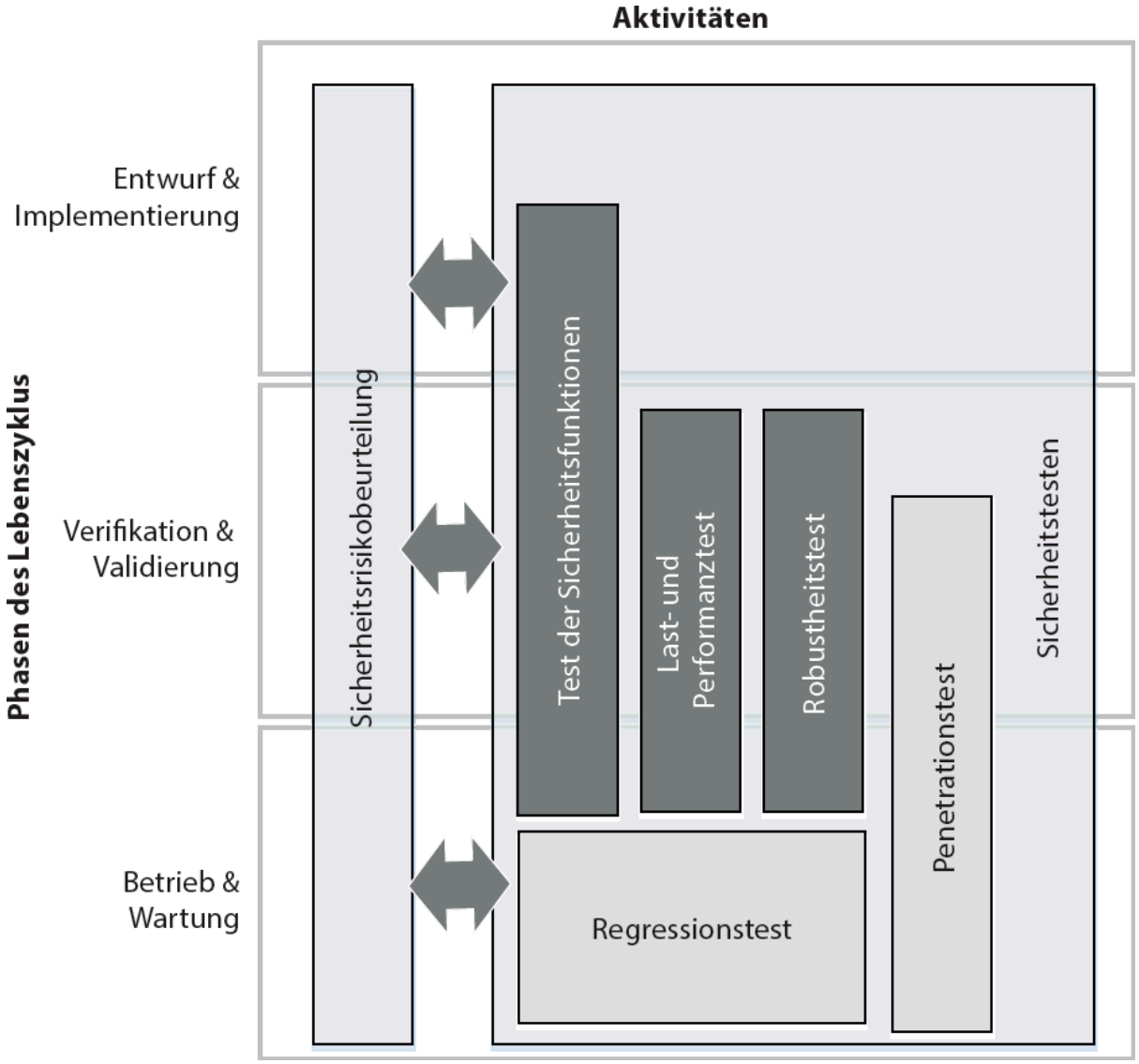

Abb. 4–4 *ETSI-Sicherheitstestdomänen im SDL (nach [ETSI EG 203 251 16])*

Die Sicherheitstestaktivitäten systematisch in den Lebenszyklus einer Software integrieren

Die Aufgabe der Qualitätssicherung ist es, die sicherheitsbezogenen Testaktivitäten im Lebenszyklusprozess zu definieren und in Bezug auf ihre Ziele, Umfänge und Ressourcen zu konkretisieren. Hierzu zählen insbesondere die Definition der Umfänge sowie der festen Zeitpunkte, zu denen sicherheitsbezogene Testaktivitäten im Lebenszyklus stattfinden sollten. Dies sind u.a.:

Sicherheitsbezogene Testaktivitäten so früh wie möglich starten

- Die Definition von Zielen, Zeitpunkten und Ressourcen für Reviewaktivitäten
- Die Definition von Zielen, Zeitpunkten und Ressourcen für das statische und dynamische Sicherheitstesten
- Die Definition von Eingangs- und Endekriterien für einzelne Entwicklungsartefakte in einem Entwicklungsprojekt bzw. für eine konkrete

Grundsätzlich lässt sich festhalten, dass sicherheitsbezogene Testaktivitäten so früh wie möglich starten sollten.

Hinweis: Frühzeitige Umsetzung von Qualitätssicherungsmaßnahmen (Shift Left)

Die frühe Umsetzung von sicherheitsbezogenen Qualitätssicherungsaktivitäten führt dazu, dass Schwachstellen und Verstöße gegen Entwicklungsrichtlinien frühzeitiger erkannt und in der Regel leichter und damit kostengünstiger beseitigt werden können.

Während in Kapitel 3 dieses Buches ein allgemeiner Sicherheitstestprozess beschrieben wurde, wird im Rahmen der folgenden Abschnitte die konkrete Integration einzelner Aktivitäten aus einem solchen Sicherheitstestprozess mit den Aktivitäten eines Softwarelebenszyklusmodells dargestellt. Die Beschreibung ist dabei so allgemein gehalten, dass eine Abbildung auf beliebige Lebenszyklusmodelle, seien es iterative oder sequenzielle, möglich sein sollte.

Exkurs: DevOps und der Softwarelebenszyklus

Neue Entwicklungsphilosophien wie Continuous »Everything« und DevOps gestalten die Übergänge zwischen den einzelnen Phasen eines Lebenszyklus deutlich transparenter und durchgängiger, als wie es der diesem Buch zugrunde liegende Lehrplan des GTB und damit auch dieses Kapitel andeutet. Die DevOps-Philosophie verzahnt und integriert organisatorisch und kulturell ehemals vollständig unabhängige Bereiche wie die Softwareentwicklung und den IT-Betrieb. Der technische Kern des Ansatzes ist eine durchgängige Virtualisierung der IT-Infrastruktur sowie eine möglichst effiziente und umfassende Automatisierung. Dies wird u.a. dadurch erreicht, dass die Praktiken, Techniken, Methoden und Werkzeuge durchgängig harmonisiert werden. Das Ziel von DevOps ist es, die Entwicklung und den Betrieb von Software dadurch effizient zu gestalten, dass Abläufe automatisiert werden und schnelle Entwicklungs- und Deploymentzyklen eine rasche Reaktion auf Kundenanforderungen und Fehler im Betrieb erlauben. Die Herausforderung bezüglich der IT-Sicherheit besteht darin, bei gleichem Automatisierungsniveau die Umsetzung der Sicherheitsanforderungen effektiv realisieren und prüfen zu können und den mit der Verzahnung der Prozesse einhergehenden Kommunikations- und Abstimmungsaufwand bewerkstelligen zu können. IT-Sicherheit muss sich sowohl organisatorisch wie auch technisch in den DevOps-Prozess integrieren und hochgradig automatisieren lassen, um die hochgesteckten Effizienz- und Qualitätsziele zu erreichen. Für den Sicherheitstest bedeutet das insbesondere die Integration automatisierter Codeanalysen, Fuzz-Tests und anderer automatisierbarer Sicherheitstests in den DevOps-Prozess. Grundsätzlich bedarf

es Mitarbeiter, die den erhöhten Anforderungen nach Kommunikation und Abstimmung gerecht werden können.

4.2 Die Rolle des Sicherheitstestens in der Anforderungsermittlung

Die Anforderungsermittlung ist zentraler Ausgangspunkt für eine systematische Softwareentwicklung. Abhängig vom gewählten Vorgehensmodell sowie den betrieblichen Vorgaben werden Anforderungen auf vielerlei Art und Weise erfasst und definiert. Ziel der Anforderungsermittlung ist es, die Software auf einem hohen Abstraktionsniveau zu beschreiben und von ihrer Umgebung abzugrenzen. Die Herausforderung besteht insbesondere darin, für die Beschreibung das richtige Abstraktionsniveau zu finden und alle relevanten Perspektiven zu berücksichtigen, aus denen Anforderungen an die Software gestellt werden müssen.

Prüfung der Sicherheitsanforderungen auf Vollständigkeit, Richtigkeit, Verständlichkeit, Eindeutigkeit, Konsistenz und Testbarkeit

Die Qualitätssicherung, d.h. auch der Sicherheitstest in der Anforderungsermittlung, hat zwei verschiedene Ziele. Einerseits sollten die Anforderungen, d.h. die direkten Ergebnisartefakte der Anforderungsermittlung, validiert und verifiziert werden. Hierzu zählt insbesondere die Prüfung der Anforderungen auf Vollständigkeit, Richtigkeit, Verständlichkeit, Eindeutigkeit, Konsistenz und Testbarkeit. Zum anderen können erste Aktivitäten initiiert werden, die einen anforderungs- und risikobasierten Sicherheitstest in den nachfolgenden Phasen des Lebenszyklus vorbereiten. Letzteres wird insbesondere durch die Durchführung einer systematischen und umfassenden Risikoanalyse unterstützt.

Hinweis: Die Prüfung der Anforderung erfolgt größtenteils durch Reviews

Die Prüfung der Anforderung auf Vollständigkeit, Richtigkeit, Verständlichkeit, Eindeutigkeit, Konsistenz und Testbarkeit erfolgt in der Regel durch Reviews, d.h. durch die Sichtung der Anforderungsartefakte und ihre Bewertung.

Prüfung der Vollständigkeit entlang systematisch definierte Abdeckungsziele

Vollständigkeit wird im Allgemeinen dadurch festgestellt, dass geprüft wird, ob alle notwendigen Stakeholder und ihre Anforderungen berücksichtigt wurden. Zu den Stakeholdern speziell beim Thema Sicherheit zählen natürlich die Anwender und Betreiber der Software, aber auch Regulierungs- und Zulassungsbehörden, die in Vertretung der Gesellschaft

besonders relevante Themen wie den Schutz personenbezogener Daten und die Absicherung kritischer Infrastrukturen regulieren.

Auf der Ebene der Entwicklungsartefakte lassen sich die folgenden Abdeckungsziele prüfen:

- **Abdeckung der Sicherheitsziele**

  Sind alle Sicherheitsziele durch die Sicherheitsanforderungen abgedeckt und ist die Konkretisierung der Ziele in Form der Anforderungen ausreichend, um die Ziele umsetzen zu können?

- **Abdeckung der Compliance-Anforderungen (z.B. zu Standards und Regulierungen)**

  Sind die Compliance-Anforderungen dokumentiert und die sich aus den Compliance-Anforderungen ergebenden Sicherheitsanforderungen spezifiziert?

- **Abdeckung der Datenschutzanforderungen**

  Sind die Datenschutzanforderungen dokumentiert und ausreichend konkretisiert, um den sicheren und datenschutzkonformen Umgang mit personenbezogenen Daten gewährleisten zu können?

- **Abdeckung der identifizierten Risiken (z.B. Vermeidung gängiger Schwachstellen, Abdeckung der Gegenmaßnahmen)**

  Sind alle Sicherheitsrisiken und die zur Minderung der Risiken notwendigen Anforderungen an Sicherheitsmaßnahmen dokumentiert?

Hinweis: Fehler durch Auslassungen

Fehler durch Auslassungen im Anforderungsmanagement können sehr teuer werden, da auf Basis der Anforderungen bereits in frühen Phasen der Softwareentwicklung weitreichende Entscheidungen für ein Softwareentwicklungsprojekt getroffen werden. Werden beispielsweise aufgrund fehlender Sicherheitsanforderungen falsche Architektur- und Technologieentscheidungen getroffen, können diese später nur unter großem Ressourcenaufwand revidiert werden. Die Gefahr, dass insbesondere Sicherheits- und Datenschutzanforderungen gegenüber rein funktionalen Anforderungen zurückfallen, ist groß, weil gerade die funktionale Perspektive in der frühen Projektphase führend ist.

Beispiel: Recht auf Vergessenwerden

Beispiel: Recht auf Vergessenwerden

Als ein gutes Beispiel dafür, dass unberücksichtigte Sicherheits- und Datenschutzanforderungen zu größeren Änderungen an der Softwarearchitektur führen können, kann das in der DSGVO geforderte »Recht auf Vergessenwerden« betrachtet werden [DSGVO 16] (vgl. auch Kap. 9). Demnach sind personenbezogene Daten unverzüglich zu löschen, sobald die Daten für den ursprünglichen Verarbeitungszweck nicht mehr notwendig sind bzw. die betroffenen Personen ihre

Einwilligung zur Datenverarbeitung widerrufen haben. Eine solche Anforderung kann im Konflikt mit modernen Softwarearchitekturen stehen, die eine dezentrale, transaktionsbezogene Speicherung von Daten vorsehen wie beispielsweise beim Event Sourcing oder in der Blockchain. Das einfache Löschen von Daten ist in solchen transaktionsbezogenen Datenstrukturen in der Regel nicht vorgesehen und würde die Integrität der gesamten Datenstruktur zerstören. Soll dennoch gelöscht werden, lässt sich das üblicherweise nur über kryptografische Verfahren realisieren, bei denen die Daten einzeln verschlüsselt werden und im Fall einer Löschanfrage der Schlüssel vernichtet wird.

Die Prüfung der Richtigkeit der Anforderungen erfolgt durch Sicherheitsexperten als Einzelfallprüfung. Verständlichkeit, Eindeutigkeit, Konsistenz und Testbarkeit werden in der Regel durch Experten des Anforderungsmanagements zusammen mit allen anderen Anforderungen geprüft. In diesem Zusammenhang haben die folgenden Aspekte der Anforderungsermittlung einen direkten Einfluss auf die Qualität der Anforderungen und sollten bei Validierung und Verifikation gesondert berücksichtigt werden.

- Es bedarf besonderer Kompetenzen, die Anforderungen der Stakeholder umfassend zu verstehen, sodass man in der Lage ist, sie in Dokumenten niederzulegen oder in Anforderungsmanagementwerkzeuge einzugeben. Speziell für Compliance- und Datenschutzanforderungen muss geprüft und sichergestellt werden, dass die entsprechenden Kompetenzen (z.B. das juristische Verständnis von Regulierungen und Gesetzestexten) vorhanden sind.
- Anforderungen sollten Vorgaben im Hinblick auf Qualitätsmerkmale wie Sicherheit, Leistung, Brauchbarkeit usw. enthalten. Diese Merkmale werden zugunsten der reinen Funktionalität jedoch häufig übersehen.
- Anforderungen ändern sich im Verlauf eines Projekts mit großer Wahrscheinlichkeit. Die Gültigkeit, Konsistenz und Aktualität aller Anforderungen sollte zu definierten Zeitpunkten immer wieder neu geprüft werden.
- Anforderungen können Lücken und Fehler enthalten. Daher ist sowohl eine Verifizierung und wie auch eine Validierung erforderlich.

Checkliste für die Prüfung der Vollständigkeit und Korrektheit von Sicherheitsanforderungen

Eine wirksame Technik bei der Evaluierung der Vollständigkeit und Korrektheit von Anforderungen ist die Nutzung einer Checkliste als Leitfaden für die Prüfung. Diese Checkliste kann eine Vielzahl von Punkten enthalten, um viele Themenbereiche abzudecken. Im Hinblick auf die sicherheitsbezogenen Merkmale sind die folgenden Fragen in Tabelle 4–2 ein guter Ausgangspunkt für eine Validierung.

Datenschutz	Wurden alle Benutzergruppen und ihre Datenschutzerfordernisse ermittelt und dokumentiert?
	Wurden alle Datentypen, die von dieser Anforderung betroffen sind, ermittelt und die entsprechenden Datenschutzerfordernisse definiert?
	Wurden die Benutzerzugriffsrechte ermittelt und dokumentiert?
Compliance (zu Sicherheitsrichtlinien)	Wurden alle relevanten Sicherheitsrichtlinien ermittelt und dokumentiert?
	Wurden Ausnahmen von Sicherheitsrichtlinien ermittelt und dokumentiert?
Gängige Schwachstellen	Wurden alle gängigen und bekannten Sicherheitsschwachstellen für die zu dokumentierende Funktion als bekannte Risiken ermittelt?
Testbarkeit	Sind die Anforderungen so formuliert, dass auf der Basis dieses Dokuments Sicherheitstests und andere Tests geschrieben werden können?
	Werden zu vage Formulierungen wie »die Verarbeitung muss sicher sein« und »Zugang wird nur autorisiertem Personal gewährt« ermittelt und präzisiert, damit sie konkret und testbar sind?
Benutzerbarkeit	Spiegeln die Anforderungen einen Sicherheitsprozess wider, der in Relation zu der zu spezifizierenden Funktion angemessen ist?
	Sind die Sicherheitsverfahren aussagekräftig und verständlich?

	Werden Maßnahmen spezifiziert, die legitimierten Benutzern, die Probleme beim Zugriff auf Informationen haben, Hilfe zur Verfügung stellen?
Leistung	Spiegeln die Anforderungen eine in Relation zu der zu spezifizierenden Funktion angemessene Wirksamkeit der Sicherheitsvorkehrungen wider?

Tab. 4–2 *Beispielfragebogen zur Evaluation von Anforderungen aus dem Lehrplan des GTB Security Tester*

4.3 Die Rolle des Sicherheitstestens beim Entwurf

Während der Entwurfsphase sind sicherheitsgefährdende Entwurfspraktiken zu ermitteln und zu vermeiden. Testbezogene Aktivitäten wie Reviews tragen zur Erkennung von Entwurfsentscheidungen bei, die wahrscheinlich anfällig für Angriffe sind. Sie steuern den Entwurf von Softwaresystemen mit starken, erkennbaren Sicherheitseigenschaften.

Definition einer sicheren Softwarearchitektur, die Analys der Angriffsoberfläche und die Durchführung einer Risiko- bzw Bedrohungsanalyse

Folgt man den Empfehlungen des Microsoft SDL, sind die wichtigen sicherheitsbezogenen Aktivitäten in der Entwurfsphase

- die Definition der Designanforderungen bzw. deren Umsetzung in einer sicheren Softwarearchitektur,
- die Analyse der Angriffsoberfläche mit dem Ziel, potenzielle Angriffsvektoren zu identifizieren, und
- die Durchführung einer Risiko- bzw. Bedrohungsanalyse auf Basis der identifizierten Angriffsvektoren.

Alle drei Aktivitäten bauen aufeinander auf und erlauben es, bereits in der Entwurfsphase schwerwiegende Sicherheitsprobleme zu identifizieren und entsprechende Gegenmaßnahmen zu etablieren, um die Angriffsoberfläche möglichst klein zu halten und die Auswirkungen eines Angriffs zu minimieren.

Prinzipien für sichere Software sind bereits in der Entwurfsphase von Bedeutung.

Die OWASP führt eine Liste mit etablierten Prinzipien für sichere Software [OWASP 16c], die bereits in der Entwurfsphase von hoher Bedeutung

sind. Im Folgenden werden exemplarisch Auszüge daraus erläutert und mithilfe von Beispielen konkretisiert.

- **Etablierung sicherer Standardeinstellungen bzw. -prozeduren**

  Die Etablierung sicherer Standardeinstellungen und -prozeduren hilft grundsätzlich dabei, Sicherheitslücken zu vermeiden, die durch Auslassungen oder Unterlassungen entstehen.

Beispiel: Etablierung sicherer Standardeinstellungen bzw. -prozeduren – Absicherung der Kommunikation als Standardeinstellung

Beispiel: Etablierung sicherer Standardeinstellungen bzw. -prozeduren – Absicherung der Kommunikation als Standardeinstellung

Softwarearchitekturen bzw. -plattformen sollten so gestaltet sein, dass gewünschte Sicherheitseigenschaften, wie z.B. die Authentisierung eines Kommunikationspartners, automatisch umgesetzt sind und nicht mehr explizit durch den Entwickler für jeden neu zu entwickelnden Service separat realisiert werden müssen. Für etablierte Technologien wie Java und .NET existieren eine Reihe von Security Frameworks wie z.B. Spring Security, Apache Shiro, ASP.NET, die es erlauben, Sicherheitsmaßnahmen plattformweit zu konfigurieren und damit als Standardeinstellungen zu etablieren. Die Herausforderung beim Testen besteht darin, diesen Sachverhalt so zu prüfen, dass auch sichergestellt ist, dass er für alle Services auch wirklich umgesetzt worden ist.

- **Verwendung des geringsten Privilegs bzw. der geringsten Berechtigung**

  Prozesse und Nutzer sollten ausschließlich die geringsten Privilegien und Berechtigungen zugewiesen bekommen, mit denen sie ihre Prozesse noch ausführen zu können. Dies umfasst alle Formen der Zugangsberechtigung, Benutzerrechte, Dateisystemberechtigungen sowie den Zugriff auf Ressourcen wie CPU, Speicher und Netzwerke.

Beispiel: Prinzip des geringsten Privilegs bzw. der geringsten Berechtigung – Nutzerkonten

Beispiel: Prinzip des geringsten Privilegs bzw. der geringsten Berechtigung – Nutzerkonten

Beim Einrichten von Benutzerkonten sollten die Berechtigungen so ausgelegt sein, dass sie genau die aktuellen Anforderungen an das Nutzerprofil widerspiegeln. So weist das BSI im IT-Grundschutz darauf hin, dass Berechtigungen bei Bedarf Schritt für Schritt anzupassen sind und nicht vorsorglich, evtl. im Hinblick auf zukünftige Aufgaben, erteilt werden sollten (»need to know«-Konzept).

- **Verwendung einer gestaffelten Verteidigung (Defense in Depth)**

  Sicherheitsmechanismen sollten sich gegenseitig so ergänzen, dass, sollte ein Mechanismus ausfallen bzw. verwundbar sein, weitere Sicherheitsmechanismen die Schutzfunktion (zumindest teilweise)

übernehmen und damit die grundsätzliche Sicherheit des Systems aufrechterhalten werden kann.

Beispiel: Prinzip der gestaffelten Verteidigung – Verschlüsselung auch für die interne Kommunikation

Beispiel: Prinzip der gestaffelten Verteidigung – Verschlüsselung auch für die interne Kommunikation

Ein gutes Beispiel für eine gestaffelte Verteidigung ist die Nutzung von Verschlüsselung auch für die interne Kommunikation. Selbst wenn die ausgetauschten Daten im Prinzip von allen Mitarbeitern gesehen werden dürfen und Firewalls den Zugriff von außen unmöglich machen sollen, realisiert der Einsatz von Verschlüsselung in firmeninternen Netzwerken bzw. in der Kommunikation zwischen lokalen Softwarekomponenten eine zusätzlichen Absicherung, die auch Daten dann noch schützt, wenn eine Firewall bzw. andere Maßnahmen des Zugriffsschutzes überwunden wurden.

- **Kein Vertrauen in die Sicherheit externer Dienste und Bibliotheken**

  Viele Organisationen oder Unternehmen nutzen die Verarbeitungsfunktionen von Drittanbietern, die in der Regel auf anderen Sicherheitsannahmen beruhen und andere Sicherheitseigenschaften besitzen als die eigene Infrastruktur. Dementsprechend ist darauf zu achten, dass externe Funktionen und Dienste auf ihre Sicherheitseigenschaften evaluiert und entsprechend dem eigenen Sicherheitsniveau integriert werden.

Beispiel: Kein Vertrauen in die Sicherheit externer Dienste und Bibliotheken – GNU C Library Buffer Overflow Vulnerability (CVE-2015-7547)

Beispiel: Kein Vertrauen in die Sicherheit externer Dienste und Bibliotheken – GNU C Library Buffer Overflow Vulnerability (CVE-2015-7547)

Die GNU C Library wird in vielen Applikationen verwendet. Im Jahr 2015 wurde entdeckt, dass eine Funktion für den DNS Lookup eine Pufferüberlauf-Schwachstelle enthält, die sich dazu nutzen lässt, bösartigen Code auszuführen bzw. das System abstürzen zu lassen. Die Schwachstelle blieb über sieben Jahre lang unentdeckt. Obwohl nach 7 Monaten eine aktualisierte Version der GNU C Library zur Verfügung stand, stellte sich trotzdem das Problem, wie die betroffenen Applikationen zu aktualisieren sind, da nicht allein die GNU C Library ausgetauscht werden musste, sondern auch die betroffenen Applikationen vor einer Aktualisierung hätten neu übersetzt werden müssen.

- **Strikte Aufgabentrennung**

  Die Trennung von Zuständigkeiten bei der Erledigung kritischer Aufgaben kann als ein etabliertes Verfahren angesehen werden, um den Missbrauch von Privilegien zu vermeiden bzw. zu verringern. Durch eine systematische Trennung von Aufgaben und Funktionsbereichen in der Software können nach diesem Prinzip auch robuste Softwaresysteme realisiert werden.

Prozesse bzw. Applikationen, die ausschließlich einen wohldefinierten und abgegrenzten Aufgaben- bzw. Funktionsbereich bedienen, können abgestimmt auf die jeweilige Kritikalität des Aufgabenbereichs gezielt geschützt werden. Im Fall ihrer Kompromittierung entsteht der Schaden dann auch nur in diesem abgegrenzten Bereich.

Beispiel: Aufgabentrennung – Hardware-Sicherheitsmodule (HSM)

Beispiel: Aufgabentrennung – Hardware-Sicherheitsmodule (HSM)

Für die effiziente und sichere Durchführung kryptografischer Funktionen und zur sicheren Verwahrung von Schlüsseln werden vermehrt Hardware Security Modules (HSM) eingesetzt. Ein HSM bietet Schutz gegen unberechtigten Zugriff auf und die Manipulation von sicherheitsrelevanten Informationen, indem diese Daten über einen eigenen Hardwarebereich geschützt werden und der Zugriff über eine definierte Hardwareschnittstelle erfolgt. Es gibt HSM, deren Sicherheit durch Zertifikate belegt werden, sodass ein dediziertes Sicherheitsniveau garantiert werden kann. Der damit einhergehende Aufwand bei der Entwicklung und Zertifizierung rechnet sich nur für standardisierte Spezialaufgaben wie z.B. das Schlüsselmanagement, d.h., wenn die im HSM verarbeiteten Daten durch sinnvolle Aufgabentrennung auf ein Minimum reduziert werden können.

- **Sicherheit einfach halten**

  Die Komplexität einer Software hat häufig einen direkten Einfluss auf die Größe der Angriffsoberfläche dieser Software. Im Hinblick auf die Sicherheit ist die Verwendung einfacher Lösungen der Verwendung komplexer Lösungen vorzuziehen, weil komplexe Software meist fehleranfälliger ist und sich deutlich schwerer prüfen und warten lässt.

Beispiel: Sicherheit einfach halten – Single Sign-on

Beispiel: Sicherheit einfach halten – Single Sign-on

Single Sign-on bietet die Möglichkeit, Nutzern über ein und denselben Authentifizierungsmechanismus Zugriff auf verschiedene Ressourcen in einer heterogenen Infrastruktur zu geben. Entgegen der landläufigen Meinung, dass unterschiedliche Authentifizierungsmechanismen mit jeweils unterschiedlichen Credentials einen höheren Grad an Sicherheit bieten, besticht ein sauber aufgesetztes Single Sign-on-System durch die Einfachheit der Nutzung und durch eine technisch einfachere Lösung (vgl. Abschnitt 5.2). Beispielsweise muss sich ein Nutzer nicht mehr verschiedene Benutzernamen und Passwörter merken, sondern hat nur noch eines. Zusätzlich erfolgt die Authentifizierung an nur einer Stelle und sensible Daten wie Passwörter müssen auch nur einmal übertragen werden. Dieses entspricht zusätzlich noch dem oben beschriebenen Prinzip Aufgabentrennung.

Die oben genannten Prinzipien sind Beispielhaft, zeigen aber bereits, dass sich Sicherheitsprinzipien nicht immer widerspruchsfrei miteinander kombinieren lassen. Beispielsweise besteht bei der gestaffelten Verteidigung die Herausforderung darin, einen angemessenen Schutz durch sich ergänzende

Absicherungsmaßnahmen zu gewährleisten, ohne das Prinzip der Einfachheit zu verletzen. Das Hinzufügen eines neuen Sicherheitsprotokolls durch eine zusätzliche Sicherheitsmaßnahme kann beispielsweise aufgrund der dadurch gewachsenen Komplexität zu neuen Risiken führen. Daher ist auch bei der Anwendung der Prinzipien für sichere Software darauf zu achten, dass diese aufeinander abgestimmt werden, sodass eine ausgewogene und dadurch optimale Sicherheitslösung realisiert werden kann.

4.4 Die Rolle des Sicherheitstestens während der Implementierung

Wie auch beim Testen funktionaler Eigenschaften wird mit dem Sicherheitstest der Software bereits in den frühen Phasen der Implementierung begonnen. Erste Tests finden auf der Ebene des Modul- bzw. Komponententests statt, sodass sicherheitsbezogene Fehler bereits in der Implementierung gefunden werden können. Die Testobjekte sind in der Regel die separat entwickelten Softwarekomponenten, die später das Gesamtsystem bilden. Statische Tests und Reviews stellen sicher, dass sowohl die organisatorisch technischen Grundlagen für die Implementierung sicherer Software existieren als auch die Implementierungsartefakte den Anforderungen einer sicheren Implementierung genügen. Nach der statischen Evaluierung bietet das dynamische Testen dieser Komponenten die erste Möglichkeit, Sicherheitseigenschaften entlang des dynamischen Verhaltens, z.B. in Reaktion auf gültige und ungültige Eingaben, zu prüfen.

4.4.1 Der statische Test von Softwarekomponenten

Statische Tests und Reviews zur Prüfung von Implementierungsartefakten

Der statische Test von Softwarekomponenten umfasst das gesamte Spektrum an Inspektionen, Walkthroughs, Audits und Reviews. Ziel der statischen Tests ist u.a. die Prüfung der Einhaltung von Best Practices bzw. Empfehlungen für die sichere Codierung. Solche Best Practices bzw. Empfehlungen gibt es viele, die sich in Granularität, Umfang und abhängig von der konkret adressierten Anwendungsdomäne bzw. Programmiersprache unterscheiden. So finden sich bei der OWASP [OWASP 10] und bei SANS [SANS 2] Empfehlungen, die speziell auf die Implementierung sicherer Webanwendungen zugeschnitten sind. Ein sehr allgemeines Beispiel für Best Practices bzw. Empfehlungen beim Programmieren sicherer Software enthält der Artikel »Top 10 Secure Coding Practices« [CMU 1]. Dort heißt es:

Beipiel: Sichere Programmierpraktiken

Beipiel: Sichere Programmierpraktiken

»Die Tests einer jeden Komponente sollten die Prüfung auf mögliche Verstöße gegen diese Praktiken einschließen:

- Validierung von Eingaben
- Compiler-Warnungen beachten
- Architektur und Entwurf gemäß den Sicherheitsrichtlinien
- Einfach halten (Keep it Simple)
- Standardmäßiges Blockieren (Default Deny)
- Einhaltung des Grundsatzes der geringstmöglichen Zugriffsrechte
- Bereinigung von Daten, die an andere Systeme geschickt werden
- Gestaffelte Sicherheitsarchitektur (Defense in Depth)
- Nutzung wirksamer Qualitätssicherungstechniken
- Anwendung eines sicheren Programmierstandards«

Die Prüfung der Umsetzung solcher Best-Practice-Checklisten findet im Allgemeinen durch Inspektionen, Audits und technische Reviews statt. Bei der Prüfung gilt es, auf die wichtigsten Anforderungen im Hinblick auf die Angriffswahrscheinlichkeit und das Schadensausmaß zu fokussieren. Eine gut dokumentierte Risikoanalyse, die eine realistische Gefährdungsmodellierung einschließt, stellt hierfür eine wichtige Grundlage dar.

Exkurs: Sichere Programmierstandards

Je stärker die Anwendung eines sicheren Programmierstandards und damit die Prüfung individueller Programmiersprachenkonstrukte Gegenstand der Prüfung ist, desto wichtiger wird der Einsatz statischer Analysewerkzeuge, mit denen sich speziell diese Programmiersprachenkonstrukte in großen Programmen effizient prüfen lassen und so eine Reihe bekannter Schwachstellen wie z.B. Pufferüberläufe und fehlende Eingangsvalidierung automatisiert ermittelt werden können. Sichere Programmierstandards existieren für viele bekannte Programmiersprachen und -plattformen (wie z.B. C, C++, Java und Perl sowie der Android-Plattform). Diese Standards definieren Richtlinien für die sichere Programmierung, sodass einerseits Sicherheitslücken vermieden werden und andererseits Programmcode entsteht, der sich besser verstehen und pflegen lässt. Beispiele sind unter anderem die MISRA C-Richtlinien für sichere C-, C++-Programmierung [MISRA 13], die ISO/IEC TS 17961:2013 [ISO 17961] sowie die CERT Secure Coding Standards [CMU 3]

Mit statischen Analysewerkzeugen Programmcode oder

Mit statischen Analysewerkzeugen lassen sich Programmcode oder Binärdaten in größerem Umfang

automatisiert prüfen. Diese Werkzeuge sind hochgradig spezialisiert und liefern gute Ergebnisse für die Programmiersprachen, auf die sie zugeschnitten sind. Ihre Stärke liegt in der Analyse vieler einzelner Komponenten. Die ganzheitliche Bewertung großer Anwendungen sowie die Analyse heterogener Technologiestacks bestehend aus Komponenten, die auf verschiedenen Programmiersprachen beruhen, stellt jedoch eine große Herausforderung dar. Statische Analysewerkzeuge ermitteln in diesem Kontext oft zu viele falsch positive Ergebnisse: Sie zeigen Schwachstellen an, die in Wirklichkeit nicht existieren. Dies passiert häufig deshalb, weil es für diese Werkzeuge extrem schwierig ist, den gesamten Kontext einer analysierten Softwarekomponente korrekt zu erfassen und zu bewerten. Speziell die Bewertung der Integrität und Absicherung von Datenflüssen, die sich in einer Anwendung über verschiedene Komponenten erstrecken, ist eine große Herausforderung. Das Problem vergrößert sich, wenn Komponenten eingesetzt werden, zu denen kein Quellcode verfügbar ist wie z.B. externe Komponenten oder Komponenten von Drittanbietern, die ohne Quellcode ausgeliefert werden.

Binärdaten automatisiert prüfen

4.4.2 Der dynamische Test von Softwarekomponenten

Der dynamische Test von Softwarekomponenten ist ein Test, bei dem die zu testende Software tatsächlich ausgeführt wird. Der dynamische Sicherheitstest sollte beginnen, sobald die ersten ausführbaren Softwarekomponenten verfügbar sind. Sogenannte Whitebox- und Glassbox-Sicherheitstests können durchgeführt werden, wenn Zugriff auf die Entwicklungsdokumente und auf den Quellcode vorliegt. Stehen Entwurfsdokumente und Quellcode nicht zur Verfügung, wie beispielsweise bei Bibliotheken und Komponenten von Drittanbietern, können nur anforderungsbasierte und risikobasierte Blackbox-Sicherheitstests durchgeführt werden.

Mit dem dynamischen Sicherheitstest beginnen, sobald die ersten ausführbaren Softwarekomponenten verfügbar sind

4.4.2.1 Whitebox- und Glassbox-Sicherheitstests

Whitebox- oder Glassbox-Tests sind Tests, die auf Grundlage von Strukturinformationen aus den Artefakten des Softwareentwurfs oder der Softwareimplementierung hergeleitet, ausgewählt, durchgeführt und ausgewertet werden. Die Strukturinformationen werden verwendet, um den Testentwurf zu steuern sowie Überdeckungsmaße und Testendekriterien zu definieren. Beim Blackbox-Testen hingegen stehen diese Strukturinformationen nicht zur Verfügung, sodass allein auf Basis der Anforderungen sowie der verfügbaren

Informationen über die Schnittstelle einer Softwarekomponente getestet werden muss.

Durch Whitebox-Sicherheitstests bzw. strukturelle Sicherheitstests lässt sich zielgerichtet die Implementierung einzelner Kontrollmechanismen prüfen. Der Einblick in die Komponentenstruktur bzw. den Quellcode erlaubt es, die Tests speziell auf die Art der Implementierung der Kontrollmechanismen abzustimmen, und ermöglicht die Messung des Überdeckungsgrades der Tests als Prozentwert der ausgeführten ausführbaren Anweisungen, als Prozentwert der ausgeführten Entscheidungsergebnisse oder als Prozentwert der durchlaufenen Logikpfade.

Beispiel: Whitebox-Test einer Zahlungsfunktion für Finanztransaktionen

Beispiel: Whitebox-Test einer Zahlungsfunktion für Finanztransaktionen

Bei der Entwicklung einer E-Commerce-Anwendung wurde im Rahmen einer Risikobeurteilung festgestellt, dass betrügerische Transaktionen zwischen der Anwendung und einer von der Anwendung genutzten externen Zahlungsschnittstelle gravierende Auswirkungen haben können. Im Rahmen einer Whitebox-Analyse wurden alle Komponentenschnittstellen identifiziert und Vertrauensgrenzen für die Komponenteninteraktionen festgelegt. In einem weiteren Schritt wurden die Datenflüsse zwischen den Komponenten identifiziert und analysiert. Es zeigte sich, dass es einen Programmpfad gab, in dem die Eingaben der Benutzer nicht überprüft oder authentifiziert wurden, sodass Zahlungen anonym durchgeführt werden konnten. Der Sachverhalt wurde in einem Testfall operationalisiert, mit dem sich anonym ein Transfer zwischen einem externen Konto an das Händlerkonto durchführen ließ. Es konnte also gezeigt werden, dass die Anwendung eine nicht autorisierte Transaktion über nicht authentifizierte Kanäle zugelassen hatte. Voraussetzung für die Ableitung und Durchführung des Testfalls war einerseits die systematische Risikoanalyse sowie andererseits die systematische Analyse des Quellcodes, d.h. das Whitebox-Vorgehen zur Testableitung (aus [Janardhanud & van Wyk 2005]).

Strukturelle Sicherheitstests können von automatisierten Analysewerkzeugen und Sicherheitsscannern durchgeführt werden.

Strukturelle Sicherheitstests sind automatisierbar.

Beispiel: Fuzz-Testing als Whitebox-Test

Beispiel: Fuzz-Testing als Whitebox-Test

Fuzz-Tests sind eine Sicherheitstesttechnik, die insbesondere die Robustheit eines Systems durch Eingabe großer Mengen von Zufallsdaten bzw. Daten mit Zufallsanteilen in die getestete Komponente bzw. das System prüft. Sicherheitsschwachstellen werden dadurch gefunden, dass diese Daten nichtintendiertes Verhalten auslösen können, das sich dazu nutzen lässt, die Integrität, Verfügbarkeit und Vertraulichkeit des Systems zu unterminieren. Beim Blackbox-Fuzzing besteht die Herausforderung insbesondere darin, das nichtintendierte Verhalten und damit die Fehlerwirkung zu erkennen sowie geeignete Testendekriterien zu definieren. Die Instrumentierung von Code kann dazu genutzt werden, beide Herausforderungen systematisch anzugehen. Durch die Instrumentierung bekommt der Tester Rückmeldung darüber, welche Konstrukte im Code tatsächlich getestet werden und ob diese in beabsichtigter Weise genutzt werden. Whitebox-Fuzz-

Tests (an kleinen Softwareblöcken, Funktionen, Klassen) ergeben so u.U. in viel kürzerer Zeit brauchbare Ergebnisse als ein Blackbox-Fuzz-Testwerkzeug. Ein Beispiel für ein bekanntes Open-Source-Werkzeug für Whitebox-Fuzz-Testing ist AFL-Fuzz [AFL 18][1].

Durch strukturelles Testen lassen sich u.a. die folgenden Sicherheitsschwachstellen ermitteln:

- Programmierfehler, die zu Speicherpufferüberläufen oder Systemausfällen führen
- Bösartiger Programmcode, der von einem internen Mitarbeiter oder Auftragnehmer eingeschleust wurde
- Zugang über »Hintertüren« wie beispielsweise bewusst eingebaute und nicht dokumentierte Zugangsschnittstellen

4.4.2.2 Anforderungsbasierte und risikobasierte Sicherheitstests

Anforderungsbasierte und risikobasierte Sicherheitstests prüfen die Umsetzung der Sicherheitsanforderungen sowie die Robustheit des Systems gegenüber zusätzlich identifizierten Risiken. Die Angemessenheit von Sicherheitstests sollte auf jeder Testebene durch den Nachweis der Abdeckung spezifizierter Sicherheitsanforderungen und identifizierter Sicherheitsrisiken ermittelt werden. Dies erfolgt ergänzend zur Bewertung der Ergebnisse aus Belastungssituationen, die in den Sicherheitsanforderungen, Sicherheitsrisikobewertungen und ähnlichen Dokumenten nicht explizit aufgeführt sind. Bei der Suche nach Schwachstellen ist grundsätzlich Kreativität gefordert, da Tester untersuchen, was sowohl die Spezifizierer wie auch die Entwickler übersehen haben oder was sich erst nach einer Integration wirklich ergeben hat und daher bis dato nie von einem Entwickler gesehen werden konnte.

4.4.2.3 Abdeckungsmaße zur Bewertung von Sicherheitstests

Eine wichtige Maßzahl für die Angemessenheit von Tests ist die Ermittlung ihres Überdeckungsgrades. Die verschiedenen in der Praxis etablierten Überdeckungsgrade ergeben sich aus der Art des durchgeführten Tests.

Anforderungsbasierte Tests prüfen das System auf korrekt umgesetzte Anforderungen.

Anforderungsbasierte Tests prüfen das System dahingehend, ob es die ihm vorgegebenen Anforderungen erfüllt. Ohne Berücksichtigung der Implementierung (Blackbox) kann der Überdeckungsgrad wie folgt gemessen werden:

- Der Prozentsatz der getesteten Anforderungen

- Der Prozentsatz der spezifizierten und getesteten Anwendungsfälle sowie – speziell beim Sicherheitstesten – der Prozentsatz der spezifizierten und getesteten Missbrauchsfälle und Bedrohungsszenarien
- Der Prozentsatz der getesteten kritischen Funktionen, Szenarien oder Aufgabenpfade

Äquivalenzklassenüberdeckung sowie paarweise und n-weise Überdeckungen als typische Datenüberdeckungskriterien

Beim datengetriebenen Testen wird das Verhalten des Systems über eine Vielzahl von Eingabedaten sowie deren Kombinationen geprüft. Dabei versucht man, so wenig Testwerte wie möglich zu verwenden, indem man den Datenraum in Äquivalenzklassen unterteilt und aus jeder Klasse einen Vertreter wählt. Das geschieht in der Annahme, dass die Elemente einer Klasse im Hinblick auf ihre Fähigkeit, Fehler zu erkennen, äquivalent sind. Paarweise und n-weise Überdeckungskriterien [Spillner & Breymann 16] sind typische Formen von Datenüberdeckungskriterien.

Beispiel: Pairwise Coverage

Beispiel: Pairwise Coverage

Gegeben sei eine Menge von Parametern und Parameterwerten in der Form:

f1 {*f11, f12, f13, f14*},

f2 {*f21, f22, f23, f24*},

f3 {*f31, f32, f33, f34*},

f4 {*f41, f42, f43, f44*}

Die Generierung der Testwerte unter Berücksichtigung der Pairwise Coverage ergibt die folgenden 16 Parametersätze aus den möglichen 256 (d.h. 4^4):

Param.	**f1**	**f2**	**f3**	**f4**
1.	*f11*	*f21*	*f31*	*f41*
2.	*f12*	*f22*	*f32*	*f41*
3.	*f13*	*f23*	*f33*	*f41*
4.	*f14*	*f24*	*f34*	*f41*
5.	*f13*	*f22*	*f31*	*f42*
6.	*f14*	*f21*	*f32*	*f42*
7.	*f11*	*f24*	*f33*	*f42*
8.	*f12*	*f23*	*f34*	*f42*
9.	*f14*	*f23*	*f31*	*f43*
10.	*f13*	*f24*	*f32*	*f43*
11.	*f12*	*f21*	*f33*	*f43*

12.	*f11*	*f22*	*f34*	*f43*
13.	*f12*	*f24*	*f31*	*f44*
14.	*f11*	*f23*	*f32*	*f44*
15.	*f14*	*f22*	*f33*	*f44*
16.	*f13*	*f21*	*f34*	*f44*

Beim modellbasierten Testen lässt sich Überdeckung mit Bezug zur Modellnotation definieren.

Das modellbasierte Testen [Winter et al. 16] ermöglicht die Ermittlung des Überdeckungsgrades im Hinblick auf eine gewählte Modellierungsnotation. Ein modellbasierter Sicherheitstest [Grossmann et al. 17] basiert dann beispielsweise auf Modellen, die Sicherheitseigenschaften, Sicherheitsfunktionalität und/oder Sicherheitsrisiken und -fehler modellieren. Wenn das Modell eine Notation mit Vor- und Nachbedingungen nutzt, können die Vorbedingungen zur Ableitung der Eingabewerte und die Nachbedingungen für die Ableitung der Testorakel verwendet werden. Mittels einer Grenzwertanalyse lassen sich entlang der Vorbedingung gültige und ungültige Eingabewerte identifizieren. Aus den entsprechenden Nachbedingungen können die erwarteten Ausgabewerte für die gültigen Eingaben abgeleitet werden. Zur Messung des Überdeckungsgrades eignen sich die durch die Vor- und Nachbedingungen definierten Ursache-Wirkungs-Ketten und der Überdeckungsgrad aller Alternativen in den Nachbedingungen. Bei algebraischen Modellierungsnotationen werden Daten in Form abstrakter Datentypen und Funktionalität mittels Axiomen spezifiziert, die das Verhältnis zwischen den Datentypen mittels mathematischer Definitionen beschreiben. Als Überdeckungskriterium wird typischerweise der Anteil bzw. der Prozentsatz der überdeckten Axiome einer Spezifikation verwendet.

Bei übergangsbasierten Modellen, die explizite Graphen mit Knoten und Kanten nutzen, gehören der Prozentsatz der Knoten (Zustände), der Prozentsatz der Übergänge (Transitionen), der Prozentsatz der Übergangspaare (Transitionspaare) sowie der Prozentsatz der Zyklen zu den üblicherweise verwendeten Überdeckungskriterien. Zu dieser Klasse von Modellen zählen insbesondere Zustandsautomaten, die in der Praxis weit verbreitet sind und durch gängige Modellierungsnotationen wie der Unified Modeling Language (UML) unterstützt werden.

Beispiel: Modellbasierter Test eines Authentisierungsvorgangs

Die Authentisierung an einer Webapplikation lässt sich durch ein Zustandsmodell modellieren. Im Rahmen eines modellbasierten Tests kann dann beispielsweise eine Überdeckung aller Authentisierungszustände avisiert werden. Die Wahl eines höheren Überdeckungsgrades,

Beispiel: Modellbasierter Test eines Authentisierungsvorgangs

beispielsweise die Überdeckung aller Zustandsübergänge, würde neben der Erreichbarkeit der einzelnen Authentisierungszustände zusätzlich auch das Auslösen der Bedingungen für den Wechsel zwischen den Authentisierungszuständen unterscheiden und prüfen können. Letzteres würde es bei einer Webapplikation erlauben, einen expliziten Logout und einen automatischen zeitabhängigen Logout, wie er in vielen Banking-Applikationen üblich ist, im Test zu unterscheiden.

Beim strukturellen Testen wird die Überdeckung der Programmstruktur bewertet.

Beim strukturellen Testen wird die tatsächliche Implementierung und ihre Struktur auf Basis von Wissen über den Programmcode analysiert. Der Testüberdeckungsgrad wird üblicherweise als Prozentsatz der durch den Test durchlaufenden Strukturelemente, d.h. über Pakete, Klassen, Methoden, Entscheidungen oder Zeilen des ausführbaren Programmcodes, einer Anwendung definiert. Bekannt sind unter anderem die folgenden Abdeckungskriterien, die sich am Kontrollflussgraphen der Software orientieren:

- Die Anweisungsüberdeckung, bei der der Prozentsatz der ausgeführten Anweisungen eines Programmcodes gemessen wird,
- die Zweigüberdeckung mit der Berücksichtigung aller Kanten im Kontrollflussgraphen,
- die Pfadüberdeckung, bei der alle möglichen Pfade durchlaufen werden müssten, sowie
- verschiedene Formen der Bedingungsüberdeckung unter Berücksichtigung der Ergebnisse der atomaren und kombinierten Teilbedingungen.

Das stärkste der oben genannten Kriterien ist die Pfadüberdeckung. Sie wird über alle ausführbaren Pfade vom Eingang bis zum Ausgang des Kontrollflussgraphen gemessen. Weil ein erschöpfendes Testen von Pfaden aufgrund von Schleifen im Allgemeinen nicht durchführbar ist, lassen sich andere, weniger strenge Kriterien heranziehen. Hierzu zählt die Überdeckung kritischer Pfade oder die Zweigüberdeckung. Die Anweisungsüberdeckung ist das schwächste Überdeckungskriterium. Weiterführende Informationen zu diesem Thema finden sich im Ausbildungsprogramm zum Certified Tester.

Exkurs: Zyklomatische Komplexität

Die zyklomatische Komplexität oder auch McCabe-Maß ist eine Maßzahl, die beschreibt, wie viele verschiedene linear unabhängige Pfade durch ein Programmelement existieren und sich mithilfe eines Kontrollflussgraphen mit Knoten (Entscheidungspunkten) und Kanten (Pfaden) visualisieren lassen. Ein Pfad gilt genau dann als linear unabhängig, wenn er mindestens eine neue Kante

enthält. Als Beispiel kann der Java-Programmabschnitt in Listingl 4–1 betrachtet werden. Die Funktion berechnet den größten gemeinsamen Teilers (ggT) nach dem klassischen euklidischen Algorithmus. Abbildung 4–5 zeigt den dazugehörigen Kontrollflussgraf mit zyklomatischer Komplexität V(G)=4.

Listing 4–1

```
int ggT(int x, int y) {

 if(x==0) return y;

  while (y != 0) {

   if (x > y) {x = x - y;}

  else {y = y - x;}

 }

 return x;

}
```

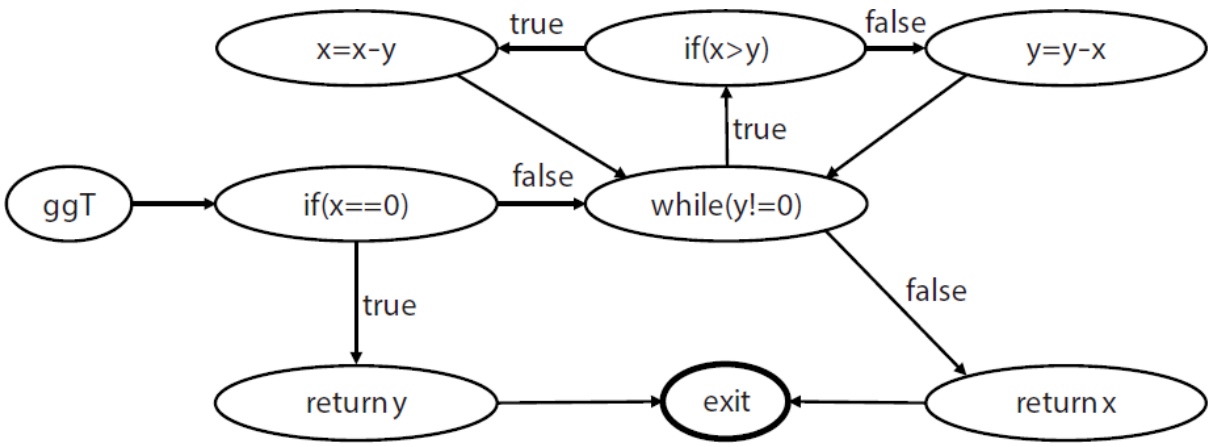

Abb. 4–5 *Kontrollflussgraph für einen Algorithmus zur Berechnung des größten gemeinsamen Teilers (ggT) (angelehnt an [Zeller 02])*

Für einen gegebenen Kontrollflussgraphen lässt sich die zyklomatische Komplexität mit der Formel V (G) = E – N + 2 berechnen, wobei E die Anzahl der Kanten und N die Anzahl der Knoten des Kontrollflussgraphen bezeichnet. Die

zyklomatische Komplexität ist eine Maßzahl für Softwarekomplexität und stellt im Kontext des strukturellen Testens einen oberen Grenzwert für die Anzahl der Tests dar, die notwendig sind um eine vollständige Zweigüberdeckung minimal realisieren zu können. Solange das Programm und der daraus resultierende Kontrollflussgraph klein ist, kann die zyklomatische Komplexität manuell berechnet werden. Bei größeren Programmen mit komplexen Kontrollflussgraphen müssen Werkzeuge zur Ableitung des Kontrollflussgraphen und der Bestimmung der zyklomatischen Komplexität eingesetzt werden.

4.5 Die Rolle des Sicherheitstestens während der Integration & Verifikation

In der Integrations- und Verifikationsphase wird das Softwaresystem sukzessive integriert und getestet. In der Regel werden begleitend zur Integration der Software Integrationstests durchgeführt. Diese Tests haben das Ziel, Fehlerzustände und Fehlerwirkungen in den Schnittstellen und im Zusammenspiel zwischen integrierten Komponenten aufzudecken.

Im Anschluss an die Softwareintegration wird im Systemtest das dann vollständig integrierte System getestet, um sicherzustellen, dass es vollständig den spezifizierten Anforderungen genügt. Sowohl im Integrationstest wie auch im Systemtest lassen sich wichtige Sicherheitstestaktivitäten verorten.

4.5.1 Sicherheitstests während der Komponentenintegration

Weil Komponenten niederer Ebenen in Subsysteme und letztlich in das komplette Zielsystem integriert werden, sind die Möglichkeiten für Sicherheitsverstöße nicht einfach die Summe der Schwachstellen in den einzelnen, separat betrachteten Komponenten. Aufgrund der Interaktionen zwischen Komponenten und mit größeren Systemen und Organisationseinheiten ergeben sich vielmehr neue Angriffsvektoren, die sich bei der Betrachtung der einzelnen Komponenten nicht erkennen lassen bzw. deren Schadwirkung unterschätzt wurde.

Hinweis:

Fehler oder Schwachstellen, die im Integrationstest entdeckt werden, beruhen häufig auf falschen Annahmen zu den Vor-und Nachbedingungen an den Komponentenschnittstellen. Sicherheitsrelevant werden diese falschen Annahmen dann, wenn sie das Vorhandensein von Sicherheitsmechanismen in einer Komponente voraussetzen, die nicht oder nicht korrekt umgesetzt worden ist. Zu den klassischen sicherheitsrelevanten Fehlern gehört die Weitergabe nicht validierter Daten. Beide Komponenten, sowohl die sendende wie auch die empfangende, gehen davon aus, dass die jeweils andere Komponente fehlerhafte Daten bereinigt. Im

schlechtesten Fall werden die Daten ungeprüft verarbeitet bzw. weitergegeben. Fehler wie diese öffnen ggf. ein Einfallstor für Angriffe wie Pufferüberläufe oder Injections bzw. offenbaren Daten, beispielsweise ausführliche Fehlermeldungen, die eigentlich nur gefiltert nach außen getragen werden sollten.

Andererseits können Interaktionen zwischen Komponenten mögliche Abläufe, die zu Sicherheitsverstößen führen, auch mindern oder blockieren. Ein gutes Beispiel dafür ist die Abschirmung fehlerhafter Komponenten durch Schutzmaßnahmen wie zusätzliche Eingabevalidierung (s.u.) oder Firewalls. Auch hier gilt: Sicherheitstester müssen kreativ sein, wenn sie nach Fehlern suchen, die von Entwicklern übersehen wurden.

Beispiel: Längenvalidierung zur Vermeidung von Pufferüberläufen

Beispiel: Längenvalidierung zur Vermeidung von Pufferüberläufen

Häufig stellt sich erst in der Integration dar, ob Sicherheitsmechanismen wie eine Längenvalidierung tatsächlich einen effektiven Schutz für nachgeordnete Komponenten bieten können. So lässt sich häufig erst in der Integration prüfen, ob wirklich alle Eingabekanäle eine effektive Längenvalidierung implementieren, um nachgeordnete Komponenten gegen Pufferüberlauf-Angriffe zu schützen.

Integrationstests dienen dazu, die Integration unterschiedlicher Komponenten, aus denen sich ein späteres System zusammensetzt, zu prüfen. Der Integrationstest ist somit die erste Prüfung der integrierten Komponenten in der erweiterten Komplexität als funktionsfähiges System oder Teilsystem. Die Integration der Komponenten und der damit einhergehende Test erfolgt in der Regel systematisch, d.h. durch eine definierte Vorgehensweise. Die Integration kann iterativ erfolgen, d.h., die Komponenten werden sukzessive integriert. Dieses kann top-down oder bottom-up erfolgen. Beim Top-down-Vorgehen wird mit dem Test der Komponenten auf der höchsten Abstraktionsschicht begonnen. Die Komponenten der tieferen Schichten werden zuerst durch Platzhalter (engl. Teststubs) ersetzt, die dann im Zuge der Integration sukzessive durch reale Komponenten ausgetauscht werden. Beim Bottom-up-Vorgehen werden logisch eng verknüpfte Komponenten zuerst integriert und diese dann sukzessive zu einem Gesamtsystem zusammengesetzt. Testtreiber (engl. Testdriver) übernehmen hier die Aufgabe, die Testeingaben auf die Komponentenschnittstellen anzuwenden.

Das Ziel des Integrationstests ist es, nachzuweisen, dass die Komponenten nach ihrer Integration spezifikationsgemäß funktionieren und dass es durch die Interaktionen zwischen den Komponenten zu keinen zusätzlichen Fehlern kommt. Hierzu werden sowohl Funktionstests, Schnittstellentests und Performanztests durchgeführt. Typische Fehler, die während eines Integrationstests gefunden

werden, sind Fehler aufgrund falscher Schnittstellennutzung, Race Conditions oder Sicherheitsprobleme, die aufgrund falscher Annahmen in Bezug auf die Schutzfunktion anderer Komponenten gemacht wurden. Zu beachten ist, dass die gewählte Integrationstestvorgehensweise (z.B. top-down oder bottom-up) sich auf den Zeitpunkt des Aufdeckens von Sicherheitsproblemen auswirken oder die Notwendigkeit für zusätzliche sicherheitsspezifische Tests erzeugen kann.

Beispiel: Testen einer Bibliothek von Drittanbietern

Beispiel: Testen einer Bibliothek von Drittanbietern

Beim Testen einer Bibliothek eines Drittanbieters wird festgestellt, dass die Bibliothek sich durch Pufferüberlauf-Angriffe in einen instabilen Zustand bringen lässt. Patches und Aktualisierungen, die den Fehler beheben, stehen nicht zur Verfügung. Durch weitere Tests muss nun sichergestellt werden, dass Eingaben, die schlussendlich von der Bibliothek verarbeitet werden, durch andere Komponenten dahingehend geprüft und beschränkt werden, sodass im integrierten System ein Pufferüberlauf nicht realisierbar ist. Der Bottom-up-Integrationsansatz führt in diesem Fall dazu, dass die Schwachstelle der Bibliothek identifiziert und dokumentiert werden kann und dass durch nachfolgende Sicherheitstests geprüft werden kann, inwiefern die Ausnutzung der Schwachstelle durch die Integration mit anderen Komponenten möglich bzw. nicht möglich ist.

Wie Komponententests müssen auch Integrationstests auf der Basis einer gut dokumentierten Risikoanalyse, die eine realistische Gefährdungsmodellierung einschließt, entworfen werden. Wenn separate Komponenten zu einem Ganzen integriert werden, ist u.U. die Nutzung eines Testrahmens (in Form von Platzhaltern und Testtreibern) notwendig, um unvollständige Aufrufpfade auch in einem System während der Integration zu testen. Wenn mehr und mehr implementierte Komponenten zum System hinzukommen, verschwindet dieser Testrahmen schrittweise. Das ermöglicht eine umfassendere Bewertung der Funktionalität sowie die Prüfung neuer Pfade zu Schwachstellen, die potenziell ausgenutzt werden könnten.

4.5.2 Sicherheitstesten während des Systemtests

Im Systemtest werden Sicherheitsanforderungen typischerweise im Kontext von Ende-zu-Ende-Testszenarien geprüft.

Systemtests sind die erste durchgängige Ausführung der vollständig integrierten Komponenten eines Systems. Obwohl sie in der Regel noch nicht in der Zielumgebung stattfinden, sollten sie nahezu vollständig die entstehenden Eigenschaften des Systems offenlegen, die vor dem Abschluss der Integration noch nicht unbedingt zu sehen waren. Im Systemtest werden Sicherheitsanforderungen typischerweise im Zusammenhang mit einer oder mehreren funktionalen Anforderungen betrachtet, sodass vollständige Ende-zu-Ende-Testszenarien entstehen.

Funktionale Anforderungen, einschließlich der sicherheitsbezogenen, sind in der Regel Muss-Anforderungen.

Beispiel: Wahrung der Integrität einer Zahlungsanweisung

Beispiel: Wahrung der Integrität einer Zahlungsanweisung

In einem Softwaresystem für Finanztransaktionen muss sichergestellt werden, dass der Betrag einer Finanztransaktion auf dem Weg zum Zahlungsdienstleister nicht verändert werden kann. Die Integrität ist durch kryptografische Verfahren sicherzustellen. Zu prüfen ist, ob das System, das die Transaktionsdaten bereitstellt, diese Verfahren unter allen Umständen anwendet und beispielsweise eine kryptografische Signatur für den Zahlungsbetrag zur Verfügung stellt und ob das System des Zahlungsdienstleisters diese Signatur auch tatsächlich prüft.

Negativszenarien und Missbrauchsfälle sollten systematisch getestet werden.

Im Systemtest können neben den Sicherheitsanforderungen auch sonstige Spezifikationen wie Anwendungsfälle, Prozessmodelle, Zustandsübergangsmodelle, Missbrauchsfälle und Risikoszenarien verwendet werden, um durchgängige Sicherheitstestszenarien ableiten zu können. Insbesondere Missbrauchsfälle und Risikoszenarien bieten durch den Perspektivwechsel, weg von den positiv formulierten Sicherheitszielen und -anforderungen hin zur Perspektive eines potenziellen Angreifers, die Möglichkeit, Schwachstellen und Angriffsszenarien gezielt und systematisch zu testen. Einen ausführlichen Überblick über die Nutzung von Missbrauchsfällen und Risikoszenarien im Kontext des Sicherheitstestens erhält man im ETSI Guide 203 251 [ETSI EG 203 251 16].

Hinweis: **Negativszenarien und Missbrauchsfälle testen**

Wenn funktionale Tests durchgeführt werden, sollte der Sicherheitstester nach Wegen suchen, Sicherheitsbarrieren gezielt zu überwinden.

Die Ziele des Sicherheitstestens während des Systemtests lassen sich wie folgt zusammenfassen:

- Durchführung von Ende-zu-Ende-Tests zur Prüfung der gesamten Funktionalität sowie der Leistungsfähigkeit des vollständigen Systems (Hardware, Software, Daten, Menschen und Verfahren) nach der Implementierung verschiedener Systemkomponenten und deren Integration in ein Komplettsystem
- Testen, dass die Sicherheitsanforderungen aus Systemperspektive richtig implementiert wurden

Die Sicherheitstests finden in der Regel bereits in einer annähernd der endgültigen Zielumgebung entsprechenden Testumgebung statt, sodass der Test auch Aspekte

der späteren Zielumgebung berücksichtigen kann. Zu diesem Zweck wird die Software normalerweise aus der Entwicklungsumgebung, in der die vorherigen Implementierungs- und Integrationsaktivitäten erfolgten, in eine spezielle Testumgebung überführt.

Hinweis: Häufige Systemtests bei iterativem Vorgehensmodell

Zu beachten ist, dass bei einigen Vorgehensmodellen, wie z.B. den iterativen Ansätzen, innerhalb kurzer Zeit neue Komponenten hinzukommen oder bestehende Komponenten verfeinert werden. Insofern können hier Systemtests viel häufiger als bei anderen, mehr sequenziellen Ansätzen vorkommen.

4.6 Die Rolle des Sicherheitstestens in der Transitionsphase

Im Verlauf der Transitionsphase wird die Software aus dem Kontext der Softwareentwicklung in ihre tatsächliche Zielumgebung überführt. Abhängig von den organisatorischen und vertraglichen Rahmenbedingungen der Softwareentwicklung, kann sich diese Phase sehr unterschiedlich gestalten.

Bei einer Auftragsentwicklung findet die Übergabe der Software an den Kunden statt. Der Kunde prüft, ob die Software seinen Erwartungen entspricht. Dies erfolgt in der Regel durch systematische Abnahmetests, die die Software in der Zielumgebung bzw. unter Bedingungen der Zielumgebung gegen eine Reihe von Abnahmekriterien prüfen. Eine Prüfung der prozessualen und technischen Voraussetzungen für den Betrieb der Software stellt zudem sicher, dass auch die Umgebung, in der die Software betrieben wird, den Anforderungen eines sicheren Betriebs entspricht.

4.6.1 Sicherheitstesten im Abnahmetest

Grundlage für Abnahmetests sind die Ende-zu-Ende-Aktivitäten der Systemnutzer.

Der Abnahmetest ist die letzte Stufe des Testens im Softwareentwicklungsprozess. Hier überzeugen sich die zukünftigen Benutzer des Systems oder deren Vertreter davon, dass das System auch in der tatsächlichen Zielumgebung die benötigten Eigenschaften besitzt und die gewünschte Funktionalität liefert.

System- und Abnahmetests sind im Wesentlichen Blackbox-Tests bzw. Stimulus-Response-Tests ohne Berücksichtigung der internen Struktur oder des Verhaltens einzelner Komponenten. Die Grundlage für die Definition von Abnahmetests stellen die Interaktionen und die Ende-zu-Ende-Aktivitäten der

Systemnutzer dar. Unabhängig davon bilden vorhergehende Komponenten- und Integrationstests jedoch eine ergänzende Bewertungsgrundlage, sodass auch für den Abnahmetest die Bewertung der internen Komponentenarchitektur und die Interaktion der Komponenten im System berücksichtigt werden können.

Das Ziel des Sicherheitstestens im Abnahmetest ist die Prüfung sicherheitsbezogener Abnahme- bzw. Akzeptanzkriterien aus der Perspektive des Nutzers bzw. eines potenziellen Angreifers. Im Mittelpunkt ihrer Definition stehen häufig die funktionalen Sicherheitsmechanismen und -prozesse, die für das jeweilige System gefordert wurden. Ein Abnahmetest setzt sich in der Regel aus den folgenden Aktivitäten zusammen:

- Installieren des Systems in seiner Betriebsumgebung
- Durchführen von Sicherheitstests entlang der Abnahmekriterien
- Entscheiden auf Basis der Testergebnisse, ob die Abnahmekriterien erfüllt sind und die Abnahme erfolgen kann

4.6.2 Definition und Pflege sicherheitsbezogener Abnahmekriterien

Im Unterschied zum Systemtest werden Abnahmetests in einer realitätsnahen bzw. in der tatsächlichen Betriebsumgebung durchgeführt und ermöglichen so eine angemessene Bewertung der Leistung, der Funktionalität und weiterer Systemeigenschaften wie beispielsweise auch der Sicherheit. Formale Grundlage für den Abnahmetest sind Abnahmekriterien, mit denen sich prüfen lässt, dass die ursprünglichen Entwicklungs- und Projektziele umgesetzt worden sind. Voraussetzung für die erfolgreiche Definition von sicherheitsbezogenen Abnahmetests ist somit das Vorhandensein von Abnahmekriterien, die sich auf die Sicherheitseigenschaften des abzunehmenden Systems beziehen.

Abnahmekriterien werden normalerweise zwischen dem Auftragnehmer und dem Auftraggeber, häufig in Form von Pflichten- und Lastenheften, ausgehandelt und sind somit fester Vertragsbestandteil einer Softwarelieferung. Sie können grundsätzlich einen eher globalen Charakter haben oder sich konkret auf einzelne Systemfunktionen beziehen. Globale Abnahmekriterien beschreiben die Maßnahmen oder Eigenschaften eines Systems, die für eine große Anzahl an Funktionen gelten sollen. Im Kontext der Software- und Systemsicherheit zählen hierzu das Vorhandensein und die Ausgestaltung sicherheitsrelevanter Maßnahmen wie Authentifizierung, Zugriffsschutz und Rechteverwaltung, die verwendeten kryptografischen Verfahren und Algorithmen inklusive der geforderten Schlüsselstärken und -längen oder auch das Vorhandensein einer durchgängigen und konsistenten Fehlerbehandlung sowie von Logging- und

Audit-Schnittstellen. Die Herausforderung im Abnahmetest besteht nun darin, das Vorhandensein der globalen Eigenschaften abschließend zu prüfen, ohne sich in Einzelfallprüfungen zu verlieren.

Beispiel: Authentisierung von Kommunikationsendpunkten und Verschlüsselung der Kommunikation

Beispiel: Authentisierung von Kommunikationsendpunkten und Verschlüsselung der Kommunikation

In den Abnahmekriterien für eine Webanwendung mit Microservice-Architektur wird festgelegt, dass sich alle Kommunikationsendpunkte authentisieren müssen und dass die Kommunikation auch zwischen den Microservices verschlüsselt erfolgen muss. Eine entsprechende Prüfung im Abnahmetest muss zeigen, dass die Webanwendung diesen Kriterien entspricht. Dies passiert in der Regel nicht dadurch, dass jede Verbindung einzeln getestet wird. Das wäre zu aufwendig und nicht wirtschaftlich. Im Abnahmetest kann exemplarisch geprüft werden, ob die Authentisierung für einzelne Verbindungen auch in der Betriebsumgebung den gewünschten Kriterien entspricht. Darüber hinaus sollte geprüft werden, dass in der Entwicklung grundlegende architektonische Muster eingehalten wurden, die für eine Endpunktauthentisierung notwendig sind, und im Komponenten- und Integrationstest die notwendigen Einzelprüfungen der Authentisierung erfolgreich absolviert worden sind. Inkrementelle Entwicklungsansätze sind hier klar im Vorteil, weil speziell globale oder invariante Abnahmekriterien für jedes Inkrement einzeln mitgetestet werden können.

In anderen Fällen werden sicherheitsspezifische Abnahmekriterien definiert, die sich auf ein konkretes Szenario oder eine konkrete Funktion beziehen. In solchen Fällen muss dann das Szenario oder die Funktion auch mit allen Ausnahmefällen geprüft werden.

Beispiel: Implementierung des »Vier-Augen-Prinzips«

Beispiel: Implementierung des »Vier-Augen-Prinzips«

In einer Software für Finanztransaktionen gibt es Funktionen wie z.B. das Anweisen von Zahlungen, die, sollte die Zahlung einen bestimmten Betrag übersteigen, die Genehmigung von zwei Personen mit einer konkreten Sicherheitsklassifizierung erfordern. Durch Tests muss geprüft werden, ob diese Funktion auch dann Bestand hat, wenn z.B. größere Geldbeträge gestückelt werden.

Unabhängig von der Art der Abnahmekriterien kann davon ausgegangen werden, dass sich im Verlauf eines Projekts Abnahmekriterien entweder ändern, die Notwendigkeit für neue Abnahmekriterien entstehen oder Abnahmekriterien wegfallen. Im Rahmen des Abnahmetests muss daher sichergestellt werden, dass veränderte bzw. neue Abnahmekriterien bei der Definition der Abnahmetests berücksichtigt und bestehende Abnahmetest entsprechend angepasst werden.

Agile und iterative Entwicklungsansätze tragen diesem Umstand Rechnung, indem Abnahmekriterien iterativ und abhängig vom Projektfortschritt für das

jeweilige Inkrement definiert werden.

Exkurs: Sicherheitsabnahmekriterien in agilen Prozessen

Agile und iterative Entwicklungsansätze definieren und detaillieren Anforderungen und Abnahmekriterien nicht zu Projektbeginn, sondern kontinuierlich unter ständiger Neubewertung der Benutzeranforderungen und des Projektstandes. Betrachtet man beispielsweise Scrum, so werden Anforderungen für jeden Iterationszyklus in User Stories heruntergebrochen und mit konkreten Abnahme- und Akzeptanzkriterien unterlegt. Letztere bilden dann die Basis für die Abnahme und den Abnahmetest der User Story im Rahmen des aktuellen Softwarerelease. Die Herausforderung für die Abnahme der Sicherheitseigenschaften besteht nun darin, die Sicherheitsanforderungen als Sicherheitsabnahmekriterien für einzelne User Stories herunterzubrechen. Das ist insofern schwierig, als User Stories in der Regel funktional definiert werden und alle nichtfunktionalen Anforderungen, wie z.B. die Sicherheitsanforderungen, nicht auf den ersten Blick in der User Story erkennbar sind. Darüber hinaus ist es für einen regulären Entwickler nahezu unmöglich, die Vielzahl der Technologien im Auge zu behalten, auf denen die Sicherheit eines Benutzerfeatures, d.h. eine User Story, aufbauen kann und muss. Bereits an dieser Stelle wird klar, dass sich das Thema Sicherheit in agilen Prozessen nur schwer delegieren lässt. Awareness und Sicherheitsexpertise wird in allen gestaltenden Teams benötigt und im Allgemeinen durch die Teammitglieder selbst repräsentiert, die entsprechend sensibilisiert, fortgebildet und ausgebildet sein müssen. Darüber hinaus finden sich in der Literatur eine Reihe von Techniken und Methoden, um die Definition von Sicherheitsabnahmekriterien zu unterstützen.

- Sicherheitsfunktionalität, die durch eigene Softwarebestandteile abgedeckt werden muss, kann in Form von sicherheitsbezogenen User Stories (Security User Stories) direkt beschrieben und mit entsprechenden Abnahmekriterien versehen werden. Eine solche User Story beschreibt dann direkt das zu realisierende Sicherheitsfeature.
- Für globale Sicherheitseigenschaften, die einen Großteil aller Features eines Systems betreffen, lassen sich global gültige Abnahmekriterien ableiten, die automatisch als Sicherheitsabnahmekriterien in alle User Stories integriert und geprüft werden müssen
- Evil Stories und Missbrauchsfälle eignen sich ergänzend dazu, Abnahmekriterien zu definieren, die konkret das Angriffspotenzial auf das zu entwickelnde System berücksichtigen.

Die Integration von Sicherheitsakzeptanzkriterien in jede User Story bedeutet, dass nichts ohne Sicherheit ausgeliefert werden kann. Die Herausforderung

besteht darin, die notwendigen Sicherheitsakzeptanzkriterien zu identifizieren und so zu skalieren, dass sie im Rahmen des Projektbudgets für jedes Softwarerelease testbar bleiben. Ergänzend muss geprüft werden, welche der Sicherheitsakzeptanzkriterien sich automatisiert testen lassen, sodass der Aufwand für die Entwicklungsteams reduziert werden kann.

4.6.3 Zusätzliche Umfänge betrieblicher Abnahmetests

Software kann nur sicher betrieben werden, wenn auch die technische und prozessuale Betriebsumgebung Sicherheit gewährleistet.

Der sichere Betrieb von Software kann weiterhin nur gewährleistet werden, wenn auch die technische und prozessuale Umgebung, in der die Software letztendlich betrieben werden soll, sicher ist. Im Rahmen des betrieblichen Abnahmetests sollten auch genau diese prozessualen und technischen Voraussetzungen für den sicheren Betrieb der Software geprüft werden. Das Ziel ist es, festzustellen, dass die sichere Integration und der sichere Betrieb der Software oder des Softwaresystems in die IT- und Prozesslandschaft des Betreibers gewährleistet werden kann. Entsprechende Tests gehören in der Regel nicht zum Abnahmetest, weil sie nicht Teil der Softwareentwicklung bzw. des Softwareentwicklungsauftrags sind, sondern als Prüfung der Infrastruktur für den Betrieb des Softwaresystems im Verantwortungsbereich des Betreibers des Softwaresystems liegen. Die Art der Tests hängt wiederum vom Organisationsmodell des Betreibers, der Art des Softwaresystems und der konkret geforderten Absicherungsmaßnahmen ab. Auf der prozessualen Ebene gehört hierzu unter anderem die Prüfung der Patch-Management- und der Incident-Response-Fähigkeiten einer Organisation in Bezug auf das zu integrierende Softwaresystem. Dabei geht es nicht nur um die Frage, ob ein Betreiber entsprechende Prozesse besitzt, sondern auch darum, wie wirksam diese auf das zu integrierende Softwaresystem angewendet werden können, d.h., ob beispielsweise für die konkrete Software Verträge zur Lieferung von Sicherheitsupdates bestehen oder ob im Fall von Sicherheitsvorfällen Unterstützung durch die Organisation erfolgt, die die Software entwickelt hat. Ähnlich sieht die Prüfung der technischen Infrastruktur für den Betrieb der Software aus. Hier muss sichergestellt werden, dass die Annahmen an die Sicherheit der Infrastruktur, die als Grundlage bei der Entwicklung der Software vorausgesetzt wurden, auch nach dem Deployment zutreffen.

Beispiel: Existenz einer Firewall mit strikter Abschirmung des Systems

Beispiel: Existenz einer Firewall mit strikter

Bei der Realisierung einer Webanwendung wurde davon ausgegangen, dass eine Firewall dafür sorgt, dass Zugriffe auf

Abschirmung des Systems

die Administrations- und Monitoring-Schnittstelle nur aus dem internen Netz der Firma erlaubt sind. Entsprechende Tests, die die Existenz und korrekte Konfiguration der Firewall prüfen, sind Voraussetzung für eine umfassende Einschätzung darüber, ob der Betrieb der Webanwendung sicher ist.

4.7 Die Rolle des Sicherheitstestens während Betrieb & Wartung

Zentrale Aufgabe des Sicherheitstestens in der Betriebs- und Wartungsphase ist es, die Sicherheit eines Softwaresystems auch nach seiner Auslieferung und Inbetriebnahme dauerhaft zu prüfen und nachzuweisen. Das Ziel besteht darin, die Schwachstellen zu finden, die sich erst im Betrieb offenbaren oder zur Betriebszeit, beispielsweise durch Wartungsarbeiten, entstanden sind. Die Hintergründe bzw. Ursachen für das Entstehen solcher Schwachstellen können sehr unterschiedlich sein. Schwachstellen im Betrieb entstehen durch unvorsichtig durchgeführte und ungenügend getestete Änderungen der Software, beispielsweise bei der Fehlerkorrektur oder der Erweiterung des Systems, können aber auch dadurch entstehen, dass in der Betriebsphase Wissen verfügbar wird, das es möglich macht, bisher unbekannte Sicherheitslücken auszunutzen. Grundsätzlich können Tests in der Wartungsphase alle in Abschnitt 2.4 beschriebenen Ziele von Sicherheitstests umfassen und dabei alle in Abschnitt 2.5.1 beschriebenen Phasen des Sicherheitstests durchlaufen. In diesem Kapitel wird speziell noch einmal auf die Themen Regressionstest, Fehlernachtest und Penetrationstest eingegangen, die speziell in der Wartungsphase eine hohe Bedeutung besitzen.

4.7.1 Sicherheitstesten als Regressions- und Fehlernachtest

Sicherheitsregressionstests bestätigen, dass das aktualisierte System die Sicherheitsanforderungen nach wie vor erfüllt.

Im Wartungstest einer Software unterscheidet man das klassische Regressionstesten und das Fehlernachtesten. Der Regressionstest testet die Kernfunktionen einer Software mit dem Ziel, zu prüfen, ob diese Kernfunktionen auch nach Änderungen der Software im spezifizierten Umfang zur Verfügung stehen. Das GTB-Glossar definiert den Regressionstest wie folgt:

Definition: Regressionstest

Testen einer bereits getesteten Komponente oder eines Systems nach einer Modifikation, um sicherzustellen, dass in nicht geänderten Bereichen durch die vorgenommenen Änderungen

keine Fehlerzustände eingebaut oder bisher maskierte Fehlerzustände freigelegt wurden. [GTB Glossar 18]

Sicherheitsregressionstesten ist eine Spezialisierung des Regressionstestens, mit dem geprüft wird, ob Änderungen an der Software die Sicherheitseigenschaften der Software beeinträchtigt haben. Während funktionale Sicherheitsregressionstests prüfen, ob definierte Sicherheitsfunktionalität nach wie vor funktioniert, kann mittels negativen Sicherheitstests bestätigt werden, dass das System Angriffen zur Überwindung der eingerichteten Sicherheitsvorkehrungen auch weiterhin erfolgreich standhält.

Beispiel: Beschränkung von Benutzerrechten

Beispiel: Beschränkung von Benutzerrechten

Auch die Administratoren einer IT-Infrastruktur sollten, allein schon aus Gründen des Datenschutzes und der Vertraulichkeit, einen beschränkten Zugriff auf Daten haben und nur Handlungen ausführen dürfen, die in ihren Benutzerrechten explizit definiert sind. Dieses ist sofort augenfällig, betrachtet man die IT eines Krankenhauses. Typischerweise darf hier ein Administrator einen neuen Mitarbeiter, z.B. einen Arzt, hinzufügen bzw. einen ausgeschiedenen Mitarbeiter deaktivieren, hat aber keinen Zugriff auf dessen Daten, d.h., er kann diese weder löschen, einsehen noch verändern. Nach einer Aktualisierung des Role-Based-Access-Control-(RBAC-)Systems in einem solchen Krankenhaus ist demzufolge in der Regression zu prüfen, ob die gewährten Rechte einerseits die typischen Aktivitäten eines Administrators zulassen (d.h. das Anlegen und Verwalten der Benutzer) und andererseits die nicht erlaubten Aktivitäten (d.h. den Zugriff auf die Patientendaten) unterbinden. Hierzu sind sowohl positive wie auch negative Ende-zu-Ende-Testszenarien zu definieren und auszuführen.

Neben anderen Bereichen gelten insbesondere Verbesserungen von Software bezüglich Benutzbarkeit oder Effizienz besonders anfällig dafür, Sicherheitseigenschaften negativ zu beeinflussen. Grundsätzlich sollten Regressionstests so weit wie möglich automatisiert werden, da sie in der Regel sehr häufig ausgeführt werden müssen.

Fehlernachtests (auch Retest genannt) sind eine spezielle Form von Wartungstests, die belegen sollen, dass eine fehlerbereinigte Software die bereits identifizierten Fehler nicht mehr enthält.

Definition: Fehlernachtest

Dynamisches Testen nach einer Fehlerkorrektur zum Zweck der Bestätigung, dass Fehlerwirkungen nicht mehr auftreten, nachdem die dafür ursächlichen Fehlerzustände korrigiert wurden. [GTB Glossar 18]

Das Ziel von sicherheitsbezogenen Regressions- und -fehlernachtests besteht darin, zu prüfen, dass durch die Wartung bzw. Fehlerbeseitigung keine neuen Schwachstellen im System entstanden sind, die bestehenden

Sicherheitsvorkehrungen auch nach einer Änderung noch wirksam sind und die bekannten Schwachstellen und Fehler tatsächlich beseitigt worden sind.

Beispiel: Überarbeitung der Benutzerauthentisierung sowie der TLS-Konfiguration

Beispiel: Überarbeitung der Benutzerauthentisierung sowie der TLS-Konfiguration

Bei einem Onlineportal wurde die Nutzerauthentisierung überarbeitet und die Sicherheitskonfiguration dahingehend verändert, dass auf ein Downgrade der SSL/TLS-Verbindung verzichtet wird. TLS 1.0, SSL 2.0 und SSL 3.0 werden nun nicht mehr unterstützt, sodass BEAST-, POODLE- und DROWN-Angriffe verhindert werden können. Benutzer sollten sich nun online am Portal anmelden können und dort sicher ihre Transaktionen tätigen können.

Im Wartungstest nach der Generalüberarbeitung der Software ist u.a. zu prüfen, dass die Transportverschlüsselung zwischen Browser und Webserver weiterhin funktioniert und sich nicht auf die Protokollversionen TLS 1.0, SSL 2.0 und SSL 3.0 zurückstufen lässt (Fehlernachtest). Darüber hinaus sollte geprüft werden, ob der Authentisierungsmechanismus funktioniert und der Zugriff auf die Nutzerdaten in der Datenbank auf den jeweiligen Nutzer beschränkt ist (Test der Kernfunktionen). Abhängig von der Art der Änderungen kann zusätzlich geprüft werden, dass keine Schwachstellen existieren, die unter Umgehung der ordnungsgemäßen Authentisierung einen Zugriff auf die Nutzerdaten erlauben.

Die Herausforderung beim Regressionstesten, insbesondere auch beim sicherheitsbezogenen Regressionstesten, besteht u.a. darin, eine möglichst optimale Menge von Testfällen zu identifizieren, die die Änderungen an der Software und ihre Auswirkungen möglichst passgenau abdecken. Hierzu können verschiedene Strategien verwendet werden. Ein kompletter Regressionstest, d.h. die Ausführung aller Regressionstestfälle, ist häufig zu aufwendig und zu teuer, speziell dann, wenn er nicht vollständig automatisiert ist. Üblich ist es insofern, die Regressionstests auf bestimmte Konfigurationen, Komponenten oder Funktionen einzuschränken. Risikobasierte Methoden der Testauswahl erlauben zusätzlich die Wahrscheinlichkeit für das Eintreten von Fehlern nach einer Änderung sowie die Kritikalität ihrer Fehlerwirkung bei der Testauswahl zu berücksichtigen. Das ist insbesondere bei sicherheitsbezogenen Regressionstests relevant, da sich Sicherheit einerseits generell gut in Form einer Risikoabschätzung beurteilen lässt (vgl. Kap. 1) und sich Sicherheitseigenschaften andererseits häufig nicht eindeutig auf einzelne Komponenten, Konfigurationen und Funktionen abbilden lassen und eine analoge Einschränkung der Regressionstests speziell für sicherheitsrelevante Tests nicht immer zielführend ist.

4.7.2 Penetrationstest

Penetrationstests dienen dazu, die Erfolgsaussichten eines

»Penetrationstests dienen dazu, die Erfolgsaussichten eines vorsätzlichen Angriffs auf

einen Informationsverbund, eines einzelnen IT-Systems oder einer Internetpräsenz abzuschätzen und daraus notwendige ergänzende Sicherheitsmaßnahmen abzuleiten beziehungsweise die Wirksamkeit von bereits umgesetzten Sicherheitsmaßnahmen zu überprüfen.« [BSI M 5.150]

vorsätzlichen Angriffs abzuschätzen.

Als Penetrationstests werden Sicherheitstests bezeichnet, die, häufig durchgeführt durch unabhängige Drittanbieter, die Sicherheit von Infrastrukturen, Netzwerken, Systemen oder einzelnen Anwendungen prüfen. Penetrationstests beziehen sich in der Regel auf ein fertiges, oftmals auch bereits in Betrieb befindliches System. Ziel ist es, unter besonderer Berücksichtigung der Motive und Perspektiven verschiedener Angreiferklassen Schwachstellen im System bzw. dem Informationsverbund zu finden. Im Unterschied zum entwicklungsbegleitenden Sicherheitstesten finden üblicherweise weder funktionale Sicherheitstests statt, noch kann die Software in den verschiedenen Stufen ihrer Entstehung geprüft werden, da in der Regel die dafür notwendigen Entwicklungsartefakte (noch) nicht zur Verfügung stehen und die Software und Systeme in bereits integrierter Form und häufig auch im ausgelieferten Zustand vorliegen. Wie stark die Einnahme der Angreiferperspektive einen Penetrationstest dominiert bzw. dominieren soll, wird üblicherweise zwischen Auftraggeber und Penetrationstester vereinbart. Während noch vor einigen Jahren auch der Entzug von potenziell verfügbaren Informationen zu Architektur und Sicherheitsmaßnahmen als förderlich für die Kreativität des Testers angesehen wurde und damit als Erfolgskriterium für den Penetrationstest galt, hat sich inzwischen etabliert, dass auch dem Penetrationstester alle verfügbaren Informationen über das zu testende System bzw. den Informationsverbund verfügbar gemacht werden sollten.

Im Unterschied zum klassischen Sicherheitstesten muss der Penetrationstest einige Besonderheiten berücksichtigen, die sich aus der Tatsache ergeben, dass die zu testenden Systeme häufig bereits im Produktivbetrieb sind. Wichtige Details und Hinweise hierzu finden sich u.a. in Kapitel 2 dieses Buches. Mit dem IS-Penetrationstest sowie dem BSI Webcheck [BSI 17a] stellt das BSI umfangreiche Hinweise zur Durchführung von Penetrationstests bereit. Das BSI führt eine Liste von zertifizierten Sicherheitsdienstleistern [BSI 17c], die den IS-Penetrationstest nach den Vorgaben des BSI durchführen können.

4.8 Was Sie in diesem Kapitel gelernt haben

In Kapitel 4 wird erläutert, was ein Softwarelebenszyklus ist und warum sich die Sicherheit von Software am besten innerhalb von Aktivitäten realisieren lässt, die

sich am Lebenszyklus der Software orientieren.

- In Abschnitt 4.1 werden die Hintergründe zu den wichtigsten sicherheitsbezogenen Aktivitäten im Rahmen eines Softwarelebenszyklus beschrieben.
- In Abschnitt 4.2 wird ausgeführt, dass bereits während der Anforderungsanalyse die Grundlage für sichere Software gelegt werden sollte und auf entsprechende Absicherungsmaßnahmen wie die Prüfung der Anforderungen aus der Sicherheitsperspektive zu achten ist. Es wird beschrieben, wie für einen gegebenen Satz von Anforderungen sicherheitsrelevante Fehler aufgedeckt und Unzulänglichkeiten identifiziert werden können.
- In Abschnitt 4.3 werden die sicherheitsbezogenen Testaktivitäten der Entwurfsphase behandelt. Es wird erläutert, wie sich Sicherheit in den Entwurfsdokumenten widerspiegelt und wie Entwurfsdokumente auf Schwachstellen und sicherheitskritische Fehler hin analysiert werden können.
- In Abschnitt 4.4 werden die implementierungsnahen Sicherheitstestaktivitäten behandelt. Dabei wird insbesondere auf die Bedeutung des Sicherheitstestens im Rahmen der Komponententests eingegangen und genauer erläutert, wie sicherheitsbezogene Codeanalysen durchgeführt, Komponententests implementiert und die daraus resultierenden Testergebnisse ausgewertet werden können.
- In Abschnitt 4.5 werden die notwendigen Sicherheitstestaktivitäten der Integrationsphase beschrieben. Ausgehend vom Komponentenintegrationstest bis hin zum Systemtest wird das notwendige Wissen vermittelt, um sicherheitsbezogene Komponentenintegrationstests wie auch Ende-zu-Ende-Testszenarien spezifizieren, umsetzen und die jeweiligen Ergebnisse interpretieren zu können.
- Abschnitt 4.6 beschreibt das Sicherheitstesten in der Transitionsphase. Grundlage für diese Testphase sind wohldefinierte Abnahmekriterien für die Sicherheitsaspekte einer Software. Es wird gezeigt, wie Abnahmekriterien für die Sicherheitsaspekte eines Softwaresystems erstellt und geprüft werden können. Darüber hinaus wird vermittelt, dass neben dem Softwaresystem auch immer die Betriebsumgebung sowie die Rahmenbedingungen für den Betrieb der Software dahingehend zu prüfen sind, ob die ursprünglichen Sicherheitsannahmen, die als Grundlage der Softwareerstellung vorausgesetzt wurden, noch gelten.
- Abschnitt 4.7 schließt den Lebenszyklus mit der Beschreibung von Sicherheitstestaktivitäten in der Wartungsphase eines Softwaresystems ab. Im Verlauf dieses Kapitels wird erläutert, was eine durchgängige Vorgehensweise für den Sicherheitsregressionstest ist und wie

Sicherheitsregressionstests systematisch spezifiziert und ausgewählt werden können. Darüber hinaus wird das Konzept des Penetrationstests eingeführt und beschrieben, wie sich dieser vom entwicklungsbegleitenden Sicherheitstest unterscheidet.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z