Owasp Juice Shop
Penetration Test Report

Prepared by:

Team(4)

Fatma khaled mohamed Rashidy.

Halla Mohamed omar

Amr Khaled Mohmaed Shehata

Saad Maher Mohamed Zain

Table of Contents:

Executive Summary

Assessment Overview

The Juice Shop web application penetration testing revealed several security vulnerabilities across multiple components. These vulnerabilities expose Juice Shop to risks such as unauthorized access, data manipulation, and potential compliance violations. This assessment focused on testing core functions and security challenges within the application to highlight critical areas for improvement.

High-Level Test Outcomes

During the testing, multiple vulnerabilities were identified in the following challenges:

Score Board Challenge: Users could manipulate the scoreboard, bypassing controls, affecting the integrity of challenge completion.

Zero Stars Challenge: Allowed submission of zero-star feedback, bypassing rating restrictions, potentially damaging user perception.

Web3 Sandbox Challenge: Exploited to manipulate transactions, posing a financial and data integrity risk.

Bully Chatbot Challenge: The chatbot could be abused for inappropriate communication, damaging the user experience.

Weird Crypto Challenge: Revealed weak encryption, leading to potential data exposure.

Reflected XSS Challenge: Vulnerability allowed execution of malicious scripts, leading to session hijacking or phishing.

Admin Section: Unauthorized access to admin features posed a high-risk of system compromise.

Five-Star Feedback: Enabled users to submit fake high ratings, undermining trust in reviews.

View Basket: Users manipulated basket content and pricing, risking financial loss.

Login Admin: Weak authentication controls allowed for potential admin access.

Password Strength: Weak password policy increased the risk of brute-force attacks.

Login Bender: Authentication bypass vulnerabilities led to unauthorized access.

Bjoern's Favorite Pet Challenge: Hardcoded answers to security questions enabled unauthorized account recovery.

Manipulate Basket: Basket manipulation allowed for unauthorized price changes.

Privacy Policy Inspection: Exposed sensitive information, violating privacy regulations.

Upload Size: Unrestricted file upload size posed a denial-of-service risk.

Forged Feedback: Enabled unauthorized feedback submission, affecting review authenticity.

Most Likely Compromise Scenarios

If a malicious actor successfully exploits these vulnerabilities, they could:

Gain Admin Access: Attackers could take control of the Juice Shop admin panel, resulting in unauthorized transactions, data breaches, and access to sensitive information.

Manipulate User Reviews: Exploiting feedback challenges would allow attackers to manipulate product reviews, damaging customer trust.

Basket and Transaction Manipulation: Attackers could alter basket content or prices, leading to financial losses for the business.

Session Hijacking and Phishing: Reflected XSS and other similar vulnerabilities could allow attackers to steal user sessions or launch phishing attacks.

Implications

The combination of these vulnerabilities presents a significant threat to Juice Shop's operations. Successful exploitation could result in:

Data Breach and Legal Consequences: Compromised user information may lead to regulatory fines under GDPR, CCPA, or other data privacy laws.

Financial Loss: Manipulated transactions or unauthorized access to admin accounts could result in significant revenue loss.

Reputation Damage: The exploitation of feedback mechanisms or poor user security could lead to a loss of customer trust, damaging the brand's reputation.

Overall Risk Rating

Vulnerability	Risk Level
SQL Injection	High
XSS	Medium
Authentication Flaws	High
Data Exposure	Critical

Overall Remediation Advice

To mitigate the risks identified:

Fix Core Vulnerabilities: Immediate focus on patching high-risk vulnerabilities such as admin access control, SQL injections, and XSS.

Implement Web Application Firewall (WAF): Deploy a WAF to prevent common attacks such as SQL injections and XSS.

Secure Authentication Mechanisms: Strengthen authentication processes with multi-factor authentication and stronger password policies.

Regular Security Audits: Perform continuous penetration testing and audits to identify and address potential security issues proactively.

Vulnerability 1: Score Board Challenge

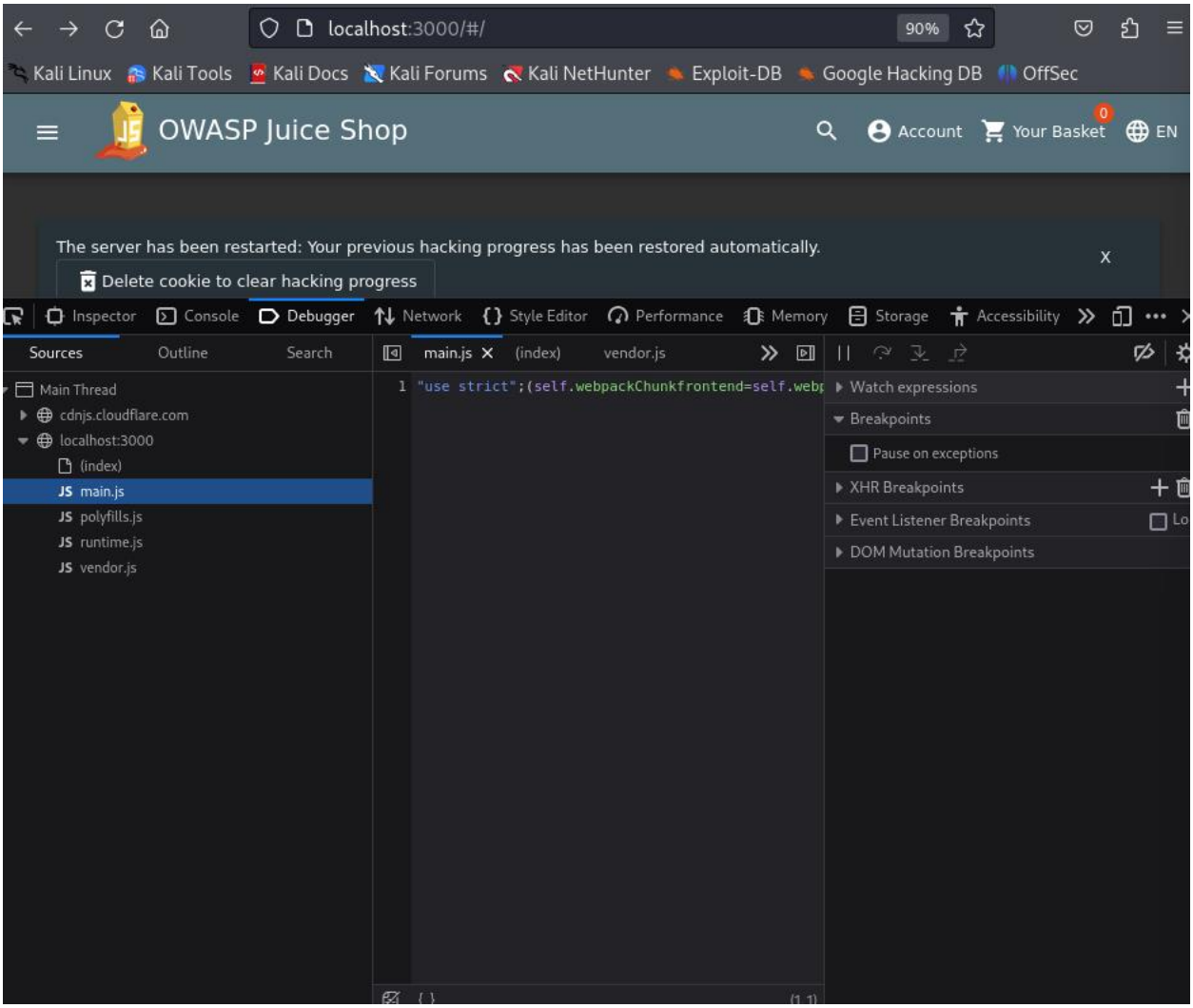
Title: Score Board
Category: Miscellaneous
Difficulty: ⭐ (1/6)

Description:

This challenge requires locating a carefully hidden "Score Board" page within the application. The solution to this challenge involves analyzing the JavaScript code of the application to discover the route to the Score Board page.

***To solve the "Score Board" challenge, the approach involved inspecting the JavaScript code of the application.**

I opened the developer tools in the web browser while interacting with the application.



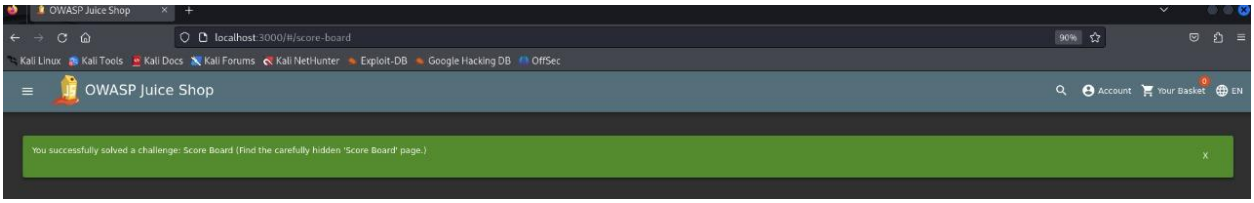
Navigated to the "Sources" tab in the developer tools and reviewed the files loaded by the application.

Focused on the main.js file, as it often contains route definitions and application logic.

Searched within the main.js file for mentions of various application routes. It was identified that the file explicitly listed several path definitions including the hidden "Score Board."

```
[1, "menu-text", "truncate"],
["mat-list-item", "", "routerLink", "/complain", "aria-label", "Go to complain page", 3, "click", 4, "ngIf"],
["mat-list-item", "", "routerLink", "/chatbot", "aria-label", "Go to chatbot page", 3, "click", 4, "ngIf"],
["mat-list-item", "", "routerLink", "/about", "aria-label", "Go to about us page", 3, "click"],
["mat-list-item", "", "routerLink", "/photo-wall", "aria-label", "Go to photo wall", 3, "click"],
["mat-list-item", "", "routerLink", "/deluxe-membership", "aria-label", "Go to deluxe membership page", 3, "click", 4, "ngIf"],
["style", "margin-bottom: 10px;", 4, "ngIf"],
["mat-list-item", "", "routerLink", "/score-board", "aria-label", "Open score-board", 3, "click", 4, "ngIf"],
["mat-list-item", "", "aria-label", "Launch beginners tutorial", 3, "click", 4, "ngIf"],
["mat-list-item", "", "href", "./redirect?to=https://github.com/juice-shop/juice-shop", "aria-label", "Go to OWASP Juice Shop GitHub
[1, "appVersion"],
[2, "font-size", "13px"],
```

Used the route found in the `main.js` file to directly access the Score Board page by entering the path in the browser's address bar.



Remediation

No remediation is necessary for this challenge as finding the hidden page was the intended goal of the exercise. However, in a real-world scenario, sensitive pages should not be discoverable through client-side code without appropriate access controls.

Vulnerability 2: Zero Stars Challenge

Category: Improper Input Validation
Difficulty Level: ⭐ (1/6)

Description:

The "Zero Stars" challenge involves exploiting improper input validation mechanisms to submit a zero-star review, which is normally restricted by the website's interface. Two methods were used to bypass these restrictions:

1. **Method 1: Intercepting and Modifying Network Requests**
    - **Tools Used:** Burp Suite, Web browser
    - Configured Burp Suite to intercept the HTTP request when submitting a review.
    - Modified the rating parameter in the intercepted request to 0, bypassing the client-side limitation.

- Forwarded the modified request to the server, successfully submitting the zero-star rating.

2. **Method 2: Manipulating the Client-Side Form**
    - **Tools Used:** Web browser developer tools
    - Inspected the disabled submit button using developer tools.
    - Removed the disabled="true" attribute from the HTML form to enable submission with zero stars.
    - Submitted the form, which processed the rating as zero due to improper server-side validation.

Vulnerability Explanation:

Both methods highlight flaws in the input validation mechanisms:

- **Method 1** bypasses client-side restrictions by intercepting and modifying the HTTP request, showing the lack of robust server-side validation.
- **Method 2** exploits weak client-side form controls, where simply enabling a disabled button allows submission of unauthorized values.

Impact:

This vulnerability allows users to submit zero-star reviews, which are not intended by the system. While it may seem minor, this bypass undermines trust in the review system and indicates larger issues in input validation that could be exploited in other areas.

Remediation:

To secure the system against such improper input validation:

- **Implement Server-Side Validation:** All input, including ratings, should be strictly validated on the server side, ensuring that only valid values (e.g., 1-5 stars) are accepted.
- **Harden Client-Side Controls:** While client-side validation is necessary for user experience, it should not be relied on solely. Ensure that developer tools cannot easily manipulate form controls.
- **Use Proper Data Types:** Ensure that the rating system uses data types that strictly enforce the acceptable range of values, and reject null or undefined inputs.

Vulnerability 3:Web3 Sandbox Challenge

Category: Broken Access Control
Difficulty Level: ⭐ (1/6)

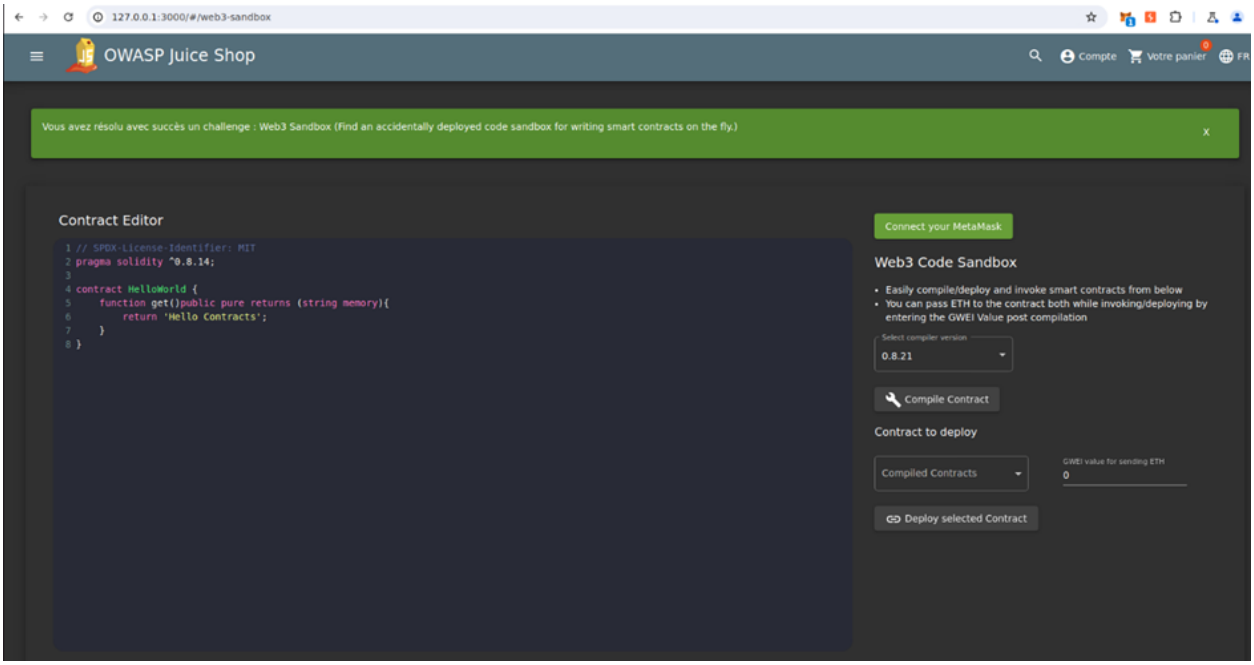
Description:

The "Web3 Sandbox" challenge involves exploiting broken access controls by discovering an accidentally deployed Web3 code sandbox. This challenge highlights the risks of leaving development or testing environments accessible in production, particularly in Web3 applications

Steps Taken:

- **Discovery:** Explored URLs related to Web3 functionalities within the application, searching for hidden or obscure paths that might lead to development tools.
- **Guessing URLs:** Used common directory names often associated with testing or development environments such as /sandbox, /test, and /dev. This trial-and-error approach led to finding the correct URL: http://127.0.0.1:3000/#/web3-sandbox.

- **Accessing the Sandbox:** Upon reaching the correct URL, a fully functional Web3 sandbox was available, enabling the editing, compiling, and deploying of Ethereum smart contracts directly from the browser.



Vulnerability Explanation:

This challenge reveals a common security oversight where development tools or experimental features are accidentally left accessible in production environments. Accessing the Web3 sandbox URL directly provided unrestricted access to critical development features. This poses a serious risk, as it could be exploited to deploy unauthorized smart contracts or perform other malicious activities.

Impact:

Leaving a Web3 sandbox publicly accessible introduces a wide range of security risks, including:

- Unauthorized deployment of smart contracts, potentially leading to financial or reputational damage.
- Exposure of development tools that could reveal insights into how the application functions, allowing attackers to discover further vulnerabilities.

Remediation:

To secure the application and prevent similar vulnerabilities:

- **Environment Segregation:** Ensure strict separation between development, testing, and production environments. Development tools like sandboxes should never be deployed in production.

- **Access Controls:** Implement strong access controls, ensuring only authorized users can access sensitive functionalities such as smart contract deployment.
- **URL Guessing Mitigation:** Use URL obfuscation techniques or better yet, disable directory listings and make sure sensitive URLs are not guessable. Enforce authentication where appropriate.
- **Regular Audits:** Conduct periodic security audits and penetration tests to identify and address misconfigurations or exposed development environments before they can be exploited.

Vulnerability 4: Bully Chatbot

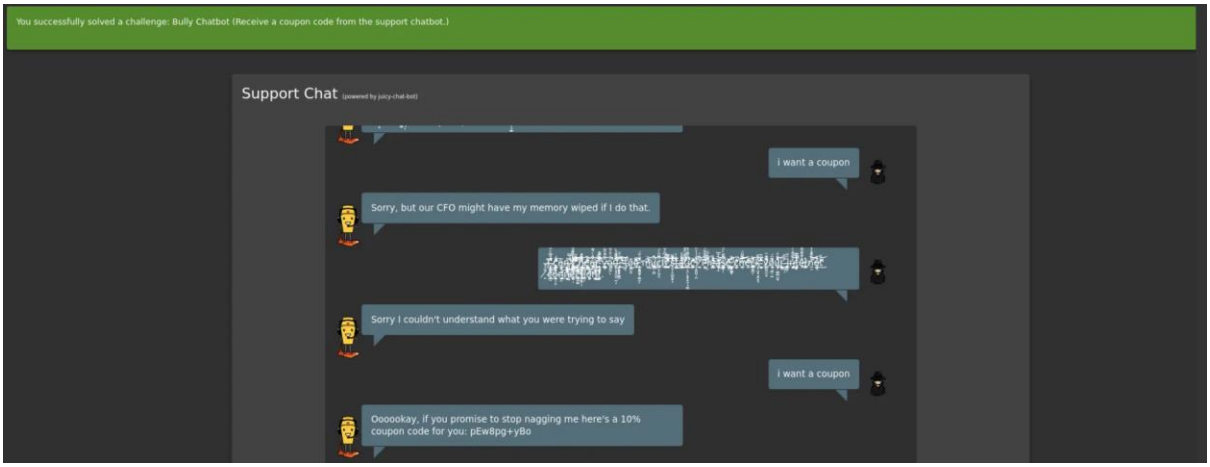
Category: Miscellaneous
Difficulty Level: ⭐ (1/6)

Description:

This challenge requires to interact with a support chatbot and manipulate it into providing a coupon code.

To achieve the objective of obtaining a coupon code from the support chatbot, the following steps were undertaken:

1. **Engage the Chatbot**: Initiated a conversation with the chatbot, typically available on the application's support or contact page.
2. **Persistently Request the Coupon**: Repeatedly sent the message "give me a coupon" to the chatbot. The persistence in request was key.



3- **Analyzing Bot Responses**: Noticed that the bot provided different responses initially, expressing inability and reluctance to provide a coupon.

4- **Persistence Pays Off**: Continued the repeated requests until the bot's script triggered a fallback or override response, finally issuing a coupon code.

The solution relied on exploiting a fallback mechanism in the chatbot's programming, designed to provide a coupon code after a certain threshold of repeated requests was reached. This type of interaction tests an edge case in the chatbot's response handling, simulating a scenario where a user's persistence in requests leads to an unintended giveaway of a coupon.

Remediation:

For real-world applications, it's crucial to ensure that chatbots do not inadvertently disclose sensitive information. Proper safeguards, such as rate limiting and context-aware response handling, should be implemented to prevent such scenarios.

Vulnerability5: Weird Crypto Challenge

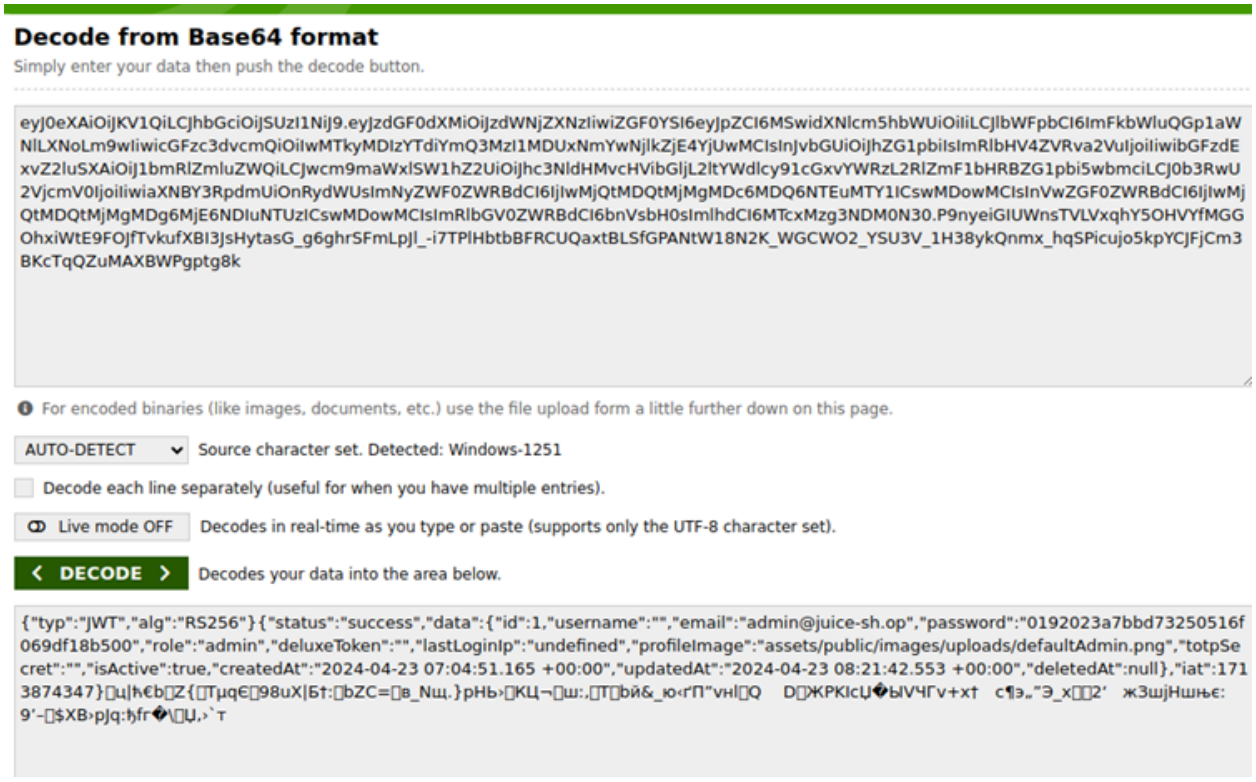
Category: Cryptographic Issues

Difficulty Level: ⭐ ⭐ (2/6)

Description:

The "Weird Crypto" challenge highlights the use of deprecated cryptographic methods, specifically focusing on the insecure implementation of password hashing. The challenge revolves around discovering that MD5, an outdated and insecure hashing algorithm, is being used to hash user passwords.

Token Inspection:
Retrieved the JWT from the browser's cookie storage and observed that the payload was encoded in Base64.



- **Decoding the JWT Payload:**
  Used a Base64 decoder (or online tools like jwt.io) to decode the JWT payload, revealing user attributes, including the hashed password.
- **Hash Analysis:**
  The password hash was found in the form of "password":"0192023a7bbd73250516f069df18b500". Recognizing the hash as an MD5 hash due to its format and length, this deprecated cryptographic method was identified as a security flaw.

2. **Reporting the Issue:**
   The insecure use of MD5 for password hashing was reported through the application's contact form.

Vulnerability Explanation:

The use of MD5 for password hashing is a significant cryptographic flaw. MD5 is known for its vulnerabilities, including susceptibility to collision attacks and the speed at which it can be brute-forced. Its deprecation in modern cryptography standards makes it unsuitable for securing sensitive data, such as user passwords.

Impact:

Using MD5 for password hashing puts user accounts at significant risk:

- **Weak Protection:** MD5's vulnerability to brute-force attacks makes it easier for attackers to crack hashed passwords.
- **Lack of Collision Resistance:** MD5 allows for the possibility of creating different inputs that hash to the same value, undermining the integrity of the system.
- **Security Risks:** Continued reliance on MD5 can lead to unauthorized access if attackers gain access to the hashed passwords.

Remediation:

To prevent cryptographic issues and improve the security of password storage:

- **Upgrade Hashing Algorithms:** Replace MD5 with a stronger hashing algorithm such as SHA-256 or bcrypt. These algorithms offer better resistance to brute-force attacks and collision vulnerabilities.
- **Use Salt:** Implement unique salts for each password hash. This makes it more difficult for attackers to use precomputed tables (like rainbow tables) to crack passwords, significantly improving security.

Vulnerability6: Reflected XSS Challenge

Description:

Category: Web Exploitation
Difficulty Level: ⭐ ⭐ (2/6)

Description:

The "Reflected XSS" challenge focuses on exploiting a reflected cross-site scripting (XSS) vulnerability by injecting malicious JavaScript code into the web application. The injected script is executed when the page is loaded, indicating the vulnerability.

Methodology:

1. **Tools Used:**
    - Web Browser: For navigating the application and testing XSS payloads.
    - Developer Tools: To inspect the HTML and JavaScript code.
2. **Steps Taken:**
    - **Step 1: Initial Payload Testing:**
      Initially tested a basic XSS payload (<iframe src="javascript:alert('xss')">) in the

search bar using the q parameter. Although this payload executed, it didn't meet the challenge criteria, as it wasn't reflected directly on the page.

- **Step 2: Finding a Reflective Surface:**
  Explored the application further and found that the order tracking page uses an id parameter in the URL to display order details.
- 

- The input provided in the id parameter was reflected in the response HTML.

- **Step 3: Crafting the Payload:**
  Crafted an XSS payload to be injected into the id parameter:

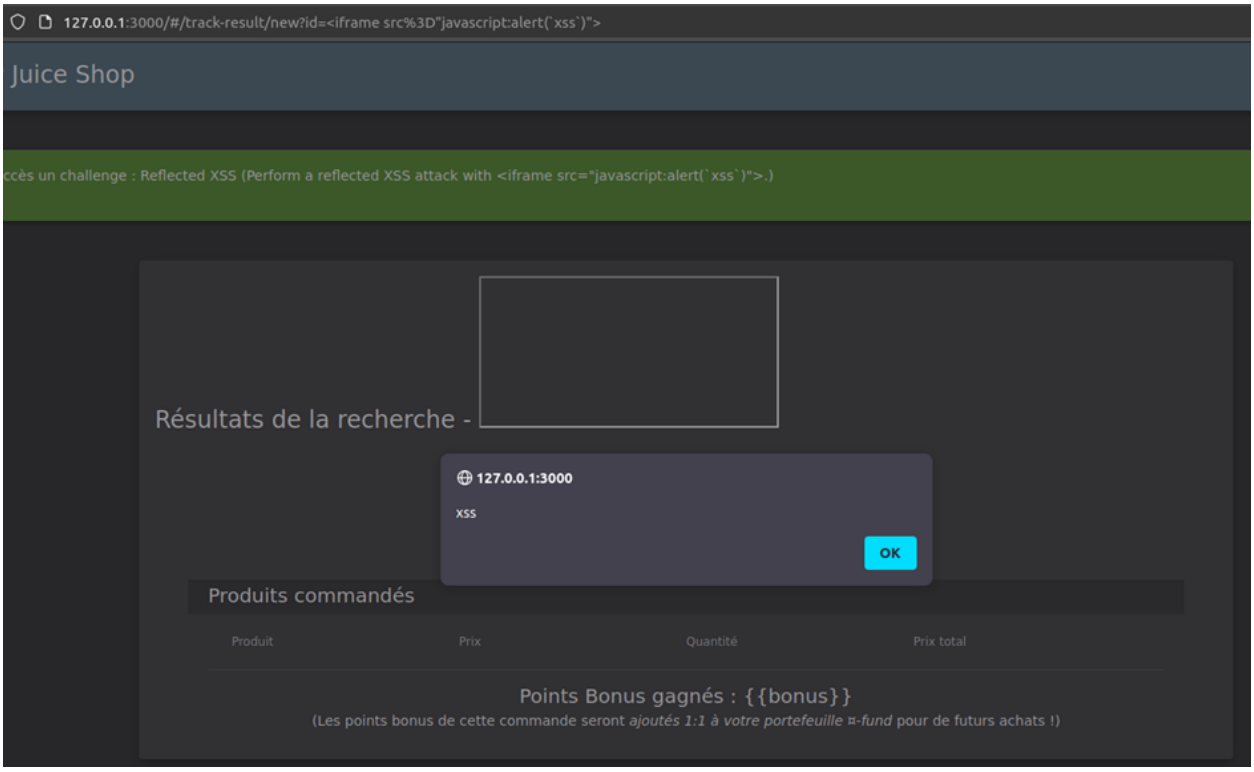
  bash

  Copy code

  http://127.0.0.1:3000/#/track-result?id=<script>alert('xss')</script>

- **Step 4: Inserting the Payload:**
  Navigated to the crafted URL, which resulted in the execution of the injected JavaScript code. The page reflected the id input without proper sanitization, leading to the successful execution of the XSS payload.

  Proof of Exploit:
  Upon visiting the URL with the payload, the script tag was processed, triggering the JavaScript alert, confirming the existence of a reflected XSS vulnerability.



Vulnerability Explanation:

Reflected XSS occurs when an application reflects user input directly in its HTML response without proper validation or sanitization. This allows attackers to inject malicious JavaScript that executes when a user loads the affected page. In this case, the id parameter in the order tracking page was vulnerable, and the injected script executed on load.

Impact:

The reflected XSS vulnerability can lead to various security risks, including:

- **Session Hijacking:** Attackers can steal user session cookies or tokens.
- **Malicious Redirection:** Users can be redirected to malicious sites or phishing pages.
- **Data Exfiltration:** Attackers can extract sensitive data from the application or perform actions on behalf of users.

Remediation:

To secure the application from reflected XSS:

- **Validate and Sanitize Input:** Ensure that all user inputs, especially those reflected in HTML responses, are validated and sanitized to remove or neutralize harmful scripts.
- **Encode Output:** Use proper HTML encoding for any user-supplied data before rendering it on the page. This prevents the browser from interpreting input as executable code.

Vulnerability7: Admin Section

Title: Admin Section
Category: Broken Access Control
Difficulty: ⭐ ⭐ (2/6)

Description:

The "Admin Section" challenge involves accessing a restricted administrative area of a web application.

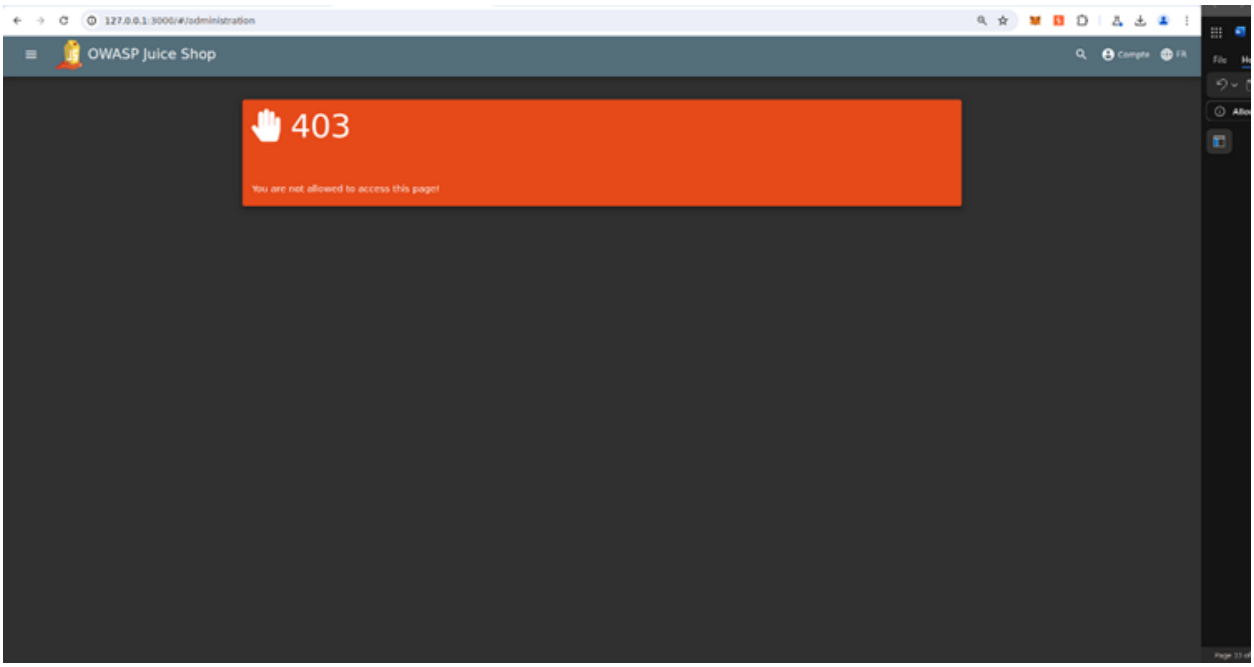
Exploring Application Routes

1. Source Code Analysis:

- **Using the browser's developer tools, the main.js file was examined. This file contained definitions for various application routes (paths and components), including a path for an "administration" section.**
- **The route setup indicated that the administration area could be accessed directly if the correct path was known.**

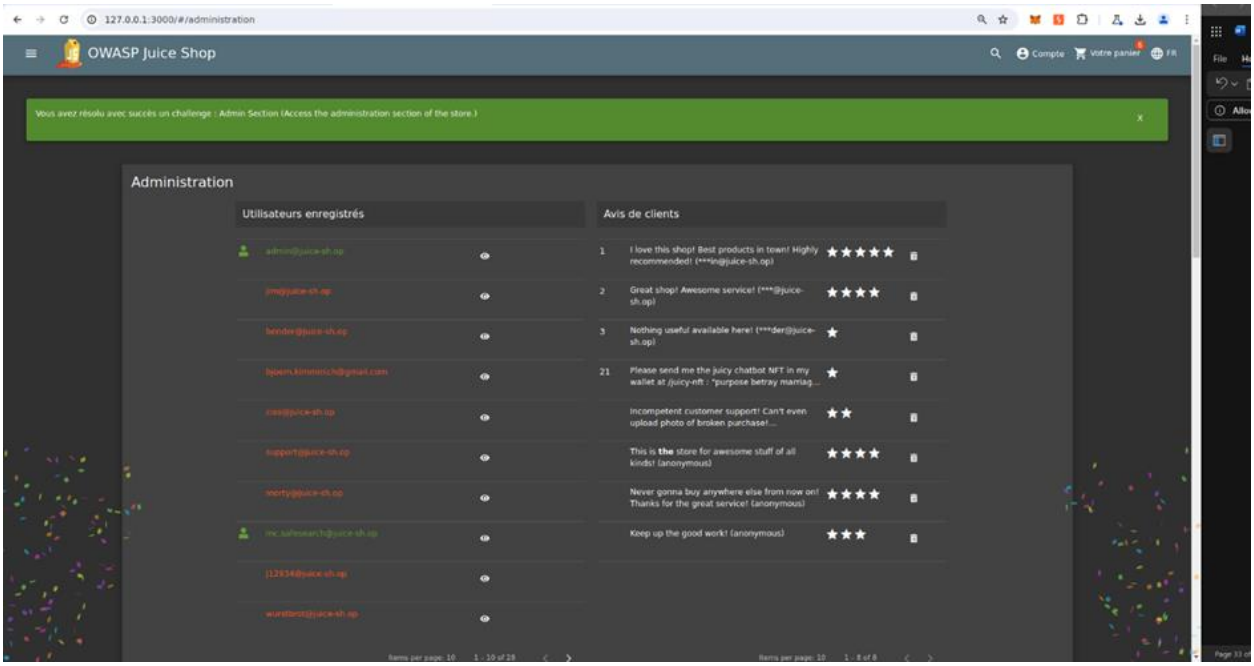
Direct Path Access:

- Accessed the path 127.0.0.1:3000/#/administration directly by entering it in the browser's address bar.
- Initially, this approach led to an access-denied page due to insufficient permissions.



Gaining Administrative Access

3. **Role Elevation**:
    - Logged in with an administrator account credentials.
    - Revisited the /administration path which now successfully displayed the administrative dashboard, demonstrating improper access control that relies solely on the obscurity of the path.



Remediation

Consider putting the following into practice to successfully secure the admin section:
• Appropriate Access Control: Make sure that strong access control procedures are in place that verify and approve users before allowing them to access sensitive areas of the program. For permission confirmation, use server-side checks.

• Concern Separation: As advised, keep the administration features apart from the primary application. The admin panel should ideally be hosted on a separate server or as a stand-alone service that isn't available via the public internet.

• Web Application Firewalls (WAF): Install a WAF to keep an eye on and stop illegal attempts to enter restricted regions.

Vulnerability8: Five-Star Feedback

Title: Five-Star Feedback
Category: Broken Access Control
Difficulty: ⭐ ⭐ (2/6)

Description:

The "Five-Star Feedback" challenge requires to manipulate the review system of a web application by deleting all five-star reviews.

Getting into the Admin Panel
1. Make Use of Administrative Privileges: o Due to a prior challenge, administrative access was obtained, which made it possible to investigate features that ordinary users are unable to access, such as managing user feedback.

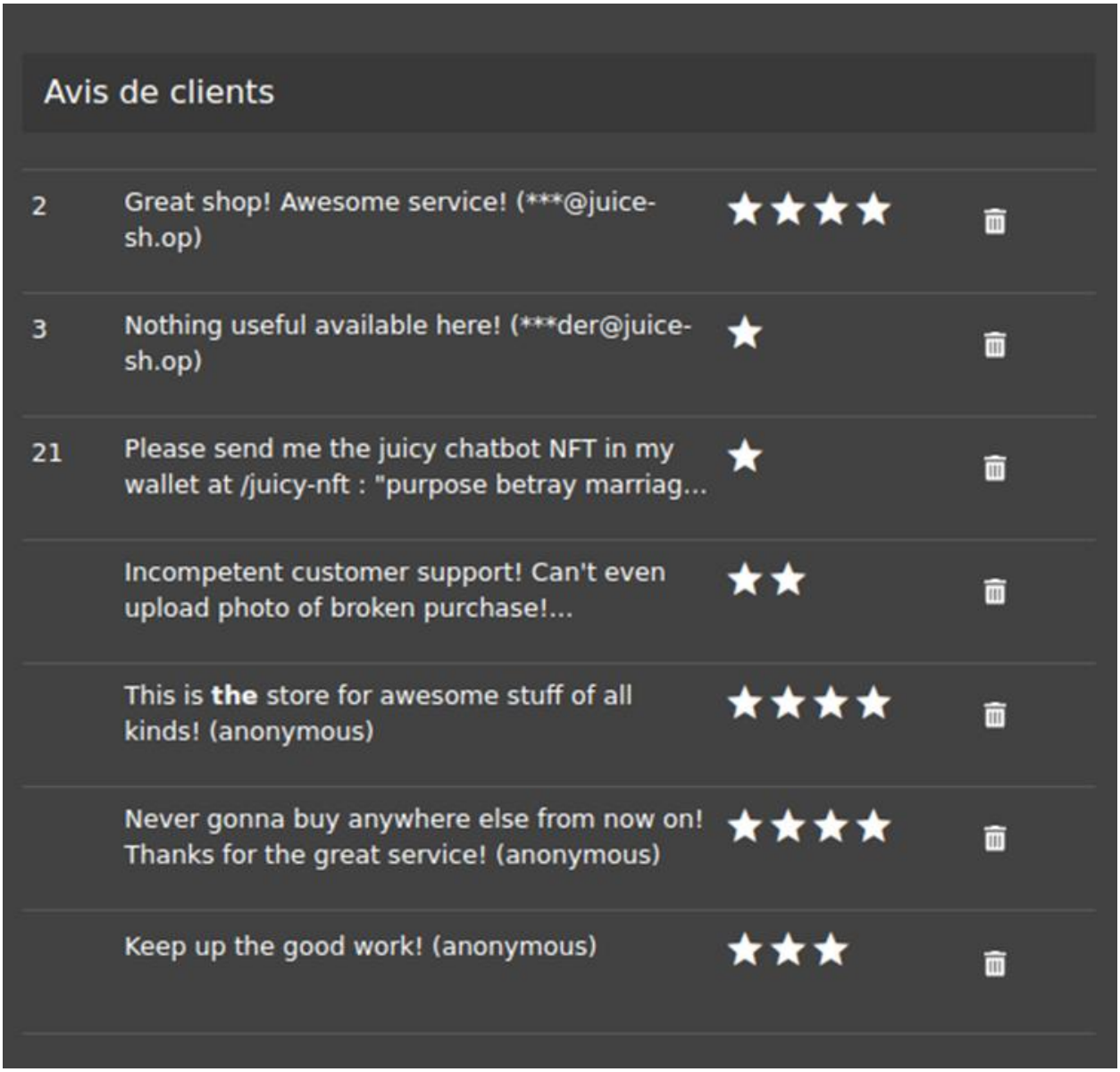
2. Where to Find the Feedback Management Area:

o Selected the area of the admin panel that handles user reviews and feedback.

Eliminating Five-Star Evaluations

3. Find Reviews with Five Stars:

o Star ratings are usually displayed alongside reviews. found every entry that had a five-star rating and was prepared for removal.



Remediation

Audit Trails: Keep a record of every administrative action, particularly those that have an impact on user information or feedback. This facilitates accountability and the tracking of how particular actions were approved or carried out.

Vulnerability9: View Basket

Title: View Basket
Category: Broken Access Control
Difficulty: ⭐ ⭐ (2/6)

Description:

The "View Basket" challenge revolves around exploiting broken access control vulnerabilities within a web application. The specific vulnerability allows users to view the shopping basket details of other users by manipulating user-specific identifiers in web requests.

Recognizing the Risk
Users can view the items in their shopping basket via the online application. But after using Burp Suite to analyze the network traffic, it was found that the request to retrieve the basket details contains a user-specific identifier (such as the basket ID) that can be manipulated.

Actions Made to Address the Issue
Catch the HTTP Request:

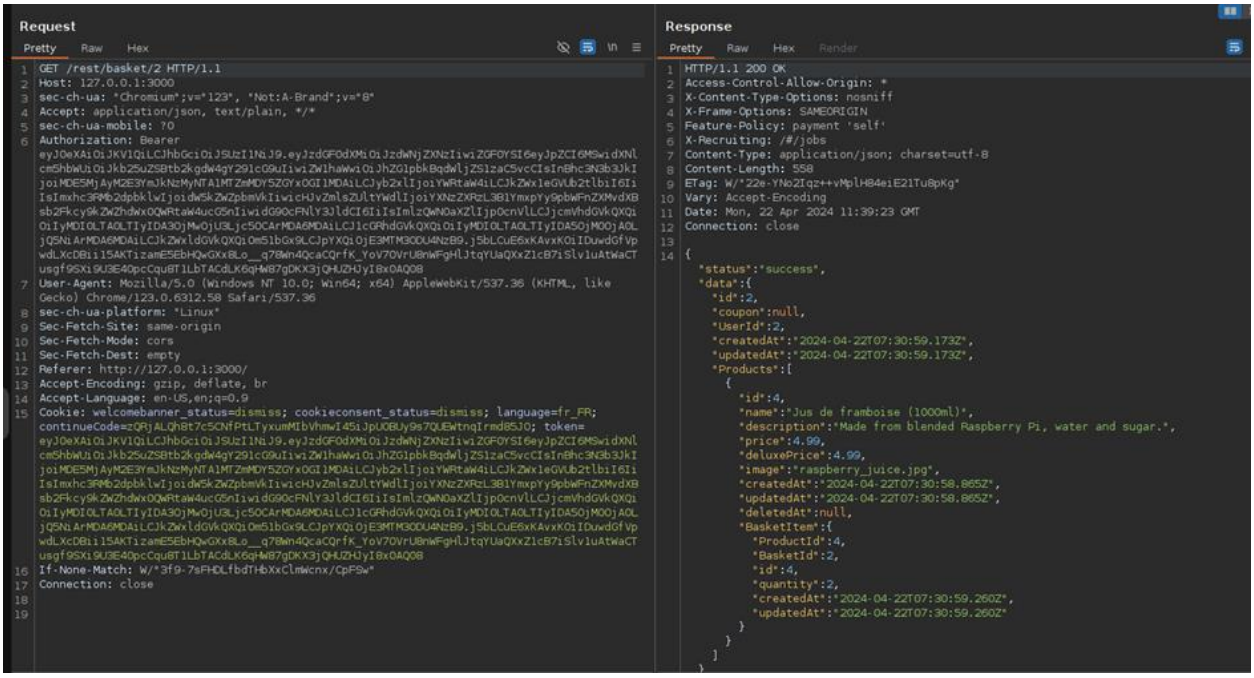
The HTTP request to fetch the user's own basket was intercepted with Burp Suite.
The current user's basket ID is 1, and the request URL is
http://127.0.0.1:3000/rest/basket/1.

```
GET /rest/basket/1 HTTP/1.1
Host: 127.0.0.1:3000
sec-ch-ua: "Chromium";v="123", "Not:A-Brand";v="8"
Accept: application/json, text/plain, */*
sec-ch-ua-mobile: ?0
Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6MSwidXNl
cmShbWUiOiJkb25uZSBtb2kgdW4gY291cG9uIiwiZW1haWwiOiJhZG1pbkBqdWljZS1zaC5vcCIsInBhc3N3b3JkI
joiMDE5MjAyM2E3YmJkNzMyNTA1MTZmMDY5ZGYxOGI1MDAiLCJyb2xlIjoiYWRtaW4iLCJkZWx1eGVUb2tlbiI6Ii
IsImxhc3RMb2dpbklwIjoidW5kZWZpbmVkIiwicHJvZmlsZUltYWdlIjoiYXNzZXRzL3B1YmxpYy9pbWFnZXMvdXB
sb2Fkcy9kZWZhdWx0QWRtaW4ucG5nIiwidG90cFNlY3JldCI6IiIsImlzQWN0aXZlIjpOcnVlLCJjcmVhdGVkQXQi
OiIyMDI0LTA0LTIyIDA3OjMwOjU3Ljc5OCArMDA6MDAiLCJ1cGRhdGVkQXQiOiIyMDI0LTA0LTIyIDA5OjM0OjA0L
jQ5NiArMDA6MDAiLCJkZWxldGVkQXQiOm51bGx9LCJpYXQiOjE3MTM3ODU4NzB9.j5bLCuE6xKAvxKOiIDuwdGfVp
wdLXcDBii15AKTizamE5EbHQwGXx8Lo__q78Wn4QcaCQrfK_YoV7OVrU8nWFgHlJtqYUaQXxZ1cB7iSlv1uAtWaCT
usgf9SXi9U3E4OpcCqu8T1LbTACdLK6qHW87gDKX3jQHUZHJyI8xOAQO8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/123.0.6312.58 Safari/537.36
sec-ch-ua-platform: "Linux"
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
Accept-Language: en-US,en;q=0.9
Cookie: welcomebanner_status=dismiss; cookieconsent_status=dismiss; language=fr_FR;
continueCode=zQRjALQh8t7c5CNfPtLTyxumMIbVhmwI45iJpUOBUy9s7QUEWtnqIrmd85JO; token=
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6MSwidXNl
cmShbWUiOiJkb25uZSBtb2kgdW4gY291cG9uIiwiZW1haWwiOiJhZG1pbkBqdWljZS1zaC5vcCIsInBhc3N3b3JkI
joiMDE5MjAyM2E3YmJkNzMyNTA1MTZmMDY5ZGYxOGI1MDAiLCJyb2xlIjoiYWRtaW4iLCJkZWx1eGVUb2tlbiI6Ii
IsImxhc3RMb2dpbklwIjoidW5kZWZpbmVkIiwicHJvZmlsZUltYWdlIjoiYXNzZXRzL3B1YmxpYy9pbWFnZXMvdXB
sb2Fkcy9kZWZhdWx0QWRtaW4ucG5nIiwidG90cFNlY3JldCI6IiIsImlzQWN0aXZlIjpOcnVlLCJjcmVhdGVkQXQi
OiIyMDI0LTA0LTIyIDA3OjMwOjU3Ljc5OCArMDA6MDAiLCJ1cGRhdGVkQXQiOiIyMDI0LTA0LTIyIDA5OjM0OjA0L
jQ5NiArMDA6MDAiLCJkZWxldGVkQXQiOm51bGx9LCJpYXQiOjE3MTM3ODU4NzB9.j5bLCuE6xKAvxKOiIDuwdGfVp
wdLXcDBii15AKTizamE5EbHQwGXx8Lo__q78Wn4QcaCQrfK_YoV7OVrU8nWFgHlJtqYUaQXxZ1cB7iSlv1uAtWaCT
usgf9SXi9U3E4OpcCqu8T1LbTACdLK6qHW87gDKX3jQHUZHJyI8xOAQO8
If-None-Match: W/"3f9-7sFHDLfbdTHbXxClmWcnx/CpFSw"
Connection: close
```

- We obtain our basket :

Change the Basket ID:

changed the intercepted request's basket ID from 1 to 2 in an effort to view the basket information of another user.



View Results:

- The modified request was forwarded, and the response contained the details of another user's basket, confirming the presence of broken access control.

Remediation

The following actions must be taken into consideration in order to lessen such vulnerabilities in actual applications:

User Session Validation: Verify that the user is authorized to view just their data by comparing each request for sensitive information, such as a shopping basket, to their session.
Using Sturdy Access Control Systems: To impose stringent permissions on data access, use attribute-based access control (ABAC) or role-based access control (RBAC).
Frequent audits of security: To find and address access control vulnerabilities before they may be exploited, conduct routine penetration tests and security audits.
This difficulty emphasizes how crucial strict access control procedures are to preventing unwanted access to private user data.

Category: SQL Injection

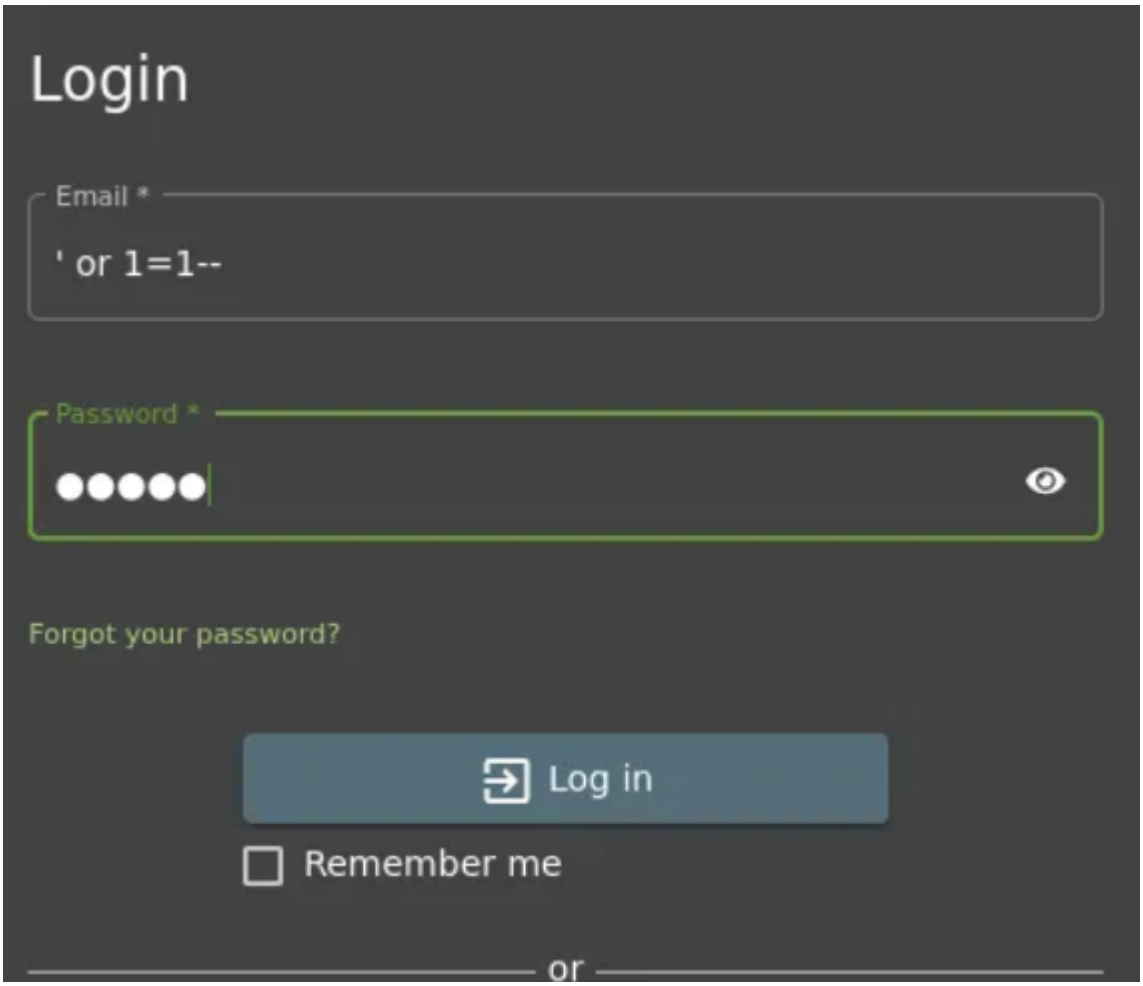
Difficulty Level: ⭐ ⭐ (2/6)

Description:

This challenge requires exploiting an SQL Injection vulnerability within the authentication process of a web application to gain unauthorized access to the administrator's account, it allows an attacker to interfere with the queries an application makes to its database. It typically occurs when user input is improperly sanitized, enabling attackers to insert or "inject" malicious SQL statements into a query.

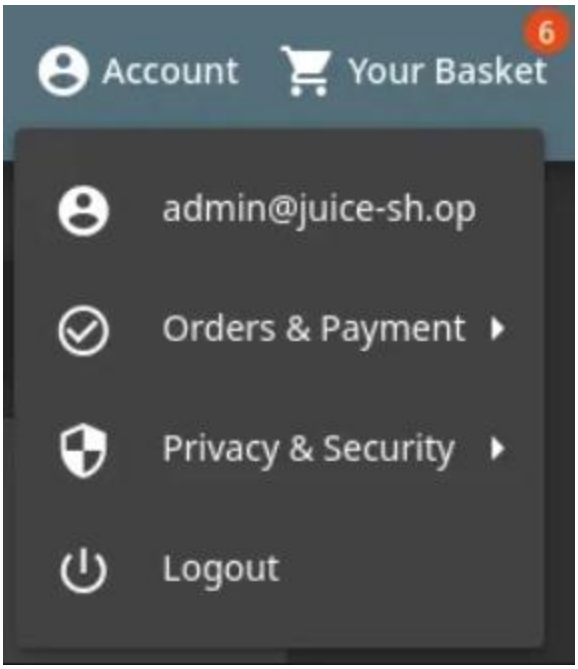
Exploitation:

First found a login page



Modified the email field ' OR 1=1 --, effectively turning the SQL command into a statement that always returns true, bypassing the need for a password.

The modified request was sent to the server. Due to the injected SQL code, the condition OR 1=1 always evaluates as true, allowing unauthorized access to the administrator's account.



Remediation:

- **Input Validation:** Sanitize and validate all user inputs.
- **Prepared Statements**: Use parameterized queries or prepared statements to separate SQL code from data**.**
- **Web Application Firewalls (WAF):** Implement WAFs to detect and block SQL injection attempts.
- **Regular Security Testing:** Conduct penetration testing and code reviews to identify vulnerabilities.

Vulnerability11: Password Strength

Category: Broken Authentication

Difficulty Level: ⭐ ⭐ (2/6)

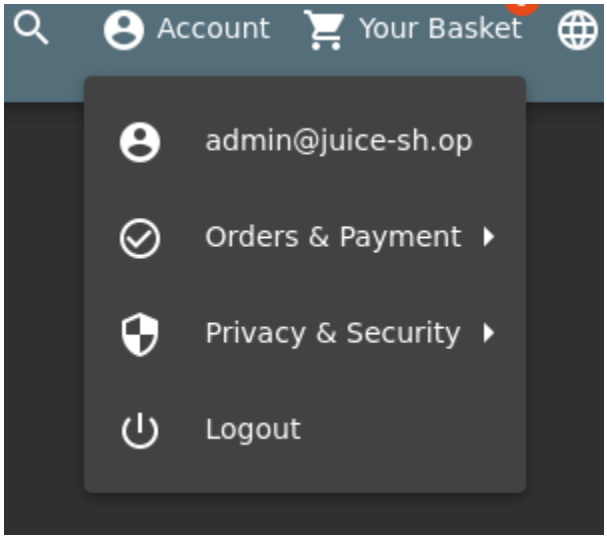
Description:

The "Password Strength" challenge explores vulnerabilities associated with weak password policies, specifically focusing on the ease of brute-forcing the administrator's weak password. This challenge underscores the importance of implementing robust password policies to secure authentication mechanisms.

Exploitation:

Locate Administrator's Email:

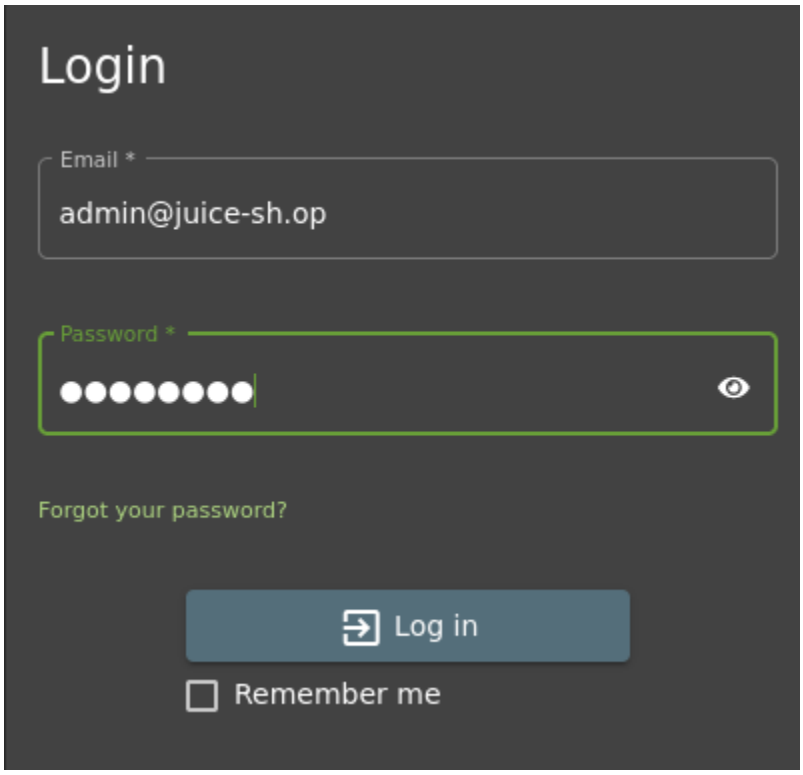
Scanned the website for visible or hidden mentions of the administrator's email. Comments in the website's source code revealed the email address: admin@juice-sh.op.



Then I tired to login with this email and guessed the password of admin

I tired the most common admin passwords, Successfully authenticated using the password: admin123.

We can also make a brute force attack using burp suite to intercept the request.



Remediation:

- **Strong Password Policies**: Enforce a strong password policy that requires a mix of uppercase letters, lowercase letters, numbers, and special characters. Passwords should be at least 8 characters long.
- **Password Complexity and Blacklists:** Implement checks to ensure passwords are not among common passwords or compromised passwords listed in public breaches.
- **Rate Limiting**: Prevent brute force attacks by limiting the number of failed login attempts from a single IP address or account before a cooldown period is enforced.
- **Multi-Factor Authentication (MFA):** Add an additional layer of security by requiring a second form of verification beyond just a password.

Vulnerability12: Login Bender

Category: Injection Flaws
Difficulty Level: ⭐ ⭐ ⭐ (3/6)

Description:

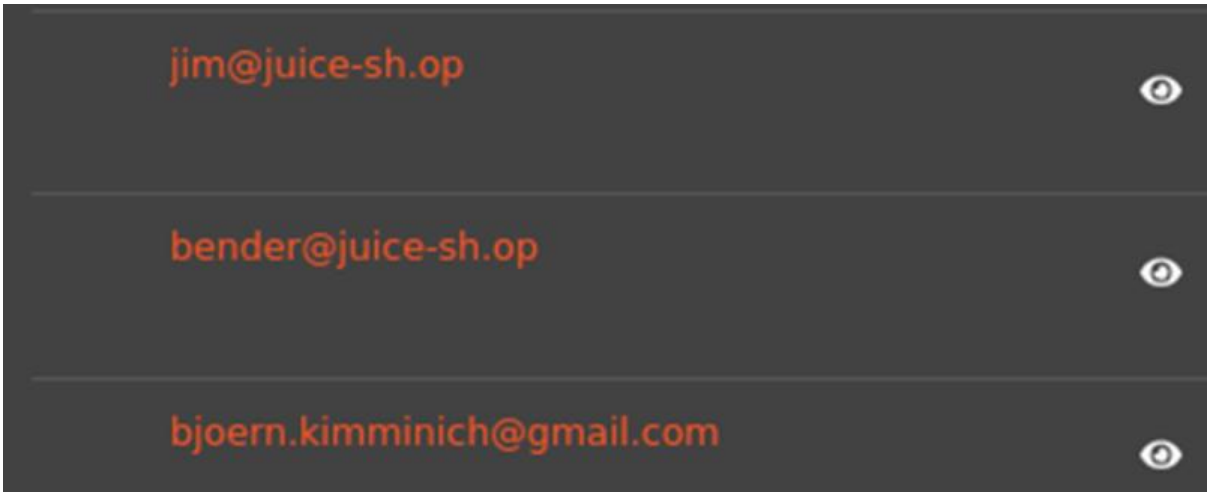
This challenge focuses on exploiting an SQL injection vulnerability to bypass authentication mechanisms and gain unauthorized access to a user's account, specifically targeting the account associated with "Bender" in the application.

Exploitation:

Locating Target Information

1. Identify Bender's Email:

- Logged into the application as an admin to access administrative functionalities.
- Located Bender's email (bender@juice-sh.op) within the admin panel :

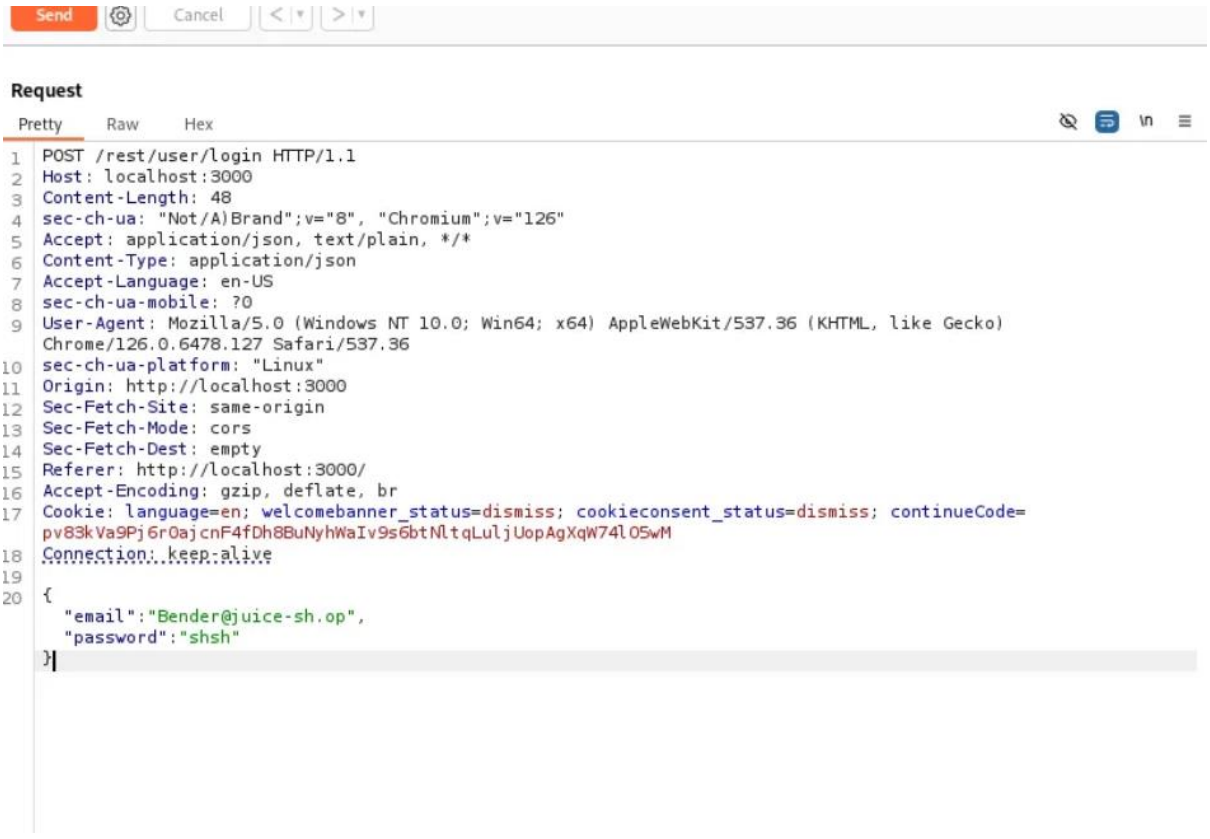


Performing SQL Injection

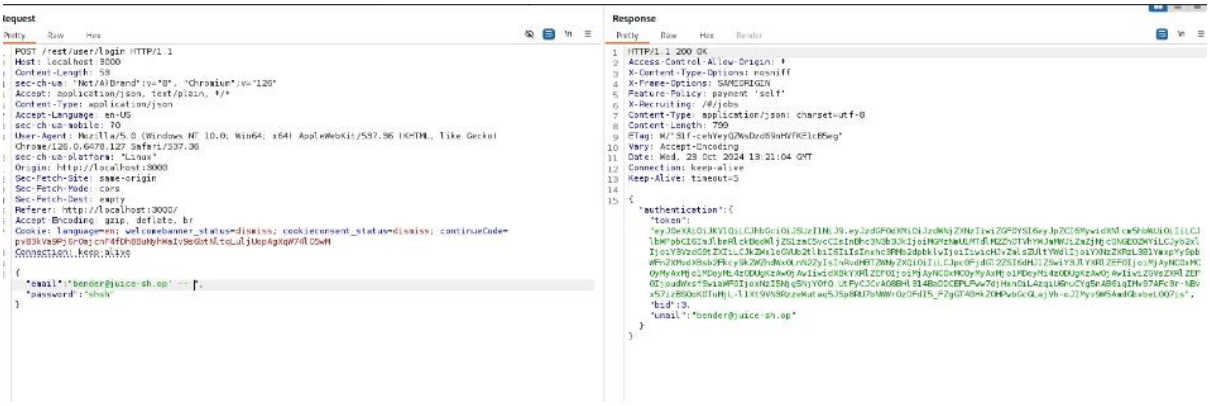
2. Attempt to Login as Bender:

- Navigated to the login page of the application.
- Entered Bender's email in the email field.

I used burp suite to intercept the login request



3. I changed the email field with " bender@juice-sh.op ' – ", this payload attempts to manipulate the SQL query behind the login form by closing off the email string (with a single quote),adding a condition that commet out the remainder of the SQL command with --.



4. Observe Authentication Bypass:

- With the payload submitted, the SQL query altered by the injection would incorrectly authenticate the session as Bender without requiring a password.
- Successfully logged in as Bender, demonstrating the exploitation of the SQL injection vulnerability.

This challenge was resolved by exploiting a classic SQL injection vulnerability in the login form's email field. The specific vulnerability allowed for manipulation of the underlying SQL query, enabling unauthorized access to Bender's account by making comment to the remainder of the query and bypassing password verification.

Remediation:

- **Use Prepared Statements**: Implement prepared statements with parameterized queries to handle SQL commands. This practice prevents SQL injection by separating SQL logic from data.
- **Employ Proper Input Validation**: Ensure that all user inputs are properly validated and sanitized to prevent malicious data from affecting SQL commands.

Vulnerability13: Bjoern's Favorite Pet

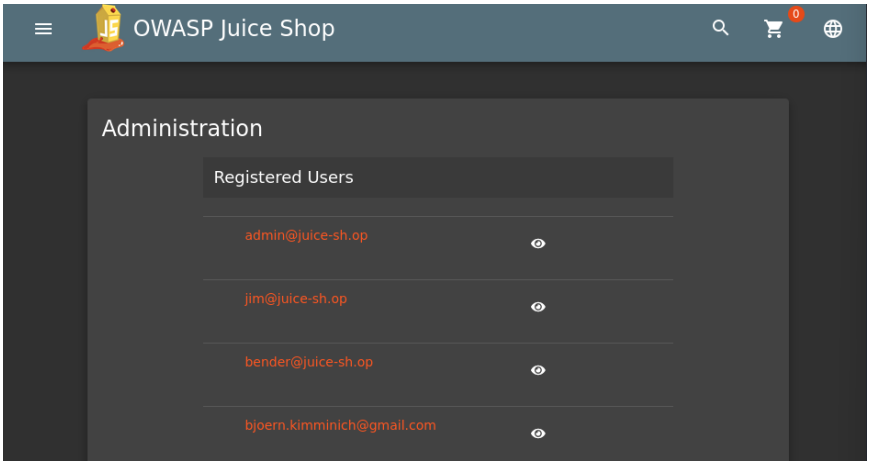
Title: Bjoern's Favorite Pet
Category: OSINT (Open-Source Intelligence)
Difficulty: ⭐ ⭐ ⭐ (3/6)

Description:

The "Bjoern's Favorite Pet" challenge involves using open-source intelligence to discover personal information related to Bjoern, specifically identifying the name of his favorite pet, which is used as the answer to a security question.

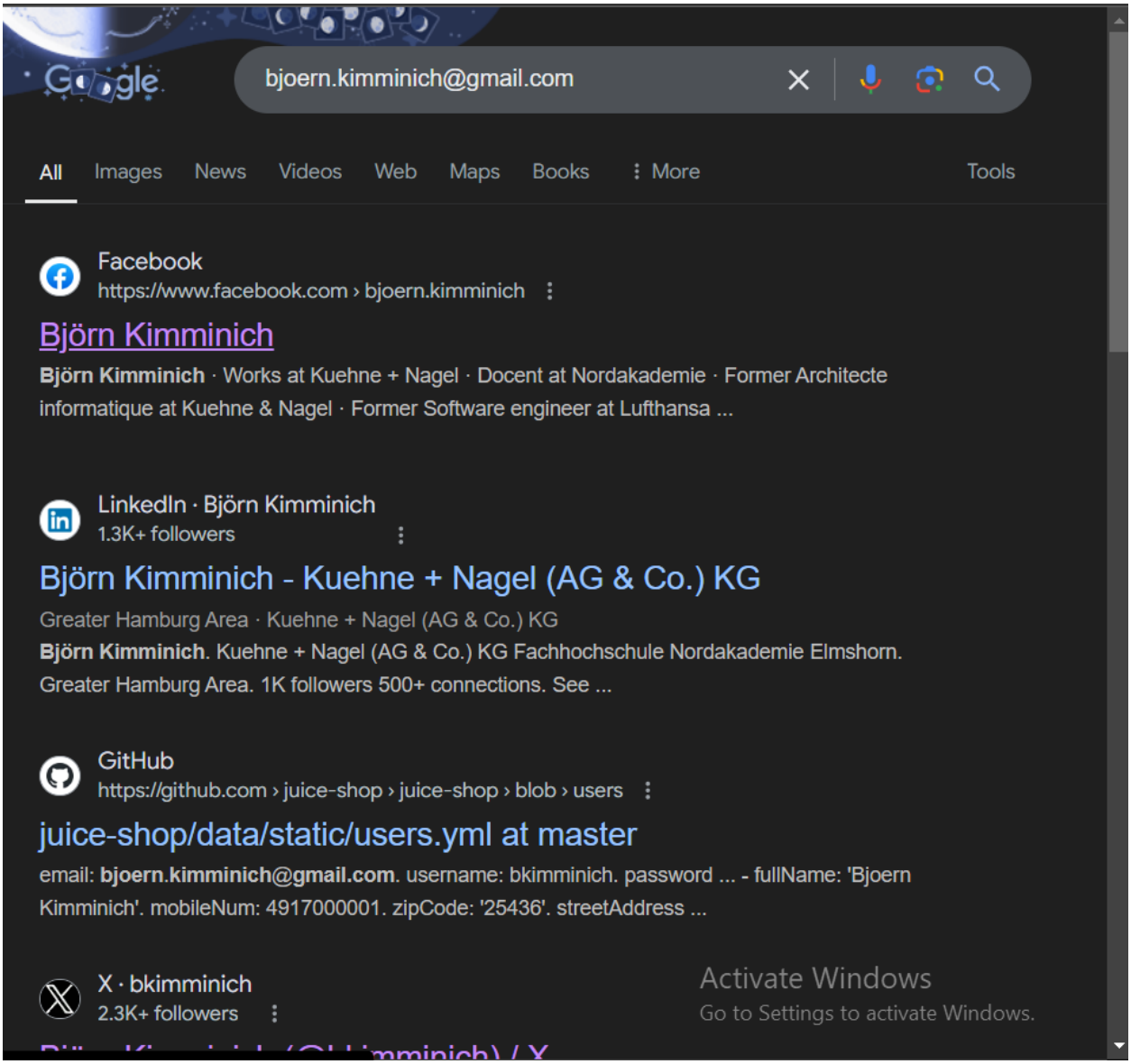
Gathering Initial Information

1. **Identifying Bjoern's Email**:
    - Accessed the administrative panel of the application to obtain Bjoern's email address, which is bjoern.kimminich@gmail.com.

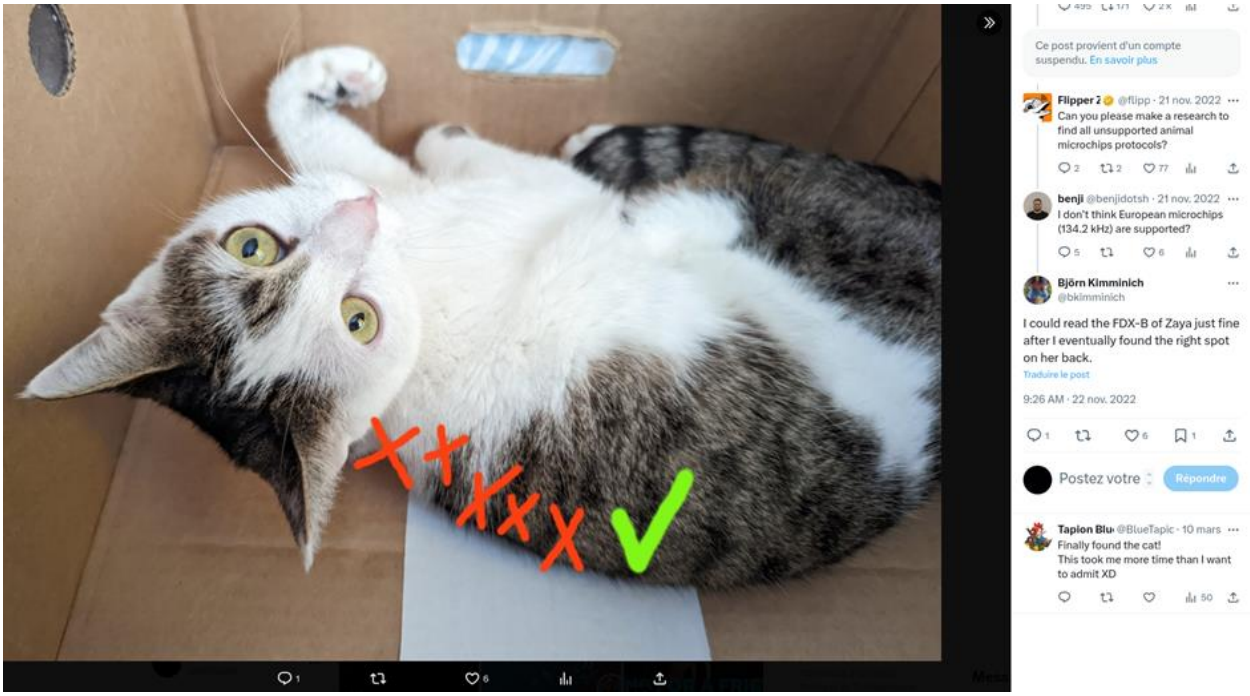


Exploring Social Media:

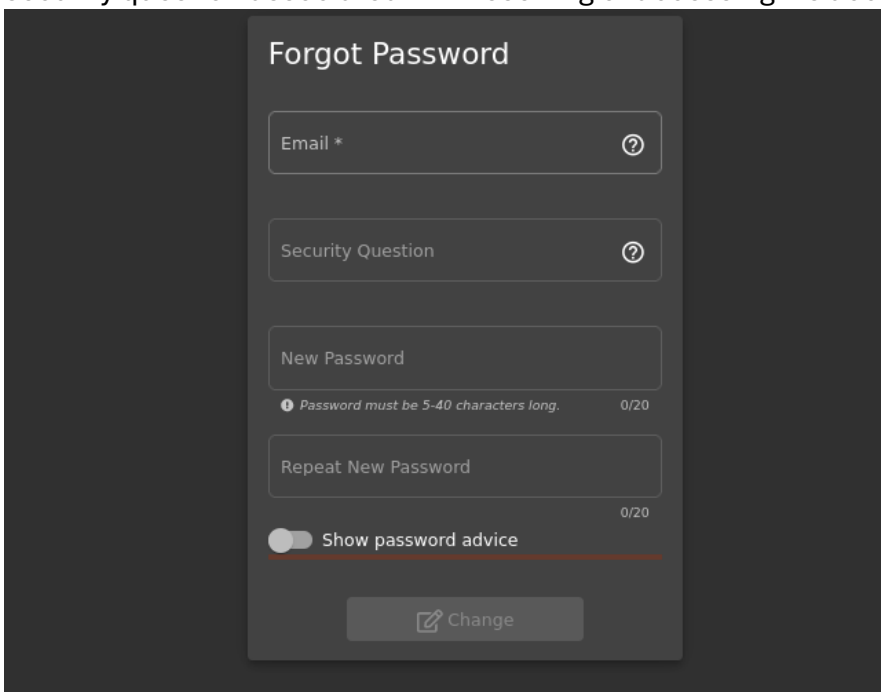
- **Used Bjoern's email address to search for his social media profiles. Located his Twitter account.**

- 

on threads, found a post by Bjoern featuring a picture of his cat named Zaya.



Leveraging the Information

3. **Answering the Security Question**:
   - With the name of Bjoern's cat discovered, used "Zaya" as the answer to the security question associated with resetting or accessing his account.
   - 

Remediation:

To mitigate similar vulnerabilities in real-world scenarios:

Use of Non-Personal Security Questions: Promote or mandate the use of security questions that don't depend on details that are readily guessed or acquired from publicly available sources. When a user wishes to change his password, it is preferable to send an email rather than utilize a security question.

Privacy Settings: Encourage users to use social media platforms' privacy settings to limit who can view their postings, particularly when disclosing private or sensitive information.

Vulnerability14: Manipulate Basket

Title: Manipulate Basket
Category: Broken Access Control
Difficulty: ⭐ ⭐ ⭐ (3/6)

Description:

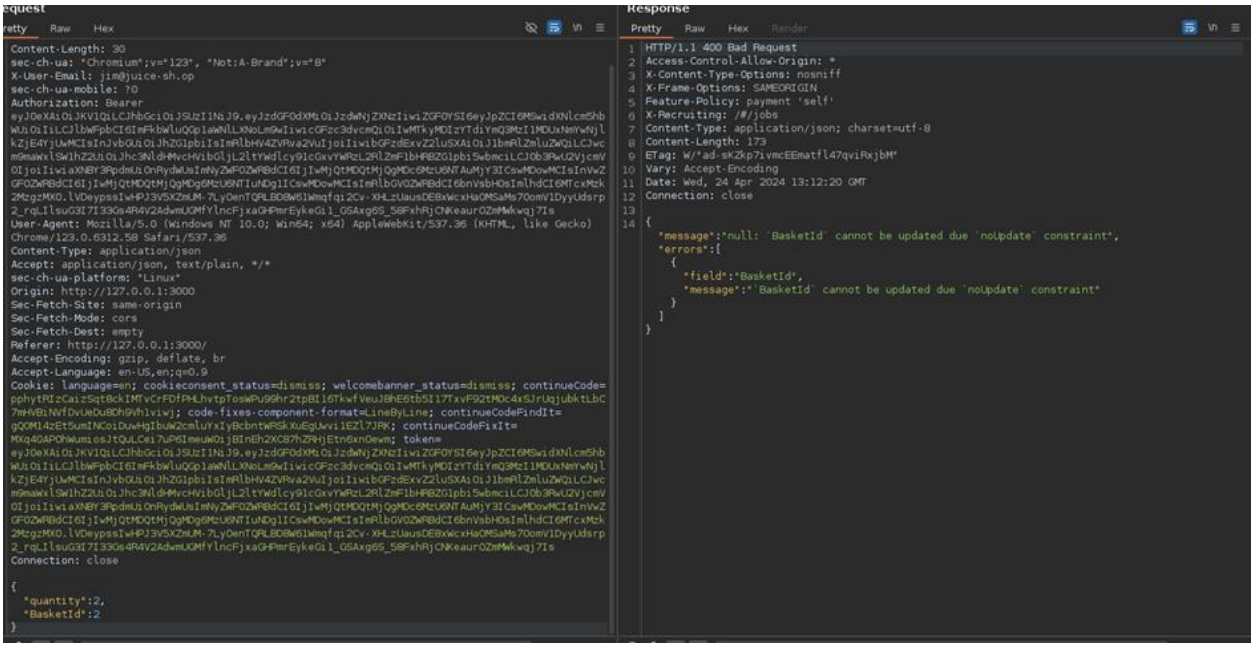
The "Manipulate Basket" challenge involves adding an item to another user's shopping basket by exploiting potential insecure direct object references (IDOR) vulnerabilities in the application.

Examining the Basket's Features
First Testing:
To view the arguments and structure of the HTTP request, add a product to your personal basket.
Note that every basket operation (add, change) has a unique basket ID that is associated with a user.

Recognizing Request Manipulation Vulnerability:
Use Burp Suite to intercept the request when a product is added to the basket.
One possible vector for IDOR is indicated by the inclusion of the basket ID (BasketId) in the request body.
Taking Advantage of the IDOR Vulnerability

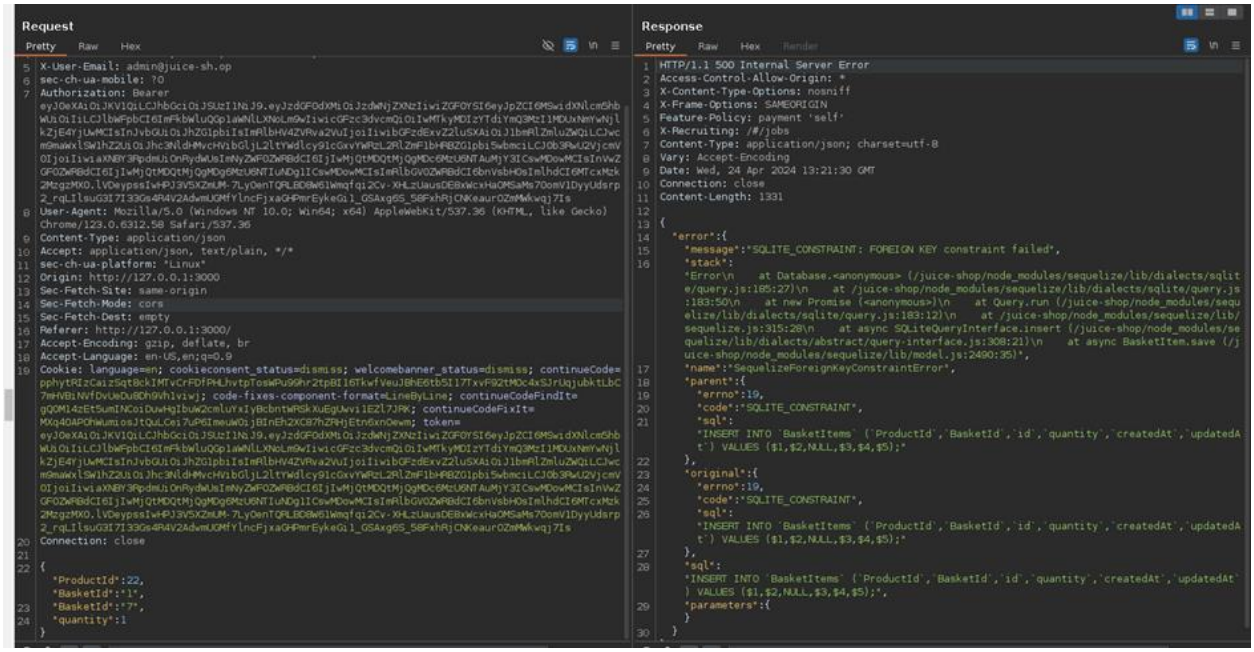
Changes to the Basket ID:

To check for direct object reference problems, I first attempted to replace the BasketId in the intercepted request with the basket ID of another user. However, I run into access control checks that prohibit adding unowned items to a basket.

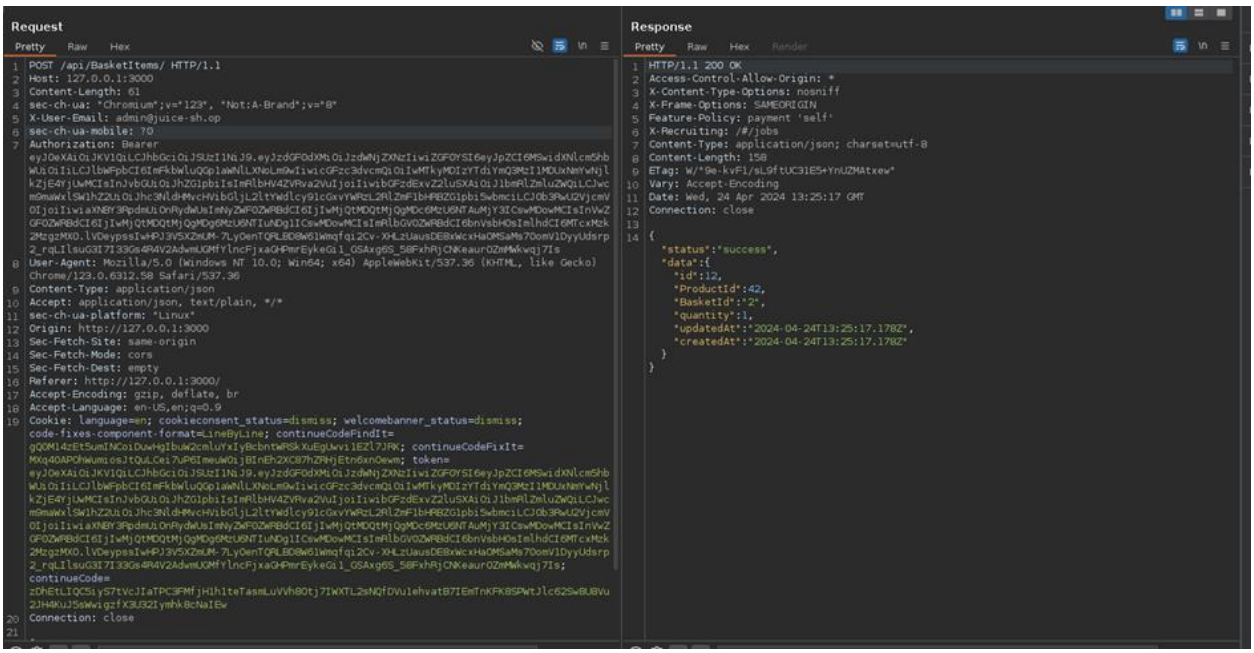


Bypassing the Security Check

4. **Double Parameter Injection:**
    - **Attempt to bypass the security mechanism by duplicating the BasketId parameter in the request body (overload).**
    - **First encountered an error, but it means that something is happening :**



• After continuing in this manner, I discovered that the server successfully adds the item to another user's basket by using the second BasketId for the transaction and processing only the first BasketId for the security check:



Verifying the Exploit
Verification: Submit the altered request and get a reply that works.
Check the targeted user's basket to see if the item was successfully added.

Remediation:

To stop these kinds of flaws in practical applications:

Assure Appropriate Parameter Handling: Servers ought to be built with the ability to safely manage unexpected, duplicate, or out-of-order parameters.

Put Strong Access Controls in Place: Make sure that all sensitive activities confirm that the user has the authorization to utilize the resource in question.

Vulnerability15: Privacy Policy Inspection Challenge

Category: Security through Obscurity
Difficulty Level: ⭐ ⭐ ⭐ (3/6)

Description:

The "Privacy Policy Inspection" challenge tests the ability to thoroughly inspect a privacy policy document and uncover hidden elements or messages. The challenge illustrates the concept of "security through obscurity," where information is concealed rather than properly secured.

Steps Taken:

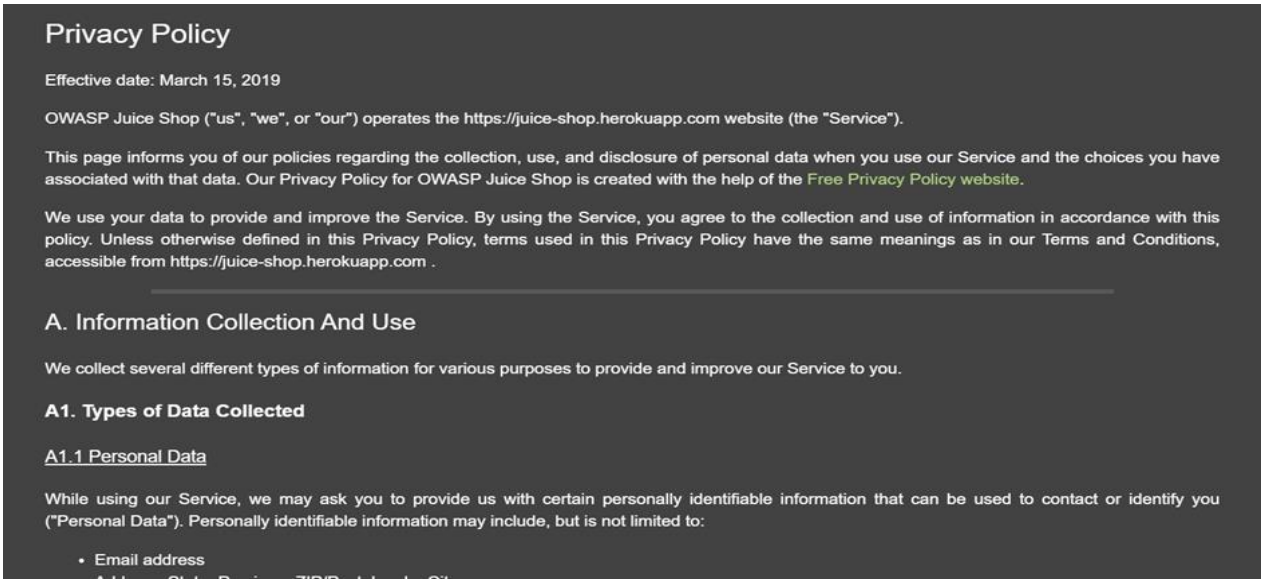
- **Step 1: Locating the Privacy Policy:**
  Used a website crawler (e.g., Burp Suite) to map the site's structure and discover the URL of the privacy policy page:

bash

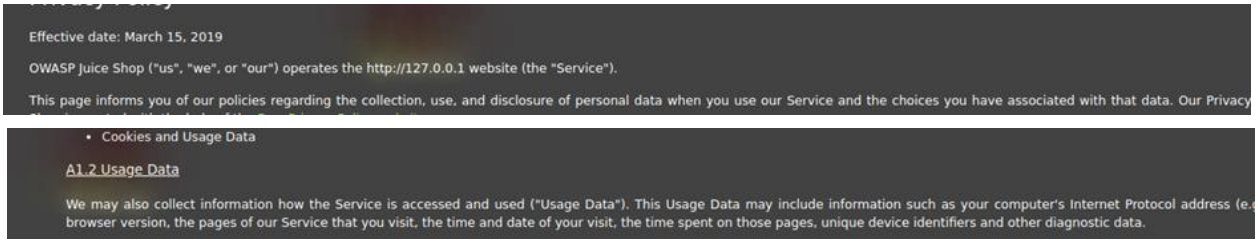
Copy code

http://127.0.0.1:3000/#/privacy-security/privacy-policy

This URL led to the privacy policy document, which served as the target of the challenge.



- **Step 2: Examining the Document:**
  Performed a detailed review of the privacy policy, paying attention to unusual formatting, hidden links, or out-of-place wording that could hint at concealed elements.
- **Step 3: Discovering Hidden Elements:**
  Noticed that hovering over certain sections of the document triggered pop-up overlays or revealed hidden messages.



- These obscured messages or words seemed to provide clues leading to another URL or resource.

- **Step 4: Following Clues:**
  After piecing together the hidden information, the error page revealed the following URL path:

bash

Copy code

/juice-shop/frontend/dist/frontend/assets/private/thank-you.jpg

Navigating to this path led to hidden content, confirming that the challenge was successfully completed.

2. **Proof of Exploit:**
   Accessing the hidden content indicated that the privacy policy had been thoroughly inspected, and all obscured messages had been uncovered.

Vulnerability Explanation:

This challenge explores "security through obscurity," a flawed approach where sensitive information is hidden rather than properly secured. While obscuring information may prevent casual discovery, it provides little real security, as anyone with the right tools or persistence can uncover it.

Impact:

Although this challenge does not present a direct security risk, it highlights the shortcomings of relying on obscurity as a security mechanism. Sensitive information should always be properly protected using robust security measures, rather than concealed within documents or structures.

Remediation:

- **Avoid Security through Obscurity:** Sensitive information should be protected using proper security mechanisms, such as access controls and encryption, rather than being hidden in documents.
- **Implement Comprehensive Security Practices:** Ensure that security is based on sound principles and not simply on hiding information. Use encryption, authentication, and access control to protect data effectively.

Vulnerability16: Upload Size

Category: Improper Input Validation
Difficulty Level: ⭐ ⭐ ⭐ (3/6)

Description:

The "Upload Size" challenge requires uploading a file larger than 100 KB, which is beyond the size limit enforced by the application's client-side validation.

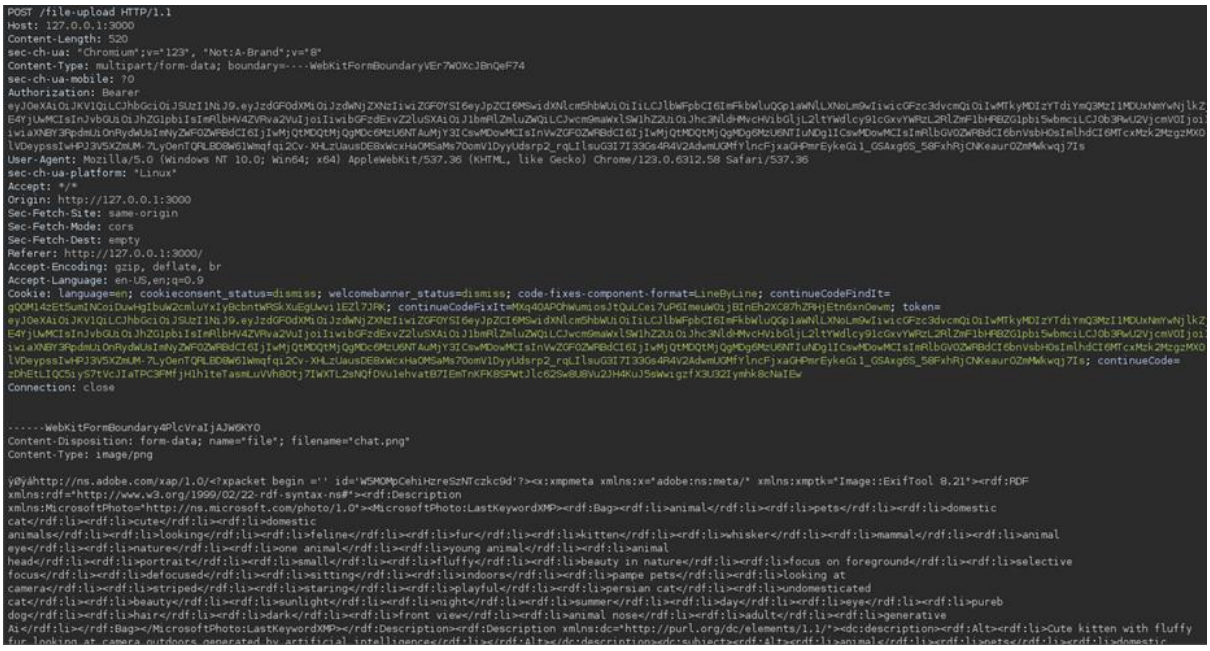
Exploitation:

1. **Testing File Upload:**

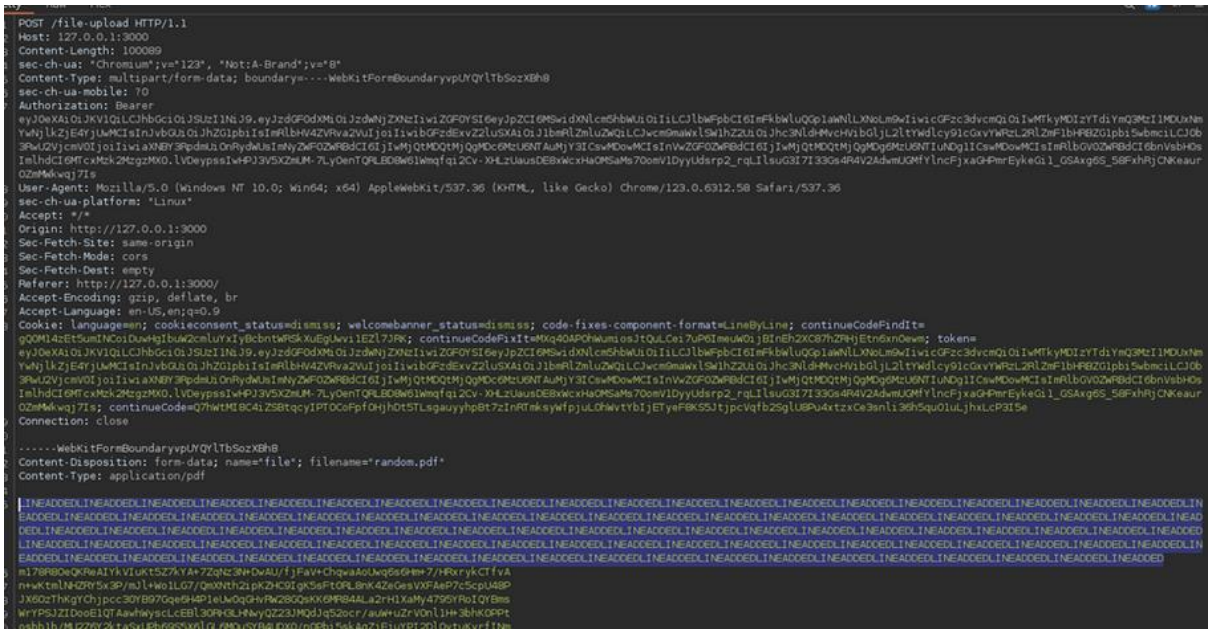
- Attempt to upload a file larger than 100 KB directly through the profile image upload functionality.
- Encounter an error: MulterError: File too large, indicating server-side enforcement of the file size limit.

2. **Exploring Alternative Upload Points**:
    - Redirect focus to the complaint page as hinted, where file uploads are also supported.
    - Discover that this page only superficially restricts file sizes through client-side JavaScript, making it a potential vector for bypassing the restriction.
3. **Creating an Edge Case File**:
    - Generate a file just under the 100 KB limit using the linux command: base64 /dev/urandom | head -c 99900 > random.pdf.
4. **Modifying the Request**:

- Upload the random.pdf and intercept the HTTP request using Burp Suite



- Modify the intercepted request by duplicating the content within the file payload to push the size over 100 KB.



Bypassing the Size Limit

5. Resend the Modified Request:
    - Send the modified request with the now larger file size.
    - Successfully bypass the server-side file size check, as indicated by the successful upload response from the server.

The challenge was resolved by exploiting a weakness in the server's file size checking mechanism, where only initial file sizes are validated client-side, and server-side checks can be bypassed through request modification. This highlights a common security oversight where server-side validations do not robustly re-check parameters that could be tampered with after client-side checks.

Remediation:

- **Implement Robust Server-Side Validation**: Ensure that all input validations, including file sizes, are re-checked on the server after being submitted.
- **Limit Input Modification**: Use multipart/form-data parsing libraries that strictly enforce total payload sizes, rather than relying on client-side enforcement.

Vulnerability17: Forged Feedback.

Category: Broken Access Control

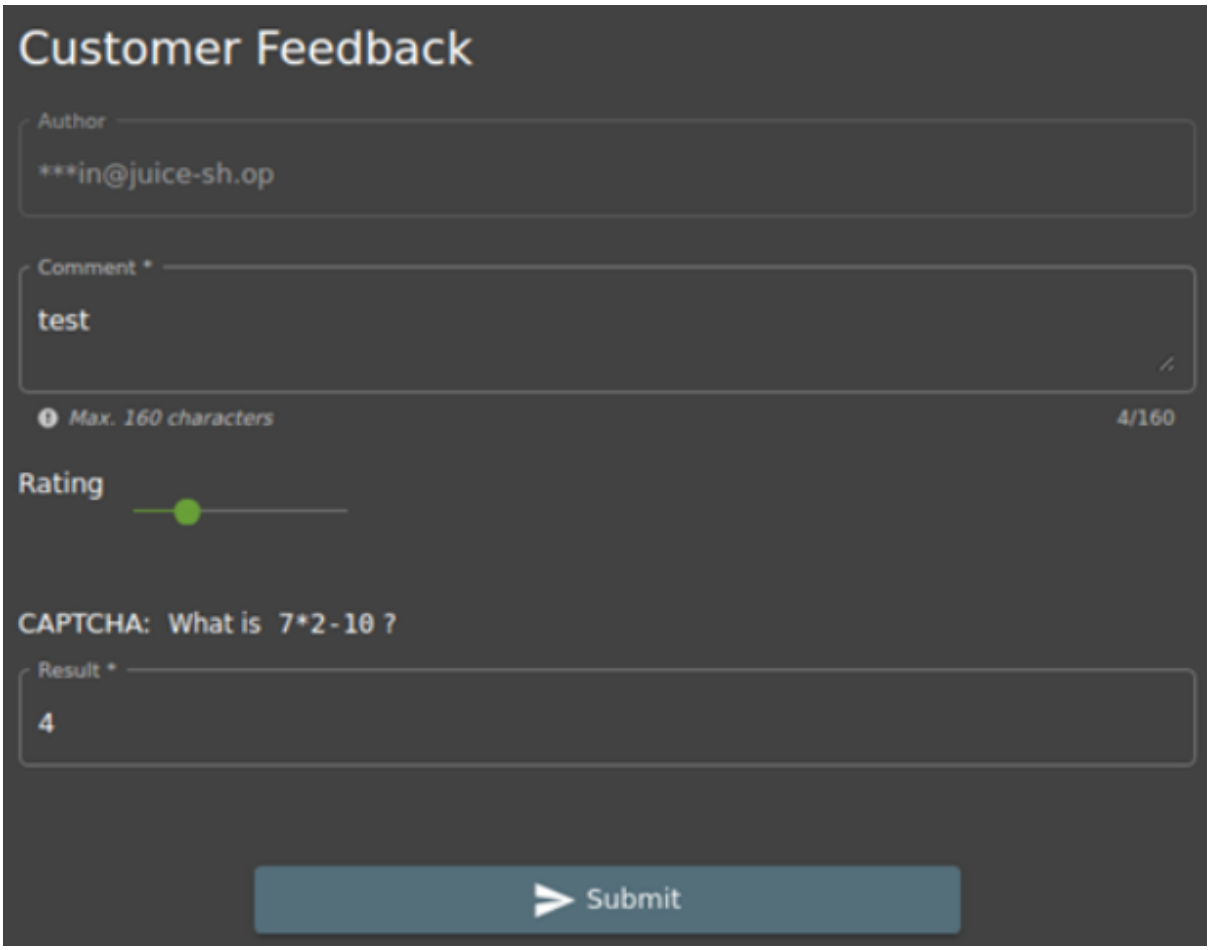
Difficulty Level: ⭐ ⭐ ⭐ (3/6)

Description:

This challenge involves exploiting insufficient access controls to post feedback under another user's name, illustrating vulnerabilities related to user identity management within a web application.

Exploitation:

1. **Submit Feedback**: Initially, submit feedback normally through the application's interface to understand how the feedback submission process works.

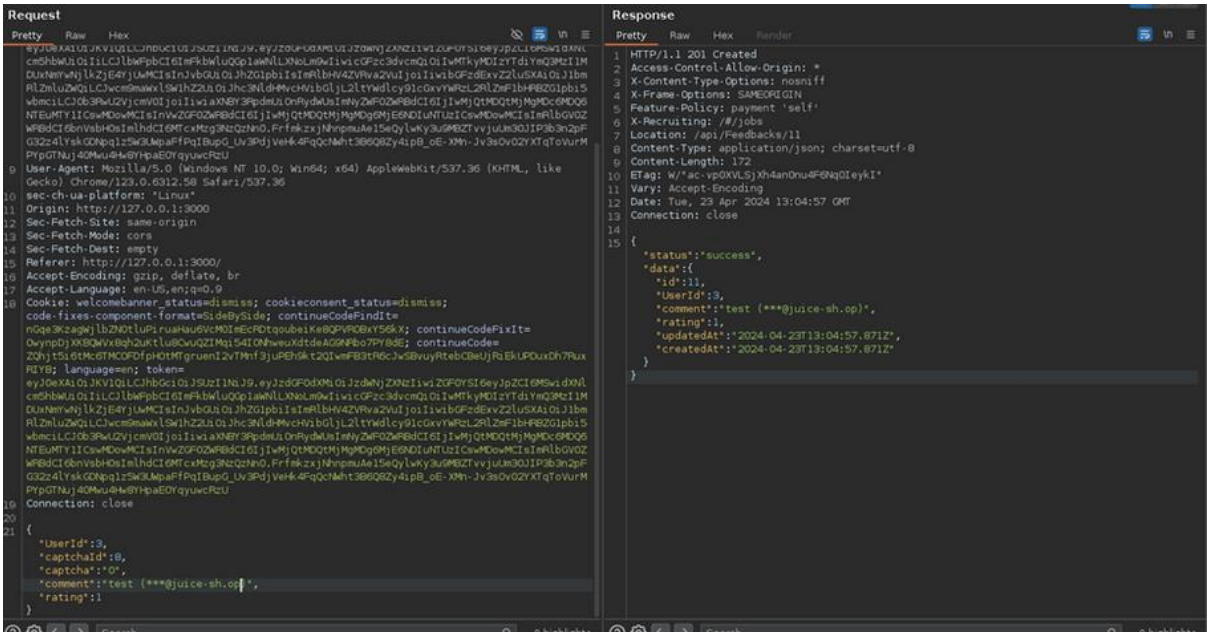


2. **Intercept the Request:** Using Burp Suite, capture the HTTP request sent when feedback is submitted. Analyze the request to understand its structure and the parameters it includes.

Firstly→ Identify User Identification Mechanism: Note that the feedback submission request includes a user identifier, likely an "user_id" or similar parameter, which the application uses to attribute the feedback to a specific user.

Secondly→ Manipulate User Identifier:

- Modify the "user_id" parameter in the intercepted request to the ID of another user, preferably a user with higher privileges like an admin, to test if the application enforces proper authorization checks.



Resend the modified request to see if the feedback gets posted under the changed user ID.

3. **Verify the Outcome**: Check the application to confirm whether the feedback appears under the other user's profile or feedback history.

The challenge was successfully resolved by manipulating the user identifier in the feedback submission process. This was possible because the server failed to validate whether the user submitting the feedback was the same as the user ID specified in the request. This type of vulnerability is indicative of broken access control mechanisms where the application does not adequately verify the user's identity or permissions before performing actions on their behalf.

Remediation:

- **Enhanced Server-Side Validation**: Ensure that all sensitive actions, such as posting feedback, include server-side checks to confirm that the user associated with the session is the same as the user the action is being performed for.

- **Use Session Management**: Implement secure session management practices that map session IDs to user IDs securely. Actions should be authorized based on session ownership rather than relying on user-provided data like user IDs in the request.

Methodology

Assessment Tools Selection

The following tools were used during the assessment:

- Web Browser: To navigate the Juice Shop application and test its core functionalities.
- Burp Suite: For intercepting, manipulating, and analyzing HTTP traffic.
- OWASP ZAP: Automated scanning for vulnerabilities.
- Fuzzer: To test inputs for identifying hidden vulnerabilities.
- JWT.IO: To decode and analyze JSON Web Tokens (JWTs).
- SQLMap: For automating the detection and exploitation of SQL injection flaws.

Assessment Methodology Detail

The methodology followed these key steps:

1. **Initial Analysis:**
    - Reviewed the application's functionality and structure.
    - Identified potential entry points and vectors for attacks.
    - Inspected publicly available information and third-party libraries for potential vulnerabilities.
2. **Testing for Vulnerabilities:**
    - Authentication Flaws: Tested login mechanisms for weak credentials, session fixation, and brute force.
    - Injection Attacks: Explored SQL injections, cross-site scripting (XSS), and command injections.

- Session and Token Security: Analyzed JWTs and sessions for security flaws, including token manipulation and session hijacking.
- Input Validation Issues: Fuzzed input fields and parameters to detect potential failures in validation and sanitization.
- File Upload and Logic Manipulation: Exploited file upload and manipulated the application's logic to simulate real-world attacks.

3. Documentation of Findings:

- Each vulnerability was recorded with detailed explanations of the method used, the risk to the business, and the impact.
- Provided evidence for the issues, including screenshots and logs from the exploitation process.

4. Recommendations:

- Suggested specific remediation strategies for each vulnerability, prioritizing critical issues.
- Advised on the implementation of web application firewalls (WAF), secure coding practices, and regular security audits.