Introduction to Binary Exploitation

THS Hacking Workshops

Andrea Continella

06-12-2021

UNIVERSITY OF TWENTE.

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.

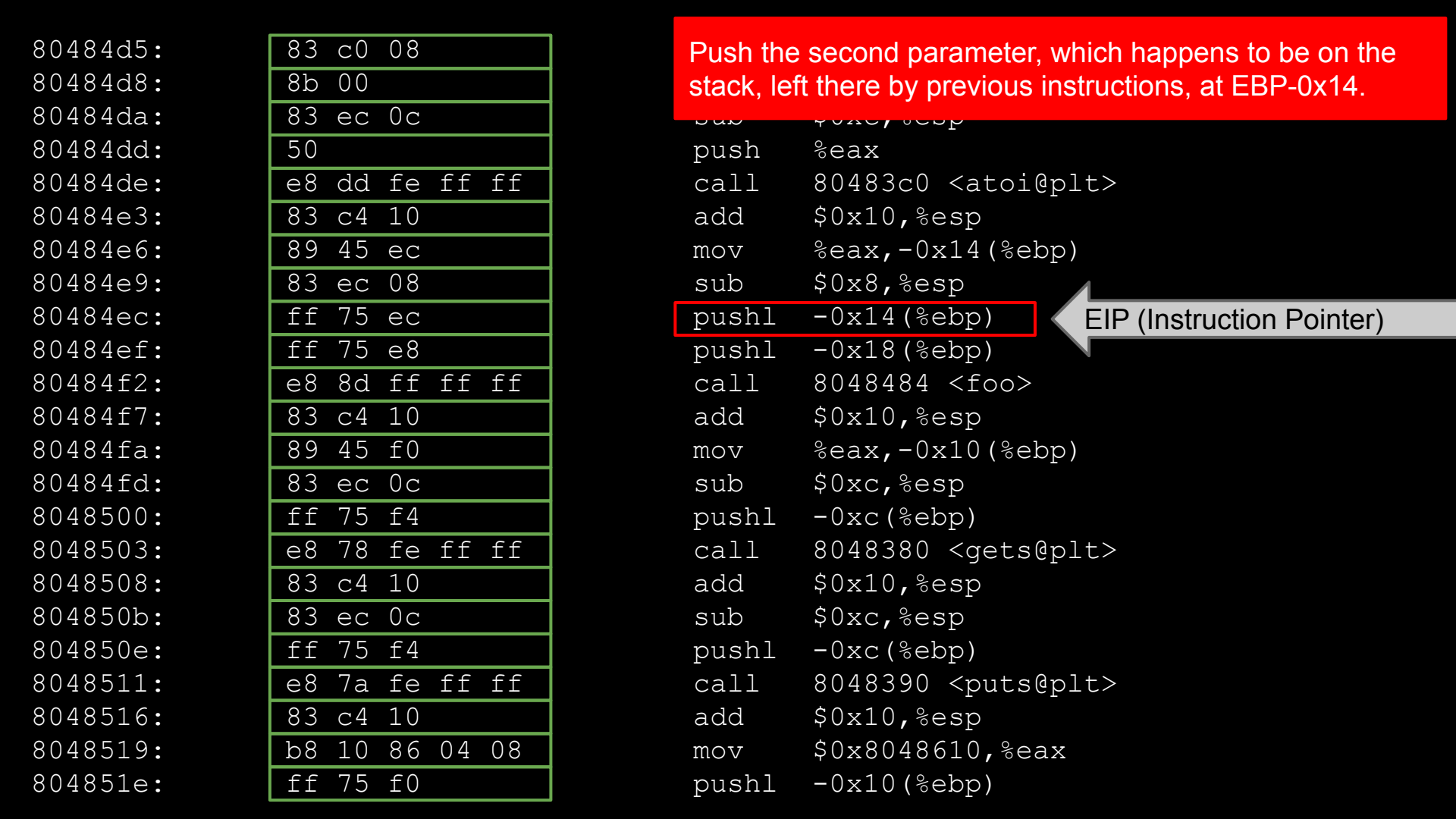
Recap on calling convention

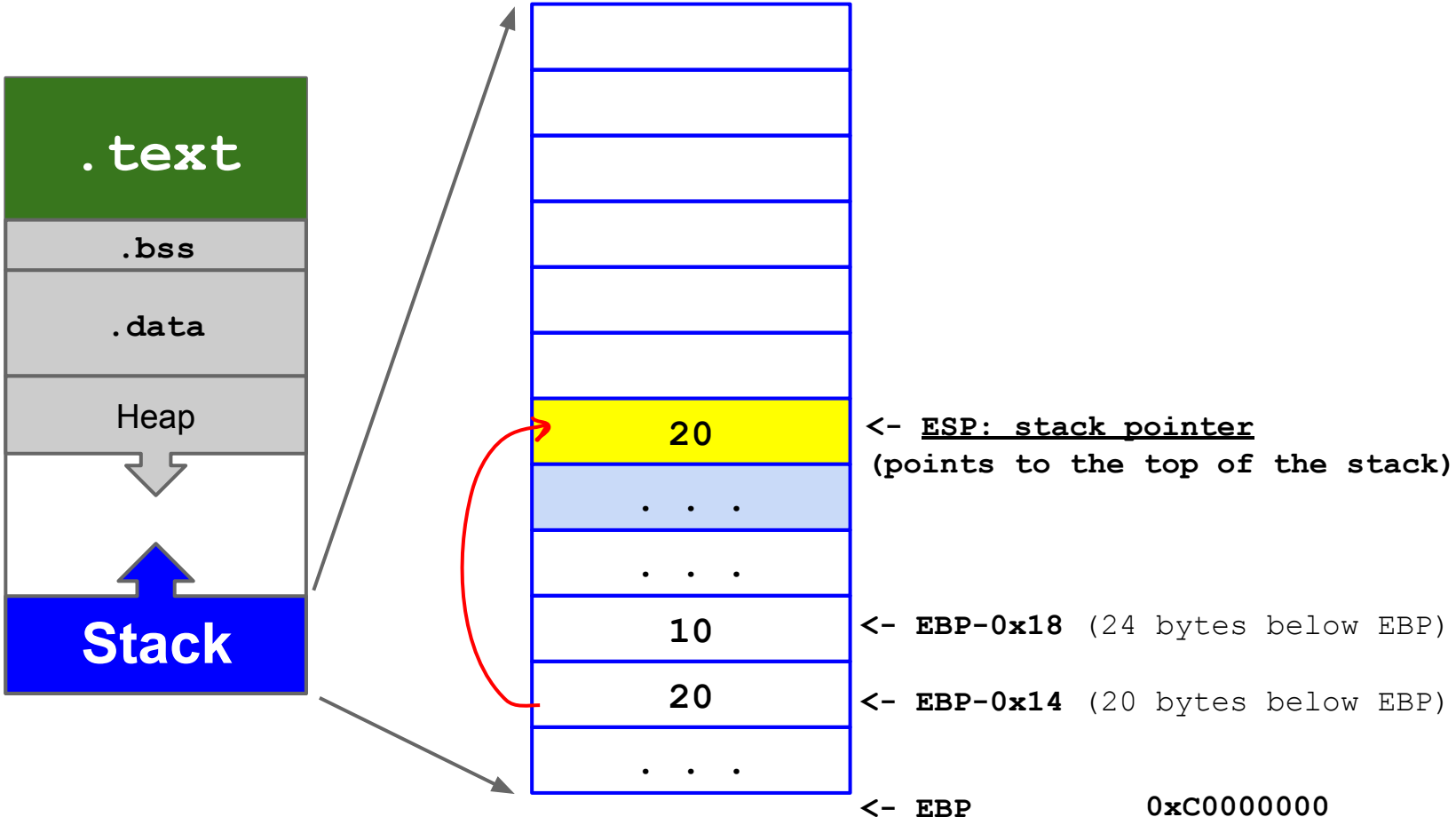
```
int foo(int a, int b) {
  int c = 14;
  return (a + b) * c;
}

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

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

  return 0;
}
```

./executable_file 10 20 <--

The foo() function receives two parameters by copy

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

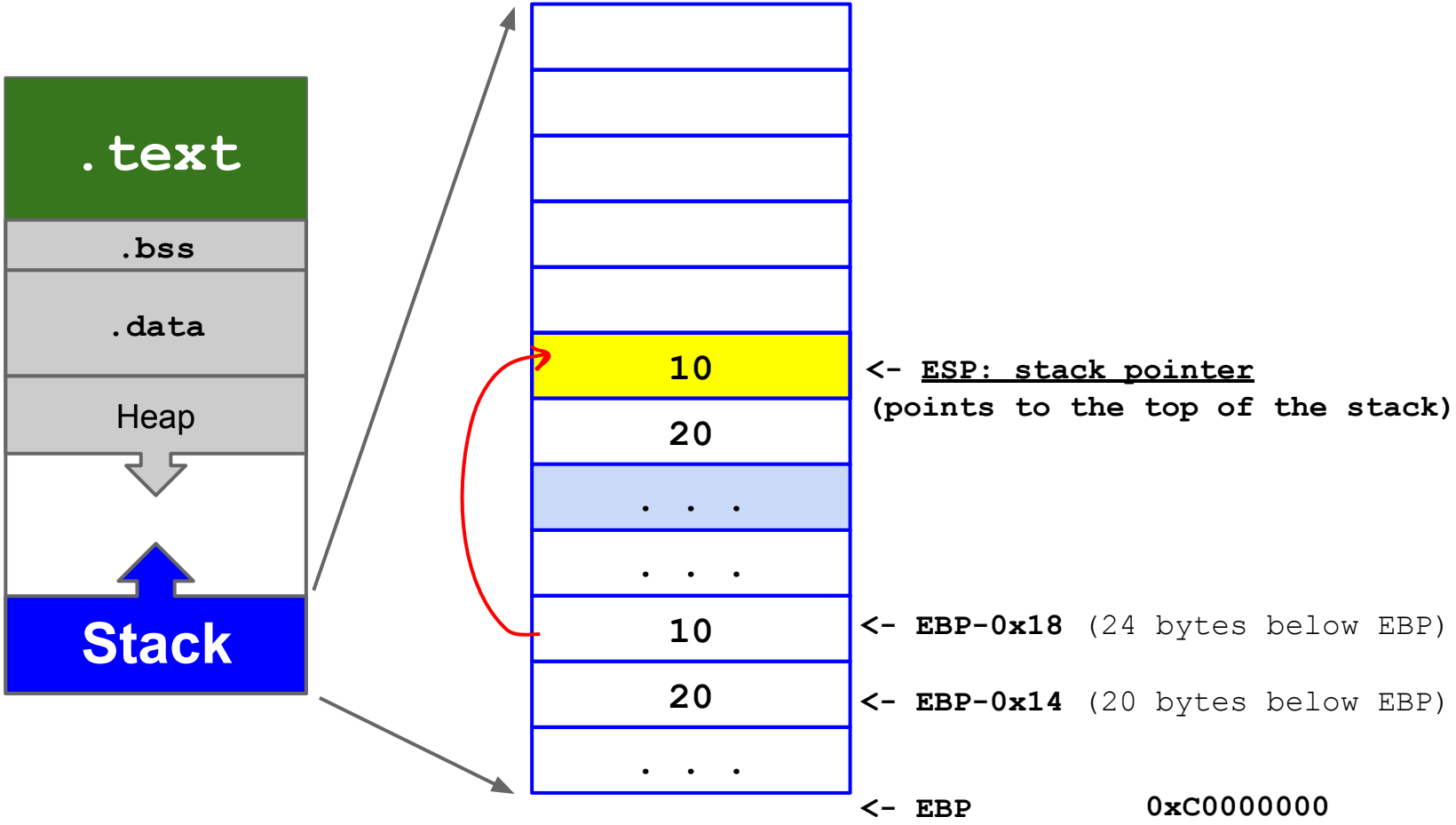

Push the second parameter, which happens to be on the stack, left there by previous instructions, at EBP-0x14.
80484d5: 83 c0 08 sub $0xc,%esp
80484d8: 8b 00 push %eax
80484da: 83 ec 0c call 80483c0 <atoi@plt>
80484dd: 50 add $0x10,%esp
80484de: e8 dd fe ff ff mov %eax,-0x14(%ebp)
80484e3: 83 c4 10 sub $0x8,%esp
80484e6: 89 45 ec pushl -0x14(%ebp)
80484e9: 83 ec 08 pushl -0x18(%ebp)
80484ec: ff 75 ec call 8048484 <foo>
80484ef: ff 75 e8 add $0x10,%esp
80484f2: e8 8d ff ff ff mov %eax,-0x10(%ebp)
80484f7: 83 c4 10 sub $0xc,%esp
80484fa: 89 45 f0 pushl -0xc(%ebp)
80484fd: 83 ec 0c call 8048380 <gets@plt>
8048500: ff 75 f4 add $0x10,%esp
8048503: e8 78 fe ff ff sub $0xc,%esp
8048508: 83 c4 10 pushl -0xc(%ebp)
804850b: 83 ec 0c call 8048390 <puts@plt>
804850e: ff 75 f4 add $0x10,%esp
8048511: e8 7a fe ff ff mov $0x8048610,%eax
8048516: 83 c4 10 pushl -0x10(%ebp)
8048519: b8 10 86 04 08
804851e: ff 75 f0
EIP (Instruction Pointer)

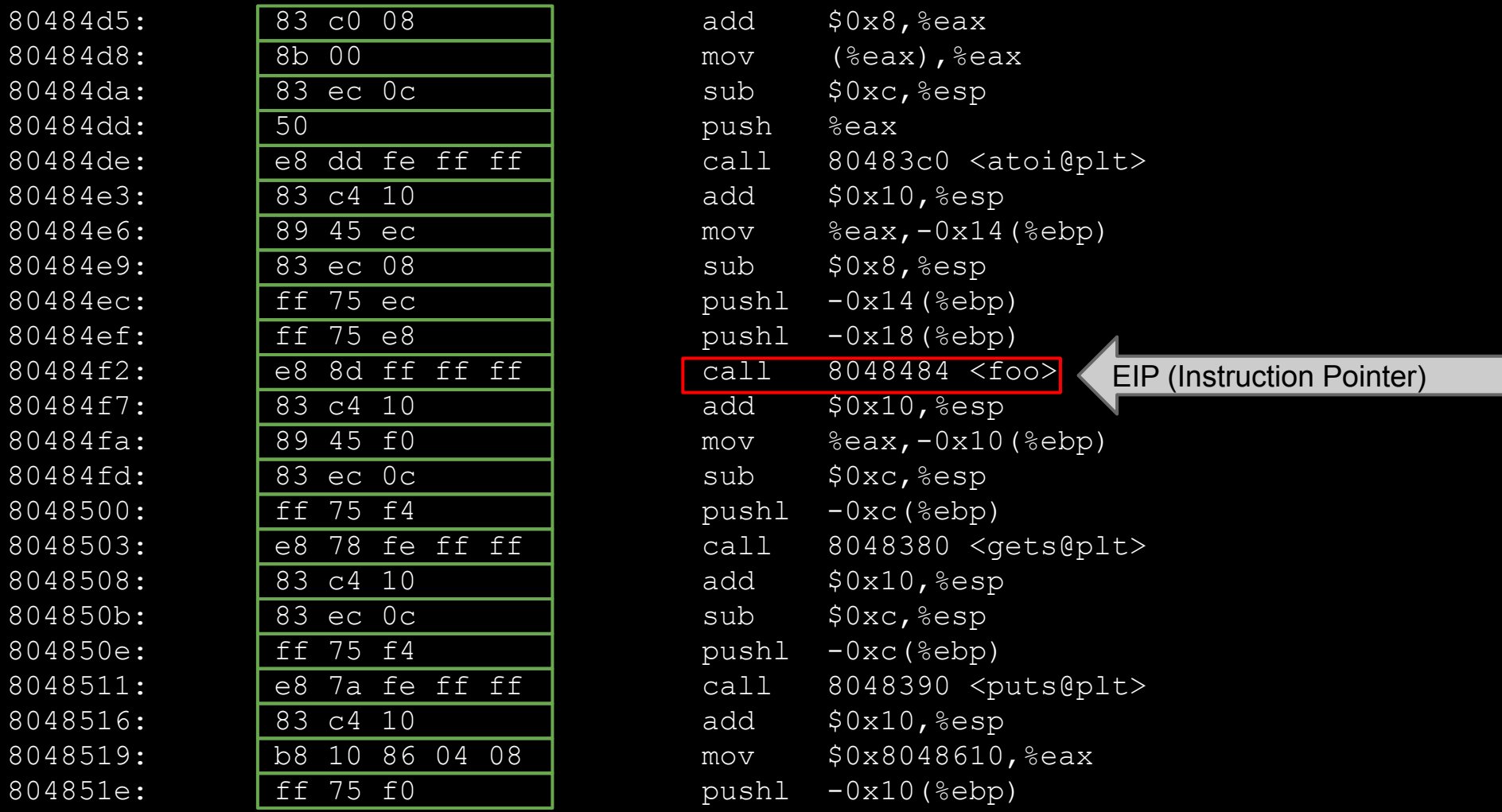

0xBFFDF000
.text
.bss
.data
Heap
Stack
20
<- ESP: stack pointer
(points to the top of the stack)
. . .
. . .
10
<- EBP-0x18 (24 bytes below EBP)
20
<- EBP-0x14 (20 bytes below EBP)
. . .
<- EBP
0xC0000000

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

Push the first parameter

EIP (Instruction Pointer)


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


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

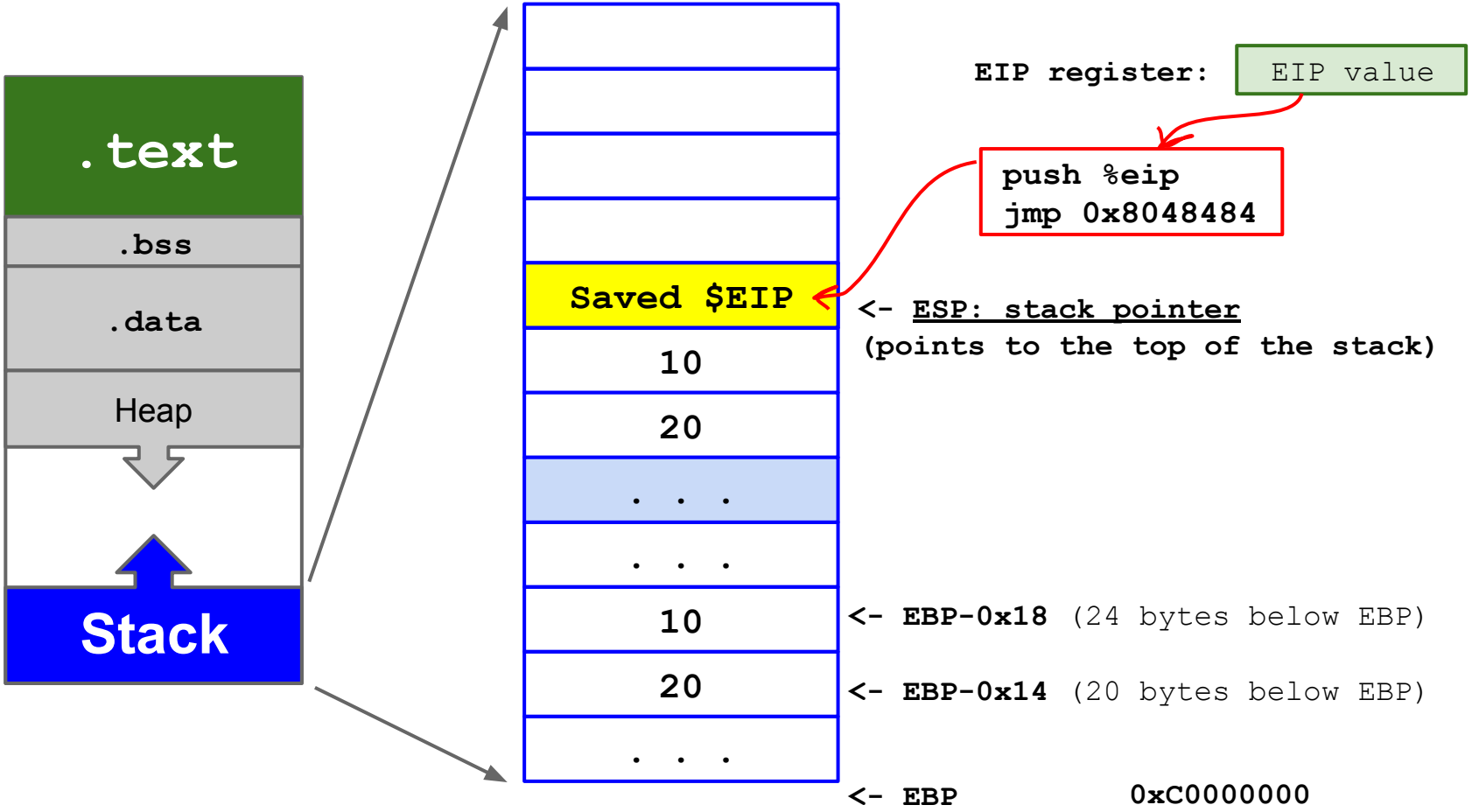
The `call` instruction

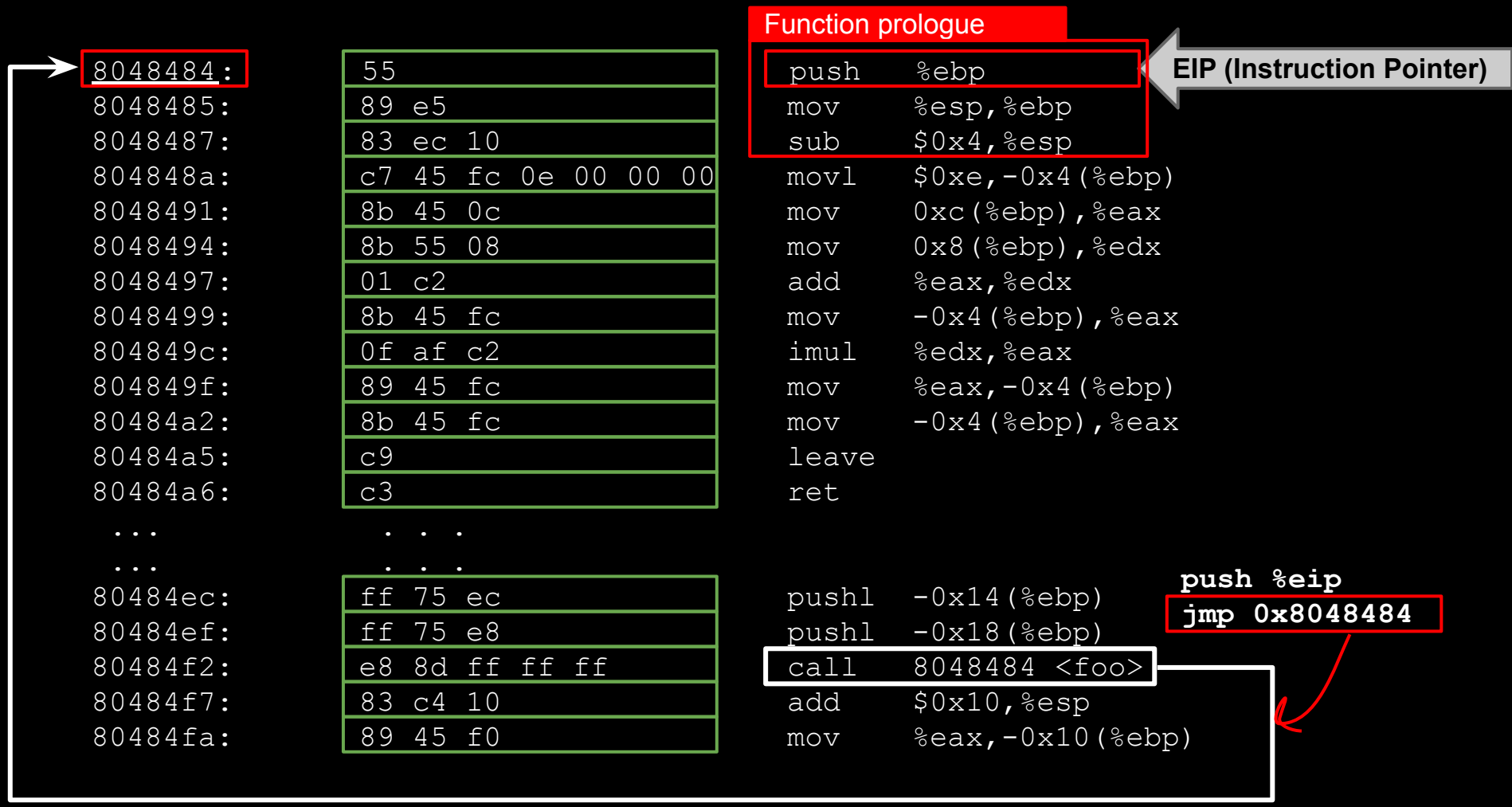
- The CPU is about to call the `foo()` function
- When `foo()` will be over, where to jump?

- 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()
```


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


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

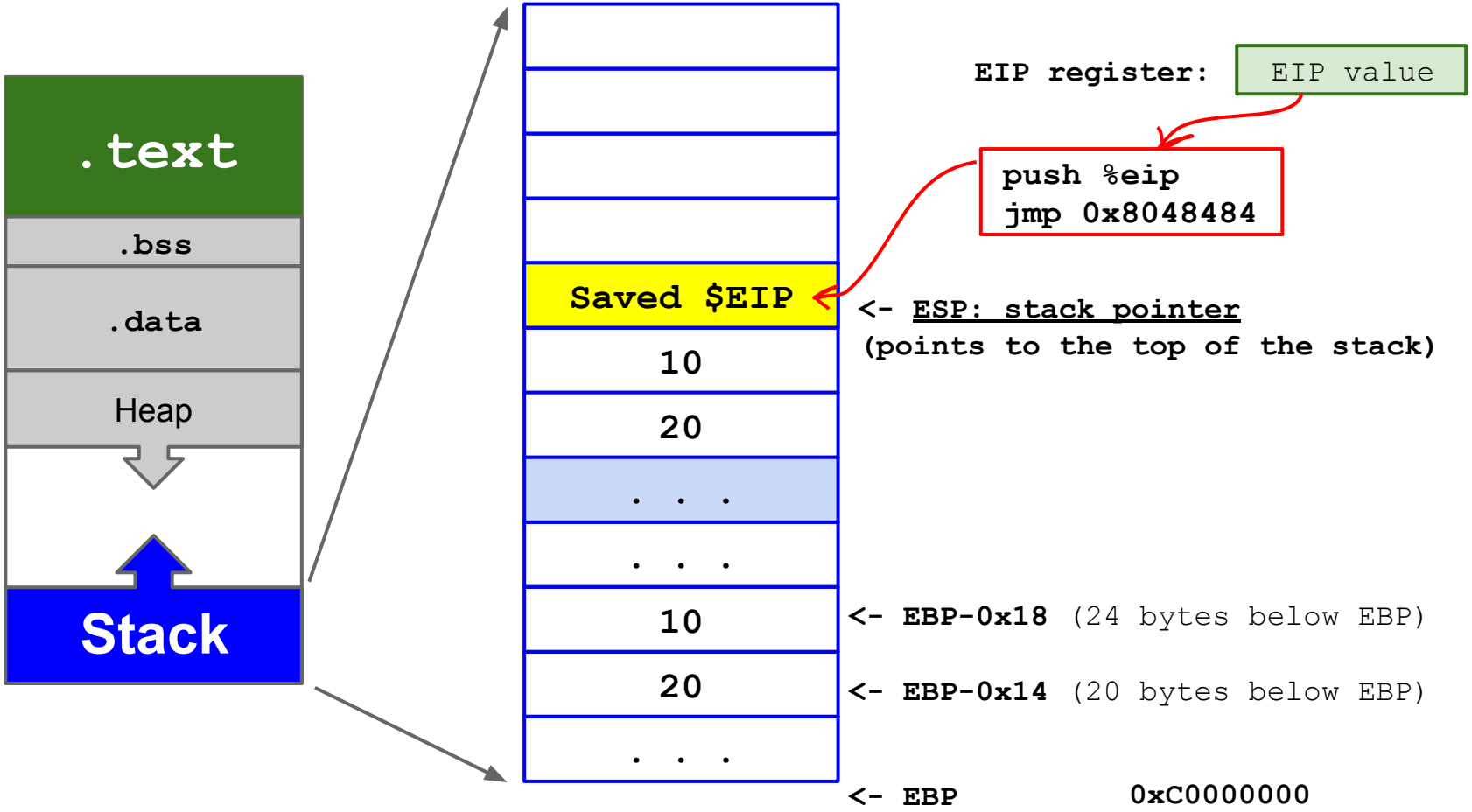
```
push    %ebp
mov     %esp,%ebp
sub     $0x4,%esp
```

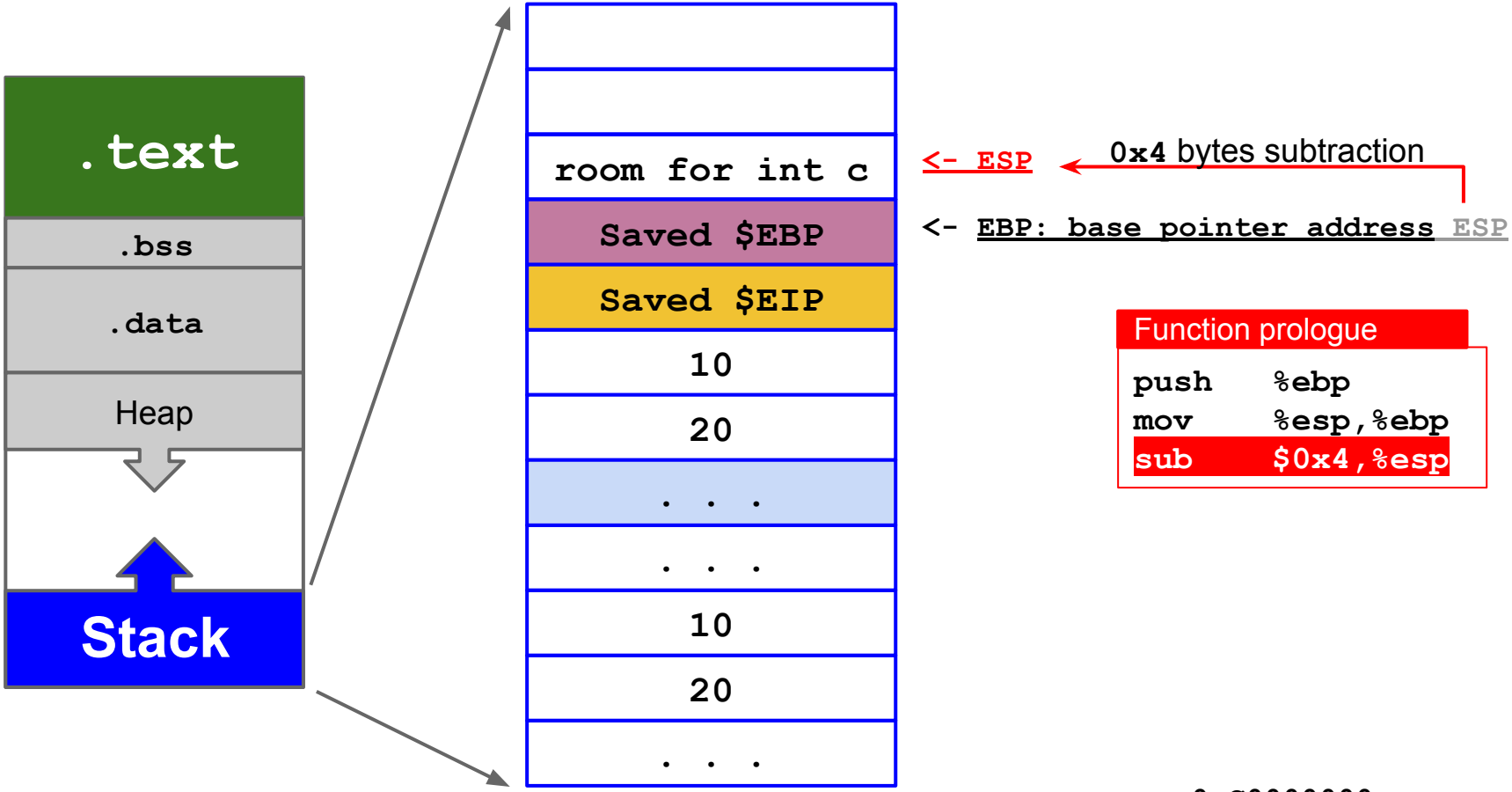
save the current stack base address onto the stack

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;
}
```


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


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

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

Function epilogue

EIP (Instruction Pointer)

Function Epilogue

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

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

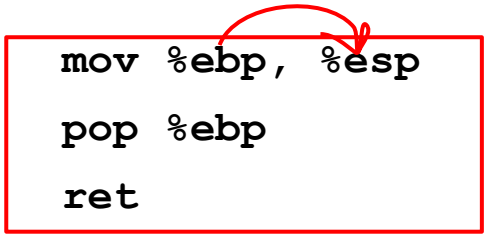
these 2 instructions translate into these 3 instructions

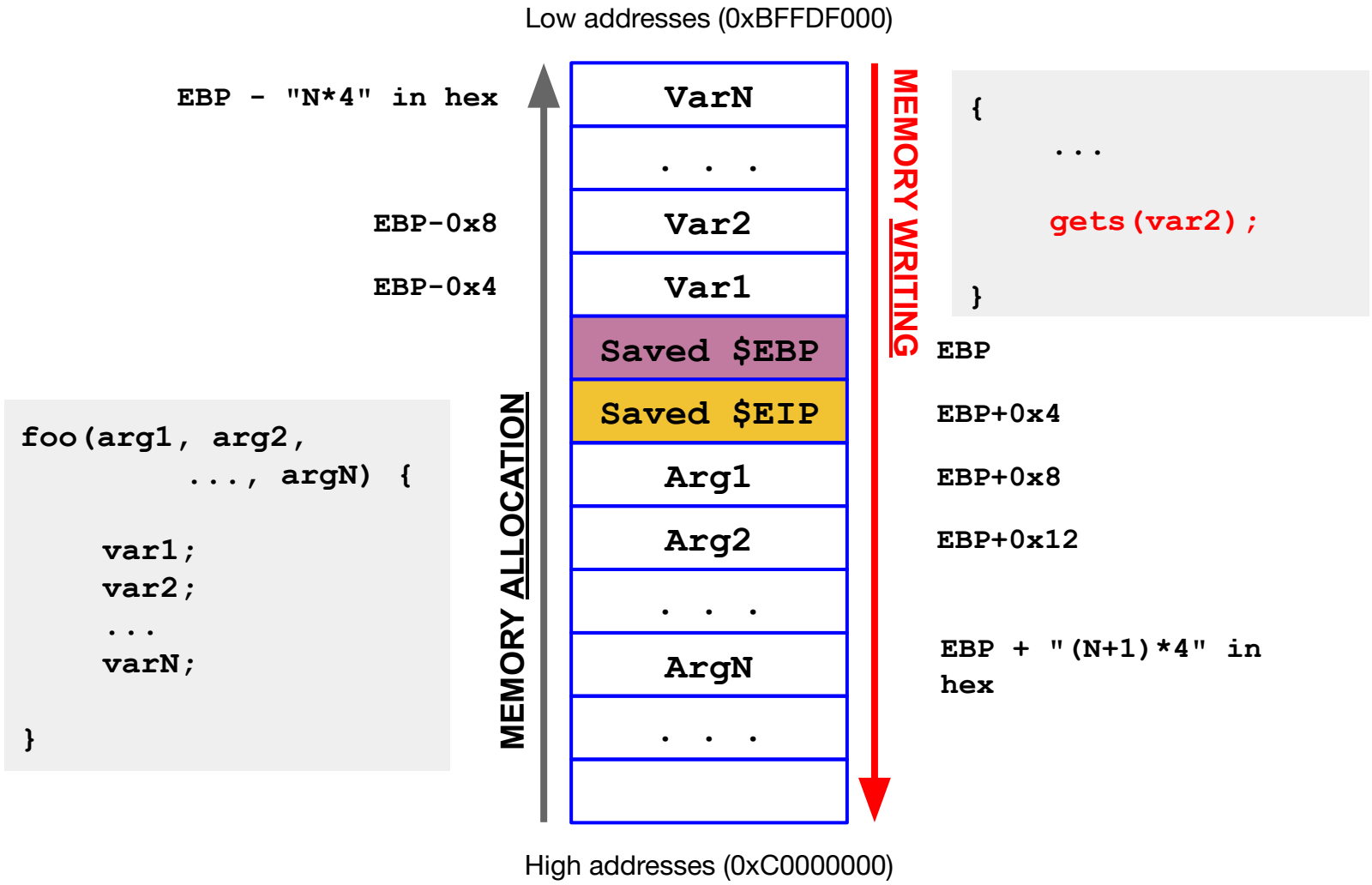
```
leave
ret
```

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

restore the **saved EBP** to registry

pop the saved EIP and jump there




Low addresses (0xBFFDF000)
EBP - "N*4" in hex
EBP-0x8
EBP-0x4
VarN
. . .
Var2
Var1
Saved $EBP
Saved $EIP
Arg1
Arg2
. . .
ArgN
. . .
MEMORY ALLOCATION
MEMORY WRITING
{
...
gets(var2);
}
EBP
EBP+0x4
EBP+0x8
EBP+0x12
EBP + "(N+1)*4" in hex
foo(arg1, arg2,
..., argN) {
var1;
var2;
...
varN;
}
High addresses (0xC0000000)

Now you know how the stack is used...


WHAT IF WE OVERWRITE
OTHER DATA

Stack Smashing

1994 idea (well explained by aleph1)

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

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

chall-0


WHAT IF WE CHANGE
THE SAVED EIP

chall-1

Where to Jump?

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

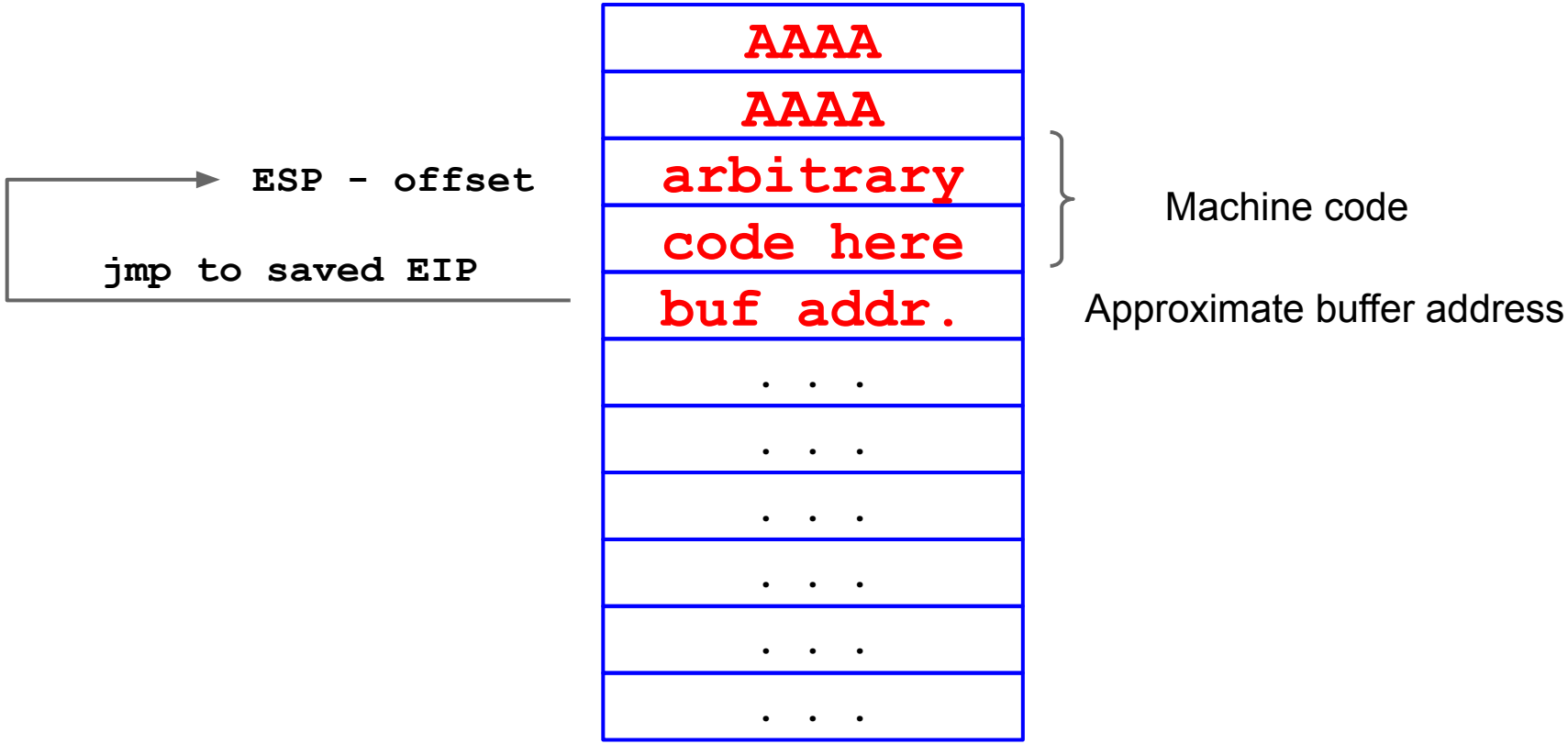
Solutions (i.e., exploitation techniques):

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

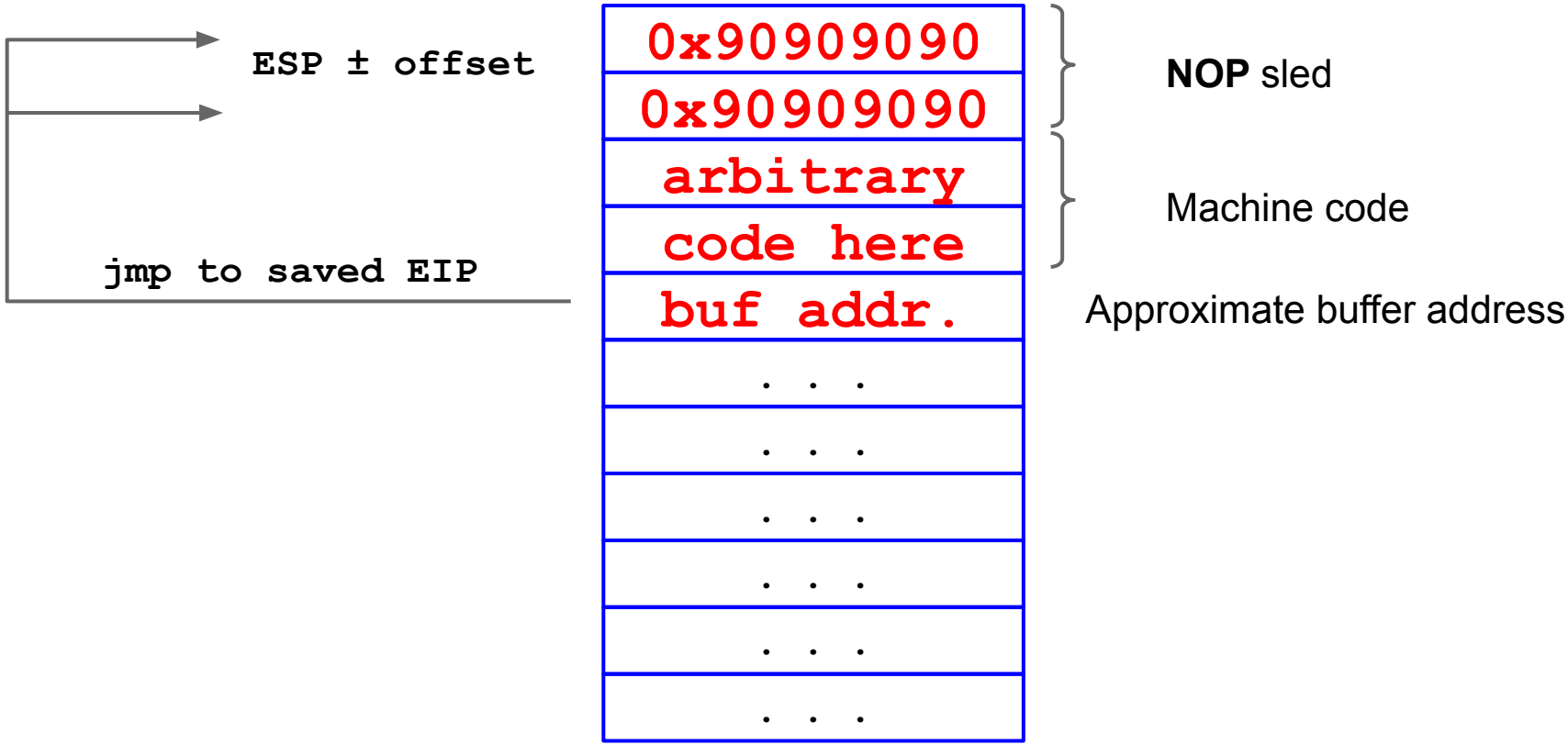
Smashing the Stack

Where to jump?

Smashing the Stack



NOP (0x90) Sled to the Resc



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")`**

Writing a Shellcode in a Nutshell

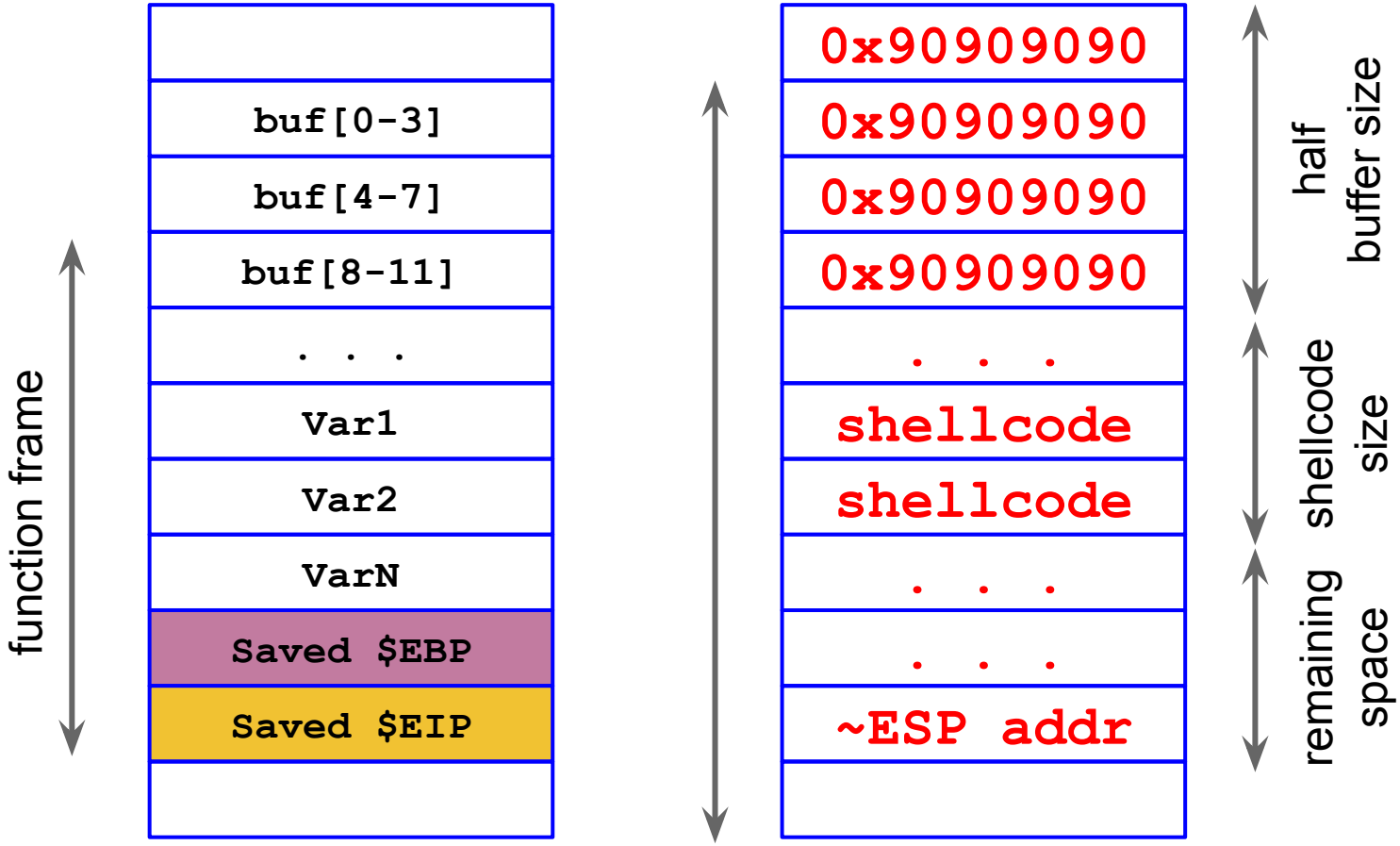
General idea: Translate **`execve("/bin/sh", NULL, NULL)`** into machine instructions

Invoke system call by executing a software interrupt through the **`int`** instruction

Trick: Addresses should be relative

Problem: Avoid zeros (replace semantically equivalent instructions)

```
shellcode = "\x31\xc0\x50\x89\xe2\x68\x2f\x2f\x73\x68\x68
\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"
```



Vulnerable program's memory layout (function frame).

Memory layout of a possible exploit.

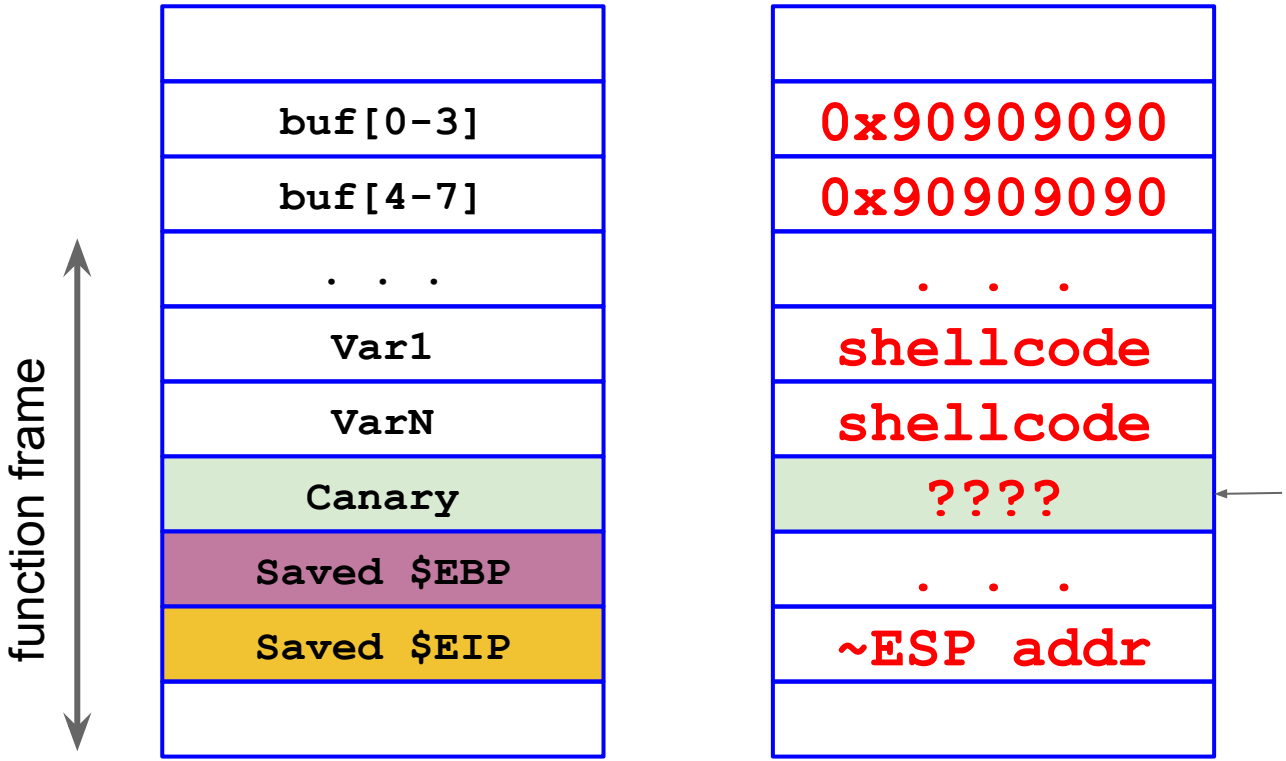
chall-2

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

Protection Mechanisms

- **Stack Canaries**
- **ASLR**: Address Space Layout Randomization
- **NX** or **DEP**: No section is executable & writable
- **RELRO**: Read-Only after Relocation
- **PIE**: Position Independent Executable
- **CFI**: Control-Flow Integrity

Stack Canaries



Vulnerable program's memory layout (function frame).

Memory layout of a possible exploit.

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

Non-executable Stack

Non-executable stack (data != code)

- No stack smashing or 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)

Built-in, Existing Functions

The address of a system library or function (e.g., 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

Address Space Layout Randomization

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
```

chall-3

```
echo 2 > /proc/sys/kernel/randomize_va_space
```


imgflip.com

Bonus: chall-4

Further Reading

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

Aleph One, *"Smashing The Stack For Fun And Profit"*, Phrack, 1996, http://phrack.org/issues/49/14.html#article
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)

[course] Modern Binary Exploitation - CSCI 4968 http://security.cs.rpi.edu/courses/binexp-spring2015/ / https://github.com/RPISEC/MBE

Thanks!

Andrea Continella

a.continella@utwente.nl
https://conand.me

@_conand