Class 3

CSC 472 Software Security

x86 Assembly & the Stack

Dr. Si Chen (schen@wcupa.edu)

Tue / Thu · 2:00 – 3:15 PM · 25 University Ave., Room 158
Fall 2026 · Week 2 · Thursday, September 3, 2026
Quiz 1 next class · wcu.ninja/csc472

Recap from last class

IA-32 has 8 general-purpose registers; the exploit-critical three are **ESP** (stack top), **EBP** (frame), **EIP** (next instruction).

x86 is **little-endian** — addresses go into payloads **LSB-first** (`0x08048abc` → `bc 8a 04 08`).

`call` saves a return address on the stack; `ret` pops it **into EIP** — the mechanism we attack.

Today: the instruction set + the stack in detail, watched live in **gdb**.

PART 1

The x86 instruction set

A few dozen instructions do almost everything — start with these

MOV

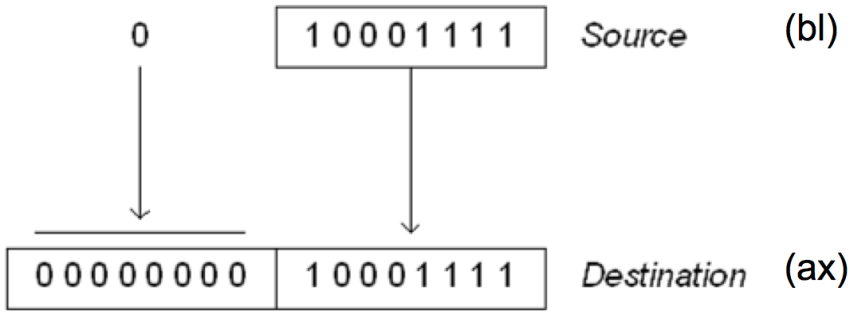
- Move **reg/mem** value to **reg/mem**
  - mov A, B is "Move B to A" (A=B)
  - Same data size

```
mov eax, 0x1337
mov bx, ax
mov [esp+4], bl
```

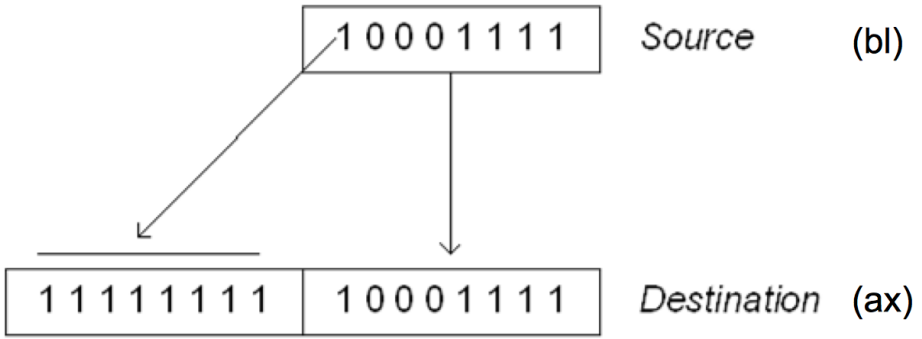
MOVZX / MOVSX

- From small register to large register
- Zero-extend (MOVZX) / sign-extend (MOVSX)
- Example: movzx ebx, al

When copy a smaller value into a larger destination, MOVZX instruction fills (extends) the upper half of the destination with zeros



MOVSX fills the upper half of the destination with a copy of the source operand's sign bit



More About Memory Access

- mov ebx, [esp + eax * 4] **Intel**
- mov (%esp, %eax, 4), %ebx **AT&T**
- mov BYTE [eax], 0x0f
  You must indicate the data size: BYTE/WORD/DWORD

MOV — the workhorse (Intel syntax)

```
mov eax, 0x1337    ; eax = 0x1337   (load an
immediate)
mov bx, ax         ; bx  = ax       (register
to register)
mov [esp+4], bl    ; store bl into memory at
esp+4
mov ebx, [esp]     ; load from memory at esp
into ebx

; source and destination must be the SAME size
movzx ebx, al      ; zero-extend  al (8-bit)
into ebx (32-bit)
movsx ebx, al      ; sign-extend  al into ebx
```

`mov A, B` means **A = B** ("move B into A"). Intel syntax: **destination first**.

Operands can be a **register**, an **immediate** (constant), or **memory** `[...]`.

Source and destination must be the **same size** (`eax`/`ax`/`al` = 32/16/8-bit).

`movzx` / `movsx` move a **small register into a large one**, zero- or sign-extended.

Intel: mov dest, src. AT&T (gdb default) reverses it: mov %src, %dest — watch out.

ADD / SUB

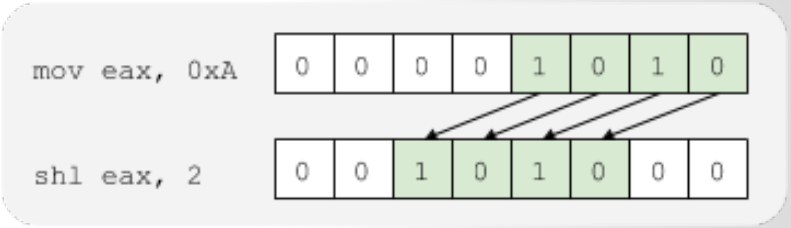
- ADD / SUB
- Normallly "reg += reg" or "reg += imm"
- Data size should be equal
  - ADD eax, ebx
  - sub eax, 123
  - sub eax, BL ; Illegal

INC / DEC

- **inc, dec** — Increment, Decrement
- The **inc** instruction increments the contents of its operand by one.
  The **dec** instruction decrements the contents of its operand by one.
- *Syntax*
  inc <reg>
  inc <mem>
  dec <reg>
  dec <mem>
- *Examples*
  DEC EAX — subtract one from the contents of EAX.
  INC DWORD PTR [var] — add one to the 32-bit integer stored at location *var*

SHL / SHR / SAR

- Shift logical left / right
- Shift arithmetic right
- Common usage: **SHL eax, 2** (when calculate memory address)



Arithmetic, shifts, and memory operands

```
add eax, ebx        ; eax += ebx
sub eax, 123        ; eax -= 123
inc eax             ; eax += 1        dec eax
; eax -= 1
shl eax, 2          ; eax <<= 2  (multiply by 4
— used for indexing)
shr eax, 1          ; eax >>= 1  (logical)
sar = arithmetic

; scaled memory addressing (array indexing):
mov ebx, [esp + eax*4]   ; load 4-byte element
eax of an array
mov BYTE  [eax], 0x0f    ; must state size:
BYTE / WORD / DWORD
```

`add`/`sub`/`inc`/`dec` do the obvious; operands must be the **same size** (`sub eax, bl` is illegal).

`shl x, 2` multiplies by 4 — compilers use shifts for **array indexing**.

`[base + index*scale]` is **scaled addressing** — how `arr[i]` becomes an address.

When the size is ambiguous, you must say `BYTE` / `WORD` / `DWORD`.

PART 2

Control flow: cmp & jumps

How if-statements and loops look in assembly

Jump

- Unconditional jump: jmp
- Conditional jump: je/jne
  and ja/jae/jb/jbe/jg/jge/jl/jle ...
- Sometime with ”cmp A, B” -- compare these two values and set eflags
- Conditional jump is decided by some of the eflags bits.

ref

The JMP Instruction

- JMP (jump) instruction causes an **unconditional jump**
- Syntax is:

JMP destination/target_label

- JMP can be used to get around the range restriction [126/127 byte]
- Flags – no change

```
TOP:

; body of the loop, say 2 instructions
DEC     CX      ; decrement counter
JNZ     TOP     ; keep looping if CX > 0
MOV     AX, BX
```

```
TOP:

; the loop body contains so many instructions
; that label TOP is out of range for JNZ. Solution is-
        DEC     CX
        JNZ     BOTTOM
        JMP     EXIT
BOTTOM:
        JMP     TOP
EXIT:
        MOV     AX, BX
```

Section 6-3: Assembly Language Programming

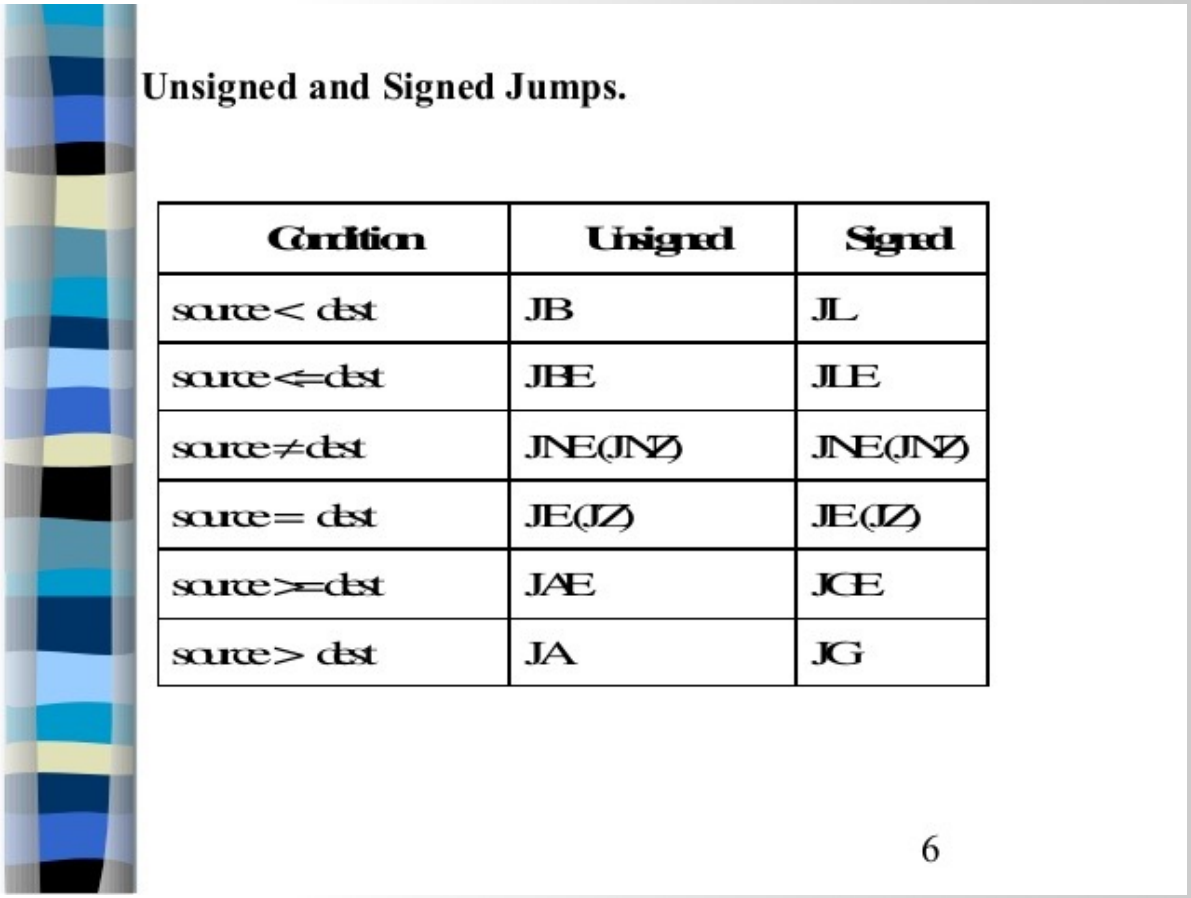
Unsigned and Signed Jumps.

Condition	Unsigned	Signed
source < dst	JB	JL
source <= dst	JBE	JLE
source ≠ dst	JNE(JNZ)	JNE(JNZ)
source = dst	JE(JZ)	JE(JZ)
source >= dst	JAE	JGE
source > dst	JA	JG

6

Jump

- ja/jae/jb/jbe are unsigned comparison
- jg/jge/jl/jle are signed comparison



Unsigned and Signed Jumps.

Condition	Unsigned	Signed
source< dst	JB	JL
source≤dst	JBE	JLE
source≠dst	JNE(JNZ)	JNE(JNZ)
source= dst	JE(JZ)	JE(JZ)
source≥dst	JAE	JGE
source> dst	JA	JG

6

CMP

- **cmp** — Compare
- Compare the values of the two specified operands, setting the condition codes in the machine status word appropriately. This instruction is equivalent to the sub instruction, except the result of the subtraction is discarded instead of replacing the first operand. *Syntax*
  cmp <reg>,<reg>
  cmp <reg>,<mem>
  cmp <mem>,<reg>
  cmp <reg>,<con>
- *Example*
  cmp DWORD PTR [var], 10
  jeq loop
- If the 4 bytes stored at location *var* are equal to the 4-byte integer constant 10, jump to the location labeled *loop*.

cmp + conditional jumps

```
cmp eax, 10         ; compute (eax - 10), set
EFLAGS, discard result
je   label          ; jump if equal      (ZF=1)
jne  label          ; jump if not equal
jmp  label          ; unconditional jump

; SIGNED comparisons:    jg jge jl jle
; UNSIGNED comparisons:  ja jae jb jbe

; C:  if (x > 10) {...}   ->   cmp eax, 10 ;
jle skip ; ...
```

`cmp A, B` is a `sub` that **throws away the result** but **sets EFLAGS** (zero, sign, carry).

A conditional jump (`je`, `jne`, `jg`, …) then branches **based on those flags**.

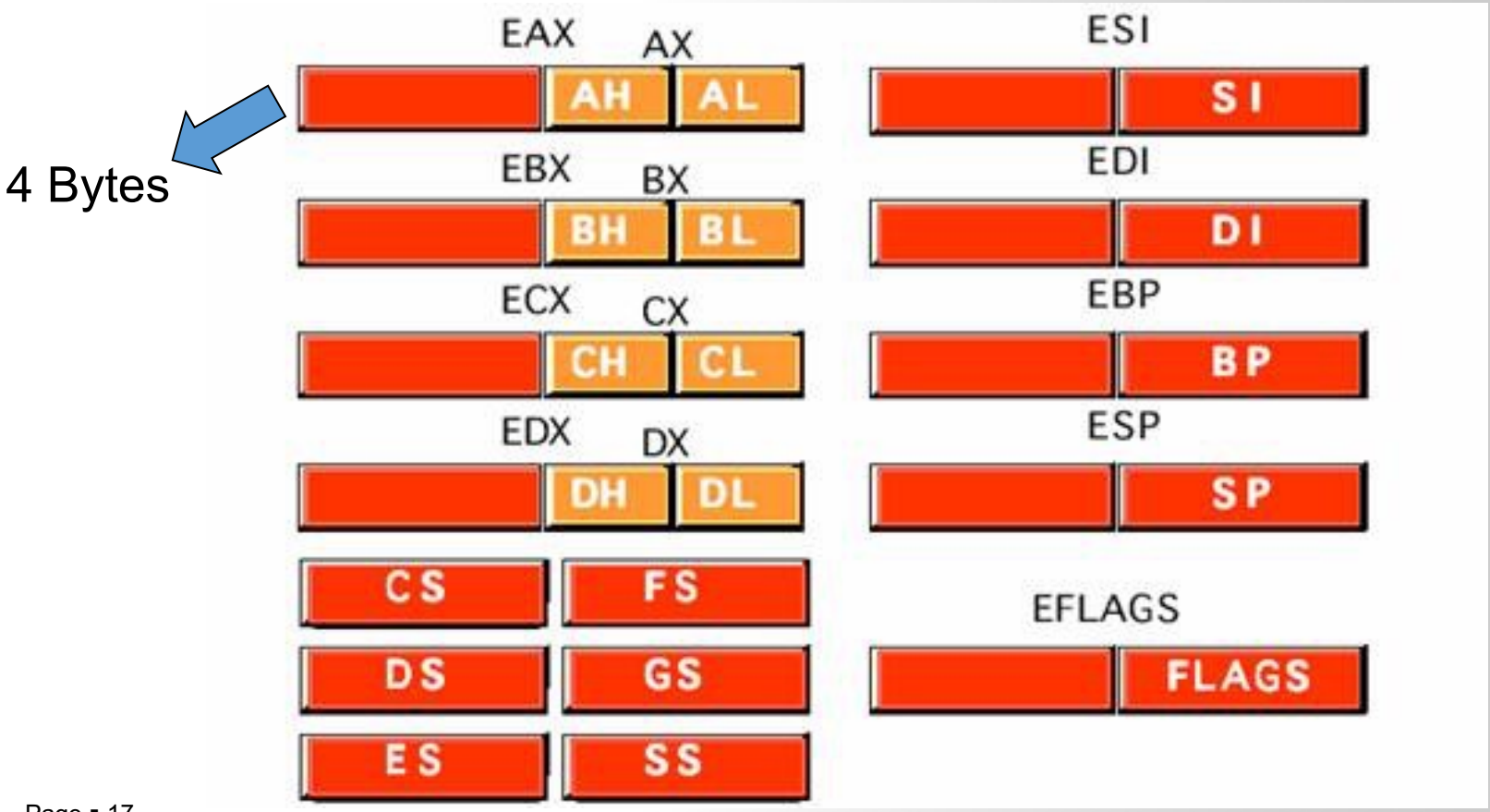
Signed (`jg/jl`) vs **unsigned** (`ja/jb`) comparisons differ — a real source of bugs.

Every `if`, `while`, and `for` compiles down to a **cmp + jump** pattern.

Signed vs unsigned confusion (jg vs ja) is a classic vulnerability — remember the distinction.

General-purpose Registers

- The **eight** 32-bit general-purpose data registers are used to hold operands for logical and arithmetic operations, operands for address calculations and memory pointers



- ## Register
- + `esp` `ebp` `esi` `edi` - *DWORD (32-bit)*
- + `sp` `bp` `si` `di` - WORD (16-bit) - *rarely used*
- + \[esp, ebp\] - *mark the range of stack frame*
- + esi, edi - *used as buffer pointer, some instruction will directly handle esi, edi*

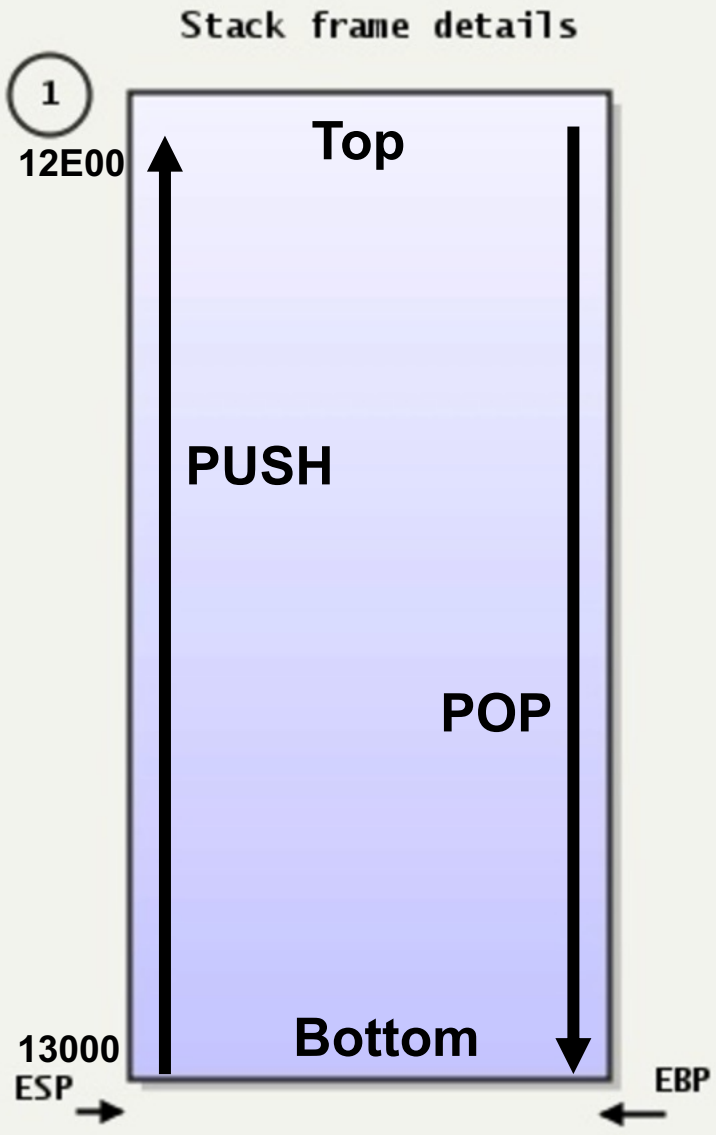
- ## Other Register
- + `eip` - *Program counter, pointing to the current line*
- + `eflags` - *cannot change the value directly, store the instruction result*
- + `cs` `ss` `ds` `es` `fs` `gs` - *segment register*

PART 3

The stack

Where local variables and return addresses live — the battleground

The Stack



Stack:

- A special region of your computer's memory that **stores temporary variables** created by each functions
- The stack is a "**LIFO**" (last in, first out) data structure
- Once a stack variable is freed, that region of memory becomes available for other stack variables.

Properties:

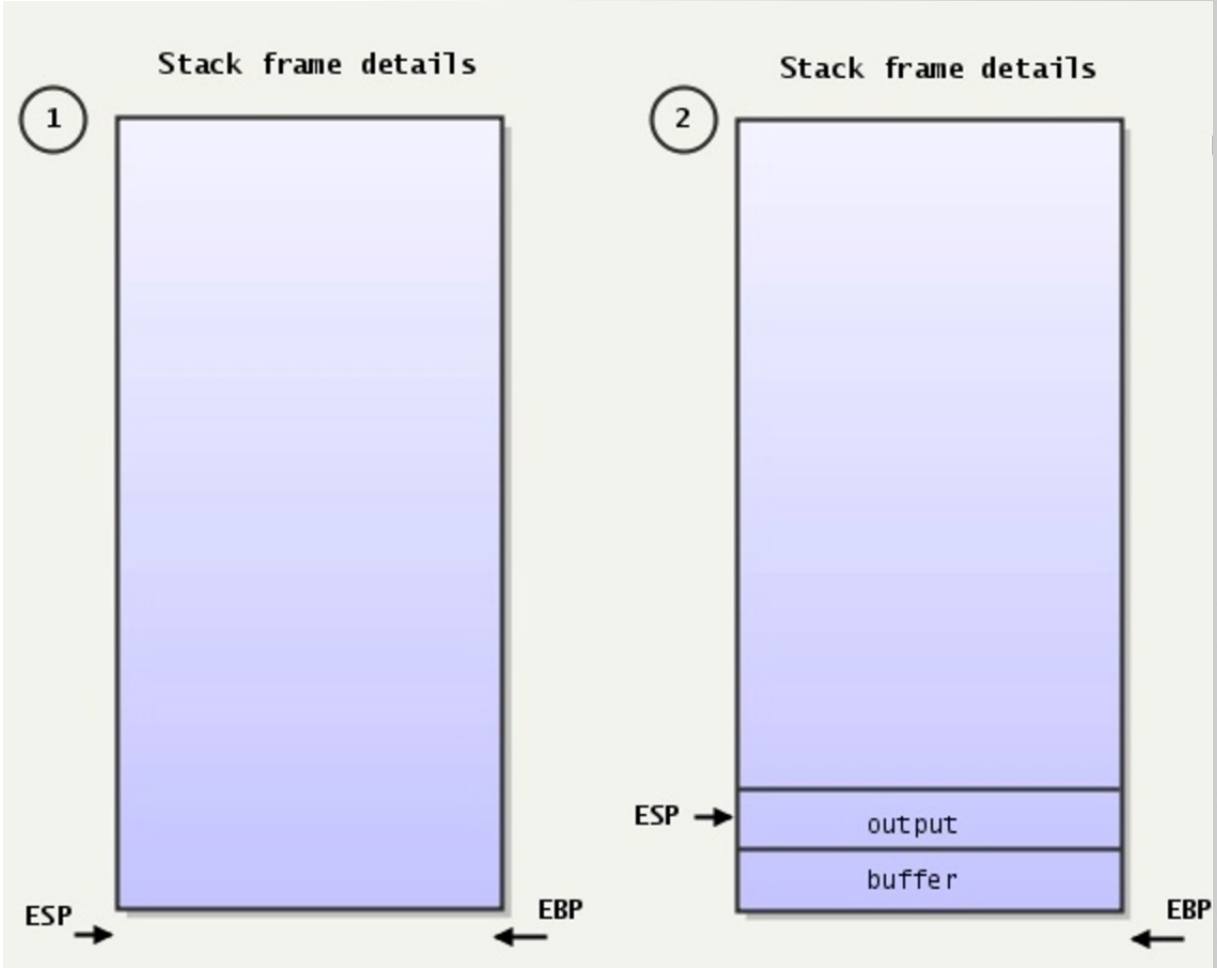
- the stack grows and shrinks as functions **push and pop** local **variables**
- there is no need to manage the memory yourself, variables are allocated and freed **automatically**
- the **stack has size limits**
- stack variables only exist while the function that created them, is running

EBP—Pointer to data on the stack
ESP—Stack pointer

The Stack

Stack:

- A special region of your computer's memory that **stores temporary variables** created by each functions
- The stack is a "**LIFO**" (last in, first out) data structure
- Once a stack variable is freed, that region of memory becomes available for other stack variables.



What the stack is

Concept

A region of memory for **per-function temporary data**: locals, saved registers, return addresses.

It is a **LIFO** structure — last in, first out.

It **grows downward** — `push` moves `ESP` to a **lower** address.

Managed **automatically**; variables vanish when the function returns.

Push / pop & layout

```
push eax   ; ESP -= 4; [ESP] = eax
pop  ebx   ; ebx = [ESP]; ESP += 4

high addr  | ...older frames... |
           | return address     |
           | saved EBP          | <- EBP
ESP -----> | local variables    |
low addr   (stack grows this way)
```

call / ret and the function frame

```
; calling a function:
call func       ; push (address of next
instruction); jmp func

func:           ; standard PROLOGUE
    push ebp        ; save caller's frame
pointer
    mov  ebp, esp   ; set up our frame
    sub  esp, 0x40  ; reserve 64 bytes for
locals (e.g. buffer[64])
    ; ... function body ...
    mov  esp, ebp   ; EPILOGUE: tear down frame
    pop  ebp        ; restore caller's frame
    ret             ; pop return address into
EIP
```

`call` pushes the **return address**; `ret` pops it back **into EIP**.

The **prologue** (`push ebp; mov ebp,esp; sub esp,N`) sets up locals; the **epilogue** tears it down.

`sub esp, 0x40` is literally where `char buffer[64]` lives — on the stack, next to the return address.

Overflow that buffer → overwrite the saved return address → control `EIP`. **This is Lab 2.**

Recognize the prologue/epilogue: it frames every function you will attack.

PART 4

The gdb workflow

Watch the registers and stack change, live — never guess

gdb essentials

```
gdb ./target                  # start
set disassembly-flavor intel # match our slides
(default is AT&T)
break main                    # set a breakpoint
run [args]                    # start running
info registers                # dump EAX..EIP,
EFLAGS
disassemble                   # show asm for the
current function
x/16xw $esp                   # examine 16 words
of memory at ESP
stepi   /  nexti              # execute one
instruction
x/s $eax                      # examine memory
as a string
```

`break` + `run` stop execution where you want to look.

`info registers` shows every register — verify EIP/ESP/EBP against your mental model.

`x/…` **examines memory** (formats: `x`=hex, `w`=word, `s`=string) — read the stack directly.

`stepi` runs **one instruction** — single-step through the prologue and watch ESP move.

Everything is **evidence**: confirm offsets and addresses here, do not guess (or trust AI).

In lab you will find the exact overflow offset with x/ on $esp — measured, not guessed.

Warm-up: read this assembly

IN-CLASS 6–8 min

No computer needed — reason it out with a neighbor.

Given:

```
push ebp
mov ebp, esp
sub esp, 0x40
mov eax, [ebp+8]
```

1. How many bytes of local space did this function reserve?

2. What is at `[ebp+8]` — a local, or something the caller put there?

3. After `sub esp, 0x40`, is `ESP` higher or lower than before? Why does that matter for an overflow?

Answers to check after:

1. 0x40 = 64 bytes of locals.

2. [ebp+8] = the first ARGUMENT

   (caller pushed it before call;

   [ebp+4] is the return address).

3. Lower — the stack grows down.

   A buffer here overflows UP toward

   the saved EBP and return address.

You just read a function prologue.

Quiz 1 + before Tuesday

Quiz 1 is next class — x86 registers, byte ordering, and reading the instructions from today.

Practice in **gdb**: `break main`, `run`, `info registers`, `disassemble`, `stepi` on any small binary.

Re-read the **prologue/epilogue** pattern until you can spot it instantly.

Course site: **wcu.ninja/csc472**. Next: **calling conventions, stack frames, and Lab 1 (Stack & Stack Frame)**.