A Browser Based Add-on to Detect Cross Site Scripting Vulnerabilities

Mwotil Alex
Reg. No: 2012/HD05/870U
Stud. No: 208000479
BCS (Mak)
mwotila@gmail.com

A Project Report Submitted for the study leading to a Project
in partial fulfilment of the requirements for the award of
a Master of Science in Computer Science of Makerere University

Supervisor

Engineer Bainomugisha (PhD)
College of Computing and Information Sciences
Makerere University

August 2014

Declaration

I, Mwotil Alex, hereby declare that the work presented is original and has never been submitted for an award to any university or institution of higher learning. I can confirm that where I have done consultations either from published material or the works of others, it has been attributed in this report.

Signature: ..... Date: .....

Approval

This project report has been submitted for examination with my approval:

Engineer Bainomugisha (PhD)
Associate Professor
School of Computing and Informatics Technology;
College of Computing and Information Sciences,
Makerere University
Signature: ..... Date: .....

Supervisor

Dedication

I dedicate this to my parents Mr. and Mrs. Sabila and all my siblings

Acknowledgement

I want to thank the Lord Almighty for the gift of life, strength and knowledge that has helped me achieve this. Surely, Ebenezer: Thus Far The Lord Has Helped Us!. The journey has not been easy but the Lord has made it seamless.

My sincere gratitude to my supervisor Dr. Engineer Bainomugisha for the guidance and encouragement. I sincerely appreciate it.

To the entire College of Computing and Information Sciences where it all began, thank you. Most notably my mentor, colleague, friend and my inspiration Mr. Richard Ssekibuule. The Lord shall surely bless you in abundance.

Last but definitely not the least, I want to thank my family and friends (Wafula Collins, Brenda Chesang, Sam Kwerit, Auma Sharon Proscovia, Sarah Kiden, Perez Matsiko), the University ICT Services (Uganda Christian University) for the support they have rendered me all through my academic years: the festivities they have held in my absence and the words of wisdom they have always imparted to me. To my brother Vincent Chebet, you are more than a blessing to me.

Abstract

Cross Site Scripting (XSS) is a form of code injection attack that exploits input validation flaws in a web application. XSS attacks have overcome buffer overflows as the most common security vulnerability and according to Common Vulnerabilities and Exposures (CVE) contributes to at least 20% of all security vulnerabilities. This is partly due to developers focusing on the logic of the application with less or even no security consideration. This has been compounded by the recent increase in deployment of web applications where XSS attacks are normally exploited. Server side and client side solutions focusing on sanitization of user inputs have been implemented over the past few years. Although the sanitization approach is considered an industry standard, its client implementation is still an open and active area of research. In this project, an XSS detection tool was developed as a Firefox browser extension and tested against a number of other existing vulnerability scanners. It retrieves attack vectors from an online repository and performs penetration tests on the application in context. It scans forms, links and paths on the target page which act as a basis for the penetration tests. Web application developers can utilize this tool to perform tests at the client side and move to correct their applications before deployment. Using four vulnerable applications and three scanners, the tool was able to detect a high number of XSS vulnerabilities in comparison.

Contents

List of Tables

List of Figures

Acronyms

XSS Cross Site Scripting

DOM Document Object Model

CVE Common Vulnerabilities and Exposures

URL Uniform Resource Locator

HTML Hyper Text Markup Language

PHP HyperText PreProcessor

AJAX Asynchronous JavaScript and XML

SSL Secure Sockets Layer

bWAPP Buggy Web Application

XSSD XSS Detection Tool

XPI Cross-Platform Install

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

WVS Web Vulnerability Scanner

LAMP Linux Apache MySQL and PHP

PHP Hypertext Preprocessor

SQL Structured Query Language

QA Quality Assurance

Chapter 1

Introduction

Web-based application development and usage has exploded in the recent few years. Developers write insecure code as a result of rapid development and deployment of these applications. The popularity of the more responsive and interactive Asynchronous Javascript and XML (AJAX) has not helped with these security concerns [15]. Javascript is the current widely used scripting language in enhancing display of web content. Developed by Netscape as a scripting language with object-oriented features, it is downloaded into browsers and executed by an embedded interpreter in the context of the user environment [39].

Javascript programs are executed under two principles of same-origin policy and sandboxing. Sandboxing mechanisms treat Javascript programs as malicious and limit their access to browser resources. The same origin policy states that a program can only execute and use resources associated with the original site. Despite these two security principles, attackers can trick users into downloading malicious Javascript programs from trusted sites [39]. The ability of an attacker to exploit a flaw in a trusted site and lure users to believe in all requests to and responses by the site as taint-free is a classic example of a type of attacks called Cross-Site Scripting.

Cross-Site Scripting (XSS) Attacks have overrun buffer overflows as the world's most common vulnerability [**3**]. There are three types of XSS Attacks: Reflected

(Persistent), Stored (Non Persistent) and DOM based. These variations of XSS attacks make even their mitigation hard as most research has only focused on one or two but not all.

Conventional security practices such as firewalls, Secure Sockets Layer (SSL), intrusion detection systems, network scanners, and passwords [10] that can be applied to traditional applications do not scale well with web-based ones. This is due to the unique features of web development technologies that encompass a number of disciplines and design concepts. The challenges faced in development of web applications and differences in execution modes also account for incompatibility in inferences between these types of applications [15].

As the prevalence of web applications hit the markets in delivering critical services such as control of statewide power grids, operation of hydroelectric dams [10], health, finance, national security and education, more sophisticated and new security loopholes are introduced. A high percentage of web applications interact with back-end databases storing sensitive data such as credit card and health information. A compromise in any of this data can lead to devastating losses in lives, financial, and privacy terms [19].

1.1 Background to the Study

XSS attacks are a form of code injection attacks. These are exploited by attackers through flaws such as improper input validation that exist in trusted web sites. There has been a steady and sharp increase in number of these type of attacks partly attributed to wide adoption of Web applications compared to Desktop applications. There are even sites such as `http://www.xssed.com` that test and display XSS vulnerabilities on public websites [15].

1.1.1 Types of XSS Attacks

According to Howard, LeBlanc, et al. [15], there are mainly three kinds of XSS Attacks:

- Reflected XSS, Nonpersistent XSS, or Type 1

  The injected code is reflected by the web server in form of an error note or search result that may include all the data sent to it in form of a web request as illustrated in Figure 1.1. This type is propagated in emails or links that are part of other web pages.

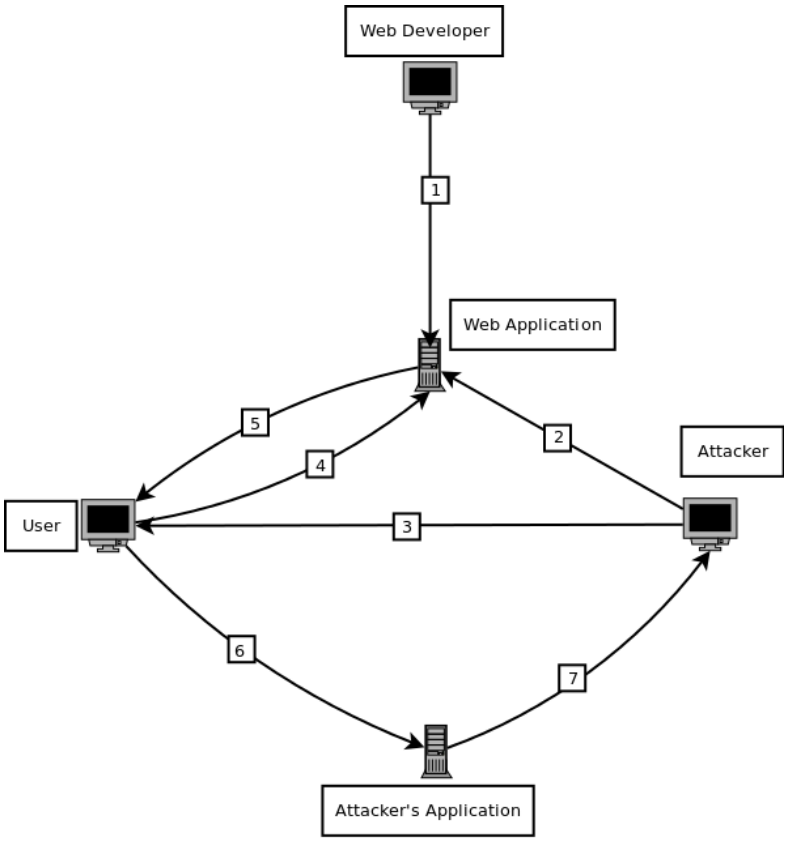
  1. Developer writes a web page with an input validation flaw. The input field can take in an invalidated query string and store it.
  2. An unsuspecting user clicks a link to the developer's web page. The link contains malicious code in the query string.
  3. The website sends the requested page back to the user containing the malicious code.
  4. The malicious code is then executed in the user's browser.

- Stored XSS, Persistent XSS, or Type 2

  This is a variant of Reflected XSS attacks. Instead of echoing back the inputs to the user, the malicious Javascript code is permanently stored on the target server. This is common in message forums, product review sites and blogging sites. This is illustrated in Figure 1.2.

  1. Developer writes a web page with an input validation flaw. It should read the stored data and echo it without validating it.
  2. A malicious user clicks sends malformed input to the server which is then stored.
  3. A user visits the site.

4. The web site sends the requested page back to the user containing the malicious code.
5. The malicious code is then executed in the user's browser.



1 - Web Developer writes a web application with an Input Flaw
2 - An Attacker Discovers the web application vulnerability by use of XSS Vectos
3 - Attacker crafts a link in the context of the trusted application that exploits the web vulnerability and transmits it to an unsuspecting user
4 - The user then follows the crafted link which is transmitted to the vulnerable web application
5 - The Users browser fetches the page from the web application. This contains the attacker's payload
6 - The payload executes in the context of the vulnerable web application and sensitive data is transmitted to the attacker controlled computer
7 - The attacker can then retrieve the data and can do anything with it

Figure 1.1: Reflected XSS

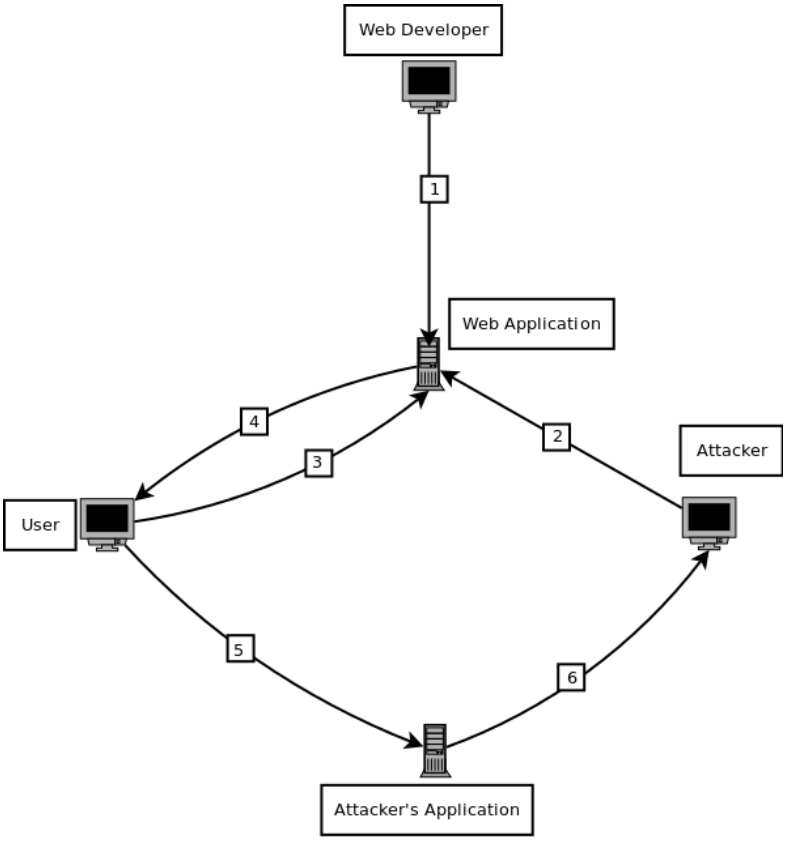


Figure 1.2: Stored XSS

- DOM-Based XSS or Type 0

  This is an XSS vulnerability that exists within client-side pages. URL request parameters accessed may write information to the HTML body or perform DOM-based operations when not validated. The requests may contain additional client-side scripts that can modify the DOM environment [30].

The common entry points of this type of attack is static HTML pages and gadgets (social networking tools, e-mail and instant messaging notification, RSS feeds, sticky notes, system information and weather data [1]). Gadgets can easily be developed by novice users with no special programming background not to mention security skills. These can render untrusted input that may contain malicious Javascript codes that can modify the DOM [15].

1.2 Statement of the Problem

XSS Attacks contribute to at least 20% of all security vulnerabilities according to CVE [**3**]. This has been compounded by the recent wide adoption of web applications. A number of techniques server side, client side or both have been subjects of research over the past years and these focus mainly on sanitization. Although sanitization of user inputs and outputs is considered as an industry standard [39] for securing web applications against Cross Site Scripting attacks, its implementation at the client-side is an open and active area of research.

1.3 Objectives

1.3.1 General Objective

The main objective of this research was to design a client side web browser add-on that employs penetration techniques to detect XSS attacks.

1.3.2 Specific Objectives

1. To review the literature related to XSS attacks and sanitization techniques.
2. To design a browser add-on that applies penetration techniques to detect XSS attacks.
3. To test and validate the add-on using commonly exploited XSS vulnerabilities.

1.4 Scope

This research focused on detection of reflected, stored and DOM-based XSS attacks at the client side. The add-on was limited to browsers developed under Gecko, being a free and open source layout engine. Mozilla Firefox browser was of particular interest given its cross-platform features and public usage.

1.5 Justification

A secure browsing environment to be provided by the development of this client-side solution will aid in the following:

1. Allow users to browse trusted sites with confidence of safe browsing.
2. Guarantee the integrity of the displayed browser content provided by web servers.
3. Provide assurance of safety of information stored by the browser and the computer.

1.6 Research Contributions

This research will act as repository of literature to future researchers with particular interest in web application security. It will also act us a working model to researchers building plugins and add-ons to enforce security at the client side of the application.

Chapter 2

Literature Review

2.1 Introduction

This section reviews some studies that have been conducted in relation to XSS attacks, mitigation techniques with emphasis on client-side solutions and the proposed research solution in the research to be conducted.

2.2 Cross Site Scripting (XSS)

Web applications that run on web servers and have their content rendered by web browsers are gaining explosive popularity as new classes of security bugs are introduced [31]. Security bugs such as Cross Site Scripting are the most prevalent ones [31]. According to a HP 2012 Cyber Risk Report [16], web applications are a substantial source of vulnerabilities with cross site scripting attacks contributing to 44.5% of the attacks. As more web applications are developed and deployed, the percentage increase in XSS attacks is still expected to rise. The existence of public repositories where these kinds of attacks are made available [24] has not helped matters on the web application end. XSS vulnerabilities allow injection of malicious inputs into victim web pages. The malicious data is then executed or interpreted spitefully by the browser on behalf of the victim's website [24]. Javascript, as a client-side programming language, is used to enhance user experi-

ence [38] but is now the main contributing factor to wide-spread XSS attacks [1].

The principal of same-origin policy [27] in interpretation of a Javascript code states that scripts loaded from one domain cannot be executed or have access to resources (code and data) in another domain [38]. This policy can be exploited by having a web browser receive malicious scripts deemed legitimate as they originate from a trusted site and hence conform to this same-origin policy [38]. In this case, the attacker takes advantage of a flaw in a trusted web application to download and execute the malicious Javascript code [39].

XSS attacks arise due to due developers' lack of security awareness and programming mistakes aided by time and financial constraints [38]. There is also explosion of new technologies such as AJAX that promote richer and more complex client-side interfaces while providing more avenues for potential XSS vulnerabilities [11]. HTML has also evolved from rendering static data to providing full support for dynamic execution of code embedded in web applications [24].
Other factors that have aided in exploitation of XSS vulnerabilities according to Di Lucca et al. [5] include:

- The exploits require no special application knowledge or skill.
- The attacks can bypass perimeter fences such as firewalls and cryptographic techniques.
- It is difficult for a victim to determine applications that are susceptible to these kinds of attacks.
- It is also difficult for developers to determine the elements of the web application that can be exploited.

2.2.1 Attack Scenario

A typical XSS vulnerability is exploited as below [15] [16]:

1. An attacker discovers an XSS vulnerability in a legitimate web application trusted by end-users.

2. The attacker then crafts a malicious URL in the context of a legitimate one. The URL can be used to perform malicious actions such as stealing passwords and any other sensitive information owned by a user. The malicious code contains Javascript controlled by the attacker.

3. The malicious XSS URL is then distributed via emails, message boards and social engineering sites.

4. An unsuspecting user then clicks the URL and the malicious Javascript is then executed and a user's sensitive information is then transmitted to the attacker.

2.2.2 XSS Effects

XSS threats target infrastructures ranging from small to complex ones managed by a number of leading IT vendors [11]. Popular sites such as Facebook, Twitter, Google, e-Bay, LiveJournal, MySpace, Orkut and Yahoo [39] [41] [37] have all fallen victims of XSS attacks in the recent past. XSS attacks can also act as vectors for other classes of attacks [1]. Worms such as Samy (affects web applications) can exploit XSS attacks for spreading and infects millions of users. These attacks can lead to user credential theft and performance of actions by an adversary on the user's behalf [32] which can be damaging. These range from loss of privacy to total compromise and include:

- Disclosure of user's private information such as cookies and sessions. An attacker can gain control of an account by use of this information.

- Disclosure and manipulation of end user private files.

- Installation of trojans.

- Modification of content presented by the browser.

2.2.3 Detection and Mitigation

XSS attack detection and mitigation techniques are not a new area in security research but variations exist in the design and implementation [39] [37] [17]. The easiest and fastest XSS mitigation technique that a client can apply is by disabling Javascript on the browser. However, this can not be implemented given a high percentage of websites now require Javascript to aid in content navigation, display and presentation. Some sites detect if Javascript is enabled on the browser and if not requests the user to turn it on before any other content is loaded on the browser.

There are currently two widely used approaches to protection against these types of attacks. These depend on the environment where it is being applied:

Server Side Solutions

Applied on the server environment. This involves fixing the input validation vulnerabilities by filtering and encoding inputs and outputs [10] at entry and exit points to the application - Sanitization. Sanitization is the industry standard solution that has been adopted by a number of developers [39]. The server side techniques are under the developer's discretion [39].

Client Side Solutions

The user environment is protected against XSS attacks. This involves techniques that can be employed to differentiate between normal and malicious Javascript codes [39]. Depending on the evaluation, rules can be written to mitigate the impacts of a pending XSS attack.

The server side mitigation techniques have received more research attention but are still limited in application. This is because it assumes the developers are security conscious and can implement sanitization facilities on their applications – something they seldom do. In addition, this may involve legacy applications being re-written by the developers. The rework that may be involved is overwhelming

in terms of effort and logistics required. It also requires willingness of the service providers to invest in the security of their web applications and services [39]. Server-side defenses introduce high overheads to programs slowing them down significantly and are language dependent. Redesign and reimplementation has to be effected for each and every language which is not a straight forward procedure [34]. The client side leverages these challenges with easy implementation as no redevelopment is required on the developer's side. A single plugin can be developed and embedded into a web browser handling hundreds of pages on a daily basis. This is easier compared to re-work of the hundreds of web applications. As new vulnerabilities are discovered, the same applications have to be edited to cater for new types of attacks.

2.3 Related Work

2.3.1 Acunetix WVS

Acunetix Web Vulnerability Scanner (WVS) [42] is a windows-based automated web application security testing tool. It probes web applications by checking for vulnerabilities such as Structured Query Language (SQL) Injections, Cross Site Scripting and other exploitable hacking vulnerabilities [11] [7].

Acunetix WVS checks for website or web application vulnerabilities accessible via a web browser. It also offers a strong and unique solution for analyzing off-the-shelf and custom web applications including those relying on client scripts such as JavaScript, AJAX and Web 2.0 web applications . It is suitable for any small, medium sized and large organizations with intranets, extranets, and websites aimed at exchanging and/or delivering information with/to customers, vendors, employees and other stakeholders [7].

It has a free 14 day trial version that can only scan for cross-site scripting vulnerabilities in a web application. Acunetix WVS uncovers a number of XSS vulnerabilities (true positives) in a web application but is not exhaustive [43].

It performs security analysis in three phases:

1. Crawling - It uses Acunetix DeepScan to automatically analyze and crawl a website in order to build the site's structure. This phase enumerates all files to ensure that they are scanned

2. Scanning - It launches a series of web vulnerability checks against files in the web application

3. Reporting - It contains an alert node used to report presence and description of the vulnerabilities

2.3.2 XSS-Me

XSS-Me [2] is an open source Firefox based tool used to test for reflected XSS. Developed by Security Compass, XSS-Me is aimed at developers, testers/Quality Assurance (QA) staff, and security auditors.

It traces possible entry points of XSS attacks by submitting HTML forms and substituting the form values with strings that are representative of an XSS attack. If the resulting HTML page sets a specific JavaScript value (document.vulnerable=true) [2] then the tool marks the page as vulnerable to the given XSS string. When enabled, it loads in the browser sidebar and identifies all the input fields on a page and iterates through a user provided list of XSS strings: opening new tabs and checking the results. After the process completes, a user is presented with a report of attacks that were performed successfully [13].

It employs the following approach in detection of XSS:

1. It submits HTML forms substituting them with strings that are a representative of an XSS attack.

2. If the resulting HTML page sets a specific JavaScript value (document.vulnerable=true), then the tool marks the page as vulnerable to the given XSS string.

XSS-Me can shield against a greater percentage of XSS attacks [22]. However, it can be only used for form test and single page analysis [20] and limited by the XSS attack vector base inbuilt to it.

2.3.3 XSSPloit

XSSploit [33] is a python Cross-Site Scripting scanner developed to help in discovery and exploitation of XSS vulnerabilities in penetration testing. When used against a website, it first crawls the whole website and identifies encountered forms that are possible entry points of XSS attacks such as comment boxes, search boxes, registration and login forms. It then analyses these forms to automatically detect existing XSS vulnerabilities as well as their main characteristics.

The vulnerabilities discovered are then exploited using the exploit generation engine of XSSploit. The engine allows choosing the desired exploit behaviour and automatically creates the exploit payload which can be directly embedded to an HTML link [33].

XSSploit crawling feature scans a page forms and its input elements. It does not check the URLs that are internal or external to the pages's URL which are also possible targets of XSS attacks.

2.4 Conclusion

XSS attacks are on the increase and no matter the defense approaches introduced, not all types can be mitigated effectively. The impacts of the attacks on commercial and popular social sites is even a reason for more worry, coupled with the wide adoption of web applications in this era. There is need to shift research focus on XSS attacks as it is now the leading web vulnerability known today.

Chapter 3

Methodology

3.1 Introduction

This section describes the methods that were used in gathering the requirements and further explain in detail how these methods were used to achieve the desired objectives

3.2 Archival Method

The review of the literature was paramount in the study of XSS attacks. Archival method was deployed here. The intention was to look for literature related and relevant to XSS attacks, the attack vectors and defense mechanisms that have been adopted. Weaknesses in the current technologies were also enumerated. It answered questions such as how it was done, why it was done, possible loopholes and improvements that can be addressed by the new tool.

Most defense mechanisms against XSS attacks have primarily focused on adoption of good coding practices such as proper input sanitization by the application developers. This alone is not sufficient as different forms of XSS attack vectors are being discovered at an alarming rate and application patches may not cope up with this. One XSS attack vector can come in different flavors due to different

encoding methods employed by browsers. Other techniques use a static array of XSS vectors and these may not scale well due to the dynamic nature of attacks.

Browser extensions have also been developed that can crawl websites and scan for XSS vulnerabilities. Examples include XSS-Me, NoXSS and XSS Rays. These are launched explicitly by the user and they open in the browser side bar from which scans can be directed. Novice users may find this difficult as most terms are expressed in terms that can be easily understood by security and Information Technology (IT) professionals and application developers with a security conscience.

Other defense mechanisms such as XSSDetection [13], a Python-scripted client-side solution, can also be used to perform penetration testing on web applications. This also is invoked by the web application user or developer who must be conversant with the Python programming language and the Linux terminal as it is run on the Linux shell.

All the above solutions have also ignored the cross-domain XSS attacks and these have been on the rise. They perform tests on only the domain that they are called from and any references to other domains from the target side are ignored. Their concentration also on one form of XSS has been a limiting factor.

3.3 Vector Exploit Method

3.3.1 Overview

A good and reliable XSS detection tool should be able to uncover a number of XSS vulnerabilities with low false positives and false negatives. This is always not the case due to a constantly changing or incomplete XSS vector database. The inability of tools to adapt to these changes limits their detection space hence the success rate – number of vulnerabilities detected as a percentage of the total number available in the application.

XSS detection tools should probe web applications for all sources and types of XSS attacks. Most tools focus on one or two of the following types of XSS: Reflected, Stored and DOM based. A measure of the completeness of the detection tool is in the types of vectors that it can be able to detect.

The types of applications and their hosting platforms should be considered by the XSS detection tools. A complete tool should be able to detect vectors irrespective of the programming languages the applications are developed in and the technologies/software they are deployed on.

In designing XSSD, answers to following questions in particular were important:

- Where is the XSS vector database located?
- What types of attacks should be detected?
- Which applications and hosting technologies should XSSD work with?
- How will XSSD detect XSS vulnerabilities?
- How should XSSD be run by the users?

XSSD was designed to be a complete XSS detection tool - ability to detect different types of XSS attacks using an online and dynamic vector database and also work with different types of web applications.

3.3.2 Black-Box Analysis

XSSD discovers XSS vulnerabilities using a black-box testing technique. Black-box analysis is a testing technique applied by web application scanners to find web application vulnerabilities when the internals of the applications are not known. It takes a client side viewpoint and relies on learning the behavior of a web application and manipulating different kinds of user supplied input [35].

Detections are done by sending a web application/server requests containing malicious scripts and examining server's HTTP response that may be executed by the web browser. If the response contains the original payload injected into the application, then the page is flagged vulnerable – the web application did not perform input validation or character encoding on the payload.

As an example, consider a vulnerable web page `http://www.trusted.com/register.php` and an XSS vector <script>alert(document.cookie);</script>. Combining the page and the vector (payload) generates a server request `http://www.trusted.com/register.php?<script>alert(document.cookie);</script>` which is then sent to the server. The server responds with the same page and payload which is then executed on the user's browser and the user's cookie is echoed on a browser display box. For testing purposes, this is sufficient but an attacker can definitely do more for example direct the user's cookie to a server he controls and steal the user's session.

XSSD uses different forms of the XSS vector <script>alert(1);</script>that evades most filter options used by most web applications.
The vector <script>alert(1);</script>provides a representation of numerous XSS vectors customized into it. Assuming an input field is not well sanitized, the vector is entered into the field and a POST request made to the server. The response is analyzed for the presence of the script and is deemed vulnerable to XSS if found.

DOM-based XSS attacks are detected using the location.hash() property of Javascript. Given a page, `http://www.trusted.com/guest.html#input=script>alert(1);</script>`. This payload is not sent to the server as the code fragment after '#' is considered as a DOM property of the client browser and server-side sanitization techniques cannot detect this type of XSS attack [29] . XSSD injects URLs with '#' and the payload. It then examines URL and executes the code after '#' and responses can be analyzed. If the code fragment injected is returned, then the page is flagged with a DOM XSS vulnerability.

3.3.3 Assumptions and Ethics

XSSD should work with both locally hosted and online applications on the assumption that there is an Internet connection to connect to the XSS vector database. The XSSD online database site should always be available and continuously updated with new vectors. The tool should allow for easy addition of new vector databases sites.

The users/developers trying to perform tests on their applications should be running Mozilla Firefox browser as the tool is specific to it. It is advisable to detect XSS vulnerabilities on applications that the user has control over. The tool's developer may not he held responsible for any exploits that attackers may take advantage of by performing penetration tests on the sites they do not own.

3.4 Design

The extension/add-on design can be divided into four different components as illustrated in Figure 3.1:

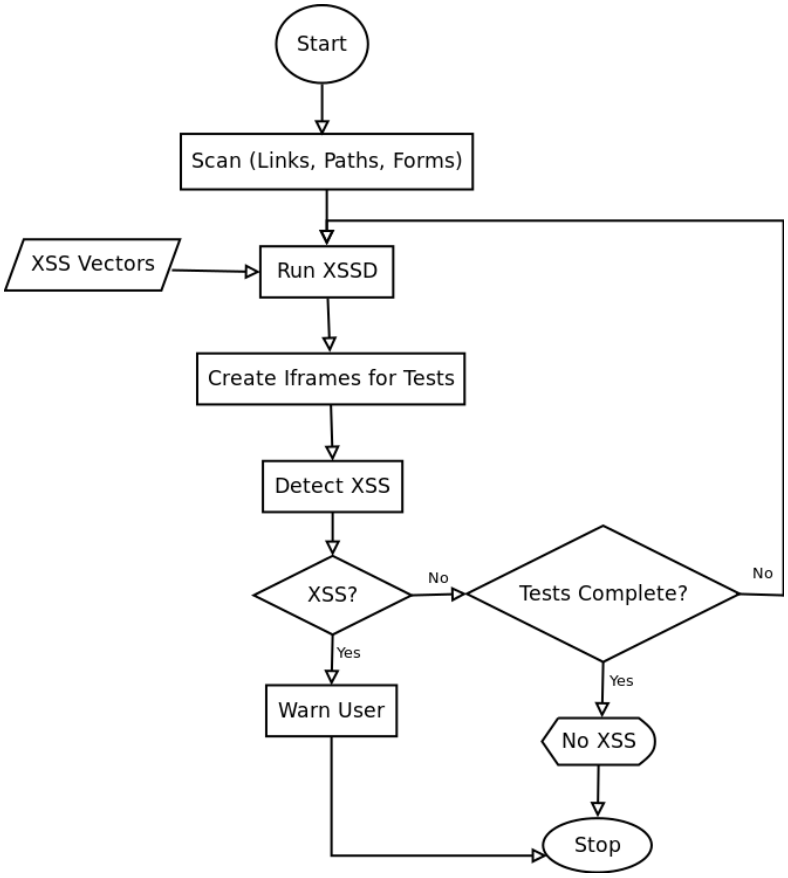


Figure 3.1: A flow chart showing how XSSD detects XSS vulnerabilities

3.4.1 Attack Vectors

An attack vector is a means by which an attacker gains access to a target system [14]. The attack vectors are some of the most common over the past few years. These are stored in an XML file and can be updated as required. The add-on has a function that retrieves and formats the XML from RSnake's updated [10] XSS Cheat Sheet [36]. The vectors can have a dedicated hosting plan and the application can allow an array of vectors to be fetched from different sources. These are fetched and parsed into an array from which they can be retrieved and tested against possible entry points of XSS attacks.

The vectors are many and can drastically reduce on the performance of the web

application. For developers, this is reasonable as nearly exhaustive tests are necessary before the application can be put into production. As for normal users, the number of XSS vectors can be reduced to enhance responsiveness of web pages.

For testing purposes, a random selection of 10 of the most common XSS attack vectors were retrieved as shown in the table 3.1:

No	Name	Code/Input
1	Standard DOM based injection single	',XSS,'
2	Standard DOM based injection double	",XSS,"
3	Standard script injection single	'><script>XSS </script>
4	Standard script injection double	"><script>XSS </script>
5	XSS Locator	';alert(String.fromCharCode(88,83,83)) //';alert(String.fromCharCode(88,83,83)) //";alert(String.fromCharCode(88,83,83))// ";alert(String.fromCharCode(88,83,83))//– ></SCRIPT>">'> <SCRIPT> alert(String.fromCharCode(88,83,83)) </SCRIPT>=&{}
6	Body Onload	<BODY ONLOAD=alert('XSS')>
7	Malformed IMG Tags	<IMG """><SCRIPT> alert("XSS") </SCRIPT>">
8	Image injection HTML breaker	'"</script>< /xml> </title>< /textarea></noscript> </style></listing>< /xmp></pre><img src=null onerror=XSS>
9	DOM based function breaker single quote	'},XSS,function x(){//
10	DOM based function breaker double quote	"},XSS,function x(){//

Table 3.1: XSS Attack Vectors

3.4.2 Scans

Scans are performed on the target site and also cross-domain for links, paths and forms. These are used as targets of XSS and the attack vectors are applied to them. The scanning component crawls web pages extracting all links/paths and forms storing them in their respective arrays.

Form method parameters are also captured from the web page such as POST and GET. Fields include text inputs and text areas that are part of the form.

3.4.3 Detection

As shown in Figure 3.1, XSSD performs detection of different types of XSS vulnerabilities through the following sequence of steps:

1. It fetches the attack vectors, sanitizes and stores them in an array
2. It then creates connections to the targets such as links and paths by using invisible iframes
3. The input vector part with the code 'XSS' is replaced with a logger or alert(1) – a common test for XSS vulnerabilities
4. For HTTP GET requests, each iframe is associated with a name which is a URL to be returned on its successful execution appended with 'xss'. If the resultant URL contains the string 'xss', then the URL from which the iframe was created is vulnerable to DOM XSS
5. For HTTP POST requests, the forms obtained by the scanning component are simulated in the iframe created. All form fields are then injected within the invisible iframes. If the resultant page response contains the script alert(1), it means the page was modified by the vector and hence vulnerable to either reflected or stored XSS attacks

3.4.4 Action

XSSD performs tests on a domain and all other paths or links from the current one. The presence of an XSS vulnerability does not mean the application user is under attack but there is a possibility of an attacker circumventing it and wreaking havoc.

Application developers have an option of performing extensive tests by using a comprehensive list of attacks fetched from `http://ha.ckers.org/xssAttacks.xml`. This can take a considerably long time to complete as the attack vectors are many. When XSS vulnerabilities are found, the developer can then patch up his/her application before deployment.

3.5 Implementation

XSSD was implemented using the following development tools:

3.5.1 Javascript

Javascript [9] is a loosely typed and interpreted programming language commonly used in browser applications that require interaction with the user. With its core language features and built-in data types that are a subject of international standards and compatibility, Javascript is considered the language of the web browser [9] [4].

In addition to the above powerful features of Javascript, it is considered as a Firefox standard for developing extensions [25]. Javascript was used to develop the logic and functionality of XSSD - the scanning, detection and reporting features.

3.5.2 XUL

XML User Interface Language (XUL) [28] is an XML-based language used for programming user interface widgets such as buttons, menus and toolbars. Javascript was used to attach functionality to the widgets. Firefox browser interface is also

developed using XUL [8].

XUL was used to program XSSD's user interface widgets such as the buttons, text inputs and labels. It was also used to position the widgets on the browser. Windows such as options/preferences were all implemented in XUL.

3.5.3 CSS

Cascading Style Sheets (CSS) [26] is a style sheet language used for presentation of documents written in a markup language such as HTML, XML or XUL. CSS was used for separation of document content from its presentation. This enabled different elements (attributes) in the document to share the same styles without the need for rewriting them [18].

CSS was also used to style the widgets in XSSD with color and fonts.

3.5.4 Distribution

XSSD will be packaged in XPI - a technology used by the Mozilla Application Suite, SeaMonkey, Mozilla Firefox, Mozilla Thunderbird and other XUL-based applications for installing Mozilla extensions that add functionality to the main application [40].

After reviews by Mozilla, XSSD will be publically accessible from the official Mozilla add-ons page `http://addons.mozilla.org/`. A local copy of the XPI file can also be installed to Firefox directly by opening it from **File** → **Open File** menu and submenu respectively.

3.5.5 Interfaces

XSSD's interface implementation can be divided into the Client and Developer Interfaces.

Client Interface

XSSD, assuming a web page has been opened, can be launched from the browser in three ways:

1. When set to auto-run, XSSD runs on every page load. This is not always advisable especially if multiple pages are going to be opened at the same time.
2. From Firefox,open the **Tools** Menu → **XSS Diffuse** → **Run XSS Diffuse**.
3. XSSD can also be directly launched from the Add-ons bar of Firefox.

The extension also has settings for optimization especially on the number of XSS vectors that need to be run. These can be made from **Tools** Menu → **XSS Diffuse** → **Options**.

Developer Interface

XSSD, like most Mozilla Firefox extensions, is developed using XUL, Javascript and CSS. XUL is used mainly in defining the structure of the interfaces, CSS to style the add-on features and Javascript to define the logic of the application. The translation files are used to localize the application.

3.6 Testing and Evaluation

3.6.1 Introduction

Tests were performed on the extension's functional components such as the ability to scan URLs and forms and its XSS vulnerability detection power. The system was later tested as a whole and features such as its performance were noted.

Testing and evaluation was necessary to confirm that XSSD operates as per its main design objective thus detect XSS vulnerabilities in a web application. Using

these results, it was easy to demonstrate the extension had met its requirements.

It was also important in providing a basis for further refinements. In some instances, errors/faults were encountered during the test runs hence a need for modification of the extension.

3.6.2 Experimental Setup

Four vulnerable web applications were used to perform tests. The applications (Wackopicko, XSS Education, Acunetix TestPHP and buggy web application (bWAPP)) are developed under the LAMP (Linux, Apache, MySQL and PHP) software stack.

1. Wackopicko [6] provides clearly defined vulnerabilities, is easily customizable and a representative of most web applications in functionality and development technologies.

2. XSS Education [12] is a light application with a set of XSS vulnerable PHP pages for testing.

3. bWAPP [21] is an insecure web application with over 100 web vulnerabilities. It aids security enthusiasts, developers and students in discovering and preventing web vulnerabilities. It is a handy application in penetration testing and ethical hacking projects.

4. Acunetix TestPHP [42] is provided by Acunetix as a test and demonstration site for their application Acunetix Web Vulnerability Scanner. It enables their new customers to test the efficiency of their application in discovering web vulnerabilities.

LAMP was chosen because of the prevalent use and popularity of its technologies. MySQL is the world's most popular open source database software initially developed by Sun Microsystems and now maintained by Oracle [23]. Apache is

used because of its dominance in the web server industry and its easy installation and integration in Linux [35]. PHP is selected because it is considered a more vulnerable programming language according to CVE [3].

All the applications selected are vulnerable to the three types of XSS attacks. They only differ in the specifics and types of XSS vectors and the number of vulnerabilities exploitable. Depending on the complexity of an application, it can have numerous URLs including cross-domain ones. To reduce on the running time of XSSD, vulnerable absolute URLs were used for tests.

The XSS attack vectors used are some of the recent in popular web applications as provided by `http://ha.ckers.org/xssAttacks.xml`. The web applications are hosted on the local machine (Apache Root Directory for LAMP applications) and accessed via `http://localhost/[Application]`.

The tests were performed using one of the latest versions of Firefox (Firefox 31.0) on a Laptop running Ubuntu 12.04.5 LTS with fairly good specifications:

- Intel Core i3-3227U Processor (1.9 GHz)
- 6 GB DDR3 RAM
- 500 GB 5400 rpm Hard Drive
- 14.0-Inch Screen
- Internet Connection

The ability of XSSD to detect the XSS vulnerabilities using the retrieved vectors in the web application was used as a validation measure of the tool.

Chapter 4

Presentation and Discussion of Results

4.1 Introduction

This chapter presents XSSD in action and compares to popular XSS web vulnerability scanners in detection speed and efficiency. The scanners under consideration are Acunetix, XSSMe and XSSPloit.

4.1.1 XSSD - Performance

Depending on the degree of an application's vulnerability, XSSD detects a number of XSS vulnerabilities with respect to the 10 XSS attack vectors in table 3.1.

In WackoPicko, XSSD shows that the application is vulnerable to all the 10 XSS attack vectors using one of the page's URLs `http://localhost/guestbook.php` in approximately 32 seconds.

In bWAPP, XSSD detects 12 XSS vulnerabilities using two of the application's vulnerable pages. The two pages are vulnerable to both stored and reflected XSS attacks. This is also the case with Acunetix TestPHP.

XSS Education is vulnerable to a total of 10 XSS attack vulnerabilities using its two pages as detected by XSSD. The combined time taken by XSSD is approximately 21 seconds.

In summary, as shown in table 4.1, XSSD detects a total of 45 XSS vulnerabilities in the four web applications using two vulnerable pages per application. XSSD takes 222.1 seconds in total which translates to 0.04 vulnerabilities per second.

No	**Applications**	**URLs**	**V(No)**	**T(s)**	**V/T**
1	WackoPicko	`http://localhost/guestbook.php`	10	31.9	0.31
		`http://localhost/users/login.php`	1	35.1	0.03
2	bWAPP	`http://localhost/bWAPP/xss_get.php`	6	41.7	0.14
		`http://localhost/bWAPP/xss_stored_1.php`	6	35.7	0.17
3	Acunetix TestPHP	`http://testphp.vulnweb.com/guestbook.php`	6	38.2	0.16
		`http://testphp.vulnweb.com/signup.php`	6	18.5	0.32
4	XSS Education	`http://localhost/xssed/basicxss/`	6	10.1	0.59
		`http://localhost/xssed/javascriptxss/`	4	10.9	0.37
Total			**45**	**222.1**	**0.20**

Table 4.1: Performance of XSSD against the four vulnerable web applications: V(No) - Number of Vulnerabilities, T(s) - Time in Seconds, V/T - Number of Vulnerabilities per second

4.1.2 Acunetix

Acunetix is a Web Vulnerability Scanner that automatically checks web applications for SQL Injection, XSS and other web vulnerabilities. It is an executable application that can be installed on a Windows Operating System. Wine Program loader was used to run it from Ubuntu 12.04.5 LTS.

4.1.3 XSSMe

An open source Mozilla Firefox add-on used to test for reflected Cross-Site Scripting and launched from the browsed page sidebar. It is developed by Security-Compass. XSSMe has a flexible database of XSS attack vectors that can be edited.

4.1.4 XSSploit

A multi-platform Cross-Site Scripting scanner and exploiter written in Python. It helps in discovering and exploiting XSS vulnerabilities in penetration tests.

4.2 Presentation of Test Results

The above XSS vulnerability scanners were put to test using the four vulnerable applications. Two vulnerable URLs per application were used on the same set of hardware and software.

The number of vulnerabilities per URL using the scanners are presented in Table 4.2

No	Applications	URLs	Acunetix	XSSMe	XSSPloit	XSSD
1	WackoPicko	http://localhost/guestbook.php	3	3	1	10
		http://localhost/users/login.php	1	3	0	1
2	bWAPP	http://localhost/bWAPP/xss_get.php	4	0	0	6
		http://localhost/bWAPP/xss_stored_1.php	4	0	0	6
3	Acunetix TestPHP	http://testphp.vulnweb.com/guestbook.php	5	9	3	6
		http://testphp.vulnweb.com/signup.php	1	3	3	6
4	XSS Education	http://localhost/xssed/basicxss/	1	3	1	6
		http://localhost/xssed/javascriptxss/	2	0	1	4
Total			**21**	**21**	**9**	**45**

Table 4.2: Number of XSS Vulnerabilities discovered

The times taken by the scanners to detect these vulnerabilities in the URLs were also recorded in Table 4.3

No	Applications	URLs	Acunetix	XSSMe	XSSPloit	XSSD
1	WackoPicko	http://localhost/guestbook.php	17.6	7.0	9.5	31.9
		http://localhost/users/login.php	6.2	6.9	12.0	35.1
2	bWAPP	http://localhost/bWAPP/xss_get.php	97.0	19	8.0	41.7
		http://localhost/bWAPP/xss_stored_1.php	70.0	17	8.0	35.7
3	Acunetix TestPHP	http://testphp.vulnweb.com/guestbook.php	131	21.7	220.0	38.2
		http://testphp.vulnweb.com/signup.php	131.0	9.0	246.0	18.5
4	XSS Education	http://localhost/xssed/basicxss/	6.1	3.3	6.5	10.1
		http://localhost/xssed/javascriptxss/	10.1	3.7	4.2	10.9
Total			**469.0**	**87.6.9**	**514.2**	**222.1**

Table 4.3: Times taken to detect XSS vulnerabilities in seconds

The results from tables 4.2 and 4.3 can then be used to compute the number of vulnerabilities discovered per second by the four XSS detection tools. This is shown in Table 4.4

	Acunetix	XSSMe	XSSPloit	XSSD
Vulnerabilities per second(V/s)	0.04	0.24	0.02	0.20
V/s as %	8.0	48.0	4.0	40.0

Table 4.4: Vulnerabilities discovered per Second and % Detection Rate

The number of XSS vulnerabilities discovered per second and expressed as a per-

centage using the four XSS detection tools is shown figure 4.1:

4.3 Discussion of Results

WackoPicko, bWAPP, Acunetix TestPHP and XSS Education are four vulnerable web applications with the three different types of XSS vulnerabilities - Reflected, Stored and DOM. They are also developed using different web technologies and hosted on the Apache platforms as shown in Table 4.5

No	**Application**	**Web Technology**	**Hosting Platform**
1	**WackoPicko**	PHP and MySQL	Apache
2	**bWAPP**	PHP and MySQL	Apache
3	**Acunetix TestPHP**	PHP and MySQL	Apache
4	**XSS Education**	PHP and Javascript	Apache

Table 4.5: Applications - Web technologies and hosting platforms

4.3.1 Performance

XSSD performs fairly well compared to other XSS detection tools as shown in Figure 4.1. Its detection rate of vulnerabilities per second is 0.20.

In WackoPicko, XSSD performs exceptionally well as it detects over 11 XSS vulnerabilities when run against the 10 selected XSS attack vectors. XSSD also provides a Proof of Concept (PoC) link for each and every vulnerability detected as depicted in Table 4.6 . This aids a developer in replicating the vulnerability and also in testing for false positives in XSSD.

In Acunetix TestPHP, XSSD still does well in comparison to the other three tools - even better than Acunetix. XXSPloit fails in applications such as bWAPP that requires a user to be logged into the page before performing any tests on it. Acunetix circumvents this by use of a login sequence specified in the Run Tests Wizard.

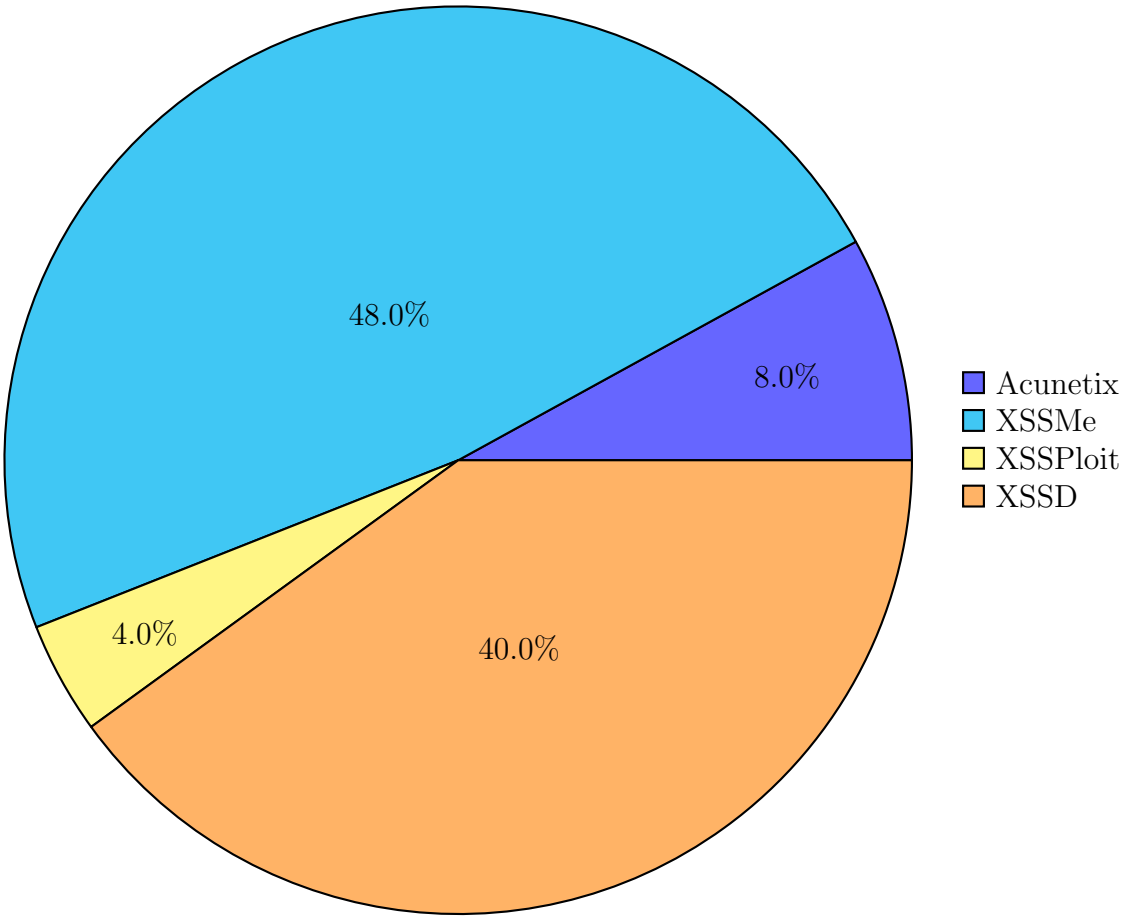


Figure 4.1: Percentage of XSS vulnerabilities discovered per second

XSSMe and XSSD are both launched from the opened web page. XSSD adds an option of autorun - it is invoked upon every new page load when set. XSSMe has a predefined set of XSS attack vectors while XSSD uses a publically available set that is usually updated. XSSMe detects mainly DOM-based XSS vulnerabilities compared to XSSD.

XSSD takes the longest times for detection. This is partly due to network latencies encountered in fetching the vectors online. It is also due to the high number of attack vectors that are retrieved and its functioning - get all entries (links, paths and forms) and run the vectors against each and every entry.

No	URL	XSS Vector	Method	PoC
1	http://localhost/guestbook.php	BODY ONLOAD	POST	http://localhost/guestbook.php?query=\%3CBODY\%20ONLOAD=alert(1)\%3E&name=\%3CBODY\%20ONLOAD=alert(1)\%3E&comment=\%3CBODY\%20ONLOAD=alert(1)\%3E&
2	http://localhost/guestbook.php	XSS Quick Test	GET	http://localhost/guestbook.php/\%27\%27\%3B!--\%22\%3Calert(1)\%3E\%3D\%26\%7B()\%7D/
3	http://localhost/guestbook.php	BODY background-image	GET	http://localhost/guestbook.php/\%3CIMG\%20SRC\%3D\%22jav\%26\%23x09\%3Bascript\%3Aalert(1)\%3B\%22\%3E/
4	http://localhost/guestbook.php	Embedded Encoded Tab POST	POST	http://localhost/guestbook.php?query=\%3CIMG\%20SRC=\%22javascript:alert(1);\%22\%3E&name=\%3CIMG\%20SRC=\%22javascript:alert(1);\%22\%3E&comment=\%3CIMG\%20SRC=\%22javascript:alert(1);\%22\%3E&
5	http://localhost/guestbook.php	SCRIPT w/Alert()	GET	http://localhost/guestbook.php?query=\%3CSCRIPT\%3Ealert(1)\%3C/SCRIPT\%3E&name=\%3CSCRIPT\%3Ealert(1)\%3C/SCRIPT\%3E&comment=\%3CSCRIPT\%3Ealert(1)\%3C/SCRIPT\%3E&

Table 4.6: Sample output content data from XSSD

4.3.2 Limitations

XSSD, despite its ability to detect XSS vulnerabilities, has a few limitations that include:

- It does not have a tool for detection of false negatives. A learning tool could be implemented using previous test results to determine these false negatives
- XSSD has a high performance drain off the web browser. This could be solved by providing users with an option of specifying the depth of scans to be performed.
- Requires an active Internet connection. The users could have an option to download all the attack vectors and host them locally.

4.3.3 Conclusion

XSSD limitations provide an avenue for further research on how it can be improved especially in terms of speed. Nonetheless, XSSD can be widely adopted by web developers to perform penetration tests on their applications before deployment hence aiding them in writing secure code.

Chapter 5

Conclusion, Recommendations and Future Works

5.1 Introduction

This section provides a summary of XSSD, recommendations proposed by the researcher and a direction for future work research

5.2 Conclusion

This research focused on a general introduction to the types of XSS attacks and the reasons as to why there is need for worry and more research in this field of security. It sets the stage by providing an analysis of the current explosion in deployment of web applications. It also provides a background study on the existing research that has embarked on XSS attacks - they have overtaken buffer over-flows as the number one security vulnerability.

XSSD, an XSS detection tool, was developed in Javascript as a Mozilla Firefox add-on. It is based on other XSS detection tools composed of three main components - Scan, Inject and Analyze. XSSD scans forms, links and paths. It then retrieves the XSS attack vectors over a computer network and injects these to the

forms, links and paths. It has an analysis component that flags GET or POST HTTP requests as XSS vulnerable.

Three XSS vulnerability scanners (Acunetix, XSSplot and XSSMe) were put to test against four XSS vulnerable applications (WackoPicko, bWAPP, XSS Education and Acunetix TestPHP). XSS vulnerabilities were detected by the scanners and XSSD performance came top in terms of the number of vulnerabilities detected and also its ability to provide a PoC to the developers/testers.

5.3 Recommendations

The researcher presents the following recommendations:

- Web application development is and will continue to be on the rise hence a need for developers to be conversant with security loopholes and implications especially XSS attacks.
- Web developers need to perform tests on their applications before they are deployed on the network. XSSD just provides a good starting point.
- XSS vulnerabilities are so dynamic especially when encoding practices are brought to picture. A single detection mechanism will not circumvent all possible attacks. XSSD just provides a developer an insight on possible entry points of attackers and it is up to the developer to provide concrete sanitization of user inputs.
- More research needs to be done on XSS attacks as they come in different forms. The discovery rate is alarming and incorporation of its study in current education setups will definitely pay back.
- XSSD only works in Mozilla Firefox hence there is need to build a platform independent application given there are a number of web browsers in the market today.

5.4 Future Work

Developers need to embrace the use of XSSD to detect XSS web vulnerabilities and improve on sanitization of user inputs in their applications. This is the first version and enhancements need to be made in order to provide more efficiency and improved performance.

1. A research on the training mechanism should be carried out. If enabled, XSSD should be able to detect false negatives and provides a training base for future detections.

2. A portability mechanism needs to be developed to enable usage in other web browsers as an extension. Browsers such as Google Chrome and Internet Explorer will then be able to also utilize XSSD.

Bibliography

[1] Prithvi Bisht and VN Venkatakrishnan. "XSS-GUARD: precise dynamic prevention of cross-site scripting attacks". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2008, pp. 23–43.

[2] Security Compass. *XSS-Me*. 2014. URL: `http://labs.securitycompass.com/exploit-me/xss-me/` (visited on 08/18/2014).

[3] The MITRE Corporation. *Common Vulnerabilities and Exposures*. URL: `http://cve.mitre.org/` (visited on 08/12/2014).

[4] Douglas Crockford. *JavaScript: The Good Parts: The Good Parts.* " O'Reilly Media, Inc.", 2008.

[5] Giuseppe A Di Lucca et al. "Identifying cross site scripting vulnerabilities in web applications". In: *Telecommunications Energy Conference, 2004. INTELEC 2004. 26th Annual International*. IEEE. 2004, pp. 71–80.

[6] Adam Doupé, Marco Cova, and Giovanni Vigna. "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.

[7] LaShanda Dukes, Xiaohong Yuan, and Francis Akowuah. "A case study on web application security testing with tools and manual testing". In: *Southeastcon, 2013 Proceedings of IEEE*. IEEE. 2013, pp. 1–6.

[8] Kenneth C Feldt. *Programming Firefox: Building rich internet applications with XUL.* " O'Reilly Media, Inc.", 2007.

[9] David Flanagan. *JavaScript: the definitive guide.* " O'Reilly Media, Inc.", 2006.

[10] Jeremiah Grossman. *XSS Attacks: Cross-site scripting exploits and defense.* Syngress, 2007.

[11] Supriya Gupta and Lalitsen Sharma. "Analysis and Assessment of Web Application Security Testing Tools". In: ().

[12] HackMe. *XSS Education.* URL: `https://hack.me/101136/xss-education.html` (visited on 07/29/2014).

[13] Mohammed H. Abu Hamada. "Client Side Action Against Cross Site Scripting Attacks". MA thesis. Islamic University – Gaza, 2012.

[14] Simon Hansman and Ray Hunt. "A taxonomy of network and computer attacks". In: *Computers & Security* 24.1 (2005), pp. 31–43.

[15] Michael Howard, David LeBlanc, et al. *24 Deadly Sins of Software Security.* McGraw-Hill/Osborne California, 2009.

[16] HP. *HP 2012 Cyber Risk Report.* Tech. rep. HP, 2012.

[17] Trevor Jim, Nikhil Swamy, and Michael Hicks. "Defeating script injection attacks with browser-enforced embedded policies". In: *Proceedings of the 16th international conference on World Wide Web.* ACM. 2007, pp. 601–610.

[18] Matthias Keller and Martin Nussbaumer. "Cascading style sheets: a novel approach towards productive styling with today's standards". In: *Proceedings of the 18th international conference on World wide web.* ACM. 2009, pp. 1161–1162.

[19] Xiaowei Li and Yuan Xue. "A survey on server-side approaches to securing web applications". In: *ACM Computing Surveys (CSUR)* 46.4 (2014), p. 54.

[20] Bei Hai Liang et al. "Research on Vulnerability Detection for Software Based on Taint Analysis". In: *Applied Mechanics and Materials* 347 (2013), pp. 3715–3720.

[21] Malik Mesellem. *bWAPP a buggy web application.* URL: `http://www.mmeit.be/bwapp/index.htm` (visited on 09/27/2014).

[22] Bhawna Mewara, Sheetal Bairwa, and Jyoti Gajrani. "Browser's defenses against reflected cross-site scripting attacks". In: *Signal Propagation and Computer Technology (ICSPCT), 2014 International Conference on*. IEEE. 2014, pp. 662–667.

[23] MySQL. *About MySQL*. URL: `http://www.mysql.com/about` (visited on 07/12/2014).

[24] Yacin Nadji, Prateek Saxena, and Dawn Song. "Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense." In: *NDSS*. 2009.

[25] Mozilla Developer Network. *About Javascript*. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript` (visited on 09/29/2014).

[26] Mozilla Developer Network. *CSS*. URL: `https://developer.mozilla.org/en-US/docs/Web/CSS` (visited on 09/29/2014).

[27] Mozilla Developer Network. *Same Origin Policy for Javascript*. 2014. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Same_origin_policy_for_JavaScript` (visited on 03/09/2014).

[28] Mozilla Developer Network. *XUL*. URL: `https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL` (visited on 09/29/2014).

[29] Suman Saha. "Consideration Points Detecting Cross-Site Scripting". In: *arXiv preprint arXiv:0908.4188* (2009).

[30] Suman Saha, Shizhen Jin, and Kyung-Goo Doh. "Detection of DOM-based Cross-Site Scripting by Analyzing Dynamically Extracted Scripts". In: ().

[31] Prateek Saxena, David Molnar, and Benjamin Livshits. *Scriptgard: Preventing script injection attacks in legacy web applications with automatic sanitization*. Tech. rep. Citeseer, 2010.

[32] Prateek Saxena, David Molnar, and Benjamin Livshits. *Scriptgard: Preventing script injection attacks in legacy web applications with automatic sanitization*. Tech. rep. Citeseer, 2010.

[33] SCRT Information Security. *XSSploit*. URL: http://www.scrt.ch/en/attack/downloads/xssploit (visited on 06/12/2014).

[34] R Sekar. "An Efficient Black-box Technique for Defeating Web Application Attacks." In: *NDSS*. 2009.

[35] David A. Shelly. "Using a Web Server Test Bed to Analyze the Limitations of Web Application Vulnerability Scanners". MA thesis. Virginia Polytechnic Institute and State University, 2010.

[36] R Snake. *XSS (Cross Site Scripting) Cheat Sheet*. URL: http://ha.ckers.org/xss.html (visited on 04/10/2014).

[37] Mike Ter Louw and VN Venkatakrishnan. "Blueprint: Robust prevention of cross-site scripting attacks for existing browsers". In: *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE. 2009, pp. 331–346.

[38] Philipp Vogt et al. "Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis." In: *NDSS*. 2007.

[39] Joel Weinberger et al. "A systematic analysis of xss sanitization in web application frameworks". In: *Computer Security–ESORICS 2011*. Springer, 2011, pp. 150–171.

[40] Wikipedia. *XPInstall - Wikipedia, The Free Encyclopedia*. 2013. URL: http://en.wikipedia.org/wiki/XPInstall (visited on 03/09/2014).

[41] Peter Wurzinger et al. "SWAP: Mitigating XSS attacks using a reverse proxy". In: *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*. IEEE Computer Society. 2009, pp. 33–39.

[42] Acunetix WVS. *Web Vulnerability Scanner v9 - User Manual*. 2014. URL: http://www.acunetix.com/vulnerability-scanner/wvsmanual.pdf (visited on 07/09/2014).

[43] Xiaohong Yuan. "A Case Study on Web Application Security Testing with Tools and Manual Testing". In: ().

Chapter 6

Appendices

6.1 Project Timeline

No	Activity	Start Date	End Date
1	Concept Paper Writing and Approval	October 2013	January 2014
2	Proposal Write-up and Approval	January 2014	March 2014
3	Methodology (Literature Review, Requirements Analysis, Design)	March 2014	April 2014
4	Implementation	April 2014	May 2014
5	Testing, Validation and Evaluation	May 2014	June 2014
6	Report Write-up and Submission	April 2014	June 2014

Table 6.1: Project Timeline

6.2 Budget (Uganda Shillings)

No	Item	Quantity	Unit Cost	Total Cost
1	Laptop	1	2,500,000	2,500,000
2	Stationery	4	10,000	40,000
3	Photocopying	500	100	50,000
4	Printing	1,000	100	100,000
5	Binding	10	10,000	100,000
6	Miscellaneous	1	1,000,000	1,000,000
				3,790,000

Table 6.2: Budget (Uganda Shillings)

6.3 XSSD Source Code

```
1  //function fetchxssvectors retrieves the XSS vectors from
       remote or local URLs (vectorsURL)
2  function fetchxssvectors (array vectorsURL, array vectors)
       {
3    //Iterate through the URLs in order to fetch the vectors
4    for (i=0; i<vectorsURL.length; i++){
5        //create a new XMLHttpRequest Object
6        var xhr = new XMLHttpRequest();
7        xhr.onreadystatechange = function() {
8          if (xhr.readyState == 4) {
9            parser = new DOMParser();
10           dom = parser.parseFromString(xhr.responseText,"
                 application/xml");
11           dom = dom.getElementsByTagName('attack');
12           for (var i=0; i<dom.length; i++){
13             //Removal of XSS potential Attacks from the
                   name attribute
14             name = dom[i].getElementsByTagName('name')[0].
                   textContent;
15             name = name.replace('>',''); name =
                   name.replace('<','');

17             //Extract the code and name portions from the
                   parsed XML
18             code = dom[i].getElementsByTagName('code')[0].
                   textContent;
19             input = unescape(code);

21             //Store the vector data in an array
22             vectors.push({input: input, name: name});
23           }
```

```
24          }
25         }
26          xhr.open('GET', vectorsURL[i] , true);
27          xhr.send(null);
28       }
29 }

31 //Scan the links, forms and paths
32 function scan() {
33     //perform some input validation
34     if (vectors_number > vectors.length){
35         alert("Check your XSS Diffuse Number of Vectors
               Option");
36     }
37     //Crawl the page for links, forms and paths
38     else {
39         scanLinks();
40         scanForms();
41         scanPaths();
42     }
43 }

45 //function runxssd to perform the detection
46 function runxss(url, method, vector){
47     //Global Iframe Settings
48     //create the invisible iframe and initialize it
49     var iframe = content.document.createElement('iframe');
50     iframe.style.display = 'none';
51     iframe.id = 'ray'+uniqueID;
52     iframe.time = timestamp();
53     iframe.name = url + '#xss';

55    //append the iframe to the body
```

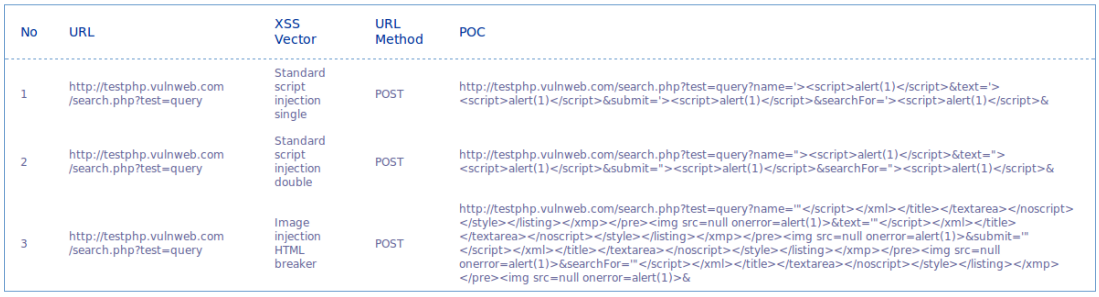
```
56      content.document.body.appendChild(iframe);

58      // Detect XSS potential when the iframe loads
59      iframe.onload = function() {
60       try {
61         //Using the location.hash property
62         if(iframe.contentWindow.location.hash.slice(1) == '
             xss') {
63           //Indicates the URL is vulnerable (Log this)
64           logxss(iframe.id);
65           if (content.document.getElementById(iframe.id)) {
66             complete();
67             //Remove the iframe
68             content.document.body.removeChild(iframe);
69           }}
70       }
71        catch(e){alert(e);}
72  }
73      //GET Method
74      if (method === 'GET') {
75         iframe.src = url;
76         content.document.body.appendChild(iframe);
77      }
78      //POST Method
79      else if (method === 'POST') {
80          //Create a replica form of the page
81          var form = '<form action="'+escape(url)+'" method="
              post" id="frm" accept-charset="UTF-8" >';
82          //Iterate through the Form Fields and create them
              in the new replica
83          for(var i in params) {
84              if(params.hasOwnProperty(i)){
85                if (excludeList.test(i)) {
```

```
86                  form += '<textarea name="' + i + '">' +
                        escape(params[i]) + '<\/textarea>';
87                  continue;
88              }
89          }
90      }
91      form += '<\/form>';
92      content.document.body.appendChild(iframe);
93      //Add the form data to iframe and submit it
94      var data = form +'<script>document.createElement("form
            ").submit.apply(document.forms[0]);<\/script>'
95      var script =
            iframe.contentWindow.document.createElement('script'
            );
96      script.type = 'application/x-javascript';
97      script.textContent = "document.writeln('" +data +";')";
98      iframe.contentWindow.document.body.appendChild(script);
99  }
100 });
101 }
102 //Log the XSS vulnerability
103 function logxss(uniqueID) {
104   uniqueID = uniqueID.replace(/[^0-9]+/,'');
105   ray = rays[uniqueID];
106   durl = escape(ray.url.replace(/&/g,'&amp;'));
107   dvector = escape(ray.vector.name);
108   dmethod = escape(ray.vector.method);
109   dpoc = escape(ray.vector.poc);

111   //Store the vulnerabilities in array
112   glb_xss.push({url:durl, vector:dvector, method: dmethod,
          poc: dpoc});
113 }
```

6.4 Sample Output



No	URL	XSS Vector	URL Method	POC
1	http://testphp.vulnweb.com /search.php?test=query	Standard script injection single	POST	http://testphp.vulnweb.com/search.php?test=query?name='><script>alert(1)</script>&text='> <script>alert(1)</script>&submit='><script>alert(1)</script>&searchFor='><script>alert(1)</script>&
2	http://testphp.vulnweb.com /search.php?test=query	Standard script injection double	POST	http://testphp.vulnweb.com/search.php?test=query?name="><script>alert(1)</script>&text="> <script>alert(1)</script>&submit="><script>alert(1)</script>&searchFor="><script>alert(1)</script>&
3	http://testphp.vulnweb.com /search.php?test=query	Image injection HTML breaker	POST	http://testphp.vulnweb.com/search.php?test=query?name='"</script></xml></title></textarea></noscript> </style></listing></xmp></pre><img src=null onerror=alert(1)>&text='"</script></xml></title> </textarea></noscript></style></listing></xmp></pre><img src=null onerror=alert(1)>&submit='" </script></xml></title></textarea></noscript></style></listing></xmp></pre><img src=null onerror=alert(1)>&searchFor='"</script></xml></title></textarea></noscript></style></listing></xmp> </pre><img src=null onerror=alert(1)>&

Figure 6.1: Sample Output on running XSSD on a vulnerable web page