Java magazin™

Java | Architektur | Software-Innovation

Wechsel zu Kubernetes

Der Berg ruft!

Ausgabe 7.2024

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

4 194586 709801 07

entwickler.de

Die Software Bill of Material in Java – Teil 1

Die Software-lieferkette absichern

Die Software Bill of Material (SBOM) gibt es schon lange, doch aktuell liest man wieder mehr über sie. Erfahre, warum die SBOM aktuell wichtiger wird, wie du als Developer:in, Hersteller:in oder Nutzer:in von Software davon profitierst und wie du sie im Kontext von Java ganz einfach erzeugen kannst.

von Tim Teulings

Die Software Bill of Material gibt es aktuell in zwei relevanten Ausprägungen. Das SPDX-Format [1] – initiiert von der Linux Foundation – gibt es seit 2011 und ist sogar ein ISO/IEC-Standard [2]; das OWASP-CycloneDX-Format [3] gibt es seit 2018.

SBOM – was ist das?

Beide Formate werden aktiv verwendet und immer wieder angepasst. Sie dienen dazu, ein Softwareprodukt einer bestimmten Version sowie dessen verwendete Komponenten oder Bibliotheken und wiederum deren eigene Abhängigkeiten, aber z. B. auch angebotene Services im Sinne einer Softwarelieferkette [4] eindeutig zu identifizieren und zu dokumentieren. Ähnlich einem Lieferschein oder der Angabe der Inhaltsstoffe von Lebensmitteln stellt die SBOM verbindliche und detaillierte Informationen über die „Inhalte“ einer Software bereit.

In jüngster Zeit gewinnt die SBOM aufgrund der weltweit steigenden Zahl von Sicherheitsvorfällen zunehmend an Relevanz. Nicht gepatchte Systeme werden Opfer eines Angriffs und müssen heruntergefahren werden. Angreifer brechen über Sicherheitslücken in Systeme ein, verschlüsseln Daten und versuchen so, größere Geldbeträge zu erpressen. Und vermehrt sind Infrastrukturen betroffen, deren Nichtverfügbarkeit direkten Einfluss auf das Leben der Bürgerinnen und

Artikelserie

Teil 1: Die Softwarelieferkette absichern
Teil 2: Dependency-Track vorgestellt

Bürger hat. Gefühlt ist aktuell in Deutschland immer mindestens eine kommunale Stadtverwaltung „down“ oder mit eingeschränktem Dienst im „Wiederaufbau“ der Infrastruktur.

In den Vereinigten Staaten ist die Lieferung von SBOMs für Dienstleister von Regierungsbehörden daher mittlerweile verpflichtend [5]. Auch in der Europäischen Union könnten SBOMs mit dem Cyber Resilience Act in bestimmten Kontexten verpflichtend werden. Das BSI hat mit dem TR-03183-2 [6] bereits erste Vorgaben dazu definiert.

Was habe ich davon?

Der Mehrwert der SBOM ist einfach zu beschreiben. Sie zu nutzen ist aus meiner Sicht auch unabhängig von einem möglichen gesetzlichen Rahmen sinnvoll, denn sie ermöglicht die effiziente Zuordnung bekannter Vulnerabilities – also gemeldeter und ggf. sogar in einer Folgeversion bereits behobener Sicherheitslücken – zu einem Softwareartefakt und ist damit ein wertvoller und einfach zu realisierender Baustein jedes aktiven Sicherheitskonzepts.

Mittels SBOM und deren regelmäßiger Prüfung bist du anderen einen Schritt voraus. Wenn das nächste Sicherheitsproblem in einer häufig genutzten Java-Bibliothek durch die Presse geht oder gar die ersten Systeme gehackt wurden, kannst du dich mit hoher Wahrscheinlichkeit bequem zurücklehnen, weil du dein Lieblingsframework schon auf die nächste gefixte Version aktualisiert, deine Kunden informiert und/oder präventive Sicherheitsmaßnahmen getroffen hast.

SBOM erstellen

Speziell im Java-Umfeld ist die Erstellung einer SBOM heutzutage sehr einfach automatisiert möglich und in den Continuous Build einer Applikation integrierbar. Im Folgenden zeigen wir das konkrete Vorgehen mittels eines Maven-Plug-ins am Beispiel einer alten Version (konkret am Stand mit dem Tag *1.5.x*) der Softwarequellen der Spring-Variante von Java Petclinic (spring-petclinic [7]), die wir mittels des folgenden Git-Befehls auschecken:

```
git clone --branch 1.5.x https://github.com/spring-projects/spring-petclinic.git
```

Im nächsten Schritt ergänzen wir die Maven *pom.xml* im Hauptverzeichnis um die Konfiguration des CycloneDX-Plug-ins für Maven [8], indem wir die *pom.xml* hinter der Zeile 208 um Listing 1 ergänzen und damit der Liste der genutzten Plug-ins ein weiteres hinzufügen.

Listing 1: Ergänzung Maven pom.xml

```
<plugin>
  <groupId>org.cyclonedx</groupId>
  <artifactId>cyclonedx-maven-plugin</artifactId>
  <version>2.7.11</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>makeAggregateBom</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <projectType>application</projectType>
    <outputFormat>json</outputFormat>
  </configuration>
</plugin>
```

Listing 2: Auszüge target/bom.json

```
...
"publisher" : "Pivotal Software, Inc.",
"group" : "org.springframework.boot",
"name" : "spring-boot-starter-actuator",
"version" : "1.5.4.RELEASE",
"description" : "Starter for using Spring Boot's Actuator which provides
        production ready features to help you monitor and manage your
                                                        application",
"scope" : "required",
"hashes" : [
 {
  "alg" : "MD5",
  "content" : "11114e627cc9966ff74bdaab464ddd49"
 },
 {
  "alg" : "SHA-1",
  "content" : "10f526d22f58a32a8c3af26ccb0a6f16f4649935"
 },
...
"licenses" : [
 {
  "license" : {
"id" : "Apache-2.0"
  }
 }
],
"purl" : "pkg:maven/org.springframework.boot/spring-boot-starter
                                        -actuator@1.5.4.RELEASE?type=jar",
"externalReferences" : [
 {
  "type" : "website",
  "url" : "http://projects.spring.io/spring-boot/"
 },
 {
  "type" : "build-system",
  "url" : "https://build.spring.io/browse/BOOT-PUB"
 },
...
```

Mit dieser simplen Erweiterung kannst du nun bereits eine SBOM im CycloneDX-Format erzeugen. Das geschieht mit folgendem Maven-Aufruf (die eventuell auftretende Warnung des Plug-ins während der Ausführung kann man getrost ignorieren):

```
mvn cyclonedx:makeAggregateBom
```

Für die Generierung muss dabei nicht mal ein Build der Software ausgeführt werden, da alle notwendigen Informationen direkt über die Maven-Projektkonfiguration und die dort hinterlegten Abhängigkeiten erlangt werden können. Es genügt der Download der Abhängigkeiten über Maven im Rahmen des Aufrufs.

Die resultierende SBOM findet man als JSON-Datei nun unter *target/bom.json* (siehe Listing 2 mit Auszügen dieser Datei). Sie beinhaltet unter anderem eine Beschreibung des CycloneDX-Maven-Plug-ins als Generator der SBOM, eine Beschreibung der spring-petclinic selbst, aber auch eine Liste aller direkten und transitiven Abhängigkeiten – jeweils unter Nennung der entsprechenden Maven-Koordinaten, diverser Hashes und der jeweils gültigen Lizenzen sowie weiterer Informationen.

Der Vorteil von Maven als Informationsbasis (oder auch Gradle – denn auch hierfür gibt es ein entsprechendes CycloneDX-Plug-in, das ähnlich einfach genutzt werden kann; Kasten: „Nutzung von Gradle statt Maven“) ist die Wiederverwendung entsprechender Build-Informationen. Das sorgt für ein hohes Maß an Korrektheit beim Inhalt der resultierenden SBOM. Lägen diese Informationen nicht vor, müsste (mit Hilfe anderer Werkzeuge wie z. B. Syft [10] oder cdxgen [11]) eine SBOM mit mehr Aufwand und höherem Interpretationsspielraum erzeugt werden – dann etwa auf Basis eines Verzeichnisinhalts, des Inhalts eines Docker-Containers und/oder entsprechender Build-Ergebnisse wie *.jar*- oder *.war*-Dateien.

Vulnerabilities-Liste erstellen

Um die Liste der bekannten Vulnerabilities zu erstellen, nutzen wir das aktuelle Release von Grype [12]. Auf der Releaseseite [13] wird das Go-Programm für verschiedenste Systeme und Installationsvarianten angeboten. Außer der eigentlichen Installation sind im Vorfeld keine weiteren Aktivitäten nötig.

Über folgenden Aufruf, der zu Beginn ggf. erst einmal die Remotedatenbank herunterlädt, lokal cacht und dann die SBOM analysiert, kommen wir zu einer tabellarischen Liste der gefundenen Probleme. Für den eigentlich Look-up benötig Grype dabei entsprechende Common Platform Enumerations (CPE) [14] als standardisierte IDs für das zu prüfende Artefakt. Da das CycloneDX-Maven-Plug-in diese nicht erzeugt, weisen wir Grype an, das für uns zu tun (Listing 3).

Nutzung von Gradle statt Maven

Nutzt man Gradle statt Maven, bietet sich die Nutzung des entsprechenden CycloneDX-Gradle Plug-ins [9] an. Man integriert es mit einer einfachen Ergänzung in die *build.gradle*:

```
plugins {
  id 'org.cyclonedx.bom' version '1.8.2'
}
```

Die SBOM selbst kann dann durch den folgenden Aufruf erzeugt werden:

```
gradle cyclonedxBom
```

Die resultierenden SBOMs findet man in Form der Dateien *build/reports/bom.xml* und *build/reports/bom.json*. Eine weitere Konfiguration ist auch hier möglich. Siehe hierzu die Dokumentation auf der verlinkten GitHub-Seite [9].

Doktor, wie schlimm ist es?

Zum Zeitpunkt der Erstellung des Artikels kamen wir in unserem Beispiel auf 35 „kritische“ Vulnerabilities, 62 der Stufe „hoch“ und 33 der Stufe „mittel“. Also in Summe 130(!) Vulnerabilities. Für 127 davon gab es bereits einen Fix in einer aktuelleren Version der entsprechenden Abhängigkeit. Da Grype für die Identifikation von Vulnerabilities diverse Datenbanken nutzt, findet man in der Ausgabe auch entsprechende IDs wie *GHSA-x2w5-5m2g-7h5m* für die GitHub Advisory Database. In der resultierenden Liste finden wir (nicht überraschend) Java-Abhängigkeiten wie Spring, Tomcat und jackson-databind. Details zu obiger Vulnerability – eine Sicherheitslücke mit einer Kritikalität von 9,8 von 10 – findest du unter [15]. Dort siehst du auch, dass sie der CVE-2018-14720 entspricht.

Über die zusätzliche Option *-o cyclonedx-json* (und die Option *--file <Datei>*) kann ich eine neue, um die CVEs angereicherte SBOM erzeugen, um diese z. B. automatisiert weiter auszuwerten. Diese Datei beinhaltet dann auch direkt die zugehörigen CVE-Nummern zu den Vulnerabilities.

Abbruch!

Über die weitere Option *--fail-on <severity>* (z. B. *--fail-on critical*) kann ich Grype anweisen, bei Vulnerabilities mit dieser oder einer höheren Kritikalität einen Fehler zurückzugeben und damit den Build abzubrechen. Sollte es zu False Positives kommen oder möchte man einzelne Vulnerabilities aus anderen Gründen unterdrücken, ist auch das über die Grype-Konfigurationsdatei möglich [16].

Schnelle automatische Prüfung

Führe ich die Prüfung der SBOM auf Vulnerabilities regelmäßig aus, werde ich also automatisch und frühzeitig über neue Sicherheitslücken informiert. Dabei benötige ich für die weiteren Scans nicht mehr den Quelltext der zu prüfenden Software. Die SBOM reicht hierfür. Und selbst bei Pressemitteilungen über Securityvorfälle kann ich schnell über eine Analyse der SBOM feststellen, ob ich betroffen bin.

Aber was ist, wenn ich viele Softwareprodukte in jeweils mehreren Version supporten muss? Oder wenn ich diverse Softwareprodukte einsetze, für die ich nur eine SBOM erhalte?

In diesem Fall möchte ich Prüfung und Management ggf. zentralisieren. Hierfür stelle ich im zweiten Teil der Artikelserie das offene und kostenlose Werkzeug Dependency-Track [16] der OWASP vor, mit dem man den Umgang mit der SBOM weiter professionalisieren kann.

Listing 3: Aufruf Grype und Teile der Ausgabe

```
grype --add-cpes-if-none sbom:target/bom.json

 ✓ Vulnerability DB                [updated]
 ✓ Scanned for vulnerabilities     [130 vulnerability matches]
   ├── by severity: 35 critical, 62 high, 33 medium, 0 low, 0 negligible
   └── by status:   127 fixed, 3 not-fixed, 0 ignored
NAME          INSTALLED  FIXED-IN  TYPE         VULNERABILITY       SEVERITY
commons-compress  1.9      1.21     java-archive  GHSA-xqfj-vm6h-2x34  High
commons-compress  1.9      1.21     java-archive  GHSA-mc84-pj99-q6hh  High
commons-compress  1.9      1.21     java-archive  GHSA-crv7-7245-f45f  High
commons-compress  1.9      1.21     java-archive  GHSA-7hfm-57qf-j43q  High
commons-compress  1.9      1.18     java-archive  GHSA-hrmr-f5m6-m9pq  Medium
dom4j             1.6.1             java-archive  GHSA-hwj3-m3p6-hj38  Critical
dom4j             1.6.1             java-archive  GHSA-6pcc-3rfx-4gpm  High
hibernate-core 5.0.12.Final 5.3.20.Final java-archive GHSA-j8jw-g6fq-mp7h High
hibernate-validator 5.3.5.Final 5.3.6.Final java-archive GHSA-xxgp-pcfc-3vgc High
hsqldb            2.3.5    2.7.1    java-archive  GHSA-77xx-rxvh-q682  Critical
...
```

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://spdx.dev

[2] ISO/IEC 5962:2021: https://spdx.dev/use/specifications/

[3] https://cyclonedx.org

[4] https://de.wikipedia.org/wiki/Software-Lieferkette

[5] https://blog.sonatype.com/cyber-readiness-and-changing-federal-government-sbom-requirements

[6] https://www.bsi.bund.de/DE/Service-Navi/Presse/Alle-Meldungen-News/Meldungen/TR-03183-2-SBOM-Anforderungen.html

[7] https://github.com/spring-projects/spring-petclinic

[8] https://github.com/CycloneDX/cyclonedx-maven-plugin

[9] https://github.com/CycloneDX/cyclonedx-gradle-plugin

[10] https://github.com/anchore/syft

[11] https://github.com/CycloneDX/cdxgen

[12] https://github.com/anchore/grype

[13] https://github.com/anchore/grype/releases

[14] https://de.wikipedia.org/wiki/Common_Platform_Enumeration

[15] https://github.com/advisories/GHSA-x2w5-5m2g-7h5m

[16] https://github.com/anchore/grype?tab=readme-ov-file#specifying-matches-to-ignore

[17] https://dependencytrack.org