

Lab Guide

Four exercises: vulnerable-server code review, crash-dump root-cause analysis, CVE-to-PoC conversion, and exploit-template completion — weaponize vulnerabilities with AI assistance.

Authorized Use Only & Defensive Framing

All vulnerable code and exploit targets are **synthetic**. This lab teaches you to read vulnerability reports and write proof-of-concept code for *authorized testing and defensive validation only*. Do not run any exploit against a system you do not own or have explicit written permission to test.

Prerequisites

- Completed the [Theory](#) reading
- Access to an AI assistant (Claude, ChatGPT, or local LLM)
- Downloaded all four files into a folder named `exploit-lab/`
- Familiarity with: memory corruption (stack/heap overflow), ASLR, NX, stack canaries, Python `pwntools` or `socket`
- Optional: a local Docker container or VM for safe exploit testing (never run on your host)

Lab Scenario

AcmeCorp has a legacy C network service that handles user authentication. A crash dump landed in the bug tracker, and a related CVE was published last week. You have the source code, the crash dump, the CVE details, and an exploit-development template. Your job: diagnose the root cause, determine if the crash is exploitable, and

produce a proof-of-concept that triggers controlled behavior — as a client-ready vulnerability report.

Download Lab Files

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

↓ `vulnerable-server.c.txt` ~4 KB

↓ `crash-dump.txt` ~5 KB

↓ `cve-details.txt` ~7 KB

↓ `exploit-template.py.txt` ~9 KB

Exercise 1 — Vulnerable Server Code Review

Open `vulnerable-server.c.txt`. Small network service, big bugs.

Task 1.1 — Find the vulnerability

You are a vulnerability researcher reviewing a C network service. Analyze the code below and produce:

1. Every memory-safety or logic bug (buffer overflow, format string, integer overflow, use-after-free, etc.).
2. For each bug: line number, bug class, and attacker prerequisites (authenticated? specific input shape?).
3. Identify which bug is most likely to cause a crash under attacker-controlled input.
4. Calculate the offset from the vulnerable write to the saved return address (stack frame analysis).
5. Note which protections would block naive exploitation:
 - Stack canary (-fstack-protector)
 - NX / DEP
 - ASLR / PIE
 - RELRO / Fortify
6. Based on the buffer sizes and control flow, rate the bug:

- Exploitable for RCE (likely / possible / unlikely)
- Exploitable for info leak
- DoS only

[paste vulnerable-server.c.txt]

✓ **Self-check: You should have a specific line number, offset, and exploitability verdict that you could present to a vendor in a responsible-disclosure report.**

Exercise 2 — Crash Dump Analysis

Open `crash-dump.txt`. A crash dump connects "theoretical bug" to "practical exploit primitive."

Task 2.1 — Root-cause and exploitability triage

Below is a crash dump (registers, stack trace, memory around \$rsp/\$rip). Analyze:

1. What faulted? (SIGSEGV on read / write / execute? invalid instruction?)
2. Which register contains attacker-controlled data?
 - RIP = attacker data -> control flow hijack
 - RSP / RBP corrupted -> stack pivot possible
 - RAX/RDI/RSI with attacker value -> arg-controlled call
3. Identify the offset between the input buffer start and the corrupted value (use the crash-dump bytes).
4. Decode any ASCII patterns (de Bruijn / cyclic / "AAAA") that reveal offsets.
5. Rate the crash by exploitability:
 - Control of RIP? (easy RCE with ROP)
 - Partial control? (harder, may need info leak)
 - Read primitive only? (info leak, chained later)
 - DoS only?
6. What is the next step an exploit dev would take?

[paste crash-dump.txt]

✓ Self-check: You should be able to answer "is this exploitable?" with Yes/Probably/No and a specific reason tied to the register state.

Exercise 3 — CVE to PoC Conversion

Open `cve-details.txt`. CVE write-ups describe vulnerabilities in enough detail to reproduce — but the step from "description" to "working PoC" is where AI assistance shines.

Task 3.1 — Turn the write-up into a test case

Below is a CVE write-up. For the purpose of AUTHORIZED DEFENSIVE TESTING, do the following:

1. Summarize the vulnerability in 3 sentences.
2. Identify:
 - Affected component and versions
 - Attack vector (network, local, physical)
 - Prerequisites (auth, specific config)
 - Impact (RCE, info disclosure, DoS, privesc)
3. Extract every indicator needed to build a detection signature (bad strings, malformed fields, protocol anomalies).
4. Write a Python proof-of-concept that:
 - Sends the malicious input
 - Detects whether the target is vulnerable
 - Is SAFE: does not execute any payload, only checks the response for the characteristic vulnerable behavior
 - Could be used by a blue team to inventory vulnerable assets after a patch window
5. Write a Suricata or Snort signature that detects exploitation attempts on the wire.

This is for defensive validation after the vendor patch.

[paste cve-details.txt]

✓ Self-check: Your PoC should be a "vulnerability checker" — it probes for the bug and reports yes/no, but does NOT deliver a malicious payload. This is

exactly the kind of script a patch-verification team would run internally.

Exercise 4 — Exploit Template Completion

Open `exploit-template.py.txt`. A template gives you the scaffolding — AI helps you fill in the gaps.

Task 4.1 — Fill in the blanks

Below is a Python exploit-development template with TODOs and placeholders. Review the code and:

1. Complete every TODO with working code, given the context of exercises 1-3. Use the offset, target, and CVE data you already analyzed.
2. Explain each filled-in section in a comment: "why this byte sequence", "what this return address points to".
3. Add safety rails: connection timeout, target-IP allowlist check, prompt-before-send for dry-run mode.
4. Add a `--check` mode that only verifies whether the target is vulnerable (no payload delivery).
5. Identify any remaining weaknesses:
 - Hardcoded offsets (will break on other versions)
 - Missing error handling
 - No sanity check on the received response

Output the completed file, then a short explanation of each change.

[paste exploit-template.py.txt]

✓ **Self-check:** Your completed script should run **without network access** in `--check` mode against a local target and report clearly whether the target is vulnerable. Never point a real exploit at a system you do not own.

Final Exercise — Build the Vulnerability Report

Task 5.1 — Full disclosure package

Here are four analyses of the same vulnerability (source code review, crash dump, CVE write-up, PoC template). Produce a vendor-ready responsible-disclosure report containing:

1. Executive summary (3 sentences)
2. Affected product and version range
3. Root cause (tied to the specific line in the source code)
4. Impact (CVSS 3.1 base score with justification)
5. Exploitation difficulty (what an attacker needs to succeed)
6. Detailed technical description
7. Reproduction steps using only the --check mode PoC
8. Recommended fix (patch the root cause, not the symptom)
9. Suggested CVE description text
10. Timeline for coordinated disclosure

Format as markdown, max 1500 words.

✓ **Self-check: Your report should connect every artifact: the source code line is referenced in the crash dump analysis, which matches the CVE description, which is reproduced by the PoC. That end-to-end chain is the hallmark of a credible disclosure.**

Reflection

- Where did the AI hesitate or refuse? How did you reframe the ask to get legitimate analysis help?
- How does "writing a vulnerability checker" differ from "writing a weaponized exploit"? Why does the distinction matter for responsible security work?
- What is the biggest risk of AI-generated exploit code in the wrong hands, and how do you mitigate it as a professional?
- Would you ever run AI-generated exploit code without auditing every line yourself? What is your personal policy?

What's Next

Lesson 09 moves from technical exploits to the **human layer** — using AI to analyze phishing campaigns, email headers, and social-engineering pretexts.