

[F25] Group Project Assignment: Exploiting Vulnerable Web Application with the Help of the Analysis Tools

Link to the GitHub — <https://github.com/sunflye/Juice-Shop-Analysis>

Link to the demo video —

https://drive.google.com/file/d/1HpV6SvUnR2wKGCTBdMcgaDIdAzXx9KPU/view?usp=share_link

Goal

Statically analyse vulnerable Juice Shop web application with **semgrep** tool. Deploy the environment and exploit three found vulnerabilities, suggest possible fixes.

Team (CBS-01) responsibilities

1. Polina Kostikova (p.kostikova@innopolis.university) — prepared the environment and implemented exploit scripts
2. Sofia Palkina (s.palkina@innopolis.university) — wrote this report and proposed defense mechanisms
3. Amir Bairamov (a.bairamov@innopolis.university) — wrote this report, prepared the environment and recorded a demo video

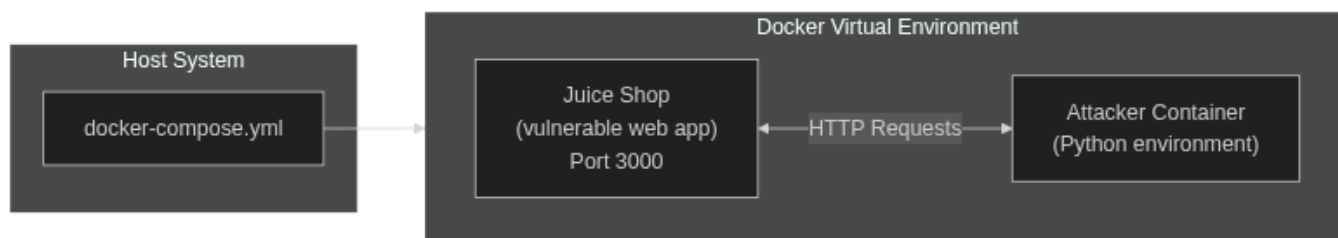
Attack surface and scenarios description

OWASP Juice Shop is probably the most modern and sophisticated insecure web application. It encompasses vulnerabilities from the entire OWASP Top Ten along with many other security flaws found in real-world applications.

Scenario:

- **Launch** the **Juice Shop web application**.
- **Learn** as much as possible about it's vulnerabilities by using **semgrep**.
- **Exploit** 3 vulnerabilities.
- **Suggest** defense mechanisms.

Schema:



Environment preparation

- Ensure you have docker and docker compose plugin installed.

```
docker -v  
docker compose version
```

- Clone this repository.

```
git clone https://github.com/sunflye/Juice-Shop-Analysis.git
```

- Change into the project directory.

```
cd Juice-Shop-Analysis
```

- Start the vulnerable Juice Shop container

```
docker compose up -d juice-shop
```

- Build the attacker container

```
docker compose build attacker
```

Detection of Vulnerabilities

First, find vulnerabilities in the **Juice Shop** source code. Run the **semgrep** tool on it.

To identify vulnerabilities in the Juice Shop source code, we used two approaches with Semgrep:

1. Custom Semgrep Rules

First, we created a custom rules file **semgrep-rules-utf8.yml** to focus on the vulnerabilities that we wanted to find. The custom rules targeted:

- XSS via **innerHTML** and **document.write** (CWE-79)
- Usage of **eval** (CWE-94/98)
- Logging of secrets, passwords, tokens, or keys (CWE-532)
- Missing CSRF protection (CWE-352)
- Possible IDOR in basket operations (CWE-639)

2. Official Semgrep Rules

Next, we decided to run Semgrep with the official JavaScript ruleset (**p/javascript**) to discover additional vulnerabilities that have been missed by our custom rules. This provided a broader view of the application's security posture.

By combining focused custom rules and the broader official ruleset, we ensured both targeted and comprehensive vulnerability discovery in the Juice Shop application.

Note: The Juice Shop repository was cloned into a folder (`juice-shop`) located next to our analysis project folder (`Juice-Shop-Analysis`). All Semgrep scans were run from the parent directory, specifying the source code folder as `juice-shop` and saving reports into the `Juice-Shop-Analysis` directory.

3. Steps to discover vulnerabilities

- Clone the official Juice Shop repository to obtain the source code for analysis:

```
git clone https://github.com/juice-shop/juice-shop.git
```

(Create a `juice-shop` folder next to your analysis project folder)

- Run Semgrep with custom rules on the Juice Shop source code and save the results to a report (run this command from the parent directory, not from inside `Juice-Shop-Analysis`):

```
semgrep --config Juice-Shop-Analysis/semgrep-rules-utf8.yml --json --output Juice-Shop-Analysis/semgrep-report.json ./juice-shop
```

- Then run Semgrep with the official JavaScript rules to identify more vulnerabilities:

```
semgrep --config "p/javascript" --json --output Juice-Shop-Analysis/semgrep-js-report2.json ./juice-shop
```

- Review the generated JSON reports (`semgrep-js-report.json` and `semgrep-js-report2.json`) in the `Juice-Shop-Analysis` folder to identify and prioritize vulnerabilities for exploitation (reports are already in the repository).

Vulnerabilities description

We will describe and continue the attack only with 3 of them: CWE-89, CWE-73, CWE-639.

1. SQL Injection (CWE-89):

- **Description:** The product constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component. Without sufficient removal or quoting of SQL syntax in user-controllable inputs, the generated SQL query can cause those inputs to be interpreted as SQL instead of ordinary user data.
- **CVSSv3 score:** 9.8 (CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H)

- **Reference:** <https://cwe.mitre.org/data/definitions/89.html>

2. Path Traversal (CWE-73):

- **Description:** The product allows user input to control or influence paths or file names that are used in filesystem operations.
- **CVSSv3 score:** 7.5 (CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N)
- **Reference:** <https://cwe.mitre.org/data/definitions/73.html>

3. Insecure Direct Object Reference (IDOR) (CWE-639):

- **Description:** The system's authorization functionality does not prevent one user from gaining access to another user's data or record by modifying the key value identifying the data.
- **CVSSv3 score:** 5.0 (CVSS:3.0/AV:N/AC:L/PR:L/UI:N/S:C/C:L/I:N/A:N)
- **Reference:** <https://cwe.mitre.org/data/definitions/639.html>

Exploits steps

We will exploit selected vulnerabilities: CWE-89, CWE-73, CWE-639.

- Exploit **SQL Injection** (CWE-89).

```
docker compose run attacker python exploit_sql_i.py
```

Target: The search endpoint at `/rest/products/search` with the `q` (query) parameter, and the login endpoint at `/rest/user/login`.

Methods: The script uses a multi-stage approach:

1. Baseline: Sends a normal query (`q=Juice`) to establish expected behavior.
2. Payload Injection: Injects classic SQLi payloads into the `q` parameter to manipulate the underlying SQL query.
 - Boolean-based: `' OR '1'='1` forces a true condition to return all products.
 - Union-based: Attempts `UNION SELECT...` to extract data from other tables.
 - Destructive: Attempts `'; DROP TABLE...` to test for modification capabilities.
3. Login Bypass: Tests SQLi on the login form by injecting payloads into the email field (e.g., `admin' --`) to bypass password checks.

Result: Confirms Boolean-based SQL Injection is successful. The payload `' OR '1'='1` returns 36 products instead of the baseline 29, demonstrating unauthorized data retrieval. Other payloads cause server errors (HTTP 500), indicating flawed input handling but potential defenses or syntax issues.

- Exploit **Path Traversal** (CWE-73).

```
docker compose run attacker python exploit_path_traversal.py
```

Target: The file retrieval endpoint at `/ftp` with the file parameter.

Methods: The script directly exploits path traversal by manipulating the file parameter to escape the intended directory (`./ftp/`). It attempts to read sensitive files from higher-level directories using traversal sequences:

- `../../../../package.json` (application config)
- `../../../../etc/passwd` (system file)
- `../../../../.env` (environment secrets)
- `../../../../config/config.json` (app config)

Result: Confirms Path Traversal vulnerability. The server returned the contents of `package.json`, `.env`, and `config.json`. However, the returned data is wrapped in an HTML error page (listing directory...), which suggests the backend attempts to list a directory instead of returning the raw file, but the traversal itself succeeded. The attempt to read `/etc/passwd` did not return expected content, possibly due to file permissions or absolute path blocking.

- Exploit **IDOR** (CWE-639).

```
docker compose run attacker python exploit_idor_basket.py
```

Target: The basket items API endpoint at `/api/BasketItems/` with the id parameter representing a user's shopping basket ID.

Methods:

1. Authentication: First logs in as a standard user (`user@juice-sh.op`) to obtain a valid JWT token.
2. Enumeration: Systematically iterates through numeric id parameters (e.g., 2, 3, 4...10), using the authenticated session to request basket data.
3. Ownership Analysis: Checks if the returned basket's `BasketId` field matches the id parameter and the current user's own bid. A mismatch indicates access to another user's basket.

Result: Confirms an Insecure Direct Object Reference (IDOR) vulnerability. The script successfully accessed the shopping basket contents of multiple other users (e.g., baskets with IDs 4, 5, 6, 7, 8), proving a complete lack of server-side authorization checks. Authenticated users can view any other user's basket items.

Defense mechanisms

Below are the recommended defense mechanisms for the vulnerabilities identified and exploited during testing: CWE-89, CWE-73, and CWE-639. The recommendations are based on the exploit results we obtained.

Defense mechanisms for CWE-89 (SQL Injection)

1. Eliminate Raw SQL with Unsafe String Concatenation:

The exploit suggests the backend might be constructing queries by directly embedding the `q` parameter into a SQL string (e.g., `"SELECT * FROM products WHERE name LIKE '%" + userInput + "%'"`). This must be replaced with parameterized queries (prepared statements). For Juice Shop's Sequelize ORM, this means using the replacement syntax or query builders that separate data from instructions.

2. Implement Strict Input Validation for the Search Parameter:

Define and enforce a strict whitelist pattern for the `q` parameter. Allow only alphanumeric characters, spaces, and a limited set of safe symbols (e.g., hyphens, apostrophes for product names like O'Juice). Reject any input containing SQL metacharacters like `'`, `"`, `;`, `--`, `#`, `/*`, `*/`, `OR`, `UNION`, `SELECT` outside of a safe context.

3. Configure Database User with Least Privilege:

The application's database user should have the minimum necessary privileges—likely only `SELECT` on the Products table. This would render destructive payloads like `'; DROP TABLE products; --` ineffective, even if injection occurs, as the user would lack `DROP` permission.

4. Implement Robust Error Handling:

Configure the application to return generic, user-friendly error messages (e.g., A search error occurred). The exploit shows detailed SQL errors (`SQLITE_ERROR: near "UNION": syntax error`), which aid attackers. Log the detailed errors internally for debugging instead.

Defense mechanisms for CWE-73 (Path Traversal)

1. Whitelist Allowed Filenames:

Do not accept user input (file parameter) as a path. Instead, maintain a whitelist of safe, known files (e.g., `"legal.md"`, `"acquisitions.md"`). Map the user's request to this list.

2. Use a Built-in Safeguard:

If you must accept a filename, use a basename function (like Node.js's `path.basename()`) to strip any directory components (`../`, `/etc/`) before processing. This neutralizes traversal sequences.

3. Validate Against a Regex Pattern:

Reject any file parameter containing path traversal characters (`..`, `/`, `\`) or sequences like `%2e%2e%2f` (URL-encoded `../`).

4. Apply the Principle of Least Privilege to the Server Process:

The Node.js process running Juice Shop should have read-only permissions strictly for the directories it needs to serve files from (e.g., `./ftp/`). It should have no read access to source directories (`./config/`, `./env`) and no execute permissions anywhere unnecessary.

Defense mechanisms for CWE-639 (IDOR)

1. Enforce Ownership Verification:

For every API request to access a `BasketItem`, the backend must validate that the requested resource belongs to the currently authenticated user (extracted from the JWT token). This check must happen on the server, before any data is fetched from the database.

2. Query Design:

Modify database queries to include the user's identity as a mandatory filter. Example: Instead of `SELECT * FROM BasketItems WHERE id = ?`, use `SELECT * FROM BasketItems WHERE id = ? AND userId = ?` (where `userId` is taken from the JWT session).

3. Replace Direct Database IDs:

Instead of exposing predictable, sequential database IDs (1, 2, 3...) in API parameters, use random, non-guessable identifiers (UUIDs) or encrypted/obfuscated tokens. This makes large-scale IDOR enumeration significantly harder, though it is not a substitute for server-side checks (1-2 mechanisms).

Difficulties faced

There were several challenges that prevented some exploits from working as intended:

- Some exploits did not work because the relevant vulnerabilities were either already fixed or not present in the current version of Juice Shop.
- In certain cases, the server would return a 401/403 error or an empty response, even though documentation or code suggested a possible vulnerability.
- Some exploits failed due to incorrect request formats, invalid parameters, or missing/incorrect authorization tokens.
- A number of PoC scripts required manual adjustments and reworking to match the actual business logic or API structure of the application.

As a result, not all planned exploits were successful, but three exploitations were successfully demonstrated.

New skills acquired during the project

Throughout the project, we gained practical experience in several key areas of cybersecurity. We learned how to **use Semgrep effectively** for static code analysis. Writing and executing exploit scripts deepened our knowledge of vulnerability exploitation techniques, including **attack automation**. Additionally, we developed the ability to analyze exploit outcomes and translate them into actionable security recommendations.

Conclusion

This project provided a hands-on exploration of web application security by analyzing and exploiting vulnerabilities in the OWASP Juice Shop. Using Semgrep for static analysis, we identified critical weaknesses such as SQL injection, path traversal, and insecure direct object references. By deploying a controlled environment and executing targeted exploits, we demonstrated how these vulnerabilities could be leveraged in real-world scenarios. The suggested defense mechanisms highlight the importance of proactive security practices in software development.