Seminar Softwareentwicklung mit Dev(Sec)Ops

Dynamic Application Security Testing (DAST)

Tobias Kiehnlein

Inhaltsverzeichnis

Glossar

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. Wie sich allerdings herausstellt, ist diese Aufgabe nicht trivial. Warum sollte also dieser Aufwand getrieben werden? Hier spielen viele Faktoren eine Rolle, allerdings wird sich diese Arbeit primär auf den Aspekt der 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 und häufig 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. Vor allem unter dem Gesichtspunkt, dass einige triviale Tests wiederholt durchgeführt werden müssen, da Codeänderungen große Auswirkungen im gesamten Projekt haben können.

Die Lösung scheint also einfach: Mit automatisierten Sicherheitstest, welche die Software auf die gängigsten und bekanntesten Lücken prüft und den Entwicklern und Sicherheitsexperten direktes Feedback gibt, um Entwickler auf mögliche Sicherheitslücken zu sensibilisieren und Sicherheitsexperten um repetitive Arbeit zu entlasten und somit auch Geld und Zeit zu sparen. So kann eine sichere und effiziente Softwareentwicklung gewährleistet werden.

2 Analyse eines optimalen DevSecOps Prozesses

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

Zur Beantwortung dieser Fragen sollte man sich zunächst anschauen, wie ein DevSecOps Prozess am besten gegliedert ist und ihn dann in seinen einzelnen Teilen zu betrachten. 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 zum Beispiel von einem Build oder Deployment abhängen können, genauso wie ein Deployment von einem Test und offensichtlich einem Build abhängen kann. Daher wird in der Arbeit im Folgenden immer ein kleiner Schritt in der Pipeline beschrieben. Diese sind chronologisch so angeordnet, wie sie auch im finalen Prozess vorhanden sein sollten.

2.1 Git Struktur/Branching

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[2] eine gute Möglichkeit ist, um sein git zu ordnen. Allerdings ist dieses Modell sehr kompliziert, weshalb von diesem Modell heutzutage oftmals abgeraten wird. Ein häufig vorgeschlagenes Alternativmodell stellt GitHub flow[3] dar. Dieses Modell ist im Gegensatz zu git-flow sehr einfach gestrickt. So einfach, dass Testumgebungen nicht wirklich anhand der gegebenen Branches dargestellt werden können. Deshalb empfiehlt sich die Verwendung eines Modells, das zwischen git-flow und GitHub flow anzuordnen ist.

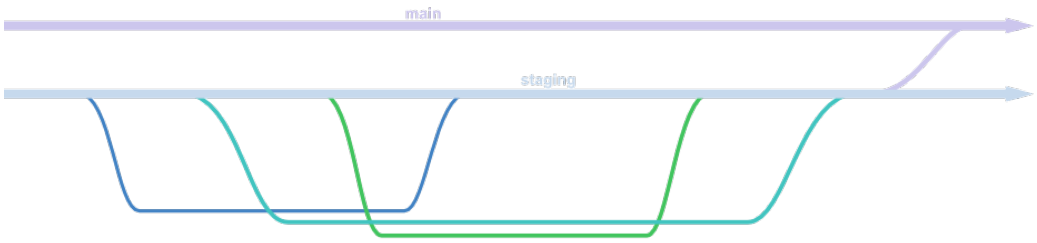

Abbildung 1: Graphische Darstellung der Kombination aus git-flow und GitHub Flow

Das hier gezeigte Modell basiert relativ stark auf GitHub flow. Es gibt einen Hauptbranch (Main), welcher zu jeder Zeit veröffentlichbar sein sollte. Im Gegensatz zu GitHub flow ist dies allerdings nicht der main Branch, sondern ein staging Branch. Dieser kann dann Nutzern als Beta-Version zur Verfügung gestellt werden. So haben Nutzer die Möglichkeit neue Features sofort zu testen und direktes Feedback in den Entwicklungsprozess zurück zu bringen. 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. Zudem darf auf den main branch allerdings nur eine getestete Version vom Staging Branch deployed werden. Entwicklung sollte allerdings, genau wie bei GitHub flow, auf dem Main und Staging branch nicht stattfinden. Hierfür werden für jedes Feature eigene Branches aufgemacht (dunkelblau, hellblau und grün in Abbildung 1). Auf diesen werden alle commits gepushed, bis das feature bereit ist, in den Staging branch gemerged zu werden. Dies bedeutet, dass das Feature vollständig funktioniert. Sobald dieses Stadium erreicht ist, wird ein Pull Request in den Staging Branch erstellt. Hiermit wird der Review Prozess eingeleitet. Zunächst laufen natürlich 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. 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 eine realistische Testumgebung Feedback in den Entwicklungsprozess zurückspielen 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.

Die Vorteile gegenüber GitHub flow liegen auf der Hand. Ein Branchingmodell, welches nur minimal komplizierter ist, bringt eine stark erhöhte Stabilität im Programm, da Features zunächst an eine kleine Testgruppe ausgerollt wird, Feedback gesammelt wird und dieses erneut in den Entwicklungsprozess einfließen kann, bevor ein Feature an alle Nutzer ausgerollt wird. Alle Vorteile von GitHub Flow gegenüber git-flow bleiben allerdings erhalten, weshalb dies immer noch ein exzellentes Modell für DevOps ist.

2.2 Coding Guidelines/Code Architecture

Beginnen sollte man hier mit den Syntax- und Architekturtests. Bevor die Funktionalität einer Software getestet wird, sollte man immer sicherstellen, dass die Architektur und Syntax den gewählten Konventionen entspricht. Für die Bewertung der Position eines Schrittes in einer Pipeline gibt es mehrere ausschlaggebenden Punkte. Wichtig sind aber vor allem, die Abhängigkeit zu anderen Schritten, sowie die Zeitkomplexität die mit dem Job einhergeht. Nun wird auch schnell offensichtlich, warum dieser Schritt zu Beginn der Pipeline angeordnet sein sollte. Es bestehen keine Abhängigkeiten zu anderen Jobs, weshalb dies problemfrei zu Beginn der Pipeline ausgeführt werden kann. Außerdem sind Syntax- und Architekturtests in relativ kurzer Zeit auszuführen. Dies ist erstrebenswert, da die Pipeline möglichst früh fehlschlagen sollte, wenn Fehler existieren, um unnötige Verzögerungen im Entwicklungsprozess zu vermeiden. Zudem lassen sich Syntaxfehler zumeist schnell beheben, um die nächsten Schritte in der Pipeline erreichen zu können.

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

3.1 Burp Suite

3.2 OWASP ZAP

3.3 GitLab

3.4 Fazit

Literatur

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

[2] *A Successful Git Branching Model*. nvie.com. URL: http://nvie.com/posts/a-successful-git-branching-model/ [8. Sep. 2021].

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

Abbildungsverzeichnis