

Portfolio Report #2 — From Finding to Fix:

Remediating OWASP Juice Shop (local instance)

Author: [your name] **Date:** 9 Oct 2026 **Target:** OWASP Juice Shop v20.2.0 source (official GitHub repo), run on `localhost:3001` on my own machine **Builds on:** [Report #1](#), where I found and documented 3 real vulnerabilities in this same app **Authorization:** Same as Report #1 — OWASP's own published training app, cloned and run entirely locally, never pointed at anyone else's system or submitted upstream (Juice Shop keeps these bugs *on purpose*, for training — this is a personal exercise, not a contribution to the project).

Why this report exists

Most junior pentest/AppSec portfolios stop at "I found a bug." Hiring managers for **AppSec / DevSecOps** roles — the role I'm targeting (see [../../cyber_roadmap.md](#)) — care just as much about whether someone can **read the vulnerable code, write a correct fix, and prove the fix doesn't break the feature it protects**. That's the actual day-to-day job: triage a finding, patch it, verify in CI, ship it.

So this report takes all 3 findings from Report #1 one step further: 1. Clone the real Juice Shop **source code** (not just the prebuilt release used in Report #1). 2. Patch each vulnerable route at the code level. 3. Re-run the exact same exploit requests from Report #1 against the patched server — proof they now fail. 4. Re-run the exact same **legitimate** requests — proof nothing broke for real users. 5. Show the actual diff, so a reviewer can judge the fix quality directly.

Summary

#	Finding (from Report #1)	Fix applied	Exploit after fix	Legit use after fix
1	SQL injection in login → admin auth bypass	Parameterized query (Sequelize named <code>replacements</code>) instead of string concatenation	<code>401 Invalid email or password</code> (was: <code>200 + admin JWT</code>)	New user registers + logs in normally: <code>200</code> + own JWT

#	Finding (from Report #1)	Fix applied	Exploit after fix	Legit use after fix
2	IDOR on <code>GET /rest/basket/:id</code>	Added explicit ownership check: requester's own <code>bid</code> (from their session token) must match the basket's <code>id</code>	<code>403 You are not allowed to access this basket (was: 200 + another user's basket contents)</code>	Same user reading their own basket: <code>200</code> + their own (empty) basket
3	SQL injection + verbose error in product search	Parameterized <code>LIKE</code> pattern instead of string concatenation	<code>200</code> + empty result set, no stack trace, no framework/DB version leaked (was: <code>500</code> + <code>SQLITE_ERROR</code> + <code>Express ^4.22.1</code> in the page title)	Normal search (<code>q=juice</code>): still returns all 35 matching products

Full before/after evidence (exact requests + responses) and the code diff: [remediation.md](#)

How I did this (reproducible, local only)

1. `git clone --depth 1 --branch v20.2.0 https://github.com/juice-shop/juice-shop.git` to a short path (`C:\js2`) — same `MAX_PATH` lesson from Report #1 applies to a fresh clone's `node_modules` too.
2. Read the actual vulnerable source in `routes/login.ts` , `routes/basket.ts` , `routes/search.ts` and patched each one directly (see diff in `remediation.md`).
3. `npm install` , then ran the **server only** via `tsx app.ts` (no full Angular frontend build needed, since I was only testing the REST API) on port 3001, alongside the frontend's required-file precondition check satisfied with placeholder files (this app refuses to boot unless specific frontend build artifacts exist on disk — I stubbed them out since I never touch the UI in this exercise).
4. Re-ran every exploit and every legitimate-use request from Report #1 against the patched server and captured the real responses below.
5. Stopped the server and left the patched clone in place at `C:\js2` for review; nothing was pushed anywhere, and no change was proposed to the real juice-shop project (these bugs are intentional training content there, not something upstream wants "fixed").

Methodology

- Fix **only** the vulnerable line(s) each finding points to — no unrelated refactors, so the diff is small enough for someone else to review in minutes.

- Every fix is paired with **two** test requests: the original attack (must now fail safely) and the original legitimate use (must still succeed) — a fix that breaks the feature it protects is not a real fix.
- Preferred the framework's own safe primitive (Sequelize's parameterized `replacements`) over hand-rolled escaping, since hand-rolled string sanitization is itself a common source of bypassable "fixes."

Remediation — code diff + before/after proof (9 Oct 2026)

Patched server: OWASP Juice Shop v20.2.0 source clone, run locally on `http://localhost:3001` .

Fix 1 — SQL Injection → Authentication Bypass (`routes/login.ts`)

The vulnerable code (before)

```
models.sequelize.query(  
  `SELECT * FROM Users WHERE email = '${req.body.email} || ''' AND password = '${security.hash(req.body.pass  
  { model: UserModel, plain: true }  
)
```

Raw string concatenation — `req.body.email` goes straight into the SQL text.

The fix (after)

```
models.sequelize.query(  
  'SELECT * FROM Users WHERE email = :email AND password = :password AND deletedAt IS NULL',  
  {  
    replacements: { email: req.body.email || '', password: security.hash(req.body.password || '') },  
    model: UserModel,  
    plain: true  
  }  
)
```

Sequelize's named `replacements` binds the values as real SQL parameters — the email/password are now always treated as *data*, never as part of the query text, so a `'` in the input can't change the query's structure no matter what's in it.

Proof — exploit now fails

Request (same payload as Report #1, Finding 1):

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

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

Response:

```
HTTP/1.1 401 Unauthorized
Invalid email or password.
```

(Report #1: this same request returned `200` with a valid admin JWT.)

Proof — legitimate login still works

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

{"email":"demo2_patch@example.com","password":"Passw0rd!123"}
```

Response: `200 OK`

```
{"authentication":{"token":"eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9...","bid":6,"umail":"demo2_patch@example.co
```

A freshly registered, real customer account logs in exactly as before — the fix changes nothing about normal behavior.

Fix 2 — Broken Access Control / IDOR (`routes/basket.ts`)

The vulnerable code (before)

```
const basket = await BasketModel.findOne({ where: { id }, include: [...] })
// ...no ownership check at all before returning `basket`
res.json(utils.queryResultToJson(basket))
```

The fix (after)

```
const requester = security.authenticatedUsers.from(req)
if (basket && requester?.bid !== basket.id) {
  res.status(403).json({ error: 'You are not allowed to access this basket.' })
  return
}
```

Added directly after the existing (unrelated) challenge-tracking check. Compares the basket id actually being requested against the **requester's own** basket id, taken from their own authenticated session (`security.authenticatedUsers.from(req)`) — not from anything the client can control in the URL or body.

Proof — exploit now fails

A low-privilege user (own basket id `6`) requests another user's basket (`1` , which belongs to a different account):

```
GET /rest/basket/1 HTTP/1.1
Authorization: Bearer <JWT for the low-priv user, bid=6>
```

Response:

```
HTTP/1.1 403 Forbidden
{"error":"You are not allowed to access this basket."}
```

(Report #1: this same request returned `200` with the full contents of basket 1, which wasn't theirs.)

Proof — legitimate use still works

The same user requests their **own** basket (`6`):

```
GET /rest/basket/6 HTTP/1.1
Authorization: Bearer <JWT for the low-priv user, bid=6>
```

Response: `200 OK`

```
{"status": "success", "data": {"id": 6, "coupon": null, "UserId": 25, "Products": []}}
```

Fix 3 — SQL Injection + Verbose Error Disclosure (routes/search.ts)

The vulnerable code (before)

```
models.sequelize.query(  
  `SELECT * FROM Products WHERE ((name LIKE '%${criteria}%' OR description LIKE '%${criteria}%') AND deleted  
  )
```

The fix (after)

```
models.sequelize.query(  
  'SELECT * FROM Products WHERE ((name LIKE :pattern OR description LIKE :pattern) AND deletedAt IS NULL) OR  
  { replacements: { pattern: `>${criteria}%` } }  
  )
```

The `%...%` wildcard wrapping now happens in JavaScript, on a value that is bound as a single SQL parameter — a stray `'` in the search term is just a literal character inside that parameter, not SQL syntax anymore.

Proof — exploit now fails

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

Response:

```
HTTP/1.1 200 OK  
{ "status": "success", "data": [] }
```

(Report #1: this same request returned `500` with the raw `SQLITE_ERROR: near "xss": syntax error` message and `OWASP Juice Shop (Express ^4.22.1)` exposed in the error page title.)

Proof — legitimate use still works

```
GET /rest/products/search?q=juice
```

Returns `200 OK` with all 35 matching products, same as before the fix.

Full code diff (git diff, unmodified)

```
diff --git a/routes/basket.ts b/routes/basket.ts
--- a/routes/basket.ts
+++ b/routes/basket.ts
@@ -22,6 +22,15 @@ export function retrieveBasket () {
    const user = security.authenticatedUsers.from(req)
    return user && id && id !== 'undefined' && id !== 'null' && id !== 'NaN' && user.bid && user?.bid !=
  })
+  // PATCHED (portfolio demo): object-level authorization check (see
+  // findings.md #2 – any authenticated user could read any basket by
+  // guessing its id). Only the basket's own owner, matched against the
+  // caller's own session token, may read it.
+  const requester = security.authenticatedUsers.from(req)
+  if (basket && requester?.bid !== basket.id) {
+    res.status(403).json({ error: 'You are not allowed to access this basket.' })
+    return
+  }
+  if (((basket?.Products) != null) && basket.Products.length > 0) {
+    for (let i = 0; i < basket.Products.length; i++) {
+      basket.Products[i].name = req.__(basket.Products[i].name)

diff --git a/routes/login.ts b/routes/login.ts
--- a/routes/login.ts
+++ b/routes/login.ts
@@ -31,7 +31,14 @@ export function login () {

  return (req: Request, res: Response, next: NextFunction) => {
    verifyPreLoginChallenges(req) // vuln-code-snippet hide-line
-    models.sequelize.query(`SELECT * FROM Users WHERE email = '${req.body.email} || ''' AND password = '${s
+  // PATCHED (portfolio demo): parameterized query instead of string concatenation.
+  // The original builds raw SQL from req.body.email/password, which lets
+  // `admin@juice-sh.op' --` truncate the password check (see findings.md #1).
+  models.sequelize.query('SELECT * FROM Users WHERE email = :email AND password = :password AND deletedAt
+    replacements: { email: req.body.email || '', password: security.hash(req.body.password || '') },
+    model: UserModel,
+    plain: true
+  })
+  .then((authenticatedUser) => { // vuln-code-snippet neutral-line loginAdminChallenge loginBenderChall
```

```

const user = utils.queryResultToJson(authenticatedUser)
if (user.data?.id && user.data.totpSecret !== '') {

diff --git a/routes/search.ts b/routes/search.ts
--- a/routes/search.ts
+++ b/routes/search.ts
@@ -20,7 +20,12 @@ export function searchProducts () {
  return (req: Request, res: Response, next: NextFunction) => {
    let criteria: any = req.query.q === 'undefined' ? '' : req.query.q ?? ''
    criteria = (criteria.length <= 200) ? criteria : criteria.substring(0, 200)
-   models.sequelize.query(`SELECT * FROM Products WHERE ((name LIKE '%${criteria}%' OR description LIKE '%
+   // PATCHED (portfolio demo): parameterized LIKE pattern instead of string
+   // concatenation (see findings.md #3 – unescaped input broke the query and
+   // leaked DB engine + framework version in the error page).
+   models.sequelize.query('SELECT * FROM Products WHERE ((name LIKE :pattern OR description LIKE :pattern)
+   replacements: { pattern: `'%${criteria}%'` }
+   })
    .then(([products]: any) => {
      const dataString = JSON.stringify(products)
      if (challengeUtils.notSolved(challenges.unionSqlInjectionChallenge)) { // vuln-code-snippet hide-st

```

Notes on scope / what this does and doesn't prove

- This fixes the exact 3 vulnerable queries found in Report #1. It is **not** a claim that the rest of the codebase is clean — Juice Shop has dozens of other intentional vulnerabilities by design, left untouched here on purpose (patching the whole app would defeat its purpose as a training target for other learners).
- The IDOR fix only covers the `GET` (read) path, matching the scope of the original finding. A full fix would need the same ownership check added to any `PUT / DELETE` routes on `/rest/basket/:id` too — flagged here rather than silently assumed.
- These patches live only in a local clone (`C:\js2`) for this exercise and were never opened as a pull request against the real juice-shop project, since the project intentionally ships these bugs for other people to practice on.