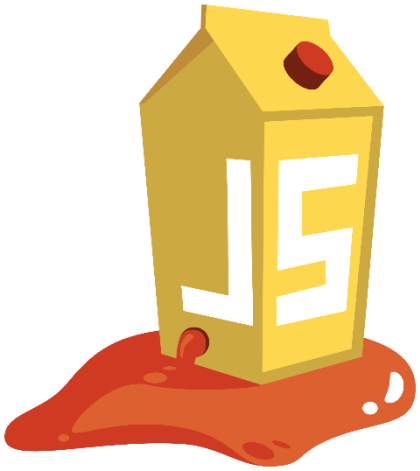
Web Application Penetration Testing Report

(OWASP JUICE SHOP)



Presented by:

- Mark Mekhael Adly
- Hassan Amgad Hassan
- Shahd Ahmed Mohamad
- Mohamad Mahmoud Ali

Table of Contents:

1. Executive Overview

- The objective of this penetration test was to evaluate the security of the Juice Shop application. The assessment aimed to identify weaknesses within the web platform, including user-facing components and backend infrastructure. This engagement focused on simulating real-world attack scenarios to measure the application's resilience against common threats.
    - **Tested Application**: Juice Shop
    - **Testing Mode**: Black-box
    - **Assessment Focus**: Web application-level vulnerabilities
    - **Approach**: Aligned with OWASP Top 10 and industry best practices

---

2. Summary of Findings:

- CSRF (Cross-Site Request Forgery)
- Broken Access Control
- IDOR (Insecure Direct Object Reference)
- Broken Authentication via Predictable OAuth State
- SQL Injection in the Login Page
- Authentication Bypass via SQL Injection
- JWT Token Disclosure
- SQLi to Bypass Login for Another User
- Login via SQLi With Non-Existent Account
- Backup File Discovery (Gobuster)
- Weak Encryption in Coupons
- Cookies Missing HttpOnly Flag

Key Security Recommendations:

- Implement anti-CSRF tokens
- Fix access control logic
- Protect object references
- Secure OAuth flows
- Prevent SQL Injection
- Secure JWT tokens
- Remove exposed backup files
- Use strong encryption
- Set HttpOnly on cookies

---

3. Risk Rating:

SECURITY LEVEL	TOTAL VULNS	EXAMPLE FINDINGS
CRITICAL	3	Broken Access Control, Broken Authentication, SQL Injection
HIGH	6	XSS, IDOR, etc.
MEDIUM	3	CORS, CSRF, etc.

3.1. Finding Overview

Ref .ID	Description	Risk	Check Status
5.1	Fake Account Creation via UNION SQL Injection	Critical	Passed
5.2	Authentication Bypass via SQL Injection	Critical	Passed
5.3	JWT Decoding Revealing Password	Critical	Passed
5.4	Bypass Login for Another User	High	Passed
5.5	Logging In with a Non-Existent Account	High	Passed
5.7	Forging Coupon Codes via Encoded Values	High	Passed
6	Broken Access Control – Unauthorized Access to Admin Panel	High	Passed
7	Accessing Another User's Basket - (IDOR)	High	Passed
8	Stored XSS in Feedback Form	High	Passed
5.6	Discovery of Backup Files (e.g., .bak via Gobuster)	Medium	Passed
9	Cross-Origin Resource Sharing (CORS) Misconfiguration	Medium	Passed
10	CSRF in /profile Endpoint	Medium	Passed

4. Testing Strategy

- A methodical process based on OWASP Top 10 2023 and modern attack methodologies guided the penetration test. Both automated tools and manual analysis were applied.
    - **Scope & Method:**
        - **Scope**: Comprehensive web application assessment including backend components
        - **Methodology**: Risk-based and OWASP-aligned testing

Tools Utilized:

- Vulnerability Scanning: *Acunetix* , *Nuclei*
- **Discovery Tools**: *subfinder* for subdomain enumeration
- **Directory Brute Forcing**: *gobuster*, *Acunetix*
- **Wordlists**: *SecLists*
- **Vulnerability Detection**: *Burp Suite Pro*
- **Exploitation Tools**: *Custom scripts*, *SQLmap*

Testing Phases:

- **Reconnaissance:**
    - Enumerated subdomains and directories to map the structure.
    - Vulnerability Scanning
    - Identified an open admin interface on port 8000.
- **Scanning:**
    - Used automated and manual tools to identify issues like XSS, SQLi, IDOR, and authentication flaws.

- **Exploitation:**
    - Demonstrated data extraction via SQLi.
    - Performed JavaScript injection to show session hijacking.
    - Proved CSRF attacks could be executed to manipulate user actions.
    - Accessed admin functions using default credentials.
- **Post-Exploitation:**
    - Extracted sensitive information including user credentials.
    - Hijacked sessions and demonstrated account takeover.
    - Used administrative privileges to manipulate backend systems.
- **Validation & Reporting:**
    - Each vulnerability was validated manually.
    - Provided visual evidence and reproduction steps.
    - Tailored remediation steps were proposed for each issue.

4.1. Reconnaissance:

Target: *demo.owasp-juice.shop*

1. **Introduction**
   - This report summarizes the reconnaissance findings for the **OWASP Juice Shop** (*demo.owasp-juice.shop*), an intentionally vulnerable web application designed for security training. The goal was to gather information about the target's infrastructure, subdomains, and exposed services.

2. **Methodology**
   - The reconnaissance phase included:
     - **Subdomain Enumeration**
     - **Directory and File Enumeration**
   - Tools used:
     - Subfinder (for all domains)
     - Nuclei (for SSL/TLS and DNS checks)
     - **Gobuster** (for directory brute-forcing)

3. **Finidngs**

   - **Directory and File Enumeration**

Path	Status	Notes
/assets	301	Static resources
/ftp	200	discovered via Gobuster, contained **.bak** files like ***coupons_2013.md.bak*** and ***package.json.bak***
/metrics	200	Prometheus metrics (unauthenticated)
/api	500	Internal server error
/Profile	500	Possible improper access control
/snippets	200	
/login	200	vulnerable to SQL Injection
/administration	200	ccessible without proper access controls

5. Vulnerability Scanning & Exploitation

5. SQL Injection:

5.1. Fake Account Creation via UNION SQL Injection

Critical

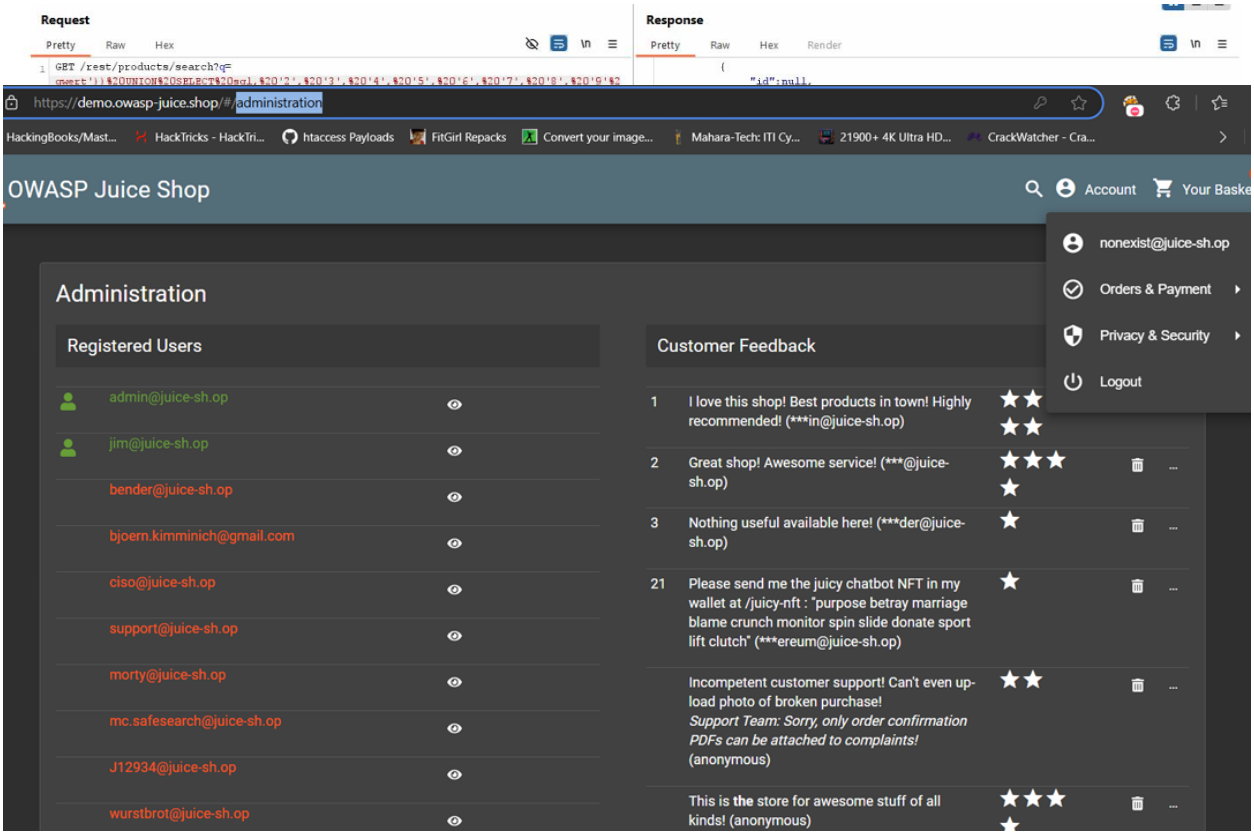
Description:

- Injected single quote ' in email field and captured the request using Burp Suite.
- Error message indicated backend is using SQLite.
- Performed the following payload via search functionality:

  banana')) UNION SELECT sql, '2', '3', '4', '5', '6', '7', '8', '9' FROM sqlite_master—
- Retrieved the database schema.
- Crafted and submitted the following UNION SELECT payload to create a fake user:

  ' UNION SELECT * FROM □SELECT 15 AS 'id', '' AS 'username', 'nonexist@juice-sh.op' AS 'email', '12345' AS 'password', 'admin' AS 'role', '123' AS 'deluxeToken', '1.2.3.4' AS 'lastLoginIp', '/assets/public/images/uploads/default.svg' AS 'profileImage', '' AS 'totpSecret', 1 AS 'isActive', '1999□08□16 14□14□41.644 □00□00' AS 'createdAt', '1999□08□16 14□33□41.930 □00□00' AS 'updatedAt', NULL AS 'deletedAt')--

- Successfully logged in as a fabricated account without registration.



- I made the non-exist account as admin and have all the authorizations:

Impact:

- Full access with arbitrary user creation through SQL injection

Recommendation:

- Use prepared statements
- Strict Input Validation
- **Error Handling:** Ensure no detailed database errors are leaked to the client during failed queries.

5.2. Authentication Bypass via SQL Injection

Critical

Description:

- SQL Injection is a critical vulnerability where attackers manipulate SQL queries by injecting malicious input. It occurs when user input isn't properly sanitized, letting attackers alter query logic and potentially gain unauthorized access or control.

Steps:

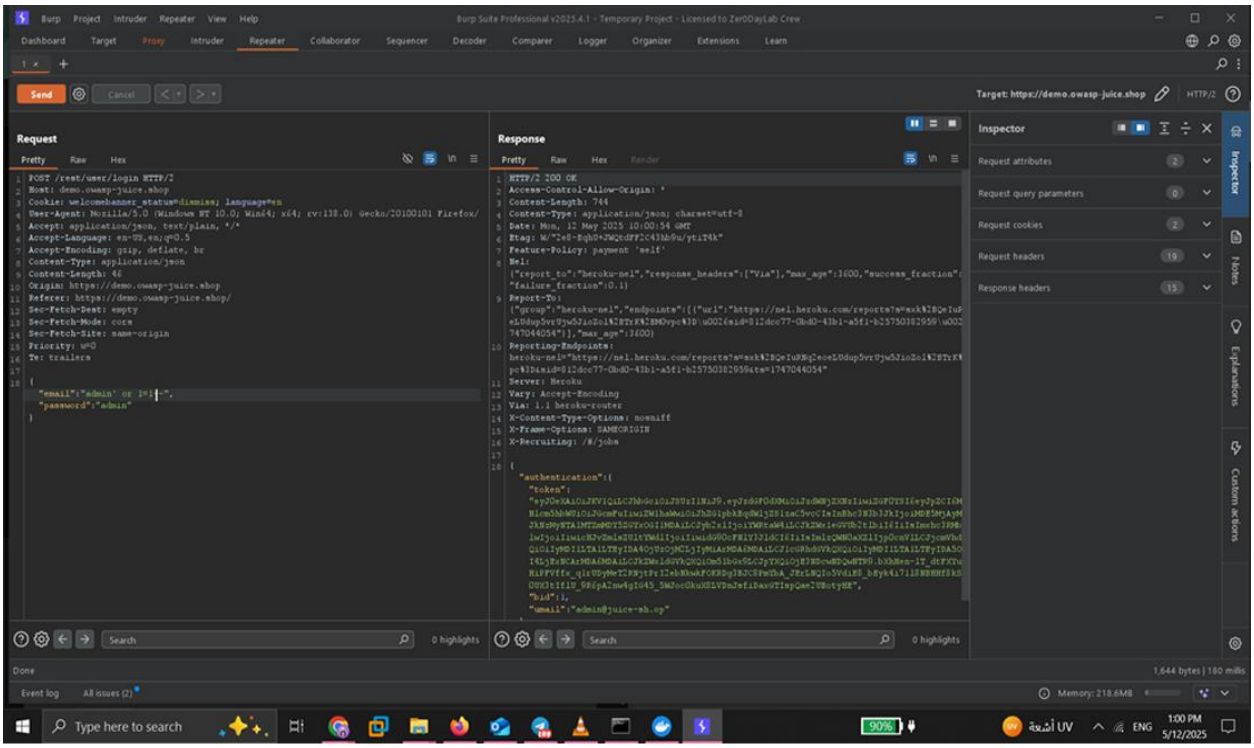
- Navigated to the login page.
- Entered the following payloads:
  - Email: ' OR 1=1—
  - Password: (any value or blank)
- Result: Gained unauthorized access as the admin user.
- Navigated to /administration to view all registered users.

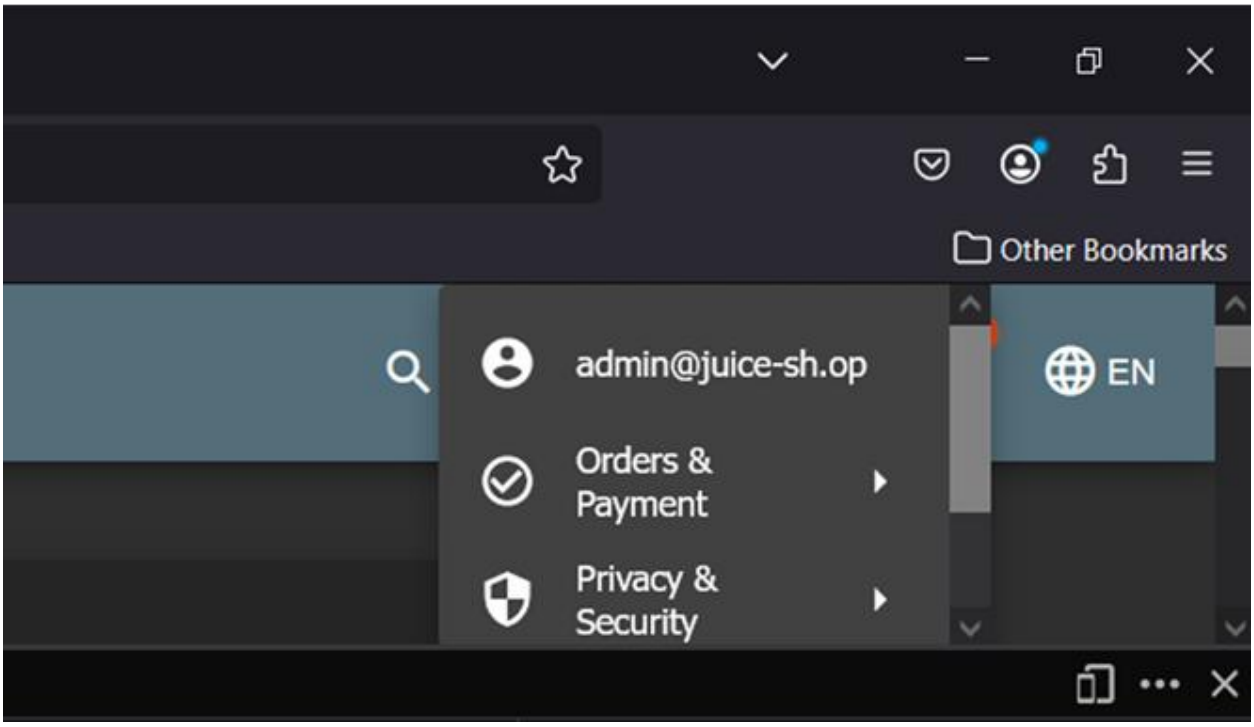
Impact:

  - **Full administrative access without valid credential**

Recommendations:

1. Use prepared statements (parameterized queries) to prevent injection
2. Sanitize and validate all user inputs.
3. Implement proper error handling to avoid exposing SQL errors
4. Apply least privilege principles on database accounts.


Request
Response
Inspector
Request attributes
Request query parameters
Request cookies
Request headers
Response headers
Type here to search


Other Bookmarks
admin@juice-sh.op
Orders & Payment
Privacy & Security
EN

5.3. JWT Decoding Revealing Password

Critical

Description:

- The application fails to properly validate JSON Web Tokens (JWTs) by accepting tokens with the algorithm (alg) field set to none. This dangerous misconfiguration effectively disables signature verification, allowing attackers to forge authentication tokens without any cryptographic proof of authenticity.

Steps:

- Used browser's Developer Tools and Cookie Editor to access the JWT token
- Decoded the JWT payload using a Base64 decoder
- Retrieved the password hash: 0192023a7bbd73250516f069df18b500
- Used an online hash identifier and confirmed the hash type as **MD5**
- Decrypted MD5 hash using public rainbow tables.
    - Password: admin123

Impact:

- **Disclosure of sensitive credentials through insecure token storage**

5.4. Bypass Login for Another User

High

Steps:

- Identified user email: (jim@juice-sh.op) from product reviews
- Attempted login with:
    - Email: jim@juice-sh.op'--
    - Password: (any value)
- Login was successful, bypassing password validation
- Extracted JWT token, decoded it, and retrieved password: Impact: ncc-170

Impact:

- Access to user accounts without valid credentials using SQL Injection

---

5.5. Logging In with a Non-Existent Account

High

Description:

- The application is vulnerable to a severe authentication bypass via SQL Injection that enables an attacker to log in without valid credentials and without the existence of a legitimate user account in the database. This vulnerability stems from improper sanitization of input fields in the login function, allowing the penetration tester to forge an ephemeral (non-persistent) user session using a manipulated SQL payload.

Impact:

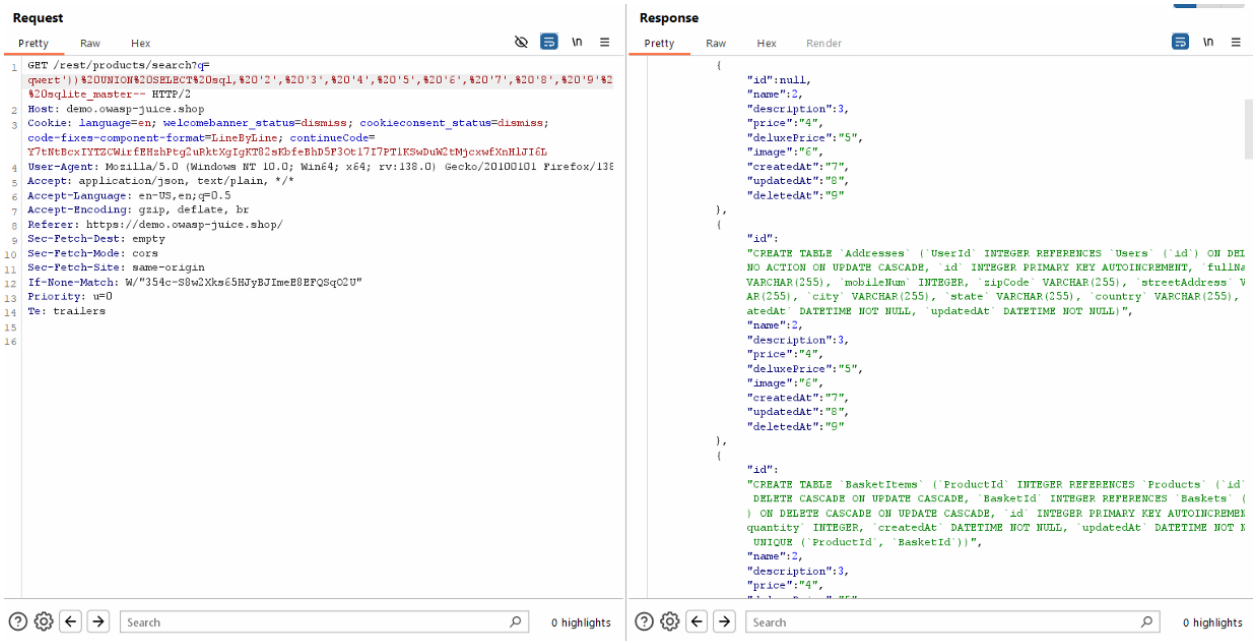
- Full authentication bypass without using a valid or stored account.
- Unauthorized access to the application, potentially under arbitrary or elevated session contexts.
- Creation of "phantom users" that are not stored in the database, evading logging, auditing, and access control checks.

Steps:

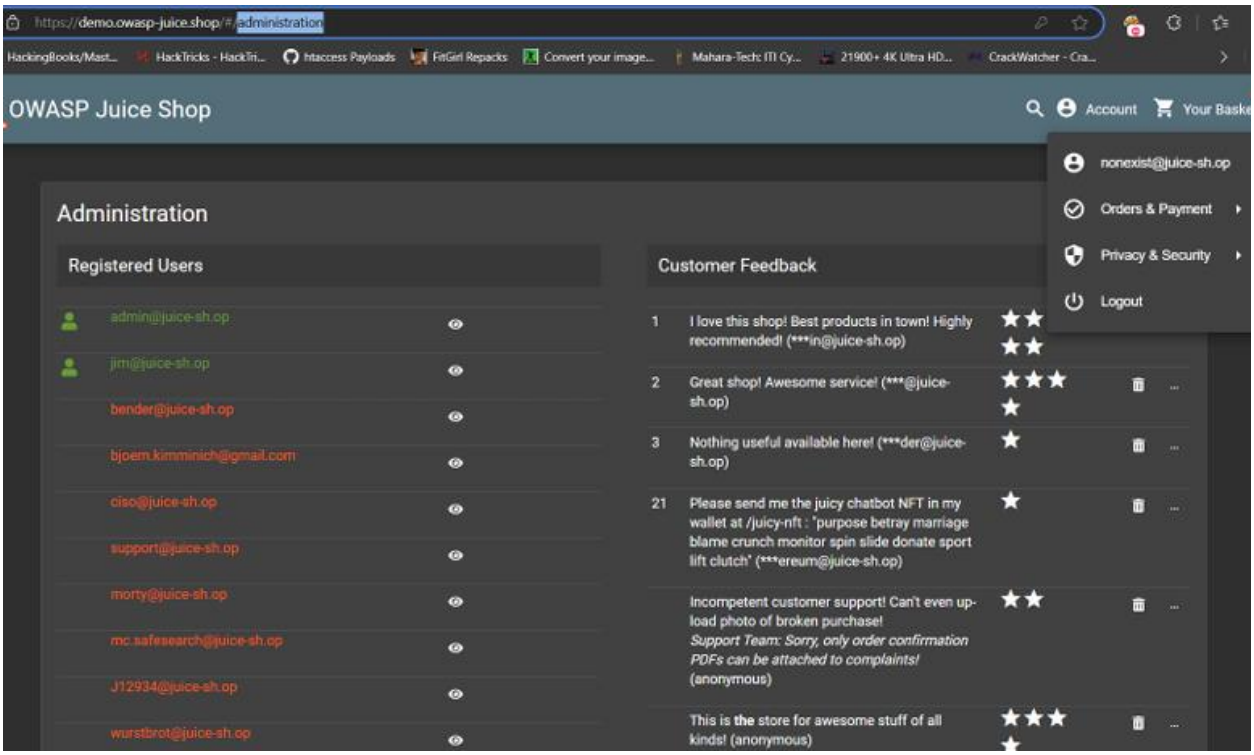
- Injected single quote ' in email field and captured the request using Burp Suite.
- Error message indicated backend is using SQLite.
- Performed the following payload via search functionality:
    - banana')) UNION SELECT sql, '2', '3', '4', '5', '6', '7', '8', '9' FROM sqlite_master—

- Retrieved the database schema.
- Crafted and submitted the following UNION SELECT payload to create a fake user:
    - ' UNION SELECT * FROM □SELECT 15 AS 'id', '' AS 'username', 'nonexist@juice-sh.op' AS 'email', '12345' AS 'password', 'admin' AS 'role', '123' AS 'deluxeToken', '1.2.3.4' AS 'lastLoginIp', '/assets/public/images/uploads/default.svg' AS 'profileImage', '' AS 'totpSecret', 1 AS 'isActive', '1999□08□16 14□14□41.644 □00□00' AS 'createdAt', '1999□08□16 14□33□41.930 □00□00' AS 'updatedAt', NULL AS 'deletedAt')—

- Successfully logged in as a fabricated account without registration.



- I made the non-exist account as admin and have all the authorizations:



5.5. Discovery of Backup Files Using Gobuster

Medium

Description:

- The application exposes a forgotten backup file (coupons_2013.md.bak) under the /ftp directory. Through null byte poisoning, a penetration tester can bypass file extension filtering mechanisms by injecting a %00 character into the filename, tricking the server into interpreting the request as one for a valid file. This misconfiguration results in the unintended exposure of sensitive internal data.

Impact:

- **Access Internal Backup Files:** Retrieve a historical backup containing sensitive sales-related information.
- **Sensitive Data Exposure:** Disclosure of internal coupon logic or promotional codes may assist attackers in abusing discounts or gaining insights into business operations.

Steps:

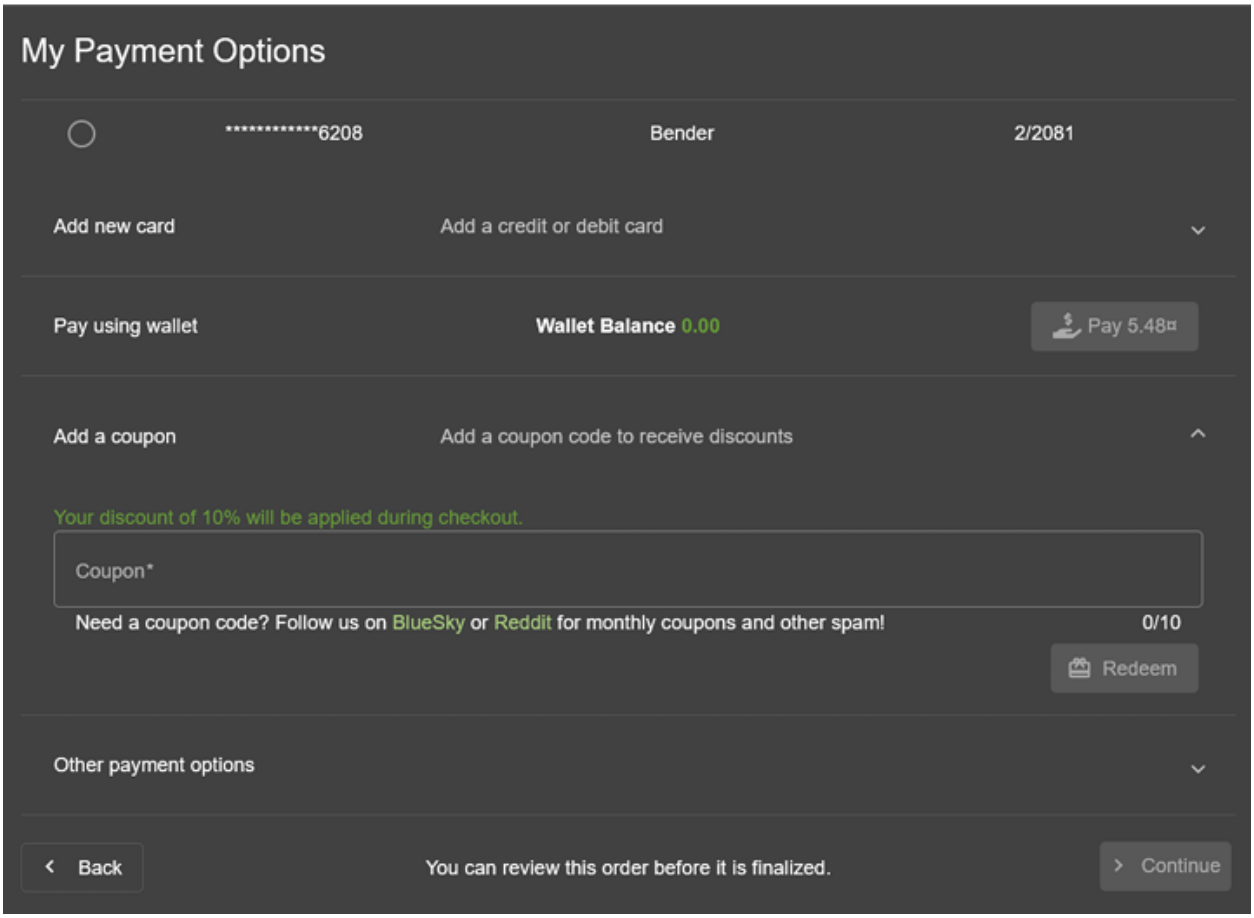
- Run Gobuster on demo.owasp-juice.shop:
  - Discovered hidden directory: /ftp
  - Located backup file: coupons_2013.md.bak (initially forbidden)
- Bypassed access restriction by URL encoding:
  - Accessed file via /ftp/coupons_2013.md%2500.bak
- File contained historical coupon codes.

5.6. Forging Coupon Codes via Encoded Values

High

Steps:

- Also downloaded package.json.bak from /ftp
- Identified that **Z85** encoding was used for coupon values
- Example Coupons:
  - pes[BgC7sn → **NOV13-10** (Valid on 2013/10/11)
  - k#*AfgC7sn → **AUG13-10** (Valid on 2013/10/8)
- Encoded **MAY25-10** to **Z85** → o*I]qh7ZKp
- Successfully redeemed the generated coupon.



Impact:

- Unauthorized redemption of coupons through reverse-engineered encoding.

6. Broken Access Control – Unauthorized Access to Admin Panel

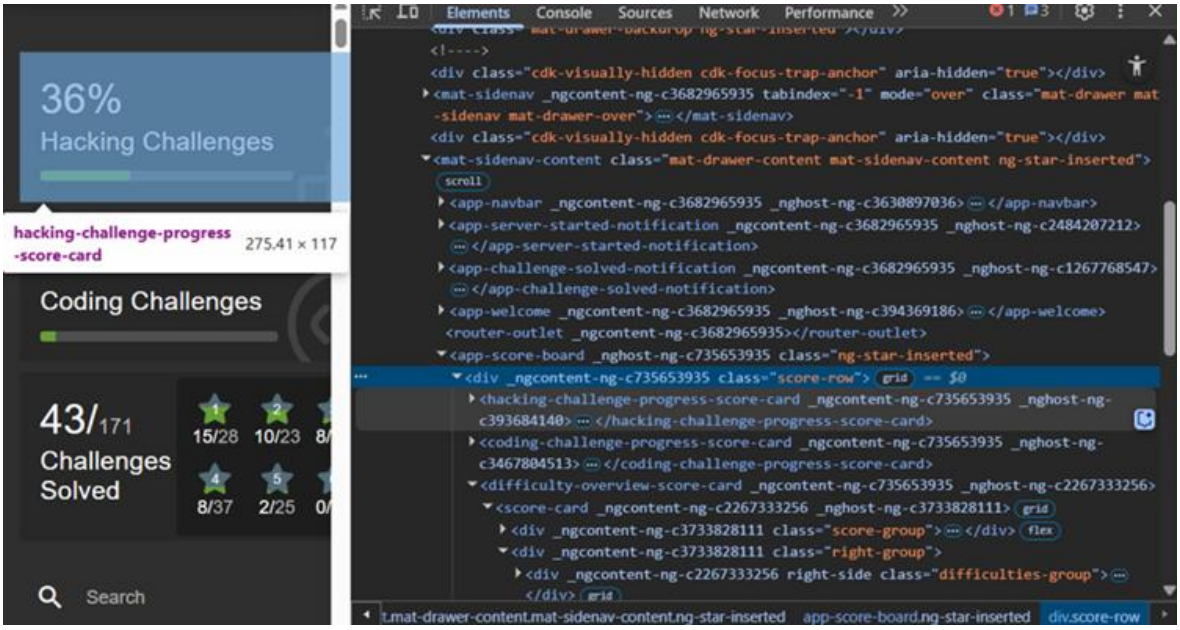
High

Description:

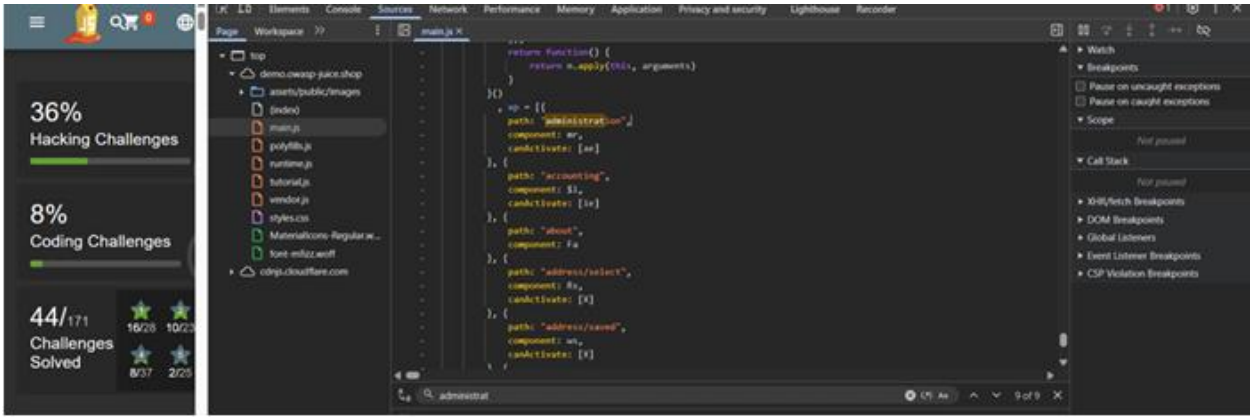
- The application allowed unauthenticated users to access the /administration route by simply entering the URL in the browser.

Steps:

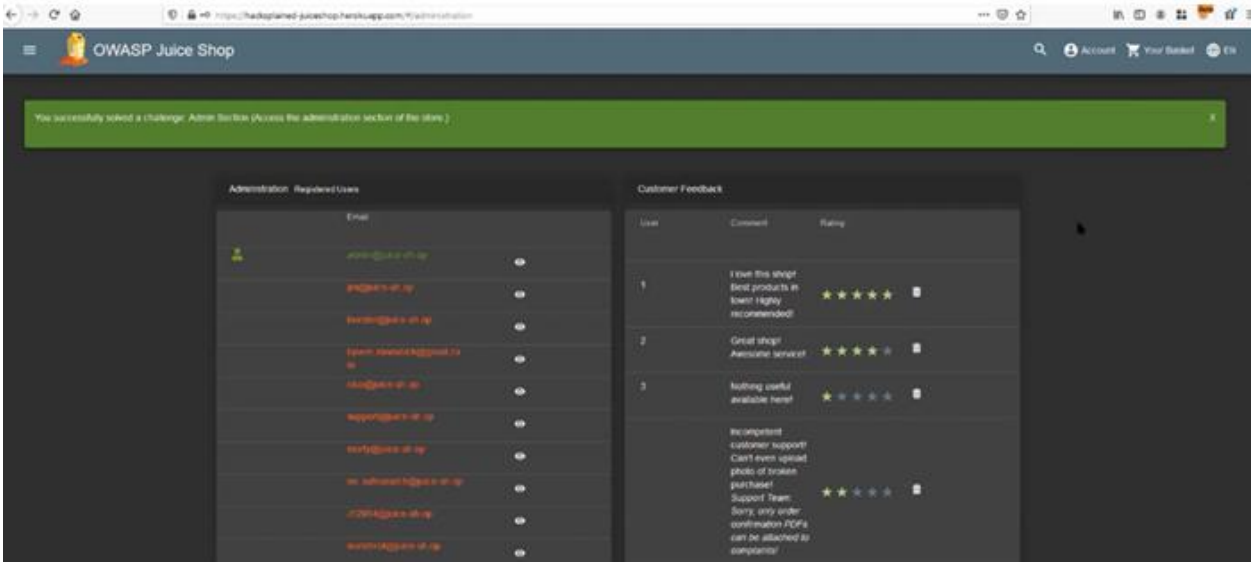
1. Step 1: Inspect the Web Page :
   - Open Developer Tools in the browser (F12).
   - Navigate to the Elements tab.
   - Noticed the presence of _ngcontent attributes.
   - Recognized the application uses Angular.



2. Step 2:Discover Hidden Routes.
   - Switched to the Sources tab in Developer Tools.
   - Opened and searched inside main.js under the debugger
   - Searched for the keyword path.
   - Found the route path: 'administration'.



3. Step 3: Access the Admin Section:
   - Navigated to:https://demo.owasp-juice.shop/#/administration
   - The admin section loaded without any authentication prompt.



Impact:

- Unauthorized access to administrative interface.
- Possible exposure of sensitive configuration.
- Risk of data manipulation or privilege abuse
- Violates principle of least privilege

Recommendation:

- Implement server-side authentication checks.
- Do not rely on obscurity (like hidden routes) to protect admin features.
- Protect routes using access control (frontend and backend).
- Monitor access logs for unauthorized URL visits.

7. Accessing Another User's Basket (IDOR)

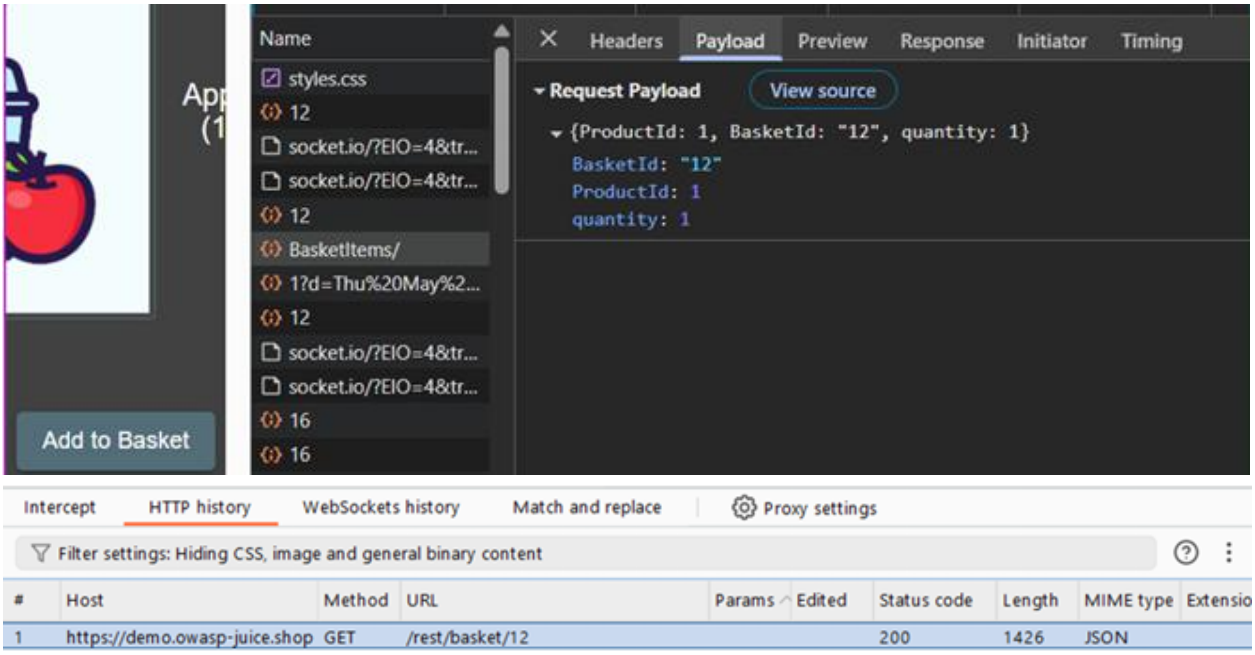
High

Description:

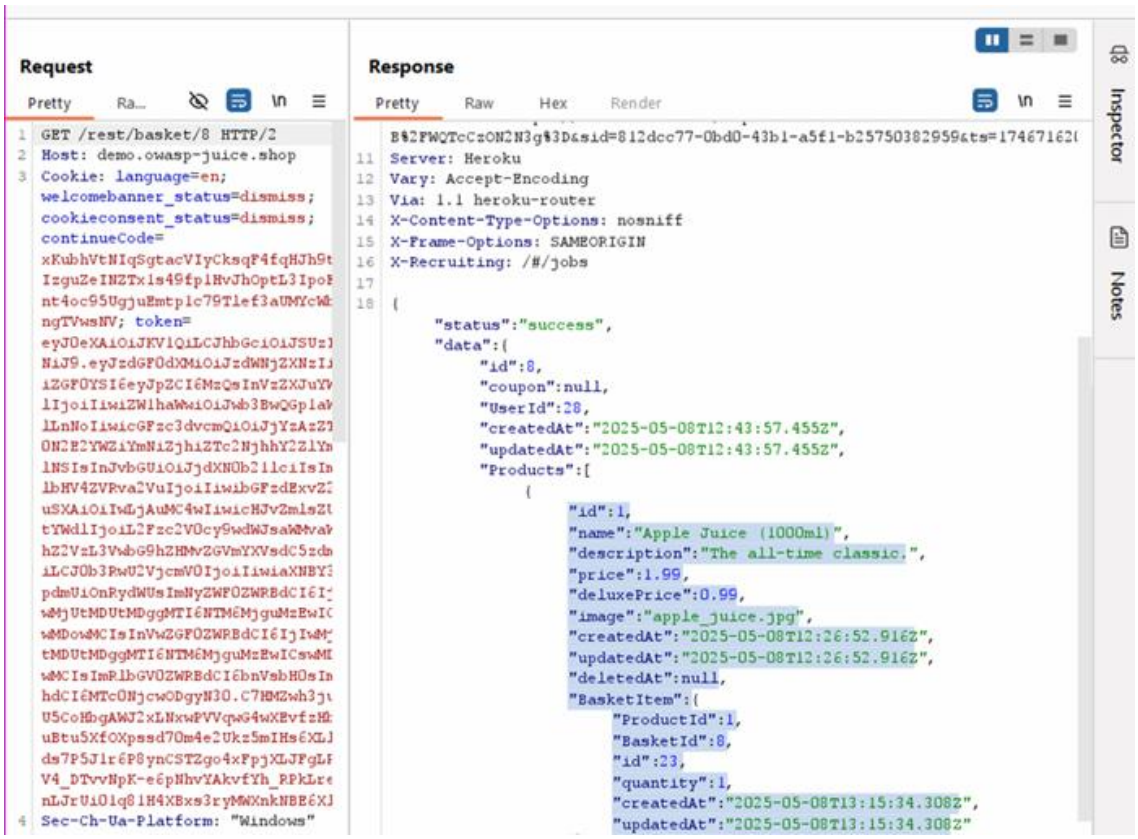
- Insecure Direct Object References (IDOR) occur when an application allows users to access or modify internal objects (like Basket IDs) without proper access control. I used Burp Suite to manipulate the Basket ID and the quantity of items in the basket, allowing me to modify another user's basket.

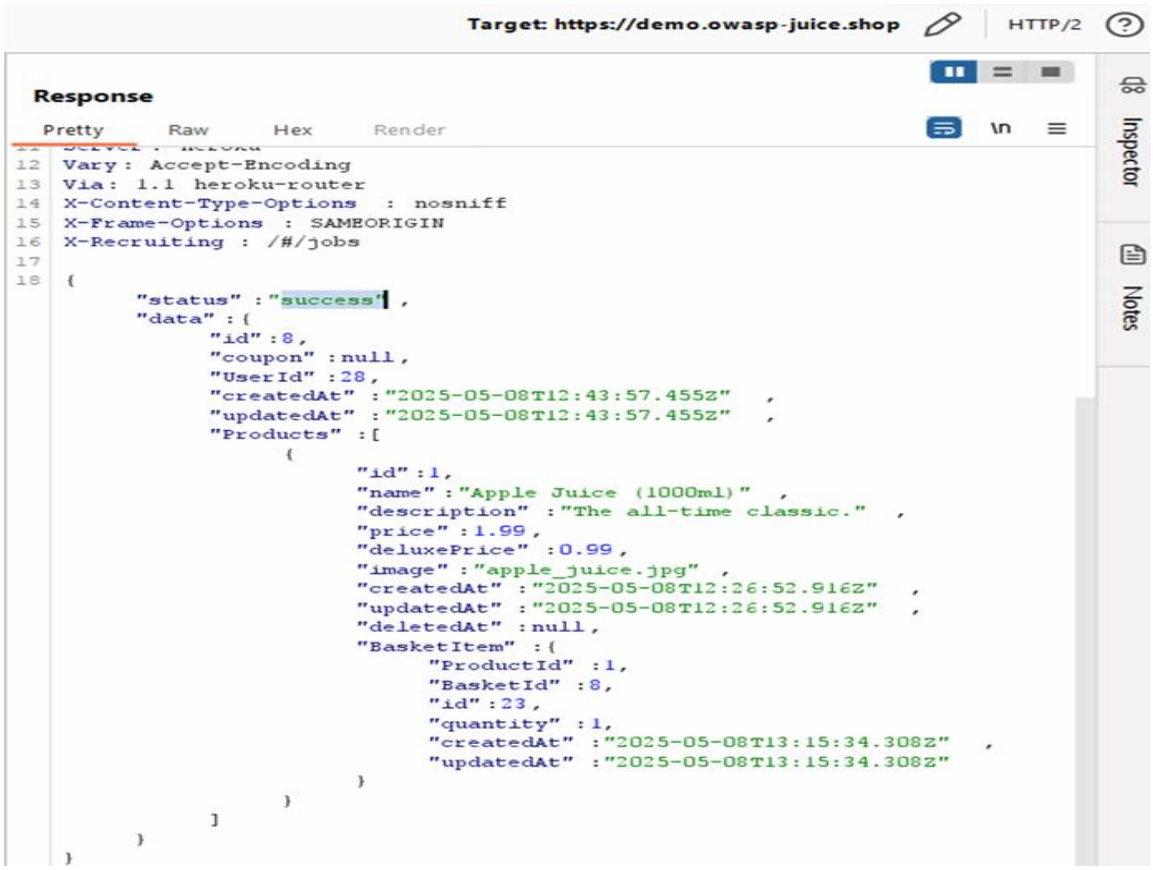
Steps:

1. Step 1: Send a POST request to access your own basket.
   - Accessed: Payload: https://demo.owasp-juice.shop/api/BasketItems/
   -
   - Payload



2. Step 2:Modify the Basketid to access another user's basket:
   - I intercepted the request using Burp Suite and modified the BasketId from 12 (my basket) to 8 (another user's basket).
   - After sending the modified request, the server responded with the details of BasketId 8, which belongs to a different user.
   - **Outcome**: The server accepted the request, and I was able to view another user's basket details.


Request
Pretty
Raw
Hex
Render
GET /rest/basket/8 HTTP/2
Host: demo.owasp-juice.shop
Cookie: language=en;
welcomebanner_status=dismiss;
cookieconsent_status=dismiss;
continueCode=
Sec-Ch-Ua-Platform: "Windows"
Response
Server: Heroku
Vary: Accept-Encoding
Via: 1.1 heroku-router
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
X-Recruiting: /#/jobs
{
"status":"success",
"data":{
"id":8,
"coupon":null,
"UserId":28,
"createdAt":"2025-05-08T12:43:57.455Z",
"updatedAt":"2025-05-08T12:43:57.455Z",
"Products":[
{
"id":1,
"name":"Apple Juice (1000ml)",
"description":"The all-time classic.",
"price":1.99,
"deluxePrice":0.99,
"image":"apple_juice.jpg",
"createdAt":"2025-05-08T12:26:52.916Z",
"updatedAt":"2025-05-08T12:26:52.916Z",
"deletedAt":null,
"BasketItem":{
"ProductId":1,
"BasketId":8,
"id":23,
"quantity":1,
"createdAt":"2025-05-08T13:15:34.308Z",
"updatedAt":"2025-05-08T13:15:34.308Z"
Inspector
Notes


Target: https://demo.owasp-juice.shop
HTTP/2
Response
Pretty
Raw
Hex
Render
Vary: Accept-Encoding
Via: 1.1 heroku-router
X-Content-Type-Options : nosniff
X-Frame-Options : SAMEORIGIN
X-Recruiting : /#/jobs
{
"status" : "success" ,
"data" : {
"id" : 8 ,
"coupon" : null ,
"UserId" : 28 ,
"createdAt" : "2025-05-08T12:43:57.455Z" ,
"updatedAt" : "2025-05-08T12:43:57.455Z" ,
"Products" : [
{
"id" : 1 ,
"name" : "Apple Juice (1000ml)" ,
"description" : "The all-time classic." ,
"price" : 1.99 ,
"deluxePrice" : 0.99 ,
"image" : "apple_juice.jpg" ,
"createdAt" : "2025-05-08T12:26:52.916Z" ,
"updatedAt" : "2025-05-08T12:26:52.916Z" ,
"deletedAt" : null ,
"BasketItem" : {
"ProductId" : 1 ,
"BasketId" : 8 ,
"id" : 23 ,
"quantity" : 1 ,
"createdAt" : "2025-05-08T13:15:34.308Z" ,
"updatedAt" : "2025-05-08T13:15:34.308Z"
}
}
]
}
}
Inspector
Notes

3. Step 3: Modify Items in Another User's Basket:

   - Using Burp Suite, I changed the quantity of an item in BasketId 8 from 1 to 10.
   - **Outcome**: The request was processed successfully, and the quantity of the item in the other user's basket was updated to 10, demonstrating that no proper access control checks were applied
   - The old Payload:

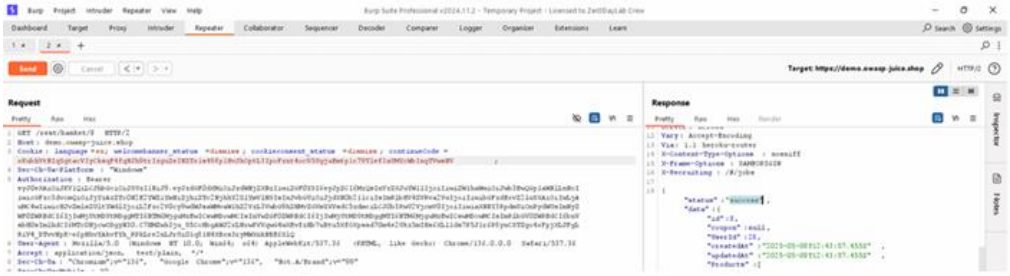

```
{
    "deleteCat": null,
    "BasketItem": {
        "ProductId": 1,
        "BasketId": 8,
        "id": 23,
        "quantity": 1,
        "createdAt": "2025-05-08T13:15:34.3082",
        "updatedAt": "2025-05-08T13:15:34.3082"
    }
}
```

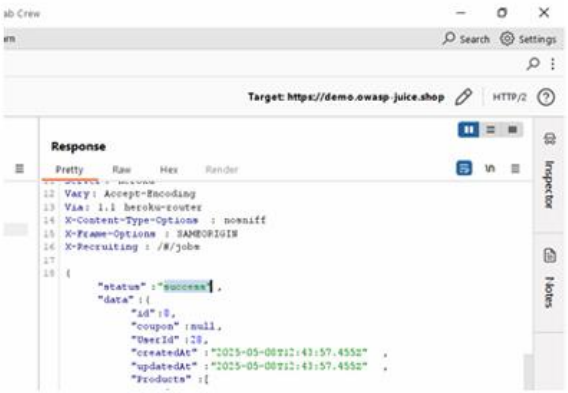

   - The new Payload:


```
{
    "deleteCat": null,
    "BasketItem": {
        "ProductId": 1,
        "BasketId": 8,
        "id": 23,
        "quantity": 10,
        "createdAt": "2025-05-08T13:15:34.3082",
        "updatedAt": "2025-05-08T13:15:34.3082"
    }
}
```

4. Step 4: Exploiting the Token Bypass:

    - I used Burp Suite to remove the Authorization token from the request header.
    - After sending the request again without the token, the server still accepted the request, indicating that the application was not properly validating authentication.
    - **Outcome:** The request was processed successfully, showing that the server did not properly authenticate the user before allowing the operation





Why This Is a Vulnerability?

- **Lack of Access Control:** The application does not validate whether the user is authorized to access or modify the data they are requesting, leading to unauthorized data exposure and modification.
- **Sensitive Data Exposure:** Internal object references like Basket IDs are exposed in the URL, allowing attackers to manipulate them and access or modify other users' data
- **Authentication Issues:** The server fails to properly authenticate requests, allowing unauthenticated users to perform actions they shouldn't be able to.
- **Data Integrity Risk:** Attackers can alter another user's basket, for example, by modifying the quantity of items, leading to potential fraudulent actions.

Impact:

- The vulnerability indicates weak access control, allowing attackers to make unauthorized changes to application data.
- The lack of proper token validation in sensitive requests means attackers can exploit this flaw to bypass authentication, potentially compromising user baskets, product inventories, or personal information

Recommendation:

- **Enforce Proper Access Control**
  - Verify user permissions before allowing access to objects.
  - Implement server-side checks for object ownership.
  - Associate resources (e.g., baskets) with owners and validate ownership before processing requests.
- **Tokens Authentication**
  - Reject requests without tokens
- **Avoid Exposing Direct Object References**
  - Do not expose internal IDs (e.g., UserId, BasketId) in URLs or requests.
  - Use indirect references (e.g., UUIDs) instead of sequential IDs
- **Monitor and Log Access to Sensitive Resources**
  - Log resource access with user context (e.g., UserId, accessed BasketId).

8. Stored XSS in Feedback Form

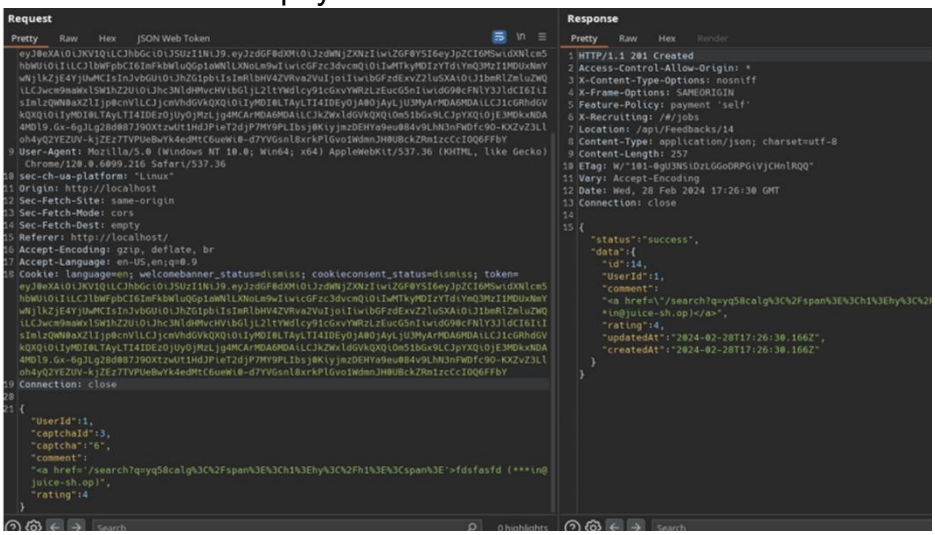
High

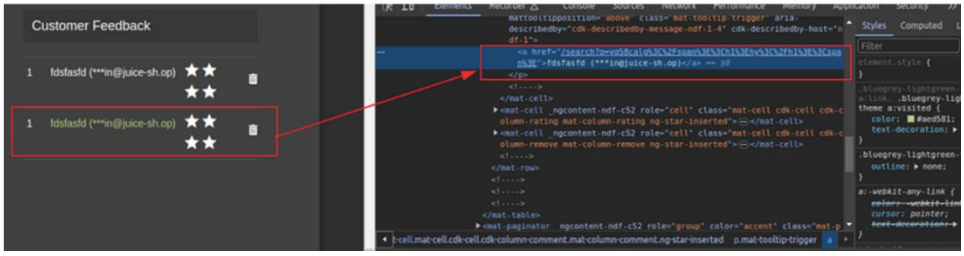
Description:

- The application fails to properly validate or encode user input before injecting it into the DOM using the innerHTML property. This makes it possible for an attacker to inject HTML tags — such as <a herf=" "> or potentially even <script> (under CSP bypass conditions) — into the administrator interface, which could escalate to full XSS under certain circumstances.
- The vulnerability is stored, meaning the payload persists in the backend and is rendered for each subsequent admin page visit

Steps to Reproduce:

1. Log in as a regular user.
2. Go to the "Customer Feedback" page.
3. Submit the following payload in the comment field:
   ClickMe
4. Rate with any number and complete CAPTCHA if required.
5. Submit the feedback.
6. Log in as admin and open the Feedback section under /administration.
7. Observe that the payload renders as a clickable link.





Impact:

- Persistent XSS allows attackers to inject malicious content that will execute in future admin sessions.
- Potential for session hijacking, phishing attacks, and privilege escalation.
- If chained with other vulnerabilities, this could lead to full compromise of the admin account.

Recommendations:

- Sanitize and escape all user input before storage and rendering.
- Replace innerHTML with safer alternatives like textContent.
- Apply a strong Content Security Policy (CSP).
- Use libraries like DOMPurify to clean dynamic HTML content.

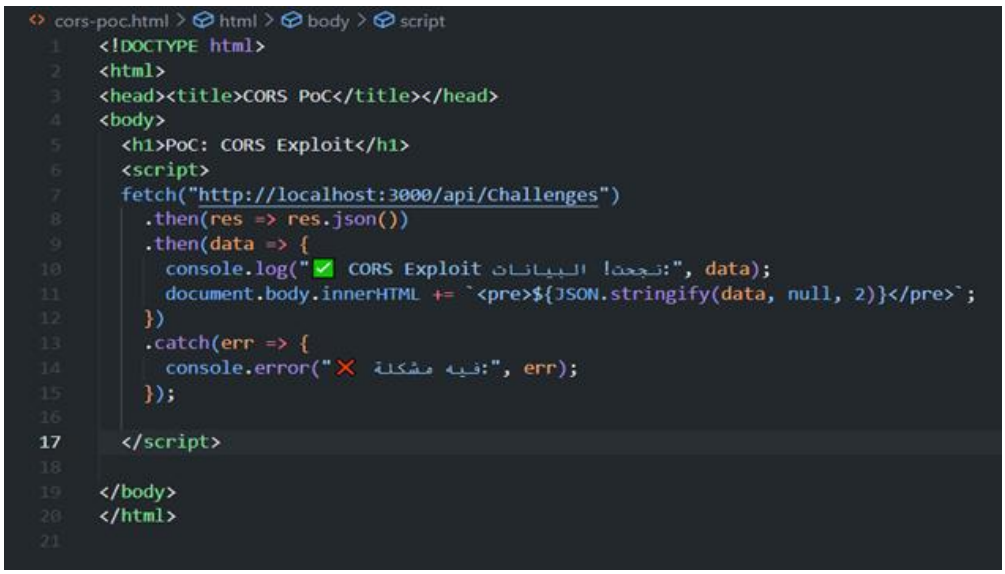
---

9. Cross-Origin Resource Sharing (CORS) Misconfiguration

Medium

Description:

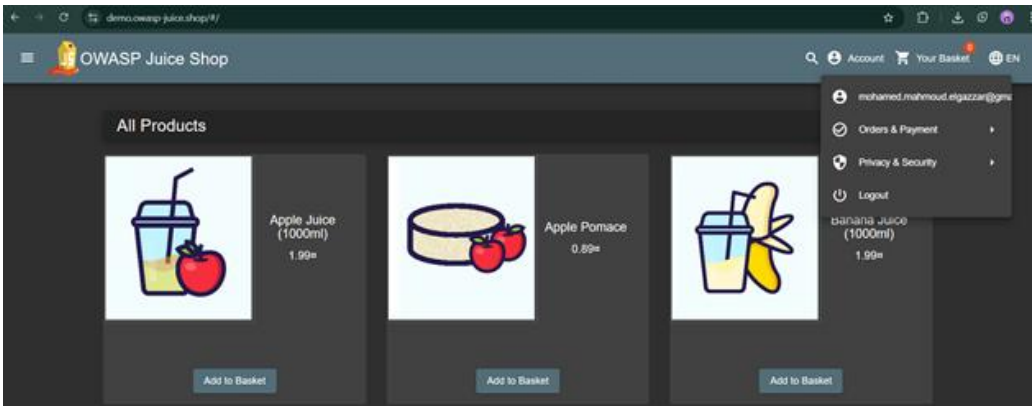
- The application accepts cross-origin requests from any origin, even with credentials, violating the Same-Origin Policy. This can allow attackers to perform authenticated requests from malicious websites
- **Proof of Concept (PoC):**
    - An attacker can lure the victim to visit a malicious HTML file (hosted on another origin like 127.0.0.1:5500) which makes an authenticated request to the Juice Shop API.
- **PoC HTML Code:**



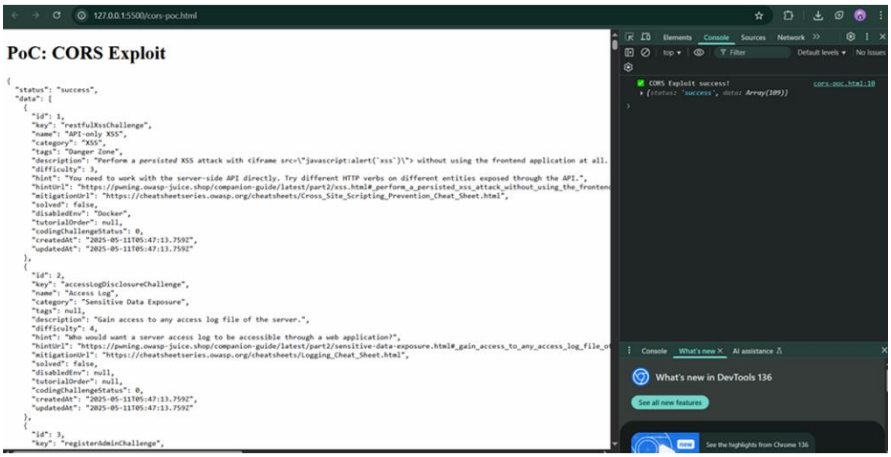
Steps:

1. Host the PoC HTML file using Live Server or any static file server on 127.0.0.1:5500.

2. Ensure Juice Shop is running on localhost:3000 and the victim is logged in.



3. Open the PoC HTML in the victim's browser.
4. Observe data being fetched from Juice Shop via console or HTML body



Impact:

- Any attacker-controlled origin can read sensitive data like user challenges or profile information if the victim is logged in. This violates the Same-Origin Policy and opens the door for session hijacking, data theft, or account manipulation.

Recommendation:

- **Restrict Access-Control-Allow-Origin to trusted origins.**
- **Avoid using Access-Control-Allow-Credentials: true unless required.**
- **Never use wildcard (*) when credentials are involved.**

---

10. CSRF in /profile Endpoint

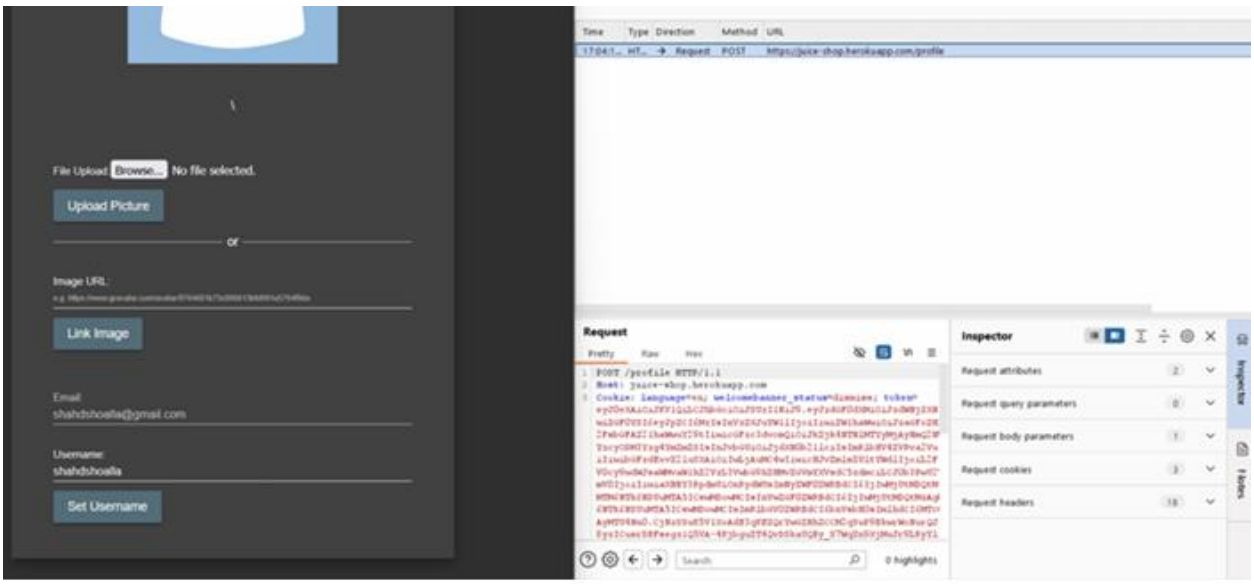
Medium

Description:

- This vulnerability allows an attacker to perform unauthorized actions on behalf of a logged-in user. In Juice Shop, I exploited this by crafting a malicious request that changed another user's data when they unknowingly clicked a link

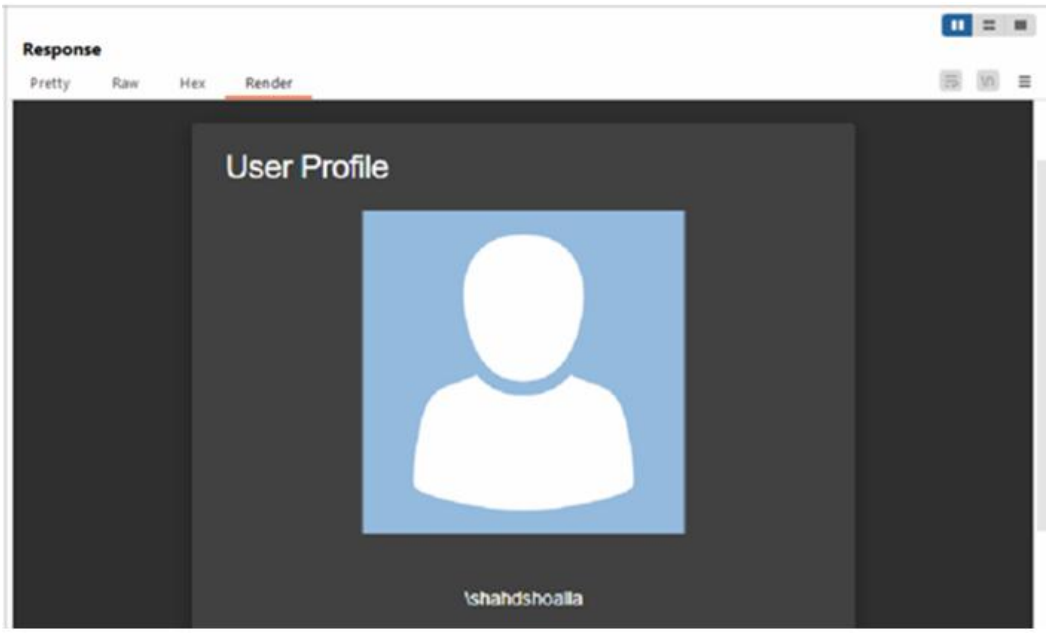
Steps:

1. **Step 1: Send Profile Update Request**
    - **Action:** Changed the username from the profile settings page
    - **Tool:** Captured the request in Burp Suite □Proxy □ HTTP history)
    - **Endpoint:** PUT /api/Users/profile
    - Headers included a Referer pointing to the Juice Shop origin

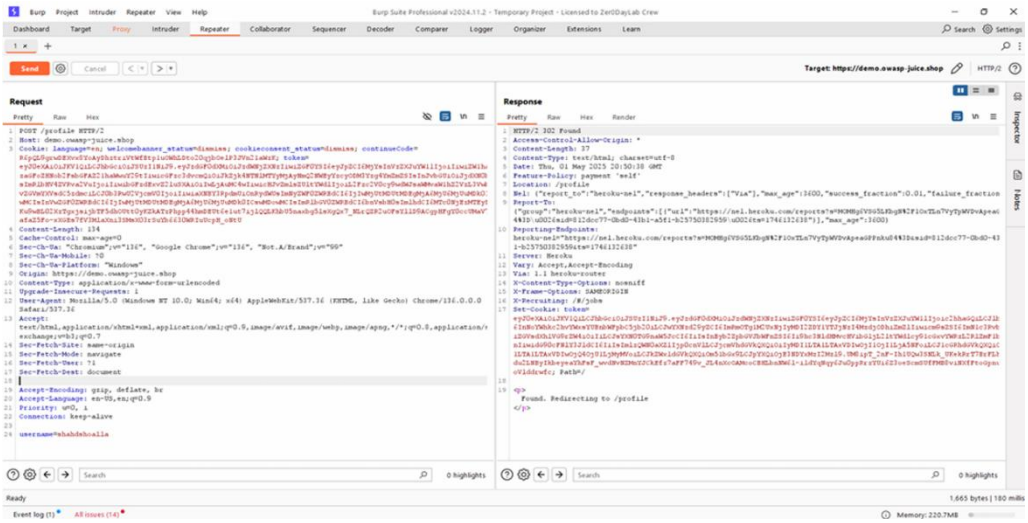


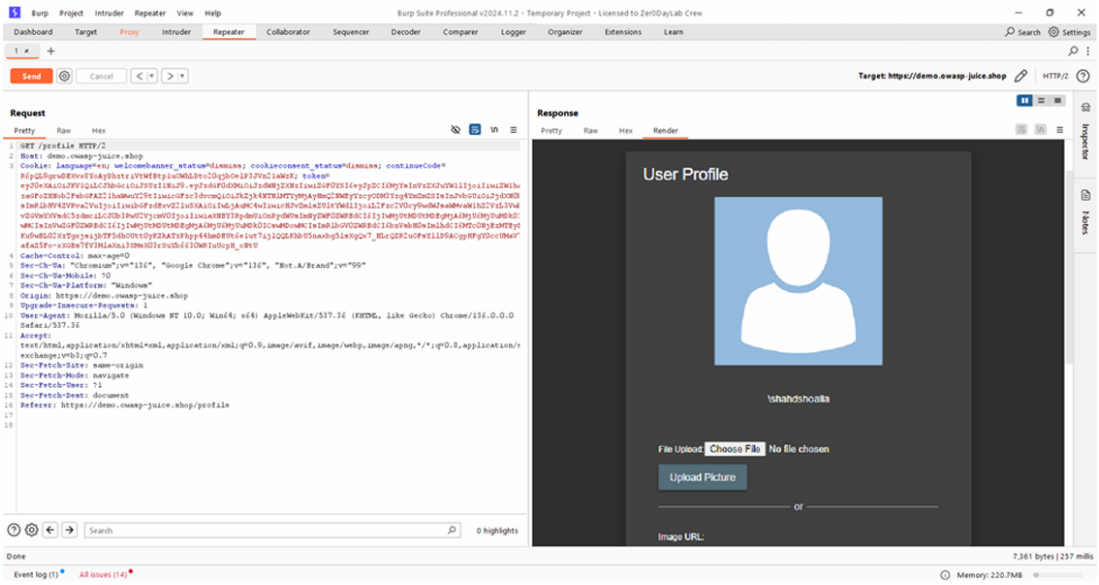

```
Origin: https://juice-shop.herokuapp.com
Referer: https://juice-shop.herokuapp.com/profile
```

2. **Step 2: Send Request to Repeater:**
    - Sent the same request to Burp Suite Repeater
    - Confirmed the server accepted the request and changed the username
    - This shows the server accepts profile updates if the user is authenticated

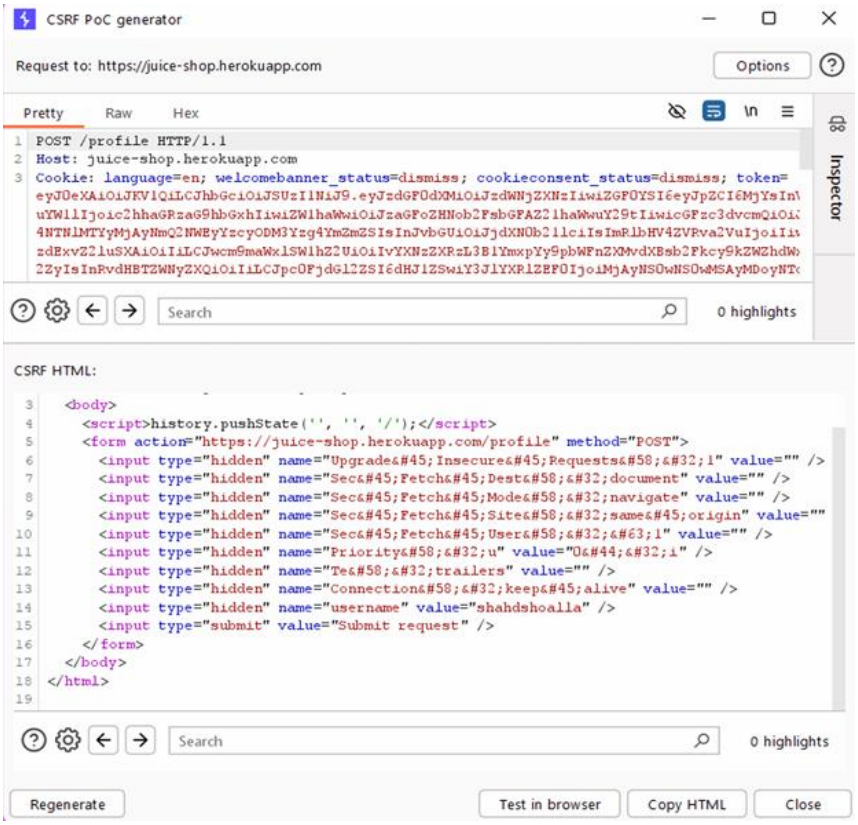


3. **Step 3: Remove Referer Header:**
    - Deleted the Referer from the request
    - Sent the request again
    - Result: Server still accepted the request

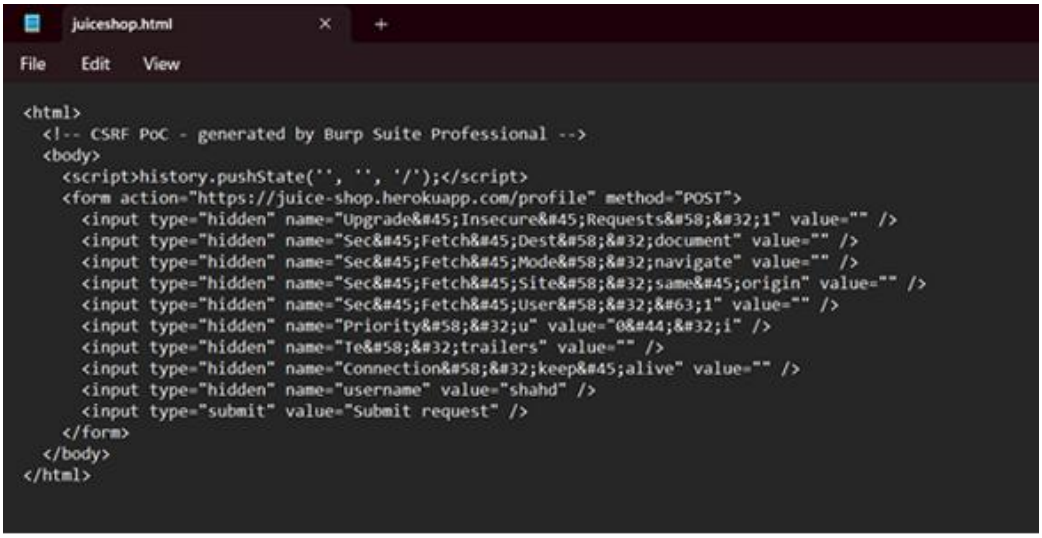


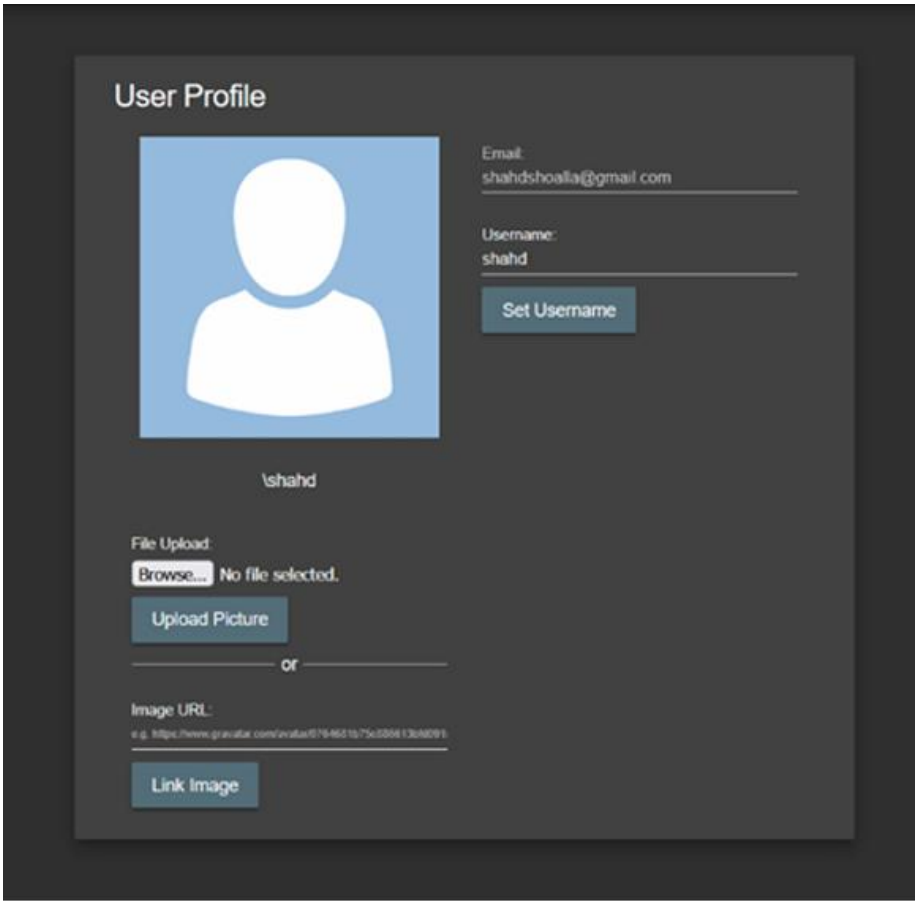


2. **Step 4: Create CSRF Exploit form:**
   - Used Burp Suite > Engagement Tools > Generate CSRF PoC
   - Got an auto-generated HTML form



3. **Step 5: Edit and Execute the Exploit:**
   - Saved the form as a .html file and change the username .
   - Opened the file in a browser
   - While logged into Juice Shop, loading this page submitted the form automatically





Impact:

- Attackers can trick logged-in users into submitting unauthorized requests
- This can lead to:
    - Profile tampering
    - Email or password changes
    - Session manipulation
- All without the user knowing or consenting

Recommendations:

1. **Require CSRF Tokens:**
    - Generate and validate a CSRF token for every sensitive form or state changing request
    - Tokens must be user-specific and unpredictable

2. **Enforce Origin or Referer Checks**
    - Reject requests that don't come from your domain
    - Validate Origin or Referer headers to ensure requests originate from a trusted source

3. **Use SameSite Cookies:**
    - Set cookies with SameSite=Strict or Lax to prevent them from being sent in cross-site requests

4. **Avoid Using GET for Sensitive Actions:**
    - Ensure that only POST, PUT, and DELETE methods are used for changing data
    - Do not allow sensitive changes via GET requests

6. Conclusion:

- The penetration test against OWASP Juice Shop uncovered a wide range of critical, high, and medium-severity vulnerabilities that reflect real-world attack vectors. The most impactful issues included SQL injection, broken authentication, insecure direct object references (IDOR), and improper access control—many of which allowed complete account takeovers and unauthorized data access. Additional findings such as CORS misconfiguration, CSRF flaws, stored XSS, and exposure of sensitive backup files highlight gaps in secure coding and deployment practices.

While Juice Shop is an intentionally vulnerable application, this assessment demonstrates the potential consequences of neglecting secure development principles. It also reinforces the importance of layered defenses, such as strict access controls, secure session handling, input validation, and comprehensive monitoring.

Addressing the identified issues with the recommended mitigations will significantly enhance the application's security posture. Future assessments should be conducted regularly, complemented by code reviews, automated scanning, and continuous security training for developers.