Sicherheitslücken durch Open-Source-Abhängigkeiten in Java-basierten, Cloud-nativen Applikationen finden und beheben

Mathias Conradt, Snyk

Dieser Artikel zeigt auf, wie sich mithilfe von Tools wie Snyk Sicherheitslücken durch Open-Source-Abhängigkeiten in Java-basierten, Cloud-nativen Applikationen finden und beheben lassen, bevor diese produktiv gestellt werden.

In modernen Softwareprojekten verlassen wir uns sehr auf Frameworks und Bibliotheken von Drittanbietern. Nur 10 bis 20 Prozent der gesamten Codebasis ist eigens geschriebener Code, die restlichen 80 bis 90 Prozent entstammen Open-Source-Abhängigkeiten durch verwendete Frameworks oder Support-Libraries. Hinzu kommt, dass neben dem eigentlichen Applikationscode und

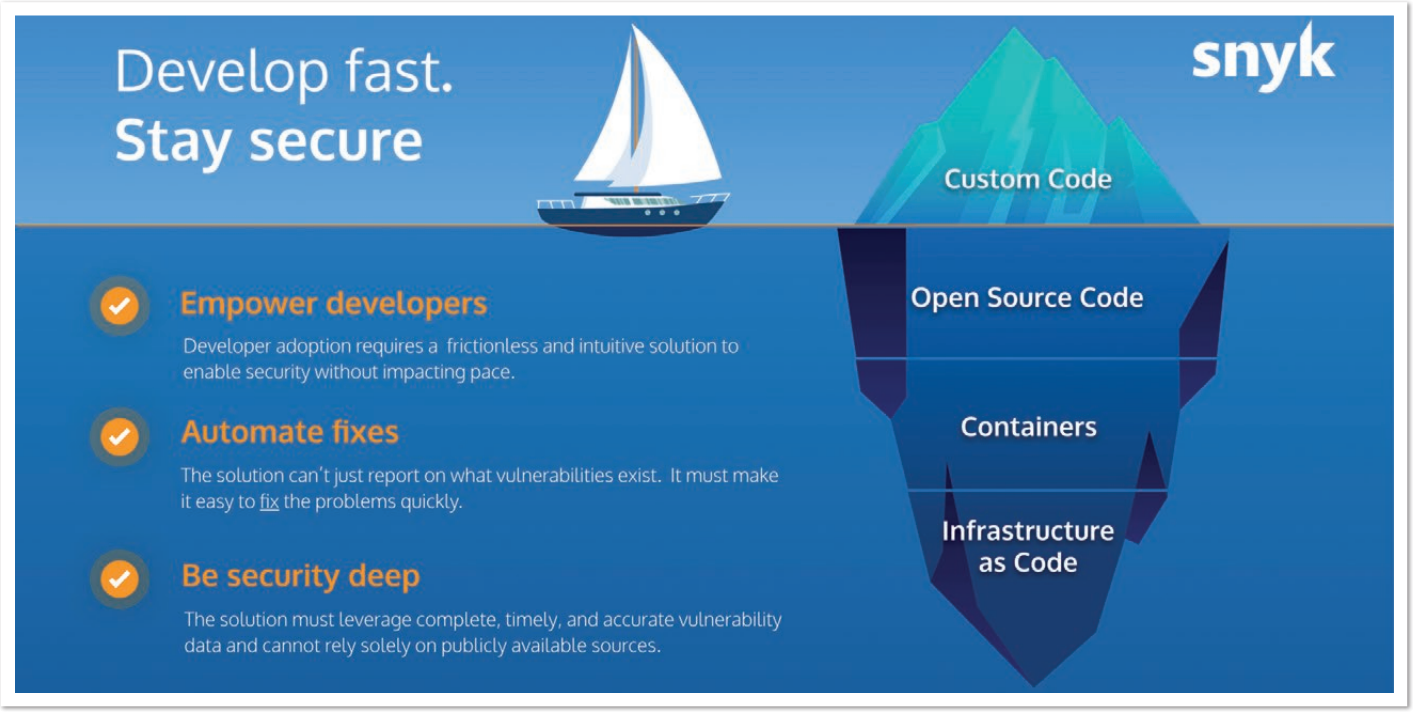

Abbildung 1: Die moderne Software-Supply-Chain (© Snyk)

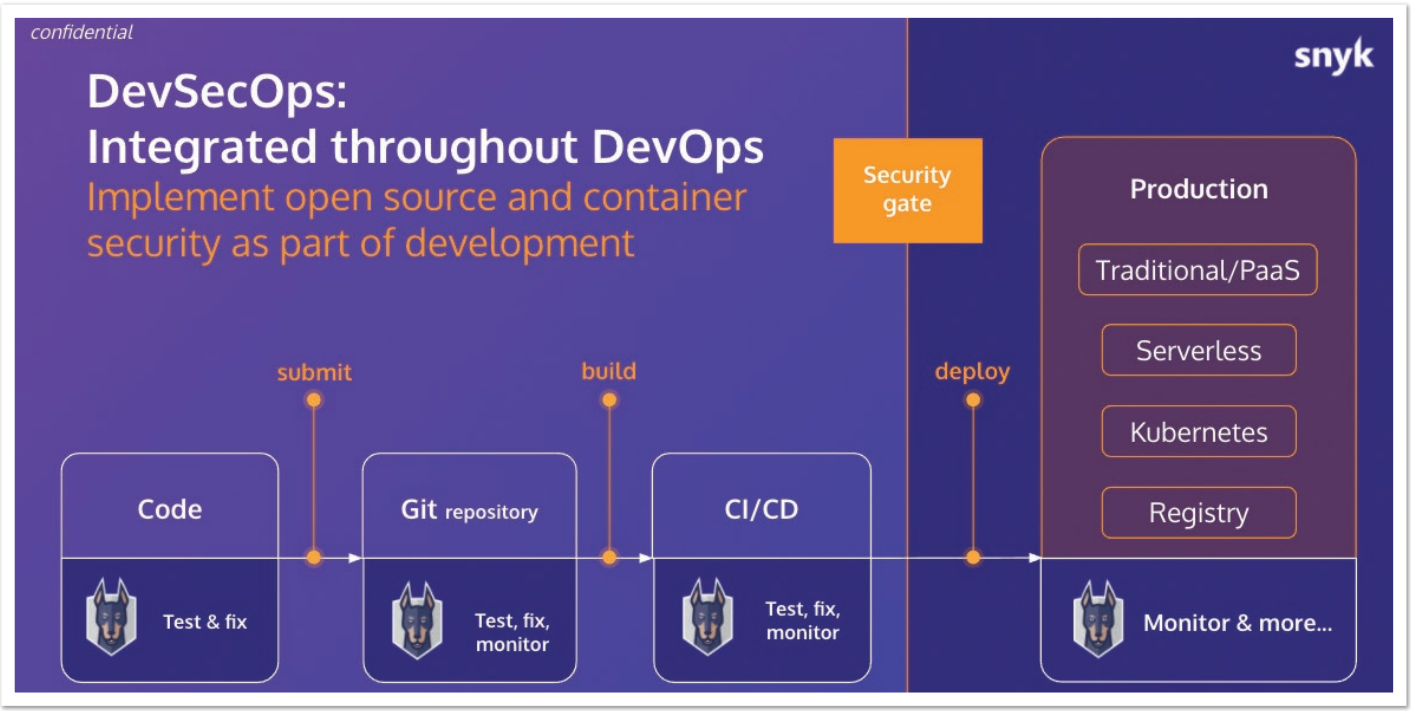

Abbildung 2: Snyk unterstützt entlang des gesamten SDLC (© Snyk)

dessen Abhängigkeiten bei Cloud-nativen Anwendungen auch noch die Container-Definition sowie Infrastruktur als Code vorliegt *(siehe Abbildung 1)*. Diese liegen etwa in Form von Dockerfiles, Kubernetes-Deployment- oder Terraform-Files vor. Dies alles bildet entsprechende Angriffsfläche, die – zumindest teilweise – im Verantwortungsbereich des Application Developer liegt.

Während ich mich als Entwickler eigentlich auf die Business-Logik konzentrieren möchte und nicht unbedingt ein Security-Experte bin, möchte ich trotzdem sicherstellen, dass meine Applikation vor Angriffen sicher ist.

Schwachstellen zu finden und zu beheben, ist dann am wenigsten kosten- und zeitintensiv, je früher ich damit links in meinem SDLC anfange („Shift-Left"), bestenfalls gleich im lokalen Terminal oder in der IDE, oder im Git Repository *(siehe Abbildung 2)*.

Snyk

Snyk ist ein Freemium-Tool, das Open-Source-Abhängigkeiten in Manifest-Files (pom.xml, build.gradle) ausliest, einen Dependency-Tree generiert und diesen dann mit seiner Schwachstellen-Datenbank vergleicht. Die Herausforderung für Entwickler, ihre Applikation in Bezug auf Schwachstellen abzusichern, ist zweigeteilt:

Abbildung 3: Generierter Dependency-Tree, generiert basierend auf der pom.xml (© Snyk)

Zum einen befinden sich 80 Prozent aller Schwachstellen in indirekten Abhängigkeiten, also jenen, denen ein Entwickler sich nicht unbedingt bewusst ist, da er diese nicht explizit in seinem pom.xml angefragt hat. Diese ganzen indirekten Abhängigkeiten teilweise in zweiter, dritter oder gar vierter Ebene zu identifizieren, ist ohne ein entsprechendes Tool mühsame Handarbeit.

Zum anderen ist das Wissen über eine Schwachstelle einer bestimmten Version einer Abhängigkeit nur die halbe Miete, wenn ich nicht weiß, wie ich diese beheben kann. Ein entsprechendes Tool gibt hierüber ebenfalls sofort Auskunft *(siehe Abbildung 3)*.

Testen in der CLI

Um ein Tool wie Snyk zu nutzen, legt man sich einen kostenfreien Account unter snyk.io an und installiert dann das zugehörige CLI *[1]* mittels `npm`, `scoop` oder `brew`, je nach System *(siehe Listing 1)*. Im Blog *[2]* ist ein CLI-Cheat-Sheet verfügbar, das einen guten Überblick über alle verfügbaren Aktionen und Optionen bietet. Um mein Projekt, in meinem Beispiel ein Maven-basiertes Projekt, zu testen, reicht der einfache, in *Listing 2* gezeigte Befehl.

```
# Installation mittels npm
npm install -g snyk
#
# alternativ: Installation mittels Brew
brew install snyk
#
# Authentifizierung
snyk auth
```

Listing 1: Snyk-CLI-Installation und -Authentifizierung

```
mvn install
snyk test
```

Listing 2: Snyk-Test via CLI

```
snyk monitor
```

Listing 3: Snyk-Monitoring via CLI

Das Ergebnis ist ein entsprechender Report mit gefundenen Schwachstellen sowie der Information, ob diese jeweils mittels Upgrade oder Patch beseitigt werden können.

Beispielsweise können alle in *Abbildung 4* gezeigten Schwachstellen (wie XSS, DoS und Remote-Code-Execution-Probleme, jeweils gelb oder rot markiert), die durch die direkte Abhängigkeit `struts2-core@2.3.20` eingeführt wurden, durch ein einfaches Upgrade auf `struts2-core@2.5.26` behoben werden. Gelb markierte Schwachstellen stehen für einen mittleren Schweregrad, wohingegen rot markierte Schwachstellen für einen hohen Schweregrad stehen.

Bei einer Upgrade-Empfehlung wird dabei versucht, den minimal möglichen Sprung auf die nächste, nicht-betroffene Version zu machen, um „Breaking Changes" zu vermeiden. Major Upgrades werden also – soweit möglich – vermieden.

Um das Projekt langfristig zu monitoren – oder gar den Snapshot, der sich in der Produktivumgebung befindet – und es der Snyk Admin UI hinzuzufügen, um somit regelmäßig auf Schwachstellen hin zu prüfen, auch wenn der Entwickler den Code selbst eine Zeit lang gar nicht anfasst, kann der folgende Befehl ausgeführt werden *(siehe Listing 3)*.

Testen in der IDE

Wer das Testen lieber in der IDE statt IntelliJ vornimmt, der kann auf das Snyk-Plug-in *[3]* für IntelliJ oder Eclipse zurückgreifen, die im jeweiligen Marketplace der IDE zu finden sind. Hier integriert sich dieser Test direkt in die IDE, die Information ist die gleiche, lediglich die Darstellung des Reports ändert sich etwas *(siehe Abbildung 5)*.

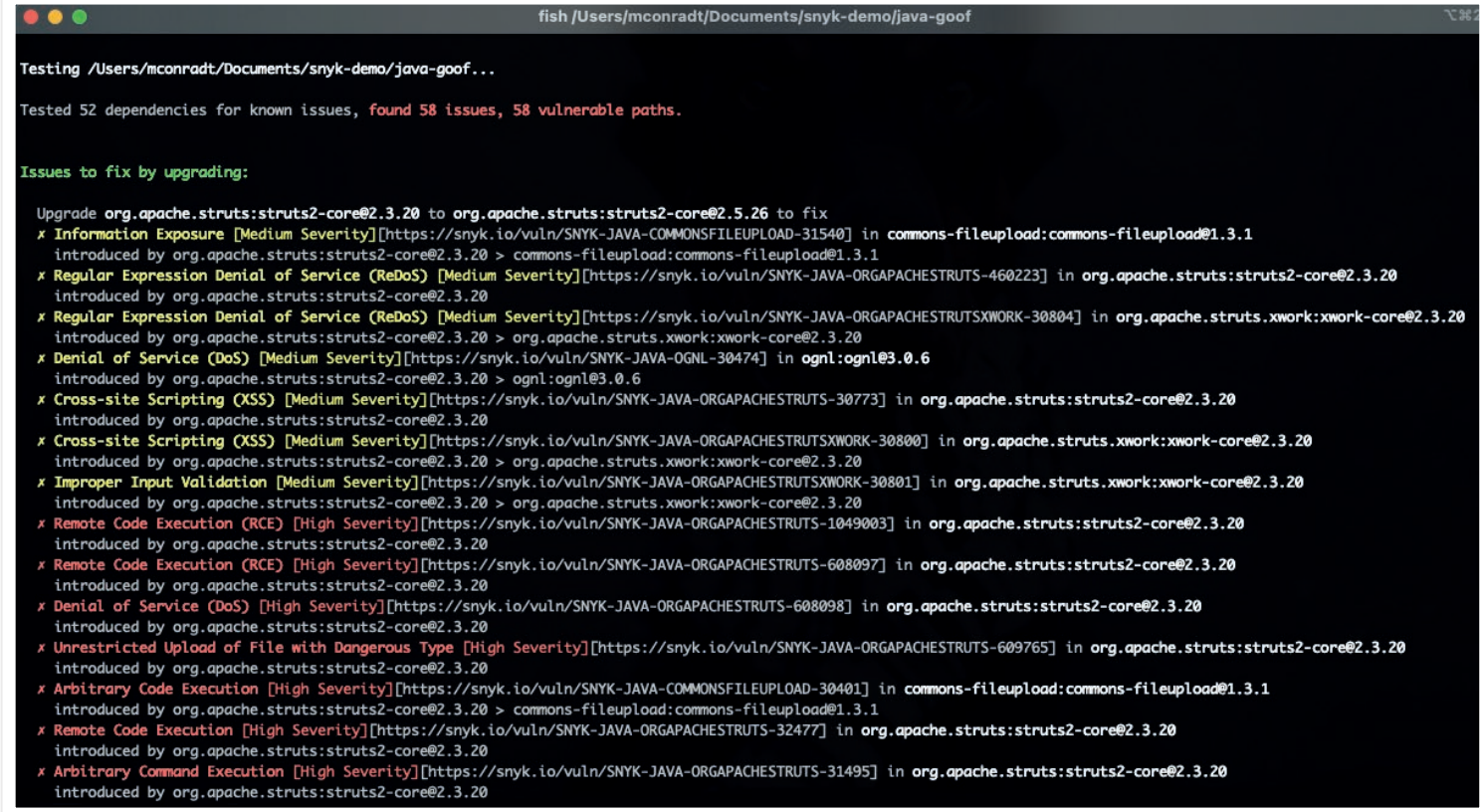

Abbildung 4: Snyk-Test im CLI (© Snyk)

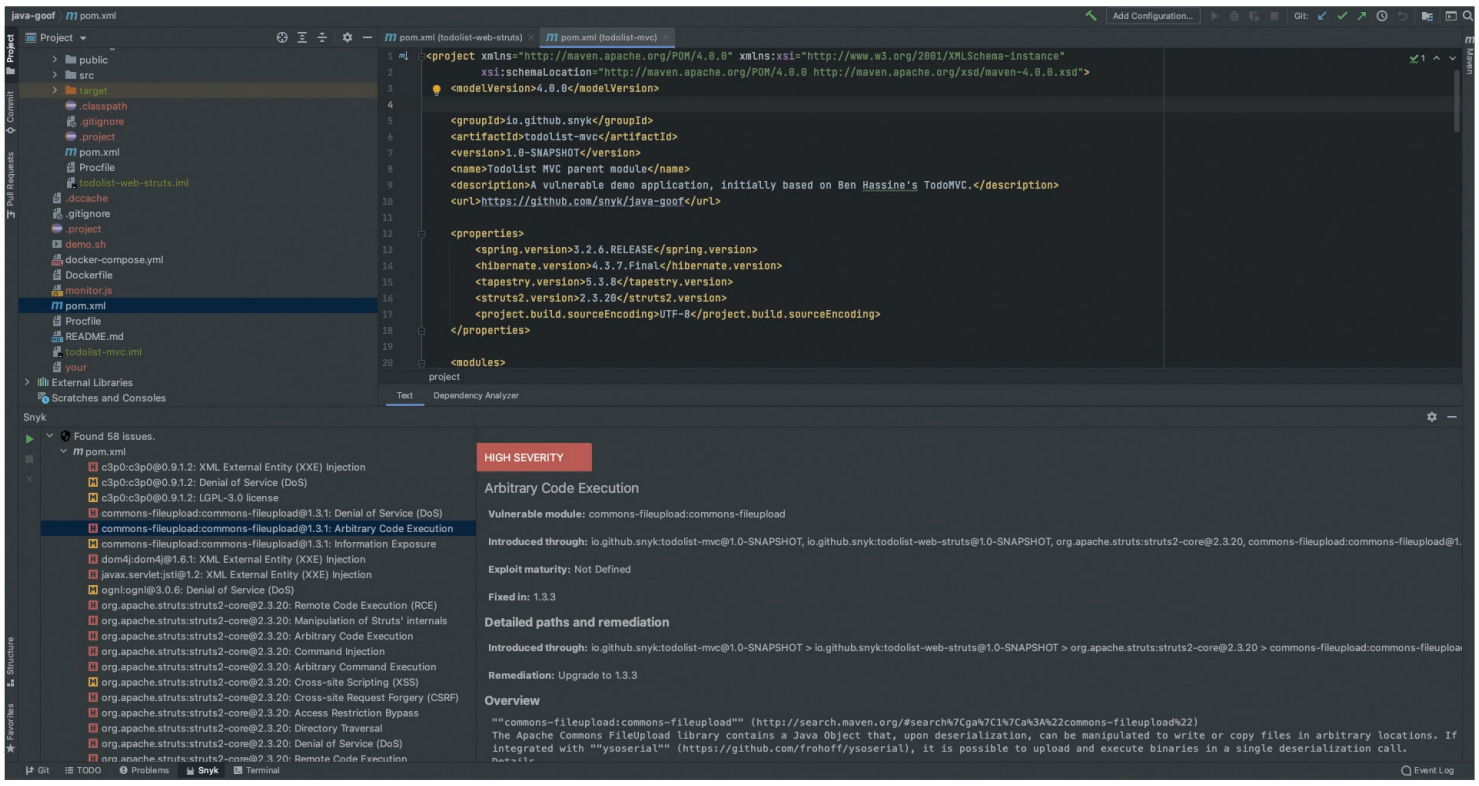

Abbildung 5: Snyk-Test in der IDE (© Snyk)

Testen des Git Repository

Snyk bietet Integrationen in GitHub, GitLab, Bitbucket, Azure Repos. Dies erlaubt das Testen, Beheben sowie das regelmäßige Monitoring von Projekten. Beispielhaft wird dies im Folgenden anhand der GitHub-Integration gezeigt. Es lassen sich alle oder einzelne Projekte einer GitHub-Organisation zu Snyk hinzufügen. Snyk identifiziert entsprechende Manifest-Dateien (pom.xml, build.gradle) und führt einen ersten Test durch.

Das Ergebnis wird daraufhin in der Weboberfläche angezeigt *(siehe Abbildung 6)*. Die Projektliste zeigt die Anzahl der gefundenen Schwachstellen mit ihrem jeweiligen Schweregrad (High/Medium/Low) an. Besteht ein Projekt aus mehreren Modulen oder Unterprojekten, werden alle Manifest-Dateien angezeigt. Mit Klick auf eine dieser Dateien gelangt man in die Report-Ansicht *(siehe Abbildung 7)*.

In dem gezeigten Beispielprojekt (pom.xml) wurden 57 Schwachstellen in 49 (direkten als auch indirekten) Abhängigkeiten gefunden.

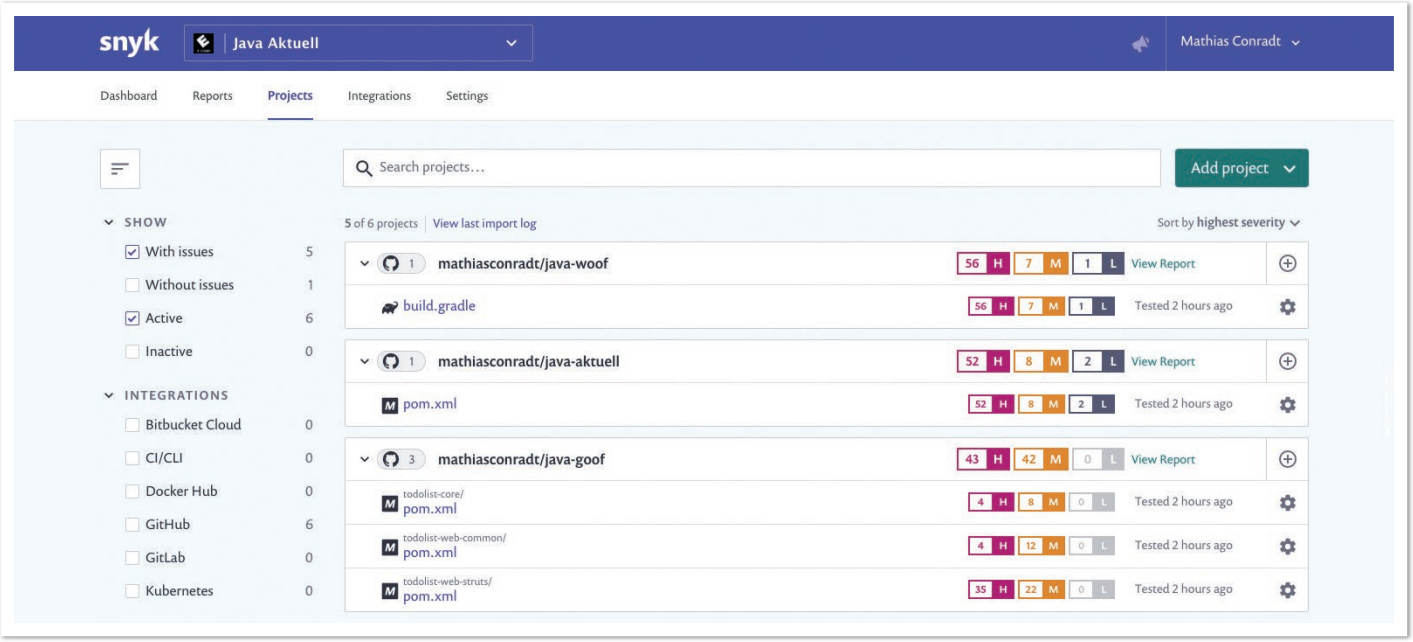

Abbildung 6: Snyk-Test mittels Git-Integration und Web UI (© Snyk)

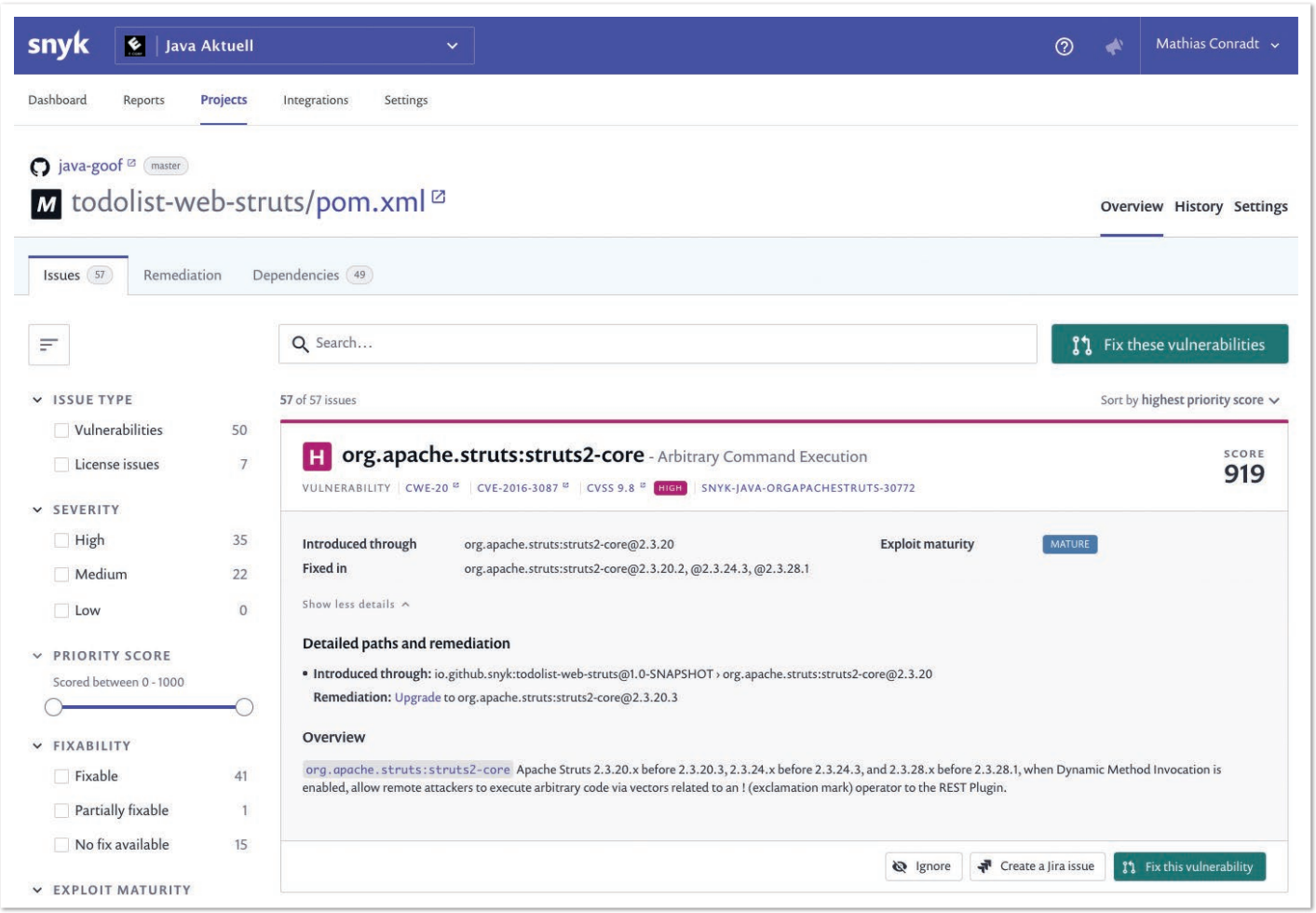

Abbildung 7: Report-Ansicht eines Projektes in der Snyk Web UI (© Snyk)

Komplexe Projekte können teilweise Hunderte von Schwachstellen aufweisen. Um den Entwickler nicht zu überfrachten und zu überfordern, ist es wichtig, eine entsprechende **Priorisierung** vorzunehmen.

Hierzu werden gefundene Schwachstellen zunächst grob nach Schweregrad High/Medium/Low gruppiert, darüber hinaus ist jeder Schwachstelle ein sogenannter Priority-Score zugewiesen. Dieser wird von Snyk berechnet und kalkuliert verschiedene Faktoren mit ein:

- CVSS Score
- Verfügbarkeit eines Fix
- „Exploit Maturity" – diese kann in drei Stufen vorliegen:
  - „Mature": Es ist ein veröffentlichter Code-Exploit verfügbar, der leicht für diese Sicherheitslücke verwendet werden kann.
  - „Proof of Concept": Ein veröffentlichtes, theoretisches Proof-of-Concept oder eine detaillierte Erklärung sind verfügbar, die zeigen, wie diese Schwachstelle ausgenutzt werden kann.
  - „No known Exploit": Für diese Sicherheitslücke wurde weder ein Proof-of-Concept-Code noch ein Exploit gefunden, beziehungsweise sind keine öffentlich verfügbar.

Der Report listet die Schwachstellen absteigend nach Priority-Score auf und gibt Details über gefundene Schwachstellen. Die Details beinhalten Informationen zum Namen und zur Kategorie der Schwachstelle, zu betroffenen Modulen und Versionen, in denen die Schwachstelle auftritt, ob und in welcher Version diese behoben ist, den Abhängigkeitspfad sowie Hintergrundinformation. Ein Link zur Snyk-Schwachstellen-Datenbank liefert optional weitere Hintergrundinformationen.

Von diesem Bildschirm aus hat der Entwickler nun die Möglichkeit, sofern ein Fix vorhanden ist, direkte einen Pull-Request zu erstellen. Mit Klick auf „Fix this Vulnerability" erstellt Snyk automatisch für die ausgewählten Schwachstellen einen Fix-Branch und Pull-Request *(siehe Abbildung 8)*.

Der Pull-Request beinhaltet das entsprechende Upgrade der direkten Abhängigkeit, durch die die Schwachstelle in das Projekt eingeführt wurde *(siehe Abbildung 9)*.

Das gezeigte Beispiel zeigt den Workflow eines **Pull-Request**, der manuell durch den Entwickler im Snyk Web UI angestoßen wurde.

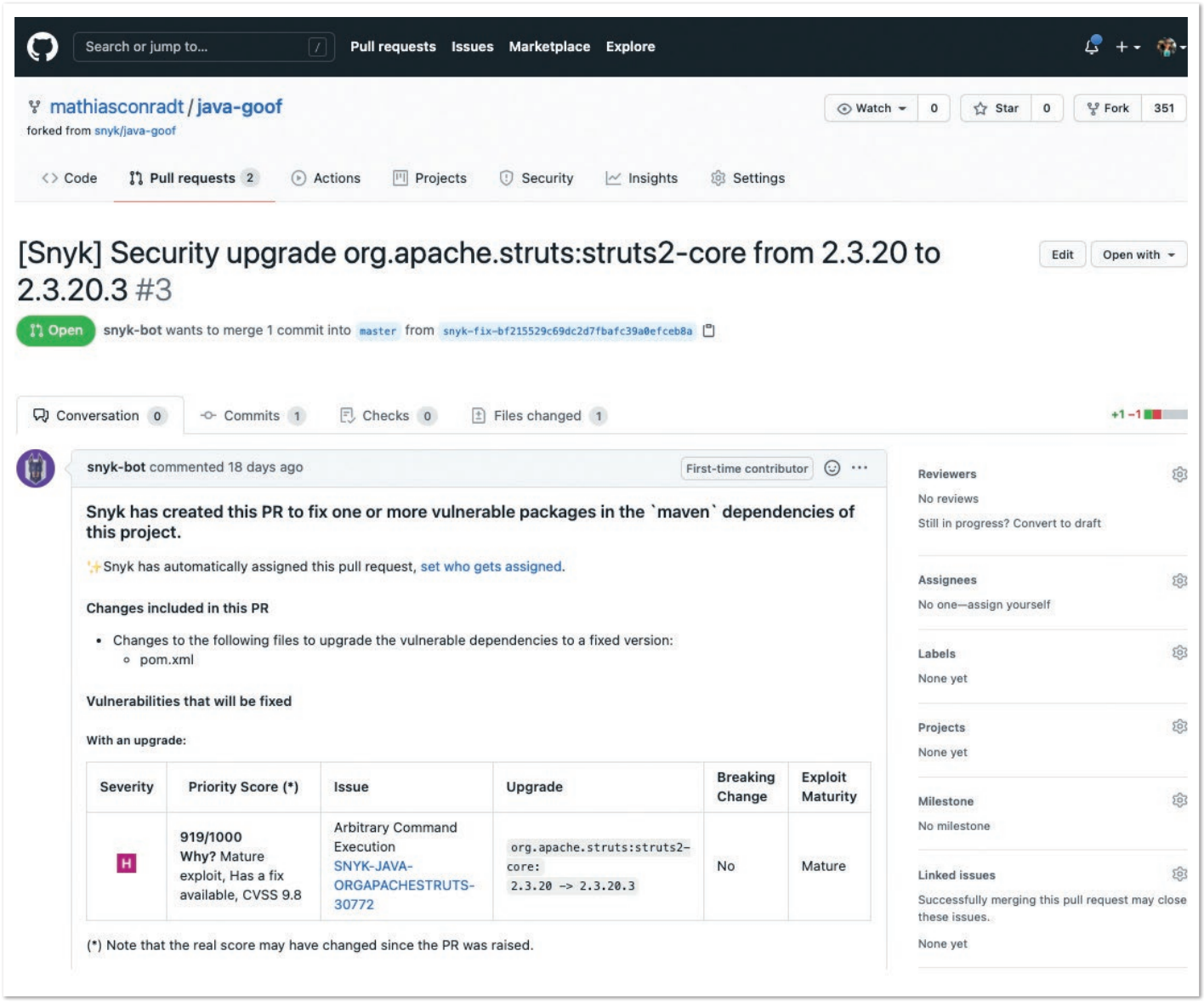

Abbildung 8: Automatisch generierter Pull-Request in GitHub (© Snyk)

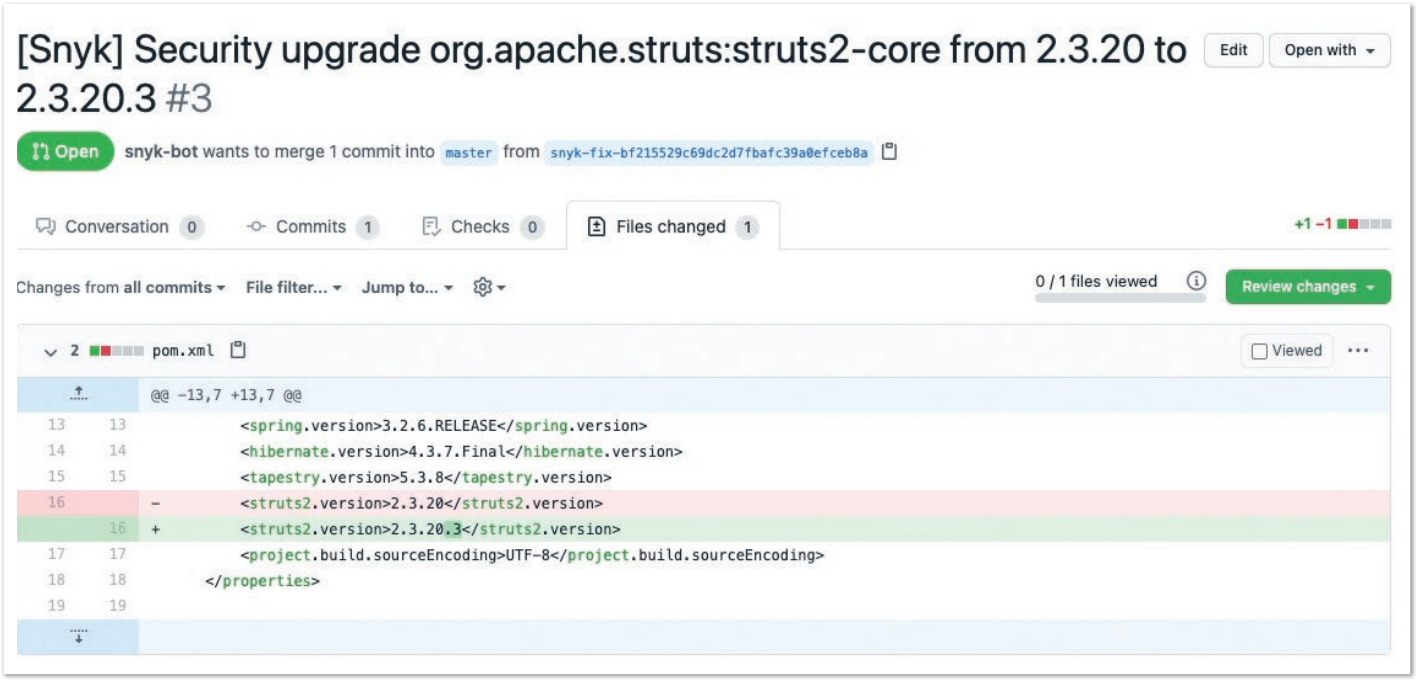

Abbildung 9: Upgrade der direkten Abhängigkeit in der pom.xml (© Snyk)

Natürlich lässt sich dies auch **automatisieren**. Es ist möglich, Snyk so zu konfigurieren, dass das Git Repository automatisch – täglich oder wöchentlich – gemonitort wird und Pull-Requests automatisch erstellt werden.

Die Datenbasis

Ein Security Tool ist nur so gut wie die dahinter liegende Datenbasis. Viele Tools sind ungenau in dem Sinne, dass sie viele „False Positives" zurückliefern, das heißt angebliche Schwachstellen, die tatsächlich keine sind. Im Falle von Snyk werden Schwachstellen aus den unterschiedlichsten Quellen immer nochmal manuell durch ein Security Research Team verifiziert, bevor sie letztlich in die eigene Datenbank Einzug halten.

Auf welcher Datenbasis basieren die gefundenen Schwachstellen? Die Datenbasis, auf der die Snyk-Schwachstellendatenbank *[4]* basiert, ist auf fünf Säulen aufgebaut:

1. Angereicherte Daten aus zahlreichen Schwachstellen-Datenbanken wie der CVE, NVD und mehr. Die aus diesen Ressourcen gewonnenen Daten werden analysiert, getestet und angereichert, bevor sie in die Snyk-Datenbank aufgenommen werden.
2. Proprietäre Forschung nach neuen Schwachstellen: Das Snyk Research Team arbeitet daran, schwerwiegende Schwachstellen in Schlüsselkomponenten aufzudecken – so wurden 2019 beispielsweise 54 Zero-Day-Schwachstellen entdeckt.
3. Threat-Intelligence-Systeme: Diese scannen Security-Bulletins, Jira-Boards und GitHub-Commits, um automatisch Schwachstellen zu identifizieren, die noch nicht gemeldet wurden.
4. Beziehung zur Community: Snyk arbeitet mit der Community zusammen und betreibt Bug Bounties.
5. Zusammenarbeit mit der akademischen Welt: Das Team arbeitet mit akademischen Einrichtungen wie Berkeley, Virginia Tech und Waterloo zusammen, um Tools, Methoden und Daten auszutauschen. Die Ergebnisse werden dann exklusiv von Snyk veröffentlicht.

Die hier gezeigten Projektbeispiele verwenden Java und Maven. Snyk unterstützt darüber hinaus viele weitere Sprachen (Java, Node, Python, .NET, Go, Ruby, Kotlin, PHP) beziehungsweise deren Package Manager.

Container Security

Das Verpacken einer Java-Anwendung in einen Container ermöglicht es, die komplette Anwendung, einschließlich der JRE, Konfigurationseinstellungen sowie Abhängigkeiten auf Betriebssystemebene und die Build-Artefakte in eigenständigen, bereitstellbaren Artefakten, den sogenannten Container-Images, zu definieren. Diese Images werden in Form von Code (zum Beispiel Dockerfiles) definiert, was eine vollständige Wiederholbarkeit bei ihrer Erstellung ermöglicht und Entwicklern die Möglichkeit gibt, dieselbe Plattform in allen Umgebungen auszuführen. Schließlich können Entwickler mit Containern einfacher mit neuen Plattformversionen oder anderen Änderungen direkt auf ihren Desktops experimentieren, ohne dass spezielle Berechtigungen erforderlich sind.

Wer seine Applikation im Docker-Container ausliefert, definiert diesen im Dockerfile und mit entsprechender Auswahl eines geeigneten Base-Image. Nur, welches Base-Image ist am geeignetsten und sichersten?

Das richtige Docker-Image wählen

Wenn wir ein Docker-Image erstellen, erstellen wir dieses Image auf der Grundlage eines Image, das wir aus Docker Hub ziehen. Dies nennen wir das Base-Image. Das Base-Image ist die Grundlage für das neue Image, das wir für unsere Java-Anwendung erstellen wollen. Das Basis-Image, das wir wählen, ist wichtig, weil es uns erlaubt, alles zu nutzen, was in diesem Image verfügbar ist. Dies hat jedoch seinen Preis – wenn ein Basis-Image eine Sicherheitslücke aufweist, erben wir diese in unserem neu erstellten Image.

Bei der Erstellung eines Docker-Image sollten wir nur die nötigsten Ressourcen zuweisen, sodass die Applikation korrekt funktioniert.

Das bedeutet, dass wir zunächst eine geeignete Java-Laufzeitumgebung (JRE) für unser Produktions-Image verwenden sollten und nicht das komplette Java Development Kit (JDK). Darüber hinaus sollte unser Produktions-Image kein Build-System wie Maven oder Gradle enthalten. Das Produkt eines Builds, zum Beispiel eine jar-Datei, sollte ausreichen.

Auch wenn wir unsere Anwendung innerhalb eines Docker-Containers bauen möchten, können wir unser Build-Image mithilfe eines mehrstufigen Builds leicht von unserem Produktions-Image trennen.

Ein Beispiel: Ich möchte ein Docker-Java-Image für meine Java-Anwendung erstellen. Es handelt sich um eine Spring-Boot-basierte Anwendung, die mit Maven gebaut wurde und Java Version 8 benötigt. Der naive Weg, dieses Docker-Java-Image zu erstellen, wäre, das Basis-Image `maven:3-openjdk-8` zu wählen.

Mithilfe von Snyk können Schwachstellen in Docker-Images mittels CLI-Befehl *(siehe Listing 4)* einfach gefunden werden. In diesem Beispiel wollen wir das `maven:3-openjdk-8`-Image testen.

Der Scan *(siehe Abbildung 10)* zeigt uns, dass das besagte Image insgesamt 118 Schwachstellen beinhaltet, die durch 179 Abhängigkeiten Einzug gehalten haben. Der Report zeigt, welche System-Library in welcher Version von einer Schwachstelle betroffen ist, und im Falle eines möglichen Fix die jeweilige Version, auf die die Library upgegradet werden sollte.

Wenn wir hingegen auf ein schmaleres Alpine-Image wechseln, ohne Maven, und auch nur auf die JRE statt auf das JDK setzen, sehen wir, dass dieses Image lediglich 17 Abhängigkeiten beinhaltet und keine bekannten Schwachstellen enthält *(siehe Listing 5 und Abbildung 11)*.

```
$ snyk container test maven:3-openjdk-8
```

Listing 4: Snyk-Test des Docker-Image OpenJDK 8

```
snyk container test adoptopenjdk/openjdk8:alpine-jre
```

Listing 5: Snyk-Test des Docker-Image `adoptopenjdk/openjdk8:alpine-jre`

```
✗ High severity vulnerability found in gcc-8/libstdc++6
  Description: Information Exposure
  Info: https://snyk.io/vuln/SNYK-DEBIAN10-GCC8-347558
  Introduced through: gcc-8/libstdc++6@8.3.0-6, apt@1.8.2.2, meta-common-packages@meta
  From: gcc-8/libstdc++6@8.3.0-6
  From: apt@1.8.2.2 > gcc-8/libstdc++6@8.3.0-6
  From: apt@1.8.2.2 > apt/libapt-pkg5.0@1.8.2.2 > gcc-8/libstdc++6@8.3.0-6
  and 2 more...

✗ High severity vulnerability found in gcc-8/libstdc++6
  Description: Insufficient Entropy
  Info: https://snyk.io/vuln/SNYK-DEBIAN10-GCC8-469413
  Introduced through: gcc-8/libstdc++6@8.3.0-6, apt@1.8.2.2, meta-common-packages@meta
  From: gcc-8/libstdc++6@8.3.0-6
  From: apt@1.8.2.2 > gcc-8/libstdc++6@8.3.0-6
  From: apt@1.8.2.2 > apt/libapt-pkg5.0@1.8.2.2 > gcc-8/libstdc++6@8.3.0-6
  and 2 more...


Organization:      mathias.conradt-axp
Package manager:   deb
Project name:      docker-image|maven
Docker image:      maven:3-openjdk-8
Platform:          linux/amd64
Licenses:          enabled

Tested 179 dependencies for known issues, found 100 issues.
```

Abbildung 10: Ergebnisse des Image-Scans von `maven:3-openjdk-8` (© Snyk)

```
└─$ snyk container test adoptopenjdk/openjdk8:alpine-jre

Testing adoptopenjdk/openjdk8:alpine-jre...

Organization:      mathias.conradt-axp
Package manager:   apk
Project name:      docker-image|adoptopenjdk/openjdk8
Docker image:      adoptopenjdk/openjdk8:alpine-jre
Platform:          linux/amd64
Licenses:          enabled

✓ Tested 17 dependencies for known issues, no vulnerable paths found.
```

Abbildung 11: Ergebnisse des Image-Scans von `adoptopenjdk/openjdk8:alpine-jre` (© Snyk)

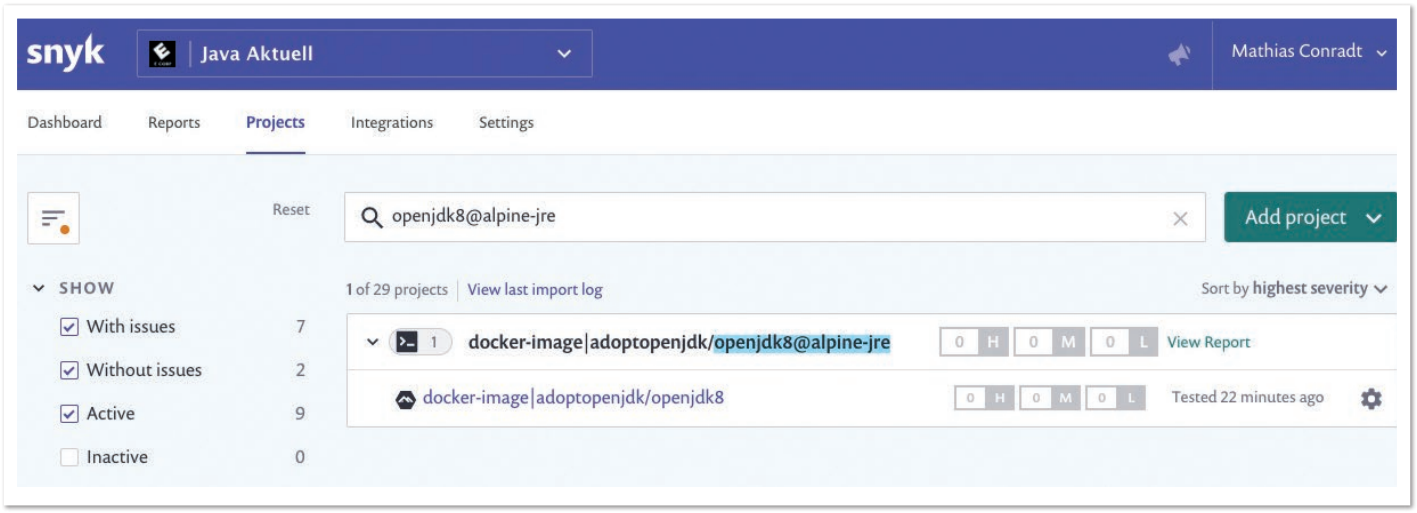

Abbildung 12: Snyk-Monitoring des Docker-Image `adoptopenjdk/openjdk8:alpine-jre` (© Snyk)

```
snyk container monitor adoptopenjdk/openjdk8:alpine-jre
```

Listing 6: Snyk-Monitoring des Docker-Image `adoptopenjdk/openjdk8:alpine-jre`

Das Docker-Image auch langfristig über die Snyk WebUI zu monitoren und bei neu gefundenen Schwachstellen informiert zu werden, ist mit dem `monitor`-Befehl in der Snyk-CLI *(siehe Listing 6 und Abbildung 12)* möglich.

Zusammenfassung

Bei Cloud-nativen Applikationen kommt der Entwickler auf mehreren Ebenen mit Code- und Open-Source-Abhängigkeiten in Berührung, sowohl bei der Applikation und deren direkten Open-Source-Dependencies selbst als auch bei den Open-Source-basierten System-Libraries im verwendeten Container-Image, in dem sie ausgeliefert wird. Diese beinhalten oftmals viele Sicherheits-schwachstellen, vor allem in älteren Versionen.

Snyk bietet hier eine kostenlose, entwicklerfreundliche Möglichkeit, all diese Komponenten entlang des gesamten DevOps-Prozesses auf Sicherheitslücken zu prüfen und diese möglichst früh zu beheben.

Referenzen

[1] *https://support.snyk.io/hc/en-us/articles/360003812538-Install-the-Snyk-CLI*
[2] *https://snyk.io/blog/snyk-cli-cheat-sheet/*
[3] *https://support.snyk.io/hc/en-us/sections/360001138118-IDE-tools*
[4] *https://snyk.io/vuln*

Mathias Conradt
Snyk
mathias.conradt@snyk.io

Mathias Conradt ist Sr. Solutions Engineer bei Snyk, einer „Cloud-native Application Security"-Plattform. Er verfügt über mehr als 20 Jahre Berufserfahrung im Bereich Software-Engineering und IT-Projektmanagement und besitzt diverse Zertifizierungen. Sein derzeitiger Schwerpunkt liegt auf Cybersecurity im Allgemeinen, wobei er sich auf Open Source Security spezialisiert hat. Mathias ist regelmäßiger Speaker bei diversen Security-bezogenen Branchenveranstaltungen.