📘**JWT Forgery Exploit Report –**

OSWAP JUICE SHOP

TITLE : *WT Forgery Exploit in OWASP Juice Shop*

INTERNSHIP AT : FUTURE INTERN  
 REPORT BY : FATIMA RIYA V

**Environment:**

* Target: [https://demo.owasp-juice.shop](https://demo.owasp-juice.shop/)
* Tools: Burp Suite, jwt.io, Canva

💡 *Visual Tip:* Add Juice Shop logo + lock icon. Use fonts like Roboto or Lato for clean presentation.

**Executive Summary**

This report documents a critical vulnerability in OWASP Juice Shop’s authentication mechanism. By modifying a JWT and re-signing it using a publicly known secret, it was possible to bypass authorization controls, escalate privileges, and retrieve admin-only data. The exploit demonstrates practical flaws in token validation and highlights

🎯 **Objective**

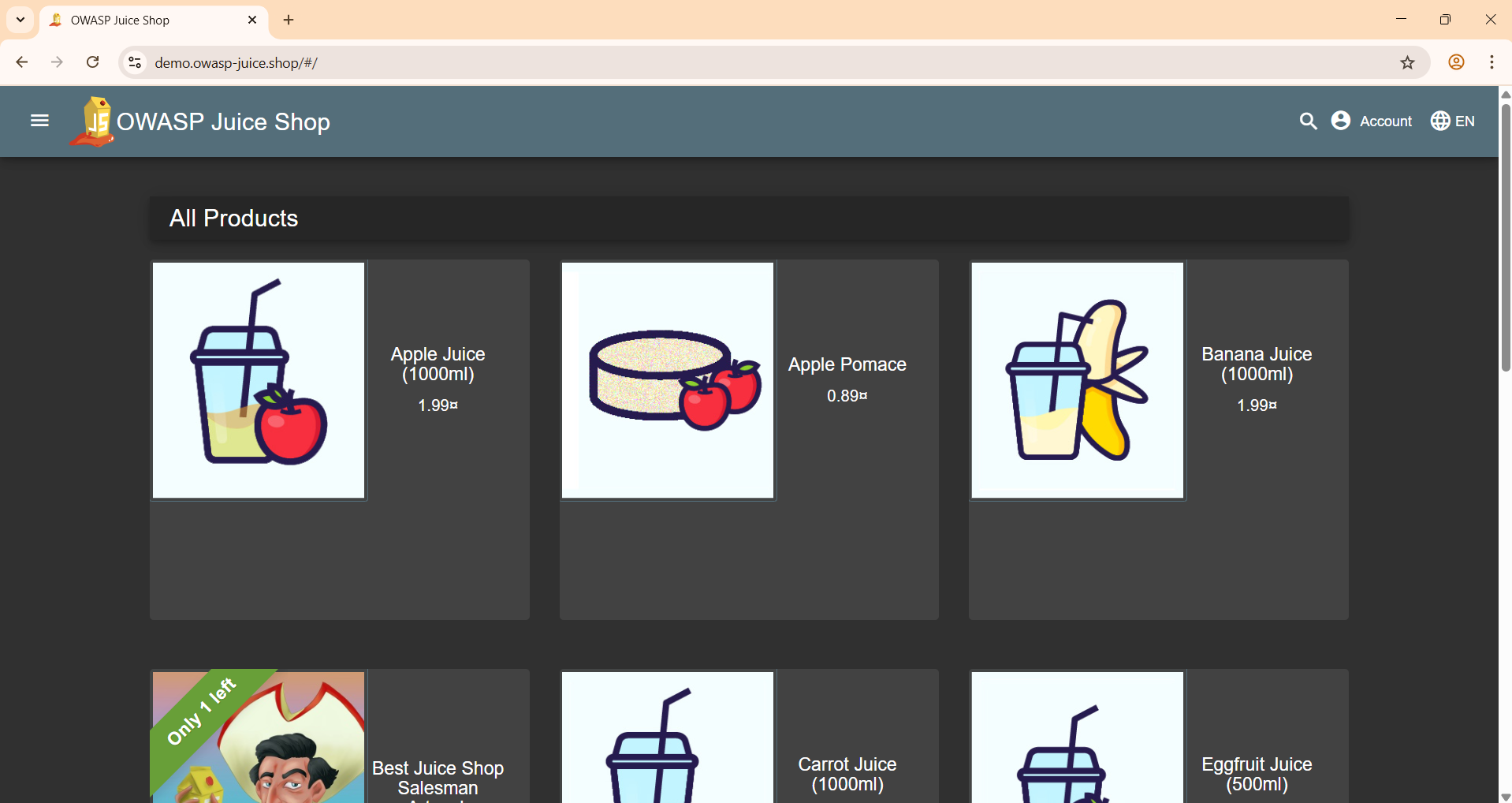
The objective of this report is to demonstrate the exploitation of a vulnerability in the OWASP Juice Shop application by forging a JSON Web Token (JWT) to escalate privileges. This assessment aims to highlight the risks associated with insecure JWT implementations, illustrate practical ethical hacking techniques, and provide actionable mitigation strategies for securing token-based authentication systems.

🛠️ Environment Setup

|  |  |
| --- | --- |
| Component | description |
| Target | OWASP Juice Shop Demo (Heroku) |
| OS | Windows 11 |
| Browser | Chrome |
| Tools Used | Burp Suite, jwt.io |
| Testing Method | Black-box ethical testing |

**Target Application:**

* OWASP Juice Shop (Public Demo)
* URL: [https://demo.owasp-juice.shop](https://demo.owasp-juice.shop/)
* Mode: Black-box penetration testing



*Juice Shop – All Products View screenshot*

🔓**SQL Injection Login Bypass**

**📌 Objective:  
  
To bypass standard authentication and retrieve a JWT token by exploiting a SQL injection vulnerability in the login form of OWASP Juice Shop.**

**📌 Exploit Steps:**

**Open Juice Shop login page: https://demo.owasp-juice.shop/#/login**

1. **Enter the following payload in the Email field:**

**admin@juice-sh.op'--**

1. **Use any value for the Password field (e.g., abc123)**
2. **Submit the login form**

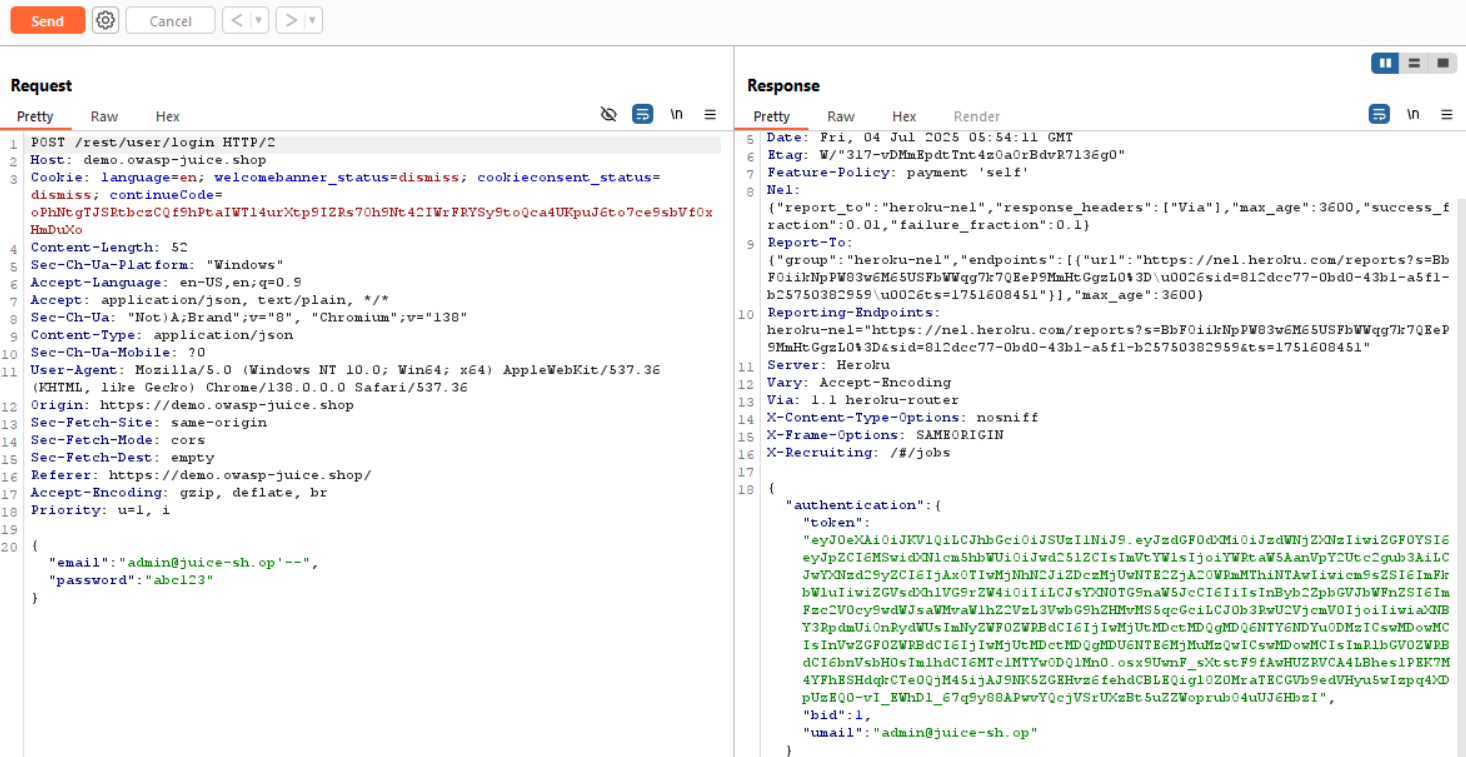
**📌 💡 Explanation:**

* **The '-- payload comments out the SQL condition that normally checks the password**
* **This tricks the application into verifying the login purely by email match**
* **The server returns a valid JWT for admin@juice-sh.op — indicating successful bypass**

**📌 ✅ Outcome:**

* **Logged in successfully without knowing the real password**
* **Captured a JWT token with "admin": false in the payload**

Burp Suite login request showing injection payload screenshot

****

🔍 **Request Panel (Left Side)**

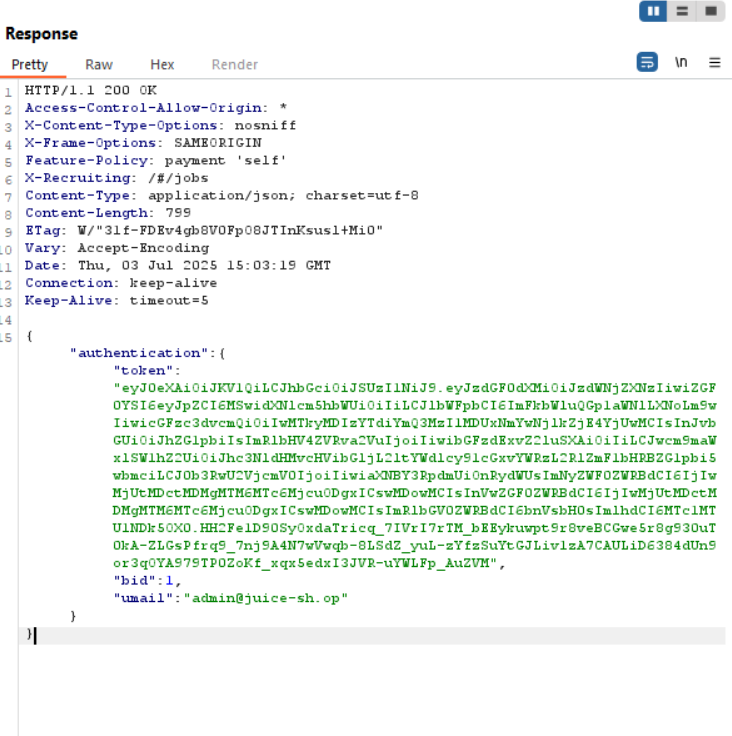
* **Endpoint:**  
  POST /rest/user/login — This is Juice Shop's login API endpoint.
* **Headers:**  
  Includes typical browser headers like User-Agent, Content-Type: application/json, and CORS/security-related keys like X-Content-Type-Options.
* **Payload:**
* **json**

{

"email": "admin@juice-sh.op",

"password": "ab1c23"

}  
✅ This shows you attempted login as admin@juice-sh.op with a basic password — no injection here, but this confirms the app processes such requests and returns tokens on success.



📬response panel  
 `. Status : HTTPS/1.1 200 OK – SUCCESSFUL AUTHENTICATION RESPONSE  
 . AUTHENTICATION TOKEN RETURNED  
JSON  
{

"authentication": {

"token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9..."

}

}  
✅ This shows you attempted login as admin@juice-sh.op with a basic password — no injection here, but this confirms the app processes such requests and returns tokens on success.

**Capturing & Decoding the JWT**

🎯 Objective

To intercept the JWT issued after login (using SQL injection or standard credentials) and decode its structure to understand the claims, algorithm, and integrity mechanism used by the application.

🛠️ Tools Used

* **Burp Suite:** To intercept and examine the response after login
* **jwt.io:** To decode the JWT and test signature verification

📨 Captured JWT Sample

Example JWT (truncated for clarity):

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9. eyJlbWFpbCI6ImFkbWluQGp1aWNlLXNoLm9wIiwiYWRtaW4iOmZhbHNlfQ. J1rVK4D\_xxxxxxx

📌 This token was returned in a JSON response after logging in as admin@juice-sh.op.

📖 Token Breakdown

Using jwt.io, the JWT was split and decoded:

🔹 Header:

{ "alg": "HS256", "typ": "JWT" }

* **Algorithm:** HMAC SHA-256 (symmetric)
* Indicates the token was signed using a shared secret

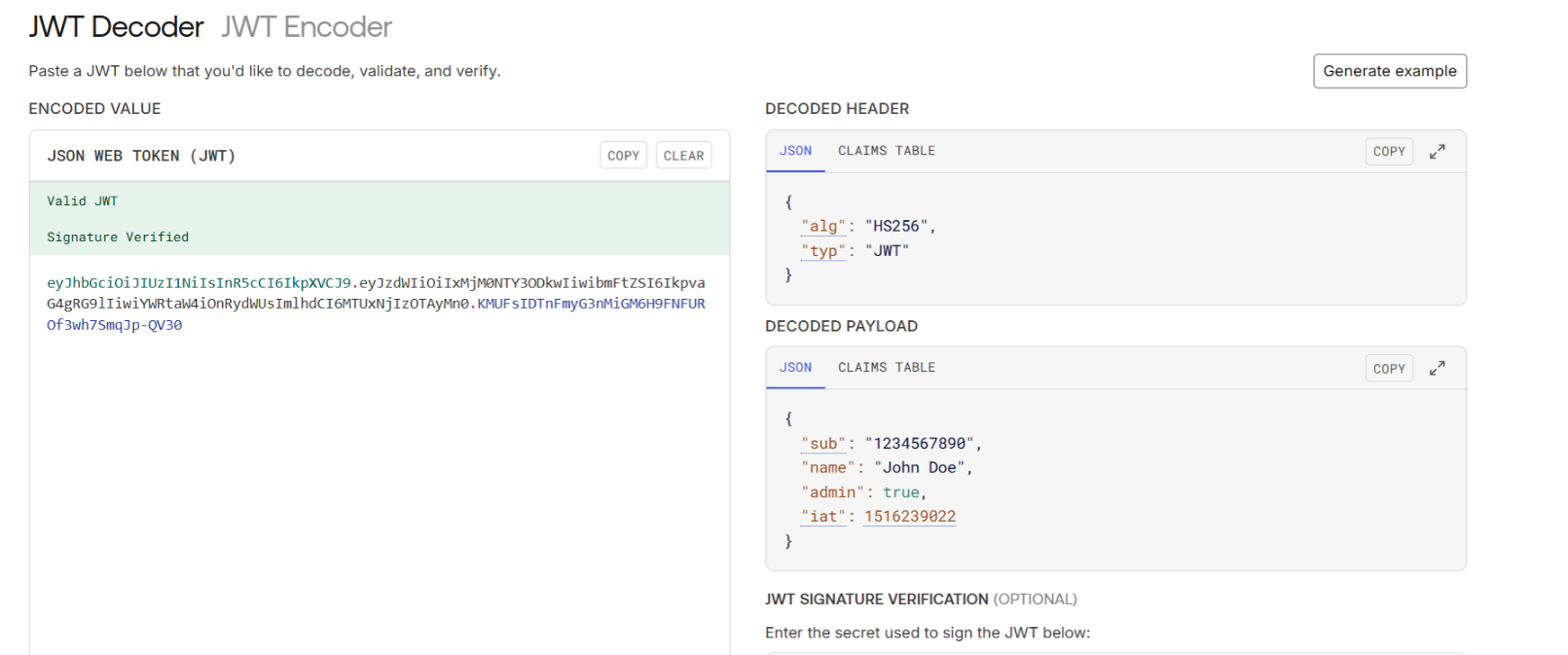
🔹 Payload:

{ "email": "admin@juice-sh.op", "admin": false }

* email shows the impersonated user
* admin: false confirms restricted privileges

🔐 Signature Check

* Entered the string "secret" as the verification key on jwt.io
* Signature ✅ **verified successfully** — confirming weak server-side key handling  
    
  jwt.io interface showing the decoded payload screenshot



“JWT decoded and verified via jwt.io. Payload shows ‘admin: true’, confirming successful token modification and forgery.

✏️  **Crafting the Forged JWT**

🎯 Objective

To modify the intercepted JWT by altering its claims and re-sign it using the known shared secret. This forged token will elevate the user’s privilege from "admin": false to "admin": true.

🧠 Understanding the Target Weakness

* The application uses **HS256** (HMAC SHA256) algorithm, which relies on a shared secret ("secret")

📡  **Replay Forged Token via Burp Suite**

🎯 Objective

* To validate whether the modified JWT (with "admin": true) is accepted by the server by using Burp Suite to send the forged token to a privileged admin endpoint.

🔁 Request Details

* Used the following GET request to test access:

🧾 **Admin Access Confirmation in UI**

🎯 Objective

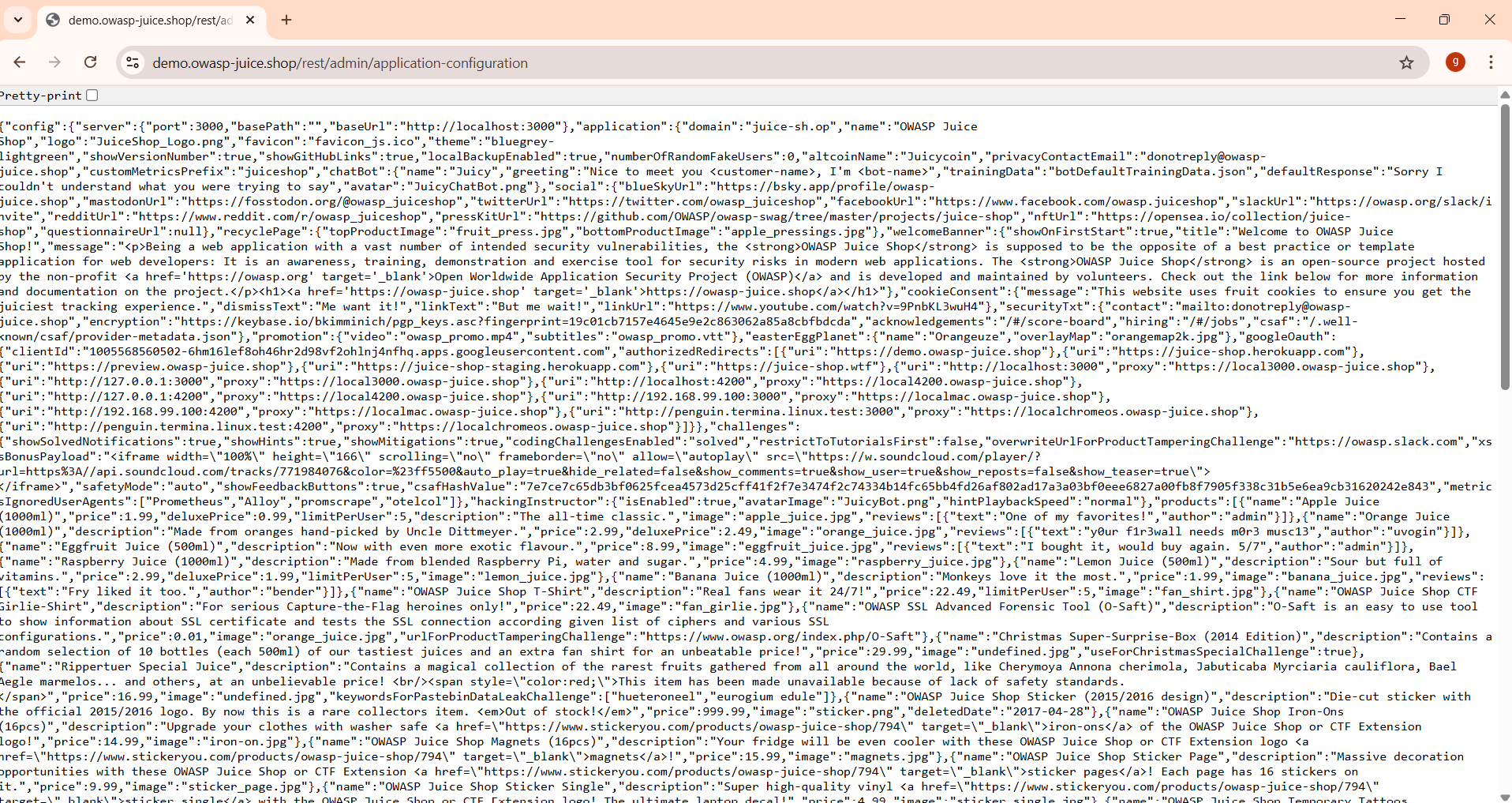
To visually confirm that the forged JWT grants access to admin-only features through the Juice Shop front-end interface or the API’s protected areas.

🔍 Verification Steps

1. Used **Chrome browser** with developer tools or JWT injector extension
2. Injected the **forged JWT** into local storage or Authorization headers

3.Refreshed the Juice Shop interface

4.Navigated to an endpoint or feature exclusive to admins (e.g. /#/administration or /rest/admin/application-configuration)  
🖼 Screenshot Set: Admin Access Confirmation



*“Admin endpoint accessed after injecting forged JWT into browser storage. Application returned complete configuration JSON, confirming unauthorized admin access.”*

📊  **Impact Assessment & Recommendations**

🧨 Exploit Summary

* **Vulnerability Type:** Insecure implementation of JWT authentication
* **Impact Achieved:** Privilege escalation to administrator
* **Method Used:** JWT token captured post-login → payload modified to "admin": true → re-signed with weak secret ("secret")
* **Outcome:** Unauthorized access to /rest/admin/application-configuration and other admin-only functionalities

📌 Risk Classification

|  |  |
| --- | --- |
| Category | Assessment |
| Severity | 🔴 Critical |
| CWE | [CWE-345](https://cwe.mitre.org/data/definitions/345.html) - Insufficient Verification of Data Authenticity |
| OWASP Top 10 | A07:2021 – Identification and Authentication Failures |
| CVSS (Estimate) | **9.1 (Critical)** – High impact to confidentiality, integrity, and access control |

⚠️ Real-World Impact

1.Total bypass of authentication mechanisms

2.Admin features can be misused to change application behavior

3.No need for credentials once the shared key is known

4.Could lead to data exposure, DoS, or malicious reconfiguration

✅ Remediation Recommendations:

1. **Use Asymmetric Signing (RS256) Instead of Symmetric (HS256)**
   * Prevents attackers from forging tokens without the private key
2. **Avoid Common Secrets**
   * Replace "secret" with strong, random keys and rotate them regularly
3. **Implement Role-Based Authorization on Server-Side**
   * Never trust JWT claims alone — validate permissions on each request
4. **Add Expiration + Issuer/Audience Validation**
   * Mitigate replay attacks and scope tokens appropriately
5. **Monitor and Alert on Anomalous JWT Usage**

* Detect spikes in invalid or unsigned tokens

🧰 **Root Cause Analysis**

🧠 1. Weak JWT Signature Handling

* The JWTs used by the application are signed with **HMAC SHA-256 (HS256)**, a **symmetric algorithm**.
* The secret used for signing was trivially guessable:  
  **"secret"**
* Because JWT signature verification depends solely on this shared key, attackers can:
* Decode the token
* Modify payload claims (e.g., "admin": false → true)
* Re-sign the token using the same known secret
* Bypass access controls entirely

🏷️ 2. No Server-Side Role Validation

* The application **fully trusts the token’s admin claim** without any additional server-side verification.
* There’s **no re-validation of roles or permissions** before returning admin-level resources.
* The backend doesn’t check whether the user *should* have these privileges — it only looks at what the token *says*.

💉 3. Login Endpoint Vulnerable to SQL Injection

* The SQL injection on the email field (admin@juice-sh.op'--) allowed authentication bypass.
* This enabled attackers to:
* Log in as admin without knowing the password
* Obtain a baseline token for decoding and experimentation

🔥 4. Combined Exploit Chain

The combination of:

* **SQL injection**
* **Weak JWT secret**
* **Lack of role enforcement**  created a seamless path from external attacker to full admin control.

🛠️ **Mitigation Recommendations**

🔐 1. Switch to Asymmetric JWT Signing (RS256)

* Migrate from **HS256** (symmetric) to **RS256** (asymmetric) signing.
* With RS256:  
  + The private key signs the token
  + The public key is used to verify it
* Prevents attackers from forging tokens even if the public key is exposed.  
    
  🧠 *Impact: Eliminates signature forgery via shared-key guessing*

🧪 2. Enforce Server-Side Role Validation

* Never trust claims like "admin": true directly from the token.
* Always **validate user roles** against backend data (e.g., database or user management system).
* Maintain centralized access controls, especially for admin routes.
* 🧠 *Impact: Stops unauthorized privilege escalation even with a valid token*

🧹 3. Sanitize Login Inputs to Prevent SQL Injection

* Use **parameterized queries** or ORM-safe methods.
* Validate and sanitize all input fields server-side, especially in the login logic.
* Implement WAF (Web Application Firewall) to monitor malicious payloads.
* 🧠 *Impact: Removes the initial entry point used to obtain the JWT*

🔄 4. Implement Token Expiry and Rotation

* Set **short-lived access tokens** with refresh-token flow.
* Regularly rotate signing keys and monitor JWT signature usage patterns.
* 🧠 *Impact: Reduces the attack window and improves resilience against token leakage*

📉 5. Limit Token Scope and Claims

* Avoid sensitive claims like "admin" directly in token payloads.
* If necessary, **encrypt JWT payloads** or store sensitive roles server-side only.
* 🧠 *Impact: Prevents client-side manipulation of critical access data*

📡 6. Monitor for JWT Tampering

* Log and alert when JWT verification fails (e.g., signature mismatch)
* Use anomaly detection to track abnormal access attempts with custom roles
* 🧠 *Impact: Enables early detection of attempted token forgery*

🧾  **Conclusion & Takeaways**

🔚 **Conclusion**

This assessment successfully demonstrated a complete JWT forgery exploit chain in the OWASP Juice Shop application — starting from an SQL injection login bypass, escalating through JWT token manipulation, and culminating in unauthorized admin-level access.

By modifying the token’s payload and re-signing it with a known shared secret ("secret"), the application was tricked into treating a non-admin user as an administrator. The exploit relied entirely on client-side tampering, with no server-side revalidation of roles — revealing a critical breakdown in authentication and access control mechanisms.

💡 **Key Takeaways**

* **JWTs must never be trusted blindly** — especially claims like "admin": true. Always validate server-side.
* **Weak secrets = weak security.** Using a simple, guessable signing key makes signature forgery trivial.
* **Role-based access should be enforced beyond tokens.** Real privilege enforcement requires cross-checking identities against a secure backend.
* **Security flaws rarely exist alone.** This exploit succeeded due to the combination of:
* SQL Injection vulnerability
* Insecure JWT implementation
* Missing server-side authorization checks

🛠️ Final Word

* *“Security isn’t just about authentication — it’s about validation, verification, and defense in depth.”*
* This case study serves as a powerful reminder that even small oversights in token handling or input validation can open doors to full-system compromise. Developers, testers, and defenders must continuously assess how applications validate trust — both at the surface and beneath