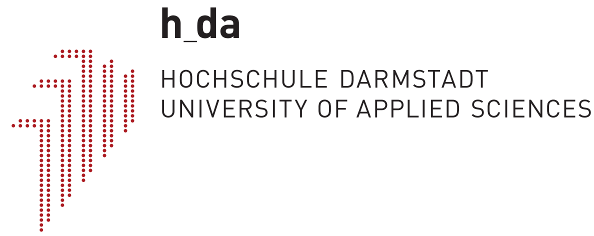


Hochschule Darmstadt

– Fachbereich Informatik–

Vergleich von Security Dependency Scanner

Wissenschaftliche Arbeit

Seminar: Problemlösung und Diskussion

vorgelegt von

Tim Stoffel

Referent : Prof. Dr.-Ing. Michael Bredel

ERKLÄRUNG

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, 18. März 2019

Tim Stoffel

ZUSAMMENFASSUNG

In vielen Softwareprojekten werden externe Softwarebibliotheken verwendet, um Standardprobleme zu lösen. Dadurch können aber neue Gefahren entstehen: In den OWASP TOP 10 findet sich auf Platz 9 das Ausnutzen von bekannten Sicherheitslücken in Fremdbibliotheken.
Um dieses Risiko zu minimieren ist es notwendig, die eingebunden Bibliotheken regelmäßig auf bekannte Sicherheitslücken zu prüfen. Ab einer gewissen Anzahl an Abhängigkeiten lässt sich diese Aufgabe kaum noch manuell vornehmen. Deshalb ist es notwendig diese Prüfung innerhalb der Continous-Integration-Pipeline automatisiert durchzuführen.
Ziel dieser Arbeit ist der Vergleich verschiedener Tools, welche die Abhängigkeiten eines Projektes anhand von CVE Datenbanken auf Sicherheitslücken prüfen. Dafür werden im ersten Schritt Anforderungen analysiert und ein Anforderungskatalog erstellt. Im zweiten Schritt werden die Tools nach diesen Anforderungen untersucht und eine Bewertung vorgenommen. Anschließend wird das Tool beispielhaft in eine Continous-Integration-Pipeline integriert.

INHALTSVERZEICHNIS

I THESIS

1 EINLEITUNG 2

2 GRUNDLAGEN 4

2.1 Begriffsklärung 4

2.2 Das Praxisprojekt 5

2.3 Praxisprojekt und Security Dependency Scanner 5

3 ABHÄNGIGE BIBLIOTHEKEN ALS SICHERHEITSRISIKO 6

3.1 Umgang mit verwundbaren Bibliotheken 6

3.2 Security Dependency Scanner als Lösung 7

3.3 Betrachtungsraum 8

4 ANFORDERUNGSANALYSE 9

4.1 Anforderungssammlung 9

4.2 Anforderungsübersicht 10

4.3 Auswertung 11

5 TOOLVERGLEICH 12

5.1 OWASP - Dependency-Check 12

5.2 OWASP - Dependency-Track 13

5.3 GitLab - Dependency Scanning 14

5.4 Vergleich der Scannergebnisse 14

5.5 Handlungsempfehlung 15

6 FAZIT 17

6.1 Zusammenfassung 17

6.2 Ausblick 17

II APPENDIX

A ANLAGEN 19

LITERATUR 24

Teil I

THESIS

1 EINLEITUNG

Bei der Entwicklung moderner und komplexer Softwareprojekte greifen die Entwickler häufig auf Bibliotheken zurück [vgl. 1, S. 3]. Diese Abhängigkeiten übernehmen definierte Aufgaben, wie z.B. Verschlüsslung oder Dateiübertragung. Die Bibliotheken wurden nicht von den Entwickelnden programmiert, sondern diese stammen aus den Paketdatenbanken der jeweiligen Programmiersprache.
Diese Bibliotheken werden auch als Pakete oder Abhängigkeiten (englisch: Dependencys) bezeichnet. Das Softwareprojekt ist von diesen Paketen abhängig, da diese Funktionen anbieten, die für die Ausführung der Software notwendig sind.
Dieses Vorgehen hat sich etabliert, da eine Vielzahl an frei verfügbaren, meist OpenSource, Bibliotheken existieren [vgl. 2, S. 389], die Standardprobleme lösen. Diese Standardprobleme sind Problemstellungen, die häufig und regelmäßig in Softwareprojekten auftauchen. Der Entwickelnde muss hier Funktionalität nicht neu entwickeln, sondern kann auf bestehende Bibliotheken zurückgreifen. Dieses Vorgehen erhöht die Effektivität des Entwicklungsteams und verkürzt die Entwicklungszeit [vgl. 1, S. 3, vgl. 3, S. 15].
Die Implementierung in den Bibliotheken ist häufig ausgereifter, als es ein Entwicklungsteam selbst lösen könnte. Die häufige Verwendung einer solchen Bibliothek in verschiedenen Projekten und Unternehmen schafft eine breite Testabdeckung, die meist zu einer hohen Qualität führt [vgl. 3, S. 15].
Die Verwendung dieser Bibliotheken beschleunigt die Entwicklung von Software enorm, da viele Funktionen schnell umgesetzt werden können [vgl. 1, S. 3]. Einige Bibliotheken sind wiederum auf die Einbindung von anderen Bibliotheken angewiesen. Dadurch steigt die Anzahl der verwendeten Abhängigkeiten in einem Softwareprojekt schnell an und es entstehen Abhängigkeitsbäume.
Durch die hohe Anzahl der Pakete steigt die Komplexität. Werden alte Versionen der Abhängigkeiten verwendet, kann dies zu Sicherheitslücken führen. Das Open Web Application Security Project (OWASP) veröffentlichte 2017 seine Top 10 der Sicherheitslücken für Webanwendungen [vgl. 4, vgl. 2, S. 389], dabei war die „Nutzung von Komponenten mit bekannten Schwachstellen“ auf Platz neun [vgl. 4, S. 16].
Um Entwickelnden den Umgang mit Abhängigkeiten zu vereinfachen, wäre es sinnvoll, eine Softwarelösung in den Softwareentwicklungsprozess miteinzubeziehen: Diese Lösung kann prüfen, ob für die verwendeten Bibliotheksversionen Sicherheitslücken gemeldet wurden. Wenn Schwachstellen entdeckt werden, können die Entwickelnden benachrichtigt werden.

In der vorliegenden Arbeit wird eine Integrationsmöglichkeit für eine Sicherheitsprüfung der Abhängigkeiten in einem Projekt vorgeschlagen. Weiter werden für eine solche Lösung Anforderungen aufgenommen. Aufgrund dieser Anforderungen werden verschiedene Programme ausgewählt. Diese Auswahl wird miteinander verglichen und daraus eine Empfehlung abgeleitet. Abschließend wird die bevorzugte Lösung innerhalb des Praxisprojektes umgesetzt.

2 GRUNDLAGEN

In diesem Kapitel werden die wichtigsten Begriffe abgegrenzt und der Kontext zum Praxisprojekt definiert.

2.1 BEGRIFFSKLÄRUNG

In der weiteren Arbeit werden die folgenden Begriffe verwendet und hier definiert:

Mit **Continuous Integration**, kurz **CI**, wird der automatisierte Prozess bezeichnet, in dem nach jedem Check-In des Entwickelnden ein zentraler Server alle Softwarebestandteile vereint, kompiliert und die Funktionstüchtigkeit mit automatischen Tests überprüft. Treten dabei Fehler auf, wird das Entwicklungsteam benachrichtigt [vgl. 5, S. 7]. Dieser Prozess basiert auf den Prinzipien des Agilen Manifests [vgl. 6].

Als **Dependency** oder **Abhängigkeit** wird eine externe Bibliothek in einem Softwareprojekt bezeichnet [vgl. 2, S. 389]. Extern kann dabei bedeuten, dass die Bibliothek aus einem anderen Team innerhalb des Unternehmens oder außerhalb der Firma stammt.
Diese Bibliotheken sind meist öffentlich und werden in den meisten Programmiersprachen in zentralen Datenbanken bereitgestellt. Ein Beispiel dafür ist das Maven Repository[1] für Java oder die NuGet Gallery[2] für .Net.

Common Vulnerability Enumeration, kurz **CVE**, ist ein Standard, nach dem Sicherheitslücken eindeutig benannt werden. Damit wird gewährleistet, dass die Schwachstelle herstellerübergreifend einmalig ist und keine Mehrfachbenennungen auftreten.
Die Identifikation erfolgt über eine eindeutige Nummer, die von der Mitre Corporation vergeben wird. Damit wird der einfache Informationsaustausch zwischen verschiedenen Datenbanken von Sicherheitslücken möglich [vgl. 2, S. 57].
Den Lücken wird meist ein Wert zugeordnet, der beschreibt, wie schwerwiegend diese Sicherheitslücke ist. Die Vergabe dieses Wertes wurde nach dem Common Vulnerability Scoring System (CVSS) standardisiert [vgl. 7, S. 5].
Eine der größten Datenbanken zu CVEs ist die National Vulnerability Database (NVD), die von dem National Institute of Standards and Technology in der USA verwaltet wird.

1 central.maven.org/maven2/
2 nuget.org/packages

Ein **Security Dependency Scanner** ist eine Software, die alle Abhängigkeiten mit ihren Versionen eines Softwareprojektes sammelt und diese dann mit CVE Datenbanken abgleicht. Werden zu den Bibliotheksversionen Sicherheitslücken gefunden, werden diese an die Entwickelnden weitergegeben [vgl. 2, S. 389].

2.2 DAS PRAXISPROJEKT

Die Firma :em engineering methods AG ist ein Beratungs- und Dienstleistungsunternehmen und Atlassian Solution Partner. Teil des Unternehmens ist eine Softwareabteilung, in der das folgende Projekt angesiedelt ist.
Als Atlassian Solution Partner entwickeln wir für unsere Kunden Plugins für die verschiedenen Atlassian Produkte, um ihre Prozesse bestmöglich durch diese Produkte zu unterstützen.
Das Praxisprojekt „Aufbau einer Continuous-Integration-Pipeline und Entwicklung eines Atlassian Plugins“ hat die Zielsetzung den Entwicklungsprozess von Plugins zu optimieren. Dazu wird eine CI-Pipeline eingeführt.
Die Zielsetzung ist hierbei den manuellen Testaufwand zu senken und die Qualität der Software zu erhöhen.

2.3 PRAXISPROJEKT UND SECURITY DEPENDENCY SCANNER

Da im Praxisprojekt eine CI-Pipeline aufgebaut wird und diese als Vorlage für zukünftige Projekte dient, ist es angebracht Sicherheitsaspekte nicht unberücksichtigt zu lassen.
Teil des Projektes ist die Einbindung in ein System zur statischen Codeanalyse[3]. Dieses System warnt die Nutzenden auch vor Programmierfehlern, die zu Sicherheitslücken führen können. Beispiel dafür sind Buffer-Overflows oder SQL Injection.
Bei der Erstellung eines Atlassian Plugins fiel auf, dass z.B. für Jira[4] über 100[5] abhängige Bibliotheken eingebunden werden. Bei dieser Anzahl ist es schwer zu prüfen, ob bekannte Sicherheitslücken existieren. Dazu kommt der Aufwand, die Sicherheitslücken zu bewerten.
Innerhalb dieser Arbeit wird geprüft, inwieweit ein solcher Prüfungsprozess in die CI-Pipeline integriert werden kann.

3 sonarqube.org
4 Ticket- und Bugtracking System
5 Eigene Zählung

3 ABHÄNGIGE BIBLIOTHEKEN ALS SICHERHEITSRISIKO

Häufig werden in Softwareprojekten veraltete Bibliotheken eingebunden. Eine Studie von Contrast Security führt aus, das über ein Viertel der heruntergeladene Java Bibliotheken aus dem „Central Repository“ bekannte Sicherheitslücken besitzen [vgl. 8, S. 14].
Die Entwicklungsteams müssen daher im Umgang mit Abhängigkeiten die Auswirkungen auf die Sicherheit des Gesamtsystems betrachten.

3.1 UMGANG MIT VERWUNDBAREN BIBLIOTHEKEN

Besteht in einer verwendeten Bibliothek eine Sicherheitslücke, müssen die Entwickelnden darauf reagieren. Zuerst ist zu prüfen, ob durch die Lücke in der Abhängigkeit ein Sicherheitsrisiko für das Softwareprojekt besteht.
Einige Abhängigkeiten werden nur für die Kompilierung des Projektes benötigt, werden also zur Laufzeit nicht verwendet. Das Risiko ist hier sehr gering. Andere Lücken der Abhängigkeit können, durch die Art und Weise wie diese innerhalb des Projektes verwendet werden, nicht auftreten. Auch hier ist kein direkter Handlungsbedarf geboten.
Wird festgestellt, dass ein Sicherheitsrisiko entsteht, müssen die Entwickelnden dem entgegenwirken. Guy Podjarny hat mithilfe der „Snyk vulnerability DB“ ermittelt, wie hoch der Anteil der durch Updates geschlossenen Sicherheitslücken ist: „In npm, 59% of reported vulnerabilities have a fix. In Maven, 90% are remediable, while that portion is 85% in RubyGems.“ [9]
Die Firma Risk Based Security führt in ihrem Report „2018 Vulnerability Trends“ aus, dass 71,1% aller Schwachstellen durch Updates geschlossen werden können [vgl. 10, S. 10].
Das Updaten der betroffenen Bibliothek senkt daher in vielen Fällen das Sicherheitsrisiko. Dieser Schritt ist in der Praxis häufig nicht einfach. Wenn z.B. ein Major Upgrade ansteht, reicht allein das Upgrade der Bibliothek nicht aus. In der Regel werden damit die Schnittstellen, über die die Software mit der Bibliothek kommuniziert, verändert und die Entwickelnden müssen zuerst prüfen, ob diese Auswirkungen auf ihr Projekt hat und gegebenenfalls ihren Programmcode anpassen. Minor Updates dagegen sollten sich meist ohne Mehraufwand durchführen lassen.
Eine weitere Herausforderung entsteht aus transitiven Abhängigkeiten. Im Beispiel, siehe Listing 3.1, verwendet die Applikation die Bibliothek B zweimal in unterschiedlichen Versionen. Taucht jetzt in B ab Version 1.2 eine Sicherheitslücke auf können die Entwickelnden die direkt eingebunden Bibliothek B@1.2.0 updaten. Die transitive Abhängigkeit B kann nur upgedatet werden, wenn A entsprechend angepasst wird. Die Entwickelnden müssen

aus diesem Grund regelmäßig prüfen, ob A ein Update bereitstellt, damit sie B updaten können [vgl. 9].

Listing 3.1: Transitive Abhängigkeiten

```
Applikation
 |-- A@2.0.3
 | +-- B@1.2.0 (transitiv)
 +-- B@1.2.0 (direkt)
```

Außerdem können Schwierigkeiten auftreten, wenn Abhängigkeitskonflikte entstehen. In einigen Programmiersprachen, wie Python oder Ruby [vgl. 9] sind Abhängigkeiten im Projekt global verfügbar. Es ist daher kaum möglich verschiedene Versionen einer Bibliothek zu verwenden. Tritt das eben genannte Beispiel auf, kann kein Update der direkten Abhängigkeit durchgeführt werden, da sonst Konflikte mit A entstehen würden. Gerade bei häufig transitiv verwendeten Abhängigkeiten wächst damit die Komplexität.
Steht kein Update bereit, das die Schwachstelle beseitigt, ist das Patchen der Lücke eine Lösung. Häufig wird ein Patch aus der Community heraus bereitgestellt, der aber noch nicht vom Herausgebenden der Abhängigkeit integriert wurde. Hier ist zu prüfen, inwieweit dieser Patch ins eigene Projekt übergangsweise eingebunden werden kann, bis der Patch über eine neue Bibliotheksversion bereitsteht.
Steht nach einer gewissen Zeit weder ein Update, noch ein Patch bereit, sollten die Entwickelnden prüfen, inwieweit sie auf die Bibliothek verzichten können und diese anschließend aus dem Projekt entfernen. Wird die Bibliothek benötigt, sollte eine Alternative gefunden werden und diese ins Projekt integriert werden.

3.2 SECURITY DEPENDENCY SCANNER ALS LÖSUNG

Durch die hohe Anzahl der Abhängigkeiten und die daraus entstehenden transitiven Abhängigkeiten wird der Abhängigkeitsbaum sehr schnell komplex. Eine manuelle Überprüfung der eingebundenen Bibliotheken nach bekannten Sicherheitslücken ist daher nicht praxistauglich.
Ein Security Dependency Scanner kann den Entwickelnden diese Arbeit abnehmen, indem er automatisch die Abhängigkeiten des Projektes auf ihre bekannten Sicherheitslücken mithilfe von CVE-Datenbanken überprüft. Eine solche Lösung kann an verschiedene Positionen des Entwicklungsprozesses eingebunden werden [vgl. 11]. Die Integration ist in der Entwicklungsumgebung auf dem Computer des Entwickelnden, der CI Umgebung oder im Artefakt-Repository möglich.
Bekommt der Entwickelnde beim Einbinden einer Abhängigkeit den Hinweis, dass eine Sicherheitslücke besteht, kann er die Einbindung entsprechend anpassen oder eine andere Bibliothek verwenden. Wird das Projekt aber lange nicht in einer Entwicklungsumgebung geöffnet, kann auf neue

Lücken nicht reagiert werden. Auch ist diese Prüfung nicht transparent für das Team, sondern nur für den jeweiligen Entwickelnden sichtbar.
Die Integration in den Buildserver hätte den Vorteil, dass die Prüfung regelmäßig ausgeführt werden könnte, z. B. jede Nacht. Auf neue Lücken, könnte dann entsprechend reagiert werden. Der Nachteil hierbei ist, dass der Entwickelnde ein verzögertes Feedback bekommt.
In einigen Firmen werden Artefakt-Repositories verwendet. Die Bibliotheken kommen so nicht direkt aus dem Internet, sondern werden innerhalb des Netzwerks zwischengespeichert. Ein Security Dependency Scanner kann hier den Bezug einer unsicheren Bibliothek direkt unterbinden [vgl. 2, S. 390].
Innerhalb des Praxisprojektes wird als Integrationspunkt der Buildserver gewählt. Die Entwicklungsumgebung ist im Unternehmen stark fragmentiert, eine Lösungsfindung für alle verwendeten Kombinationen wäre zu aufwendig. Der Buildserver kann die Bibliotheken auch langfristig überwachen. Bisher wird kein Artefakt-Repository verwendet, daher kann hier keine Integration vorgenommen werden.
Damit wird eine zwingende Anforderung an die Lösung eine Einbindung in die CI-Pipeline sein. Die Integration muss so erfolgen, dass der Buildserver eine Überprüfung der Abhängigkeiten anstoßen kann.

3.3 Betrachtungsraum

In der Recherche vor der Anforderungsanalyse wurde nach Kandidaten gesucht, die geprüft werden sollten. Dabei wurden Programme aufgenommen, die die Abhängigkeiten aus Softwareprojekten auslesen und mit Datenbanken über Sicherheitslücken vergleichen. Das Softwareprojekt kann für diese Überprüfung in Form von Quellcode oder als kompilierte Binärdatei vorliegen.
Sonatype betreut mit dem OSS Index[1] eine weit verbreitete Datenbank zu Sicherheitslücken. Dazu entwickelt Sonatype verschiedene Lösungen, wie Audit.Net, DepShield, ossindex-gradle-plugin, Maven plugin for Sonatype OSS Index, AppScan, AuditJS und DevAudit, die auf unterschiedliche Programmiersprachen spezialisiert sind. Die Tools werden für die Anforderungsprüfung nicht zusammengefasst, da eine Verwendung aller Tools zu einer starken Fragmentierung führen würde.
Die in Tabelle A.1 im Anhang genannten Tools werden auf die Anforderungen, die im folgenden Kapitel definiert werden, geprüft.

1 ossindex.sonatype.org

4 ANFORDERUNGSANALYSE

Um die bestmögliche Lösung zu finden, wurden zu Beginn die formalen Anforderungen aufgenommen. Dazu wurden alle Softwarearchitekten und die beteiligten Entwickler des Unternehmens befragt.
Im ersten Schritt werden alle in Betracht gezogenen Anwendungen auf die Anforderungen geprüft.
Im zweiten Schritt werden die Anwendungen, die diese Anforderungen erfüllen prototypisch installiert und eingerichtet. Anschließend werden verschiedene Projekte beispielhaft überprüft und die Ergebnisse verglichen.

4.1 ANFORDERUNGSSAMMLUNG

Die Anforderungen wurden mit den Projektbeteiligten definiert und nach ihrer Verifizierbarkeit formuliert.

Einfache Integration in die Continous-Integration-Pipeline
Die Lösung soll sich einfach in die CI-Pipeline einbinden lassen. Das kann über eine Comand-Line-Interface erfolgen, dass direkt auf dem Buildserver ausgeführt wird, oder als ein Plugin für den Buildserver. Auch ist eine Einbindung über eine eigene Serveranwendung, die vom Buildserver angesprochen wird, denkbar. Eine solche Serveranwendung sollte möglichst als Docker Container bereitstehen, damit sie schnell in Betrieb genommen werden kann. Eine Integration als Buildplugin für Maven oder Gradle hätte den Vorteil, dass es auch einfach auf dem Entwicklerrechner verwendet werden könnte.
Diese Anforderung gilt als erfüllt, wenn eine der genannten Integrationsmöglichkeiten vorhanden ist.

Software muss OnPremise nutzbar sein
Die Daten darüber, ob eine CVE für eine bestimmte Version einer Bibliothek vorliegt, stammt in der Regel aus bestimmten Datenbanken außerhalb des Firmennetzes. Die Lösung soll einen Client bereitstellen, über den die erforderlichen Daten gesammelt werden und mit der Datenbank verglichen werden. Ein Upload des gesamten Quellcodes darf dabei nicht erfolgen.
Wird eine solcher Client bereitgestellt, gilt die Anforderung als erfüllt.

Betrieb der Software muss kostenlos sein
Da das Management für diesen Bestandteil der CI-Pipeline kein Geld investieren möchte, muss die Lösung kostenlos einsetzbar sein.
Wenn die Lösung kostenlos für OnPremise nutzbar ist und eine CI-Integration möglich ist, wird die Anforderung erfüllt. Darüber hinaus soll der Preis

bei ca. 400 Prüfungen pro Monat kostenlos bleiben. Die Zahl ergibt sich aus den durchschnittlich 20 Projekten in Entwicklung oder mit Wartungsvertrag bei mindestens einer Ausführung pro Arbeitstag.

Unterstützung der gängigen Programmiersprachen
Innerhalb des Unternehmens werden die Programmiersprachen C# mit .Net, Java, Python und JavaScript verwendet. Die abhängigen Bibliotheken werden über die für die Sprachen üblichen Paketmanager eingebunden. Für C# ist das NuGet, für Java Gradle und Maven, für Python PyPi und für JavaScript npm.
Die Lösung sollte mindestens drei der vier genannten Programmiersprachen unterstützen. Dadurch würden die meisten Softwareprojekte von der Softwarelösung profitieren und eine Fragmentierung vermieden werden.

Ergebnisse
Die Lösung sollte möglichst alle Abhängigkeiten erkennen und die entsprechenden Sicherheitslücken zuordnen.
Diese Anforderung wird überprüft, indem die Ergebnisse der Lösungen verglichen werden.

Lösung sollte OpenSource sein
Um verifizieren zu können, wie die Software arbeitet, sollte die Software OpenSource sein.
Diese Anforderung ist eine Soll-Anforderung.

4.2 ANFORDERUNGSÜBERSICHT

Die Anforderungen lassen sich in Muss- und Soll-Anforderungen unterteilen. Muss-Anforderungen müssen in jedem Fall erfüllt werden. Die Ergebnisse der Softwarelösungen werden erst im zweiten Schritt betrachtet. Einen Überblick über die Anforderungen liefert die Tabelle 4.1.

Tabelle 4.1: Anforderungsübersicht

Muss-Anforderungen	
Betrieb der Software muss kostenlos sein	
Software muss OnPremise nutzbar sein	
Einfache Integration in die CI-Pipeline	*CLI, Docker, Buildserverplugin, Buildsystem*
Unterstützung der gängigen Programmiersprachen	*Java, C#, Python, npm*
Soll-Anforderungen	
OpenSource	

4.3 AUSWERTUNG

Von den in 3.3 genannten Tools, erfüllen nur drei die Muss-Anforderungen, siehe Tabelle 4.2. Alle gemeinsam sind OpenSource und erfüllen damit auch die Soll-Anforderung. Eine Übersicht über alle betrachteten Lösungen kann Tabelle A.2 im Anhang entnommen werden.
Beide OWASP Tools unterstützen alle geforderten Programmiersprachen. Der GitLab Dependency Scanner kann keine NuGet Abhängigkeiten überprüfen.

Tabelle 4.2: Ergebnisse der Anforderungsanalyse

Name	CI Integration	Java	NuGet	Python	npm
Dependency-Check	Ja[a,b,c]	Ja	Ja	Ja	Ja
Dependency-Track	Ja[a]	Ja	Ja	Ja	Ja
GitLab Dependency Scanning	Ja[c]	Ja	Nein	Ja	Ja

[a] Plugin für Buildserver [b] Gradle und Maven Plugin [c] Docker Container

5 TOOLVERGLEICH

Die folgenden drei Lösungen wurden in die CI-Pipeline integriert und die Leistungsfähigkeit der Tools anhand verschiedenen Projekten verglichen.

Tabelle 5.1: Verwendete Softwareversionen

Name	Hersteller	Version
Dependency-Check	OWASP	5.0.0-M1
Dependency-Track	OWASP	3.4.0
GitLab Dependency Scanning	GitLab	11-9-stable

5.1 OWASP - DEPENDENCY-CHECK

OWASP Dependency-Check (DC) ist eine Software des Open Web Application Security Projects (OWASP). Das OWASP ist eine unabhängige Organisation mit der Zielsetzung, die Sicherheit in Anwendungen zu erhöhen. Die erste Version von Dependency Check ist 2012 erschienen [vgl. 12, S. 8].
DC ist als Plugin für Ant, Gradle, Maven, Jenkins oder SBT verfügbar, um in den Buildvorgang einbezogen zu werden. Mit dem Kommandozeilenprogramm ist es möglich, kompilierte Projekte zu scannen. Auch ein Plugin für Sonarqube ist vorhanden [vgl. 13, S. 13].
DC unterstützt dabei Java, NodeJS, NuGet, npm und Ruby. Die Scanner für Autoconf, CMake, CocoaPods, PHP Composer, Python, JavaScript und Swift sind experimentell [vgl. 14].
Der Scanner sucht nach Abhängigkeiten und ordnet diesen eine Common Platform Enumeration (CPE) zu. CPE ist ein Standard für die einheitliche Namensvergabe von Softwareprodukten [vgl. 15]. Die Zuordnung der CPEs erfolgt heuristisch. Dazu vergibt das Tool einen „Evidence Score". Dieser Score sagt aus, inwieweit die Zuordnung zwischen Abhängigkeit und CPE sicher ist. Die CPE wird mit der National Vulnerability Database (NVD) verglichen und die CVEs gesammelt. Die NVD wird dabei lokal zwischengespeichert.
Die gefundenen Abhängigkeiten und CVEs können als Report in den Formaten XML, HTML, CSV oder JSON ausgegeben werden. Der HTML Report ist dabei visuell gut aufbereitet, so kann ein Einwickelnder die Daten gut einsehen. Die Detailinformationen zu den CVEs sind über einen Link zur NVD erreichbar. Ein Beispielreport ist im Anhang bei Abbildung A.1 einsehbar.
Die Einbindung in die CI-Pipline erfolgte bei den Java Projekten via Maven Plugin. Alle anderen Projekte wurden mit dem Kommandozeilenprogramm überprüft. Das Tool wurde via Script auf dem Buildserver ausgeführt.

Neben der Integration in den Buildserver, kann auch jeder Entwickelnde auf seinem Computer DC ausführen, um ein direktes Feedback zu bekommen. Dabei ist zu empfehlen die NVD zentral im Netzwerk zwischenzuspeichern, damit das Abfragelimit der NVD nicht erreicht wird.

5.2 OWASP - DEPENDENCY-TRACK

OWASP Dependency-Track (DT) wird wie DC von dem OWASP verwaltet. Diese Lösung läuft als Webserver und nicht als Plugin oder Kommandozeilenprogramm. DT versteht sich als Plattform, um die Abhängigkeiten von Projekten im Unternehmen zu überwachen und zu verwalten. Die Plattform wird als Docker Container oder Java Applikation bereitgestellt.
DT erkennt selbst keine Abhängigkeiten in einem Projekt. Es liest entweder sogenannte BOMs oder die Reports von DC im XML Format. BOM steht für Bill of Material. Hierbei handelt es sich um XML Dateien die nach den Standards CycloneDX[1] oder SPDX[2] aufgebaut sind [vgl. 16]. Diese Dateien listen alle Abhängigkeiten mit ihren Versionen und Lizenzen eines Projektes auf. Die Zuordnung von Abhängigkeiten zu den CVE Datenbanken erfolgt hierbei über die PackageURL.
Die Lösung unterstützt Abhängigkeiten aus folgenden Repositories RubyGems, Maven, npm, PyPi und NuGet [vgl. 17].
In DT können BOMs zu Projekten hinzugefügt werden. Die Lösung prüft alle sechs Stunden ob CVEs zu den Abhängigkeiten bestehen, nachdem es die Datenquellen aktualisiert hat. Die Daten zu den CVEs stammen aus der NVD, npm, VulnDB und Sonatype OSS Index [vgl. 17]. Der Zugriff auf VulnDB und Sonatype OSS Index ist nur mit einer Registrierung bei den jeweiligen Datenbanken möglich. Die Software stellt nach dem ersten Abgleich mit der NVD, die Datenbank der NVD im Netzwerk zu Verfügung. Diese lässt sich anschließend z.B. in DC verwenden.
Die CVEs zu Abhängigkeiten in den Projekten sind über die Weboberfläche einsehbar. Dazu werden die Details zu den CVEs, wie z.B. der Score in der Oberfläche visualisiert. Die dazugehörigen Einträge in den Datenbanken sind verlinkt. Weiter können Benachrichtigungen gesendet werden, wenn neue Sicherheitslücken in den verwendeten Abhängigkeiten auftauchen. Auch wird dargestellt, welche Bibliotheken in der Firma verwendet werden und in welchen Projekten diese eingebunden sind.
Um die Lösung in eine CI-Pipeline zu integrieren, wurde auf dem Buildserver bei jedem Build ein BOM erzeugt. Dazu würde das für die jeweilige Programmiersprache kompatible CycloneDX Kommandozeilenprogramm verwendet. Das BOM wurde anschließend via REST an den DT Server übermittelt.

1 cyclonedx.org
2 spdx.org

5.3 GITLAB - DEPENDENCY SCANNING

Dependency Scanning (DS) von GitLab ist ein Bestandteil der GitLab Plattform [vgl. 18] und als Docker Container verfügbar.
Dabei analysiert der Container, um welches Projekt es sich handelt und startet danach einen, auf die Programmiersprache spezialisierten Container. Der spezialisierte Container sucht nach den Abhängigkeiten und übermittelt sie an den GitLab Server. Von diesem Server erhält der Scanner die Sicherheitslücken zu allen Versionen der Abhängigkeit. Aus diesen Informationen wird ein Report im JSON Format erzeugt. Gleichzeitig werden die Ergebnisse innerhalb der Kommandozeile präsentiert, siehe dazu Abbildung A.3 im Anhang.
DS unterstützt JavaScript, Python, Ruby, Java und PHP [vgl. 19]. Die Daten zu den Sicherheitslücken stammen von GitLab.
Für die Integration in den Buildserver wurde der Docker Container mit einem Skript gestartet und der Report den Entwickelnden zu Verfügung gestellt. Die JSON Datei ist dabei von Menschen schwer lesbar. Details zu den CVEs sind in dem Report verlinkt.

5.4 VERGLEICH DER SCANNERGEBNISSE

Für den weiteren Vergleich der Tools wurden mehrere Projekte ausgewählt. Damit sollen die Lösung hinsichtlich ihrer Erkennung von Abhängigkeiten und ihres Auffindens von Sicherheitslücken überprüft werden. Die Ergebnisse sind dabei nicht vollständig, sondern geben einen Eindruck über die Erkennungsraten.
Zu diesem Zweck wurden verschiedene Projekte innerhalb der Firma ausgewählt, die Namen der Projekte wurde verändert. Zur Betrachtung wurden zwei Java Projekte mit Maven und zwei Projekte mit NuGet Paketverwaltung in C# herangezogen. Jeweils eins dieser Projekte wurde seit mindestens einem Jahr nicht gewartet. Zusätzlich wurde ein npm Projekt mit TypeScript und Angular und ein Python Projekt ausgewählt.
Die Abhängigkeiten des Python Projektes wurden in einer requirements.txt, in einer setup.py und als Metadaten im Paketformat egg[3] definiert. Bei den anderen Projekten ist das Format für die Definition der Abhängigkeiten durch das Repository oder die Buildumgebung vorgegeben.
Die Lösungen wurden wie genannt integriert und konnten die Projekte überprüfen. DT erkannte in dem Python Projekt keine Abhängigkeiten. Die Ergebnisse können der Tabelle 5.2 entnommen werden.
Die Resultate der Prüfungen sind heterogen. DC und DT erkannten unterschiedlich viele Abhängigkeiten pro Projekt. DS gab nicht aus, wie viele Bibliotheken erkannt werden. Der Unterschied bei der Erkennung der Abhängigkeiten ergibt sich aus den unterschiedlichen Algorithmen, die bei der Erkennung angewendet werden.

3 peak.telecommunity.com/DevCenter/PythonEggs

Tabelle 5.2: Ergebnisse der Scanner

	Dependency-Check		Dependency-Track		Dependency Scanning
Projekt	Dep.	Vuln.	Dep.	Vuln.	Vuln.
Java_Old	301	108	325	71	4
Java	36	0	32	0	0
.NET_Old	122	0	105	16	x
.NET	24	0	18	0	x
Python	x	x	6	0	0
npm	10561	1	86	0	0

Dep. = Anzahl Abhängigkeiten Vuln. = Anzahl Sicherheitslücken

Bei den Java Projekten gab es Unterschiede bei der Erkennung der transitiven Abhängigkeiten. Bei der Generierung der BOM für DT wurden die transitiven Abhängigkeiten zuverlässiger erkannt, das führt zu unterschiedlichen Zahlen bei dem Java_Old Projekt. DC erkannte die Duplikate bei „Shaded Libaries“ nicht zuverlässig und führte diese zusätzlich auf, dies ist beim Java Projekt ersichtlich.
DC führte bei den .NET Projekten die „package.config“ als Abhängigkeit auf. Diese Datei spezifiziert die Abhängigkeiten, stellt selbst aber keine Bibliothek dar. Im Projekt .NET_Old erkannte DC 47 mal „package.json“ als Bibliothek. Zusätzlich erkannte DC nicht immer alle Versionen einer Abhängigkeit in einem Projekt.
Im Python Projekt erkannte nur das Tool zur BOM Erzeugung die Abhängigkeiten anhand der „requirements.txt“.
Im Kontext des npm Projektes waren die Unterschiede bei der Erkennung immens. DT erkannte die transitiven Abhängigkeiten im Projekt nicht, DC schon.
Bei der Zuordnung von Bibliotheken zu den CVEs waren die Zahlen unterschiedlich, da alle Tools verschiedene Datenbanken als Quelle verwenden, von denen keine vollständig ist. Insgesamt fand DS von GitLab deutlich weniger Fehler als die anderen beiden Lösungen. Die Datenbank von GitLab scheint demnach weniger umfangreich zu sein als die der NVD, die von DC und DT verwendet wird.
Manchmal ordneten DC und DT Sicherheitslücken falsch zu. Bei DC trat dies häufiger auf, als bei DT. Beide Tools bieten die Möglichkeit, wiederkehrende Falschmeldungen zu unterdrücken.

5.5 HANDLUNGSEMPFEHLUNG

Die Lösungen vereinfachen die Suche nach Sicherheitslücken in den verwendeten Abhängigkeiten deutlich. Die Zeit zur Konfiguration ist kurz im Vergleich zum Aufwand die Abhängigkeiten manuell zu überprüfen.
Innerhalb des Praxisprojektes wird DC und DT parallel verwendet werden. DC dient dazu, den Entwickelnden direktes Feedback zu geben. So wird im

Idealfall bei der Entwicklung keine Abhängigkeit eingebunden, die Sicherheitslücken aufweist.
DT wird die Abhängigkeiten kontinuierlich überwachen und die zuständigen Teams entsprechend benachrichtigen. Die BOM Erzeugung wird auf dem Buildserver erfolgen und von da automatisch in DT übertragen. Weiter ist DT damit ein Inventarisierungssystem über alle verwendeten Abhängigkeiten. Durch die Oberfläche wird ein guter Überblick gegeben, welche Lizenzen die Bibliotheken verwenden, sodass bei Lizenzänderungen reagiert werden kann.
Da beide Tools unterschiedliche Algorithmen verwenden, ist die Erkennungsrate der Abhängigkeiten größer und senkt damit das Risiko unsichere Bibliotheken zu verwenden.
Die Erkennungsrate der CVEs von DS waren im Vergleich zu gering, daher ist das Tools als einzige Datenquelle nicht zu empfehlen und findet keine Verwendung im Projekt.

6 FAZIT

6.1 Zusammenfassung

Damit eine Software die Sicherheitslücken in Abhängigkeiten finden kann, benötigt es die Bibliotheken in maschinenlesbarer Form. Gleichzeitig muss die Identifikation der Abhängigkeit erfolgen, da die Benennung nicht einheitlich ist. Beide Vorgänge sind noch nicht ausgereift. Die Ergebnisse zeigen, dass die Lösungen unterschiedliche Abhängigkeiten erkannt haben. Die Verfahren dazu sind noch unzuverlässig und fehleranfällig. Entwickelnde sollten darauf achten, die Abhängigkeiten über die für die Programmiersprache geltenden Konventionen und nicht über Umwege einzubinden. In Programmiersprachen, in denen die Konvention nicht so starr sind, wie z.B. Python, scheint die Erkennung eine Herausforderung zu sein.
Aber auch die Zuordnung von den Namen der Bibliotheken in der Software zu einer CPE oder PackageURL ist fehleranfällig. Gerade die Lösung DC ordnete regelmäßig CVEs falsch zu, die Versionsnummern stimmten nicht überein.
Insgesamt sollten sich Entwickelnden nicht vollständig auf die Tools verlassen. Es ist anzuraten, bei der Auswahl von Bibliotheken gründlich zu recherchieren und die Abhängigkeiten in den Projekten regelmäßig zu aktualisieren. Doch geben die Lösungen im jetzigen Entwicklungsstand eine gute Hilfestellung, um manuelle Arbeit zu reduzieren.

6.2 Ausblick

Die Überprüfung der Abhängigkeiten ist nur ein Bestandteil eines sicherheitsorientierten Entwicklungsprozesses. Es können weitere Tools und Prozesse einbezogen werden, um die Sicherheit zu steigern. Ein nächster Schritt kann die Integration von statischer Codeanalyse sein, die auf gängige Sicherheitsfehler prüft. Eine Software mit Unterstützung für die Programmiersprache Java ist Find Security Bugs[1].
Wenn ein Projekt eine hohe Testabdeckung besitzt, kann das Updaten der Abhängigkeiten auch maschinell erfolgen. Damit können, wie in 3.1 ausgeführt, die meisten Sicherheitslücken beseitigt werden.
Lösungen wie z.B. Renovate[2] übernehmen das Updaten automatisch. Dazu wird ein Pull-Request erstellt. Erfüllt dieser alle Tests, wird die Änderung in den Hauptzweig integriert. Haben die Projekte innerhalb des Unternehmens eine ausreichend hohe Testabdeckung, kann eine solche Software evaluiert werden.

1 find-sec-bugs.github.io
2 renovatebot.com

Teil II

APPENDIX

ANLAGEN

A

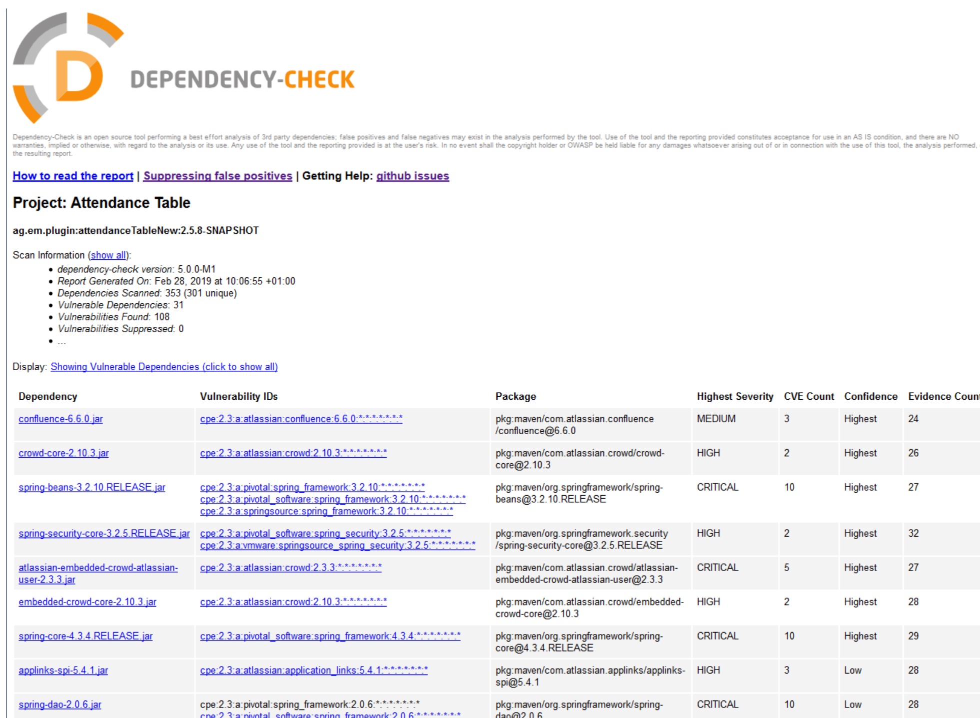
Tabelle A.1: Betrachtungsraum

Name	Hersteller	Link
AppScan	Sonatype	www.sonatype.com/appscan
Audit.Net	Sonartype	github.com/OSSIndex/audit.net
AuditJS	Sonatype	www.npmjs.com/package/auditjs
Black Duck Hub	Synopsys	www.blackducksoftware.com
bundler-audit	rubysec	github.com/rubysec/bundler-audit
Dependabot	Dependabot	dependabot.com
Dependency-Check	OWASP	github.com/jeremylong/DependencyCheck
Dependency-Track	OWASP	dependencytrack.org
DepShield	Sonartype	depshield.github.io
DevAudit	Sonatype	github.com/OSSIndex/DevAudit
DotNET Retire	Retire.NET Project	github.com/RetireNet/dotnet-retire
FlexNet Code Insight	Flexera	flexera.com/products/software-composition-analysis/flexnet-code-insight.html
Fossa	Fossa Inc.	fossa.com
GitLab Dependency Scanning	GitLab	gitlab.com/gitlab-org/security-products/dependency-scanning
Hakiri	OpsLog, LLC	hakiri.io
Ion Channel SA	Ion Channel	ionchannel.io
NPM Audit	NPM	www.npmjs.com
Open Source Lifecycle Management	WhiteSource Software	www.whitesourcesoftware.com
Open-source vulnerability assessment tool	SAP	github.com/SAP/vulnerability-assessment-tool
ossindex-gradle-plugin	Sonartype	github.com/OSSIndex/ossindex-gradle-plugin
PHP Security Checker	SensioLabs	github.com/sensiolabs/security-checker
Retire.js	RetireJS Project	retirejs.github.io
SafeNuGet	OWASP	github.com/OWASP/SafeNuGet
Safety	pyup	pyup.io/safety
Snyk	Snyk	snyk.io
SourceClear	Veracode	www.sourceclear.com
Xray	JFRog	jfrog.com/xray
Maven plugin for Sonatype OSS Index	Sonartype	sonatype.github.io/ossindex-maven/maven-plugin

Tabelle A.2: Vergleich der Lösungen

Name	Hersteller	Kostenlos	Open Source	On Premise	CI Integration	Java	NuGet	Python	npm
AppScan	Sonatype	Ja	Nein	Ja	Nein	Ja	Ja	Ja	Ja
Audit.Net[a]	Sonatype	Ja	Ja	Ja	Nein	Nein	Ja	Nein	Nein
AuditJS[a]	Sonatype	Ja	Ja	Ja	Ja[c]	Nein	Nein	Nein	Ja
Black Duck Hub	Synopsys	Nein	Nein	Ja	Ja[d]	Ja	Ja	Ja	Ja
bundler-audit	rubysec	Ja	Ja	Ja	Ja[c]	Nein	Nein	Nein	Nein
Dependabot	Dependabot	Nein	Nein	Nein[b]	Nein[b]	Ja	Ja	Ja	Nein
Dependency-Check	OWASP	Ja	Ja	Ja	Ja[d,e,f]	Ja	Ja	Ja	Ja
Dependency-Track	OWASP	Ja	Ja	Ja	Ja[d]	Ja	Ja	Ja	Ja
DepShield[a]	Sonatype	Ja	Nein	Nein[b]	Nein[b]	Ja	Ja	Ja	Ja
DevAudit[a]	Sonatype	Ja	Ja	Ja	Ja[c]	Nein	Ja	Nein	Ja
DotNET Retire	Retire.NET Project	Ja	Ja	Ja	Ja[c]	Nein	Ja	Nein	Nein
FlexNet Code Insight	Flexera	Nein	Nein	Ja	Ja[d,e]	Ja	Ja	Ja	Ja
Fossa	Fossa Inc.	Nein	Nein	Ja	Ja[d]	Ja	Ja	Ja	Ja
GitLab Dependency Scanning	GitLab	Ja	Ja	Ja	Ja[f]	Ja	Nein	Ja	Ja
Hakiri	OpsLog, LLC	Nein	Ja	Ja	Ja[c]	Nein	Nein	Nein	Nein
Ion Channel SA	Ion Channel	Nein	Nein	Ja	Ja[c,f]	Ja	Nein	Ja	Ja
NPM Audit	NPM	Ja	Ja	Ja	Ja[c]	Nein	Nein	Nein	Ja
OS Lifecycle Management	WhiteSource Software	Nein	Nein	Ja	Ja[d,e]	Ja	Ja	Ja	Ja
OS vulnerability assessment tool	SAP	Ja	Ja	Ja	Ja[d,e]	Ja	Nein	Nein	Ja
ossindex-gradle-plugin[a]	Sonatype	Ja	Ja	Ja	Ja[e]	Ja	Nein	Nein	Nein
PHP Security Checker	SensioLabs	Ja	Ja	Ja	Ja[c]	Nein	Nein	Nein	Nein
Retire.js	RetireJS Project	Ja	Ja	Ja	Ja[c]	Nein	Nein	Nein	Teilweise[g]
SafeNuGet	OWASP	Ja	Ja	Ja	Ja[c]	Nein	Ja	Nein	Nein
Safety	pyup	Ja	Ja	Ja	Ja[c]	Nein	Nein	Ja	Nein
Snyk	Snyk	Nein	Nein	Ja	Ja[d]	Ja	Ja	Ja	Ja
SourceClear	Veracode	Nein	Nein	Nein	Ja[c,d,e]	Ja	Nein	Ja	Ja
Xray	JFRog	Nein	Nein	Ja	Ja[d]	Ja	Ja	Ja	Ja
Maven plugin for OSS Index[a]	Sonatype	Ja	Ja	Ja	Ja[e]	Ja	Nein	Nein	Nein

[a] Client für Sonatype OSS Index [b] nur via GitHub [c] CLI [d] via Plugin für Buildserver [e] Gradle und Maven Plugin
[f] Docker Container [g] Analyse von JS Code im Browser



DEPENDENCY-CHECK

Dependency-Check is an open source tool performing a best effort analysis of 3rd party dependencies; false positives and false negatives may exist in the analysis performed by the tool. Use of the tool and the reporting provided constitutes acceptance for use in an AS IS condition, and there are NO warranties, implied or otherwise, with regard to the analysis or its use. Any use of the tool and the reporting provided is at the user's risk. In no event shall the copyright holder or OWASP be held liable for any damages whatsoever arising out of or in connection with the use of this tool, the analysis performed, or the resulting report.

How to read the report | Suppressing false positives | Getting Help: github issues

Project: Attendance Table

ag.em.plugin:attendanceTableNew:2.5.8-SNAPSHOT

Scan Information (show all):

- *dependency-check version*: 5.0.0-M1
- *Report Generated On*: Feb 28, 2019 at 10:06:55 +01:00
- *Dependencies Scanned*: 353 (301 unique)
- *Vulnerable Dependencies*: 31
- *Vulnerabilities Found*: 108
- *Vulnerabilities Suppressed*: 0
- ...

Display: Showing Vulnerable Dependencies (click to show all)

Dependency	Vulnerability IDs	Package	Highest Severity	CVE Count	Confidence	Evidence Count
confluence-6.6.0.jar	cpe:2.3:a:atlassian:confluence:6.6.0:*:*:*:*:*:*:*	pkg:maven/com.atlassian.confluence /confluence@6.6.0	MEDIUM	3	Highest	24
crowd-core-2.10.3.jar	cpe:2.3:a:atlassian:crowd:2.10.3:*:*:*:*:*:*:*	pkg:maven/com.atlassian.crowd/crowd-core@2.10.3	HIGH	2	Highest	26
spring-beans-3.2.10.RELEASE.jar	cpe:2.3:a:pivotal:spring_framework:3.2.10:*:*:*:*:*:*:* cpe:2.3:a:pivotal_software:spring_framework:3.2.10:*:*:*:*:*:*:* cpe:2.3:a:springsource:spring_framework:3.2.10:*:*:*:*:*:*:*	pkg:maven/org.springframework/spring-beans@3.2.10.RELEASE	CRITICAL	10	Highest	27
spring-security-core-3.2.5.RELEASE.jar	cpe:2.3:a:pivotal_software:spring_security:3.2.5:*:*:*:*:*:*:* cpe:2.3:a:vmware:springsource_spring_security:3.2.5:*:*:*:*:*:*:*	pkg:maven/org.springframework.security /spring-security-core@3.2.5.RELEASE	HIGH	2	Highest	32
atlassian-embedded-crowd-atlassian-user-2.3.3.jar	cpe:2.3:a:atlassian:crowd:2.3.3:*:*:*:*:*:*:*	pkg:maven/com.atlassian.crowd/atlassian-embedded-crowd-atlassian-user@2.3.3	CRITICAL	5	Highest	27
embedded-crowd-core-2.10.3.jar	cpe:2.3:a:atlassian:crowd:2.10.3:*:*:*:*:*:*:*	pkg:maven/com.atlassian.crowd/embedded-crowd-core@2.10.3	HIGH	2	Highest	28
spring-core-4.3.4.RELEASE.jar	cpe:2.3:a:pivotal_software:spring_framework:4.3.4:*:*:*:*:*:*:*	pkg:maven/org.springframework/spring-core@4.3.4.RELEASE	CRITICAL	10	Highest	29
applinks-spi-5.4.1.jar	cpe:2.3:a:atlassian:application_links:5.4.1:*:*:*:*:*:*:*	pkg:maven/com.atlassian.applinks/applinks-spi@5.4.1	HIGH	3	Low	28
spring-dao-2.0.6.jar	cpe:2.3:a:pivotal:spring_framework:2.0.6:*:*:*:*:*:*:* cpe:2.3:a:pivotal_software:spring_framework:2.0.6:*:*:*:*:*:*:*	pkg:maven/org.springframework/spring-dao@2.0.6	CRITICAL	10	Low	28

Abbildung A.1: Dependency-Check Report

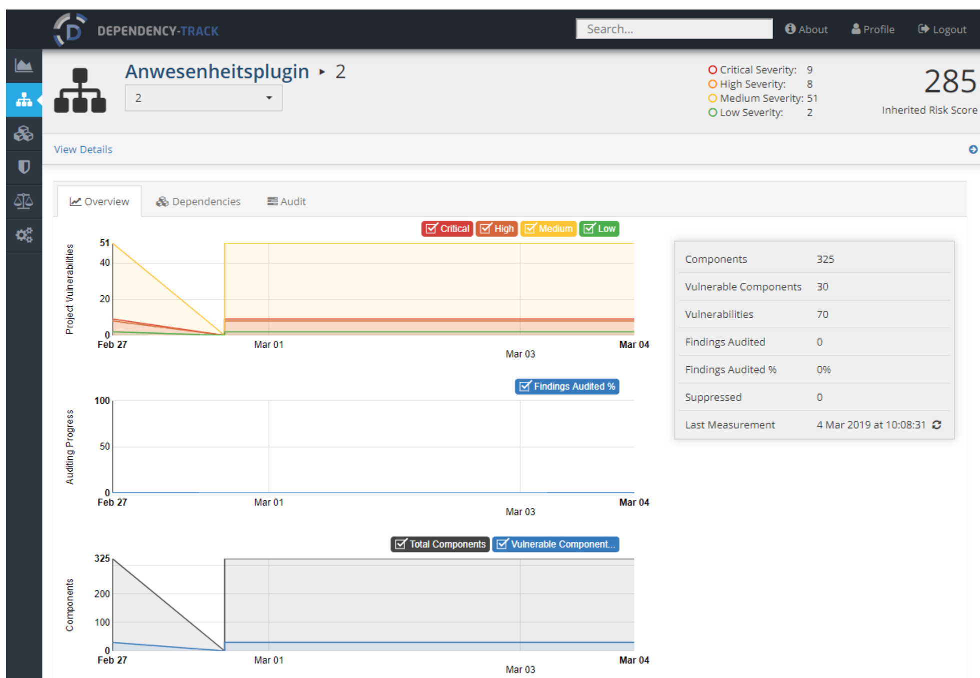


Abbildung A.2: Dependency-Track Projektansicht

```
+-----------------------------------------------------------------------------------------------+
| Severity   | Tool       | Identifier                                                          |
+-----------------------------------------------------------------------------------------------+
| Unknown    | Gemnasium  | CVE-2017-7957                                                       |
|                                                                                               |
| Denial of Service in com.thoughtworks.xstream/xstream                                         |
| Solution: Upgrade to the latest version                                                       |
| In pom.xml                                                                                    |
+-----------------------------------------------------------------------------------------------+
| Unknown    | Gemnasium  | CVE-2013-7285                                                       |
|                                                                                               |
| Remote code execution due to insecure XML deserialization in                                  |
| com.thoughtworks.xstream/xstream                                                              |
| Solution: Upgrade to the latest version                                                       |
| In pom.xml                                                                                    |
+-----------------------------------------------------------------------------------------------+
| Unknown    | Gemnasium  | CVE-2016-1000031                                                    |
|                                                                                               |
| Remote Code Execution in commons-fileupload/commons-fileupload                                |
| Solution: Upgrade to latest version.                                                          |
| In pom.xml                                                                                    |
+-----------------------------------------------------------------------------------------------+
| Unknown    | Gemnasium  | CVE-2016-9879                                                       |
|                                                                                               |
| Encoded "/" in path variables in org.springframework.security/spring-security-core           |
| Solution: Upgrade to the latest version                                                       |
| In pom.xml                                                                                    |
+-----------------------------------------------------------------------------------------------+
```

Abbildung A.3: Ausgabe des GitLab Dependency Scanners

LITERATUR

[1] Mohamed Aymen Saied, Ali Ouni, Houari Sahraoui, Raula Gaikovina Kula, Katsuro Inoue und David Lo. "Improving reusability of software libraries through usage pattern mining". In: *Journal of Systems and Software* 145 (2018), S. 164–179. ISSN: 01641212. DOI: 10.1016/j.jss.2018.08.032. URL: https://ink.library.smu.edu.sg/cgi/viewcontent.cgi?article=5306&context=sis_research (besucht am 05. 03. 2019).

[2] Matthias Rohr. *Sicherheit von Webanwendungen in der Praxis*. Wiesbaden: Springer Fachmedien Wiesbaden, 2018. ISBN: 978-3-658-20144-9. DOI: 10.1007/978-3-658-20145-6. URL: https://link.springer.com/content/pdf/10.1007%2F978-3-658-20145-6.pdf (besucht am 14. 02. 2019).

[3] Lars Heinemann. "Effective and Efficient Reuse with Software Libraries". Dissertation. München: Technische Universität München, 2012. URL: https://www.cqse.eu/publications/2012-effective-and-efficient-reuse-with-software-libraries.pdf (besucht am 05. 03. 2019).

[4] Andrew van der Stock u. a. *OWASP Top 10 - 2017 (Deutsche Version 1.0)*. 2017. URL: https://www.owasp.org/images/9/90/OWASP_Top_10-2017_de_V1.0.pdf (besucht am 13. 02. 2019).

[5] Joachim Baumann, Ronald Hötschl und Bastian Spanneberg. *Continuous Delivery: Ein Überblick*. Heidelberg: dpunkt.verlag, 2013. (Besucht am 10. 01. 2019).

[6] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas. *Prinzipien hinter dem Agilen Manifest*. 2016. URL: http://agilemanifesto.org/iso/de/principles.html (besucht am 13. 02. 2019).

[7] Inc. FIRST.Org. "CVSS v3.0 Specification". In: (2015). URL: https://www.first.org/cvss/cvss-v30-specification-v1.8.pdf (besucht am 04. 03. 2019).

[8] Jeff Williams und Arshan Dabirsiaghi. *The Unfortunate Reality of Insecure Libraries*. Hrsg. von Contrast Security. Columbia, 2014. URL: https://cdn2.hubspot.net/hub/203759/file-1100864196-pdf/docs/Contrast_-_Insecure_Libraries_2014.pdf?t=1501279222788 (besucht am 05. 03. 2019).

[9] Guy Podjarny. *Mitigating known security risks in open source libraries*. 2018. URL: https://www.oreilly.com/ideas/mitigating-known-security-risks-in-open-source-libraries (besucht am 11. 02. 2019).

[10] Risk Based Security. *2018 VulnerabilityTrend*. Hrsg. von Risk Based Security. 2019. URL: https://pages.riskbasedsecurity.com/2018-ye-vulnerability-quickview-report (besucht am 05. 03. 2019).

[11] Guy Podjarny. *Integrating continuous testing for improved open source security*. 2018. URL: https://www.oreilly.com/ideas/integrating-continuous-testing-for-improved-open-source-security (besucht am 17. 02. 2019).

[12] Jeremy Long. *Depending on Vulnerable Libraries*. 2018. URL: https://jeremylong.github.io/DependencyCheck/general/dependency-check.pdf (besucht am 02. 03. 2019).

[13] Johannes Schnatterer. "Automatisierte Überprüfung von Sicherheitslücken in Abhängigkeiten von Java Projekten". In: *Java aktuell* 2017.01 (2017). (Besucht am 02. 03. 2019).

[14] Jeremy Long. *Dependency-Check Documentation - File Type Analyzers*. URL: https://jeremylong.github.io/DependencyCheck/analyzers/index.html (besucht am 02. 03. 2019).

[15] *Common Platform Enumeration (CPE) - Security Content Automation Protocol | CSRC*. 2016. URL: https://csrc.nist.gov/projects/security-content-automation-protocol/scap-specifications/cpe (besucht am 02. 03. 2019).

[16] Steve Springett. *Dependency-Track Documentation - Terminology*. 2019. URL: https://docs.dependencytrack.org/terminology/ (besucht am 02. 03. 2019).

[17] Steve Springett. *Dependency-Track Documentation - Ecosystem Overview*. 2019. URL: https://docs.dependencytrack.org/integrations/ecosystem/ (besucht am 02. 03. 2019).

[18] *GitLab Documentation - Dependency Scanning*. URL: https://docs.gitlab.com/ee/user/project/merge_requests/dependency_scanning.html (besucht am 02. 03. 2019).

[19] Olivier Gonzalez. *GitLab.org - Dependency Scanning*. URL: https://gitlab.com/gitlab-org/security-products/dependency-scanning/tree/master (besucht am 02. 03. 2019).