15/18-213 Recitation 5

SEPTEMBER 22, 2014

PENGJU (JIMMY) JIN

SECTION E

Agenda

- Bomb lab reminder
- Machine Programming Procedures
  - Stack Frames
  - Function calls in x86(IA32) and x86-64 (briefly)
- Demo
- Buffer lab Preview

Reminder

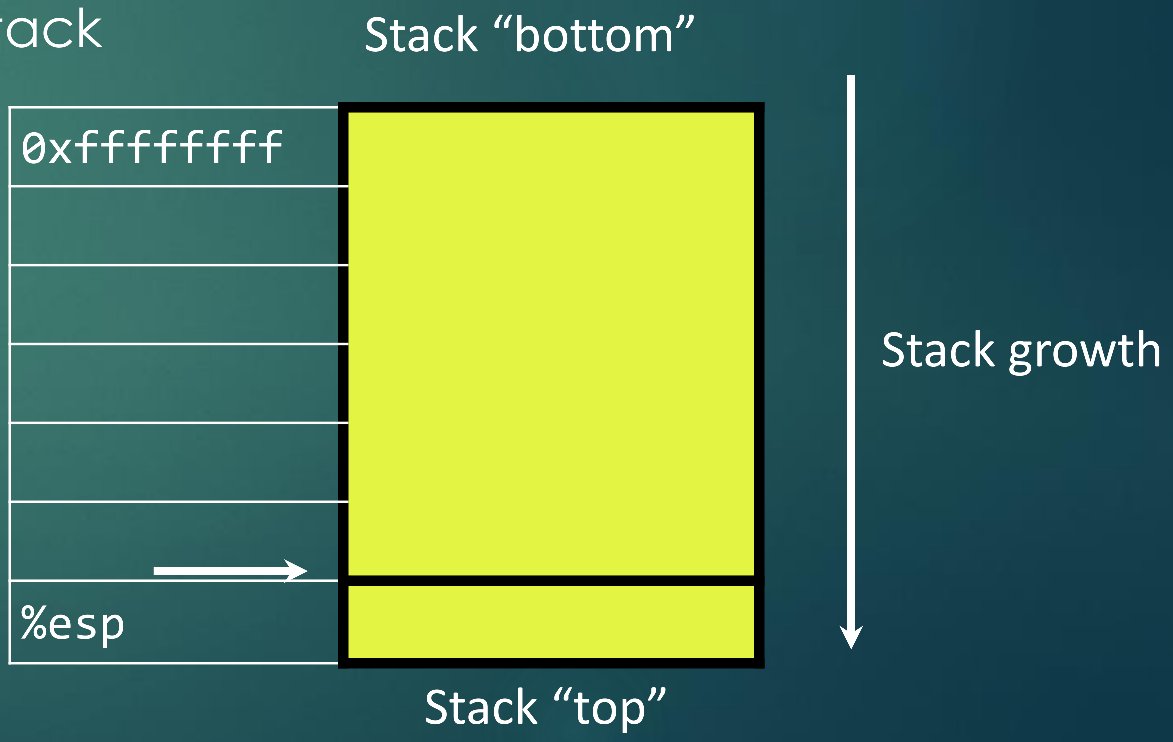
- In case you didn’t know, bomb lab is due tomorrow.
- If you haven’t started it yet, good luck.
- Buflab comes out tomorrow night.

Register Recap

- Caller saved: %eax, %ecx, %edx
  - Must be saved **before** a function call (by the caller) if needed.
- Callee saved: %ebx, %edi, %esi
  - Must save these (by the callee) before any work if needed in the child function.
- Base pointer: %ebp (IA32)
- Stack pointer: %esp
- Instruction pointer: %eip

IA32 Stack

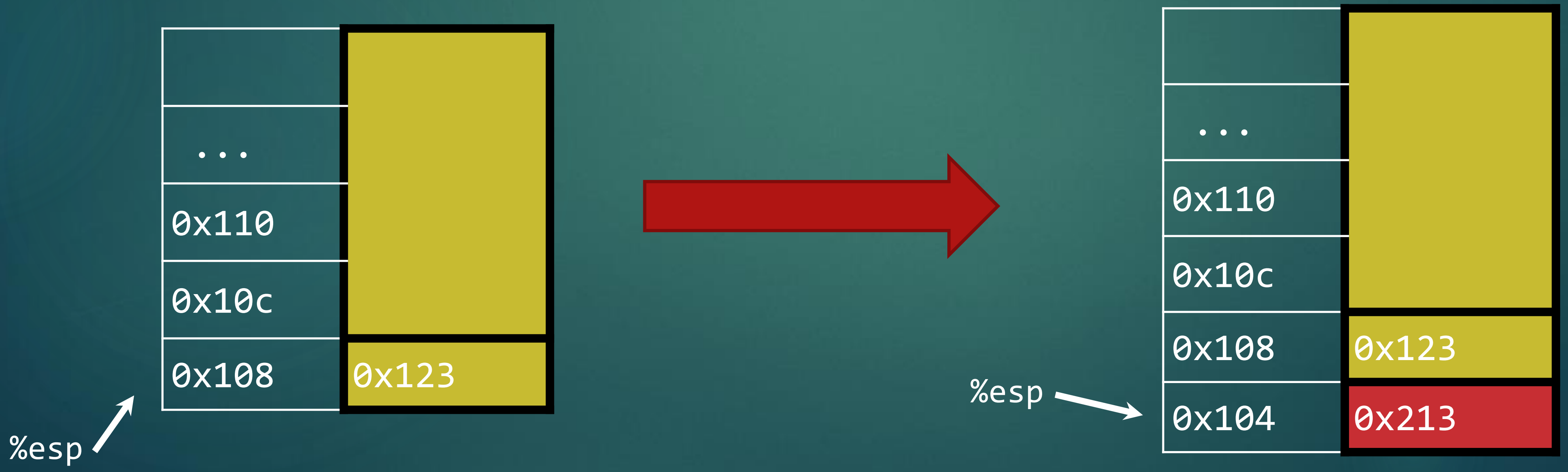
- "Upside down" stack
- Grows downward => new things are pushed to a lower memory address
- %esp holds the ADDRESS of top of the stack
- %esp has the lowest address



Stack Operation - Push

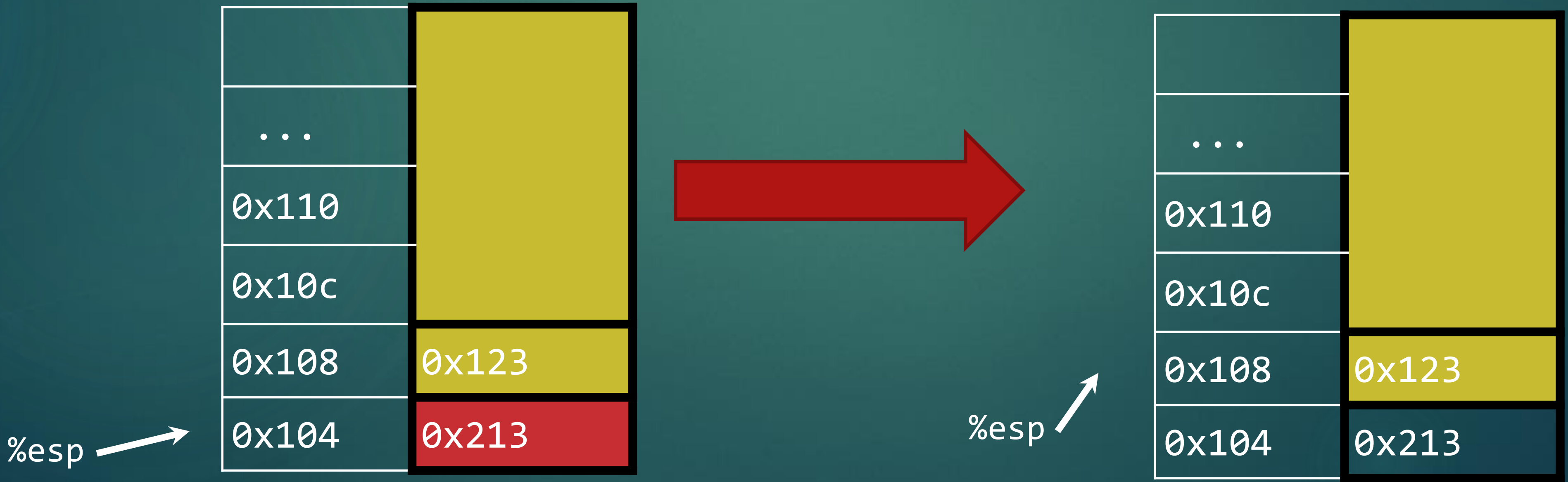
- pushl src → subl $4, %esp

  movl src, (%esp)
- Frist move the stack pointer to a lower (empty) address
- Then move the value into the empty address.



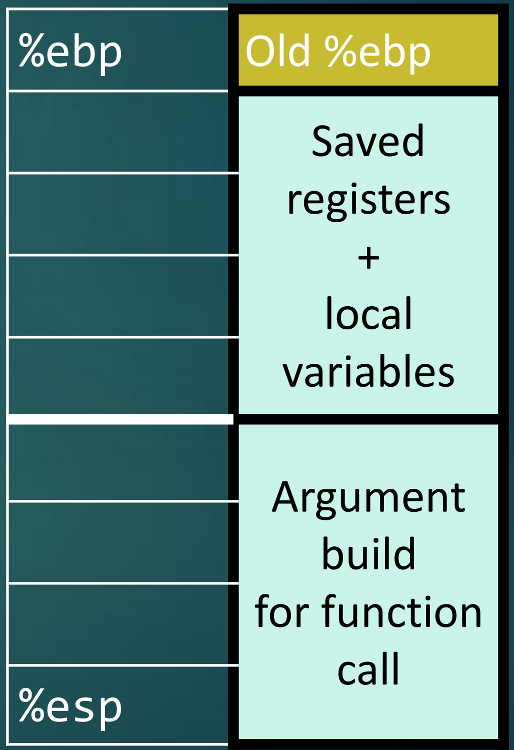
Stack Operation - pop

- popl dest → movl (%esp), dest
  addl $4, %esp
- Move the value stored at the top of the stack to dest.
- The address now becomes empty, move the stack pointer up.



Stack frames

- Every function call is given a new stack frame.
- Stack frames are in the stack memory region growing down.
- Things included in the frame:
  - Local variables (scalars, arrays, structs)
    - Scalars: if the compiler couldn't allocate enough registers
  - Space to save callee saved registers
  - Space to put computations
  - A way to give arguments and call other functions
  - A way to grab arguments

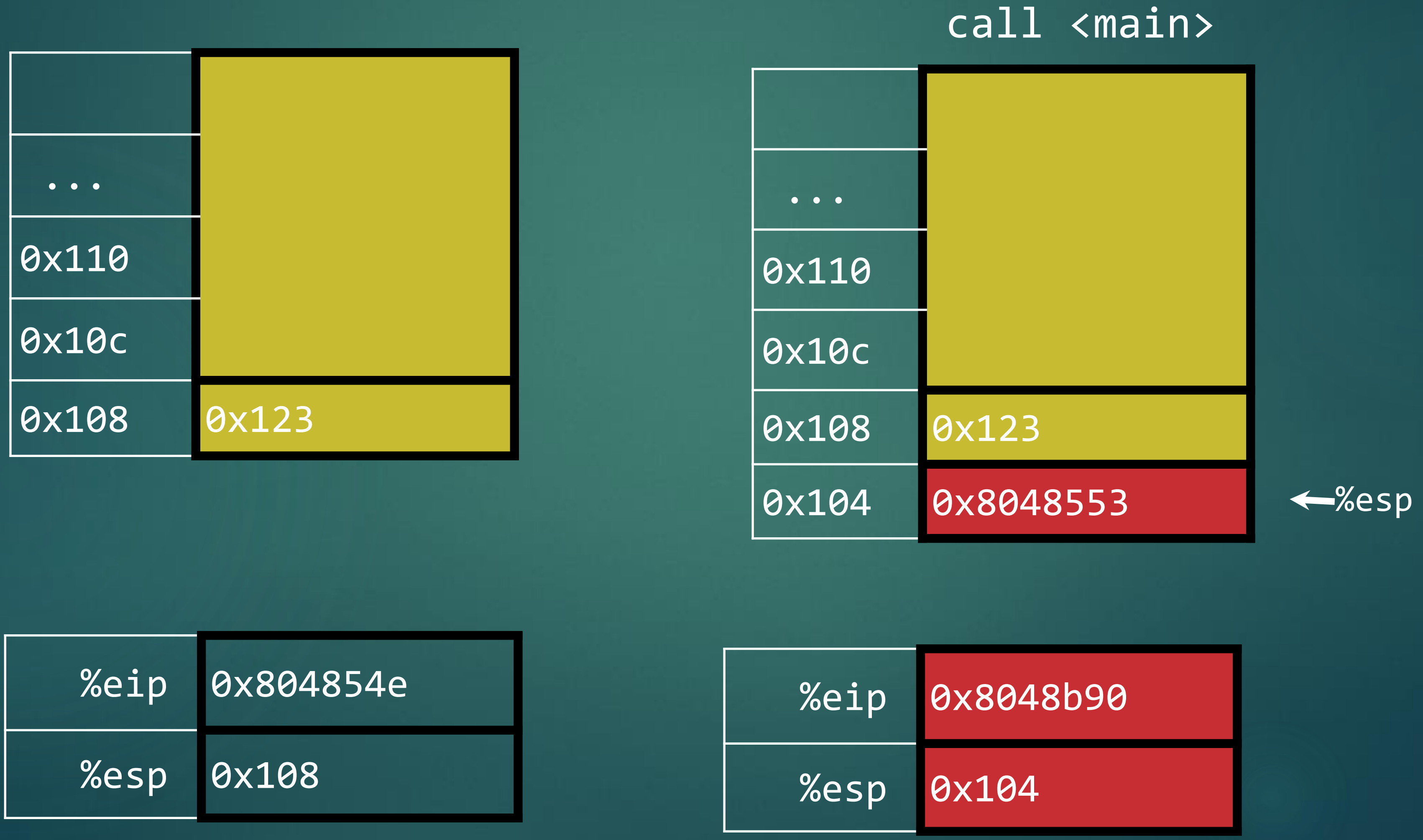

%ebp
Old %ebp
Saved registers + local variables
Argument build for function call
%esp

Function Calls - setup

- Shifting the stack frame when a function is called.
- Creating a new stack frame:
    - Parent function: call label (ex: call <add>)
        - Push the return address (the address of next instruction after the call)
    - Child function: push %ebp, move $esp to %ebp, decrease $esp
        - Save the old $ebp to the stack
        - Move the $esp to $ebp, $ebp now points at the $ebp of the parent function.
        - Decrease $esp, creating space for new function.

Visualization

- `804854e: e8 3d 06 00 00   call 8048b90 <main>`
- `8048553: 50                pushl %eax`



...	
0x110	
0x10c	
0x108	0x123

%eip	0x804854e
%esp	0x108

call <main>

...		
0x110		
0x10c		
0x108	0x123	
0x104	0x8048553	←%esp

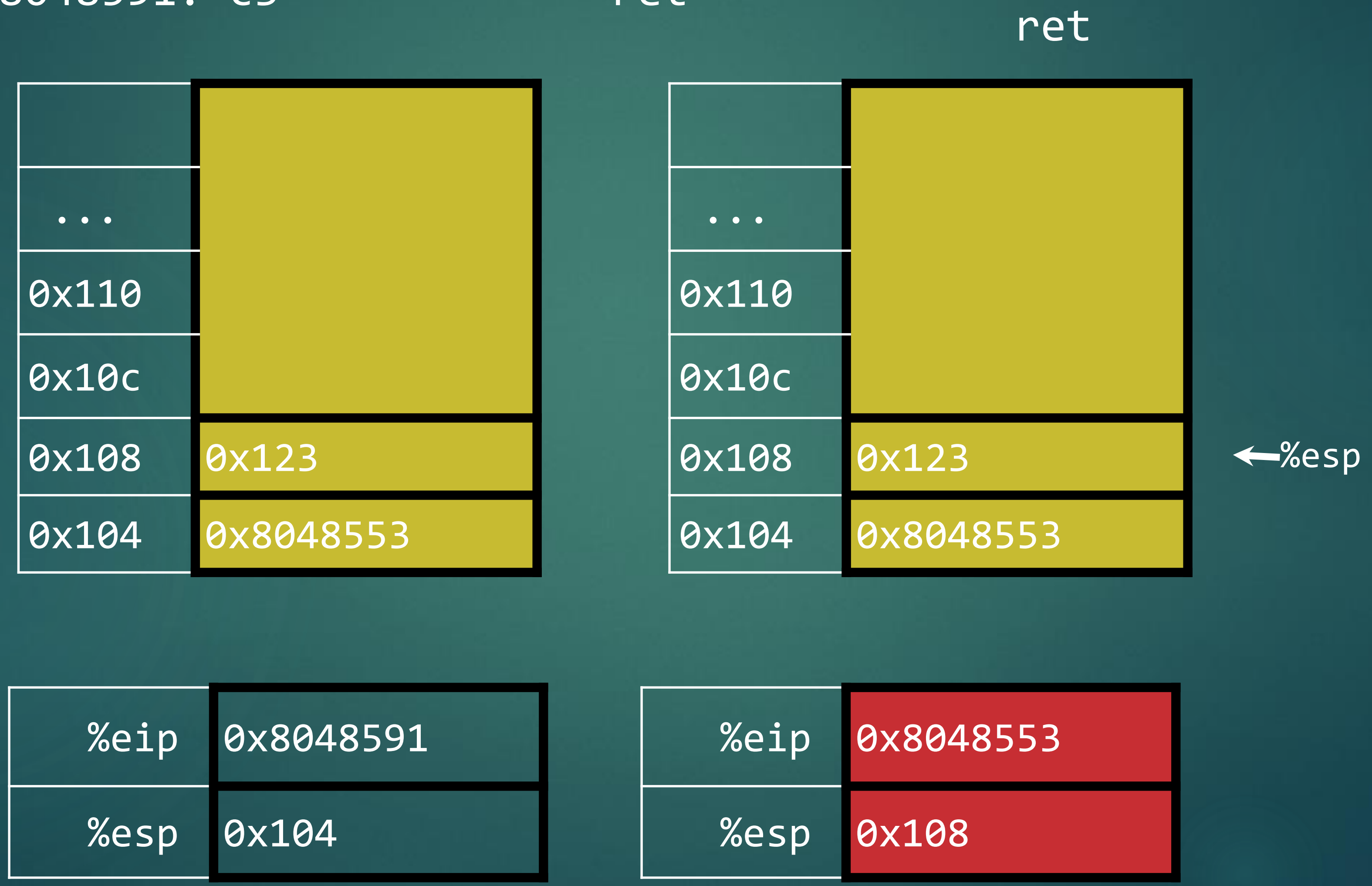
%eip	0x8048b90
%esp	0x104

Function Calls - Return

- Child function calls **leave** then **ret**
- **leave** – two machine operation combined into one
  - move the $ebp to $esp (esp now points at the base of the stack)
  - pops the stack into $ebp (ebp is restored back to the ebp of parent function)
- **ret** – pop return address into $eip, the function call is over.

Returning

- `8048591:  c3                    ret`



...	
0x110	
0x10c	
0x108	0x123
0x104	0x8048553

%eip	0x8048591
%esp	0x104

ret

...		
0x110		
0x10c		
0x108	0x123	←%esp
0x104	0x8048553	

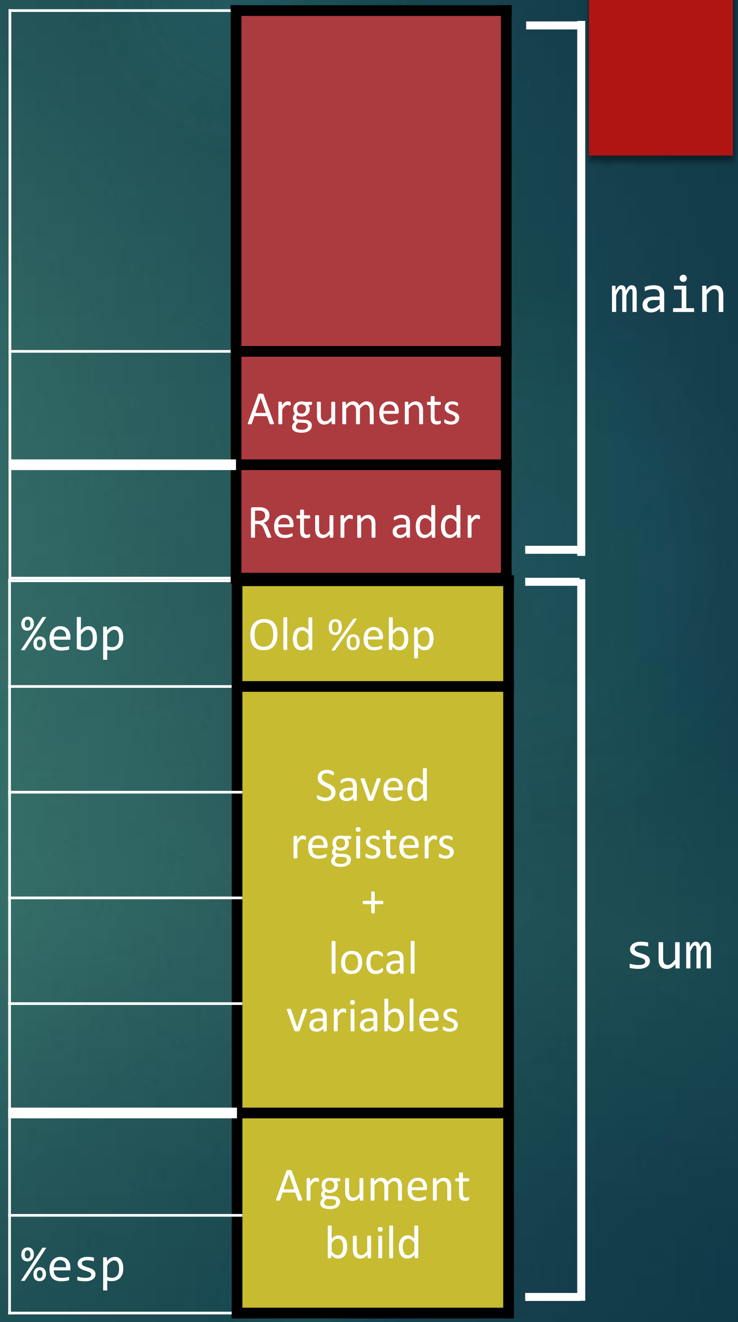
%eip	0x8048553
%esp	0x108

Function calls and stack frames

- Suppose you have

```
int main(void)
{
    int x = 3;
    return sum(x, 0);
}
```

- sum grabs arguments by reaching up the caller's stack frame!
- If we scale up this example, we see that arguments should be pushed in reverse order



Demo

X86-64

- If you understand 32bit machines, 64 bit is easy.
  - No more frame pointers (%ebp is now a free register)
  - Many arguments are passed in registers
  - Less stack manipulation, more use of registers
- Overall a lot less stack usage
  - Good for performance
- You are expected to know how the stack works for 64 bits

Buffer lab Overview

- Hacking the IA32 function call procedure.
- Overflows the stack frame memory space and over-writes some important information (return address).
- A thorough understanding of procedure call is needed.

Details on Buffer Lab

- Disassembling your code using objdump
- USE GDB
- Find out how long to make your inputs.
- Write exploits to divert program execution

Buffer Lab Tricks

- Canaries
  - Detect overrun buffers
  - Sit at the end of the buffer
  - If the array overflows, hopefully we can detect this with a change in the canary value
- NOP sleds
  - The nop instruction means “no operation”
  - Used to “pad” instructions (or exploits)

Buffer Lab Tools

- ./makecookie andrewID
  - Makes a unique “cookie” based on your AndrewID
- ./hex2raw
  - Use the hex generated from assembly to pass raw strings into bufbomb
  - Use with –n in the last stage
- ./bufbomb –t andrewID
  - The actual program to attack
  - Always pass in with your AndrewID so you will be graded on autolab
  - Use with –n in the last stage

How to Input Answer

- Put your byte code exploit into a text file
  - Then feed it through hex2raw
- Later stages: write(corruption) assembly
  - Compiling
  - Gcc –m32 –c example.s
  - Get the byte codes
  - Objdump –d example.o > outfile
  - Feed it thrught hex2raw

Buffer Lab Hint

- **Read every line of the handout.**
- **Good luck have fun**