**Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering**

Suche und Korrektur von Schwachstellen in Software-Bibliotheken

Search and Fix of Vulnerabilities inside Software Libraries

Bachelorarbeit

im Studiengang Informatik

von

Tim Anders

**Prüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Prof. Joel Greenyer
Betreuer: M. Sc. Fabien Patrick Viertel**

Hannover, 13.03.2019

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 13.03.2019

____

Tim Anders

Zusammenfassung

Sicherheitslücken in Software-Bibliotheken können, je nach Einsatzort und schwere, beträchtliche Schäden verursachen. Regelmäßig werden Schwachstellen in Software-Bibliotheken entdeckt und durch Updates behoben, welche aber nicht automatisch heruntergeladen werden und damit oftmals eine eigentlich geschlossene Schwachstelle noch immer angreifbar ist.
Im Rahmen dieser Bachelorarbeit wurde ein Programm entwickelt welches durch einen Monitor gestartet wird sobald eine Schwachstelle auftritt und für die Bibliothek in der diese liegt prüft ob es eine neuere Version gibt welche, sofern sie verfügbar ist. Im Anschluss wird diese heruntergeladen um die Schwachstelle in der veraltete Bibliothek zu sichern. Als Quelle der neuen Bibliotheken wird das Maven Repository genutzt, welches zum Zeitpunkt der Erstellung dieser Arbeit 13,62 Millionen Bibliotheken enthält und das Standart Repository der Software Apache Maven, eines der meist genutzten Build-Management-Tools weltweit ist.

Abstract

Security vulnerabilities in software-libraries can cause, depending on point of use and severity, significant damage. Searching and fixing of such vulnerabilities happen on a regular basis but manual replacing of vulnerable library versions can very time consuming for the developers.
This bachelor thesis is about the development of a tool that checks for a vulnerable library if there is a newer version available, downloads it, and replaces the code of the vulnerable function with the newer, safer one. The library will still be useable, so that any reconfiguration of the project will be unneccessary, saving much time and effort for the developers. Also this tool creates two text files that contain the older, vulnerable code and the newer, safer code. The source for retrieving the newer versions of the libraries is the Maven central repository, which contains at the time roughly 13,62 millions of libraries and uses Apache Maven, one of the most used build-management tools worldwide.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Problemstellung 1
1.2 Lösungsansatz 2
1.3 Struktur der Arbeit 2

2 Grundlagen **4**
2.1 Maven Repository 4
2.2 Call Graph 6
2.3 Common Vulnerabilities and Exposures 6
2.4 National Vulnerability Database 6
2.5 Monitor 7
2.6 Vulnerability Checker 7

3 Verwandte Arbeiten **8**

4 Konzept und Planung **9**
4.1 Ziele 9
4.2 Anforderungen 9
4.3 Programmablauf 11
4.3.1 Lokale Suche 11
4.3.2 Download 11
4.3.3 Call Graph 12
4.3.4 Dateisuche, Extraktion & Methodensuche 12
4.3.5 Beheben der Schwachstelle 12
4.3.6 Datenspeicherung 13

5 Implementierung **15**
5.1 Datenbanken 15
5.2 Initialisierung 16
5.3 Lokale Suche 17
5.4 Download 18
5.5 Call Graph 20
5.6 Datei Suche 21
5.7 Extraktion & Methodensuche 21
5.8 Beheben der Schwachstelle 21
5.9 Datenspeicherung 22

6 Evaluation **25**
6.1 Testfälle und Ergebnisse 25

6.1.1 Erstellung eines Tests 25
6.2 Ergebnisse & Interpretation 27

7 Zusammenfassung **30**
7.1 Risiken für die Verwendbarkeit 30
7.2 Ausblick 31

8 Anhang **32**
8.1 DVD Inhalte und Erklärungen 32

1 Einleitung

In der heutigen Zeit wird man von verschiedenster Software in fast allen Bereichen des täglichen Lebens unterstützt. Soziale Netzwerke, Smartphones, Onlinebanking und vieles mehr sind kaum noch aus unseren Leben weg zu denken und bestimmen die Art wie wir Leben und miteinander kommunizieren. Doch leider wächst mit der Zunahme von Software und deren Komplexität auch die Anzahl der Sicherheitslücken [3]. Dabei sind es oftmals die gleichen Fehler und Angriffsvektoren welche Software angreifbar machen. Die *OWASP Top Ten* [2] sind eine Liste der Zehn meist auftretenden Sicherheitslücken in Web-Applikationen. Auf Platz 9 befindet sich *Using components with known vulnerabilities* [5], welches das Kernproblem dieser Bachelorarbeit darstellt. Genauer heißt es:

„Komponenten wie Bibliotheken, Frameworks etc. werden mit den Berechtigungen der zugehörigen Anwendung ausgeführt. Wird eine verwundbare Komponente ausgenutzt, kann ein solcher Angriff von Datenverlusten bis hin zu einer Übernahme des Systems führen. Applikationen und APIs,die Komponenten mit bekannten Schwachstellen einsetzen, können Schutzmaßnahmen unterlaufen und so Angriffe mit schwerwiegenden Auswirkungen verursachen.“[5]

Denn auch wenn neue, und damit sicherere, Versionen veröffentlicht werden, scheinen Entwickler diese nicht zu nutzen. Es ist also ein Programm erforderlich welches eine unsichere Bibliothek erkennt, auf eine neue Version prüft, und Anschließend sichert.

1.1 Problemstellung

Damit ein solches Programm funktioniert ist es nötig zu erkennen ob in einer Bibliothek eine Schwachstelle enthalten ist. Es ist außerdem nötig zu wissen welche Methode genau die Schwachstelle enthält um diese zu ersetzen, dazu werden die Parameter welche diese Funktion erhält benötigt um sie eindeutig ausfindig zu machen. Sobald bekannt ist welche Bibliothek die Schwachstelle enthält muss nach einer neuen Version gesucht und, falls verfügbar, heruntergeladen werden zum beheben der Schwachstelle. Nach Abschluss dieses Vorgangs sollte sichergestellt sein dass die Bibliothek noch immer lauffähig ist um eine Neukonfiguration des zugehörigen Programms zu vermeiden. Es gibt zwar Monitore, welche zur Laufzeit eines Programms dieses beobachten und Anomalien in ihren Aktionen registrieren [11], diese bemerken aber besagte Anomalien nur, sie tragen nicht direkt zur Behebung eines Sicherheitsproblems bei.

1.2 Lösungsansatz

Um diese Probleme zu lösen wurde der Library Checker (LC) entwickelt, welcher von einem Monitor einen Programmaufruf einer Bibliothek erhält, welche vermutlich eine Schwachstelle beinhaltet, um eben diese zu beheben. Die Form eines Beispielaufrufs ist in Abbildung 4.2 zu sehen. Durch solch einen Aufruf ist es möglich die Bibliothek zu bestimmen welche die Schwachstelle enthält. Sobald diese ausfindig gemacht wurde kann im Anschluss ein Aufruf an das Maven Central Repository[1] (siehe Kapitel 2.1) erfolgen, welches als Quelle für den download von weiteren Bibliotheken dient. Falls im Repository eine neue Version zu der betroffenen Bibliothek verfügbar ist wird diese heruntergeladen und derart verarbeitet dass die alte, fehlerhafte Version der Methode durch die neue, sicherere Version ersetzt wird. Eine Darstellung des ersten Konzepts vom Library Checker ist in Abbildung 1.1 zu sehen.

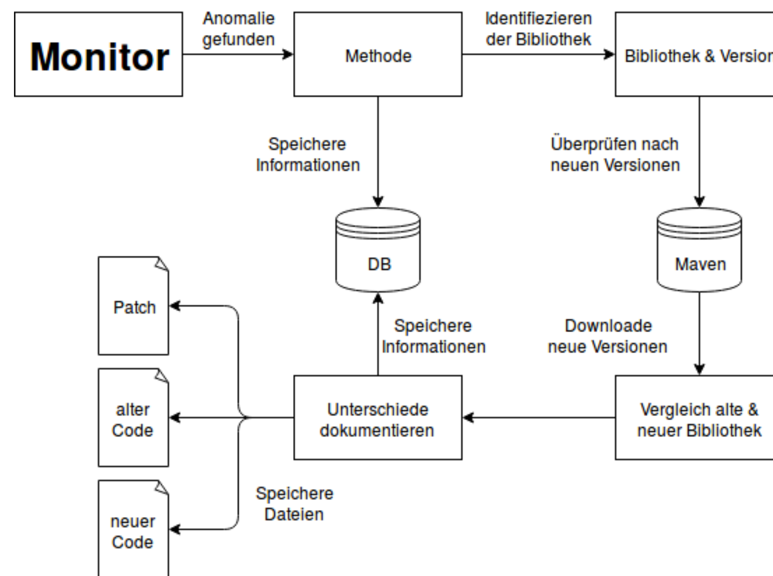

Abbildung 1.1: Eerstes Konzept des Library Checkers

1.3 Struktur der Arbeit

Die Arbeit ist wie folgt Strukturiert: In Kapitel 1 wird die Problemstellung und der dazugehörige Lösungsansatz beschrieben. In Kapitel 2 werden Grundlagen erläutert die einem besseren Verständnis der Arbeit dienen. In Kapitel 3 wird auf verwandte Arbeiten hingewiesen, den derzeitigen Stand, und wie sich diese Arbeit von ihnen abgrenzt. In Kapitel 4 werden genauer die Planung und Konzepte der Arbeit erläutert, dabei werden die Aufgaben des Programms in einzelne Abschnitte geteilt und diese erklärt.

[1] *https://mvnrepository.com/repos/central*

Kapitel 5 erläutert den Programmfluss und die Implementierung der einzelnen Komponenten des Programms. Kapitel 6 behandelt die Ergebnisse des Programms und deren Bewertung. Anschließend folgt in Kapitel 7 eine Zusammenfassung der Ergebnisse, Ideen zur Optimierung und ein Ausblick auf zukünftiges.

2 Grundlagen

In diesem Kapitel werden die nötigsten Grundlagen erläutert die einem besseren Verständnis dieser Arbeit und der Erstellung des Library Checkers dienen. Diese umfassen das Maven Repository, eine öffentlich verfügbare Sammlung von Software-Bibliotheken, das Schwachstellen-Klassifizierungs-Schema CVE, der National Vulnerability Database, dem Monitor, und dem Vulnerability Checker, einem Programm welches als Ergebnis einer Masterarbeit am Institut für Praktische Informatik der Leibniz Universität Hannover hervorging.

2.1 Maven Repository

Das Maven Repository ist eine seit 2004 bestehende Sammlung von Softwarebibliotheken, welche zur Zeit annähernd 14 Millionen Bibliotheken umfasst. Diese können durch das weit verbreitete *Apache Maven Build-Management Tool* mit äußerst geringem Aufwand in ein Projekt integriert und genutzt werden.

Es dient als Online-Sammlung von Softwarebibliotheken welche verschiedensten Programmiersprachen oder sogar Plattformen angehören können. Das Repository unterteilt sich ebenfalls in kleinere Repositories, oftmals von namhaften Firmengruppen wie Google, Sonatype oder Atlassian. In dieser Arbeit wird jedoch ausschließlich auf die Nutzung für das größte, das Maven Central Repository eingegangen, welches standardmäßig als Hauptquelle des Apache Maven Build-Management Tools dient. Dieses wächst seit 2004 stetig. Im Jahr 2006 hat sich die Anzahl der Projekte sogar fast verdoppelt und jährlich steigt die Anzahl an Artefakten in diesem Repository um durchschnittlich 43% [1]. Das gesamte Repository enthält dabei 3,53 Millionen indexierte *.jar*-Dateien, welche sich in bereits kompilierte und unkompilierte Dateien unterteilen lassen. Im Zuge dieser Arbeit wurde das Repository gescannt um die Anzahl und Verteilung der Bibliotheken zu ermitteln. Unter diesen waren 1067170 bereits kompilierte und 525070 unkompilierte jar-Bibliotheken enthält. Zusätzlich sind auch noch 490210 JavaDoc Dateien für die Bibliotheken verfügbar, die Aufteilung wurde in Abbildung 2.1 dargestellt. Das Verhältnis, zu wie vielen kompilierten Bibliotheken unkompilierte vorhanden sind, beträgt fast 33% (Siehe Abbildung: 2.2). Diese Information ist für die Entwicklung des Programms entscheidend, da für eine möglichst flächendeckende Sicherung von Bibliotheken eine hohe Zahl von kompilierten und unkompilierten Versionen vorliegen muss. Es wurde zudem auch gezählt wie viele Bibliotheken in einer Ordnerstruktur im Maven Central Repository enthalten sind. So liegt der Anteil von Bibliotheken in den Ordnern *com* (Commercial) und *org* (Organisation) bei etwa 82%. Was darauf schließen lässt dass ein Großteil der angebotenen Bibliotheken von größeren, namhaften Firmen und Organisationen zur Verfügung gestellt wird. Was für diese

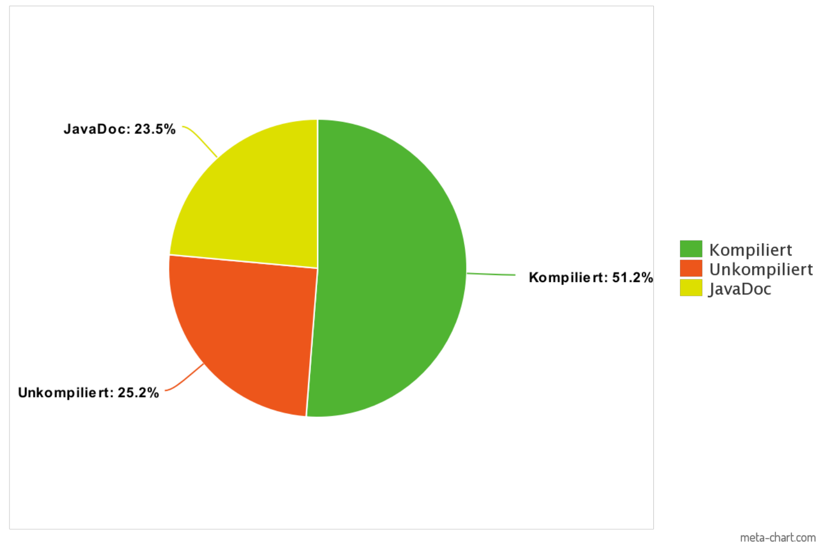

Abbildung 2.1: Prozentuale Aufteilung des Maven Scans

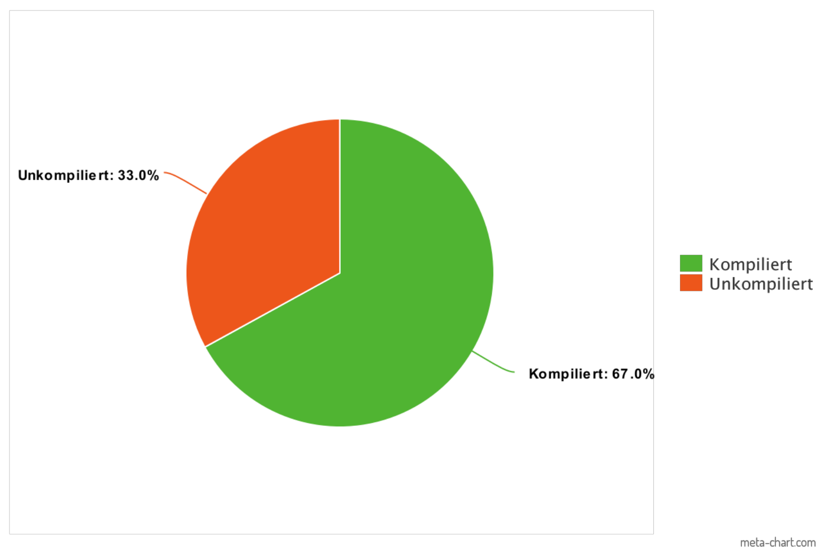

Abbildung 2.2: Verhältnis von Kompilierten zu Unkompilierten Jar-Dateien

Bibliotheken eine erhöhte Qualität von Code und Dokumentation bedeuten könnte.

2.2 Call Graph

Ein Call Graph stellt die Aufrufe von Methoden in einem Programm dar. Er kann unter anderem genutzt werden um die Abhängigkeiten zwischen Methoden zu ermitteln und alle Ober- oder Unteraufrufe einer Methode zu bestimmen. Ein Beispiel für einen Call Graph ist in Abbildung 4.3 zu sehen.

2.3 Common Vulnerabilities and Exposures

Der Industriestandart *Common Vulnerabilities and Exposures* (CVE) [13] wurde im Jahr 1999 von MITRE[1], einer Amerikanischen Organisation zum Betrieb von Forschungsinstituten, gegründet und ist eine einheitliche Namenskonvention für Schwachstellen in Software. Dabei werden diese durch eine laufende Nummer (die *CVE-ID*) eindeutig Kennbar gemacht. Seit 2014 sieht die Form einer CVE-ID wie folgt aus:

CVE-2018-10086

Diese Beispiel ID definiert die Schwachstelle mit der Nummer 10086 welche im Jahr 2016 aufgenommen wurde, vor 2014 wurden lediglich 4-stellige laufende Zahlen verwendet, dies musste allerdings geändert werden, da die Anzahl der gefundenen Schwachstellen das Limit von 10.000 übertroffen hat. Durch diese eindeutige Kennung ist ein vereinfachter Austausch von Informationen zwischen dritten möglich. Meist wird zu einer Schwachstelle auch ein sogenannter *Common Vulnerability Scoring System*-Score (CVSS) [4] berechnet, welcher durch eine ganze Zahl von 0 bis 10 die schwere der Schwachstelle angibt.

2.4 National Vulnerability Database

Die *National Vulnerability Database*[2] (NVD) ist eine seit 2005 bestehende Datenbank zur Speicherung und Analyse von Schwachstellen welche vom *National Institute of Standarts and Technology*[3] (NIST) gegründet wurde und unter anderem vom *Amerikanischen Department of Homeland Security*[4] gesponsert wird [6]. Sie wird von der offiziellen CVE Liste gefüllt und bietet weitere Möglichkeiten zur Suche von Schwachstellen durch Angabe von beispielsweise Betriebssystem, Produktnamen, Versionsnummer und mehr [7].

[1] *https://www.mitre.org/*
[2] *https://nvd.nist.gov/*
[3] *https://www.nist.gov/*
[4] *https://www.dhs.gov/*

2.5 Monitor

Ein *runtime execution monitor*, also ein Monitor der ein Programm zur Laufzeit überprüft, analysiert das Verhalten von Programmen. Sollte in einem Programm eine Anomalie auftreten könnte es sein dass bösartiger Code diese ausgelöst hat, was auf eine Schwachstelle in der Programmierung hindeuten kann [14]. Die genaue Funktionsweise ist sehr komplex und wird in dieser Arbeit nicht näher behandelt.

2.6 Vulnerability Checker

Der *Vulnerability Checker* ist ein konfigurierbarer Schwachstellenprüfer, welcher als Ergebnis einer Masterarbeit von Leif Erik Wagner im Jahr 2017 am Institut für praktische Informatik der Gottfried Wilhelm Leibniz Universität entstanden ist. Dieser ist als Eclipse Plug-In direkt in die IDE integriert und überprüft die NVD auf ein Vorhandensein der Bibliotheken welche in einem Projekt genutzt werden. Wichtig bei der Überprüfung einer Bibliothek sind die Metadaten welche in der zugehörigen Manifest Datei stehen, diese werden benötigt um die zugehörige CPE[5] und damit die Schwachstelle, welcher der CPE zugeordnet wird zu ermitteln [8]. Wenn Sicherheitslücken gvorhanden sind welche dieselbe CPE aufweisen, werden diese von der Datenbank heruntergeladen. Sollte in einer Bibliothek eine Schwachstelle vorliegen, so wird dem Entwickler durch das farbliche markieren der Bibliothek im *Package Explorer* auf das vorhanden sein und die schwere der Schwachstelle hingewiesen (Siehe Abbildung 2.3). Darüber hinaus kann der Vulnerability Checker mit externen SQL-Datenbanken

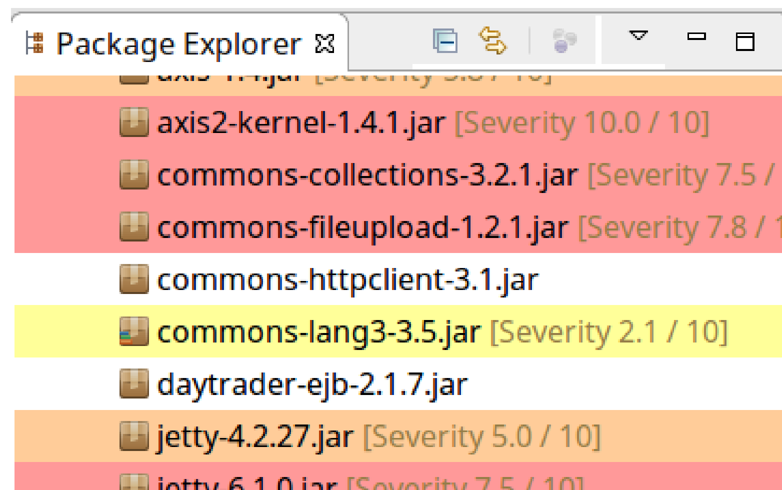

Abbildung 2.3: Package Explorer in Eclipse mit aktivem Vulnerability Checker

erweitert werden, sodass er nicht nur von den Informationen der NVD abhängig ist.

[5] *https://nmap.org/book/output-formats-cpe.html*

3 Verwandte Arbeiten

Die Arbeit *The Maven repository dataset of metrics, changes, and dependencies* [9] der Autoren Steven Raemaekers, Arie van Deursen, Joost Visser ist eine Arbeit welche vom eigens erstellten *Maven Dependency Dataset*, und dessen Erstellung handelt. Diese behandelt auch um Themen wie *Call Graphs*, Vererbung und andere Zusammenhänge. Auch wenn das Datenset die Analyse von 148,253 jar Dateien beinhaltet und aus dem Jahr 2013 stammt, ist sie jedoch zu alt um die Relevanz eines Programms, welches ausschließlich jar Dateien verarbeiten kann, zu untermauern.

Die Arbeit *The Bug Catalog of the Maven Ecosystem* [10] behandelt die statische Fehleranalyse mit dem Tool *FindBugs*[1] von etwa 265 Gigabyte jar Dateien aus dem Maven Central Repository. Das erstellen des Datensets wurde durch 2 Prozesse erreicht. Dem *data cleaning*, in welchem, nachdem alle Dateien geladen wurden, nach Java Dateien gefiltert werden, da FindBugs ausschließlich mit Java Bytecode funktioniert. Und dem Prozess *FindBugs Results Collection*, in dem die Java Bytecode Dateien durch FindBugs verarbeitet und im Anschluss die Ergebnisse akkumuliert wurden um Aussagekräftige Statistiken zu erhalten. Diese enthalten nicht nur prozentuale Angaben der Bug Kategorien, sondern auch die Korrelationen dieser zueinander.

Das Paper *Are Third-Party Libraries Secure? A Software Library Checker for Java* [15] von Fabien Patrick Viertel, Fabian Kortum und Kurt Schneider befasst sich mit der Entwicklung eines Eclipse-Tools das direkt in der IDE integriert ist und die Entwickler während der Entwicklung direkt unterstützt. Dieses Eclipse Tool baut auf der Basis des Vulnerability Checkers von Leif Erik Wagner auf, dessen Funktionsweise im späteren Verlauf der Arbeit genauer erläutert wird.

[1] *http://findbugs.sourceforge.net/*

4 Konzept und Planung

In diesem Kapitel werden die Ziele und Anforderungen, sowie Grundlegende Konzepte und verschiedene Methoden zur Lösung der einzelnen Abschnitte des *Library Checkers* erläutert. Der Programmablauf lässt sich grob in folgende Teilbereiche einordnen: Lokale Suche, Download, erstellen des Call Graphen, Datei Suche, Extraktion & Methodensuche sowie Beheben der Schwachstelle und die Datenspeicherung. Eine vereinfachte Darstellung des Schemas wird in Abbildung 4.1 noch einmal verdeutlicht.

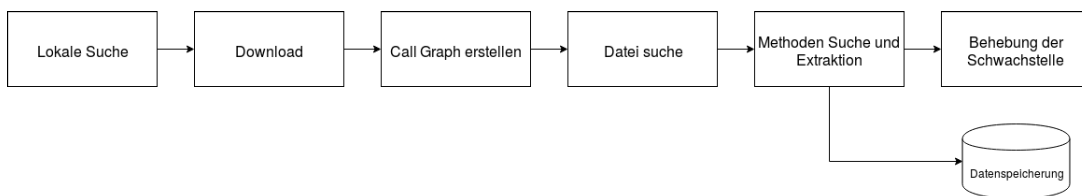

Abbildung 4.1: Abstrahierter Flowgraph des Library Checkers

4.1 Ziele

Der Library Checker hat Primär als Ziel fehlerhafte Bibliotheken, durch das Ersetzen von schwachstellenbehafteten Methoden, sicherer zu machen. Dabei sollen Zwei Dateien erstellt werden, wobei eine den alten, fehlerhaften Programmcode und die andere den neuen, sicheren Code enthält. Durch diese können beispielsweise fehlerhafte Methoden für die *Code Clone Detection*[1] verwendet werden. Des weiteren soll eine Datenbank erstellt werden, die zu jeder fehlerhaften Bibliothek Informationen zum Zeitstempel, CVE-ID (falls vorhanden), Methodenname, Funktionsname und weitere Informationen speichert.

4.2 Anforderungen

Die Ziele des Library Checkers lassen sich auf folgende Anforderungen zusammenfassen:

[1] *https://de.wikipedia.org/wiki/Quelltextklon*

Beheben der Schwachstelle ohne beeinträchtigung der funktionalität

Der Library Checker muss in der Lage sein die Methode, welche die Schwachstelle enthält, durch eine aktuellere, sicherere Methode zu ersetzen, ohne dabei die Funktionalität der Bibliothek zu beeinträchtigen.

Anlegen der Datenbank

Das Programm soll eine Datenbank anlegen, die mindestens folgende Informationen zu der vom Monitor gefundenen Schwachstelle enthält:

- Zeitstempel des Monitors
  Der Zeitpunkt an dem der Monitor einen Befund geliefert hat.
- Zeitstempel der Behebung der Schwachstelle
  Der Zeitpunkt an dem die Methode das Ersetzen der schwachstellenbehafteten Bibliothek durchführt. Falls noch keine neue Bibliothek und damit noch keine Möglichkeit zur Behebung der Schwachstelle vorliegt wird ein *n/A* eingetragen.
- Bibliotheksname
  Der Name der betroffenen Bibliothek. Ist nicht gleichzusetzen mit dem Attribut des *Implementation-Product* in der Manifest Datei der Bibliothek, da diese oftmals nicht vorhanden ist.
- Bibliotheksversion
  Die Version der betroffenen Bibliothek. Auch hier ist sie nicht mit den Informationen aus der Manifest Datei gleichzusetzen.
- Methodenname
  Der Name der Methode die dem Monitor aufgefallen ist.
- CVE-ID
  Diese Spalte wird in der Datenbank eine Sammlung von möglichen CVE-IDs anlegen. Da es möglich ist das ein und dieselbe Bibliothek mehrere CVE-IDs hat.

Anlegen des fehlerhaften Codes

Das Programm muss eine Datei erstellen in welcher der fehlerhafte Code gespeichert ist.

Anlegen des Patch Codes

Das Programm muss eine Datei erstellen in welcher der aktualisierte Code gespeichert ist.

4.3 Programmablauf

Im folgenden werden die einzelnen Phasen des Programmablaufs, welche in Abbildung 4.1 dargestellt wurde, erläutert. Dabei wird lediglich auf das grundlegende Konzept zur Lösung eines Problems eingegangen, die genaue Implementierung dieser Phasen ist im Kapitel 5 nachzulesen.

4.3.1 Lokale Suche

In der ersten Phase des Programms, der lokalen Suche, muss das Programm den genauen Speicherpfad der fehlerhaften Bibliothek ausfindig machen. Über einen Monitor, der zur Laufzeit Auffälligkeiten in der Funktionsweise einer Bibliothek bemerkt die auf eine mögliche Schwachstelle hindeuten könnten, wird der Library Checker gestartet. Diesem wird ein Funktionsaufruf übergeben, der alle nötigen Informationen enthält um die gesuchte Bibliothek ausfindig zu machen. So ein Funktionsaufruf kann aussehen wie in Abbildung 4.2 dargestellt. Anhand dieser Informationen kann das Programm

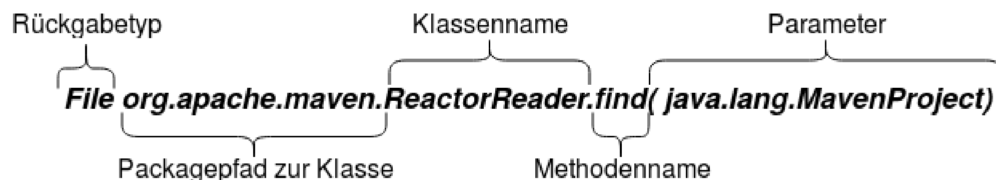

Abbildung 4.2: Programmaufruf des Monitors

nun die vorhandenen Bibliotheken nach der Klasse durchsuchen welche die beschriebene Methode enthält. Im folgenden ermittelt das Programm den Namen, die Version, den Hersteller und andere Informationen der Bibliothek um diese im weiteren Verlauf verarbeiten zu können.

4.3.2 Download

In der Download Phase baut das Programm eine Verbindung zum Maven Central Repository auf. In dieser überprüft es dieses auf die Existenz von neuen Versionen der Bibliothek, sowie dessen Inhalt. Da es wichtig ist zu wissen welche Dateitypen in dieser Datei enthalten sind, da dass das Programm sowohl kompilierte sowie unkompilierte jar-Dateien einer Bibliothek für einen reibungslosen Ablauf der Bearbeitung benötigt (mehr dazu in der genaueren Implementierung ab Kapitel 5). Es werden im Anschluss alle nötigen Dateien heruntergeladen und temporär gespeichert.

4.3.3 Call Graph

In dieser Phase muss das Programm einen *Call Graph* erstellen der die Methodenaufrufe in einer Bibliothek untereinander darstellt und es dadurch ermöglicht zu überprüfen welche Methoden von der vermeintlich fehlerhaften Methode aufgerufen werden. Dies ist wichtig, da der Monitor eine Methode *loginUser* als fehlerhaft ansehen kann, aber tatsächlich ist die Methode *checkPassword* fehlerhaft. Dabei ist zu beachten das sowohl Ober- als auch Unteraufrufe einer Methode im Call Graph mit aufgenommen werden, was bedeutet das auf eine Bildung von Zyklen oder rekursiven Aufrufen geachtet werden muss. Es sollte zudem möglich sein die Erstellung eines solchen Call Graphs zu unterbinden, weil sie, vor allem in großen Bibliotheken sehr zeitaufwändig ist. Abbildung 4.3 zeigt einen Beispielauszug eines Call Graphs des Programms.

```
M:org.apache.tomcat.util.net.openssl.OpenSSLEngine:renegotiate() (O)org.apache.tomcat.util.net.openssl.OpenSSLEngine:clearLastError()
M:org.apache.tomcat.util.net.openssl.OpenSSLEngine:renegotiate() (S)org.apache.tomcat.jni.SSL:renegotiate(long)
M:org.apache.tomcat.util.net.openssl.OpenSSLEngine:renegotiate() (O)org.apache.tomcat.util.net.openssl.OpenSSLEngine:checkLastError()
M:org.apache.tomcat.util.net.openssl.OpenSSLEngine:renegotiate() (S)org.apache.tomcat.jni.SSL:getHandshakeCount(long)
M:org.apache.tomcat.util.net.openssl.OpenSSLEngine:renegotiate() (S)org.apache.tomcat.jni.SSL:doHandshake(long)
M:org.apache.tomcat.util.net.openssl.OpenSSLEngine:renegotiate() (O)org.apache.tomcat.util.net.openssl.OpenSSLEngine:checkLastError()
M:org.apache.tomcat.util.net.openssl.OpenSSLEngine:checkLastError() (S)org.apache.tomcat.jni.SSL:getLastErrorNumber()
M:org.apache.tomcat.util.net.openssl.OpenSSLEngine:checkLastError() (S)org.apache.tomcat.jni.SSL:getErrorString(long)
M:org.apache.tomcat.util.net.openssl.OpenSSLEngine:checkLastError() (I)org.apache.juli.logging.Log:isDebugEnabled()
M:org.apache.tomcat.util.net.openssl.OpenSSLEngine:checkLastError() (S)java.lang.Long:toString(long)
M:org.apache.tomcat.util.net.openssl.OpenSSLEngine:checkLastError() (M)org.apache.tomcat.util.res.StringManager:getString(java.lang.String,java.lang.Object[])
M:org.apache.tomcat.util.net.openssl.OpenSSLEngine:checkLastError() (I)org.apache.juli.logging.Log:debug(java.lang.Object)
M:org.apache.tomcat.util.net.openssl.OpenSSLEngine:checkLastError() (O)javax.net.ssl.SSLException:<init>(java.lang.String)
M:org.apache.tomcat.util.net.openssl.OpenSSLEngine:clearLastError() (S)org.apache.tomcat.jni.SSL:getLastErrorNumber()
M:org.apache.tomcat.util.net.openssl.OpenSSLEngine:memoryAddress(java.nio.ByteBuffer) (S)org.apache.tomcat.jni.Buffer:address(java.nio.ByteBuffer)
M:org.apache.tomcat.util.net.openssl.OpenSSLEngine:getHandshakeStatus() (S)org.apache.tomcat.jni.SSL:pendingWrittenBytesInBIO(long)
```

Abbildung 4.3: Beispiel eines Call Graphs

4.3.4 Dateisuche, Extraktion & Methodensuche

In der Phase der Extraktion & Methodensuche wird die Bibliothek welche die Schwachstelle enthält zunächst entpackt. Hier muss unterschieden werden ob es sich um eine jar-Datei handelt die nur *class*-Dateien enthält[2]. Sollte es sich um bereits kompilierten Code handeln muss dieser zunächst durch einen Java Dekompilierer[3] dekompiliert werden um ihn für unsere Zwecke brauchbar zu machen. Wenn die Dateien nun alle in Quellcode vorliegen, kann das Programm damit beginnen, in dem vom Monitor übergebenen Packagepfad nach der Datei zu suchen, welche die gegebene Methode enthält. Diese muss im Anschluss für weitere Zwecke aus beiden Dateien, der alten und neuen Version, extrahiert werden.

4.3.5 Beheben der Schwachstelle

Das Beheben der Schwachstelle ist eine der Phasen mit der das Programm seine Hauptfunktionalität erfüllt. Zu diesem Zeitpunkt ist nun alles bekannt was nötig ist um

[2] *Da im Maven Repository oftmals kompilierter und unkompilierter Code enthalten ist werden beide, sofern sie verfügbar sind, in der Download Phase heruntergeladen*

[3] *http://www.javadecompilers.com/*

den Code der fehlerhaften Methode in der unsicheren Bibliothek durch den der sichereren Methode aus der neuen zu ersetzen. Nach Abschluss dieser Phase ist die Bibliothek in Projekten die diese nutzen genauso einsatzfähig wie vorher, allerdings wurde die fehlerhafte Methode automatisch durch eine sicherere ersetzt.

4.3.6 Datenspeicherung

In dieser Phase legt das Programm in einer Datenbank einen Eintrag mit allen im Programmfluss gesammelten Informationen an. Dabei wird es auf eine Datenbank mit gespeicherten CVE-IDs zugreifen um zu überprüfen ob die gefundene Schwachstelle bereits bekannt ist und gegebenenfalls darauf verweist. Die Datenbank auf welche das Programm zugreift ist in diesem Fall dieselbe die vom Vulnerability Checker erstellt wird. Sie enthält unter anderem die Tabelle *configuration_items* die in Abbildung 4.4 dargestellt ist: Diese Tabelle bietet die Möglichkeit über Informationen in der Manifest

configuration_items	
cve_id	TEXT
cpe_uri	TEXT
part	TEXT
vendor	TEXT
product	TEXT
version	TEXT
include_previous_versions	INTEGER
operation_id	INTEGER

Abbildung 4.4: Tabelle configuration_items des Vulnerability Checker [8]

Datei einer Bibliothek einige CVE-IDs zu bestimmen, falls welche verfügbar ist. Genauere Informationen zur Implementierung sind im Abschnitt 5.1 gegeben.
Nachdem auf diese Datenbank zugegriffen wurde kann nun eine eigene Datenbank mit all den gesammelten Informationen gefüllt werden. Wie diese schematisch aussieht ist in Abbildung 4.5 zu sehen. Über das genaue Vorgehen der Implementierung ist im folgenden Kapitel mehr zu lesen.

LibraryCheckerTable
FIRST_TIMESTAMP: TEXT
PATCH:TIMESTAMP: TEXT
LIBRARY_NAME: TEXT
LIBRARY_VERSION: TEXT
CLASS_NAME: TEXT
METHOD_NAME: TEXT
CVE_ID: TEXT
VULNERABILITY_PATH: TEXT
PATCH_PATH: TEXT

Abbildung 4.5: Aufbau der LibraryChecker Tabelle

5 Implementierung

In diesem Kapitel wird im Detail auf die Implementierung der Funktionen eingegangen, insbesondere auf die Gründe warum sie in einer bestimmten Weise implementiert wurde. Dabei wird an den Phasen des vorhergehenden Kapitels orientiert. Abbildung 4.1 zeigt einen stark vereinfachten Ablauf des Programms. Im folgenden wird bei den einzelnen Phasen der Implementierung jeweils ein detaillierterer Ablaufplan gegeben.

5.1 Datenbanken

Das Programm nutzt zwei Datenbanken. Eine um die Informationen, welche im Verlauf des Programms gesammelt wurden, zu speichern und eine um die Suche nach vorhandenen CVE-IDs zu ermöglichen. Letztere kann durch eine Änderung in der Konfigurationsdatei jederzeit ausgetauscht werden. Es wurde sich bei der Implementierung für eine SQLite Datenbank[1] entschieden, da diese einfach zu implementieren ist und vom Vulnerability Checker ebenfalls genutzt wird. Die Datenbank des Library Checkers umfasst lediglich nur eine Tabelle (vgl. Abbildung 4.5), dessen Spalten sich an den Anforderungen aus 4.2 orientieren. Um Bibliotheken auf bereits bestehende CVE-IDs zu überprüfen wurde eine Abfrage an die Datenbank, genauer den *configurations_table* (Siehe Abbildung 4.4), des Vulnerability Checkers realisiert. In dieser werden über die Informationen aus der Manifest Datei einer Bibliothek alle CVE-IDs geladen welche eventuell für diese infrage kommen. Dabei werden lediglich drei Informationen aus der Manifest-Datei gebraucht.

Implementation_Product
: Das Produkt, also die Bibliothek an sich.

Implementation_Vendor
: Der Entwickler/Verbreiter der Bibliothek.

Implementation_Version
: Die Versionsnummer der Bibliothek.

Darüber hinaus wird noch eine Vierte Information, der *Implementation-Title* genutzt. Dieser wird auf die gleiche Weise wie das Implementation-Product verwendet falls dieser nicht verfügbar ist. Diese beiden Begriffe scheinen oftmals Synonym in der Manifest-Datei genutzt zu werden. Je mehr dieser Informationen in der Manifest-Datei

[1] *https://sqlite.org/index.html*

vorhanden sind desto genauer wird die spätere CVE-ID Suche einer Bibliothek. Sollte keine der Informationen zugänglich sein wird keine CVE-ID Suche in der Vulnerability Checker Datenbank durchgeführt, weil die Menge der Ergebnisse zu umfangreich wäre. In einem späteren Abschnitt in diesem Kapitel wird die genaue Suche nach diesen Informationen und die anschließende Datenspeicherung genauer erläutert.

5.2 Initialisierung

Bei Programmstart müssen zunächst einige Konstanten geladen werden. Dazu gehören unter anderem Einstellungen und Pfadkonstanten. Die Konfigurationsdatei besteht aus Key-Value paaren, die Zeilenweise durchlaufen werden um die Werte zu laden. Die Konfigurationsdatei besteht aus folgenden Werten:

Local_Repository_Path
Der Pfad in dem alle Dateien die aus dem Programm hervorgehen abgelegt werden. Wird relativ zur ausführenden jar-Datei angelegt.

Database_Name
Der Name der Library Checker Datenbank in der alle gesammelten Informationen zu einer Schwachstelle gespeichert werden.

Database_Path
Der Pfad zur Library Checker Datenbank.

Monitored_Folder
Die Ordnerstruktur in der die betroffenen Bibliotheken sich befinden müssen. Kann der Projektordner eines Programms sein. Wird standardmäßig auf den Ordner *temporaryMonitoredJars* im Local_Repository_Path gesetzt.

Create_Call_Graph
Ein Wahrheitswert ob der Call Graph erstellt werden soll für eine Methode oder ob nicht noch nach weiteren Methodenaufrufen gesucht werden soll. Wird standardmäßig auf *false* gesetzt.

NVD_Database_Location
Der Pfad an dem die Datenbank aus der die vorhandenen CVE-IDs geladen werden sollen gespeichert ist.

NVD_Database_Name
Der Name der Datenbank aus der die CVEs zum Vergleich geladen werden.

Manifest_Genauigkeit

Die Genauigkeitsstufe welche die Anzahl der benötigten Informationen aus der Manifest Datei für die Suche nach CVE-IDs angibt. Ist standardmäßig auf 2 gesetzt.

Aus diesen Informationen werden zudem auch noch Ordnerkonstanten für weitere Unterordner erstellt, die im Local_Repository_Ordner angelegt werden und zur eindeutigen Aufteilung des späteren Outputs des Programms dienen, um eine leichtere Navigation des Nutzers ermöglichen. Alle nötigen Ordner werden dabei bei Ausführung des Programms von selbst erstellt falls sie noch nicht vorhanden sein sollten. Die erstellten Ordner und ihre Funktion lautet wie folgt:

Database
Der Ordner an dem die Library Checker Datenbank angelegt wird.

PatchFiles
Der Ordner an dem alle Methodenblöcke, welche eine gefundene Schwachstelle beheben, abgespeichert werden.

VulnerabilityFiles
Der Ordner an dem alle Methodenblöcke gespeichert werden, welche durch den Monitor als Schwachstelle markiert wurden.

Zudem wird beim während der Laufzeit des Programms ein Ordner erstellt welcher temporär alle Downloads und entpackte Dateien enthält. Dieser wird allerdings bei Beendigung des Programms automatisch gelöscht.

5.3 Lokale Suche

Nachdem alle Konstanten gesetzt sind kann nun damit angefangen werden die eigentlichen Funktionalitäten umzusetzen. Die lokale Suche, also die Suche nach der fehlerhaften Bibliothek im Projektordner, ist in 4 Unteraufgaben zu unterteilen die in Abbildung 5.1 dargestellt sind.
Das Programm erhält beim Start von einem Monitor einen Methodenaufruf, die eine

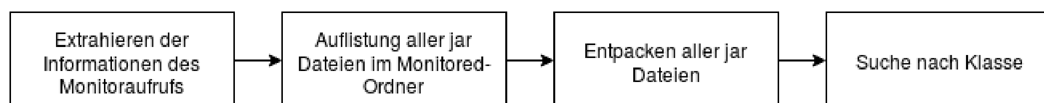

Abbildung 5.1: Prozess der lokalen Suche

eventuelle Schwachstelle darstellen könnte. Um im weiteren Verlauf des Programms alle nötigen Informationen zur Hand zu haben müssen diese aus dem Methodenaufruf extrahiert werden. Ein Beispiel eines solchen Methodenaufrufs ist in Abbildung 4.2

dargestellt. Das extrahieren der Informationen wird im Programm durch triviale String-Umformung realisiert und ebenfalls in Konstanten gespeichert.

Nachdem extrahiert wurde was genau gesucht wird ist es an der Zeit zu wissen in welcher Bibliothek sich die gesuchte Klasse, und damit die zugehörige Methode, befindet. Da dem Programm in der Konfigurationsdatei eine Ordnerstruktur vorgegeben ist in der sich die vom Monitor überwachten Bibliotheken befinden, kann in dieser danach gesucht werden. Dabei durchläuft das Programm diese Ordnerstruktur rekursiv und sammelt in einer Liste alle Dateien welche eine bestimmte Endung aufweisen. Da der Library Checker vorerst nur für Java Bibliotheken konzipiert wurde, wird ausschließlich nach Dateien mit der Endungen *".jar"* gesucht.

Sobald das Programm eine Liste mit allen möglichen jar-Dateien welche der Monitor beobachtet hat, müssen diese anschließend auf den Inhalt geprüft werden. Dabei wird unterschieden ob es sich bei dem Inhalt um unkompilierte Java Dateien mit der Endung *.java*, welche Quellcode in Textdateien enthalten handelt, oder ob es sich um kompilierte Dateien mit der Endung *.class*, welche kompilierten Bytecode enthalten, handelt. Diese Unterscheidung der Dateien ist durch eine Überprüfung der Endung leicht festzustellen. Sollte es sich um eine Datei mit Quellcode handeln kann dieser direkt nach der Methode, welche die Schwachstelle enthält, durchsucht werden. Sollte es sich bei der Datei allerdings um Bytecode handeln, muss dieser zunächst dekompiliert werden.

Das dekompilieren von class-Dateien erfolgt durch das Tool *Class File Reader* (CFR)[2] von Lee Benfield, weil dieses auch die neuesten Versionen von Java unterstützt, was unabdingbar für die Sicherstellung der Funktionsweise bei neueren Bibliotheken in der Zukunft ist.

Da das Programm nun zu jeder Bibliothek und den enthaltenen Klassen die Quelltexte hat, können diese nun nach dem Package und der Methode durchsucht werden. Dabei werden die Dateien in einem Scanner einzeln eingelesen und nach der Methode im passenden Package durchsucht. Die Suche liefert so die passende Klasse in einer Bibliothek welche im folgenden auf Updates überprüft und weiter verarbeitet werden können.

5.4 Download

Die Download-Phase des Programms ist in 2 Phasen zu unterteilen. Die erste ist das durchsuchen des Maven Central Repositorys nach dem Bibliotheksnamen welcher in der vorherigen Phase ermittelt wurde. Das Repository ist hierarchisch aufgebaut. Anhand dieser Eigenschaft ist es möglich sich eine URL so zu erstellen dass der Bereich, welcher nach der Bibliothek durchsucht werden muss, drastisch eingegrenzt wird. Das Format für die URL einer Bibliothek sieht Beispielsweise wie folgt aus: Dies wäre die Form der URL die zum Download der Maven-core Bibliothek, mit der Versionsnummer 3.6.0, aus dem Maven Projekt der Apache Software Foundation führt. Da das Programm alle erforderlichen Informationen bereits verfügbar hat, kann es nun

[2] *http://www.benf.org/other/cfr/*

Abbildung 5.2: Aufbau einer Maven Repository URL

nach der neuesten Version suchen. Diese wird der Einfachheit halber über den Quelltext der URL *http://central.maven.org/maven2/org/apache/maven/maven-core/* ermittelt. Der Quelltext eines solchen Links sieht wie folgt aus:

```
21 <a href="../">../</a>
22 <a href="0.1/" title="0.1/">0.1/</a>                                    2015-05-19 01:15      -
23 <a href="0.1.2/" title="0.1.2/">0.1.2/</a>                                  2015-05-20 22:14      -
24 <a href="0.2.0/" title="0.2.0/">0.2.0/</a>                                  2015-08-27 00:39      -
25 <a href="0.3.0/" title="0.3.0/">0.3.0/</a>                                  2015-10-27 21:08      -
26 <a href="0.4.1/" title="0.4.1/">0.4.1/</a>                                  2015-12-12 00:57      -
27 <a href="0.5.0/" title="0.5.0/">0.5.0/</a>                                  2016-11-04 14:42      -
28 <a href="0.5.1/" title="0.5.1/">0.5.1/</a>                                  2017-07-25 17:59      -
29 <a href="0.5.2/" title="0.5.2/">0.5.2/</a>                                  2018-02-09 22:49      -
30 <a href="0.8.0/" title="0.8.0/">0.8.0/</a>                                  2018-08-20 18:43      -
31 <a href="0.8.1/" title="0.8.1/">0.8.1/</a>                                  2018-10-08 20:34      -
32 <a href="maven-metadata.xml" title="maven-metadata.xml">maven-metadata.xml</a>          2018-10-08 20:34      598
33 <a href="maven-metadata.xml.md5" title="maven-metadata.xml.md5">maven-metadata.xml.md5</a>      2018-10-08 20:34      32
34 <a href="maven-metadata.xml.sha1" title="maven-metadata.xml.sha1">maven-metadata.xml.sha1</a>     2018-10-08 20:34      40
```

Da auch hier die Bibliotheken in Ordnern hierarchisch sortiert vorliegen durchläuft das Programm den Quelltext nach der letzten Zeile welche ein Ordner ist, dieser wird zwangsläufig zum Speicherort der neuesten Bibliothek führen. Anhand des Namens dieses Ordners kann das Programm die neueste Versionsnummer bestimmen.

Dem Programm sind nun alle nötigen Informationen der aktuellen, fehlerhaften, Bibliothek bekannt, sowie die neueste Versionsnummer. Es kann in der zweiten Phase des Downloads nun damit anfangen die Bibliotheken herunterzuladen. Dabei lädt es drei verschiedene Varianten herunter. Zum einen lädt das Programm die aktuell vorhandene Bibliothek im Quelltext Format, damit im späteren Verlauf diese editiert werden kann, heruntergeladen, da der Aufruf vom Monitor zur Laufzeit kommt, und es eventuell nicht möglich ist auf den Ordner der aktuell verwendeten Bibliothek zuzugreifen während sie genutzt wird. Zum anderen wird sowohl die kompilierte, als auch die Quelltext Variante der neuesten Bibliothek heruntergeladen. Durch die kompilierte Version wird im weiteren Verlauf der Call Graph erstellt (siehe 5.5). Durch die unkompilierte Variante, welche Quelltext enthält, können die neuen Methoden die die betroffen Methode ersetzen sollen, vereinfacht extrahiert werden, mehr dazu in Abschnitt 5.7.
Zudem ist anzumerken, dass es bereits ein automatisches Download Tool für Maven Artefakte gibt, das Maven Invoker Plugin, dieses hat sich allerdings im laufe dieser Arbeit als unflexibel und zu langsam herausgestellt, da es für jeden Download eine eigene virtuelle Maschine startet. Diese erzeugt eine beträchtliche Verzögerung des Programms. Zudem war es nicht sehr zuverlässig, bestimmte Jar-Dateien im Repository zu finden und an eine gewünschte Stelle (den temporären Ordner) zu speichern.

5.5 Call Graph

Der Call Graph ist eine wichtige Komponente im Programm und dient zur Bestimmung von Methodenaufrufen in der Bibliothek untereinander. Es gibt viele verschiedene Bibliotheken die einen Call Graphen auf unterschiedlichste Weise erstellen, im Zuge dieser Arbeit wurde die Open-Source *java-callgraph*-Bibliothek von Georgios Gousios genutzt[12], welche ausschließlich mit kompilierten Bibliotheken funktioniert. Es wurde diese Bibliothek zum erstellen eines Call Graphen für die Erstellung eines Call Graphen gewählt da sie einen Methodenaufruf in einem einfach zu lesenden String verarbeitet, welche nach folgendem Muster dargestellt.

M:class1:<method1>(arg_types) (typeofcall)class2:<method2>(arg_types)

Eine Abbildung welche einen Call Graph aus einer Bibliothek darstellt ist in 4.3 zu sehen. Anhand dieser einfachen Darstellung eines Methodenaufrufs A an eine Methode B ist es möglich die Liste aller Aufrufe so zu Filtern dass nur die gewünschte Methode und all ihre Aufrufe im folgenden Verlauf bearbeitet werden. Dies ist durch das wiederholte durchlaufen der Liste sichergestellt und wird in Abbildung 5.3 noch einmal verdeutlicht. Am Ende dieser Phase besitzt das Programm eine Liste von Metho-

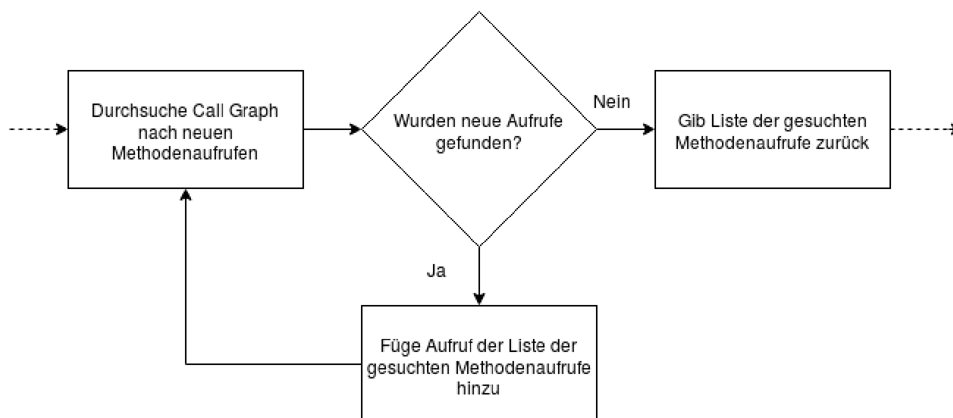

Abbildung 5.3: Durchlaufen des Call Graphs

denaufrufen welche aus der ursprünglich gesuchten Methode hervorgehen. In diesen Methodenaufrufen sind die Namen der Klassen und Methoden enthalten, sowie deren Parameter die für eine eindeutige Zuordnung nötig sind. Die folgenden Phasen der Datei Suche, der Extraktion & Methodensuche und des Behebens der Schwachstelle laufen in einer Schleife für jeden Methodenaufruf eines Call Graphen einzeln ab umm alle Klassen, Methoden und ihre Untermethoden zu betreffen.

5.6 Datei Suche

In dieser Phase werden, durch die Informationen eines Methodenaufruf, die Klassen in der bereits extrahierten fehlerhaften und in der neuen Version der Quelltext Bibliothek gesucht, die bei Start der Phase ebenfalls entpackt wurde, um diese für String-Operationen nutzbar zu machen. Das Durchsuchen nach einer bestimmten Klasse wird auch hier durch rekursives Durchlaufen aller Dateien in der entpackten Ordnerstruktur erreicht. Es kann sich hier nicht mehr auf den durch den Monitoraufruf übergebenen Packagepfad verlassen werden, weil es möglich ist dass die Methode sich im Zuge eines Updates nun in einem anderen Package befindet. Nach Durchlauf der rekursiven Suche sind dem Programm nun die alte und neue Klasse bekannt welche die gesuchte Methode im Quelltext enthalten.

5.7 Extraktion & Methodensuche

Da nun die Klassen, welche die gesuchten Methoden enthalten, bekannt sind und im Quelltext vorliegen, können diese nach der gesuchten Methode durchsucht werden. Die Suche nach dieser in einer Datei wird durch einen zeilenweisen Durchlauf durch einen Scanners realisiert. Sobald eine Zeile sowohl den passenden Methodennamen, den Rückgabetyp und die zugehörigen Parameter enthält, stoppt dieser und beginnt den Start und das Ende des Methodenblocks als Indizes zu ermitteln. Nachdem diese ermittelt wurden wird die Methode durch einen *substring* extrahiert und abgespeichert. Dieser Vorgang wird für beide Klassen, die alte und neue einmal durchlaufen, um die alte, fehlerhafte und die neue, sicherere Version der Methode als Strings verfügbar zu haben.

5.8 Beheben der Schwachstelle

Dem Programm sind jetzt alle nötigen Informationen gegeben, um das eigentliche Sichern einer Bibliothek durchzuführen. Dabei durchläuft das Programm den (eventuell entpackten) Quelltext der vorhandenen Bibliothek abermals. Da nicht anzunehmen ist, dass die Indizes zur Methodenextraktion aus dem vorherigen Schritten auch hier die Methode nochmals eingrenzen, muss auch hier zunächst nach dieser gesucht werden. Dies geschieht auf die gleiche Weise wie zuvor, durch ein zeilenweises Durchlaufen des Quelltextes, bis die Methode mit passenden Parametern gefunden wurde. Sobald diese ermittelt ist wird auch diese extrahiert und vorsorglich für späteren Nutzung gespeichert. Im Anschluss wird aus dem Quelltext die alte Methode entfernt und durch die neuere ersetzt.

5.9 Datenspeicherung

Sind alle Methodenaufrufe des Call Graphen durchlaufen und bearbeitet, kann im Anschluss mit der Speicherung aller gesammelter Informationen in die Datenbank begonnen werden.
Es wurden bis hier folgende Informationen zur Bibliothek gesammelt:

- Zeitstempel wann der Monitor diese Bibliotheksversion erstmals registriert hat.
- Zeitstempel der letzten Bearbeitung (Falls bei erstmaligen vorkommen keine neue Version vorhanden dann wird bei erneutem anschlagen des Monitors geguckt ob es 48h her ist seit das letzte mal geguckt wurde.
- Der Klassenname der fehlerhaften Methode.
- Der Methodenname welcher die Schwachstelle enthält.
- Der Name der Bibliothek welche die Klasse und damit die fehlerhafte Methode enthält.
- Die Versionsnummer der Bibliothek.
- Alter Methoden Block (wird in einem Ordner gespeichert, in der Datenbank steht ein String als Referenz zum Speicherort)
- Neuer Methoden Block (wird ebenfalls in einem Ordner gespeichert, in der Datenbank steht ein String als Referenz zum Speicherort)

Zusätzlich werden jetzt auch noch die Manifest Daten ausgelesen, um eine Zuordnung zu möglichen CVE-IDs zu ermöglichen. Die Daten aus der Manifest Datei welche dafür gebraucht werden lauten:

Implementation_Product
Das Produkt, also die Bibliothek an sich.

Implementation_Vendor
Der Entwickler/Verbreiter der Bibliothek.

Implementation_Version
Die Versionsnummer der Bibliothek.

Falls das Implementation-Product leer sein sollte, wir stattdessen nach dem Implementation-Title gesucht. Je mehr von diesen Informationen verfügbar sind, desto besser, da dadurch die Zuordnung von CVE-IDs genauer ist.
Das Programm führt eine SQL-Abfrage in folgender Abbildung an die Datenbank des Vulnerability Checkers durch um alle möglichen CVE-IDs zu erhalten.

select * from configuration_items where vendor like 'Apache' and version like

'8.0.3' group by operation_id

Dies ist eine Beispielabfrage welche bei einem der Tests für die Bibliothek Apache Tomcat 8.0.3 entstanden ist. Sollte keine dieser Informationen verfügbar sein, wird das Programm mit Standarteinstellungen nicht nach CVE-IDs suchen, da sonst die Anzahl der Ergebnisse zu groß und damit wieder zu unbefriedigend wäre. Die Anzahl lässt sich über die *Genauigkeit* in der Konfigurationsdatei angeben. In dieser sind Werte von 0 bis einschließlich 3 möglich. Standartmäßig ist eine 2 eingestellt, was bedeutet dass mindestens zwei der Informationen gegeben sein müssen, um eine CVE-ID Suche durchzuführen.

Da es sehr hilfreich ist eine Schwachstelle zu einer eventuell bestehenden CVE-ID zu verlinken, wird außerdem eine Verbindung zu einer weiteren Lokalen Datenbank hergestellt. In diesem Fall wird die Datenbank, welche vom Vulnerability Checker erstellt wird genutzt. Diese enthält eine Sammlung von CVE's die es von der NVD heruntergeladen hat. Ein großer Vorteil bei der Nutzung einer weiteren Datenbank ist, dass diese durch eine Änderung in der Konfigurationsdatei schnell ausgetauscht werden kann, was das Programm flexibler macht. Diese zusätzlich verbundene Datenbank wird auf das bereits bekannte Vorkommen der Bibliothek geprüft, ob eine CVE-ID zu gefundener Schwachstelle bereits vorliegt. Sollte dies der Fall sein wird sie ebenfalls in den Datenbankeintrag der Bibliothek aufgenommen.

Am Ende einer Datenspeicherung könnte die Datenbank aussehen wie in Abbildung 5.4. Hierfür wurde die Bibliothek tomcat-coyote-8.0.30.jar genutzt, welche die CVE-ID *CVE-2014-0075* aufweist. Die Spalte der CVE-IDs ist vergrößert in Abbildung 4.2 zu sehen. Wie in dieser zu sehen ist wurde diese CVE-ID (neben anderen möglichen zu dieser Bibliothek) gefunden.

FIRST_TIMESTAMP	PATCH_TIMESTAMP	LIBRARY_NAME	LIBRARY_VERSION ^	METHOD_NAME	CVE_ID	VULNERABILITY_PATH	PATCH_PATH
Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
10-03-2019 19:29:30	10-03-2019 19:29:37	ChunkedInputFilter.java	8.0.30	parseChunkHeader	CVE-2016-0706, CVE-2016-0...	/home/tim/.m2/VulnerabilityC...	/home/tim/.m2/Vulnerabili...

Abbildung 5.4: Erste Speicherung eines Dateneintrags

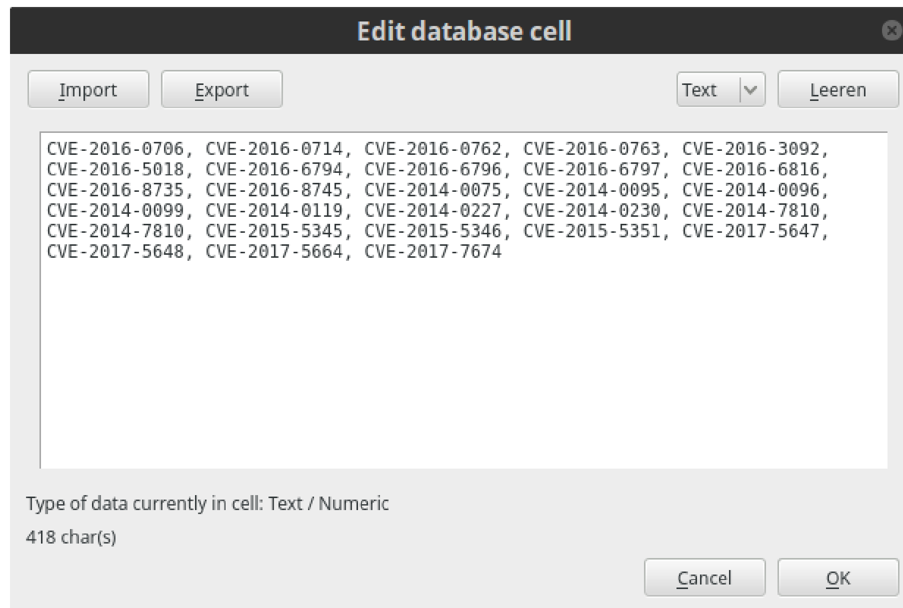

Abbildung 5.5: Mögliche CVE-IDs der Bibliothek tomcat-coyote-8.0.30.jar

6 Evaluation

In diesem Kapitel werden die Einstellungen und Ergebnisse für Tests mit Bibliotheken erklärt und dargestellt. Es wurden sowohl Bibliotheken mit bereits bekannten Schwachstellen und zugehörigen CVE-IDs getestet, als auch Bibliotheken welche keine bekannten Schwachstellen enthalten.

6.1 Testfälle und Ergebnisse

Um das Programm zu testen, wurde ein Testset erstellt, das Bibliotheken mit bereits bekannten Schwachstellen und zugehörigen CVE-IDs enthält. Um diese zu finden, wurde auf der NVD nach Bibliotheken mit vorhandenen CVE-IDs gesucht, die einen Verweis auf die Klasse und Methode geben, weil es sonst aufgrund von mangelnden Informationen nicht möglich ist die Bibliothek korrekt zu testen. Auch sollte im besten Fall bereits ein Patch-Code vorliegen, um das korrekte ersetzen genau zu überprüfen.

Das Testset enthält keine Bibliotheken ohne bereits bekannte Schwachstelle und CVE-ID. Da es ohne genaue Informationen keinen Sinn macht eine beliebige Methode auf Änderungen zu überprüfen und gegeben falls zu ersetzen.
Die Liste der getesteten Bibliotheken und ihre CVE-IDs ist in Abbildung 6.1 einzusehen. Da der Prozess eine Bibliothek zu finden welche bereits eine CVE-ID erhalten hat und genaue Informationen zu Klasse und Methode enthält wird die Erstellung der Tests an einem Beispiel gezeigt, das Prozedere wurde für andere Bibliotheken exakt wiederholt.

6.1.1 Erstellung eines Tests

Für die Erstellung eines Tests für den Library Checker sind folgende Informationen notwendig:

CVE-ID

Die Bibliothek welche getestet werden soll sollte eine CVE-ID vorweisen, der Library Checker ist zwar in der Lage auch ohne CVE-IDs eine Bibliothek zu sichern, allerdings ist es mit einer CVE-ID leichter nachzuvollziehen was genau wo ersetzt wird da oftmals Patches vorhanden sind.

Klassenname

Für die Erstellung eines Tests muss der Monitoraufruf ßimuliert"werden, das

heißt es muss ein String übergeben werden welcher, wie im späteren Einsatzumfeld, alle nötigen Informationen enthält. Der genaue Aufbau eines solchen Aufrufs war in 4.2 zu sehen.

Methodenname
Sehr hilfreich für die Erstellung eines Tests ist eine genaue Angabe zu der Funktion welche die Schwachstelle aufweist. Es ist zwar auch möglich die nötigen Informationen selbst herauszufinden, etwa durch das Vergleichen der alten und neuen Versionen von Methoden, aber dies manuell über den Browser zu ermitteln ist ein sehr zeitintensiver Prozess.

Diese Informationen finden sich alle über die NVD Suche im Browser. Bei Eingabe des Suchbegriffs "*.java*"werden einige Seiten an Ergebnissen zurückgegeben, die sowohl eine CVE-ID, als meist auch den Klassennamen, der Schwachstelle darlegen. Bei einigen Ergebnissen ist sogar die Methode angegeben welche diese enthält.

Allerdings sind nicht alle Ergebnisse für die Erstellung eines Tests relevant. Da das Programm vorläufig dafür konzipiert wurde, sich mit Dateien aus dem Maven Central Repository zu versorgen, muss zunächst überprüft werden ob die Bibliothek eines dieser Ergebnisse auch dort enthalten ist. Auch hier ist die Suche händisch durchzuführen. Es ist zudem zu überprüfen ob es sich bei betroffener Bibliothek überhaupt um eine Java Bibliothek handelt welche einzeln verfügbar ist, da Plugins oder Bibliotheken aus anderen Programmiersprachen sich nicht für das Programm eignen.

Da nicht immer genau zu sehen welche Methode in welcher Klasse die Schwachstelle verursacht, wurden zu Testzwecken für solche Fälle beliebige Methoden gewählt, die sich zwischen den Versionen geändert haben, um eine vergleichbar reale Situation zu schaffen.

Für den Beispieltestfall wurde nach einer CVE-ID gesucht, welche das Schlagwort "*.java*ënthält. Es wurden einige Ergebnisse zurückgegeben und zufällig eins ausgewählt. Die CVE-ID war *CVE-2014-0075*, eine Schwachstelle für die Apache Tomcat 8.0.3 Bibliothek (dessen Datei den Namen "*tomcat-coyote-8.0.30.jar*"hat). Auch die Methode welche die Schwachstelle verursacht hat, ist angegeben. Es handelt sich um die Methode *parseChunkHeader* in der Klasse *ChunkedInputFilter.java*. Diese Klasse kann nun in der Ordnerstruktur ausfindig gemacht werden um den Packagepfad zu ermitteln. Im Anschluss muss lediglich noch der Rückgabetyp heraus gelesen werden um einen Monitoraufruf zu erstellen mit dem das Programm die Bibliothek überprüft. Der Rückgabetyp findet sich unter der genannten Methode in der Datei und ist *boolean*. Der Monitoraufruf würde also wie folgt aussehen:

boolean org.apache.coyote.http11.filters.ChunkedInputFilter.parseChunkHeader()

Dabei wird die wie bereits in Abbildung 5.4 erstellt welche die in Abbildung 5.5 dargestellten CVE-IDs gespeichert hat. Dabei ist zu sehen, dass die CVE-ID tatsächlich gefunden wurde und Teil der Speicherung wurde. Es muss also lediglich noch geprüft werden ob die Methode erfolgreich ersetzt wurde. Ein schneller Vergleich des Methodenblocks der neuesten Bibliothek und der, die gerade bearbeitet wurde, liefert das

Ergebnis dass die Methode erfolgreich ersetzt wurde. Zusätzlich wurde noch überprüft, ob die IDE, in diesem Fall Eclipse, die Bibliothek noch akzeptiert und nutzt, auch dies funktioniert.

6.2 Ergebnisse & Interpretation

Die Vollständige Liste der Testfälle, sowie deren CVE-ID, verwundbare Klasse und die Ergebnisse dazu finden sich in Abbildung 6.1. Es wurden einige Testfälle zur Bewer-

CVE-ID	Bibliothek + Version	Betroffene Klasse	Betroffene Methode
2018-20433	c3p0 0.9.5.2	C3P0ConfigXmlUtils.java	extractXmlConfigFromInputStream
2018-17297	Hutool All 4.2.1	ZipUtil.java	unzip
2014-0075	Apache Tomcat 8.0.30	ChunkedInputFilter.java	parseChunkHeader
2015-0263	Apache Camel core 2.14.0	XmlConverter.java	toSAXSourceFromStream
2018-11248	FileDownloader 1.7.3	FileDownloadUtils.java	ZUFALL
2018-12418	JunRar 1.0.0	Archive.java	ZUFALL
2012-6153	Apache HttpClient 4.2.2	AbstractVerifier.java	ZUFALL

Abbildung 6.1: Testfälle des Library Checkers

tung der Funktionsweise des Programms erstellt. Dabei wurden, wie der Tabelle zu entnehmen ist, verschiedene Bibliotheken, aus verschiedenen Jahren von verschiedenen Herstellern genutzt. Des weiteren wurde die Betroffene Klasse angegeben, da diese in der CVE-ID gegeben war. Die betroffene Methode allerdings wurde bei der Hälfte der Testfälle zufällig gewählt, weil diese meistens nicht angegeben ist.

Die Ergebnisse und Interpretation der einzelnen Tests wird im folgenden Abschnitt zusammengefasst. Dabei ist zu beachten dass alle Testfälle mit der Standartkonfiguration der Konfigurationsdatei durchgeführt wurden.

Testfall 1: CVE-2018-20433

Die Bibliothek mit dem genauen Namen *c3p0-0.9.5.2-sources*
ist fehlerhaft Verlaufen, da keine Dateien aus dem Maven Repository geladen werden konnten. Dies liegt an dem Namen der Bibliothek, dieser wird vom Programm nicht richtig verarbeitet da er nicht den üblichen Konventionen entspricht, dieser enthält nämlich Zahlen im Namen, wobei Zahlen eigentlich nur in der Version verwendet werden dürfen. Dies führt zu einer inkorrekten Verarbeitung im Programm und damit zu falschen Links im Maven Repository.

Testfall 2: CVE-2018-17297

In diesem Test kann die jar-Datei nicht erfolgreich entpackt werden. Es ist möglich dass der Fehler in der Bibliothek liegt, da dieser Testfall der einzige ist indem dies passiert.

Testfall 3: CVE-2014-007

Auch dieser Test verlief erfolgreich. Wie schon in Abbildung 5.4 und 5.5 zu sehen ist liefert dieser Test sogar eine Anzahl von CVE-IDs, wobei die gesuchte CVE-ID, CVE-2014-0075 vorhanden ist.
Dazu wurde die fehlerhafte Methode auch korrekt ersetzt. Bei einem Anschließenden Vergleich der Methode, welche sich in der Bibliothek nach dem Test befindet, und jener welche in der neuesten Version im Repository zu finden ist, befinden sich laut *https://www.diffchecker.com/diff* keine Unterschiede.
Sowohl die fehlerhafte Version der Methode, als auch die sichere, wurden in eigenen Textdateien in der Verzeichnisstruktur des Library Checkers hinterlegt.

Testfall 4: CVE-2015-0263

Dieser Test schlug fehl. Es kann nicht die neue Version der Methode in einer Datei gespeichert werden, weil die neue Version der Datei die Quelltext enthält nicht gefunden werde konnte. Es wird allerdings der Quelltext der alten Version extrahiert und gespeichert. Außerdem wird ein Eintrag in der Datenbank erstellt, welcher die bisher gesammelten Informationen enthält.

Testfall 5: CVE-2018-11248

In diesem Test wurde eine zufällige Methode als vermeintlich fehlerhaft gewählt. Das Programm hat allerdings auch hier keine Sicherung erreichen können. Da auch hier eine fehlerhafte Benennung der Bibliothek der Grund ist. Anstelle des Namens FileDownloader, wie das Projekt im Maven Repository heißt, wurde diese Bibliothek mit *library-1.7.3-sources* betitelt. Was auch hier entgegen der Konvention ist und deshalb unmöglich im Repository zu finden ist. Aus diesem Grunde ist es dem Programm unmöglich eine Sicherung vorzunehmen.

Testfall 6: CVE-2018-12418

Das Programm kann den Namen der Bibliothek nicht verarbeiten. Es konstruiert einen Link zum Maven Repository um nach der neuesten Version zu crawlen, dieser ist aber fehlerhaft. Dies liegt an der Benennung der Bibliothek, da diese sich nicht an die Konventionen von Maven hält.

Testfall 7: CVE-2012-6153

In diesem Testfall werden zwar die Bibliotheken heruntergeladen, allerdings fehlt in der neuesten Version die gesuchte Methode. Dies führt dazu dass das Programm lediglich den alten Code ausliest und speichert. Es wird zudem auch ein Eintrag in der Datenbank erstellt, dieser ist aber nur spärlich mit Informationen befüllt, da diese fehlen.

Auch wenn nur 2 Tests vollständig erfolgreich, und 3 teilweise erfolgreich, verliefen ist dies ein Aussagekräftiges Ergebnis. Da diese von größeren Firmen wie HuTool oder Apache stammen. Im Abschnitt 2.1 wird auf die Ergebnisse vom Crawler eingegangen, welche Aussagen, dass das Repository einen Prozentanteil von über 80% der Bibliotheken im Kommerziellen und Organisationssektor hat. Da Bibliotheken, welche

dort liegen, mit hoher Wahrscheinlichkeit von professionellen Programmierern erstellt wurden, ist davon auszugehen dass sie sich an die Konventionen von Maven halten, und das Programm so für einen Großteil der Bibliotheken funktionieren könnte. Da die Fehler in den Testfällen durch eine zweimalige falsche Benennung der Grund für ein vorzeitigen Abbruch des Programms war.

Interessant wären Ergebnisse eines größeren Testsets, da das diese Vermutung bestätigen oder widerlegen könnte. Das Erstellen eines solchen Testsets ist sehr aufwändig, da, wie schon am Anfang des Kapitels erwähnt, die Suche nach passenden CVE-IDs für diesen Zweck mit einigen Schwierigkeiten verbunden ist.

7 Zusammenfassung

Das Ziel, die Entwicklung eines Programms, welches schwachstellenbehaftete Bibliotheken durch einen Monitoraufruf ausfindig macht und die fehlerhaften Methoden automatisch behebt wurde umgesetzt. Das Programm ist in der Lage sich mit einem Online Repository zu verbinden um auf eine große Sammlung Bibliotheken zuzugreifen. Zudem ist das Programm in der Lage eine Datenbank anzulegen und diese, mit allen Informationen zu einem Befund, aktuell zu halten. Dabei greift es auf eine CVE Sammlung der NVD zu und verhilft somit einer einfachen Zuordnung von Bibliotheken zu bekannten Schwachstellen.

Während der Entwicklung des Programms wurden viele Design Entscheidungen getroffen, so zum Beispiel wurde ein einfacher Downloader entworfen welcher die URL des Maven Central Repositorys rekursiv auf der Suche nach der Bibliothek durchläuft und diese dann herunterlädt. Im Gegensatz zum Nutzen des bereits vorhandenen Plugins, dem Maven Invoker, da dieser langsamer und unflexibler ist.

Die Testfälle des Programms haben gezeigt, dass die Einhaltung von Konventionen eine wichtige Rolle bei automatisierten Abläufen wie der Suche nach Bibliotheken oder deren Versionen. Auch sollten Entwickler bei der Erstellung einer jar-Datei eine Manifest Datei nicht nur obligatorisch anlegen, sondern auch zu einem gewissen Grad ausfüllen, da diese viele wichtige Informationen zur Bibliothek enthalten kann.

Zum Abschluss ist zu sagen das ich glaube dass solche Programme, sowie Security Monitore und Code Clone Detection, ein Standard in IDE's wie Eclipse werden können und sollten, da sie die Entwicklung nachhaltig unterstützen und allgemein zu einer sichereren IT-Infrastruktur beitragen die Millionen von Menschen betrifft.

7.1 Risiken für die Verwendbarkeit

Das Programm verlässt sich zu einem nicht zu vernachlässigbaren Anteil auf Konventionen. So müssen die Bibliotheken welche im Maven Central Repository gespeichert sind, den von Maven genannten Speicherkonventionen einhalten. Das bedeutet eine klare und einfache Namensgebung welche folgende Form haben könnte: *apache-tomcat-8.0.0-sources.jar*. Sollte eine Bibliothek namentlich davon Abweichen ist es unwahrscheinlich dass das Programm sie korrekt verarbeiten wird. Diese Entscheidung über die Bibliotheksnamen zu gehen ist allerdings auch wichtig, da sich auf die Daten in der Manifest Datei zu verlassen auch keine Abhilfe schafft. Diese sind auch in einigen Fällen nicht korrekt ausgefüllt, was sich auch in der späteren Suche nach bestehenden CVE-IDs bemerkbar macht.

Diese Risiken können allerdings für größere, namhaftere Bibliotheken und Firmen zum Teil ignoriert werden, da diese sich zum größten Teil an die von Maven geforderten

Konventionen halten. So ist auch der Scan Vorgang des Maven Central Repositorys, dessen Ergebnisse in Kapitel 2.1 ausführlich erklärt wurde, ein aussagekräftiger Faktor wie das Repository aufgebaut ist und inwiefern das Programm eine Relevanz hat in Bezug auf seine Korrektheit.

7.2 Ausblick

Das Programm kann bereits einen großen Nutzen liefern und bei der Sicherung von namhaften Bibliotheken zum Einsatz kommen. Sollten umfangreichere und flexiblere Versionen dieses Programms entwickelt werden könnte dies zu einer verbesserten Sicherung von Anwendungen führen, da die Nutzung von fehlerhaften Bibliotheken ein weit verbreitetes Problem ist. Das Programm könnte auch noch weitergehend erweitert und verbessert werden wie in den folgenden Punkten aufgeführt.

Es könnten zum einen mehr Repositories als das Maven Central Repository für die Suche eingebunden werden um mehr Bibliotheken zu beziehen. Insbesondere Plattformen wie GitHub könnten genutzt werden um die Reichweite an Bibliotheken stark zu erhöhen. Allerdings ist dazu einzuwenden dass diese oft von Hobby-Entwicklern programmiert wurden und dort in der Regel wenig Wert auf Sicherheit gelegt wird, zudem werden die angebotenen Bibliotheken dort oftmals nicht weiter entwickelt.

Da dieses Programm die schwachstellenbehafteten Methodenblöcke in einer Ordnerstruktur lokal speichert kann dadurch eine Datenbank aufgebaut welche die Code Clone Detection unterstützt.

Das Programm könnte zudem dahingehend modifiziert werden um nicht ausschließlich mit Java Code zu arbeiten. Dadurch könnte eine noch größere Abdeckung von Bibliotheken erfolgen.

Aber es muss sich nicht nur am Programm etwas ändern, wie die Tests gezeigt haben sind die einhalten von Namenskonventionen und die korrekte Nutzung von Manifest-Dateien ein beträchtlicher Faktor für die Korrektheit dieses Programms. Sollten sich mehr Nutzer an diese Konventionen halten so könnte es allen zugute kommen.

8 Anhang

8.1 DVD Inhalte und Erklärungen

Auf der beiliegenden DVD befindet sich:

- Der Library Checker als Projektordner.
- Der Library Checker als ausführbare jar-Datei.
- Die Bibliotheken welche für die Tests genutzt wurden.
- Die Monitoraufrufe der Bibliotheken um die Tests zu starten.
- Die genutzte NVD Datenbank
- LaTeX-Dokumente
- Der Maven Crawler welcher zum Scannen des Maven Central Repository genutzt wurde. (Als Projektordner)
- Der CrawlerScanner welcher eine Textdatei des Maven Crawlers auswertet. (Als Projektordner)
- Ein Handbuch zur Nutzung der Tools.

Literaturverzeichnis

[1] https://javalibs.com/charts/central Zugriff 08.02.2019

[2] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project Zugriff 12.02.2019

[3] https://www.focus.de/digital/computer/internet-forscher-zahl-der-sicherheitsluecken-in-software-steigt_id_4548730.html Zugriff 12.02.2019

[4] https://www.first.org/cvss/ Zugriff 13.02.2019

[5] https://www.owasp.org/images/9/90/OWASP_Top_10-2017_de_V1.0.pdf Deutsche Übersetzung, Zugriff 14.02.2019

[6] https://nvd.nist.gov/general Zugriff 14.02.2019

[7] https://cve.mitre.org/about/cve_and_nvd_relationship.html Zugriff 14.2.2019

[8] Wagner, Leif Erik; *Konzept und Entwicklung eines Schwachstellenprüfers für Java-Bibliotheken*, Institut für Praktische Informatik 2017

[9] https://ieeexplore.ieee.org/document/6624031 Zugriff 14.2.2019

[10] Mitropoulos, Dimitris; Karakoidas, Vassilios; Louridas, Panos; Gousios, Georgios; Spinellis, Diomidis; *The Bug Catalog of the Maven Ecosystem*; Athens University of Economics and Business, Greece; Delft University of Technology, Netherlands

[11] Shiva, S.; Dharam, Ramya; Shilya, Vivek; *Runtime Monitors as Sensors of Security Systems*

[12] https://github.com/gousiosg/java-callgraph java-callgraph Bibliothek; Georgios Gousios; letzter Zugriff 25.2.2019

[13] https://cve.mitre.org/ Zugriff 3.3.2019

[14] Fiskiran A.M; Lee R.B.; IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings

[15] Viertel, Fabien Patrick; Kortum, Fabian; Schneider, Kurt: Are Third-Party Libraries Secure? A Software Library Checker for Java