Digital Business University
of Applied Sciences

Cyber- & IT-Security (M.Sc.)

Cyber- & IT-Security: Implementation &
Application

Sommersemester 2024

DevSecOps in der Praxis: Sicherheit als integraler Bestandteil von Continuous Integration & Delivery

Studienarbeit

Eingereicht von:
Elias Häussler

Matrikelnummer:
200094

Dozent:
David Lübeck

20. September 2024

Inhaltsverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
AWS	Amazon Web Services
BDST	Behaviour Driven Security Testing
CaC	Compliance as Code
CAMS	Culture, Automation, Measurement, and Sharing
CD	Continuous Delivery
CD	Continuous Deployment
CI	Continuous Integration
CIS	Center for Internet Security
CS	Continuous Security
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DAST	Dynamic Application Security Testing
FOSS	Free Open Source Software
IaC	Infrastructure as Code
IAST	Interactive Application Security Testing
IDE	Integrated Development Environment
KI	Künstliche Intelligenz
KICS	Keeping Infrastructure as Code Secure
KMS	Key Management Service
ML	Machine Learning
NVD	National Vulnerability Database
OpenVAS	Open Vulnerability Assessment Scanner
OWASP	Open Worldwide Application Security Project
RASP	Runtime Application Self-Protection
SaaS	Software as a Service
SaC	Security as Code
SAMM	Software Assurance Maturity Model
SAS	Security API Scanning
SAST	Static Application Security Testing

SBOM Software Bill of Materials

SCA Software Composition Analysis

SDLC Software Development Lifecycle

WAF Web Application Firewall

WAST Web Application Security Testing

XSS Cross-Site-Scripting

YAML YAML Ain't Markup Language

ZAP Zed Attack Proxy

1 Einleitung

Im modernen Zeitalter der Software- und Anwendungsentwicklung zeigt sich immer häufiger, dass kritische Sicherheitslücken in Softwarecode während der Entwicklung unentdeckt bleiben und Software dadurch anfällig für eine Vielzahl von Cyberangriffen wird. Prominente Beispiele sind etwa das *Equifax*-Datenleck aus dem Jahr 2017 oder der *Heartbleed*-Bug aus dem Jahr 2014. Beide Fälle zeigen, dass schon durch das Auslassen kleinerer Sicherheitsüberprüfungen ein enormer finanzieller und reputativer Schaden entstehen kann. Aber auch unzureichende oder zu spät im Software-Entwicklungszyklus durchgeführte Sicherheitsüberprüfungen können schwerwiegende Folgen haben.

Viele dieser Vorfälle haben zur Erkenntnis geführt, dass der Sicherheitsaspekt bei der Software-Entwicklung einen stärkeren Fokus erhalten muss. Ähnliche Erkenntnisse führten bereits zum *DevOps*-Konzept, das sich schon seit vielen Jahren bewährt hat. Die moderne DevOps-Praxis fordert mittlerweile eine frühzeitige Integration umfassender Sicherheitsmaßnahmen, um Schwachstellen frühzeitig zu erkennen und zu beheben. Wenn überhaupt, wurden Sicherheitsmaßnahmen historisch meist erst nach der Entwicklung während des Betriebs durchgeführt. Durch das Konzept der *Continuous Security* sollen sie heute kontinuierlich in alle Phasen der Software-Entwicklung eingebunden werden (sog. *DevSecOps*).

1.1 Problemstellung

Die Idee von DevOps liegt einerseits in einer ganzheitlichen Betrachtung von Software-Entwicklung und -Betrieb. Andererseits soll damit ein agiler Prozess manifestiert werden, durch den schnelle Ergebnisse erzielt und einfache Handhabung ermöglicht werden soll. DevOps verfolgt daher auch das Prinzip *Culture, Automation, Measurement, and Sharing (CAMS)*. Eine Erweiterung dieses Prozesses um ausgewählte Sicherheitsmechanismen darf sich nicht nachteilig auf dessen Performance auswirken. Ansonsten besteht die Gefahr, dass durch die häufig ohnehin schon mangelnde Sicherheitskultur im DevOps-Umfeld eine Einführung notwendiger Sicherheitsmaßnahmen erfolglos sein wird.

Damit ein Übergang von DevOps zu DevSecOps gelingen kann, müssen bisherige Prozesse umgestaltet und eine neue Sicherheitskultur etabliert werden. Dies kann mithilfe von Continuous Security erreicht werden, indem Sicherheitslücken frühzeitig erkannt werden, wodurch die Gesamtqualität der Software oder Anwendung erhöht wird. Zur Annäherung dieser Ziele wird in der vorliegenden Arbeit auf folgende Fragestellungen eingegangen:

- Wie können Sicherheitsmaßnahmen effektiv in bestehende DevOps-Prozesse integriert werden, ohne die Agilität des Gesamtprozesses zu beeinträchtigen?

- Welche Tools sind für die Implementierung von Continuous Security in einer DevSecOps-Pipeline geeignet?
- Welche Herausforderungen entstehen durch die Integration von Sicherheitsmaßnahmen in DevSecOps-Pipelines?

1.2 Aktueller Forschungsstand

DevSecOps und die Anwendung von Continuous Security sind keinesfalls unbekannte Verfahren und Konzepte. Es existieren eine Vielzahl an Best Practices zur frühen und kontinuierlichen Integration von Sicherheitsmaßnahmen in bestehende DevOps-Prozesse. Der Begriff *DevSecOps* wird dabei teilweise unterschiedlich ausgelegt; es existieren auch die Bezeichnungen *SecDevOps* und *DevOpsSec* (Zeeshan, 2020). Darüber hinaus stellten Lombardi und Fanton (2023) das Konzept des *CyberDevOps* vor – eine weiterentwickelte, noch konsequentere Umsetzung von Continuous Security. In anderen Publikationen werden einzelne Sicherheitsmaßnahmen fokussiert behandelt, um etwa mit wenig Integrationsaufwand ein hohes Maß an Sicherheit und Agilität zu gewährleisten und beide Kernaspekte miteinander zu kombinieren (siehe etwa Pan (2019)).

1.3 Inhalte der Arbeit

In dieser Arbeit steht die Auswahl und Integration einzelner Sicherheitsmechanismen in DevSecOps-Pipelines im Vordergrund. Der Fokus liegt dabei auf einer einfach verständlichen Formulierung von Zielen, Maßnahmen und Herausforderungen, die bei der Anwendung von Continuous Security entstehen; eine Messung hinsichtlich der Effektivität einzelner Maßnahmen erfolgt nicht. Des Weiteren umfasst die Arbeit lediglich die Betrachtung von Maßnahmen während der Software-Entwicklung, wohingegen Sicherheitsaspekte zum Software-Betrieb nicht im Fokus stehen.

Zunächst werden in Kapitel 2 die Grundlagen für das weitere Verständnis dieser Arbeit vermittelt. Anschließend werden in Kapitel 3 Ziele definiert und Maßnahmen erarbeitet, die im späteren Verlauf in Kapitel 4 im Rahmen eines praktischen Beispiels angewendet werden. In diesem Kapitel werden zudem Herausforderungen bei der Anwendung der Sicherheitsmaßnahmen benannt und mögliche Lösungsansätze aufgezeigt. Abschließend werden die Ergebnisse in Kapitel 5 zusammengefasst und ein Ausblick auf weitere Entwicklungsansätze gegeben.

2 Grundlagen

2.1 Terminologie

Nachfolgend werden einige grundlegende Begriffe bestimmt, die im Laufe dieser Arbeit Erwähnung finden. Darüber hinaus wird vorausgesetzt, dass ein gewisses Grundverständnis in den Bereichen *Software-Entwicklung* und *Versionskontrolle* besteht.

2.1.1 DevOps & DevSecOps

DevOps beschreibt die Verschmelzung von Entwicklung (engl. *development*) und Betrieb (engl. *operations*) einer Software oder Anwendung. Dabei handelt es sich um ein bewährtes Konzept, um „Software schneller und in besserer Qualität auszuliefern und einen möglichst reibungslosen und störungsfreien Betrieb zu gewährleisten“ (Hebing & Manhembué, 2024, S. 219). Mit dem Begriff *DevSecOps* wird dieses Konzept um die Einbindung automatisierter Sicherheitsmaßnahmen erweitert (Kumar & Goyal, 2021). Dadurch wird erreicht, dass Sicherheitsaspekte während des gesamten Entwicklungs- und Betriebszyklus mitgedacht und aktiv umgesetzt werden. Die Betrachtung erfolgt dabei zumeist im Kontext des sog. *Software Development Lifecycle (SDLC)*.

2.1.2 Continuous Integration (CI) & Continuous Delivery (CD)

Die Konzepte *Continuous Integration (CI)* und *Continuous Delivery (CD)* stellen essenzielle Methodiken für eine robuste kontinuierliche Software-Entwicklung dar. CI beschreibt einen automatisierten Prozess „zum frühzeitigen Aufdecken von Fehlern und Reduzieren von Integrationskosten“ (Hanschke, 2017, S. 30). CD zielt darauf ab, „den integrierten Code automatisch auch in einer bereits funktionsfähigen Umgebung [...] zur Verfügung zu stellen“ (Hebing & Manhembué, 2024, S. 214). Eine Erweiterung von Continuous Delivery stellt *Continuous Deployment (CD)* dar, bei dem Softwarecode automatisiert in eine Produktiv-Umgebung gebracht wird, wenn alle vorherigen Prozesse erfolgreich verlaufen sind (Hebing & Manhembué, 2024). Diese Konzepte werden meist gemeinsam angewendet und in sog. *Pipelines* ausgeführt, die hierzu eine Reihe vordefinierter Skripte auslösen, welche „für einen schnellen, wiederholbaren, zuverlässigen und risikoarmen Prozess für das Bauen, Testen und Ausliefern neuer Softwarereleases“ sorgen (Hanschke, 2017, S. 47).

2.1.3 Continuous Security (CS)

Das Bindeglied zwischen DevOps und DevSecOps im Kontext von CI/CD stellt *Continuous Security (CS)* dar. CS umfasst eine Reihe automatisierter Sicherheitsaufgaben, die in jede Phase des CD/CD-Prozesses integriert werden, wodurch die Software in ihrem gesamten Lebenszyklus mit entsprechenden Sicherheitsaspekten angereichert

wird (Kumar & Goyal, 2021). Der Grundgedanke hinter CS folgt dabei dem *Shift-Left-Security*-Ansatz: Betrachtet man den SDLC als horizontale Zeitachse, so besagt jener Ansatz, dass Sicherheitsmaßnahmen weiter links – oder früher – im Lebenszyklus angesiedelt sein sollten als die typischen Phasen, die früher als Einstiegspunkte für Sicherheitstests und Schutzmaßnahmen dienten (Malone, 2023).

2.2 Klassischer Aufbau einer CI/CD-Pipeline

Mithilfe von CI/CD-Pipelines wird erreicht, dass Softwarecode gebaut, verifiziert, getestet, veröffentlicht und ausgeliefert wird. Typischerweise werden Pipelines in unterschiedliche *Stages* unterteilt, die zumeist aufeinander aufbauen oder voneinander abhängig sein können. In einer klassischen CI/CD-Pipeline sind in der Regel folgende Stages enthalten:

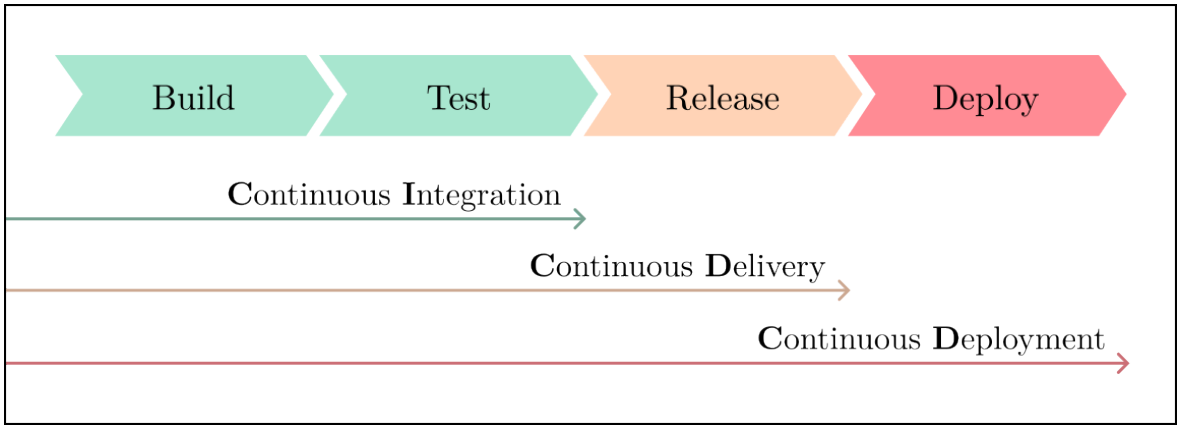

Abbildung 1: Klassischer Aufbau einer CI/CD-Pipeline

1. **Build**: In dieser Stage wird verifiziert, dass die Software korrekt gebaut werden kann. Hierzu werden meist Artefakte generiert, die den Zustand der Software zum aktuellen Stand abbilden. Häufig werden hierzu auch (externe) Abhängigkeiten geladen.

2. **Test**: Diese Stage beinhaltet alle Maßnahmen zum automatisierten Testen des Softwarecodes. Hierzu zählen beispielsweise das Einhalten von Code-Richtlinien, aber auch das Ausführen von Unit-Tests und statischer Code-Analyse.

3. **Release**: Wenn der Softwarecode alle Integritätstests bestanden hat, wird er veröffentlicht. Bei Cloud-Anwendungen erfolgt dies normalerweise erst auf einem (möglicherweise internen) Test- oder *Staging*-System, auf dem zusätzliche Qualitätssicherungsmaßnahmen erfolgen können.

4. **Deploy**: Hat der Softwarecode alle vorherigen Entwicklungs- und Bereitstellungszyklen erfolgreich durchlaufen, wird er schlussendlich produktiv gestellt, was häufig mit einer automatisierten Bereitstellung auf einem Produktivsystem erfolgt.

3 Continuous Security in CI/CD-Pipelines

Es existieren verschiedene Ansätze, um Software, die mit klassischen CI/CD-Pipelines ausgeliefert wird, um geeignete Sicherheitsmaßnahmen und -tools zu erweitern. Wichtig bei der Auswahl dieser Maßnahmen ist es, stets das Prinzip von CS zu verfolgen und den SDLC in allen relevanten Stages ausreichend abzusichern.

3.1 Ziele

Nachfolgend werden zunächst eine Reihe von Zielen und Zielanforderungen benannt, die für den Aufbau einer mit CS angereicherten CI/CD-Pipeline zu erfüllen sind. Es handelt sich dabei um praktische Anforderungen im Sinne der DevSecOps-Methodik.

3.1.1 Konsequente Anwendung des Shift-Left-Security-Ansatzes

Über allen Sicherheitsmaßnahmen steht das Ziel, niemals Softwarecode, dem nicht ausreichende Sicherheitsüberprüfungen vorausgegangen sind, in Produktiv-Umgebungen auszuführen. Aus diesem Gedanken heraus entstand überhaupt erst das agile Konzept *DevSecOps*. Anders als im klassischen DevOps-Prozess sollen Maßnahmen wie Penetration-Testing und statische Code-Analyse nicht erst im Betrieb einer Software oder Anwendung durchgeführt werden, sondern bereits in den vorgelagerten Entwicklungsprozess eingebettet werden (Lombardi & Fanton, 2023).

3.1.2 Betrachtung der Software in ihrer Gesamtheit

Darüber hinaus liegt ein wesentliches Erfolgskriterium in der ganzheitlichen Betrachtung der Software oder Anwendung. Dies umfasst neben selbst entwickeltem Softwarecode auch die Schwachstellenanalyse eingesetzter (externer) Abhängigkeiten wie zum Beispiel Bibliotheken von Drittanbietern (Lombardi & Fanton, 2023). Es muss sichergestellt sein, dass der gesamte ausgelieferte Softwarecode inklusive aller integrierten externen Bestandteile als sicher anzusehen ist.

3.1.3 Sicherstellung der Integrität ausgelieferter Software

Wenn Softwarecode in Produktiv-Umgebungen ausgeliefert wird, muss sichergestellt sein, dass alle Bestandteile den Anforderungen an sicheren Softwarecode entsprechen. Diese Anforderungen können beispielsweise von den Organisationen selbst definiert sein oder konkreten Standards wie dem *Software Assurance Maturity Model (SAMM)* des *Open Worldwide Application Security Project (OWASP)* folgen (Hsu, 2018). Softwarecode kann dann als integer bezeichnet werden, wenn er den gewählten Anforderungskatalog während des gesamten SDLC erfüllt.

3.1.4 Umsetzung geeigneter Tests für eine ausreichende Testabdeckung

Einzelne Bestandteile des Softwarecodes müssen während des gesamten SDLC permanent eine Reihe geeigneter Tests durchlaufen. Es muss sichergestellt sein, dass die ausgelieferte Software auf unterschiedliche Aspekte hin getestet wird, um eine ausreichend hohe Testabdeckung zu erreichen und dadurch die sichere und robuste Ausführung der Software zu gewährleisten. Dieses Ziel kann auch unter dem Begriff *Continuous Testing* zusammengefasst werden (Kumar & Goyal, 2021).

3.1.5 Automatisierung aller Sicherheitsmaßnahmen

DevOps erfordert bereits in allen Bereichen ein hohes Maß an Automatisierung, um schnelle Entwicklung, Auslieferung und Feedback von Endnutzern zu erhalten. Auch im DevSecOps-Bereich ist es erforderlich, die Ausführung aller entwickelten Sicherheitsmaßnahmen zu automatisieren. Nur dadurch kann der agile Ansatz, den DevOps verfolgt, weiterhin konsequent angewendet werden. Der Aspekt der Automatisierung ist darüber hinaus Teil des CAMS-Modells, das charakteristisch für DevOps ist (Myrbakken & Colomo-Palacios, 2017).

3.2 Maßnahmen

Nachfolgend werden einige Maßnahmen vorgestellt, um die zuvor genannten Zielvorgaben (siehe Abschnitt 3.1) in einer CI/CD-Pipeline umzusetzen. Die Auflistung beinhaltet die am weitesten verbreiteten Maßnahmen. Sie ist nicht vollständig und kann je nach Anwendungsfall variieren und erweitert werden. Ein praktisches Beispiel zur Anwendung dieser Maßnahmen ist in Abschnitt 4.3 skizziert.

3.2.1 Software Composition Analysis (SCA)

Software Composition Analysis (SCA) fasst Werkzeuge zusammen, um die Abhängigkeiten einer Software, etwa zu externen Bibliotheken, zu analysieren und zu scannen. Die Analyse umfasst unter anderem den Abgleich der verwendeten Lizenzen von Drittanbietern sowie das Scannen externer Abhängigkeiten auf bekannte Schwachstellen und Sicherheitslücken, mit denen die Software oder eine dahinter liegende Anwendung gefährdet werden könnte (Feio et al., 2024; Lombardi & Fanton, 2023).

Für die Analyse wird häufig eine sog. *Software Bill of Materials (SBOM)* angelegt und mit einer Schwachstellen-Datenbank wie der *National Vulnerability Database (NVD)* abgeglichen. Die SBOM kann dabei als eine Art Inventar angesehen werden, in dem alle Komponenten einer Software, inklusive der eingesetzten Versionen, der Beziehungen zueinander und weiteren Informationen der einzelnen Anbieter abgebildet sind (Kağızmandere & Arslan, 2024).

Die Durchführung einer SCA ist auch im Hinblick auf die Lizenzierung der Software von besonderer Relevanz. Häufig sind die eingesetzten externen Bibliotheken unter verschiedenen Lizenzen veröffentlicht, welche unter Umständen strikte Lizenzierungsmodelle verfolgen. Die Berücksichtigung dieser Modelle und die Benennung der eingesetzten Bibliotheken und deren Lizenzen ist dann von zentraler Bedeutung, um alle rechtlichen Aspekte durch die Veröffentlichung der Software zu berücksichtigen (Yang, 2018).

SCA gilt als eine wichtige Sicherheitsmaßnahme für das Ziel der ganzheitlichen Betrachtung der Software (siehe Abschnitt 3.1.2), da alle verwendeten Komponenten – auch jene von Drittanbietern – in die Analyse miteinbezogen werden (Feio et al., 2024).

3.2.2 Static Application Security Testing (SAST)

Static Application Security Testing (SAST) umfasst Werkzeuge zur vollständigen Überprüfung von Softwarecode. Der Fokus liegt dabei auf der Analyse unsicherer Code-Muster und bekannter Schwachstellen im Softwarecode (Lombardi & Fanton, 2023; Schreider, 2020). Eine Orientierungshilfe für die Schwachstellenanalyse bieten beispielsweise die *Common Weakness Enumeration (CWE) Top 25*[1] oder *OWASP Top Ten*[2]. Dabei handelt es sich um standardisierte Dokumente mit Auflistungen der zum Zeitpunkt der Veröffentlichung häufigsten und schwerwiegendsten Schwachstellen in Softwarecode.

Häufig wird SAST auch als *White-Box-Testing* bezeichnet, da Softwarecode nicht ausgeführt, sondern lediglich analysiert wird, und die Tests somit vollen Einblick in das Design und den Code der Software oder Anwendung haben (Schreider, 2020). Bei diesem Testverfahren kann im Gegensatz zum *Black-Box-Testing* eine Schwachstelle gezielt auf eine bestimmte Stelle im Softwarecode zurückgeführt werden.

Mit SAST kann somit das Ziel zur Sicherstellung der Integrität ausgelieferter Software (siehe Abschnitt 3.1.3) verfolgt werden.

3.2.3 Dynamic Application Security Testing (DAST)

Im Gegensatz zu SAST werden beim *Dynamic Application Security Testing (DAST)* Tests durchgeführt, mit denen festgestellt werden soll, wie eine laufende Anwendung auf bösartige Anfragen reagiert (Rangnau et al., 2020). Für die Durchführung werden Testfälle in Form manipulierter Anfragen definiert und die Anwendung im Kontext ihrer Laufzeitumgebung während der gesamten Anfrage vom Client bis zu den Backend-Systemen analysiert (Schreider, 2020). Die Herausforderung besteht darin, diejenigen Informationen in der Antwort zu identifizieren, die auf das Vorhandensein einer Sicherheitslücke hinweisen. So kann die Software beispielsweise bei Ein-/

[1] https://cwe.mitre.org/top25/
[2] https://owasp.org/www-project-top-ten/

Ausgabe-Validierungsproblemen auf eine Anfälligkeit von *Cross-Site-Scripting (XSS)* oder *SQL-Injection* hin untersucht werden (Rangnau et al., 2020; Schreider, 2020). Die unterschiedlichen Testszenarien können beispielsweise in *Web Application Security Testing (WAST)*, *Security API Scanning (SAS)* und *Behaviour Driven Security Testing (BDST)* unterteilt werden (Rangnau et al., 2020).

DAST wird auch als *Black-Box-Testing* bezeichnet, da für die Schwachstellenanalyse der dahinter liegende Softwarecode nicht bekannt ist. Vielmehr wird das Verhalten der Anwendung aus Sicht eines Angreifers betrachtet, um daraus vorhandene Sicherheitslücken abzuleiten. Zu den möglichen Testverfahren zählen unter anderem *Fuzz-Tests*, *Penetrationstests*, *Anwendungs-Sicherheitsscans* sowie Tests zum *Proxy-Scanning* (Schreider, 2020).

Auch DAST verfolgt damit das Ziel zur Sicherstellung der Integrität ausgelieferter Software (siehe Abschnitt 3.1.3).

3.2.4 Interactive Application Security Testing (IAST)

Ein neuartigeres Verfahren zur Schwachstellenanalyse stellt *Interactive Application Security Testing (IAST)* dar. Es kombiniert DAST mit SAST und sorgt dafür, dass die Testfälle in Form manipulierter Anfragen mit einer detaillierten Überwachung des internen Anwendungsstatus einhergehen. Dadurch wird eine Korrelation zwischen Anfrage und Software-Verhalten auf Ebene des Softwarecodes hergestellt, was eine exakte Identifizierung der Stellen im Softwarecode erlaubt, an denen die Behebung einer bestimmten Schwachstelle vorgenommen werden sollte (Stanciu, 2023).

Auf Anwendungsebene wird zur Analyse meist ein sog. *Runtime Application Self-Protection (RASP)-Agent* eingesetzt. RASP ist ein Verfahren, das normalerweise in eine *Web Application Firewall (WAF)* integriert wird, um frühzeitig Angriffe zu erkennen und geeignete Abwehrmaßnahmen zu ergreifen (Hsu, 2018).

Aufgrund seiner Zusammensetzung wird IAST auch als *Gray-Box-Testing* bezeichnet. Das Zusammenspiel beider Techniken reduziert insbesondere die *False-Positive*-Rate in Testszenarien und damit auch den sog. *false sense of security* (Pan, 2019).

3.2.5 Erweiterte Unit- und Integrationstests

Ein weiteres benanntes Ziel ist die Umsetzung geeigneter Tests für eine ausreichende Testabdeckung der Software (siehe Abschnitt 3.1.4). Hierzu existieren in einer klassischen CI/CD-Pipeline bereits verschiedene Maßnahmen, darunter etwa die Durchführung von Akzeptanz-, Integrations-, Performance-, Smoke- oder Unit-Tests. Je nach Verfahren werden entweder einzelne Bestandteile des Softwarecodes innerhalb der Anwendung getestet (ähnlich des *White-Box-Testings*) oder die Durchführung der Tests erfolgt von außen in die laufende Anwendung hinein (vgl. *Black-Box-Testing*). Durch

den Einsatz der unterschiedlichen Testverfahren kann eine entsprechend hohe Testabdeckung erreicht werden (Kumar & Goyal, 2021).

Im Kontext von CS ist es von besonderer Bedeutung, die bestehenden Tests um Testfälle zu erweitern, mit denen die Software auf bestimmte Sicherheitsaspekte hin untersucht wird. Dies betrifft insbesondere diejenigen Tests, die auf Ebene des Softwarecodes ausgeführt werden, also vor allem Unit- und Integrationstests. Ziel dieser Testfälle ist es, die Software auf negative Einflüsse hin zu untersuchen, die beispielsweise in der Planungsphase des SDLC in einem sog. *Bedrohungsmodell* (*threat model*) definiert wurden (Kumar & Goyal, 2021).

3.2.6 Secrets-Management

Häufig enthält Softwarecode Bestandteile, deren Informationen geheim gehalten werden sollen. Darunter zählen etwa Anmeldedaten, Zugangsschlüssel oder Zertifikate jeglicher Art. Diese als *Secrets* zusammengefassten Informationen müssen so abgespeichert sein, dass durch Offenlegung des Softwarecodes kein Dritter an sie gelangen kann. Aus dieser Anforderung heraus hat sich das sog. *Secrets-Management* entwickelt. Es beschreibt Verfahren, um geheime Informationen über einen sicheren Weg verfügbar zu machen, ohne sie in den Softwarecode integrieren zu müssen (Hughes, 2022).

Das Secrets-Management zielt neben der sicheren Speicherung geheimer Informationen auch auf die frühzeitige Erkennung unsicher gespeicherter Informationen im Softwarecode ab. Hierfür existieren Scanner, die in den SDLC integriert werden können, um etwa Passwörter, Application Programming Interface (API)-Schlüssel oder persönliche Access-Tokens zu erkennen. Sie arbeiten auf Basis vordefinierter Regeln, die jedoch zumeist auch beliebig erweitert werden können (Hughes, 2022).

Der Einsatz eines effektiven Secrets-Managements zielt ebenfalls auf die Sicherstellung der Integrität ausgelieferter Software (siehe Abschnitt 3.1.3) ab, entfaltet seine Wirkung jedoch vor allem durch die Automatisierung der Sicherheitsmaßnahme (siehe Abschnitt 3.1.5), wodurch noch vor Auslieferung der Software das potenzielle Offenlegen geheimer Informationen verhindert werden kann.

3.2.7 Container-Sicherheit

Eine Software oder Anwendung kann mittels *Containerisierung* in *Container-Images* überführt und mithilfe von Containervirtualisierungstechniken wie *Docker* zur Laufzeit ausgeführt werden. In diesen Fällen ist es notwendig, die CI/CD-Pipeline um weitere Maßnahmen zu erweitern, um diese Container-Images umfassend zu analysieren, bevor sie in einer Laufzeitumgebung ausgeführt werden (Hsu, 2018).

Die Analyse beinhaltet eine Untersuchung auf bekannte Schwachstellen, beispielsweise anhand von Datensätzen aus der *Common Vulnerabilities and Exposures (CVE)*-Datenbank. In dieser werden nahezu alle jemals bekannt gewordenen Schwachstellen

veröffentlicht, inklusive einer Angabe, in welche Sicherheitsstufe die Schwachstelle einzuordnen ist und welche Softwareversionen davon betroffen sind (Hsu, 2018).

Außerdem wird geprüft, ob bewährte Sicherheitsverfahren, die beispielsweise regelmäßig in den *Center for Internet Security (CIS) Docker Benchmarks* veröffentlicht werden, verfolgt und korrekt eingehalten werden (Hsu, 2018). Dadurch wird auch sichergestellt, dass Container-Images frei von Problemen aufgrund möglicher Fehlkonfigurationen sind (Lombardi & Fanton, 2023).

3.2.8 Maßnahmen für Infrastructure as Code (IaC)

Infrastructure as Code (IaC) ist ein Verfahren, bei dem jegliche Systemkonfiguration als programmierbare Einheit behandelt wird. Dies umfasst das gesamte Ökosystem einer Anwendungsinfrastruktur, wie zum Beispiel Server, Geräte, Betriebssystem und andere physische und virtuelle Ressourcen. Sie können so je nach Bedarf und mithilfe standardisierter Vorlagen konfiguriert und entsprechend skaliert werden (Kumar & Goyal, 2021). Darunter fallen auch auch Sicherheitskonfigurationen wie Dateiberechtigungen, Firewall-Regeln oder Datenbank-Verbindungen. Die Konfiguration als Programmcode macht es einfacher, nicht autorisierte Infrastrukturänderungen zu überwachen und im Zweifel die Konfiguration auf frühere spezifische Versionen zurückzusetzen (Hsu, 2018). Ähnliche Ansätze verfolgen auch die beiden Verfahren *Security as Code (SaC)* und *Compliance as Code (CaC)*.

Der Einsatz von IaC stellt keine konkrete Sicherheitsmaßnahme dar, die in eine CI/CD-Pipeline integriert werden kann. Vielmehr handelt es sich um eine Methodik, um die Anwendungssicherheit zu erhöhen und auf sichere Konfigurationseinstellungen zurückzugreifen. Wenn IaC genutzt wird, können spezielle Analyse-Werkzeuge in die Pipeline integriert werden, um sicherzustellen, dass die einzelnen Konfigurationseinstellungen gültig und sicher sind. Damit geht automatisch eine Härtung der Infrastruktur sowie die Durchführung zusätzlicher Sicherheitstests einher (Kumar & Goyal, 2021).

4 Praktische Umsetzung von Continuous Security

Die vorhergehende Beschreibung von Zielen und gängigen Sicherheitsmaßnahmen zeigt eindrucksvoll, dass klassische CI/CD-Pipelines im Bereich der Software-Sicherheit noch viele Lücken aufweisen. Dabei lassen sich viele der genannten Maßnahmen durch existierende Tools einfach in bestehende Pipelines integrieren, um den praktischen Nutzen von CS entlang des gesamten SDLC wirken zu lassen.

In diesem Abschnitt wird beispielhaft dargestellt, welche Ansätze es zur Integration solcher Tools gibt und wie sich die um CS erweiterten Pipelines von klassischen CI/CD-Pipelines unterscheiden.

4.1 Implementierungsansätze

Nachfolgend werden Ansätze für die Implementierung von CS in einer CI/CD-Pipeline vorgestellt. Außerdem wird die Wirksamkeit der beschriebenen Sicherheitsmaßnahmen untersucht.

4.1.1 Vergleich verschiedener DevOps-Umgebungen

Mit der stetigen Weiterentwicklung automatisierter DevOps-Lösungen im Bereich der Software- und Anwendungsentwicklung haben sich auch die Anforderungen von Unternehmen und Softwareanbietern an geeignete DevOps-Infrastrukturen deutlich erhöht. Mittlerweile existieren unzählige Anwendungen, mit denen CI/CD-Pipelines konfiguriert und ausgeführt werden können. Neben reinen CI/CD-Anwendungen, beispielsweise *Circle CI*, *Jenkins* oder *Travis CI*, existieren auch vollintegrierte DevSecOps-Plattformen wie *Azure DevOps*, *Bitbucket*, *GitHub* oder *GitLab*.

Solche DevOps-Umgebungen können entweder als Cloud-basierte *Software as a Service (SaaS)*-Lösung oder als selbst gehostete *On-Premise*-Lösung genutzt werden. Bei der Entwicklung einer DevSecOps-Pipeline ist es von besonderer Bedeutung, dass die gewählte Anwendung in einer sicheren Umgebung ausgeführt wird, um die Verwaltung von Softwarecode und die isolierte Ausführung von CI/CD-Pipelines zu gewährleisten.

Daher muss für die Wahl der DevOps-Umgebung ein Anforderungskatalog erstellt werden, nach dessen Kriterien die einzelnen Plattformen und Integrationstechniken untersucht und bewertet werden. Folgende Kriterien können zum Beispiel nützlich sein (nach Zeeshan (2020)):

- **Datenschutz**: Sichere und datenschutzkonforme Speicherung von Softwarecode und Build-Artefakten
- **Integrität**: Kontrolle über Ausführung, Sicherung und Wiederherstellung der gespeicherten Daten

- **Isolation**: Ausführung von CI/CD-Pipelines in isolierten Umgebungen (Container, virtuelle Maschinen)
- **Skalierbarkeit**: Anpassung der Plattform-Infrastruktur entsprechend der Unternehmensanforderungen
- **Verfügbarkeit**: Hohe Erreichbarkeit der Plattform, um das Prinzip der Kontinuität zu gewährleisten
- **Schnittstellen**: Möglichkeit der Anbindung an externe Anwendungen zur Effizienzsteigerung während des gesamten SDLC
- **Kosten**: Überschaubare Kosten für Hosting, Wartung und Monitoring der DevOps-Plattform

Für die nachfolgende beispielhafte Umsetzung von CS wird der Fokus auf einer für *GitLab CI* entwickelten Pipeline liegen. GitLab bezeichnet sich selbst als „the leading DevSecOps platform“ (GitLab, 2024c) und erfüllt viele der genannten Kriterien, sowohl in der SaaS- als auch in der On-Premise-Lösung. Die Plattform stellt verschiedene Features wie *IaC Scanning*, *SAST*, *Secret Detection*, *License Scanning*, *Dependency Scanning*, *Fuzz Testing*, *Container Scanning* oder *DAST* nativ zur Verfügung (GitLab, 2024a). Es muss jedoch bedacht werden, dass einige Features wie *DAST* oder *Dependency Scanning* nicht kostenfrei nutzbar sind.

4.1.2 Tools zur Umsetzung der Sicherheitsmaßnahmen

Die Anwendung der in Abschnitt 3.2 beschriebenen Sicherheitsmaßnahmen erfolgt mithilfe verschiedener Tools, die zum Teil als *Free Open Source Software (FOSS)* frei verfügbar sind. Es existieren jedoch auch proprietäre Softwarelösungen, die in der Regel als SaaS nutzbar sind. Nachfolgend werden einige Tools aufgelistet, die für den Einsatz der genannten Maßnahmen infrage kommen, wobei der Fokus auf Open Source-Software liegt, da sich diese in der Regel leichter und kostengünstiger in bestehende CI/CD-Pipelines integrieren lässt. Die Zusammenstellung ist abgeleitet aus Feio et al. (2024), Hsu (2018), Kumar und Goyal (2021) und Lombardi und Fanton (2023); proprietäre Tools sind *kursiv* dargestellt:

- **Software Composition Analysis (SCA)**:
  FOSSA, OWASP Dependency-Check, Retire.js, *Snyk*
- **Static Application Security Testing (SAST)**:
  Actuary, Bandit, Dagda, GitHound, PMD, Semgrep, SonarQube, SpotBugs
- **Dynamic Application Security Testing (DAST)**:
  Metasploit, Nmap, Open Vulnerability Assessment Scanner (OpenVAS), OWASP Zed Attack Proxy (ZAP), Wapiti

- **Interactive Application Security Testing (IAST)**:
  Contrast Security, *Veracode*
- **Erweiterte Unit- und Integrationstests**:
  JUnit, Mocha, RSpec, TestNG, xUnit
- **Secrets-Management**:
  Amazon Web Services (AWS) Key Management Service (KMS), Ansible Vault, chef-vault, Docker secrets, git-secrets, HashiCorp Vault
- **Container-Sicherheit**:
  Clair, Docker Bench for Security, Grype, Syft, Trivy
- **Maßnahmen für Infrastructure as Code (IaC)**:
  Ansible Lint, Chef InSpec, Cloud Custodian, Keeping Infrastructure as Code Secure (KICS), Puppet Lint

Die Auswahl einzelner Tools hängt von vielen Faktoren ab. Dazu zählen etwa die Programmiersprache des Softwarecodes, System- und infrastrukturelle Anforderungen an die Software oder Anwendung, aber auch die Auswahl der möglicherweise eingesetzten IaC-Software und der gewählten DevSecOps-Plattform. Darüber hinaus ist es nicht immer notwendig oder sinnvoll, für alle genannten Sicherheitsmaßnahmen eigene Tools in die Pipeline zu integrieren.

4.1.3 Wirksamkeit der Sicherheitsmaßnahmen

Häufig können mit einzelnen Tools mehrere Sicherheitsmaßnahmen kombiniert werden. Außerdem unterscheiden sich die einzelnen Maßnahmen in ihrem Wirkungsgrad teilweise deutlich. Das bedeutet, dass es nicht für jede Software sinnvoll ist, in ihrem Entwicklungsprozess immer zwingend alle genannten Sicherheitsmaßnahmen konsequent umzusetzen, wenn die eingesetzten Tools ihre Wirkung nicht entfalten können. Dies würde im schlimmsten Fall zu Performanceeinbußen bei der Ausführung der Pipeline und zu einem verlorenen Sicherheitsbewusstsein bei den Anwender*innen führen.

Maßnahme	Wirksamkeit	Kommentar
SCA	sehr hoch	v.a. bei großem Abhängigkeitsbaum
SAST	sehr hoch	leicht bedienbar, einfach zu integrieren
DAST, **IAST**	hoch	insbesondere bei Web-Anwendungen
Erweiterte Unit- und Integrationstests	mittel	häufig erst nach erstmaligem Exponieren einer Schwachstelle
Secrets-Management	sehr hoch	unerlässlich zum Schutz sensibler Daten
Container-Sicherheit	sehr hoch	insbesondere in Produktivumgebungen
Maßnahmen für IaC	sehr hoch	notwendig für sichere Infrastruktur

Tabelle 1: Wirksamkeit einzelner Sicherheitsmaßnahmen einer CI/CD-Pipeline

- **SCA** zeugt vor allem beim Einsatz vieler externer Abhängigkeiten von einer **sehr hohen Wirksamkeit**. Moderne Software enthält meist Abhängigkeiten zu externen Bibliotheken, die wiederum eigene Abhängigkeiten definieren. So entsteht alleine durch diese transitiven Abhängigkeiten ein großer, häufig sehr unübersichtlicher Abhängigkeitsbaum (*dependency tree*). SCA-Tools sind in der Lage, diesen Baum korrekt aufzulösen und alle eingesetzten Abhängigkeiten auf bekannte Schwachstellen zu prüfen. Bei Open Source-Software ist zudem wichtig, dass die Lizenzen der verwendeten Abhängigkeiten mit der Lizenz der Software konform sind. Auch hier entfaltet SCA seine hohe Wirkung, indem es das Lizenzmanagement übernimmt (Dietrich et al., 2023).

- **SAST** stellt als statisches Analyseverfahren ein relativ einfaches und leicht zu bedienendes Werkzeug für die Analyse von Softwarecode dar. Statische Code-Scanning-Technologien gelten gemeinhin auch als die effektivsten Methoden zur Identifizierung von Sicherheitsproblemen, und das schon in einer frühen Phase des SDLC (Hsu, 2018). Die Wirksamkeit von SAST kann daher generell für die meisten Anwendungsfälle als **sehr hoch** angesehen werden. Häufig werden SAST-Tools daher schon bei der Entwicklungsphase mittels entsprechender Plugins in das *Integrated Development Environment (IDE)* integriert.

- **DAST** und **IAST** erfordern – wie jede andere Software, mit der Black-Box- und Gray-Box-Tests durchgeführt werden – etwas mehr Aufwand in der Implementierung und benötigen in der Regel auch mehr Zeit für die Durchführung der Tests. Nichtsdestotrotz ist die Durchführung von DAST und IAST ein sehr wichtiger und kritischer Teilschritt im SDLC, da sie Schwachstellen sichtbar macht, die nicht direkt mittels statischer Code-Analyse im Softwarecode erkennbar sind (Rangnau et al., 2020). Somit kann DAST und IAST im Anwendungskontext ebenfalls eine **hohe Wirksamkeit** zugeschrieben werden. Vor allem bei Web-Anwendungen ist die Nutzung sehr sinnvoll und zeigt in vielen Fällen auch schnell eine hohe Wirksamkeit.

- **Erweiterte Unit- und Integrationstests** müssen abhängig vom zu testenden Softwarecode entwickelt werden und sind deshalb in der Entwicklungs- und Wartungsphase aufwändiger als andere Tools. Einzelne Testfälle können aus den Ergebnissen dynamischer Tests wie DAST oder aus den Monitoringwerten einer beispielsweise eingesetzten WAF abgeleitet werden (Zeeshan, 2020). Für CS zeigen sie daher eine eher **mittlere Wirksamkeit**, da zum Zeitpunkt der Architektur häufig bereits Schwachstellen exponiert wurden und die Entwicklung von Testfällen lediglich ein erneutes Exponieren derselben Schwachstelle verhindert.

- **Secrets-Management** ist eine sehr wichtige Sicherheitsmaßnahme, um die versehentliche Veröffentlichung geheimer Informationen zu verhindern. Häufig kön-

nen offengelegte Passwörter, API-Token und andere Sicherheitsmerkmale zu einer schnellen und folgenschweren Manipulation von Unternehmensdaten führen, die in vielen Fällen auch hohe finanzielle Einbußen nach sich ziehen können (Hughes, 2022). Ein gutes Secrets-Management ist daher unerlässlich und kann gemeinhin als eine Maßnahme von **sehr hoher Wirksamkeit** angesehen werden.

- **Container-Sicherheit** als eine weitere statische Sicherheitsmaßnahme ist ähnlich wirksam wie SAST. Der Fokus der Analyse liegt hierbei jedoch auf containerisierten Artefakten wie zum Beispiel Docker-Images. Da diese Container häufig auch in Produktivumgebungen eingesetzt werden, ist es besonders wichtig, alle verwendeten Images umfassend auf Schwachstellen zu untersuchen (Zeeshan, 2020). Geschieht dies frühzeitig im SDLC, kann das Scannen und Analysieren von Containern eine **sehr hohe Wirksamkeit** entfalten.

- **Maßnahmen für IaC** sind in Kombination mit dem Secrets-Management unerlässliche Werkzeuge, um Produktivumgebungen ausreichend abzusichern. Beim Einsatz von IaC ist es besonders wichtig, zu prüfen, ob die definierten Konfigurationseinstellungen für die einzelnen Infrastrukturen keine Sicherheitslücken enthalten. Da Produktivumgebungen in der Regel direkten Zugriffen der Endanwender ausgesetzt sind, zeigt sich schon beim Erkennen kleinerer Fehlkonfigurationen eine **sehr hohe Wirksamkeit** (Hughes, 2022).

4.2 Herausforderungen

Jede noch so wirksame Sicherheitsmaßnahme birgt auch Risiken und stellt Herausforderungen bei Integration und Anwendung innerhalb einer CI/CD-Pipeline. Daher werden nachfolgend einige Herausforderungen benannt, die es bei der praktischen Umsetzung von CS-Maßnahmen zu beachten gibt.

4.2.1 Etablierung einer notwendigen Sicherheitskultur

Damit der Übergang von DevOps zu DevSecOps gelingen kann, müssen alle beteiligten Entwickler*innen und Anwender*innen in die Notwendigkeit der Maßnahmen eingeführt werden. Es gilt, ein gemeinsames Verständnis für Sicherheit und die daraus resultierenden Maßnahmen zu entwickeln, und aus diesem Verständnis heraus eine weit verbreitete Sicherheitskultur zu etablieren. Nur so kann der Shift-Left-Security-Ansatz konsequent realisiert werden. Wird diese Kultur nicht gefördert oder gelingt es nicht, sie zu fördern, kann die Implementierung von Sicherheitsmaßnahmen zu einer großen Herausforderung werden. Die Maßnahmen können dann aufgrund ihrer Komplexität als zusätzliche Belastung angesehen werden, was im schlimmsten Fall dazu führen kann, dass sie von Anwender*innen umgangen oder nicht ausreichend priorisiert behandelt werden.

Eine Sicherheitskultur kann etabliert werden, indem allen am SDLC beteiligten Mitarbeiter*innen geeignete Schulungen angeboten werden. Es erfordert umfangreiche Investitionen, um Entwickler*innen, Software-Architekt*innen und anderen Beteiligten die notwendige Sicherheitskompetenz näher zu bringen. Sinnvoll ist außerdem der Aufbau eines eigenen Security-Teams, welches sich um die Definition aller notwendigen Sicherheitsrichtlinien und -maßnahmen kümmert und deren Implementierung koordiniert (Lombardi & Fanton, 2023).

4.2.2 Umgang mit False-Positives & False-Negatives

Der bereits erwähnte *false sense of security* ist eine direkte Konsequenz aus der Herausforderung mit dem Umgang von *False-Positives*. Ein False-Positive entsteht, wenn eine Sicherheitsmaßnahme eine Schwachstelle oder Sicherheitslücke meldet, die in Wahrheit gar nicht vorhanden ist. Häufen sich diese Meldungen, kann dies dazu führen, dass Anwender*innen einzelne Sicherheitsmaßnahmen umgehen oder sie mit weniger Priorität behandeln. Vor allem SAST ist anfällig für False-Positives, aber auch DAST zeigt Schwächen bei der fehlerfreien Schwachstellenanalyse. Es hat sich gezeigt, dass vor allem IAST eine niedrige False-Positive-Rate aufweist, weshalb es bei der Implementierung ggf. anderen Sicherheitsmaßnahmen vorzuziehen ist (Pan, 2019).

Demgegenüber stehen *False-Negatives* – Schwachstellen und Bedrohungen, die von keinem Analysetool erkannt werden. Eine hohe False-Negative-Rate stellt eine weitaus größere Herausforderung als eine hohe False-Positive-Rate dar, da jede Sicherheitsmaßnahme nur so gut wie ihre Implementierung ist. Häufig hängt eine hohe Erkennungsrate auch von externen Faktoren wie der Pflege bekannter Schwachstellen in Schwachstellen-Datenbanken ab (Ponta et al., 2018). Daher kann es sinnvoll sein, unterschiedliche Tools und verschiedene Datenquellen für die Schwachstellenanalyse zu nutzen, um die Schwächen einzelner Anwendungen mit den Stärken anderer Anwendungen auszugleichen.

4.2.3 Sicherstellung von Agilität und Performance

Das US-amerikanische Marktforschungsinstitut *Gartner Inc.* hat in einer 2015 durchgeführten Befragung festgestellt, dass über drei Viertel der Befragten in der Anwendung von Sicherheitsmaßnahmen ein Hemmnis der Agilität und Performance von DevOps-Prozessen sehen (MacDonald & Head, 2016). Ähnlich wie bei dem Umgang mit False-Positives kann auch dies dazu führen, dass Anwender*innen Sicherheitsmaßnahmen einfach deaktivieren oder anderweitig umgehen könnten, damit die Agilität und Performance ihrer CI/CD-Pipelines weiterhin gewährleistet wäre.

Die zugehörige von MacDonald und Head (2016) veröffentlichte Forschungsarbeit liefert einige Beispiele, um DevSecOps-Prozesse agil und performant durchzuführen:

- Limitierung auf differentielle Scans von geändertem Softwarecode und nachgelagerten Modulen bei Durchführung von SAST und DAST

- Präferierte Durchführung von IAST gegenüber SAST und DAST
- Reduktion der False-Positive-Rate, selbst wenn dies zu einem höheren Risiko von False-Negatives führt
- Fokus der Scan-Resultate auf Probleme mit dem höchsten Schweregrad
- Automatisierung aller Sicherheitsmaßnahmen

4.3 Beispielhafte Umsetzung

In dem nachfolgenden Beispiel wird eine bestehende CI/CD-Pipeline um die zuvor benannten Sicherheitsmaßnahmen erweitert. Als Vorlage dient die in Abschnitt 2.2 vorgestellte Pipeline.

Die Pipeline wird mittels GitLab CI ausgeführt. In GitLab erfolgt die Konfiguration von Pipelines mittels YAML-Dateien – in der Regel als `.gitlab-ci.yml`-Datei im Wurzelverzeichnis des Softwarecode-Repositories. In jeder Pipeline-Stage sind sog. *Jobs* definiert, in denen einzelne Skripte ausgeführt werden.

4.3.1 Auswahl der notwendigen Pipeline-Stages

Einzelne Jobs werden in GitLab-Pipelines anhand der Reihenfolge ihrer zugehörigen Stages ausgeführt. Stages sind in der Regel voneinander abhängig, d.h. ein Job wird nur ausgeführt, wenn alle Jobs der vorhergehenden Stages erfolgreich waren.

Dies ist insbesondere für die Durchführung der Integrationstests von Bedeutung: Da diese eine lauffähige Anwendung voraussetzen, muss vor Ausführung der Tests sichergestellt sein, dass mittels CD eine Auslieferung der Anwendung auf das Testsystem erfolgt ist. Daher wird eine zusätzliche Stage *Integrate* eingeführt, die nach der *Release*-Stage ausgeführt wird:

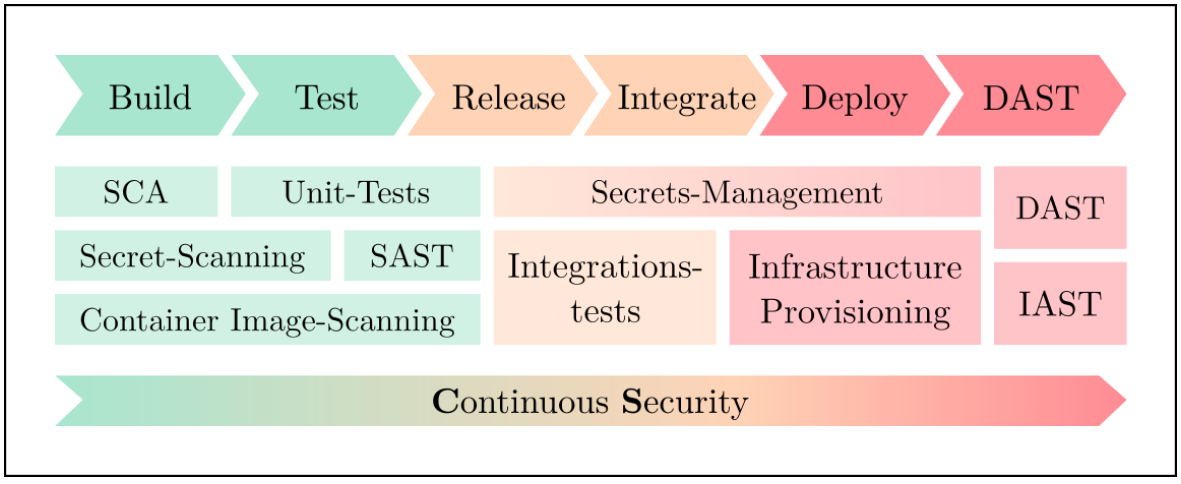

Abbildung 2: Aufbau einer um CS-Maßnahmen erweiterten CI/CD-Pipeline

Aus dem in Abbildung 2 dargestellten Pipeline-Schema wird ersichtlich, dass zusätzlich eine Stage für die Ausführung von DAST (und IAST) eingeführt wird. Dies ist notwendig, da die Ausführung von DAST auf Basis einer von GitLab bereitgestellten Vorlage erfolgt, welche die Ausführung innerhalb dieser Stage erfordert (siehe Abschnitt 4.3.2). Zusammenfassend ergibt sich folgende Konfiguration der einzelnen Stages in der `.gitlab-ci.yml`-Datei:

```
stages:
  - build
  - test
  - release
  - integrate
  - deploy
  - dast
```

Listing 1: Definition von Pipeline-Stages in `.gitlab-ci.yml`

4.3.2 Einbindung von Vorlagen für einzelne Sicherheitsmaßnahmen

Wie bereits erwähnt, ermöglicht GitLab die Nutzung vordefinierter Skripte für die Ausführung einzelner Sicherheitsmaßnahmen. Diese können als Vorlagen in die Pipeline-Konfiguration eingebunden werden. GitLab bietet unter anderem Vorlagen für SCA, SAST, DAST, Secret-Scanning und Container-Scanning an. Diese werden wie folgt eingebunden:

```
include:
  - template: Jobs/Container-Scanning.gitlab-ci.yml
  - template: Jobs/Dependency-Scanning.gitlab-ci.yml
  - template: Jobs/SAST.gitlab-ci.yml
  - template: Jobs/SAST-IaC.gitlab-ci.yml
  - template: Jobs/Secret-Detection.gitlab-ci.yml
  - template: Security/DAST.gitlab-ci.yml
```

Listing 2: Einbindung von Job-Vorlagen in `.gitlab-ci.yml`

Neben den hier erwähnten Vorlagen bietet GitLab noch erweiterte Vorlagen für Sicherheitsmaßnahmen an, die ebenfalls in die Pipeline integriert werden können. Einige dieser Vorlagen sind jedoch nicht kostenfrei nutzbar.

4.3.3 Definition eigener Jobs für weitere Sicherheitsmaßnahmen

GitLab stellt nicht für jede notwendige Sicherheitsmaßnahme eine Vorlage zur Verfügung. Darüber hinaus kann es durchaus auch gewünscht sein, nicht die Standard-Vorlagen von GitLab zu verwenden, sondern auf andere Tools zurückzugreifen. In diesen Fällen müssen eigene Jobs definiert werden, in denen die entsprechenden Tools aufgerufen werden.

In dieser beispielhaften Umsetzung wird etwa ein zusätzlicher Scan (sog. *Security-Gate*) mithilfe der proprietären Anwendung *Contrast Security* in die Pipeline eingefügt. Dies erlaubt zum Beispiel die Ausführung von IAST, direkt nachdem ein Deployment in die Produktivumgebung erfolgt ist:

```
93  security_gate:
94    image: ghcr.io/contrast-security-oss/integration-verify:latest
95    stage: dast
96    variables:
97      API_KEY: $CONTRAST_API_KEY
98      API_URL: $CONTRAST_API_URL
99      ORG_ID: $CONTRAST_ORG_ID
100     AUTH_HEADER: $CONTRAST_AUTH_HEADER
101     APP_NAME: $CI_PROJECT_NAME
102     BUILD_NUMBER: $CI_COMMIT_SHORT_SHA
103   script:
104     - /usr/bin/env python3 /verify.py
105   rules:
106     - if: $CI_COMMIT_TAG
```

Listing 3: Definition eines *Security-Gate*-Jobs in `.gitlab-ci.yml`

4.4 Ergebnisse

Dank des breiten Feature-Spektrums von GitLab ist es innerhalb kürzester Zeit möglich, eine bestehende CI/CD-Pipeline zu einer DevSecOps-Pipeline zu erweitern. Das einfache Konfigurationsschema von GitLab CI ermöglicht zudem das systematische Hinzufügen eigener Jobs, um zusätzliche Sicherheitsmaßnahmen in die Pipeline zu integrieren. Je nach verwendetem Preismodell bietet GitLab neben der üblichen Pipeline-Ansicht (siehe Abbildung 3) auch eine erweiterte Ansicht an, bei der die Ergebnisse der einzelnen Tools abgebildet werden. Eine solche übersichtliche Darstellung fördert insbesondere den Aspekt der Agilität, der von performanten DevSecOps-Pipelines gefordert wird.

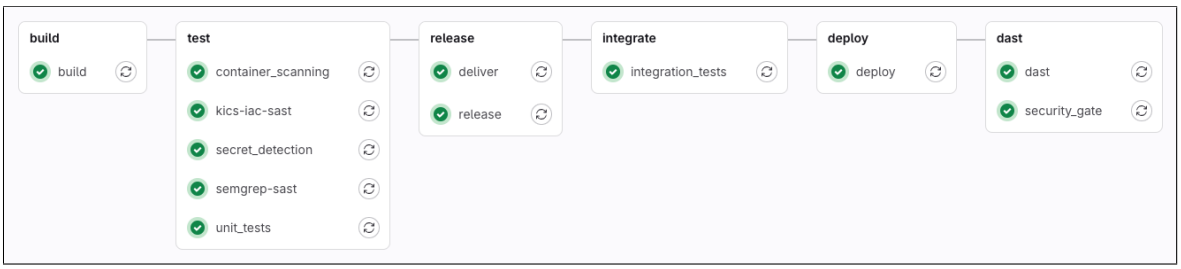

Abbildung 3: Ansicht einer einfachen DevSecOps-Pipeline in GitLab

5 Zusammenfassung

5.1 Ergebnisse

DevSecOps und im Speziellen die Anwendung von Continuous Security in CI/CD-Pipelines sind bewährte Konzepte, mit denen die Entwicklung und der Betrieb von Software sicherer gestaltet werden kann. Dies zeigt sich allein schon durch die Masse an verfügbaren Sicherheitsmaßnahmen wie SCA, SAST und DAST. Konzepte wie IaC verdeutlichen zudem die vielfältigen Möglichkeiten zur Etablierung eines Shift-Left-Security-Ansatzes im gesamten SDLC. Die Wirksamkeit vieler Sicherheitsmaßnahmen ist sehr hoch, wobei niedrigere Wirkungsgrade einzelner Maßnahmen häufig durch die Vielfalt und den Funktionsumfang anderer Maßnahmen ausgeglichen werden können.

Der große Funktionsumfang von DevSecOps-Plattformen und die Vielzahl an vorhandenen Tools ermöglicht eine einfache Integration von Sicherheitsmaßnahmen in bestehende CI/CD-Pipelines. GitLab etwa bietet für viele Maßnahmen bereits unterschiedlich umfangreiche Vorlagen an, die einfach in bestehende DevOps-Prozesse integriert werden können. Zwar entstehen durch die Integration der Sicherheitsmaßnahmen zusätzliche Pipeline-Stages und dadurch potenziell höhere Laufzeiten, jedoch ermöglicht die flexible Anordnung und Konfiguration einzelner Jobs auch eine asynchrone Ausführung, wodurch die Laufzeiten idealerweise niedrig bleiben.

5.2 DevSecOps und das CAMS-Modell

Die Akzeptanzkriterien, die sich aus dem CAMS-Modell ableiten, können auch auf den DevSecOps-Bereich ausgeweitet werden und finden dort bereits Anwendung. Dies zeigt etwa der im Juni 2024 von GitLab auf Basis einer zuvor durchgeführten Umfrage veröffentlichte *Global DevSecOps Report*. Dort wurde Sicherheit als das Feld mit der höchsten Priorität für laufende Unternehmensinvestionen angegeben. Auch dem Bereich der Automatisierung wird mit Platz 4 eine hohe Priorität zugewiesen, wobei sogar 67% der Befragten angaben, ihren gesamten SDLC komplett automatisiert oder zumindest teilautomatisiert zu haben. Im Bereich *Culture* zeigt der Bericht zudem deutlich, dass sich in kurzer Zeit die für DevSecOps notwendige Sicherheitskultur etabliert hat. Die eingangs aufgeführte These des Gleichgewichts aus Sicherheit und Performance kann in dem Bericht hingegen nicht bestätigt werden. Ein Großteil der befragten Sicherheits- und Betriebsteams gaben an, dass sie ein Viertel oder mehr ihrer Zeit für die Wartung und Integration der notwendigen Maßnahmen aufbringen müssen (GitLab, 2024b).

5.3 Herausforderungen

Die Integration eines gesamten, eigenständigen Bereichs in bestehende Strukturen ist mitunter kein leichtes Unterfangen und bringt allein durch die Komplexität der ein-

zelnen Prozesse viele Herausforderungen mit sich. In dieser Arbeit wurden die Herausforderungen bei der Integration von Sicherheitsmaßnahmen in bestehende CI/CD-Pipelines untersucht. Es wurde deutlich, dass vor allem die Etablierung einer notwendigen Sicherheitskultur und die Sicherstellung von Agilität und Performance Faktoren sind, die mitunter nicht nur auf einer technischen Ebene lösbar sind, sondern auch den Faktor Mensch miteinbeziehen müssen. Der von GitLab vorgestellte Bericht zeigt allerdings, dass in vielen Unternehmen bereits ein gestärktes Sicherheitsverständnis vorhanden ist und sich neben Sicherheitsexpert*innen auch Entwickler*innen und Software-Architekt*innen der Notwendigkeit geeigneter Sicherheitsmaßnahmen bewusst sind (GitLab, 2024b).

Ein im Zusammenhang mit Sicherheit häufig genannter Begriff ist darüber hinaus der *false sense of security*. Er entsteht, wenn eine hohe False-Positive-Rate aus der Durchführung von Sicherheitsmaßnahmen resultiert. Um False-Positives und auch False-Negatives möglichst gering zu halten, empfiehlt sich eine sorgfältige Auswahl und Kombination der eingesetzten Maßnahmen und Tools.

5.4 Ausblick

Die vorgestellten Maßnahmen und Tools bieten einen einfachen Einstieg in DevSecOps-Pipelines. Sie sind keineswegs vollständig und können auch nicht als *Best Practice* angesehen werden. Vielmehr verdeutlichen sie, inwieweit Sicherheit in bestehenden DevOps-Prozessen eine Rolle spielt und welche Auswirkungen sie auf bestehende Prozesse haben kann. Schon jetzt zeigt sich, dass ein entscheidender Faktor die gesamte DevOps- und DevSecOps-Welt nachhaltig bereichern könnte: Künstliche Intelligenz (KI).

KI wird in dem vorgestellten GitLab-Bericht auf Platz 2 der Top-Investitionen geführt. 78% der Befragten gaben an, bereits KI-Tools zur Software-Entwicklung zu nutzen oder in den nächsten zwei Jahren planen, dies zu tun (GitLab, 2024b). KI kann für alle Bereiche im DevOps- und DevSecOps-Umfeld ein relevanter und möglicherweise bald auch führender Bestandteil werden. Yadav et al. (2022) stellen etwa ein Verfahren vor, bei dem mittels Machine Learning (ML) Modelle trainiert werden, die in DevSecOps-Prozesse integriert werden können. Dadurch sollen einfachere Analysen und weniger fehleranfällige Resultate erzielt werden. Bannon und Laplante (2024) zeigen auf, welche Möglichkeiten zur Integration von KI es bereits jetzt gibt und welche Herausforderungen der Einsatz von KI in DevSecOps mit sich bringt.

Auch aktuelle und vergangene Sicherheitsvorfälle zeigen: DevOps ohne *Sec* scheint ein Konzept der Vergangenheit zu sein. Um mit den wachsenden Sicherheitsrisiken Stand halten zu können, benötigt es ein weit verbreitetes Verständnis für die Notwendigkeit ausgeklügelter DevSecOps-Prozesse. Vorfälle wie das *Equifax*-Datenleck oder der *Heartbleed*-Bug müssten allein schon durch die Einfachheit ihrer zugrunde liegenden Sicherheitslücken in der heutigen Zeit niemals wieder auftreten.

Literaturverzeichnis

Bannon, T. T., & Laplante, P. (2024). Infusing Artificial Intelligence Into Software Engineering and the DevSecOps Continuum. *Computer (Long Beach, Calif.)*, *57*(9), 140–148.

Dietrich, J., Rasheed, S., Jordan, A., & White, T. (2023). On the Security Blind Spots of Software Composition Analysis. *arXiv preprint arXiv:2306.05534*.

Feio, C., Santos, N., Escravana, N., & Pacheco, B. (2024). An Empirical Study of DevSecOps Focused on Continuous Security Testing. *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 610–617.

GitLab. (2024a). Application security | GitLab [Online; abgerufen am 6. September 2024]. https://docs.gitlab.com/ee/user/application_security/

GitLab. (2024b). GitLab 2024 Global DevSecOps Report | GitLab [Online; abgerufen am 20. September 2024]. https://about.gitlab.com/developer-survey/

GitLab. (2024c). The most-comprehensive AI-powered DevSecOps platform | GitLab [Online; abgerufen am 5. September 2024]. https://about.gitlab.com

Hanschke, I. (2017). *Agile in der Unternehmenspraxis: Fallstricke Erkennen und Vermeiden, Potenziale Heben* (1. Aufl.). Springer Fachmedien Wiesbaden.

Hebing, M., & Manhembué, M. (2024). *Data Science Management: Vom Ersten Konzept Bis Zur Governance Datengetriebener Organisationen* (1. Aufl.). O'Reilly Verlag GmbH & Co. KG.

Hsu, T. (2018). *Hands-on security in DevOps: Ensure continuous security, deployment, and delivery with DevSecOps* (1st edition). Packt.

Hughes, C. (2022). Keeping secrets in a devsecops cloud-native world. *CSO (Online)*.

Kağızmandere, Ö., & Arslan, H. (2024). Vulnerability analysis based on SBOMs: A model proposal for automated vulnerability scanning for CI/CD pipelines. *International journal of information security science*, *13*(2), 33–42.

Kumar, R., & Goyal, R. (2021). When Security Meets Velocity: Modeling Continuous Security for Cloud Applications using DevSecOps. In *Innovative Data Communication Technologies and Application* (S. 415–432, Bd. 59). Springer Singapore Pte. Limited.

Lombardi, F., & Fanton, A. (2023). From DevOps to DevSecOps is not enough. CyberDevOps: an extreme shifting-left architecture to bring cybersecurity within software security lifecycle pipeline. *Software quality journal*, *31*(2), 619–654.

MacDonald, N., & Head, I. (2016). DevSecOps: How to seamlessly integrate security into DevOps. *Gartner Research*.

Malone, Z. (2023). What Executives Should Know About Shift-Left Security. *CIO*.

Myrbakken, H., & Colomo-Palacios, R. (2017). DevSecOps: A Multivocal Literature Review. *Software Process Improvement and Capability Determination*, 17–29.

Pan, Y. (2019). Interactive Application Security Testing. *2019 International Conference on Smart Grid and Electrical Automation (ICSGEA)*, 558–561.

Ponta, S. E., Plate, H., & Sabetta, A. (2018). Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 449–460.

Rangnau, T., Buijtenen, R. v., Fransen, F., & Turkmen, F. (2020). Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines. *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*, 145–154.

Schreider, T. (2020). *Building an effective cybersecurity program* (Second edition.). Rothstein Publishing.

Stanciu, A.-M. (2023). Theoretical Study of Security for a Software Product. In *Intelligent Sustainable Systems* (S. 233–242, Bd. 578). Springer.

Yadav, B., Choudhary, G., Shandilya, S. K., & Dragoni, N. (2022). AI Empowered DevSecOps Security for Next Generation Development. In *Frontiers in Software Engineering* (S. 32–46, Bd. 1523). Springer International Publishing AG.

Yang, E. (2018). Fuzz testing & software composition analysis in software engineering. *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 1–3.

Zeeshan, A. A. (2020). *DevSecOps for .NET Core: Securing Modern Software Applications* (1st Edition). Apress L. P.

Eigenständigkeitserklärung

Ich trage die Verantwortung für die Qualität des Textes sowie die Auswahl aller Inhalte und habe sichergestellt, dass Informationen und Argumente mit geeigneten wissenschaftlichen Quellen belegt bzw. gestützt werden. Die aus fremden Quellen direkt oder indirekt übernommenen Texte, Gedankengänge, Konzepte, Grafiken usw. in meinen Ausführungen habe ich als solche eindeutig gekennzeichnet und mit vollständigen Verweisen auf die jeweilige Quelle versehen. Alle weiteren Inhalte dieser Arbeit (Textteile, Abbildungen, Tabellen etc.) ohne entsprechende Verweise stammen im urheberrechtlichen Sinn von mir.

Hiermit erkläre ich, dass ich die vorliegende Studienarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle sinngemäß und wörtlich übernommenen Textstellen aus fremden Quellen wurden kenntlich gemacht.

Die vorliegende Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

Erklärung zu (gen)KI-Tools

Verwendung von (gen)KI-Tools

Ich versichere, dass ich mich (gen)KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Ich verantworte die Übernahme jeglicher von mir verwendeter Textpassagen vollumfänglich selbst. In der Übersicht verwendeter (gen)KI-Tools habe ich sämtliche eingesetzte (gen)KI-Tools, deren Einsatzform sowie die jeweils betroffenen Teile der Arbeit einzeln aufgeführt. Ich versichere, dass ich keine (gen)KI-Tools verwendet habe, deren Nutzung der Prüfer bzw. die Prüferin explizit schriftlich ausgeschlossen hat.

Hinweis: Sofern die zuständigen Prüfenden bis zum Zeitpunkt der Ausgabe der Aufgabenstellung konkrete (gen)KI-Tools ausdrücklich als nicht anzeige-/kennzeichnungspflichtig benannt haben, müssen diese nicht aufgeführt werden.

Ich erkläre weiterhin, dass ich mich aktiv über die Leistungsfähigkeit und Beschränkungen der unten genannten (gen)KI-Tools informiert habe und überprüft habe, dass die mithilfe der genannten (gen)KI-Tools generierten und von mir übernommenen Inhalte faktisch richtig sind.

Übersicht verwendeter (gen)KI-Tools

Die (gen)KI-Tools habe ich, wie im Folgenden dargestellt, eingesetzt.

(gen)KI-Tool	**Einsatzform**	**Betroffene Teile der Arbeit**
ChatGPT	Generierung von ersten Ideen für eine geeignete Gliederung	Gesamte Arbeit
	Textanalyse selbstständig formulierter Texte	Abschnitt 1.1
DeepL	Übersetzung einzelner Textpassagen englischer Literatur zum besseren Verständnis	Gesamte Arbeit

Münster, 20. September 2024

Elias Häußler