

Secure Coding Practices: Lab

Student worksheet · OWASP Juice Shop

For the bugs you exploited today, now find the vulnerable line and pick the fix.

Open the **Score Board**: `http://localhost:3000/#/score-board`. A solved challenge with a <> icon has a **coding challenge**. Click it: first **Find it** (click the line you think is the bug), then **Fix it** (pick the version that repairs it). Juice Shop marks you at once and explains every wrong option, so this is a conversation with the real code.

Note: a coding challenge only opens once its hacking challenge is solved. If a <> is locked, go solve that challenge first.

1. SQL Injection — Login Admin

1. **Find it:** the login query builds SQL by pasting the email *into* the query text, a template string like `...email = '${req.body.email}'...`. Click that line.
2. **Fix it:** pick the **parameterized query** — fixed query text with a placeholder, the email passed separately as a value. Not the one that keeps building the string by hand.

Why: with a parameter, the input stops being code and becomes just a value. The apostrophe you used this morning can no longer break out of the string and change the query.

2. Cross-Site Scripting — DOM XSS

1. **Find it:** the line `this.searchValue = this.sanitizer.bypassSecurityTrustHtml(...)`. It switches *off* the framework's escaping for the search text. Click it.
2. **Fix it:** pick the option that **removes** `bypassSecurityTrustHtml` and assigns the value plainly, so Angular escapes it and the search text is shown as text.

Why: the bug is turning protection *off*; the fix is leaving it *on*. The name `bypassSecurityTrust` is the warning. With escaping on, an `<iframe>` is just characters on the page, not a real element.

3. Weak Authentication — Password Strength

1. **Find it:** passwords are stored with `md5()`, a **fast** hash. Click that line.
2. **Fix it:** pick the option that uses a **slow** password hash such as `bcrypt`. (In real code you also add strength rules and a login lockout.)

Why: MD5 is fast, so trying the whole common-password list is fast. A slow hash and a lockout make that guessing impractical.

4. Misconfiguration — Confidential Document

1. **Find it:** the line `app.use('/ftp', serveIndex('ftp', { icons: true }))`. It lists every file in the folder. Click it.
2. **Fix it:** pick the option that **removes the listing** and does not expose `/ftp`. (Hiding just the listing is not enough — files still download by name.)

Why: no logic bug here: a setting opened the door. Every feature left on is attack surface.

5. Do more on your own

Open any other solved challenge with a <> icon and do find-and-fix: **Login Jim**, **Login Bender**, **Database Schema**, **Bonus Payload**, **Reset Jim's Password**. For each, write one line: *what was the root cause, in your own words?*

If stuck: **No <> icon:** that challenge has no coding challenge (e.g. Reflected XSS, Error Handling, Forged Feedback, Unsigned JWT); skip it. **"Close" but not accepted:** read the explanation — several options compile, but only one truly fixes it.

What you did: every fix is the same idea

Vulnerability	The mistake	The fix
SQL injection	input glued into a query	parameterized query
XSS	input written as HTML	escape output; no bypass
weak auth	fast hash, weak default	slow hash, strong rules, lockout
misconfiguration	a feature left on	turn it off

Look down the fix column: it is two habits, over and over. **Keep data as data** — parameters for SQL, escaping for HTML, so input never becomes code. **Decide trust on the server** — identity, price, author, role, never taken from the client. That is secure coding in one breath.