SECURITY ASSESSMENT REPORT

Task 1 – Web Application Security Testing

Internship: Future Interns
Intern Name: *Mayur Sandipan Jadhav*
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:**

- OWASP ZAP (Scanning & passive analysis)
- Web Browser (manual payload testing_

3. Vulnerability Assessments

Vulnerability: Simple SQL Injection

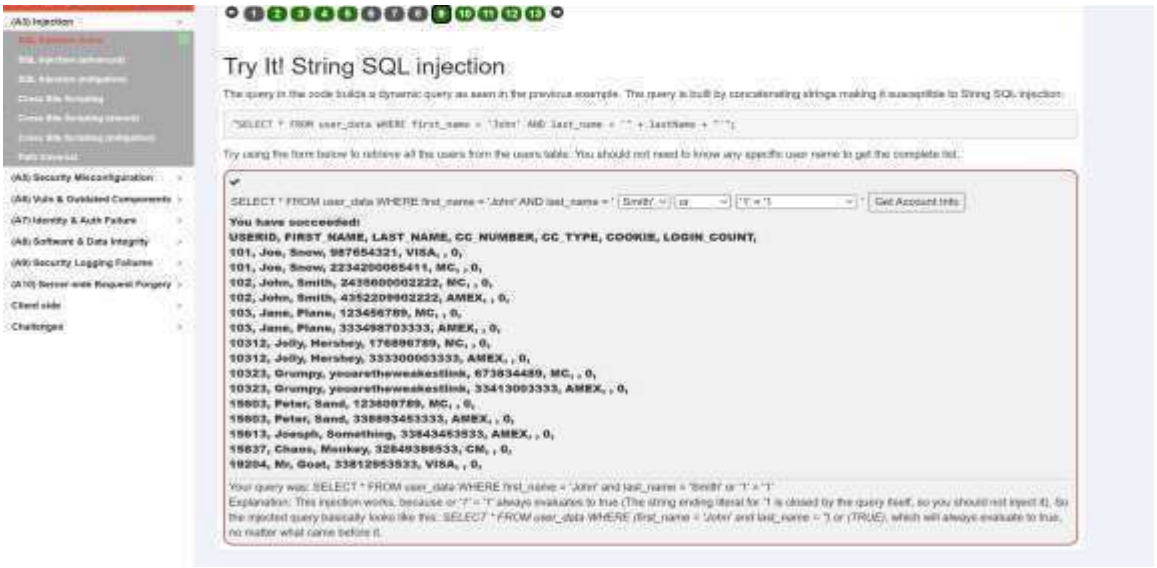
Description: Injected `' OR '1'='1` in a login field to bypass authentication.

How Discovered: Manual input in the login form.

Why It's Dangerous: Allows attackers to gain unauthorized access to accounts.

Mitigation: Use parameterized queries (prepared statements)

.Screenshot :



Vulnerability: Numeric SQL Injection

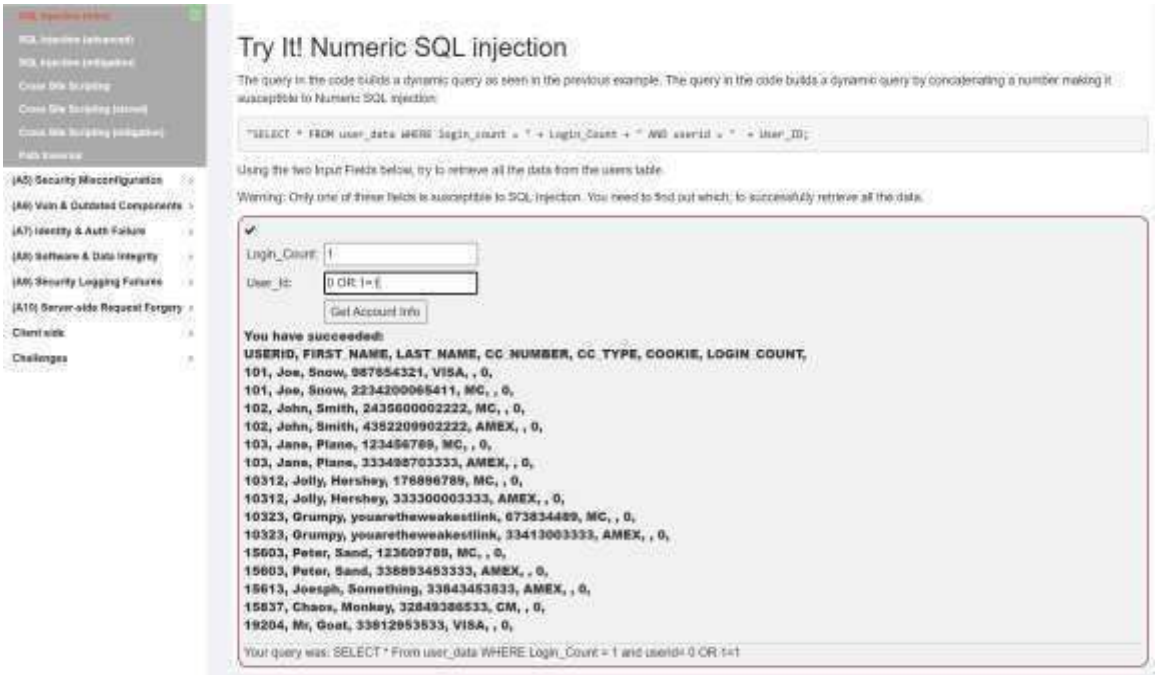
Description: Used `OR 1=1` in the numeric input field.

How Discovered: Entered numeric injection in the `User_ID` field.

Why It's Dangerous: Can allow attackers to retrieve or manipulate all data.

Mitigation: Type check input and use parameterized queries.

Screenshot:



Vulnerability: String SQL Injection with Comment

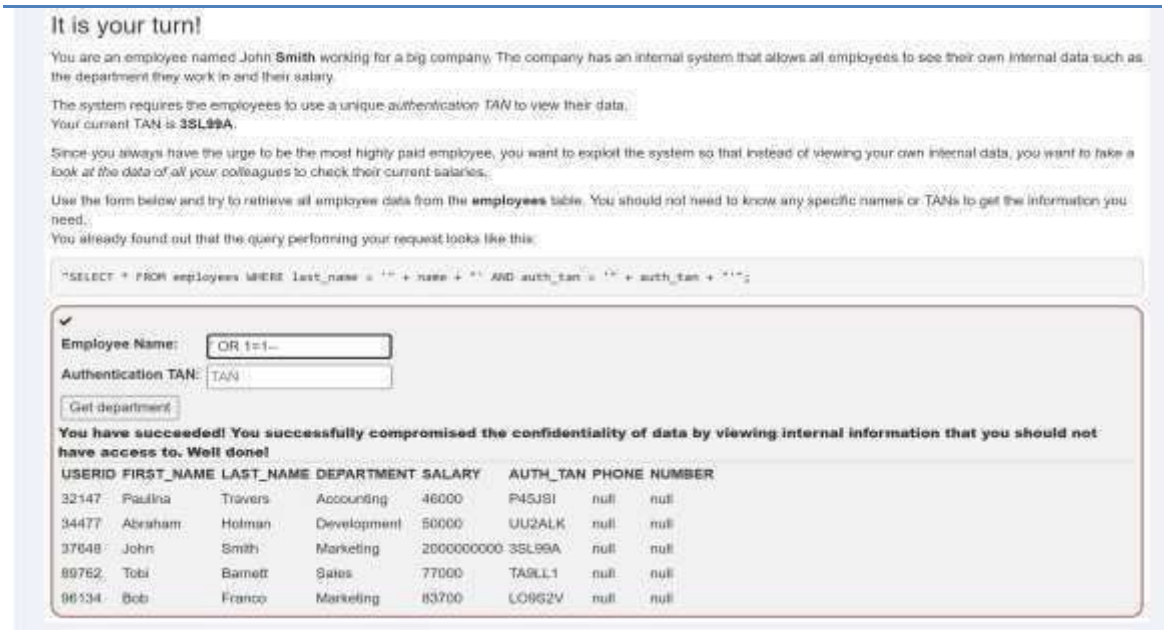
Description: Used `' OR '1'='1' --` to bypass login.

How Discovered: Input in username field with SQL comment to ignore rest of query.

Why It's Dangerous: Ignores password checks and leads to unauthorized access.

Mitigation: Escape input, use ORM, and validate data.

Screenshot:



Vulnerability: Compromising Integrity via Query Chaining

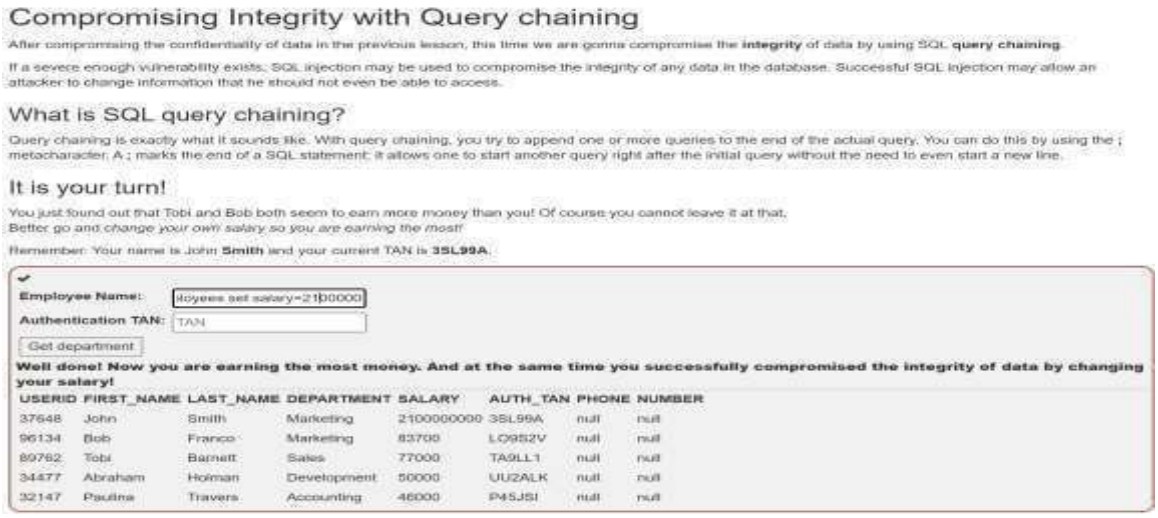
Description: Injected additional queries using `;` to change salary of a user.

How Discovered: Used chaining in name input (e.g., `Smith'; UPDATE salaries SET amount = 99999 WHERE user = 'Smith`).

Why It's Dangerous: Allows changing critical information like salaries.

Mitigation: Disable multi-query execution; validate input.

Screenshot:



Vulnerability: Compromising Availability (DCL Injection)

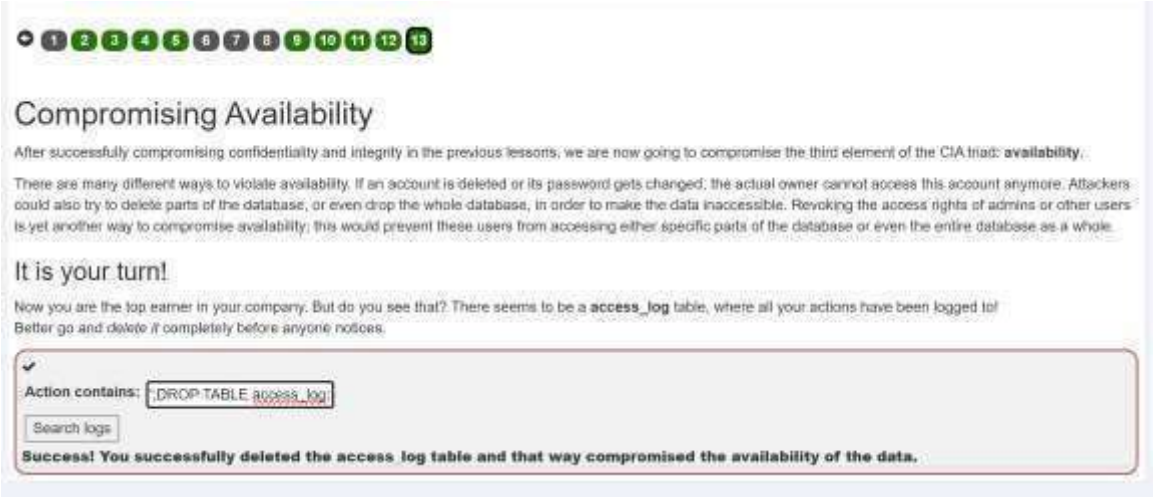
Description: Dropped a database table (`DROP TABLE access_log`).

How Discovered: Used SQL injection to run `DROP TABLE` command.

Why It's Dangerous: Destroys data and affects application functionality.

Mitigation: Restrict DDL/DCL privileges; use DB accounts with least privilege.

Screenshot:



Mitigation Summary :

- Use prepared statements and parameterized queries.

- Implement input validation and whitelisting.

- Employ least privilege principle in database roles.

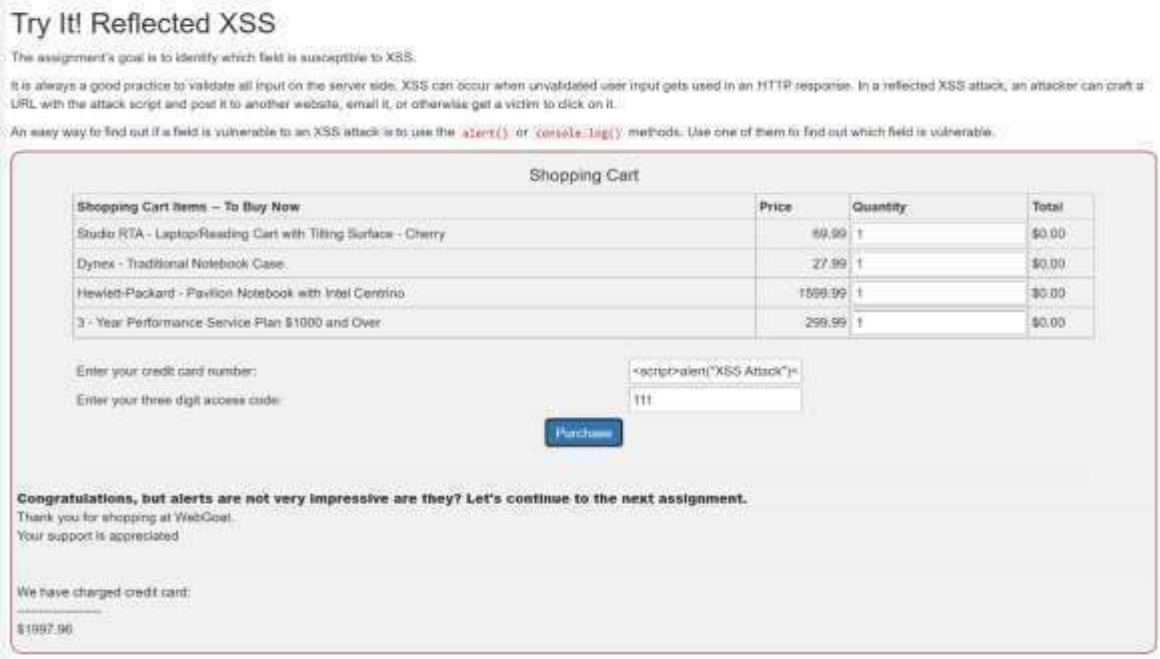
- Use Web Application Firewalls (WAFs) and secure coding practices.

Cross Site Scripting (XSS)

Vulnerability: Reflected XSS

Description: Injected <script>alert('XSS')</script> in a URL/query parameter which was immediately reflected on the page.

- **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.
- **Screenshot:**

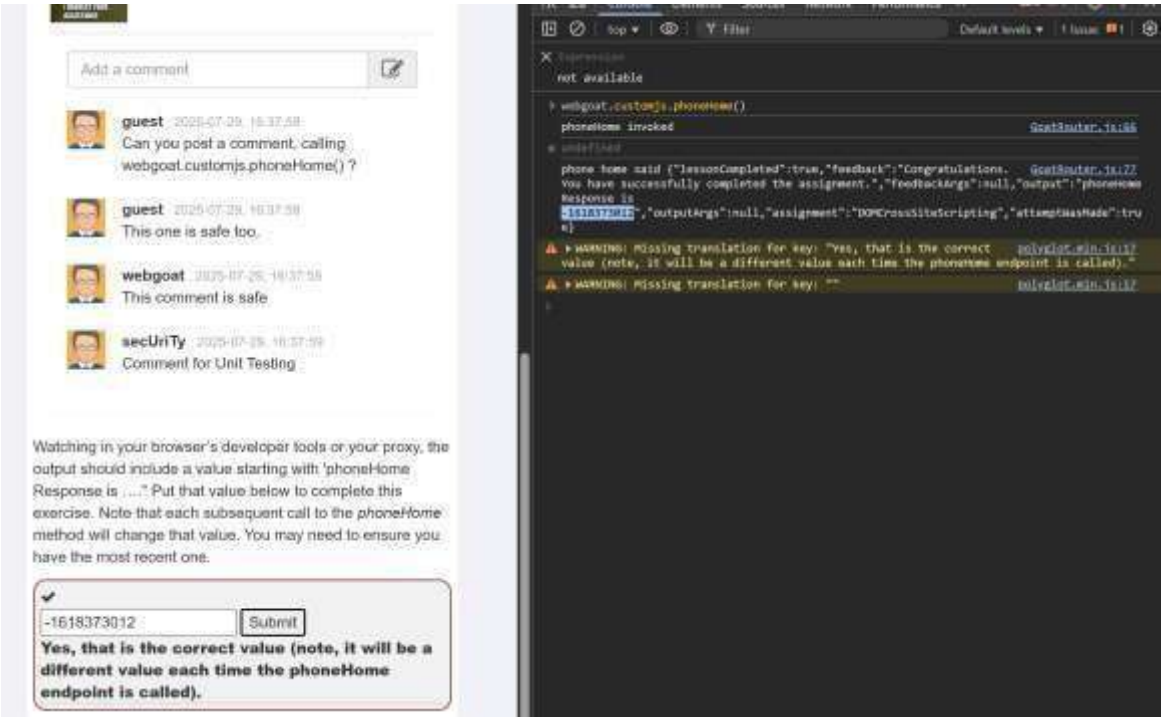


Vulnerability: Stored XSS

Description: Injected malicious JavaScript into the console of field that persisted in the application.

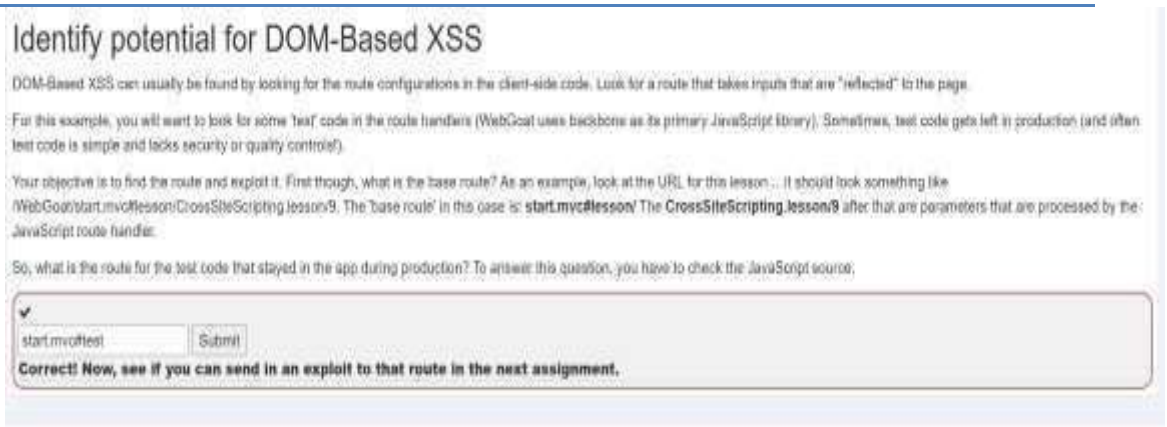
- **How Discovered:** Input was stored and later executed when viewing the message or comment.

- **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).
- **Screenshot:**



Vulnerability: DOM-Based XSS

- **Description:** Injected script into the fragment identifier (#) of the URL, like start.mvc#test/<script>alert('DOM')</script>.
- **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.
- **Screenshot:**



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)

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.

Screenshot: CODE:

```
<form accept-charset="UNKNOWN" id="basic-csrf-get" method="POST" name="form1" target="_blank" successcallback=""
action="http://127.0.0.1:8080/WebGoat/csrf/basic-get-flag">
        <input name="csrf" type="hidden" value="false">
        <input type="submit" name="submit" fdprocessedid="517je">
</form>
```

RESULT:

Mitigation:

- Use CSRF tokens.
- Avoid using GET requests for state-changing operations.

Vulnerability: Post a Review on Someone Else's Behalf

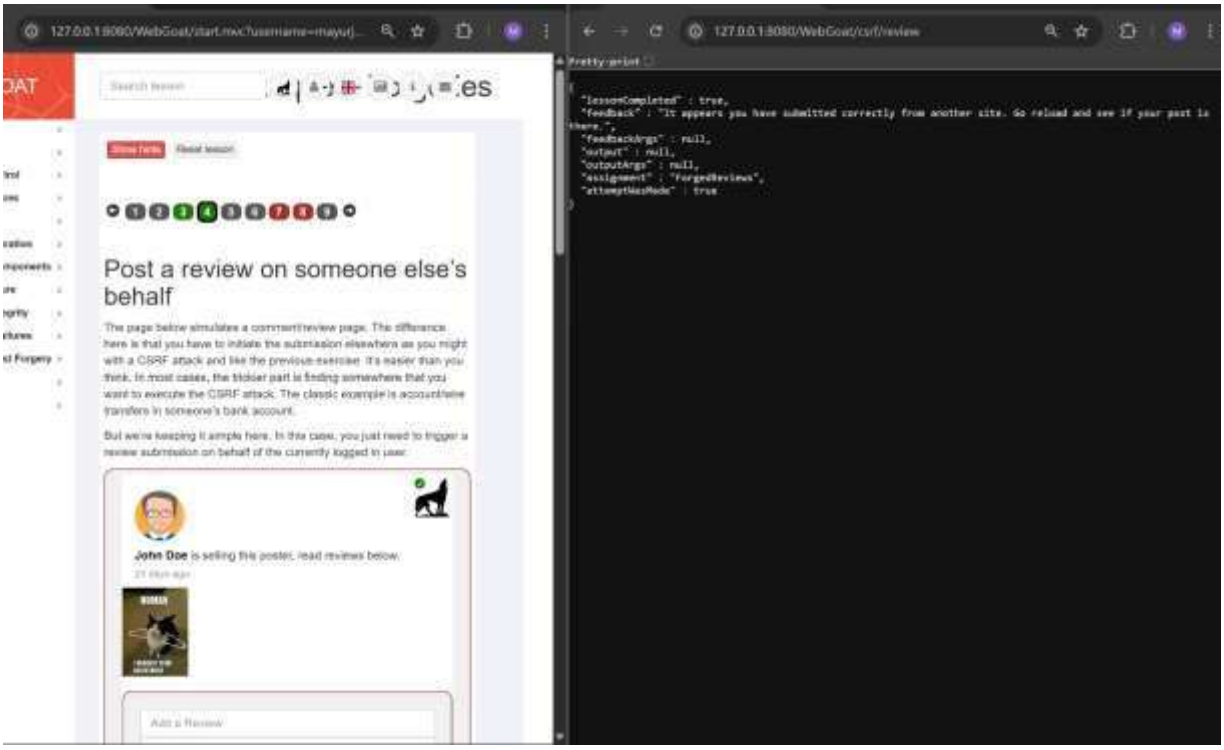
Description:
This task showed how CSRF can be exploited to post content as another user.

Steps:

- Constructed a POST request with pre-filled values.
- Executed it while authenticated to simulate unauthorized posting. **Screenshot: CODE:**

```
<form class="attack-form" accept-charset="UNKNOWN" id="csrf-review" method="POST"
name="review-form" successcallback=""
action="http://127.0.0.1:8080/WebGoat/csrf/review">
<input class="form-control" id="reviewText" name="reviewText" placeholder="Add a
Review" type="text" fdprocessedid="8f7z2n">
<input class="form-control" id="reviewStars" name="stars" type="text"
fdprocessedid="vr9rn">
<input type="hidden" name="validateReq" value="2aa14227b9a13d0bede0388a7fba9aa9">
<input type="submit" name="submit" value="Submit review" fdprocessedid="hlaix">
</form>
```

RESULT:



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.

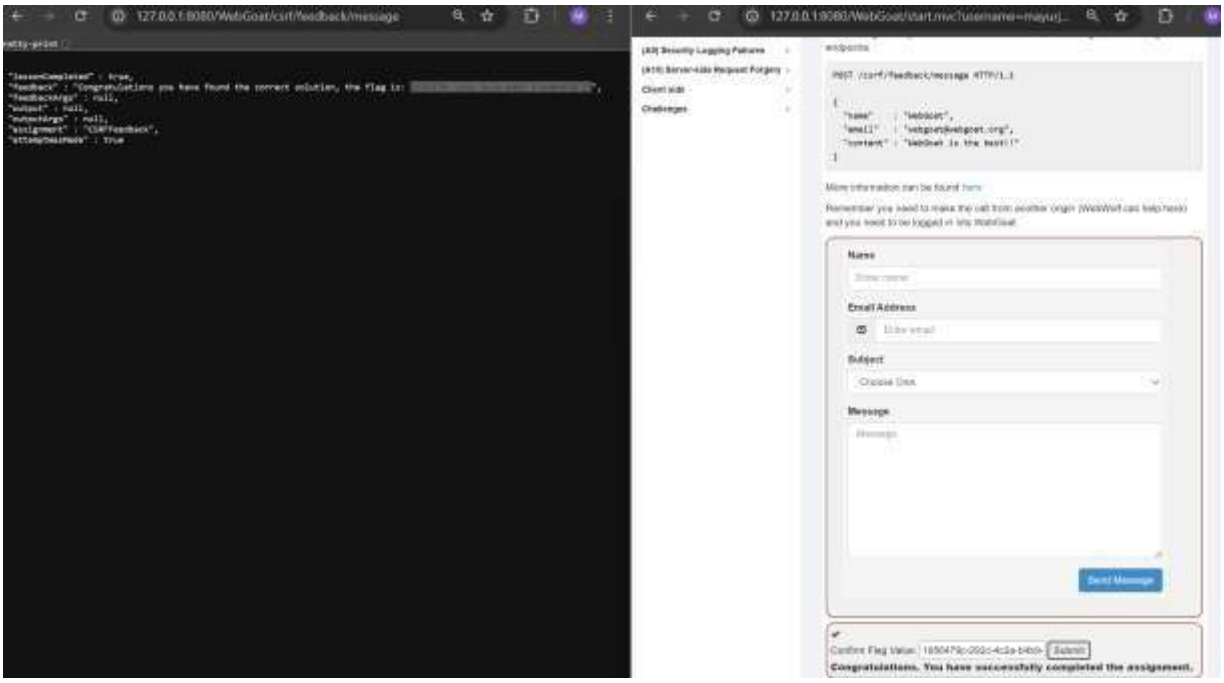
Steps:

- Attempted a CSRF attack using a content type the server didn't accept.
- Observed the server's behavior and rejection.

Screenshot:
CODE:

```
<html>
<title>Json CSRF POC</title>
<center>
<h1>JSON CSRF POC</h1>
<form action=http://127.0.0.1:8080/WebGoat/csrf/feedback/message method=post enctype="text/plain">
<input name='{"name":"WebGoat","email":"Webgoat@webgoat.org","content":"WebGoat is the best!!","ignore_me":"' value='test"}'type='hidden'>
<input type=submit value="Submit">
</form>
</center>
</html>
```

RESULT:



Mitigation:

- Accept only JSON requests.
- Implement proper CSRF token validation.

Vulnerability: Login CSRF Attack

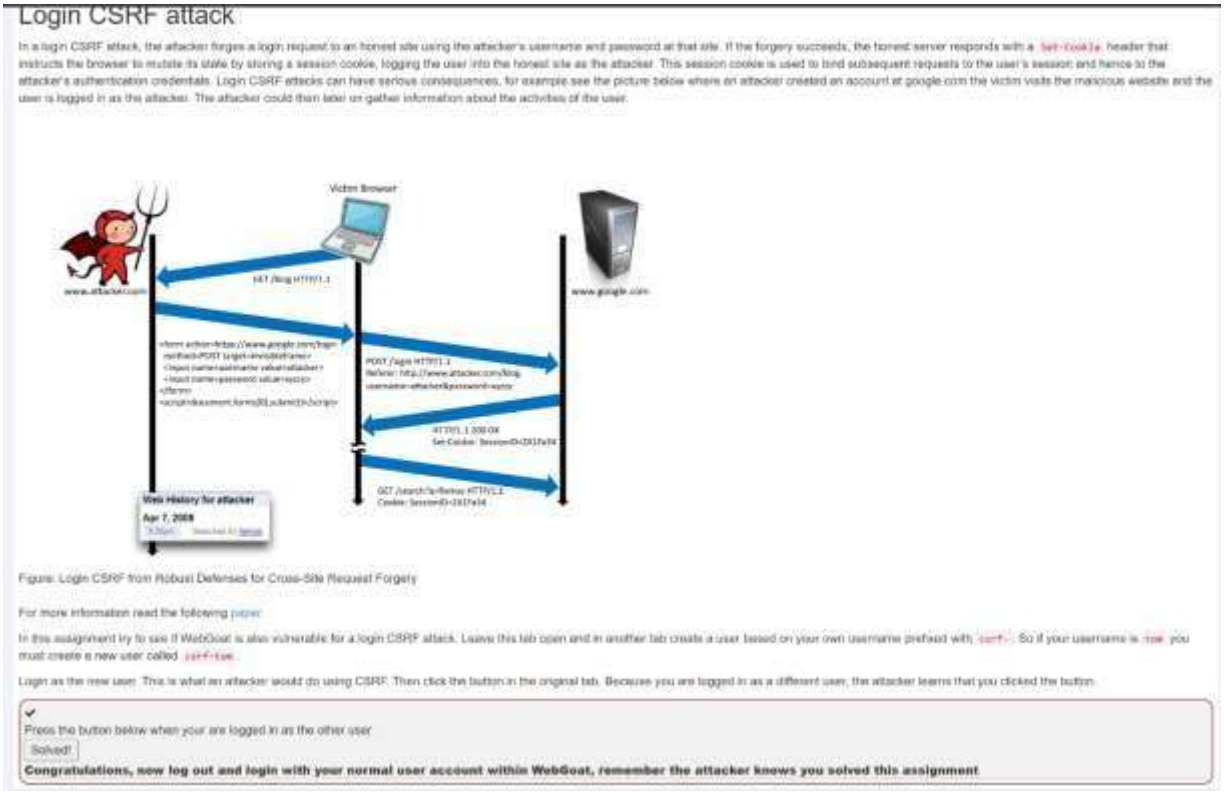
Description:
This task highlights how a malicious actor could log a victim into an attacker-controlled account.
Steps:

- Built a form that auto-submitted login credentials.
- Demonstrated that the victim was logged into the attacker's account.

Screenshot: CODE:

```
<form action=="http://127.0.0.1:8080/WebGoat/login" method="POST" style="width :300px;">
<input type="hidden" name ="username" value="csrf-mayurjadav">
<input type="hidden" name ="password" value="webgoat">
<button type="submit">Sign in</button>
</form>
<script>document.login.submit()</script>
```

RESULT*:*



Mitigation:

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

Mitigation Summary :

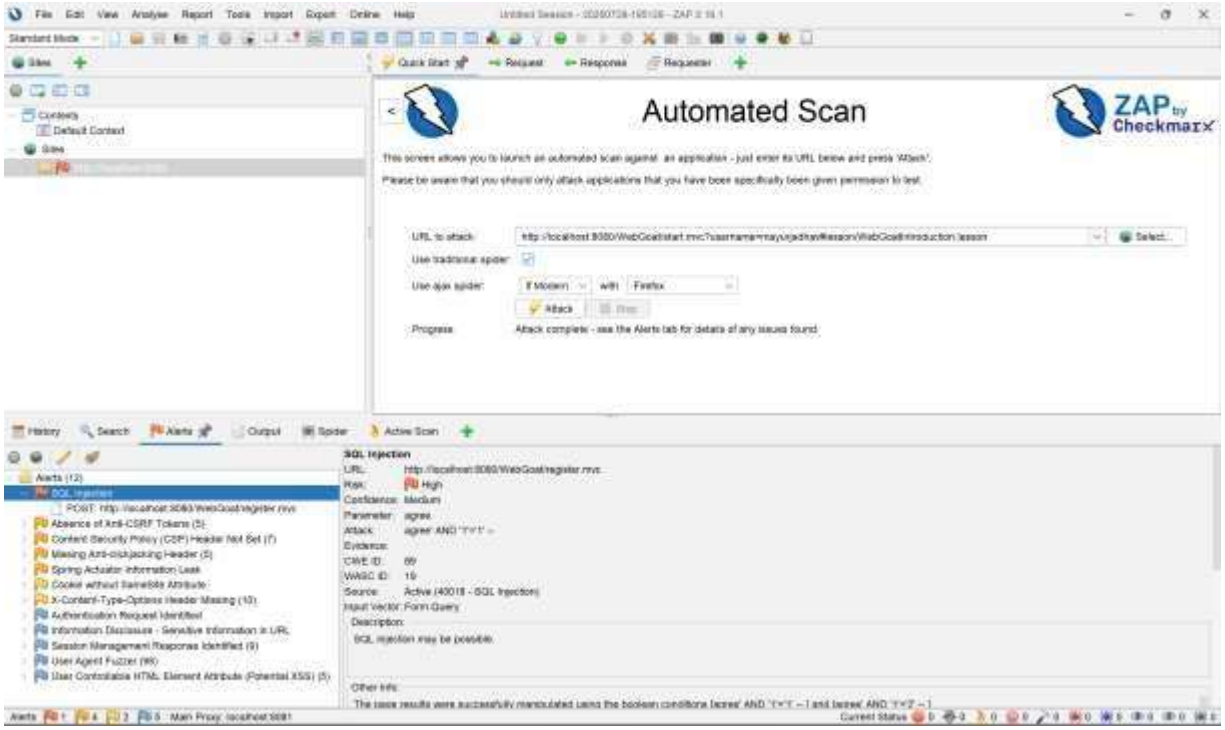
- CSRF is a dangerous vulnerability often overlooked due to its simplicity.
- Mitigation requires server-side enforcement like CSRF tokens and secure cookie handling.
- Modern browsers provide mechanisms like SameSite cookie attributes that help prevent CSRF.

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/WebGoatIntroduction.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.

Screenshot:



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.

---

5. Conclusion

During this security assessment, three critical vulnerabilities were successfully identified in a deliberately vulnerable test application. Attacks were tested manually and through OWASP ZAP. SQL Injection, XSS, and CSRF were exploited to understand real-world consequences and learn defense strategies. By applying OWASP principles and mitigation techniques, modern web applications can be safeguarded from these attacks.

---

6. References

- OWASP Top 10 Security Risks (owasp.org)
- WebGoat Documentation
- OWASP ZAP Tool Documentation