Seminar Softwareentwicklung mit Dev(Sec)Ops

Dynamic Application Security Testing (DAST)

Tobias Kiehnlein

Inhaltsverzeichnis

1 Einleitung

Mit der Verbreitung agiler Methoden und moderner Softwareentwicklung wird stets das Ziel verfolgt in kurzen Abständen Änderungen am Code vorzunehmen und diesen zu veröffentlichen. Nicht selten ist die Veröffentlichung einer Software ein fehleranfälliger und/oder komplexer Prozess, was zur Folge hat, dass Release-Zyklen länger werden, Deployments eine unangenehme Last werden und Sicherheit oftmals völlig außer Acht gelassen wird.

Gerade deshalb sollte man sich über die Prozesse Gedanken machen, die nach der Implementation des Codes stehen. Dies stellt allerdings keine triviale Aufgabe dar, weshalb sich die Frage stellt, warum dieser Aufwand in Kauf genommen werden sollte. Auch wenn hier viele Faktoren eine Rolle spielen, wir sich diese Arbeit auf die Anwendungssicherheit fokussieren. Nicht selten haben Entwickler keine oder nur wenig Kenntnisse im Bereich der IT-Sicherheit. Nicht grundlos befinden sich seit Jahren bekannte Sicherheitslücken wie SQL-Injections, XSS oder XSRF immer noch unter den häufigsten Sicherheitslücken im Web.[1]

Wie lässt sich nun ein derart fundamentales Problem in der IT Industrie lösen? Um eine Software bestmöglich vor Sicherheitslücken zu schützen, bleibt nur die Option, diese regelmäßig auf Schwachstellen zu untersuchen, um gerade häufigen und leicht zu findenden Sicherheitslücken vorbeugen zu können. Dies bringt allerdings einen hohen Kostenfaktor in das Projekt, da Sicherheitsexperten zumeist relativ teuer sind. Dies wird dadurch verstärkt, dass einige triviale Tests wiederholt durchgeführt werden müssen, da kleinste Codeänderungen schnell große Auswirkungen im gesamten Projekt haben können.

Eine einfache Lösung stellen daher automatisierte Sicherheitstests dar, welche die Software auf die gängigsten und bekanntesten Lücken prüft. Diese können daraufhin direktes Feedback an Entwickler und Sicherheitsexperten geben, um Entwickler auf mögliche Sicherheitslücken zu sensibilisieren und Sicherheitsexperten um repetitive Arbeit zu entlasten. Somit können Zeit, Geld und Nerven gespart und eine sichere und effiziente Softwareentwicklung gewährleistet werden.

2 Analyse eines optimalen DevSecOps Prozesses

Was ist DevSecOps? Warum benötigt man DevSecOps? Wie sollte DevSecOps aussehen? Was muss man beachten?

Diese Fragen sollen in der folgenden Analyse Schritt für Schritt beantwortet werden und bilden die Basis für das grundlegende Verständnis von DevSecOps Prozessen.

2.1 DevOps

Etymologisch betrachtet ist DevOps ein Kofferwort aus den Komponenten „Development“ und „Operations“. Die Bedeutung dieses Begriffs hingegen ist nicht klar und eindeutig definiert. DevOps ist vielmehr ein Verständnis für ein Ziel, welches versucht wird zu erreichen, eine Philosophie Prozesse zu optimieren und ein „Ansatz, wie die Zusammenarbeit zwischen Softwareentwicklung und IT-Betrieb verbessert werden kann.“[2]

Microsoft Azure, einer der führenden Betreiber im DevOps Segment, beschreibt in einem Blogpost diese Philosophie als eine Art Brücke zwischen IT-Betrieb, Qualitätstechnik und Sicherheit[3]:

> DevOps ermöglicht es zuvor getrennten Rollen wie Entwicklung, IT-Betrieb, Qualitätstechnik und Sicherheit, sich zu koordinieren und zusammenarbeiten, um bessere und zuverlässigere Produkte zu liefern. Durch die Einführung der DevOps-Kultur mit DevOps-Methoden und -Tools können Teams besser auf die Anforderungen ihrer Kunden reagieren, das Vertrauen in ihre eigenen Anwendungen steigern und Geschäftsziele schneller erreichen.

Zusammenfassend lässt sich sagen, dass DevOps ein Mindset, eine Philosophie oder ein Prozess ist, der den von Natur aus zyklischen Prozess der Softwareentwicklung automatisieren, koordinieren, strukturieren und beschleunigen soll. Was zunächst komplex klingt kann heutzutage mithilfe vieler Tools auf dem Markt verhältnismäßig leicht erreicht werden. Im Laufe des Seminars wurde hierfür primär die Plattform GitLab CI eingesetzt, welche auch hier später als Beispiel dienen wird. Wichtig ist allerdings ein Verständnis dafür zu

entwickeln, dass GitLab nicht die Lösung für DevOps ist, sondern nur ein Tool unter vielen, um die Strukturen eines Entwickler- und Operationsteams elegant umsetzen zu können.

2.2 Left Shift in DevOps

Der Begriff „Left Shift“ beschreibt den Ansatz Tests so früh wie möglich, also weiter „links“ auf einer Zeitachse, anzuordnen. Dadurch entsteht ein erhöhter Fokus auf Tests und somit im Idealfall auch ein höherer Qualitätsstandard bei der Entwicklung der Software.[4] Dies geschieht durch das direkte Feedback, welches nun an jeden Commit[1] geknüpft ist. Somit können schlechter Stil und Fehler direkt behoben werden, ohne weitere negative Effekte auszulösen.

Dies muss nicht ausschließlich auf Unit-/Integration- und End-To-End-Tests, sondern kann auch auf Security Tests, hier im Speziellen DAST, bezogen werden. Hierbei kann man sogar so weit gehen, dass alle sicherheitsrelevanten Tests nach jedem Commit ausgeführt werden sollten, selbst wenn diese eine hohe Zeitkomplexität mit sich bringen. Dies hat den Vorteil, dass Entwickler umgehendes Feedback über alle Sicherheitslücken, die in dem aktuellen Commit gefunden werden, erhalten. Daher wird dies auch von vielen Experten empfohlen, unter anderem GitLab, deren Tool die Grundlage für die Umsetzung einer Pipeline in diesem Seminar geboten hat.[5] Dadurch, dass Sicherheitslücken nun gemeinsam mit den Testergebnissen direkt an die Entwickler zurückgegeben werden, fließt das Beheben von Sicherheitslücken nahtlos in den Prozess der Behebung von Fehlern und der Entwicklung neuer Features ein.

2.3 DevSecOps

Etymologisch betrachtet wird das Kofferwort DevOps lediglich um „Security“ erweitert. Eine gute Integration des Sicherheitsaspekts in einen DevOps Prozess ist allerdings nicht sehr einfach, da hierbei die Entwicklung stark ausgebremst werden kann. Dennoch ist der Sicherheitsaspekt bei einer Software nicht zu unterschätzen und sollte als zentraler Bestandteil einer DevSecOps Pipeline gesehen werden. IT-Sicherheit ist grundsätzlich ein Wettrennen. Während einige Entwickler/Security Experten versuchen eine Software zu schützen, versuchen Hacker Schwachstellen zu finden und zu nutzen. Außerdem ist IT-Sicherheit ein sehr komplexes Thema, welches in nahezu jedem Fall Experten benötigt.

Allerdings lassen sich auch viele Teile der Überprüfung auf Schwachstellen automatisieren und nach links shiften. Hierfür gibt es verschiedene Punkte und Möglichkeiten, an denen man ansetzen kann. Einige davon werden später genauer erläutert, weshalb sie hier nur eine kurze Erwähnung finden sollen. Ein wichtiger Bestandteil bei der Umsetzung einer DevSecOps Pipeline ist ein guter Feedbackloop zwischen Entwicklern und Sicherheitsexperten, die helfen können gefundene Sicherheitslücken einzuordnen und falls nötig zu beheben. Dies kann beispielsweise durch automatische Issues[2] erreicht werden, die von den entsprechenden Schritten angelegt werden und den Entwicklern und Sicherheitsexperten die Möglichkeit geben über vorhandene Lücken zu diskutieren. Bei gefundenen Sicherheitslücken im DAST oder bei der Verwendung des integrierten Vulnerability Checkers bietet GitLab die Möglichkeit mit einem Klick ein Issue für eine Sicherheitslücke in den integrierten GitLab Issue Boards anzulegen. Wie genau dieser Loop insgesamt aussehen kann und sollte wird später noch graphisch visualisert (Siehe Abbildung 2).

2.4 Entwicklungsprinzipien

Der Bereich der agilen Entwicklung hat in den letzten Jahren viele verschiedene Verfahren entwickelt, um Projekte zu managen und agil zu arbeiten. „Agile Softwareentwicklung zeichnet sich durch selbstorganisierende Teams sowie eine iterative und inkrementelle Vorgehensweise aus.“[6] Der Kern agiler Softwareentwicklung ist also, das Kurzhalten von Planungsphasen und damit einhergehender kontinuierlicher weiterer Planung zur

[1] Ein Commit beschreibt eine Menge an Änderungen, welche vom Entwickler in das Versionskontrollsystem eingefügt werden. Im Fall dieses Seminars ist das Versionskontrollsystem git, bzw. GitLab. Nähere Informationen dazu sind später in Kapitel 2.5 zu finden.

[2] Ein Umstand der Aufmerksamkeit und möglicherweise Lösung bedarf. Oftmals Aufgaben oder Probleme.

Laufzeit des Projekts. Ähnlich wie bei DevOps ist das Ziel, sich möglichst schnell und flexibel auf Änderungen anpassen zu können, kontinuierlich neuen Code vorweisen zu können und sich „in regelmäßigen, kurzen Abständen mit dem Kunden [abstimmen]“[6] zu können und dabei die Codequalität zusätzlich positiv zu beeinflussen. Hier greifen DevOps und agile Methoden Hand in Hand. Agile Methoden können sehr schnell sehr komplex werden, weshalb sie in dieser Arbeit nur so weit beschrieben werden, dass der Zusammenhang zur DevSecOps Philosophie offensichtlich werden. Daher soll hier ein grundlegendes Verständnis für Kanban und Scrum gegeben werden, da agile Entwicklung eine wichtige Voraussetzung für ein gutes und erfolgreiches DevOps orientiertes Projekt ist.

2.4.1 Kanban

Wie bei vielen agilen Methoden steht im Zentrum von Kanban ein Board für die visualisierung einzelner Arbeitsschritte. Auf diesem sind alle zu tätigen Aufgaben eingetragen, welche Schritt für Schritt in Bearbeitung gehen können. Zur Unterscheidung dieser Zustände führt man mehrere, aber mindestens drei, Spalten ein. Diese sind meist mit den Begriffen „Backlog“, „In Progress“ und „Completed“ gekennzeichnet. Im Backlog befinden sich alle noch nicht begonnenen Aufgaben, die bei Bedarf in Bearbeitung gehen können. In diesem Fall wird die Karte aus der Spalte „Backlog“ in die Spalte „In Progress“ bewegt, um für alle Mitarbeitenden zu signalisieren, dass an dieser Aufgabe bereits gearbeitet wird. Nach Fertigstellung einer dieser Karten wird diese in die Spalte „Completed“ verschoben und ist damit erledigt. Um verschiedene Wichtigkeitsstufen abbilden zu können ist möglich verschiedene sog. Lanes zu verwenden. Diese werden als einzelne Reihen dargestellt, von denen in der Regel obere Reihen vor unteren Reihen bearbeitet werden sollen. Zudem limitiert man die Anzahl maximal erlaubter Aufgaben in Bearbeitung, um zu starkes Multitasking und damit einhergehender sinkender Effizienz zu vermeiden.

Positiv lässt sich hier anmerken, dass durch ein sehr offenes, transparentes Prinzip ein gleichmäßiger Workflow entsteht, der sich in den meisten Situationen einfach in bestehende Prozesse integrieren lässt.

Dem entgegen steht ein Problem, welches dadurch entsteht, dass keinerlei Möglichkeiten zur Zeitplanung in Kanban vorgesehen sind. Dies kann dazu führen, dass Deadlines nicht eingehalten werden können, da sich Entwickler oder Manager leichter in der Aufgabenverteilung verschätzen. Zudem ist es bei Kanban nötig, alle vorhandenen Aufgaben in einzelne unabhängige Schritte aufteilen zu können. Dies kann allerdings nicht immer gewährleistet werden, sollte in der Softwareentwicklung allerdings meistens kein Problem sein, da grundsätzlich Abstraktion in kleine Pakete in jeder Codebasis wünschenswert ist.[7]

2.4.2 Scrum

Atlassian, zieht bei der Beschreibung von Scrum eine sehr passende Football Analogie heran[8]:

> Scrum ist ein Framework, das die Zusammenarbeit in Teams unterstützt. *Scrum* steht im Rugby für *Gedränge* – und genau wie ein Rugbyteam, das für das große Spiel trainiert, sind Teams mit Scrum in der Lage, durch Erfahrungen zu lernen, sich bei der Problembehebung selbst zu organisieren sowie ihre Erfolge und Niederlagen zu reflektieren, um sich kontinuierlich zu verbessern.

Ähnlich wie bei Kanban werden Aufgaben aufgeteilt in „Backlog“, „In Progress“ und „Completed“. Allerdings liegt ein weitaus größerer Fokus auf der Zeitplanung, die bei Kanban außer Acht gelassen wird. Im Gegensatz zu Kanban werden die Aufgaben nicht einfach bearbeitet, sobald die vorherige Aufgabe abgeschlossen wird, sondern es wird für einen kurzen Zeitraum, meist zwei Wochen, geplant, welche Aufgaben zu erledigen sind. Dieser Zeitraum nennt sich Sprint und stellt die Basis von Scrum dar. In regelmäßigen Meetings legt ein sog. Scrum Master die einzelnen Aufgaben fest, die im nächsten Sprint erledigt werden sollen und weist diese oftmals direkt Mitarbeitern zu, die diese Aufgaben im Laufe des Sprints erledigen sollen.

Dies sorgt für eine enorm hohe Struktur und ein außerordentliches Zeitmanagement, welches in Kanban so nicht gewährleistet werden kann. Allerdings sind zweiwöchige (einmal pro Sprint) Besprechungen oftmals nicht flexibel genug, um schnellen Entscheidungen eines Kunden zu entsprechen, was zu Planungsschwierigkeiten führen kann.

2.4.3 Fazit

Die beiden oben genannten Methoden bieten eine gute Grundlage für DevSecOps. Insgesamt muss jedes Team für sich entscheiden, welches Entwicklungsprinzip für sie am besten funktioniert. Alle Prinzipien einmal auszuprobieren ist definitiv empfehlenswert, um den optimalen Workflow für sein Team und seine Anwendungsfälle finden zu können.

2.5 Git Struktur/Branching

Bevor eine Betrachtung der einzelnen Schritte im Prozess stattfinden kann, muss zunächst die Grundstruktur des Versionskontrollsystems (VCS) geklärt sein. In diesem Seminar wurde als VCS GitLab (git) verwendet, weshalb dieses auch hier als Grundlage dienen soll. Wenn es um das Thema Struktur im git geht, entstehen häufig hitzige Diskussionen unter Entwicklern. Die Menge an möglichen Modellen ist scheinbar endlos. In der Vergangenheit hat sich oft gezeigt, dass git-flow[9] eine gute Möglichkeit ist, um ein git Repository zu strukturieren. Git Flow besteht aus zwei permanenten Branches. Der erste ist der master, welcher die fertigen Releases widerspiegeln soll. Der zweite ist der develop, auf dem der aktuelle Entwicklungsstand abgebildet werden soll. Zusätzlich werden vom develop für jedes Feature eigene branches erstellt, die nach der Fertigstellung auch in diesen wieder zurück gemerged werden. Analog existieren hotfix branches, welche allerdings nicht vom develop sondern vom Master erstellt werden und lediglich dazu dienen schnell Fehler aus Releases zu beheben. Diese werden nach Fertigstellung sowohl direkt in den master als auch in den develop gemerged. Als letztes kommen noch die Release Branches hinzu. Diese führen zu einem finalen Release und sollten nur noch Bugfixes enthalten. Diese werden direkt wieder zurück in den develop gemerged. Nachdem der Release keine bekannten Bugs mehr enthält, wird dieser in den master gemerged und veröffentlicht. Wie allerdings schon auffällt stellt sich dieses Modell als verhältnismäßig komplex heraus und ist speziell für unerfahrene Entwickler schwer zu erlernen und oftmals auch hinderlich durch die starre Struktur.

Daher haben sich im Laufe der Zeit viele alternative Modelle entwickelt, die versuchen diese Problem zu lösen. Ein häufig vorgeschlagenes solches Alternativmodell stellt GitHub flow[10] dar. Dieses Modell ist im Gegensatz zu git-flow sehr einfach gestrickt und besteht lediglich aus einem Hauptbranch, welcher den Release Stand widerspiegelt. Zudem kommen nur noch die Feature branches, welche allerdings im Gegensatz zu git-flow direkt in den master gemerged werden und veröffentlicht werden.

Dieses Modell ist offensichtlich sehr einfach, was allerdings auch wieder Probleme mit sich bringt. Test-/Staging-Umgebungen können nicht wirklich ausgelassen werden und es existiert keinerlei Kontrolle über die Menge an Features die ausgerollt wird.

Daher empfiehlt sich ein Modell, welches die Vorteile von git-flow und GitHub Flow kombiniert.

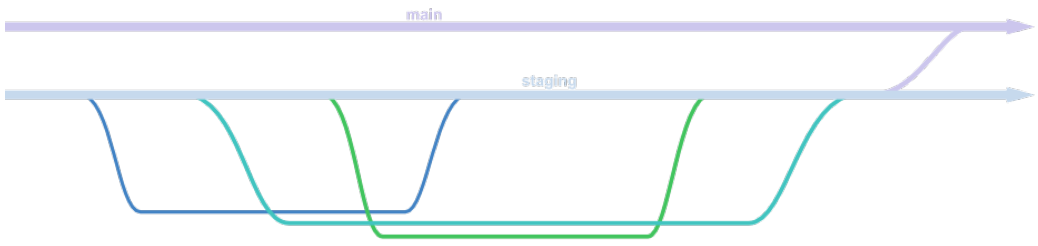

Abbildung 1: Graphische Darstellung der Kombination aus git-flow und GitHub Flow

Das hier gezeigte Modell basiert relativ stark auf GitHub flow. Es existieren zwei parallele Branches, welche nach und nach immer wieder auf den gleichen Stand gebracht werden sollen. Der erste ist ein staging branch,

auf welchen alle Features nach Fertigstellung analog zu GitHub Flow gemerged werden sollen. Dieser Branch sollte dennoch nur deploybaren Code nach aktueller Kenntnis über Bugs enthalten. Dieser kann dann Nutzern als Beta-Version zur Verfügung gestellt werden und erst nach einer Evaluation tatsächlich deployed werden. So haben Nutzer die Möglichkeit neue Features sofort zu testen und direktes Feedback in den Entwicklungsprozess einzubringen. Im Gegensatz zu GitHub flow ist dies allerdings nicht der Branch, der in Production veröffentlicht wird. Dafür gibt es den main branch, für den die gleichen Kriterien gelten wie für den Staging Branch. Außerdem sollte auf den main branch nur vom staging branch gemerged werden, nachdem dieser von den Beta Testern ausführlich auf Bugs getestet wurde. Entwicklung sollte allerdings, genau wie bei GitHub flow, auf dem Main und Staging branch nicht stattfinden. Hierfür werden für jedes Feature eigene Branches erstellt (dunkelblau, hellblau und grün in Abbildung 1). Auf diesen werden alle Commits gepushed, bis das Feature vollständig funktioniert und somit bereit ist, in den Staging branch gemerged zu werden. Sobald dieses Stadium erreicht ist, wird ein Pull Request in den Staging Branch erstellt, wodurch ein Review Prozess beginnt. Zunächst laufen alle automatisierten Tests und stellen die Codequalität und -funktionalität bestmöglich fest. Im nächsten Schritt sollte der Merge Request von anderen Personen bestätigt werden und mögliche Fehler und Unstimmigkeiten sollten behoben werden. Hier gibt es auch die Möglichkeit über den Code zu diskutieren und ihn bestmöglich zu optimieren. Sobald diese Schritte durchlaufen sind, kann das Feature in den Staging Branch gemerged werden. Nun rollt die Pipeline das Feature automatisch an alle Beta Nutzer aus, sodass der Feedbackprozess aus dem Betatest beginnen kann. Unter der Annahme, dass hierbei keine Fehler auftreten kann man nun mit erhöhter Sicherheit und Stabilität den Staging Branch nach einem Pull Request in den Main Branch mergen, was ein Deployemnt für alle Nutzer auslöst.

Somit ist ein Branchingmodell, welches nur minimal komlizierter als GitHub Flow geworden ist. Dennoch bietet es im finalen Produkt eine stark erhöhte Stabilität, da Features von kleineren Testgruppen verwendet werden und deren Feedback gesammelt wieder in den Entwicklungsprozess einfließen kann. Nutzer können sich also frei entscheiden, ob sie lieber experimentellere Software nutzen wollen, um möglichst schnell neue Features zu erhalten, oder ob Stabilität ein größerer Faktor ist und ein Wunsch nach geregelteren Releases besteht. Alle Vorteile von GitHub Flow gegenüber git-flow bleiben allerdings erhalten, weshalb dies immer noch ein exzellentes Modell für DevOps ist.

2.6 DevSecOps Pipelines

Pipelines bilden die Grundlage in der Umsetzung eines erfolgreichen DevSecOps Projekts. Grundsätzlich lassen sich Pipelines in drei Bereiche unterteilen: Testen, Bauen und Deployen. Allerdings lassen sich diese Schritte nicht immer einfach nacheinander ausführen, da Schritte im Test beispielsweise von einem Build oder Deployment abhängen können, genauso wie ein Deployment von einem Test und offensichtlich einem Build abhängen kann. Daher werden einige der wichtigsten Schritte im Folgenden nacheinander einzeln betrachtet und analysiert. Aufgrund der teils hohen Komplexität einiger Schritte werden alle einzelnen Schritte nur angeschnitten, um einen Überblick über die nötigen Funktionen einer guten Pipeline zu erhalten. Diese sind chronologisch so angeordnet, wie sie auch im finalen Prozess vorhanden sein sollten.

2.6.1 Coding Richtlinien/Code Architektur

Nachdem nun klar ist, in welcher Struktur der Code im VCS vorzufinden ist, gilt es nun die einzelnen Schritte in der Pipeline zu betrachten. Hierbei sollte mit den Syntax- und Architekturtests begonnen werden. Bevor die Funktionalität einer Software getestet wird, sollte immer sichergestellt sein, dass die Architektur und Syntax den gewählten Konventionen entspricht. Für die Bewertung der Position eines Schrittes in einer Pipeline gibt es mehrere ausschlaggebenden Punkte, welche bei jedem Job in der Pipeline betrachtet werden sollten.

- Welche Abhängigkeiten existieren zu anderen Schritten in der Pipeline?
- Wie hoch ist die Zeitkomplexität des Jobs?
- Wann in der Pipeline muss dieser Job ausgeführt werden?

Welche Abhängigkeiten existieren zu anderen Schritten? Um die Qualität des Source Codes zu testen bedarf es ausschließlich des Codes. Es gibt keine weiteren Abhängikkeiten. Somit gibt es auch keine anderen Jobs, welche vor diesem Job auszuführen sind. Damit sollte dieser Schritt zu Beginn der Pipeline angeordnet sein.

Da auch keine anderen Tests Abhängigkeiten zu Architektur- und Syntaxtests haben lassen sich diese auch problemfrei zu anderen Tests parallelisieren.

Wie hoch ist die Zeitkomplexität des Jobs? Syntax- und Architekturtests sind in relativ kurzer Zeit auszuführen, da keine externen Abhängigkeiten oder komplexe Build Prozesse nötig sind. Dies ist speziell in den ersten Jobs äußerst erstrebenswert, da die Pipeline möglichst früh fehlschlagen sollte, wenn Fehler existieren, um unnötige Verzögerungen im Entwicklungsprozess zu vermeiden. Zudem ist auch die Behebung von Syntaxfehlern zumeist relativ schnell erledigt, sodass zügig die weiteren Schritte der Pipeline erreicht werden können.

Wann muss dieser Job ausgeführt werden? Bei dieser Frage ist die Zeitkomplexität oftmals ein großer Faktor. Tests, die kaum Zeit in Anspruch nehmen kann man ohne große Einschränkungen häufiger ausführen. Dementsprechend empfiehlt es sich hier, die Tests bei jedem Commit auszuführen, um stets eine gute Codequalität gewährleisten zu können und bereits früh im Prozess korrigieren zu können, falls die Architektur nicht den gesetzten Standards entspricht.

2.6.2 Vulnerability Checker

Bei der Entwicklung großer Projekte ist es kaum möglich externe Abhängigkeiten zu vermeiden. Für nahezu jedes Framework und jede Programmiersprache existieren Package Manager, die den Prozess externe Pakete ins Projekt einzufügen extrem vereinfachen. Diese Leichtigkeit löst allerdings ein Problem aus. Viele Libraries wurden selbst von unerfahrenen Entwicklern geschrieben, oder wurden in sehr kurzer Zeit entwickelt, was zu Fehlern und vor allem zu Sicherheitslücken führen kann. Die Entwicklercommunity kann diese zwar häufig relativ schnell finden und identifizieren, aber Maintainer können die Sicherheitslücken oftmals nicht sofort schließen. Daher ist ein wichtiger Faktor einer guten Pipeline Abhängigkeiten auf bekannte Lücken zu prüfen. Hierfür gibt es Datenbanken, die von der Community gepflegt werden und mit nahezu allen bekannten Sicherheitslücken gefüllt sind. In diesem Job in der Pipeline geht es also darum, die Quelldateien, zumeist Abhängigkeitsdateien der Paketmanager (z.B. package.json oder requirements.txt), gegen diese Datenbanken zu prüfen und den Entwicklern Feedback zu geben, falls bekannte Sicherheitslücken existieren.

Welche Abhängigkeiten existieren zu anderen Schritten? Auch hier lässt sich feststellen, dass keine Abhängigkeiten zu anderen Schritten in der Pipeline existieren. Lediglich zum Source Code (wobei möglicherweise sogar einzelne Dateien reichen, wie zum Beispiel package.json oder requirements.txt) und zu den Vulnerability Datenbanken existieren externe Abhängigkeiten. Deshalb lässt sich dieser Schritt zusammen mit den vorher genannten parallel ausführen.

Wie hoch ist die Zeitkomplexität des Jobs? Die Zeitkomplexität ist relativ gering, wenn man einen der gängigen Paketmanager verwendet. (z.B. npm, pip) Viele Tools zur Bewertung von Paketen verwenden eine Datenbank, in der sie sehr schnell überprüfen können, ob für die angegebene Version Sicherheitslücken gemeldet sind.

Wann muss dieser Job ausgeführt werden? Wie bereits zuvor erlaubt es die geringe Zeitkomplexität diesen Schritt auch häufiger auszuführen. Zudem existieren keine weiteren Abhängigkeiten, weshalb der Job auch gut parallel ausgeführt werden kann. Selbst, wenn dies aber nicht der Fall wäre sollte man diese Stufe dennoch bei jedem Commit ausführen und überprüfen, ob neu hinzugekommene Pakete bekannte Schwachstellen enthalten. Allerdings ist dies leider noch nicht vollständig ausreichend, da Sicherheitslücken auch später entdeckt

werden können. Daher empfiehlt es sich, diese Tests je nach aktuellem Entwicklungsstand und der Häufigkeit von Commits in einem Scheduled Lauf der Pipeline ebenfalls durchzuführen, um neue Sicherheitslücken möglichst schnell zu finden, auch wenn an dem Projekt nicht mehr aktiv entwickelt wird.

2.6.3 Lizenzüberprüfung

Die Komplexität vieler Lizenzen macht es Entwicklern oft nicht leicht zu entscheiden, welche Pakete sie problemfrei verwenden dürfen. Auch Abhängigkeiten von Abhängigkeiten, etc. können zum Problem werden. Hierbei eine gute Übersicht zu behalten ist ohne dem richtigen Tooling nahezu unmöglich. Außerdem liegt die Entscheidung der erlaubten Lizenzen oftmals nicht in der Hand der Entwickler, sondern einer Rechtsabteilung. Mithilfe einer guten Lizenzprüfung kann man hier also eine Brücke schlagen für eine nahtlose Kooperation zwischen Entwicklern und rechtlichen Lizenzaspekten, ohne dass Entwickler einen tiefen Einblick in Lizensierungsmodelle benötigen und ohne, dass die Rechtsabteilung irgendwelche Informationen über Softwareentwicklung benötigt, um eine korrekte Lizenzierung validieren zu können.

Welche Abhängigkeiten existieren zu anderen Schritten? Da die Lizenzüberprüfung lediglich Informationen über die eingebundenen Pakete benötigt, wie auch zuvor schon bei dem Vulnerability Checker verhält es sich auch bei den Abhängigkeiten genau gleich. Außer einer äußeren Abhängigkeit zu den entsprechenden Paketquellen gibt es keine weiteren Abhängigkeiten, im Speziellen nicht zu anderen Schritten in der Pipeline. Daher lässt sich auch dieser Job problemfrei zu Beginn der Pipeline parallel zu den vorhergegangen Schritten ausführen.

Wie hoch ist die Zeitkomplexität des Jobs? Da hierbei lediglich Lizenzen gegen Regeln geprüft werden müssen, ist dieser Job sehr schnell ausführbar, besonders, wenn nur neue oder aktualisierte Lizenzen geprüft werden.

Wann muss dieser Job ausgeführt werden? Da ein Paket in einer festen Version seine Lizenzierung nicht einfach ändern kann, reicht es völlig aus bei jedem Commit zu prüfen, ob sich Pakete verändert haben. Sobald eine Veränderung in den verwendeten Paketen zu finden ist, sollte die Pipeline die Lizenzen der entsprechenden Lizenzen überprüfen.

2.6.4 Dynamic Application Security Testing (DAST)

Dynamic Application Security Testing beschreibt den Prozess eine laufende Anwendung mit einem Black Box Verfahren aktiv auf Sicherheitslücken zu überprüfen. Black Box Verfahren bedeutet, dass keinerlei interne Informationen wie zum Beispiel Secrets oder Source Code für die Überprüfung zur Verfügung stehen. Dies stellt daher das realistischste aller möglichen automatisierten Testverfahren dar. Die *aktive* Überprüfung auf Sicherheitslücken bedeutet, dass tatsächlich mit dem Ziel die Anwendung von außen zu zerstören, oder Daten zu manipulieren vorgegangen wird. Daher sollten diese Tests niemals auf einer produktiven Umgebung ausgeführt werden. Dennoch sollen die Ergebnisse der Tests repräsentativ für ein Echtweltszenario sein. Daher empfielt es sich, eine Testumgebung zu verwenden, die nahezu identisch zur Produktivumgebung aufgesetzt ist. Docker bietet hier vor allem in Kombination mit GitLab CI eine sehr gute Möglichkeit dies zu erreichen, wie im letzten Kapitel näher beschrieben wird.

Welche Abhängigkeiten existieren zu anderen Schritten? Da dieser Schritt eine fertige Anwendung testen soll, ist dieser stark Abhängig von allen Schritten, die nötig sind, um die Anwendung zu bauen und zu deployen. Diese lassen sich nur sehr schwer generalisieren, da jede Anwendung unterschiedliche Buildprozesse hat. Dennoch wird DAST aufgrund der starken Abhängigkeit meist als letzter Schritt in der Pipeline ausgeführt.

Wie hoch ist die Zeitkomplexität des Jobs? Die Zeitkomplexität von DAST ist sehr hoch. Dies hängt mit der Komplexität und der Verlangsamung von Test durch die interaktion mit graphischen Bedienelementen zusammen. Bei DAST werden Websiten zunächst nach allen möglichen Routen der Anwendung abgesucht. Dazu wird ein Crawler verwendet welcher Schritt für Schritt versucht alle Links, Buttons und Eingabefelder zu bedienen um im Idealfall alle erreichbaren Routen zu finden. Je nach Komplexität der Anwendung kann dies allein bereits bis zu 30 Minuten dauern. Sobald alle zu Testenden Seiten gefunden sind, werden die gewünschten Sicherheitslücken auf der Website gesucht. Hierzu wird versucht mit gezielten Eingaben ein Fehlverhalten auszulösen. Da dies viele Schritte und Versuche mit sich bringt, kann die Dauer einer DAST Stage bis zu mehreren Stunden an Zeit in Anspruch nehmen. Somit ist nicht nur die starke Abhängigkeit, sondern auch die hohe Zeitkomplexität des Jobs ein Grund dafür ihn am Ende der Pipeline auszuführen.

Wann muss dieser Job ausgeführt werden? Hier gibt es viele verschiedene Auffassungen. Mindestens vor jeder Veröffentlichung sollte die Pipeline laufen, um gewährleisten zu können, dass die gröbsten Sicherheitslücken gefunden wurden. Desto öfter die Pipeline ausgeführt wird desto schneller können Sicherheitslücken behoben werden und desto direkter ist das Feedback an die Entwickler. Dies ist grundsätzlich empfehlens- und wünschenswert. Deshalb wir auch oft, zum Beispiel auch von GitLab empfohlen, die Pipeline bei jedem Commit laufen zu lassen.[5] Dem entgegen steht die hohe Zeitkomplexität und die damit verbundene Auslastung der Serverinfrastruktur. Daher muss jedes Entwicklerteam selbst abwägen, wie oft dieser Schritt durchgeführt werden sollte.

2.6.5 Visualisierung eines optimalen DevSecOps Prozesses

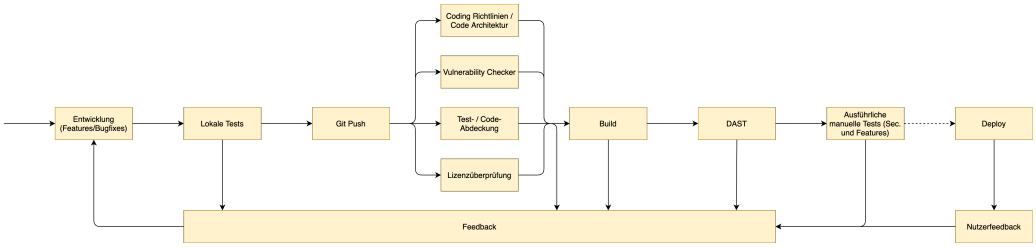

Abbildung 2: Graphische Darstellung eines optimalen DevSecOps Prozesses

Softwareentwicklung ist immer ein zyklischer Prozess, bei dem Feedback aus möglichst vielen Quellen wieder in die Entwicklung zurückfließen sollten. Dieser Prozess wird mit dem oben gezeigten Schema optimiert und beginnt mit den ersten Zeilen Code in der Entwicklung. Sobald die ersten Features lokal getestet sind, können Sie in ein VCS gepusht werden. Dies löst die DevOps Pipeline aus, welche den neuen Code auf Sicherheitslücken, Testabdeckung, Richtlinien, Architektur und Lizenzen prüft. Alle hierbei gesammelten Informationen sollen den Entwicklern wieder als Feedback zur Verfügung stehen. Sollten hierbei keine kritischen Probleme gefunden werden, wird die Anwendung gebaut und für ein Deployment vorbereitet. Alle Fehler werden ebenfalls direkt als Feedback an die Entwickler zurückgegeben. Sobald die Anwendung gebaut wurde, wird sie in der DAST Stage aktiv auf Sicherheitslücken überprüft. Sollten unbehandelte Sicherheitslücken gefunden werden, werden diese ebenfalls als Feedback an die Entwickler zurückgegeben. Wenn nun keine Auffälligkeiten vorhanden sind, ist die Software bereit, je nach aktuellem Branch, auf die entsprechenden Umgebungen deployed zu werden, um sich schlussendlich noch das Feedback der Nutzer einholen zu können und den Prozess von vorne zu beginnen.

3 Vergleich zwischen Burp Suite, OWASP ZAP und GitLab für den Einsatz im DAST

3.1 Burp Suite

> Burp Suite Professional is the web security tester's toolkit of choice. Use it to automate repetitive testing tasks - then dig deeper with its expert-designed manual and semi-automated security testing tools. Burp Suite Professional can help you to test for OWASP Top 10 vulnerabilities - as well as the very latest hacking techniques.[11]

Mit diesen Worten beschreibt Portswigger deren Tool Burp Suite, eines der größten, bekanntesten und ohne Frage besten Frameworks, wenn es um das Thema Anwendungssicherheit geht. Allerdings kommt mit großer Popularität und großer Funktionalität meistens auch ein großer Preisfaktor ins Spiel. Bei Burp Suite wird hierbei in zwei unterschiedliche Preismodelle untergliedert. Burp Suite Community Edition und Burp Suite Professional. Burp Suite Community Edition ist kostenfrei, während Burp Suite Professional für 349€ pro Jahr zu erwerben ist.

Burp Suite bietet alle Features, die man von einem Tool zum dynamic application security testing erwarten kann und sollte. Besonders relevant sind hier die in den OWASP TOP 10[12] gelisteten Sicherheitslücken, welche alle von Burp Suite gefunden werden können.

3.2 OWASP ZAP

> The world's most widely used web app scanner. Free and open source. Actively maintained by a dedicated international team of volunteers.[13]

Diese Beschreibung von ZAP bringt es gut auf den Punkt. ZAP ist ein Open Source Projekt, welches von vielen Entwicklern weltweit kostenlos zur Verfügung gestellt wird. Daraus entstehen auch durchaus Nachteile, wie zum Beispiel eine relativ kleine Dokumentation, vor Allem im Vergleich zu Burp Suite. Das Basisfeatureset ist bei OWASP ZAP allerdings nahezu identisch, wie bei Burp Suite. Dennoch ist Burp Suite in der Erkennung einiger Sicherheitslücken, sowie der Abdeckung durchaus noch überlegen gegenüber dem Community Tool ZAP.[14]

3.3 GitLab

GitLab selbst unterstützt keinerlei DAST Funktionalität. Allerdings stellt GitLab ab dem Ultimate Tier eine nahtlose Integration zu OWASP ZAP zur Verfügung. Diese ermöglicht es die volle Funktionalität von ZAP in wenigen Zeilen in eine GitLab CI Pipeline zu integrieren. Hierin liegt auch die Stärke von GitLab im Bereich DAST. Bei der Konfiguration von Tools wie Burp Suite oder ZAP werden oft Experten benötigt, was mit einem hohen Kostenaufwand verbunden ist. GitLab CI ermöglicht es mit geringen Mehrkosten eine solche Pipeline aufzusetzen, zu konfigurieren und eine hohe Application Security gewährleisten zu können. Allerdings ist GitLab Ultimate mit $1188 USD pro Jahr pro Nutzer ebenfalls kein billiges Tool.[15] Aufgrund der vielseitigkeit setzen bereits viele Unternehmen auf GitLab aus anderen Gründen und können daher GitLabs DAST Anbindung mehrkostenfrei nutzen. Besonders in diesem Fall empfiehlt sich die Verwendung von GitLab CI in einer DAST Pipeline.

3.4 Fazit

Obwohl Burp Suite einen geringfügig größeren Funktionsumfang hat bietet ZAP mit einer Open Source Lösung eine würdige Alternative. Mit der Einfachheit in der Konfiguration von GitLab kann allerdings keines der beiden Tools mithalten. Da in dem Seminar bereits mit GitLab gearbeitet wurde hat sich die Verwendung der integrierten Tools offensichtlich angeboten, weshalb die Entscheidung bei der Auswahl des Tools auf GitLab CI gefallen ist.

4 GitLab CI im Einsatz als DAST Tool

Die Integration einer DAST Stage in GitLab CI wurde von den Entwicklern so leicht wie möglich gemacht. Wie bei jedem Sicherheitstest auf einer laufenden Anwendung muss diese laufen. Dabei kann man im Fall von GitLab stark von Containerization profitieren. GitLab bietet die Möglichkeit, zuvor gebaute Container in der eigenen Container registry zu speichern und später im DAST Job als sog. Service einzubinden[3]. Diese Services laufen parallel zu dem Container, der für die aktuelle Stage gestartet wird und wird automatisch im selben Docker Netzwerk gestartet, sodass eine einfache Kommunikation zwischen den verschiedenen Containern möglich ist. Somit wird keine dedizierte Testumgebung benötigt, die bei jedem Einsatz eines DAST Tools wieder zurückgesetzt wird. Dennoch sind die Ergebnisse vergleichbar mit einem realen Szenario, vorausgesetzt die Anwendung wird später in den gleichen Containern deployed, was heutzutage aber meistens der Fall sein sollte.

4.1 Nötige Stages und Jobs für die Pipeline

Da in diesem Teil eine stark reduzierte Pipeline betrachtet wird, die lediglich DAST Funktionalität enthalten soll, werden lediglich zwei Stages und Jobs benötigt. Die erste Stage enthält das Bauen und Hochladen eines Dockercontainers, der in der zweiten Stage als Service eingebunden werden kann, damit er getestet werden kann

4.2 Build und Upload eines Dockercontainers in die GitLab Registry

Der erste Schritt, um die Dockercontainer in GitLab CI bauen zu können ist, Docker im Container zur Verfügung zu stellen. Dazu wird das offizielle `docker:dind`[4] Docker image verwendet, welches als service in die Pipeline eingebunden wird.

Zudem wird der Job in die richtige Stage eingeordnet, was mit `stage: deploy` geschieht.

Nun stehen alle nötigen Informationen zur Verfügung um Dockerimages bauen und hochladen zu können. Der erste Schritt hierbei ist die Anmeldung in einer Container Registry, in der das Image hochgeladen werden soll. Dazu wird in der Pipeline als erstes der Befehl

```
echo $CI_JOB_TOKEN | docker login -u gitlab-ci-token --password-stdin $CI_REGISTRY
```

ausgeführt.

Hierbei werden zwei Umgebungsvariablen verwendet, welche von GitLab CI automatisch zur Verfügung gestellt werden. [16]

- **CI_JOB_TOKEN:** Dies ist ein einmaliger Token der für den Job generiert wurde und als Passwort in der integrierten Container registry verwendet werden kann, solange der Job läuft.
- **CI_REGISTRY:** Dies ist die Adresse der integrierten Container Registry von GitLab

Im nächsten Schritt wird nun das letzte bestehende docker image aus der registry gepulled. Dies ist dafür da, um die Möglichkeiten von Docker caches zu nutzen und unnötige komplexe Schritte vermeiden zu können, kann aber auch weggelassen werden, wenn dieses Feature nicht erwünscht ist, oder kein Zeitgewinn dadurch in der Pipeline entsteht.

Nun muss das Dockerfile nur noch mit

```
docker build --tag $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA --tag $CI_REGISTRY_IMAGE:latest .
```

gebaut und getagged werden. Es empfielt sich die Images zusätzlich zum *latest* Tag auch mit dem Hash des aktuellen Commits zu taggen, um im Fehlerfall stets eine Verbindung zwischen gebautem Docker image und commit herstellen zu können. Hierfür werden erneut zwei Umgebungsvariablen verwendet:[16]

[3] Die Grundlagen im Umgang mit Docker und Container Registries werden im Folgenden vorausgesetzt. Nähere Informationen können hier (Get Started) und hier (Container Registries) nachgelesen werden.

[4] dind steht für *Docker in Docker* und stellt eine und sichere Möglichkeit dar, Docker in einer isolierten Umgebung zu verwenden.

- **CI_REGISTRY_IMAGE:** Die Adresse der Container Registry für das aktuelle Projekt.
- **CI_COMMIT_SHA:** Der Hash des commits auf dem die Pipeline läuft

Nun müssen die fertig gebauten Images nur noch mit `docker push` hochgeladen werden. Die fertige deploy stage sieht nun also so aus:

```
deploy:
  services:
  - name: docker:dind
    alias: dind
  image: docker:19.03.5
  stage: deploy
  script:
    - echo $CI_JOB_TOKEN | docker login -u gitlab-ci-token --password-stdin $CI_REGISTRY
    - docker pull $CI_REGISTRY_IMAGE:latest || true
    - docker build --tag $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA --tag $CI_REGISTRY_IMAGE:latest .
    - docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
    - docker push $CI_REGISTRY_IMAGE:latest
```

4.3 Konfiguration der DAST Stage in GitLab CI

Der größte Teil der Konfiguration dieser Stage wird von einem GitLab Template eingefügt.[17] Dies ist wie bereits zuvor erwähnt auch der größte Vorteil von GitLab gegenüber anderen Tools. Das Einfügen des Templates ist denkbar einfach:

```
include:
  - template: DAST.gitlab-ci.yml
```

Nun muss nur noch das zuvor gebaute Docker Image als service in die Pipeline eingebunden werden:

```
services: # use services to link your app container to the dast job
  - name: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
    alias: flask_demo
```

Nachdem das Template und der Application Service eingefügt wurden lässt sich die restliche Stage über Umgebungsvariablen konfigurieren. Die dafür nötigen Variablen werden nun im Folgenden nacheinander erklärt.

4.3.1 Grundlegende Konfiguration

Mit lediglich drei essentiellen Variablen lässt sich ein vollständiger DAST Prozess in GitLab CI nun konfigurieren.

DAST_WEBSITE Diese Umgebungsvariable gibt an, unter welcher Adresse die zu testende Anwendung zu finden ist. Da das Deployment mit einem Docker Service gemacht wurde, reicht es hier den Hostname des Containers zu verwenden und das passende Protokoll und den passenden Port hinzuzufügen.

DAST_FULL_SCAN_ENABLED Diese Variable entscheidet darüber, ob der ZAP Baseline Scan[18] oder der ZAP Full Scan[19] verwendet wird. Der Baseline Scan attackiert die Anwendung nicht aktiv und dauert nur wenige Minuten. Der Full Scan verwendet aktive Methoden und ist damit in der Lage deutlich mehr Sicherheitslücken zu finden. Dies geschieht auf Kosten der Laufzeit und kann durchaus mehrere Minuten bis Stunden dauern, abhängig von der Größe der zu testenden Software.

DAST_BROWSER_SCAN Mit DAST_BROWSER_SCAN aktiviert verwendet GitLab einen Browser basierten Crawler, um ein möglichst realistisches Szenario bereits beim Crawlen der Website zu schaffen.

4.3.2 Anmeldung

Die Konfiguration mit den oben genannten Variablen kann allerdings nur Bereiche testen, die öffentlich zugänglich sind, da keine Authentifizierung erfolgt ist. In vielen Anwendungen würde hierbei also nicht viel mehr als ein *Registrierungs-/Anmeldeformular* und eine *Unauthorized Page* getestet werden. Daher ist es wichtig, dass der DAST Prozess sich auch in der Anwendung anmelden kann. Die Konfiguration dieser Anmeldung ist ebenfalls denkbar einfach und wird wieder über Umgebungsvariablen gesteuert. Diese werden nun im Folgenden beschrieben und schließen die Konfiguration der DAST stage ab.

DAST_AUTH_URL Die Url zur Anmeldeseite, auf der Username und Password eingegeben werden können.

DAST_USERNAME Der Username, mit den passenden Rechten und Links zu den gewünschten Seiten, die getestet werden sollen.

DAST_PASSWORD Das Passwort für den Nutzer, der im *DAST_USERNAME* angegeben wurde.

DAST_USERNAME_FIELD Selektor für den Username Input. Dieser kann verschieden angegeben werden. Die mächtigste Möglichkeit ist einen CSS Selektor anzugeben. Dies dient dazu, dass mithilfe von Selenium ein Login stattfinden kann, um die Anwendung auch im angemeldeten Status zu testen.

DAST_PASSWORD_FIELD Selektor für den Passwort Input. Analog zu *DAST_USERNAME_FIELD*.

DAST_SUBMIT_FIELD Selektor für den Input der die Anmeldung bestätigt und ausführt. Analog zu *DAST_USERNAME_F*

DAST_EXCLUDE_URLS Hier werden URLs angegeben welche nicht im Scan beachtet werden sollen. Hier sollte die Logout Seite angegeben werden, da es sonst passieren kann, dass der Test den Nutzer ausloggt bevor alle Tests abgeschlossen werden konnten.

DAST_AUTH_VERIFICATION_URL Diese Url dient dazu, zu überprüfen, ob die Anmeldung erfolgreich war. Hier sollte also die Url angegeben werden, auf die nach dem Login weitergeleitet wird.

DAST_AUTH_VERIFICATION_SELECTOR Dieser Selektor hat die selbe Funktionalität wie die *DAST_AUTH_VERIFICATI* allerdings auf Ebene einzelner HTML elemente im DOM. Die Funktionsweise ist analog zum *DAST_USERNAME_FIELD*.

DAST_AUTH_REPORT Wenn diese Option auf „true“ gesetzt ist wird ein erweiterter Report ausgegeben, der speziell beim Debugging eines fehlschlagenden Logins sehr nützlich sein kann.

4.4 Vollständige DAST Konfiguration für GitLab CI

Kombiniert sieht die fertige vollständige Konfiguration für eine DAST Pipeline in GitLab CI also wie Folgt aus:

```
stages:
  - deploy
  - dast
include:
  - template: DAST.gitlab-ci.yml

variables:
  DAST_WEBSITE: http://flask_demo:5000
  DAST_AUTH_URL: $DAST_WEBSITE/login
  DAST_USERNAME: "example@mail.com"
  DAST_PASSWORD: "password" # use custom, masked GitLab variable here
  # a selector describing the element containing the username field at the sign-in HTML form
  DAST_USERNAME_FIELD: "id:email"
  # a selector describing the element containing the password field at the sign-in HTML form
  DAST_PASSWORD_FIELD: "id:password"
  # the selector of the element that when clicked will submit the login form or
  # the password form of a multi-page login process
  DAST_SUBMIT_FIELD: "css:button[type='submit']"
  # optional, URLs to skip during the authenticated scan; comma-separated,
  # no spaces in between
  DAST_EXCLUDE_URLS: "$DAST_WEBSITE/logout"
  # optional, used to verify authentication is successful by expecting this URL once the
  # login form has been submitted
  DAST_AUTH_VERIFICATION_URL: "$DAST_WEBSITE/profile"
  # optional, used to verify authentication is successful by expecting a selector to be
  #present on the page once the login form has been submitted
  DAST_AUTH_VERIFICATION_SELECTOR: "id:greeting"
  # optionally output an authentication debug report
  DAST_AUTH_REPORT: "true"
  # do a full scan, including active attacks
  DAST_FULL_SCAN_ENABLED: "true"
  DAST_BROWSER_SCAN: "true"

services: # use services to link your app container to the dast job
  - name: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
    alias: flask_demo

deploy:
  services:
  - name: docker:dind
    alias: dind
  image: docker:19.03.5
  stage: deploy
  script:
    - echo $CI_JOB_TOKEN | docker login -u gitlab-ci-token --password-stdin $CI_REGISTRY
    - docker pull $CI_REGISTRY_IMAGE:latest || true
    - docker build --tag $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA --tag $CI_REGISTRY_IMAGE:latest .
    - docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
    - docker push $CI_REGISTRY_IMAGE:latest
```

Literatur

[1] Invicti: *The Invicti AppSec Indicator Spring 2021 Edition: Acunetix Web Vulnerability Report*. en-US. 2021. URL: https://www.acunetix.com/white-papers/acunetix-web-application-vulnerability-report-2021/ [4. Sep. 2021].

[2] *DevOps*. de. Page Version ID: 211412763. Apr. 2021. URL: https://de.wikipedia.org/w/index.php?title=DevOps&oldid=211412763 [9. Sep. 2021].

[3] *Was ist DevOps? Eine Erläuterung | Microsoft Azure*. de. URL: https://azure.microsoft.com/de-de/overview/what-is-devops/ [9. Sep. 2021].

[4] Dr. Darrell R. Schrag: *DevOps: Shift Left to Reduce Failure*. en-US. Juni 2016. URL: https://devops.com/devops-shift-left-avoid-failure/ [4. Sep. 2021].

[5] GitLab: *A Seismic Shift in Application Security*. 2020. URL: https://about.gitlab.com/resources/downloads/gitlab-seismic-shift-in-application-security-whitepaper.pdf?mkt_tok=MTk0LVZWQy0yMjEAAAF89o7m5_lSbp4OyUGkBoLTYNLI10vTCNhjRBSkS_tiABEs-7sZLwaqkk.

[6] *Agile Softwareentwicklung*. de. Page Version ID: 214321057. Juli 2021. URL: https://de.wikipedia.org/w/index.php?title=Agile_Softwareentwicklung&oldid=214321057 [17. Sep. 2021].

[7] *Kanban*. de. URL: https://www.ionos.de/digitalguide/websites/web-entwicklung/kanban/ [17. Sep. 2021].

[8] Atlassian: *Scrum – Was es ist, was nicht und warum es so großartig ist*. de. URL: https://www.atlassian.com/de/agile/scrum [17. Sep. 2021].

[9] *A successful Git branching model*. en. URL: http://nvie.com/posts/a-successful-git-branching-model/ [8. Sep. 2021].

[10] *Understanding the GitHub flow · GitHub Guides*. URL: https://guides.github.com/introduction/flow/ [8. Sep. 2021].

[11] *Burp Suite Professional*. URL: https://portswigger.net/burp/pro [19. Sep. 2021].

[12] *OWASP Top Ten Web Application Security Risks | OWASP*. en. URL: https://owasp.org/www-project-top-ten/ [19. Sep. 2021].

[13] *The ZAP Homepage*. en. URL: https://www.zaproxy.org/ [19. Sep. 2021].

[14] Aat Team: *Burp Suite vs OWASP ZAP - Which is Better?* en-US. März 2021. URL: https://allabouttesting.org/burp-suite-vs-owasp-zap-which-is-better/ [19. Sep. 2021].

[15] *GitLab Pricing*. en. URL: https://about.gitlab.com/pricing/ [19. Sep. 2021].

[16] *Predefined variables reference | GitLab*. en-us. URL: https://docs.gitlab.com/ee/ci/variables/predefined_variables.html [19. Sep. 2021].

[17] *Dynamic Application Security Testing (DAST) | GitLab*. en-us. URL: https://docs.gitlab.com/ee/user/application_security/dast/ [18. Sep. 2021].

[18] *OWASP ZAP Baseline Scan*. URL: https://www.zaproxy.org/docs/docker/baseline-scan/ [19. Sep. 2021].

[19] *OWASP ZAP Full Scan*. URL: https://www.zaproxy.org/docs/docker/full-scan/ [19. Sep. 2021].

Abbildungsverzeichnis