# Target: DVWA and OWASP Juice Shop

## Threat Mapping and Penetration Testing Report

PRATHAM VERMA | 10-JUNE-2025

# Table of Content

# Introduction

In the ever-evolving landscape of cybersecurity, web applications have become increasingly critical to modern digital infrastructure while simultaneously emerging as one of the most commonly exploited attack surfaces. Due to their widespread use, complex user interactions, and frequent integration with sensitive data and third-party services, web applications remain a primary focus for cybercriminals. Threat actors continuously discover and exploit vulnerabilities introduced during development, deployment, or maintenance phases. As such, understanding and mitigating these risks is vital for ensuring data protection, privacy, and the trust of end users.

This report focuses on a comprehensive threat mapping and vulnerability assessment of two deliberately insecure web applications: **Damn Vulnerable Web Application (DVWA)** and **OWASP Juice Shop**. These platforms are specifically designed for security training and testing purposes, replicating real-world attack vectors aligned with the OWASP Top 10 framework — a globally recognized standard for web application security risks.

The primary objective of this assessment is to identify and document the presence of critical and high-risk vulnerabilities, explore realistic attack scenarios, and understand how attackers may leverage these flaws to compromise systems. Additionally, it aims to assess the associated risk levels and propose actionable mitigation techniques that can be implemented in production-grade environments. The scope includes vulnerabilities such as SQL Injection, Cross-Site Scripting (XSS), Broken Authentication, and Insecure Access Controls, among others.

Through this analysis, the report serves as a foundational guide for security professionals, developers, and students. It reinforces the importance of secure coding practices, highlights the consequences of poor configurations, and promotes the adoption of regular security assessments, monitoring, and defense-in-depth strategies to enhance the security posture of modern web applications.

# Executive Summary

This threat assessment report presents a comprehensive analysis of two purposely vulnerable web applications: Damn Vulnerable Web Application (DVWA) and OWASP Juice Shop. These platforms are widely utilized in the cybersecurity community for training, simulation, and vulnerability testing, providing a controlled environment to replicate real-world scenarios. The goal of this evaluation is to identify, understand, and assess security weaknesses based on the OWASP Top 10 vulnerability categories, which are considered industry benchmarks for web application security.

Through detailed manual and automated testing using advanced tools such as Burp Suite, OWASP ZAP, this assessment has uncovered multiple high-impact vulnerabilities that are commonly found in production environments. Key findings include:

- **SQL Injection (SQLi)** and **Blind SQL Injection** vulnerabilities, which enable attackers to manipulate database queries and retrieve sensitive information through crafted payloads and time-based inference techniques.
- **Cross-Site Scripting (XSS)** vulnerabilities in all three major forms — Stored, Reflected, and DOM-Based — which could be exploited by attackers to hijack user sessions, deface web pages, or execute arbitrary scripts within the user's browser.
- **Broken Authentication** mechanisms, which expose the application to credential-stuffing and brute-force attacks due to a lack of rate-limiting and CAPTCHA enforcement.
- **Broken Access Control** flaws that allow unauthorized users to gain access to restricted functions or data, posing serious threats to the confidentiality and integrity of user information.
- **Sensitive Data Exposure** due to poor handling of personal and financial information, combined with insecure transmission channels or misconfigured APIs.

These vulnerabilities mirror those frequently exploited in real-world cyberattacks, reinforcing the urgent need for organizations to integrate security into all stages of the development lifecycle. The insights and lessons from this exercise serve as a valuable resource for improving web application resilience. Each identified issue is accompanied by a technical explanation, risk evaluation, real-world exploitation example, and tailored mitigation strategy aimed at strengthening the overall security posture.

Ultimately, this report underlines the importance of continuous security testing, secure coding standards, and security awareness for development and operational teams. With the growing sophistication of cyber threats, proactive measures and informed security practices are essential to safeguard applications from exploitation.

# Identified Vulnerabilities

## DVWA Vulnerabilities:

| Vulnerability | Description | Severity |
|---|---|---|
| **SQL Injection (SQLi)** | Unsanitized user input allows database queries manipulation. | **High** |
| **Blind SQL Injection** | SQLi attack without visible error messages, tested via time-based techniques. | **High** |
| **Stored XSS** | Malicious script stored in database and executed on user visits. | **Medium** |
| **Reflected XSS** | Injected script reflected in user input (e.g., URL parameters). | **Medium** |
| **DOM-Based XSS** | Client-side JavaScript manipulates the DOM to execute scripts. | **Medium** |
| **Cross-Site Request Forgery (CSRF)** | Attacker tricks victim into executing unwanted actions. | **Low** |

## OWASP Juice Shop Vulnerabilities:

| Vulnerability | Description | Severity |
|---|---|---|
| **Broken Access Control** | Users can access unauthorized resources. | **High** |
| **Security Misconfiguration** | Misconfigured headers, error handling, and debug mode enabled. | **Medium** |
| **Sensitive Data Exposure** | API leaks personal & payment information. | **High** |
| **Broken Authentication** | No rate limiting allows brute-force login attempts. | **High** |
| **Captcha Bypass** | CAPTCHA protection is easily bypassed using automation and OCR tools. | **Medium** |

# Proof of Concept (PoC) Evidence

## Vulnerability #1: SQL Injection (SQLi) in DVWA

During the assessment of DVWA, I identified a classic case of SQL Injection vulnerability in the login module of the application. To begin the exploitation, I navigated to the login page and filled the **username** field with the payload a' **UNION SELECT 1,2**--, while leaving the **password** field blank. Using Burp Suite's intercept feature, I captured the login request before submission and carefully modified the parameters to include the payload. The injected SQL statement was designed to bypass the authentication logic by appending a malicious **UNION SELECT** clause that forces the database to return results from a crafted query. Once I forwarded the manipulated request, the application authenticated me as a valid user without checking for the password, indicating that the injected payload successfully altered the backend SQL logic.

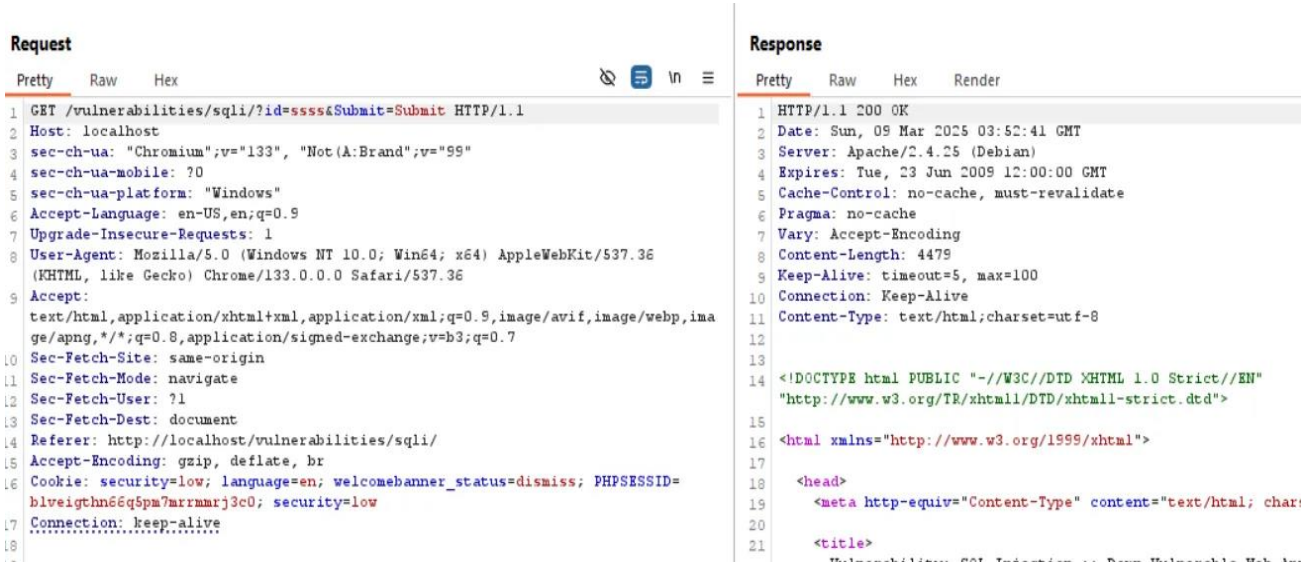
The SQL Injection works by closing the original query string and merging it with a crafted query that aligns with the expected number of output columns. Since the database accepted the query and returned a valid result, it clearly showed that the application directly embedded user input into SQL queries without any form of sanitization or use of parameterized statements. This demonstrated a serious flaw in how user inputs were handled in authentication logic.

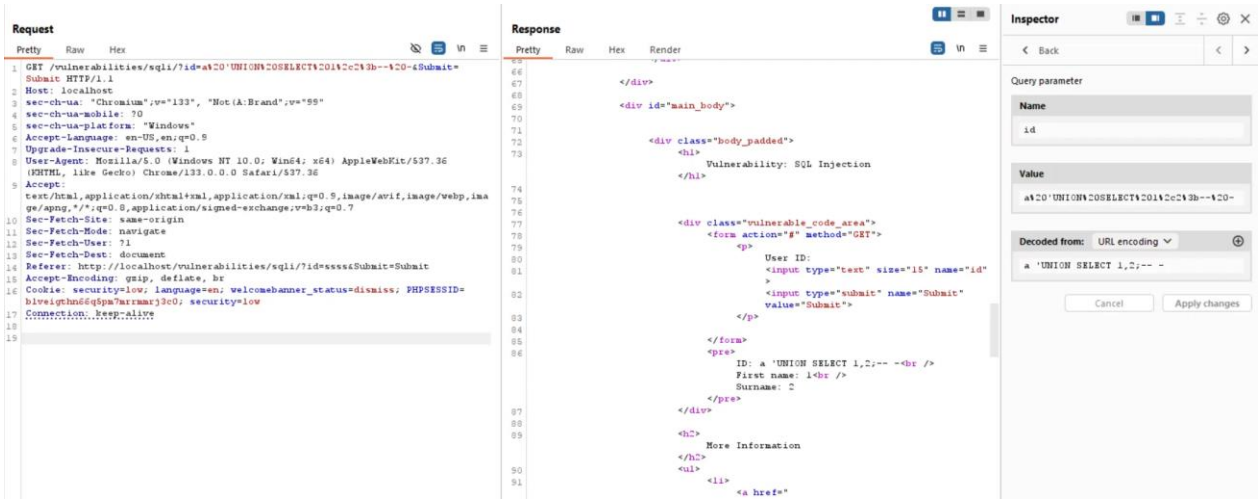
**Key technical details:**

- **Intercepted URL:** /login.php
- **Payload used:** **a' UNION SELECT 1,2-- -**
- **Injection method:** Inline SQL injection via login form
- **Tools used:** DVWA (Low security mode), Burp Suite (Intercept & Forward)
- **Observation:** Access granted without valid credentials, confirming query manipulation

This attack clearly highlights how improper input handling can lead to full system compromise. Therefore, it's vital for developers to adopt secure coding practices, especially around user input that interacts with backend queries.

# Proof of Concept (PoC):




Request
Pretty Raw Hex
GET /vulnerabilities/sqli/?id=a%20'UNION%20SELECT%201%2c2%3b--%20-&Submit=Submit HTTP/1.1
Host: localhost
sec-ch-ua: "Chromium";v="133", "Not(A:Brand";v="99"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Accept-Language: en-US,en;q=0.9
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/133.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost/vulnerabilities/sqli/?id=ssss&Submit=Submit
Accept-Encoding: gzip, deflate, br
Cookie: security=low; language=en; welcomebanner_status=dismiss; PHPSESSID=blveigthn66q6pm7mrrmmrj3c0; security=low
Connection: keep-alive
Response
Pretty Raw Hex Render

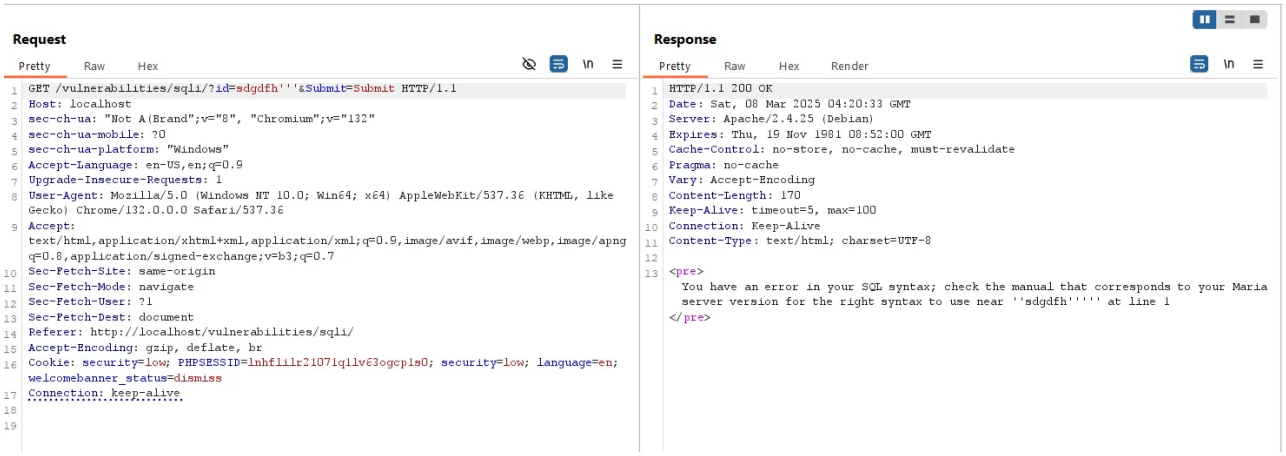
<h1>
Vulnerability: SQL Injection
</h1>

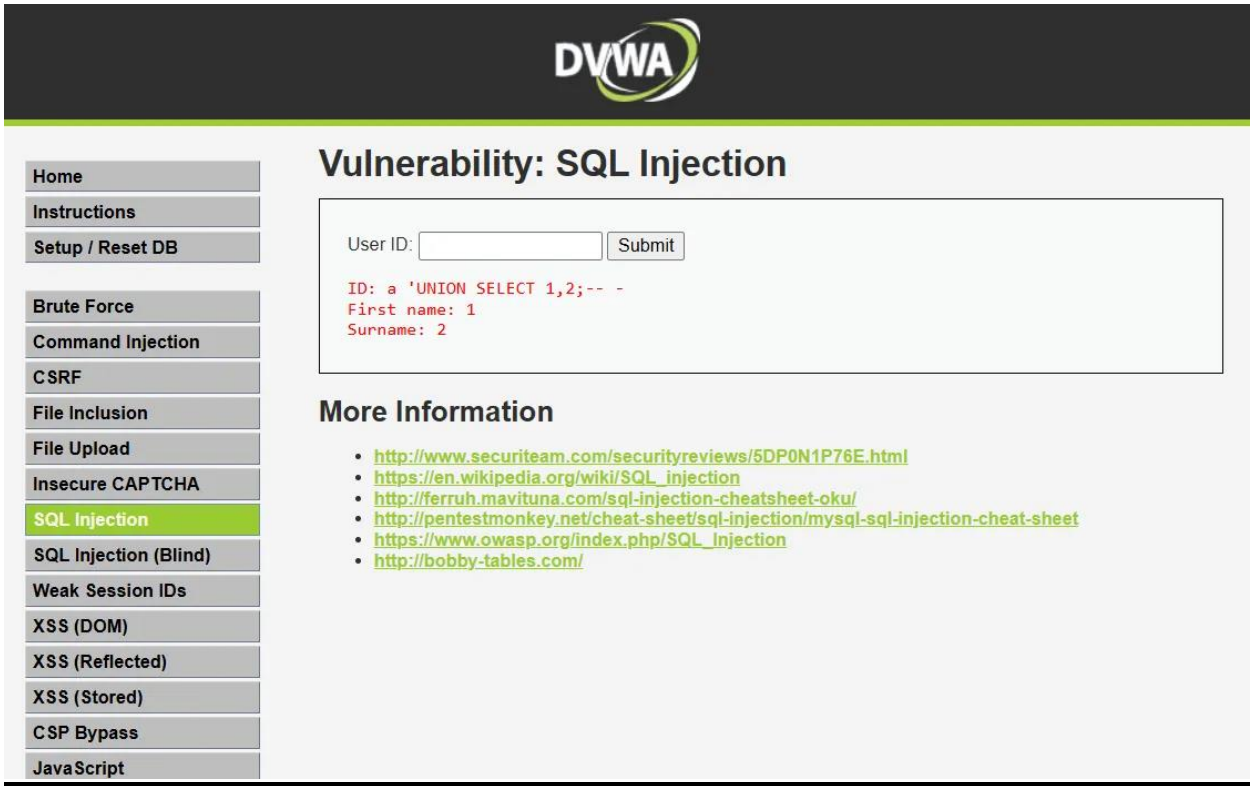
<form action="#" method="GET">

User ID:
<input type="text" size="15" name="id">
<input type="submit" name="Submit" value="Submit">

</form>
<pre>
ID: a 'UNION SELECT 1,2;-- -<br />
First name: 1<br />
Surname: 2
</pre>

<h2>
More Information
</h2>
<ul>
<li>
<a href="
Inspector
Back
Query parameter
Name
id
Value
a%20'UNION%20SELECT%201%2c2%3b--%20-
Decoded from: URL encoding
a 'UNION SELECT 1,2;-- -
Cancel
Apply changes


Request
Pretty Raw Hex
GET /vulnerabilities/sqli/?id=sdgdfh'''&Submit=Submit HTTP/1.1
Host: localhost
sec-ch-ua: "Not A(Brand";v="8", "Chromium";v="132"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Accept-Language: en-US,en;q=0.9
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/132.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost/vulnerabilities/sqli/
Accept-Encoding: gzip, deflate, br
Cookie: security=low; PHPSESSID=lnhflilr21071qllv63ogcpls0; security=low; language=en; welcomebanner_status=dismiss
Connection: keep-alive
Response
Pretty Raw Hex Render
HTTP/1.1 200 OK
Date: Sat, 08 Mar 2025 04:20:33 GMT
Server: Apache/2.4.25 (Debian)
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Vary: Accept-Encoding
Content-Length: 170
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
<pre>
You have an error in your SQL syntax; check the manual that corresponds to your Maria server version for the right syntax to use near ''sdgdfh''''' at line 1
</pre>


DVWA
Home
Instructions
Setup / Reset DB
Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA
SQL Injection
SQL Injection (Blind)
Weak Session IDs
XSS (DOM)
XSS (Reflected)
XSS (Stored)
CSP Bypass
JavaScript
Vulnerability: SQL Injection
User ID:
Submit
ID: a 'UNION SELECT 1,2;-- -
First name: 1
Surname: 2
More Information
http://www.securiteam.com/securityreviews/5DP0N1P76E.html
https://en.wikipedia.org/wiki/SQL_injection
http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet
https://www.owasp.org/index.php/SQL_Injection
http://bobby-tables.com/

# Vulnerability #2: Blind SQL Injection in DVWA

The Blind SQL Injection vulnerability was identified within the "SQLi (Blind)" module of DVWA. This form of SQLi is more subtle compared to standard SQL injection, as it does not return visible error messages or data output. Instead, it relies on indirect clues such as timing delays to infer whether the injection was successful.

To initiate the attack, I entered the payload:

**1' AND SLEEP (5) --**

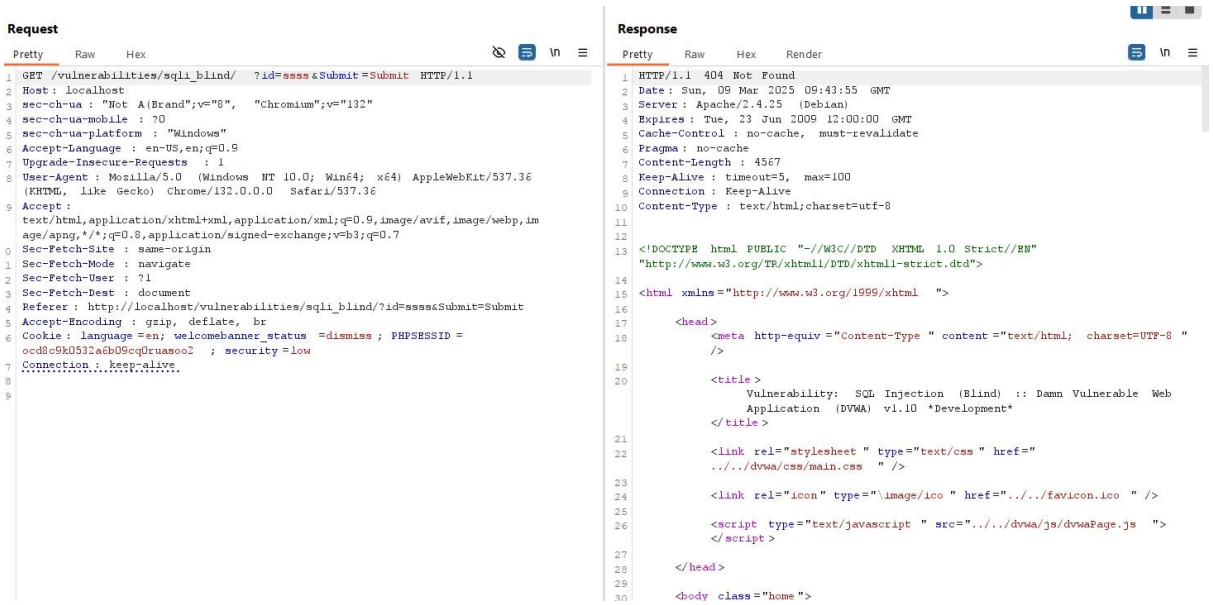
into the user input field. This payload instructs the backend database to pause for 5 seconds before responding if the input is being processed as part of a SQL query. If the query execution takes significantly longer than normal, it indicates that the application is vulnerable to SQL injection.

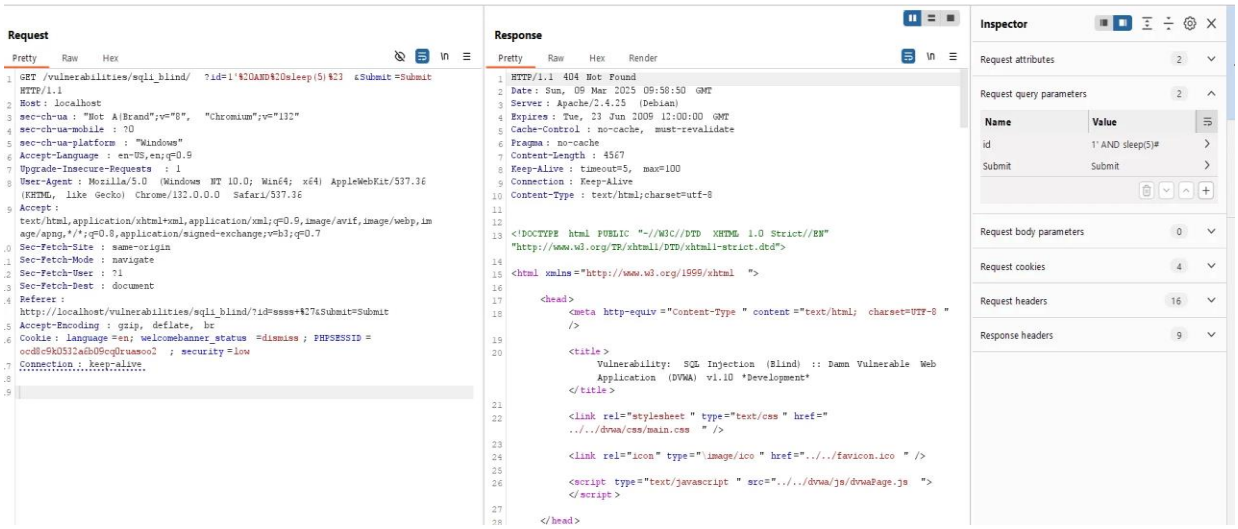
To confirm this behavior, I used Burp Suite to send the request and carefully monitored the response time. A noticeable 5-second delay validated that the **SLEEP (5)** function was executed successfully on the database server, meaning the input had been processed without any sanitization.

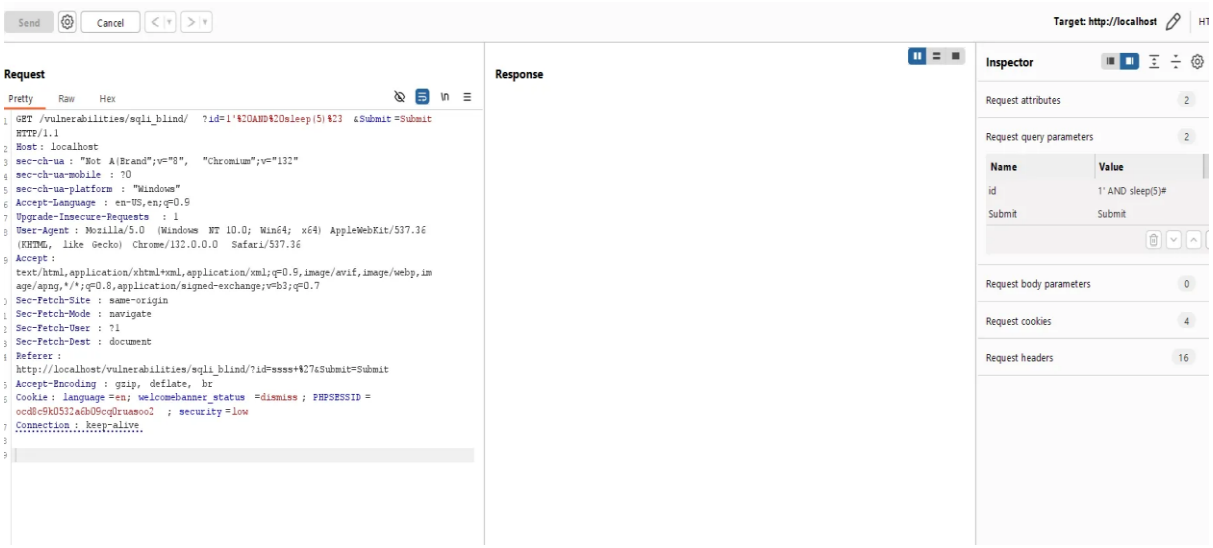
**Key Execution Points:**

- Accessed the SQLi (Blind) module in DVWA.
- Entered time-based payload **1' AND SLEEP (5) --**
- Observed a clear delay in the server response.
- Verified that database command execution confirms vulnerability.

This approach, known as time-based blind SQL injection, is especially powerful in scenarios where no error messages are displayed, allowing attackers to infer the behavior of the database silently. Iterative payloads can be used to extract specific database content one character at a time.






## Vulnerability #3: Stored Based XSS (Cross Site Scripting) in DVWA

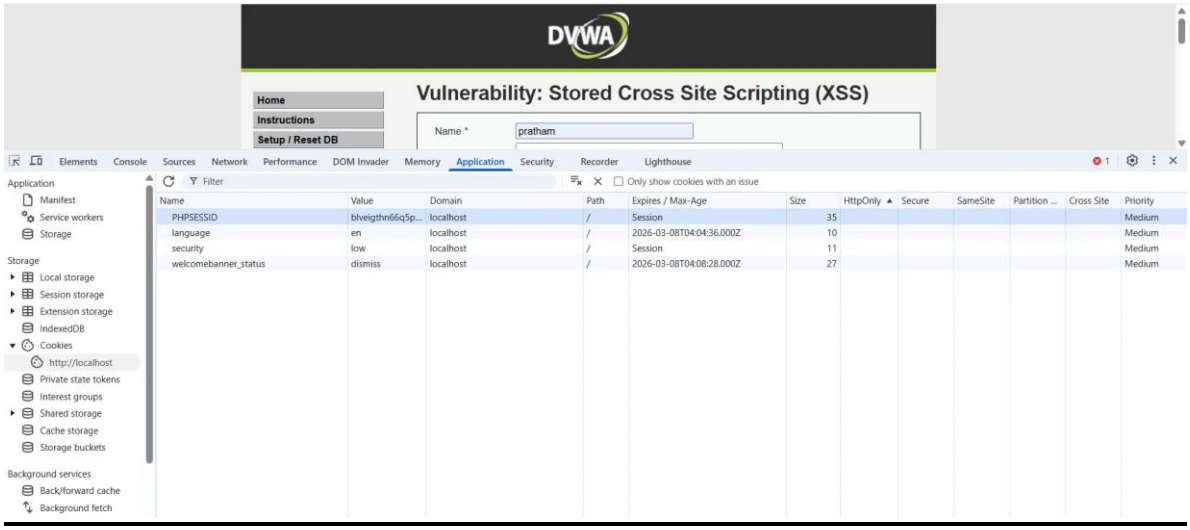
The Stored XSS vulnerability was identified in DVWA's "XSS (Stored)" module, which allows users to post messages that are displayed to all other users who visit the page. To perform this attack, I crafted the payload **<script> alert('XSS') </script>** and submitted it through the message/comment submission form.

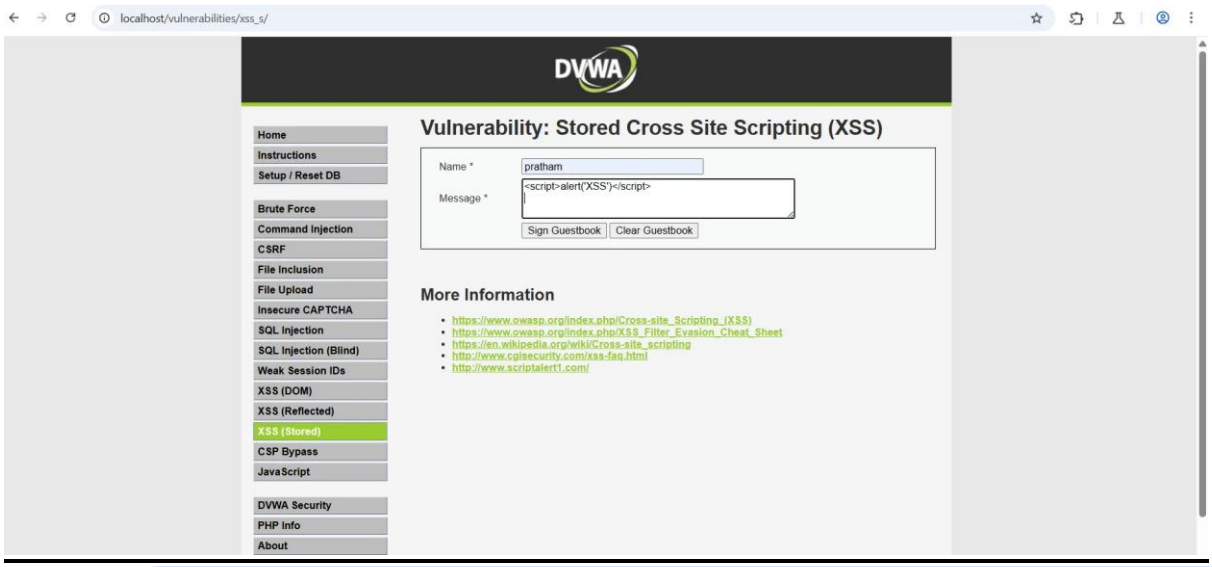
Once submitted, the payload was stored on the server and rendered as part of the web page every time it was accessed by another user. When the page was revisited, the browser executed the malicious JavaScript code, resulting in an alert box being displayed — clear evidence of script execution within the browser context.

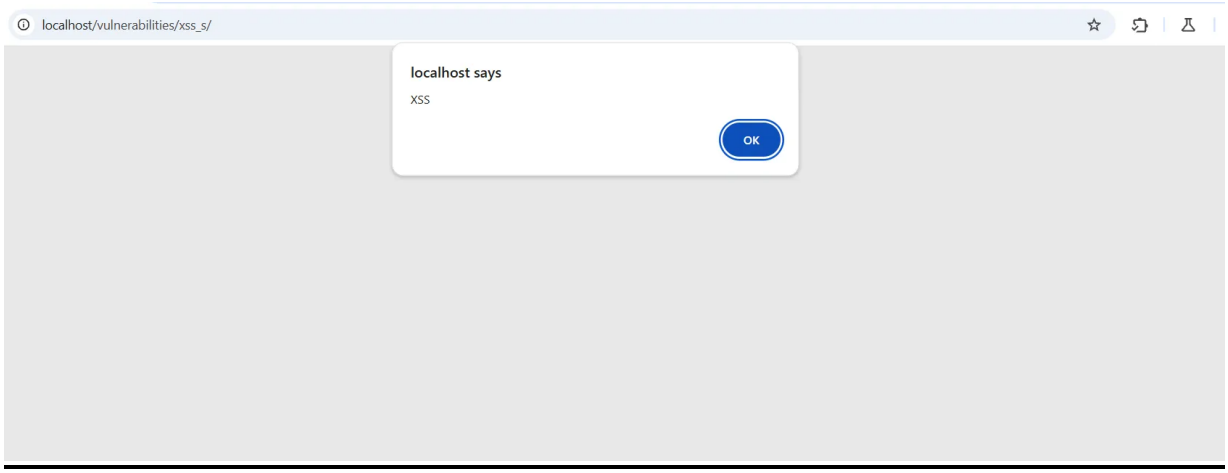
This vulnerability can be exploited by attackers to inject persistent JavaScript into the application, which is then executed on every user's browser. Such attacks can lead to session hijacking, phishing attacks, redirection to malicious sites, or complete control over the user's browser interactions with the application.

**Key Execution Steps:**

- Navigated to the Stored XSS module in DVWA.
- Injected JavaScript payload into the comment field.
- Submitted the payload and observed its persistent storage.
- Revisited the page and confirmed script execution via an alert box.



# Vulnerability #4: Reflected Based XSS (Cross-Site Scripting) in DVWA

The Reflected XSS vulnerability was discovered within the **"XSS (Reflected)"** module of DVWA. This type of XSS occurs when user-supplied data in the HTTP request is immediately returned by the server within the response HTML, without proper sanitization or output encoding. Unlike Stored XSS, the payload is not stored on the server but reflected back to the client, making it effective for attacks via malicious links or URLs.

**To test this vulnerability, I crafted the following URL:**

```
http://<DVWA-URL>/vulnerabilities/xss_reflected/?name=<script>alert('you are hacked')</script>
```

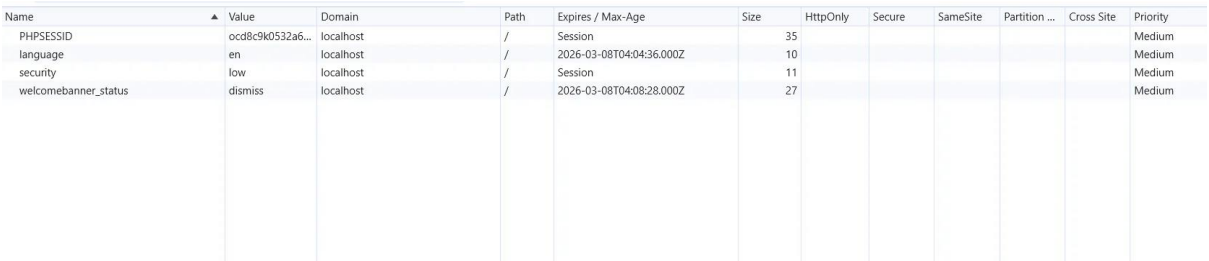
I manually entered this URL into the browser. Upon loading the page, the JavaScript **alert()** function executed immediately, displaying a popup with the message "you are hacked." This indicated that the input passed via the name parameter was embedded directly into the HTML output without being encoded or filtered, allowing the malicious script to run in the user's browser.

To verify the vulnerability's behavior further, I intercepted the request using Burp Suite and confirmed that the server response contained the raw payload. This validated that the backend was not handling input securely and reflected it as-is in the HTML response.

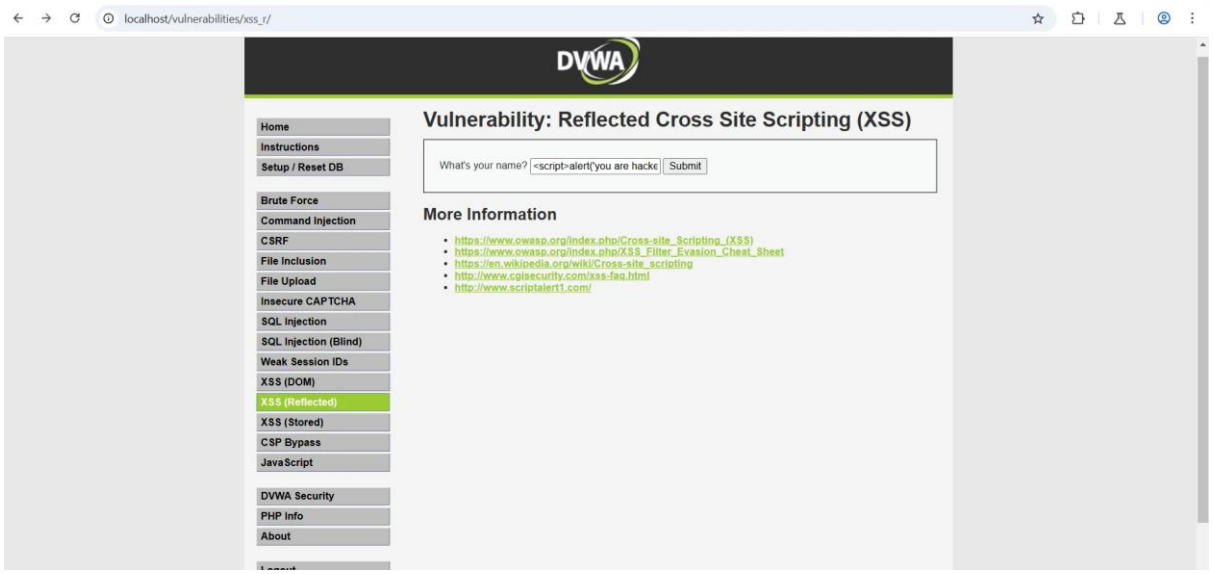
**Key Execution Steps:**

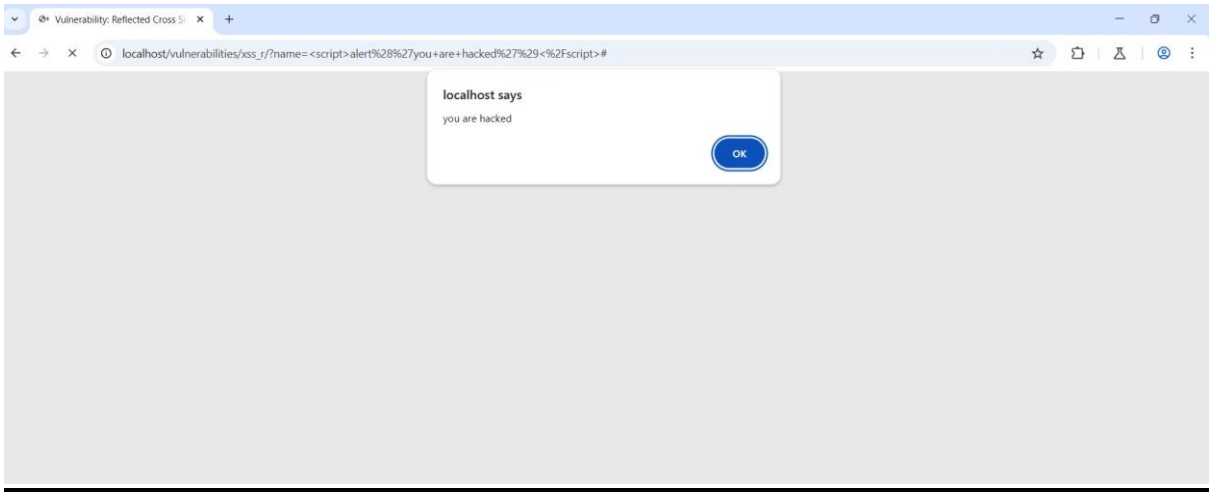
- Constructed a URL with a malicious script in the query parameter.
- Loaded the URL in a browser to observe immediate script execution.
- Used Burp Suite to confirm the payload was reflected without encoding.
- Verified that no input sanitization or output escaping was performed.

This proves that any user tricked into clicking a crafted link can have malicious JavaScript executed in their browser session, leading to session theft, unauthorized actions, or phishing. Reflected XSS is particularly dangerous in phishing scenarios where attackers embed malicious links in emails or third-party websites.



| Name | Value | Domain | Path | Expires / Max-Age | Size | HttpOnly | Secure | SameSite | Partition ... | Cross Site | Priority |
|---|---|---|---|---|---|---|---|---|---|---|---|
| PHPSESSID | ocd8c9k0532a6... | localhost | / | Session | 35 | | | | | | Medium |
| language | en | localhost | / | 2026-03-08T04:04:36.000Z | 10 | | | | | | Medium |
| security | low | localhost | / | Session | 11 | | | | | | Medium |
| welcomebanner_status | dismiss | localhost | / | 2026-03-08T04:08:28.000Z | 27 | | | | | | Medium |




## Vulnerability #5: DOM-Based XSS (Cross-Site Scripting) in DVWA

The DOM-Based XSS vulnerability was discovered in DVWA under the "**XSS (DOM)**" module. Unlike traditional XSS attacks where the malicious input is reflected by the server, DOM-Based XSS occurs entirely on the client side. The vulnerability arises due to insecure manipulation of user-controlled input directly within the browser's Document Object Model (DOM) using JavaScript.

To test this, I navigated to the module's vulnerable page and crafted the following URL containing a malicious script embedded in the default parameter:

```
http://<DVWA-URL>/vulnerabilities/xss_d/?default=french<script>alert('you are hacked')</script>
```

Upon pasting this URL into the browser, the script executed immediately, and the alert box popped up displaying the message "you are hacked." This proved that the input provided in the default parameter was being accessed and injected into the page's DOM using insecure JavaScript methods, such as innerHTML, without any sanitization or encoding.

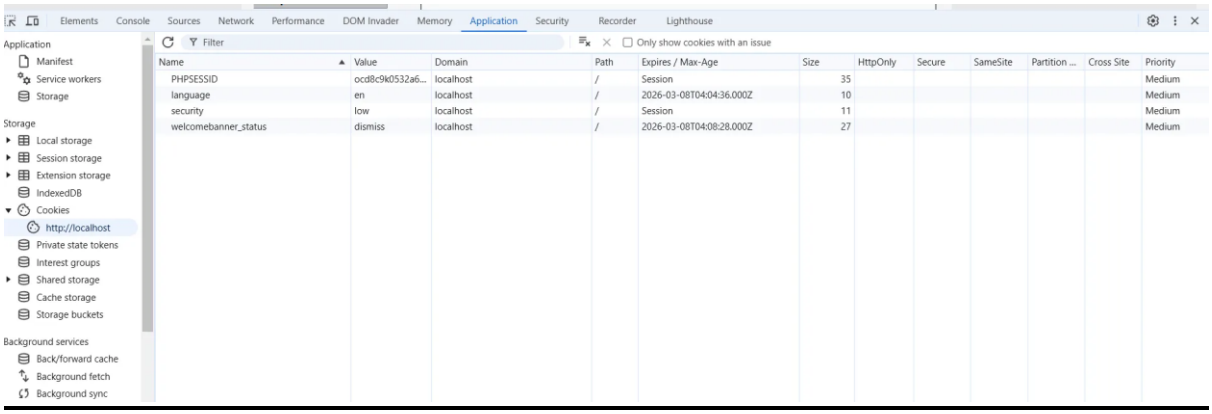
To confirm the vulnerability, I also reviewed the page source and observed that the client-side script dynamically inserted the default parameter's value directly into the DOM, which triggered the

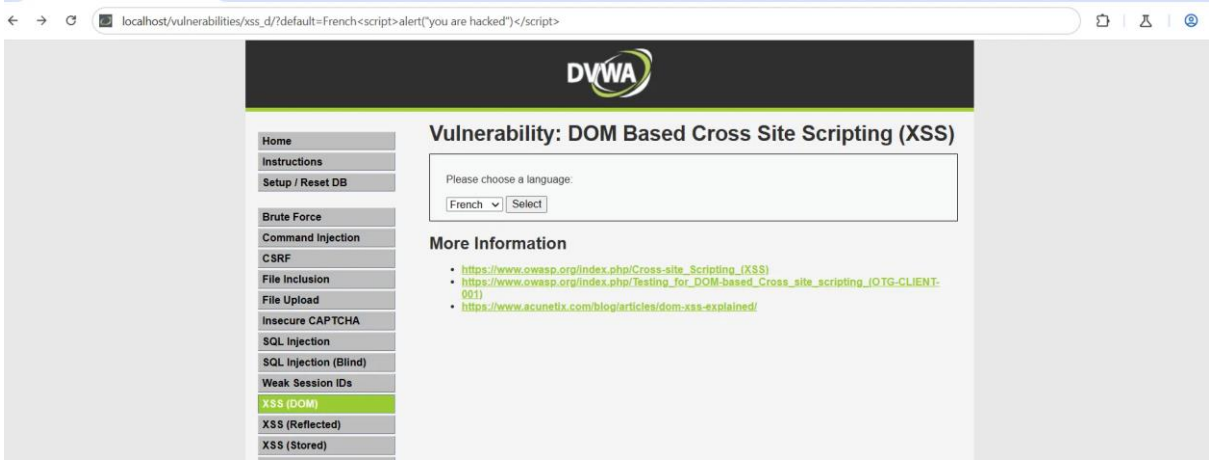
embedded <script> payload. Since this script executes purely in the browser context without any interaction with the backend server, traditional input validation on the server would not mitigate it — instead, DOM-based sanitization techniques must be applied.

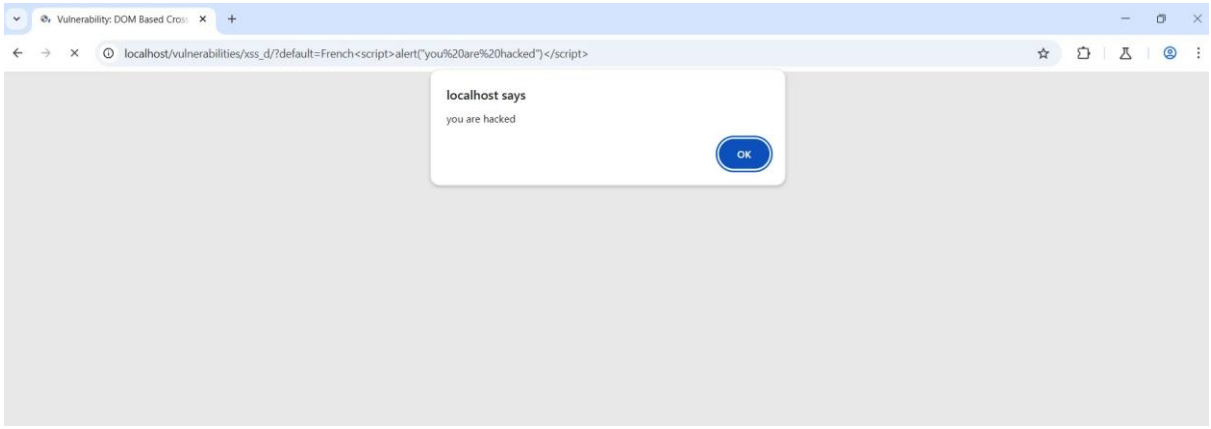
**Key Execution Steps:**

- Accessed the vulnerable page in DVWA's DOM XSS module.
- Crafted a malicious payload within the default parameter.
- Appended it to the URL and loaded the page.
- Observed script execution via an alert box in the browser.
- Verified that the vulnerability stemmed from unsafe DOM manipulation using innerHTML or similar methods.

This vulnerability highlights the importance of client-side security controls. Attackers can exploit it to run malicious JavaScript within the context of the affected user's session, leading to session hijacking, unauthorized actions, or theft of sensitive data — all without any server-side involvement.






## Vulnerability #6: CSRF (Cross-Site Request Forgery) in DVWA

The Cross-Site Request Forgery (CSRF) vulnerability was identified in the "CSRF" module of DVWA. This type of vulnerability allows an attacker to perform unauthorized actions on behalf of an authenticated user without their knowledge or explicit consent. In DVWA, the vulnerable functionality involves the ability to change a user's password via a GET or POST request without any CSRF protection mechanisms in place.

**To exploit this issue, I crafted a malicious URL that mimics the password change request:**

```
http://<DVWA-URL>/vulnerabilities/csrf/?password_new=password&password_conf=password&change=change#
```

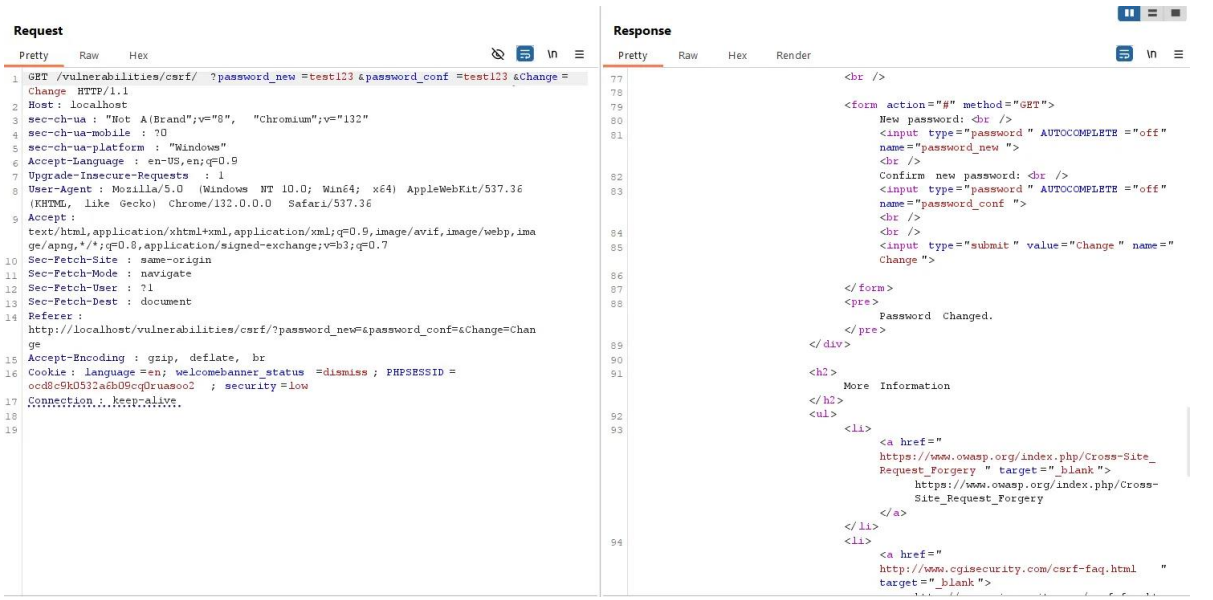
If a user who is already authenticated (logged in) clicks this link or is tricked into loading it (via a hidden image, iframe, or script on a malicious site), the request gets sent automatically with the user's active session cookie. Since DVWA does not validate the origin of the request and lacks CSRF tokens to verify request legitimacy, the server processes the change without prompting for the user's confirmation or password.

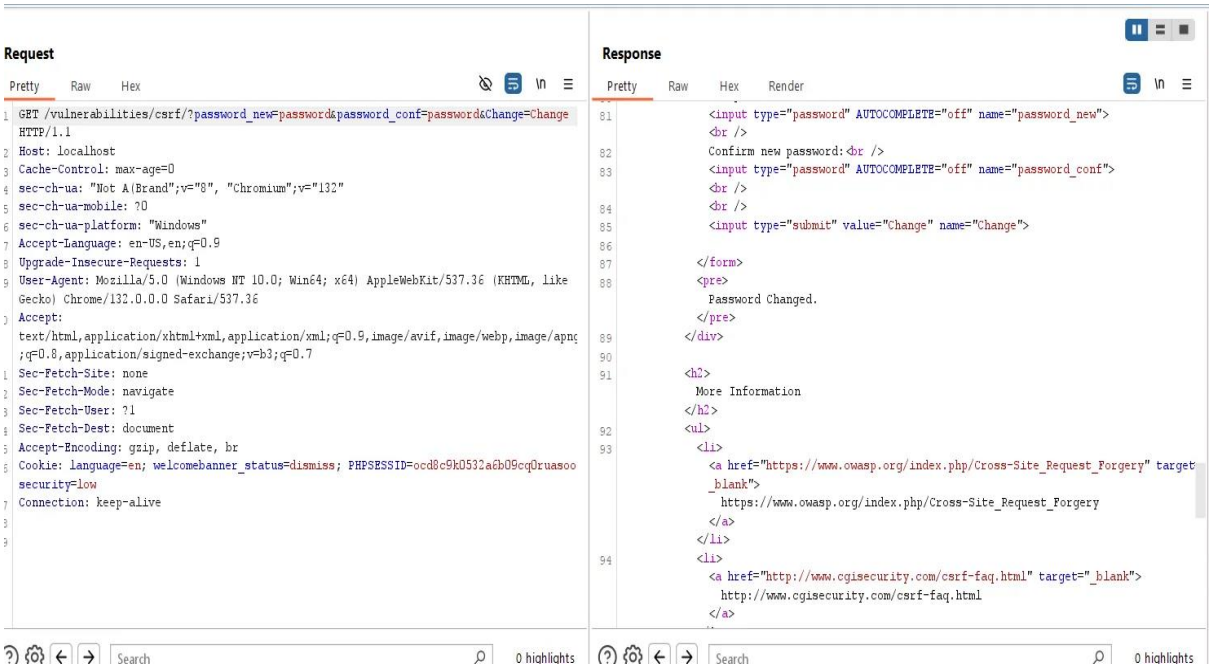
In a real-world scenario, this vulnerability could allow attackers to silently change user account settings, including credentials, by embedding such requests in phishing emails or compromised web pages. The lack of anti-CSRF tokens and the reliance on session cookies alone for authentication makes this possible.

**Key Execution Steps:**

- Navigated to the CSRF vulnerability page in DVWA.
- Observed the password change form lacked CSRF protection.
- Crafted a malicious URL mimicking a valid password change request.
- Tested it by loading the URL in an authenticated session.
- Confirmed that the user's password was changed without any prompt or notification.

This attack highlights how simple web requests — when not properly validated — can be used to execute unauthorized commands on behalf of unsuspecting users, especially when session cookies are sent automatically by the browser.




**Pratham Verma**

# <u>Vulnerability #1</u>: Broken Access Control in OWASP Juice Shop

This vulnerability was discovered by attempting to access restricted functionalities that should be available only to administrators. In OWASP Juice Shop, I noticed that navigating directly to the /administration endpoint — which is designed for administrative tasks — did not trigger any access restrictions when logged in as a regular user. There was no visible "Admin" link in the interface for normal users, but by manually entering the URL in the browser's address bar, I was able to reach the admin panel without authentication prompts or error messages.

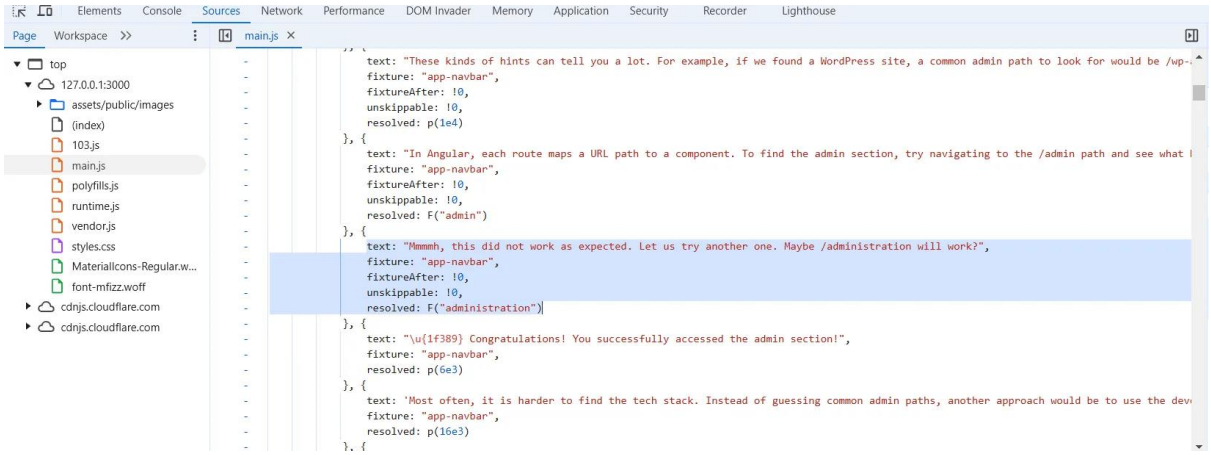
This indicates a failure in enforcing proper server-side authorization. The application hides admin functionality from the UI for unauthorized users, but it does not prevent access on the backend. This flaw reveals that the access control logic is implemented only on the client side (such as hiding elements using Angular or JavaScript), which can easily be bypassed by directly invoking restricted endpoints.

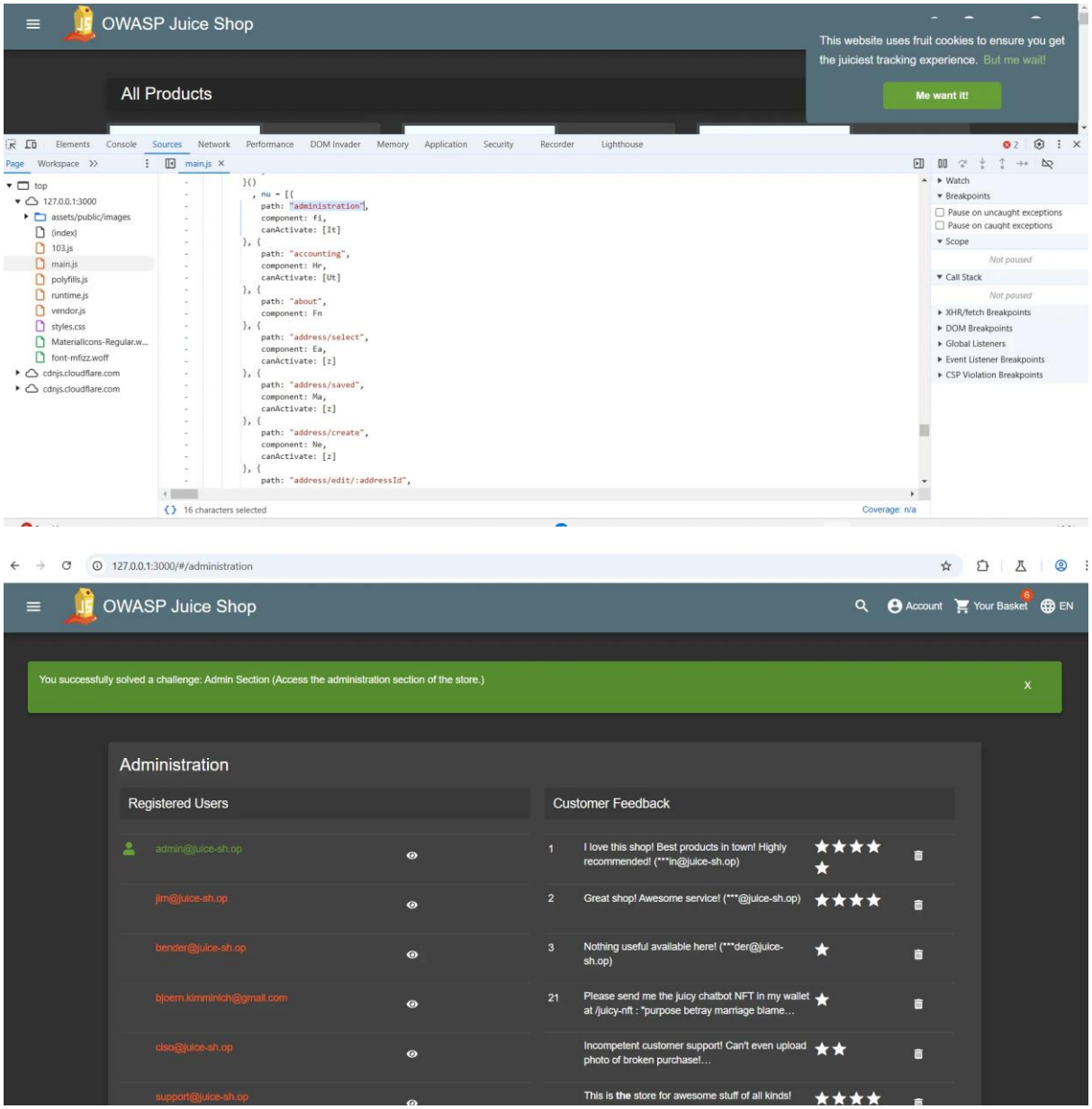
Upon successful access, I could view features intended exclusively for administrators, such as user management and audit logs. No privilege checks or tokens were required to validate the user's role, making it trivial for any authenticated user to elevate their privileges simply by modifying the URL.

**Key Execution Steps:**

- Logged in as a standard (non-admin) user.
- Attempted to access the admin page directly by visiting /administration.
- Found that the page loaded successfully without any access restrictions.
- Confirmed the presence of admin-only content and options available to a regular user.

This misconfiguration demonstrates a classic Broken Access Control issue, where failure to validate roles on the server side leads to unauthorized privilege escalation. Let me know if you'd like to add a browser-based screenshot flow or mitigation demo for this.






## Vulnerability #2: Security Misconfiguration in OWASP Juice Shop

This vulnerability was discovered while interacting with the Juice Shop's **/rest/products/ API endpoint**. During testing, I noticed that the API responses included unnecessary metadata, verbose error messages, and information that is typically useful for debugging during development but should not be exposed in a production environment. For instance, the response headers and JSON structures revealed internal object properties, server behaviors, and even framework-specific information that could assist an attacker in fingerprinting the system or crafting targeted exploits.

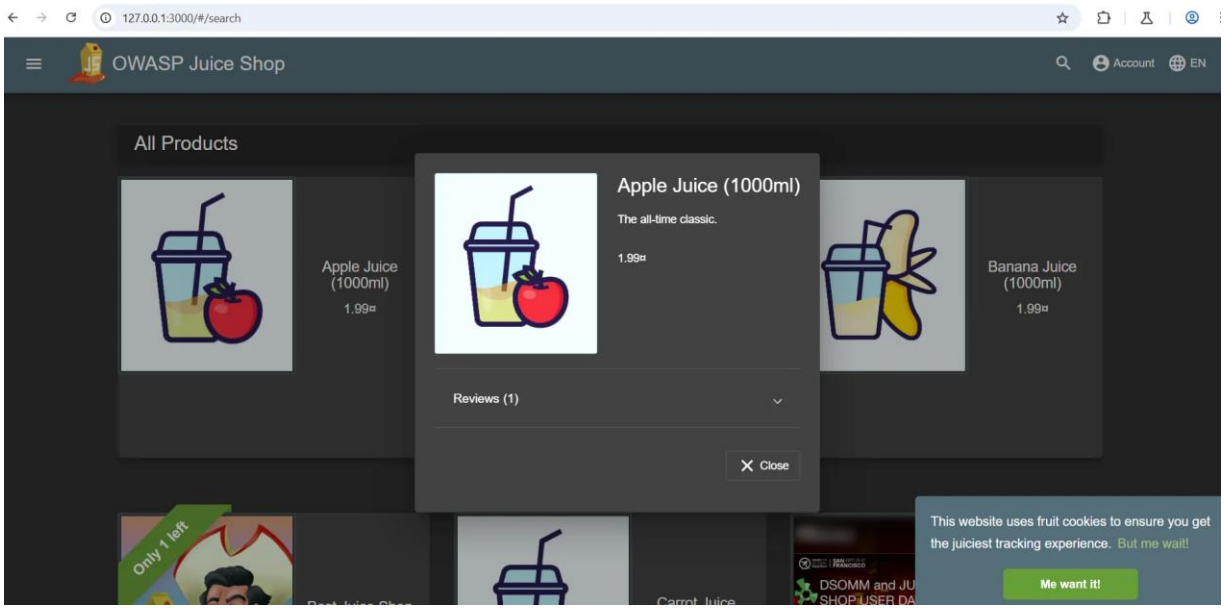
While analyzing network traffic via developer tools and intercepting requests with Burp Suite, I observed headers like **X-Powered-By: Express,** which explicitly disclosed the backend framework being used. Additionally, some endpoints returned full stack traces or internal error messages when supplied with malformed requests or invalid input. These details, while helpful for developers during testing, offer attackers insight into the internal workings of the application, including routes, libraries, and potential misconfigurations.

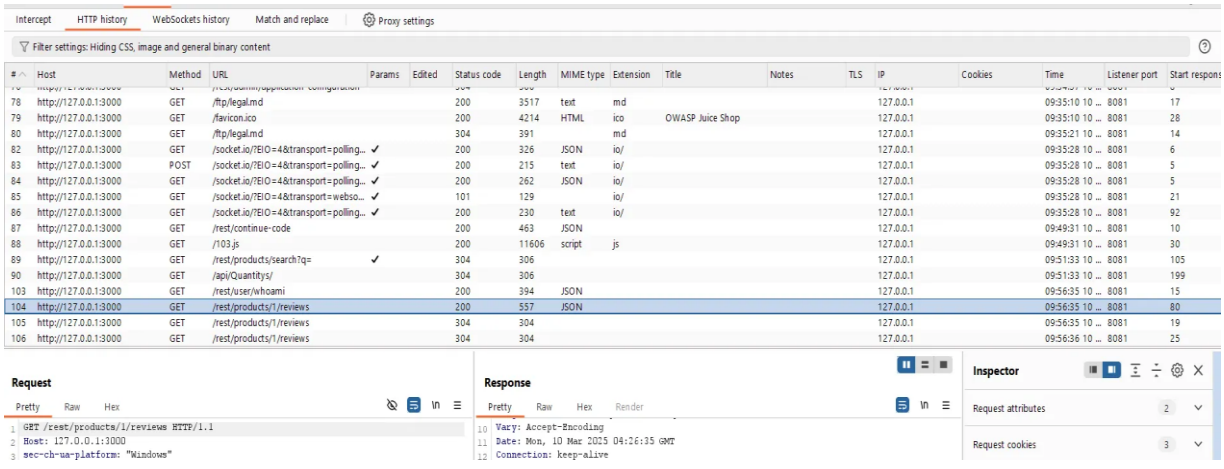
This kind of verbose output points to a lack of hardened production settings — a textbook case of Security Misconfiguration. It highlights that the application is likely running in debug mode or lacks proper exception handling, logging configurations, and security headers.

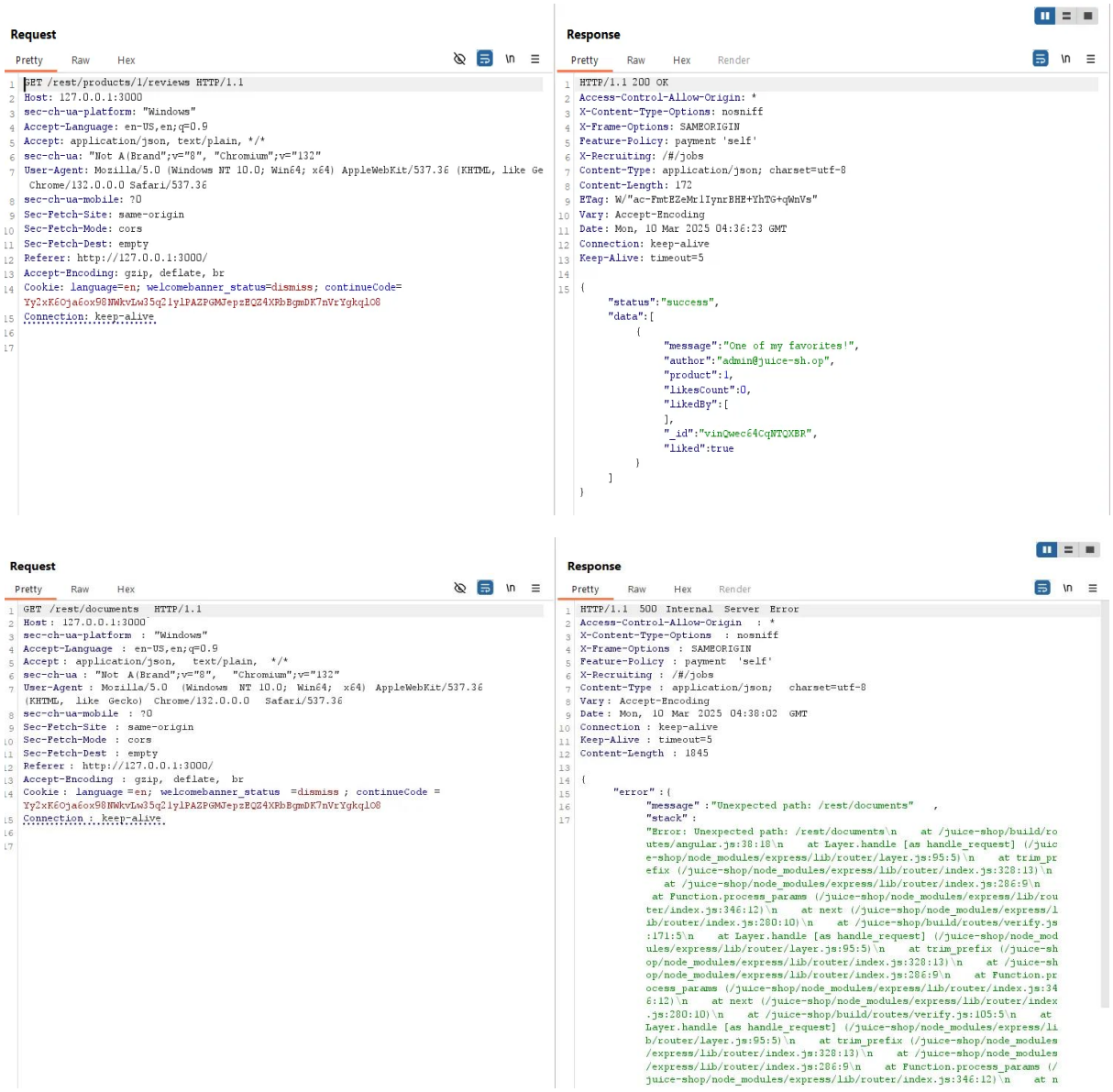
Key Execution Steps:

- Accessed the **/rest/products/** endpoint and observed verbose responses.
- Inspected response headers (e.g., **X-Powered-By**) disclosing framework details.
- Sent malformed input to test error handling and received detailed error messages.
- Confirmed exposure of internal structures, suggesting insecure configuration.

This vulnerability emphasizes the importance of disabling debug tools, hiding sensitive headers, and customizing error responses before moving an application to production.





**Request**

```
GET /rest/products/1/reviews HTTP/1.1
Host: 127.0.0.1:3000
sec-ch-ua-platform: "Windows"
Accept-Language: en-US,en;q=0.9
Accept: application/json, text/plain, */*
sec-ch-ua: "Not A(Brand";v="8", "Chromium";v="132"
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Ge
 Chrome/132.0.0.0 Safari/537.36
sec-ch-ua-mobile: ?0
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://127.0.0.1:3000/
Accept-Encoding: gzip, deflate, br
Cookie: language=en; welcomebanner_status=dismiss; continueCode=
Yy2xK6Oja6ox98NWkvLw35q21ylPAZPGMJepzEQZ4XRbBgmDK7nVrYgkqlO8
Connection: keep-alive
```

**Response**

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Feature-Policy: payment 'self'
X-Recruiting: /#/jobs
Content-Type: application/json; charset=utf-8
Content-Length: 172
ETag: W/"ac-FmtEZeMr1IynrBHE+YhTG+qWnVs"
Vary: Accept-Encoding
Date: Mon, 10 Mar 2025 04:36:23 GMT
Connection: keep-alive
Keep-Alive: timeout=5

{
    "status":"success",
    "data":[
        {
            "message":"One of my favorites!",
            "author":"admin@juice-sh.op",
            "product":1,
            "likesCount":0,
            "likedBy":[
            ],
            "_id":"vinQwec64CqNTQXBR",
            "liked":true
        }
    ]
}
```



**Request**

```
GET /rest/documents HTTP/1.1
Host : 127.0.0.1:3000
sec-ch-ua-platform : "Windows"
Accept-Language : en-US,en;q=0.9
Accept : application/json, text/plain, */*
sec-ch-ua : "Not A(Brand";v="8", "Chromium";v="132"
User-Agent : Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/132.0.0.0 Safari/537.36
sec-ch-ua-mobile : ?0
Sec-Fetch-Site : same-origin
Sec-Fetch-Mode : cors
Sec-Fetch-Dest : empty
Referer : http://127.0.0.1:3000/
Accept-Encoding : gzip, deflate, br
Cookie : language =en; welcomebanner_status =dismiss ; continueCode =
Yy2xK6Oja6ox98NWkvLw35q21ylPAZPGMJepzEQZ4XRbBgmDK7nVrYgkqlO8
Connection : keep-alive
```

**Response**

```
HTTP/1.1 500 Internal Server Error
Access-Control-Allow-Origin : *
X-Content-Type-Options : nosniff
X-Frame-Options : SAMEORIGIN
Feature-Policy : payment 'self'
X-Recruiting : /#/jobs
Content-Type : application/json; charset=utf-8
Vary : Accept-Encoding
Date : Mon, 10 Mar 2025 04:38:02 GMT
Connection : keep-alive
Keep-Alive : timeout=5
Content-Length : 1845

{
    "error" : {
        "message" : "Unexpected path: /rest/documents" ,
        "stack" :
        "Error: Unexpected path: /rest/documents\n    at /juice-shop/build/ro
        utes/angular.js:38:18\n    at Layer.handle [as handle_request] (/juic
        e-shop/node_modules/express/lib/router/layer.js:95:5)\n    at trim_pr
        efix (/juice-shop/node_modules/express/lib/router/index.js:328:13)\n
            at /juice-shop/node_modules/express/lib/router/index.js:286:9\n
         at Function.process_params (/juice-shop/node_modules/express/lib/rou
        ter/index.js:346:12)\n    at next (/juice-shop/node_modules/express/l
        ib/router/index.js:280:10)\n    at /juice-shop/build/routes/verify.js
        :171:5\n    at Layer.handle [as handle_request] (/juice-shop/node_mod
        ules/express/lib/router/layer.js:95:5)\n    at trim_prefix (/juice-sh
        op/node_modules/express/lib/router/index.js:328:13)\n    at /juice-sh
        op/node_modules/express/lib/router/index.js:286:9\n    at Function.pr
        ocess_params (/juice-shop/node_modules/express/lib/router/index.js:34
        6:12)\n    at next (/juice-shop/node_modules/express/lib/router/index
        .js:280:10)\n    at /juice-shop/build/routes/verify.js:105:5\n    at
        Layer.handle [as handle_request] (/juice-shop/node_modules/express/li
        b/router/layer.js:95:5)\n    at trim_prefix (/juice-shop/node_modules
        /express/lib/router/index.js:328:13)\n    at /juice-shop/node_modules
        /express/lib/router/index.js:286:9\n    at Function.process_params (/
        juice-shop/node_modules/express/lib/router/index.js:346:12)\n    at n
```

## Vulnerability #3: Sensitive Data Exposure in OWASP Juice Shop

This vulnerability was uncovered by navigating to the /ftp endpoint of the Juice Shop application, which is publicly accessible without authentication. Upon accessing this route, I discovered exposed files and directories that contained sensitive information such as user credentials, internal documentation, and plain-text data that should never be available to unauthenticated users. Some of the files were downloadable and included contents like backup logs, unencrypted passwords, and possibly customer or system data.

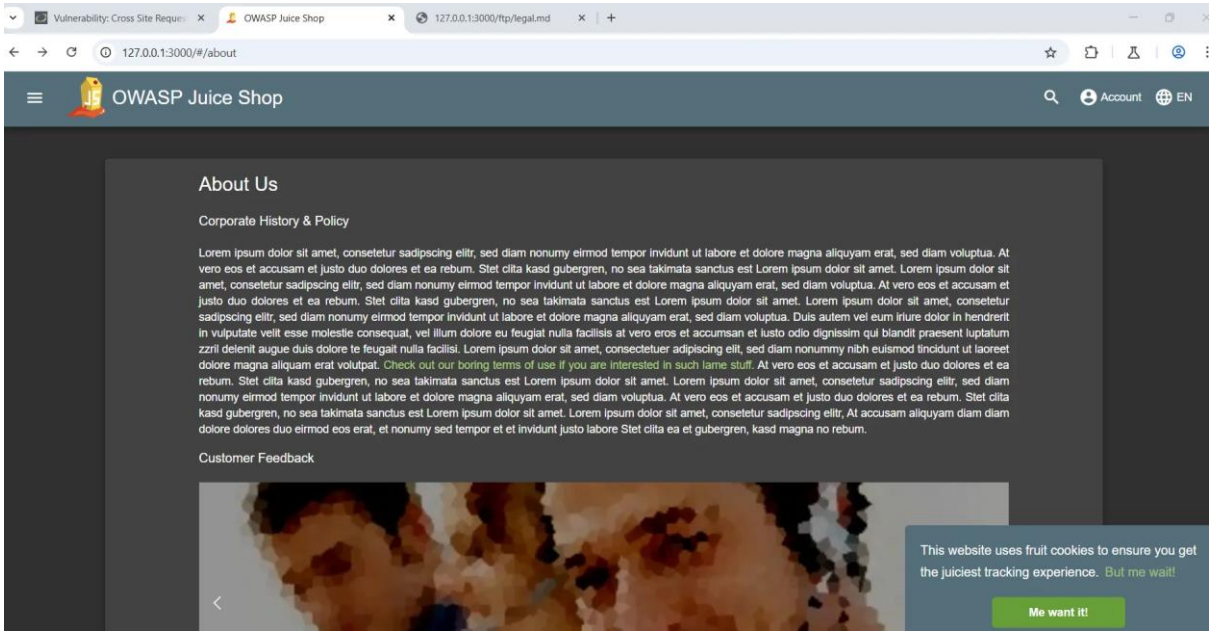
To confirm this exposure, I accessed various file paths directly via the browser and intercepted the traffic using Burp Suite. The absence of authentication or access control mechanisms on the /ftp route allowed unrestricted browsing and downloading of sensitive files. Furthermore, the communication was initially conducted over HTTP (in a test environment), making it possible for attackers on the same network to eavesdrop or perform man-in-the-middle (MitM) attacks and capture this sensitive data in transit.

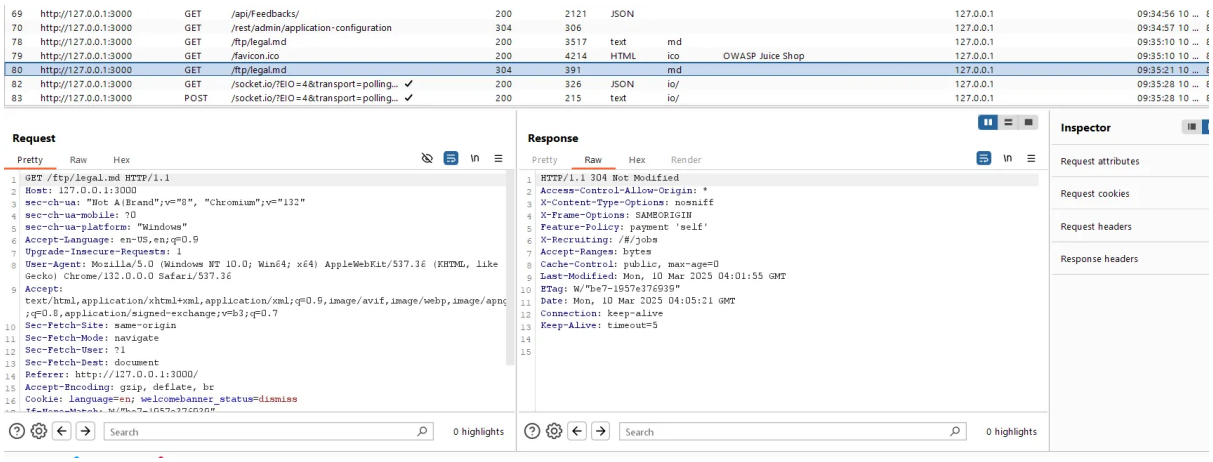
This vulnerability demonstrates how improper data storage and lack of access controls can directly lead to sensitive data exposure, especially when sensitive files are publicly hosted or left unprotected. The presence of such files on a live web server reflects insecure file management practices and violates basic principles of data confidentiality and integrity.

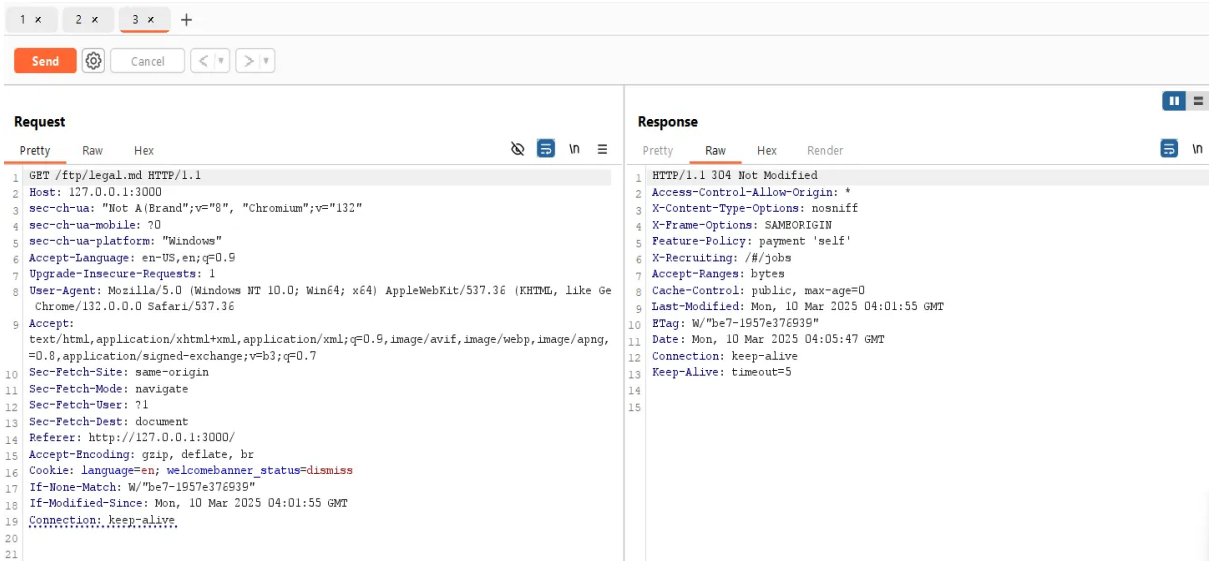
**Key Execution Steps:**

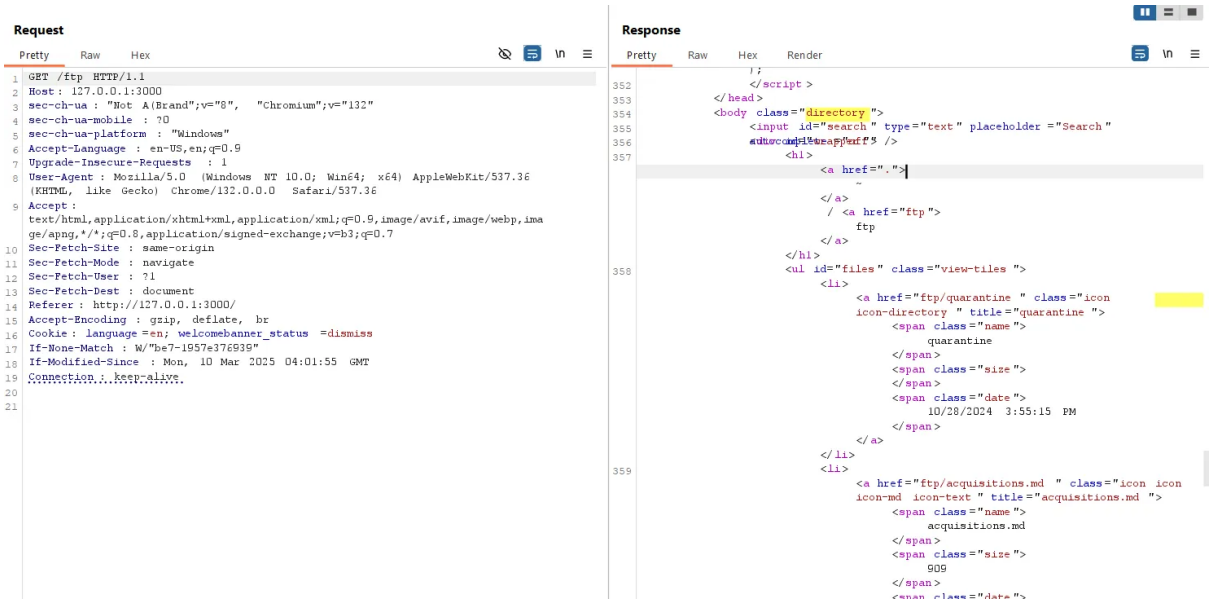
- Accessed /ftp endpoint in browser and found publicly visible files.
- Opened and downloaded exposed files containing potentially sensitive data.
- Verified lack of authentication and encryption over the connection.
- Intercepted traffic via Burp Suite and confirmed plain-text transmission.

This issue highlights the critical need for secure file handling, proper access control mechanisms, and encryption protocols in any production-grade web application.




1 × 2 × 3 × +
Send
Cancel
Request
Pretty Raw Hex
GET /ftp/legal.md HTTP/1.1
Host: 127.0.0.1:3000
sec-ch-ua: "Not A(Brand";v="8", "Chromium";v="132"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Accept-Language: en-US,en;q=0.9
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Ge
Chrome/132.0.0.0 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,
=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://127.0.0.1:3000/
Accept-Encoding: gzip, deflate, br
Cookie: language=en; welcomebanner_status=dismiss
If-None-Match: W/"be7-1957e376939"
If-Modified-Since: Mon, 10 Mar 2025 04:01:55 GMT
Connection: keep-alive
Response
Pretty Raw Hex Render
HTTP/1.1 304 Not Modified
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Feature-Policy: payment 'self'
X-Recruiting: /#/jobs
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Last-Modified: Mon, 10 Mar 2025 04:01:55 GMT
ETag: W/"be7-1957e376939"
Date: Mon, 10 Mar 2025 04:05:47 GMT
Connection: keep-alive
Keep-Alive: timeout=5


Request
Pretty Raw Hex
GET /ftp HTTP/1.1
Host: 127.0.0.1:3000
sec-ch-ua : "Not A(Brand";v="8", "Chromium";v="132"
sec-ch-ua-mobile : ?0
sec-ch-ua-platform : "Windows"
Accept-Language : en-US,en;q=0.9
Upgrade-Insecure-Requests : 1
User-Agent : Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/132.0.0.0 Safari/537.36
Accept :
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,ima
ge/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site : same-origin
Sec-Fetch-Mode : navigate
Sec-Fetch-User : ?1
Sec-Fetch-Dest : document
Referer : http://127.0.0.1:3000/
Accept-Encoding : gzip, deflate, br
Cookie : language =en; welcomebanner_status =dismiss
If-None-Match : W/"be7-1957e376939"
If-Modified-Since : Mon, 10 Mar 2025 04:01:55 GMT
Connection : keep-alive
Response
Pretty Raw Hex Render
</script>
</head>
<body class="directory ">
<input id="search" type="text" placeholder="Search"
<h1>
<a href="."
/
</a>
/ <a href="ftp">
ftp
</a>
</h1>
<ul id="files" class="view-tiles ">
<li>
<a href="ftp/quarantine " class="icon
icon-directory " title="quarantine ">

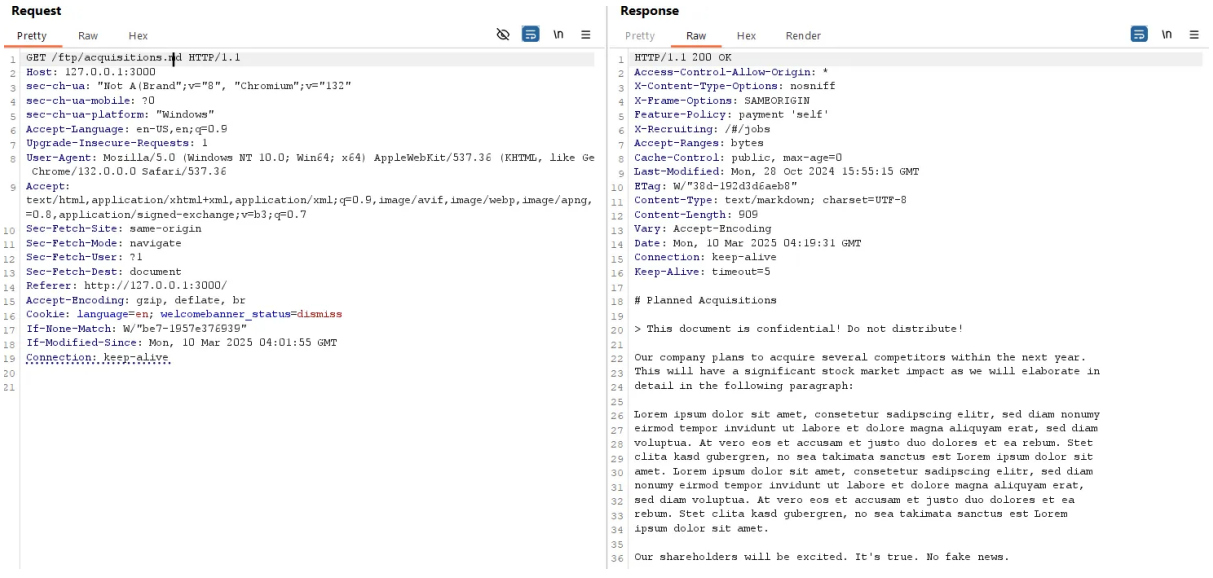
quarantine

10/28/2024 3:55:15 PM

</a>
</li>
<li>
<a href="ftp/acquisitions.md " class="icon icon
icon-md icon-text " title="acquisitions.md ">

acquisitions.md

909


Request
Pretty Raw Hex
GET /ftp/acquisitions.md HTTP/1.1
Host: 127.0.0.1:3000
sec-ch-ua: "Not A(Brand";v="8", "Chromium";v="132"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Accept-Language: en-US,en;q=0.9
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Ge
Chrome/132.0.0.0 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,
=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://127.0.0.1:3000/
Accept-Encoding: gzip, deflate, br
Cookie: language=en; welcomebanner_status=dismiss
If-None-Match: W/"be7-1957e376939"
If-Modified-Since: Mon, 10 Mar 2025 04:01:55 GMT
Connection: keep-alive
Response
Pretty Raw Hex Render
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Feature-Policy: payment 'self'
X-Recruiting: /#/jobs
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Last-Modified: Mon, 28 Oct 2024 15:55:15 GMT
ETag: W/"38d-192d3d6aeb8"
Content-Type: text/markdown; charset=UTF-8
Content-Length: 909
Vary: Accept-Encoding
Date: Mon, 10 Mar 2025 04:19:31 GMT
Connection: keep-alive
Keep-Alive: timeout=5
# Planned Acquisitions
> This document is confidential! Do not distribute!
Our company plans to acquire several competitors within the next year.
This will have a significant stock market impact as we will elaborate in
detail in the following paragraph:
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy
eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam
voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet
clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit
amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam
nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat,
sed diam voluptua. At vero eos et accusam et justo duo dolores et ea
rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem
ipsum dolor sit amet.
Our shareholders will be excited. It's true. No fake news.

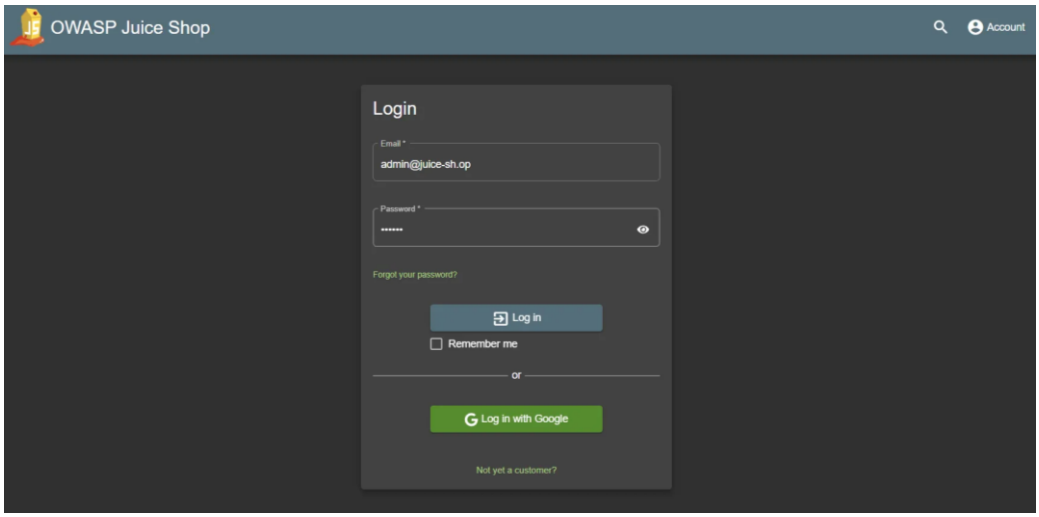
# Vulnerability #4: Broken Authentication in OWASP Juice Shop

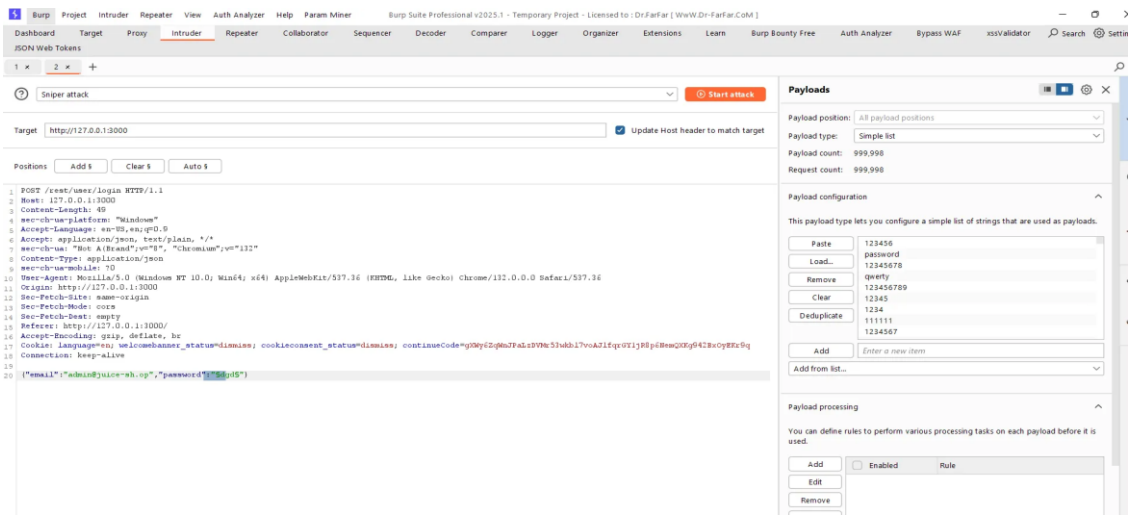
The broken authentication vulnerability was identified by targeting the **/login** endpoint of OWASP Juice Shop. The login mechanism fails to implement sufficient protection against automated login attempts, making it susceptible to brute force attacks. During testing, I crafted a simple script using Burp Suite Intruder to send multiple login attempts with different username-password combinations. There were no rate-limiting mechanisms, **CAPTCHA** challenges, or account lockout triggers even after dozens of failed attempts.

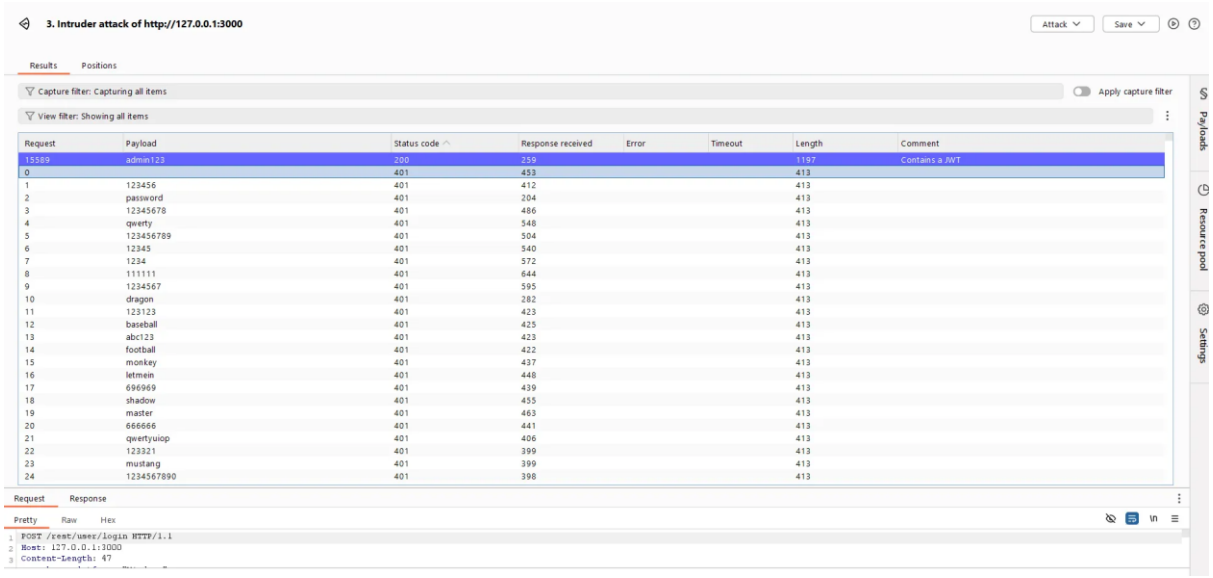
Using a basic password list and a known email address (obtained from the application itself or through enumeration), I was able to repeatedly send login requests without any sign of blocking or alerting. Eventually, a correct credential pair was found, which allowed access to a legitimate user account. This behavior confirmed that the authentication mechanism lacked essential security controls.

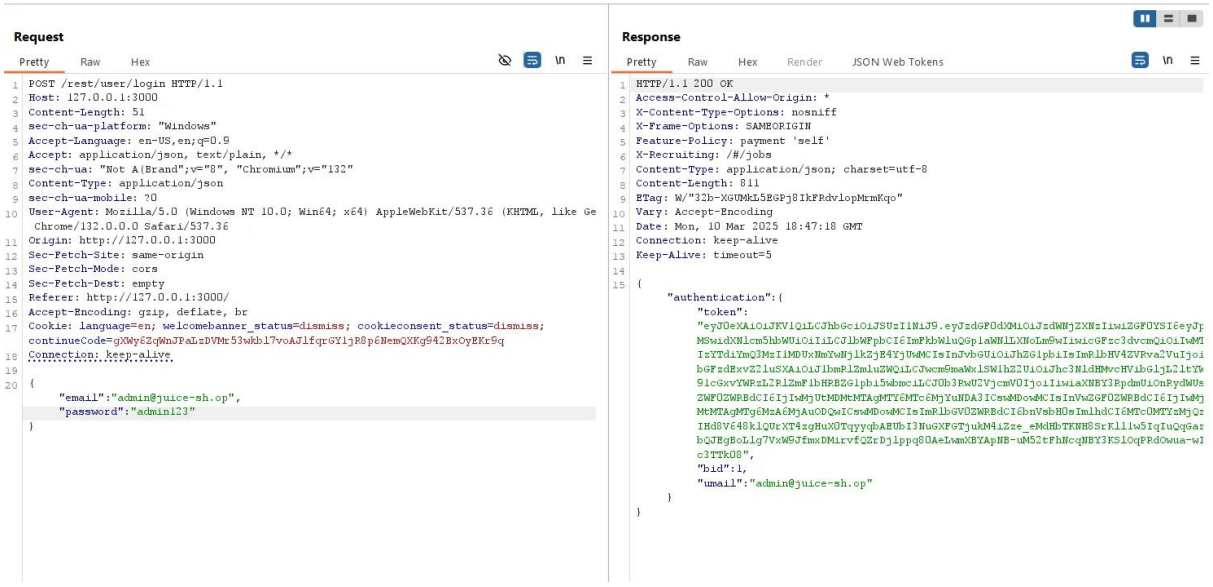
**Key Execution Steps:**

- Accessed the **/login** endpoint in OWASP Juice Shop.
- Intercepted login request with Burp Suite to analyze the request structure.
- Used Burp Suite Intruder to automate login attempts with a wordlist.
- Observed that no CAPTCHA, rate limiting, or account lockout was in place.
- Successfully logged in with a valid credential after multiple attempts.






## Vulnerability #5: CAPTCHA Bypass in OWASP Juice Shop

The **CAPTCHA** bypass vulnerability was discovered on the /contact page of OWASP Juice Shop, which features a feedback submission form protected by a simple CAPTCHA challenge. This CAPTCHA is intended to prevent automated or bot-driven submissions. However, upon inspecting the request structure through Burp Suite, it was clear that the CAPTCHA validation was implemented weakly on the client side and could be bypassed programmatically.

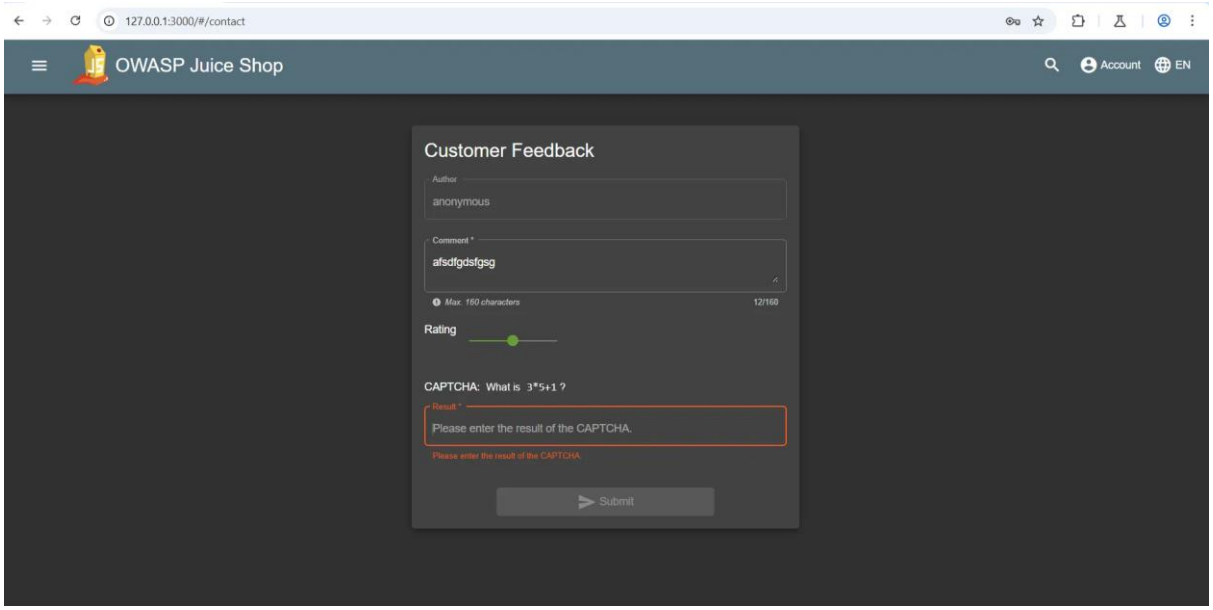
To exploit this, I captured the POST request generated upon submitting the form and noticed that the CAPTCHA answer was included directly in the request payload as a visible parameter. By replaying this request with automated tools like Burp Suite Repeater or scripting it using Python with the requests library, I was able to submit multiple forms without solving the CAPTCHA manually. The server did not perform additional backend verification, meaning the CAPTCHA could be ignored or brute-forced easily.

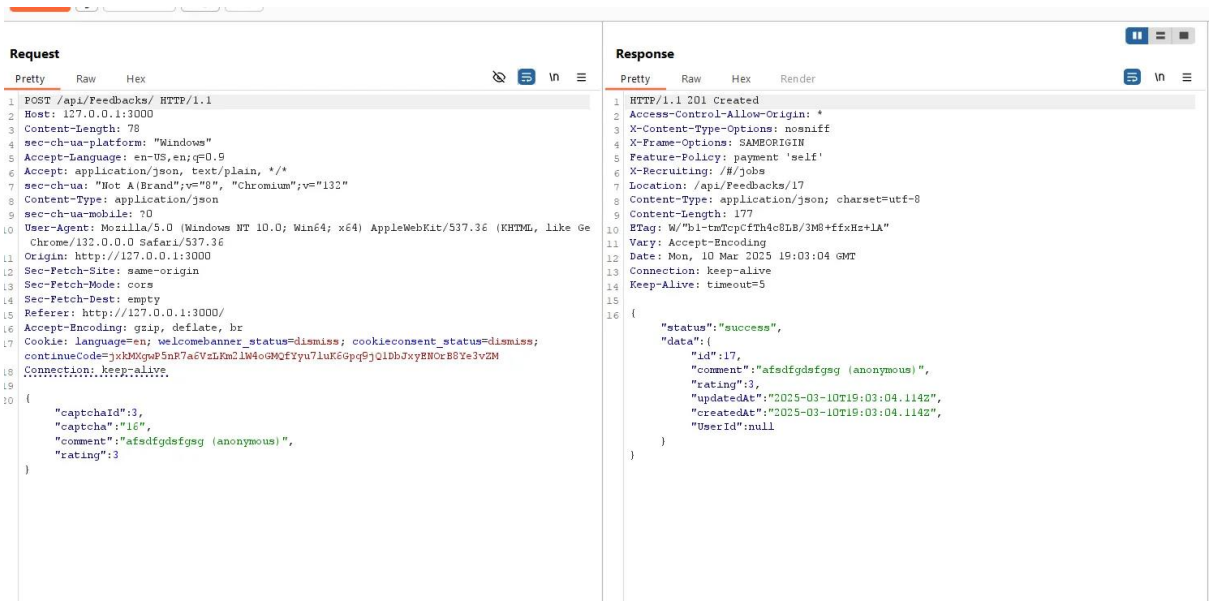
This vulnerability significantly weakens the defense against bots and spamming. Attackers can script large-scale automated feedback submissions, phishing payloads, or denial-of-service attempts by flooding the system with junk messages — all while evading CAPTCHA checks.

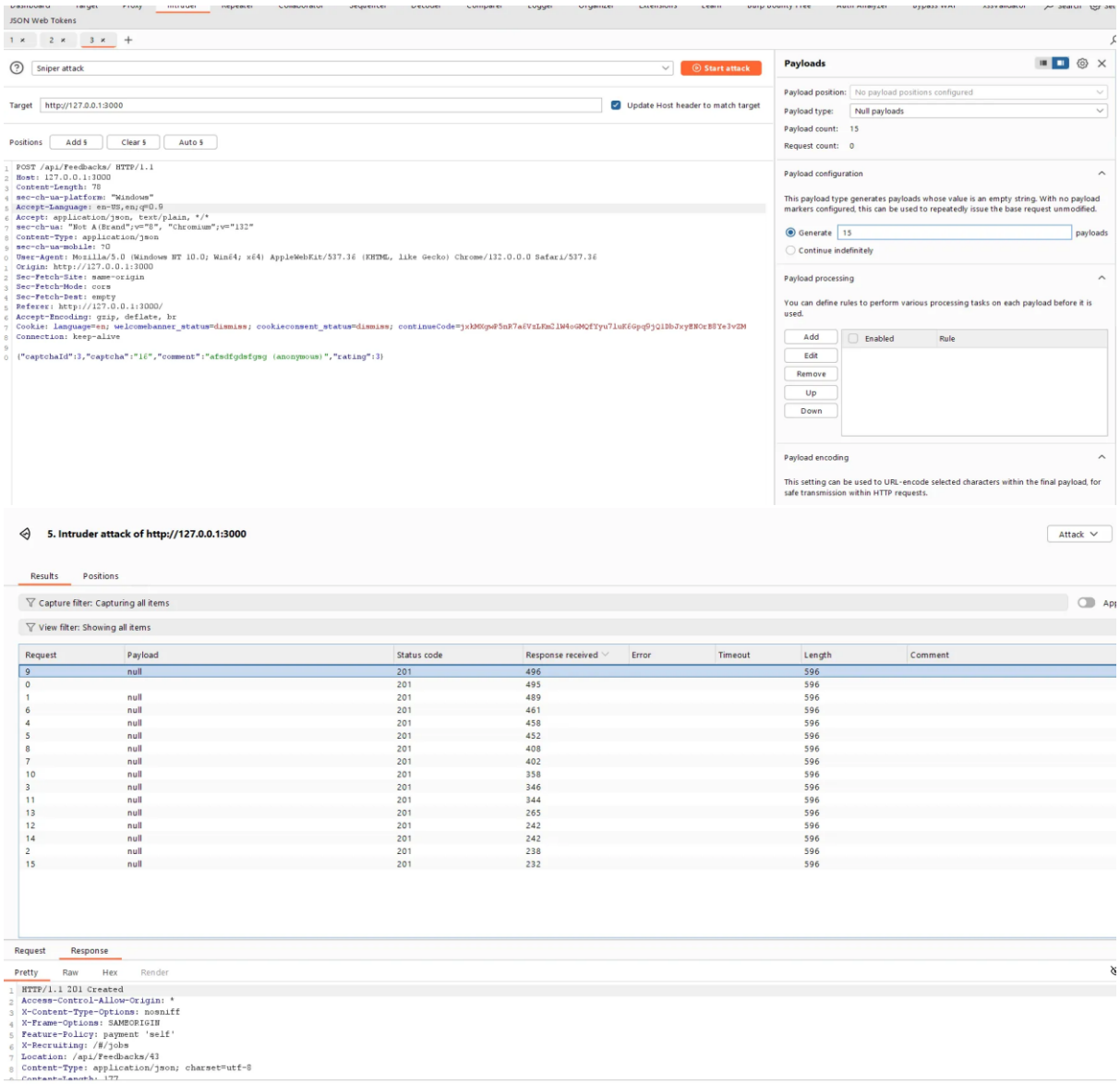
Key Execution Steps:

- Navigated to the **/contact** feedback form.
- Observed CAPTCHA parameter in the request payload via Burp Suite.
- Captured and replayed the request using Burp Repeater without solving CAPTCHA.
- Automated the request with a custom script to submit feedback repeatedly.
- Confirmed that CAPTCHA challenge was bypassed and not validated server-side.

This shows the implementation of CAPTCHA is superficial and ineffective against automation, which could allow malicious actors to abuse the system functionality at scale.

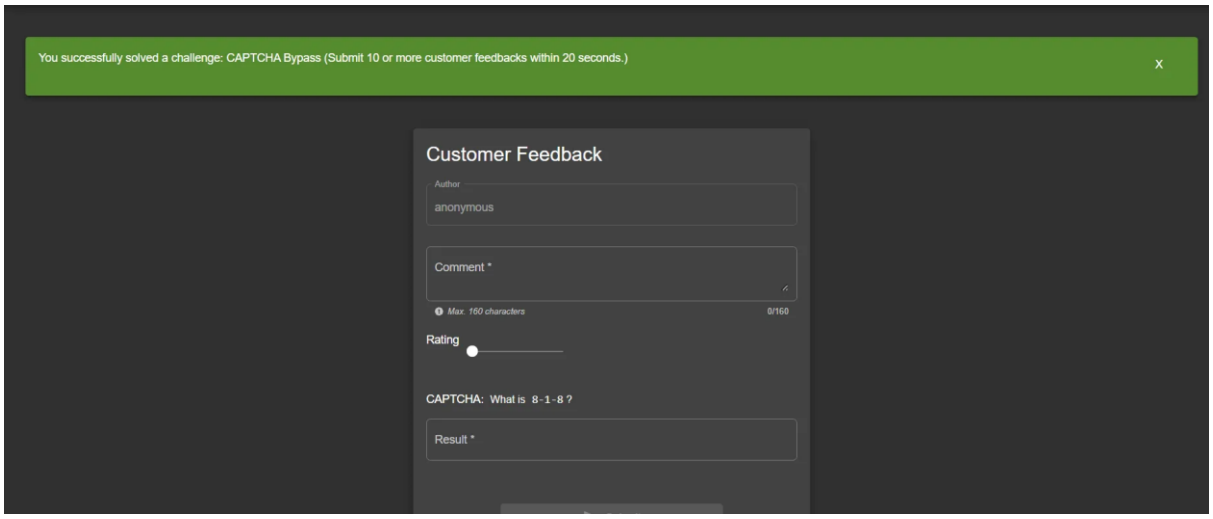











**Pratham Verma**

# Risk Analysis

Risk analysis involves systematically evaluating the likelihood of a vulnerability being discovered and exploited, along with the magnitude of harm it can cause to an organization. In this assessment, we considered multiple dimensions such as technical exploitability, ease of detection, attacker motivation, exposure to the public, and the value of impacted assets.

- **High-risk vulnerabilities** like SQL Injection and Broken Access Control are particularly alarming due to their ease of exploitation and potential for severe damage. These flaws can be weaponized by attackers to perform unauthorized operations, extract sensitive data, or even take full control over affected systems.
- **Medium-risk vulnerabilities** such as Stored and Reflected XSS often require some form of user interaction or social engineering, reducing their probability of mass exploitation. However, when combined with phishing campaigns or other vulnerabilities, their risk escalates.
- **Low to medium-risk issues** like CAPTCHA Bypass or weak CSRF protections may not cause immediate harm but can lead to larger compromises when chained with more critical flaws.

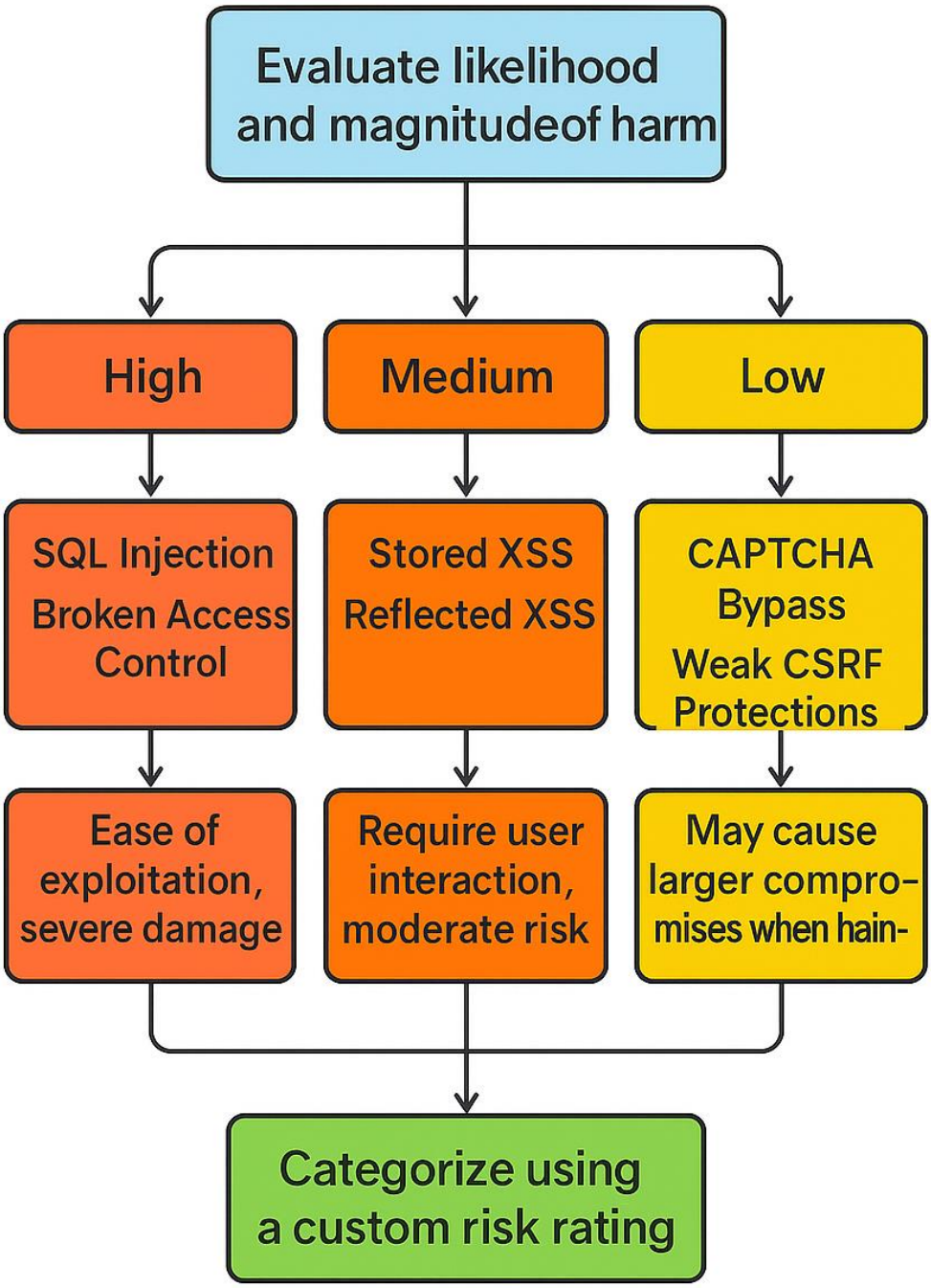
Each vulnerability was carefully categorized using a custom risk rating that balances CVSS scores, real-world impact, application context, and business relevance. Additional factors such as frequency of exploitation in the wild, likelihood of discovery by threat actors, and complexity of attack execution were also incorporated.

Moreover, risk analysis also considered the degree of exposure — such as whether the vulnerable component is internet-facing — and how the flaw may contribute to lateral movement or privilege escalation.

This comprehensive approach to risk profiling enables informed decision-making for vulnerability management. It helps security teams establish a risk-based prioritization strategy, align remediation tasks with business objectives, and allocate resources where they are most needed.

Ultimately, this section emphasizes that even seemingly low-severity vulnerabilities can have a cascading impact, particularly when combined with other weaknesses, thus reinforcing the need for a defense-in-depth approach.

# Risk Analysis Diagram




Pratham Verma

# Impact Analysis

Impact analysis focuses on understanding the consequences of successful exploitation of identified vulnerabilities. It considers both the direct and indirect effects on confidentiality, integrity, availability, business continuity, legal obligations, and brand reputation.

- For **high-risk vulnerabilities** like SQL Injection or Broken Authentication, the potential impact includes complete compromise of backend databases, unauthorized access to user accounts, exposure of Personally Identifiable Information (PII), and manipulation or deletion of critical data. These attacks can not only lead to immediate financial losses but also invite legal action under data protection regulations.
- **Medium-risk vulnerabilities** such as Cross-Site Scripting (XSS) may not directly grant system access, but their ability to execute scripts in a victim's browser can be devastating. XSS can facilitate session hijacking, drive-by downloads, or redirect users to malicious websites. In cases of Stored XSS, the persistent nature of the attack can repeatedly affect every user who views the page.
- **Lower severity vulnerabilities** like debug exposure or CAPTCHA bypass might seem benign at first but often assist attackers in gathering intelligence, testing attack feasibility, or automating further exploitation. Such footholds, once established, can significantly raise the threat level of an application.

This analysis also accounts for the ripple effects of exploitation. For instance, a compromised web application may lead to downstream impacts such as data leakage, loss of customer trust, operational downtime, or regulatory penalties. In regulated industries, this could further lead to audits, fines, or reputational fallout.

Understanding the impact also supports cost-benefit analysis for remediation efforts. By mapping vulnerabilities to business-critical functions, organizations can assess which threats pose the greatest risk to operations and reputation. This ensures not just technical resilience, but strategic and operational preparedness in the face of evolving cyber threats.

# Real World Thinking

Real-world thinking bridges the gap between theoretical vulnerability assessments and the actual consequences organizations face during live cyberattacks. While tools like DVWA and OWASP Juice Shop provide controlled environments for identifying vulnerabilities, attackers in the real world exploit the same types of weaknesses in production applications, often with devastating outcomes.

For example, SQL Injection vulnerabilities like those observed in DVWA have been exploited in major breaches such as the Heartland Payment Systems and TalkTalk incidents, where attackers exfiltrated millions of user records by manipulating poorly secured database queries. These real cases validate the critical nature of even basic injection flaws and highlight the importance of server-side validation and parameterized queries.

Similarly, Cross-Site Scripting (XSS), often viewed as a medium-risk issue in lab environments, has been used in phishing campaigns, browser-based crypto miners, and session hijacking attacks. Persistent (Stored) XSS can act as a launchpad for full account takeover, especially when combined with insecure session handling.

Security misconfigurations, as found in OWASP Juice Shop, are another frequent real-world issue. Misconfigured S3 buckets, exposed admin panels, and forgotten debugging interfaces have led to serious data breaches and public embarrassment for companies across various sectors.

Broken Access Control remains one of the most common issues in real-world applications. Attackers often crawl through application endpoints or use forceful browsing to reach unauthorized resources—something made even easier when developers rely solely on client-side access restrictions without enforcing roles server-side.

The concept of **chained vulnerabilities** is particularly relevant in practical scenarios. In a real-world attack, an adversary may first identify a low-severity flaw like a CAPTCHA bypass, then use it to automate brute-force login attempts (Broken Authentication), and finally exploit a SQL Injection vulnerability to extract sensitive data. This layered exploitation approach mirrors real attacker behavior and emphasizes why every vulnerability—regardless of its standalone severity—must be addressed.

From a defensive perspective, real-world thinking also stresses the importance of proactive security measures such as threat modeling, security automation in CI/CD pipelines, regular penetration testing, and security awareness training for developers and employees.

Ultimately, applying real-world thinking allows organizations to go beyond checkbox compliance and adopt a security-first mindset that anticipates attacker behavior, strengthens layered defenses, and aligns security initiatives with business priorities.

**Pratham Verma**

# Recommendations and Mitigations

The following section outlines practical, actionable steps to mitigate the vulnerabilities discovered during the assessment of DVWA and OWASP Juice Shop. The recommendations are grouped into technical, procedural, and strategic controls to support both immediate remediation and long-term resilience:

**1. Input Validation & Output Encoding**

- Enforce strict input validation on all user-supplied data using allowlists.
- Sanitize inputs at both client and server-side to prevent injection attacks.
- Encode all dynamic output (HTML, JavaScript, URL) to neutralize XSS vectors.

**2. Use of Prepared Statements and ORM**

- For SQL Injection prevention, adopt parameterized queries and ORM frameworks that abstract raw SQL queries.
- Avoid string concatenation when building database commands.

**3. Strong Authentication Controls**

- Enforce multi-factor authentication (MFA) for both users and admin accounts.
- Implement CAPTCHA on login, registration, and sensitive action forms to prevent brute force attacks.
- Lock accounts temporarily after multiple failed login attempts.

**4. Access Control Mechanisms**

- Apply server-side Role-Based Access Control (RBAC) for each endpoint.
- Never rely on client-side access enforcement such as hidden elements or JavaScript redirects.
- Regularly audit authorization rules to avoid privilege escalation.

**5. Security Misconfiguration Hardening**

- Disable debug mode and error stack traces in production.
- Use security headers such as Content-Security-Policy, X-Content-Type-Options, X-Frame-Options, etc.
- Remove unnecessary services, directories, and verbose error messages.

**6. Secure Data Handling**

- Encrypt sensitive data at rest using industry-standard algorithms (AES-256).
- Enforce TLS (HTTPS) for all data in transit with modern cipher suites.
- Regularly rotate encryption keys and securely store secrets using vaults.

**7. CSRF Protection**

- Use synchronizer tokens or double-submit cookies for all state-changing requests.
- Set SameSite=strict or lax attributes for cookies to prevent cross-origin misuse.

**8. Secure Software Development Lifecycle (SSDLC)**

- Integrate Static and Dynamic Application Security Testing (SAST/DAST) tools in CI/CD pipelines.
- Conduct regular code reviews, penetration testing, and threat modeling sessions.
- Train developers on secure coding practices and the OWASP Top 10.

**9. Monitoring and Logging**

- Deploy a centralized logging system and SIEM to detect suspicious activity.
- Log access attempts, authentication events, and anomaly patterns.
- Enable real-time alerting for high-risk events (e.g., SQL errors, admin access).

**10. Periodic Security Audits**

- Schedule regular vulnerability assessments and red team simulations.
- Perform patch management to fix known CVEs and outdated components.
- Engage third-party security experts for unbiased evaluation.

By applying a layered security model — known as Defense in Depth — these recommendations will collectively reduce the attack surface, improve visibility, and prevent exploitation of both common and advanced vulnerabilities.

# Conclusion

The comprehensive security assessment of DVWA and OWASP Juice Shop has shed light on a variety of vulnerabilities that mirror those frequently encountered in real-world web applications. This exercise demonstrates how the lack of secure development practices, misconfigurations, and insufficient validation mechanisms can lead to critical security breaches. Vulnerabilities such as SQL Injection, Broken Authentication, and Broken Access Control represent high-impact threats that, if exploited, can grant unauthorized access, compromise sensitive data, or even lead to complete system takeover. These findings reinforce the need for organizations to adopt a proactive, rather than reactive, approach to application security.

Importantly, this evaluation also highlighted medium and low-risk issues—such as Cross-Site Scripting (XSS), CAPTCHA bypass, and misconfigured security headers—that are often dismissed due to their perceived low severity. However, when chained with other flaws, they can significantly increase the overall risk exposure. Attackers are adept at leveraging multiple small missteps to orchestrate full-scale compromises. Thus, even seemingly minor vulnerabilities deserve immediate attention and mitigation. The interconnected nature of web technologies means that a holistic understanding of application behavior, data flow, and trust boundaries is critical for effective security planning.

This report also emphasizes that security is not a one-time audit or a static checklist item. Rather, it is a continuous process that must be embedded into every phase of the Software Development Life Cycle (SDLC). Regular code reviews, automated security scanning, manual penetration testing, and ongoing training for developers and administrators must become standard practices. Beyond technical measures, fostering a security-first culture across development, operations, and management teams is essential to building resilient systems.

In closing, the vulnerabilities identified and discussed in this report should serve as a wake-up call and an opportunity for improvement. They highlight both the technical and organizational changes necessary to protect modern web applications. By treating security as a shared responsibility, integrating best practices, and continuously adapting to new threats, organizations can better safeguard their digital infrastructure, maintain user trust, and ensure long-term business continuity.