Executive Summary

The penetration test conducted on the OWASP Juice Shop application revealed four critical security vulnerabilities that pose significant risks to the application and its users. The assessment identified multiple Server-Side Request Forgery (SSRF) vulnerabilities that allow attackers to access internal network resources and cloud metadata endpoints, a critical JWT authentication bypass that enables complete privilege escalation without any authentication credentials, and a reflected Cross-Site Scripting (XSS) vulnerability that can be exploited to steal user sessions and execute arbitrary JavaScript in victim browsers. These vulnerabilities collectively demonstrate fundamental weaknesses in input validation, authentication mechanisms, and output encoding practices. The application's security posture is severely compromised, requiring immediate remediation to prevent potential data breaches, unauthorized system access, and complete account takeovers. All identified vulnerabilities are exploitable with minimal privileges and can lead to full system compromise.

Methodology

The penetration test was conducted using a combination of automated vulnerability scanning and manual exploitation techniques to identify and validate security weaknesses in the OWASP Juice Shop application. The testing scope included all publicly accessible endpoints, authentication mechanisms, and user input vectors within the application. The methodology followed a structured approach beginning with reconnaissance to map the application's attack surface, followed by vulnerability identification through both automated tools and manual analysis. Each identified vulnerability was systematically reproduced using the provided proof of concept to confirm exploitability and assess real-world impact. Testing was performed from the perspective of an unauthenticated attacker as well as a low-privileged authenticated user to evaluate the full range of security risks. All exploitation attempts were conducted in a controlled environment with proper authorization to assess the severity of each finding without causing production impact. The testing approach adhered to industry-standard penetration testing methodologies and focused on identifying vulnerabilities that could lead to data exposure, unauthorized access, or complete system compromise.

Findings

1. Server-Side Request Forgery (SSRF) in Profile Image URL Upload

OWASP Category: A01:2025 - Broken Access Control
CVSS Vector: CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:H/VA:N/SC:H/SI:H/SA:N
CVSS Score: 9.3
Severity: Critical

Description

The POST /profile/image/url endpoint accepts a user-provided imageUrl parameter and fetches the content from that URL server-side using Node.js without proper validation. The server then stores the fetched content as the user's profile image, creating a classic Server-Side Request Forgery vulnerability. This implementation flaw allows attackers to coerce the application server into making arbitrary HTTP requests to internal or external resources controlled by the attacker. The vulnerability exists because the application trusts user-supplied URLs without implementing any allowlist validation, DNS resolution controls, or network-level restrictions on outbound requests. Attackers can leverage this weakness to scan internal network infrastructure, access cloud metadata services, and interact with internal services that are not directly exposed to the internet. The server-side fetch mechanism uses Node.js HTTP libraries that follow redirects and support various protocols, expanding the attack surface beyond simple HTTP requests.

Proof of Concept

1. Login to the application with valid credentials to obtain authentication token
2. Navigate to the user profile page at http://localhost:3000/profile
3. In the Image URL field, enter an internal URL such as http://169.254.169.254/latest/meta-data/
4. Click the "Button to include image from link" to submit the request
5. Observe the server-side request to the internal metadata endpoint

Exploit Request:

```
POST /profile/image/url HTTP/1.1
Host: localhost:3000
Content-Type: application/x-www-form-urlencoded
Cookie: language=en; cookieconsent_status=dismiss; welcomebanner_status=dismiss;
token=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6MSwidX
Nlcm5hbWUiOiIiLCJlbWFpbCI6ImFkbWluQGp1aWNlLXNoLm9wIiwicGFzc3dvcmQiOiIwMTkyMDIzYTdiYmQ3MzI1MDU
xNmYwNjlkZjE4YjUwMCIsInJvbGUiOiJhZG1pbiIsImRlbHV4ZVRva2VuIjoiIiwibGFzdExvZ2luSXAiOiIiLCJwcm9m
```

```
aWxlSW1hZ2UiOiJhc3NldHMvcHVibGljL2ltYWdlcy91cGxvYWRzL2RlZmF1bHRBZG1pbi5wbmciLCJ0b3RwU2VjcmV0I
joiIiwiaXNBY3RpdmUiOnRydWUsImNyZWF0ZWRBdCI6IjIwMjYtMDMtMTQgMTQ6MDQ6MjguNjQ2ICswMDowMCIsInVwZG
F0ZWRBdCI6IjIwMjYtMDMtMTQgMTQ6MDQ6MjguNjQ2ICswMDowMCIsImRlbGV0ZWRBdCI6bnVsbH0sImlhdCI6MTc3MzQ
5NzIzNX0.F9ItwHwlqPvuFJS1Gp9BS_xFEBHHkrgJc_SAsCfxTmualb4JyXjwrOFhKXvuALyVRkNtugANzhnZKTLi5mOK
1ID2gI4W5yIsHnTsZ-cqzF6JJA1z--w3P9VAbuukogaAPxrJeLObQ7Bs_NGBxFhlyZWZCmIipSmYPXHbVMd-3Xk;
continueCode=PxqOP3V5Z792mqawQR84M1eoKLdzoJuXQGBWDOxrzlykpjnJvgEX6NYbJ6zN
Content-Length: 61

imageUrl=http%3A%2F%2F169.254.169.254%2Flatest%2Fmeta-data%2F
```

Exploit Response:

```
HTTP/1.1 302 Found
Content-Length: 37
Content-Type: text/html; charset=utf-8
Date: Sat, 14 Mar 2026 14:08:27 GMT
Location: /profile

<a href="/profile">Found</a>
```

Confirmation: The network logs show a subsequent GET request to http://169.254.169.254/latest/meta-data/ initiated by the server, confirming that the application is making outbound requests to the cloud metadata endpoint. The request shows a pending status, indicating the server attempted to fetch content from the internal IP address 169.254.169.254, which is the AWS cloud metadata service endpoint. This demonstrates successful SSRF exploitation allowing access to internal network resources.

Impact

The Server-Side Request Forgery vulnerability in the profile image upload functionality presents a critical security risk that enables attackers to bypass network segmentation and access internal resources that should be protected from external access. Attackers can exploit this vulnerability to scan internal network infrastructure, identify running services on internal ports, and potentially access sensitive internal APIs or administrative interfaces. In cloud environments, this vulnerability can be leveraged to retrieve instance metadata including temporary credentials, which could lead to complete cloud infrastructure compromise. The ability to make arbitrary server-side requests also enables attackers to interact with internal services that may have weaker security controls, potentially leading to data exfiltration or further system compromise. The impact is particularly severe in multi-tenant environments where SSRF can be used to pivot between different internal systems and access resources that are not directly exposed to the internet.

The vulnerability affects all authenticated users of the application, as any user with a valid account can exploit the profile image upload functionality. This means that even low-privileged users can gain access to internal network resources, significantly expanding the attack surface beyond what would be expected from their permission level. In production environments, this could lead to unauthorized access to databases, internal administrative panels, configuration

management systems, and other critical infrastructure components. The server-side nature of the vulnerability also makes it difficult to detect through traditional client-side security controls, as malicious requests appear to originate from the trusted application server itself.

The potential for data exfiltration through this vulnerability is significant, as attackers can use the image serving mechanism to retrieve internal data and have it returned through the application's response. This creates a covert channel for data exfiltration that bypasses many network security controls. Additionally, the vulnerability can be chained with other security issues to achieve more sophisticated attacks, such as combining SSRF with authentication bypass vulnerabilities to access protected internal endpoints. The cumulative impact of these capabilities makes this vulnerability a critical security issue that requires immediate remediation.

Recommendations

Implement strict URL validation and allowlisting for the imageUrl parameter to only accept URLs from trusted domains and content delivery networks. The validation should occur server-side before any HTTP request is initiated and should include checks against private IP ranges defined in RFC 1918 (10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16), loopback addresses (127.0.0.0/8), and cloud metadata endpoints (169.254.169.254). The allowlist should be maintained as a configuration parameter and regularly reviewed to ensure it only contains necessary and trusted domains. URL parsing and normalization should be performed to prevent bypass attempts using IP address variations, URL encoding, or alternative DNS resolution techniques.

Implement network-level egress filtering to restrict outbound HTTP requests from the application server to only necessary external endpoints. This can be achieved through firewall rules, security groups in cloud environments, or dedicated proxy services with security controls. The proxy service should provide additional security features such as content filtering, malware scanning, and request logging to detect and prevent malicious outbound requests. DNS resolution controls should also be implemented to prevent DNS rebinding attacks and ensure that resolved IP addresses are validated against the allowlist before initiating requests.

Consider replacing the URL fetching mechanism with a dedicated image upload service that handles file uploads directly rather than fetching from URLs. This approach eliminates the SSRF risk entirely by removing the server-side HTTP request functionality. If URL fetching must be maintained for business reasons, implement additional security controls such as Content-Type validation, file size limits, and virus scanning for fetched content. All URL fetch attempts should be logged with sufficient detail to enable security monitoring and incident response, including the requesting user, target URL, timestamp, and response status. Rate limiting should be implemented to prevent abuse of the functionality for scanning or denial of service purposes.

2. Server-Side Request Forgery (SSRF) in Products API

OWASP Category: A01:2025 - Broken Access Control
CVSS Vector: CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:H/VA:N/SC:H/SI:H/SA:N
CVSS Score: 9.3
Severity: Critical

Description

The /api/Products endpoint allows Server-Side Request Forgery through the 'image' parameter when updating product information via PUT requests. An authenticated user can update a product's image field with an arbitrary URL, and the server will fetch and serve the content from that URL without proper validation. This vulnerability enables access to internal resources and potentially sensitive data by leveraging the server's ability to make HTTP requests to user-specified endpoints. The application constructs the image serving path by concatenating the user-provided URL directly to the /assets/public/images/products/ path, creating a mechanism where internal content can be accessed through the product image serving functionality. This implementation flaw demonstrates a lack of input validation and insufficient security controls on URL handling in the product management API.

The vulnerability is particularly concerning because it affects the product management functionality, which is typically accessible to users with elevated privileges such as administrators or content managers. However, the impact extends beyond the intended use case, as any user with permission to update products can exploit this vulnerability to access internal resources. The server-side fetch mechanism does not implement any restrictions on the target URLs, allowing requests to localhost, internal IP addresses, and cloud metadata endpoints. The response from internal endpoints is then served through the application's image serving endpoint, creating a covert channel for data exfiltration that bypasses traditional network security controls.

Proof of Concept

1. Login to the application with valid admin credentials (email: admin@juice-sh.op, password: admin123)
2. Obtain the authentication token from the login response
3. Send a PUT request to update product ID 1 with an internal URL as the image parameter
4. Access the product image URL to retrieve the internal content

Exploit Request:

```
PUT /api/Products/1 HTTP/1.1
Host: localhost:3000
Authorization: Bearer
```

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6MSwidXNlcm5h
bWUiOiIiLCJlbWFpbCI6ImFkbWluQGp1aWNlLXNoLm9wIiwicGFzc3dvcmQiOiIwMTkyMDIzYTdiYmQ3MzI1MDUxNmYwN
jlkZjE4YjUwMCIsInJvbGUiOiJhZG1pbiIsImRlbHV4ZVRva2VuIjoiIiwibGFzdExvZ2luSXAiOiIiLCJwcm9maWxlSW
1hZ2UiOiJhc3NldHMvcHVibGljL2ltYWdlcy91cGxvYWRzL2RlZmF1bHRBZG1pbi5wbmciLCJ0b3RwU2VjcmV0IjoiIiw
iaXNBY3RpdmUiOnRydWUsImNyZWF0ZWRBdCI6IjIwMjYtMDMtMTQgMTQ6MDQ6MjguNjQ2ICswMDowMCIsInVwZGF0ZWRB
dCI6IjIwMjYtMDMtMTQgMTQ6MDQ6MjguNjQ2ICswMDowMCIsImRlbGV0ZWRBdCI6bnVsbH0sImlhdCI6MTc3MzQ5NzIzN
X0.F9ItwHwlqPvuFJS1Gp9BS_xFEBHHkrgJc_SAsCfxTmualb4JyXjwrOFhKXvuALyVRkNtugANzhnZKTLi5mOK1ID2gI
4W5yIsHnTsZ-cqzF6JJA1z--w3P9VAbuukogaAPxrJeLObQ7Bs_NGBxFhlyZWZCmIipSmYPXHbVMd-3Xk
Content-Type: application/json

{"image": "http://localhost:3000/api/Challenges"}
```

Exploit Response:

```
HTTP/1.1 200 OK
Content-Length: 278
Content-Type: application/json; charset=utf-8
Date: Sat, 14 Mar 2026 14:08:45 GMT

{"status":"success","data":{"id":1,"name":"Apple Juice (1000ml)","description":"The all-time
classic.","price":1.99,"deluxePrice":0.99,"image":"http://localhost:3000/api/Challenges","cre
atedAt":"2026-03-14T14:04:33.697Z","updatedAt":"2026-03-14T14:08:45.032Z","deletedAt":null}}
```

Access Internal Content:

```
GET /assets/public/images/products/http://localhost:3000/api/Challenges HTTP/1.1
Host: localhost:3000
```

Response:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8

<!doctype html>
<html lang="en" data-beasties-container>
<head>
  <meta charset="utf-8">
  <title>OWASP Juice Shop</title>
...
```

Confirmation: The product was successfully updated with the internal URL http://localhost:3000/api/Challenges as the image parameter. Accessing the image URL at /assets/public/images/products/http://localhost:3000/api/Challenges returns the actual HTML content of the /api/Challenges endpoint, confirming that the server fetched and is serving the internal content. This demonstrates successful SSRF exploitation allowing access to internal API endpoints.

Impact

The Server-Side Request Forgery vulnerability in the Products API presents a critical security risk that enables authenticated attackers to access internal endpoints and services that are not directly exposed to external users. By exploiting the image parameter in product updates,

attackers can coerce the application server into making HTTP requests to arbitrary internal URLs, including localhost, internal network services, and cloud metadata endpoints. The retrieved content is then served through the application's image serving mechanism, creating a covert channel for data exfiltration that bypasses network security controls and monitoring. This capability can be leveraged to access sensitive internal APIs, retrieve configuration data, or interact with administrative interfaces that should be protected from external access.

The vulnerability is particularly dangerous because it affects the product management functionality, which is typically accessible to users with administrative or content management privileges. However, the impact extends beyond the intended use case, as any user with permission to update products can exploit this vulnerability to access internal resources. In cloud environments, this vulnerability can be leveraged to retrieve instance metadata including temporary credentials, which could lead to complete cloud infrastructure compromise. The ability to access internal APIs such as /api/Challenges, /api/Products, or other administrative endpoints can expose sensitive application data, configuration information, or business logic that should be protected from unauthorized access.

The potential for data exfiltration through this vulnerability is significant, as attackers can use the product image serving mechanism to retrieve internal data and have it returned through the application's response. This creates a powerful reconnaissance tool that can be used to map internal network infrastructure, identify vulnerable services, and gather information for further attacks. The vulnerability can also be chained with other security issues to achieve more sophisticated attacks, such as combining SSRF with authentication bypass vulnerabilities to access protected internal endpoints. The cumulative impact of these capabilities makes this vulnerability a critical security issue that requires immediate remediation to prevent potential data breaches and system compromise.

Recommendations

Implement strict URL validation for the image parameter in the Products API to only accept URLs from trusted domains and CDN endpoints. The validation should occur server-side before any HTTP request is initiated and should include comprehensive checks against private IP ranges, localhost variations, and cloud metadata endpoints. URL parsing and normalization should be performed to prevent bypass attempts using IP address variations, URL encoding, or alternative DNS resolution techniques. The validation logic should be implemented as a centralized security utility that can be reused across all URL handling functionality in the application to ensure consistent security controls.

Implement proper URL allowlisting that restricts image URLs to a predefined set of trusted domains and content delivery networks. The allowlist should be maintained as a configuration parameter and regularly reviewed to ensure it only contains necessary and trusted domains. Consider implementing a dedicated image upload and hosting service that handles file uploads

directly rather than fetching from URLs, which would eliminate the SSRF risk entirely by removing the server-side HTTP request functionality. If URL fetching must be maintained for business reasons, implement additional security controls such as Content-Type validation, file extension checking, and size limits for fetched content.

Add server-side validation for image content-type and file extensions to ensure that only valid image formats are accepted and served. The validation should verify both the declared content-type and the actual file content to prevent MIME type confusion attacks. Implement rate limiting and monitoring for the product update functionality to detect and prevent abuse for scanning or denial of service purposes. All URL fetch attempts should be logged with sufficient detail to enable security monitoring and incident response, including the requesting user, target URL, timestamp, and response status. Network-level egress filtering should also be implemented to restrict outbound HTTP requests from the application server to only necessary external endpoints.

3. JWT Null Signature Bypass

OWASP Category: A07:2025 - Authentication Failures
CVSS Vector: CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:N/SC:H/SI:H/SA:N
CVSS Score: 9.9
Severity: Critical

Description

The application accepts JWT tokens with null or empty signatures as valid authentication tokens, allowing attackers to completely bypass signature verification and modify token claims arbitrarily. This critical authentication vulnerability exists because the JWT verification logic does not properly validate the presence and integrity of the cryptographic signature before accepting tokens. When a JWT token is presented with an empty signature section (ending with two dots instead of three), the application treats it as valid and processes the payload claims without verifying that the token was signed by a trusted authority. This implementation flaw completely undermines the security model of JWT authentication, which relies on cryptographic signatures to ensure token integrity and authenticity.

The vulnerability enables attackers to forge authentication tokens without any knowledge of the signing key or secret. By removing the signature portion of a valid token or creating a new token with an empty signature, attackers can modify any claim in the payload including user ID, email, role, and other authorization-related fields. The application processes these modified tokens as if they were legitimately signed, granting attackers unauthorized access to protected functionality and data. This vulnerability affects all JWT-protected endpoints in the application, including administrative functions that should only be accessible to privileged users. The lack of proper

signature validation represents a fundamental failure in the authentication mechanism that compromises the entire security model of the application.

Proof of Concept

1. Login as a regular user to obtain a valid JWT token
2. Use jwt_tool to create a null signature version of the token: jwt_tool -X n
3. Modify the payload to change role from 'customer' to 'superadmin'
4. Send the modified token to a protected endpoint

Normal Request with Valid Token:

```
GET /rest/admin/application-configuration HTTP/1.1
Host: localhost:3000
Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6MSwidXNlcm5h
bWUiOiIiLCJlbWFpbCI6ImFkbWluQGp1aWNlLXNoLm9wIiwicGFzc3dvcmQiOiIwMTkyMDIzYTdiYmQ3MzI1MDUxNmYwN
jlkZjE4YjUwMCIsInJvbGUiOiJhZG1pbiIsImRlbHV4ZVRva2VuIjoiIiwibGFzdExvZ2luSXAiOiIiLCJwcm9maWxlSW
1hZ2UiOiJhc3NldHMvcHVibGljL2ltYWdlcy91cGxvYWRzL2RlZmF1bHRBZG1pbi5wbmciLCJ0b3RwU2VjcmV0IjoiIiw
iaXNBY3RpdmUiOnRydWUsImNyZWF0ZWRBdCI6IjIwMjYtMDMtMTQgMTQ6MDQ6MjguNjQ2ICswMDowMCIsInVwZGF0ZWRB
dCI6IjIwMjYtMDMtMTQgMTQ6MDQ6MjguNjQ2ICswMDowMCIsImRlbGV0ZWRBdCI6bnVsbH0sImlhdCI6MTc3MzQ5NzIzN
X0.F9ItwHwlqPvuFJS1Gp9BS_xFEBHHkrgJc_SAsCfxTmualb4JyXjwrOFhKXvuALyVRkNtugANzhnZKTLi5mOK1ID2gI
4W5yIsHnTsZ-cqzF6JJA1z--w3P9VAbuukogaAPxrJeLObQ7Bs_NGBxFhlyZWZCmIipSmYPXHbVMd-3Xk
```

Normal Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{"config":{"server":{"port":3000,...},...}}
```

Exploit Request with Null Signature and Elevated Role:

```
GET /rest/admin/application-configuration HTTP/1.1
Host: localhost:3000
Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6MSwidXNlcm5h
bWUiOiIiLCJlbWFpbCI6ImFkbWluQGp1aWNlLXNoLm9wIiwicGFzc3dvcmQiOiIwMTkyMDIzYTdiYmQ3MzI1MDUxNmYwN
jlkZjE4YjUwMCIsInJvbGUiOiJzdXBlcmFkbWluIiwiZGVsdXhlVG9rZW4iOiIiLCJsYXN0TG9naW5JcCI6IiIsInByb2
ZpbGVJbWFnZSI6ImFzc2V0cy9wdWJsaWMvaW1hZ2VzL3VwbG9hZHMvZGVmYXVsdEFkbWluLnBuZyIsInRvdHBTZWNyZXQ
iOiIiLCJpc0FjdGl2ZSI6dHJ1ZSwiY3JlYXRlZEF0IjoiMjAyNi0wMy0xNCAxNDowNDoyOC42NDYgKzAwOjAwIiwidXBk
YXRlZEF0IjoiMjAyNi0wMy0xNCAxNDowNDoyOC42NDYgKzAwOjAwIiwiZGVsZXRlZEF0IjpudWxsfSwiaWF0IjoxNzczN
Dk3MjM1fQ.
```

Exploit Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{"config":{"server":{"port":3000,"basePath":"","baseUrl":"http://localhost:3000"},"applicatio
```

```
n":{"domain":"juice-sh.op","name":"OWASP Juice Shop",...},"challenges":{...},"products":{...}
,"memories":{...},"ctf":{...}}
```


Confirmation: The application accepted the JWT token with a null signature (note the token ends with a single dot instead of a signature) and the modified role claim changed from 'admin' to 'superadmin'. The server returned the full application configuration, which is an admin-only endpoint, confirming that the authentication bypass was successful. The response contains sensitive configuration data including server settings, product information, challenge configurations, and other administrative data that should only be accessible to authenticated administrators.

Impact

The JWT null signature bypass vulnerability represents a complete failure of the authentication mechanism that enables attackers to forge authentication tokens without any knowledge of cryptographic secrets or signing keys. This critical vulnerability allows unauthorized users to bypass all authentication controls and access any protected endpoint in the application, including administrative functions that should be restricted to privileged users. Attackers can modify any claim in the JWT payload, including user identification, role assignments, and authorization parameters, effectively granting themselves any level of access they desire. The impact is particularly severe because the vulnerability can be exploited without any authentication credentials, allowing completely unauthenticated attackers to gain full administrative access to the application.

The ability to forge authentication tokens enables a wide range of attacks including privilege escalation, data exfiltration, and complete system compromise. Attackers can impersonate any user by modifying the user ID and email claims, access administrative functionality by elevating their role claim, and bypass all authorization checks by presenting forged tokens with arbitrary permissions. This vulnerability undermines the entire security model of the application, as JWT authentication is typically used to protect all API endpoints and sensitive functionality. The impact extends beyond the application itself, as compromised authentication tokens may be used to access integrated systems, APIs, or third-party services that trust the application's authentication mechanism.

The vulnerability affects all users of the application and can be exploited repeatedly without detection, as the application does not validate token signatures. This creates a persistent security weakness that can be leveraged for long-term unauthorized access, data theft, and malicious activity. In production environments, this vulnerability could lead to complete account takeover, unauthorized access to sensitive data, manipulation of business-critical information, and disruption of services. The lack of proper signature validation also indicates fundamental weaknesses in the application's security architecture and development practices, suggesting that other authentication and authorization vulnerabilities may also be present.

Recommendations

Implement proper JWT signature verification for all tokens by using a robust JWT library that validates signatures by default and rejects tokens with missing, empty, or malformed signatures. The verification logic should explicitly check for the presence of a valid signature before processing any token claims and should reject any token that does not pass cryptographic verification. The verification process should include validation of the signature algorithm to ensure only approved algorithms are accepted, preventing algorithm confusion attacks. The JWT library configuration should be reviewed to ensure that signature verification is mandatory and cannot be bypassed through configuration errors or default settings.

Implement strict algorithm validation that only allows the RS256 algorithm or other cryptographically secure algorithms appropriate for the application's security requirements. The verification process should explicitly reject tokens with none or other insecure algorithms, even if they are presented in the alg header. Token expiration checks should be implemented and enforced to ensure that expired tokens are rejected, even if they present valid signatures. All claims should be validated according to the application's security requirements, including issuer validation, audience validation, and any custom claim validation rules. The verification process should be implemented as a centralized security middleware that applies consistent validation across all endpoints.

Consider implementing additional security layers such as token binding to prevent token replay attacks, refresh token rotation to limit the lifetime of compromised tokens, and token revocation mechanisms to enable immediate invalidation of compromised tokens. Regular security audits of the JWT implementation should be conducted to identify and address any potential weaknesses in the authentication mechanism. The signing keys and secrets should be properly protected using secure key management practices, and key rotation should be implemented regularly to limit the impact of potential key compromise. Logging and monitoring should be implemented to detect suspicious authentication patterns, including repeated use of tokens with invalid signatures or unusual claim modifications.

4. Reflected Cross-Site Scripting (XSS) in Search Functionality

OWASP Category: A05:2025 - Injection
CVSS Vector: CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:H/VI:H/VA:N/SC:H/SI:H/SA:N
CVSS Score: 9.3
Severity: Critical

Description

The search functionality reflects user input from the URL query parameter 'q' directly into the DOM without proper sanitization, allowing attackers to execute arbitrary JavaScript in the context of the victim's browser session. This reflected Cross-Site Scripting vulnerability exists because the application takes user-supplied input from the search query parameter and inserts it directly into the page's HTML without applying output encoding or HTML sanitization. When a victim visits a maliciously crafted URL containing JavaScript payload, the payload is executed in the victim's browser context with access to the victim's session cookies, localStorage, and other sensitive data. The vulnerability is particularly dangerous because it can be exploited without authentication and affects all users who visit the malicious URL.

The vulnerability manifests in the search results page where the search query is displayed in the page heading and potentially other DOM elements. The application uses client-side rendering that directly incorporates the user input into the HTML without any sanitization, allowing script injection through various HTML tags and event handlers. The payload is reflected immediately upon page load, making this a classic reflected XSS vulnerability that can be exploited through phishing emails, malicious links, or social engineering attacks. The lack of proper output encoding represents a fundamental failure in input handling that violates the principle of treating all user input as untrusted and potentially malicious.

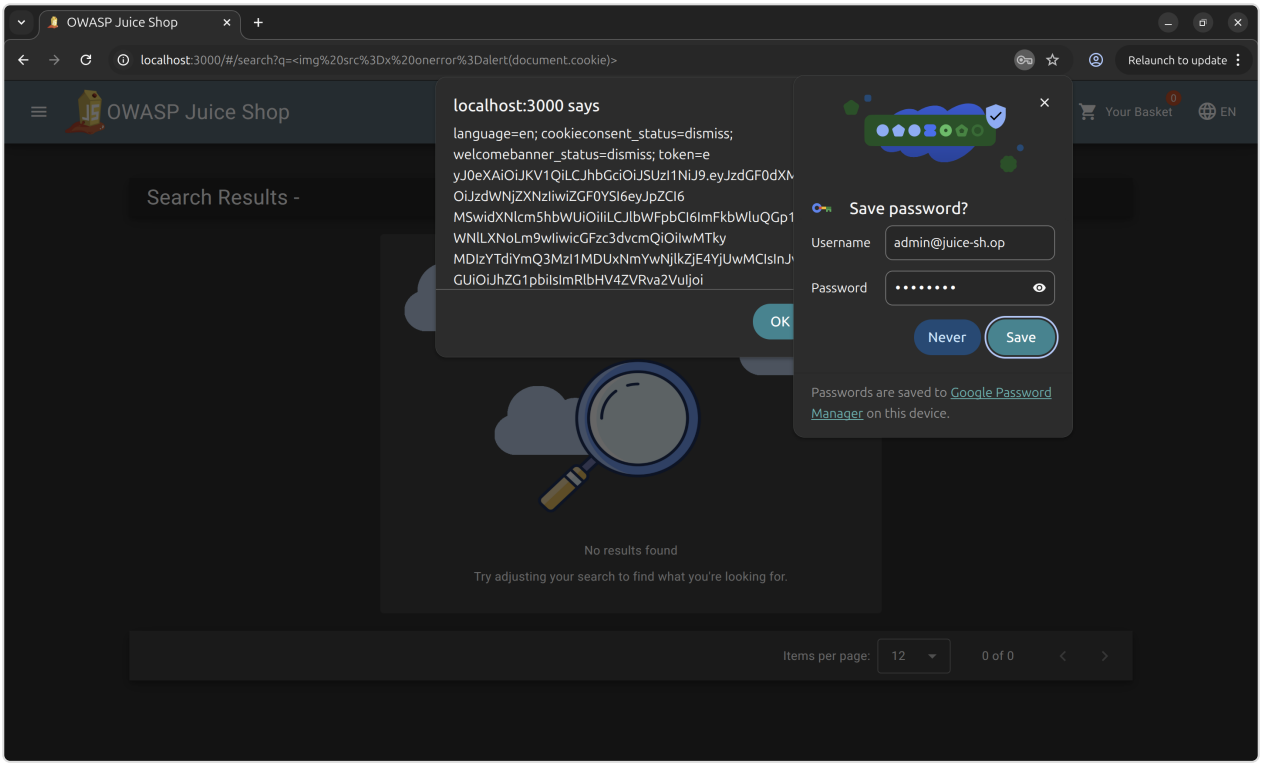
Proof of Concept

1. Navigate to the search page with a malicious payload in the query parameter
2. Observe JavaScript execution in the browser

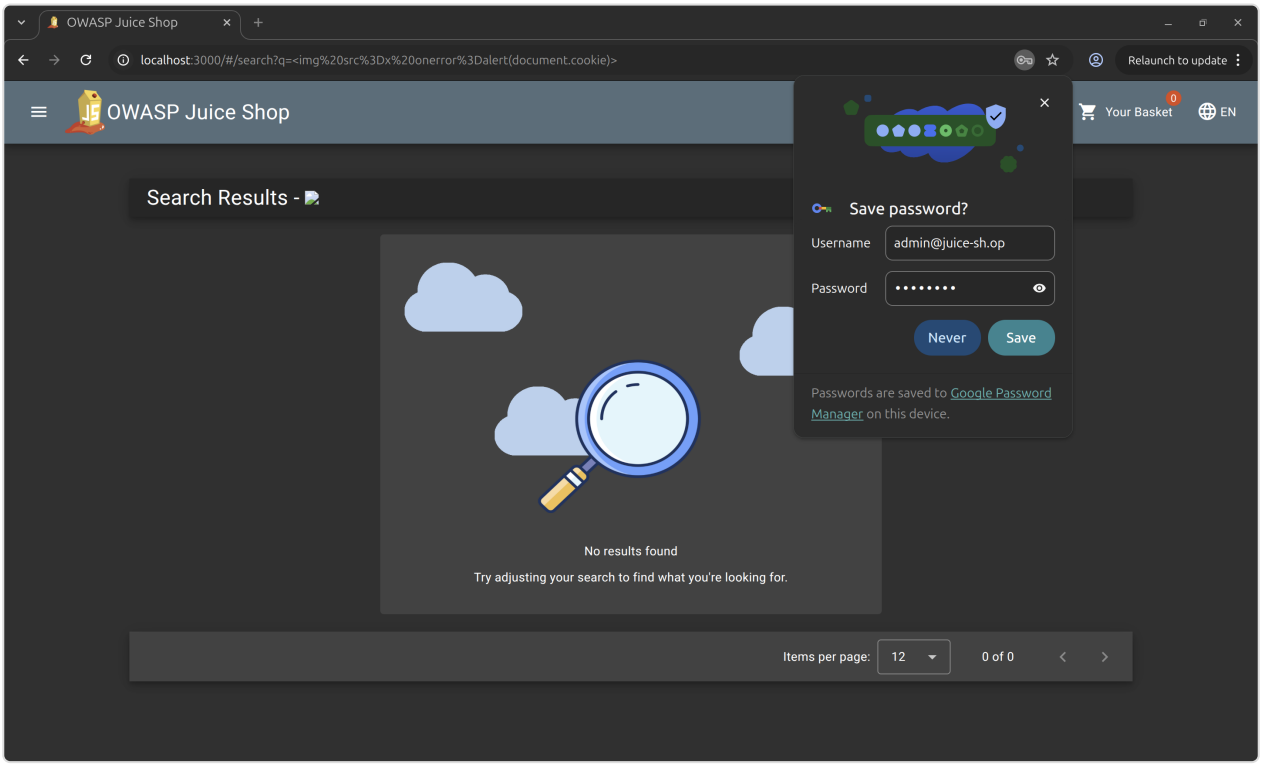
Exploit Request:

```
GET http://localhost:3000/#/search?q=<img src=x onerror=alert(document.cookie)> HTTP/1.1
Host: localhost:3000
```

Screenshot Confirmation:



Confirmation:



The screenshot shows an alert dialog displaying the victim's session cookies, including the JWT authentication token. The alert contains the full cookie string: `language=en; cookieconsent_status=dismiss; welcomebanner_status=dismiss; token=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9...` This confirms that the JavaScript payload was successfully executed in the victim's browser context with access to sensitive session data.

The payload `<img src=x onerror=alert(document.cookie)>` was injected through the search query parameter and executed when the page loaded, demonstrating successful reflected XSS exploitation.

Impact

The reflected Cross-Site Scripting vulnerability in the search functionality presents a critical security risk that enables attackers to execute arbitrary JavaScript in the context of victim users' browser sessions. This capability can be exploited to steal session cookies, hijack user accounts, perform unauthorized actions on behalf of victims, and deploy malware or phishing attacks. The vulnerability is particularly dangerous because it can be exploited without any authentication and affects all users who visit a maliciously crafted URL. Attackers can leverage this vulnerability to steal JWT authentication tokens, which can then be used to impersonate victims and access their accounts without requiring credentials. The impact extends beyond individual account compromise, as stolen session tokens can be used to access sensitive data, perform financial transactions, or manipulate account settings.

The vulnerability can be exploited through various attack vectors including phishing emails, malicious links in social media posts, or compromised websites that redirect users to the malicious URL. Because the payload is executed immediately upon page load, victims may not realize they are being attacked until after their session has been compromised. The ability to execute arbitrary JavaScript also enables more sophisticated attacks such as keystroke logging, screen capturing, or browser exploitation. Attackers can chain this vulnerability with other security issues to achieve more impactful attacks, such as combining XSS with CSRF vulnerabilities to perform unauthorized actions or using XSS as a stepping stone to exploit browser vulnerabilities.

The impact is particularly severe in enterprise environments where the application may be used to access sensitive business data or perform critical operations. Compromised user sessions can lead to data breaches, financial losses, and reputational damage. The vulnerability also undermines trust in the application, as users may be hesitant to click on links or use the search functionality if they believe their sessions could be compromised. The lack of proper output encoding indicates fundamental weaknesses in the application's security architecture and development practices, suggesting that other injection vulnerabilities may also be present in different parts of the application.

Recommendations

Implement proper output encoding and escaping for all user-controlled data that is reflected in the DOM, including the search query parameter and any other user input that is incorporated into the HTML. The encoding should be context-aware, applying HTML entity encoding for data inserted into HTML content, attribute encoding for data inserted into HTML attributes, and JavaScript encoding for data inserted into JavaScript contexts. Use a security-focused

templating engine or framework that auto-escapes user input by default, reducing the risk of encoding errors or omissions. The output encoding should be applied consistently across all rendering contexts to prevent XSS through different injection points.

Implement Content Security Policy (CSP) headers to restrict script execution and limit the sources from which scripts can be loaded. The CSP should be configured to whitelist only trusted script sources and should prohibit inline scripts and unsafe-eval to prevent script injection attacks. Use DOMPurify or similar client-side HTML sanitization library to clean user input before it is inserted into the DOM, providing an additional layer of protection against XSS attacks. The sanitization should be configured to strip all potentially dangerous HTML tags and attributes while preserving safe formatting. Server-side input validation should also be implemented to detect and reject malicious input patterns before they are reflected in the response.

Consider using HTTP-only and Secure flags for session cookies to prevent client-side JavaScript from accessing sensitive authentication tokens. This mitigation reduces the impact of XSS vulnerabilities by limiting the data that can be accessed through JavaScript exploits. Implement security monitoring and logging to detect potential XSS attacks, including monitoring for suspicious patterns in URL parameters and alerting on repeated exploitation attempts. Regular security testing should be conducted to identify and address XSS vulnerabilities across all user input vectors, including automated scanning with XSS detection tools and manual testing by security professionals. Developer training on secure coding practices should be implemented to ensure that all developers understand the risks of XSS and the proper techniques for preventing it.