Security Assessment Report

Manual Pentest

Demo Webapp

http://testphp.vulnweb.com

Report dated : April 3, 2025

TABLE OF CONTENT

Executive Summary

A security assessment was conducted on **[testphp.vulnweb.com]** from **[31-03-2025] to [03-04-2025]** to evaluate the application's security posture. The assessment was performed using **manual penetration testing techniques** to identify vulnerabilities and assess potential security risks.

This engagement aimed to:

- Identify **security loopholes** and **business logic errors** that could pose threats to the system, infrastructure, or data.
- Evaluate the **effectiveness of existing security controls** in mitigating risks.
- Recommend **technical security best practices** to enhance the application's security posture.
- Assess the **impact of discovered vulnerabilities**, including **data exposure, financial risks, and reputational damage** if exploited by malicious actors.
- Provide **clear, actionable recommendations** for mitigating identified vulnerabilities.

Scope of Assessment

Assessment Type	Assessment Name	Assessment date	Scope	Tools
Web app	Demo webapp	31-03-25 to 3-3-25	http://testphp.vulnweb.com	1]Burpsuite 2]Kali Linux

RESOLUTION STATICS

SEVERITY	CRITICAL	HIGH	MEDIUM	LOW
Sql injection Boolean based	✓			
Sql injection Error based	✓			
Cross-site Scripting	✓			
Version Disclosure			✓	
Broken Authentication		✓		
CSRF			✓	

Details of Vulnerability found:

1]SQL Injection:Boolean Based (' OR '1'='1' --)

Description:

SQL Injection (SQLi) is a critical web application vulnerability that occurs when an application fails to properly sanitize user input before including it in SQL queries. **Boolean-Based SQL Injection** occurs when an attacker manipulates SQL queries using conditional logic (e.g., `OR '1'='1'`) to extract sensitive information or bypass authentication.

On **testphp.vulnweb.com**, it was observed that the application is vulnerable to **Boolean-Based SQL Injection** in the **search functionality**, allowing attackers to manipulate database queries by injecting Boolean conditions.

Impact:

- **Data Exposure**: Attackers can retrieve sensitive information from the database, such as usernames, passwords, and financial data.
- **Authentication Bypass**: Malicious users can gain unauthorized access to accounts by modifying SQL queries.
- **Data Manipulation**: Attackers may alter, delete, or insert records, leading to data integrity issues.
- **Application Compromise**: A successful SQL injection attack can lead to full database compromise, allowing further exploitation like privilege escalation.

Steps to Reproduce (Proof of Concept - PoC):

1. Identify the vulnerable endpoint:

- Navigate to the **login page** of `testphp.vulnweb.com`.
- Enter a **normal username and password query** like:
- `Enter: test`
- Observe the results.

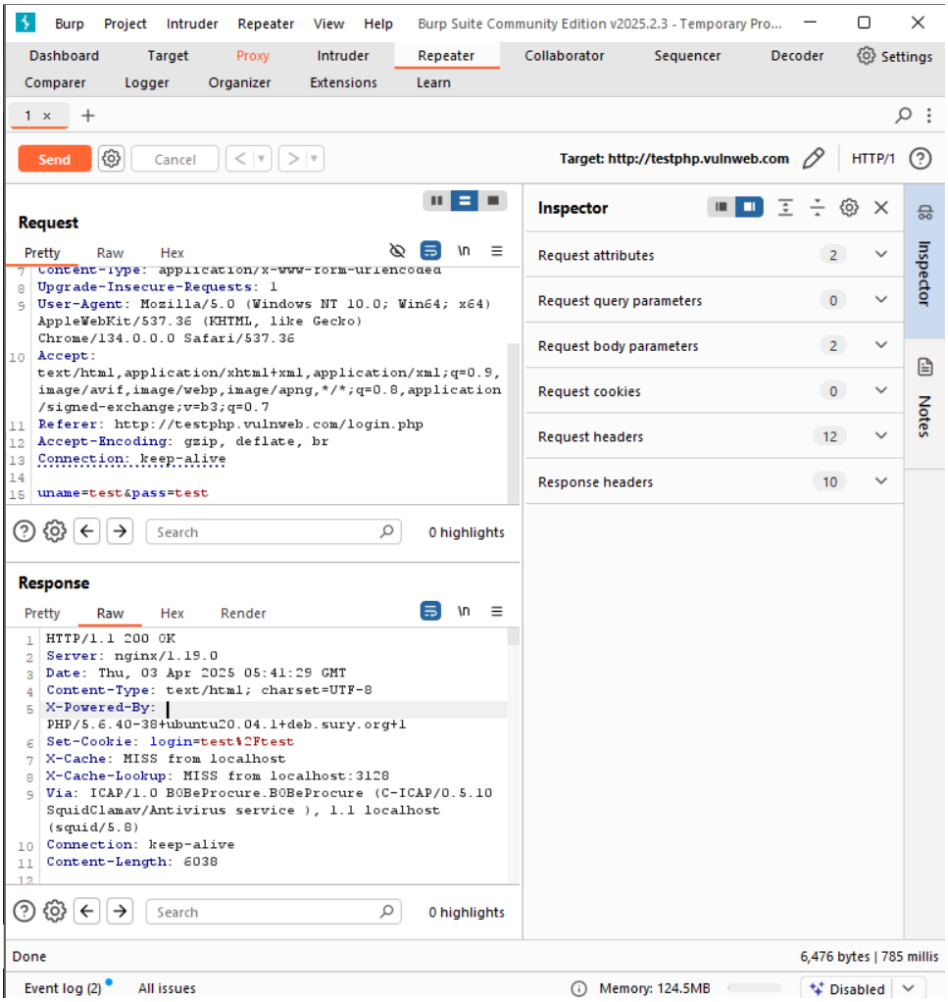
2. Inject a Boolean-Based SQLi payload:

- In the username bar, enter the following payload:
- `' OR '1'='1' --`
- If the application is vulnerable, it will **return all records** from the database instead of filtering results.

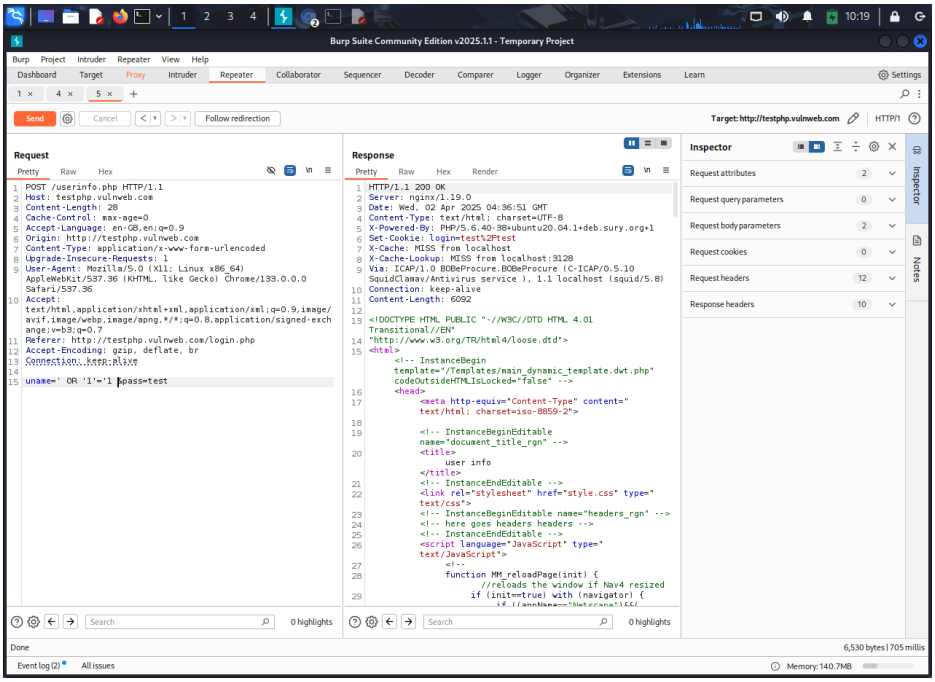
3. Validate the vulnerability:

- Try entering:
- `' OR '1'='2' --`
- If this **removes** results while the previous payload **returned all results**, it confirms a Boolean-Based SQL Injection vulnerability.

1] Inserting real data



2]Inserting Boolean based payload



Suggested Fix (Mitigation):

To mitigate SQL Injection vulnerabilities, implement the following security best practices:

Use Prepared Statements (Parameterized Queries):

- Instead of directly concatenating user input in SQL queries, use parameterized queries to prevent injection.

Input Validation & Sanitization:

- Validate user input using **allowlists** and reject suspicious characters like `'`, `"`, `--`, `;`, and `#`.
- Enforce proper **data types** (e.g., ensure ID parameters are integers).

Use Web Application Firewall (WAF):

- Deploy a **WAF** like ModSecurity to detect and block SQL injection attempts.

Implement Least Privilege Principle:

- Ensure the database user account has **minimal privileges** to reduce potential damage.
- Use **read-only** database access where possible.

Enable Logging & Monitoring:

- Log SQL errors and user activities to detect suspicious behavior early.
- Implement **Intrusion Detection Systems (IDS)** to identify SQL injection attempts.

Severity: High

Payload: ' OR '1'='1' --

2. SQL Injection – Error-Based (ORDER BY 1 --)

Description:

Error-Based SQL Injection is a technique where an attacker manipulates an SQL query to force the database to generate an error message. These error messages often reveal crucial details about the database structure, such as table names, column names, and database versions.

On **testphp.vulnweb.com**, it was found that injecting ORDER BY clauses with incorrect column numbers can generate database errors, exposing the internal database structure.

Impact:

- **Information Disclosure**: Attackers can extract database details like table names, column names, and DBMS version.
- **Further Exploitation**: Once database structure is known, attackers can perform more sophisticated SQL injection attacks.
- **Database Compromise**: Combined with other SQLi techniques, it can lead to full database control.

Steps to Reproduce (Proof of Concept - PoC):

1. Identify the vulnerable endpoint:

- Navigate to the **categories** of `testphp.vulnweb.com`.
- Click on categories 1
- Observe the results.

2. Inject an `ORDER BY` payload:

- Enter the following payload in the search field:

  ```
  test' ORDER BY 1 --
  ```

- If the query executes successfully, try incrementing the column index:

  ```
  test' ORDER BY 2 --
  ```

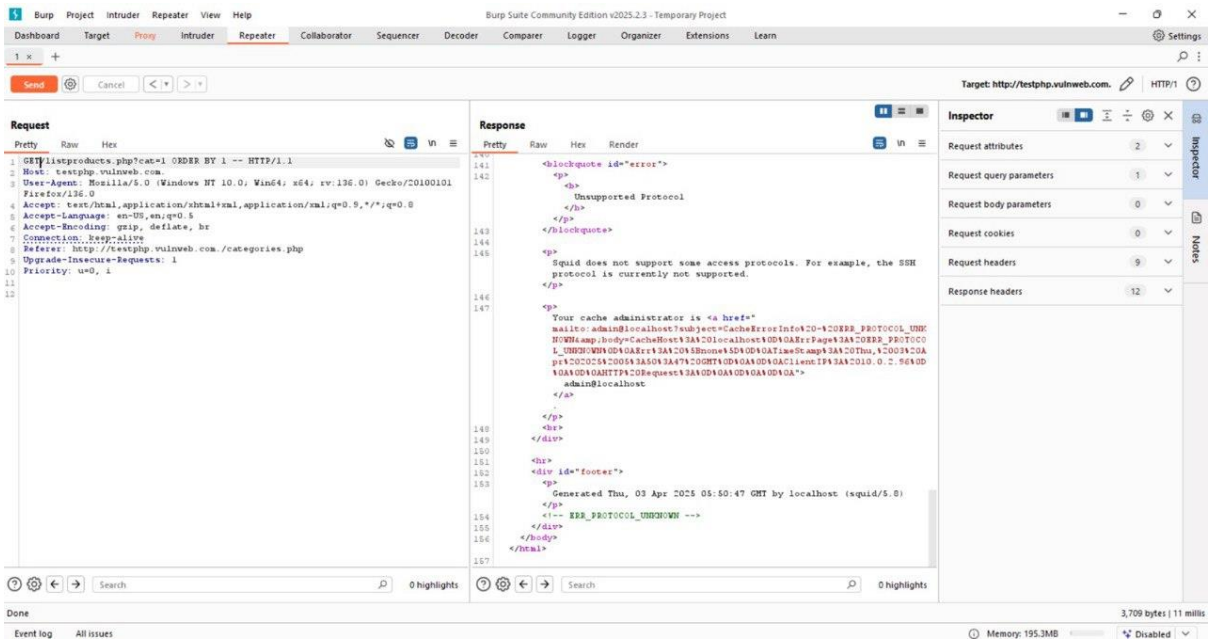
  ```
  test' ORDER BY 3 --
  ```

- Continue increasing the column number (`ORDER BY 4 --`, `ORDER BY 5 --`, etc.) until an **error message appears**.

3. Identify the number of columns:

- If an error occurs at `ORDER BY 6 --`, it means the table has **5 columns** (since column 6 does not exist).

1] After inserting payload



Suggested Fix (Mitigation):

To mitigate SQL Injection vulnerabilities, implement the following security best practices:

Use Prepared Statements (Parameterized Queries):

- Instead of directly concatenating user input in SQL queries, use parameterized queries to prevent injection.

Input Validation & Sanitization:

- Validate user input using **allowlists** and reject suspicious characters like `'`, `"`, `--`, `;`, and `#`.
- Enforce proper **data types** (e.g., ensure ID parameters are integers).

Use Web Application Firewall (WAF):

- Deploy a **WAF** like ModSecurity to detect and block SQL injection attempts.

Implement Least Privilege Principle:

- Ensure the database user account has **minimal privileges** to reduce potential damage.
- Use **read-only** database access where possible.

Severity: High

Payload: ORDER BY 1 –

3] Event-Based XSS (`<img src=x onerror=alert(1)>`)

Description:

Event-Based Cross-Site Scripting (XSS) occurs when user-supplied input is injected into an HTML attribute, such as an `<img>` tag, and executed via JavaScript event handlers like `onerror`.
In this case, an attacker can insert a malicious payload inside an input field that allows HTML, causing the script to execute when the browser encounters a broken image.

This vulnerability allows attackers to execute arbitrary JavaScript in a victim's browser, leading to session hijacking, data theft, or defacement.

Impact:

Session Hijacking: Attackers can steal authentication cookies and impersonate users.
Phishing Attacks: Malicious scripts can modify page content to trick users into entering credentials.
Data Theft: Sensitive data, such as user credentials or payment details, can be stolen.
Defacement: Attackers can modify the web page's appearance and inject malicious content.

Steps to Reproduce (Proof of Concept - PoC):

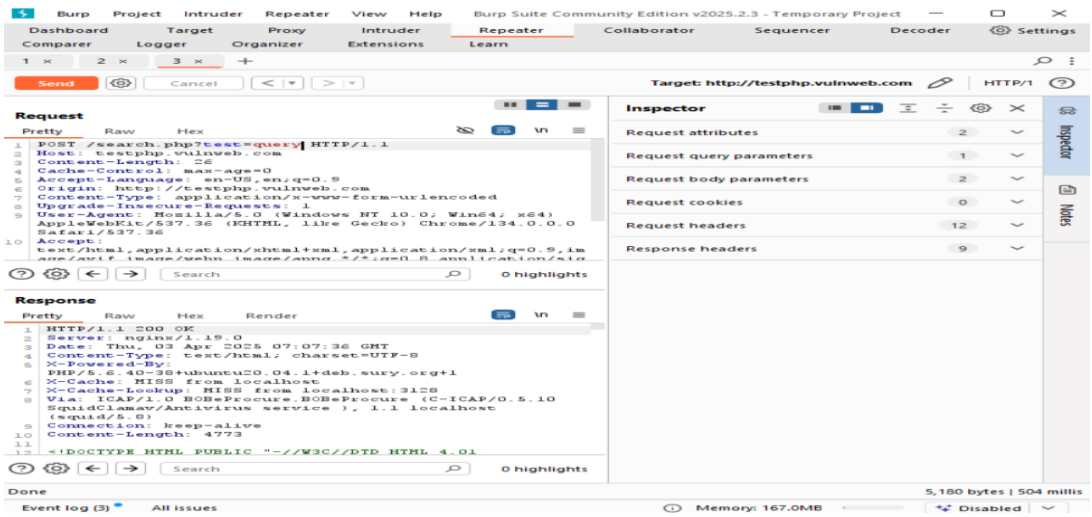
1]Injecting the payload into a vulnerable input field

- **Navigate to the vulnerable input field** on search box in the target website testphp.vulnweb.com
- **Enter the following payload and submit it:**

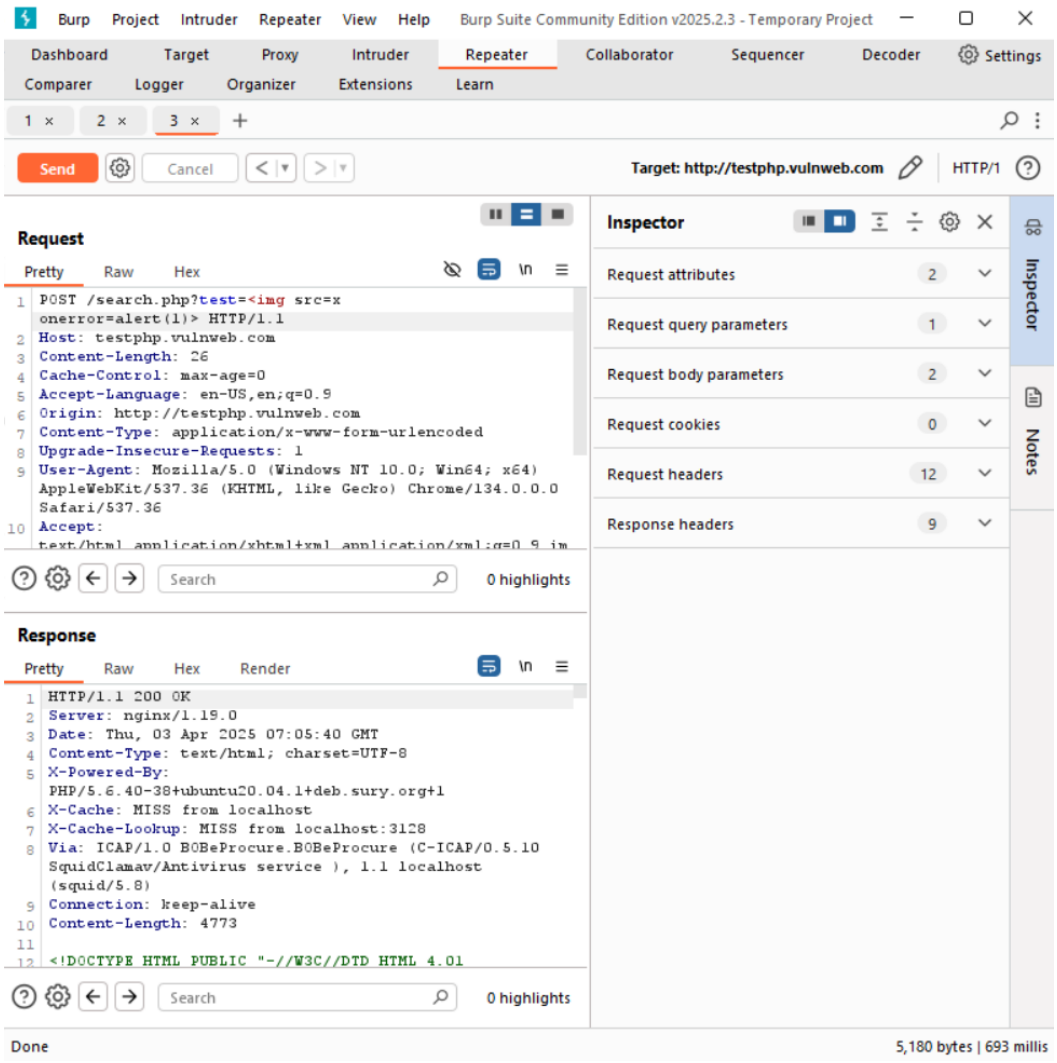
```
<img src=x onerror=alert(1)>
```

2.]Observe that the JavaScript executes, triggering an alert box.

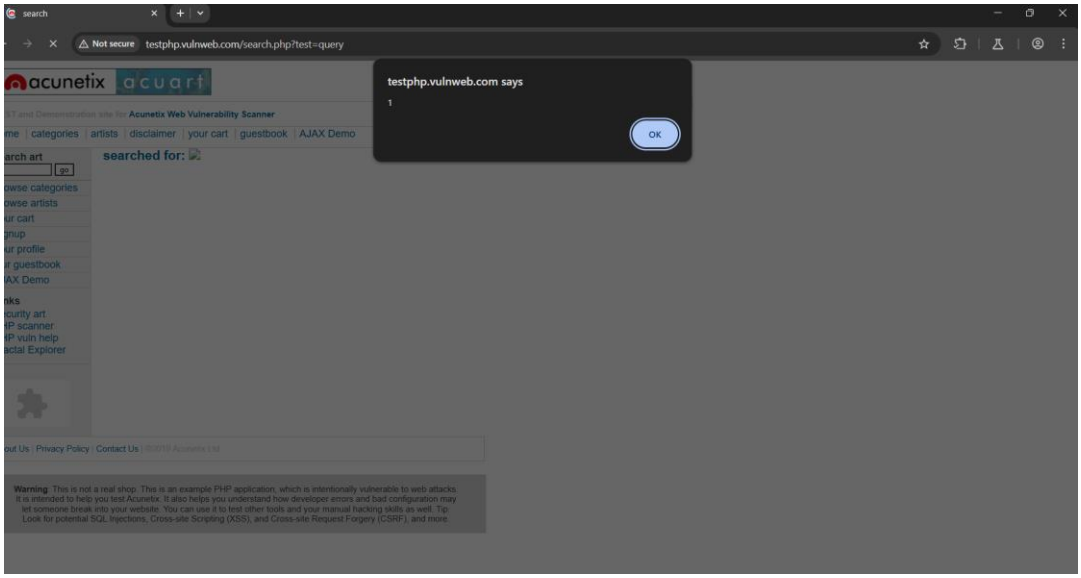
1] Before adding payload



2] After adding payload



3] Results:Alert popup



Suggested Fixes (Mitigation):

Input Validation: Sanitize user input to remove HTML tags and JavaScript event attributes (`onerror`, `onclick`, etc.).

Output Encoding:

- Encode < and > characters to prevent execution:
  - **HTML Encoding:** Convert < to `<` and > to `>`

Content Security Policy (CSP):

- Implement **CSP headers** to block inline JavaScript execution:

Use Security Libraries:

- Employ security libraries like **DOMPurify** to sanitize input.

Severity: High

Payload: <img src=x onerror=alert(1)>

4] VAPT Report Entry for Stored XSS (`<script>alert('XSS')</script>`)

Description:

Stored Cross-Site Scripting (XSS) occurs when an attacker injects malicious JavaScript code into a web application, and the payload gets **permanently stored** in the database. This payload is then executed whenever a victim visits the affected page.

In this case, an attacker can inject JavaScript code (e.g., `<script>alert('XSS')</script>`) into user input fields such as **comments, usernames, feedback sections, or support tickets**. When another user visits the page, the script executes in their browser.

This vulnerability **compromises user security** and can lead to **session hijacking, data theft, phishing, and website defacement**.

Impact:

Session Hijacking: Attackers can steal authentication cookies and impersonate users.
Phishing Attacks: Users can be tricked into entering credentials on fake login pages.
Data Theft: Sensitive data like email, passwords, or payment details can be extracted.
Worm Propagation: The XSS script can create self-replicating payloads affecting multiple users.
Defacement & Malware Injection: Attackers can modify the website's appearance and insert malicious links.

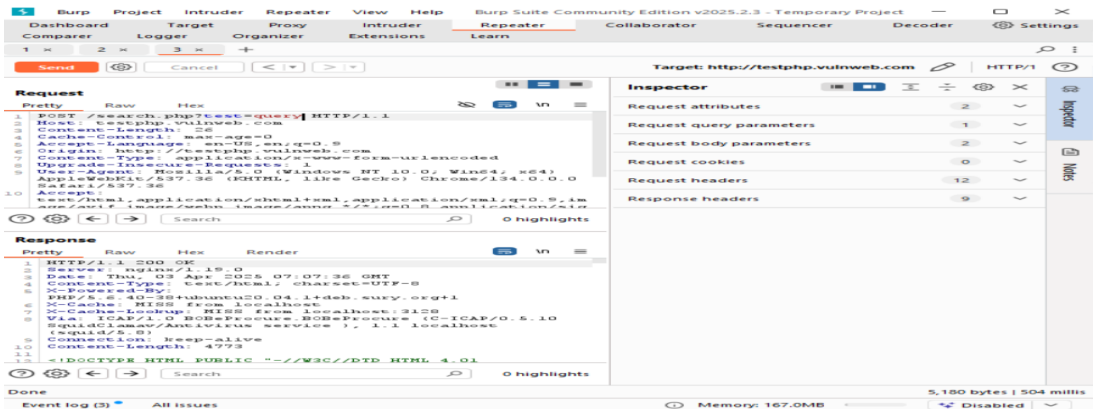
Steps to Reproduce (Proof of Concept - PoC):

1.**Navigate to a search input field** on the target website(testphp.vulnweb.com).
2. **Enter and submit the following payload:**

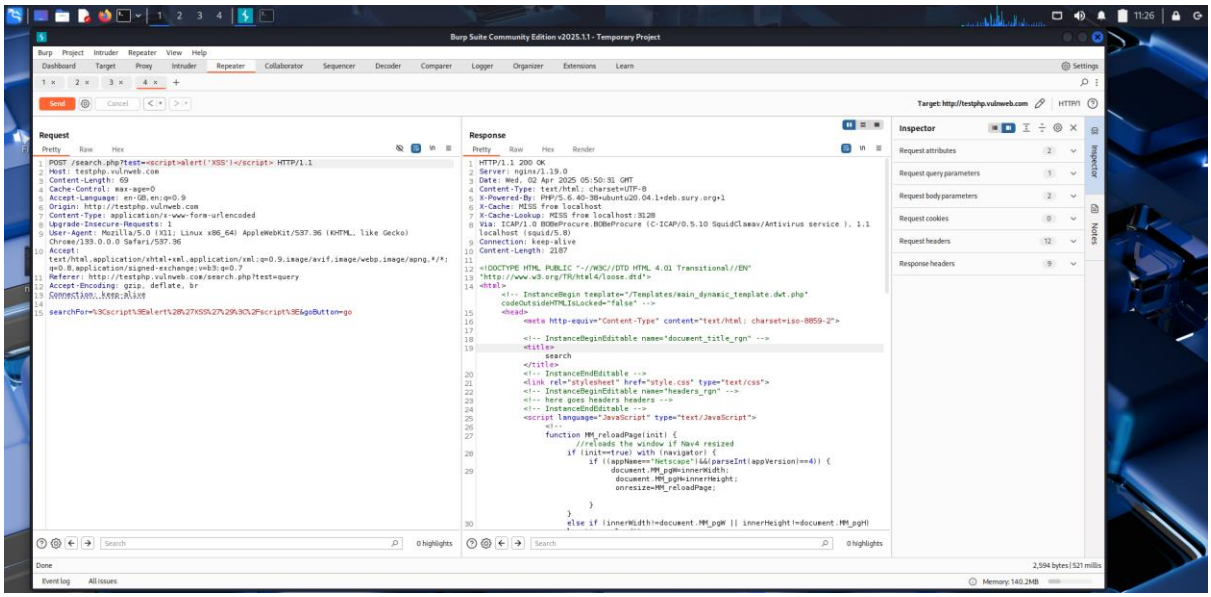
```
<script>alert('XSS')</script>
```

3. **Check if the script gets stored and executed** when the page reloads.
4. **If another user visits the affected page**, their browser will execute the script, displaying the alert box.

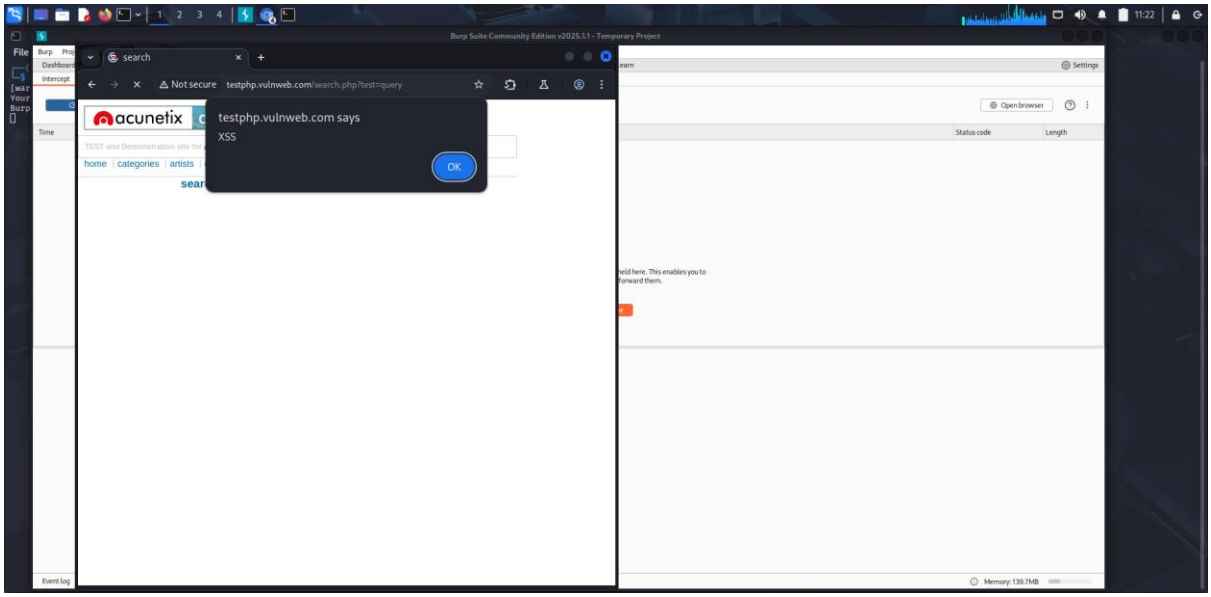
1] Before adding payload



2] Inserting payload



3]Results



Recommended Fixes (Mitigation):

Input Validation & Sanitization:

- **Sanitize user input** to remove harmful JavaScript tags:
  - Remove `<script>`, `<iframe>`, `<object>`, etc.
- Use **allowlists** to permit only safe input formats (e.g., plain text).

Output Encoding:

- Encode < and > characters to prevent execution:

Content Security Policy (CSP):

- Implement **CSP headers** to block inline JavaScript execution:
- `Content-Security-Policy: script-src 'self'; object-src 'none'`

Use Security Libraries:

- Employ security libraries like **DOMPurify** to clean input before storage.

Database-Level Protection:

- **Escape input data** before storing it in the database.

Severity: Critical

Payload: (`<script>alert('XSS')</script>`)

5] Version Disclosure

Description:

Version Disclosure occurs when a web application **leaks sensitive information** about its underlying technologies, such as the **web server version, database version, framework, or CMS**. Attackers can use this information to identify vulnerabilities specific to that version and launch targeted exploits.

Common sources of version disclosure:
1]HTTP Response Headers (e.g., `Server: Apache/2.4.49` or `X-Powered-By: PHP/7.4.3`)
2]Error Messages that expose framework details
3]JavaScript Files that reference specific libraries with version numbers
4]Meta Tags in HTML (e.g., `<meta name="generator" content="WordPress 5.9">`)

Impact:

Targeted Exploitation: Attackers can look up known vulnerabilities for disclosed versions.
Brute Force & Automated Scanning: Hackers may use tools like **Shodan, Wappalyzer, or Nmap** to scan for outdated software.
Increased Attack Surface: Even if the version itself isn't vulnerable, attackers might use this data for **social engineering attacks**.

Steps to Reproduce (Proof of Concept - PoC):

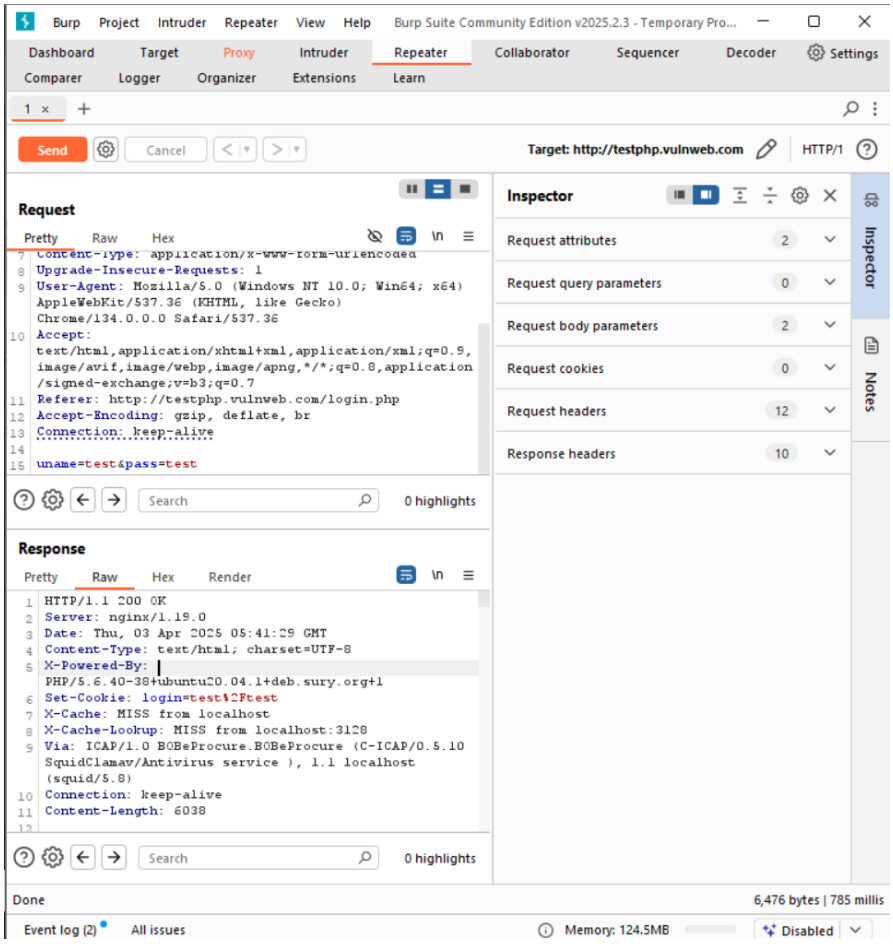
Identifying the Web Server Version from Response Headers

1. Open **Burp Suite, ZAP, or Developer Tools (F12 in Chrome/Firefox)**

2.If you see version details like:

```
Server: Apache/2.4.49
X-Powered-By: PHP/7.4.3
```

This confirms the vulnerability.



Recommended Fixes (Mitigation):

Hide Version Information in Headers:

- Remove or modify headers in **Apache/Nginx/PHP configuration**:

Customize Error Pages:

- Configure the web application to display **generic error messages** instead of revealing software versions.

Remove Meta Tags with Version Details:

- Avoid using `<meta name="generator">` in HTML.

Implement a Web Application Firewall (WAF):

- Use a WAF (e.g., **Cloudflare, ModSecurity**) to filter out requests attempting to extract version details.

Severity: Medium

6] Broken Authentication via Cluster Bomb Attack (Intruder - Burp Suite)

Description:

Broken authentication occurs when **weak authentication mechanisms** allow attackers to bypass login security through **brute force** or **credential stuffing** techniques.

Using **Burp Suite Intruder** with the **Cluster Bomb attack type**, attackers can inject a **large set of username-password combinations** to guess valid credentials and gain unauthorized access.

In this attack:
1]The **username and password fields** are tested with multiple payload lists.
2]Intruder's Cluster Bomb attack tries every possible combination of values.
3]If a **successful response code or behavior change** is detected, authentication is compromised.

Impact:

Account Takeover: Attackers can compromise user accounts and escalate privileges. **Data Theft:** Unauthorized access can expose sensitive data.
System Compromise: If an admin account is breached, attackers can manipulate the application.

Steps to Reproduce (PoC) :

Step 1: Intercept the Login Request

1.Open **Burp Suite** and ensure **Intercept Mode** is **ON**.
2. Try logging in with any username and password.
3. In **Burp Suite's HTTP history**, find the `POST` request to the login endpoint

4. Right-click on the request → **Send to Intruder**.

Step 2: Configure Intruder Attack Type

1. In Burp Suite, go to **Intruder → Positions** tab.
2. Set **Attack Type** to **Cluster Bomb**.
3. Select the `username` and `password` parameters and **add payload markers**:

```
username=§USER§&password=§PASS§
```

4. This setup will try multiple usernames and multiple passwords in combinations.

Step 3: Load Username and Password Lists

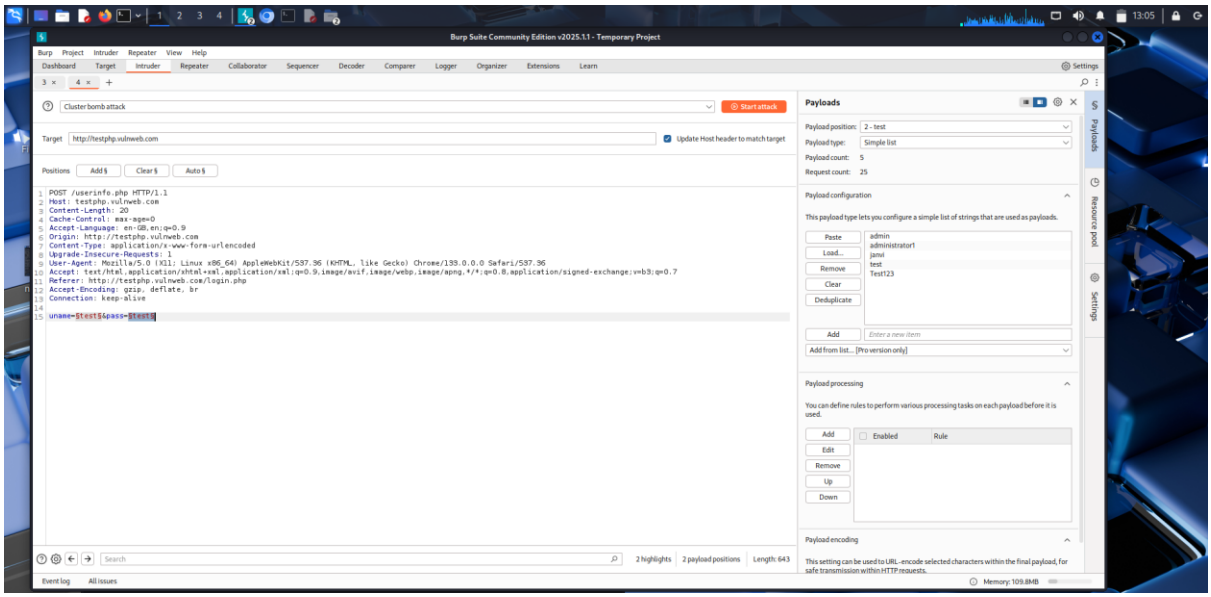
1.Navigate to the **Payloads** tab.
2. For **Payload Set 1 (USER)**, load a list of common usernames (`admin, test, root, user1, guest`).

3.For **Payload Set 2 (PASS)**, load common passwords (`123456`, `password`, `admin`, `qwerty`, `test`, `test123`).
4□.Click **Start Attack**.

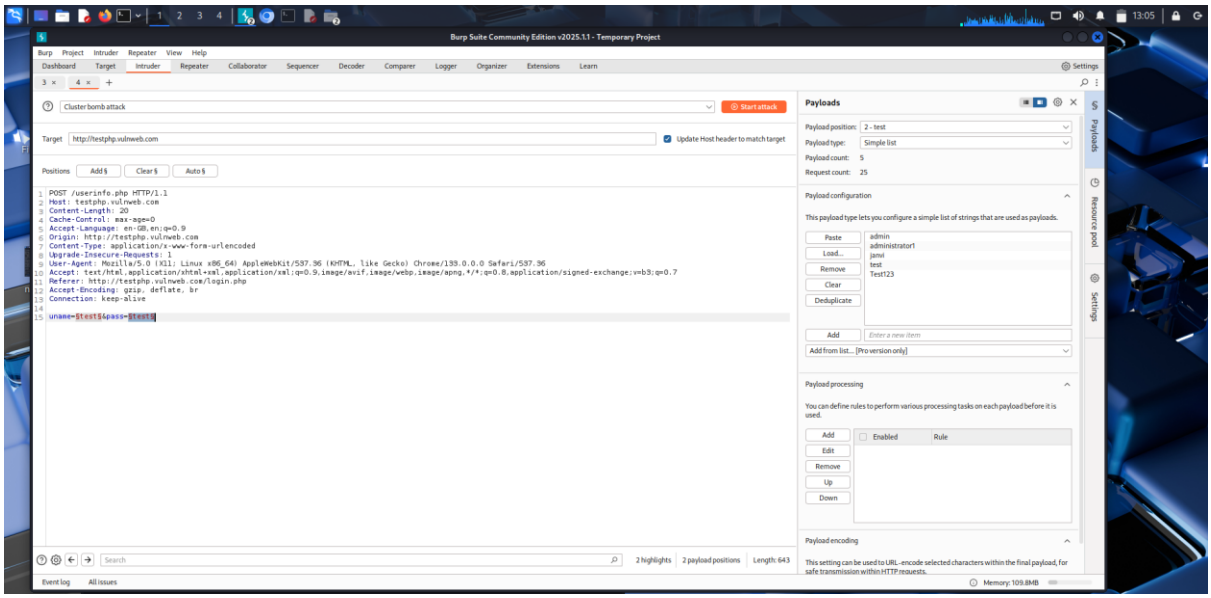
Step 4: Analyze Responses

1.Look for **response differences** (e.g., status code `200`, different response length).
2. A successful login will typically return a different response than failed attempts.

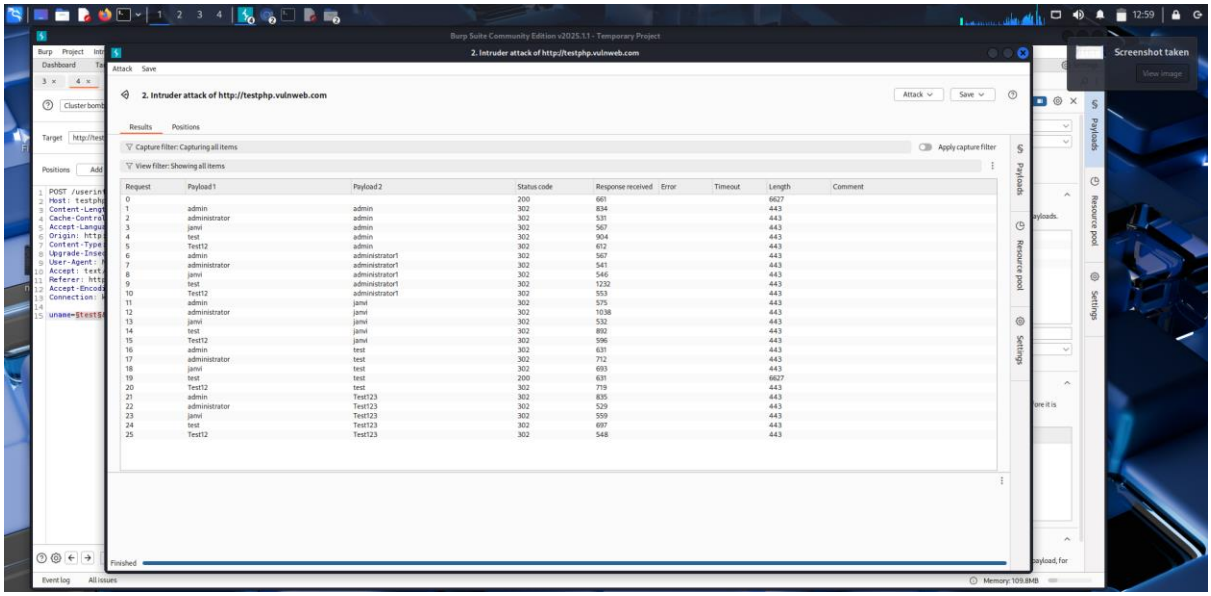
1]Adding payload Markers



2]Adding payload 2



3]Results with status Code



Recommended Fixes (Mitigation):

Implement Account Lockout Mechanism:

- Lock accounts after multiple failed attempts (e.g., after **5 incorrect attempts**).

Use Strong Password Policies:

- Enforce **complex passwords** (e.g., min 12 characters, mix of uppercase, lowercase, numbers, symbols).

Enable Multi-Factor Authentication (MFA):

- Require **OTP, email verification, or biometric authentication** for login.

Rate Limiting & CAPTCHA:

- Use **CAPTCHA** after a certain number of failed login attempts.

Monitor and Log Failed Login Attempts:

- Detect brute-force attempts by logging **IP addresses and failed logins**.

Implement Web Application Firewall (WAF):

- Block automated attacks using **Cloudflare, ModSecurity, or AWS WAF**.

Severity: High

7] Cross-Site Request Forgery (CSRF) in Search Functionality

Description:

Cross-Site Request Forgery (CSRF) is a **web security vulnerability** that allows an attacker to trick an authenticated user into unknowingly executing unwanted actions on a web application.

In this case, **the search functionality** of `testphp.vulnweb.com` is vulnerable to CSRF.

- Since the **search request is a POST request**, an attacker can craft a **malicious HTML form** to **silently submit a search request on behalf of a logged-in user**.
- If the user is authenticated and clicks on a **malicious link**, the search request gets executed **without their consent**.

Impact:

User Data Manipulation: Attackers can perform unauthorized actions using the victim's session.
Information Exposure: An attacker can execute searches on behalf of the victim and potentially retrieve sensitive data.
Stored CSRF Risks: If CSRF is used to modify stored data (e.g., updating user profiles), it could have **long-term impacts**.

Steps to Reproduce (PoC - Proof of Concept):

Step 1: Identify the Search Request

1.Perform a **search query** on `testphp.vulnweb.com`.
2.Use **Burp Suite** to inspect the **POST request**.

Step 2: Create a CSRF Exploit Page

An attacker can create an HTML file (`csrf_exploit.html`) containing the **malicious CSRF form**:

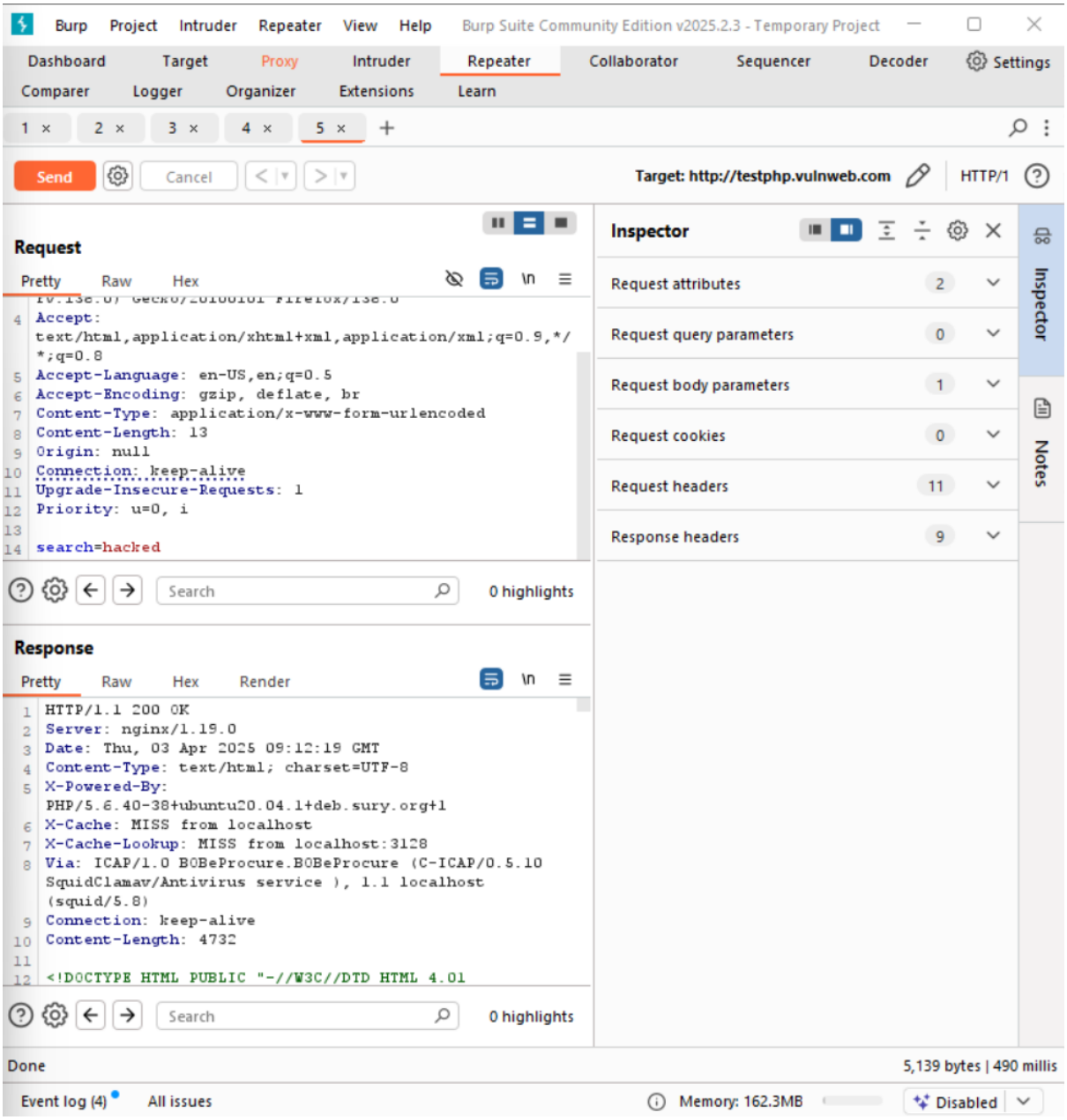
```
<!DOCTYPE html>
<html>
  <body onload="document.getElementById('csrfForm').submit()">
    <form id="csrfForm" action="http://testphp.vulnweb.com/search.php" method="POST">
      <input type="hidden" name="search" value="hacked">
    </form>
  </body>
</html>
```

Step 3: Trick the Victim

- Share the link with a logged-in victim through:
  a]Email (phishing attack)
  b]Social media posts
  c]Embedded in an **image** or **button**

Step 4: Observe the Request Execution

- If the victim clicks the link **while logged into** `testphp.vulnweb.com`, their browser automatically submits the **search request** without their knowledge.
- This **proves CSRF vulnerability** in the search functionality.



Recommended Fixes (Mitigation):

Implement CSRF Tokens:

- Require a **unique token** in every **POST request**, making CSRF attacks ineffective.
- Example:
- `<input type="hidden" name="csrf_token" value="random_token_value">`
- The server should verify this token before processing any requests.

Use SameSite Attribute in Cookies:

- Configure session cookies to include the `SameSite` attribute:
- `Set-Cookie: session_id=xyz123; Secure; HttpOnly; SameSite=Strict`
- This prevents cookies from being sent during **cross-site requests**.

Enable User Authentication for Sensitive Actions:

- Ensure that search requests **require user confirmation** (e.g., a CAPTCHA).

Verify Referrer Headers:

- Reject requests where the `Referer` or `Origin` header does not belong to `testphp.vulnweb.com`.

Implement CORS Policy Restrictions:

- Restrict which origins are allowed to send requests to the application.

Severity: Medium