

Letterkenny Institute of Technology

A thesis submitted in partial fulfilment of

the requirements for the Master of Science in Computing in

Enterprise Applications Development Letterkenny Institute of Technology

A comparison of vulnerabilities found in source libraries against third party dependencies

Author: Supervisor:

Ben Craig Ruth Lennon

Submitted to the Higher Education and Training Awards Council (HETAC)

May 2014

**Declaration**

I hereby certify that the material, which I now submit for assessment on the programmes of study leading to the award of ***Master of Science in Computing in Enterprise Application Development***, is entirely my own work and has not been taken form the work of others except to the extent that such work has been cited and acknowledged within the text of my own work. No portion of the work contained in this thesis has been submitted in support of an application for another degree or qualification to this or any other institution.

Signature of candidate: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Acknowledgements**

This thesis would not have been possible without the support of many people and I would like to thank all those who assisted me in its completion.

# Abstract

Recent research by Williams and Dabirsiaghi (2012) suggests that many software projects that are in use today are unknowing Trojan horses. The use of many and multiple third party libraries greatly increase the risk that a given software project has become exposed to a known vulnerability. On the other hand, it has been claimed by Yu et al (2011) that the general ability of programmers to write secure code is inadequate. Developer awareness of the need to write secure application code is surprisingly low.

This dissertation examines the effects of using these third party libraries. A large range of open source projects are examined and then compared against their third party dependencies. This is to determine whether more security vulnerabilities are exposed via third-party dependencies, or by the source code that uses those dependencies.

This is an important area of research, as it could help guide future policy in dealing with third-party library usage. From an enterprise perspective, it is important to determine how best to ensure delivery of secure applications to customers. From a business perspective, it is important to keep delivery costs down, while preserving security and without compromising customer integrity. Usage of third-party libraries helps keep costs down, but must be compared to the possibility of introducing potential vulnerabilities.

Software development companies must decide which provides the better investment; usage of third-party libraries that may be cost-effective but vulnerable, or training staff to write more secure code. This dissertation attempts to find out if more attention is needed in educating developers to write more secure code or highlighting the potential pitfalls in using common third-party libraries.

# Table of Contents

[Abstract III](#_Toc385926175)

[Table of Contents IV](#_Toc385926176)

[List of Figures VII](#_Toc385926177)

[List of Tables VIII](#_Toc385926178)

[Code Listings IX](#_Toc385926179)

[Abbreviations X](#_Toc385926180)

[Glossary of Terms XI](#_Toc385926181)

[1 Introduction 1](#_Toc385926182)

[1.1 Background and Objective 2](#_Toc385926183)

[1.2 Problem Statement 3](#_Toc385926184)

[1.3 Aims 3](#_Toc385926185)

[1.4 Hypothesis 4](#_Toc385926186)

[2. Survey 5](#_Toc385926187)

[2.1 Third Party Dependencies 5](#_Toc385926190)

[2.2 Using Components with Known Vulnerabilities 5](#_Toc385926191)

[2.3 Cataloguing these Vulnerabilities 8](#_Toc385926192)

[2.4 Available tools 9](#_Toc385926193)

[2.5 Heartbleed Bug 10](#_Toc385926194)

[3 Design 13](#_Toc385926195)

[3.1 Approach to analysis 13](#_Toc385926196)

[3.2 Notes on the Open Source projects used 13](#_Toc385926197)

[3.3 Notes on the Analysis tools used 14](#_Toc385926198)

[3.4 Dependency Check Analyser – Java Utility 14](#_Toc385926199)

[3.4.1 Requirements 14](#_Toc385926200)

[3.4.2 Architecture 15](#_Toc385926201)

[3.4.3 UML Diagrams 16](#_Toc385926202)

[3.4.4 Design considerations 20](#_Toc385926203)

[3.4.5 Analysis Strategies 21](#_Toc385926204)

[4 Implementation 24](#_Toc385926205)

[4.1 Tools and Components Used 24](#_Toc385926206)

[4.1.1 Dependency Check Tool 24](#_Toc385926207)

[4.1.2 Microsoft Threat Modelling Tool 2014 24](#_Toc385926208)

[4.2 Project Implementation 25](#_Toc385926209)

[4.3 Analysis Implementation 25](#_Toc385926210)

[4.3.1 Phase One Analysis Implementation 25](#_Toc385926211)

[4.3.2 Phase Two Analysis Implementation 26](#_Toc385926212)

[4.3.3 Phase Three Analysis Implementation 27](#_Toc385926213)

[4.3.4 Phase Four Analysis Implementation 28](#_Toc385926214)

[5 Testing and Results 37](#_Toc385926215)

[5.1 Phase One Results 37](#_Toc385926216)

[5.2 Phase Two Results 40](#_Toc385926217)

[5.2.1 Findbugs 42](#_Toc385926218)

[5.2.2 gt2 43](#_Toc385926219)

[5.2.3 Netbeans 43](#_Toc385926220)

[5.3 Phase Three Results 44](#_Toc385926221)

[5.3.1 Top Ten Lists 45](#_Toc385926222)

[5.4 Phase Four Results 51](#_Toc385926223)

[6 Conclusions 52](#_Toc385926224)

[6.1 Initial Analysis 52](#_Toc385926225)

[6.2 Third Party Library Usage 52](#_Toc385926226)

[6.3 Analysis of open source projects 52](#_Toc385926227)

[6.4 Security Tools 53](#_Toc385926228)

[6.5 Vulnerability Identification System 53](#_Toc385926229)

[6.6 Zero vulnerabilities may not mean zero vulnerabilities 55](#_Toc385926230)

[7 References 56](#_Toc385926231)

[8 Appendices 59](#_Toc385926232)

[8.1 Appendix A - Projects used from the Qualitas Corpus 59](#_Toc385926233)

# List of Figures

[Figure 1: Quarterly Statistics 9](#_Toc385926234)

[Figure 2: The Heartbleed Bug 11](#_Toc385926235)

[Figure 3 : Application Architecture 15](#_Toc385926236)

[Figure 4: Analysis Controller Class Diagram 17](#_Toc385926237)

[Figure 5: Analysis Strategy Class Diagram 19](#_Toc385926238)

[Figure 6: XML Tree Strcture 21](#_Toc385926239)

[Figure 7: Duplicate Dependencies 22](#_Toc385926240)

[Figure 8: OWASP Threat Modelling Process (Owasp.org, 2014) 29](#_Toc385926241)

[Figure 9: The Struts Architecture 32](#_Toc385926242)

[Figure 10: Struts Data Flow Diagram 33](#_Toc385926243)

[Figure 11: Example output from Dependency Report 37](#_Toc385926244)

[Figure 12: Duplicate Vulnerabilities distorting Phase One analysis 38](#_Toc385926245)

[Figure 13: Percentage of Vulnerabilities Comparison 39](#_Toc385926246)

[Figure 14: Initial analysis 40](#_Toc385926247)

[Figure 15: Unique Third Party Vulnerabilities vs Unique Source Library Vulnerabilities 41](#_Toc385926248)

[Figure 16: Outliers removed 42](#_Toc385926249)

[Figure 17: Number of Vulnerabilities per Source Library compared to their respective Third Party Libraries. 48](#_Toc385926250)

[Figure 18: A vulnerability in the CVE database 53](#_Toc385926251)

[Figure 19: The same vulnerability in the NIST database 54](#_Toc385926252)

# List of Tables

[Table 1: Struts Threat Identification 34](#_Toc385926253)

[Table 2: Mapping threats to vulnerabilities 35](#_Toc385926254)

[Table 3: Number of Vulnerable Third Party Dependencies per Project 38](#_Toc385926255)

[Table 4: Vulnerable Source Libraries per Project 39](#_Toc385926256)

[Table 5: Number of Vulnerabilities per 3rd Party Library 45](#_Toc385926257)

[Table 6: The top ten projects with the highest ratio of vulnerabilities compared to their respective source libraries 47](#_Toc385926258)

[Table 7: Number of Vulnerabilities per Source Library 48](#_Toc385926259)

[Table 8: Projects with the highest number of unique vulnerable third party libraries 49](#_Toc385926260)

[Table 9: Projects with the highest number of vulnerable source libraries 50](#_Toc385926261)

[Table 10: Vulnerability Distribution 51](#_Toc385926262)

# Code Listings

[Code Listing 4.1: Some Code **Error! Bookmark not defined.**](#_Toc367433536)

# Abbreviations

CPE Common Platform Enumeration

CSS Cross-Site Scripting

CSRF Cross-Site Request Forgery

CSV Comma Separated Values

CVE Common Vulnerability and Exposure

CVSS Common Vulnerability Scoring System

CWE Common Weakness Enumeration

DBA Database Administrator

DFD Data Flow Diagram

DoS Denial-of-Service

GUI Graphical User Interface

GWT Google Web Toolkit

HTTP Hyper-Text Transfer Protocol

MVC Model-View-Controller

NIST National Institute of Standards and Technology

OWASP Open Web Application Security Project

SDL Security Development Lifecycle

SQL Structured Query Language

SSL Secure Socket Layer

URL Uniform Resource Locator

XML Extensible Markup Language

XSRF Cross-Site Request Forgery

XSS Cross-Site Scripting

# Glossary of Terms

|  |  |
| --- | --- |
| Dependency | A Third Party Library. |
| Library | A Jar File. This can be either a Dependency, or a Source Jar File. |
| Project | A single body of source code and accompanying artefacts, such as documentation and execution scripts. It has Dependencies and can be built into Source Jar Files. |
| Source Jar File | A Jar File that is produced by building a Project. |
| Vulnerability | An identified bug in a Library that could result in a security threat. |

# Introduction

Modern programming techniques depend upon third party libraries. The culture of code reuse provides many benefits, such as avoiding the need to re-invent the wheel. However, one downside of this approach is that the security of third party libraries are generally hidden from the development teams that are using those libraries. Generally, a library will be used to perform a common, well-understood task, such as email functionality or XML parsing. However the level of confidence that a developer can have in third party libraries is comparatively low.

Does this problem show up due to the usage of open-source library implementations? One might imagine that proprietary applications developed in-house by reputable development houses should be free from security flaws. However the evidence, which we will study in the literary review and in the analysis, would appear to contradict this assumption.

Proprietary products suggest that they provide 'security through obscurity', since no-one can access their source code. The evidence appears to contradict this claim. In contrast, open-source software believes in the 'many eyes' approach to security. This entails having as many application developers looking at the code for security holes, testing and verifying for themselves. The evidence appears to suggest that this can result in a much more robust system design, which will be examined later in this dissertation. Kerckhoffs’s principle (Kerckhoffs, 1883) roughly states that “a system's security should be provided by the strength of its algorithm, not by keeping the algorithm secret".

One exception to the rule is the recent case of the Heartbleed bug. This was an issue discovered in an open source project, which was itself used by a large number of dependent projects and websites, such as Twitter and Facebook. This issue is discussed in more detail in Section 2.5.

This paper analyses many open source projects, identifying many different types of security vulnerabilities in the process. These are all discussed and examined in the Literary Survey.

## Background and Objective

The use of third party libraries with known vulnerabilities has previously been documented by OWASP as a Security Misconfiguration issue. The 2013 Top Ten upgraded this problem to a vulnerability of its own, citing the fact that component-based development has increased in recent years. Component-based development centres on the idea of reusing components, resulting in heavy dependency on third party libraries. However, a recent survey by Williams and Dabirsiaghi (2012) suggests that up to 26% of all downloaded libraries contain known vulnerabilities. It also been noted (OWASP, 2013) that many developers are not even aware of all of the dependencies that their application is using. This is a worrying trend, and indicative of the need for education in this area.

A second interesting aspect of this particular vulnerability is that it can encompass any of the other nine vulnerabilities identified by OWASP in their Top Ten list. For example, a third party library may contain an SQL Injection vulnerability. Usage of this library would put the developer’s own code at risk of an SQL Injection attack. In the author’s eyes, this is definitely a cause for concern, and another reason to examine this topic in details.

The author is also interested in the area of code quality, especially using static analysis tools to gauge the quality of a code base. This type of tool typically examines the quality of the code, but fails to examine the security aspects of the code. The author hopes to examine tools that are freely available and how they can be used to assess how secure a software project is.

Finally, the analysis of vulnerable software would not be complete without some examination of threat modelling. This exercise helps to identify threats that could exist based on an abstract definition of a system. This is useful in guiding and educating developers in writing secure software. Coupling threat modelling with vulnerability identification…

## Problem Statement

The OWASP Top Ten 2013 contained a new entry, namely “using components with known vulnerabilities”. The author believed that this would be a fertile ground for research, and that this area would provide opportunities for examining an area that may not have had as much focus in the past.

The author believes that it would be helpful to distinguish between vulnerabilities in third party libraries and vulnerabilities that are introduced by developers in their own code. Is it more likely that an attack might originate from reusing open-source libraries, or are developers more likely to introduce these vulnerabilities in source code that they are responsible for writing? The research question of particular note for this work is: “Do open source projects contain more security vulnerabilities in their third party dependencies or in their own inherent code?”

This is an important question to try to answer, because it may help identify if the quality of industry code is more secure or less secure than the open source projects that many of these systems are built upon. The privatised software industry may have a lot to learn from open source projects. It may be impossible for private software houses to allow open access to their source code, but approaches such as bug-finding days and outsider review sessions may help incorporate some of the ideas in a relatively safe way.

## Aims

* To prove that the inherent source code of open source projects contain more security vulnerabilities than their third party dependencies.
* This thesis investigates vulnerabilities resultant from failure to keep third party libraries up-to-date.
* Techniques and resources available to examine and scan source code and libraries are presented.

## Hypothesis

It is the author’s belief that the source code that developers themselves write will contain more vulnerabilities than the open source libraries that their projects are dependent on. In the author’s own experience of working on multiple enterprise software projects, the tendency has been noted that the developers are traditionally not very well educated on the specificities of writing secure code. Third party dependencies on the other hand have the benefit of the ‘many eyes’ approach to secure coding. The hypothesis for this thesis is:

The inherent source code of open source projects contains more security vulnerabilities than their third party dependencies.

# Survey

This section contains a literary review of the current state of third party libraries, common attacks attempted upon them, and previous research work in this field. Tools and resources that are available to developers in order to assist them in writing secure applications are also discussed.

# 



## Third Party Dependencies

Reusable components have become a regular fixture of modern software development. Object oriented programming has always included the idea of reuse at its core. Coupled with the open-source software movement and licensing agreements such as the GNU General Public License, it heralded a new era of software development sometimes labelled Component Based Software Engineering.

The benefits of reuse have been well documented (Li et al. 2007), (Mohagheghi et al. 2007). Developers did not have to reinvent the wheel, and productivity could increase by using well-known and well-tested components that were freely available. The use of open source code also meant the developers could examine the third-party libraries that they were using, and even contribute to those projects in order to help other developers.

However, alongside those benefits, there were also some disadvantages to reusing open source code. Some licensing agreements meant that if an enterprise development firm were to use open source code in its proprietary solutions, then it was forced to publish all code that used it. A second problem was the possibility of bugs or security issues in the third party libraries. Some of the open source projects were actively being worked on and patched; however many became dormant for various reasons. This often left development teams with potential issues hidden deep in their own products.

## Using Components with Known Vulnerabilities

This section will largely stay, probably with some editing & re-writing.

OWASP recognised this issue in their 2013 Top Ten Vulnerability list, adding a new vulnerability. This is the use of components with known vulnerabilities, and it is the primary focus of this paper.

Modern programming practice results in a high dependency on third-party applications. Security issues can occur when programmers fail to realise that the libraries and frameworks that they are using can contain known security flaws. Since these flaws are published and well-known, it is relatively easy for a hacker to exploit these types of vulnerabilities. The question has to be asked, why would a software company allow the usage of such potentially damaging libraries?

One potential reason may be just a lack of awareness. Companies often encourage the use of open-source software as a cheap alternative to writing their own components, and indeed this does make sense. Coupled with this is the idea of the "many eyes" approach to security; this refers to the fact that many eyes have looked at the code, and quite a lot of bug fixes and security patches would have been implemented based on that feedback.

However the "many eyes" advantage completely falls over if a company fails to implement steps to keep their libraries and components up-to-date. So what prevents this from happening?

The simplest solution may be that developers are simply under too much pressure to fix their own bugs and develop their own enhancements that they cannot devote the necessary time to updating their dependencies. Business analysts and sales people often used new features and enhancements to generate revenue; it is unlikely that a new release of a product whose changelog simply reads “updated third-party libraries” will generate much in the way of revenue.

Another consideration is how can developers determine if they are at risk? In terms of open-source libraries, OWASP provides some useful tools to help determine the risk level. One such tool is the OWASP Dependency Check, which is a tool that scans a Java project, determines what third party libraries and versions are being used, and then correlates that information with several online databases that are used to catalogue these types of issues. The tool then produces a report describing the potential risks that may be hidden in their software.

However for commercial libraries it may be more difficult to determine the level of risk. Commercial vendors are much less likely to publish known vulnerabilities, even in much older versions of their libraries. The reason for this is obvious; since the tool generates revenue for the company, they do not want to publicise anything that may result in a negative light being cast on that product.

Interestingly, Williams and Dabirsiaghi (2012) have suggested the four most downloaded vulnerable libraries are:

* GWT
* Xerces
* Spring MVC
* Struts 1.x

This puts into perspective how even big players in the open source world are affected by vulnerable code.

Consider also the fact that third-party libraries are generally run with all the permissions that are afforded to the application itself. This means that if a user were to install a Java application that used Spring, and the application required administrator user privileges, the vulnerable aspects of the Spring code may also be afforded the admin user rights.

In the previous release of the OWASP Top Ten, using unsecure components was included as part of Security Misconfiguration. However, with the latest release, the company decided to give this particular vulnerability a section entirely of its own.

## Cataloguing these Vulnerabilities

Developers are typically unaware of the types of vulnerabilities that can exist in third-party libraries. With this in mind, efforts have been made to start cataloguing these issues in public databases. Some examples of these databases are:

* <http://nvd.nist.gov/>
* <http://www.exploit-db.com/>
* <http://cve.mitre.org/>
* <http://www.osvdb.com/>

The first database is the US Government’s attempt at cataloguing and identifying these issues. It contains close to 60,000 known and published vulnerabilities across a whole host of platforms and software types.

The Exploit-DB website contains information on approximately 28,000 exploits; this particular resource is aimed more at penetration testers.

CVE stands for Common Vulnerability and Exposure. It aims to provide a standard approach to how the vulnerabilities are catalogued and defined. On the 1st of January 2014 it adjusted the syntax for reporting vulnerabilities. This was primarily due to an initial restriction in how bugs were reported. The original ID allow for a max of 4 digits in the vulnerabilities identifier, for a single calendar year. This meant a maximum of 9,999 bugs could be identified each year. The rule has been adjusted to ensure a minimum of four digits, but allowing expansion up to any size. The syntax further defines a description, status, references, votes from the community, and comments from the community.

The OSVDB database is focused solely on open source software. It currently claims to have reports on over 100,000 vulnerabilities. It also provides statistical information per quarter on which types of vulnerabilities have been reported most.



Figure 1: Quarterly Statistics

## Available tools

OWASP provide a number of utilities in helping to determine a project’s potential level of risk. One particularly useful utility is the OWASP Dependecy Check Tool. This tool scans packaged code files, such as JAR, WAR and EAR files in a Java setting, and checks whether they contain any known and published vulnerabilities. It checks the particular JAR file, determines the project and the version, then correlates that information with the list of vulnerabilities as published by the CVE website.

This tool was used to analyse several popular open-source projects that are in popular use today. The results are as follows:

<TODO: add these results in a lovely table>

<TODO: This will all actually go into results section>

After using the OWASP Dependency Check tool for some time, it became clear that it had some limitations, at least in respect of the analysis that was being performed for this dissertation. The tool itself is useful for individual project analysis but is not intended to be used to amalgamate results from multiple projects, and subsequently provide reports on those various projects. These limitations meant that a bespoke tool was required, one that could take the output from multiple runs of the OWASP Dependency Tool and combine them, allowing for direct comparison of the same properties across multiple projects, as well across third-party libraries and the relevant source libraries. Section 3 contains information on the design of this bespoke tool, while Section 4 includes details on its implementation.

## Heartbleed Bug

In early April 2014, a security flaw was identified in OpenSSL. It has typically been referred to as the “Heartbleed Virus”; however this label is incorrect, since the problem is exploitable through human error rather than an external, malicious program.

The bug was in relation to how OpenSSL implemented its heartbeat functionality, hence the name “Heartbleed”. It has been identified as a programming error on a single line of code, which “leaves all forms of Internet data open to hackers”. (TODO: add reference to web page). It exploits OpenSSL’s heartbeat, as illustrated in Figure 2.

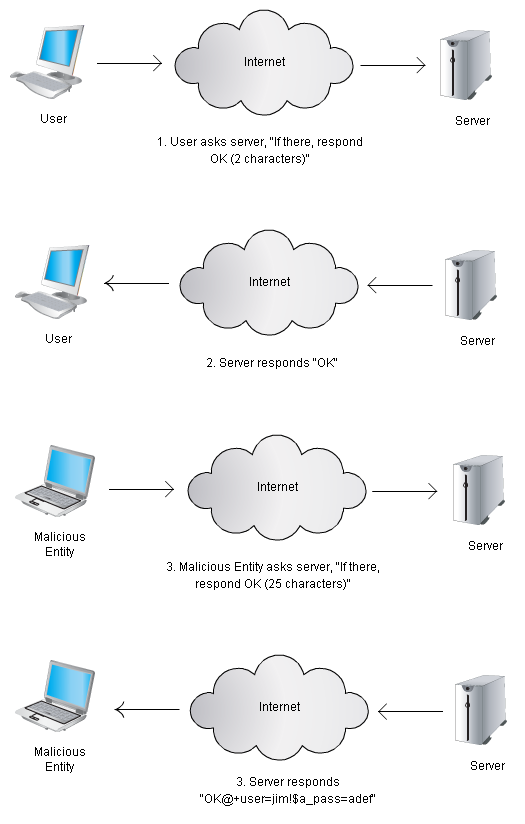


Figure 2: The Heartbleed Bug

It took an incident such as this to highlight some of the deficiencies in writing secure code. OpenSSL has a single full-time employer, with a further ten programmers contributing regularly. Many applications use OpenSSL for security purposes, which mean this is an excellent example of a vulnerability in a third-party library having catastrophic consequences for the projects that use the dependency.

This bug allows a malicious entity to exploit a part of the system. The malicious entity is impersonating a valid user, which is commonly referred to as spoofing. The bug allows for information disclosure, including sensitive information such as passwords and credit card information. It is a prime example of an extremely exploitable security flaw, and touches on many of the ideas that will be discussed during the threat modelling exercise.

# Design

This chapter contains two distinct sections. The first discusses the planning of the analysis. It discusses the approach taken, the tools that were utilised and the open source projects that were examined.

The second section discusses the design of a bepoke analysis tool. This tool was designed to complement the OWASP Dependency Check tool. The Dependency Check tool is designed to be run against a single project. It is not intended for aggregating multiple projects together and performing statistical analysis on the results.

The Dependency Check tool produces an XML file for each project it examines. Considering the amount of projects that were involved in this analysis, examining each of these XML files by hand would have been time-consuming and error-prone. It was decided to construct a utility which could parse large volumes of the XML files produced by the OWASP Dependency Check tool, and perform various types of analysis based on the results.

## Approach to analysis

The general approach taken in analysing the open source projects is summarised below.

1. Scan with the OWASP Dependency Check tool.
2. Parse the XML produced by the OWASP Dependency Check tool.
3. Create various reports based on the parsed XML.
4. Examine the reports, looking for unusual results.
5. Use the reports to guide the examination of projects and vulnerabilities in more detail.

This approach was taken for third party libraries first, followed by the project’s compiled jar files. This approach was refined and improved upon over time, the details of which are documented in Section 4.3 Analysis Implementation.

### Notes on the Open Source projects used

The analysis was performed initially on 112 open source projects, available via the Qualitas Corpus (Tempero et al. 2010). These projects represent a large percentage of the most popular open source tools used in the Java world. Some refinement on what projects were included in the final analysis was required, and has been documented in the chapter 4, “Implementation”.

The Qualitas Corpus is a “curated collection of software systems” (Qualitascorpus.com. 2014). It is intended for use by academic studies, and in particular for static analysis of the open source project contained within. It initially appeared in a paper published by Tempero et al. (2010), but has been updated multiple times since.

There are several different versions of the Corpus. One version includes every single version of every project. A second version only contains the most recent version **that has been catalogued**. This is the version that was used for this analysis.

It is important to make the distinction between the open-source projects and the open-source dependencies. Through this dissertation, open-source projects will be referred to as source jar files or source libraries, whereas dependencies will generally be referred to as third party dependencies.

### Notes on the Analysis tools used

The analysis made use of the OWASP Dependency Check tool. This tool was developed as a counter-measure against the rise in vulnerable third party libraries. It allows developers to scan their projects, building a list of all dependencies and their respective versions as it does so. It then consults the CVE database (cve.mitre.org, 2014) and determines how many of those dependencies contain vulnerabilities. It generates either an XML report or an HTML report with details on what published vulnerabilities it has found.

The OWASP Dependency Check tool is designed to run against a single project. Several scripts and utilities were designed to assist with the analysis process. First, a shell script was created that allowed the tool to be run against multiple projects. A second shell script was then created to gather all the generated report files into a single location, labelling them properly. Thirdly, a java utility was created that would take the XML output by the dependency check tool and perform various additional types of analysis on the data. This approach allowed for much more in-depth analysis and comparison of the information that is output by the OWASP Dependency Check tool alone. The process of refinement of the scripts and tools is discussed in Section 4.

One additional point-of-interest in relation to the dependency check tool is that it checks an online database for the latest reported vulnerabilities. This ensures that the analysis is as up-to-date as possible. One side effect of this is that running the same analysis twice could potentially give different results depending on which day it was run. This was kept in mind as the tool was used in the subsequent analysis.

## Dependency Check Analyser – Java Utility

### Requirements

The following requirements were initially identified.

* The ability to parse multiple XML files that are output from the OWASP Dependency Check tool
* Using the parsed data, perform various types of analysis, such as vulnerability distribution and identifying unique vulnerabilities.
* Output the data in a format that is easy to read, such as a CSV file that can further be analysed in Microsoft Excel.

These high-level functional requirements were complimented with the following non-functional requirements.

* The analysis should not take an unreasonable amount of time to run. It was decided that the full analysis of all projects, for both source and third party libraries should take no more than an hour.
* Easily add new calculations and types of analysis as they are identified, without making major structural changes to the code.
* The project should be written in Java, purely to coincide with the author’s experience at writing applications in Java.

### Architecture

After identifying the requirements, a number of architecture layers were envisioned.

|  |  |  |  |
| --- | --- | --- | --- |
| Input Layer | Controller | Factory | UI |
| Domain Layer |
| Analysis Layer |
| Presentation Layer |

Figure 3 : Application Architecture

The Input Layer would be responsible for parsing the data that is output from the OWASP Dependency Check tool. It should be abstract enough to handle not only XML files but potentially other types of file in the future.

The Domain Layer should define the objects that the data is parsed into. These objects are then used consistently through the rest of the system. They are essentially an object representation of the XML data.

The Analysis Layer will contain multiple analysis classes. These will all have a common interface. This layer should be easily extendable to allow for additional types of analysis in the future.

The Presentation Layer should be an abstract layer that is concerned with presenting the results from the analysis to the user. It should initially output the data to a CSV file, but it should allow for future adjustment, either to a different type of file, or even as input to another application.

The Controller Layer is the layer that will dictate the control flow of the application. It calls into the interfaces at each layer, and is responsible for passing messages between each layer.

The Factory Layer will contain utility classes that simplify the construction process of the various implementation classes that are used. It hides the complexity of the application from the user.

The UI Layer will deal with user interaction. Initially this will simply be a case of calling the command line and passing in a path to the location of the reports generated by the Dependency Check tool. The UI layer is completely abstracted from the implementation layer underneath, so it would be easy to implement a much nicer GUI for example, or expose the tool as a web service.

### UML Diagrams

Class diagrams representing the class structure were initially designed then adjusted as the code evolved and was refactored. The final class diagrams can be seen in Figure 3 and Figure 4.



Figure 4: Analysis Controller Class Diagram

Figure 3 describes the relationships in the AnalysisController and its related hierarchy. The AbstractAnalysisController contains the three key instance variables. The interface AnalysisParser represents the ‘Input Layer’ in the architecture table. The AnalysisPresenter interface diplays the data in a human-readable format and corresponds to the ‘Presentation Layer’ in the architecture table. The final member variable is a List of AnalysisStrategy objects. Using a List allows us to add new AnaylsisStrategy objects dynamically later. The performAnalysis() method defined in the AbstractAnalysisController class then iterates through each AnalysisStrategy object and performs its specific analysis, which is implemented in the DefaultAnalysisController.

Currently, the project contains a single implementation of the AnalysisParser, the DirectoryAnalysisParser. This implementation works at a directory level. The constructor takes a single argument called ‘path’. This will be the full directory path that contains all of the XML reports. When the parse() method is invoked, it goes to this directory, runs through every file in the directory, then uses JAXB to unmarshall the XML file into a Java object in memory. An exception will be thrown if it hit any other type of file. The parse() method returns a List of Analysis objects, one for every XML file in the directory.

The AnalysisPresenter interface is implemented by the CsvFileAnalysisPresenter. This implementation has a number of responsibilities. Firstly, it creates the output directory and file. In order to avoid files being overridden, and to try and help identify output files after the fact, a time-stamped directory structure was used. This class also includes protected methods for writing the headings and the data. It was decided to make these protected, to allow subclasses to over-ride them if necessary. For example, one current limitation on the output is that we can only produce a CSV file with two columns. However, we could easily overwrite both the writeHeadings() method and the writeData() method to write lots of fields out to the CSV file.

Finally, the presentAnalysis() method inherited from the interface is implemented. This simply takes an AnalysisResult object, creates the file, writes the headings, and finally extracts whatever data that is needed from the AnalysisResult object to fill out the data in the CSV file. In this way, the same AnalysisResult object can be used to produce many different types of output reports.

A subclass was later added that extended the CsvFileAnalysisPresenter. In this instance, the data is not written to a new file each time. Rather, the data is appended to the same file as before. This allows for comparison of the results from multiple runs of the Analysis tool. This was used for a short while to ensure consistent results in the output. This class was eventually not used, but it has been kept in place to give an indication of how the project could be extended in the future.

The class diagram in Figure 3 mentions the interface for the AnalysisStrategy, which is defined in more detail in Figure 4.



Figure 5: Analysis Strategy Class Diagram

The AnalysisStrategy interface defines two methods. The first, performAnalysis(List<Analysis>) is implemented in the AbstractAnalysisStrategy. The first thing this method does is call into the abstract method initialise(). By making this method abstract, it forces implementers to include it in the concrete Strategy object at the end of the hierarchy chain. The initialise object is intended to setup some of the details of the analysis, such as the name of the analysis, and the headings that will eventually end up in the CSV file.

Next, the performAnalysis() method iterates through each of the Analysis objects that it has been provided with, and calls into the second abstract method, runSpecificAnalysis(Analysis). This method then contains the individual calculations required for the given analysis.

The implementation of these classes evolved as the code evolved. Initially the performAnalysis() method was implemented in each concrete strategy. But after implementing several of these, it became clear that the exact same steps were being run in each of them, resulting in a lot of duplicated code. Also, the initialisation of the class was being called from within the loop, which meant it was needlessly be run hundreds of times.

The solution was to refactor the code, extracting the common method up into the abstract class, and moving the code that changes into abstract methods of their own, allowing each subclass to implement them as required. The result was less duplicated code and a much cleaner implementation.

### Design considerations

The Java utility was designed using several key design patterns.

* The various types of analysis are implemented using the Strategy design pattern.
* Both the Factory pattern and Builder pattern were used for constructing objects. This could also have been achieved using dependency injection, but that was considered outside of scope for this project.
* The entire application essentially use the MVC pattern, with the View being implemented as the output layer, the Controller implemented in the Controller layer, and the rest of the logic and domain objects making up the Model.
* A Façade pattern was considered early in the design phase, but was superseded by the simpler Strategy pattern.

The entire application was written with the SOLID principles in mind, namely:

* Single-responsibility: Every interface and class has a single, identifiable responsibility. If a class started to contain too much functionality, it was refactored out into a separate class.
* Open-closed principle: The entire application is designed to be closed for modification but open for extension. From the very early stages, the Analysis classes were intended to be very easy to add to.
* Liskov substitution: This principle states that “objects in a program should be replaceable with instances of their subtypes without altering the correctness of the program”. This means coding to the interface, and making sure that the code is not tied to any specific implementation. This is true for each of the layers, which are easily replaceable without affecting any of the other layers.
* Interface Segregation Principle: this states that it is better to have many interfaces than having a single general-purpose interface. Basically, this states that subclasses should not have to implement methods that it does care about. In this application’s design, great care was taken to ensure that the interface only specified methods that were directly relevant to that interface’s goal, and that each method was required.
* Dependency Inversion Principle: This refers to the idea that concrete implementation should not call out to other concrete implementation. It is another wording of the “don’t call us, we’ll call you” idea. In this project, dependency injection is implemented using Factory classes, but these could easily be replaced with Spring or a similar dependency injection framework.

### Analysis Strategies

The intention of the “Total Number of Vulnerabilities” strategy was to count how many individual vulnerabilities were associated with each dependency.



Figure 6: XML Tree Strcture

Figure 5 shows a tree representation of the XML produced by the OWASP Dependency Check tool. It is clear that this particular dependency has seven associated vulnerabilities. This particular strategy should be able to run across an entire XML file and give a count of vulnerabilities for all dependencies.

Sometimes the same vulnerability can be reported multiple times for a given project. This can happen when multiple, independent dependencies have a common vulnerability in their hierarchy. Figure 6 highlights the situation.



Figure 7: Duplicate Dependencies

In this example, a project named Compiere has a dependency on both scheduler-plugin.jar and el-api.jar. Both of these jar files have a dependency on a JBoss library which contains the vulnerability CVE-2007-1157. If we were to use the “Total Number of Vulnerabilities Strategy”, this would count as two distinct and separate vulnerabilities.

It was decided that it would be useful to remove any duplicated vulnerabilities such as this. With that in mind, the “Total Number of Unique Vulnerabilities” strategy was designed to behave much the same as the “Total Number of Vulnerabilities” strategy. The only difference was that it would maintain a map of all vulnerabilities that it had found so far, and if it found vulnerability it had already encountered, then it would not increment the count.

The “Total Number of Libraries” strategy was similar in scope to the “Total Number of Vulnerabilities” strategy. It was considered useful to determine how many libraries were actually being referenced for each project. This would allow for some further deduction, such as the number of vulnerabilities per library.

The “Total Number of Unique Vulnerabilities per Library” strategy built on top of the strategies that had been implemented before it. This calculation would be relatively straight-forward; by simply calculating the number of unique vulnerabilities and then divide by the number of libraries.

Another type of metric that was considered useful was the total number of libraries that contained a vulnerability of any kind. This type of analysis is useful when we don’t want to focus on particular vulnerabilities, but instead want to discuss just libraries that contain vulnerabilities of any kind. This was implemented in the “Total Number of Vulnerable Libraries” strategy.

Incidentally, when the OWASP Dependency Check tool is set to output in HTML format, this is the main metric that is used. This was one reason for some confusion in the initial analysis phase (discussed in **Section X.Y**).

# Implementation

This chapter is broken into two broad sections. The first discusses the implementation of the code used to analyse the XML files produced by the OWASP Dependency Check Tool. The second discusses the approach used in how the analysis itself was actually performed and refined over time.

## Tools and Components Used

The following section discusses the various tools and utilities that were used to help facilitate this implementation. It is interesting to note that even a small application that is used to analyse XML files requires many third party tools and utilities.

### Dependency Check Tool

**TODO: discuss…**

### Microsoft Threat Modelling Tool 2014

Microsoft has created their own implementation of a Security Development Lifecycle, or SDL. This process provides advice and prescriptive steps to follow in order to ensure best security practices when writing code. One of the more popular tools is the Threat Modelling Tool. This tool allow for the creation of Data Flow Diagrams, or DFDs. These DFDs allow developers to visualize their applications, typically before any code has been implemented. However the tool can still provide business value when applied in a retrospective fashion.

The tool allows developers to model their application, focusing on interactions between components (both internal and external), as well as boundaries between the various modules. The tool provides assistance and advice in creating a valid workflow between the components. Once the developer is satisfied with the diagram, the tool will generate a report identifying potential areas of concern. Note that this tool is primarily used for identifying threats that may exist. It is up to the security professional or the developer doing the analysis to determine if vulnerabilities actually exist or not.

Using the Threat Modelling Tool alongside the Dependency Check Tool provides very powerful analytical opportunities, which have been highlighted in this dissertation.

## Project Implementation

**TODO: write about the code here!**

## Analysis Implementation

The analysis was performed in distinct phases, with the results from each phase directing the flow of subsequent analysis. The work-flow is represented in the timeline in Figure 8 below.

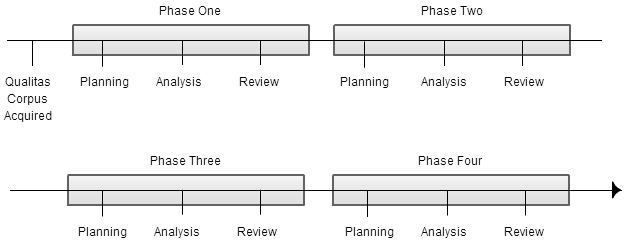


Figure 8: Analysis Timeline

### Phase One Analysis Implementation

#### Planning

Initially, a number of projects were analysed from the command line using the dependency check tool. After the generation of several dependency reports by hand, it was decided to construct a script that could help automate these steps. The manual process at this stage consisted of:

1. Identify all third party jar files.
2. Copy them into a new directory.
3. Analyse these jar files to create a vulnerability report for third party jar files.
4. Identify all source jar files.
5. Copy these files into a new directory.
6. Analyse these jar files to create a vulnerability report for the source jar files.

Note that the jar files were copied into new directories in order to keep the original structure of the Qualitas Corpus intact.

#### Analysis

In order to simplify the process and make it much more efficient, a script was written to automate these steps. This particular script was constructed in two stages.

The first stage automated steps 3 and 6. The script was run and a project name was passed in. The script then constructed the required directory structure (SystemName/thirdParty, SystemName/Source, etc). It then asked the user to copy all jar files into the pre-created directories. Once the user clicked on continue, the script would carry on and automatically analyse both the source jar files and the third party jar files. This was relatively straight-forward, as the only variable in this instance was the name of the project itself.

#### Review

However this approach still required a considerable number of manual steps. Stage two was concerned with distinguishing between any third party libraries and any source jar files. Upon examination it was discovered that the majority of the projects had source jar files that included the name of the system itself in its bundled jar file. It was decided to copy all jar files from the project into the thirdParty directory, and then move only the jar files that contained the project name into the source directory. It was acknowledged at this point that this was not a full-proof strategy but at least a stepping stone to a better solution.

The approach worked relatively well as a feasibility test, and allowed the author to generate the initial analysis, which can be seen in Section 5.1**.**

### Phase Two Analysis Implementation

#### Planning

Several problems were identified with the initial approach to the analysis. Firstly, the results were being output in a HTML form. In order to use these results by a down-stream application, it became necessary to output the data in more structured and malleable format. Luckily the tool provided an option to produce the results in XML. From Phase Two on, all down-stream analysis was based on the XML output by the dependency check tool.

However the data that was output by the XML tool did not match up precisely with the data that was output when the tool was set to HTML. This was troubling, but it was decided to continue with the XML output for further analysis but eventually reconcile and document the differences. It was expected that the tool would be able to produce consistent results regardless of the output method. It became clear on later examination the reason for the inconsistencies. When set to produce output in HTML format, the tool provides a count of the number of vulnerable libraries in a project. This can often be a very different measure to the number of individual, unique vulnerabilities in a project.

#### Analysis

XML was subsequently used as a building block for further exploratory analysis. The analysis was run consistently, every day for a week to ensure that the results were consistent when the output was set to XML. This proved to be the case.

The next step was to parse the XML data from these files into memory, where they could be analysed by a bespoke tool allowing for the creation of more complex reports.

It was decided to use JAXB (Java API for XML Binding) to import the XML files into Java objects. In order to do this, an XSD schema file was required. Unfortunately, the OWASP Dependency Check tool did not provide a standard XSD file for their XML files. Instead, the website “freeformatter” was used[[1]](#footnote-1). This allowed the author to paste in raw XML into the textbox, and then generate an XSD file for that XML. The largest XML file available to the author was selected for this purpose. It was reasoned that the largest XML file would probably contain most of the elements that need to be defined as part of the schema. Once complete, xmllint was used to validate the remaining XML files. If they failed, the schema was updated accordingly. For example, some items were set to mandatory because they had appeared in the initial XML file, but later inspection of alternative XML files revealed these items to be optional.

This process was repeated until all XML files passed with the new schema. This schema was then used by JAXB to generate the required ‘model’ classes. Once these domain classes were in place, the next step was to implement the unmarshalling code, which was done in the DirectoryAnalysisParser class, described **in the code implementation section above (TODO: add!).** The schema was kept with the project, and used as the definition of all data objects. If a future analysis failed as it was being unmarshalled from XML into an object, it would indicate that the schema was not quite finished and required further adjusting.

Once all of the accompanying boilerplate code was put in place, it became possible to implement some strategies for analysing the code. The results were collected and timestamped accordingly, with the results documented in **section x.y.**

#### Review

TODO: What were the problems with Phase Two?

### Phase Three Analysis Implementation

#### Planning

Over time, each project was examined in more detail. As each project was manually inspected, it became clear some refinements to the process were required. It became evident that that analysis should only be performed on active projects. The Qualitas Corpus maintains metadata on each of the projects included. This includes information on the status of each project. Only projects marked as active were included in the analysis. This reduced the total number of projects being analysed down to 85.

Secondly, it was determined that a number of the remaining 85 projects were dependent on third-party libraries that were not bundled along with the project within the Qualitas Corpus. It was deemed unfair to analyse these projects, since a full analysis of the project with its dependencies was impossible to obtain. It would result in unfair comparisons between projects. This reduced the set of projects to a final number of 76, which are documented in Appendix A.

Upon reflection, it also became evident that the method for determining whether a Jar file was a third-party dependency or a compiled source jar file was inadequate. Since all projects built their jar files differently and produced them in various types of output directory, there was no way of writing a simple script to copy third-party jar file into one location with the source jar files in another. In order to maintain the integrity of the analysis, each individual jar file had to be examined to determine if it was a third party dependency or a compiled jar file.

This process required the examination of the source directories inside the Jar file themselves. From this, it became possible to tell if the jar file originated from the project itself or if it was a ‘foreign’ third-party dependency.

The end result of this refinement produced some striking differences in the results from phase two (**TODO: add in some comparison about this in the results section).**

#### Analysis

TODO

#### Review

TODO

### Phase Four Analysis Implementation

Once the analysis of all projects was complete from both a third-party and source library perspective, the decision was made to examine the vulnerabilities from a threat-modelling viewpoint. The projects which were identified as the biggest offenders were targeted, and a typical threat-modelling process was applied. This would help identify the threat level of each of the vulnerabilities that had been highlighted up to this point. It had already been established that viewing vulnerabilities from a simplified viewpoint can result in misguided assumptions and conclusions. It was decided that a project should be examined using a well-established threat-modelling pattern, in order to quantify the real threat of all of these vulnerabilities that had been highlighted.

Threat modelling is a process that is employed to ascertain how secure an application is. It typically involves identifying threats and vulnerabilities, and subsequently identifying the risk level associated with these threats and vulnerabilities. It is important to understand and use the correct terminology when discussing Threat Modelling. Defining these terms will help frame this analysis. McGrath (2013) defines the terms thusly:

* A **threat** is an event that can damage an asset by performing an attack
* A **vulnerability** is a flaw in a systems security that allows a threat to take place.
* An **attack** is an attempt by an adversary to exploit a vulnerability.
* A **risk** is the likelihood of an attack.

With these definitions in mind, it becomes clear that identifying vulnerabilities is only part of the process. In this phase of analysis, it was hoped that a clearer perspective of a system’s security would emerge.

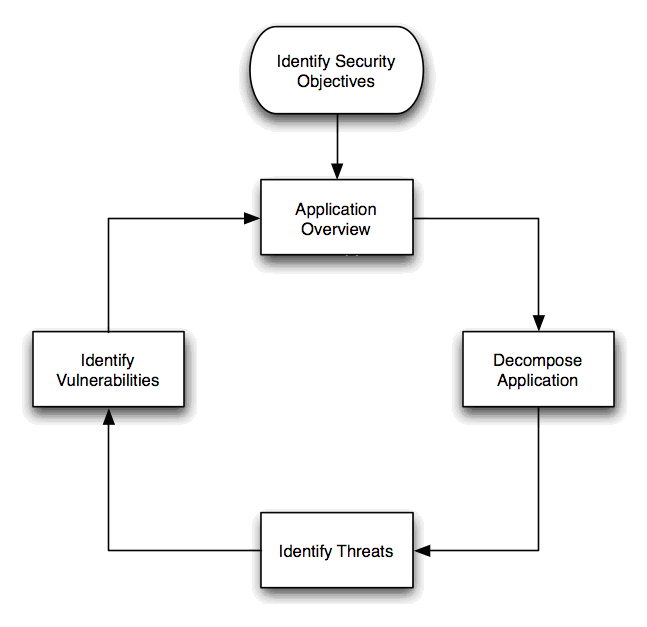


Figure 9: OWASP Threat Modelling Process (Owasp.org, 2014)

Figure 7 shows the typical threat modelling process, as recommended by OWASP. This will guide the steps required to perform a threat analysis of several of the open source projects.

The steps were identified as follows:

1. Identify project that we wish to examine
2. Identify security objectives. This would need to be examined in the context of the application that is being examined.
3. Create the Application Overview. This is used to identify components, data flows and trust boundaries. In particular, we are interested in possible entry points for attackers to exploit. This may be difficult to do, since the architecture of the project in question may not be fully understood.
4. Decompose the application. This involves identifying the specific components within the application that effect security. This is required to allow for explicit definition of potentially unsecure components. Typically these components will be found where business layers interact, or where the system is retrieving or sending data to and from external sources.
5. Identify the threats. This process generally revolves around identifying common threats and determining if they are applicable to a particular component in the system under analysis. STRIDE and TRIKE are two common approaches to threat identification.
6. Identify vulnerabilities. Once the vulnerabilities have been identified, an assessment is performed to identify the risk score. DREAD is one common approach to risk assessment.

In relation to the analysis already performed, step 6 has already been performed. This highlights where the process of identifying vulnerabilities appears in a secure development lifecycle. The analysis performed in phase four will help frame the analysis performed up to this point.

#### Identify Projects

Threat modelling was initially performed on Struts. The reason for this choice was due to the fact that a relatively large number of vulnerabilities had been identified for this application, increasing the likelihood of being able to match up threats to their associated vulnerabilities.

#### Identify Security Objectives

One of the main reasons to perform this step is to identify what is important to a given organisation or project. In other words, identify the information and assets that need to be secure. To help with this analysis, OWASP have defined five broad security objectives:

* Identity.
* Financial.
* Reputation.
* Privacy and regulatory.
* Availability guarantees.

Struts is a popular web application framework. It provides authentication for web sites, immediately making it vulnerable to identity issues. Due to the fact that Struts can be applied and used in a very wide domain, it is very likely that it could be used in a setting that involves monetary transactions. A website that uses Struts is directly tied to the security of Struts; if a bug is uncovered within the Struts framework that effects the website that uses Struts, it risks hurting its own reputation because of it. Again, due to the fact that websites are in the public domain, they are very much tied to privacy and regulatory issues. Finally, it is vital for a website to remain available, so a vulnerability found within Struts that compromises availability is a major concern.

Even performing this simple action of identifying security objectives helps to clarify how insufficient it is to measure security purely on the existence of vulnerabilities.

#### Application Overview and Decomposition

At this stage of a typical threat modelling analysis, the application is typically examined from an architectural point-of-view. This is typically performed by the developers of the application. In this instance, with the absence of the original developers, an approximate architecture was examined, combined with investigating sections of the source code itself to allow for an approximate decomposition of the application. The Microsoft SDL tool was used to examine the various threats that could exist from an abstract, design point-of-view.

The first step was to identify the architecture of the struts framework, which can be seen below in Figure 8 (Das, 2014).

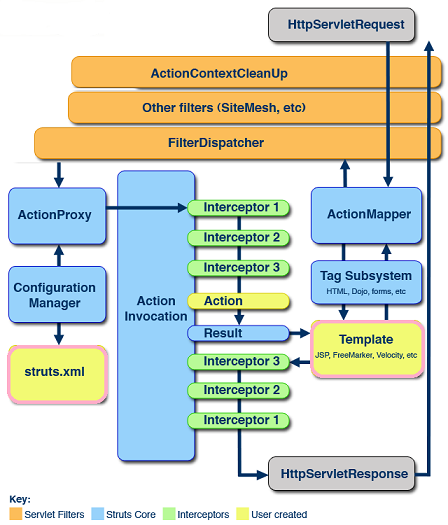


Figure 10: The Struts Architecture

In a typical threat modelling situation, an application would be examined by its developers who would have the knowledge required to identify the components within a system. Boundaries are a particular concern, and would be examined in detail. This type of analysis is usually performed using data flow diagrams, which help identify boundaries and data stores.

In relation to this particular analysis, comparing the results of threat modelling with the already-identified vulnerabilities will hopefully help identify holes in the security analysis.

#### Threat Identification

Struts was analysed using the Microsoft Threat Modelling Tool. This was performed with the simplest use case in mind, that of a browser accessing the Struts Web Server. The interaction is shown in Figure 9 below.

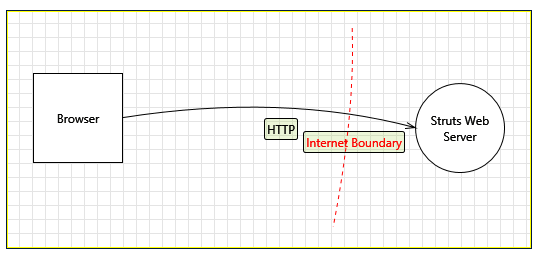


Figure 11: Struts Data Flow Diagram

This simplified representation of the Struts application generated a total of eleven potential threats, which are identified in table 1 below. If Struts were examined in depth, it is likely that many more threats would be identified.

|  |  |  |
| --- | --- | --- |
| **Threat** | **Category** | **Description** |
| Spoofing the Browser External Entity | Spoofing | Could lead to information disclosure |
| Cross Site Scripting | Tampering | Could be subjest to CSS if it does not sanitize the input data |
| Elevation Using Impersonation | Elevation of Privilege | May result in incorrect privileges being applied |
| Spoofing the Web Server Process | Spoofing | Could lead to information disclosure |
| Potential Lack of Input Validation for Web Server | Tampering | Could lead to DoS or elevation of privilege attack |
| Potential Data Repudiation by Web Server | Repudiation | The web server could claim that it did not receive data from a source outside the trust boundary |
| Data Flow Sniffing | Information Disclosure | Data flowing across HTTP may be sniffed by an attacker. |
| Potential Process Crash or Stop for Web Server | Tampering | Could result in a DoS attack |
| Data Flow HTTP Is Potentially Interrupted | Tampering | Could result in a DoS attack |
| Web Server May be Subject to Elevation of Privilege Using Remote Code Execution | Elevation of Privilege | Browser may be able to remotely execute code for the Struts Web Server |
| Elevation by Changing the Execution Flow in Web Server | Elevation of Privilege | An attacker may pass data into the Struts Web Server in order to change the flow of program execution |

Table 1: Struts Threat Identification

#### Vulnerability Identification

The next step in a typical threat modelling exercise is to identify the vulnerabilities. This is generally accomplished by examining all the identified threats, and subsequently trying to identify specific instances of vulnerabilities that exist in the system that could lead to the exploitation of one of the pre-identified threats.

In this analysis, a total of thirty three vulnerabilities had already been identified for Struts. These were categorised based on the threats above. The results are highlighted in Table 2.

|  |  |
| --- | --- |
| **Threat** | **Identified Vulnerabilities** |
| Spoofing the Browser External Entity | None |
| Cross Site Scripting | CVE-2001-1772 CVE-2011-2087 CVE-2012-1007 |
| Elevation Using Impersonation | CVE-2011-5057 CVE-2011-0392 CVE-2012-0393 CVE-2012-4387 CVE-2013-0248 CVE-2013-4152 CVE-2013-4310 CVE-2013-6429 CVE-2013-7315 |
| Spoofing the Web Server Process | None |
| Potential Lack of Input Validation for Web Server | CVE-2008-6504 CVE-2012-0391 CVE-2012-0838 CVE-2013-2248 CVE-2013-2251 |
| Potential Data Repudiation by Web Server | None |
| Data Flow Sniffing | CVE-2011-2088 |
| Potential Process Crash or Stop for Web Server | CVE-2006-6916 CVE-2013-6429 CVE-2007-0185 CVE-2013-4152 CVE-2013-7315 |
| Data Flow HTTP Is Potentially Interrupted | None |
| Web Server May be Subject to Elevation of Privilege Using Remote Code Execution | CVE-2013-4310 CVE-2007-0184 |
| Elevation by Changing the Execution Flow in Web Server | CVE-2012-0392 CVE-2010-1622 CVE-2012-0394 CVE-2013-1965 CVE-2013-1966 CVE-2013-2115 CVE-2013-2134 CVE-2013-2135 |

Table 2: Mapping threats to vulnerabilities

Several of the threats that were identified during the threat modelling phase, currently do not have any accompanying vulnerabilities. In a real-life situation, this may drive further analysis of the application around those areas, since the vulnerabilities might exist but have not yet been identified and reported. At this stage of the threat modelling analysis, the threats and vulnerabilities would be ranked, to determine which threats are the most dangerous for this particular project. This would ensure that preventative efforts are made in the highest priority areas.

Threat modelling is not a process that is run once. It is a constantly evolving lifecycle. Once the highest priority threats have been mitigated, the process can start again, with further refinements being made as the developers start to understand the scope and capabilities of the system, as well as the types of vulnerabilities that can exist. A second reason for running periodic threat modelling sessions is the emergence of newly reported vulnerabilities and bugs.

# Testing and Results

The following table examines each of the analysis phases…

|  |  |
| --- | --- |
| **Analysis Phase** | **Conclusion** |
| Phase 1 | Tool was set to HTML output. This meant it only provided the number of vulnerable JAR files per project. This was identified as a misleading metric to examine vulnerable software. |
| Phase 2 | Options were adjusted in the tool allowing much more useful data presented in XML format. However the volume of XML files was too much for manual analysis, leading to the creation of the bespoke Java utility. Results were good, however misleading due to several issues (discussed below) |
| Phase 3 | Phase 2 analysis focused on some defunct projects, some projects that were contained in the Qualitas Corpus but did not include their dependencies, and finally the process to identify if a jar was third-party or source was refined and validated. This lead to results that were much more correct. |
| Phase 4 | Threat Modelling analysis was performed on a single/several project(s). This provided insight into how threat modelling is useful in identifying potential threats, even before the identification of specific vulnerabilities. It also pointed at areas where vulnerabilities may exist, but have not been found yet. |

## Phase One Results

The approach to the analysis performed during phase one is discussed in **Section 4.3.1**. This analysis provided an early indication as to the health of each individual project and can be thought of as a feasibility study. The results that came out of phase one greatly influenced the direction that subsequent analysis took.

Phase one analysis used the OWASP Dependency Check Tool ‘out-of-the-box’. When the tool is run in this fashion, it generates a single HTML page per project that it is analysing. An example appears in Diagram 7 below.

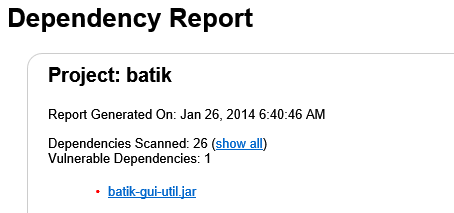


Figure 12: Example output from Dependency Report

The results gathered in this phase gave an indication as to how many vulnerable dependencies existed for each project. However it does not provide any information on how many individual vulnerabilities are found in each library. This type of analysis simply provided an indication of where problems may occur. As discussed in section X.Y, this may not be the best metric to use to identify insecure libraries.

Table 3 below represents the project that contained the highest number of vulnerable dependencies.

|  |  |
| --- | --- |
| **System** | **Vulnerabilities** |
| compiere | 141 |
| wct | 19 |
| jag | 15 |
| netbeans | 14 |
| exoportal | 12 |
| hibernate | 10 |
| gt2 | 8 |
| hadoop | 7 |
| heritrix | 6 |
| roller | 6 |

Table 3: Number of Vulnerable Third Party Dependencies per Project

After generating this initial table, it was decided to examine the outlier in much more detail, to help determine why this particular application would contain so many vulnerable dependencies. This investigation proved extremely useful, as it highlighted the short-comings of measuring security with only a single metric.

Upon examination, it was discovered that although Compiere did contain 141 vulnerable libraries, this figure was misleading. Of the 141 vulnerable libraries, there were only 97 actual vulnerabilities being referenced. So for example, one vulnerable library might reference the same vulnerability several times, because more than one of its own third party libraries might reference it. However, this was also misleading, because across multiple libraries, the same vulnerability could be referenced. So library X and library Y could both refer to the same vulnerability, thus resulting in a misleading total count. Once this was taken into consideration, the total number of unique vulnerabilities that were affecting Compiere became 17. Figure 12 highlights the problem.

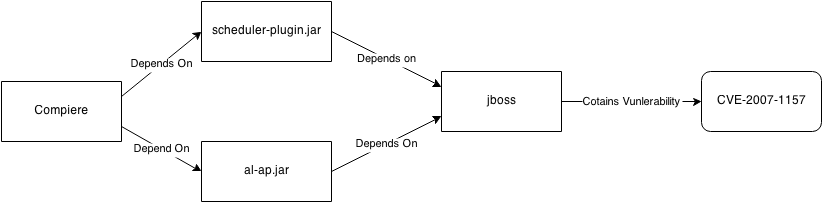


Figure 13: Duplicate Vulnerabilities distorting Phase One analysis

This immediately highlighted the problems in using a single metric to find vulnerabilities. The OWASP Dependency Check tool was examined to see if it provided options for more robust information. It was discovered that it did, in the form of XML documents as opposed to HTML documents. This provided the added benefit of well-formed data, making it easier to parse and manipulate by another application, which is exactly what was done in Phase Two.

Phase one also included analysis on the source libraries, the results of which are contained in Table 4 below.

|  |  |
| --- | --- |
| **System** | **Vulnerabilities** |
| JBoss | 26 |
| Netbeans | 11 |
| gt2 | 5 |
| Struts | 4 |
| Ireport | 2 |
| Maven | 2 |
| Batik | 1 |
| Hadoop | 1 |
| Heritrix | 1 |
| nakedobjects | 1 |
| Poi | 1 |
| springframework | 1 |

Table 4: Vulnerable Source Libraries per Project

Again, this analysis suffered from similar problems to the third party dependencies. These two tables were essentially disregarded once Phase One was complete.

The final piece of analysis that was performed during Phase One was to compare the average number of Vulnerable Libraries for third party libraries against the equivalent for source libraries. The results can be seen in Figure 8.

Figure 14: Percentage of Vulnerabilities Comparison

The total number of vulnerable dependencies identified in third party libraries was greater than the total number of vulnerable source libraries by a factor of three to one. This was suggestive of the fact that third party libraries are less secure than their inherent source jar file counterparts. **Figure 10** below provides a bird’s eye view of the data.

Figure 15: Initial analysis

The web container JBoss contains 185 source jar files, with a total of 26 vulnerable source libraries, making it the worst offender for source jar vulnerabilities, with an average of 0.14.

The ERP project Compiere referenced 339 third party libraries, and the initial analysis for that project alone identified 141 vulnerable dependencies, making it the worst offender for third party vulnerable dependencies, with an average of 0.42.

The initial analysis indicated that the project’s source libraries contained less vulnerabilities than their referenced third party counterparts, as identified by Figure 8. The total number of source jar files scanned was 2251, with 56 vulnerable jar files discovered. This gave an average of 0.02. On the other hand, the total number of third party jar files that were scanned was 3829, with 299 vulnerable dependencies discovered, providing an average of 0.08.

## Phase Two Results

The initial analysis was inconclusive, due to the fact that it was based on a single metric. Adjustments were made to the dependency check tool, which are discussed in the implementation section 4.3.2. Once the tool was set to output the data in XML format, the results were consistent, as well as much easier to work with.

Figure 16: Unique Third Party Vulnerabilities vs Unique Source Library Vulnerabilities

The line graph in Figure 11 represents the number of unique vulnerabilities found in all Jar files in each project. The Jar files are separated into third-party jar files and source jar files that are built from the source code. It is clear from this diagram that there are three outlier projects that contain more 3rd party vulnerabilities than any other. These were identified as:

|  |  |
| --- | --- |
| **System** | **No. of Vulnerabilities** |
| Netbeans | 125 |
| gt2 | 112 |
| Findbugs | 99 |

Table 5: Three Outlier Projects

In order to give some perspective to the remaining projects, these outliers were removed for individual analysis, and the graph was generated again. It can be seen in **Figure 12**.

Figure 17: Outliers removed

The scale of this diagram is notably smaller, with 60 being the max compared to 140 in the first chart. From examining both these graphs, it immediately becomes clear that there are more vulnerabilities to be found in the third-party libraries than in the source libraries. However, **Figure 8** does indicate a few notable exceptions. Top of this list is Tomcat, with a total of 39 unique vulnerabilities found in the source code for this project, compared to a single vulnerability in the third party dependencies. A second notable exception is struts with 22 exceptions in the source code, compared to 11 in the third party libraries.

The outliers were subsequently examined in more detail below.

### Findbugs

Findbugs contains only 16 third-party libraries, yet had a large number of vulnerabilities identified in those libraries. This explains the high concentration identified in Figure 9 below. Upon further examination, it became clear that of these 16 libraries, only two contain vulnerabilities. The two vulnerable Jar files were identified as:

* mysql-connector-java-5.1.7-bin.jar
* AppleJavaExtensions.jar

Next, each Jar file was examined. It was discovered that the AppleJavaExtensions.jar file contained only a single vulnerability, identified as CVE-2010-0538. This meant that a single jar file, mysql-connector-java-5.1.7-bin.jar, accounted for 98 of the total 99 vulnerabilities identified in the Findbugs application.

Upon further examination, all of these 98 vulnerabilities are tied to early versions of the MySQL product. The exact phrasing is:

“Unspecified vulnerability in the MySQL Server component in Oracle MySQL 5.1.72 and earlier, 5.5.34 and earlier, and 5.6.14 and earlier”

Oracle has released MySQL version 5.7 which, according to the information provided by NIST, would resolve all 98 of these outstanding issues.

### gt2

GeoTools have 112 vulnerabilities and 111 third-party libraries. However, the analysis revealed that all 112 vulnerabilities are found in only 3 jar files:

|  |  |
| --- | --- |
| **Jar File** | **No. of vulnerabilities** |
| mysql-connector-java-5.1.5.jar | 108 |
| batik-util-1.7.jar | 3 |
| gnu-regexp-1.1.4.jar | 1 |

Table 6: GeoTools vulnerable dependencies

In a similar fashion to Findbugs, the vast majority of vulnerabilities were identified in the Oracle mysql connector component.

### Netbeans

Netbeans contained far more third party libraries than any other open source project, with 270 libraries, 89 more than its nearest rival, JBoss. However, off these 270 libraries, only 14 contained vulnerabilities. This is further indicative of the need for full transparency when investigating security issues, and not to simply focus on statistics alone.

The 14 identified vulnerable jar files are listed below in table 7.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name of Jar File** | **No. of vulnerabilities** | **CVE numbers** | **Description** |
| mysql-connector-java-5.1.6-bin.jar | 87 |  |  |
| jetty-6.0.2.jar | 7 |  |  |
| servlet-api-2.5-6.0.2.jar | 7 |  |  |
| spring-core-3.0.2.RELEASE.jar | 6 |  |  |
| webserver.jar | 4 |  |  |
| spring-2.5.6.SEC01.jar | 4 |  |  |
| mysql-connector-java-3.1.12-bin.jar | 3 |  |  |
| svnjavahl-1.6.0.jar | 1 |  |  |
| winp-1.14-patched.jar | 1 |  |  |
| postgresql-8.3-603.jdbc3.jar | 1 |  |  |
| org-openide-util-lookup.jar | 1 |  |  |
| org-openide-util-lookup\_ja.jar | 1 |  |  |
| nexus-indexer-2.0.0-shaded.jar | 1 |  |  |
| svnClientAdapter-1.6.0.jar | 1 |  |  |

Table 7: Netbeans vulnerabilities

Oracle’s MySQL connector is again the most prevalent source of the vulnerabilities. In this instance, we can see the Netbeans has a dependency on 3.1.12 of the connector, as well as 5.1.6. The number of vulnerabilities is much greater in version 5.1.6, by a ratio of 29 to 1. This could indicate that vulnerabilities were added by developers between version 3.1.12 and 5.1.6. However, it could equally be true that the same vulnerabilities do exist in version 3.1.12 but have never been catalogued.

## Phase Three Results

A total of 38 projects had no vulnerabilities in their third party libraries. It’s important to note however that this simply means there are currently no reported vulnerabilities in the CVE database.

### Top Ten Lists

The data was correlated and a number of top ten lists were constructed which are discussed below.

#### Number of Vulnerabilities per Third Party Library

Initially, the analysis revolved around the ratio of unique vulnerabilities that were showing up per library, which can be seen in Table 3. This table contains the ten projects that have the highest number of unique vulnerabilities per third party library.

|  |  |
| --- | --- |
| **Project name** | **Number of unique vulnerabilities per library** |
| findbugs | 6.1875 |
| mvnforum | 1.125 |
| heritrix | 1.1154 |
| gt2 | 1.0090 |
| roller | 0.5082 |
| marauroa | 0.5 |
| netbeans | 0.4630 |
| tapestry | 0.3421 |
| jgrapht | 0.3333 |
| tomcat | 0.3333 |

Table 8: Number of Vulnerabilities per 3rd Party Library

It became immediately obvious that Findbugs contained a much larger ratio of vulnerabilities in its third party libraries than any other project, beating the second place item by a factor of six to one.

The second interesting point is that only the top 4 projects have a ratio above 1. This is indicative of the fact that most third party libraries that are used by open source projects do not contain any reported security vulnerabilities.

The third interesting conclusion is in relation to the types of applications that are showing up in the top ten list. The Qualitas Corpus contains a categorisation of each project which has been included in the chart above. However it is worth examining two of these projects in more detail since understanding the context of a project may be just as important as examining the raw figures provided by tools such as the dependency check tool.

Findbugs is a tool that allows for static analysis of source code. Generally it will be run in some type of development environment. The likelihood of it being used by a malicious entity on the web is low. Therefore, the threat level for this project would be considered quite low when typical threat modelling techniques are applied.

mvnforum is an open-source bulletin board. This type of application could run in several types of settings; it may be used internally by a company to record employee’s ideas or it may be run publicly to allow customer to discuss the company’s products. The context that mvnforum could be used in may make it a bigger threat than findbugs, regardless of the fact that findbugs has a much higher ratio of vulnerabilities. This means that the threat level for this project would be much higher than that of Findbugs.

Scanning the types of application that appear in the top ten list, it becomes apparent that several of these vulnerabilities may not be exploitable in an enterprise setting. gt2 is a library that provides tools for generating geo-spatial graphs. Marauroa is a game. Netbeans will typically run in a development environment and jgrapht is a utility for generating graphs.

Next, the equivalent project’s source libraries were examined, to allow for comparison, the results of which can be seen in Figure 14.

Table 9: The top ten projects with the highest ratio of vulnerabilities compared to their respective source libraries

This graph highlights that all of the projects in this particular top ten list have more vulnerabilities in their third-party libraries than in their respective source libraries, with the exception of Tomcat. Again, this is indicative of the fact that there are more vulnerabilities in 3rd party dependencies.

#### Number of Vulnerabilities per Source Library

The jar files that were generated from the project’s source code were analysed next, the results of which can be viewed in Table 5.

|  |  |  |
| --- | --- | --- |
| **Project name** | **Number of unique vulnerabilities per library** | **Type** |
| struts | 4.4 | Middleware |
| tomcat | 3 | Middleware |
| hadoop | 1 | Middleware |
| rssowl | 1 | Tool |
| heritrix | 0.6667 | Tool |
| jruby | 0.6 | Programming Language |
| ireport | 0.4444 | Diagram generator/data visualization |
| springframework | 0.1923 | Middleware |
| quartz | 0.1667 | Middleware |
| argouml | 0.1481 | Diagram generator/data visualization |

Table 10: Number of Vulnerabilities per Source Library

This analysis indicates that more vulnerabilities are found in third party libraries than the equivalent source libraries. Findbugs, with the most vulnerabilities per third-party library, presented 6.2 bugs per jar file. Compare this to Struts which had the most number of vulnerabilities per source jar file, which sits at just under 4.5 vulnerabilities per library. Again, it should be noted the context of each of these applications. Struts is a web application framework, and maybe used in a live enterprise setting.

Figure 18: Number of Vulnerabilities per Source Library compared to their respective Third Party Libraries.

Figure 16 plots the number of vulnerabilities in a project’s source library jar files against the number of vulnerabilities found in that project’s respective third party libraries.

Tomcat and Heritrix appear on both lists, in contrasting contexts. We can clearly see that Tomcat has much more vulnerabilities buried in its source libraries, whereas Heritrix contains more vulnerabilities in its third party dependencies.

#### Number of Vulnerable Third Party Libraries

A secondary metric that was examined was the total number of unique vulnerable libraries per project. The results of analysing each projects third party libraries are showing in **Table 6.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Project name** | **Total Number of Libraries** | **Total Number of Vulnerable Libraries** | **Type** |
| compiere | 175 | 85 | tool |
| jboss | 181 | 29 | Middleware |
| wct | 95 | 19 | tool |
| netbeans | 270 | 14 | IDE |
| roller | 61 | 6 | Tool |
| ireport | 88 | 5 | Diagram generator/data visualization |
| mvnforum | 48 | 5 | Tool |
| struts | 40 | 5 | Middleware |
| hibernate | 17 | 5 | database |
| hadoop | 49 | 4 | Middleware |

Table 11: Projects with the highest number of unique vulnerable third party libraries

When the OWASP Dependency Check tool is set to output its reports in HTML format, this is the primary metric that is produced. Despite its appearance as an important metric, it may be quite an ineffectual measurement of security in a project. For example, although Compiere reports 85 vulnerable libraries out its total 175 libraries, there are only 17 unique vulnerabilities.

When compared with the “Number of Vulnerabilities per Third Party Library” top ten list, it was noted that roller is the only project that appears on both lists. Again, this points towards the issues that can occur when focusing on one particular statistic over another. Statistics help to build a picture, but judgement needs to be made to determine the difference between a genuine problem and a statistical anomaly.

Examining the types of projects that show up on this list, six out of the ten consist of tools and IDE’s, both of which are unlikely to be found in a live enterprise setting. JBoss, Hibernate, Struts and Hadoop are much more likely to appear in such a setting.

We should also examine the total number of libraries that each project reference. For example, in JBoss, 16% of all libraries that are used contain security vulnerabilities. Hibernate on the other hand contains flaws in 29% of all of its libraries. Again, all of this information can only be used as an indicator of the extent of the vulnerabilities in each project.

#### Number of Vulnerable Source Libraries

The same analysis was performed against the source jar files, the results of which are documented in Table 7.

|  |  |  |  |
| --- | --- | --- | --- |
| **Project name** | **Total Number of Libraries** | **Total Number of Vulnerable Libraries** | **Type** |
| eclipse\_SDK | 342 | 123 | IED |
| jboss | 52 | 38 | Middleware |
| netbeans | 1539 | 11 | IDE |
| tomcat | 13 | 6 | Middleware |
| struts | 5 | 5 | Middleware |
| gt2 | 76 | 5 | SDK |
| jruby | 5 | 4 | Programming Language |
| maven | 47 | 3 | parser / generators / make |
| hadoop | 3 | 2 | Middleware |
| ireport | 9 | 2 | Diagram generator/data visualization |

Table 12: Projects with the highest number of vulnerable source libraries

This list was compared to the top ten “Number of Vulnerabilities per Source Library”. The Eclipse IDE was identified as having the highest number of vulnerabilities, and yet did not even appear on the list highlighting the ratio of vulnerabilities to libraries. Netbeans is the third highest, with 11 vulnerable source libraries, and yet the percentage of vulnerable source jar files is a staggeringly low 0.7%.

Once again, the types of projects that are showing up on this list are largely tools and utilities as opposed to components that might be used in live enterprise settings. JBoss, Tomcat and Hadoop are the big exceptions in that sense.

#### Top Ten Vulnerabilities identified in third party libraries

This analysis only really made sense to run against the third party libraries. Analysis of the source libraries made no sense, since each project would only return information on vulnerabilities in that specific project only, therefore it did not seem productive to try and gather accumulative data. The results of this analysis can be seen in Table 8 below.

|  |  |  |
| --- | --- | --- |
| **CVE Identifier** | **Vulnerability ID Distribution** | **Vulnerable Dependency** |
| CVE-2013-7315 | 13 | Spring Framework |
| CVE-2013-6429 | 13 | Spring Framework |
| CVE-2011-2730 | 13 | Spring Framework |
| CVE-2013-4152 | 13 | Spring Framework |
| CVE-2012-0213 | 8 | Apache POI |
| CVE-2007-6059 | 8 | JavaMail |
| CVE-2010-1622 | 7 | Spring Framework |
| CVE-2013-0248 | 7 | Apache Commons FileUpload |
| CVE-2007-5615 | 6 | Mortbay Jetty |
| CVE-2011-4461 | 6 | Mortbay Jetty |
| CVE-2009-1524 | 6 | Mortbay Jetty |
| CVE-2009-1523 | 6 | Mortbay Jetty |

Table 13: Vulnerability Distribution

This analysis helped identify what the common vulnerabilities were that showed up across multiple projects’ third-party dependencies.

The first four vulnerabilities in the list are all identified in the Spring Framework library. Initially, it seemed unusual that there would 13 occurrences of four distinct vulnerabilities, leading the author to believe that they may in fact all refer to the same vulnerability. These appeared to be the case on further examination. However, they are reported as distinct vulnerabilities because they affect different versions. So for example, if the issue appeared in Spring 3.2.4 and Spring 3.2.5 it would be addressed as two individual vulnerabilities.

## Phase Four Results

**TODO**

# Conclusions

The conclusions should be based on the research work presented in the previous chapters. No new work should be presented here. The author’s analyses and conclusions should be clearly presented. The conclusions should link back to the hypothesis and aims of the research work.

## Initial Analysis

The initial analysis seemed to disprove the author’s hypothesis. The overall average number of vulnerabilities in third party dependencies was much higher than the equivalent source jar files. TODO: fill out

## Third Party Library Usage

The analysis was indicative of the third party libraries being more secure than the Jar files that were generated from the source code. This result seems to disprove the original hypothesis. TODO: fill out

## Analysis of open source projects

The original hypothesis was built upon the author’s own experience in the world of enterprise development. However the analysis was performed on open source projects. These types of systems benefit from the “many eyes” approach to development. It would be interesting for future research to be carried out on enterprise version of software; however the lack of availability of enterprise source code would make this type of research quite difficult.

Despite the findings in this dissertation, the author is still of the opinion that many enterprise applications are written with much less emphasis placed on security than many of the open sourced libraries that they are built on.

## Security Tools

The OWASP dependency check tool appeared to contain some notable flaws. **TODO: put in issues with HTML / XML, discrepancies in results, etc.**

Secondly, it was apparent that a tool such as the dependency check tool does need some technical expertise to get the most out of it. For example, FindBugs proved to be the project with the highest ratio of third party vulnerabilities, sitting at 5.5 vulnerabilities per library. On initial inspection this may seem much worse than Tomcat, which had another reasonably high score of 2 vulnerabilities per library. However the context that each application is used in became much more important in identify the threat level of these vulnerabilities. FindBugs is a static analysis tool, which is likely to be run on a development machine to examine the quality of source code. It does not store persistent data, nor does it run in a production-style environment. Tomcat on the other hand is often deployed as a web container in production settings. This provides 24/7 serving of web pages, and therefore the vulnerabilities found in Tomcat may be a bigger threat to a development team than the vulnerabilities identified in something like FindBugs.

## Vulnerability Identification System

The classification of all vulnerabilities identified can be found at <http://cve.mitre.org/> and <http://nvd.nist.gov/>. An example of a published vulnerability from CVE is shown in **Figure 7**.



Figure 19: A vulnerability in the CVE database

NIST provides supplements this with additional information. The NIST reference to the same vulnerability can be seen in **Figure 8**.



Figure 20: The same vulnerability in the NIST database

NIST includes additional information, such as details on the severity of the vulnerability (using a CVSS score). This is useful information in helping to determine the threat level of each of these vulnerabilities and some basic indication of how complex it is to exploit this vulnerability. However, the key information is still stored in a text field, namely in the “Overview” or “Description” section. There are a number of issues with using raw text. Firstly, it makes it difficult to classify the vulnerability. Does it fit with one of the existing OWASP Top Ten issues, and if it does it would be useful to have that classification included in a separate field. The reason this would be so useful is because the OWASP Top Ten document contains good advice on how to handle each type of vulnerability. With the current classification system, there is still a bit of manual analysis required to determine what type of vulnerability we are dealing with before being able to apply the preventative advice in the OWASP guide.

Another common problem with this type of analysis is the identification of false positives. For example, if jar files are constructed without some key elements, then this may prevent the identification of the risk in that Jar file. This again demonstrates how security analysis requires an amount of security awareness to help determine when these false positives occur.

## Zero vulnerabilities may not mean zero vulnerabilities

Several projects were analysed that returned zero security vulnerabilities in its respective libraries. However, this is similar to the idea of testing a product until there are no more bugs in it. Just because a tester has not found all the bugs, does that mean there are no more bugs? In other words, the existence of vulnerabilities proves that there are vulnerabilities, but the non-existence of vulnerabilities does not necessarily mean there are no vulnerabilities. Rather, it indicates that there are currently no reported vulnerabilities to date.

# References

Avramescu, G., Bucicoiu, M., Rosner, D. and Tapus, N. 2013. Guidelines for Discovering and Improving Application Security. pp. 560--565.

Cve.mitre.org. 2014. CVE -Common Vulnerabilities and Exposures (CVE). [online] Available at: http://cve.mitre.org/ [Accessed: 4 Feb 2014].

Das, G. 2014. *Struts2 Architecture Flow Diagram - GopalDas.Org*. [online] Available at: http://gopaldas.org/struts/struts2/struts2-architecture-flow-diagram/ [Accessed: 14 Apr 2014].

Eshete, B., Villafiorita, A. and Weldemariam, K. 2011. Early Detection of Security Misconfiguration Vulnerabilities in Web Applications. pp. 169--174.

Heartbleed Bug: What is it, W., 2014. *Heartbleed Bug: What is it, Who is handling our security?*. [online] Inferse.com. Available at: <http://www.inferse.com/14435/heartbleed-bug-handling-security/> [Accessed 21 Apr. 2014].

Huluka, D. and Popov, O. 2012. Root cause analysis of session management and broken authentication vulnerabilities. pp. 82--86.

Jackson Higgins, Kelly, 2008. Hacker's Choice: Top Six Database Attacks. [online] Available at <http://www.darkreading.com/security/application-security/211201064/hacker-s-choice-top-six-database-attacks.html>

Jovanovic, N., Kirda, E. and Kruegel, C. 2006. Preventing cross site request forgery attacks. pp. 1--10.

A. Kerckhoffs "La cryptographie militaire", J. Sci. Milit., vol. IX, pp.5 -38 and 161&ndash;191 1883 and Feb. 1883

Li, J., Gupta, A., Arvid, J., Conradi, R. and Others. 2007. The empirical studies on quality benefits of reusing software components. 2 pp. 399--402.

Mcgrath, M. 2013. Threat Modelling for Legacy Enterprise Applications.

Mohagheghi, P. and Conradi, R. 2007. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empirical Software Engineering*, 12 (5), pp. 471--516.

OWASP, 2013. "Top ten most critical web application security vulnerabilities". Available: <https://www.owasp.org/index.php/Top_10_2013>

Owasp.org. 2014. *File:Threat Model Flow.gif - OWASP*. [online] Available at: https://www.owasp.org/index.php/File:Threat\_Model\_Flow.gif [Accessed: 12 Apr 2014].

Pope, C. 2013. Warning over fake Revenue scam messages. The Irish Times, 18th Nov.

Qualitascorpus.com. 2014. *Qualitas Corpus*. [online] Available at: http://qualitascorpus.com/ [Accessed: 18 Mar 2014].

Raemaekers, S., Van Deursen, A. and Visser, J. 2011. Exploring risks in the usage of third-party libraries.*Software Improvement Group, Tech. Rep*.

Stuttard, D. and Pinto, M. 2013. The web application hacker's handbook. Hoboken, N.J.: Wiley.

Sullivan, B. and Liu, V. 2012. Web application security. New York: McGraw-Hill.

Sun, Y. and He, D. 2012. Model Checking for the Defense against Cross-Site Scripting Attacks. pp. 2161--2164.

Takamatsu, Y.; Kosuga, Y.; Kono, K., "Automated detection of session management vulnerabilities in web applications," Privacy, Security and Trust (PST), 2012 Tenth Annual International Conference on , vol., no., pp.112,119, 16-18 July 2012 doi: 10.1109/PST.2012.6297927

Tempero, Ewan; Anslow, Craig; Dietrich, Jens; Han, Ted; Li, Jing; Lumpe, Markus; Meltod, Hayden; Noble, James, 'Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies' 2010 Asia Pacific Software Engineering Conference (APSEC2010), pp336–345, December 2010

Williams, J. and Dabirsiaghi, A. 2012. *The Unfortunate Reality of Insecure Libraries*. [pdf] https://www.aspectsecurity.com/uploads/downloads/2012/03/Aspect-Security-The-Unfortunate-Reality-of-Insecure-Libraries.pdf [Accessed: 8 Jan 2014].

Yu, H., Jones, N., Bullock, G. and Yuan, X. Y. 2011. Teaching secure software engineering: Writing secure code. pp. 1--5.

# Appendices

## Appendix A - Projects used from the Qualitas Corpus

|  |  |
| --- | --- |
| **System** | **Description** |
| Ant | A Java library and command-line tool for supporting processes described in build files as targets and extension points dependent upon each other. |
| Antlr | A parser-generator framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages |
| Aoi | 3D modelling and rendering studio. |
| Argouml | UML modeling tool |
| Aspect | aspect-oriented extension to Java |
| Azureus | A P2P file sharing client using the bittorrent protocol |
| c\_jdbc | Database cluster middleware that allows any Java application to transparently access a cluster of distributed and replicated databases through JDBC(tm) |
| Castor | A data binding framework that provides Java-to-XML binding, Java-to-SQL persistence, and more. |
| Cayenne | A persistence framework providing object-relational mapping (ORM) and remoting services |
| Checkstyle | A development tool to check that code adheres to a coding standard. |
| Cobertura | Calculates the percentage of code accessed by tests |
| Collections | Types that extend and augment the Java Collections Framework |
| Compiere | ERP and CRM business system |
| Derby | A relational database |
| Drjava | Lightweight development environment for Java |
| eclipse\_SDK | An open extensible development platform |
| Emma | Toolkit for measuring and reporting Java code coverage |
| Findbugs | Providesstatic analysis to look for bugs in Java code. |
| Fitlibraryforfitnesse | Provides general-purpose fixtures (and runners) for storytests with Fit and FitNesse |
| Freecol | Turn-based strategy game |
| Freecs | Chatserver |
| Freemind | Mind-mapping tool |
| Galleon | Media server for the TiVo(R) DVR |
| Ganttproject | Tool for project scheduling and management |
| gt2 | Library that provides tools for geospatial data |
| Hadoop | A set of utilities including FileSystem, RPC, and serialization libraries. |
| Heritrix | Extensible, web-scale, archival-quality web crawler project |
| Hibernate | Projects allowing utilisation of POJO-style domain models |
| Hsqldb | SQL relational database engine |
| Htmlunit | API that allows invocation of HTML pages, fill out forms, click links, etc. |
| Informa | News aggregation library |
| Ireport | Visual reporting tool for JasperReports |
| James | Library for mail communication |
| Jasperreports | Reporting engine |
| Javacc | Parser generator |
| Jboss | Application Server |
| Jchempaint | Library for bio- and cheminformatics and computational chemistry |
| Jedit | Development environment |
| Jena | Framework for Semantic Web applications |
| jFin\_DateMath | Libraryfor financial date arithmetic |
| Jfreechart | Chart library |
| Jgrapht | Graph library for graph-theory objects and algorithms |
| Jgroups | Toolkit for reliable multicast communication |
| Jhotdraw | Two-dimensional graphics framework |
| Jmeter | Application for measuring performance |
| Jpf | Runtime engine that dynamically discovers and loads plug-ins |
| Jre | Java Runtime Environment |
| Jruby | Ruby interpreter written in Java |
| Jspwiki | Extensible WikiWiki engine built using J2EE components (Java, servlets, JSP) |
| Jstock | JStock is a free stock market software system, providing Stock watchlist, Intraday stock price snapshot, Stock indicator editor, Stock indicator scanner and Portfolio management. |
| Jtopen | Library supporting the client/server and internet programming models to a system running IBM i |
| Jung | An extendible language for the modeling, analysis, and visualization of data that can be represented as a graph or network |
| Junit | Unit testing framework |
| Marauroa | Multiplayer online game engine |
| Maven | Software project management and comprehension tool |
| Megamek | Online version of BattleTech board game |
| Mvnforum | Bulletin board (forum) |
| Nekohtml | HTML scanner and tag balancer |
| Netbeans | Integrated Development Environment |
| Picocontainer | Inversion of Control (IoC) container |
| Pmd | Java source issue detector |
| Proguard | Java class file shrinker, optimizer, obfuscator, and preverifier |
| Quartz | Job scheduling service |
| Roller | Multi-user and group-blog server |
| Rssowl | News feed reader |
| Sablecc | Parser generator |
| Springframework | Lightweight container, providing centralized, automated configuration and wiring of application objects |
| squirrel\_sql | Graphical program that will allow viewing of the structure of a JDBC compliant database |
| Struts | Framework for creating Java web applications |
| Tapestry | Framework for creating dynamic, robust, highly scalable web applications |
| Tomcat | Implementation of the Java Servlet and JavaServer Pages technologies |
| Velocity | Templating engine. |
| Wct | The Web Curator Tool (WCT) is an open-source workflow management application for selective web archiving. |
| Webmail | Web mail server |
| Weka | A collection of machine learning algorithms for data mining tasks |
| Xerces | Supports creation and maintenance of XML parsers |

1. <http://www.freeformatter.com/xsd-generator.html> [↑](#footnote-ref-1)