

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
 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
-

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:

SQL Injection (intro)	
SQL Injection (advanced)	
SQL Injection (mitigation)	
Cross Site Scripting	
Cross Site Scripting (stored)	
Cross Site Scripting (mitigation)	
Path traversal	
(A5) Security Misconfiguration	>
(A6) Vuln & Outdated Components	>
(A7) Identity & Auth Failure	>
(A8) Software & Data Integrity	>
(A9) Security Logging Failures	>
(A10) Server-side Request Forgery	>
Client side	>
Challenges	>

Try It! String SQL injection

The query in the code builds a dynamic query as seen in the previous example. The query is built by concatenating strings making it susceptible to String SQL injection:

```
"SELECT * FROM user_data WHERE first_name = 'John' AND last_name = '' + lastName + '";
```

Try using the form below to retrieve all the users from the users table. You should not need to know any specific user name to get the complete list.



SELECT * FROM user_data WHERE first_name = 'John' AND last_name = 'Smith' 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, 333498703333, 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, Joesph, Something, 33843453533, AMEX, , 0,
15837, Chaos, Monkey, 32849386533, CM, , 0,
19204, Mr, Goat, 33812953533, VISA, , 0,
```

Your query was: SELECT * FROM user_data WHERE first_name = 'John' and last_name = 'Smith' or '1' = '1'

Explanation: This injection works, because or '1' = '1' always evaluates to true (The string ending literal for '1' is closed by the query itself, so you should not inject it). So the injected query basically looks like this: SELECT * FROM user_data WHERE first_name = 'John' and last_name = '' or TRUE, which will always evaluate to true, no matter what came before it.

2) Vulnerability: Numeric SQL Injection:

SQL Injection (intro)	
SQL Injection (advanced)	
SQL Injection (mitigation)	
Cross Site Scripting	
Cross Site Scripting (stored)	
Cross Site Scripting (mitigation)	
Path traversal	
(A5) Security Misconfiguration	>
(A6) Vuln & Outdated Components	>
(A7) Identity & Auth Failure	>
(A8) Software & Data Integrity	>
(A9) Security Logging Failures	>
(A10) Server-side Request Forgery	>
Client side	>
Challenges	>

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: 0

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, 333498703333, 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,
```

3) Vulnerability: String SQL Injection with Comment:

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: ' OR '1'='1

Authentication TAN: TAN

Get department

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
32147	Paulina	Travers	Accounting	46000	P45JSI	null
34477	Abraham	Holman	Development	50000	UU2ALK	null
37648	John	Smith	Marketing	64350	3SL99A	null
89762	Tobi	Barnett	Sales	77000	TA9LL1	null
96134	Bob	Franco	Marketing	83700	LO9S2V	null

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:

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 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.

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 Tilting Surface - Cherry	69.99	1 <input type="text"/>	\$0.00
Dynex - Traditional Notebook Case	27.99	1 <input type="text"/>	\$0.00
Hewlett-Packard - Pavilion Notebook with Intel Centrino	1599.99	1 <input type="text"/>	\$0.00
3 - Year Performance Service Plan \$1000 and Over	299.99	1 <input type="text"/>	\$0.00

Enter your credit card number:

4128 3214 0002 1999

Enter your three digit access code:

111

Purchase

Try again. We do want to see a specific JavaScript mentioned in the goal of the assignment (in case you are trying to do something fancier).

Thank you for shopping at WebGoad.

Your support is appreciated

We have charged credit card-4128 3214 0002 1999

\$1997.96

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).

The screenshot shows a web application interface with a list of comments and a form to submit a new comment. The comments are:

- Guest** 2025-08-22, 11:46:04: You can post a comment, calling `webgoat.customjs.phoneHome()` ?
- guest** 2025-08-22, 11:46:04: This one is safe too.
- webgoat** 2025-08-22, 11:46:04: This comment is safe
- seCurity** 2025-08-22, 11:46:04: Comment for Unit Testing

Below the comments is a form with a text input field and a "Submit" button. Below the form, a message states: "Watching in your browser's developer tools or your proxy, the output should include a value starting with 'phoneHome Response is ...'. Put that value below to complete this exercise. Note that each subsequent call to the `phoneHome` method will change that value. You may need to ensure you have the most recent one."

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.

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/lesson/9. The 'base route' in this case is: **start.mvc/lesson/** The **CrossSiteScripting.lesson/9** 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.

Try It! DOM-Based XSS

Some attacks are "blind." Fortunately, you have the server running here, so you can tell if you are successful. Use the route you just found and see if you can use it to reflect a parameter from the route without encoding to execute an internal function in WebGoat. The function you want to execute is:

webgoat.customjs.phoneHome()

Sure, you could use console/debug to trigger it, but you need to trigger it via a URL in a new tab.

Once you trigger it, a subsequent response will come to your browser's console with a random number. Put that random number below.

☒

Correct!

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

[Show hints](#) [Reset lesson](#)

➡ 1 2 3 4 5 6 7 8 9 ⬅

Basic Get CSRF Exercise

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

Submit


Confirm Flag

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

✓

Confirm Flag Value:

Submit

✔ 

Congratulations! Appears you made the request from your local machine.
Correct, the flag was 38408

Mitigation:

- Use CSRF tokens.

- Avoid using GET requests for state-changing operations.

Vulnerability: Post a Review on Someone Else's Behalf

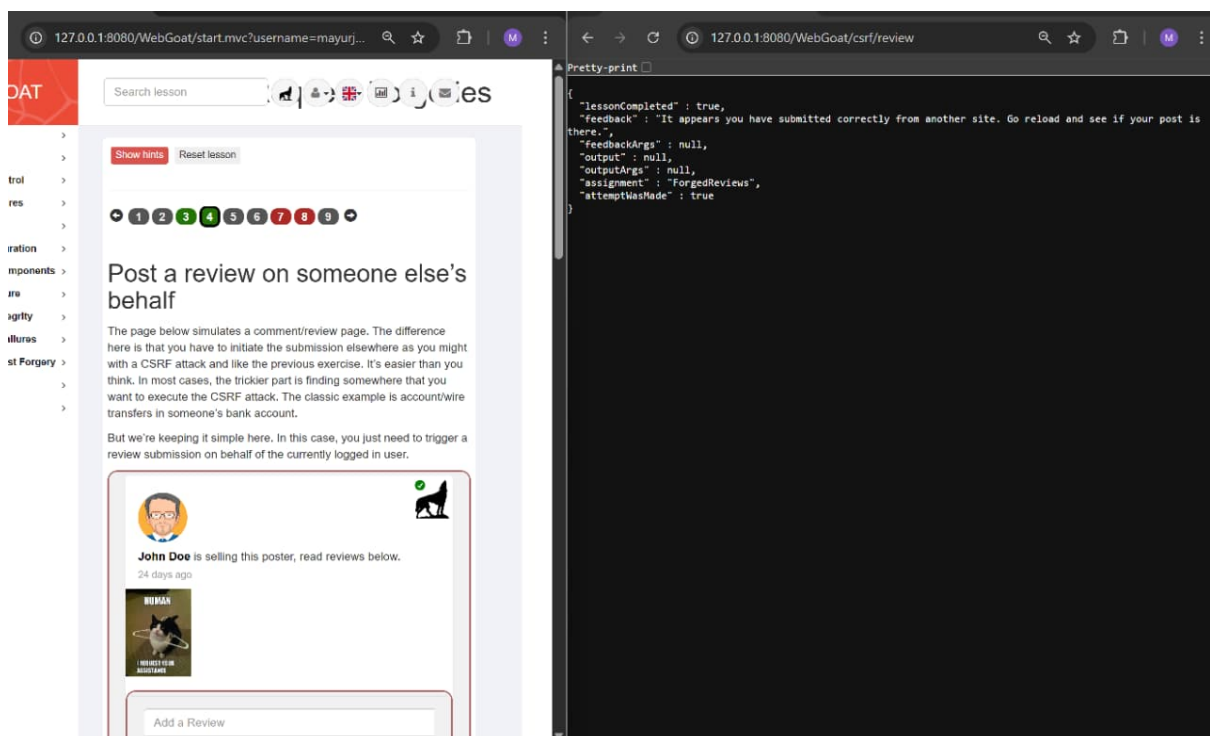
Description:

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 Content-Type

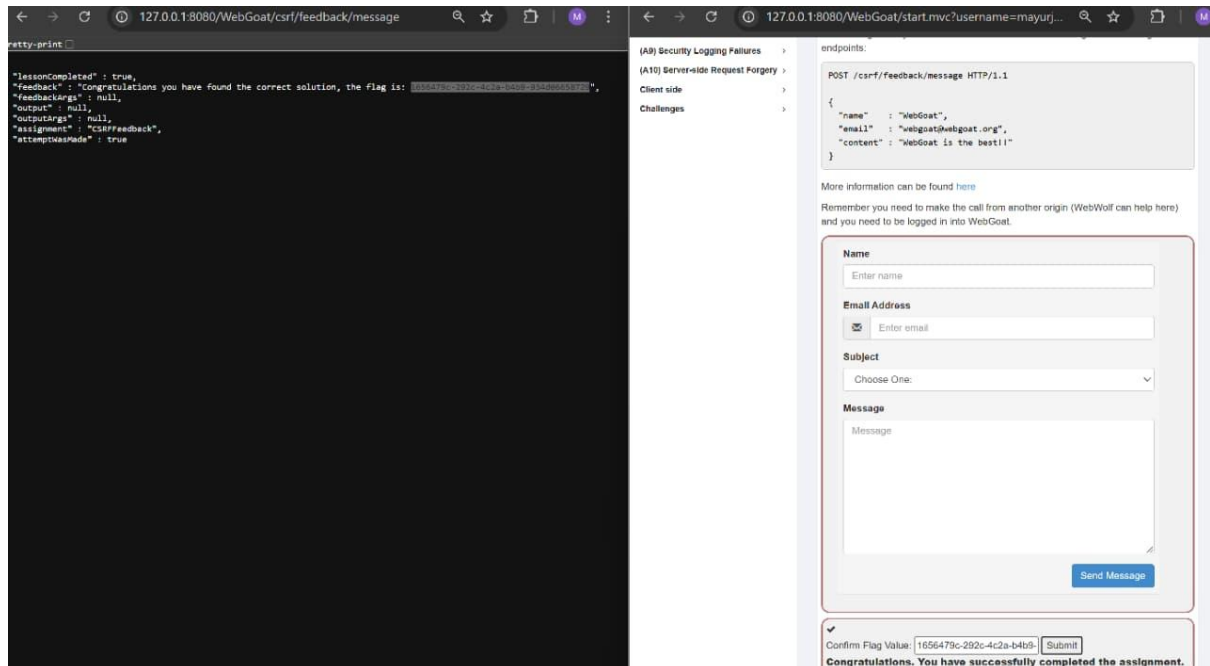
Description:

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.



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.

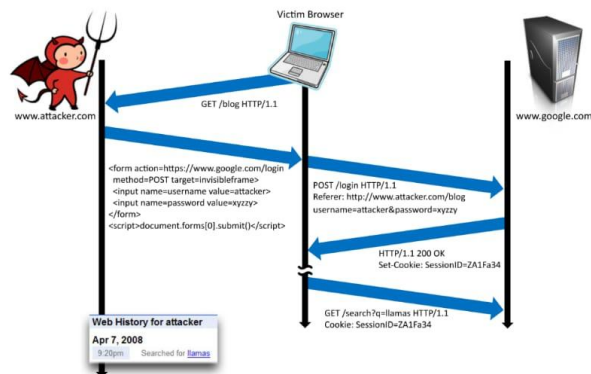


Figure: Login CSRF from Robust Defenses for Cross-Site Request Forgery

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 you are logged in as the other user

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:**

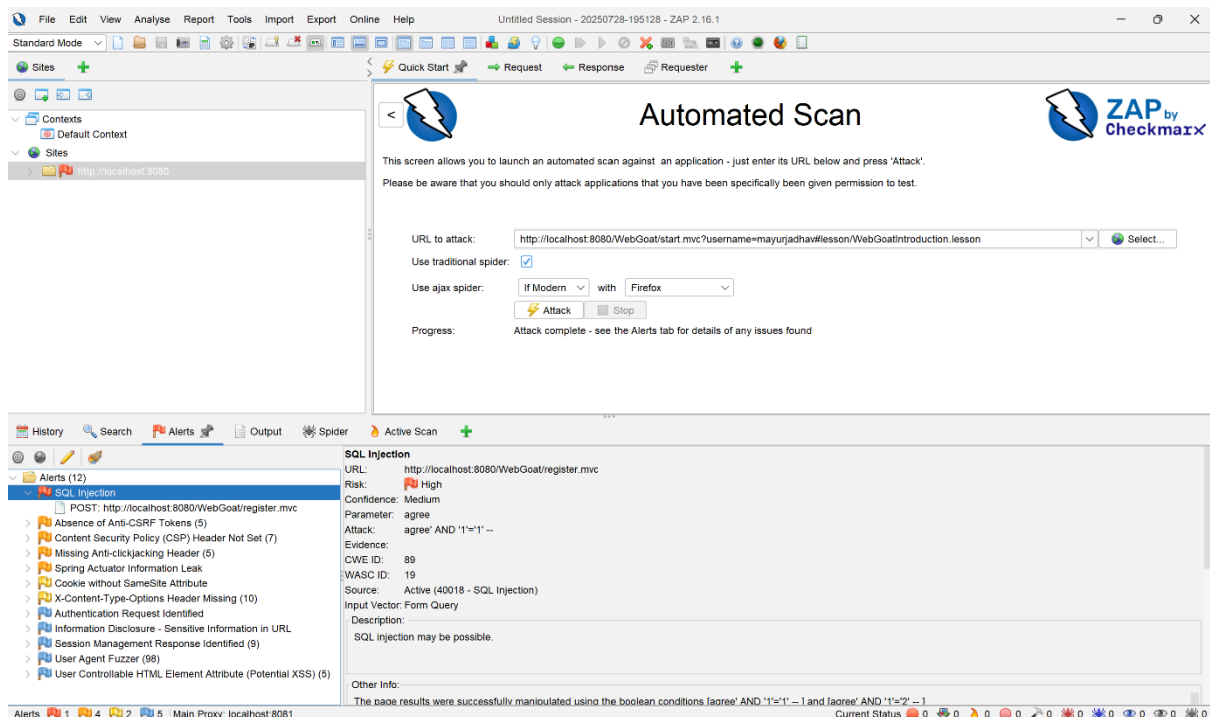
http://localhost:8080/WebGoat/start.mvc?username=mayurjadhav#lesson/WebGoatIntroduction.lesson

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.



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.