Week 5: Lecture A
All About Applications

Tuesday, September 17, 2024

Announcements

- **Project 1: Crypto** released (see **Assignments** page on course website)
  - **Deadline: this Thursday**, September 19th by 11:59 PM

Project 1: Cryptography

Deadline: Thursday, September 19 by 11:59PM.

Before you start, review the course syllabus for the Lateness, Collaboration, and Ethical Use policies.

You may optionally work alone, or in teams of **at most two** and submit **one project per team**. If you have difficulties forming a team, post on **Piazza's Search for Teammates** forum. Note that the final exam will cover project material, so you and your partner should collaborate on each part.

The code and other answers your group submits must be entirely your own work, and you are bound by the University's Student Code. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone outside your group. You may consult published references, provided that you appropriately cite them (e.g., in your code comments). **Don't risk your grade and degree by cheating!**

Complete your work in the **CS 4440 VM**—we will use this same environment for grading. You may not use any **external dependencies**. Use only default Python 3 libraries and/or modules we provide you.

Helpful Resources

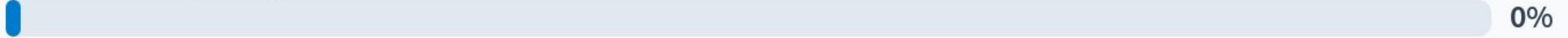
- The CS 4440 Course Wiki
- VM Setup and Troubleshooting
- Terminal Cheat Sheet
- Python 3 Cheat Sheet
- PyMD5 Module Documentation
- PyRoots Module Documentation

Table of Contents:

- Helpful Resources
- Introduction
- Objectives
- Start by reading this!
  - Working in the VM
  - Testing your Solutions
- Part 1: Hash Collisions
  - Prelude: Collisions
  - Prelude: FastColl
  - Collision Attack
  - What to Submit
- Part 2: Length Extension
  - Prelude: Merkle-Damgår
  - Length Extension Attack
  - What to Submit
- Part 3: Cryptanalysis
  - Prelude: Ciphers
  - Cryptanalysis Attack
  - Extra Credit
  - What to Submit
- Part 4: Signature Forgery
  - Prelude: RSA Signatures
  - Prelude: Bleichenbacher
  - Forgery Attacks
  - What to Submit

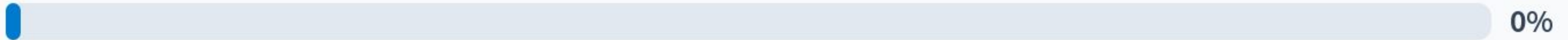
Progress on Project 1

Finished everything!



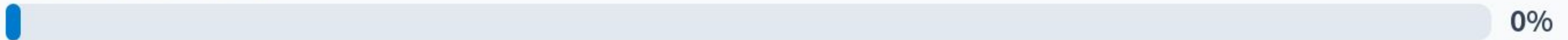
0%

Finished Parts 1 – 3



0%

Finished Parts 1 – 2

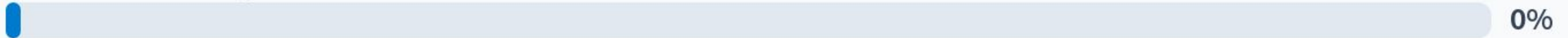


0%

Finished Part 1

0%

Haven't started :(



0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Announcements

- **Project 2: AppSec** released
  - **Deadline:** Thursday, October 17th by 11:59PM

Project 2: Application Security

Deadline: Thursday, October 17 by 11:59PM.

Before you start, review the course syllabus for the Lateness, Collaboration, and Ethical Use policies.

You may optionally work alone, or in teams of **at most two** and submit **one project per team**. If you have difficulties forming a team, post on **Piazza's Search for Teammates** forum. Note that the final exam will cover project material, so you and your partner should collaborate on each part.

The code and other answers your group submits must be entirely your own work, and you are bound by the University's Student Code. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone outside your group. You may consult published references, provided that you appropriately cite them (e.g., in your code comments). **Don't risk your grade and degree by cheating!**

Complete your work in the **CS 4440 VM**—we will use this same environment for grading. You may not use any **external dependencies**. Use only default Python 3 libraries and/or modules we provide you.

Helpful Resources

- The CS 4440 Course Wiki
- VM Setup and Troubleshooting
- Terminal Cheat Sheet
- GDB Cheat Sheet
- x86 Cheat Sheet
- C Cheat Sheet

Table of Contents:

- Helpful Resources
- Introduction
- Objectives
- Start by reading this!
  - Setup Instructions
  - Important Guidelines
- Part 1: Beginner Exploits
  - Target 0: Variable Overwrite
  - Target 1: Execution Redirect
  - What to Submit
- Part 2: Intermediate Exploits
  - Target 2: Shellcode Redirect
  - Target 3: Indirect Overwrite
  - Target 4: Beyond Strings
  - What to Submit
- Part 3: Advanced Exploits
  - Target 5: Bypassing DEP
  - Target 6: Bypassing ASLR
  - What to Submit
- Part 4: Super L33T Pwnage
  - Extra Credit: Target 7
  - Extra Credit: Target 8
  - What to Submit
- Submission Instructions

Wiki Updates

CS 4440 Wiki: All Things CS 4440

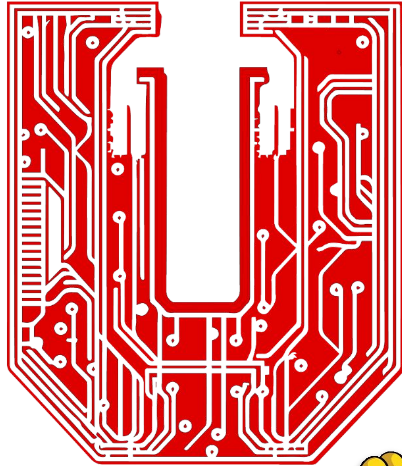
This Wiki is here to help you with all things CS 4440: from setting up your VM to introducing the languages and tools that you'll use. Check back here throughout the semester for future updates.

Have ideas for other pages? Let us know on Piazza!

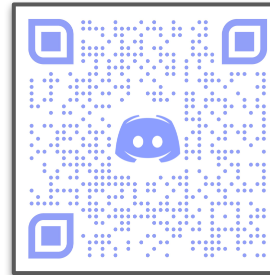
Tutorials and Cheat Sheets

Page	Description
VM Setup & Troubleshooting	Instructions for setting up your CS 4440 Virtual Machine (VM).
Terminal Cheat Sheet	Navigating the terminal, manipulating files, and other helpful tricks.
Python 3 Cheat Sheet	A gentle introduction to Python 3 programming.
x86 Assembly Cheat Sheet	Common x86 instructions and instruction procedures.
C Cheat Sheet	Information on C functions, and storing and reading data.
GDB Cheat Sheet	A quick reference for useful GNU Debugger (GDB) commands.
JavaScript Cheat Sheet	A gentle introduction to relevant JavaScript commands.

Announcements



utahsec



See Discord for meeting info!

utahsec.cs.utah.edu

Questions?

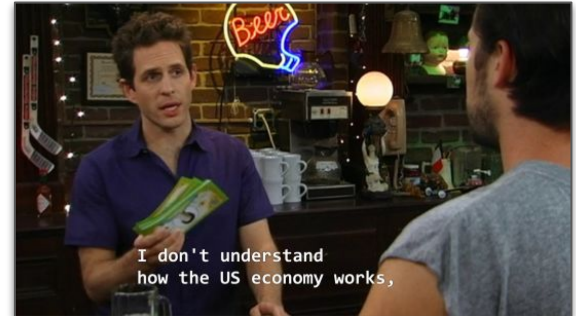
Last time on CS 4440...

Cryptocurrency
Distributed Consensus
Mining
Fairness

“The Gang Invents a New Currency”

- **Cryptocurrency**
  - Invented in 2008 (Bitcoin) by **Satoshi Nakamoto**
  - His/their real identify remains a mystery
  - **Modern cryptocoins**: Bitcoin, Litecoin, Ethereum

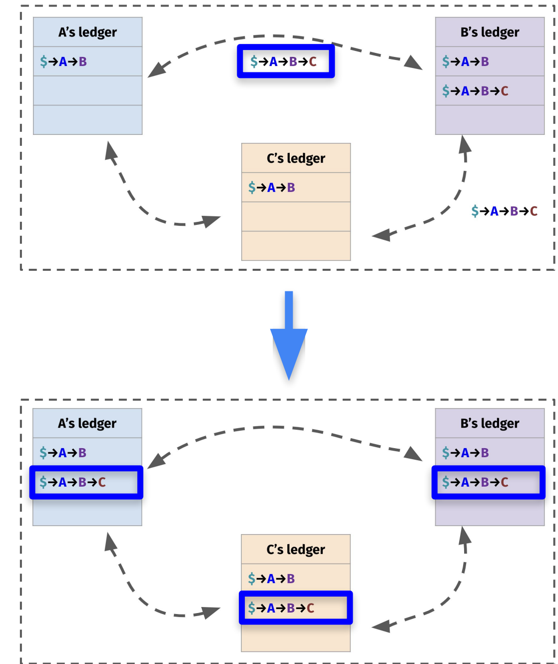
- **Key Principles**
  - Integrity
  - Distributed Consensus
  - Cryptographic Hash Function
  - Public-key Crypto
  - Proof-of-Work



Transactions

- Traditional banking uses a **“centralized” ledger**
  - You have as much **$$$** as your bank (and US Govt.) says!

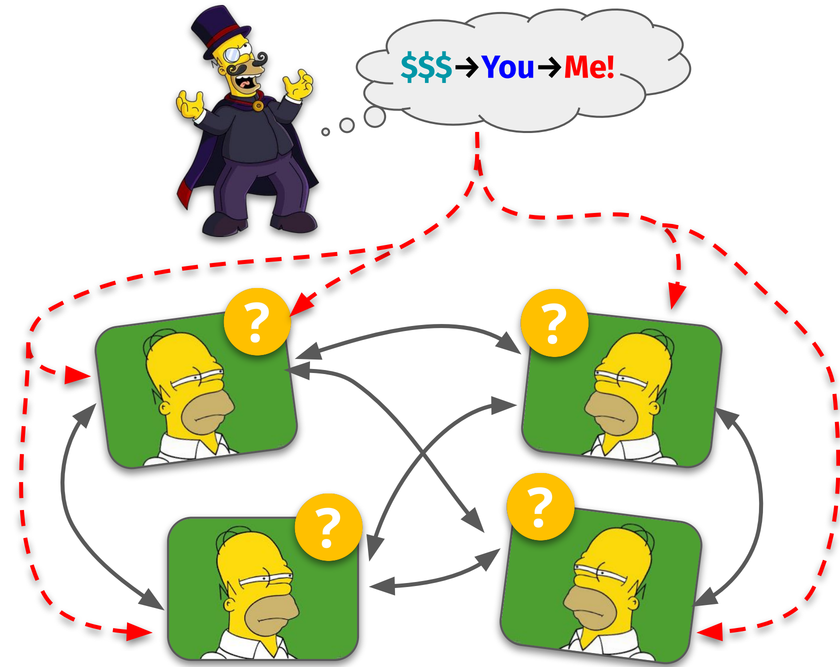
- Cryptocurrency = **Distributed Public Ledger**
  - Everyone has access to every transaction
  - Everyone knows how much money everyone else has
  - Transactions are chained using previous transactions
  - To determine how much money you have, must search the list of transactions to determine your balance
  - Trust that **< 50%** of the network is corrupt



Security

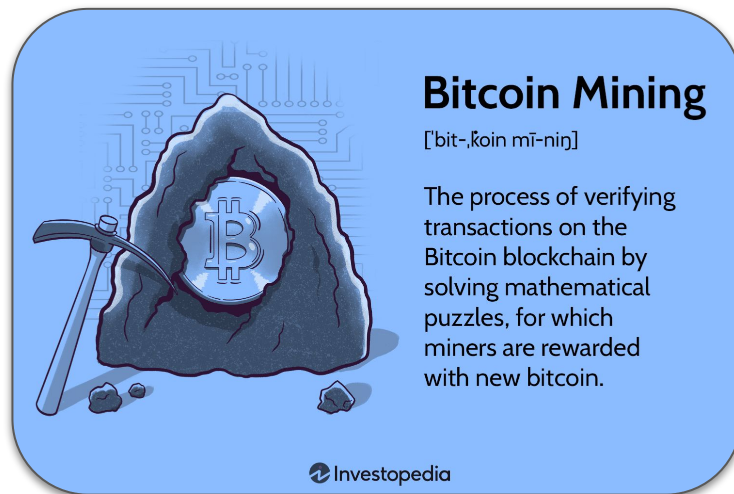
- Transactions must be **“committed”**
  - Resource intensive and competitive
  - Requires massive computing power to fool
  - Need to out-compute the entire network
  - Can't work “ahead” due to block chaining

- Security via **“distributed consensus”**
  - It’s hard to to fool *everyone* in the room
  - Specifically, have to fool **51%** of network
    - Majority vote wins
    - Longer ledger wins



"Mining" Cryptocurrency

- We want to print our own money!
- **Super high-level idea:** reward who first "validates" a transaction
  - Validators are called "miners"
  - Given a small commision

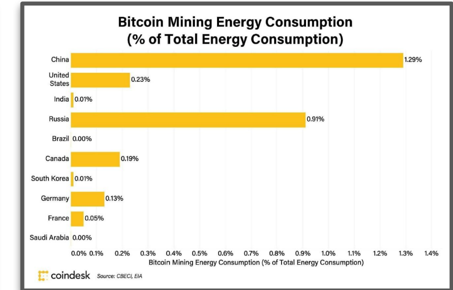


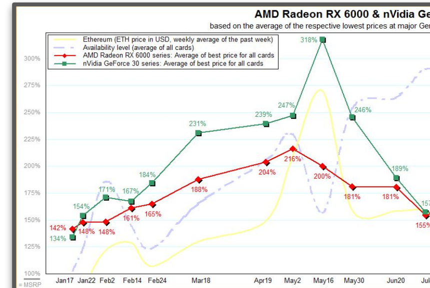
"Mining" Cryptocurrency

- We want to print our own money!
- **Super high-level idea:** reward who first "validates" a transaction
  - Validators are called "miners"
  - Given a small commision
- Ideally: a **fair process** (no entry fee)
  - Anyone can start mining!

“Mining” Cryptocurrency

- In practice, not really fair…
  - Hardware and GPU cost
  - Electricity cost
  - Environmental cost
  - More money gives an advantage!





"Mining" Cryptocurrency

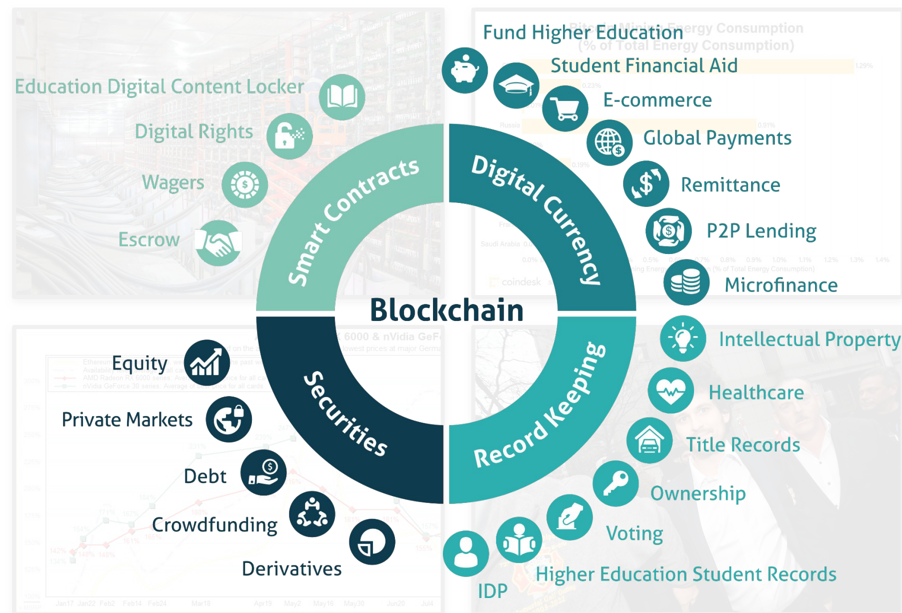
- In practice, not really fair...
  - Hardware and GPU cost
  - Electricity cost
  - Environmental cost
  - More money gives an advantage!

- **Don't buy into the hype!**

“Mining” Cryptocurrency

- In practice, not really fair…
  - Hardware and GPU cost
  - Electricity cost
  - Environmental cost
  - More money gives an advantage!

- **Don’t buy into the hype!**
  - Blockchain has **other cool uses**



Questions?

This time on CS 4440...

Program Execution
Virtual Memory
The Stack
Stack Corruption

Coding Challenge

- As part of a job interview, you are tasked with writing a program—in **C**—that:

  (1) reads characters from the user; and
  (2) prints out the reverse of that message.

- You are expected to write a working program in less than 5 minutes. **Go!**

Coding Challenge

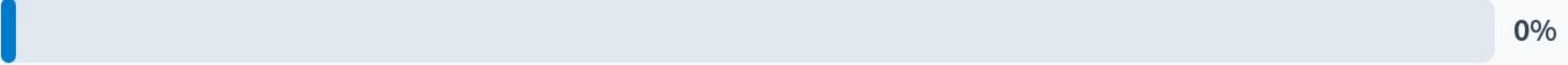
- If you wrote a program like:

```
int main(void) {
    char buffer[40];
    gets(buffer);
    // Saves user input
    // into the buffer
}
```

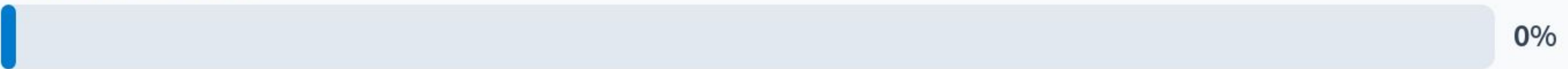
This program will...

ALWAYS run normally!

NEVER run normally!



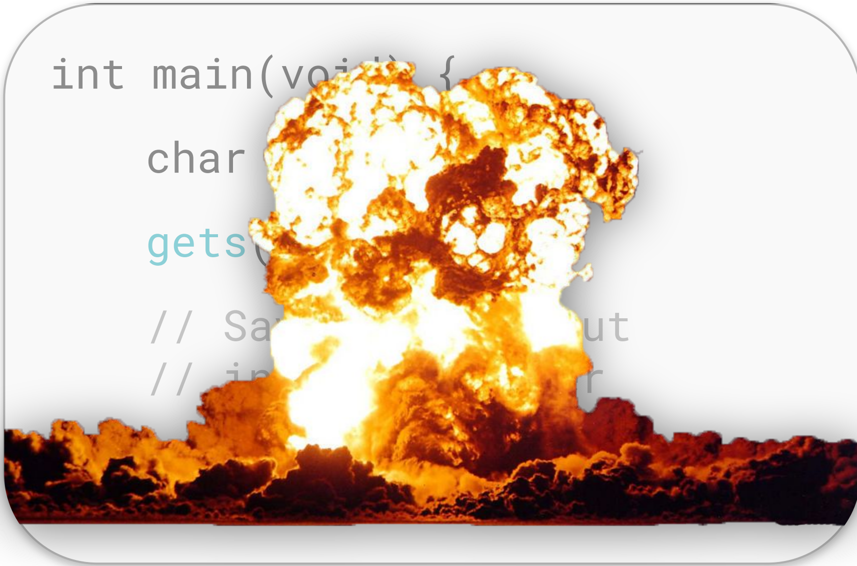
None of the above!

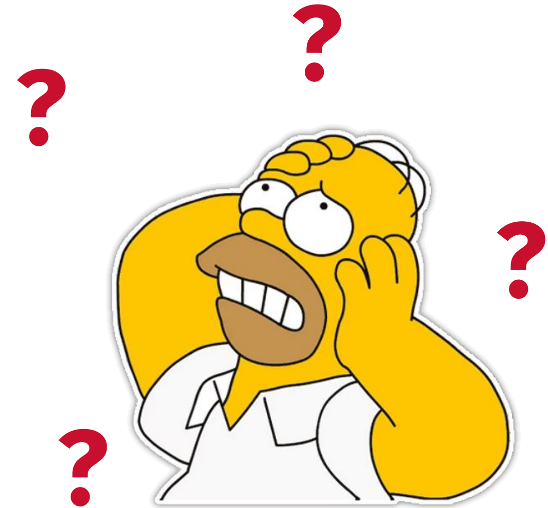


Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Coding Challenge

- If you wrote a program like:





Coding Challenge

- If you wrote a program like:

CWE-242: Use of Inherently Dangerous Function

Weakness ID: 242
Abstraction: Base
Structure: Simple

View customized information: Conceptual | Operational | Mapping-Friendly | **Complete**

Description

The product calls a function that can never be guaranteed to work safely.

Extended Description

Certain functions behave in dangerous ways regardless of how they are used. Functions in this category were often implemented without taking security concerns into account. The gets() function is unsafe because it does not perform bounds checking on the size of its input. An attacker can easily send arbitrarily-sized input to gets() and overflow the destination buffer. Similarly, the >> operator is unsafe to use when reading into a statically-allocated character array because it does not perform bounds checking on the size of its input. An attacker can easily send arbitrarily-sized input to the >> operator and overflow the destination buffer.

Attacking Computer Systems

- **Problem:** attacker can't load their own code on to the system

Attacking Computer Systems

- **Problem:** attacker can’t load their own code on to the system

- **Opportunity:** the attacker can interact with **existing programs**

Attacking Computer Systems

- **Problem:** attacker can't load their own code on to the system
- **Opportunity:** the attacker can interact with **existing programs**
- **Challenge:** make the system do **what you want**... using only the existing programs on the system that you can interact with

Software Exploitation

- **Goal:** take over a system by exploiting an application on it

Software Exploitation

- **Goal:** take over a system by exploiting an application on it

- **Exploit technique 1: code injection**
  - Insert your own code (as an input)
  - Redirect the program to execute it

Software Exploitation

- **Goal:** take over a system by exploiting an application on it
- **Exploit technique 1: code injection**
  - Insert your own code (as an input)
  - Redirect the program to execute it
- **Exploit technique 2: code reuse**
  - Leverage the program's existing code
  - Execute it in a way it wasn't intended to

Software Exploitation

- **Goal:** take over a system by exploiting an application on it
- **Exploit technique 1: code injection**
  - Insert your own code (as an input)
  - Redirect the program to execute it
- **Exploit technique 2: code reuse**
  - Leverage the program's existing code
  - Execute it in a way it wasn't intended to
- **Attack vector: memory corruption**

Program Execution

What is execution?

- **Double-clicking a shortcut** on your desktop

What is execution?

- **Double-clicking a shortcut** on your desktop

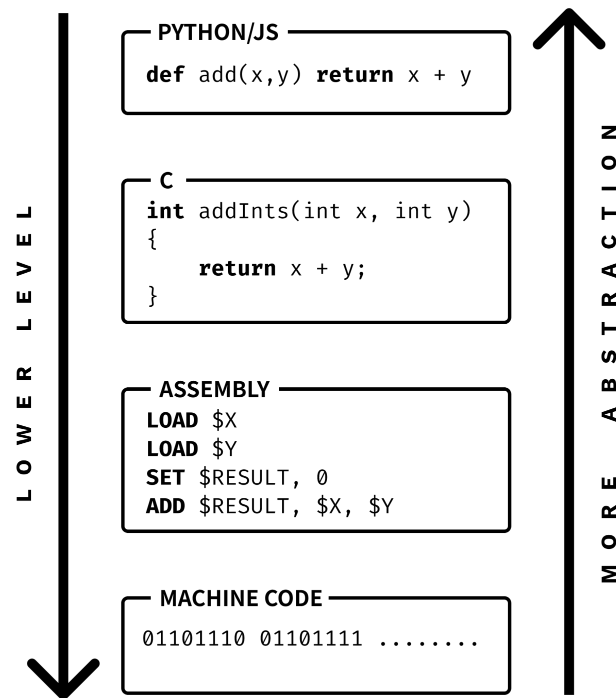
- Tapping **an app icon** on your smartphone

What is execution?

- **Double-clicking a shortcut** on your desktop
- Tapping **an app icon** on your smartphone
- **“Hey Siri, play Midnights on Spotify”**

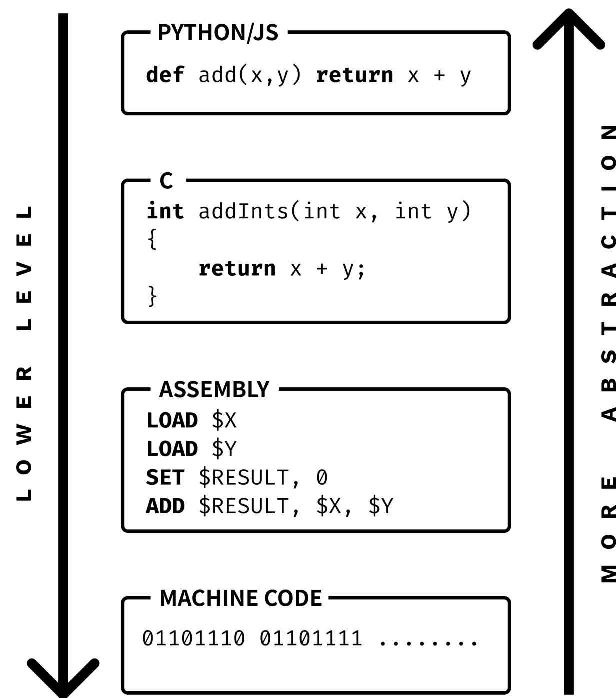
What *really* is execution?

- Programs made up of **instructions**



What *really* is execution?

- Programs made up of **instructions**
- **High-level:** programming languages
  - **Higher level:** interpreted (Python, JS, etc.)
  - **Lower level:** compiled (C/C++, Rust, Go)

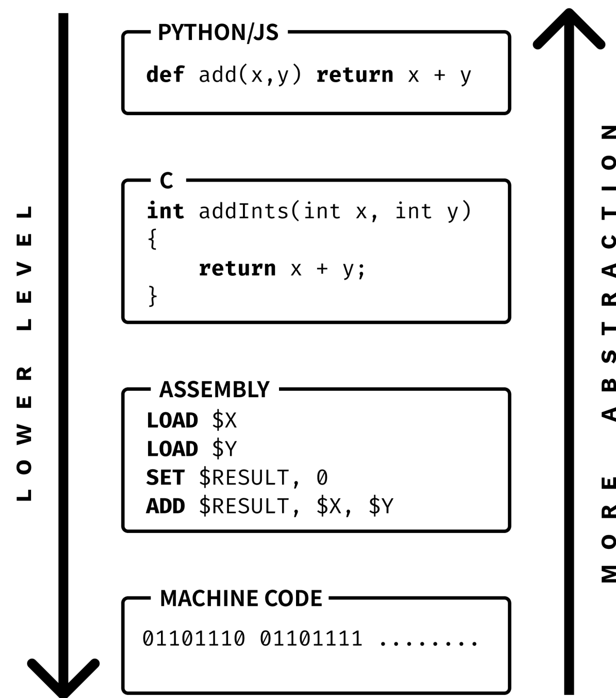


What *really* is execution?

- Programs made up of **instructions**

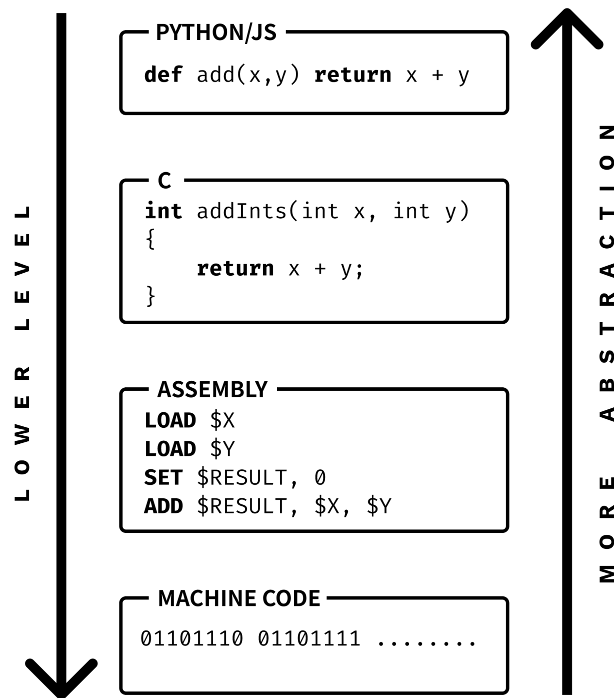
- **High-level:** programming languages
  - **Higher level:** interpreted (Python, JS, etc.)
  - **Lower level:** compiled (C/C++, Rust, Go)

- **Low-level:** assembly and machine code
  - **Machine code =** what the computer executes
  - **Assembly =** one level higher (human-readable)



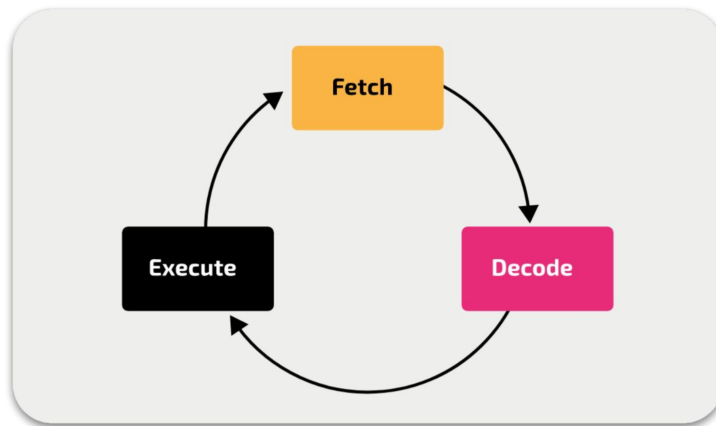
What *really* is execution?

- Programs made up of **instructions**
- **High-level:** programming languages
  - **Higher level:** interpreted (Python, JS, etc.)
  - **Lower level:** compiled (C/C++, Rust, Go)
- **Low-level:** assembly and machine code
  - **Machine code =** what the computer executes
  - **Assembly =** one level higher (human-readable)
- **Execution** = **executing instructions**



What *really* is execution?

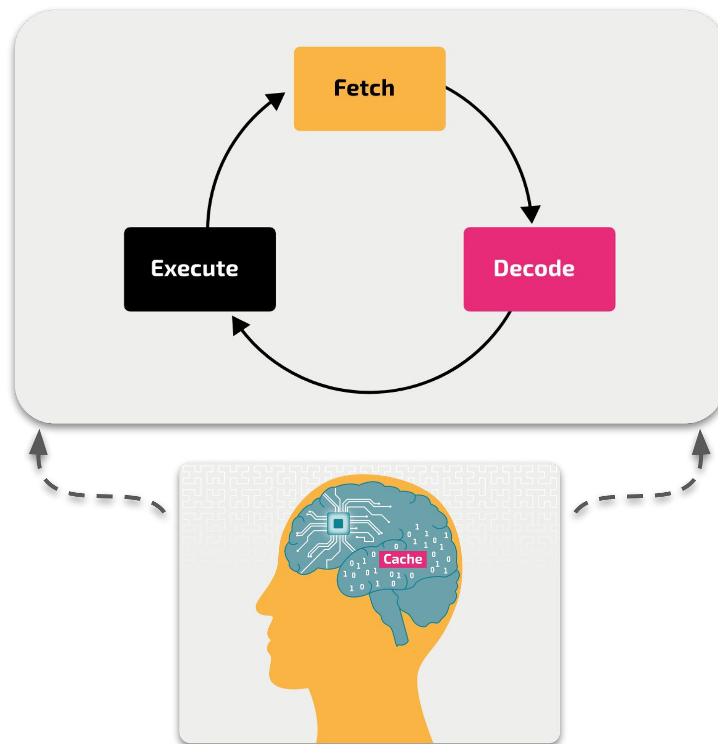
- Execution comprised of three steps
  - **Fetch** an instruction from the program
  - **Decode** the instruction into what it does
  - **Execute** that instruction



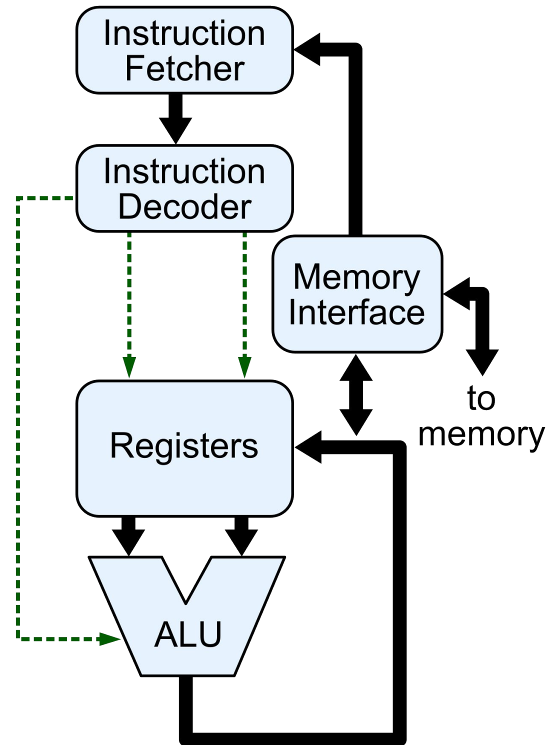
What *really* is execution?

- Execution comprised of three steps
  - **Fetch** an instruction from the program
  - **Decode** the instruction into what it does
  - **Execute** that instruction

- Execution is the job of the **CPU**
  - Central Processing Unit
  - The brain of your computer

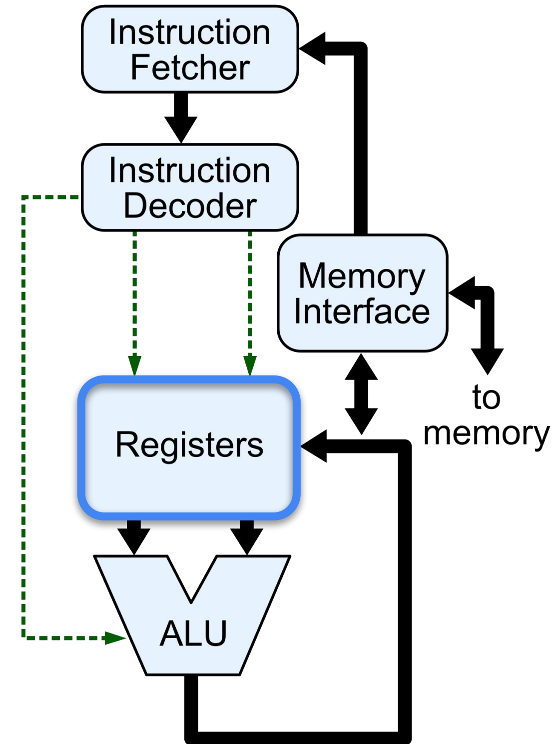


The CPU



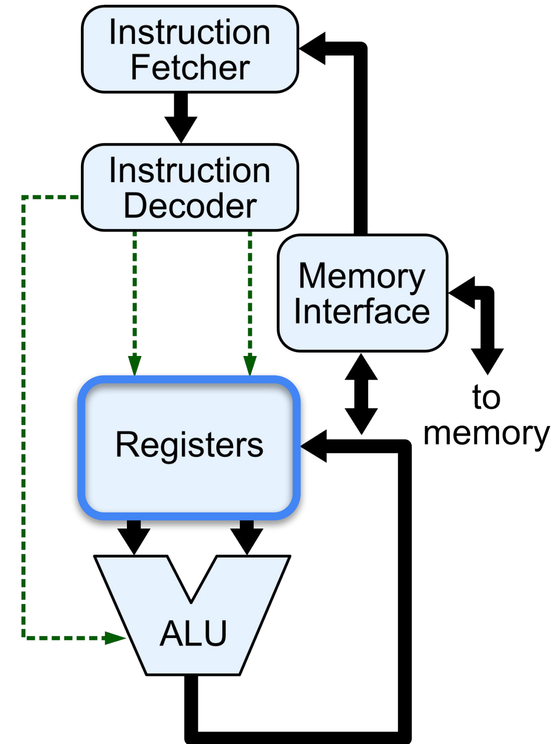
The CPU

- CPU state held in **registers**
  - Analogous to source code variables



The CPU

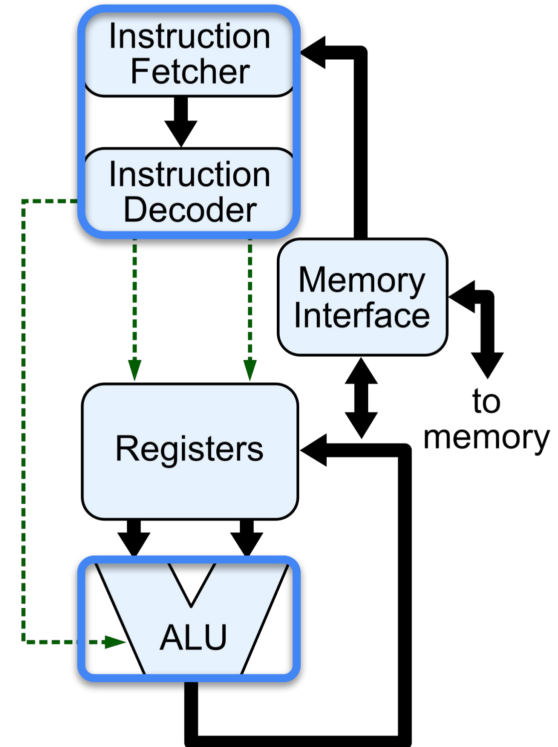
- CPU state held in **registers**
  - Analogous to source code variables
- **General-purpose** registers:
  - **EAX**, **EBX**, **ECX**, **EDX**, **EDI**, **ESI**
- **Special-purpose** registers:
  - **EIP** = Instruction Pointer
  - **ESP** = Stack Pointer
  - **EBP** = Frame/Base Pointer



The CPU

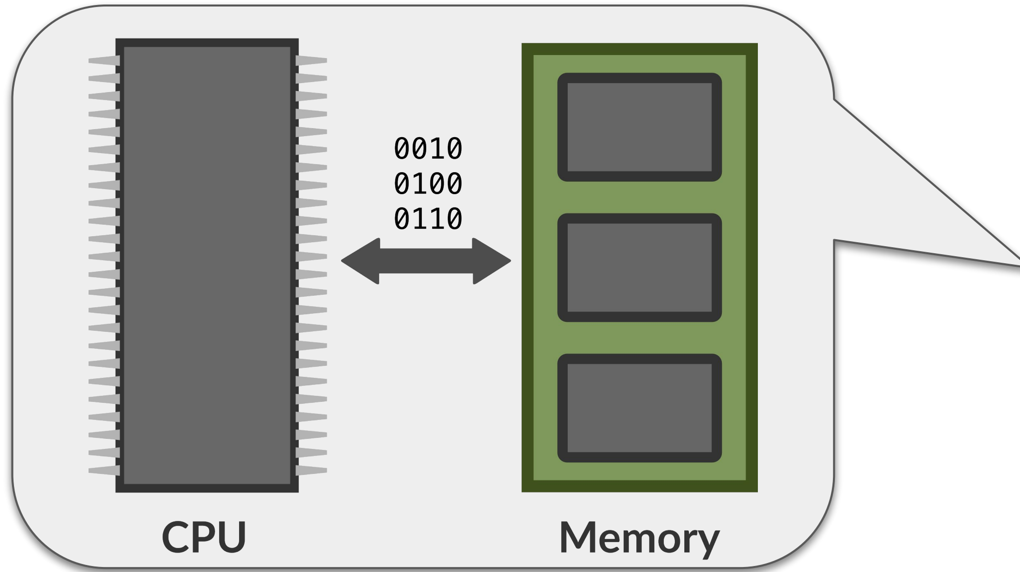
- State modified by assembly **instructions**
  - ADD, SUB, XOR, CMP, CALL, JMP, RET
  - **And many more!**

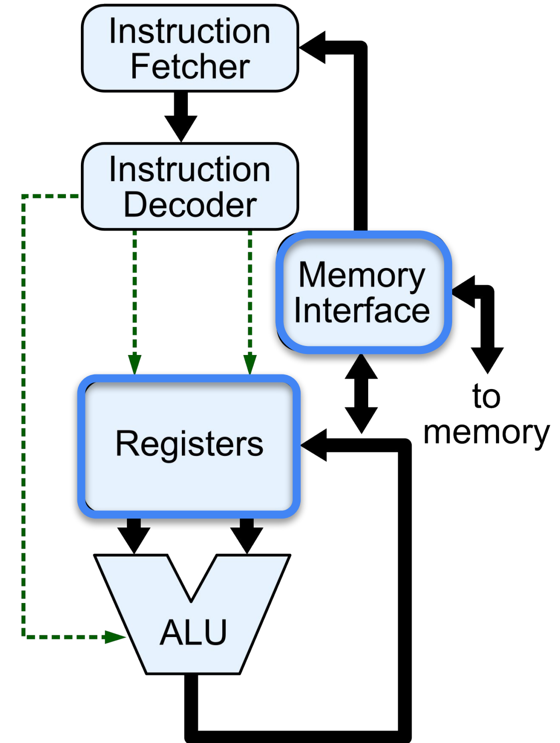
- Assembly instruction syntaxes
  - **AT&T** = `Instruction Source Destination`
  - **Intel** = `Instruction Destination Source`
  - Example: **MOV** SRC, DST versus **MOV** DST, SRC
  - This lecture: **AT&T syntax**



The CPU

- **Software state** = **registers** and **memory**

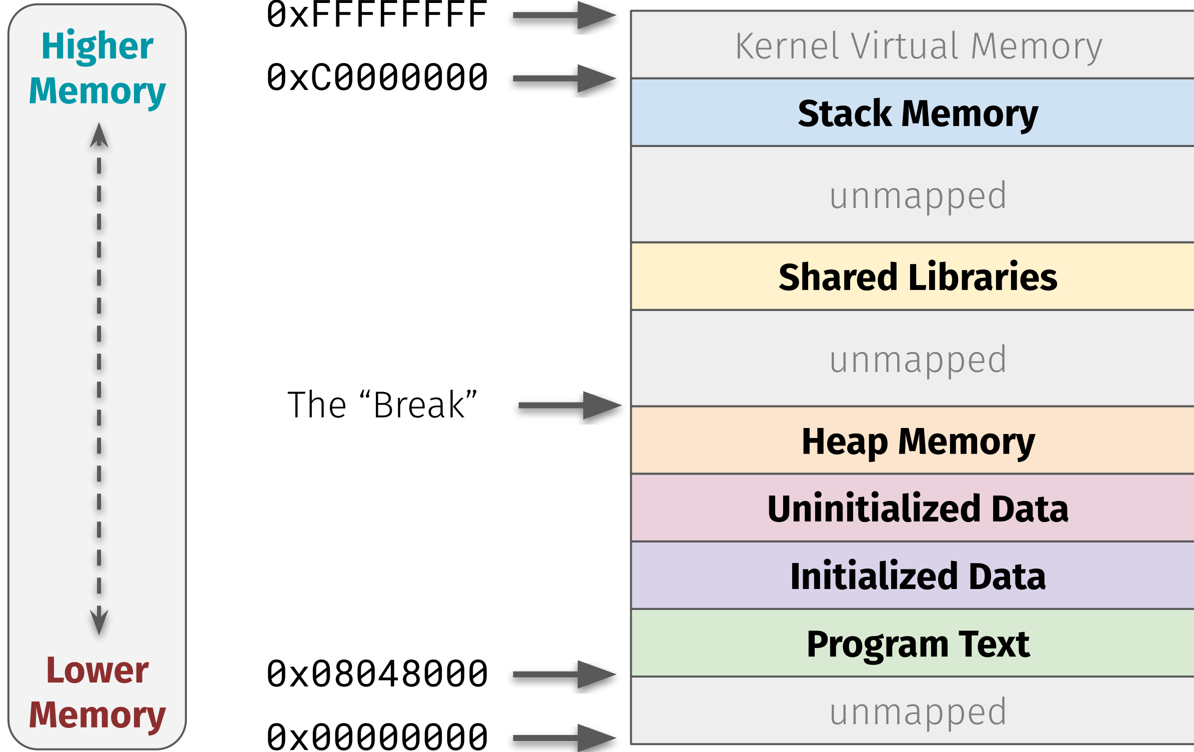




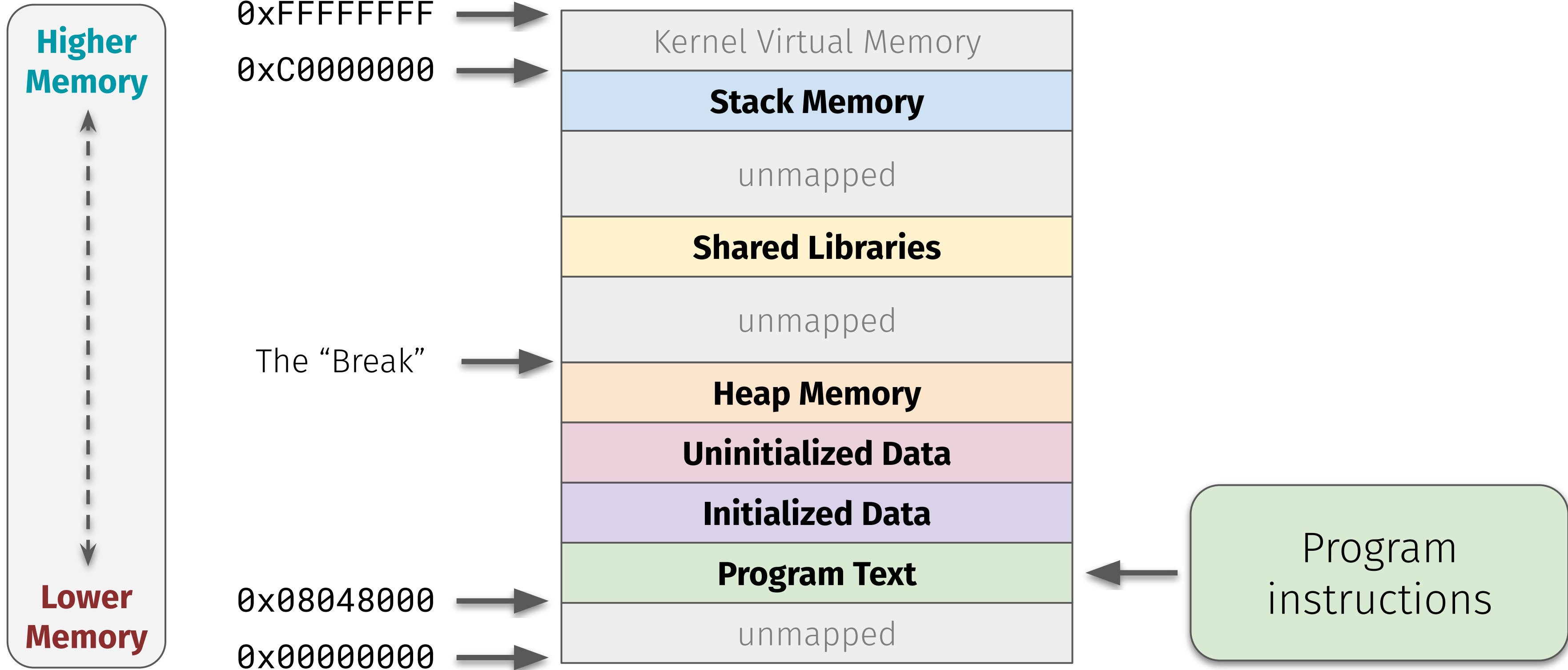
Questions?

Process Virtual Memory

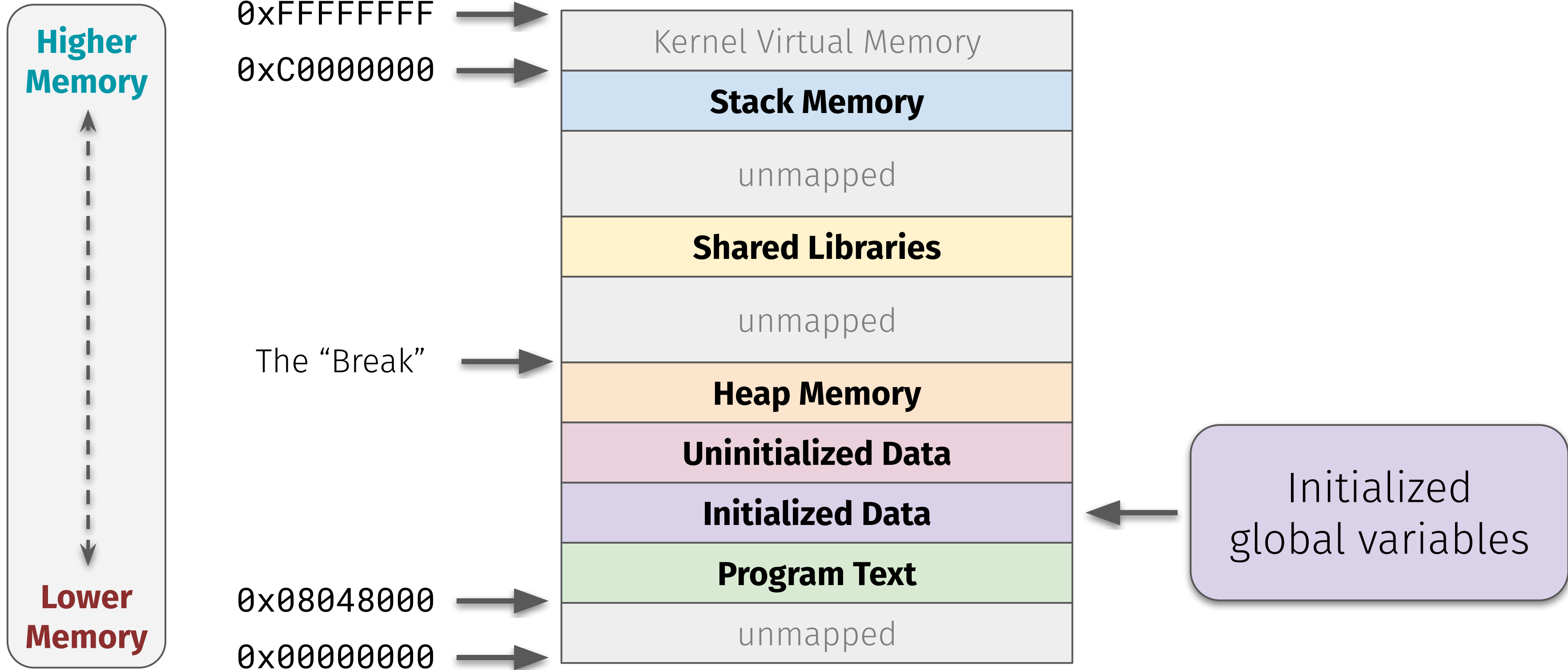
Memory layout of a 32-bit Linux process



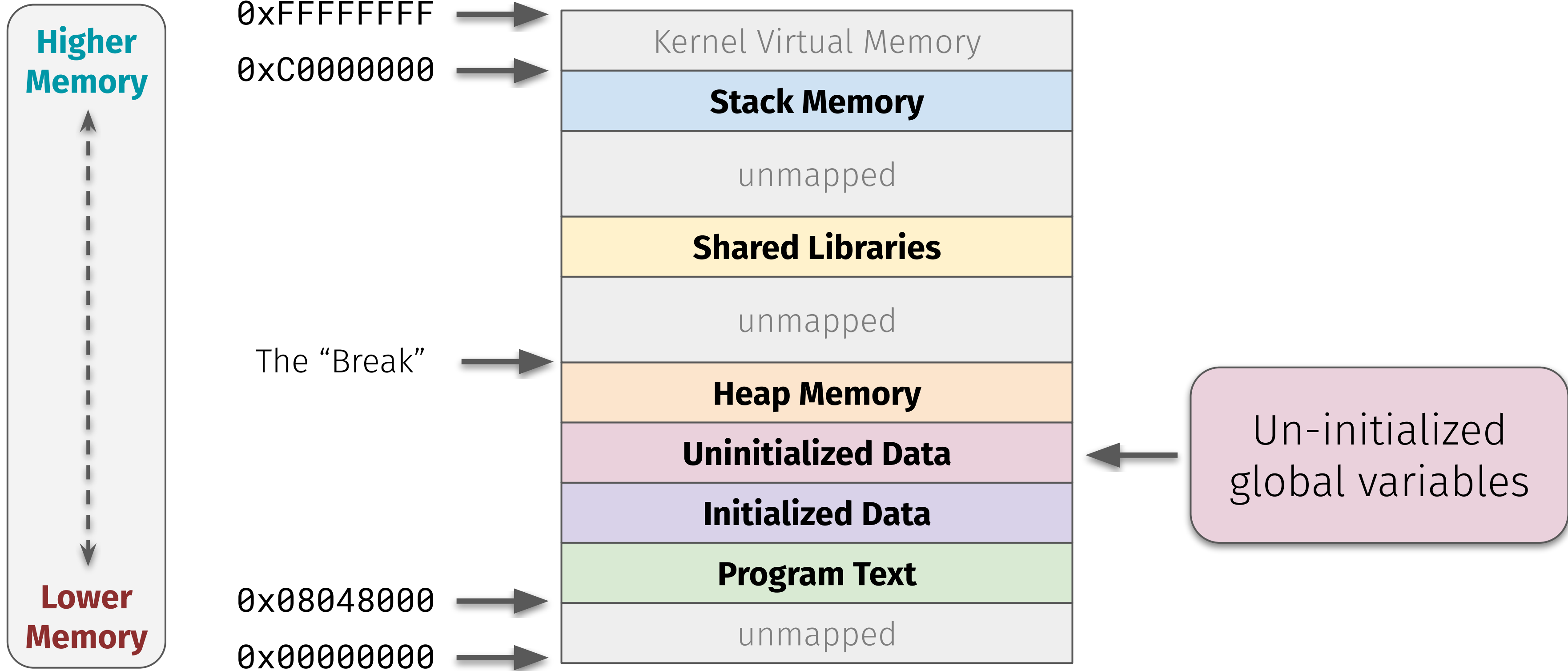
Memory layout of a 32-bit Linux process



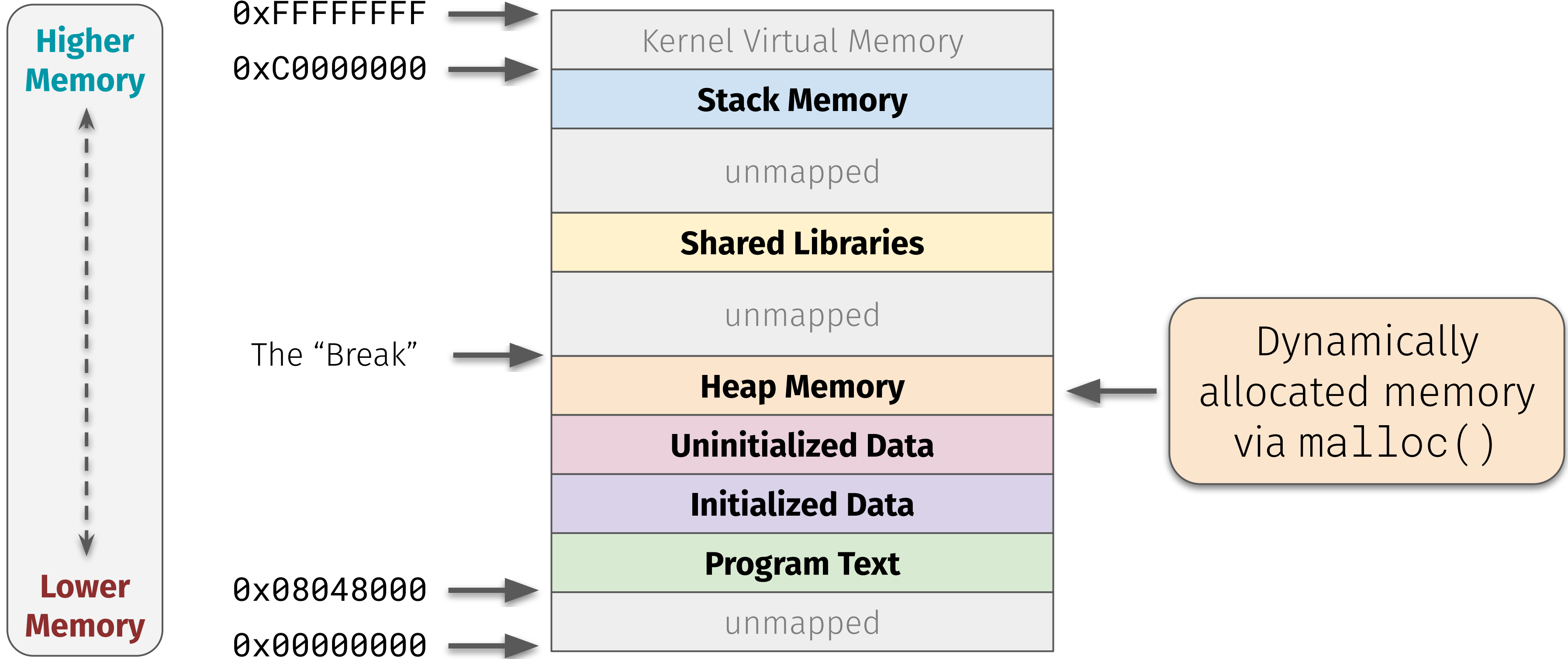
Memory layout of a 32-bit Linux process



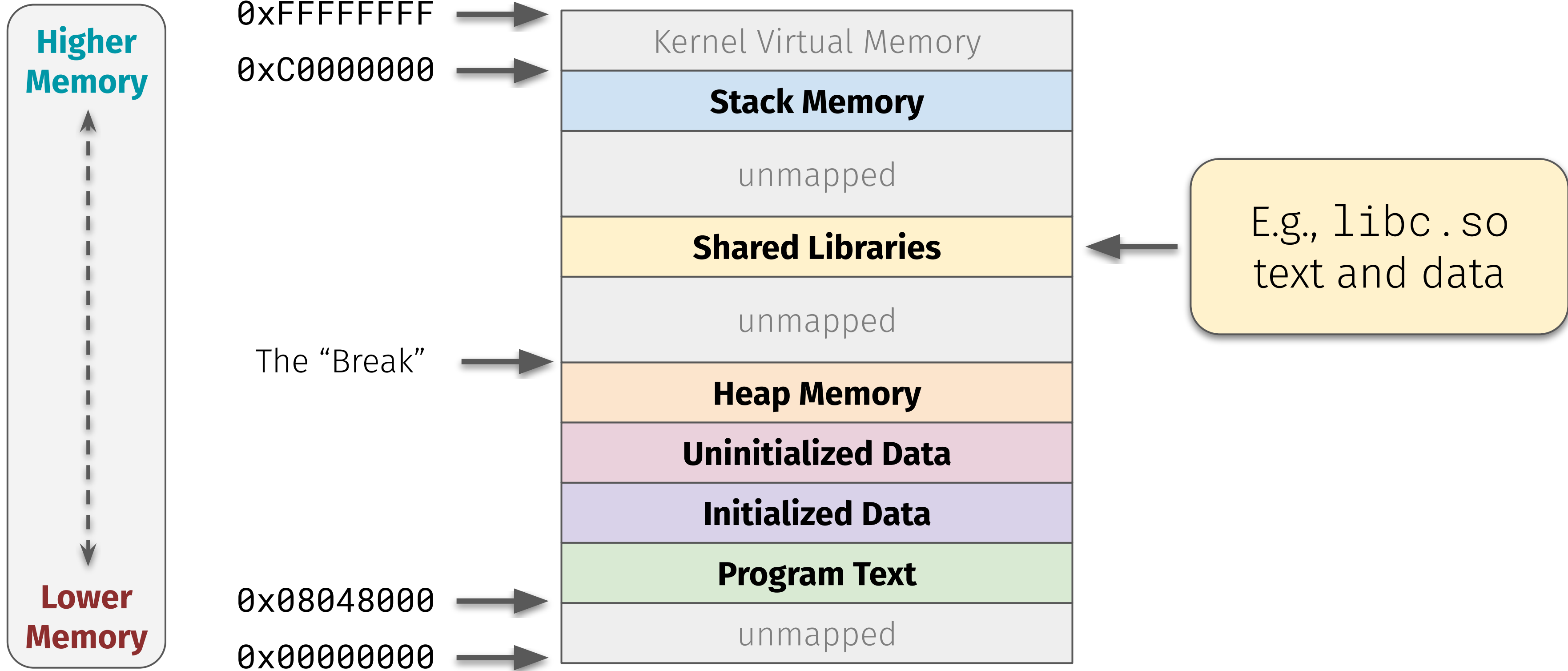
Memory layout of a 32-bit Linux process



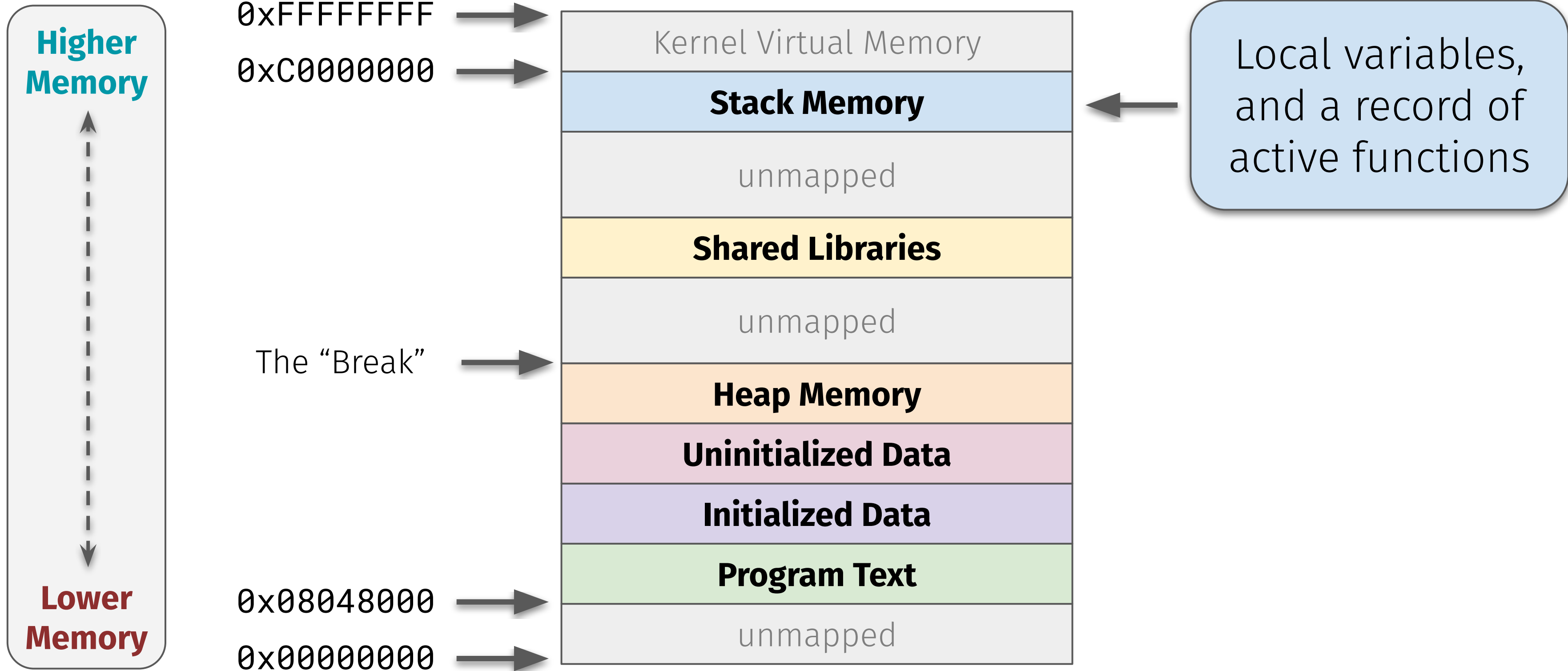
Memory layout of a 32-bit Linux process



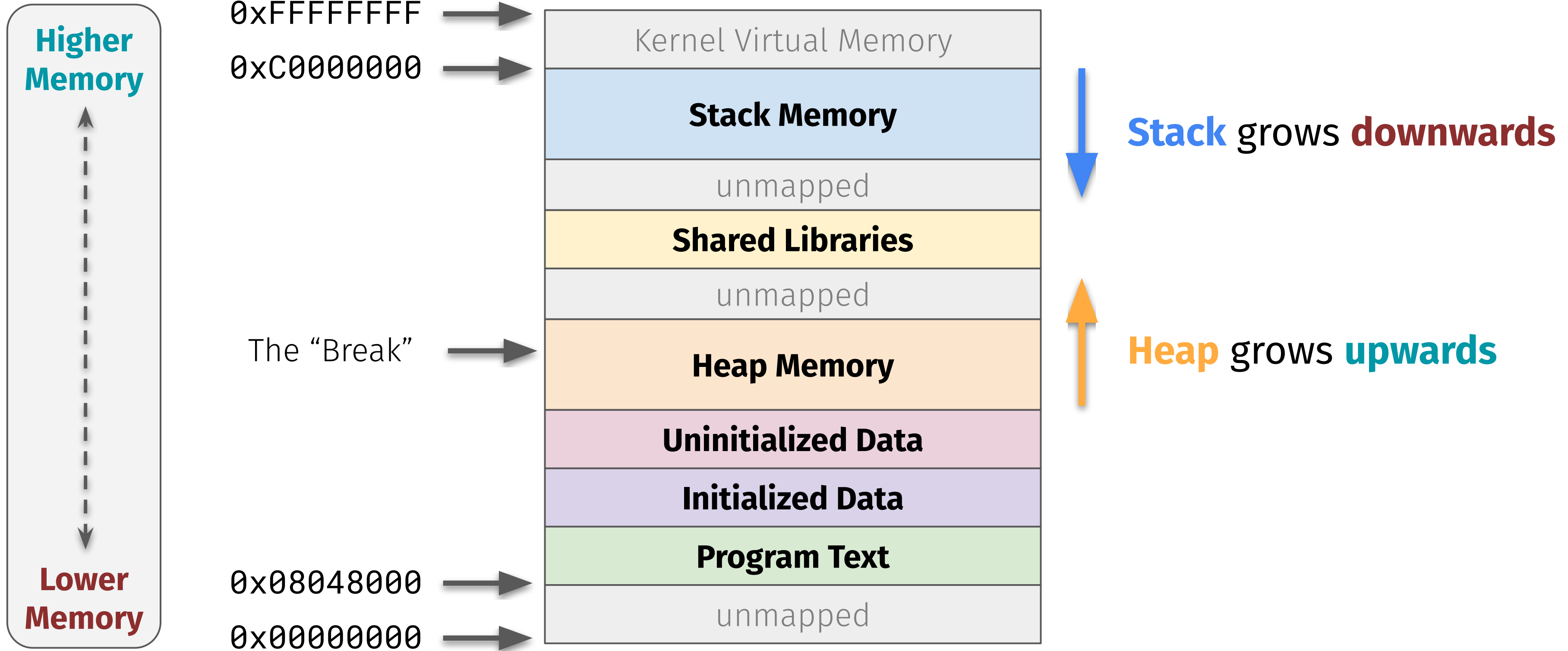
Memory layout of a 32-bit Linux process



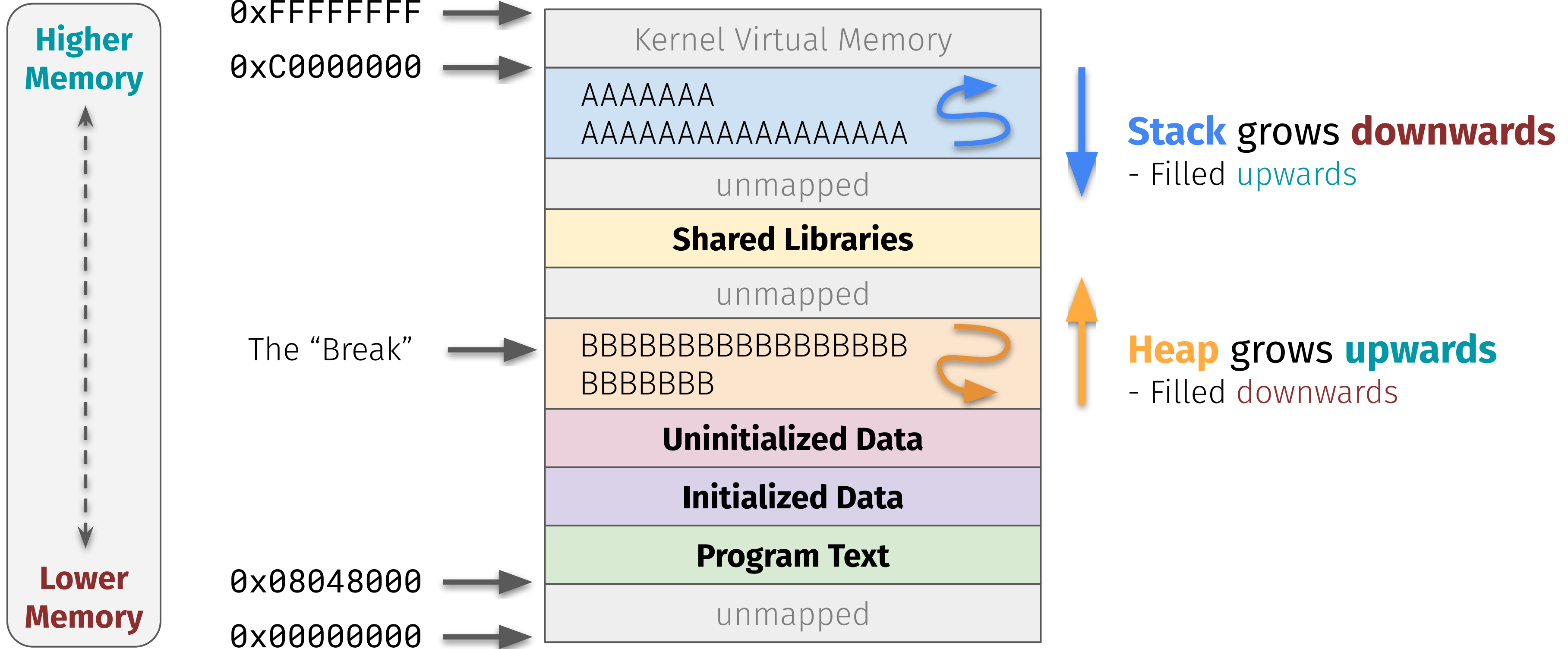
Memory layout of a 32-bit Linux process



Memory layout of a 32-bit Linux process



Memory layout of a 32-bit Linux process

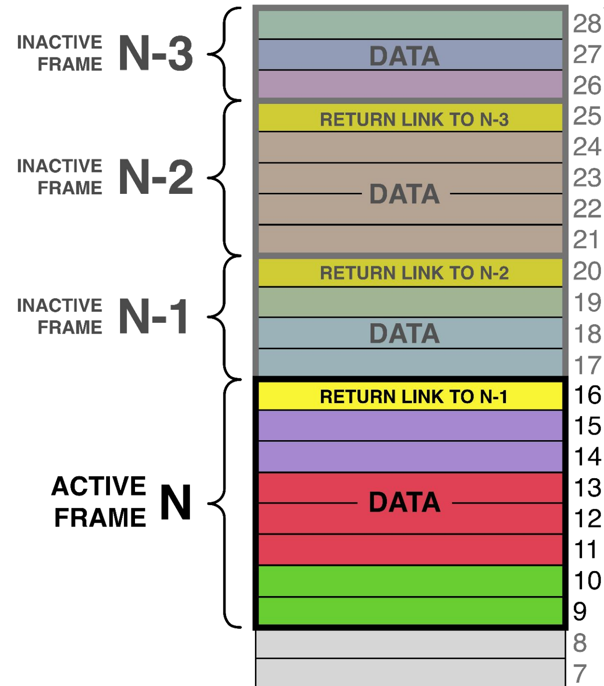


Questions?

The Stack

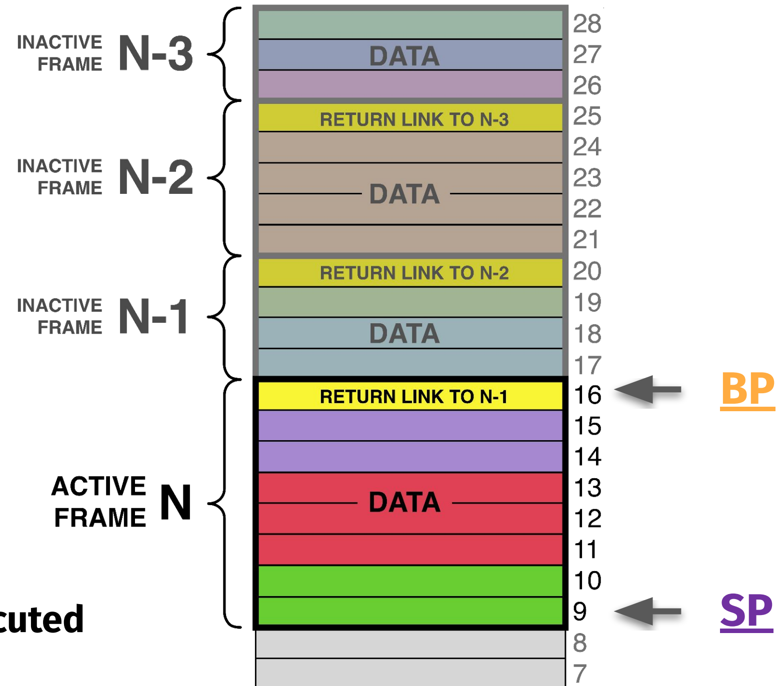
The Stack

- Memory for **storing function data**
  - Arguments
  - Local variables
  - Return address
- Provides a running “record” of the **active subroutine(s)** in a program



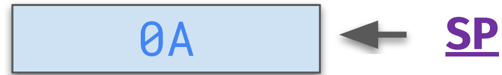
The Stack

- Begins at **highest address**
- Grows toward **lower addresses**
  - Think of it as a stack of plates that grows upside-down
- Three key registers to know:
  - EBP = The **Frame/Base** Pointer
    - **Highest address** of current frame
  - ESP = The **Stack** Pointer
    - Denotes the top of the stack
    - **Topmost (lowest) address** of the stack
  - EIP = Address of **next instruction to be executed**



Stack Operation

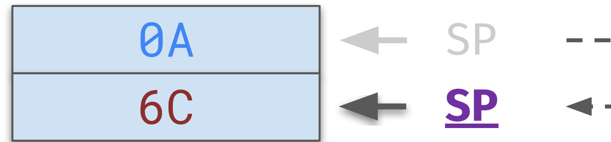
1. **Push `0x0A`**



Push sends data to the **topmost** area of the stack

Stack Operation

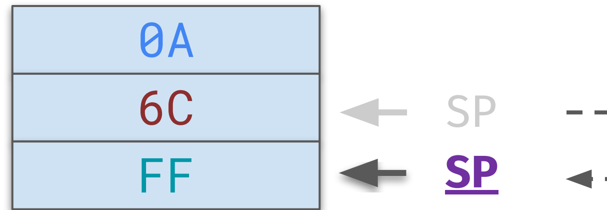
1. Push 0x0A
2. **Push 0x6C**



Stack grows →
move SP down!

Stack Operation

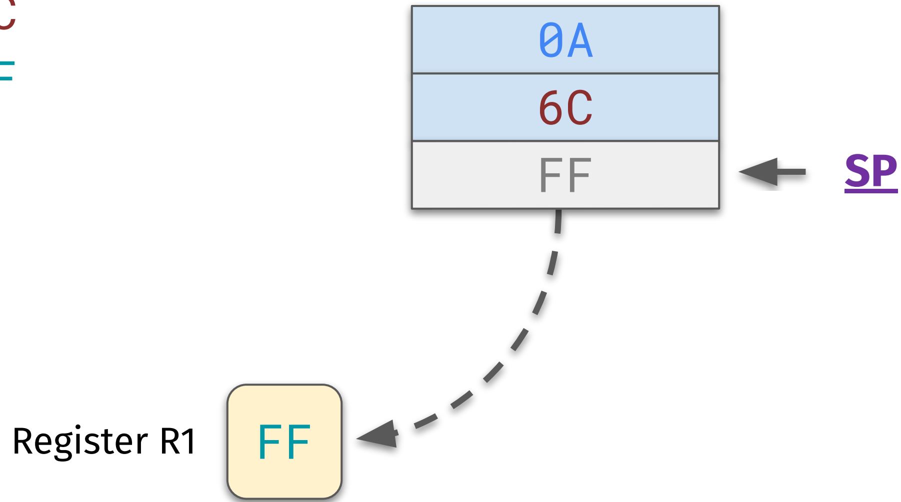
1. Push 0x0A
2. Push 0x6C
3. **Push 0xFF**



Stack grows →
move SP down!

Stack Operation

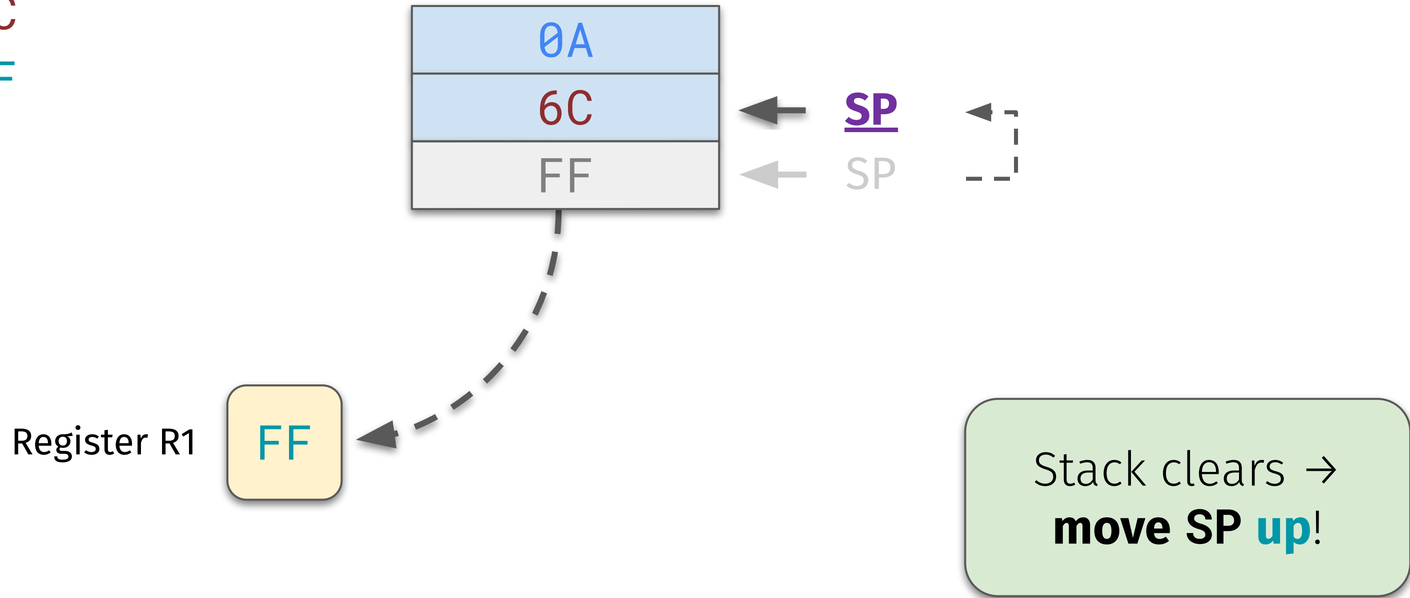
1. Push 0x0A
2. Push 0x6C
3. Push 0xFF
4. **Pop R1**



Pop sends data at top of stack to a **register**

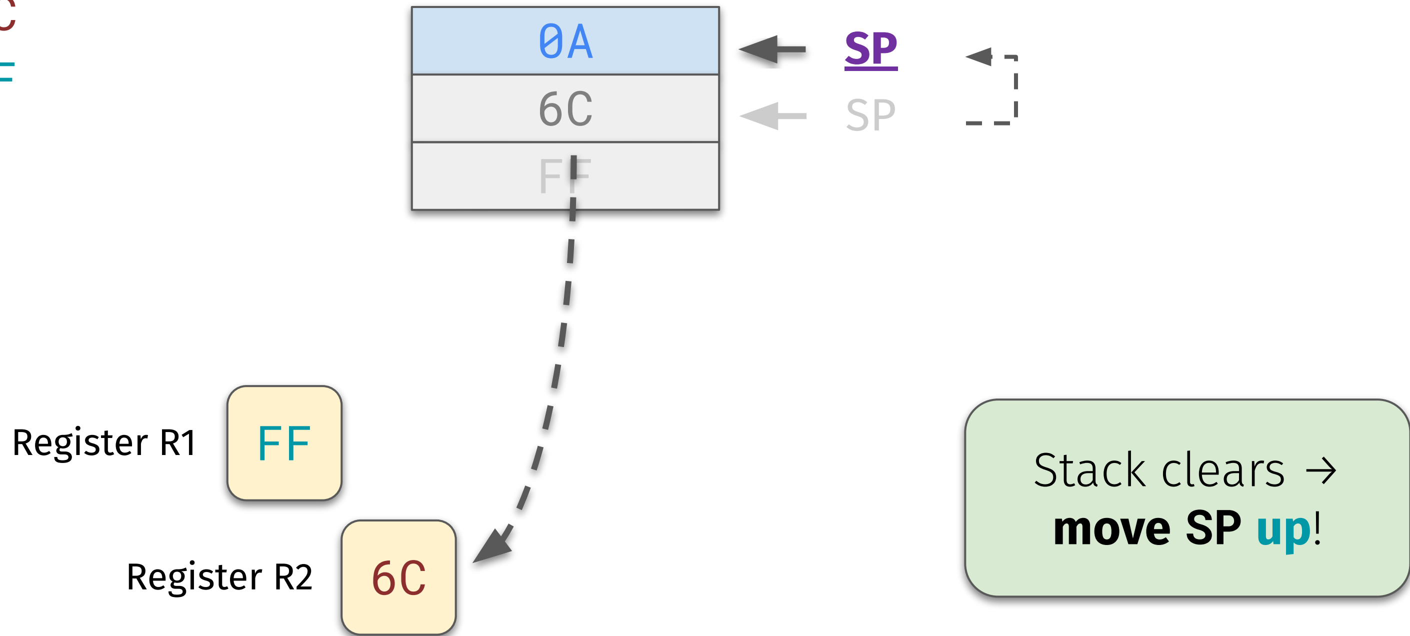
Stack Operation

1. Push 0x0A
2. Push 0x6C
3. Push 0xFF
4. **Pop R1**



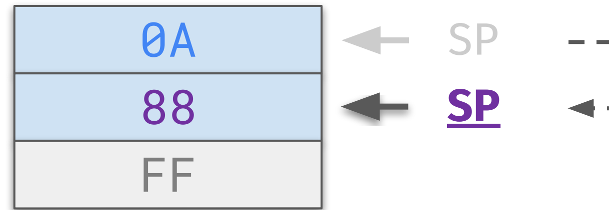
Stack Operation

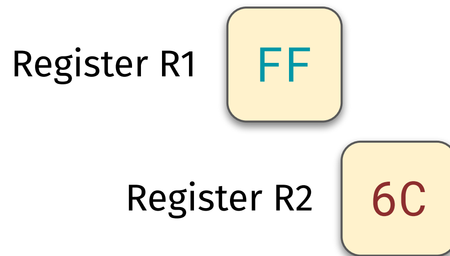
1. Push 0x0A
2. Push 0x6C
3. Push 0xFF
4. Pop R1
5. **Pop R2**



Stack Operation

1. Push 0x0A
2. Push 0x6C
3. Push 0xFF
4. Pop R1
5. Pop R2
6. **Push 0x88**

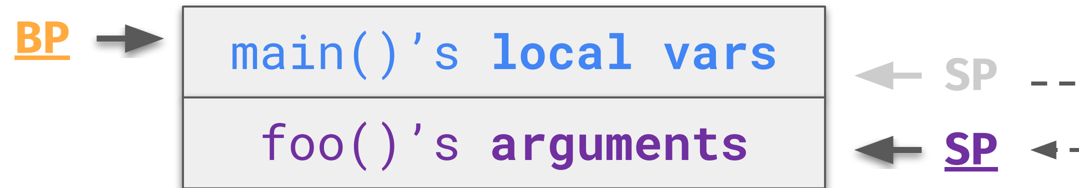




Stack grows →
move SP down!

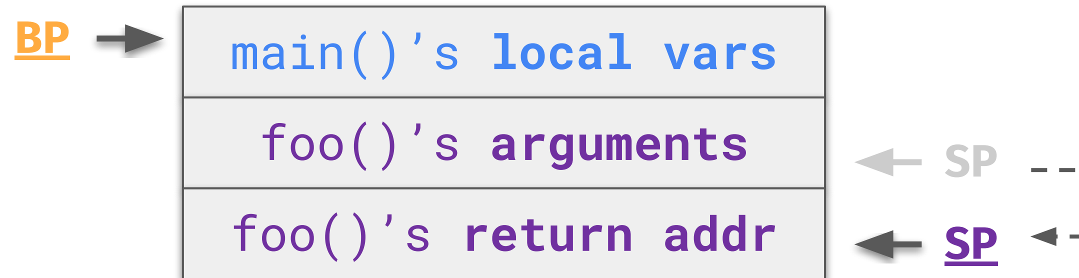
Stack Frames

- Assume `main()` calls `foo()`



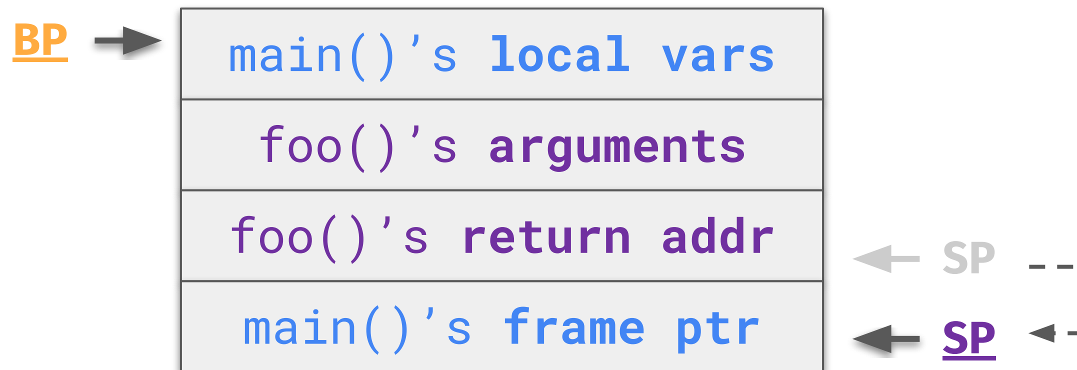
Stack Frames

- Assume `main()` calls `foo()`



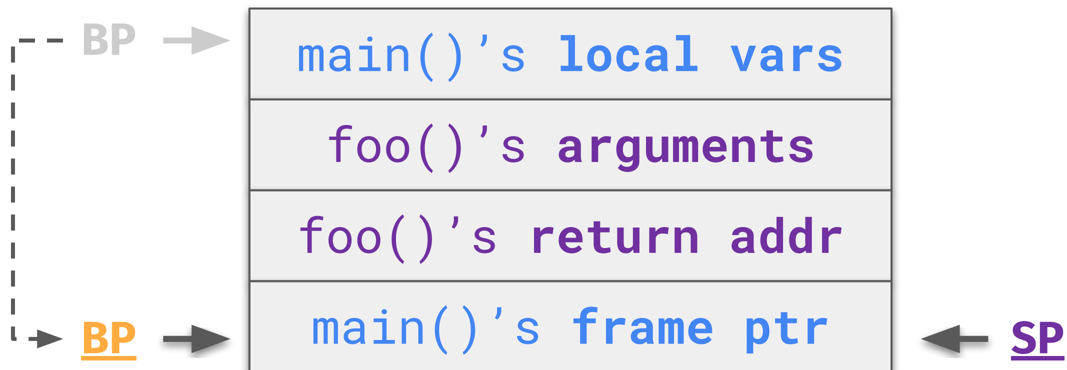
Stack Frames

- Assume `main()` calls `foo()`



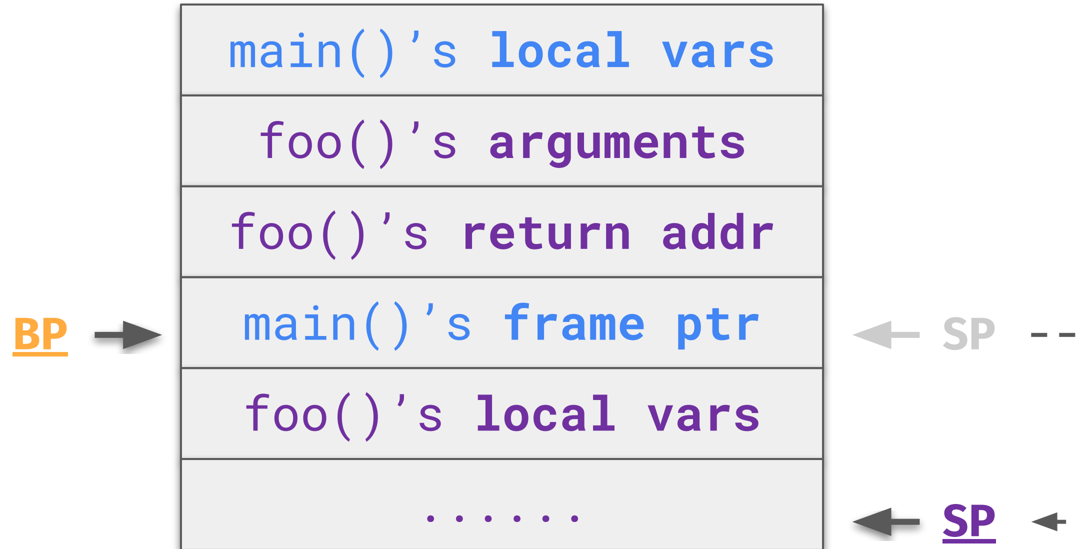
Stack Frames

- Assume `main()` calls `foo()`



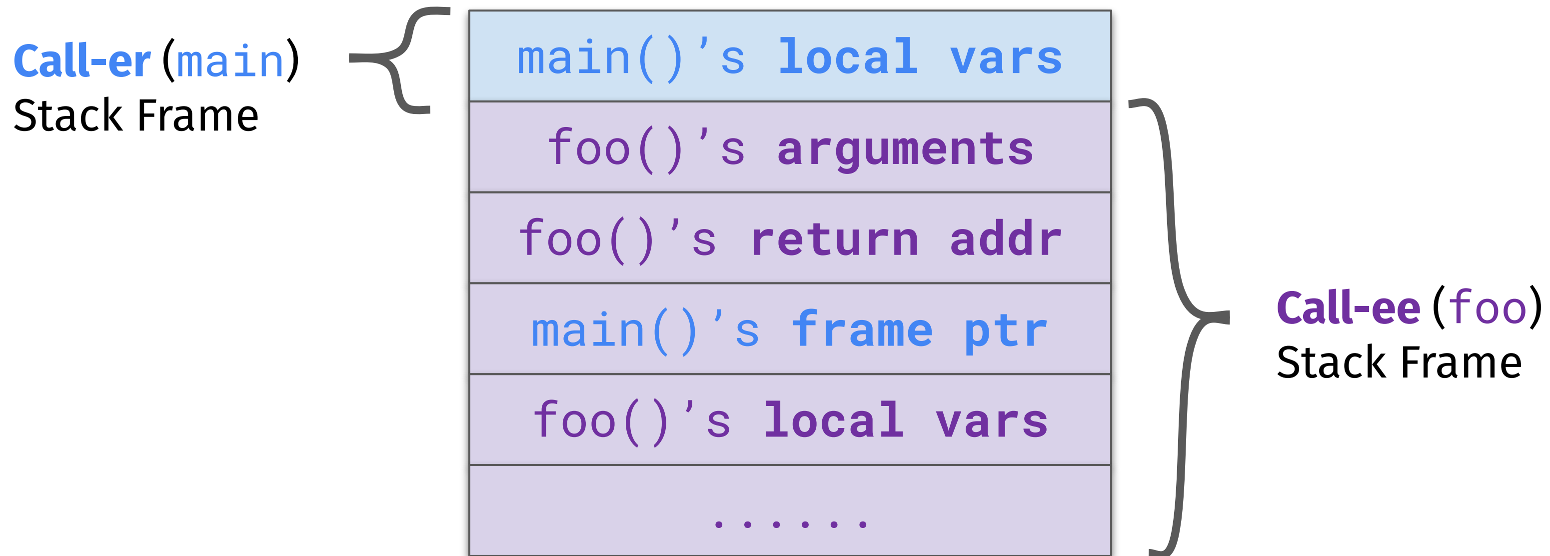
Stack Frames

- Assume `main()` calls `foo()`



Stack Frames

- Assume `main()` calls `foo()`



Example Program

```
void foo(int a, int b)
{
    char buf1[10];
}


void main()
{
    foo(3,6);
}
```

```
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```

Example Program

```
main:
    pushl  %ebp
    movl   %esp, %ebp
    subl   $8, %esp
    movl   $6, 4(%esp)
    movl   $3, (%esp)
    call   foo
    leave
    ret
```

Example Program

```
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```

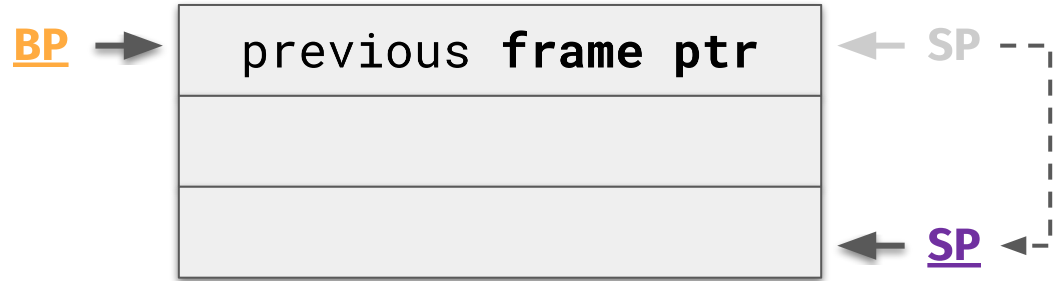
previous frame ptr ← SP

Example Program

```
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```

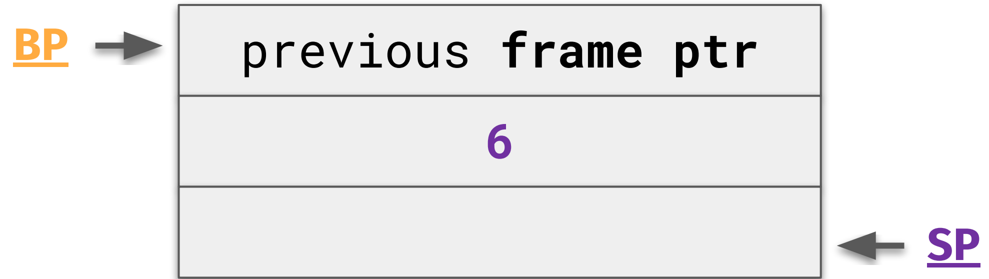
Example Program

```
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```



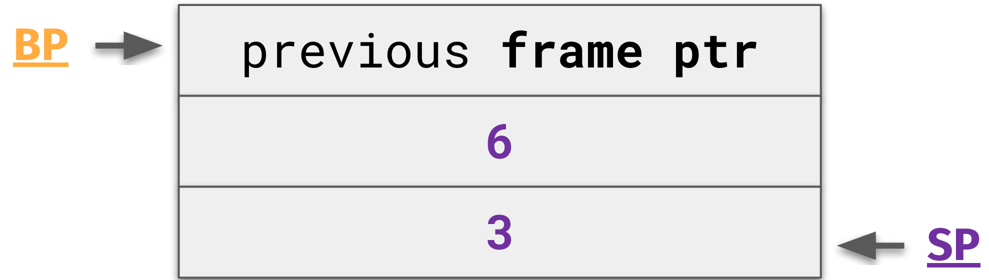
Example Program

```
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```



Example Program

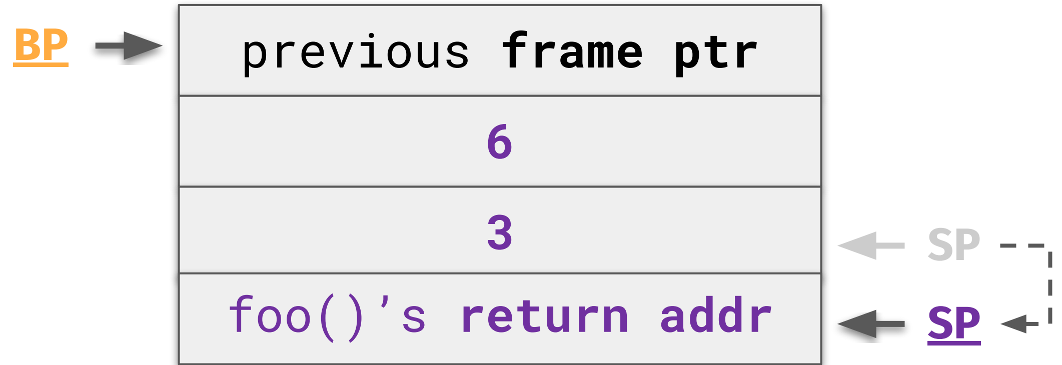
```
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```



Function args are pushed in reverse

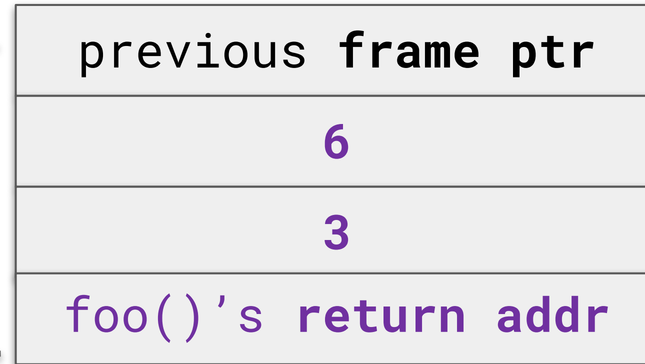
Example Program

```
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```



Example Program

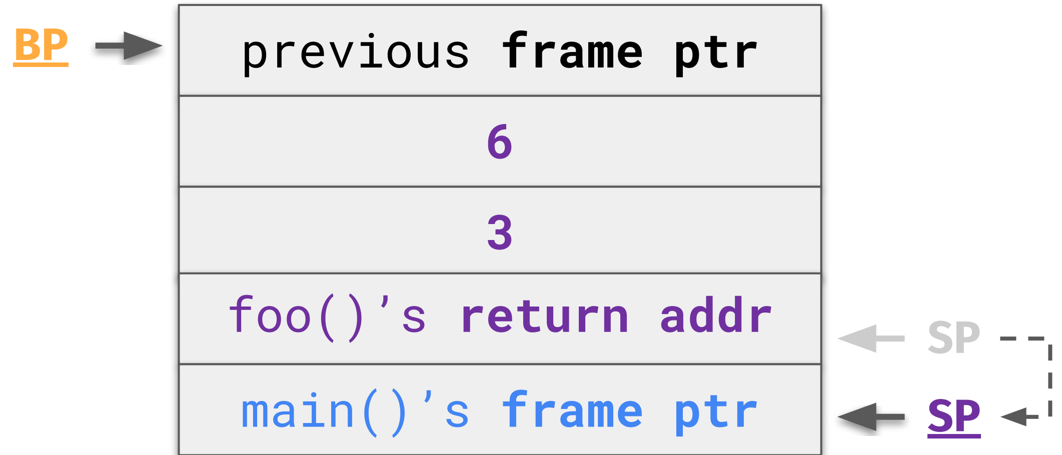
```
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```



foo will return to **main**'s **post-call** instruction

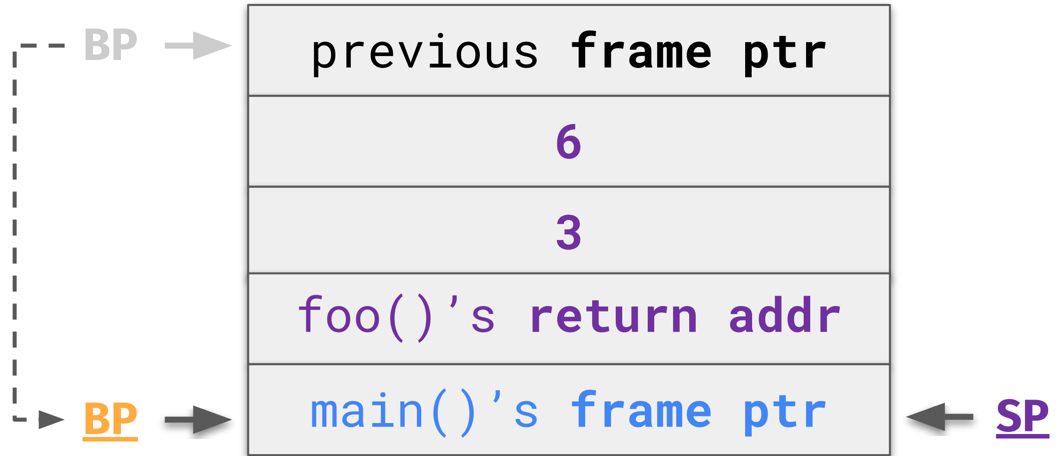
Example Program

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    leave
    ret
```



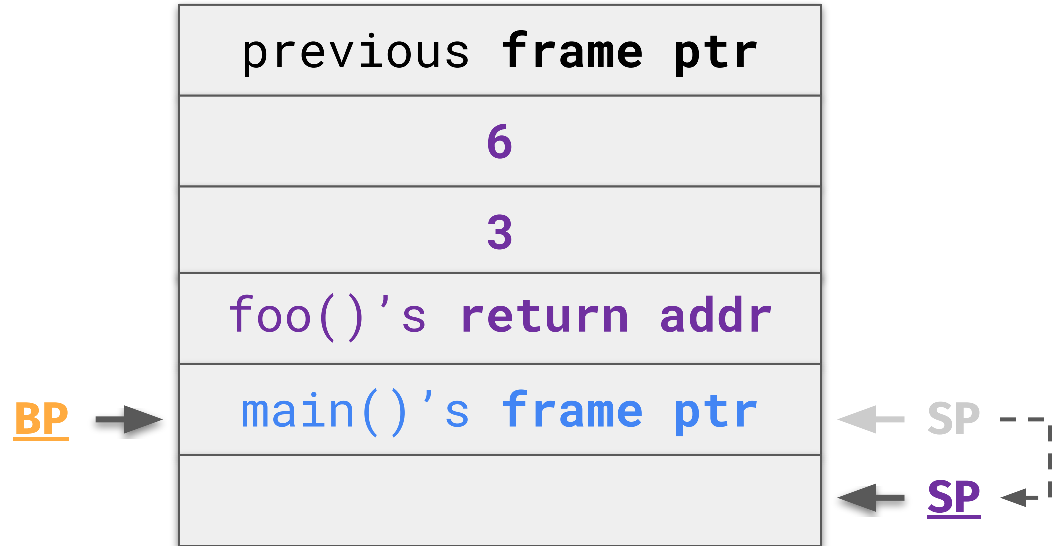
Example Program

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    leave
    ret
```



Example Program

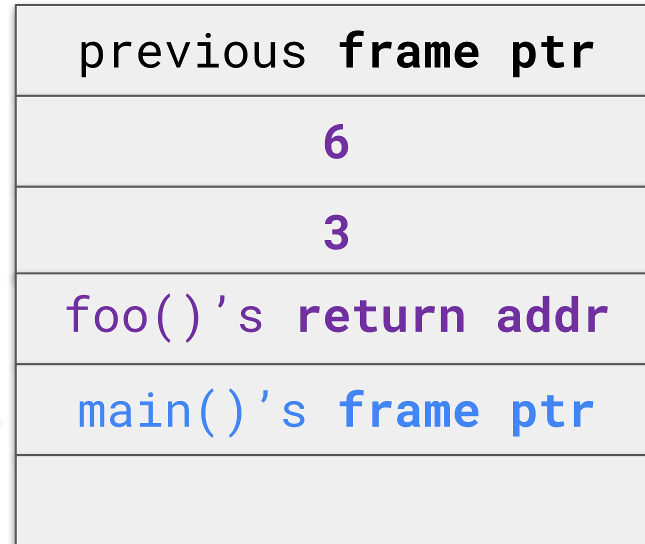
```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    leave
    ret
```



Example Program

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    leave
    ret
```

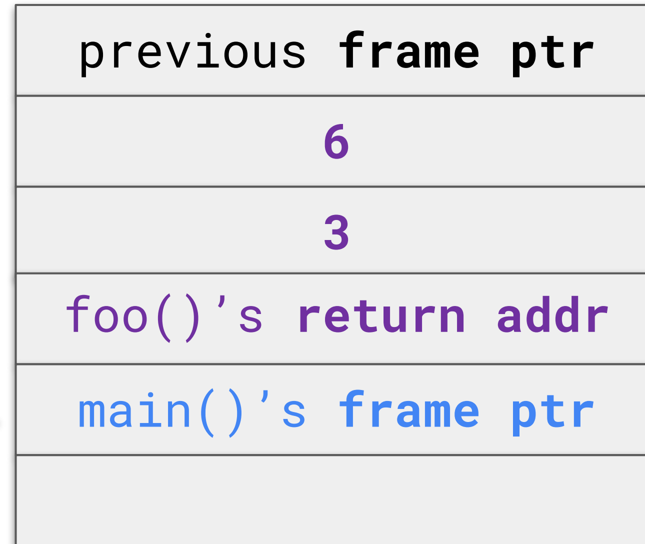
```
mov %ebp, %esp
pop %ebp
```



Example Program

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    leave
    ret
```

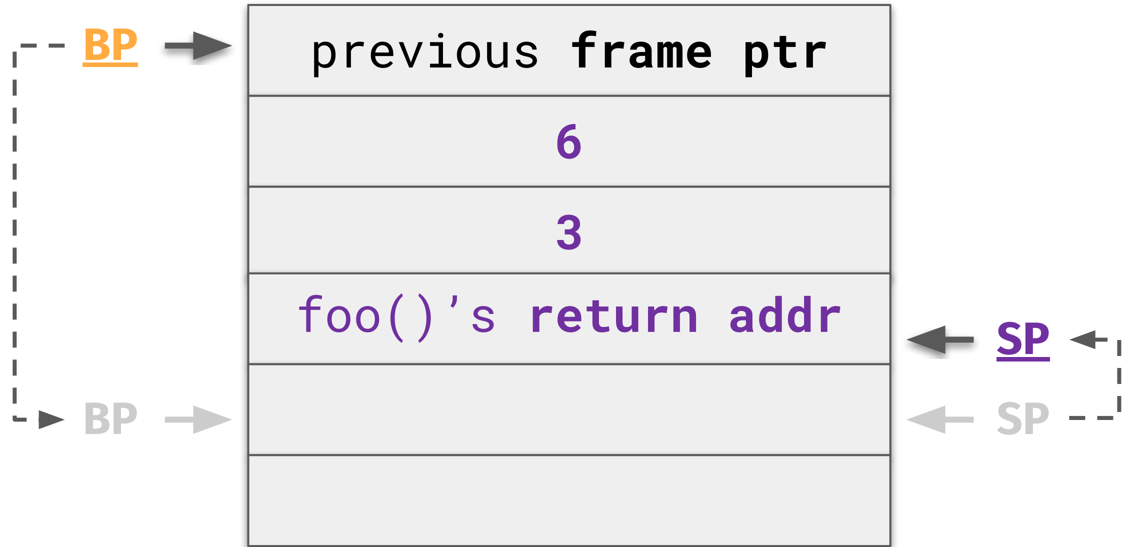
```
mov %ebp, %esp
pop %ebp
```



Example Program

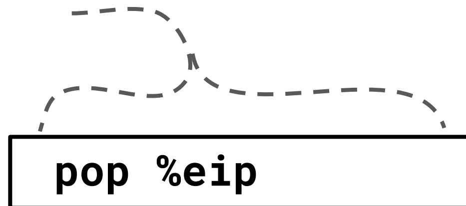
```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    leave
    ret
```

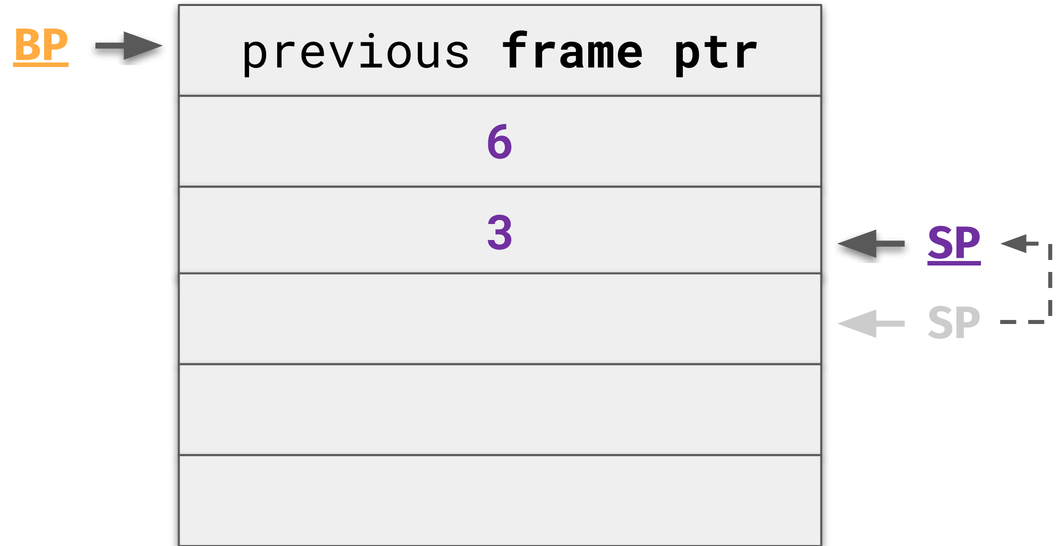
```
mov %ebp, %esp
pop %ebp
```



Example Program

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    leave
    ret
```





Questions?

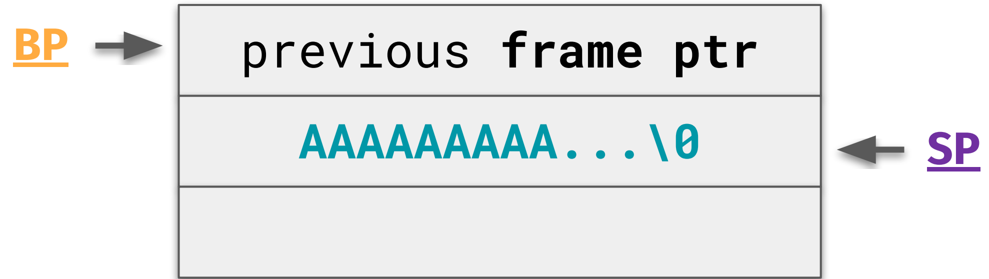
Stack Corruption

Vulnerable Program

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char buf[256];
    memset(buf, ‘A’, 255);
    buf[255] = ‘\x00’;
    foo(buf);
}
```

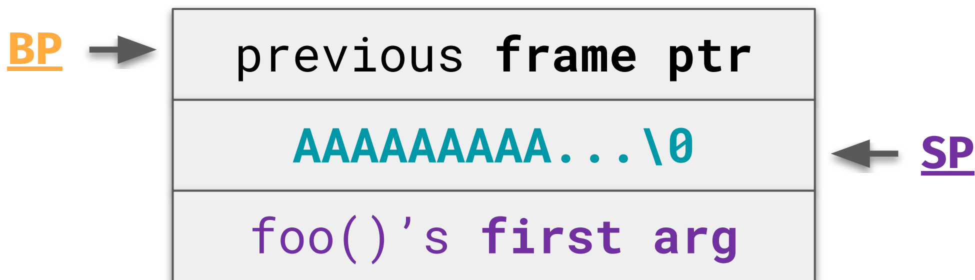
Vulnerable Program

```
void foo(char *str) {
      char buffer[16];
      strcpy(buffer, str);
}
void main() {
      char buf[256];
      memset(buf, ‘A’, 255);
      buf[255] = ‘\x00’;
      foo(buf);
}
```



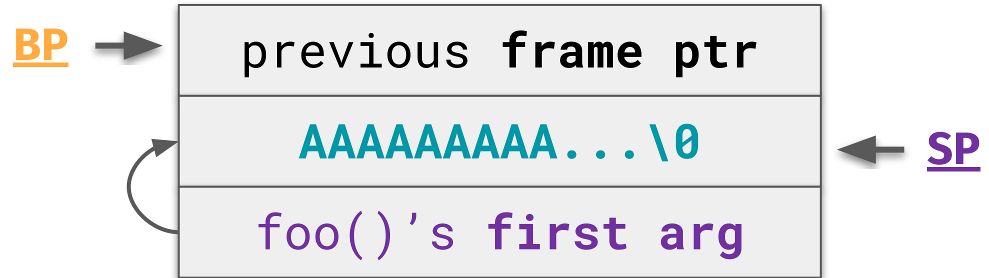
Vulnerable Program

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```



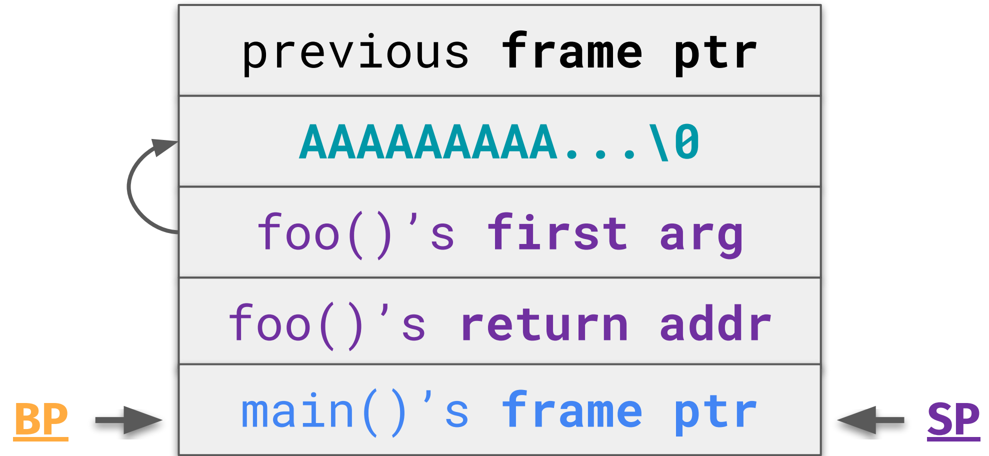
Vulnerable Program

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```



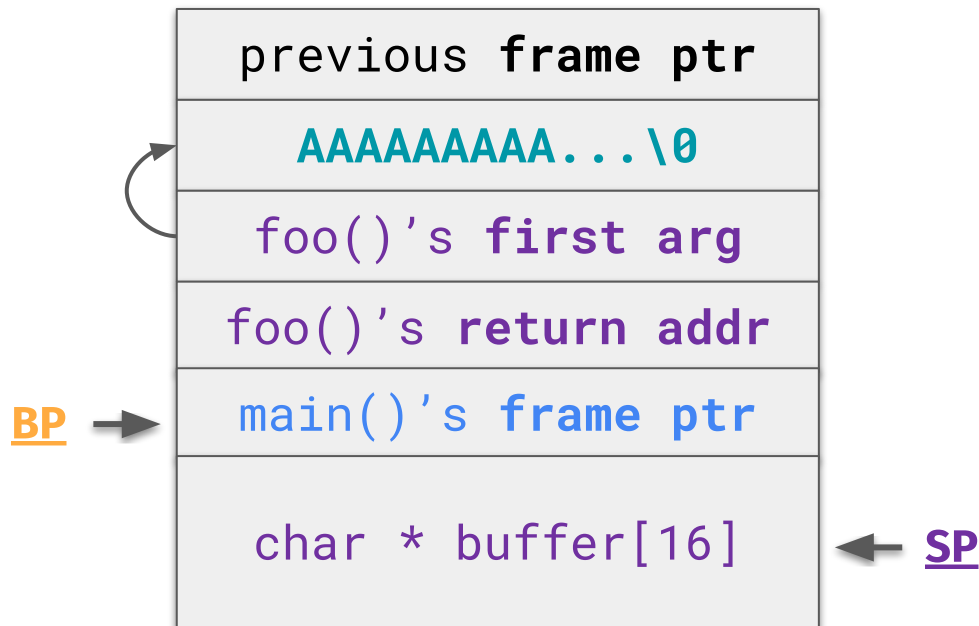
Vulnerable Program

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```



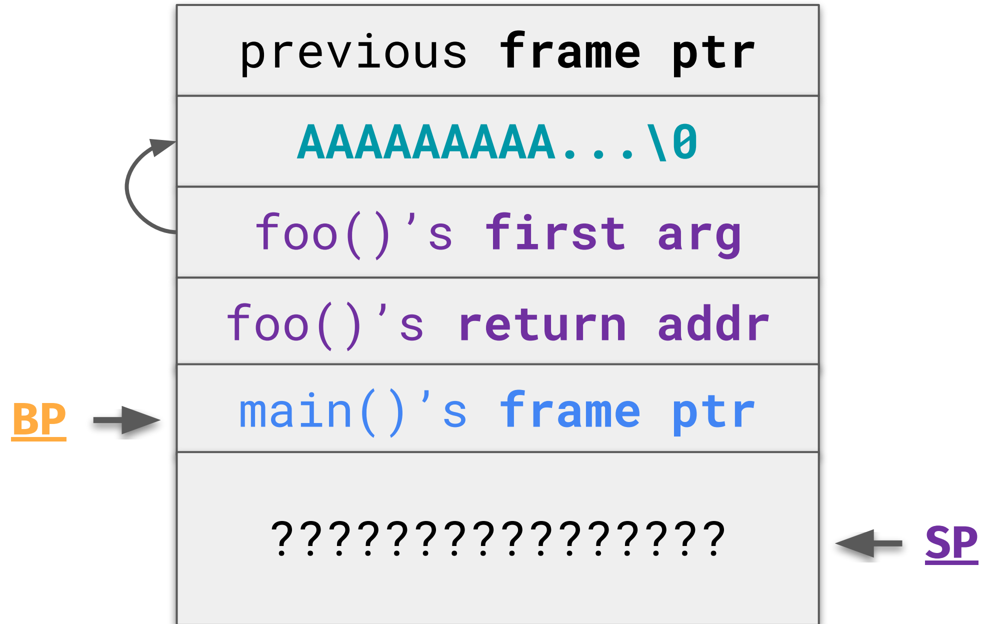
Vulnerable Program

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```



Vulnerable Program

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```

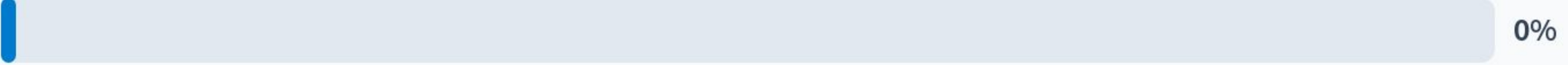


What will happen when we execute strcpy?

It will copy only as many bytes as the buffer can hold.

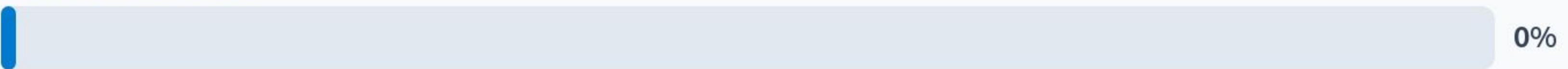
0%

It will realize we're trying to copy more bytes than there's room for, and exit.



0%

None of the above

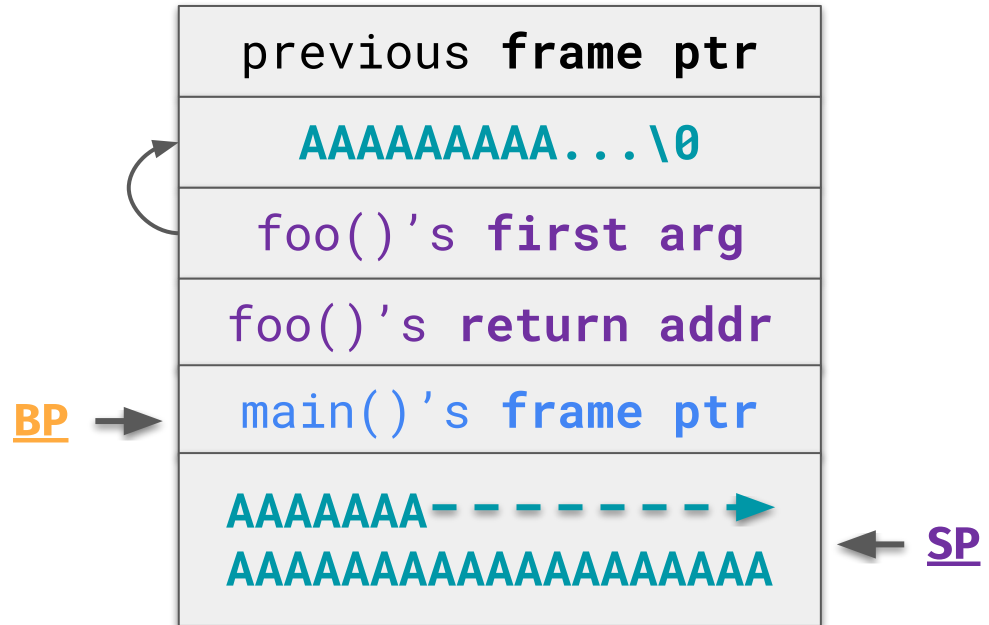


0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

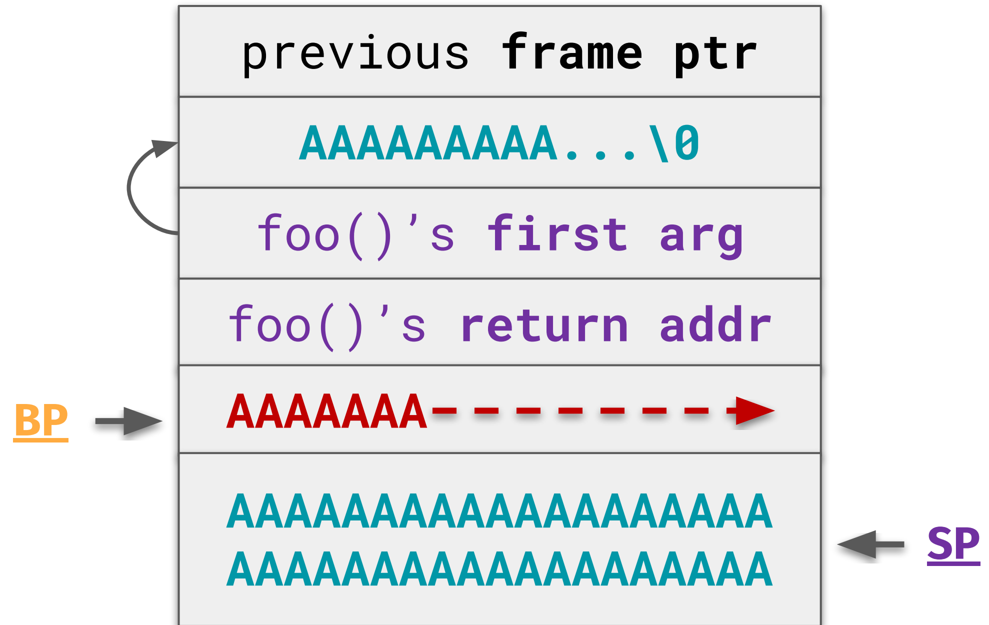
Buffer Overflow!

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```



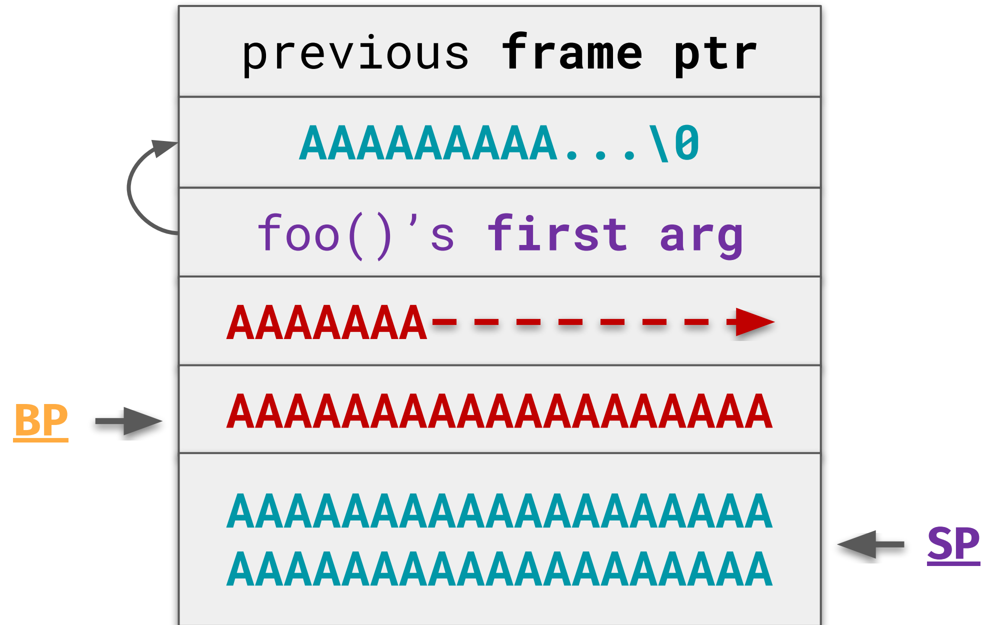
Buffer Overflow!

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```



Buffer Overflow!

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```



Buffer Overflow!

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```

Why does it overflow?

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```

`char *strcpy(char *dest, const char *src);`

The **strcpy**() function copies the string pointed to by *src*, including the terminating null byte ('\0'), to the buffer pointed to by *dest*. The strings may not overlap, and the destination string *dest* must be large enough to receive the copy. *Beware of buffer overruns!* (See BUGS.)

Why does it overflow?

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char buf[256];
    memset(buf, ‘A’, 255);
    buf[255] = ‘\x00’;
    foo(buf);
}
```

char *strcpy(char ****dest*, const char ****src***);**

The **strcpy**() function copies the string pointed to by *src*, including the terminating null byte ('\0'), to the buffer pointed to by *dest*. The strings may not overlap, and the destination string *dest* must be large enough to receive the copy. *Beware of buffer overruns!* (See BUGS.)

We are copying **256 bytes** into a **16-byte** buffer!

Why does it overflow?

```
void foo(char *str) {
    cha
    str
}

void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```

ar *src);

pointed
ll byte
The
strings may not overlap, and the destination
string *dest* must be large enough to receive the
copy. *Beware of buffer overruns!* (See BUGS.)

Observation: *any* stack objects **within reach** of the overflow can be **overwritten**!

We are copying **256 bytes** into a **16-byte** buffer!

Why does it overflow?

Observation: *any* stack objects **within reach** of the overflow can be **overwritten**!

Examples: local **variables**, function **arguments**, **return addresses**, etc.!

Why does it overflow?

Tony Hawk's Pro Strcpy

TONY HAWK'S PRO STRCPY

Watch on YouTube

https://icode4.coffee/?p=954

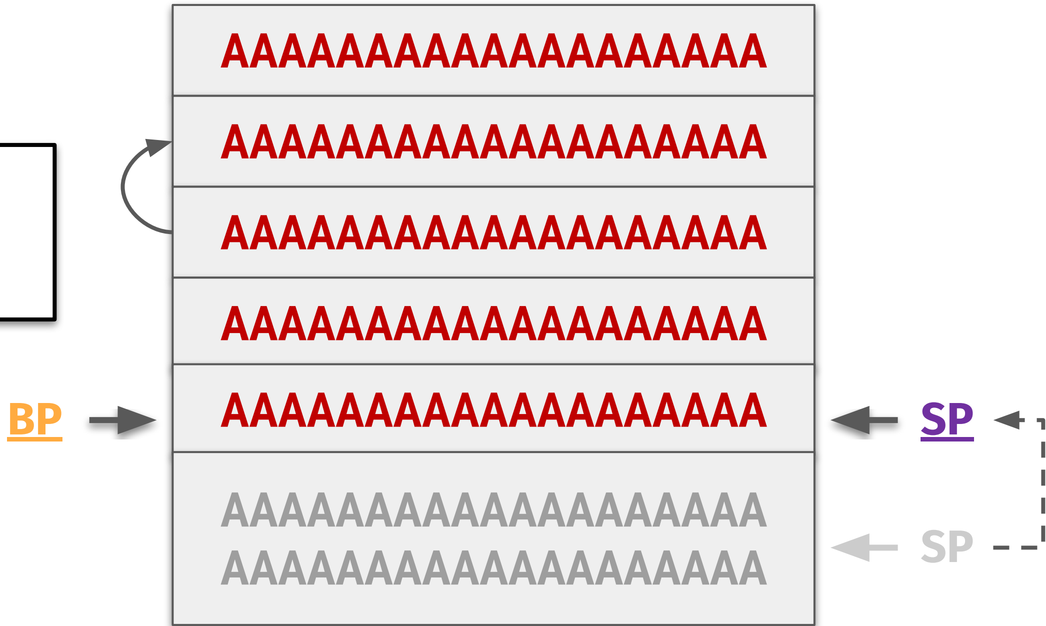
Buffer Overflow (continued)

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```

Buffer Overflow (continued)

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```

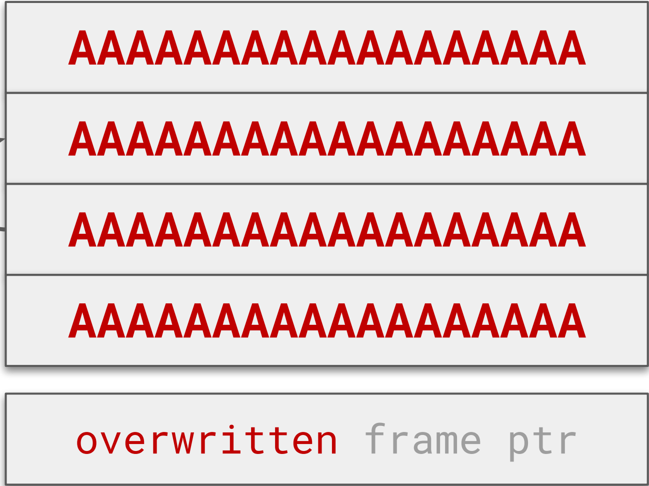
```
mov %ebp, %esp
pop %ebp
pop %eip
```



Buffer Overflow (continued)

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main()
    char bu
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```

```
mov %ebp, %esp
pop %ebp
pop %eip
```

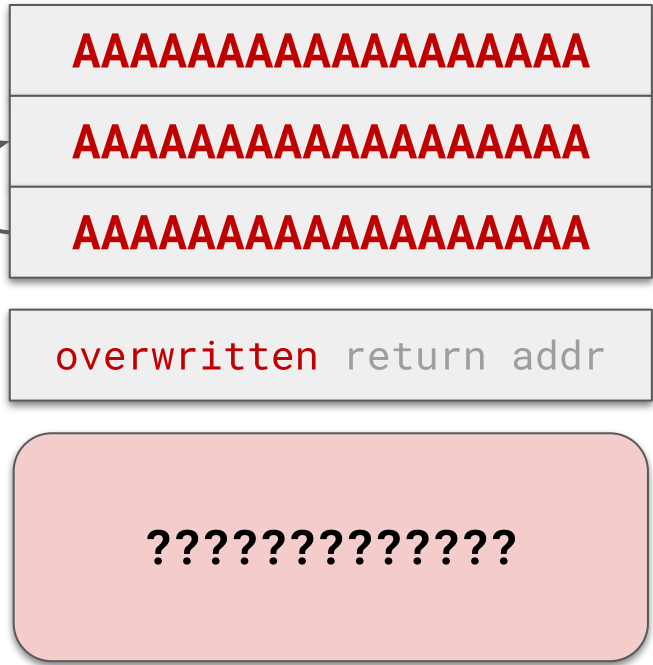


SP

Buffer Overflow (continued)

```
void foo(char *str) {
      char buffer[16];
      strcpy(buffer, str);
}
void main()
      char bu
      memset(buf, 'A', 255);
      buf[255] = '\x00';
      foo(buf);
}
```

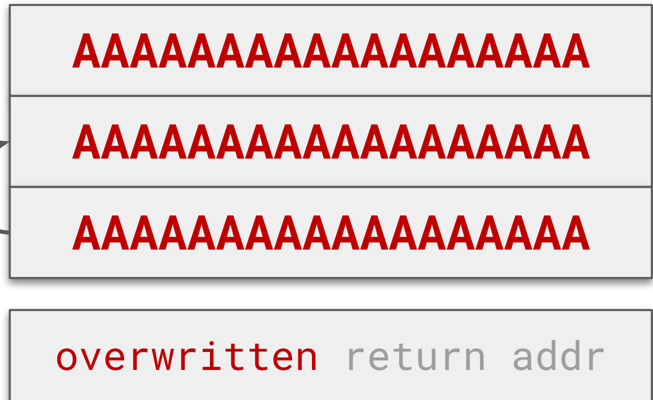
```
mov %ebp, %esp
pop %ebp
pop %eip
```



SP

Buffer Overflow (continued)

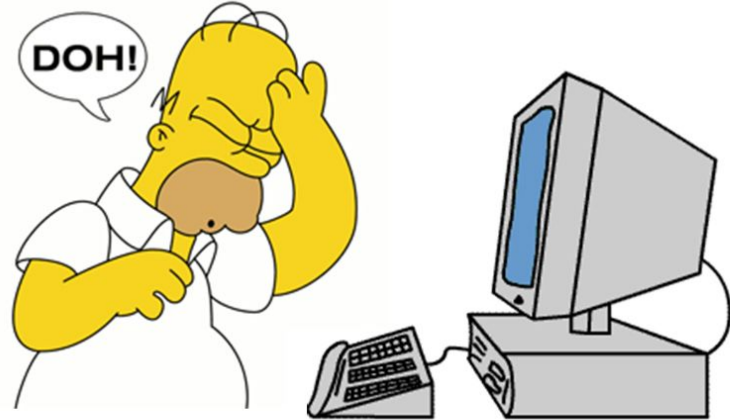
```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main() {
    char bu...
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```



Execution will return to a **garbage address!**

"AAAA" = 0x41414141

Buffer Overflow (continued)




```
segmentation fault.
(Core dumped)
```

Redirecting Execution

```
void foo(char *str) {

}

void

  char bu
  memset(buf, 'A', 255);
  buf[255] = '\x00';
  foo(buf);
}
```

pop %eip

SP

overwritten return addr

Execution will return to a **garbage address!**

"AAAA" = 0x41414141

Observation: when a function returns, execution continues to ***whatever*** its **return address** is…

Redirecting Execution

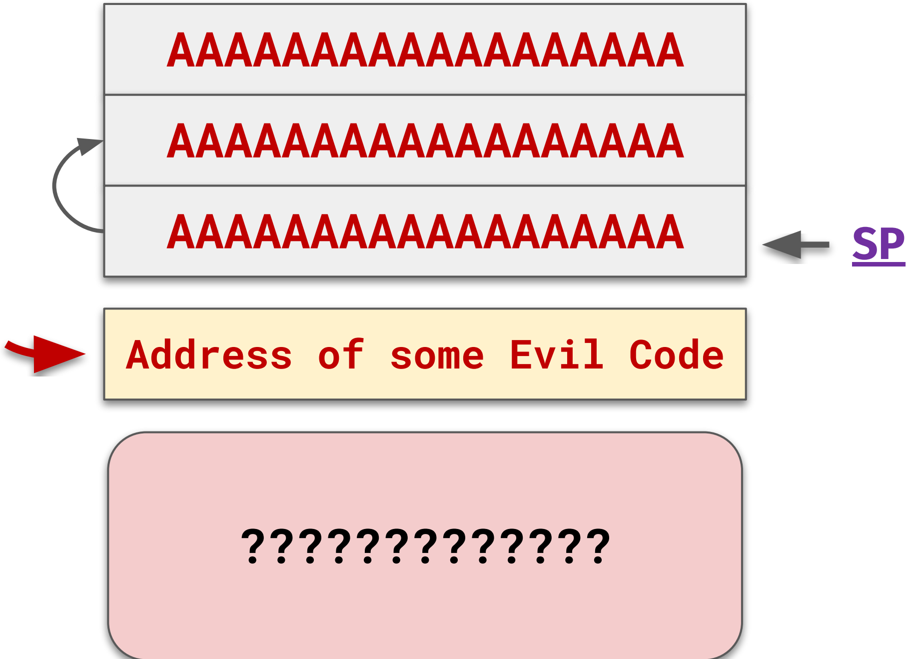
Observation: when a function returns, execution continues to ***whatever*** its **return address** is…

Implication: If Mallory **overwrites** the return address with **something else**, it will be **executed**!

Redirecting Execution

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main() {
    char bu
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```

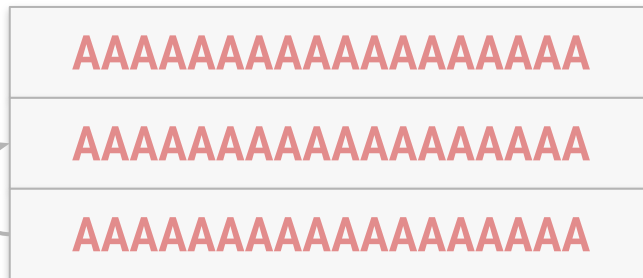
```
mov %ebp, %esp
pop %ebp
pop %eip
```



Redirecting Execution

```
void foo(char *str) {
     char buffer[16];
     strcpy(buffer, str);
}
void main() {
     char buf[256];
     memset(buf, 'A', 255);
     buf[255] = '\x00';
     foo(buf);
}
```

```
mov %ebp, %esp
pop %ebp
pop %eip
```



Execution will return to **the Evil Code's address!**

Redirecting Execution

```
void foo(char *str) {
    char buffer[16];
    strcpy(buf
}
void main()
    char bu
    memset(bu
}
```

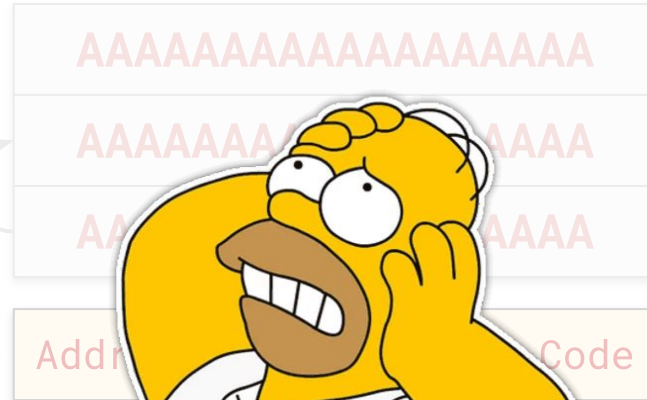
AAAAAAAAAAAAAAAAAA

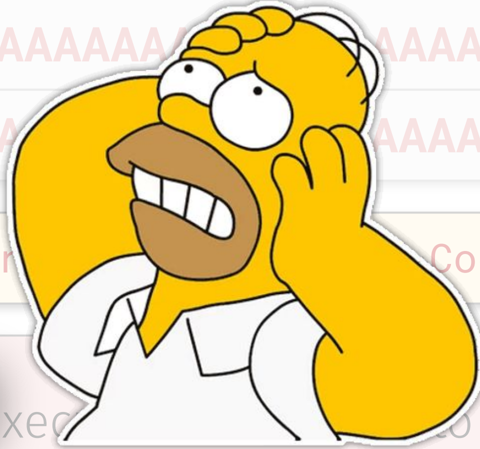
AAAAAAAA AAAA

AA AAAA

SP

Addr Code





Exec o the Evil Code's address!

Questions?

Next time on CS 4440...

Attacking Applications