Fachhochschule Kiel
University of Applied Sciences Kiel
Fachbereich Informatik und Elektrotechnik

Master Thesis
im Studiengang Information Technology

Evaluierung und Implementierung von *DevOps*-Strategien zur Erhöhung der Sicherheit von Webanwendungen für Startups

Öffentliche Version

Erst-Gutachter: Prof. Dr. rer. nat. Nils Gruschka
Zweit-Gutachter Prof. Dr. Robert Manzke
Vorgelegt von: Timo Pagel (917019)
Datum: 15.07.2016

Änderungen

Datum	Änderung
03.08.2016	Referenzierung der Tabelle I.2 berichtigen.
03.08.2016	Erläuterung, dass OWASP Top Ten A9-Werkzeuge ggf. zu viele Informationen an Anbieter weitergeben.
14.06.2017	Optimierung von Zitaten

Lizenz

Zusammenfassung

Insbesondere in Startups, bei welchen Ressourcen häufig knapp sind, ist die Priorisierung der Implementierung der *DevOps*-Strategien zur Erhöhung der Informationssicherheit von Webanwendungen von hoher Bedeutung. Im Laufe dieser Arbeit ist ein generisches *DevOps*-Sicherheits-Reifegradmodell entwickelt, evaluiert und in einem Startup unter Einsatz von *Open Source* Software teilweise implementiert worden.

Grundlage der Entwicklung sind die *DevOps*-Dimensionen Technologie, Test und Kultur. Neue Ansätze, wie beispielsweise eine Team-Sicherheitsprüfung, sind in das Modell eingegangen.

Die unteren beiden Ebenen des generischen *DevOps*-Sicherheits-Reifegradmodells sind für Startups und Organisationen einfach zu erreichen, während sie für die dritte und vierte Ebene erweiterte Ressourcen, insbesondere Zeit zur Implementierung, benötigen.

Angreifer sind intelligent, mit neuen Technologien ausgerüstet und zielstrebig. Startups handeln unter Anleitung des generischen *DevOps*-Sicherheits-Reifegradmodells ebenso.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Abkürzungsverzeichnis

Ajax Asynchronous javaScript and XML

AST Abstract Syntax Tree

Aufs Advanced multi layered unification filesystem

BSI Bundesamt für Sicherheit in der Informationstechnik

CI Continuous Integration

CVE Common Vulnerabilities and Exposures

DAST Dynamic Application Security Testing

DevOps Kombination aus Entwickler (Englisch developer) und Betrieb (Englisch operations)

DOM Document-Object-Model

GDOSR Generisches DevOps-Sicherheits-Reifegradmodell

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IP Implementierungspunkt

IPC Inter-Process Communication

IT Informations Technologie

JSON JavaScript Object Notation

Knight Knight Capital Americas LLC

NoSQL Not only SQL

NVD National Vulnerability Database

OWASP Open Web Application Security Project

REST Representational State Transfer

RLP Das Retail Liqudity Programm der New York Stock Exchange

SAST Static Application Security Testing

SDOMM Security DevOps Maturity Model

SMTP Simple Mail Transfer Protocol

SQL Structured Query Language

SSH Secure Shell

SVN Apache™ Subversion®

TCP Transmission Control Protocol

TLS Transport Layer Security

TSP Team-Sicherheitsprüfungen

UDP User Datagram Protocol

URL Uniform Resource Locator

VCS Version Control System

XSS Cross-Site Scripting

Zap OWASP Zed Attack Prox

Kapitel 1

Einleitung

1.1 Problemstellung und Ziel der Arbeit

Der Begriff *DevOps* setzt sich zusammen aus Entwicklung (Englisch *development*) und IT-Betrieb (Englisch *operations*). *DevOps* ist ein kultureller Ansatz, durch welchen Konflikte zwischen Entwicklung und IT-Betrieb abgebaut werden sollen, um schneller Software auszuliefern. Hier wird häufig das Beispiel einer imaginären Mauer zwischen Entwicklung und IT-Betrieb herangezogen. Die Entwicklung entwickelt eine Anwendung, „wirft“ die Anwendung über die Mauer und kümmert sich nicht mehr um den Betrieb auf der Produktionsumgebung.
DevOps arbeiten i.d.R. nach agilen Methoden, welche Informationssicherheit nicht explizit einschließen. Deshalb wird eine Strategie zur kulturellen Integration von Informationssicherheit in die Entwicklung und den Betrieb für Startups benötigt. Weiter wird eine Strategie benötigt, die das Teilen von Wissen im Bereich Informationssicherheit fördert.
Die Nutzung von *DevOps*-Strategien wie kontinuierliche Verteilung oder Virtualisierung mittels Containern birgt die Gefahr, dass Informationssicherheit aufgrund von Unwissenheit im Umgang mit den neuen Technologien vernachlässigt wird. Nathan McCauley, *Director of Security* von *Docker* sagte 2015 auf der *DockerCon* [46]

> Security is the thread that needs to cut across all of the Docker projects.

Dies zeigt die hohe Anforderung, Informationssicherheit bei der Einführung neuer *DevOps*-Technologien zu beachten und die Systeme zu härten. Härten ist in dieser Arbeit definiert als ein Prozess, eine Methode, ein Werkzeug oder eine Kombination aus diesen, um die Informationssicherheit einer Software oder eines Systems zu erhöhen [177].
Auf der anderen Seite stellt hoch frequentierte Verteilung von Webanwendungen hohe Anforderungen an die Informationssicherheit. Traditionell wird eine Sicherheitsprüfung der Webanwendung einmal im Jahr durchgeführt, was bei hoch frequentierter Verteilung nicht ausreichend ist. Spätestens bei der Verteilung ist schnelle Rückmeldung zum Sicherheitsniveau notwendig. Schnelle Rückmeldungen können durch automatisierte Sicherheitstests gegeben werden, weshalb Strategien, Methoden, Einsatzorte und Werkzeuge evaluiert werden müssen.
Fehlendes Wissen im Bereich Informationssicherheit, sowohl bei Anwendungsentwicklung als auch beim

IT-Betrieb, können zur Vernachlässigung des Themas Informationssicherheit führen oder auch zu falscher Priorisierung bei der Einführung. Insbesondere bei Startups, bei welchen Ressourcen häufig knapp sind, ist die richtige Reihenfolge der Implementierung von Strategien und die Härtung von Systemen zur Erhöhung der Informationssicherheit der Webanwendungen von hoher Bedeutung. Auch wenn Startups unterschiedliche Schwerpunkte an Informationssicherheit stellen werden, so wird die Hypothese aufgestellt, dass ein Instrument entwickelt werden kann, um Startups bei der Priorisierung der *DevOps*-Strategien zur Erhöhung der Informationssicherheit ihrer Webanwendung zu unterstützen.

1.2 Abgrenzung von behandelten Themen

Der Begriff „Startup“ kommt aus dem Englischen und Bedeutet „neu gegründet“, in wirtschaftlichem Kontext ein neu gegründetes, junges Unternehmen. Startups verstärken den Prozess des Verschmelzens der klassischen Industrie mit der Informations-Technologie [85]. Charakteristisch ist ein geringes Startkapital, wobei häufig die Kapitalbasis durch Erhalt von Kapital durch Dritte gestärkt wird [12]. Entsprechend wird im Folgenden nur auf kostenfreie Methoden und Software eingegangen. Bei *Open Source* Software geben Mitwirkende Verbesserungen einer Software an den Projektleiter, welcher im Gegenzug die Software kostenfrei anbietet [160, S. 21]. Eine Verbesserung kann z.B. die Übersetzung einer Anleitung oder die Weiterentwicklung der Anwendung sein.
Im Rahmen dieser Arbeit wird Sicherheit als Informationssicherheit aufgefasst, also die Sicherung von Integrität, Verfügbarkeit und Vertraulichkeit einer Webanwendung. Auch wenn Hochverfügbarkeit die Verfügbarkeit eines Systems erhöht, wird dies aus Zeitgründen nicht behandelt. Die Provisionierung von Systemen wird aus Zeitgründen nur kurz angeschnitten.
Es wird nicht auf Informationssicherheit im Allgemeinen eingegangen, sondern nur auf die Berührungspunkte mit *DevOps*. Entsprechend wird z.B. nicht auf die Härtung der Infrastruktur im Allgemeinen, sondern nur auf die *DevOps* spezifischen Strategien und Technologien eingegangen. Weiterhin wird nicht erläutert, wie eine Webanwendung im Allgemeinen sicher zu erstellen ist.
Weiter wird nicht generell für ein Unternehmen der Hochsicherheit entwickelt, entsprechend können Systeme nicht vollständig abgesichert werden. Also werden Systeme und Anwendungen im Rahmen dieser Arbeit gehärtet, mit dem Wissen um ggf. weiterhin bestehende Risiken.
Da nach W3Techs [54] 67,2% aller Webserver über Unix und davon 50% von Linux betrieben werden, wird im Folgenden nur auf die Server-Version des *Open Source* Betriebssystems *Ubuntu* [52] von Canonical Ltd. eingegangen.

1.3 Kapitelgliederung der Arbeit

Diese Arbeit gliedert sich in acht Kapitel. In diesem Kapitel steht die Problemstellung und das Ziel der Arbeit im Vordergrund.
Im nächsten Kapitel werden Grundlagen und verwandte Arbeiten vorgestellt. Insbesondere wird der Begriff *DevOps* eingeführt und eine Umfrage zu gemeinsamen Zielen in IT-Bereichen durchgeführt.
Anschließend werden im dritten Kapitel kulturelle und organisatorische Maßnahmen vorgestellt. Dabei wird aufgezeigt, wie Sicherheit in agile Vorgehensmodelle integriert werden sollte und wie durch

Team-Sicherheitsprüfungen Austausch im Team im Bezug auf Sicherheit gefördert und die Sicherheit der Webanwendung nachhaltig erhöht werden kann.
Kapitel vier beschäftigt sich mit *DevOps*-Technologien. Insbesondere mit der Härtung der Infrastruktur und des Erzeugungs- und Verteilungsprozesses. Weiter wird dargestellt, wie sicherheitsrelevante Informationen aus Systemen und Anwendungen gesammelt und visualisiert werden können.
Um die eingeführten Sicherheitsmaßnahmen zu verifizieren, werden automatisierbare Sicherheitstests in Kapitel fünf vorgestellt.
Im sechsten Kapitel wird aus den vorher vorgestellten Methoden und Technologien ein generisches Sicherheits-Reifegradmodell entwickelt, mit Experten evaluiert und anschließend in Kapitel sieben implementiert.
Den Abschluss dieser Arbeit stellt das Fazit in Kapitel acht.

Kapitel 2

Grundlagen und verwandte Arbeiten

In diesem Kapitel wird zunächst der Begriff *DevOps* eingeführt. Danach sind *DevOps* anhand eines Negativ-Beispiels vorgestellt und es wird analysiert, wie durch *DevOps* eine Verbesserung erlangt werden kann. Danach wird die generelle Bedeutung von *DevOps* für Organisationen und der Bezug zu Informationssicherheit erläutert. Tests spielen bei der Erhöhung der Informationssicherheit durch *DevOps*-Strategien eine große Rolle, deshalb werden die Grundlagen ausführlich erläutert. Anschließend wird eine durchgeführte Umfrage zur Identifizierung von gemeinsamen Zielen von *DevOps* und Sicherheit vorgestellt. Zum Abschluss wird die Entwicklung von Reifegradmodellen eingegangen.

2.1 Einführung des Begriffs *DevOps*

Nach Bass et al. [66, S. 2] beschreibt *DevOps* einen Satz von Strategien zur Reduzierung der Zeit zwischen dem Einchecken einer Änderung und der Bereitstellung auf dem Produktionssystem unter Sicherung der Qualität.
Historisch sind Entwicklung und IT-Betrieb getrennt durch ihren Aufgabenbereich. Dabei entstehen jedoch Barrieren. Beispielsweise benötigen Entwickler bei der Einführung eines Buttons zur Aufnahme von Audio die Bibliothek *ffmpeg*, welches Audio-Dateien umwandeln kann. Hier muss i.d.R. eine Anfrage beim Administrator erfolgen, welcher die Bibliothek auf den System-Umgebungen installiert. Dieser Prozess kann dabei Zeit in Anspruch nehmen.
Da keine einheitliche Definition von *DevOps* existiert, haben Willis und Edwards 2010 den Begriff *CAMS* geprägt, welcher für *Culture*, *Automation*, *Measurement* und *Sharing* steht [249]. *CAMS* versucht die Werte von *DevOps* zusammenzufassen. Im Detail bedeuteten diese folgendes:

- *Culture*: Software ist entwickelt von und für Menschen.
- *Automation*: Durch Automation können schnelle Rückmeldungen erfolgen.
- *Measurement*: Einsatz von Maßnahmen zur Erhöhung der Qualität.
- *Sharing*: Nutzen einer Kultur zum Teilen von Ideen, Prozessen und Werkzeugen.

Durch *DevOps* werden „Silos“, also Abteilungen wie Softwareentwicklung, Qualitäts-Management und System-Management aufgelöst und Wissen in Teams aus unterschiedlichen Disziplinen gebündelt um einen kontinuierlichem Verbesserungsprozess zu erzeugen [127].

Zwischen agilen Vorgehensmodellen wie *Scrum* und *DevOps* gibt es Zusammenhänge. Die größte Gemeinsamkeit laut Prof. Dr. Hasselbring der Christian-Albrechts-Universität zu Kiel (siehe Anhang B.3) kann unter *fail fast* zusammengefasst werden. Agile Methoden sowie *DevOps*-Strategien setzen darauf, dass Fehler passieren können, aber möglichst zeitnah Rückmeldungen mit Informationen zur Korrektur erfolgen.
Seit 2015 existiert die Abwandlung *BizDevOps*, welche zusätzlich Geschäftsverständnis zu *DevOps*-Strategien hinzufügt [123].

2.2 Fallbeispiel „Knightmare on Wall Street“

Am ersten August 2012 setzte die *Knight Capital Americas LLC* (nachfolgend *Knight*) ein neue Funktion in Produktion und erlitt innerhalb von 45 Minuten einem Schaden von ca. 460 Millionen Dollar. [219] Der Vorfall ist seither als „Knightmare on Wall Street“ in der Presse zu finden [232, 196, 193].

2.2.1 Hintergrund

Knight war ein amerikanisches Unternehmen, welches an der *New York Stock Exchange* handelte. Der Handel erfolgte durch Algorithmen, bei welchen Wertpapiere automatisch in hoher Frequenz gekauft und verkauft werden. [196, 193]
Die *Securities and Exchange Commission* kündigte am 3. Juli 2012 das *Retail Liqudity Programm* an (kurz *RLP*), welches am 1. August umgesetzt wurde. Dadurch wurden die Handelsspannen, sogenannte *Spreads*, zwischen Kauf- und Verkaufskursen verengt und erlaubten Händlern wie *Knight*, Wertpapiere statt mit einem *Spread* von 0,01 Dollar, nun mit einem *Spread* von 0,001 Dollar zu kaufen bzw. zu verkaufen. Der Handel sollte dadurch insgesamt liquider werden. [218]
Als Vorbereitung aktualisierte *Knight* seinen hochfrequenten algorithmischen Order-Router SMARS, welcher automatisch Kauf- und Verkaufsaufträge (Orders) am Markt platziert. Eine der Hauptfunktionen von SMARS ist die Aufteilung von großen Orders (Vater-Orders) in mehrere kleine Orders (Kind-Orders), damit der Markt die tatsächliche Ordergrößen nicht erkennt und Kurse durch die tatsächliche Ordergröße nicht beeinflusst werden. Gleichzeitig finden sich durch die kleinen Ordergrößen leichter Käufer bzw. Verkäufer. Dabei gilt, je größer eine Order ist, desto mehr kleine Einzelorders werden geschaffen. [219]
Die Aktualisierung von SMARS zur Vorbereitung auf das *RLP* sollte unbenutzten Quellcode entfernen, welcher als *Power Peg* bezeichnet wurde. Dieser kann über einen *Feature Toggle,* in diesem Fall über einen boolschen Parameter, aktiviert und deaktiviert werden. *Power Peg* prüft die gehandelten Kind-Orders und stoppt die Vater-Order, sobald die Gesamtstückzahl des jeweiligen Wertpapiers verkauft bzw. gekauft ist. Die Order-Zählprüfung wurde 2005 in einen früher ausgeführten Teil von SMARS verschoben, *Power Peg* jedoch seitdem nicht genutzt oder entfernt. Knight hat den *Power Peg*-Quellcode durch den neuen Quellcode für *RLP* ersetzt. [219]

2.2.2 Ablauf

Seit dem 27. Juli 2012 hat *Knight* die neue *RLP*-Funktionalität auf sieben der acht Produktionsserver manuell durch eine Person Stück für Stück verteilt. Ein Server wurde dabei vergessen. [219]

Am 1. August, als *RLP* eingeführt wurde, wurden Orders mit dem *Feature Toggle* an die acht Server übergeben. Die sieben Server mit dem *RLP*-Quellcode haben erfolgreich die *RLP*-Funktion aufgerufen, der achte Server dagegen den alten *Power Peg*-Quellcode. Der *Power Peg*-Quellcode hat wie ursprünglich geplant funktioniert, jedoch ohne die Order-Zählprüfung. Entsprechend wurden Kauf- und Verkaufsaufträge kontinuierlich hochfrequent platziert, jedoch wurde vom Algorithmus nicht registriert, sobald die Gesamtstückzahl erreicht wurde. Dies kann als Endlos-Schleife für den Kauf und Verkauf von Wertpapiern angesehen werden. Um 9:30 Uhr wurde der Markt geöffnet und erhöhtes Orderaufkommen am Markt verzeichnet. Es wurde auf Produktionssystemen zur Laufzeit Fehleranalyse betrieben. Dabei wurde u.a. der *RLP*-Quellcode von den sieben Servern mit eigentlich korrektem *RLP* entfernt, so dass diese wie der achte Server ohne Prüf-Funktion Orders erstellt haben. Für 212 Vater-Orders wurden Millionen Kind-Orders erstellt. [219]

2.2.3 Fehlende Schutzmaßnahmen

Vergleichende Statistiken für ein- und ausgehende Orders waren nicht definiert. Auch gab keine Grenzen für gesammelte Statistiken oder eine Visualisierung derselben. *Knight* nutzte ein Überwachungs-System, welches am ersten August eine Anomalie erkannt hat und gemeldet hat. Die Meldung der Anomalie wurde ignoriert. Des Weiteren war das Überwachungs-System nicht in der Lage, bei Anomalien den Aktienhandel zu stoppen. Knight hatte keine Entwicklungs-, Verteilung- oder Incident-Response Prozesse definiert. Eine Dokumentation für einen „Not-Aus“ fehlte. [219]

2.2.4 Fazit

Das Management von *Knight* hat keine *DevOps*-Strategien eingeführt, welche die Risiken von Fehlern reduziert hätten. Dabei wäre wahrscheinlich durch eine bessere Kultur kein erhöhter Zeitdruck bei System-Administratoren und Entwicklern entstanden. Durch Automatisierung wäre ein definierter Erzeugungs- und Verteilungs-Prozess vorhanden gewesen, also wären alle Server mit der korrekten RLP-Funktion ausgestattet gewesen. Durch Maßnahmen zur Erhöhung der Qualität, beispielsweise Modultests, wäre die Software weniger fehleranfällig gewesen. Durch Teilung von Wissen hätten System-Administratoren bei der Fehleranalyse die Software nicht auf einen vorherigen Stand, ohne Rücksetzung der Konfiguration mit dem *Feature Toggle*, zurück gesetzt. Weiterhin wäre die Meldung der Anomalie durch Teilung von Wissen besser ausgewertet worden und eine früher Analyse wäre möglich gewesen.

2.3 Informationssicherheit und *DevOps*

Nach dem „2015 State of DevOps Report“ [201, S. 4] mit 4976 befragten Organisationen, können Organisationen durch *DevOps*-Strategien 30 mal schneller eine Verteilung ihrer Software durchführen und benötigen dabei 200 mal weniger Zeit als Konkurrenten ohne *DevOps*-Strategien. Gemessen wurde dabei die Zeit zwischen dem Einchecken des Quellcodes in das Versionkontrollsystem bis der Quellcode in der Produktivumgebung aktiv ist. Dabei treten 60 mal weniger Fehler, wie beispielsweise Schwachstellen, auf. Wenn ein Fehler auftritt, ist die mittlere Reparaturzeit 168 mal schneller, was ein Hinweis auf die Stabilität der eingesetzten Techniken gibt.

Der „2016 State of DevOps Report“ [202] geht zum ersten Mal auf das Qualitätskriterium Sicherheit ein. Er zeigt auf, dass hoch performante Organisationen, die *DevOps*-Strategien verwenden, sich 50 Prozent weniger Zeit als niedrig performante Organisationen mit der Korrektur von Sicherheitsvorfällen beschäftigen. Der Report empfiehlt, Sicherheit in den gesamten Entwicklungsprozess Software-Entwicklungszyklus zu integrieren. Dies umfasst unter anderem:

- Anforderungsanalysen und Architekturentwürfe,
- Priorisierung von bekannten Schwachstellen,
- Entscheidungen eine Anwendung freizugeben, obwohl Schwachstellen enthalten sind.

Eine organisatorische *DevOps*-Strategie ist das Teilen von Informationen, also auch von Risiken. Jeder, egal ob Entwickler oder System-Adminstrator, ist z.B. zuständig für unterschiedliche Qualitätskriterien wie Integrität, Verfügbarkeit, Vertraulichkeit, Zuverlässigkeit und Benutzbarkeit. Entsprechend sind Entwickler auch für funktionsfähigen und optimierten Quellcode in einer Produktionsumgebung verantwortlich. Durch Verteilung der Verantwortung auf mehrere Köpfe werden Risiken reduziert, da unterschiedliche Erfahrungen zu höherer Qualität führen [201, S. 23].
Sicherheitsprüfungen können im klassischem Wasserfallmodell der Test-Phase zugeordnet werden. Das Testen wird jedoch zu spät in der Entwicklung angewendet. Entwickler benötigen direkte Rückmeldung beim Einchecken von Quellcode in das Versionskontrollsystem, beispielsweise bei der Implementierung einer neuen Funktion. Beinhaltet diese eine *Cross-Site Scripting* (kurz *XSS*) Schwachstelle, so sollte möglichst schnell eine Rückmeldung mit Informationen zur Schwachstelle erfolgen. Auch Kostenreduktion durch frühe Rückmeldung spielt hier eine große Rolle. Eine Schwachstelle verursacht laut AT&T 6,5 mal mehr Kosten wenn diese in Produktion statt in der Qualitätskontrolle gefunden wird [60].

2.4 Sicherheitstests

Das *National Institute of Standards and Technology* (kurz *NIST*) [235, S. 2ff] zeigt, dass aufgrund von ungenügenden Sicherheitstest hohe Kosten für Organisationen entstehen. Entsprechend sollten Testfälle definiert und ausgeführt werden.
Nachdem ein Testfall ausgeführt wurde, wird das beobachtete Verhalten mit dem vorgesehenem Verhalten verglichen, was in einem Testurteil resultiert [150, S. 368].
Tests können über die Dimensionen „Ziel“, „Bereich“ und „Zugänglichkeit“ definiert werden [111, 255], welche in Abb. 2.1 dargestellt sind.
Funktionale Tests verifizieren und validieren die Funktionalität während nicht-funktionale Tests Qualitätseigenschaften wie Sicherheit, Stabilität und Performance verifizieren und validieren. Der Bereich beschreibt die Granularität des Tests. Modul-Tests (Englisch *unit tests*) verifizieren die kleinst mögliche Komponente, z.B. eine Klasse bei objektorientierter Entwicklung. Integrations-Tests verifizieren Unter-Einheiten eines Systems, wobei dieses noch nicht komplett ist. System-Tests validieren das gesamte System, wobei Akzeptanztests und Regressionstests eine Untergruppe bilden [111, S. 4]. Akzeptanztests dienen Benutzern, Kunden oder anderen Entitäten die spezifizierten Anforderungen an ein System oder eine Komponente zu prüfen [203]. Regressionstests validieren die Funktionalität nach einer Änderung, so

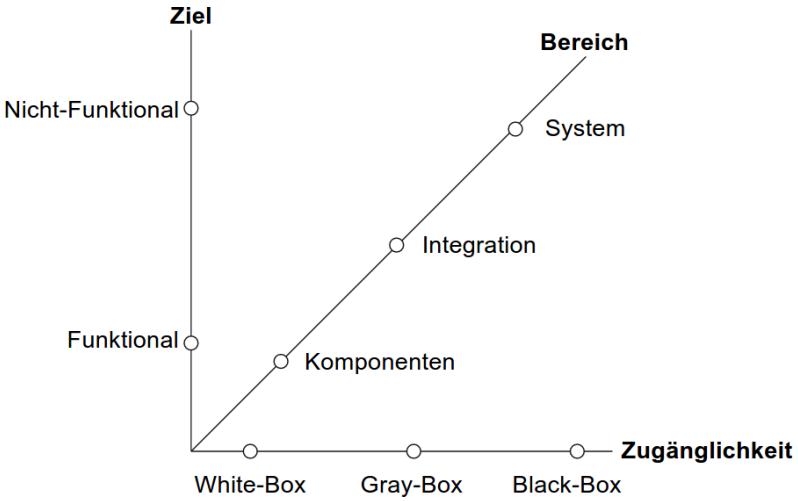


Abbildung 2.1: Diagramm mit den Test-Dimensionen Ziel, Bereich und Zugänglichkeit in Anlehnung an [111, S. 3]

dass keine unerwarteten Effekte auftreten und die Komponenten weiterhin den Anforderungen entsprechen [203].

Statische Quellcode-Analyse wurde ursprünglich in FORTRAN intraprozedural, also auf eine einzelne Prozedur in Isolation, angewendet [146]. Die Schwierigkeit liegt dabei in Bedingungen, da eine Prozedur i.d.R. mehrere Programmpfade aufweist, jedoch nicht alle Programmpfade bei Ausführung abgedeckt werden [157]. Durch ein abstraktes Modell der Webanwendung werden Informationen komprimiert, können aber auch verloren gehen.

Eine statische Sicherheitsanalyse (Englisch *static application security testing*, kurz *SAST*) untersucht ein Entwicklungsartefakt auf potenzielle Sicherheitslücken. Dabei wird die Anwendung nicht ausgeführt [74, S. 4-2]. Es besteht beim Test Zugriff auf den gesamten Quellcode und ggf. auf das nicht ausgeführte Compilat. Methoden sind z.B. manuelle Reviews oder automatische Quellcodeanalysen. Da hier auf interne Informationen wie Quellcode zurückgegriffen wird, nennt sich dieses Verfahren *White Box*-Test.

Die dynamische Sicherheitsanalyse (Englisch *dynamic application security testing*, kurz *DAST*) untersucht eine Anwendung bei der Ausführung als *Black Box* und verbindet sich typischerweise von außerhalb. Es wird eine Laufzeitumgebung benötigt.

Während eine statische Analyse z.B. den Datenfluss auf Grundlage des Quellcodes testet, wird bei der dynamischen Analyse die Anwendung von außen durchsucht und alle möglichen Eingaben erkannt und getestet.

Während *DAST* vor und zwischen Komponenten wie einer Firewall oder einem Einbruchserkennungs-System eingesetzt werden kann, analysiert *SAST* die Anwendung selbst [137].

SAST kann beispielsweise genutzte veraltete Bibliotheken aufdecken oder im Quellcode Formatierungsverletzungen auffinden. Statische Analysen zur Untersuchung von Webanwendungen bieten den Vorteil eines konsistenten Tests, welcher nicht durch einen Entwickler beeinflusst wird.

Herausforderungen von *DAST* sind die komplette Abdeckung aller Pfade einer Anwendung sowie die anschließende Auffindung von Angriffsvektoren in den Pfaden. Bei *SAST* besteht die Herausforderung in der Priorisierung der durchzuführenden Tests.

Chess und West sprechen in [90, S. 22] statischer Analyse den Vorteil eines unabhängigen Tests zu, bei

welchem nicht wie bei dynamischen Prüfungen gegebenfalls nur „interessante“ Teile geprüft werden. Da bei statischer Analyse besonders viele falsch Positive gemeldeten werden, kann die statische Analyse auch auf wichtige Teile des Quellcodes eingeschränkt werden. Bei statischer Analyse wird die Ursache eines Defekts aufgedeckt und nicht nur ein Symptom davon. Dadurch können Entwickler besser den Defekt beheben. Mittels statischer Analyse können Defekte während der Entwicklung aufgedeckt werden. Wird eine neue Schwachstelle bekannt, kann statische Analyse effizient den Quellcode eines großen Projekts testen. [90, S. 22]

Ein statisches Werkzeug hat die Eigenschaft *sound*, wenn es für eine bestimmte Annahme keine falsch Negativen[1] erstellt. Nachteilig ist, dass die Reduzierung von falsch Negativen häufig zu Lasten einer erhöhten Anzahl von falsch Positiven[2] geht. Sicherheitsexperten haben erfahren, dass eine hohe Anzahl von falsch Positiven zu 100 Prozent falsch Negativen führt, da Personen aufhören Meldungen zu beachten. Ein Werkzeug ist *unsound*, wenn es falsch Positive minimiert und dafür falsch Negative auftreten. [89] Während aus akademischer Sicht ein Werkzeug *sound* sein sollte, ist die Minimierung von falsch Positiven für den praktischen Einsatz von hoher Bedeutung [90, S. 30].

Der Test-Prozess besteht aus den Phasen Planung, Entwurf, Implementierung, Ausführung und Evaluierung [67]. Planung ist die Aktivität der Erstellung oder Aktualisierung eines Test-Plans [95]. Der Test-Plan enthält dabei das Ziel, die Aktivitäten, die Testabdeckung, die Stop-Kriterien und den Bereich. Während der Entwurfs-Phase wird der Testplan in konkrete Bedingungen und abstrakte Tests umgewandelt. Bei der Implementierung werden die abstrakten Tests ausführbar. Dies beinhaltet auch die Vorbereitung von Test-Daten, Protokollierung oder Skripts zur automatisierten Ausführung von Tests. In der Ausführung werden die Tests ausgeführt und relevante Informationen protokolliert und überwacht. Bei der Evaluierung werden Stop-Kriterien geprüft und die Ergebnisse in Form eines Berichts zusammengefasst. [111]

Sicherheitsanforderungen können positiv oder negativ ausgedrückt werden. Bei positiven Anforderungen werden explizit die vorgesehen Sicherheitsmaßnahmen formuliert, während bei negativen Anforderungen formuliert wird, was das System nicht tun soll [172]. Die Anforderung „Das Benutzerkonto wird nach drei fehlgeschlagenen Anmeldeversuchen gesperrt“ ist positiv [111, S. 3], während die Anforderung „Die Anwendung soll nicht von nicht authentifizierten Benutzern kompromittiert oder missbraucht werden können“ negativ ist [172]. Die Beispiele zeigen, dass negative Anforderungen und Tests schwerer zu formulieren und zu implementieren sind als positive [58].

Schwachstellen-Tests identifizieren unbeabsichtigte Schwachstellen im System. Dabei wird versucht, sich in die Lage eines Angreifers hineinzuversetzen, ein System anzugreifen und Schwachstellen auszunutzen [106]. Negatives Testen ist entsprechend schwer zu automatisieren [58].

Wird ein Risiko durch einen Sicherheitsbeauftragten gefunden, so erfolgt eine Meldung an einen Entwickler oder System-Administrator. Sicherheitsbeauftragten fehlt häufig das Wissen, Risiken zu beheben. Hinzu kommt, dass die Behebung von Risiken häufig nicht im Aufgabengebiet eines Sicherheitsbeauftragten liegt. Die Illustration im Anhang J.1 auf Seite 158 verdeutlicht diesen Umstand scherzhaft. [98]

Entsprechend müssen Entwickler und System-Administratoren in der Lage sein, Risiken zu verstehen und zu beheben.

[1] Es sind also nicht gemeldete Defekte vorhanden.

[2] Es sind also Defekte gemeldet, die keine Defekte sind.

Zur Durchführung von Penetrations-Tests kann das „Durchführungskonzept für Penetrationstests“ vom Bundesamt für Sicherheit in der Informationstechnik (kurz BSI) [10] zur Orientierung genutzt werden.
Zur Durchführung von Blackbox-Schwachstellen-Tests für Webanwendungen kann sich z.B. am *OWASP Testing Guide v4* [172] orientiert werden. *OWASP* steht für *Open Web Application Security Project*. Der *OWASP Testing Guide v4* erläutert anhand von Beispielen, wie auf bestimmte Schwachstellen geprüft werden sollte. Für *Whitebox*-Prüfungen von Webanwendungen eignet sich der *OWASP Code Review Guide 2.0* [228]. Hier wird erläutert, wie Quellcode-Reviews durchgeführt werden sollten und auf welche Schwachstellen geprüft werden sollte.

2.5 Umfrage zu gemeinsamen Zielen in IT-Bereichen

Von Timo Pagel wurde zwischen dem 8. März und dem 12. Juni 2016 eine Experten-Umfrage zur Identifizierung von gemeinsamen Zielen und Konfliktpunkten der Bereiche Anwendungs-Entwicklung, IT-Betrieb und IT-Sicherheit durchgeführt. Der Fragebogen ist mit der *Open Source*-Version von *LimeSurvey* [21] erstellt und der digitale Fragebogen war unter http://survey.timo-pagel.de einsehbar. *LimeSurvey* bietet eine Export-Funktion an, so dass der Fragebogen in ein schriftliches Format exportiert werden kann.
Der Fragebogen (siehe Anhang C.1 auf Seite 119) beginnt mit einer Einleitung, in welcher wie von Pilshofer in [195] empfohlen, folgende Punkte dargestellt sind: Ziel der Untersuchung, Kurzbeschreibung vom Fragenden, Betonung der Wichtigkeit jeder Antwort, den Hinweis alle Fragen ehrlich auszufüllen und die Eingrenzung der Teilnehmer.
Antworten des Fragebogens werden i.d.R. über eine fünf-Punkte-Likert-Skala von „gar nicht / sehr gering“ mit Wertigkeit 1 bis „sehr hoch“ mit Wertigkeit 5 erfasst. Zusätzlich kann angegeben werden, dass der genannte Begriff unbekannt ist.
Der Fragebogen besteht aus acht Frage-Gruppen. Hierbei werden zunächst arbeitsbezogene und anschließend demographische Fragen gestellt. Der arbeitsbezogene Teil enthält Fragen nach Modellen und Konzepten, Eigenschaften, Tests sowie Techniken aus der täglichen Arbeit.
Zu bewertende Modelle und Konzepte sind das „Fail Fast“-Konzept, das Konzept des „Test Driven Development“, das Vorgehen nach dem „Wasserfall-Modell“, „agiles Vorgehen“ und „DevOps“. Hier soll ermittelt werden, inwieweit *DevOps*-Strategien in Unternehmen eingesetzt werden und ob hier agil oder nach klassischem Vorgehen gearbeitet wird.
Zu bewerten ist danach die Wichtigkeit von „Qualität“ und anschließend die Wichtigkeit der qualitativen Kriterien „Performance“, „Stabilität“ und „Skalierbarkeit“ aus der ISO/IEC 25010:2011 [138]. Zu bewertende sicherheitsbezogene Kriterien sind „Integrität“, „Vertraulichkeit“, „Verfügbarkeit“ und „Verantwortlichkeit“ nach ISO27001 und dem *NIST* [41, 229]. Die Anordnung von generellen zu speziellen Fragen, also in diesem Fall von zunächst der gesamt Qualität und anschließend zu spezifischen Qualitätskriterien, ist von Lietz [161, S. 257] empfohlen, da die Beantwortung einer speziellen Frage vor einer später gestellte allgemeine Frage den Probanden in der Einschätzung beeinflusst. Dies war bei umgekehrter Reihenfolge nicht nachweisbar.
In der Kategorie Tests wird der Stellenwert der Tests „Sicherheits-Test“, „Regressions-Test“, „Akzeptanz-Test“, „Unit-Test“, „Penetrations-Test“ und „Last-Test“ vom Befragten bewertet. So wird ermittelt, welche Tests in Unternehmen wie stark genutzt werden.

Techniken sind aufgeteilt in zwei Fragen. Zunächst wird die Häufigkeit der Nutzung der Techniken „Continuous Delivery“, „Feature Toggles“, „Protokollierung“, „Metriken“, „Automation“ und „Infrastructure as Code“ ermittelt. In der darauf folgenden Frage wird der Einfluss der Techniken und Vorgehen „Häufiges Deployment“, „agiles Vorgehen“ und „Feature Toggles“ bewertet.
Dabei reicht die Skala von „Sehr negativ“ mit Wertigkeit 1 bis „Sehr positiv“ mit Wertigkeit 5. So ist es möglich Konflikte der Techniken und Vorgehen zwischen Anwendungs-Entwicklung, IT-Betrieb und IT-Sicherheit zu erkennen.
Fragen zum sozio-demografischen Hintergrund werden im letzten Teil des Fragebogens gestellt, da Probanden nachweislich zu Beginn einer Befragung nur bedingt persönliche Daten preisgeben [161, S. 41ff.]. Im demographischem Teil ist neben der Jahre Berufserfahrung und dem Geschlecht nach dem Wissen in den Bereichen Software-Entwicklung, System-Management, Netzwerk-Management, IT-Sicherheit und Projekt-Management gefragt. So wird im Sinne von *DevOps* nicht nach dem Beruf „DevOps“ gefragt. Vage Selbsteinschätzungen können beim Fragebogen bemängelt werden [161, S. 255]. Eine Alternative stellt die Frage nach Beruferfahrungs-Jahren in den unterschiedlichen Bereichen dar. Dies beinhaltet die Möglichkeit, dass Probanden in einem Bereich durch Trainings ein hohen Ausbildungsgrad erreicht haben, in einem anderem Bereich jedoch keine Ausbildung erhalten haben aber länger gearbeitet haben. Dies verfälscht die Fragestellung ebenfalls.

2.5.1 Ergebnisse

93 Befragte aus unterschiedlichen Bereichen haben an der Umfrage teilgenommen. Dabei wurde der schriftliche Fragebogen bei den Experten-Treffen „OWASP Stammtisch Hamburg“ in Hamburg am 08.03.2016, auf der „CeBIT“ in Hannover am 15.03.2016, bei der Studie „Team-Sicherheitsprüfungen“ in einem Kieler Unternehmen am 25.04.2016, beim *„DevOps* HH Meetup“ in Hamburg am 27.04.2016, bei den „DevOps Days“ in Kiel vom 12.05.2016 bis 13.05.2016 und beim „KoSSE-Tag“ in Kiel am 01.06.2016 verteilt. Zusätzlich wurden vereinzelt Experten angesprochen, welche den digitalen Fragebogen ausgefüllt haben. Als „Erfahren“ in einem Bereich wird Wissen mit „Hoch“ angesehen und als „Experte“ wird Wissen mit „Sehr hoch“ angesehen.
41 Befragte verfügen in mindestens einem Bereich über Experten-Wissen und entsprechend verfügen 52 Befragte über kein Experten-Wissen in einem Bereich. Experten-Wissen ist in den Bereichen Software-Entwicklung 22 mal, System-Management 13 mal, Projekt-Management 9 mal, IT-Sicherheit 7 mal und Netzwerk-Management 1 mal vertreten. Erfahrnen-Wissen ist in den Bereichen Software-Entwicklung 43 mal, Projekt-Management 35 mal, System-Management 15 mal, IT-Sicherheit 20 mal und Netzwerk-Management 13 mal vertreten.
Da nur ein Befragter Experte im Bereich „Netzwerk-Management“ angegeben hat, ist diese Gruppe aus der weiteren Betrachtung ausgeschlossen.
Die Ergebnisse der Befragung nach Wissen in den unterschiedlichen Bereichen sind im Anhang in Tabelle C.1 auf Seite 121 aufgeführt. Im arithmetischem Mittel $\bar{x}$ haben die Befragten 9,72 Jahre Berufserfahrung, wobei die Standardabweichung σ bei 7,27 liegt.
Zur Auswertung, welche von *LimeSurvey* als Komma separierte Liste vorliegen, wurde eine Web-Anwendung[3]

[3]Der Quellcode ist inklusive der exportierten Antworten als Komma separierte Liste unter https://github.com/wurstbrot/survey verfügbar.

erstellt. Diese ist einsehbar unter http://survey.timo-pagel.de/results/. Die Antworten sind als Histogramm und Tabelle angegeben. Weiterhin kann nach jeder Kategorie innerhalb einer Frage aufsteigend und absteigend sortiert werden. In dem Histogramm können Kategorien wie „Keine Antwort“ ausgeblendet werden. Zusätzlich kann nach dem angegebenen Wissen in den unterschiedlichen Bereichen gefiltert werden, um die Sicht einer speziellen Gruppe aufzuzeigen. Dies vereinfacht die multidimensionale Auswertung der Antworten.

Zur Untersuchung wird für jede Kategorie das arithmetische Mittel $\bar{x}$ und die Standardabweichung σ angegeben, „Bedeutung unbekannt“ und „Keine Antwort“ wird dabei nicht berücksichtigt. „Gar nicht / sehr gering“ hat dabei den Wert 1, „Gering“ den Wert 2, „Mittel“ den Wert 3 usw.

Bei der Frage nach der Wichtigkeit der Anwendung von Modellen und Konzepten in der täglichen Arbeit ist die Kategorie „Agiles Vorgehen“ mit $\bar{x} = 4,25$ am höchsten gefolgt von „DevOps“ und „Test Driven Development“. 19 (20,43%) Befragten ist die Bedeutung von „Fail Fast“ unbekannt. Das „Vorgehen nach Wasserfall-Modell“ ist mit $\bar{x} = 2,19$ am unwichtigsten. Weitere Details sind Tabelle C.2 auf Seite 121 im Anhang zu entnehmen. Die gleiche Reihenfolge der Wichtigkeit ergibt sich bei Aggregation der Antworten von Sicherheits-Erfahrenen und Sicherheits-Experten.

Eine Aufschlüsselung der Wichtigkeit der Eigenschaften der in Produktion befindlichen Systeme und Anwendungen für alle Befragten und für Experten der Bereiche sind in den Abb. 2.2 dargestellt. Die Unter-Abbildung „Bewertung der Gesamtheit aller Befragten“ erfasst auch die 52 Befragten ohne Experten-Wissen in einem Bereich.

In Tabelle C.3 auf Seite 122 im Anhang sind dagegen alle Antworten der jeweiligen Kategorien aufgeführt. Während Integrität für die Gesamtheit der Befragten am viert wichtigsten ist, ist Integrität für Sicherheits-Experten am Wichtigsten.

Der Stellenwert von Tests in der täglichen Arbeit über die Gesamtheit aller Befragten und speziell aus Sicht eines Sicherheits-Experten wird in Abb. 2.3 auf Seite 14 aufgezeigt. 20% der Sicherheits-Erfahrenen beurteilen den Stellenwert von „Sicherheits-Tests“ mit „Gering“, 40% mit „Mittel“, 30% mit „Hoch“ und 10% mit „Sehr hoch“.

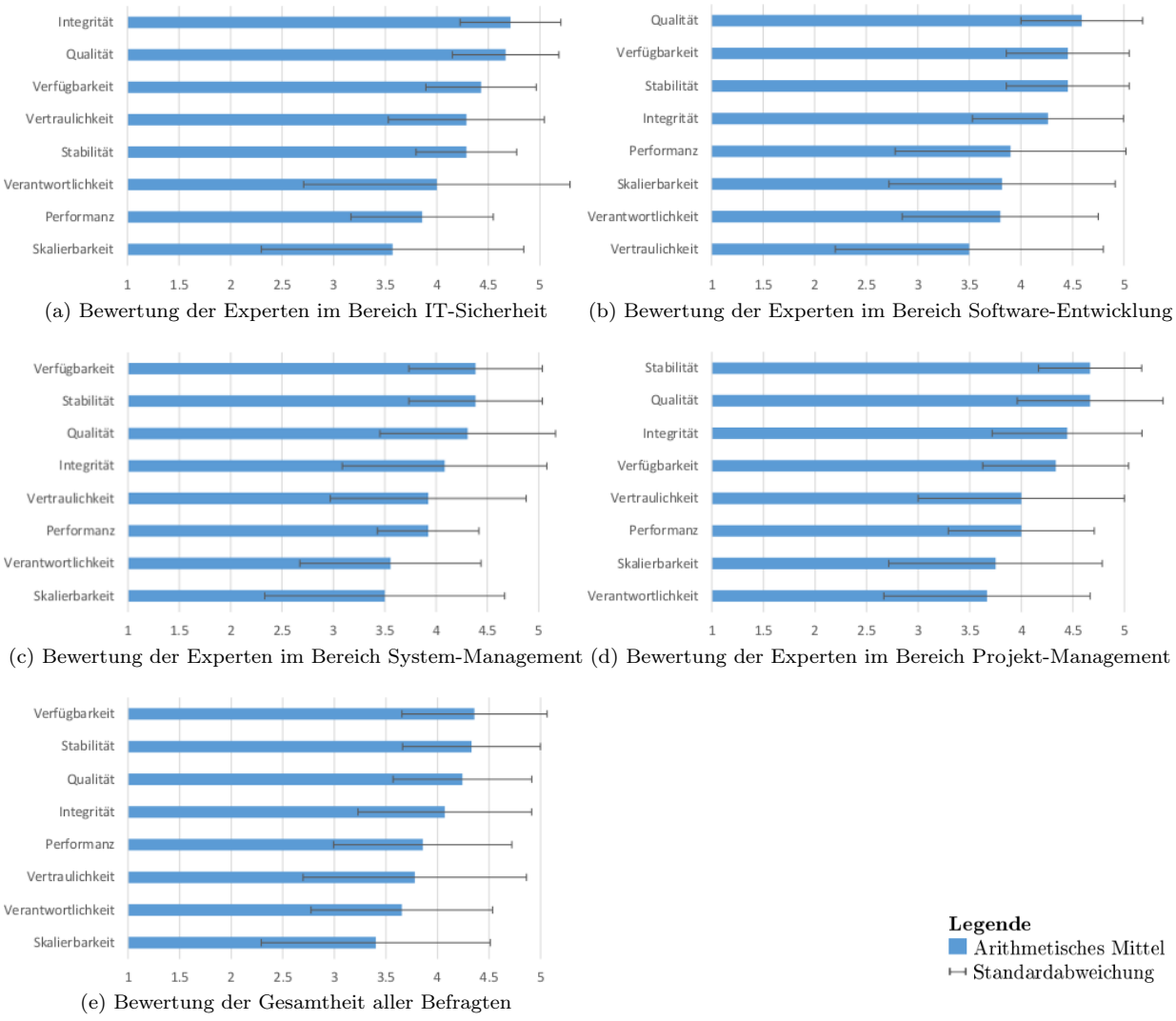
Die Häufigkeit des Nutzens einer Technologie in der täglichen Arbeit über alle Befragten und aufgeschlüsselt nach Experten wird in Abb. 2.4 auf Seite 15 aufgezeigt.

Für die Ergebnisse des Einflusses von Techniken auf die tägliche Arbeit wurde, wie bei den vorherigen Fragen, das arithmetische Mittel gebildet. Dem Wert 1 ist „Sehr negativ“ zugewiesen, der Wert 2 „Negativ“, der Wert 3 „Kein Einfluss“ usw. Abb. 2.5 auf Seite 16 zeigt das arithmetische Mittel für alle Befragten sowie Experten der unterschiedlichen Bereiche.

2.5.2 Evaluierung

„Agiles Vorgehen“ wurde als sehr viel wichtiger als das „Vorgehen nach Wasserfall-Modell“ bewertet, was den Trend zu einem agilen Vorgehen bestätigt. Das Modell „DevOps“ ist am zweitwichtigsten. Das unterstreicht die Wichtigkeit der Erforschung von Sicherheit in *DevOps*.

Sicherheits-Experten bewerten als einzige Experten-Gruppe „Integrität“ als wichtigste Eigenschaft der in Produktion befindlichen Systeme. Hier ist ein Konflikt mit den anderen Bereichen zu erwarten, welche „Stabilität“, „Qualität“ und „Verfügbarkeit“ als wichtigste Eigenschaften bewertet haben. Beispielswei-



(a) Bewertung der Experten im Bereich IT-Sicherheit

(b) Bewertung der Experten im Bereich Software-Entwicklung

(c) Bewertung der Experten im Bereich System-Management

(d) Bewertung der Experten im Bereich Projekt-Management

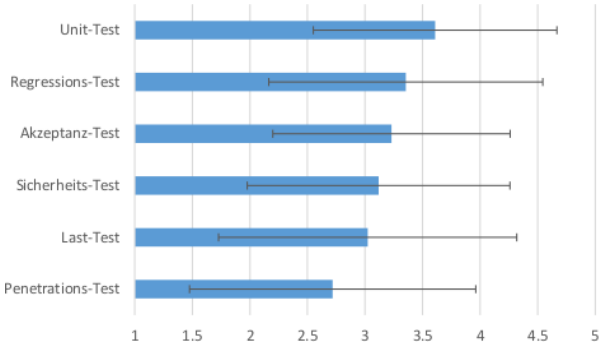
(e) Bewertung der Gesamtheit aller Befragten

Abbildung 2.2: Ergebnisse der Befragung nach der Wichtigkeit von Qualitätskriterien aufgeschlüsselt nach Expertengruppen und der Gesamtheit aller Befragten

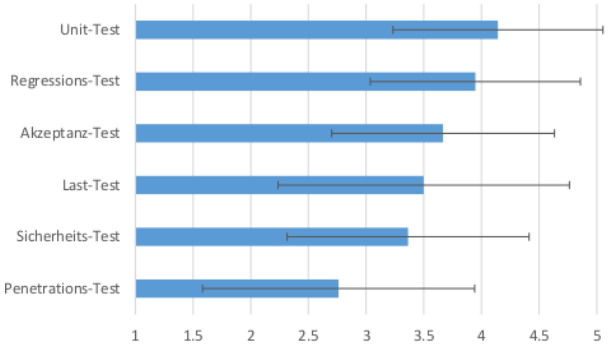
se wird Replikation in verteilten System verwendet, um die Verfügbarkeit zu verbessern. Es kann sich jedoch die Verfügbarkeit in verteilten Systemen verschlechtern, wenn die Konsistenz der Replikate in Fehlersituationen, wie einem Rechnerausfall, eingehalten werden muss [185].

„Vertraulichkeit“ wird von Software-Entwicklungs-Experten als am unwichtigsten bewertet, während es von Sicherheits-Experten als am viert-wichtigsten bewertet ist. Nur für Projekt-Management-Experten ist Verfügbarkeit am viert-wichtigsten. Dies ist gegen die Erwartung, da bei Projekt-Management-Experten ein erhöhtes Geschäftsverständnis vorhanden ist. Bei Beeinträchtigung der „Verfügbarkeit“ sind Betriebsmittel implizit nicht benutzbar [183, S. 14]. Damit kann kein Umsatz mehr geniert werden. Für Experten im Bereich Software-Entwicklung ist „Verfügbarkeit“ nach „Qualität“ am zweitwichtigsten. Dies ist ebenfalls gegen die Erwartung, da bei *DevOps* davon ausgegangen wird, dass die Software-Entwicklung eine Anwendung produziert und nicht mehr für die Produktivumgebung zuständig fühlt. Die Bewertung der „Performance“ ist am fünft Wichtigsten und entspricht dagegen wieder der Erwartung. Die Auswahl von Qualitätskriterien ist für Software-Entwickler dabei nicht sehr hoch. Für Software-Entwickler sind wahrscheinlich Software-Qualitätskriterien wie „Benutzbarkeit“ und „Änderbarkeit“ von hoher Wichtigkeit.

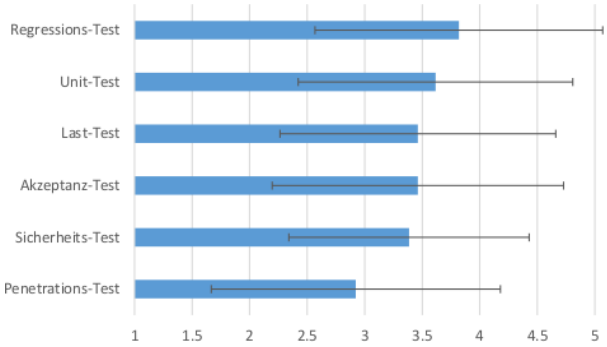
Unter allen Befragten hat der „Unit-Test“ den höchsten Stellenwert aller Tests. „Sicherheits-Tests“ sind an



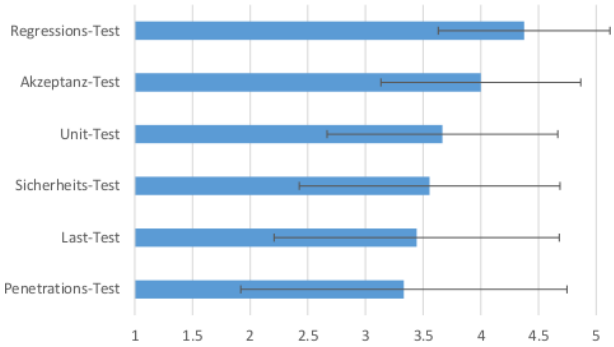
(a) Bewertung der Experten im Bereich IT-Sicherheit



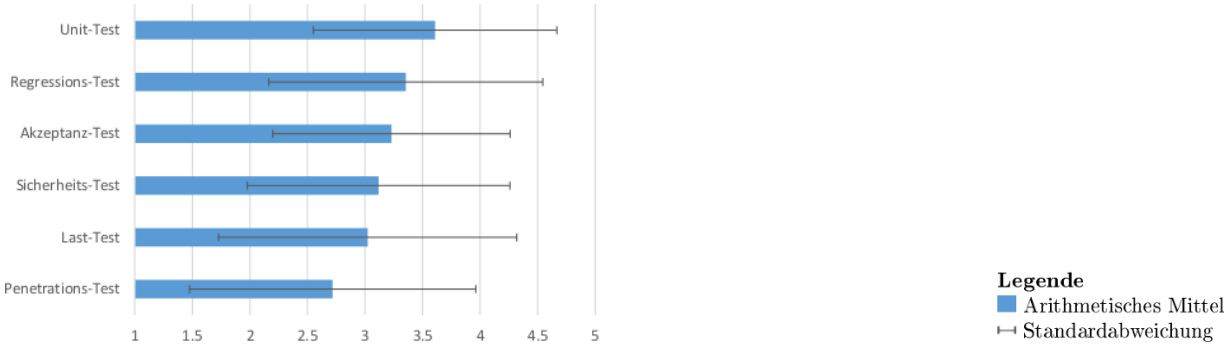
(b) Bewertung der Experten im Bereich Software-Entwicklung



(c) Bewertung der Experten im Bereich System-Management



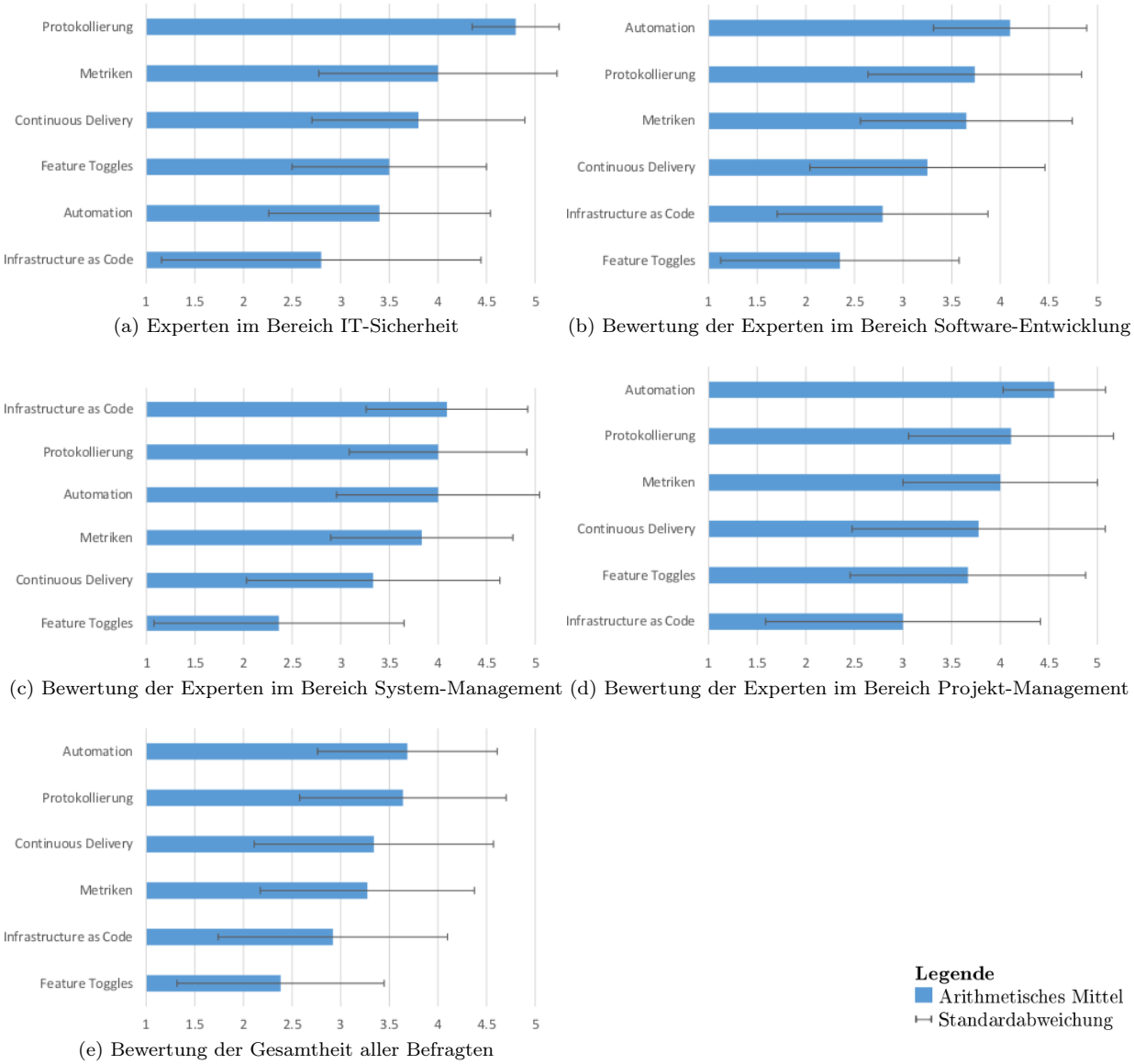
(d) Bewertung der Experten im Bereich Projekt-Management



(e) Bewertung der Gesamtheit aller Befragten

Abbildung 2.3: Ergebnisse der Befragung nach dem Stellenwert von Tests aufgeschlüsselt nach Expertengruppen und der Gesamtheit aller Befragten

Position vier und Penetrationstests haben den niedrigsten Stellenwert. Für Sicherheits-Experten haben erwartungsgemäß „Sicherheits-Tests“ den höchsten Stellenwert, gefolgt von „Penetrations-Tests“. Dies zeigt, dass je nach Gruppe verschiedene Tests Vorrang haben. Da „Unit-Tests“ und „Sicherheits-Tests“ häufig auf unterschiedlichen Umgebungen eingesetzt werden, sind Konflikte hier unwahrscheinlich.
Unter allen Bereichen ist „Automation“ die am meist genutzte Technik, gefolgt von „Protokollierung“ und „Continuous Delivery“. Unter Sicherheits-Experten werden „Protokollierung“ und „Metriken“ am meisten genutzt, wahrscheinlich weil diese Techniken am ehesten dem Bereich Sicherheit zuzuordnen sind. Durch Metriken können Angriffe wie ein *Bruteforce*-Angriff visualisiert und Gegenmaßnahmen ergriffen werden. Durch Webserver-Protokolle lassen sich beispielsweise SQL-Injections in der *URL* ermitteln. In allen Bereichen, abgesehen von Sicherheit, sind „Feature Toggles“ am wenigsten genutzt. Für Sicherheits-Experten haben „Feature Toggles“ einen höheren Wert als Automation und „Infrastructure as Code“. Dies lässt sich ggf. dadurch erklären, dass Sicherheits-Experten „Feature Toggles“ zum Aktivieren und Deaktivieren von



(a) Experten im Bereich IT-Sicherheit

(b) Bewertung der Experten im Bereich Software-Entwicklung

(c) Bewertung der Experten im Bereich System-Management

(d) Bewertung der Experten im Bereich Projekt-Management

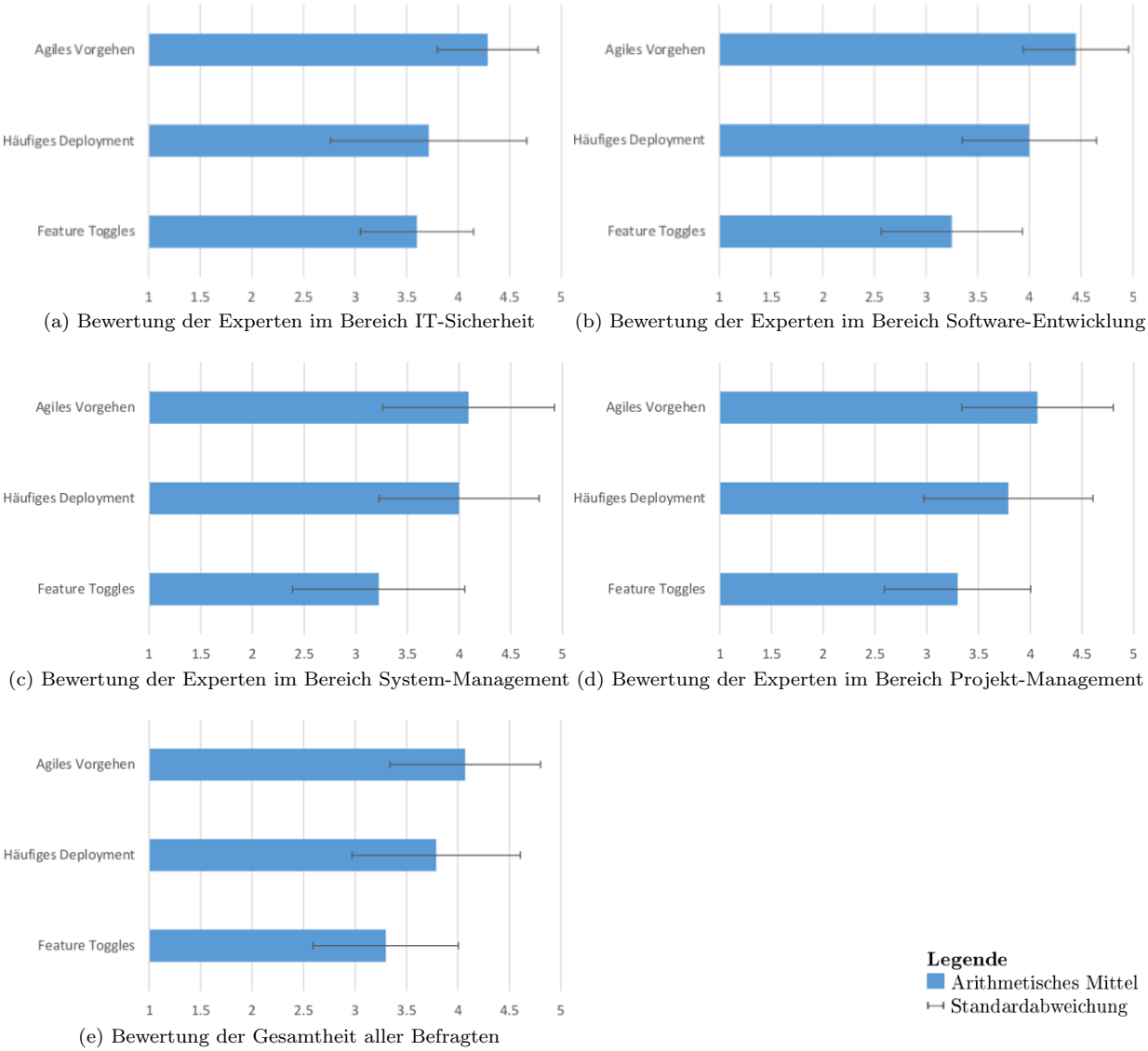
(e) Bewertung der Gesamtheit aller Befragten

Abbildung 2.4: Ergebnisse der Befragung nach der Wichtigkeit genutzter Techniken aufgeschlüsselt nach Expertengruppen und der Gesamtheit aller Befragten

Sicherheitsmaßnahmen in Anwendungen nutzen. Erwartungsgemäß ist der Stellenwert von „Infrastructure as Code“ für System-Manager am höchsten.

Einige Teilnehmer verstanden unter dem Begriff „Vertraulichkeit“ das Vertrauen der Kunden in eine Anwendung. Entsprechend sollte bei zukünftigen Umfragen der Begriff erläutert werden. Weiterhin war „Häufiges Deployment“ zunächst mit „Hoher Deployment Zyklus“ betitelt. „Hoher Deployment Zyklus“ ist dabei nicht eindeutig, da unter diesem Begriff auch ein verlängerter „Deployment Zyklus“ verstanden werden kann. Entsprechend ist der Punkt umbenannt. Es wurde ebenfalls von Befragten angemerkt, bei der Auswahl des Geschlechts auch die Möglichkeit „Andere“ anzugeben, damit unschlüssige Personen sich nicht festlegen müssen.

Beim Ausfüllen des Fragebogens haben u.a. zwei Befragte des gleichen Teams ihre Antworten verglichen und bemerkt, dass sie unterschiedliche Qualitätsanforderungen an die gleiche Anwendung stellen. Beide arbeiten als Software-Entwickler. Dies zeigt, dass auch Befragte aus dem gleichem Team unterschiedlich bewerten. Im Anschluss möchten sich die Befragten mit ihrem Team treffen, um ein gemeinsames



(a) Bewertung der Experten im Bereich IT-Sicherheit

(b) Bewertung der Experten im Bereich Software-Entwicklung

(c) Bewertung der Experten im Bereich System-Management

(d) Bewertung der Experten im Bereich Projekt-Management

(e) Bewertung der Gesamtheit aller Befragten

Abbildung 2.5: Ergebnisse der Befragung nach dem Einfluss von genutzten Techniken aufgeschlüsselt nach Expertengruppen und der Gesamtheit aller Befragten

Verständnis von Qualitätsanforderungen für ihre Anwendungen im Team aufzubauen.

Bei zukünftigen Befragungen sollten vermehrt Software-Qualitätskriterien mit aufgenommen werden.

2.6 Reifegradmodelle

2.6.1 Verwandte Reifegradmodelle

Reifegradmodelle dienen der Ermittelung der Qualität von Prozessen.

Für die Bestimmung des Reifegrades in Bezug auf Sicherheit kann beispielsweise das *Open Software Assurance Maturity Model* [88] genutzt werden. Es hilft Organisationen, Sicherheitsanforderungen zu formulieren und eine Implementierungsstrategie zu erstellen [88].

Schneider hat das *Security DevOps Maturity Model* [49] (kurz *SDOMM*) 2015 entworfen. Aufgrund seiner Erfahrungen hat Schneider ein Modell zur Ermittlung der Qualität von Sicherheitstests in einer Organisation entwickelt. Das Modell verfügt über die vier Dimensionen dynamische Tiefe, statische Tiefe, In-

tensität und Konsolidierung. Die Dimensionen gehen auf das automatische Testen von Webanwendungen ein. Die Intensität gibt Auskunft über die Stärke der Prüfungen und Konsolidierung über die Behandlung von gefundenen Schwachstellen. Auf Nachfrage am 20.02.2016 wurde bestätigt keine Dokumentation zum Entwurf des Modells veröffentlicht ist.

2.6.2 Entwicklung von Reifegradmodellen

Die Entwicklung von Reifegradmodellen kann nach Bruin und et al. [101, S. 2] in die in Abb. 2.6 gezeigten sechs Phasen unterteilt werden.

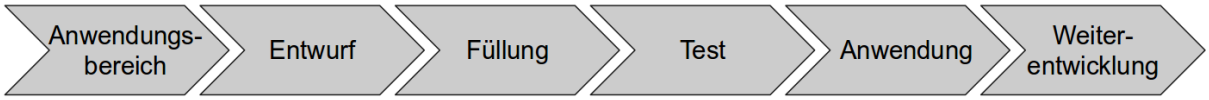


Abbildung 2.6: Entwicklungs-Phasen eines Reifegradmodells in Anlehnung an [101, S. 2]

In der ersten Phase wird der „Anwendungsbereich“ definiert. Hierbei ist ein domänenspezifisches oder domänenneutrales Modell (wie *Total Quality Management*) zu definieren. Im Weiteren werden die an der Entwicklung des Modells beteiligten Stakeholder beschrieben. [101, S. 2ff.]
In der Phase „Entwurf“ ist die Architektur zu entwerfen und die Zielgruppe zu definieren. Das Modell kann eindimensional oder multidimensional sein. Häufig werden mehrere Ebenen von 1 „Initial“ bis 5 „Optimierend“, wie beispielsweise beim *Capability Maturity Model* [191], genutzt. Anschließend wird die Architektur mit konkreten Merkmalen „gefüllt“. Schwierig bei einem multidimensionalem Modell ist, dass sich die unterschiedlichen Dimensionen nicht Redundanzen aufweisen, aber ein vollständiges Bild abgeben. Häufig wird diese Phase durch explorative Studien unterstützt. In der Phase „Test“ wird das Modell auf Relevanz, Validität, Verlässlichkeit und Generalisierbarkeit geprüft. Dabei überschneidet sich diese Phase mit der Vorherigen, da bereits in Experteninterviews geprüft werden kann. In der Phase „Anwendung“ wird das Modell, möglichst von einer dritten Partei, implementiert. In der letzten Phase wird das Modell „weiterentwickelt“ und an neue Gegebenheiten angepasst. [101, S. 2ff.]

Kapitel 3

Kulturelle und organisatorische Maßnahmen

Bei der Einführung von *DevOps* liegt der Fokus häufig auf Technologien, die *Culture* und das *Sharing* in *CAMS* werden jedoch vergessen. Das kann dazu führen, dass die Einführung von *DevOps*-Strategien scheitert. Grundsätzlich sollte eine Organisation Sicherheitsziele kommunizieren, damit alle Mitarbeiter entsprechend handeln können.
Entwickler und System-Administratoren müssen die Komplexität der genutzten Umgebungen, Dienste und Werkzeuge verstehen, entsprechend muss dieses Verständnis geschaffen werden [167, S. 2]. Vor der Einführung von neuen Technologien und Prozessen sollten entsprechend Schulungen erfolgen, in welchen die neue Technologie oder der neue Prozess vorgestellt wird. Ohne ist die Wahrscheinlichkeit, dass neue Technologien oder Prozesse nicht genutzt werden, sehr hoch.
Während agile Methoden eine Änderung der Denkweise anstreben, strebt *DevOps* eine Kulturveränderung an. Beispielsweise kann es bei agiler Vorgehensweise auftreten, dass technische Schulden, wie die Prüfung aller Änderungen auf Sicherheit, aufgebaut werden. Bei *DevOps* wird dies durch erhöhten Austausch zwischen den Mitgliedern eines Teams mit interdisziplinären Kompetenzen vermieden.
DevOps-Teams arbeiten i.d.R. nach einem agilem Vorgehen, um möglichst schnell Software ausliefern zu können. Entsprechend wird im Folgendem darauf eingegangen, wie Sicherheit in agile Methoden integriert werden kann. Eine Kulturveränderung wird am Beispiel der Reduzierung der Fehleranfälligkeit von Personen und Systemen vorgestellt. Anschließend wird durch eine Studie evaluiert, ob Teams gemeinsam eine Prüfung der Sicherheit ihrer Anwendungen durchführen sollten um sich für das Thema Sicherheit erweitert zu sensibilisieren.

3.1 Integration von Sicherheit in agile Methoden

Agile Methoden werden genutzt, um effizienter entwickeln zu können. Dabei helfen insbesondere kleine Zyklen, in welchen am Anfang definiert wird, welche Aufgaben am Ende des Zyklus erledigt sein sollen. In diesem Abschnitt wird die Kenntnis von agilen Entwicklungs-Methoden vorausgesetzt, diese sind z.B. im *Scrum Guide* [216] erläutert. Hier ist u.a. folgendes definiert:

> Das Entwicklungsteam besteht aus Profis, die am Ende eines jeden Sprints ein fertiges Inkrement übergeben, welches potentiell auslieferbar ist.

Zum Ende des *Sprints* sollten also alle Änderungen die gewünschten Qualitätsanforderungen, auch die an die Sicherheit, erfüllen. Dabei wird Sicherheit in agilen Methoden häufig nicht beachtet. Eine Ursache kann sein, dass Sicherheit die Entwicklung von neuen Funktionen behindert [256]. Deshalb ist es notwendig aufzuzeigen, wie Sicherheit in agile Methoden integriert werden kann.
Bei der Integration von Sicherheit in agile Methoden sollten nach Siponen et al. in [223] darauf geachtet werden, dass

1. Der Sicherheitsansatz die agilen Phasen adaptiert ist,
2. die Integration einfach ist und die Entwicklung nicht behindert,
3. Sicherheit in alle agilen Phasen integriert wird und Hilfestellung und Werkzeuge bereitstellt und
4. Sicherheits-Komponenten sich bei schnell verändernden Anforderungen ebenfalls schnell in Iterationen anpassen lassen.

Im Folgenden wird anhand der agilen Entwicklungs-Methode *Scrum* die Integration von Sicherheit in die unterschiedlichen agilen Phasen beschrieben.
Bei der Erstellung des *Sprint Plannings* und *Sprint Backlogs* sollten Sicherheitsanforderungen definiert werden. Der *Product Owner* sollte also Sicherheits- und Datenschutz-Risiken, Bedrohungsanalyse, Spezifikation von positiven Sicherheitsanforderungen, Spezifikation von nichtfunktionalen Sicherheitsanforderungen und *Abuse User Stories* (kurz *AbUser Stories*) beachten [110]. Eine Bedrohungsanalyse sollte während der Anforderungsanalyse bei der Aufteilung in *Users Stories* und beim Entwurf durchgeführt werden [240]. Um den Prozess möglichst einfach zu halten, kann diese mit einer einfachen Risikomatrix, in welcher das Schadenpotential und die Wahrscheinlichkeit des Eintritts eines Risikos geschätzt wird [240], durchgeführt werden. Wird mehr Sicherheit gewünscht kann eine Bedrohungsanalyse gestützt durch Modellierung der Architektur genutzt werden. Dafür kann *Microsoft's © Threat Modeling Tool 2016* [173] eingesetzt werden.
Bei positiven Sicherheitsanforderungen (siehe Abschnitt 2.4) werden diese als *User Story* beschrieben [240].
AbUser Stories sind eine Erweiterung der in agilen Methoden etablierten *User Stories*. Hier wird die Sicht eines Angreifers eingenommen und *Stories* mit negativen Sicherheitsanforderungen mit der Folge eines negativen Geschäftswert beschrieben [192]. Während *User Stories* die Anforderungen eines Endbenutzers beschreiben, beschreiben *AbUser Stor*ies die Sicht eines Angreifers. Es handelt sich hier um die Beschreibung einer Funktion, die das System verbieten sollte [83, S. 27]. In einer *Shop*-Webanwendung kann die *User Story* „Als Kunde möchte ich Produkte in den Warenkorb hinzufügen, um diese zu kaufen“ durch die *AbUser Story* „Als Angreifer möchte ich Anfragen manipulieren um die Preise der Produkte im Warenkorb zu ändern“ ergänzt werden [110]. Dabei wird die Implementierung nicht vorgegeben und es können einer *User Story* mehrere *AbUser Stories* zugewiesen werden. Es gilt zu vermeiden, Sicherheitsanforderungen und *AbUser Stories* während des Sprints niedriger zu priorisieren oder zu ignorieren, da sonst technische Schulden im Bereich Sicherheit aufgebaut werden [59]. Zur Unterstützung bei der Erstellung von *AbUser Stories* kann ein generischer Katalog für *AbUser Stories,* wie in [59], unterstützen.

Hier erhalten die generischen *AbUser Stories* u.a. die Sicht aus der Rolle eines Architekten, Entwicklers und Testers. In erweiterten *AbUser Stories* wird die Rolle eines Sicherheits-Experten hinzugefügt, hier werden z.B. Sicherheits-Audits und automatische Sicherheits-Tests adressiert.
Im *Product Backlog* wird die Bedrohungsanalyse auf Geschäftsprozess-Ebene gepflegt, *AbUser Stories* erstellt, Sicherheitsmaßnahmen genutzt und Akzeptanzkriterien für Sicherheit definiert [110].
Beim *Sprint Planning* erfolgt eine technische Bedrohungsanalyse [240]. *AbUser Stories* und Sicherheits-Testfälle werden konkret beschrieben [110].
Im *Daily Scrum* werden neue Risiken diskutiert und ggf. Aufgaben neu geplant.
Beim *Sprint* sollten Sicherheitsrichtlinien beachtet, mit einem Sicherheits-Experten im *Pair-Programming* entwickelt und Sicherheits-Reviews durchgeführt werden [110]. Kontinuierliche Sicherheits-Reviews nach Änderungen können die Sicherheit erhöhen. Entwickler können vor jeder *Pull*-Anfrage ein Quellcode-Review durch eine zweite Person durchführen. System-Administratoren sollten jede Änderung des Produktionssystems mit einer zweiten Person durchführen oder bei der Nutzung eines Versionskontrollsystems für Infrastruktur-Artefakte Änderungen via Review freigeben. Während der Entwicklung ist zu empfehlen eine Bedrohungsmodellierung, Richtlinien zur sicheren Entwicklung, Sicherheits-Reviews und Sicherheits-Tests zu nutzen [110].
Ein *Scrum*-Team nutzt vor der Freigabe des Ergebnisses eines Sprints *Definition of Done*, in welchem Fertigstellungskriterien festgelegt sind. Die Fertigstellungskriterien müssen entsprechend vorher durch Qualitätsanforderungen definiert worden sein [33]. Bei der Freigabe können durch den *Product Owner* Restrisiken akzeptiert werden [240].
Ideal ist ein Sicherheits-Experte in jedem Team, welcher bei allen agilen Phasen unterstützt [207, S. 20]. Um Kosten zu sparen, werden häufig keine Sicherheits-Experten, sondern *Security-Champions* in einem Team eingesetzt, welche für Sicherheit in ihrem Team verantwortlich sind [175]. *Security-Champions* nehmen dann z.B. an wöchentlichen oder monatlichen Treffen zum Thema Sicherheit teil. In den Treffen werden Sicherheitsvorfälle diskutiert und Schulungen vorgenommen.
Um die kontinuierliche Erhöhung der Sicherheit transparent aufzuzeigen, können Sicherheits-Metriken genutzt werden. Beispielsweise kann ermittelt, visualisiert und kommuniziert werden, wie viele Risiken unterschiedlicher Kritikalität eine statische Analyse für jede neue Software-Version meldet.

3.2 Reduzierung der Fehleranfälligkeit durch organisatorische Maßnahmen

War Games sind ein Ansatz der Gamifizierung, bei denen ein Sicherheitsexperte ein sicherheitsrelevantes Szenario entwickelt und vorbereitet [72]. Beispielsweise den Ausfall einer Netzwerkschnittstelle eines Datenbankservers oder ein *Bruteforce*-Angriff auf Benutzerkonten. Anschließend wird das Angriffsszenario auf einer produktionsnahen Umgebung ausgeführt. Das zugehörige Team, welches für die Betreuung des Systems und der Anwendung zuständig ist, hat die Aufgabe das System wieder zu reaktivieren oder den Angriff zu analysieren und Gegenmaßnahmen einzuleiten. Dadurch werden Notfallpläne geübt und das Risiko, dass eine Person von der Situation bei einem realem Szenario überfordert ist, minimiert.
Die Netflix Inc. geht einen Schritt weiter und lässt Produktionssysteme zufällig ausfallen. John Ciancutti

erläutert dies mittels [92]:

> The best way to avoid failure is to fail constantly.

Die Netflix Inc. nutzt dafür einen Dienst, welche automatisiert zufällig andere Dienste in Produktion ausschaltet. Dadurch wird sichergestellt, dass in der Produktionsumgebung keine Änderungen manuell auf einem System installiert werden. Entsprechend müssen alle Änderungen, auch an Betriebssystem-Konfigurationen, versioniert werden.

3.3 Team-Sicherheitsprüfungen

Der chinesische Philosoph Lao Tzu (sechstes Jahrhundert vor Christus) sagte:

> If you tell me, I will listen. If you show me, I will see. If you let me experience, I will learn.

Lao Tzu zielt damit auf die Basis für aktives Lernen. Inzwischen haben Forschungen ergeben, dass interaktive Elemente, welche aktive Mitarbeit erzwingen, die Effektivität der Entwicklung von Fähigkeiten erhöhen [73].
In Organisationen können Sicherheits-Experten eingesetzt werden um einen Vortrag zu Webanwendungs-Sicherheit zu halten. Häufig werden zur Optimierung in Vorträgen praxisnahe Webanwendungen mit potentiellen Schwachstellen integriert, so dass Teilnehmer die Chance haben, selbst auf Schwachstellen zu untersuchen. Ein Lehrkonzept für das Modul „Sicherheit in Webanwendungen“, entwickelt und an der Fachhochschule Kiel angewandt von Timo Pagel, welches auch Organisationen zur Wissensvermittlung nutzen können, ist unter [189] abgelegt.
Bei Workshops und Vorträgen werden häufig andere Technologien als in Teams beziehungsweise Organisationen verwendet. Entsprechend benötigen Organisationen ein Konzept, durch dessen Implementierung praxisnah und nachhaltig das Sicherheitsbewusstsein sowie das Wissen im Bereich Webanwendungs-Sicherheit der Mitarbeiter erhöht wird.
Ein Ansatz ist eine Team-Sicherheitsprüfung (TSP), welche von Benjamin Pfänder vorgeschlagen wurden (siehe Anhang B.5 auf Seite 114) und hier evaluiert worden sind. TSPen sind als Ergänzung zu bestehenden Maßnahmen wie *Security Champions*, Schulungen und Workshops zu verstehen.
Skeptiker wie der Senior Web-Sicherheitsexperte Matthias Rohr empfehlen, sich statt auf TSPen mehr auf *Security Champions* zu konzentrieren (siehe Anhang B.9 auf Seite 116). Entsprechend erfolgt die Evaluierung der TSP durch eine explorative Studie.

3.3.1 Motivation und Hypothesen

Qualitätsprüfungen sollten möglichst von unabhängigen Personen durchgeführt werden [250]. Diese sollten nicht am Entwurf oder an der Implementierung der zu prüfenden Anwendung beteiligt sein. Entsprechend ermöglichen TSPen einen unabhängigen Sicherheits-Test. Im Fokus stehen dabei die nachhaltige Erhöhung der Sicherheit der Webanwendung durch das Aneignen von Wissen, das Teilen von Wissen und das Schaffen von Bewusstsein im Bereich Sicherheit. Dieses wird unterstützt durch die praktische Anwendung von Sicherheitswerkzeugen an einer Webanwendung und deren Systeme mit gleichen oder ähnlichen Technologien wie im Team.

Bei einer TSP prüft Team A die Webanwendung von Team B und Team B die Webanwendung von Team A, dargestellt in Abbildung 3.1 (links).

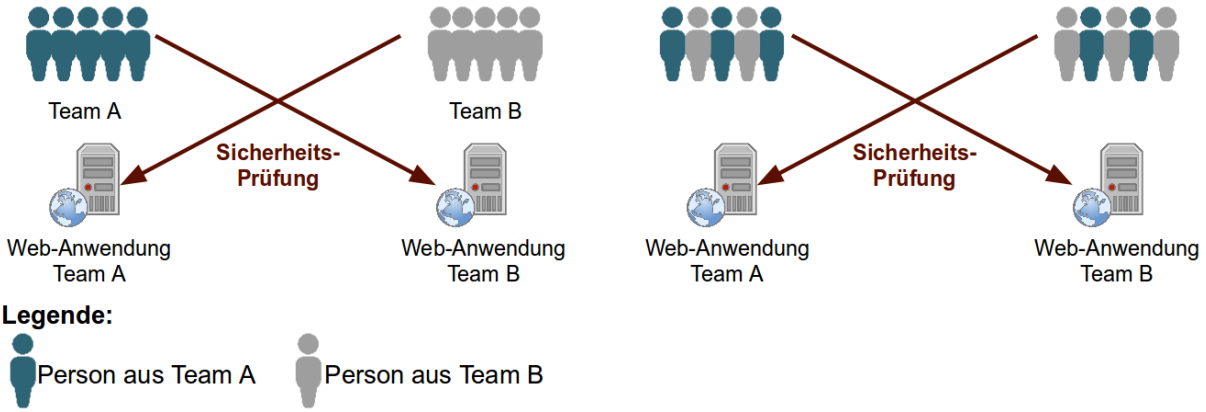


Abbildung 3.1: Homogene (links) und inhomogene (rechts) Team-Sicherheitsprüfungen

Alternativ kann eine inhomogene Team-Zusammensetzung genutzt werden, wie in Abbildung 3.1 (rechts) dargestellt. Die Hypothesen zu den Vorteilen (nachfolgend Vorteil-Hypothese genannt) und Nachteilen (nachfolgend Nachteil-Hypothese genannt) der beiden TSPen sind in Tabelle 3.1 erfasst. Durch die Entstehung von Gesprächen während der TSP und durch die Aneignung von Wissen entsteht ein erhöhtes Sicherheitsbewusstsein. Dabei kann besonders eine Mischung von Erfahrenen und Neulingen einen hohen Austausch von Wissen erzeugen.
Ein erhöhtes soziales Risiko ergibt sich bei inhomogenen Teams. Beispielsweise gibt es Personen, die sich nur ungern an neue Team-Partner gewöhnen. Entsprechend muss Transparenz geschaffen werden, damit jedes Team-Mitglied das gemeinsame Ziel immer vor Augen hat.
Bei der Durchführung muss darauf geachtet werden, dass das Ergebnis der Team-Prüfung keinen Einfluss auf Gehaltsboni, Mitarbeiter- oder Team-Bewertungen hat.
Team-Mitglieder sollen ihre Kompetenzen in den Bereichen Entwicklung, System-Administration und Sicherheit erweitern und dabei die Kompetenzen der anderen Mitarbeiter in der Organisation besser kennen lernen. Je nach Webanwendung eigenen sich Team-Mitglieder fachliches Wissen der zu prüfenden Webanwendung an. Das erlernte Wissen trägt zur nachhaltigen Erhöhung der Sicherheit von den zwei zu untersuchenden Webanwendungen bei.

3.3.2 Vorbereitung und Planung der Studie

Die Studie zur Prüfung der Hypothesen findet bei einem Kieler Unternehmen (nachfolgend KielUnt genannt), welches acht Mitarbeiter im technischen Bereich beschäftigt, statt. Die Studie besteht aus einer Schulung, einer homogenen TSP und einer inhomogenen TSP. Vor der Schulung wird ein populistischer Flyer (siehe Anhang G.2) verteilt, beispielsweise wird die homogene Team-Sicherheitsprüfung „Blackbox Hacking“ und die inhomogene „Whitebox Hacking“.
Hausinterne System-Administratoren, welche die Systeme von KielUnt u.a. administrieren, sind auch nach Erläuterung der Vorteile durch den Anleiter der Studie (siehe Abschnitt 2.1) von einer gemeinsamen Prüfung ausgeschlossen worden.
Bevor TSPen stattfinden können, sollten Grundlagen im Bereich Sicherheit gelegt sein. Es wird deshalb

Art	Vorteile	Nachteile
Homogene und inhomogene Teams	1. Erhöhung des Sicherheitsbewusstseins; Team-Mitglieder achten auch nach der Prüfung verstärkt auf Sicherheit.	10. „Verursacher“ einer Schwachstelle wird ggf. bloßgestellt.
	2. Team-Mitglieder eignen sich Wissen in den Bereichen System-Administration, Entwicklung und Sicherheit an.	11. Abstimmungsaufwand mit Kunden, falls Team A nicht die Webanwendung von Team B einsehen darf.
	3. Teilnehmer teilen Wissen im Bereich Sicherheit.	12. Eine Sicherheits-Prüfung kann effizienter durch einen Sicherheits-Experten alleine (oder mit einem Sicherheits-Team) durchgeführt werden.
	4. Validierung von Sicherheits-Maßnahmen und Sicherheits-Metriken während der Prüfung.	
	5. Eine unabhängige Prüfung durch Dritte erfolgt.	
Homogene Teams	6. Das Team wächst stärker zusammen.	13. Teams könnten in Wettbewerb geraten und sich weniger gegenseitig helfen.
		14. Anwendungsinterna sind fremden Teams unbekannt.
Inhomogene Teams	7. Mitglieder aus Team A lernen Kompetenzen von Mitgliedern aus Team B kennen.	15. Erhöhtes soziales Risiko.
	8. Anwendungsinterna sind bekannt, so dass eine tiefere Analyse stattfinden kann.	
	9. Es ist einfacher, den operativen Betrieb während der Prüfung zu gewährleisten, da bei Prüfungen an unterschiedlichen Tagen aus beiden Teams Personen verfügbar sind.	

Tabelle 3.1: Vor- und Nachteile homogener und inhomogener Teams

eine dreistündige Schulung erfolgen, die detaillierte Planung ist in Anhang G.3 aufgeführt.

Die homogene TSP findet eine Woche nach der Schulung statt. Engagierte Mitarbeiten können sich so vorbereiten, beispielsweise durch eine Sicherheitsprüfung des mit Schwachstellen versehenem *Juice Shops* [148] oder durch Nutzung von Sicherheits-Werkzeugen.

Am Anfang jeder TSP wird die Grundregel „Ziel: Erhöhung der Sicherheit und gemeinsames Lernen“ an einem Whiteboard im Studien-Raum notiert.

Nach einem Monat erfolgt eine zweite Befragung. In dieser wird die Hypothese, dass Team-Sicherheitsprüfungen langfristig das Sicherheitsbewusstsein stärkt und damit die Sicherheit erhöht, geprüft.

Gute Kommunikation über das Thema Sicherheit sollte belohnt werden, beispielsweise mittels Abklatschen (Englisch *High-Fives*), Dankeskarten, Schokolade oder gemeinsamen Mittagessen [156]. Weitere Vor- und Nachteile zu einer Belohnung unter Berücksichtigung des Zeitpunkts sind im Anhang G.4 auf Seite 135 festgehalten. Empfohlen wurde ein Frühstück mit zwei Wochen Versetzung durchzuführen.

Mit dem Abteilungsleiter fand am 30.03.2016 ein Treffen zur Planung statt, die Ergebnisse wurden auf einem Whiteboard festgehalten (siehe Anhang G.1 auf Seite 131). Hier wurde der Ablauf besprochen und Voraussetzungen geplant. KielUnt hat sich für ein Mittagessen während der Prüfung entschieden. Weiter wurde KielUnt darauf aufmerksam gemacht, dass zwei möglichst produktionsnahe Test-Systeme

der jeweiligen Anwendungen zur Durchführung der homogenen TSP benötigt werden. Damit möglichst noch viel Wissen vorhanden ist, sollte auf die Schulung innerhalb von zwei Wochen die homogene TSP erfolgen und innerhalb drei Wochen da drauf die inhomogene TSP stattfinden. Entsprechend wurde der 19.04.2016 für die Schulung, der 25.04.2016 für die homogene und der 23.05.2016 für die inhomogene TSP festgehalten. Es ist also eine Woche mehr Zeit zwischen der homogenen und der inhomogenen TSP als vorgeschlagen.
Während bei homogenen Teams kein oder wenig Wissen über die zu prüfende Webanwendung vorhanden ist, sind bei inhomogenen Teams Anwendungsinterna einzelnen Team-Mitgliedern bekannt. Entsprechend bieten sich hier Whitebox-Prüfmethoden an, welche zunächst als Vortrag vorgestellt werden müssen.

3.3.3 Durchführung und Evaluierung der Schulung

Die Schulung fand am 19.04.2016 statt. Die Einordnung der beiden vorgegebenen Risiken wurde erfolgreich durch die Probanden vorgenommen. Probanden haben von sich aus vier weitere Vorschläge für Risiken für deren Webanwendungen in einer Risikomatrix bewertet (siehe Anhang G.6). Dabei wurde auch erkannt, dass Risiken, welche kein Schadenpotential beziehungsweise keine Wahrscheinlichkeit aufweisen, nicht in die Matrix einzutragen sind.
Während der Schulung wurden die Probanden durch Zuruf-Abfragen über ihr Wissen im Bereich Sicherheit befragt. Die PHP-Funktion `password_hash()`, welche zum sicheren Erzeugen von *Hash*-Werten von Passwörtern genutzt wird, war zwei Probanden bekannt. Das *HTTPOnly Flag* für *Cookies*, welches den Zugriff beziehungsweise die Manipulation von einem (Sitzungs-)*Cookie* via *JavaScript* unterbindet, war keinem Proband bekannt.
Drei Probanden empfanden die Schulung als zu oberflächlich und mit zu wenig Praxis. Es wurde gewünscht, den *Juice Shop* mit mehreren Werkzeugen selbst zu untersuchen. Praktische Anwendung wie im Lehrkonzept unter [189] ist sehr zu empfehlen, konnte jedoch im Rahmen dieser Thesis nicht durchgeführt werden.

3.3.4 Durchführung der homogenen Team-Sicherheitsprüfung

Die homogene Team-Sicherheitsprüfung fand am 25.04.2016 ab 10:00 Uhr mit sechs Probanden statt. Während der aktiven Durchführung hat Timo Pagel die Rolle des Anleiters. Frau Bernecker-Bendixen, eine unabhängige Unternehmensberaterin, ist als Beobachterin hinzugezogen.
Die Probanden haben den Fragebogen zur Identifizierung von Gemeinsamkeiten zwischen unterschiedlichen IT-Bereichen ausgefüllt (siehe Anhang C.1 auf Seite 119). Die Ergebnisse sind in Tabelle G.2 auf Seite 136 im Anhang aufgeführt. 66% der Probanden haben sich selbst mit einem mittlerem Wissen in Sicherheit eingeschätzt, wobei ein Proband den Fragebogen nicht ausgefüllt hat.
Die Probanden erhalten eine virtuelle Maschine[1]. Die virtuelle Maschine enthält DAST- und SAST-Werkzeuge. Dabei wurden die SAST-Werkzeuge aus der Anleitung entfernt, da diese für die Blackbox-Tests nicht benötigt werden.
Bevor die Probanden anfangen, wird an die Durchführung der Risikobewertung von gefunden Schwachstellen erinnert. Dafür werden unter die am Whiteboard notierte Grundregel zwei Risikomatrizen skizziert, in

[1] Welche unter https://github.com/wurstbrot/securityTesting zur Verfügung steht und via *vagrant [37]* provisioniert wird.

welche die Probanden während der Studie Risiken eintragen sollen. Es folgt eine Erinnerung an die Aufteilung des Teams in spezielle Bereiche wie Authentifizierung, Kryptographie und Eingabebehandlung, welche die Probanden selbstständig vornehmen sollen.
Die Test-Systeme waren zum Start von KielUnt nicht aufgesetzt. Um 11:00 Uhr waren beide Test-Systeme aufgesetzt, jedoch fehlten noch Daten, beispielsweise Produkte in dem zu prüfenden Shop-System. Entsprechend konnte nur bedingt geprüft werden. Um 11:45 Uhr waren die beiden Test-Systeme aufgesetzt. In der Zwischenzeit haben die Probanden nach Möglichkeit das Produktionssystem untersucht. Bei einem Proband lief die vorbereitete mit Werkzeugen versehene virtuelle Maschine nicht. Während der TSP entstand Kommunikation über Risiken und Werkzeuge innerhalb der Teams als auch zwischen den Teams. Beispielsweise wurde Team-übergreifend diskutiert, ob beim Erkennen einer Sitzungsübernahme die Sitzung ungültig gesetzt werden sollte. Zusätzlich wurden auch Informationen zu bestehender Absicherung von Webanwendungen ausgetauscht, beispielsweise die Umbenennung des *PHP*-Sitzungs-*Cookies* von *PHPSESSID* in *SESSION*, um möglichst wenig Informationen über die genutzte Programmiersprache *PHP* preiszugeben. Bis 12:44 Uhr haben die Probanden jeweils für sich selbst geprüft und es entstand wenig Austausch. Um 12:44 Uhr wurde das erste Mal eine Aufsplittung innerhalb eines Teams angeregt und Aufgaben wurden aufgeteilt.
Die Beobachterin hat bemerkt, dass für die Vorbereitung des Mittagessens durch den Abteilungsleiter viele E-Mails mit weiteren, nicht an der TSP beteiligten, Mitarbeitern von KielUnt für ca. eine Stunde gewechselt wurden. Das Mittagessen wurde nicht wie vorher besprochen vom Geschäftsführer angekündigt, sondern während der TSP vom Abteilungsleiter. Von 13:00 bis 13:30 Uhr wurde mit allen Team-Mitgliedern Mittag an einem Tisch gespeist.
Beim *Ajax-Spider* mittels *OWASP Zed Attack Proxy* [188] (kurz *Zap*) war zunächst die *HTTP*-Authentifizierung des Test-Systems hinderlich, da diese nicht automatisch durch *Zap* eingegeben werden konnte und die Google Chrome-Fenster des *Ajax-Spiders* automatisch geschlossen wurden. Dies wurde durch Wechsel auf *Mozilla Firefox* gelöst, in welchem beim Aufruf durch den *Ajax-Spider* gewartet wird, bis die *HTTP*-Authentifizierungsdaten pro Browser-Fenster manuell eingegeben sind.
Die Teams haben auf Schwachstellen in der Administrations-Oberfläche, dem Benutzerbereich und *REST*-Schnittstellen geprüft. Dabei wurde auf Benutzung von Komponenten mit bekannten Schwachstellen in *JavaScript*-Bibliotheken, *XSS*, fehlerhafte Autorisierung auf Anwendungsebene, Fehlermeldungen in Authentifizierungs-Management und Schnittstellen, Fingerabdrücke (Englisch *Fingerprinting*) von Diensten, Ordner-Traversierung, Sitzungs-Management, *SQL-Injections* und Verschlüsselung geprüft.
In der Schulung empfohlen, jedoch nicht geprüft wurden *CSRF*, Prozesse der Erzeugung und Verteilung, unsichere direkte Objektreferenzen und die Firewall-Konfiguration. Eine Prüfung auf Komponenten mit bekannten Schwachstellen der eingesetzten serverseitigen Frameworks blieb aus. Obwohl durch den Anleiter darauf hingewiesen wurde, dass es sich hier um eine schnell durchzuführende Prüfung mit hohem Effekt handelt.
Während der TSP wurde von einem Team zwölf und von dem anderem Team elf mal nach Unterstützung gefragt. Unterstützung erfolgte bei der Bedienung von Werkzeugen als auch bei der Analyse von manuell gefundenen oder durch Werkzeuge gemeldeten Risiken.
Nachdem ein Proband aus jedem Team ein Risiko entdeckt hatte, wurde er durch den Anleiter auf die Besprechung des Risikos mit seinem Team aufmerksam gemacht. Dadurch entwickelte sich erhöhte

Zusammenarbeit und Kommunikation innerhalb des jeweiligen Teams. Bemerkte Risiken wurden zum Teil nicht bewertet und nicht in die Risikomatrix übernommen. Der Abteilungsleiter hat Kommentare wie „Das Risiko ist bekannt und nicht wichtig“ zu durch andere Probanden gefundenen Risiken gegeben. Es sind insgesamt neun Risiken identifiziert worden, wovon zwei als hohes Risiko eingestuft wurden. Die Risikomatrizen der beiden Teams sind im Anhang G.7 auf Seite 137 abgelegt. Ein hohes Risiko ist beispielsweise die Auslieferung des Sitzungs-*Cookies* ohne *HTTPOnly*-Flag. Dabei wurde das Risiko zunächst aufgrund des unter https://blog.codinghorror.com/protecting-your-cookies-httponly/ veröffentlichten Artikels von 2008 als gering eingeschätzt. In dem Artikel wird erläutert, dass Browser das *HTTPOnly*-Flag nicht beachten da es eine neue Technologie ist. Nachdem durch den Anleiter erläutert wurde, dass alle modernen Browser das *HTTPOnly*-Flag unterstützen, wurde das Risiko als hoch eingestuft. Für jedes Risiko wurden gemeinsam Gegenmaßnahmen erörtert. Beispielsweise die Nutzung des *HTTPOnly*-Flags für das Sitzungs-Cookie.

3.3.5 Evaluierung der homogenen Team-Sicherheitsprüfung

Die geringe Anzahl von notierten Risiken kann an einem abschwächendem Kommentar zu gefundenen Risiken vom Abteilungsleiter liegen, so die Beobachterin. Weiterhin haben die Probanden wahrscheinlich wenig Praxis bei der der Einschätzung des Schadens und der Wahrscheinlichkeit von Risiken.
Abschwächende Kommentare des Abteilungsleiters können bei den Probanden zu einer niedrigen Priorisierung vom Qualitätsmerkmal Sicherheit führen. Der Abteilungsleiter wurde darauf von der Beobachterin aufmerksam gemacht. Für die Studie mit inhomogenen Teams wird der Ablauf zum Notieren von Risiken optimiert. Hier sollen zunächst die Risiken ohne Bewertung gesammelt und anschließend gemeinsam der Schaden und die Wahrscheinlichkeit geschätzt werden. Dies soll die Hürde des Schätzens nehmen.
Aufgrund der Kombination eines nicht einsatzbereiten Systems beim Start und des offenen Ablaufs bei der TSP, haben die Probanden zunächst jeder für sich eine Untersuchung vorgenommen, ohne sich abzusprechen. Entsprechend wurde von unterschiedlichen Probanden auf gleiche Risiken geprüft.
Vorteil-Hypothese 3. ist durch Kommunikation innerhalb der Teams bestätigt.
Vorteil-Hypothese 4. wird indirekt bestätigt. Beispielsweise kann bestätigt werden, dass über die auf SQL-Injection geprüften Eingaben keine SQL-Injection möglich sind.
Es wurde kein mal danach gefragt, wer für eine Schwachstelle verantwortlich ist. Entsprechend kann Nachteil-Hypothese 10. durch Wiederholung der Regeln entgegengewirkt werden und deshalb nicht bestätigt werden.
KielUnt benötigt vertraglich nicht die Einverständniserklärung seiner Kunden um die TSP durchzuführen, hat diese jedoch im Vorfeld informiert. Entsprechend kann Nachteil-Hypothese 11. teilweise bestätigt werden. Bei der homogenen TSP wurden insgesamt 23 Risiken aufgedeckt, wobei zwei hohe Risiken bereits geschlossen wurden. Die Risiken wurden zum Teil auch durch Hinweise des Anleiters aufgedeckt, entsprechend kann Nachteil-Hypothese 12. bestätigt werden. Es entstand sechs mal Kommunikation zwischen den Teams, bei welcher sich die Teams gegenseitig geholfen haben. Nachteil-Hypothese 14. kann entsprechend bestätigt werden.
Die vom Abteilungsleiter mit anderen Mitarbeitern gewechselten E-Mails haben den Abteilungsleiter stark abgelenkt und damit die TSP beeinträchtigt. Weiter kann das Bestellen für andere Mitarbeiter den

Belohnungs-Effekt beeinflussen. Hier ist unklar, ob die anderen Mitarbeiter auch eingeladen wurden oder selbst gezahlt haben. Damit ist das Mittagessen nichts besonderes mehr. Das Mittagessen der inhomogenen TSP sollte durch einen Mitarbeiter, welcher nicht an der TSP teilnimmt, organisiert werden damit der Abteilungsleiter nicht damit belastet wird. Im Weiteren lässt sich aus der Mittagessen-Planung und dem nicht aufgesetztem Test-System eine niedrige Priorisierung der TSP von KielUnt ableiten.
Bei der Befragung am Ende der TSP gaben die Team-Mitglieder an sich nur ungern gegenseitig während der TSP geholfen zu haben, was Nachteil-Hypothese 13. bestätigt. Trotzdem wird es als Vorteil gesehen, die Prüfung gemeinsam mit beiden Teams durchzuführen.
Durch eine Befragung vier Wochen nach der TSP wurde ermittelt, das die Probanden verstärkt auf Sicherheit im Rahmen des möglichen achten, dies Bestätigt Vorteil-Hypothese 1. Dabei wurde darauf verwiesen, dass bei Auftragsarbeit der Kunde hier immer das letzte Wort hat und häufig Benutzbarkeit vorzieht und möglichst kostengünstige Entwicklung möchte. Zusätzlich wurden die Probanden befragt, ob ein erhöhtes Team-Gefühl im Anschluss an die TSP empfunden wurde. Dies wurde verneint. Entsprechend kann Vorteil-Hypothese 6. nicht bestätigt werden.
Ein Proband hat besonders das strukturierte Heranführen an Sicherheits-Werkzeuge gefallen. Positiv kam die gemeinsame Behandlung des Themas Sicherheit an. Trotz der Kritik, zu wenig Praxis mit den Werkzeugen vor der Team-Prüfung zu haben, empfand jeder Proband die TSP als Bereicherung und man freut sich auf den nächsten Termin.
In der Befragung einen Monat nach der Prüfung wurde angegeben, dass die beiden als hoch eingestuften Risiken innerhalb von zwei Wochen behoben wurden.
Zap wurde ebenfalls im Anschluss während der Entwicklung einer Schnittstelle genutzt, um die Sicherheit zu prüfen. Ebenfalls wurde bei Quellcode-Reviews, welche bei jeder Überführung von Quellcode in das Versionskontrollsystem durchgeführt werden, verstärkt auf Sicherheit geachtet. Dies bestätigt Vorteil-Hypothese 2. Durch die TSP wurde deutlich, dass Sicherheit in Modul- und Akzeptanztests integriert werden sollten. Bei der Anforderungsanalyse für neue Funktionalitäten wurde dagegen nicht verstärkt auf Sicherheit geachtet, da diese hauptsächlich von Betriebswirten durchgeführt werden.

3.3.6 Durchführung der inhomogenen Team-Sicherheitsprüfung

Die inhomogene TSP wurde von KielUnt aufgrund erhöhter Auslastung vom 23.05.2016 auf den 07.06.2016 verschoben. Die TSP starte um 9:00 Uhr mit sechs Probanden, wobei ein Proband im Bezug auf die homogene TSP gewechselt wurde. Während der aktiven Durchführung hat Timo Pagel die Rolle des Anleiters. Die Ergebnisse zu den Kenntnissen der Probanden in unterschiedlichen IT-Bereichen sind in Tabelle G.3 auf Seite 136 aufgeführt. Es wurde darauf geachtet, dass alle Probanden direkt im Anschluss an die TSP die Fragen komplett ausfüllen. Entsprechend sind alle Fragen komplett beantwortet und die Zeilen „Nicht beendet“ und „Keine Antwort“ werden nicht aufgeführt.
Damit Prüfungen, wie z.B. auf *CSRF* oder die Benutzung von Komponenten mit bekannten Schwachstellen des serverseitigen Frameworks nicht ausbleiben wird bei der inhomogenen TSP der Ablauf verstärkt vorgeben. Dafür ist eine Prüfpunktliste erstellt (siehe Anhang G.8.3 auf Seite 138). Diese orientiert sich an dem *OWASP Testing Guide v4* und dem *OWASP Code Review Guide 2.0*. Der *OWASP Testing Guide v4* stellt eine Prüfpunktliste mit Referenz-Nummer, der Kategorie und dem Testnamen zur Verfügung.

Unter [194] wird eine darauf aufbauende Liste angeboten, welche zusätzlich eine Kurz-Beschreibung und Werkzeuge auflistet.

In der Prüfpunktliste sind einfach zu prüfende Punkte aufgenommen, in dem *OWASP Testing Guide v4* ist beispielsweise der Punkt *Buffer Overflow* aufgeführt, auf welchen schwierig zu prüfen ist und entsprechend in der Prüfpunktliste nicht aufgeführt wird. Die Prüfpunktliste ist in unterschiedliche Kategorien aufgeteilt, wie z.B. *Identity Management Testing*, *Authentication Testing* und *Data Validation Testing*. Die Punkte wurden um Zeit zu sparen nicht ins Deutsche übersetzt. Die Prüfpunktliste enthält für jeden Punkt einen Referenz-Identifikator, eine Kurz-Beschreibung, mögliche Prüf-Werkzeuge und ein freies Feld zum Abhacken von jedem Punkt.

Die TSP beginnt mit einem Vortrag[2], in welchem u.a. eine Erläuterung zu den in der virtuellen Maschine bereitgestellten *SAST*-Werkzeugen erfolgt. Da KielUnt in den zu prüfenden Projekten *PHP* und *JavaScript* einsetzt, werden auf diese besonders Rücksicht genommen, ein genutztes *SAST*-Werkzeug ist beispielsweise *RIPS* [99].

Es werden erweiterte Techniken wie das Mitschneiden der Kommunikation zwischen Komponenten vorgestellt. Es können mittels *MySQL-Proxy* die *SQL*-Kommandos mitgeschnitten werden um die Prüfung auf *SQL-Injections* zu optimieren. Mittels *Zap* kann die *HTTP*(*s*)-Kommunikation zwischen Komponenten wie *Microservices* und Web-Diensten mitgeschnitten und geprüft werden.

Zunächst wurden die Teams in zwei Gruppen aufgeteilt und die zu prüfende Anwendung zugewiesen. Anschließend wurde jeder Person zu prüfende Kategorien aus der Prüfpunktliste zugeteilt.

Die Probanden erhalten die virtuelle Maschine aus der homogenen Prüfung und eine Anleitung für *DAST*- und *SAST*-Werkzeuge.

Die Probanden haben sich den *OWASP Testing Guide v4* und den *OWASP Code Review Guide 2.0* als auch *OWASP Cheat Sheets*[3] geöffnet, um hier weitere Informationen während der Prüfung zu erhalten.

Es sind unbekannte Risiken, bekannte Risiken und Risiken, welche als behoben angenommen wurden, identifiziert worden. Ein unbekanntes Risiko ist beispielsweise eine Fehlermeldung mit Angabe der Version eines genutzten Frameworks von einem externen Anbieter. Auf Architekturebene ist ein gefundenes unbekanntes Risiko beispielsweise das Entfernen von nicht druckbaren Zeichen mittels der *PHP*-Funktion `trim()`, ohne vorher auf eine Maximallänge zu prüfen. Die Untersuchung der Architektur wurde von einem Proband und dem Anleiter gemeinsam durchgeführt. Ein bekanntes Risiko ist beispielsweise die Auslieferung einer Webseite mit *HTTP* ohne Verschlüsselung, auch bei Authentifizierungen. Ein Zertifikat ist hier bereits beantragt. Ein als behoben angenommenes Risiko ist die Registrierung mit schwachen Passwörtern. Hier wurde fälschlich angenommen, dass komplexe Passwörter erzwungen werden.

Ein Proband hat sich geweigert ein gefundenes Risiko bei der Abfrage von Produkten in der zu prüfenden *Shop*-Webanwendung aufzuschreiben. Hier konnte durch Manipulation des GET-Parameters *count* die Anzahl von Produkten in einer *Shop*-Webanwendung unbegrenzt hoch gesetzt werden, welches den Webserver an das Limit seines verfügbaren Arbeitsspeicher brachte und in einer Fehlermeldung resultierte. Bei dem Aufruf wird die Anzahl der Produkte und deren komplette Produktinformationen eingelesen.

[2]Die Folien zu dem Vortrag sind unter https://docs.google.com/presentation/d/16i_lKAxrS6bhrq8C0Kqvtr6UnQY-zr9HNPDeJRwltks/edit?usp=sharing einsehbar, bereitgestellt am 08.06.2016

[3]Die *OWASP Cheat Sheets* geben einen knappen Überblick über unterschiedliche Sicherheits-Themen. Eine Übersicht ist unter https://www.owasp.org/index.php/OWASP_Cheat_Sheet_Series#tab=Master_Cheat_Sheet gegeben. Abgerufen am 08.06.2016.

Ausgenutzt werden kann dies für einen *Denial of Service*, bei welchem mehrere Anfragen auf den Produktkatalog mit einer Anzahl von 1000 Produkten abgesendet werden, wobei jede Anfrage bereits an die Arbeitsspeicher-Limitierung stößt. Auf den Hinweis, dass es sich um ein aufzuschreibendes Risiko handelt, reagierte der Proband aggressiv. Auch nach Erläuterung des Risikos wollte der Proband dieses nicht notieren, so dass der Anleiter das Risiko notiert hat.
Während der TSP wurde 32 mal nach Unterstützung gefragt. Unterstützung erfolgte bei der Bedienung von Werkzeugen, beim Verstehen der Aufgaben aus der Prüfpunktliste und bei der Analyse von manuell gefundenen oder durch Werkzeuge gemeldeten Risiken.
Um 14:45 Uhr wurden die gefundenen Risiken in eine Risikomatrix übertragen. Für jedes Risiko wurden gemeinsam Gegenmaßnahmen erörtert. Für die Ausgabe der Version des Frameworks eines externen Anbieters sollte dieser über den Umstand informiert werden um dies zu unterbinden. Als eine der ersten Schritte bei der Eingabebehandlung sollte die Maximallänge definiert und geprüft werden [217]. Es wurde vom Anleiter die Nutzung einer globalen Maximallänge wie z.B. von 255 Zeichen vorgeschlagen, welche vor alle weiteren Behandlungen und Prüfungen erfolgt. Um den *Denial of Service* über den Produktkatalog zu verhindern, wurde von den Probanden vorgeschlagen die Anzahl der Produkte zu begrenzen. Vom Anleiter wurden zusätzlich die Möglichkeit der Performance-Optimierung eingebracht, so dass bei der Erstellung des Produktkataloges nicht benötigte Daten möglichst frühzeitig aus dem Arbeitsspeicher entfernt werden.
Abgeschlossen wurde die Erstellung der Risikomatrizen um 16:00 Uhr, welche im Anhang G.8.1 auf Seite 137 und Anhang G.8.2 auf Seite 138 abgelegt sind. Es sind von der ersten Gruppe elf und von der zweiten Gruppe zwölf Risiken identifiziert worden, wovon insgesamt neun als hohes Risiko eingestuft worden sind.

3.3.7 Evaluierung der inhomogenen Team-Sicherheitsprüfung

Die Vorteil-Hypothesen 1., 2., 3., 4. und 5. und die Nachteil-Hypothesen 11. und 12. können wie bei der homogenen TSP bestätigt werden. Die Nachteil-Hypothese 10. kann wie in der homogenen TSP nicht bestätigt werden, da Gegenmaßnahmen ergriffen wurden.
Bei der Befragung am Ende der TSP wurde gewünscht, für jedes Werkzeuge ein detaillierte Anleitung zu liefern. Der *OWASP Testing Guide v4* liefert solche Beispiele, weshalb eine separate Dokumentation als redundant angesehen wird. Es haben mehrere Probanden den *OWASP Testing Guide v4* und den *OWASP Code Review Guide 2.0* während der TSP als zusätzliche Anleitung genutzt, trotzdem sollte zum Start der nächsten TSP exemplarisch ein Risiko aus der Prüfpunktliste anhand einem der beiden Anleitungen untersucht werden.
Weiter haben die Probanden angegeben, Kompetenzen der anderen Team-Mitglieder durch die TSP nicht besser zu kennen, da diese bereits vorher bekannt waren. Entsprechend kann Vorteil-Hypothese 7. nicht bestätigt werden.
Weil die beiden TSP aus Zeitgründen im Rahmen der Thesis jeweils an einem Tag mit beiden Teams stattgefunden haben, kann Vorteil-Hypothese 9. nicht bestätigt werden.
Es wurde angemerkt, dass Wissen zur Benutzung der Werkzeuge wie *Zap* zum Teil wieder vergessen wurde. Dies zeigt, dass *Zap* in der täglichen Arbeit nicht genutzt wurde. Aufgrund des für die inhomogenen TSP später geplanten Termins als empfohlen und die zusätzliche Verschiebung war die Zeitspanne zu groß und

Wissen ist bereits vergessen worden. Daraus lässt sich weiter ableiten, dass TSP nur mit einem Team durchgeführt werden sollte, welches bereits kontinuierlich Schulungen im Bereich Sicherheit genießt.
Die aggressive Haltung von dem einen Proband liegt wahrscheinlich an der neuen Umgebung, in welcher dieser mit einer neuen Umgebung konfrontiert wird. Dies bestätigt das soziale Risiko bei den TSP und damit Nachteil-Hypothese 15.
Die Probanden gaben an, durch Besprechung von Anwendungsinterna besser Prüfen zu können. Dadurch konnten Risiken schneller in der Prüfpunktliste bearbeitet werden. Beispielsweise wurde die Authentifizierung in einer Anwendung ohne Verschlüsselung direkt besprochen sowie festgestellt, dass hier bereits ein Zertifikat beantragt ist, so dass keine weiteren unmittelbaren Aktionen notwendig sind. Dies bestätigt Hypothese 8.

3.3.8 Fazit beider Prüfungen und Ausblick

Es fanden vier Probanden die inhomogenen TSP besser als die homogenen TSP und ein Proband anders herum. Bei den homogenen TSP wurde der Reiz, die Anwendung des „Gegner-Teams“ zu prüfen betont. Bei der inhomogenen Prüfung wurde die vorgegebene Prüfpunktliste als starke Verbesserung genannt sowie der erhöhte Austausch untereinander betont. Die signifikante Steigerung von neun Risiken bei der homogenen zu 23 Risiken bei der inhomogenen TSP bestätigt die Bewertung der Probanden.
Es kann die erhöhte Anzahl von gefundenen Risiken bei der inhomogenen TSP nicht eindeutig auf den Wechsel von homogenen zu inhomogenen Teams zurückgeführt werden, da ein Methodenwechsel von Blackbox-Prüfungen auf Whitebox-Prüfungen vorgenommen wurde. Zusätzlich wurde durch eine Prüfpunktliste die Herangehensweise bei der inhomogenen Prüfung stark verbessert. Auch hatten die Probanden bei der inhomogenen TSP mehr Zeit, da nicht auf das Test-System gewartet werden musste.
Die Vorteil-Hypothesen 1., 2., 3., 4., 5. und 8. und die Nachteil-Hypothesen 12., 13., 14. und 15. konnten durch die TSP bestätigt werden. Die Vorteil-Hypothesen 6., 7., und 9. und die Nachteil-Hypothese 10. konnten nicht bestätigt werden, Nachteil-Hypothese 11. konnte teilweise bestätigt werden.
Ist das Ziel der TSP nur die Prüfung der Sicherheit einer Webanwendung, so sollte auf die TSP verzichtet werden. Ein Sicherheits-Experte ist effizienter und Ressourcen günstiger. Beispielsweise hat dieser alle Werkzeuge bereits installiert und kennt sich mit diesen aus. Auch das Wissen im Bereich Sicherheit sollte ausgeprägter sein.
Die 23 Unterstützungen bei der homogenen und 32 Unterstützungen bei der inhomogenen TSP zeigen, dass ein Anleiter mit Wissen im Bereich Webanwendungs-Sicherheit während der TSP zwingend erforderlich ist.
Bei traditionellen Sicherheitsprüfungen wird ein Prüfbericht nach Durchführung vorgelegt. Dieser wird i.d.R. vom Management gelesen und anschließend Anweisungen an die Entwickler und System-Administratoren gegeben. Diese müssen das Risiko verstehen und Maßnahmen konzipieren und implementieren. Im Fall einer TSP versteht der Teilnehmer das Risiko während der TSP und kann es anschließend einfacher beheben.
TSPen können also als Prozess gesehen werden, bei welchem das Thema Sicherheit immer weiter verinnerlicht wird, so die Beobachterin.
Der Abteilungsleiter hat zum Ende der inhomogenen TSP betont, wie gut im die Prüfung gefallen hat.

Zukünftig möchte KielUnt mehr Wissen und Sensibilisierung im Bereich Sicherheit schaffen.
Wären System-Administratoren zu der TSP zugelassen worden, wären wahrscheinlich mehr Prüfungen im Bereich der Systeme erfolgt. Beispielsweise blieb ein in der Schulung vorgestellter Port-Scan aus. Der Ausschluss von System-Administratoren bei der TSP verdeutlicht, dass „Silos“ in Unternehmen nach wie vor stark genutzt werden und das Prinzip der *DevOps* noch nicht bei allen Unternehmen angewendet wird.
Die Einordnung von Risiken in einer Risikomatrix kann als erster Schritt zu einer Risikoanalyse bei jedem Sprint angesehen werden.
Für zukünftige TSPen sollte die Prüfpunktliste in Deutsch übersetzt werden.
Um so größer eine Organisation, um so weniger kennen Mitarbeiter die Kompetenzen von anderen Mitarbeitern. Hypothese 7 sollte in einer großen Organisation mit mehrere Teams durchgeführt werden, um zu bestätigen, dass Teilnehmer durch TSPen die Kompetenzen von anderen Teilnehmern kennen lernen.
Zukünftig sollte vor den TSPen mehr Werbung gemacht werden, da nur fünf der acht Mitarbeiter vollständig und zwei teilweise an den TSPen teilgenommen haben.

Kapitel 4

DevOps-Technologien

In diesem Kapitel werden zunächst Ansätze zur Härtung der Infrastruktur vorgestellt um anschließend das für *DevOps* besonders wichtige Erzeugungs- und Verteilungssystem zu härten. Danach wird erläutert, wie durch Erfassung und Auswertung von Messpunkten und Protokollen ein „Blindflug“ vor und nach der Verteilung auf eine Produktionsumgebung verhindert werden kann.

4.1 Härtung der *DevOps*-Infrastruktur

Bei *DevOps* wird die Infrastruktur möglichst automatisiert, damit „auf Knopfdruck“ neue Umgebungen erzeugt werden können. Dazu muss die Infrastruktur versioniert, die Architektur der Anwendung möglichst als getrennte Microservices vorliegen und die einzelnen Anwendungen virtualisiert werden.

4.1.1 *Infrastructure as Code*

Bei *Infrastructure as Code* wird die Idee eines Software-definierten Rechenzentrums aufgegriffen. Es wird die Konfiguration aller Systeme versioniert. Dadurch entsteht erhöhte Flexibilität und Schnelligkeit beim Aufsetzen von Systemen [127]. Werkzeuge um eine Provisionierung von Systemen durchführen zu können sind z.B. *Puppet* [30] und *Chef [4]*. Dadurch lassen sind insbesondere große Serverfarmen einfach pflegen. In Kombination mit Virtualisierung können Test-Systeme vergleichsweise schnell hoch- und runter gefahren werden. Beispielsweise um eine Umgebung zur Durchführung von Tests durch Kunden bereitzustellen oder *War Games* zu veranstalten.

4.1.2 *Microservice*-Architektur

Bei der *Microservice*-Architektur werden fachliche Funktionen aufgeteilt und jeweils als Dienst angeboten [91, S. 1]. *Microservices* können einzeln geprüft werden und reduzieren dadurch die Testzeit. Durch Reduktion der Komplexität und Erhöhung der Testbarkeit wird die Wahrscheinlichkeit von Schwachstellen reduziert. So können im Rahmen von Integrationstests Eingaben, z.B. über eine *REST*-Schnittstelle, validiert werden. Mittels Akzeptanztest können Benutzereingaben in die Webanwendung, welche an den *Microservice* übergeben werden, getestet werden.

4.1.3 Virtualisierung

Bei der Virtualisierung können Desktop- als auch Server-Systeme virtualisiert werden. Die virtuellen Systeme operieren dabei wie physikalische Server und können bei Providern häufig gemietet werden, beispielsweise bei Amazon EC2 [56] oder DreamHost [105].
Isolierung und Ressourcenkontrolle von Prozessen erhöht die Sicherheit eines Systems. Durch Isolierung kann ein Prozess nicht einen anderen Prozessen auf dem gleichem System beeinflussen. Durch Ressourcenkontrolle werden einem Prozess definierte Ressourcen zur Verfügung gestellt. Ressourcen sind beispielsweise die Anzahl der Prozessorkerne, der verfügbare Arbeitsspeicher oder auch Zugriff auf Ordner im Dateisystem. Dadurch ist sichergestellt, dass ein Prozess nicht mehr Ressourcen als zugewiesen beansprucht. Isolierung und Ressourcenkontrolle wird traditionell via Virtualisierung geschaffen. Eine Alternative dazu ist die Container basierte Virtualisierung.

4.1.4 Container-basierte Virtualisierung

Container basierte Virtualisierung gilt als leichtgewichtig, da diese den Betriebssystemkern des Host-Systems nutzt. Die generelle Architektur ist in Abb. 4.1 aufgezeigt. Hier wird deutlich, dass in einem Container nur die zu isolierende Anwendung und seine Abhängigkeiten vorhanden sein sollten. Die Container-Umgebung setzt hier auf der Betriebssystem-Ebene auf. Container sollten statuslos sein, damit diese ausgewechselt werden können [167, S. 67]. Nur die Nutzdaten sollten persistent außerhalb des Containers gespeichert werden.

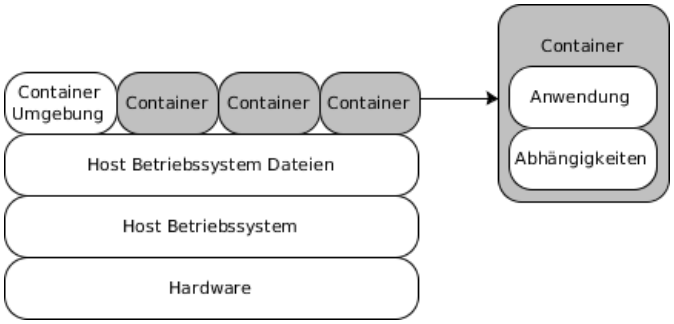


Abbildung 4.1: Architektur Container-basierter Virtualisierung

Durch Nutzung von Namensräumen (Englisch *Namespaces*) und Kontrollgruppen (Englisch *Control Groups)* des Betriebssystemkerns können unterschiedliche isolierte Umgebungen für Anwendungen geschaffen werden. Beispiele für Container-basierte Virtualisierung sind *Docker* [104], *Linux Container* [47] und *OpenVZ* [48]. Ein Container kann Betriebssystem-Funktionalitäten wie *init*, *sshd*, *cron*, *logd* usw. beinhalten. Kommunikation zwischen Containern oder zwischen dem Container und dem Host sind so performant wie normale *Linux Interprozess Kommunikation* (Englisch *Inter-Process Communication*, kurz *IPC*).
Mittels Namensräumen, welche durch den *Linux* System-Aufruf `setns()` gesetzt werden können, isolieren sich Prozesse in *Linux*.
Mittels Kontrollgruppen können Prozesse gruppiert und die Ressourcen limitiert werden. Dabei werden i.d.R. Prozessorzeit und Arbeitsspeicher limitiert. Bei Limitierung der Ressourcen ist zu beachten, dass ein Container zusätzlich zu den zugewiesenen Ressourcen auch nicht zugewiesene Ressourcen sehen kann. Beispielsweise kann ein Prozess in einem Container alle vorhandenen Prozessorkerne sehen. Beim Ver-

such sich selbst zu optimieren ist es möglich, dass ein Prozess im Container auf einen nicht erlaubten Prozessorkern zugreifen möchte. Der Quellcode für die Kontrollgruppen ist dabei den entsprechenden Funktionalitäten zugeordnet, Beispiele sind im Anhang F auf Seite 130 aufgeführt.
Bei der Erstellung eines Containers werden *Linux*-Betriebssystemkern Namensräume für die Isolation von Prozessen genutzt, die Wichtigsten sind:

- *ipc* Namensraum: Genutzt für Zugriffsmanagement von *IPC*-Ressourcen [209, S. 13].
- *net* Namensraum: Genutzt für Netzwerkmanagement auf ISO/OSI Ebene 2 und 3, beispielsweise für die Erstellung der virtuellen Ethernet-Netzwerkschnittstellen *veth* (ISO/OSI-Ebene 2).
- *mnt* Namensraum: Genutzt für das Mangement von Bereitstellungspunkten (Englisch Mount-Point) und Dateisystemen zwischen Containern und dem Host. Der `chroot()` System Aufruf nutzt beispielsweise den *mnt* Namensraum. [209, S. 10]
- *pid* Namensraum: Genutzt für Prozessisolation, so dass Prozesse gruppiert und die Sichtbarkeit (beispielsweise das *proc*-Dateisystem) als auch die Interaktionsmöglichkeiten (durch Senden von Signalen) eingeschränkt werden können. [209, S. 9]
- *uts* Namensraum: Das *Unix Timesharing System* wird genutzt für die Isolation von Betriebssystemkern- und Versionidentifikatoren, beispielsweise für den System Aufruf `sethostname()`.

Container können beispielsweise die Dateisysteme *B-tree file system* (Btrfs) oder *advanced multi layered unification filesystem* (*Aufs*) nutzen. Beide sind „Kopieren-beim-Schreiben"-Dateisysteme (Englisch *copy-on-write*).
Das Netzwerk kann i.d.R. als virtuelle Netzwerkschnittstelle (*vethX*) oder als Netzwerkbrücke betrieben werden. Die Softwarefirewall *iptables* [145] kann zusätzlich zum Schutz von Containern eingesetzt werden. Beispielsweise um einen Dienst, welcher auf einem Port auf dem Host-System lauscht, nur einer definierten Internetprotokoll-Adresse zugänglich zu machen.
Eine Schwachstelle von Containern sind System-Aufrufe, welche nicht den *Namespace* beachten. Als Lösung können die System-Aufrufe einer Anwendung limitiert werden. Dafür wird die Anwendung, beispielsweise mittels dem *Linux*/*Unix*-Werkzeug *strace* [168], auditiert und die benutzten System-Aufrufe einer Positivliste hinzugefügt. Die System-Aufrufe der Positivliste können mittels *seccomp* [57] für einen Prozess limitiert werden. Dabei gilt zu beachten, dass Anwendungen heutzutage mehrmals die Woche mit neuen Features verteilt werden, so dass Audits mit jeder Veröffentlichung einer neuen Version erfolgen sollten. Wird ein nicht in der Positivliste enthaltener System Aufruf durch eine neue Software Version benötigt und dieser ist nicht in der Positivliste enthalten führt dies zu Inkompatibilitäten.
Betriebssystemkern-*Capabiltities* und *NSA Security-Enhanced Linux* (kurz *SELinux*) sind Implementierungen im *Linux*-Betriebssystemkern um feingranulare Zugriffskontrolle zu gewährleisten. Auf Standard-Linux System hat beispielsweise nur der privilegierte *root*-Benutzer das Recht sich an Ports kleiner 1024 zu binden. Mittels Betriebssystemkern-*Capabilities* oder *SELiunx* kann das Recht zum Binden an einen Port kleiner 1024 (z.B. Port 80 für einen Webserver) für einen Prozess eingeräumt werden, ohne das dieser als privilegierter Benutzer laufen muss.
Dies kann beispielsweise über *Type Enforment* und Rollen-basierter Zugriffskontrolle gewährleistet werden. Während bei der Rollen-basierten Zugriffskontrolle Ressourcen geschützt werden, wird in Kom-

bination mit *Type Enforment* ein Typ für eine Gruppe von Ressourcen definiert. Diese Kombination wird von *SELinux* genutzt. So kann beispielsweise der Gruppe von Dateien im Ordner */var/www/* der Typ *webroot_t* zugewiesen werden. Anschließend kann der Webserver-Prozess das Recht zum Verändern von Dateien des Typs *webroot_t* zugewiesen werden. Während im Linux-Umfeld das Recht zum Lesen, Schreiben sowie Ausführen an Benutzer gebunden werden kann, bietet *SELinux* die Möglichkeit Rechte feingranularer festzulegen [151, S. 291]. Dabei können beispielsweise *getattr*-Operationen, wie das Auslesen der Dateisystem-Attribute Besitzer, Zeit der letzten Modifizierung usw. weiter eingeschränkt werden.

4.1.5 *Hypervisor*-basierte Virtualisierung

Bei Container-basierter Virtualisierung werden die Container auf einem „Host“ ausgeführt. Bei *Hypervisor*-basierter Virtualisierung werden die virtuellen Maschinen, auch Gastsysteme genannt, auf einem „Wirtsystem“ ausgeführt.
Durch *Hypervisor*-basierte Virtualisierung wird die Hardwareebene emuliert. Dadurch können physische Bestandteile eines Rechners der virtuellen Maschine vorgetäuscht werden. Es kann ein komplettes Betriebssystem in einer virtuellen Maschine gestartet werden. Dies ermöglicht auch heterogene Virtualisierung unterschiedlicher Betriebssysteme. So kann auf dem Wirtsystem *Linux* in einer virtuelle Maschine *Microsoft Windows* virtualisiert werden. Da virtuelle Maschinen komplett virtualisiert sind, können nicht die Ressourcen (z.B. den Betriebssystemkern) des Wirtsystems genutzt werden.
Es ist evaluiert, dass Virtualisierung i.d.R. mehr Ressourcen benötigt als Container basierte Isolation [112, 251]. *VMWare ESX* [38], *Xen* [64] und *Kernel Virtual Machine* (kurz *KVM*) [149] sind Beispiele für *Hypervisor*-basierte Virtualisierung.

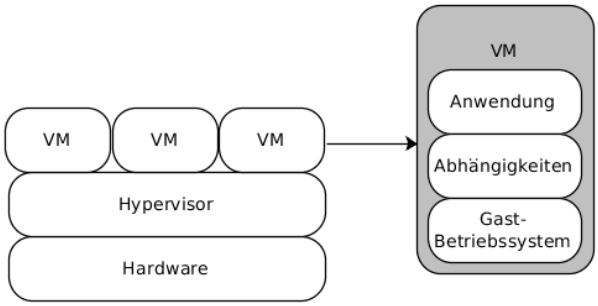


Abbildung 4.2: Architektur Hypervisor-basierter Virtualisierung

Es wird zwischen Typ-1 und Typ-2 *Hypervisor* unterschieden. Während bei Typ-1 die Hardware die Virtualisierung übernimmt, ist bei Typ-2 das Betriebssystem vom Wirt für die Virtualisierung zuständig. Entsprechend ist Typ-1 Virtualisierung zu bevorzugen, welche beispielsweise durch Intel VT-x/AMD-x ermöglicht und durch *KVM* genutzt wird [253]. *KVM* wird zwar i.d.R. auf einem bestehendem Betriebssystem installiert, nutzt aber direkt die Hardware und ist damit vom Typ-1 [120].
Virtuelle Maschinen besitzen eine definierte Anzahl an virtuellen Prozessorkernen sowie eine definierte Anzahl an virtuellem Arbeitsspeicher. Zusätzlich zu einer definierten Größe kann ein Maximalwert für Ressourcen wie Prozessorkerne und Arbeitsspeicher angegeben werden. Dabei kann virtueller Arbeitsspeicher nicht mehr Seiten enthalten als der reale Arbeitsspeicher bietet.
Virtuelle Maschinen besitzen zwei *Scheduler*, einen im Wirtsystem und einen in der virtuellen Maschi-

ne selbst. Um dies zu umgehen, können physische Prozessorkerne an virtuellen Prozessorkerne gepinnt werden [112, 43].
Eine virtuelle Maschine kommuniziert über emulierte Geräte und *hypercalls*, welche durch den *Hypervisor* kontrolliert werden, mit dem Wirtsystem beziehungsweise mit der Außenwelt. Deshalb ist es für einen Angreifer besonders schwer aus der virtuellen Maschine auszubrechen. Auch wenn vereinzelt Schwachstellen zur Rechteausweitung bekannt geben wurden, können virtuelle Maschinen als sicherer als Container basierte Virtualisierung angesehen werden.

4.1.6 *Docker*

Die Container basierte Virtualisierung ist schwierig zu konfigurieren. *Docker* bietet vorkonfigurierte Isolation von Prozessen durch Arbeitsbereiche (Container). Dafür werden alle für einen Prozess benötigten Abhängigkeiten definiert und in einem *Container* abgelegt. Während bei der Virtualisierung ein Betriebssystem mit virtuellen Geräten emuliert wird, benutzt *Docker* die Ressourcen vom Betriebssystemkern und verbraucht deshalb weniger Ressourcen vom Host im Gegensatz zur Virtualisierung. Häufig genannte Vorteile sind u.a. die Skalierbarkeit und die stabile Architektur [167, S. 9]. Bei der Automatisierung der Infrastruktur, insbesondere von dem Erzeugungs- und Verteilungsprozess kann *Docker* empfohlen werden, da eine isolierte Anwendung hier schneller hochfährt als ein komplettes Betriebssystem.

Komponenten

Docker nutzt eine Klient-Server Architektur, dargestellt in Abb. 4.3. Der *Docker* Klient kommuniziert mit dem *Docker* Daemon, welcher für das Erzeugen (*build*), Abholen (*pull*) und Ausführen (*run*) von Containern zuständig ist. Wobei die Erzeugung eines Containers und die Ausführung auf unterschiedlichen Hosts stattfinden können.

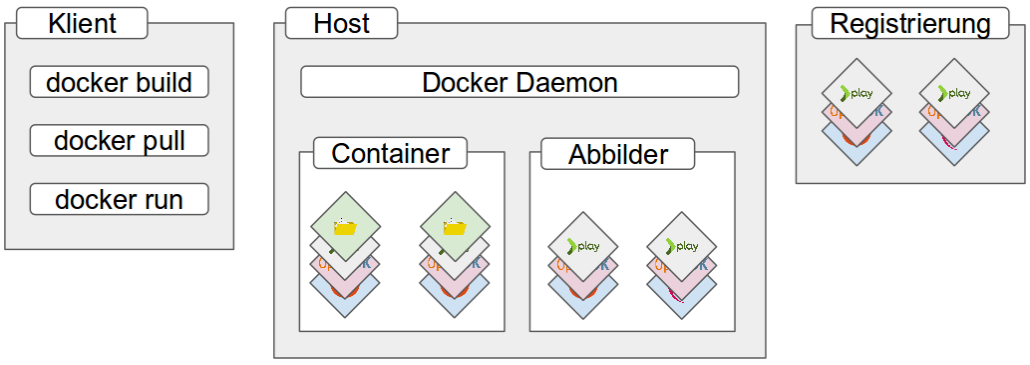


Abbildung 4.3: *Docker* Komponenten
Bildquellen der Symbole: [87, 184, 239, 243]

Ein *Docker* Abbild (Englisch *Image*) ist eine schreibgeschützte Vorlage für einen Container. Hat der *Docker* Daemon ein Abbild nicht vorrätig, kann er dieses bei einer Registrierung (Englisch *Registry*) anfordern, welche das Abbild, sofern vorhanden, an den *Docker* Daemon ausliefert. Registrierungen sind öffentliche oder private Speicher von Abbildern. Ein Abbild kann beispielsweise Systemwerkzeuge von dem Betriebssystem Ubuntu sowie einen Webserver und eine Webanwendung enthalten.
Ein Beispiel für den Lebenszyklus eines Containers ist in Abb. 4.4 dargestellt. Dabei werden alle Befehle vom Klienten erteilt. Mittels dem Befehl `docker build` wird ein Abbild auf Host 1 nach den Spezifika-

tionen aus der Datei *Dockerfile* erzeugt (Punkt 1 und 2).

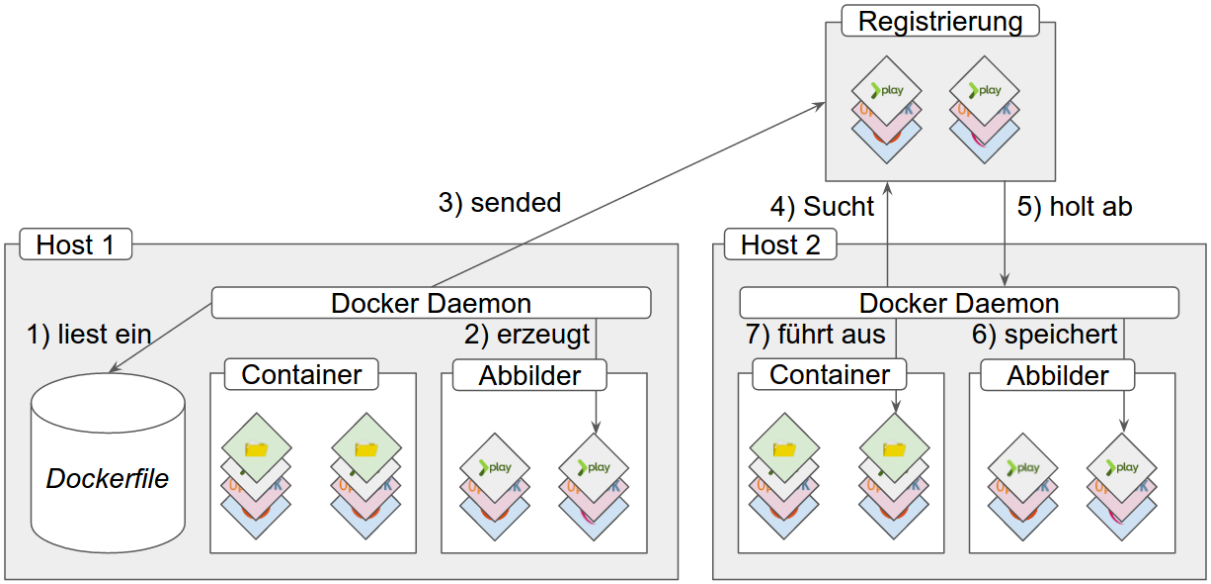


Abbildung 4.4: Steuerung von dem *Docker* Daemon und der Registrierung durch einen Klienten
Bildquellen der Symbole: [87, 184, 239, 243]

Das erzeugte Abbild kann an die Registrierung via `docker push` gesendet werden (Punkt 3). Nachdem ein Abbild gesendet wurde, kann dieses mittels `docker search` bei der Registrierung gesucht (Punkt 4) und mittels `docker pull` von Host 2 abgeholt (Punkt 5) und lokal gespeichert (Punkt 6) werden. Durch `docker run` wird ein Abbild als Container ausgeführt (Punkt 7), hier können zusätzlich Parameter, beispielsweise um Port-Weiterleitungen zu definieren, angegeben werden.

Dateisystem

Docker verwendet i.d.R. das Ebenen basierte Dateisystem *Aufs*. Ein Beispiel ist in 4.5 dargestellt. Auf der

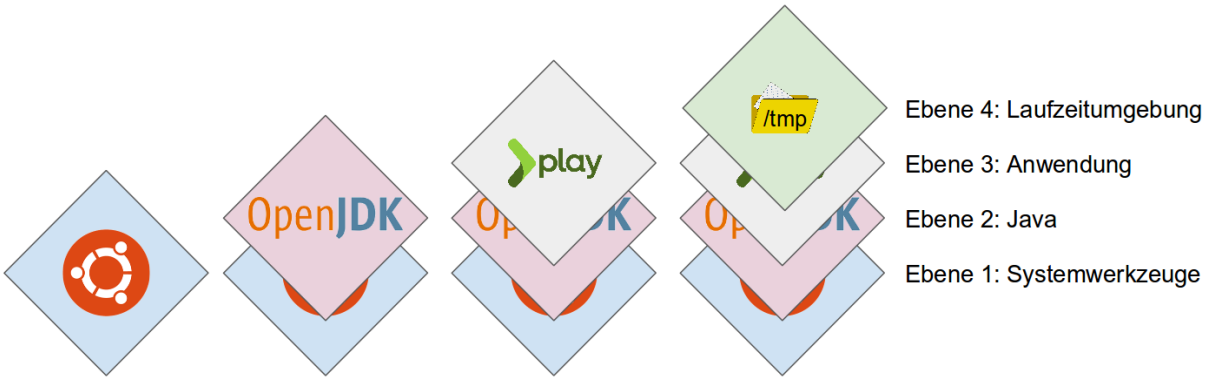


Abbildung 4.5: *Docker* Dateisystem-Ebenen
Bildquellen der Symbole: [87, 184, 239, 243]

untersten Ebene sind Betriebssystemwerkzeuge, wie *chmod* zum Verändern von Dateirechten, enthalten. Auf der zweiten Ebene wird Java installiert, welches von der Anwendung der dritten Ebene benötigt wird. Sobald der Container gestartet wird, wird eine Ebene für die Laufzeitumgebung geschaffen, um beispielsweise in dem Order */tmp* temporäre Dateien zwischenzuspeichern. Durch die Ebenen ist es möglich, dass eine andere *Java*-Anwendung das *Java*-Abbild der zweiten Ebene nutzt, ohne dieses Abbild neu erzeu-

gen zu müssen oder die bestehende Java-Anwendung der dritten Ebene beeinflusst. Zur Erzeugung eines Abbilds wird der Befehl `docker build -t <Name>:<Tag> .` im Ordner mit dem *Dockerfile* ausgeführt. Das offizielle *Dockerfile* für Ubuntu 16.04 ist wie folgt definiert:

```
1 FROM scratch
2 ADD ubuntu-xenial-core-cloudimg-amd64-root.tar.gz /
3
4 # a few minor docker-specific tweaks
5 # see https://github.com/docker/docker/blob/master/contrib/mkimage/
    debootstrap
6 RUN echo '#!/bin/sh' > /usr/sbin/policy-rc.d \
7         && echo 'exit 101' >> /usr/sbin/policy-rc.d \
8         && chmod +x /usr/sbin/policy-rc.d \
9         [...]
10
11 # enable the universe
12 RUN sed -i 's/^#\s*\(deb.*universe\)$/\1/g' /etc/apt/sources.list
13
14 # overwrite this with 'CMD []' in a dependent Dockerfile
15 CMD ["/bin/bash"]
```

Listing 4.1: Dockerfile zur Erzeugung von

Quelle: [121]

Das Protokoll der Erstellung bei Ausführung des Befehls `docker build -t ubuntu:16.04 .` ist im Anhang unter A auf Seite 110 aufgeführt.
Bei der Übergabe von Arbeitsverzeichnissen sollte darauf geachtet werden, dass diese durch Prozesse im Container ggf. bearbeitet werden. Entsprechend sollte z.B. die Datei */etc/localtime*, welche als Arbeitsverzeichnis eingebunden wird um die Uhrzeit des Hosts im Container zu nutzen, nur mit Lese-Zugriff ausgestattet sein.

Datei- und Prozessrechte von Benutzern

Es sollten generell nur Ordner und Dateien an einen *Docker*-Container gebunden werden, die benötigt werden. Benutzer, welcher der Betriebssystemgruppe *docker* hinzugefügt werden, sollten wie privilegierte Benutzer behandelt werden. Mittels *Docker* kann privilegierter Zugriff erlangt werden, entsprechend folgt ein Beispiel in Anlehnung an [181] zur Verdeutlichung. Das in Listing 4.1 gezeigte *Dockerfile* definiert das Arbeitsverzeichnis */stuff*. Über den *Docker*-Befehl `docker run -v $PWD:/stuff -t docker-image`

Listing 4.1 *Dockerfile* mit Arbeitsverzeichnis [181]

```
1 FROM ubuntu:latest
2
3 ENV WORKDIR /stuff
4 RUN mkdir -p $WORKDIR
5 VOLUME [ $WORKDIR ]
6 WORKDIR $WORKDIR
```

`/bin/sh -c 'cp /bin/cat /stuff && chmod a+s /stuff/cat'` wird ein Container mit dem aktuellen Datei-Pfad gebunden an das Arbeitsverzeichnis */stuff* und gestartet. Nach Start des Containers wird eine Binärdatei, also in diesem Fall */bin/cat* in das Arbeitsverzeichnis kopiert und das *suid*-Dateirecht gesetzt.

Dadurch wird die Binärdatei bei Ausführung mit den Rechten des Besitzers, in diesem Fall der privilegierte Benutzer *root*, und nicht mit den Rechten des Ausführers gestartet. Anschließend wird auf dem Host der Befehl `./cat /etc/shadow` ausgeführt, welcher die Ausgabe der Passwort-Datei */etc/shadow* zur Folge hat. Es können aber auch andere beliebige Binärdateien im Container kopiert und auf dem Host mit privilegierten Rechten ausgeführt werden.

Es sollte in einem *Dockerfile* immer ein Benutzer zur Ausführung angelegt und für die Ausführung genutzt werden. Nachfolgend ist ein Beispiel gezeigt, in welchem der Benutzer *example* angelegt und anschließend bei der Ausführung genutzt wird:

Listing 4.2 *Dockerfile* mit Arbeitsverzeichnis und Benutzer in Anlehnung an [181]

```
1 FROM ubuntu:latest
2
3 ENV WORKDIR /stuff
4 RUN mkdir -p $WORKDIR
5 VOLUME [ $WORKDIR ]
6 WORKDIR $WORKDIR
7
8 RUN adduser --system --no-create-home --disabled-password --disabled-
    login --shell /bin/sh example
9 USER example
```

Wird der Befehl `docker run -v $PWD:/stuff -t docker-image /bin/sh -c 'cp /bin/cat /stuff && chmod a+s /stuff/cat'` für einen Container auf Basis von Listing 4.2 genutzt und anschließend versucht die Datei */etc/shadow* auf dem Host auszulesen, wird die Fehlermeldung *./cat: /etc/shadow: Permission denied* ausgegeben.

In Listing 4.3 wird die Prozessliste auf dem Host nach Start einer *MySQL*-Datenbank gezeigt, wobei im *Dockerfile* der Benutzer *mysql* anlegt wird. Hier läuft der Prozess auf dem Host unter der Benutzer-Idenfikationsnummer 999, während in dem Container (siehe Listing 4.4) die *MySQL*-Datenbank unter dem Benutzer *mysql* mit der Benutzer-Idenfikationsnummer 33 läuft. Alle im Container erzeugten Dateien gehören auf dem Host dem Benutzer mit Idenfikationsnummer 33. Entsprechend ist bei der Erstellung eines Benutzers im *Dockerfile* die Benutzer-Idenfikationsnummer mit dem Container startendem Benutzer auf dem Host abzustimmen.

Listing 4.3 Prozessliste auf dem Host

```
1 UID          PID  PPID  C STIME TTY          TIME CMD
2 999         4236  4098  0 08:37 ?        00:00:00 mysqld
3 [...]
```

Listing 4.4 Prozessliste in dem *Docker*-Container

```
1 UID          PID  PPID  C STIME TTY          TIME CMD
2 mysql          1     0  0 07:37 ?        00:00:00 mysqld
3 root          40     0  0 07:50 ?        00:00:00 bash
4 [...]
```

Netzwerksicherheit

Sobald der *Docker* Daemon gestartet wird, erzeugt er die virtuelle Netzwerkbrücke *docker0* auf dem Host selbst und sucht eine freies privates Subnetz nach RFC-1918. Sobald ein Container gestartet wird, wird in der Standardkonfiguration eine virtuelle Schnittstelle auf dem Host erzeugt, z.B. *veth33a64a1*. Diese Schnittstelle wird mit der Schnittstelle *eth0* im Container verbunden. Um *Firewall*-Regeln mittels *iptables* zu konfigurieren, wird die *iptables-chain DOCKER* erstellt. Wird ein Container mit der Option *-p <host-port>:<container-port>* gestartet, wird eine entsprechende *iptables*-Regel innerhalb der *DOCKER-chain* erzeugt. Optional kann die Schnittstelle auf dem Host mittels *-p <host-interface><host-port>:<container-port>*, also beispielsweise *127.0.0.1:80:9000* um den Port 9000 vom Container an Port 80 der Schnittstelle *localhost* des Hosts zu binden.

Patch-Management

Da Container keine Anwendungsdaten beinhalten, können diese zur Aktualisierung der Anwendung in einem Container mit einem neuem Abbild gestartet werden. Beispielsweise kann eine MySQL-Datenbank durch folgende Befehle aktualisiert werden:

```
1 docker pull mysql
2 docker stop my-mysql-container
3 docker rm my-mysql-container
4 docker run --name=my-mysql-container [...] mysql
```

Erfolgt die Aktualisierung beispielsweise mittels `apt-get` innerhalb des Containers, wäre dieser nicht mehr unveränderlich. Entsprechend ist davon abzusehen. *Docker* liefert keine automatischen Informationen über Aktualisierungen eines Basis-Abbilds. Eine Möglichkeit stellt die Abfrage aller Abbilder und Ausführung von `docker pull <image>` dar. So kann die Ausgabe auf „Downloaded newer image“ geprüft werden und eine Information, z.B. via E-Mail versendet werden.

4.2 Härtung des Erzeugungs- und Verteilungssystem

In diesem Abschnitt wird ein typisches Erzeugungs- und Verteilungssystem vorgestellt. Anschließend werden durch eine Bedrohungsanalyse die Anforderungen an die Validierung eines Abbilds festgelegt um danach eine gehärtete Architektur für das Erzeugungs- und Verteilungssystem darzulegen.
Hauptziel dieses Abschnitts ist den Erzeugungs- und Verteilungsprozess glaubhaft zu machen. Die Erzeugung eines Artefakts erfolgt nach einer definierten sortierten Abfolge von Schritten, in welchen Aktionen ausgeführt werden. Kontinuierliche Verteilung *(*Englisch *continuous delivery*) ist ein Vorgehen um den Softwarelieferprozess zu verbessern und wird umgesetzt mit Methoden wie kontinuierlicher Integration (Englisch *continuous integration*, kurz *CI*) und Testautomatisierung. Während bei kontinuierlicher Integration die Überführung eines erzeugten Abbilds in die Produktionsumgebung manuell erfolgt, wird bei kontinuierlicher Verteilung dieser Schritt ebenfalls automatisiert [200]. Entsprechend erhält der Entwickler erhöhte Verantwortlichkeit im Entwicklungsprozess und es werden mehr Tests durchgeführt, da mehr Vertrauen in die Anwendung benötigt wird.

Eine Verteilung beschreibt Aktivitäten zur Installation von Software. Dabei muss beachtet werden, dass moderne Teams agil arbeiten und in hoher Frequenz neue Software-Bausteine entwickeln.

4.2.1 Vorstellung eines gängigen Erzeugungs- und Verteilungsprozesses

Abb. 4.6 zeigt eine gängige Architektur für die Erzeugung und Verteilung von Webanwendungen. Der

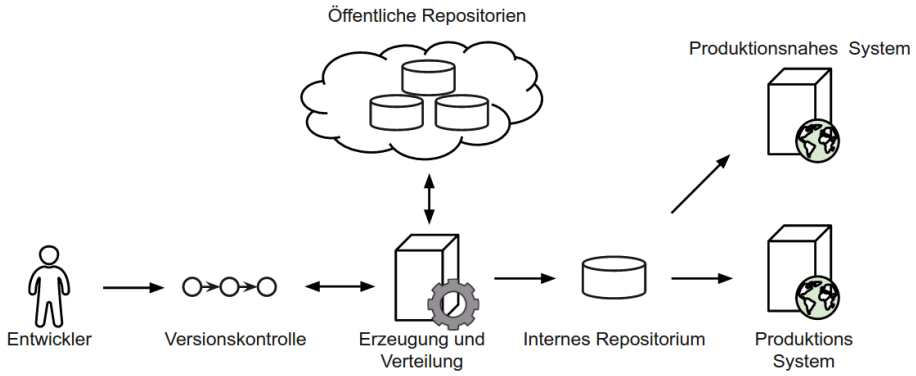


Abbildung 4.6: Beispiel einer Erzeugungs- und Verteilungs-Architektur in Anlehnung an [153]

Erzeugungs- und Verteilungsprozess startet mit einem Entwickler, welcher Quellcode in das Versionskontrollsystem (Englisch *Version Control System*, kurz VCS) überführt. Dieses informiert automatisch den Erzeugungs- und Verteilungsserver (*CI*-Server). Der *CI*-Server holt sich den neuen Quellcode aus der Versionskontrolle und startet mit dem Erzeugungsprozess. Hier wird der interne Quellcode mit verschiedenen öffentlichen Bibliotheken gemischt. Das resultierende Artefakt wird in ein internes Repositorium überführt. Ein Artefakt kann beispielsweise ein Archiv wie eine *.jar*-Datei oder eine Betriebssystem-Paketmanagement-Datei wie eine *.deb*-Datei oder eine *.rpm*-Datei sein. Anschließend wird das Artefakt in eine virtuelle Umgebung, z.B. als *Docker*-Abbild gelegt. Da Artefakte und Abbilder versioniert sein sollten, wird das Abbild in ein Repositorium überführt, welches für die Bereitstellung und die Versionierung des Abbilds zuständig ist.
Die möglichst produktionsnahe Umgebung für Tests holt sich das Abbild vom internen Repositorium ab. Sind alle Tests erfolgreich und eine Freigabe ist erteilt, wird das gleiche Abbild auf die Produktionsumgebung verteilt. Die Freigabe kann manuell oder automatisch erfolgen. Zusätzlich können Nutzer des Produktionssystems (häufig Kunden) auch Zugang zu einem produktionsnahem System für Tests erhalten und eine Freigabe nach Test-Abschluss erteilen.
Bei mehreren Produktionssystemen kann die Verteilung zunächst auf einen kleinen Teil der Systeme erfolgen um Auswirken mittels Metriken zu analysieren. Werden keine Anomalien festgestellt, kann die Verteilung auf alle Systeme erfolgen. Herausforderung sind dabei häufig Datenbanken, da diese i.d.R. komplett umgestellt werden müssen.
Defekte bei der Erzeugung und Verteilung lassen sich in drei Kategorien einteilen. Das verteilte Abbild ist nicht valide. Dazu kann es kommen, wenn die Erzeugung nicht der einem definiertem Erzeugungsprozess folgt oder das zu verteilende Abbild nicht das gleiche ist, welche verteilt wird. Weiter kann ein Abbild erzeugt werden, welches jedoch nicht durch alle im Erzeugungs- und Verteilungsprozess definierten Schritte geht. Dazu kann es kommen, wenn ein Abbild für Benutzer zur Verfügung steht, ohne alle Tests durchlaufen zu haben. Ist Zugriff auf eine Produktionsumgebung durch eine andere Umgebung möglich, handelt

es sich um die dritte Kategorie. Dies tritt auf, wenn eine produktionsfremde Umgebung wie eine Entwicklungsumgebung oder eine Testumgebung direkten unbeabsichtigten Zugriff auf das Produktionssystem erhält. [65]

4.2.2 Sicherung der Integrität des Erzeugungs- und Verteilungsprozesses

Interviews mit Experten wie Björn Kimminich haben ergeben, dass das *CI*-System einen sehr hohen Schutzbedarf hat (siehe Anhang B.1 auf Seite 112). Wird das *CI*-System erfolgreich angriffen, können alle nachfolgendem Systeme kompromittiert werden, auch das Produktionssystem und der Browser von Endbenutzern der Webanwendung.
CI-Systeme wie *Jenkins* [19] haben eine monolithisch Architektur, sobald ein Schritt im Erzeugungs- und Verteilungsprozess kompromittiert ist, können alle Schritte jedes Erzeugungs- und Verteilungsprozesses auf dem System kompromittiert werden. Angriffe werden nach Bass et al. [65] drei Kategorien zugeordnet:

1. Angriff auf eine Komponente des Erzeugungs- und Verteilungsprozess von außerhalb der Umgebung. Dabei kann der Angreifer den Prozess übernehmen und anschließend versuchen Privilegien zu erhöhen.
2. Indirekte Angriffe auf eine Komponente des Erzeugungs- und Verteilungsprozess ohne direkten Netzwerkzugriff. Durch Manipulation von Software-Bibliotheken oder Betriebssystem-Abbildern, welche von öffentlichen Repositorien abgeholt werden, kann infizierter Quellcode in den Erzeugungsprozess eingeschleust werden [174]. Dabei kann z.B. eine Hintertür eingeschleust werden, welche auf dem Produktionssystem genutzt wird.
3. Angriffe auf Netzwerkpunkte um die Verbindung zwischen dem *CI*-System und öffentlichen Repositorien oder zwischen dem *CI*-System und internen Systemen wie dem internem Repositorium oder dem Produktionssystem.

Nach Bass et al. [65] ergeben sich drei Ziele zur Sicherung der Integrität bei der Erzeugung und Verteilung von Artefakten und Abbildern:

1. Die Reihenfolge von Aufträgen kann nur von autorisierten Personen geändert werden.
2. Aufträge können nur durch autorisierte Personen geändert werden.
3. Das Ergebnis eines Auftrags ist genau so wie beschrieben.

Damit eine gehärtete Erzeugungs- und Verteilungs-Architektur entwickelt werden kann, muss zunächst ein Erzeugungs- und Verteilungsprozess definiert werden, welcher in Abb. 4.7 dargestellt ist. Durch einen definierten Erzeugungsprozess kann das versehentliche Löschen von Daten vermieden werden. Als erstes wird der Quellcode aus dem Versionskontrollsystem abgeholt. Die Absicherung des Versionskontrollsystems wird hier nicht weiter betrachtet, wohl aber die Auswirkung bei unbefugten Zugriff auf das Versionskontrollsystem. Neben der Möglichkeit eigenen Quellcode zu manipulieren, kann in dem Versionskontrollsystem *Apache™ Subversion®* (kurz SVN) [51] über die Eigenschaft `svn:externals` definiert werden, automatisch Quellcode von externen SVN-Repositorien nachzuladen [94, S. 81].
Anschließend werden Bibliotheken aus externen Repositorien abgeholt. Da Bibliotheken aus externen Repositorien i.d.R. nicht signiert sind, kann die Echtheit nicht nachgewiesen werden [153].

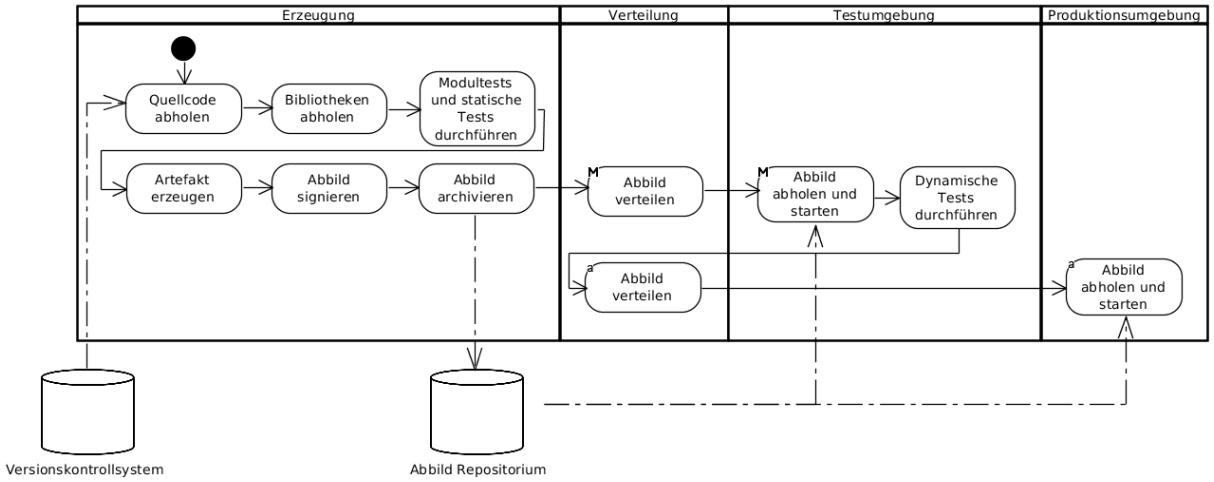


Abbildung 4.7: Schritte eines Erzeugungs- und Verteilungsprozesses in Anlehnung an [65]

Bevor das endgültige Artefakt erzeugt wird, können Modultests und statische Tests durchgeführt werden, pro Testart in einer virtuellen Umgebung.
Anschließend wird das Abbild signiert und im Repositorium archiviert. Die Verteilung steuert in der Zielumgebung das Abholen und das Starten des Abbilds. Auf der Testumgebung werden zusätzlich dynamische Tests durchgeführt.
Bei der Verteilung gilt zu beachten, dass das selbe Artefakt auf die Testumgebung und auf die Produktionsumgebung verteilt wird [231, S. 89].
Die Orchestrierung instrumentiert alle Schritte, d.h. startet einen Schritt und überwacht das Ergebnis um den nächsten Schritt zu starten oder die Erzeugung beziehungsweise Verteilung zu stoppen.
Aus den einzelnen Schritten ergibt sich Tabelle 4.1 auf Seite 55, welche Maßnahmen zur Sicherung der Integrität darlegt.
Zu jedem Schritt werden benötigte Zugriffsrechte und Netzwerkzugriffe aufgezeigt. Weiterhin wird die Komplexität (Komp.), Kritikalität (Krit.) und Auswirkung aufgeführt. Zwar ist der Erzeugungs- und Verteilungsprozess noch nicht sicher, aber stark gehärtet da sich die einzelnen Schritte beziehungsweise Komponenten durch verminderte Privilegien und virtuelle Umgebungen sich nicht gegenseitig beeinflussen können.
Im Schritt „Bibliotheken abholen“ werden alle Bibliotheken aus externen Repositorien geladen und anschließend den weiteren Schritten mit lesendem Zugriff zur Verfügung gestellt. Der Schritt „Bibliotheken abholen“ ist in [65] nicht explizit angegeben. Dr. Ralph Holz, einer der Autoren von [65], hält auf Nachfrage das Hinzufügen vom Schritt „Bibliotheken abholen“ für einen praktikablen Ansatz (siehe B.12 auf Seite 118 im Anhang). Dabei gibt er zu bedenken, dass externe Bibliotheken bei Kompromittierung des Versionskontrollsystems oder des Erzeugungsprozesses Schadcode ausliefern können. Selbst die Prüfung einer bereitgestellten Signatur für die Bibliothek würde hier nicht den Schadcode erkennen, da die Signierung während der kompromittierten Erzeugung erfolgt. Bei der Nutzung von externen Bibliotheken ist grundsätzlich ein kompletter Quellcode-Review dieser zu empfehlen, um Schadcode in der Bibliothek auszuschließen. Entsprechend muss dann eine eigene Erzeugung und Bereitstellung der Bibliotheken erfolgen. In der Regel sind Quellcode-Reviews und eine eigene Erzeugung und Bereitstellung sehr Zeitintensiv, so dass kleine und mittelständische Unternehmen i.d.R. nicht die finanziellen Mittel dazu haben.

Weiter kann der vom *Debian GNU* Projekt zum Teil genutzte Ansatz der reproduzierbaren Erzeugung verwendet werden. Dabei wird davon ausgegangen, dass die Erzeugung eines Artefakts auf unterschiedlichen Systemen deterministisch ist. Wird ein Artefakt anschließend auf mehreren Erzeugungs-Systemen erzeugt und in unterschiedliche Repositorien gelegt, so sollten diese identisch sein. Entsprechend kann von unterschiedlichen Repositorien das Artefakt abgeholt und eine Prüfsumme erstellt werden, stimmen diese überein, wurde kein Erzeugungs-Server kompromittiert. Dafür muss dokumentiert werden, wie das Erzeugungs-System aufgesetzt ist.
Unterschiedliche Systeme haben unterschiedliches Verhalten. Beispielsweise ist der Dateimodus eines symbolischen Link im *Portable Operating System Interface* nicht definiert. Entsprechend können unterschiedliche Systeme unterschiedliche Dateimodi aufweisen. [32]
Weiter muss beispielsweise die Art der Archivierung angepasst werden, da das Werkzeug *tar* Dateien in der auf der Festplatte gespeicherten Reihenfolge in ein Archiv hinzufügt. Entsprechend muss, um eine deterministische Erzeugung sicherzustellen, die Reihenfolge alphabetisch mittels Umleitung der Ausgabe von dem Werkzeug *find* in das Werkzeug *sort* erfolgen um es anschließend mittels *tar* zu packen [31]. Das Werkzeug *find* findet rekursiv Dateien innerhalb eines Ordners, *sort* sortiert diese alphabetisch und *tar* erhält so eine sortierte Liste von Dateien, welche in das zu erstellende sortiert Archiv hinzugefügt werden.
Anschließend erfolgen Modultests, statische Tests und die Erzeugung eines Artefakts mit lesendem Zugriff auf die Bibliotheken. Ist eine Bibliothek oder ein Abbild bösartig und möchte z.B. Kontakt mit einem Server des Angreifers aufnehmen, kann dies an der Firewall registriert und eine Alarmmeldung ausgegeben werden. Beim Testen besteht nur lesender Zugriff auf das erzeugte Artefakt.
Sofern für das Testen weitere *CI*-System-*Plugins* notwendig sind, können diese in der virtuellen Testumgebung installiert werden, so können Schwachstellen in *Plugins* des *CI*-Systems entgegen gewirkt werden. Ist ein *Plugin* bösartig, so kann es nur noch die Testumgebung kompromittieren. [153]
Um Angreifern mit Zugriff auf das Versionskontrollsystem oder auf externe Repositorien entgegen zu wirken, sollte für jeden Schritt jeweils eine frisch aufgesetzte virtuelle Umgebung genutzt werden.

4.2.3 Sicherung der Vertraulichkeit und Verfügbarkeit beim Erzeugungs- und Verteilungsprozess

Bei der Verteilung sollten sicherheitsrelevante Konfigurationsparameter ausgetauscht werden. Sicherheitsrelevante Konfigurationsparameter sind beispielsweise Zugangsdaten für Dienste wie Datenbanken, externe Schnittstellen oder *SMTP*-Server. Bevorzugt sind die Werte der Konfigurationsparameter oder die gesamte Konfiguration dabei verschlüsselt abgelegt. Fraglich ist der Ablageort des Schlüssels. Der Schlüssel kann beispielsweise auf dem Betriebssystem abgelegt sein, mit Zugriffsrechten nur für die Anwendung [14]. Wird der Schlüssel bei Anwendungsstart als Umgebungsparameter übergeben, verlagert sich die sichere Speicherung ebenfalls auf das Betriebssystem. Alternativ kann er beim Startvorgang des Betriebssystem eingegeben werden und im Arbeitsspeicher gehalten werden [14]. Dies widerspricht jedoch den *DevOps*-Strategien, bei welchem der Startvorgang automatisch erfolgen sollte. Auch kann der Schlüssel im Quellcode abgelegt werden [14]. Hier kann er jedoch von Entwicklern und bei Kompromittierung des Versionskontrollsystems ausgelesen werden. Der Schlüssel kann auch auf die unterschiedlichen Ablageorte

aufgeteilt werden [14].

Um Fehler durch einen möglichst einfachen Prozess bei der Verschlüsselung und Signierung für Entwickler zu minimieren, kann das Konsolen-Werkzeug *keyzar* [103] verwendet werden. Bei Schlüsselerstellung kann zwischen symmetrischer und asymmetrischer Verschlüsselung gewählt werden. In *Java* und *python* lässt sich *keyzar* als Bibliothek integrieren [103].

Durch das Einspielen einer Aktualisierung eines Datenbankschemas können Fehler auftreten, welche zu Datenverlust führen können. Dadurch kann die Verfügbarkeit und Integrität beeinträchtigt werden. Entsprechend kann der Verteilungsprozess um ein Backup erweitert werden, welches vor jeder Verteilung durchgeführt wird.

Da Schwachstellen in Weboberflächen für *CI*-Systeme wie in [42, 213, 50] mehrfach nachgewiesen wurden, sollte das *CI*-System nicht öffentlich verfügbar sein. Zusätzlich sollte es über eine Rollen-basierte Zugriffskontrolle verfügen um Aktionen zu Systemen und Benutzern zuordnen zu können sowie Aktionen einschränken zu können.

Damit auf der Testumgebung mit gleichen Bedingungen wie auf der Produktionsumgebung getestet wird, ist Entwicklern angeraten nicht Umgebungs-basierte Variablen wie in Listing 4.5 zu nutzen, sondern Konfigurations-Parameter basierte Entwicklung wie in Listing 4.6. Dies bietet zusätzlich den Vorteil, dass Konfigurationsparameter in unterschiedlichen Kompositionen geprüft werden können. Ist eine Funktion defekt, kann diese über das *Feature Toggle* ohne erneute Erzeugung und Verteilung deaktiviert werden. Wird ein *Bruteforce*-Angriff erkannt, können Sicherheitsfunktionen wie den *Completely Automated Public Turing test to tell Computers and Humans Apart* (kurz Captcha) in Listing 4.6 über den *Feature Toggle* aktiviert werden, so Benjamin Pfänder (siehe Anhang B.5 auf Seite 114).

Listing 4.5 Beispiel für Umgebungs-basierte Entwicklung

```
1 if (host == 'production') {
2    // Enable Captcha
3 }
```

Listing 4.6 Beispiel für Parameter-basierte Entwicklung

```
1 if (getConfig('enableCaptcha')) {
2    // Enable Captcha
3 }
```

CI-Systeme wie *Jenkins* [19] bieten unterschiedliche *Plugins*, welche aus dem Internet geladen werden. Sind *Plugins Open Source*-Projekte, kann hier eine Schwachstelle versucht werden einzuschleusen um hinterher *CI*-Systeme mit dem genutztem *Plugin* anzugreifen. Auch können *Plugins* versehentlich implementierte Schwachstellen beinhalten, z.B. eine *XSS*-Schwachstelle [68]. Eine manuelle Prüfung von dem Quellcode von genutzten *Plugins* kann helfen, ist aber sehr zeitintensiv.

Plugins können die Sicherheit erhöhen, beispielsweise werden beim *Mask Passwords Plugin* [24] Passwörter in der Konfiguration der Erzeugung in Variablen abgelegt und beim Anzeigen des Status maskiert angezeigt [70, S. 71]. Das *Audit Trail Plugin* [176] zeichnet auf, wer wann welche Aktion durchgeführt hat [70, S. 78]. In Kombination sollte auch das *JobConfigHistory Plugin* [78] eingesetzt werden, welches es erlaubt Änderungen in der Konfiguration eines Erzeugungsauftrags nach zu verfolgen.

4.3 Informationsgewinnung

In der Praxis steht im Vordergrund Ereignisse zu identifizieren und manuelle Trendanalysen durchzuführen. In der aktuellen Forschung dagegen wird aus erfolgten Ereignissen eine Vorhersage für die weitere Nutzung getroffen.
In der Forschung werden Trendanalysen aufgrund von Metriken, z.B. für Defekte von Software [166, 129] oder für die Vorhersage von der Nutzung eines Dienstes, verwendet [252]. Im Rahmen der Forschung von Webanwendungs-Sicherheit ist interessant, wie hoch die Wahrscheinlichkeit ist, dass ein seit fünf Minuten aktiver *Denial of Service* weitere fünf Minuten anhält. In diesem Kapitel wird vorgestellt, welche Fragen bei der Überwachung von Webanwendungen gestellt werden müssen und wie Metriken gesammelt, ausgewertet und visualisiert werden können.

4.3.1 Traditionelle Überwachung

Überwachung umfasst sämtliche Bereiche eines Systems, inklusive Hardware, Betriebssystem, Anwendung, Sicherheit, Verfügbarkeit oder auch Nutzungsverhalten von Anwendern. Die Hauptaufgabe ist die Feststellung von Abweichungen zwischen Ist-Zustand und Soll-Zustand zur Erkennung, Beurteilung und Behebung von Anomalien in IT-Systemen. Beispielsweise der abrubte Ausfall einer Festplatte. Weitere Aufgaben sind die Prüfung und Kontrolle bestimmter IT-Parameter und die Dokumentation der Einhaltung gesetzlicher oder vertraglicher Vorgaben. [131, S. 9]
Während Nutzungsverhalten bei Webseiten i.d.R. über externe *Web-Analytics*-Werkzeuge erfasst wird, werden Hardware, Betriebssystem, Anwendung, Sicherheit und Verfügbarkeit häufig über interne Überwachungs-Lösungen erfasst.
Empfehlenswert ist die Festlegung von Grenzwerten, so dass bei Überschreitung ein vorher definierter Plan befolgt werden kann. Insbesondere für Risiken wie einen *Denial of Service* sollte ein Notfallplan im Vorfeld erstellt werden.
Bei der Implementierung einer Überwachung können nach Hein [131, S. 10] folgende Fragen unterstützen:

- Sind alle aktiven Hardware-Komponenten stetig gleichbleibend funktionstüchtig?
- Sind alle Systeme und Dienste aktiv?
- Sind alle Sicherheitsfunktionen aktiv?
- Arbeiten die Webseiten einwandfrei?
- Sind alle Datenbanken performant?
- Wie sind Fehlerquellen in virtualisierten Umgebungen aufzufinden?
- Ist das tägliche Backup erfolgreich durchgeführt?
- ...

Zur Überwachung können folgende Kategorien gebildet werden:

- Performance[147, S. 17]: Seitenlade-Zeiten, Antwortszeiten, ...
- Kapazität [147, S. 17]: Festplattenplatz, Arbeitsspeicher, Bandbreite, ...
- Verfügbarkeit [147, S. 17]: Verfügbarkeit eines Systems oder von Teilen

- Durchsatz [147, S. 17]: Durchsatz auf unterschiedlichen Ebenen (Web, Datenbank, Netzwerk, ...)
- *Service Level Agreements* [147, S. 17]: Verfügbarkeit, Stabilität, Sicherheit, ...
- *Key Performance Indicators* [147, S. 17]: Gewinn pro Minute, durchschnittliche Benutzeranzahl, ...
- Benutzerverhalten [147, S. 17]: Registrierungen, Absprungraten, Klickraten, ...
- Sicherheit: Zugriffe, Rechte, Einbruchserkennung, ...

Eine traditionelle Überwachungslösung ist *Icinga* [17], welches auf *Nagios* [26] basiert [109]. Unternehmen wechseln oftmals von *Nagios* zu *Icinga* [124]. Häufig werden ergänzend Metriken von Systemen und Anwendungen gesammelt und aufbereitet. Dies führt zu redundanter Datenhaltung und sollte vermieden werden.

4.3.2 Metriken

Galileo Galilei (154-1642) postulierte nicht Messbares messbar zu machen. Auch Sicherheits-Berater wie das *Center for Internet Security* empfehlen in [237, S. 3] Metriken als eine der wichtigsten Sicherheitsmaßnahmen für Organisationen.
Häufig ist die Sichtbarkeit der Sicherheit von Webanwendungen und Systemen erst bei Eintritt von Sicherheitsvorfällen gegeben. Damit das Sicherheitsniveau der Webanwendung abgeschätzt werden kann, müssen Metriken definiert werden. [83, S. 10]
Erfassung und Instrumentierung von Sicherheitsmetriken hat zusätzlich zur Überwachung folgende Aufgaben:

- Kommunikation der Wirksamkeit: Durch Sicherheitsmetriken kann die Wirksamkeit von Sicherheitsmaßnahmen besser erhoben und kommuniziert werden. Insbesondere können Aufwände im Bereich Sicherheit besser verstanden und über Visualisierung wahrgenommen werden [205].
- Standards: Durch Metriken werden indirekt Standards zur Performance-Messung, z.B. die Anzahl der veralteten Abhängigkeiten in einem Projekt pro Team, integriert [128, S. 28].
- Optimierung der Wirksamkeit: Nach Redman [206] ist was nicht gemessen ist, nicht gemanagt. Entsprechend können durch Sicherheitsmetriken Sicherheitsmaßnahmen optimiert werden.
- Trendanalyse: Durch Metriken können Trends, beispielsweise steigende Nutzung von Festplattenplatz über einen Zeitraum, ermittelt und Maßnahmen ergriffen werden. Insbesondere bei Performance-Analysen können Trendanalysen bei Priorisierung und Entscheidungen unterstützen.
- Diagnose-Hilfe: Bei einem aktiven Angriff können Metriken wertvolle Informationen bieten. So kann z.B. geprüft welchen, aus welchen Ländern ein *Bruteforce*-Angriff erfolgt und ggf. ganze Internetprotokoll Adress-Blöcke gesperrt werden.
- Konformitäts-Übersicht: Metriken können die Konformität zu einem Standard demonstrieren [205].
- Vergleichspunkte: Es kann verglichen werden, wie effizient das eigene Sicherheitsprogramm vergleichen mit dem von anderen Organisationen ist [128, S. 28]. Weiterhin können unterschiedliche Umgebungen verglichen werden [205].

Während qualitative Informationen möglicherweise unpräzise erscheinen, können quantitative Informationen unvollständig sein [128, S. 35]. Beispielsweise können Zutritte zum Rechenzentrum durch einen

elektronischen Sensor erfasst werden. Bei der Zutritts-Metrik werden organisatorische und soziale Aspekte, wie das Tür aufhalten eines Kollegen, jedoch nicht berücksichtigt. Einige Forscher warnen davor Sicherheit quantiativ zu erfassen, da Zahlen nicht vertraut werden kann oder die Zahlen zu ungenau sind [242]. Im Rahmen dieser Arbeit wird davon ausgegangen, dass Metriken empirisch erfassbar sind. Entsprechend stellen Metriken ein effizientes wissenschaftliches Werkzeug zur Messung der Sicherheit von Webanwendungen dar und tragen damit zur Erhöhung deren Sicherheit bei.
Gute Metriken erfüllen folgende Charakteristiken:

- Befriedigung eines bestimmten Geschäftsbedarfs: Bevor eine Metrik erhoben wird, sollte geprüft werden, ob die kommunizierten Informationen benötigt werden. Eine Metrik ohne Sinn sollte nicht erhoben werden, um Ressourcen zu schonen [205, S. 6]. Dies kann beispielsweise bei der Nutzung von vorkonfigurierten Erhebungswerkzeugen wie *collectd* [115] auftreten, wenn keine weitere Anpassung vorgenommen wird. *collectd* wird zum Erfassen von Betriebssystem-Statistiken verwendet. Brotby fasst die Befriedigung eines bestimmten Geschäftsbedarfs in [79, S. 10] zusammen mit der Frage „Who needs to know what when?“ (Deutsch Wer benötigt was wann?).
- Konsistente Messung: Eine Metrik kann nicht vergleichen werden, nicht einmal mit sich selbst, wenn diese nicht in einem definierten konsistenten Zustand erzeugt wird [140, S. 26].
- Kostengünstige Erzeugung: Je länger die Erhebung einer Metrik benötigt, um so weniger wertvoll ist diese [140, S. 24]. So kann eine Metrik beispielsweise manuell erhoben werden, dies ist jedoch ressourcenintensiv und hat ggf. ein niedriges Intervall. Deshalb sollten Metriken automatisiert erhoben werden.
- Erbringung quantifizierbarer Informationen: Die Metrik muss einen quantifizierbaren Wert messen, bevorzugt ausgedrückt als Kardinalzahl oder Prozentwert. Die Messung sollte nicht subjektiv, beispielsweise durch Ampelfarben, erfolgen. Dadurch würde die Vergleichbarkeit der Metrik verloren gehen. [205, S. 7]

  Es kann sinnvoll sein prozentuale Werte zu bevorzugen, beispielsweise kann ein Virenscanner fünf infizierte Dateien in der Produktivumgebung auffinden. Werden im nächsten Monat sieben infizierte Dateien gefunden, kann dies zu falschen Schlussfolgerungen führen, wenn die Serverfarm von vier auf acht Server während des zweiten Monats erweitert wurde. [205, S. 7]
- Ausgedrückt als Einheit: Zusätzlich zu der Anforderung als Zahl ausgedrückt zu werden, sollte die Metrik mindestens als eine Einheit dargestellt werden. Beispielsweise „Durchschnittliche infizierte Dateien pro Server“. [205, S. 8]

Folgende Fragen können beim Entwurf von Metriken durchlaufen werden:

1. Ziel: Was ist das Ziel der Metrik? Welche Risiken werden erfasst?
2. Datensammlung: Aus welchen Quellen werden für die Rohdaten gesammelt? Ist der Prozess manuell oder automatisch?
3. Datenverarbeitung: Aus welchen Daten wird die Metrik erzeugt? Wie werden die Daten verarbeitet und wie wird eine Analyse über Zeit durchgeführt?
4. Visualisierung: Wie werden die verarbeiteten Daten numerisch präsentiert und visualisiert?

Metrik-Kategorien und Quellen

Nach [140, S. 44f.] existieren folgende Kategorien für Metriken im Bereich Sicherheit:

- Umfassende Verteidigung: Um Risiken von Sicherheitsvorfällen verursacht durch Externe verstehen zu können, messen Organisationen die Effektivität von Verteidungssystemen wie Anti-Virus Software, Anti-Spam Systemen, Firewalls und Angriffserkennungssystemen.
- Abdeckung und Kontrolle: Es ist wichtig Kontrollmaßnahmen zu prüfen, deshalb wird die Effektivität von Konfiguration, Patching und Schwachstellen Management für Systeme geprüft.
- Verfügbarkeit und Stabilität: Umsatzstarke Systeme müssen verfügbar sein. Metriken wie die mittlere Reperaturzeit zeigen die Abhängigkeit zwischen Sicherheit und Profit auf.
- Webanwendung: Selbst erstellte oder fertige Webanwendungen müssen Sicherheit mit berücksichtigen, um unnötige Bloßstellung zu vermeiden. Anwendungs-Sicherheitsmetriken wie Schwachstellen-Zählung helfen bei der Quantifizierung von eigen erzeugten Risiken und Risiken durch Bibliotheken.

Im Folgendem werden die einzelnen Metriken zu den Metriken-Kategorien vorgestellt. Dabei wird nur auf Metriken eingegangen, welche im Sinne von *DevOps* automatisch erfasst werden können. Manuell zu erfassende Metriken, beispielsweise die vergangene Zeit seit dem letztem Sicherheitstraining, werden nicht aufgeführt.
Umfassende Verteidigungs-Metriken beinhalten das Sammeln von Informationen zu Anti-Virus- und Anti-Rootkit-Lösungen, Firewalls, Netzwerken und Angriffen. Für Firewalls und Netzwerke können u.a. die in Tabelle E.1 auf Seite 127 im Anhang vorgestellten Metriken gesammelt werden. Beispielsweise kann die Anzahl der eingehenden Verbindung nach *TCP / UDP*-Port gemessen werden und beinhaltet implizit die Anzahl der eingehenden Verbindung nach *TCP / UDP*-Protokoll. Durch Sammeln der Internetprotokoll-Adressen von eingehenden Verbindungen kann der geografische Quell-Standort ermittelt werden. Aufgrund von Instrumentierung als Weltkarte mit Hitzekarte (siehe Abb. E.1 für ein Beispiel) werden beispielsweise *Denail of Service*-Angriffe geografisch eingegrenzt und Quell-Standorte beziehungsweise Internetprotokoll-Adressblöcke können blockiert werden um den Angriff zu unterbinden.
Quantifizierung von Angriffen ist schwierig, da diese Abhängig von der Genauigkeit von Angriffserkennungssystemen sind. Metriken zur Erfassung von Angriffen sind ebenfalls in Tabelle E.1 dargestellt.
Durch Abdeckungs- und Kontroll-Metriken wird aufgezeigt wie effektiv das Sicherheits-Programm einer Organisation ist. Sicherheits-Programme sind u.a. durch unternehmensweite Richtlinien gestützt, allerdings werden diese nicht immer eingehalten. Abdeckung ist der Grad zu welcher eine bestimmte Sicherheitskontrolle für eine bestimmte Zielgruppe mit allen Ressourcen angewendet wird. Der Kontroll-Grad zeigt die tatsächliche Anwendung von vorgegebenen Sicherheits-Standards und -Richtlinien. Entsprechend werden durch Abdeckungs- und Kontroll-Metriken Lücken bei der Umsetzung von Richtlinien und Standards aufgezeigt. Umfassende Abdeckungs- und Kontroll-Metriken beinhalten das Sammeln von Informationen zu Anti-Virus Software sowie Anti-*Rootkits*, Patch Management, Server-Konfiguration und Schwachstellen-Management. In Tabelle E.2 auf Seite 128 im Anhang werden die wichtigsten vorgestellt.
Sicherheit, Verfügbarkeit und Stabilität bieten Konfliktpotential, da das Schließen einer Sicherheitslücke einen System-Ausfall herbeiführen kann. Insbesondere ungeplante Ausfallzeiten sollten erfasst werden, da diese zu Vertragsstrafen führen können. Typischerweise werden diese über eine Periode, beispielsweise ein

Jahr, erfasst. Eine Übersicht gibt Tabelle E.3 auf Seite 129 im Anhang.
Webanwendungen sind i.d.R. geschäftsbetrieben, so dass der Messbarkeit der Sicherheit von Webanwendungs-Quellcode und Konfiguration ein besonders hoher Stellenwert zugetragen werden sollte. Das Messen von Sicherheit in Quellcode ist allerdings schwierig. Jaquith [140, S. 74f.] schlägt statische und dynamische Methoden zur Prüfung der Sicherheit einer Anwendung vor. In Tabelle 4.2 wird eine Auswahl von Metriken, welche durch dynamische und statische Methoden erfasst werden, vorgestellt.
Zusätzlich sollten während der Ausführung der Anwendung in Produktion Messpunkte gesetzt werden [245], welche in die Bereiche Anfragen, Authentifizierung, Sitzung, Eingabe, Dateibehandlung, Benutzer-Trend, System-Trend und Klient gegliedert werden können. In Tabelle 4.3 ist für jeden Bereich ein Beispiel vorgestellt.

Aufbereitung von Rohdaten

Um aus Rohdaten Erkentnisse gewinnen zu können, müssen diese zunächst aufbereitet werden. Dabei unterstützen Methoden wie das arithmetische Mittel, Standardabweichung, Gruppierung und Aggregation, Zeitreihenanalyse und Quantil Analyse [140, S. 134].
Das arithmetische Mittel kann helfen, um einen Überblick über Vorgänge zu erhalten. Trotzdem kann dieses auch irreführend sein, beispielsweise beim arithmetischem Mittel der erfolglosen Anmeldeversuche für alle Webanwendungen einer Organisation. Werden hier z.B. 1000 Anmeldeversuche am Tag über alle Webanwendungen angegeben, sagt dies nichts da drüber aus, wie viele erfolglose Anmeldeversuche eine bestimmte Webanwendung aufweist. Im Weiteren können sich die Rohdaten stark unterscheiden. Die Rohdaten {5, 5, 5, 5, 5, 5, 5, 5} und {1, 2, 5, 6, 9, 6, 2, 1} für die aktiven Benutzer über einen Tag haben beide das arithmetische Mittel 5. Da die Standardabweichung nicht angegeben ist, kann nicht erkannt werden, dass für den zweiten Datensatz wahrscheinlich eine höhere Serverkapazität benötigt wird, um die Lastspitze von 9 abzufangen. [140, S. 136]

Informationsvisualisierung von Metriken

Metriken können textuell ausgegeben werden, für eine schnelle Erfassung von komplexen und umfassenden Daten ist die Visualisierung, z.B. als Balkendiagramm, Kreisdiagramm oder Streudiagramm stark unterstützend. Dabei sollte die Erfassung, Speicherung und Visualisierung einer Metrik an die zu kommunizierenden Informationen angepasst werden.
Informationsvisualisierung ist keine triviale Aufgabe und beinhaltet Komposition, Farben, Typographie, Anordnung und Platzausnutzung [140, S. 159].
Mögliche Datenvisualisierungen für Sicherheitsmetriken sind:

- Zähler: Zähler erfassen einen numerischen Wert und können hoch und runter gezählt und auf Null zurück gesetzt werden.
- Kreisdiagramme: Kreisdiagramme zeigen den Prozentsatz einer Kategorie des Ganzen an.
- Ringdiagramm: Das Ringdiagramm ist eine Abwandlung des Kreisdiagramms, hier werden zusätzlich Segmente gebildet.
- Säulendiagramm: Säulendiagramme gehören zu den Stabdiagrammen, hier sind die „Stäbe“ senkrecht angeordnet und zeigen die Kombination von zwei Variablen.

- Balkendiagramme: Wie Säulendiagramme, aber mit vertikal statt horizontal gelegter x-Achse [154]. Entsprechend zeigen Histogramme relative Häufigkeiten durch Flächeninhalte.
- Histogramme: Histogramme zeigen Proportionen für diskrete (nicht nummerische) und fortlaufende (nummerische) Spalten [225, S. 10].
- Geografische Diagramme: Karten zeigen geografische Verhältnisse, z.B. in Kombination mit einer Hitzekarte.
- Liniendiagramme: Darstellung eines Trends, z.B. als Funktion der Zeit.

Für Sicherheitsmetriken sind Liniendiagramme, Säulendiagramme und Balkendiagramme von erweitertem Interesse. Es kann sinnvoll sein die Werte in einem Balkendiagramm absteigend zu sortieren. Ein Ringdiagramm kann beispielsweise genutzt werden, um die Vorfälle pro Abteilung und Team darzustellen. Bei der Erstellung von Metriken gilt zu beachten, dass Rohdaten i.d.R. nicht sortiert vorliegen.

Werkzeuge

Um Metriken zu Speichern, kann *Graphite* [13] verwendet werden. Hier werden numerische Zeitreihen gespeichert. Eine Übersicht über eine typische Architektur bietet Abb. 4.8. Hier wird *collectd* zum Sammeln von Betriebssystem-Metriken verwendet, Anwendungen wie das *playframework* können mittels *statsD* oder direkt an *Graphite* Metriken senden. *Graphite* ist für die Speicherung der Metriken zuständig. *statsD* stellt unterschiedliche Bibliotheken zur Instrumentierung von Metriken für Klienten bereit, welche Daten an den *statsD*-Server zur Aggregation senden können, welcher diese anschließend an *Graphite* leitet. *Grafana* holt die Metriken von *Graphite* ab und visualisiert die Metriken in einem *Dashboard* auf Anfrage im Browser.

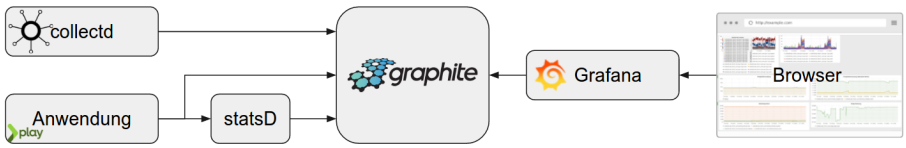


Abbildung 4.8: *Graphite*-*Grafana*-Architektur
Bildquellen: [116, 239, 13, 257, 117] und Bildschirmfoto von einem *Grafana*-Dashboard.

Während Metriken sehr gut die Anzahl der Ausnahmen zur Laufzeit einer Anwendung vermitteln können, können Metriken nicht die detaillierten Informationen der Ausnahmen vermitteln. Beispielsweise wird im Webserver *Apache™* ein Fehlerprotokoll geführt, in welchem Laufzeitfehler einer *PHP*-Webanwendung protokolliert werden. Entsprechend müssen System- und Anwendungs-Protokolle sinnvoll erfasst werden.

4.3.3 Protokollierung

Wie die Umfrage zu gemeinsamen Zielen in IT-Bereichen zeigt, werden Protokolle von allen Gruppen mindestens als zweitwichtigste Technik angegeben. Protokolle werden, insbesondere vom IT-Betrieb, zur Lokalisierung von Fehlern genutzt. Unter Protokollen wird hier die Ereignisprotokollierung verstanden. Dabei kann diese z.B. in einer Datei oder einer Datenbank erfolgen. Insbesondere bei Nutzung von Virtualisierung und *Microservice*-Architekturen treten erhöhte Mengen an dezentralen Protokollen auf. Dies verlangsamt die Analyse bei Fehlern, da auf jedem System einzeln die Protokolle nachverfolgt werden

müssen. Dabei ist insbesondere in Bezug auf Sicherheit eine schnelle Analyse notwendig, beispielsweise für das Identitäts- und Zugriffsmanagement.

Grundlagen der Protokollierung

Der Prozess der Protokollierung besteht nach Jansen [139] aus den Schritten: Sammeln, Vorbereiten und Analysieren. Beim Sammeln werden aus einer Protokoll-Quelle Daten empfangen. Beim Vorbereiten werden die gesammelten Daten normalisiert, also von Fehlern sowie überflüssigen Informationen bereinigt und formal aufbereitet gespeichert. Bei der Analyse werden Erkenntnisse aus den vorbereiteten Daten gewonnen. Indirekt lässt sich eine Ereigniskorrelation ableiten. Also das Auffinden von zusammenhängenden Ereignissen. Anschließend können Erkenntnisse überwacht werden, dass bedeutet das Erkennen und das Melden von Mustern [227].
Eine Korrelation über ein oder mehrere Protokolle kann schwer sein [238, S. 7]. Unter Korrelation wird das automatische Erkennen von Zusammenhängen aus einzelnen Ereignissen verstanden, wobei abhängig von dem erkannten Ereignis eine Aktion ausgeführt wird. Angriffe, welche auf individuelle Komponenten stattfinden, können so aufgedeckt werden. Als Beispiel führt Reißmann in [208] einen *Bruteforce*-Angriff auf den privilegierten System-Benutzer *root* via *SSH*-Dienst eines Server an. Es wird durch den Vergleich des Anmelde-Aufkommens mit einem durch den Holt-Winters Algorithmus [80] errechneten Erwartungswerts analysiert, wann ein Angriff stattfindet. Anschließend wird geprüft, ob in dieser Zeit ein erfolgreiche Anmeldung erfolgte. Ist dies der Fall, wird ein Alarm ausgegeben.
Um eine Korrelation von Ereignissen zu unterstützen, sollten Protokolle zentral erfasst werden [81, S. 544]. Herausforderung ist dabei der Kontext, also unterschiedliche Zeitzonen, Formate und die Sprache [238, S. 7].
Der Prozess der Protokollierung von Jansen wird von Steinegger et al. [227] erweitert, so dass der Prozess die Schritte: Sammeln, Normalisieren und Korrelieren, Verwahren, Analysieren und Überwachen aufweist.
Protokollierung erfolgt i.d.R. in wenig strukturierten Daten, welche ein Muster aufweisen. Beispielsweise erfolgt die Protokollierung von Zugriffen beim Webserver *Apache™* wie folgt:

```
127.0.0.1 - - [19/Jun/2016:06:56:03 +0200] "GET /xampp/status.php HTTP/1.1"200 3891
```

Mit Werkzeugen können diese Informationen in ein strukturiertes Format wie die *JSON* überführt werden.
Neben Protokollen von Systemen und Fehlern in Anwendungen können Protokolle auch bei Aufgaben wie Lasttests unterstützen [142].

Werkzeuge

Zum Sammeln und Normalisieren von Protokollen wird häufig *logstash* verwendet. *logstash* besteht im Wesentlichem aus einem Eingabe-*Plugin*, einem Filter-*Plugin* und einem Ausgabe-*Plugin*. Das Eingabe-*Plugin* nimmt Daten aus unterschiedlichen Quellen entgegen. Quellen sind z.B. *Logback* beziehungsweise *log4j*, Dateien oder der *Linux-Syslog*-Dienst *rsyslog*. Anschließend werden die Daten mittels Filter-*Plugin* verarbeitet. Hier werden unnötige Informationen entfernt, geografische Daten verarbeitet, Zeitstempel hinzugefügt oder Daten anonymisiert. Das Ausgabe-*Plugin* übergibt die Protokolle zur Speicherung, z.B. mittels der Volltext-Suchmaschine *Elasticsearch* [27].
l*ogstash* oder auch *rsyslog* können die Eingabe übernehmen. Der Protokoll-Eintrag vom Webserver wird

bei *logstash* wie folgt als *JSON* normalisiert und anschließend an einen Dienst zur Speicherung gesendet:

```
{
         "message" => "127.0.0.1 - - [19/Jun/2016:06:56:03 +0200] \"GET /
            status.php HTTP/1.1\" 200 3891",
      "@timestamp" => "2016-06-19T08:56:03.000Z",
        "@version" => "1",
            "host" => "web1.example.com",
        "clientip" => "127.0.0.1",
           "ident" => "-",
            "auth" => "-",
       "timestamp" => "19/Jun/2016:06:56:03 +0200",
            "verb" => "GET",
         "request" => "/status.php",
     "httpversion" => "1.1",
        "response" => "200",
           "bytes" => "3891"
}
```

ElasticSearch nimmt Protokolle via *HTTP REST*-Schnittstelle entgegen und nutzt die *NoSQL*-Technologie *Apache Lucene™* um die Speicherung möglichst performant durchzuführen.
Nach der Speicherung werden die Protokolle visualisiert. Dafür kann, wie für die Visualisierung von Metriken, *Grafana* oder *Kibana* verwendet werden. Mittels *ElastAlert* kann eine Alarmierung aufgrund der in *ElasticSearch* gespeicherten Protokollen erfolgen.
Die Nutzung von *ElasticSearch, logstash* und *Kibana* wird als *ELK-Stack* bezeichnet und hat einen hohen Verbreitungsgrad.
Wird *Docker* verwendet, so kann das Protokoll einer gestarteten Anwendung, z.B. mittels *logstash,* vom Dateisystem des Hosts gelesen und an *ElasticSearch* übergeben werden. Sofern die Anwendung es unterstützt, kann das Protokoll auch direkt an *ElasticSearch* gesendet werden. Beispielweise kann in der Konfiguration der Bibliothek *Logback* ein `LogstashSocketAppender` genutzt werden, um das Anwendungs-Protokoll via *UDP* an einen *logstash*-Server zu senden.
Traditionell nutzt der IT-Betrieb Protokolle. Bei der Software wird häufig Fehleranalyse mit einem Debugger betrieben und nur wenig Protokolle verwendet [107]. Entsprechend muss auf Seite von Entwicklung mehr Überzeugungsarbeit geleistet werden. Überzeugend ist, dass Anwendungsprotokolle nicht selbstständig geprüft werden müssen, sondern dass diese zentral gespeichert werden. Treten Protokolleinträge mit hoher Kritikalität auf, beispielsweise Ausnahmen in Java, so kann automatisch eine Alarmierung erfolgen.
Shang zeigt in [221], dass die Erhöhung der Qualität durch Protokollierung Entwicklung und IT-Betrieb näher zusammenbringt. In diesem Sinne kann Protokollierung auch Sicherheit an die beiden Bereiche herantragen.

4.3.4 Echtzeit-Überwachung

Wenn Metriken und Protokolle mittels *ELK-Stack* oder *Graphite* und *Grafana* erfasst und visualisiert werden, können Grenzwerte festgelegt und Alarme erzeugt werden. Im Gegensatz zu traditioneller Überwachung, bei welcher Protokolle häufig zeitversetzt analysiert werden, erfolgt so eine Alarmierung in Echtzeit. *Kibana* und *Grafana* bieten dabei interaktive Statistiken, in welchen einfach ein Zeitraum ausgewählt werden kann.

Wird *ElasticSearch* zur Speicherung verwendet, kann mittels *elastalert* [163] alarmiert werden, welches die Daten in *ElasticSearch* auswertet. Wird stattdessen *Graphite* verwendet, kann mittels *seyren* [53] alarmiert werden. Für die Oberfläche *Grafana* ist für Juli 2016 geplant, Alarmierung in der Oberfläche konfigurieren zu können [45].

#	Schritt	Funktion	Zugriffsrechte	Komp. / Krit.	Netzwerkzugriff	Auswirkungen bei Kompromittierung
1	Orchestrierung	Auslösung von jedem Schritt der Erzeugung	Lesen von Konfiguration	Niedrig / Hoch	TLS mit Verteilung	Ändern der Reihenfolge der Schritte
2	Quellcode abholen	Abholung von Quellcode vom VCS in den lokalen Arbeitsbereich	Lesen vom VCS; Schreiben in lokalen Arbeitsbereich (LA); Keine Ausführung	Niedrig / Hoch	TLS mit Verteilung	Abholen von falschem Quellcode; Manipulation von Quellcode
3	Bibliotheken abholen	Abholung von Bibliotheken aus externen Repositorien	Lesen vom LA von # 2; Schreiben in LA	Niedrig / Hoch	Zugriff auf externe Repositorien	Ersetzen durch bösartige Bibliotheken; Manipulation von Bibliotheken
4	Modultests und statische Tests	Ausführung von automatisierten Tests gegen den Quellcode	Lesen vom LA von # 2 und 3; Schreiben in LA	Hoch / Niedrig	Zugriff auf Aktualisierungen für Abhängigkeitsprüfungen	Kann den Prozess stoppen
5	Artefakt erzeugen	Erzeugung eines Artefakts aus Quellcode und Bibliotheken	Lesen vom LA von # 2 und 3; Schreiben und ausführen in LA	Hoch / Hoch	Keinen	Manipulation des Artefakts
6	Abbilder abholen	Holt Abbilder aus externen Repositorien	Lesen vom LA von # 2; Schreiben in LA	Niedrig / Hoch	Zugriff auf externe Repositorien	Abholen von falschen Abbildern; Manipulation von Abbildern
7	Abbild erzeugen	Erzeugung eines Abbilds aus Artefakt und externem Abbild	Lesen vom LA von # 5 und 6; Schreiben in LA	Mittel / Hoch	Keinen	Ersetzen des Abbilds mit einem bösartigem
8	Abbild signieren	Signierung des Abbilds	Lesen vom LA von # 7	Sehr niedrig / Niedrig	Keinen	Kann Prozess stoppen
9	Abbild archivieren	Überführung des Abbilds in internes Repositorium	Lesen vom LA von # 7; Schreibzugriff auf internes Repositorium	Sehr niedrig / Hoch	TLS mit internem Repositorium	Ersetzen des Abbilds mit einem bösartigem; Herausgabe von Anmeldedaten (internes Repositorium)
10	Abbild verteilen	Verteilung des Abbilds auf Testumgebung und Produktionsumgebung	Lesen von Konfiguration	Niedrig / Hoch	Zugriff auf Umgebungen	Ersetzen des Abbilds mit einem bösartigem; Herausgabe von Anmeldedaten (internes Repositorium)
11	Abbild abholen	Abbild abholen vom internen Repositorium	/	Niedrig / Hoch	TLS mit internem Repositorium	Ersetzen des Abbilds mit einem bösartigem; Herausgabe von Anmeldedaten (internes Repositorium)
12	Dynamische Tests durchführen	Durchführung dynamischer Tests	Lesen vom internem Repositorium; Lesen vom LA von Schritt #2; Schreiben in LA	Hoch / Niedrig	Zugriff auf Testumgebung	Herausgabe von Anmeldedaten (internes Repositorium); Kann Prozess stoppen

Tabelle 4.1: Härtung der Erzeugungs- und Verteilungsschritte in Anlehnung an [65]

Kategorie	Metrik (Einheit)	Zweck	Quellen
Dynamische Methoden	Schwachstellen-Zählung	Zeigt extern identifizierte Schwachstellen durch Implementierungs- oder Entwurfs-Fehler.	DAST-Werkzeuge
	Anzahl der Schwachstellen pro Anwendung - Nach Geschäftseinheit - Nach Kritikalität	Misst die Anzahl an Schwachstellen, die ein Angreifer ohne vorheriges Wissen erlangt.	DAST-Werkzeuge
Statische Methoden	Tausend Quellcode-Zeilen nach Modulen	Zeigt die aggregierte Größe eines Moduls.	SAST-Werkzeuge
	Schwachstellen pro tausend Quellcode-Zeilen	Zeigt die Bedrohungsrate von Schwachstellen pro entwickelte Zeile Quellcode.	SAST-Werkzeuge

Tabelle 4.2: Auswahl von Sicherheitsmetriken für Webanwendungen ermittelt durch Werkzeuge in Anlehnung an [140, S. 74f.]

Bereich	Metrik (Einheit)	Zweck
Anfrage	Fehlende, redundante oder zusätzliche Daten in einer Anfrage	Erkennung von unerwarteten Daten in HTTP-Headern.
Authentifizierung	Geografische Abweichung	Erkennung von einem ungewöhnlichen geografischen Standort eines Benutzers beim Anmelden.
Sitzung	Änderung eines Benutzer-Agenten während einer Sitzung	Erkennung von Zugriffen durch Dritte.
Eingabe	Fehelende, redundante oder zusätzliche Daten für Eingaben	Erkennung der Manipulation von Eingaben.
Dateibehandlung	Null-Byte Zeichen in Datei-Anfragen	Erkennung von Null-Byte Zeichen in Dateinamen.
Benutzer-Trend	Geschwindigkeit der Nutzung	Eine hohe Geschwindigkeit der Nutzung indiziert automatisierte Werkzeuge.
System-Trend	Hohe Anzahl an Abmeldungen	Eine hohe Anzahl an Abmeldungen in der gesamten Anwendung deutet auf *XSS* und *CSRF* hin.
Klient	Ungewöhnliches Klienten Verhalten	Erkennung von ungewöhnlichem Verhalten durch Klienten. Beispielsweise kann ein Benutzer-Agent ausgelesen über JavaScript sich von dem in HTTP-Anfragen gesendeten unterscheiden, was auf ein Konsolen-Werkzeug wie *curl* hindeutet.

Tabelle 4.3: Auswahl an Sicherheitsmetriken ermittelt in Webanwendungen in Anlehnung an [245]

Kapitel 5

Vorstellung automatisierbarer Sicherheitstests

Nachdem die Infrastruktur gehärtet ist, sollte das Sicherheitsniveau der Webanwendung und der Infrastruktur festgestellt werden. In diesem Abschnitt wird auf dynamische und statische Sicherheitstests eingegangen um anschließend aufzuzeigen, wie gefundene Risiken behandelt werden sollten. Zum Abschluss werden die Testarten evaluiert.
Es werden nur Werkzeuge vorgestellt, welche automatisiert werden können. Werkzeuge wie *jsprime* [190] und *RIPS* [99], welche die Eingabe via Webbrowser (beziehungsweise *HTTP* Post) benötigen und keinen einfach strukturierten Bericht ausgeben, sind nicht berücksichtigt.
Ein manueller Sicherheits-Audit ist effektiver im Auffinden von Risiken als ein rein Werkzeug-basierter Test, der Audit jedoch nicht automatisierbar [172, S. 6].

5.1 Dynamische Sicherheitstests

Bei dynamischen Tests wird die Laufzeitumgebung getestet. Als Werkzeug wird dafür häufig ein Web-Schwachstellenscanner, welcher auf mehrere Schwachstellen testet, eingesetzt.

5.1.1 Web-Schwachstellenscanner

Zunächst wird ein *Spider* (auch *Crawler* genannt) eingesetzt, welcher alle Pfade einer Webanwendung erfasst. Auf der Datenbasis vom *Spider* testet der Scanner auf Angriffsvektoren.
Zap ist ein auf *Java* basierender Schwachstellenscanner für Webanwendungen. Ein Benutzer kann *HTTP*-Anfragen durch den *Proxy* von *Zap* leiten und so einer internen Liste von *Zap* hinzufügen. Der *Spider* testet direkt die Webanwendung und fügt die kompletten Anfragen und Antworten der internen Liste inklusive Parameter hinzu. Der Scanner prüft alle Parameter jeder *URL* der internen Liste.
Es wird zwischen einem *HTTP*- und einem *Ajax-Spider* unterschieden. Der *HTTP-Spider* ruft Webseiten per *HTTP* ab und sucht in dem ausgeliefertem *HTML* der Seite nach Links und Formularfeldern. Der *HTTP-Spider* parst das *HTML* einer Webseite und besucht alle gefundenen Links und füllt alle Formulare aus und schickt diese ab. HTTP-Anfragen und HTTP-Antworten werden einer internen Liste hinzugefügt.

Moderne Webseiten nutzen *JavaScript* und das *Asynchronous JavaScript and XML*-Modell (kurz *Ajax*-Modell), welches den *Document-Object-Mode*l-Baum (kurz *DOM*-Baum) im Browser zur Laufzeit verändert. Veränderungen können beispielsweise die Einbettung von Links oder Formularen sein. Oft werden Daten dafür mittels *Ajax* vom Webserver geholt beziehungsweise Formulare über *Ajax* abgesendet. Beispielsweise wird in Webanwendungen auch auf den komplett geladenen *DOM*-Baum gewartet, bevor mittels *JavaScript* Bilder nachgeladen werden. Solche *DOM*-Änderungen bleiben dem normalen *Spider* verborgen, was zu einer schlechten Abdeckung in der internen Liste führt. [170]
Ein einfaches Beispiel ist der Link `<a href="javascript:location.href='page.html'">Webseite</a>`, hier wird das Attribut *href* des *a*-Tags zur Laufzeit von *JavaScript* interpretiert. Der *HTTP-Spider* kann die im *a*-Tag verlinkte Webseite also nicht finden und den Pfad nicht erfassen.
Jede Änderungen am *DOM*-Baum wird als separater Status gespeichert. Seitenstatus können in Form eines Status-Fluss-Graphen visualisiert werden 5.1. Für das Beispiel in Abb. 5.1 sind zwei Buttons vorhanden. Ausgangspunkt ist Status *S1* nach laden der Webseite. Es können durch Klick auf den ersten

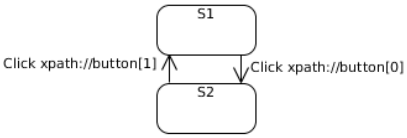


Abbildung 5.1: Visualisierung mittels Status-Fluss Graph

Button Inhalte nachgeladen und in den *DOM*-Baum integriert werden, was durch *S2* repräsentiert ist. Das Laden von Inhalten wird durch den *Proxy* in *Zap* mitgeschnitten und der neue Pfad wird der internen Liste hinzugefügt. Wird auf den zweiten Button der Anwendung geklickt, werden die geladenen Inhalte wieder entfernt und *S1* liegt vor.
Um Änderungen am *DOM*-Baum durchführen zu können, wird ein *Spider* auf Basis eines Browsers benötigt. In *Zap* wird dies über den *Ajax-Spider* realisiert. Der *Ajax-Spider* ist in *Zap* mittels *Crawljax* [170] implementiert.

5.1.2 Statuserkennung mit *Crawljax*

Zap startet *Crawljax*, welcher anschließend ein (oder mehrere) Browserfenster öffnet und Elemente innerhalb der Seite klickt beziehungsweise Formulare ausfüllt. Versucht *Crawljax* eine nicht im Bereich befindliche Seite zu laden, liefert Zap *„Not in Scope“* zurück. Dabei wird die Anfrage an die Webseite von *Zap* blockiert.
Folgende Komponenten werden von *Crawljax* genutzt: [170]

- *Browser*: Dient als Schnittstelle zur Ausführung von *Javascript*.
- *Robot*: Simuliert Benutzereingaben wie beispielsweise einen Klick.
- *Controller*: Hat Zugriff auf den *DOM*-Baum des *Browsers* und erkennt so Änderungen am *DOM*-Baum. Im Weiteren führt der Controller Aktionen über den *Robot* aus und ist Zuständig für die Aktualisierung der *State Machine* bei Änderungen am *DOM*-Baum.
- *State Machine*: Speichert den Statusfluss als Graph und hält eine Referenz auf den aktuellen Status.

5.1.3 Erhöhung der Abdeckung

Herausforderung ist die Pfad Abdeckung möglichst hoch zu halten. Dabei muss z.B. Authentifizierung mit unterschiedlichen Rollen beachtet werden.
Nicht immer ist es sinnvoll, die gesamte Anwendung abzudecken. Beispielsweise bei wiederkehrenden Webseiten mit der gleichen Anwendungslogik und dem gleichem *Template*. Dies ist häufig bei *Shop*-Webanwendungen der Fall, in welchen mehrere tausend oder Millionen von Artikeln auf wenige *Templates* mit gleicher Anwendungslogik fallen. Hier sollten Artikel mit unterschiedlichen *Templates* beziehungsweise Anwendungslogik in einer Positivliste festgehalten werden, um diese jeweils einmal zu testen. Die anderen *Templates* können über eine Negativliste mit Platzhaltern ausgeschlossen werden.
Häufig beinhalten Webanwendungen Arbeitsabläufe, bei welchen bestimmte Aktionen in einer definierten Reihenfolge durchgeführt werden müssen. Der *Spider* ist i.d.R. nicht in der Lage die Arbeitsabläufe sequentiell korrekt auszuführen. Entsprechend müssen Arbeitsabläufe, wie ein Bestellprozess in einer *Shop*-Webanwendung, zusätzlich implementiert werden. Beispielsweise kann *cURL* [6] mit *Zap* als *Proxy* genutzt werden, um Arbeitsabläufe zu definieren und in die interne Liste von *Zap* aufzunehmen. Alternativ nutzen Entwickler häufig Modul- und Akzeptanztests um die Funktionalität sicherzustellen, sofern vorhanden, können diese durch *Zap* als Proxy in die internen Liste aufgenommen werden.
Wird eine *Microservice*-Architektur verwendet, bei welcher *Microservices* z.B. via *Simple Object Access Protocol* kommunizieren, so sollte diese auch geprüft werden.
Die Abdeckung selbst kann in *Java* manuell mittels *OWASP Code Pulse* [204] geprüft werden. Hier wird zur Laufzeit aufgezeigt, welche Bereiche der Anwendung zur Ausführung gekommen sind.

5.1.4 Aktives und passives Testen

Bei einem Passwort basiertem Authentifizierungs-System, wie häufig bei Webanwendungen und der Infrastruktur eingesetzt, sollten automatisch die Passwörter der Benutzer getestet werden. Wird ein Passwort Authentifizierungs-Versuche mittels *Bruteforce*-Angriff versucht herauszufinden, handelt es sich um einen aktiven Test [197, S. 11]. Wird das aktive System nicht aufgerufen oder verändert, beispielsweise beim Testen der Zugriffsrechte einer Passwortdatei oder beim Vergleichen von Passwort-Hashwerten, handelt es sich um passives Testen [197, S. 11].
Zap analysiert mitgeschnittene Anfragen und Antworten und kann so z.B. erkennen, dass ein *HTTP-Only-Flag* für ein Cookie eines Authentifizierungsvorgangs nicht gesetzt ist. Diese Art der Analyse wird als passiver Scan bezeichnet. Der aktive Scan dagegen prüft auf Grundlage der Pfade alle Parameter auf Schwachstellen. Dabei wird der Wert eines Parameters modifiziert. Angenommen in der internen Liste mit Pfaden liegt die *URL* http://localhost:8080/bodgeit/?page=entry vor. Auf eine unerlaubte Dateieinbindung wird dann über die *URL* http://localhost:8080/bodgeit/?page=/etc/passwd durch Modifizierung des *page*-Parameters getestet. Optional können auch weitere *HTTP*-Kopfzeilen wie *Cookies* geprüft werden.

5.1.5 Erstellung von Anwendungstests während der Entwicklung

Während in klassischen Vorgehensmodellen (z.B. dem V-Modell) Tests nachgelagert durchgeführt werden, sind Tester und Entwickler bei agilem Vorgehen gleichberechtigt [250, S. 259].

Es ist empfohlen, dass Entwickler nicht ihren eigenen Quellcode testen, da Entwickler bei der Testkonzeption zu einem Beweis von Fehlerfreiheit tendieren. Das Ziel der Tests ist jedoch das Auffinden von Fehlern, entsprechend sollten Team-Kollegen Tests durchführen. Ausnahme sind die Tests, welche direkt bei der Programmierung entworfen und implementiert werden [83, S. 33]. Modultests können die aus *AbUser Stories* entstandenen negativen Anforderungen testen. Die *AbUser Story* „Als Angreifer möchte ich mittels *Cross-Site Scripting* die Webseite manipulieren“ führt zu der Implementierung einer Ausgabebereinigung. Die Implementierung dieser kann mittels Modultest geprüft werden. Es wird auf Schwachstellen in Sub-Routinen, Funktionen, Modulen, Bibliotheken usw. getestet. Dabei muss beachtet werden, dass negatives Testen nicht trivial ist. Beispielsweise kann vergessen werden auf Schwachstellen, welche durch Metazeichen entstehen, zu testen. Es ist unwahrscheinlich, dass der Tester hier zufällige Eingaben zum Testen erzeugt. [71, S. 894]
Beispielsweise kann die Methode `checkInputCharactersAreValid()` Eingaben auf nicht zulässige Zeichen testen. Sie gibt `true` bei ausschließlich erlaubten beziehungsweise `false` bei nicht erlaubten Eingaben zurück. Ein Modultest kann diese Methoden auf Richtigkeit testen, indem die Methode mit erlaubten und nicht erlaubten Eingaben aufgerufen wird. Ist das Zeichen „`'`“ nicht erlaubt und wird als Eingabe verwendet, so muss die Methode `false` zurückliefern. Entspricht der Rückgabewert der Erwartung, gilt der Modultest als bestanden, sonst als nicht bestanden. [82, S. 23]
In Java werden Modultests häufig mit *JUnit* [20] implementiert.
Die *AbUser Story* „Als Angreifer möchte ich Anfragen manipulieren um die Preise der Produkte im Warenkorb zu ändern“ kann mittels Integrations-Tests geprüft werden. Dabei werden *HTTP*-Anfragen abgesendet, ausgeführt und das Ergebnis geprüft. Integrationstests, z.B. mittels *HttpUnit* [16], sind als Blackbox-Tests und benötigen mehr Zeit bei der Durchführung als Modultests. Mittels Integrationstests sollten insbesondere Eingaben geprüft werden als auch das Ausbrechen aus dem gegebenem Kontrollfluss der Webanwendung [234]. Nach [71, S. 894] eigenen sich Integrations- und Akzeptanztests insbesondere zum Prüfen der Authentifizierung und Autorisierung einer Anwendung.
Akzeptanz-Tests können mittels *Selenium* [34] oder *HtmlUnit* [75] realisiert werden. Im Unterschied zu Integrationstests operieren diese direkt auf der Präsentationsschicht und benötigen entsprechend einen Browser. Aufgrund dessen benötigen diese mehr Zeit. Beispielsweise können Akzeptanztests Hashfunktionen für Passwörter, welche im *JavaScript* genutzt werden, testen.

5.1.6 Lasttests

Lasttests ermitteln die Maximalauslastung eines Systems durch absenden mehrerer Anfragen und gehören damit zu den dynamischen aktiven Tests. Ein Test Plan kann schnell manuell erstellt werden, indem ein Proxy zum Aufzeichnen der eingeschlagenen Pfade in der Webanwendung verwendet wird [125, S. 54].
Besser ist es repräsentatives Benutzerverhalten des Produktionssystems künstlich nachzustellen [215]. Die eingeschlagenen Pfade in der Webanwendung werden durch Messpunkte gesammelt. Anschließend wird die Wahrscheinlichkeit des Einschlagens jeden Pfades und die zugehörige Frequenz berechnet und daraus einen Test Plan erstellt [132]. Dabei muss die Sitzung des Benutzers beachtet werden und das genutzte Werkzeug zur Lasterzeugung *Cookies* unterstützen. Das Werkzeuge *Apache JMeter*™ [2] kann zur Ausführung eines Test Plans genutzt werden und unterstützt *Cookies*.

5.1.7 Invarianttests

Es sollte geprüft werden, dass bei normalen Anwendungstests keine Fehlermeldungen vorhanden sind [69]. Weiter sollten keine toten Links zu permanent nicht erreichbaren *URLs* vorhanden sein [171]. Ist beispielsweise ein Link zu einer bereits frei verfügbaren Domain vorhanden, kann ein Angreifer diese Domain erwerben und hier beliebige Aktionen durchführen.

5.1.8 Übersicht über Werkzeuge

Tabelle 5.1 gibt einen Überblick über eine Auswahl von Werkzeugen. Durch Angabe der letzten Änderungen wird die Aktualität des Projekts aufgezeigt. Änderungen nach dem 14. April 2016 sind in dieser und den folgenden Übersichten über Werkzeuge nicht berücksichtigt.
The Hacker's Choice Hydra (kurz *THC Hydra*) erzeugt bei korrekter Konfiguration keine falsch Positiven, ebenso ein Lasttest.

Test-Kategorie	**Name**	**Integration**	**Beschreibung**	**Letzte Änderung**	**Behandlung von falsch Positiven**
Anwendungstest	HtmlUnit [75]	Maven, IDE	Tests im emuliertem Browser	April 2016	/
Anwendungstest	HttpUnit [16]	Ant, IDE, Maven	Tests auf HTTP-Ebene	April 2016	/
Anwendungstest	JUnit [20]	Ant, IDE, Maven	Tests auf Modul-Ebene	April 2016	/
Anwendungstest	Selenium [34]	IDE, Maven	Tests im Browser	April 2016	/
Lasttest	Apache JMeter™ [2]	Konsole	Test auf Benutzerverhatlen	April 2016	/
Passwort-Sicherheit	THC Hydra [36]	Konsole	Test auf Passwort-Stärke von Benutzern	April 2016	/
SQL-Injection	sqlmap [226]	Konsole	Erweiterter Test auf SQL-Injection	April 2016	Nein
Web-Schwachstellen	arachni [158]	Konsole, Web	Test auf Web-Schwachstellen	April 2016	Ja
Web-Schwachstellen	OWASP Zap [188]	GUI, Konsole, REST-Schnittstelle	Test auf Web-Schwachstellen	April 2016	Nein
Web-Schwachstellen	w3af [210]	GUI, Konsole	Test auf Web-Schwachstellen	Februar 2016	Nein

Tabelle 5.1: Auswahl von Werkzeugen zum dynamischen Testen von Webanwendungen

5.2 Statische Sicherheitstests

Bei der statischen Analyse wird der Quellcode oder die compilierte Form untersucht, eine aktive Laufzeitumgebung wie bei dynamischen Tests wird nicht benötigt. Ansätze wie *JFlow [178]*, welche zusätzliche Annotation innerhalb von Quellcode benötigen, werden nicht betrachtet.

5.2.1 Typprüfung

Die Typprüfung ist stark limitiert beim Auffinden von Defekten. Der *Java*-Quellcode in Listing 5.1 wird nicht kompilieren, da Typ `int` nicht eine Variable vom Typ `short` mit weniger Präzision zugewiesen kann [90, S. 25]. Ist die Typumwandlung vom Entwickler beabsichtigt kann er jedoch nur mittels expliziter Typumwandlung die falsch Positiv-Meldung vom *Java*-Compiler umgehen.

Listing 5.1 Beispiel eines falsch Negativen einer Typprüfung [90, S. 25]

```
1 short s = 0;
2 int i = s; // Allowed
3 short r = i; // Not allowed
```

5.2.2 *String Matching* Algorithmen

Bei *String Matching* Algorithmen werden Zeichenketten innerhalb des Quellcodes erkannt und angezeigt. Quellcode mit dem Konsolen-Werkzeug *grep* nach Begriffen oder Mustern zu durchsuchen stellt eine der einfachsten Formen der *String Matching* Algorithmen dar, ist aber auch sehr unflexibel [89, 241]. Werden in einem *PHP*-Projekt mittels dem Konsolenwerkzeug *grep* die Begriffe *mysql_ query* und *$_ GET* aus Quellcode gefiltert, werden neben Zeilen mit möglicher SQL-Injection wie `mysql_query(" SELECT * FROM user WHERE id = '".$_GET["id"]."'");` auch gegen SQL-Injection abgesicherte Zeilen wie `mysql_query("SELECT * FROM user WHERE id = '".mysql_real_escape_string($_GET["id "])."'");` gefunden.

Studien wie [141, 241] belegen eine erhöhte Anzahl von falsch Positiven bei *String Matching* Algorithmen. Auch ist erhöhtes Experten-Wissen bei der Analyse notwendig. Das Werkzeug *PHP Security Scanner* [113] nutzt einen *String Matching* Algorithmus zum Auffinden von Aufrufen mit `mysql_query()`. Anschließend wird geprüft, ob Eingabe-Parameter direkt genutzt werden. Moderne *PHP*-Webanwendungen nutzen allerdings häufig `mysqli_query()` beziehungsweise die *PHP Data Objects*-Erweiterung, so dass der *PHP Security Scanner* hier falsch Negative aufweisen wird.

Die *Stilanalyse* beschäftigt sich mit der Überprüfung von Programmkonventionen. Dabei wird zwischen einem Regelwerk, Ausschluss von Konstrukten und Still unterschieden. Bei einem Regelwerk werden Organisatons-, Abteilungs- oder Teamübergreifende Vorschriften überprüft. Eine Regel kann korrektes Einrücken von Quellcode sein. Zeile fünf in Listing 5.2 suggeriert, dass `else` zur ersten Bedingung gehört, grammatikalisch gehört es jedoch zur zweiten Bedingung. Es resultiert wahrscheinlich ein Logikfehler, da der Entwickler für x den Wert 1 annimmt, während x jedoch der Wert 10 zugewiesen ist. Je nach anschließender Nutzung von x, kann dies die Sicherheit der Webanwendung beeinträchtigen. Beispielsweise innerhalb einer Zugriffskontrolle.

Beim Ausschluss von Konstrukten werden Fehler begünstigende Konstrukte erkannt. Werkzeuge wie *PMD* [96] können solche Fehler finden.

Stilanalysen werden häufig von Entwicklungsumgebungen wie Eclipse bereits während der Entwicklung durchgeführt. Es können beispielsweise Quellcode-Formatierungs-Richtlinien hinterlegt werden, so dass automatisch Vorschriften eingehalten werden.

Listing 5.2 Beispiel für falsche Einrückung in *Java* in Anlehnung an [211]

```
1 int x = 1; int y = 0;
2 if(x == 1)
3         if(y == 10)
4                 x = -10;
5 else
6         x = 10;
```

5.2.3 Erkennung von Quellcode-Duplikaten

Da Quellcode-Duplikate die Stabilität von Anwendungen einschränken können, sollten diese vermieden werden [152]. Werkzeuge wie *PMD* können Duplikate in Quellcode mit einem *Copy-Paste-Detector* erkennen und melden. Duplizierter Quellcode ist erkennbar, nicht jedoch duplizierte Funktionalität [159].

5.2.4 Erkennung von Komponenten mit bekannten Schwachstellen

Nach dem *Data Breach Investigations Report* [236, S. 15] wurden 99.9% der ausgenutzten gemeldeten Schwachstellen von Anwendungen bereits über ein Jahr vorher im *Common Vulnerabilities and Exposures* (*CVE*) gelistet. Entsprechend sollte die Prüfung auf Bibliotheken mit bekannten Schwachstellen besonders priorisiert und regelmäßig durchgeführt werden, wie auch im *Maßnahmenkatalog Organisation* der IT-Grundschutz-Kataloge vom BSI [81, S. 2070] empfohlen.

Abhängigkeitsmanagement-Systeme wie *bower* (für *JavaScript*), *Maven* (für *Java*), *npm* (für *JavaScript*) und *composer* (für *PHP*) definieren abhängige Bibliotheken und deren Version in einer Konfigurations-Datei, z.B. *pom.xml* in Maven.

Werkzeuge wie der *OWASP Dependency Check* [162] untersuchen zusätzlich Dateien im Projektverzeichnis. Dabei erfolgt eine Zuordnung von Fragmenten, wie z.B. dem Dateinamen oder der Einbindung einer Bibliothek im Quelltext, zu Bibliotheken inklusive ihrer Version [187]. In *Java* kann z.B. der Name der Bibliothek *commons-collections-3.2.jar* der Bibliothek *Apache Commons Collections™* [5] der Version 3.2 zugeordnet werden. Die Informationen aus Fragmenten und Abhängigkeitsmanagement-Systemen werden mit der *National Vulnerability Database* (kurz *NVD*) abgeglichen und bei einer Übereinstimmung inklusive Kritikalität und *CVE*-Identifizierung gemeldet [86].

Es können serverseitig je nach Programmiersprache unterschiedliche Werkzeuge genutzt werden. Dabei sind klientenseitige Prüfungen via *HTTP* für *JavaScript* als Browser-Plugin und als *Plugin* für Scanner wie *Zap* möglich. Zusätzlich können Abhängigkeitswerkzeuge in die Entwicklungsumgebung integriert oder auf der Kommandozeile gestartet werden um auf Dateisystem-Ebene zu prüfen, was gegenüber der klientenseitigen Prüfung den Vorteil bietet, dass der gesamte Quellcode auf dem Dateisystem geprüft wird. Extern eingebundenes *JavaScript* wird dadurch nicht geprüft. Da verschleierter Quellcode für statische Analysewerkzeuge ein Erschwernis darstellt, sollte der Quellcode ohne Verschleierung bei der statischen Analyse vorliegen [244]. Für *JavaScript* kann *Retire.js* [182] und für *PHP* sowie *Java* der *OWASP Dependency Check* genutzt werden.

Werkzeuge wie der *OWASP Dependency Check* untersuchen zusätzlich zu direkten Abhängigkeiten transitive Abhängigkeiten, also von Bibliotheken eingebundene Bibliotheken. Wird eine Schwachstelle in einer Bibliothek mit transitiver Abhängigkeit gefunden, kann ein Überblick mit einem Abhängigkeitsgraphen,

z.B. erzeugt mittels dem *Apache™ Maven Dependency Plugin* [44], gewonnen werden.
Werkzeuge wie der *SensioLabs Security Cheker* [199] fragen bei einem Webservice von *SensioLabs* nach Schwachstellen für eine verwendete Bibliothek und deren Version an. Da alle Abhängigkeiten so an Dritte zum Testen auf Komponenten mit bekannten Schwachstellen übermittelt werden, sollte abgewogen werden ob der Nutzen der Information über Komponenten mit bekannten Schwachstellen überwiegt oder das Risiko der Übermittlung von Abhängigkeiten eines Projekts an Dritte zu hoch ist.

5.2.5 Datenflussanalyse

Während bei *String Matching* Algorithmen und Abhängigkeitsprüfen mit dem Schwerpunkt Sicherheit das Vorhandensein einer Art von Schwachstelle durch vorher definierte Regeln bewiesen wird, wird bei der Datenflussanalyse bewiesen, dass eine spezielle Art von Schwachstelle nicht vorhanden ist. Bei der Datenflussanalyse wird von Programmpunkt zu Programmpunkt der Fluss von Eigenschaften und Variablenwerten analysiert. Datenflussanalysen können so beispielsweise die Abwesenheit von Laufzeitfehlern erkennen. [220, S. V]
Bei Webanwendungen wird insbesondere die Maskierung von Daten bei der Ausgabe geprüft [144, S. 9]. Im Wesentlichen werden die Phasen *Frontend* und *Backend* durchlaufen. In der Phase *Frontend* wird der Quellcode analysiert und daraus ein abstrakter Syntaxbaum (Englisch *abstract syntax tree*, AST) erstellt. Während Quellcode i.d.R. durch Einrückungen formatiert ist um eine Struktur zu schaffen, wird beim abstrakten Syntaxbaum die syntaktische Struktur in einem Baum dargestellt. Dafür wird eine lexikalische Analyse, d.h. der Quellcode wird normalisiert und *Tokens* erkannt und klassifiziert. *Tokens* können u.a. Schlüsselwörter, Bezeichner, Zahlen und Operatoren sein. Anschließend erfolgt eine syntaktische Analyse, in welcher die Struktur des Programms herausgefunden wird. Ein Parser unterscheidet z.B. zwischen Ausdrücken, Anweisungen, Deklarationen usw. und überführt diese in einen Syntaxbaum. Abschließend erfolgt die semantische Analyse, bei welcher Bedingungen des syntaktischen Syntaxbaumes durch statischen Informationen, wie die Typbestimmung bei Operationen, getestet wird. Beispielsweise muss bevor eine Variable verwendet wird, diese deklariert worden sein. [144, S. 10ff.]
FindBugs [61] nutzt eine intraprozedurale Datenflussanalyse und findet so beispielsweise Laufzeitfehler wie in Listing 5.3 [134, 133]. In dem Beispiel wird eine *Nullpointer*-Ausnahme geworfen, da x in Zeile 2

Listing 5.3 Beispiel für einen Laufzeitfehler in *Java* in Anlehnung an [211]

```
1 String x = "I am initialized";
2 x = null;
3 x.length();
```

auf *null* gesetzt wird. Wird die Ausnahme nicht gefangen, so beeinträchtigt diese das Laufzeitverhalten der Webanwendung und die Verfügbarkeit ist ggf. beeinträchtigt. *FindBugs* findet viele Defekte nicht um die Analyse möglichst einfach zu halten und um möglichst wenig falsch Positive zu genieren [61]. Ein Beispiel ist in Listing 5.4 gegeben. Obwohl die gleiche Variable für die beiden Bedingungen genutzt wird, wird die *Nullpointer*-Ausnahme im Fall *condition* ist wahr nicht erkannt. Häufig sind solche Bedingungen unabhängig von einander, so dass der Pfad der Ausführung nicht einfach bestimmt werden kann [61]. Deshalb prüft *FindBugs* nur garantierte Pfade [61]. Entsprechend ist *FindBugs unsound*.

Listing 5.4 Beispiel für einen falsch Negativen in *FindBugs*

```
1 public testConditons(Boolean condition) {
2         String x = "I am initialized";
3         if(condition)  {
4                 x = null;
5         }
6         if(condition) {
7                 x.length();
8         }
9 }
```

Werkzeuge wie *FindBugs* können in die Entwicklungsumgebung integriert werden, ein Bildschirmfoto einer Entwicklungsumgebung ist im Anhang in Abb. L.4 auf Seite 205 zu finden. *FindSecurityBugs* [11], eine Erweiterung für *FindBugs* findet Aufrufe von unsicheren Methoden. Beispielsweise die Benutzung der unsicheren Klasse `java.util.Random`, das ungeprüfte Einlesen von Dateipfaden, oder die Nutzung von einem unsicherem Hashalgorithmus wie MD5.
Während *FindBugs Bytecode* analysiert, analysiert *PMD Java*-Quellcode. Dabei werden z.B. Methoden gefunden, welche ein internes Feld einer Klasse *a* an eine andere Klasse *b* zurück geben. Als Gegenmaßnahme wird angegeben, die Methode `clone()` auf ein Feld zu nutzen, damit das Feld der Klasse *a* nicht durch Methoden der Klasse *b* verändert werden kann.
Bei dynamisch typisierten Sprachen wie *JavaScript* können sich Typen einer Variable während der Ausführung ändern, was für die Datenflussanalyse eine Herausforderung darstellt [130]. Weiter erschwert das Parsen und Ausführen von Zeichenketten zur Laufzeit mittels der Funktion `eval()` die Analyse [130]. Mittels dem Werkzeug *ESLint* [254] lässt sich *JavaScript*-Quellcode auf sicherheitskritische Operationen mittels AST hin untersuchen [40]. Beispielsweise wird getestet, ob die Funktion `eval()` aufgerufen wird. Der Aufruf von `eval()` stellt ein hohes Sicherheitsrisiko dar, da es beliebige Zeichenketten mit beliebigen Inhalten wie Quellcode mit Datendeklarationen ausführt [233, S. 11]. *ESLint* selbst ist nicht auf Sicherheit spezialisiert, die Erweiterungen *eslint-plugin-scanjs-rules* [77] und *eslint-plugin-no-unsafe-innerhtml* [76] jedoch schon. Die Erweiterung *eslint-plugin-scanjs-rules* testet z.B. auf *XSS* beim Funktionsaufruf `setTimeout()` oder beim Schreiben in den *DOM*-Baum via `document.write()`.
Werkzeuge wie *Pixy* [143] für *PHP* lesen nur eine einzelne *PHP*-Datei als Eingabe ein. Bei objektorientierter Programmierung wird fffür eine Unterteilung in Klassen und Methoden geworben, deshalb ist das Testen moderner *PHP*-Webanwendung mit *Open Source* Werkzeugen schwierig.

5.2.6 Übersicht über ausgewählte Werkzeuge

In Tabelle 5.3 ist eine Auswahl von praxistauglichen Werkzeugen zum statischen Testen vorgestellt. Werkzeuge, welche nur zur Validierung von Programmkonventionen genutzt werden, sind nicht aufgeführt. PS steht für „Programmiersprache“ und „BFP“ für die Behandlung falsch Positiver.
Besondere Herausforderung ist die genutzte Methode der Projekte herauszufinden und zu kategorisieren. In Tabelle I.2 im Anhang ist eine erweiterte Übersicht über alle getesteten Werkzeuge.

PS	Name	Methode	Integration	Beschreibung	Letzte Änderung	BFP
.NET, Java, Scala	OWASP Dependency Check [162]	Abhängigkeiten	Ant, Jenkins, Konsole, Maven	Erkennung veralteter Bibliotheken.	April 2016	Ja
Java, Scala	FindBugs [61] mit FindSecurity-Bugs [11]	Stilanalyse, Datenfluss-analyse	GUI, IDE, Jenkins, Konsole, Maven	Erkennung von Laufzeit- und Logikfehlern und die Nutzung gefährlicher Methoden.	April 2016	Ja
Java-Script	ESLint mit den Plugins: eslint-plugin-scanjs-rules *[77]* und eslint-plugin-no-unsafe-innerhtml *[76]*	Stilanalyse	Konsole, IDE	Erkennung nicht maskierter Ausgaben.	Januar 2016	Nein
Java-Script	retire.js [182]	Abhängigkeiten	Browser, Konsole, Plugin für Schwach-stellens-canner	Erkennung veralteter Bibliotheken via HTTP oder auf dem Dateisystem.	April 2016	Ja
PHP	SensioLabs Security Checker [199]	Abhängigkeiten	Konsole	Erkennung veralteter Bibliotheken über *composer*.	November 2015	Nein

Tabelle 5.3: Auswahl von praxistauglichen Werkzeugen zum statischen Testen von Webanwendungen

5.3 *Interactive Application Security Testing*

Das Forschungsinstitut Gartner hat 2011 den Bedarf an einen Hybrid von statischen und dynamischen Methoden erkannt, welche es *Interactive Application Security Testing* (kurz *IAST*) genannt hat [165]. Dabei werden *DAST*-Methoden genutzt um eine Anwendung anzugreifen und gleichzeitig die Anwendung instrumentiert [164]. Erste Ansätze der Laufzeit-Instrumentierung wurde bereits 2004 in [135] vorgestellt. Durch *IAST* ist es möglich eine angeforderte *URL* mit durch die Webanwendung aufgerufenen Methoden und Klassen in Beziehung zu setzen [198]. Hauptvorteil ist die Reduktion von falsch positiv Meldungen, beispielsweise bei Blind SQL-Injections. Via *DAST* kann ein SQL-Injection Test durchgeführt werden und gleichzeitig kann serverseitig eine Validierung der *SQL*-Anfragen erfolgen [136]. Auch zeitverzögerte Angriffe wie persistentes *XSS* oder *Second Order SQL Injections* sind einfacher aufzudecken [198].
IAST reduziert die falsch Negativen, ist allerdings schwer zu implementieren [212]. Da *IAST* eine neue Methode ist konnten keine *Open Source*-Werkzeuge gefunden werden. Entsprechend wird nicht weiter auf diesen Bereich eingegangen, auch wenn die Methode starkes Forschungspotential hat.

5.4 Testen der Infrastruktur

Die vorgestellten *DAST*- und *SAST*-Werkzeuge testen hauptsächlich die Anwendung auf Sicherheit. Es sind unterschiedliche *Open Source* Werkzeuge zum dynamischen und statischem Testen der Infrastruktur auf dem Markt.

Um so mehr virtualisierte Umgebungen genutzt werden, um so mehr Betriebssysteme müssen auf Komponenten mit bekannten Schwachstellen geprüft werden. Das Werkzeug *apticron* [29] wird klassisch von System-Administratoren verwendet und liefert Informationen über anstehende Aktualisierungen. Dabei wird eine E-Mail mit allen Aktualisierungen versendet. So wird ein System-Administrator mit Meldungen zu Aktualisierungen „überflutet“ und kann diese nur schwer priorisieren. Das Werkzeug *Vuls* [39] dagegen gleicht von einem Linux-Betriebssystem Pakete und deren Version mit Einträgen in der *NVD* ab. Wird eine Übereinstimmung gefunden, wird diese mit Angabe der Kritikalität ausgegeben. Die Ausgabe kann entsprechend nach Kritikalität gefiltert werden.

Da Dienste wie der *Apache*™-Webserver nicht nur in Produktionsumgebungen, sondern auch in Entwicklungsumgebungen genutzt werden, ist die Konfiguration nicht für ein Produktionssystem optimiert. Beispielsweise wird in der Standard-Installation eine *HTTP*-Kopfzeile mit der Version des Webservers ausgeben. Dies wird über den Parameter *ServerSignature* aktiviert oder deaktiviert.

Häufig können Sicherheitsexperten nicht Entwickeln, so dass Anforderungen in Form von Textdokumenten, *Stories* oder Modellen definiert werden. Welche der Entwickler anschließend umsetzt. Dabei können Missverständnisse entstehen oder Anforderungen vergessen werden. Eine Verbesserung des Prozesses kann verhaltensgetriebene Softwareentwicklung (Englisch *Behavior Driven Development*) bringen. Hier werden Anforderungen vor der Entwicklung textuell in einer universellen Sprache erfasst [224]. Das Werkzeug *InSpec* [18] bietet die Möglichkeit System-Anforderungen mittels verhaltensgetriebener Softwareentwicklung zu definieren. Ein Beispiel ist in Listing 5.5 gegeben. Hier wird getestet, ob dem Parameter *ServerSignature* der Wert *Off* zugewiesen ist.

Listing 5.5 Regel zum Testen deaktivierter Versionsnummern im Webserver von *Apache*™

```
1 describe apache_conf do
2   its('ServerSignature') { should eq 'Off' }
3 end
```

InSpec bietet Regeln für unterschiedliche Anwendungen. Dabei wird jedoch beispielsweise der in Listing 5.5 gezeigte Fall nicht abgedeckt. Entsprechend ist manuelle Optimierung der Regeln notwendig. Ist eine Regel einmal definiert, kann diese für mehrere Systeme genutzt werden. Durch die Anpassung oder Deaktivierung von Regeln werden falsch Positiv-Meldungen eliminiert. Da bei *DevOps* Entwickler auch mit in die Infrastruktur einbezogen werden und Änderungen, z.B. mittels *Docker* durchführen können, ist dieser Ansatz besonders unterstützend. Die Entwicklungsumgebung kann produktionsnah mittels *Docker* oder *vagrant* aufgesetzt werden. Wird eine Änderung an der Konfiguration durchführt, können Sicherheits-Anforderungen direkt validiert werden.

Eine Auswahl von Werkzeugen ist in Tabelle 5.4 aufgeführt.

Name	Beschreibung	Letzte Änderung	Praxis-tauglich	Behandlung von falsch Positiven
Docker Bench for Security [119, S. 76]	Test der Docker-Konfiguration	April 2016	Ja	Nein
THC Hydra [36]	Test auf Passwort-Stärke von SSH-Benutzern	April 2016	Ja	/
Iniscan [97]	Test der php.ini	März 2016	Ja	Nein
InSpec [18]	Test der Server-Konfiguration	April 2016	Ja	/
Lynis [93]	Test der Server-Konfiguration und von einem *Dockerfile*	April 2016	Ja	Nein
openVAS [28]	Test von Server-Systemen	April 2016	Ja	Ja
testssl.sh [247]	Test der TLS/SSL-Verschlüssellung	April 2016	Ja	Nein
Vuls [39]	Test auf Komponenten mit bekannten Schwachstellen	April 2016	Ja	Nein

Tabelle 5.4: Werkzeuge zum Testen der Infrastruktur

5.5 Konsolidierung und Prüf-Intensität

Nach Ayewah et al. [62] muss nicht jedes gefundene Problem behoben werden. Insbesondere beim Testen auf Sicherheit können durch Stilanalysen gefundene Probleme ggf. vernachlässigt werden.

Um falsch Positive zu minimieren, können unterschiedliche Werkzeuge genutzt werden, ergeben zwei Werkzeuge die gleichen Ergebnisse, steigt die Wahrscheinlichkeit von einem richtig Positiven, so der *Chief Strategy Officer* eines auf statische Tests spezialisiertes Unternehmen (siehe Anhang B.10 auf Seite 117).

Zur Aggregation von Ergebnissen kann beispielsweise das in der Grundversion kostenfreie Werkzeug *ThreadFix* [102] genutzt werden. Erweiterte Funktionen wie mehrere Benutzer sind kostenpflichtig.

Auch sollten insbesondere bei der Einführung eines Werkzeugs für ein bestehendes Projekt zunächst nur als hoch eingestufte Risiken behandelt werden, da die Masse an Meldungen sonst zu groß wird. Wird ein falsch Positiver gemeldet, so sollte auch die dazugehörige Regel des Werkzeugs geprüft und ggf. deaktiviert werden. Die Eingrenzung auf für die Anwendung wichtige Bereiche und den Ausschluss von Bibliotheken bei *String Matching* Algorithmen und Datenflussanalysen reduziert die Anzahl der gemeldeten potentiellen Risiken ebenfalls. Dabei sollten kontinuierlich weitere Bereiche hinzugefügt werden, um die Abdeckung stetig zu steigern.

Teams sollten nach Möglichkeit nur Meldungen für Risiken erhalten, welche auch durch diese behoben werden können. In kleinen Organisationen sind häufig klientenseitige Teams und serverseitige Teams vorhanden. Hier können Meldungen statischer Werkzeuge direkt Teams adressieren. In größeren Organisationen sind Teams häufig für spezielle Funktionen, im Quellcode durch Pakete getrennt, zuständig. Diese können entsprechend Teams zugeordnet werden. Bei einer *Microservice*-Architektur sind Teams häufig für einzelne *Microservices* zuständig, so dass hier eine Zuordnung ebenfalls erfolgen kann.

Unnötige Testes bei universellen Werkzeugen wie *Zap* können deaktiviert werden. Nutzt eine Anwendung ausschließlich eine *NoSQL*-Datenbank und keine SQL-Datenbank, so muss auf *SQL-Injection* nicht getestet werden.

Sollte die Anzahl der gemeldeten Risiken nach Anwendung der vorgestellten Maßnahmen nach wie vor zu groß sein, kann die Regel eingeführt werden, dass geänderte Module (beispielsweise Klassen) bei Änderung getestet und alle Meldungen getestet werden. Wird ein richtig Positiver identifiziert, so wird dieser

behoben. Wird ein falsch Positiver identifiziert, so wird dieser als falsch Positiv markiert. Meistens nutzen Werkzeuge dafür eine spezielle Datei, in welcher die falsch Positiven definiert werden. Falsch Positive können für *FindBugs* und für das *Plugin FindSecurityBugs* über Anmerkungen (Englisch *annotations*) direkt im Quellcode markiert werden.
Grundsätzlich bietet es sich an gefundene Risiken zu visualisieren um Trends zu identifizieren. Das *CI*-System *Jenkins* bietet dafür das *Analysis Collector Plugin* [1], welches z.B. vom *OWASP Dependency Check Plugin* genutzt. Hier wird der Trend der letzten Tests aufgezeigt und in die Kategorien Niedrig, Mittel und Hoch aufgeteilt. Zusätzlich kann definiert werden, ab wie vielen Meldungen für die einzelnen Kategorien die Erzeugung gestoppt werden soll.

5.6 Einsatzpunkte für Sicherheitstests von Webanwendungen

DevOps benötigen möglichst frühe Rückmeldung. Entsprechend sollten Sicherheitstests nach Möglichkeit während der Entwicklung stattfinden, diese jedoch nicht beeinträchtigen [202, S. 28].
Der in Abb. 4.6 gezeigte Erzeugungs- und Verteilungsprozess ist in Abb. 5.2 um das Modul- und Integrationssystem und die jeweiligen empfohlenen Tests angereichert. In Klammern ist jeweils ein Beispiel für ein Werkzeug angegeben.

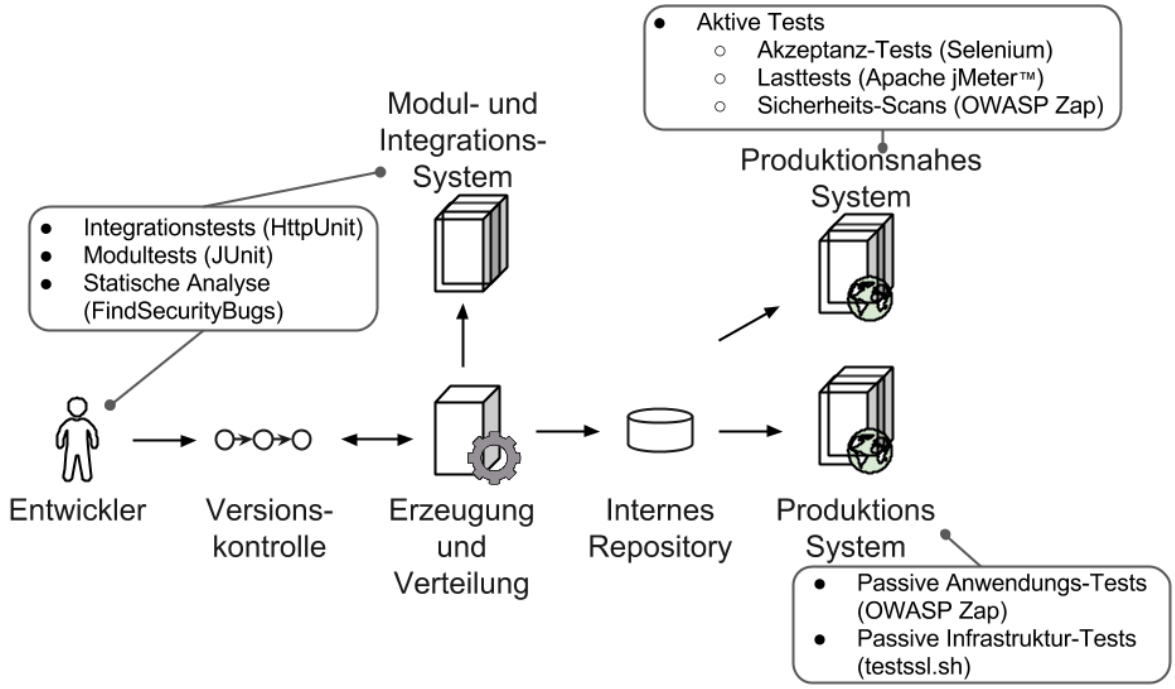


Abbildung 5.2: Einsatzorte von Tests

5.7 Evaluierung dynamischer, statischer und interaktiver Sicherheits-Tests

Der *OWASP Code Review Guide 2.0* vergleicht die Effizient von automatischen Quellcode-Scannern (*SAST*), automatischen Web-Schwachstellenscannern (*DAST*), manuellen Penetrations-Tests und manuelle Quellcode-Reviews für die *OWASP Top Ten 2013*, dargestellt in Tabelle 5.5. Bei manuellen

Risiko	SAST	DAST	Manueller Penetrations-Test	Quellcode-Review
Injection	Teilweise	Gute Chance	Gute Chance	Gute Chance
Fehler in Authentifizierung und Session-Management	Keine	Gute Chance	Gute Chance	Gute Chance
Cross-Site Scripting	Keine	Gute Chance	Gute Chance	Gute Chance
Unsichere direkte Objektreferenzen	Keine	Gute Chance	Gute Chance	Teilweise
Sicherheitsrelevante Fehlkonfiguration	Keine	Teilweise	Teilweise	Gute Chance
Verlust der Vertraulichkeit sensibler Daten	Teilweise	Teilweise	Gute Chance	Gute Chance
Fehlerhafte Autorisierung auf Anwendugsebene	Keine	Teilweise	Gute Chance	Gute Chance
Cross-Site Request Forgery	Keine	Gute Chance	Gute Chance	Gute Chance
Nutzung von Komponenten mit bekannten Schwachstellen	Teilweise	Teilweise	Teilweise	Gute Chance
Ungeprüfte Um- und Weiterleitungen	Keine	Gute Chance	Gute Chance	Gute Chance

Tabelle 5.5: Vergleich von Sicherheits-Test-Methoden für Webanwendungen nach [228]

Penetrations-Tests und Quellcode-Reviews wird davon ausgegangen, dass Wissen über die Erkennung von Schwachstellen vorhanden ist. Dabei wir deutlich, dass automatische Werkzeuge hilfreich sind, jedoch manuelle Penetrations-Tests sowie manuelle Quellcode-Reviews mit Experten nicht ersetzen, aber die tägliche Arbeit des Experten erleichtern können.

Für *DAST*-Methoden ist nur limitiertes Wissen über die Anwendung notwendig. Es kann jede Webanwendung, egal welche Technologie verwendet ist, getestet werden. Unterschiedliche Technologien kann auch zu ineffizient führen, da je nach Technologie unterschiedliche Schwachstellen unterschiedlich stark auftreten und damit unterschiedlich stark geprüft werden sollten [100]. *DAST* ist limitiert, beispielsweise sind zeitversetzte Angriffe wie persistentes *XSS* sind schwer zu erkennen. Ferner sind *DAST*-Methoden sehr zeitintensiv. Häufig existiert nur ein kleines Test-Fenster, welches schnell aufgebraucht ist. Für komplexe Technologien und Protokolle wie *Flash*, *HTML* 5, *JavaScript* und *JSON* wird es schwieriger Schwachstellen zu erkennen, entsprechend wird ein voll funktionsfähiger Browser benötigt. Auch ist das Testen auf den Verlust der Vertraulichkeit sensibler Daten, beispielsweise die Art der Speicherung eines Passworts, mit *DAST* nur schwer möglich [198]. Bei automatischen *DAST*-Werkzeugen werden viele falsch Positive gemeldet. Zusätzlich ist die Anzahl der falsch Negativen hoch.

Eine hohe Abdeckung zu erreichen kann je nach Webanwendung mehrere Stunden dauern. Ein Schwachstellen-Scan kann, insbesondere bei hoher Prüfintensität, ebenfalls mehre Stunden dauern. Entsprechend sollten solche zeitintensiven Tests nicht sequentiell im Erzeugungs- und Verteilungsprozesses integriert, sondern parallel dazu laufen.

Bei *SAST* dagegen überwiegen die falsch Positiven, i.d.R. wird hier eine sehr hohe Anzahl gemeldet, so dass stark aussortiert werden muss. Je nach Programmiersprache kann *SAST* ggf. nicht kompilierte Bibliotheken testen. Auch ist die Laufzeitumgebung nicht vorhanden, so dass Prüfungen wie Fehlerbehandlung, System-Konfiguration, Authentifizierung nicht durchgeführt werden können.

Bei der *Crawljax* der aktuellen Version 3.5 wird das *JavaScript*-Event *mouseover* nur bedingt unterstützt.

Wird ein Formular-Element wie das *select*-Tag ohne Formular mit einem Klick-*Event-Listener* genutzt, wird dieses nicht von *Crawljax* angeklickt. Fragmentbezeichner der *URL* werden zwar von Crawljax unterstützt, jedoch konnte keine Funktion zur Berücksichtigung von Fragmentbezeichnern in *Zap* gefunden werden. Dies zeigt, dass automatische dynamische Prüfung zwar möglich sind, jedoch manuell die Abdeckung geprüft werden sollte.

Während durch *String Matching* Algorithmen und Datenflussanalysen gefundene Risiken wahrscheinlich schnell behoben werden können, kann die Aktualisierung einer Bibliothek vergleichsweise hohen Aufwand bedeuten. Beim „OWASP Stammtisch Hamburg“ am 17.05.2016 wurde von Experten angemerkt, dass Schwachstellen in von einer Webanwendung genutzten Bibliothek häufig von externen Angreifern nicht erkannt und ausgenutzt werden (siehe Anhang auf Seite 117). Schwachstellen wie *XSS*, welche durch Datenflussanalysen aufgedeckt werden, können schnell erkannt und ausgenutzt werden. Das Beheben der einer *XSS*-Schwachstelle kann insbesondere bei objektorientierter Entwicklung sehr viel schneller als das Austauschen oder Aktualisieren einer Bibliothek erfolgen.

Die Bibliothek *Apache Commons Collections*™ 3.2 enthält eine Schwachstelle, so dass bei Deserialisierung nicht vertrauenswürdiger Daten die resultierenden Daten nicht validiert werden [118, 9]. Es ist zwar für die Bibliothek nutzende Produkte wie den *Oracle WebLogic Server* [7, 8] ein *CVE*-Identifikator erzeugt, nicht jedoch für die Schwachstelle selbst. So kann die Bibliothek vom *OWASP Dependency Check* nicht als Komponente mit bekannter Schwachstelle gefunden werden. Es bleibt entsprechend ein Risiko, dass eine Bibliothek keine *CVE*-Identifikator erhalten hat und damit nicht gefunden wird.

Beide Methoden haben Vor- und Nachteile, weshalb i.d.R. eine Komposition aus beiden Methoden zur Erhöhung der Qualität der Sicherheitstests genutzt wird [63, 172].

Kapitel 6

Generisches *DevOps* Sicherheits-Reifegradmodell (GDOSR)

Eine Strategie von *DevOps* ist die kontinuierliche Verbesserung. *DevOps*-Strategien befolgende Teams haben i.d.R. keine Expertise im Bereich Sicherheit, deshalb kann eine Herausforderung die kontinuierliche Verbesserung der Sicherheit darstellen. Ein Reifegradmodell bietet eine Anleitung zur schrittweisen Erhöhung der Sicherheit sowie Visualisierung des Implementierungs-Grades. In diesem Kapitel wird beschrieben, welchen Anwendungsbereich das Modell hat, wie es entworfen wurde, beispielhaft eine Dimension beschrieben und die Evaluierung mit Experten vorgestellt.

6.1 Definition des Anwendungsbereichs

Durch das generische *DevOps*-Sicherheits-Reifegradmodell (kurz GDOSR) werden *DevOps*, Sicherheitsexperten sowie das Management in die Lage versetzt, den aktuellen Implementierungs-Grad in Bezug auf die Sicherheit der Webanwendung auf einen Blick festzustellen. Das Modell gibt Handlungsempfehlungen in Form von Implementierungspunkten (kurz IP) für die nächsten durchzuführenden Schritte. Im Rahmen dieser Arbeit wird das GDOSR von Timo Pagel entwickelt und geprüft. Die Evaluierung findet sowohl durch Sicherheitsexperten, Entwickler, System-Administratoren als auch durch IT-Entscheider statt.
Schneider lässt im *Security DevOps Maturity Model* Sicherheit in den *DevOps*-Dimensionen „Erzeugung- und Verteilung“, „Informationsgewinnung“, „Infrastruktur“ und „Organisation und Kultur“ außen vor. Weiterhin ist das *Security DevOps Maturity Model* nicht nachhaltig dokumentiert. Entsprechend wird ein generisches Modell benötigt und hier vorgestellt, welches die Dimensionen berücksichtigt.
Das GDOSR ergänzt bestehende Reifegradmodelle zur Einführung von Sicherheit und Erfassung des Sicherheitsniveaus wie das *OWASP Software Assurance Maturity Model*.

6.2 Entwicklungsphase „Entwurf“

Das GDOSR besteht aus fünf Dimensionen, die Unter-Dimensionen haben können. Insgesamt sind 98 IPe erfasst. Die Dimensionen und Aufgaben sind wie folgt:

- Erzeugung und Verteilung
  - Erzeugung: Härtung der Erzeugung der Artefakte der Webanwendung
  - Verteilung : Härtung der Verteilung der Artefakte der Webanwendung
- Informationsgewinnung
  - Überwachung und Metriken: Integration von Überwachung und Metriken
  - Protokollierung Integration von Protokollierung
- Infrastruktur: Härtung der Infrastruktur
- Kultur und Organisation: Integration von Sicherheit in den agilen Entwicklungszyklus
- Test und Verifizierung
  - Dynamische Tiefe: Feststellung des Sicherheitsniveaus der Webanwendung und ihrer Systeme durch *DAST*
  - Statische Tiefe: Feststellung des Sicherheitsniveaus der Webanwendung und ihrer Systeme durch *SAST*
  - Test-Intensität: Anpassung der Intensität der Tests
  - Konsolidierung: Aggregation und Behandlung von Alarmen
  - Anwendungstest: Verifizierung der Implementierung
  - Infrastruktur: Feststellung des Sicherheitsniveaus der Infrastruktur

Es wird möglichst vermieden das GDOSR in Abhängigkeit zu Technologien oder Werkzeugen zu beschreiben, damit es eine möglichst lange Gültigkeit hat.
In GDOSR sind vier Ebenen vorhanden. Ausgangspunkt ist kein Verständnis für Sicherheit. Auf der 1. Ebene ist grundlegende Sicherheit in und durch *DevOps*-Strategien implementiert. Ist Ebene 2 implementiert, hat das Startup erweitertes Verständnis für Sicherheit in und durch *DevOps*-Strategien erlangt. Auf der 3. Ebene wird durch die vollständige Implementierung ein hohes Verständnis für Sicherheit in und durch *DevOps*-Strategien erlangt. Ist Ebene 4 implementiert, ist ein sehr hohes Verständnis für Sicherheit in und durch *DevOps*-Strategien erlangt.

6.3 Entwicklungsphase „Füllung“

6.3.1 Vorgehen der Entwicklung

Zunächst wurde das GDOSR in einer Tabelle (siehe http://gdosr.timo-pagel.de/ oder Anhang K.1 auf Seite 159), einem Dokument und einer Präsentation gepflegt. In der Tabelle wurden den Dimensionen Implementierungspunkte (kurz IPe) zugeordnet. In dem Dokument wurden Entscheidungen dokumentiert und in der Präsentation die „Schwere der Implementierung“ und der „Nutzen“ für Sicherheit grafisch dargestellt. Wurde ein IP umbenannt, musste in allen drei Dokumenten eine Anpassung erfolgen. Dieses Verfahren stellte sich als zu umständlich heraus.

Ein Werkzeug zur Erstellung von Reifegradmodellen konnte nicht gefunden werden. Entsprechend wurde eine Webanwendung[1] erstellt, in welcher IPe definiert und Dimensionen zugeordnet werden können. Jeder IP hat zusätzlich eine Risikobeschreibung und die Beschreibung einer Gegenmaßnahme. Weiter ist der „Nutzen“ und die „Schwere der Implementierung“ angegeben. Während der „Nutzen“ direkt geschätzt wird, ist die „Schwere der Implementierung“ aufgeteilt in die „Benötigte Zeit“ mit 50 Prozent Gewichtung, das „Benötigte Wissen“ mit 25 Prozent Gewichtung und den Punkt „Benötigte Ressourcen“ mit 25 Prozent Gewichtung.

Die „Benötigte Zeit“ zur Implementierung beinhaltet die Einführung im Rahmen von Schulungen, die Diskussion mit Kollegen und die technische Implementierung. Das „Benötigte Wissen“ zeigt auf, wie viele Bereiche der IT abgedeckt werden müssen und mit welcher Expertise. Beispielsweise muss zur Einführung des IPes „Kontrollierte Netzwerke für virtuelle Umgebungen“ Expertise im Bereich IT-Systemadministration und Netzwerk-Sicherheit vorhanden sein. Unter dem Punkt „Benötigte Ressourcen“ sind Systeme zu verstehen, welche für die Durchführung benötigt werden. Die „Benötigte Zeit“ hat eine höhere Wertigkeit, da es häufig einfacher beziehungsweise kostengünstiger ist einen neuen Server zu bestellen, als an einer mehrtägigen Implementierung zu arbeiten.

Dabei reicht die Skala für den „Nutzen“ und die Unterpunkte der „Schwere der Implementierung“ von „Sehr gering“ mit Wertigkeit 1 bis „Sehr hoch“ mit Wertigkeit 5. Es wird auf die Angabe von konkreten Tagen zur Implementierung verzichtet. Eine konkrete Angabe der Tage kann die implementierenden Personen unter starken negativen Zeitdruck setzen, so Stefan Rieber (siehe Experten-Interview im Anhang B.4 auf Seite 114). Negativer Zeitdruck kann entstehen, wenn das Management konkrete Tage zur Implementierung durch das GDOSR vorgelegt bekommt und die implementierenden Personen mehr Zeit benötigen. Weiterhin hängt die „Schwere der Implementierung“ stark von dem Wissen in den IT-Bereichen Entwicklung, System-Administration und Sicherheit ab, so dass eine generelle Schätzung in Tagen nur mit hoher Ungenauigkeit möglich ist.

Ferner sind die durch einen IP gewährleisteten Sicherheitseigenschaften aufgeführt und beschrieben. Bei der Dimension „Kultur und Organisation“ wird auf die Angabe von Sicherheitseigenschaften verzichtet, da durch die Einführung von kulturellen und organisatorischen Maßnahmen alle Sicherheitseigenschaften gewährleistet werden können. Optional ist ein Hinweis zur Implementierung und ein Kommentar.

Die Webanwendung bietet eine Tabelle mit allen IPen, eine Detailansicht für einen IP, eine Identifizierung des Implementierungs-Grades als Diagramm durch Markierung einzelner IP, eine grafische Darstellung des „Nutzens“ und der „Schwere der Implementierung“ und eine Darstellung der Abhängigkeiten zwischen IPen.

Bei dem Diagramm zum Anzeigen von „Nutzen“ und „Schwere der Implementierung“ kann einerseits die Schwere des IPes selbst, andererseits die Summe aus der „Schwere der Implementierung“ des IPes selbst und die „Schwere der Implementierung“ abhängiger IPe abgelesen werden.

Beispielsweise ist, um eine „Statische Analyse für alle Bereiche“ durchführen zu können, die Implementierung einer „Statischen Analyse für wichtige serverseitige Bereiche“ sinnvoll. Dadurch reduziert sich die Implementierungszeit, da Schulungszeit reduziert werden kann und eine technische Implementierung bereits erfolgt ist.

Zudem können die Dimensionen im Diagramm gefiltert werden. So kann ein einfacher Vergleich des

[1] Der Quellcode ist unter https://github.com/wurstbrot/generic-DevOps-Security-MaturityModel unter veröffentlicht.

„Nutzens“ und der „Schwere der Implementierung“ von IPen aus unterschiedlichen Dimensionen erfolgen. Der „Nutzen“ und die „Schwere der Implementierung“ sind dabei Hauptargument für die Zuordnung des IP zu den Ebenen. Wobei die Abhängigkeiten zwischen IPen auch berücksichtigt werden. Ein zentraler IP ist „Definierter Erzeugungsprozess“, so dass dieser Ebene 1 zugeordnet ist.
Tabelle 6.1 gibt eine Übersicht über die Anzahl der IPe pro Ebene und Dimension.

Unter-Dimension	Ebene 1	Ebene 2	Ebene 3	Ebene 4	Anzahl pro D
Erzeugung	1	1	3	2	7
Verteilung	1	2	3	2	8
Überwachung und Metrik	1	2	5	4	12
Protokollierung	1	1	1	1	4
Infrastruktur	2	3	5	3	13
Kultur und Organisation	2	3	4	4	13
Dynamische Tiefe	1	2	2	4	9
Statische Tiefe	1	1	2	4	8
Test-Intensität	1	1	1	1	4
Konsolidierung	2	1	2	4	9
Anwendungstest	1	1	1	2	5
Infrastrukturtest	1	1	1	3	6
Anzahl pro Ebene	15	19	30	34	98

Tabelle 6.1: Anzahl der Implementierungspunkte pro Ebene und Dimension (D)

In Tabelle 6.2 ist eine Übersicht über das arithmetische Mittel für den „Nutzen“ und die Schwere der Implementierung („Schwere“) pro Ebene und Dimension gegeben. Zusätzlich ist die Standardabweichung σ aufgeführt. Die „Schwere der Implementierung“ aus transitiven IP geht nicht mit ein.

6.3.2 Priorisierung und Vermeidung von Redundanzen

Da externe Angriffe auf Organisationen zu ca. 85% auftreten, während interne Angriffe nach [236, S. 15] zu 15% beziehungsweise nach [126, S. 24] nur zu 7% auftreten, ist die Bedrohung aus externen Angriffen im Modell höher bewertet als diejenigen internen aus Angriffen. Der IP „Rollen-basierte Authentifizierung und Autorisierung“ ist deshalb erst auf Ebene 3 angeordnet.
Außerdem ist bei der Priorisierung der Maßnahmen der geschätzte „Nutzen“ sowie die geschätzte „Schwere der Implementierung“ berücksichtigt. Die nach der Implementierung anfallenden Wartungsarbeiten sind stark von den genutzten Technologien und Methoden abhängig und damit noch schwerer als die „Schwere der Implementierung“ generisch zu prognostizieren. Auf eine Berücksichtigung der anfallenden Wartungsarbeiten wird deshalb verzichtet.
Die erste Ebene soll möglichst einfach erreichbar sein. Deshalb wird in der Unter-Dimension „Dynamische Tiefe“ der IP „Einfacher Scan“ vom IP „Berücksichtigung von Rollen“ der zweiten Ebene getrennt.
Der IP „Interne Systeme sind einfach geschützt“ wird über die „Infrastruktur“ abgebildet, bei Webanwendungen häufig über eine *HTTP*-Authentifizierung. Der IP „Rollen-basierte Authentifizierung und Autorisierung“ ist ebenfalls nur der Dimension „Infrastruktur“ zugeordnet, um keine Redundanz zu erzeugen.

6.3.3 Vorstellung der Dimension „Test und Verifizierung“

Die Dimension „Test und Verifizierung“ besteht aus den Unter-Dimensionen „Dynamische Tiefe“, „Statische Tiefe“, „Konsolidierung“, „Anwendungstest“ und „Infrastrukturtest“.
Im Folgenden wird die Unter-Dimension „Statische Tiefe“ detailliert beschrieben, die IP sind in Abb. 6.1 inklusive ihrer Abhängigkeiten dargestellt. Ein Bildschirmfoto der Webanwendung mit den Abhängigkeiten der Dimension „Statische Tiefe“ ist im Anhang K.1 auf Seite 190 abgelegt.

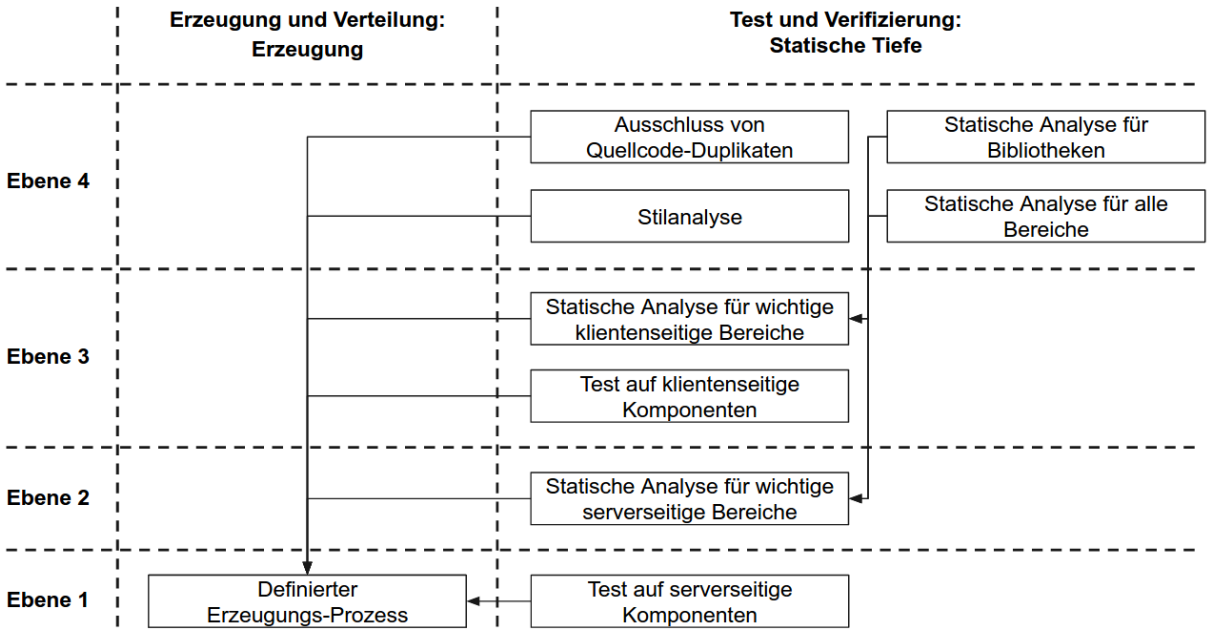


Abbildung 6.1: Abhängigkeiten von den Implementierungspunkten der Dimension „Statische Tiefe“

Der IP „Test auf serverseitige Komponenten“ der ersten Ebene enthält die Informationen:

- Risiken und Gegenmaßnahmen
  - Risiko: Eingesetzte serverseitige Komponten können Fehler enthalten, so dass die Informationssicherheit beeinträchtigt wird. Diese können u.a. erst nach Verteilung der Webanwendung bekannt werden.
  - Gegenmaßnahme: Tests auf serverseitige Komponenten mit bekannten Schwachstellen werden regelmäßig durchgeführt, beispielsweise jede Nacht.
- Nutzen und Schwere der Implementierung
  - Nutzen: Sehr hoch
  - Benötigtes Wissen: Sehr wenig
  - Benötigte Zeit: Wenig
  - Benötigte Ressourcen (Systeme): Sehr wenig
- Gewährleistete Sicherheitseigenschaften
  - Integrität: Durch Tests auf Komponenten mit bekannten Schwachstellen ist die Wahrscheinlichkeit geringer, dass durch Schwachstellen in Komponenten Daten von nicht autorisierten Personen oder Systemen verändert werden können.

  - Verfügbarkeit: Durch Tests auf Komponenten mit bekannten Schwachstellen ist die Wahrscheinlichkeit geringer, dass Schwachstellen in Komponenten ausgenutzt werden, um die Verfügbarkeit des Systems zu beeinträchtigen.
  - Vertraulichkeit: Durch Tests auf Komponenten mit bekannten Schwachstellen ist die Wahrscheinlichkeit geringer, dass durch Schwachstellen in Bibliotheken Daten von nicht autorisierten Personen oder Systemen eingesehen werden können.

- Sonstiges

  - Abhängigkeiten: Definierter Erzeugungs-Prozess
  - Implementierung: *OWASP Dependency Check*, *retirejs*

Namen der IPe sollten möglichst kurz sein, entsprechend wird auf den Postfix „mit bekannten Schwachstellen“ verzichtet. Der Begriff Bibliothek wird nicht verwendet, da auch eingesetzte *Frameworks* Schwachstellen enthalten können. Die Gegenmaßnahme beschreibt, wie das beschriebene Risiko entschärft wird. Weiterhin werden die durch Patch-Management gesicherten Sicherheitseigenschaften Integrität, Verfügbarkeit und Vertraulichkeit aufgeführt und beschrieben [222]. Danach wird die Abhängigkeit von dem IP „Definierter Erzeugungs-Prozess“ aufgeführt. Ohne einen definierten Erzeugungs-Prozess ist es schwieriger den Test zu definieren. Zudem wird durch einen definierten Erzeugungsprozess sichergestellt, dass Bibliotheken immer auf die gleiche Weise eingebunden werden und damit nicht aus dem Test ausgeschlossen werden. Abschließend sind zwei Werkzeuge zur Implementierung aufgeführt.
Im GDOSR wird in der Dimension „Statische Tiefe“ die gleiche Reihenfolge der IP wie in *SDOMM* genutzt, allerdings wird in GDOSR zwischen serverseitigen und klientenseitigen IP unterschieden. Dadurch können serverseitige IP höher als klientenseitige IP priorisiert werden. Zusätzlich werden in GDOSR auf Ebene 4 die IP „Ausschluss von Quellcode-Dupliakten“ und „Stilanalyse“ aufgeführt, welche beide einen „Nutzen“ von „Sehr wenig“ aufweisen.
Der IP „Standardeinstellungen für Test-Intensität“ der Unter-Dimension „Test-Intensität“ hindert Personen bei der Implementierung an der Änderung der „Test-Intensität“, welche insbesondere bei statischen Werkzeugen zu erhöhten falsch Positiven führen kann und bei dynamischen Werkzeugen zu einer erhöhten Testdauer führt. Der „Nutzen“ ist mit „Sehr wenig“ bewertet.
Die IPe „Angepasste Test-Intensität“ und „Test-intensität ist hoch eingestellt“ benötigen erweiterte Ressourcen, insbesondere für dynamische Tests.
Bei der Unter-Dimension „Anwendungstest“ wurden bei der Priorisierung die Ergebnisse der Umfrage (siehe Abschnitt 2.3) berücksichtigt. Da Modultests mit dem höchsten Stellenwert bewertet wurden, werden diese auf Ebene 1 mit Bezug auf Sicherheit unter dem IP „Modultests mit Sicherheitsbezug“ eingeordnet.

6.4 Identifizierung des Implementierungs-Grades

Um den aktuellen Grad der Implementierung in einer Organisation schrittweise festzulegen, kann in der Webanwendung unter dem Reiter „Identifizierung des Implementierung-Grades“ durch Klick auf einen IP dieser als umgesetzt markiert werden. In dem zugehörigen Netzdiagramm mit Hitzekarte (siehe Abb. 6.2)

wird so der Implementierungs-Grad erfasst. Die einzelnen Zellen enthalten dann einen Grün-Ton, welcher je nach Implementierungs-Grad der entsprechenden Dimension und Ebene dunkler wird.

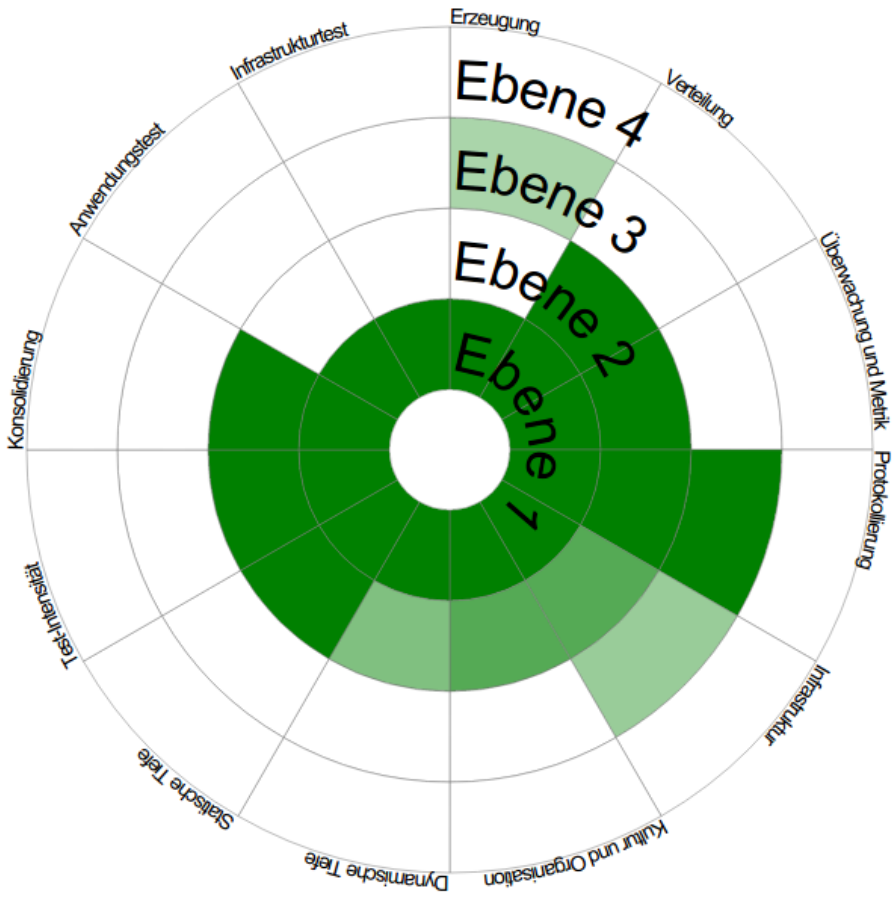


Abbildung 6.2: Netzdiagramm mit Hitzekarte zur Identifizierung des Implementierungs-Grades

Typisch für Reifegradmodelle ist die Erfassung des Implementierungs-Grades beziehungsweise Reifegrades mittels Radardiagramm. Hier wird der nächste „Grad“ einer Dimension erst erreicht, wenn eine Ebene komplettiert ist. Es kann also nicht erkannt werden, wenn eine Ebene bereits teilweise implementiert ist. Weil der Fortschritt i.d.R. vor dem Management verantwortet werden muss, ist dies nicht ausreichend. Entsprechend ist ein Netzdiagramm mit Hitzekarte erstellt worden, aus der auch ein niedriger Implementierungs-Grad abgelesen werden kann.

Da IPe innerhalb einer Dimension nicht aufeinander aufbauen müssen, kann der Implementierungs-Grad einer Dimension einer höheren Ebene größer sein als der Grad einer niedrigeren Ebene. Beispielsweise kann ein IP bereits vor der Einführung des Modells umgesetzt worden sein. Der IP „Gleiches Artefakt für Umgebungen“ der Ebene 3 kann z.B. durch Nutzung von *Docker* mit einem Repositorium erreicht werden. Ist dies im IP „Definierter Erzeugungs-Prozess“ so vorgesehen, kann „Gleiches Artefakt für Umgebungen“ als implementiert markiert werden. Der IP „Regelmäßiger Test“ der Ebene 2 ist dadurch jedoch noch nicht implementiert.

6.5 Evaluierung in der Entwicklungsphase „Test“

Die Evaluierung der Dimension „Erzeugung“ fand am 27.04.2016 beim „DevOps Meetup HH“ mit 30 *DevOps*-Experten statt. Ein Großteil der Teilnehmer hat eine Ausbildung zum Fachinformatiker absolviert. Die Einladung ist im Anhang D.2 auf Seite 125 einsehbar. Zunächst wurden die unterschiedlichen IPe für die Dimension im Rahmen eines Vortrags[2] vorgestellt. Anschließend erfolgte eine qualitative Befragung der Teilnehmer, während der diese gemeinsam den „Nutzen“ für Sicherheit und die „Schwere der Implementierung“ der IPe bewerten. Der „Nutzen“ ist von „Sehr klein“ (kurz SK) bis „Sehr hoch“ (kurz SH) zu bewerten und die „Schwere der Implementierung“ von „Sehr einfach“ bis „Sehr schwer“. Um den Bewertungsprozess möglichst einfach zu gestalten, wird auf die einzelne Angabe von Wissen, Zeit und Ressourcen verzichtet.

Im Rahmen der Bewertung wurden die IPe „definierter Erzeugungs-Prozess“, „Ein Artefakt“ und „Versionierte Artefakte“ bewertet, siehe Tabelle 6.3. Anschließend wurde die Befragung aufgrund eines sehr

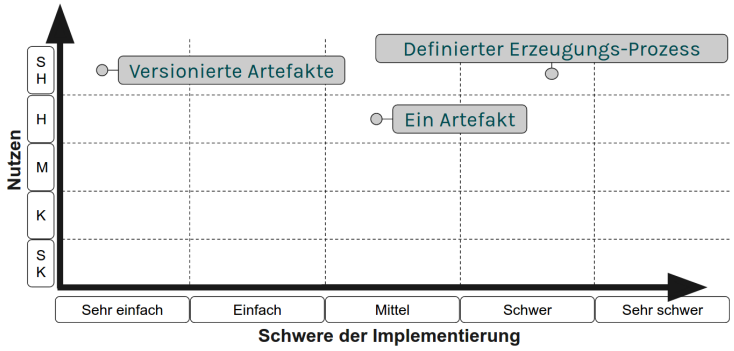


Abbildung 6.3: Qualitative Umfrage zur Dimension Erzeugung

hohen Zeitbedarfs für Diskussionen für jeden IP eingestellt. Dabei war der „Nutzen“ für die Erhöhung der Sicherheit schnell bewertet. Die „Schwere der Implementierung“ enthielt jedoch pro IP hohen Diskussionsbedarf. Je nach Projektumfeld der Teilnehmer wurde für den IP „Definierter Erzeugungs-Prozess“ „Mittel“, „Schwer“ und „Sehr schwer“ angegeben. Die Diskussion dauerte zehn Minuten und es ergab sich eine Einigung bei einer „Schwere der Implementierung“ von „Schwer“. Bei dem Punkt „Erstellung eines Artefakts“ für Test- und Produktionsumgebung wurde zunächst die Definition von Artefakt geklärt, da dies auch ein Umgebungsartefakt sein kann, also z.B. eine virtuelle Maschine. Virtuelle Maschinen sind der Dimension „Infrastruktur“ zugeordnet, daher ist eine Optimierung des Titels und der Risikobeschreibung erforderlich. Der Titel wurde von „Ein Artefakt“ in „Gleiches Artefakt“ und die Risikobeschreibung von „Es wird ein unterschiedliches Artefakt [...] verwendet. [...]“ in „es wird ein unterschiedliches Artefakt der Anwendung [...] verwendet. [...]“ geändert. Die Einordnung in eine „Mittel“ schwere Implementierung erfolgte nach acht Minuten Besprechung. „Versionierte Artefakte“ sind durch Techniken wie *Docker* einfach umzusetzen, entsprechend konnte hier nach fünf Minuten eine Einigung erzielt werden. Konsens war, dass der IP „Versionierte Artefakte“ auch durch einen Ordner mit einer Versionsnummer umgesetzt werden kann, aber besser ein Abbild-Repositorium mit Artefakten genutzt werden sollte. Da für die Einschätzung 15 Minuten geplant waren und bei dem Vortrag ein Zeitvorgabe vorhanden war, wurde die Befragung gestoppt. Statt dessen wurde die ausgearbeitete Einschätzung von Timo Pagel vorgestellt und

[2] Die Folien zum Vortrag sind unter `https://docs.google.com/presentation/d/12NU_QOgZuWPMbhOUzX44h__UOe9ONpjoZtj6L8ud3v8/edit?usp=sharing` abgelegt.

um Kommentare gebeten. Dabei wurde der ausgearbeiteten Einschätzung bis auf den IP „Austausch von Konfigurationsparametern“ zugestimmt. In großen Umgebungen mit mehreren tausend virtuellen Umgebungen kann die Konfiguration sehr komplex werden. Anschließend wurde die benötigte Zeit für den IP von „Sehr klein“ auf „Klein“ gesetzt. Dabei wurde nicht „Mittel“ gewählt, da die Relation zu anderen IP gewahrt werden sollte.
Die Evaluierung der Dimension „Statische Tiefe“ fand am 17.05.2016 beim „OWASP Stammtisch Hamburg“ mit 12 Web-Sicherheits-Experten statt. Ein Großteil der Teilnehmer hat ein Studium absolviert. Die Einladung ist im Anhang D.1 auf Seite 124 aufgeführt und eine Zusammenfassung der Experten-Befragung ist im Anhang B.11 auf Seite 117 zu finden. Zunächst wurden die möglichen Einsatzpunkte für sicherheitsbezogene Tests im Rahmen eines Vortrags[3] vorgestellt. Anschließend wurde auf Methoden für statische Tests eingegangen und danach IPe für die Dimension „Statische Tiefe“ vorgestellt. Da die Befragung beim „DevOps Meetup HH“ zu komplex war, wurde diese optimiert. Die Teilnehmer haben eine nicht sortierte Liste von IPen erhalten und sollten diese gemeinsam priorisieren. Hier wurde jedoch direkt via „Nutzen“ für Sicherheit und „Schwere der Implementierung“ die Priorisierung argumentiert.
Die Priorisierung wurde wie folgt festgelegt:

- Test auf Komponenten mit bekannten Schwachstellen
- Statische Analyse für wichtige Backend-Bereiche
- Statische Analyse für wichtige Frontend-Bereiche
- Statische Analyse für alle Bereiche
- Ausschluss von Quellcode-Duplikaten
- Stilanalyse
- Statische Analyse für Bibliotheken

Reibungspunkt war die Priorisierung von dem IP „Test auf Komponenten mit bekannten Schwachstellen“ (siehe 5.7 auf Seite 69), welcher anschließend in „Test auf serverseitige Komponenten“ und „Test auf klientenseitige Komponenten“ aufgeteilt wurde.
Die Titel der IPe „Statische Analyse für wichtige Frontend-Bereiche“ und „Statische Analyse für wichtige Backend-Bereiche“ waren für die Teilnehmer nicht eindeutig. Initial wurde von einigen Teilnehmern vermutet, dass das „Frontend“ eine *PHP*-Anwendung und das „Backend“ eine daran gekoppelte Anwendung, z.B. eine Warenwirtschafts-Anwendung, ist. Entsprechend erfolgte eine Umbenennung der Titel in „Statische Analyse für wichtige klientenseitige Bereiche“ und „Statische Analyse für wichtige serverseitige Bereiche“.
Die Experten haben den Nutzen der IPe „Stilanalyse“ und „Ausschluss von Quellcode-Duplikaten“ als sehr gering eingestuft. Entsprechend wurden diese von „Ebene 3“ auf „Ebene 4“ überführt.
Weiter wurde angemerkt, dass die Definition von „wichtigen Bereichen“ in einer Organisation wahrscheinlich nicht immer eindeutig ist. Abgesehen von der „Stilanalyse“ bestätigten die Experten die vorher im Modell definierte Reihenfolge in den Dimensionen.

[3] Die Folien zum Vortrag sind unter `https://docs.google.com/presentation/d/10cu8g1DKX_gLiwjEAfhzwmeJNNHO4Gef2ZKZzHwAwX8/edit?usp=sharing` abgelegt.

Da das Reifegradmodell erst mit der Veröffentlichung dieser Arbeit bekannt wird, ist es noch nicht durch ein drittes, unabhängiges Unternehmen implementiert worden. Diese Arbeit[4] ist (unter Maskierung vertraulicher Informationen) unter der „Creative Commons Namensnennung 4.0 International Lizenz“ und das Reifegradmodell ist unter der „GNU General Public License Version 3“ veröffentlicht, um den Prozess zu beschleunigen.

[4] Siehe http://files.timo-pagel.de/studium/DevOpsSecurity.pdf.

	Ebene 1		Ebene 2		Ebene 3		Ebene 4	
Dimension	**Nutzen**	**Schwere**	**Nutzen**	**Schwere**	**Nutzen**	**Schwere**	**Nutzen**	**Schwere**
Erzeugung	4.00 σ=0.00	2.50 σ=0.50	2.00 σ=0.00	1.00 σ=0.00	3.00 σ=0.82	1.92 σ=0.49	2.00 σ=0.00	2.00 σ=0.00
Verteilung	4.00 σ=0.00	1.75 σ=0.43	4.00 σ=0.00	1.63 σ=0.48	2.67 σ=0.94	1.67 σ=0.47	2.00 σ=0.00	1.25 σ=0.43
Überwachung und Metrik	5.00 σ=0.00	2.00 σ=0.00	4.00 σ=1.00	3.00 σ=1.58	4.00 σ=1.10	3.40 σ=1.20	4.50 σ=0.50	2.81 σ=1.42
Protokollierung	2.00 σ=0.00	1.00 σ=0.00	4.00 σ=0.00	2.50 σ=0.87	5.00 σ=0.00	1.00 σ=0.00	3.00 σ=0.00	4.00 σ=0.00
Infrastruktur	4.50 σ=0.50	3.25 σ=0.66	3.67 σ=0.94	3.00 σ=0.82	4.00 σ=0.89	3.00 σ=1.00	3.67 σ=0.94	4.08 σ=0.95
Kultur und Organisation	4.00 σ=0.00	1.00 σ=0.00	3.33 σ=0.47	1.92 σ=0.64	3.00 σ=0.00	2.13 σ=0.78	3.20 σ=1.17	3.45 σ=1.32
Dynamische Tiefe	2.00 σ=0.00	2.25 σ=0.83	3.00 σ=1.00	2.50 σ=0.87	5.00 σ=0.00	2.25 σ=0.83	3.00 σ=1.22	4.06 σ=1.25
Statische Tiefe	5.00 σ=0.00	1.50 σ=0.50	4.00 σ=0.00	1.75 σ=0.43	2.50 σ=0.50	1.63 σ=0.48	2.25 σ=1.30	1.69 σ=0.98
Test-Intensität	1.00 σ=0.00	1.00 σ=0.00	1.00 σ=0.00	2.25 σ=0.83	2.00 σ=0.00	3.00 σ=0.00	3.00 σ=0.00	3.50 σ=0.87
Konsolidierung	4.00 σ=0.00	1.00 σ=0.00	3.00 σ=0.00	1.75 σ=0.43	2.50 σ=0.50	1.75 σ=0.43	2.00 σ=0.00	2.44 σ=0.86
Anwendungstest	3.00 σ=0.00	3.25 σ=0.83	2.00 σ=0.00	3.25 σ=0.83	1.00 σ=0.00	3.25 σ=0.83	2.50 σ=0.50	3.25 σ=1.39
Infrastrukturtest	5.00 σ=0.00	1.00 σ=0.00	4.00 σ=0.00	1.75 σ=0.43	1.00 σ=0.00	1.25 σ=0.43	2.67 σ=0.47	2.00 σ=1.08

Legende

Nutzen	Schwere
Sehr hoch	Sehr einfach
Hoch	Einfach
Mittel	Mittel
Gering	Schwer
Sehr gering	Sehr Schwer

Tabelle 6.2: Bildschirmfoto der Webanwendung über das arithmetische Mittel für den Nutzen und die Schwere der Implementierung pro Dimension und Ebene

Kapitel 7

Implementierung des generischen Reifegradmodells

Die Implementierung, also die Entwicklungsphase „Anwendung“, des GDOSRs erfolgt in einem anonymen Startup (nachfolgend anoStartup). Die Implementierung erfolgt am Beispiel von *Open Source* Software, Organisationen können zur Optimierung auch kommerzielle Produkte nutzen.
Timo Pagel ist für die Implementierung des Modells verantwortlich und verfügt über ein Ausbildung zum Systemadministrator und über Erfahrung als Webentwickler. Erfahrung im Bereich Sicherheit hat Timo Pagel durchs Studium als auch als Dozent. anoStartup nutzt für die Bereitstellung einer Webanwendung das *playframework*, welche im Folgenden als Beispiel verwendet wird. Die Webanwendung verfügt über 50.000 Zeilen Quellcode, ohne Bibliotheken.
Der Grad der Implementierung kann in dem Netzdiagramm mit Hitzekarte in Abb. 7.1 abgelesen werden. Auf die Dokumentation der Tests nach der jeweiligen Implementierung wird aus Platzgründen verzichtet.

Modul und Integrationstests wurden im Januar 2016 implementiert, nach einer Aktualisierung des *playframeworks* im Juni 2016 von Version 2.2.0 auf Version 2.4.3 sind diese nicht mehr ausführbar. Bei Einführung weiterer Modul- und Integrationstest ist geplant, nichtfunktionale Modul- und Integrationstests mit Sicherheitsbezug wieder einzuführen. Bis anoStartup funktionale Modul- und Integrationstests einsetzt, sind nichtfunktionale Tests zu aufwendig im Verhältnis zum Nutzen. Aufgrunddessen ist die Unter-Dimension „Anwendungstest“ für anoStartup nicht relevant und im Netzdiagramm zur Identifizierung des Implementierungs-Grades kariert dargestellt.
Die IP der Dimension „Dynamische Tiefe“ wurden vom Autor bereits in einem anderem anonymen Unternehmen eingeführt und die Durchführbarkeit bestätigt, deshalb werden statische Tests priorisiert um ebenfalls die Durchführbarkeit zu bestätigen.
Eine Übersicht über die im Rahmen des Modells implementierten Systeme bietet Abb. 7.2 auf Seite 85. Dabei wird auf die Darstellung von Systemen ohne Bezug zu der Webanwendung auf dem physikalischem Server *luke.example.com* verzichtet.
Die Implementierung wird im Folgenden am Beispiel der Dimension „Test und Verifizierung“ vorgestellt, die komplette Implementierung ist im Anhang L auf Seite 191 aufgeführt.

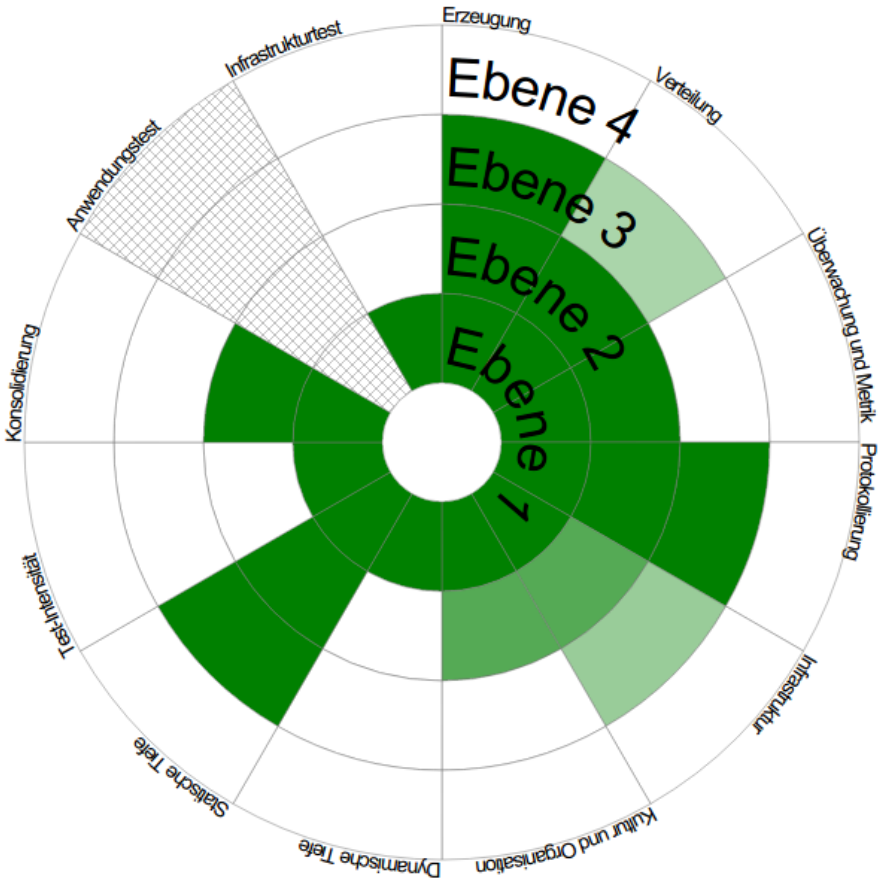


Abbildung 7.1: Netzdiagramm mit Hitzekarte zur Identifizierung des Implementierungs-Grades der ano-Startup

7.1 Implementierung von Ebene 1 der Dimension „Test und Verifizierung“

Dynamische Tiefe: Einfacher Scan

Implementierungs-Dauer: 1 Tag

Das *Zaproxy Plugin* [55] für *Jenkins* wird zum Starten des *Spiders* und des Scanners genutzt. Zunächst konnte auf die zu testende Anwendung nicht zugegriffen werden, da die *HTTP*-Authentifizierung vom System *stage.example.com* dies nicht zugelassen hat. Der Zugriff auf *stage.example.com* aus dem internen Netzwerk wurde anschließend erlaubt.

Statische Tiefe: Test auf serverseitige Komponenten

Implementierungs-Dauer: 1 Tag

Um die Webanwendung zu erstellen und alle Abhängigkeiten aus dem lokalem Verzeichnis *~/ivy2* in das aktuelle Arbeitsverzeichns zu überführen, wird `./activator docker:stage` ausgeführt und anschließend mittels Starten des *Docker*-Abbilds *anoStartup/docker-owasp-dependency-check* die Abhängigkeiten im Projekt getestet. Das *Docker*-Abbild *anoStartup/docker-owasp-dependency-check* wird automatisch via *hub.docker.com* neu erzeugt, wenn in dem zugehörigem *git*-Projekt[1] eine Änderung erfolgt. Ein Bild-

[1] Siehe https://github.com/wurstbrot/Docker-OWASP-Dependency-Check.

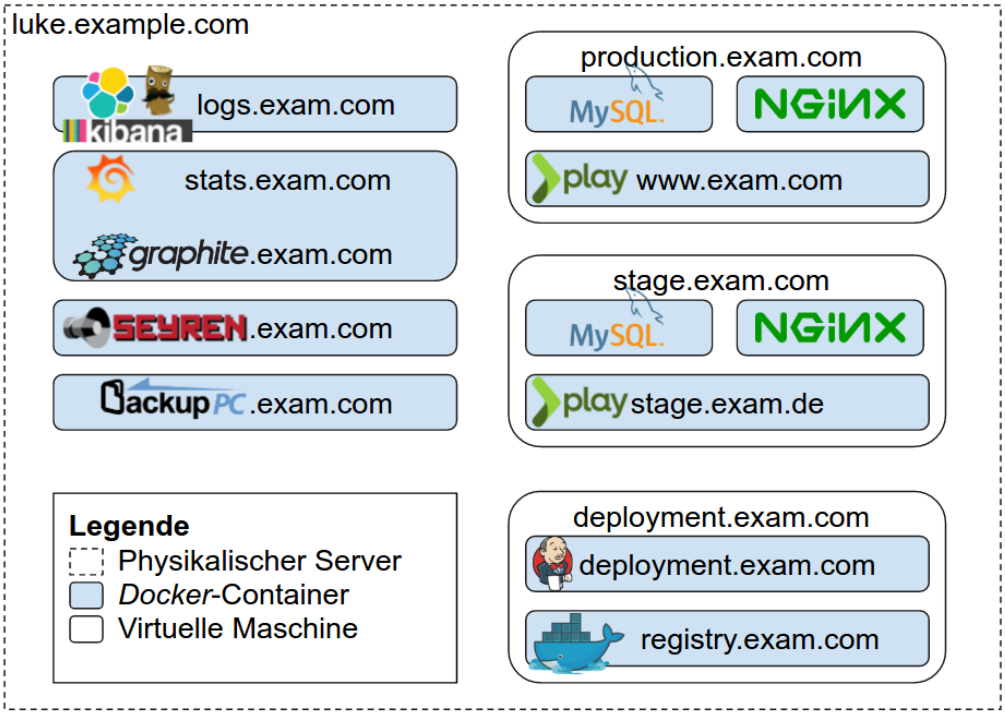


Abbildung 7.2: System-Übersicht der anoStartup
Bildquellen: [23, 27, 108, 257, 13, 53, 3, 25, 179, 239, 19, 104]

schirmfoto der Konfiguration in dem *Jenkins*-Auftrag ist im Anhang L.1 abgelegt.

Es wird ein textueller Abhängigkeitsbaum mittels *sbt-dependency-graph* durch den *Bash*-Befehl `sbt dependencyTree` erstellt um die ursprünglich eingebundenen Bibliotheken zu ermitteln. So ist u.a. die Bibliothek *deadbolt* von Version 2.4.3 auf 2.5.0 aktualisiert wurden.

Die Bibliothek *Apache Commons BeanUtils* wird durch die Bibliothek *OWASP esapi* in der Version 1.8.3 eingebunden und die Bibliothek *OWASP esapi* liegt bereits in der aktuellen Version vor. Deshalb wird auf eine Aktualisierung von *org.owasp.esapi* gewartet. Entsprechend wurde die Meldung in die falsch Positiv-Datei vom *OWASP Dependency-Check* eingebunden.

Weitere Informationen zur Implementierung, zu den Risiken in den Bibliotheken sowie zur Behandlung der Risiken sind im Anhang im Abschnitt L.1.5 auf Seite 194 abgelegt.

Die bekannten Risiken wurden so von insgesamt 16 kritischen Risiken und 87 mit Kritikalität „Mittel“ auf 11 kritische Risiken und 21 mit Kritikalität „Mittel“ reduziert. Es kann eine Bibliothek durch transitive Abhängigkeiten mehrere Risiken enthalten.

Es wurde festgelegt, dass die 11 kritischen Bibliotheken im Anschluss an diese Arbeit nach Möglichkeit zu aktualisieren.

Test-Intensität: Standardeinstellungen für Intensität

Implementierungs-Dauer: /

Hier müssen keine Anpassungen vorgenommen werden.

Konsolidierung: Behandlung kritischer Alarme

Implementierungs-Dauer: 1 Tag

Zap speichert gefundene Risiken im Format *Extensible Markup Language* (kurz *XML*) welche anschließend auf der Konsole in *Jenkins* ausgegeben werden. Die Ausgabe enthält für jede Schwachstelle u.a. das Attribut *riskcode* mit einem numerischem Wert. Dabei entspricht 0 einer Information, 1 der Kritikalität Niedrig, 2 Mittel und 3 Hoch. Die Ausgabe wird nach „<riskcode>“ gefiltert und auf Kritikalität größer 2 getestet. Ist dies der Fall, wird *Found Defect: <Kritikalität>* ausgegeben. Das *Log Parser Plugin* [22] testet anschließend, ob *Found Defect* ausgegeben wurde. Wurde es mindestens einmal ausgegeben, wird der Auftrag mit einem *Error* markiert. Das *Bash*-Skript, welches beim Durchlauf von dem *OWASP-Zap*-Auftrag jeden Tag um 07:15 Uhr ausgeführt wird, ist im Anhang L.1 abgelegt.

Konsolidierung: Einfache Falsch-Positiv-Behandlung

Implementierungs-Dauer: 1 Tag
In der Oberfläche von *Zap* kann durch Doppelklick auf ein gefundenes Risiko das „Vertrauen“ als falsch Positiver eingestuft werden. Damit *Jenkins* auf die falsch Positiven von *Zap* zugreifen kann, wird die Sitzung von *Zap* in dem Ordner *security* in der Versionskontrolle des Projekts gespeichert und beim Testen von *zaproxy* geladen.
Im *OWASP Dependency-Check* kann eine „Susspression“-Datei zur Markierung falsch Positiver im Format *XML* angegeben werden. Die Datei *dependency-check-suppression.xml* ist ebenfalls im Ordner *security* abgelegt.

Anwendungstest: Modultests mit Sicherheitsbezug

Implementierungs-Dauer: 3 Tag
Die Anwendung bietet eine Schnittstelle, von welcher ein Benutzername und ein Passwort gefordert wird. Beispielhaft wurde die Klasse `ApiLoginUnitTest` erstellt, welche mittels der Methoden `loginWithExistingUser()` und `loginWithNonExistingUser()` testet, dass sich bestehende Benutzer erfolgreich anmelden können und Zugriff nicht ohne hinterlegten Benutzer erfolgen kann. Der Quellcode ist im Anhang **??** auf Seite ?? abgelegt.
Alle Tests werden auf *Jenkins* mittels `./activator test` ausgeführt.

Infrastrukturtest: Test auf System-Aktualisierungen

Implementierungs-Dauer: 1 Tag
Über Aktualisierungen für Betriebssysteme wird über *apticron* [29] informiert. Werkzeuge wie *watchtower* ermöglichen, auf Aktualisierungen von *Docker*-Abbildern zu testen, das Abbild herunterzuladen und automatisch den Container neu zu starten. Dies birgt jedoch die Gefahr, dass eine Aktualisierung, insbesondere wenn persistent gespeicherte Daten aktualisiert werden, nicht klappt und der Dienst nicht mehr verfügbar ist. Entsprechend wurde ein Skript (siehe Listing 7.1) erstellt, um nächtlich mittels *cronjob* eine Information via E-Mail zu versenden. Das Skript ist auf allen Umgebungen mit *Docker* installiert und mittels *cronjob* aufgerufen.
In dem Skript wird zunächst für jedes Abbild ein `pull` ausgeführt und die Ausgabe, gefiltert auf „Downloaded newer image“, in */tmp/dockerupdate* umgeleitet. Anschließend wird getestet, ob mindestens eine Zeile in */tmp/dockerupdate* vorhanden ist. Ist das der Fall, wird eine E-Mail mit der Information versandt.

Anschließend kann manuell eine Aktualisierung erfolgen. Während *apticron* jeden Tag eine E-Mail mit allen zur Verfügung stehenden Aktualisierungen sendet, sendet das Skript die E-Mail nur einmal, da nach dem `pull` ein aktuelles Abbild vorliegt.

Listing 7.1 Skript zur Erzeugung von E-Mails bei aktualisierten Abbildern für *Docker*-Container

```
1 #!/bin/bash
2 MAILTO="systems@example.com"
3 for i in $(docker images | awk '(NR>1) && ($2!~/none/) {print $1":"$2}')
    ; do
4         docker pull $i | grep "Downloaded newer image" >> /tmp/
              dockerupdate;
5 done
6
7 if [ $(cat /tmp/dockerupdate | wc -l ) -gt 0 ]; then
8         mail -s "Es ist mindestens ein neues Docker-Abbild auf $HOSTNAME
              vorhanden" $MAILTO < /tmp/dockerupdate fi
9 rm /tmp/dockerupdate
```

7.2 Implementierung von Ebene 2 der Dimension „Test und Verifizierung“

Statische Tiefe: Statische Analyse für wichtige serverseitige Bereiche

Implementierungs-Dauer: 4 Tage

Für die serverseitgen Komponenten wird *FindBugs* mit dem *FindSecurityBugs Plugin* genutzt. Dies ist einerseits in die Entwicklungsumgebung *JetBrains IntelliJ* integriert (siehe Anhang L.4 auf Seite 205 für ein Bildschirmfoto) und andererseits erfolgt auf *Jenkins* periodisch ein Test.

Zunächst wurde *FindBugs* mittels *findbugs4sbt* in den Erzeugungsprozess integriert. Allerdings unterstützt *findbugs4sbt* noch keine Plugins. Deshalb wurde *findbugs* heruntergeladen, in dem Ordner *security* des Quellcodes der Webanwendung abgelegt und wird mittels Skript *security/findbugs.bash* (siehe Anhang L.7 auf Seite 204) durch *Jenkins* gestartet. Durch die gescheiterte Integration mittels *findbugs4sbt* kommt die erhöhte Implementierungs-Dauer zustande.

Für den Aufruf mittels *Jenkins* als auch für die Integration in die Entwicklungsumgebung wird die Kategorie *Security* getestet. Weiterhin wird der Test auf *ES_COMPARING_STRINGS_WITH_EQ* (nachfolgend ES_C) der Kategorie *Bad Practice* in der Datei *findbugs-include-filter.xml* aktiviert (siehe Anhang L.6 auf Seite 203). Mittels *ES_C* wird getestet, ob Objekte mittels == Operator vergleichen werden. Werden Objekte mittels == Operator vergleichen, so werden die Referenzen beider Objekte vergleichen und es kann vorkommen, dass zwei Zeichenketten nicht identisch sind [114].

Beim ersten Durchlaufen wurden in der Kategorie *Security* 36 Warnungen und keine kritischen Fehler gemeldet. Es wurden zwei Warnungen für ES_C gemeldet.

Ein richtig Positiver mit Sicherheitsbezug ist ein Zeichenkenvergleich mittels == Operator in Java. Ein erstelltes *Event* kann mittels Passwort vor nicht berechtigten Zugriffen geschützt werden. In der Webanwendung wird mittels Vorbedingung in einer Methode ermittelt (siehe Listing 7.2 auf der nächsten Seite für ein vereinfachtes Beispiel), ob das korrekte Passwort angegeben wurde. Das würde dazu führen,

dass die Seite des *Events* ausgeliefert wird, obwohl ein falsches Passwort angegeben wurde. Der Vergleich wurde durch Aufruf der Methode `equals()` korrigiert.

Listing 7.2 Richtig Positiver eines Zeichenkettenvergleich ermittelt durch *FindBugs*

```
1 if (input.get("password") != event.getAccessPassword()) {
2    return redirect("/"); // Access not allowed
3 }
4 // Display site
```

Ein falsch Positiver ist beispielsweise die in dem Bildschirmfoto in Abb. L.4 auf Seite 205 gezeigte, als Mittel eingestufte, Warnung für die Nutzung eines vorhersagbaren Zufallsgenerator. In diesem Fall werden Bilder zufällig für die Ausgabe sortiert. Entsprechend ist eine Vorhersage der Reihenfolge der Bilder nicht kritisch und die Meldung wird mittels der Annotation `@SuppressFBWarnings("PREDICTABLE_RANDOM")` als falsch Positiver markiert.
Nach Behebung der richtig Positiven und Markierung von falsch Positiven sind 26 Warnungen nicht behandelt.

Konsolidierung: Alarme sind einfach visualisiert

Implementierungs-Dauer: 1 Tag
Die Visualisierung vom *OWASP Dependency-Check Plugin* erfolgt über das *Plugin* direkt als Trend.
Die Visualisierung der Schwachstellen von *Zap* und *FindBugs* erfolgt über das *Log Parser Plugin*. In Abb. 7.3 zeigt die Statistik von *FindSecurityBugs*. Zusätzlich bieten beide Werkzeuge einen *HTML*-Bericht, welcher über das *HTML Plugin* [15] in *Jenkins* angezeigt werden kann.

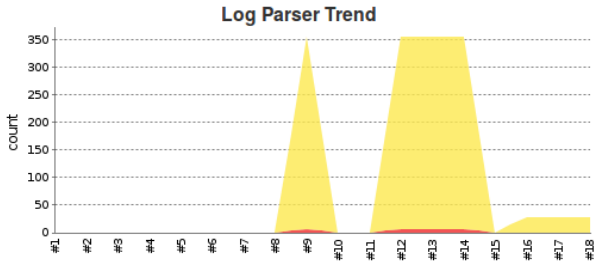


Abbildung 7.3: Visualisierung für den Trend des Tests von *FindSecurityBugs* mittels *Log Parser Plugin*

Anwendungstest: Integrationstests mit Sicherheitsbezug

Implementierungs-Dauer: 3 Tage
Die Anwendung bietet eine Schnittstelle, von welcher ein Benutzername und ein Passwort gefordert wird. Beispielhaft wurde die Klasse `ApiLoginTest` erstellt, welche mittels der Methoden `loginWithExistingUser()` und `loginWithNonExistingUser()` testet, dass sich Benutzer erfolgreich via *HTTP* anmelden können und Zugriff nicht ohne hinterlegten Benutzer erfolgen kann. Der Quellcode ist im Anhang L.9 abgelegt. Wie auch bei den „Modultests mit Sicherheitsbezug“ gilt, dass der Integrationstest nicht mehr ausführbar ist.

Infrastrukturtest: Prüfung der Konfiguration von virtuellen Umgebungen

Implementierungs-Dauer: 1 Tage

Mittels *Docker Bench for Security* wird die D*ocker*-Konfiguration virtueller Umgebungen manuell getestet. Der Aufruf, bei welchem alle Bindungen des Dateisystems nur lesenden Zugriff erhalten, ist wie folgt:

Listing 7.3 Aufruf von *Docker Bench for Security*

```
1 docker run -it --net host --pid host --cap-add audit_control \
2     -v /var/lib:/var/lib:ro \
3     -v /var/run/docker.sock:/var/run/docker.sock:ro \
4     -v /usr/lib/systemd:/usr/lib/systemd:ro \
5     -v /etc:/etc --label docker_bench_security:ro \
6     docker/docker-bench-security
```

Hier werden Meldungen wie *Container Restrict network traffic between containers* mit Kritikalität *WARN* erzeugt. Die Restriktion von Netzwerken ist erfasst unter dem IP „Kontrollierte Netzwerke für virtuelle Umgebungen“.

Da der Test nur manuell erfolgte, wird dieser IP nicht als ab geschlossen markiert.

7.3 Implementierung von Ebene 3 der Dimension „Test und Verifizierung“

Statische Tiefe: Statische Analyse für wichtige klientenseitige Bereiche

Implementierungs-Dauer: 3 Tage

Der Test erfolgt auf *Jenkins* mittels *ESlint* in einem *Docker*-Container. Der Aufruf in *Jenkins* ist im Anhang in Listing L.14 auf Seite 208 aufgeführt.

Das *Dockerfile* zum Erzeugen des *Docker*-Containers ist unter https://github.com/wurstbrot/retire.js aufgeführt. Der automatische Erzeugungsdienst von *hub.docker.com* erzeugt nächtlich aus dem *Dockerfile* das Abbild *anoStartup/eslint-docker*.

ESlint nutzt eine Regeldatei[2], welche *scanjs*-Regeln beinhaltet. Es wird nur der *JavaScript*-Quellcode von dem Modul *editor* getestet, dabei sind keine kritischen Fehler und 11 Warnungen gemeldet worden.

Statische Tiefe: Test auf klientenseitige Komponenten

Implementierungs-Dauer: 3 Tage

Der Test auf klientenseitige Komponenten mit bekannten Schwachstellen erfolgt mittels *retire.js* als *Docker*-Container. Dafür ist ein *git*-Projekt auf github.com (siehe https://github.com/wurstbrot/retire.js) angelegt, welches über *hub.docker.com* regelmäßig erzeugt wird und unter der Bezeichnung *anoStartup/retire.js* zur Verfügung steht. Entsprechend kann über *Jenkins* die Webanwendung vom Versionskontrollsystem abgeholt, die Bibliotheken mittels `./activator docker:stage` abgeholt und der *Docker*-Container gestartet werden. Mittels *Log Parser Plugin* für *Jenkins* wird auf Vorkommnis von „severity: high“ für

[2]Kopiert von https://github.com/18F/compliance-toolkit/blob/master/configs/static/.eslintrc, Abruf am 07.07.2016.

Fehler, „severity: middle“ für Warnungen und „severity: low“ für Informationen getestet. Wird ein Fehler gefunden, so wird der Auftrag als fehlerhaft markiert. Das von *Jenkins* ausgeführte Skript wird ist im Listing L.15 auf Seite 209 abgelegt.

7.4 Evaluierung der Implementierung

Die Implementierung der Ebene 1 erfolgte innerhalb von 24 Tagen. Teile der Implementierung der Ebene 2 erfolgte innerhalb 22 Tagen und Teile der Implementierung der Ebene 3 erfolgte innerhalb 11 Tagen.
Durch das Auffinden von Risiken mittels automatisierter Tests ist das Sicherheitsniveau der Webanwendung transparenter geworden. Auch hat sich gezeigt, dass nicht alle Risiken behoben werden können, beispielsweise die transitiven Abhängigkeiten gefunden durch den *OWASP Dependency Check*. Hier muss entschieden werden, nicht aktualisierbare Bibliotheken zu entfernen, als falsch Positiv zu markieren oder den Test fehlschlagen zu lassen. anoStartup hat sich dafür entschieden, jedem Test als fehlgeschlagen zu markieren, bis die Risiken behoben sind.
Es wurde versucht, *Docker*-Container mit hohem Festplattendurchsatz, wie den Backupdienst, in einer virtuellen Maschine auszuführen. Hier wurden starke Performance-Engpässe festgestellt, weshalb *Docker*-Container mit hohem Festplattendurchsatz auf dem physikalischem Server ausgeführt wird.
Die Implementierung der Echtzeit-Überwachung mit modernen *DevOps*-Technologien hat im Vergleich zu traditioneller Überwachung noch schwächen. Bei *Icinga* kann angeben werden, wie häufig ein Test fehlgeschlagenen sein muss, bis dieser kritisch ist. Bei *seyren* konnte eine entsprechende Funktion nicht gefunden werden. Auch ist die Konfiguration komplexer. Durch Plugins wie *check-graphite* kann *Icinga* Metriken von Graphite abfragen. Dadurch wird die Komplexität der Konfiguration jedoch nicht genommen. Beim aktuellem Technologiestand wird daher empfohlen, klassische Überwachungslösungen mit Schnittstellen zu neuen Technologien wie Graphite zu nutzen.
Weiterhin zeigt das ausblenden der Unter-Dimension „Anwendungstest“ die Adaptierung des Modells auf Bedürfnisse von Organisationen.

Kapitel 8

Fazit und Ausblick

8.1 Fazit

In der vorliegenden Arbeit wird gezeigt, dass *DevOps*-Strategien zur Erhöhung der Sicherheit von Startups genutzt werden können und Sicherheit in *DevOps*-Strategien zu integrieren ist.
Dafür fand zunächst eine Umfrage zur Identifizierung von gemeinsamen Zielen statt. Am Wichtigsten ist der Gesamtheit aller Befragten die Verfügbarkeit. Im Bereich IT-Sicherheit wird aber von Experten Integrität als am Wichtigsten bewertet, entsprechend können Konflikte auftreten.
Kulturell wird aufgezeigt, welche Möglichkeiten es zur Integration von Sicherheit in agile Methoden gibt. Als bedeutendes kulturelles Element dieser Arbeit fanden Team-Sicherheitsprüfungen statt, bei welchen ein Team die Sicherheit der Webanwendung eines anderen Teams prüfte. Sechs Vorteile und vier Nachteile sind bestätigt worden, fünf Hypothesen wurden nicht bestätigt. Alle Probanden sind durch die kulturelle Maßnahme aber verstärkt für das Thema Sicherheit sensibilisiert und haben ihr Wissen geteilt.
Die wichtigste Erkenntnis aus Team-Sicherheitsprüfungen ist, dass das Hauptziel nicht die Erhöhung der Sicherheit der Webanwendung sein kann, sondern das Teilen von Wissen der Teilnehmer. Ein Sicherheits-Experte kann aufgrund von Experten-Wissen eine Prüfung effizienter durchführen.
Die Team-Sicherheitsprüfung sollte allerdings nur mit Vorwissen im Bereich Informationssicherheit durchgeführt werden, welches also durch eine Schulung vermittelt wurde.
In dieser Arbeit ist weiterhin aufgezeigt, wie durch *DevOps*-Stragien eingesetzte Systeme und Anwendungen gehärtet und auf Sicherheit getestet werden können. Dabei liegt der Fokus der Härtung des Erzeugungs- und Verteilungsprozesses auf Sicherung der Integrität. Herausforderung ist das Testen der Integrität von Artefakten aus öffentlichen Repositorien.
Herausgestellt hat sich, dass es beim statischen Testen der Sicherheit am Wichtigsten ist, auf Komponenten mit bekannten Schwachstellen zu testen. Dynamisches Testen bietet gegenüber statischem Testen den Vorteil von reduzierter Anzahl an falsch Positiven. Daher sollte eine Kombination aus dynamischen und statischen Tests genutzt werden.
Um die Ergebnisse dieser Arbeit zu priorisieren und zu veranschaulichen, ist ein generisches *DevOps* Sicherheits-Reifegradmodell entwickelt worden. Dieses stellt die Erkenntnisse in fünf Dimensionen vor und ist in vier Ebenen zur Priorisierung aufteilt. Die Dimensionen sind „Erzeugung und Verteilung“, „Informationsgewinnung“, „Infrastruktur“, „Kultur und Organisation“ und „Test und Verifizierung“. Die

ersten beiden Ebenen sind leicht zu erreichen, während die nächsten beiden Ebenen eine erhöhte „Schwere der Implementierung“ aufweisen.
Insgesamt sind 98 Implementierungspunkte in das Modell aufgenommen, welche u.a. ein Risiko beschreiben und eine Gegenmaßnahme vorstellen.
Die Dimension „Erzeugung und Verteilung“ wurde teilweise beim „DevOps HH Meetup“ in Hamburg evaluiert, während die Unter-Dimension „Statische Tiefe“ der Dimension „Test und Verifizierung“ beim „OWASP Stammtisch Hamburg“ in Hamburg evaluiert wurde. Experten setzen unterschiedliche Schwerpunkte bei der Priorisierung der Implementierungspunkte, wobei immer ein Kompromiss erzielt wurde.
Die Implementierung von Ebene 1 des Modells innerhalb von 24 Tagen und die Implementierung von Teilen von Ebene 2 innerhalb von 22 Tagen beim Kieler Startup anoStartup zeigen, dass die höheren Ebenen des Modells für Startups mit wenig IT-Mitarbeitern schwer zu erreichen sind, daher bleiben sie Startups und Organisationen mit erhöhtem Kapital und Mitarbeitern vorbehalten. Das Modell kann von Startups und Organisationen adaptiert werden.
Die gesteckten Ziele dieser Arbeit wurden also erreicht.

8.2 Ausblick

Das Sicherheitsniveau bei der anoStartup als bisher kleines Startup sollte durch weitere Implementierung des Modells erhöht werden. Das Modell sollte außerdem bei weiteren unabhängigen Startups implementiert werden, um eine unvoreingenommene Bewertung zu erhalten.
Die Webanwendung zur Visualisierung des Modells kann durch eine visuelle Unterteilung der Dimensionen in dem Netzdiagramm mit Hitzekarte verbessert werden. Dadurch könnten zusammenhängende Dimensionen besser wahrgenommen werden.
Weiterhin sollte die inhomogene Team-Sicherheitsprüfung von weiteren Organisationen erprobt werden, um den Vorteil gegenüber einer inhomogenen Team-Sicherheitsprüfung zu bestätigen.
Es konnten keine freien Werkzeuge zur Evaluierung von *Interactive Application Security Testing* gefunden werden. Hier besteht entsprechend noch Bedarf der Entwicklung von freien Werkzeugen für Webanwendungen.

Literaturverzeichnis

[1] *Analysis Collector Plugin.* https://wiki.jenkins-ci.org/display/JENKINS/Analysis+Collector+Plugin, Abruf: 27.05.2016

[2] *Apache JMeter™.* http://jmeter.apache.org/, Abruf: 07.04.2016

[3] *Backuppc.* http://backuppc.sourceforge.net/, Abruf: 12.07.2016

[4] *Chef - Code Can | Chef.* https://www.chef.io/, Abruf: 28.06.2016

[5] *Commons Collections.* https://commons.apache.org/proper/commons-collections/index.html, Abruf: 30.05.2016

[6] *curl.* https://curl.haxx.se/, Abruf: 22.06.2016

[7] *CVE-2015-4852.* http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4852, Abruf: 30.05.2016

[8] *CVE-2015-7501.* http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7501, Abruf: 30.05.2016

[9] *CWE-502: Deserialization of Untrusted Data.* http://cwe.mitre.org/data/definitions/502.html, Abruf: 30.05.2016

[10] Bundesamt fur Sicherheit in der Informationstechnik: Durchführungskonzept für Penetrationstests. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/Penetrationstest/penetrationstest_pdf.pdf, Abruf: 09.06.2016

[11] *Find Security Bugs.* http://find-sec-bugs.github.io/, Abruf: 23.06.2016

[12] *GABLER WIRTSCHAFTSLEXIKON - Start-up-Unternehmen.* http://wirtschaftslexikon.gabler.de/Definition/start-up-unternehmen.html, Abruf: 19.06.2016

[13] *Graphite.* https://graphiteapp.org/, Abruf: 07.06.2016

[14] *How to encrypt a properties file.* https://www.owasp.org/index.php/How_to_encrypt_a_properties_file, Abruf: 07.05.2016

[15] *HTML Publisher Plugin.* https://wiki.jenkins-ci.org/display/JENKINS/HTML+Publisher+Plugin, Abruf: 27.06.2016

[16] *HttpUnit.* http://httpunit.sourceforge.net/, Abruf: 26.06.2016

[17] Icinga | Open Source Monitoring. https://www.icinga.org/, Abruf: 05.07.2016

[18] *InSpec: Inspect Your Infrastructure.* https://github.com/chef/inspec, Abruf: 27.05.2016

[19] *Jenkins.* https://www.jenkins.io, Abruf: 21.05.2016

[20] *JUnit.* http://junit.org/junit4/, Abruf: 25.06.2016

[21] *LimeSurvey - the most popular FOSS survey tool on the web.* https://www.limesurvey.org/de/, Abruf: 14.06.2016

[22] *Log Parser Plugin.* https://wiki.jenkins-ci.org/display/JENKINS/Log+Parser+Plugin, Abruf: 28.06.2016

[23] *logstash/docs/static/images/.* https://github.com/elastic/logstash/tree/master/docs/static/images, Abruf: 12.07.2016

[24] *Mask Passwords Plugin.* https://wiki.jenkins-ci.org/display/JENKINS/Mask+Passwords+Plugin, Abruf: 15.05.2016

[25] *MySQL Logo Downloads.* https://www.mysql.com/about/legal/logos.html, Abruf: 12.07.2016

[26] *Nagios - The Industry Standard In IT Infrastructure Monitoring.* https://www.nagios.org/, Abruf: 05.07.2016

[27] *Open Source, Distributed, RESTful Search Engine https://www.elastic.co/products/elasticsearch.* https://github.com/elastic/elasticsearch, Abruf: 12.07.2016

[28] *OpenVAS - OpenVAS - Open Vulnerability Assessment System Community Site.* http://www.openvas.org/index.de.html, Abruf: 02.07.2016

[29] *Package: apticron (1.1.57).* https://packages.debian.org/jessie/apticron, Abruf: 20.06.2016

[30] *Puppet - The shortest path to better software.* https://puppet.com/, Abruf: 28.06.2016

[31] *ReproducibleBuilds FileOrderInTarballs.* https://wiki.debian.org/ReproducibleBuilds/FileOrderInTarballs, Abruf: 10.06.2016

[32] *ReproducibleBuilds Howto.* https://wiki.debian.org/ReproducibleBuilds/Howto, Abruf: 05.06.2016

[33] Microsoft: SDL for Agile. https://www.microsoft.com/en-us/SDL/discover/sdlagile-bucket.aspx, Abruf: 25.06.2016

[34] *Selenium.* http://www.seleniumhq.org/, Abruf: 12.07.2016

[35] *StatsD + Graphite + Grafana 2 + Kamon Dashboards.* https://github.com/kamon-io/docker-grafana-graphite, Abruf: 26.04.2016

[36] *THC-Hydra.* https://www.thc.org/thc-hydra/, Abruf: 04.07.2016

[37] *Vagrant by HashiCorp.* https://www.vagrantup.com/, Abruf: 09.06.2016

[38] *VMware Virtualization for Desktop & Server, Application, Public & Hybrid Clouds.* http://www.vmware.com/, Abruf: 21.06.2016

[39] *Vulnerability scanner for Linux, agentless, written in golang.* https://github.com/future-architect/vuls, Abruf: 20.06.2016

[40] *Working with Rules - ESLint.* http://eslint.org/docs/developer-guide/working-with-rules, Abruf: 29.05.2016

[41] ISO27001: Information Security Management System (ISMS) standard. (2005)

[42] *SecurityFocus, Jenkins Multiple HTML Injection Vulnerabilities.* http://www.securityfocus.com/bid/52055/. Version: 2012, Abruf: 07.05.2016

[43] Oracle: Hard Partitioning With Oracle VM Server for x8. Version: 2013. http://www.oracle.com/technetwork/server-storage/vm/ovm-hardpart-168217.pdf, Abruf: 04.05.2016. 2013

[44] *Apache™ Maven Dependency Plugin.* http://maven.apache.org/plugins/maven-dependency-plugin/. Version: 2015, Abruf: 18.05.2016

[45] *building alerting system for grafana #2209.* https://github.com/grafana/grafana/issues/2209. Version: 2015, Abruf: 06.07.2016

[46] *Docker To Defang Root Privilege Access.* http://www.informationweek.com/software/enterprise-applications/docker-to-defang-root-privilege-access/d/d-id/1321023. Version: 2015, Abruf: 07.05.2016

[47] *LXC.* https://linuxcontainers.org/. Version: 2015, Abruf: 07.05.2016

[48] *OpenVZ Virtuozzo Containers Wiki.* https://openvz.org/Main_Page. Version: 2015, Abruf: 07.05.2016

[49] *Security DevOps Maturity Model (SDOMM).* https://www.christian-schneider.net/SecurityDevOpsMaturityModel.html. Version: Mai 2015, Abruf: 07.05.2016

[50] *SecurityFocus, Jenkins Multiple Cross Site Scripting and Directory Traversal Vulnerabilities.* http://www.securityfocus.com/bid/52384. Version: 2015, Abruf: 07.05.2016

[51] *Apache™ Subversion®.* https://subversion.apache.org/. Version: 2016, Abruf: 07.05.2016

[52] *Scale out with Ubuntu Server.* http://www.ubuntu.com/server. Version: 2016, Abruf: 27.06.2016

[53] *scobal/seyren: An alerting dashboard for Graphite.* https://github.com/scobal/seyren. Version: 2016, Abruf: 06.07.2016

[54] *Usage of operating systems for websites.* https://w3techs.com/technologies/overview/operating_system/all. Version: 2016, Abruf: 27.06.2016

[55] *zaproxy/zaproxy: The OWASP ZAP core project.* https://github.com/zaproxy/zaproxy. Version: 2016, Abruf: 20.06.2016

[56] Amazon Web Services, Inc.: *AWS | Amazon Elastic Compute Cloud (EC2) – Cloud Server*. https://aws.amazon.com/de/ec2/. Version: 2015, Abruf: 01.02.2016

[57] Arcangeli, Andrea: *seccomp: secure computing support*. http://git.kernel.org/cgit/linux/kernel/git/tglx/history.git/commit/?id=d949d0ec9c601f2b148bed3cdb5f87c052968554. Version: March 2005, Abruf: 23.01.2016

[58] Arkin, B. ; Stender, S. ; McGraw, G.: Software penetration testing. In: *IEEE Security Privacy* 3 (2005), Jan, Nr. 1, S. 84–87. http://dx.doi.org/10.1109/MSP.2005.23. – DOI 10.1109/MSP.2005.23. – ISSN 1540–7993

[59] Asthana, Vishal u. a.: *Practical Security Stories and Security Tasks for Agile Development Environments*. http://safecode.org/publication/SAFECode_Agile_Dev_Security0712.pdf. Version: 2012, Abruf: 12.06.2016

[60] at&t: Integrating Security Testing into Quality Control. (2011). http://www.business.att.com/content/whitepaper/Integrated_Security_QC_wp.pdf, Abruf: 07.07.2016

[61] Ayewah, Nathaniel ; Hovemeyer, David ; Morgenthaler, J ; Penix, John ; Pugh, William: Using static analysis to find bugs. In: *Software, IEEE* 25 (2008), Nr. 5, S. 22–29

[62] Ayewah, Nathaniel ; Pugh, William ; Morgenthaler, J ; Penix, John ; Zhou, YuQian: Evaluating static analysis defect warnings on production software. In: *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* ACM, 2007, S. 1–8

[63] Bachmann, Ruediger ; Brucker, Achim: Developing secure software. In: *Datenschutz und Datensicherheit* 38 (2014), Nr. 4, S. 257–261

[64] Barham, Paul ; Dragovic, Boris ; Fraser, Keir ; Hand, Steven ; Harris, Tim ; Ho, Alex ; Neugebauer, Rolf ; Pratt, Ian ; Warfield, Andrew: Xen and the art of virtualization. In: *ACM SIGOPS Operating Systems Review* Bd. 37 ACM, 2003, S. 164–177

[65] Bass, Len ; Holz, Ralph ; Rimba, Paul ; Tran, An ; Zhu, Liming: Securing a deployment pipeline. In: *Release Engineering (RELENG), 2015 IEEE/ACM 3rd International Workshop On* IEEE, 2015, S. 4–7

[66] Bass, Len ; Weber, Ingo ; Zhu, Liming: *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015

[67] Bath, Graham ; McKay, Judy: *The Software Test Engineer's Handbook: A Study Guide for the ISTQB Test Analyst and Technical Test Analyst Advanced Level Certificates 2012*. Rocky Nook, Inc., 2014

[68] Beck, Daniel ; Glick, Jesse: *CVE-2016-3101*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3101, Abruf: 23.04.2016

[69] Benedikt, Michael ; Freire, Juliana ; Godefroid, Patrice: VeriWeb: Automatically testing dynamic web sites. In: *In Proceedings of 11th International World Wide Web Conference (WWW'2002* Citeseer, 2002

[70] Berg, Alan: *Jenkins Continuous Integration Cookbook - Second Edition.* Packt Publishing Ltd, 2015

[71] Bidgoli, Hossein: *Handbook of Information Security, Information Warfare, Social, Legal, and International Issues and Security Foundations.* Bd. 2. John Wiley & Sons, 2006

[72] Boer, Jan de ; Houwelingen, Jan ; Adriani, Paul ; Bosman, Joost ; Gerretsen, Jaap ; Sala, Michiel: *All To Play: How Games Can Change IT.* http://128.199.47.117/wp-content/uploads/2016/01/C-2015-2-Boer.pdf. Version: 2015

[73] Bonwell, Charles ; Eison, James: *Active Learning: Creating Excitement in the Classroom. 1991 ASHE-ERIC Higher Education Reports.* Association for the Study of Higher Education.; ERIC, 1991 http://files.eric.ed.gov/fulltext/ED336049.pdf

[74] Bourque, Pierre ; Fairley, Richard u. a.: *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0.* IEEE Computer Society Press, 2014

[75] Bowler, Mike: *HtmlUnit.* http://htmlunit.sourceforge.net/, Abruf: 26.06.2016

[76] Braun, Frederik: *Disallow unsafe HTML templating (no-unsafe-innerhtml).* https://github.com/mozfreddyb/eslint-plugin-no-unsafe-innerhtml, Abruf: 29.05.2016

[77] Braun, Frederik: *Additional ScanJS Rules for ESLint.* GitHub, Inc. https://github.com/mozfreddyb/eslint-plugin-scanjs-rules. Version: 2015, Abruf: 23.04.2016

[78] Brausch, Stefan u. a.: *JobConfigHistory Plugin.* https://wiki.jenkins-ci.org/display/JENKINS/JobConfigHistory+Plugin, Abruf: 21.05.2016

[79] Brotby, W.: *Information security management metrics: a definitive guide to effective security monitoring and measurement.* CRC Press, 2009

[80] Brutlag, Jake: Aberrant Behavior Detection in Time Series for Network Monitoring. In: *LISA* Bd. 14, 2000, S. 139–146

[81] Bundesamt für für Sicherheit in der Informationstechnik: *IT-Grundschutz; 15. Ergänzungslieferung.* https://download.gsb.bund.de/BSI/ITGSK/IT-Grundschutz-Kataloge_2016_EL15_DE.pdf. Version: 2016, Abruf: 07.07.2016

[82] Bundesamt für Sicherheit in der Informationstechnik: Leitfaden zur Entwicklung sicherer Webanwendungen - Empfehlungen und Anforderungen an Auftraggeber aus der öffentlichen Verwaltung. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/Webanwendungen/Webanw_Auftragnehmer_pdf.pdf?__blob=publicationFile&v=1, Abruf: 04.06.2016

[83] Bundesamt für Sicherheit in der Informationstechnik: Leitfaden zur Entwicklung sicherer Webanwendungen - Empfehlungen und Anforderungen an die Auftragnehmer. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/Webanwendungen/Webanw_Auftragnehmer_pdf.pdf?__blob=publicationFile&v=1, Abruf: 04.05.2016

[84] Burn, Oliver u. a.: *checkstyle*. http://checkstyle.sourceforge.net/releasenotes.html. Version: 2016, Abruf: 23.04.2016

[85] Bühler, Joachim u. a.: Denken. Gründen. Wachsen. – 9 Vorschläge für eine wirksame Start-up-Politik. In: *BITKOM* https://opus4.kobv.de/opus4-hwr/frontdoor/deliver/index/docId/233/file/Loeckel,Birte_BA_2014.pdf, Abruf: 19.06.2016

[86] Cadariu, Mircea ; Bouwers, Eric ; Visser, Joost ; Deursen, Arie van: Tracking known security vulnerabilities in proprietary software systems. In: *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on* IEEE, 2015, S. 516–519

[87] Canonical Ltd.: *Ubuntu logo*. http://design.ubuntu.com/brand/ubuntu-logo, Abruf: 07.05.2016

[88] Chandra, Pravir: *Software Assurance Maturity Model*. http://www.opensamm.org/, Abruf: 23.02.2016

[89] Chess, Brian ; McGraw, Gary: Static analysis for security. In: *IEEE Security & Privacy* (2004), Nr. 6, S. 76–79

[90] Chess, Brian ; West, Jacob: *Secure programming with static analysis*. Pearson Education, 2007 http://www.e-reading.club/bookreader.php/142130/Secure_programming_with_Static_Analysis.pdf

[91] Christ, Jochen: paydirekt: Microservices machen die Organisation agil. In: *OBJEKTspektrum* (2016)

[92] Ciancutti, John: *5 Lessons We've Learned Using AWS*. http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html#, Abruf: 25.06.2016

[93] CISOfy: *Lynis – Open source auditing*. https://cisofy.com/lynis/. Version: 2015, Abruf: 07.05.2016

[94] Collins-Sussman, Ben ; Fitzpatrick, Brian ; Pilato, C: *Version Control with Subversion–For Subversion 1.7*. O'Reilly Media, 2011 http://svnbook.red-bean.com/en/1.7/svn-book.pdf

[95] Committee, Software & Systems u. a.: IEEE standard for software and system test documentation. In: *Fredericksburg, VA, USA: IEEE Computer Society* (2008)

[96] Copeland, Tom ; Dangel, Andreas u. a.: *PMD*. https://pmd.github.io/. Version: 2016, Abruf: 23.04.2016

[97] Cornutt, Chris u. a.: *Scanner for PHP.ini*. 2016

[98] Curiel, Johanna: *[Owasp-leaders] Are we helping Hackers or helping Application security?* http://lists.owasp.org/pipermail/owasp-leaders/2016-May/016578.html, Abruf: 21.05.2016

[99] Dahse, Johannes: RIPS-A static source code analyser for vulnerabilities in PHP scripts. In: *Retrieved: February* 28 (2010), S. 2012

[100] DB Networks Inc.: *How to Instrument for Advanced Web Application Penetration Testing.* http://www.dbnetworks.com/pdf/how-to-instrument-for-advanced-web-application-penetration-testing.pdf, Abruf: 06.04.2016

[101] De Bruin, Tonia ; Freeze, Ronald ; Kaulkarni, Uday ; Rosemann, Michael: Understanding the main phases of developing a maturity assessment model. (2005)

[102] Denim Group Ltd.: *ThreadFix.* http://www.denimgroup.com/media/pdfs/ThreadFix_brochure.pdf. Version: 2013, Abruf: 13.05.2016

[103] Dey, Arkajit ; Weis, Stephen: Keyczar: A Cryptographic Toolkit. (2008)

[104] Docker, Inc.: *Docker - Build, Ship, and Run Any App, Anywhere.* https://www.docker.com/, Abruf: 03.07.2016

[105] DreamHost, LLC: *Web Hosting,VPS, Dedicated and WordPress Hosting – DreamHost.* https://www.dreamhost.com/. Version: 2015, Abruf: 01.02.2016

[106] Du, Wenliang ; Mathur, Aditya: Testing for software vulnerability using environment perturbation. In: *Quality and Reliability Engineering International* 18 (2002), Nr. 3, S. 261–272

[107] Ebert, C. ; Gallardo, G. ; Hernantes, J. ; Serrano, N.: DevOps. In: *IEEE Software* 33 (2016), May, Nr. 3, S. 94–100. http://dx.doi.org/10.1109/MS.2016.68. – DOI 10.1109/MS.2016.68. – ISSN 0740–7459

[108] Elasticsearch BV: *docs/kibana/.* https://github.com/docker-library/docs/tree/master/kibana, Abruf: 12.07.2016

[109] Erk, Bernd: *Nagios is forked: Icinga is unleashed.* https://www.icinga.org/2009/05/06/announcing-icinga/. Version: 2009, Abruf: 01.06.2016

[110] Falk, Andreas: Agil, aber sicher? Security im agilen Entwicklungsprozess. (2016). https://www.owasp.org/images/1/13/Agil_aber_sicher_owasp_muenchen_-_Andreas_Falk.pdf, Abruf: 11.06.2016

[111] Felderer, Michael ; Büchler, Matthias ; Johns, Martin ; Brucker, Achim ; Breu, Ruth ; Pretschner, Alexander: Security Testing: A Survey. In: *Advances in Computers* (2015)

[112] Felter, Wes ; Ferreira, Alexandre ; Rajamony, Ram ; Rubio, Juan: An updated performance comparison of virtual machines and Linux containers. In: *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, 2015, S. 171–172

[113] Ferrara, Anthony: *A Static Analyzer Security Scanner (for PHP).* https://github.com/ircmaxell/php-security-scanner. Version: 2015, Abruf: 06.04.2016

[114] Finifter, Matthew ; Mettler, Adrian ; Sastry, Naveen ; Wagner, David: Verifiable Functional Purity in Java. In: *Proceedings of the 15th ACM Conference on Computer and Communications Security.* New York, NY, USA : ACM, 2008 (CCS '08). – ISBN 978–1–59593–810–7, 161–174

[115] Forster, Florian: *collectd – The system statistics collection daemon.* https://collectd.org/, Abruf: 11.07.2016

[116] Forster, Florian: *File:Logo der Software collectd.svg.* https://commons.wikimedia.org/wiki/File:Logo_der_Software_collectd.svg, Abruf: 11.07.2016

[117] Gabovitch, Iwan: *Clipart.* https://openclipart.org/detail/212730/web-browser, Abruf: 11.07.2016

[118] Gabriel Lawrence, Chris: *Marshalling Pickles - how deserializing oobject can ruin your day.* http://de.slideshare.net/frohoff1/appseccali-2015-marshalling-pickles, Abruf: 30.05.2016

[119] Gallagher, Scott: *Mastering Docker.* Packt Publishing Ltd, 2015

[120] Gaur, Nitin u. a.: *KVM Myths - Uncovering the Truth about the Open Source Hypervisor.* https://www.ibm.com/developerworks/community/blogs/ibmvirtualization/entry/kvm_myths_uncovering_the_truth_about_the_open_source_hypervisor?lang=en. Version: 2012, Abruf: 04.06.2016

[121] Gravi, Tianon: *docker-brew-ubuntu-core/Dockerfile [...].* https://github.com/tianon/docker-brew-ubuntu-core/blob/d7f2045ad9b08962d9728f6d9910fa252282b85f/xenial/Dockerfile. Version: 2016, Abruf: 23.01.2016

[122] Grover, Varun: An Empirically Derived Model for the Adoption of Customer-based Interorganizational Systems*. In: *Decision sciences* 24 (1993), Nr. 3, 603–640. https://www.researchgate.net/profile/Varun_Grover/publication/229748221_An_Empirically_Derived_Model_for_the_Adoption_of_Customer-Based_Interorganizational_Systems/links/00b7d518a9d5c42e07000000.pdf

[123] Gruhn, Volker ; Schäfer, Clemens: BizDevOps: Because DevOps is Not the End of the Story. In: *Intelligent Software Methodologies, Tools and Techniques.* Springer, 2015, S. 388–398

[124] Haen, Christophe ; Bonaccorsi, Enrico ; Neufeld, Niko: Distributed monitoring system based on icinga. In: *Proceedings of ICALEPCS2011* (2011), S. 1149–1152

[125] Halili, Emily: *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites.* Packt Publishing Ltd, 2008

[126] Harris, Kamala ; General, Attorney: California data breach report. (2016). https://oag.ca.gov/sites/all/files/agweb/pdfs/dbr/2016-data-breach-report.pdf?

[127] Harrison, Reuven: Reducing complexity in securing heterogeneous networks. In: *Network Security* 2015 (2015), Nr. 10, S. 11–13

[128] Hayden, Lance: *IT Security Metrics: A Practical Framework for Measuring Security & Protecting Data*. McGraw Hill Professional, 2010

[129] He, Peng ; Li, Bing ; Liu, Xiao ; Chen, Jun ; Ma, Yutao: An empirical study on software defect prediction with a simplified metric set. In: *Information and Software Technology* 59 (2015), S. 170–190

[130] Hedin, Daniel ; Sabelfeld, Andrei: Information-Flow Security for a Core of JavaScript. In: *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*. Washington, DC, USA : IEEE Computer Society, 2012 (CSF '12). – ISBN 978–0–7695–4718–3, 3–18

[131] Hein, Matthias: Aufgaben eines modernen Monitorings: Streitfragen. In: *IT-Administrator Sonderheft. Monitoring - Methoden und Werkzeuge zur Leistungs- und Verfügbarkeitsüberwachung von IT-Systemen* (2015)

[132] Hoorn, Andre van ; Vögele, Christian ; Schulz, Eike ; Hasselbring, Wilhelm ; Krcmar, Helmut: Automatic Extraction of Probabilistic Workload Specifications for Load Testing Session-Based Application Systems. In: *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools (ValueTools 2014)*, 2014, 139–146

[133] Hovemeyer, David ; Pugh, William: Finding more null pointer bugs, but not too many. In: *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* ACM, 2007, S. 9–14

[134] Hovemeyer, David ; Spacco, Jaime ; Pugh, William: Evaluating and tuning a static analysis to find null pointer bugs. In: *ACM SIGSOFT Software Engineering Notes* Bd. 31 ACM, 2005, S. 13–19

[135] Huang, Yao-Wen ; Yu, Fang ; Hang, Christian ; Tsai, Chung-Hung ; Lee, Der-Tsai ; Kuo, Sy-Yen: Securing web application code by static analysis and runtime protection. In: *Proceedings of the 13th international conference on World Wide Web* ACM, 2004, S. 40–52

[136] IBM Corporation: *IBM Knowledge Center - Glass box overview.* `http://www.ibm.com/support/knowledgecenter/SSPH29_8.6.0/com.ibm.help.common.infocenter.aps/c_GlassBoxScanning.html`, Abruf: 23.04.2016

[137] IBM Deutschland GmbH: *Funktionsweise der Korrelation (Hybridanalyse).* `http://www-01.ibm.com/support/knowledgecenter/SSW2NF_8.8.0/com.ibm.ase.help.doc/topics/c_how_correlation_works.html?lang=de`, Abruf: 08.06.2016

[138] ISO/IEC: ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. ISO/IEC, 2010

[139] Jansen, Bernard: Search log analysis: What it is, what's been done, how to do it. In: *Library & information science research* 28 (2006), Nr. 3, S. 407–432

[140] Jaquith, Andrew: *Security metrics*. Pearson Education, 2007

[141] Jaspan, Ciera ; Chen, I ; Sharma, Anoop u. a.: Understanding the value of program analysis tools. In: *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion* ACM, 2007, S. 963–970

[142] Jiang, Z. ; Hassan, A. ; Hamann, G. ; Flora, P.: Automatic identification of load testing problems. In: *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, 2008. – ISSN 1063–6773, S. 307–316

[143] Jovanonic, Nenad ; Klee, Oliver: *Pixy*. `https://github.com/oliverklee/pixy`. Version: 2014, Abruf: 07.05.2016

[144] Jovanovic, Nenad: *DISSERTATION - Web Application Security*, Technischen Universität Wien, Diss., 2007. `https://github.com/oliverklee/pixy/blob/master/doc/2007-thesis-jovanovic.pdf`, Abruf: 12.04.2016

[145] Kadlecsik, Jozsef u. a.: *The netfilter.org ïptables"project*. `http://www.netfilter.org/projects/iptables/index.html`, Abruf: 23.06.2016

[146] Kam, John ; Ullman, Jeffrey: Global data flow analysis and iterative algorithms. In: *Journal of the ACM (JACM)* 23 (1976), Nr. 1, S. 158–171

[147] Kavis, Mike: *Nagios is not a monitoring strategy*. `http://devops.com/2014/04/15/nagios-monitoring-strategy/`. Version: 2014, Abruf: 02.07.2016

[148] Kimminich, Björn: *Juice Shop*. `https://bkimminich.github.io/juice-shop/#/`, Abruf: 07.07.2016

[149] Kivity, Avi ; Kamay, Yaniv ; Laor, Dor ; Lublin, Uri ; Liguori, Anthony: kvm: the Linux virtual machine monitor. In: *Proceedings of the Linux symposium* Bd. 1, 2007, S. 225–230

[150] König, Hartmut: *Protocol Engineering: Prinzip, Beschreibung und Entwicklung von Kommunikationsprotokollen*. Springer-Verlag, 2013

[151] Kopper, Karl: *The Linux Enterprise Cluster: build a highly available cluster with commodity hardware and free software*. No Starch Press, 2005

[152] Krinke, Jens: Is cloned code more stable than non-cloned code? In: *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on* IEEE, 2008, S. 57–66

[153] Krüger, Andreas: Achtung: Baustelle in Gefahr - Risiken beim Bau von Anwendungen und Lösungsansätze für die Sicherheit. In: *iX Developer 2016* Effektiver entwickeln (2016)

[154] Küchenhoff, Helmut: Vorlesung: Statistik I für Statistiker, Mathematiker und Informatiker. `http://www.statistik.lmu.de/~groll/DeskriptiveStatistikWS20102011/Folien_4.pdf`

[155] Kullmann, Heide-Marie ; Seidel, Eva ; Erwachsenenbildung, Deutsches: *Lernen und Gedächtnis im Erwachsenenalter*. Bertelsmann, 2000

[156] Lackey, Zane: *Tips for Building a Modern Security Engineering Organization | Georgian Partners*. https://georgianpartners.com/tips-for-building-a-modern-security-engineering-organization/. Version: 2016, Abruf: 26.02.2016

[157] Landi, William: Undecidability of Static Analysis. In: *ACM Lett. Program. Lang. Syst.* 1 (1992), Nr. 4, 323–337. http://dx.doi.org/10.1145/161494.161501. – DOI 10.1145/161494.161501. – ISSN 1057–4514

[158] Laskos, Tasos: *arachni - web application security scanner framework*. http://www.arachni-scanner.com/. Version: 2016, Abruf: 07.05.2016

[159] LaToza, Thomas ; Myers, Brad: Hard-to-answer questions about code. In: *Evaluation and Usability of Programming Languages and Tools* ACM, 2010, S. 8

[160] Lerner, Josh ; Tirole, Jean: The scope of open source licensing. In: *Journal of Law, Economics, and Organization* 21 (2005), Nr. 1, S. 20–56

[161] Lietz, Petra: Research into questionnaire design. In: *International Journal of Market Research* 52 (2010), Nr. 2, S. 249–272

[162] Long, Jeremy u. a.: *OWASP Dependency Check*. https://www.owasp.org/index.php/OWASP_Dependency_Check, Abruf: 07.05.2016

[163] Long, Quentin u. a.: *Easy & Flexible Alerting With ElasticSearch*. https://github.com/Yelp/elastalert. Version: 2016, Abruf: 04.07.2016

[164] MacDonald, Neil: *Interactive Application Security Testing*. http://blogs.gartner.com/neil_macdonald/2012/01/30/interactive-application-security-testing/, Abruf: 04.06.2016

[165] MacDonald, Neil ; Feiman, Joseph: Magic Quadrant for Dynamic Application Security Testing. In: *Gartner, https://info. veracode. com/analyst-report-gartner-dastmagic-quadrant. html, accessed* (2011)

[166] Madeyski, Lech ; Jureczko, Marian: Which process metrics can significantly improve defect prediction models? An empirical study. In: *Software Quality Journal* 23 (2015), Nr. 3, S. 393–422

[167] Matthias, Karl ; Kane, Sean: *Docker Up & Running*. O'Reilly, 2015

[168] McGrath, R ; Akkerman, W u. a.: *strace - linux syscall tracer*. https://sourceforge.net/projects/strace/?source=navbar, Abruf: 07.05.2016

[169] Medeiros, Ibéria: *WAP - Web Application Protection*. http://awap.sourceforge.net/. Version: 2015, Abruf: 16.04.2016

[170] Mesbah, A. ; Bozdag, E. ; Deursen, A. v.: Crawling AJAX by Inferring User Interface State Changes. In: *Web Engineering, 2008. ICWE '08. Eighth International Conference on*, 2008, S. 122–134

[171] Mesbah, Ali ; Van Deursen, Arie: Invariant-based automatic testing of AJAX user interfaces. In: *Proceedings of the 31st International Conference on Software Engineering* IEEE Computer Society, 2009, S. 210–220

[172] Meucci, Matteo ; Muller, Andrew: *OWASP Testing Guide v4*. `https://www.owasp.org/index.php/OWASP_Testing_Project`. Version: 2015, Abruf: 23.01.2016

[173] Microsoft: *Microsoft Threat Modeling Tool 2016*. `https://www.microsoft.com/en-us/download/details.aspx?id=49168`. Version: 2016, Abruf: 07.05.2016

[174] Mike Perry, Hans: Reproducible Builds, 2014

[175] Milligan, D.: Securing a railway control system. In: *System Safety and Cyber Security, 9th IET International Conference on*, 2014, S. 1–6

[176] Mota, Aneudy u. a.: *Audit Trail Plugin*. `https://wiki.jenkins-ci.org/display/JENKINS/Audit+Trail+Plugin`, Abruf: 21.05.2016

[177] Mourad, Azzam ; Laverdière, Marc-André ; Debbabi, Mourad: Security hardening of open source software. In: *PST*, 2006, S. 43

[178] Myers, Andrew: JFlow: Practical Mostly-static Information Flow Control. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA : ACM, 1999 (POPL '99). – ISBN 1–58113–095–3, 228–241

[179] NGINX, INC.: *NGINX | High Performance Load Balancer, Web Server, & Reverse Prox*. `https://www.nginx.com`, Abruf: 12.07.2016

[180] Nickolaevsky, Alex ; Licht, Roni ; Or, Itai: *Multijob Plugin*. `https://wiki.jenkins-ci.org/display/JENKINS/Multijob+Plugin`. Version: 2016, Abruf: 07.07.2016

[181] Nieto, Carlos: *Using the docker command to root the host*. `http://reventlov.com/advisories/using-the-docker-command-to-root-the-host`, Abruf: 07.05.2016

[182] Oftedal, Erlend: *Retire.js*. `http://retirejs.github.io/retire.js/`, Abruf: 07.05.2016

[183] Oppliger, Rolf: *Computersicherheit: Eine Einführung*. Springer-Verlag, 2013

[184] Oracle America, Inc.: *OpenJDK*. `http://openjdk.java.net/`, Abruf: 07.05.2016

[185] Osrael, Johannes ; Froihofer, Lorenz ; Gladt, Matthias ; Goeschka, Karl: Adaptive voting for balancing data integrity with availability. In: *OTM Confederated International ConferencesÖn the Move to Meaningful Internet Systems"* Springer, 2006, S. 1510–1519

[186] OWASP Foundation: *OWASP Dependency-Check Plugin*. `OWASPDependency-CheckPlugin`, Abruf: 25.06.2016

[187] OWASP Foundation: *How does dependency-check work?* `http://jeremylong.github.io/DependencyCheck/general/internals.html`. Version: 2016, Abruf: 07.05.2016

[188] OWASP Foundation: *OWASP Zed Attack Proxy Project*. https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project. Version: 2016, Abruf: 07.05.2016

[189] Pagel, Timo: *Lehrkonzept für das Modul Sicherheit in Webanwendungen*. http://files.timo-pagel.de/lehre/lehrkonzept.pdf. Version: 2015, Abruf: 07.05.2016

[190] Patnaik, Nishant ; Sahoo, Sarathi: JavaScript static security analysis made easy with JSPrime, 2013

[191] Paulk, Mark: *Capability maturity model for software*. Wiley Online Library, 1993

[192] Peeters, Johan: Agile security requirements engineering. In: *Symposium on Requirements Engineering for Information Security*, 2005

[193] Philips, Matthew: *Knight Shows How to Lose $440 Million in 30 Minutes*. http://www.bloomberg.com/news/articles/2012-08-02/knight-shows-how-to-lose-440-million-in-30-minutes. Version: 2012, Abruf: 07.05.2016

[194] Phongthiproek, Prathan: *tanprathan/OWASP-Testing-Checklist*. https://github.com/tanprathan/OWASP-Testing-Checklist, Abruf: 07.07.2016

[195] Pilshofer, Birgit: Wie erstelle ich einen Fragebogen. In: *Ein Leitfaden für die Praxis* 2 (2001)

[196] Platt, Eric: *Check Out Knight Capital's Furious Recovery As A Player In The Markets*. http://www.businessinsider.com/knight-capital-market-share-2012-8?IR=T. Version: 2012, Abruf: 15.04.2016

[197] Polk, Timothy: *Automated tools for testing computer system vulnerability*. DIANE Publishing, 1992

[198] Positive Technologies - Application Inspector Team: *HOW TO REVEAL APPLICATION VULNERABILITIES? SAST, DAST, IAST AND OTHERS*. http://www.ptsecurity.com/upload/iblock/96a/96ad349b5ea77f0133e6a65be80096ee.pdf, Abruf: 07.06.2016

[199] Potencier, Fabien u. a.: *SensioLabs Security Checker*. https://github.com/sensiolabs/security-checker. Version: 2015, Abruf: 17.06.2016

[200] Pulkkinen, Ville: Continuous Deployment of Software. In: *Cloud-Based Software Engineering*

[201] Puppet Labs, Pricewaterhouse: *2015 State of DevOps Report*. https://puppetlabs.com/2015-devops-report. Version: 2015, Abruf: 19.02.2016

[202] Puppet Labs, Pricewaterhouse: *2016 State of DevOps Report*. https://puppet.com/resources/white-paper/2016-state-of-devops-report. Version: 2016, Abruf: 27.06.2016

[203] Radatz, Jane ; Geraci, Anne ; Katki, Freny: IEEE standard glossary of software engineering terminology. In: *IEEE Std* 610121990 (1990), Nr. 121990, S. 3

[204] Radwan, H. ; Prole, K.: Code Pulse: Real-time code coverage for penetration testing activities. In: *Technologies for Homeland Security (HST), 2015 IEEE International Symposium on*, 2015, S. 1–6

[205] Rathbun, D ; Homsher, L: Gathering security metrics and reaping the rewards. In: *SANs Institute* (2009)

[206] Redman, Thomas: The impact of poor data quality on the typical enterprise. In: *Communications of the ACM* 41 (1998), Nr. 2, S. 79–82

[207] Reed, J.: *DevOps in Practice*. O'Reilly Media, Inc., 2014

[208] Reissmann, Sven: *Korrelation und Aggregation von Logging-Informationen verteilter Systeme unter Anwendung von Complex Event Processing*, Hochschule Fulda, Masterthesis, 2015. `https://0x80.io/pub/files/Masterthesis.pdf`, Abruf: 25.06.2016

[209] Reshetova, Elena ; Karhunen, Janne ; Nyman, Thomas ; Asokan, N: Security of OS-level virtualization technologies. In: *Secure IT Systems*. Hamburg, Germany : Springer, 2014 (Vortrag beim 31C3)

[210] Riancho, Andres: *w3af - Open Source Web Application Security Scanner*. `http://w3af.org/`, Abruf: 07.05.2016

[211] Rutar, Nick ; Almazan, Christian ; Foster, Jeffrey: A comparison of bug finding tools for Java. In: *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on* IEEE, 2004, S. 245–256

[212] SA, High-Tech: *Web Application Security Testing: SAST, DAST or IAST?* `https://www.htbridge.com/blog/web-application-security-testing-sast-dast-or-iast.html`. Version: 2015, Abruf: 07.05.2016

[213] Sakai, Minoru: *SecurityFocus, Jenkins Multiple Cross Site Scripting and Directory Traversal Vulnerabilities*. `http://www.securityfocus.com/bid/52384/`. Version: 2012, Abruf: 07.05.2016

[214] Schreiber, Thomas ; Hoffmann, Thomas: Sicherheit von Webanwendungen-Maßnahmenkatalog und Best Practices. In: *Bundesamt für Sicherheit in der Informationstechnilk, Version* 1 (2006)

[215] Schulz, Eike ; Goerigk, Wolfgang ; Hasselbring, Wilhelm ; Hoorn, Andre van ; Knoche, Holger: Model-Driven Load and Performance Test Engineering in DynaMod. In: *Proceedings of the Workshop on Model-based and Model-driven Software Modernization (MMSM '14)* Bd. 34, Softwaretechnik-Trends, August 2014, 38–39

[216] Schwaber, Ken ; Sutherland, Jeff ; Beedle, Mike: Der Scrum Guide: Der gültige Leitfaden für Scrum. (2013). `http://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-DE.pdf#zoom=100`

[217] Scott, David ; Sharp, Richard: Abstracting Application-level Web Security. In: *Proceedings of the 11th International Conference on World Wide Web*. New York, NY, USA : ACM, 2002 (WWW '02). – ISBN 1–58113–449–5, 396–407

[218] Securities and Exchange Commission: *Release No. 67347*. `https://www.sec.gov/rules/sro/nyse/2012/34-67347.pdf`. Version: 2012, Abruf: 07.05.2016

[219] Securities and Exchange Commission: *Release No. 70694*. https://www.sec.gov/litigation/admin/2013/34-70694.pdf. Version: 2012, Abruf: 07.05.2016

[220] Seidl, Helmut ; Wilhelm, Reinhard ; Hack, Sebastian: *Übersetzerbau: Band 3: Analyse und Transformation*. Springer-Verlag, 2009

[221] Shang, W.: Bridging the divide between software developers and operators using logs. In: *2012 34th International Conference on Software Engineering (ICSE)*, 2012. – ISSN 0270–5257, S. 1583–1586

[222] Shauver, Dan: Beyond Patch Management / SANS™ Institute. Version: 2004. https://www.sans.org/reading-room/whitepapers/bestprac/patch-management-1420, Abruf: 24.06.2016. 2004

[223] Siponen, Mikko ; Baskerville, Richard ; Kuivalainen, Tapio: Integrating security into agile development methods. In: *System Sciences, 2005. HICSS'05. Proceedings of the 38th Annual Hawaii International Conference on* IEEE, 2005, S. 185a–185a

[224] Solís, Carlos ; Wang, Xiaofeng: A study of the characteristics of behaviour driven development. In: *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on* IEEE, 2011, S. 383–387

[225] Soukup, Tom ; Davidson, Ian: *Visual data mining: Techniques and tools for data visualization and mining*. John Wiley & Sons, 2002

[226] Stampar, Miroslav ; Damele, Bernardo u. a.: *sqlmap*. http://sqlmap.org/. Version: 2016, Abruf: 22.03.2016

[227] Steinegger, Roland ; Hintz, Nadina ; Hippchen, Benjamin ; Binder, Georg ; Röser, Florian ; Abeck, Sebastian: Analyse von Logs mit Open-Source-Werkzeugen. In: *Software-Technologien und Prozesse: Open Source Software in der Industrie, KMUs und im Hochschulumfeld 5. Konferenz STEP, 3.5. 2016 in Furtwangen* Walter de Gruyter GmbH & Co KG, 2016, S. 7

[228] Stock, Gary: OWASP Code Review Guide 2.0. In: *The OWASP Foundation Guidelines* (2016)

[229] Stoneburner, Gary ; Goguen, Alice ; Feringa, Alexis: 800-30. In: *Risk management guide for information technology systems* (2002), S. 800–30

[230] Stuttard, Dafydd ; Pinto, Marcus: *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. John Wiley & Sons, 2011

[231] Swartout, Paul: *Continuous Delivery and DevOps–A Quickstart Guide*. Packt Publishing Ltd, 2014

[232] Süddeutsche.de GmbH: *Investoren beenden "Knightmare on Wall Street"*. http://www.sueddeutsche.de/wirtschaft/nach-software-panne-bei-aktienhaendler-investoren-beenden-knightmare-on-wall-street-1.1433571. Version: 2012, Abruf: 23.01.2016

[233] T. Bray, Ed.: The JavaScript Object Notation (JSON) Data Interchange Format / Internet Engineering Task Force. Version: 2014. https://tools.ietf.org/html/rfc7159.html. RFC Editor, 2014 (7159). – 1–56 S.. – ISSN 2070–1721

[234] Tappenden, A. ; Beatty, P. ; Miller, J. ; Geras, A. ; Smith, M.: Agile security testing of Web-based systems via HTTPUnit. In: *Agile Development Conference (ADC'05)*, 2005, S. 29–38

[235] Tassey, Gregory: The economic impacts of inadequate infrastructure for software testing. In: *National Institute of Standards and Technology, RTI Project* 7007 (2002), Nr. 011

[236] Team, Verizon: 2015 Data Breach Investigations Report. (2015)

[237] The Center for Internet Security: *Critical Security Controls forEffective Cyber Defense - Version 6.0.* https://www.cisecurity.org/critical-controls/download.cfm?f=CSC-MASTER-VER%206.0%20CIS%20Critical%20Security%20Controls%2010.15.2015. Version: 2015, Abruf: 07.03.2016

[238] Turnbull, James: *The Logstash Book; Log management made easy.* 2016

[239] Typesafe Inc. u. a.: *Play Framework - Build Modern & Scalable Web Apps with Java and Scala.* https://www.playframework.com/, Abruf: 07.03.2016

[240] Vähä-Sipilä, Antti: *Software security in agile product management.* https://fokkusu.fi/agile-security/Software%20security%20in%20agile%20product%20management.pdf. Version: 2011

[241] Viega, John ; Bloch, Jon-Thomas ; Kohno, Yoshi ; McGraw, Gary: ITS4: A static vulnerability scanner for C and C++ code. In: *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference* IEEE, 2000, S. 257–267

[242] Voas, Jeffrey ; Ghosh, Anup ; McGraw, Gary ; Charron, FACF ; Miller, KAMK: Defining an adaptive software security metric from a dynamic software failure tolerance measure. In: *Computer Assurance, 1996. COMPASS'96, Systems Integrity. Software Safety. Process Security. Proceedings of the Eleventh Annual Conference on* IEEE, 1996, S. 250–263

[243] vorre: *Clipart - Folder.* https://openclipart.org/detail/74023/folder, Abruf: 07.03.2016

[244] WALLACH, Dan: Technical Perspective Smartphone Security'Taint'What It Used to Be. In: *Communications of the ACM* 57 (2014), Nr. 3

[245] Watson, Colin u. a.: *AppSensor DetectionPoints.* https://www.owasp.org/index.php/AppSensor_DetectionPoints. Version: 2015, Abruf: 03.04.2016

[246] Wetter, Dirk: *OWASP Top 10: Zwei Jahre danach.* SP Gabler Verlag, 2012

[247] Wetter, Dirk: *testssl.sh: Testing TLS/SSL encryption.* https://testssl.sh/. Version: 2016, Abruf: 07.05.2016

[248] Wichers, D u. a.: OWASP Top 10. In: *Open Web Application Security Project (OWASP)* (2013)

[249] Willis, John: *DevOps Culture (Part 1) - IT Revolution.* `http://itrevolution.com/devops-culture-part-1/`. Version: Mai 2012, Abruf: 07.05.2016

[250] Witte, Frank: Testmanagement und Softwaretest. (2016)

[251] Xavier, Miguel ; Neves, Marcelo ; Rossi, Fabio ; Ferreto, Tiago ; Lange, Tobias ; De Rose, Cesar: Performance evaluation of container-based virtualization for high performance computing environments. In: *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on* IEEE, 2013, S. 233–240

[252] Yanggratoke, R. ; Ahmed, J. ; Ardelius, J. ; Flinta, C. ; Johnsson, A. ; Gillblad, D. ; Stadler, R.: Predicting real-time service-level metrics from device statistics. In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2015. – ISSN 1573–0077, S. 414–422

[253] Younge, Andrew ; Henschel, Robert ; Brown, James ; Von Laszewski, Gregor ; Qiu, Judy ; Fox, Geoffrey: Analysis of virtualization technologies for high performance computing environments. In: *Cloud Computing (CLOUD), 2011 IEEE International Conference on* IEEE, 2011, S. 9–16

[254] Zakas, Nicholas: *ESLint - The pluggable linting utility for JavaScript and JSX.* `http://eslint.org/`. Version: 2014, Abruf: 07.05.2016

[255] Zander, Justyna ; Schieferdecker, Ina ; Mosterman, Pieter: *Model-based testing for embedded systems.* CRC press, 2011

[256] Zurko, Mary ; Simon, Richard: User-centered Security. In: *Proceedings of the 1996 Workshop on New Security Paradigms.* New York, NY, USA : ACM, 1996 (NSPW '96). – ISBN 0–89791–944–0, 27–33

[257] Ödegaard, Torkel: *grafana/public/img/fav_dark_32.png.* `https://github.com/grafana/grafana/blob/dc9dcd045daece1436c5c40e293f408b71399537/public/img/fav_dark_32.png`, Abruf: 11.07.2016

Anhang A

Protokoll der Erzeugung vom *Docker*-Abbild *ubuntu:16.04*

```
1  Sending build context to Docker daemon  47.6 MB
2  Sending build context to Docker daemon
3  Step 0 : FROM scratch
4   --->
5  Step 1 : ADD ubuntu-xenial-core-cloudimg-amd64-root.tar.gz /
6   ---> a95eed35be4a
7  Removing intermediate container 18c538464b4d
8  Step 2 : RUN echo '#!/bin/sh' > /usr/sbin/policy-rc.d   && echo 'exit
       101' >> /usr/sbin/policy-rc.d     && chmod +x /usr/sbin/policy-rc.d
                  && dpkg-divert --local --rename --add /sbin/initctl
           && cp -a /usr/sbin/policy-rc.d /sbin/initctl    && sed -i 's/^
       exit.*/exit 0/' /sbin/initctl             && echo 'force-unsafe-io' >
        /etc/dpkg/dpkg.cfg.d/docker-apt-speedup             && echo 'DPkg::
       Post-Invoke { "rm -f /var/cache/apt/archives/*.deb /var/cache/apt/
       archives/partial/*.deb /var/cache/apt/*.bin || true"; };' > /etc/apt/
       apt.conf.d/docker-clean   && echo 'APT::Update::Post-Invoke { "rm -f
       /var/cache/apt/archives/*.deb /var/cache/apt/archives/partial/*.deb /
       var/cache/apt/*.bin || true"; };' >> /etc/apt/apt.conf.d/docker-clean
          && echo 'Dir::Cache::pkgcache ""; Dir::Cache::srcpkgcache "";' >>
       /etc/apt/apt.conf.d/docker-clean              && echo 'Acquire::
       Languages "none";' > /etc/apt/apt.conf.d/docker-no-languages
               && echo 'Acquire::GzipIndexes "true"; Acquire::
       CompressionTypes::Order:: "gz";' > /etc/apt/apt.conf.d/docker-gzip-
       indexes
9   ---> Running in 3698c3ff1ebc
10 Adding 'local diversion of /sbin/initctl to /sbin/initctl.distrib'
```

```
11  ---> 63b41cc64802
12 Removing intermediate container 3698c3ff1ebc
13 Step 3 : RUN sed -i 's/^#\s*\(deb.*universe\)$/\1/g' /etc/apt/sources.
       list
14  ---> Running in ed0d1390fd5a
15  ---> b801b6b3a634
16 Removing intermediate container ed0d1390fd5a
17 Step 4 : CMD /bin/bash
18  ---> Running in f96f77b00e4c
19  ---> 99e4a2d62a80
20 Removing intermediate container f96f77b00e4c
21 Successfully built 99e4a2d62a80
```

Anhang B

Experten-Interview Protokolle

B.1 Zusammenfassung des Interviews mit Björn Kimminich

Berufsbezeichnung: IT Architect & Security Officer, Kühne + Nagel
Durchführung: 03.02.2016
Ein Deployment-Server hat einen sehr hohen Schutzbedarf.
Weiter wurde angemerkt, dass der *ELK-Stack* leider keine Rollen-basierte Zugriffskontrolle enthält, in welcher Rechte für bestimmte Personen für bestimmte Server gesetzt werden.

B.2 Zusammenfassung des Interviews mit Christian Schneider

Berufsbezeichnung: Software Developer, Whitehat Hacker & Trainer, Christian Schneider Softwareentwicklung & IT-Security
Durchführung: 20.02.2016
Dokumentation zum *Security DevOps Maturity Model* existiert nicht. Das Model basiert auf persönlichen Erfahrungen. Es ist geplant ab November 2016 weitere Blog-Posts zu veröffentlichen.
Die Beurteilung der Implementierung sollte nicht in Personentagen erfolgen, sondern beispielsweise in Aufwandsklassen. Dies hat mehrere Gründe. Einerseits arbeiten Teams agil, so dass eine andere Größenordnung genutzt wird. Auch ist fraglich, wie viel KnowHow in einem Team steckt. Dabei müssen bei der Implementierung Kenntnisse im Bereich Entwicklung, System-Administration sowie Sicherheit vorhanden sein. Die ersten Ebenen („Gurte“) sind mit vergleichsweise wenig Aufwand zur Einführung für ein Team erreichbar. Aufwände bei der zu programmierenden Implementierung von Security-Tests können schnell groß werden und daher sind diese Stufen erst in höheren "Gurten" im Modell erscheinen, damit sich der vergleichsweise hohe Aufwand lohnt. Denn die einfacheren "Gurte" kommen mit teilweise deutlich geringerem Aufwand einher und liefern bereits erste Ergebnisse. Erst wenn dieses Potenzial ausgereizt ist, lohnt sich der Mehrwert der höheren Aufwände der höheren Gurte.
Im Weiteren spielen Abhängigkeiten wie:

- Projektziel,
- Know-How und

- Vorgehensweise

eine Rolle.
In den ersten Ebenen des Reifegradmodells sollte entsprechend mit möglichst wenig Programmierung das Ziel erreicht werden können.
Abgrenzung:

- Machbarkeit: Es sollte in kleinen Schritten ohne kommerzielle Lösungen die Implementierung möglich sein.
- Kultur der Leute

Es kann zu Erodierung bei zu vielen False-Positives, beispielsweise kann die Blockierung eines Builds aufgrund eines False-Positives zu „Backfire“ führen. Insbesondere SAST-Lösungen haben erhöhte False-Positives, weshalb nur bei hoher Kritikalität ein Alarm erzeugt werden sollte. Tests mittels DAST-Lösungen haben weniger False-Positives, dafür auch mehr False-Negatives.
Daher sind sinnvollerweise beide kombiniert einzusetzen und konsolidiert im Ergebnis zu betrachten (siehe SDOMM-Achse "Konsolidierung").
Interessant ist auch IAST als Mix zwischen SAST und DAST. Generell kann man auch kommerzielle Lösungen im SDOMM gut einbinden, das Modell schreibt auch nichts zu konkreten Werkzeuge vor, sondern zeigt (Stichwort einfache Einführung) mögliche Beispiele anhand von verfügbaren OpenSource Lösungen auf.
SecDevOps sollte als Job-Enabler gesehen werden.
Bei DevOps geht es auch um Personen und Kultur, entsprechend sollten Sicherheitsexperten mit in die Teams, beispielsweise in Form eines „Security-Champions“.

B.3 Zusammenfassung des Interviews mit Prof. Dr. Wilhelm Hasselbring

Berufsbezeichnung: Prof., Universität zu Kiel
Datum: 22.02.2016
Wie kann Sicherheit in DevOps-Kultur integriert werden?

Sensibilisierung für Sicherheit

Welche Kriterien sind zur Bewertung der Abzisse „Einfachheit der Implementierung“ notwendig?

Benötigtes Wissen

IT-Bereiche:

Sicherheit
Entwicklung
System-Administration

Zeit

Benötigte System-Ressourcen

Überschneidung von Sicherheit und *DevOps*, beispielsweise ist folgendes durch *DevOps* gegeben:

- Ausfallsicherheit und Zuverlässigkeit
- Funktionstests und Lasttests
- Negationstests

Sehen Sie einen Zusammenhang zwischen DevOps und agilen Methoden wie Scrum?
Auf jeden Fall. *DevOps* gibt frühes Feedback (Stichwort: Fail Fast).

B.4 Zusammenfassung des Interviews mit Stefan Rieber

Berufsbezeichnung: Leiter IT, Institut für Weltwirtschaft an der Universität Kiel
Datum: 23.02.2016
Gehört Burn-Out Prävention zu *DevOps*?
Der Nutzen des Reifegradmodells errechnet sich aus Opportunitätskosten.
Maßrahmendichte erhöht generischen Nutzen. Ggf. sind ITIL-Level interessant.
Berechnung des Aufwands der Implementierung wird mit „sehr gering“, „gering“, „mittel“, „hoch“, „sehr hoch“ empfohlen, um die ausführenden Personen nicht unter Druck zu setzen. Werden konkrete Tage verwendet, so müssen sich die Personen vor dem Management verantworten.
DevOps fehlt ggf. eine Diskussionsplattform.
Bewertungsempfehlung für die Evaluierung einer Dimension: IT-Sicherheitsexperten erhalten eine (Unter-)Dimension und füllen diese auf Papier aus.

B.5 Zusammenfassung des Interviews mit Benjamin Pfänder

Berufsbezeichnung / Unternehmen: Software / Security Ingenieur, Iteratec GmbH
Datum: 09.03.2016
Feature Toggles können neben der Gefahr „Toten Quellcode“ einer Webanwendungen zu bergen auch für Sicherheitsmaßnahmen genutzt werden, welche bei Angriff aktiviert werden. Beispielweise CAPTCHAs.
Bei „Sehr hohes Verständnis von Sicherheitspraktiken bei Skalierung“ sollten Teams von sich aus motiviert auf Sicherheit zu achten.
Security-Champions sind leider unterschiedlich stark motiviert.
Wir überlegen Sicherheitsprüfungen durch die Teams selbst durchführen zu lassen. Hier prüft ein Team die Webanwendung eines anderen Teams.

B.6 Zusammenfassung des Interviews mit Matthias Rohr

Berufsbezeichnung: Senior Consultant, Secodis GmbH
Datum: 08.03.2016
Artikel „Sicherheit im Software-Entwicklungsprozess“ -> „Die Möglichkeit, auch intern, zumindest einfache Sicherheitstests durchführen zu können, ist jedoch gerade bei agilen Projekten mit laufenden, potentiell sicherheitsrelevanten, Neuerungen von großer Wichtigkeit. „

- Wo können diese helfen?
- Welche Prüfungen sind am Wichtigsten?

B.7 Zusammenfassung des Interviews mit einer anonymen Person

Berufsbezeichnung: Entwickler in einem auf Sicherheits-Metriken spezialisiertem Unternehmen
Datum: 15.03.2016
Erfassung von „Top Ten Talker“ und „Bottom Top Ten“.
Mit Kibana v4 kann eine Weltkarte mit Hitzekarte einfach integriert werden.
Gute Metriken sind z.B.:

- Erfassung von Quell und Ziel-IPes
- DNS-Anfragen mit Ziel-Auflösung
- Protokolle als Pie (e.g. www)
- Protokolle als Pie (e.g. UDP/TCP)

Erfassung von Metriken kann nach folgendem Schema erfolgen:
Use Case -> Angriffe erkennen -> Spots auswählen, erfassen, visualisieren und Threadsholt setzen.

B.8 Zusammenfassung des Interviews mit Sabine Bernecker-Bendixen

Berufsbezeichnung: Beraterin, sof-IT
Datum: 25.03.2015
Wie kann man Wettbewerb vermeiden?
Nur bei einem extremen Teamgedanken wird Konkurenzkampf entstehen, wobei es auch auf das Umfeld ankommt. Beispielsweise ob eine Leistungsbewertung mit der Team-Sicherheitsprüfung einhergeht. Es sollte keine Visualisierung von unterschiedlichen Reifegraden in Produkten durch eine Führungskraft im Anschluss stattfinden.
Vor den Team-Sicherheits-Prüfungen muss über Sinn und Zweck aufgeklärt werden.
Wie kann man Bloßstellung vermeiden?
Beim Feedback auf Sachebene bleiben mit gemeinsamen Ziel vor Augen.
Stichwort „Fehlerkultur“, in Deutschland geht es oft nach dem Motto „Fehler können andere machen, ich aber nicht“. Gemeinsame Ziel klar machen. Fehler gehören zum Arbeitsalltag dazu. Es kommt stark auf das Unternehmen und Fragestellungen wie „Wie umgehen mit Fehlern?“ „Wie geht das Team und die Führung mit Fehlern von anderen um, wie geht jede einzelne Person mit Fehlern um?“. Dabei geht es auch um Commitment und Vorbild-Funktion. Es sollte auf Sachebene ohne Schuldzuweisung beim Feedback agiert werden. Bei Fehlern kann z.B. nach dem Grund für die Lösung gefragt werden, um die Lösung anschließend zu verbessern.
Regel festhalten: Wie gehen wir mit Feedback um? Ziel: Wir wollen hohe Qualität.
Welche Nachteile sehen Sie bei gemischten Teams?

Es kann interne Rangelleihen geben, beispielsweise wer darf was sagen darf. Es muss sich miteinander arrangiert werden. Ggf. wird sich dann weniger auf eigentliche fachliche Aufgabe konzentriert. Es gibt Personen, die sich nur ungerne an neue Leute gewöhnen. Es geht auch um Vertrauen. *DevOps*: Leute aus unterschiedlichen Bereichen und miteinander, klar kommen. Es muss sich ausgetauscht werden. In gemischten Teams kann Wissen ausgetauscht werden.
Es ist etwas neues, auch auf menschlicher Seite. Hier stellt sich wieder die Frage, wie mit Fehlern umgegangen wird.
John Willis „Wenn Kultur nicht funktioniert, kann man den Rest vergessen.“. Bei CAMS, was für Culture, Automation, Measurement and Sharing steht, wird häufig Culture vergessen.
Personen können es als Herausforderung sehen oder resignieren. Ziele können ggf. nicht erfüllt werden.
Was halten Sie von dem Versuch, den Versuch mit gebundenen und gemischten Teams und unterschiedlichen fachlichen Ansätzen durchzuführen?
Aufgrund der unterschiedliche Methodik kann es zu Abweichungen kommen.
Frau Bernecker-Bendixen empfiehlt, bei gebundenen Teams im Unternehmen den Versuch mit gemischte Teams durchzuführen. Es ist eine andere Vorgehensweise mit besonderem Interesse.
Wie würden Sie die Befragung im Anschluss gestalten?
Es sollten Fragen nach dem Gefühl gestellt werden, also nach Vor- und Nachteilen, dies technisch und persönlich.

B.9 Zusammenfassung des E-Mail-Interviews mit Matthias Rohr

Berufsbezeichnung: Senior Consultant, Secodis GmbH
Datum: 31.03.2016
1) Sind Team-Sicherheitsprüfung sinnvoll?
ja, in der Theorie. Praktisch wird die Einführung in den meisten Fällen kaum funktionieren. Wir haben aktuell Mühe genug überhaupt Sicherheitskompetenz in Agilen Teams aufzubauen. Entsprechende Satelliten (oder Security Champions) sind dann diejenigen, die Sicherheitsthemen in Teams adressieren sollen und auch limitierte Sicherheitstests durchführen können, bzw. hierzu qualifiziert werden. In der Praxis ist es schon schwer genug, diese (im Falle von Feature-Teams handelt es sich dabei stets um Entwickler) überhaupt dazu zu bringen, ihre Aufgabe in ihrem eigenen Team nachzukommen. Andere Teams dann noch zusätzlich zu testen wird relativ schwierig. Zumal hierfür auch noch enorm viel Abstimmung erforderlich ist.
Das ist eine klassische Aufgabe die durch eine zentrale Funktion erfolgen kann und sollte. Wenn lokale Teams sich gegenseitig pentesten oder ähnliches dann ggf. im Rahmen von Challenges und nur wenn dies in die jeweilige Kultur passt.
2) Würdest du bei der Team-Sicherheitsprüfung auf SAST, DAST oder beide Methoden setzen? Dabei gilt zu beachten, dass das Management wohl den Vorteil sieht, jedoch sicherlich Zeit-Einschränken geben wird.
Keines von beiden, dies sind beides Tools die lokal im Team oder über eine zentrale Stelle (Product Owner, also Security oder Entwicklungs-Stabs-Teams) konfiguriert und betrieben werden müssen. Grade in agilen Entwicklungen erhält zudem immer mehr auch IAST (also dynamische Codeanalyse) als Kombination von

DAST und SAST Bedeutung. Security Tests können im letzteren auch durch fachliche Tester angestoßen werden.

3) Siehst du weitere Vor- oder Nachteile bei gebundenen oder gemischten Teams?

Nein, ich sehe das eher als (1)einen klassischen Austausch, der besser über Security Champion Jour Fixes erfolgen sollte oder im Rahmen von bestimmten sicherheitsrelevanten Integration. (2) könnte auch ein Team-Direkter Austausch sinnvoll sein, wenn einzelne Teams sicherheitsrelevante Überschneidungspunkte haben, entweder dynamisch oder regelmäßig. Hängt sehr von der Aufteilung der Teams ab.

4) Es könnte z.B. eine SQL-Injection aufgedeckt werden. Wie würdest du das Bloßstellen von einzelnen Team-Mitgliedern versuchen zu unterbinden? Bei einem Sicherheits-Code-Review ist man ja in einer ähnlichen Situation.

Das hängt wieder sehr stark von der jeweiligen Kultur ab. Manche haben damit kein Problem andere wiederum schon. Generell anonymisiert man solche Findings aber.

5) Welche Anmerkungen hast du zu dem Versuch?

Ich würde schauen, ob du den Scope nicht eher etwas ausweitest und zwar auf generelle Team-Übergreifende Zusammenarbeit in Bezug auf Security. *DevOps* ist hier natürlich sehr Interessant, grundsätzlich sind aber natürlich auch agile Teams ohne *DevOps* hiervon betroffen. Zudem würde ich wie gesagt weg gehen von strikten Unterteilungen in DAST und SAST.

B.10 Zusammenfassung des Interviews mit einer anonymen Person

Berufsbezeichnung: Chief Strategy Officer in einem auf statische Tests spezialisiertem Unternehmen

Datum: 12.05.2016

Tritt eine falsch positive Meldung auf, sollte geprüft werden ob die zugehörige Regel ggf. für zukünftige Tests deaktiviert wird.

Eine Methode um Meldungen bei der Untersuchung auf Schwachstellen zu Priorisieren ist die Aggregation von Meldungen aus unterschiedlichen Werkzeugen. Wird eine Schwachstelle von zwei unabhängigen Werkzeugen erkannt, so sinkt die Wahrscheinlichkeit von einem falsch Positiven.

B.11 Zusammenfassung der Evaluierung beim OWASP Stammtisch Hamburg

Teilnahme: 12 Sicherheitsexperten

Datum: 12.05.2016

Der Unterschied zwischen Integrationstests und Akzeptanztests im Bezug auf Sicherheit war zunächst nicht deutlich. Hier würde ein Quellcode-Beispiel stark helfen.

Der *OWASP Dependency Check* durchsucht auch transitive Abhängigkeiten, diese sind jedoch schwer aufzufinden.

Unklar war wie das retire.js-Docker-Abbild aktuell gehalten wird.

Bei Werkzeugen sollte zusätzlich zur Kritikalität auch das Vertrauen des Werkzeugs in die gefundene

Schwachstelle mit einbezogen werden, insbesondere bei der Einführung. Nachteilig ist, dass Jenkins-Plugins wie der OWASP Dependency Check diese nicht mit einbeziehen und entsprechend selbst entwickelt werden müsste. Die Aufteilung in wichtige und unwichtige Bereiche kann zu erhöhtem Diskussionsbedarf führen.
Die Einführung eines Werkzeugs sollte zwingend von einer Schulung der Mitarbeiter begleitet werden. Bei der Markierung von falsch Positiven kann es vorkommen, dass Entwickler ggf. auf vorher verwendete falsch Positiv-Markierungen für neuen Quellcode zurückgreifen, ohne diese verstanden zu haben. So würde eine potentielle Schwachstelle unentdeckt bleiben. Dagegen können Peer-Reviews duch eine zweite Personen helfen.
Sicherheit kann als Prozess gesehen werden, welcher neben dem normalem Entwicklungsprozess läuft. Beispielsweise kann Thread-Modelling nicht bei jedem Sprint durchgeführt werden. Bei großen Architektur-Änderungen sollte Thread-Modelling durchgeführt werden. Sicherheit fängt auch schon bei der Auswahl der Architektur, beispielsweise des Frameworks an. Dabei kann die Frage gestellt werden, wie anfällig ein Framework für bestimmte Risiken ist.

B.12 Zusammenfassung des E-Mail-Inteview mit Dr. Ralph Holz

Berufsbezeichnung: Lecturer in Networks and Security, School of IT Cybersecurity & Usable Security, Human-Centred Technologies Cluster, University of Sydney
Datum: 07.06.2016
[...]
Bzgl. der externen Repos gibt es zwei Dinge zu beachten. Das eine ist, dass der Fetcher vom Rest der Build-Umgebung isoliert wird. Das wuerde ich genau so machen, wie Sie es auch vorgeschlagen haben.
Leider ist ist das aber vielleicht Verteidigung an der falschen Stelle: wenn wir Abhaengigkeiten zu externen Repos haben, deren Integritaet wir nicht weiter pruefen koennen, weil sie vielleicht selbst einem kompromittierten Build/Commit-Prozess unterliegen, dann kann ein Angreifer auf diese Weise Schadcode in unser System schleusen. Beim npm-Zusammenbruch vor ein paar Wochen hat man ja gesehen, wieviele automatisierte Build-Prozesse tatsaechlich externe Abhaengigkeiten dieser Art haben. Eigentlich furchtbar.
Leider hilft gegen diese Art von Code Injection nur das Verifizieren des importierten Codes - und da wird es fuer schwach finanzierte Projekte natuerlich schwer.
[...]

Anhang C

Studie zur Identifizierung von gemeinsamen Zielen unterschiedlicher IT-Bereiche

C.1 Fragebogen Identifizierung von gemeinsamen Zielen unterschiedlicher Bereiche

Identifizierung von gemeinsamen Zielen der Bereiche Anwendungs-Entwicklung, IT-Betrieb und IT-Sicherheit

Herzlich Willkommen zur Umfrage "Identifizierung von gemeinsamen Zielen der Bereiche Anwendungs-Entwicklung, IT-Betrieb und IT-Sicherheit".

Timo Pagel ist Forscher im Bereich Sicherheit in Webanwendungen und führt diese Befragung durch.

Jede Antwort ist hilfreich und unterstützt bei der Erstellung eines generischen DevOps-Sicherheits-Reifegradmodells.

Sofern Sie über ein Jahr Berufserfahrung im IT-Umfeld haben, geben Sie bitte Ihre Beurteilung ab.

Es gibt keine „richtige“ oder „falsche“ Antwort. Ihre persönliche Meinung zählt!

Diese Umfrage enthält 8 Fragen.

Arbeitsbezogene Angaben

[]Bitte bewerten Sie die Wichtigkeit der folgenden Modelle und Konzepte in Ihrer täglichen Arbeit.

Bitte wählen Sie die zutreffende Antwort für jeden Punkt aus:

	Gar nicht / Sehr gering	Gering	Mittel	Hoch	Sehr hoch	Bedeutung unbekannt
Vorgehen nach Wasserfall-Modell	○	○	○	○	○	○
Agiles Vorgehen	○	○	○	○	○	○
Test Driven Development	○	○	○	○	○	○
Fail Fast	○	○	○	○	○	○
DevOps	○	○	○	○	○	○

[]Bitte bewerten Sie die Wichtigkeit der folgenden Eigenschaften Ihrer in Produktion befindlichen Systeme und Anwendungen.

Bitte wählen Sie die zutreffende Antwort für jeden Punkt aus:

	Gar nicht / Sehr gering	Gering	Mittel	Hoch	Sehr hoch	Bedeutung unbekannt
Qualität	○	○	○	○	○	○
Integrität	○	○	○	○	○	○
Performanz	○	○	○	○	○	○
Stabilität	○	○	○	○	○	○
Vertraulichkeit	○	○	○	○	○	○
Verfügbarkeit	○	○	○	○	○	○
Verantwortlichkeit	○	○	○	○	○	○
Skalierbarkeit	○	○	○	○	○	○

C.2 Ergebnisse der Befragung

Bereich	Ø	Keines / Sehr gering	Gering	Mittel	Hoch	Sehr hoch	Bedeutung unbekannt	Keine Antwort
Software-Entwicklung	3,85 (0,73)	3 (3,23%)	4 (4,30%)	22 (23,66%)	43 (46,24%)	22 (23,66%)	0 (0,00%)	0 (0,00%)
Projekt-Management	3,25 (0,92)	6 (6,45%)	18 (19,35%)	24 (25,81%)	35 (37,63%)	9 (9,68%)	0 (0,00%)	1 (1,08%)
System-Management	3,20 (0,84)	2 (2,15%)	18 (19,35%)	45 (48,39%)	15 (16,13%)	13 (13,98%)	0 (0,00%)	0 (0,00%)
IT-Sicherheit	3,05 (0,78)	3 (3,23%)	23 (24,73%)	40 (43,01%)	20 (21,51%)	7 (7,53%)	0 (0,00%)	0 (0,00%)
Netzwerk-Management	2,58 (0,75)	8 (8,60%)	38 (40,86%)	32 (34,41%)	13 (13,98%)	1 (1,08%)	0 (0,00%)	1 (1,08%)

Tabelle C.1: Ergebnisse der Befragung nach Wissen in unterschiedlichen IT-Bereichen

Modelle und Konzepte	Ø (σ)	Gar nicht / Sehr gering	Gering	Mittel	Hoch	Sehr hoch	Bedeutung unbekannt	Keine Antwort
Agiles Vorgehen	4,27 (0,56)	0 (0,00%)	4 (4,30%)	7 (7,53%)	41 (44,09%)	40 (43,01%)	1 (1,08%)	0 (0,0%)
DevOps	3,48 (1,11)	6 (6,45%)	12 (12,90%)	27 (29,03%)	21 (22,58%)	23 (24,73%)	2 (2,15%)	2 (2,15%)
Test Driven Development	3,28 (0,98)	6 (6,45%)	16 (17,20%)	28 (30,11%)	27 (29,03%)	13 (13,98%)	3 (3,23%)	0 (0,0%)
Fail Fast	3,07 (1,48)	8 (8,60%)	17 (18,28%)	18 (19,35%)	22 (23,66%)	8 (8,60%)	19 (20,43%)	1 (1,08%)
Vorgehen nach Wasserfall-Modell	2,19 (1,03)	30 (32,26%)	29 (31,18%)	18 (19,35%)	13 (13,98%)	1 (1,08%)	1 (1,08%)	1 (1,08%)

Tabelle C.2: Ergebnisse der Befragung nach der Wichtigkeit von Modellen und Konzepten

Qualitäts-kriterium	**Ø**	**Gar nicht / Sehr gering**	**Gering**	**Mittel**	**Hoch**	**Sehr hoch**	**Bedeu-tung unbe-kannt**	**Keine Ant-wort**
Verfügbar-keit	4,36 (0,46	0 (0,00%)	1 (1,08%)	9 (9,68%)	38 (40,86%)	44 (47,31%)	0 (0,00%)	1 (1,08%)
Stabilität	4,33 (0,49	0 (0,0%)	0 (0,00%)	10 (10,75%)	41 (44,09%)	40 (43,01%)	0 (0,00%)	2 (2,15%)
Qualität	4,24 (0,41	0 (0,0%)	0 (0,00%)	12 (12,90%)	45 (48,39%)	34 (36,56%)	2 (2,15%)	0 (0,00%)
Integrität	4,07 (0,82	0 (0,0%)	4 (4,30%)	15 (16,13%)	37 (39,78%)	29 (31,18%)	7 (7,53%)	1 (1,08%)
Performance	3,86 (0,76	0 (0,0%)	4 (4,30%)	29 (31,18%)	34 (36,56%)	24 (25,81%)	0 (0,00%)	2 (2,15%)
Vertrau-lichkeit	3,78 (0,94	1 (1,08%)	13 (13,98%)	20 (21,51%)	28 (30,11%)	29 (31,18%)	2 (2,15%)	0 (0,00%)
Verantwort-lichkeit	3,65 (1,08	0 (0,0%)	6 (6,45%)	30 (32,26%)	27 (29,03%)	15 (16,13%)	13 (13,98%)	2 (2,15%)
Skalierbar-keit	3,40 (1,00	3 (3,23%)	18 (19,35%)	28 (30,11%)	25 (26,88%)	18 (19,35%)	1 (1,08%)	0 (0,00%)

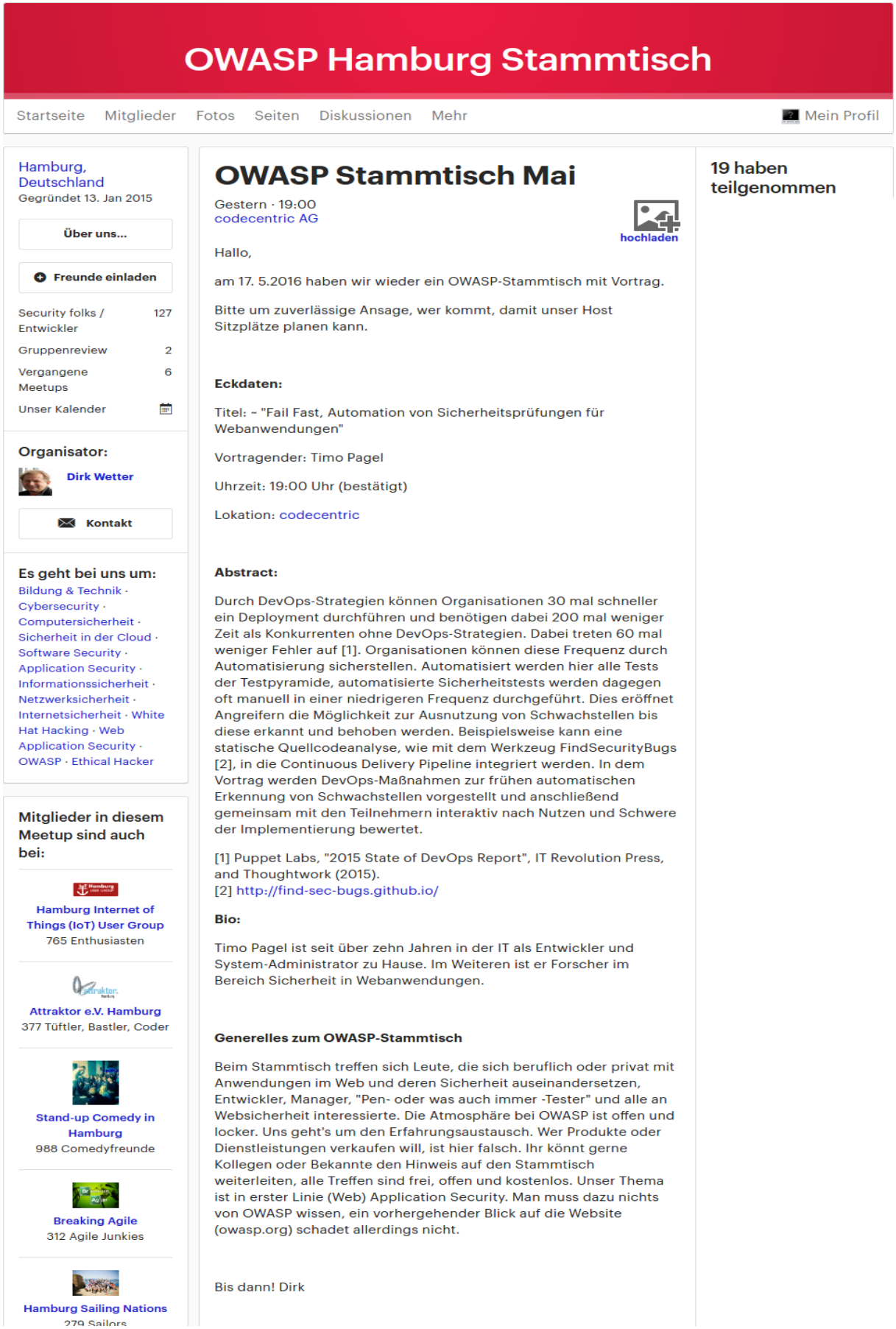
Tabelle C.3: Ergebnisse der Befragung nach der Wichtigkeit von Qualitätskriterien

Anhang D

Einladungen für Evaluierungen

D.1 OWASP Stammtisch Hamburg

D.2 DevOps HH Meetup



OWASP Hamburg Stammtisch

Startseite Mitglieder Fotos Seiten Diskussionen Mehr Mein Profil

Hamburg, Deutschland
Gegründet 13. Jan 2015

Über uns...

Freunde einladen

Security folks / Entwickler 127
Gruppenreview 2
Vergangene Meetups 6
Unser Kalender

Organisator:

Dirk Wetter

Kontakt

Es geht bei uns um:
Bildung & Technik · Cybersecurity · Computersicherheit · Sicherheit in der Cloud · Software Security · Application Security · Informationssicherheit · Netzwerksicherheit · Internetsicherheit · White Hat Hacking · Web Application Security · OWASP · Ethical Hacker

Mitglieder in diesem Meetup sind auch bei:

Hamburg Internet of Things (IoT) User Group
765 Enthusiasten

Attraktor e.V. Hamburg
377 Tüftler, Bastler, Coder

Stand-up Comedy in Hamburg
988 Comedyfreunde

Breaking Agile
312 Agile Junkies

Hamburg Sailing Nations
279 Sailors

OWASP Stammtisch Mai

Gestern · 19:00
codecentric AG

hochladen

Hallo,

am 17. 5.2016 haben wir wieder ein OWASP-Stammtisch mit Vortrag.

Bitte um zuverlässige Ansage, wer kommt, damit unser Host Sitzplätze planen kann.

Eckdaten:

Titel: ~ "Fail Fast, Automation von Sicherheitsprüfungen für Webanwendungen"

Vortragender: Timo Pagel

Uhrzeit: 19:00 Uhr (bestätigt)

Lokation: codecentric

Abstract:

Durch DevOps-Strategien können Organisationen 30 mal schneller ein Deployment durchführen und benötigen dabei 200 mal weniger Zeit als Konkurrenten ohne DevOps-Strategien. Dabei treten 60 mal weniger Fehler auf [1]. Organisationen können diese Frequenz durch Automatisierung sicherstellen. Automatisiert werden hier alle Tests der Testpyramide, automatisierte Sicherheitstests werden dagegen oft manuell in einer niedrigeren Frequenz durchgeführt. Dies eröffnet Angreifern die Möglichkeit zur Ausnutzung von Schwachstellen bis diese erkannt und behoben werden. Beispielsweise kann eine statische Quellcodeanalyse, wie mit dem Werkzeug FindSecurityBugs [2], in die Continuous Delivery Pipeline integriert werden. In dem Vortrag werden DevOps-Maßnahmen zur frühen automatischen Erkennung von Schwachstellen vorgestellt und anschließend gemeinsam mit den Teilnehmern interaktiv nach Nutzen und Schwere der Implementierung bewertet.

[1] Puppet Labs, "2015 State of DevOps Report", IT Revolution Press, and Thoughtwork (2015).
[2] http://find-sec-bugs.github.io/

Bio:

Timo Pagel ist seit über zehn Jahren in der IT als Entwickler und System-Administrator zu Hause. Im Weiteren ist er Forscher im Bereich Sicherheit in Webanwendungen.

Generelles zum OWASP-Stammtisch

Beim Stammtisch treffen sich Leute, die sich beruflich oder privat mit Anwendungen im Web und deren Sicherheit auseinandersetzen, Entwickler, Manager, "Pen- oder was auch immer -Tester" und alle an Websicherheit interessierte. Die Atmosphäre bei OWASP ist offen und locker. Uns geht's um den Erfahrungsaustausch. Wer Produkte oder Dienstleistungen verkaufen will, ist hier falsch. Ihr könnt gerne Kollegen oder Bekannte den Hinweis auf den Stammtisch weiterleiten, alle Treffen sind frei, offen und kostenlos. Unser Thema ist in erster Linie (Web) Application Security. Man muss dazu nichts von OWASP wissen, ein vorhergehender Blick auf die Website (owasp.org) schadet allerdings nicht.

Bis dann! Dirk

19 haben teilgenommen

Abbildung D.1: Einladung für den OWASP Stammtisch Hamburg am 17.05.2016
Quelle: http://www.meetup.com/de-DE/OWASP-Hamburg-Stammtisch/events/230963508/, abgerufen am 14.07.2016

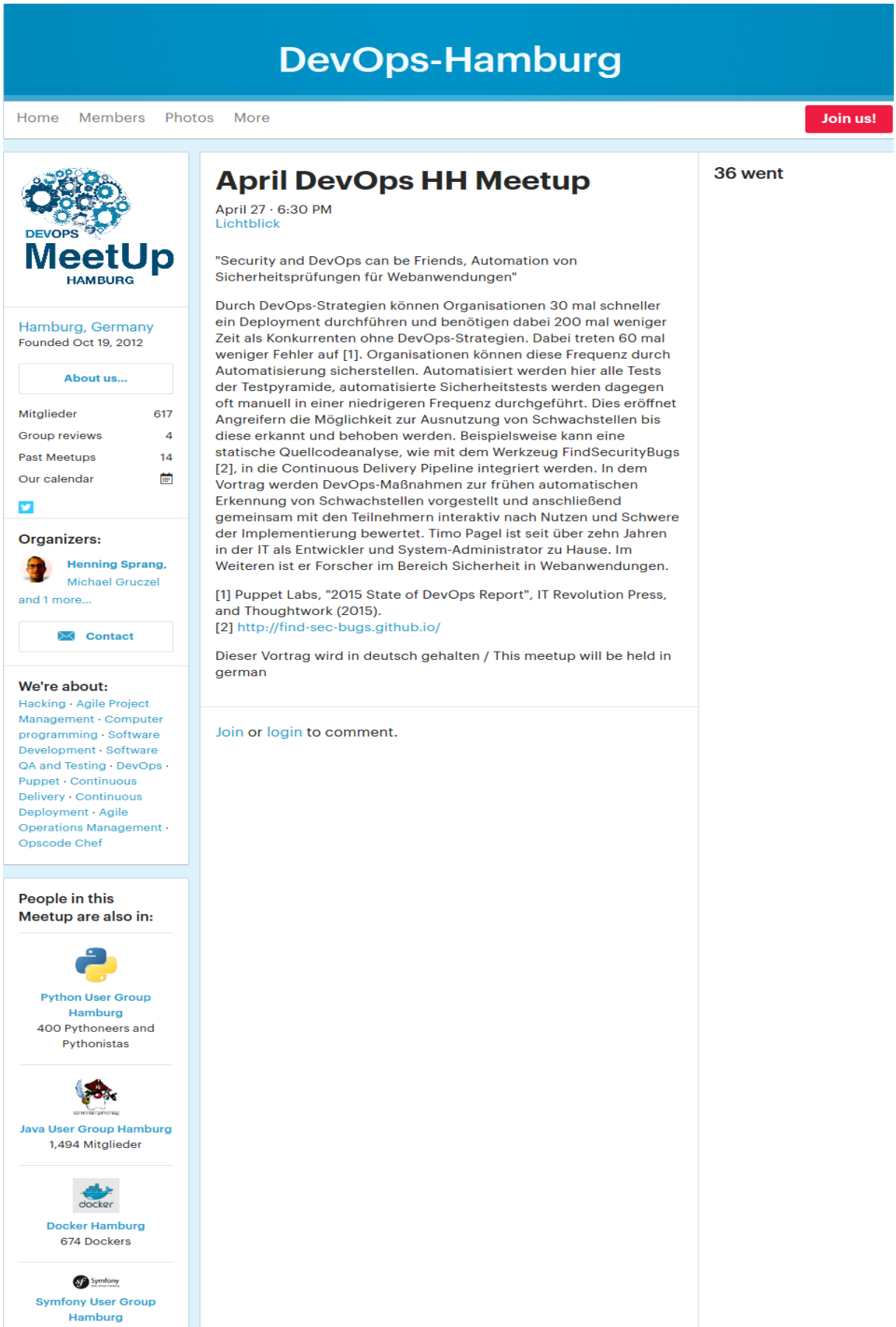


Abbildung D.2: Einladung für das *DevOps* HH Meetup
Quelle: http://www.meetup.com/de-DE/DevOps-Hamburg/events/230040965/, abgerufen am 14.07.2016

Anhang E

Metriken

E.1 Erfassung von Metriken nach Kategorie, Einheit, Zweck und Quelle

Kategorie	Metrik (Einheit)	Zweck	Quellen
Anti-Virus und Anti-Rootkit	Anzahl der aufgedeckten Viren / Rootkits nach Server	Indikator für die Infizierungsrate auf Servern	- Anti-Virus und Anti-Rootkit Software
	Vorfälle mit manueller Säuberung als Anzahl / Prozentwert	Zeigt den relativen erfoderlichen Aufwand zum Säubern	- Anti-Virus und Anti-Rootkit Software - Ticket-System
Firewall und Netzwerk	Anzahl der Regeländerungen einer Firewall - Nach Geschäftseinheiten; Nach Typ	Schlägt das Niveau der Anforderung an Sicherheits-Komplexität vor	- Firewall-Management-Systene - Zeiterfassungs-Systeme - Projekt-Mangement-Systeme
	Anzahl der eingehender Verbindungen - Nach TCP/UDP-Port; Nach Servertyp oder Servergruppe; Nach Quell-Standort	Absoultes Niveau von eingehenden Aktivitäten	- Firewall-Management-System - Server-Schnittstellen
	Anzahl der ausgehender Verbindungen - Nach TCP/UDP-Port; Nach Servertyp oder Servergruppe; Nach Quell-Standort	Absoultes Niveau von ausgehenden Aktivitäten	- Firewall-Management-System - Server-Schnittstellen
	Rate der Netzwerk Dienste - Alle Ports; Unnötigen Ports; Nach System-Typ	Identifiziert Netzwerk-Einstiegspunkte an Systemen	- Netzwerk-/Port-Scan Software
Angriffe	Anzahl von Angriffen	Absolute Anzahl der erkannten Angriffe	- IDS - Manuelle Datenquellen
	Rate von Web-Sessions zu Angreifern auf drei Ebenen: - Gemeldete (initiale IDS-Ereignisse); Verdächtigte (gefilterte Alarme / eskaliserte Alarme); Angeifer (manuell bestätigte)	Zeigt die auf kleinster Ebene erzeugten Sicherheits-Ereignisse verglichen mit den tatsächlichen Angriffen	- IDS - Manuelle Datenquellen

Tabelle E.1: Auswahl von Verteidigungs-Metriken in Anlehnung an [140, S. 48ff.]

Kategorie	Metrik (Einheit)	Zweck	Quellen
Anti-Virus und Anti-Rootkit	Server mit Anti-Viren Software als Anzahl / Prozentwert	Umfang der Anti-Viren Kontrollen	- Anti-Virus Software
	Server mit Anti-Rootkit Software als Anzahl / Prozentwert	Umfang der Anti-Rootkit Kontrollen	- Anti-Rootkit Software
	Server mit akutellen Anti-Viren-Signaturen als Anzahl / Prozentwert	Konformität zu Service Level Agreements oder Richtlinien	- Anti-Virus Software
	Server mit aktueller Anti-Rootkit Software als Anzahl / Prozentwert	Konformität zu Service Level Agreements oder Richtlinien	- Anti-Rootkit Software
Patch Management	Systeme unter Richtlinien-Patch-Vorgabe - Nach Server - Nach kritischen Systemen - Nach Betriebssystem - Nach Geschäftseinheit - Nach geografischen Standort	Identifizierung von Lücken im Patch Management-Prozess	- Patch Management Software - Schwachstellen-Management Systeme - System Management Software
	Anzahl wartender Patches - Nach Kritikalität - Nach Geschäftseinheit - Nach geografischen Standort	Identifiziert aktuelle Arbeitslast für anzuwendene Patches	- Patch Management Software
	Latenz wartender Patches	Zeigt das Angriffsfenster bei fehlenden Patches	- Patch Management Software
System Konfiguration	Prozentuale Anzahl der Systeme mit Konformität zu Organisations-Standards	Zeigt die Konformität von System-Konfigurationen zu Standards	- Server Management Software
	Systeme konfiguriert durch zentrale Provisionierungs-Software	Systeme, welche entfernt administriert werden können	- Server Management Software
Schwach-stellen Management	Frequenz der Durchführung von Schwachstellen-Scans	Zeigt wie häufig Schwachstellen-Scans durchgeführt werden	- Schwachstellen Management Software
	Schwachstellen pro System - Nach Kritikalität; - Nach Typ - Nach Gut (Englisch Asset)	Deutet die relative Ebene des potentials von unsicherheit basierend auf der Anahl der Schwachstellen pro System an	- Schwachstellen Management Software
	Monatliche Anzahl an gefundenen Schwachstellen - Nach Kritikalität - Nach Geschäftseinheit - Nach Geografie - Nach System-Typ	Rohe Anzahl an Schwachstellen über Zeit erzeugen ein Gesamtbild der Schwachstellen-Arbeitslast	- Schwachstellen Management Software

Tabelle E.2: Abdeckungs- und Kontroll-Metriken in Anlehnung an [140, S. 54ff.]

Kategorie	Metrik (Einheit)	Zweck	Quellen
Betriebszeit	System-Betriebszeit als prozentualer Wert / Stunde - Für kritische Systeme - Für alle Systeme	Verfügbarkeit Messung für kritische Systeme und andere	- Überwachungs Software - Tabellenkalkulation - Manuelle Erfassung
	Ungeplante System-Ausfälle als prozentualer Wert	Zeigt die Abweichung vom Soll-Zustand. Eine hohe Anzahl deutet auf eine wenig kontrolliertes Umgebung hin.	- Überwachungs Software mit manueller Angabe von Wartungs-Zeiträumen
Veränder-ungs-management	Anzahl an Änderungen pro Tag	Misst die Menge an Änderungen für eine Produktionsumgebung	- Veränderungsman-gement Software

Tabelle E.3: Auswahl von Verfügbarkeits- und Stabilitäts-Metriken in Anlehnung an [140, S. 68f.]

E.2 Visualisierung der Verbindungs-Quell-Standorte als Weltkarte mit Hitzekarte

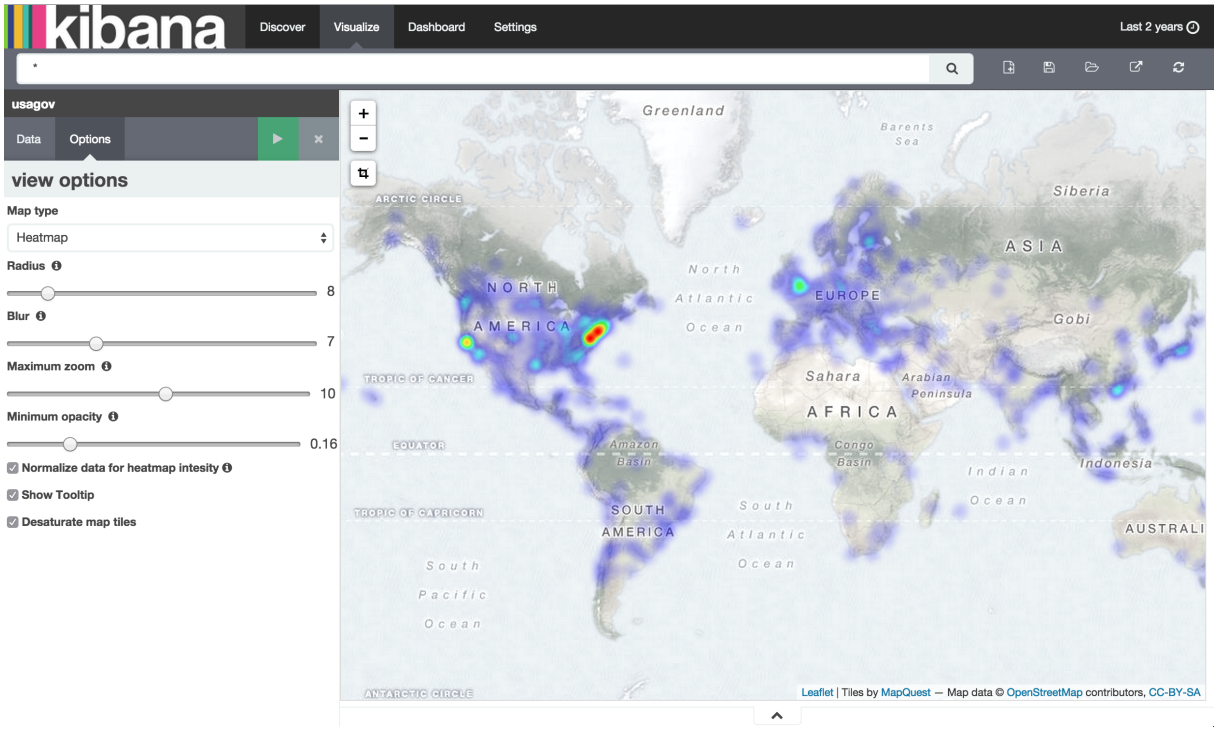


Abbildung E.1: Visualisierung der Verbindungs-Quell-Standorte als Weltkarte mit Hitzekarte
Quelle: https://www.elastic.co/blog/kurrently-kibana-2015-05-01, Abgerufen: 24.03.2016

Anhang F

Beispiele für die Zuordnung des Quellcodes von Kontrollgruppen

- Arbeitsspeicher: mm/memcontrol.c
- Prozessor: kernel/cpuset.c
- Netzwerk: net/core/netprio_cgroup.c
- Geräte: security/device_cgroup.c

Anhang G

Team-Sicherheitsprüfungen

G.1 Dokumentation der Besprechung von Team-Sicherheitsprüfungen

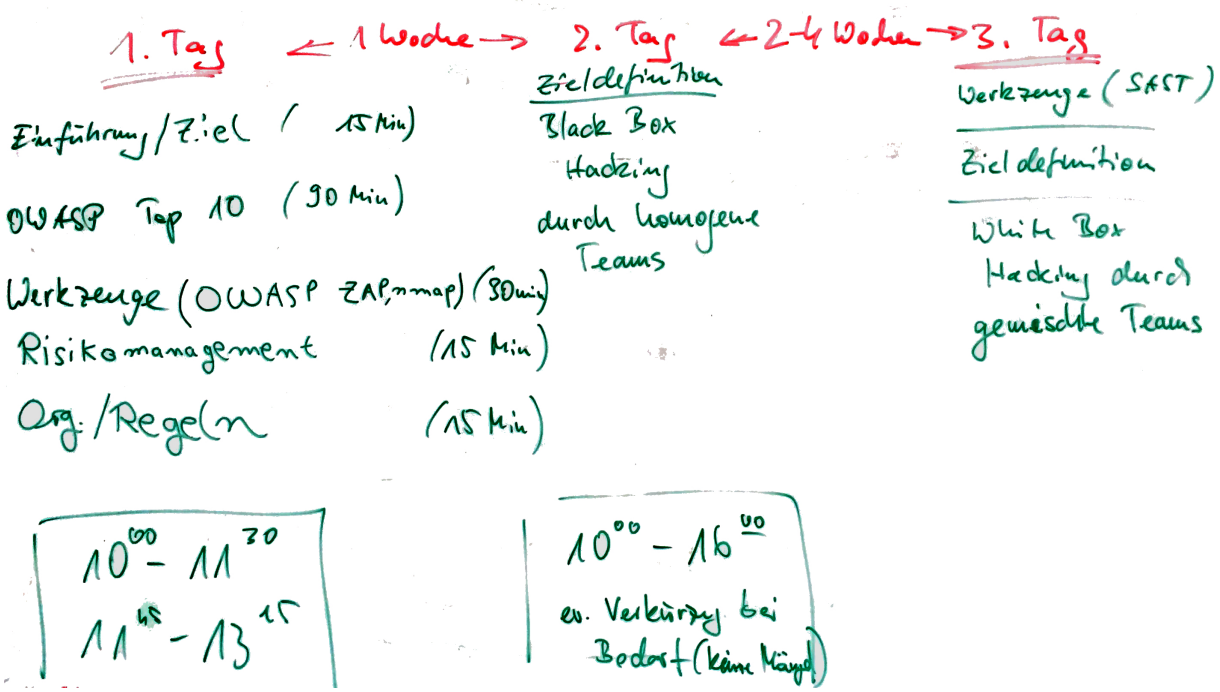


Abbildung G.1: Vorgehen der Team-Sicherheitsprüfungen 1/2

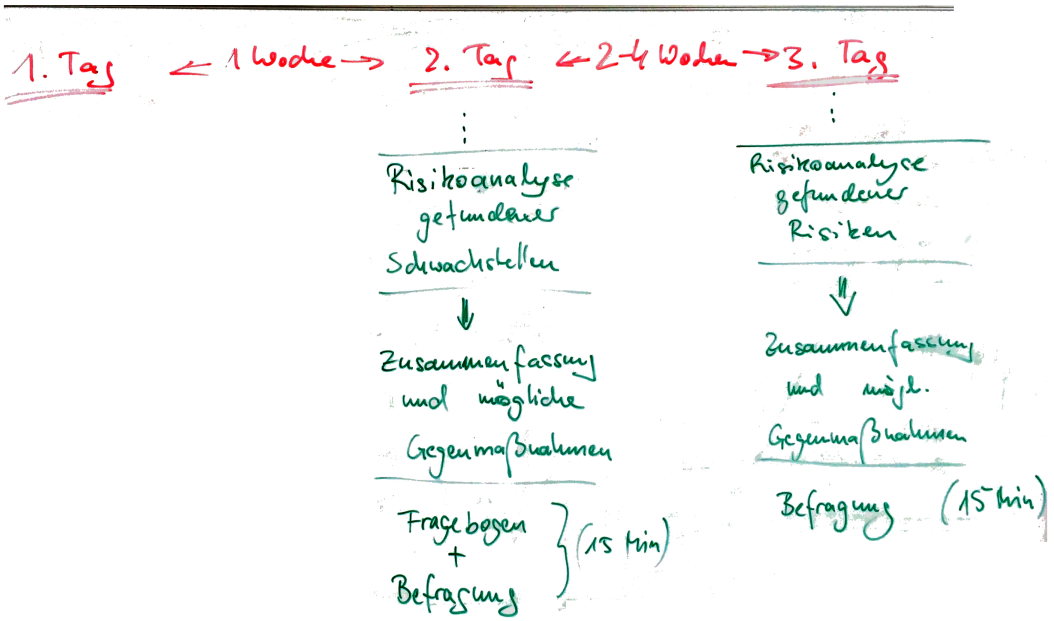


Abbildung G.2: Vorgehen der Team-Sicherheitsprüfungen 2/2

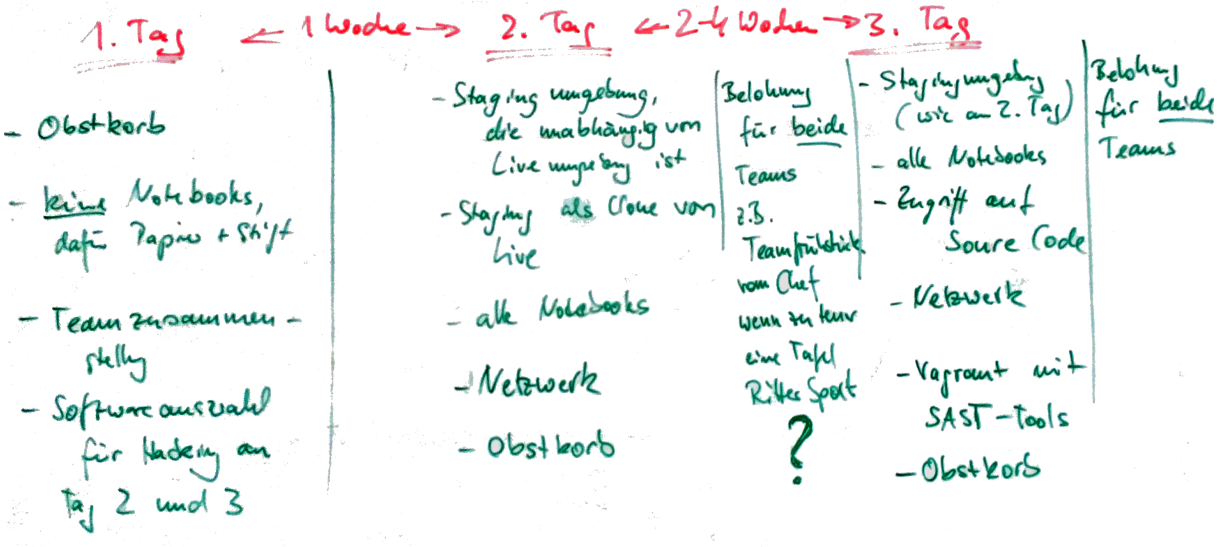


Abbildung G.3: Vorbereitung der Team-Sicherheitsprüfungen

G.2 Flyer für Team-Sicherheitsprüfungen

Timo Pagel

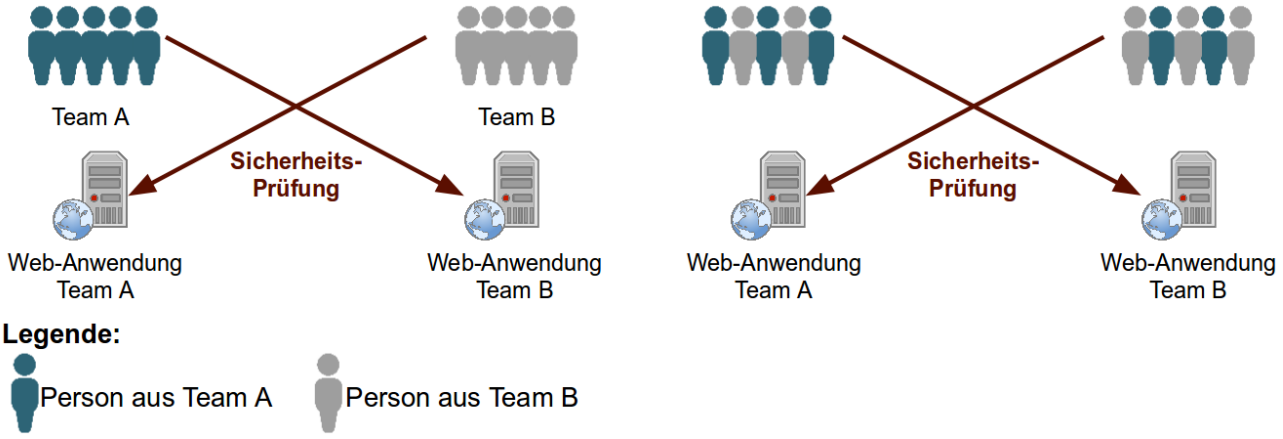
Team-basierte Sicherheitschecks

Teilnehmer eignen sich Wissen und Bewusstsein im Bereich Sicherheit an, um nachhaltig die Sicherheit von Webanwendungen zu erhöhen.

Web-Security Training (18.04.2016, 11:00 bis 14:30 Uhr)

In dem Training wird auf die OWASP Top Ten, coole Werkzeuge, eine einfache Risikobewertung sowie organisatorische Maßnahmen und Regeln eingegangen. Bitte bringt Stift und Papier mit.

Hacking

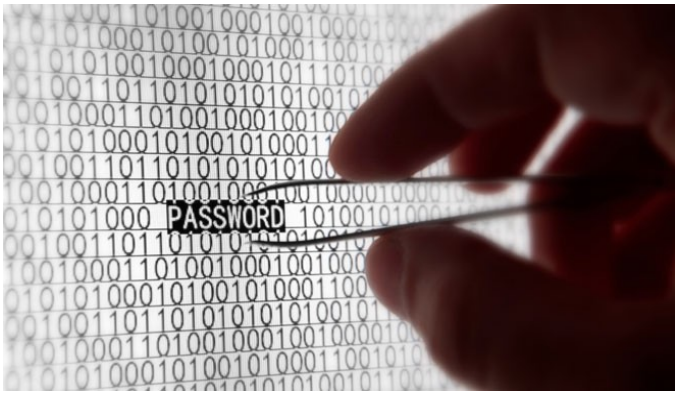


Blackbox Hacking (25.04.2016, 10:30 bis 16:30 Uhr)

Beim Blackbox Hacking testet Team A die gestartete Webanwendung von Team B und Team B die gestartete Webanwendung von Team A (links in der Abbildung). Bitte bringt einen Laptop mit, um dynamische Security-Werkzeuge nutzen zu können.

Whitebox Hacking (23.05.2016, 10:30 bis 16:30 Uhr)

Beim Whitebox Hacking werden die Teams gemischt (rechts in der Abbildung) und erhalten vollen Zugriff auf die zu testende Webanwendung (Source Code). Dynamische und statische Security-Werkzeuge werden zum Einsatz kommen. *Insider* bringen dabei also Anwendungsinterna mit ein und unterstützen bei manuellen Inspektionen. Bitte bringt einen Laptop mit.



Bildquelle: https://securitybreakers.wordpress.com/2014/08/24/classification-of-hackers/

G.3 Planung der Schulung

Es wird das empfohlene Lehrkonzept aus Zeitmangel im Rahmen dieser Thesis nicht angewendet, sondern eine stark verkürzte Schulung[1] ohne praxisnahe Elemente gehalten. Das Schulungsprogramm ist in Tabelle G.1 abgebildet.

Inhalt	Zeit
Einführung und Ziele	15 Minuten
OWASP Top Ten Teil 1	75 Minuten
Pause	15 Minuten
OWASP Top Ten Teil 2	15 Minuten
Werkzeuge und Literatur	30 Minuten
Einfache Risikobewertung	15 Minuten
Organisation und Regeln	15 Minuten

Tabelle G.1: Schulungsprogramm

Die Schulung beginnt mit einer Vorstellung des Vortragenden, gefolgt von der Vorstellung des generellen Zieles der Team-Sicherheitsprüfungen sowie einer kurzen Übersicht über die Inhalte der TSP. Anschließend wird Sicherheit als Qualitätsmerkmal betont sowie die Bedeutung von Sicherheit anhand von in der Vergangenheit ausgenutzten Schwachstellen und deren Auswirkungen aufgezeigt.
Die OWASP Top Ten[2] können im Rahmen von 90 Minuten nicht vollständig vorgestellt werden. Entsprechend wird nur auf die Themen „Injection“, „Cross-Site Scripting“, „unsichere direkte Objektreferenzen“, „sicherheitsrelevante Fehlkonfiguration“, „Cross-Site Request Forgery“ (CSRF) und „Nutzung von Komponenten mit bekannten Schwachstellen“ eingegangen. Im Anschluss erfolgt eine Ergebnissicherung durch einen Arbeitsauftrag via Zurufabfrage, unterstützt durch das Medium Whiteboard. Dabei wird das in Abb. G.4 auf der nächsten Seite dargestellte Diagramm iterativ von Aktion 1 bis 6 an der Tafel skizziert. Die Teilnehmer erläutern, welche Sicherheitsmaßnahmen jeweils getroffen werden müssen, um Risiken zu verhindern.
Anschließend erfolgt die Vorstellung von Blackbox-Testing mit dem Web-Schwachstellenscanner *OWASP Zed Attack Proxy [188]*, da dies eine Grundlage für die Ausführung von Sicherheits-Tests bildet. Weitere spezialisierte Werkzeuge, wie *sqlmap* [226] zum Testen auf SQL Injection und *testssl* [247] zum Testen der TLS/SSL-Konfiguration des Produktions-Servers, werden vorgeführt. Danach wird ein Verfahren zur einfachen Bewertung von Risiken vorgestellt. Anschließend wird auf den OWASP Testing Guide v4 und das Durchführungskonzept für Penetrations-Tests vom BSI hingewiesen. Darauf folgend sollen die Team-Mitglieder die Wahrscheinlichkeit der Ausnutzung sowie den entstanden Schaden von zwei beispielhaften Risiken schätzen und in eine Risiko-Matrix eintragen. Als erstes ist eine SQL-Injection zu analysieren, welche eine mittel bis hohe Eintrittswahrscheinlichkeit und eine sehr hohe Auswirkung hat [248]. Anschließend ist das Fehlen eines Zugangsschutzes für eine Gutschein-Liste zu bewerten. Die Gutscheine sind gültig und ausschließlich in Zeitschriften veröffentlicht. Die Schwachstelle hat eine geringe Eintrittswahrscheinlichkeit und eine geringe Auswirkung. Zum Abschluss erläutert der Dozent Regeln für die TSP. Bei TSPen steht das gemeinsame Ziel der Erhöhung der Sicherheit der Webanwendungen und das

[1] Die Folien zur Schulung unter https://docs.google.com/presentation/d/15fjLFKysgOl-SSkXvqeThpz3nM4pxh4a4aOeQ6VjYjY/edit?usp=sharing abgelegt.
[2] Die OWASP Top Ten listet die höchsten Risiken für Webanwendungen, basierend auf dem Schadenpotential für ein Unternehmen und der Eintrittswahrscheinlichkeit der Ausnutzung der Schwachstelle [246].

gemeinsame Lernen im Vordergrund. Entsprechend ist es unwichtig, wer eine Schwachstelle implementiert hat oder ein Risiko nicht gesehen hat.

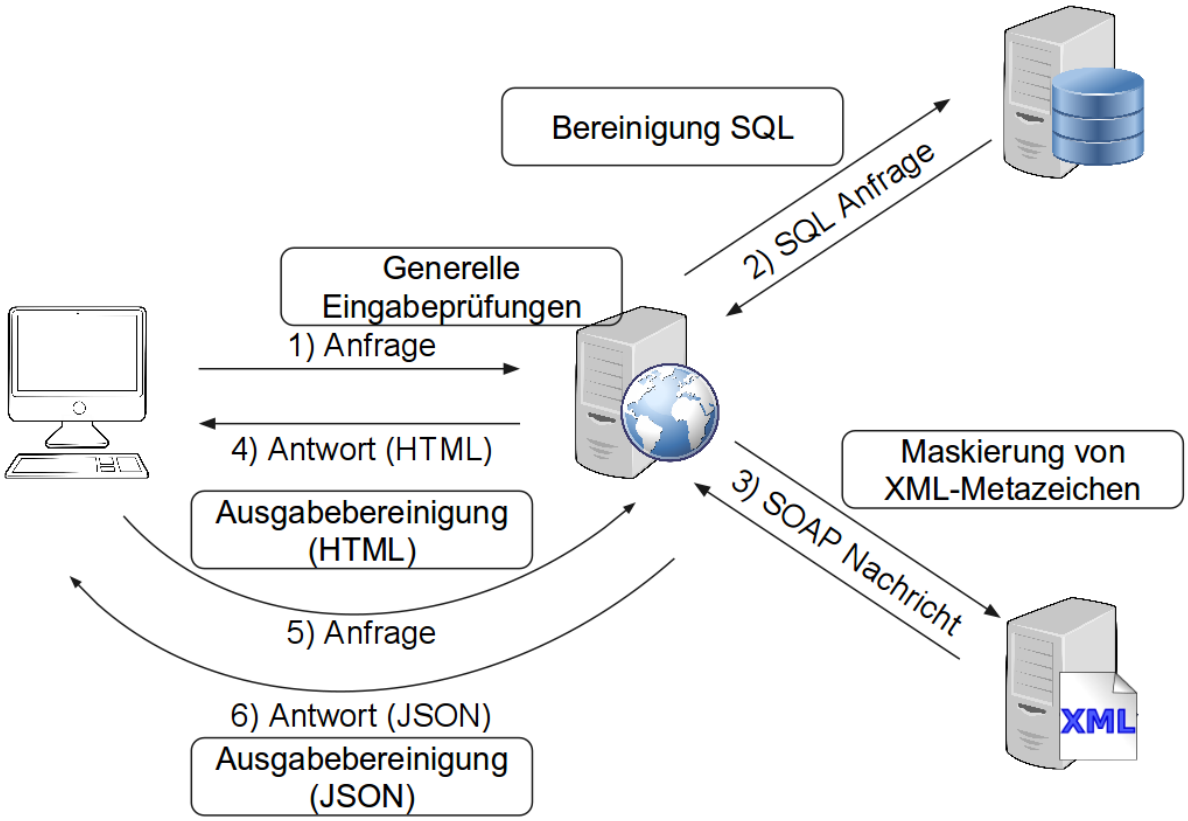


Abbildung G.4: Diagramm zur Ergebnissicherung in der Schulung in Anlehnung an [230, S. 27]

G.4 Vor- und Nachteile einer Belohnung unter Berücksichtigung des Zeitpunkt eines gemeinsamen Essens

Es könnte eine Belohnung (für beide Teams) in Form eines Team-Mittagessens während der Prüfung geben. Dies kann dazu führen, dass die Team-Sicherheitsprüfung bei einigen Personen in guter Erinnerung bleibt. Das Mittagessen kann dabei als „Trigger“ fungieren, so dass bei Erinnerung an das positive Gefühl beim Mittagessen inhaltliche Informationen abgerufen werden können. Bei einigen Personen kann es auch vorkommen, dass das Mittagessen als positive Haupterinnerung an die Team-Sicherheitsprüfung bleibt auch wenn der inhaltliche Bezug fehlt. Dadurch die Vergegenwärtigung eines zuvor gelernten Sachverhalts Informationen vom Arbeitsgedächtnis in das deklarative Langzeitgedächtnis überführt werden [155, S. 28], kann eine Belohnung, z.B. ein Team-Frühstück, mit ein bis zwei Wochen Versetzung sinnvoll sein. Organisatorische Änderungen hängen stark von der Verpflichtung der Führungsspitze ab [122, S. 623], entsprechend sollte die Belohnung durch die Führungsspitze angekündigt werden.

Wissen	**Software-Entwicklung**		**System-Management**		**Netzwerk-Management**		**IT-Sicherheit**		**Projekt-Management**	
	Anzahl	%	Anzahl	%	Anzahl	%	Anzahl	%	Anzahl	%
Keines	0	0,00	0	0,00	0	0,00	0	0,00	0	0,00
Gering	0	0,00	0	0,00	2	33,33	1	16,67	2	33,33
Mittel	1	16,67	5	83,33	2	33,33	4	66,67	1	16,67
Hoch	4	66,67	0	0,00	0	0,00	0	0,00	1	16,67
Sehr hoch	0	0	0	0,00	0	0,00	0	0,00	0	0,00
Nicht beendet	1	16,67	1	16,67	1	16,67	1	16,67	1	16,67
Keine Antwort	0	0,00	0	0,00	1	16,67	0	0,00	1	16,67

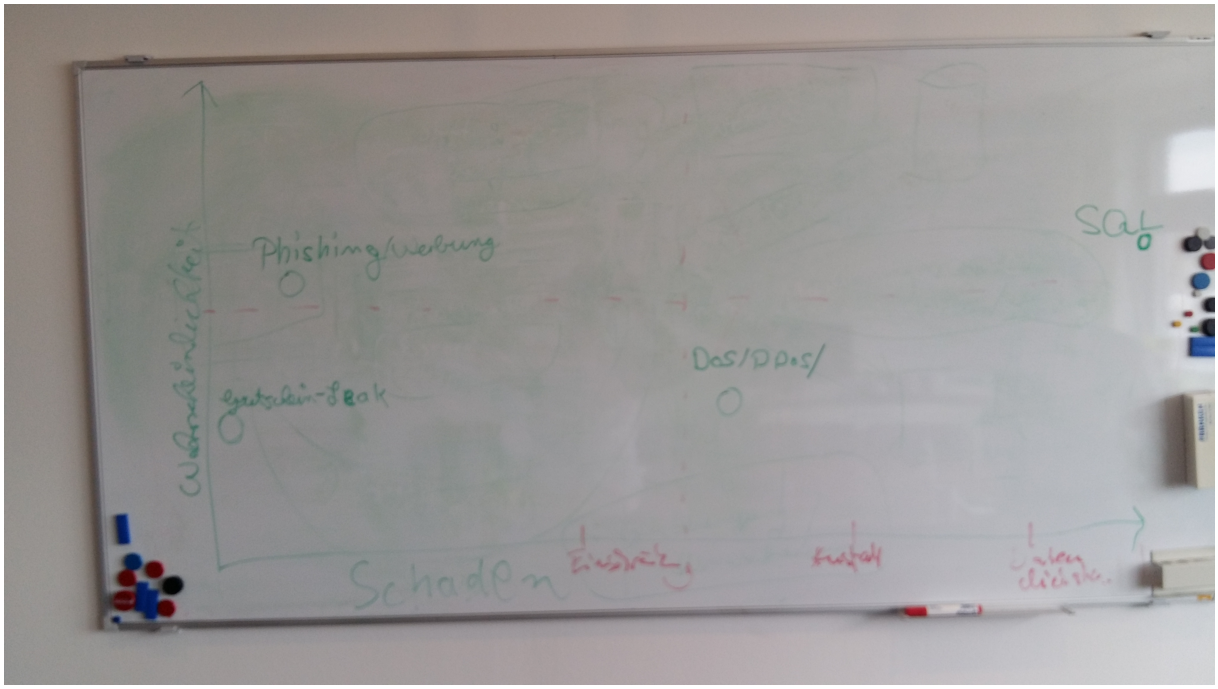
Tabelle G.2: Wissensverteilung der Teilnehmer der homogenen Team-Sicherheitsprüfung

Wissen	**Software-Entwicklung**		**System-Management**		**Netzwerk-Management**		**IT-Sicherheit**		**Projekt-Management**	
	Anzahl	%	Anzahl	%	Anzahl	%	Anzahl	%	Anzahl	%
Keines	0	0,00	0	0,00	0	0,00	0	0,00	1	16,67
Gering	0	0,00	0	0,00	2	33,33	1	16,67	1	16,67
Mittel	0	0,00	4	66,67	3	50,00	4	66,67	2	33,33
Hoch	4	66,67	2	33,33	1	16,67	1	16,67	2	33,33
Sehr hoch	2	33,33	0	0,00	0	0,00	0	0,00	0	0,00

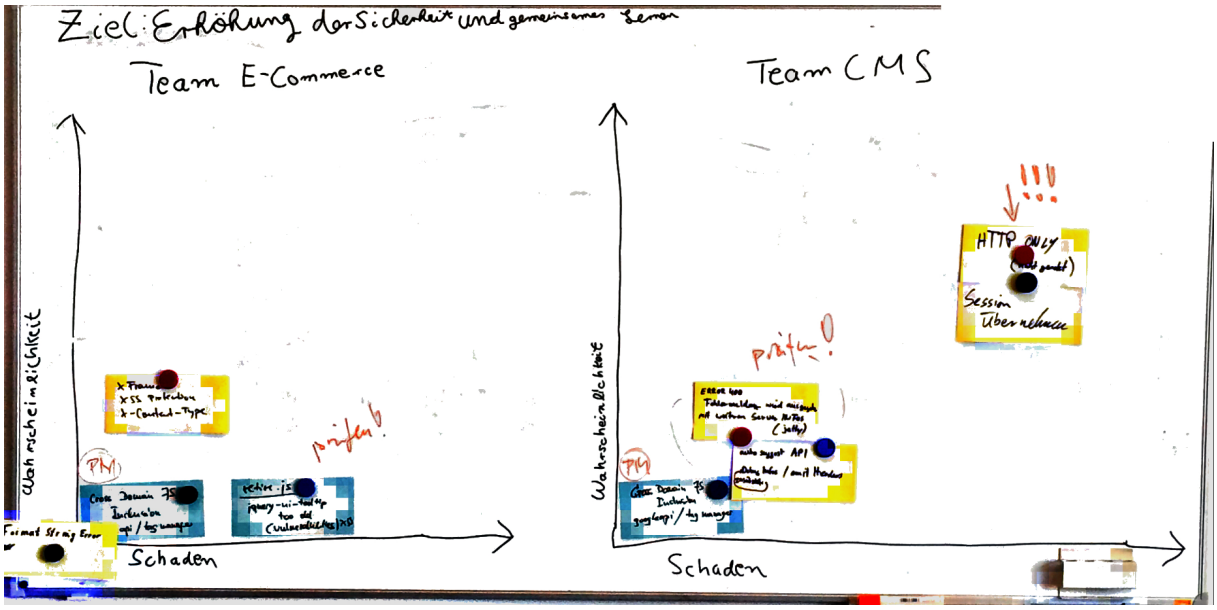
Tabelle G.3: Wissensverteilung der Teilnehmer der inhomogenen Team-Sicherheitsprüfung

G.5 Wissensverteilung der Teilnehmer

G.6 Von Schulungs-Teilnehmern erstellte Risikomatrix

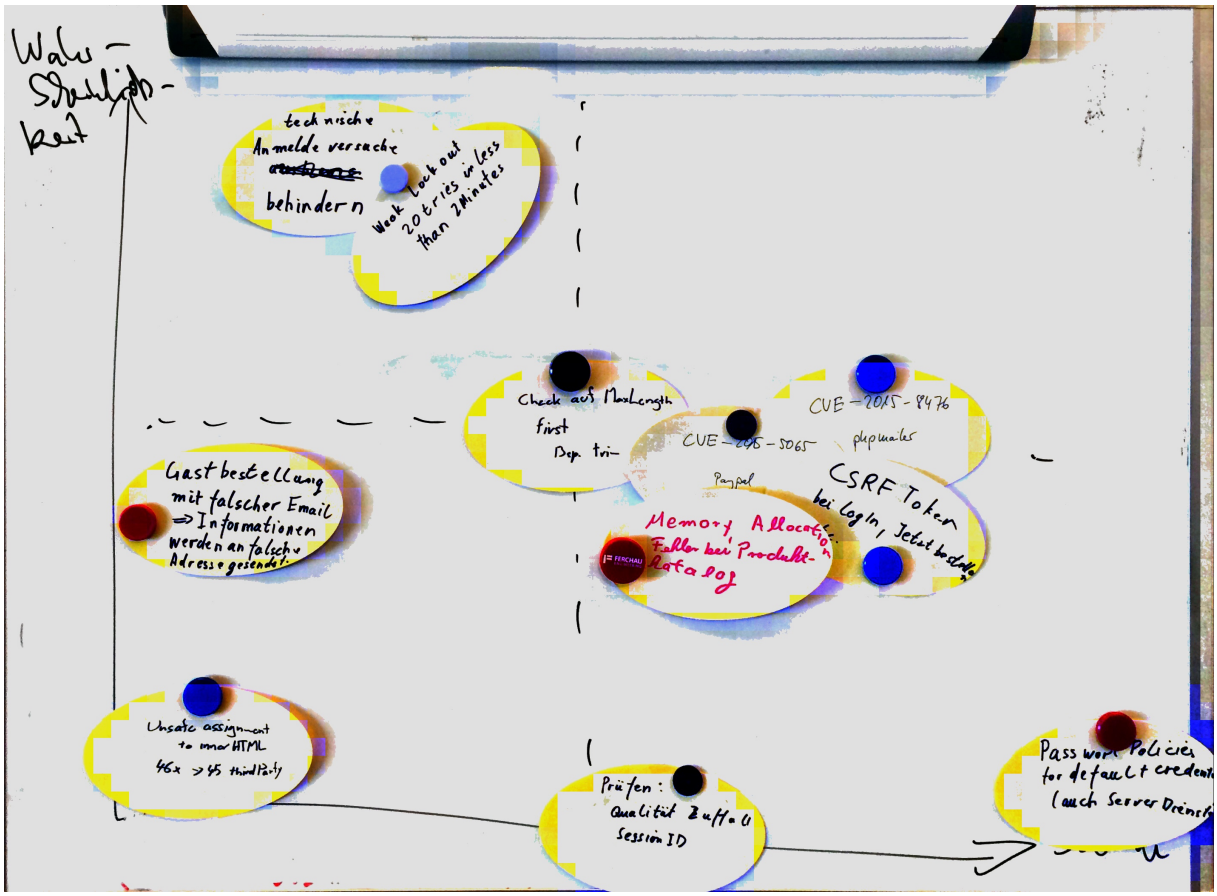


G.7 Ergebnisse der homogenen Team-Sicherheitsprüfungen

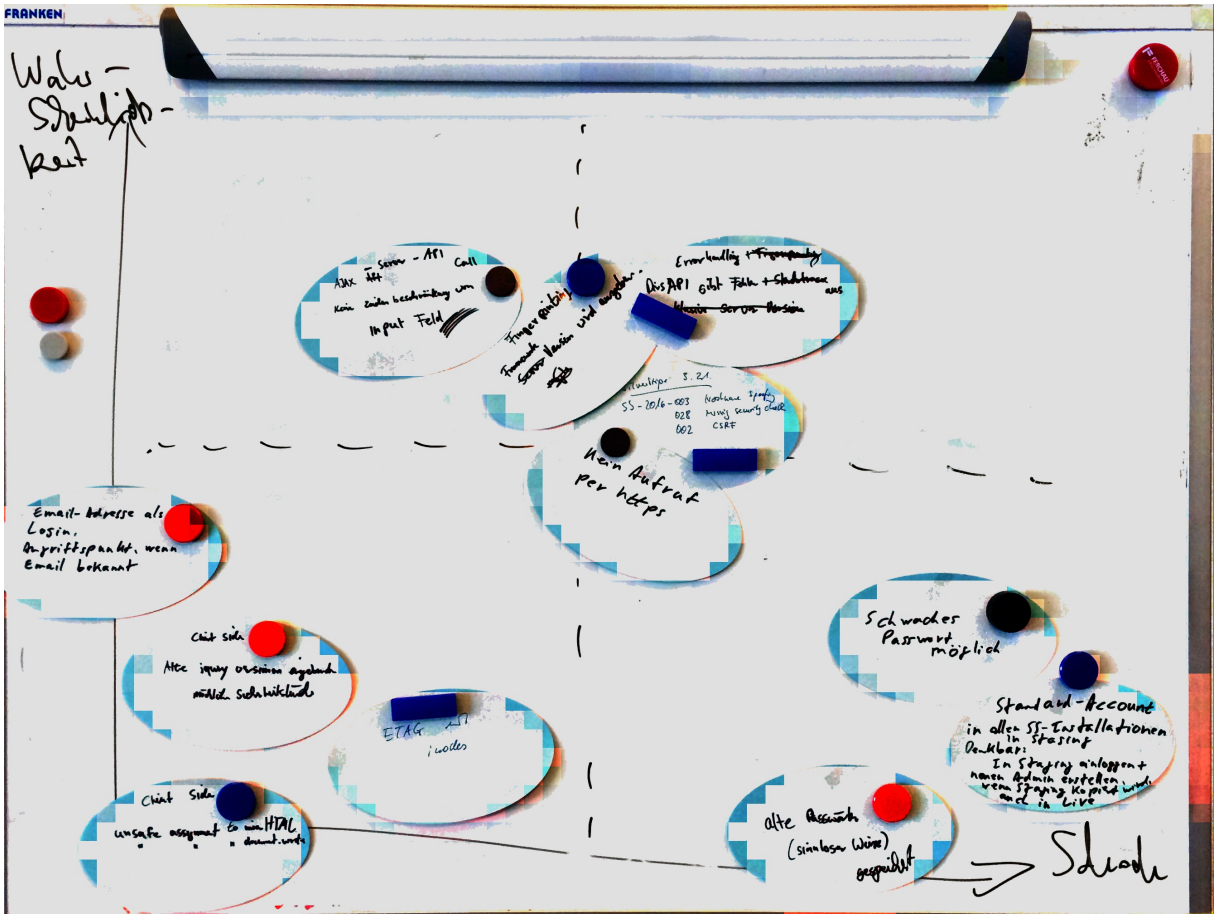


G.8 Ergebnisse der inhomogenen Team-Sicherheitsprüfungen

G.8.1 Gruppe E-Commerce



G.8.2 Gruppe CMS



G.8.3 Prüfpunktliste für Team-Sicherheitsprüfungen

Simplified Testing Guide v4 Checklist and Code Review Guide v2.0 alpha Checklist

By Prathan Phongthiproek, Timo Pagel

More Information: *https://kennel209.gitbooks.io/owasp-testing-guide-v4/*
https://www.owasp.org/images/7/78/OWASP_AlphaRelease_CodeReviewGuide2.0.pdf

Information Gathering	Test Name	Description	Tools	Done
OTG-INFO-002	Fingerprint Web Server	*Find the version and type of a running web server to determine known vulnerabilities and the appropriate exploits. Using "HTTP header field ordering" and "Malformed requests test".*	*Httprint, Httprecon, Desenmascarame*	
OTG-INFO-003	Review Webserver Metafiles for Information Leakage	*Analyze robots.txt and identify <META> Tags from website.*	*Browser, curl, wget*	
OTG-INFO-004	Enumerate Applications on Webserver	*Find applications hosted in the webserver (Virtual hosts/Subdomain), non-standard ports, DNS zone transfers*	*Webhosting.info, dnsrecon, Nmap, fierce, Recon-ng, Intrigue*	
OTG-INFO-005	Review Webpage Comments and Metadata for Information Leakage	*Find sensitive information from webpage comments and Metadata on source code.*	*Browser, curl, wget*	
OTG-INFO-006	Identify application entry points	*Identify from hidden fields, parameters, methods HTTP header analysis*	*Burp proxy, ZAP, Tamper data*	
OTG-INFO-007	Map execution paths through application	*Map the target application and understand the principal workflows.*	*Burp proxy, ZAP*	
OTG-INFO-008	Fingerprint Web Application Framework	*Find the type of web application framework/CMS from HTTP headers, Cookies, Source code, Specific files and folders.*	*Whatweb, BlindElephant, Wappalyzer*	
OTG-INFO-009	Fingerprint Web Application	*Identify the web application and version to determine known vulnerabilities and the appropriate exploits.*	*Whatweb, BlindElephant, Wappalyzer, CMSmap*	
OTG-INFO-010	Map Application Architecture	*Identify application architecture including Web language, WAF, Reverse proxy, Application Server, Backend Database*	*Browser, curl, wget*	

Configuration and Deploy Management Testing	Test Name	Description	Tools	Done
OTG-CONFIG-001	Test Network/Infrastructure Configuration	*Understand the infrastructure elements interactions, config management for software, backend DB server, WebDAV, FTP in order to identify known vulnerabilities.*	*Nessus, Iniscan, WAP*	
OTG-CONFIG-002	Test Application Platform Configuration	*Identify default installation file/directory, Handle Server errors (40*,50*), Minimal Privilege, Software logging.*	*Browser, Nikto*	
OTG-CONFIG-003	Test File Extensions Handling for Sensitive Information	*Find important file, information (.asa , .inc , .sql ,zip, tar, pdf, txt, etc)*	*Browser, Nikto*	
OTG-CONFIG-004	Backup and Unreferenced Files for Sensitive Information	*Check JS source code, comments, cache file, backup file (.old, .bak, .inc, .src) and guessing of filename*	*Nessus, Nikto, Wikto*	
OTG-CONFIG-005	Enumerate Infrastructure and Application Admin Interfaces	*Directory and file enumeration, comments and links in source (/admin, /administrator, /backoffice, /backend, etc), alternative server port (Tomcat/8080)*	*Burp Proxy, dirb, Dirbuster, fuzzdb, Tilde Scanner*	
OTG-CONFIG-006	Test HTTP Methods	*Identify HTTP allowed methods on Web server with OPTIONS. Arbitrary HTTP Methods, HEAD access control bypass and XST*	*netcat, curl*	
OTG-CONFIG-tpag	Test for Known Vulnerabilities Server Side	*Test for dependencies with known vulnerabilities*	*Sensio Labs Security Checker*	

Identity Management Testing	Test Name	Description	Tools	Done
OTG-IDENT-001	Test Role Definitions	*Validate the system roles defined within the application by creating permission matrix.*	*Burp Proxy, ZAP*	
OTG-IDENT-002	Test User Registration Process	*Verify that the identity requirements for user registration are aligned with business and security requirements:*	*Burp Proxy, ZAP*	
OTG-IDENT-004	Testing for Account Enumeration and Guessable User Account	*Generic login error statement check, return codes/parameter values, enumerate all possible valid userids (Login system, Forgot password)*	*Browser, Burp Proxy, ZAP*	
OTG-IDENT-005	Testing for Weak or unenforced username policy	*User account names are often highly structured (e.g. Joe Bloggsaccount name is jbloggs and Fred Nurks account name is fnurks) and valid account names can easily be guessed.*	*Browser, Burp Proxy, ZAP*	
OTG-IDENT-006	Test Permissions of Guest/Training Accounts	*Guest and Training accounts are useful ways to acquaint potential users with system functionality prior to them completing the authorisation process required for access.Evaluate consistency between access policy and guest/training account access permissions.*	*Burp Proxy, ZAP*	
OTG-IDENT-007	Test Account Suspension/Resumption Process	*Verify the identity requirements for user registration align with business/security requirements. Validate the registration process.*	*Burp Proxy, ZAP*	

Authentication Testing	Test Name	Description	Tools	Done
OTG-AUTHN-001	Testing for Credentials Transported over an Encrypted Channel	*Check referrer whether its HTTP or HTTPs. Sending data through HTTP and HTTPS.*	*Burp Proxy, ZAP*	
OTG-AUTHN-002	Testing for default credentials	*Testing for default credentials of common applications, Testing for default password of new accounts.*	*Burp Proxy, ZAP, Hydra*	
OTG-AUTHN-003	Testing for Weak lock out mechanism	*Evaluate the account lockout mechanism's ability to mitigate brute force password guessing. Evaluate the unlock mechanism's resistance to unauthorized account unlocking.*	*Browser*	
OTG-AUTHN-004	Testing for bypassing authentication schema	*Force browsing (/admin/main.php, /page.asp?authenticated=yes), Parameter Modification, Session ID prediction, SQL Injection*	*Burp Proxy, ZAP*	
OTG-AUTHN-005	Test remember password functionality	*Look for passwords being stored in a cookie. Examine the cookies stored by the application. Verify that the credentials are not stored in clear text, but are hashed. Autocompleted=off?*	*Burp Proxy, ZAP*	
OTG-AUTHN-006	Testing for Browser cache weakness	*Check browser history issue by clicking "Back" button after logging out. Check browser cache issue from HTTP response headers (Cache-Control: no-cache)*	*Burp Proxy, ZAP, Firefox add-on CacheViewer2*	
OTG-AUTHN-007	Testing for Weak password policy	*Determine the resistance of the application against brute force password guessing using available password dictionaries by evaluating the length, complexity, reuse and aging requirements of passwords.*	*Burp Proxy, ZAP, Hydra*	
OTG-AUTHN-009	Testing for weak password change or reset functionalities	*Test password reset (Display old password in plain-text?, Send via email?, Random token on confirmation email ?), Test password change (Need old password?), CSRF vulnerability ?*	*Browser, Burp Proxy, ZAP*	

Authorization Testing	Test Name	Description	Tools	Done
OTG-AUTHZ-001	Testing Directory traversal/file include	*dot-dot-slash attack (../), Directory traversal, Local File inclusion/Remote File Inclusion.*	*Burp Proxy, ZAP, Wfuzz, WAP*	
OTG-AUTHZ-002	Testing for bypassing authorization schema	*Access a resource without authentication?, Bypass ACL, Force browsing (/admin/adduser.jsp)*	*Burp Proxy (Autorize), ZAP*	
OTG-AUTHZ-003	Testing for Privilege Escalation	*Testing for role/privilege manipulate the values of hidden variables. Change some param groupid=2 to groupid=1*	*Burp Proxy (Autorize), ZAP*	
OTG-AUTHZ-004	Testing for Insecure Direct Object References	*Force changing parameter value (?invoice=123 -> ? invoice=456)*	*Burp Proxy (Autorize), ZAP*	
CODE-AUTHZ-01	Testing for password expiration	*Does application support password expiration?*	*Browser*	
CODE-AUTHZ-02	Testing for user privileges documentation	*User and role based privileges are documented*		

Data Validation Testing	Test Name	Description	Tools	Done
OTG-INPVAL-001	Testing for Reflected Cross Site Scripting	*Check for input validation, Replace the vector used to identify XSS, XSS with HTTP Parameter Pollution.*	*Burp Proxy, ZAP, Xenotix XSS, pixy, RIPS, WAP*	
OTG-INPVAL-002	Testing for Stored Cross Site Scripting	*Check input forms/Upload forms and analyze HTML codes*	*Burp Proxy, ZAP, BeEF, XSS Proxy, pixy, RIPS, WAP*	
OTG-INPVAL-003	Testing for HTTP Verb Tampering	*Craft custom HTTP requests to test the other methods to bypass URL authentication and authorization.*	*netcat, ZAP*	
OTG-INPVAL-005	MySQL Testing	*Identify MySQL version, Single quote, Information_schema, Read/Write file.*	*SQLMap, Mysqloit, Power Injector, RIPS*	
OTG-INPVAL-007	Testing for ORM Injection	*Testing ORM injection is identical to SQL injection testing*	*Hibernate, Nhibernate*	
OTG-INPVAL-008	Testing for XML Injection	*Check with XML Meta Characters ', " , <>, <!--/-->, &, <![CDATA[/]]>, XXE, TAG*	*Burp Proxy, ZAP, Wfuzz, RIPS*	
OTG-INPVAL-011	IMAP/SMTP Injection	*• Identifying vulnerable parameters with special characters (i.e.: \, ‘, “, @, #, !, |) • Understanding the data flow and deployment structure of the client • IMAP/SMTP command injection (Header, Body, Footer)*	*Burp Proxy, ZAP*	

OTG-INPVAL-012	Testing for Code Injection	*Enter OS commands in the input field. ?arg=1; system('id')*	*Burp Proxy, ZAP, Liffy, Panoptic, RIPS*
	Testing for Local File Inclusion	*LFI with dot-dot-slash (../../), PHP Wrapper (php://filter/convert.base64-encode/resource); Is open_basedir set in server configuration?*	*Burp Proxy, fimap, Liffy, RIPS*
	Testing for Remote File Inclusion	*RFI from malicious URL ?page.php?file=http://attacker.com/malicious_page*	*Burp Proxy, fimap, Liffy*
OTG-INPVAL-013	Testing for Command Injection	*Understand the application platform, OS, folder structure, relative path and execute OS commands on a Web server. %3Bcat%20/etc/passwd test.pdf+|+Dir C:*	*Burp Proxy, ZAP, Commix*
CODE-INPUTVAL-01	Testing of Input Validation for all requests	*Does the centralized validation get applied to all requests and all the inputs?*	
CODE-INPUTVAL-02	Testing of centralized validation check	*Does the centralized validation check block all the special characters?*	
CODE-INPUTVAL-03	Testing for untrusted inputs (e.g. external	*Are all the untrusted inputs validated?Input data is constrained and validated for type, length, format, and range.*	

Session Management Testing	Test Name	Description	Tools	Done
OTG-SESS-002	Testing for Cookies attributes	*Check HTTPOnly and Secure flag, expiration, inspect for sensitive data.*	*Burp Proxy, ZAP*	
OTG-SESS-003	Testing for Session Fixation	*The application doesn't renew the cookie after a successfully user authentication.*	*Burp Proxy, ZAP*	
OTG-SESS-005	Testing for Cross Site Request Forgery	*URL analysis, Direct access to functions without any token.*	*ZAP*	
OTG-SESS-006	Testing for logout functionality	*Check reuse session after logout both server-side and SSO.*	*Burp Proxy, ZAP*	
CODE-SESS-01	Testing for Sessions in URLs	*No session parameters are passed in URLs*		
CODE-SESS-02	Testing for session storage	*Session storage is secure*		
CODE-SESS-03	Testing for session timeouts	*Session inactivity timeouts are enforced*		

Error Handling	Test Name	Description	Tools	Done
OTG-ERR-001	Analysis of Error Codes	*Locate error codes generated from applications or web servers. Collect sensitive information from that errors (Web Server, Application Server, Database)*	*Burp Proxy, ZAP*	
OTG-ERR-002	Analysis of Stack Traces	*• Invalid Input / Empty inputs* *• Input that contains non alphanumeric characters or query syntax* *• Access to internal pages without authentication* *• Bypassing application flow*	*Burp Proxy, ZAP*	

Cryptography	Test Name	Description	Tools	Done
OTG-CRYPST-001	Testing for Weak SSL/TSL Ciphers, Insufficient Transport Layer Protection	*Identify SSL service, Idectify weak ciphers/protocols (ie. RC4, BEAST, CRIME, POODLE)*	*testssl.sh, SSL Breacher*	
OTG-CRYPST-003	Testing for Sensitive information sent via unencrypted channels	*Check sensitive data during the transmission: • Information used in authentication (e.g. Credentials, PINs, Session identifiers, Tokens, Cookies…) • Information protected by laws, regulations or specific organizational policy (e.g. Credit Cards, Customers data)*	*Burp Proxy, ZAP, Curl*	
CODE-CRYPT-01	Testing for resend cryptographic functions	*Are cryptographic functions used by the application the most recent version of these protocols, patched and process in place to keep them updated? (e.g. password hashes)*		
CODE-CRYPT-01	Testing for encrypted credentials	*Are database credentials stored in an encrypted format*		
CODE-CRYPT-02	Testing for unencrypted channels	*Is the data sent on encrypted channel? Does the application use HTTPClient for making external connections?*	*Proxy*	
CODE-CRYPT-03	Testing for keys in code	*Keys are not held in code.*		
Logging and Audit	**Test Name**	**Description**	**Tools**	**Result**
CODE-LOG-01	Test of sensitive information in logs	*Are logs logging personal information, passwords or other sensitive information?*		
CODE-LOG-02	Test for logging of connections	*Do audit logs log connection attempts (both successful and failures)?*		
CODE-LOG-03	Test for logs with malicious behaviors	*Is there a process(s) in place to read audit logs for unintended/malicious behaviors?*		

Business logic Testing	Test Name	Description	Tools	Done
OTG-BUSLOGIC-001	Test Business Logic Data Validation	*• Looking for data entry points or hand off points between systems or software.* *• Once found try to insert logically invalid data into the application/system.*	*Burp Proxy, ZAP*	
OTG-BUSLOGIC-002	Test Ability to Forge Requests	*• Looking for guessable, predictable or hidden functionality of fields.* *• Once found try to insert logically valid data into the application/system allowing the user go through the application/system against the normal busineess logic workflow.*	*Burp Proxy, ZAP*	
OTG-BUSLOGIC-003	Test Integrity Checks	*•Looking for parts of the application/system (components i.e. For example, input fields, databases or logs) that move, store or handle data/information.* *• For each identified component determine what type of data/information is logically acceptable and what types the application/system should guard against. Also, consider who according to the business logic is allowed to insert, update and delete data/information and in each component.* *• Attempt to insert, update or edit delete the data/information values with invalid data/information into each component (i.e. input, database, or log) by users that .should not be allowed per the busines logic workflow.*	*Burp Proxy, ZAP*	
OTG-BUSLOGIC-004	Test for Process Timing	*• Looking for application/system functionality that may be impacted by time. Such as execution time or actions that help users predict a future outcome or allow one to circumvent any part of the business logic or workflow. For example, not completing transactions in an expected time.* *• Develop and execute the mis-use cases ensuring that attackers* *can not gain an advantage based on any timing.*	*Burp Proxy, ZAP*	
OTG-BUSLOGIC-005	Test Number of Times a Function Can be Used Limits	*• Looking for functions or features in the application or system that should not be executed more that a single time or specified number of times during the business logic workflow.* *• For each of the functions and features found that should only be executed a single time or specified number of times during the business logic workflow, develop abuse/misuse cases that may allow a user to execute more than the allowable number of times.*	*Burp Proxy, ZAP*	

OTG-BUSLOGIC-006	Testing for the Circumvention of Work Flows	• Looking for methods to skip or go to steps in the application process in a different order from the designed/intended business logic flow. • For each method develop a misuse case and try to circumvent or perform an action that is "not acceptable" per the the business logic workflow.	Burp Proxy, ZAP
OTG-BUSLOGIC-007	Test Defenses Against Application Mis-use	Measures that might indicate the application has in-built self-defense: • Changed responses • Blocked requests • Actions that log a user out or lock their account	Burp Proxy, ZAP
OTG-BUSLOGIC-008	Test Upload of Unexpected File Types	• Review the project documentation and perform some exploratory testing looking for file types that should be "unsupported" by the application/system. • Try to upload these “unsupported” files an verify that it are properly rejected. • If multiple files can be uploaded at once, there must be tests in place to verify that each file is properly evaluated. PS. file.phtml, shell.phPWND, SHELL~1.PHP	Burp Proxy, ZAP
OTG-BUSLOGIC-009	Test Upload of Malicious Files	• Develop or acquire a known “malicious” file. • Try to upload the malicious file to the application/system and verify that it is correctly rejected. • If multiple files can be uploaded at once, there must be tests in place to verify that each file is properly evaluated.	Burp Proxy, ZAP
CODE-BUSLOG-01	Test for system privileges	Does the design use any elevated OS/system privileges for external connections/commands?	
CODE-BUSLOG-02	Test for reduced privileges	Are privileges reduce whenever possible?	
CODE-BUSLOG-03	Testing for protected classes	*Classes that contain security secrets (like passwords) are only accessible through protected API’s*	
CODE-BUSLOG-04	Testing for server side validation	*Data is validated on server side*	
CODE-BUSLOG-05	Testing for XML schema	*Is all XML input data validated against an agreed schema?*	
CODE-BUSLOG-06	Testing for output validation	*Is output that contains untrusted data supplied input have the correct type of encoding (URL encoding, HTML encoding)?*	

Client Side Testing	Test Name	Description	Tools	Result
OTG-CLIENT-001	Testing for DOM based Cross Site Scriptir	*Test for the user inputs obtained from client-side JavaScript Objects*	*Burp Proxy, DOMinator, jsprime, eslint*	
OTG-CLIENT-002	Testing for JavaScript Execution	*Inject JavaScript code: www.victim.com/?javascript:alert(1)*	*Burp Proxy, ZAP, eslint*	
OTG-CLIENT-003	Testing for HTML Injection	*Send malicious HTML code: ?user=<img%20src='aaa'%20onerror=alert(1)>*	*Burp Proxy, ZAP*	
OTG-CLIENT-004	Testing for Client Side URL Redirect	*Modify untrusted URL input to a malicious site: (Open Redirect) ?redirect=www.fake-target.site*	*Burp Proxy, ZAP*	
OTG-CLIENT-005	Testing for CSS Injection	*Inject code in the CSS context : • www.victim.com/#red;-o-link:'javascript:alert(1)';-o-link-source:current; (Opera [8,12]) • www.victim.com/#red;-:expression(alert(URL=1)); (IE 7/8)*	*Burp Proxy, ZAP*	
OTG-CLIENT-006	Testing for Client Side Resource Manipulation	*External JavaScript could be easily injected in the trusted web site www.victim.com/#http://evil.com/js.js*	*Burp Proxy, ZAP*	
OTG-CLIENT-007	Test Cross Origin Resource Sharing	*Check the HTTP headers in order to understand how CORS is used (Origin Header)*	*Burp Proxy, ZAP*	
OTG-CLIENT-009	Testing for Clickjacking	*Discover if a website is vulnerable by loading into an iframe, create simple web page that includes a frame containing the target.*	*Burp Proxy, ClickjackingTool*	
OTG-CLIENT-011	Test Web Messaging	*Analyse JavaScript code looking for how Web Messaging is implemented. How the website is restricting messages from untrusted domain and how the data is handled even for trusted domains*	*Burp Proxy, ZAP*	
OTG-CLIENT-012	Test Local Storage	*Determine whether the website is storing sensitive data in the storage. XSS in localstorage http://server/StoragePOC.html#<img src=x onerror=alert(1)>*	*Chrome, Firebug, Burp Proxy, ZAP*	
CODE-CLIENT-01	Test for Known Vulnerabilities Client Side	*Test for dependencies with known vulnerabilities*	*retire.js*	
CODE-CLIENT-02	Testing of output encoding	*Has the correct encoding been applied to all data being output by the application*		
CODE-CLIENT-03	Testing for WSDL validity	*Web service endpoints address in Web Services Description Language (WSDL) is checked for validity*		

Considerations

Skills required
Not Applicable
Security penetration skills
Network and programmaing skills
Advanced computer user
Some technical skills
no technical skills

Motive
Not Applicable
Low or no reward
Possible reward
High reward

Opportunity
Full access or expensive resources required
Special access or resources required
Some access or resources required
No access or resources required

Loss of confidentiality
Not Applicable
Minimal non-sensitive data disclosed
Extensive non-sensitive data disclosed
Extensive critical data disclosed
All data disclosed

Loss of Integrity
Not Applicable
Minimal slightly corrupt data
Minimal seriously corrupt data
Extensive slightly corrupt data
Extensive seriously corrupt data
All data totally corrupt

Loss of Availability
Not Applicable
Minimal secondary services interrupted
Minimal primary services interrupted
Extensive primary services interrupted
All services completely lost

Intrusion Detection
Not Applicable
Active detection in application
Logged and reviewed
Logged without review
Not logged

Privacy violation
Not Applicable
One individual
Hundreds of people
Thousands of people
Millions of people

Loss of Accountability
Not Applicable
Attack fully traceable to individual
Attack possibly traceable to individual
Attack completely anonymous

Financial damage
Not Applicable
Damage costs less than to fix the issue
Minor effect on annual profit
Significant effect on annual profit
Backruptcy

Reputation damage
Not Applicable
Minimal damage
Loss of major accounts
Loss of goodwill
Brand damage

Non-Compliance
Not Applicable
Minor violation
Clear violation
High profile violation

Population Size	Easy of Discovery	Ease of Exploit	Awareness
Not Applicable	Not Applicable	Not Applicable	Not Applicable
System Administrators	Practically impossible	Theoretical	Unknown
Intranet Users	Difficult	Difficult	Hidden
Partners	Easy	Easy	Obvious
Authenticated users	Automated tools available	Automated tools available	Public knowledge
Anonymous Internet users			

Anhang H

Quellcode mit DOM-basierter XSS-Schwachstelle

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>DOM XSS Example</title>
<style>
.active {
        background-color: grey;
}
</style>
</head>
<body>
<div id="chooseLanguage">
Select your language (active:
<script>
function getLanguageFromFragment() {
        console.log(document.location.href.substring(document.location.
           href.indexOf("default=")+8));
        return document.location.href.substring(document.location.href.
           indexOf("default=")+8);
}

document.write(getLanguageFromFragment());
</script>):
<select>
<script>
```

```
25 function isLanguageSelected(language) {
26         if(getLanguageFromFragment() == language) {
27                 return true;
28         }else {
29                 return false;
30         }
31 }
32
33 var languages = {
34         "de" : "Deutsch",
35         "en" : "English"
36 };
37 for(var i in languages) {
38         var option="<OPTION value='"+i+"'";
39         if(isLanguageSelected(i)) {
40
41         option = option +"selected";
42         }
43         option = option +">"+languages[i]+"</OPTION>";
44         document.write(option);
45 }
46 </script>
47 </select>
48 </div>
49
50 <script>
51 var elem = document.getElementById('chooseLanguage');
52 elem.addEventListener('mouseover', mouseOver);
53 elem.addEventListener('mouseout', mouseOut);
54 elem.addEventListener('click', click);
55
56   function mouseOver() {
57     elem.classList.add("active");
58   }
59
60   function mouseOut() {
61     elem.classList.remove("active");
62   }
63   function click(el) {
```

```
64          if(el.target.value != document.location.href.substring(document.
                location.href.indexOf("default=")+8)) {
65                  location.href="#default="+el.target.value;
66          }
67    }
68 </script>
69 </body>
70 </html>
```

Anhang I

Übersicht ausgewählte über Werkzeuge zur statischen Analyse

Prog.-Sprache	**Name**	**Technologie**	**Integration**	**Beschreibung**	**Letzte Änderung**	**Praxistauglich**
Viele[1]	PMD [96]	Stilanalyse, *Copy/Paste Detector*	Ant, IDE, Maven	Erkennung von Programmkonvention-Verletzungen, unbenutztem Quellcode, „schlechte“ Praktiken und Quellcode-Duplikaten.	April 2016	Ja
.NET, Java, Scala	OWASP Dependency Check [162]	Abhängigkeiten	Ant, Konsole, Jenkins, Maven	Erkennung veralteter Bibliotheken.	April 2016	Ja
Java	Checkstyle [84]	Stilanalyse	Ant, IDE, Jenkins, Konsole, Maven	Erkennung von Programmkonvention-Verletzungen	April 2016	Ja

[1] C, C++, C#, Fortran, Java, JavaScript, PHP, Ruby, XML

Java, Scala	FindBugs [61] mit FindSecurity-Bugs [11]	Stilanalyse, Datenfluss-analyse	Konsole, Jenkins, IDE, Maven	Erkennung von Laufzeit- und Logikfehlern und die Nutzung gefährlicher Methoden.	April 2016	Ja
Java-Script	ESLint mit den Plugins: *eslint-plugin-scanjs-rules [77]* und *eslint-plugin-no-unsafe-innerhtml*	Stilanalyse	Konsole, IDE	Erkennung nicht maskierter Ausgabe.	Januar 2016	Ja
Java-Script	retire.js	Abhängigkeiten	Konsole, Browser, (Scan-)Plugin	Erkennung veralteter Bibliotheken via HTTP oder auf Dateisystemebene.	April 2016	Ja
PHP	PHP Security Scanner [113]	String Matching Algorithmus	Konsole	Erkennung nicht maskierter Funktionsaufrufe bei der Nutzung von `mysql_query`	Juli 2015	Begrenzt
PHP	PHPCodeSniffer	Stilanalyse	Konsole	Erkennung und Behebung von Programmkonvention-Verletzungen	April 2016	Ja
PHP	Pixy [143]	Datenfluss-analyse	IDE	Erkennung nicht maskierter Funktionsaufrufe (SQLi / XSS). Operiert auf einzelnen Dateien.	Dezember 2014	Nein
PHP	SensioLabs Security Checker [199]	Abhängigkeiten	Konsole	Erkennung veralteter Bibliotheken über *composer*.	November 2015	Ja

PHP	WAP [169]	Datenfluss-analyse	Konsole	Erkennung nicht maskierter Funktionsaufrufe.	November 2015	Nein
Ruby	bundler-audit	Abhängigkeiten	Konsole	Erkennung veralteter Bibliotheken und unsicherer Bezugsquellen (HTTP).	April 2016	Ja

Tabelle I.2: Übersicht über ausgewählte Werkzeuge zur statischen Analyse

Anhang J

Illustration von Konflikten zwischen Penetrations-Testern und Entwicklern

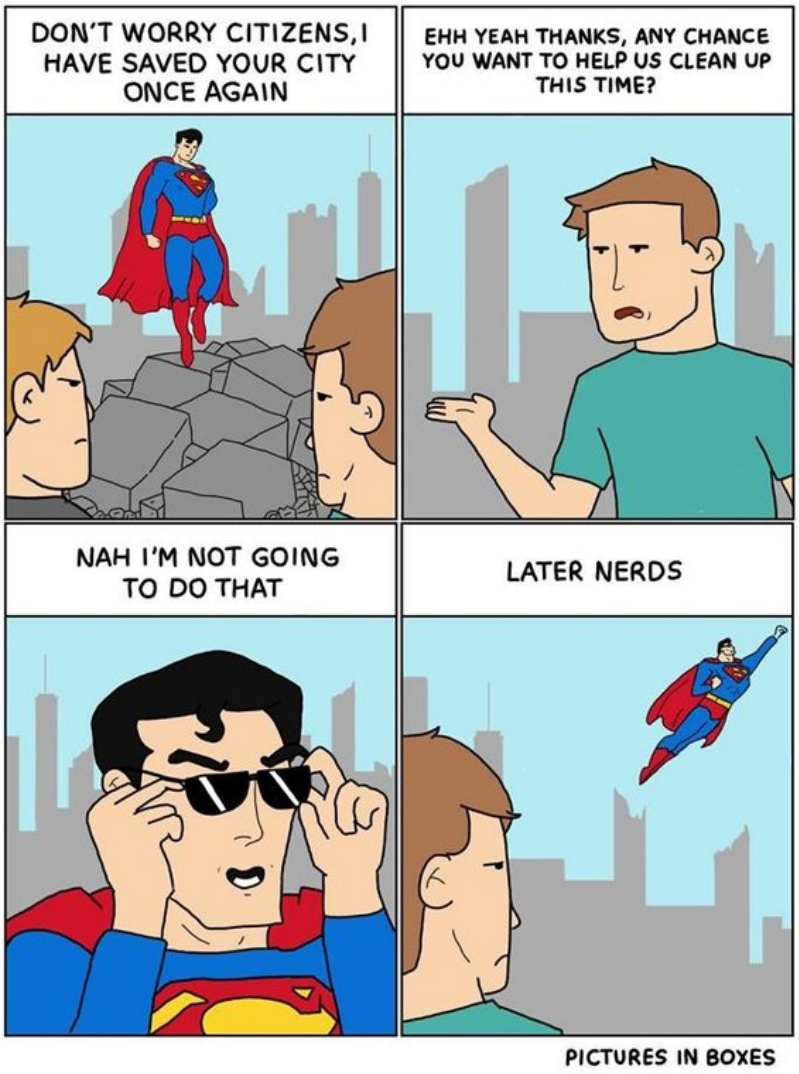


Abbildung J.1: Illustration von Konflikten zwischen Penetrations-Testern und Entwicklern
Quelle: https://twitter.com/pencilsareneat/status/724711158863790084, abgerufen am 21.05.2016

Anhang K

Generisches *DevOps*-Sicherheit Reifegradmodell

K.1 Übersicht über alle Dimensionen und Ebenen

Siehe folgende Seiten.

Generisches DevOps-Sicherheits-Reifegradmodell

Dimension	Unter-Dimension	Ebene 1: Grundverständnis von Sicherheitspraktiken	Ebene 2: Erweitertes Verständnis von Sicherheitspraktiken	Ebene 3: Hohes Verständnis von Sicherheitspraktiken	Ebene 4: Sehr hohes Verständnis von Sicherheitspraktiken bei Skalierung
Erzeugung und Verteilung	Erzeugung	• Definierter Erzeugungs-Prozess	• Regelmäßiger Test	• Gleiches Artefakt für Umgebungen • Versionierte Artefakte • Versionierung der Konfiguration	• Artefakte sind signiert • Erzeugung von Artefakten in virtuellen U mgebungen
Erzeugung und Verteilung	Verteilung	• Definierter Verteilungs-Prozess	• Austausch von Konfigurationsparametern • Backup vor Verteilung	• Lückenlose Verteilung • Umgebungsunabhängige Konfigurationsp arameter • Verschlüsselung von Konfigurationsparm etern	• Kunden-Rückmeldungs-Umgebung • Selektierte Verteilung
Informationsgewin nung	Überwachung und Metrik	• Einfache Anwendungs- und System-Metri ken	• Alarmierung • Visualisierte Metriken	• Deaktivierung ungenutzter Metriken • Erweiterte Verfügbarkeits- und Stabilitäts metriken • Erweiterte Webanwendungsmetriken • Sinnvolle Metriken-Gruppierung • Zielgerichtete Alarmierung	• Abdeckungs- und Kontroll-Metriken • Bildschirme zeigen Metriken • Metriken sind kombiniert mit Tests • Verteidungsmetriken
Informationsgewin nung	Protokollierung	• Zentrale Protokollierung	• Visualisierte Protokollierung	• Ausnahmen von Anwendungen werden e rfasst	• Korrelation von Sicherheits-Ereignissen
Infrastruktur	Infrastruktur	• Interne Systeme sind einfach geschützt • Produktiv-Umgebung und Test-Umgebun g	• Anwendungen laufen in virtuellen Umgeb ungen • Produktionsnahe Umgebung steht Entwic klern zur Verfügung • Prüfung von Quellen eingesetzter Softwar e	• Kontrollierte Netzwerke für virtuelle Umge bungen • Produktions-Artifakte sind versioniert • Provisionierung von Systemen • Rollen basierte Authentifizierung und Aut orisierung • Virtuelle Umgebungen sind limitiert	• Betriebssystem-Aufrufe von Containern si nd limitiert • Microservice-Architektur • Zufälliges herunterfahren von Systemen
Kultur und Organisation	Kultur und Organisation	• Informations-Sicherheits-Ziele sind komm uniziert	• Erstellung einfacher AbUser Stories • Gute Kommunikation wird belohnt • Pro Team ist ein Sicherheitsverantwortlic her definiert	• Durchführung von einfachen Bedrohungs analysen • Review bei jeder neuen Version • Sicherheits-Lessoned-Learned • Sicherheitsprüfungen gemeinsam mit Ent wicklern und System-Administratoren	• Durchführung von Team-Sicherheitsprüfu ngen • Durchführung von War Games • Durchführung von erweiterten Bedrohung sanalysen • Ein Sicherheitsexperte pro Team • Erstellung erweiterter AbUser Stories

Dimension	Unter-Dimension	Ebene 1: Grundverständnis von Sicherheitspraktiken	Ebene 2: Erweitertes Verständnis von Sicherheitspraktiken	Ebene 3: Hohes Verständnis von Sicherheitspraktiken	Ebene 4: Sehr hohes Verständnis von Sicherheitspraktiken bei Skalierung
Test und Verifizierung	Dynamische Tiefe	• Einfacher Scan	• Abdeckung von Rollen • Dynamischer Spiderdurchlauf	• Abdeckung sequentieller Aktionen • Abdeckung versteckter Pfade	• Abdeckung von nicht erkannten Eingabe-Vektoren • Abdeckung von serverseitigen Komponenten • Abdeckungsanalyse • Nutzung zusätzlicher Web-Security-Scanner
Test und Verifizierung	Statische Tiefe	• Test auf serverseitige Komponenten	• Statische Analyse für wichtige serverseitige Bereiche	• Statische Analyse für wichtige klientenseitige Bereiche • Test auf klientenseitige Komponenten	• Ausschluss von Quellcode-Dupliakten • Statische Analyse für Bibliotheken • Statische Analyse für alle Bereiche • Stilanalyse
Test und Verifizierung	Test-Intensität	• Standardeinstellungen für Test-Intensität	• Deaktivierung unnötiger Prüfungen	• Angepasste Test-Intensität	• Test-intensität ist hoch eingestellt
Test und Verifizierung	Konsolidierung	• Behandlung kritischer Alarme • Einfache Falsch-Positiv-Behandlung	• Alarme sind einfach visualisiert	• Alarme adressieren Teams • Behandlung von mittelschweren Alarmen	• Aggregation von Alarmen • Alarme sind erweitert visualisiert • Behandlung von allen Alarmen • Reproduzierbare Alarme
Test und Verifizierung	Anwendungstest	• Modultests mit Sicherheitsbezug	• Integrationstests mit Sicherheitsbezug	• Akzeptanztests mit Sicherheitsbezug	• Post-Verteilungs-Prüfung • Sehr hoch abdeckende Komponenten-, Integrations- und Akzeptanztests mit Sicherheitsbezug
Test und Verifizierung	Infrastrukturtest	• Test auf System-Aktualisierungen	• Prüfung der Konfiguration von virtuellen Umgebungen	• Test auf schwache Passwörter	• Erweiterte System-Prüfung • Lasttests • Visualisierung von System-Updates

K.2 Detaillierte Auflistung aller Implementierungspunkte

Siehe folgende Seiten.

Dimension Erzeugung und Verteilung

Unter-Dimension Erzeugung

Definierter Erzeugungs-Prozess

Risiken und Maßnahmen

Risiko: Die Erzeugung kann bei jedem mal unterschiedlich durchgeführt werden. Wird ein Fehler dabei gemacht können sicherheitsrelevante Konfigurationen fal sch gesetzt werden.
Gegenmaßnahme: Es existiert ein definierter automatisierter Prozess für die Erzeugung, welcher manuell angestoßen werden kann.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Es wird die Verfügbarkeit erhöht, da Fehler reduziert werden.
Integrität: Es wird die Wahrscheinlichkeit reduziert versehentlich Daten zu verändern.

Sonstiges

Implementierung: Jenkins, Docker

Regelmäßiger Test

Risiken und Maßnahmen

Risiko: Vom pushen von Quellcode in die Versionskontrolle bis zur Rückmeldung, dass dieser Quellcode eine Schwachstelle enthält, kann Zeit vergehen. Dadur ch ist es für den Entwickler schwieriger, gepushten Quellcode nachzuvollziehen und die Schwachstelle nachhaltig zu beseitigen.
Gegenmaßnahme: Bei jedem Push oder periodisch wird eine Verteilung auf eine Testumgebung durchgeführt und automatisch Tests- und Verifikationen durchg eführt, mindestens für die geänderten Quellcode-Bereiche.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Sehr wenig (eine Disziplin)
Benötigte Zeit: Sehr wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Integrität: Für jede Änderung der Software erfolgt zeitnah eine Prüfung. Bei versehentlichem Einführen von Schwachstellen in den Quellcode wird eine Rückmel dung gegeben, so dass die ungewollte Schwachstelle entfernt werden kann.

Sonstiges

Abhängigkeiten: Definierter Erzeugungs-Prozess

Gleiches Artefakt für Umgebungen

Risiken und Maßnahmen

Risiko: Es wird ein unterschiedliches Artefakt beziehugnsweise eine unterschiedliche Abbildung der Anwendung für die Testumgebung und die Produktionsumg ebung verwendet. Entsprechend können auf der Produktionsumgebung unerwartete Effekte auftreten.
Gegenmaßnahme: Das gleiche Artefakt der Anwendung von der Testumgebung wird auf der Produktionsumgebung verwendet.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Integrität: Es ist sicher gestellt, dass das selbe Artefakt der Testumgebung auf die Produktionsumgebung deployed wird.
Verfügbarkeit: Es ist sicher gestellt, dass nur geprüfte Artefakte auf der Produktionsumgebung verwendet werden, so dass keine ungeprüften ggf. fehlerhaften Artefakte deployed werden.

Sonstiges

Abhängigkeiten: Definierter Erzeugungs-Prozess
Implementierung: Docker

Versionierte Artefakte

Risiken und Maßnahmen

Risiko: Ein Artefakt enthält eine Schwachstelle oder verursacht unerwartete Effekte, nachdem es auf Produktion ausgerollt wurde.
Gegenmaßnahme: Verteilungen werden versioniert. Es ist einfach auf eine vorherige Version zurück zu greifen.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Mittel

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch versionierte Artefakte kann bei Fehlern bei Verteilungen schnell auf eine vorherige Version umgeschaltet werden.
Integrität: Durch versionierte Artefakte ist die Integrität von Software-Artefakten sicher gestellt.

Sonstiges

Abhängigkeiten: Gleiches Artefakt
Implementierung: Docker

Versionierung der Konfiguration

Risiken und Maßnahmen

Risiko: Es ist nicht nachvollziehbar, wie die Konfiguration der Erzeugungsumgebung verändert wurde.
Gegenmaßnahme: Durch Versionierung der Konfiguration von Aufträgen im Continuous Integration Server können Änderungen nachvollzogen werden.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Integrität: Durch versionierte Konfiguration ist die Integrität der Konfiguration der Erzeugungsumgebung sichergestellt.

Sonstiges

Abhängigkeiten: Definierter Erzeugungs-Prozess
Implementierung: JobConfigHistory Plugin in Jenkins

Artefakte sind signiert

Risiken und Maßnahmen

Risiko: Manipuliert ein Angreifer ein Artefakt oder ein Abbild, wird dies ggf. nicht bemerkt.
Gegenmaßnahme: Durch Signierung und Signatur-Prüfungen von Artefakten und Abbildern ist sichergestellt, dass eine Manipulation oder der Austausch eines Artefakts beziehugnsweise Abbilds bemerkt wird.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Wenig

Gewährleistete Sicherheitseigenschaften

Integrität: Durch Signierung von Artefakten und Abbildern ist sichergestellt, dass eine Manipulation oder der Austausch eines Artefakts beziehugnsweise Abbild s bemerkt wird.

Sonstiges

Abhängigkeiten: Definierter Erzeugungs-Prozess

Erzeugung von Artefakten in virtuellen Umgebungen

Risiken und Maßnahmen

Risiko: Erlangt ein Angreifer Zugriff auf das Versionskontrollsystem eines Projekts oder auf die Konfiguration zur Erzeugung, kann dieser ggf. Zugriff auf das Erz eungs-System erlangen und dadurch andere Erzeugungsaufträge kompromittieren.
Gegenmaßnahme: Jeder Schritt der Erzeugung findet in einer seperaten virtuellen Umgebung statt.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Wenig

Gewährleistete Sicherheitseigenschaften

Integrität: Da Erzeungsaufträge sich nicht gegenseitig beeinflussen können, ist die Integrität nicht durch andere Erzeugungsaufträge gefährdet.

Sonstiges

Abhängigkeiten: Definierter Erzeugungs-Prozess

Unter-Dimension Verteilung

Definierter Verteilungs-Prozess

Risiken und Maßnahmen

Risiko: Verteilungen können unterschiedlich durchgeführt werden. Wird ein Fehler bei der Verteilung gemacht, welcher manuell korrigiert werden muss, kann di e Verfügbarkeit beeinträchtigt werden. Mögliche Scenarien sind entsprechend: Es können Sicherheits-Tests, welche das Abbild validieren vergessen werden. Es wird ein Abbild erzeugt, allerdings ein anderes Abbild deployed.
Gegenmaßnahme: Durch einen definierten Verteilungs-Prozess wird die Verfügbarkeit erhöht, da Fehler reduziert werden.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Es wird die Verfügbarkeit erhöht, da Fehler reduziert werden.
Integrität: Es wird die Wahrscheinlichkeit reduziert versehentlich Daten zu löschen.

Sonstiges

Implementierung: Jenkins, Docker

Austausch von Konfigurationsparametern

Risiken und Maßnahmen

Risiko: Angreifer, welche Zugang zum Quellcode und damit zur Konfiguration erhalten, können schutzwürdige Informationen wie Datenbank-Zugänge einsehen.
Gegenmaßnahme: Bei Verteilungen werden schutzbedürftige Konfigurationsparameter je nach Umgebung gesetzt. So kann beispielsweise der Datenbank-Zug ang über Umgebungsvariablen gesetzt werden.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Vertraulichkeit: Nur autorisierte Personen/Systeme erhalten Zugriff auf vertrauliche Konfigurationsparameter.
Integrität: Nur autorisierte Personen/Systeme können vertrauliche Konfigurationsparameter verändern.

Sonstiges

Backup vor Verteilung

Risiken und Maßnahmen

Risiko: Durch das Einspielen einer Aktualisierung in einer DBMS-Software können Fehler auftreten, welche zu Datenverlust führen.
Gegenmaßnahme: Automatische Backups werden vor der Verteilung neuer Software durchgeführt, sofern die Datenmenge dies in einer angemessenen Zeit zul ässt. Wiederherstellung ist geprüft.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Sehr wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Es wird die Verfügbarkeit erhöht, da bei Störungen eine Möglichkeit zur Wiederherstellung eines früheren Stands möglich ist.
Integrität: Durch ein Backup kann die Integrität von Daten nach einem Angriff geprüft werden.

Sonstiges

Abhängigkeiten: Definierter Verteilungs-Prozess

Lückenlose Verteilung

Risiken und Maßnahmen

Risiko: Durch eine Verteilung ist die Verfügbarkeit des Systems beeinträchtigt.
Gegenmaßnahme: Es ist ein lückenloser Verteilungs-Prozess definiert.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Es besteht weniger Beeinträchtigung der Verfügbarkeit bei Verteilungen.

Sonstiges

Abhängigkeiten: Definierter Verteilungs-Prozess
Implementierung: Docker, Webserver

Umgebungsunabhängige Konfigurationsparameter

Risiken und Maßnahmen

Risiko: Es werden unterschiedliche Aktionen in der Testumgebung und der Produktionsumgebung ausgeführt. Beispielsweise folgender Quellcode: if (host == 'p roduction') {} else {}
Gegenmaßnahme: Es werden Umgebungsvariablen oder Parameter beim Starten des Artefakts verwendet. Verhalten wird nur über Konfiguration gesteuert und nicht über Hostnamen o.ä..

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Sehr wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch vermeidung unterschiedlicher Aktionen in Test- und Produktionsumgebung ist gleiches Verhalten sicher gestellt und damit die Verfügbarke it erhöht.

Sonstiges

Abhängigkeiten: Gleiches Artefakt
Implementierung: Docker

Verschlüsselung von Konfigurationsparmetern

Risiken und Maßnahmen

Risiko: Angreifer, welche Zugang zum Quellcode und damit zur Konfiguration erhalten, können schutzwürdige Informationen wie Datenbank-Zugänge einsehen.
Gegenmaßnahme: Bei Verteilungen werden schutzbedürftige Konfigurationsparameter je nach Umgebung gesetzt. So kann beispielsweise der Datenbank-Zug ang über Umgebungsvariablen gesetzt werden.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Vertraulichkeit: Nur autorisierte Personen/Systeme erhalten Zugriff auf vertrauliche Konfigurationsparameter.
Integrität: Nur autorisierte Personen/Systeme können vertrauliche Konfigurationsparameter verändern.

Sonstiges

Abhängigkeiten: Austausch von Konfigurationsparmetern

Kunden-Rückmeldungs-Umgebung

Risiken und Maßnahmen

Risiko: Es sind schwer durch Automatisierung zu findene Schwachstellen in der Anwendung vorhanden.
Gegenmaßnahme: Kunden haben Zugriff auf eine Vor-Produktions-Version und können das System prüfen.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Sehr wenig (eine Disziplin)
Benötigte Zeit: Sehr wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Selektierte Verteilung

Risiken und Maßnahmen

Risiko: Durch eine Verteilung kann die Verfügbarkeit des Systems gefährdet sein.
Gegenmaßnahme: Es wird nur auf einen Server die Verteilung angewendet und anschließend eine Post-Verteilungs-Prüfung vorgenommen, nur wenn diese erf olgreich ist, wird auf weitere Server deployed.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Sehr wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Es besteht weniger Beeinträchtigung der Verfügbarkeit bei Verteilungen.

Sonstiges

Abhängigkeiten: Post-Verteilungs-Prüfung, Produktiv-Umgebung und Test-Umgebung

Dimension Informationsgewinnung

Unter-Dimension Überwachung und Metrik

Einfache Anwendungs- und System-Metriken

Risiken und Maßnahmen

Risiko: Systemadministratoren und Entwickler müssen, um einen Überblick über verschiedene virtuelle Systeme zu erlangen, sich auf diesen einloggen. Insbes ondere Entwicklern ohne Linux-Kentnisse fällt die Auswertung von Protokollen auf Grundlage der Linux-Befehle cat, grep und awk schwer.
Gegenmaßnahme: Einfache Anwendungs- und System-Metriken sind erfasst.

Nutzen und Schwere der Implementierung

Nutzen: Sehr hoch
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Wenig

Gewährleistete Sicherheitseigenschaften

Integrität: Durch Metriken während eines Angriffs können Informationen gewonnen werden, durch welche ein Angriff auf die Integrität von Daten abgewehrt wer den kann.
Verfügbarkeit: Durch Trendanalysen können ungewollte Systemausfälle verhindert werden.
Vertraulichkeit: Durch Metriken während eines Angriffs können Informationen gewonnen werden, durch welche ein Angriff abgewehrt werden kann.

Alarmierung

Risiken und Maßnahmen

Risiko: Es wird zu spät gemerkt, wenn Systeme ungewöhnliches verhalten aufweisen.
Gegenmaßnahme: Grenzen für Metriken sind definiert und das System alarmiert.

Nutzen und Schwere der Implementierung

Nutzen: Sehr hoch
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Sehr viel
Benötigte Ressourcen (Systeme): Sehr viele

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch frühzeitig Alarmierung können Systemausfälle verhindert werden.

Sonstiges

Abhängigkeiten: Visualisierte Metriken

Visualisierte Metriken

Risiken und Maßnahmen

Risiko: Metriken werden mangelhaft dargestellt und können deshalb nur begrenzt ausgewertet werden.
Gegenmaßnahme: Metriken sind visuell in Echtzeit dargestellt. Dabei unterstützt eine benutzerfreundliche Bedienoberfläche.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Sehr wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Wenig

Gewährleistete Sicherheitseigenschaften

Integrität: Durch visualisierte Metriken während eines Angriffs können erweiterte Informationen gewonnen werden, durch welche ein Angriff auf die Integrität vo n Daten abgewehrt werden kann.
Verfügbarkeit: Durch visualisierte Trendanalysen können ungewollte Systemausfälle verhindert werden.
Vertraulichkeit: Durch visualisierte Metriken während eines Angriffs können erweiterte Informationen gewonnen werden, durch welche ein Angriff abgewehrt we rden kann.

Sonstiges

Abhängigkeiten: Anwendungs- und System-Metriken

Deaktivierung ungenutzter Metriken

Risiken und Maßnahmen

Risiko: Durch sammeln ungenutzter Metriken werden Ressourcen verschwendet, welche für sicherheitsrelevante Dienste genutzt werden könnten.

Gegenmaßnahme: Durch Deaktivierung ungenutzter Metriken stehen mehr Ressourcen zur Verfügung.

Nutzen und Schwere der Implementierung

Nutzen: Sehr hoch
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Sehr viel
Benötigte Ressourcen (Systeme): Sehr viele

Sonstiges

Abhängigkeiten: Visualisierte Metriken

Erweiterte Verfügbarkeits- und Stabilitätsmetriken

Risiken und Maßnahmen

Risiko: Es sind nicht ausreichend Metriken erfasst um alle Trends zu erfassen oder bei einem Angriff ausreichend Informationen zu erhalten.
Gegenmaßnahme: Erweiterte Metriken um die Verfügbarkeit und Stabilität zu erfassen. Insbesondere ungeplante Ausfallzeiten sollten erfasst werden, da diese zu Vertragsstrafen führen können. Typischerweise werden diese über eine Periode, beispielsweise ein Jahr, erfasst.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Trendanalysen aufgrund erweiterter Metriken können ungewollte Systemausfälle verhindert werden.

Sonstiges

Abhängigkeiten: Anwendungs- und System-Metriken, Visualisierte Metriken

Erweiterte Webanwendungsmetriken

Risiken und Maßnahmen

Risiko: Das Sicherheitsniveau der Webanwendung ist unbekant.
Gegenmaßnahme: Alle Ergebnisse aus der Dimension Test- und Verifizierung werden instrumentiert.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Wenig

Gewährleistete Sicherheitseigenschaften

Integrität: Durch Instrumentierung der durchgeführten Metriken wird das Sicherheitsniveau kommuniziert und die Sicherheit der Webanwendung langfristig erhöht.
Verfügbarkeit: Durch Instrumentierung der durchgeführten Metriken wird das Sicherheitsniveau kommuniziert und die Sicherheit der Webanwendung langfristig erhöht.
Vertraulichkeit: Durch Instrumentierung der durchgeführten Metriken wird das Sicherheitsniveau kommuniziert und die Sicherheit der Webanwendung langfristig erhöht.

Sonstiges

Abhängigkeiten: Anwendungs- und System-Metriken, Visualisierte Metriken

Sinnvolle Metriken-Gruppierung

Risiken und Maßnahmen

Risiko: Da sicherheitsrelevante Metriken nicht gruppiert sind, kann es zu Verzögerungen bei der Analyse von Vorfällen kommen.
Gegenmaßnahme: Metriken sind sinnvoll gruppiert.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Viel
Benötigte Ressourcen (Systeme): Wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch erhöhte Reaktionszeit können Systemausfälle verhindert werden.

Zielgerichtete Alarmierung

Risiken und Maßnahmen

Risiko: Es werden falsche Personen über einen Vorfall informiert.
Gegenmaßnahme: Durch zielgerichtete Information über Vorfälle kann besser auf Vorfälle reagiert werden.

Nutzen und Schwere der Implementierung

Nutzen: Sehr hoch
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Sehr viel
Benötigte Ressourcen (Systeme): Sehr viele

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch erhöhte Reaktionszeit können Systemausfälle verhindert werden.

Sonstiges

Abhängigkeiten: Alarmierung

Abdeckungs- und Kontroll-Metriken

Risiken und Maßnahmen

Risiko: Die Effektivität von Kontrollmanahmen wie Konfiguration, Patch und Schwachstellen Management ist unbekannt.
Gegenmaßnahme: Einführung von Abdeckungs- und Kontroll-Metriken. Durch Abdeckungs- und Kontroll-Metriken wird aufgezeigt wie effektiv das Sicherheits-P rogramm einer Organisation ist. Sicherheits-Programme sind u.a. durch unternehmensweite Richtlinien gestützt, allerdings werden diese nicht immer eingehalte n. Abdeckung ist der Grad zu welcher eine bestimmte Sicherheitskontrolle für eine bestimmte Zielgruppe mit allen Ressourcen angewendet wird. Der Kontroll-Gr ad zeigt die tatsächliche Anwendung von vorgegebenen Sicherheits-Standards und -Richtlinien. Entsprechend werden durch Abdeckungs- und Kontroll-Metriken Lücken bei der Umsetzung von Richtlinien und Standards aufgezeigt. Umfassende Abdeckungs- und Kontroll-Metriken beinhalten das Sammeln von Information en zu Anti-Virus Software sowie Anti-Rootkits, Patch Management, Server-Konfiguration und Schwachstellen-Management.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Sehr viel
Benötigte Ressourcen (Systeme): Wenig

Gewährleistete Sicherheitseigenschaften

Integrität: Durch Kenntnis der Effektivität von Kontrollmanahmen können Bedrohungen, welche die Integrität gefährden abgewehrt werden.
Verfügbarkeit: Durch Kenntnis der Effektivität von Kontrollmanahmen können Bedrohungen, welche die Verfügbarkeit gefährden abgewehrt werden.
Vertraulichkeit: Durch Kenntnis der Effektivität von Kontrollmanahmen können Bedrohungen, welche die Vertraulichkeit gefährden abgewehrt werden.

Sonstiges

Abhängigkeiten: Anwendungs- und System-Metriken, Visualisierte Metriken

Bildschirme zeigen Metriken

Risiken und Maßnahmen

Risiko: Sicherheitsrelevante Informationen, z.B. bei einem Angriff, werden verspätet erkannt.
Gegenmaßnahme: Ein intern zugänglicher Bildschirm zeigt sicherheitsrelevante Metriken.

Nutzen und Schwere der Implementierung

Nutzen: Sehr hoch
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Sehr wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch erhöhte Reaktionszeit können Systemausfälle verhindert werden.

Sonstiges

Abhängigkeiten: Sinnvolle Metriken-Gruppierung

Metriken sind kombiniert mit Tests

Risiken und Maßnahmen

Risiko: Änderungen führen zu erhöhter Last aufgrund Programmierfehler.
Gegenmaßnahme: Erweiterte Metriken werden bei Tests aufgezeichnet und ausgewertet.

Nutzen und Schwere der Implementierung

Nutzen: Sehr hoch
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch erhöhte Sichtbarkeit von Metriken bei Tests wird die Verfügbarkeit erhöht.

Sonstiges

Abhängigkeiten: Sinnvolle Metriken-Gruppierung

Verteidungsmetriken

Risiken und Maßnahmen

Risiko: Angriffserkennungssysteme wie eine Paketfilter-Firewalls oder Web-Application-Firewall erkennen und blockieren Angriffe, jedoch ist unbekannt wie viel e Angriffe abgewehrt werden und es wird ggf. nicht erkannt, wenn ein Angriff stattfindet.
Gegenmaßnahme: Einführung von Verteidigungs-Metriken. Verteidigungs-Metriken beinhalten das Sammeln von Informationen zu Anti-Virus- und Anti-Rootkit-Lösungen, Firewalls, Netzwerken und Angriffen. Beispielsweise kann die Anzahl der eingehenden Verbindung nach TCP/UDP-Port gemessen werden und beinh altet implizit die Anzahl der eingehenden Verbindung nach TCP/UDP-Protokoll. Durch Sammeln der Internetprotokoll-Adressen von eingehenden Verbindungen kann der geografische Quell-Standort ermittelt werden.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Sehr viel
Benötigte Ressourcen (Systeme): Wenig

Gewährleistete Sicherheitseigenschaften

Integrität: Während eines Angriffs können Informationen gewonnen werden, durch welche ein Angriff auf die Integrität von Daten abgewehrt werden kann.
Verfügbarkeit: Es können ungewollte Systemausfälle verhindert werden.
Vertraulichkeit: Während eines Angriffs können Informationen gewonnen werden, durch welche ein Angriff abgewehrt werden kann.

Sonstiges

Abhängigkeiten: Anwendungs- und System-Metriken, Visualisierte Metriken

Unter-Dimension Protokollierung

Zentrale Protokollierung

Risiken und Maßnahmen

Risiko: Protokolle sind nicht sichtbar und können bei Einbruch in ein System manipuliert werden.
Gegenmaßnahme: Protokolle werden zentral erfasst.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Sehr wenig (eine Disziplin)
Benötigte Zeit: Sehr wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch erhöhte Sichtbarkeit von Protokollen auf einem zentralen System wird die Verfügbarkeit erhöht.
Integrität: Durch sammeln von Protokollen auf einem zentralen Protokoll-System können Protokolle schwerer manipuliert werden. Durch erhöhte Sichtbarkeit kö nnen Angriffe erkannt und Maßnahmen ergriffen werden.

Sonstiges

Implementierung: rsyslog

Visualisierte Protokollierung

Risiken und Maßnahmen

Risiko: Protokolle werden mangelhaft dargestellt und können deshalb nur begrenzt ausgewertet werden. Insbesondere Entwickler können die in Dateien erfasst en Protokolle mittels ungewohnten Werkzeugen wie 'cat', 'grep' und 'less' schwer auswerten.
Gegenmaßnahme: Protokolle sind in einer Oberfläche in Echtzeit dargestellt. Dabei unterstützt eine benutzerfreundliche Bedienoberfläche inklusive Visualisieru ng von einfachen Protokoll-Metriken.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Sehr wenig (eine Disziplin)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Mittel

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch erhöhte Sichtbarkeit von Protokollen auf einem zentralen System wird die Verfügbarkeit erhöht.
Integrität: Durch erhöhte Sichtbarkeit von Protokollen während eines Angriffs können erweiterte Informationen gewonnen werden, durch welche ein Angriff auf die Integrität von Daten abgewehrt werden kann.
Vertraulichkeit: Durch erhöhte Sichtbarkeit von Protokollen während eines Angriffs können erweiterte Informationen gewonnen werden, durch welche ein Angrif f abgewehrt werden kann.

Sonstiges

Abhängigkeiten: Zentrale Protokollierung
Implementierung: ELK-Stack

Ausnahmen von Anwendungen werden erfasst

Risiken und Maßnahmen

Risiko: Treten Ausnahmen in Anwendungen auf, werden diese verzögert oder gar nicht manuell geprüft.
Gegenmaßnahme: Ausnahmen werden Instrumentiert und zentral Protokolliert. Zusätzlich wird ein Alarm gemeldet.

Nutzen und Schwere der Implementierung

Nutzen: Sehr hoch
Benötigtes Wissen: Sehr wenig (eine Disziplin)
Benötigte Zeit: Sehr wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch erhöhte Sichtbarkeit von Ausnahmen wird die Verfügbarkeit erhöht.

Sonstiges

Abhängigkeiten: Grafische Auswertung, Alarmierung

Korrelation von Sicherheits-Ereignissen

Risiken und Maßnahmen

Risiko: Sicherheits-Ereignisse werden nicht korreliert, so dass Zusammennhänge zwischen Ereignissen nicht erkannt werden.
Gegenmaßnahme: Sicherheits-Ereignisse werden korreliert. Beispielsweise erhöhte Anmeldeverusuche mit erfolgreichen Anmeldungen.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Viel (zwei Disziplinen)
Benötigte Zeit: Viel
Benötigte Ressourcen (Systeme): Viel

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch erhöhte Sichtbarkeit von Ausnahmen die Verfügbarkeit erhöht.
Integrität: Durch Korrelation von Ereignissen können Angriffe schneller erkannt und Maßnahmen ergriffen werden.

Sonstiges

Abhängigkeiten: Grafische Auswertung, Alarmierung

Dimension Infrastruktur

Unter-Dimension Infrastruktur

Interne Systeme sind einfach geschützt

Risiken und Maßnahmen

Risiko: Angreifer erhalten Zugriff auf interne Systeme ohne Authentifizierung und können Daten mitschneiden.
Gegenmaßnahme: Alle internen Systeme sind mit einfacher Authentifizierung und Verschlüsselung geschützt.

Nutzen und Schwere der Implementierung

Nutzen: Sehr hoch
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Mittel

Gewährleistete Sicherheitseigenschaften

Authentifizierung: Durch Zugriffsschutz ist sichergestellt, dass nur Autorisierte schutzwerte Informationen einsehen können.
Vertraulichkeit: Durch eine verschlüsstelte Verbindung ist sichergestellt, dass nur Autorisierte schutzwerte Informationen einsehen können.

Sonstiges

Abhängigkeiten: Definierter Verteilungs-Prozess
Implementierung: HTTP-Authentifizierung, TLS, VPN

Produktiv-Umgebung und Test-Umgebung

Risiken und Maßnahmen

Risiko: Sicherheits-Tests werden aufgrund mangelnder Test-Umgebungen nicht durchgeführt.
Gegenmaßnahme: Es existiert eine Produktiv-Umgebung und mindestens eine Test-Umgebung

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Sehr viele

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch ausreichende Tests ist das System stabil

Integrität: Durch ausreichende Tests ist sichergestellt, dass Daten nicht versehentlich bei einer Verteilung gelöscht werden.

Sonstiges

Abhängigkeiten: Definierter Verteilungs-Prozess

Anwendungen laufen in virtuellen Umgebungen

Risiken und Maßnahmen

Risiko: Durch einen Einbruch erlangt ein Angreifer Zugriff auf alle auf einem Server laufenden Anwendungen.
Gegenmaßnahme: Anwendungen laufen in virtuellen Umgebungen.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Sehr viele

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: availability

Produktionsnahe Umgebung steht Entwicklern zur Verfügung

Risiken und Maßnahmen

Risiko: Erstellung produktionsnaher Umgebungen ist schwer. Tritt eine Schwachstelle nur in der Produktionsumgebung auf, ist es schwierig diese auf einer lokal en Entwicklungsumgebung nachzuvollziehen.
Gegenmaßnahme: Durch Nutzung von einer virtuellen Umgebung und Ablage der Konfiguration im Quellcode, lässt sich eine Produktionsumgebung nachstelle n.

Nutzen und Schwere der Implementierung

Nutzen: Sehr hoch
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Mittel

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch eine produktionsnahen Umgebung können Entwickler bereits während der Entwicklung Fehler erkennen und diese korrigieren, so reduzier t das Risko durch eine Verteilung die Verfügbarkeit des System zu gefährden.

Sonstiges

Abhängigkeiten: Definierter Verteilungs-Prozess

Prüfung von Quellen eingesetzter Software

Risiken und Maßnahmen

Risiko: Genutzte Software wird ohne Prüfung der Quelle geladen und verwendet. Software kann dabei ein Paket des Betriebssystems, ein Abbild eines Betriebs systems, ein geladenes Plugin für einen Continuous Integration-Server oder eine Bibliothek in einer Anwendung sein.
Gegenmaßnahme: Jede Software-Quelle ist manuell auf Vertraulichkeit geprüft.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Integrität: Durch Prüfung von Paket-Quellen ist sichergestellt, dass System-Pakete nur auf autorisierte Daten zugreifen.
Verfügbarkeit: Durch Prüfung von Paket-Quellen ist sichergestellt, dass System-Pakete nicht die Verfügbarkeit beeinträchtigen.
Vertraulichkeit: Durch Prüfung von Paket-Quellen ist sichergestellt, dass System-Pakete nur auf autorisierte Daten zugreifen.

Kontrollierte Netzwerke für virtuelle Umgebungen

Risiken und Maßnahmen

Risiko: Virtuelle Umgebungen können auf Sockets anderer virtueller Umgebungen zugreifen, auch wenn dies nicht notwendig ist.
Gegenmaßnahme: Die Kommunikation zwischen virtuellen Umgebungen ist reguliert.

Nutzen und Schwere der Implementierung

Nutzen: Sehr hoch
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Mittel

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Regulierung verhindert die Beeinträchtigung der Verfügbarkeit von Diensten.

Vertraulichkeit: Regulierung zwischen virtuellen Umgebungen verhindert nach einem erfolgreichen Angriff auf eine virtuelle Umgebung den Zugriff auf weitere n icht autorisierte Dienste in anderen virtuellen Umgebungen.

Sonstiges

Abhängigkeiten: Definierter Verteilungs-Prozess
Implementierung: Eigene Netzwerke, Firewalls

Produktions-Artifakte sind versioniert

Risiken und Maßnahmen

Risiko: Ein vorheriges Produktionsartefakt lässt sich nicht wieder starten, wenn die Verteilung einer neuen Version nicht klappt.
Gegenmaßnahme: Alle Produktions-Artifakte sind versioniert.

Nutzen und Schwere der Implementierung

Nutzen: Sehr hoch
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Mittel

Gewährleistete Sicherheitseigenschaften

Integrität: Da ein System 'versioniert' ist, können ungewollte Änderungen identifiziert werden.
Verfügbarkeit: Durch Versionierung können alle Artifakte jeder Zeit in der selben Konfiguration auf einer Hardware erzeug werden.

Sonstiges

Abhängigkeiten: Definierter Verteilungs-Prozess

Provisionierung von Systemen

Risiken und Maßnahmen

Risiko: Manuelles Aufsetzen von System-Umgebungen kann zu fehlerhaften Konfigurationen sowie zu diskrepanzen bei redundanten Systemen führen.
Gegenmaßnahme: Mittels automatisierter Provisionierung werden System-Umgebungen aufgesetzt (Stichwort: Infrastructure as Code).

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Sehr viel
Benötigte Ressourcen (Systeme): Sehr viele

Gewährleistete Sicherheitseigenschaften

Integrität: Da ein System 'versioniert' ist, können ungewollte Änderungen identifiziert werden.
Verfügbarkeit: Durch automatische Provisionierung kann ein System jeder Zeit in der selben Konfiguration auf einer Hardware erzeug werden.

Rollen basierte Authentifizierung und Autorisierung

Risiken und Maßnahmen

Risiko: Da jeder auf einem System jede Aktion ausführen darf, ist nicht prüfbar wer eine Aktion, wie die Änderung einer Konfiguration auf dem Erzeugungs- und Verteilungsserver, ausgeführt hat.
Gegenmaßnahme: Nutzung von Rollen-basierter Authentifizierung und Autorisierung, ggf. verbunden mit einem zentralem Authentifizierungs-Server.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Vertraulichkeit: Vertrauliche Informationen über interne Systeme sind geschützt.
Integrität: Nur autorisierte Personen/Systeme können z.B. eine Verteilung anstoßen. Dadurch wird eine versehentliche Verteilung eines Software-Artefakts mit F ehlern vermieden.

Sonstiges

Abhängigkeiten: Definierter Verteilungs-Prozess, Definierter Erzeugungs-Prozess
Implementierung: Verzeichnisdienst, Plugins

Virtuelle Umgebungen sind limitiert

Risiken und Maßnahmen

Risiko: Wird eine Anwendung in einer virtuellen Umgebung angegriffen oder hat einen Defekt, kann dies zu erhöhter Ressourcen-Nutzung führen, wodurch auc h andere Anwendung auf dem gleichem Server stark beeinträchtigt werden können.
Gegenmaßnahme: Alle virtuellen Umgebungen besitzen Limitierungen für Arbeitsspeicher, Festplattendurchsatz, Festplattenplatz und Prozessoren.

Nutzen und Schwere der Implementierung

Nutzen: Mittel

Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Mittel

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Da alle Anwendungen/Prozesse limitiert sind, können sich diese nicht beziehungsweise nur gering gegenseitig beeinflussen.

Sonstiges

Abhängigkeiten: Anwendungen laufen in virtuellen Umgebungen

Betriebssystem-Aufrufe von Containern sind limitiert

Risiken und Maßnahmen

Risiko: Eine Schwachstelle von Containern sind System-Aufrufe, welche nicht den Namespace beachten
Gegenmaßnahme: Betriebssystem-Aufrufe von Anwendungen in virtuellen Umgebungen sind limitiert und auf einer Positivliste eingetragen

Nutzen und Schwere der Implementierung

Nutzen: Sehr hoch
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Mittel

Gewährleistete Sicherheitseigenschaften

Integrität: Durch eine Positiv-Liste kann die Modifizierung von Daten nach einem erfolgreichem Angriff eingeschränkt werden.
Autorisierung: Prozesse können nur definierte System-Aufrufe benutzen.
Vertraulichkeit: Prozesse können nur definierte System-Aufrufe benutzen und entsprechend nur auf autorisierte Daten zugreifen.

Sonstiges

Abhängigkeiten: Anwendungen laufen in virtuellen Umgebungen
Implementierung: seccomp, strace

Microservice-Architektur

Risiken und Maßnahmen

Risiko: Komponenten sind komplex und schwer testbar.
Gegenmaßnahme: Es ist eine Mikroservice-Architektur genutzt.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Viel (zwei Disziplinen)
Benötigte Zeit: Sehr viel
Benötigte Ressourcen (Systeme): Sehr viele

Gewährleistete Sicherheitseigenschaften

Integrität: Durch Reduktion der Komplexität und Erhöhung der Testbarkeit wird die Wahrscheinlichkeit von Schwachstellen reduziert.
Verfügbarkeit: Durch Reduktion der Komplexität und Erhöhung der Testbarkeit wird die Wahrscheinlichkeit von Schwachstellen reduziert.
Vertraulichkeit: Durch Reduktion der Komplexität und Erhöhung der Testbarkeit wird die Wahrscheinlichkeit von Schwachstellen reduziert.

Zufälliges herunterfahren von Systemen

Risiken und Maßnahmen

Risiko: Durch manuelle Änderungen an Systemen sind diese nicht auswechselbar. Die Verfügbarkeit eines redundanten Systems kann beeinträchtigt werden.
Gegenmaßnahme: Durch zufälliges Runterfahren von redundandten Systemen wird sichergestellt, dass alle Änderungen versioniert sind und die Systeme tatsä chlich hochverfügbar sind.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Sehr viel
Benötigte Ressourcen (Systeme): Sehr viele

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch zufälliges Runterfahren von redundandten Systemen wird sichergestellt, dass alle Änderungen versioniert sind und die Systeme tatsächlic h hochverfügbar sind.

Dimension Kultur und Organisation

Unter-Dimension Kultur und Organisation

Informations-Sicherheits-Ziele sind kommuniziert

Risiken und Maßnahmen

Risiko: Mitarbeiter kennen die Sicherheits-Ziele der Organisation nicht.

Gegenmaßnahme: Durch transparente Kommunikation der Sicherheitsziele durch das Management der Organisation wird der Wert von Sicherheit für das Unter nehmen klargestellt.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Sehr wenig (eine Disziplin)
Benötigte Zeit: Sehr wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Erstellung einfacher AbUser Stories

Risiken und Maßnahmen

Risiko: Bei der Erstellung von User Stories werden negative Stories aus Sicht eines Angreifers nicht eingenommen und entsprechend nicht beachtet.
Gegenmaßnahme: Es werden einfache AbUser Stories in agilen Phasen gepflegt. Beispielsweise im Product Backlog und Sprint Backlog.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gute Kommunikation wird belohnt

Risiken und Maßnahmen

Risiko: Mitarbeiter nehmen das Thema Informations-Sicherheit nicht ernst.
Gegenmaßnahme: Gute Kommunikation über das Thema Informations-Sicherheit wird belohnt. Beispielsweise mittels T-Shirts, girfcards und 'High-Fives'

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Pro Team ist ein Sicherheitsverantwortlicher definiert

Risiken und Maßnahmen

Risiko: Da kein Sicherheits-Verantwortlicher definiert ist, fühlt sich niemand im Team für Informations-Sicherheit verantwort.
Gegenmaßnahme: Pro Team ist ein Sicherheitsverantwortlicher (häufig 'Security-Champion genannt') definiert.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Durchführung von einfachen Bedrohungsanalysen

Risiken und Maßnahmen

Risiko: Bedrohungen werden nicht identifiziert.
Gegenmaßnahme: Bedrohungen werden anhand einer einfachen Risikomatrix mit Schadenpotential und Wahrscheinlichkeit des Eintritts geplegt. Im Product B acklog wird eine Bedrohungsanalyse auf Geschäftsprozess-Ebene gepflegt. Beim Sprint Planning erfolgt eine technische Bedrohungsanalyse.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Sehr wenig

Review bei jeder neuen Version

Risiken und Maßnahmen

Risiko: Eine Person hat wenig Wissen im Bereich Sicherheit und implementiert Schwachstelle oder vergisst Gegenmaßnahmen zu ergreifen.
Gegenmaßnahme: Bei jeder neuen Version eines Systems oder einer Anwendung (in der Entwicklung bei der Überführung einer Branch in den Master, in der S ystem-Administration via Konfigurations-Änderung) wird ein Review aller Änderungen durch eine zweite Person durchgeführt.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Sicherheits-Lessoned-Learned

Risiken und Maßnahmen

Risiko: Da Informations-Sicherheits-Vorfälle nicht diskutiert werden, kann aus diesn nicht gelernt werden. Entsprechend können diese öfter auftreten.
Gegenmaßnahme: Durch Sicherheits-Lessoned-Learned, bei welcher Informations-Sicherheits-Vorfälle erörtert werden, erhalten Mitarbeiter Einblick in Informations-Sicherheit und ein erhöhtes Bewusstsein für Sicherheit.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Sehr wenig

Sicherheitsprüfungen gemeinsam mit Entwicklern und System-Administratoren

Risiken und Maßnahmen

Risiko: Sicherheitsprüfungen des Quellcodes durch Externe schaffen kein Verständnis für Sicherheit bei Entwicklern und System-Administratoren.
Gegenmaßnahme: Periodische Sicherheitsprüfungen des Quellcodes, bei welchem ein Sicherheitsexperte zusammen mit Entwicklern und System-Adminstratoren Quellcode prüft, erhöhen die Sicherheit und verbreiten Wissen.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Durchführung von Team-Sicherheitsprüfungen

Risiken und Maßnahmen

Risiko: Teams sind ungenügend auf für das Thema Sicherheit sensibilisiert.
Gegenmaßnahme: Teams prüfen die Webanwendung eines anderen Teams. Dadurch wird die Sicherheit der Webanwendung, das Sicherheits-Bewustsein und das Wissen im Bereich Sicherheit erhöht. Zusätzlich können neue soziale Kontakte in einer Organisation entstehen.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Viel (zwei Disziplinen)
Benötigte Zeit: Viel
Benötigte Ressourcen (Systeme): Wenig

Durchführung von War Games

Risiken und Maßnahmen

Risiko: Notfallpläne sind nicht eingeübt und Personen können im Fall einer Bedrohung überfordert mit der Situation sein.
Gegenmaßnahme: War Games sind ein Ansatz der Gamifizierung, bei denen ein Sicherheitsexperte ein Angriffsszenario entwickelt und vorbereitet. Beispielsweise den Ausfall einer Netzwerkschnittstelle eines Datenbankservers oder ein Bruteforce-Angriff auf Benutzerkonten. Anschließend wird das Angriffsszenario auf einer produktionsnahen Umgebung ausgeführt. Das zugehörige Team, welches für die Betreuung des Systems und der Anwendung zuständig ist, hat die Aufgabe das System wieder zu reaktivieren oder den Angriff zu analysieren und Gegenmaßnahmen einzuleiten.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Viel (zwei Disziplinen)
Benötigte Zeit: Sehr viel
Benötigte Ressourcen (Systeme): Sehr viele

Durchführung von erweiterten Bedrohungsanalysen

Risiken und Maßnahmen

Risiko: Bedrohungen werden ungenügend identifiziert.
Gegenmaßnahme: Bedrohungen werden modelliert.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Viel (zwei Disziplinen)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Wenig

Ein Sicherheitsexperte pro Team

Risiken und Maßnahmen

Risiko: Security-Champions habe kein Experten-Wissen und können Sicherheit, z.B. via Stories, nicht auf hohem Niveau integrieren.
Gegenmaßnahme: Durch einen Web-Sicherheitsexperten kann Sicherheit hinreichend, z.B. via Stories oder durch einen Penetrations-Tests, in Sprints integriert werden

Nutzen und Schwere der Implementierung

Nutzen: Sehr hoch
Benötigtes Wissen: Viel (zwei Disziplinen)

Benötigte Zeit: Sehr viel
Benötigte Ressourcen (Systeme): Sehr wenig

Erstellung erweiterter AbUser Stories

Risiken und Maßnahmen

Risiko: Sicherheitsrelevante Betrachtungen werden zu Oberflächlich durchgeführt.
Gegenmaßnahme: Es werden AbUser Stories beschrieben, bei welchem erweitertes Wissen eines Sicherheits-Experten notwendig ist.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Viel (zwei Disziplinen)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Sonstiges

Abhängigkeiten: Erstellung einfacher AbUser Stories

Dimension Test und Verifizierung

Unter-Dimension Dynamische Tiefe

Einfacher Scan

Risiken und Maßnahmen

Risiko: Mangelhafte Sicherheitsprüfungen. Nach einer Verteilung können einfache Schwachstellen lange Zeit unerkannt in der Produktionsumgebung vorhande n sein.
Gegenmaßnahme: Ein einfacher Scan wird mit einem Web-Security-Scanner regelmäßig durchgeführt. Sofern die Prüfung in angemessener Zeit erfolgt, währe nd jeder Verteilung.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Syst ems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhö ht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Sy stems erhöht.

Sonstiges

Abhängigkeiten: Definierter Erzeugungs-Prozess

Abdeckung von Rollen

Risiken und Maßnahmen

Risiko: Teile der Anwendung, insbesondere welche mit Authentifizierung, sind beim Spidern mit einem Web-Security-Scanner nicht abgedeckt.
Gegenmaßnahme: Integration von Authentifizierung mit verschiedenen Rollen und Session Management

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Syst ems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhö ht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Sy stems erhöht.

Sonstiges

Abhängigkeiten: Einfacher Scan

Dynamischer Spiderdurchlauf

Risiken und Maßnahmen

Risiko: Teile der Anwendung, insbesondere welche mit vom Browser interpretierten dynamischen Inhalten wie JavaScript, sind beim Spidern mit einem Web-Security-Scanner nicht abgedeckt.
Gegenmaßnahme: Nutzung eines Spiders welcher dynamische Inhalte ausführt.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Systems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhöht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Systems erhöht.

Sonstiges

Abhängigkeiten: Abdeckung von Rollen

Abdeckung sequentieller Aktionen

Risiken und Maßnahmen

Risiko: Sequenziellen Aktionen wie Workflows sind beim Spidern mit einem Web-Security-Scanner nicht abgedeckt.
Gegenmaßnahme: Seqenzielle Aktionen werden definiert, so dass der Scanner diese in der korrekten Reihenfolge prüft.

Nutzen und Schwere der Implementierung

Nutzen: Sehr hoch
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Sehr wenig

Sonstiges

Abhängigkeiten: Abdeckung von Rollen
Implementierung: cURL

Abdeckung versteckter Pfade

Risiken und Maßnahmen

Risiko: Versteckte Pfade, wie beispielsweise APIs werden nicht abgedeckt.
Gegenmaßnahme: Versteckte Pfade werden abgedeckt.

Nutzen und Schwere der Implementierung

Nutzen: Sehr hoch
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Sonstiges

Abhängigkeiten: Abdeckung von Rollen
Implementierung: cURL

Abdeckung von nicht erkannten Eingabe-Vektoren

Risiken und Maßnahmen

Risiko: Teile der Anwendung, insbesondere welche mit speziell formatierten oder kodierten Parametern (z.B. Suchmaschinenoptimierte Parameter in der URL oder der Base64-Kodierte Parameter), werden beim Erfassen bestehender Pfade mit einem Web-Security-Scanner nicht abgedeckt.
Gegenmaßnahme: Spezielle Parameter und Kodierungen sind in eingesetzten Web-Security-Scannern definiert.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Sehr viel (drei oder mehr Disziplinen)
Benötigte Zeit: Sehr viel
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Systems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhöht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Systems erhöht.

Sonstiges

Abhängigkeiten: Abdeckung von Rollen

Abdeckung von serverseitigen Komponenten

Risiken und Maßnahmen

Risiko: Serverseitige Kommunikation, wie bei der Nutzung von Microservices, ist ungeprüft.
Gegenmaßnahme: Backend-Kommunikation ist aufgezeichnet und ist geprüft.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Viel (zwei Disziplinen)
Benötigte Zeit: Sehr viel
Benötigte Ressourcen (Systeme): Wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Syst ems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhö ht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Sy stems erhöht.

Sonstiges

Abhängigkeiten: Einfacher Scan

Abdeckungsanalyse

Risiken und Maßnahmen

Risiko: Teile der Anwendung sind beim Spidern mit einem Web-Security-Scanner nicht abgedeckt.
Gegenmaßnahme: Prüfung mittels Abdeckungsanalyse-Werkzeuge welche Teile beim Spidern beziehungsweise bei der Nutzung von eigenen Werkzeugen nic ht angesprochen werden um Justierung vornehmen zu können.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Viel (zwei Disziplinen)
Benötigte Zeit: Sehr viel
Benötigte Ressourcen (Systeme): Mittel

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Syst ems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhö ht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Sy stems erhöht.

Sonstiges

Implementierung: OWASP Code Pulse

Nutzung zusätzlicher Web-Security-Scanner

Risiken und Maßnahmen

Risiko: Ein Web-Security-Scanner ist ggf. nicht optimiert für alle genutzten Technologien. Entsprechend können Schwachstellen unerkannt bleiben.
Gegenmaßnahme: Es sind weitere spezielle Scanner eingesetzt.

Nutzen und Schwere der Implementierung

Nutzen: Sehr wenig
Benötigtes Wissen: Sehr viel (drei oder mehr Disziplinen)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Sehr viele

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Syst ems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhö ht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Sy stems erhöht.

Sonstiges

Abhängigkeiten: Abdeckung von Rollen

Unter-Dimension Statische Tiefe

Test auf serverseitige Komponenten

Risiken und Maßnahmen

Risiko: Eingesetzte serverseitige Komponten können Fehler enthalten, so dass die Informationssicherheit beeinträchtigt wird. Diese können u.a. erst nach Verte ilung der Webanwendung bekannt werden.
Gegenmaßnahme: Tests auf serverseitige Komponenten mit bekannten Schwachstellen werden regelmäßig durchgeführt, beispielsweise jede Nacht.

Nutzen und Schwere der Implementierung

Nutzen: Sehr hoch
Benötigtes Wissen: Sehr wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Integrität: Durch Tests auf Komponenten mit bekannten Schwachstellen ist die Wahrscheinlichkeit geringer, dass durch Schwachstellen in Komponenten Daten von nicht autorisierten Personen oder Systemen verändert werden können.
Verfügbarkeit: Durch Tests auf Komponenten mit bekannten Schwachstellen ist die Wahrscheinlichkeit geringer, dass Schwachstellen in Komponten ausgenutz t werden, um die Verfügbarkeit des Systems zu beeinträchtigen.
Vertraulichkeit: Durch Tests auf Komponenten mit bekannten Schwachstellen ist die Wahrscheinlichkeit geringer, dass durch Schwachstellen in Komponten Da ten von nicht autorisierten Personen oder Systemen eingesehen werden können.

Sonstiges

Abhängigkeiten: Definierter Erzeugungs-Prozess
Implementierung: OWASP Dependency Check

Statische Analyse für wichtige serverseitige Bereiche

Risiken und Maßnahmen

Risiko: Wichtige Teile der serverseitigen Webanwendung enthalten Schwachstellen in der Implementierung.
Gegenmaßnahme: Es wird eine statische Analyse, in Form von String Matching Algorithmen und/oder Datenflussanalysen, für wichtige Teile der serverseitigen Webanwendung durchgeführt. Die statische(n) Analyse(n) wird automatisiert durchgeführt und nach Möglichkeit in die Entwicklungsumgebung integriert.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Syst ems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhö ht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Sy stems erhöht.

Sonstiges

Abhängigkeiten: Definierter Erzeugungs-Prozess
Implementierung: eslint, FindSecurityBugs, jsprime

Statische Analyse für wichtige klientenseitige Bereiche

Risiken und Maßnahmen

Risiko: Wichtige Teile der klientenseitigen Webanwendung enthalten Schwachstellen in der Implementierung.
Gegenmaßnahme: Es wird eine statische Analyse, in Form von String Matching Algorithmen und/oder Datenflussanalysen, für wichtige Teile der klientenseitige n Webanwendung durchgeführt.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Syst ems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhö ht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Sy stems erhöht.

Sonstiges

Abhängigkeiten: Definierter Erzeugungs-Prozess
Implementierung: eslint, FindSecurityBugs, jsprime

Test auf klientenseitige Komponenten

Risiken und Maßnahmen

Risiko: Eingesetzte klientenseitige Komponten können Fehler enthalten, so dass die Informationssicherheit beeinträchtigt wird. Diese können u.a. erst nach Vert eilung der Webanwendung bekannt werden.
Gegenmaßnahme: Tests auf klientenseitige Komponenten mit bekannten Schwachstellen werden regelmäßig durchgeführt, beispielsweise jede Nacht.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Sehr wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Integrität: Durch Tests auf Komponenten mit bekannten Schwachstellen ist die Wahrscheinlichkeit geringer, dass durch Schwachstellen in Komponenten Daten von nicht autorisierten Personen oder Systemen verändert werden können.
Verfügbarkeit: Durch Tests auf Komponenten mit bekannten Schwachstellen ist die Wahrscheinlichkeit geringer, dass Schwachstellen in Komponten ausgenutz t werden, um die Verfügbarkeit des Systems zu beeinträchtigen.
Vertraulichkeit: Durch Tests auf Komponenten mit bekannten Schwachstellen ist die Wahrscheinlichkeit geringer, dass durch Schwachstellen in Komponten Da ten von nicht autorisierten Personen oder Systemen eingesehen werden können.

Sonstiges

Abhängigkeiten: Definierter Erzeugungs-Prozess
Implementierung: retirejs

Ausschluss von Quellcode-Dupliakten

Risiken und Maßnahmen

Risiko: Quellcode-Dupliakte können die Stabilität beeinträchtigen.
Gegenmaßnahme: Erkennung und Meldung von Duplikaten in Quellcode.

Nutzen und Schwere der Implementierung

Nutzen: Sehr wenig
Benötigtes Wissen: Sehr wenig (eine Disziplin)
Benötigte Zeit: Sehr wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Syst ems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhö ht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Sy stems erhöht.

Sonstiges

Abhängigkeiten: Definierter Erzeugungs-Prozess
Implementierung: PMD

Statische Analyse für Bibliotheken

Risiken und Maßnahmen

Risiko: Von der Webanwendung genutzt Bibliotheken enthalten unbekannte Schwachstellen in der Implementierung.
Gegenmaßnahme: Es wird eine statische Analyse, in Form von String Matching Algorithmen und/oder Datenflussanalysen, für serverseitige und klientenseitige Bibliotheken durchgeführt.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Viel
Benötigte Ressourcen (Systeme): Wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Syst ems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhö ht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Sy stems erhöht.

Sonstiges

Abhängigkeiten: Statische Analyse für wichtige klientenseitige Bereiche, Statische Analyse für wichtige serverseitige Bereiche

Statische Analyse für alle Bereiche

Risiken und Maßnahmen

Risiko: Teile der Webanwendung enthalten Schwachstellen in der Implementierung.
Gegenmaßnahme: Es wird eine statische Analyse, in Form von String Matching Algorithmen und/oder Datenflussanalysen, für alle Bereiche der serverseitigen und klientenseitigen Webanwendung durchgeführt. Externe Bibliotheken werden nicht geprüft.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Systems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhöht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Systems erhöht.

Sonstiges

Abhängigkeiten: Statische Analyse für wichtige klientenseitige Bereiche, Statische Analyse für wichtige serverseitige Bereiche
Implementierung: eslint, FindSecurityBugs, jsprime

Stilanalyse

Risiken und Maßnahmen

Risiko: Durch falsche Einrückung werden Schwachstellen eingeführt.
Gegenmaßnahme: Durch Überprüfung von Programmkonventionen (Style Guides) ist sichergestellt, dass diese eingehalten werden.

Nutzen und Schwere der Implementierung

Nutzen: Sehr wenig
Benötigtes Wissen: Sehr wenig (eine Disziplin)
Benötigte Zeit: Sehr wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Systems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhöht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Systems erhöht.

Sonstiges

Implementierung: PMD

Unter-Dimension Test-Intensität

Standardeinstellungen für Test-Intensität

Risiken und Maßnahmen

Risiko: Durch Zeitdruck und Unwissenheit werden falsche Annahmen für die Intensität getroffen.
Gegenmaßnahme: Es ist die Standardeinstellungen für die Intensität von Werkzeugen genutzt.

Nutzen und Schwere der Implementierung

Nutzen: Sehr wenig
Benötigtes Wissen: Sehr wenig (eine Disziplin)
Benötigte Zeit: Sehr wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Deaktivierung unnötiger Prüfungen

Risiken und Maßnahmen

Risiko: Prüfungen nehmen stark Ressourcen in Anspruch.
Gegenmaßnahme: Unnötige Prüfungen sind deaktiviert. Benutzt eine Webanwendung die Mongo-Datenbank, kann ggf. auf eine SQL-Injection-Prüfung verzichtet werden.

Nutzen und Schwere der Implementierung

Nutzen: Sehr wenig
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Sehr wenig

Angepasste Test-Intensität

Risiken und Maßnahmen

Risiko: Scans führen zu viele oder zu wenig Scans für unterschiedliche Schwachstellen durch.
Gegenmaßnahme: Die Test-Intensität ist angepasst.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Mittel

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Systems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhöht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Systems erhöht.

Test-intensität ist hoch eingestellt

Risiken und Maßnahmen

Risiko: Durch zu niedriger Scan-Intensität werden Schwachstellen nicht aufgedeckt.
Gegenmaßnahme: Möglichst alle Schwachstellen-Tests werden periodisch mit hoher Test-Intensität durchgeführt.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Sehr viele

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Systems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhöht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Systems erhöht.

Unter-Dimension Konsolidierung

Behandlung kritischer Alarme

Risiken und Maßnahmen

Risiko: Mangelhafte Auswertung der erfolgten Sicherheitsprüfungen.
Gegenmaßnahme: Akzeptanzkretieren für gefundene Schwachstellen sind definiert. Empfehlung ist hier nur als kritisch eingestufte Schwachstellen/Alarme zu behandeln. Sofern die genutzten Werkzeuge es anbieten, kann auch das Vertrauen (Englisch Confidence) mit zur Einstufung herangezogen werden. Entsprechend wird die Erzeugung markiert oder gestoppt, wenn Schwachstellen mit einer Sicherheitseinstufung über der definierten Akzeptanz gefunden werden.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Sehr wenig (eine Disziplin)
Benötigte Zeit: Sehr wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Systems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhöht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Systems erhöht.

Einfache Falsch-Positiv-Behandlung

Risiken und Maßnahmen

Risiko: Aufgrund von mehrfach falsch positiv gemeldeten Schwachstellen werden neue Warnungen ignoriert.

Gegenmaßnahme: Falsch positiv gemeldete Schwachstellen werden, auf Basis von Werkzeugen, markiert und bei der nächsten Prüfung nicht mehr gemeldet.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Sehr wenig (eine Disziplin)
Benötigte Zeit: Sehr wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Sonstiges

Alarme sind einfach visualisiert

Risiken und Maßnahmen

Risiko: Es ist unklar, wie viele Alarme im Monat entstehen.
Gegenmaßnahme: Alarme werden einach visualisiert um einfache Trendanalyse durchführen zu können.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Sonstiges

Abhängigkeiten: Visualisierte Metriken

Alarme adressieren Teams

Risiken und Maßnahmen

Risiko: Jedes Team muss jeden Alarm prüfen, so kann Frust entstehen.
Gegenmaßnahme: Alarme werden Teams zugewiesen, so dass keine Ressourcen verschwendet werden.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch freie Ressourcen kann die Verfügbarkeit von Informationen im gesamten Systems erhöht werden.
Integrität: Durch freie Ressourcen kann die Integrität von Informationen im gesamten Systems erhöht werden.
Vertraulichkeit: Durch freie Ressourcen kann die Vertraulichkeit von Informationen im gesamten Systems erhöht werden.

Sonstiges

Implementierung: Bei SAST: Serverseitige/klientenseitige Teams können einfach erfasst werden. Bei Mikroservice-Architektur können einzelne Mikroservices i. d.R. Teams zugewiesen werden. Bei DAST: Schwachstellen sind klassifiziert und können serverseitigen und klientenseitigen Teams zugewiesen werden.

Behandlung von mittelschweren Alarmen

Risiken und Maßnahmen

Risiko: Mittelschwere Alarme werden nicht beachtet.
Gegenmaßnahme: Akzeptanzkretieren für gefundene Schwachstellen sind definiert. Empfehlung ist mittelschwere Meldungen ebenfalls zu behandeln.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Systems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhöht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Systems erhöht.

Sonstiges

Kommentar: Falsch Positiv-Sortierung ist Zeitaufwendig.

Aggregation von Alarmen

Risiken und Maßnahmen

Risiko: Die Wartung von Alarmen und falsch Positiven in unterschiedlichen Werkzeugen und Definitionen erhöht den Aufwand stark.

Gegenmaßnahme: Aggregation von Alarmen, dabei werden auch doppelte Alarme nach Möglichkeit zusammengeführt.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Mittel
Benötigte Ressourcen (Systeme): Wenig

Alarme sind erweitert visualisiert

Risiken und Maßnahmen

Risiko: Aufgrund der einfachen Visualisierung von Alarmen sind zusammenhänge nicht auf den ersten Blick erkennbar.
Gegenmaßnahme: Alarme werden als Metrik erfasst visualisiert.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Viel
Benötigte Ressourcen (Systeme): Sehr wenig

Sonstiges

Abhängigkeiten: Visualisierte Metriken

Behandlung von allen Alarmen

Risiken und Maßnahmen

Risiko: Alarme mit Schwere 'Einfach' werden nicht beachtet.
Gegenmaßnahme: Akzeptanzkretieren für gefundene Schwachstellen sind definiert. Empfehlung ist Meldungen mit der Schwere 'Einfach' ebenfalls zu behandeln.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Systems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhöht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Systems erhöht.

Reproduzierbare Alarme

Risiken und Maßnahmen

Risiko: Alarme können von Entwicklern / System-Adminstratoren ggf. nur schwer nachvollzogen werden.
Gegenmaßnahme: Alarme enthalten den Ablauf der Aktionen um die gemeldete Schwachstelle einfacher reproduzieren zu können.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch nachhaltige Behebung von Schwachstellen kann die Verfügbarkeit von Informationen im gesamten Systems erhöht werden.
Integrität: Durch nachhaltige Behebung von Schwachstellen kann die Integrität von Informationen im gesamten Systems erhöht werden.
Vertraulichkeit: Durch nachhaltige Behebung von Schwachstellen kann die Vertraulichkeit von Informationen im gesamten Systems erhöht werden.

Sonstiges

Implementierung: ZEST

Unter-Dimension Anwendungstest

Modultests mit Sicherheitsbezug

Risiken und Maßnahmen

Risiko: Schwachstellen sind unbeachbsichtigt Implementiert.
Gegenmaßnahme: Integration von sicherheitsrelevanten Modultests für geschäftskritische Bereiche. Dadurch können Schwachstellen wie fehlende Authentifizierung erkannt werden.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Viel
Benötigte Ressourcen (Systeme): Wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Systems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhöht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Systems erhöht.

Sonstiges

Implementierung: JUnit
Kommentar: Die Integration von Modultests findet schon während der Entwicklung statt, es wird auf Schwachstellen in Sub-Routinen, Funktionen, Module, Bibliotheken usw. geprüft.

Integrationstests mit Sicherheitsbezug

Risiken und Maßnahmen

Risiko: In der Anwendung sind grundlegende Fehler bei der Benutzung eines Frameworks möglich, ohne das diese erkannt werden.
Gegenmaßnahme: Implementierung grundlegender Sicherheitstests als und Integrationstests. Beispielsweise kann die Authentifizierung und Autorisierung (Zugriffskontrolle) geprüft werden.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Viel
Benötigte Ressourcen (Systeme): Wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Systems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhöht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Systems erhöht.

Sonstiges

Implementierung: HttpUnit

Akzeptanztests mit Sicherheitsbezug

Risiken und Maßnahmen

Risiko: In der Anwendung sind grundlegende Fehler bei der Benutzung eines Frameworks möglich, ohne das diese erkannt werden.
Gegenmaßnahme: Implementierung grundlegender Sicherheitstests als Akzeptanztests. Beispielsweise können sicherheitsrelevante Funktionen in JavaScript geprüft werden.

Nutzen und Schwere der Implementierung

Nutzen: Sehr wenig
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Viel
Benötigte Ressourcen (Systeme): Wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Systems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhöht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Systems erhöht.

Sonstiges

Implementierung: HtmlUnit, Selenium

Post-Verteilungs-Prüfung

Risiken und Maßnahmen

Risiko: Durch eine Verteilung auf die Produktionsumgebung können Mikroservices gestört sein, z.B. wenn die Datenbank nicht erreicht werden kann.
Gegenmaßnahme: Integrationstests prüfen die Produktionsumgebung um sicher zu stellen, dass Funktionen, z.B. bereitgestellt durch Mikroservices oder externe Dienste, erreichbar sind.

Nutzen und Schwere der Implementierung

Nutzen: Wenig
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Systems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhöht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Systems erhöht.

Sonstiges

Abhängigkeiten: Definierter Verteilungs-Prozess

Sehr hoch abdeckende Komponenten-, Integrations- und Akzeptanztests mit Sicherheitsbezug

Risiken und Maßnahmen

Risiko: Es sind nicht alle Teile der Anwendung mit Sicherheitsprüfungen versehen.
Gegenmaßnahme: Implementierung grundlegender Sicherheitstests als Integrations- und/oder Akzeptanztests für alle Teile (auch Bibliotheken) der Anwendung.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Sehr viel (drei oder mehr Disziplinen)
Benötigte Zeit: Sehr viel
Benötigte Ressourcen (Systeme): Mittel

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Systems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhöht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Systems erhöht.

Unter-Dimension Infrastrukturtest

Test auf System-Aktualisierungen

Risiken und Maßnahmen

Risiko: Das Betriebssystem oder seine Dienste enthalten bekannte Schwachstellen.
Gegenmaßnahme: Prüfung auf Aktualisierungen und bei veralteter Software Meldung an einen Verantwortlichen, welcher die Patches einspielt.

Nutzen und Schwere der Implementierung

Nutzen: Sehr hoch
Benötigtes Wissen: Sehr wenig (eine Disziplin)
Benötigte Zeit: Sehr wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Integrität: Durch Prüfung und Einspielen von System-Aktualisierungen ist die Wahrscheinlichkeit, dass System-Komponenten die Integrität von Informationen beeinträchtigen verringert.
Verfügbarkeit: Durch Prüfung und Einspielen von System-Aktualisierungen ist die Wahrscheinlichkeit, dass System-Komponenten die Verfügbarkeit beeinträchtigen verringert.
Vertraulichkeit: Durch Prüfung und Einspielen von System-Aktualisierungen ist die Wahrscheinlichkeit, dass System-Komponenten vertrauliche Informationen preisgeben verringert.

Prüfung der Konfiguration von virtuellen Umgebungen

Risiken und Maßnahmen

Risiko: Virtuelle Umgebungen birgen die Gefahr sicherheitskritisch Konfiguriert zu sein.
Gegenmaßnahme: Mit Hilfe von Werkzeugen wird die Konfiguration von virtuellen Umgebungen geprüft.

Nutzen und Schwere der Implementierung

Nutzen: Hoch
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Systems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhöht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Systems erhöht.

Sonstiges

Implementierung: Docker Bench for Security, Docker Security Scan, openVAS

Test auf schwache Passwörter

Risiken und Maßnahmen

Risiko: Mitarbeiterkonten und priviligierte Benutzerkonten sind mit schwachen Passwörtern geschützt.
Gegenmaßnahme: Automatische BruteForce-Angriffe auf Benutzer-Konten von Mitarbeitern sowie Standard-Konten wie 'administrator'.

Nutzen und Schwere der Implementierung

Nutzen: Sehr wenig
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Sehr wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Ändern von schwachen Passwörtern kann die Verfügbarkeit von Informationen im gesamten Systems erhöht werden.
Integrität: Durch Erkennung und Ändern von schwachen Passwörtern kann die Integrität von Informationen im gesamten Systems erhöht werden.
Vertraulichkeit: Durch Erkennung und Ändern von schwachen Passwörtern kann die Vertraulichkeit von Informationen im gesamten Systems erhöht werden.

Sonstiges

Implementierung: HTC Hydra

Erweiterte System-Prüfung

Risiken und Maßnahmen

Risiko: Systeme wie Firewalls können nach einer Anpassung sicherheitskritisch konfiguriert sein.
Gegenmaßnahme: Automatische Prüfung von Infrastruktur-Systemen wie Firewalls.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Systems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhöht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Systems erhöht.

Lasttests

Risiken und Maßnahmen

Risiko: Es ist unbekannt wie viele Anfragen das System bedienen kann und wie sich das System bei vielen Anfragen verhält.
Gegenmaßnahme: Last-Tests werden periodisch ausgeführt.

Nutzen und Schwere der Implementierung

Nutzen: Mittel
Benötigtes Wissen: Mittel (zwei Disziplinen)
Benötigte Zeit: Wenig
Benötigte Ressourcen (Systeme): Sehr viele

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Flaschenhälsen ist die Verfügbarkeit von Informationen im gesamten Systems erhöht.

Visualisierung von System-Updates

Risiken und Maßnahmen

Risiko: Kritische System-Updates werden nach zu langer Zeit eingespielt.
Gegenmaßnahme: Durch Visualisierung wird die Kritikalität von Systemupdates deutlich und deshalb schneller eingespielt.

Nutzen und Schwere der Implementierung

Nutzen: Wenig

Benötigtes Wissen: Wenig (eine Disziplin)
Benötigte Zeit: Sehr wenig
Benötigte Ressourcen (Systeme): Sehr wenig

Gewährleistete Sicherheitseigenschaften

Verfügbarkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Verfügbarkeit von Informationen im gesamten Syst ems erhöht.
Integrität: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Integrität von Informationen im gesamten Systems erhö ht.
Vertraulichkeit: Durch Erkennung und Behebung von Schwachstellen bevor diese in Produktion gehen ist die Vertraulichkeit von Informationen im gesamten Sy stems erhöht.

K.3 Bildschirmfoto der Abhängigkeiten „Statische Tiefe“

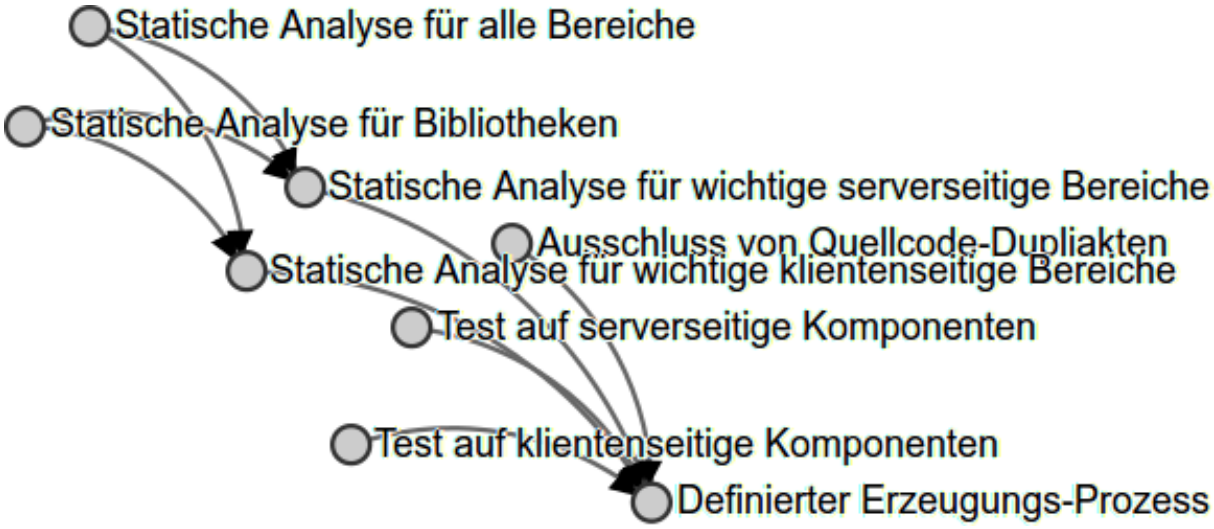


Abbildung K.1: Bildschirmfoto der Abhängigkeiten der Dimension „Statische Tiefe“

Anhang L

Implementierung des Modells

L.1 Implementierung von Ebene 1

L.1.1 Erzeugung und Verteilung

Erzeugung: Definierter Erzeugungsprozess

Implementierungs-Dauer: 4 Tage
Die Definition des Erzeugungs- und Verteilungsprozesses erfolgt in dem *CI*-System *Jenkin*s. *Jenkins* läuft in einem *Docker*-Container und ist erreichbar unter http://deployment.fhunii.com.
Die Erzeugung findet dabei auf dem *Jenkins-Slave stage.example.com* statt. Der Erzeugungsprozess auf dem Stage-System besteht aus:

1. Abholen des Quellcodes von dem Versionskontrollsystem.
2. Starten der MySQL-Datenbank, sofern diese nicht gestartet ist.
3. Ausführen von `./activator clean` um das Projekt vorzubereiten.
4. Ausführen von `./activator docker:publishLocal` um das Projekt zu erzeugen und in ein *Docker*-Abbild zu legen.
5. Auslesen der Internetprotokoll-Adresse der Datenbank sowie die Container-ID des gestarteten Containers der Anwendung, sofern gestartet.
6. Stoppen des gestarteten Containers der Webanwendung.
7. Zurücksetzen der Datenbank.
8. Vorbereiten der Parameter, u.a. die Internetprotokoll-Adresse der Datenbank, zum Starten des Containers der Webanwendung.
9. Starten des Containers mit den vorbereiteten Parametern.
10. Markieren des erzeugten Abbilds mit *latest* sowie der Auftragsnummer in *Jenkins* und Senden des Abbilds an die interne Registrierung.

Die von *Jenkins* ausgeführten Schritte in Form von eines *Bash*-Skripts sind im Listing im Anhang L.3 auf Seite 198 aufgeführt.

Verteilung: Definierter Verteilungsprozess

Implementierungs-Dauer: 4 Tage

Auf dem Produktionssystem *www.example.com* ist ebenfalls ein *Jenkins*-Slave eingerichtet, welcher das Abholen und das Starten eines neuen Abbilds durch „Knopfdruck“ im *Jenkins* veranlasst. Die Verteilung auf das Produktionssystem besteht aus:

1. Abholen des Abbilds mit Markierung *latest* von der internen Registrierung.
2. Stoppen des Containers der Webanwendung.
3. Backup der Datenbank
4. Auslesen der Internetprotokoll-Adresse der Datenbank.
5. Vorbereiten der Parameter, u.a. die Internetprotokoll-Adresse der Datenbank, zum Starten des Containers.
6. Starten des Containers der Webanwendung mit den vorbereiteten Parametern.
7. Senden des Abbilds an die interne Registrierung.

Nachteil des Prozesses ist, dass die Markierung *latest* nicht eindeutig ist. So kann eine Erzeugung ausgeführt werden, diese manuell getestet und als Stabil definiert werden, eine weitere von einer anderen Person ausgeführt werden und die Verteilung gestartet werden. Durch Absprache der Geschäftsführer vor jeder Verteilung auf das Produktionssystem kann der Fall einer nicht stabilen Verteilung ausgeschlossen werden.

Beim Zeitpunkt des Backups liegt ein Konflikt zwischen Integrität und Verfügbarkeit vor. Wird das Backup vorm Stoppen der Webanwendung durchgeführt, ist die Verfügbarkeit erhöht aber die Integrität nicht gewährleistet. Wird es anders herum durchgeführt, ist die Integrität gesichert, die Verfügbarkeit jedoch vermindert. Es wurde sich für die Sicherung der Integrität entschieden.

L.1.2 Informationsgewinnung

Überwachung und Metrik: Einfache Anwendungs- und System-Metriken

Implementierungs-Dauer: 5 Tage

Um Anwendungsmetriken zu erfassen, wird die Bibliothek *kamon.io* in die Webanwendung integriert. *kamon.io* erfasst von sich aus Java-Metriken wie die Anzahl der *Threads* oder die Nutzung des *Heaps*. Die Metriken werden an den in der Konfiguration hinterlegten *statsD*-Server gesendet. Der Server ist mittels eines von *kamon.io* bereitgestellten *Docker*-Containers [35] mit *statsD*, *Graphite*, *Grafana* und einem *Dashboard* aufgesetzt. Der *Docker*-Container ist damit entgegen die Empfehlung, eine Aufgabe pro Container, erstellt. Weiterhin werden alle Prozesse privilegierter als *root*-Benutzer ausgeführt, was die Sicherheit beeinträchtigt. Das Risiko wurde dem Projekt gemeldet[1].

Es wird viel Festplattendurchsatz erwartet und deshalb wird der *Docker*-Container auf dem Wirtsystem *luke.example.com* gestartet.

Weiterhin werden erfolgreiche Anmeldungen und nicht erfolgreiche Anmeldungen in der Anwendung gezählt und an *Graphite* übergeben.

[1] Siehe https://github.com/kamon-io/docker-grafana-graphite/issues/66.

System-Metriken werden über *collectd* erfasst. Zur Erfassung *Docker* spezifischer Werte wie für Netzwerkdurchsatz, Arbeitsspeicher-Nutzung und CPU-Nutzung wird das *docker-collectd-plugin* genutzt.

Protokollierung: Zentrale Protokollierung

Implementierungs-Dauer: 1 Tag
Die zentrale Protokollierung erfolgt mittels *rsyslogd*. Alle virtuellen Umgebungen, abgesehen von *Docker* Containern, erhalten den Konfigurations-Eintrag `*.* @@luke.example.com:514` in der Datei */etc/rsyslog.d/10-rsyslog.conf*, um zentral alle Protokolle zu speichern.

L.1.3 Infrastruktur

Interne Systeme sind einfach geschützt

Implementierungs-Dauer: 1 Tag
Alle internen Systeme wie *Jenkins* (https://deployment.fhunii.com), der *Docker*-Container von *kamon.io* (https://stats.fhunii.com) und *stage.example.com* sind über eine *HTTP*-Authentifizierung mittels *nginx* geschützt. Weiter ist der Zugriff nur über *HTTPs* möglich. Während die Webanwendung https://www.fhunii.com TLSv1 zur Benutzerfreundlichkeit unterstützt wird, unterstützen interne Systeme nur TLSv1.1 und TLSv1.2.
Wird *Grafana* (von *kamon.io*) in Kombination mit *HTTP*-Authentifizierung genutzt, so muss der Benutzer zur *HTTP*-Authentifizierung ebenfalls in *Grafana* existieren. Sonst wird die Fehlermeldung *{"message":"Basic auth failed"}* angezeigt.

Produktiv-Umgebung und Test-Umgebung

Implementierungs-Dauer: 1 Tag
Eine produktionsnahe Umgebung für Tests ist auf der virtuellen Maschine *stage.example.com* eingerichtet. Hier werden Anfragen für https://stage.fhunii.de bedient. Die .de-Domain ist notwendig um eine produktionsnahe Test-Umgebung durch Platzhalter für *Subdomains* zu garantieren.

L.1.4 Kultur und Organisation

Informations-Sicherheits-Ziele sind kommuniziert

Implementierungs-Dauer: 1 Tag
Die Geschäftsführer der anoStartup haben gemeinsam die Sicherheitsziele besprochen. Dabei wurde erkannt, dass Sicherheit eine große Rolle für das Unternehmen darstellt. Es ist festgelegt, konform zum „Sicherheit von Webanwendungen-Maßnahmenkatalog und Best Practices“ [214] vom BSI zu entwickeln und jedem neuem Mitarbeiter im Bereich IT ein Exemplar auszuhändigen.

L.1.5 Test und Verifizierung

Dynamische Tiefe: Einfacher Scan

Implementierungs-Dauer: 1 Tag

Das *Zaproxy Plugin* [55] für *Jenkins* wird zum Starten des *Spiders* und des Scanners genutzt. Zunächst konnte auf die zu prüfende Anwendung nicht zugegriffen werden, da die *HTTP*-Authentifizierung vom System *stage.example.com* dies nicht zugelassen hat. Der Zugriff auf die Anwendung aus dem internen Netzwerk wurde erlaubt.

Statische Tiefe: Test auf serverseitige Komponenten

Implementierungs-Dauer: 1 Tag

Um die Webanwendung zu erstellen und alle Abhängigkeiten aus dem lokalem Verzeichnis *~/ivy2* in das aktuelle Arbeitsverzeichns zu überführen, wird `./activator docker:stage` ausgeführt und anschließend mittels Starten des *Docker*-Abbilds *anoStartup/docker-owasp-dependency-check* die Abhängigkeiten im Projekt getestet. Das *Docker*-Abbild *anoStartup/docker-owasp-dependency-check* wird automatisch via *hub.docker.com* neu erzeugt, wenn in dem zugehörigem *git*-Projekt[2] eine Änderung erfolgt. Ein Bildschirmfoto der Konfiguration in dem *Jenkins*-Auftrag ist im Anhang L.1 abgelegt.

Ursprünglich war geplant, das *OWASP Dependency-Check Plugin* [186] für *Jenkins* der Version 1.4.0 zu nutzen. Allerdings hat dieses nicht Aktualisierungen der *NVD* nicht abgeholt. Entsprechend konnten zunächst keine Komponenten mit Schwachstellen gefunden werden.

Es wird ein textueller Abhängigkeitsbaum mittels *sbt-dependency-graph* durch den *Bash*-Befehl `sbt dependencyTree` erstellt um die ursprünglich eingebundenen Bibliotheken zu ermitteln. So ist u.a. die Bibliothek *deadbolt* von Version 2.4.3 auf 2.5.0 aktualisiert wurden.

Die bekannten Risiken wurden so von insgesamt 16 kritischen Risiken und 87 mit Kritikalität „Mittel“ auf 11 kritische Risiken und 21 mit Kritikalität „Mittel“ reduziert. Es kann eine Bibliothek durch transitive Abhängigkeiten mehrere Risiken enthalten.

Es wurde festgelegt, dass die 11 kritischen Bibliotheken im Anschluss an diese Arbeit nach Möglichkeit zu aktualisieren.

Test-Intensität: Standardeinstellungen für Intensität

Implementierungs-Dauer: /

Hier müssen keine Anpassungen vorgenommen werden.

Konsolidierung: Behandlung kritischer Alarme

Implementierungs-Dauer: 1 Tag

Zap speichert gefundene Risiken im Format *Extensible Markup Language* (kurz *XML*) welche anschließend auf der Konsole in *Jenkins* ausgegeben werden. Die Ausgabe enthält für jede Schwachstelle u.a. das Attribut *riskcode* mit einem numerischem Wert. Dabei entspricht 0 einer Information, 1 der Kritikalität Niedrig, 2 Mittel und 3 Hoch. Die Ausgabe wird nach „<riskcode>“ gefiltert und auf Kritikalität größer 2 getestet. Ist dies der Fall, wird *Found Defect: <Kritikalität>* ausgegeben. Das *Log Parser Plugin* [22] testet anschließend, ob *Found Defect* ausgegeben wurde. Wurde es mindestens einmal ausgegeben, wird der Auftrag mit einem *Error* markiert. Das *Bash*-Skript, welches beim Durchlauf von dem *OWASP-Zap*-Auftrag jeden Tag um 07:15 Uhr ausgeführt wird, ist unter L.1 abgelegt.

[2] Siehe https://github.com/wurstbrot/Docker-OWASP-Dependency-Check.

Abbildung L.1: Bildschirmfoto der Einrichtung des *OWASP Dependency Checks*

Konsolidierung: Einfache Falsch-Positiv-Behandlung

Implementierungs-Dauer: 1 Tag

In der Oberfläche von *Zap* kann durch Doppelklick auf ein gefundenes Risiko das Vertrauen als falsch Positiver eingestuft werden. Damit *Jenkins* auf die falsch Positiven von *Zap* zugreifen kann, wird die Sitzung von *Zap* in dem Ordner *security* in der Versionskontrolle des Projekts gespeichert und beim Testen von *zaproxy* geladen.

Im *OWASP Dependency-Check* kann eine „Susspression"-Datei zur Markierung falsch Positiver im Format *XML* angegeben werden. Die Datei *dependency-check-suppression.xml* ist ebenfalls im Ordner *security* abgelegt.

Listing L.1 Implementierung der Schwachstellen-Zählung für *Zap* in *Jenkins*

```
for SERIOUS_DEFECT in $(cat /home/jenkins/workspace/OWASP-ZAP/zapReport/
    report_${BUILD_NUMBER}.xml | grep "<riskcode>"); do
        SERIOUS_DEFECT_SERVITY=$(echo $SERIOUS_DEFECT | sed 's/<riskcode
           >//' | sed 's/<\/riskcode>//');
    if [ $SERIOUS_DEFECT_SERVITY -gt 2 ]; then
                echo "Found Defect: $SERIOUS_DEFECT_SERVITY" > /tmp/
                   deletemeLogparser # The command itself is printed out
            rm /tmp/deletemeLogparser
    fi done

if [ ! -e rules.txt ]; then
        echo "error /Found Defect/" >  rules.txt
fi
```

Anwendungstest: Modultests mit Sicherheitsbezug

Implementierungs-Dauer: 3 Tag

Die Anwendung bietet eine Schnittstelle, von welcher ein Benutzername und ein Passwort gefordert wird. Beispielhaft wurde die Klasse `ApiLoginUnitTest` erstellt, welche mittels der Methoden `loginWithExistingUser()` und `loginWithNonExistingUser()` testet, dass sich bestehende Benutzer erfolgreich anmelden können und Zugriff nicht ohne hinterlegten Benutzer erfolgen kann. Der Quellcode ist in der öffentlichen Version nicht verfügbar.

Alle Tests werden auf *Jenkins* mittels `./activator test` ausgeführt.

Infrastrukturtest: Test auf System-Aktualisierungen

Implementierungs-Dauer: 1 Tag

Über Aktualisierungen für Betriebssysteme wird über *apticron* [29] informiert. Werkzeuge wie *watchtower* ermöglichen, auf Aktualisierungen von *Docker*-Abbildern zu testen, das Abbild herunterzuladen und automatisch den Container neu zu starten. Dies birgt jedoch die Gefahr, dass eine Aktualisierung, insbesondere wenn persistent gespeicherte Daten aktualisiert werden, nicht klappt und der Dienst nicht mehr verfügbar ist. Entsprechend wurde ein Skript (siehe Listing L.2) erstellt, um nächtlich mittels *cronjob* eine Information via E-Mail zu versenden. Das Skript ist auf allen Umgebungen mit *Docker* installiert und mittels *cronjob* aufgerufen.

In dem Skript wird zunächst für jedes Abbild ein `pull` ausgeführt und die Ausgabe, gefiltert auf „Downloaded newer image“, in */tmp/dockerupdate* umgeleitet. Anschließend wird getestet, ob mindestens eine Zeile in */tmp/dockerupdate* vorhanden ist. Ist das der Fall, wird eine E-Mail mit der Information versandt. Anschließend kann manuell eine Aktualisierung erfolgen. Während *apticron* jeden Tag eine E-Mail mit allen zur Verfügung stehenden Aktualisierungen sendet, sendet das Skript die E-Mail nur einmal, da nach dem `pull` ein aktuelles Abbild vorliegt.

Listing L.2 Skript zur Erzeugung von E-Mails bei aktualisierten Abbildern für *Docker*-Container

```
#!/bin/bash
MAILTO="systems@example.com"
for i in $(docker images | awk '(NR>1) && ($2!~/none/) {print $1":"$2}')
    ; do
        docker pull $i | grep "Downloaded newer image" >> /tmp/
           dockerupdate;
done

if [ $(cat /tmp/dockerupdate | wc -l ) -gt 0 ]; then
        mail -s "Es ist mindestens ein neues Docker-Abbild auf $HOSTNAME
            vorhanden" $MAILTO < /tmp/dockerupdate fi
rm /tmp/dockerupdate
```

L.2 Implementierung von Ebene 2

L.2.1 Erzeugung und Verteilung

Erzeugung: Regelmäßiger Test

Implementierungs-Dauer: 1 Tag

Es wird ein regelmäßiger Test in *Jenkins* jede Nacht ausgeführt. Die Sicherheitstests sind jeweils als einzelne Aufgabe in *Jenkins* definiert mittels *Multijob Plugin [180]* gruppiert (siehe Abb. L.2). Jeder statische Tests holt sich den Quellcode der Webanwendung, erzeugt diese und testet anschließend. Zukünftig sollte die Webanwendung nur einmalig erzeugt und anschließend von den unterschiedlichen Tests genutzt werden.

S	W	Job	Last Success	Last Failure	Last Duration	Console
		Sicherheits-Tests	N/A	47 min	24 min	
		Statische Sicherheits-Tests				
		OWASP-Dependency-Check	N/A	47 min	8 min 27 sec	
		FindSecurityBugs	N/A	N/A	8 min 51 sec	
		Retire.JS	N/A	N/A	N/A	
		ESlint	N/A	N/A	N/A	
		Dynamische Sicherheits-Tests				
		OWASP-ZAP	N/A	N/A	39 sec	

Abbildung L.2: Bildschirmfoto der regelmäßigen Tests mittels *Multijob Plugin*

Verteilung: Austausch von Konfigurationsparametern

Implementierungs-Dauer: 1 Tag

Es werden beim Starten des *Docker*-Containers der Anwendung umgebungsspezifische Konfigurationsparameter gesetzt (siehe Listing L.3 auf der nächsten Seite).

Für die Produktionsumgebung werden u.a. die Parameter „db.default.password“ mit dem Passwort der Produktionsdatenbank, „anoStartup.domain“ mit dem Wert „example.com“ und „anoStartup.subdomain“ mit dem Wert „www“. Das Passwort der Datenbank wird über das *Mask Passwords Plugin* gesetzt, so

Listing L.3 Implementierung der Erzeugung und Verteilung in *Jenkins*

```
# Backup database
DUMP_CMD='exec mysqldump -h"$MYSQL_PORT_3306_TCP_ADDR" -P"
    $MYSQL_PORT_3306_TCP_PORT" -uroot -p"$MYSQL_ENV_MYSQL_ROOT_PASSWORD"
    --all-databases'
docker run -i --link anoStartup-mysql:mysql --rm mysql sh -c $DUMP_CMD >
     ~/mysql/backups/backup-$(date +%Y-%m-%d-%T).sql 2>&1 | grep -v
    Warning

docker pull registry.example.com:5000/anoStartup-multi

if [ "$CONTAINER_ID" != "" ];then
        docker stop -t "40" $CONTAINER_ID
fi

# Prepare Docker parameters
CONTAINER_PARAMETER="-Dconfig.file=/opt/docker/conf/application.prod.
    conf -DanoStartup.domain=example.com db.default.password=\"
    $dbpassword\" -DanoStartup.subdomain=www -Dplay.evolutions.db.default
    .autoApply=true -Dplay.evolutions.db.default.autoApplyDowns=false"
CONTAINER_ID=$(docker ps | grep anoStartup-multi | awk '{print $1}')

MYSQL_CONTAINER_ID=$(docker ps | grep anoStartup-mysql | awk '{print $1
    }')
MYSQL_IP=$(docker inspect --format '{{ .NetworkSettings.IPAddress }}'
    $MYSQL_CONTAINER_ID)
DB_URL_STRING="jdbc:mysql://$MYSQL_IP/anoStartupeditor"
DB_DOCKER_PAREMTER="db.default.url=$DB_URL_STRING"
CONTAINER_PARAMETER="$CONTAINER_PARAMETER -D$DB_DOCKER_PAREMTER"

MOUNTS="-v /etc/localtime:/etc/localtime:ro -v /var/www/
    anoStartupuploads:/var/www/anoStartupuploads"
DOCKER_PARAMETER="--restart=always -d $MOUNTS -p 9000:9000 --name www.
    example.com"
DOCKER_CMD="$DOCKER_PARAMETER registry.example.com:5000/anoStartup-multi
     $CONTAINER_PARAMETER"

# Start docker
docker run $DOCKER_CMD
```

dass es in dem Ausführungsprotokoll nicht zu sehen ist.

Durch Nutzung von „anoStartup.domain“ und „anoStartup.subdomain“ kann ein weiteres System mit einer anderen Domain, z.B. für Tests, einfach aufgesetzt werden.

Verteilung: Backup vor Verteilung

Implementierungs-Dauer: 1 Tag

Vor der Verteilung wird ein Backup der Produktionsdatenbank angelegt (siehe Listing L.3 im Anhang). Auf das Backup der durch Benutzer hochgeladenen Dateien wird verzichtet.

L.2.2 Informationsgewinnung

Überwachung und Metrik: Visualisierte Metriken

Implementierungs-Dauer: 1 Tag

Die von *collectd* gesammelten System-Informationen werden mittels *statsD*-Plugin von *collectd* an den *Graphite*-Dienst von *kamon.io* gesendet. Anschließend werden die Metriken in einem *Dashboard* angezeigt. Das *Dashboard* für virtuelle Maschinen ist in Abb. L.3 dargestellt.

In der öffentlichen Version ist das Bildschirmfoto nicht verfügbar.

Abbildung L.3: Bildschirmfoto der Übersicht für virtuelle Maschinen in *Grafana*

Überwachung und Metrik: Alarmierung

Implementierungs-Dauer: 4 Tage

Die Alarmierung für die in *Graphite* gehaltenen Daten erfolgt mittels *seyren*. Das Skript zum Starten von *seyren* auf *luke.example.com* ist in Listing L.4 abgelegt.

Listing L.4 *Bash*-Skript zum Starten von *seyren*

```
#!/bin/bash
GRAPHITE_URL=http://192.168.122.1:81
SMTP_HOST=zimbra.example.com
SMTP_USER=system-seyren
SMTP_PASSWORD="XXX"
SMTP_FROM=XXX

if [ ! -e mongodata ]; then
        mkdir mongodata
        chmod -R 777 mongodata
fi

docker rm -f mongo-seyren
docker run \
        -d \
        -p 27017:27017 \
        --restart=always \
        --name mongo-seyren \
        -v $(pwd)/mongodata:/data/db \
    --device-read-bps=/dev/mapper/lvmraid1secure-anoStartup--seyren:1mb \
    --device-write-bps=/dev/mapper/lvmraid1secure-anoStartup--seyren:1mb \
    --memory="512m"  \
    --cpuset-cpus="0" \
        mongo:3.0.1

docker rm -f seyren
docker run \
        -d \
        -p 192.168.122.1:8085:8080 \
        --link mongo-seyren:mongodb \
        -e GRAPHITE_URL=$GRAPHITE_URL \
    -e SMTP_HOST=$SMTP_HOST \
        -e SMTP_USER=$SMTP_USER \
        -e SMTP_PASSWORD=$SMTP_PASSWORD \
        -e SMTP_FROM=$SMTP_FROM \
        -it \
        --restart=always \
        --name seyren \
        -v /etc/hosts:/etc/hosts:ro \
        --device-read-bps=/dev/mapper/lvmraid1secure-anoStartup--seyren:1mb \
        --device-write-bps=/dev/mapper/lvmraid1secure-anoStartup--seyren:1mb \
        --memory="768m"  \
        --cpuset-cpus="0" \
        usman/docker-seyren
```

Um die prozentuale Festplattenbelegung aus dem belegtem und freiem Festplattenplatz zu errechnen, wird zunächst die Summe aus beiden gebildet um anschließend den Prozentsatz zu errechnen. *syren* verwendet die Syntax von *Graphite*, in welcher die prozentuale Festplattenbelegung vom Wurzelverzeichnis des Systems *luke.example.com* wie folgt errechnet wird:

`asPercent(collectd.luke.df-root.df_complex-used,sumSeries(collectd.luke.df-root.df_complex-free,collectd.luke.df-root.df_complex-used))`

Um bei einem unbekannten Status alarmiert zu werden, wird der Wert *NULL* in 100 mittels `transformNull()` umgewandelt:

```
transformNull(asPercent(collectd.luke.df-root.df_complex-used,sumSeries(collectd.luke.df-root.df_complex-free,collectd.luke.df-root.df_complex-used)),100)
```

Überschreitet die prozentuale Festplattenbelegung 90 Prozent, so wird eine Warnung erstellt und via E-Mail versendet. Überschreitet die prozentuale Festplattenbelegung 95 Prozent, so wird ein Fehler erstellt und via E-Mail versendet. Weiterhin werden Grenzwerte für die Last über die letzten fünf Minuten, die CPU-Auslastung und die Arbeitsspeicherauslastung erstellt.
Die Überwachung der Verfügbarkeit der Webanwendung via *HTTP* erfolgt über eine Überwachungslösung des *Internet Service Providers*.

Protokollierung: Visualisierte Protokollierung

Implementierungs-Dauer: 2 Tage
Es wird auf den *ELK*-Stack umgestellt. Dieser wird als *Docker*-Container auf dem Wirstsystem gestartet, da viel Festplattendurchsatz zu erwarten ist. Entsprechend wird die *rsyslogd*-Konfiguration auf das Format *JSON* umgestellt und die Protokolleinträge an *luke.example.com* mit Port 514 gesendet (siehe Listing L.5 auf der nächsten Seite).

L.2.3 Infrastruktur

Anwendungen laufen in virtuellen Umgebungen

Implementierungs-Dauer: 2 Tage
Bei starken Festplatten- und Netzwerk-Operationen empfiehlt sich die Nutzung von *Docker* anstelle von virtuellen Maschinen. Zunächst war der Backupdienst innerhalb einer virtuellen Maschine und hat auf andere virtuelle Maschinen für das Backup zugegriffen. Dies führte zu starken Performance-Problemen. Nach Migration zu *Docker* auf *luke.example.com* sind diese verringert wurden. Beispielsweise ist das Backup des anoStartup-Mailservers *zimbra* von 121 auf 80 Minuten bei gleicher Größe gesunken. Das entspricht einer Geschwindigkeit von 4,3 Megabyte pro Sekunde zu 6,6 Megabyte pro Sekunde also eine Steigerung um 53,49 Prozent.
In virtuellem Maschinen mittels *KVM* laufen *deployment.example.com*, der *production.example.com* und *stage.example.com*. Die Hauptanwendungen laufen in *Docker*. Anwendungen wie *collectd* dagegen laufen direkt auf den jeweiligen virtuellen Maschinen. Auf *stage.example.com* erfolgt die Verteilung sowie Auslieferung der Webanwendung für Testzwecke. Sobald anoStartup mehr Kapital hat, wird ein Server mit mehr Arbeitsspeicher bestellt und die Aufgaben auf zwei virtuelle Maschinen verteilt.
anoStartup hat virtuelle Maschinen und *Docker* von Anfang an genutzt, so dass nur teilweise Anwendungen in *Docker* überführt worden sind.
Docker-Container werden immer mit der Option *–restart=always* gestartet, damit die Container auch nach einem Neustart automatisch hochfahren.

Produktionsnahe Umgebung steht Entwicklern zur Verfügung

Implementierungs-Dauer: /

Listing L.5 Konfiguration von *rsyslogd* in der Datei */etc/rsyslog.d/10-rsyslog.conf* zur Protokollierung im Format *JSON*

```
template(name="ls_json"
         type="list"
         option.json="on") {
           constant(value="{")
             constant(value="\"@timestamp\":\"")          property(name="
                timegenerated" dateFormat="rfc3339")
             constant(value="\",\"@version\":\"1")
             constant(value="\",\"message\":\"")          property(name="
                msg")
             constant(value="\",\"host\":\"")             property(name="
                hostname")
             constant(value="\",\"logsource\":\"")        property(name="
                fromhost")
             constant(value="\",\"severity_label\":\"")   property(name="
                syslogseverity-text")
             constant(value="\",\"severity\":\"")         property(name="
                syslogseverity")
             constant(value="\",\"facility_label\":\"")   property(name="
                syslogfacility-text")
             constant(value="\",\"facility\":\"")         property(name="
                syslogfacility")
             constant(value="\",\"program\":\"")          property(name="
                programname")
             constant(value="\",\"pid\":\"")              property(name="
                procid")
             constant(value="\",\"priority\":\"")         property(name="
                pri")
             constant(value="\",\"rawmsg\":\"")           property(name="
                rawmsg")
             constant(value="\",\"syslogtag\":\"")        property(name="
                syslogtag")
             constant(value="\",\"appname\":\"")          property(name="
                app-name")
           constant(value="\"}\n")
         }

*.*    @luke.example.com:5544;ls_json
```

Durch die Nutzung von *Docker* lässt sich eine Produktionsumgebung nachstellen. Dieser Punkt ist bereits durch die Implementierung eines „Definierten Erzeugungs- und Verteilungsprozess“ umgesetzt.

L.2.4 Kultur und Organisation

Gute Kommunikation wird belohnt

Implementierungs-Dauer: 1 Tag

Bei Besprechungen zwischen den beiden Geschäftsführern wird gute Kommunikation im Bereich Informationssicherheit gegenseitig mit Schokolade belohnt.

Pro Team ist ein Sicherheitsverantwortlicher definiert

Implementierungs-Dauer: /

Timo Pagel ist für die Sicherheit der Systeme und Anwendungen verantwortlich.

L.2.5 Test und Verifizierung

Statische Tiefe: Statische Analyse für wichtige serverseitige Bereiche

Implementierungs-Dauer: 4 Tage

Für die serverseitgen Komponenten wird *FindBugs* mit dem *FindSecurityBugs Plugin* genutzt. Dies ist einerseits in die Entwicklungsumgebung *JetBrains IntelliJ* integriert (siehe Anhang L.4 auf Seite 205 für ein Bildschirmfoto) und andererseits erfolgt auf *Jenkins* periodisch ein Test.

Zunächst wurde *FindBugs* mittels *findbugs4sbt* in den Erzeugungsprozess integriert. Allerdings unterstützt *findbugs4sbt* noch keine Plugins. Deshalb wurde *findbugs* heruntergeladen, in dem Ordner *security* des Quellcodes der Webanwendung abgelegt und wird mittels Skript *security/findbugs.bash* (siehe Anhang L.7 auf der nächsten Seite) durch *Jenkins* gestartet. Durch die gescheiterte Integration mittels *findbugs4sbt* kommt die erhöhte Implementierungs-Dauer zustande.

Für den Aufruf mittels *Jenkins* als auch für die Integration in die Entwicklungsumgebung wird die Kategorie *Security* getestet. Weiterhin wird der Test auf *ES_COMPARING_STRINGS_WITH_EQ* (nachfolgend ES_C) der Kategorie *Bad Practice* in der Datei *findbugs-include-filter.xml* aktiviert (siehe Anhang L.6). Mittels *ES_C* wird getestet, ob Objekte mittels == Operator vergleichen werden. Werden

Listing L.6 Konfiguration *security/findbugs-includes-filter.xml* zur Angabe der zu testenden Kategorien

```
<FindBugsFilter>
    <Match>
        <Bug category="SECURITY"/>
    </Match>
    <Match>
        <Bug code="ES_COMPARING_STRINGS_WITH_EQ" />
    </Match>
</FindBugsFilter>
```

Objekte mittels == Operator vergleichen, so werden die Referenzen beider Objekte vergleichen und es kann vorkommen, dass zwei Zeichenketten nicht identisch sind [114].

Beim ersten Durchlaufen wurden in der Kategorie *Security* 36 Warnungen und keine kritischen Fehler gemeldet. Es wurden zwei Warnungen für ES_C gemeldet.

Ein richtig Positiver mit Sicherheitsbezug ist ein Zeichenkenvergleich mittels == Operator in Java. Ein erstelltes *Event* kann mittels Passwort vor nicht berechtigten Zugriffen geschützt werden. In der Webanwendung wird mittels Vorbedingung in einer Methode ermittelt (siehe Listing L.8 auf der nächsten Seite für ein vereinfachtes Beispiel), ob das korrekte Passwort angegeben wurde. Das würde dazu führen, dass die Seite des *Events* ausgeliefert wird, obwohl ein falsches Passwort angegeben wurde. Der Vergleich wurde durch Aufruf der Methode `equals()` korrigiert.

Ein falsch Positiver ist beispielsweise die in dem Bildschirmfoto in Abb. L.4 auf Seite 205 gezeigte, als Mittel eingestufte, Warnung für die Nutzung eines vorhersagbaren Zufallsgenerator. In diesem Fall werden Bilder zufällig für die Ausgabe sortiert. Entsprechend ist eine Vorhersage der Reihenfolge der Bilder nicht kritisch und die Meldung wird mittels der Annotation `@SuppressFBWarnings("PREDICTABLE_RANDOM")` als falsch Positiver markiert.

Nach Behebung der richtig Positiven und Markierung von falsch Positiven sind 26 Warnungen nicht behandelt.

Listing L.7 *Bash*-Skript *security/findbugs.bash* zum Starten von *FindBugs*

```
#!/bin/bash

LIB_PATH=$(pwd)/../target/docker/stage/opt/docker/lib/
FILES_TO_CHECK=$(find $LIB_PATH | grep "editor.editor\|listing.listing\|
    subdomain.subdomain" | sed 's/\n/ /')
CLASS_PATH_TO_CHECK=$(find ../target/scala-2.11/classes/ | grep class$ |
     sed 's/\n/ /')
FILES_TO_CHECK="$FILES_TO_CHECK $CLASS_PATH_TO_CHECK"

REPORT_EXTENSION=$1
OUTPUT_XML=reports/findbugs-$REPORT_EXTENSION.xml

./findbugs/bin/findbugs \
        -auxclasspath $LIB_PATH \
        -medium \
        -effort:default \
        -xml \
        -output $OUTPUT_XML \
        -include findbugs-include-filters.xml \
        -exclude findbugs-exclude-filters.xml \
        -projectName "anoStartup" \
        $FILES_TO_CHECK

java -cp findbugs/lib/findbugs.jar edu.umd.cs.findbugs.
    PrintingBugReporter -html $OUTPUT_XML > reports/findbugs-
    $REPORT_EXTENSION.html
```

Listing L.8 Richtig Positiver eines Zeichenkettenvergleich ermittelt durch *FindBugs*

```
if (input.get("password") != event.getAccessPassword()) {
   return redirect("/"); // Access not allowed
}
// Display site
```

Konsolidierung: Alarme sind einfach visualisiert

Implementierungs-Dauer: 1 Tag

Die Visualisierung vom *OWASP Dependency-Check Plugin* erfolgt über das *Plugin* direkt als Trend. Die Visualisierung der Schwachstellen von *Zap* und *FindBugs* erfolgt über das *Log Parser Plugin*. In Abb. L.5 auf Seite 206 zeigt die Statistik von *FindSecurityBugs*. Zusätzlich bieten beide Werkzeuge einen *HTML*-Bericht, welcher über das *HTML Plugin* [15] in *Jenkins* angezeigt werden kann.

Anwendungstest: Integrationstests mit Sicherheitsbezug

Implementierungs-Dauer: 3 Tage

Die Anwendung bietet eine Schnittstelle, von welcher ein Benutzername und ein Passwort gefordert wird. Beispielhaft wurde die Klasse `ApiLoginTest` erstellt, welche mittels der Methoden `loginWithExistingUser()` und `loginWithNonExistingUser()` testet, dass sich Benutzer erfolgreich via *HTTP* anmelden können und Zugriff nicht ohne hinterlegten Benutzer erfolgen kann. Der Quellcode ist im Anhang L.9 auf der nächsten Seite abgelegt. Wie auch bei den „Modultests mit Sicherheitsbezug“ gilt, dass der Integrationstest nicht mehr ausführbar ist.

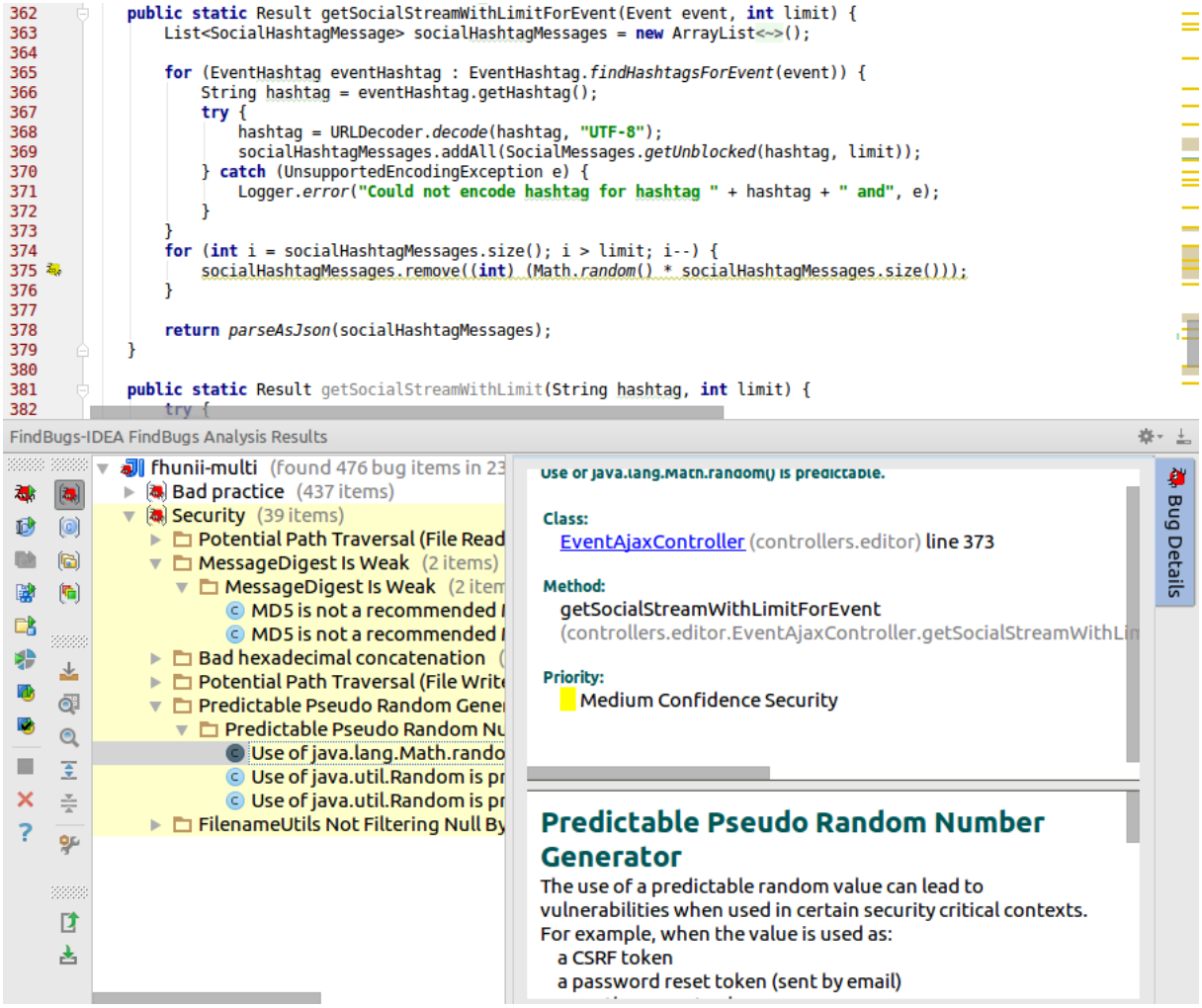


Abbildung L.4: Bildschirmfoto eines gefundenen falsch Postiven in der Entwicklungsumgebung mittels *FindSecurityBugs*

Listing L.9 Integrationstest *ApiLoginIntegrationTest*

Ist in der öffentlichen Version nicht verfügbar.

Infrastrukturtest: Prüfung der Konfiguration von virtuellen Umgebungen

Implementierungs-Dauer: 1 Tage

Mittels *Docker Bench for Security* wird die D*ocker*-Konfiguration virtueller Umgebungen manuell getestet. Der Aufruf, bei welchem alle Bindungen des Dateisystems nur lesenden Zugriff erhalten, ist wie folgt:

Listing L.10 Aufruf von *Docker Bench for Security*

```
1 docker run -it --net host --pid host --cap-add audit_control \
2     -v /var/lib:/var/lib:ro \
3     -v /var/run/docker.sock:/var/run/docker.sock:ro \
4     -v /usr/lib/systemd:/usr/lib/systemd:ro \
5     -v /etc:/etc --label docker_bench_security:ro \
6     docker/docker-bench-security
```

Hier werden Meldungen wie *Container Restrict network traffic between containers* mit Kritikalität *WARN* erzeugt. Die Restriktion von Netzwerken ist erfasst unter dem IP „Kontrollierte Netzwerke für virtuelle

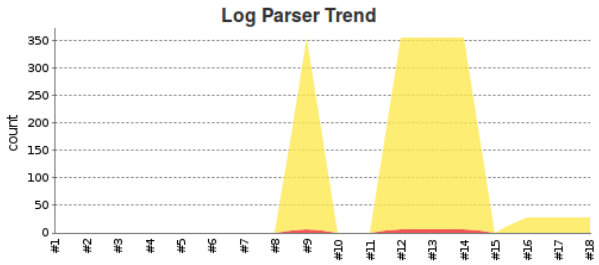


Abbildung L.5: Visualisierung für den Trend von des Tests *FindSecurityBugs* mittels *Log Parser Plugin*

Umgebungen“.
Da der Test nur manuell erfolgte, wird dieser IP nicht als ab geschlossen markiert.

L.3 Implementierung von Ebene 3

L.3.1 Erzeugung und Verteilung

Erzeugung: Gleiches Artefakt für Umgebungen

Implementierungs-Dauer: /
Durch die Nutzung von *Docker* lässt sich das gleiche Artefakt auf der Test- und Produktionsumgebung starten. Dieser Punkt ist bereits durch die Implementierung eines „Definierten Erzeugungs- und Verteilugnsprozess“ umgesetzt.

Erzeugung: Versionierung der Konfiguration

Implementierungs-Dauer: 1 Tag
Die Versionierung der Konfiguration der Auftrage ist mittels *JobConfigHistory Plugin* in *Jenkins* implementiert.

Verteilung: Umgebungsunabhängige Konfigurationsparameter

Implementierungs-Dauer: 1 Tag
Im Quellcode der Anwendung werden häufig die Methoden `isDev()` und `isProd()` zur Identifizierung der Entwicklungs- beziehungsweise Produktionsumgebung aufgerufen. Beispielsweise wurde mittels `isDev()` in der Anwendung getestet, ob die Komponente *fetching*, welche noch in Entwicklung ist, initialisiert werden soll. Die Methode ist durch die Abfrage des Konfigurationsparameters *anoStartup.fetching.active* ausgetauscht.

L.3.2 Informationsgewinnung

Informationsgewinnung: Ausnahmen von Anwendungen werden erfasst

Implementierungs-Dauer: 1 Tag
Jede auf oberster Ebene geworfene Ausnahme innerhalb der Anwendung wird mit Kritikalität *Error* zentral protokolliert. Weiterhin wird jeder Protokolleintrag mit Kritikalität *Error* als Metrik via `log.`

`MetricAppender` erfasst. Mittels *seyren* erfolgt eine Alarmierung. Der Quellcode innerhalb der Anwendung ist wie folgt:

Listing L.11 Erzeugung der Metrik *error-log* in der Klasse *log.MetricAppender*

```
1 [...]
2 public class MetricAppender extends AppenderBase<ILoggingEvent>{
3     @Override
4     protected void append(ILoggingEvent eventObject) {
5         if(eventObject.getLevel().isGreaterOrEqual(Level.ERROR)) {
6             GraphMeasurement.measureApplicationFunctionalMetric("error-log
                  ");
7         }
8     }
9 }
```

L.3.3 Infrastruktur

Produktions-Artifakte sind versioniert

Implementierungs-Dauer: /

Durch Nutzung eines Repositoriums sind alle Anwendungsartefakte versioniert. Die virtuelle Maschine *production.example.com* ist dagegen nicht versioniert. Entsprechend kann dieser Punkt nicht als abgeschlossen markiert werden.

Virtuelle Umgebungen sind limitiert

Implementierungs-Dauer: 2 Tage

Limitierungen für Arbeitsspeicher, CPU sowie Festplattenoperationen werden in der Konfigurationsdatei der virtuellen Maschine vorgenommen. Der Festplattenplatz wird mittels extra logischer Partition durch das Wirtsystem vorgegeben und in der virtuellen Maschine eingebunden.

Limitierungen können wie in Listing L.12 am Beispiel der virtuellen Maschine *anoStartup-stage* in der Konfiguration definiert werden.

Listing L.12 Auszug der Konfiguration einer virtuellen Maschine mit Limitierungen

```
1  <vcpu placement='static' cpuset='4-5'>2</vcpu>
2  <memory unit='KiB'>1955136</memory>
3  <currentMemory unit='KiB'>1955136</currentMemory>
4  <devices>
5     <disk type='file' device='disk'>
6       <source file='/dev/lvmraid1secure/anoStartup-stage'/>
7       <iotune>
8         <read_iops_sec>150</read_iops_sec>
9         <write_iops_sec>150</write_iops_sec>
10      </iotune>
11      [...]
12    </disk>
```

Dabei werden in Zeile zwei die virtuelle Prozessorkerne 4 und 5 der virtuellen Maschine zugewiesen. In Zeile drei und vier ist der zur Verfügung stehende Arbeitsspeicher definiert. In Zeile sieben wird die

logische Partition */dev/lvmraid1secure/anoStartup-stage* als Festplattenspeicher definiert. In Zeile acht und neun sind die Lese- und Schreib-Operationen pro Sekunde auf 150 begrenzt.

Docker-Container sind wie in Listing L.13 aufgezeigt begrenzt. Der Schreib- und Lesedurchsatz ist jeweils auf 1 Megabyte die Sekunde begrenzt, der Arbeitsspeicher auf 768 Megabyte und es steht nur der erste Prozessorkern zur Verfügung. Weiterhin ist der Speicherplatz des Ordner mit den persistenten Daten des Containers mittels logischer Partition begrenzt. Auf *luke.example.com* sind alle Abbilder und Container von *Docker* unter */var/lib/docker* gespeichert, der Speicherplatz des Ordner ist mittels logischer Partition begrenzt.

Listing L.13 Auszug der Konfiguration eines *Docker*-Containers mit Limitierungen

```
1 docker run [...]
2         --device-read-bps=/dev/mapper/lvmraid1secure-anoStartup--seyren
              :1mb \
3         --device-write-bps=/dev/mapper/lvmraid1secure-anoStartup--seyren
              :1mb \
4         --memory="768m"  \
5         --cpuset-cpus="0" \
6 [...]
```

L.3.4 Test und Verifizierung

Statische Tiefe: Statische Analyse für wichtige klientenseitige Bereiche

Implementierungs-Dauer: 3 Tage

Der Test erfolgt auf *Jenkins* mittels *ESlint* in einem *Docker*-Container. Der Aufruf in *Jenkins* ist in Listing L.14 aufgeführt.

Listing L.14 Klientenseitiger Test auf Schwachstellen mittels *ESlint* in *Jenkins*

```
1 cd security
2 touch reports/eslint.html
3 touch reports/eslint.txt
4 chmod 755 reports/eslint.html
5 chmod 755 reports/eslint.txt
6
7 docker run -v $(pwd)/..:/src -v $(pwd)/reports:/reports anoStartup/
      eslint-docker eslint -o /reports/eslint.html -f html /src/modules/
      editor/public/javascripts
8 docker run -v $(pwd)/..:/src -v $(pwd)/reports:/reports anoStartup/
      eslint-docker eslint -o /reports/eslint.txt -f compact /src/modules/
      editor/public/javascripts
9
10 cat reports/eslint.txt
```

Das *Dockerfile* zum Erzeugen des *Docker*-Containers ist unter https://github.com/wurstbrot/retire.js aufgeführt. Der automatische Erzeugungsdienst von *hub.docker.com* erzeugt nächtlich aus dem *Dockerfile* das Abbild *anoStartup/eslint-docker*.

ESlint nutzt eine Regeldatei[3], welche *scanjs*-Regeln beinhaltet. Es wird nur der *JavaScript*-Quellcode von dem Modul *editor* getestet, dabei sind keine kritischen Fehler und 11 Warnungen gemeldet worden.

[3]Kopiert von https://github.com/18F/compliance-toolkit/blob/master/configs/static/.eslintrc.

Statische Tiefe: Test auf klientenseitige Komponenten

Implementierungs-Dauer: 3 Tage

Der Test auf klientenseitige Komponenten mit bekannten Schwachstellen erfolgt mittels *retire.js* als *Docker*-Container. Dafür ist ein *git*-Projekt auf github.com (siehe https://github.com/wurstbrot/retire.js) angelegt, welches über *hub.docker.com* regelmäßig erzeugt wird und unter der Bezeichnung *anoStartup/retire.js* zur Verfügung steht. Entsprechend kann über *Jenkins* die Webanwendung vom Versionskontrollsystem abgeholt, die Bibliotheken mittels `./activator docker:stage` abgeholt und der *Docker*-Container gestartet werden. Mittels *Log Parser Plugin* für *Jenkins* wird auf Vorkommnis von „severity: high“ für Fehler, „severity: middle“ für Warnungen und „severity: low“ für Informationen getestet. Wird ein Fehler gefunden, so wird der Auftrag als fehlerhaft markiert. Das von *Jenkins* ausgeführte Skript ist im Listing L.15 aufgeführt.

Listing L.15 *Bash*-Skript zur Durchführung des Test auf klientenseitige Komponenten mit bekannten Schwachstellen

```
1  ./activator clean
2  ./activator docker:stage
3
4  REPORT_FILE=retirejs.report.txt
5  if [ -e $REPORT_FILE ]; then
6          rm $REPORT_FILE fi
7  WORSPACE_PATH=/home/jenkins/build/workspace/$JOB_NAME
8  DOCKER_PATH=/usr/src/app
9
10 docker run --rm -v $WORSPACE_PATH:$DOCKER_PATH anoStartup/retire.js --
       ignorefile /usr/src/app/security/.retireignore --outputpath
       $DOCKER_PATH/$REPORT_FILE || true
11
12 cat $WORSPACE_PATH/$REPORT_FILE
```