Tracking known security vulnerabilities in third-party components

Master's Thesis

Mircea Cadariu

Tracking known security vulnerabilities in third-party components

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Mircea Cadariu
born in Brasov, Romania

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Software Improvement Group
Rembrandt Tower, 15th floor
Amstelplein 1 - 1096HA
Amsterdam, the Netherlands
www.sig.eu

©2014 Mircea Cadariu. All rights reserved.

Tracking known security vulnerabilities in third-party components

Author: Mircea Cadariu
Student id: 4252373
Email: m.cadariu@tudelft.nl

Abstract

Known security vulnerabilities are introduced in software systems as a result of depending on third-party components. These documented software weaknesses are hiding in plain sight and represent the lowest hanging fruit for attackers. Despite the risk they introduce for software systems, it has been shown that developers consistently download vulnerable components from public repositories. We show that these downloads indeed find their way in many industrial and open-source software systems. In order to improve the status quo, we introduce the Vulnerability Alert Service, a tool-based process to track known vulnerabilities in software projects throughout the development process. Its usefulness has been empirically validated in the context of the external software product quality monitoring service offered by the Software Improvement Group, a software consultancy company based in Amsterdam, the Netherlands.

Thesis Committee:

Chair: Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor: Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
Company supervisor: Prof. Dr. Joost Visser, Software Improvement Group
Company co-supervisor: Dr. Eric Bouwers, Software Improvement Group
Committee Member: Prof. Dr. Jan van den Berg, Faculty TBM, TU Delft
Committee Member: Assoc. Prof. Dr. Andy Zaidman, Faculty EEMCS, TU Delft

Preface

While the name written on the first page is mine, I am only partially responsible for the document you are currently holding in your hand. It is the output of many peopole's efforts and dedication and I would like to express my gratitude in the following lines.

I would like to thank Prof. Dr. Joost Visser, for welcoming me to conduct my master thesis in the form of an internship at the Software Improvement Group (SIG) and for his advice at multiple stages of the project. At SIG, I am very grateful to have had the opportunity to work closely with Dr. Eric Bouwers, my daily supervisor, whose guidance throughout the project I sincerely appreciate. I also thank Prof. Arie van Deursen, for his academic supervision, thorough review of the thesis document and for all the Friday morning discussions which have always resulted in the discovery of new avenues to explore within the boundaries of my research topic.

It has been a great pleasure for me to grow professionally and personally in the SIG environment surrounded by so many highly skilled professionals. A warm "thank you" in this regard goes to Dennis Bijlsma, Theodoor Scholte and Axel Eissens, who have helped me with technological aspects throughout my internship and provided me with valuable improvement suggestions for the thesis document. I would also like to express my gratitude for the numerous conversations with the members of the research department, which have always had the result of challenging me with new perspectives. I also grew fond of the spirit of camaraderie that was present among our team of interns.

Finally, I would also like to thank my parents and my girlfriend Merete, for their continuous support and motivation.

Mircea Cadariu
Delft, the Netherlands
August 27, 2014

Contents

Preface

1 Introduction
1.1 Definitions 3
1.2 Research Context 4
1.3 Problem Statement 5
1.4 Research Method 5
1.5 Research Questions 5
1.6 Thesis structure 6

2 Requirements for a method to track vulnerable components
2.1 Research Subjects 7
2.2 Interview phases 8
2.3 The requirements table 8
2.4 Discussion 9
2.5 Conclusion 10

3 Vulnerability Knowledge Providers
3.1 Database Selection 11
3.2 Vulnerability Information Elements 12
3.3 Comparison Matrix 13
3.4 Discussion 13
3.5 Interviewing 15
3.6 Conclusion 16

4 The Vulnerability Alert Service
4.1 Process description 18
4.2 The Vulnerability Checker 20
4.3 Vulnerability checker reliability 25
4.4 Conclusion 29

5 Known Vulnerabilities in software project dependencies
5.1 Research Questions 31
5.2 Software Subjects 31
5.3 Tooling 32

5.4 Maven study design 32
5.5 Maven study results 34
5.6 Proprietary projects study design 36
5.7 Proprietary projects study results 36
5.8 Conclusion 39

6 Evaluating the Vulnerability Alert Service
6.1 Study design 41
6.2 Interview Findings 44
6.3 Observations Findings 45
6.4 Discussion 46
6.5 Threats to validity 46
6.6 Conclusion 47

7 Related Work
7.1 Known Vulnerabilities 49
7.2 Empirical Studies on Known Vulnerabilities 50
7.3 The Security of Maven Components 50

8 Conclusions and Future Work
8.1 Answering the research questions 53
8.2 Future Work 55

Bibliography

A Interview-Requirements Introductory Text

B Interview Form

C Requirements Formulation and Justification

D Interview - Vulnerability Databases
D.1 Introduction 73
D.2 Result 74

List of Figures

1.1 Projects,third-party components and known vulnerabilities 2
1.2 Vulnerability Lifecycle 3

4.1 The Vulnerability Alert Service 17
4.2 DependencyCheck scanning process 22
4.3 Exaple POM file 24
4.4 Jetty 6.1.20 in the POM file 24
4.5 Process for obtaining the ground truth dataset for recall study 26

5.1 The study execution procedure 33
5.2 Frequency Distributions for number of components and number of vulnerable components 35
5.3 Vulnerable vs Clean projects with regards to known vulnerabilities in their dependencies 37
5.4 Distribution of the number of vulnerable libraries across proprietary software projects 37

6.1 Usefulness Evaluation Study Design 42
6.2 Useful vs. Not Useful Alerts 45

Chapter 1

Introduction

Nowadays, software underpins most of the important transactions that enable current businesses. These transactions have to be conducted in a safe and secure manner, otherwise malicious attackers can leverage them to their own needs, which often results in severe negative consequences for the users and product owners. Damages range from big financial loss to losing the clients' trust. One of the most resonating examples to illustrate the potential losses comes from the electronic payments domain - in 2008, the payment processor of Heartland Payment Systems suffered from an SQL Injection attack and relinquished access to 134 million credit and debit cards [5]. The company lost 50% of its market capitalization and spent over $42 million on settlement costs.

Given these potential losses, protection against security breaches is of paramount importance. Traditional security defense mechanisms, such as anti-virus scanners, firewalls or intrusion detectors, do not directly tackle the security problem, they only "clean up the mess that unsecure software leaves" [23]. Therefore, these software tools target only the symptoms of insecure software. Exactly like with the wellness of our health, fighting disease symptoms, while neglecting the immune system in the long term is not adequate. To improve the immune system, effort should be invested into developing more secure software in the first place, through involving strong security awareness right in the software development process [23].

The *OWASP Top Ten* [10] is one initiative that promotes security awareness in the context of software development. It assembles and describes the most critical software security flaws that expose applications to malicious threats. Amongst its entries, we find at the time of writing, *Using Components with Known Vulnerabilities* [8], as a result of the exploitation risk they introduce. Despite this guideline, approximately 1 out of 4 library downloads from the Maven Central Repository features a software component with a known vulnerability [75], despite OWASP's guidelines that suggests avoiding them.

Known vulnerabilities, besides heavily downloaded, are also easy to track by attackers. To illustrate this problem, consider Shodan[1], a search engine which has been shown to be useful in identifying Internet-facing industrial control systems [17]. In our context, it can show the specific web server implementations that underpin active web applications. A

[1] http://www.shodanhq.com/, accessed May 2014

simple keyword search query on Shodan containing the term *Jetty 6.1.1* retrieves the Internet addresses of 464 hosts that expose their online services using this open-source web server [67] which features several known vulnerabilities (at the time of writing, associated with this component are 7 known vulnerabilities[1], among which one is considered high severity[2]). The same mechanism can be used to enlist the machines which are suffering from the Heartbleed vulnerability[3], by querying for *openssl*. Thus, the ingredients for an automated large scale exploit that targets hosts which run on third-party software with known vulnerabilities are set. We can conclude that known vulnerabilities lower the bar for the skill required to successfully exploit software systems, and malicious attackers do not think twice before exploiting this low hanging fruit. Actually, studies show an increase in exploits after the vulnerability disclosure as high as three orders of magnitude [16].

This is the status quo that we aim to challenge. Thus, the goal of this thesis is to propose a method to continuously track known vulnerabilities in third party components of software systems and assess its usefulness in a relevant context. The driving thought is that known vulnerabilities are indeed widespread and easy to exploit, but they can also be leveraged for benign purposes, such as to improve security awareness and ultimately determine corrective measures that reduce the opportunity for security breaches.

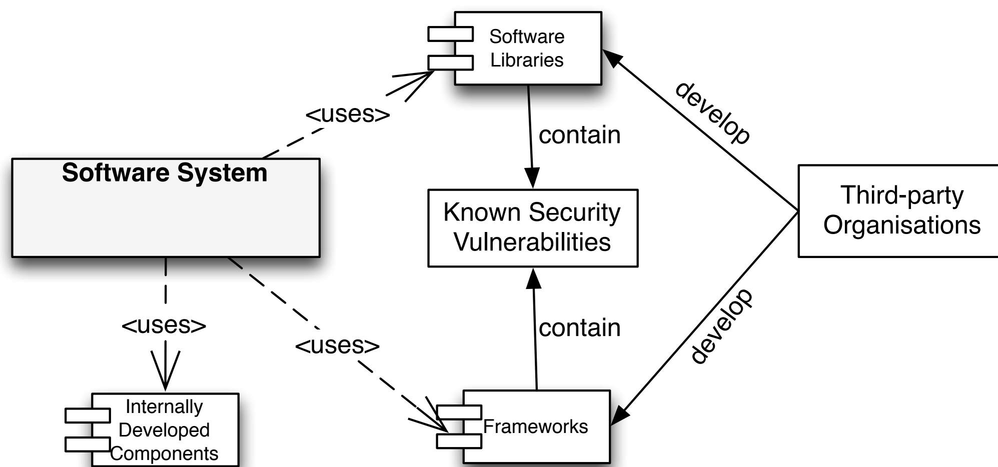

Figure 1.1: Projects,third-party components and known vulnerabilities

[1] http://nvd.nist.gov/view/vuln/search-results?adv_search=true&cpe=cpe%3a%2fa%3amortbay%3ajetty%3a6.1.1%3arc0
[2] http://nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-4611
[3] http://heartbleed.com/

1.1 Definitions

Nowadays, software systems include multiple components, some of which are developed "in-house", or by third-party organizations. Some of the third-party components have known vulnerabilities, as shown in Figure 1.1. In this context, we explain *known vulnerability* and *component* as used in the scope of this thesis.

1.1.1 Known Vulnerabilities

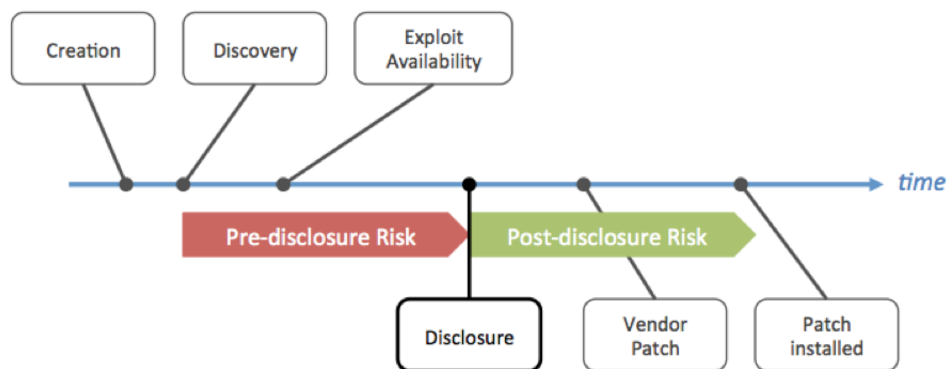

Figure 1.2: Vulnerability Lifecycle

According to the ISO 27005 standard, a vulnerability is *"A weakness of an asset or group of assets that can be exploited by one or more threats"*[34]. An asset is *"anything that can have value to the organization, its business operations and their continuity, including information resources that support the organization's mission"*[33]. Vulnerabilities have a lifecycle, which starts from its creation when it is introduced in the source code by the developer, and finishes when a patch for it is installed, illustrated in Figure 1.2 [28]. In the context of this thesis, we focus on the *Post-disclosure Risk* segment, vulnerabilities that have been disclosed and therefore considered *publicly known*. Among these, we consider vulnerabilities which have been assigned a CVE identifier, the most authoritative standardization scheme for known vulnerabilities.

The Common Vulnerabilities and Exposures (CVE) identifiers were introduced in 1999 to enable interoperability and comparison between the existing security tools at the time [1] [40]. They are currently maintained by MITRE [7], an American not-for-profit organization that has the responsibility, among others, to manage the research and federally funded development centers.

1.1.2 Components

In the literature we can find many definitions for a software component. The most charismatic comes from Michael Feathers: *"A **component** is an object in a tuxedo. That is, a*

piece of software that is dressed to go out and interact with the world" [11]. On a more pragmatic note, Clements defines it as being *"an implementation unit of software that provides a coherent unit of functionality"* [24]. Additionally, Szyperski stated that *"software components enable practical reuse of software parts and amortization of investments over multiple applications"* [70].

Within this thesis, we focus on *third-party* software components as an instantiation of the definitions provided by Clements and Szyperski. Therefore, we target components developed within an external organization. They may be proprietary or open-source. Examples of commonly used third-party components belonging to the Java ecosystem include libraries such as Log4j[1] but also frameworks such as Apache Struts[2]. We use Martin Fowler's definition of libraries and frameworks [**?**]:

*A **library** is essentially a set of functions that you can call, these days usually organized into classes. Each call does some work and returns control to the client.*

*A **framework** embodies some abstract design, with more behavior built in. In order to use it you need to insert your behavior into various places in the framework either by subclassing or by plugging in your own classes. The framework's code then calls your code at these points.*

1.2 Research Context

The research presented in this thesis has been conducted within the Software Improvement Group (SIG), a consultancy company based in Amsterdam, the Netherlands. Among its services, it provides Software Risk Assessment [72], Software Monitoring [38] [39] and Security Risk Assessment [77]. The professionals that enable SIG's consultancy services take the role of external quality evaluators for the software systems developed by the client companies. The client companies range from banks to national public transportation companies and governmental organizations.

The Software Monitoring service is used to track the evolution of the product quality indicators throughout the software product's development lifecycle [15]. At regular intervals, representatives of the client companies upload the most recent version of the code base. Alerts are produced on the detection of noteworthy maintainability specific measurements deviations from the previously uploaded snapshot.

The alerts are aggregated in a user interface called the Monitor Control Center (MCC). The MCC is staffed by technical consultants who after validating alert usefulness proceed in conducting a root cause analysis of the signalled quality deviation. In case the root cause analysis results in concrete and actionable findings, the technical consultants proceed to communicate them to the designated responsibles within the client company.

[1] http://logging.apache.org/log4j/2.x/
[2] http://struts.apache.org//

1.3 Problem Statement

We formulate our research goal using the the Goal/Question/Metric template [14]:

In this project, we aim to improve the the process of detecting security vulnerabilities in software projects caused by known vulnerabilities in projects' third party components, from the viewpoint of a software evaluator, in the context of evaluating external software systems.

In the context of external quality evaluations, investigations to find vulnerable components are tedious due to the moving parts inherent to the process – a multitude of projects, programming ecosystems, components and vulnerabilities.

The pull-based method in which, at regular intervals, projects are proactively inspected does not scale for this context. What scales, is the opposite approach, push-based, in which events are generated that draw attention to the elements worthy of investigation. In line with the maintainability oriented approach described in Section 1.2, these events can be called *vulnerability alerts*, and can signal the presence of a known vulnerability in a dependency used in a software project. Our aim is to construct a tool-based mechanism that produces such alerts to occur and evaluate their added value in our research context.

We consider the research to be successful if we gather enough evidence to support the claim that the generated events are *useful* for the context from which they emerged. Evidence of usefulness are indicators that the process has indeed improved, at least in terms of efficiency, as parts of the process have been automated.

1.4 Research Method

Research activities in software engineering are differentiated by the type of problem they tackle [26]. With *knowledge problems*, the research approach entails collecting evidence that fails to refute knowledge claims, through using, for example, controlled experiments in which theories are probed. The other type of research topics are the so-called *practical problems*, solved in a specific context, but for which analysis is conducted to understand the potential to be generalized across the immediate research context. Our research task matches the latter case. Therefore, we employ its corresponding research method.

Following Wieringa et al.'s guidelines for reporting on a practical problems [73][74], we separate the research process in four phases: (1) problem analysis, (2) requirements analysis, (3) solution design and validation and (4) solution evaluation. We group the research questions according to these steps.

1.5 Research Questions

Our main research question is:

- ***How can we automatically produce useful alerts triggered by vulnerable components found in software projects?***

In turn, answering this question means answering the following sub-questions:

- **Requirements Analysis**
  - *Which design considerations are important for methods to track components with known vulnerabilities?*
- **Solution Design and Validation**
  - *Where can we find vulnerability data?*
  - *How can we produce alerts on discovering vulnerable components in software projects?*
- **Solution Evaluation**
  - *Are the produced alerts useful within the research context?*

1.6 Thesis structure

Each research phase introduced in Section 1.4 is discussed in a separate thesis chapter. This chapter is the output of the Problem Analysis phase. In Chapter 2, we present the important design considerations that emerged from the Requirements Analysis research phase. In Chapter 3, we survey the sources from which vulnerability information can be obtained through manual or automatic means. The Vulnerability Alert Service, our tool-based process to track known vulnerabilities in third-party components of software projects in the context of external software quality monitoring is presented in Chapter 4. The rest of the chapters contain empirical studies that focus either on the prevalence of the problem of depending on components with known vulnerabilities in practice or on the usefulness of the proposed solution in the research context.

Chapter 2

Requirements for a method to track vulnerable components

Using the GQM method template [14], we define the second phase in our research project:

> *Our goal is to understand the important requirements for solutions to track vulnerable components from the viewpoint of external quality evaluators in the context of external software quality evaluations.*

The mechanism through which we gather this information is interviews with the potential users of the method selected from our research context. Researcher-administered interviewing was selected as a method as it is a efficient way to capture the users' opinion on the matter. We borrow Polychniatis's interviewing procedure as it was put in practice with good results within the same context [53]. This chapter presents the outcome of this study.

2.1 Research Subjects

The SIG consultancy services are executed by two types of professionals: technical and general consultants. Technical consultants analyse software and report (technical) findings. To reduce the workload an alert service performs an automated scan for noteworthy changes every time new source code is uploaded, as descrined in Section 1.2. Together with the general consultants they translate these findings into practical advice for customers. Four technical consultants and three general consultants were selected to participate in this stage of the research process as subjects. The (slightly) larger number of technical consultants compared to the number of general consultants can be attributed to the fact that monitoring the presence of known vulnerabilities in third-party components of software projects falls under the responsibility of the technical department and is in its nature a technical aspect of software development. Nonetheless, we also wanted to capture the general consultant's opinion on the matter, so we also invited a comparable number for our interviews.

2.2 Interview phases

Each interview meeting consisted of three phases: an *introduction* phase followed by two phases, one for *gathering requirements* and one for their *prioritization*. Interviews were limited to 30 minutes, in order to focus the discussion and favor gathering the top of mind requirements which would be considered the most important. In the next sections, we provide details on the interview phases.

The introduction phase begun with the author's personal introduction, for the purpose of the interviewee getting acquainted with the interviewer. This was followed by a short introduction of the research project aims and the goals of the current interview (the interviewee received the introductory text included in Appendix A). This stage ended with a question/answer period in which the interviewee could clarify any doubts. After all questions were answered and the interviewee understood the context and the purpose of the current activity, the interview progressed to the next stage. This introductory phase lasted for a maximum of 5 minutes in order to allow enough time for the subsequent phases which involve collecting research data through their responses.

The next stage had the purpose of establishing the requirements of the solution. The form included in Appendix B was used. Firstly, the interviewer stated the question from the form, and wrote down the interviewees responses in the form's table. This phase was constricted to 15 minutes, to allow ample time for the prioritization phase.

The final stage of the interview aimed at assigning priorities for the requirements, using the MoSCoW method [51]. This method was chosen because it provided a structured procedure for requirements prioritization. In addition, it is a well-known approach in the research context, so using it removed the need for explanations. For each requirement gathered in the first phase, the consultant assigned one of the following labels next to its entry in the table:

- *MUST* (M)– A requirement that was highly important for the solution to be considered a success.
- *SHOULD* (S)– An important requirement for the solution, but not vital to its success.
- *COULD* (C)– A "nice to have" requirement – not necessary but desirable.
- *WON'T* (W)– A requirement which had been agreed not to be integrated in the prototype artifact, but could be considered in the future.

This step lasted until the end of the interview, therefore it was constrained to 10 minutes. This time frame proved to be too large, as the subjects in most cases went through the requirements list and assigned the priorities almost instantly.

2.3 The requirements table

An overview of the results from this research phase is presented in Table 2.1. Each column represents the subjects' responses and each row presents the requirements gathered from the

	GC1	GC2	GC3	TC1	TC2	TC3	TC4
Up-to-date and reliable data source	M	M	M	M		M	M
Filtering		M	S	M		S	M
Multi-language support			M		S	M	M
Automation		M		S		M	
Scalability			M			M	C
Severity Level	M	M					
Ability to extend the data source				S	C		C
Upgrade Suggestion	S	C		W			
Ease of use		S			S		
Upgrade Consequences	W	C		W			
Accuracy					M		
Confidentiality			M				
Robustness							C
Proprietary libraries						C	
Extensible to other technologies			S			C	

Table 2.1: Requirements and their prioritization by consultants

interviews. For a detailed description of each requirement, we refer the reader to Appendix C. When multiple consultants expressed the same demand with different wording, we merged the description into a single representative term and searched for an established definition. For example, when the interviewees said that libraries and known vulnerabilities should "provide a usable ratio between true positives and false positives", we considered that they were referring to the concept of *accuracy*.

In each table cell, we inserted the priority rating – assigned by the consultant (represented in the current column) to the requirement (represented in the current row). To be able to rank the requirements according to their "general priority" in the table, we converted the four priority levels pertaining to the MoSCoW method, to their integral number counterparts (values ranging from 1 to 4). We then summed up all the cells in which the consultants mentioned the respective requirement and assigned it a priority. After we obtained a sum for all the requirements, we used this value to sort the requirements accordingly.

2.4 Discussion

From the consultants' responses we observed that the most important requirement was to base the solution on a high quality data source (up-to-date and reliable).

Through interviewing at SIG we also discovered a requirement that we would have not thought about without being involved in the context of use. In the consultancy practice, non-disclosure agreements are signed which disallow the consultants to offer information regarding the subject software systems to third-parties. This brought up the confidentiality requirement, where we were explicitly requested to only process information on SIG premises.

This means avoiding any external calls with information from the software systems, because that leads to unwanted information disclosure towards third parties.

A rather surprising insight from the gathered data was that only one consultant (technical) suggested accuracy to be a requirement. An important prerequisite of useful solutions are signalling vulnerabilities in libraries which indeed are part of the input software systems. An explanation for this omission might be that the consultants took the accuracy of the output for granted, so they were inclined to think about more “compelling” requirements, instead of correct matching to start with.

We also learned that filtering was another important function for the consultants. It was expressed by the majority of subjects with high priority levels. Filtering allows the users to flexibly declare which alerts are interesting and should be signaled or should be omitted. Also, a smaller number of consultants selected multi-language support to be a high priority requirement, which derived from the fact that they work with projects written in various programming languages.

Scalability was suggested by half of the consultants, but each time with high priority. It was proposed more often by technical consultants than general consultants. This is because as technical consultants, they targeted their requirements not only on the function of the tool, but also on construction attributes of it. Another example of the difference in focus was the fact that more general consultants than technical consultants proposed the feature of upgrading to a newer version of an outdated library, with a known vulnerability. They desired firstly an indication that a newer version exists, and information on the consequences of a possible upgrade.

2.5 Conclusion

In this chapter we presented the requirements for solutions for tracking known vulnerabilities in software projects in the context of external product quality monitoring. We will use it in the subsequent research steps as a frame of reference for screening relevant technologies.

Chapter 3

Vulnerability Knowledge Providers

We learned during the research step presented in the previous chapter that the most important requirement for methods to track known vulnerabilities in software projects is to use an up-to-date and reliable data source. This shows the importance of the quality of the data source in the context of external software product monitoring. Therefore, we have conducted a survey and comparison of the active publicly available vulnerability providers in order to understand the current landscape with its viable options in the light of this requirement. We also conducted an interview in order to understand what are the most important elements that these vulnerability knowledge providers offer in order to obtain another reference point useful for their critical evaluation. The findings of the interview along with the survey are used to define the vulnerability data source that we will use throughout our research project. This study was conducted also by Roschke et al. for a different purpose, attack graph construction [59].

3.1 Database Selection

We obtain the databases set through a search engine using the query *vulnerability database* (January 2014). We include every resource that we could find in which individual vulnerabilities are stored in a structured manner. The vulnerability databases that we found and included this study are:

- National Vulnerability Database (NVD) from NIST [49]
- the Open Source Vulnerability Database [52]
- VDB from DragonSoft [25]
- the advisories from Secunia [63]
- Symantecs SecurityFocus database [69]
- Securiteam database [64]
- IBM Internet Security Systems XForce database [32]

- Rapid7 vulnerability and exploit database [58]
- Carnegie Mellon‘s Software Engineering Institute Security Notes [65]
- HPI-VDB from the Hasso-Plattner-Institut [4]

3.2 Vulnerability Information Elements

The vulnerability information elements are the names of the columns provided by the vulnerability databases. We created the set of total vulnerability information elements by collecting all the unique column names of our set of vulnerability databases. Each description is taken from the vulnerability database documentation where existent, otherwise we investigated a number of entries in order to be able to assign a relevant description.

- *CVE identifier* – uniquely identifies vulnerabilities across databases, described also in Section 1.1.1 (e.g.: CVE-2013-2165[1])
- *CWE identifier* - Common Weaknesses and Exposures identifier, it is used to label vulnerability *types* [3] (e.g.: Missing Encryption of Sensitive Data[2])
- *Impact* - the type of attack that the vulnerability allows (e.g: Denial of Service)
- *CVSS* - the severity of the vulnerability [43] (e.g.: high)
- *Description* - a textual description of the vulnerability (e.g.: Cross-site scripting (XSS) vulnerability in web/servlet/tags/form/FormTag.java in Spring MVC in Spring Framework 3.0.0 before 3.2.8 and 4.0.0 before 4.0.2 allows remote attackers to inject arbitrary web script or HTML via the requested URI in a default action[3].)
- *Solution* - the solution to eliminate the vulnerability, e.g.: upgrate to the subsequent minor version
- *Attack From* - where does an exploiting attack originate from (e.g.: remote site)
- *Popularity* - the number of people that viewed the vulnerability entry in vulnerability database (for web-based interfaces)
- *Discoverer* - the entity or person that discovered this vulnerability
- *Loss Type* - the type of negative consequence that happens upon a successful exploit (e.g: Availability, Confidentiality, Integrity)
- *Similar Vulnerabilities* - Other vulnerabilities which have been identified to be similar with the current vulnerability

[1] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-2165
[2] http://cwe.mitre.org/data/definitions/311.html
[3] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-1904

- *OS* - the target operating system
- *CPE id* - the common platform enumeration identifier which is used to identify applications by specifying its vendor, product identifier and version, among others [21] (e.g.: cpe:/a:springsource:spring_framework:3.0.1)
- *Vulnerable Software* - other ways for identifying vulnerable applications, other than using CPE ids.
- *Publishing Date* - the date at which the vulnerability was introduced in the database
- *Date Public* - the date at which the vulnerability was disclosed by the vendor or other entity
- *Date Last Update* - the date at which the entry in the vulnerability database was modified
- *Discovery Date* - the date at which the vulnerability was discovered
- *References* - links to external urls, such as the vendor web page where the vulnerability was first announced (e.g.: the link to the the bug tracker entries that developers created for the vulnerability)

3.3 Comparison Matrix

Using the vulnerability information elements presented in the previous section we create a Comparison Matrix presented in Table 3.1. In addition, we include in the table the number of vulnerability elements, the number of records and whether or not the database maintainers offer an XML export or other machine-processable means of consuming the data. We include the XML export aspect aligned with the *automation* requirement presented in the previous chapter.

3.4 Discussion

As we can see from the Comparison Matrix, some vulnerability databases identify their contents using the standardized CVE identifier, while others use their own custom way. Among these, some do however reference the respective CVE entry for the vulnerability in the *References*.

Half of the vulnerability databases specify the impact of the vulnerabilities, thereby providing the means to categorize the vulnerabilities according to the type of attack they allow for (e.g: Cross-Site Scripting, Denial of Service). The majority of databases (7 out of 10) provide the severity of the vulnerability, by giving the standardized CVSS score used to rate the severity of vulnerabilities [2].

Half of the vulnerability databases contain the “Attack From” element which states the locality from which the attack allowed by this vulnerability could be expected - local or remote.

	NVD	OSVDB	Secunia	DragonSoft	SecurityFocus	SecuriTeam	X-Force	Rapid7	CERT SEI	HPI-VDB
CVE ID	x	x	x	x	x	x			x	x
CWE ID										x
Impact	x	x	x	x			x		x	
CVSS	x	x		x			x	x	x	x
Description	x	x	x			x	x	x	x	x
Solution	x	x	x	x			x	x	x	
Attack From	x	x	x	x	x					
Popularity		x	x							
Discoverer		x	x			x			x	
Loss Type	x									
Vulnerability Type	x									
Similar Vulns										x
OS				x			x			
CPE ID	x									x
Vuln. Soft.	x	x	x		x	x				
Publishing Date	x	x	x	x	x	x	x	x	x	
Date Public	x	x						x	x	
Date Last Update		x			x	x		x	x	x
References	x	x		x			x	x	x	
Nr. of vuln. el.	13	13	9	8	5	6	7	7	10	7
Nr. of records	59217	100392	48731	5457	58920	?	88937	?	?	58578
XML export	yes	no	no	no	no	no	no	no	no	no

Table 3.1: Vulnerability Databases

Out of all the databases, only two do not provide a textual description of a vulnerability. This description could be used in order to understand, among others, the context of the vulnerability, such as the preconditions or the postconditions of an attack. A description of the solutions that could be applied to remove the vulnerability is included in all but the SecurityFocus and SecuriTeam databases, this usually means a suggestion to upgrade to a specific version in which the vulnerability was removed.

Two databases, Secunia Advisories and OSVDB propose a Popularity indicator, which represents the number of times the web page for a specific vulnerability entry was displayed.

With regards to referencing the affected software by each vulnerability, most of the databases have their own way of identifying the affected software, only NVD and HPI-VDB use the standardized CPE (Common Platform and Enumeration) identifier.

Almost all the databases include the Publishing date, which means the date in which the specific vulnerability entry was introduced in the database.

A smaller number of databases includes in addition the Date Public field, which could be used to retrieve the date at which the vulnerability was firstly officially disclosed.

The Date Last Update value could be found in 6 out of the 10 considered databases, and represents the date at which the specific vulnerability entry in the database was modified.

The NVD database from NIST is the only one which provides an XML export with the database contents. The other databases provide only web browser access, therefore leveraging the information contained in them would involve an intricate process which includes HTML scraping. This brings forward the intuition that the other providers do not publish their data to be retrieved programatically, even if they make it available to the public through the browser.

Given the initial impression after the analysis, the NVD database seems a prime candidate to be used as the data source for vulnerability information that we can use for the rest of our research project. But this brings the question whether we miss out on some information by not including the information from the other databases. What exactly can we miss out? Firstly, the *descriptive power* of the vulnerabilities stored in the NVD database can be increased by matching on the CVE identifier in multiple databases and augmenting the vulnerability information elements in the NVD with the elements present in the other databases for that CVE identifier. For example, the NVD CVE vulnerabilities do not contain a Popularity attribute, but we may be able to provide it by linking with other databases by CVE id. Secondly, in the other databases, there are entries that are not labeled with a CVE, so including only the NVD database as knowledge provider could *limit the total number of vulnerabilities* that can be retrieved.

3.5 Interviewing

In order to validate our choice, an interview was set up with an SIG consultant with extensive experience in security assessments. We wanted to find out which are the most important vulnerability information elements and see how many of them are contained in NVD.

3.5.1 Design

The interview model employed is based on the MoSCoW [51] method, because it provides a structured approach to acquire data for prioritization pruposes. The elements that have to be rated by the interviewee are all the possible vulnerability information elements that are present in the range of vulnerability databases, which we are interested in understanding their priority and hence their importance.

3.5.2 Results

The result of the interview is present in this documents Appendix D. For the NVD database, the interview subject gave the responses leading to the following aggregated results:

- MUST - 2/2 of the MUST elements are present in NVD
- SHOULD - 4/7 of the SHOULD elements are present from NVD
- COULD - 7/9 of the COULD elements are present from NVD
- WONT - 0/1 of the WONT elements are present from NVD

3.6 Conclusion

The conclusion that stems from this analysis is that we can confidently use the NVD database throughout the rest of our research. The NVD contains most of the important vulnerability information elements that describe a known vulnerability, so motivation to devise a solution for unifying the vulnerability databases is supported only for the quantity of vulnerabilities consideration.

Chapter 4

The Vulnerability Alert Service

In this chapter, we present the Vulnerability Alert Service, an event-based process for tracking third-party components with known vulnerabilities in software projects throughout their development.

We start this chapter with a series of steps which form the basis for the process. Some of them are automated by a software application – a vulnerability checker, which we present after the process steps. For presenting the vulnerability checker, we firstly show the rationales involved in selecting one solution among the few alternatives identified. We then show the way we extended it in order to be applied in our research context as part of the Vulnerability Alert Service. Finally, we show the results of an assessment of the software tool with regards to its reliability for the purpose of identifying vulnerable components in software projects.

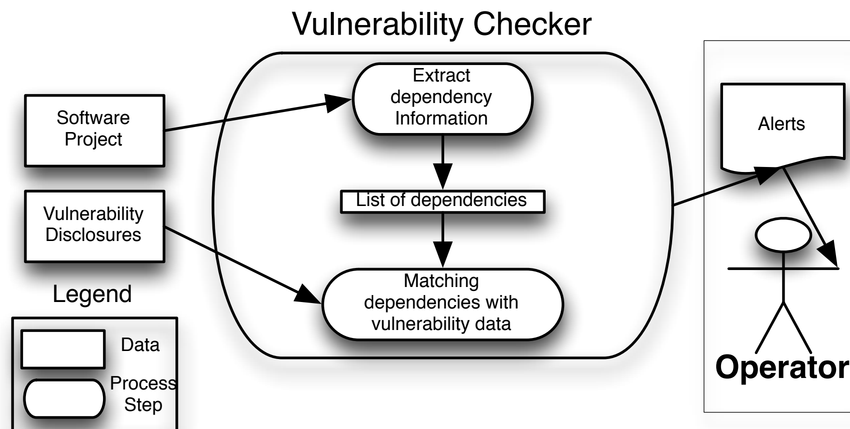

Figure 4.1: The Vulnerability Alert Service

4.1 Process description

The process is illustrated in Figure 4.1. As *process input* we have two elements: a software project and vulnerability data. Using Fawcett et al.'s *activity monitoring problems* [27] terminology, the inputs are derived from the *positive activity indicators*. For our context, these are: a project is found to include a library with known vulnerabilities, or a vulnerability was disclosed that is found to affect one of the libraries of the monitored projects.

This input is destined for the *vulnerability checker* which has two tasks: extract dependency data, recognize them and match them with known vulnerabilities. The software projects are input for the extracting task, which produces a list of recognized dependencies. This list and vulnerability disclosures are input for the matching task.

Upon a successful match, the application generates an alert, which is consumed by a human operator. After acknowledging them, the operator proceeds to filter the alerts based on usefulness, and then reports them to the interested party.

After presenting the overall process, we proceed to explain the tasks of extracting, recognizing and matching.

4.1.1 Extracting dependency data

We can extract dependency-related information from software projects from multiple sources. Consider for example Java projects. We can use any of these sources, depending on the state of the input project:

- bytecode – the set of instructions to be executed on the Java Virtual Machine resulted by translating the Java application's resources into executable format by the Java compiler. It integrates the instructions originating from the custom code, but also from the third-party code from which we want to extract dependency information.

- JAR files from the project's directory – archive files to distribute Java resources to be used in software projects. These can be used for gathering information about the presence of third-party libraries and frameworks within the software application's filesystem directory. Relevant content can be obtained from the code contents or manifest (documentation) files.

- import statements in source code – Java's mechanism to include functionality from different source files into the current source file.

- build manifest files – these files are processed by build management applications when projects are being built, and they mention third-party dependencies in specific locations.

The first approach intuitively implies that we are matching by "content", so matching on subsets of the application code. The second and the third imply that we try to match on application "name". The next step after extracting information from either of these sources is dependency recognition.

4.1.2 Recognizing dependencies

Recognizing dependencies means translating the dependency list that was received after the extraction process to a list of CPEs. The CPE identifiers are described in detail in [22]. For example, the CPE identifier for the Java library Apache Commons FileUpload is:

`cpe:/a:apache:commons_fileupload:1.0`.

In the case we extract information from imports or build manifest files, we are working at the same level of abstraction with CPEs – *application names*. If we want to use the bytecode, then we are working with one level of abstraction lower. In this case, we have to determine from which third-party component (represented by its CPE name) do specific code fragments come from. One solution may be to build binary search engines such as Rendez-Vous, by Khoo et al. [36].

Given these two possible approaches we selected *name matching* for the task of automatically identifying vulnerable libraries. We weighted a set of positive and negative arguments. We took into consideration the following positive arguments:

- the approach is more lightweight – less data to be processed and stored than when working with snippets of binary code
- it removes the need to always obtain the source code of an application for which a vulnerability is disclosed
- avoid the need to abstract from language-specific elements (e.g: compiler specific meta-data in binary code)

The trade-off that we have to make is in terms of accuracy. Using the name-based approach may result in for example, false positives as a result of naming incoherences. When there are less elements used to describe a successful match (only applications names), then it is harder to provide an accurate system than when with many elements (sets of bytecode instructions belonging to applications).

At this step, we do not make reference to the requirements established in Chapter 3. Those are requirements for the integrated solution which includes the process. Here, we address a trade-off that has to be made in the context of a specific part of the tooling that automatically matches project dependencies with their known vulnerabilities.

4.1.3 Matching dependencies with known vulnerabilities

Each known vulnerability in the NVD database XML export specifies the application in which it appears, using the CPE identifier. Therefore the next step from the list of CPEs to a list of vulnerabilities can be constructed by retrieving all vulnerabilities for a specific CPE.

This name-based approach could be used, theoretically, across language ecosystems, as names of applications represent invariants – many programming ecosystem guidelines recommend using build management software in which dependencies are declared in build manifests.

4.2 The Vulnerability Checker

Efficiency is increased through automation, and we have two options for integrating automatic vulnerability checking in our process. We can either build a software tool that satisfies the identified requirements from scratch, or we can conduct research to see what the state of the art has to offer and select one from the available options. We have chosen the latter, as the focus of the research study was to investigate the usefulness of monitoring projects for known vulnerabilities using alerts in the given research context and not on building a perfect tool. In order to make a tool selection, we have first created an inventory of existing proprietary and open source alternatives for the vulnerability checker.

4.2.1 Alternatives

To identify existing proprietary tools, we used the following search query in the Google search engine: *scanner known vulnerabilities in third party libraries*. The first 150 result pages were inspected in detail. We also explored the links present in this set of result pages.

In order to find open-source alternatives, we used Github, a widely used web-based hosting service for open-source software development. To keep the search generic, we used the term “cve” in the search interface, receiving 257 results. We have analyzed the retrieved suggestions and eliminated the ones which were not considered relevant to our use-case (the ones which do not feature repositories for projects that are related to known vulnerability scanning). We were left with 17 open-source projects that were considered relevant, and added to the candidates list.

4.2.2 Selecting one of the alternatives

We have created two tables based on the alternative solutions. In the first table, the columns represent the requirements that we previously gathered from the research context. In the second table, the columns represent indicators of open-source project activity. The activity was assessed by looking at three elements: how many users are subscribed to repository changes notifications (watch and starred counters), how many people forked the repository (the operation in which users start their own projects using the repository’s current state as starting point), the number of pull requests in the past month, the number of commits in the same time frame and the number of users which contributed throughout time to the repository.

The first decision we took was to discard the proprietary solutions. Due to licensing issues, using proprietary software would make it difficult for other researchers to reproduce experiments presented in this research document. Speaking about experiments, proprietary systems would also not allow potential needed modifications that would actually allow experiments. Furthermore, proprietary systems do not enable us to learn about the underlying method they use for implementing their features.

Using open-source software eliminates these problems. Therefore, we have decided to adopt an open-source solution for our research. The specific tool that was selected is a result of investigating the extent to which each open-source option satisfies the requirements (trace

requirements to specific constructs in the source code) gathered from the research context and how active it is. We decided that robustness should not be evaluated properly by only looking at the source code, therefore we could not include it in our analysis results. For an overview of these and the requirements-related considerations, we propose the tables on page 21.

Inspecting the tables we see the following trend: there are two categories of requirements. The categories are evidently separate. The first category encompasses requirements which most of the tools have: up-to-date data source, automation, ease of use, confidentiality. On the other hand, the second group contains requirements that almost none of the alternatives have: filtering, scalability, upgrade suggestions, upgrade consequences. These requirements could be taken into consideration by tool builders, as our research shows that there is strong interest in them, but at the moment tool builders do not consider them within the scope of the tools they are developing. Feature-wise, OWASP Dependency Check contains most of the features that we gathered from our research context.

Based on this analysis, we have selected OWASP's Dependency Check to be used as a vulnerability checker as it distinguished itself among the alternatives, as it features most of the requirements and shows signs of a relatively active development community.

Application name	Up-to-date DS.	Filtering	Multi-language	Automation	Scalability	Ease of use	Extensible to other languages
Dependency Check	✓	✓	✓	✓	–	✓	✓
Red Hat Victims	✓	–	–	✓	–	✓	–
SafeNuGet	–	–	–	✓	–	–	–
Retire JS	✓	–	–	✓	–	✓	–
RubySec	✓	–	–	✓	–	✓	–
CVE Search	✓	–	✓	✓	–	✓	✓
Fidius	✓	–	✓	✓	–	✓	✓
CVE Checker	–	–	✓	✓	–	✓	✓
ArchCVE	✓	–	–	✓	–	✓	–
Rails CVE Engine	✓	–	–	✓	–	✓	✓
CVE Search Ruby	✓	–	✓	✓	–	✓	–
CVE Easy	✓	–	–	✓	✓	–	✓
CVE Watch	–	–	✓	✓	–	✓	✓

Application name	Severity	DS. Extension	Confidentiality	Upgrade sugg.	Upgrade cons.	Accuracy	Robustness	Proprietary libs.
Dependency Check	✓	–	✓	–	–	–	?	–
Red Hat Victims	✓	✓	✓	–	–	✓	?	–
SafeNuGet	–	✓	✓	–	–	✓	?	–
Retire JS	–	–	✓	–	–	✓	?	–
RubySec	✓	✓	✓	–	–	✓	?	–
CVE Search	✓	✓	✓	–	–	–	?	–
Fidius	✓	✓	✓	–	–	–	?	–
CVE Checker	–	–	✓	–	–	✓	?	–
ArchCVE	–	–	✓	–	–	✓	?	–
Rails CVE Engine	✓	–	✓	–	–	–	?	–
CVE Search Ruby	✓	–	–	–	–	–	?	–
CVE Easy	✓	–	–	–	–	–	?	–
CVE Watch	–	–	✓	–	–	–	?	–

Application Name	Watch	Star	Fork	Merged Pull reqs/last month	Commits/last month	Comitters
Dependency Check	27	62	31	3	40	6
Red Hat Victims	9	3	9	2	5	7
SafeNuGet	10	8	3	0	0	2
Retire JS	25	363	19	0	2	9
RubySec	56	180	27	4	11	18
CVE Search	10	33	21	0	0	2
Fidius	3	15	8	0	0	3
CVE Checker	5	15	1	0	1	2
Arch CVE	2	3	1	0	0	2
Python CVE Tracker	1	2	0	0	0	1
Rails CVE Engine	1	0	0	0	0	1
CVE Easy	1	5	1	0	0	1
CVE Watch	2	3	0	0	0	1

4.2.3 OWASP Dependency Check

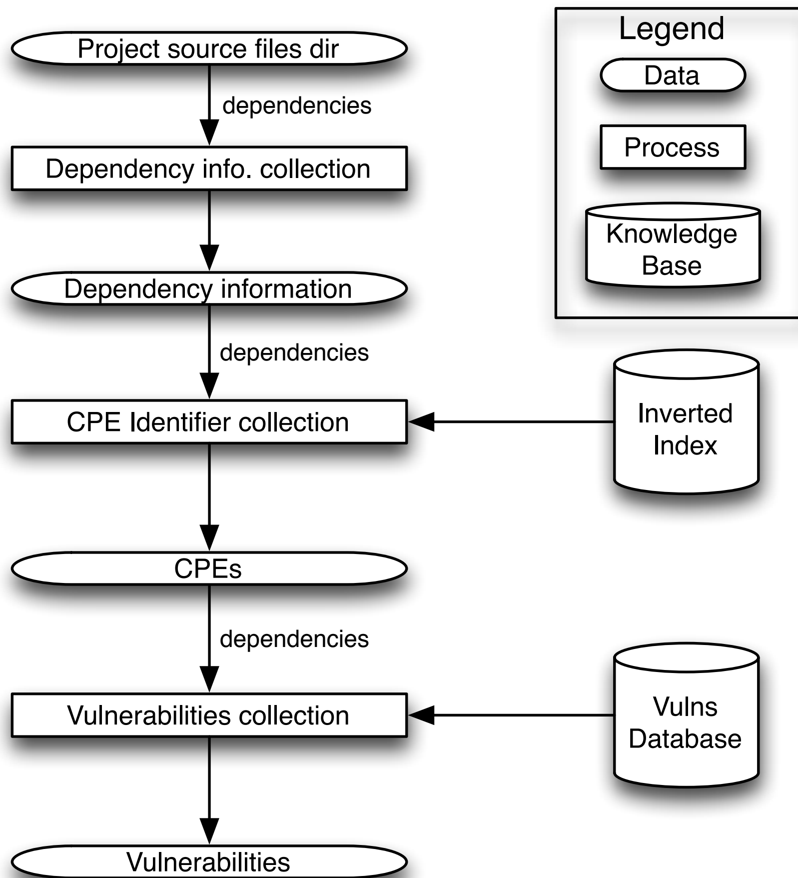

Figure 4.2: DependencyCheck scanning process

OWASP Dependency Check fits into the Vulnerability Alert Service process as the vulnerability checker application. It overlaps with the *extract dependency data*, *recognize dependencies* and *match dependency with known vulnerabilities* steps outlined in Figure 4.1. In terms of its software architecture, Dependency Check resembles the Pipes and Filters [30] pattern. The scanning process is represented graphically in Figure 4.2. The data that flows through the application is composed of *lists of dependencies*. This list of dependencies is received as parameter at each step of the scanning process, and every step is represented by an *analyzer*. Every analyzer has a well-defined task: enrich the individual dependencies in the list with analyzer-specific knowledge. Out of the box, it can extract dependency information (using sources such as JAR files for Java-based applications (JARAnalyzer), DLLs for C#-based applications, etc), recognize them using an Apache Lucene-based inverted index

(encapsulated in a CPEAnalyzer) and match recognized dependencies with their respective known CVE vulnerabilities (NVDCVEAnalyzer). We elaborate on each step below.

4.2.4 Extracting dependency information from build manifests

We can not always rely on the JARs being among the software project's contents. Current best practices suggest the usage of build management applications in which dependencies are declared in build manifest files and integrated at build time. For this reason we extended the already existing analysis functionality for extraction in the context of Java projects.

Maven is the most common build tool for Java systems, which means that we do not exclude many projects on behalf of their build management tool. Therefore we provided the ability to read Maven POM files for extracting dependency information encapsulated in the POMAnalyzer.

POM files

The Project Object Model [6] contains all the configuration information for a project, including dependency data. Using these files, developers can declare application dependencies on third party libraries using an XML format. At build time, the POM files are inspected and the declared project dependencies are downloaded from internal or external software repositories (with regards to the organization under which the project is developed). The default and principal location for external third-party components is the Maven Central repository[1].

A POM file includes project dependencies either under the `dependencyManagement` parent tag or under the `dependencies` parent tag. In the former case, we can specify dependencies which will be inherited by children POM files (this feature is provided in order to allow for POM refactoring for eliminating duplication). In the latter case, we specify dependencies which can not be inherited by children POM files and are relevant only for the current project.

An example of a POM file [50] is presented in Figure 4.4. For the POM given as example, the developers have included two third-party dependencies, namely, Acegi Security version 1.0.0 and the testing framework Junit, version 3.8.1. This information can be extracted from within the "coordinates" tags – groupId, artefactId,version.

Extension implementation details

Based on the textual contents of POM files and using the specific XML schema for them, relevant Java objects are created using JAXB[2]. Through their getter methods, these objects provide the necessary information about the input software project's dependencies.

For this extension we benefitted from the fact that Dependency Check's architecture enables developers to integrate new *analyzers* through its plugin-based extension points. We followed the template inferred from investigating the other analyzers and built the POMAnalyzer.

[1] http://search.maven.org/

[2] https://jaxb.java.net/

```
...
<dependencies>
    <dependency>
        <groupId>org.acegisecurity</groupId>
        <artifactId>acegi-security</artifactId>
        <version>1.0.0</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
    </dependency>
</dependencies>
...
```

Figure 4.3: Exaple POM file

4.2.5 Dependency Recognition

The inverted index enables efficient *simple keyword-based matching*. To show how it works, we provide the example of Jetty (version 6.1.22), a popular Java open source web server. In the Maven POM files, developers would integrate this component in the following way, under the `dependency` tags, as discussed in Section 4.2.4:

```
...
<dependencies>
<dependency>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>jetty</artifactId>
        <version>6.1.20</version>
</dependency>
</dependencies>
...
```

Figure 4.4: Jetty 6.1.20 in the POM file

The POMAnalyzer would extract “org.mortbay.jetty”, “jetty” and “6.1.20”. They would then be tokenized into the set: {“org”,“mortbay”,“jetty”} and a query will be formed with its elements for index lookup. The query would retrieve a positive result: a CPE identifier (cpe:/a:mortbay:jetty:6.1.20) as both its tokens (“mortbay” and “jetty”) are present in the query. We place one under the other for better illustration:

org.*mortbay*.**jetty jetty** 6.1.20
cpe:/a:*mortbay*:**jetty**:6.1.20

4.2.6 Matching with known vulnerabilities

The list of CPEs are used to query an embedded data base which mirrors the NVD dataset. It is used to retrieve the CVEs for each CPE in the list. After creating a list of matches it then creates vulnerability reports in various formats such as HTML, XML or plain text.

4.3 Vulnerability checker reliability

At this point we have a software application which can be used on Maven projects to obtain the known vulnerabilities associated with the set of third-party dependencies expressed in the POM file. We were interested then in understanding how much can the output of this tool can be trusted when being deployed to scan real software applications, before doing so. Therefore, we have designed a study with the purpose to obtain insight on the tool's reliability using realistic inputs. This study is motivated also because accuracy was mentioned as being a requirement.

Within the GQM [14] format, our goal for this study is:

The goal of this study is evaluating the tool's reliability in the context of Maven applications scanning from the perspective of the user.

The first thing we did when studying reliability is locate the process steps (extract, recognize and match) which can generate *wrong* results in the vulnerability reports. Wrong results may mean either a:

- *false positive* – a component which for which *other* component's vulnerabilities are assigned
- *false negative* – vulnerabilities are not successfully assigned to a project component which for which the vulnerabilities should have been assigned

  These faults can be created by the recognizing step, due to the fact that the organizations that propose the identification schemes for CPEs and third-party components are different. This means that *differences in naming the same application* may exist. The other two steps, extraction and matching are straightforward. Extracting means adjusting the tool such that it can extract dependency data from the build manifest files for different programming languages. Matching means retrieving CVE vulnerabilities for a CPE which is a facile task given that all the CVEs in NVD make referece to their respective CPE, the applications which they refer to.

4.3.1 Precision and Recall

One way to study false positive and false negative rate is using the precision and recall measures[41]. Precision is *the fraction of retrieved results which are relevant*. Recall

is *the fraction of the relevant results which are retrieved* from the data set.

4.3.2 Study design

The precision study needs a dataset of third-party component identification elements that may potentially output false positives after being used as input for scanning. From the scanner output files, we create a list of matched pairs, in which one component identification element list was matched with one or multiple CPEs. We manually label a random selection of 50 matches as being either true positive or false positive, and then aggregate these in order to get the false positive rate to be used in the formula for Precision. The labelling procedure consists of looking for left-hand side elements that would be more appropriate as a match for the right-hand side.

For studying recall, we need to contruct a dataset of components identification elements for which we *already know* that it has already one or more CVE entries assigned. This way, we can create specific inputs for which we expect the tool to flag vulnerabilities, and see how many of the expected flags actually get raised. The false negative rate will be determined by how many of these input elements do not get flagged, when in fact they should have.

4.3.3 Datasets

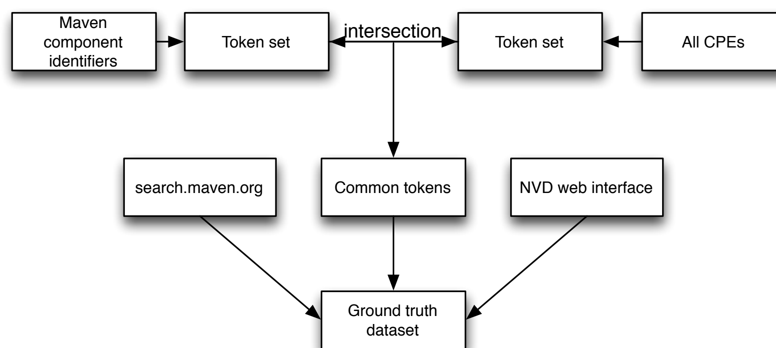

Figure 4.5: Process for obtaining the ground truth dataset for recall study

We reused a dataset of 148,253 components hosted in Maven Central from previous work [54] [56] [55] by Raemaekers et al.

The procedure to find the Maven components for which we know that vulnerabilities have been disclosed is illustrated in Figure 4.5 and can be summarized in the following steps:

- Tokenize the Maven components information elements into individual words and constructing a set of tokens with them. By tokenization, we mean splitting the

strings at the "dot" character and removing the common prefixes, such as "org", "com", which do not distinguish components. Example: groupid:`org.acegisecurity`, artefactid: `acegi-security`, version:`1.0.7` => add "acegisecurity" and "acegi-security" to the set of tokens.

- In the same way, create a set of tokens which appear in the identification elements for vulnerable applications. In this case, tokenization means splitting according the colon character. Example: `cpe:/a:acegisecurity:acegi-security:1.0.0` => add "acegisecurity" and "acegi-security" to the set
- Execute a set intersection, and obtain the list of tokens which appear in both token sets. Example: "acegi-security" and "acegisecurity".
- Manually investigate the resulting intersected set and reconstruct the set of pairs. Example:
  (`org.acegisecurity`,`acegi-security` ,
  `cpe:/a:acegisecurity:acegi-security`)

Following this procedure, we have constructed our ground truth dataset, composed of the previously mentioned pairs found.

4.3.4 Results

Precision

As we can see from Table 4.1, most of the matches are false positives. Among the total of 50 matches, 7 were matched correctly, yielding a precision value of 7/50≈0.14.

Maven Component	CPE	Type of match
mule-transport-jettymule.transports2.1.1	cpe:/a:jetty:jetty:2.1.1	FP
alarm-snmp-raow2.jasmine.monitoring1.2.1	cpe:/a:snmp:snmp	FP
jetty-ioeclipse.jetty7.2.0.RC0	cpe:/a:jetty:jetty:7.2.0.rc0	FP
maven-jboss-pluginmaven1.3	cpe:/a:jboss:jboss	FP
wicketapache.wicket1.5-M2.1	cpe:/a:apache:wicket:1.5.m2	**TP**
acegi-security-jettyacegisecurity0.8.1.1	cpe:/a:acegisecurity:acegi-security:0.8.1.1	FP
apache.felix.karaf.shell.sshapache.felix.karaf.shell1.6.0	cpe:/a:ssh:ssh:1.6.0	FP
example-jetty-embeddedeclipse.jetty7.4.2.v20110526	cpe:/a:jetty:jetty:7.4.2.v20110526	FP
struts2-coreapache.struts2.0.14	cpe:/a:apache:struts:2.0.14	**TP**
cvsjvnet.hudson.plugins1.2	cpe:/a:cvs:cvs:1.2	FP
org-netbeans-api-progressnetbeans.api	cpe:/a:progress:progress:-	FP
miniswicketstuff1.4.9	cpe:/a:minis:minis:1.4.9	FP
apache.servicemix.bundles.jetty-bundleapache.servicemix.bundles6.1.12rc1_1	cpe:/a:jetty:jetty:6.1.12.rc1	FP
acegi-security-tigeracegisecurity1.0.5	cpe:/a:acegisecurity:acegi-security:1.0.5	FP
util-fileow2.util1.0.22	cpe:/a:file:file:1.0.22	FP
xfire-aegiscodehaus.xfire1.2.6	cpe:/a:codehaus:xfire:1.2.6	FP
cargo-core-container-jettycodehaus.cargo0.7	cpe:/a:jetty:jetty:0.7	FP
cargo-core-container-jbosscodehaus.cargo1.0-beta-1	cpe:/a:jboss:jboss	FP
enunciate-xfirecodehaus.enunciate1.9-RC1	cpe:/a:codehaus:xfire:1.9.rc1	FP
mysql-connector-javamysql5.1.16	cpe:/a:mysql:mysql:5.1.16	FP
jettymortbay.jetty6.1.12rc1	cpe:/a:jetty:jetty:6.1.12.rc1	**TP**
jquerywicketstuff1.4.9.2	cpe:/a:jquery:jquery	FP
vaadinvaadin6.4.0	cpe:/a:vaadin:vaadin:6.4.0	**TP**
flex-messaging-coreadobe.flex3.2.0	cpe:/a:adobe:flex:_sdk:3.2	FP
lift-paypalliftweb1.0.2	cpe:/a:liftweb:lift:1.0.2)	FP
apache.servicemix.bundles.jetty-bundleapache.servicemix.bundles6.1.14_1	cpe:/a:jetty:jetty:6.1.14.1	FP
jettymortbay.jetty6.1.14	cpe:/a:jetty:jetty:6.1.14	**TP**
ssh-slavesjvnet.hudson.plugins0.13	cpe:/a:ssh:ssh:0.13	FP

tinymcewicketstuff1.4.9	cpe:/a:tinymce:tinymce:1.4.9	FP
apache.karaf.tooling.testingapache.karaf.tooling2.2.1	cpe:/a:apache:apache_test:2.2.1	FP
cas-server-support-openidjasig.cas3.2-RC4	cpe:/a:openid:openid	FP
animation4j-coregemserk.animation4j0.0.8	cpe:/a:jcore:jcore:0.0.8	FP
sisu-inject-beansonatype.sisu2.1.1-NEXUS-4312	cpe:/a:sonatype:nexus:2.1.1	FP
mortbay.jettyjetty5.1.5rc1	cpe:/a:jetty:jetty:5.1.5.rc1	**TP**
nexus-maven-pluginsonatype.plugins1.3.2.1	cpe:/a:sonatype:nexus:1.3.2.1	FP
openejb-cxfapache.openejb3.1.1	cpe:/a:apache:cxf:3.1.1	FP
org-apache-derbykenai.nbpwr10.4.1.3-201002241055	cpe:/a:apache:derby:10.4.1.3.201002	FP
filecedarsoft.commons.old2.0.6	cpe:/a:file:file:2.0.6	FP
tiles-coreapache.tiles2.0.3	cpe:/a:apache:tiles:2.0.3	**TP**
selenium-perl-client-driverseleniumhq.selenium.client-drivers1.0	cpe:/a:perl:perl	FP
example-jetty-embeddedeclipse.jetty7.4.0.RC0	cpe:/a:jetty:jetty:7.4.0.rc0	FP
geronimo-axisapache.geronimo.modules2.1.4	cpe:/a:apache:axis:2.1.4	FP
fabric3-jettycodehaus.fabric30.6.5	cpe:/a:jetty:jetty:0.6.5	FP
xbean-tigerapache.xbean3.1	cpe:/a:tiger:tiger:3.1	FP
apache.karaf.shell.sshapache.karaf.shell2.1.4	cpe:/a:ssh:ssh2:2.1.4	FP
jquerywicketstuff1.4.17	cpe:/a:jquery:jquery	FP
cxf-rt-transports-http-jettyapache.cxf2.4.0	cpe:/a:apache:cxf:2.4.0	**TP**
ortbay.ftpjetty5.1.1RC0	cpe:/a:ftp:ftp	FP
jquerywicketstuff1.4.12.1	cpe:/a:jquery:jquery	FP
qpid-brokerapache.qpid	cpe:/a:apache:qpid:-	FP

Table 4.1: Precision Table

Recall

As we can see from Table 4.2, the majority are correct matches. The recall value for this dataset is 47/59≈0.80.

CPE	Maven Group ID;Artefact ID;Version	Matched
caucho:resin	com.caucho;resin;3.0.9	✓
geoserver:geoserver	org.geoserver;geoserver;1.5.2	x
jasig:uportal	org.jasig.uportal;uportal;4.0.1	✓
jcraft:jzlib	com.jcraft;jzlib;0.0.6	✓
apache:shindig	org.apache.shindig;shindig;2.5.0	✓
apache:syncope	org.apache.syncope;syncope;1.0.0	✓
apache:sling	org.apache.sling;sling;2.1.0	✓
redhat:resteasy	org.jboss.resteasy;resteasy-jaxrs-all;1.0.0	x
zkoss:zk_framework	org.zkoss.zk;zk;5.0.0	x
sun:javamail	javax.mail;mail;	x
jgroups:jgroup	org.jgroups;jgroups;3.0.0	x
netty_project:netty	io.netty;netty;3.6.0	x
apache:james	org.apache.james;james-project;1.8.2	✓
neo4j:neo4j	org.neo4j;neo4j;1.9.2	✓
jsecurity:jsecurity	org.jsecurity;jsecurity;0.9.0	✓
apache:solr	org.apache.solr;solr;1.0.0	✓
apache:zookeper	org.apache.zookeeper;zookeeper	✓
springsource:spring_framework	org.springframework;spring-core;3.0.0	x
apache:coyote_http_connector	org.apache.tomcat;coyote;1.1	x
apache:cloudstack	org.apache.jclouds.api;cloudstack;2.0.1	✓
apache:myfaces	org.apache.myfaces;myfaces;2.0.0	✓
apache:commons_fileupload	commons_fileupload;commons_fileupload;1.3	x
apache:jackrabbit	org.apache.jackrabbit;jackrabbit;1.4	✓
apache:tomcat	org.apache.tomcat;tomcat;8.0.1	✓
apache:httpclient	org.apache.httpcomponents;httpclient;4.0	✓
apache:continuum	org.apache.continuum;continuum;1.1	✓
apache:hbase	org.apache.hbase;hbase;0.92.2	✓
apache:xalan-java	xalan;xalan;2.7.1	x
apache:axis	org.apache.axis;axis;1.3	✓
apache:wicket	org.apache.wicket;wicket;1.4.0	✓
xfire:xfire	xfire;xfire;1.2.4	✓
apache:cxf	org.apache.cxf;cxf;2.4.0	✓
vaadin:vaadin	com.vaadin;vaadin;4.1.1	✓
apache:hadoop	org.apache.hadoop;hadoop-main;1.0.0	✓
jfacets:jfacets	net.sourceforge.jfacets;jfacets;	✓
apache:camel	org.apache.camel;camel;1.0.0	✓

apache:struts	struts;struts;2.0.0	x
apache:tiles	org.apache.tiles;tiles;2.1.0	✓
apache:archiva	org.apache.archiva;archiva;1.2	✓
opensymphony:webwork	com.opensymphony;webwork;2.1.3	✓
liftweb:lift	net.liftweb;lift;2.1	✓
apache:derby	org.apache.derby;derby;10.5.3.0	✓
liferay:liferay_portal:::enterprise	com.liferay.portal;portal-web;6.1.2	✓
apache:activemq	org.apache.activemq;activemq-core;5.4.1	✓
codehaus:xfire	xfire;xfire-core;1.2.6	✓
apache:commons-compress	org.apache.commons;commons-compress;1.0	✓
mortbay:jetty	org.mortbay.jetty;jetty;6.1.22"	✓
apache:openjpa	org.apache.openjpa;openjpa;1.0.0	✓
acegisecurity:acegi-security	acegisecurity;acegi-security;1.0.0	✓
apache:geronimo	org.apache.geronimo;geronimo;1.0.0	✓
apache:qpid	org.apache.qpid;qpid;0.5	✓
jruby:jruby	org.jruby;jruby;0.5	✓
apache:poi	org.apache.poi;poi;2.0	✓
apache:axis2	org.apache.axis2;axis2;1.6	✓
appfuse:appfuse	org.appfuse;appfuse;2.0-rc1	✓
apache:cocoon	org.apache.cocoon;cocoon;2.1	✓
libvirt:libvirt	org.libvirt;libvrt;0.9.12	x
sonatype:nexus	org.sonatype.nexus;nexus;2.6.3	✓
jboss:seam	org.jboss.seam;seam;2.2.1	✓

Table 4.2: Recall Table

4.3.5 Discussion

The approach can also be applicable to other build tools than Maven, with the condition that they work in a similar fashion – third-party dependencies are included in build manifest files by name which are read by the build tool at build time.

From what we can see, the precision value is low for Dependency Check in the context of Maven component identifiers. This is the result of the naming overlaps that happen in the case of Maven components: `org.acegisecurity`, `acegi-security` contains the token "acegi-security" and "acegisecurity" but the same goes for `org.acegisecurity`, `acegi-security-jetty` which would yield an incorrect match.

The recall is high, with only a few failed expected matches, we can be reasonably confident that the tool would find most of the vulnerable libraries which should be flagged.

How these values translate into actual usefulness in practice is still an open question, because libraries are not of equal popularity, some appear more often in projects than others.

4.4 Conclusion

This chapter has presented the elements of the Vulnerability Alert Service, our tool-based process to track vulnerable components in the context of external software product quality monitoring. We started with introducing the process steps, then focused on the ones which are automated using a software tool. We then continued

to present the tool, describing the way we extended it in order to be applicable in our research context in order to be evaluated. We ended the chapter with presenting the results of the reliability study of the tool on a set of dependencies using the Precision and Recall measures.

Chapter 5

Known Vulnerabilities in software project dependencies

We wanted to investigate the prevalence of the problem of projects depending on vulnerable components in practice, using open-source and proprietary systems. It is important to understand whether this problem is *common* or not in practice and how often it occurs in order to motivate automated or partially automated solutions such as our Vulnerability Alert Service. If it is common, then solutions are welcomed indeed, because they would contribute to solving more efficiently a prevalent software security deficiency.

On the same note, if more libraries integrated in software projects mean more vulnerable libraries, then we have extra motivation for tooling. In that case, automatic approaches would allow for fixing more problems than manual inspection in the same unit of time, as it provides an efficient way to process modern software applications with an ever-increasing number of third-party components.

5.1 Research Questions

Based on the previously stated objectives, we formulate two research questions:

– *How prevalent is the problem of depending on third party libraries with known vulnerabilities in practice?*
– *What is the relation between the number of dependencies and the number of vulnerable dependencies in a software application?*

5.2 Software Subjects

Our set of open source subjects are the software projects hosted on Maven Central, a popular component repository for third-party open-source Java components. It is

basically the same dataset we used in Section 5.2, but now we do not consider their dependencies, but the components themselves.

As proprietary systems we use systems currently or previously monitored or analysed by SIG. We have included 75 proprietary systems in our study. They come mostly from large Dutch companies which operate in the banking, public transportation, governmental, consultancy, electronic payments and household utilities fields. Technologically, these projects are Java projects built with Maven which include their third-party dependencies in POM files. For reasons of confidentiality, the source code or POMs of our set of software subjects cannot be provided for replication. The sizes of these projects in terms of lines of code range from 1759 to 968143, and in terms of number of dependencies they range from 4 to 238.

The proprietary systems are used for answering the first question, not also the second one. The reason for this is that in the case of proprietary systems, we do not have vulnerability data about the proprietary libraries.

Another notable difference between the two datasets is the fact that our proprietary systems are actual projects which make use of libraries in order to enable a higher-level goal, while our open-source systems ones are libraries themselves. This introduces differences in their size, and naturally to the amount of libraries both types of systems use. It may also mean that they manage their dependencies differently.

5.3 Tooling

The OWASP Dependency Check tool [9] version 1.0.5 was selected to support the vulnerability scanning. At the moment of conducting this experiment, this was the current version of the application. It has been shown previously in this document that this tool is an adequate option with regards to vulnerability scanning tooling for the context of this thesis.

5.4 Maven study design

For the first question expressed in Section 5.1, we engage in a descriptive study to obtain the prevalence rate for software projects with vulnerable components among the total set of software projects. We target to obtain a simple value which shows *how many projects have at least one component with at least one known vulnerability among all the projects in our dataset*.

On the other hand, for the second question, the decision was to engage in conducting a *correlation study* to find the answer, as we are interested in characterizing a relationship.

We provide a Github repository with the artefacts needed to replicate this study[1].

[1] https://github.com/mcadariu/vuln

5.4.1 Hypothesis

Our *null hypothesis* for the second question is:

H0: *The number of vulnerable libraries is not correlated with the number of libraries in software projects.*

Consequently, we define the *alternative hypothesis*:

H1: *The number of vulnerable libraries is correlated with the number of libraries in software projects.*

5.4.2 Independent and Dependent Variables

The variables for our experiment refer to the set of third-party dependencies found in the build manifest files of the subject software projects. For this study, the set of dependencies come from their POM files.

Our **independent variable** is the *cardinality of the set of dependencies*. The **dependent variable** is the cardinality of the set of *vulnerable* dependencies of projects, as given by our vulnerability scanner.

5.4.3 Process

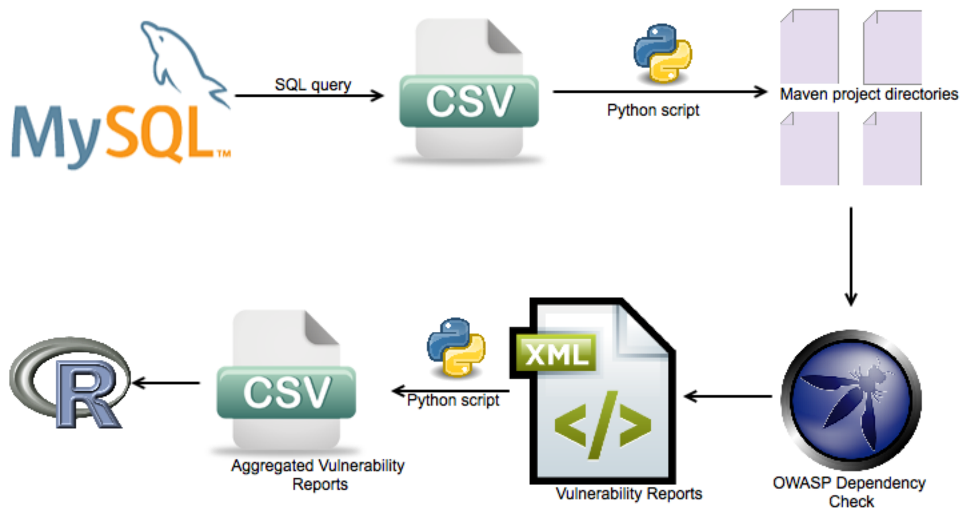

Figure 5.1: The study execution procedure

In order to research answers for the stated research questions, we need to go through a series of steps, depicted in Figure 5.1:

- Create input data – acquire the software subjects and make inventory of dependencies
- Scan the projects with the vulnerability scanner
- Prepare data for analysis – aggregate for each project the number of components and the number of vulnerable components in separate columns and prepare histograms with frequency distributions

These steps provide the input for the analysis procedure.

5.4.4 Analysis procedure

For the first question, we investigate the frequency distribution of the number of vulnerable components per software project using a histogram.

For the second question, we need to conduct a correlation analysis. In order to select the right statistical analysis, we need to understand first the distribution from which our sample data comes from. After having insight on the distribution of our data we can select the right statistical test.

5.5 Maven study results

Almost one out of five open-source software project includes one or more known vulnerabilities in their dependencies.

This finding can be derived from the descriptive statistics tables that we show in Table 5.1 and 5.2.

Nr. of projects	12100
Min. nr. deps.	1
1st. Quartile	2
Median	4
Mean	5.44
3rd Quartile	7
Max nr. deps.	171

Table 5.1: Descriptive Statistics – Project Components

As we can see from Table 5.1, the projects feature in general a small number of dependencies. As well, we can see that the total number of projects is high. This will contribute to the significance of the conclusions drawn.

From Table 5.2, regarding descriptive statistics if we consider the vulnerable components of projects, usually they do not contain any vulnerability. When projects do have vulnerable components, their number can amount as high as 22, the maximum value.

Nr. of projects with known vulns.	2456
Min. nr. deps with vulns.	0
1st. Quartile	0
Median	0
Mean	0.46
3rd Quartile	1
Max nr. deps with vulns.	22

Table 5.2: Descriptive statistics – Vulnerable Project Components

The frequency distribution of the number of vulnerable components found in projects can be observed in Figure 5.2. The findings resemble the general observation that dependencies follow a power law distribution. Almost all projects have fewer than 50 dependencies.

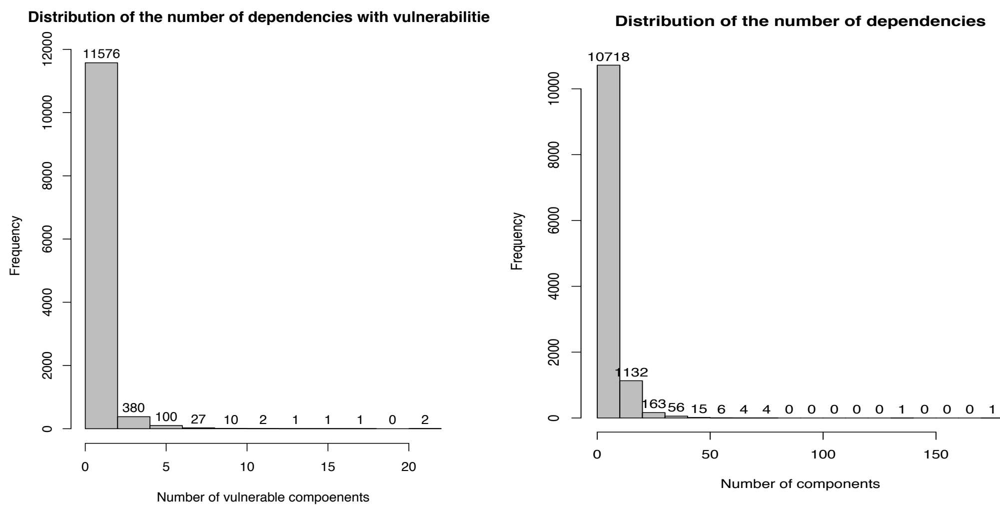

Figure 5.2: Frequency Distributions for number of components and number of vulnerable components

5.5.1 Hypothesis Testing

As we see in the histograms, our input data for the correlational study is not normally distributed. Therefore, we can use the Spearman rank correlation test, because it does not make any assumption with regards to the distribution of the data.

We use the tresholds proposed by Hopkins [31], therefore a significant correlation value higher than 0.3 indicates a moderate correlation. A correlation higher than 0.5 indicates a

strong correlation. For the correlation to be significant, the p-value has to be lower than 0.01, therefore there is less than 1% chance that the correlation value is due to chance.

By conducting this test, we observe that we have a moderate correlation of the number of components and the number of vulnerable components (Spearman's rho$\approx$0.32). This result is statistically significant, as the p-value is less than 0.01, thus we can confidently reject the null hypothesis.

5.5.2 Discussion

A confounding factor that may influence the validity of the conclusions drawn is the accuracy of the tooling used to scan the projects' dependencies. If it produces many false positives, then the number of vulnerable components that are flagged may be larger than what it should be. False negatives would lead to an amount of reported vulnerable components that is less than what it should be.

The alternative would be manual analysis, which would be more accurate, but it is infeasible to do considering the high amount of projects some of which include dozens of libraries.

5.6 Proprietary projects study design

In order to analyze the proprietary projects we applied a procedure which includes automatic and manual steps. In contrast with the procedure described in Section 5.4.3, this one includes manual steps that have the purpose to curate the scan results to provide for more accurate conclusions. In this study we look only at prevalence, as we do not have access to all project libraries, for example the proprietary ones. The procedure consists of the following steps:

- Automatically scan projects – We used our vulnerability scanner to derive vulnerability reports for each project showing the dependencies which have CVE vulnerabilities.
- Manually curate the output files – We have learned in Section 4.3 when the tool goes wrong and produces a false positive and when it does not recognize a library with known vulnerabilities. This knowledge was integrated in a manual process to curate the project reports of false positives and false negatives. The reports were stored in a simple text file. The smaller number of projects in our dataset compared to our study presented in Section 5.4.3 allowed this manual process to take place.
- Automatically aggregate the results – We have programmatically aggregated the reports in order to present the results and allow for futher analysis in line with the goals of this chapter.

5.7 Proprietary projects study results

We found vulnerable libraries in 54 projects out of the 75 that we analyzed.

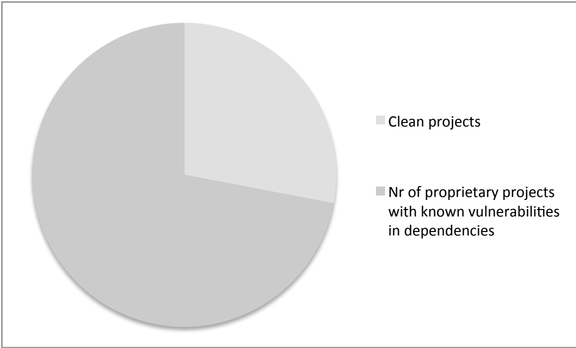

Figure 5.3: Vulnerable vs Clean projects with regards to known vulnerabilities in their dependencies

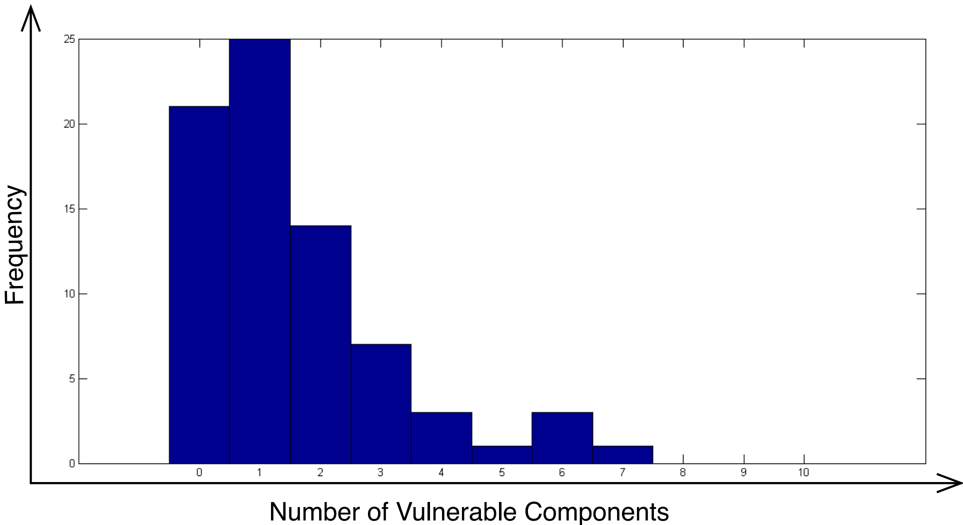

Figure 5.4: Distribution of the number of vulnerable libraries across proprietary software projects

Figure 5.3 provides a graphical illustration of the prevalence of the problem among our data set.

With the histogram presented in Figure 5.4, we seek to understand quantitatively the frequent state of proprietary systems in terms of their vulnerable components. We see that the most frequent case which happens in practice is that there is one vulnerable component in a proprietary system. We can also observe that some proprietary systems have as many as 7 components with known vulnerabilities.

	Vulnerable component	Nr. of occurences
1	Spring Framework 2.5.6	12
2	Apache POI 3.8	5
3	Spring Framework 3.0.6	4
4	Apache CXF 2.6.0	4
5	Apache Axis 1.4	4
6	Acegi-Security 1.0.3	4
7	Spring Framework 3.1.1	3
8	Spring Framework 3.0.5	3
9	Apache POI 3.6	3
10	Apache CXF 2.5.1	3

Table 5.3: Top 10 vulnerable components found in proprietary projects

Top 10 vulnerable components in proprietary systems

In Table 5.3, we can see the ranking with regards to the number of occurences of specific libraries across our dataset of software projects. By a good distance, the most common library that we found in our projects is the Spring Framework version 2.5.6, which in the Top 10 includes other versions being a very popular application framework in the software development industry.

Spring Framework version 2.5.6 was found to be the most prevalent library with known vulnerabilities accross the proprietary projects we have analyzed. Therefore we provide a closer look into its vulnerabilities in the next section, also in relation with the vulnerability date of discovery and fix date.

Spring Framework 2.5.6

This library has three known vulnerabilities:

- *CVE-2013-6429* (severity:medium) [1] – This vulnerability allows attackers to retrieve the contents of arbitrary files using crafted XML input files on the application server through the external entity resolution capability of Spring's XML processor. Through browsing its online description we find out that this vulnerability was found in 2010 and fixed in a release in the same year.
- *CVE-2011-2730* (severity:high) [2] – This vulnerability is documented by the discoverers in a white paper [3]. By exploiting this vulnerability attackers can execute methods on the application's objects to get access to sensitive information. It was found in 2012 and fixed in a release in the same year.

[1] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-6429

[2] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-2730

[3] https://docs.google.com/document/d/1dc1xxO8UMFaGLOwgkykYdghGWm$_2 Gn0iCrxFsympqcE/editheading = h.eimq2kjcg8r2$

- *CVE-2010-1622* (severity:medium) [1] – this vulnerability allows for execution of arbitrary code, but investigating the published exploit [2] shows that this vulnerability is intricate and not straightforward to take advantage of. It was found in 2013 and fixed in a release in the same year.

In all cases, the fixes were released before the data at which we know that the projects were still in development (as they were being currently monitored and snapshot dates indicated the year 2014 and fixes were in 2010,2012,2013). Therefore, fixes were available at the time of our analysis.

All the findings that were generated through our research such as the above one have been gathered and are pending disclosure to the project owners after a thorough investigation. At the time of writing it is unclear what the suitable disclosure procedure is, as some project owners may be surprised if they are notified of aspects for which they did not request explicitly.

5.8 Conclusion

The numbers that arose from our analysis are alarming. Projects which feature known vulnerabilities as a result of their selection of third-party libraries cover the majority of our total project set for proprietary systems. Even more, the majority of them depend on a component which have high severity vulnerabilities that can lead to attackers having access to sensitive information. This would have highly negative consequences if exploited. Also, the fact that one out of six open-source projects have at least one known vulnerability in their dependencies represents represents makes the alarming state more evident. Investment in tooling that aids in detecting this problem in practice is worthwhile.

[1] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-1622

[2] http://www.exploit-db.com/exploits/13918/

Chapter 6

Evaluating the Vulnerability Alert Service

We have so far identified and diagnosed a practical problem in our research context and proceeded to design and introduce a solution [73]. An alert-based methodology which makes use of software tooling to track vulnerable components was integrated in the daily operations of our host company, and the first alerts have been produced. The alert's contents are the software project for which they are relevant, the CVE identifier and a link to their online description from the NVD database.

We look to assess the alerts' *usefulness* as perceived by prospective users of our methodology in our research context – the external software quality assessors that overlook the software project in terms of its quality attributes throughout its evolution.

We use Gopal's operationalization of the concept of usefulness [29]. In his view, the subjects of the study are useful if they are actively used in a decision making process. Where applicable, we investigated whether alerts actually contribute to decisions specific for our research context. This view was already successfully used in the same research context for a different goal – metric usefulness evaluation [19].

The above considerations enable us to formulate our goal for the current research step, following the GQM template [14]:

The objective of this study is to understand the usefulness of the vulnerability alerts, from the point of view of external quality assessors, in the context of external quality-oriented software monitoring of software projects.

6.1 Study design

For evaluation, we constructed a three-part evaluation study, as shown in Figure 6.1. The first step is embedding the monitoring process in the daily operations of our host company and alerts are raised. The second part covers the task of collecting relevant information from the research context with regards to the alert's usefulness. This data is then used in the subsequent step, data analysis.

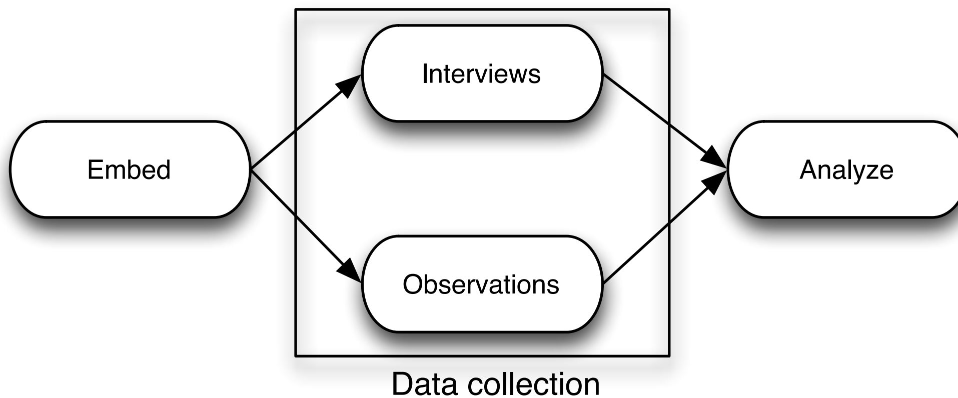

Figure 6.1: Usefulness Evaluation Study Design

6.1.1 Embedding

Our embedding scenario mimicks the one used by our host company for monitoring the evolution of quality attributes of software systems throughout time, described in Section 1.2. For a period of time, every project which was uploaded on the premises of the host company was given as input for the Vulnerability Alert Service process. The generated alerts are redirected to the author instead of the Monitor Control Center.

6.1.2 Data collection

The data collection phase is split in two parts, distinguished by the aspect we want to focus on in our evaluation. For the first part, we have taken the role of the person receiving the vulnerability alerts (the operator, in the process described in Chapter 4) and removed the false positives. Some of these true positives were brought in attention of the technical consultants responsible for monitoring these specific projects for which the alerts are relevant. In these disclosure meetings, *semi-structured interviews* combining open and closed questions were asked in order to offer preliminary insights on the added value of alerts in their context.

The second part of the data collection step is geared in the direction of an *observational-exploratory* study, in the sense that we did not take the role of the operator to curate the alerts in any way. For a period of time, the technical consultant responsible with observing the Monitor Control Center for noteworthy events in the monitored projects was informed of the alerts that were issued the day before, and he/she was asked to point out the useful ones which were written down on a note. This way, we can get insight on the usefulness of the alert methodology as a whole in their real context of use.

6.1.3 Analysis

The results of the first data collection step was subjected to a qualitative analysis. This means we looked for patterns in responses and pointed out when there is agreement between responses and when there is strong or weak disagreement. The design of the second data collection step promotes quantitative analysis, due to the bounded range and specific response options that the interviewees were offered to select. Here, we are interested in the proportion of useful alerts among the total.

6.1.4 Timing

In contrast with the procedure for our requirements-gathering interviews, we do not limit the time of the interview. The requirements interviews were bounded in order to capture the top of mind in terms of requirements to identify the most important ones. With this study we focus on obtaining rich evaluation related information from the interviewees.

6.1.5 Descriptive statistics

For an overview of the number of subjects (alerts and technical consultants) that we used as input for the data collection, Table 6.1 is shown. For the second data collection step, the one in which observations are recorded, we do not provide ratios for the alerts/week and average alerts per project as in the first one. The focus for that step is not *on the process* like in the second, but on a bounded list of specific alerts.

Data collection step	Nr. of alerts	Nr. of TCs [1] participating	Nr. of projects
Interview-based	4	4	4
Observation-based	449	4	34

Table 6.1: Descriptive statistics for the data collection steps

6.1.6 Questions

For the first data collection step, the following set of interview questions were created.

- *Would you have investigated the presence of known vulnerabilities in third-party libraries in the monitored projects without our method?*
- *Do you find this specific alert useful?*
- *Do you find the time taken to process alerts in this form to be adequate?*
- *What would be your action after acknowledging this alert?*
- *How soon after acknoledging the alert will you take action?*
- *What do you think will be the outcome of this action?*
- *What are other aspects that you considered besides the alert when analyzing what would the resulting action be?*
- *Do you think this alert signals the presence of a class of problems with this project?*

6.2 Interview Findings

We have grouped the technical consultants' answers according to each question and aggregated as described in Section 6.1.3.

Tracking dependencies with known vulnerabilities in the absence of our method. The technical consultants responded that they would not do it. This can be explained by practical reasons since doing it without a tool such as the VAS is very time consuming. One pointed out though, that if he would consider security an important concern given the purpose of the application, he would indeed conduct a systematic review of the project's dependencies to find weaknesses.

Usefulness of the alerts. All the interviewees found the proposed alerts useful. The majority also specified that the alert itself is useful, but every alert should be followed by a more in-depth analysis, to understand whether or not the vulnerability may indeed cause compromise the application. It was not surprising that all the technical consultants found the set of alerts useful, as they indicate potential exploits of the relevant software project.

One of the interviewees included mentioned the domain of the application as a result of being notified of the vulnerability. The fact that the application deals with sensitive data makes it an interesting application security-wise. This means that it features more interest for security-related alerts of all kind. This enforces the idea extracted from the previous answer, in which the purpose of the application also contributes to alerts being useful, as they are more valuable when security is an important concern for the software application.

Another interviewee specified that the actual vulnerable functionality described in the alert made it useful, as it relates to functionality that can be communicated to the development team in a straightforward manner. He continued his remark saying that from his experience, sometimes known vulnerabilities are more intricate and more difficult to turn into concrete improvement advice.

Alert processing time. All technical consultants responded that the time it took them for processing an alert was very short.

Action after acknowledging the alert. The technical consultants answered that they would notify the clients of these findings. One interviewee added that he would send an e-mail explaining that with a security scan a risk has been found, the library which can be exploited, an example of a use-case in which it could occur and alternatives for minimizing this risk. The other interviewees did not mention these aspects besides the notification.

Time between alert and follow-up action. None of the alerts presented were considered critical, so times expressed by the technical consultants ranged from one week to two-three weeks, depending on the agreed schedule for monitoring-related reports.

Expected outcome from the action. With regards to the outcome of the recommendation, the subjects could not say for sure what the contact person will decide after being notified of this security vulnerability, but they do expect a response from them. One interviewee pointed out that they may investigate the possibility of upgrading to a newer version in which the vulnerability is fixed. Another one added to this, that another solution would be removing functionality.

Other aspects considered when deciding for a course of action upon being alerted. As expected the subjects mentioned that they would look at whether or not the vulnerability

description matches some functionality that the client actually uses. Other aspects that go into their thought process, is as one interviewee mentioned, whether or not they can think of a use-case in which they can put into context a potential security breach as a result of leaving vulnerable libraries in the project scope. Another subject also indicated that he thinks about how important security is as a quality attribute of the project under observation.

Class of problems. The majority of the subjects said that known vulnerability alerts specific to a system shows that long-term maintenance is not a priority. One of them justified this response using a project in which maintenance effort is not scheduled, as it will soon be retired and a new system that is currently developed will take its place.

6.3 Observations Findings

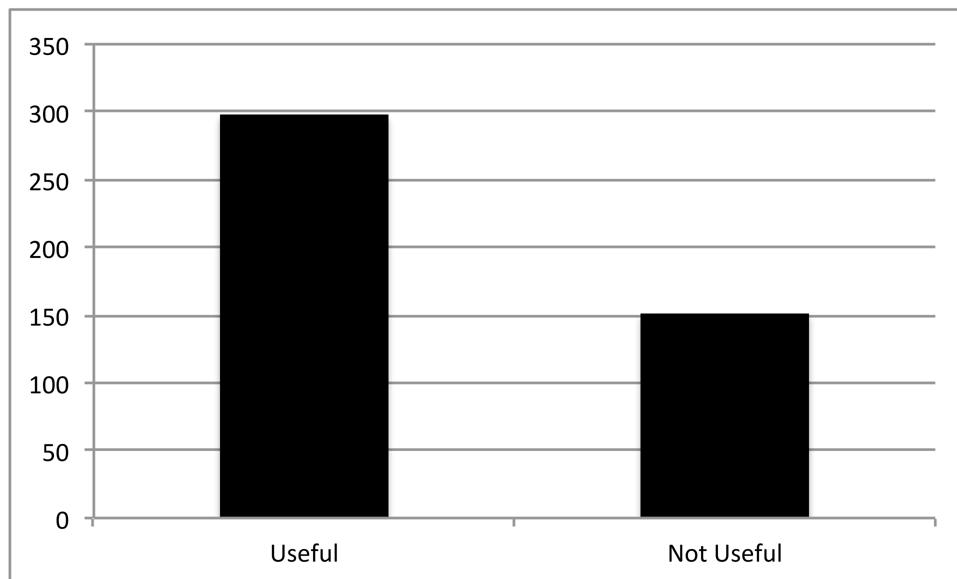

Figure 6.2: Useful vs. Not Useful Alerts

In Figure 6.2, we show the number of useful alerts. As we can see, one third of the total number of collected alerts are not useful, whereas two thirds were considered useful by the technical consultants.

A distinguished contribution was communicating one of the findings[1] produced in our research to the security officer at a large dutch banking organization. He was pleased with this finding and insisted on the application of corrective measures to remove the security vulnerability from their product. In addition, he expressed his interest in this type of findings and encouraged the responsibles on behalf of our host company to continue contributing with such security-related findings.

[1]http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-3700

6.4 Discussion

During the interview sessions, we noticed how the technical consultants approach each alert. In order to decide whether they are useful or not, they firstly confirmed the use of the libraries and how the signalled library is used in the project. Whenever they actually found a trace of the library in the project, even though the library was not referenced in the source code or used only for testing purposes, they considered the alert to be useful. The reason is that even though these findings do not make the application exploitable right now, it would be desirable to remove this possible threat to prevent any future usages that may lead to vulnerabilities.

The quantitative study shows that there are more useful alerts. During our analysis, we have found that a large part of the not useful alerts were triggered by a single false positive – the MySQL connector Java library was mistakenly matched with vulnerabilities of an old verision of the MySQL database. This amounted to over 100 false positive alerts that were naturally discarded as being not useful. If we remove this element, the useful elements vs. unuseful would be around 6 to 1.

This may appear in contrast with the earlier finding in which the tool provided many false positives which would make for a low number of useful results. A possible explanation may be that the false positives were generated because the input data contained libraries which would not be often encountered in practice.

6.5 Threats to validity

In analyzing the threats to validity we consider the guidelines proposed by Wohlin et al. [76].

6.5.1 Conclusion validity

Investigating conclusion validity means checking whether the results provided by the study is convincing enough to support the conclusions drawn. The first evaluation step, the one based on interviews, could have used more subjects – more participants, more projects, more vulnerabilities. We used the triangulation procedure to increase the study's validity by combining the results of two studies, one based on interviews and one based on usage observations.

6.5.2 Construct Validity

Questionnaires may reflect the researcher's expectation for a specific answer [37]. It is inevitable, as researchers, to have intuitions with regards to the expected answers when preparing the question list, especially after preparing the interview materials. Nonetheless, useful studies are the ones in which bias from the intuitions are avoided. If they are not, it would negatively influence the validity of the answers and thus, the quality of the research conclusions. We have attempted to overcome this type of bias by specifying clear and unambiguous questions that do not suggest answers. A possible further improvement would be to alternate positive with negative questions to break the answer momentum.

There is a risk of interviewer bias as a result of the fact that the author of the thesis is asking the questions. This risk is considered minor, due to the fact that interviewees may potentially use the proposed solution, therefore they will be inclined to give honest responses because their future way of working may be influenced by the outcome of the current study.

Another situation in which researcher bias has to be avoided, is during the data collection phase, specifically in our first data collection step, the one based on interviews. The author conducts the interviews and then conducts an analysis based on them. This may influence the validity of the conclusions in the sense that the meaning of the communicated information by the interviewees may be distorted. Ideally, this type of interview would be conducted by two or multiple researchers, and conclusions cross-validated. Furthermore, validation with the interviewee could have been conducted at the end of the interview.

6.5.3 External Validity

With external validity we are interested in describing the potential to generalize the research results outside the research context. The use of technical consultants coming from a single company limits the generalizability. In order to generalize the results beyond the research context, we have to replicate the experiment using subjects from multiple organizations that resemble a more heterogenuous group. This was not possible for our research, as the proprietary systems we used are confidential.

Another threat is the fact that we have used only Maven-based Java systems. Our findings can be generalized to comparable technologies (Ruby when using Gems). These technologies also require dependencies declared in specific files before build time, and the way in which dependencies are declared is also similar – we input application names, which can be extracted and matched with CPEs.

6.6 Conclusion

Our evaluation shows that our method produces useful security-related alerts consistently originating from the presence of known vulnerabilities in third party libraries of software projects. We arrive at this conclusion after an evaluation study which includes quantitative and qualitative data analysis on the basis of industry systems and alerts triggered on their behalf. We also show that despite the fact that the precision study conducted in Section 4.3.1 did not come out very positive, it was because of the very general way of assessing this aspect. For that study, we used any component that can be downloaded from the Maven Central. In practice though, we see only a subset of components actually being used consistently in software projects. The fact that we see more useful alerts than not useful shows that the solution does not show high precision if we allow anything as input, but it performs well in the context of common practice.

Chapter 7

Related Work

Our research primarily builds on prior work on the theme software product quality monitoring [15]. We have also surveyed related literature on the major topis of this thesis. In this chapter, we present previous reserach conducted on usages of known vulnerabilities, empirical studies that consider projects which feature known vulnerabilities in their dependencies, and how Maven compoenents were studied from the software security perspective.

7.1 Known Vulnerabilities

We have used known vulnerabilities for project-centric empirical studies and for introducing a monitoring practice for them in the context of external software product quality monitoring. In addition, we investigated the usefulness of the method in our research context. In the literature, we can find other usages of known vulnerability datasets. In this section we present them. We grouped the related literature according to the type of knowledge the papers propose related to known vulnerabilities. Therefore, we could identify four groups: *characterization*, *tooling*, *structuring vulnerability knowledge* and *modelling*.

The papers on characterization have the goal to create descriptive knowledge on CVE vulnerabilities. They approach the known vulnerability datasets from different angles for the purpose of describing their characteristics under different views. Massacci et al., for example, analyzed whether specific databases are adequate for being used in responding the various research questions that researchers have engaged with [42]. Neuhaus et al., used the topic modelling technique from natural language processing in order to provide an overview of trends with regards to vulnerability types [48]. The vulnerability lifecycle, was studied by Shahzad et al. in a large-scale exploratory analysis [66]. Scholte et al. [61] [62] how known vulnerabilities regarding web applications have evolved through time in the last decade. Regarding the patching behavior in response to vulnerabilities, Temizkan identifies the elements which influence the release of a patch as a result of a disclosed vulnerability [71]. Allodi et al. found that the vulnerability data in NVD is not a good predictor of actual attacks in the wild [13]. In our work, we also look at the characteristics of the datasources, but for a different purpose. We researched their fitness with regards to our stated goal in our specific research context.

A series of papers have the goal to create a structured representation of vulnerability data [47] [35]. This would make them easier to process by a computer. The authors use techniques borrowed from the Semantic Web research field. By using data from the NVD, our vulnerability dataset is structured, but if we would like to extend the method to be able to track vulnerabilities in sources other than NVD, then these techniques will be worth considering.

A set of papers use the vulnerability data for training various kinds of models which are used for vulnerability-related predictions. Bozorgi et al.'s model rates vulnerabilities based on chances to be exploited in the future [20]. Alhazmi et al.'s model predicts the quantity of undiscovered vulnerabilities in software applications [12]. Rahimi et al. propose a stochastic model based on source code properties which is able to predict the rate of vulnerability discovery [57]. Finally, Zhang et al. investigated the predictive power of the entries of the NVD database [79]. This paper can also be classified as a *characterization* paper, but we mention it in the *modelling* category because it serves this purpose.

7.2 Empirical Studies on Known Vulnerabilities

Xia et al. [78] investigated the usage of outdated, third-party code with known vulnerabilities in open source applications. Using a set of open-source components, they investigated their usage among other projects. They used repository mining and code cloning techniques to achieve this. Their study shows that many projects rely on outdated third-party dependencies and that and that developers generally do not take corrective measures due to reasons such as "we update when any problem happens to avoid introducing new bugs", or "cannot update because of multiple dependnencies and compatibility issues". The empirical study conducted differs from ours in the approach. Our approach is project-centric, in the sense that we scan multiple projects to insight on the known vulnerabilities within them, while Xia et al. adopt the component-centric approach, in which they trace usage of a set of components throughout open-source projects. Nonetheless, both studies provide the same insight – this is not an isolated problem, it appears often in software projects.

On the same note, in a recent publication from Aspect Security it is presented that 1 out of 4 downloads from the Maven repository is a component with at least one known vulnerability [75]. With our empirical study, we confirm the assumption that this high number of downloads is reflected in many projects in practice include components with known vulnerabilities.

7.3 The Security of Maven Components

Mitropoulos et al. [46] and Saini et al. [60] use the Maven components in security-related research. Using the FindBugs static analysis tool, they investigate the security vulnerabilities of the components to create bug catalogs. Due to the fact that the Maven Central repository contains different versions of the components, it allows studies that focus on the evolution of security bugs over time as well [44] [45].

The primary difference with our work is that we focus on *known vulnerabilities*, while the tool used in the previously mentioned papers finds security vulnerabilities by inspecting the abstract program representation of the Maven componenets.

Another difference is in how we use the components for our work. We firstly used them to study the reliability of the open-source vulnerability scanner that we leveraged in our method to track known vulnerabilities in software projects in the context of external software product quality monitoring. In addition, we use the Maven components as *projects with dependencies*, in order to empirically study the prevalence of the problem of depending on vulnerable libraries in practice.

Chapter 8

Conclusions and Future Work

This makes the following contributions:

- We define the problem of tracking vulnerable components in software projects in the context of external software quality monitoring
- We present the prevalence of the problem in practice through an empirical study using open-source and proprietary software systems
- We survey the current vulnerability knowledge providers, assess their fitness for our purpose and motivate our decision to leverage one of them for our research
- We derive a set of requirements for methods to track vulnerable components
- We propose a method for tracking vulnerable components that partially integrates the requirements
- We embed the method in the daily operations of our host company and evaluate the usefulness of its findings with the potential future users of the method

Evidence for the relevance of our approach comes from the positive change we made within the dutch banking organization's application, described in Section 6.4. It shows that there is real interest in the industry in findings produced by the Vulnerability Alert Service, which advocates its usefulness.

8.1 Answering the research questions

In this section, we revisit the research questions introduced at the beginning of this thesis and present the answers.

In order to be able to respond to the principal question, we devised a series of sub-questions that we answer first.

> *What are important requirements for methods to track vulnerable libraries in the context of external software quality evaluations?*

In order to answer this question, we have organized interviews with subjects coming from the research context – technical and general consultants working for our host company.

The result of the interviewing sessions is a list of requirements along with their prioritization. The most important requirement was considered having access to a quality datasource in terms of its update frequency and coverage of vulnerabilities contained within. Part of the requirements found their way into the software application that underpins the final solution.

Where can we find vulnerability data?

In order to answer this question, we have engaged in a survey to find the current vulnerability knowledge providers. We created a set of vulnerability providers that we compared using their contents. After understanding how do they compare with eachother, we have devised an interview. This was done to help us understand the point of view of the subjects selected from our research context on what is important to use from the total range of information elements of the vulnerability providers. The decision that emerged from this investigation is that the NVD database is adequate for our research purposes and will be used throughout the rest of our project.

How can we produce alerts on the presence of known vulnerabilities in software projects?

In order to select a software tool that produces these alerts automatically, we firstly surveyed the state of the art in order to derive a list of candidate solutions. Then, the requirements gathered from answering the previous research question were used to study the fitness for purpose of each. In the end one open-source tool showed a good coverage of the requirements and was selected to be used in our research project. Before integrating it in the daily operations of our host company, we studied its reliability in terms of producing useful alerts, using established measures from the Information Retrieval field – precision and recall. As subject data for this study we leveraged the Maven dataset of open-source components. We then proceeded to introduce the monitoring operation at our host company. We introduced known vulnerability scanning as an extra automated regularly scheduled investigation of a software system and alerts started to arrive, signaling the presence of known vulnerabilities in client software systems.

How prevalent is the problem in practice?

The prevalence study had the goal to understand how common is the problem of depending on vulnerable libraries. For this, we considered two datasets containing open-source and proprietary software projects. In order to quantify prevalence, we executed our vulnerability scanner on the project set and aggregate the results.

Are the alerts useful?

We gathered a collection of alerts over a period of time and studied their usefulness with the cooperation of the technical consultants, the prospective users of this method. In order to study usefulness, we organized focused interviews on the basis of individual alerts but also a long-raning observation phase in which we observe the technical consultants how they process alerts.

Answering to all the sub-questions allowed us to answer the principal question:

How can we automatically produce useful alerts in the context of signalling known vulnerabilities in third party libraries in software projects?

The way in which we can produce useful alerts related to the known vulnerabilities present in software applications is by integrating our process to track known vulnerabilities in software projects. Its byproducts, security alerts, have been empirically proven to be useful in our research context on a consistent basis.

8.2 Future Work

In our view, future work revolves mainly around replicating our study in the same setting for confirming our findings, but also in a different setting, in order to understand the impact of the context on the conclusions drawn. Another element of future work relevant for our study are extensions to the contributions. In this chapter, we present the areas of study. For these elements we consider that other researchers may find opportunities to extend the work presented in this thesis.

8.2.1 Requirements Analysis

The Requirements Analysis research phase was based on the input coming from a population drawn from a single company which offers consulting services on matters of software quality. Further research can explore the changes brought in the gathered requirements by changing the research context to, for example, software development companies which would like to introduce tracking known vulnerabilities introduced by third party dependnencies in their software development process.

8.2.2 Vulnerability Knowledge Provider

As our study has shown, currently, the NVD database has proven to be the most useful vulnerability database for our research. This is due to its fitness for our research goal and facile programmatic access. This database contains known vulnerabilities which have been assigned a standardized CVE identifier. But for a vulnerability to be *known*, it does not necesarily need to go through the process that leads to a CVE assignment. Some security vulnerabilities are known before receiving a CVE, such as when users of open-source projects signal security vulnearbilities. Ideally, tracking known vulnerabilities would mean indexing every source of information that contains pointers to security threats.

8.2.3 Vulnerability Scanner

An open-source implementation of a known vulnerability scanner, Dependency Check, was used in our research. Future work may use different vulnerability scanners in the same context or a different scanner in the same context and analyze the changes in the outcome of the study thus extending the body of knowledge of this topic. Also, extensions to Dependency Check are welcome, that would cover the entire range of requirements elicited from the research context, or other requirements if study replications result in gathering different

requirements. Furthermore, we can extend the reliability study to not only the Maven ecosystem, but to Python's PyPi, and Ruby Gems. Besides the complete set of requirements, the ideal solution would also be capable of leveraging known vulnerabilities as they are disclosed, as it may happen that vulnerabilities are disclosed during the regular scheduled scans of software projects.

An interesting requirement that if solved, may produce much added value for the problem domain is providing update consequences in the case there exists newer versions of detected libraries in which known vulnerabilities are fixed. This may mean, for example, messages such as "if you update library X to this specific version, you will remove this vulnerability but as a consequence you have to update also libraries Y and Z". In this regard, the application of *association rule mining* techniques on a large relevant dataset such as the Maven projects with all their versions may give an impression of how often this is the case in practice, how many times the update of specific libraries lead to the update of the same others. The frequent cases in which specific libraries were updated at the same time can be studied to see how these situations can be recognized automatically and how can this functionality be integrated in tools.

8.2.4 Prevalence studies

The prevalence studies that we conducted in order to understand how common is the problem of depending on vulnerable components in practice can be extended to projects written in different programming languages and built using different build systems. This would enable us to understand whether the problem is prevalent across programming ecosystems or does it represent a concern only for specific language ecosystems. In our study we focused on Java projects built with Maven, but the study can be extended to for example, Python projects which declare dependencies in *requirements.txt* files and use PyPi as a dependency manager.

8.2.5 Risk analysis

The question that we set with regards to the concept of risk analysis is whether or not by having a library or framework with known vulnerabilities within the software system makes the project exploitable. We have generally seen throughout our research that CVE data makes reference to very intricate execution contexts in which the vulnerabilities can be exploited, but quantitative data is needed to study this aspect using real software systems.

Bibliography

[1] About CVE identifiers. http://cve.mitre.org/cve/identifiers/. Last visited 2014-01-19.

[2] CVSS Home Page at NVD NVD-CVSS. http://nvd.nist.gov/cvss.cfm. Last visited 2013-12-02.

[3] CWE description document. https://cwe.mitre.org/documents/views/view-evolution.html. Last visited 2014-08-07.

[4] Hasso-Plattner-Institut – database for vulnerability analysis. Last visited 2014-08-03.

[5] Heartland security breach. http://www.computerworld.com/s/article/9176507/Heartland_breach_expenses_pegged_at_140M_so_far. Last visited 2014-01-19.

[6] Maven - POM Reference. https://maven.apache.org/pom.html. Last visited 2014-03-13.

[7] Mitre homepage. http://www.mitre.org/. Last visited 2014-01-19.

[8] OWASP - Using Components with Known Vulnerabilities. https://www.owasp.org/index.php/Top_10_2013-A9-Using_Components_with_Known_Vulnerabilities. Last visited 2014-01-19.

[9] OWASP Dependency Check Home Page. https://www.owasp.org/index.php/OWASP_Dependency_Check. Last visited 2014-03-13.

[10] OWASP top 10 Home Page. https://www.owasp.org/index.php/Top_10_2013-Table_of_Contents. Last visited 2014-01-19.

[11] Portland Pattern Repository: Component Definition. http://c2.com/cgi/wiki?ComponentDefinition. Last visited 2014-01-19.

[12] Omar H. Alhazmi, Yashwant K. Malaiya, and Indrajit Ray. Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers & Security*, 26(3):219–228, 2007.

[13] Luca Allodi and Fabio Massacci. A preliminary analysis of vulnerability scores for attacks in wild. In *Proceedings of the ACM CCS Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, 2012.

[14] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. Experience factory. *Encyclopedia of software engineering*, 1994.

[15] Dennis Bijlsma, José Pedro Correia, and Joost Visser. Automatic event detection for software product quality monitoring. In *2012 Eighth International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 30–37. IEEE, 2012.

[16] Leyla Bilge and Tudor Dumitras. Before we knew it: an empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 833–844. ACM, 2012.

[17] Roland Bodenheim, Jonathan Butts, Stephen Dunlap, and Barry Mullins. Evaluation of the ability of the shodan search engine to identify internet-facing industrial control devices. *International Journal of Critical Infrastructure Protection*, 7(2):114–123, 2014.

[18] Andre Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, pages 195–203. ACM, 2000.

[19] Eric Bouwers, Arie van Deursen, and Joost Visser. Evaluating usefulness of software metrics: an industrial experience report. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 921–930. IEEE Press, 2013.

[20] Mehran Bozorgi, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. Beyond heuristics: learning to classify vulnerabilities and predict exploits. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 105–114. ACM, 2010.

[21] Cheikes Brant, David Waltermire, and Karen Scarfone. Common Platform Enumeration: Naming Specification Version 2.3. *NIST Interagency Report 7695*, 2011.

[22] Andrew Buttner and Neal Ziring. Common platform enumeration (CPE) specification. `http://cpe.mitre.org/files/cpe-specification_2.1.pdf`. Last visited 2014-04-30.

[23] Brian Chess and Jacob West. *Secure programming with static analysis*. Pearson Education, 2007.

[24] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating software architectures*. Tsinghua University Press, 2003.

[25] DragonSoft. DragonSoft Vulnerability Database Home Page. `http://vdb.dragonsoft.com/`. Last visited 2013-11-29.

[26] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*, pages 285–311. Springer, 2008.

[27] Tom Fawcett and Foster Provost. Activity monitoring: Noticing interesting changes in behavior. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 53–62. ACM, 1999.

[28] Stefan Frei. The known unknowns. *NSS Labs*, 2013.

[29] Anandasivam Gopal, Tridas Mukhopadhyay, and Mayuram S. Krishnan. The impact of institutional forces on software metrics programs. *IEEE Transactions on Software Engineering,*, 31(8):679–694, 2005.

[30] Gregor Hohpe and Bobby Woolf. Enterprise integration patterns. In *9th Conference on Pattern Language of Programs*, pages 1–9, 2002.

[31] Will Hopkins. A new view of statistics. *Internet Society for Sport Science*, 2000.

[32] IBM. IBM XForce IBM Internet Security Systems. `http://search.iss.net/`. Last visited 2013-11-29.

[33] British Standard Institute. Information technology – security techniques – management of information and communications technology security – part 1: Concepts and models for information and communications technology security management bs iso/iec 13335-1-2004. 2004.

[34] ISO/IEC. Information technology – security tecniques-information security risk management. *ISO/IEC FIDIS 27005*, 2008.

[35] Arnav Joshi, Ravendar Lal, Tim Finin, and Anupam Joshi. Extracting cybersecurity related linked data from text. In *2013 IEEE Seventh International Conference on Semantic Computing (ICSC),*, pages 252–259. IEEE, 2013.

[36] Wei Ming Khoo, Alan Mycroft, and Ross Anderson. Rendezvous: A search engine for binary code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 329–338. IEEE Press, 2013.

[37] Barbara A. Kitchenham and Shari Lawrence Pfleeger. Principles of survey research: part 3: constructing a survey instrument. *ACM SIGSOFT Software Engineering Notes*, 27(2):20–24, 2002.

[38] Tobias Kuipers and Joost Visser. A tool-based methodology for software portfolio monitoring. In *Software Audit and Metrics*, pages 118–128, 2004.

[39] Tobias Kuipers, Joost Visser, and Gerjon De Vries. Monitoring the quality of outsourced software. In *Proc. Int. Workshop on Tools for Managing Globally Distributed Software Development (TOMAG 2007). Center for Telematics and Information Technology (CTIT), The Netherlands*, pages 3–12, 2007.

[40] David Mann and Steven Christey. Towards a common enumeration of vulnerabilities. *The MITRE Corporation*, 4(5), 1999.

[41] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[42] Fabio Massacci and Viet Hung Nguyen. Which is the right source for vulnerability studies? an empirical analysis on mozilla firefox. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, pages 4–12. ACM, 2010.

[43] Peter Mell, Karen Scarfone, and Sasha Romanosky. A complete guide to the common vulnerability scoring system version 2.0. In *Published by FIRST-Forum of Incident Response and Security Teams*, pages 1–23, 2007.

[44] Dimitris Mitropoulos, Georgios Gousios, and Diomidis Spinellis. Measuring the occurrence of security-related bugs through software evolution. In *2012 16th Panhellenic Conference on Informatics (PCI)*, pages 117–122. IEEE, 2012.

[45] Dimitris Mitropoulos, Vassilios Karakoidas, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. Dismal code: Studying the evolution of security bugs. In *Proceedings of the LASER Workshop*, pages 37–48, 2013.

[46] Dimitris Mitropoulos, Vassilios Karakoidas, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. The bug catalog of the maven ecosystem. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 372–375. ACM, 2014.

[47] Varish Mulwad, Wenjia Li, Anupam Joshi, Tim Finin, and Krishnamurthy Viswanathan. Extracting information about security vulnerabilities from web text. In *2011 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, volume 3, pages 257–260. IEEE, 2011.

[48] Stephan Neuhaus and Thomas Zimmermann. Security trend analysis with cve topic models. In *2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*, pages 111–120. IEEE, 2010.

[49] NIST. National Vulnerability Database Home Page NVD - Home. `http://nvd.nist.gov/`. Last visited 2013-11-29.

[50] Timothy M. O'Brien. *Maven: The Definitive Guide*. O'Reilly, 2008.

[51] International Institute of Business Analysis. *A Guide to the Business Analysis Body of Knowledge.* 2009.

[52] OSVDB. Open Source Vulnerability Database Home Page. `http://osvdb.org/`. Last visited 2013-11-29.

[53] Theodoros Polychniatis. Detecting dependencies across programming languages. Master's thesis, Utrecht University, 2012.

[54] Steven Raemaekers, Arie van Deursen, and Joost Visser. The maven repository dataset of metrics, changes, and dependencies. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 221–224. IEEE Press, 2013.

[55] Steven Raemaekers, Gabriela F. Nane, Arie van Deursen, and Joost Visser. Testing principles, current practices, and effects of change localization. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 257–266. IEEE Press, 2013.

[56] Steven Raemaekers, Arie van Deursen, and Joost Visser. Measuring software library stability through historical version analysis. In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 378–387. IEEE, 2012.

[57] Sanaz Rahimi and Mehdi Zargham. Vulnerability scrying method for software vulnerability discovery prediction without a vulnerability database. *IEEE Transactions on Reliability*, 62(2):395–407, 2013.

[58] Rapid7. Rapid7 Vulnerability and Exploit Database. `http://www.rapid7.com/db/`. Last visited 2013-11-29.

[59] Sebastian Roschke, Feng Cheng, Robert Schuppenies, and Christoph Meinel. Towards unifying vulnerability information for attack graph construction. In *Information Security*, pages 218–233. Springer, 2009.

[60] Vaibhav Saini, Hitesh Sajnani, Joel Ossher, and Cristina V Lopes. A dataset for maven artifacts and bug patterns found in them. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 416–419. ACM, 2014.

[61] Theodoor Scholte, Davide Balzarotti, and Engin Kirda. Have things changed now? an empirical study on input validation vulnerabilities in web applications. *Computers & Security*, 31(3):344–356, 2012.

[62] Theodoor Scholte, Davide Balzarotti, and Engin Kirda. Quo vadis? a study of the evolution of input validation vulnerabilities in web applications. In *Financial Cryptography and Data Security*, pages 284–298. Springer, 2012.

[63] Secunia. Secunia Advisories. `http://secunia.com/advisories/`. Last visited 2013-11-29.

[64] Securiteam. Securiteam Database CVE. `http://www.securiteam.com/cves/`. Last visited 2013-11-29.

[65] SEI. SEI Security Notes Search Vulnerability Notes. `http://www.kb.cert.org/vuls/html/search/`. Last visited 2013-11-29.

[66] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X. Liu. A large scale exploratory analysis of software vulnerability life cycles. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 771–781. IEEE Press, 2012.

[67] Shoban Search Engine. Jetty 6.1.1 Keyword Search on Shoban. `http://www.shodanhq.com/search?q=jetty+6.1.1`. Last visited 2014-04-30.

[68] Abraham Silberschatz, Peter Galvin, and Greg Gagne. *Operating system concepts*, volume 4. Addison-Wesley Reading, 1998.

[69] Symantec. Symantec Security Focus. http://www.securityfocus.com/vulnerabilities. Last visited 2013-11-29.

[70] Clemens Szyperski. *Component software: beyond object-oriented programming*. Pearson Education, 2002.

[71] Orcun Temizkan, Ram L. Kumar, SungJune Park, and Chandrasekar Subramaniam. Patch release behaviors of software vendors in response to vulnerabilities: an empirical analysis. *Journal of Management Information Systems*, 28(4):305–338, 2012.

[72] Arie Van Deursen and Tobias Kuipers. Source-based software risk assessment. In *International Conference on Software Maintenance 2003*, pages 385–388. IEEE, 2003.

[73] Roel Wieringa and Joel Heerkens. The methodological soundness of requirements engineering papers: a conceptual framework and two case studies. *Requirements engineering*, 11(4):295–307, 2006.

[74] Roel Wieringa, Neil Maiden, Nancy Mead, and Colette Rolland. Requirements engineering paper classification and evaluation criteria: a proposal and a discussion. *Requirements Engineering*, 11(1):102–107, 2006.

[75] James Williams and Anand Dabirsiaghi. The unfortunate reality of insecure libraries. *Aspect Security, Inc.*, March 2012.

[76] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer, 2012.

[77] Haiyun Xu, Jeroen Heijmans, and Joost Visser. A practical model for rating software security. In *SERE (Companion)*, pages 231–232, 2013.

[78] Pei Xia Makoto Matsushita Norihiro Yoshida and Katsuro Inoue. Studying reuse of out-dated third-party code in open source projects. *Computer Software (translated from Japanese)*, 30(4):98–104, 2013.

[79] Su Zhang, Doina Caragea, and Xinming Ou. An empirical study on using the national vulnerability database to predict software vulnerabilities. In *Database and Expert Systems Applications*, pages 217–231. Springer, 2011.

Appendix A

Interview-Requirements Introductory Text

Developers, in their practice, tend to avoid "reinventing the wheel". The strongest justification for this behavior is the current time-to-market demands of the software industry. This results in developers being inclined to firstly search for a third-party software library that covers part of the project requirements, before attempting to develop the necessary functionality from scratch. In some cases, library code makes up for as much as 80% of the total size of the total project code [75].

However, in software development, with regards to including third-party libraries in projects, "you are what you eat". Along with the desired functionality, we include the library's problems too, such as security vulnerabilities, which could be exploited by malicious attackers to gain the full privileges of the application, for their own use. In a recent study, researchers at Aspect Security analysed 113 million downloads from the Maven Central Repository and declared that 29.8 million (26%) of the library downloads have previously disclosed vulnerabilities[75]. In this context it should be noted that, probably, not all downloaded dependencies were actually used as project dependencies in the development process, but the fact that more than 1 out of 4 library downloads have a known exploitable vulnerability illustrates the gravity of the situation.

What would be desirable in this context, is to establish and evaluate a method to automatically leverage disclosure data, to inform the developers of the known security risks associated with their library selection throughout the evolution of a project, and inform them of possible solutions, such as upgrading to a newer version in which the security vulnerability was removed.

Appendix B

Interview Form

Question: What are the requirements for an artefact that would enable you to monitor the presence of known vulnerabilities in third party libraries built as an extension to the Monitor Alert System?

Requirement	Priority

Appendix C

Requirements Formulation and Justification

In this appendix, we present a more detailed description of the gathered requirements. The list below is not sorted.

Basic requirement –

Formulation:
The consultant shall be notified through an alert when:

- A vulnerable library is included in a snapshot (new snapshots of the current state of the source code are received every week).
- A new vulnerability is disclosed that affects one of the dependent libraries of a monitored project.

Justification:
Traditionally, consultants have to spend a large amount of time manually identifying library versions and investigating security disclosure information sources at each snapshot upload. These alerts alleviate them from manually keeping track of included libraries and vulnerability information.

Up-to-date and reliable vulnerability data source –

Formulation:
The data source utilized shall always include the most recent vulnerability information, coming from a reliable entity.

Justification:
Through implementing this requirement, the risk of omitting a disclosed vulnerability of which the artefact is not aware of, is avoided. Furthermore, if the data source is coming from a reliable entity it assures that consultants trust the artefact's output and utilize it in their practice.

Accuracy –

Formulation:
In “automatic detection” use-cases such as ours – detecting vulnerable libraries – there will be proposals in the artefact’s output which are relevant (true positives) or not relevant (false positives). The solution shall provide a “useful ratio” between true positives and false positives.

Justification:
False positives are to be expected, but they do not provide useful information for the consultant and may also lead to useless investigations and time spent pointlessly.

Vulnerability Severity –

Formulation:
The report included in every alert shall contain indication of the vulnerability’s severity level.

Justification:
Knowing the severity of a vulnerability is very important in the consultancy practice, as it allows the consultant to understand how large associated risk is.

Enriching the vulnerability data –

Formulation:
It shall be possible to manually extend the initial data source used by the artefact for including new vulerability information on already existing libraries in the data store, or create new library entries with respective vulnerability information.

Justification:
The consultants may read online sources, such as blog posts, and they should be able to integrate the new information in the artefact.

Filtering –

Formulation:
It shall be possible to define custom filters that suppress alerts, according to defined custom rules.

Justification:
This feature enables alerts pertaining to a specific library to be ignored, for instance, when the consultant wants to “acknowledge” certain alerts so that they don’t reoccur at every snapshot upload if the vulnerable library was not removed.

Multi-language support –

Formulation:
It shall be possible to support multiple programming languages – the initial range of languages that is to be supported is Java and C#, as the majority of currently monitored projects are built using these programming languages.

Justification:
The projects evaluated by the consultants are written in various programming languages, therefore the tool should have the ability to be applied in a broad range of situations.

Confidentiality –

Formulation:
The data source provider shall not be able to obtain information on which libraries/vulnerabilities the monitored projects include.

Justification:
In the extreme case, the information about the projects' vulnerabilities could be used in order to compromise the clients' projects and buisnesses.

Ease of use –

Formulation:
The artefact shall be easily deployed to monitor a project.

Justification:
The consultants will not be inclined to use the artefact, and will resort to alternative solutions, if it requires significant overhead to be deployed for monitoring a project, such as excessive configuration.

Easily extensibile to other languages –

Formulation:
It shall be easy to extend the artefact to be used on a currently unsupported technology.

Justification:
This feature is important because there are instances where the consultants encounter a project to be monitored being written in a programming language that they have not encountered before, so they should be able to easily extend the artefact to provide support for this new situation.

Robustness –

Formulation:
Robustness is “the ability of a computer system to cope with errors during execution or the ability of an algorithm to continue to operate despite abnormalities in input or calculations”[68].

Justification:
This requirement prevents the situation in which, due to an error caused by one software project, all of the currently monitored projects are affected.

Detect Outdated Libraries and provide upgrade suggestion –

Formulation:
The artefact shall be capable of indicating that a newer version of a used library is available, in which the vulnerability is removed.

Justification:
This feature removes the need to manually investigate the version information of a library and then check whether a newer version is available, that does not contain the vulnerability.

Detailed information about the consequences of upgrade –

Formulation:
In the context of a possible upgrade scenario, the artefact shall provide the consultant with detailed insight into the consequences of upgrade, such as the cost of making the application “compatible” with the new library version.

Justification:
This feature removes the need for the consultant to conduct a trade-off analysis, with regards to the development consequences of upgrading to a newer version, before advising the client.

Scalability –

Formulation:
Scalability is “the ability of a system, network, or process to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth”[18].

Justification:
Scalability is important due to the fact that the number of projects that have to be monitored is consistently increasing.

Proprietary libraries –

Formulation:
The system shall be able to signal the presence of known vulnerabilities in proprietary libraries.

Justification:
Developers do not only utilize open-source libraries, but also proprietary libraries, therefore the system should be able to accommodate for this fact.

Automation –

Formulation:
The amount of manual work that operating the artefact requires shall be kept to a minimum. The ideal solution is automated, where possible.

Justification:
By reducing the manual work required to operate the artefact, we improve the consultants' efficiency.

Appendix D

Interview - Vulnerability Databases

D.1 Introduction

The purpose of this research is to develop, implement and evaluate a method to scan and monitor the presence of known vulnerabilities in third party dependencies of software projects.

Vulnerability information is stored in so-called vulnerability databases, which collect the publicly disclosed vulnerabilities coming from the application vendors and other entities.

Vulnerability information is encapsulated in these databases in a non-uniform manner - a common schema is not enforced, and moreover their contents differs too. The first step of the applied research method is to match the technical consultants requirements regarding the vulnerability information needed in applying the security practice with the vulnerability description contained in the vulnerability databases.

The result of this interview counts for understanding whether there is a need to unify the vulnerability databases using the common CVE identifier to gain more descriptive power, or using a single data source would suffice, given that the project has a finite deadline and not all databases offer straightforward programmatic access.

I would really appreciate if you can rate the importance of each vulnerability description element in the table found on page 3, labeling each as one of the following: MUST, SHOULD, COULD, WONT. These values pertain to the MoSCoW method for software engineering requirements prioritization, and are explained below :

- MUST: Describes a requirement that must be satisfied in the final solution for the solution to be considered a success.
- SHOULD: Represents a high-priority item that should be included in the solution if it is possible. This is often a critical requirement but one which can be satisfied in other ways if strictly necessary.
- COULD: Describes a requirement which is considered desirable but not necessary. This will be included if time and resources permit.
- WON'T: Represents a requirement that stakeholders have agreed will not be implemented in a given release, but may be considered for the future. (note: occasionally the word "Would" is substituted for "Won't" to give a clearer understanding of this choice)

D.2 Result

	MUST	SHOULD	COULD	WONT
CVE ID				x
CWE ID		x		
Impact	x			
CVSS		x		
Description			x	
Solution		x		
Attack From		x		
Popularity			x	
Discoverer				x
Loss Type			x	
Vulnerability Type			x	
Similar Vulns			x	
OS		x		
CPE ID			x	
Vulnerable Software	x			
Publishing Date			x	
Date Public			x	
Date Last Update		x		
References		x		

Table D.1: Vulnerability Database Interview Results Table