SECURITY ASSESSMENT REPORT

Task 1 – Web Application Security Testing

Internship: Future Interns

Intern Name: Nilesh kisan Chavan Date: August 2025

Table of Contents

---

1. Introduction

The objective of this project was to conduct a vulnerability assessment on a deliberately vulnerable web application using OWASP security standards. As part of the internship program, we analyzed common web vulnerabilities and learned how malicious hackers exploit weaknesses in web applications. The findings were compiled into this professional security report.

---

2. Tools and Environment

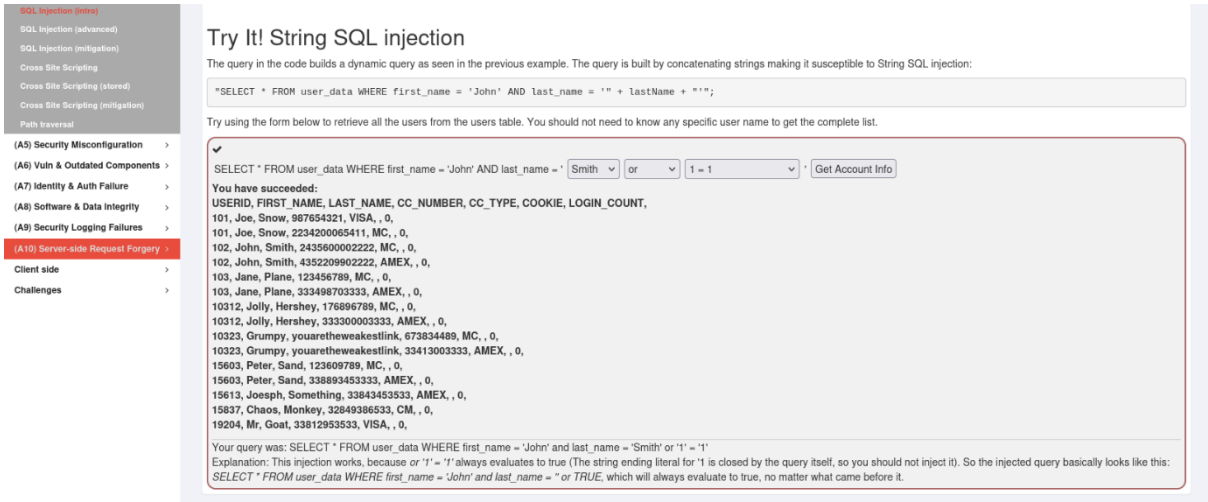
Vulnerable Web Application: WebGoat

Operating System: Kali Linux (Virtual Machine)

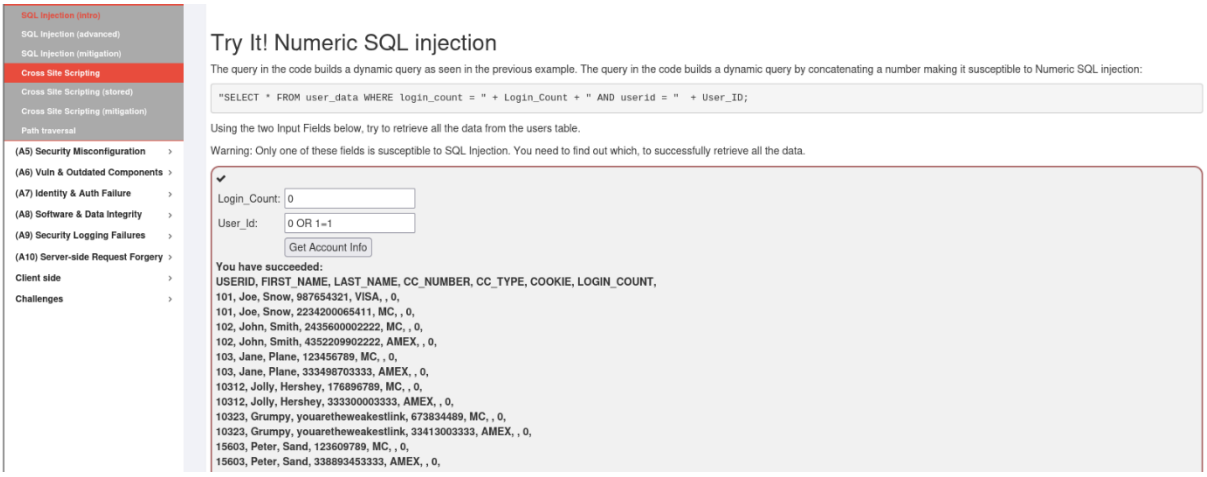
Security Tools Used: o OWASP ZAP (Scanning & passive analysis) o Web Browser (manual payload testing_

3. Vulnerability Assessments

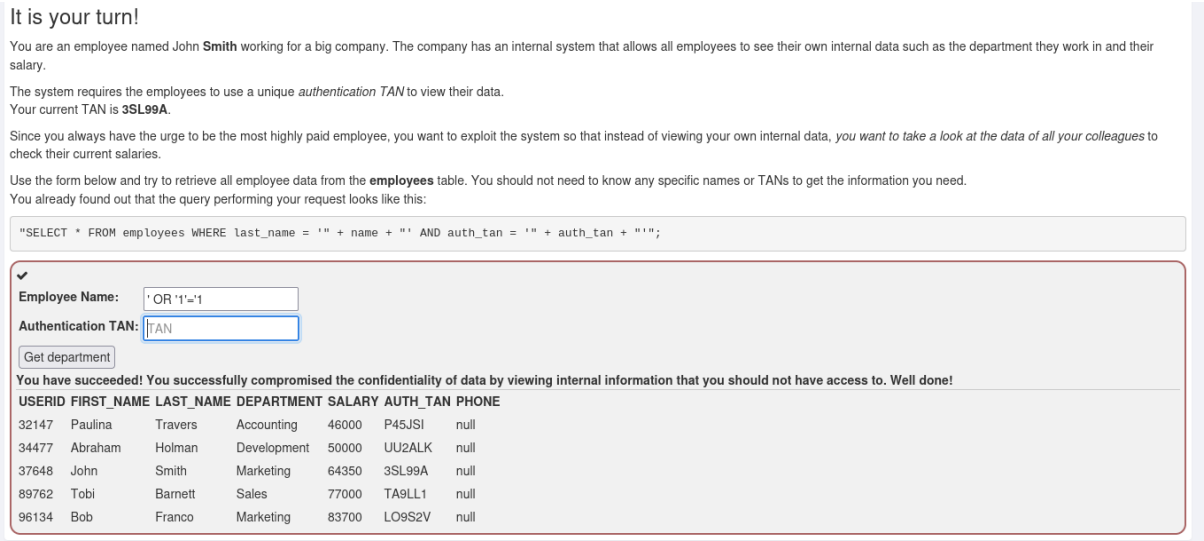
1) Vulnerability: Simple SQL Injection:



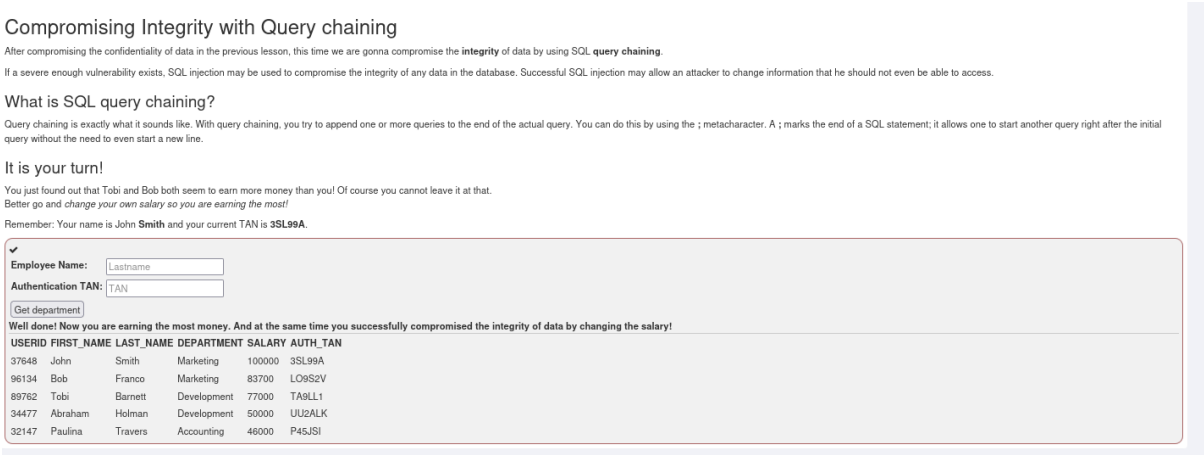
2) Vulnerability: Numeric SQL Injection:



3) Vulnerability: String SQL Injection with Comment:



4) Vulnerability: Compromising Integrity via Query Chaining:



Compromising Integrity with Query chaining

After compromising the confidentiality of data in the previous lesson, this time we are gonna compromise the **integrity** of data by using SQL **query chaining**.

If a severe enough vulnerability exists, SQL injection may be used to compromise the integrity of any data in the database. Successful SQL injection may allow an attacker to change information that he should not even be able to access.

What is SQL query chaining?

Query chaining is exactly what it sounds like. With query chaining, you try to append one or more queries to the end of the actual query. You can do this by using the **;** metacharacter. A **;** marks the end of a SQL statement; it allows one to start another query right after the initial query without the need to even start a new line.

It is your turn!

You just found out that Tobi and Bob both seem to earn more money than you! Of course you cannot leave it at that.
Better go and *change your own salary so you are earning the most!*

Remember: Your name is John **Smith** and your current TAN is **3SL99A**.

✔

Employee Name: Lastname

Authentication TAN: TAN

Get department

Well done! Now you are earning the most money. And at the same time you successfully compromised the integrity of data by changing the salary!

USERID	FIRST_NAME	LAST_NAME	DEPARTMENT	SALARY	AUTH_TAN
37648	John	Smith	Marketing	100000	3SL99A
96134	Bob	Franco	Marketing	83700	LO9S2V
89762	Tobi	Barnett	Development	77000	TA9LL1
34477	Abraham	Holman	Development	50000	UU2ALK
32147	Paulina	Travers	Accounting	46000	P45JSI

Mitigation Summary :

- Use prepared statements and parameterized queries.
- Use Web Application Firewalls (WAFs) and secure coding practices.
- Implement input validation and whitelisting.
- Employ least privilege principle in database roles.

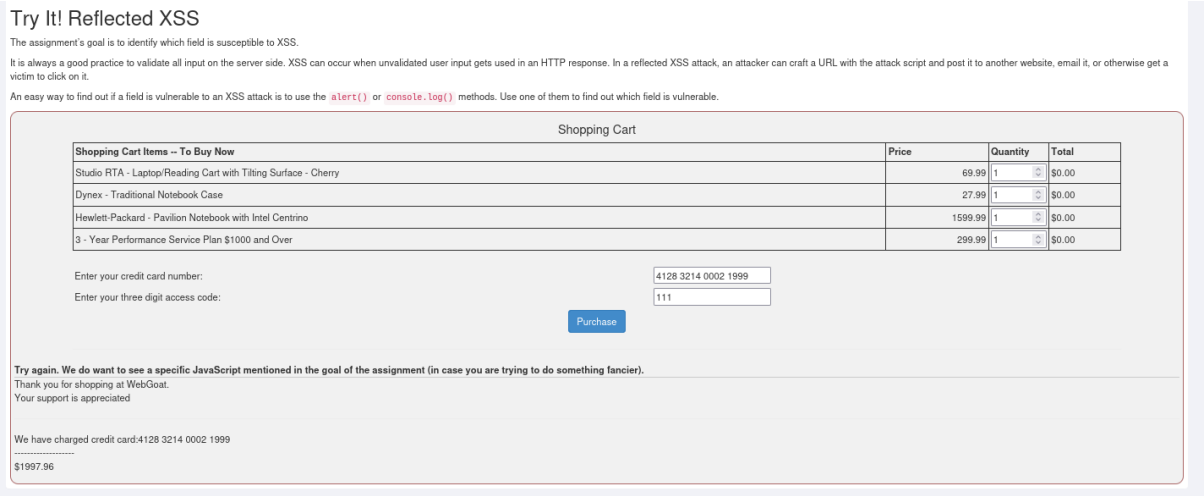
Cross Site Scripting (XSS)

1)Vulnerability: Reflected XSS

How Discovered: Manual test by passing script payload in URL or search field.

Why It's Dangerous: Can be used to steal session cookies or perform actions on behalf of the user.

Mitigation: Encode output using HTML entity encoding; validate and sanitize input.

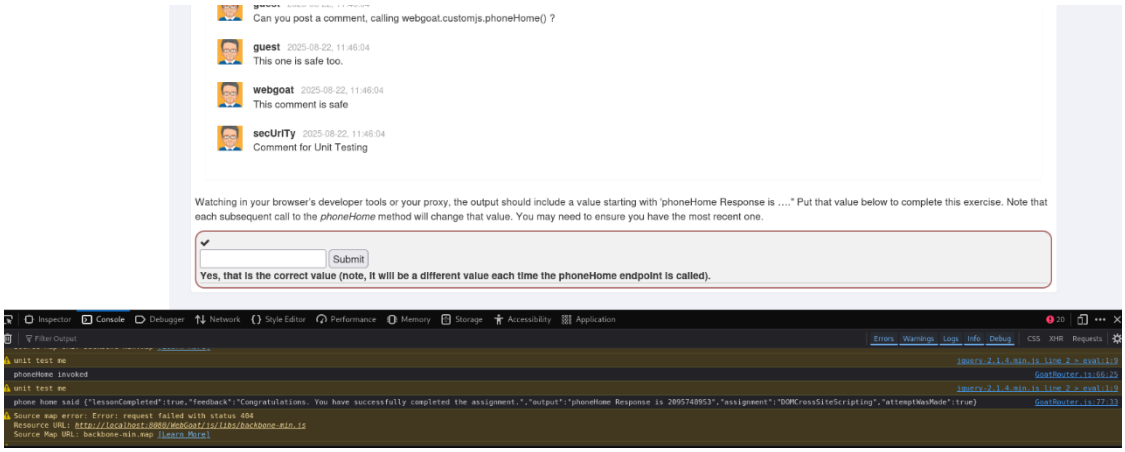


2)Vulnerability: Stored XSS

How Discovered: Input was stored and later executed when viewing the message or comment.

Why It's Dangerous: Auto-executes whenever data is loaded, affects every user who accesses that page.

Mitigation: Sanitize input on entry and encode on output; use CSP (Content Security Policy).

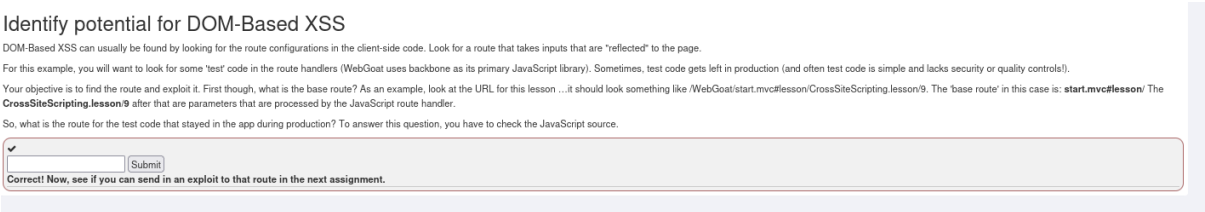


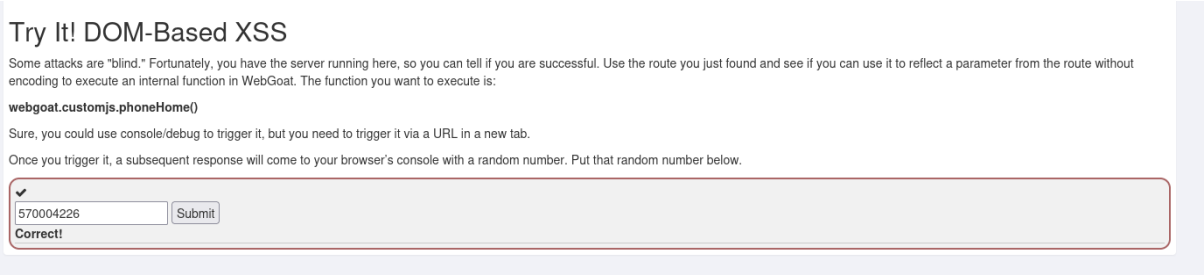
3)Vulnerability: DOM-Based XSS

How Discovered: Observed that the JavaScript handled input from the URL fragment without sanitization.

Why It's Dangerous: Attacker can modify page content or perform actions in user context without reloading the page.

Mitigation: Use secure JavaScript libraries; sanitize input within the DOM; avoid unsafe DOM manipulations.





Mitigation Summary :

Sanitize and validate all user inputs, both client- and server-side.

Encode output based on context (HTML, JavaScript, URL).

Implement Content Security Policy (CSP) to restrict execution of inline scripts.

Avoid directly injecting user input into the DOM.

Use secure frameworks and libraries that automatically handle XSS defense (e.g., React, Angular).

Cross Site Request Forgery(CSRF)

Module : Cross-site Request Forgery(CSRF)

The purpose of this exercise is to understand and exploit CSRF (Cross-Site Request Forgery) vulnerabilities in a controlled environment using WebGoat and then learn how to mitigate them. CSRF vulnerabilities occur when malicious sites trick authenticated users into submitting unwanted actions to a web application.

Vulnerability: Basic GET CSRF

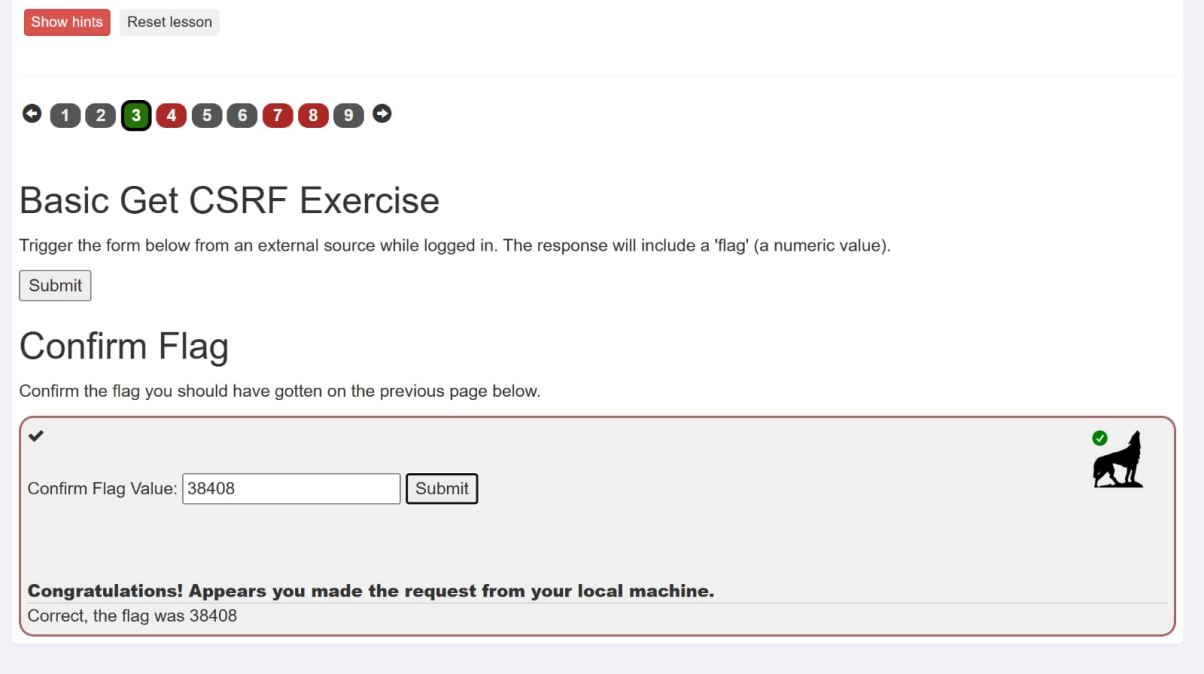
Description:

This task demonstrates how a GET request can be used to trigger state-changing operations on behalf of an authenticated user.

Steps:

- Identified the hidden form containing CSRF token set to false.
- Replicated the request from an external page using an HTML form.

Submitted the form to receive the flag



Mitigation:

- Use CSRF tokens.

- Avoid using GET requests for state-changing operations.

Vulnerability: Post a Review on Someone Else's Behalf

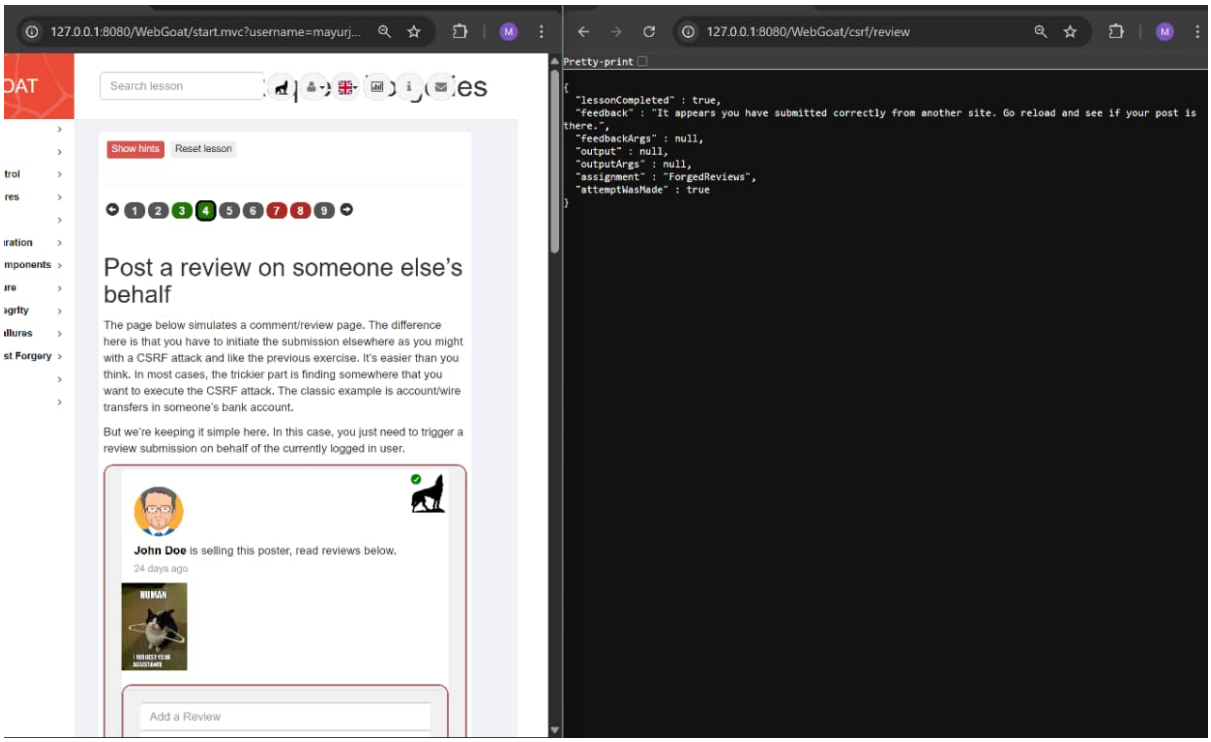
Description:

This task showed how CSRF can be exploited to post content as another user.

Steps:

- Constructed a POST request with pre-filled values.

Executed it while authenticated to simulate unauthorized posting



Mitigation:

Enforce CSRF tokens.

Verify the origin of requests with Referer or Origin headers.

Vulnerability: CSRF and Conteny-Type

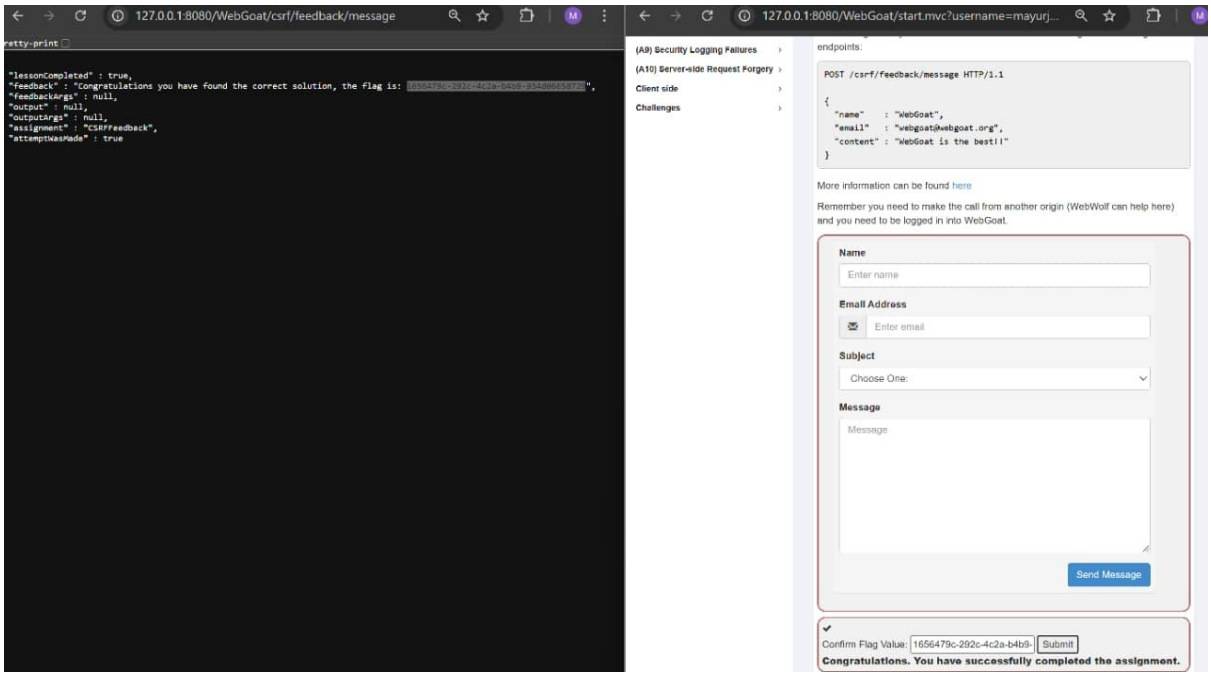
Description:

This task demonstrates how certain content types (like application/json) can be blocked from CSRF attacks.

Steps:

Attempted a CSRF attack using a content type the server didn't accept.

Observed the server's behavior and rejection.



Mitigation:

Accept only JSON requests.

Implement proper CSRF token validation

Vulnerability: Login CSRF Attack

Description:

This task highlights how a malicious actor could log a victim into an attacker-controlled account.

Steps:

Built a form that auto-submitted login credentials.

Demonstrated that the victim was logged into the attacker's account.

Login CSRF attack

In a login CSRF attack, the attacker forges a login request to an honest site using the attacker's username and password at that site. If the forgery succeeds, the honest server responds with a `Set-Cookie` header that instructs the browser to mutate its state by storing a session cookie, logging the user into the honest site as the attacker. This session cookie is used to bind subsequent requests to the user's session and hence to the attacker's authentication credentials. Login CSRF attacks can have serious consequences, for example see the picture below where an attacker created an account at google.com the victim visits the malicious website and the user is logged in as the attacker. The attacker could then later on gather information about the activities of the user.

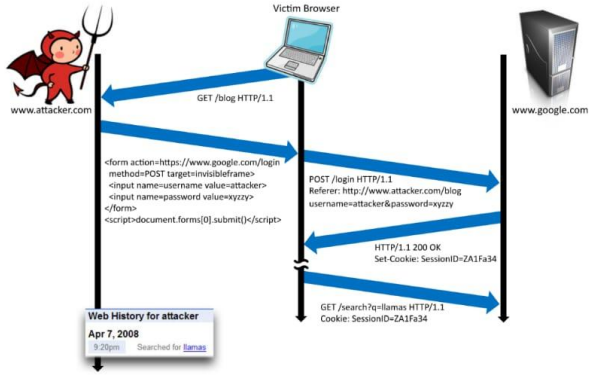


Figure: Login CSRF from Robust Defenses for Cross-Site Request Forgery

For more information read the following paper.

In this assignment try to see if WebGoat is also vulnerable for a login CSRF attack. Leave this tab open and in another tab create a user based on your own username prefixed with `csrf-`. So if your username is `tom` you must create a new user called `csrf-tom`.

Login as the new user. This is what an attacker would do using CSRF. Then click the button in the original tab. Because you are logged in as a different user, the attacker learns that you clicked the button.

✔

Press the button below when your are logged in as the other user

Solved!

Congratulations, now log out and login with your normal user account within WebGoat, remember the attacker knows you solved this assignment

Mitigation:

- Use SameSite cookies.
- Require re-authentication for sensitive actions.
- Implement CSRF tokens even on login endpoints.

OWASP ZAP Scan

Tool Used: OWASP ZAP (Zed Attack Proxy)

Purpose: Identify web application vulnerabilities, including CSRF, SQL Injection, missing headers, etc. **Scan Target:**

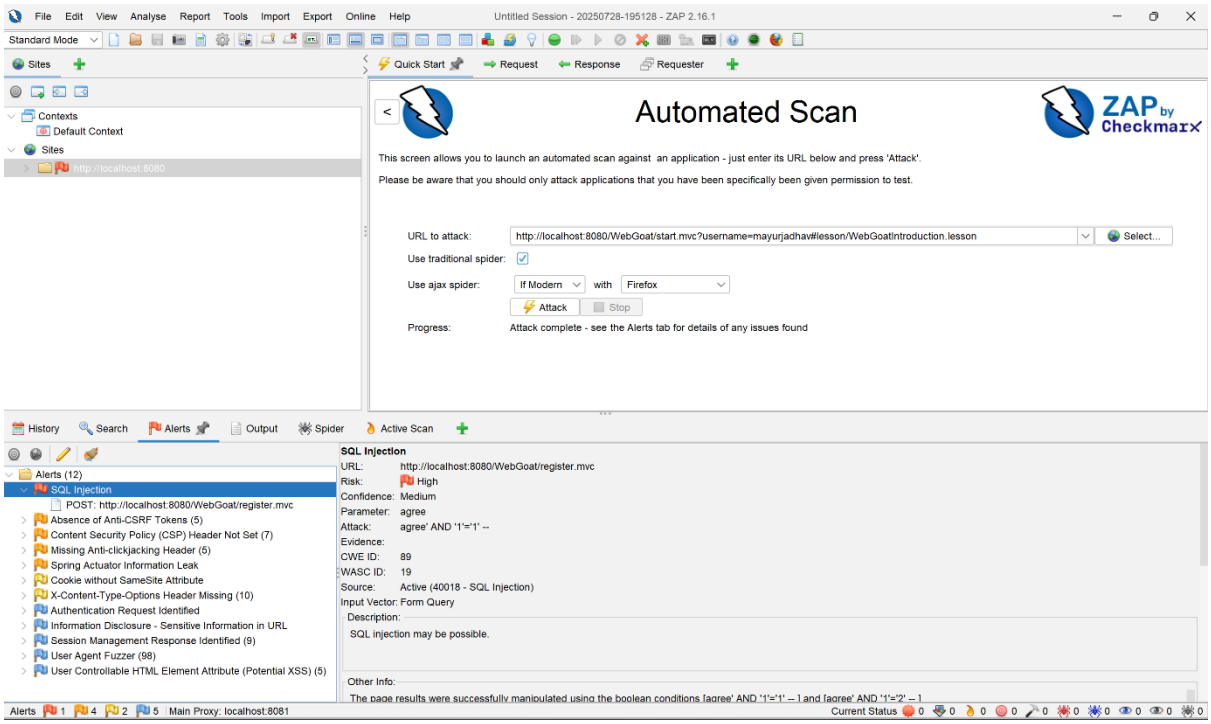
http://localhost:8080/WebGoat/start.mvc?username=mayurjadhav#lesson/WebGoatI ntroducti on.lesson

Notable Finding Related to CSRF:

Absence of Anti-CSRF Tokens: Detected in multiple requests (5 instances)

Risk: Medium

Description: Anti-CSRF tokens are not implemented in sensitive requests, making the app vulnerable to CSRF attacks.



Additional Findings :

SQL Injection (High Risk)

Missing CSP and Clickjacking protection headers

Cookie without SameSite attribute

CSRF Mitigation Recommendations Based on ZAP Scan

Implement CSRF tokens on all state-changing requests.

Add SameSite=Strict or Lax to session cookies.

Set X-Frame-Options: DENY or SAMEORIGIN to prevent clickjacking.

Include a Content-Security-Policy header.