**Cyber Security Project**

Introduction to Network Security Basics and Web Application Security

Prince Mitchell C.E

07-26-2025

Table of Contents

[Introduction 3](#_Toc204420838)

[Tools and methods used 3](#_Toc204420839)

[Steps taken to complete task 4](#_Toc204420840)

[SQL Injection 5](#_Toc204420841)

[Descriptions provided by OWASP ZAP 6](#_Toc204420842)

[Cross-Site Scripting (XSS) 7](#_Toc204420843)

[Descriptions provided by OWASP ZAP 7](#_Toc204420844)

[Cross-Site Request Forgeries (CSRF) 8](#_Toc204420845)

[Descriptions provided by OWASP ZAP 10](#_Toc204420846)

[Challenges faced and how they were overcome 11](#_Toc204420847)

[Results and outcomes 11](#_Toc204420848)

# Introduction

In today's digital world, web applications are a very essential tool we use in our day-to-day life. We have them on our phones, laptops, servers, IoT, and most web applications contain sensitive information, which are common targets for cyberattacks. Understanding how web applications can be exploited is a critical step toward building secure systems. This project will explore common web application vulnerabilities by analyzing an intentionally vulnerable web application using OWASP ZAP to scan for these vulnerabilities. Through hands-on investigation, insight into how attackers identify and exploit security flaws such as SQL Injection, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF) will be gained. This exercise will not only highlight the importance of secure coding practices but also recognize and defend against real-world threats in web environments.

# Tools and methods used

The task was to install and set up WebGoat, a vulnerable web application on the local machine and to use OWASP ZAP to scan, record traffic, inspect traffic, modify requests and responses to and from web application to get reports on a range of known vulnerabilities from the web application, focusing on identifying at least one instance each of SQL Injection, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF). However, the following tools were launched on a Kali Linux Virtual Machine. It is always advised to perform in a safe environment or a sandbox to avoid unknown attacks and unknown vulnerabilities when analyzing and scanning for vulnerabilities in a network.

A computer screen shot of a program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

# Steps taken to complete the task

After the download of OWASP ZAP, a few configurations were performed to have the ZAP application scan and provide detailed information about URLs entered in the browser. A proxy server was created with the name localhost to use port =1234, which created the connection between the browser and the ZAP application. The following step was to configure the local server on the ZAP app to match with port and name created on the proxy server.

A screenshot of a computer

AI-generated content may be incorrect.

The second step was to copy the URL provided by WebGoat after launching on Kali Linux (127.0.0.0.1:8080/WebGoat) and to paste it into the ZAP app under automated scan. There were other options like manual exploration, but the automated scan was chosen because it is already set to connect to the web browser. Another configuration performed was to add the URL to the scope and context to give additional focus, and the next was to select a mode to perform the scan. Standard mode was selected to actively scan the web application for vulnerabilities. After a long wait, the scan was completed, and the following step was to select the Alert section, in this section were detailed Alert flags of vulnerabilities and their descriptions were provided by OWASP Zap from the web application. There were eight notified alerts, three medium priority, two low priority alerts, and three informational priority alerts, each displaying a description of the vulnerability and a remedy to counter that. Among these vulnerabilities displayed were a performed SQL Injection, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF).

A screenshot of a computer

AI-generated content may be incorrect.

## SQL Injection

Few vulnerabilities were manually exploited using SQL basic Knowledge and techniques on the web application, which is necessary for the task. One of the samples performed described a company table named Employees with rows and columns, detailing the user\_ID, first\_name, last\_name, and authentication number saved in the database. The task was to fetch for the department of Bob Franco. With the SQL query typed into the input field = select department from employees where first\_name = 'Bob' and last\_name= 'Franco'; the department name marketing, for Bob Franco, was successfully retrieved from the database. The following screenshots display more samples, statically performed SQL Injections followed by a successful response.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

## Descriptions provided by OWASP ZAP

The alerts provided by OWASP ZAP describe the cause of the vulnerability, which states that no known anti-CSRF token was found in the following HTML form. It also defines the vulnerability to be a low risk, missing an anti-CSRF token and provides solutions, one of these remedies suggests the use of a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid, and not to use the GET method for any request that triggers a state change. It also provided links referencing cheat sheets to disable vulnerability to keep sensitive data safe.

A computer screen with text

AI-generated content may be incorrect.

## Cross-Site Scripting (XSS)

This is the second type of vulnerability that was manually exploited using an HTML query, by creating a script to perform tasks that were not the developer’s original intent. Cross-Site Scripting can result in stealing session cookies, creating false requests and false fields on a page to collect credentials and perform many more attacks. In one of the samples relating to XSS, the task was to identify an input field vulnerable to a crafted XSS attack. After performing on but input fields, the first input field was discovered to be vulnerable after the HTML script = <script> alert (“Cyber Security Internship”) </script> was inserted. The screenshot below displays the response from the web application. This is how sensitive information gets leaked without the user’s consent.

A screenshot of a computer

AI-generated content may be incorrect.

## Descriptions provided by OWASP ZAP

The alerts provided by OWASP ZAP describe how the weakness was found on the web application after it was manually exploited. First, it identified the vulnerability to be a low-risk alert with a missing X-Content-Type-Options Header. It described the problem to be from the Anti-MIME-Sniffing header X-Content-Type-Options, which was not set to 'no sniff'; this allowed older versions of Firefox or any web browsers to perform MIME-sniffing on the response body, potentially causing the response body to be interpreted. It further provided more information and description of the issue, backing it up with a solution to discard the found vulnerability, one of which suggests ensuring an appropriate set of the Content-Type header to “no sniff” to all web pages and the web application servers. It also stated that the standard-compliant and updated web browsers resist performing any type of sniffing, ensuring data safety. Below is a screenshot detailing the response section and the alerts from OWASP ZAP.

A screenshot of a computer

AI-generated content may be incorrect.

## Cross-Site Request Forgeries (CSRF)

The vulnerability of CSRF has similar attacks to XSS. With XSS, the attacker focuses on his/her targets by sending the crafted script through a link. This target can be a person, a website, or a server compromising the trust, but with CSRF, the crafted link is sent through a victim trusted by a website or a web application. This kind of attack is difficult to detect and can cause serious damage if the website or target is vulnerable. In one of the samples exploited manually, the task was to send a crafted HTML query from a different host to the target. With the crafted query (<form class="attack-form" accept-charset="UNKNOWN" id="csrf-review" method="POST" name="review-form" successcallback="" action="http://localhost:8080/WebGoat/csrf/review"><input type="hidden" name="reviewText" type="text" value="PseudoTime!"><input type="hidden" name-"stars" type= “text” value="text" value="1">

<input type="hidden" name="validateReq" value="2aa14227b9a13d0bede0388a7fba9aa9"> <input type="submit" name="submit" value="Submit review">) inserted into the input field from a different host, Beiring 127.0.0.1:9090 to the target 127.0.0.1:8080, sending a hidden message, which is unknown to the victim. The screenshots display another example of this attack and how sensitive information gets leaked without the user’s consent.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A cat with a white collar

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

## Descriptions provided by OWASP ZAP

The alerts provided by OWASP ZAP describe how the weak spot was found on the web application after it was manually exploited. First, it identified the vulnerability to be a low-risk alert with a missing X-Content-Type-Options Header. It described the problem to be from the Anti-MIME-Sniffing header X-Content-Type-Options, which was not set to 'no sniff'; this allowed older versions of Firefox or any web browsers to perform MIME-sniffing on the response body, potentially causing the response body to be interpreted. It further provided more information and description of the issue, backing it up with a solution to discard the found vulnerability, one of which suggests ensuring an appropriate set of the Content-Type header to “no sniff” to all web pages and the web application servers. It also stated that the standard-compliant and updated web browsers resist performing any type of sniffing, ensuring data safety.

A screenshot of a computer

AI-generated content may be incorrect.

# Challenges faced and how they were overcome

At the beginning of the task, getting the web application downloaded on Kali Linux was the first encounter. After several trials, it was later discovered that the operating system was running on the old version of **Java SE Development Kit. Downloading** the latest version solved the first issue. After the launch of **OWAP ZAP**, the next step is configure it to listen, scan and detect any URLs from a browser on a local port was difficult, having the idea and not the solution was the second encounter, to get the solution, watching videos relating to the situation on YouTube, by creating a Firefox proxy server and performing the necessary configurations, the issue was dissolved. Another encounter was having to shut down ports 8080 and 9090 each day, when trying to analyze the request and response from the web application. Unfortunately, that was the only remedy to getting the port free for a fresh connection. The most encountered faced was trying to manually exploit vulnerabilities in the web application after several trials, but watching videos relating to the situation on YouTube helped gain more knowledge and experience.

# Results and outcomes

After analyzing each request and response from the web application, thirty-five was the total record of alerts were detected after manually performing the exploits. OWASP ZAP displayed other alerts, detailing other hidden vulnerabilities in the web application, which is a must to look out for. Understanding the descriptions and applying the solutions will help mitigate vulnerabilities in the future, ensuring the security of sensitive data from web attackers.