

Portfolio Report #1 — Security Assessment of OWASP Juice Shop (local instance)

Author: [your name] **Date:** 9 Oct 2026 **Target:** OWASP Juice Shop v20.2.0, run on `localhost:3000` on my own machine **Target type:** OWASP's own intentionally-vulnerable training application — built and published specifically for people to practice finding these exact classes of bugs (<https://github.com/juice-shop/juice-shop>) **Authorization:** No authorization needed or sought — this is OWASP's publicly released training app, run entirely on my own local machine, never pointed at anyone else's system. This is **not** a real client's software and this report should never be read as "a company got hacked."

Full findings + evidence: [findings.md](#)

Why this report exists

I'm building a security-testing portfolio. Rather than claim certifications I don't have, I set up a real, legally-safe target (an app OWASP themselves publish for this purpose), found real bugs in it the way a junior AppSec/pentest engineer would, and wrote them up the way I'd write a real client report — severity, evidence, business impact, fix. This is meant to show *how I test and document*, not to claim OWASP's own demo app as a "client."

Summary of findings

#	Finding	OWASP category	Severity
1	SQL injection in the login form → full authentication bypass as admin	A03:2021 Injection	Critical
2	Broken access control (IDOR) on the shopping basket API — any logged-in user can read any other user's basket	A01:2021 Broken Access Control	High
3	SQL injection in the product search endpoint + verbose error disclosure (DB engine, framework version)	A03:2021 Injection / A05:2021 Security Misconfiguration	Medium

How I ran this (reproducible, local only)

1. Downloaded the official Windows release zip from the project's GitHub releases page.
2. Extracted it to a short path (`C:\js\`) — Windows has a 260-character path limit and this app's dependency tree has some very deeply nested file paths that silently fail to extract under a long base path. That cost me the most time, and is itself a useful lesson in reproducible test setups.
3. Ran it with `node build/app.js` — no Docker needed.
4. Tested against `http://localhost:3000` only, using `curl` , never touched any machine but my own.
5. **I did not run this against `demo.owasp-juice.shop`** (the public hosted demo) — the project's own docs say that instance is for previewing only, not for hacking practice, so I set up my own local copy instead.

Methodology

- Black-box testing against the REST API as an unauthenticated and then low-privilege authenticated user.
 - Started from the OWASP Top 10 (2021) as a checklist, prioritizing Injection and Broken Access Control since they're the highest-impact, most common categories in real-world findings.
 - Every finding below has: the exact request, the exact (unmodified) response, why it matters in business terms, and a concrete fix — not just "found a vuln."
-

Findings — OWASP Juice Shop v20.2.0 (local instance, 9 Oct 2026)

Finding 1 — SQL Injection → Authentication Bypass (Critical)

Endpoint: `POST /rest/user/login` **OWASP category:** A03:2021 — Injection **CVSS 3.1 (my estimate):** 9.8 (Critical) — `AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H`

What's happening

The login endpoint builds its SQL query by directly concatenating the submitted email into the query string instead of using parameterized queries / an ORM query builder safely. A classic SQL comment

injection (' --) truncates the rest of the `WHERE` clause, including the password check, so the query returns the first matching user regardless of password.

Proof of concept

Request:

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

{"email":"admin@juice-sh.op' --","password":"anything-at-all"}
```

Response (200 OK, trimmed):

```
{
  "authentication": {
    "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9...<JWT omitted, decodes to role:\"admin\">",
    "bid": 1,
    "umail": "admin@juice-sh.op"
  }
}
```

Decoding the JWT payload confirms `"role":"admin"` and the real admin account's record — reached with an arbitrary password, no credential guessing or brute force involved.

Business impact

Full account takeover of **any known email address**, including the administrator, with zero knowledge of the real password. On a real e-commerce platform this is a direct path to: reading/editing all customer PII and order history, issuing refunds, changing prices, and (if the admin panel allows it, which it does here) modifying site content. This is the single worst class of bug a web app can have — it defeats authentication entirely.

Root cause

Building SQL by string concatenation with user input instead of using parameterized queries/prepared statements.

Remediation

1. Use parameterized queries or the ORM's safe query builder for **every** query that includes user input — no string concatenation into SQL, ever.
2. Add an automated test that specifically tries `' -- , ' OR '1'='1`, and similar payloads against every auth endpoint, and fail the build if any returns 200.
3. Defense in depth: rate-limit login attempts and alert on repeated SQL-error responses from the same IP — a string of 500s with `SQLITE_ERROR` / `ORA-` /etc. in the body is a strong signal of someone probing for injection.

Finding 2 — Broken Access Control / IDOR on Shopping Basket API (High)

Endpoint: `GET /rest/basket/:id` **OWASP category:** A01:2021 — Broken Access Control (Insecure Direct Object Reference) **CVSS 3.1 (my estimate):** 6.5 (Medium-High) —
`AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N`

What's happening

The basket endpoint takes a basket ID straight from the URL and returns that basket's contents to **whoever is authenticated**, without checking that the basket actually belongs to the requesting user. Basket IDs are small sequential integers, so they're trivially guessable (1, 2, 3...).

Proof of concept

1. Registered a brand-new, completely unrelated customer account (`demo_user_...@example.com` , assigned user id 25, empty basket).
2. Logged in as that new user and got a valid JWT for **that** account only.
3. Using that low-privilege token, requested basket id `1` (which belongs to user id `1` , the admin account, and already had real items in it):

```
GET /rest/basket/1 HTTP/1.1
Host: localhost:3000
Authorization: Bearer <JWT for user id 25>
```

Response (200 OK):

```
{
  "status": "success",
  "data": {
```

```
"id": 1,
"UserId": 1,
"Products": [
  {"name": "Apple Juice (1000ml)", "BasketItem": {"quantity": 2, ...}},
  {"name": "Orange Juice (1000ml)", "BasketItem": {"quantity": 3, ...}},
  {"name": "Eggfruit Juice (500ml)", "BasketItem": {"quantity": 1, ...}}
]
}
```

A user with zero relationship to account 1 can read exactly what's in their cart. The same missing check almost certainly extends to modifying/emptying other users' baskets (`PUT` / `DELETE` on the same route), which I did not additionally exercise since read access alone is enough to prove the control is missing — on a real client engagement I'd confirm write access only with explicit go-ahead, since that starts to affect other users' actual data.

Business impact

On a real store, this leaks what every customer has in their cart (purchase intent, which is competitively/privacy sensitive), and if write operations have the same missing check, an attacker could tamper with other customers' carts/orders. It also means basket IDs can't be trusted as an access boundary anywhere else in the codebase that reuses this pattern.

Root cause

Missing object-level authorization check: the handler fetches `Basket.findByPk(req.params.id)` without verifying `basket.UserId === req.user.id`.

Remediation

1. On every object-fetch-by-ID endpoint, add an explicit ownership/role check before returning data — not just "is this a valid, authenticated token."
 2. Prefer deriving the basket ID from the authenticated session (`req.user.id` → look up *their* basket) instead of trusting a client-supplied ID at all, where the data model allows it.
 3. Add authorization tests per role: "user A's token must never be able to read/write user B's resource," run against every `:id`-style route, not just the ones someone remembers to test by hand.
-

Finding 3 — SQL Injection + Verbose Error Disclosure on Search (Medium)

Endpoint: GET /rest/products/search?q= **OWASP category:** A03:2021 — Injection, plus A05:2021 — Security Misconfiguration (verbose errors) **CVSS 3.1 (my estimate):** 5.3 (Medium) — AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:L

What's happening

The product search endpoint also builds its query unsafely: a single quote in the search term breaks the SQL statement and the raw database error is returned to the client in a full HTML error page, along with the backend framework name and version in the page title.

Proof of concept

Request:

```
GET /rest/products/search?q=<img src=x onerror=alert('xss')>
```

Response (500, trimmed):

```
<title>Error: SQLITE_ERROR: near "xss": syntax error</title>
<h1>OWASP Juice Shop (Express ^4.22.1)</h1>
```

A normal search (?q=juice) behaves correctly and returns 35 matching products, confirming this isn't just a broken feature — it's unsanitized input reaching the database layer.

Business impact

On its own this is a lower-severity finding than #1 (no auth bypass was demonstrated here specifically), but it confirms a second, independent injection point in the codebase — meaning the unsafe-query pattern is systemic, not a one-off mistake — and the verbose error (DB engine + exact web framework version) hands an attacker a head start on picking exploits for known CVEs in that exact version.

Root cause

Same class of bug as Finding 1 (unsafe query construction), plus a production error handler that returns stack-trace-style detail to the client instead of a generic message.

Remediation

1. Same fix as Finding 1 — parameterize this query too. Given two independent instances of the same bug class were found with only a few minutes of testing, I'd recommend a full code-level grep for raw SQL string concatenation across the codebase rather than patching endpoints one at a time.
 2. Turn off verbose/stack-trace error pages in production; log full details server-side only, return a generic message + request ID to the client.
 3. Remove or suppress framework name/version from default error pages and the `Server` header.
-

Overall recommendation

All three findings trace back to **unsafe query construction** (Findings 1 & 3) and **missing per-object authorization** (Finding 2) — two of the most common root causes behind real breaches. I'd prioritize Finding 1 for same-day fix (it's a full auth bypass), then a codebase-wide audit for the same SQL pattern before closing Finding 3, then add the missing ownership checks for Finding 2 and write a regression test per finding so these can't silently come back.