Class

4

CSC 472 Software Security

Calling Conventions & Stack Frames

Dr. Si Chen (schen@wcupa.edu)

Tue / Thu · 2:00 – 3:15 PM · 25 University Ave., Room 158
Fall 2026 · Week 3 · Tuesday, September 8, 2026
Quiz 1 today (first 12 min) · Lab 1 starts Thursday · wcu.ninja/csc472

Quiz 1 — 10 multiple-choice, 12 minutes

- **Four topics only:** IA-32 **registers** (`ESP` / `EBP` / `EIP`), **little-endian** byte ordering, the basic `mov`, and `push` / `pop`.
- **Not on this quiz:** `cmp` and the conditional jumps, the prologue, gdb commands. Those come back later.
- One answer per question. **1 point each, 10 total.** Closed notes: no phones, no AI, no neighbors. I will tell you when to start and stop.
- **Online: wcu.ninja/csc472/quiz1** — enter your **name and student ID**, answer, then Submit. Your answers are saved in the browser as you go, so a refresh loses nothing.

Recap from last class

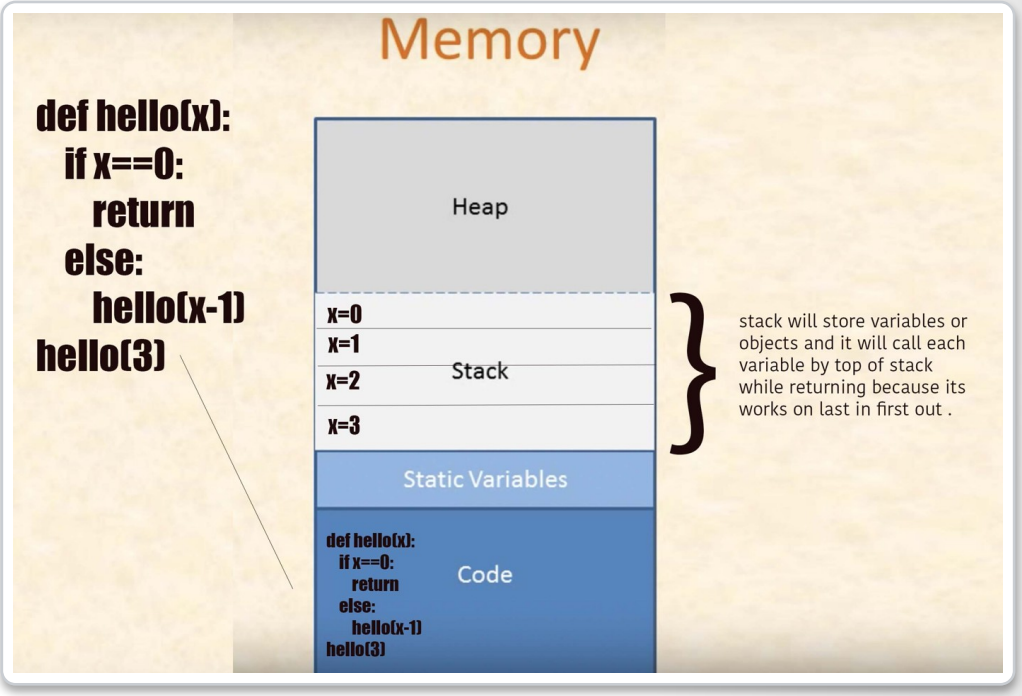
- The stack is **LIFO** and **grows down**: `push` lowers `ESP`, `pop` raises it.
- `call` pushes the **return address**; `ret` pops it into **EIP** — the mechanism every stack attack targets.
- Prologue `push ebp; mov ebp, esp; sub esp, N` — you read one and found: **64 bytes of locals**, `[ebp+8]` = first argument.
- **gdb** is the microscope: `break`, `run`, `info registers`, `disassemble`, `x/16xw $esp`, `stepi`.
- Today: **why** that prologue exists — the **calling convention** and the **stack frame** it builds.

PART 1

Functions need frames

Every call creates a frame; every return destroys one — the stack is the machinery of function calls

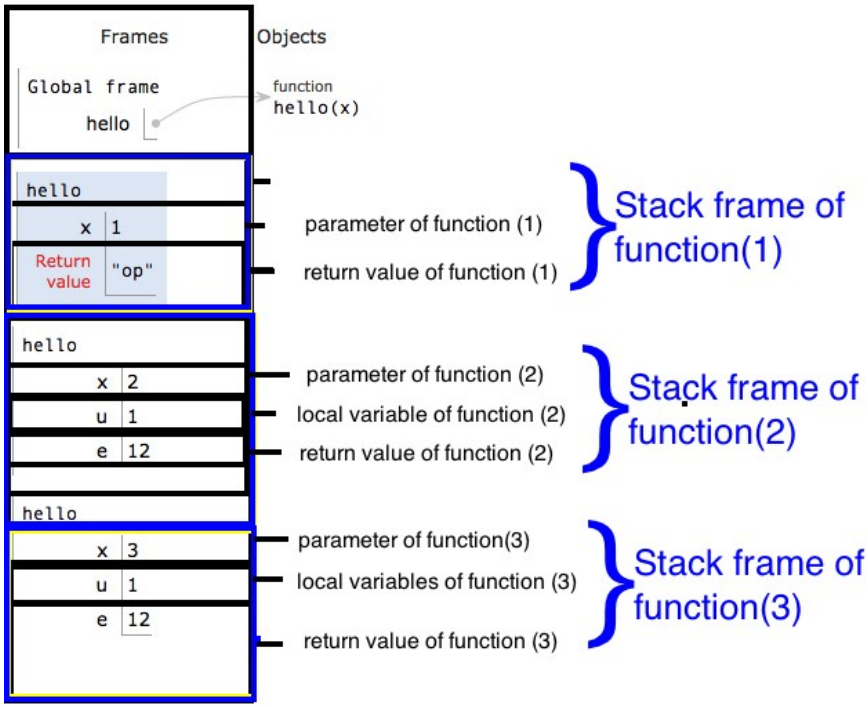
A process in memory



- **Code** (text): the instructions. **Static**: globals. **Heap**: `malloc`. **Stack**: function calls.
- A recursive `hello(3)` pushes `x=3, 2, 1, 0` — **one frame per call**, LIFO.
- The stack is the only region that is **rewritten on every call and return** — that churn is where control-flow data lives.
- Heap security is Week 11; **this month is the stack**.

Reused from ch04 (2024): the stack region holds one frame per active call.

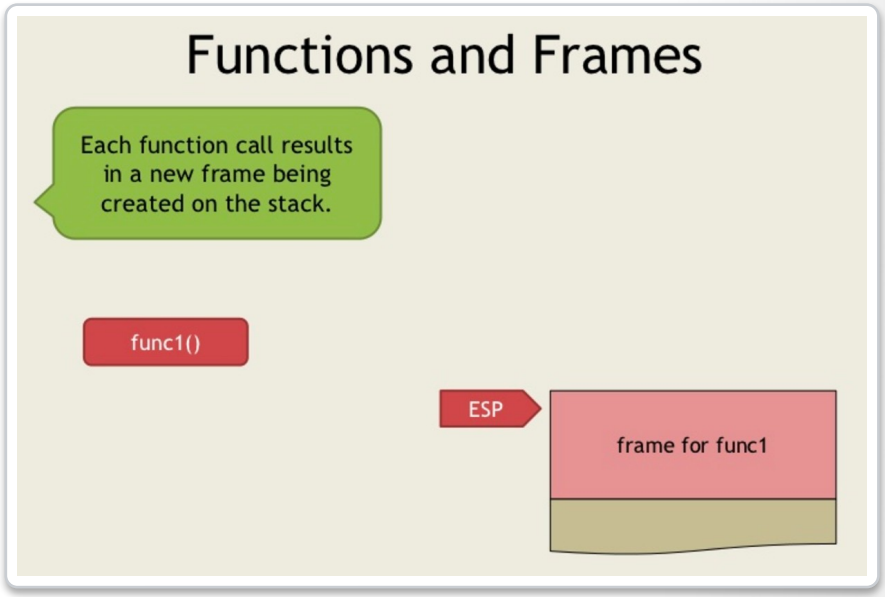
What is inside one frame?



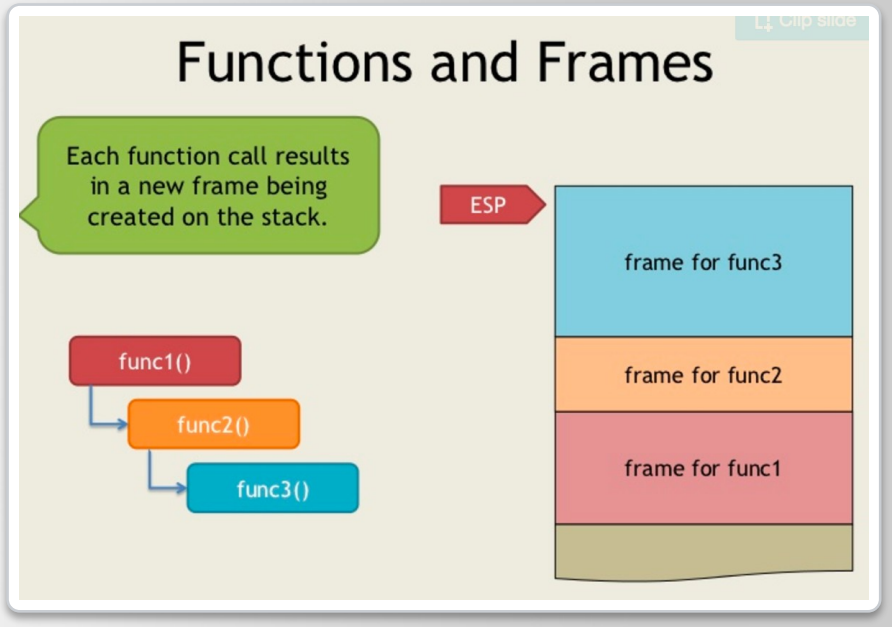
- A **stack frame** = everything one function invocation needs: **parameters**, **local variables**, bookkeeping for **returning**.
- Three calls of `hello(x)` → three frames, each with its own `x`, `u`, `e`.
- In C on x86 the "bookkeeping" is concrete bytes: the **return address** and the **saved EBP**.
- Python Tutor draws it as boxes; the CPU stores it as **32-bit words at ESP-relative addresses**.

Frames as a debugger visualizes them (Python Tutor style) — the same idea we will see in raw memory.

Frames are created on call…

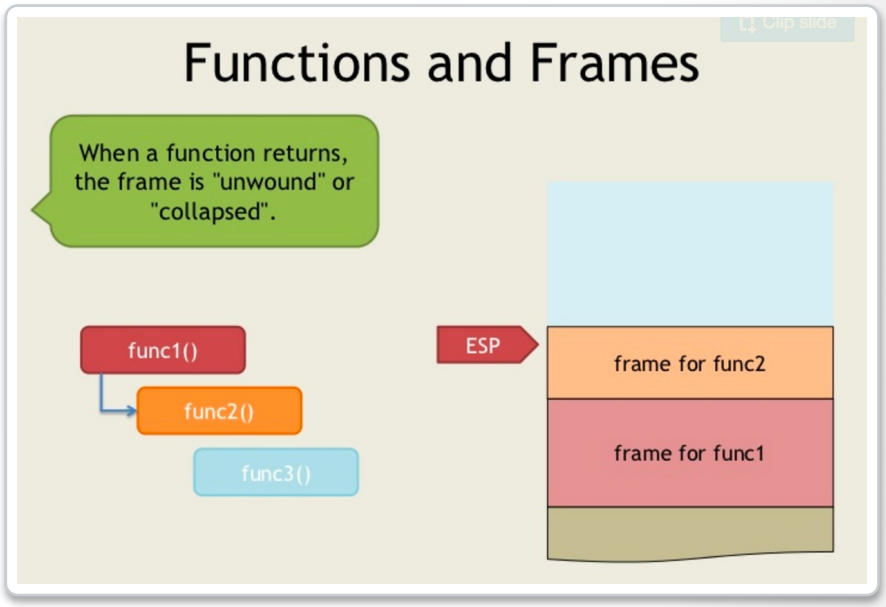


func1() runs: one frame. ESP marks the top.

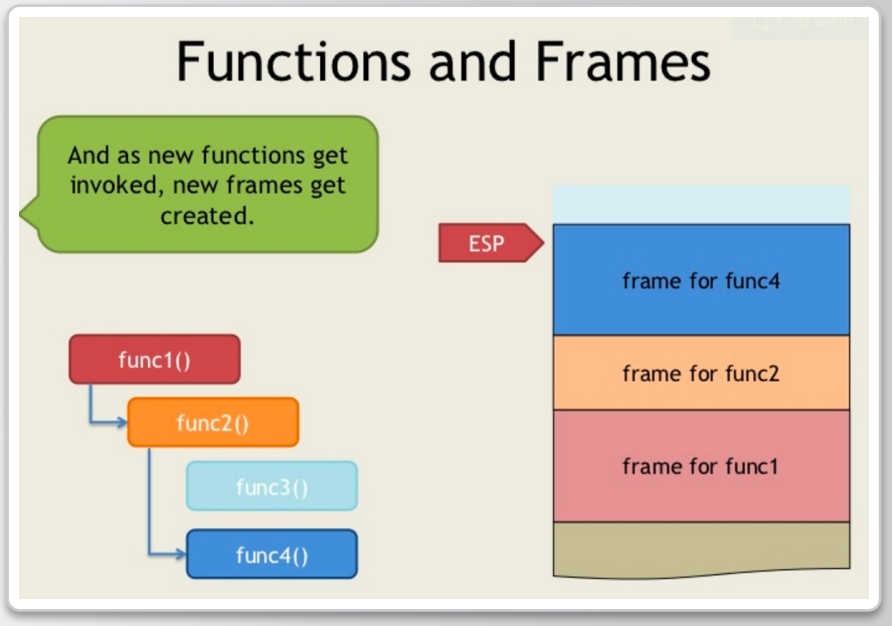


func1 → func2 → func3: each call pushes a new frame; ESP moves down (drawn as "up").

…and unwound on return



func3 returns: its frame is "collapsed" — the memory is NOT erased, just abandoned.



func2 calls func4: the new frame REUSES the same addresses func3 just left.

PART 2

The calling convention (cdecl)

A contract between caller and callee: where arguments go, who cleans up, where the result comes back

cdecl: the caller's side and the callee's side

Caller

- **1.** Push arguments **right-to-left** (last argument first) — so `arg1` ends up at the **lowest** address.
- **2.** `call func` — pushes the **return address**, jumps.
- **3.** After return: **caller removes the arguments** (`add esp, 4×n`).
- **4.** Reads the result from **EAX**.

- Caller-saved registers: **EAX, ECX, EDX** (assume they are clobbered).

Callee

```
func:
  push ebp          ; save caller frame
  mov  ebp, esp     ; my frame base
  sub  esp, N       ; room for locals
  ...               ; args: [ebp+8],[ebp+12]
  mov  eax, ...     ; result in EAX
  mov  esp, ebp     ; epilogue
  pop  ebp          ;   (= leave)
  ret               ; ret addr -> EIP

; callee-saved:
;   EBX, ESI, EDI, EBP
```

Other conventions you will meet

cdecl (32-bit C, Linux)

Args on the **stack**, right-to-left.

Caller cleans up.

Result in **EAX**.

Our labs.

stdcall (Win32 API)

Same argument order.

Callee cleans up: `ret 8`.

Result in EAX.

fastcall

First 2 args in **ECX, EDX**, rest on stack.

Callee cleans up.

Compiler-specific details.

x86-64 System V (Linux 64-bit)

First 6 args in **RDI, RSI, RDX, RCX, R8, R9**.

Result in **RAX**; stack 16-byte aligned.

Return address still on the **stack** — ROP still works (Week 7).

Anatomy of a frame: the EBP-relative map

```
higher addresses
+----------------+
| arg2           | [ebp+12] caller
| arg1           | [ebp+8]  caller
| return address | [ebp+4]  CALL
| saved EBP      | [ebp]    <- EBP
| local 1        | [ebp-4]
| local 2        | [ebp-8]
| buffer[...]    | [ebp-N]  <- ESP
+----------------+
lower addresses   (grows down)
```

- **Positive** offsets from EBP: things the **caller** put there (return address, arguments).
- **Negative** offsets: **my locals**, laid out by `sub esp, N`.
- EBP does not move during the body — so `[ebp+8]` **always** means "first argument". ESP may move (pushes for nested calls).
- A buffer at `[ebp-N]` grows **upward** when overrun: locals → saved EBP → **return address**.

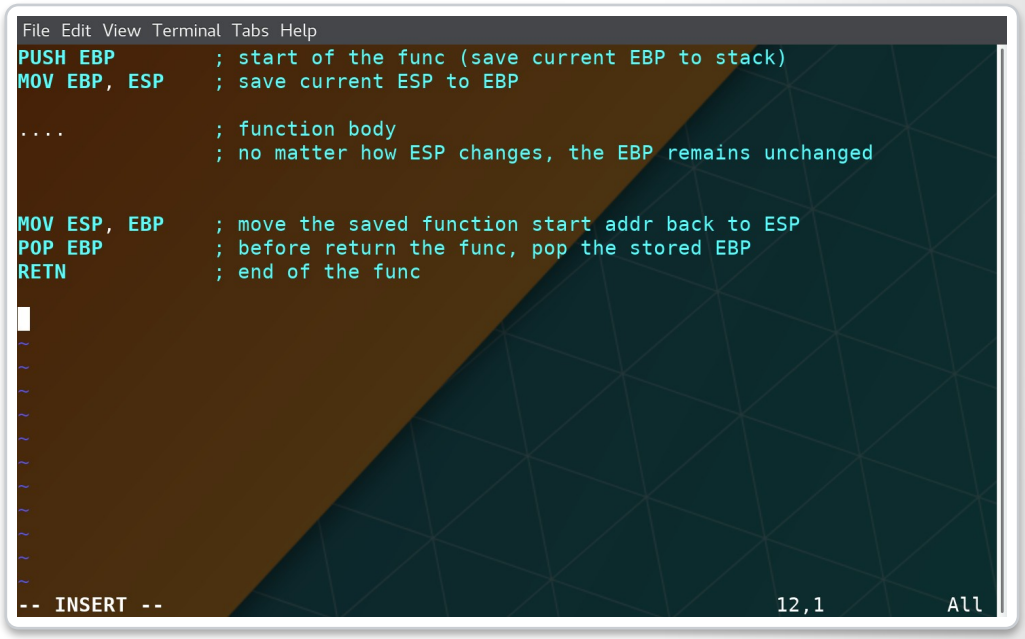
Memorize this picture. Every lab question this month is a question about it.

PART 3

Prologue & epilogue, line by line

Reading compiler-generated code as frame construction and destruction

The prologue / epilogue — my 2024 terminal notes



- `PUSH EBP` — remember the **caller's** frame base.
- `MOV EBP, ESP` — **my** frame base is wherever the stack top is now.
- Body: ESP may change (pushes, calls) but **EBP stays fixed** — a stable anchor for `[ebp±k]`.
- `MOV ESP, EBP` then `POP EBP` — discard my locals, restore the caller's base.
- `RETN` — pop the return address into EIP.

Same code every compiled C function starts and ends with (gcc -O0, 32-bit).

StackFrame.c compiled: gcc -m32 -O0 (Intel)

```
add:
    push ebp
    mov  ebp, esp
    sub  esp, 0x10       ; room for x,y
    mov  eax, [ebp+8]    ; a (arg1)
    mov  [ebp-4], eax    ; x = a
    mov  eax, [ebp+0xc]  ; b (arg2)
    mov  [ebp-8], eax    ; y = b
    mov  edx, [ebp-4]
    mov  eax, [ebp-8]
    add  eax, edx        ; result->EAX
    leave                ; mov esp,ebp
                         ; pop ebp
    ret
```

- `long add(long a, long b) { long x = a; long y = b; return x + y; }` — the C from my 2024 StackFrame.c.
- Arguments arrive at `[ebp+8]` and `[ebp+0xc]` — **exactly** the map from the previous slide.
- Locals `x`, `y` live at `[ebp-4]`, `[ebp-8]` inside the `0x10` bytes reserved (gcc keeps ESP **16-byte aligned**).
- The result is left in **EAX** before `leave; ret`.
- Your gdb output will show addresses and AT&T order unless you `set disassembly-flavor intel`.

Typical gcc -m32 -O0 output for my 2024 StackFrame.c; your offsets may differ slightly.

main's side of the same call

```
main:
    push ebp
    mov  ebp, esp
    sub  esp, 0x10
    mov  DWORD [ebp-4],1 ; a = 1
    mov  DWORD [ebp-8],2 ; b = 2
    push DWORD [ebp-8]  ; b (arg2) FIRST
    push DWORD [ebp-4]  ; a (arg1) last
    call add            ; push ret; jump
    add  esp, 8         ; caller cleanup
    push eax            ; result->printf
    push OFFSET fmt     ; "%d\n"
    call printf
    add  esp, 8
```

- The two `push`es are the **argument copies**: `a` and `b` now exist **twice** on the stack — as `main`'s locals *and* as `add`'s arguments.
- `add esp, 8` after the call is the **cdecl caller cleanup** signature.
- The `push eax; push fmt; call printf` pattern is how you recognize a `printf` call in any binary.

Frame construction, one stepi at a time

What ESP / EBP point to at each step

- **Before call add:** ESP → `[a]`, above it `[b]`, above that main's frame.
- **After call:** ESP → return address (into `main`, the `add esp, 8` line).
- **After push ebp:** ESP → saved EBP (main's frame base).
- **After mov ebp, esp:** EBP = ESP. Now `[ebp+4]` = return address, `[ebp+8]` = a, `[ebp+0xc]` = b.
- **After sub esp, 0x10:** 16 bytes below EBP are `add`'s locals.
- **At ret:** ESP is back at the return address; `ret` pops it into EIP.

The gdb session (Thursday, live)

```
(gdb) break add
(gdb) run
(gdb) x/6xw $esp
0xffffd50c: <ret addr>  0x00000001
            0x00000002  ...
(gdb) stepi             ; push ebp
(gdb) stepi             ; mov ebp,esp
(gdb) p/x $ebp
(gdb) x/xw $ebp+4       ; return addr
(gdb) x/xw $ebp+8       ; arg1 = 1
(gdb) x/xw $ebp+12      ; arg2 = 2
(gdb) info frame
(gdb) finish            ; run to return
(gdb) p $eax            ; -> 3
```

AI knows cdecl. It cannot see YOUR stack.

~56%

of AI-generated code samples passed basic security tests

Veracode, 2025 GenAI Code Security Report

45%

of AI-generated code introduced an OWASP Top-10 weakness in the same study

Veracode, 2025

1st

AI-discovered zero-day exploited in the wild, confirmed by Google Threat Intelligence

GTIG, 2026

Ask an LLM "what is at [ebp+8]?" and it will answer correctly — for a **textbook** function. Ask it about the binary on badger and it is guessing. `x/xw $ebp+8` is not guessing. Use AI to **explain**, use gdb to **verify** — every lab report needs the gdb evidence, not the explanation.

Draw the frame

IN-CLASS ⏱ 8 min

- With a neighbor, on paper. `int f(int p, int q) { int t = p - q; return t * 2; }` is called as `f(7, 3)`.

- **1.** Write the caller's instructions to call it (which value is pushed first?).
- **2.** Draw the frame right after `sub esp, 0x10`: label `[ebp+12]`, `[ebp+8]`, `[ebp+4]`, `[ebp]`, `[ebp-4]`.
- **3.** Which instruction(s) compute `t * 2`? Would you expect `imul` at `-O0`? (Hint: `add eax, eax`.)
- **4.** After `ret`, what does the caller execute next, and why?

Check your answers:

1. push 3 ; push 7 ; call f ;

   add esp, 8 (q first, p last)

2. [ebp+12]=3 (q) [ebp+8]=7 (p)

   [ebp+4]=return addr

   [ebp]=saved EBP
[ebp-4]=t=4

3. add eax, eax (or shl eax,1) —

   ×2 is cheaper as add/shift;

   imul is rare for constants.

4. add esp, 8 — the return address

   points at the caller cleanup.

Lab 1 starts Thursday — before then

- **Lab 1: Stack & Stack Frame** — gdb on a small binary on **badger**; a PDF report answering 5 questions with **gdb evidence** (screenshots + your frame diagrams). Handout posted on **wcu.ninja/csc472** and D2L.
- Make sure your **badger CTF login** works tonight; email me if it does not.
- Put `set disassembly-flavor intel` in `~/.gdbinit` on badger so gdb matches the slides.
- Re-draw today's EBP-relative frame map from memory. Then check it against this deck.
- Thursday: **compiler-generated code in gdb, live** — `disas`, `x/`, `info frame`, `finish` — then you start Lab 1 in class.

Sources

- Saumil Shah, "How Functions Work" (stack-frame figures, reused from my 2024 ch04 deck).
- Intel 64 and IA-32 Architectures Software Developer's Manual, Vol. 1 §6 (procedure calls, stack).
- System V ABI, Intel386 Architecture Processor Supplement (cdecl); AMD64 ABI Draft (x86-64 calling convention).
- GCC manual: -O0 / -fno-omit-frame-pointer; -mpreferred-stack-boundary (16-byte alignment).
- Veracode, "2025 GenAI Code Security Report" (56% pass rate; 45% introduce OWASP Top-10 flaws).
- Google Threat Intelligence Group (GTIG), 2026 report on the first AI-developed zero-day exploited in the wild.
- Python Tutor (pythontutor.com) frame visualization style, as used in my 2024 deck.
- GDB manual: info frame, finish, x command formats.