



SECURITY ASSESSMENT

Juice Shop Vulnerabilities Assessment Report

Submitted to: Development Team
Security Analysts: Team 2

Date of Testing: 10/7/2024
Date of Report Delivery: 10/1/2024

Team members:

- Hatem Abdelfatah (Team Coordinator) – hatemabdefatah53@gmail.com
- Amr Khaled – ak4162957@gmail.com
- Yehia Mahmoud – ym5555940@gmail.com
- Kareem Mostafa – eng.kareem.mostafa88@gmail.com

Table of Contents

Contents

- SECURITY ENGAGEMENT SUMMARY 3**
 - ENGAGEMENT OVERVIEW 3
 - SCOPE..... 3
 - EXECUTIVE RISK ANALYSIS..... 3
 - EXECUTIVE RECOMMENDATION 3
- SIGNIFICANT VULNERABILITY SUMMARY 4**
 - High Risk Vulnerabilities 4
 - Medium Risk Vulnerabilities..... 4
 - Low Risk Vulnerabilities 4
- SIGNIFICANT VULNERABILITY DETAIL 5**
 - ASSESSMENT TOOLSET SELECTION..... 7
 - ASSESSMENT METHODOLOGY DETAIL 8

Security Engagement Summary

Engagement Overview

The development team at Dojuicer requested a vulnerability assessment of a legacy web application to identify potential security risks. The goal was to understand the security posture of the application and propose mitigations to enhance security and reduce risk.

Scope

The assessment focused on the legacy web application, emphasizing vulnerabilities that could compromise data integrity and continuity. This scope was appropriate given the critical nature of the application in the organization's operations.

Executive Risk Analysis

The overall risk associated with the identified vulnerabilities during this engagement is **High**. The assessment revealed multiple significant vulnerabilities, including DOM XSS, Broken Access Control, and Sensitive Data Exposure. These vulnerabilities, if exploited, could lead to severe impacts on the organization's data security and user trust.

Executive Recommendation

Remediation efforts are warranted to address the identified vulnerabilities. Prioritization should be given to DOM XSS and Broken Access Control vulnerabilities due to their potential for widespread exploitation. Implementing robust input validation, access controls, and regular security audits is essential to mitigate these risks effectively.

Significant Vulnerability Summary

High Risk Vulnerabilities

- DOM XSS
- SSRF (Server-Side Request Forgery)
- Login Admin (SQL Injection)

Medium Risk Vulnerabilities

- CSRF (Cross-Site Request Forgery)
- Broken Access Control (Web3 Sandbox)

Low Risk Vulnerabilities

- Sensitive Data Exposure (Forgotten Sales Backup)
- Error Handling

Significant Vulnerability Detail

DOM XSS

- **Risk Level:** High
- **Vulnerability Detail:** The application was susceptible to DOM-based XSS through an unfiltered input field. The exploit involved injecting JavaScript into an iframe source attribute.
- **Impact:** If exploited, users could be subjected to malicious script execution, risking data integrity and user trust.
- **Remediation:** To prevent DOM-based XSS attacks like the one demonstrated, the following strategies should be employed:
 - **Use Frameworks that Automatically Handle XSS:** Modern frameworks like Angular, React, and Vue.js have built-in mechanisms to prevent XSS by escaping or sanitizing user inputs by default. Ensure these features are enabled and properly configured.
 - **Sanitize User Inputs:** Apply rigorous input validation and sanitization on both the client and server-side to prevent malicious scripts from being executed.
 - **Content Security Policy (CSP):** Implement CSP headers to restrict the sources of executable scripts and mitigate the impact of any successful XSS attacks.

SSRF (Server-Side Request Forgery)

- **Risk Level:** High
- **Vulnerability Detail:** The application accepted user-supplied URLs for fetching images, leading to SSRF vulnerabilities.
- **Validation Evidence:** The payload targeting an internal resource was successfully fetched, confirming the exploit.
- **Impact:** An attacker could manipulate the server to request internal resources, compromising sensitive data.
- **Remediation:**
 - **Validate and Sanitize Input:** Ensure all user-provided URLs are properly validated and sanitized to prevent the server from making unintended requests.
 - **Restrict URL Fetching:** Restrict the fetching of URLs to only allow known safe domains or paths.
 - **Pre-fetch client-side:** If possible, pre-fetch image client-side, and only send to server the resulting image.

CSRF (Cross-Site Request Forgery)

- **Risk Level:** Medium
- **Vulnerability Detail:** The application allowed unauthorized changes to user settings without adequate protection.
- **Validation Evidence:** A crafted HTML page successfully changed a user's username without consent.
- **Impact:** This could lead to unauthorized changes to user profiles, undermining user trust.
- **Remediation:** To protect against CSRF vulnerabilities:
 - **Use Anti-CSRF Tokens:** Ensure that each form submission includes a server-side validated token.
 - **Adopt Same-Site Cookies:** Configure cookies to be only sent in requests originating from the same site the cookie was set.
 - **Implement CORS Policies:** Properly configure Cross-Origin Resource Sharing (CORS) policies to restrict resources to trusted domains only.

Broken Access Control (Web3 Sandbox)

- **Risk Level:** Medium
- **Vulnerability Detail:** An unintended code sandbox was accessible, allowing unauthorized interactions with smart contracts.
- **Validation Evidence:** Direct access to the sandbox was achieved through URL guessing.
- **Impact:** Unauthorized users could deploy potentially harmful smart contracts.
- **Remediation:** To prevent such issues and secure web applications from similar vulnerabilities:
 - **Environment Segregation:** Ensure that development, testing, and production environments are strictly segregated. Tools and features that are used for development and testing should not be available on production servers.
 - **Access Controls:** Implement robust access controls to restrict access to administrative or sensitive functionalities only to authorized users.
 - **URL Guessing Mitigation:** Employ techniques like URL obfuscation or better yet, disable directory listing and ensure sensitive URLs are not easily guessable.
 - **Regular Audits:** Conduct regular security audits and penetration testing to identify and rectify security lapses such as exposed internal tools or functionalities.
-

Sensitive Data Exposure (Forgotten Sales Backup)

- **Risk Level:** Low
- **Vulnerability Detail:** A backup file was publicly accessible, containing sensitive data.
- **Validation Evidence:** Accessed the backup file using NULL byte injection.
- **Impact:** Exposure of sensitive information could damage the organization's reputation and lead to compliance issues.
- **Remediation:**
 - **Proper Access Controls:** Ensure that backup files are not stored in publicly accessible directories. Use proper access control mechanisms to restrict access.
 - **Regular Audits and Cleanups:** Perform regular audits of storage locations and cleanup legacy or unused files to prevent accidental exposure.
 - **Input Sanitization:** Enhance security measures to sanitize user inputs, especially in file retrieval functionalities, to prevent directory traversal or file inclusion attacks.

Error Handling

- **Risk Level:** Low
- **Vulnerability Detail:** The application revealed backend details due to improper error handling, leading to the exposure of sensitive information such as server configuration and version.
- **Validation Evidence:** When accessing a restricted or non-existent file path, a 403 error message disclosed that the server was running **Express JS version 4.17.1**. This is considered verbose and provides useful information to potential attackers.
- **Impact:** Attackers could leverage this information to target known vulnerabilities in the disclosed version of Express JS, increasing the risk of exploitation. This could affect server integrity, leading to data breaches or further exploitation of other vulnerabilities.
- **Remediation:**
 - **Implement Generic Error Messages:** Customize error pages to provide less information about the server's backend or software versions.
 - **Error Handling Best Practices:** Review and apply best practices for error handling that avoid revealing details about the server's architecture, software stack, or directory structure.
 - **Regular Updates and Patching:** Keep server software updated to mitigate known vulnerabilities, especially when version details might be disclosed inadvertently.

Login Admin (SQL Injection)

- **Risk Level:** High
- **Vulnerability Detail:** The web application's login form was vulnerable to SQL Injection, allowing unauthorized access to the administrator's account by manipulating the SQL query used for authentication.
- **Validation Evidence:** By submitting the payload ' or 1=1;-- in the email field, the login query was modified to always evaluate as true, granting admin access without a valid password.
- **Impact:** If exploited, this vulnerability could allow attackers to gain full control over the application, leading to the exposure of sensitive data, unauthorized system changes, and further compromise of the organization's infrastructure.
- **Remediation:**
 - **Use of Prepared Statements:** Utilize prepared statements with parameterized queries to prevent SQL injection. This technique ensures that the SQL interpreter recognizes the code and data separately.
 - **Input Validation:** Implement strict validation for all user inputs to ensure only expected data types and formats are processed.
 - **Error Handling:** Configure error messages to avoid revealing details about the database structure or any SQL syntax errors.
 - **Security Testing:** Regularly conduct security assessments and penetration testing to identify and mitigate vulnerabilities like SQL injection.

Methodology

1. Reconnaissance
2. Vulnerability Identification
3. Exploitation
4. Post Exploitation

Assessment Toolset Selection

The following tools were utilized during the vulnerability assessment:

- **Web Browser:** Used to interact with the application and test vulnerabilities.
- **Burp Suite:** This tool was essential for intercepting, analyzing, and manipulating HTTP requests, allowing for the discovery and exploitation of vulnerabilities such as CSRF and SSRF.

Assessment Methodology Detail

DOM XSS

1. Identify the Injection Point:

- The application's search functionality was identified as a possible vector for XSS attacks. The search parameter q in the URL was used to manipulate the DOM.

2. Craft the Payload:

- The XSS payload used was: `<iframe src="javascript:alert(Hatem Abdelfatah)">`. This script is designed to execute JavaScript directly from the iframe source attribute, triggering an XSS alert.

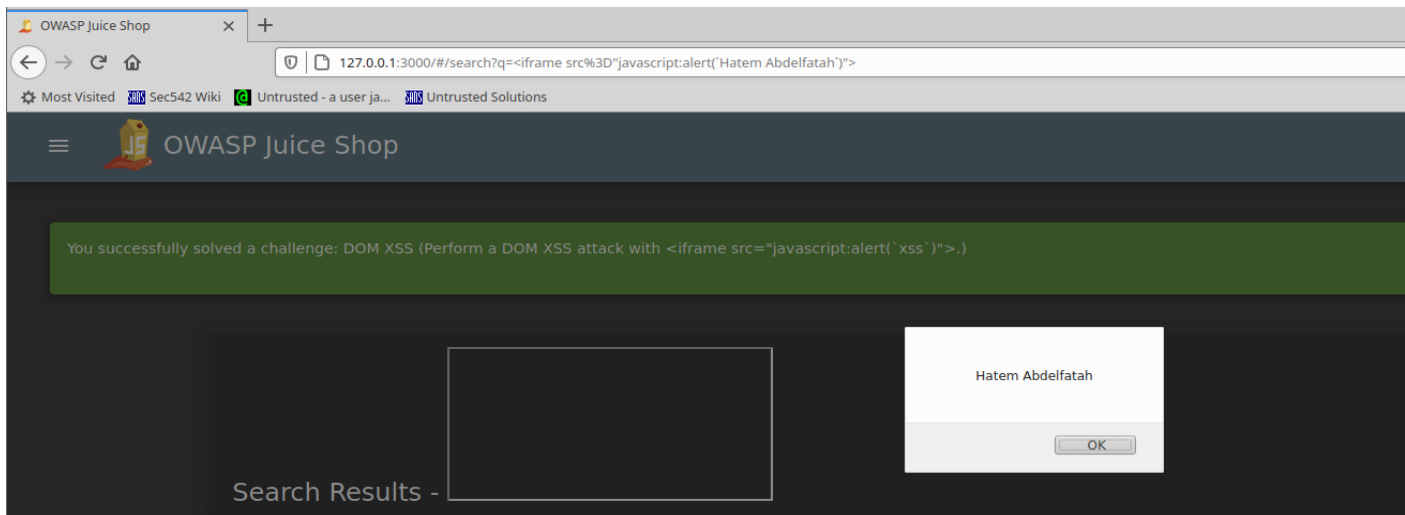
3. Encode and Inject the Payload:

The payload was URL-encoded to bypass basic URL parsing and injected into the application via the search URL

[http://127.0.0.1:3000/#/search?q=%3Ciframe%20src%3D%22javascript:alert\(%60Hatem%20Abdelfatah%60\)%22%3E](http://127.0.0.1:3000/#/search?q=%3Ciframe%20src%3D%22javascript:alert(%60Hatem%20Abdelfatah%60)%22%3E)

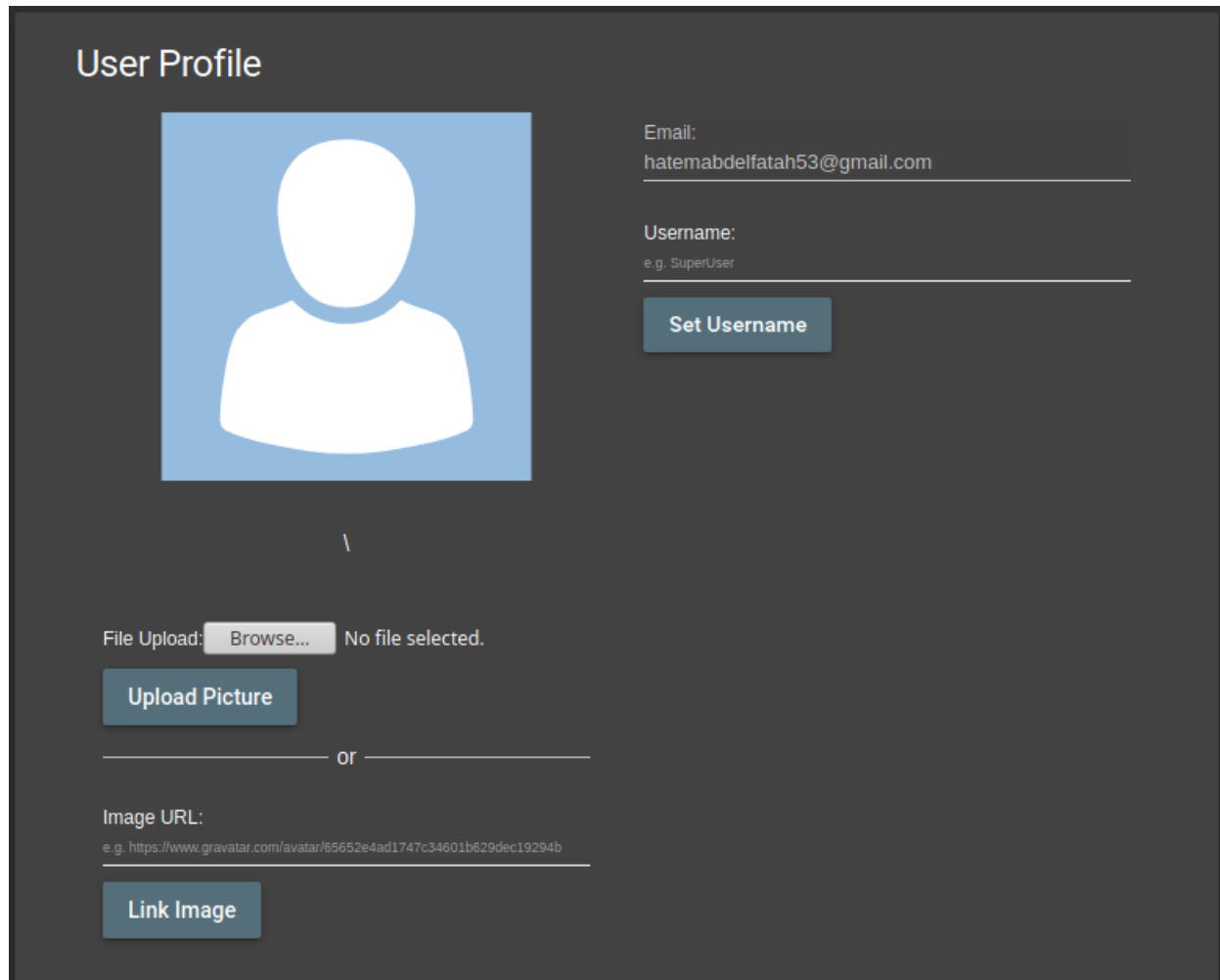
4. Execute the Attack:

- Navigating to the manipulated URL resulted in the execution of the malicious script, displaying an alert box indicating a successful XSS attack.

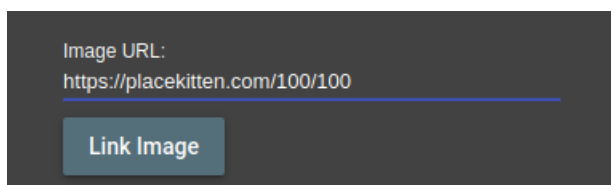


SSRF (Server-Side Request Forgery)

1. **Identify Potential SSRF Points:** Knowing the challenge involves SSRF, I looked for places within the Juice Shop application where external URLs are fetched by the server. The user profile section, where users can set their profile image via a URL, seemed a likely candidate.

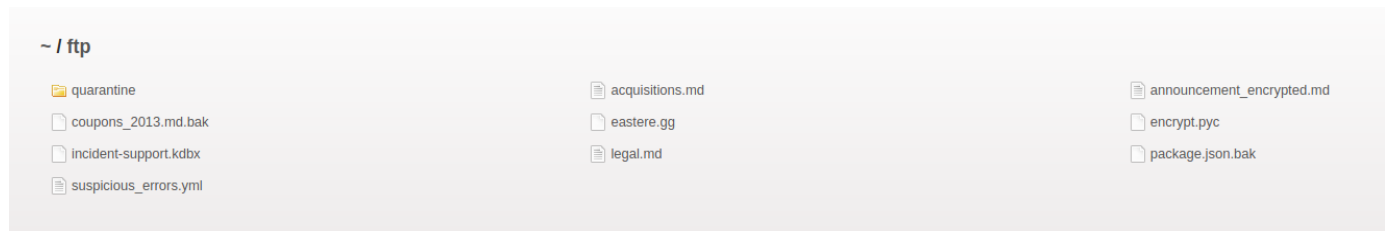


2. **Testing the Theory:** Using Burp Suite, I intercepted the request made when changing the profile image and observed that it accepts an external URL for the image. This behavior suggests that the server fetches the image from the URL provided by the user, which is a classic setup for SSRF vulnerabilities.



In fact, using Burp I found that my URL was directly injected in imageURL parameter of the request, which means that URL will be fetched server-side.

3. **Exploring the Quarantine Directory:** Following hints provided in the challenge description, I explored URLs contained in malware files hosted under the /ftp/quarantine/ directory on the server.



Each of these URLs was a potential trigger for the SSRF.

4. **Manipulating the Image URL:** Initially, I attempted to use the URLs directly from the malware files as the image source in the user profile to trigger the SSRF. However, these attempts did not succeed
5. **Finding the Trigger:** Malwares files were the only ones that I didn't opened. So I took the decision to uncompile them to understand how they work.

Within the malware files, I noticed a specific URL:

http://localhost:3000/solve/challenges/server-side?key=tRy_H4rd3r_n0thIng_iS_Imp0ssibl3

This URL appeared to be designed specifically for this challenge, likely a trigger for the SSRF. In fact, nothing happened when I tried to only visit the URL.

6. **Exploiting the SSRF:** Changing the profile image URL to the one discovered in the malware file and applying the changes caused the server to make a request to the internal URL. This internal request solved the challenge.

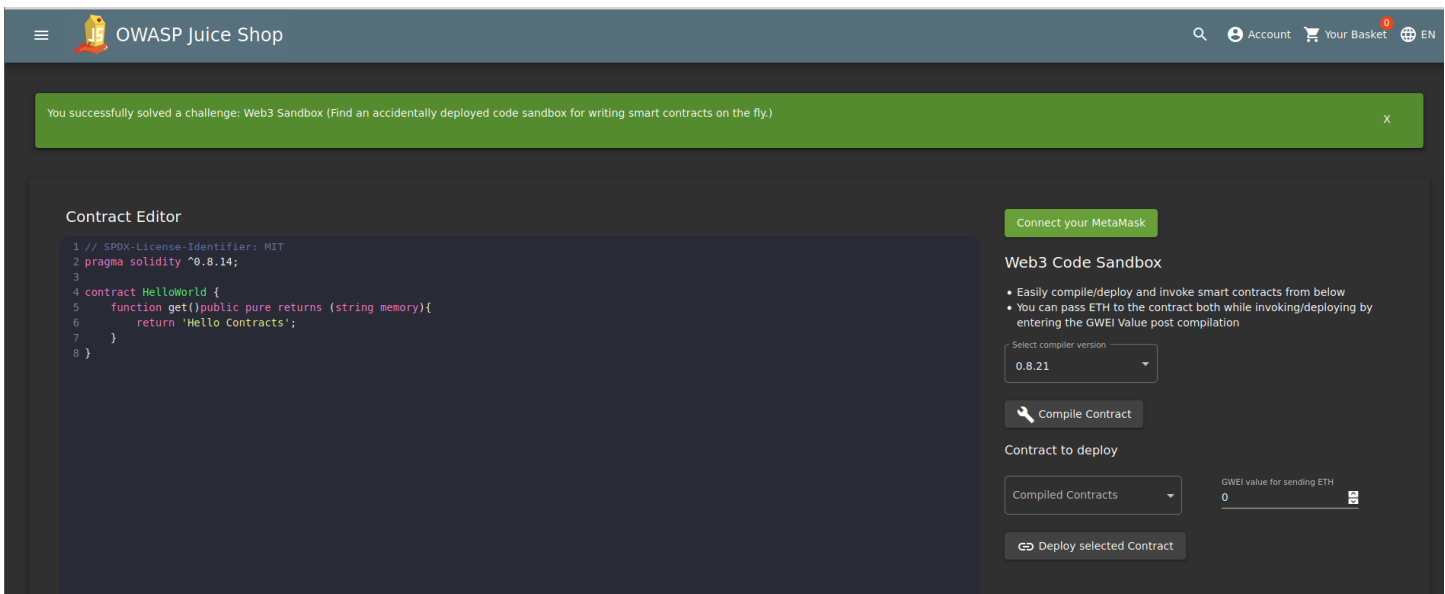
CSRF (Cross-Site Request Forgery)

1. Endpoint Discovery:

- Use Burp Suite to intercept and examine HTTP requests made while changing the username through the application's profile page.
- Identify the POST request that changes the username, specifically noting the URL and form parameters.

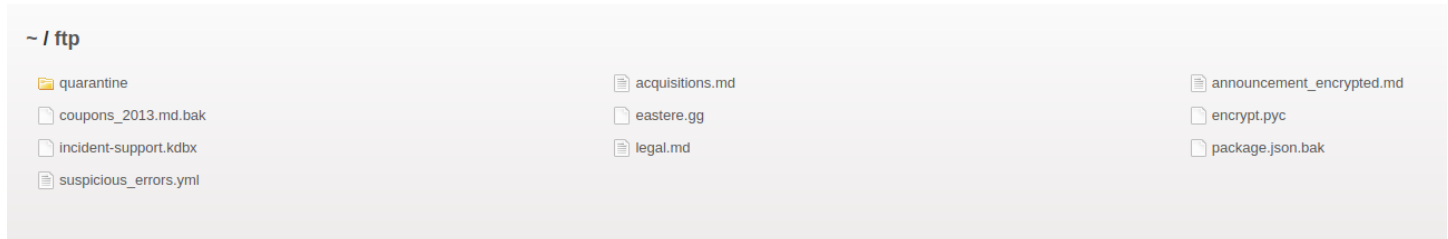
Broken Access Control (Web3 Sandbox)

1. **Discovery:** Began by exploring different URLs related to Web3 functionalities on the application, suspecting that the sandbox might be located at a somewhat obscure or less obvious path.
2. **Guessing URL:** Utilized a simple guessing approach, inspired by common directory names associated with testing or development environments like /sandbox, /test, /dev, or similar. In this case, it involved trying variations until hitting the correct URL: `http://127.0.0.1:3000/#/web3-sandbox`.
3. **Accessing the Sandbox:** Upon navigating to the correct URL, found a fully functional Web3 code sandbox environment. This sandbox included features for editing, compiling, and deploying Ethereum smart contracts directly from the browser.



Sensitive Data Exposure (Forgotten Sales Backup)

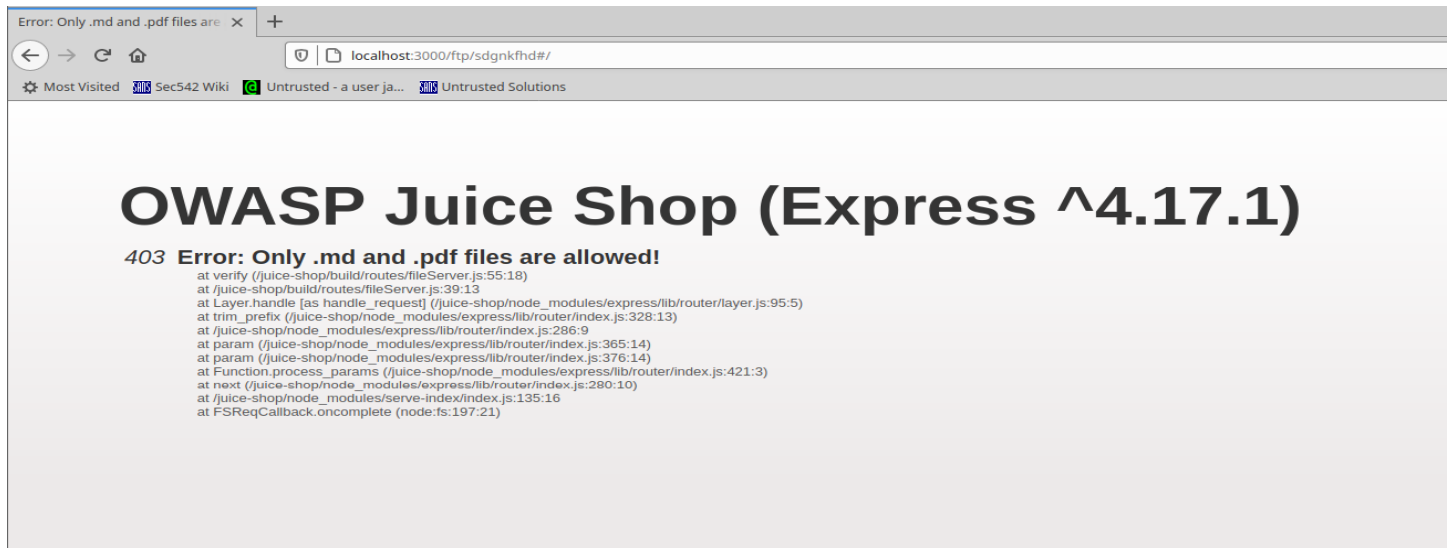
1. **Access the FTP Server:** Navigated to the FTP server directory listing at 127.0.0.1:3000/ftp, which was discovered in a previous challenge (Privacy Policy). This server hosted various files, including backups and configuration files.
2. **Identify the Target File:** Scanned through the list of files and identified coupons_2013.md.bak as a likely candidate for containing sensitive sales data due to its naming convention indicating it's a backup file.



3. **Exploit NULL Byte Injection:** Utilized a NULL byte injection technique to access the backup file. Many web applications do not properly sanitize input, allowing for directory traversal or file inclusion attacks if the application is improperly handling file names.
 - **Construct the URL:** Appended a NULL byte character, represented in URL encoding as %00, to the file request to potentially bypass any restrictions or parsing issues by the server. This was tried because older or misconfigured servers might treat the NULL byte as a string terminator, thus ignoring any file extension checks or other controls.
 - **Request the File:** Accessed the URL `http://127.0.0.1:3000/ftp/coupons_2013.md.bak%00.md` in a browser.
4. **Review the Data:** Upon successful retrieval, reviewed the content of coupons_2013.md.bak for any sensitive information, confirming the exposure of sales data or discount coupons that should not have been publicly accessible.

Error Handling

1. **Experiment with URL Paths:** Accessed various non-existent or restricted URL paths on the application's server to trigger error responses.
2. **Analyze Error Messages:** Carefully examined the error messages for any signs of verbose or detailed information leakage.



Login Admin (SQL Injection)

1. **Intercept Login Request:** Using Burp Suite, the HTTP POST request sent during a normal login attempt was intercepted. This request included user credentials in the form of JSON data.
2. **Modify the SQL Query:** Modified the email field in the JSON payload to ' or 1=1;--, effectively turning the SQL command into a statement that always returns true, bypassing the need for a password.

