Class

5

CSC 472 Software Security

Stack Frames in gdb & Lab 1 Kickoff

Dr. Si Chen (schen@wcupa.edu)

Tue / Thu · 2:00 – 3:15 PM · 25 University Ave., Room 158
Fall 2026 · Week 3 · Thursday, September 10, 2026
Lab 1: Stack & Stack Frame starts today · bring your laptop ·
wcu.ninja/csc472

Recap from Tuesday

- **cdecl**: args pushed **right-to-left**, `call` pushes the return address, result in **EAX**, **caller** cleans up (`add esp, 4n`).
- The frame map: `[ebp+8]` arg1, `[ebp+4]` return address, `[ebp]` saved EBP, `[ebp-4]…` locals.
- Prologue `push ebp; mov ebp, esp; sub esp, N` **builds** the frame; `leave; ret` **destroys** it.
- `a` and `b` existed **twice** on the stack: as `main`'s locals and as the pushed argument copies.
- Today: see all of it in **gdb**, then start **Lab 1**.

PART 1

Reading compiler-generated code

What gcc adds that the C programmer never wrote — and why -O0 is our friend

The same C, two optimization levels

gcc -m32 -O0 (our labs)

```
; int add(int a, int b)
;   { int x = a, y = b; return x + y; }

push ebp
mov  ebp, esp
sub  esp, 0x10
mov  eax, [ebp+8]
mov  [ebp-4], eax      ; x = a
mov  eax, [ebp+0xc]
mov  [ebp-8], eax      ; y = b
mov  edx, [ebp-4]
mov  eax, [ebp-8]
add  eax, edx
leave
ret
```

gcc -m32 -O2

```
mov  eax, [esp+8]      ; b
add  eax, [esp+4]      ; + a
ret

; no push ebp / mov ebp,esp
; no locals in memory
; args read relative to ESP

; labs compile with:
;  -O0 -fno-omit-frame-pointer
;  -fno-stack-protector
;  -z execstack  -no-pie
; so the frame map stays visible.
```

Things gcc adds without asking

Compiler habits to recognize

- **Alignment padding**: `sub esp, 0x10` for two 4-byte locals — the stack is kept **16-byte aligned** (SSE requires it). Offsets are not always "tight".
- `leave` instead of `mov esp, ebp; pop ebp` — one instruction, same effect.
- `lea` for arithmetic: `lea eax, [edx+eax*1]` adds without touching flags.
- **Strength reduction**: `x * 2` → `add eax, eax` or `shl eax, 1`; `x * 5` → `lea eax, [eax+eax*4]`. `imul` is rare for constants.
- **Stack protector** (canary) when enabled: `mov eax, gs:0x14; mov [ebp-0xc], eax` in the prologue — Week 10.

Multiply by a constant

```
; what you might see for  r = p*2 + q*2
mov  eax, [ebp+8]     ; p
add  eax, eax         ; p*2   (not imul)
mov  edx, [ebp+0xc]   ; q
add  edx, edx         ; q*2
add  eax, edx         ; p*2 + q*2

; -O2 might instead say:
mov  eax, [esp+4]
add  eax, [esp+8]
add  eax, eax         ; (p+q)*2
```

gdb defaults to AT&T syntax — know both

Translation rules

- **Operand order reversed**: `mov %esp, %ebp` (AT&T) = `mov ebp, esp` (Intel).
- Registers get `%`, immediates get `$`: `sub $0x10, %esp`.
- Memory: `0x8(%ebp)` = `[ebp+8]`; `(%eax,%edx,4)` = `[eax+edx*4]`.
- Size suffix on the mnemonic: `movl` (32-bit), `movw`, `movb`.
- Fix it once: `set disassembly-flavor intel` (put it in `~/.gdbinit`).

The same 5 instructions, twice

```
(gdb) disas add          # AT&T (default)
  push   %ebp
  mov    %esp,%ebp
  sub    $0x10,%esp
  mov    0x8(%ebp),%eax
  mov    %eax,-0x4(%ebp)

(gdb) set disassembly-flavor intel
(gdb) disas add
  push   ebp
  mov    ebp,esp
  sub    esp,0x10
  mov    eax,DWORD PTR [ebp+0x8]
  mov    DWORD PTR [ebp-0x4],eax
```

PART 2

Inspecting frames in gdb

break · x/ · info frame · bt · finish — evidence for every claim in your report

The frame-inspection toolkit

```
break *add        # FIRST instruction (pre-
prologue)
break add         # AFTER the prologue
run
x/8xw $esp        # 8 words at stack top
p/x $ebp          # a register, in hex
x/xw $ebp+4       # the return address
x/2xw $ebp+8      # the two arguments
info frame        # gdb analyzes the frame
bt                # backtrace: call chain
up / down         # move between frames
display/i $pc     # show next insn per step
stepi / nexti     # one insn (nexti skips calls)
finish            # run to return; shows EAX
x/s $eax          # memory as a string
```

- `break add` vs `break *add`: gdb's plain breakpoint lands **after** the prologue — to watch the prologue run you need the `*address` form.
- `info frame` prints the saved EIP, the saved EBP and where the arguments are — **compare it to your hand-drawn map**.
- `bt` shows every active frame: `#0 add`, `#1 main` — the call chain from the previous slides, live.
- `finish` runs to the return and shows the **return value** — confirms "result in EAX".

Reading a stack dump: what are these numbers?

```
0xffffd508 | +0x0000: 0x00000003   ← $esp
0xffffd50c | +0x0004: 0x00000004
0xffffd510 | +0x0008: 0x00000000
0xffffd514 | +0x000c: 0x00000000
0xffffd518 | +0x0010: 0x00000004
0xffffd51c | +0x0014: 0x00000003
```

- A `x/6xw $esp` dump before a `call`. Each row: **address** | **offset from ESP** | **32-bit value**.
- Words are printed **little-endian-corrected**: `0x00000003` is the value 3, even though the bytes in memory are `03 00 00 00`.
- The same two values appear **twice**, 16 bytes apart. From Tuesday's `main` you know exactly why — and you will explain it in your own words in Lab 1.
- Habit: annotate every dump with **what each word is** (arg, return address, saved EBP, local) before you write a sentence.

From my 2024 Lab 1 handout. Reports that annotate dumps like this get full credit.

Your report must show evidence, not vibes

Evidence standard

- **Bad:** "The function creates a stack frame and stores the variables."
- **Good:** "Lines `push ebp; mov ebp,esp; sub esp,0x10` at `0x565561ad–0x565561b3` create main's frame; `x/4xw $ebp-0x10` after them shows p=3 at `[ebp-0x8]` and q=4 at `[ebp-0x4]` (screenshot 2)."
- Every claim: **instruction address + gdb output + one sentence of interpretation**.
- Draw the frame **by hand** (photo is fine) with the real addresses from your session.
- Screenshots must be **yours** (your prompt, your addresses) — identical screenshots across reports are an integrity case.

What to hand in

```
Report skeleton (PDF, 2–4 pages):

1. Setup: binary, gdb flavor, ~/.gdbinit
2. disas main  – annotated listing
   Q1, Q2, Q3 with x/ evidence
3. disas multiply_by_two – annotated
   Q4, Q5 with evidence
4. One hand-drawn frame diagram
   (main + multiply_by_two frames)
5. One paragraph: what surprised you

Filename: Lastname_Firstname_Lab1.pdf
```

Use AI as a tutor — verify it as a witness

The pilot rule, applied to Lab 1

- **Good uses this week:** "explain `lea eax, [edx+eax*1]`", "what does `info frame` output mean?", "why might gcc reserve 0x10 bytes for two ints?"
- **Bad use:** pasting the lab questions and submitting the answer. Models answer for a **generic** binary; yours has its own addresses, padding, and instruction choices — and I check.
- Research reality: in a 2025 METR randomized trial, experienced developers using AI tools were **19% slower** on their own codebases while *believing* they were 20% faster. The gap between feeling productive and being correct is exactly what gdb closes.
- Rule for Lab 1: every AI-assisted explanation must be **confirmed by a gdb command you ran**. Cite the command.

A verifying workflow

```
Prompt that helps:
"Here is gdb output for a function
 compiled with gcc -m32 -O0:
 <paste disas>
 Explain each instruction. Then tell
 me which gdb commands would CONFIRM
 the values of the locals."

Then RUN those commands.
If the model and gdb disagree,
gdb is right. Write down why.
```

PART 3

Lab 1: Stack & Stack Frame

Your first graded hacking lab — reverse a tiny binary and prove what its frames look like

Lab 1 in one screen

```
# 1. connect (your badger account)
ssh <you>@badger
cd csc472/lab1

# 2. set up gdb once
echo "set disassembly-flavor intel" \
     >> ~/.gdbinit

# 3. analyze
gdb ./lab1
(gdb) disas main
(gdb) disas multiply_by_two
(gdb) break *<addr of the call>
(gdb) run
(gdb) x/8xw $esp     # Q3 evidence
(gdb) stepi ... finish   # see EAX
```

- The target is a **12-line C program**: `main` sets `p = 3`, `q = 4`, calls `multiply_by_two(p, q)`, prints the result.
- **Five questions** (1 pt each): frame creation in `main`; how `p`/`q` are set; the stack dump before the call; the arithmetic in `multiply_by_two` (and why not `mul`); which register carries the result.
- **Deliverable:** a PDF report with annotated `disas` listings, `x/` evidence, and a hand-drawn frame diagram. Upload to **D2L**.
- **Due:** Thursday, **September 17, 2026, 11:59 PM** (D2L is authoritative). Late = no credit.

Full handout: wcu.ninja/csc472 → Lab 1 (PDF). Individual work. Attribution required for anything borrowed.

Lab 1 — start now (I will circulate)

LAB TIME ⏱ rest of class

- **Step 0 (5 min):** log in to badger, add the gdbinit line, `gdb ./lab1`, `disas main`. Raise a hand if anything fails.
- **Step 1:** Before reading any further, **predict** on paper: how many bytes will `main` reserve? Where will `p` and `q` live relative to EBP?
- **Step 2:** Verify with `break *<call addr>`, `run`, `x/8xw $esp`. Mark each word in the dump.
- **Step 3:** `disas multiply_by_two`. Find the arithmetic. Predict EAX, then `finish` to check.
- **Stuck?** Ask a neighbor to read your disassembly aloud — most errors are AT&T/Intel confusion or reading `[ebp+8]` as a local.

Checkpoints before you leave:

□ gdb shows Intel syntax

□ You have disas of BOTH functions saved (copy to a text file)

□ One x/ dump with the call's arguments identified

□ You know what finish printed

Finish the report at home.

Office hours: Tue/Thu 12:30–2:00

(by appt), Wed 1:00–3:00, Rm 131.

Before next class (Tue, Sep 15)

- **Lab 1 report** due **Thu Sep 17, 11:59 PM** on D2L — start tonight while the gdb session is fresh.
- Next week: **system calls, shellcode, and payload structure** — how to make the machine run *your* instructions once you control EIP.
- Prep (20 min): read about `int 0x80` and the Linux 32-bit syscall table (`execve` = 11). Try `strace /bin/true` on badger.
- Quiz 2 is in Week 5 (Sep 22): stack frames, calling conventions, and everything from Lab 1.
- Course site: **wcu.ninja/csc472**. Slides + Lab 1 PDF are posted.

Sources

- GDB manual (sourceware.org/gdb): breakpoints, x command, info frame, backtrace, finish, display.
- GCC manual: -O0/-O2, -fomit-frame-pointer, -fno-stack-protector, -mpreferred-stack-boundary.
- Intel 64 and IA-32 SDM Vol. 2: LEA, LEAVE, IMUL; Vol. 1 §6.2 stack alignment.
- System V ABI i386 supplement (cdecl); AT&T vs Intel syntax: GNU as manual §9.15.3.
- Becker, Rush, Barnes, Rein (METR), "Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity," July 2025 (19% slower; 20% perceived speed-up).
- CSC 472 Fall 2024 Lab 1 handout and stack-dump figure (Dr. Si Chen), adapted for Fall 2026.
- Saumil Shah, "How Functions Work" (frame diagrams, Class 4).