

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)

August 2013

**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.

# Table of Contents

[Abstract III](#_Toc382506145)

[Table of Contents IV](#_Toc382506146)

[Table of Figures VII](#_Toc382506147)

[Code Listings VIII](#_Toc382506148)

[Abbreviations IX](#_Toc382506149)

[Glossary of Terms X](#_Toc382506150)

[1 Introduction 1](#_Toc382506151)

[1.1 Background and Objective 1](#_Toc382506152)

[1.2 Problem Statement 2](#_Toc382506153)

[1.3 Aims 3](#_Toc382506154)

[1.4 Hypothesis 3](#_Toc382506155)

[1.5 Outline of Report 3](#_Toc382506156)

[2. Survey 5](#_Toc382506157)

[2.1 Types of Attacks 5](#_Toc382506160)

[2.1.1 SQL Injection 5](#_Toc382506161)

[2.1.2 Broken Authentication and Session Management 8](#_Toc382506162)

[2.1.3 Cross-Site Scripting 10](#_Toc382506163)

[2.1.4 Insecure Direct Object References 10](#_Toc382506164)

[2.1.5 Security Misconfiguration 11](#_Toc382506165)

[2.1.6 Sensitive Data Exposure 12](#_Toc382506166)

[2.1.7 Missing Function Level Access Control 12](#_Toc382506167)

[2.1.8 Cross-site request forgery 12](#_Toc382506168)

[2.1.9 Unvalidated re-directs and forwards 14](#_Toc382506169)

[2.2 Using Components with Known Vulnerabilities 14](#_Toc382506170)

[2.3 Cataloguing these Vulnerabilities 16](#_Toc382506171)

[2.4 Available tools 18](#_Toc382506172)

[2.5 Penetration Testing 18](#_Toc382506173)

[3 Design 19](#_Toc382506174)

[3.1 Approach to analysis 19](#_Toc382506175)

[3.2 Notes on the Open Source projects used 19](#_Toc382506176)

[3.3 Notes on Analysis tools used 19](#_Toc382506177)

[3.4 Dependency Check Analyser – Java Utility 20](#_Toc382506178)

[3.4.1 Requirements 20](#_Toc382506179)

[3.4.2 Architecture 21](#_Toc382506180)

[3.4.3 UML Diagrams 22](#_Toc382506181)

[3.4.4 Design considerations 26](#_Toc382506182)

[3.4.5 The Analysis Strategies 27](#_Toc382506183)

[4 Implementation 28](#_Toc382506184)

[4.1 Project Implementation 28](#_Toc382506185)

[4.2 Analysis Implementation 28](#_Toc382506186)

[4.2.1 Phase One Analysis Implementation 28](#_Toc382506187)

[4.2.2 Phase Two Analysis Implementation 29](#_Toc382506188)

[4.2.3 Phase Three Analysis Implementation 30](#_Toc382506189)

[5 Testing and Results 33](#_Toc382506190)

[5.1 Phase One Results 33](#_Toc382506191)

[5.2 Phase Two Results 34](#_Toc382506192)

[5.3 Phase Three Results 36](#_Toc382506193)

[5.4 Top Ten Lists 37](#_Toc382506194)

[5.4.1 Number of Vulnerabilities per Third Party Library 37](#_Toc382506195)

[5.4.2 Number of Vulnerabilities per Source Library 38](#_Toc382506196)

[6 Conclusion 40](#_Toc382506197)

[6.1 Initial Analysis 40](#_Toc382506198)

[6.2 Third Party Library Usage 40](#_Toc382506199)

[6.3 Analysis of open source projects 40](#_Toc382506200)

[6.4 Security Tools 40](#_Toc382506201)

[6.5 Vulnerability Identification System 41](#_Toc382506202)

[6.6 Zero vulnerabilities may not mean zero vulnerabilities 43](#_Toc382506203)

[7 References 44](#_Toc382506204)

[8 Appendices 46](#_Toc382506205)

# Table of Figures

[Figure 1: Cross Site Request Forgery 13](#_Toc382506206)

[Figure 2: Quarterly Statistics 17](#_Toc382506207)

[Figure 3: Analysis Controller Class Diagram **TODO: needs updating!** 23](#_Toc382506208)

[Figure 4: Analysis Strategy Class Diagram **TODO: need updating!** 25](#_Toc382506209)

[Figure 5: Percentage of Vulnerabilities Comparison 33](#_Toc382506210)

[Figure 6: Initial analysis 34](#_Toc382506211)

[Figure 7: Unique Third Party Vulnerabilities vs Unique Source Library Vulnerabilities 35](#_Toc382506212)

[Figure 8: Outliers removed 36](#_Toc382506213)

[Figure 9: Number of Vulnerabilities per 3rd Party Library 37](#_Toc382506214)

[Figure 10: Number of Vulnerabilities per Source Library 38](#_Toc382506215)

[Figure 11: A vulnerability in the CVE database 41](#_Toc382506216)

[Figure 12: The same vulnerability in the NIST database 42](#_Toc382506217)

# Code Listings

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

# Abbreviations

CPE Common Platform Enumeration

CSS Cross-Site Scripting

CSRF Cross-Site Request Forgery

CSV Comma Separated Values

CVE Common Vulnerability and Exposure

CVSS Common Vulnerability Scoring System

CWE Common Weakness Enumeration

DBA Database Administrator

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.

# 



## 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.

* 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 1: 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.

## Using Components with Known Vulnerabilities

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

The 2013 OWASP Top Ten list contained a new vulnerability. This is the use of components with known vulnerabilities, and 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 2: 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>

## Penetration Testing

May remove…

Penetration testing refers to allowing testers to try and attack a system in a controlled environment. This helps to identify holes and potential vulnerabilities in a system. This is a relatively new field of testing, and one that is heavily promoted by OWASP.

# Design

## Approach to analysis

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.

A bespoke analysis tool was then created that would take the XML output by the dependency check tool and perform various additional types of analysis on the data. This allowed for much more in-depth analysis and comparison of the information that is output by the dependency check tool.

## Notes on the Open Source projects used

The analysis was performed 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”.

Analysis was initially performed against all third-party jar files. Once complete, analysis was performed against jar files that were produced by the project itself. The result was the ability to determine the number of vulnerabilities per jar file, allowing a fair comparison between the third party dependencies and the compiled source jar files.

**TODO: Add more detail…**

## Notes on Analysis tools used

The dependency checker 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 to parse all of these files, searching for the various key words before finally constructing a usable CSV file with all the results.

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. (**TODO: discuss WHY each type of analysis was important)**
* 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 suggested

|  |  |  |  |
| --- | --- | --- | --- |
| 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.

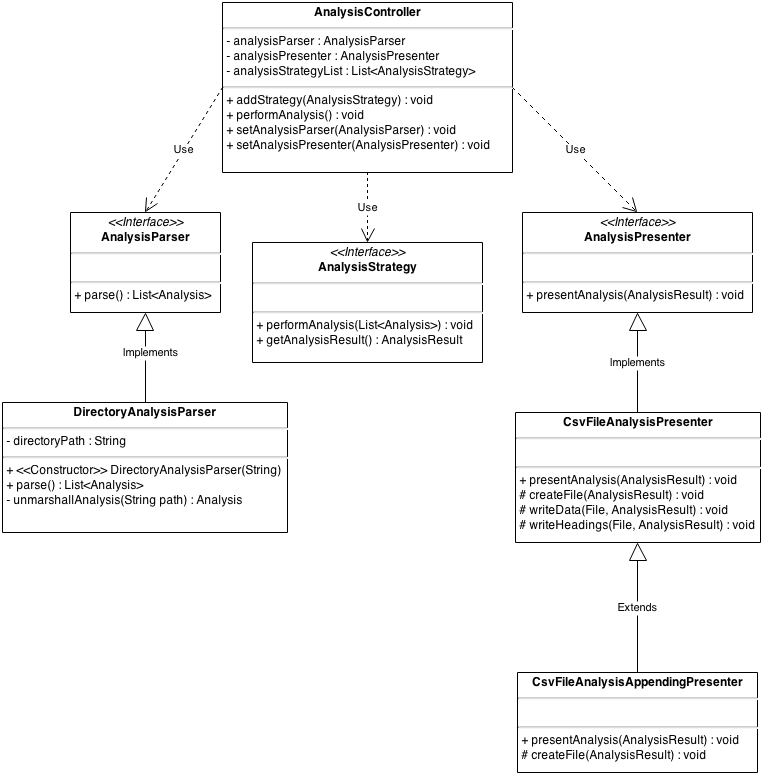


Figure 3: Analysis Controller Class Diagram **TODO: needs updating!**

Figure 3 describes the relationships in the AnalysisController and its related hierarchy. The AnalysisController 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 AnalysisController class then iterates through each AnalysisStrategy object and performs its specific analysis.

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 has two implementations. The first is a simple 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 produced output files after the fact, a time-stamped directory structure was used for the output files. 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 diagram contains the interface for the AnalysisStrategy, which is defined in more detail in Figure 4.

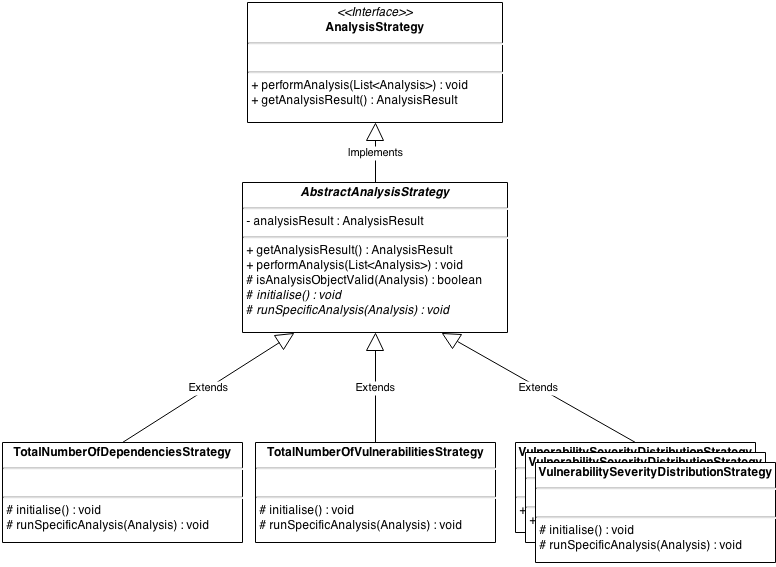


Figure 4: Analysis Strategy Class Diagram **TODO: need updating!**

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 suvstitution: 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.

### The Analysis Strategies

TODO! Discuss each of the different types of analysis. Why these analyses were constructed? What results did I hope to see out of each? How did one analysis strategy lead into the next?

# Implementation

This chapter is broken into two broad sections. The first discusses the

## 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 (**todo: include initial analysis tables and point to them here).**

### 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 77 **(TODO: Add this list of projects as an appendix)**.

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

**ALL OF THIS SECTION WILL BE ADJUSTED!**

The approach to the initial analysis is discussed in **Section 4.1**. This analysis provided an early indication as to the health of each individual project.

The total number of vulnerabilities identified in third party libraries was greater than the total number of vulnerabilities identified in source jar files by a factor of three to one, as highlighted in **Figure 3**. This suggests that the third party libraries are less secure than their inherent source jar file counterparts. TODO: redo these charts with the latest figures, since the initial analysis cannot be trusted!

Figure 5: Percentage of Vulnerabilities Comparison

**Figure 4** provides a bird’s eye view of the data. The full analysis can be found in Appendix A.

Figure 6: Initial analysis

**TODO: The information below might be completely out-of-date now!**

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

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

The initial analysis indicated that the project’s own source jar files contained much less vulnerabilities than their referenced third party counterparts, as identified by Figure 3. The total number of source jar files scanned was 2251, with 56 vulnerabilities discovered. This gave an average of 0.02 vulnerabilities per jar file. On the other hand, the total number of third party jar files that were scanned was 3829, with 299 vulnerabilities discovered, providing an average of 0.08 vulnerabilities per jar file.

## Phase Two Results

**The information above is incorrect!**

The initial analysis provided some inconsistent results **TODO: write about these inconsistencies in the section above**. Adjustments were made to the dependency check tool to ensure consistency. 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 7: Unique Third Party Vulnerabilities vs Unique Source Library Vulnerabilities

The results of the initial analysis can be seen in **Figure 7**. This line graph 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 8**.

Figure 8: 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.

## 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 Figure 9.

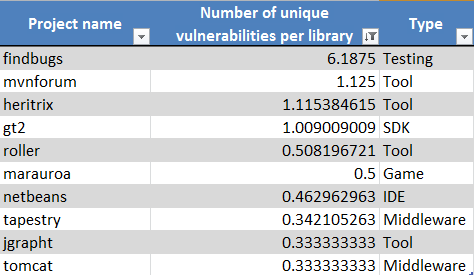


Figure 9: Number of Vulnerabilities per 3rd Party Library

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.

Finbugs is a tool that allows for static analysis of source code. Generally it will be run in some type of development environment.

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.

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.

### 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 Figure 10.

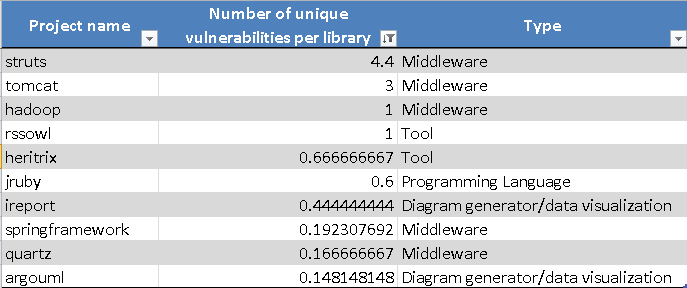


Figure 10: Number of Vulnerabilities per Source Library

This analysis indicates that more vulnerabilities are found in third party libraries than the equivalent source libraries. It also suggests that context is extremely important.

# Conclusion

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 11: 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 12: 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

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.

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

Some Important large diagram 10

The following appendices may be found on the accompanying memory stick.

Appendix B

Class Diagrams

These are held in the CDiagrams folder within the Design folder

Appendix C

Swim Land Diagrams

These are held in the SLDiagrams folder within the Design folder

Appendix D

Tests

These are held in the Tests folder.

Appendix E

Code

These are held in the Code folder. All instructions for set-up and running of the code are held in a README file.