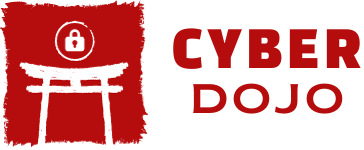
SECURITY ASSESSMENT

Juice Shop Vulnerabilities Report



Submitted to: DEPI- Ministry of Communications and Information Technology

Pen tester Name: 1- Abdelfattah Ahmed Sharaf

2-Talaat Amr Talaat Abazaa   
  
Date of Testing: 20 – 10 - 2024  
Date of Report Delivery: 22 – 10 - 2024

|  |
| --- |
|  |

**Table of Contents**

Contents

[**Security Engagement Summary 2**](#_heading=h.30j0zll)

[Engagement Overview 2](#_heading=h.3znysh7)

[Scope 2](#_heading=h.tyjcwt)

[Risk Analysis](#_heading=h.1pxezwc) 2

[Recommendation](#_heading=h.49x2ik5) 2

[**Significant Vulnerability Summary 3**](#_heading=h.lnxbz9)

[High Risk Vulnerabilities 3](#_heading=h.35nkun2)

[Medium Risk Vulnerabilities 3](#_heading=h.1ksv4uv)

[Low Risk Vulnerabilities 3](#_heading=h.44sinio)

[**Significant Vulnerability Detail 4**](#_heading=h.2jxsxqh)

[Broken Access ControL 4](#_heading=h.z337ya)

[Sensitive information Exposure 4](#_heading=h.z337ya)

DOM Based Cross Site Scripting [4](#_heading=h.z337ya)

[Weak Password Complexity RequirementsVulnerability Name 4](#_heading=h.z337ya)

[Application Error Message 5](#_heading=h.1y810tw)

[**Methodology 6**](#_heading=h.3whwml4)

[Assessment Tools Selection 6](#_heading=h.2bn6wsx)

[Assessment Methodology Detail 6](#_heading=h.qsh70q)

# Security Engagement Summary

## Engagement Overview

The engagement involved assessing the security of the Juice Shop application. The engagement aims to identify weaknesses that could lead to unauthorized data access, manipulation, or other security threats, helping to improve its security posture. The team follows established security assessment methodologies to identify and evaluate vulnerabilities effectively

## Scope

The scope of this engagement includes the Juice Shop application, specifically focusing on its web interface, API backend, database, etc. This scope is appropriate as it encompasses critical components of the application that are most susceptible to attacks. By limiting the assessment to these areas, we can identify vulnerabilities that pose the highest risk to the application's confidentiality, integrity, and availability.

|  |  |  |
| --- | --- | --- |
| Type | Name | Scope |
| Web application | OWASP Juice shop | http://localHost:3000/ |

## Executive Risk Analysis

**Overall Risk Level: High**

The assessment identified a high-risk level for the Juice Shop application due to several critical and high-severity vulnerabilities that could result in unauthorized access, data breaches, or service disruptions. Key findings include:

* **Critical Vulnerabilities**: Multiple instances of SQL Injection and Cross-Site Scripting (XSS) were discovered. These vulnerabilities pose a severe threat, as they could allow attackers to manipulate the database, access sensitive user data, or execute malicious code within users' browsers.

The high-risk rating reflects the potential impact on data security, user privacy, and the organization’s reputation if these vulnerabilities are exploited. This risk level indicates an immediate need for remediation to protect both user data and system functionality.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| no | Vulnerability | Risk | Impact | Score | Attack Difficulty |
| 1 | Reach to JavaScript web code | Critical | Broken Access Control | 9.2 | Medium |
| 2 | Reach to sensitive URL | Critical | Data exposure | 9.0 | Easy |
| 3 | SQL injection | Critical | Send or reave query for database | 9.5 | Medium |
| 4 | DOM Based Cross Site Scripting | Critical | Print anything in the web page | 7.5 | Easy |
| 5 | URL decoder | medium | Modify in web application | 7.0 | Easy |

## Executive Recommendation

Remediation of some of the vulnerabilities identified in the report is highly suggested as it would have a direct impact on the business. Some of the vulnerabilities are such which have a very high likelihood of being exploited and could cause huge harm to the organization and the web application. The remediation efforts would include making changes in the source code of the program as well as configuring proper rules in the deployment of various servers to make sure unauthorized people do not have access to information not for them.

### 1. ****Enforce Access Control on the Server Side****

* Never rely solely on client-side code (JavaScript) for access control. Ensure that sensitive data and actions are validated and authorized on the server side before processing.
* Apply role-based or permission-based access checks in your backend code to restrict users from unauthorized actions or resources.

### 2. ****Use JWT (JSON Web Tokens) or Session Tokens****

* Implement authentication with secure tokens like JWT, ensuring they have short expiration times and are securely managed (e.g., stored in HTTP-only cookies or secured local storage).
* Regularly validate these tokens on the server side to confirm the user's access rights.

### 3. ****Minify and Obfuscate JavaScript Code****

* Minifying and obfuscating JavaScript code can deter attackers by making the code harder to read. Although this doesn't directly prevent access control issues, it adds a layer of complexity for anyone trying to tamper with your code.

### 4. ****Use Prepared Statements and Parameterized Queries****

* Avoid concatenating user input directly into SQL queries. Instead, use prepared statements with parameterized queries, which safely bind user input to the query.
* Most frameworks and databases support prepared statements. For example, in Java with JDBC:

String query = "SELECT \* FROM users WHERE username = ? AND password = ?";

PreparedStatement pstmt = connection.prepareStatement(query);

pstmt.setString(1, username);

pstmt.setString(2, password);

ResultSet rs = pstmt.executeQuery();

* Parameterized queries ensure that user inputs are treated as data, not executable code.

### 5. ****Use ORM (Object-Relational Mapping) Libraries****

* Using ORM libraries like Hibernate (Java), Sequelize (Node.js), or Django ORM (Python) helps prevent SQL Injection by abstracting query creation and handling data safely.
* ORMs typically generate SQL queries for you and often parameterize queries automatically.

### 6. ****Avoid Dynamic Queries Wherever Possible****

* Avoid constructing SQL queries dynamically using user input. For instance:

String query = "SELECT \* FROM users WHERE username = '" + username + "'";

* Instead, always use prepared statements or ORM methods to avoid risking SQL injection through string concatenation.

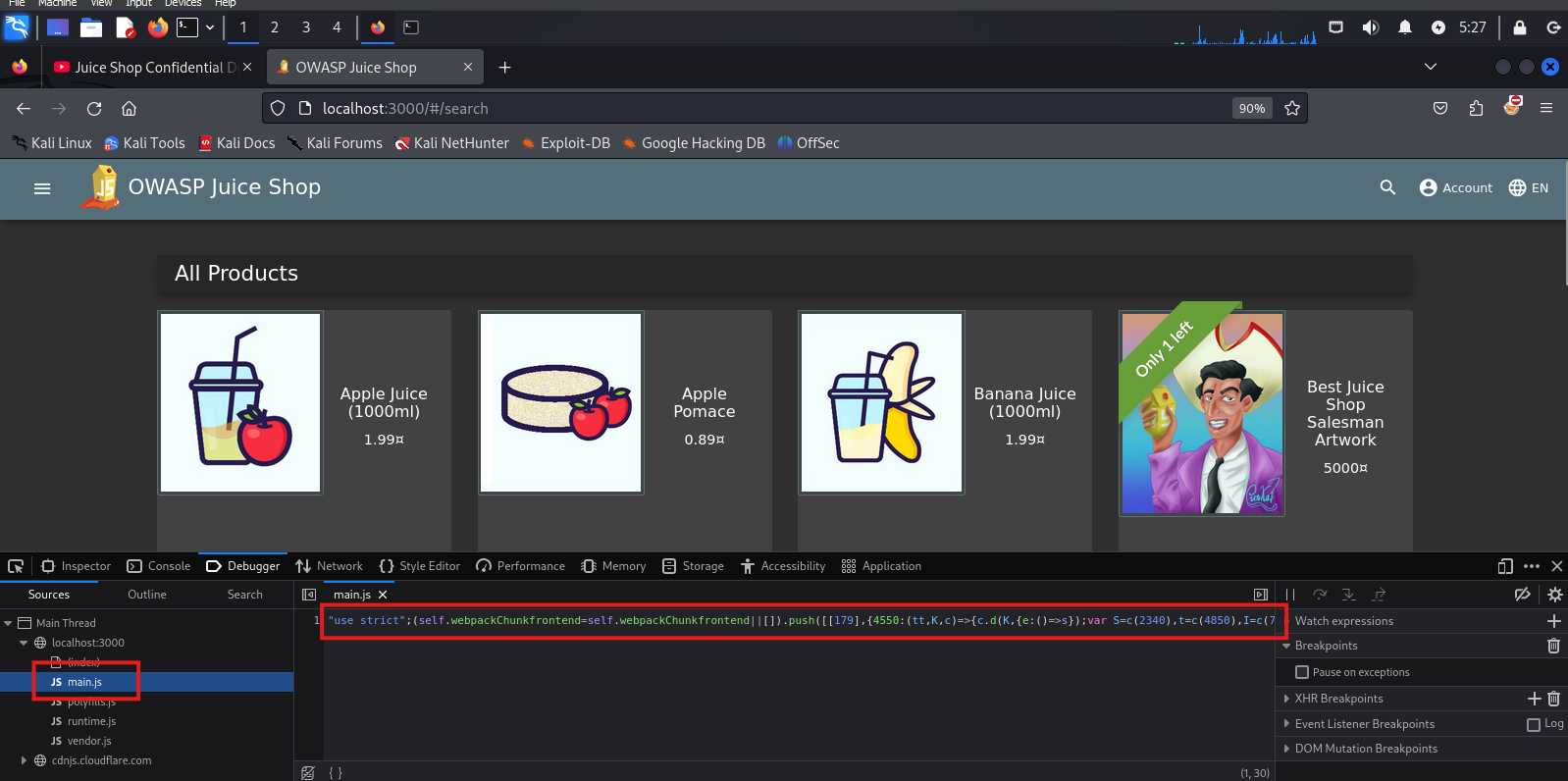
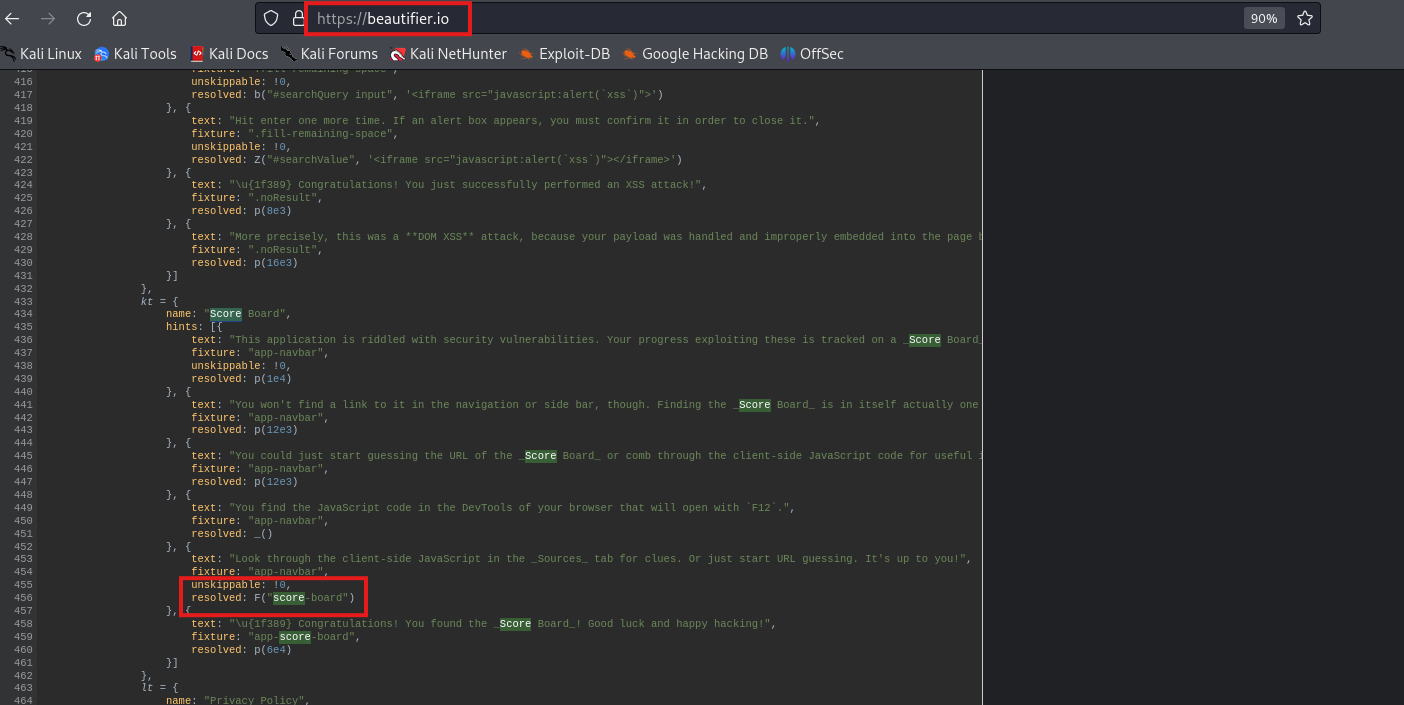
# Significant Vulnerability Summary

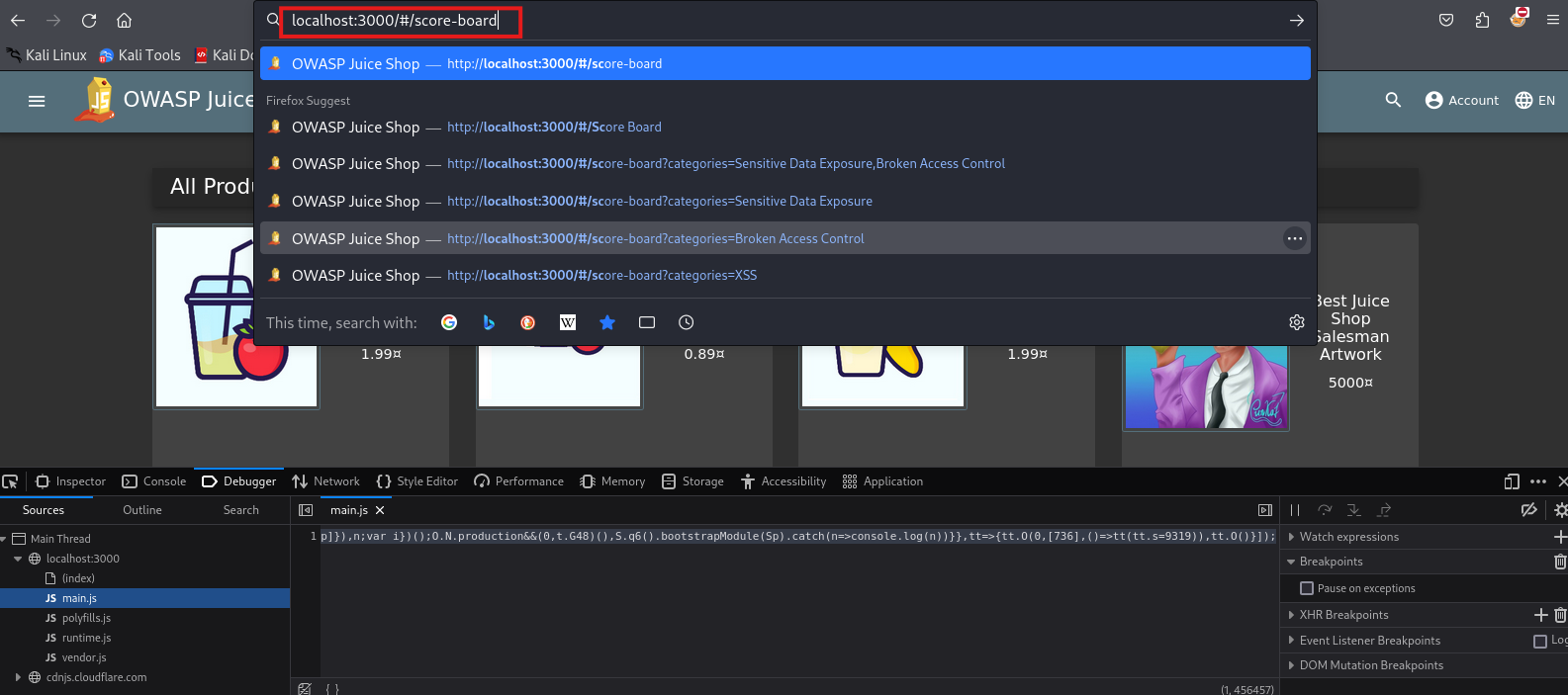
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| no | Vulnerability | Risk | Impact | Score | Attack Difficulty |
| 1 | Reach to JavaScript web code | Critical | Broken Access Control | 9.2 | Medium |
| 2 | Reach to sensitive URL | medium | Data exposure | 9.0 | Easy |
| 3 | SQL injection | Critical | Send or reave query for database | 9.5 | Medium |
| 4 | DOM Based Cross Site Scripting | normal | Print anything in the web page | 7.5 | Easy |
| 5 | URL decoder | normal | Modify in web application | 7.0 | Easy |
| 6 | Weak Password Complexity Requirements | medium | Broken Authentication | 7.0 | Medium |
| 7 | Application Error Message | low |  | 6.5 | Easy |

# Significant Vulnerability details

## Broken access control

It reaches nay attacker to saw the flow of web application and with some smart can access just by add or guess the supp URL

Here by F12 you can go to main.js code and to saw the code

by beaty the code you can saw all the supp URL by his name like (‘Socre-board’)

**Probability of exploiting**

This attack has a high probability of being exploited as it’s easy to perform.

**Impacted party**

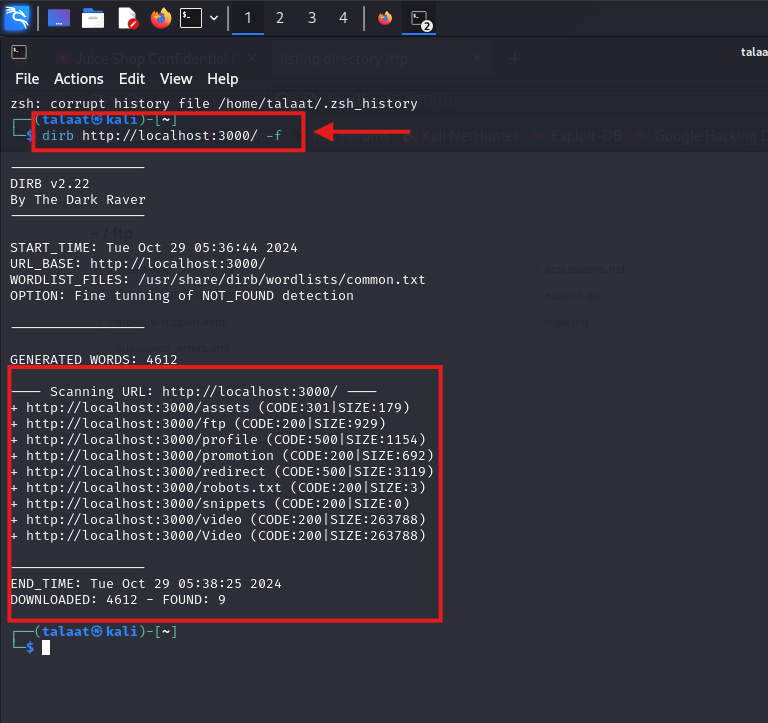
Every user on the web application.

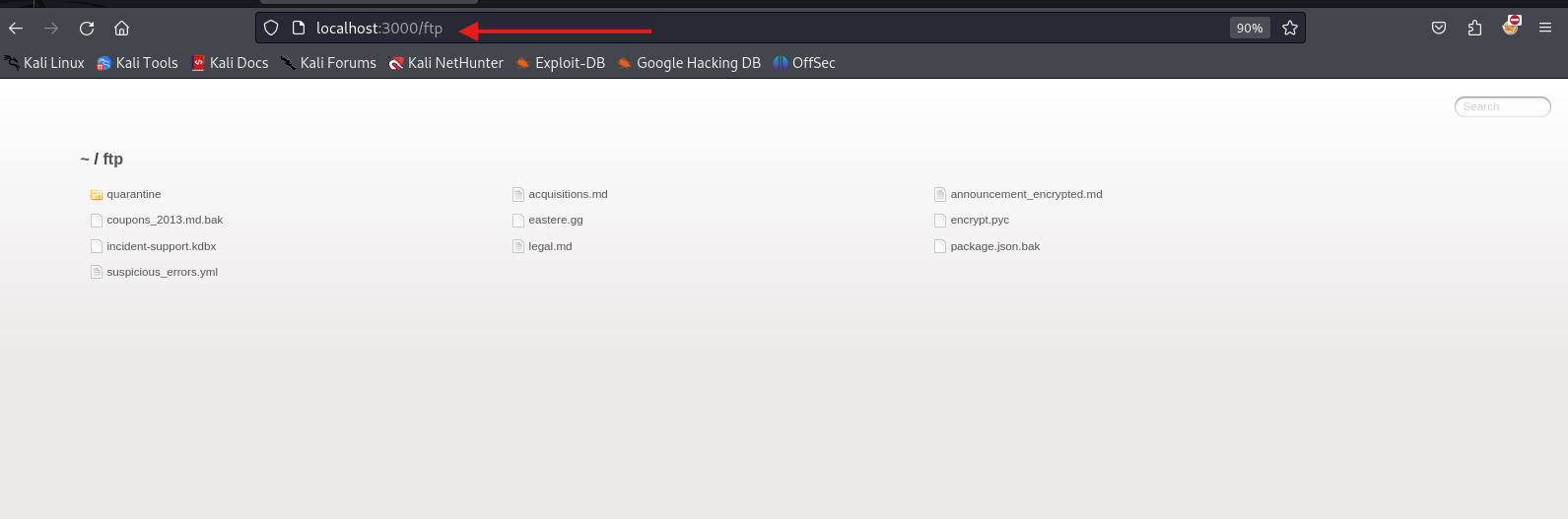
**Remediation**

* Never rely solely on client-side code (JavaScript) for access control. Ensure that sensitive data and actions are validated and authorized on the server side before processing.
* Apply role-based or permission-based access checks in your backend code to restrict users from unauthorized actions or resources.

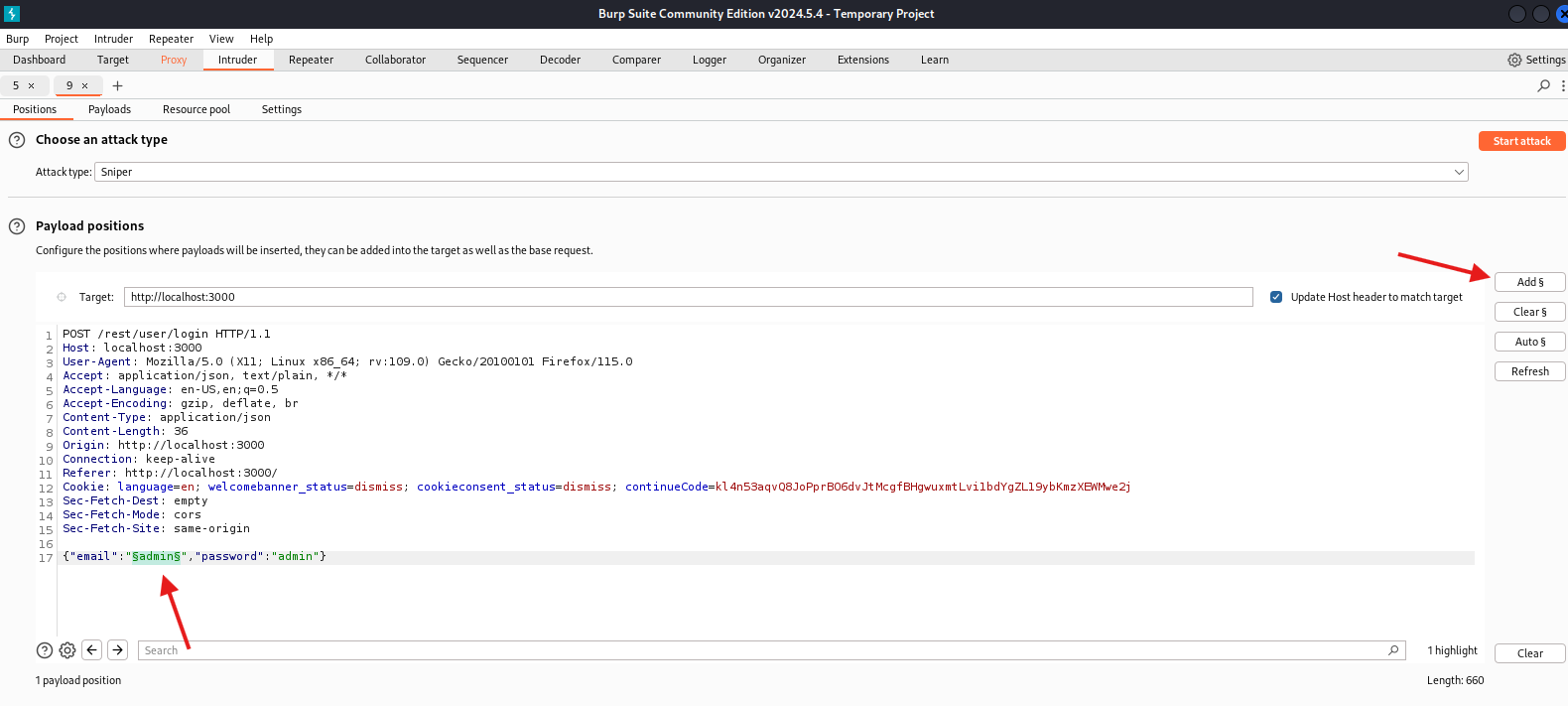
## Data exposure

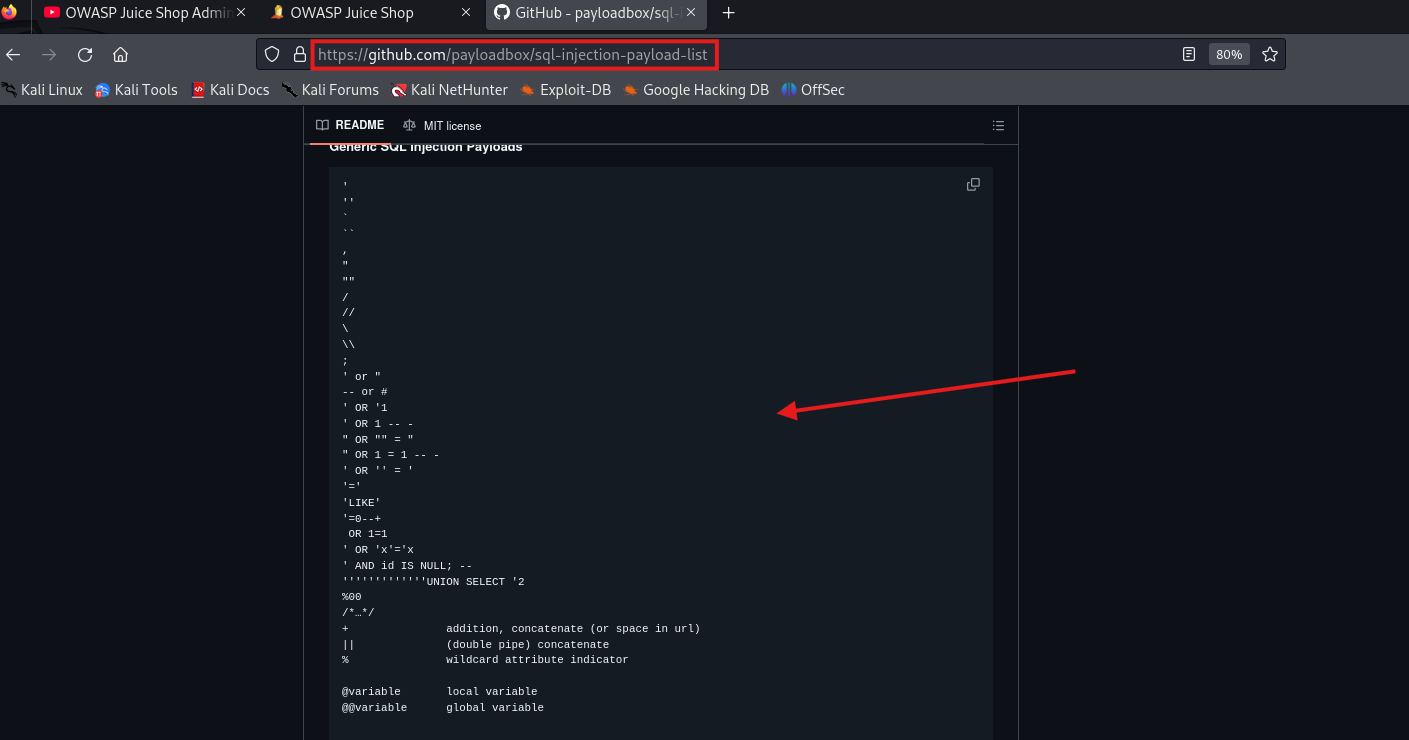
with such tool like dirb you can get the the word-list file of web application

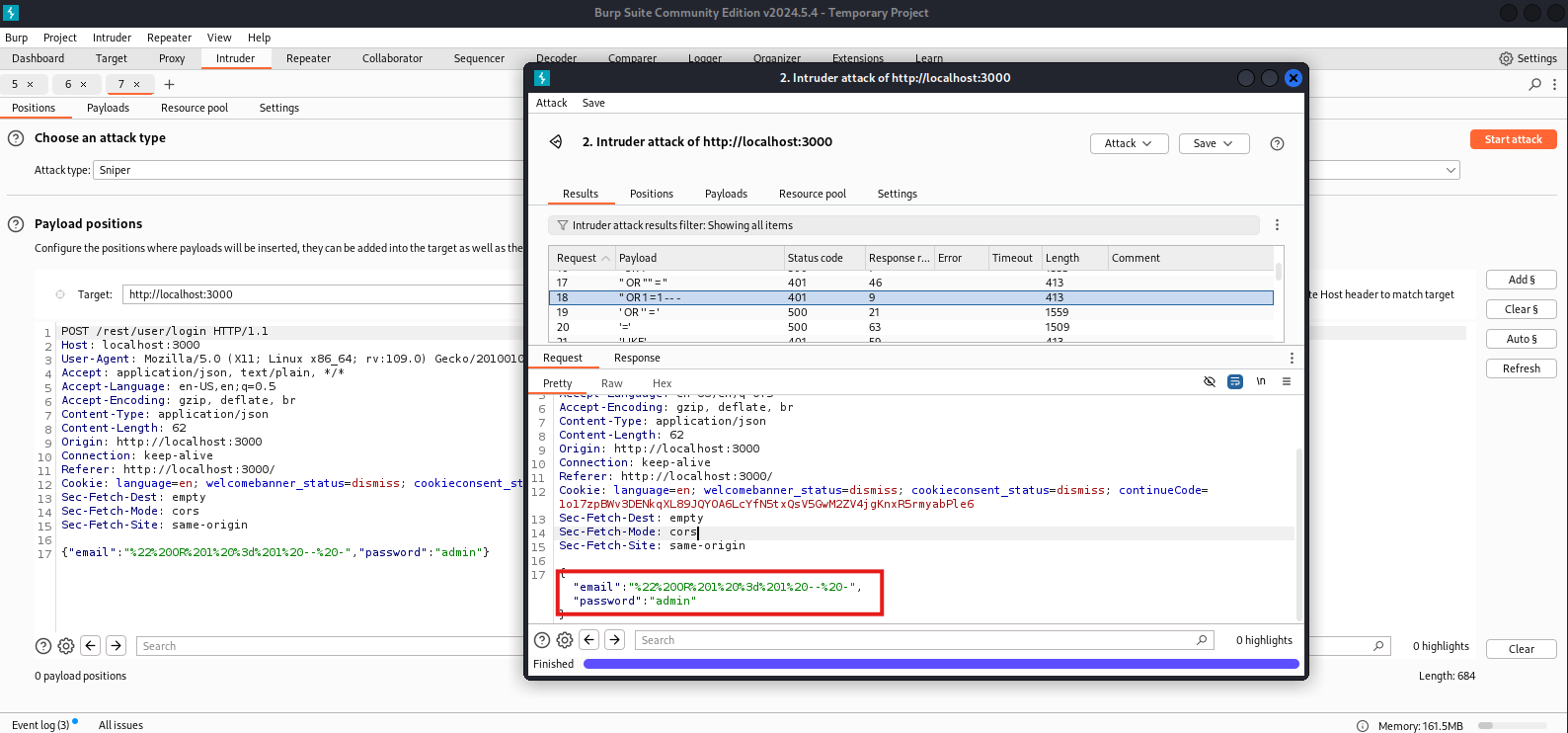
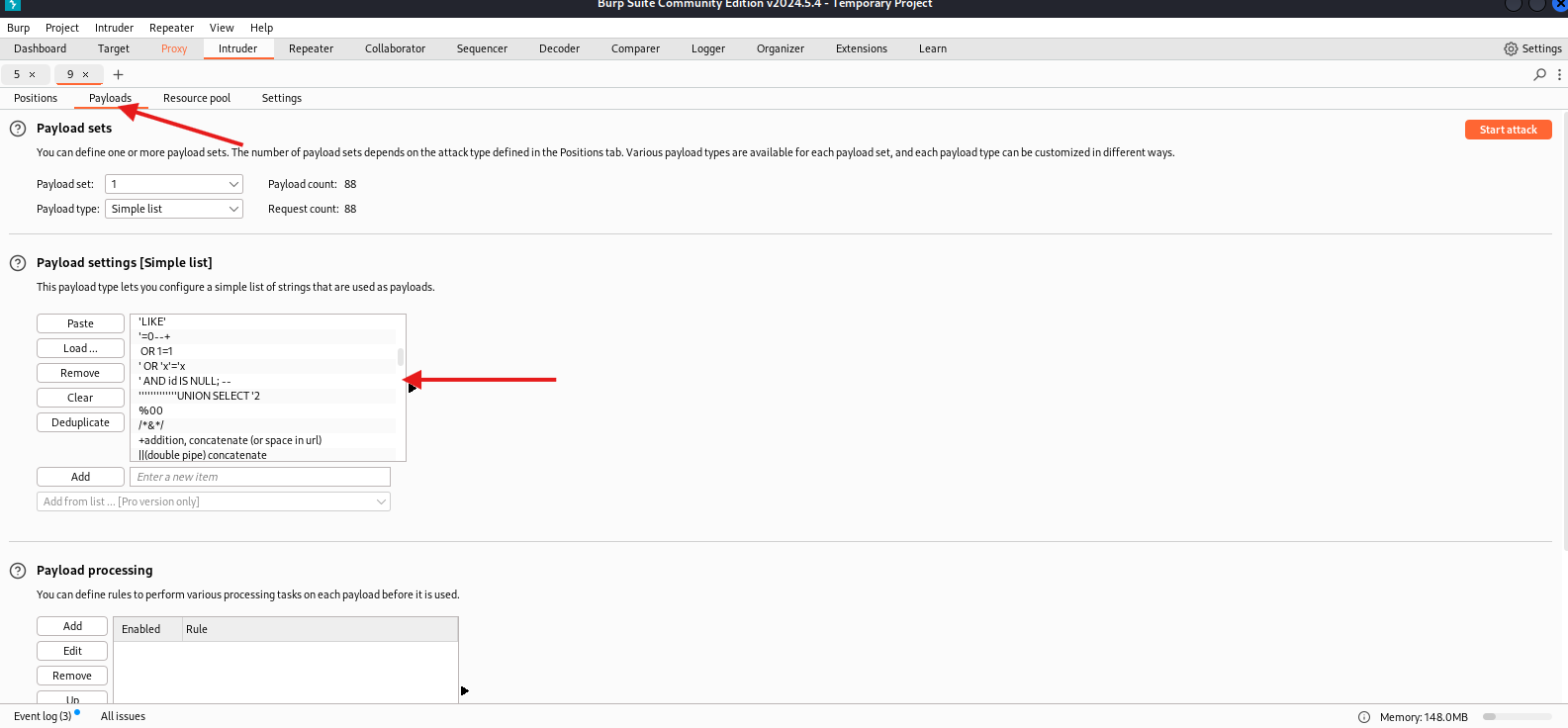


Then get all this end points that’s lead do file and data 

## SQL injection

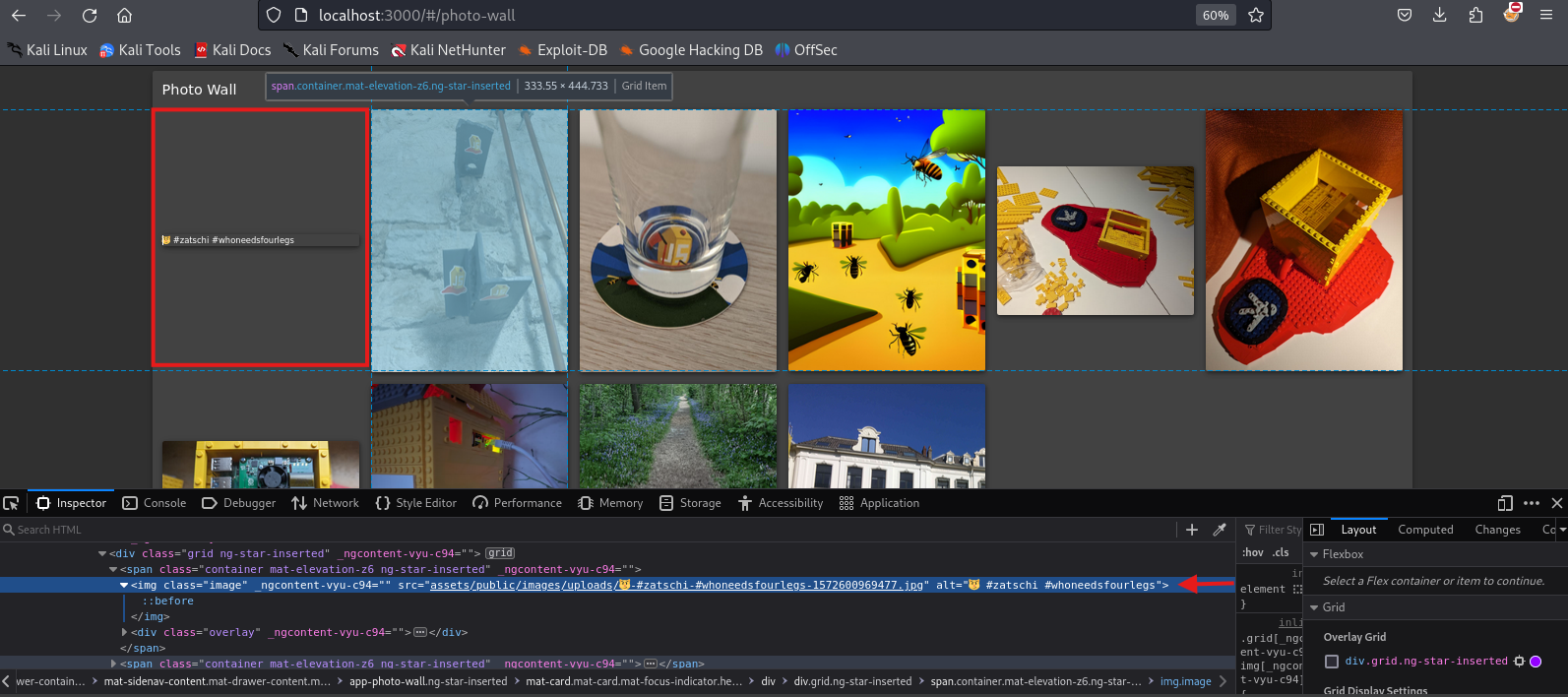
with Burp Suite tool and track the traffic of login page trying sniper attack to inject SQL payload to try a lot of ways and saw the response if it works 

add payload to try it buy get the payload sql injection from GitHub

add here 

Also, keep in mind that DOM XSS and other types of XSS are not mutually exclusive. Your application can be vulnerable to both reflected/stored XSS and DOM XSS. The good news is that if user input is handled properly at the foundation level (e.g. your framework), you should be able to mitigate all XSS vulnerabilities.

## URL decoder

Here form editing the URL of the image as it decoded by some knowledge you can recognize the coding on the url and reflect the decoding

## XSS

A screenshot of a computer

Description automatically generatedThe vulnerability was identified by testing SQL injection payloads against the web app login page. The vulnerability was validated by logging in as one of the customers with having to authenticate.

**Probability of exploiting**

This attack has a high probability of being exploited as it’s easy to perform.

**Impacted party**

Every user on the web application.

**Remediation**

* The preferred option is to use a safe API, which avoids using the interpreter entirely, provides a parameterized interface, or migrates to Object Relational Mapping Tools (ORMs).  
  Note: Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data or executes hostile data with EXECUTE IMMEDIATE or exec().
* Use positive server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.
* For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter.  
  Note: SQL structures such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report-writing software.

**Weak Password Complexity Requirements**

**Risk Level: Medium**

**Executive Summary**

There are three common types of password guessing attacks. The first is a brute-force attack in which attackers try every combination of every letter in order to eventually find the correct password. Dictionary attacks utilize a list of common passwords such asPassword1andabc123. The third type of attack is a hybrid attack in which the attacker uses common passwords that have been mangled with brute-force techniques. For instance, the attacker might try the wordSecret followed by every possible 2-digit numeral and symbol combination. This can be successful when users tack on numbers and symbols to the end of their password to comply with password requirements

**Probability of Exploit/Attack**

Secure Ideas found that while the application attempts to enforce the use of complex passwords, the password complexity requirements are weaker than recommended for this type of application. Secure Ideas found that the application tested allowed passwords such asadmin123and password123. These types of passwords are commonlyfound in widely-accessible dictionaries. As a matter of fact, Secure Ideas commonly uses thePassword123string against systems that implement account lockout due to it commonly being found as the password for accounts in web applications..

**Impact of Exploit**

If exploited, this vulnerability could have the following impact:

* **User Groups**: Individual users may have their accounts compromised if attackers can successfully guess weak passwords.
* **Departments**: Security and IT teams would need to respond to account compromise incidents, increasing their workload.
* **Business Continuity/Revenue**: If multiple accounts are compromised, it could lead to a data breach, resulting in reputational damage, customer distrust, and potential revenue loss.

**Potential Remediation**

To address this vulnerability, the following actions are recommended:

1. **Implement Strong Password Policies**: Enforce requirements for password complexity, including a minimum of 8 characters with a mix of uppercase letters, lowercase letters, numbers, and special characters.
2. **Enable Account Lockout Mechanisms**: Implement a limit on failed login attempts to prevent brute-force attacks.
3. **Encourage Password Management Tools**: Educate users on secure password practices and recommend using password managers to create and store complex passwords.

These measures will reduce the likelihood of account compromise due to weak passwords, enhancing overall security for user accounts.

**Application Error Message**

**Risk Level: Low**

**Executive Summary**

The Application Error Message vulnerability was identified during testing by intentionally triggering errors within the Juice Shop application. It was observed that the application displays overly detailed error messages, including sensitive information such as server paths, database structure, or internal configuration details. These verbose error messages can inadvertently aid attackers by revealing insights into the application’s internal workings.

**Evidence of Validation**

The vulnerability was validated by causing an error through invalid inputs in various fields (e.g., invalid SQL input, unexpected characters in a URL). The application responded with a detailed error message instead of a generic message, exposing information about the application’s underlying technology and structure.

**Probability of Exploit/Attack**

The probability of exploitation is moderate, given that this information could help attackers in understanding the system's infrastructure, although it does not provide direct access to the application. However, in combination with other vulnerabilities, such information could assist in developing more targeted attacks.

**Impact of Exploit**

If exploited, this vulnerability could have the following impact:

* **User Groups**: Users would indirectly be affected as attackers may use these insights to bypass security measures or discover additional vulnerabilities.
* **Departments**: Security and IT teams would need to handle additional security reviews to address any related weaknesses exposed by the detailed error messages.
* **Business Continuity/Revenue**: While the risk is lower, reputational damage could occur if attackers leverage these insights to breach the system or exfiltrate data.

**Potential Remediation**

Remediation for this vulnerability would include identifying all different errors and handling them gracefully with special landing pages for the same:

1. **Display Generic Error Messages**: Replace detailed error responses with generic messages such as “An error occurred. Please try again.” to prevent sensitive information exposure.
2. **Log Detailed Errors Internally**: Configure error handling so that detailed error messages are only logged internally and not displayed to end-users.
3. **Perform Regular Code Reviews**: Ensure code handling user inputs and errors does not inadvertently expose sensitive application details.

# Methodology

**Quick Overview**

The penetration test was conducted following the OWASP Testing Guide's standard methodology, which ensures a comprehensive assessment of web application security. The testing process is divided into three key phases: \*\*Information Gathering\*\*, \*\*Vulnerability Identification\*\*, and \*\*Exploitation\*\*. Each phase is designed to identify vulnerabilities and evaluate the effectiveness of security controls within the OWASP Juice Shop environment. The overall goal of the assessment was to discover, analyze, and report any vulnerabilities or weaknesses that could potentially be exploited by malicious attackers.

**Assessment Tool Set Selection**

To facilitate a thorough assessment of OWASP Juice Shop, a combination of automated and manual tools were utilized. These tools were selected based on their ability to detect various classes of vulnerabilities and ease of integration into the testing workflow.

- \*\*Burp Suite\*\* (Professional): Used for intercepting requests, performing active and passive scans, and automating testing of input validation vulnerabilities.

- \*\*OWASP ZAP\*\*: Employed for automated vulnerability scanning and testing of common web application vulnerabilities.

- \*\*Nessus\*\*: Used for identifying vulnerabilities, misconfigurations, and weaknesses on the infrastructure side.

- \*\*Nikto\*\*: A web server scanner used for basic enumeration and discovery of known vulnerabilities.

- \*\*SQLMap\*\*: For identifying and exploiting SQL injection vulnerabilities.

- \*\*Firefox Developer Tools\*\*: Used for manual exploration, input testing, and verifying the behavior of web functionalities.

- \*\*Kali Linux\*\*: The operating system used to consolidate the testing tools and environment.

- \*\*Custom Scripts\*\*: Tailored scripts were used where necessary for automated interaction with various web application components and to attempt exploitation of discovered vulnerabilities.

**Assessment Methodology Detail**

The assessment was conducted using a mix of automated and manual techniques in line with the \*\*OWASP Testing Guide\*\*. Below is the detailed breakdown of the testing methodology:

Information Gathering

- \*\*Reconnaissance and Fingerprinting\*\*: The initial phase involved collecting information about the OWASP Juice Shop’s server configuration, technologies in use, and potential points of entry. Automated tools like \*\*Nmap\*\* were used for port scanning, and \*\*Nikto\*\* was used to identify potential weaknesses in the server configuration.

- \*\*Enumeration of Endpoints\*\*: All publicly accessible endpoints, parameters, and features of the Juice Shop web application were enumerated to create a testing scope. Tools like \*\*Burp Suite\*\* and \*\*OWASP ZAP\*\* were employed to map out the application.

Vulnerability Identification

- \*\*Input Validation Testing\*\*: Tests were performed to identify weaknesses in input validation mechanisms. This included:

- Testing for \*\*SQL injection\*\* using \*\*SQLMap\*\* and manual input fuzzing.

- \*\*Cross-Site Scripting (XSS)\*\* testing, both reflected and stored, by inserting crafted payloads in all input fields.

- \*\*Cross-Site Request Forgery (CSRF)\*\* testing to check for unprotected requests.

- \*\*Session Management Testing\*\*: The application’s authentication and session management mechanisms were examined, including cookie handling, session expiration, and the presence of secure flags. \*\*Burp Suite\*\* and \*\*Firefox Developer Tools\*\* were used to manipulate cookies and session tokens.

- \*\*Authentication and Authorization\*\*: Tests were conducted to validate the effectiveness of authentication mechanisms and to check if users with lower privileges could access restricted resources. \*\*Burp Suite\*\* was used to tamper with user roles, session IDs, and HTTP methods to bypass authentication controls.

- \*\*File Upload Vulnerabilities\*\*: OWASP Juice Shop’s file upload functionality (if present) was tested to ensure that malicious files could not be uploaded and executed.

Exploitation

- \*\*Privilege Escalation\*\*: Attempts were made to elevate privileges by exploiting vulnerabilities found in authorization checks or by leveraging poorly protected user sessions.

- \*\*Injection Attacks\*\*: Identified input vulnerabilities were exploited using tools like \*\*SQLMap\*\* for SQL injection or manual payload insertion for \*\*XSS\*\*.

- \*\*Data Exfiltration\*\*: In cases where sensitive data exposure was found (e.g., via \*\*IDOR\*\*, SQLi), data was exfiltrated to validate the impact of the vulnerability.

- \*\*Exploitation of Business Logic Flaws\*\*: Manual testing was conducted to identify business logic vulnerabilities, ensuring that security controls enforced by the business processes were functioning as intended.

Reporting and Verification

- \*\*Results Documentation\*\*: Each vulnerability identified was documented with its impact, risk rating, and proof-of-concept. Screenshots and HTTP request/response logs were included for each successful exploit.

- \*\*Remediation Recommendations\*\*: For each vulnerability, mitigation steps and security best practices were recommended, in alignment with OWASP's guidelines.

This concluded the vulnerability assessment methodology portion of this report.