Die Lehren aus Log4j

Wie kontinuierliche Software Composition Analysis (SCA) hilft, Risiken in der Software Supply Chain zu erkennen und zu beheben.

Von Stephan Kaps

Es ist Anfang Dezember 2021. Eine Sicherheitslücke wird veröffentlicht. Sie trägt den Namen CVE-2021-44228. Das Bundesamt für Sicherheit in der Informationstechnik (BSI) gibt eine Cybersicherheitswarnung der Stufe Rot heraus. Risiko kritisch. Ein CVSSv3 Score von 10 von 10 Punkten. Der Spiegel schreibt „Wie löscht man ein brennendes Internet?“ und selbst in der Tagesschau vom 13.12.2021 findet Log4Shell Erwähnung.

Was war passiert?

Die Sicherheitslücke in der Open-Source-Java-Bibliothek Log4j war deshalb so dramatisch, weil diese in sehr vielen Java-Anwendungen eingesetzt wird – und niemand einen Überblick hatte, wo überall. Entsprechend hektisch forschten Softwarehersteller und IT-Abteilungen weltweit, ob ihre Anwendungen betroffen waren.

Es war nicht das erste Mal, dass eine Schwachstelle in einer Open-Source-Bibliothek für Überstunden in IT-Abteilungen gesorgt hat. Doch was haben wir daraus gelernt? Welche Erkenntnisse wurden gewonnen und welche Maßnahmen ergriffen, um bei der nächsten Zero-Day Vulnerability effizienter agieren zu können? Ist es möglich, schneller herauszufinden, in welchen Produkten eine Bibliothek in einer bestimmten Version verwendet wird?

Was technisch genau die Ursache ist und wie sich die Log4j-Sicherheitslücke ausnutzen lässt, liegt nicht im Fokus dieses Artikels. Es geht vielmehr darum, aufzuzeigen, wie sich eine kontinuierliche Software Composition Analysis (SCA) einrichten lässt, um Risiken in der Software Supply Chain frühzeitig zu identifizieren.

In den vergangenen Monaten traten vermehrt Fälle von ernsthaften Verwundbarkeiten in der Software Supply Chain zu Tage, häufig in Open-Source-Bibliotheken, die wie Log4j in vielen Softwareprodukten weltweit zum Einsatz kommen. Der Fall Log4j gab dem Autor letztendlich den Anstoß, sich Maßnahmen zu überlegen und umzusetzen, die dabei helfen, die Risiken in der Software Supply Chain schneller und effizienter zu identifizieren.

Was ist zu tun?

Zunächst ist es wichtig herauszufinden, welche Produkte betroffen sind, und zu recherchieren, ob bereits Patches bereitstehen, um die Lücke zu schließen. Das Log4j-Entwicklerteam hat zügig eine neue Version veröffentlicht. Allerdings folgten weitere Nachbesserungen, sodass IT-Abteilungen mehrfach patchen mussten.

Bei selbst entwickelten Anwendungen lässt sich durch die Analyse der Maven-POM-Dateien herausfinden, wo Log4j verwendet wird. Bei zugekaufter Software gestaltet sich das schwierig, da der Sourcecode meist nicht vorliegt. In solchen Fällen ist es notwendig, die Hersteller zu kontaktieren, um festzustellen, ob das Produkt gefährdet ist – und wann mit einem Patch zu rechnen ist. Einige Hersteller haben aktiv auf ihren Websites informiert, ob ihr Produkt betroffen ist. Meist jedoch nur, um Entwarnung zu geben.

In vielen Anwenderunternehmen ist jedoch unklar, wer für mutmaßlich betroffene Produkte verantwortlich ist und sich um Informationen und Patches bemühen sollte. Der IT-Betrieb installiert die Software in der Regel nur, und den Fachbereichen, die mit den Anwendungen arbeiten, fehlt die notwendige

iX-TRACT

- Sicherheitslücken in Open-Source-Bibliotheken bedrohen komplette Software Supply Chains, wie das Beispiel Log4j gezeigt hat.
- Um die Risiken frühzeitig erkennen und geeignete Maßnahmen einleiten zu können, sollten Unternehmen auf die kontinuierliche Software Composition Analysis (SCA) vertrauen.
- Software Bills of Materials (SBOMs) verschaffen eine Übersicht der Anwendungslandschaft inklusive der darin verwendeten Bibliotheken und Komponenten sowie deren Abhängigkeiten.

```
<component type="library" bom-ref="pkg:maven/org.apache.poi/poi@3.17?type=jar">
  <publisher>Apache Software Foundation</publisher>
  <group>org.apache.poi</group>
  <name>poi</name>
  <version>3.17</version>
  <description>Apache POI - Java API To Access Microsoft Format Files</description>
  <scope>optional</scope>
  <hashes>
    <hash alg="MD5">243bc3d431e4fadb79738719504c64f7</hash>
    <hash alg="SHA-1">0ae92292a2043888b40d418da97dc0b669fde326</hash>
    <hash alg="SHA-256">30181821dd2e849727b638b9e329aeff4a64f3445c4142b13cf7a18bb3552edd</hash>
    <hash alg="SHA-384">f9a4db2b945cdba835b3c076c3f2bfa72767a5c02f81a57ffb4a5782651d5da66453c03e5c1be0505d1edad8a35c6f82</hash>
    <hash alg="SHA-512">a0c018d8999356000b1077b343aac1a2e9050f84e3097e28b77869be9fc23475d4ac59bf375a2c96b6d824f06ba2d1873ee8d4495fb9bb412f848379ac7d11fb</hash>
    <hash alg="SHA3-256">84ba8a8001b44c04048d5ab1200640f0f5d541832319 62cb180de7d7cfa7a386</hash>
    <hash alg="SHA3-384">2101589e294edd8d55f29be193e74c84afb8db754566b08b369c6c235e969af25ee8962e70f7c3dc262f7cbdebe3d57d</hash>
    <hash alg="SHA3-512">cc8b929d7f54aa98c606977bab56fcfe198e0bb21403b7b2a1bff6bf2940901aee9d47bde493a98af376b4b1e75d110284023bd174b7774d454fac6e14b0606c</hash>
  </hashes>
  <licenses>
    <license>
      <id>Apache-2.0</id>
    </license>
  </licenses>
  <purl>pkg:maven/org.apache.poi/poi@3.17?type=jar</purl>
  <externalReferences>
    <reference type="website"><url>http://www.apache.org/</url></reference>
    <reference type="mailing-list"><url>http://mail-archives.apache.org/mod_mbox/poi-user/</url></reference>
  </externalReferences>
</component>
```

Ausschnitt einer SBOM im XML-Format zum Auflisten verwendeter Bibliotheken und Komponenten (Abb. 1).

Kompetenz. Es kann daher sinnvoll sein, Product Owner für Kaufsoftware zu etablieren.

Im Falle von Log4Shell standen rasch Tools zur Verfügung, um die Infrastruktur zu härten – beispielsweise durch ein Middleware-Plug-in für den Reverse Proxy und Loadbalancer Traefik. Betroffene erhielten darüber hinaus konkrete Hilfestellung über sichere Konfigurationsmaßnahmen, die ein Ausnutzen der Sicherheitslücke verhindern sollten, wie etwa das Flag `log4j.formatMsgNoLookup=true`. Nutzer einer Web-Application-Firewall durften zumindest darauf hoffen, die Angriffe erkennen und blockieren zu können. Blieben die bisher genannten Maßnahmen wirkungslos, hieß der letzte Ausweg: Ausschalten!

Lehren aus Log4j und was zu tun ist

Die Herausforderung für viele Unternehmen besteht darin, einen Weg zu finden, wie sie im Fall einer neu bekannt gewordenen Schwachstelle rasch klarstellen können, ob die in ihrer IT-Landschaft eingesetzte Software betroffen ist. Um dieser Herausforderung zu begegnen, müssen sich folgende Fragen zügig beantworten lassen:

1. Welche Anwendung verwendet eine von Schwachstellen betroffene Komponente?
2. Und falls ja, in welcher Version?
3. Gibt es einen Überblick über die gesamte Anwendungslandschaft?

Wertvolle Hilfe zum Beantworten dieser Fragen leisten Software Bills of Materials (SBOMs). Dabei handelt es sich um die einer Zutatenliste vergleichbare Auflistung aller in einer Software verwendeten Bibliotheken und Komponenten. SBOMs haben ein standardisiertes Format (SPDX oder CycloneDX), sind maschinenlesbar und beinhalten alle direkten und transitiven Abhängigkeiten sowie deren hierarchische Beziehung. SBOMs können darüber hinaus noch weitere Informationen etwa zu Lizenzen enthalten.

In den USA verpflichtet bereits eine Executive Order aus dem Mai 2021 Hersteller, die Software an die Regierung verkaufen wollen, dazu, ihren Produkten SBOMs beizulegen (siehe ix.de/zf49). In Europa verbreitet sich diese Vorgehensweise erst langsam.

SBOM erzeugen

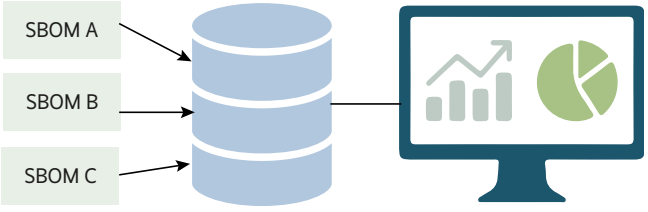
Eine SBOM ist eine JSON- oder XML-Datei in einem definierten Format. Die Website des OWASP-Projekts CycloneDX listet eine Vielzahl von Tools auf, mit denen sich SBOMs für Software in diversen Programmiersprachen erzeugen lassen. Für Java existiert ein Maven-Plug-in, das sich einfach in den Build-Prozess integrieren lässt:

```
mvn org.cyclonedx:cyclonedx- ↵
mven-plugin:makeAggregateBom
```

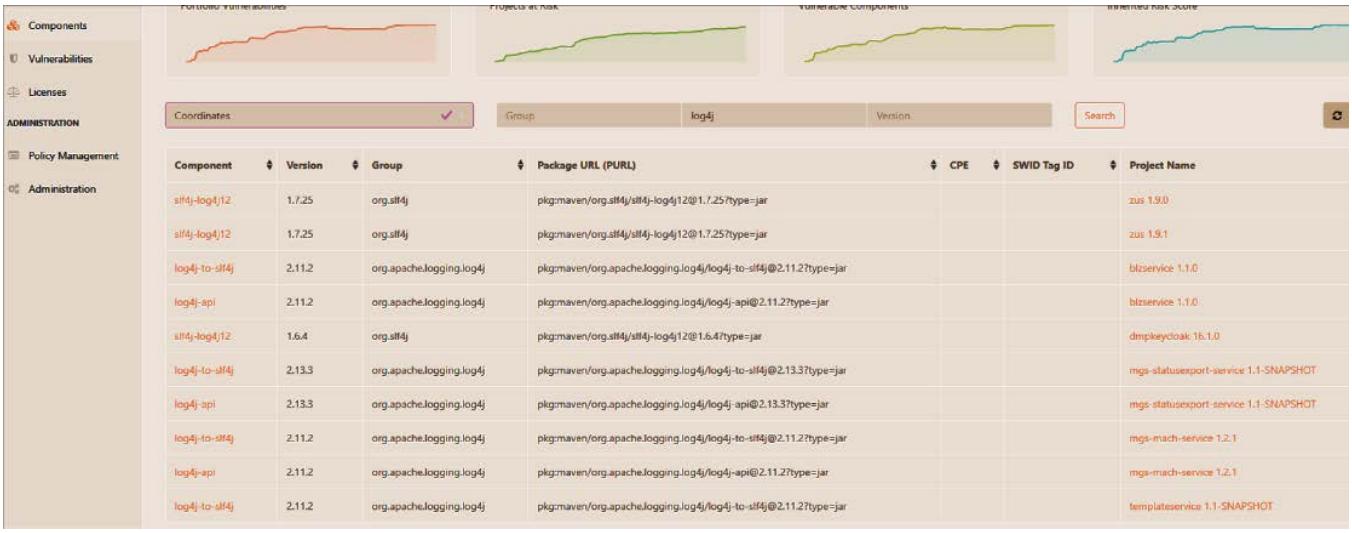
Besteht eine Anwendungslandschaft aus 60 eigenentwickelten Produkten und bestehen diese im Schnitt aus drei Komponenten oder Modulen, dann ergeben sich daraus 180 JSON-Dateien. Sie alle manuell zu analysieren wäre ein zu hoher Aufwand und würde zudem nur eine Momentaufnahme abbilden, da bei jeder Änderung am Code weitere neue Dateien und Abhängigkeiten hinzukommen. Stattdessen empfiehlt es sich, sämtliche SBOMs zentral zusammenzuführen, um sie analysieren zu können.

Dafür existieren Tools wie die Open-Source-Software Dependency-Track von OWASP (siehe ix.de/zf49). Das Werkzeug lässt sich als Docker-Container leicht in Betrieb nehmen und durch LDAP oder OpenID an firmeninterne Verzeichnisdienste anbinden. Jede importierte SBOM erzeugt eine Version innerhalb eines Projektes. Im Falle von Java sind diese Angaben identisch mit den Inhalten der POM-Datei. In Dependency-Track stehen somit alle Informationen zu verwendeten Third-Party-Libraries zur Verfügung und das Tool kann sie gegen diverse verfügbare Kataloge von bekannten Verwundbarkeiten wie etwa NIST prüfen. Es ermittelt zu jedem Treffer einen Risiko-Score und in Summe auch für die gesamte Version des Projektes. Anhand des Scores lässt sich rasch entscheiden, welchem Produkt die größte Aufmerksamkeit gelten sollte.

Ab jetzt sind Zahlen sichtbar, die eine eventuell harte Realität offenlegen. Jetzt ist transparent, was vorher niemand so genau wusste. Das kann bei den Verantwortlichen zu schlaflosen Nächten füh-



Für die Analyse sollten sämtliche SBOMs zentral zusammenfließen (Abb. 2).



Suche nach Komponenten und deren jeweiligen Versionen im Dependency-Track (Abb. 3).

ren, denn die schiere Masse an zu behebenden Schwachstellen ist nahezu unmöglich abzuarbeiten – zumal fast täglich neue hinzukommen. Der psychische Faktor einer solchen Analyse ist daher nicht zu unterschätzen.

Dependency-Track ermöglicht die Suche nach Komponenten. Das ist exakt die Funktion, mit der sich die erste der oben formulierten Herausforderungen meistern lässt. Alle Produkte, die ihre SBOM in diversen Versionen bereitstellen, lassen sich nach den exakten Versionen einer Komponente durchsuchen. Je nach Größe der Anwendungslandschaft und Zahl der zu untersuchenden Verwundbarkeiten erspart das unter Umständen viele Stunden an manuellem Aufwand.

Darüber hinaus ist eine Suche nach konkreten Verwundbarkeiten anhand ihrer CVE-Nummer möglich. So lässt sich eine Übersicht erstellen, in welchen Produkten diese auftreten.

Dependency-Track bietet auch die Möglichkeit, Policies zu konfigurieren (allerdings nur sehr einfache) und Alerts einzurichten, sodass bei deren Nichteinhaltung alarmiert wird – beispielsweise per Webhook an bestimmte Chatsysteme. Mittlerweile gibt es neben Dependency-Track eine große Menge weiterer Tools und Frameworks zur Software Components Analysis, zum Erstellen von SBOMs und für die Automatisierung dieser Vorgänge. Eine Marktübersicht folgt in der nächsten Ausgabe von iX.

Sofern sich im Rahmen von Audits ergibt, dass bestimmte CVEs für einzelne Produkte oder die gesamte Anwendungslandschaft nicht relevant sind, lassen sich diese auf eine Ausschlussliste setzen. Dadurch bleiben sie bei der Risiko-Score-Ermittlung unberücksichtigt und der gesamte Prozess bleibt damit schlanker. Für die Integration in den Build-Prozess stehen eine API sowie diverse Plug-ins bereit, etwa für Jenkins (siehe ix.de/zf49). Durch die Jenkins-Integration ist es möglich, Quality Gates zu definieren, vergleichbar zu statischen Codeanalysen, die wiederum den Build-Prozess abbrechen lassen. Zu empfehlen ist deren Einsatz im Nightly Build. Damit lassen sich auch die Produkte regelmäßig untersuchen, an denen aktuell nicht entwickelt wird.

Was ist mit der Ausführungsschicht?

Leider deckt die Analyse auf Anwendungsebene (Inhalt der POM) in den meisten Fällen nur einen Teil der gesamten Wahrheit auf. Jede Anwendung benötigt weitere Zusatzsoftware, um lauffähig zu sein. Im Fall einer Java-Webanwendung könnte dies etwa ein Wildfly-Application-Server sein, der in einem Container läuft.

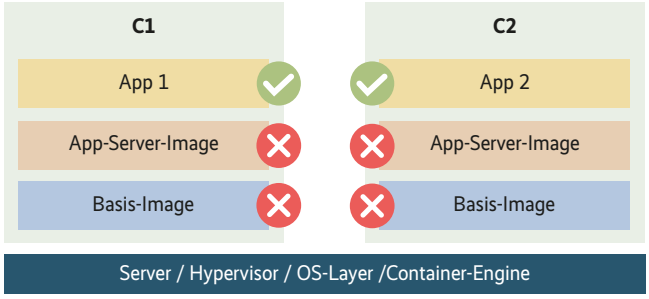
Sowohl der App-Server als auch das Basis-Image des Containers bringen weitere Abhängigkeiten mit, die wiederum bekannte Verwundbarkeiten enthalten können. Um diese zu analysieren, existieren ebenfalls verschiedene Open-Source-Tools wie Syft (siehe ix.de/zf49). Es ist in der Lage, Images oder auch Inhalte eines Filesystems zu analysieren und daraus eine SBOM zu generieren. Syft ist lediglich ein Go-Binary – der Befehl lässt sich per CLI ausführen.

In einem dem Autor bekannten realen Beispiel hatte die Anwendung selbst nur 12 Komponenten, die gesamte Ausführungsschicht aber über 1200. Angesichts dieser Menge ist es schier aussichtslos, prüfen zu wollen, ob eine anfällige Funktion einer Bibliothek wirklich verwendet wird. Hier lautet die Devise: patchen, patchen, patchen. Darüber hinaus sollten Entwicklerinnen und Entwickler ihre Basis-Images so klein wie möglich halten, um die Angriffsfläche insgesamt zu verkleinern. Dabei kann docker-slim helfen oder das Verwenden schlanker Images wie Alpine oder debian-slim.

Wer Docker-Desktop einsetzt, kann seit April 2022 mit dem Befehl `docker`

```
ge('Stat. Analysen') {
 steps {
     withMaven(maven: 'Maven 3.8.6') {
         sh "mvn -U spotbugs:spotbugs org.cyclonedx:cyclonedx-maven-plugin:makeAggregateBom"
     }
     withCredentials([string(credentialsId: 'deptrack-apikey-text', variable: 'API_KEY')]) {
         dependencyTrackPublisher artifact: 'target/bom.xml', projectName: "${env.PROJECT}",
         projectVersion: "${env.OWNVERSION}", synchronous: true, dependencyTrackApiKey: API_KEY
         failedNewCritical: 3, failedNewHigh: 5, failedNewLow: 15, failedNewMedium: 10,
         failedTotalCritical: 5, failedTotalHigh: 10, failedTotalLow: 50, failedTotalMedium: 30
         unstableNewCritical: 1, unstableNewHigh: 2, unstableNewLow: 10, unstableNewMedium: 5,
         unstableTotalCritical: 1, unstableTotalHigh: 5, unstableTotalLow: 30, unstableTotalMed
     }
 }
```

Ausschnitt aus der Jenkins-Pipeline: Verwenden des Dependency-Track-Plug-ins zur Analyse der Maven-POM-Dateien (Abb. 4).



Neben Anwendungen (App1, App2 …) muss die Analyse in jedem Container (C1, C2 …) auch weitere Schichten wie App-Server- und Basis-Images einschließen (Abb. 5).

`sbom` SBOMs seiner lokalen Container-Images erzeugen.

Wie geht es weiter?

Alle Systeme melden nun ihre Zutatenliste (SBOM) an eine zentrale Stelle. Deren Analyse liefert einen Überblick über die gesamte Anwendungslandschaft. Darauf aufbauend lassen sich im Build-Prozess Quality Gates setzen, die im Ernstfall einen Abbruch auslösen. Aus den Alarmierungen ließen sich zudem Issues ableiten und in das zugehörige Trackingsystem einbinden.

Im Sinne von Shift Security Left lassen sich Sicherheitsüberprüfungen weiter vorne im Entwicklungsprozess integrieren, beispielsweise über statische Codeanalysen (SAST) mit FindSecurityBugs oder dynamische (DAST) mit dem Zed Attack Proxy. Werkzeuge wie Dependabot oder Renovate helfen, kleinere Komponentenupdates zu automatisieren, indem sie das Update als Merge- beziehungsweise Pull-Request einstellen, der im Anschluss lediglich zu mergen ist.

Sind neue Sicherheitslücken offiziell bestätigt, kann schnelles Handeln häufig entscheidend sein. Dabei helfen Konzepte wie Continuous Delivery respektive Continuous Deployment, mit denen sich neue Versionen der eigenen Software schneller für den Produktivbetrieb bereitstellen lassen.

Neben dem Einsatz minimalisierter Basis-Images für die Ausführungsschicht bieten sich viele weitere Best Practices an, um sichere oder gehärtete Container zu betreiben – etwa durch Verwenden nicht privilegierter Benutzer, das Begrenzen der für den Container zur Verfügung stehenden Ressourcen oder das Einschränken der Fähigkeiten (Capabilitites) auf die zwingend notwendigen (siehe ix.de/zf49).

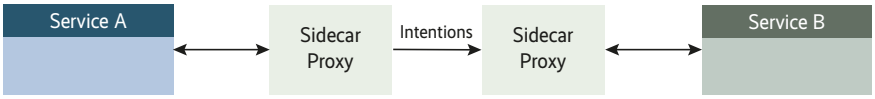
Auf dem Weg zu einem Zero-Trust-Network leisten Service Meshes wertvolle Dienste. In der Regel starten diese zu jedem Service Sidecar-Container, die im Anschluss die gesamte Kommunikation steuern. Über Regeln (Intentions) lässt sich festgelegen, wer mit wem überhaupt sprechen darf. Das trägt zu einer Mikrosegmentierung und damit zu erhöhter Sicherheit bei. Darüber hinaus lässt sich die Kommunikation mit Mutual TLS (Clientzertifikaten) verschlüsseln und absichern. Diese wiederum können und sollten Entwicklerinnen und Entwickler mit Secrets-Management-Tools wie HashiCorp Vault automatisch erzeugen und in kurzen Abständen rotieren.

Abbildung 6 stellt die Funktionsweise des Service Mesh vereinfacht dar. Für die Services A und B wird ein Proxy vorgeschaltet, der sämtliche Kommunikation übernimmt. Intentions legen fest, dass Service A mit Service B kommunizieren darf, jedoch nicht umgekehrt.

Was tun mit zugekaufter Software?

Da die Anwendungslandschaft einer Organisation meist nicht nur aus eigenentwickelter Software besteht, sondern im Gegenteil häufig sogar der Anteil zugekaufter Software deutlich größer ist, besteht hier Handlungsbedarf. Dabei stellt sich erneut die Frage, welche Produkte tatsächlich von Schwachstellen betroffen sind und wo man diese Information herbekommt. Steht ein Softwarearchiv zur Verfügung, etwa in Form einer JAR- oder WAR-Datei, und ist die Runtime bekannt (beispielsweise JBoss Wildfly in Version x.y), lässt sich beides per Syft analysieren, um wiederum eine SBOM zu erzeugen. Diese lässt sich anschließend in Dependency-Track importieren, um Analysen durchzuführen.

Empfehlenswert ist es, in Ausschreibungen die Anforderung aufzunehmen, dass der Softwarehersteller bei jeder ausgelieferten Version eine SBOM mitliefern muss. Darüber hinaus sollten Unternehmen Service-Level-Agreements (SLAs) vereinbaren, die den Anbieter im Falle einer kritischen Sicherheitslücke zum zeitnahen Liefern von Patches verpflichten.



Mikrosegmentierung mit Service Meshes (Abb. 6).

Fazit: Gezielte Vorsorge schützt vor Schwachstellen

Die im Artikel beschriebenen Methoden und Techniken helfen dabei, einen Überblick über die gesamte Anwendungslandschaft zu erzeugen und Sicherheitslücken in der Software Supply Chain frühzeitig zu identifizieren, um so das Risiko zu reduzieren, Opfer einer Cyberattacke zu werden.

Die wichtigsten empfehlenswerten Maßnahmen noch einmal zusammengefasst:

- Generieren von SBOMs für eigenentwickelte Software (automatisiert im Rahmen von CI/CD);
- Verpflichtung der Hersteller von Software zum Bereitstellen einer SBOM zu jeder ausgelieferten Version (Aufnahme in Ausschreibungen);
- Einführung einer Software zur kontinuierlichen Analyse von SBOMs;
- Ausführungsschicht überwachen;
- Plan zum Umgang mit den Analyseergebnissen festlegen;
- kontinuierlich fixen und deployen.

Die durch die beschriebenen Maßnahmen entstehende Transparenz löst sicherlich nicht bei allen Betroffenen Glücksgefühle aus. Doch in Anbetracht der täglich zunehmenden Cyberattacken wäre es grob fahrlässig, nach dem Motto „Was ich nicht weiß …“ zu verfahren – zumal es als Entschuldigung im Schadensfall inakzeptabel ist. Andererseits sollten Softwarehersteller aktiv SBOMs bereitstellen und sich um kontinuierliches Aktualisieren ihrer Software und der darin enthaltenen Third-Party-Libraries bemühen.

Eine Infografik fasst die wichtigsten Erkenntnisse zusammen (siehe ix.de/zf49). Happy patching! (map@ix.de)

Quellen

Weiter gehende Informationen und Links zu den erwähnten Tools: ix.de/zf49

STEPHAN KAPS

leitet die Softwareentwicklung im Bundesamt für Soziale Sicherung und ist Gründer der Java User Group Bonn.