UNIVERSITY OF NAIROBI

SCHOOL OF COMPUTING AND INFORMATICS

A HYBRID ALGORITHM FOR DETECTING WEB-BASED APPLICATIONS VULNERABILITIES

BY

MUIRURI, CHRIS KARUMBA

REG NO. P53/72671/2014

SUPERVISOR

SAMUEL RUHIU

A research project report submitted to in partial fulfilment for the requirements of an award of the degree of Master of Science in Distributed Computing Technology of the University of Nairobi

December 2015

DECLARATION

This research project report is my original work and has not been presented for any award in any other university.

Signature …………………………………… Date ……………………………………...

Muiruri Chris Karumba

Reg No. P53/72671/2014

This research project has been submitted as part of fulfillment of the requirements for the award of MSc Distributed Computing Technology at the University of Nairobi, with my approval as University Supervisor.

Signature:………………………………………Date………………………………

Samuel Ruhiu,

School of Computing and Informatics

DEDICATION

This research project is dedicated to my wife, Leanne, my son, Wayne Muiruri, Josephat Kung'u, Patrick Ngwiri and my dear parents, for their continued support and encouragement.

ACKNOWLEDGEMENT

I wish to extend my sincere appreciation to my supervisor Mr Samuel Ruhiu for his invaluable advice and guidance. The MSc project assessment panel Prof. Peter Waiganjo Wagacha, Dr. Dan Orwa and Christopher Moturi who scrutinized my work and provided valuable advice.

I'm grateful to my family for their support, and above of all, I thank God for the gift of wisdom and strength to complete this project successfully. To any other person or institution that might have assisted me financially or otherwise, may the Lord bless you abundantly!

.

Abstract

Web applications have gained popularity over the years and have become an integral part of our daily lives interaction. We use these applications on a regular basis to intract with our friends and family, purchase items online and access bank accounts among others.

However, these appliactions are not 100% secure, they are subject to a wide array of vulnerablities such as such as SQL injection, Cross site tracing , cross site refrence forgery and server side injections among others.To discover these weaknesses, web application scanners are used to report vulnerabilities found.

The main objective of this study is to perform a comparative study of open source vulnerability testing tools, study their algorithm for these tools and propose an improved hybrid algorithm. A simulation to test and validate the hybrid algorithm was also developed.

This research focuses on six of the open source web scanning tools which, were tested against four web based applications with known vulnerabilities to compare the tools capabilites and features. In addition, the algorithm of these tools were scrutinized with an aim of producing a hybrid algorithm that will be more accurate in detecting web vulnrabilities.

The experimental results were compared with the existing open source tools to confirm the effectiveness of the approach used.

The research concluded that open source tools have the capacity to detect vulnerabilities in the test cases performed. However, none of the tools have the capacity to detect all the vulnerabilities. For this reason there is need to improve web scanning tools and increases their detection accuracy.

Table of Contents

CHAPTER 1 : INTRODUCTION....1

1.1 Background Information. 1

1.2 Statement of the Problem....3

1.3 Main Objective....4

1.4 Justification of the Study....4

1.5 Limitations of the Study....5

CHAPTER 2 : LITERATURE REVIEW 6

2.0 Introduction....6

2.2 Classification of Web Scanning Tools....7

2.3 Types of Web Application Vulnerabilities....8

2.4 Web application Vulnerabilities Testing Tools....14

2.5 Algorithms....15

2.6 Hybrid Algorithms....18

2.7 Web Based Applications....19

2.8 Web Vulnerability Scanning Tools....20

2.9 Algorithms Used by Existing Tools....22

2.10 Related Studies....24

2.11 Research Overview....25

CHAPTER 3 : RESEARCH METHODOLOGY....27

3.0 Introduction....27

3.1 Research Design....27

3.3 Target Population....31

3.4 Sampling Procedure....31

3.5 Sample Size....32

3.6 Data Collection....32

3.7 Data Analysis....34

3.8 Data Presentation....35

3.9 Limitation and Assumptions....35

CHAPTER 4 : RESULTS AND DISCUSSION 36

4.1 Proposed Hybrid Algorithm 36

4.2 Simulation Implementation 36

4.3 Data Analysis 53

4.4 Data Description 53

4.5 web Applications Scanning Results..53

4.6 Data Presentation 55

4.7 Discussion 61

CHAPTER 5 : CONCLUSION AND RECOMMENDATIONS......................63

5.1 Mapping Research Objectives to the Methodology 63

5.2 Limitations 63

5.3 Conclusion..64

References..67

Appendices...70

Appendix 1: Screen shot captured during the scanning process 70

Appendix 2: User manual 73

Appendix 3: Source code samples...74

List of Figures

Figure 2:1 Common web vulnerabilities 9
Figure 2:2 Attack module algorithm. 23
Figure 2:3 An extract from SQL detection module 23
Figure 2:4 Algorithm used by Vega – sample 23
Figure 2:5 Research Overview diagram 26
Figure 3:1 Hybrid Algorithm methodology 28
Figure 3:2 Hybrid Algorithm Design 30
Figure 3:3 Illustration of the web application scanning process. 31
Figure 3:4 Vulnerability scanning process 34
Figure 4:1 Overview Flowchart 38
Figure 4:2 Crawling Flowchart (1) 41
Figure 4:3 Scanning Flowchart (2) 43
Figure 4:4 Flowchart SQL Injection (3) 44
Figure 4:5 Cross Site Scripting Flowchart (4) 47
Figure 4:6 Cross Site Request Forgery Flowchart (5) 48
Figure 4:7 Command Injection Flowchart (6) 50
Figure 4:8 X-Path Injection Flowchart (7) 52
Figure 4:9 Web Scanning Tools Accuracy: 55
Figure 4:10 Tools Consistency 55
Figure 4:11 Vulnerabilities vs. Tools 56
Figure 4:12 Vulnerabilities VS Scanning Tools 57
Figure 4:13 Vulnerabilities detected in Zero WebApp 57
Figure 4:14Vulnerabilities discovered in phpBB 58
Figure 4:15 The figure below shows wapitis capability to discover vulnerabilities rated as high 58
Figure 4:16 Weighted Average for all the WVS 59

List of Tables

Table 2:1Severity of web vulnerabilities 14

Table 3:1 vulnerability scanners and web applications... 33

Table 4:1Summary of vulnerabilities detected by the web scanning tools 53

Table 4:2 Vulnerabilities discovered by Vega 53

Table 4:3Vulnerabilities discovered by W3AF 54

Table 4:4Vulnerabilities discovered by Websecurify 54

Table 4:5Vulnerabilities discovered by Arachni 54

Table 4:6Vulnerabilities discovered by Wapiti 54

Table 4:7Vulnerabilities discovered by Zed Attack Proxy (ZAP) 54

Table 4:8Vulnerabilities discovered by CK AppScan (Simulation)... 54

Table 4:10 Time taken to scan various Applications 56

Table 4:11 Weighted Average 59

Table 4:12 Wapiti’s Weighted Average 59

Table 5:1Mapping research objectives to the methodology.. 63

List of Abbreviations

CSRF- Cross-Site Request Forgery

HTTPS- (HTTP Secure) is a protocol for secure communication over a computer network

HTTP- Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems.

IT – Information Technology

ICT- Information Communication Technology

LDAP – Lightweight Directory access protocol

LFI – local file inclusion

OWASP Open Web Application Security Project (OWASP), an international, non-profit organization whose goal is to improve software security across the globe

RFI- Remote files inclusion

RXSS – Reflected cross-site scripting

SQL- Structured Query Language

SQL I - structured query language injection

SSI- server side injection

WASSEC - Web Application Security Scanner Evaluation Criteria - is a set of guidelines to evaluate web application scanners on their ability.

WAVSEP- Web Application Vulnerability Scanner Evaluation Project. A vulnerable web application designed to help to assess the features, quality and accuracy of web application vulnerability scanners. This evaluation platform contains a collection of vulnerable web pages that can be used to test the various properties of web application scanners.

W3AF- Web application attack and audit framework

WVS- web vulnerability scanner

XML- stands for Extensible Mark-up Language. XML was designed to describe data

XST – Cross-sitetracing

XSS- Cross Site Scripting

ZAP – Zed attack proxy

Definition of terms

Crawling is described as the action taken by a program as it browses from page to page on a website. The crawler component will visit a starting web page and parse the provided links, crawling to all pages in an application.

Fuzzing component is the element in a web scanner that handles input that exposes vulnerability; it generates its data.

Port - is a logical channel of communication that is associated with a process or a service.

Vulnerability is a loophole or weakness in the application, which can be a design flaw or an implementation bug that allows an attacker to cause harm to the stakeholders of an application

CHAPTER 1 : INTRODUCTION

1.1 Background Information

The number and importance of web applications have increased rapidly over the years (Jovanovic, Kruegel and Kirda, 2014). many organizations have embraced these technologies to explore new business opportunities and some companies have been forced to adopt the electronic commerce by their customers or competitors.

Web applications have gained popularity and have become part of our daily lives interaction. At the same time, new web application vulnerabilities emerge every now and then and endanger the use of the web-based applications. Therefore, manual code inspection or security audits must be done by highly trained experts who are labour-intensive, expensive, and prone to errors. (Kals et al, 2014).

Today employees are constantly responding to requests from both inside and outside the organization's corporate network. While this has enormous benefits, it also present a challenge since it provides a weak access point that can be exploited by hackers to gain unauthorized access company information.

While the internet infrastructure is developed by very experienced experts with security flaws and solutions at the back of their mind, some of the web applications are developed by novice programmers who have little or no knowledge of about web application security. For this reason,they produce avulnerable web application that can be hacked exposing the organization's confidential information.

Many organizations using web-based applications, experience one or more forms of security breaches. For instance, hackers may gain access to company data, unauthorized programs steal customer's login credentials and send them to cyber criminals, viruses may also be used to execute illegal transactions as well as other fraudulent activities. Hackers are also known to deface company's website and deny users access to services. Whereas some companies shy away from publicizing such information to avoid negative reputation, the news find their way to the public domain in one way or another. There is a need to identify the security lapse in various organizations and come up with ways of minimizing cybercrime.

Doupe et al (2010) presents an evaluation of eleven black-box web vulnerability scanners composed of different types of vulnerabilities with different challenges to the crawling capabilities of the tools. The results of the evaluation show that crawling is a task that is as critical and challenging to the overall ability to detect vulnerabilities as the vulnerability detection techniques themselves, and that many classes of vulnerabilities are completely overlooked by these tools, and thus research is required to improve the automated detection of these flaws.

Cyber-Attacks have increased over the years, in the recent times. Some of the incidences reported includeWalmart where hackers broke into the computer system used by the development team to steal information from cash register. Home Depot cyber-attack in which cyber criminals stole information of about 60million card numbers. (Snyder, 2014). Target (TGT) breach with hackers have got access to personal information of about 110 million customers. Apple attack in 2014 to *i*Cloud account stealing personal information including renown celebrities and finally the Sony pictures attack where cybercriminal got access to the unreleased movies leading to cancelation of the film (Granville 2015)

Closer home here in Kenya, police arrested 77 Chinese suspected cyber criminals and recovered equipment's that is useful for infiltrating bank accounts, government servers, Kenya's M-Pesa mobile banking system and ATMs. The NIC Bank Kenya hacking in which cyber-criminal accessed the bank database and threatened to expose the data if they did not receive a ransom. Overnight hacking of 103 Government of Kenya websites (Sharma,2012) and the hacking of Kenya Defence Forces (KDF) social media accounts. For this reason, existing vulnerabilities in any web application should be fixed without delay.

Currently, there are many different types of web application vulnerabilities (Stuttard & pinto, 2011).These include and not limited to: - SQL injection, X-path injection, cross-site scripting, cross site tracing, cross-site request forgery, local file inclusion, remote file inclusion, HTTP response splitting, command injection, server-side injection LDAP injection, buffer overflow, and session hijacking. All these can be exploited by cyber criminals to compromise the security of an enterprise web application.

The development of a web-based application is labor intensive exercise, this coupled with the limited time given to developers to deliver the applications and within a limited budget. Developers find difficulties while developing secure and high-quality applications within

constrain of time and budget. In most cases, they end up compromising the security of the web application under development.

Analyzing web based application manually is one of the ways to find vulnerabilities, but it quickly becomes a slow, tedious process especially considering the sheer number of websites, the complexity and size of modern websites. For this reasons, there is a shift towards the use of automated tools to test the vulnerabilities of the web-based applications (Doupe, Cova & Vigna, 2010). These tools are both commercial and open source. Some of the open source web scanners are not updated on a regular basis and therefore, they do not have the capacity to detect all the existing vulnerabilities. This study focuses on identifying the open source tools, benchmarking them and analyzing the algorithms to come up with hybrid and improved algorithms that can be integrated into sophisticated tools with a user-friendly interface to allow them to be used by novice web application programmers.

1.2 Statement of the Problem

With advancement in web technologies and shift from traditional desktop application to web-based solutions, the popularity of web-based applications has grown tremendously. Today, the web-based applications are used in security-critical environments, such as medical, financial and military systems (Stuttard & Pinto, 2011). Although the internet infrastructure is developed by experienced programmers with security concerns in their mind, some of the web applications are engineered by less experienced consulting programmers with little or no knowledge about security. This exposes the web application to various vulnerabilities and provides avenues for cyber criminals to gain unauthorized access to confidential information.

In one of the recent studies by the Ponemon Institute, they found out that that 45% of breaches exceed $500,000 in losses. In the largest of incidents, many Fortune-listed companies have given shareholder guidance that the losses would vary from a few dollars to millions of dollars. For this reason, it is prudent to do something in a proactive manner to avert or reduce harm before a cyber-attack.

Past studies have concentrated in benchmarking open source web vulnerability scanners to find out their capabilities and limitations. There is need to analyze different algorithms, identify their strength and weaknesses, with an aim of coming up with a hybrid algorithm that is superior, performs faster and can work on more inputs and in a complex situation. In this

proposal, the researcher seeks to benchmarks different web vulnerability scanners, identify these tools and suggest an improved algorithm that can be adopted while developing tools.

1.3 Main Objective

The primary objective is to perform a comparative study of open source vulnerability testing tools, study their algorithm for these tools and propose an improved hybrid algorithm.

1.3.1 Specific Objective

The specific objectives are:-

i. To identify different open source vulnerabilities scanning tools for web application
ii. To analyse the tools against set metrics
iii. To study algorithm for these tools
iv. To propose an improved hybrid algorithm
v. To test and validate the hybrid algorithm

1.3.2 Research Questions

i. What are the different open source vulnerability tools available for scanning web applications?
ii. What metrics can be used to comparatively analyse web application scanning tools?
iii. What are the algorithms employed by these tools?
iv. What improved algorithm can the web application scanning tools use?
v. How can we test the hybrid algorithm?
vi. How can we validate the hybrid algorithm?

1.4 Justification of the Study

This project produced an improved hybrid algorithm. The finding of the study contributes to the body of knowledge by providing the necessary literature for researchers and academicians interested in the study web application security and more specifically the vulnerability that exists and the capabilities of open source vulnerability testing tool that exist.

Further, the study forms a basis for further studies to be conducted on improved vulnerability testing tool algorithms. To the web application developers, it will act as an eye opener on the open source tools that are at their disposal, their capabilities and the impacts in applying the tools to come up with secure web applications thereby safeguarding the organization's information systems and networks. For application testers, the research will help them to use

more accurate tools. The web application can effect such improvements. The consumers will get secure web application and, therefore, get good return for their investments in addition to secure web applications.

1.4.1 Hybrid Algorithm Justification

The current algorithms have weaknesses, for this reason they take long period of time to scan web applications and the results are not 100% accurate. Below find the reasons why the hybrid algorithm was created.

i. To increase the vulnerabilities detection accuracy
ii. To scan web vulnerabilities within acceptable time frame
iii. To increase efficiency in the scanning web applications

1.4.2 Challenges with Existing Algorithms

The algorithms employed by the tools have some shortcomings since they are not able to discover all the vulnerabilities that exist in web based applications. They are sophisticated and produce inaccurate results by reporting vulnerabilities that do not exist.

1.5 Limitations of the Study

There are limitations to the use of web vulnerability scanners, as these tools are not a 100% accurate. Scanning a web application using some such tools does not guarantee its safety and security. Web application scanners are weak at finding multiple vulnerabilities such as encryption flaws and information disclosure flaws. In addition, the random data generated by the fuzzing component during the scanning process may not discover all the vulnerabilities. (Shelly, 2010)

One of the problems with vulnerability assessment tools is that they often report vulnerabilities that do not exist. These bogus findings are called "false positives." (Antunes & Vieira, 2012).

In conclusion, there is no single solution. Open source web vulnerability scanners play a critical role in discovering loopholes; however, they do not offer a complete solution for vulnerability detection. As a matter of fact there is no single tool whether open source or proprietary that can discover all vulnerabilities.

CHAPTER 2 : LITERATURE REVIEW

2.0 Introduction

The use of computers, tablets and smart phones has greatly increased over the past few years, as noted by Stuttard & Pinto (2011) web applications have been developed to perform practically every useful function you could implement online. These include-Online Shopping, Social Networking, Gambling & Online casino, Banking, Web search, Auctions, Webmail, and Interactive web pages among others. In a report published by Whitehat "86% of all websites tested by Whitehat Sentinel had at least one serious vulnerability, and most of the time, far more than one – 56% to be precise. (Whitehat, 2015)

According to Shema (2011), many organizations rely upon customized web applications to implement business processes. These may include full-blown applications, or consist of modules such as online, login pages shopping carts, and other kinds of dynamic content. Some of these software applications in your network could be developed in-house. In addition, some may be legacy websites with no designated ownership or support. Manually analyzing all of these for loopholes and prioritizing their importance for remediation can be a daunting task without organizing efforts and using automated tools to improve accuracy and efficiency.

Employees are continuously responding to requests from both inside and outside the organization's corporate network using gadgets such as tablets, smartphones or laptops. While this has enormous benefits, the negative drawback is the fact that hackers may take advantage of connectivity to gain unauthorized access to vital company information. For this reason, it is imperative for any company to ensure that they protect their web applications and reduce the possibility of a security breach to their electronic system. Testing the weakness of web applications with automated penetration testing tools produces relatively quick results. Currently, there are many such tools, both commercial and open source.

Yu et al (2011) noted that web application security vulnerabilities are on the increase. These vulnerabilities allow attackers to perform undesirable actions that range from gaining unauthorized account access, to obtaining confidential data such as credit card numbers and in some extreme cases, they threaten to reveal the identities of intelligence personnel.

In a one of the recent article published in the "The New York Times", Kevin Granville (2015) reported that, In November 2014, a huge attack that wiped clean several internal data centres. It out rightly led to the cancellation of the theatrical release of "The Interview," a film about the fictional assassination of Kim Jong-un, the North Korean leader. Contracts, film budgets, salary lists, entire films and Social Security numbers were stolen, including - to the dismay of top executives leaked emails that included criticisms of one of the top celebrities Angelina Jolie and undesirable remarks about the USA President. President Obama administration said that, North Korea was behind the attack.

The WikiLeaks saga demonstrates the consequences of information leakage. The above cyber-attack examples highlight some of the common incidences. The truth is most of the cyber-attacks are never reported. In a report published on February (2015) by White Hat Security, the authors notes that "unfortunately and unsurprisingly, website security breaches have become an everyday occurrence. As a matter fact, hacked websites and web applications have become so frequent that only gross data breaches get enough attention to make headlines. The rest get to suffer quietly away from the public domain."

2.2 Classification of Web Scanning Tools

Web scanning tools can be classified into three categories, namely white-box, black-box and grey-box

2.2.1 White-box

White box web scanner checks the source code of the any web application to detect vulnerabilities. Through the analysis of the code, the white box tool can be able to identify vulnerabilities found in the application. The main advantage of using white box tools is the fact that they are able to identify more weaknesses, however, they are known to report vulnerabilities that do not exist. This is simply because the analysis of the code may overestimate the program paths that the program can execute. (Mirjalili, Nowroozi, & Alidoosti 2014)

Limitations of white-box tools

The main drawback of using the white box tools is that they are application specific, if a white box tool is meant for PHP, it will not work with other applications.

2.2.2 Black-box

Black box web scanners do not check the source code; instead they interact with the application just like a user using a web browser. They are comprised of three components

- Crawling component which is responsible for browsing from one page to the other on the web application and parsing the provided links crawling to all the pages in an application.
- Fuzzing is a component mutates or generates inputs either structurally or randomly and inserts it into the web application to discover vulnerabilities. The quality of any fuzzing component is determined by the number of inputs that are used to find vulnerabilities.
- Analyser- checks the results of the attacker and determines which ones were successful or not. (Park , 2015)

Limitations of black-box tools

There is no guarantee that all the vulnerabilities in any given web application will be reported. In addition, tools that embrace this approach are known to report false negatives.

2.2.3 Grey-box

Grey box is a hybrid approach that combines both the black box techniques and white box techniques. The main objective is to generate all the vulnerabilities that can be detected by the white box method and test them using the black box approach. If the test is successful, then, it will be reported by the tool.

Grey-box takes advantage of black box approach; however, it also inherits the weaknesses of black box tools. For this reason, these tools are not popular.

2.3 Types of Web Application Vulnerabilities

As described by Stuttard and Pinto (2011), the number and the importance of Web applications have increased rapidly over the last few years. These vulnerabilities include - Cross-Site Scripting (XSS), Overflow Buffer Overflow, Server side injection (SSI), Command Injection, HTTP Response Splitting, Remote file inclusion, Local file inclusion, X-Path injection, Cross-site tracing and Cross-Site Request Forgery among others.

The number and impact of security weaknesses in such applications has grown as well. As illustrated on the figure below.

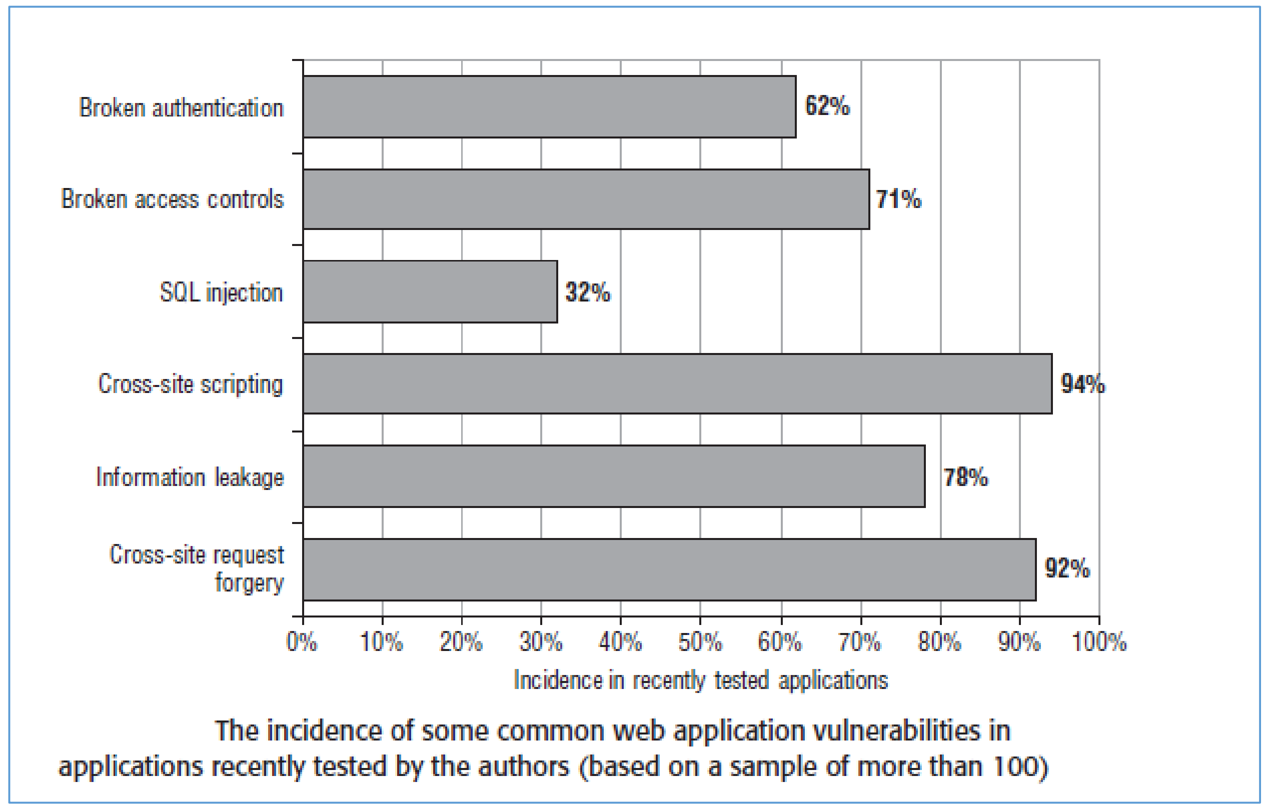


Figure 2:1 Common web vulnerabilities

Source: The web applications hacker's handbook 2nd Edition. Stuttard and Pinto (2011)

- Broken authentication (62%)
- Broken access controls (71 %)
- SQL injection (32%)
- Cross-site scripting (94%)
- Information leakage (78%)
- Cross-site request forgery (92%)

2.3.1 Cross-Site Scripting (XSS)

One of the main goals of XSS attacks is to steal the credentials (using a cookie) of an authenticated user using malicious JavaScript code. Due to the sandbox model, JavaScript has access only to cookies that belong to the website from which the malicious JavaScript originated. All XSS attacks circumvent the sandbox model by injecting malicious JavaScript into the output of susceptible applications that have access to the wanted cookies. The sandbox model involves the isolation of computing environment used by software testers or developers to test new programming code.

Kalman (2014) describes XSS as sanitization failure, whereby the attacker provides a web application JavaScript tags on input. When the link is returned to the user, un-sanitized, the user internet browser will execute it. This can be as simple as persuading a user to click on a link. Once the link has been clicked, the script will execute and perform undesirable actions

Prevention

XSS attacks can be prevented by designing web applications in a way that they don't return HTML tags to the client. Or by using regular expressions to strip away HTML tags.

2.3.2 SQL Injection

Dougherty, (2012) argues that SQL injection vulnerabilities are caused by software applications that accept data from an un-trusted source such as the internet. Howard, LeBlanc and Viega (2010) argues that an invalidated user input is used to construct an SQL algorithm which is then executed by the web server, for instance, the query used by a user's login. Data from un-rusted sources can lead to failure of validation and sanitation and sub which can be used to dynamically construct an SQL query to the database backing that application. However, argue that in the "SELECT *FROM users WHERE username='entered username' AND password='entered password'." If an attacker enters the string x' OR '1'='1 in both the username and the password. The query converts to "SELECT * FROM users WHERE username='x' OR '1'='1' AND password='x' OR '1'='1' " and because '1' is always equal to '1', this query is true for all records in the database. Khoury (2011)

2.3.3 X-Path Injection

As noted by Van der Loo & Poll (2011) X-Path injection is pretty much similar to SQL injection. The main difference between these two vulnerabilities is that SQL injection takes place in SQL database, whereas X-Path injection occurs in an XML, since X-Path is query language for XML data. Just like SQL injection the attack is based on sending malformed information to the web applications. This way the attacker can discover how the XML data is structured or access data that he has no authority to access.

2.3.4 Cross-Site Tracing

Cross-site tracing often abbreviated as XST as an attack that abuses the HTTP TRACE function. This function can be used to test web applications as the web server replies the same data that is sent to it via the TRACE command. An attacker can trick the web

application in sending its normal headers via the TRACE command. This allows the attacker to be able to read information in the header such as a cookie. (Shelly, 2010)

There are three types of XSS namely;

- Stored XSS – whereby the attackers code is stored on the web server
- Reflected XSS - whereby the attackers code is added to a link in the web application
- DOM based XSS- whereby the attacker's code is not injected in the web application but instead uses existing java scripts on the web page to write scripts.

2.3.5 Cross-Site Request Forgery

Cross-Site Request Forgery, is an attack where a malicious script tricks a user's browser into loading a request that performs an action on a web application that user is currently authenticated to. For example an attacker might post the following HTML on a website or send it in an HTML email *<img src="http://www.bank.com/transfer_money?amount=10000&target_account=12345">.* If the user is authenticated at his bank website (at http://www.bank.com) when this link is loaded it would transfer 10000 from the user's account to bank account number 12345. (Howard, LeBlanc and Viega, 2010)

CSRF is the issuance of requests by 3rd party websites to a target site, say your bank using your internet browser and cookies while your session is still active. For instance, if you are logged on your bank's websites on one of the tabs, another tab in the browser can misuse the credentials on behalf of the attacker and do something the attacker instructs it to do. (Kalman, 2014)

2.3.6 Local File Inclusion

It is also known as path traversal or directory traversal. In this types of vulnerability, a file on the same server as the one where the web application is running is included on the page. For example a web application with the URL *http://www.example.com/index.php?file=some_file.txt* by manipulating the file parameter the attacker might be able to load a file that he should not be able to see.

As noted by Nagpal, Chauhan & Singh, (2015) local file inclusion vulnerability occurs when a web page is not properly sanitized and allows directory traversal characters such as dot or dash to be injected. This kind of attack can lead to:

- Code execution on a web server

- DOS denial of service attacks
- Disclosure of confidential information
- Code execution on the client side.

Local file inclusion occurs due to the use of user input without proper validation.

2.3.7 Remote File Inclusion

Van der Loo, (2011) argues that remote file inclusion is the same as local file inclusion, except for that the file that is included is a file from a different server than the one the web application runs on. An example of this vulnerability is the same as for local file inclusion. However, instead of changing the file name parameter to a local file, the attacker enters a path to a remote file. Testing for this vulnerability is also similar to local file inclusion. However, instead of a path to a local file a path to a remote file is used.

2.3.8 HTTP Response Splitting

As noted by Kalman (2014). HTTP response splitting is an attack whereby the attacker can control data that is used in a HTTP response header and appends a new line in this data. If the web application uses a redirect using GET parameter. The attacker can insert a new line to the value of GET parameter and add customized headers. This type of attack is experienced when, data is provided to a web application via untrusted sources such as HTTP or when the data is included in a HTTP response header without proper checking malicious characters.

For this attack to be successful, the application must allow carriage return or line feed in the header. The underlying platform must also be prone to injection of such characters (%Od or %Oa)

2.3.9 Command Injection

Command injection as the ability of the attacker to send a command to a web server from a remote location. The attacker will simply specify the remote server IP address followed by the desired commands. This command will eventually be executed and perform the desired action.

Command injection is possible when an application passes unsafe user supplied data forms, HTTP headers or cookies. The criminals provide the operating system with commands

executed with privileges of a vulnerable application. These kinds of attacks are possible largely due to insufficient input validation on the web pages. If the web developer or programmer puts the necessary data validation measures, command injection attacks can be reduced significantly.

2.3.10 Server Side Injection (SSI)

SSI attack involves, the attacker enters SSI directives on the web server. These commands are executed directly on the web server and making undesirable changes to the web application. SSI attack allows web applications by injecting scripts in HTML web pages or executing arbitrary codes remotely. An attack will be successful if the web application allows the execution of SSI code execution without proper validation. For instance, one of the known vulnerabilities in SSI exist in IIS version 4 and 5, which allows cyber criminals to obtain system privileges via buffer overflow failure in a dll file (ssinc.dll). By creating a malicious webpage, the criminals perform undesirable actions or perform fraud. (Mirjalili, Nowroozi, & Alidoosti 2014)

2.3.11 Buffer Overflow

Van der Loo & Poll (2011) defines buffer overflow as an attempt to store more data on the provided buffer than the buffer can store. Testing buffer overflow is straightforward; the tool simply inputs random data to see if errors will pop up by trying to store more data. A buffer overflow may be triggered by input variables that are designed to execute code. This results in unpredictable program behavior such as incorrect results, crash, and security breach or memory access errors among others, by overwriting the local variables near the buffer in the memory changing the behavior of the program that may be used by an attacker.

Another way cyber criminals could benefit from the buffer overflow is by overwriting the return address in the stack frame after the function returns the values; they are sent to the return address specified by the criminal.

Web scanning tools can be grouped into the categories as illustrated by the table below (Tripathi et al, 2011)

Table 2:1Severity of web vulnerabilities

Severity	Vulnerability
High	SQL Injection Blind SQL Cross site Scripting & Reflected cross site scripting Cross site reference forgery &Cross site tracing Command line injection &Server side injection
Medium	Local file inclusion Remote file inclusion Buffer overflow LDAP
Low	Xpath Presence of a backup file
Informational	Email disclosure Blank content

Key:

High Severity: a vulnerability is considered to be high, if the consequences of such a vulnerability are dire. For instance if the attacker is able to get sensitive information or take over the operations of a web application.

Medium Severity: A vulnerability is classified as medium if it fails to be categorized as high or low.

Low Severity: This type of vulnerability does not produce valuable information or control over a web application but it provides a potential attacker with useful information that may useful in exploiting other vulnerabilities.

Informational severity- this kind of vulnerability is considered to be inconsequential

2.4 Web application Vulnerabilities Testing Tools

Web Application Vulnerability Scanners are the automated programs that checks web applications for known security loopholes such as cross-site scripting, SQL injection, command execution, directory traversal and insecure server configuration (Stuttard & Pinto. 2011). A large number of both commercial and open source tools are available, and all these tools have their own strengths and weaknesses

The tools crawl a web application and identify application layer vulnerabilities either by inspecting them for suspicious attributes or manipulating HTTP messages. For very complex cases (projects.webappsec.org) the tools mimic external attacks from hackers, provide effective methods for detecting a range of vulnerabilities. They may also configure and test peripheral defences such as web application firewalls (Bau et al, 2010) Most of the penetration testing tools use a technique that is called fuzz testing. Fuzzing also known as fault injection is a highly automated testing technique that covers numerous boundary cases using invalid data as application input to ensure better the absence of exploitable vulnerabilities.

The main advantage of using penetration testing tools it that it is a relatively fast and easy way to detect certain security vulnerabilities. They further noted that unlike traditional black box testing, in which an ethical hacker tries to attack the web application, penetration testing tools, can be used by a person with little or no knowledge about security. Only the analysis of the result has to be done by a person with knowledge about security. (Alssir & Ahmed, 2012)

However, despite the advantages, penetration testing tools have limitations. Penetration testing tools cannot find all vulnerabilities. They do a poor job at finding vulnerabilities like information disclosure and encryption flaws, access control flaws, identification of hardcoded back-doors or multi-vector attack. Further, the use of random data also fails to uncover vulnerabilities unless the fuzzy process is repeated several times. Penetration testing tools do not have any specific goal to work toward to, and, therefore, they try to attack any possible risk. (Austin & Williams, 2011)

2.5 Algorithms

An algorithm can be defined as a formula or procedure for solving a problem. It comprises of a stepwise instructions. Algorithms can be used to do automated reasoning, perform calculation, and data processing, and automated reasoning.

Algorithms that employ similar problem-solving techniques are grouped together. The classification may not be exhaustive but the main aim is to highlight different ways of attacking problems.

Classes of Algorithms

i. Brute force algorithms
ii. Greedy algorithms
iii. Divide and conquer
iv. Recursive algorithms
v. Dynamic programming algorithms
vi. Backtracking algorithms
vii. Randomized algorithms
viii. Branch and bound algorithms

Brute Force

Brute force is also known as exhaustive search. In this algorithm every aspect of a possible solution is considered until the optimal solution is found. The algorithms stop to execute when a solution is found. Brute force uses permutations when doing searches; it is one of the easiest techniques to use. (Leiserson, 2012)

Benefits of Brute force

It has the ability to arrive at an optimal solution especially for small population

Weakness of brute force

when the number you have a large population and the algorithms will execute for a long period before a solution is found. However, the weakness can be addressed by using heuristics or optimization

Heuristics is using the so called rule of thumb to assist you to make a decision on the possibilities to check first

Optimization – provides a quick way to eliminate some possibilities without the need to explore them fully.

Greedy Algorithms

Greedy Algorithms are, short-sighted, simple to use and straightforward. They are known to find the best solution in a given case. The reason this algorithm is considered as short sighted is simply because, they look for the best solution now without taking into consideration about the future. They are very easy to implement or use and sometimes produce acceptable results. Greedy Algorithms sometimes produce a good solution but not the best solution.

Weakness of Greedy Algorithms

Sometimes they produce misleading solutions rather than providing the best solution.

Divide and Conquer

Divide and conquer are considered to be very efficient. The problem at hand is divided into smaller units, known as sub-problems. The sub-problems are solved in a recursive manner and eventually the solutions are combined to form a solution to the original issue. In short the main problem is decreased significantly. A good example is a binary search

Weakness of Divide and Conquer

Sub problems may overlap and thus produce undesirable or inaccurate results.

Recursive Algorithm

A recursive algorithm comprises of two main parts:

i. A recursive section that defines the solution with a smaller instance of the problem.
ii. A non-recursive section that contains the limiting condition. A good example of this algorithm is quick sort and merge sort

Merge sort, involves the division of an array into two equal parts and recursively sorting the two parts, and they merge them after they have been sorted. The process of division stops when the elements cannot be divided any more.

Dynamic Programming Algorithms

Dynamic Programming algorithms are known to keep past results in the memory and use them to arrive at new results. Divide and conquer employs a top-down approach to arrive at a solution. While Dynamic Programming algorithms starts with the largest part a problem and continually subdivide it further until the base case reached. In dynamic programming starts with the most simple case and work in a systematic manner to get values needed. Solutions derived from sub problems are stored in the memory and re-used to give a solution. For this reason, it has been considered to be one of the best approaches to the best solution in cases where there are multiple solutions.

Backtracking Algorithms

Backtracking algorithms are known to employ a recursive search technique. It will simply search to check whether a solution has ever been found. If the solution exists, it will be returned. If no solution is found, it will return a failure. In very simple terms, it is based on an extensive recursive search.

Randomized Algorithms

Randomized algorithms employ a random number in the process of doing computations to arrive at a solution. For instance, using a random prime number to one of the possible divisors when using an array.

Branch and bound algorithms

Branch and bound algorithms are mostly used for optimization. As the algorithm executes, it forms a tree of sub problems. The original issue is considered as the root and sub-problems form the branches. At every branch or node, a feasible solution is considered for the sub problem. All the sub problems are solved and form part of the ultimate solution. The tree continues to grow until a base case is reached.

Weakness of Branch and bound algorithms

In some special cases where the problem or issue at hand is very complex, the tree will be comprised of many nodes or branches, which will take the time to compute the best solution.

2.6 Hybrid Algorithms

Clearly, the abundance of vulnerabilities in web-based applications and their increasing popularity make a strong case for the need for improved techniques and tools for their security assessment. New testing tools and methodologies have been proposed that aim at identifying and removing flaws by exercising an instance of an application with unexpected, random, or faulty input. These approaches are appealing since testing can be performed even on applications whose source code is not available, in addition to being independent of any application and therefore being reusable on different applications and finally being characterized by the absence of false positives, i.e. flaws found through testing correspond to actual bugs in the application code. However, they lack completeness in that they cannot be able to detect all the vulnerabilities in and existing web application. (Austin & Williams, 2011)

According to Doupe et al (2010) most of open source web vulnerability scanners (WVS),have a lot of limitation in detecting the vulnerabilities and further that they lack better support for well known, pervasive technologies such as flash and JavaScript. It is recommended that there is a need for more complex algorithms to track the state of the application under test and execute "deep “crawling.

The idea of a hybrid algorithm is based on the concept of mixing two or more algorithms. This is inspired by the assumption that new algorithm will perform better than the parent algorithms independently. The hybrid algorithm utilizes the strengths of the individual algorithms such that the resulting new algorithm performs better, its faster in giving results and can handle bigger inputs with the higher level of complication.

2.7 Web Based Applications

OWASP WebGoat – This is a web application that consists of at least 69 known vulnerabilities divided into various categories. The main purpose of WebGoat is to act as a platform for people to learn how different vulnerabilities can be detected. The core benefit of this application is the fact that it is loaded with a tomcat webserver which is installed on the testing computer. This guarantees the same results regardless of the PC it is installed. In addition since the application runs locally, the results are not subject to internet connection speeds.

The only drawback of using this tool is that, it's very resource intensive, takes a long time to setup and authentication uses a username and password which employs a cookie to manage the sessions. This indeed makes it difficult to run some of the web scanning tools.

Mutillidae- this is another fully fledged web application loaded with a webserver several webpages that consist of various vulnerabilities. It runs on a web browser and consists of categorized OWASP 2013 and 2010 Top vulnerabilities. During the installation, XAMP, Apache and MySQL must be installed for it to run properly. This particular tool was chosen since it consists of well-known OWASP vulnerabilities. The application comprises of PHP scripts and codes that have been engineered to produce the desired results. The only drawback of using this application is the fact that it results are subject to the customized configuration that can be changed while setting up the webserver.

The main advantage of using this application is the fact that all the vulnerabilities in this application are well-known and it is regularly updated on a regular basis to include new vulnerabilities.

Zero.webappsecurity.com

This application was invented by Hewlett Packard popularly known as HP. It is a web application that has been in existence for a while and has been used to test various

applications. It runs from a server on the internet. This means you are not required to install a web server, and the results are not subjected to the configurations made on the client side.

The main drawback with this application is the fact that the results may be affected by your internet speeds. In addition, the number of vulnerabilities is not explicitly stated. However, this application was chosen to compare it with other application.

phpBB

This is one of the popular forums for software used by many people; it offers a real application with vulnerabilities that exist. This simply means that the loops holes have not been designed on purpose. The main advantage of this application is the fact that unlike the other applications whose vulnerabilities are implemented by design, the vulnerabilities contained here are real. Indeed, this is the ultimate challenge and to see if the tools have the capacity to detect vulnerabilities.

The main disadvantage of this tool is the fact that all the vulnerabilities are not known, although there are hints on the application, it is not explicitly stated where the vulnerabilities are located. In addition, for you to run this application, you are required to install a webserver. This means that the results collected may be positively or negatively affected by the customized configurations on the server.

2.8 Web Vulnerability Scanning Tools

The following open source web scanners were used by the researcher in this study.

- Wapiti
- Websecurify
- Arachni
- W3af
- Zed Attack Proxy
- Vega

i. Wapiti

This is an open source web scanning tool that allows you to perform a security audit of a web application. It employs a black box approach; this simply means it does not study the

code. Instead, it checks all the web pages and identifies forms found on a webpage where it can insert or inject data. Form its website, they have indicated that it can detect,

- Local and remote file inclusions
- X-Path injections (PHP, ASP, JSP and SQL injections)
- XSS (Cross-site scripting)
- CRLF
- XXS
- Check presence of backups that can disclose confidential or sensitive information

Wapiti is known to support POST HTTP and GET techniques of web application attack

General features include

- Ability to provide comprehensive reports after completion of a scan
- Authentication using various methods such as NTLM, Kerberos or basic
- Ability to define or limit the scope of the scan to a folder, webpage or a domain
- Updated to understand recently release web development technologies such as HTML5 among others.

Wapiti is accessible via CLI (Command line tool)

ii. Arachni

Arachni is a full blown web scanning program that runs on a Linux platform; it is useful when evaluating the security of online web applications. Unlike other utilities in this category, arachnid is known to consider the dynamic nature of web applications and apply the complexity required to detect loopholes in such applications. It has been designed to operate within a web browser, for this reason, it can detect client side code and make use of advance web development technologies such a HTML 5, java scripts and Ajax among others.

It is versatile and can cover a wide range of scanners such as ruby libraries, and multi-user or multi scan web platform.

Arachni can be deployed via

i. Ruby library – for scripted or highly customised scans
ii. CLI- command line interface
iii. Web interface – for multi scan, multi user or even multi dispatcher management
iv. Distributed system using agent or load balancing.

From Arachni's website, they indicated that it can detect XSS, Code injections, CSRF, File inclusion, path traversal SQL Injection and non-SQL injections

iii. Websecurify

Websecurify has a graphical user interface that makes it very easy to use. Once you start a test, it will be performed automatically and there are very few options for customisation. It is indeed a powerful tool that provides automatic or manual scanning approaches. Once vulnerability has been detected, it is presented at the end of the scanning process. The report shows all the vulnerabilities detected as well as the recommended solutions.

iv. W3AF (Web application attack and audit framework)

This is an open source web scanner that has been developed using Python. The main aim of this project is to develop a framework to assist users to detect all types of web vulnerabilities. Once a scan has been completed, the results are displayed on a HTML file. It is comprised of a GUI interface and employs plugins to perform the attacks.

v. Zed Attack Proxy

This is an open source web scanning tools that uses a GUI interface; it is meant for new developers as well as experienced programmers. Simply input the URL of the application you would like to scan and wait for the scan to be completed and review the report.

vi. Vega

Vega is an opens source WVS that uses a GUI interface. It classifies the scan alert summary into four categories namely; High, Medium, Low or Info. A report with each of the above-mentioned categories groups consist of all vulnerabilities found, and the quantity is accessible once the scan is completed.

2.9 Algorithms Used by Existing Tools

Wapiti's Algorithm.

Wapiti employs the black-box approach; this simply means that it does not check the source code of the application. Instead, it works by scanning the web pages of the application and extracting forms and links or scripts that generate errors. It is capable of detecting XSS, LDAP, Command line injections, Database injection (ASP, SQL, and Xpath), LFI and RFI, CRLF, Search for backup copies. Below find sample algorithm extract. (The rest of the algorithm can be found on the appendix)

Figure 2:2 Attack module algorithm.

```
> define member function: attack(self, http_resources, forms)
  START
    1. Is self.doGET true?
        if NO, then goto 4
        else, then goto 2
    2. foreach http_res in http_resources
        2.1 Initialize url = http_res.url
        2.2 Is self.verbose equal to 1?
            if YES, then write log, _("+ attackGET {url}")
            else, then goto 2.3
        2.3 Call self.attackGET(http_res)
        2.4 Is socket.error occured?
            if YES, then write log, _('error: {repr(str(se[0]))} while attacking {url}'
            else, then goto 2.5
        2.5 Is request timeout?
            if YES, then write log, _('error: timeout while attacking {url}'
            else, goto next loop
       next loop http_res
    3. go to END
    4. Is self.doPOST true?
        if NO, then go to END
        else, then go to 5
    5. foreach form in forms
        5.1 Is self.verbose equal to 1?
            if YES, write log, _("+ attackPOST {form.url} from {form.referer}"
            else, then goto 5.2
        5.2 Call self.attackPOST(form)
        5.3 Is socket.error occured?
            if YES, then write log, _('error: {repr(str(se[0]))} while attacking {url}')
            else, then goto 5.4
        5.4 Is request timeout?
            if YES, then write log, _('error: timeout while attacking {url}'
```

Figure 2:3 An extract from SQL detection module

```
> define member function: attackPOST(self, form)
  START
  1. Initialize variables:
       payload = "\xbf'\"(", filename_payload = "'\"(", and err = ""
       get_params = form.get_params
       post_params = form.post_params
       file_params = form.file_params
       referer = form.referer
  2. for params_list in [get_params, post_params, file_params]
     for i in xrange(len(params_list))
    2.1 Backup saved_value = params_list[i][1]
    2.2 Is params_list is file_params?
      if YES, then put params_list[i][1] = ["__SQL__", params_list[i][1][1]]
      else, then put params_list[i][1] = "__SQL__"
    2.3 Initialize param_name = self.HTTP.quote(params_list[i][0]), and
        attack_pattern = HTTP.HTTPResource(form.path,
           method=form.method, get_params=get_params, post_params=post_params,
           file_params=file_params)
    2.4 Is attack_pattern not in self.attackedPOST?
      if YES, then goto 2.5, else goto 2.11
    2.5 Append attack_pattern to self.attackedPOST
    2.6 Is params_list is file_params?
      if YES, then put params_list[i][1][0] = filename_payload
      else, then put params_list[i][1] = payload
    2.8 Initialize evil_req = HTTP.HTTPResource(form.path,
```

Figure 2:4 Algorithm used by Vega – sample

```
// code-injection.js

1. Declare module(object) and initialize.(PROCESS)
    module.name           = "Eval Code Injection",
    module.category       = "Injection Modules",
    module.differential = false;

2. Declare alteredRequests(array) and set empty.

3. Push two strings "phpinfo();", and "echo(str_repeat('vega',5));" into array alt


function: initialize(ctx)
START
    1. Declare path state variable, ps, and set value as ctx.getPathState()

    2. Initialize phpRegex = /\.php$/;

    3. Get Uri of ps, and check if it matches with phpRegex(Regular Expression),
       and put result(boolean) into isPHP.

    4. Is isPHP is true? (DECISION)
        if NO, then goto END
        else
            Is isParametric, a property of ps,  is true? (DECISION)
            if NO, then goto END
            else
                Call ctx.submitAlteredRequest(process, alteredRequests[0], false,
END

function: checkContent(res, ctx, index)
START
    1. Initialize phpInfoRegex = /<h1 class="p">PHP Version [0-9.]+<\/h1>/;

    2. Initialize echoStringRegex = /vegavegavegavegavega/;
```

Weaknesses identified in the algorithms

i. The use of one method to discover vulnerabilities. For instance while discovering SQL some of the algorithms analyzed used only one method by simply checking for special character without checking for Boolean values.

ii. Some of the tools have not been engineered to check certain types of vulnerabilities

2.10 Related Studies

Several studies that have been conducted in the field of web application vulnerability detection tools. Bau et al (2010) on testing eight web vulnerability scanners (WVSs), showed that WV needs to be improved in detection of both the "stored" and second-order forms of XSS and SQLI, and in understanding of active content and type of scripting languages language used for web development such as java script among others. Khoury (2011) analyzed three black box WVSs against stored SQLI. The results showed that stored SQLI could not be detected even when these automated scanners are taught to exploit the type of

vulnerability. He proposed an increased a detection rate in WVSs for SQL injection vulnerability.

Myers (2011) discussed techniques applicable to black box testing, for reducing the number of false positives. Fonseca et al (2014) used automated tools together with other fault-injection methodologies to test both SQLI and XSS detection performance of 3 WVSs.
All these studies concentrated benchmarking the capabilities of WVSs, for black box testing. This study will benchmark the same type of WVSs in an effort to find the best scanners and then further and analyze their algorithms and use the findings to suggest a hybrid algorithm.

In another study conducted by Dessiatnikoff et al (2011) they presented a new algorithm for discovering web applications vulnerabilities using a black-box approach. Their main objective was to improve the detection accuracy and efficiency of existing web vulnerability scanners and to move a step forward to automation the detection process. In this particular study they concentrated on SQL injection. The proposed algorithm was based on the automatic grouping of the responses returned by the web applications using a complex data clustering techniques that fires inputs that lead to successful detection of vulnerabilities.

2.11 Research Overview

The scope of this study involves the following

i. Identification of open source vulnerability tools to be used in the testing if the selected web applications
ii. Benchmark the tools against the set metrics
iii. Study the algorithm of the selected tools by identifying strength and weaknesses of each algorithm
iv. Propose an improved hybrid algorithm
v. Test and validate the improved algorithm.

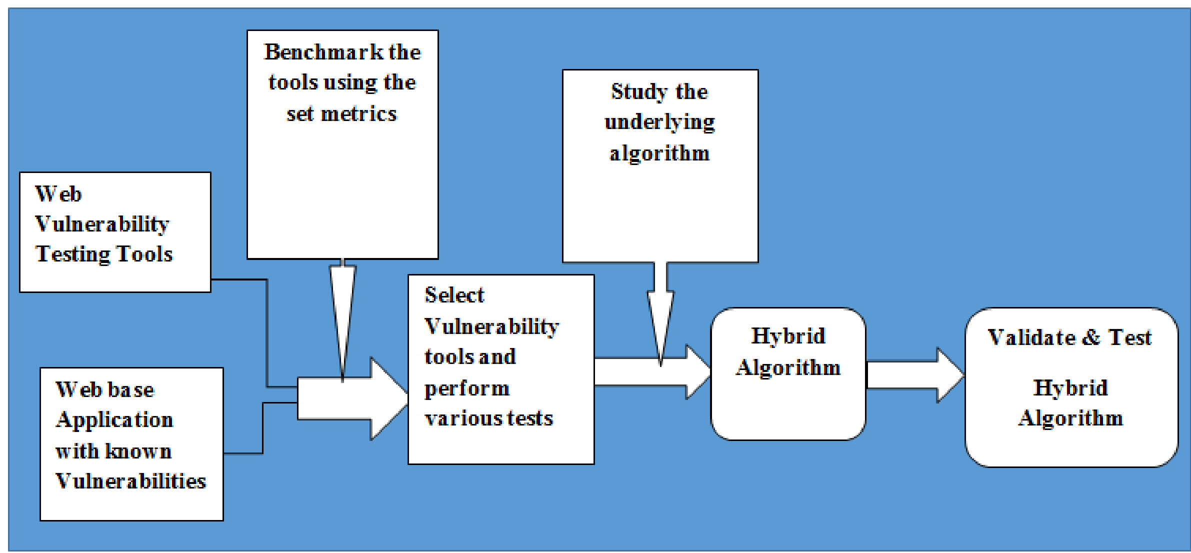


Figure 2:5 Research Overview diagram

CHAPTER 3 : RESEARCH METHODOLOGY

3.0 Introduction

This chapter encompasses the research design that is used in this research project. It also addresses the target population, sample size determination as well as sampling procedure. The chapter also covers the methods of data collection, validity and reliability of research instruments, data analysis and ethical issues in the research.

3.1 Research Design

Research Design can be described as a systematic and organized effort to investigate the exact problem to provide a solution (Sekaran, 2011). Quantitative research is categorized as either experimental or descriptive research. The motive of descriptive research is to enhance familiarity with the phenomena and to formulate a more specific research problem or hypothesis after to gaining new insight towards the subject. In contrast, experimental research aims at testing the cause and effects of relationships among variables. In descriptive research, discovered that researchers do not have direct control towards independent variables. This is because their manifestations have already occurred or inherently cannot be manipulated.

This study has used descriptive research design. A descriptive research is a process of collecting data to assist in answering questions regarding the current state of the subjects in the study (Mugenda and Mugenda 2009). Kothari (2009) defines it as a description of the present state of a phenomenon, determining the type of prevailing conditions, attitudes and practices while seeking accurate descriptions. However, to successfully achieve the goals of the research a combination of a qualitative and quantitative approach be applied. In this research project quantitative approach was adopted.

3.2 Hybrid Algorithm Methodology

A hybrid algorithm was designed based on the logic expressed in the diagram below. The major milestone of this algorithm is to reduce the time taken during the scanning process as well as increase the detection accuracy.

The hybrid algorithm is derived from existing algorithms with a goal of increasing the vulnerability scanning accuracy and the time it takes to scan any given web application. Although the accuracy may not be achieved 100% emphasis, an effort has been put to raise it

above the existing tools. The results of the tests will be benchmarked with OWASP results which is updated on a regular basis.

The hybrid algorithm is based on the idea of carefully, combining desirable features of various components so that the new algorithm has the ability to discover vulnerabilities that could not be detected. However the combination is not done blindly, it is based on various factors such as optimization and sophistication among others with an aim of increasing efficiency.

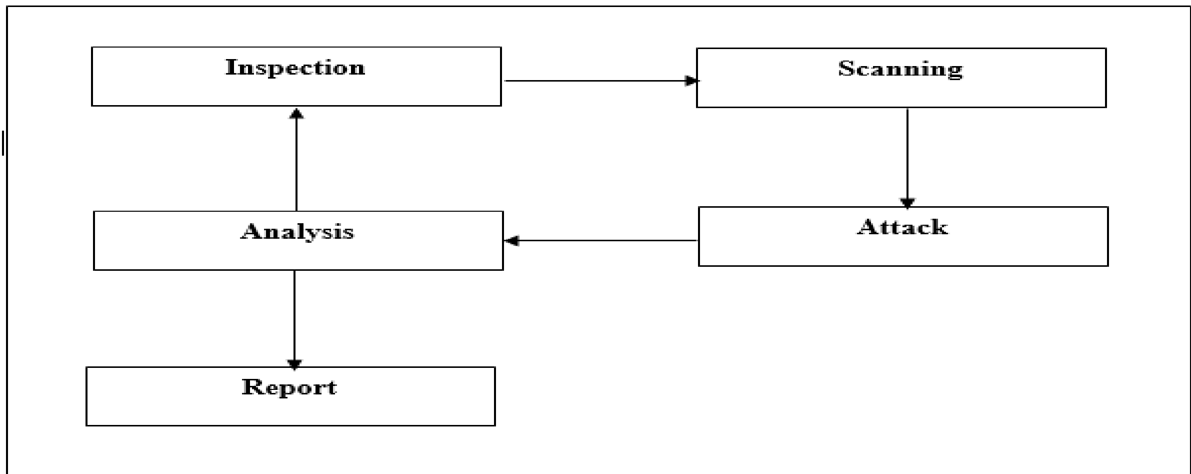


Figure 3:1 Hybrid Algorithm methodology

The hybrid algorithm comprises of five phases as shown in the figure above. Inspection or crawling this phase focuses on looking for information about the web application. The more the details found on this stage, the more successful the entire scanning process will be. Once the first phase is completed, the Scanning process begins, which involves, recognizing the weaknesses that exist in the web application. Once the vulnerabilities are discovered they are analysed in the next phase and then a report is displayed at the end of the entire process.

Trade-offs (Time vs. Accuracy)

If the algorithm is designed to scan all the possible web vulnerabilities, such a program would take a very long time to scan. This would be unrealistic, since users want a program that takes less time to detect any vulnerability. For this reason, accuracy has not been fully optimised. However the fuzzing and crawling components have been engineered to work efficiently and deliver acceptable results.

Accuracy was given a lower priority while scanning time was awarded a priority since it would not be worthwhile to create an algorithm that is 100% accurate but takes a long period of time to scan. In a real world scenario, it would not be practical for users to wait for time consuming web scanners. As a matter of fact an application that takes long to scan would be ignored by the users.

Simulation Design

A program to test and validate the hybrid algorithm is built, based on the flowchart below. This simulation will be useful in testing and validating the hybrid algorithm.

The user will input the URL (uniform resource locator) of the web application to be scanned and click on the scan button.
The scanning process involves crawling and parsing and the discovery of the vulnerabilities, this process is repeated until all the vulnerabilities have been discovered. Once this process is completed, the analysis is done and finally a report is displayed showing the discovered vulnerabilities discovered and their location.
The scanning process includes, crawling and fuzzing. After the scanning process is completed, the results are submitted for analysis and a report is displayed.

Input: The URL of the web application to be tested. This is provided by the user who initiates the web scanning process.

Processing: This involves crawling all the web pages, fuzzing, and identification of any weakness and firing inputs to check for any vulnerability.

Output: The results of processing are analysed and presented in a report format.

Contents of the Scanning Report

Although the report displayed depends on the tool used some of the common features include

- Number of vulnerabilities discovered
- Name or type of the vulnerabilities detected
- Quantity or number of vulnerabilities discovered
- Location or the webpage where the vulnerability has been detected

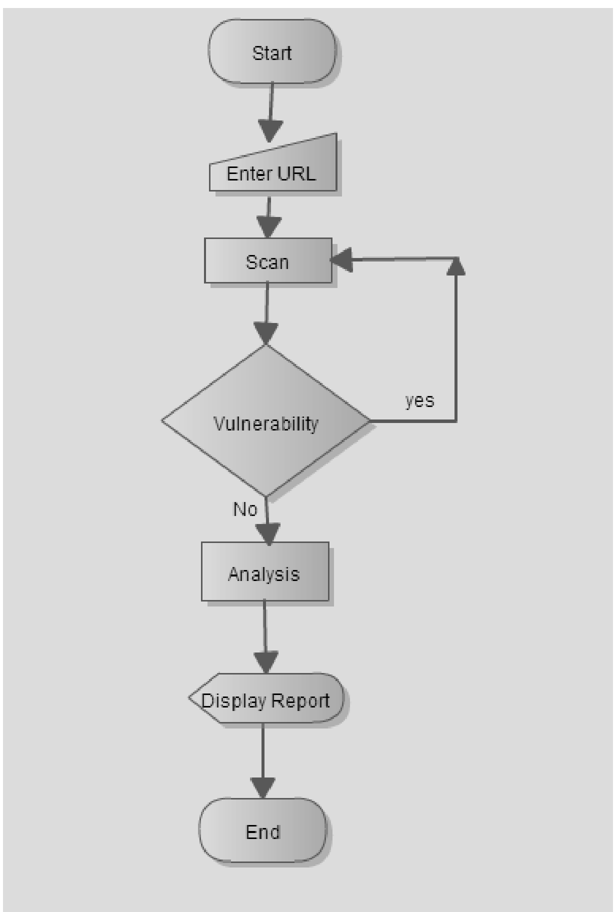


Figure 3:2 Hybrid Algorithm Design

Some of the web scanning tools such as Vega classified the results according to severity of the vulnerabilities. The serious loopholes are classified as high while the others are classified as medium or low depending on the consequences of the vulnerability. See appendix for a sample of the scanning report.

The figure 3.3 below illustrates the scanning process

The user will initiate the scanning process; the application will crawl the web based application and gather the information required to generate the attack. In the attack phase, the fuzzing component will generate the required input to test the vulnerability. During this process the attack module will interact with web application like a user in the real world. After this process the response will be analysed and the process repeated until all the vulnerabilities are discovered. When the attack process is completed a report about the discovered vulnerabilities is displayed.

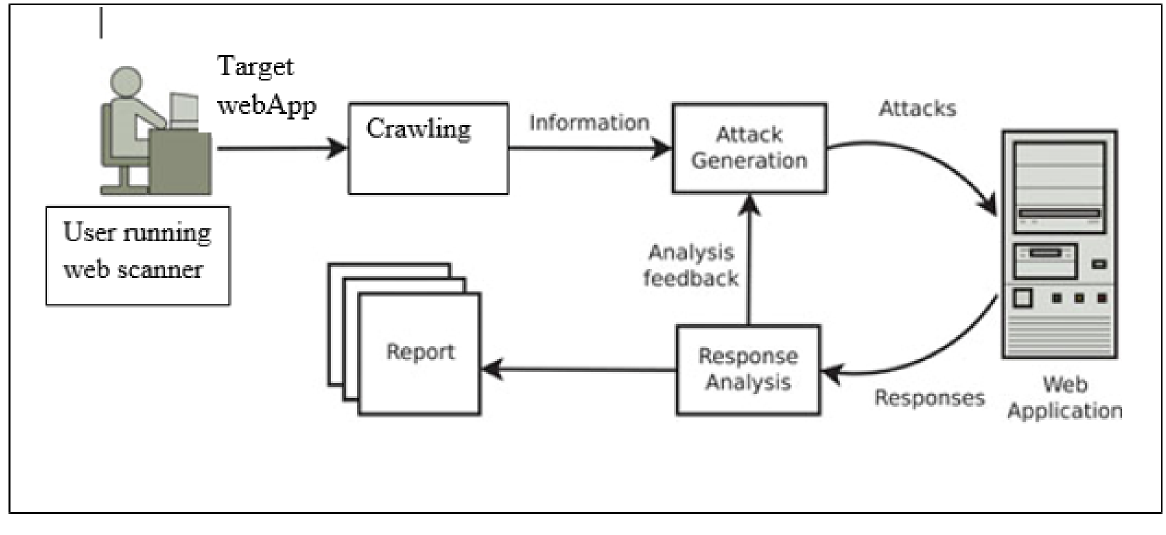


Figure 3:3 Illustration of the web application scanning process.

3.3 Target Population

The study will target opens source web application vulnerability testing tools. Any open source web application vulnerability scanning tool will be eligible to be used in this study; further the study will target webs applications all of which will have known vulnerabilities. The vulnerabilities will include but not limited to SQL Injection, Cross-Site Scripting (XSS),X-Path injection, Cross-Site Request Forgery, Command Injection ,Cross site tracing , file inclusion, Remote file inclusion, HTTP Response Splitting, server side injection (SSI) and Buffer overflow .

3.4 Sampling Procedure

Purposive sampling based on various categories or classifications will be used to select the tools that will be used in the study. The tools selected from lists available on various online portals that classify open source web scanning tools in reference to various factors such as their capability and accuracy of detecting vulnerabilities.

The website should have well known vulnerabilities in advance which the tools will be benchmarked with. Website applications from WAVSEP by Chen (2014) a benchmarking platform designed to assess the detection accuracy of web application scanners will be used. The agenda of WAVSEP's test cases is to provide a broad understanding of which detection barriers that each scanning tool can bypass, and which common vulnerability variations each tool can discover. The metrics used in this research include, detection accuracy, number of

vulnerabilities detected, Time taken to scan a given web application, consistency and stability, features available. These metrics are similar to those used by McQuade (2014)

3.5 Sample Size

Copper and Schindler (2008) suggested that to design the sample, the following should be used: parameters of interest, sampling frame, the target population, the appropriate sampling method and the required sample size of the sample. Purposive sampling will be used to select web applications with known vulnerabilities as well as the tools for scanning the chosen applications. This is because purposive sampling accords the researcher the leeway to target cases that had the required information.However all the selected tools will be benchmarked with OWASP top ten lists of vulnerabilities. Analysis will be performed against the set metrics to chosen tools which is a representative of the sample. The algorithms of these tools will be analyzed and be used to suggest an improved hybrid algorithm.

Tools that were not selected for this research

- **i.** Commercial, tools were not considered in this study since the source code is not available for scrutiny. In addition, these tools are pretty expensive and out of reach for some people.
- **ii.** Tools that are no longer available for download. The researcher did not consider such tools since could not be downloaded.
- **iii.** Tools that after installation could not work well for one reason or the other. Some tools would just hang in the middle of the scanning process and fail to continue even after the process is restarted several times.
- **iv.** Tools that have not been updated for a while, such tools were not chosen since the accuracy of a vulnerability report produced by such tools could not be guaranteed

3.6 Data Collection

This study will primarily rely on quantitative data. Data will be collected about the detection accuracy, the number of vulnerabilities detected, reliability, consistency and stability, features available of the tools. Data will be collected through observation and examination of the reports displayed at the end of the scanning process. These metrics have been used a study that was conducted by McQuade (2014).Each web scanning tool was tested against the web applications with all the relevant settings configured.

Metrics

i. **Detection accuracy** – the number of vulnerabilities detected by the tools. This is expressed in terms of percentage.
ii. **Time** – the time taken by any of the tools under study to scan a given web application.
iii. **Consistency and reliability**– this was arrived at after running the same tool several times against the same web application under the same conditions and configurations and comparing the results.

Experimental design

The experiments were performed by running seven opens source web application scanners on four web applications with known vulnerabilities. These web applications were installed on virtual machines which have similar configurations and resources.

Resources required for the experiment

i. Computer configured with four virtual machines
ii. Operating system: windows 8.1 professional edition
iii. Hypervisor such as VMware
iv. Web servers such as apache, tom cat and Xamp

These web applications were installed on virtual machines which had similar configurations and resources. The virtual machines specifications are processor, 2.6 GHZ Core i5, 2 GB RAM, 100 GB HDD and running on windows 8.1 professional edition.

In an effort to make sure that we have a similar test environment, similar configurations were used on the virtual machines regardless of the test conducted. Each scanning tool was run against identical yet distinct environment. This was critical to ensure that we obtain actual results without deviation due to different resources.

The following tools were used in the experiment

Open Source Web scanning tools	Web applications with known vulnerabilities
• Wapiti • Websecurify • Arachni • W3af • Zed Attack Proxy • Vega	• OWASP WebGoat • Mutillidae • zero.weapplication.com • phpBB

Table 3:1 vulnerability scanners and web applications

Ck AppScan – A tool that was developed by the researcher to test and validate the hybrid algorithm.

Testing procedure

In this research, all the selected web scanning tools were run on the aforementioned web applications and the results recorded. Below find the testing procedure

i. Launch the web scanning tool
ii. Enter the url of the web application to be tested
iii. Click on the scan button and wait for the scanning process to be completed
iv. If the scan is carried out successfully, a report will is displayed with the results. During the scanning process web vulnerabilities discussed in section 2.3 are scanned and if found they will be displayed in the report.
v. Repeat this process for all the tools.

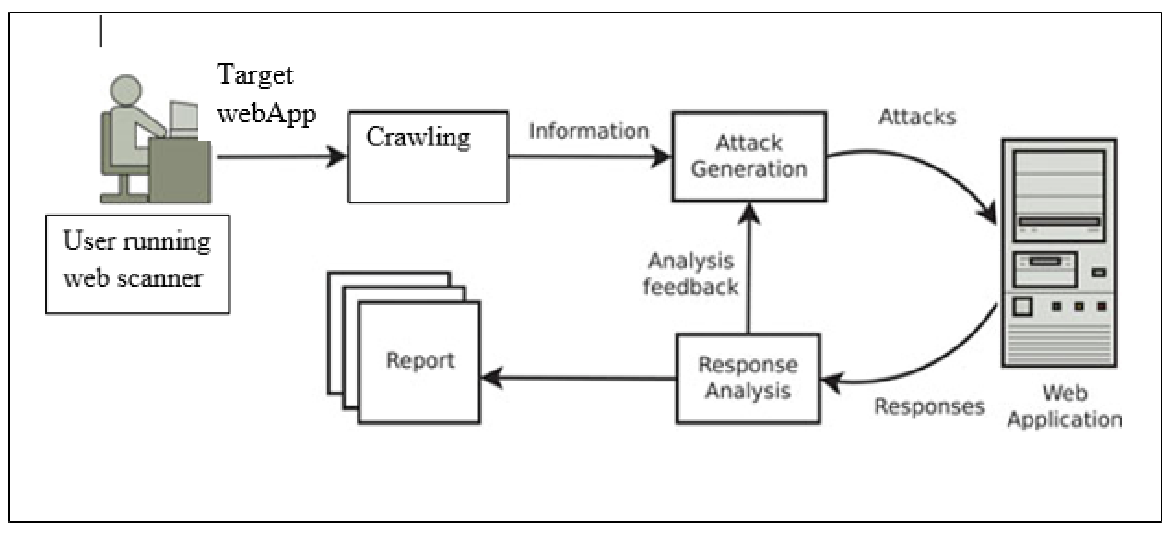


Figure 3:4 Vulnerability scanning process

3.7 Data Analysis

Analysis of data is the process that brings order, structure and meaning to the mass of information collected about the various web vulnerabilities that exist. The data collected will be analysed using simple descriptive statistics. These include central measures of central tendency, mode, mean and measures of dispersion such as percentages and ranks. Quantitative analysis will be employed by the researcher together with statistical methods to analyse collected data. This will be in terms of tables, pie charts and bar charts. Data was also analysed qualitatively wherein data analysis was inductive.

3.8 Data Presentation

Data collected will be analysed using descriptive statistical software packages. Descriptive statistics such as frequencies, percentages and means will be used. Saunders et al..(2011). The research results are presented in a form of, bar graphs, pie charts and tables for ease of understanding. The tools will be ranked based to the metrics set.

Descriptive statistics is a simple quantitative summary of a data set that has been collected. It helps in understanding the data set and provides the required details that will assist you to put the date into perspective. Descriptive statistics enables understanding of the data through values and graphical representation.

3.9 Limitation and Assumptions

The assumption is that different web vulnerability detection tools have different capabilities. The limitation will be on how to choose the tools to be used in the study. Different tool are built with different vulnerabilities in mind and be used on different platforms. This means that there is a possibility of choosing “tool A” to perform a test, which” tool A” may not well suited to discover.

CHAPTER 4 : RESULTS AND DISCUSSION

4.1 Proposed Hybrid Algorithm

The hybrid algorithm was designed with an aim of improving weaknesses that were found with existing algorithms. A black box approach will be adopted with an aim of improving application scanners. The tool used to test and validate the proposed hybrid algorithm will demonstrate the improved capability of the tool. During the development of this algorithm, divide and conquer approach was adopted. This means that the code is engineered to crawl all the Webpages in a web application and scan for the various vulnerabilities independently.

The proposed hybrid algorithm employs the same concept as the divide and conquer algorithm. This simply means that the each vulnerability is scanned by a module in the source code independently.

The hybrid algorithm consist of

i. **Crawler** - this is a program that browses from one webpage to the other on a web application gathering information about the application
ii. **Fuzzing component** – this is an element in a web scanner that handle the input and expose the vulnerability. It generates date and fires the input in the application. The quality of any given fuzzing component is determined by the number of inputs that are used to find vulnerabilities.
iii. **Analyser** - this is a component that analyses the results that are submitted by the fuzzing component and determines whether the attack was successful or not.
iv. **Report generator**. – This component organises the results of the scanning process and present it in a suitable form.

4.2 Simulation Implementation

In this section, the simulation implementation is discussed. All the technologies used are listed below. The following items are discussed

i. Coding – Explanation of the source codes used is done and sample of the code is attached as part of the appendix
ii. Testing – a series of tests were conducted to test and validate the hybrid algorithm
iii. Installation – installation instructions are attached as part of the appendix
iv. Documentation – user manual is provided as part of the appendix

Implementation tools

The following tools were used during the development of the simulation to test the hybrid algorithm.

i. Windows operating system
ii. Approach – object oriented
iii. Programming language: Java

Choice of programming language

The platform chosen for the development of the program is Java. This choice was arrive at since the researcher is well versed with the language and has a wealth of experience in developing applications using java.

Development of the simulation

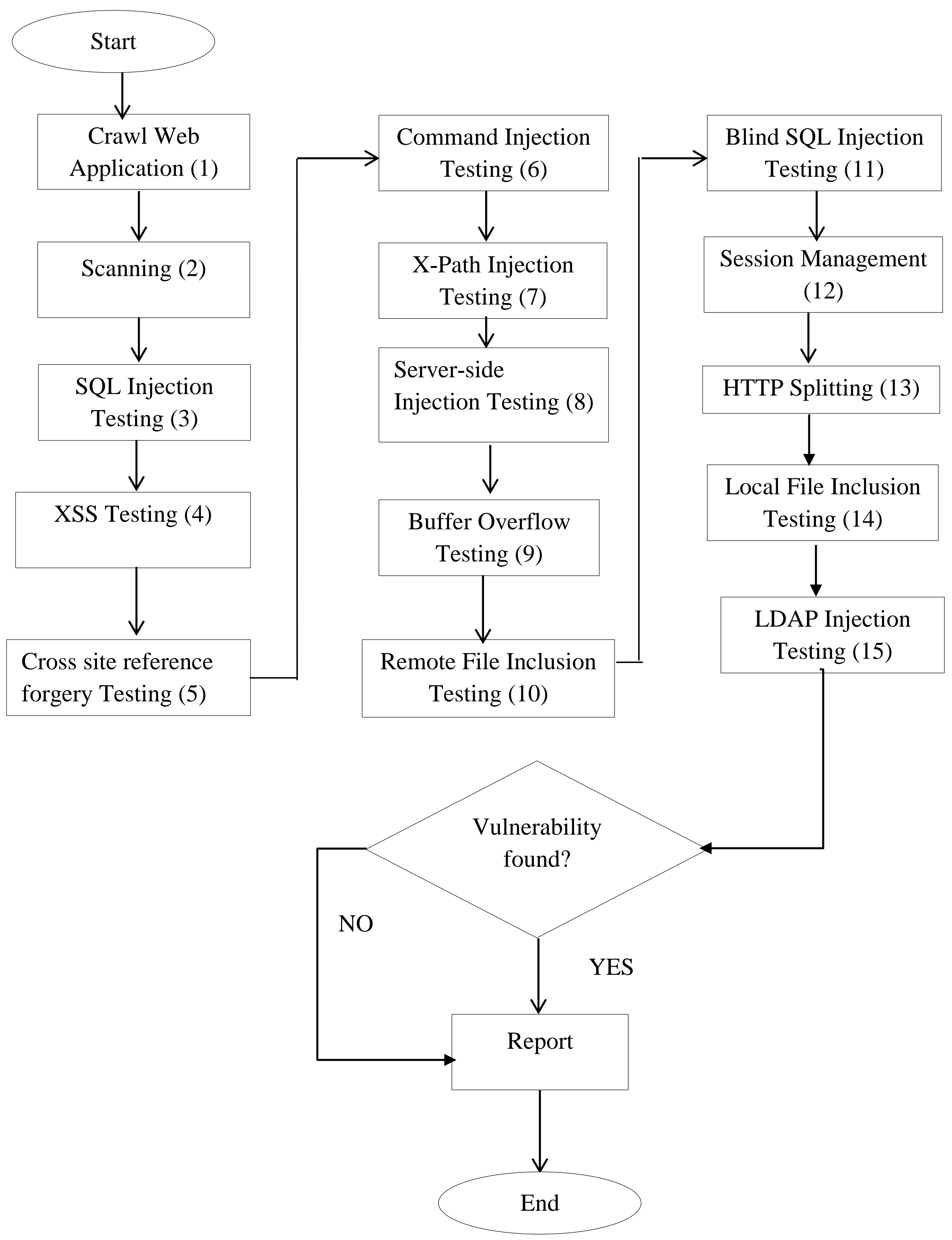
The simulation was divided into various components or modules. Each module deals with the discovery of a particular type of web vulnerability. The source code samples are provided as part of the appendix.

Testing the Algorithm

The algorithm was be tested by translating it in to a simulation developed using java development platform. The simulation will be run against the four web applications and the results collected about detection accuracy, the time taken to scan a given application as well as the reliability and consistency. After the testing process, the results of the simulation were compared with the other opens source web scanners.

Below find the hybrid algorithm

Figure 4:1 Overview Flowchart



Crawling

1) Identify the root of the website (the home page url)
2) Mark the page as visited and push it into a queue
3) Traverse down to identify the immediate sub folders / sub urls
4) Mark the urls as visited and add them to the queue
5) For each url in the url queue
    a. Traverse down to identify sub urls
    b. Mark them as visited and push them in to queue
    c. Repeat step 5 until a dead end is reached
    d. Once dead end is reached remove the url in the immediate top level from the queue
6) Urls in the visited urls array/list it the complete set of urls for the web application

Pseudocode

Ask user to specify the starting URL on web and file type that web App should crawl.

Add the URL to the visited list of URLs and the url queue to search.

While not empty (the list of URLs in url queue search)

{

Take the first URL in from the list of URLs

Mark this URL as already searched URL.

If the URL protocol is not HTTP then

break;

go back to while

If robots.txt file exist on site then

If file includes .Disallow. statement then

break;

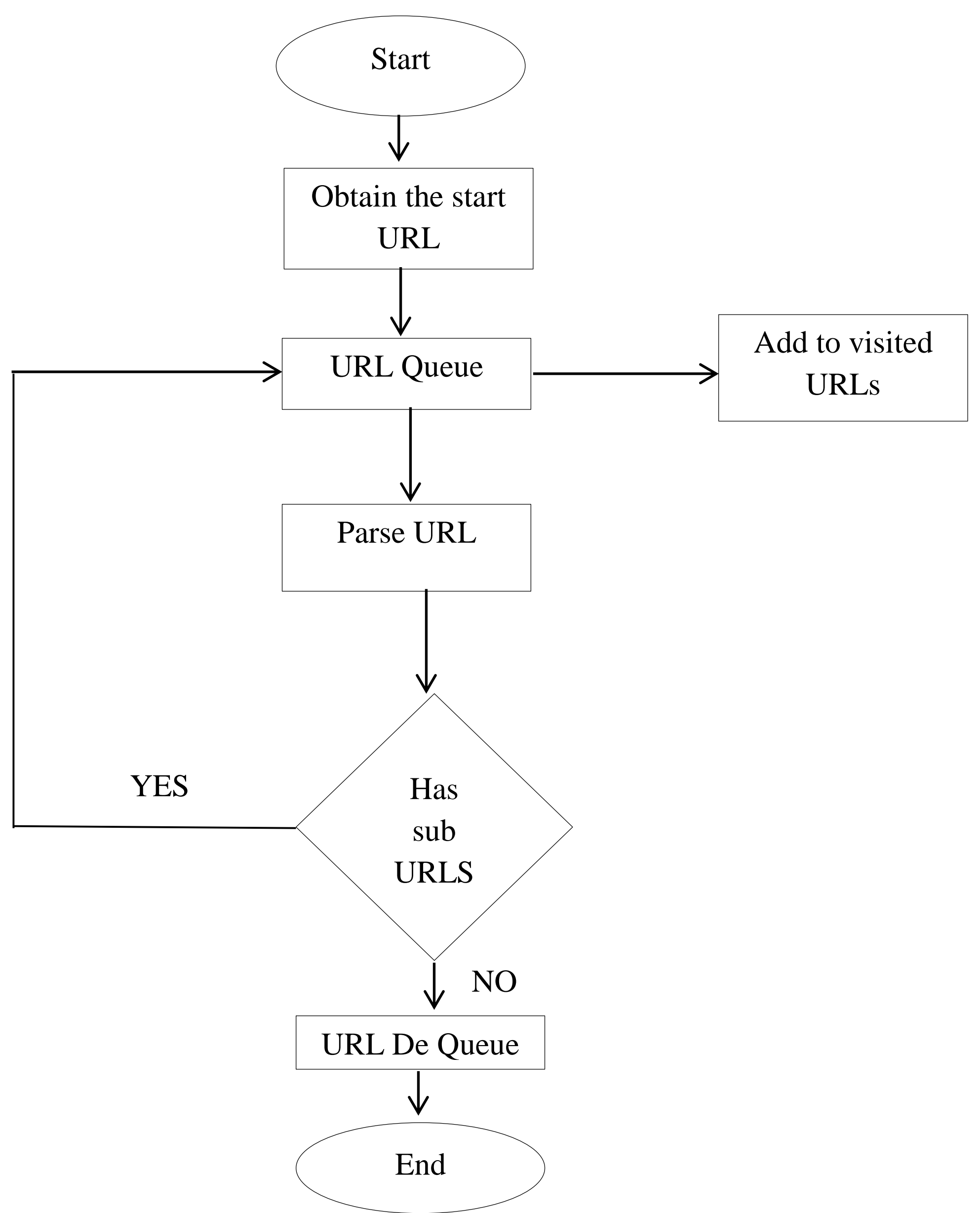
go back to while

Open the URL

If the opened URL is not HTML file then

```
            Break;
            Go back to while
    Iterate the HTML file
    While the html text contains another link {
            If robots.txt file exist on URL/site then
                    If file includes .Disallow. statement then
                            break;
                            go back to while
            If the opened URL is HTML file then
                    If the URL isn't marked as searched then
                            Mark this URL as already searched URL.
                    Else if type of file is user requested
                    Add to list of files found.
    }
}
```

Figure 4:2 Crawling Flowchart (1)

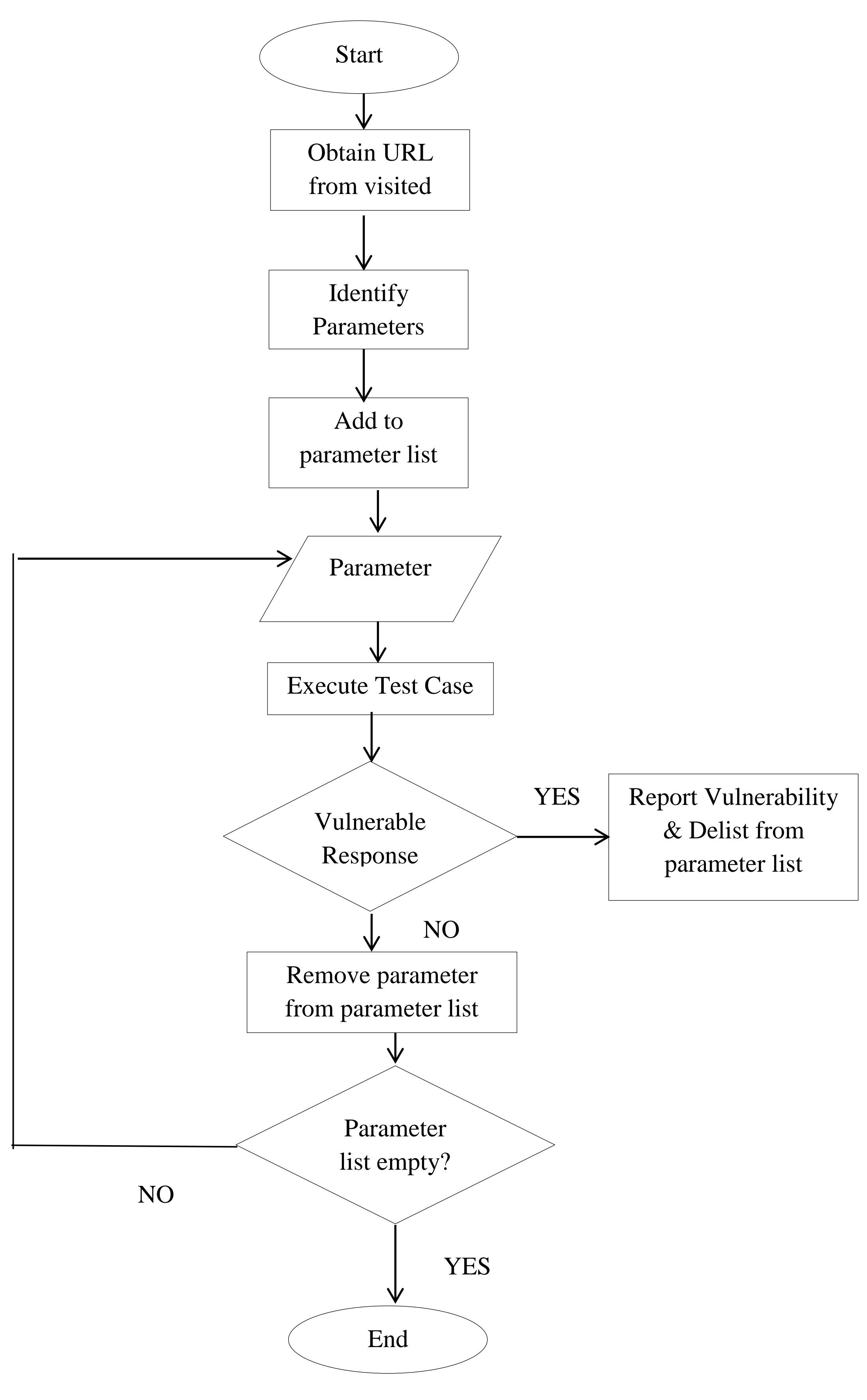


Scanner Module Algorithm

1) For each url in the list of visited urls
    a. Identify all parameters
    b. Push parameters in to parameter list
    c. For each parameter in the parameter queue
        i. Execute scripts/test cases under each of the test categories (sql injection, xss etc)

           Note: Each test category includes a finite list of test cases / scripts
        ii. Verify the response to identify malicious character set
        iii. Remove parameter from parameter queue
2) Report Vulnerabilities

Figure 4:3 Scanning Flowchart (2)



SQL Injections Discovery

The scanning method described in the algorithm below is used as a method two for checking SQL injection by looking for any special characters, and Boolean characters and keywords in the input fields of a web based application. It has a compilation of all the special characters such as <=>{([',&+=<>=])} and a comprehensive collection of major keywords such as update, select, intersect, union insert, delete, drop, truncate and Boolean characters such as , 'AND' 'or '|'or',’

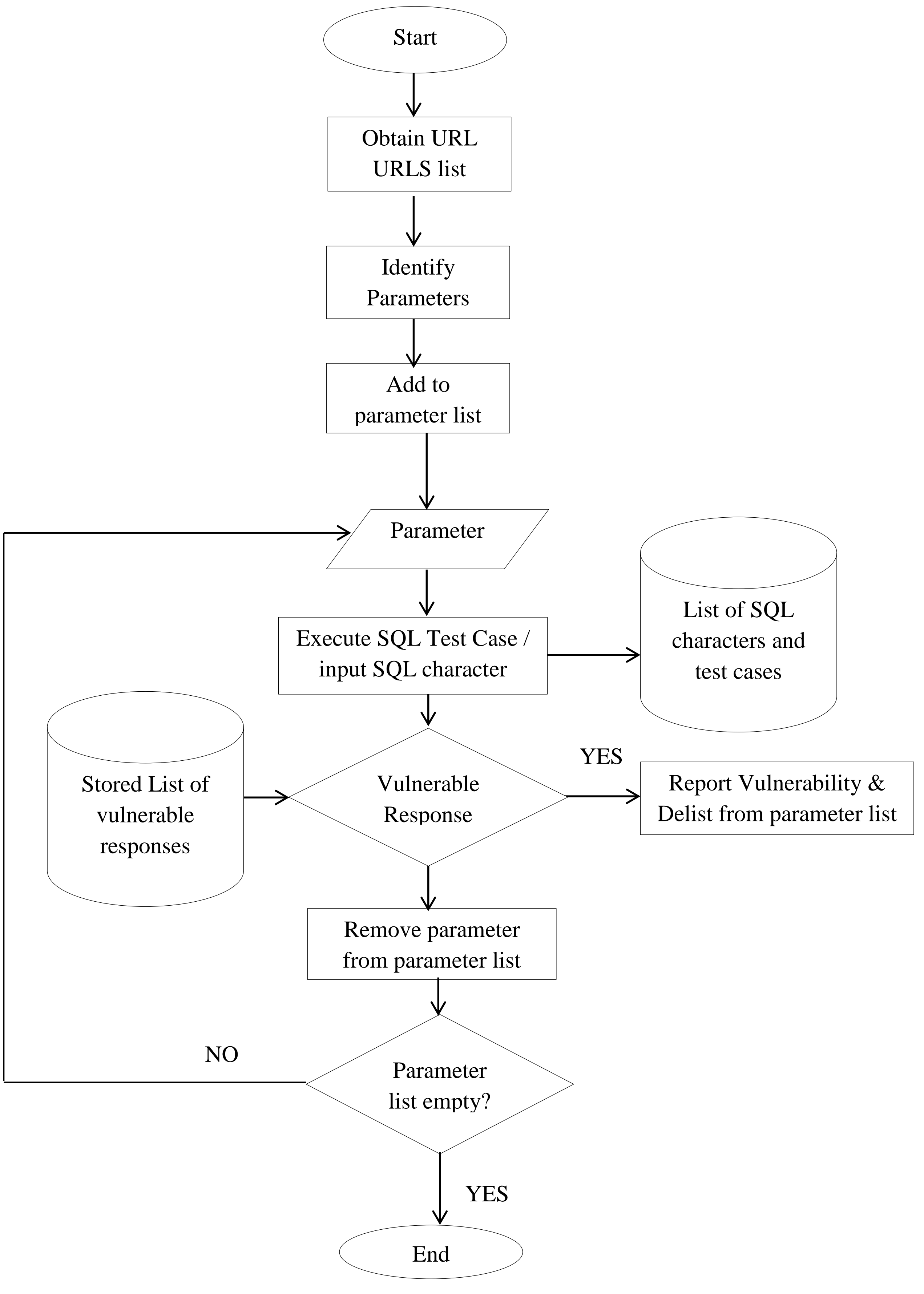
SQL Injection occurs due to invalidated user input. For instance, when a user logs in using a username and a password. ‘’SELECT * FROM users WHERE username=‘user_name’ AND password = ‘entered_password’” . SQL injection testing tries this “SELECT * FROM users WHERE username=x OR ‘1’ = ‘1’ since one is always equal to ‘1’ this query is true for all the records in the database. Using the actual inputs like a user interacting with a web browser, the values are tested against the database. If a mismatch is found in the results are submitted to the vulnerability information collector and then resets the Http request.

The algorithm below detects SQLIs in an effective manner. Which can be applied for any real web-based applications wherever the user and the database interacts

SQL Injection Algorithm

1) initialize sql characters in an array
2) create two maps or lists to store the sql error messages
   i. one for storing specific database error messages like oracle, mysql, microsoft sql error messages etc
   ii. Other for storing generic database error messages
3) Initialize error values in to the maps/list mentioned above
4) Initialize the scanner method – the scanner accepts the http message as input from the the crawler - http message has details on each request or url with the parameter list
5) For each parameter in the http message
   i. Input sql characters from the sql characters array
   ii. Verify the response to check for any matches on error messages from the two maps or lists
   iii. If a match occurs -Flag as sql vulnerability
   iv. Else - Repeat step 5 until the end of parameter list is reached
6) End

Figure 4:4 Flowchart SQL Injection (3)

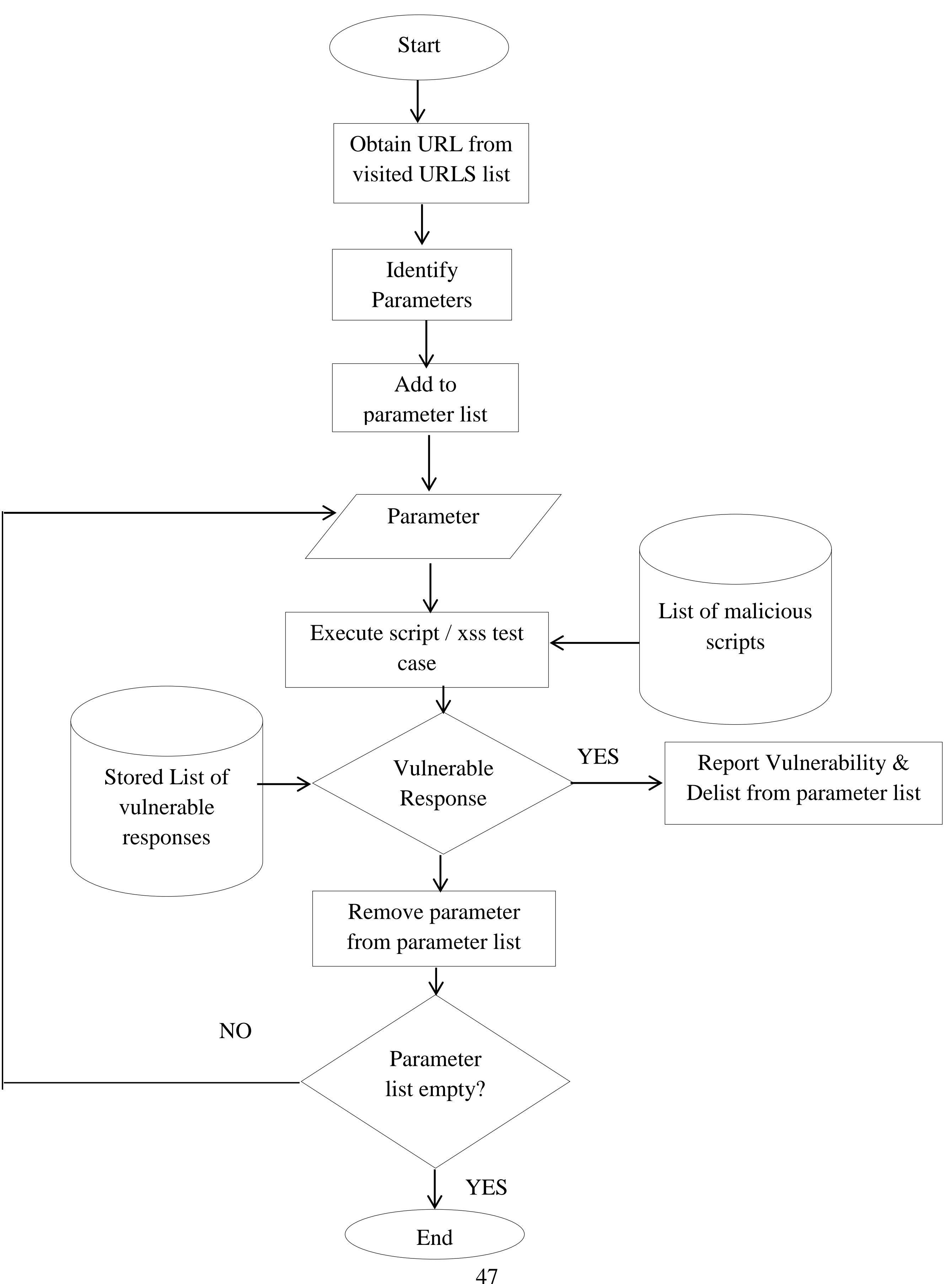


Cross Site Scripting

Algorithm

3) For each url in the list of visited urls
    a. Identify all parameters
    b. Push parameters in to parameter list
    c. For each parameter in the parameter queue
        i. Supply a **script** or a ***XSS test case*** as input to the parameter and pass the request
        ii. Verify the response to identify the supplied script or test case reflected back
4) Report the vulnerability if the response has a script

Figure 4:5 Cross Site Scripting Flowchart (4)

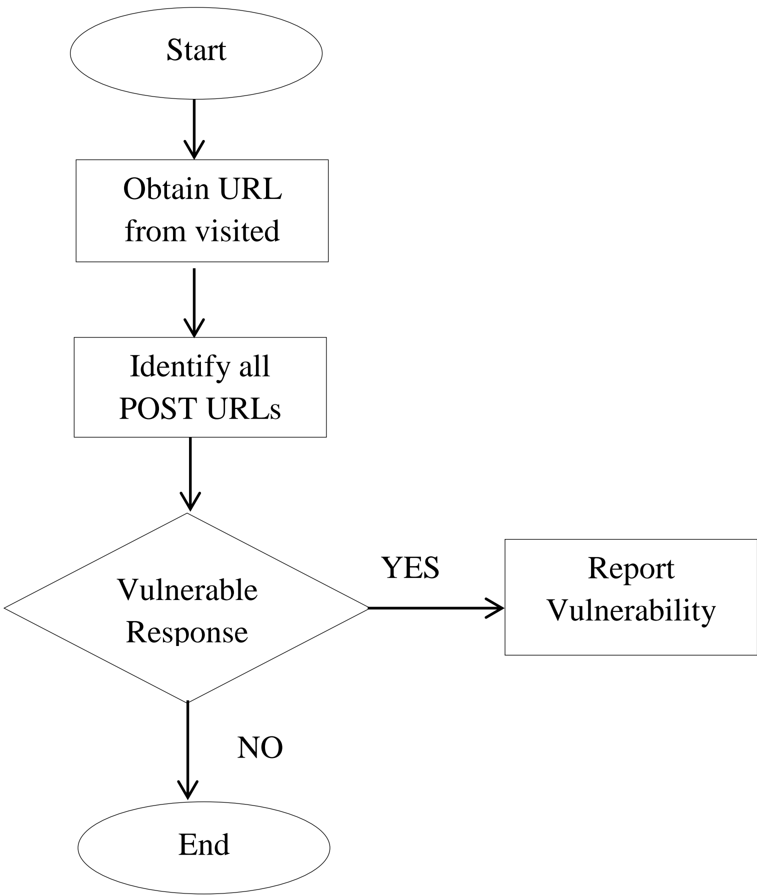


Cross Site Request Forgery

Algorithm

1) For each url in the list of visited urls
    a. Identify all **POST** requests
    b. Verify if the request has a random token attached to it
2) Report the vulnerability if the request does not include a random token

Figure 4:6 Cross Site Request Forgery Flowchart (5)

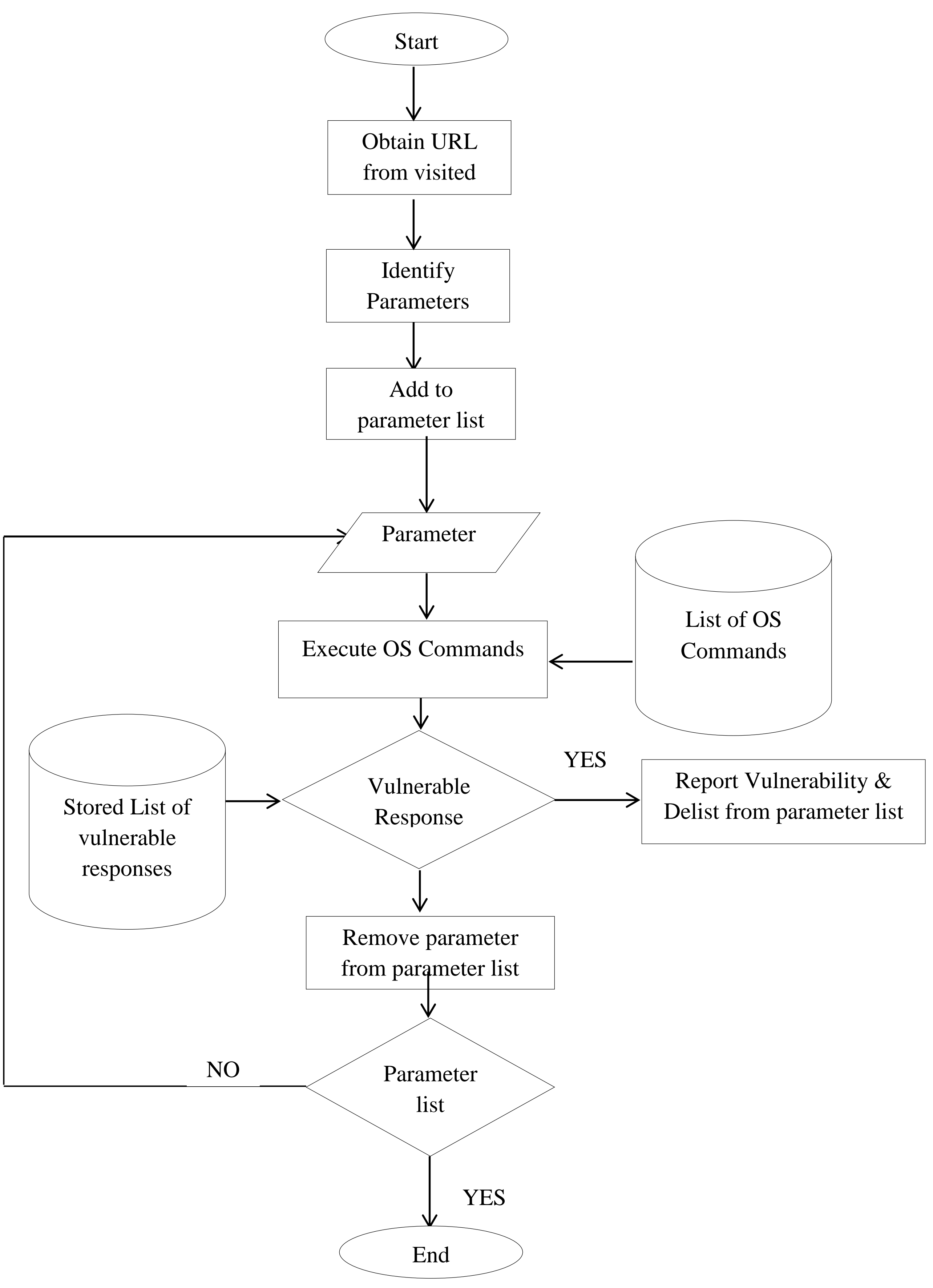


Command Injection

Algorithm

1) For each url in the list of visited urls
   a. Identify all parameters
   b. Push parameters in to parameter list
   c. For each parameter in the parameter queue
      i. Supply an ***OS command as input***
      ii. Verify the response to identify any ***directory structure***
2) Report the vulnerability if the response has a directory structure

Figure 4:7 Command Injection Flowchart (6)

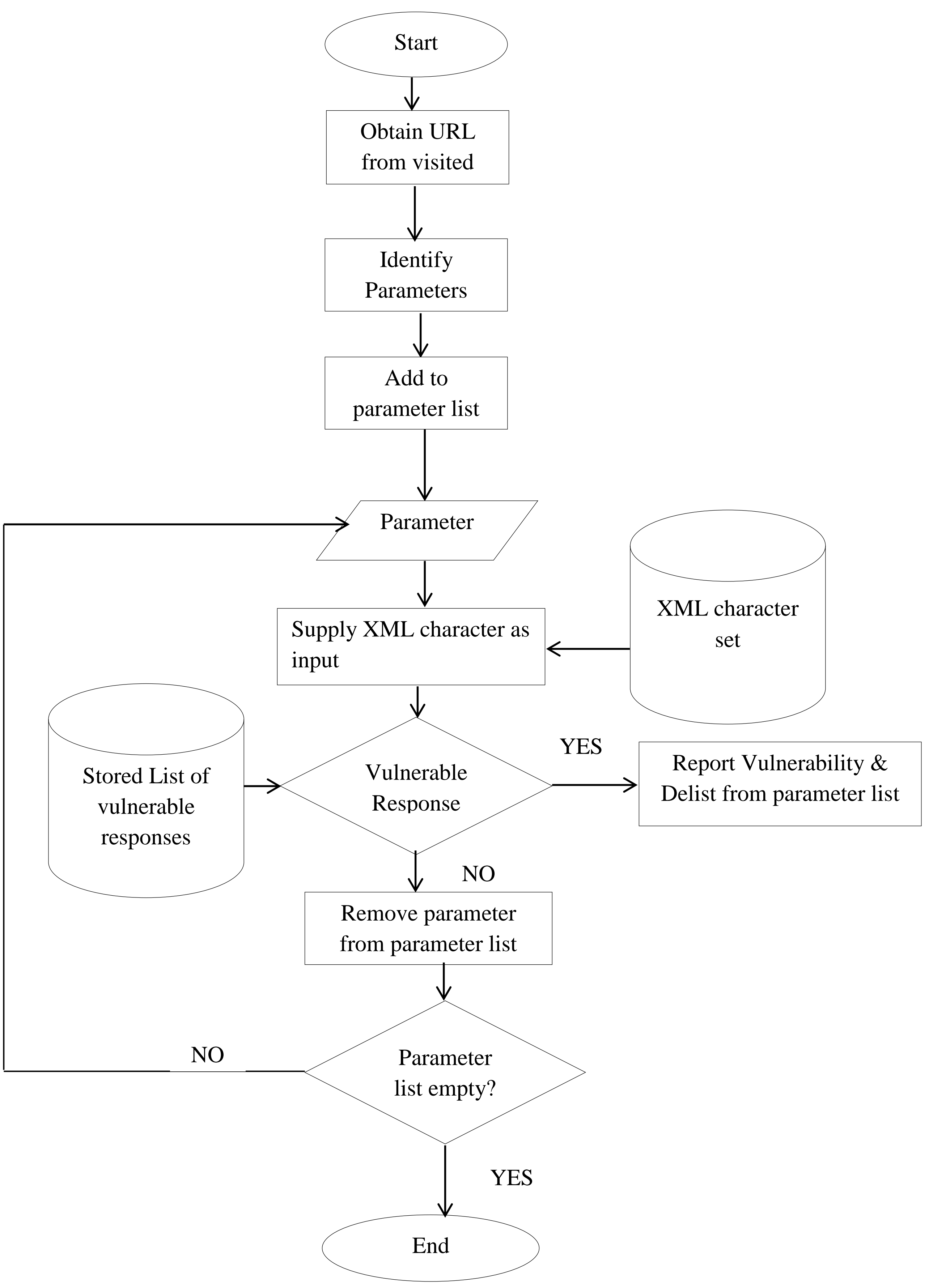


X-Path Injection

Algorithm

1) For each url in the list of visited urls
    a. Identify all parameters
    b. Push parameters in to parameter list
    c. For each parameter in the parameter queue
        i. Supply a xml character as input
        ii. Verify the response to identify any ***xml error messgae***
2) Report the vulnerability if the response has an xml error message

Figure 4:8 X-Path Injection Flowchart (7)



4.3 Data Analysis

This section comprises of data analysis as stipulated in the research methodology, presentation of findings in tables as well as summary and interpretation on findings with regard to the vulnerabilities that exist in various web applications.

4.4 Data Description

This section provides a description of results presented in tabular form. The table shows the list of tools and the vulnerabilities can detect.

Table 4:1Summary of vulnerabilities detected by the web scanning tools

No.	Vulnerability	Wapiti	Arachni	Websecurify	W3af	Vega	ZAP
1	Remote file inclusion	✓	✓	✓	✓	✓	
2	Local file inclusion	✓		✓	✓	✓	
3	Cross site crossing		✓		✓	✓	✓
4	XSS	✓	✓	✓			
5	CSRF		✓	✓	✓	✓	✓
6	Command Injection	✓	✓		✓		✓
7	SQL Injection	✓	✓	✓	✓	✓	
8	LDAP Injection	✓	✓		✓		✓
9	Buffer overflow				✓	✓	
10	X-path Injection	✓	✓		✓		✓
11	Session management			✓			
12	SSI injection				✓	✓	
13	HTTP Splitting	✓	✓		✓	✓	✓
14	Blind SQL Injection	✓	✓		✓	✓	✓

4.5 web Applications Scanning Results

The simulation results were evaluated by comparing the performance of the hybrid algorithm with the graphs and tables below highlights findings as described in Table 4.1. The results are as illustrated in the tables and figures below.

Table 4:2 Vulnerabilities discovered by Vega

	Vega														
vulnerabilities	RFI	LFI	XST	XSS	CSRF	CI	BSQL	SQLI	LDAP	BO	X-Path	SM	SSI	HTTP	**Total**
Webgoat	X	X		X	X	X	X	X			X				**8**
Mutillide	X	X		X		X	X	X		X	X			X	**9**
Zero_Webapp	X	X		X		X		X		X				X	**7**
PHPBB	X	X		X		X		X			X		X	X	**8**

Table 4:3 Vulnerabilities discovered by W3AF

W3af															
vulnerabilities	RFI	LFI	XSS	XSS	CSRF	CI	BSQL	SQLI	LDAP	BO	X-Path	SM	SSI	HTTP	**Total**
Webgoat	X	X	X	X			X	X	X		X				**8**
Mutillide	X			X	X			X			X			X	**6**
Zero_Webapp		X	X	X	X		X	X	X	X	X				**9**
PHPBB			X	X	X	X	X	X			X			X	**7**

Table 4:4 Vulnerabilities discovered by Websecurify

Websecurity															
vulnerabilities	RFI	LFI	XST	XSS	CSRF	CI	BSQL	SQLI	LDAP	BO	X-Path	SM	SSI	HTTP	**Total**
Webgoat	X		X	X	X		X	X			X		X	X	**9**
Mutillide	X	X	X	X				X			X		X	X	**8**
Zero_Webapp			X	X	X		X	X	X		X	X	X	X	**10**
PHPBB	X	X		X	X			X						X	**6**

Table 4:5 Vulnerabilities discovered by Arachni

Arachni															
vulnerabilities	RFI	LFI	XST	XSS	CSRF	CI	BSQL	SQLI	LDAP	BO	X-Path	SM	SSI	HTTP	**Total**
Webgoat	X	X	X	X	x			X		X	X	X	X	X	**11**
Mutillide	X	X	X	X	X			X			x			X	**7**
Zero_Webapp	X		X	X				X		X					**5**
PHPBB	X	X	X	X	X			x			X	X		X	**9**

Table 4:6 Vulnerabilities discovered by Wapiti

Wapiti															
vulnerabilities	RFI	LFI	XST	XSS	CSRF	CI	BSQL	SQLI	LDAP	BO	X-Path	SM	SSI	HTTP	**Total**
Webgoat	X	X		X			X	X			X			X	**7**
Mutillide	X	X	X	X			X	X			X			X	**8**
Zero_Webapp	X	X	X	X		X	X	X	X		X	X	X	X	**12**
PHPBB	X	X	X	X			X	X	X		X	X	X	X	**11**

Table 4:7 Vulnerabilities discovered by Zed Attack Proxy (ZAP)

Zap															
vulnerabilities	RFI	LFI	XST	XSS	CSRF	CI	BSQL	SQLI	LDAP	BO	X-Path	SM	SSI	HTTP	**Total**
Webgoat	X		X	X	X			X			X			X	**7**
Mutillide			X	X				X				X		X	**5**
Zero_Webapp	X	X	X		X						X		X		**6**
PHPBB	X	X	X		X			X			X	X	X		**8**

Table 4:8 Vulnerabilities dis covered by CK AppScan (Simulation)

CK AppScan															
vulnerabilities	RFI	LFI	XST	XSS	CSRF	CI	BSQL	SQLI	LDAP	BO	X-Path	SM	SSI	HTTP	**Total**
Webgoat	X	X	X	X	X	X	X	X			X	X		X	**11**
Mutillide			X	X	X		X	X		X	X			X	**8**
Zero_Webapp	X	X	X	X	X	X		X		X	X		X		**12**
PHPBB	X	X	X	X	X	X		X		X	X	X	X	X	**12**

4.6 Data Presentation

A visual representation of the tools accuracy

Figure 4:9 Web Scanning Tools Accuracy:

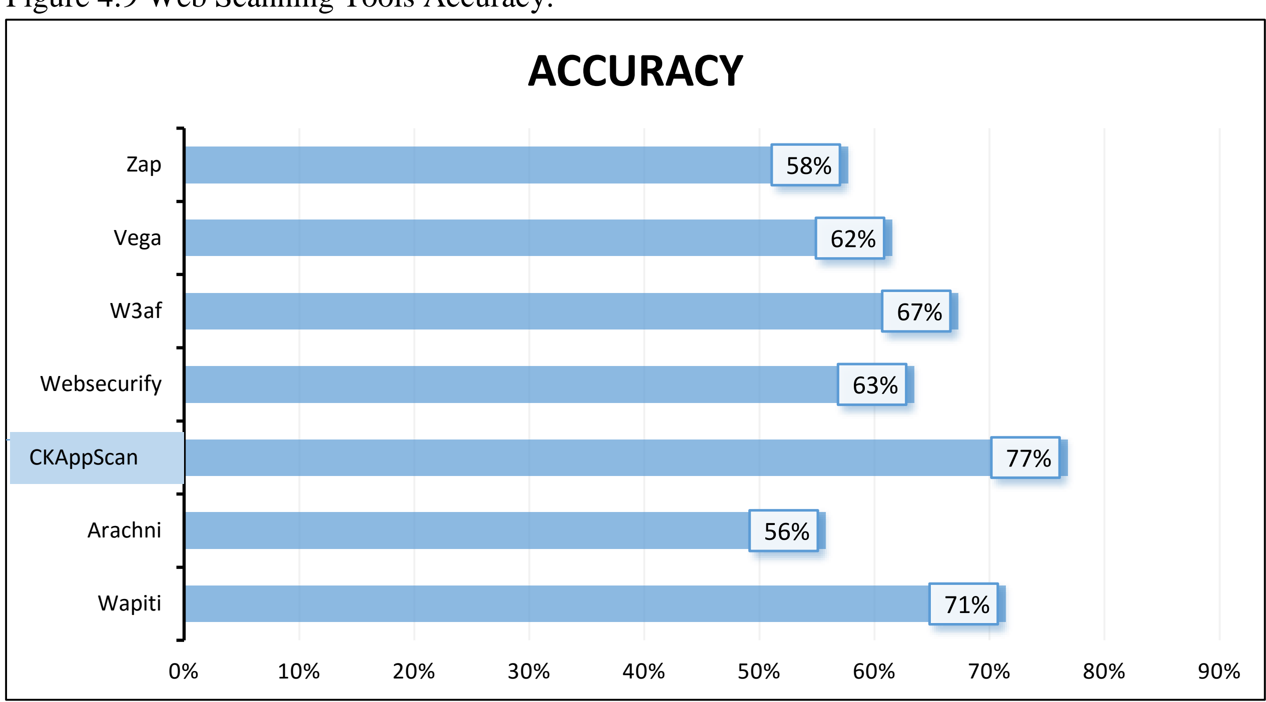


Figure 4:10 Tools Consistency

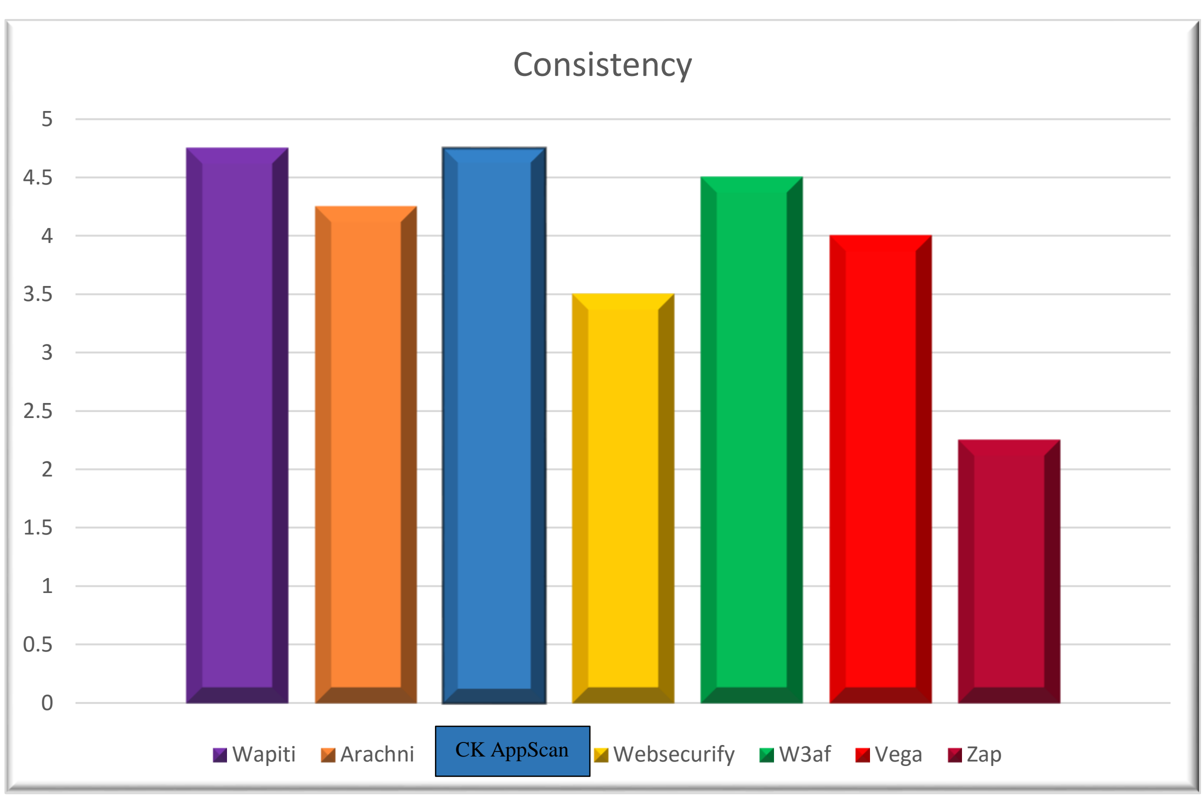


Table 4:9 Time taken to scan various Applications

Time (Minutes)	Wapiti	Arachni	Websecurify	**Ck AppScan**	W3af	Vega	Zap
WebGoat	47	36	21	**72**	88	94	45
Mutillidae	68	44	29	**80**	96	46	51
Zero_Webapp	24	52	31	**96**	78	63	64
PHPBB	33	63	27	**74**	92	71	78
Average	43	48.75	27	**80.5**	88.5	68.5	59.5

Time taken while scanning the web applications using the web scanners the results displayed on this table do not agree with results from studies or research done previously since, the approach and testing environment was different

Figure 4:11 Vulnerabilities vs. Tools

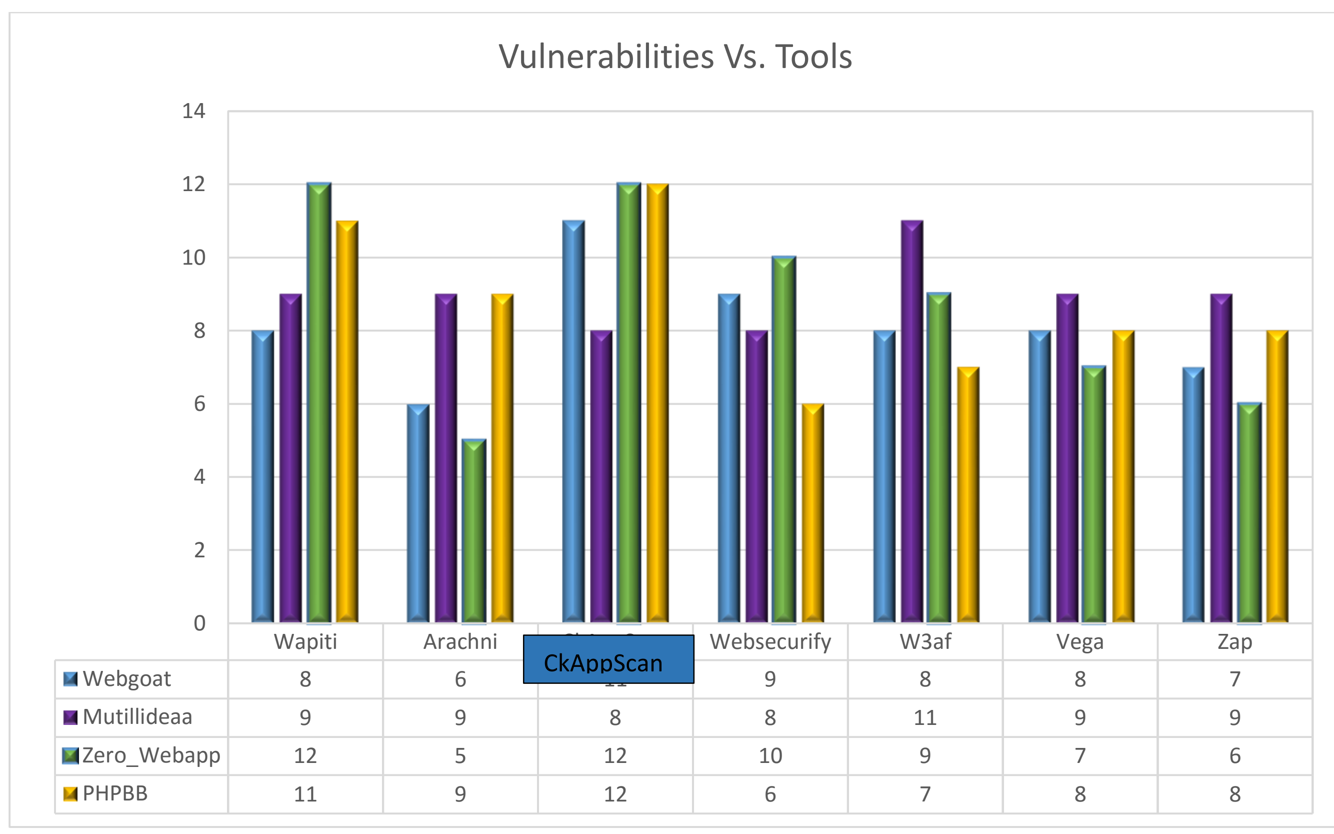


Figure 4:12 Vulnerabilities VS Scanning Tools

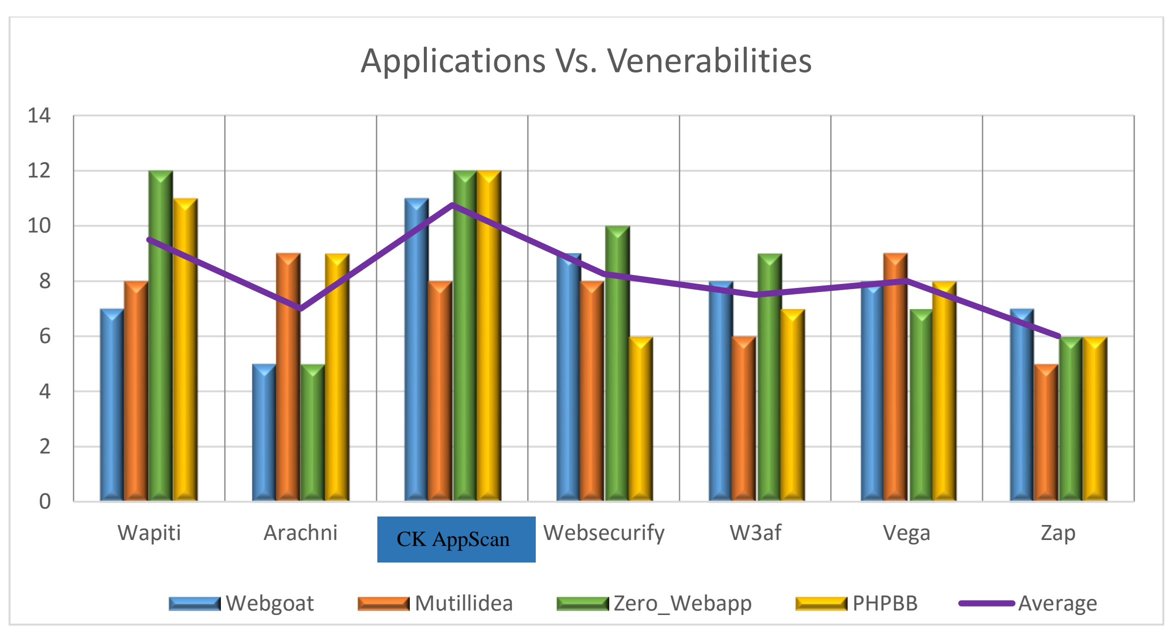


Figure 4:13 Vulnerabilities detected in Zero WebApp

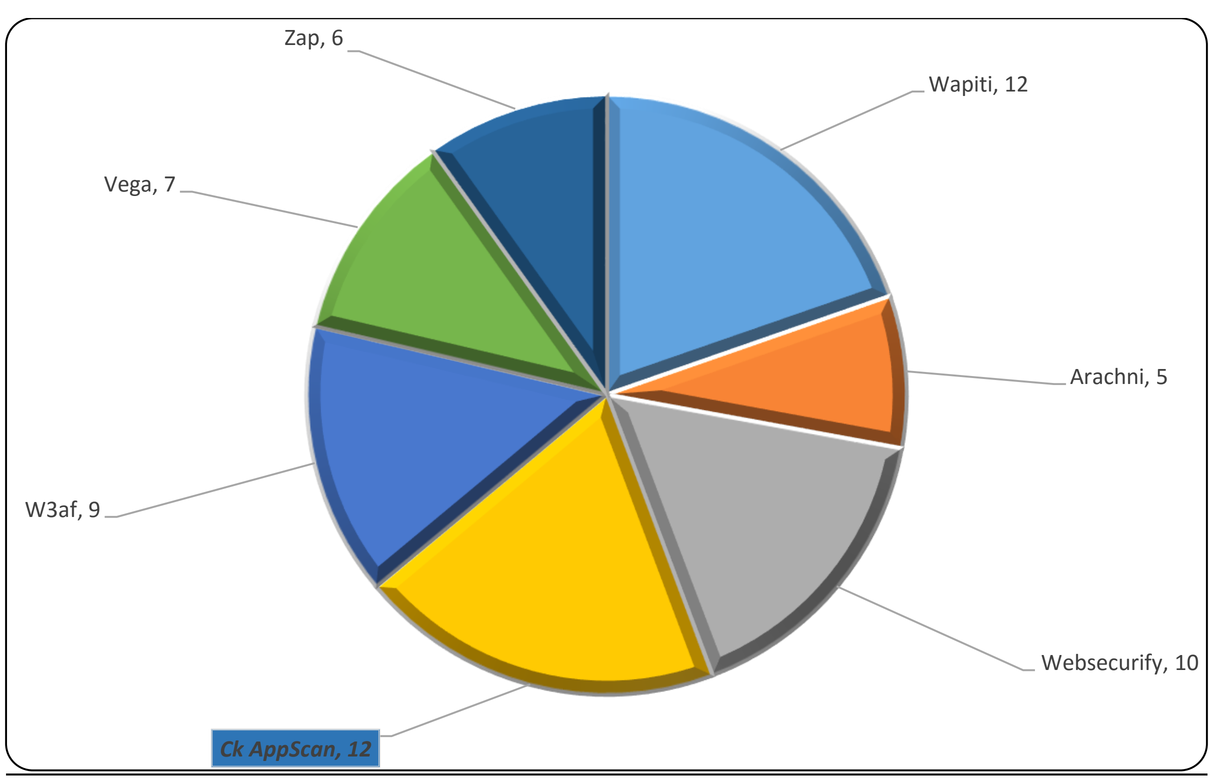


Figure 4:14Vulnerabilities discovered in phpBB

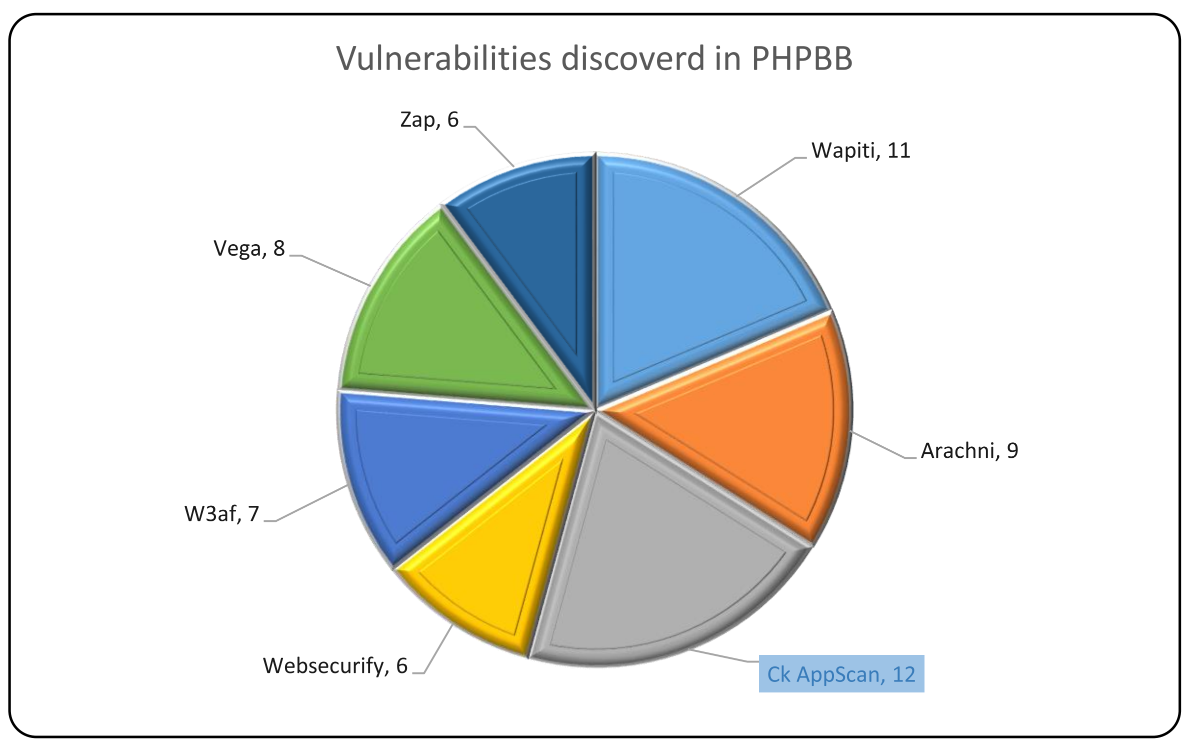


Figure 4:15 The figure below shows wapitis capability to discover vulnerabilities rated as high .These include, XST, XSS, CSRF, CI, BSQL and SQLI

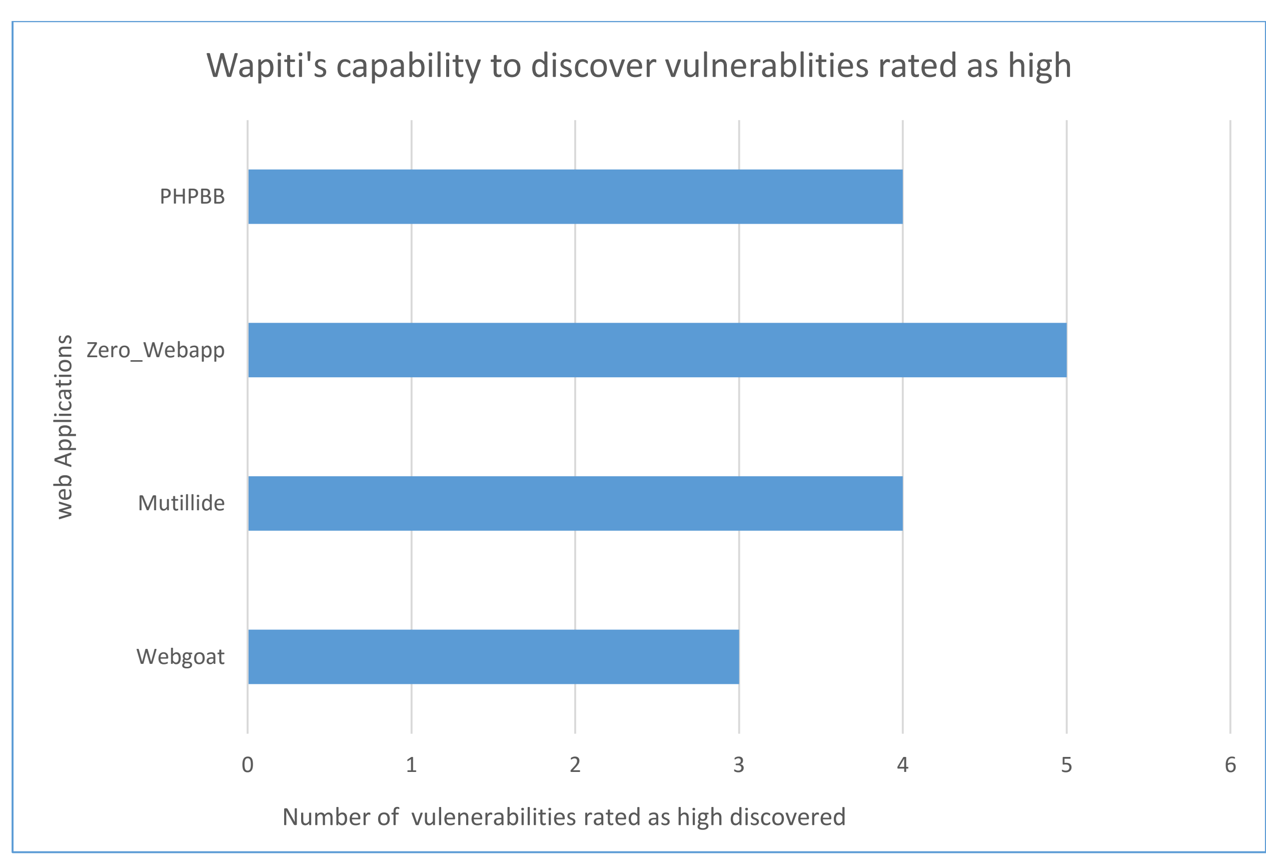


Table 4:10 Weighted Average

Severity	**Vulnerability**	**Assigned Weight**
High	SQL Injection (SQLI)& Blind SQL (BSQLI) Cross site Scripting (XSS)& Reflected cross site scripting Cross site reference forgery (CSRF) &Cross site tracing (XST) Command line injection (CI) &Server side injection (SSI)	3
Medium	Local file inclusion (LFI) & Remote file inclusion (RFI) Buffer overflow (BO) & LDAP	2
Low	Xpath &Session management (SM)	1

Total number of vulnerabilities = 14 **Assigned weight**: High = 3, medium = 2, and low = 1

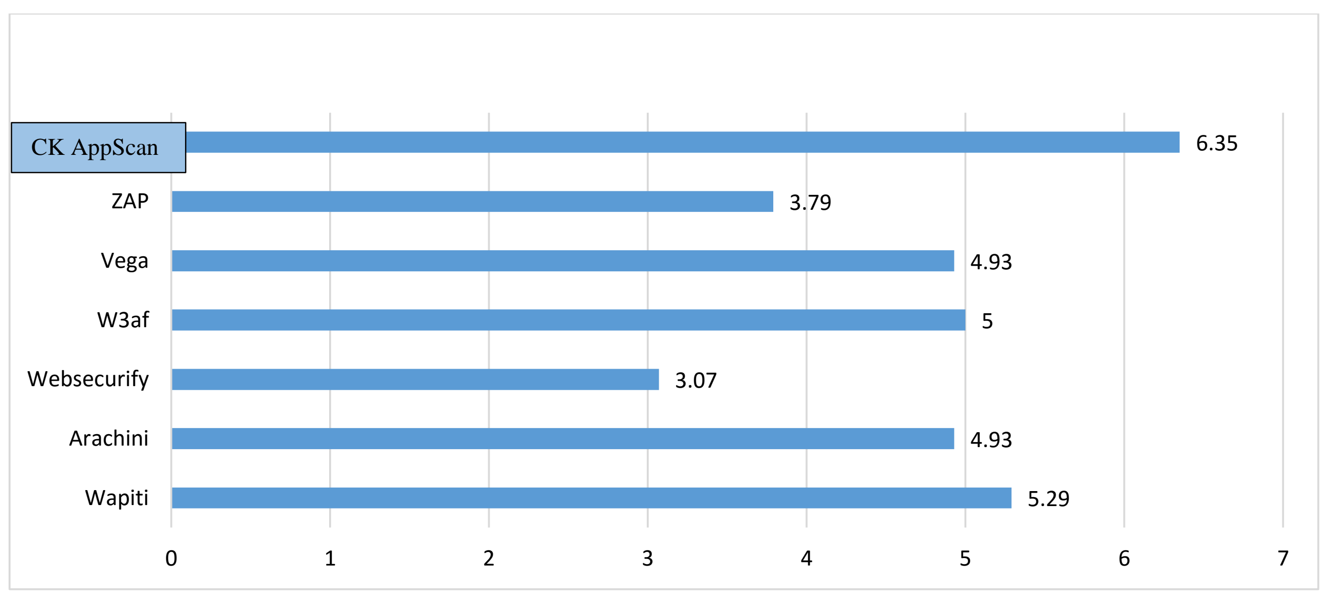
Formula: = SUM ((1st Element * Weight of 1st Element), (2nd Element * Weight of 2nd Element), … , (nth Element * Weight of nth Element)) / Total number of vulnerabilities.

Table 4:11 Wapiti's Weighted Average

	Wapiti															**Weighted Average**
	RFI	LFI	XST	XSS	CSRF	CI	BSQL	SQLI	LDAP	BO	X-Path	SM	SSI	HTTP	**Total**	
Webgoat	2	2		3			3	3			1			1	**14**	**1.00**
Mutillide	2	2	3	3			3	3			1			1	**17**	**1.21**
Zero_Weba	2	2	3	3		3	3	3	1		1	1	1	1	**23**	**1.64**
PHPBB	2	2	3	3			3	3	1		1	1	1	1	**20**	**1.43**
															Total	**5.29**

NB: Blank means that no vulnerabilities were detected.

Figure 4:16 Weighted Average for all the WVS



Summary results for the tools used.

Wapiti - this tool can be rated as above average, it was able to detect most of the vulnerabilities in almost all the categories. However, it could not discover any of CRLF and buffer overflow weaknesses. Wapiti offers high performance and runs smoothly with minimal errors.

W3AF - is a fairly powerful web scanner in discovering vulnerabilities; it has a poor reporting structure when compared to other tools that were sampled in this research. The same sentiments were echoed by McQuade (2014). This tool does not classify the severity of the vulnerabilities detected. It failed to discover buffer overflow, command line injections, BSQL, the only category it excelled was in the detection of XST.

Zap – performance can be classified as poor. It did not perform well in web vulnerability discovery. As indicated by Van der loo (2011) the tool fails to excel in any category. As a matter of fact it failed to discover any session management and LDAP vulnerabilities.

Ck AppScan – performed better than all the other tools tested. It was able to discover a number of weaknesses, however the tool takes a bit longer to do the scanning process. However, if failed to detect LDAP due to the complex nature of detecting this vulnerability

Vega - did not discover cross site scripting session management and LDAP vulnerabilities in any of the applications tested. However it reported excellent results in the detection of SQLI RFI, LFI and XSS.

Arachni - offers a web based GUI interface. It's quite fast and the report presentation is fairly good. This was also reported by McQuade (2014) in one of the studies he conducted about open source vulnerability scanners. Arachni also offers a highly customizable command line interface that is recommended for manual scanning. However its performance in this study was not the best. It failed to detect command line injections, LDAP, and blind SQL. However it provided excellent results in reporting XSS, XST, SQLI, RFI and LFI

Websecurify – this tools excelled in the detection of XSS, SQLI, and HTTP only. It failed to detect buffer overflow errors and all the other vulnerabilities were discovered in a sporadic manner. For this reason it did not perform well.

4.7 Discussion

A comparative study of web vulnerability scanners has also been performed by other researchers from different parts of the world. Although the tools and web applications used are not similar, the vulnerabilities are the same.

In a study conducted by Doupe et al (2010) they used Acunetix WVS, burp scanner, IBM's rational app scan, hailstorm, N-stalker, mileScan, Grendel-scan, NTO spider, W3AF, and HP web inspect against a web application known as wackPicko. They tested vulnerabilities such as XSS, SQL injection, file inclusion, file exposure and command line injection. The conclusion of the study was similar to those drawn by this study. They found out that crawling modern web based applications is indeed a serious challenge for many WVSs. There should be improved and more sophisticated algorithms needed to perform deep crawling. During the development of the hybrid algorithm the researcher was able to develop an algorithm that was able to detect the aforementioned vulnerabilities. This was achieved by employing a sophisticated method of discovering the weaknesses.

In a study conducted by Fonseca et al (2014) shows that many open source WVS have a low ability to detect vulnerability. This is in line with the results analysed after the end of this study. The researcher has developed a more sophisticated algorithm that address this concern and increased the number of vulnerabilities detected.

Khoury (2011) analysed three state-of–art black box WVSs against stored SQLI, and their results showed that stored (persistent) SQLI are not detected. The researcher was able to detect persistent SQL injections by fuzzing web applications using complex discovery algorithms.

Shelly (2010) performed a similar study by using several penetration tools to analyse the performance of several WVS. She used a mix of commercial and open source tools such as wapiti, Grendel-scan, Acunetix WVS, N-stalker, W3AF, and hailstorm. These tools were run against a modified version of BuggyBank web based application. They tools were tested for SQLI, XSS, buffer overflow and session management. The conclusion of this study was that the testing of WVS using secure and non-secure applications is indeed a suitable method to discover web vulnerabilities. In addition, she reported that for the discovery of non-traditional instances of XSS, SQLI, buffer overflow, malicious file execution and session management flows, more research needs to be done to improve the detection mechanisms used by these

tools. The researcher addressed this issue by use of advanced heuristics and permutations during the detection process.

The hybrid algorithm was able to address concerns raised by previous researchers in different studies. This was achieved by adoption of more than one method during the vulnerability discovery process as well as improvement of the existing vulnerability detection methods. For instance it was noted that most of the WVS use either GET or POST method to detect weaknesses. The use of the two methods requires more scanning time nevertheless, more accurate results are realised.

Attack Analysis Proficiencies

By analyzing how each of the web scanning tools discovered vulnerabilities, this information provided the researcher with an insight on how the tools sampled works and shed more light on the areas which can be considered for future research and enhancements.

In a nutshell most of the tools would do the crawling process using the POST or GET parameters. Once the inputs on the web application have been detected, the scanning tool would attempt inject some values in the application and analyze the response. Since these tools have been developed using different algorithms, they use different approaches in their detection mechanism. For instance some of the tools would use numerical values such as 1,2,3,4 while other tools would use letters of the alphabet or even leave the field blank. The option used by the tools had an impact on the results produced.

During the process of XSS discovery, most of the tools used the same method. They input a compilation of special characters such as,)(*&^\<=>\ . if one of the combinations is changed in the response received and XSS attack is detected.

The number of webpages detected by the various tools was not the same. This is simply because the WVS use different crawling methods. Some of the tools used the POST method while others used the GET method.

In summary different opens source tools use different methods to discover the same types of vulnerabilities. For this reason, different results were realized.

CHAPTER 5 : CONCLUSION AND RECOMMENDATIONS

From the analysis of the data collected, the following discussions, conclusions and recommendations were arrived at. These conclusions and recommendations were focused on addressing the objectives of this study.

5.1 Mapping Research Objectives to the Methodology

The table below illustrates how the research objectives were achieved

Table 5:1Mapping research objectives to the methodology

Research Objectives	How they were achieved
To identify different open source vulnerabilities scanning tools for web application	Literature review was done on existing tools. Six tools were chosen depending on various factors.
To analyze the tools against set metrics	The following metrics to evaluate the tools were identified: -Time taken to scan web applications -Detection accuracy -Consistency and reliability
To study algorithm for these tools	Reviewed the underlying algorithm used by existing tools
To propose an improved hybrid algorithm	Designed a hybrid algorithm
To test and validate the hybrid algorithm	A program was developed to simulate the functionality of the algorithm. This program was subjected to the same test and compared its performance with the selected open source web scanning tools.

5.2 Limitations

During the study the following limitations were encountered:

i. The configuration of web scanning tools. Some of the tools required to be configured using different settings to ensure that they work perfectly. This proofed to be a dauting task mailnly due to poor documentation.

ii. Installation of the web applications on the local PC and configuring diffreent web servers. All the web applications required to be set up on a diffrernt PC , if more than

one web based application was installed on the same computer, the applications would conflict and would not function properly.

iii. Testing the tools under similar environment, this was achieved by creating virtual machines with the same specifications and configurations interms of operaring systems, CPU, and RAM and HDD

iv. During the scanning process some of the tools sampled would hang at one point or another for no apparent reason.

v. Resources on the testing enviroment. Since the experiments were carried out using virual machines, setting up the virtual enviroment on the testing PC proofed to be a dauting task. To over come this, virtual machines that were not in use were switched off so that the resources were free to be used by a different VM running at the moment.

vi. The development of the simulation was a very complex task

5.3 Conclusion

The open source tools have the capacity to detect vulnerabilities in the test cases performed. However, none of the tools have the capacity to detect all the vulnerabilities. The same conclusion was arrived at by McQuade (2014). The researcher concluded that there is no "silver bullet" in this area of Information Technology or any other black box vulnerability tools do not have the capacity to discover all the web vulnerabilities as identified for comparison purposes by WAVSEP

Conclusion on specific tools

wapiti- produced impressive results, with a fairly easy to interpret the report. As a matter of fact, it reported the highest numbers of SQL injections in the WebGoat application.

W3af –was able to discover many vulnerabilities, however, it did not produce excellent test results in any category.

Websecurify offers a user-friendly graphical user interface (GUI), this makes it very easy to use. However it does not detect some of the top ten list of OWASP. For instance, it failed to detect blind SQL injection in some applications.

Vega - provides one of the best reports when compared to all the other tools used in this study the vulnerabilities detected are classified into four categories namely: high, medium, low and info. See appendix for a sample of Vega report. This is categorization is very useful and provides a guide to the user on the vulnerabilities that should be given priority when sealing the weaknesses. The tool is easy to use and provides a user friendly graphical user interface.

Zed Attach Proxy popularly known as Zap takes a long time to scan the applications. However it's able to discover some vulnerabilities.

Ck AppScan generated better results overall when compared to other tools used, although the tool has higher detection accuracy when compared to the other tool. The only drawback is that it was reported to take a longer time to scan than most of the web scanners that were used in this study. Its performance is not 100% accurate but it has a higher capacity to detect more vulnerabilities when compared to the other tools.

Conclusion about the hybrid algorithm

The proposed hybrid algorithm is extensive in the execution of its detection mechanism against web application vulnerabilities. The proposed hybrid algorithm reports more vulnerabilities and presents a proficient manner while reporting discovered vulnerabilities. However since the proposed hybrid algorithm did not scan 100% of the existing vulnerabilities. There is need to increase the algorithm crawling component in order to ensure that it executed "deep" crawling. In addition the results presented shows that the proposed algorithm needs to be optimised to do the scanning in a short period of time. More research is needed to come up with a sophisticated algorithm that has the capacity to detect more vulnerabilities.

4.5 Suggestions for Further Research

i. **Suggest a solution to the vulnerabilities discovered**

After analyzing the reports of all the tools used in this study, none of the tools sampled have suggested has a remedy for the vulnerabilities reported. In my own opinion, it would be prudent to suggest the way the source code should be structured to fix the vulnerabilities detected.

ii. **Improved fuzzing component**

Since the hybrid algorithm fails to detect all the vulnerabilities, there is a need to use a more advanced logic in the fuzzing component of the algorithm to get getter results. The fuzzing component is responsible for firing the necessary inputs to determine whether vulnerabilities exist or not. The fuzzing logic used by Ck AppScan should be improved further to increase the detection accuracy.

iii. Reduced scanning time

The tool developed by the researcher takes a long period of time to scan a web based application. For this reason, it is important to improve the overall scanning mechanisms of the hybrid algorithm and reduce the scanning time without compromising on the detection accuracy.

References

Alssir, F. T., & Ahmed, M. (2012). Web Security Testing Approaches: Comparison Framework. In Proceedings of the 2011 2nd International Congress on Computer Applications and Computational Science (pp. 163-169). Springer Berlin Heidelberg.

Antunes & Vieira (2012). Defending against web application vulnerabilities. Computer, (2), 66-72.

Bau, J., Bursztein, E., Gupta, D., & Mitchell, J. (2010). State of the art: Automated black-box web application vulnerability testing. In Security and Privacy (SP), 2010 IEEE Symposium on (pp. 332-345). IEEE.

Chen, S. (2014). wavsep. Available: http://sectooladdict.blogspot.com/2014/02/wavsep-web-application-scanner.html. [Accessed 09 July 2015.]

Dessiatnikoff, A., Akrout, R., Alata, E., Kaaniche, M., & Nicomette, V. (2011). A clustering approach for web vulnerabilities detection. InDependable Computing (PRDC), 2011 IEEE 17th Pacific Rim International Symposium on (pp. 194-203). IEEE.

Dougherty, C. (2012).Practical Identification of SQL Injection Vulnerabilities. 2012. US-CERT-United States Computer Emergency Readiness Team. Citado na, 34. . [Accessed: 08th June 2015]

Doupe, A., Cova, M., & Vigna, G. (2010). Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. In Detection of Intrusions and Malware, and Vulnerability Assessment (pp. 111-131). Springer Berlin Heidelberg. [Accessed: 10th June 2015]

Fonseca, J., Vieira, M., & Madeira, H. (2014). Evaluation of Web Security Mechanisms using Vulnerability & Attack Injection. Dependable and Secure Computing, IEEE Transactions on, 11(5), 440-453.

Granville, K . (2015).Nine Recent Cyber-attacks against Big Businesses. New York Times [online] Available from : http://www.nytimes.com/interactive/2015/02/05/technology/recent-cyberattacks.html?_r=1. [Accessed 08 July 2015.]

Howard, M., LeBlanc, D., & Viega, J. (2010). 24 deadly sins of software security [electronic book]: Programming flaws and how to fix them. New York: McGraw-Hill.

Jovanovic, N., Kruegel, C., & Pixy, E. K. (2010). A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In Proceedings of the 2006 IEEE symposium on Security and Privacy, Washington, DC, IEEE Computer Society (pp. 258-263).

Kalman., G. (2014). Ten Most Common Web Security Vulnerabilities.[online] Available from: http://www.toptal.com/security/10-most-common-web-security-vulnerabilities [Accessed 08 July 2015.]

Kals, S., Kirda, E., Kruegel, C., & Jovanovic, N. (2014). A web vulnerability scanner. In Proceedings of the 15th international conference on World Wide Web (pp. 247-256). ACM.

Khoury, N., Zavarsky, P., Lindskog, D., & Ruhl, R. (2011). Testing and assessing web vulnerability scanners for persistent SQL injection attacks. In Proceedings of the First International Workshop on Security and Privacy Preserving in e-Societies (pp. 12-18). ACM.

Kothari, C. R. (2009). Quantitative Techniques, 3E. Vikas publishing house PVT LTD

McQuade, K. (2014). Open Source Web Vulnerability Scanners: The Cost Effective Choice?. In Proceedings of the Conference for Information Systems Applied Research ISSN (Vol. 2167, p. 1508). [Accessed: 18th June 2015]

Mirjalili, M., Nowroozi, A., & Alidoosti, M. (2014). A survey on web penetration test.

Mugenda, O. Mugenda (2009) Research Methods: Quantitative and Qualitative Approaches. Nairobi: ACTS.

Myers, G. J., Sandler, C., & Badgett, T. (2011). The art of software testing. John Wiley & Sons.

Nagpal, B., Chauhan, N., & Singh, N. (2015). Defending Against Remote File Inclusion Attacks on Web Applications. i-Manager's Journal on Information Technology, 4(3), 25.

Park, N. (2015). Detection Experimentation and Validation of Web Applications using Both Static and Dynamic Analysis. International Information Institute (Tokyo). Information, 18(5 (A)), 1735.

Tripathi, A., & Singh, U. K. (2011). On prioritization of vulnerability categories based on CVSS scores. In Computer Sciences and Convergence Information Technology (ICCIT), 2011 6th International Conference on (pp. 692-697). IEEE

Saunders, M. N., Saunders, M., Lewis, P., & Thornhill, A. (2011). Research methods for business students, 5/e. Pearson Education India.

Sekaran, U. (2011). Research methods for business: A skill building approach. John Wiley & Sons.

Shelly, D.A. (2010) .Using a Web Server Test Bed to Analyse the Limitations of Web Application Vulnerability Scanners. Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia. [Accessed: 10th June 2015]

Shema. M, (2011). Web Application Security for Dummies. England: John Wiley & Sons Ltd. P27-68.

Snyder, B. (2014). 5 huge cyber security breaches at companies you know. Available from: http://fortune.com/2014/10/03/5-huge-cybersecurity-breaches-at-big-companies/. [Accessed 08 July 2015.]

Stuttard, D., & Pinto, M. (2011). The web application hacker's handbook: discovering and exploiting security flaws. John Wiley & Sons. Inc. p33-80, p200-243.

Van der Loo, F. (2011). Comparison of penetration testing tools for web applications (Doctoral dissertation, Master thesis, Radboud University Nijmegen. http://www. ru. nl/publish/pages/578936/frank_van_der_loo_scriptie. pdf).[Accessed: 08th June 2015]

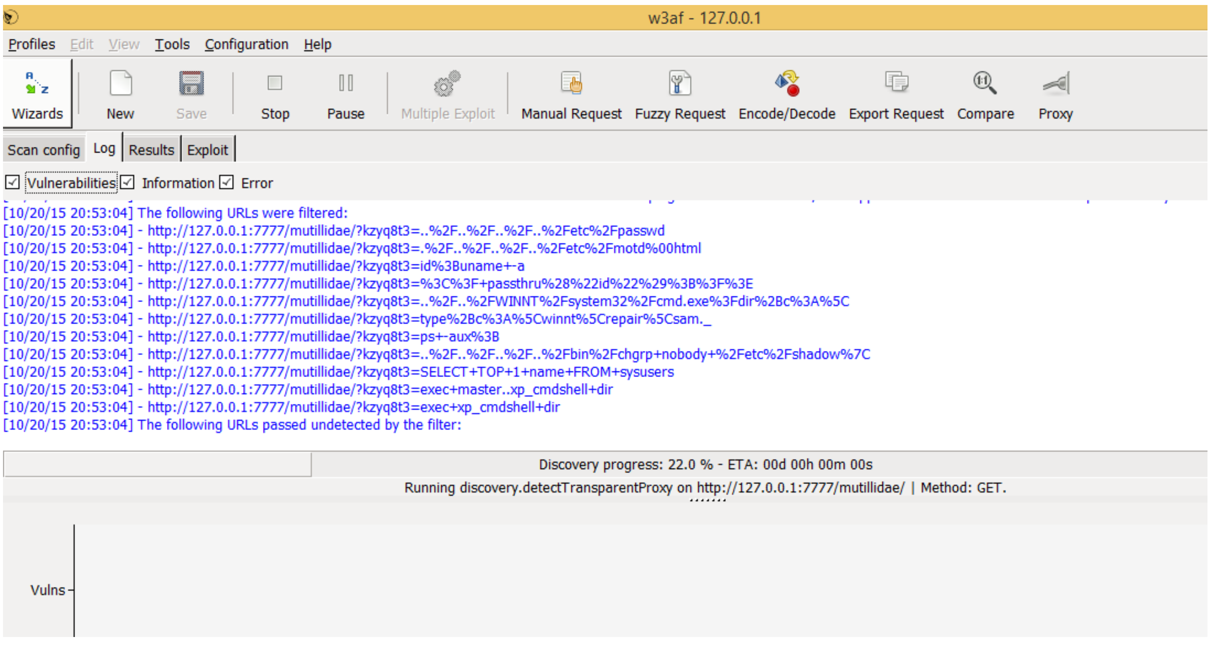
WhiteHat Security team. (2015). WhiteHat Security Statistics Report 2015. Available From: https://www.whitehatsec.com/statistics-report/featured/2015/05/21/statsreport.html. [Accessed 09 July 2015.]

Yu, Y., Yang, Y., Gu, J., & Shen, L. (2011). Analysis and suggestions for the security of web applications. In Computer Science and Network Technology (ICCSNT), 2011 International Conference on (Vol. 1, pp. 236-240). IEEE.

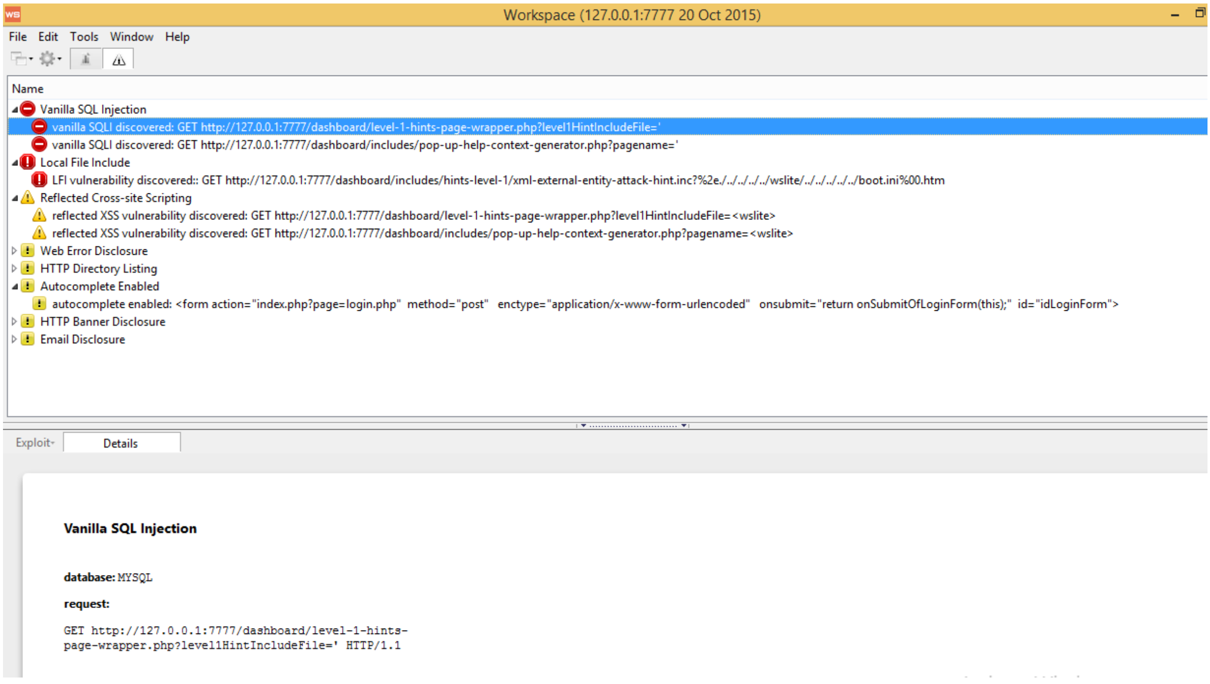
Appendices

Appendix 1: Screen shot captured during the scanning process

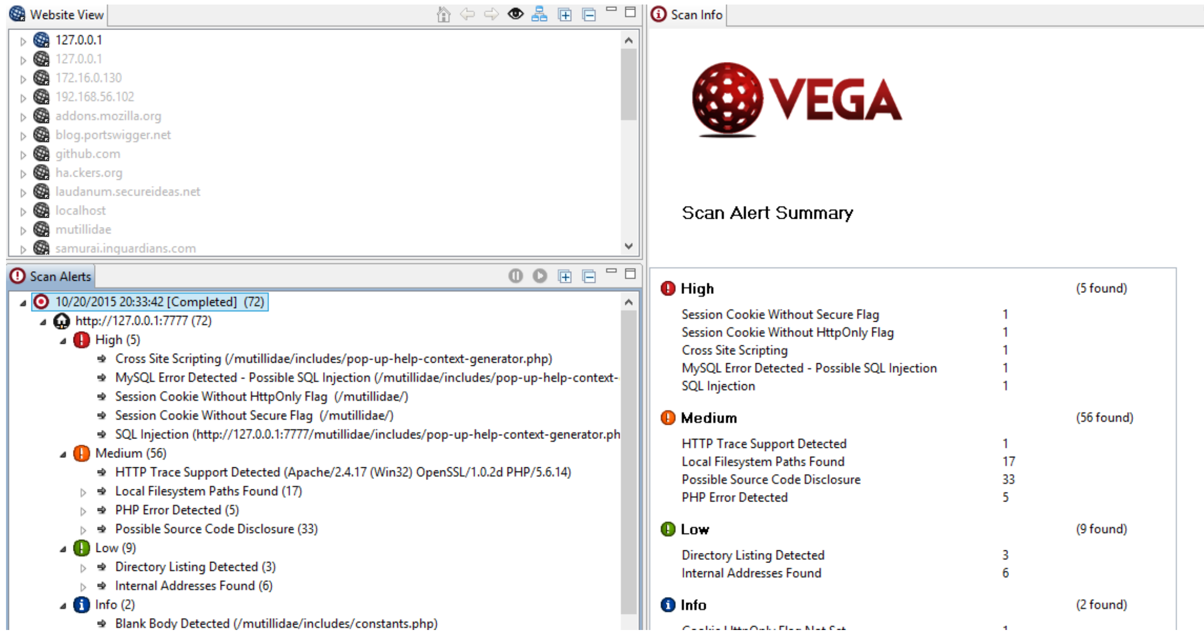
W3AF Scanning report.



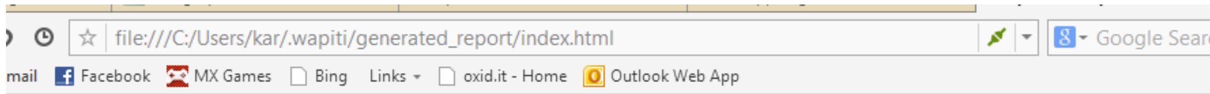
Websecurify scanning report



Vega's scanning report



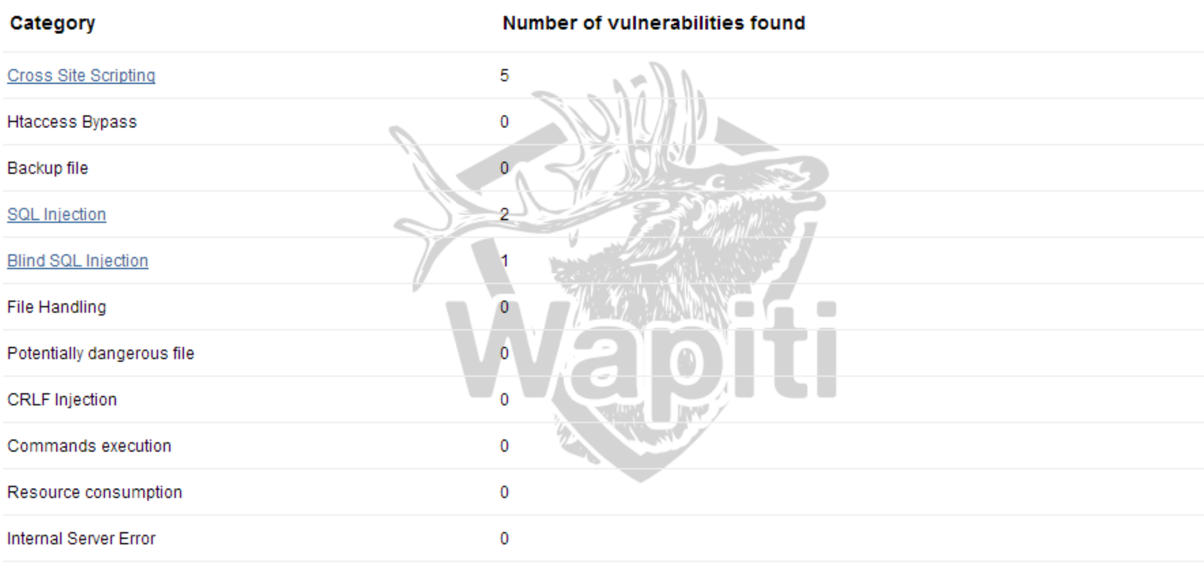
Wapiti scanning report for Mutillidae



Wapiti vulnerability report for http://127.0.0.1:7777/mutillidae

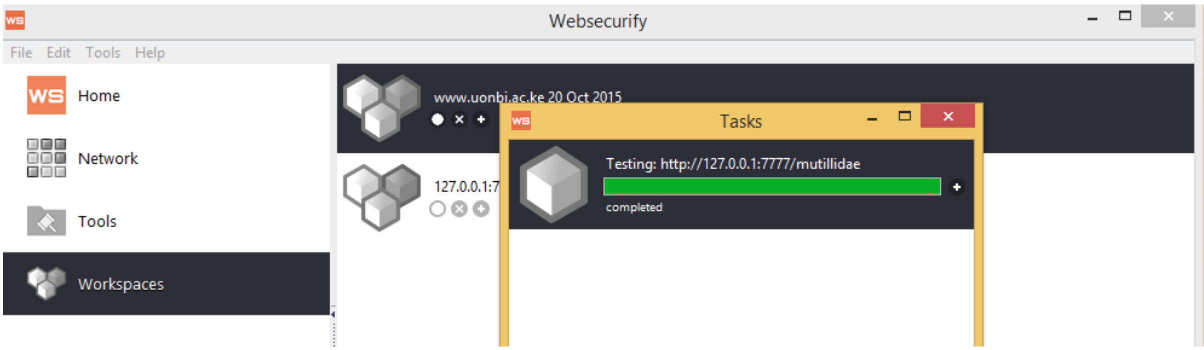
Date of the scan: Tue, 20 Oct 2015 15:13:25 +0000. Scope of the web scanner : folder

Summary



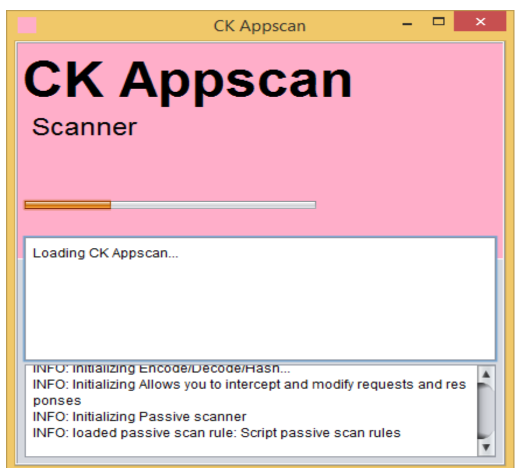
Category	Number of vulnerabilities found
Cross Site Scripting	5
Htaccess Bypass	0
Backup file	0
SQL Injection	2
Blind SQL Injection	1
File Handling	0
Potentially dangerous file	0
CRLF Injection	0
Commands execution	0
Resource consumption	0
Internal Server Error	0

Websecurify Scanning in progress

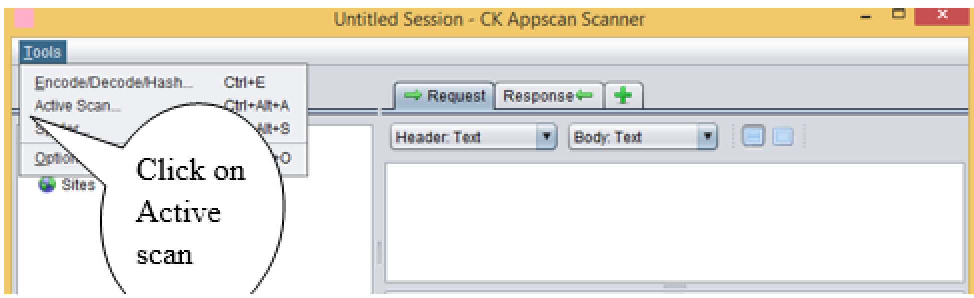


Appendix 2: User manual

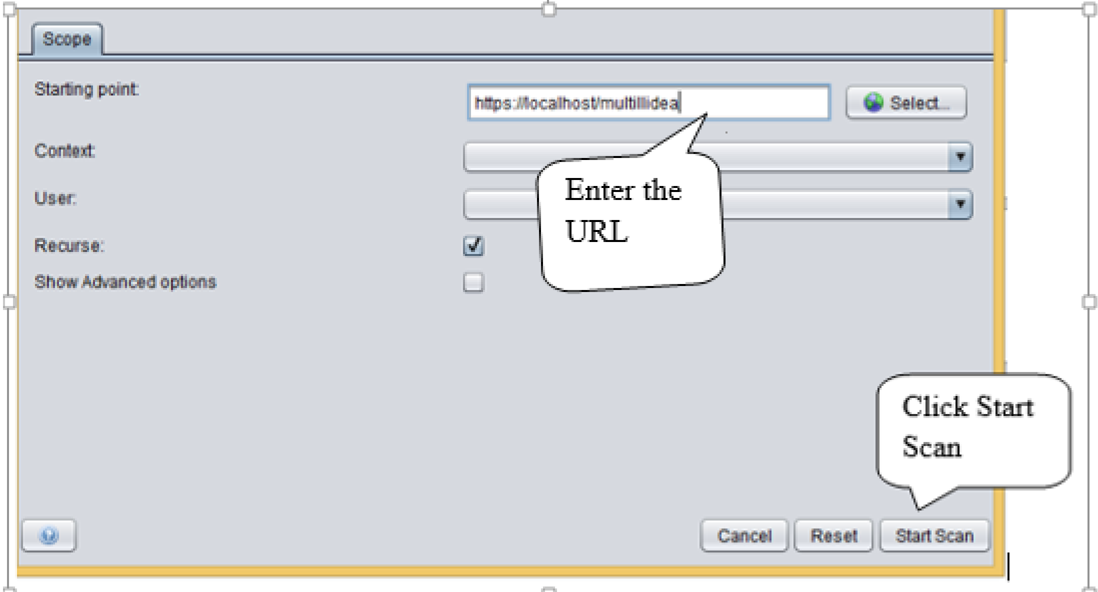
Click on the CK AppScan Icon on the desktop



From the tools menu, select active scan



Enter the URL or path for the application to be scanned and click on the start scan button. A report of the vulnerabilities found will be displayed.



Appendix 3: Source code samples

SQL Injection Detection

Initialization of SQL Characters

```
public static final String SQL_ONE_LINE_COMMENT = " -- ";
private static final String[] SQL_CHECK_ERR = {"'", "\"", ";", ")", "(",
"NULL", "'\""};
```

Creation of Maps to preserve SQL Error Messages

Two maps are created. One to preserver specific database error messages (mysql, sql, oracle etc) and the other one to preserve generic database error messages

```
private static final Map<Pattern, String> SQL_ERROR_TO_SPECIFIC_DBMS = new
LinkedHashMap<>();

private static final Map<Pattern, String> SQL_ERROR_TO_GENERIC_DBMS = new
LinkedHashMap<>();
```

Initialization of Error Values

Microsoft SQL Server Error Messages

```
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\Qcom.mysql.jdbc.exception
s\\E", PATTERN_PARAM), "MySQL");
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\Qorg.gjt.mm.mysql\\E",
PATTERN_PARAM), "MySQL");
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\QThe used SELECT
statements have a different number of columns\\E", PATTERN_PARAM),
"MySQL");
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\Qcom.microsoft.sqlserver.
jdbc\\E", PATTERN_PARAM), "Microsoft SQL Server");
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\Qcom.microsoft.jdbc\\E",
PATTERN_PARAM), "Microsoft SQL Server");
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\Qcom.inet.tds\\E",
PATTERN_PARAM), "Microsoft SQL Server");
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\Qcom.microsoft.sqlserver.
jdbc\\E", PATTERN_PARAM), "Microsoft SQL Server");
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\Qcom.ashna.jturbo\\E",
PATTERN_PARAM), "Microsoft SQL Server");
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\Qweblogic.jdbc.mssqlserve
r\\E", PATTERN_PARAM), "Microsoft SQL Server");
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\Q[Microsoft]\\E",
PATTERN_PARAM), "Microsoft SQL Server");
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\Q[SQLServer]\\E",
PATTERN_PARAM), "Microsoft SQL Server");
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\Q[SQLServer 2000 Driver
for JDBC]\\E", PATTERN_PARAM), "Microsoft SQL Server");
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\Qnet.sourceforge.jtds.jdb
c\\E", PATTERN_PARAM), "Microsoft SQL Server");
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\Q80040e14\\E",
PATTERN_PARAM), "Microsoft SQL Server");
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\Q800a0bcd\\E",
PATTERN_PARAM), "Microsoft SQL Server");
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\Q80040e57\\E",
PATTERN_PARAM), "Microsoft SQL Server");
```

Oracle error messages

```
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\Qoracle.jdbc\\E",
PATTERN_PARAM), "Oracle");
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\QSQLSTATE[HY\\E",
PATTERN_PARAM), "Oracle");
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\QORA-00933\\E",
PATTERN_PARAM), "Oracle");
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\QORA-06512\\E",
PATTERN_PARAM), "Oracle");  //indicates the line number of an error
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\QSQL command not properly
ended\\E", PATTERN_PARAM), "Oracle");
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\QORA-00942\\E",
PATTERN_PARAM), "Oracle");  //table or view does not exist
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\QORA-29257\\E",
PATTERN_PARAM), "Oracle");  //host unknown
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\QORA-00932\\E",
PATTERN_PARAM), "Oracle");  //inconsistent datatypes
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\Qquery block has
incorrect number of result columns\\E", PATTERN_PARAM), "Oracle");
SQL_ERROR_TO_SPECIFIC_DBMS.put(Pattern.compile("\\QORA-01789\\E",
PATTERN_PARAM), "Oracle");
```

Generic error messages

```
SQL_ERROR_TO_GENERIC_DBMS.put(Pattern.compile("\\Qcom.ibatis.common.jdbc\\E
", PATTERN_PARAM), "Generic SQL RDBMS");
SQL_ERROR_TO_GENERIC_DBMS.put(Pattern.compile("\\Qorg.hibernate\\E",
PATTERN_PARAM), "Generic SQL RDBMS");
SQL_ERROR_TO_GENERIC_DBMS.put(Pattern.compile("\\Qsun.jdbc.odbc\\E",
PATTERN_PARAM), "Generic SQL RDBMS");
SQL_ERROR_TO_GENERIC_DBMS.put(Pattern.compile("\\Q[ODBC Driver
Manager]\\E", PATTERN_PARAM), "Generic SQL RDBMS");
SQL_ERROR_TO_GENERIC_DBMS.put(Pattern.compile("\\QSystem.Data.OleDb\\E",
PATTERN_PARAM), "Generic SQL RDBMS");   //System.Data.OleDb.OleDbException
SQL_ERROR_TO_GENERIC_DBMS.put(Pattern.compile("\\Qjava.sql.SQLException\\E"
, PATTERN_PARAM), "Generic SQL RDBMS");
```

Scanner Code

```
public void scan(HttpMessage msg, String param, String origParamValue) {
//Note: the "value" we are passed here is escaped. we need to unescape it
before handling it.
//as soon as we find a single SQL injection on the url, skip out. Do not
look for SQL injection on a subsequent parameter on the same URL
//for performance reasons.
      boolean sqlInjectionFoundForUrl = false;
      String sqlInjectionAttack = null;
      HttpMessage refreshedmessage = null;
      String mResBodyNormalUnstripped = null;
      String mResBodyNormalStripped = null;
for (int sqlErrorStringIndex = 0;sqlErrorStringIndex < SQL_CHECK_ERR.length
&& !sqlInjectionFoundForUrl && doSpecificErrorBased &&
countErrorBasedRequests < doErrorMaxRequests; sqlErrorStringIndex++) {
String[] prefixStrings;
if (origParamValue != null) {
// Removed getURLDecode()
prefixStrings = new String[]{"", origParamValue};
} else {
prefixStrings = new String[]{""};
}
```

```
for (int prefixIndex = 0; prefixIndex < prefixStrings.length &&
!sqlInjectionFoundForUrl; prefixIndex++) {

//new message for each value we attack with
      HttpMessage msg1 = getNewMsg();
      String sqlErrValue = prefixStrings[prefixIndex] +
      SQL_CHECK_ERR[sqlErrorStringIndex];
      setParameter(msg1, param, sqlErrValue);

//System.out.println("Attacking [" + msg + "], parameter [" + param + "]
with value ["+ sqlErrValue + "]");

//send the message with the modified parameters
sendAndReceive(msg1, false); //do not follow redirects
countErrorBasedRequests++;

//now check the results against each pattern in turn, to try to identify a
database, or even better: a specific database.
//Note: do NOT check the HTTP error code just yet, as the result could come
back with one of various codes.
Iterator<Pattern> errorPatternIterator =
SQL_ERROR_TO_SPECIFIC_DBMS.keySet().iterator();

while (errorPatternIterator.hasNext() && !sqlInjectionFoundForUrl) {
Pattern errorPattern = errorPatternIterator.next();
String errorPatternRDBMS = SQL_ERROR_TO_SPECIFIC_DBMS.get(errorPattern);

//if the "error message" occurs in the result of sending the modified
query, but did NOT occur in the original result of the original query
//then we may may have a SQL Injection vulnerability
StringBuilder sb = new StringBuilder();
if (!matchBodyPattern(getBaseMsg(), errorPattern, null) &&
matchBodyPattern(msg1, errorPattern, sb)) {
//Likely a SQL Injection. Raise it
String extraInfo = Constant.messages.getString(MESSAGE_PREFIX +
"alert.errorbased.extrainfo", errorPatternRDBMS, errorPattern.toString());
//raise the alert, and save the attack string for the "Authentication
Bypass" alert, if necessary
sqlInjectionAttack = sqlErrValue;
bingo(Alert.RISK_HIGH, Alert.CONFIDENCE_MEDIUM, getName() + " - " +
errorPatternRDBMS, getDescription(),
null,
param, sqlInjectionAttack,
extraInfo, getSolution(), sb.toString(), msg1);

//log it, as the RDBMS may be useful to know later (in subsequent checks,
when we need to determine RDBMS specific behaviour, for instance)
getKb().add(getBaseMsg().getRequestHeader().getURI(), "sql/" +
errorPatternRDBMS, Boolean.TRUE);

sqlInjectionFoundForUrl = true;
continue;
}
//bale out if we were asked nicely
if (isStop()) {
log.debug("Stopping the scan due to a user request");
return;
} //end of the loop to check for RDBMS specific error messages

if (this.doGenericErrorBased && !sqlInjectionFoundForUrl) {
errorPatternIterator = SQL_ERROR_TO_GENERIC_DBMS.keySet().iterator();
```

```
while (errorPatternIterator.hasNext() && !sqlInjectionFoundForUrl) {
Pattern errorPattern = errorPatternIterator.next();
String errorPatternRDBMS = SQL_ERROR_TO_GENERIC_DBMS.get(errorPattern);

//if the "error message" occurs in the result of sending the modified
query, but did NOT occur in the original result of the original query
//then we may may have a SQL Injection vulnerability
StringBuilder sb = new StringBuilder();
if (!matchBodyPattern(getBaseMsg(), errorPattern, null) &&
matchBodyPattern(msg1, errorPattern, sb)) {
//Likely a SQL Injection. Raise it
String extraInfo = Constant.messages.getString(MESSAGE_PREFIX +
"alert.errorbased.extrainfo", errorPatternRDBMS, errorPattern.toString());
//raise the alert, and save the attack string for the "Authentication
Bypass" alert, if necessary
sqlInjectionAttack = sqlErrValue;
bingo(Alert.RISK_HIGH, Alert.CONFIDENCE_MEDIUM, getName() + " - " +
errorPatternRDBMS, getDescription(),
null,
param, sqlInjectionAttack,
extraInfo, getSolution(), sb.toString(), msg1);

//log it, as the RDBMS may be useful to know later (in subsequent checks,
when we need to determine RDBMS specific behaviour, for instance)
getKb().add(getBaseMsg().getRequestHeader().getURI(), "sql/" +
errorPatternRDBMS, Boolean.TRUE);

sqlInjectionFoundForUrl = true;
continue;
}
//bale out if we were asked nicely
if (isStop()) {
log.debug("Stopping the scan due to a user request");
return;
}
} //end of the loop to check for RDBMS specific error messages

}

}
} } }
```