CRaC: Java-Startzeiten in Amazon EKS verkürzen S. 40

D3: Daten gekonnt in Szene setzen S. 76

KI-gestützte vs. modellgetriebene Entwicklung S. 87

Javamagazin™

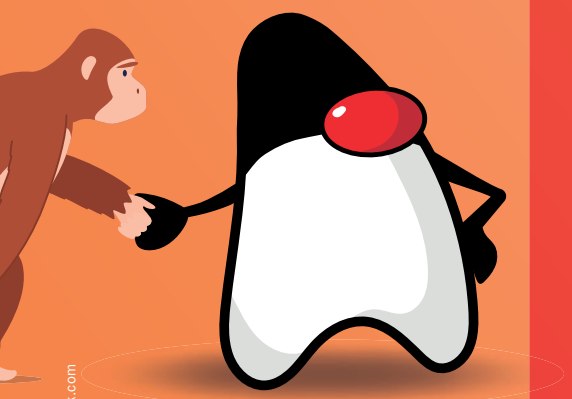
Java | Architektur | Software-Innovation

GenAI

Evolution

Revolution

Apokalypse?



Ausgabe 8.2024

Deutschland € 9,80
Österreich € 10,80
Schweiz sFr 19,50
Luxemburg € 11,15

4 194586 709801 08

entwickler.de

Die Software Bill of Material in Java – Teil 2

SBOMs verwalten mit Dependency-Track

Im ersten Teil der Serie habe ich gezeigt, wie man mit einem Plug-in für Maven oder Gradle sehr einfach eine SBOM erzeugen und sie mittels Grype auswerten kann, um bekannte CVEs in den Projektabhängigkeiten zu identifizieren und bei Erreichen bestimmter Schwellenwerte den Build abzubrechen. Für ein einfaches Projekt reicht dieses Tooling. Aber was ist, wenn ich viele Softwareartefakte habe? Wie kann ich den Überblick behalten?

von Tim Teulings

Entwickle ich ein Softwareprodukt, so kann ich mir über mein Build-Tool eine SBOM erstellen und sie z. B. einfach auf einem spezifischen Fileshare zur weiteren Verarbeitung und regelmäßigen Auswertung speichern. So weit, so gut. Aber wie sieht es aus, wenn ich mehrere Versionen meiner Software unterstützen muss? Eventuell arbeite ich ja nicht nur an einem Produkt, sondern an mehreren. Vielleicht habe ich intern weitere Komponenten mit einem eigenen Lebenszyklus im Einsatz, und eigentlich erwarte ich ja von den Herstellern der Software, die ich nutze, auch SBOMs, die ich auswerten möchte. Schnell bin ich bei einer größeren Anzahl von SBOMs, die es zu verwalten und auch regelmäßig auszuwerten gilt. Ein Tool hierfür wäre praktisch und dazu ein Prozess, den ich nicht in jeder Build Pipeline wieder und wieder neu abbilden muss. Zudem sollte die Analyse kontinuierlich und nicht nur im Fall von Änderungen und neuen Pipeline Runs stattfinden. Schließlich können neue Sicherheitslücken jederzeit gefunden werden.

Mit Dependency-Track [1] bietet das Open Worldwide Application Security Project (OWASP) [2] ein kostenloses Open-Source-Werkzeug, das es ermöglicht, SBOMs aus verschiedenen Quellen zentral zu sammeln und manuelle sowie automatische Analysen durchzuführen. Es ist leicht zu installieren und bietet umfangreiche Möglichkeiten. Perfekt, um in das Thema tiefer einzusteigen. Wir schauen uns das Werkzeug im Folgenden einmal an.

Testinstallation Dependency-Track

Für einfache Tests lässt sich Dependency-Track am besten über docker-compose [3] installieren [4] und schnell initial starten.

```
curl -LO https://dependencytrack.org/docker-compose.yml
docker-compose up -d
```

Danach kann man unter *http://localhost:8080/* das Web-UI erreichen und unter Port 8081 das umfangreiche REST API. Standard-Log-in/Passwort sind *admin/admin* und müssen beim ersten Log-in angepasst werden.

Im Bereich Administration sind danach einige Konfigurationen vorzunehmen: Unter Access Management/Teams legt man ein neues Team an und gibt diesem – mittels Auswahl aus der Liste – folgende Berechtigungen (Permission): *BOM_UPLOAD*, *PORTFOLIO_MANAGEMENT*, *PROJECT_CREATION_UPLOAD*, *VIEW_PORTFOLIO*, *VIEW_VULNERABILITY* sowie *VIEW_POLICY_VIOLATION* (**Abb. 1**). Das in der Übersicht zum ausgewählten Team angezeigte API-Token sollte man sich kopieren.

Artikelserie

Teil 1: Der Lieferschein für deine Software
Teil 2: SBOMs verwalten mit Dependency-Track

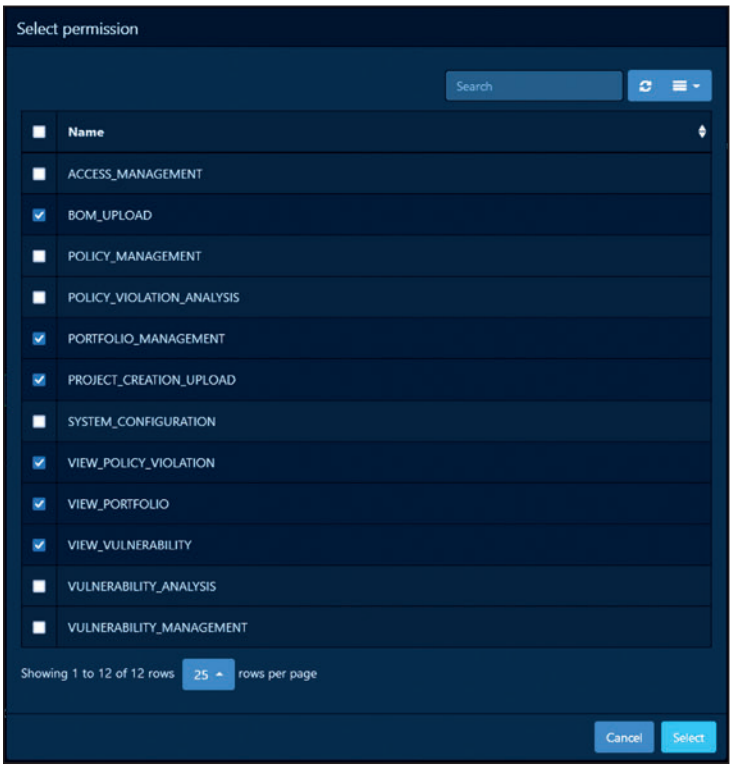


Abb. 1: Dialog zur Auswahl der Rechte eines Teams mit den hier notwendigen Rechten

Im Task Scheduler sollten für den Test die Zeiten auf eine Stunde reduziert werden. Ebenso sollt man Enable mirroring via API unter Vulnerability Sources/National Database aktivieren. Schließlich muss OSV für das Ökosystem Maven unter Vulnerability Sources/Google OSV Advisories (beta) aktiviert werden. Damit hat man alle frei zugänglichen Quellen für CVEs und für Maven-Artefakte aktiviert. Für weitere, kommerzielle Quellen benötigt man eigene API-Keys.

Danach werden die Container durchgestartet. Im Hintergrund sollte nach dem Neustart zeitnah die Replikation der Vulnerabilities-Datenbanken anlaufen. Das ist in den Logs des Backends zu sehen.

Upload meiner BOM mit curl ...

Der Upload der BOM kann grundsätzlich mittels eines einfachen *curl*-Aufrufs über das Dependency-Track API realisiert werden. Wie im ersten Teil der Serie demonstriere ich das am Beispiel einer alten Version (Git Branch 1.5.x) der spring-petclinic (Listing 1). In diesem Listing muss das API-Token des Projektteams (siehe Installationsanleitung) ergänzt werden.

Listing 1: Upload-Token via curl

```
curl -v -X "POST" "localhost:8081/api/v1/bom" \
-H 'Content-Type: multipart/form-data' \
-H "X-Api-Key: <Token>" \
-F "autoCreate=true" \
-F "projectName=spring-petclinic" \
-F "projectVersion=1.5.1"\
-F "bom=@target/bom.json"
```

Beim Aufruf von *curl* muss darauf geachtet werden, dass die eigentliche Verarbeitung der SBOM in Dependency-Track asynchron geschieht. Der erfolgreiche Upload mit der Bestätigung über *HTTP 200* bestätigt also nur die Annahme und ist kein Signal für den erfolgreichen Abschluss der Analyse.

... oder auch mit Maven

Das Maven-Plug-in dependency-track-maven-plugin [5] bietet mehr Möglichkeiten. In der *pom.xml* muss hierfür die Liste der Plug-ins entsprechend ergänzt werden. Eine Beispielkonfiguration finden Sie in Listing 2. Auch hier muss der API Key aus der Installation ergänzt werden. Vorsicht: Diesen im automatischen Build bitte als *Secret* behandeln und nicht als Teil der *pom.xml* einfach einchecken und im Git-Repository damit für immer sichtbar machen.

Der Upload der SBOM kann dann mit dem folgenden Aufruf durchgeführt werden:

```
mvn dependency-track:upload-bom
```

Den Build abbrechen lassen

Das Maven-Plug-in wartet dabei auf die erfolgreiche Verarbeitung durch Dependency-Track. Daher hat man die Möglichkeit, direkt danach ein weiteres Goal aufzurufen und damit den Build auf Basis der Konfiguration abzubrechen, wenn Schwellenwerte überschritten werden:

```
mvn dependency-track:findings
```

Dieser Aufruf erzeugt zusätzlich zu den umfangreichen Ausgaben XML- und HTML-Reports im target-Verzeichnis. In unserem Fall findet Dependency-Track in den diversen Datenbanken insgesamt 205 Vulnerabilities, davon 47 critical und 88 high (der Wert kann bei euch abweichen). Aufgrund der in der Plug-in-Konfiguration definierten Grenzwerte (*findingThresholds*) bricht Maven daher auch mit einem Fehler ab. Über das weitere Goal *dependency-track:metrics* werden eine Reihe von Statistiken angezeigt. Auch hier kann bei Überschreitung von Grenzwerten (entsprechende *metricsThresholds*) abgebrochen werden.

Schließlich kann über das Goal *dependency-track:score* auf Überschreitung des summarischen *RiskScore*-Limits (*inheritedRiskScoreThreshold* in der Plug-in-Konfiguration) geprüft und so auch über diese Metrik der Build abgebrochen werden (in unserem Fall ist dieser berechnete Wert *1120*). In den beiden letzten Fällen werden die nötigen Werte allerdings trotz Wartens beim Upload asynchron berechnet, hier muss man also manuell warten oder regelmäßig prüfen.

Das UI

Schauen wir nun in das UI von Dependency-Track (im Fall der Nutzung von docker-compose ist das *http://lo-*

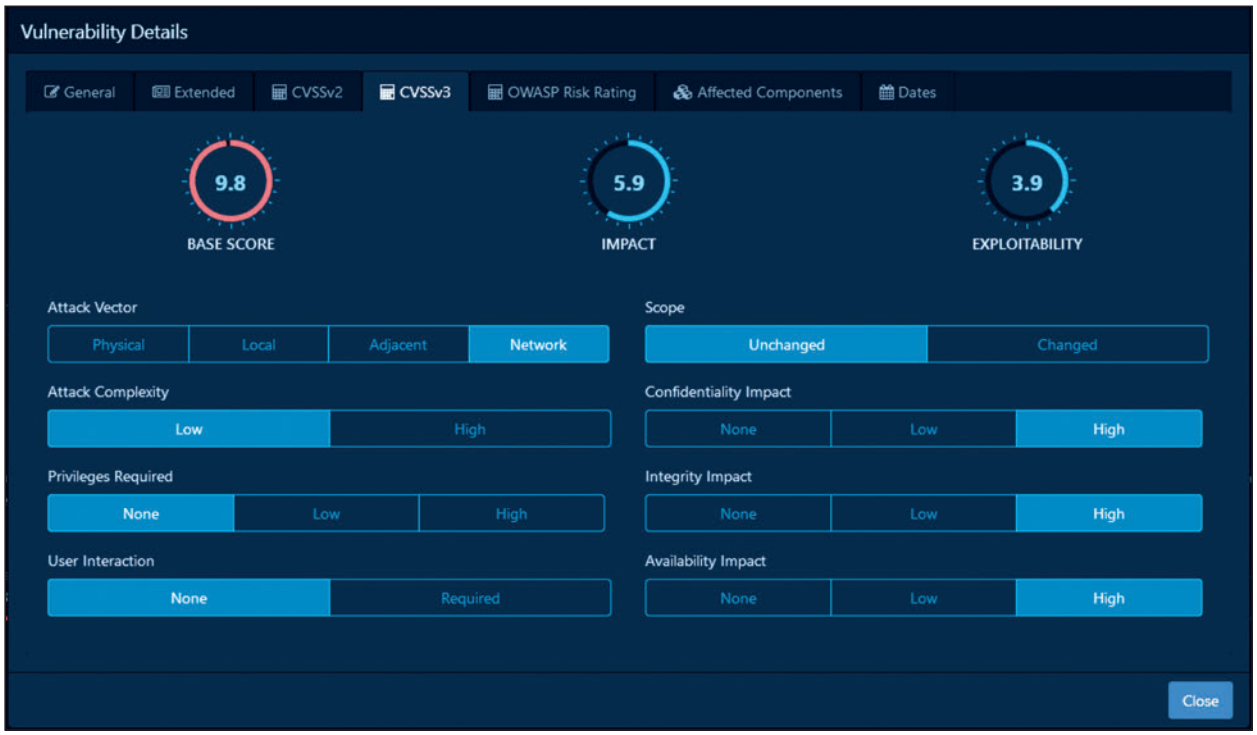


Abb. 2: Details zur Bewertung von CVE-2022-22965; hier ist für einen einfachen Exploit nur Netzwerkzugriff nötig – mit hohen möglichen Schäden

calhost:8080), finden wir unter PROJECTS die relevanten Informationen zu allen Projekten und ihren verschiedenen Versionen – in unserem Fall wäre das aktuell spring-petclinic 1.5.1.

Betrachtet man sich dieses Projekt im Detail, sieht man zunächst einige Charts mit entsprechenden Trendanzeigen. Darüber hinaus kann man sich u. a. eine Liste der Abhängigkeiten, den Abhängigkeitsgraphen und alle auffindbaren Vulnerabilities anzeigen lassen. Bei den Vulnerabilities kann tief in die Details navigiert werden – bis hin zur Anzeige des CVEs mit allen Attributen und einer genaueren Beschreibung des Ratings (**Abb. 2**).

Unter dem Reiter EXPLOIT PREDICTIONS gibt Dependency-Track die Exploit-Wahrscheinlichkeit auf Basis eines sogenannten EPSS-Werts [6] an. Bei einigen wenigen CVEs liegt die Wahrscheinlichkeit laut Übersicht (**Abb. 3**) nahe *1.0*. Ein alarmierendes Ergebnis! Denn *1.0* besagt eine Exploit-Wahrscheinlichkeit von 100 Prozent in den nächsten 30 Tagen. Daher sollte ein Wert, der nahe *1* liegt, zu sofortigen Maßnahmen führen.

Über die Reiter COMPONENTS und VULNERABILITIES können Suchen bzw. Drill-Downs über die identifizierten Komponenten oder die bekannten Vulnerabilities über alle Projekte und deren Versionen gestartet werden. Über den Reiter LICENSES werden zentral Lizenzen gepflegt und kategorisiert. Über den nächsten Reiter POLICY MANAGEMENT können unter anderem Regeln zur Verwendung von Lizenzen definiert werden. Werden diese Regeln verletzt, erkennt das Plug-in sogenannte Policy Violations, was den Build brechen kann. So kann ich z. B. eine Policy definieren, die einen Fehler meldet, wenn eine Abhängigkeit eine Apache-2.0-Lizenz hat. Nach einem erneuten Upload der SBOM (und einer dadurch angestoßenen erneuten Analyse) sind diese Verletzungen im entsprechenden Reiter zu sehen (**Abb. 4**).

Regelmäßig und automatisch

Das eigentlich Spannende an Dependency-Track ist, dass es im Hintergrund regelmäßig die aktiven Projekte auf neue Vulnerabilities scannt. Das muss man also nicht mehr manuell in den Build Pipelines der verschiedenen Projekte umsetzen. Für eine Signalisierung bei Änderungen in der Bewertung könnte der integrierte Alerting-Mechanismus genutzt werden. Alternativ könnte man

Listing 2: Ergänzung Maven POM

```
<plugin>
  <groupId>io.github.pmckeown</groupId>
  <artifactId>dependency-track-maven-plugin</artifactId>
  <version>1.7.0</version>
  <inherited>false</inherited>
  <configuration>
    <dependencyTrackBaseUrl>http://localhost:8081</
                                        dependencyTrackBaseUrl>
      <apiKey>...API-Key...</apiKey>
      <bomLocation>target/bom.json</bomLocation>
      <updateProjectInfo>true</updateProjectInfo>
      <findingThresholds>
        <high>0</high>
        <critical>0</critical>
      </findingThresholds>
      <inheritedRiskScoreThreshold>5</inheritedRiskScoreThreshold>
      <failOnError>true</failOnError>
  </configuration>
</plugin>
```

Abb. 3: Ein Ausschnitt der Exploit-Predictions-Ansicht für petclinic auf der Testumgebung des Autors

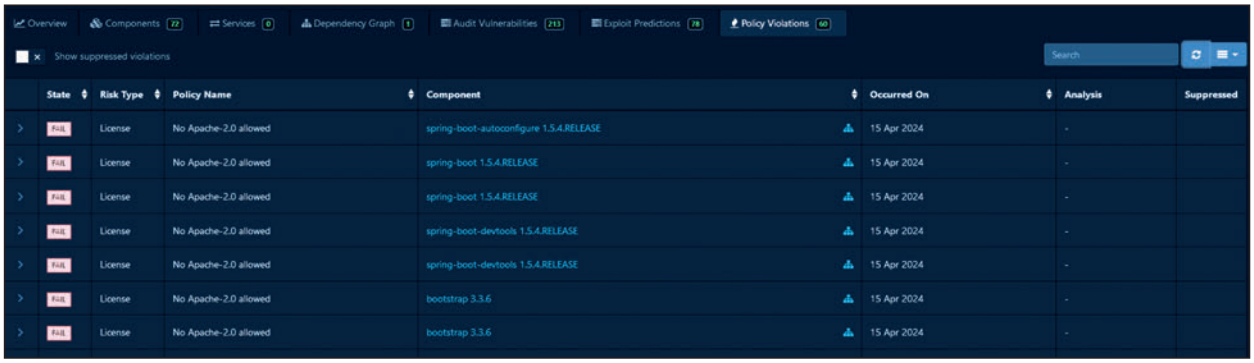


Abb. 4: Ein Ausschnitt der Ansicht Policy Violations für petclinic auf der Testumgebung des Autors

auch direkt das REST API nutzen, um beispielsweise Projekte regelmäßig auf die Überschreitung von Schwellenwerten zu prüfen.

Ein weiteres wichtiges Feature sind die einfachen Workflowfunktionalitäten in den Detailansichten zu den Audit Vulnerabilities und Policy Violations eines Projekts. Sie ermöglichen es, Findings zu analysieren, zu bewerten, zu kategorisieren und sie am Ende zu schließen und somit sukzessive abzuarbeiten.

Integrativ

Dependency-Track bietet neben den genannten Funktionalitäten zahlreiche Integrationsmöglichkeiten. So können z. B. E-Mails verschickt oder Nachrichten an Mattermost, Slack oder Teams gesendet werden. Zudem können weitere, ggf. kostenpflichtige Analyzer für Vulnerability- und License-Scans herangezogen werden. Und natürlich ist Java nicht die einzige Programmiersprache, die unterstützt wird.

Zusammenfassung

Damit endet unser kleiner Streifzug durch die Welt der SBOMs. Ich hoffe, ich konnte zeigen, wie nützlich sie sind und wie einfach die zugehörigen Werkzeuge genutzt werden können. Mein Tipp: Erzeugen Sie „einfach mal“ eine SBOM und prüfen Sie, inwieweit Ihre Software wirklich sicher ist. In dem Fall wünsche ich Ihnen viel Spaß beim Erstellen des ein oder anderen Tickets – für die Anpassung des Builds oder das Aktualisieren der ein oder anderen Abhängigkeit!

Der Artikel bezieht sich auf die Version 4.10.1, seitdem ist die 4.11.0 releast worden. Die aktuelle Version ist 4.11.3 (es gab daraufhin mehrere Bugfixes in den Versionen 4.11.1-4.11.3).

Tim Teulings ist als Senior-Solution-Architekt bei Opitz Consulting tätig. Er unterstützt Softwareentwicklungsteams dabei, schnell, einfach und entspannt Software zu bauen, die perfekt zum Kunden passt. Die Themenbereiche Modernisierung und Integration gehören in dieser Funktion zu seinem täglichen Geschäft. Zu Tims Schwerpunkten zählen entsprechende Tools, Frameworks, Methoden, Vorgehen, Architekturen und Techniken.

Links & Literatur

[1] https://dependencytrack.org

[2] https://owasp.org – ein bekanntes weltweites Security-Projekt, das kostenlose Standards, Methodiken, Ressourcen und Werkzeuge bereitstellt

[3] https://docs.docker.com/compose

[4] https://docs.dependencytrack.org/getting-started/deploy-docker

[5] https://github.com/pmckeown/dependency-track-maven-plugin

[6] https://www.first.org/epss/articles/prob_percentile_bins