FA2022 Week 14

PWN II

Kevin

Announcements

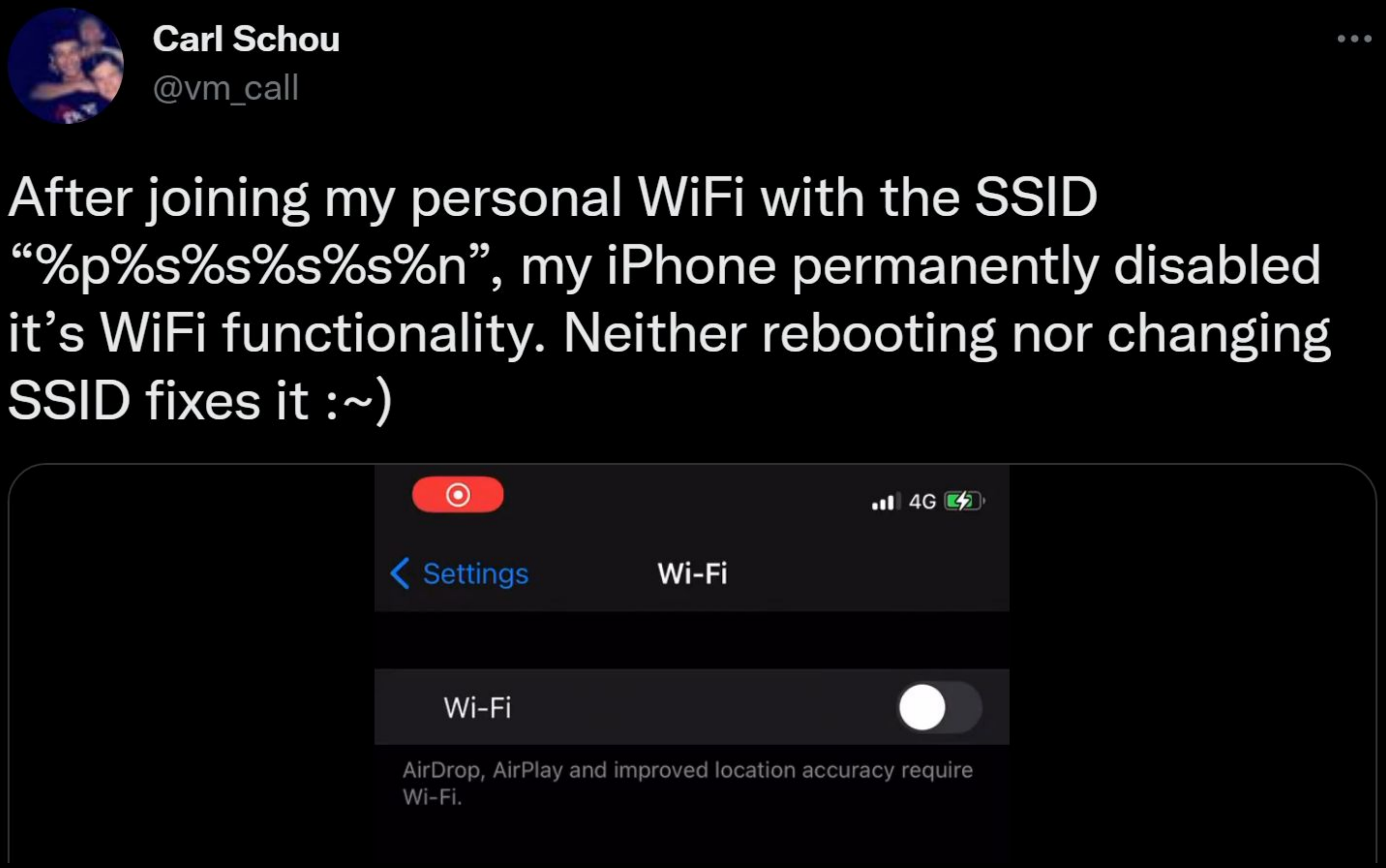
- HackTheBox University starts tomorrow!

- Sunday - grad talk

- Next Thursday is our last meeting!

ctf.sigpwny.com

sigpwny{%n}



Carl Schou
@vm_call

After joining my personal WiFi with the SSID “%p%s%s%s%s%n”, my iPhone permanently disabled it’s WiFi functionality. Neither rebooting nor changing SSID fixes it :~)

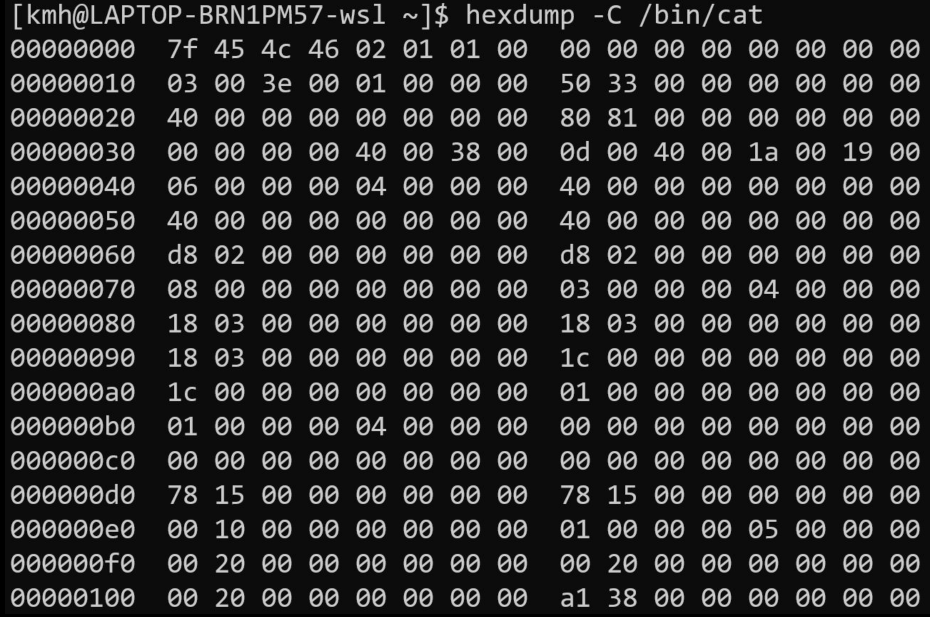
4G
Settings
Wi-Fi
Wi-Fi
AirDrop, AirPlay and improved location accuracy require Wi-Fi.

Review: what is pwn?

- More descriptive term: **binary exploitation**
- Exploits that abuse the mechanisms behind how compiled code is executed
    - Dealing with what the CPU actually sees and executes on or near the hardware level
- Most modern weaponized/valuable exploits fall under this category
- This is real stuff!!
    - Corollary: this is hard stuff. Ask for help, or if you don’t need help, help your neighbors :)

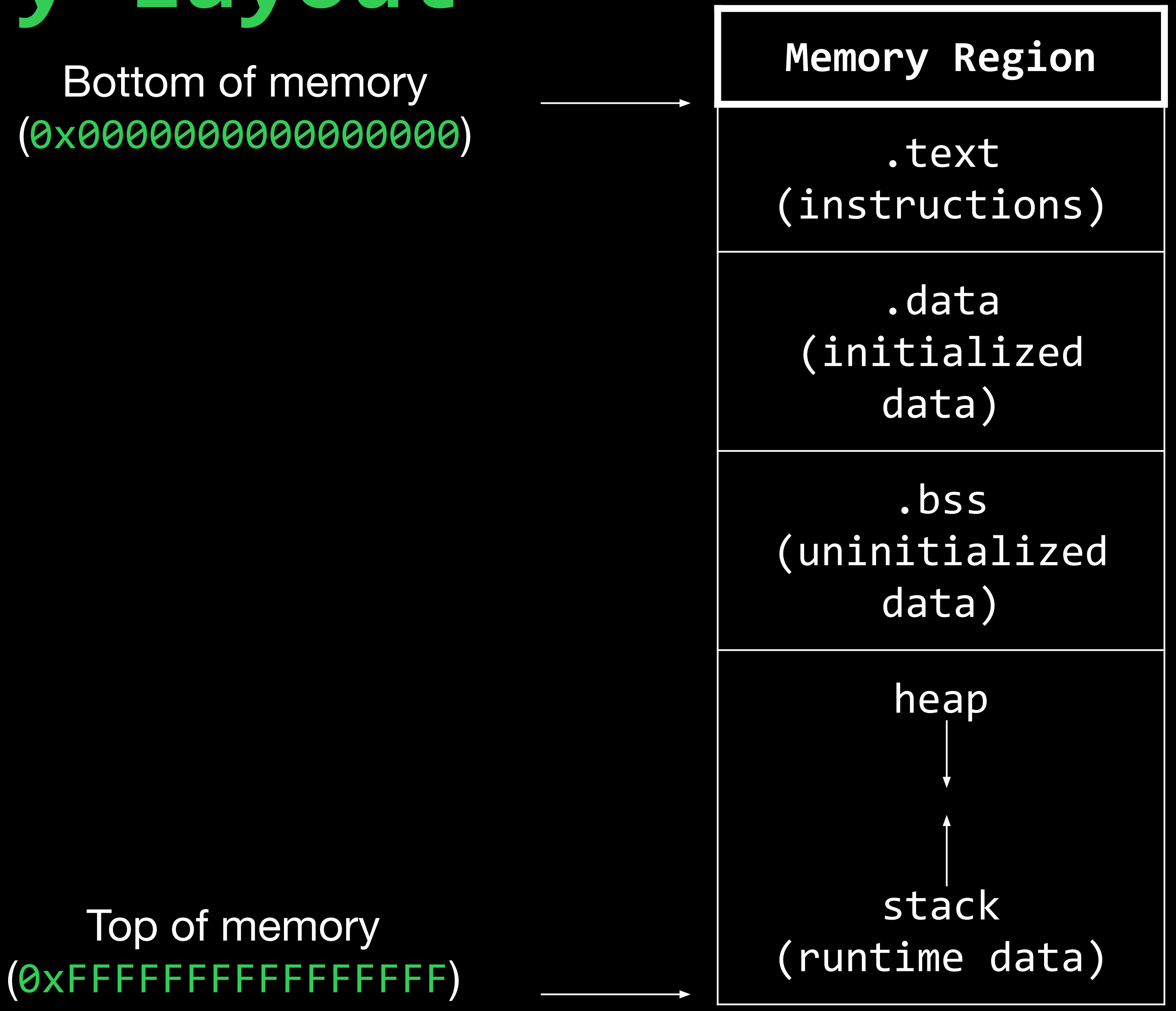
Memory Overview

- Programs are just a bunch of numbers ranging from 0 to 255 (**bytes**)
- Each number is stored at an "address" in the range `0x0-0xFFFFFFFFFFFFFFFF`
  - Think of it as a massive array/list
- Bytes in a program serves one of two purposes
  - **Instructions**: tells the processor what to do
  - **Data**: has some special meaning, used by the instructions
    - Examples: part of a larger number, a letter, a memory address



```
[kmh@LAPTOP-BRN1PM57-wsl ~]$ hexdump -C /bin/cat
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00
00000010  03 00 3e 00 01 00 00 00  50 33 00 00 00 00 00 00
00000020  40 00 00 00 00 00 00 00  80 81 00 00 00 00 00 00
00000030  00 00 00 00 40 00 38 00  0d 00 40 00 1a 00 19 00
00000040  06 00 00 00 04 00 00 00  40 00 00 00 00 00 00 00
00000050  40 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00
00000060  d8 02 00 00 00 00 00 00  d8 02 00 00 00 00 00 00
00000070  08 00 00 00 00 00 00 00  03 00 00 00 04 00 00 00
00000080  18 03 00 00 00 00 00 00  18 03 00 00 00 00 00 00
00000090  18 03 00 00 00 00 00 00  1c 00 00 00 00 00 00 00
000000a0  1c 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00
000000b0  01 00 00 00 04 00 00 00  00 00 00 00 00 00 00 00
000000c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000000d0  78 15 00 00 00 00 00 00  78 15 00 00 00 00 00 00
000000e0  00 10 00 00 00 00 00 00  01 00 00 00 05 00 00 00
000000f0  00 20 00 00 00 00 00 00  00 20 00 00 00 00 00 00
00000100  00 20 00 00 00 00 00 00  a1 38 00 00 00 00 00 00
```

Memory Layout



Bottom of memory
(0x0000000000000000)

Memory Region

.text
(instructions)

.data
(initialized data)

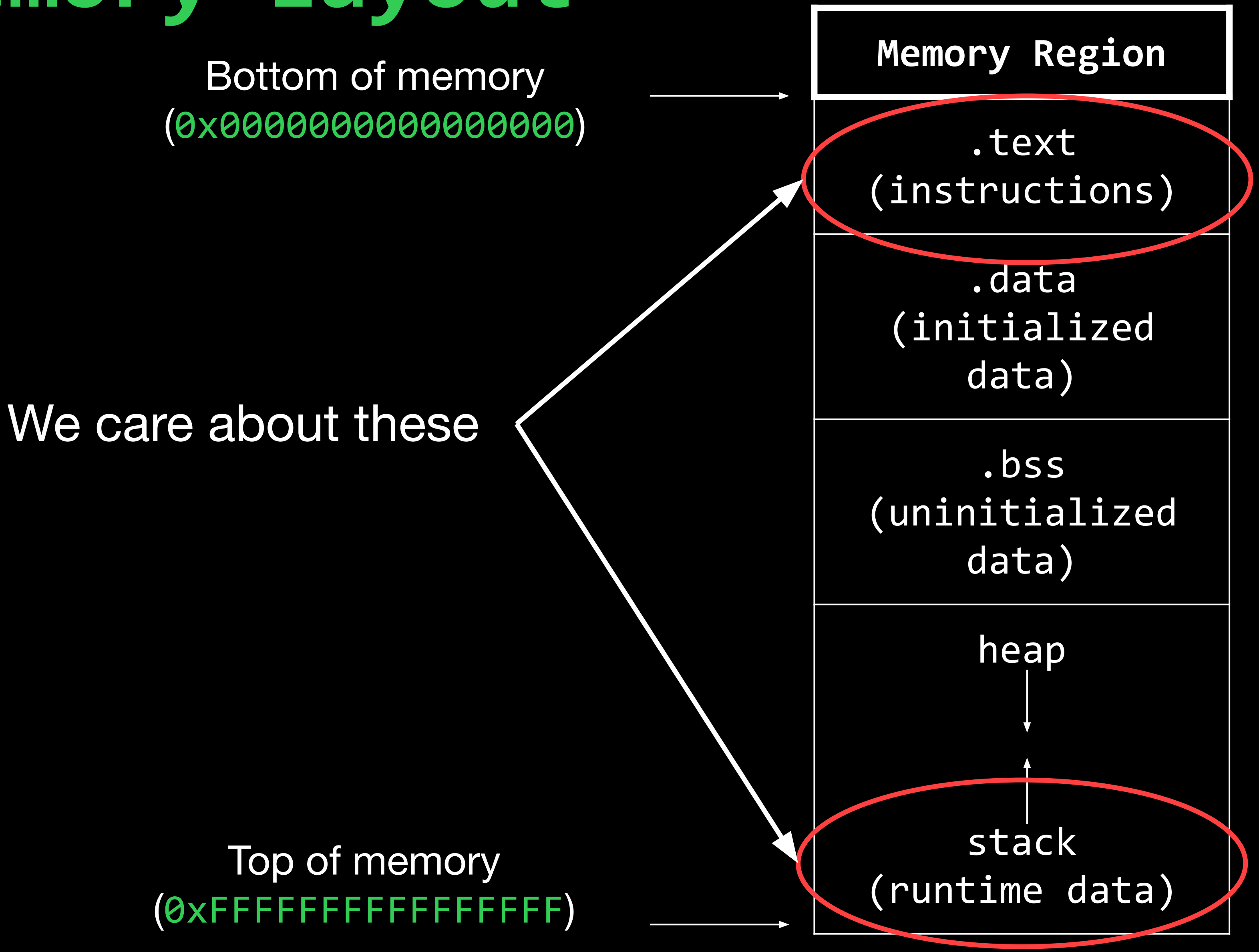
.bss
(uninitialized data)

heap

stack
(runtime data)

Top of memory
(0xFFFFFFFFFFFFFFFF)

Memory Layout



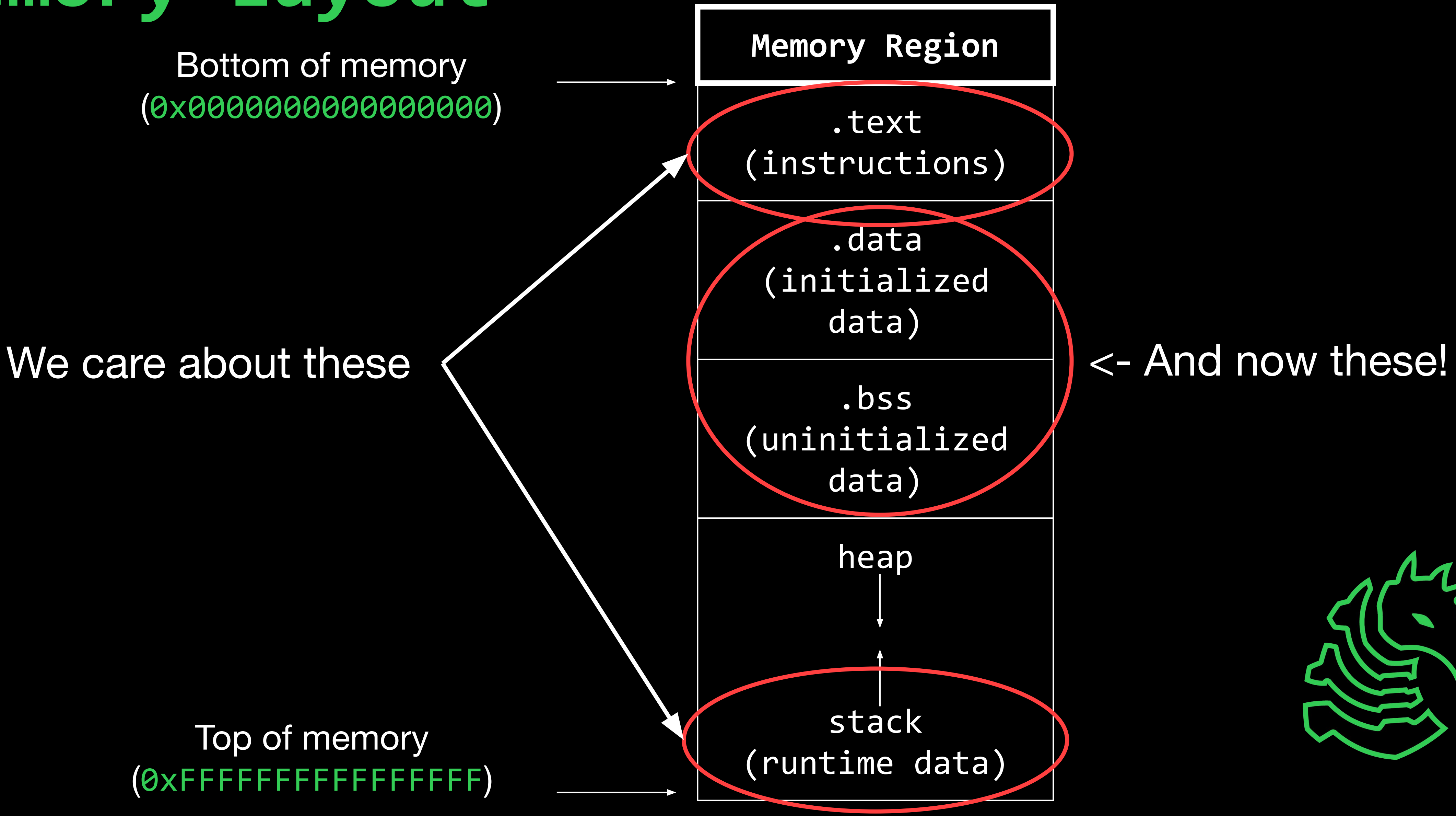
Bottom of memory
(0x0000000000000000)

We care about these

Top of memory
(0xFFFFFFFFFFFFFFFF)

Memory Region
.text (instructions)
.data (initialized data)
.bss (uninitialized data)
heap
stack (runtime data)

Memory Layout



Bottom of memory (0x0000000000000000)

Memory Region

.text (instructions)

.data (initialized data)

.bss (uninitialized data)

heap

stack (runtime data)

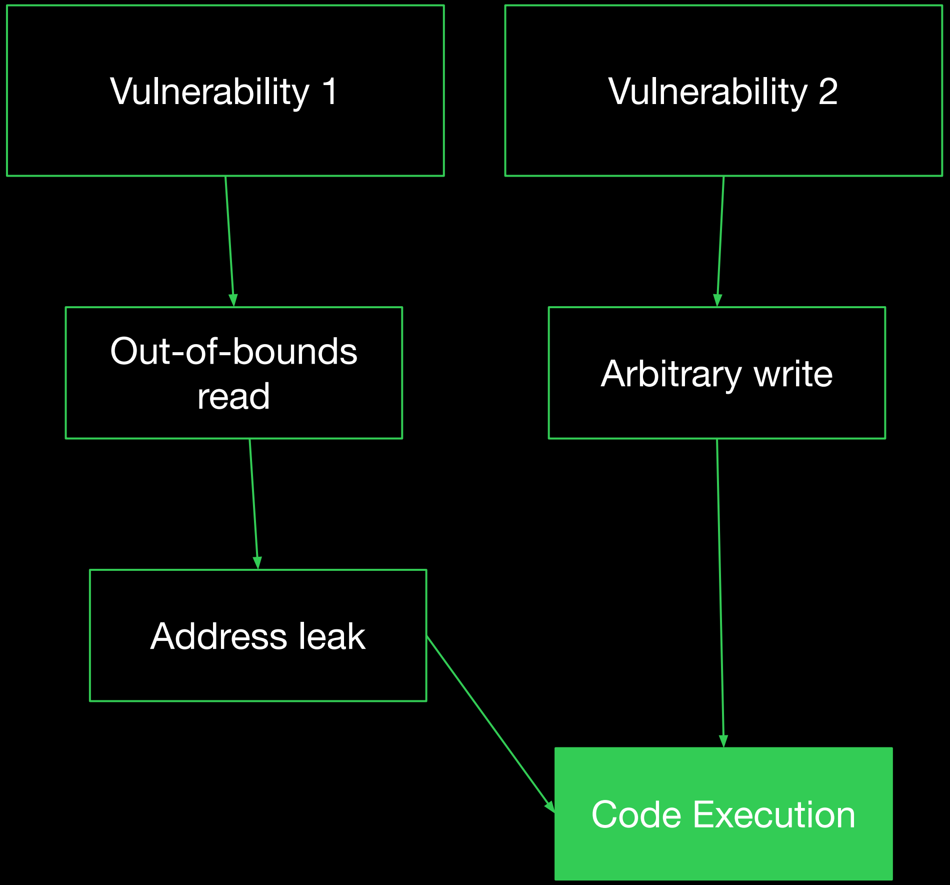
Top of memory (0xFFFFFFFFFFFFFFFF)

We care about these

<- And now these!

Exploit Primitives

- "Building blocks" of an exploit
- Common primitives
  - Read
    - Arbitrary read (read from anywhere)
    - Uncontrolled read (read starting from some address)
  - Write
    - Arbitrary write (write anything anywhere)
    - Uncontrolled write (write something anywhere)
    - Also uncontrolled write (write anything somewhere)
  - Leak
    - Usually done with a read, but not always
    - Necessary because addresses are often **randomized**



Dangerous function of the day: printf()

- **Formatted** print function
  - printf("Hello %s!", "Kevin"); // prints ‘Hello Kevin!’
  - printf("My favorite number is %d", 1337);
    - ‘My favorite number is 1337’
  - printf("%s, my favorite number is %d", "Kevin", 1337);
    - ‘Kevin, my favorite number is 1337’
  - %s and %d are **format specifiers**
    - Tells the function to read the next argument as a certain data type
      - %s -> string, %d -> decimal integer, %p -> pointer, etc.
- What if it’s just used as a print function?
  - printf(name) // name is controlled by the user
  - If name is ‘Kevin’, prints ‘Kevin’

Dangerous function of the day: printf()

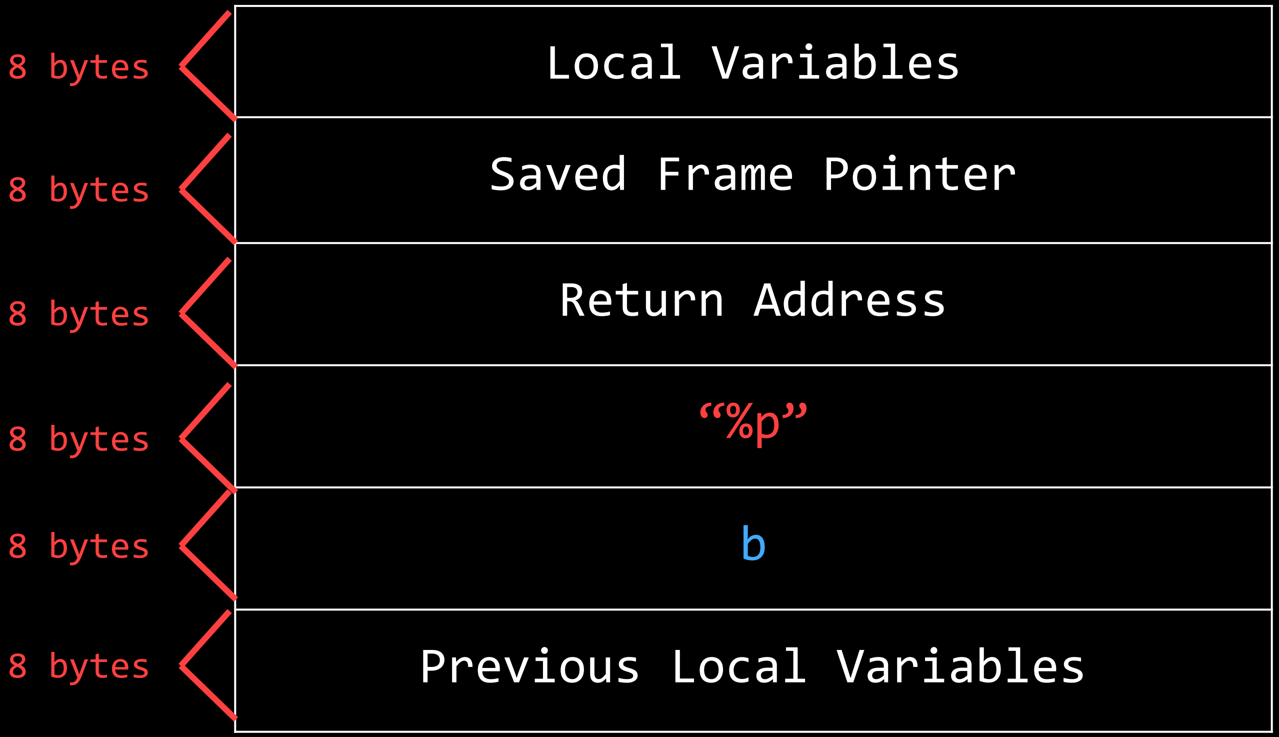
- **Formatted** print function
  - `printf("Hello %s!", "Kevin"); // prints ‘Hello Kevin!’`
  - `printf("My favorite number is %d", 1337);`
    - `‘My favorite number is 1337’`
  - `printf("%s, my favorite number is %d", "Kevin", 1337);`
    - `‘Kevin, my favorite number is 1337’`
  - `%s` and `%d` are **format specifiers**
    - Tells the function to read the next argument as a certain data type
      - `%s` -> string, `%d` -> decimal integer, `%p` -> pointer, etc.
- What if it’s just used as a print function?
  - `printf(name) // name is controlled by the user`
  - If name is ‘%s’, prints…

Primitive: Stack Read

- `%p` format specifier
  - `printf("%p", 0x13371337);`
    - Prints ‘`0x13371337`’
- `printf("%p");`

Review: The Stack

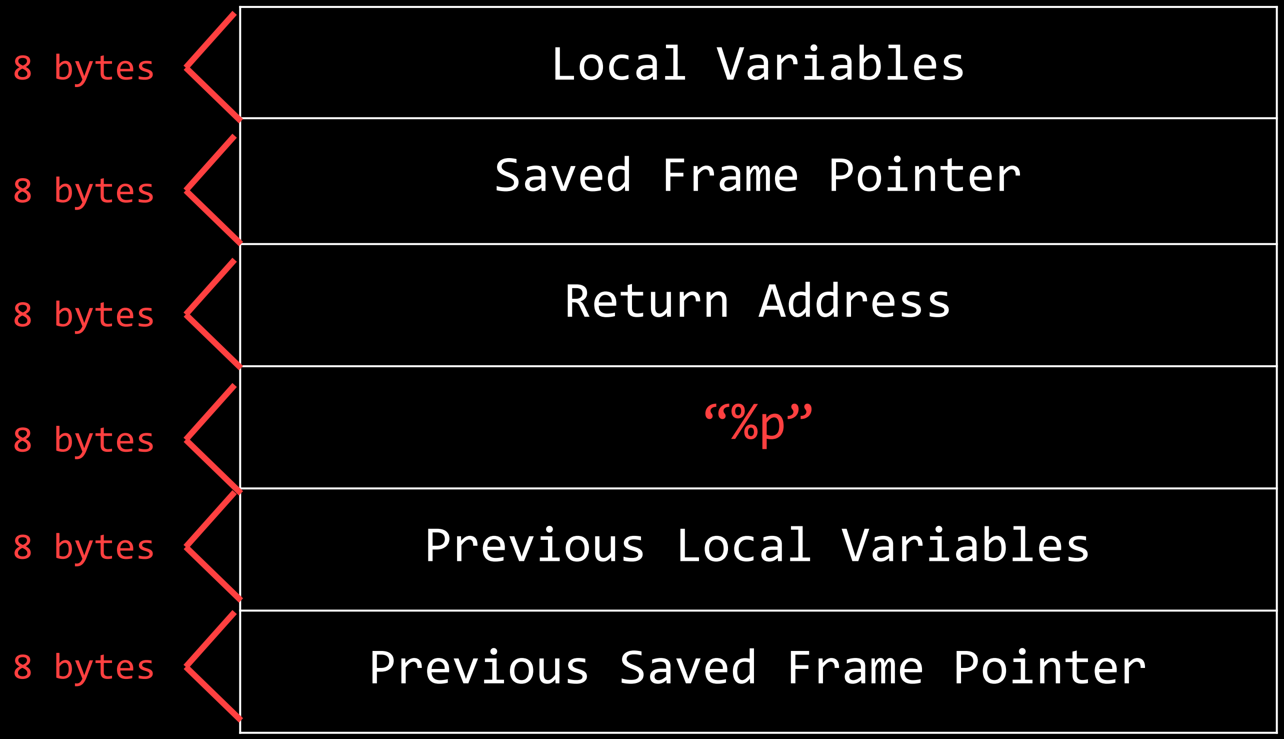
```
printf("%p", b);
```



8 bytes	Local Variables
8 bytes	Saved Frame Pointer
8 bytes	Return Address
8 bytes	"%p"
8 bytes	b
8 bytes	Previous Local Variables

Review: The Stack

```
printf("%p");
```



8 bytes	Local Variables
8 bytes	Saved Frame Pointer
8 bytes	Return Address
8 bytes	"%p"
8 bytes	Previous Local Variables
8 bytes	Previous Saved Frame Pointer

Primitive: Stack Read

- `%p` format specifier
  - `printf("%p", 0x13371337);`
    - Prints ‘`0x13371337`’
- `printf("%p");`
  - Whatever is next on the stack!
  - Send a lot of %p’s and you’ll dump the stack 8 bytes at a time
  - Figure out which data is the thing you want :)
    - If the string ‘sigpwny{’ were on the stack, you might see:
      - 0x7b796e7770676973
      - These are **hexadecimal ASCII values**, online converters may be useful
- Note:
  - %p interprets data as **little endian**

Primitive: Arbitrary Read

- `%s` format specifier
  - `printf("%s", "hello");`
    - Prints ‘hello’
  - `printf("%s", 0x12345678);`
    - Prints the string starting from memory address 0x12345678
  - `printf("%3$s", 0x100, 0x200, 0x300);`
    - Prints the string starting from memory address 0x300 (3rd argument)

Primitive: Arbitrary Read

- `char name[64]; // stored on stack`
- `fgets(name, 64, stdin); // ‘%n$p’ <- n is a number`
- `printf(name);`
- For some n, the %n$p will print name!
  - E.g. 0x70243525
- Key idea:
  - Format specifiers read from the stack, and name is on the stack
  - Format specifiers can reference our input!
- If name is ‘%n$s’ (for correct n)
  - Prints the string starting from a memory address **in our input**

Primitive: Arbitrary Read

- `char name[64]; // stored on stack`
- `fgets(name, 64, stdin);`
- `printf(name);`
- If name is ‘%n$s____\x11\x22\33\x44\x55\x66\x77\x88’ (for correct n)
  - Prints the string starting from memory address 0x8877665544332211
  - We can read from memory addresses contained **in our input**
- Note: why the underscores?
  - Each argument is 8 bytes: len(‘%n$s____’) == 8, so the address is aligned correctly. **Pad to a multiple of 8 bytes before the address.**
- Testing strategy:
  - Develop with %n$p instead of %n$s and verify the correct address gets printed
  - Then switching to %s will make it read from the correct address!

Primitive: Arbitrary Write

- `%n` format specifier
  - Writes the number of bytes previously printed to the given address
  - `printf("%n", &number);`
    - `number = 0;`
  - `printf("AAAA%n", &number);`
    - `number = 4;`
  - `printf("%500p%n", 1, &number);`
    - `number = 500;`
    - ‘%500p’ means format as pointer, padding to 500 characters
      - In this case, ‘0x1’ preceded by 497 spaces
      - Easy way to print a given number of bytes

Primitive: Arbitrary Write

- char name[64]; // stored on stack
- fgets(name, 64, stdin); // ‘%n$p’ <- n is a number
- printf(name);
- If name is ‘%500p%n$n_______\x11\x22\33\x44\x55\x66\x77\x88’ (for correct n)
  - Writes 500 to memory address 0x8877665544332211
- Testing strategy:
  - Same technique as arbitrary read:
    - Develop with %n$p instead of %n$n and verify the correct address is printed
    - Then switching to %n will make it write to the correct address!
- Note: by default, %n writes 4 bytes
  - To write fewer bytes, add h before n to write half the number
    - %hn writes 2 bytes, %hhn writes 1 byte
    - This is important for the challenge!

PIE and Leaks

- **PIE** stands for Position Independent Executable
- **Mitigation** to make exploit development harder
- The binary is loaded into memory at a random address
  - Starts with 0x55 or 0x56, ends with 3 0s (i.e. 0x55xxxxxxx000)
- You will see a PIE address when you read from the stack!
  - Applicable challenges: Leak And Read, Grander Finale
- The addresses output by `objdump` will be offsets from the random base address
  - Find the offset of the original address by grepping for the last 3 digits
  - Subtract that from the leak, and add the offset of the thing you want
- **These challenges will be hard**
  - Ask questions, Google things you don't understand, it will take a while to grasp these concepts!

Global Offset Table and Procedure Linkage Table

- Functions such as gets, printf, and puts are not compiled into the binary
- They are **linked** in another binary called a shared library
- The **PLT** contains stub functions for each linked function
  - These stubs call function addresses stored inside the **GOT**
- To get code execution, overwrite a GOT pointer so that the PLT will call the wrong function!
  - Where to write? Run `readelf -r <binary>`
- **These challenges will be hard**
  - Ask questions, Google things you don’t understand, it will take a while to grasp these concepts!

Delivering Your Exploit

Quirk: Little endianness

- Numbers are little endian in x86-64
  - The least significant ("littlest") byte is stored first
- `0x1122334455667788` is stored in memory as
  `88 77 66 55 44 33 22 11`
  - 88 is the **least significant** because it means 0x88 x 256^0 = 0x88
  - 11 is the **most significant** because it means 0x11 x 256^7 = massive number

Getting function and global variable addresses

With objdump:

Function: `> objdump -d chal | grep "<main>:"`

```
00000000004011ce <main>:
```

Variable: `> objdump -d chal | grep "<flag>"`

```
  401358:       48 8d 3d 61 2d 00 00    lea    0x2d61(%rip),%rdi        # 4040c0 <flag>
```

Or with GDB:

```
> gdb ./chal
> i addr main
Symbol "main" is at 0x4011ce in a file compiled without debugging.
```

echo

- "echoes" your input
- Enable escape codes: `echo -e ...`
  - `\xNN -> 0xNN`
- Can only be used if your exploit is the same every time

```
> echo -e '\x01\x02\x03\x04' | ./chal
```

```
> echo -e '\x01\x02\x03\x04' | nc ...
```

Pwntools

```
from pwn import *

# Connect to Stack 0 server with netcat
conn = remote('chal.sigpwny.com', 1351)

# Read first line
print(conn.recvline())

# Write exploit
conn.sendline('A' * 8)

# Interactive (let user take over)
conn.interactive()
```

```
> python3 -m pip install pwntools
```

Pwntools

```python
from pwn import *
conn = remote(...)

# Address of win function
WIN_ADDR = 0x0804aabb

# Overflow stack
exploit = b'A' * 8

# Push win address after overflow
# p64(number) is a pwntools function that converts the
# number WIN_ADDR to a proper little-endian address
exploit += p64(WIN_ADDR)

# Send exploit
conn.sendline(exploit)
conn.interactive()
```

Next Meetings

2022-12-02 - **Tomorrow**

- HackTheBox University CTF

2022-12-04 - **This Sunday**

- "Human Perceptions and Roles Under Emerging Machine Learning Threats" from grad student Jaron Mink
- Fourth iteration of our research talks with SPRI!

2022-12-08 - **Next Thursday**

- Multiparty Computation with Michael
- Final meeting of the semester!

Challenges!

- Meeting flag:
  - sigpwny{%n}
- Go through the challenge in the PWN II category.
  - The last three are hard and require understanding of GOT/PLT and/or PIE. They will likely require more time to solve than you have during this meeting. Work on them at home, and ask for help in discord :)
- This stuff is confusing, so ask for help
  - If you understand it, help the people around you

SIGPwny