Ethical Hacking and Buffer Overflow

CMPSC 403 Fall 2021

September 14, 2021

Take the Reflection Quiz 3

Check Discord for the link

Ethical Hacking

Ethical Hacking

- **Ethical Hacking** aka **penetration testing** aka **white hat hacking**
    - Legal hacking using the same tools and techniques penetrators use
    - Do not damage or steal
    - Evaluate systems security and report back
- Other types of hacking
    - Black hat hacking: hackers participating in malicious or destructive activities
    - Gray hat hacking: use their skills sometimes for personal gain and for common good
- CEH certification

Secure Programs

When are programs secure?

- <u>Formally</u>: When it does exactly what it should
  - No less, no more
  - How do we know what it is supposed to do?
- <u>In practice</u>: When it does **not** do bad things
  - Delete/corrupt files
  - Crash the system
  - Send passwords
- What if the program does not do bad things but could?

Unintended Functionality Leads to Unintended Consequences

- Complex system contain unintended functionality
- This unintended functionality can be triggered
    - Exploitation of **vulnerabilities**
- **Software vulnerability:** a bug that allows an unprivileged user capabilities that should be denied to them
    - Many types, including:
        - Bugs that violate “control flow integrity”

Exploiting Vulnerabilities

- Low-level details of how exploits work
    - How can a remote attacker get victim program to execute their code?
- <u>Threat model</u>: Victim code is handling input that comes from across a security boundary
- <u>Security policy:</u> Want to protect integrity of execution and confidentiality of data from being compromised by malicious and highly skilled users of the system

x86 Assembly and Call Stack

Number Representation

Units of Measurement

- In computers, all data is represented as bits
    - **Bit**: a binary digit, 0 or 1
- Names for groups of bits
    - 4 bits = 1 **nibble**
    - 8 bits = 1 **byte**
-

Hexadecimal

- 4 bits can be represented as 1 hexadecimal digit (base 16)
- Note: For clarity, we add **0b** in front of bits and **0x** in front of hex

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

Binary	Hexadecimal
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Think Along: Bits and Hex

- How many bits/nibbles/bytes in 0b1000100010001000?
  - Note: For clarity, we add 0b in front of bits
  - Answer: 16 bits, 4 nibbles, 2 bytes

- How would you write 0b11000110 in hex?
  - Note: For clarity, we add 0x in front of hex
  - Answer: 0xC6

Running C Programs

CALL (Compiler, Assembler, Linker, Loader)

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

C code

Compiler →

```
add_one:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
    mov 8(%ebp), %eax
    mov %eax, -4(%ebp)
    inc -4(%ebp)
    mov -4(%ebp), %eax
    leave
    ret
```

Assembly code
(RISC-V, x86)

Assembler →

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

Machine code
(raw bits)

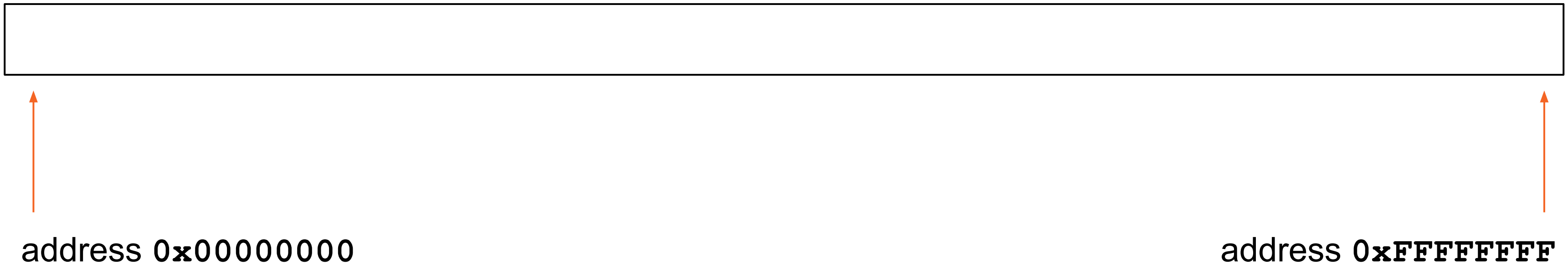
CALL (Compiler, Assembler, Linker, Loader)

- **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
- **Loader**: Sets up memory space and runs the machine code

Memory Layout

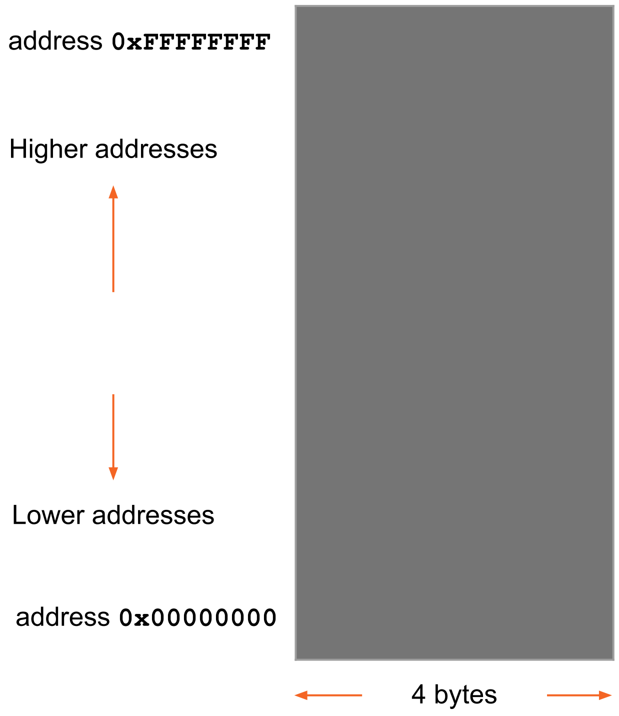
C Memory Layout

- At runtime, the loader tells your 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, memory has 64-bit addresses
- Each address refers to one byte, which means you have 2^{32} bytes of memory



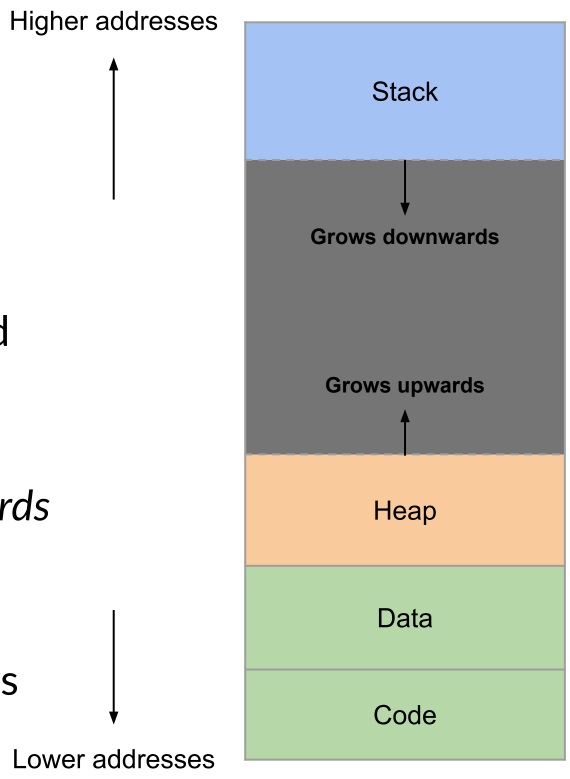
C Memory Layout

- Often drawn vertically for ease of viewing
  - But memory is still just a long array of bytes



x86 Memory Layout

- Code
  - The program code itself (also called “text”)
- Data
  - Static variables, allocated when the program is started
- Heap
  - Dynamically allocated memory using **malloc** and **free**
  - As more and more memory is allocated, it grows *upwards*
- Stack
  - Local variables and stack frames
  - As you make deeper and deeper function calls, it grows *downwards*



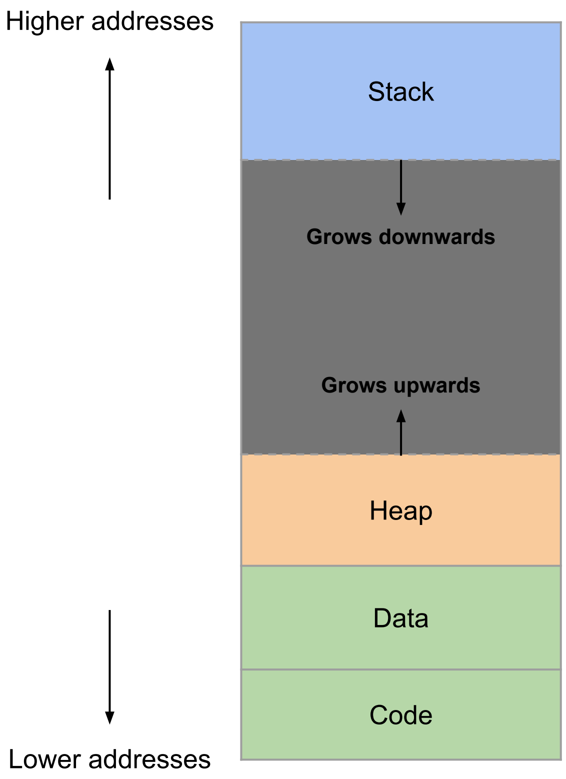
<u>Think Along</u>: Stacks and Heaps

Name **three** differences between stacks and heaps

- Stack variables can’t be resized, heap variables can be resized
- Stack-based memory allocation is faster than heap-based allocation (last-in, first-out)
- Heaps have garbage collection and stacks do not
- Stacks are limited in size
- Stacks are allocated in a contiguous block

Registers

- Registers are located on the CPU
  - This is different from the memory layout
  - Memory: addresses are 32-bit numbers
  - Registers: addresses are names (**ebp**, **esp**, **eip**)



X86 Architecture

x86

- It’s the most commonly used instruction set architecture in consumer computers!
  - You are probably using an x86 computer right now…unless you’re on a phone, tablet, or recent Mac
- Little-endian
  - The least-significant byte of multi-byte numbers is placed at the first/lowest memory address
- Variable-length instructions
  - When assembled into machine code, instructions can be anywhere from 1 to 16 bytes long

X86 Registers

- Storage units as part of the CPU architecture (not part of memory)
- Only 8 main general-purpose registers:
  - EAX, EBX, ECX, EDX, ESI, EDI: General-purpose
  - ESP: Stack pointer (similar to **sp** in RISC-V)
  - EBP: Base pointer (similar to **fp** in RISC-V)
  - We will discuss ESP and EBP in more detail later
- Instruction pointer register: EIP

X86 Syntax

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

X86 Assembly

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

Opcode | Source | Destination

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

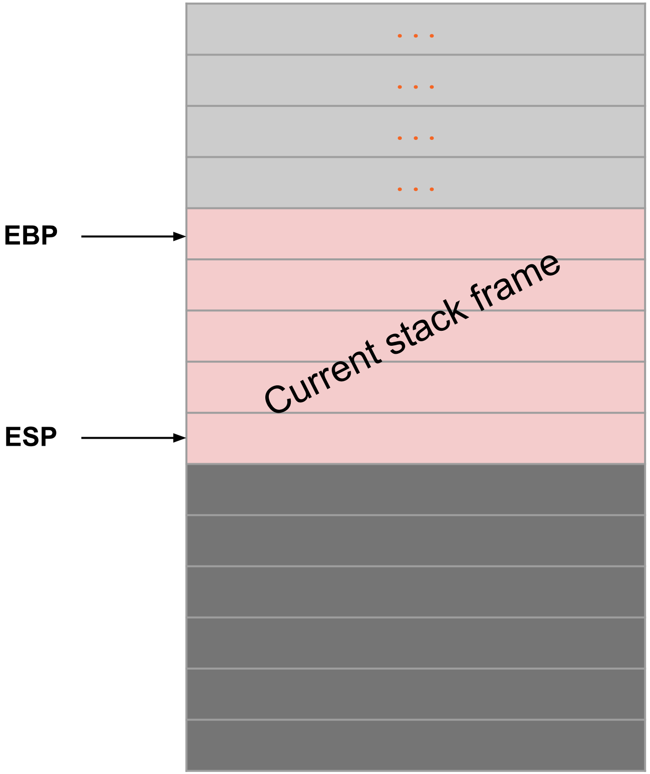
Stack Layout

Stack Frames

- When your code calls a function, space is made on the stack for local variables
  - This space is known as the **stack frame** for the function.
  - The stack frame goes away once the function returns.
- The stack starts at higher addresses. Every time your code calls a function, the stack makes extra space by growing down
  - <u>Note</u>: Data on the stack, such as a string, is still stored from lowest address to highest address. “Growing down” only happens when extra memory needs to be allocated.

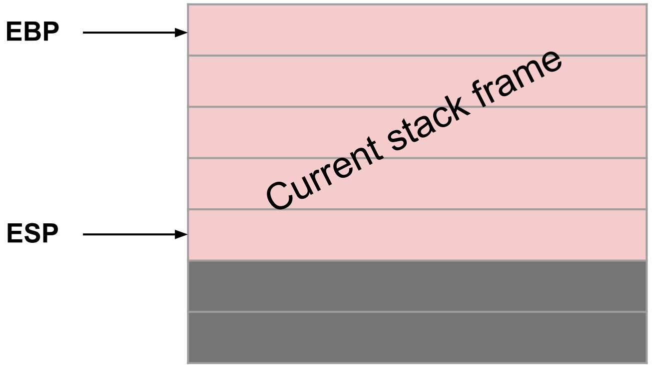
Stack Frames

- To keep track of the current stack frame, we store two pointers in registers
  - The EBP (base pointer) register points to the top of the current stack frame
  - The ESP (stack pointer) register points to the bottom of the current stack frame



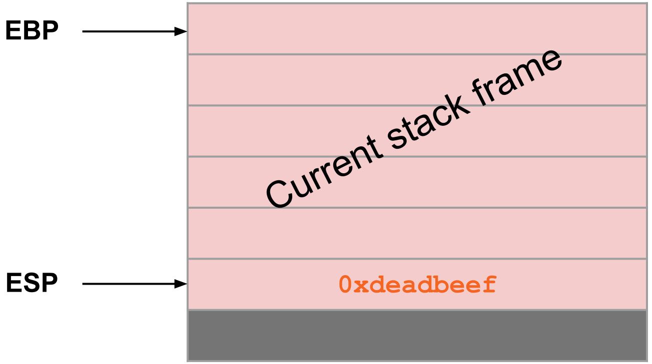
Pushing and Popping

- 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 of the stack



EAX = `0xdeadbeef`
EBX = ...

Before **`push %eax`**

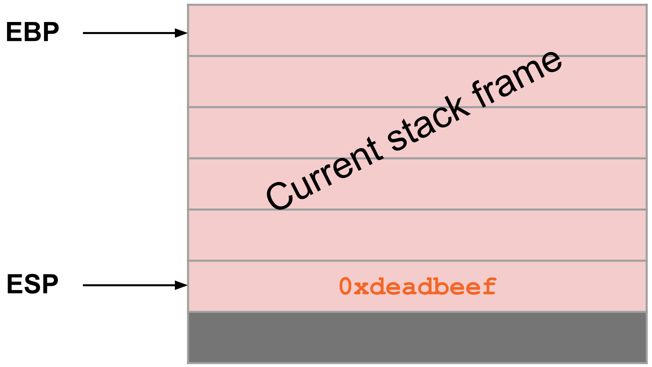


EAX = `0xdeadbeef`
EBX = ...

After **`push %eax`**

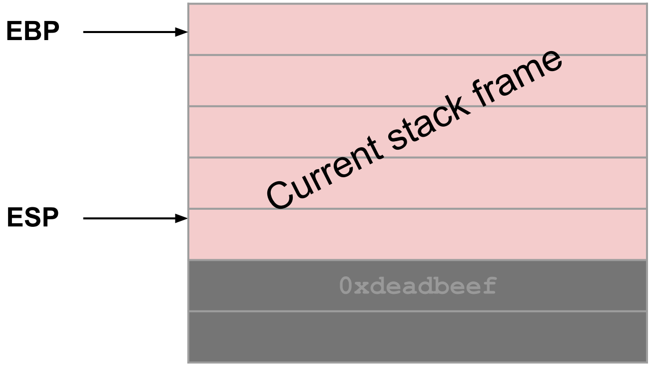
Pushing and Popping

- The **pop** instruction removes an element from the stack
  - Load the value from the lowest value 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 = `0xdeadbeef`
EBX = ...

After `pop %eax`

x86 Stack Layout

- 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 are stored with the first variable at the *lowest* address