

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 case study on the common impediments for failure to update 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 3](#__RefHeading__3866_1545008337)

[Table of Contents 4](#__RefHeading__3868_1545008337)

[Table of Figures 5](#__RefHeading__3870_1545008337)

[Table of Tables 6](#__RefHeading__3872_1545008337)

[Code Listings 7](#__RefHeading__3874_1545008337)

[Abbreviations 8](#__RefHeading__3876_1545008337)

[Glossary of Terms 10](#__RefHeading__3878_1545008337)

[1Introduction 10](#__RefHeading__3880_1545008337)

[1.1Background and Objective 10](#__RefHeading__3882_1545008337)

[1.2Problem Statement 10](#__RefHeading__3884_1545008337)

[1.3Aims 2](#__RefHeading__3886_1545008337)

[1.4Hypothesis 2](#__RefHeading__3888_1545008337)

[1.5Outline of Report 2](#__RefHeading__3890_1545008337)

[2.Survey 3](#__RefHeading__3892_1545008337)

[2.1Third Party Dependencies 3](#__RefHeading__3894_1545008337)

[2.2Using Components with Known Vulnerabilities 3](#__RefHeading__3896_1545008337)

[2.3Cataloguing these Vulnerabilities 4](#__RefHeading__3898_1545008337)

[2.4Available tools 5](#__RefHeading__3900_1545008337)

[2.5Types of Attacks 5](#__RefHeading__3902_1545008337)

[2.5.1SQL Injection 6](#__RefHeading__3904_1545008337)

[2.5.2Broken Authentication and Session Management 7](#__RefHeading__3906_1545008337)

[2.5.3Cross-Site Scripting 7](#__RefHeading__3908_1545008337)

[2.5.4Insecure Direct Object References 8](#__RefHeading__3910_1545008337)

[2.5.5Security Misconfiguration 8](#__RefHeading__3912_1545008337)

[2.5.6Sensitive Data Exposure 9](#__RefHeading__3914_1545008337)

[2.5.7Missing Function Level Access Control 9](#__RefHeading__3916_1545008337)

[2.5.8Cross-site request forgery 9](#__RefHeading__3918_1545008337)

[2.5.9Unvalidated re-directs and forwards 10](#__RefHeading__3920_1545008337)

[3Design 11](#__RefHeading__3922_1545008337)

[3.1Approach to analysis 11](#__RefHeading__3924_1545008337)

[3.2Notes on the Open Source projects used 11](#__RefHeading__3926_1545008337)

[3.3Notes on the Analysis tools used 11](#__RefHeading__3928_1545008337)

[3.4Dependency Check Analyser – Java Utility 12](#__RefHeading__3930_1545008337)

[3.4.1Requirements 12](#__RefHeading__3932_1545008337)

[3.4.2Architecture 12](#__RefHeading__3934_1545008337)

[3.4.3UML Diagrams 13](#__RefHeading__3936_1545008337)

[3.4.4Design considerations 14](#__RefHeading__3938_1545008337)

[3.4.5Analysis Strategies 15](#__RefHeading__3940_1545008337)

[4Implementation 17](#__RefHeading__3942_1545008337)

[4.1Tools and Components Used 17](#__RefHeading__3944_1545008337)

[4.1.1Ubuntu / VMWare 17](#__RefHeading__3946_1545008337)

[4.1.2JAXB 17](#__RefHeading__3948_1545008337)

[4.1.3xmllint 17](#__RefHeading__3950_1545008337)

[4.1.4Agile 17](#__RefHeading__3952_1545008337)

[4.1.5Git 17](#__RefHeading__3954_1545008337)

[4.2Project Implementation 17](#__RefHeading__3956_1545008337)

[4.3Analysis Implementation 17](#__RefHeading__3958_1545008337)

[4.3.1Phase One Analysis Implementation 18](#__RefHeading__3960_1545008337)

[4.3.2Phase Two Analysis Implementation 18](#__RefHeading__3962_1545008337)

[4.3.3Phase Three Analysis Implementation 19](#__RefHeading__3964_1545008337)

[5Testing and Results 21](#__RefHeading__3966_1545008337)

[5.1Phase One Results 21](#__RefHeading__3968_1545008337)

[5.2Phase Two Results 22](#__RefHeading__3970_1545008337)

[5.2.1Findbugs 23](#__RefHeading__3972_1545008337)

[5.2.2gt2 23](#__RefHeading__3974_1545008337)

[5.2.3Netbeans 24](#__RefHeading__3976_1545008337)

[5.3Phase Three Results 25](#__RefHeading__3978_1545008337)

[5.3.1Top Ten Lists 25](#__RefHeading__3980_1545008337)

[6Conclusions 31](#__RefHeading__3982_1545008337)

[6.1Initial Analysis 31](#__RefHeading__3984_1545008337)

[6.2Third Party Library Usage 31](#__RefHeading__3986_1545008337)

[6.3Analysis of open source projects 31](#__RefHeading__3988_1545008337)

[6.4Security Tools 31](#__RefHeading__3990_1545008337)

[6.5Vulnerability Identification System 31](#__RefHeading__3992_1545008337)

[6.6Zero vulnerabilities may not mean zero vulnerabilities 32](#__RefHeading__3994_1545008337)

[7References 33](#__RefHeading__3996_1545008337)

[8Appendices 35](#__RefHeading__3998_1545008337)

[8.1Appendix A - Projects used from the Qualitas Corpus 35](#__RefHeading__4000_1545008337)

# Table of Figures

[Figure 1: Quarterly Statistics 5](#Figure!0|sequence)

[Figure 2: Cross Site Request Forgery 9](#Figure!1|sequence)

[Figure 3: Analysis Controller Class Diagram 13](#Figure!2|sequence)

[Figure 4: Analysis Strategy Class Diagram 14](#Figure!3|sequence)

[Figure 5: XML Tree Strcture 15](#Figure!4|sequence)

[Figure 6: Duplicate Dependencies 16](#Figure!5|sequence)

[Figure 7: Example output from Dependency Report 21](#Figure!6|sequence)

[Figure 8: Duplicate Vulnerabilities distorting Phase One analysis 21](#Figure!7|sequence)

[Figure 9: Percentage of Vulnerabilities Comparison 22](#Figure!8|sequence)

[Figure 10: Initial analysis 22](#Figure!9|sequence)

[Figure 11: Unique Third Party Vulnerabilities vs Unique Source Library Vulnerabilities 22](#Figure!10|sequence)

[Figure 12: Outliers removed 23](#Figure!11|sequence)

[Figure 16: Number of Vulnerabilities per Source Library compared to their respective Third Party Libraries. 27](#Figure!12|sequence)

[Figure 20: A vulnerability in the CVE database 31](#Figure!13|sequence)

[Figure 21: The same vulnerability in the NIST database 32](#Figure!14|sequence)

# Table of Tables

[Table 1: Number of Vulnerable Third Party Dependencies per Project 29](#Table!1|sequence)

[Table 2: Vulnerable Source Libraries per Project 22](#Table!0|sequence)

[Table 3: Number of Vulnerabilities per 3rd Party Library 25](#Table!2|sequence)

[Table 4: The top ten projects with the highest ratio of vulnerabilities compared to their respective source libraries 26](#Table!3|sequence)

[Table 5: Number of Vulnerabilities per Source Library 27](#Table!4|sequence)

[Table 6: Projects with the highest number of unique vulnerable third party libraries 28](#Table!5|sequence)

[Table 7: Projects with the highest number of vulnerable source libraries 29](#Table!6|sequence)

[Table 8: Vulnerability Distribution 30](#Table!7|sequence)

# Code Listings

# 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

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

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 not having to keep re-inventing 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 (**Note to self: reference that last sentence – this has come out of nowhere)**. Kerckhoffs’s principle (1883) roughly states that “a system's security should be provided by the strength of its algorithm, not by keeping the algorithm secret".

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 (**by OWASP; reference!)** 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.

## 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.

## Outline of Report

**TODO: revise as I go along…**

Section 2 will contain a Literary Review of the current state of third-party dependencies.

Section 3 will discuss the approach to the analysis, the tools that were used, and the projects that were examined.

Section 4 will contain details on how the actual analysis was carried out; including scripts and tools that were developed to aid in performing the analysis.

Section 5 contains the results of the analysis.

Section 6 discusses the conclusions based on the results of the analysis. It will examine if the hypothesis agreed with the actual findings.

# Survey

**Note: A lot of this may be cut. Will only keep info on vulnerabilities that are actually found during the analysis of the open source systems.**

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 Checker. 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>

## Types of Attacks

Unfortunately, there is a distinct lack of accurate and reliable information regarding the types of attacks and their frequency. This is due in no small part to the fact that companies rarely publish their vulnerabilities. The following attacks have all been identified by OWASP in its annual Top Ten list.

* SQL Injection
* Broken Authentication and Session Management
* Cross-site scripting (XSS)
* Insecure Direct Object References
* Security Misconfiguration
* Sensitive Data Exposure
* Missing Function Level Access Control
* Cross-Site Request Forgery (XSRF)
* Unvalidated Redirects and Forwards
* Using Components with Known Vulnerabilities

### SQL Injection

This refers to the manipulation of raw String variables within a program, in order to perform operations that are inappropriate on a database. This has been the top threat for the past 3 editions of the OWASP Top Ten Web Application vulnerabilities list, 2007, 2010 and 2013.

There are a number of reasons why this type of attack is so prevalent. Firstly, it represents a relatively easy approach to back-door entrance to an application. Because of its comparative ease, it represents a considerable risk to an enterprise application. It is reasonable to expect that even a semi-skilled hacker would be able to expose an SQL injection attack which would be able to corrupt data, if not actually exporting sensitive data.

The reason these attacks are so relatively easy, is that the SQL might be accessible via a URL. Of course all URL parameters should be encrypted (which we will discuss later), but even if they are not, they should not expose the required information to build up an SQL query. This is true even in the case of a simple 'Select' statement.

A URL may inadvertently contain the connection information to a database, perhaps even including username & password info. This is gold-dust to potential malicious entities on the web.

All code that accesses a database should be secure. Connection strings should not be hard-coded. They should be stored separately, in encrypted configuration files. URL’s should not expose this information.

Using raw SQL in a user interface application or raw URL string can cause major issues. Getting unauthorized information back from a system using a simple SQL string is a real security hole, and can result in massive loss of information – which is the single most important asset to most businesses. A rival could get details on a company’s suppliers or customers. There could also be legal ramifications. It’s fair to say that leaking of information in this way could potentially put a company out of business.

This could be resolved by controlling the SQL language better. A string should not simply be appended to an already-existing SQL string. Java provides the PreparedStatement class for creating more secure SQL queries.

Often simple queries can lend themselves well to Denial-of-Service (DoS) attacks. If a hacker were able to perform even a simple count operation, run it on a high-spec machine in a forever loop, opening new connections for each query, then this would eventually take the DB out of action.

A count operation can help an attacker identify a potential table in a database that may contain large amounts of useful information. For example, a database may use a cryptic table name for customer information, but an attacker may be able to ascertain that a table containing 500 entries may have useful information to exploit.

Another potential security loop-hole is the handling of exceptions. Displaying an SQL Exception to the user may provide information on how the system works behind the front-end.

A fourth approach can be obtaining useful information by inference. A database may contain information on employees. A simple count may reveal the number of employees on each team. A second count may show the total wage for each department. Both of these seem to be fairly innocuous pieces of information; however by combining the information, an intruder would very easily be able to determine an average wage per employee. This type of information could be very useful to a rival company for example, or even members on different teams.

SQL Injections attacks are considered ‘low-tech’, which means it is much more available to more people. If there are much more people using SQL Injection techniques as a malicious tool, then there is much more likelihood that an SQL Injection attack will be successful. In particular with Web Applications, it is “a lot easier to execute a SQL injection attack on a Web application that front-ends a database than on the database itself” (Jackson Higgins, 2008).

Secondly, developers often write poor code to access their database. The problem is that DBA’s manage the actual database but generally speaking it is developers who write the code that access the DBA. This means there is a knowledge gap – the DBA is well-versed in handling and dealing with potential cracks in a database system. The developer is more worried about the functionality of the system. Therefore the point where an enterprise application meets its database is often a security hole, albeit an unintentional one.

A third opinion is that development teams incorrectly rely solely on the protection provided by Web application firewalls. Of course, any security measure is vulnerable, and the best approach is always to have multiple security checks within a system. So this over-reliance on a single security solution, no matter how effective, is immediately questionable.

Of course, the fourth reason is the numbers game. Any developer or database administrator who believes they will be able to secure a system against every possibility is fooling themselves. The best they can do is try to implement multiple security measures, write secure code, avoid hard-coded SQL statements, and apply relevant security firewalls, as well as keeping security software up-to-date. Broken Authentication and Session Management

The OWASP Top Ten describes SQL injection attacks as the single most prominent risk. However it also describes Broken Authentication as more widespread.

### Broken Authentication and Session Management

Session management refers to the ability of a web application to keep track of information pertaining to a single user. For example, a user may log-in to their online banking web application. Security is vitally important in this area; however it would be needlessly cumbersome to require the user to enter their credentials before every operation they perform on the web application. Instead, a user is authenticated once at the start of the 'session'. At a minimum, this involves requesting the username and password from the user, although many respected web sites are now requiring multiple forms of authentication, by verifying users’ emails or mobile phone numbers. Once the user has been authenticated, they will have access to the web application for the lifetime of the session. The session typically ends when the user logs out, or when a defined timeout period has elapsed.

Due to the nature of these sessions, they are often targeted as a weak point in a web application's usage. If a malicious entity were able to get a user’s session ID, they would effectively be able to pose as that user for that length of time. This could obviously present potential identity theft or fraud attacks. Retrieving these session IDs might be as trivial as reading a non-encrypted URL. A further weakness may be in how a developer implements a web application's timeout functionality. Rather than allowing the session to timeout, the session may stay active. This means that if user were to forget to explicitly log out of a site, instead simply shutting down the browser, then the next person to start up the browser will do so with their session information.

OWASP have ranked this particular vulnerability as one of the top three most widespread weaknesses. Takamatsu et al. (2012) have argued that the two main reasons for this are the lack of the necessary skills to create secure web applications, and secondly the act of checking for these vulnerabilities is a "tedious, time-consuming and daunting task". OWASP themselves have suggested that many developers frequently write their own authentication and session management components. However the implementation of adequately secure versions of these tools is not trivial. The recommended approach is to use accepted third-party utilities for authentication and session management. In that instance, we must ensure that the third-party utilities are themselves secure, which can lead to further problems which are the main focus of this paper.

Huluka et al. (2012) suggest part of the problem is in the evolution of web applications themselves. The evolution from static web sites, to dynamic client-server web applications has required some adjustments in how web applications are put together. Design patterns such as the Model View Controller (MVC) provide excellent design, but they do increase the amount of code required to implement a web site. More code generally results in more bugs and vulnerabilities.

Stuttard et al. (2013) describe an easily exploitable example of just such an attack. Suppose an attacker receives a token from a website in the form of an integer. The hacker may be able to simply increment this token number in future communication in order to pose as a different user.

### Cross-Site Scripting

Cross-Site Scripting (sometimes referred to as CSS or XSS) occurs when "an application takes untrusted data and sends it to a web browser without proper validation or escaping". The key issue in this scenario is that the data that is being used is not being checked before it is put to use. Validation should occur at the earliest point-of-entry of the data, to ensure it is in the correct and expected format, and that its content cannot cause malicious intent.

This type of vulnerability has been identified as the single most prevalent form of web application exploit by OWASP (2013). OWASP uses a metric from 1 to 3 in determining the prevalence of a threat, but due to how prevalent XSS is, it has its own rating of 4. XSS allows “attackers to insert client-side script into web-pages viewed by other users” (Avramescu et al. 2013). It is so prevalent that even the large-scale, social network websites are not immune, with Facebook and Twitter both having suffered XSS attacks (Sun et al. 2012).

### Insecure Direct Object References

This type of attack essentially means that a user is given direct access to an object in the system that they should not have any access to. This type of security concern can happen when a system has multiple different roles within a system, each with a varying degree of access control. The accounts manager may be able to access the "Employee Wages"; however other employees probably shouldn't have access to this data.

As another example, suppose an application stores details on users, including their phone number and address. To avoid an SQL injection attack, a competent programmer may write a parameterized SQL statement such as the following:

SELECT Name, PhoneNo, Address FROM User WHERE UserID = ?

The issue could happen in the following scenario. Suppose a user's ID is 1234. They could be presented with a front-end requesting their ID (as a crude example). But what's to stop the user entering 1235. In this way, they may be able to access an object (or row in a table) that they should not have access to.

OWASP argues that many applications simply fail to implement proper verification on users and user roles within a system.

<**TODO:** need to fill this out a bit!>

### Security Misconfiguration

The configuration settings for any security tools that are used in a system are crucial to the overall security of that system. Many applications provide well-known default configuration settings, which could be exploitable if they are not adjusted. For example, many applications use a default administrator username and password pair of "admin" and "admin". Since this information is published and universally-known, this could pose a real threat to the security of a web application

(**Note to self**: this is one of the exploits that I hope to display using Tomcat and its default security settings).

A second consideration is the upgrading of the various tools and utilities that are used by an application. In this instance, we are not just talking about the security utilities, but any third-party tools or libraries that may be used by a web application. This crosses over with ninth vulnerability, usage of third-party software with known vulnerabilities. (**Note to self**: this is obviously an important concept for this paper, so expand this accordingly...)

Eshete et al. (2011) have pointed out that "misconfiguration can happen at any level of an application stack, including the underlying platform, web server, database server, framework, and business logic code". This highlights how dangerous this vulnerability is.

**TODO:** Look again at Eshete's paper - loads of good info in this for this section...

### Sensitive Data Exposure

Simply put, this type of attack occurs when website fail to implement the proper types of security measures to protect sensitive data such as credit card numbers, customer information and username/password combinations. A primitive example of this type of attack might be where a website does not use SSL or any type of encryption when accepting credit card information from a customer. This is a highly simplistic version of this attack, and it would be hoped that this type of attack generally does not exist in the wild.

**TODO:**

Check the OWASP document for more details

### Missing Function Level Access Control

In this type of attack, a request is sent to the application code sitting behind the web front end. Typically, this request is not accessible via the actual GUI of the web site. However the function can be access by constructing a URL to hit the functionality in the background.

The OWASP guide suggests three questions to help decide if an application is vulnerable to this type of attack (**TODO**: reference these properly):

1. Does the UI show navigation to unauthorized functions?
2. Are server side authentication or authorization checks missing?
3. Are server side checks done that solely rely on information provided by the attacker?

### Cross-site request forgery

Sullivan et al. (2012) have described cross-site request forgery as similar to cross-site scripting, however it works in "a completely opposite way". Whereas cross-site scripting refers to the user being 'served' a malicious page or servlet from a browser that it seemingly trusts, cross-site request forgery describes a situation where the web server is fed information that it believes comes from a trusted source; in other words, the user.

This type of attack refers to the situation where a malicious entity attempts to send HTTP requests from the victim's browser to a web page. This type of attack works due to the fact that the web site in questions trusts the user that it is dealing with. Essentially, this attack is a form on temporary identity theft in order to perform some malicious activity on the victim web-site.



Figure 2: Cross Site Request Forgery

In the diagram above, the user sends a request to a banking web site. The user logins in, the bank accepts the user's login, and assumes that any requests coming from this browser are trustworthy. In this type of attack, the malicious entity, or mallet, is able to hi-jack the browser. They are then able to send any URL request to the server, in this case a request that 50,000 euro is deposited into their bank account.

OWASP have identified CSS as much more common-place that CSRF; however Sullivan et al. (2012) claims that it is easier to write a CSRF attack than a CSS attack. The reason for this is that CSRF requires much less scripting knowledge.

How does the attacker gain access to the user’s browser in the first place? This can occur via social engineering; or alternatively by sending on a link to a user to follow which may then invoke sinister code on the trusting web server.

**TODO:**

* Check the OWASP document for more details
* Need to further go through the reference below for more detail on this type of attack.

### Unvalidated re-directs and forwards

This type of attack occurs where a malicious entity sends a user to a web page that they did not intend to visit. The page may be a forgery of an official looking page, much like the recent revenue.ie scam that actually sent a text to a large number of people informing them that they're tax refund was now available and asking them to enter their details at a website that looked quite like the official revenue.ie website. Unfortunately, there were enough people who fell for the scam to make this type of social engineering profitable. [Pope, 2013]

Generally speaking however, this type of attack will not require any knowledgeable involvement by the user. They will click on a website, or a re-direct button, and inadvertently be re-directed to a malicious website. Once the attacker has succeeded in this first step, they can then attack the unassuming user with malware or a cross-site scripting attack.

According to the OWASP guide, this type of attack is relatively uncommon. Also, it should be relatively straight-forward to identify where this type of attack may happen.

# Design

## 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.

## 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.

## 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 unmanageable 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 |

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 3: 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 4: 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

#### Total Number of Vulnerabilities

The intention of this particular strategy was to count how many individual vulnerabilities were associated with each dependency.



Figure 5: 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.

#### Total Number of Unique Vulnerabilities

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 6: 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 decide that it would be useful to remove any duplicated vulnerabilities such as this. With that in mind, the “Total Number of Unique Vulnerabilities” 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.

#### Total Number of Libraries

This 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.

#### Total Number of Unique Vulnerabilities per Library

This 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.

#### Total Number of Vulnerable 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.

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.

### Ubuntu / VMWare

The Qualitas Corpus was designed to be unpacked and viewed in a Linux environment. It contains a number of shell scripts for setting up directories and unpacking tar files. In order to peruse the projects properly, an instance of Ubuntu was installed on a Virtual Machine. This was hosted using the free utility VMWare Player.

### JAXB

Java API for XML Binding allows Java to easily integrate with and use XML files. It provides functionality to convert Java objects into XML, a process known as marshalling, as well as the opposite functionality, unmarshalling XML into Java Objects. It also allows the generation of POJO’s from a pre-defined XSD Schema file.

In this project, JAXB was used to generate the model classes discussed in section x.y. It was also used to parse the XML files output by the OWASP Dependency Check tool, resulting in Java objects that could be examined and manipulated in memory. This was achieved using JAXB’s unmarshalling capability. There was no need to use the marshalling function for this project.

### xmllint

This Linux utility allows a user to validate an XML file against a schema. This was useful in ensuring that the XSD created to validate the XML produced by the OWASP Dependency Check tool was correct. This XSD was required by JAXB to generate the model classes.

### Agile

An agile approach was largely employed during this project. This allowed for prioritisation of tasks, as well as making sure that the overall goal was achievable.

### Git

Git and GitHub was used for source control for this project. This included the source code as well as all associated documentation.

## Project Implementation

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

## Analysis Implementation

### Phase One Analysis Implementation

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.

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.

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.

This approach worked relatively well and helped the author generate the initial analysis, which can be seen in Section 5.1**.**

### Phase Two Analysis Implementation

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.

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 (<http://www.freeformatter.com/xsd-generator.html>). 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.**

### Phase Three Analysis Implementation

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).**

TODO:

Some potentially interesting metrics:

* Average no. of security vulnerabilities found in third party dependencies per project.
* Average no. of security vulnerabilities found in source code per project.
* Comparison of the above two reports
* Most prevalent type of attack
* Comparison of like-for-like projects
* Analysis of the open-source big players; Google, Apache, etc.
* Et cetera

# Testing and Results

## 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. 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.

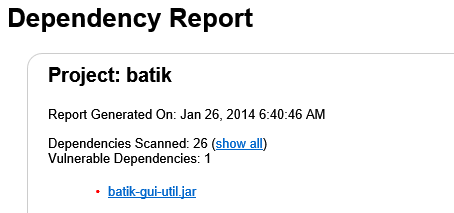


Figure 7: Example output from Dependency Report

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

Table 1: Number of Vulnerable Third Party Dependencies per Project

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

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 8 highlights the problem.

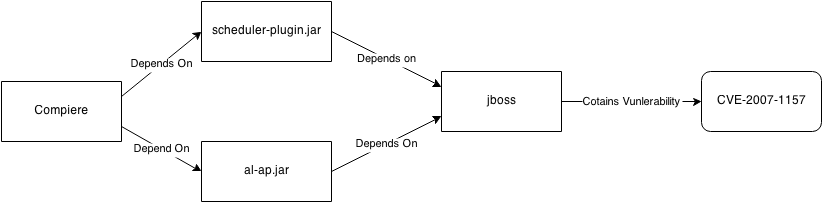


Figure 8: Duplicate Vulnerabilities distorting Phase One analysis

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 2 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 2: 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 9: 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 10: 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 11: 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:

* Findbugs (99 vulnerabilities)
* gt2 (112 vulnerabilities)
* Netbeans (125 vulnerabilities)

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 12: 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:

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

* batik-util-1.7.jar (3 vulnerabilities)
* gnu-regexp-1.1.4.jar (1 vulnerability)
* mysql-connector-java-5.1.5.jar (108 vulnerabilities)

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.

* svnjavahl-1.6.0.jar (1 vulnerability)
* winp-1.14-patched.jar (1 vulnerability)
* spring-core-3.0.2.RELEASE.jar (6 vulnerabilities)
* mysql-connector-java-3.1.12-bin.jar (3 vulnerabilities)
* postgresql-8.3-603.jdbc3.jar (1 vulnerability)
* spring-2.5.6.SEC01.jar (4 vulnerabilities)
* org-openide-util-lookup.jar (1 vulnerability)
* servlet-api-2.5-6.0.2.jar (7 vulnerabilities)
* org-openide-util-lookup\_ja.jar (1 vulnerabilitiy)
* mysql-connector-java-5.1.6-bin.jar (87 vulnerabilities)
* nexus-indexer-2.0.0-shaded.jar (1 vulnerability)
* jetty-6.0.2.jar (7 vulnerabilities)
* webserver.jar (4 vulnerabilites)
* svnClientAdapter-1.6.0.jar (1 vulnerability)

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 | 1.115384615 |
| gt2 | 1.009009009 |
| roller | 0.508196721 |
| marauroa | 0.5 |
| netbeans | 0.462962963 |
| tapestry | 0.342105263 |
| jgrapht | 0.333333333 |
| tomcat | 0.333333333 |

Table 3: 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 4: 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 | 0.666666667 | Tool |
| jruby | 0.6 | Programming Language |
| ireport | 0.444444444 | Diagram generator/data visualization |
| springframework | 0.192307692 | Middleware |
| quartz | 0.166666667 | Middleware |
| argouml | 0.148148148 | Diagram generator/data visualization |

Table 5: 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 13: 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 6: 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 7: 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 8: 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.

# 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 14: 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 15: 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].

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

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.

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

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 |