

Lab Guide

Four exercises across four languages: Python web app, Node.js API, C authentication, and YAML config — hunt vulnerabilities with AI as your co-auditor.

Authorized Use Only

All source code is **synthetic** and intentionally vulnerable. Do not reuse this code in any real project. The vulnerability classes you learn to spot here apply to real audits only when you have explicit written authorization.

Prerequisites

- Completed the [Theory](#) reading
- Access to an AI assistant (Claude, ChatGPT, or local LLM)
- Downloaded all four files into a folder named `code-audit-lab/`
- Familiarity with: OWASP Top 10, CWE categories, basic Python / JavaScript / C syntax
- Optional: `semgrep`, `bandit`, `eslint-plugin-security` for cross-checking

Lab Scenario

AcmeCorp has asked you to audit four code artifacts from their repo before a production push. Each file is short enough to read end-to-end, but contains at least one exploitable bug. Your job: find every vulnerability, rank them by exploit value, and produce a fix recommendation the dev team can act on within one sprint.

Download Lab Files

Save all four files into a folder named `code-audit-lab/` before starting the exercises.

↓ [vulnerable-webapp.py.txt](#) ~5 KB

↓ [vulnerable-api.js.txt](#) ~5 KB

↓ [legacy-auth.c.txt](#) ~4 KB

↓ [config-leak.yaml.txt](#) ~4 KB

Exercise 1 — Python Web App Audit

Open [vulnerable-webapp.py.txt](#). This is a Flask/Django-style web application.

Task 1.1 — Full vulnerability pass

You are a senior application security auditor. Review the Python web application below and produce:

1. A numbered list of every vulnerability, each with:
 - CWE ID (e.g., CWE-89 SQL Injection)
 - Exact line numbers
 - Severity (Critical / High / Medium / Low)
 - Attacker prerequisites (authenticated? network position?)
 - Proof-of-concept payload or curl command
2. Specifically look for: SQL injection, command injection, path traversal, SSRF, deserialization, XSS, auth bypass, IDOR, hard-coded secrets, insecure crypto, and race conditions.
3. The single most exploitable finding and a one-paragraph "how an attacker would chain this to RCE".
4. A unified diff (git-style) fix for the top 3 findings.

[paste vulnerable-webapp.py.txt]

✓ **Self-check: A thorough audit of this file should yield at least 5 distinct vulnerabilities. If the AI finds fewer, ask "what else did you miss?" — it usually finds 1-2 more on a second pass.**

Exercise 2 — Node.js API Audit

Open `vulnerable-api.js.txt`. This is an Express-style REST API.

Task 2.1 — Authentication and authorization review

Below is a Node.js/Express API. Focus your audit on:

1. Authentication flaws: weak JWT handling, missing signature verification, algorithm confusion (none/HS256/RS256), token replay, missing expiry checks.
2. Authorization flaws: missing ownership checks (IDOR), broken object-level access control, role checks done on client side, mass assignment.
3. Input validation: NoSQL injection, prototype pollution, regex DoS.
4. Async pitfalls: race conditions in check-then-act patterns, missing await on security-critical operations.
5. Dependency usage: any deprecated or known-vulnerable packages referenced by import.

For each finding output:

- File:line
- Vulnerability class
- Exploit scenario
- Minimal fix (show the corrected lines)

[paste vulnerable-api.js.txt]

✓ Self-check: Node APIs almost always have at least one IDOR and one JWT issue. You should be able to name both by the end of this exercise.

Exercise 3 — Legacy C Auth Audit

Open `legacy-auth.c.txt`. This is a small C authentication module — the kind of code that lives forever in enterprise systems.

Task 3.1 — Memory safety and logic flaws

Below is a C authentication module. Audit for:

1. Memory safety bugs: buffer overflow (stack/heap), off-by-one, format string, use-after-free, double-free, integer overflow/underflow.
2. String handling pitfalls: unbounded strcpy/strcat/sprintf/gets, missing null terminators, assumptions about fixed sizes.
3. Authentication logic flaws: timing attacks on memcmp/strcmp, weak random number generation, predictable tokens, password stored in plaintext or with reversible encoding.
4. Integer handling: signed/unsigned confusion, truncation in size arguments.
5. For each finding, give:
 - Line number
 - Bug class (CWE)
 - Whether it is exploitable for code execution, info leak, or auth bypass
 - A patch (show before/after)

Rank findings by exploitability. Identify if any chain to RCE.

[paste legacy-auth.c.txt]

✓ Self-check: C code of this size usually has 2-4 memory safety issues. You should have at least one that the AI classifies as "exploitable for code execution."

Exercise 4 — Config Leak Audit

Open `config-leak.yaml.txt`. Configuration files are where production secrets and bad defaults hide.

Task 4.1 — Extract secrets and misconfigurations

Below is a YAML configuration file. Extract:

1. Every secret, credential, token, API key, or connection string (even if it looks like a "placeholder" — flag it anyway).
2. Every overly-permissive setting: debug modes on, wildcard CORS,

```
allow-all CSP, TLS verification disabled, weak ciphers, default
admin accounts.
```

3. Every service endpoint exposed to the internet that should not be.
4. Any reference to internal infrastructure that leaks topology (hostnames, private IPs, cluster names).
5. Any dangerous feature flag combinations (e.g., "dev_mode: true" in production, "allow_unauthenticated: true" on an admin API).

Output:

- Table of findings (key, value, risk, fix)
- The 3 changes with the largest blast-radius reduction

```
[paste config-leak.yaml.txt]
```

✓ **Self-check: Config leaks usually contain at least one real-looking credential and one "debug: true" type setting. You should walk away with a credential you could try against a live endpoint and a misconfiguration that would change the attack surface.**

Final Exercise — Build the Audit Report

Task 5.1 — Cross-artifact correlation

In a fresh AI conversation, paste the outputs of Exercises 1–4 and ask:

```
Here are four audit outputs for four files from the same
repository. Produce:
```

1. A consolidated findings table (all files, all vulns, deduped).
2. Any vulnerability that spans multiple files (e.g., a secret leaked in config that is used as an HMAC key in the Python app).
3. The top 5 findings ranked by CVSS 3.1 base score (estimate).
4. A 1-week remediation plan: what the dev team should fix in sprint 1, sprint 2, and backlog.
5. Three detection rules (Semgrep or CodeQL syntax) that would have caught the highest-severity issue in CI before merge.

```
Output as markdown, no more than 1200 words.
```

✓ **Self-check:** Your final report should identify at least one cross-file chain (e.g., "secret from config.yaml is used as a weak HMAC key in webapp.py, enabling signature forgery"). That cross-reference is the highest-value output of a code audit.

Reflection

- Which language gave the AI the most trouble? Where did it miss bugs a human would catch?
- Did the AI hallucinate vulnerabilities that were not real? How did you tell?
- How would you verify an AI-reported finding in a real engagement before reporting it to a client?
- What is the ceiling of AI-assisted audit — at what point would you still need a human with domain context?

What's Next

Lesson 05 narrows the focus to the web layer specifically — using AI to triage web-application scanner output and reconstruct exploit chains from Burp/ZAP artifacts.