

# Secure coding through integration of public information security sources to eclipse development environment

**Sivert Lunde**

**Master's thesis in Software Engineering at**

Department of Computer science, Electrical  
engineering and Mathematical sciences,  
Western Norway University of Applied Sciences

Department of Informatics,  
University of Bergen

June 2022



**Western Norway  
University of  
Applied Sciences**



## Abstract

The use of open source components in software development has been growing at a rapid pace for a number of years. This increase in use of open source software is accompanied by an increase in the risk of security vulnerabilities. With an extensive amount of research and time spent towards the development of tools to help mitigate security vulnerabilities in developers' own code, the issue of identifying vulnerabilities in the open source components they use has been rather neglected by comparison. Public security source such as NVD, CVE and CWE already contain an enormous amount of data on both security vulnerabilities in general, as well as specific known instances of vulnerabilities in software. The primary goal of this thesis is to develop a plugin for the Eclipse development environment which seeks to connect developers to these public security sources directly in their IDE. The plugin will specifically be targeted at Maven projects, and will help mitigate potential vulnerabilities by scanning the dependencies of a project and finding any potential vulnerability data for them registered in the NVD. The plugin will be evaluated by utilizing open source dependencies and projects in various tests which seek to identify its performance related to soundness and completeness, as well as runtime performance. The results show a precision of 93%, a recall of 65% and an accuracy of 80%. The runtime performance is shown to be moderate with a linear growth depending on the number of dependencies being scanned. This thesis contributes to research by shedding a light on an under-developed field of software security mitigation and proposes a prototype plugin to help solve the issue.

## **Acknowledgements**

I would like to extend my deeply appreciative gratitude towards my supervisor, Tosin Daniel Oyetoyan. Throughout the project, he has been patient with me regardless of unproductive periods, or times of low motivation. He has kept me focused on the work at hand, adapted to the challenges we have faced along the way, and pushed me when I have needed to be pushed. This project would truly not have been possible without him.

I'd like to thank Kronbar - one of, if not the greatest places on earth. Kronbar, along with its wonderful people, have been an essential part of my social life and the reason I have stayed sane and happy throughout my education.

Finally, my family has earned a moment in the spotlight for their unwavering support for as long as I can remember.

Thank you, all!

# Contents

<b>1</b>	<b>Introduction</b>	<b>8</b>
<b>2</b>	<b>Background</b>	<b>10</b>
2.1	Detecting Security Vulnerabilities . . . . .	10
2.1.1	Development Stage . . . . .	11
2.1.2	Deployment stage . . . . .	11
2.1.3	Maintenance Stage . . . . .	12
2.2	Public Security Sources . . . . .	12
2.2.1	Common Vulnerability Exposures (CVE) . . . . .	12
2.2.2	Common Weakness Enumeration (CWE) . . . . .	13
2.2.3	National Vulnerability Database (NVD) . . . . .	14
2.3	Open Source Software . . . . .	14
2.3.1	Use of Open Source Software . . . . .	15
2.3.2	Open Source Example: Maven Dependency . . . . .	16
2.3.3	Vulnerability in Open Source . . . . .	16
<b>3</b>	<b>Methodology</b>	<b>18</b>
3.1	RQ1: Development of plugin . . . . .	19
3.2	RQ2: Measuring performance of the plugin in terms of complete- ness and soundness using open source dependencies . . . . .	20
3.2.1	Collecting test data . . . . .	20
3.2.2	Testing the plugin . . . . .	21
3.3	RQ3: Measuring performance of the plugin in terms of runtime using open source maven projects . . . . .	23
3.3.1	Collecting test data . . . . .	24
3.3.2	Testing the plugin . . . . .	24
<b>4</b>	<b>Design and Implementation</b>	<b>26</b>
4.1	Architecture . . . . .	26
4.2	Extracting dependencies . . . . .	27
4.3	Retrieving data from NVD . . . . .	28
4.4	Filtering NVD results . . . . .	30
4.5	Marking vulnerable dependencies . . . . .	30
4.6	Viewing results . . . . .	31
4.7	Storage . . . . .	32
4.8	Preferences . . . . .	33
<b>5</b>	<b>Evaluation</b>	<b>34</b>

5.1	RQ2: Measuring performance of the plugin in terms of completeness and soundness using open source dependencies . . . . .	34
5.1.1	Soundness and Completeness . . . . .	34
5.1.2	Version handling . . . . .	36
5.1.3	Summary of results for RQ2 . . . . .	37
5.2	RQ3: Measuring performance of the plugin in terms of runtime using open source maven projects . . . . .	37
5.2.1	Testing impact of project size on runtime . . . . .	38
5.2.2	Testing performance on vulnerable vs non-vulnerable dependencies . . . . .	38
5.2.3	Testing scalability of plugin as number of dependencies grows . . . . .	39
5.2.4	Testing impact of storage solution on runtime . . . . .	40
5.2.5	Summary of results of RQ3 . . . . .	40
5.3	Evaluating plugin requirements . . . . .	41
<b>6</b>	<b>Discussion</b>	<b>42</b>
6.1	Challenges . . . . .	42
6.1.1	Thesis scope . . . . .	42
6.1.2	Eclipse plugin development . . . . .	43
6.2	Limitations . . . . .	43
6.2.1	Recall results . . . . .	43
6.2.2	Lack of testing . . . . .	44
<b>7</b>	<b>Related work</b>	<b>45</b>
7.1	Vulnerability detection and classification . . . . .	45
7.2	Use and impact of information sources . . . . .	46
7.3	Developer assistance during development . . . . .	46
<b>8</b>	<b>Conclusion</b>	<b>48</b>
	<b>Appendices</b>	<b>54</b>
<b>A</b>	<b>Source code</b>	<b>55</b>
<b>B</b>	<b>Plugin documents</b>	<b>56</b>
<b>C</b>	<b>NVD API JSON response schema</b>	<b>57</b>

# List of Figures

2.1	Security Testing Techniques in the Software Development Life Cycle [15]	10
2.2	Registering a new CVE record	13
2.3	The Open Source Definition	15
4.1	Simple data model for the plug-in	27
4.2	Simple sequence chart of scanning process	27
4.3	Example of dependency added to pom.xml	28
4.4	Example showing part of JSON response from NVD API	29
4.5	Example of CPE result from NVD API	30
4.6	Custom Eclipse marker	31
4.7	Marker view in Eclipse	31
4.8	Results tab, dependency list	32
4.9	Results tab, CVE details	32
4.10	Storage structure	33
4.11	Plugin preference page	33
5.1	Line graph of runtimes of various number of dependencies	39
5.2	Bar chart comparing runtimes with and without storage	40

# List of Tables

2.1	Example of a CVE-record . . . . .	13
2.2	Example of CWE-record . . . . .	13
2.3	Advantages of Open Source Software . . . . .	16
3.1	Requirement table for plugin . . . . .	18
3.2	Maven dependencies used for testing . . . . .	22
3.3	Confusion matrix of vulnerability labels . . . . .	23
3.4	Projects used for testing . . . . .	24
4.1	Examples of Eclipse PDE API packages . . . . .	26
5.1	Dependency vulnerability labels . . . . .	35
5.2	Precision, Recall and Accuracy for dependency labels . . . . .	35
5.3	Number of actual vs found CVEs . . . . .	35
5.4	Updated version vulnerability labels . . . . .	36
5.5	Precision, Recall and Accuracy for updated version test . . . . .	36
5.6	Naming mismatch between artifactId and NVD product name . . . . .	37
5.7	Project runtimes . . . . .	38
5.8	Vulnerable vs non-vulnerable runtimes . . . . .	38
5.9	Runtimes of various number of dependencies . . . . .	39
5.10	Runtimes of various number of dependencies with storage . . . . .	40
5.11	Evaluation of requirements . . . . .	41

# Chapter 1

## Introduction

This project aims to improve security during code development by identifying and linking source code artifacts (dependencies) to their closest knowledge unit(s) in the public information repositories. Public security sources (PSS) [29] can be defined as information sources that provide information regarding vulnerabilities, threats, attacks, risks, affected assets or available countermeasures (based on ISO and Std, 2009).

The issue of improving code security has been approached from many different angles. Several papers look at various approaches for scanning source code and identifying security vulnerabilities using mainly machine learning or static analysis tools. Many of these static analysis tools and machine learning models have proven themselves adept at analyzing the code and identifying vulnerabilities. While they are useful for scanning your own code, they generally do not consider the fact that a large portion of software consists of open source libraries and frameworks.

Synopsys' 2021 Open Source Security and Risk Analysis (OSSRA) report [39] showed that apps in 2020 averaged 528 open source components - an increase of 259% over the previous 5 years. They also found that as the use of open source code increases, so does the amount of vulnerabilities - with 84% of codebases having at least one vulnerability and the average codebase having 158 vulnerabilities.

The National Vulnerability Database (NVD) [35] is a public security source run by the United States government. Their website keeps track of vulnerability data on all things software related. If a software component has been identified to contain a vulnerability, the NVD will have a dedicated page about the specific vulnerability, containing any pertinent information to the issue. Among several other categories, maven dependencies are an example of such a software component.

In an attempt to address the security risks of the increasing number of vulnerabilities in open source code, this project will look at how the data from NVD can be used to provide crucial vulnerability information to developers, directly in their development environment. We will specifically look to create a plugin for



the Eclipse Integrated Development Environment which will identify vulnerabilities in maven dependencies, as maven is a commonly used project management and build tool. This will result in developers being notified directly in Eclipse of vulnerable open source components and will also provide them with relevant information to help mitigate the issues.

Answering the following three research questions will demonstrate how the proposed plugin contributes to improving security related to open source software (OSS):

- RQ1: How can we integrate public security information to detect vulnerable OSS libraries in the developer environment?
- RQ2: What is the performance of the OSS vulnerability detection tool in terms of completeness and soundness?
- RQ3: What is the performance of the OSS vulnerability detection tool in terms of runtime under different project sizes and dependencies?

The rest of the thesis is structured as follows: Chapter 2 provides background on the central aspects this research is built upon. In chapter 3, the plan for evaluating the resulting plugin is presented. Chapter 4 explains how the plugin is designed and implemented. In chapter 5, the results of the tests run on the plugin are provided as well as an evaluation of the results. In chapter 6, we look at the project as a whole and discuss some of the challenges faced as well as looking at how certain limitations could have been prevented. Chapter 7 provides information about prior research relating to the central aspects of this thesis. Finally, in chapter 8, we conclude the project and detail some of the future work that can be done related to this thesis.

# Chapter 2

## Background

The thesis is built on three broad areas namely; detecting security vulnerabilities, public information security sources and open source software. The following sections will introduce the three categories as background to this study.

### 2.1 Detecting Security Vulnerabilities

The issue of finding and fixing security vulnerabilities in software is a well researched topic. The consequences of vulnerable software are potentially enormous, as demonstrated repeatedly by news headlines describing data leaks in big corporations or services being unavailable due to attacks against them. Felderer et al. discuss how important it is to apply the appropriate security testing techniques and how these techniques are "essential to perform effective and efficient security testing" [15]. In this section, we are looking at how this issue is dealt with throughout the software development life cycle (SDLC) (see figure 2.1).

It is important to consider security during all stages of the SDLC. If the design of a product is inherently insecure, there is not much one can do to mitigate design vulnerabilities in the later stages of the SDLC. This applies to the requirements gathering phase of the product as well - it is necessary to consider the security implications of any requirement in order to properly design and implement secure solutions. While these are vital parts of creating secure products, this section will focus on the various methods and tools used throughout the SDLC to discover security vulnerabilities in the code.

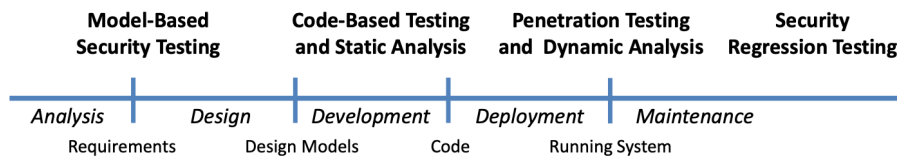


Figure 2.1: Security Testing Techniques in the Software Development Life Cycle [15]

### 2.1.1 Development Stage

A vast majority of research related to vulnerability detection is tied to the implementation stage of the SDLC. This is the stage where developers can get quick feedback on faults in their code before it causes issues in testing, or gets deployed and becomes a vulnerability.

This is the stage where static code analysis (SCA) is run. SCA is a method of analysing the source code of a program before it is run - i.e. checking the text and syntax of the code [18]. There are different ways of performing this check. The most common method is to compare the text to a set of predefined coding rules. This can be effective in discovering issues such as information flow vulnerabilities, weak cryptography and command injection to name a few. Examples of tools implementing this method are Spotbugs [34], ESVD [28] and SonarQube [33].

Another method for static analysis which has seen a steady increase in research over the years is the use of machine learning (ML) [17]. The specific approach to training a ML model differs a lot, and results in models with varying degrees of precision and capabilities. The term "static analysis" typically refers to the rules checking method described above, but the use of ML follows the same principle of analysing the code in its static form, before it is run. Whereas rule matching has the benefit of its rules generally applying to every codebase on which it is run, ML models are typically trained on a small number of codebases, giving them better performance on the code it is trained on, than a completely new codebase. Section 7 provides related studies about the use of machine learning in research.

Both these methods give the advantages of speed and depth in that they heavily outperform a human performing the same job, and that they check every nook and cranny of the code, giving results that should cover the entire product. At the same time, no one static code analyser has proven itself capable of detecting every type of security vulnerability in the code and none can guarantee 100% accuracy in its results. They will always be prone to false positives, meaning that a reported vulnerability is a non-issue, and false negatives, meaning an actual vulnerability is not reported by the tool [22].

Another important security testing technique in the development stage is the testing phase of the SDLC. This involves the developer or a dedicated tester either writing executable tests, or simply testing the results manually. This is what's referred to as dynamic code analysis (DCA). DCA does not have clearly declared methods the same way SCA does. It is rather a loosely defined process of testing the results of the code. There are multiple libraries made for writing tests and emulating live application data such as JUnit [21], Mockito [24] and Cucumber [14], but these do not perform any checks on behalf of the developer. It is the developer who implements their desired tests to verify desired behavior.

### 2.1.2 Deployment stage

When code has been written and tested, it reaches the deployment stage where depending on the development pipeline, a series of tools can perform various checks on the code. Often times, tools used in the development and testing

stages are also run during the deployment stage in order to ensure there is a barrier of checks before code is deployed. In addition to this, you could set up a plethora of other code-checking tools such as OWASP's Dependency-Check [16]. The deployment stage is also where user acceptance testing is done, but this is often more focused on features, rather than finding security vulnerabilities.

### **2.1.3 Maintenance Stage**

In the final stage of the SDLC, the product is deployed and needs to be maintained. This is generally a phase where bugs are found through use of the product and these need to be addressed as is standard procedure for any product in production.

Another security vulnerability detection method found at this stage is the use of penetration testing. While this method belongs to both the testing and deployment stages as well, I put it here as it is a vulnerability detection method performed on the product as a whole. Penetration testing is simply put the process of using white-hat hackers in an attempt to penetrate or exploit the product. A white-hat hacker is someone with the skills of a hacker, capable of doing harm to products and services - but is someone who uses those skills at the request of the owners of the product, in a simulated environment where no damage is inflicted. This results in potential vulnerabilities which could have been exploited by malicious agents being found and thereafter addressed. This is an important security vulnerability detection method, which when done well and often, keeps you on top of any vulnerable situation.

## **2.2 Public Security Sources**

Public security sources can be defined as information sources that provide information regarding vulnerabilities, threats, attacks, risks, affected assets or available countermeasures (based on ISO and Std, 2009) [29]. In this thesis we will focus on the three sources; Common Vulnerabilities and Exposures (CVE), Common Weakness Enumeration (CWE) and the National Vulnerability Database (NVD).

### **2.2.1 Common Vulnerability Exposures (CVE)**

"The mission of the CVE Program is to identify, define, and catalog publicly disclosed cybersecurity vulnerabilities" [12]. This is done by registering a new record for each vulnerability discovered by organizations that have partnered with the CVE Program. Each record goes through a thorough process with certain information requirements before it is published, as described in figure 2.2.

The current version of CVE offers limited information to developers as most entries only have a unique id, a description and possibly other fields such as references, as exemplified in table 2.1.

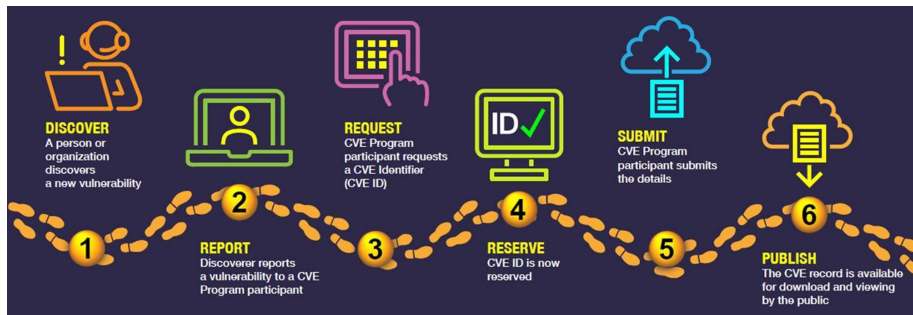


Figure 2.2: Registering a new CVE record

CVE-ID	Description
CVE-2022-29817	In JetBrains IntelliJ IDEA before 2022.1 reflected XSS via error messages in internal web server was possible

Table 2.1: Example of a CVE-record

### 2.2.2 Common Weakness Enumeration (CWE)

”CWE is a community-developed list of software and hardware weakness types. It serves as a common language, a measuring stick for security tools, and as a baseline for weakness identification, mitigation, and prevention efforts” [13]. Where CVE registers individual instances of vulnerabilities, CWE keeps track of more general vulnerability data. Among the information you will find in a CWE entry are the following:

- Exploitation likelihood
- Code examples
- Related CVE entries
- Potential mitigations
- Detection methods
- Common consequences
- Related attack patterns

CWE-ID	Description
CWE-1173	The application does not use, or incorrectly uses, an input validation framework that is provided by the source language or an independent library.

Table 2.2: Example of CWE-record

### 2.2.3 National Vulnerability Database (NVD)

NVD is essentially an extension of CVE. "The NVD is the U.S. government repository of standards based vulnerability management data." [35]. On their website they specify that they perform analysis on CVEs that have been published to the CVE Dictionary. This analysis results in more detailed information compared to the original CVE entries, including a severity score, references to CWE-entries if applicable and references to specific software affected by the issue, using Common Platform Enumeration (CPE).

"Common Platform Enumeration (CPE) is a standardized method of describing and identifying classes of applications, operating systems, and hardware devices present among an enterprise's computing assets." [9]. This is one of the most important features of NVD relating to this thesis. CPEs contain a URI describing the affected software, which can easily be used for searching and matching against dependencies. The following example (taken from Common Platform Enumeration: Naming Specification Version 2.3, page 23) shows how a well formed name (WFN) is translated to a URI:

Suppose one had created the WFN below to describe this product: Microsoft Internet Explorer 8.0.6001 Beta (any edition):

```
wfn:[part="a",vendor="microsoft",product="internet_explorer",  
version="806001",update="beta",edition=ANY]
```

This WFN binds to the following URI:

```
cpe:/a:microsoft:internet_explorer:8.0.6001:beta
```

A particularly important feature of NVDs use of CPE is its clarity concerning versions. Software is not a static concept where you are stuck with what you got indefinitely. New versions of software are released constantly which generally introduce a mix of new features and bug fixes. A product can thus not be labeled "vulnerable" as a whole, as a vulnerability in one version is not necessarily present in another version of the same product. In cases where a CVE entry applies to multiple versions, NVD keeps track of which version introduced the issue and which version fixed it if it has been fixed.

## 2.3 Open Source Software

Open source software (OSS) has been around since the 1980s when Richard Stallman, who was sick of the increasing amount of necessary, proprietary software, coined the term "free software" [5]. The use of OSS has been consistently growing for decades. Researchers have been talking about its increasing popularity for more than 20 years, and recent reports such as Synopsys' 2021 Open Source Security and Risk Analysis (OSSRA) report [39] found that apps in 2020 averaged 528 open source components - an increase of 259% over the previous 5 years.

Open source is simply defined by many as software with source code that can be

viewed and modified by anyone. There is however a more technical definition, "The Open Source Definition" (OSD), overseen by the non-profit organization the Open Source Initiative (OSI). OSD states ten criteria which must be complied for a license for software to be considered open source (see figure 2.3). These criteria ensure that developers can freely plug OSS into their own code and is the big reason why the use of OSS continues to increase. Why write code from scratch if someone has already made the feature available to you for reuse?

- |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ol style="list-style-type: none"><li>1. <b>Free Redistribution:</b> There shall be no restrictions on the redistribution of aggregate software distributions with the OSS as a component.</li><li>2. <b>Source Code:</b> The source code of a product must be obtainable.</li><li>3. <b>Derived Works:</b> A modified or derived work of the product, must be allowed to be distributed under the same license.</li><li>4. <b>Integrity of The Author's Source Code:</b> The license may restrict distribution of a modified version of the product so long as it allows for distribution of patch-files, modifying the program at build time.</li><li>5. <b>No Discrimination Against Persons or Groups:</b> Any person or group of persons must be allowed to use the product.</li><li>6. <b>No Discrimination Against Fields of Endeavor:</b> The product must be allowed to be used in any field.</li><li>7. <b>Distribution of License:</b> The license must extend the rights of the product to all redistributions.</li><li>8. <b>License Must Not Be Specific to a Product:</b> The rights of the product must extend to all software distributions.</li><li>9. <b>License Must Not Restrict Other Software:</b> The license is independent and cannot restrict other programs.</li><li>10. <b>License Must Be Technology-Neutral:</b> The program must allow use with any technology.</li></ol> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Figure 2.3: The Open Source Definition

### 2.3.1 Use of Open Source Software

The reasons for using OSS are many. Perhaps the most obvious one being convenience. With an ever growing pool of open source solutions available to any developer - the odds of finding existing, appropriate solutions to your problems are not slim. There is rarely a need to reinvent the wheel, so using OSS makes sense from a developer's perspective. This argument applies to proprietary solutions as well, so it goes to show why existing solutions are used in general. Looking at what separates open source from proprietary software, the most common factors found amongst websites explaining the advantages of OSS [6][38][47] can be found summarized in table 2.3.

Several reports have looked at the use of OSS in businesses and codebases in recent years. OpenUKs "State of Open: The UK in 2021" [25] ran a survey which found that 89% of UK businesses run open source software. Synopsis' 2022 OSSRA report [40] showed that of almost 2500 codebases scanned, 97% of them contained open source. Not only that, but as much as 78% of all codebases were composed of open source. These findings show how prevalent OSS is in both business and in software development.

Factor	Advantage
Security	Code is viewed and maintained by a large number of people ensuring fewer bugs and faster responses to fix those that make it through.
Customisability	The source code can be freely customized to better suit the needs of the user.
Stability	Continued development of the software is not dependant on a single person or group of people. If the original creators stop maintaining the software, others can step in.
Interoperability	OSS is generally developed to be as compatible as possible.
Cost	Minimal to no cost

Table 2.3: Advantages of Open Source Software

### 2.3.2 Open Source Example: Maven Dependency

An example relevant to this thesis of how OSS is used, is the use of dependencies in maven projects. Maven is a popular build tool for java. A dependency in this setting is some piece of software that your project depends on to function. Dependencies are uploaded by their respective creators to the central maven repository, which keeps track of all versions of the software. When a developer wants to use a dependency, they add its descriptive parameters to their project's "pom.xml" file as demonstrated below:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13.2</version>
  <scope>test</scope>
</dependency>
```

This tells maven to download version 4.13.2 of JUnit to your local maven repository and enables you to use the dependency's API in your project. The source code that produced the API is not visible locally in the developer's environment, so when using maven dependencies, developers rarely inspect the source code itself.

### 2.3.3 Vulnerability in Open Source

Despite security being listed as one of the factors why people use OSS, security will always be an issue in software development. The 2022 OSSRA report [40] found that 81% of the codebases audited contained at least one open source vulnerability.

The nature of open source is the fact that anyone can view the source code, and this is an argument used by many to underline the claim of high security



in OSS. In the case of the maven example in section 2.3.2 however, I mentioned that the practical use of dependencies rarely involves inspecting the source code. Without explicitly reviewing the code or scanning it for vulnerabilities by other means, a developer essentially chooses to trust that dependency when it in reality has the potential to introduce critical security vulnerabilities.

I also talked about how a security advantage to OSS is a generally faster response time to releasing bug fixes. Given the ease of releasing new versions to the maven repository, it's fully possible a newer version of the software is available which is not affected by an older vulnerability. While this should bode well for OSS security, Synopsis found [40] that 85% of codebases "contained open source that was more than four years out-of-date" - showing the need for systems that check your dependencies and inform you of their known vulnerabilities.

# Chapter 3

## Methodology

This chapter describes how the three research questions related to developing a tool that utilizes public security information to detect vulnerable OSS libraries are addressed and evaluated. This will be done in three parts:

1. Developing a plugin for the Eclipse IDE which will identify vulnerable dependencies and connect developers to relevant vulnerability data from public security sources directly in their development environment.
2. Measuring the precision and recall of our plugin by running it on open source components.
3. Measuring the runtime of the plugin by running it on maven projects of varying sizes.

The introduction of public security sources informing developers of vulnerabilities in their dependencies should make developers either find different versions/dependencies which are not vulnerable, or have them mitigate the vulnerabilities themselves if possible. This will result in fewer vulnerabilities in the code and improved security. This assumption is discussed further in chapter 6.

The following requirements for the plugin were identified:

Requirement	Type	Related to
Plugin should link to nvd	Functional	RQ1
Plugin should provide developers with information in development environment	Functional	RQ1
Plugin should be able to detect vulnerable libraries correctly	Functional	RQ2
Plugin should have zero false positives and false negatives	Functional	RQ2
Plugin should scale on large projects	Non-Functional	RQ3

Table 3.1: Requirement table for plugin

### 3.1 RQ1: Development of plugin

In order to connect developers and vulnerability data and answer RQ1; "How can we utilize public security information to detect vulnerable OSS libraries in the developer environment?", a plugin for the Eclipse IDE will be developed. This plugin will facilitate simple and immediate access to relevant vulnerability data from the three public security sources CVE, CWE and NVD mentioned in section 2.2.

As discussed in the background, most codebases today consist of more open source code than original code (i.e. code written specifically for a single product). In spite of this, there is a much heavier research focus on tools and techniques that implement static code analysis or other similar methods to identify vulnerabilities in developers' own code. This plugin will address the neglected issue of vulnerabilities in open source components, by bringing a much needed focus specifically to vulnerabilities in dependencies that are used in maven projects.

The goals for the plugin are that it:

1. Identifies vulnerable dependencies and appropriately mark them as such.
2. Fetches and displays information relevant to developers in an easily accessible manner.
3. Provides performance that makes it appealing for developers to use.

For this plugin to provide any significant value, it needs to be accurate. This means that when developers execute a scan of one of their projects, they should be confident that any vulnerability data from the public security sources, related to any of their dependencies, will be found and that the vulnerable dependencies will be clearly marked with an appropriate marker. If this is not achieved, it will fall short of users' expectations and will ultimately not be considered a viable plugin to help improve security.

Next it is important that the plugin fetches data that is relevant to developers. NVDs API provides a substantial amount of data (see appendix C) which contains a lot of useful information, but also much that is irrelevant to developers. The plugin should efficiently sort the data so that the developer is left with enough information to guide a decision making process about a vulnerable dependency, but not so much information that they are overwhelmed and unable to use the results.

A key component in the success of the plugin is whether or not it is intuitive to use. Eclipse is already packed with buttons, menus and views, so there is no need to add more chaos. This plugin is meant to be a simple, but useful addition to the workflow of a developer by helping mitigate vulnerabilities and requiring little to no effort to do so. When performing a scan of a project, the user should be informed of the progress and be presented with the results upon a completed scan - without having to further interact with menu options or other intrusive user interface elements. The results of the scan should be presented in an easily comprehensible manner, where the user can interact with and view the vulnerability data fetched from NVD.

Finally, the plugin’s performance needs to meet the expectations of the users. This point ties into how intuitive the plugin will be, as a long processing time could result in developers thinking the scan has failed, or even making them second guess whether or not the plugin is something they wish to continue using. To help with performance, the results of each scan should be stored so that they potentially could be re-used at a later stage.

Overall, the criteria mentioned in this section describe a product which should provide good value to developers. If the plugin meets these criteria to an acceptable level, users will have few reasons not to incorporate it into their workflow and their routines during development. At that point it will be up to them to address the concerns raised by the plugin in whatever way necessary to mitigate security vulnerabilities in their products.

## 3.2 RQ2: Measuring performance of the plugin in terms of completeness and soundness using open source dependencies

The approach to answering RQ2; “What is the performance of the OSS vulnerability detection tool in terms of completeness and soundness?”, is to run a case study where the plugin is tested by running it on open source components. By having a good sample of vulnerable and non-vulnerable dependencies, as well as various versions of them, the results will give a good indication of how well the plugin performs.

### 3.2.1 Collecting test data

To build a ground truth dataset of vulnerable and non-vulnerable dependencies, a total of 40 unique dependencies are collected, split equally between 20 vulnerable and 20 non-vulnerable. “Unique” in this setting is defined to mean that two different versions of a dependency are counted as one towards the total number of dependencies. Vulnerability is determined by there being CVE entries registered on the dependency or not.

When finding a dependency for testing, there must be a thorough check into whether or not it is vulnerable. Without the proper labels as a starting point, the results will have little significance and will be easily scrutinized. With this in mind, the following process for collecting the sample of dependencies is used.

The website “mvnrepository.com” [1] (henceforth referred to as “the maven repository”) is used as a starting point. This site offers the ability to browse through maven dependencies which have been sorted into a number of categories. Using this functionality, a diverse set of dependencies is collected, with dependencies used for purposes such as “Logging Frameworks” and “I/O Utilities”.

When a dependency is selected as a candidate, the next step is to label it. Each dependency will be labeled as one of the following depending on the test case being run:

- $V = \textit{This dependency is vulnerable}$

- *NV = This dependency is not vulnerable*
- *FV = This is a version of a dependency where vulnerabilities of a previous version have been fixed, but it still contains other vulnerabilities*
- *FNV = This is a fixed version of a previously vulnerable dependency, no longer vulnerable*

The initial batch of dependencies are labeled either V or NV. Once the baseline test with the 40 dependencies has been run, the two labels FV and FNV are introduced to describe different versions of the dependencies.

The maven repository contains detailed information about every dependency in its registry. Among this data is a "Vulnerabilities" field which lists CVE entries associated with the dependency. As any information regarding the accuracy of the vulnerabilities lists is not available, nor any information saying how often the lists are updated, this information can not be depended upon alone. Each CVE entry listed by the maven repository is therefore checked before it is registered to the dependency in the sample data.

The next step is to utilize the search functionality of CVE. Different variations of the details of the dependency (i.e. artifactId, groupId, organization) will be used as keywords, to cast as wide a net as possible for finding potentially relevant CVE entries. From CVE, the built-in linking to its associated NVD entry is followed in order to find the affected versions. These steps should provide well vetted dependencies as the test sample.

For each dependency included in the sample, the CVE entries associated with it and which versions each CVE entry applies to are noted. The date and time when the dependency has been properly vetted is also registered. By doing this, new CVE entries being registered every day and the fact that changes to the vulnerability status of a dependency may occur between the time it is collected, and the dependency being used in the tests of the plugin, is taken into account.

The collected dependencies are presented in table 3.2.

### 3.2.2 Testing the plugin

While the practical use of the OSS vulnerability detection tool will be to separate vulnerable dependencies from non-vulnerable dependencies, the tool's performance in terms of completeness and soundness essentially boils down to being able to find CVE entries. As this is the measuring stick, tests are run which focus on all the CVE entries found, and not just on whether or not the plugin identifies a dependency as vulnerable.

In order to get as accurate an assessment as possible, multiple tests are run with different combinations of dependencies. As the plugin will only look at dependencies and not the code of a project, these tests will be performed with a shell of a project containing only the open source components being analyzed, inside the project's "pom.xml" file. Therefore the first step is to create a new, empty Maven project.

Each test will then involve the following steps:

1. Add the relevant selection of dependencies to the shell project

GroupId	ArtifactId	Version	Label	Number of CVEs found
log4j	log4j	1.2.17	V	6
commons-io	commons-io	2.4	V	1
com.fasterxml.jackson.core	jackson-databind	2.13.2	V	1
org.springframework	spring-core	5.3.14	V	3
org.apache.derby	derby	10.6.2.1	V	2
org.eclipse.jetty	jetty-io	11.0.1	V	5
junit	junit	4.13	V	1
com.google.code.gson	gson	2.8.5	V	1
org.infinispan	infinispan-core	11.0.5.Final	V	1
com.puppycrawl.tools	checkstyle	5.6	V	2
com.google.guava	guava	18	V	2
com.h2database	h2	1.4.200	V	3
org.jsoup	jsoup	1.13.1	V	1
org.apache.httpcomponents	httpclient	4.2	V	3
com.mchange	c3p0	0.9.5.2	V	1
com.alibaba	fastjson	1.2.7	V	1
org.codehaus.jackson	jackson-mapper-asl	1.9.13	V	1
ch.qos.logback	logback-classic	1.1.3	V	1
org.hibernate	hibernate-core	5.4.2.Final	V	1
org.mybatis	mybatis	3.4.6	V	1
com.typesafe.akka	akka-actor_2.13	2.6.14	NV	0
io.dropwizard.metrics	metrics-core	4.2.3	NV	0
org.clojure	tools.cli	0.3.5	NV	0
org.springframework.boot	spring-boot-autoconfigure	2.6.6	NV	0
com.google.dagger	dagger	2.41	NV	0
com.squareup.okhttp	okhttp	2.7.5	NV	0
p6spy	p6spy	3.9.1	NV	0
com.itextpdf	itextpdf	5.5.13	NV	0
com.microsoft.azure	azure-core	0.9.3	NV	0
com.googlecode.usc	jdbcdslog	1.0.6.2	NV	0
com.github.scala-incubator.io	scala-io-file_2.10.2	0.4.2	NV	0
org.hsqldb	hsqldb	2.4.1	NV	0
org.mockito	mockito-core	3.12.4	NV	0
joda-time	joda-time	2.9.9	NV	0
com.sun.mail	jakarta.mail	2.0.1	NV	0
org.ccil.cowan.tagsoup	tagsoup	1.2.1	NV	0
com.github.benmanes.caffeine	caffeine	2.9.3	NV	0
io.micrometer	micrometer-core	1.8.4	NV	0
jdepend	jdepend	2.9.1	NV	0
com.andrewmcveigh	cljs-time	0.5.2	NV	0

Table 3.2: Maven dependencies used for testing

2. Execute a scan on the project
3. Export the results for analysis
4. Note the CVEs found for each dependency
5. Mark each CVE entry found as TP or FP (see table 3.3)
6. Note the number of CVE entries not found by the plugin = FN

We start with a baseline test of all of the 40 dependencies in the same project. The results will then be compared to the data found during the dependency sample collection process. A number of the vulnerable dependencies will have several CVE entries associated with them, while others may have only one. The next step in testing will then be to look at alternate versions of the dependencies. These alternate versions will be labeled either FV or FNV and have a different set of CVE entries than the previously tested version. The purpose of this step is to test how well the plugin takes versions into account. Running these tests will cover the various use cases (as suggested by the labels) of the plugin using representative data.

		Actual	
		Positive	Negative
Predicted	Positive	TP	FP
	Negative	FN	TN

Table 3.3: Confusion matrix of vulnerability labels

Once all tests have been run, the performance in terms of completeness and soundness is evaluated based on the precision and recall of the plugin. This is explained further in section 5 where we look at the results.

### 3.3 RQ3: Measuring performance of the plugin in terms of runtime using open source maven projects

Another important aspect of any successful software development tool is asked in the third research question; "What is the performance of the OSS vulnerability detection tool in terms of runtime under different project sizes and dependencies?", with runtime being the key. While accuracy and value of the information gained from the plugin are the most vital parts, runtime performance plays an important part in establishing the plugin as a viable option for any developer to use. It is also important to establish that the plugin functions properly regardless of the size of project on which it is run.

The runtime performance of this plugin will be tested by collecting three open source maven projects from GitHub, and running the plugin on them. These

projects will be of varying sizes, ranging from small independently developed projects to large scale enterprise software. In addition, they will each be injected with a number of extra dependencies. This will provide a good coverage of different combinations of project sizes and numbers of dependencies.

### 3.3.1 Collecting test data

For the test projects, the main criteria is for a project to represent a project size on the mentioned range from small to large, which is not already represented. As the plugin doesn't concern itself with the code, the contents of the projects are irrelevant, only their size will be a factor in these tests.

GitHub does not display information such as lines of code (LOC) or number of files, so in order to fetch this, the command line interface "cloc" (short for "count lines of code") is used.

In addition to the projects, a number of dependencies are needed which will be injected into the projects. Rather than finding new dependencies specifically for these tests, the dependencies from the first case study are re-used.

Project	Number of files	Lines of code	Memory size
OSS-Maven	35	2457	217 kB
Log4j 2	3429	252 148	55 MB
BioJava	1619	902 697	118 MB

Table 3.4: Projects used for testing

### 3.3.2 Testing the plugin

There are many scenarios we wish to discover the runtime performance of. We will start by looking into the impact of project size on the plugin. This will be done by taking the three projects of different sizes and running the plugin a total of three times each, with the same set of forty dependencies used every scan.

Next we want to test how well the plugin scales related to runtime performance. This test will be performed by scanning four different amounts of dependencies. For the first scan, the project will have a total of ten dependencies. For the second, this number is increased to twenty, before the third scan utilizes the forty dependencies collected for RQ2. For each of these tests, an equal amount of vulnerable- and non-vulnerable dependencies will be used. By using the same ratio of vulnerable to non-vulnerable dependencies for each test, the results will be comparable to help determine any trends for the first three sets. The fourth set of dependencies will consist of the 121 dependencies used in the log4j project. This set will have a different composition to the previous three sets, but will offer insight into runtime performance after a significant increase in dependencies.

In addition to the tests mentioned above, we want to discover how the plugin will be in a more practical setting. To simulate the natural progression of development, one of the projects is selected and the same tests are performed,



starting at ten dependencies and progressing through twenty, up to forty dependencies. The difference between the two sets of tests will be that this time the tests are performed without deleting the results of the previous scans. This will demonstrate the impact of implementing a storage solution and leave two sets of results which can be compared where the only differentiating factor is the use of storage.

Finally we want to test the difference in performance between scanning vulnerable dependencies versus scanning non-vulnerable dependencies. This will be done by scanning each of the two sets of 20 dependencies separately. Similarly to the previous tests, this will also be done incrementally with an increasing number of dependencies for each scan.

Through the tests described in this section, we should learn the following:

- What impact does project size have on runtime performance?
- How well does the plugin handle an increasing amount of dependencies - is there a linear time difference depending on the number of projects?
- What impact does storing the results of the previous scans have on runtime performance
- What is the difference in runtime between scanning a project filled with vulnerable dependencies versus scanning a project with no vulnerable dependencies

## Chapter 4

# Design and Implementation

In this chapter, we will take a closer look at how the vulnerability detection plugin is designed and implemented. We'll start by getting an overview of the architecture and what the plugin is built upon. Next we will look in more detail how it is implemented by looking at the various components that enable this plugin to connect developers to relevant information from the public security sources. The components will be presented in a chronological manner from the perspective of executing a scan on a project.

### 4.1 Architecture

This plugin in its entirety has been developed for Eclipse using Eclipse's Plugin Development Environment (PDE). This environment provided all the tools needed to develop, test and deploy the plugin. The PDE API Tools were frequently used to work with the components of Eclipse needed to implement components such as actions triggered by menu items, interaction with the workspace and projects, and the generation of custom views to the user interface, to name a few. Examples of the packages in the API can be seen in table 4.1.

Package	Description
org.eclipse.core.runtime	Provides core support for plug-ins and the plug-in registry
org.eclipse.core.resources	Provides basic support for managing a workspace and its resources.
org.eclipse.jface.action	Provides support for shared UI resources such as menus, tool bars, and status lines.

Table 4.1: Examples of Eclipse PDE API packages

When we look at the responsibilities of this plugin, we can see that it needs to interact with the data it fetches in a number of ways. To accommodate this, four models which are connected with one another were made.

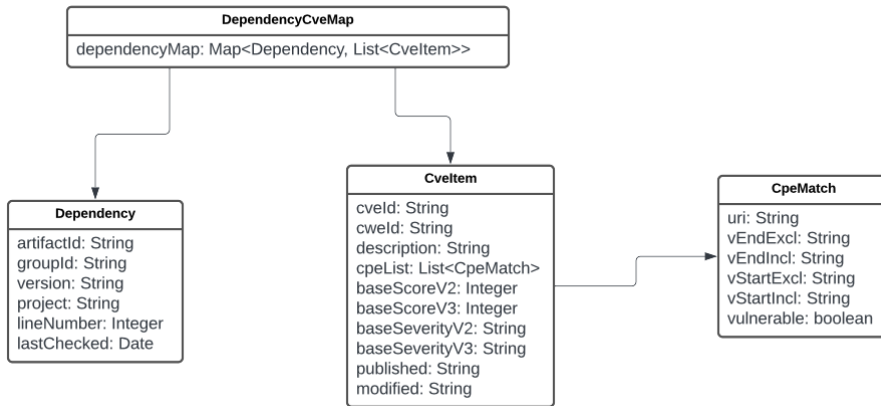


Figure 4.1: Simple data model for the plug-in

The data this plugin will be dealing with are dependencies, and potentially vulnerability data associated with them. In order to distinguish dependencies from one another their identifying attributes are stored. Information specific to a dependency is also kept track of, which will be needed throughout the scanning process. This will be explained in the following sections.

In section 4.3, we look at the structure of the data fetched from NVD. The two models "CveItem" and "CpeMatch" were implemented to accommodate this structure and nest relations between data in a logical manner.

The following sections describe the components as presented by figure 4.2.

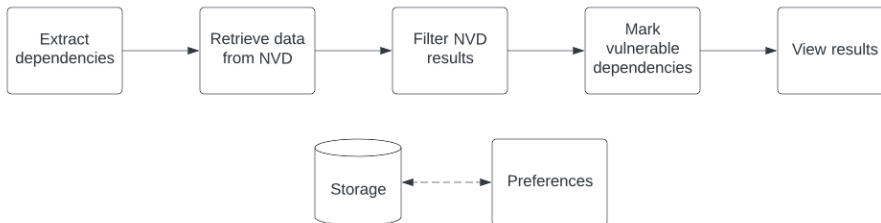


Figure 4.2: Simple sequence chart of scanning process

## 4.2 Extracting dependencies

This plugin is implemented to work with maven projects. While there are several other types of projects that would benefit from a vulnerable dependency checking tool like this, the focus was narrowed down in order to create a proof of concept plugin which demonstrates the potential of improving security by checking dependencies. Selecting maven as the target building tool facilitates easy access to a project's dependencies, as the dependencies are conveniently imported in a standardized manner as demonstrated in figure 4.3.

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.13.2</version>
</dependency>
```

Figure 4.3: Example of dependency added to pom.xml

When a scan is initiated, the first step is for the plugin to create a reference to the "pom.xml" file. This file is then parsed by a custom XML-reader. This is done so that the line number of each dependency is stored for later use when markers are registered. Every XML-node from the "pom.xml" file starting with "<dependency>" is then passed to a Dependency-object constructor which stores its groupId, artifactId, version, line number, the project it belongs to and a timestamp of when it was fetched.

Next, the plugin checks if a scan has been run for the project previously, as the scans are stored for later use. If there is previous data available, the plugin will cross-check its newly fetched list of dependencies from the project, with the ones fetched from storage. Details about how and what is stored are in section 4.7. When cross-checking, the plugin separates dependencies that already have valid data from storage, from those that do not.

### 4.3 Retrieving data from NVD

All the dependencies which do not already have valid data from storage, are passed on so that the plugin can attempt to fetch vulnerability data for each of them. In cases where a scan is being run for the very first time on a project, or all the previous data is no longer valid, every dependency is checked in this step. What constitutes "valid" is explained in section 4.8.

The check is done by querying NVD's API. The API allows retrieval of collections of CVE entries and offers a wide range of optional parameters to limit or filter the results. It offers the same data which can be found by utilizing the search functionality on the NVD website. The API is open to anyone, but has a limited amount of queries in a given time window, unless the user requests an API key. Responses from the API are provided in JSON format.

For this plugin, the "cpeMatchString" parameter from the API is utilized in order to find applicable CVE entries. This parameter allows to find all CVE entries associated with the given CPE. As mentioned in section 2.2.3, all CPEs follow a naming standard called well formed name (WFN). In NVD's documentation of how the various attributes that make up a WFN are formed, they state that the product attribute "... SHOULD describe or identify the most common and recognizable title or name of the product." [10].

With this information, and after checking a number of maven dependencies, the "artifactId" attribute of a maven dependency was found to correspond to the "product" attribute of a CPE in all the dependencies checked. The job of querying NVD for each dependency was then a simple matter of using the

```
"cve":{
  "data_type":"CVE",
  "data_format":"MITRE",
  "data_version":"4.0",
  "CVE_data_meta":{
    "ID":"CVE-2019-1010218",
    "ASSIGNER":"cve@mitre.org"
  },
}
```

Figure 4.4: Example showing part of JSON response from NVD API

dependency's "artifactId" as part of the query parameters.

## 4.4 Filtering NVD results

The results from querying NVD are passed on to a custom parser. This parser takes each set of dependencies and their results and constructs the appropriate java objects of the classes `CveItem` and `CpeMatch` (as can be seen in section 4.1).

CPE matches are formatted in a variety of ways. The two most important parameters are the CPE-uri and the versions which the CVE entry applies to. Sometimes the version is mentioned directly in the uri, which means the entry only applies to that specific version. Most often however, the response will include up to four of the possible elements; *versionEndExcluding*, *versionStartIncluding*, *versionEndIncluding*, *versionStartExcluding*.

```
"cpe_match" : [ {  
  "vulnerable" : true,  
  "cpe23Uri" :  
  "cpe:2.3:a:imapfilter_project:imapfilter:*:*:*:*:*:*:*",  
  "versionEndIncluding" : "2.6.12"  
  } ]
```

Figure 4.5: Example of CPE result from NVD API

Because of the many different possibilities in how affected versions are presented in the NVD response, there is a need to account for a wide range of combinations when we are filtering the results.

The result of the custom parser is a single `DependencyCveMap` java object. This object contains mappings from the dependencies that have just been checked to all their corresponding `CveItems`. Each `CveItem` contains the information deemed the most relevant from a CVE entry. Not all the information available is included as this would be far too much data to present to developers in a small view.

Each `CveItem` also contains a list of all its affected CPEs as `CpeMatch` objects. These objects contain the CPE-uri which functions as a description of the product, in addition to the aforementioned version descriptors and are used to identify whether or not the `CveItem` applies to a dependency.

## 4.5 Marking vulnerable dependencies

When vulnerability data has been fetched for all the dependencies in the "pom.xml" file, those which are considered vulnerable are identified and markers are created for them. When identifying the vulnerable dependencies, the plugin runs through the data of each one to see whether there are `CveItems` linked with them in the vulnerability map, and whether those `CveItems` apply to the specific version used in the project.

A new marker type was created in the plugin's "plugin.xml" file (see figure 4.6). This custom marker type ensures that the vulnerable dependency markers are

sorted in a category of its own in Eclipse's marker view, making them easier to find and inspect.

```
<extension
  id="vulnerableDependency"
  name="Vulnerable Dependency"
  point="org.eclipse.core.resources.markers">
  <super type="org.eclipse.core.resources.problemmarker"/>
  <attribute name="artifactId" />
  <attribute name="groupId" />
  <attribute name="version" />
  <persistent value="true" />
</extension>
```

Figure 4.6: Custom Eclipse marker

Creating a new marker type also allowed to add the attributes used to identify and compare dependencies. This was necessary to being able to delete the marker associated with a dependency, should the developer choose to remove a dependency from the results of a scan.

Vulnerable Dependency (3 items)					
🚩	The dependency [com.fasterxml.jackson.core]	pom.xml	/TestingPlugin	line 15	Vulnerable Depe
🚩	The dependency [log4j] may be vulnerable	pom.xml	/TestingPlugin	line 36	Vulnerable Depe
🚩	The dependency [org.apache.derby] may be v	pom.xml	/TestingPlugin	line 26	Vulnerable Depe

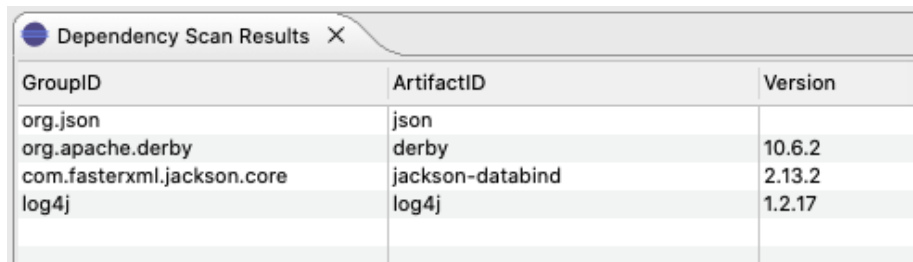
Figure 4.7: Marker view in Eclipse

## 4.6 Viewing results

Upon a completed scan of a project when vulnerable dependencies are identified and appropriate markers are created, the plugin automatically opens the detailed results of the scan in a new tab in the Eclipse environment. The challenge of this tab is to give an overview of the vulnerable dependencies, without overwhelming the developer with too much information.

To avoid this issue, the vulnerability information is sorted into stages of importance. The first stage and the one deemed the most important, is a general overview of all the dependencies which were identified to be vulnerable. This can be seen in figure 4.8. If the developer finds one of the vulnerable labeled dependencies to not be of any concern, or to be falsely labeled, they can simply remove it from the results via a right-click option. This also triggers the removal of that dependency's marker, as discussed in the previous section.

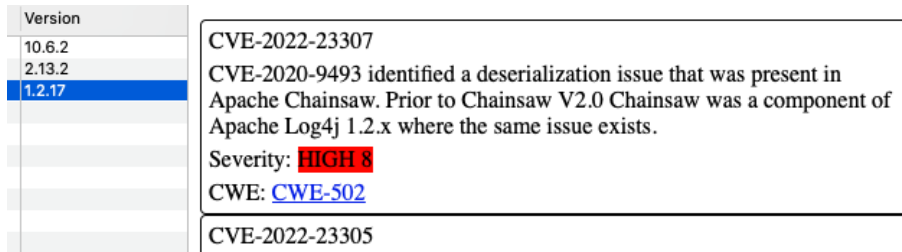
The next stage of information is the basic vulnerability information gathered from associated CVE entries. As mentioned in section 4.3, the NVD API provides a lot of information - not all of which is as important as the rest. This view is generated in a split view next to the list of vulnerable dependencies when the developer selects an item from the list. The detailed view of the CVE entries contains the ID of the CVE, its description as displayed on the NVD website,



GroupID	ArtifactID	Version
org.json	json	10.6.2
org.apache.derby	derby	2.13.2
com.fasterxml.jackson.core	jackson-databind	2.13.2
log4j	log4j	1.2.17

Figure 4.8: Results tab, dependency list

the severity of the issue and finally a link to a CWE-id if this is present in the data.



Version	
10.6.2	
2.13.2	
1.2.17	<p>CVE-2022-23307</p> <p>CVE-2020-9493 identified a deserialization issue that was present in Apache Chainsaw. Prior to Chainsaw V2.0 Chainsaw was a component of Apache Log4j 1.2.x where the same issue exists.</p> <p>Severity: <b>HIGH 8</b></p> <p>CWE: <a href="#">CWE-502</a></p>
	CVE-2022-23305

Figure 4.9: Results tab, CVE details

With these two steps, it is assumed that developers are provided with enough information to either act upon, or to encourage them to look up more details which can easily be done by using the provided CVE- and CWE IDs.

In addition to the results tab automatically opening, the developer has the option to open the most recent scan results of a project by right clicking the project in the project view. If the developer wants to export the results, or simply view the vulnerability data for all the dependencies in a single page, they also have an option to open the results in an html page. This action dynamically generates html using a `DependencyCveMap` as input, stores the file on the computer and proceeds to open the generated html-file in the developer's default program for handling the file type.

## 4.7 Storage

Storage has been mentioned a couple of times throughout this chapter. This was implemented as part of the process in order to provide better performance and to enable the option to view old results without having to run a new scan.

Data is stored by utilizing the project's workspace. For every workspace in Eclipse, there is a sub-directory called ".metadata" which already contains sub-directories for every plugin you have. By working with this directory, we set up a structure that allows to store data for each project individually, as can be seen in figure 4.10.



This structure was chosen as opposed to storing all dependency data in one large file, because it provides better performance when fetching and storing data. Having a single storage of data would require filtering, removal and merging of new and old data every time a scan is run. Every dependency also contains project-specific data such as line number which would not translate to any other project. While there is an argument that could be made for a central storage solution, where vulnerability data from a scan could be used in multiple projects, it was decided that the vulnerability data of a project could follow that project's life cycle. The addition of a new dependency in a project should warrant a new check of it either way.

```
$Workspace_directory
-> .metadata
  -> .plugins
    -> dependency-scanner
      -> $Workspace_name
        -> $Project_name
          -> data.txt
```

Figure 4.10: Storage structure

The data is stored to a ".txt" file using Java's `ObjectOutputStream` and `ObjectInputStream` classes. These allow to simply provide an object of the class "DependencyCveMap" as a parameter and store it as an object, rather than having to parse it and perform a more manual process involving splitting the data into several files. Similarly, when fetching the stored data, providing the path of the file returns the data as a "DependencyCveMap" object.

## 4.8 Preferences

"Valid data" has been mentioned a couple of times throughout this chapter. As data is being stored, there is a need to set a limit for how long the plugin can rely on that data before it could be considered outdated. This limit should not be hard coded and final, as projects do not necessarily have the same requirements. It was therefore decided to provide this as a preference option to the developer.

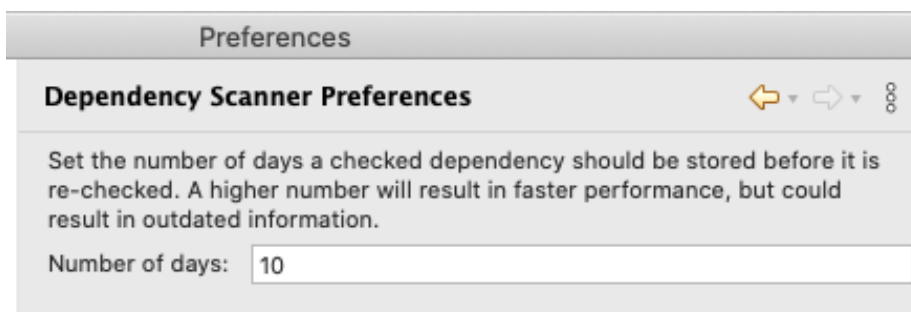


Figure 4.11: Plugin preference page

# Chapter 5

## Evaluation

This chapter presents the results of evaluating the open source vulnerability tool by reviewing the results from the various tests described in chapter 3. This evaluation looks at each of chapter 3's sections individually before taking a look at the results as a whole.

### 5.1 RQ2: Measuring performance of the plugin in terms of completeness and soundness using open source dependencies

Through the process described in chapter 3 40 unique dependencies have been identified to be used in the testing of this plugin. Each dependency has been thoroughly examined for any available vulnerability data in NVD. While there are some vendors who are more prevalent in the public security sources, such as Apache with their long lineup of products, an emphasis was put on having as diverse test data as possible. This resulted in 18 unique vendors for each of the vulnerable, and non-vulnerable dependencies respectively.

#### 5.1.1 Soundness and Completeness

Determining the soundness and completeness of the plugin consisted of a simple test to see how many vulnerable dependencies it could correctly identify. This was done by adding all 20 vulnerable and 20 non-vulnerable dependencies (40 total) to a maven project's "pom.xml" file and performing a scan.

Table 5.1 shows how the 40 dependencies from the initial test were identified. Using this data the precision, recall and accuracy of the plugin is calculated.

As can be seen in tables 5.1 and 5.2, the plugin produced varying results. A precision of 0.929 shows a relatively high confidence in that dependencies marked as vulnerable by the plugin, are in fact vulnerable. While precision is important to avoid a situation where developers need to constantly verify that a dependency

		Actual	
		Vulnerable	Non-vulnerable
Predicted	Vulnerable	TP = 13	FP = 1
	Non-vulnerable	FN = 7	TN = 19

Table 5.1: Dependency vulnerability labels

Metric	Formula	Result
Precision	$\frac{TP}{TP+FP}$	0.929
Recall	$\frac{TP}{TP+FN}$	0.650
Accuracy	$\frac{TP+TN}{TP+FP+FN+TN}$	0.800

Table 5.2: Precision, Recall and Accuracy for dependency labels

is vulnerable, the recall metric is far from good enough. With a score of 0.650, this shows that many vulnerable dependencies are not picked up by the plugin.

Set of dependencies	Actual number of associated CVEs	Number of found CVEs
Vulnerable dependencies	38	24
Non-vulnerable dependencies	0	2
Correctly labeled vulnerable	25	24
Incorrectly labeled non-vulnerable	13	0

Table 5.3: Number of actual vs found CVEs

In table 5.3, we look at the plugin’s proficiency at finding CVE entries associated with a dependency. This table shows the relation between sets of dependencies and how many CVE entries the plugin found for them. If we look specifically at the set of correctly labeled vulnerable dependencies, we see that 24 out of 25 CVE entries were found. This suggests that given that the plugin labels a dependency as vulnerable, the developer can be confident in the CVE entries they are presented with.

### 5.1.2 Version handling

Another important aspect of the plugin’s ability to detect vulnerable dependencies is its version handling. In order to test this, all true positives from the previous test were used as the test data. The false negatives were not included as these would not offer any comparison. All dependencies in this test had their versions updated to fit one of two categories:

- FV = This is a version where some CVE entries from the initial test has been fixed, but the version still contains vulnerabilities.
- FNV = This is a version where all CVE entries from the initial test has been fixed. The version is no longer vulnerable.

The versions were updated and resulted in five dependencies in category FV and eight dependencies in category FNV.

		Actual	
		Vulnerable	Non-vulnerable
Predicted	Vulnerable	TP = 5	FP = 0
	Non-vulnerable	FN = 0	TN = 8

Table 5.4: Updated version vulnerability labels

Table 5.4 shows how the 13 previously vulnerable dependencies were identified after their versions were updated. The performance metrics are once again calculated.

Metric	Formula	Result
Precision	$\frac{TP}{TP+FP}$	1.000
Recall	$\frac{TP}{TP+FN}$	1.000
Accuracy	$\frac{TP+TN}{TP+FP+FN+TN}$	1.000

Table 5.5: Precision, Recall and Accuracy for updated version test

The versions of the dependencies used in the initial test contained a total of 25 CVE entries. The updated versions used for this test contained a total of 9 CVE entries. As a result of the new versions being introduced, 17 CVE entries were correctly filtered out, and 1 new CVE entry was found which did not apply in the initial test. While this test covers a small sample, tables 5.4 and 5.5 tell us that the plugin handles versions perfectly well.

### 5.1.3 Summary of results for RQ2

The results from this test show that seven dependencies were mistakenly labeled as non-vulnerable and as a consequence thirteen CVE entries were not found. It also shows that one CVE entry was not found among the correctly labeled vulnerable dependencies. Finally, one non-vulnerable dependency was mistakenly labeled as vulnerable.

ArtifactId of dependency	Product name in NVD/CPE
commons-io	commons_io
spring-core	spring_framework
jetty-io	jetty
junit	junit4
infinispan-core	infinispan
logback-classic	logback
hibernate-core	hibernate_orm
httpclient	httpclient, commons-httpclient

Table 5.6: Naming mismatch between artifactId and NVD product name

All the shortcomings of these two tests share the same cause. The dependency's maven artifactId does not always correspond to its product name registered in NVD as can be seen in table 5.6. This suggests that relying on artifactId as being the most recognizable name of the product is insufficient. With a more sophisticated method of acquiring the correct product name used by NVD, the plugin would perform significantly better as it has shown that it is capable of finding the correct CVE entries given a correct vulnerability label.

## 5.2 RQ3: Measuring performance of the plugin in terms of runtime using open source maven projects

In testing the runtime performance of the plugin, a plethora of experiments to check various possible factors were run. For every experiment described in this section, all tests at all levels were run three times. In places where all are not presented, the average was used. Runtime was recorded by wrapping the scanning process of the plugin with the following code:

```
long startTime = System.nanoTime();
...execute scanning process here
long endTime = System.nanoTime();
```

The nanosecond result of `*startTime - endTime*` was printed to the screen in an information box after each scan, where it was copied and inserted into a spreadsheet.

### 5.2.1 Testing impact of project size on runtime

For this experiment, three projects of varying sizes were found to determine whether or not the size of a project has any impact on the plugin’s runtime. The projects are presented in table 3.4

To determine what, if any, impact the project size has on runtime, the number of variables is limited by using the exact same set of dependencies for each project.

Project	Dependencies	1st scan	2nd scan	3rd scan	Average
OSS-Maven	40	40.67	40.79	41.19	40.88
Log4j 2	40	41.79	42.70	38.01	40.83
BioJava	40	38.47	42.76	39.60	40.28

Table 5.7: Project runtimes

As clearly showed in table 5.7, the size of the project has no impact on the runtime of the plugin performing a scan.

### 5.2.2 Testing performance on vulnerable vs non-vulnerable dependencies

This experiment is testing if there as a difference in runtime performance between vulnerable and non-vulnerable dependencies. As this is an experiment related to runtime and not soundness or completeness, metrics such as precision and recall are irrelevant in this case. Vulnerable and non-vulnerable dependencies are separated based on what the plugin labels them. With only 13 dependencies labeled vulnerable from the test data used in section 5.1, more test data was added from the results of various tests run during development.

Vulnerable	Dependencies	1st scan	2nd scan	3rd scan	Average
No	15	13.61	15.23	13.53	14.12
Yes	15	19.92	21.99	20.31	20.74

Table 5.8: Vulnerable vs non-vulnerable runtimes

Table 5.8 shows a difference in performance between scanning vulnerable vs non-vulnerable dependencies. With more computing and filtering needed for non-vulnerable dependencies, this is expected. The average difference of 6.62 seconds suggests that vulnerable dependencies take nearly 50% longer to scan. This is however not as dramatic a difference as it might seem due to the fact that the scan of vulnerable and non-vulnerable dependencies average around 1.38 and 0.94 seconds respectively.

### 5.2.3 Testing scalability of plugin as number of dependencies grows

An important feature of any plugin is that it can easily scale. In the case of this plugin, that translates to having dependable and predictable performance as the number of dependencies being scanned increases. For this experiment, four sets of increasing amounts of dependencies was tested.

Dependencies	Composition	1st scan	2nd scan	3rd scan	Average
10	4 V, 6 NV	10.78	10.30	10.75	10.61
20	8 V, 12 NV	22.35	20.80	23.37	22.17
40	16 V, 24 NV	42.63	39.18	40.74	40.85
121	8 V, 113 NV	105.58	102.12	110.04	105.91

Table 5.9: Runtimes of various number of dependencies

The sets of 10, 20 and 40 dependencies all have the same ratio of vulnerable and non-vulnerable dependencies as this has a noticeable impact on performance as shown in section 5.2.2.

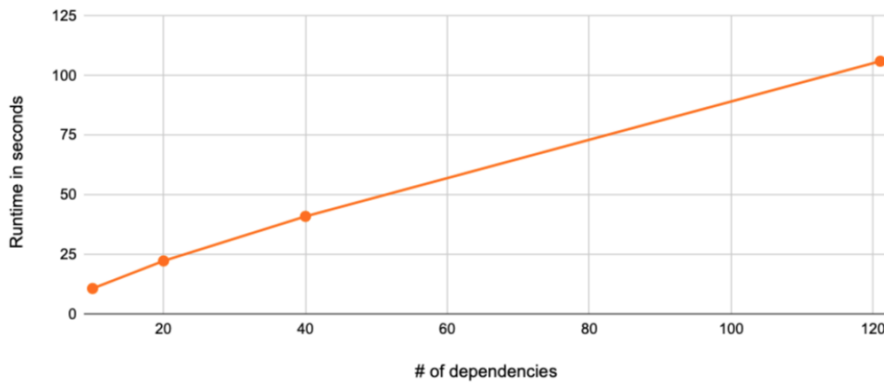


Figure 5.1: Line graph of runtimes of various number of dependencies

As mentioned in chapter 3.3.2, the set of 121 dependencies has a different composition of vulnerable and non-vulnerable dependencies compared to the other three sets. This is reflected in the average runtime being lower per dependency as a result of a larger share of non-vulnerable dependencies. Whereas the sets of 10, 20 and 40 dependencies roughly average a runtime of one second per dependency, the largest set of 121 has an average runtime of 0.87 seconds per dependency. With a similar composition, it is reasonable to assume this set would also average one second per dependency. This experiment shows that there is a linear increase in runtime as the number of dependencies increases, as demonstrated in figure 5.1.

## 5.2.4 Testing impact of storage solution on runtime

The final experiment into the runtime performance of the plugin is to investigate the impact the implemented storage solution has. The previous experiment looking at scalability consisted of deleting the results between each scan as this is an automatically enabled feature. For this experiment, the same tests are performed using the exact same dependencies. The difference is that only after having scanned the set of 40 dependencies, are the scan results deleted, and the tests repeated two more times to get average runtimes.

Dependencies	Composition	1st iteration	2nd iteration	3rd iteration	Average
10	4 V, 6 NV	10.89	10.55	11.22	10.86
20	8 V, 12 NV	13.08	15.16	14.64	14.29
40	16 V, 24 NV	20.91	21.51	21.74	21.38

Table 5.10: Runtimes of various number of dependencies with storage

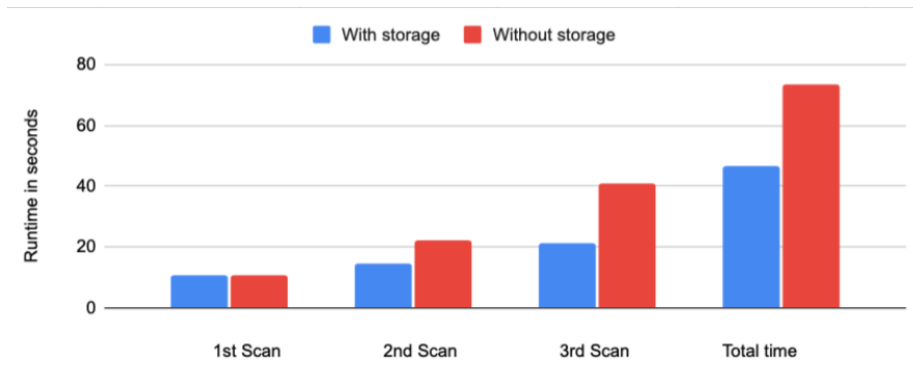


Figure 5.2: Bar chart comparing runtimes with and without storage

The figure above depicts the runtime of each step in a process of scanning 10, 20 and 40 dependencies in order. The solution without storage is fetching data from NVD for all dependencies at each step, whereas the solution with storage is simply building on top of the previous scan.

Figure 5.2 demonstrates that the use of saving and persistently storing the results of a scan gives significantly better performance. Rather than querying NVD for data, the plugin simply fetches the data it already has from the developer's file system. The runtime addition of interacting with the file system is insignificant.

## 5.2.5 Summary of results of RQ3

Throughout section 5.2 we have seen that this plugin provides predictable and moderately scalable performance in all experiments. Neither the number of files, the number of lines of code, nor the memory size of a project has any



impact on runtime. Vulnerable dependencies are shown to take almost 50% longer to scan than non-vulnerable dependencies, but the importance of this is downplayed given the small units of time that are compared. A consistent linear runtime performance is proven to correlate with the number of dependencies being scanned. Finally the impact the storage solution has on the runtime is demonstrated when projects are scanned incrementally over time as more dependencies are added to a project.

### 5.3 Evaluating plugin requirements

To summarize the evaluation, we revisit the requirements table from chapter 3 to see how the plugin turned out compared to its expectations. Each requirement is given a degree of completeness (DOC) rating on a scale from 1 to 5; with 1 being not completed and 5 being fully completed.

Requirement	Type	Related to	DOC
Plugin should link to nvd	Functional	RQ1	5
Plugin should provide developers with information in development environment	Functional	RQ1	5
Plugin should be able to detect vulnerable libraries correctly	Functional	RQ2	3
Plugin should have zero false positives and false negatives	Functional	RQ2	4
Plugin should scale on large projects	Non-Functional	RQ3	3

Table 5.11: Evaluation of requirements

# Chapter 6

## Discussion

In this chapter we take a step back and look at the thesis as a whole. Some challenges encountered during development is discussed which significantly impacted the final product. Next the major limitations of this thesis are addressed and how these limitations could have been prevented.

### 6.1 Challenges

This thesis has had its fair share of challenges as any research is bound to have. Looking back at the last year, the majority of the challenges encountered can be split into two main categories; the scope of the thesis, and the development of the eclipse plugin.

#### 6.1.1 Thesis scope

This is perhaps the most natural challenge of any master's thesis research, and one that is expected and anticipated by both the student as well as the supervisor. Nonetheless its impact on the final product cannot be underestimated. The main goal of the thesis of wanting to improve security during development by integrating public security sources has remained intact throughout the project.

At the start of the project, we envisioned a plugin which looked at the source code at a file level and identified vulnerabilities in developers' own code using machine learning. The information regarding the identified vulnerability would then be linked to relevant information from public security sources. In order to accomplish this, the arduous task of creating a data set for training and evaluating a model was started.

This approach was abandoned in the middle of the project as the scope proved to be larger than expected and due to the fact that we identified a more specific need for vulnerability detection tools for open source components. With more time to work towards the goal we ended up setting for the thesis, it is believed the plugin could have covered a larger pool of project types (currently only supports maven projects) and be used to scan different kinds of open source components.

### 6.1.2 Eclipse plugin development

The second major challenge of this project was the development of the plugin itself - specifically working with the Eclipse Plug-in Development Environment (PDE). While most mature software libraries and frameworks have been used extensively by developers across the world, and also provide detailed documentations for their users, this unfortunately cannot be said about the Eclipse PDE.

When developing a plugin for Eclipse which should interact with their UI and runtime, you need to utilize their API. Having to navigate their documentation online was a large obstacle standing in the way of swift development. Unlike most libraries used in a development environment, packages from the Eclipse PDE did not provide any javadoc legible in the IDE. A very small sample of third party assisting resources such as tutorials and forum posts made for a large portion of time spent trying to understand the framework - time which could have been spent improving the plugin.

During the final stages of development when it came time to deploy and test the plugin, we encountered severe difficulties installing it to our Eclipse environments. This resulted in several days being spent troubleshooting the issue. Only after installing fresh versions of Eclipse, were we able to install the plugin and test it. As these issues were replicated by both student and supervisor, without managing to discover a logical cause for the issue, we were forced to abandon a planned testing phase of the plugin involving distributing it to selected students and professional developers.

## 6.2 Limitations

This thesis and its associated product, the open source vulnerability detection tool has two main limitations. The moderate recall results identified during testing shows that the plugin does not fulfill the expectations we set for it. Second, the plugin has not been tested to the extent we would have liked. This has resulted in assumptions being made which are discussed in this section.

### 6.2.1 Recall results

As shown in the evaluation of this plugin and as mentioned multiple times in the thesis, the plugin does not adequately fulfill the functional requirements of being able to detect vulnerable libraries correctly, and having zero false positives and negatives. When exploring the false predictions made by the plugin, it was evident that they were all caused by the artifactId of the maven dependency not correlating with the product name used by NVD to identify the product.

The assumption that this connection would be sufficient was made when a sample of dependencies from the maven repository was used to compare the two. In the limited sample, the two properties correlated for all the dependencies. The assumption was reinforced by NVDs description of how they choose the names of CPEs - stating that the product attribute "SHOULD describe or identify the most common and recognizable title or name of the product". Tests run during development also returned results suggesting this was a good solution.

In hindsight, it is apparent that too little time was spent testing this assumption. Had it been discovered earlier that our requirements for this plugin called for a more sophisticated method of connecting dependencies to their associated CPE, we could have taken steps to explore other options. Given enough time, this would more than likely have resulted in far better performance regarding soundness and completeness.

### **6.2.2 Lack of testing**

As mentioned briefly in section 6.1.2, we had originally planned to test the plugin by having students and professional developers use the plugin in their own projects. This would have involved the participants receiving the plugin via direct communication and having them provide feedback in the form of a questionnaire. Through this questionnaire, we wanted to receive qualitative data on how the use of the plugin would be perceived by developers in general and to either prove or disprove our assumption that this plugin can help improve security during development.

Without the qualitative data on these topics, we are left having to assume, as mentioned in chapter 3 that developers being informed of vulnerabilities in their dependencies will make them either find different versions/dependencies which are not vulnerable, or have them mitigate the vulnerabilities themselves if possible. This is not an unreasonable assumption to make, but for research purposes it is not sufficient in order to prove the value this plugin can offer.

# Chapter 7

## Related work

At the start of this research, we conducted a research review to discover what aspects of the thesis have been thoroughly investigated, and to uncover what our research can offer to the pool of knowledge. We used the snowballing methodology for discovering research as described by Wohlin [45]. We found articles that provide a good coverage of the relevant research within the following topics; vulnerability detection and classification, use of information sources and developer assistance during development.

### 7.1 Vulnerability detection and classification

The issue of identifying vulnerable software has been approached from a multitude of angles. Shin and Williams [31] look at how the complexity of software is related to software security and how this can be used to predict vulnerabilities. Bosu and Carver [7] investigate the efficiency of peer reviews and how well they function to find security vulnerabilities.

There are many static analysis tools which attempt to find bugs and vulnerabilities by analysing software at various levels of abstraction. Albreiki et al. [4] found in 2014 that these tools are successful to some extent, but are not enough to uncover all weaknesses. Examples of these tools are OWASP's dependency-check [16] and Snyk [32]. Dependency-check is a CLI-tool which can also be integrated in build processes. Much like this thesis, it identifies vulnerable open source components using public security sources, but it is not integrated in the development environment. Snyk offers a multitude of tools for detecting vulnerabilities in both your own code, as well as open source components, but these tools are not available for free.

Another approach showing promising results is the use of machine learning. Scandariato et al. [30] and Hovsepian et al. [20] share a similar approach as they try to identify components of a software containing vulnerabilities by using the source code text itself as features for various machine learning algorithms. Sultana et al. [36] and Medeiros et al. [23] attempt to gather various combinations of software metrics and use these as the features for training models. In a

different study, Sultana et al. [37] do a comparison of the use of software metrics and nano-patterns as features for a number of models. Similarly, Walden et al. [41] compare the uses of software metrics and text mining to identify vulnerabilities.

## 7.2 Use and impact of information sources

In a 2016 study, Acar et al. [3] looked at the impact of information sources on code security. They surveyed android developers with varying levels of experience to see which sources they frequently used to solve problems and found that the sources had very different impacts on code security. In a follow up study in 2017, Acar et al. [2] performed a survey to try to understand the ecosystem of security advice software developers are using. Similarly in a 2009 paper, Brandt et al. [8] performed a study where they looked at how developers interleave web foraging, learning and writing code in their routines.

Wijayasekara et al. [44][43] attempted during their two studies in 2012 and 2014 to identify hidden impact bugs (HIBs), bugs identified as vulnerabilities long after they are made public, by text mining bug databases. In 2018, Sauerwein et al. [29] performed an analysis of how public security sources are used in research and practice. In this analysis they identified 68 different PSS and classified them based on their selected dimensions. Using PSS, Salen [27] attempted in 2021 to classify discussion forum posts as security or not-security related. She also showed how PSS can provide an additional layer of context for the forum posts.

## 7.3 Developer assistance during development

One of the more researched areas of helping developers with their coding during development, is the concept of connecting various aspects of the development environment to online resources. In 2014, Ponzanelli et al. [26] developed a plug-in for the Eclipse IDE which sought to offer developers knowledge relevant to their work, directly in the development environment. The plug-in used text-mining to find pertinent discussions from Stack Overflow so that the developer wouldn't have to spend time on formulating their problems as queries. Before this, Cordeiro et al. [11] performed an almost identical study in 2012 where they used exception stacks from Eclipse to retrieve helpful resources from the web and presented this data directly in Eclipse. Prior to this even, Goldman and Miller [19] developed a system which connected Eclipse and Firefox so that the two platforms could benefit from each others' information such as editing history in Eclipse and recent browsing in Firefox. The system used the link to develop a number of tools such as having the web browser focus on the developer's current context from the IDE.

Another development assistance approach is to help the developer by providing more general knowledge which many might not always remember. Xie et al. [46] developed a proof-of-concept plug-in for Eclipse in 2011, which reminded developers of secure programming practices directly inside the IDE. A similar approach was used by Whitney et al. [42] in 2015 when they made an IDE tool which would integrate secure coding education to provide learning opportuni-

ties.

## Chapter 8

# Conclusion

There is a need for more tools to help developers mitigate the ever growing amount of security vulnerabilities that follow the use of open source components. This approach focuses on bringing certified good resources and developers closer together at a stage in the development where it can have the most significant impact in preventing security vulnerabilities.

The thesis has developed a prototype plugin for the Eclipse development environment, which has a lot of potential given some further development. The results of various evaluations show a recall of 65%, a precision of 93% and an accuracy of 80%. The plugin has a well functioning connection to NVD, a vulnerability database for open source libraries. We have data structures which facilitate fast and simple handling of the abundance of information returned with each CVE entry. If more data is desired in the views presented to developers, the models can easily be expanded upon to hold more relevant information.

A storage solution has been implemented which considers how developers work differently from one another. Developers working with security-critical components might want to check for new data every time they perform a scan, while other might want slightly faster performance by only updating data they consider to be outdated. This option is available to them through the properties of the plugin, which as almost all other parts of this plugin, is easily expandable if we wanted to offer developers more customization options. Finally, we are left with a plugin which has proven to be moderately scalable with large projects - ensuring that it is attractive to developers regardless of the size of their projects.

The future work for this plugin and the research surrounding it needed to elevate the value of both, is centered around improving the performance of the plugin and performing more tests to validate its usefulness. As mentioned in chapter 6, we would have liked to perform a qualitative test by having real world developers trying the plugin and providing useful feedback. Before potentially reaching that point, we need to make sure that they can be confident in the results produced by the plugin. Utilizing more search features, and expanding upon the method for acquiring the most recognizable name for a product would better connect the dependencies of a project to the growing amount of data offered by NVD.



This thesis is contributing to shed more light on an area of software security which certainly needs more tools and research looking into it than it currently has. The extent of how much the use of open source components in software development is growing has been mentioned, and also how prevalent vulnerabilities in these components are. The plugin produced helps raise a valid argument that developers need more tools at their disposal during the development stages, and specifically targeted at the parts of their software that they are not responsible for creating themselves.

# Literature and References

- [1] @frodriguez. *MVN Repository*. URL: <https://mvnrepository.com> (visited on May 19, 2022).
- [2] Yasemin Acar et al. “Developers Need Support, Too: A Survey of Security Advice for Software Developers.” In: *2017 IEEE Cybersecurity Development (SecDev)*. 2017, pp. 22–26. DOI: 10.1109/SecDev.2017.17.
- [3] Yasemin Acar et al. “You Get Where You’re Looking for: The Impact of Information Sources on Code Security.” In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 289–305. DOI: 10.1109/SP.2016.25.
- [4] Hamda Hasan AlBreiki and Qusay H. Mahmoud. “Evaluation of static analysis tools for software security.” In: *2014 10th International Conference on Innovations in Information Technology (IIT)*. 2014, pp. 93–98. DOI: 10.1109/INNOVATIONS.2014.6987569.
- [5] Bill F. Appelbe. “The Future of Open Source Software.” In: *J. Res. Pract. Inf. Technol.* 35 (2003), pp. 227–236.
- [6] Unknown author. *What is open source*. URL: <https://opensource.com/resources/what-open-source> (visited on May 11, 2022).
- [7] Amiangshu Bosu and Jeffrey C. Carver. “Peer Code Review to Prevent Security Vulnerabilities: An Empirical Evaluation.” In: *2013 IEEE Seventh International Conference on Software Security and Reliability Companion*. 2013, pp. 229–230. DOI: 10.1109/SERE-C.2013.22.
- [8] Joel Brandt et al. “Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code.” In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’09. Boston, MA, USA: Association for Computing Machinery, 2009, pp. 1589–1598. ISBN: 9781605582467. DOI: 10.1145/1518701.1518944. URL: <https://doi.org/10.1145/1518701.1518944>.
- [9] Brant A. Cheikes, David Waltermire, and Karen Scarfone. *Common Platform Enumeration: Naming Specification Version 2.3*. Tech. rep. National Institute of Standards and Technology, 2011.
- [10] Brant A. Cheikes, David Waltermire, and Karen Scarfone. *Common Platform Enumeration: Naming specification Version 2.3*. URL: <https://nvlpubs.nist.gov/nistpubs/Legacy/IR/nistir7695.pdf> (visited on May 21, 2022).
- [11] Joel Cordeiro, Bruno Antunes, and Paulo Gomes. “Context-based recommendation to support problem solving in software development.” In: *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*. 2012, pp. 85–89. DOI: 10.1109/RSSE.2012.6233418.

- [12] The MITRE Corporation. *CVE Website*. URL: <https://www.cve.org/> (visited on May 9, 2022).
- [13] The MITRE Corporation. *CWE Website*. URL: <https://cwe.mitre.org/> (visited on May 9, 2022).
- [14] *Cucumber*. URL: <https://cucumber.io> (visited on May 30, 2022).
- [15] Michael Felderer et al. “Security Testing: A Survey.” In: Mar. 2016, pp. 1–51. ISBN: 9780128051580. DOI: 10.1016/bs.adcom.2015.11.003.
- [16] The OWASP foundation. *OWASP Website*. URL: <https://owasp.org/www-project-dependency-check/> (visited on May 28, 2022).
- [17] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. “Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey.” In: *ACM Comput. Surv.* 50.4 (Aug. 2017). ISSN: 0360-0300. DOI: 10.1145/3092566. URL: <https://doi.org/10.1145/3092566>.
- [18] Alexander S. Gillis. *Static analysis*. URL: <https://www.techtarget.com/whatis/definition/static-analysis-static-code-analysis> (visited on May 30, 2022).
- [19] Max Goldman and Robert C. Miller. “Codetrail: Connecting source code and web resources.” In: *Journal of Visual Languages & Computing* 20.4 (2009). Special Issue on Best Papers from VL/HCC2008, pp. 223–235. ISSN: 1045-926X. DOI: <https://doi.org/10.1016/j.jvlc.2009.04.003>. URL: <https://www.sciencedirect.com/science/article/pii/S1045926X09000263>.
- [20] Aram Hovsepyan et al. “Software Vulnerability Prediction Using Text Analysis Techniques.” In: *Proceedings of the 4th International Workshop on Security Measurements and Metrics*. MetriSec ’12. Lund, Sweden: Association for Computing Machinery, 2012, pp. 7–10. ISBN: 9781450315081. DOI: 10.1145/2372225.2372230. URL: <https://doi.org/10.1145/2372225.2372230>.
- [21] *JUnit*. URL: <https://junit.org/junit5/> (visited on May 30, 2022).
- [22] Matt. *False positive and false negative in software testing*. URL: <https://testfully.io/blog/false-positive-false-negative/> (visited on May 30, 2022).
- [23] Nádia Medeiros et al. “Vulnerable Code Detection Using Software Metrics and Machine Learning.” In: *IEEE Access* 8 (2020), pp. 219174–219198. DOI: 10.1109/ACCESS.2020.3041181.
- [24] *Mockito*. URL: <https://site.mockito.org> (visited on May 30, 2022).
- [25] OpenUK. *State of Open: The UK in 2021*. Tech. rep. OpenUK, 2021.
- [26] Luca Ponzanelli et al. “Mining StackOverflow to Turn the IDE into a Self-Confident Programming Prompter.” In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 102–111. ISBN: 9781450328630. DOI: 10.1145/2597073.2597077. URL: <https://doi.org/10.1145/2597073.2597077>.
- [27] Anja Fonn Salen. “Utilizing public repositories to improve the decision process for security defect resolution and information reuse in the development environment.” MA thesis. The University of Bergen, 2021.
- [28] Luciano Sampaio. *Early Vulnerability Detection for Supporting Secure Programming (ESVD)*. URL: <https://thecodemaster.net/early->

- vulnerability-detection-supporting-secure-programming/ (visited on May 30, 2022).
- [29] Clemens Sauerwein et al. “An analysis and classification of public information security data sources used in research and practice.” In: *Computers & Security* 82 (2019), pp. 140–155. ISSN: 0167-4048. DOI: <https://doi.org/10.1016/j.cose.2018.12.011>. URL: <https://www.sciencedirect.com/science/article/pii/S0167404818304978>.
  - [30] Riccardo Scandariato et al. “Predicting Vulnerable Software Components via Text Mining.” In: *IEEE Transactions on Software Engineering* 40.10 (2014), pp. 993–1006. DOI: 10.1109/TSE.2014.2340398.
  - [31] Yonghee Shin and Laurie Williams. “An Empirical Model to Predict Security Vulnerabilities Using Code Complexity Metrics.” In: *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM '08*. Kaiserslautern, Germany: Association for Computing Machinery, 2008, pp. 315–317. ISBN: 9781595939715. DOI: 10.1145/1414004.1414065. URL: <https://doi.org/10.1145/1414004.1414065>.
  - [32] *Snyk website*. URL: <https://docs.snyk.io/introducing-snyk> (visited on May 28, 2022).
  - [33] SonarSource. *SonarQube*. URL: <https://www.sonarqube.org> (visited on May 30, 2022).
  - [34] *Spotbugs*. URL: <https://spotbugs.github.io> (visited on May 30, 2022).
  - [35] National Institute of Standards and Technology - NIST. *NVD Website*. URL: <https://nvd.nist.gov/> (visited on May 9, 2022).
  - [36] Kazi Zakia Sultana, Vaibhav Anu, and Tai-Yin Chong. “Using software metrics for predicting vulnerable classes and methods in Java projects: A machine learning approach.” In: *Journal of Software: Evolution and Process* 33.3 (2021). e2303 smr.2303, e2303. DOI: <https://doi.org/10.1002/smr.2303>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2303>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2303>.
  - [37] Kazi Zakia Sultana, Byron J. Williams, and Amiangshu Bosu. “A Comparison of Nano-Patterns vs. Software Metrics in Vulnerability Prediction.” In: *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. 2018, pp. 355–364. DOI: 10.1109/APSEC.2018.00050.
  - [38] Inc Synopsis. *What is open source software*. URL: <https://www.synopsys.com/glossary/what-is-open-source-software.html> (visited on May 11, 2022).
  - [39] Inc. Synopsis. *Open Source Security and Risk Analysis Report*. Tech. rep. Synopsys, Inc., 2021.
  - [40] Inc. Synopsis. *Open Source Security and Risk Analysis Report*. Tech. rep. Synopsys, Inc., 2022.
  - [41] James Walden, Jeff Stuckman, and Riccardo Scandariato. “Predicting Vulnerable Components: Software Metrics vs Text Mining.” In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 2014, pp. 23–33. DOI: 10.1109/ISSRE.2014.32.
  - [42] Michael Whitney et al. “Embedding Secure Coding Instruction into the IDE: A Field Study in an Advanced CS Course.” In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education. SIGCSE '15*. Kansas City, Missouri, USA: Association for Computing Machinery,

- 2015, pp. 60–65. ISBN: 9781450329668. DOI: 10.1145/2676723.2677280. URL: <https://doi.org/10.1145/2676723.2677280>.
- [43] Dumidu Wijayasekara, Milos Manic, and Miles McQueen. “Vulnerability identification and classification via text mining bug databases.” In: *IECON 2014 - 40th Annual Conference of the IEEE Industrial Electronics Society*. 2014, pp. 3612–3618. DOI: 10.1109/IECON.2014.7049035.
- [44] Dumidu Wijayasekara et al. “Mining Bug Databases for Unidentified Software Vulnerabilities.” In: *2012 5th International Conference on Human System Interactions*. 2012, pp. 89–96. DOI: 10.1109/HSI.2012.22.
- [45] Claes Wohlin. “Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering.” In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. EASE ’14. London, England, United Kingdom: Association for Computing Machinery, 2014. ISBN: 9781450324762. DOI: 10.1145/2601248.2601268. URL: <https://doi.org/10.1145/2601248.2601268>.
- [46] Jing Xie et al. “ASIDE: IDE Support for Web Application Security.” In: *Proceedings of the 27th Annual Computer Security Applications Conference*. ACSAC ’11. Orlando, Florida, USA: Association for Computing Machinery, 2011, pp. 267–276. ISBN: 9781450306720. DOI: 10.1145/2076732.2076770. URL: <https://doi.org/10.1145/2076732.2076770>.
- [47] Zebanza. *9 reasons to choose Open Source*. URL: <https://www.zebanza.be/why-open-source/> (visited on May 11, 2022).

# Appendices

# Appendix A

## Source code

The source code for the plugin can be found at the following link:

<https://github.com/sivertlunde/MasterThesis.git>

## Appendix B

# Plugin documents

This links to a google drive folder containing the plugin as an installable .jar file in a zipped format, installation instructions and a user guide:

<https://drive.google.com/drive/folders/1fZmO1CciKRZV6NkamvaXWu6gIIALaMZz?usp=sharing>



## Appendix C

# NVD API JSON response schema

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "JSON Schema for NVD Vulnerability Data Feed version 1.1",
  "id": "https://scap.nist.gov/schema/nvd/feed/1.1/nvd_cve_feed_json_1.1.schema",
  "definitions": {
    "def_cpe_name": {
      "description": "CPE name",
      "type": "object",
      "properties": {
        "cpe22Uri": {
          "type": "string"
        },
        "cpe23Uri": {
          "type": "string"
        },
        "lastModifiedDate": {
          "type": "string"
        }
      }
    },
    "required": [
      "cpe23Uri"
    ]
  },
  "def_cpe_match": {
    "description": "CPE match string or range",
    "type": "object",
    "properties": {
      "vulnerable": {
        "type": "boolean"
      },
      "cpe22Uri": {
```

```

        "type": "string"
    },
    "cpe23Uri": {
        "type": "string"
    },
    "versionStartExcluding": {
        "type": "string"
    },
    "versionStartIncluding": {
        "type": "string"
    },
    "versionEndExcluding": {
        "type": "string"
    },
    "versionEndIncluding": {
        "type": "string"
    },
    "cpe_name": {
        "type": "array",
        "items": {
            "$ref": "#/definitions/def_cpe_name"
        }
    }
},
"required": [
    "vulnerable",
    "cpe23Uri"
]
},
"def_node": {
    "description": "Defines a node or sub-node in an NVD applicability statement."
    "properties": {
        "operator": {"type": "string"},
        "negate": {"type": "boolean"},
        "children": {
            "type": "array",
            "items": {"$ref": "#/definitions/def_node"}
        },
        "cpe_match": {
            "type": "array",
            "items": {"$ref": "#/definitions/def_cpe_match"}
        }
    }
},
"def_configurations": {
    "description": "Defines the set of product configurations for a NVD applicabil
    "properties": {
        "CVE_data_version": {"type": "string"},
        "nodes": {
            "type": "array",

```

```

        "items": {"$ref": "#/definitions/def_node"}
    }
},
"required": [
    "CVE_data_version"
]
},
"def_subscore": {
    "description": "CVSS subscore.",
    "type": "number",
    "minimum": 0,
    "maximum": 10
},
"def_impact": {
    "description": "Impact scores for a vulnerability as found on NVD.",
    "type": "object",
    "properties": {
        "baseMetricV3": {
            "description": "CVSS V3.x score.",
            "type": "object",
            "properties": {
                "cvssV3": {"$ref": "cvss-v3.x.json"},
                "exploitabilityScore": {"$ref": "#/definitions/def_subscore"},
                "impactScore": {"$ref": "#/definitions/def_subscore"}
            }
        },
        "baseMetricV2": {
            "description": "CVSS V2.0 score.",
            "type": "object",
            "properties": {
                "cvssV2": {"$ref": "cvss-v2.0.json"},
                "severity": {"type": "string"},
                "exploitabilityScore": {"$ref": "#/definitions/def_subscore"},
                "impactScore": {"$ref": "#/definitions/def_subscore"},
                "acInsufInfo": {"type": "boolean"},
                "obtainAllPrivilege": {"type": "boolean"},
                "obtainUserPrivilege": {"type": "boolean"},
                "obtainOtherPrivilege": {"type": "boolean"},
                "userInteractionRequired": {"type": "boolean"}
            }
        }
    }
},
"def_cve_item": {
    "description": "Defines a vulnerability in the NVD data feed.",
    "properties": {
        "cve": {"$ref": "CVE_JSON_4.0_min_1.1.schema"},
        "configurations": {"$ref": "#/definitions/def_configurations"},
        "impact": {"$ref": "#/definitions/def_impact"},
        "publishedDate": {"type": "string"},
    }
}

```

```

        "lastModifiedDate": {"type": "string"}
    },
    "required": ["cve"]
}
},
"type": "object",
"properties": {
    "CVE_data_type": {"type": "string"},
    "CVE_data_format": {"type": "string"},
    "CVE_data_version": {"type": "string"},
    "CVE_data_numberOfCVEs": {
        "description": "NVD adds number of CVE in this feed",
        "type": "string"
    },
    "CVE_data_timestamp": {
        "description": "NVD adds feed date timestamp",
        "type": "string"
    },
    "CVE_Items": {
        "description": "NVD feed array of CVE",
        "type": "array",
        "items": {"$ref": "#/definitions/def_cve_item"}
    }
},
"required": [
    "CVE_data_type",
    "CVE_data_format",
    "CVE_data_version",
    "CVE_Items"
]
}

```