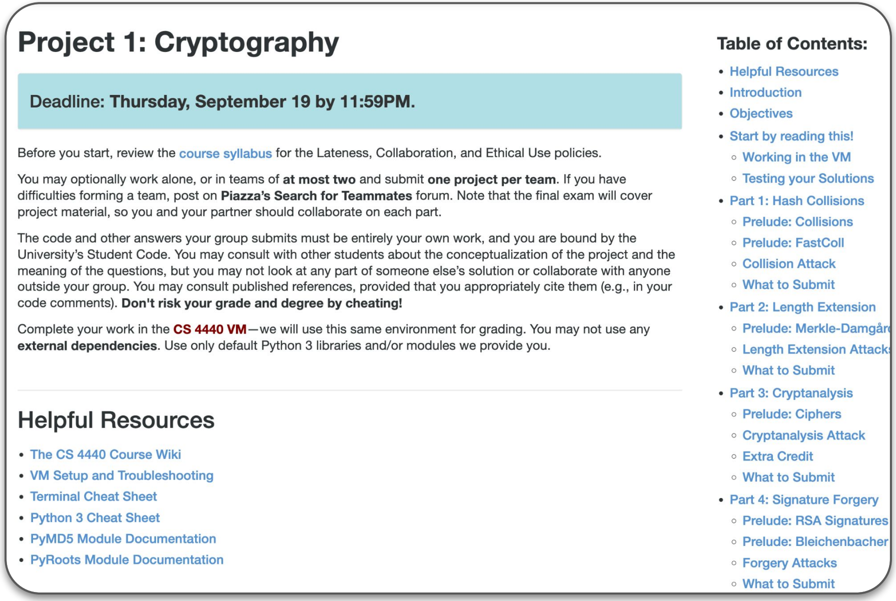
Week 5: Lecture B

Attacking Applications

Thursday, September 19, 2024

Announcements

- **Project 1: Crypto**
  - **Deadline: tonight** 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.

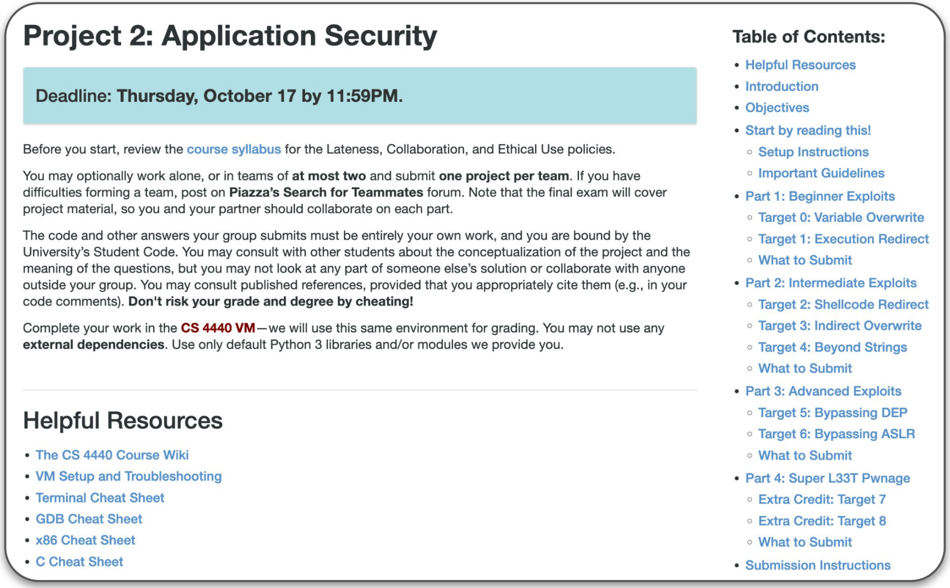
Helpful Resources

- 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:

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:

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

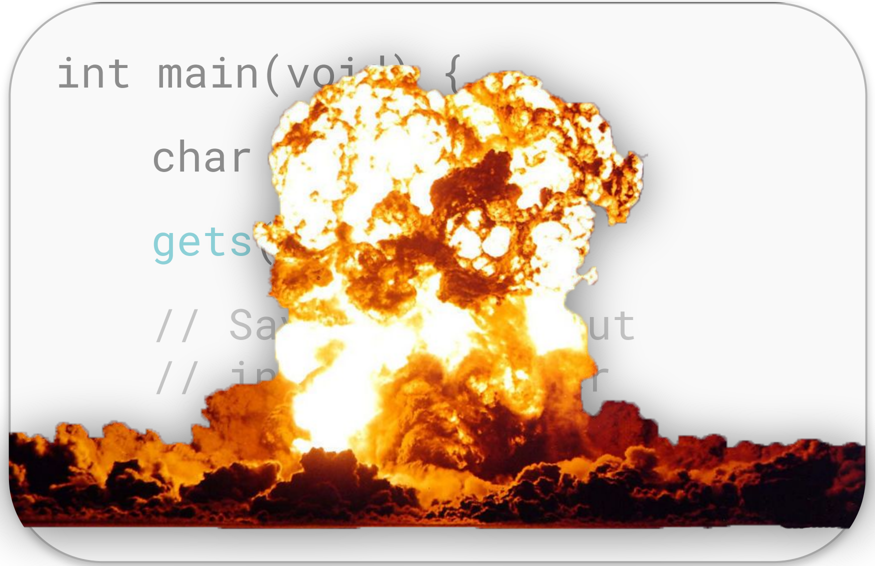
Questions?

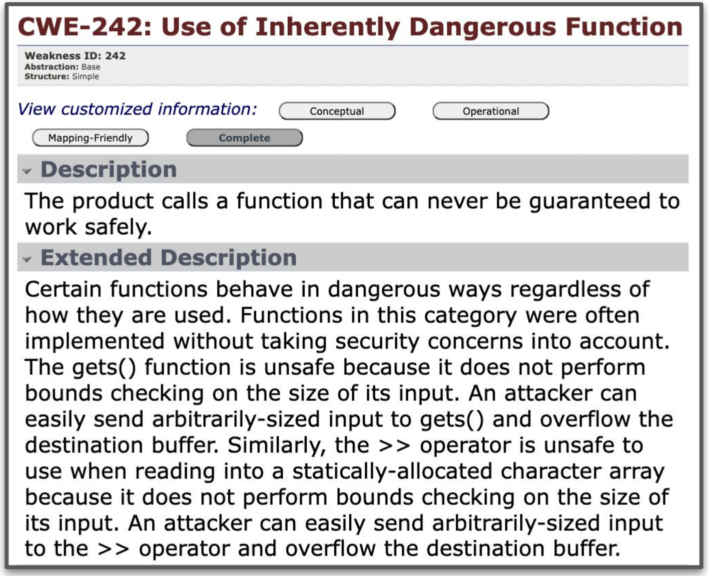
Last time on CS 4440…

Program Execution
Virtual Memory
The Stack
Stack Corruption

Insecure Code

- Software bugs lead to **unintended behavior**





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.

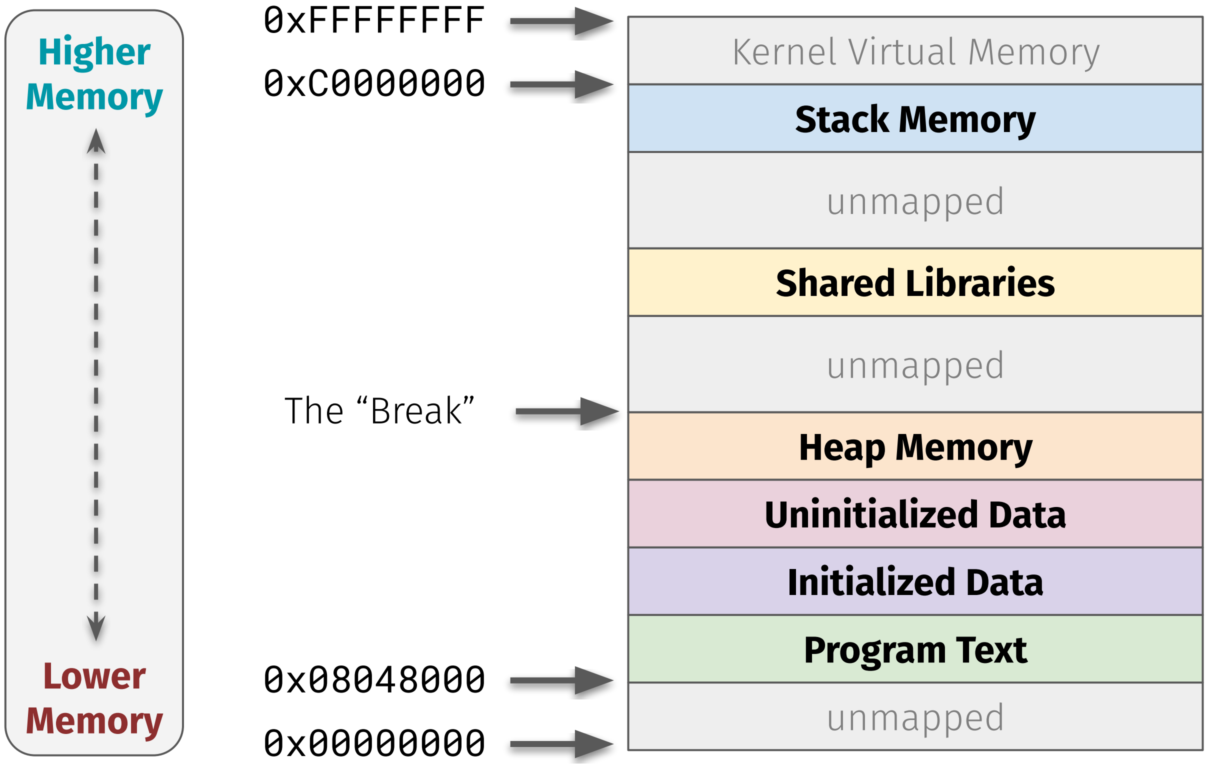
Attacking Computer Systems

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

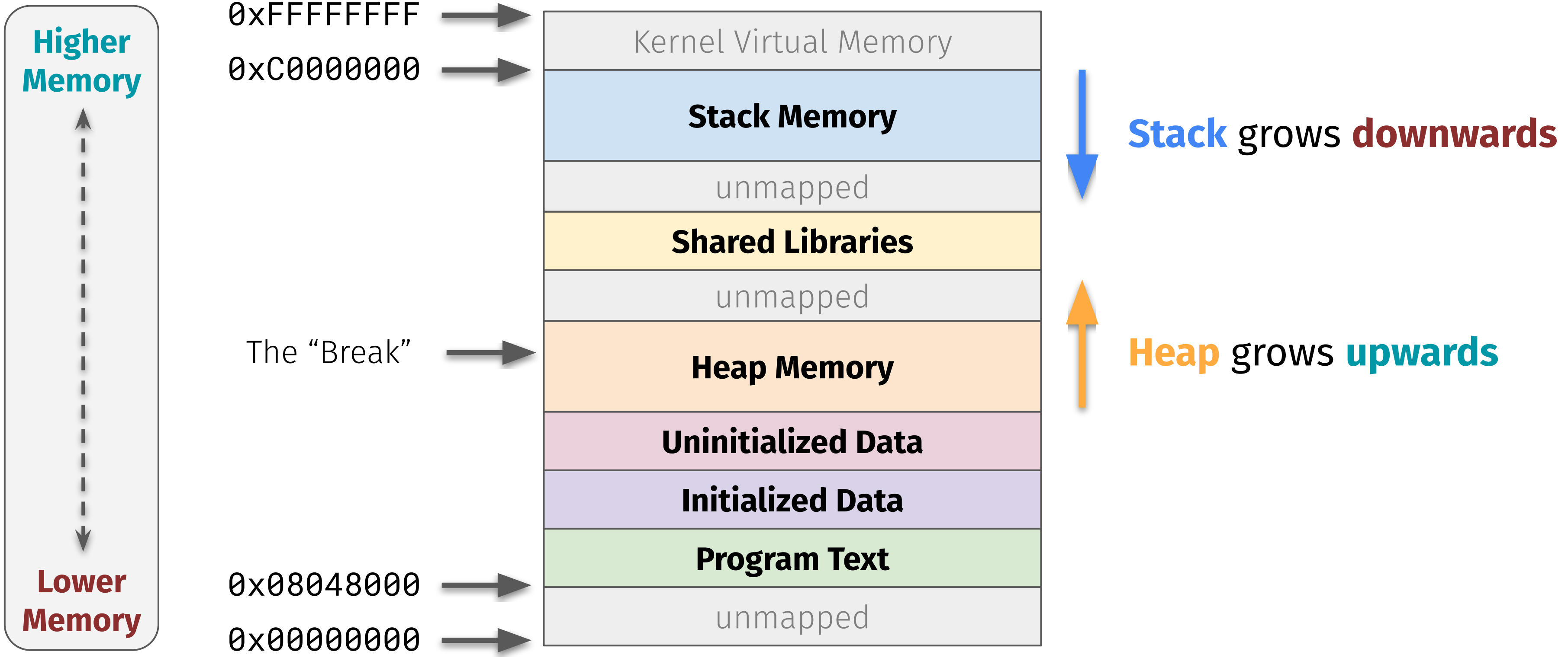
Software Exploitation

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

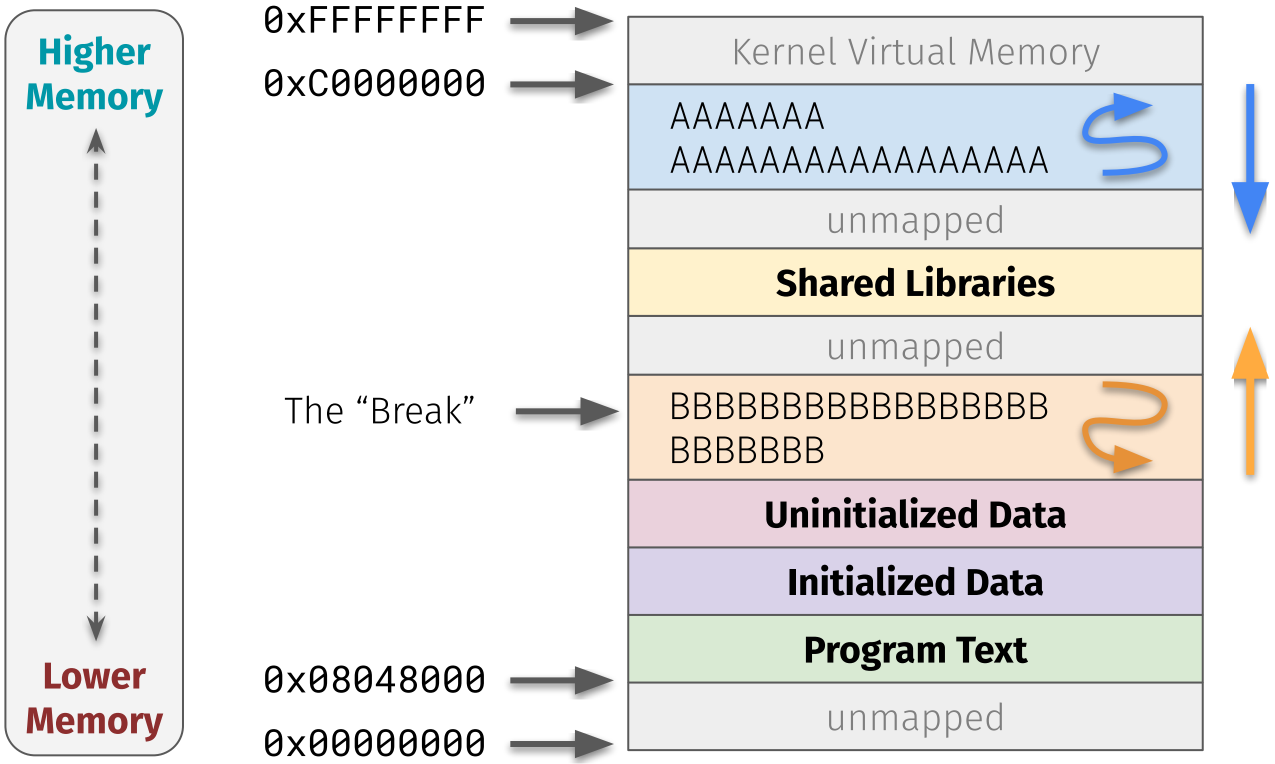
Virtual Memory



Virtual Memory



Virtual Memory

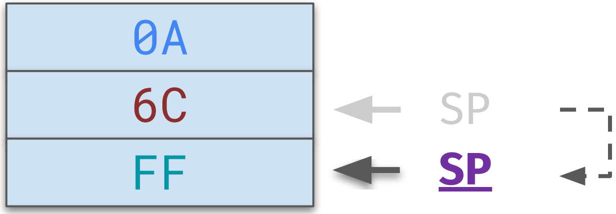


Stack grows **downwards**
- Filled upwards

Heap grows **upwards**
- Filled downwards

Stack Operation

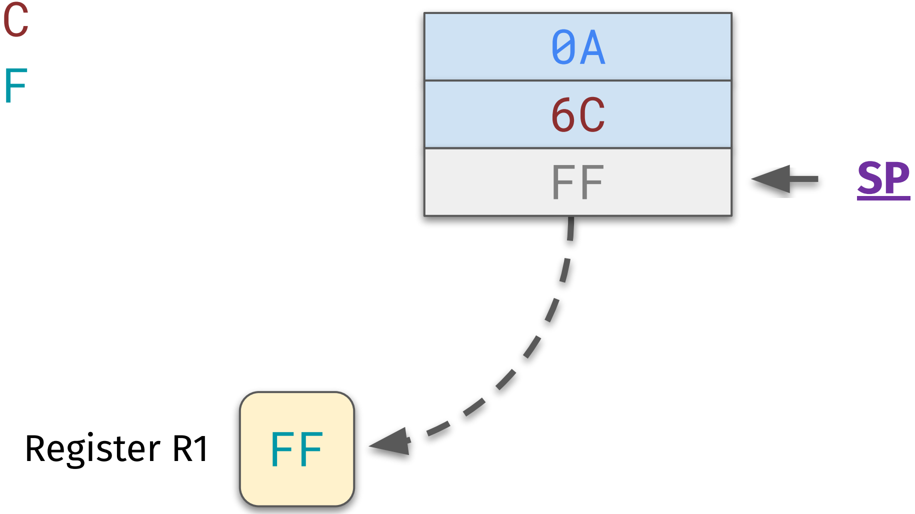
1. Push 0x0A
2. Push 0x6C
3. **Push 0xFF**



Stack grows →
move SP down!

Push and Pop

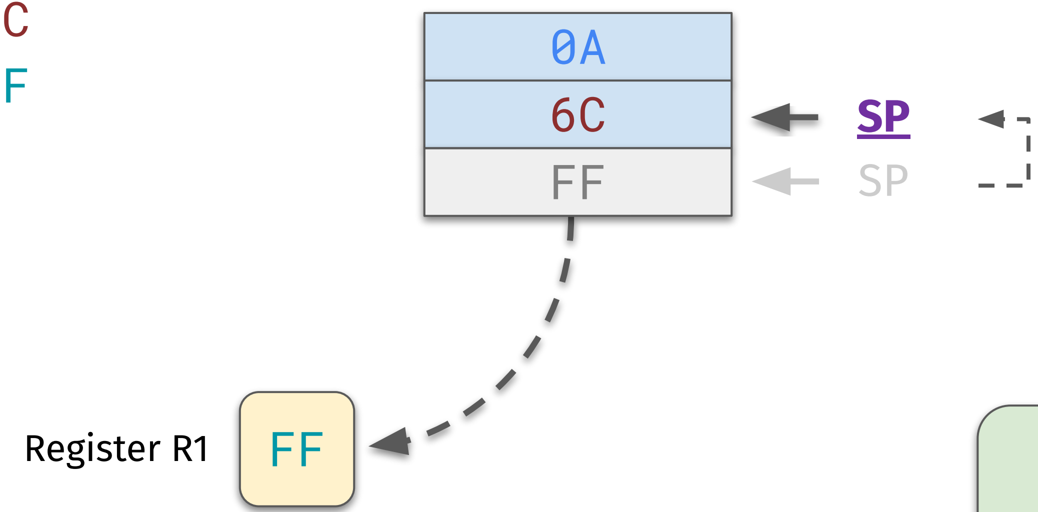
1. Push 0x0A
2. Push 0x6C
3. Push 0xFF
4. **Pop R1**



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

Push and Pop

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



Stack clears →
move SP up!

Stack Frames

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

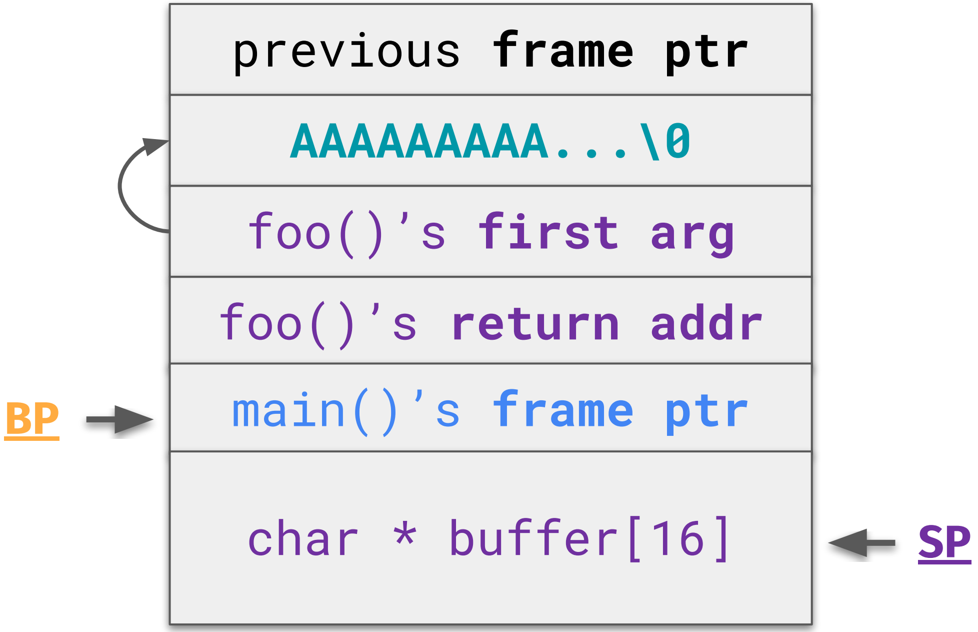
Call-er (`main`) Stack Frame

`main()'s local vars`
`foo()'s arguments`
`foo()'s return addr`
`main()'s frame ptr`
`foo()'s local vars`
`......`

Call-ee (`foo`) Stack Frame

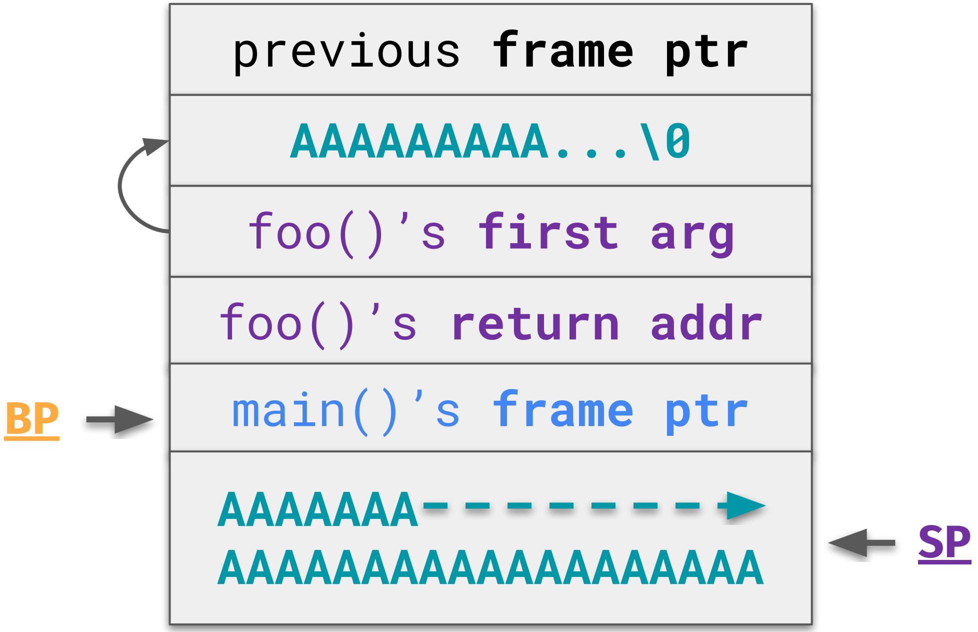
Buffer Overflow!

```
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!

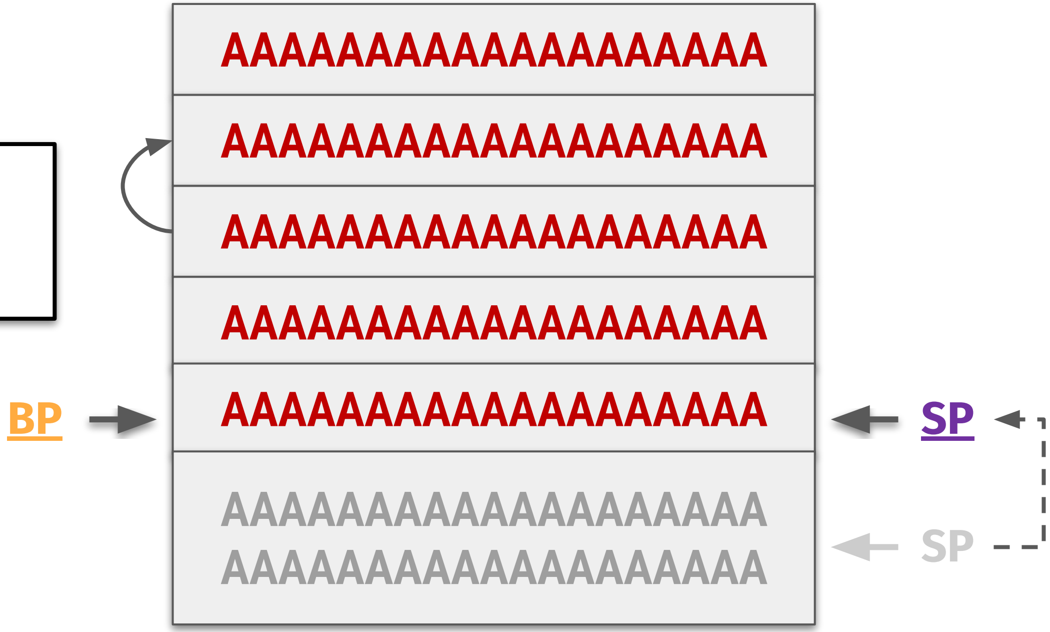
```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```



Buffer Overflow!

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

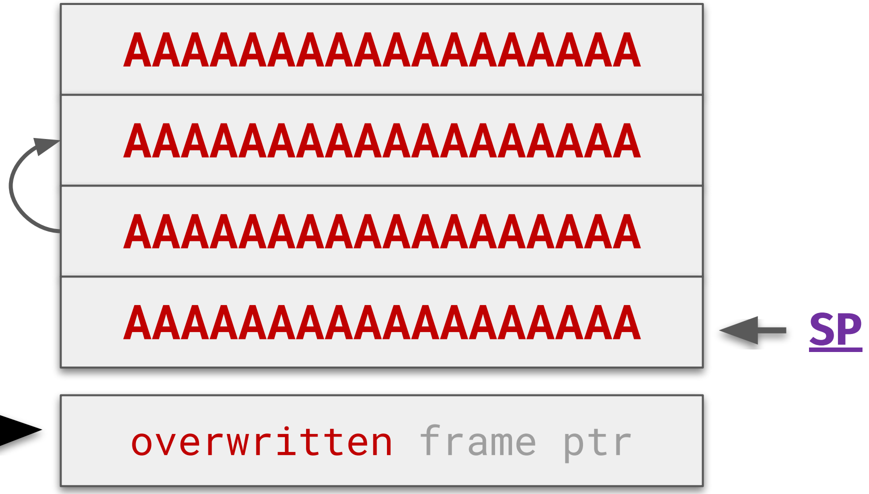
```
mov %ebp, %esp
pop %ebp
pop %eip
```



Buffer Overflow!

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

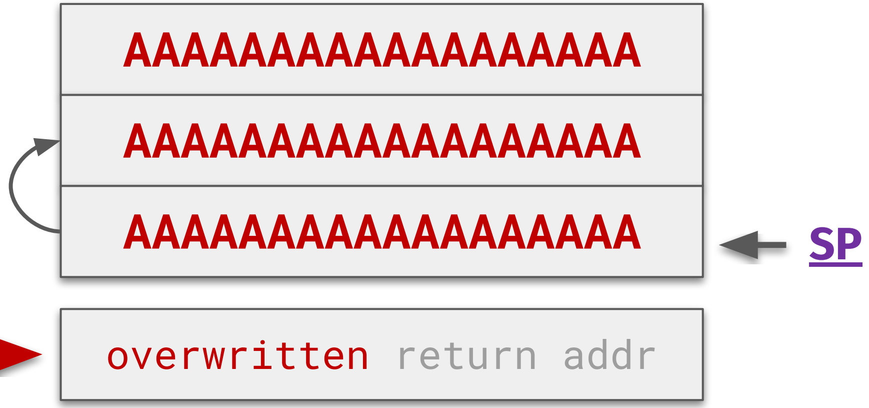
```
mov %ebp, %esp
pop %ebp
pop %eip
```



Buffer Overflow!

```
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
```



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

"AAAA" = **0x41414141**

Questions?

This time on CS 4440…

Shellcode
Constructing Exploits
Pointer Dereferences
Integer Overflows

What goals would an attacker have?

- Controlling a local **variable**
  - E.g., setting variable `grade` to an A+

- Redirect execution to some **function**
  - E.g., calling function `print_good_grade()`

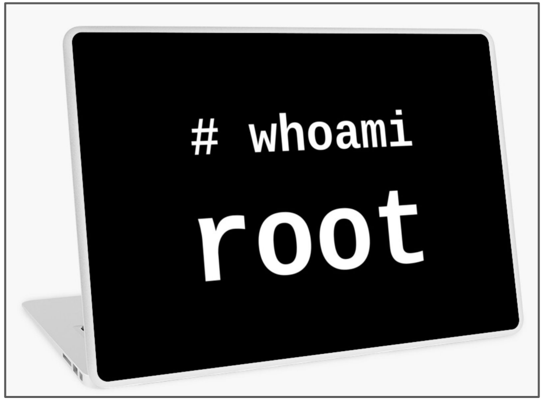
What goals would an attacker have?

- Controlling a local **variable**
  - E.g., setting variable `grade` to an A+
- Redirect execution to some **function**
  - E.g., calling function `print_good_grade()`
- Make the program **execute evil code**
  - **Ideal goal:** gain **root access** to the system

Shellcode

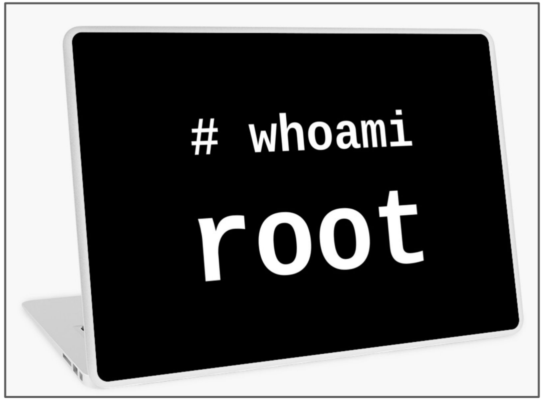
Shellcode

- **Attacker goal:** make program open a **root shell**
  - Root-level permissions = **total system ownage**
  - **You'll do this in Project 2!**
- **Shellcode** = code to open a root shell
  - Inject this somewhere and **direct execution to it**



Shellcode

- **Attacker goal:** make program open a **root shell**
  - Root-level permissions = **total system ownage**
  - **You'll do this in Project 2!**

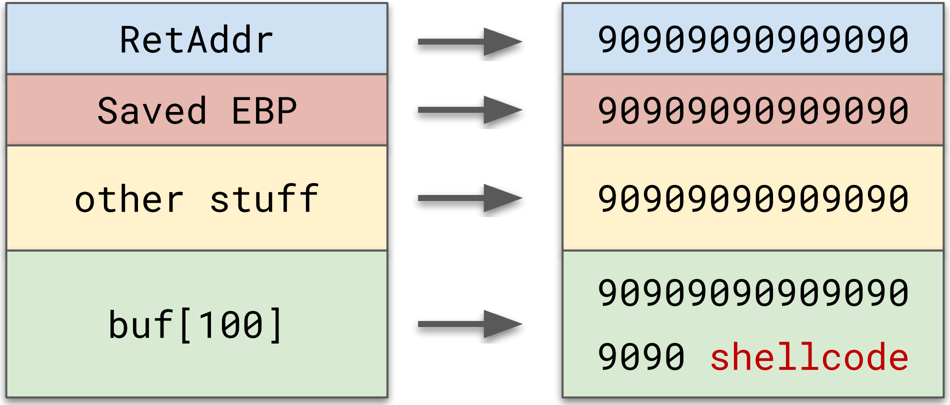


- **Shellcode** = code to open a root shell
  - Inject this somewhere and **direct execution to it**
  - Basic structure:
    1. Call `setuid(0)` to set user ID to "`root`"
    2. Open a shell with `execve("/bin/sh")`

`setuid(0)` + `execve("/bin/sh")`

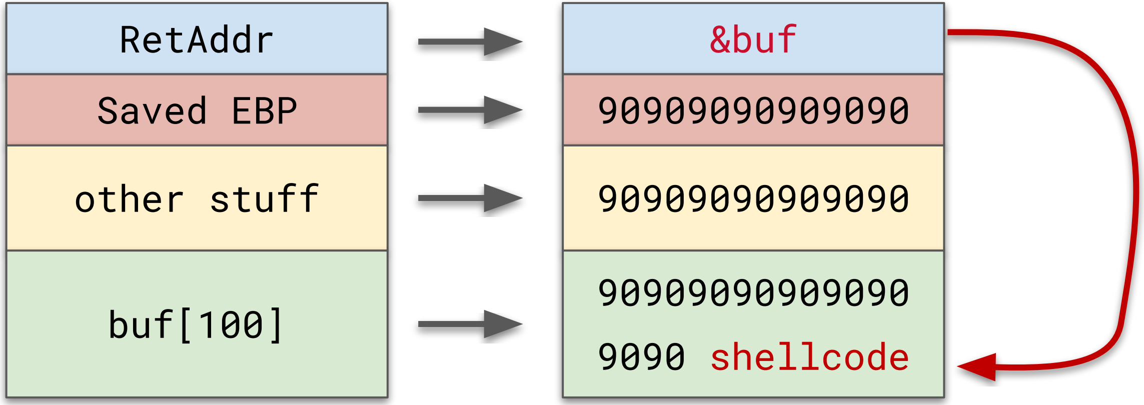
Executing Shellcode

- **Problem**: how can we construct our attack to **execute** our shellcode?



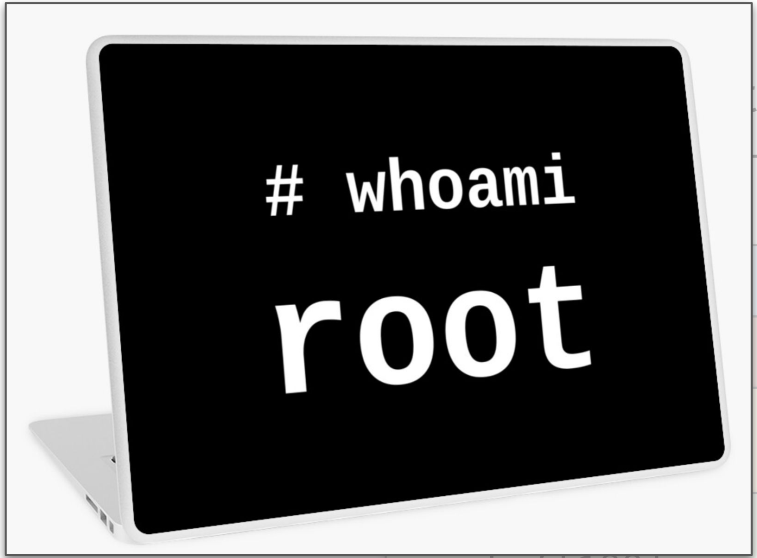
Executing Shellcode

- **Problem**: how can we construct our attack to **execute** our shellcode?
- **Solution:** overwrite **RetAddr** with the address of *where* our shellcode is!
  - We put our shellcode in the **buffer**—so its **starting address** is the buffer's location!



Executing Shellcode

- **Problem**: how can we construct our attack to **execute** our shellcode?



buf[100]

9090 shellcode

Questions?

Constructing Exploits

Project 2 Overview

- **We give you some binaries to exploit**
  - Limited to some rudimentary attacks
    - These don’t exist anymore in practice
    - See **Targets 7–8** for more “realistic” ones

- Various obstacles and defenses to beat
  - **Targets 0–2:** None… **unbounded** overflow!
  - **Target 3:** **Bounded** overflow (`strncpy()`)
  - **Target 4:** Requires a **two-step** exploit
  - **Target 5:** **DEP** (non-executable stack)
  - **Target 6:** **ASLR** (randomized stack location)

Project 2 Overview

- **These challenges seem daunting**
  - We are covering **C**, **x86**, **GDB**, etc.

- **Common questions that I'm seeing:**
  - "I have absolutely zero experience with **C programming**!"
  - "I'm trying to draw the stack but I don't know **assembly**!"
  - "How do I calculate the **exact number of padding** bytes?"
  - "I don't know **where to look** to find this thing in memory!"
  - "My attack should be working, **but it SEGFAULTS**… why?!?!"

Project 2 Overview

- These challenges seem **daunting**
  - We are covering **C**, **x86**, **GDB**, etc.

- Common questions that I'm seeing:
  - "I have absolutely zero experience with **C programming**!"
  - "I'm trying to draw the stack but I don't know **assembly**!"
  - "How do I calculate the **exact number of padding** bytes?"
  - "I don't know **where to look** to find this thing in memory!"
  - "My attack should be working, **but it SEGFAULTS**… why?!"

No expertise necessary!
You'll use just a few skills…

Where to begin?

- Mnemonic device to help guide your attack-planning thought process

D : Dive into the **source code**

E : Estimate the **stack frame**

N : **NOP-out** the entire frame

N : NOP-out the **return address**

I : **Inspect** program's memory

S : **Setup** and **stabilize** attack!

This acronym is silly…

But the **high-level steps** will get you a long way!

D.E.N.N.I.S.

Dive into the source code

Dive into the Source Code

- **Objective: understanding the program**
- **Challenge:** understanding **C programming**

```
int main(int argc, char *argv[])
{
    char grade[5];
    char name[10];
    strcpy(grade, "nil");
    gets(name);
    printf("%s,%s", name, grade);
}
```

Experience with C?

None (that's totally okay!) 0%

Some 0%

Lots! 0%

Dive into the Source Code

- **Objective: understanding the program**
- **Challenge:** understanding **C programming**
  - **Don't sweat it—we don't expect you to master C!**

```
int main(int argc, char *argv[])
{
    char grade[5];
    char name[10];
    strcpy(grade, "nil");
    gets(name);
    printf("%s,%s", name, grade);
}
```

Dive into the Source Code

- **Objective: understanding the program**

- **Challenge:** understanding **C programming**
  - **Don't sweat it—we don't expect you to master C!**

- Ideas from other **OOP languages** carry over
  - **Functions**
  - **Local variables**
  - **Function arguments**
  - Same building blocks as Java, Python, C++, etc.
  - *Finding the "best" order of teaching you these remains an unsolved problem in CS education!*

```
int main(int argc, char *argv[])
{
    char grade[5];
    char name[10];
    strcpy(grade, "nil");
    gets(name);
    printf("%s,%s", name, grade);
}
```

Dive into the Source Code

- **Objective: understanding the program**
- **Challenge:** understanding **C programming**
  - **Don't sweat it—we don't expect you to master C!**
- **Need more info about a function?**
  - **Answer:** locate and read its **manpage**
    - Short for "manual page"
  - E.g., "How is **`strcpy`** different from **`strncpy`**?"
    - https://linux.die.net/man/3/strcpy
    - Many other helpful resources on the web

strcpy(3) - Linux man page

Name

strcpy, strncpy - copy a string

Synopsis

```
#include <string.h>

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

char *strncpy(char *dest, const char *src, size_t n);
```

Description

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.)

The **strncpy**() function is similar, except that at most *n* bytes of *src* are copied. **Warning**: If there is no null byte among the first *n* bytes of *src*, the string placed in *dest* will not be null-terminated.

If the length of *src* is less than *n*, **strncpy**() writes additional null bytes to *dest* to ensure that a total of *n* bytes are written.

Dive into the Source Code

- **Objective: understanding the program**
- **Challenge:** understanding **C programming**
  - **Don't sweat it—we don't expect you to master C!**
- See the **C Cheat Sheet** on the CS 4440 Wiki

CS 4440 Wiki: C Cheat Sheet

The following gives a quick overview of C concepts most relevant to Project 2.

We recommend you familiarize yourself with other detailed C resources. Some great examples are:

- W3 Schools' C Tutorial
- Learn-C's Interactive C Tutorial
- The Linux Man Pages

Functions

Declarations

Function **declarations** include a function's name, the type of the data it returns, and its arguments.

```
void hello()          // This function's return type is "void", meaning it returns nothing.
int add(int a, int b) // This function returns an integer, and takes in two integers a and b.
char *gets(char *s)   // This function returns a char pointer, and takes in one as an arg.
```

C seems daunting, but **you don't need to master it—just understand the basics**, and keep a link or two bookmarked for the rest!

Dive into the Source Code

- **Objective: understanding the program**

- **Fundamental questions to consider:**
  1. What is my **target function**?
  2. What **variables** does it have?
  3. How is data **written** to stack?
  4. **How far** can data be written?
  5. What is **the goal** of my attack?

Example: Target 0

- **Objective: understanding the program**
- **Fundamental questions to consider:**
  1. What is my **target function**?
  2. What **variables** does it have?
  3. How is data **written** to stack?
  4. **How far** can data be written?
  5. What is **the goal** of my attack?

```
int main(int argc, char *argv[])
{
      char grade[5];
      char name[10];
      strcpy(grade, "nil");
      gets(name);
      printf("%s,%s", name, grade);
}
```

Example: Target 0

- **Objective: understanding the program**

- **Fundamental questions to consider:**
  1. What is my **target function**?
     - `main()`
  2. What **variables** does it have?
     - `char grade[5], char name[10]`
  3. How is data **written** to stack?
     - `gets(name)`
  4. **How far** can data be written?
     - As far as we want!
  5. What is **the goal** of my attack?
     - To **overwrite** char `grade[5]`!

```
int main(int argc, char *argv[])
{
     char grade[5];
     char name[10];
     strcpy(grade, "nil");
     gets(name);
     printf("%s,%s", name, grade);
}
```

Target Reconnaissance

Target	What is our attack's **goal**?	How to **write** up the stack?	How **far** can we write?
0	Overwrite **Variable**	`gets()`	**Unbounded**
1	Redirect to **Function**	`strcpy()`	**Unbounded**
2	Redirect to **Shellcode**	`strcpy()`	**Unbounded**

Target Reconnaissance

Target	What is our attack's goal?	How to write up the stack?	How far can we write?
0	Overwrite **Variable**	`gets()`	**Unbounded**
1	Redirect to **Function**	`strcpy()`	**Unbounded**
2	Redirect to **Shellcode**	`strcpy()`	**Unbounded**
3	Redirect to **Shellcode**	`strncpy()`	**Bounded**
4	Redirect to **Shellcode**	`fread()`	**Bounded**

Bounded vs. Unbounded Writes

- **Targets 0–2** permit **unbounded** writes
  - We can overwrite **anything** in the higher stack memory
  - Thanks to dangerous functions `gets()` and `strcpy()`
  - Definitely don't use these functions in your own code!

Bounded vs. Unbounded Writes

- **Targets 0–2** permit **unbounded** writes
  - We can overwrite **anything** in the higher stack memory
  - Thanks to dangerous functions `gets()` and `strcpy()`
  - Definitely don't use these functions in your own code!

- **Targets 3–4** are **bounded** writes… limited reach!
  - **Target 3:** we can only write `8 + sizeof(buf)` bytes
  - **Target 4:** we can only write `count` bytes (via `fread()`)

Bounded vs. Unbounded Writes

- **Targets 0–2** permit **unbounded** writes
  - We can overwrite **anything** in the higher stack memory
  - Thanks to dangerous functions `gets()` and `strcpy()`
  - Definitely don't use these functions in your own code!

- **Targets 3–4** are **bounded** writes… limited reach!
  - **Target 3:** we can only write `8 + sizeof(buf)` bytes
  - **Target 4:** we can only write `count` bytes (via `fread()`)

For **bounded** writes, we have to get creative and **find a way to overwrite** what we want!

Questions?

Overcoming Bounded Writes: Pointer Dereferencing

Overcoming Bounded Writes

- **What observations can we make?**
  - Can they break the program's assumptions?

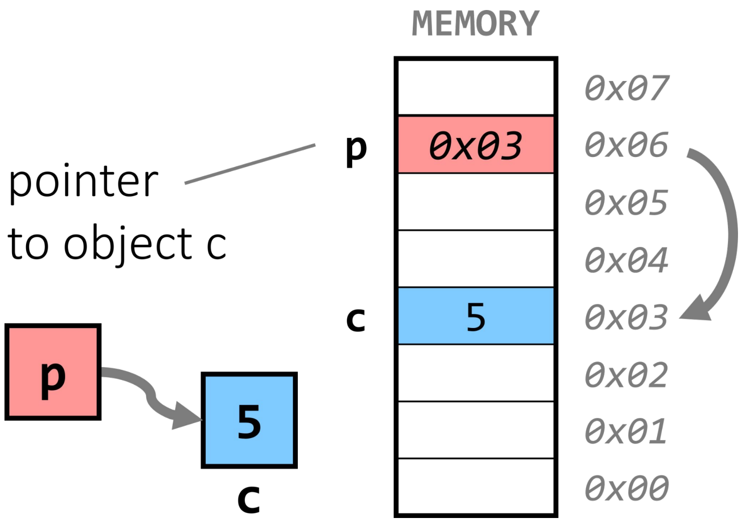
- **Target 3: ???**

```
int *p;
int  a;
*p = a;
```

Overcoming Bounded Writes

- **What observations can we make?**
  - Can they break the program's assumptions?
- **Target 3:** a **pointer dereference**

```
int *p;
int  a;
*p = a;
```



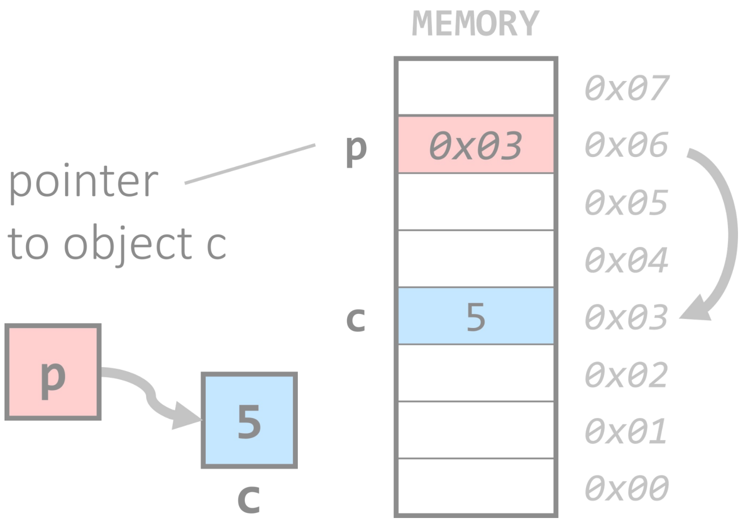
  - If we set `*p = 5`, **whatever p points to** will be updated to 5

Overcoming Bounded Writes

- **What observations can we make?**
  - Can they break the program's assumptions?

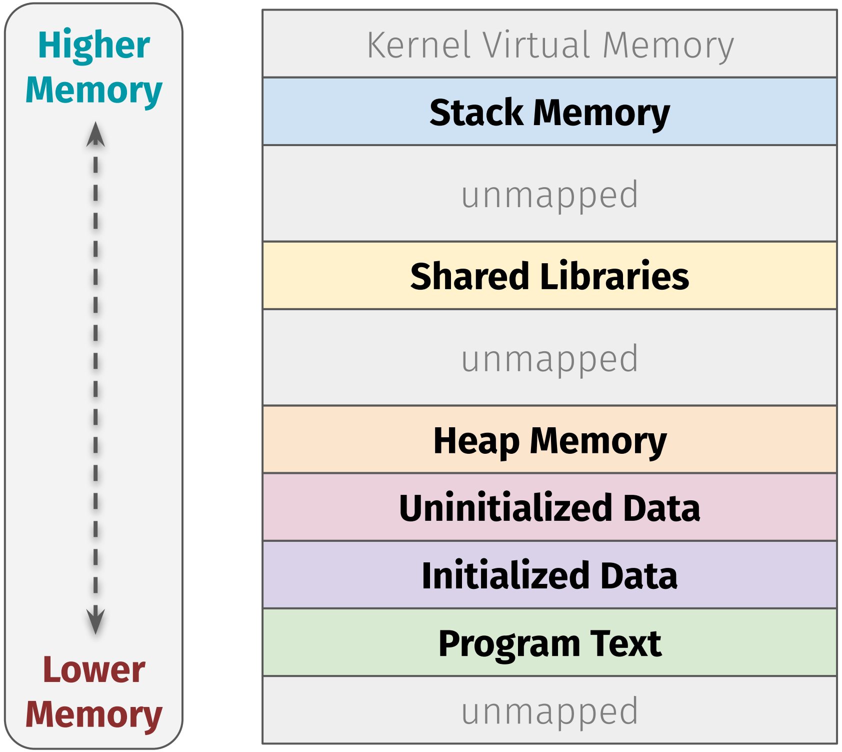
- **Target 3:** a **pointer dereference**

```
int *p;
int  a;
*p = a;
```



  - If we set `*p = 5`, **whatever** p **points to** will be updated to 5
  - If we take control over **both** a *and* p, we can **change arbitrary objects in memory**

Recap: Process Virtual Memory



Recap: Process Virtual Memory

Higher Memory

Lower Memory

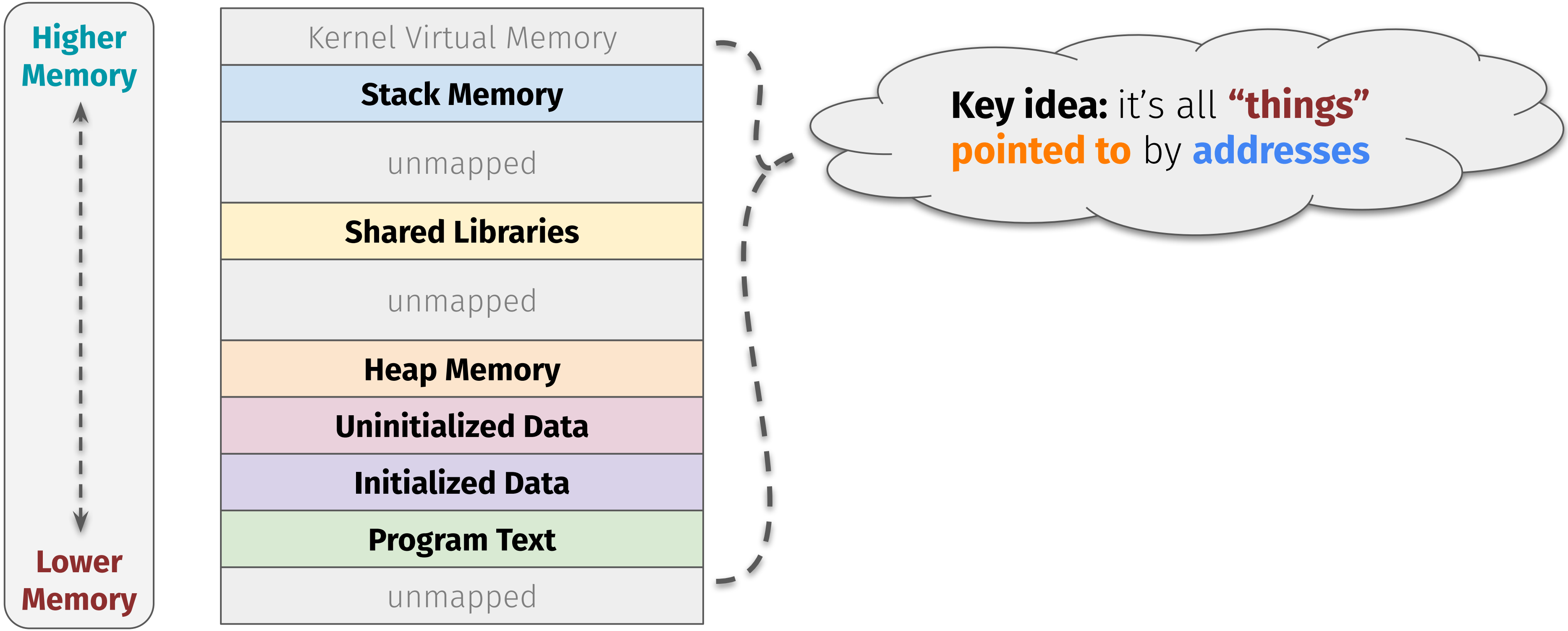
Kernel Virtual Memory
Stack Memory
unmapped
Shared Libraries
unmapped
Heap Memory
Uninitialized Data
Initialized Data
Program Text
unmapped

Local variables, and a record of active functions

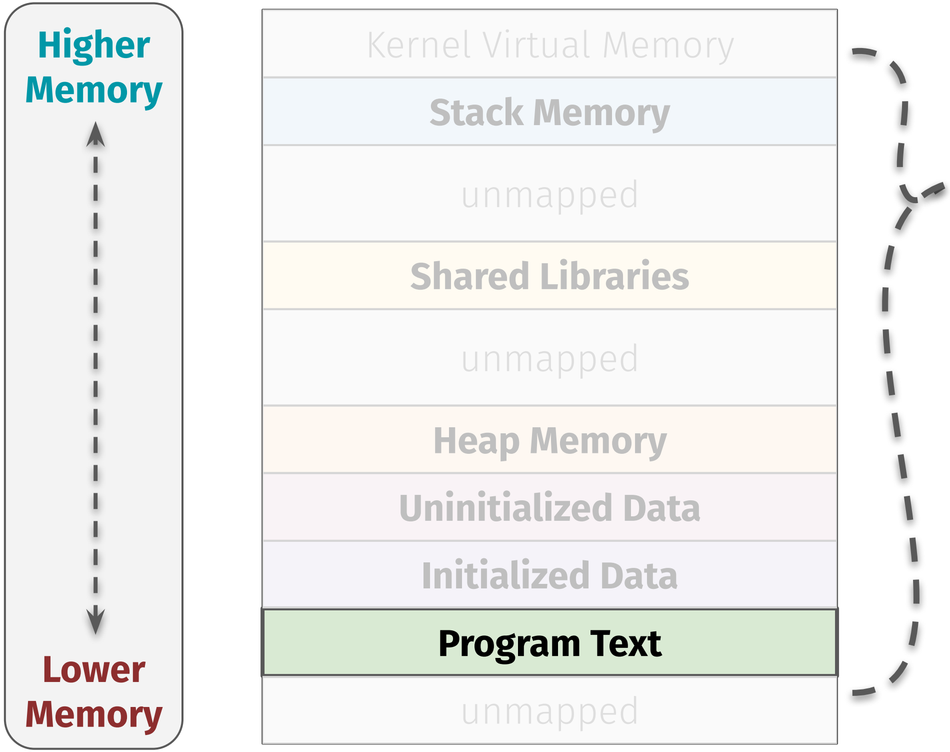
(and a whole bunch of other stuff…)

Program instructions

Recap: Process Virtual Memory



Recap: Process Virtual Memory

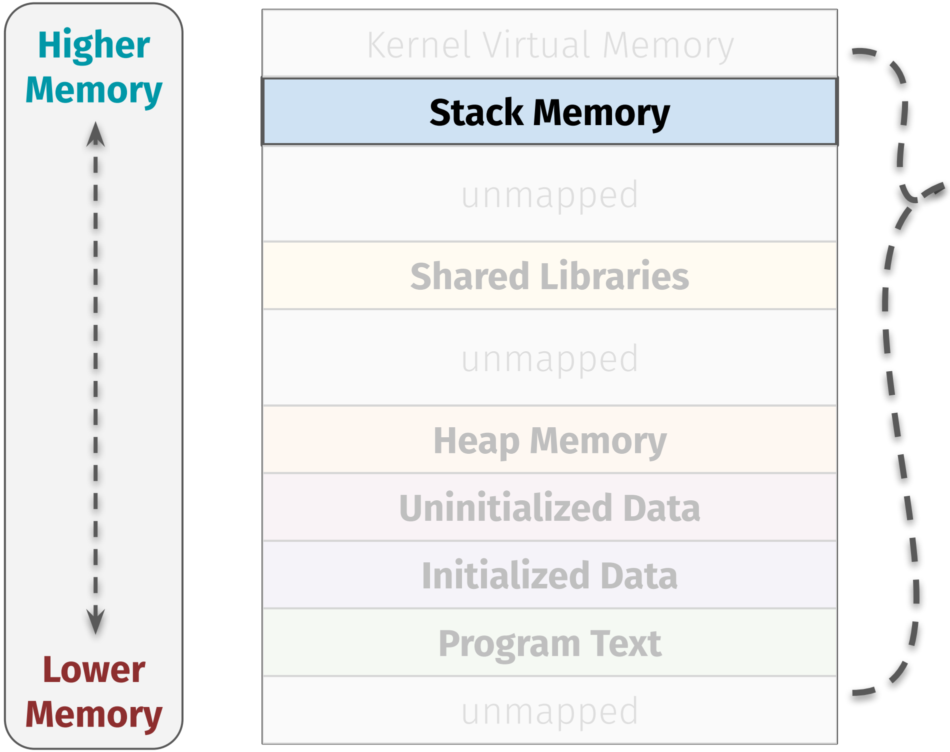


Key idea: it's all **"things" pointed to** by **addresses**

Example: instructions in the **Program Text:**

```
$ disas vulnerable:

0x0804a17b <+0>:     endbr32
0x0804a17f <+4>:     push   %ebp
0x0804a180 <+5>:     mov    %esp,%ebp
0x0804a182 <+7>:     push   %ebx
```

Recap: Process Virtual Memory



Key idea: it's all **"things" pointed to** by **addresses**

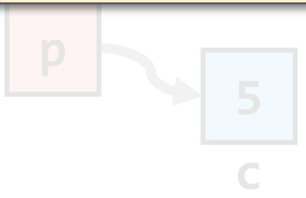
Example: payload NOPs in **Stack Memory**:

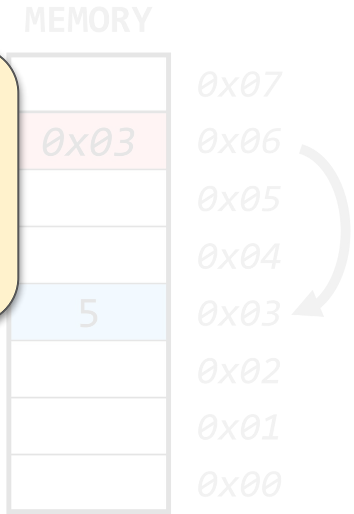
```
$ x/32xw 0xfff6d8cc

0xfff6d8cc:  0x90909090  0x90909090
0xfff6d8d4:  0x90909090  0x90909090
0xfff6d8dc:  0x90909090  0x90909090
0xfff6d8e4:  0x90909090  0x90909090
```

Leveraging Pointer Dereferences

- **What observations can we make?**
  - Can they
- **Target 3:** a

```
int
int  a;
*p = a;
```





Target 3: the return address is stored on the stack. In other words, an **address in stack memory points to** a slot **containing it**.

- If we set `*p = 5`, **whatever** p **points to** will be updated to 5
- If we take control over **both** a *and* p, we can **change arbitrary objects in memory**

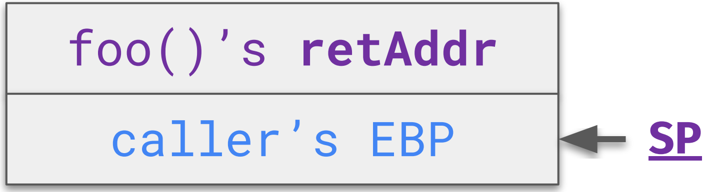
Leveraging Pointer Dereferences

Target 3: the return address is stored on the stack. In other words, an **address in stack memory points to** a slot **containing it**.

We can **exploit the dereference** to overwrite the **value** a **stack memory address points** to!

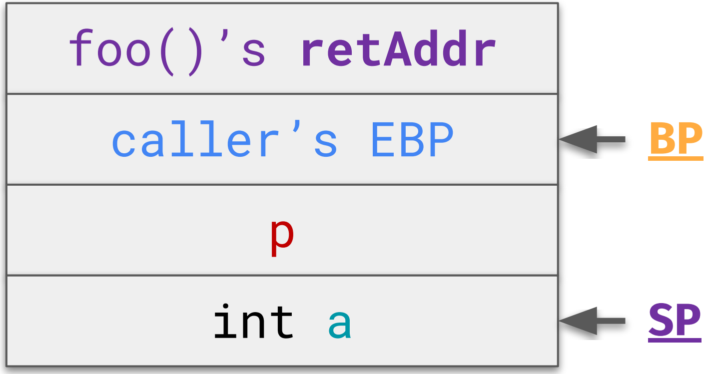
Indirect Memory Overwrite

```
void foo(char *str) {
    int *p;
    int  a;
    *p = a;
}
```



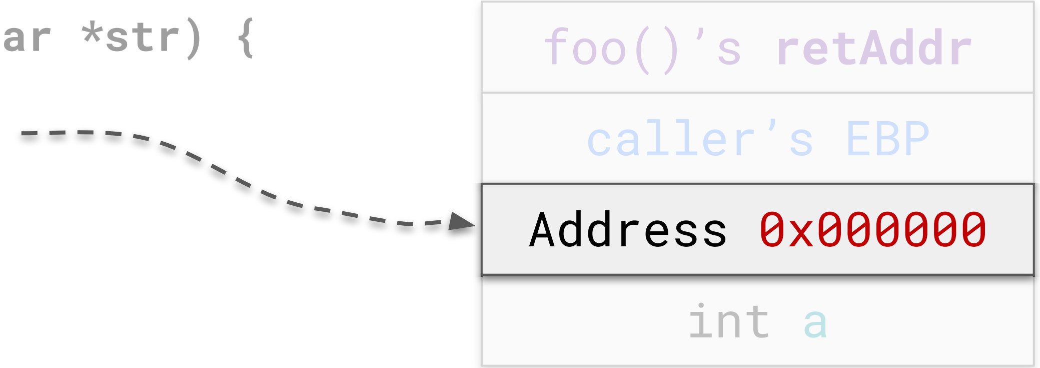
Indirect Memory Overwrite

```
void foo(char *str) {
    int *p;
    int  a;
    *p = a;
}
```



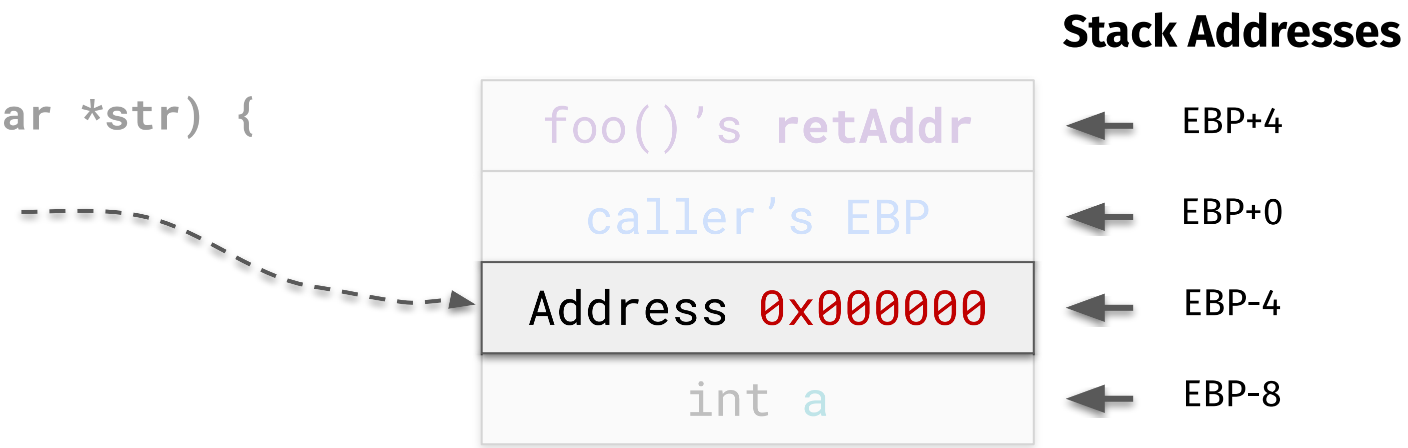
Indirect Memory Overwrite

```
void foo(char *str) {
    int *p;
    int  a;
    *p = a;
}
```



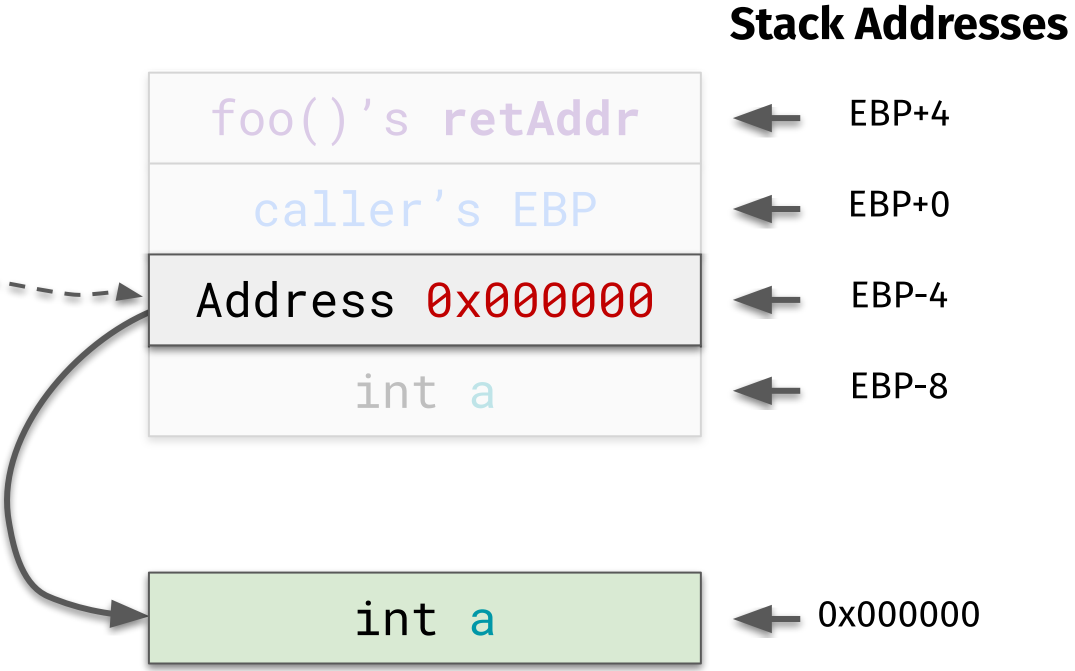
Indirect Memory Overwrite

```
void foo(char *str) {
    int *p;
    int  a;
    *p = a;
}
```



Indirect Memory Overwrite

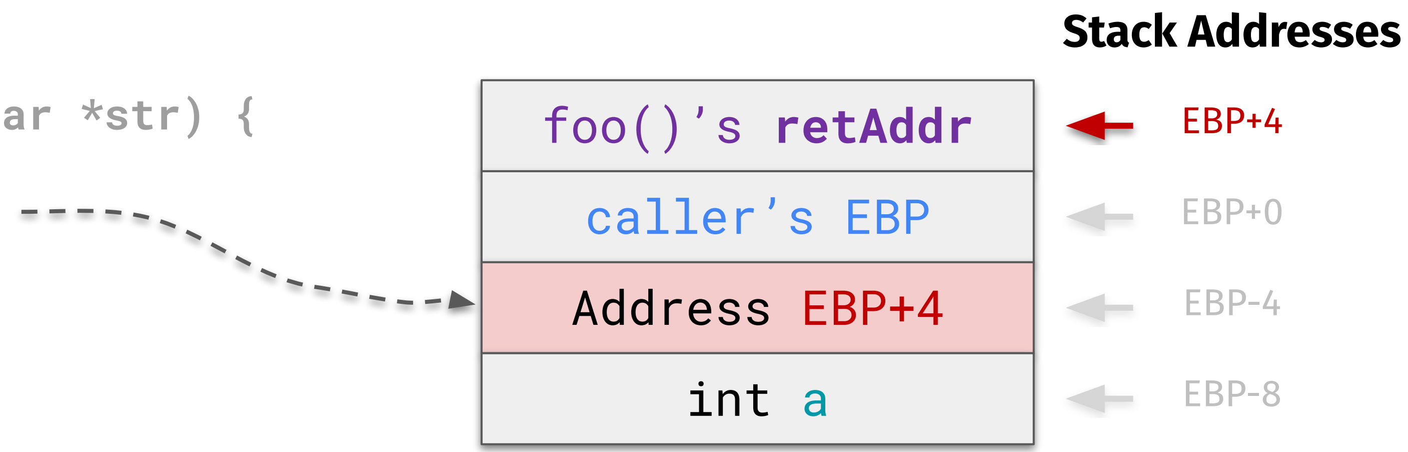
```
void foo(char *str) {
    int *p;
    int  a;
    *p = a;
}
```



Contents of 0x000000 updated to a

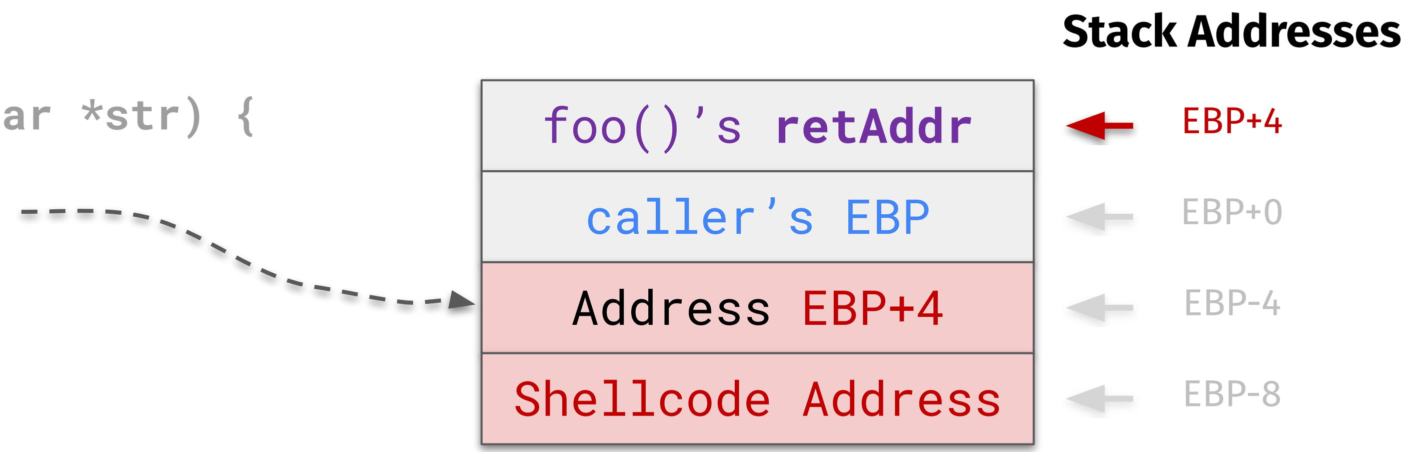
Indirect Memory Overwrite

```
void foo(char *str) {
    int *p;
    int  a;
    *p = a;
}
```



Indirect Memory Overwrite

```
void foo(char *str) {
    int *p;
    int  a;
    *p = a;
}
```



Indirect Memory Overwrite

```
void foo(char *str) {
    int *p;
    int  a;
    *p = a;
}
```

Stack Addresses

Shellcode Address	EBP+4
caller's EBP	EBP+0
Address EBP+4	EBP-4
Shellcode Address	EBP-8

Contents of EBP+4 **updated to the** shellcode address**!**

Target Reconnaissance

Target	What is our attack's goal?	How to write up the stack?	How far can we write?
0	Overwrite **Variable**	`gets()`	**Unbounded**
1	Redirect to **Function**	`strcpy()`	**Unbounded**
2	Redirect to **Shellcode**	`strcpy()`	**Unbounded**
3	Redirect to **Shellcode**	**Dereference Return Addr's stack location**	
4	Redirect to **Shellcode**	`fread()`	

Now update your **high-level plan**!

Other Overwritable Objects

- **Not just return addresses!**
  - Function pointers
  - Arbitrary data
  - C++ exceptions
  - C++ objects
  - Heap memory freelist
  - **Any code pointer!**

Questions?

Overcoming Bounded Writes: Integer Overflows

Overcoming Bounded Writes

- **What observations can we make?**
  - Can they break the program's assumptions?

- **Target 4: ???**

```
alloca( count * 4 );    // allocate our buffer
fread( &buf[i], 4, count, f ); // fill buffer
```

Overcoming Bounded Writes

- **What observations can we make?**
  - Can they break the program's assumptions?

- **Target 4:** a **potential mismatch** of **buffer's size** versus the **data read into it**

```
alloca( count * 4 );   // allocate our buffer
fread( &buf[i], 4, count, f ); // fill buffer
```

Range of count:

[0, ¼(MAX_UINT))

[0,   MAX_UINT)

Overcoming Bounded Writes

- **What observations can we make?**
  - Can they break the program's assumptions?

- **Target 4:** a **potential mismatch** of **buffer's size** versus the **data read into it**

```
alloca( count * 4 );    // allocate our buffer
fread( &buf[i], 4, count, f ); // fill buffer
```

Range of count:

[0, ¼(MAX_UINT))

[0, MAX_UINT)

  - If we perform an **integer overflow** on count, alloca() creates an **artificially small** buffer
  - The resulting fill operation will **exceed the buffer's size**, resulting in a buffer overflow!

Integer Overflows

- **Integer overflows** behave differently from stack buffer overflows

32-bit Integer Range:

Unsigned: `[0, (2^32 - 1)]`
`[0, 4294967295]`

Signed: `[-2^31, (2^31 - 1)]`
`[-2147483648, 2147483647]`

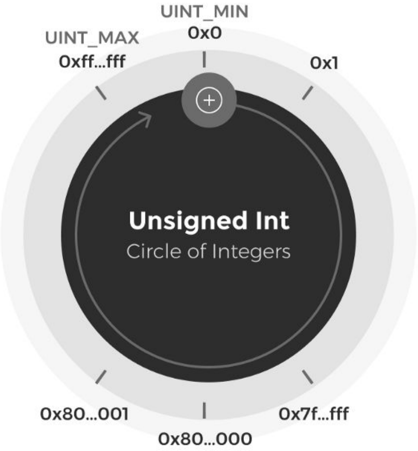
Integer Overflows

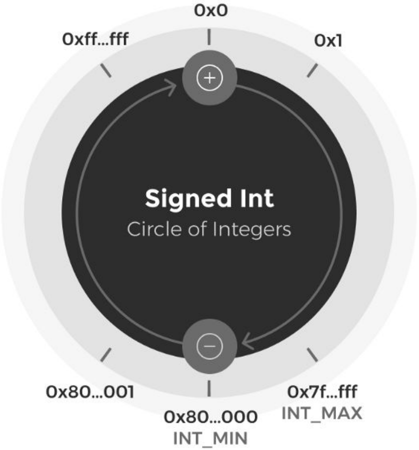
- **Integer overflows** behave differently from stack buffer overflows
  - Really just integer "**wrap-arounds**"

32-bit Integer Range:

Unsigned: [0, (2^32 - 1)]
[0, 4294967295]

Signed: [-2^31, (2^31 - 1)]
[-2147483648, 2147483647]





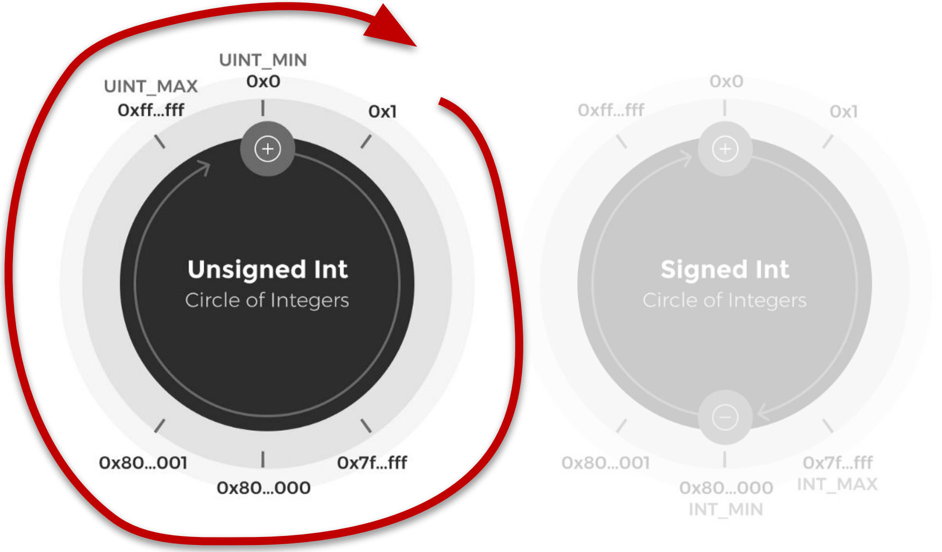
Integer Overflows

- **Integer overflows** behave differently from stack buffer overflows
  - Really just integer "**wrap-arounds**"

32-bit Integer Range:

Unsigned: `[0, (2^32 - 1)]`
`[0, 4294967295]`

Signed: `[-2^31, (2^31 - 1)]`
`[-2147483648, 2147483647]`



- **Overflowing an unsigned integer "wraps around" to a very small integer!**
  - E.g., `0xFFFFFFFF + 2 = 0x00000002`

Example Integer Overflow

- **What is unsafe about this code?**

```
void foo(char *array, int len)
{
    int buf[100];

    if(len >= 100) {
        return;
    }

    memcpy(buf, array, len);
}
```

Example Integer Overflow

- **What is unsafe about this code?**

```
void foo(char *array, int len)
{
    int buf[100];

    if(len >= 100) {
        return;
    }

    memcpy(buf, array, len);
}
```

```
void *memcpy (void *dest,
const void *src, size_t n);
```

Example Integer Overflow

- **What is unsafe about this code?**

```
void foo(char *array, int len)
{
    int buf[100];

    if(len >= 100) {
        return;
    }

    memcpy(buf, array, len);
}
```

```
void *memcpy (void *dest,
const void *src, size_t n);
```

`size_t n` must be a **signed int**

Example Integer Overflow

- **What is unsafe about this code?**

```
void foo(char *array, int len)
{
    int buf[100];

    if(len >= 100) {
        return;
    }

    memcpy(buf, array, len);
}
```

```
void *memcpy (void *dest,
const void *src, size_t n);
```

size_t n must be a **signed int**

memcpy interprets a **negative** **len** as a huge unsigned value!

Example Integer Overflow

- **What is unsafe about this code?**

```
void foo(char *array, int len)
{
    int buf[100];

    if(len >= 100) {
        return;
    }

    memcpy(buf, array, len);
}
```

```
void *memcpy (void *dest,
const void *src, size_t n);
```

size_t n must be a **signed int**

memcpy interprets a **negative len** as a huge unsigned value!

OVERFLOW—Copy **way more than 100 bytes** into dst buffer!

Example Integer Overflow

- What is u

```
void fo
{
    int

    if(l

    }

    memo
}
```

d *dest,
ize_t n);

signed int

negative
ned value!

way more
dst buffer!

Overcoming Bounded Writes

- **What observations can we make?**
  - Can they break the program's assumptions?

- **Target 4:** a **potential mismatch** of **buffer's size** versus the **data written to it**

```
alloca( <MAX_UINT );    // allocate our buffer
fread( &buf[i], 4, count, f ); // fill buffer
```

Range of count:

[0, ¼(MAX_UINT))

[0,   MAX_UINT)

  - If we perform an **integer overflow** on count, `alloca()` creates an **artificially small** buffer
  - The resulting fill operation will **exceed the buffer's size**, resulting in a buffer overflow!

Overcoming Bounded Writes

- **What observations can we make?**
  - Can they
- **Target 4:** a … ta written to it

Target 4: a very large **count** will trigger an **integer overflow** in the buffer's allocation, wrapping **MAX_UINT** to a **very small size**.

```
alloca( <MAX_UINT );    // allocate our buffer
fread( &buf[i], 4, count, f ); // fill buffer
```

…ge of count:

[0, ¼(MAX_UINT))

[0, MAX_UINT)

  - If we perform an **integer overflow** on count, alloca() creates an **artificially small** buffer
  - The resulting fill operation will **exceed the buffer's size**, resulting in a buffer overflow!

Overcoming Bounded Writes

Target 4: a very large **`count`** will trigger an **integer overflow** in the buffer's allocation, wrapping **`MAX_UINT`** to a **very small size**.

Since we later write **`count`** **elements** into the buffer, this will trigger a **buffer overflow**… allowing **overwriting of objects** **up the stack**!

Target Reconnaissance

Target	What is our attack's **goal**?	How to **write** up the stack?	How **far** can we write?
0	Overwrite **Variable**	`gets()`	**Unbounded**
1	Redirect to **Function**	`strcpy()`	**Unbounded**
2	Redirect to **Shellcode**	`strcpy()`	**Unbounded**
3	Redirect to **Shellcode**	`strncpy()`	**Bounded**
4	Redirect to **Shellcode**	**Integer Overflow on buf's allocation size**	

Now update your **high-level plan**!

Questions?

D.E.N.N.I.S.

Estimate the stack frame

Estimating the Stack

- **Objective: understand the memory layout**
  - What is needed for our attack to be successful?

- **Fundamental questions to consider:**
  1. What stack objects do we **control**?
  2. What stack objects can we **reach**?
  3. What's our desired **final stack state**?

```
void vulnerable(char *arg)
{
    char buf[100];
    strcpy(buf, arg);
}
```

Estimating the Stack

- **Objective: understand the memory layout**
  - What is needed for our attack to be successful?

- **Fundamental questions to consider:**
  1. What stack objects do we **control**?
     - `char buf[100]`
  2. What stack objects can we **reach**?
     - Everything upwards of `buf`!
  3. What's our desired **final stack state**?
     - Inject our shellcode within our vulnerable buffer `buf`
     - Overwrite `vulnerable()`'s return address with `buf`'s address!

```
void vulnerable(char *arg)
{
    char buf[100];
    strcpy(buf, arg);
}
```

Drawing the Stack: Where to even begin?

- Many of you will try to draw the stack based on the assembly…

```
Dump of assembler code for function vulnerable:
    0x0804a17b <+0>:     endbr32
    0x0804a17f <+4>:     push   %ebp
    0x0804a180 <+5>:     mov    %esp,%ebp
    0x0804a182 <+7>:     push   %ebx
    0x0804a183 <+8>:     sub    $0x74,%esp
    0x0804a186 <+11>:    call   0x804a208 <__x86.get_pc_thunk.ax>
    0x0804a18b <+16>:    add    $0x9fe75,%eax
    0x0804a190 <+21>:    sub    $0x8,%esp
    0x0804a193 <+24>:    pushl  0x8(%ebp)
    0x0804a196 <+27>:    lea    -0x6c(%ebp),%edx
    ...
```

Drawing the Stack: Where to even begin?

- Many of you will try to draw the stack based on the assembly...

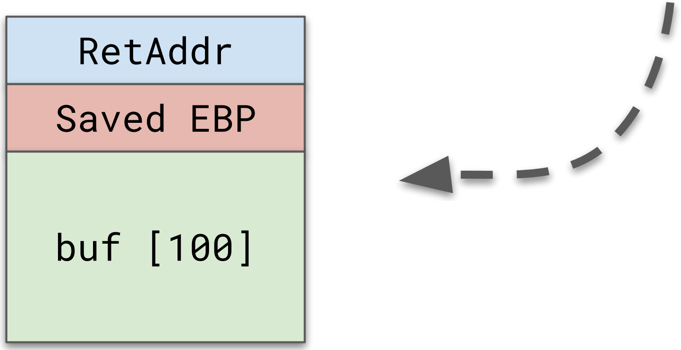
```
Dump of assembler code for function vulnerable:
   0x0804a17b <+0>:     endbr32
   0x0804a17f <+4>:     push   %ebp
   0x0804a180 <+
   0x0804a182 <+
   0x0804a183 <+
   0x0804a186 <+
   0x0804a18b <+
   0x0804a190 <+21>:    sub    $0x8,%esp
   0x0804a193 <+24>:    pushl  0x8(%ebp)
   0x0804a196 <+27>:    lea    -0x6c(%ebp),%edx
   ...
```

Ditch the assembly... draw your stack based on the **source code**!

Drawing the Stack

- **Identify your target function**
  - E.g., `vulnerable()` in this case
- **Each frame contains a few key things:**
  1. The function's return address
     - Address of next instruction to when the current function returns
  2. The caller's saved frame pointer
     - Where EBP will get "reset" to when the current function returns
  3. The function's local variables
     - E.g., `char buf[100]`
     - **Find these from the source code!**

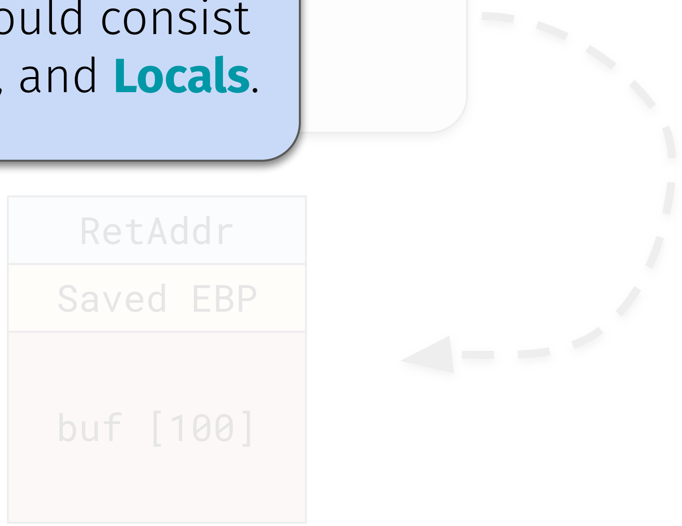
```
void vulnerable(char *arg){
      char buf[100];
      strcpy(buf, arg);
}
```



Drawing the Stack

Your **high-level stack diagram** should consist of the **Return Address**, **Saved EBP**, and **Locals**.

- Identify your target function
  - E.g., vul...
- Each frame...
  1. The function's return address
     - Address of next instruction to when the current function returns
  2. The caller's saved frame pointer
     - Where EBP will get "reset" to when the current function returns
  3. The function's local variables
     - E.g., `char buf[100]`



Drawing the Stack

Your **high-level stack diagram** should consist of the **Return Address**, **Saved EBP**, and **Locals**.

No assembly required—just look at the **source**!

Drawing the Stack

Your **high-level stack diagram** should consist of the **Return Address**, **Saved EBP**, and **Locals**.

No assembly required—just look at the **source**!

You need to get comfortable with this—highly recommended to revisit **All About Applications**

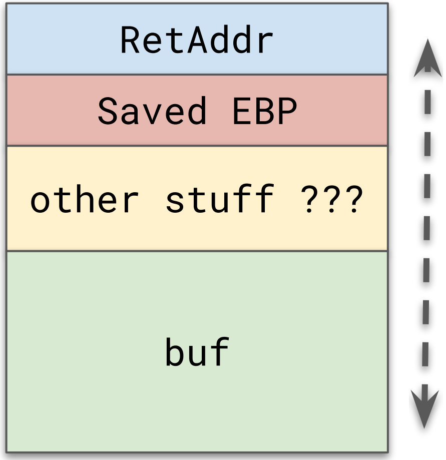
D.E.N.N.I.S.

NOP-out everything inside the frame!

Then, NOP-out just the return address!

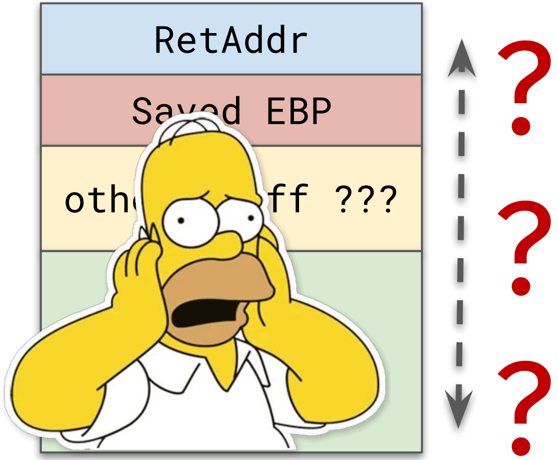
Building your Attack

- **Question:** how to calculate the **exact amount** of overflow to reach the return address?
  - Read the assembly code line by line
  - Revisit and tweak your stack diagram
  - If it doesn't work, go back and look at more assembly



Building your Attack

- **Question:** how to calculate the **exact amount** of overflow to reach the return address?
  - Read the assembly code line by line
  - Revisit and tweak your stack diagram
  - If it doesn't work, go back and look at more assembly
- **Don't do this—you will go insane reading x86**



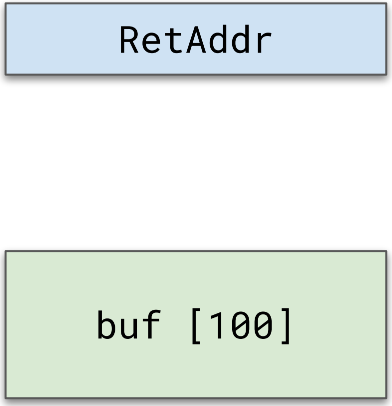
Building your Attack

- **Question:** how to calculate the **exact amount** of overflow to reach the return address?
  - Read the assembly code line by line
  - Revisit and tweak your stack diagram
  - If it doesn't work, go back and look at more assembly
- **Don't do this**—**you will go insane reading x86**

Ditch the assembly... guesstimate your padding with a few **heuristics**!

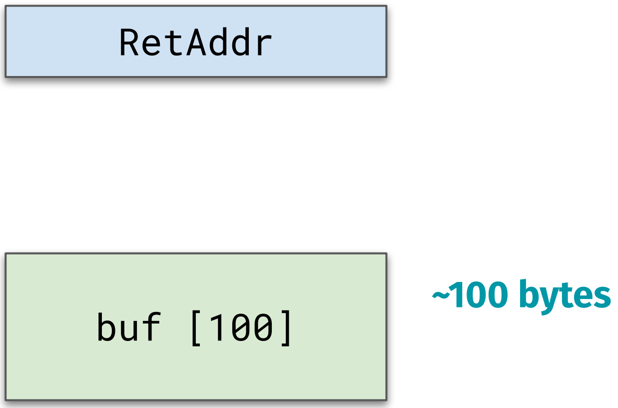
Padding Heuristics

- **How large** is our vulnerable buffer?
  - E.g., char `buf[100]`



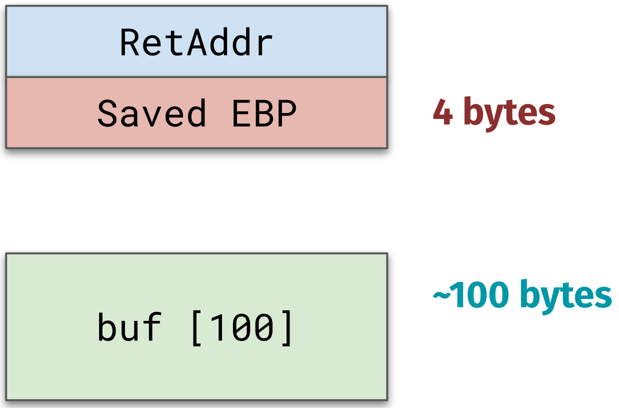
Padding Heuristics

- **How large** is our vulnerable buffer?
  - E.g., char `buf[100]`
  - Need **at least 100 bytes** to overflow!
    - Compilers may add **a few "extra" bytes** for memory alignment



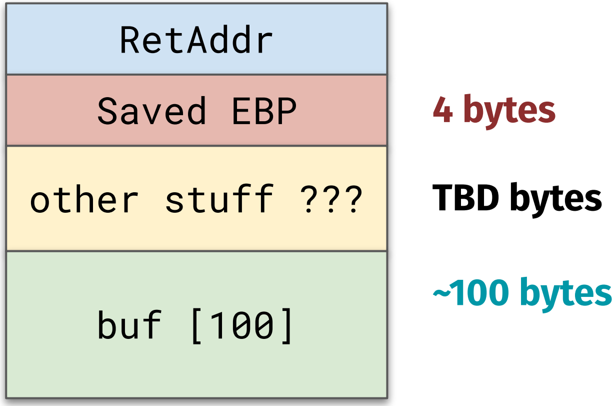
Padding Heuristics

- **How large** is our vulnerable buffer?
  - E.g., char `buf[100]`
  - Need **at least 100 bytes** to overflow!
    - Compilers may add **a few "extra" bytes** for memory alignment
- **Saved EBP** = an extra **four bytes**



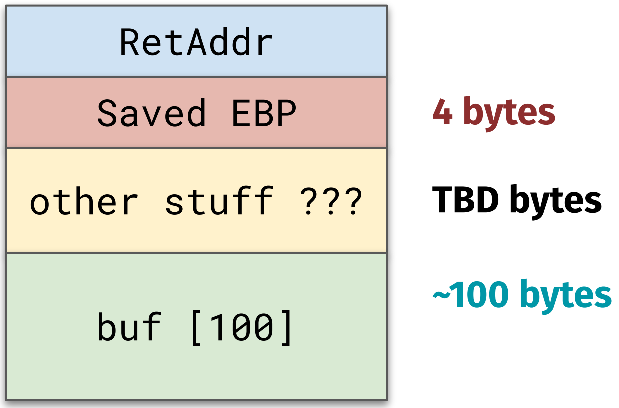
Padding Heuristics

- **How large** is our vulnerable buffer?
  - E.g., char `buf[100]`
  - Need **at least 100 bytes** to overflow!
    - Compilers may add **a few "extra" bytes** for memory alignment
- **Saved EBP** = an extra **four bytes**
- **Other things above our buffer?**
  - Other locals (e.g., `count` in Target 3)
  - Passed-by-reference function args
  - Other compiler-added artifacts



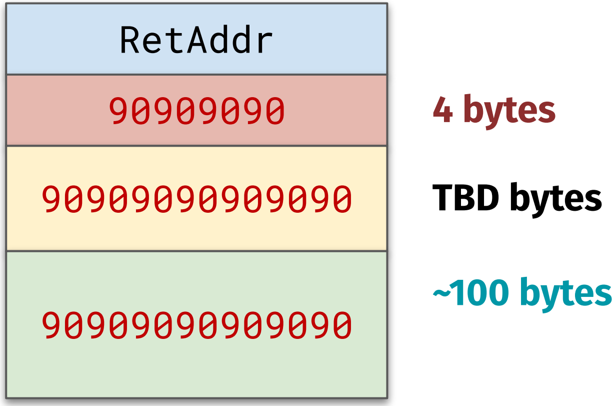
Write an Initial Payload

- Use guesstimated payload bytes as **lower bound** for an initial attempt
  - E.g., we know our payload is **104+ bytes**



Write an Initial Payload

- Use guesstimated payload bytes as **lower bound** for an initial attempt
  - E.g., we know our payload is **104+ bytes**
- **Goal:** overwrite the return address with a **controlled**, **friendly** **payload**
  - E.g., **104 bytes** of NOP instructions
- **Did it overwrite the return address?**
  - If **yes**—`SEGFAULT` on `0x90909090`
  - If **not**—program terminates gracefully



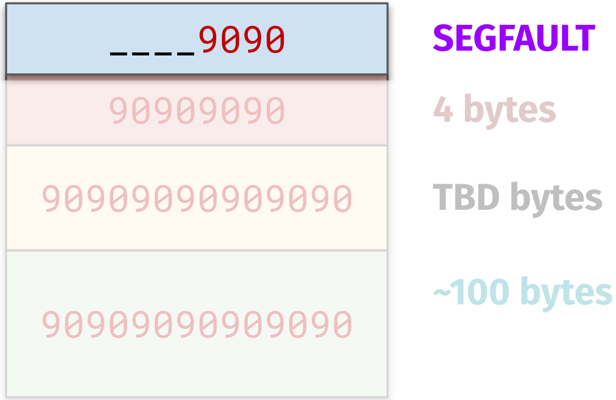
Write an Initial Payload

- Use guesstimated payload bytes as **lower bound** for an initial attempt
  - E.g., we know our payload is **104+ bytes**
- **Goal:** overwrite the return address with a **controlled**, **friendly** **payload**
  - E.g., **104 bytes** of NOP instructions
- **Did it overwrite the return address?**
  - If **yes**—**SEGFAULT** on `0x90909090`
  - If **not**—program terminates gracefully

Keep **increasing until** program **SEGFAULT**

Refine your Payload

- **Keep a table** of attempts and results
  1. `b'\x90' * 104 → normal exit`
     - **Too little!** Didn't overwrite anything
  2. `b'\x90' * 120 → SEGV on 0x90909090`
     - **Too much!** Complete RetAddr overwrite
  3. `b'\x90' * 114 → SEGV on 0x08049090`
     - **We're close—just two bytes over!**
     - Our payload should be **112 bytes**



Tweak it to figure out the **exact payload size**

Refine your Payload

- **Keep a table** of attempts and results
  1. b'\x90' * 104 → normal exit
     - Too
  2. b'\x90'
     - Too
  3. b'\x90'
     - We
     - Ou

____9090	SEGFAULT
	4 bytes
	TBD bytes
	~100 bytes

Tweak it to figure out the **exact payload size**

SEGFAULTS are your friend—they indicate you're **on the right track** (overwriting things)!

Use them and **iteratively refine** your payload!

D.E.N.N.I.S.

Inspect the program's memory

Find the Buffer!

- After finding the distance to the return address, we now must **overwrite it**
  - **Recall:** the return address is our golden ticket to **controlling the program's execution**
  - Instead of a normal return, we want to **redirect execution** to our **shellcode-laden buffer**

Find the Buffer!

- After finding the distance to the return address, we now must **overwrite it**
  - **Recall:** the return address is our golden ticket to **controlling the program's execution**
  - Instead of a normal return, we want to **redirect execution** to our **shellcode-laden buffer**

- **Approach:** pick a **known, friendly payload** and locate it in memory
  - Goal is to find **the start of your buffer!**

Find the Buffer!

- After finding the distance to the return address, we now must **overwrite it**
  - **Recall:** the return address is our golden ticket to **controlling the program's execution**
  - Instead of a normal return, we want to **redirect execution** to our **shellcode-laden buffer**

- **Approach:** pick a **known, friendly payload** and locate it in memory
  - Goal is to find **the start of your buffer!**

- **Helpful GDB commands:**
  - `info proc mapping`
    - Locate the stack's **boundaries**
    - E.g., `0xfff6d000` to `0xffffe000`

```
$ info proc mapping // list all memory segments
Start Addr   End Addr       Size     Offset objfile
0x8048000  0x8049000     0x1000        0x0 target2
0x8049000  0x80b8000    0x6f000     0x1000 target2
0x80b8000  0x80e8000    0x30000    0x70000 target2
0x80e8000  0x80ea000     0x2000    0x9f000 target2
0x80ea000  0x80ec000     0x2000    0xa1000 target2
0x80ec000  0x810e000    0x22000        0x0 [heap]
0xf7ff8000 0xf7ffc000     0x4000        0x0 [vvar]
0xf7ffc000 0xf7ffe000     0x2000        0x0 [vdso]
0xfff6d000 0xffffe000    0x91000        0x0 [stack]
```

Find the Buffer!

- After finding the distance to the return address, we now must **overwrite it**
  - **Recall:** the return address is our golden ticket to **controlling the program's execution**
  - Instead of a normal return, we want to **redirect execution** to our **shellcode-laden buffer**

- **Approach:** pick a **known, friendly payload** and locate it in memory
  - Goal is to find **the start of your buffer!**

- **Helpful GDB commands:**
  - `find minAddr,maxAddr,"string"`
    - Search memory for address of `string`
    - Use **stack boundaries** from before

```
$ b *vulnerable+45 // breakpoint after buf filled
Breakpoint 1, 0x0804a1a8 in vulnerable… target2.c:8

$ r "AAAA" // run program with "AAAA" as its input
Breakpoint 1, 0x0804a1a8 in vulnerable… target2.c:8

$ find 0xfff6d000,0xffffe000,"AAAA"
0xfff6d8cc // this is likely where buffer begins!
0xfffed930 // when in doubt, pick the lower address
```

Find the Buffer!

- After finding the distance to the return address, we now must **overwrite it**
  - **Recall:** the return address is our golden ticket to **controlling the program's execution**
  - Instead of a normal return, we want to **redirect execution** to our **shellcode-laden buffer**

- **Approach:** pick a **known, friendly payload** and locate it in memory
  - Goal is to find **the start of your buffer!**

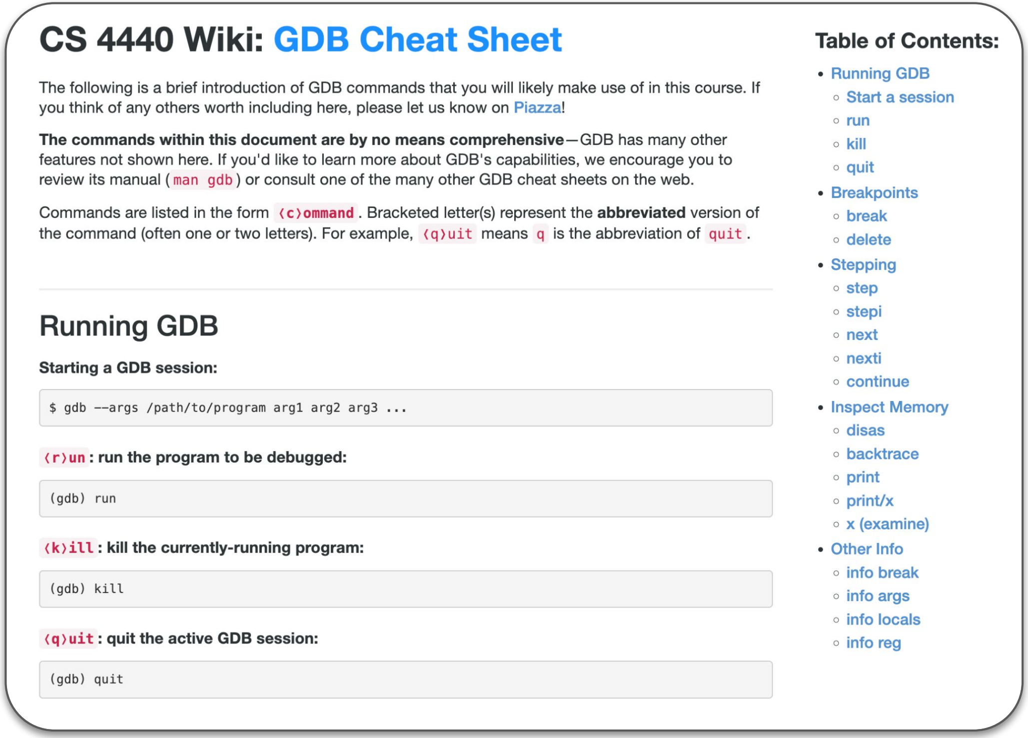
- **Helpful GDB commands:**
  - `x/32xw, 0xDEADBEEF`
    - Show bytes at address `0xDEADBEEF`
    - **Inspect candidates** from previous step

```
$ b *vulnerable+45 // breakpoint after buf filled
Breakpoint 1, 0x0804a1a8 in vulnerable… target2.c:8

$ r “AAAA” // run program with “AAAA” as its input
Breakpoint 1, 0x0804a1a8 in vulnerable… target2.c:8

$ x/32xw 0xfff6d8cc // look for “AAAA” bytes here
0xfff6d8cc: 0x41414141 0x00000000 0x00000000 ...
0xfff6d8d0: 0x00000000 0x00000000 0x00000000 ...
```

Other GDB Resources

- Other GDB resources:
  - [CS 4440 GDB Cheat Sheet]
  - [Beej's GDB Tutorial]
  - [Tudor's GDB Tutorial]

- Many others on the web!



CS 4440 Wiki: GDB Cheat Sheet

The following is a brief introduction of GDB commands that you will likely make use of in this course. If you think of any others worth including here, please let us know on Piazza!

The commands within this document are by no means comprehensive—GDB has many other features not shown here. If you'd like to learn more about GDB's capabilities, we encourage you to review its manual (`man gdb`) or consult one of the many other GDB cheat sheets on the web.

Commands are listed in the form `(c)ommand`. Bracketed letter(s) represent the **abbreviated** version of the command (often one or two letters). For example, `(q)uit` means `q` is the abbreviation of `quit`.

Running GDB

Starting a GDB session:

```
$ gdb --args /path/to/program arg1 arg2 arg3 ...
```

`(r)un`: **run the program to be debugged:**

```
(gdb) run
```

`(k)ill`: **kill the currently-running program:**

```
(gdb) kill
```

`(q)uit`: **quit the active GDB session:**

```
(gdb) quit
```

Table of Contents:

Experience with GDB?

None (that's totally okay!) 0%

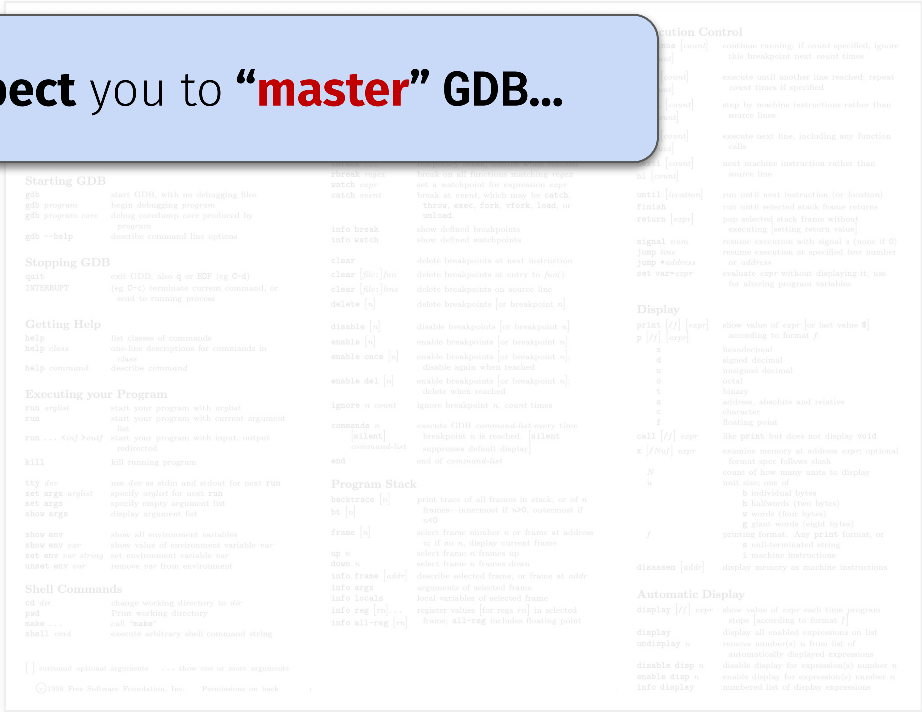
Some 0%

Lots! 0%

Not with GDB, but other debuggers 0%

Other GDB Resources

- Other GDB
  - CS 4440
  - Beej's GD
  - Tudor's GDB Tutorial


Starting GDB
gdb start GDB, with no debugging files
gdb program begin debugging program
gdb program core debug coredump core produced by program
gdb --help describe command line options

Stopping GDB
quit exit GDB; also q or EOF (eg C-d)
INTERRUPT (eg C-c) terminate current command, or send to running process

Getting Help
help list classes of commands
help class one-line descriptions for commands in class
help command describe command

Executing your Program
run arglist start your program with arglist
run start your program with current argument list
run ... <inf >outf start your program with input, output redirected
kill kill running program
tty dev use dev as stdin and stdout for next run
set args arglist specify arglist for next run
set args specify empty argument list
show args display argument list
show env show all environment variables
show env var show value of environment variable var
set env var string set environment variable var
unset env var remove var from environment

Shell Commands
cd dir change working directory to dir
pwd Print working directory
make ... call "make"
shell cmd execute arbitrary shell command string

[ ] surround optional arguments ... show one or more arguments
©1998 Free Software Foundation, Inc. Permissions on back

rbreak regex break on all functions matching regex
watch expr set a watchpoint for expression expr
catch event break at event, which may be catch, throw, exec, fork, vfork, load, or unload.
info break show defined breakpoints
info watch show defined watchpoints
clear delete breakpoints at next instruction
clear [file:]fun delete breakpoints at entry to fun()
clear [file:]line delete breakpoints on source line
delete [n] delete breakpoints [or breakpoint n]
disable [n] disable breakpoints [or breakpoint n]
enable [n] enable breakpoints [or breakpoint n]
enable once [n] enable breakpoints [or breakpoint n]; disable again when reached
enable del [n] enable breakpoints [or breakpoint n]; delete when reached
ignore n count ignore breakpoint n, count times
commands n [silent] command-list execute GDB command-list every time breakpoint n is reached. [silent suppresses default display]
end end of command-list

Program Stack
backtrace [n] / bt [n] print trace of all frames in stack; or of n frames—innermost if n>0, outermost if n<0
frame [n] select frame number n or frame at address n; if no n, display current frame
up n select frame n frames up
down n select frame n frames down
info frame [addr] describe selected frame, or frame at addr
info args arguments of selected frame
info locals local variables of selected frame
info reg [rn]... register values [for regs rn] in selected frame; all-reg includes floating point
info all-reg [rn]

Execution Control
continue [count] continue running; if count specified, ignore this breakpoint next count times
step [count] execute until another line reached; repeat count times if specified
stepi [count] step by machine instructions rather than source lines
next [count] execute next line, including any function calls
nexti [count] / ni [count] next machine instruction rather than source line
until [location] run until next instruction (or location)
finish run until selected stack frame returns
return [expr] pop selected stack frame without executing [setting return value]
signal num resume execution with signal s (none if 0)
jump line resume execution at specified line number
jump *address or address
set var=expr evaluate expr without displaying it; use for altering program variables

Display
print [/f] [expr] / p [/f] [expr] show value of expr [or last value $] according to format f:
x hexadecimal
d signed decimal
u unsigned decimal
o octal
t binary
a address, absolute and relative
c character
f floating point
call [/f] expr like print but does not display void
x [/Nuf] expr examine memory at address expr; optional format spec follows slash
N count of how many units to display
u unit size; one of
b individual bytes
h halfwords (two bytes)
w words (four bytes)
g giant words (eight bytes)
f printing format. Any print format, or
s null-terminated string
i machine instructions
disassem [addr] display memory as machine instructions

Automatic Display
display [/f] expr show value of expr each time program stops [according to format f]
display display all enabled expressions on list
undisplay n remove number(s) n from list of automatically displayed expressions
disable disp n disable display for expression(s) number n
enable disp n enable display for expression(s) number n
info display numbered list of display expressions

We do NOT expect you to “master” GDB…

Other GDB Resources

- Other GDB r
  - CS 4440
  - Beej's GD
  - Tudor's GDB Tutorial

We do NOT expect you to “**master**” GDB…

However, you should **keep a link or two** handy for quick referencing. **See the CS 4440 Wiki!**

Other GDB Resources

- Other GDB r
  - CS 4440 (
  - Beej's GD
  - Tudor's GDB Tutorial

We do NOT expect you to **"master"** GDB...

However, you should **keep a link or two** handy for quick referencing. **See the CS 4440 Wiki!**

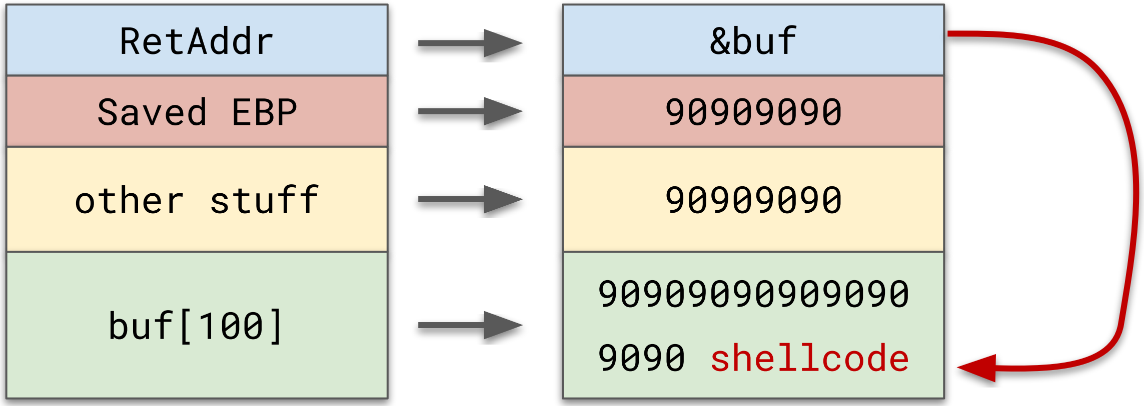
You will definitely be faced with GDB-style debugging scenarios in your future careers!

D.E.N.N.I.S.

Setup and stabilize your attack!

We're almost there!

- **By this point**, we've identified our **padding length** and **buffer start address**
  - Now, introduce our **shellcode** and finalize the attack payload!



Troubleshooting

- E.g., “My attack **segfaults** and I don’t know why!”

- **Check your padding!**
  - Are you correctly overwriting the return address?

- **Check your payload order!**
  - If **shellcode first**, you must jump to buffer’s **exact start**!
  - If **NOPs first**, you can jump **anywhere** in the NOP slide!

- **Check your destination!**
  - Perform memory inspection to look for **known**, **friendly** payloads
  - Be sure to set breakpoints on a location **after the buffer is filled**!

Troubleshooting

- E.g., “My attack **segfaults** and I don’t know why!”

Most troubleshooting requires just a little **trial and error!**

Look for signs of progress (e.g., **overwriting** stack objects), and **test** whether your payload tweaks **changes things**!

  - Perform memory inspection to look for **known**, **friendly** payloads
  - Be sure to set breakpoints on a location **after the buffer is filled**!

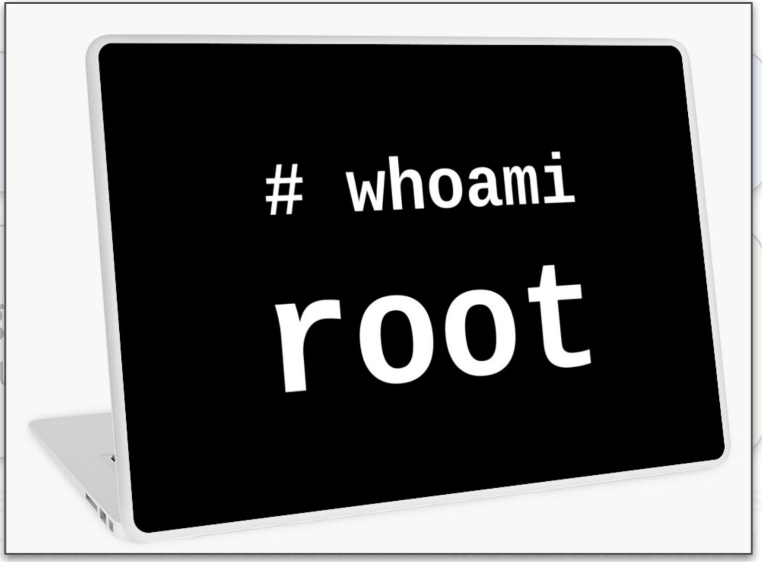
Troubleshooting

- E.g., "My attack **segfaults** and I don't know why!"

Most troubleshooting

Look for signs of prog
and **test** whether you

- Perform memory in
- Be sure to set break



Questions?

Next time on CS 4440…

Defending Applications

And beating those defenses!