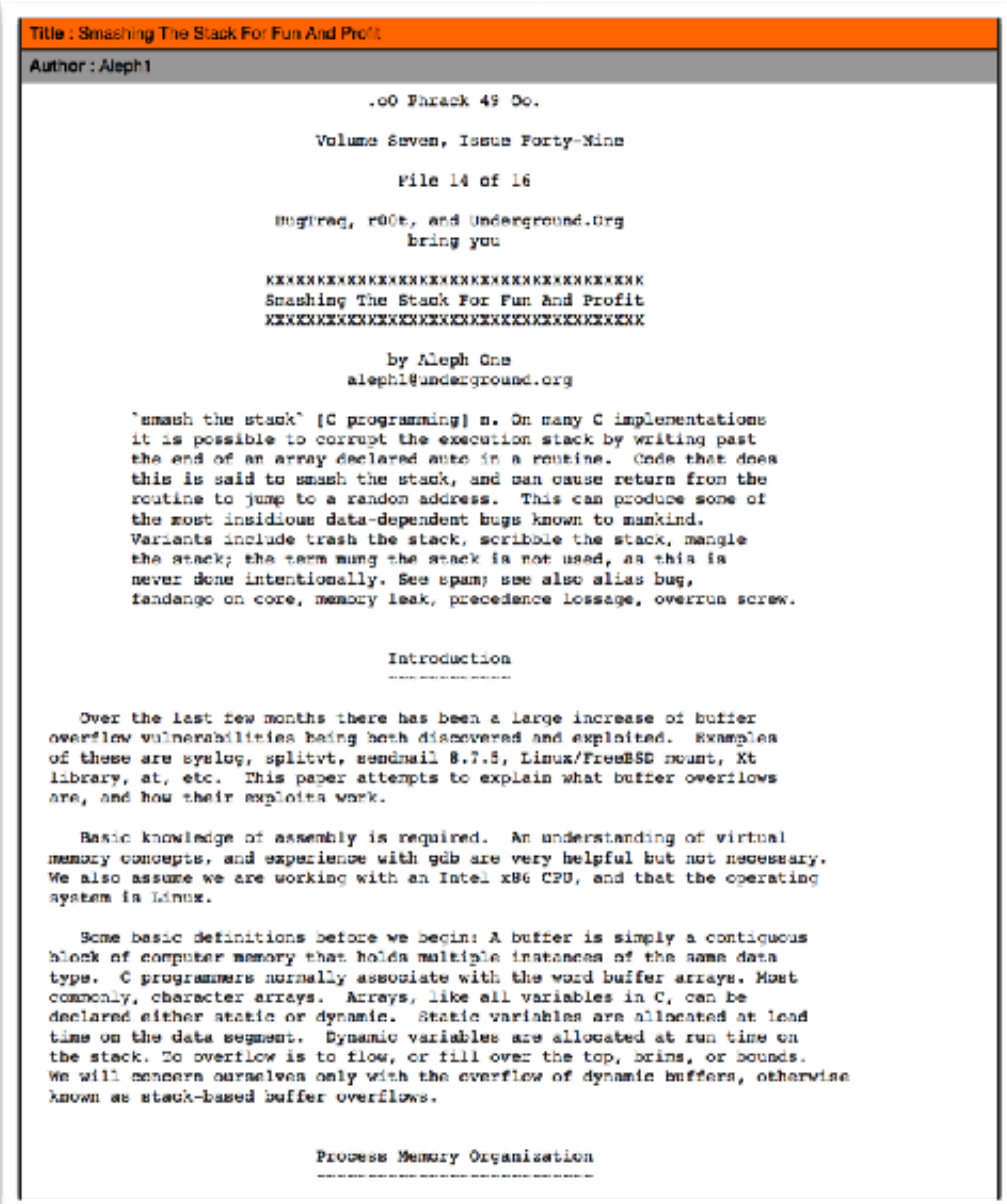
CLASSIC
MEMORY ATKS & DEFS

GRAD SEC

SEP 07 2017

TODAY'S PAPERS



Title : Smashing The Stack For Fun And Profit

Author : Aleph1

```
                          .oO Phrack 49 Oo.

                    Volume Seven, Issue Forty-Nine

                            File 14 of 16

                  BugTraq, r00t, and Underground.Org
                              bring you

                 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                 Smashing The Stack For Fun And Profit
                 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

                           by Aleph One
                      aleph1@underground.org

        `smash the stack` [C programming] n. On many C implementations
        it is possible to corrupt the execution stack by writing past
        the end of an array declared auto in a routine.  Code that does
        this is said to smash the stack, and can cause return from the
        routine to jump to a random address.  This can produce some of
        the most insidious data-dependent bugs known to mankind.
        Variants include trash the stack, scribble the stack, mangle
        the stack; the term mung the stack is not used, as this is
        never done intentionally. See spam; see also alias bug,
        fandango on core, memory leak, precedence lossage, overrun screw.


                             Introduction
                             ~~~~~~~~~~~~

   Over the last few months there has been a large increase of buffer
overflow vulnerabilities being both discovered and exploited.  Examples
of these are syslog, splitvt, sendmail 8.7.5, Linux/FreeBSD mount, Xt
library, at, etc.  This paper attempts to explain what buffer overflows
are, and how their exploits work.

   Basic knowledge of assembly is required.  An understanding of virtual
memory concepts, and experience with gdb are very helpful but not necessary.
We also assume we are working with an Intel x86 CPU, and that the operating
system is Linux.

   Some basic definitions before we begin: A buffer is simply a contiguous
block of computer memory that holds multiple instances of the same data
type.  C programmers normally associate with the word buffer arrays. Most
commonly, character arrays.  Arrays, like all variables in C, can be
declared either static or dynamic.  Static variables are allocated at load
time on the data segment.  Dynamic variables are allocated at run time on
the stack.  To overflow is to flow, or fill over the top, brims, or bounds.
We will concern ourselves only with the overflow of dynamic buffers, otherwise
known as stack-based buffer overflows.


                      Process Memory Organization
                      ~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks*

Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton†, Jonathan Walpole,
Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle and Qian Zhang
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
immunix-request@cse.ogi.edu, http://cse.ogi.edu/DISC/projects/immunix

Abstract

This paper presents a systematic solution to the persistent problem of buffer overflow attacks. Buffer overflow attacks gained notoriety in 1988 as part of the Morris Worm incident on the Internet. While it is fairly simple to fix individual buffer overflow vulnerabilities, buffer overflow attacks continue to this day. Hundreds of attacks have been discovered, and while most of the obvious vulnerabilities have now been patched, more sophisticated buffer overflow attacks continue to emerge.

We describe StackGuard: a simple compiler technique that virtually eliminates buffer overflow vulnerabilities with only modest performance penalties. Privileged programs that are recompiled with the StackGuard compiler extension no longer yield control to the attacker, but rather enter a fail-safe state. These programs require *no* source code changes at all, and are binary-compatible with existing operating systems and libraries. We describe the compiler technique (a simple patch to gcc), as well as a set of variations on the technique that trade-off between penetration resistance and performance. We present experimental results of both the penetration resistance and the performance impact of this technique.

1 Introduction

This paper presents a systematic solution to the persistent problem of buffer overflow attacks. Buffer overflow attack gained notoriety in 1988 as part of the Morris Worm incident on the Internet [23]. Despite the fact that fixing individual buffer overflow vulnerabilities is fairly simple, buffer overflow attacks continue to this day, as reported in the SANS Network Security Digest:

> Buffer overflows appear to be the most common problems reported in May, with degradation-of-service problems a distant second. Many of the buffer overflow problems are probably the result of careless programming, and could have been found and corrected by the vendors, before releasing the software, if the vendors had performed elementary testing or code reviews along the way.[4]

The base problem is that, while individual buffer overflow vulnerabilities are simple to patch, the vulnerabilities are profligate. Thousands of lines of legacy code are still running as privileged daemons (SUID root) that contain numerous software errors. New programs are being developed with more care, but are often still developed using unsafe languages such as C, where simple errors can leave serious vulnerabilities.

The continued success of these attacks is also due to the "patchy" nature by which we protect against such attacks. The life cycle of a buffer overflow attack is simple: A (malicious) user finds the vulnerability in a highly priv-

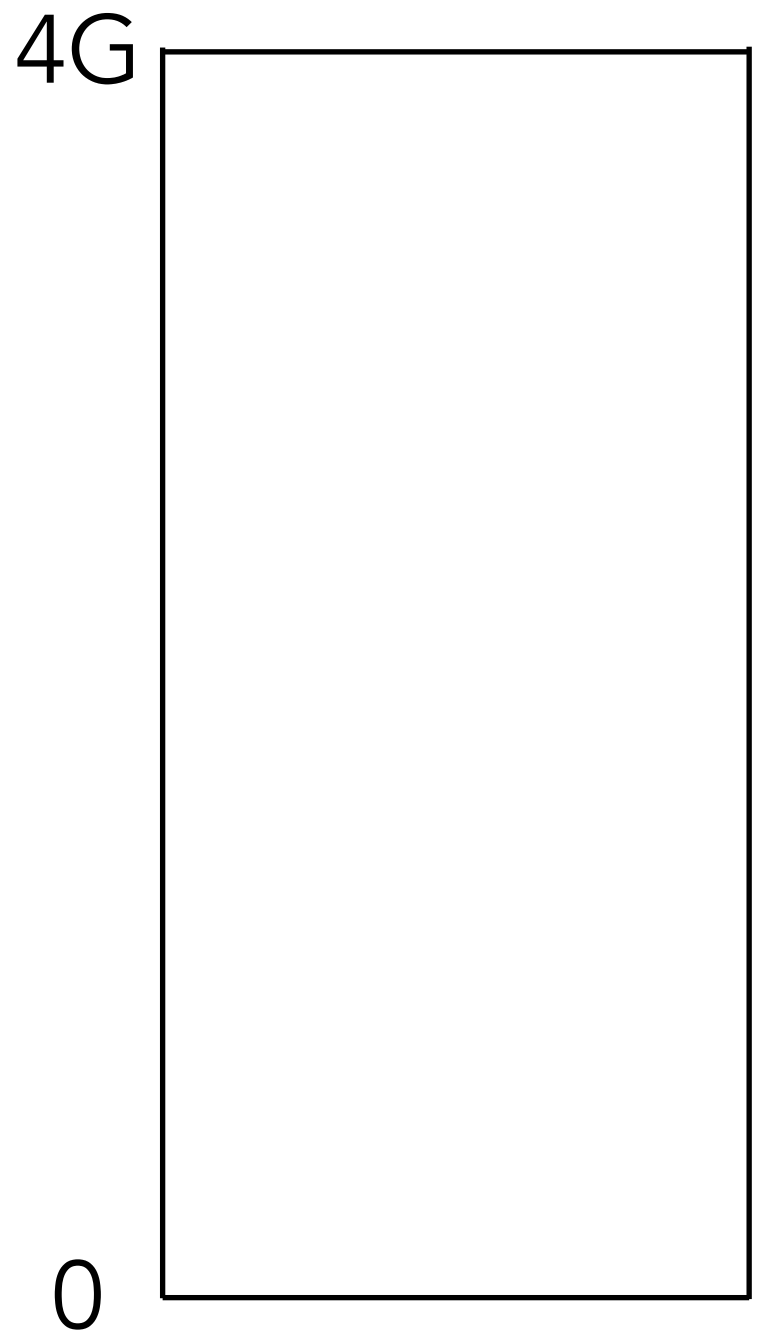
*This research is partially supported by DARPA contracts F30602-96-1-0331 and F30602-96-1-0302.

†Ryerson Polytechnic University

REFRESHER

- How is program data laid out in memory?
- What does the stack look like?
- What effect does calling (and returning from) a function have on memory?
- We are focusing on the Linux process model
    - Similar to other operating systems

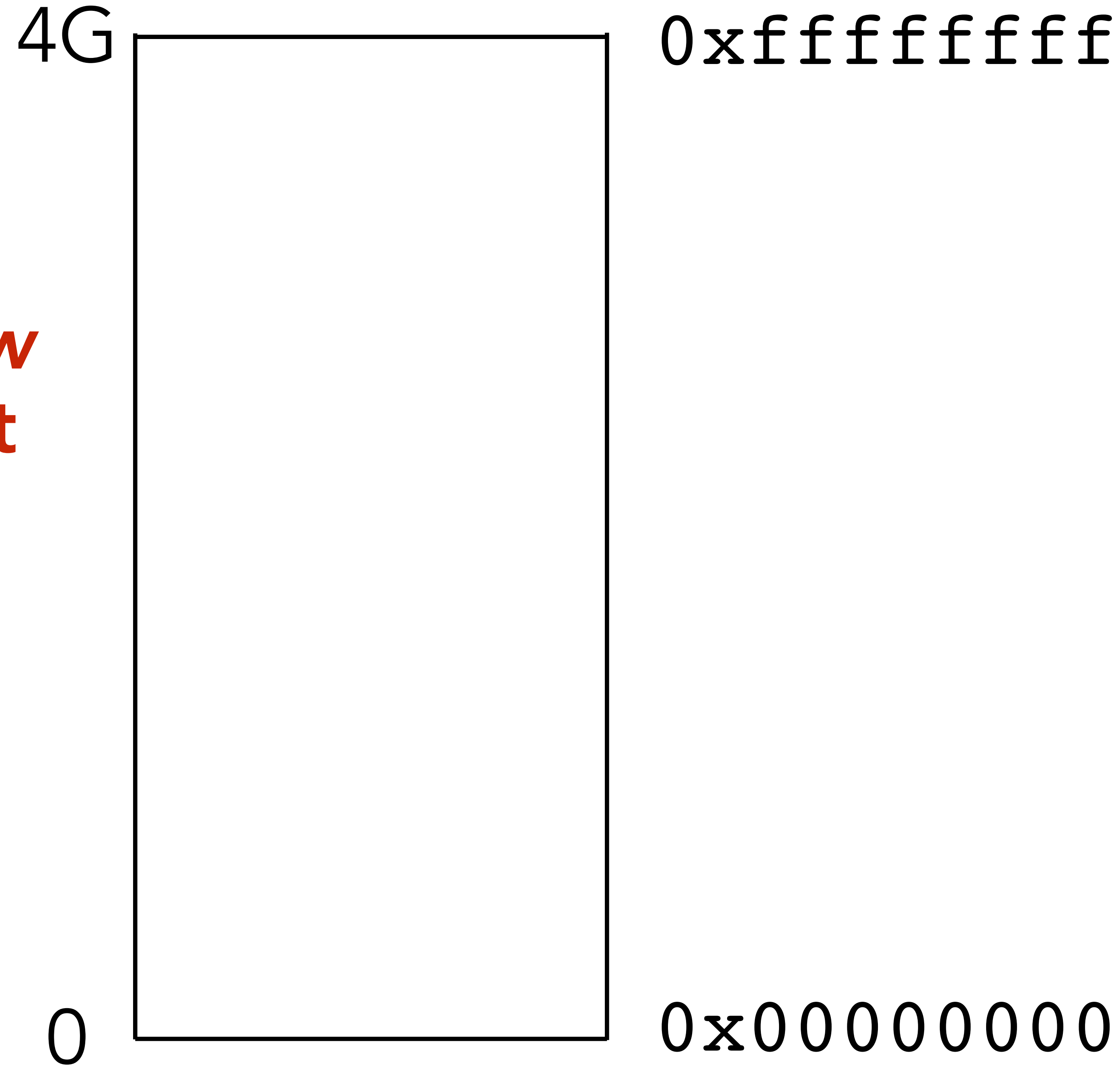
ALL PROGRAMS ARE STORED IN MEMORY



ALL PROGRAMS ARE STORED IN MEMORY

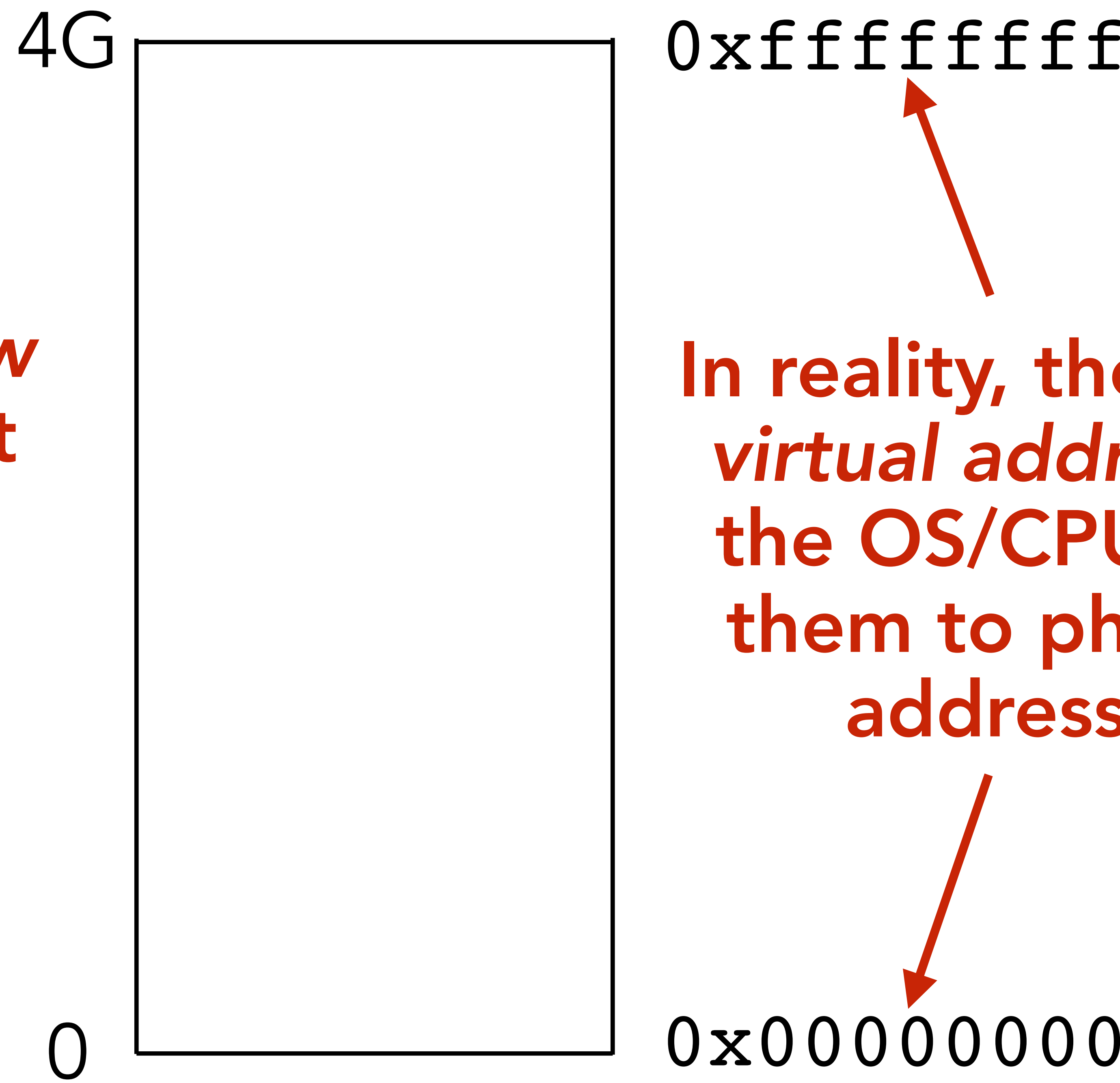
ALL PROGRAMS ARE STORED IN MEMORY

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



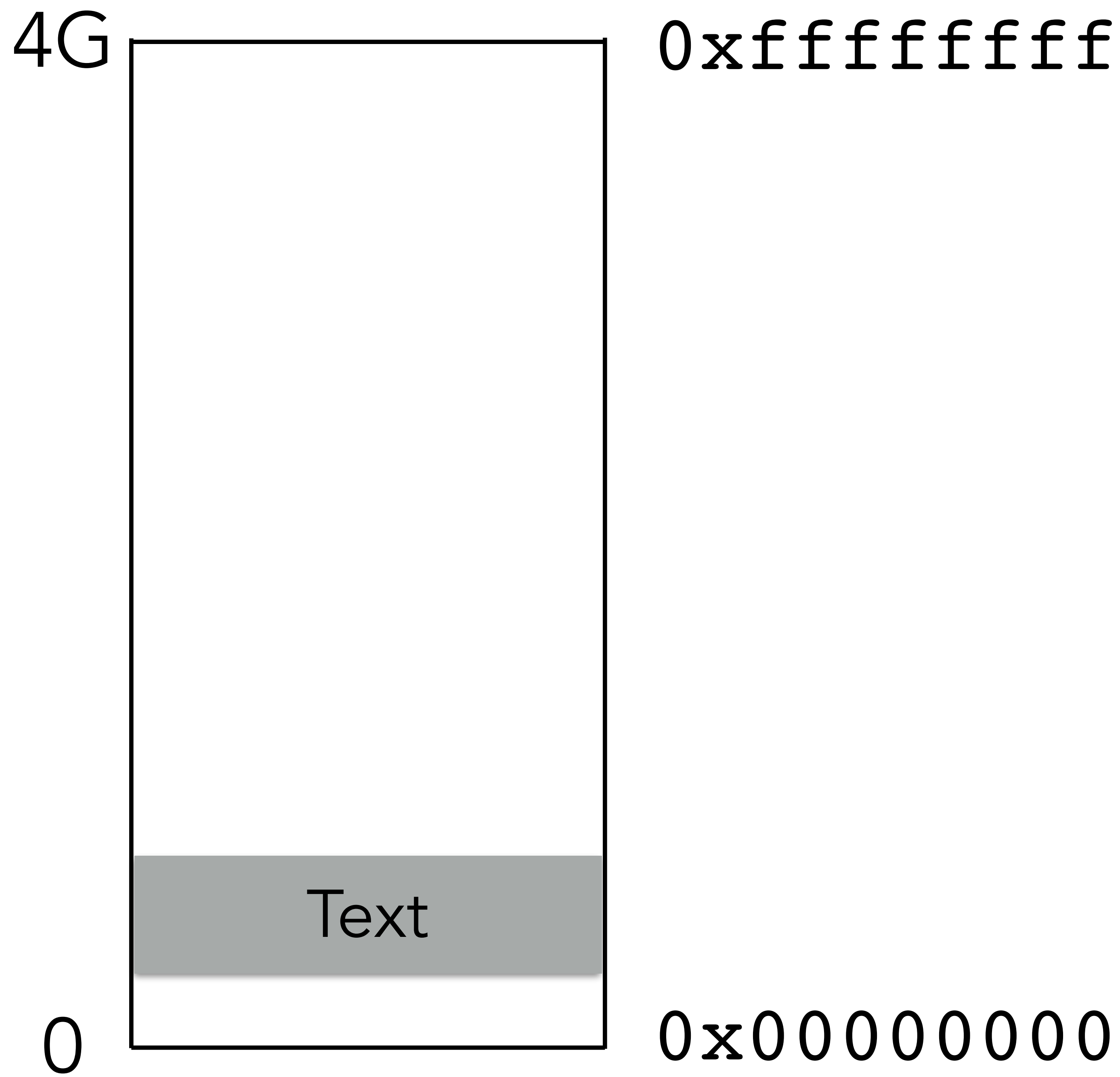
ALL PROGRAMS ARE STORED IN MEMORY

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

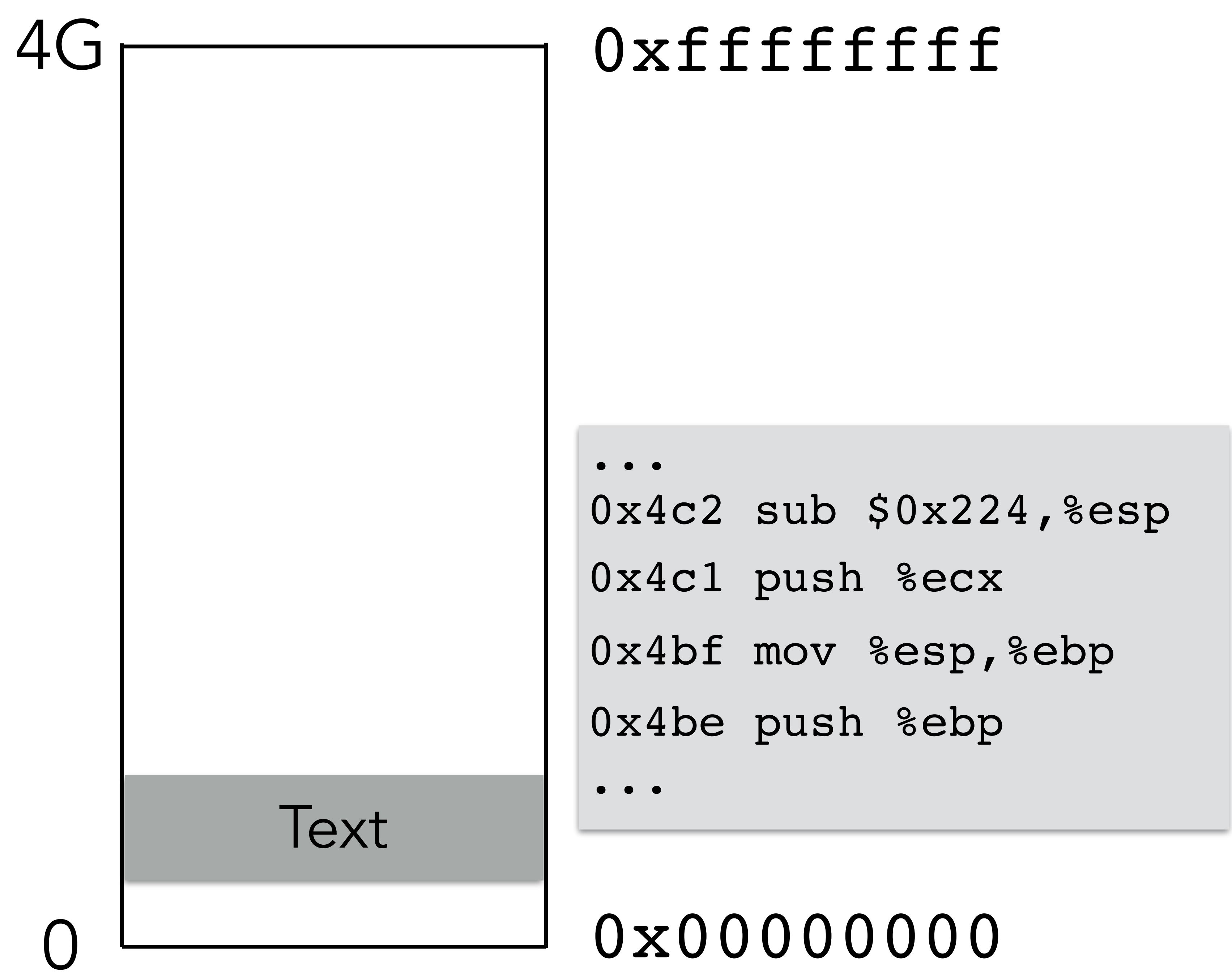


In reality, these are *virtual addresses*; the OS/CPU map them to physical addresses

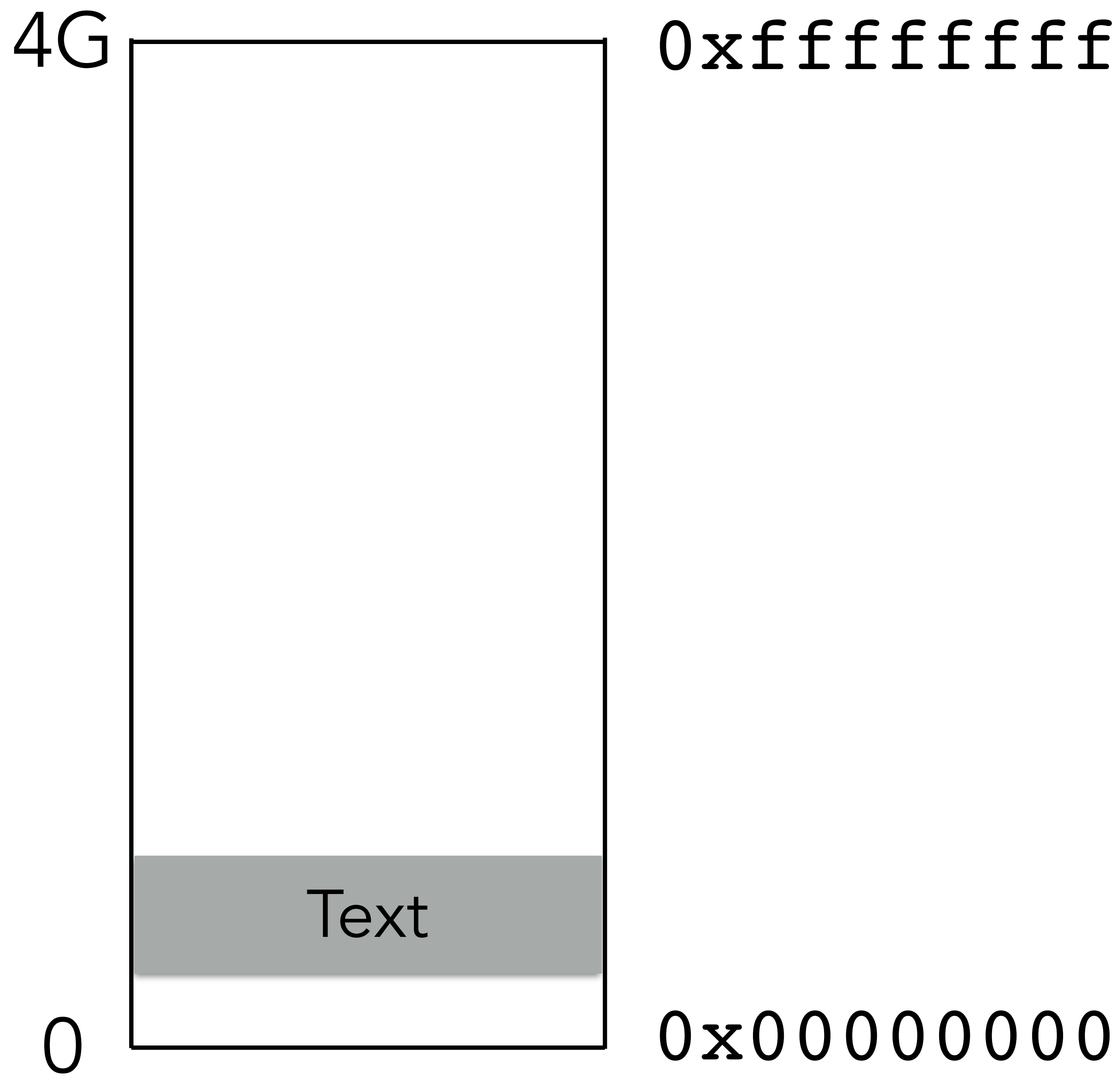
THE INSTRUCTIONS THEMSELVES ARE STORED IN MEMORY



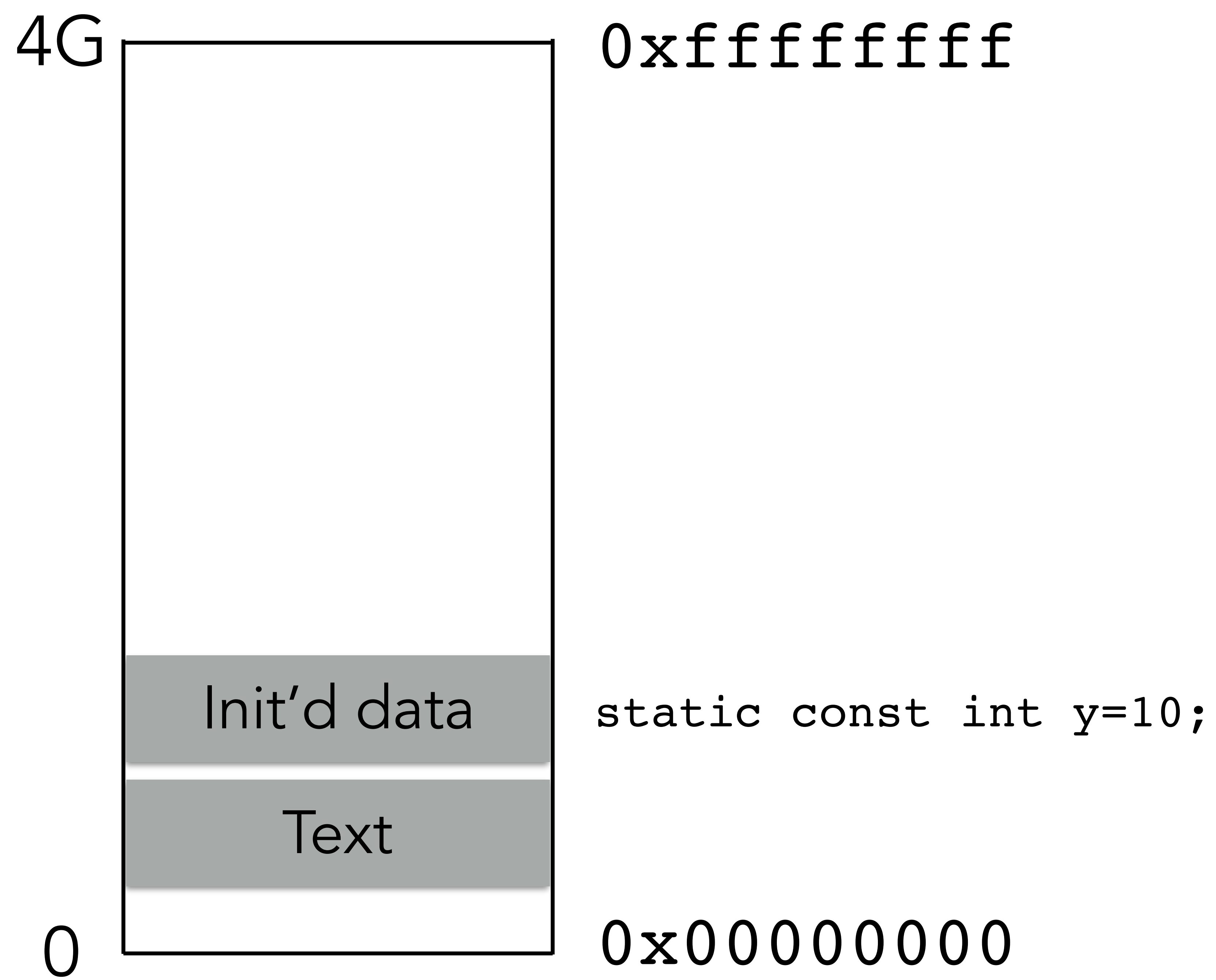
THE INSTRUCTIONS THEMSELVES ARE STORED IN MEMORY



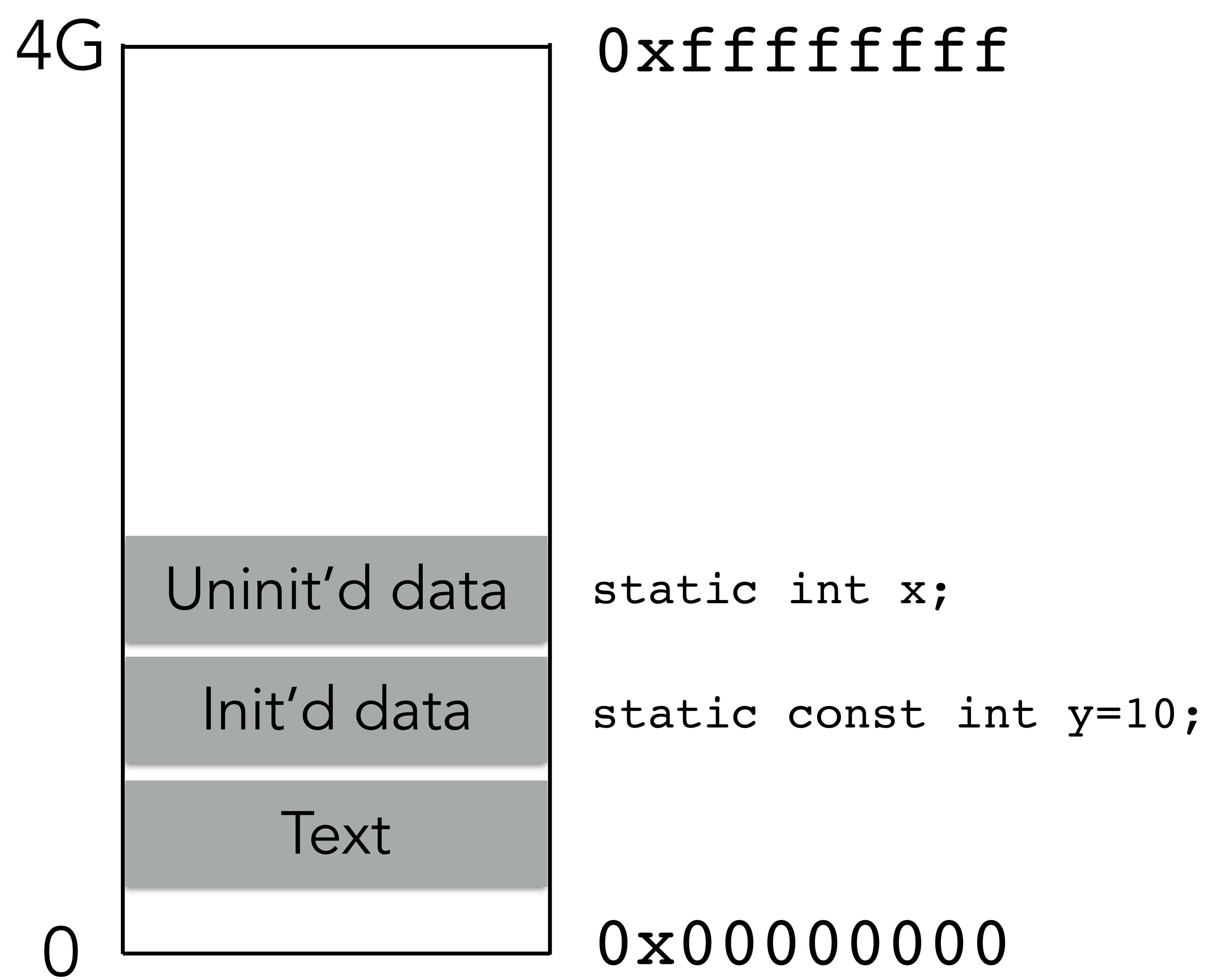
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



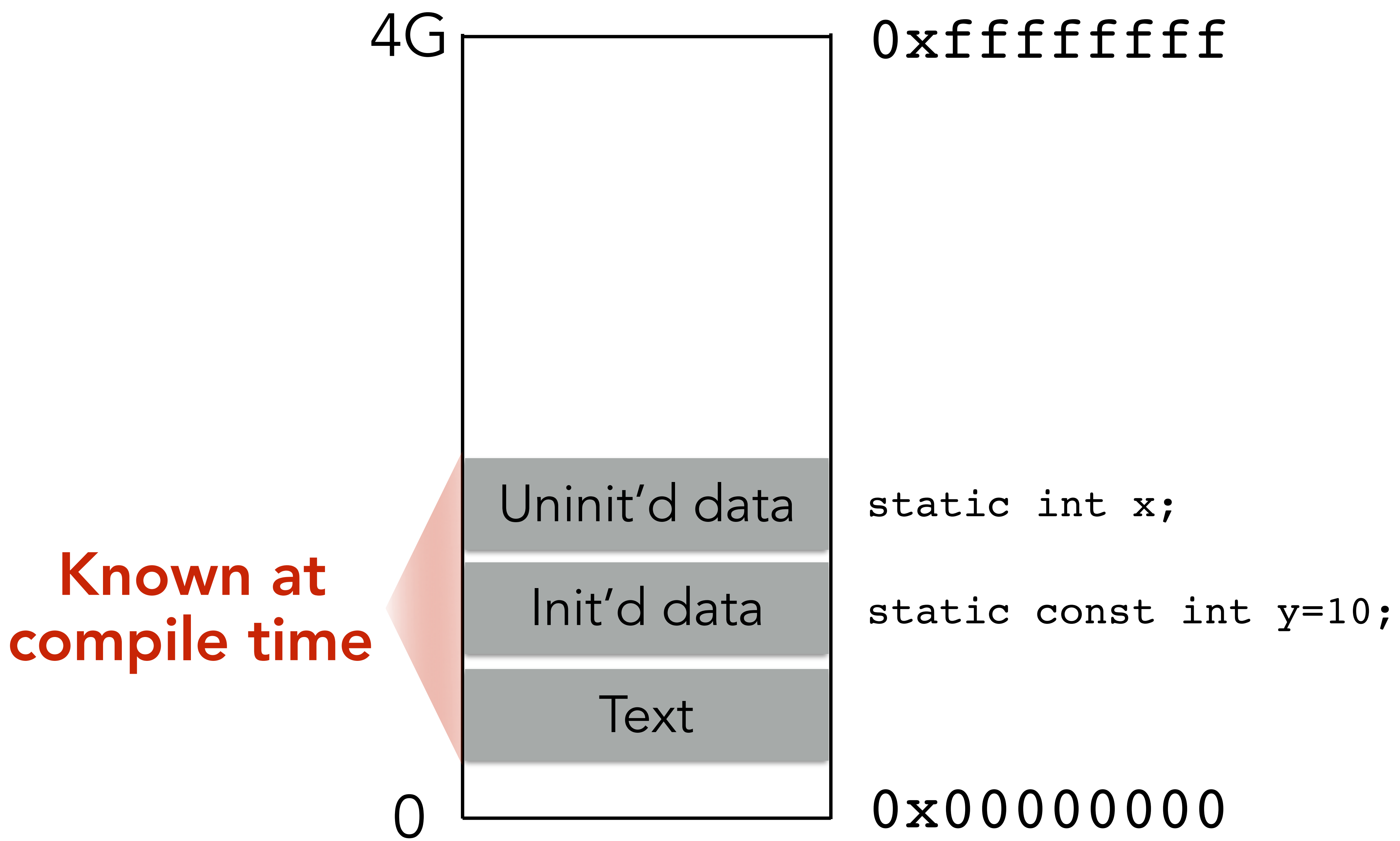
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



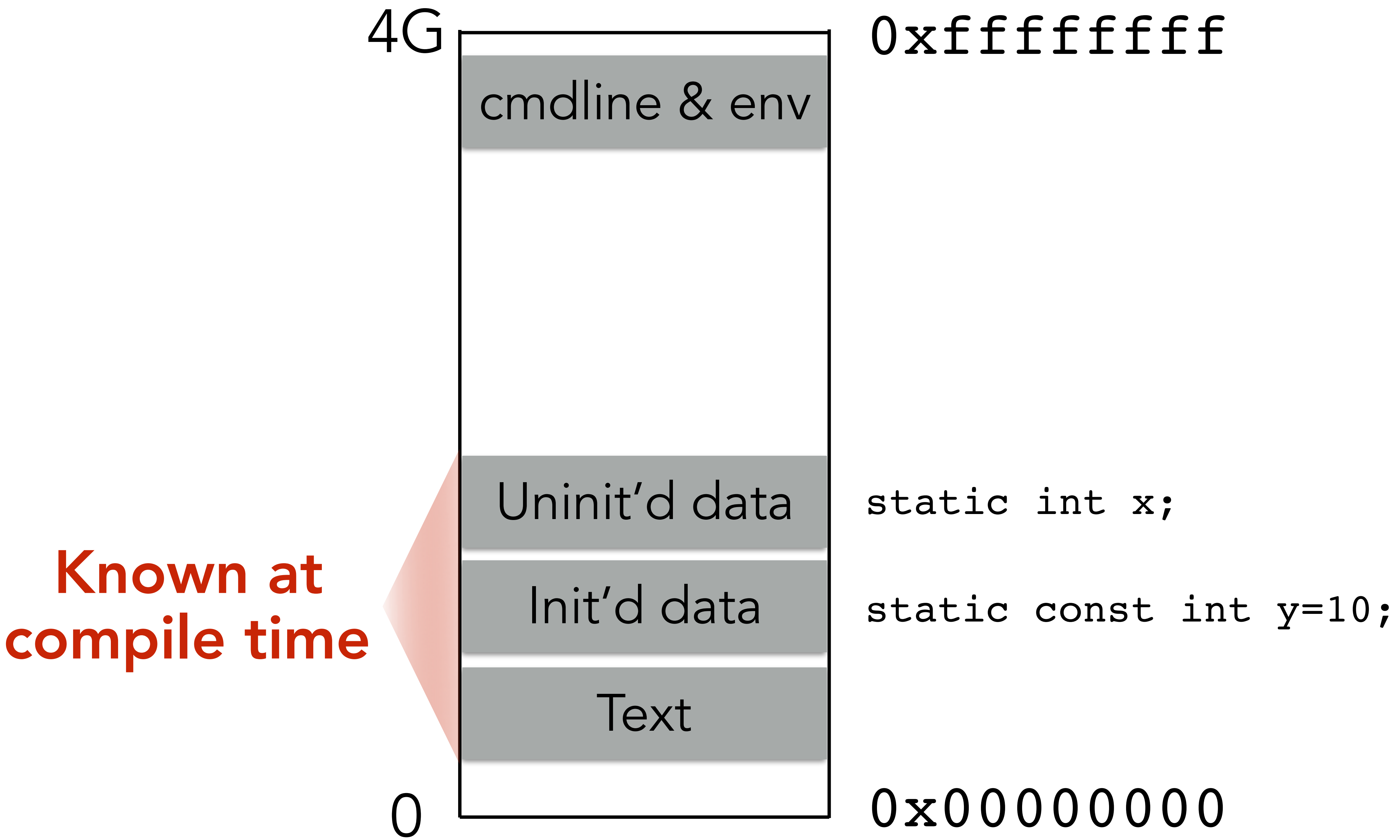
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



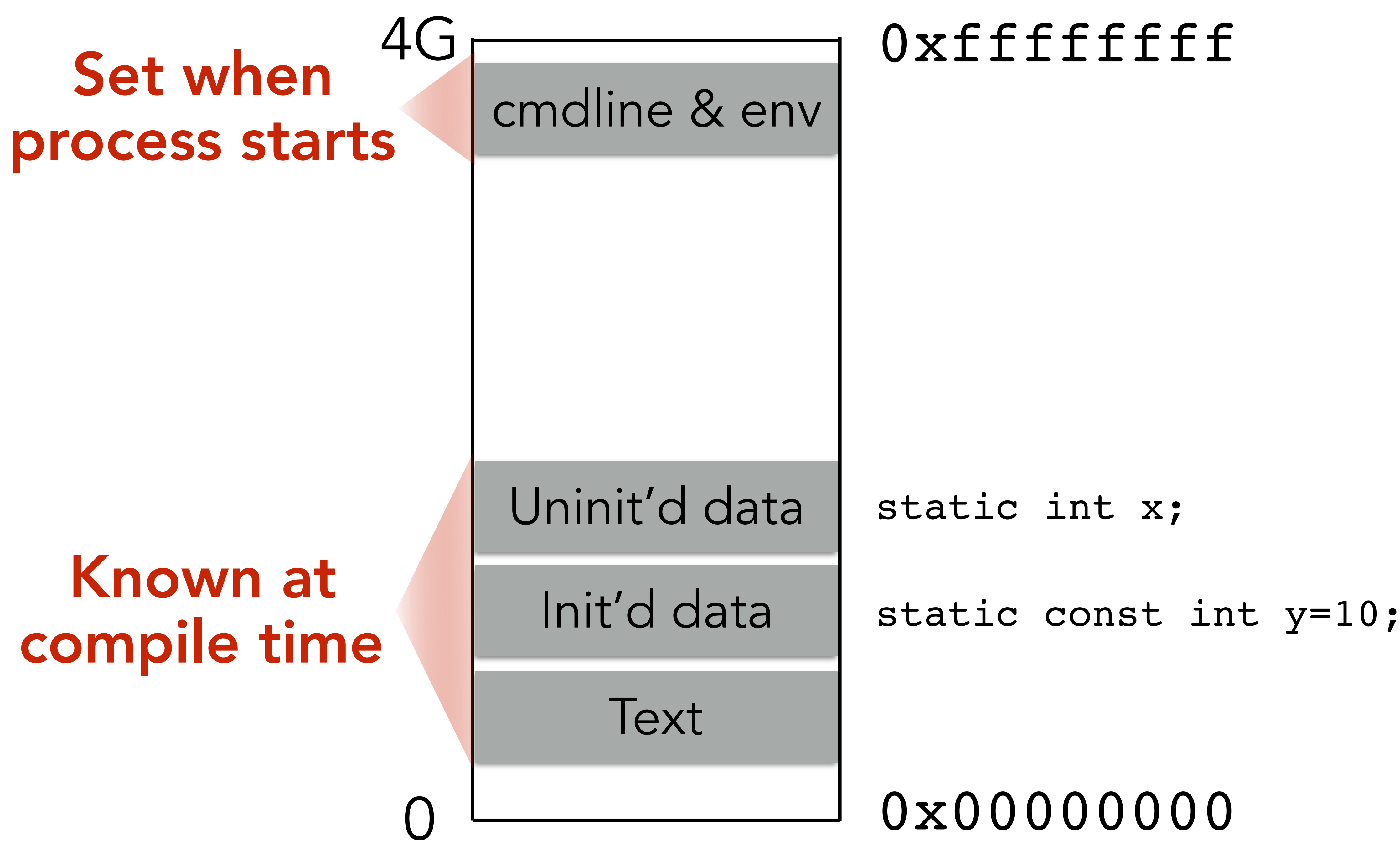
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



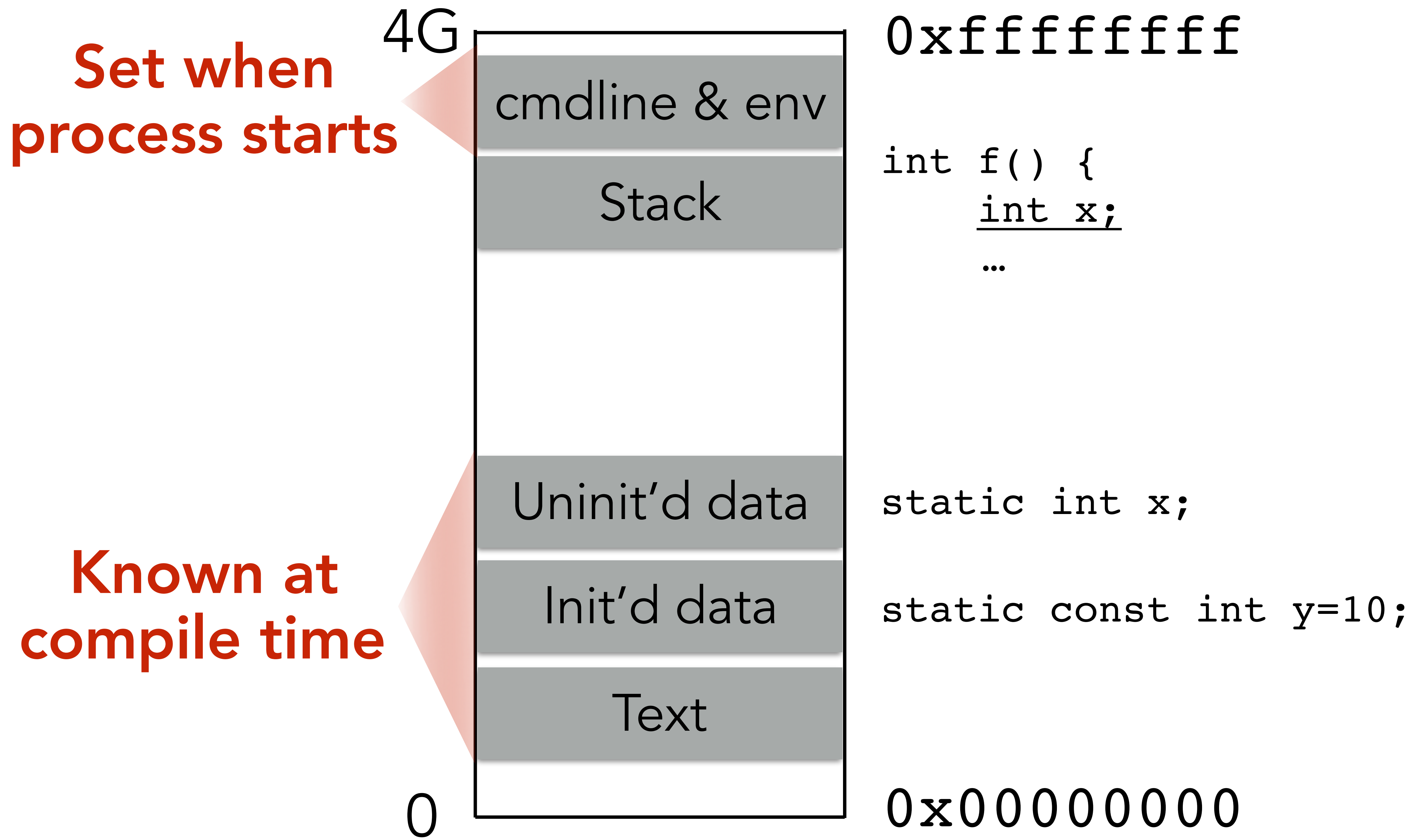
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



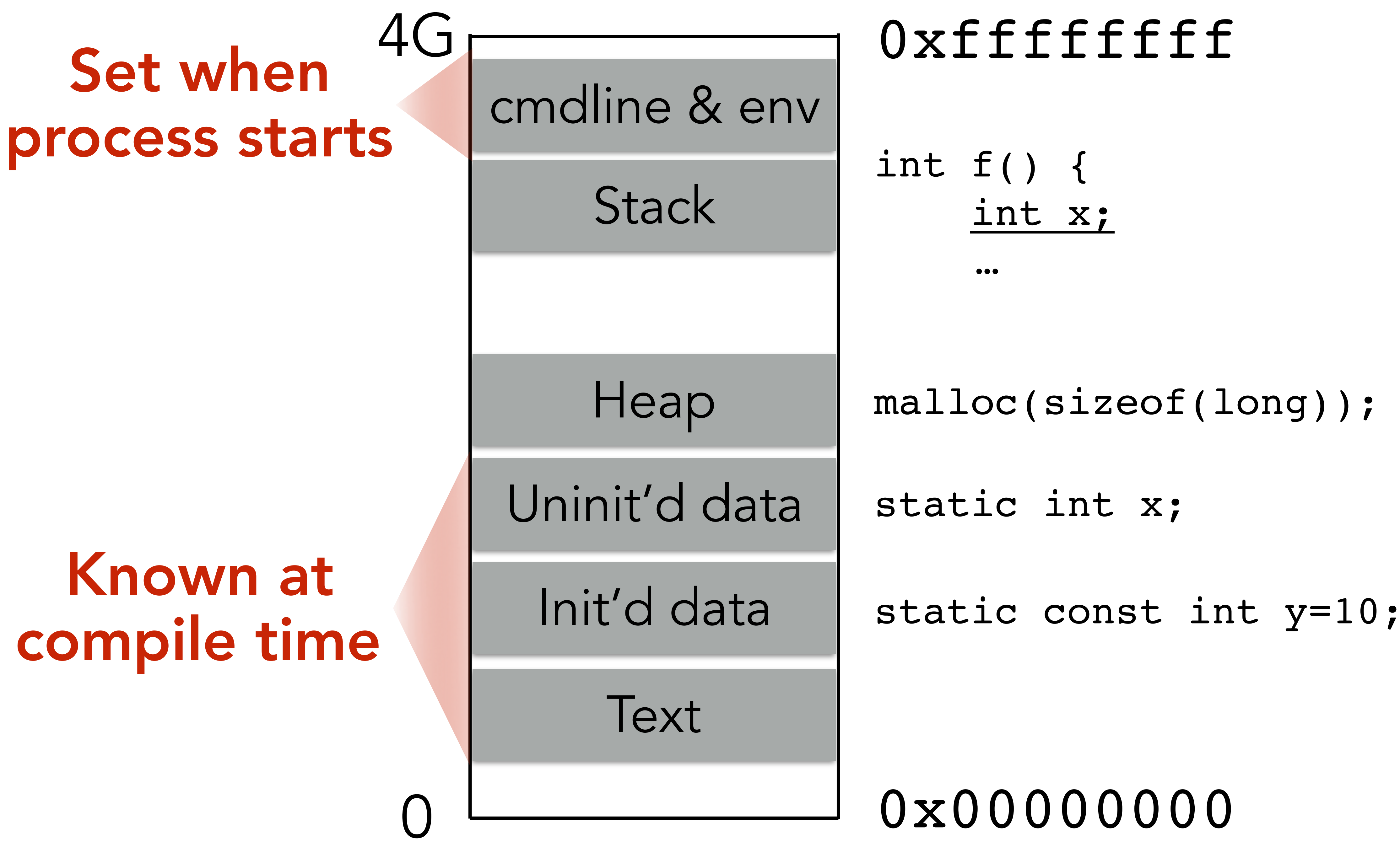
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



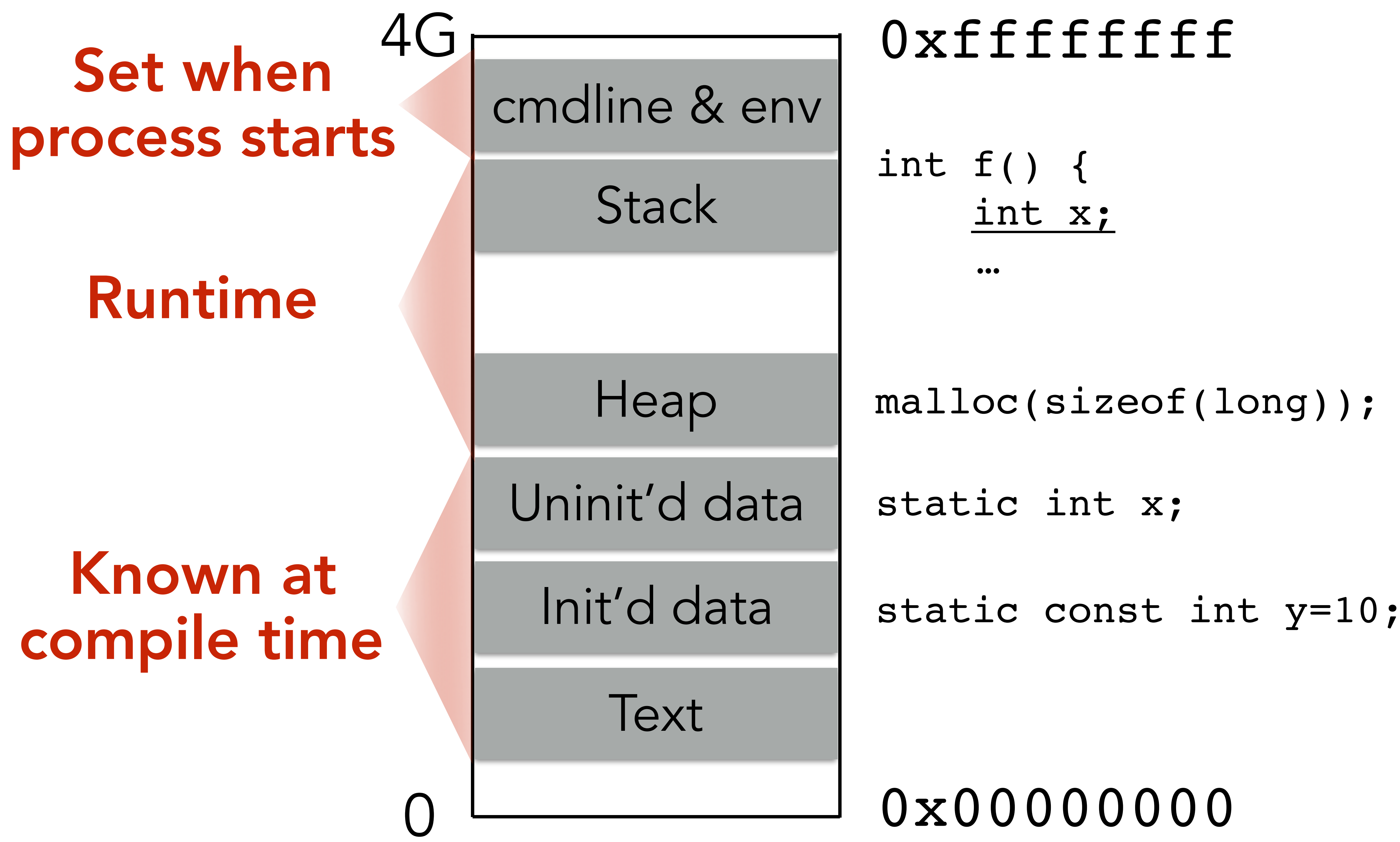
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



DATA'S LOCATION DEPENDS ON HOW IT'S CREATED

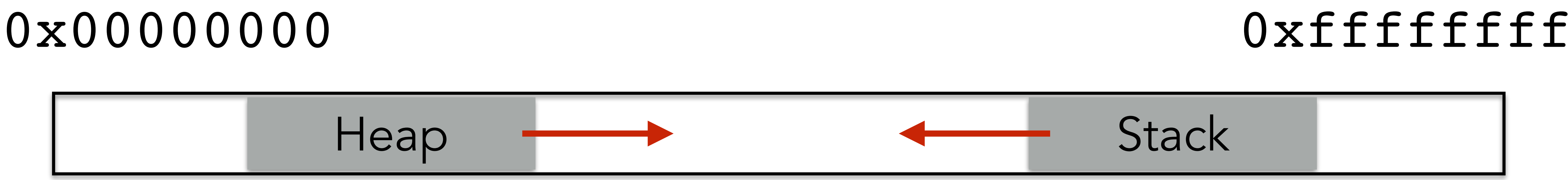


DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

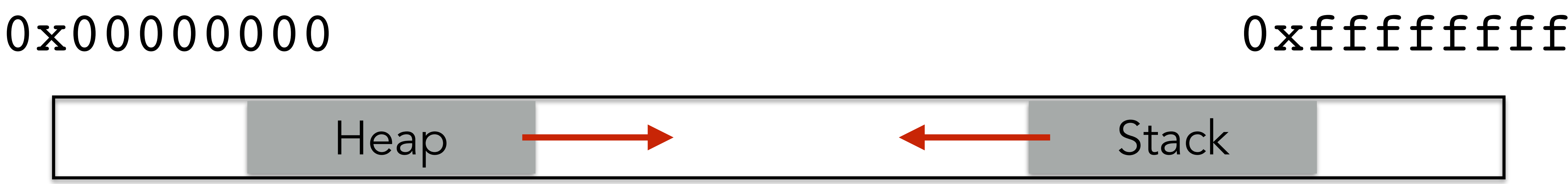
Stack and heap grow in opposite directions



WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

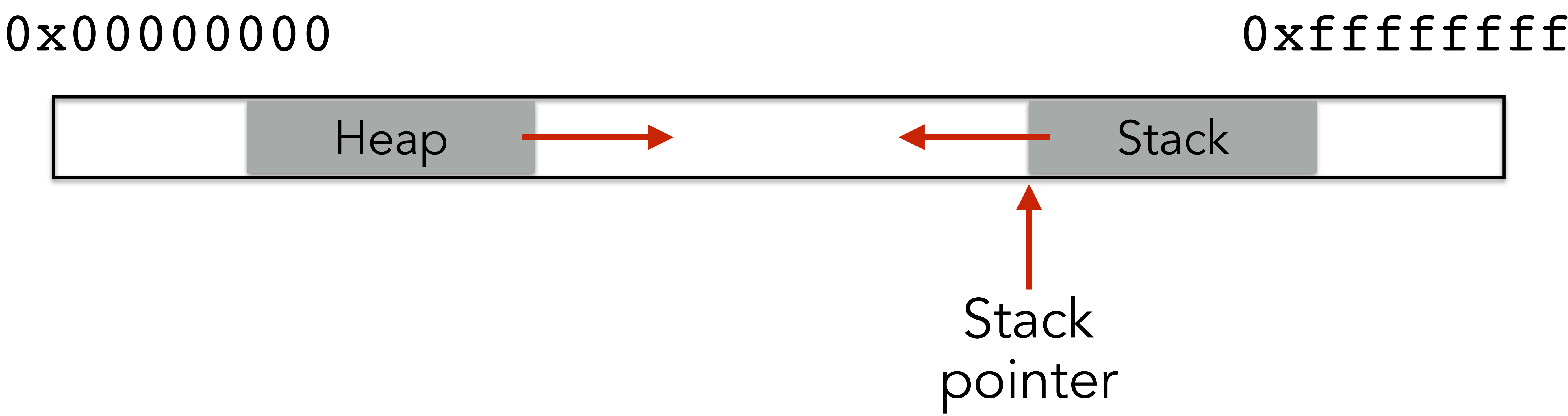
Compiler provides instructions that adjusts the size of the stack at runtime



WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

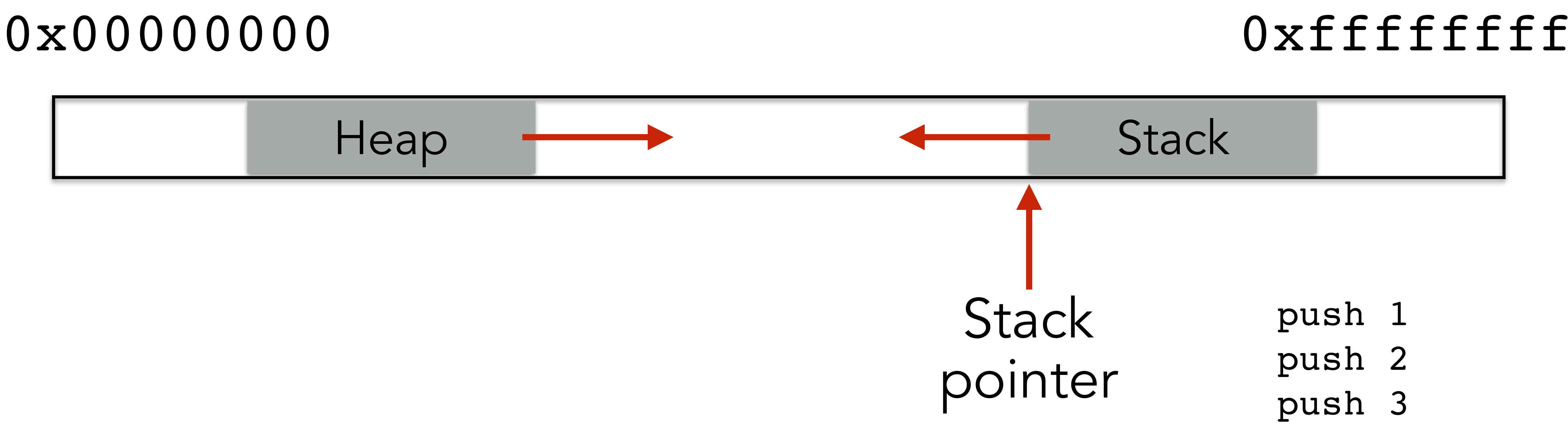
Compiler provides instructions that adjusts the size of the stack at runtime



WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

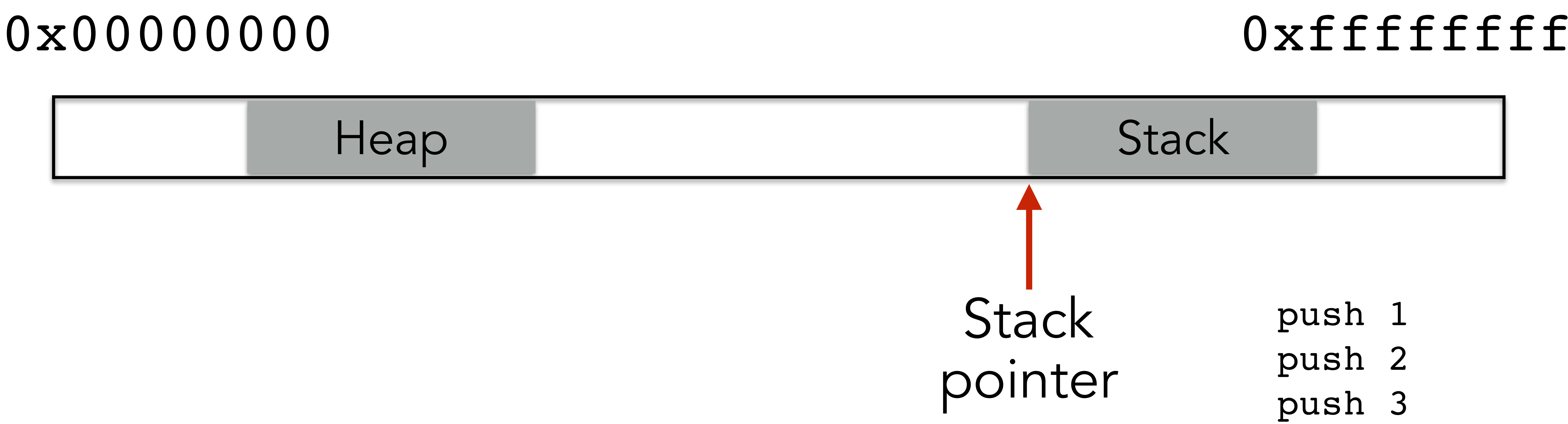
Compiler provides instructions that adjusts the size of the stack at runtime



WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

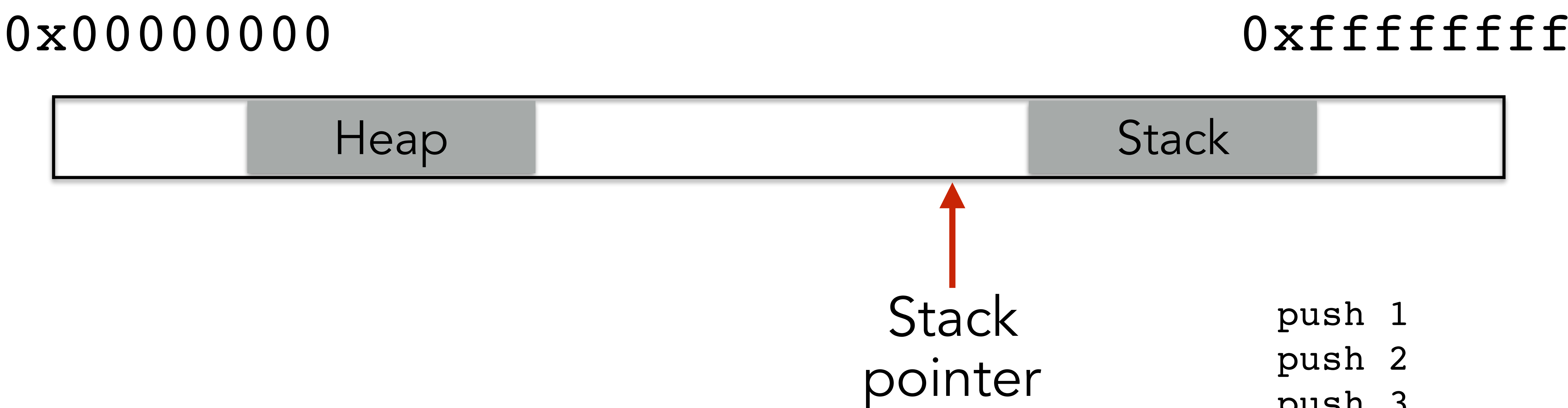
Compiler provides instructions that
adjusts the size of the stack at runtime



WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

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

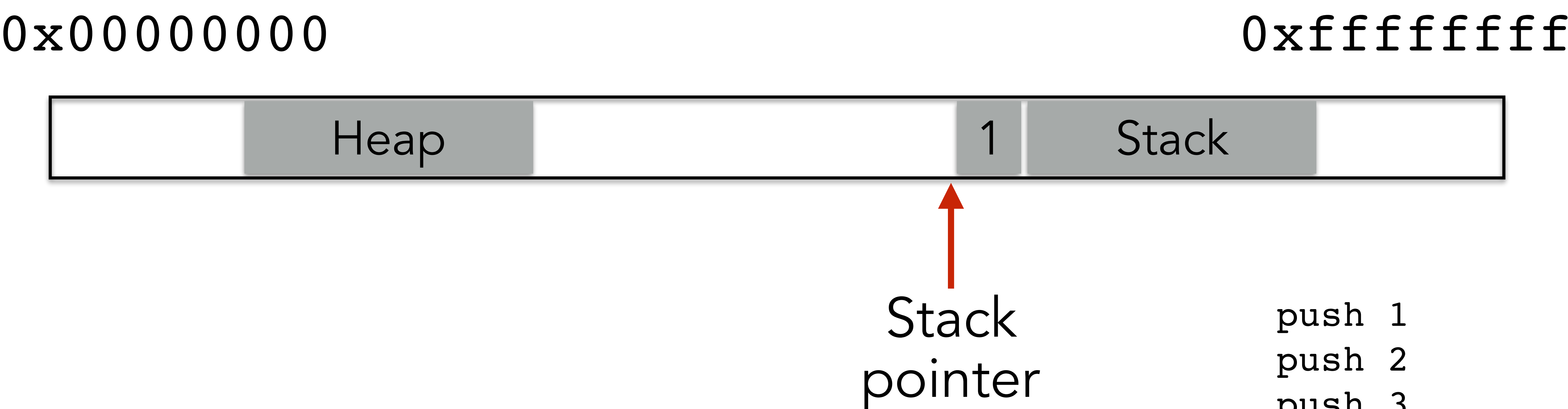



```
push 1
push 2
push 3
```

WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

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

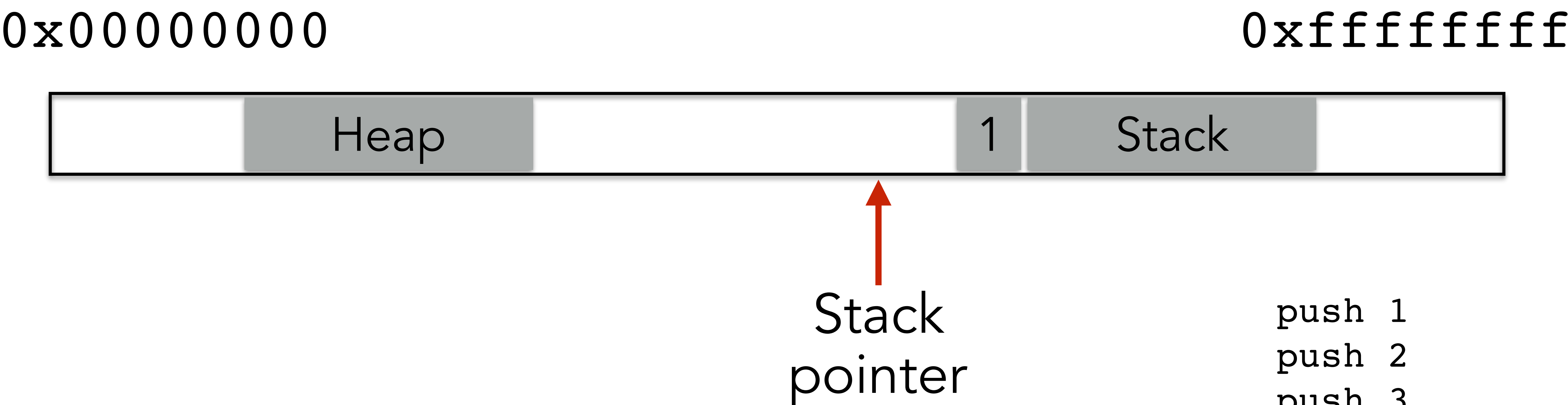



```
push 1
push 2
push 3
```

WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

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

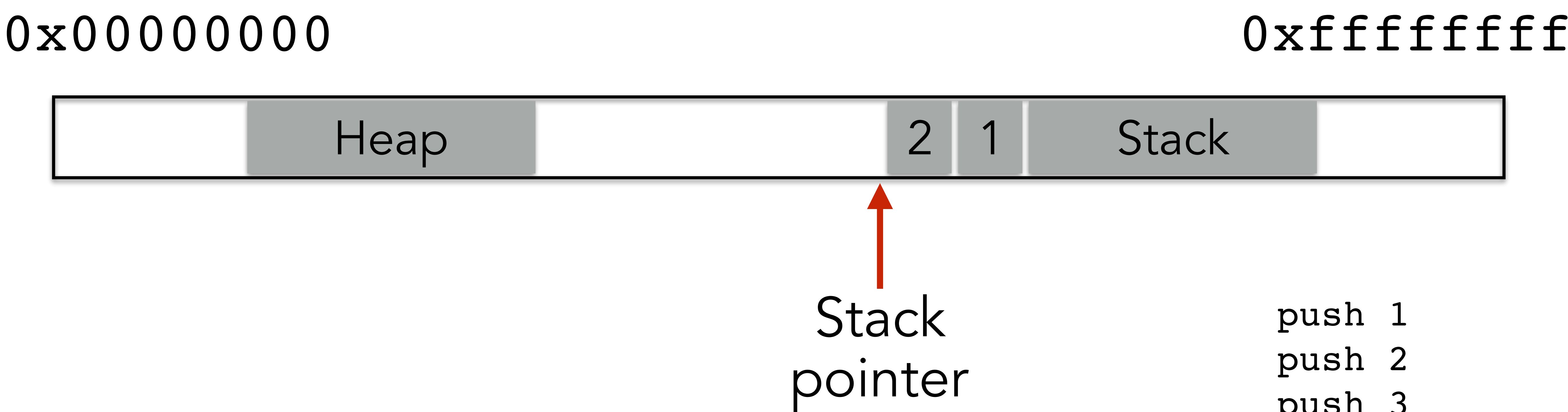



```
push 1
push 2
push 3
```

WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

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

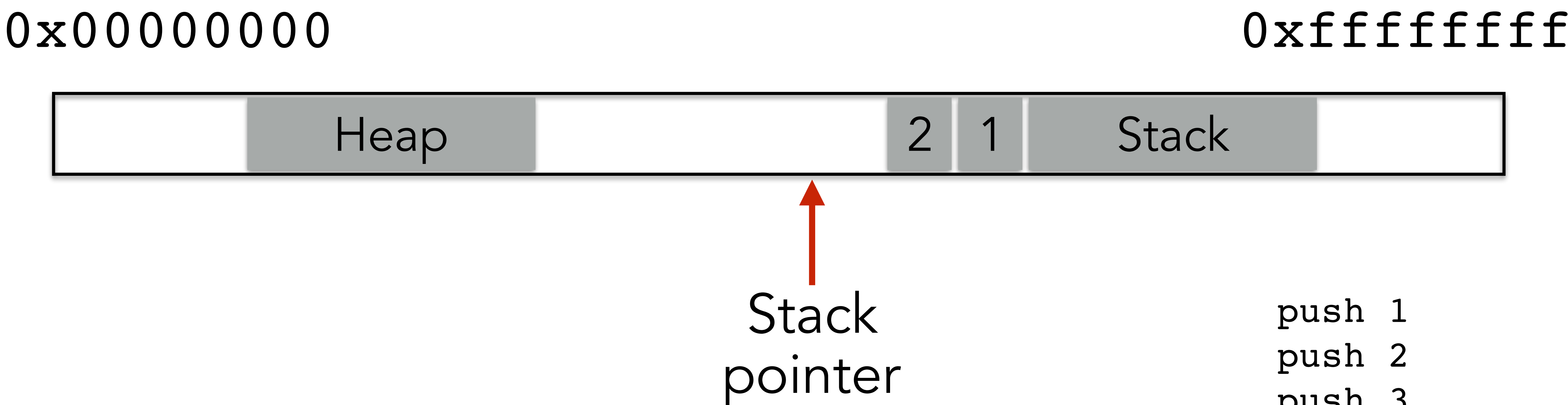



```
push 1
push 2
push 3
```

WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

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

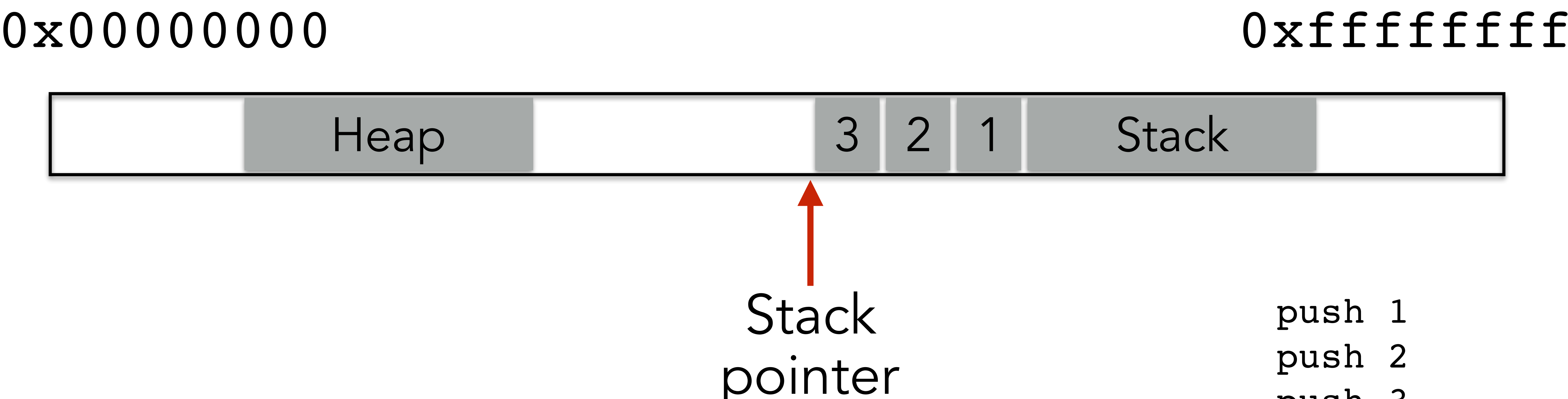



```
push 1
push 2
push 3
```

WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

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

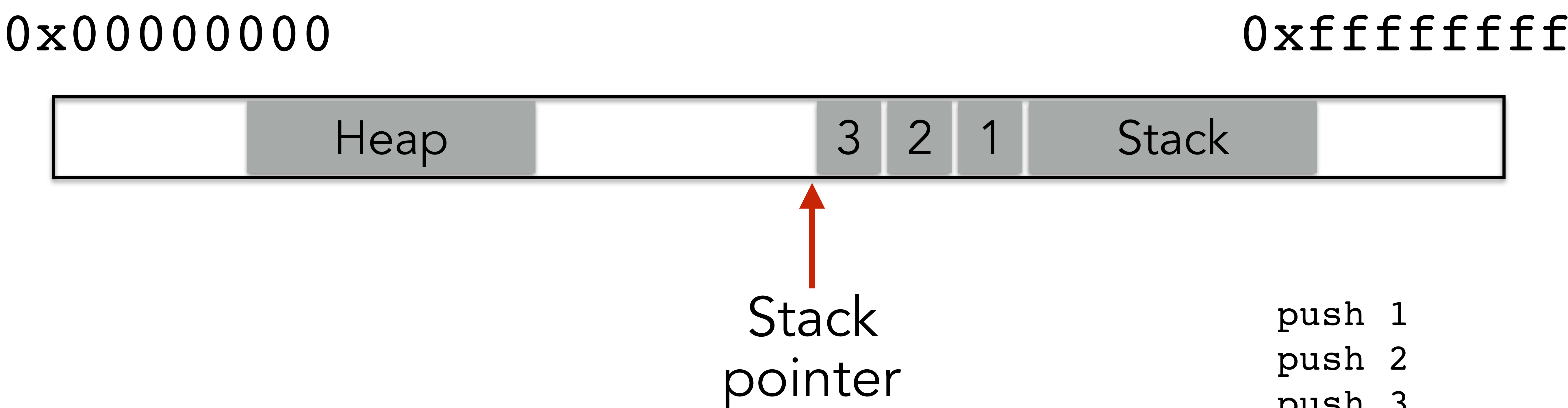



```
push 1
push 2
push 3
```

WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

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

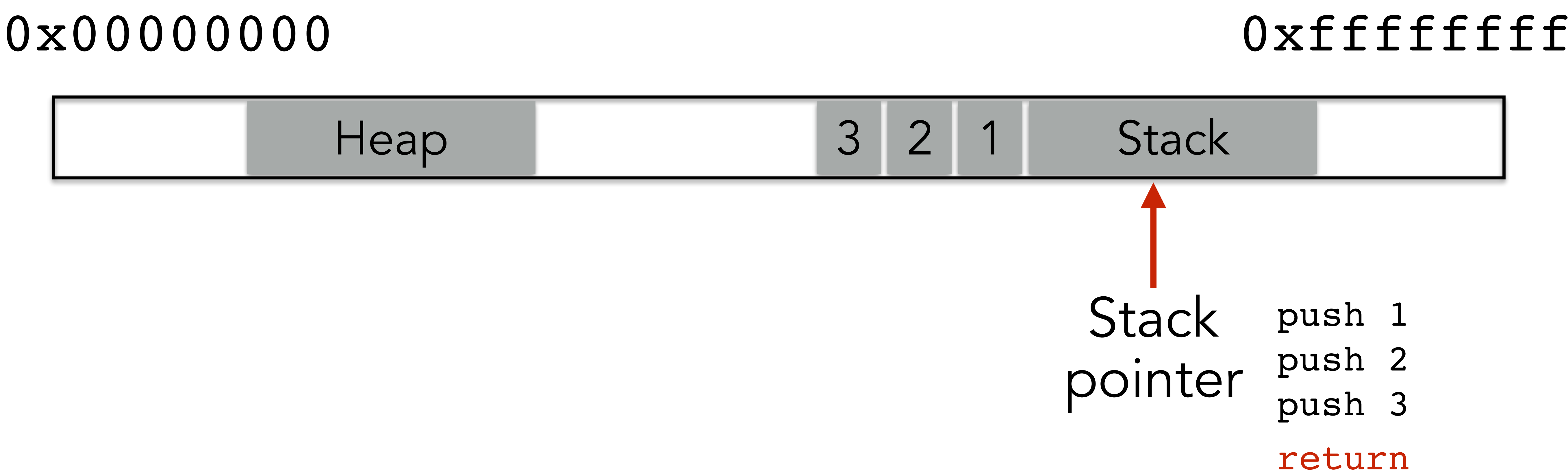



```
push 1
push 2
push 3
return
```

WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

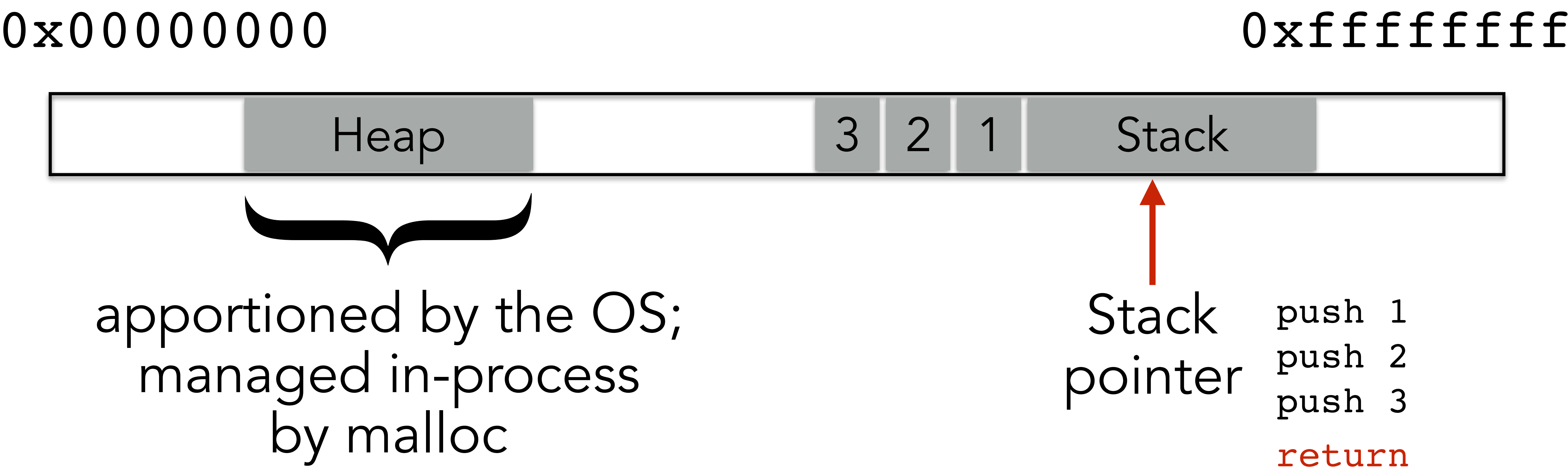
Compiler provides instructions that adjusts the size of the stack at runtime



WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

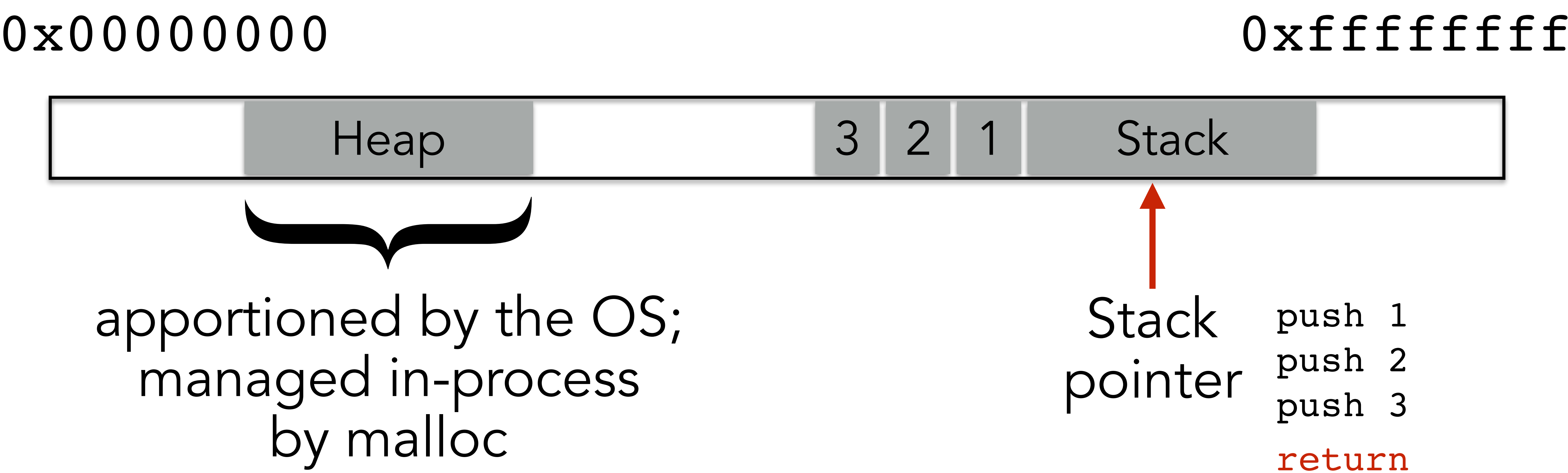
Compiler provides instructions that adjusts the size of the stack at runtime



WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

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



Focusing on the stack for now

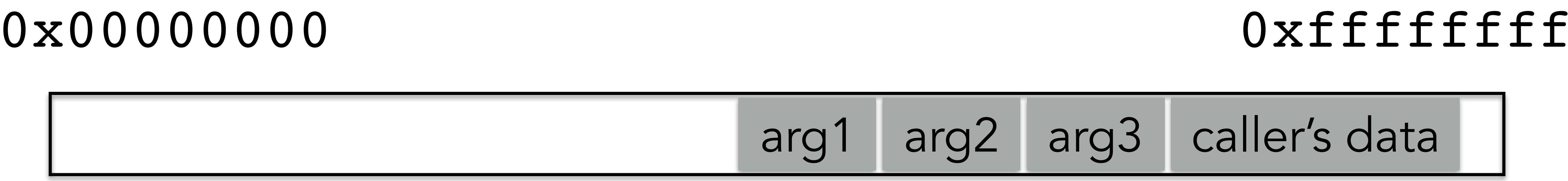
STACK LAYOUT WHEN CALLING FUNCTION

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

0x00000000 0xffffffff

STACK LAYOUT WHEN CALLING FUNCTION

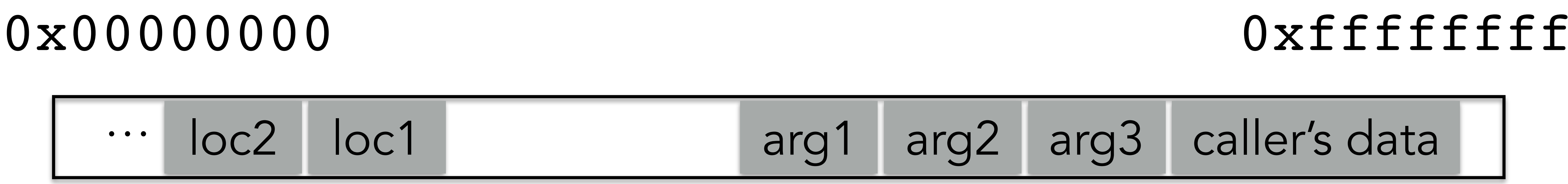
```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```



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;
    ...
}
```

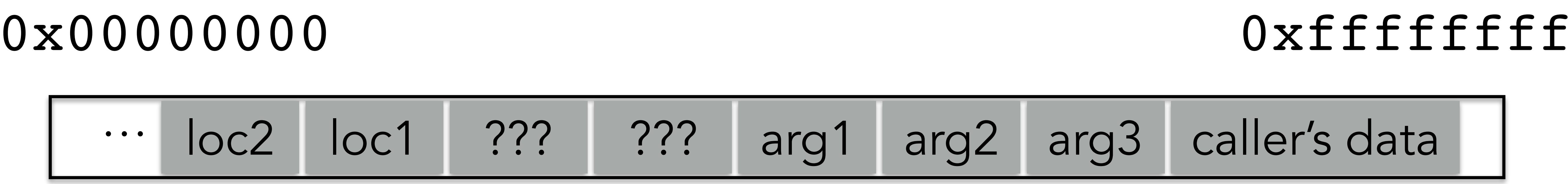


Local variables pushed in the same order as they appear in the code

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

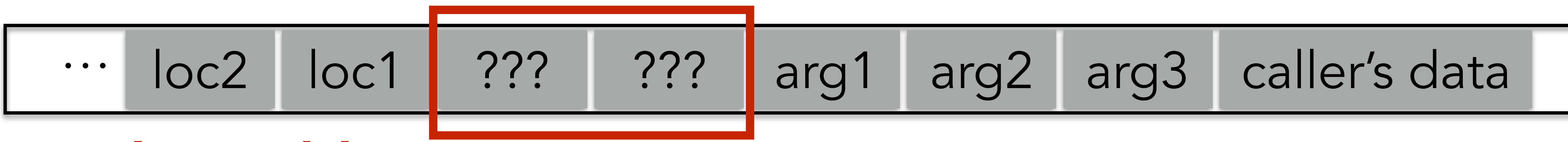
Arguments pushed in reverse order of code

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

0x00000000 0xffffffff

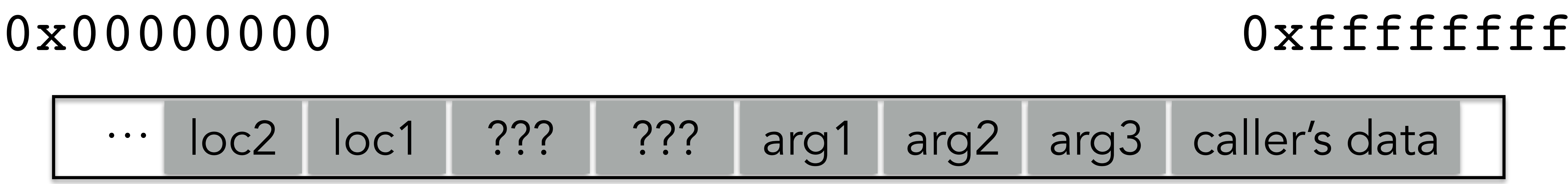


Local variables pushed in the same order as they appear in the code

Arguments pushed in reverse order of code

ACCESSING VARIABLES

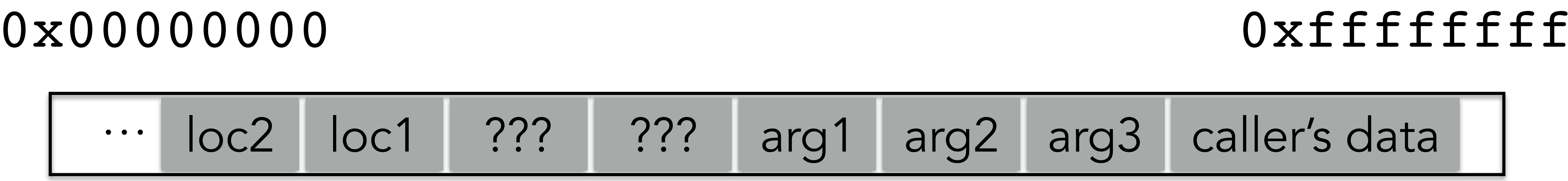
```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    loc2++;
}
```



ACCESSING VARIABLES

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

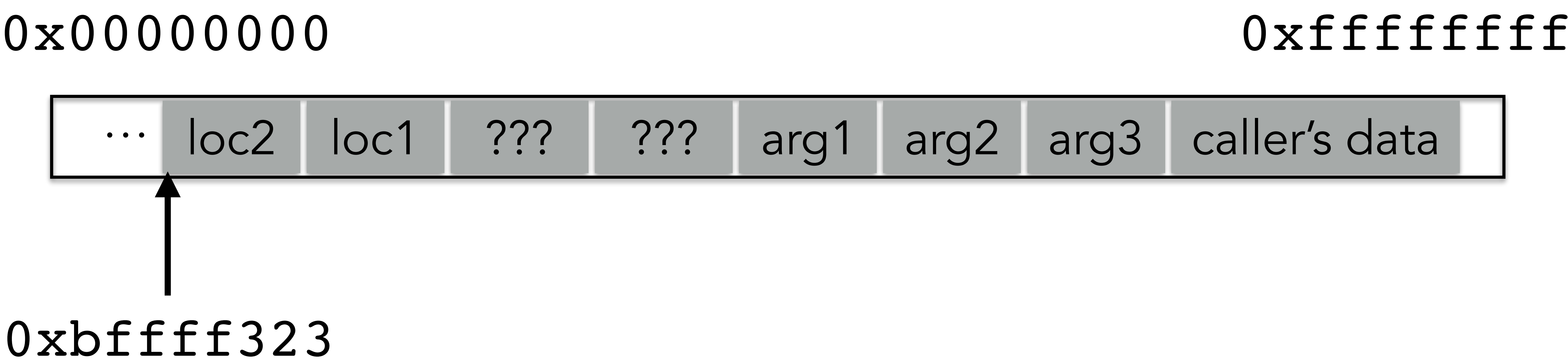
Q: Where is (this) loc2?



ACCESSING VARIABLES

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

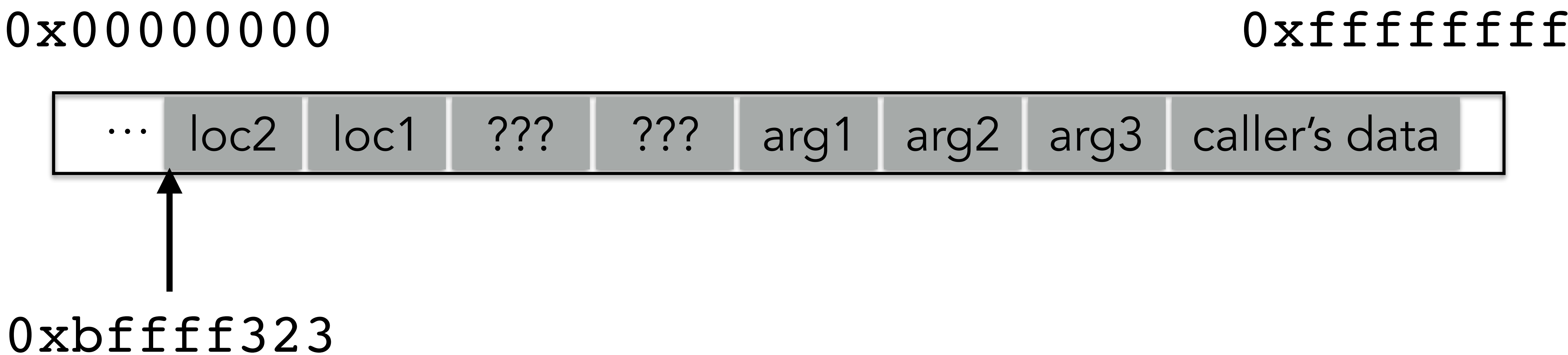
Q: Where is (this) loc2?



ACCESSING VARIABLES

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

Q: Where is (this) loc2?

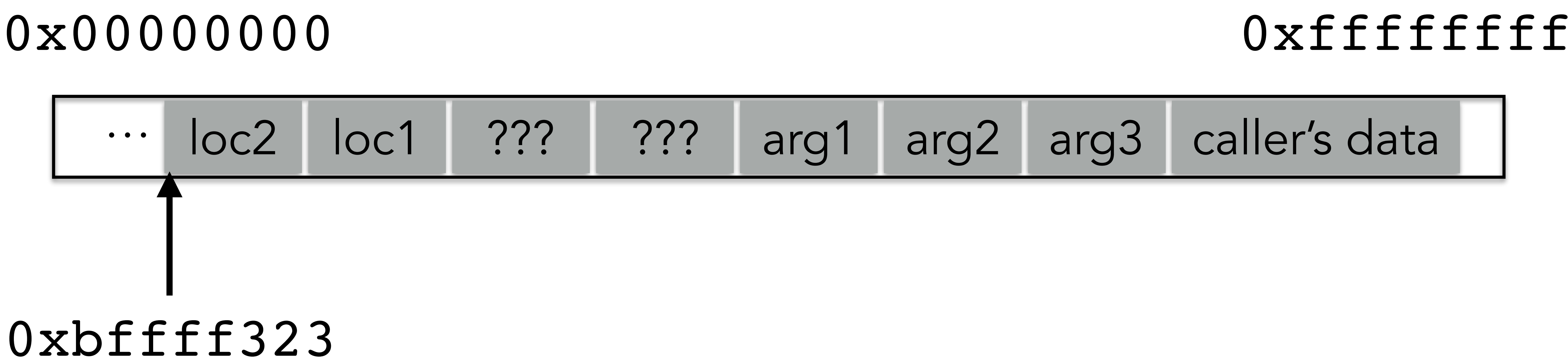


Undecidable at compile time

ACCESSING VARIABLES

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

Q: Where is (this) loc2?



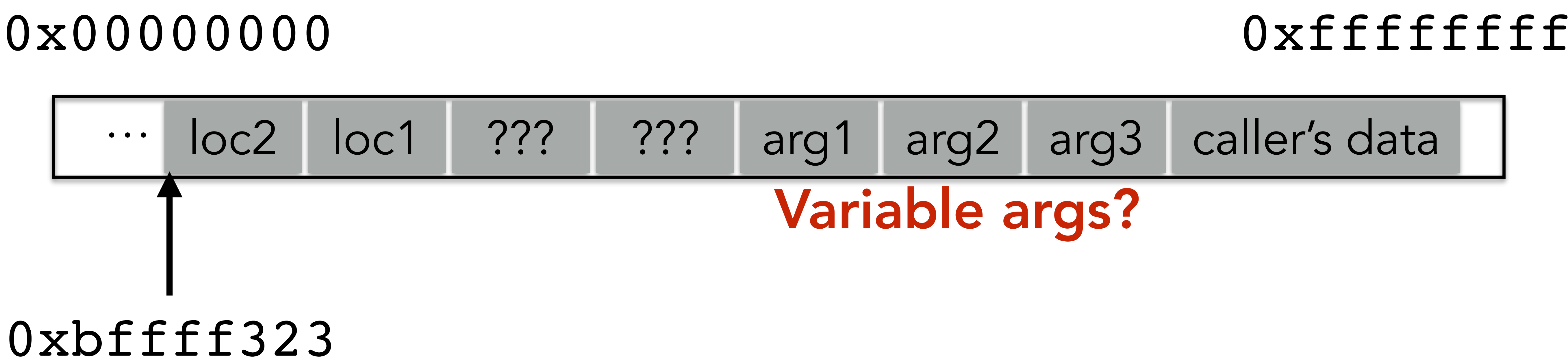
Undecidable at compile time

- I don't know where loc2 is,

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

ACCESSING VARIABLES

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

Q: Where is (this) loc2?

0x00000000 0xffffffff

…	loc2	loc1	???	???	arg1	arg2	arg3	caller's data
	4B	4B	4B	4B	Variable args?			

0xbffff323

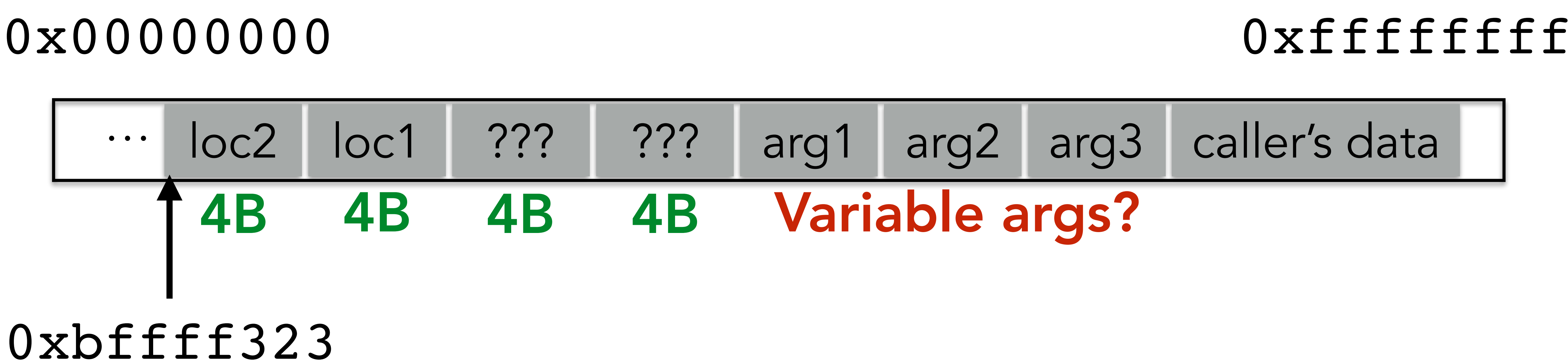
Undecidable at compile time

- I don't know where loc2 is,
- and I don't know how many args

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?

0x00000000 0xffffffff

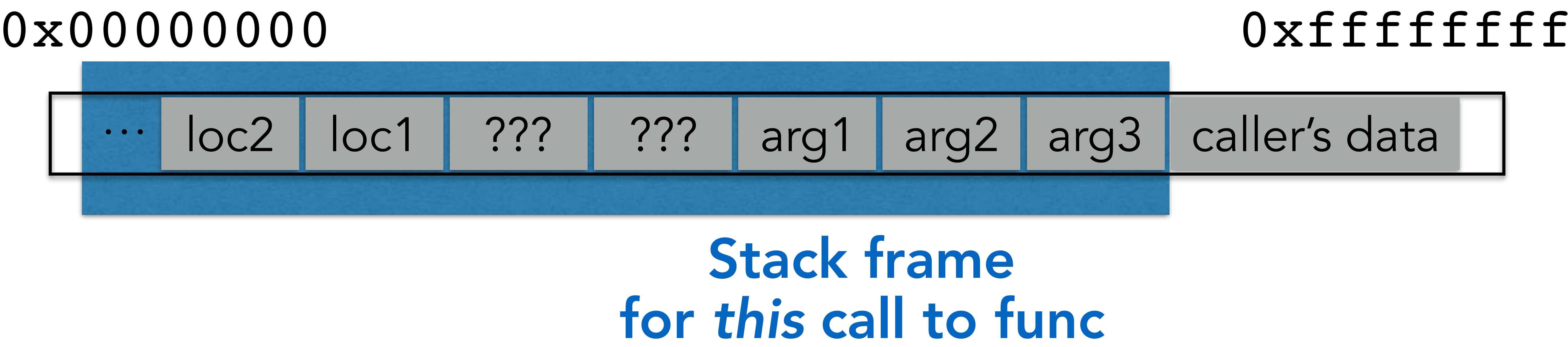
…	loc2	loc1	???	???	arg1	arg2	arg3	caller's data

- 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?

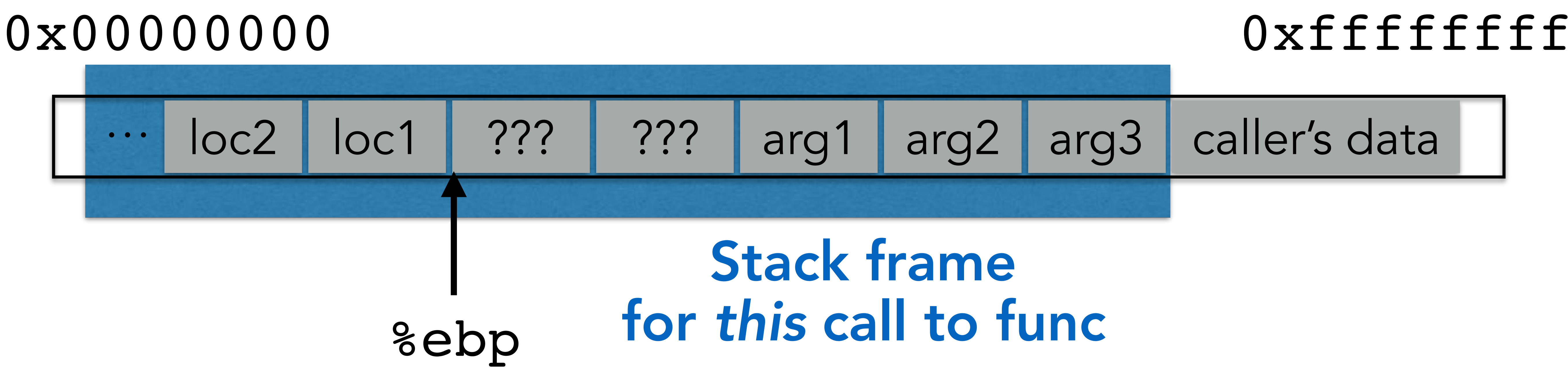


- I don't know where loc2 is,
- and I don't know how many args
- *but* loc2 is *always* 8B before "???"s

ACCESSING VARIABLES

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

Q: Where is (this) loc2?



Frame pointer

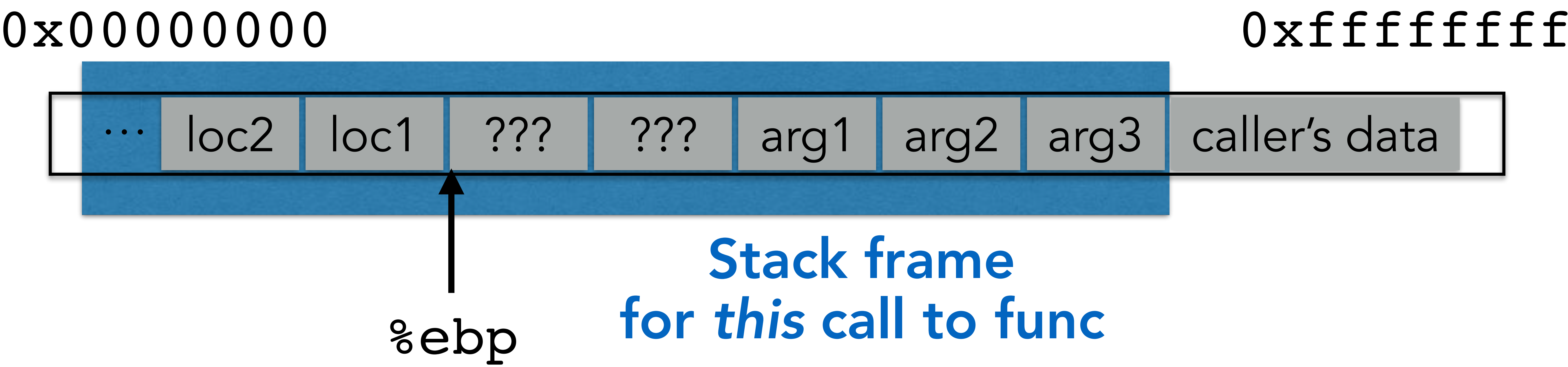
- I don't know where loc2 is,
- and I don't know how many args
- *but* loc2 is *always* 8B before "???"s

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

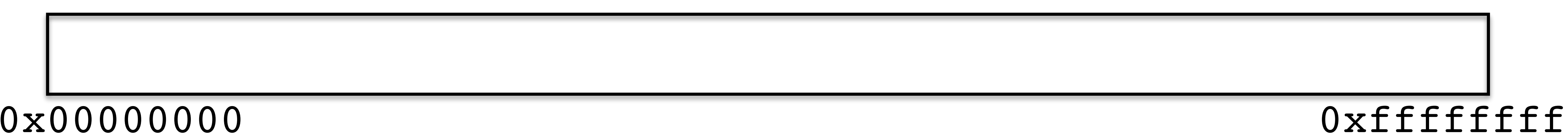
- I don't know where loc2 is,
- and I don't know how many args
- *but* loc2 is *always* 8B before "???"s

NOTATION

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

NOTATION

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



NOTATION

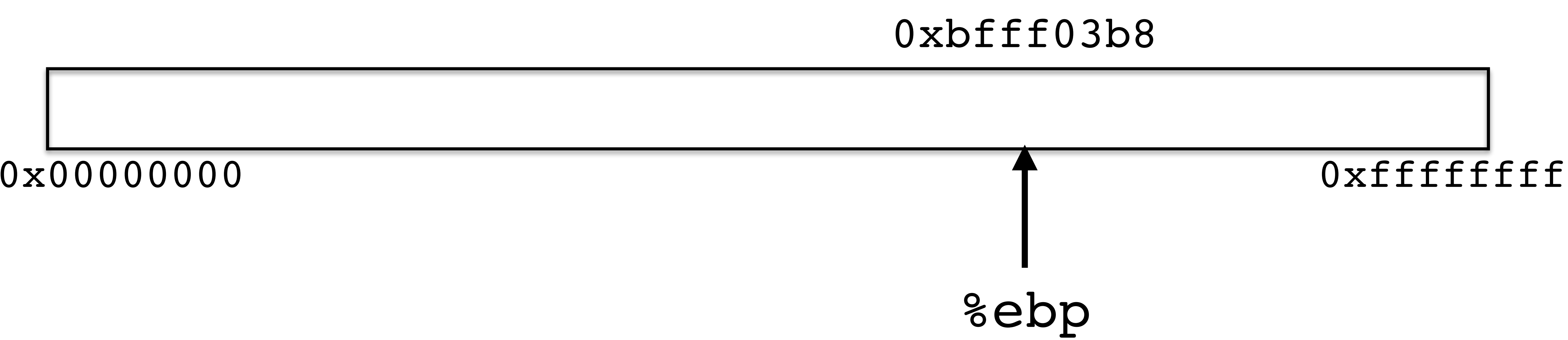
`0xbfff03b8` `%ebp` A memory address

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

`0x00000000` `0xffffffff`

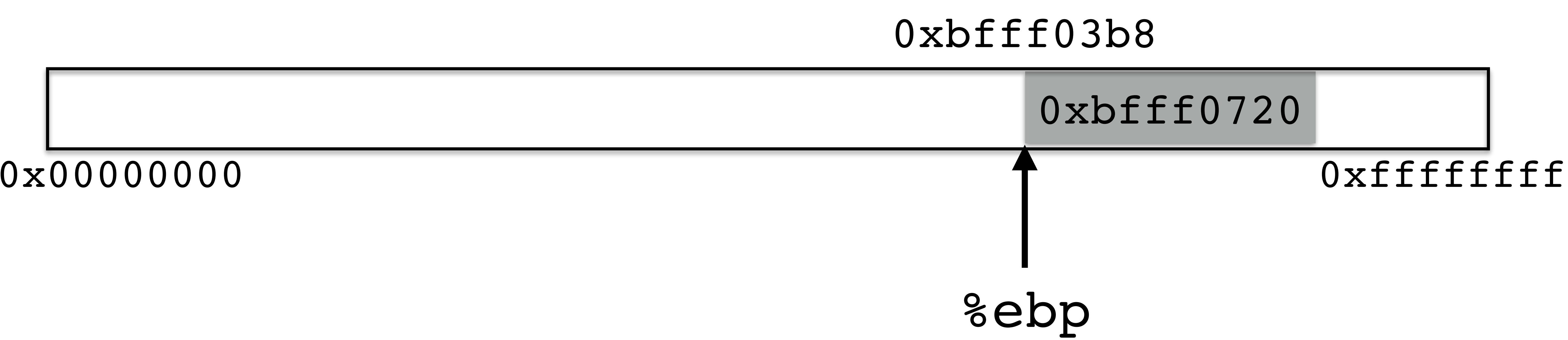
NOTATION

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



NOTATION

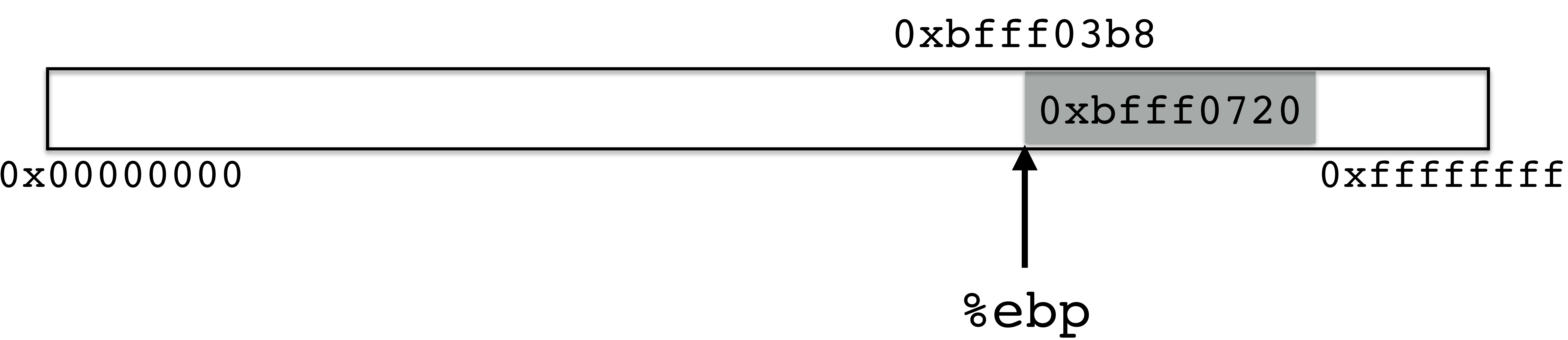
`0xbfff03b8`	`%ebp`	A memory address
`0xbfff0720`	`(%ebp)`	The value at memory address %ebp (like dereferencing a pointer)



NOTATION

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

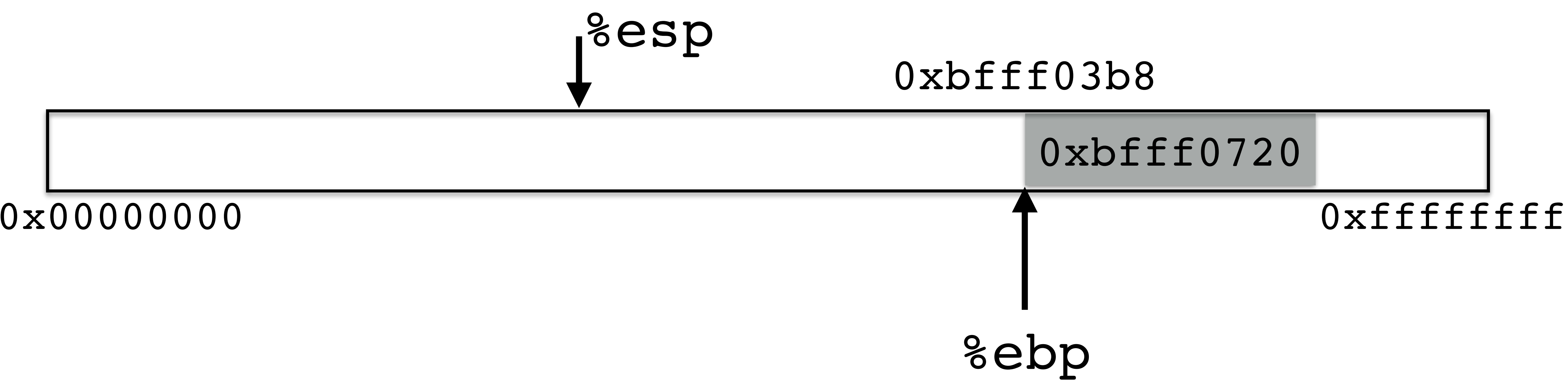
```
pushl %ebp
```



NOTATION

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

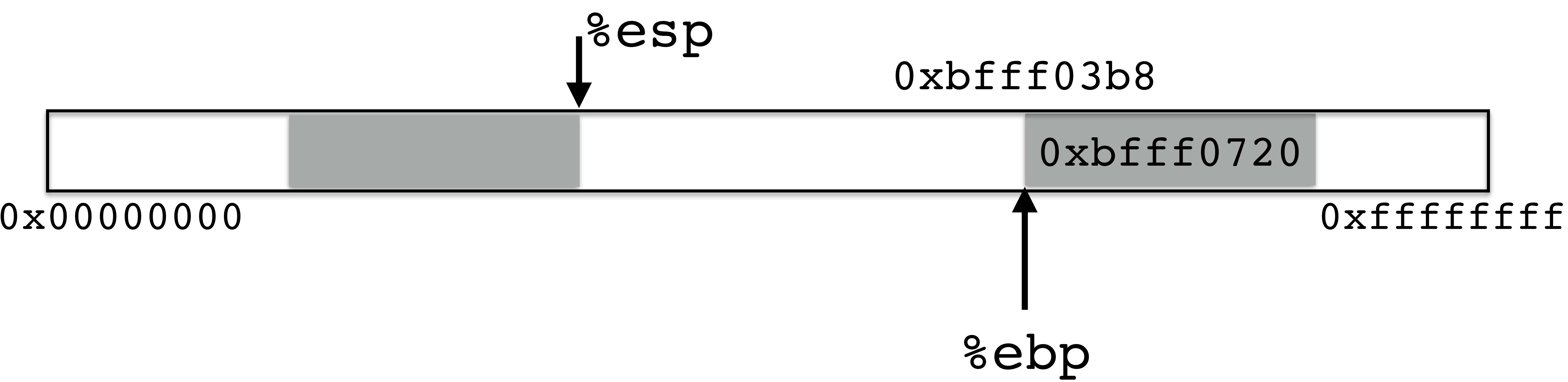
`pushl %ebp`



NOTATION

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

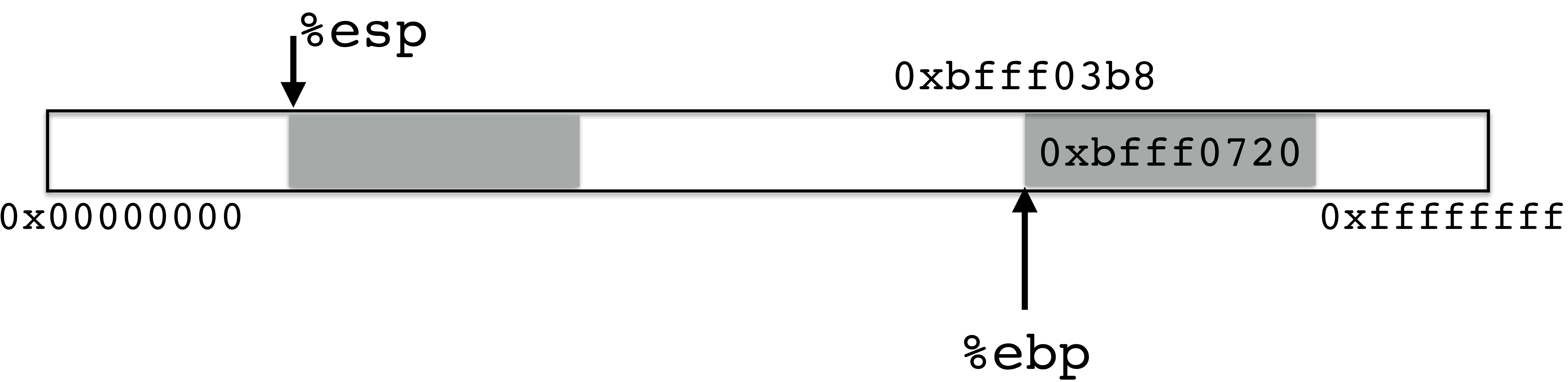
`pushl %ebp`



NOTATION

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

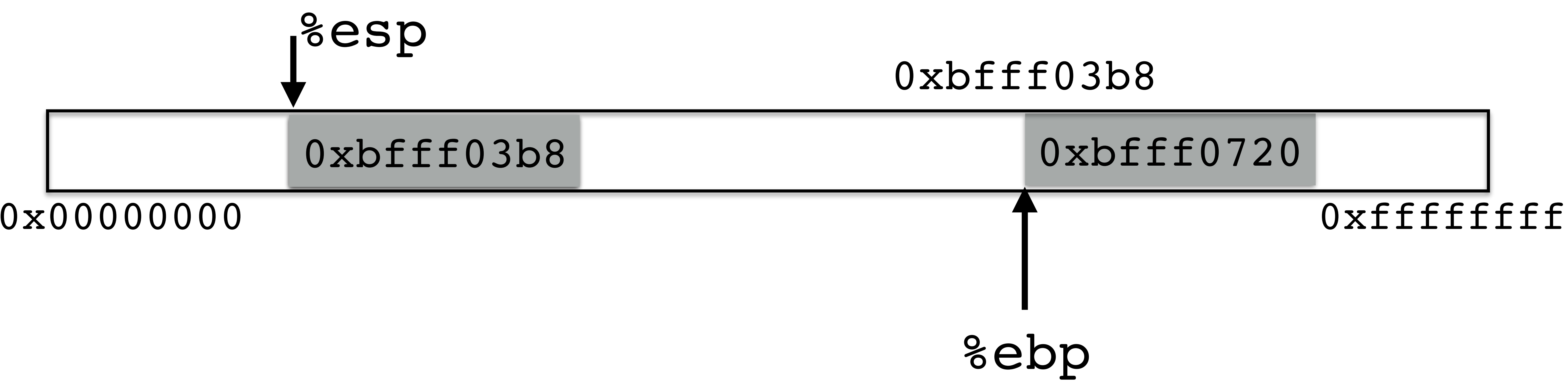
```
pushl %ebp
```



NOTATION

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

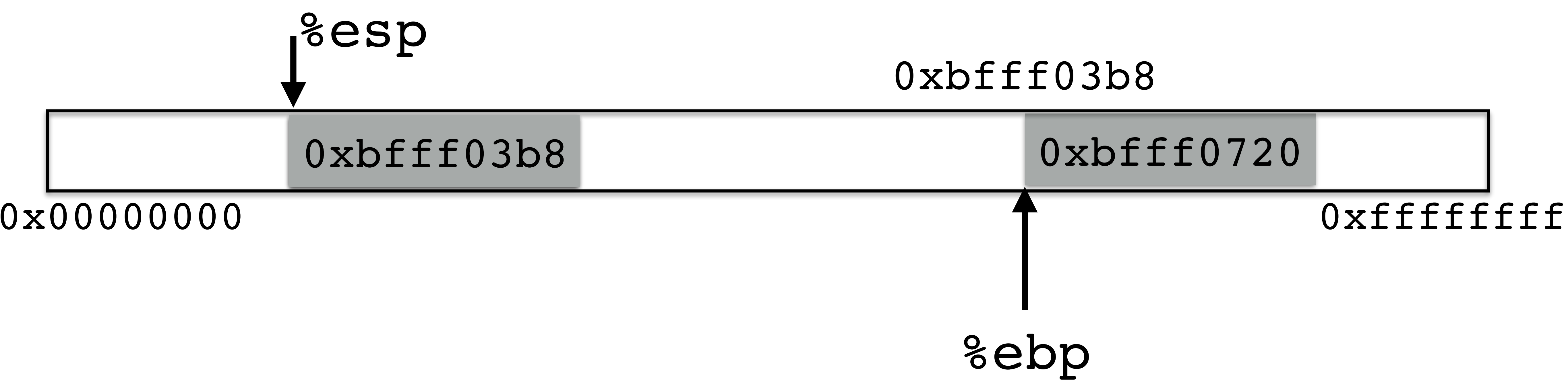
```
pushl %ebp
```



NOTATION

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

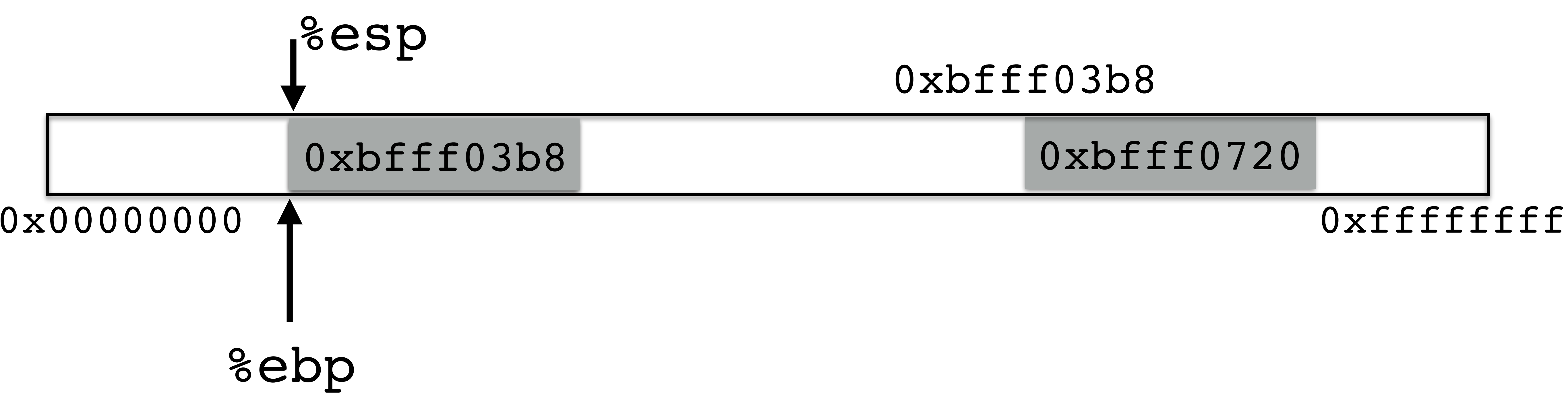
```
pushl %ebp
movl  %esp %ebp  /* %ebp = %esp */
```



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 */
```

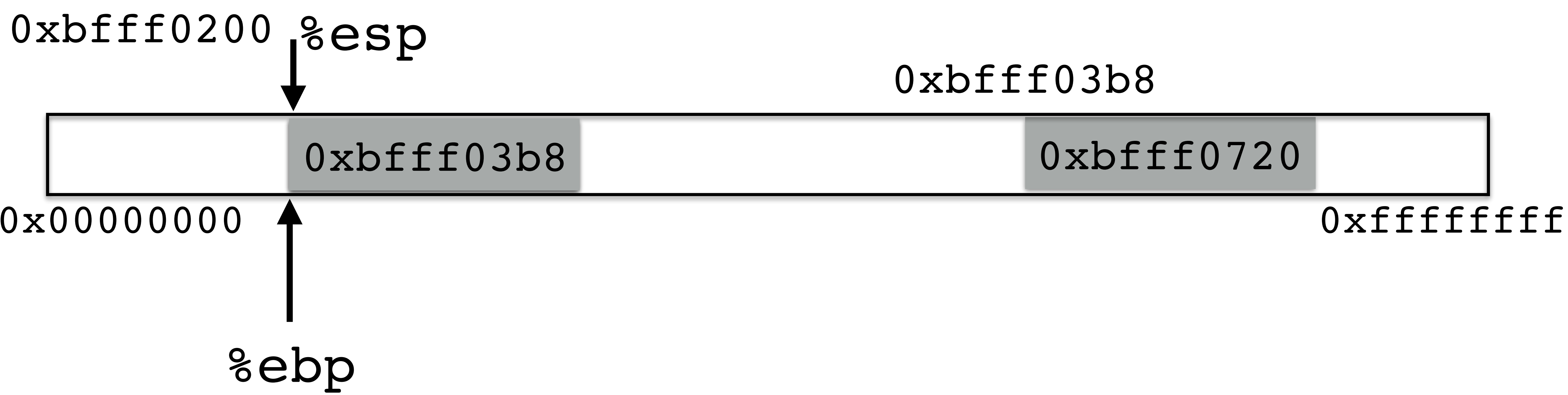


NOTATION

`0xbfff03b8` `%ebp` A memory address

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

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

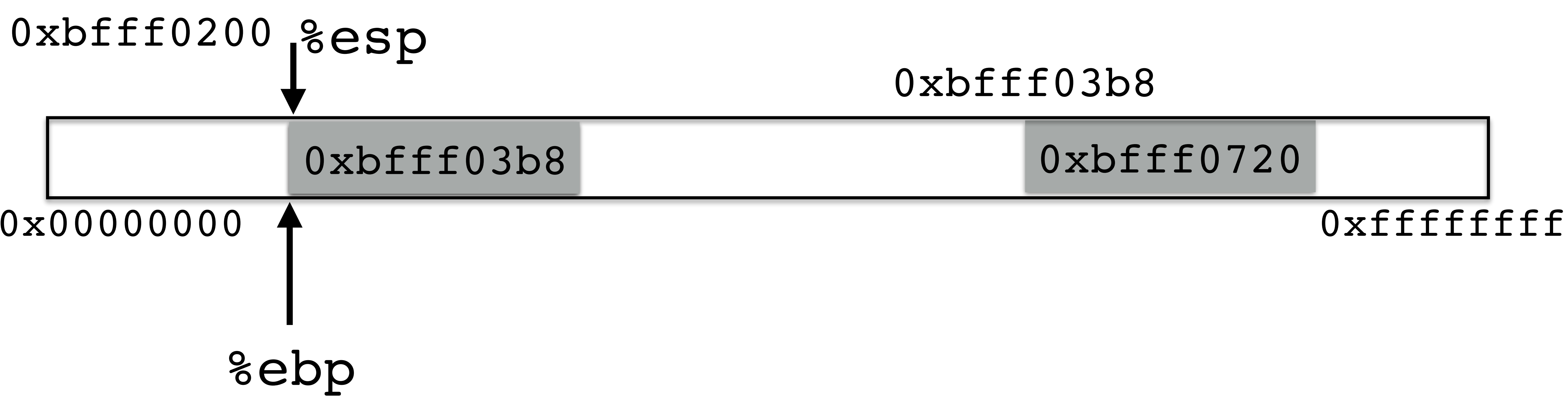


NOTATION

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

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

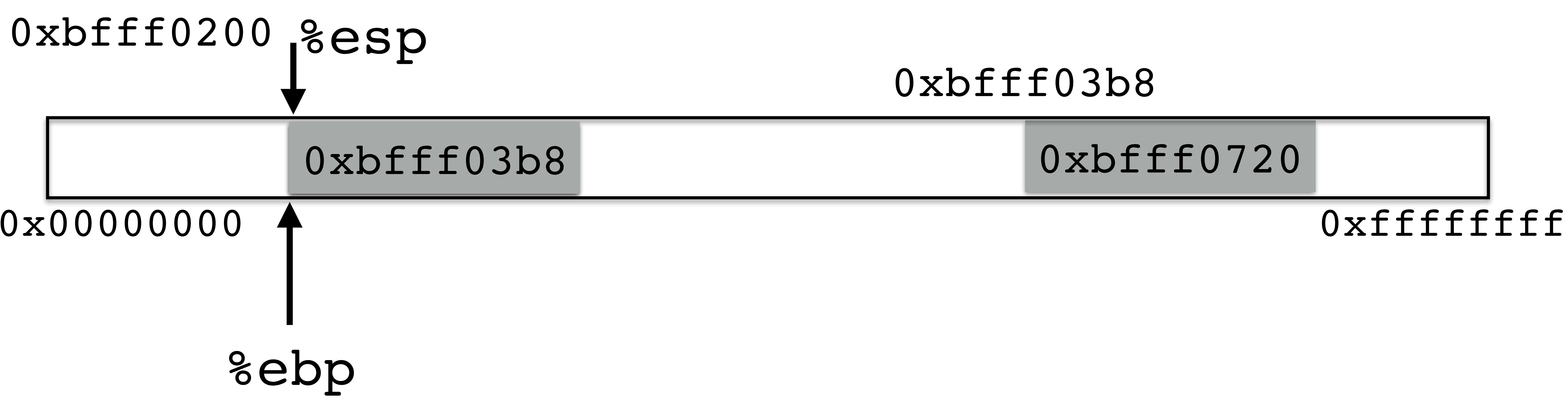
```
pushl %ebp
movl  %esp %ebp /* %ebp = %esp */
```



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 */
```

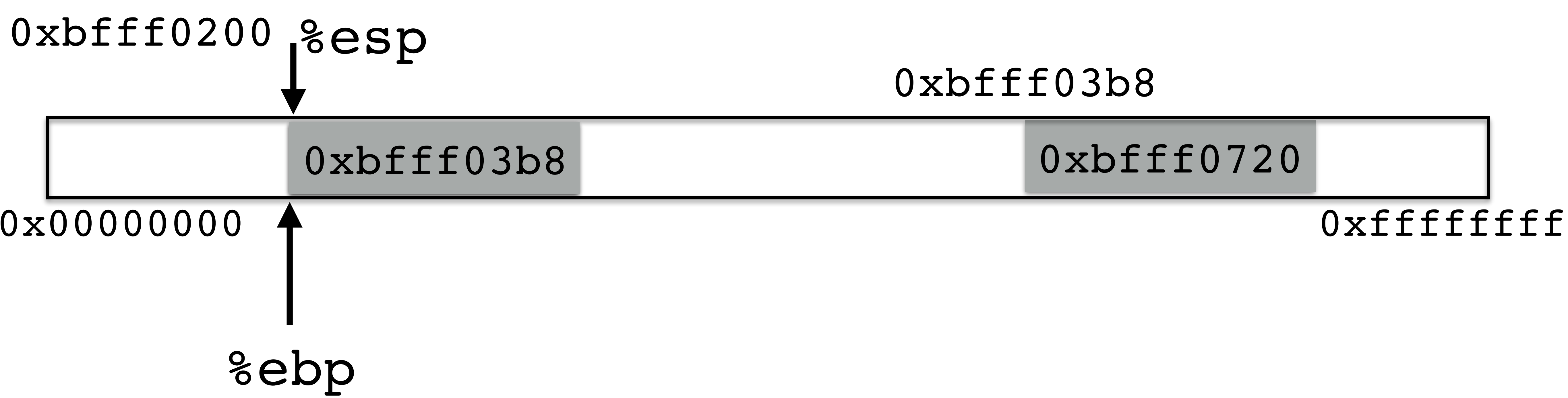


NOTATION

~~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) */
```

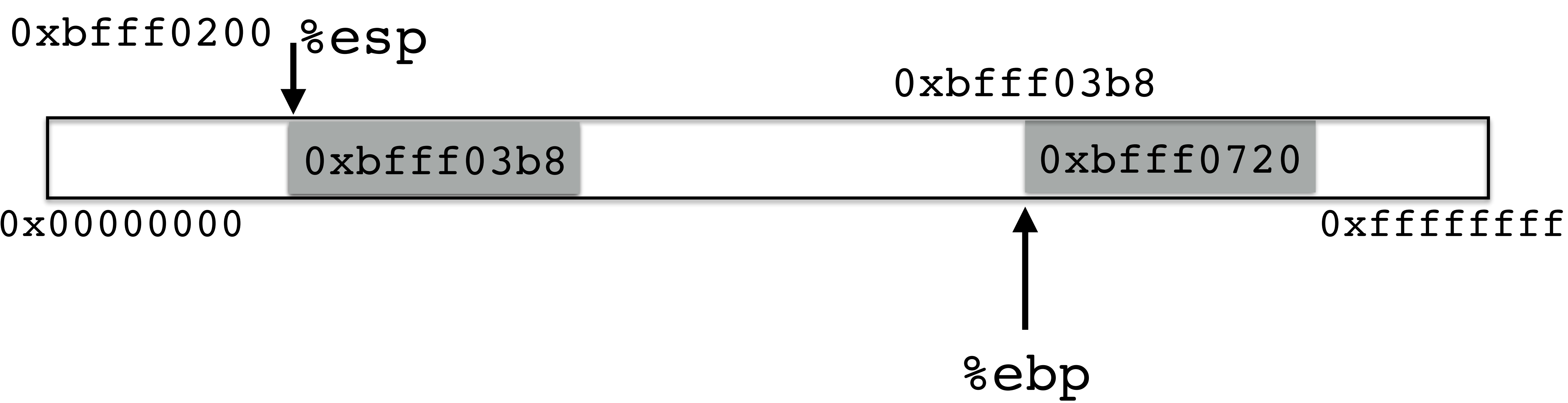


NOTATION

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

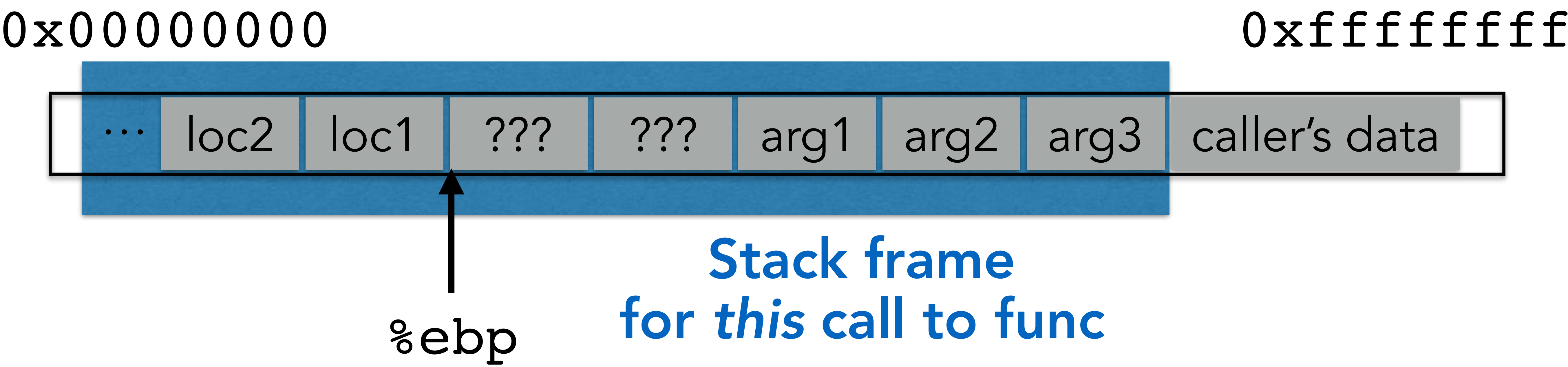
~~0xbfff0720~~
0xbfff03b8 | (%ebp) | The value at memory address %ebp (like dereferencing a pointer)

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



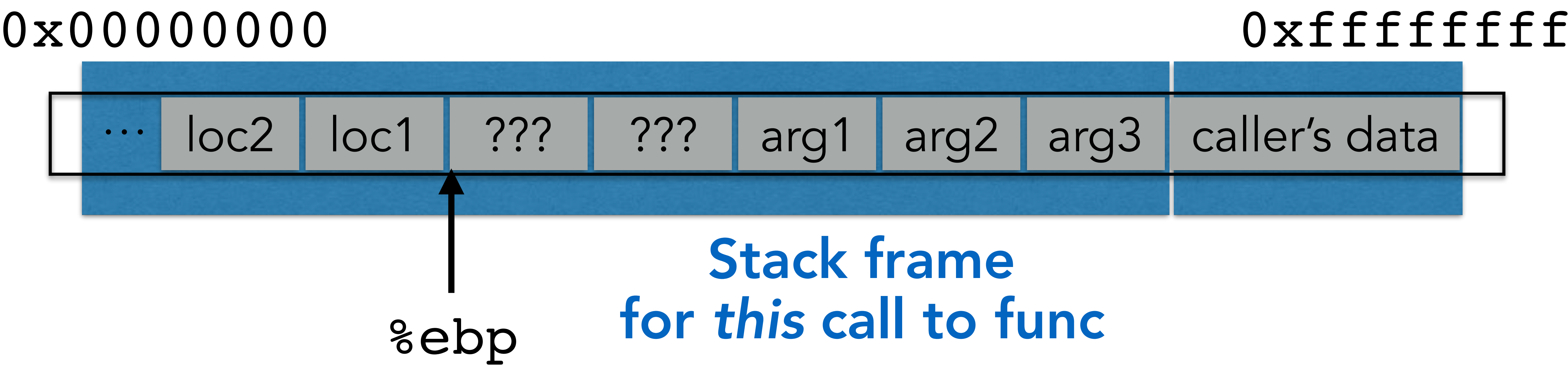
RETURNING FROM FUNCTIONS

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



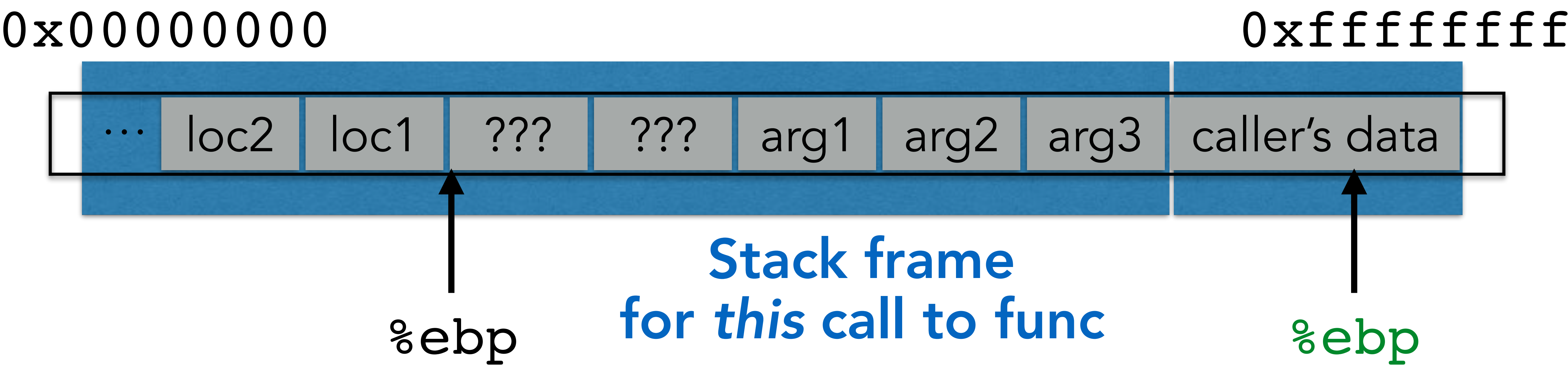
RETURNING FROM FUNCTIONS

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



RETURNING FROM FUNCTIONS

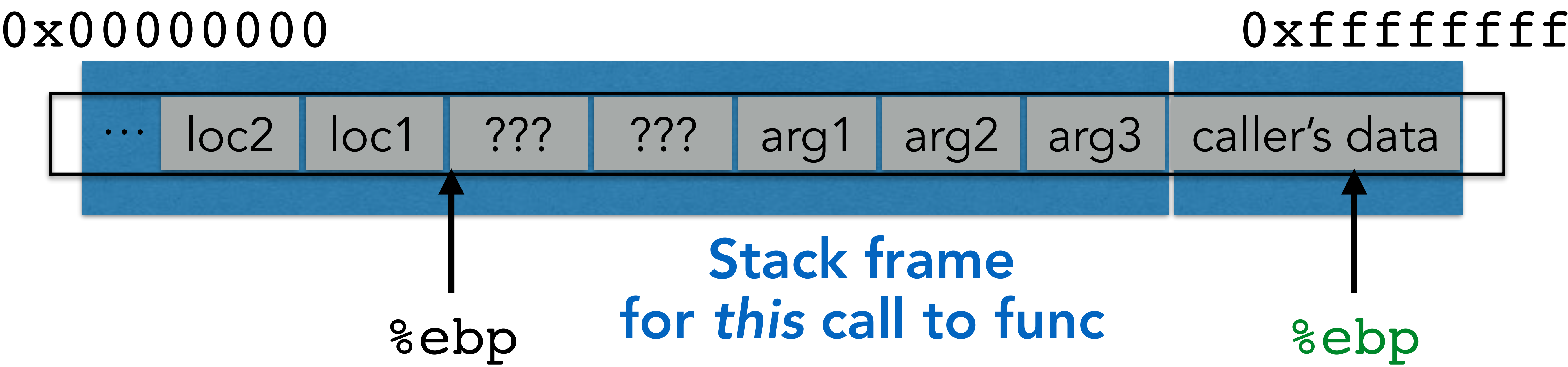
```
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}
```



RETURNING FROM FUNCTIONS

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

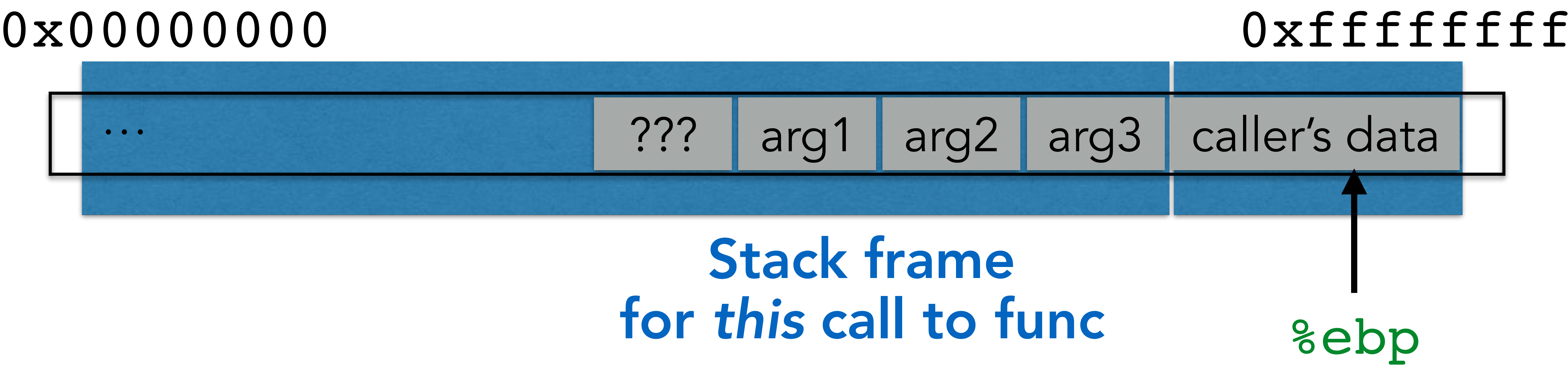
Q: How do we restore %ebp?



RETURNING FROM FUNCTIONS

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

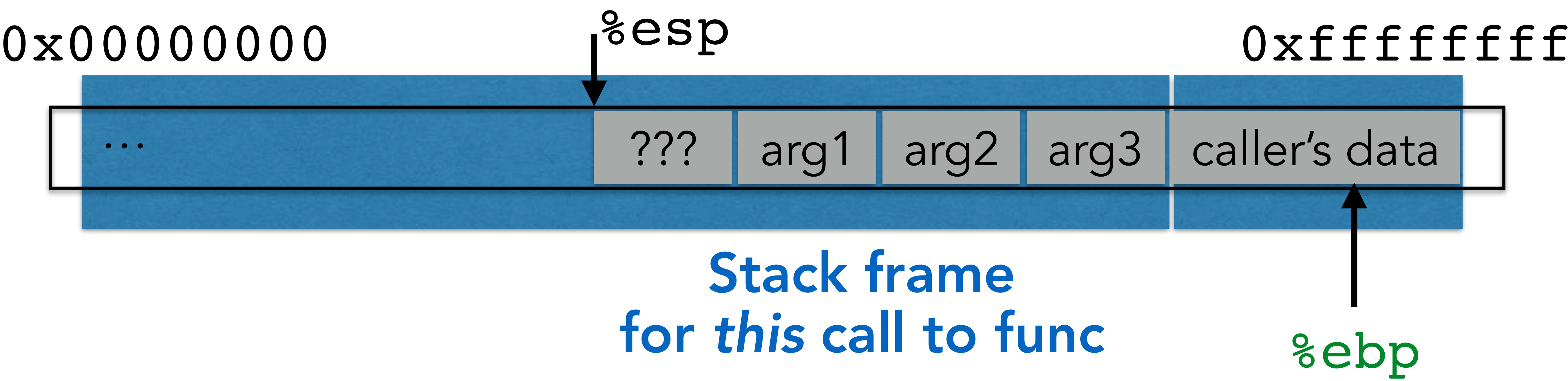
Q: How do we restore %ebp?



RETURNING FROM FUNCTIONS

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

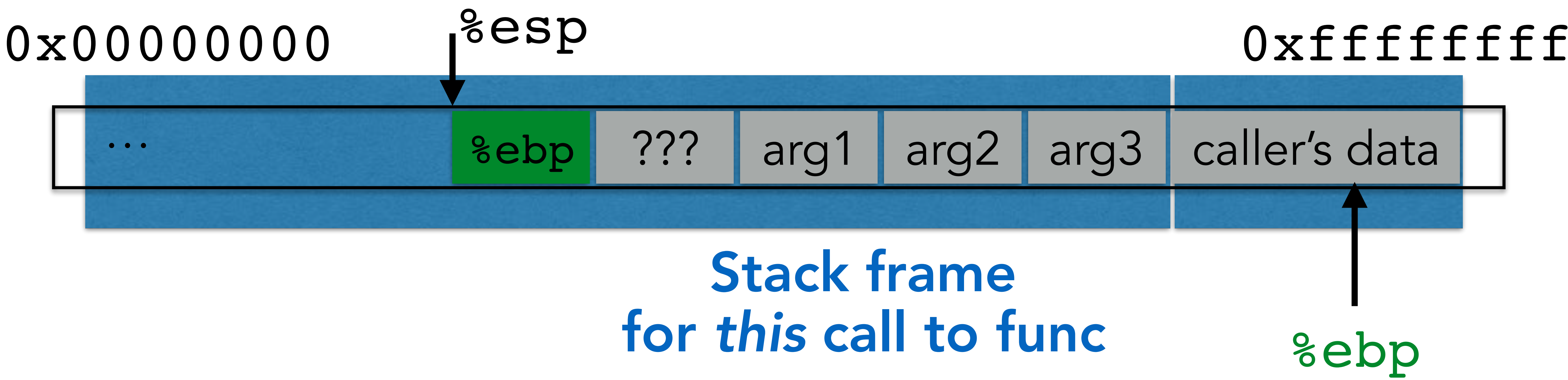
Q: How do we restore %ebp?



RETURNING FROM FUNCTIONS

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

Q: How do we restore %ebp?

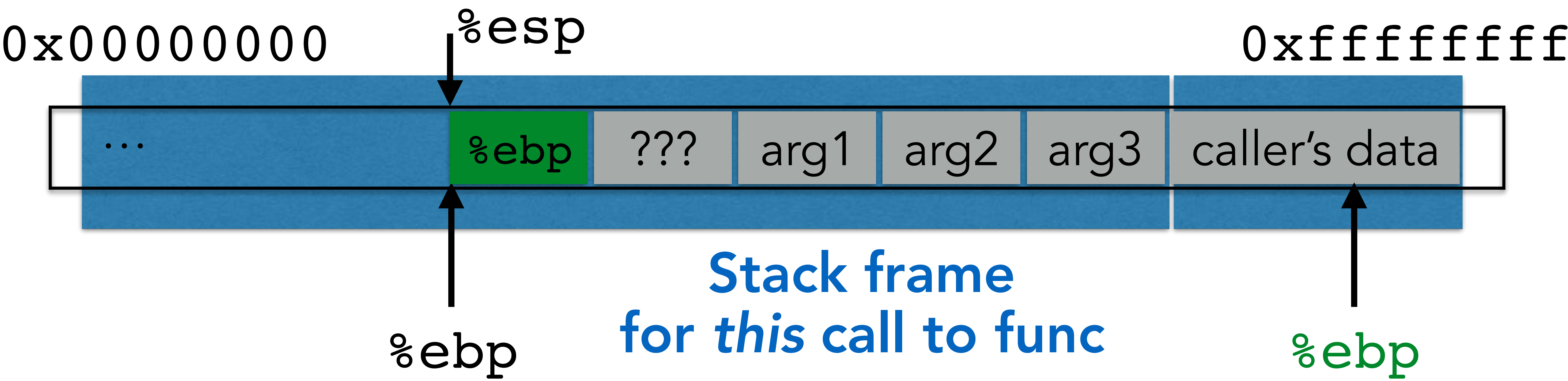


1. Push %ebp before locals

RETURNING FROM FUNCTIONS

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

Q: How do we restore %ebp?

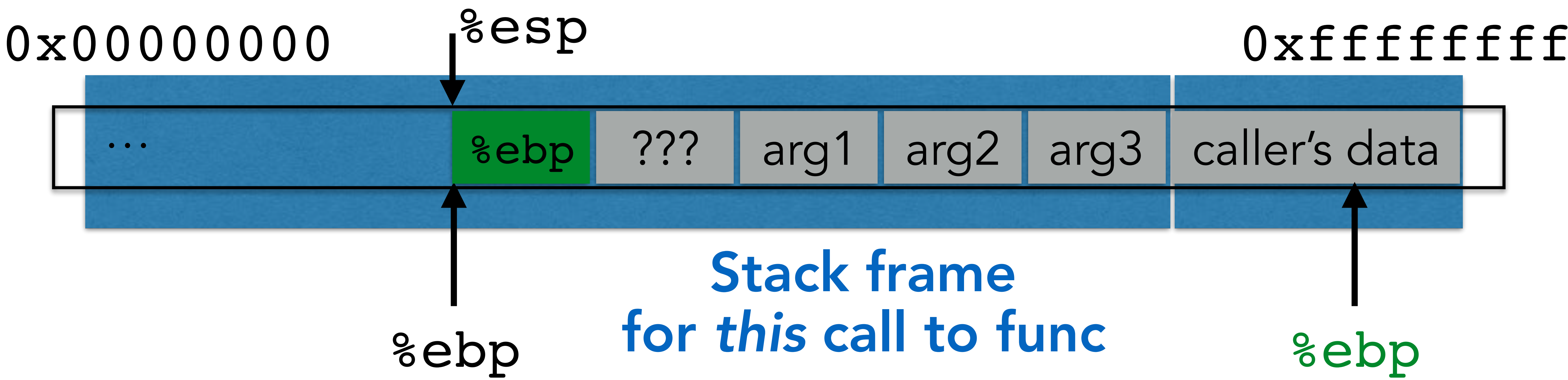


1. Push %ebp before locals
2. Set %ebp to current %esp

RETURNING FROM FUNCTIONS

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

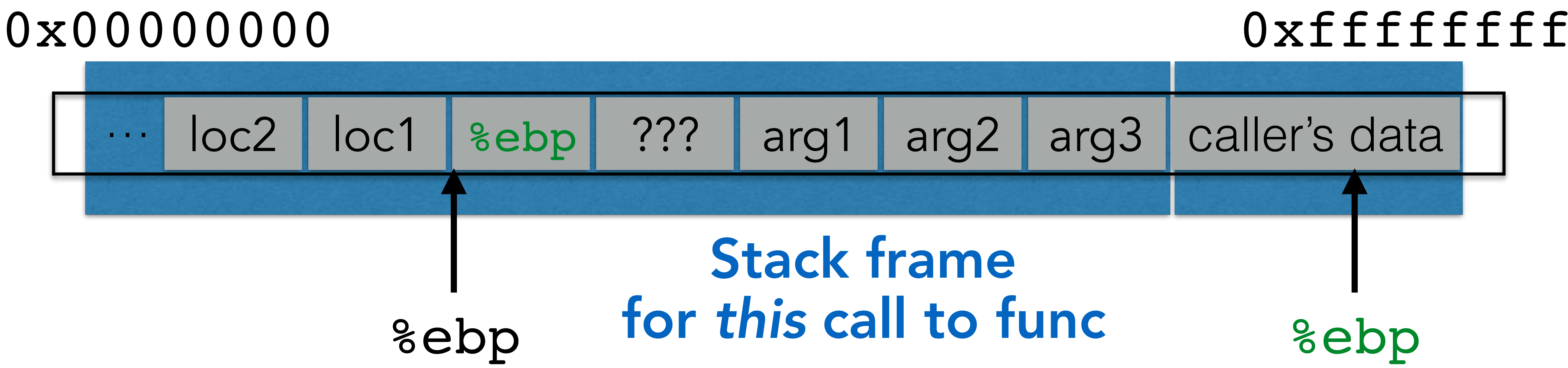
Q: How do we restore %ebp?



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

RETURNING FROM FUNCTIONS

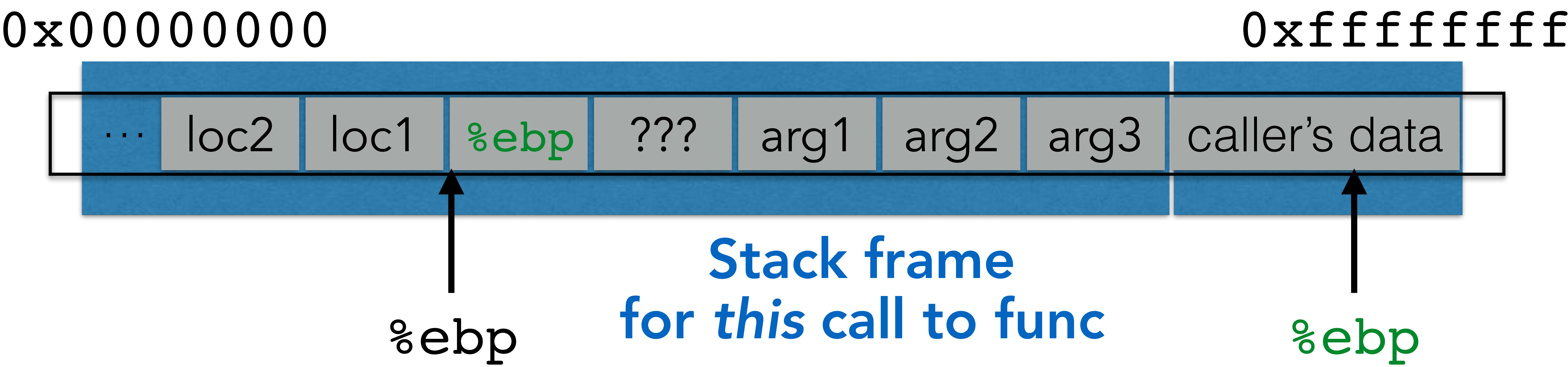
```
int main()
{
     ...
     func("Hey", 10, -3);
     ...
}
```



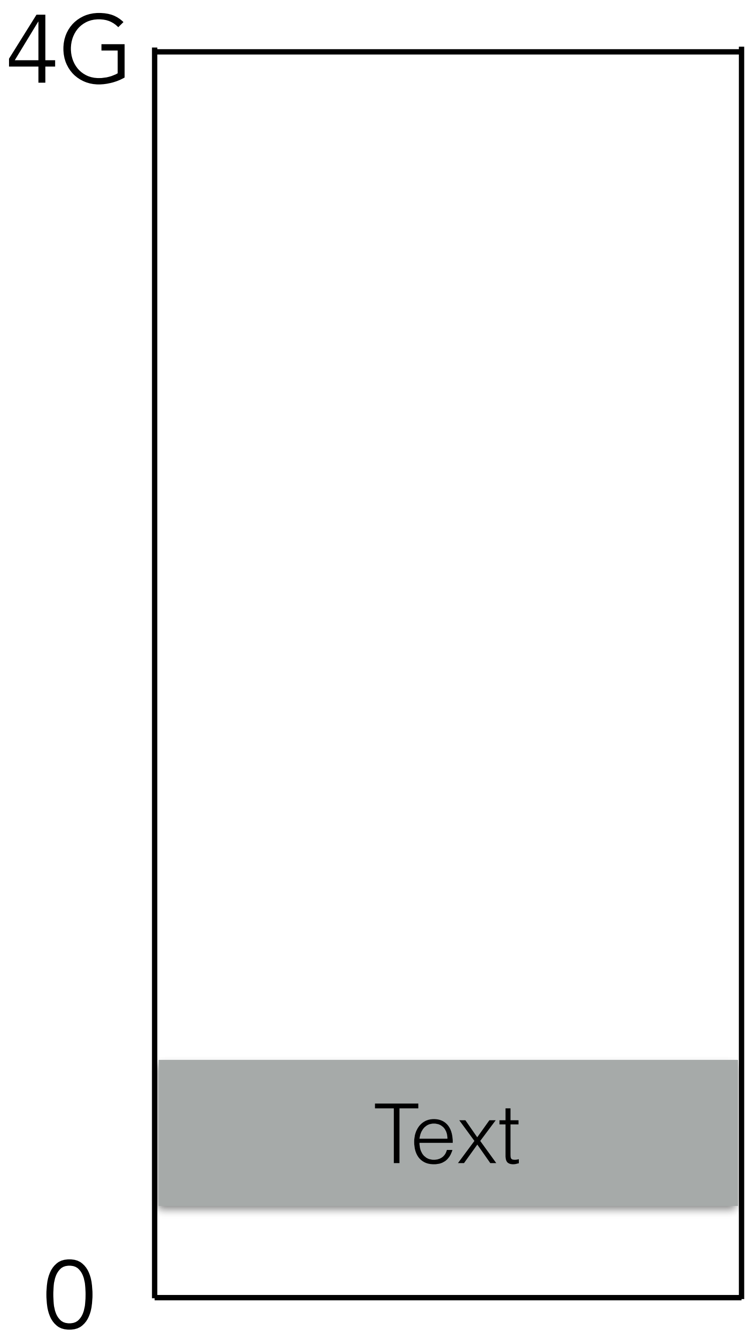
RETURNING FROM FUNCTIONS

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

Q: How do we resume here?



INSTRUCTIONS THEMSELVES ARE IN MEMORY

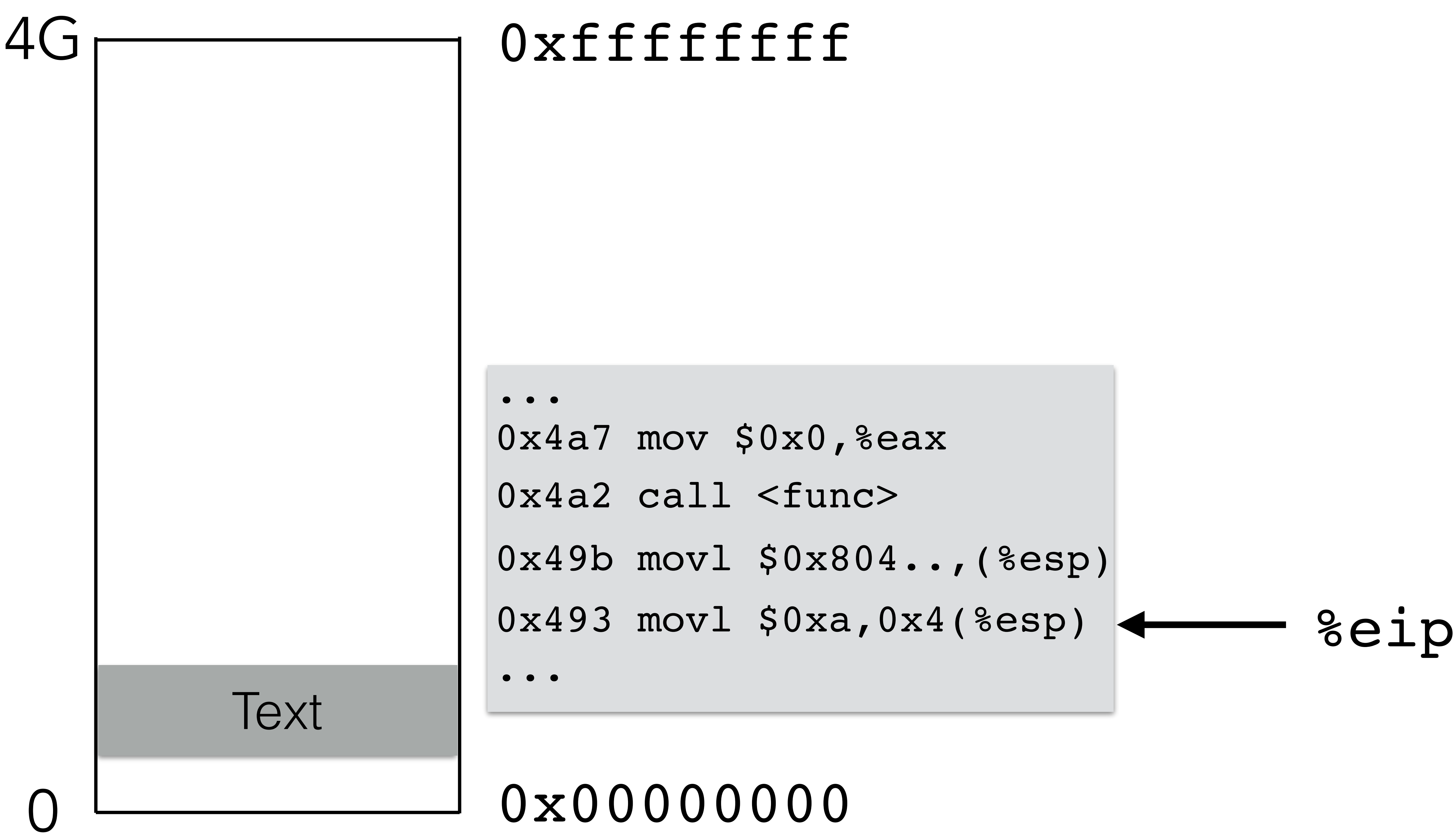


0xffffffff

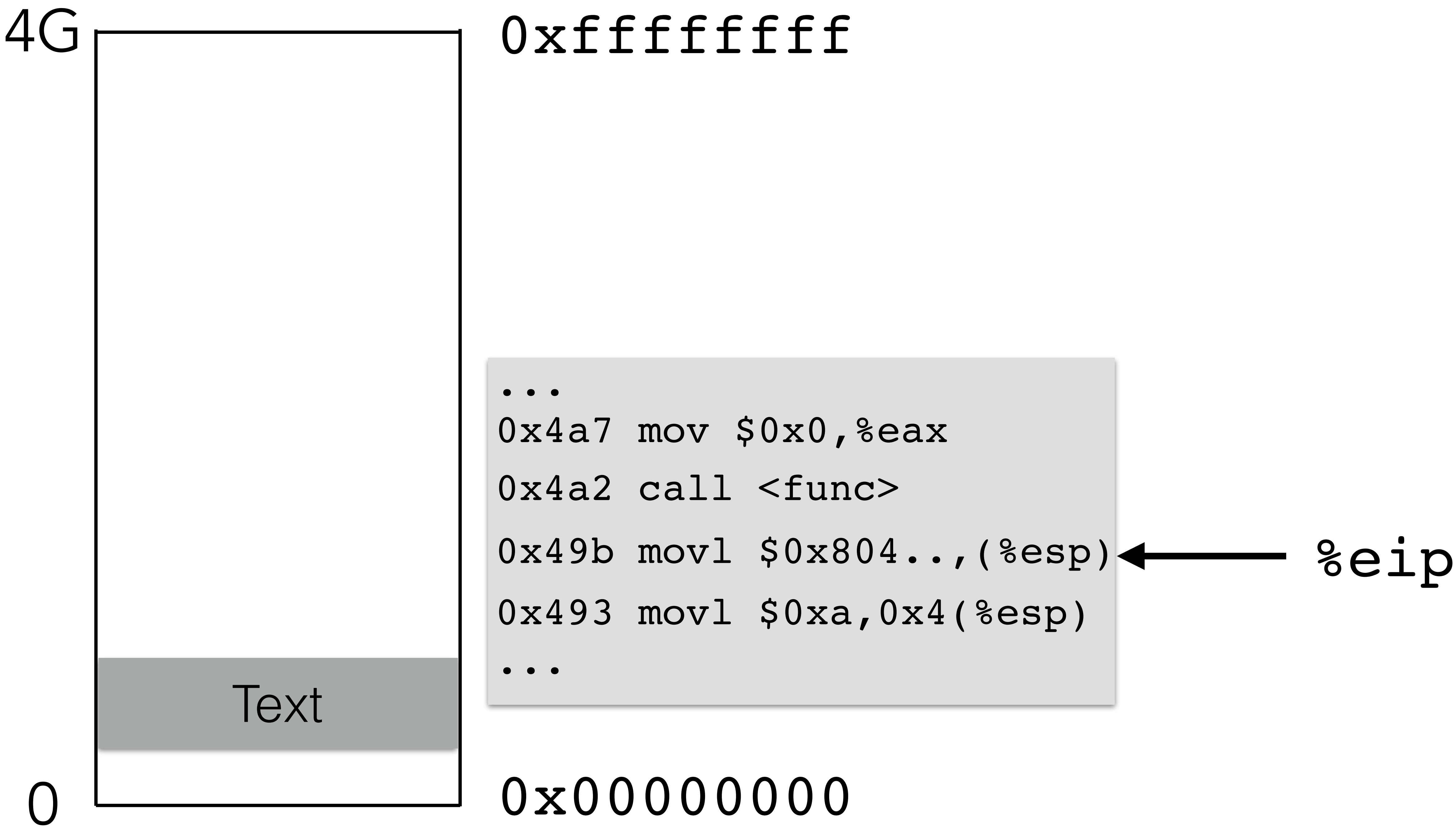
```
...
0x4a7 mov $0x0,%eax
0x4a2 call <func>
0x49b movl $0x804..,(%esp)
0x493 movl $0xa,0x4(%esp)
...
```

0x00000000

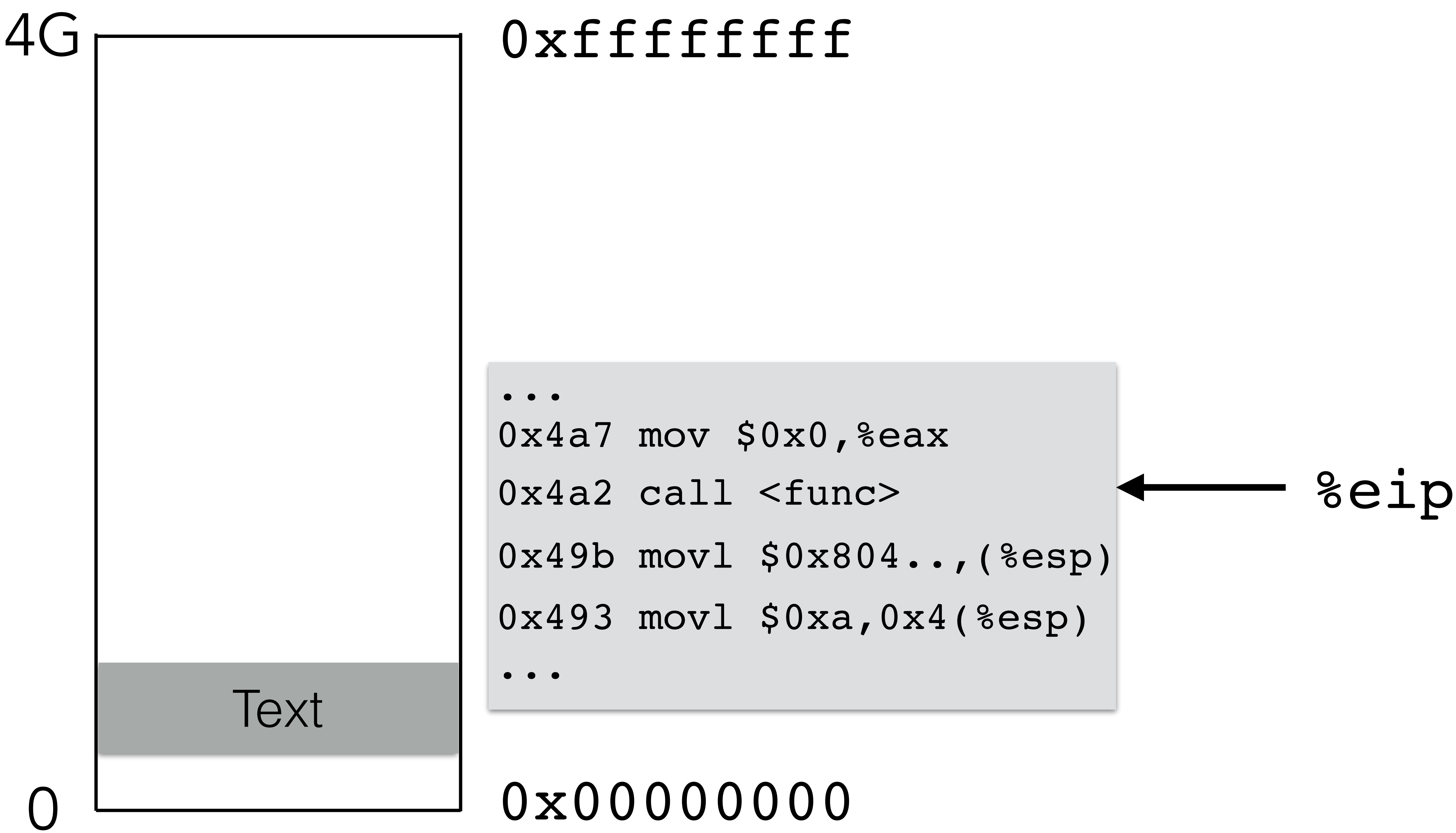
INSTRUCTIONS THEMSELVES ARE IN MEMORY



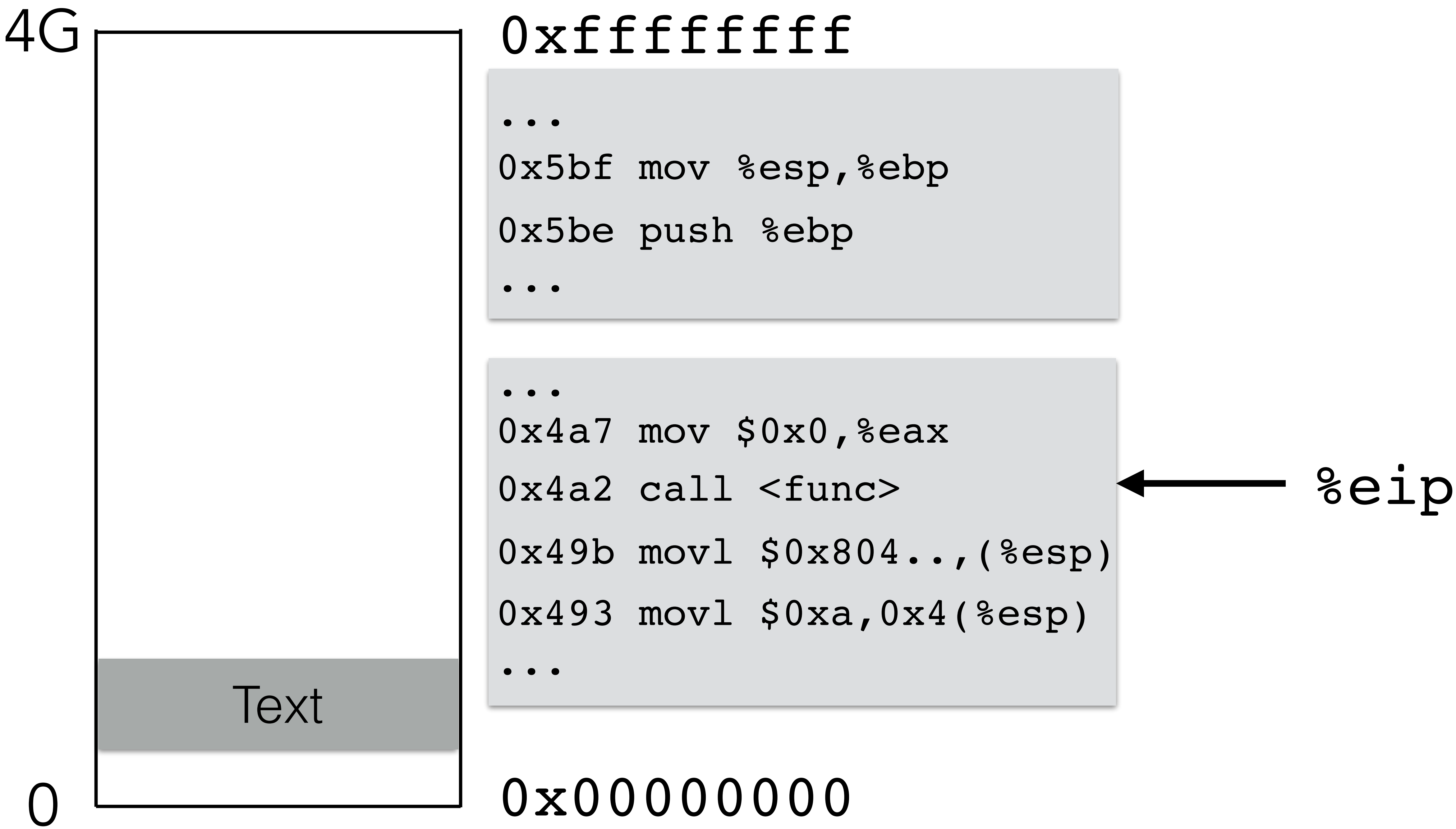
INSTRUCTIONS THEMSELVES ARE IN MEMORY



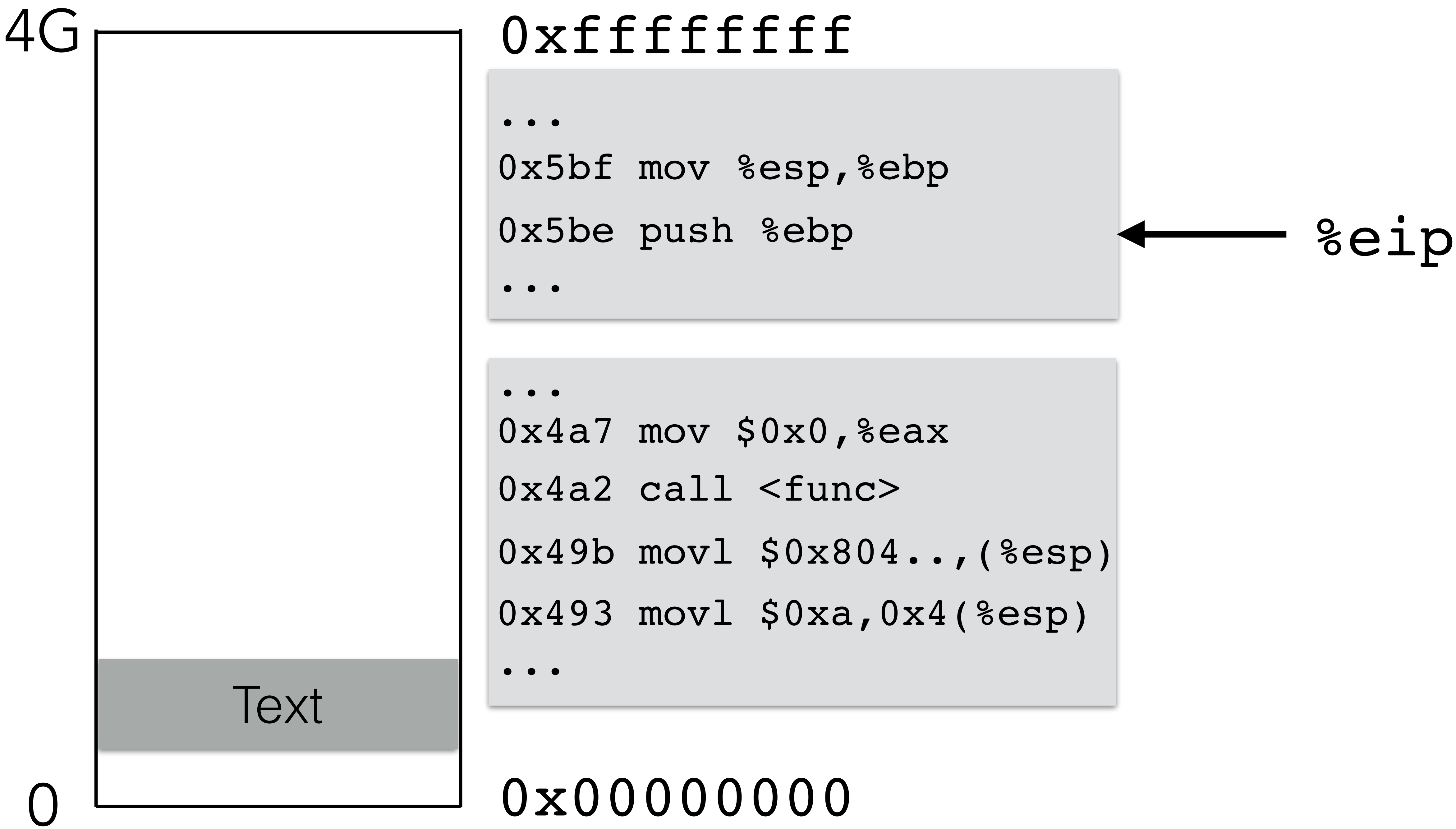
INSTRUCTIONS THEMSELVES ARE IN MEMORY



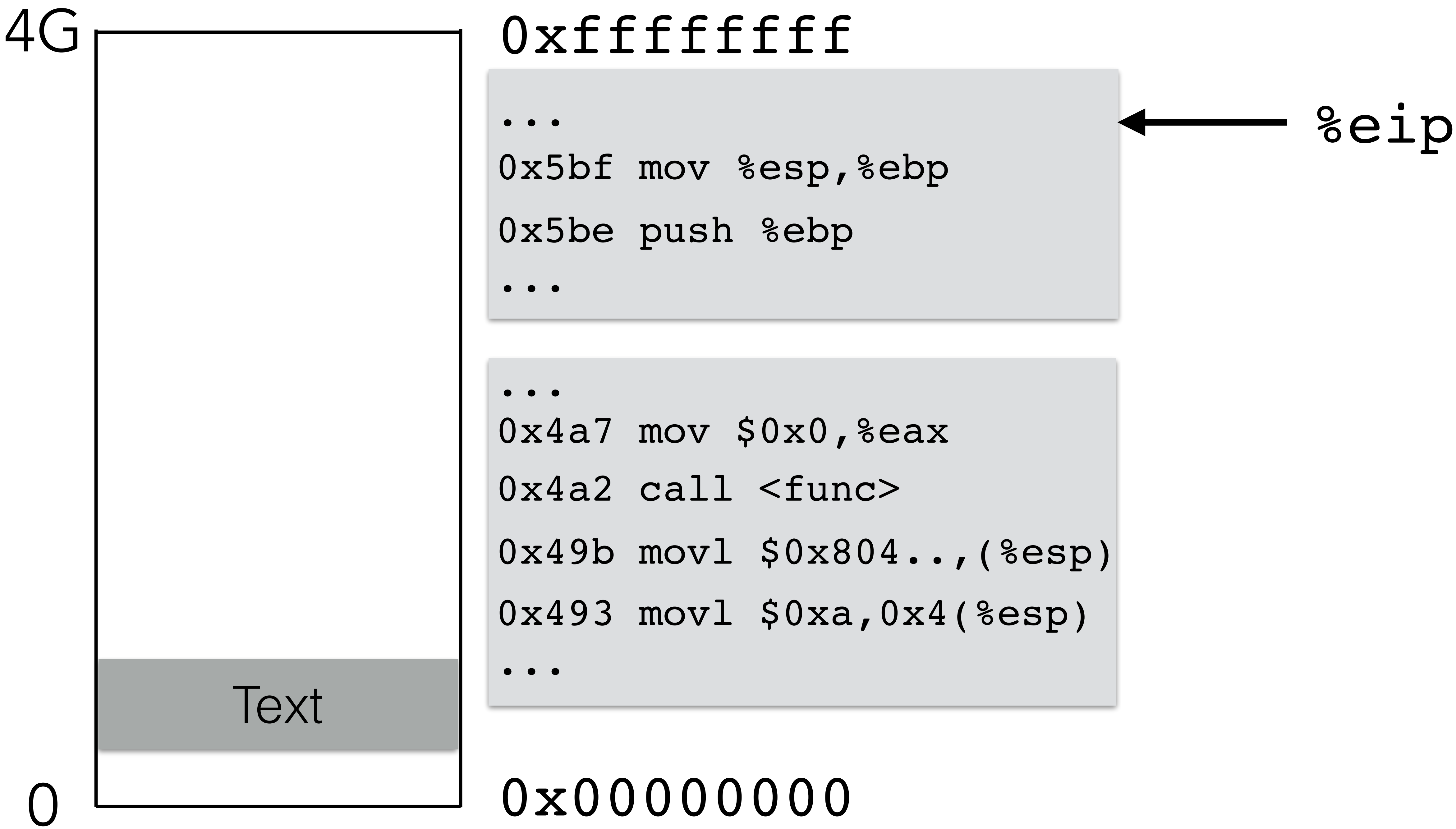
INSTRUCTIONS THEMSELVES ARE IN MEMORY



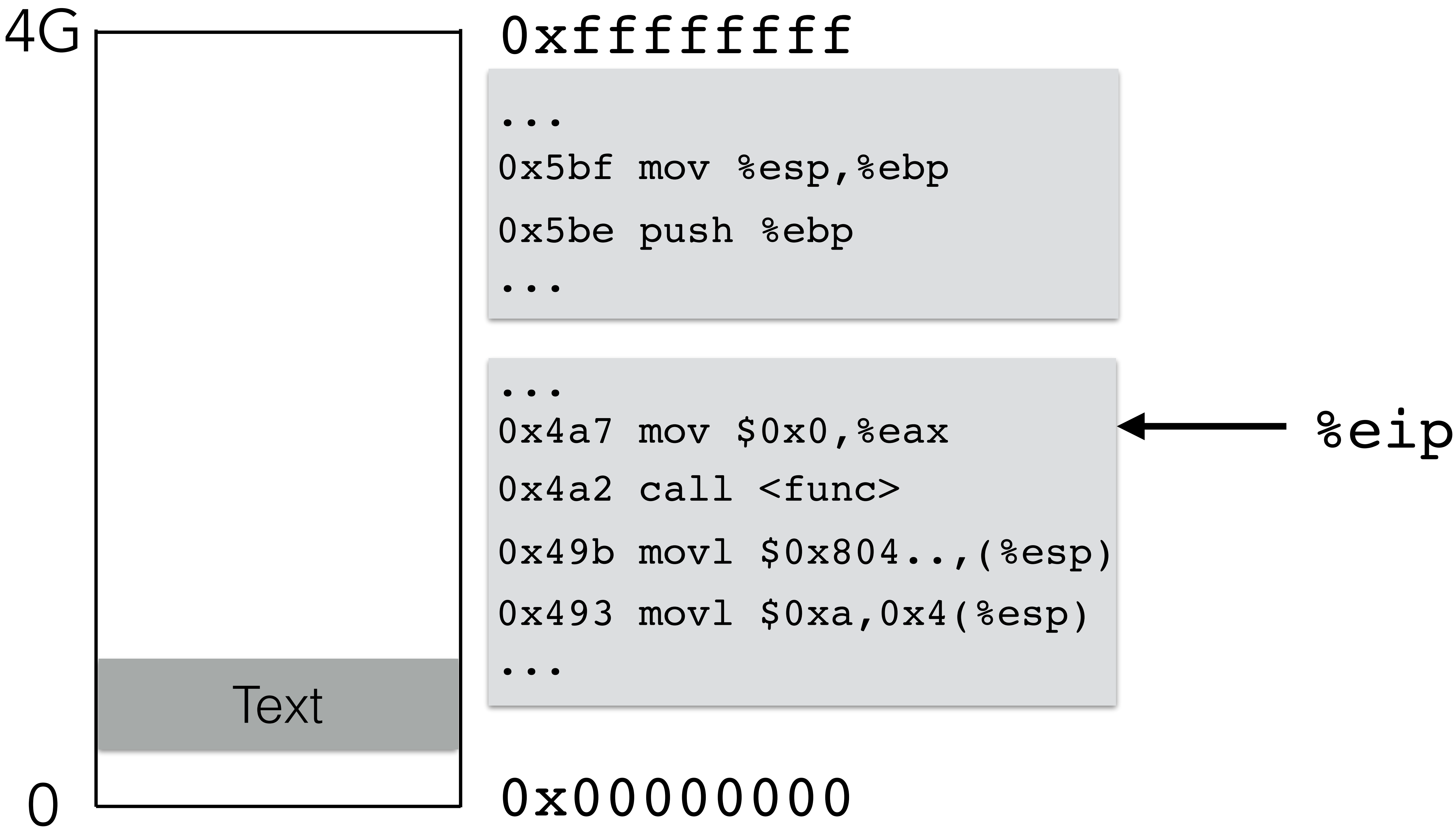
INSTRUCTIONS THEMSELVES ARE IN MEMORY



INSTRUCTIONS THEMSELVES ARE IN MEMORY

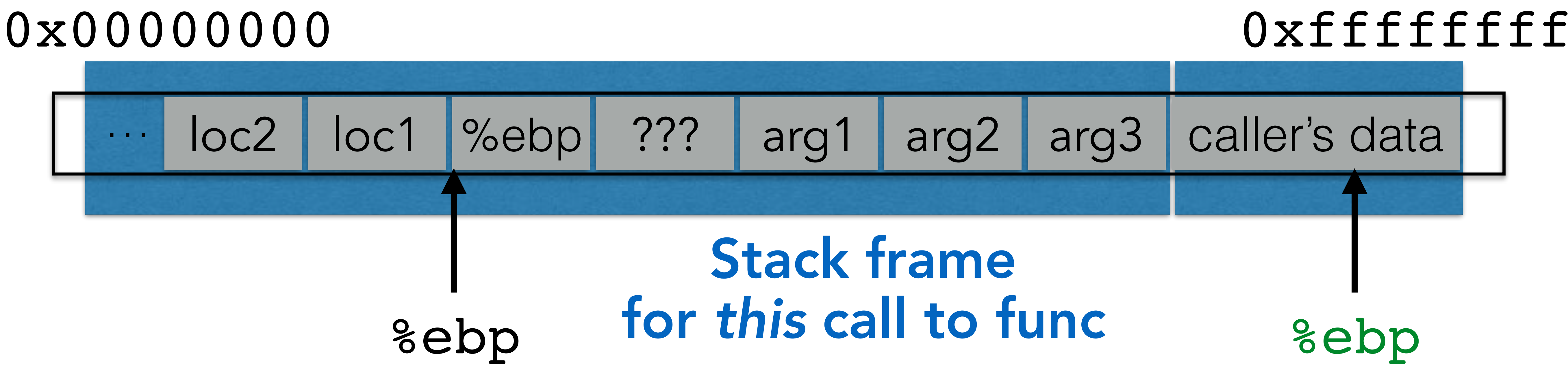


INSTRUCTIONS THEMSELVES ARE IN MEMORY



RETURNING FROM FUNCTIONS

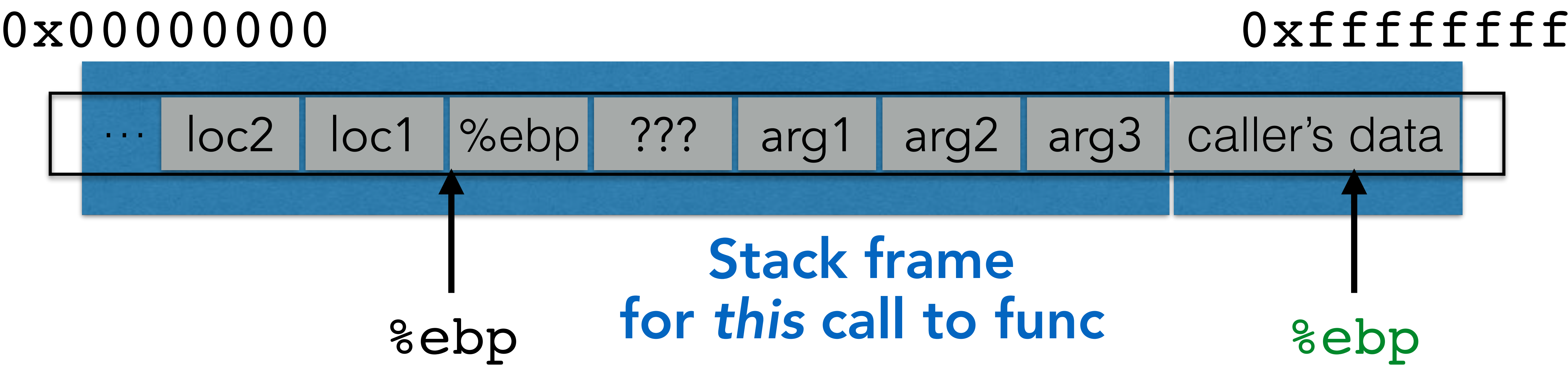
```
int main()
{
     ...
     func("Hey", 10, -3);
     ... Q: How do we resume here?
}
```



RETURNING FROM FUNCTIONS

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

Q: How do we resume here?

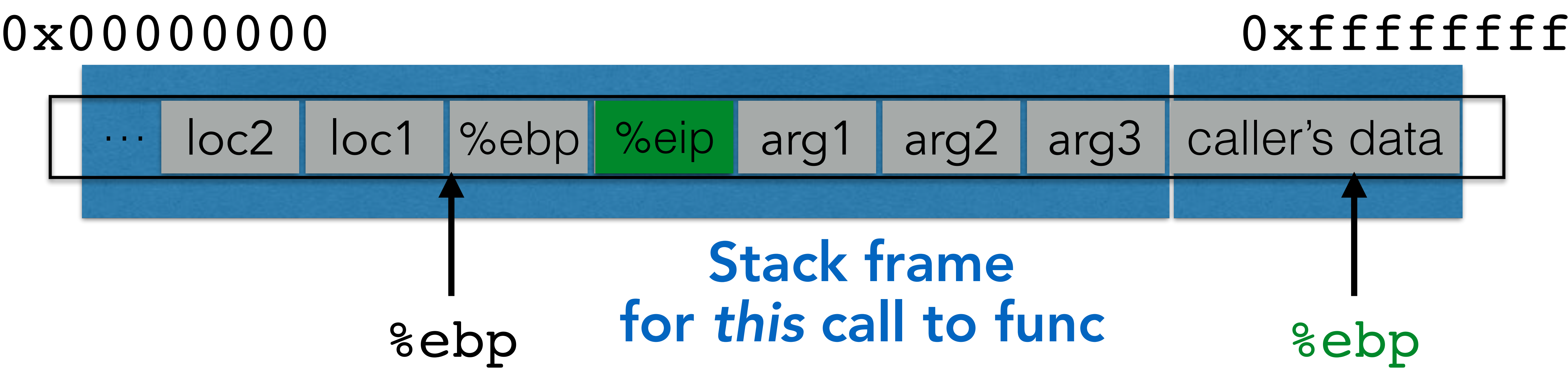


Push next %eip before call

RETURNING FROM FUNCTIONS

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

Q: How do we resume here?

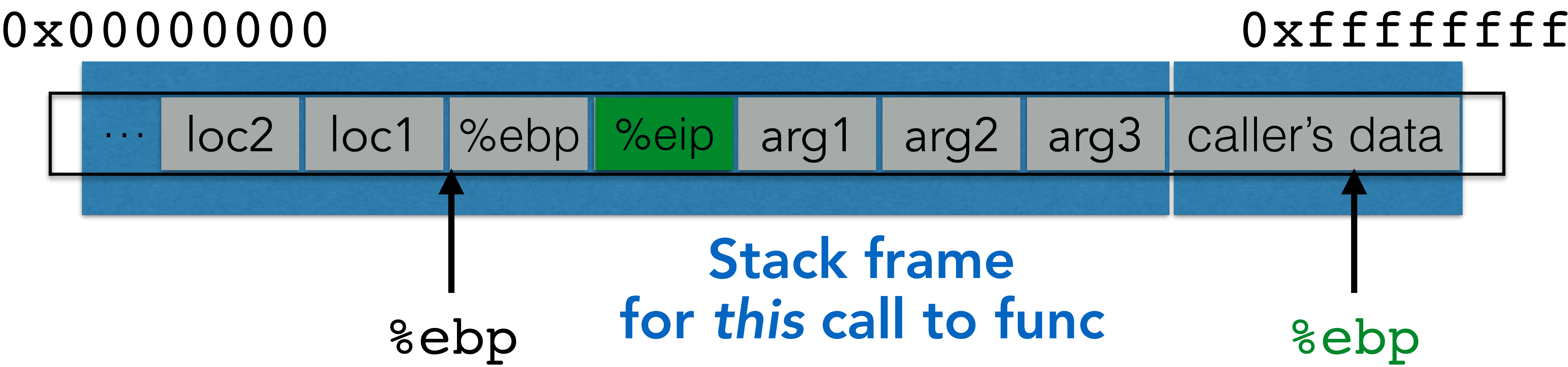


Push next %eip before call

RETURNING FROM FUNCTIONS

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

Q: How do we resume here?



Set %eip to 4(%ebp)
at return

Push next %eip
before call

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**

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

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 OVERFLOW
ATTACKS

BUFFER OVERFLOWS: HIGH LEVEL

- Buffer =
  - Contiguous set of a given data type
  - Common in C
    - All strings are buffers of `char`'s
- Overflow =
  - Put more into the buffer than it can hold
- Where does the extra data go?
- Well now that you're experts in memory layouts…

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

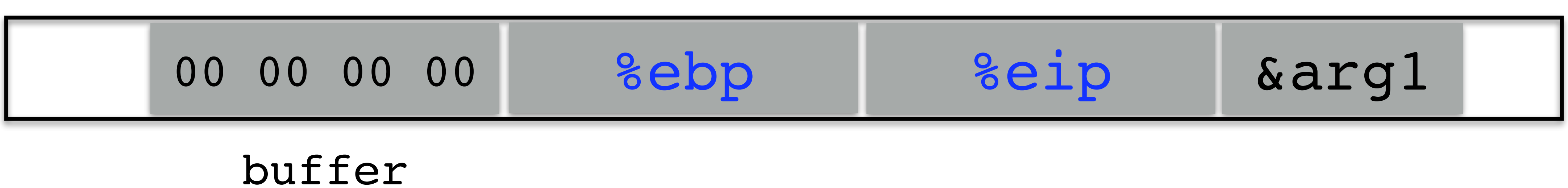
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

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);
    ...
}
```

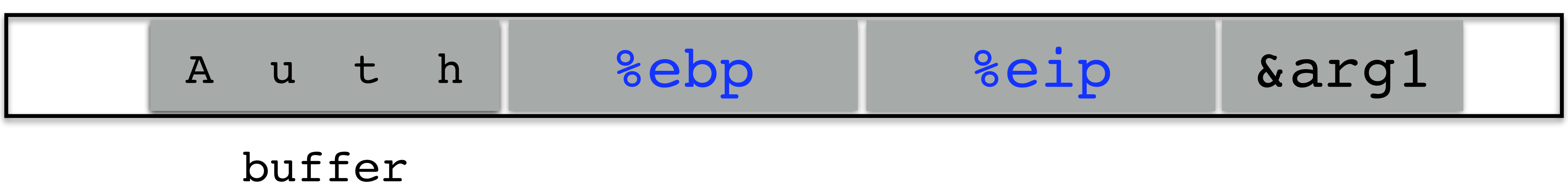
A BUFFER OVERFLOW EXAMPLE

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

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



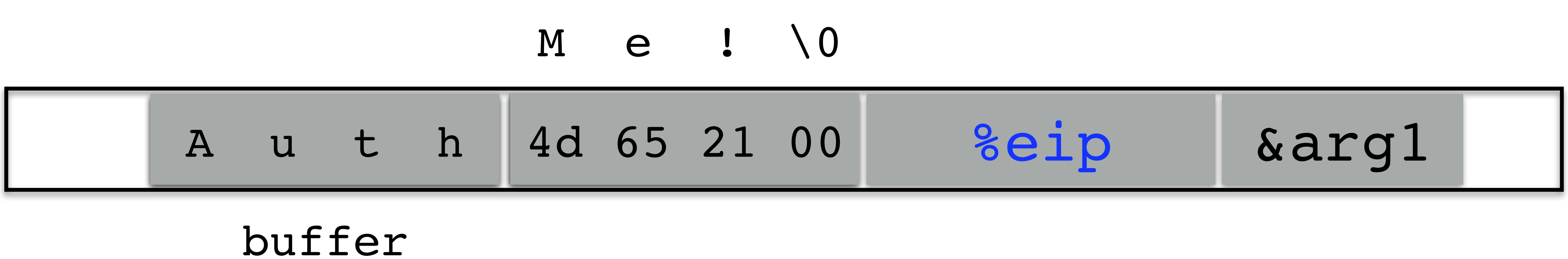
A BUFFER OVERFLOW EXAMPLE

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

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



A BUFFER OVERFLOW EXAMPLE

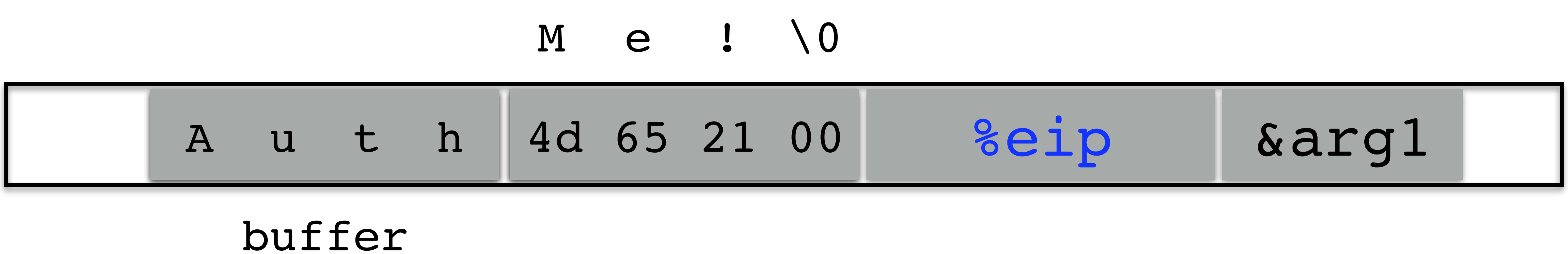
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

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);
    ...
}
```

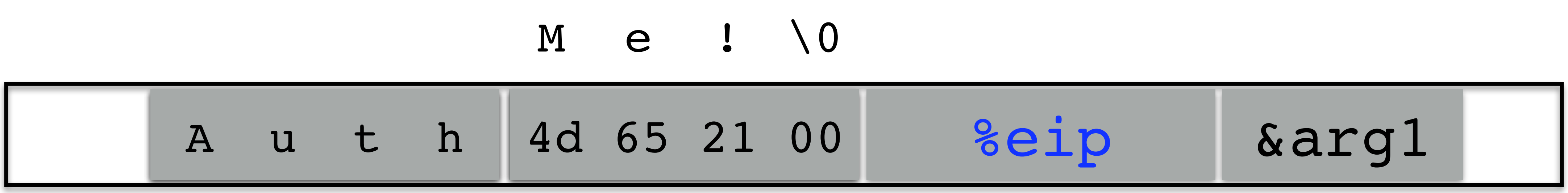
Upon return, sets %ebp to 0x0021654d



A BUFFER OVERFLOW EXAMPLE

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

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

Upon return, sets %ebp to 0x0021654d



buffer

SEGFAULT (0x00216551)

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);
    ...
}
```

&arg1

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);
    ...
}
```

	00 00 00 00	%ebp	%eip	&arg1	

authenticated

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);
    ...
}
```

	00 00 00 00	00 00 00 00	%ebp	%eip	&arg1	
	buffer	authenticated				

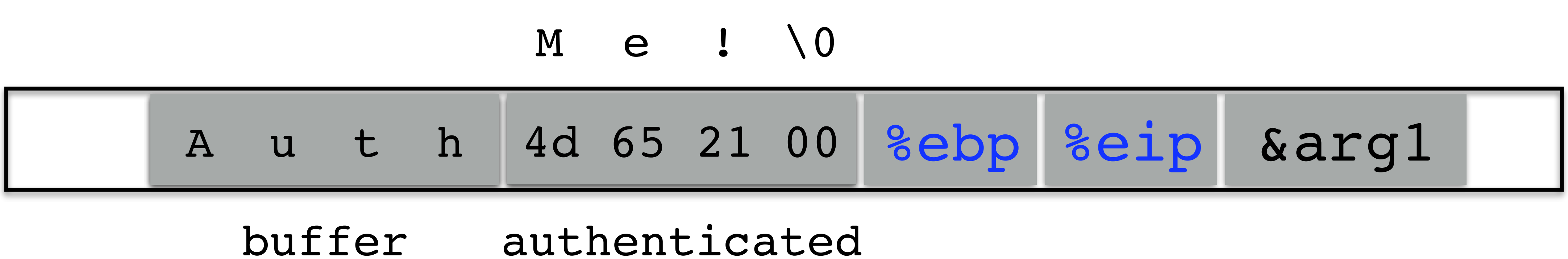
A BUFFER OVERFLOW EXAMPLE

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

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

	A u t h	00 00 00 00	%ebp	%eip	&arg1	
	buffer	authenticated				

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’

		M e ! \0			
	A u t h	4d 65 21 00	%ebp	%eip	&arg1
	buffer	authenticated			

```c
void vulnerable()
{
    char buf[80];
    gets(buf);
}
```

```c
void vulnerable()
{
    char buf[80];
    gets(buf);
}
```

```c
void still_vulnerable()
{
    char *buf = malloc(80);
    gets(buf);
}
```

```
void safe()
{
    char buf[80];
    fgets(buf, 64, stdin);
}
```

```c
void safe()
{
    char buf[80];
    fgets(buf, 64, stdin);
}
```

```c
void safer()
{
    char buf[80];
    fgets(buf, sizeof(buf), stdin);
}
```

IE's Role in the Google-China War

By Richard Adhikari
TechNewsWorld
01/15/10 12:25 PM PT

A A Text Size
Print Version
E-Mail Article

The hack attack on Google that set off the company's ongoing standoff with China appears to have come through a zero-day flaw in Microsoft's Internet Explorer browser. Microsoft has released a security advisory, and researchers are hard at work studying the exploit. The attack appears to consist of several files, each a different piece of malware.

Computer security companies are scurrying to cope with the fallout from the Internet Explorer (IE) flaw that led to cyberattacks on Google and its corporate and individual customers.

The zero-day attack that exploited IE is part of a lethal cocktail of malware that is keeping researchers very busy.

"We're discovering things on an up-to-the-minute basis, and we've seen about a dozen files dropped on infected PCs so far," Dmitri Alperovitch, vice president of research at McAfee Labs, told TechNewsWorld.

The attacks on Google, which appeared to originate in China, have sparked a feud between the Internet giant and the nation's government over censorship, and it could result in Google pulling away from its business dealings in the country.

Pointing to the Flaw

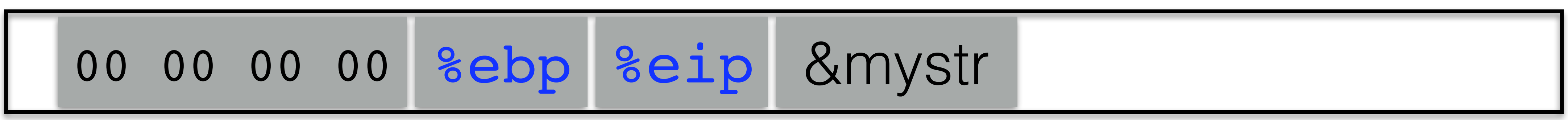
The vulnerability in IE is an invalid pointer reference, Microsoft said in security advisory 979352, which it issued on Thursday. Under certain conditions, the invalid pointer can be accessed after an object is deleted, the advisory states. In specially crafted attacks, like the ones launched against Google and its customers, IE can allow remote execution of code when the flaw is exploited.

USER-SUPPLIED STRINGS

- In these examples, we were providing our own strings
- But they come from users in myriad aways
    - Text input
    - Network packets
    - Environment variables
    - File input…

WHAT'S THE WORST THAT CAN HAPPEN?

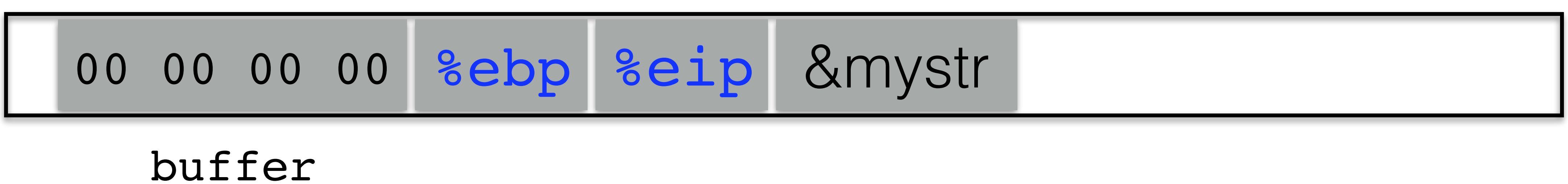
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```



buffer

WHAT'S THE WORST THAT CAN HAPPEN?

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

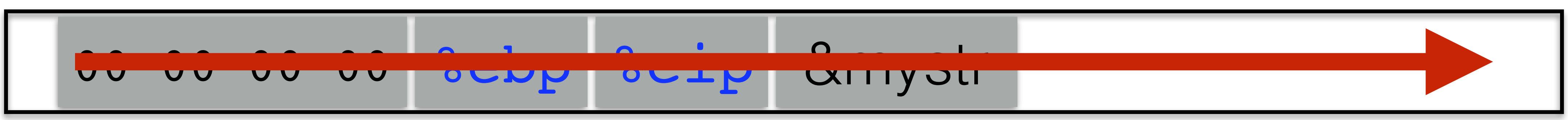


`strcpy` will let you write as much as you want (til a '\0')

WHAT'S THE WORST THAT CAN HAPPEN?

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

All ours!

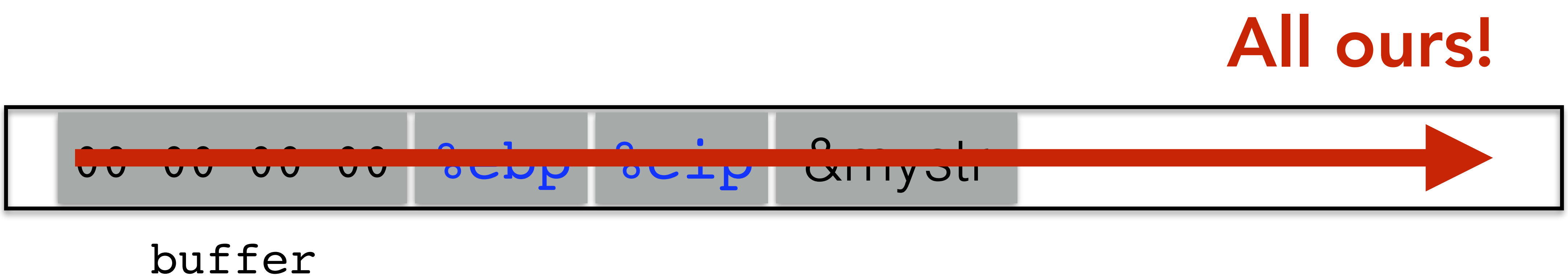


buffer

`strcpy` will let you write as much as you want (til a '\0')

WHAT'S THE WORST THAT CAN HAPPEN?

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



`strcpy` will let you write as much as you want (til a '\0')

What could you write to memory to wreak havoc?

FIRST A RECAP: ARGS

```
#include <stdio.h>

void func(char *arg1, int arg2, int arg3)
{
    printf("arg1 is at %p\n", &arg1);
    printf("arg2 is at %p\n", &arg2);
    printf("arg3 is at %p\n", &arg3);
}

int main()
{
    func("Hello", 10, -3);
    return 0;
}
```

FIRST A RECAP: ARGS

```
#include <stdio.h>

void func(char *arg1, int arg2, int arg3)
{
    printf("arg1 is at %p\n", &arg1);
    printf("arg2 is at %p\n", &arg2);
    printf("arg3 is at %p\n", &arg3);
}

int main()
{
    func("Hello", 10, -3);
    return 0;
}
```

What will happen?

`&arg1 < &arg2 < &arg3?`   `&arg1 > &arg2 > &arg3?`

FIRST A RECAP: LOCALS

```
#include <stdio.h>

void func()
{
    char loc1[4];
    int  loc2;
    int  loc3;
    printf(“loc1 is at %p\n”, &loc1);
    printf(“loc2 is at %p\n”, &loc2);
    printf(“loc3 is at %p\n”, &loc3);
}

int main()
{
    func();
    return 0;
}
```

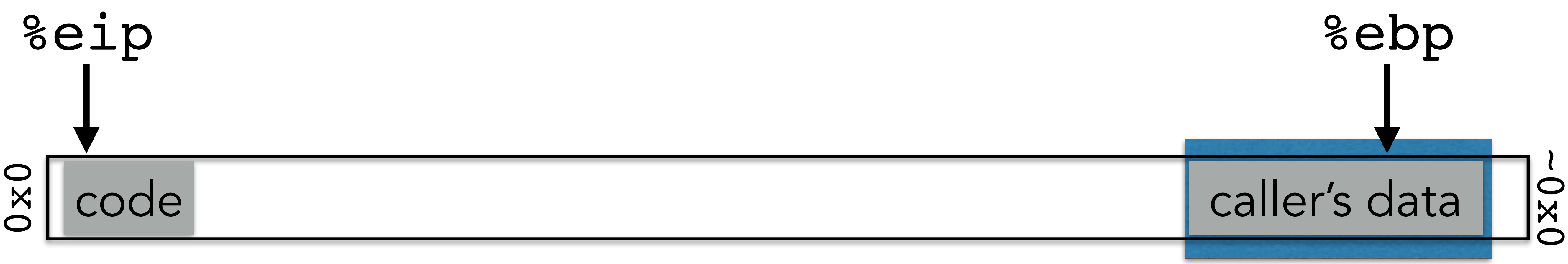
FIRST A RECAP: LOCALS

```
#include <stdio.h>

void func()
{
    char loc1[4];
    int  loc2;
    int  loc3;
    printf("loc1 is at %p\n", &loc1);
    printf("loc2 is at %p\n", &loc2);
    printf("loc3 is at %p\n", &loc3);
}

int main()
{
    func();
    return 0;
}
```

What will happen?

`&loc1 < &loc2 < &loc3?`   `&loc1 > &loc2 > &loc3?`

STACK & FUNCTIONS: SUMMARY



STACK & FUNCTIONS: SUMMARY

Calling function:

1. **Push arguments** of the function you're calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

STACK & FUNCTIONS: SUMMARY

Calling function:

1. **Push arguments** of the function you're calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

STACK & FUNCTIONS: SUMMARY

Calling function:

1. **Push arguments** of the function you're calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

STACK & FUNCTIONS: SUMMARY

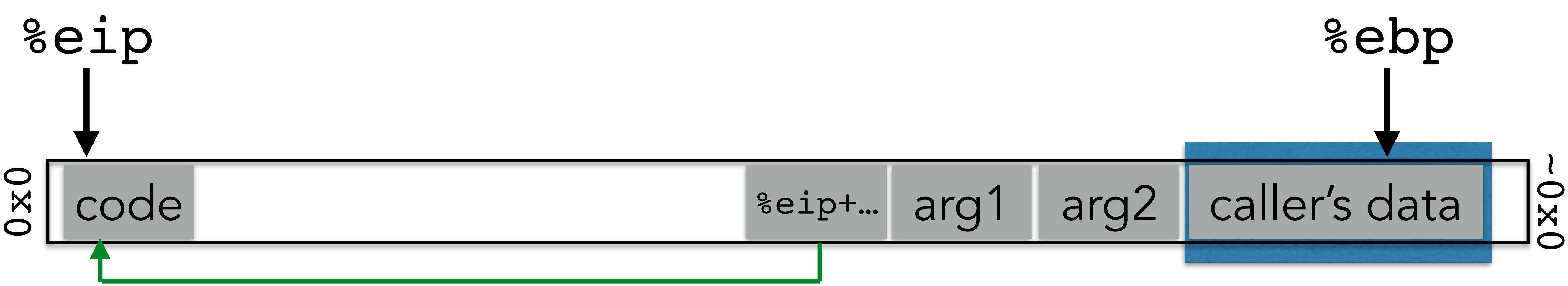
Calling function:

1. **Push arguments** of the function you're calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

STACK & FUNCTIONS: SUMMARY

Calling function:

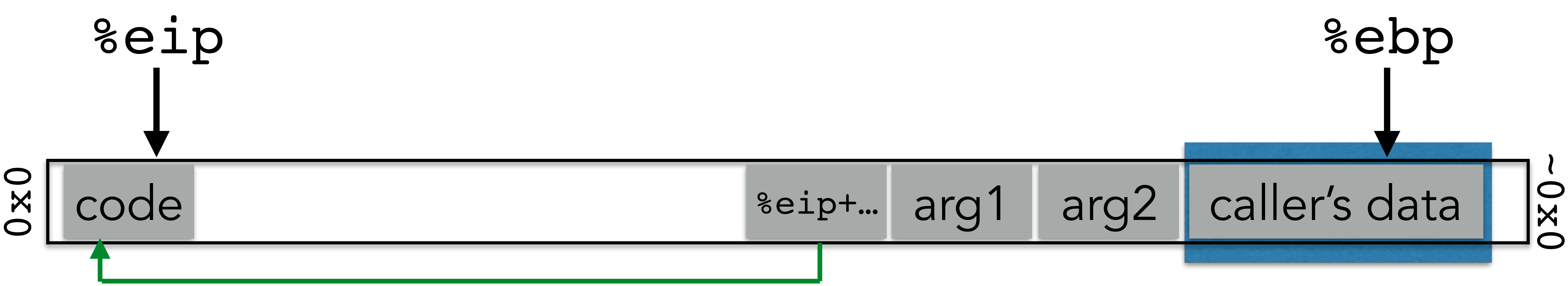
1. **Push arguments** of the function you're calling onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**



STACK & FUNCTIONS: SUMMARY

Calling function:

1. **Push arguments** of the function you're calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**



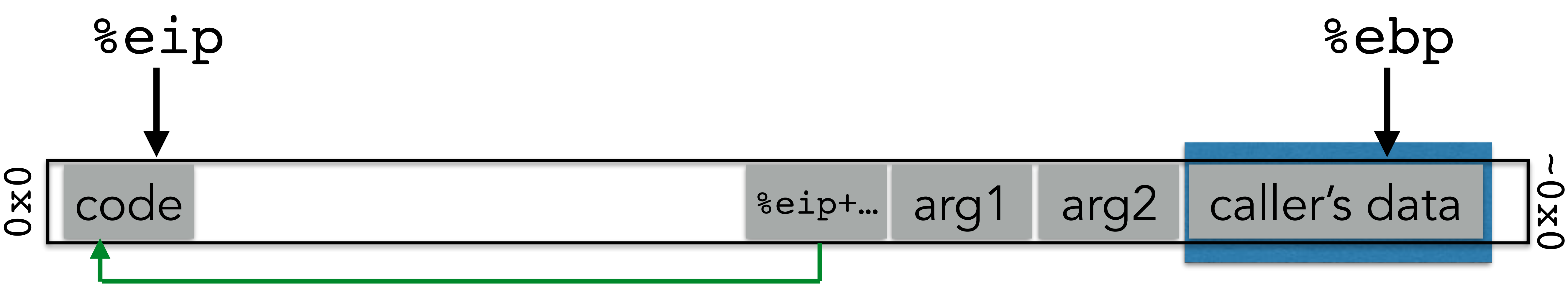
STACK & FUNCTIONS: SUMMARY

Calling function:

1. **Push arguments** of the function you're calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

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



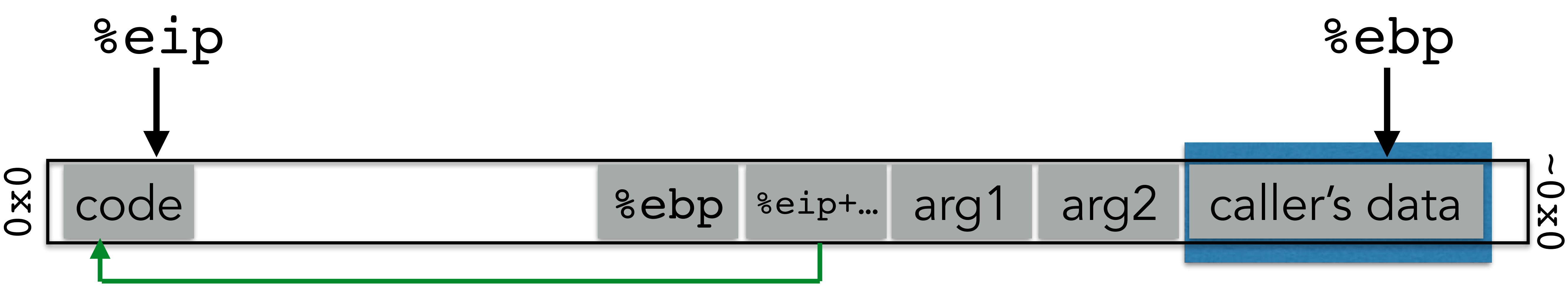
STACK & FUNCTIONS: SUMMARY

Calling function:

1. **Push arguments** of the function you're calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

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



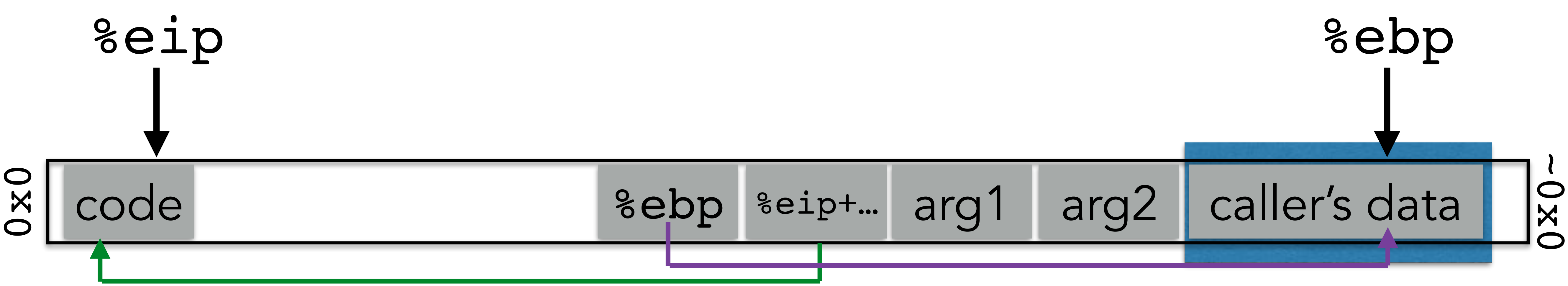
STACK & FUNCTIONS: SUMMARY

Calling function:

1. **Push arguments** of the function you're calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

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



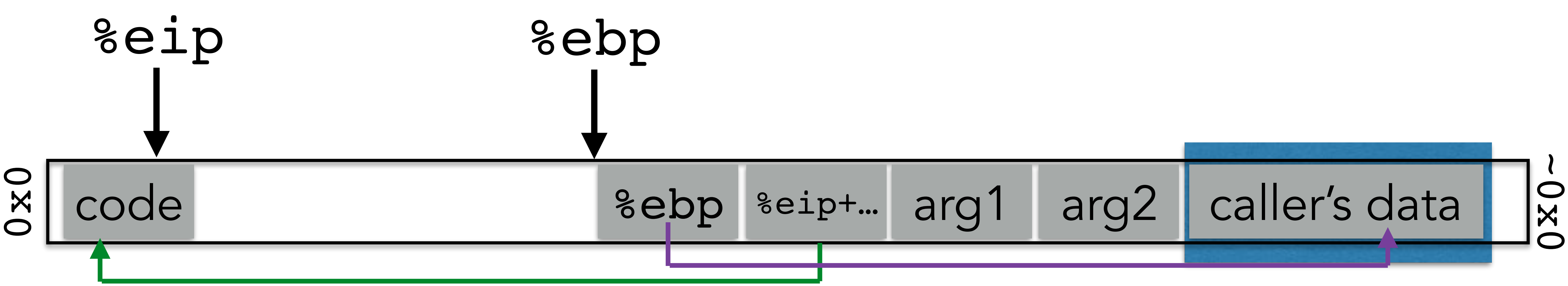
STACK & FUNCTIONS: SUMMARY

Calling function:

1. **Push arguments** of the function you're calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

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



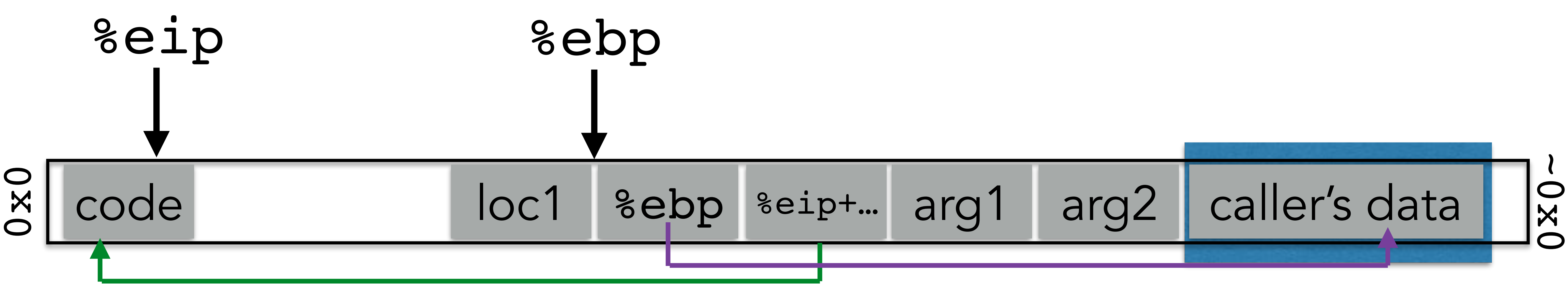
STACK & FUNCTIONS: SUMMARY

Calling function:

1. **Push arguments** of the function you're calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

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



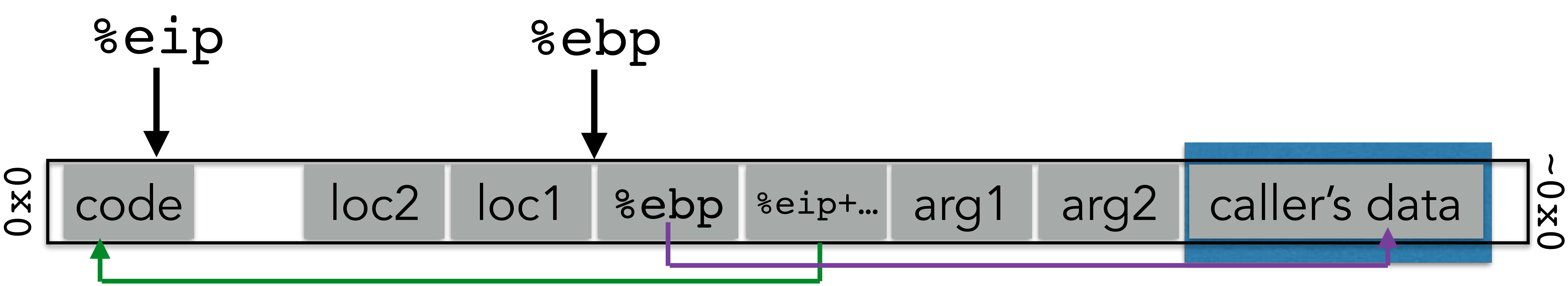
STACK & FUNCTIONS: SUMMARY

Calling function:

1. **Push arguments** of the function you're calling 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: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

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



STACK & FUNCTIONS: SUMMARY

Calling function:

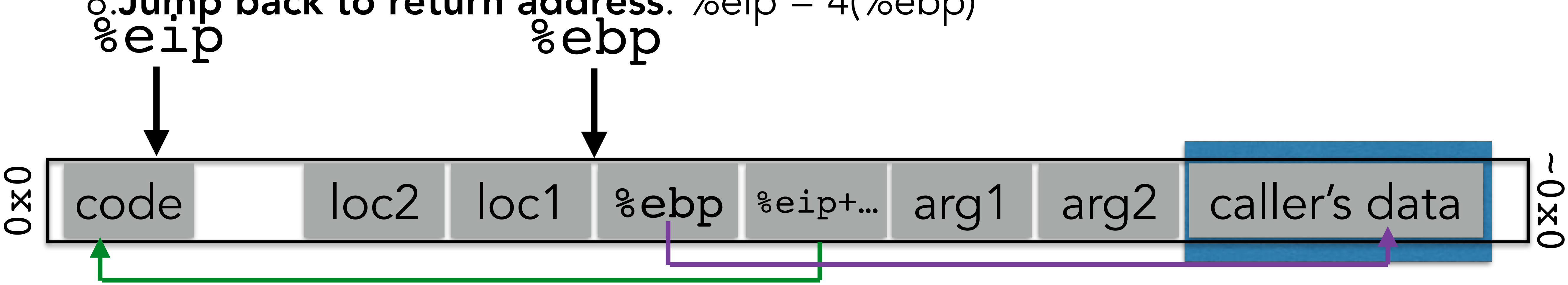
1. **Push arguments** of the function you're calling onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

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)
8. **Jump back to return address**: %eip = 4(%ebp)



STACK & FUNCTIONS: SUMMARY

Calling function:

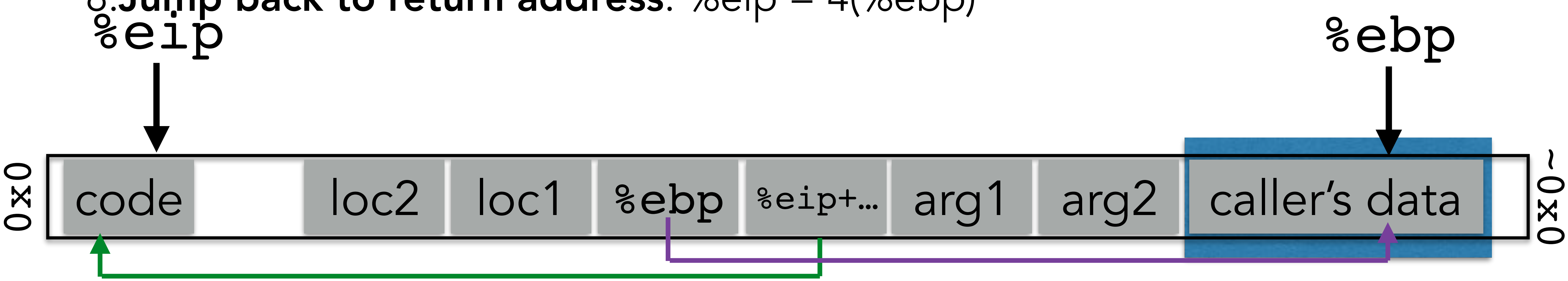
1. **Push arguments** of the function you're calling onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

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)
8. **Jump back to return address**: %eip = 4(%ebp)



STACK & FUNCTIONS: SUMMARY

Calling function:

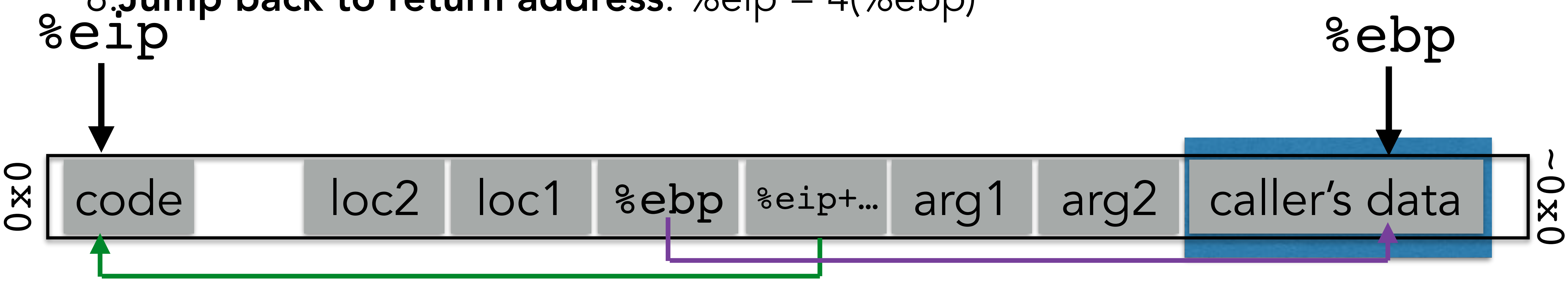
1. **Push arguments** of the function you're calling onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

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)
8. **Jump back to return address**: %eip = 4(%ebp)

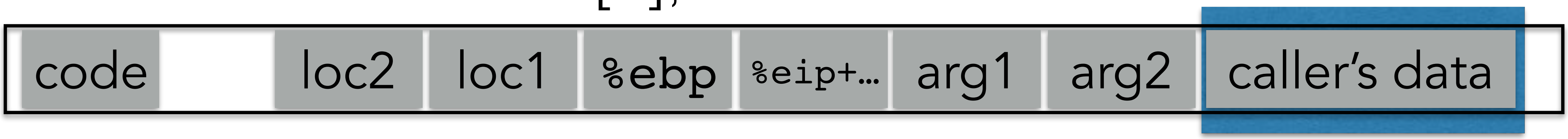


GDB: YOUR NEW BEST FRIEND

`i f`	Show **i**nfo about the current **f**rame (prev. frame, locals/args, %ebp/%eip)
`i r`	Show **info** about **reg**isters (%ebp, %eip, %esp, etc.)
`x/<n> <addr>`	E**x**amine <n> bytes of memory starting at address <addr>
`b <function>` `s`	Set a **b**reakpoint at <function> **s**tep through execution (into calls)

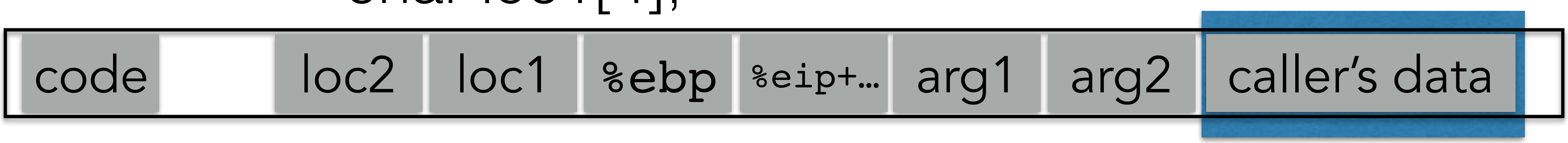
BUFFER OVERFLOW

char loc1[4];



BUFFER OVERFLOW

char loc1[4];




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

etc.

BUFFER OVERFLOW

char loc1[4];


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

etc.

BUFFER OVERFLOW

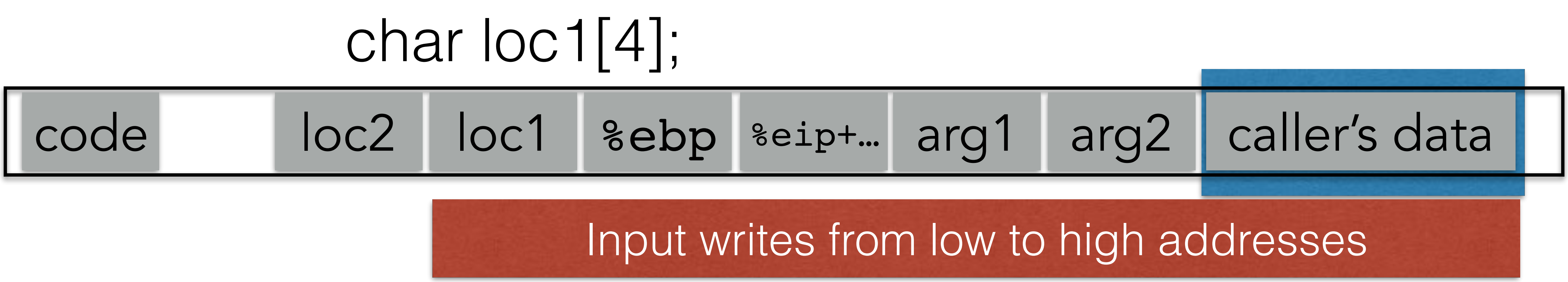
char loc1[4];


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

etc.

BUFFER OVERFLOW

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




```
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);
    ...
}
```

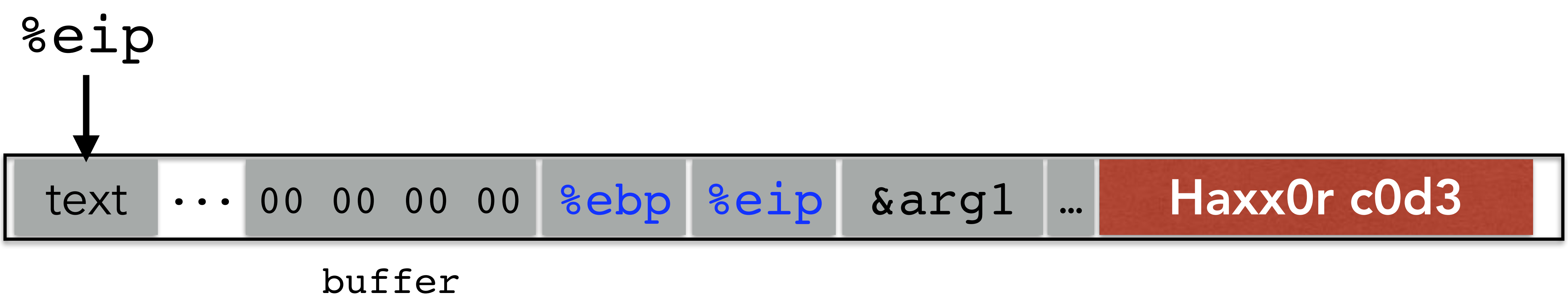
HIGH-LEVEL IDEA

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

(1) Load our own code into memory

HIGH-LEVEL IDEA

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

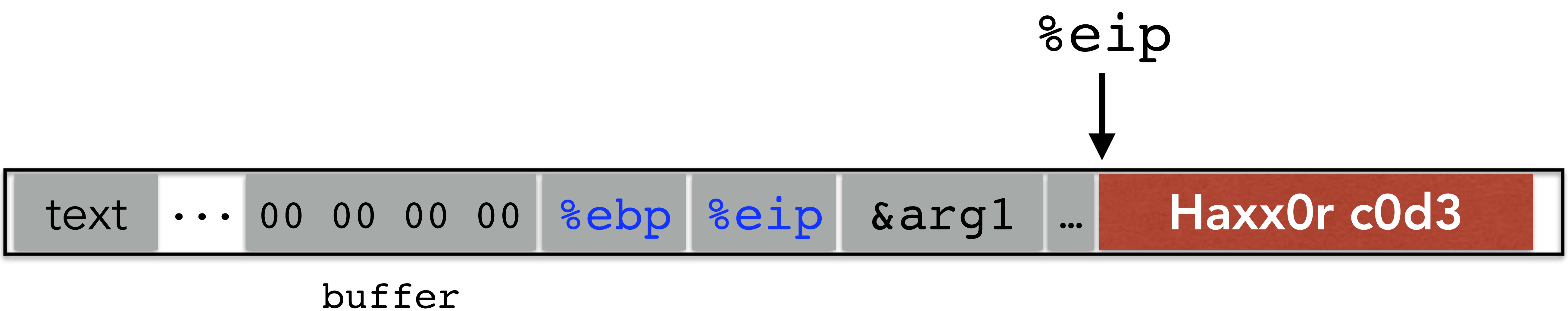


(1) Load our own code into memory

(2) Somehow get %eip to point to it

HIGH-LEVEL IDEA

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

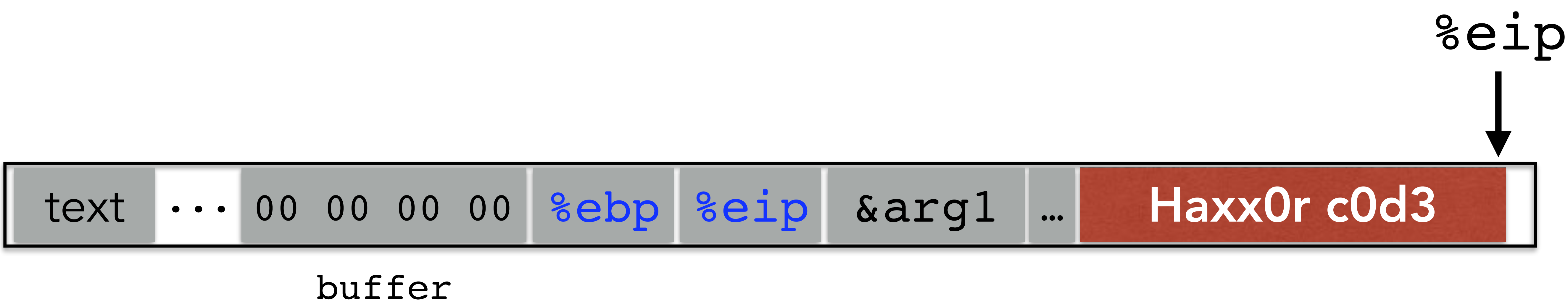


(1) Load our own code into memory

(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

THIS IS NONTRIVIAL

- Pulling off this attack requires getting a few things really right (and some things sorta right)

- Think about what is tricky about the attack
    - The key to defending it will be to make the hard parts *really* hard

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?

- 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);
}
```

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
...
```

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
...
```

SHELLCODE

```c
#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

```
#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
...
```

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

Machine code

(Part of) your input

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

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
- Consider passwd: root owns it, but users can run it
  - getuid() will return who ran it (real userid)
  - seteuid(0) to set the **e**ffective 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

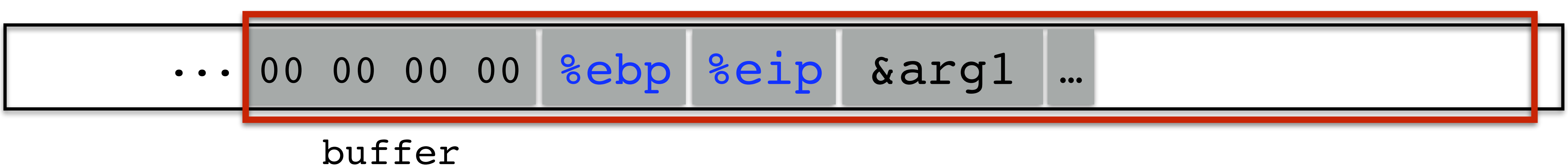
CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

- ***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

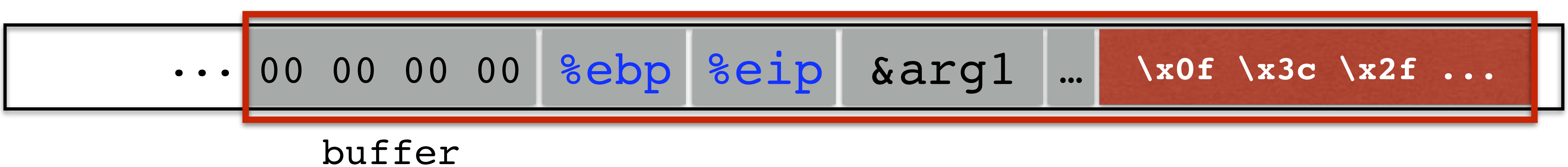
- ***All we can do is write to memory from buffer onward***
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running



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



Thoughts?

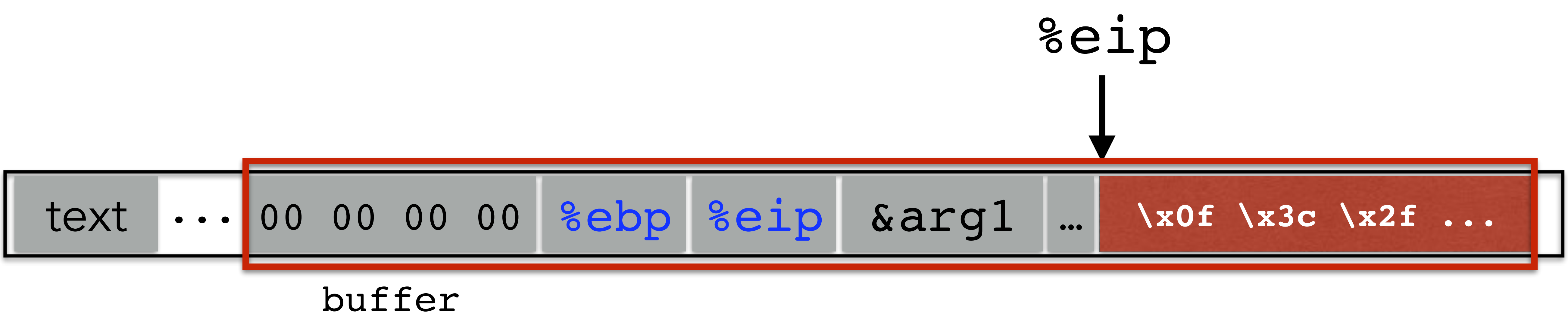
CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

- ***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

Thoughts?

CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

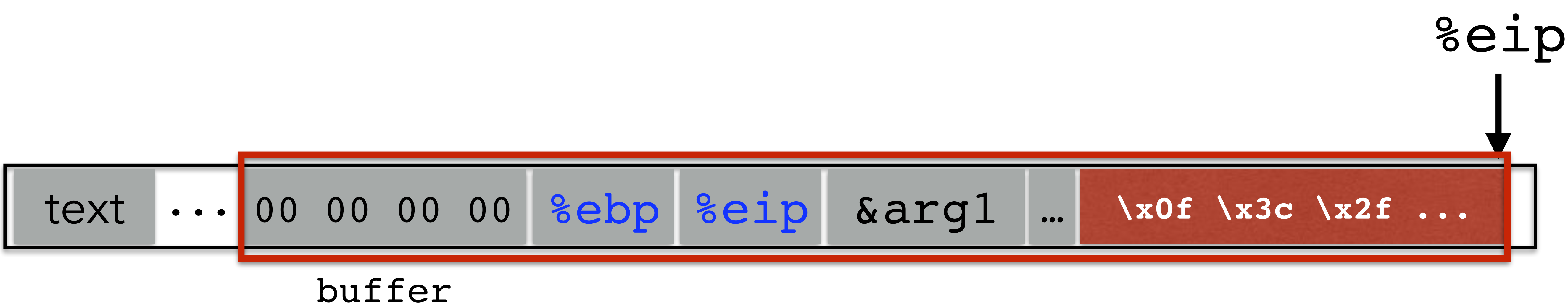
- ***All we can do is write to memory from buffer onward***
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running



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



Thoughts?

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**

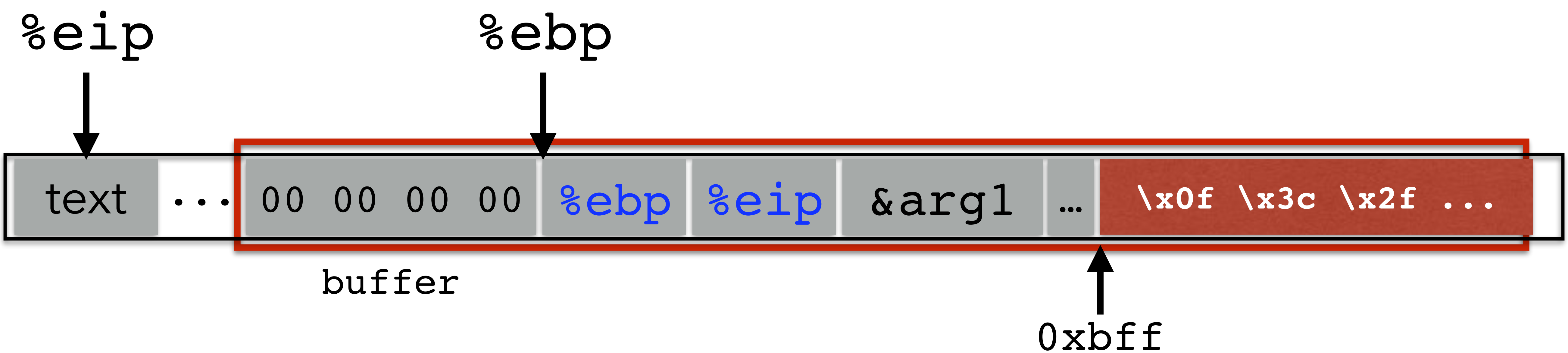
Called function:

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 %ebb

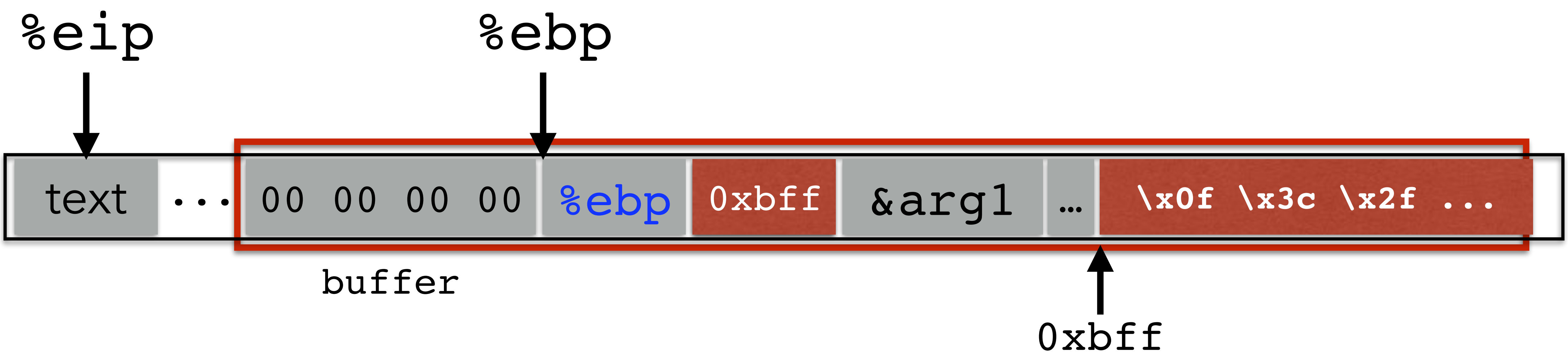
Returning function:

7. **Reset the previous stack frame**: %ebp = (%ebp)
8. **Jump back to return address**: %eip = 4(%ebp)

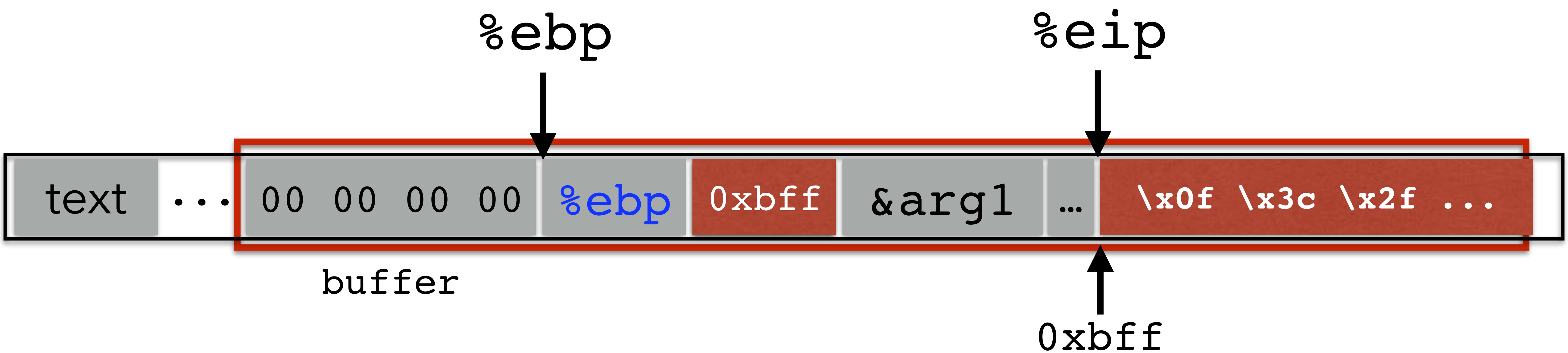
HIJACKING THE SAVED %EIP



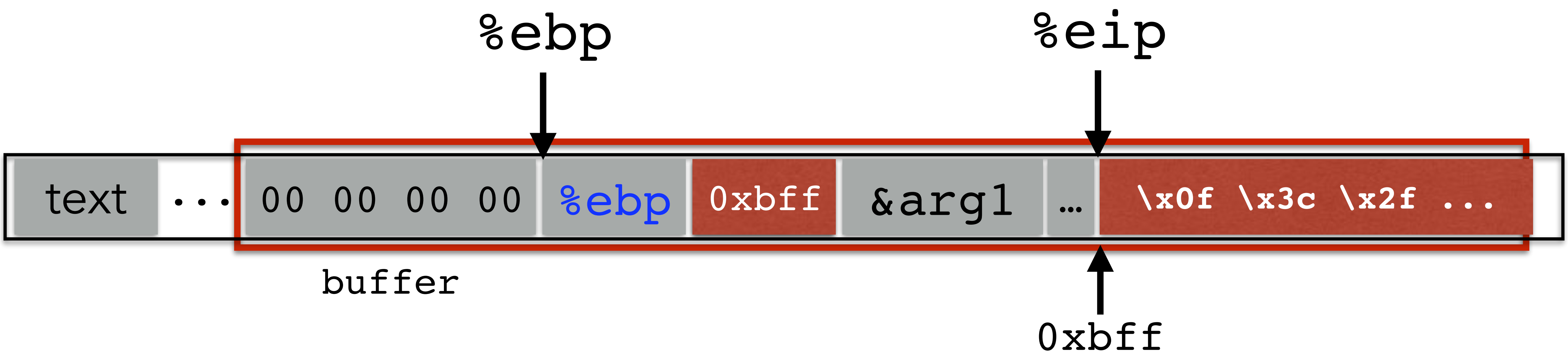
HIJACKING THE SAVED %EIP



HIJACKING THE SAVED %EIP



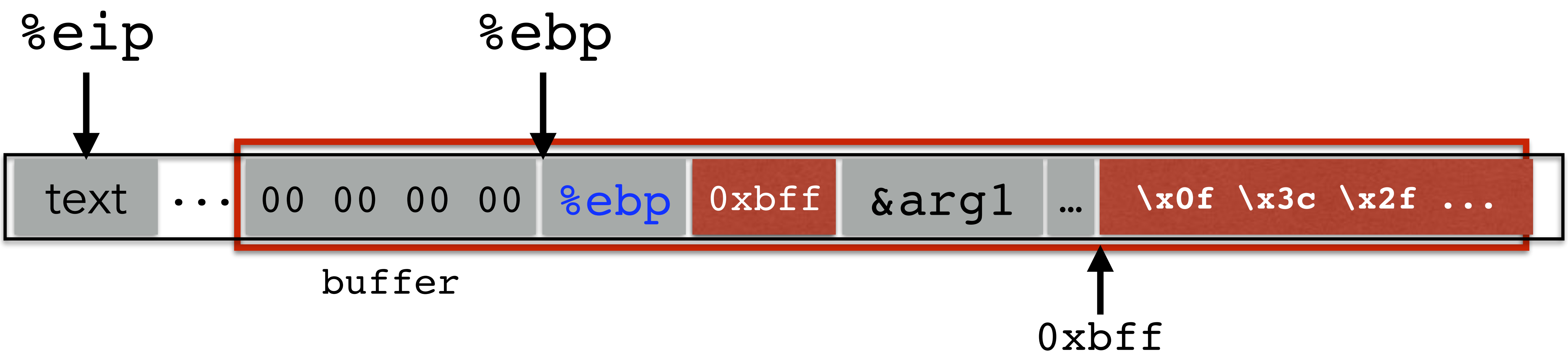
HIJACKING THE SAVED %EIP



But how do we know the address?

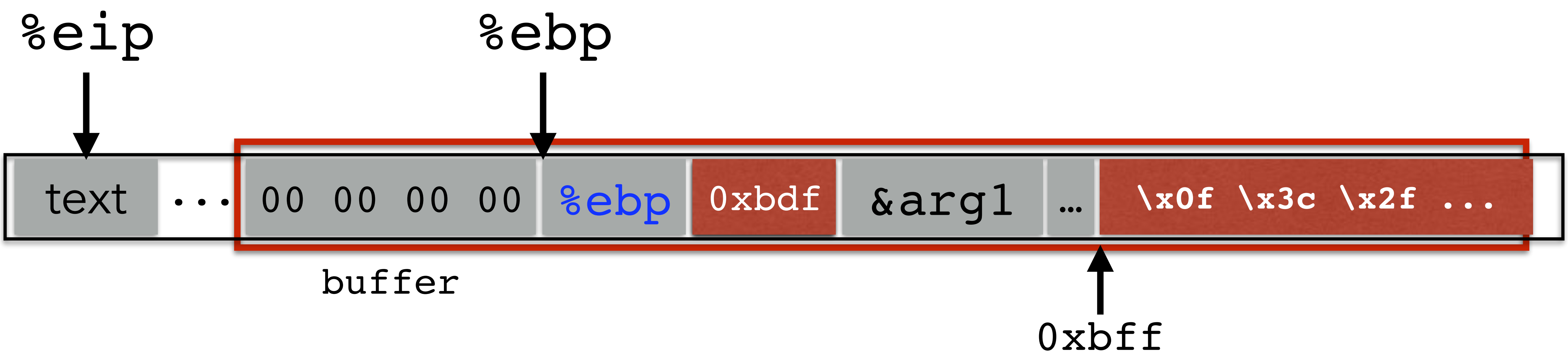
HIJACKING THE SAVED %EIP

What if we are wrong?



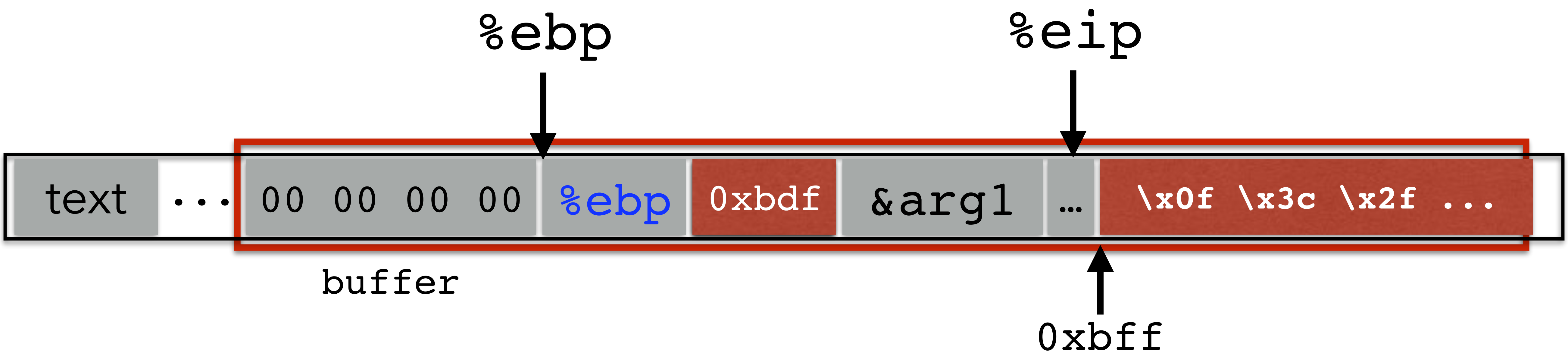
HIJACKING THE SAVED %EIP

What if we are wrong?



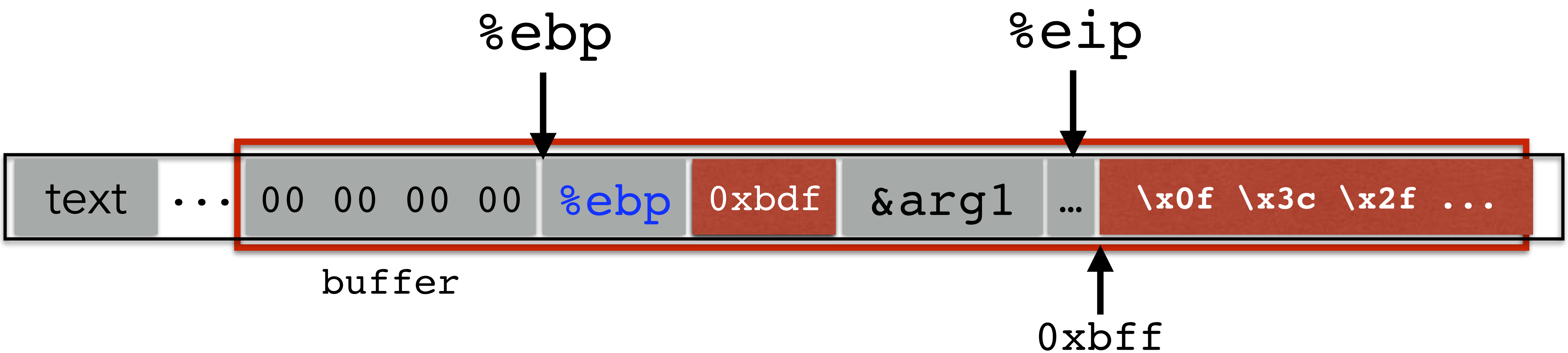
HIJACKING THE SAVED %EIP

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

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

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!

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!

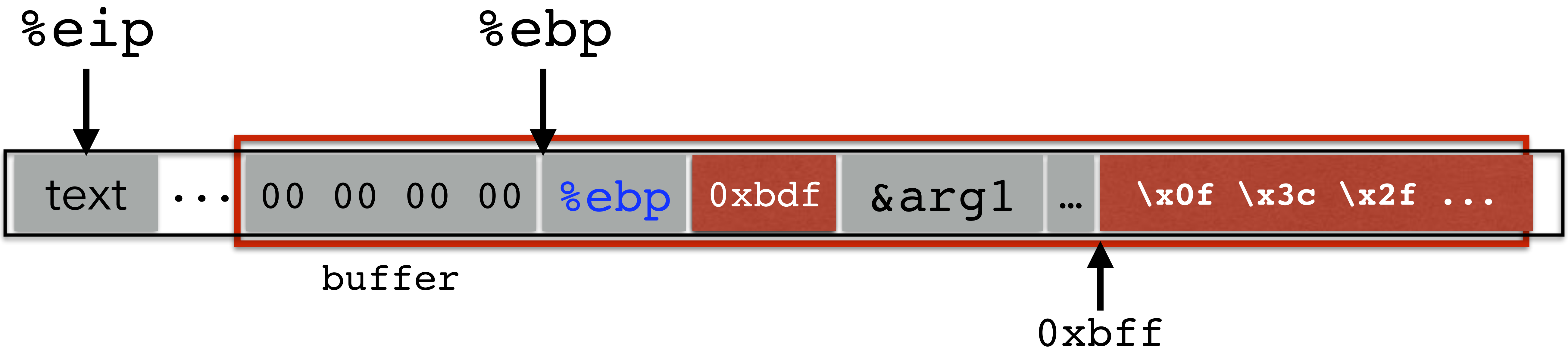
- Worst case scenario: it's a 32 (or 64) bit memory space, which means 2^{32} (2^{64}) possible answers

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} (2^{64}) 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)

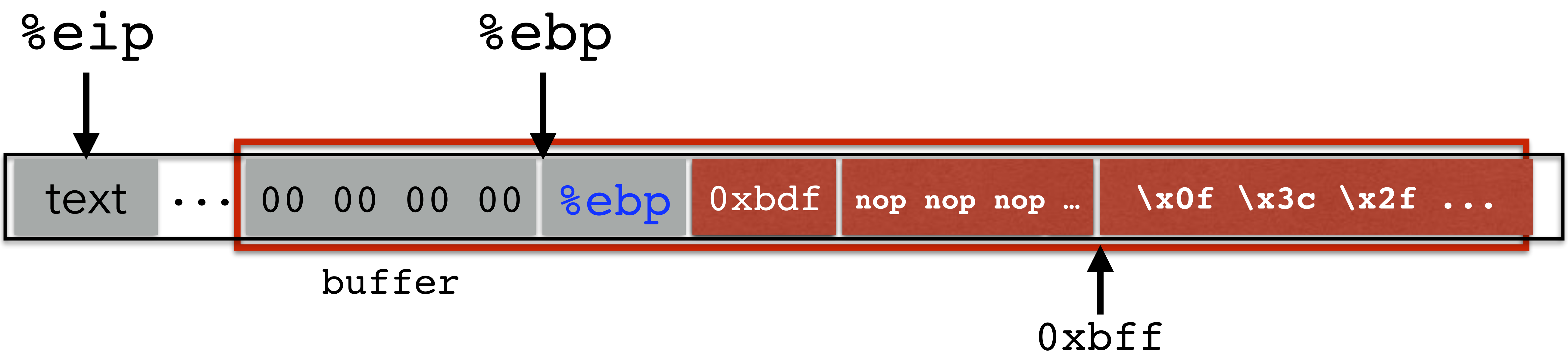
IMPROVING OUR CHANCES: NOP SLEDS

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



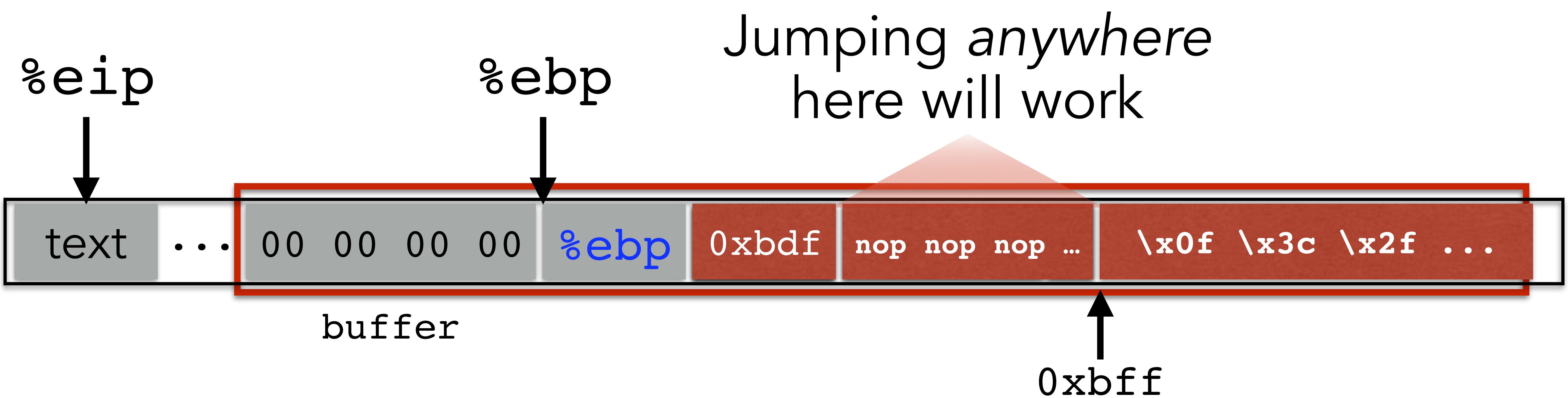
IMPROVING OUR CHANCES: NOP SLEDS

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



IMPROVING OUR CHANCES: NOP SLEDS

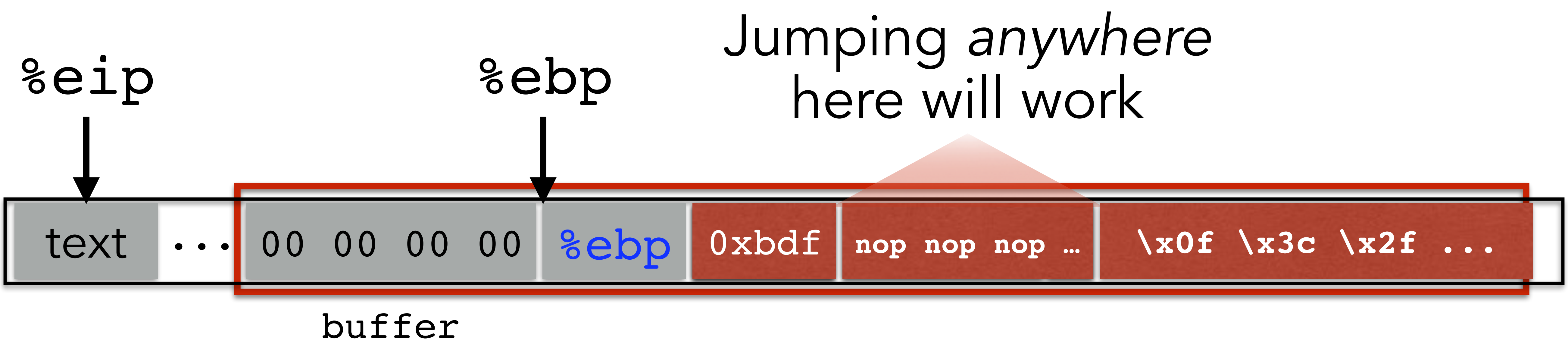
nop is a single-byte instruction
(just moves to the next instruction)



IMPROVING OUR CHANCES: NOP SLEDS

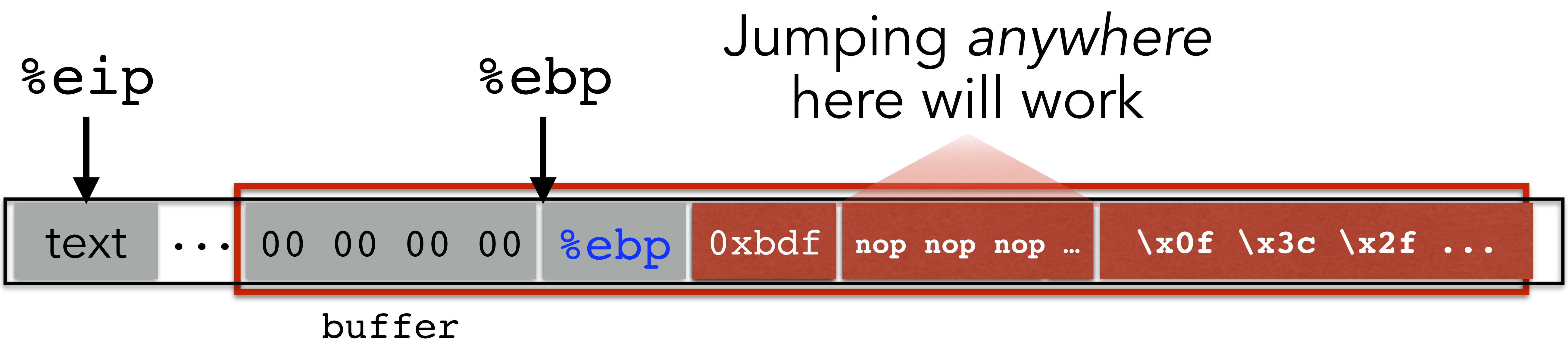
nop is a single-byte instruction

(just moves to the next instruction)



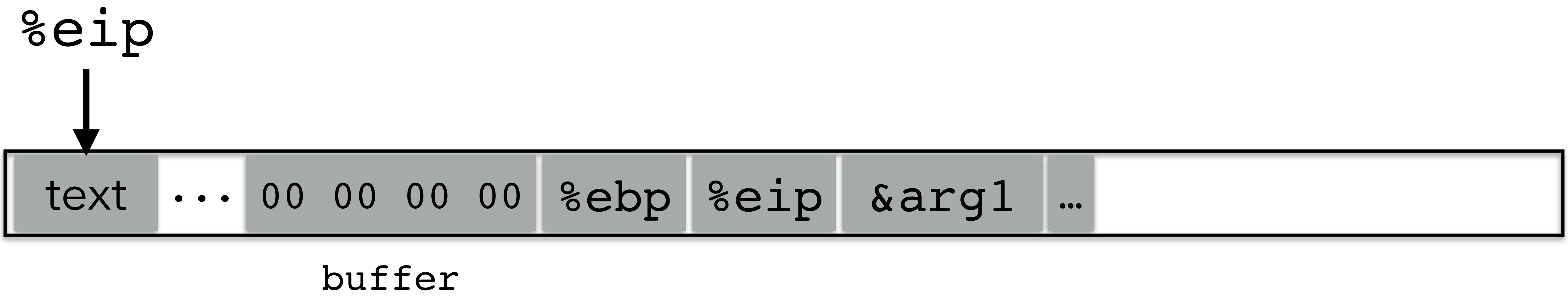
IMPROVING OUR CHANCES: NOP SLEDS

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



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

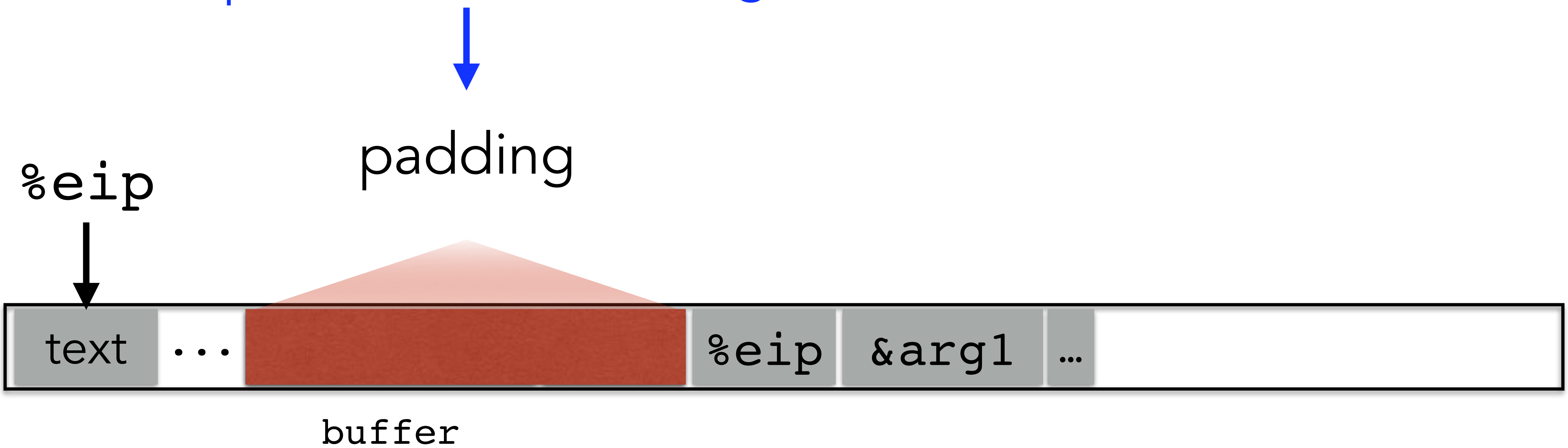
BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER



BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

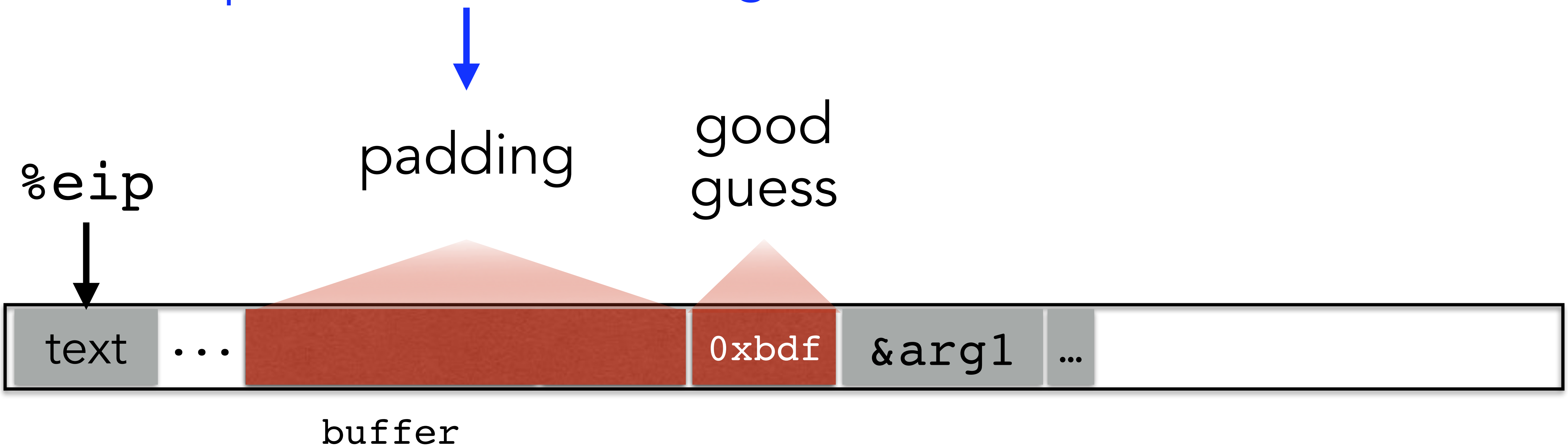
BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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



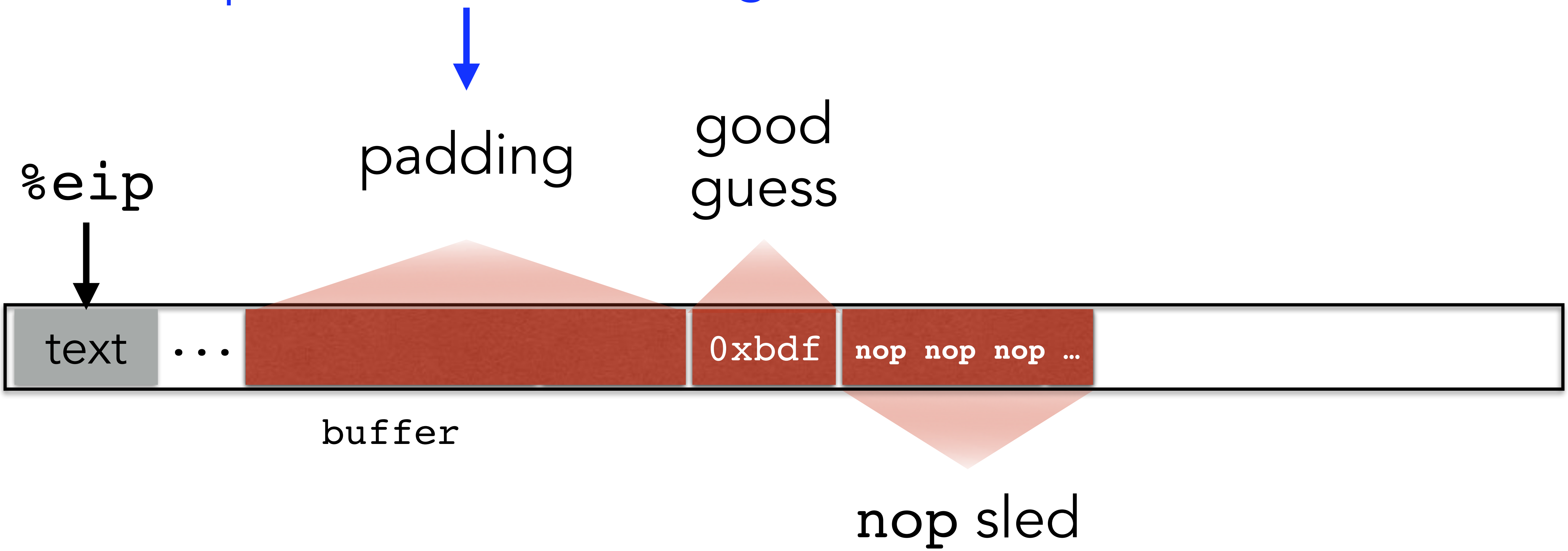
BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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



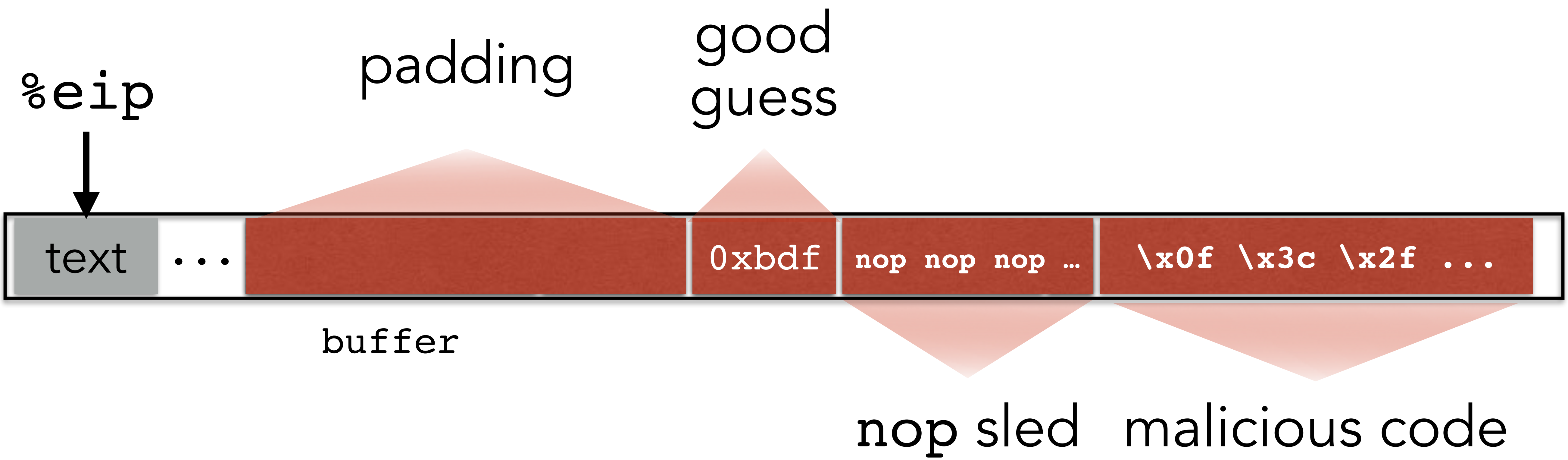
BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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



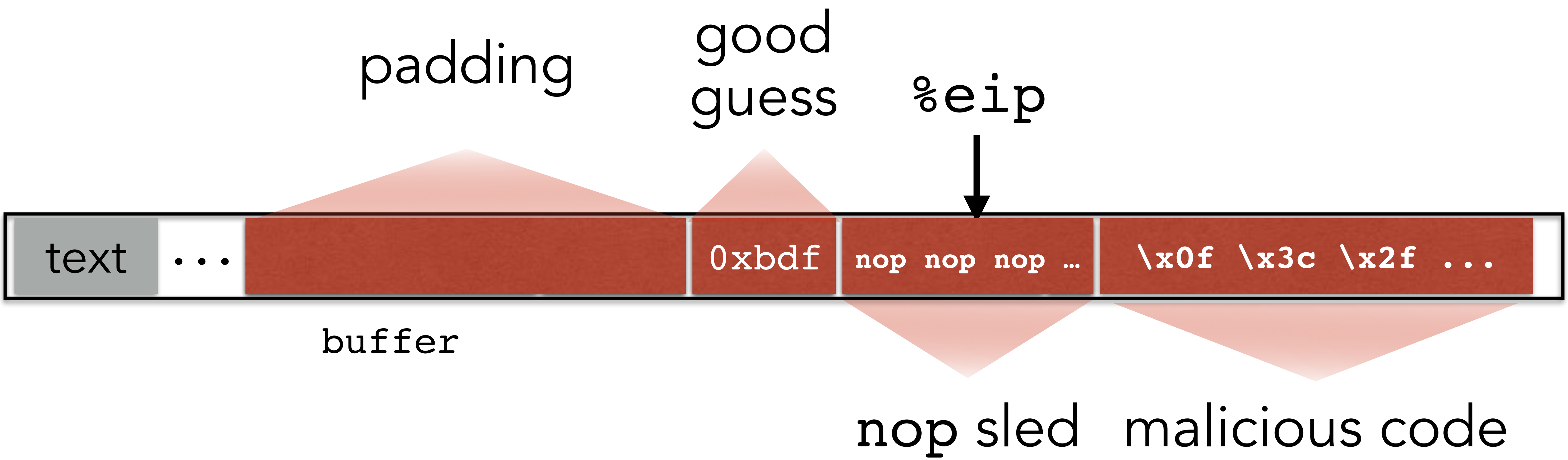
BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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



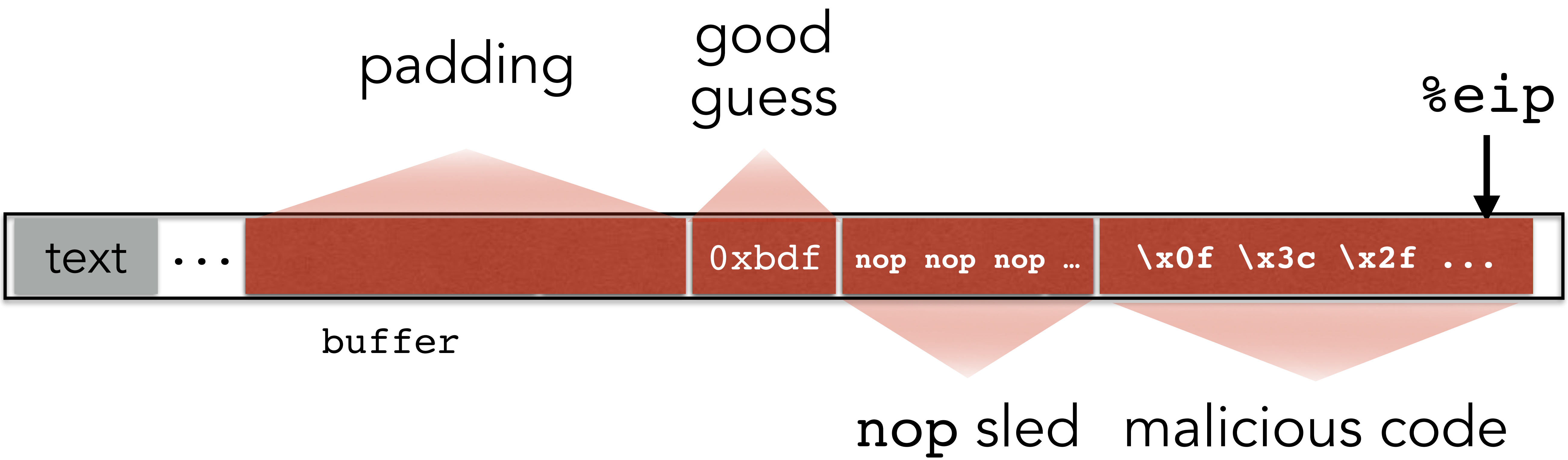
BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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



BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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



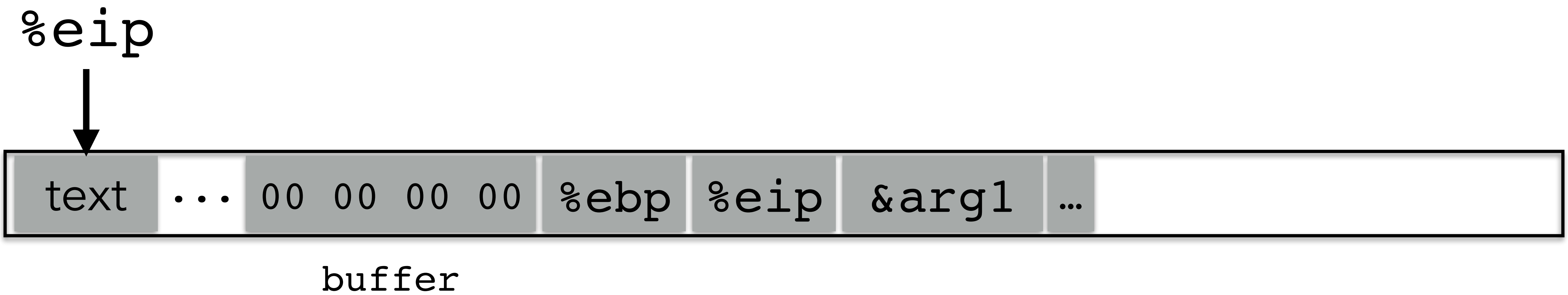
BUFFER OVERFLOW DEFENSES

RECALL OUR CHALLENGES

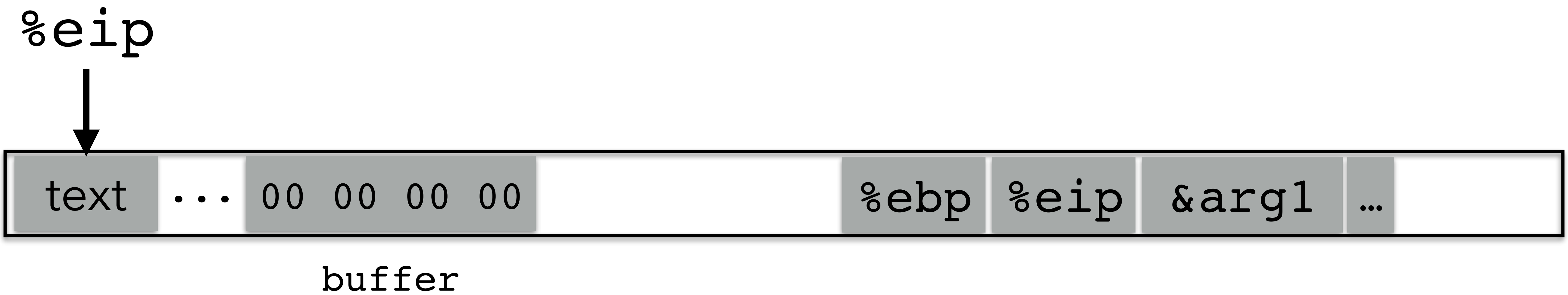
How can we make these even more difficult?

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

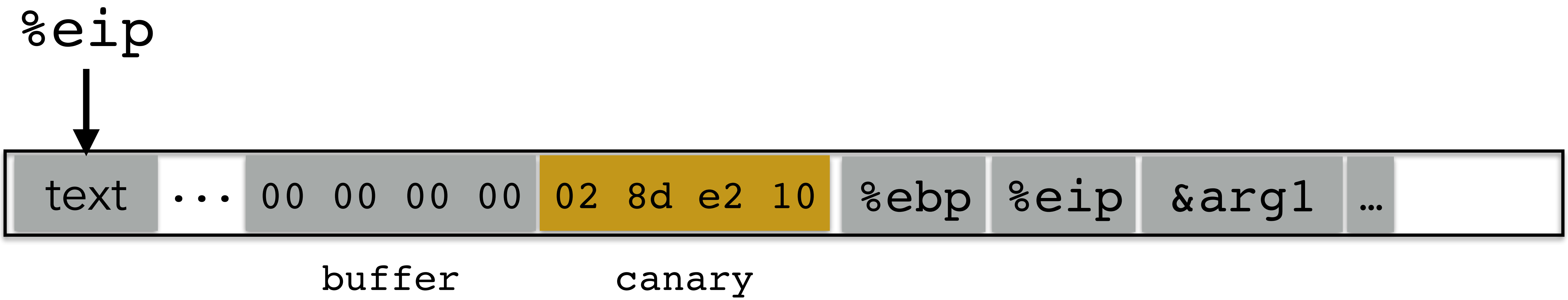
DETECTING OVERFLOWS WITH CANARIES



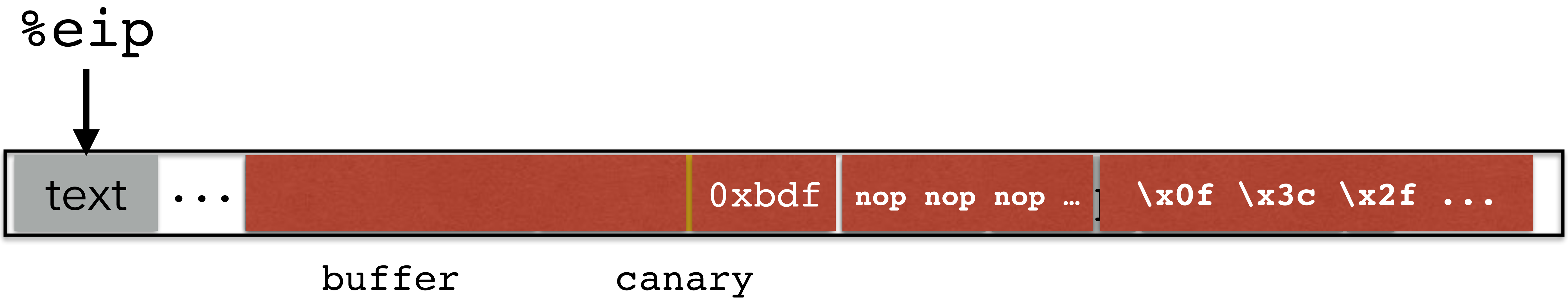
DETECTING OVERFLOWS WITH CANARIES



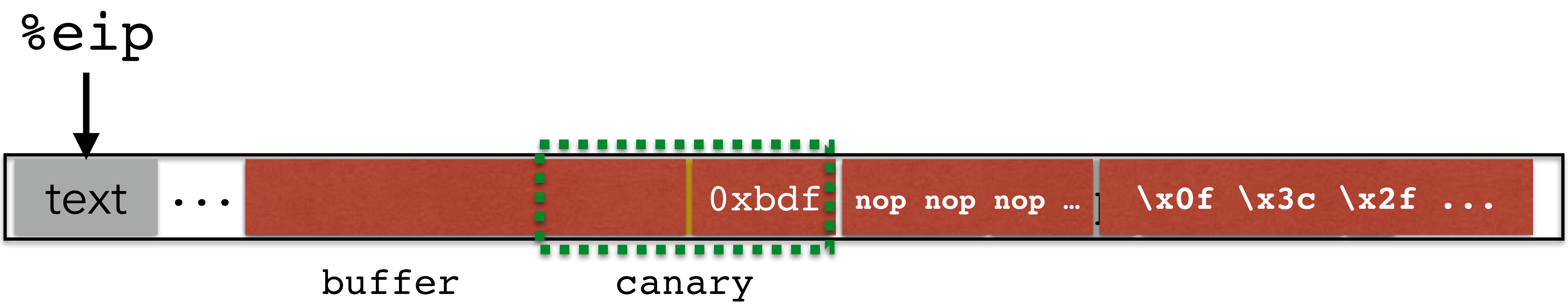
DETECTING OVERFLOWS WITH CANARIES



DETECTING OVERFLOWS WITH CANARIES

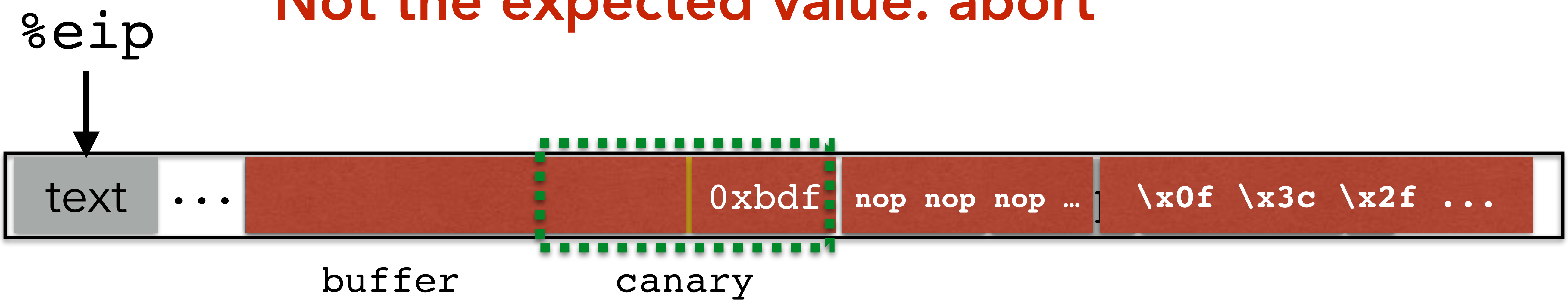


DETECTING OVERFLOWS WITH CANARIES



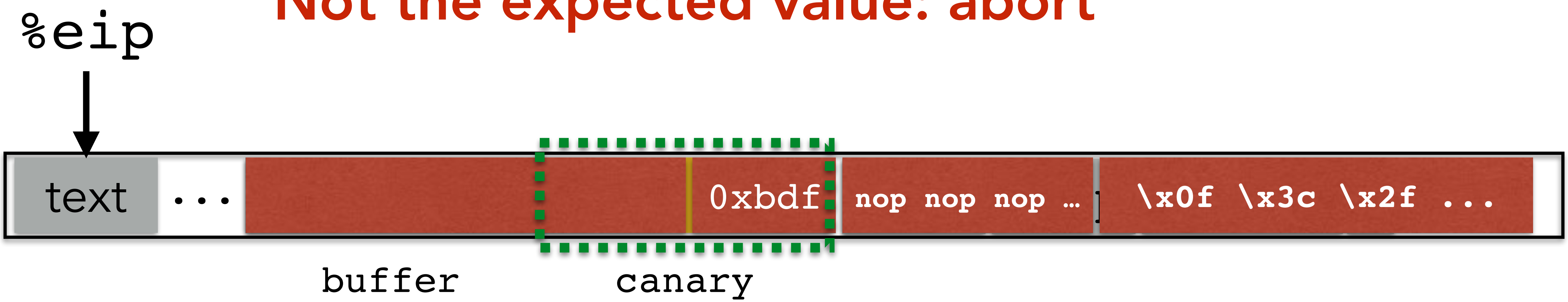
DETECTING OVERFLOWS WITH CANARIES

Not the expected value: abort



DETECTING OVERFLOWS WITH CANARIES

Not the expected value: abort



What value should the canary have?

CANARY VALUES

From StackGuard [Wagle & Cowan]

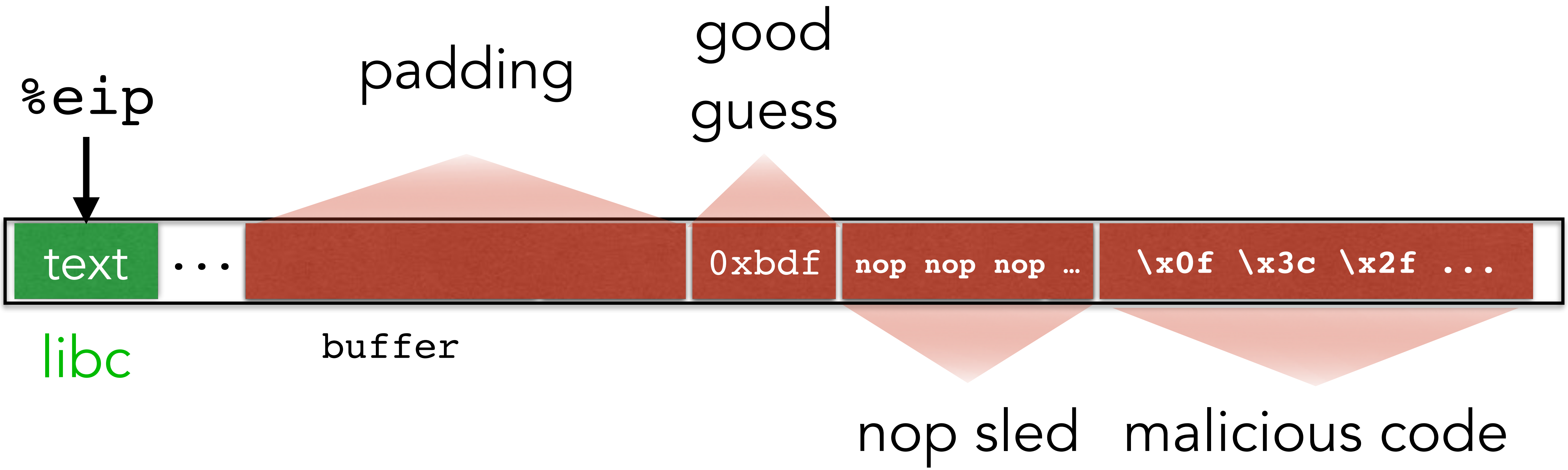
1. Terminator canaries (CR, LF, NULL, -1)
   - Leverages the fact that scanf etc. don’t allow these
2. Random canaries
   - Write a new random value @ each process start
   - Save the real value somewhere in memory
   - Must write-protect the stored value
3. Random XOR canaries
   - Same as random canaries
   - But store canary XOR some control info, instead

RECALL OUR CHALLENGES

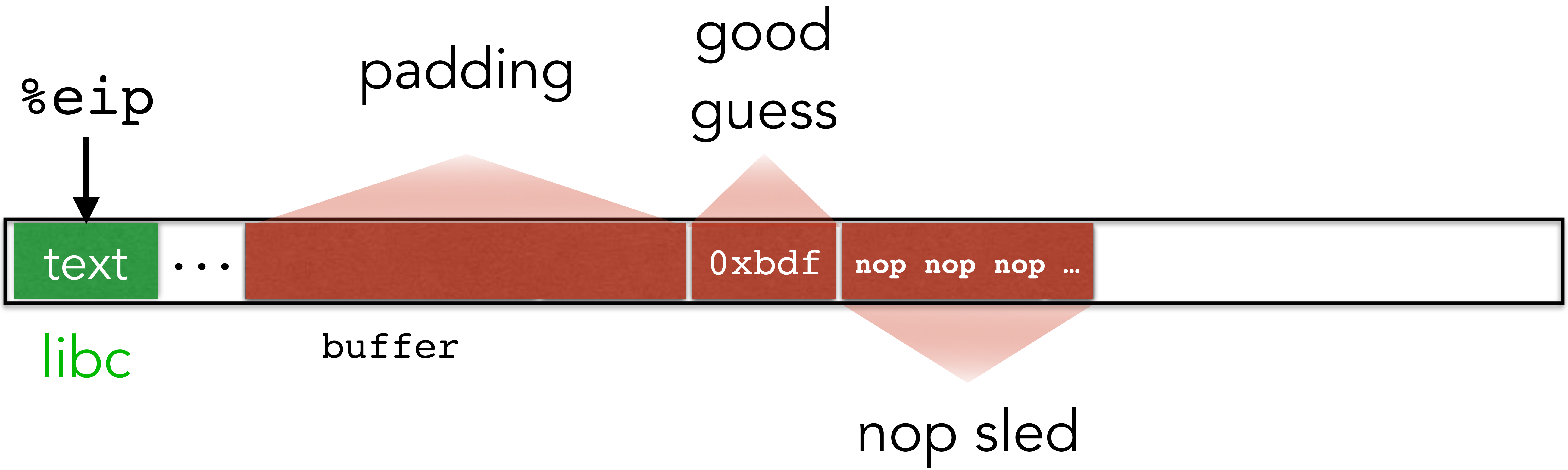
How can we make these even more difficult?

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

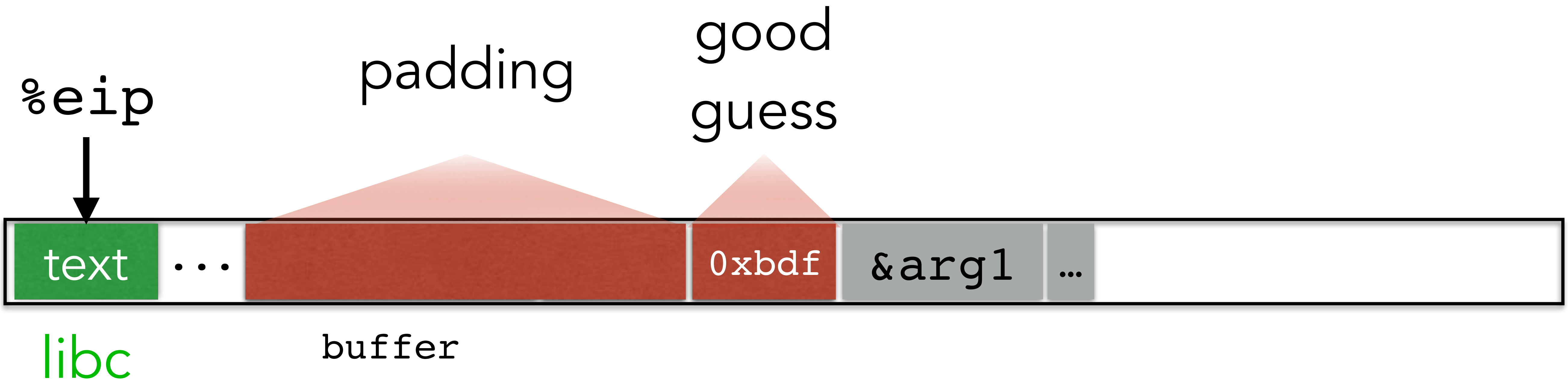
RETURN TO LIBC



RETURN TO LIBC

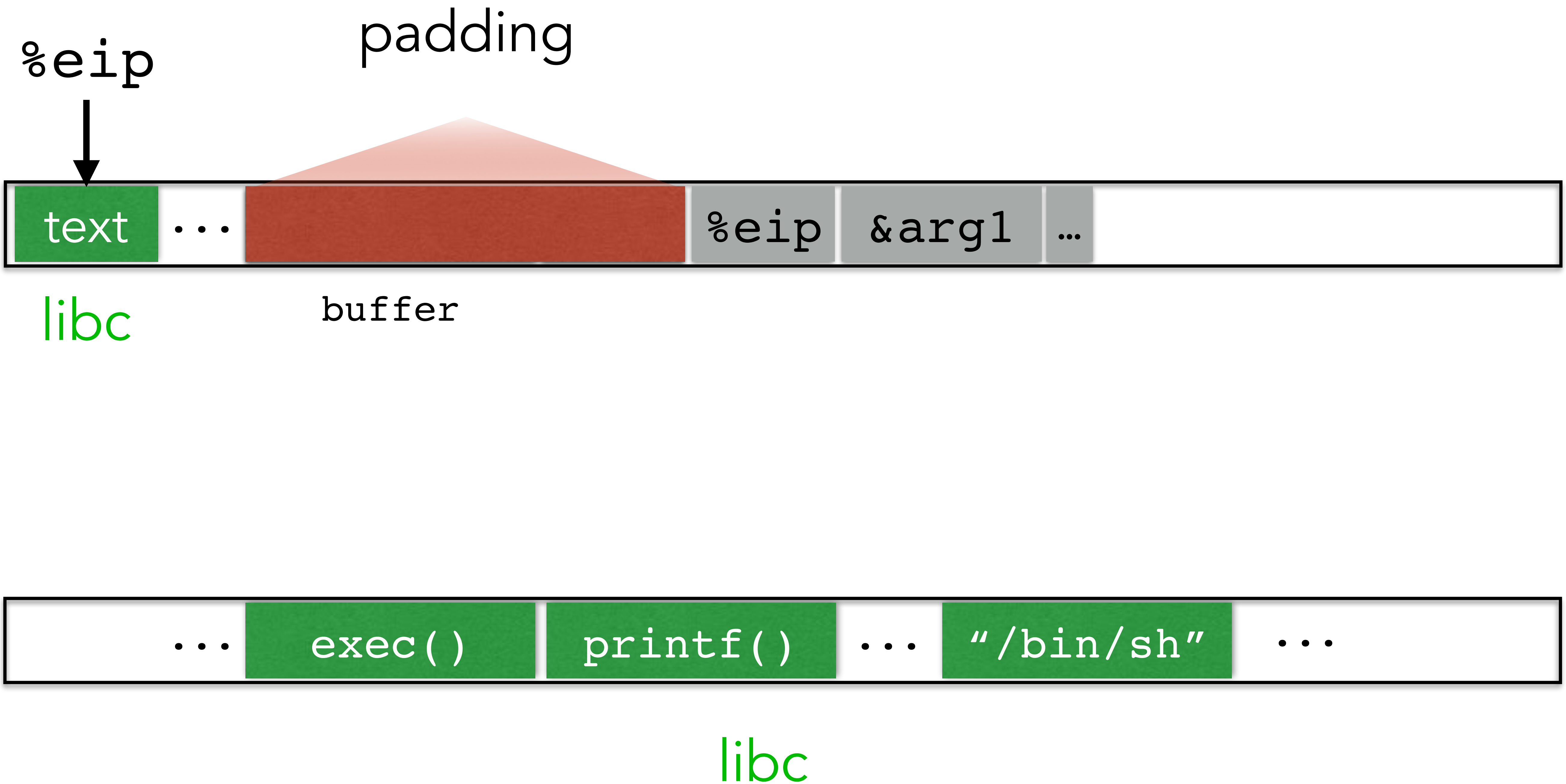


RETURN TO LIBC

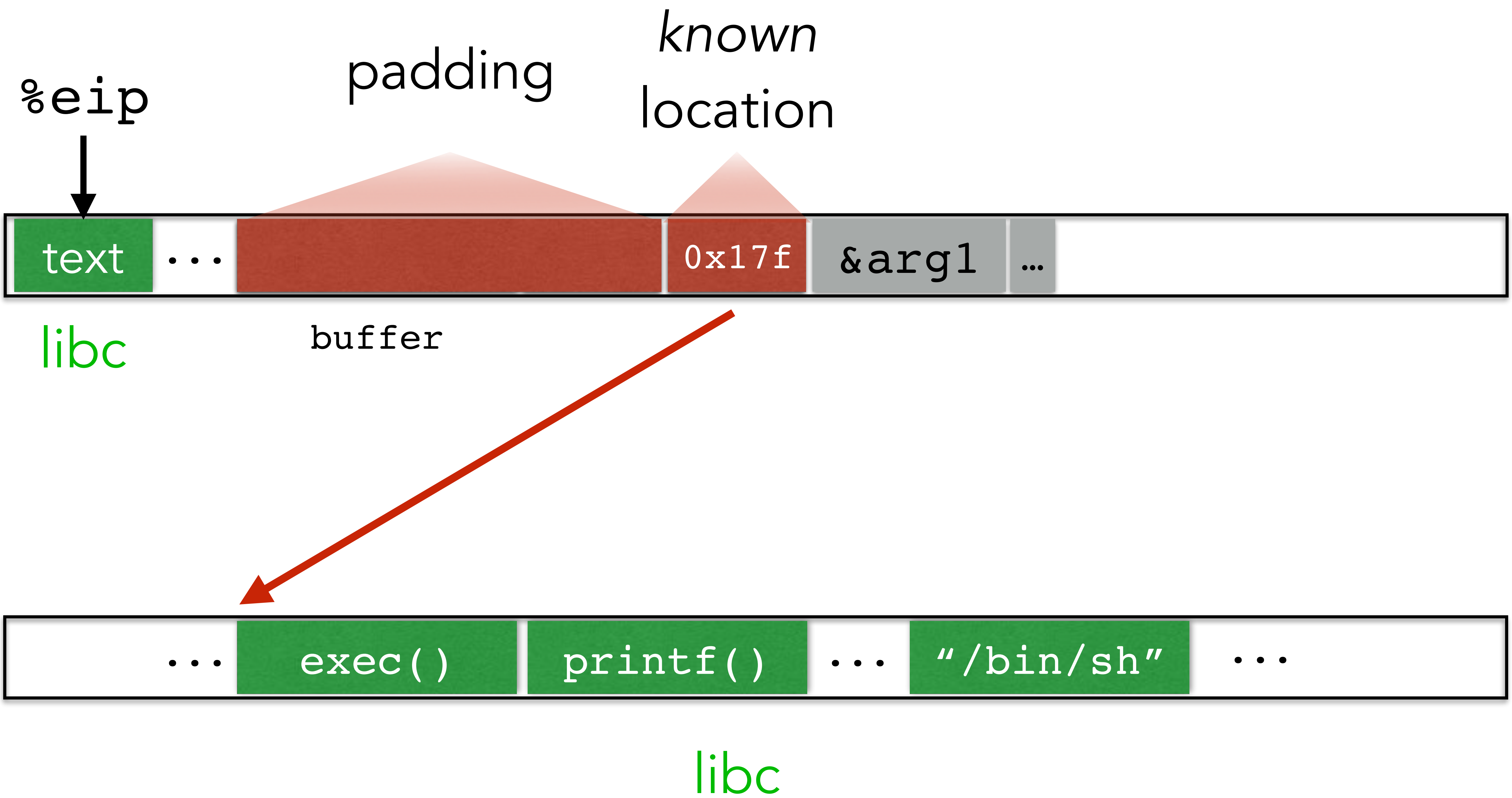


RETURN TO LIBC

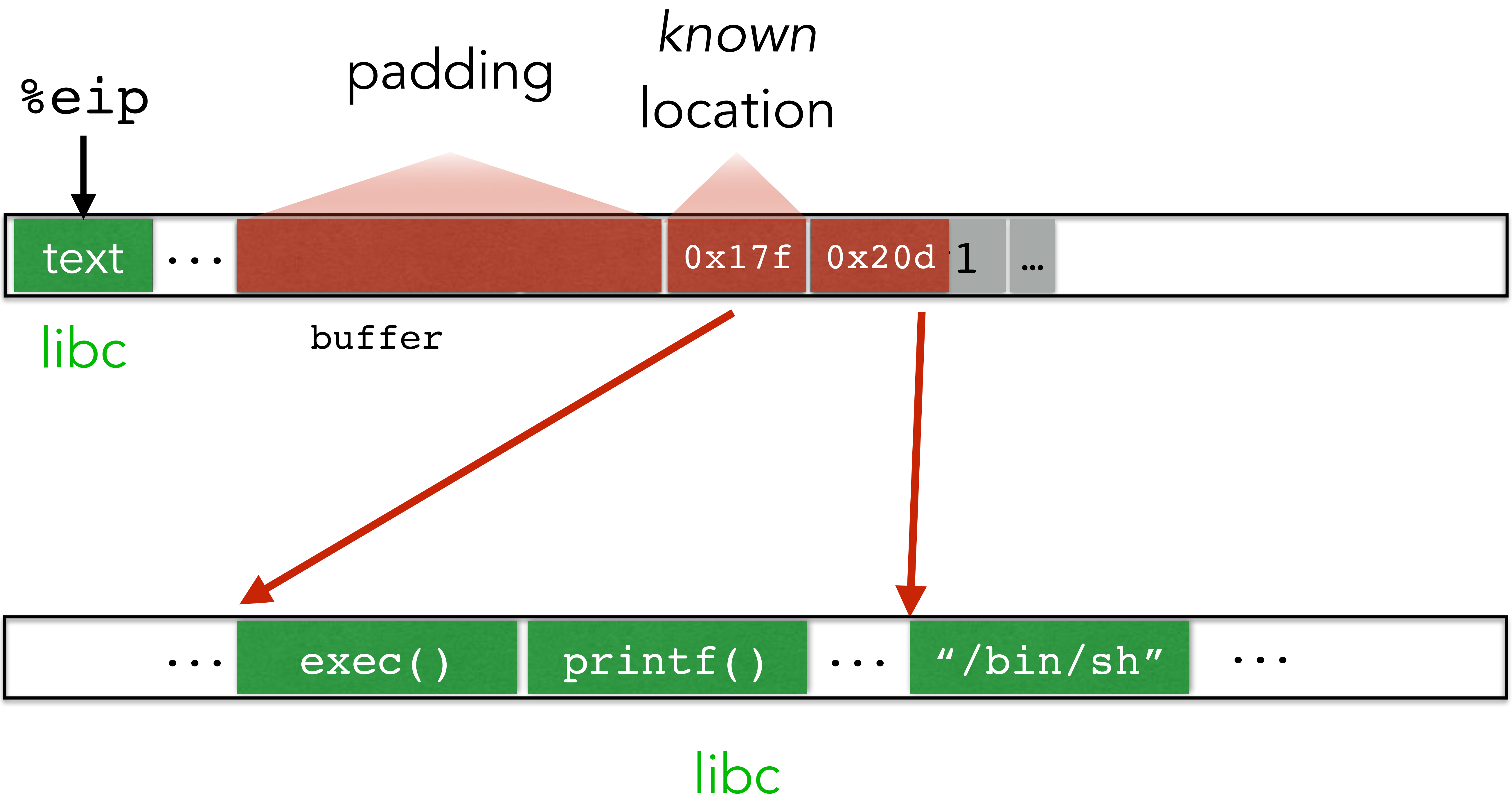
RETURN TO LIBC



RETURN TO LIBC



RETURN TO LIBC



RECALL OUR CHALLENGES

How can we make these even more difficult?

- Putting code into the memory (no zeroes)

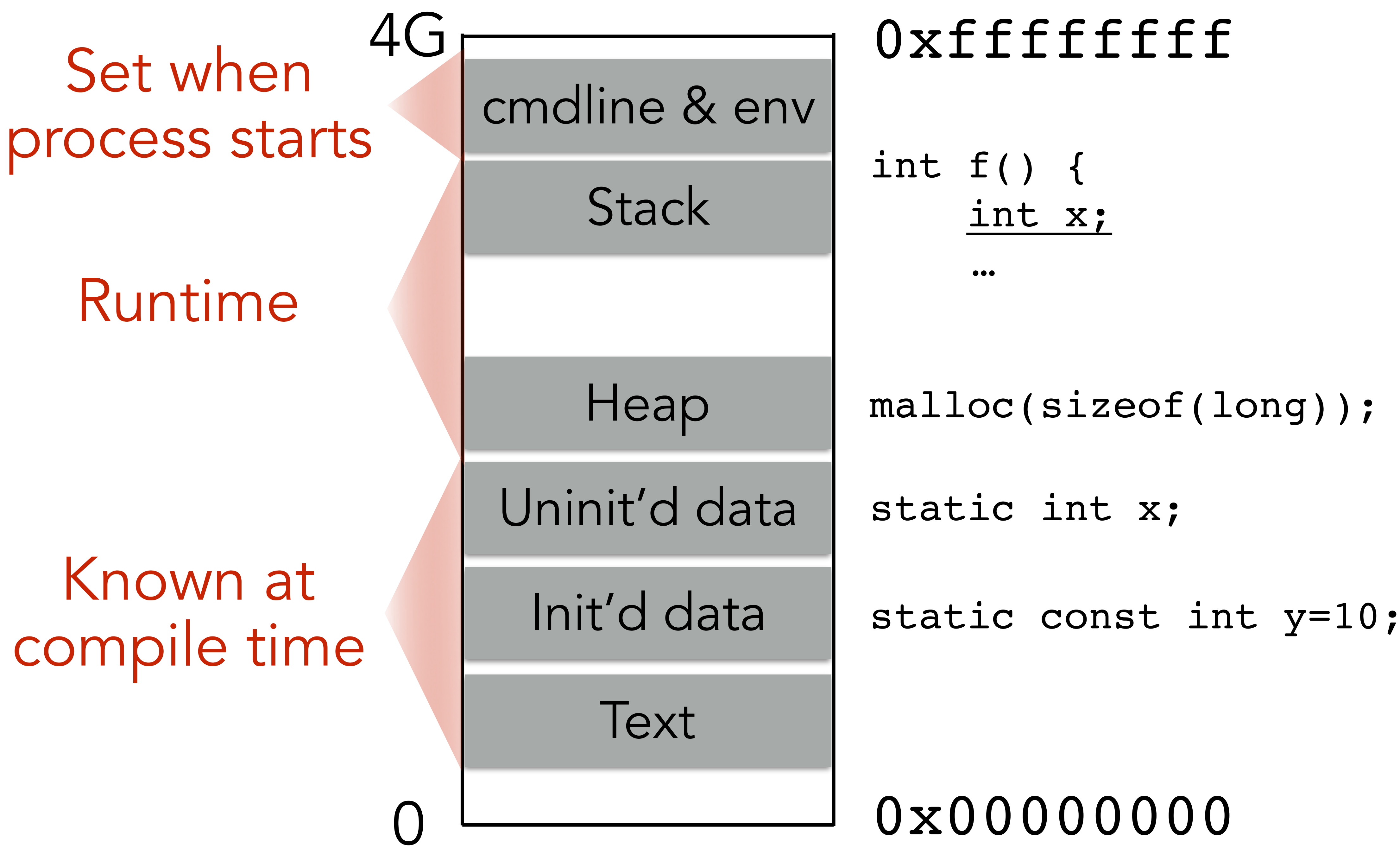
  Option: Make this detectable with canaries

- Getting %eip to point to our code (dist buff to stored eip)

  Non-executable stack doesn't work so well

- Finding the return address (guess the raw address)

ADDRESS SPACE LAYOUT RANDOMIZATION



Randomize where exactly these regions start

ADDRESS SPACE LAYOUT RANDOMIZATION

On the Effectiveness of Address-Space Randomization

Hovav Shacham, Stanford University, hovav@cs.stanford.edu
Matthew Page, Stanford University, mpage@stanford.edu
Ben Pfaff, Stanford University, blp@cs.stanford.edu
Eu-Jin Goh, Stanford University, eujin@cs.stanford.edu
Nagendra Modadugu, Stanford University, nagendra@cs.stanford.edu
Dan Boneh, Stanford University, dabo@cs.stanford.edu

ABSTRACT

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

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

Categories and Subject Descriptors

D.4.6 [Operating Systems]: Security and Protection

General Terms

Security, Measurement

Keywords

Address-space randomization, diversity, automated attacks

1. INTRODUCTION

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

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

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

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

Shortcomings of ASLR

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

RECALL OUR CHALLENGES

How can we make these even more difficult?

- Putting code into the memory (no zeroes)

    Option: Make this detectable with canaries

- Getting %eip to point to our code (dist buff to stored eip)

    Non-executable stack doesn't work so well

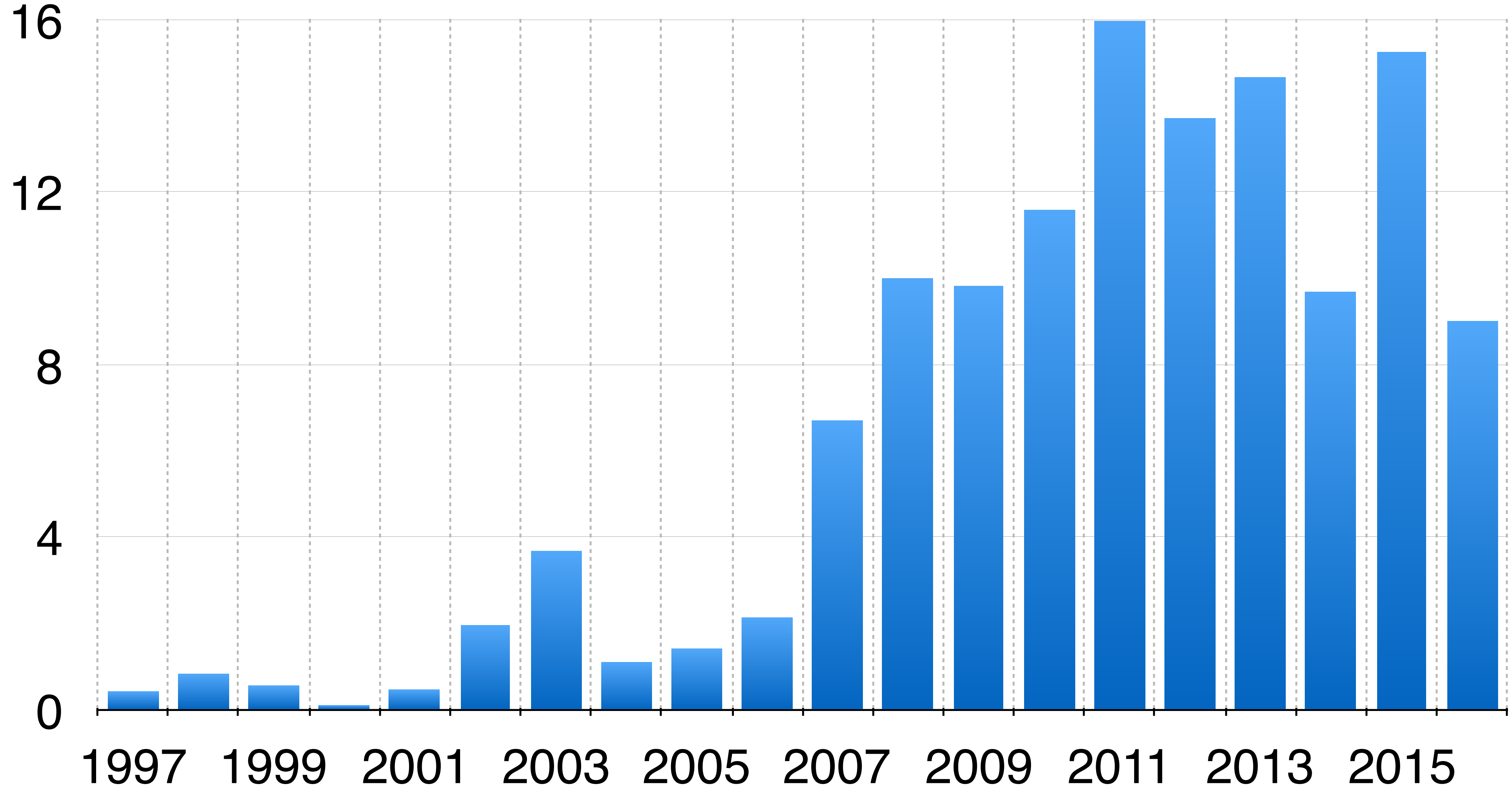
- Finding the return address (guess the raw address)

    Address Space Layout Randomization (**ASLR**)

Best defense: Good programming practices

BUFFER OVERFLOW PREVALENCE

Significant percent of *all* vulnerabilities



[Data from the National Vulnerability Database]

```c
void safe()
{
    char buf[80];
    fgets(buf, 80, stdin);
}
```

```c
void safer()
{
    char buf[80];
    fgets(buf, sizeof(buf), stdin);
}
```

```c
void safe()
{
    char buf[80];
    fgets(buf, 80, stdin);
}
```

```c
void safer()
{
    char buf[80];
    fgets(buf, sizeof(buf), stdin);
}
```

```c
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

```c
void safe()
{
    char buf[80];
    fgets(buf, 80, stdin);
}
```

```c
void safer()
{
    char buf[80];
    fgets(buf, sizeof(buf), stdin);
}
```

```c
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

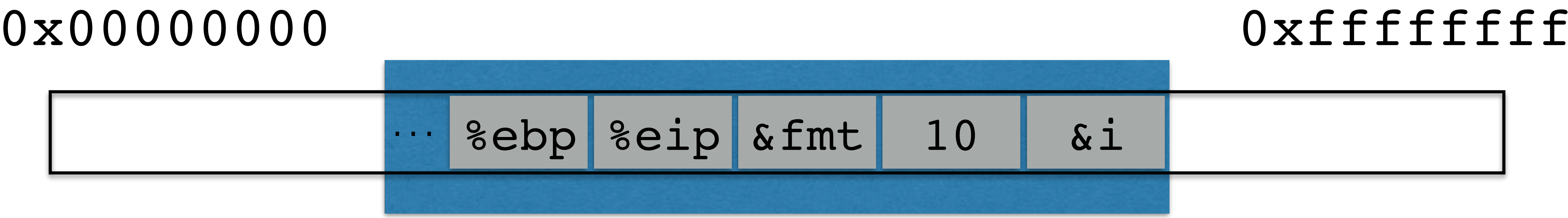
FORMAT STRING
VULNERABILITIES

PRINTF FORMAT STRINGS

```
int i = 10;
printf(“%d %p\n”, i, &i);
```

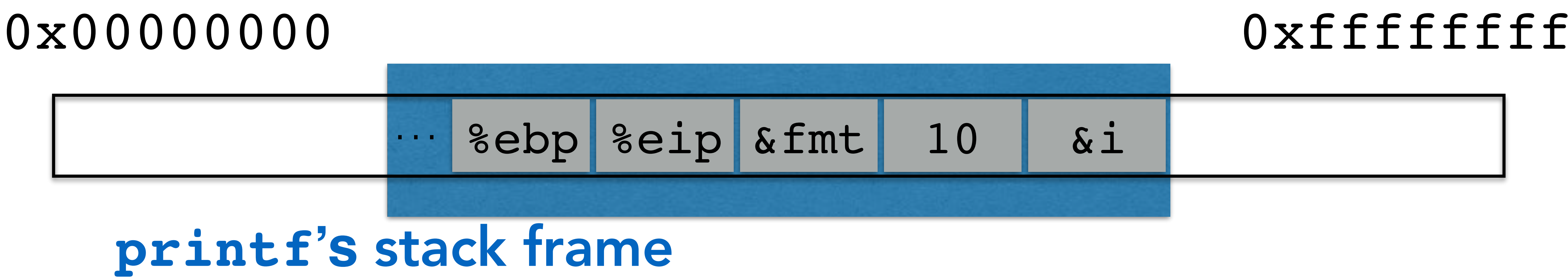
PRINTF FORMAT STRINGS

```
int i = 10;
printf("%d %p\n", i, &i);
```



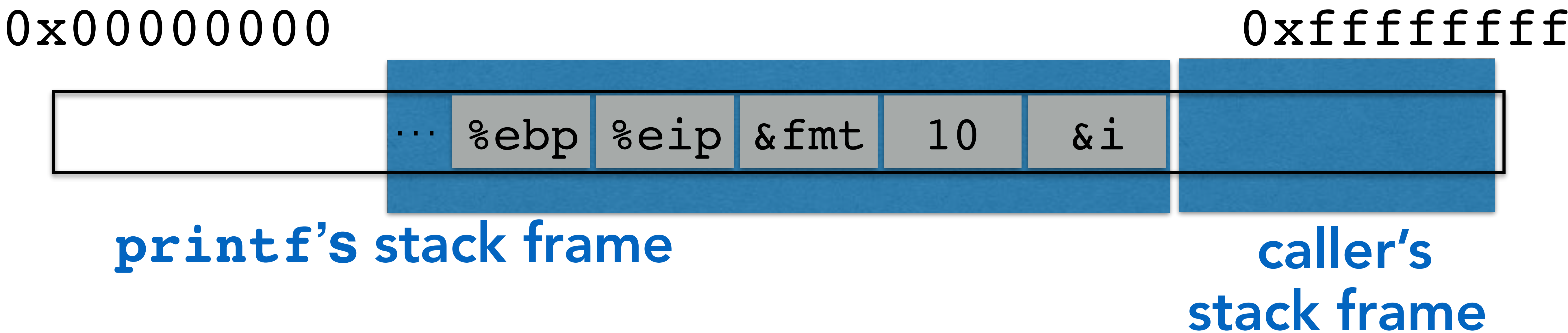
PRINTF FORMAT STRINGS

```
int i = 10;
printf("%d %p\n", i, &i);
```



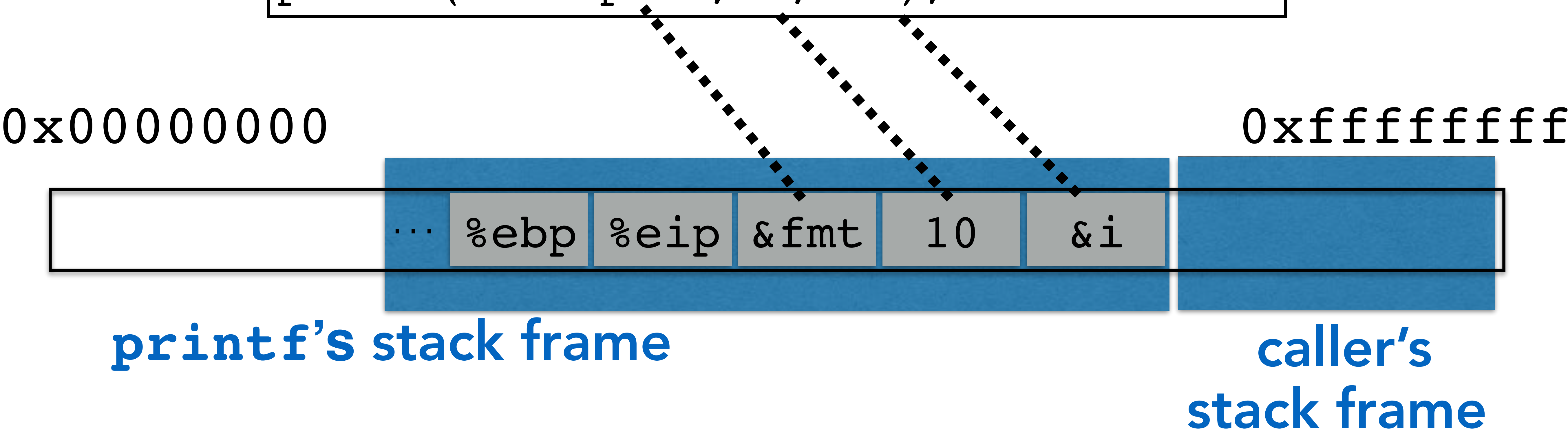
PRINTF FORMAT STRINGS

```
int i = 10;
printf("%d %p\n", i, &i);
```

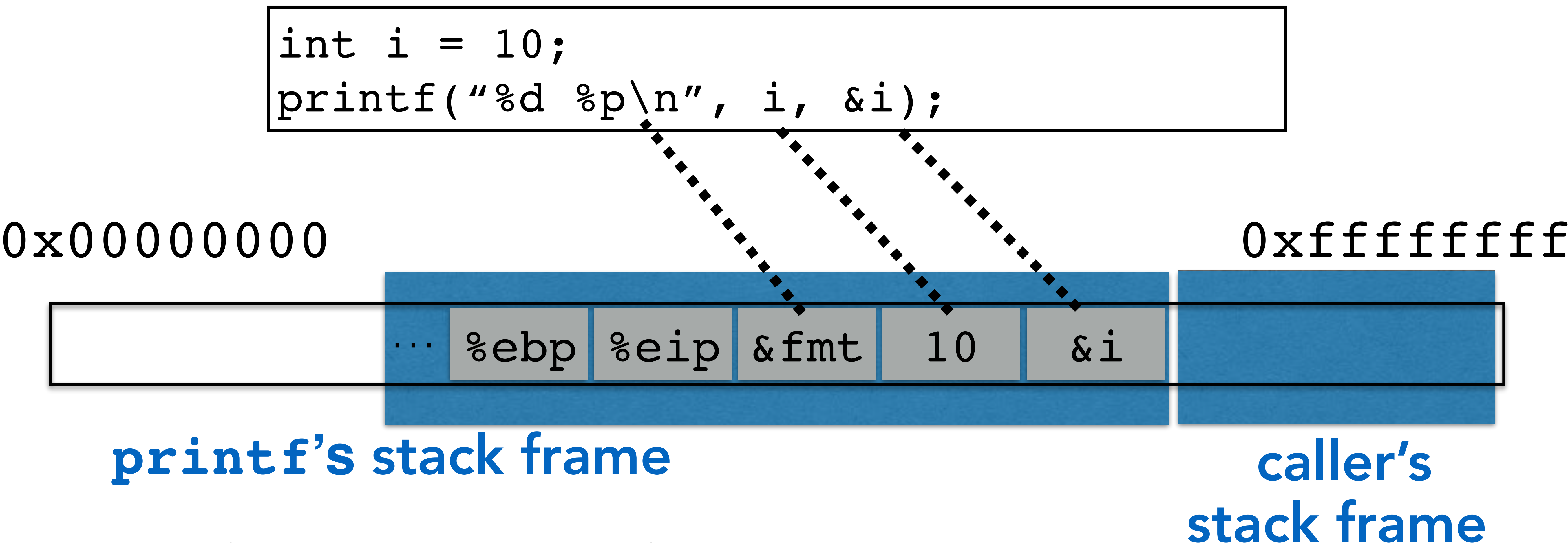


PRINTF FORMAT STRINGS

```
int i = 10;
printf("%d %p\n", i, &i);
```



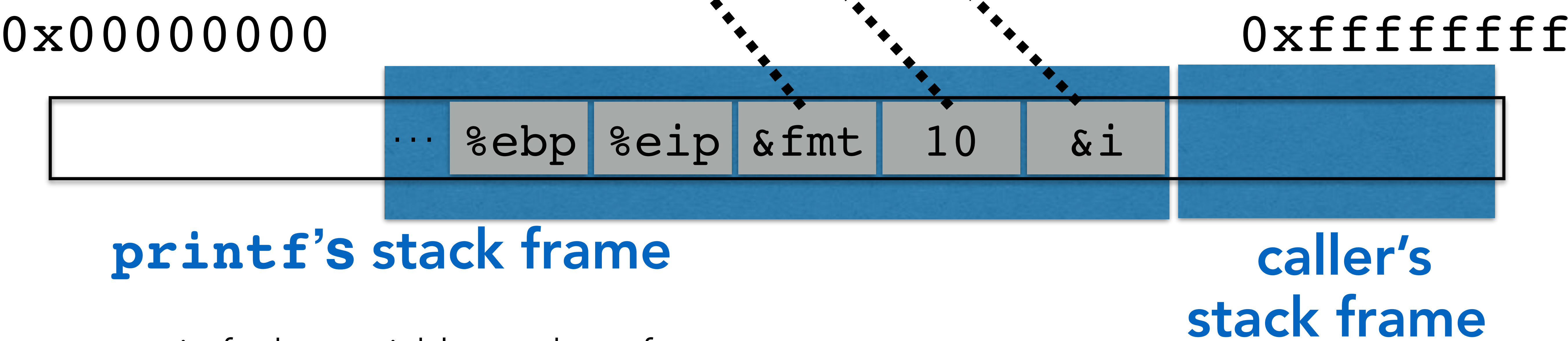
PRINTF FORMAT STRINGS



- printf takes variable number of arguments
- printf pays no mind to where the stack frame "ends"
- It presumes that you called it with (at least) as many arguments as specified in the format string

PRINTF FORMAT STRINGS

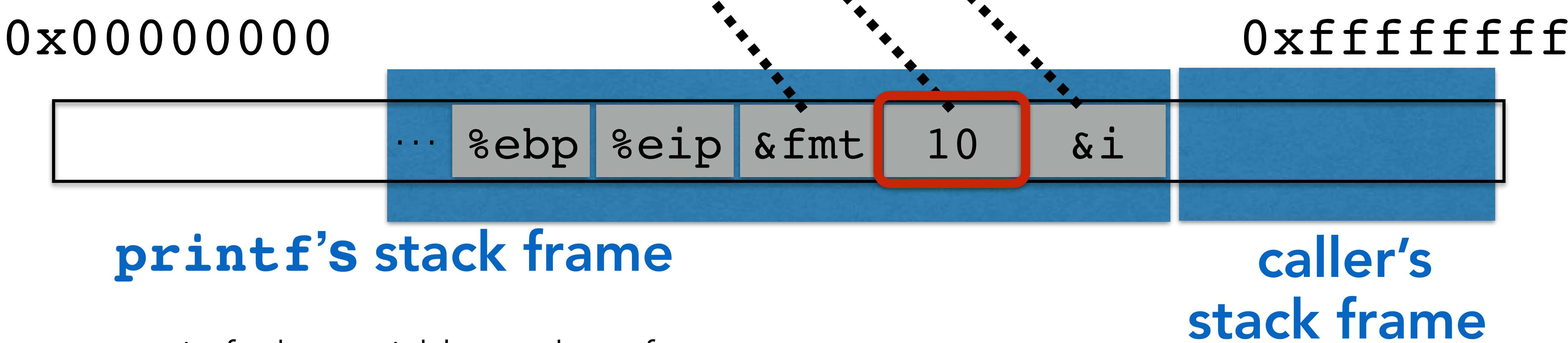
```
int i = 10;
printf("%d %p\n", i, &i);
```



- printf takes variable number of arguments
- printf pays no mind to where the stack frame "ends"
- It presumes that you called it with (at least) as many arguments as specified in the format string

PRINTF FORMAT STRINGS

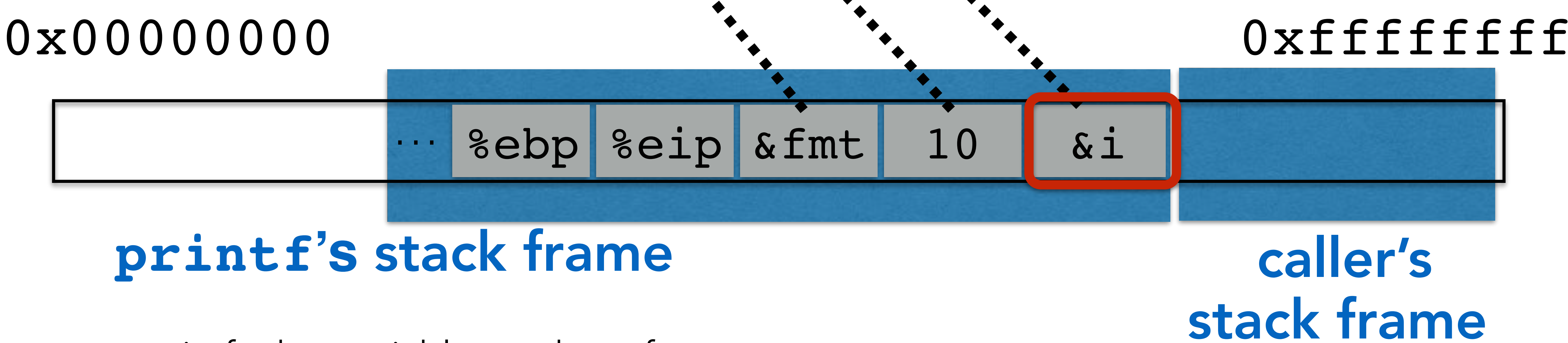
```
int i = 10;
printf("%d %p\n", i, &i);
```



- printf takes variable number of arguments
- printf pays no mind to where the stack frame "ends"
- It presumes that you called it with (at least) as many arguments as specified in the format string

PRINTF FORMAT STRINGS

```
int i = 10;
printf("%d %p\n", i, &i);
```



- printf takes variable number of arguments
- printf pays no mind to where the stack frame "ends"
- It presumes that you called it with (at least) as many arguments as specified in the format string

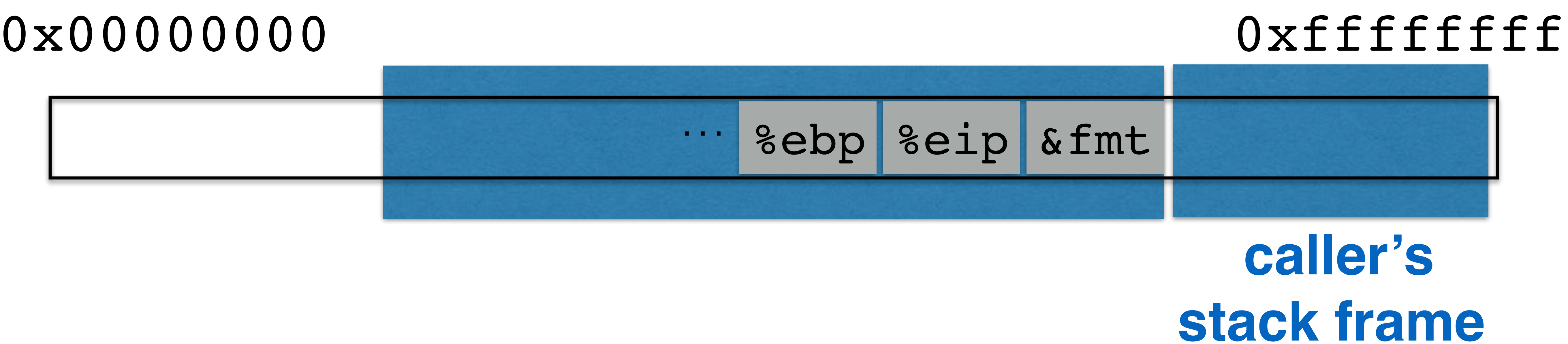
```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

```c
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

“%d %x"

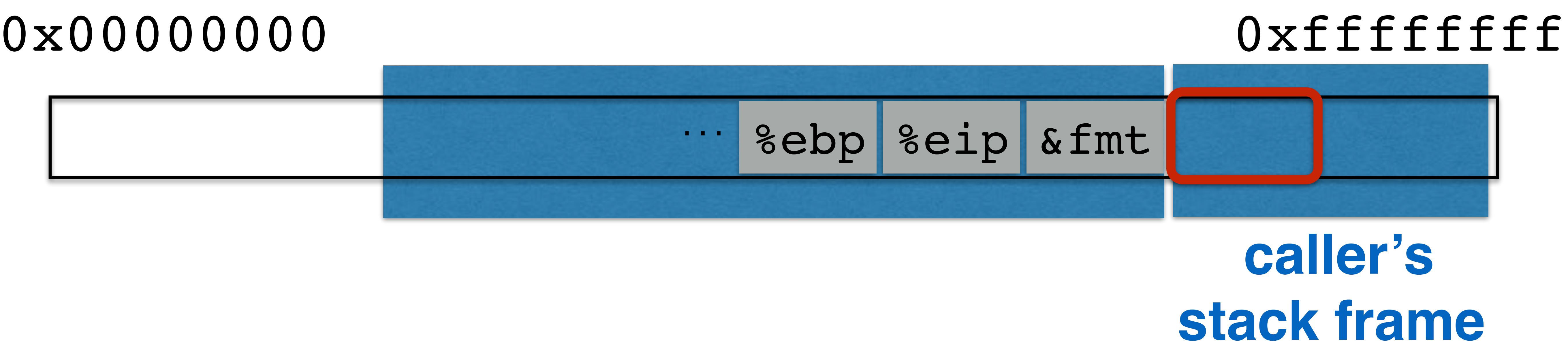
```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

“%d %x"



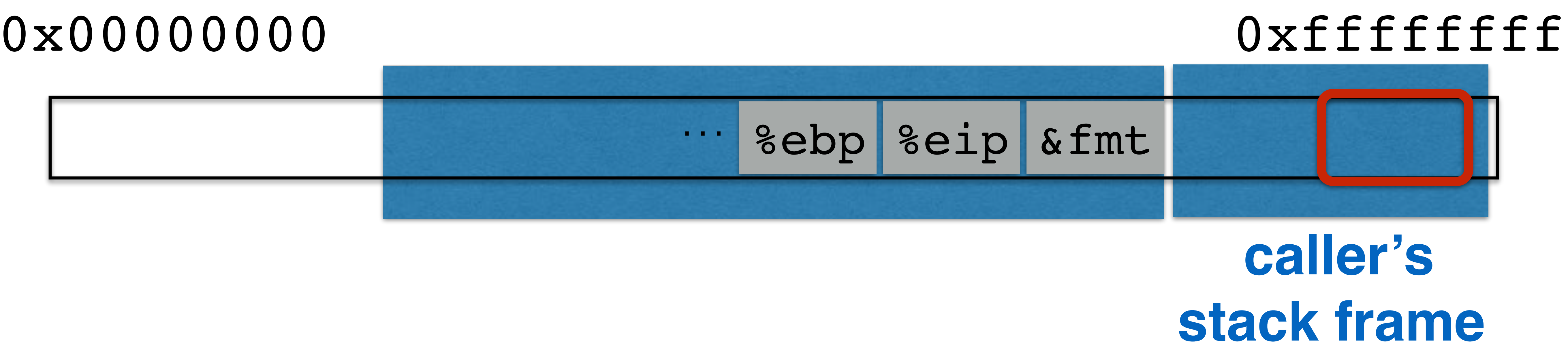
```c
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

“%d %x"



```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

"%d %x"



FORMAT STRING VULNERABILITIES

FORMAT STRING VULNERABILITIES

- `printf(“100% dml”);`

FORMAT STRING VULNERABILITIES

- `printf(“100% dml”);`
  - Prints stack entry 4 byes above saved %eip

FORMAT STRING VULNERABILITIES

- `printf(“100% dml”);`
  - Prints stack entry 4 byes above saved %eip

- `printf(“%s”);`

FORMAT STRING VULNERABILITIES

- `printf(“100% dml”);`
  - Prints stack entry 4 byes above saved %eip
- `printf(“%s”);`
  - Prints bytes *pointed to* by that stack entry

FORMAT STRING VULNERABILITIES

- `printf(“100% dml”);`
  - Prints stack entry 4 byes above saved %eip
- `printf(“%s”);`
  - Prints bytes *pointed to* by that stack entry
- `printf(“%d %d %d %d …”);`

FORMAT STRING VULNERABILITIES

- `printf(“100% dml”);`
    - Prints stack entry 4 byes above saved %eip
- `printf(“%s”);`
    - Prints bytes *pointed to* by that stack entry
- `printf(“%d %d %d %d …”);`
    - Prints a series of stack entries as integers

FORMAT STRING VULNERABILITIES

- `printf(“100% dml”);`
  - Prints stack entry 4 byes above saved %eip
- `printf(“%s”);`
  - Prints bytes *pointed to* by that stack entry
- `printf(“%d %d %d %d …”);`
  - Prints a series of stack entries as integers
- `printf(“%08x %08x %08x %08x …”);`

FORMAT STRING VULNERABILITIES

- `printf(“100% dml”);`
  - Prints stack entry 4 byes above saved %eip
- `printf(“%s”);`
  - Prints bytes *pointed to* by that stack entry
- `printf(“%d %d %d %d …”);`
  - Prints a series of stack entries as integers
- `printf(“%08x %08x %08x %08x …”);`
  - Same, but nicely formatted hex

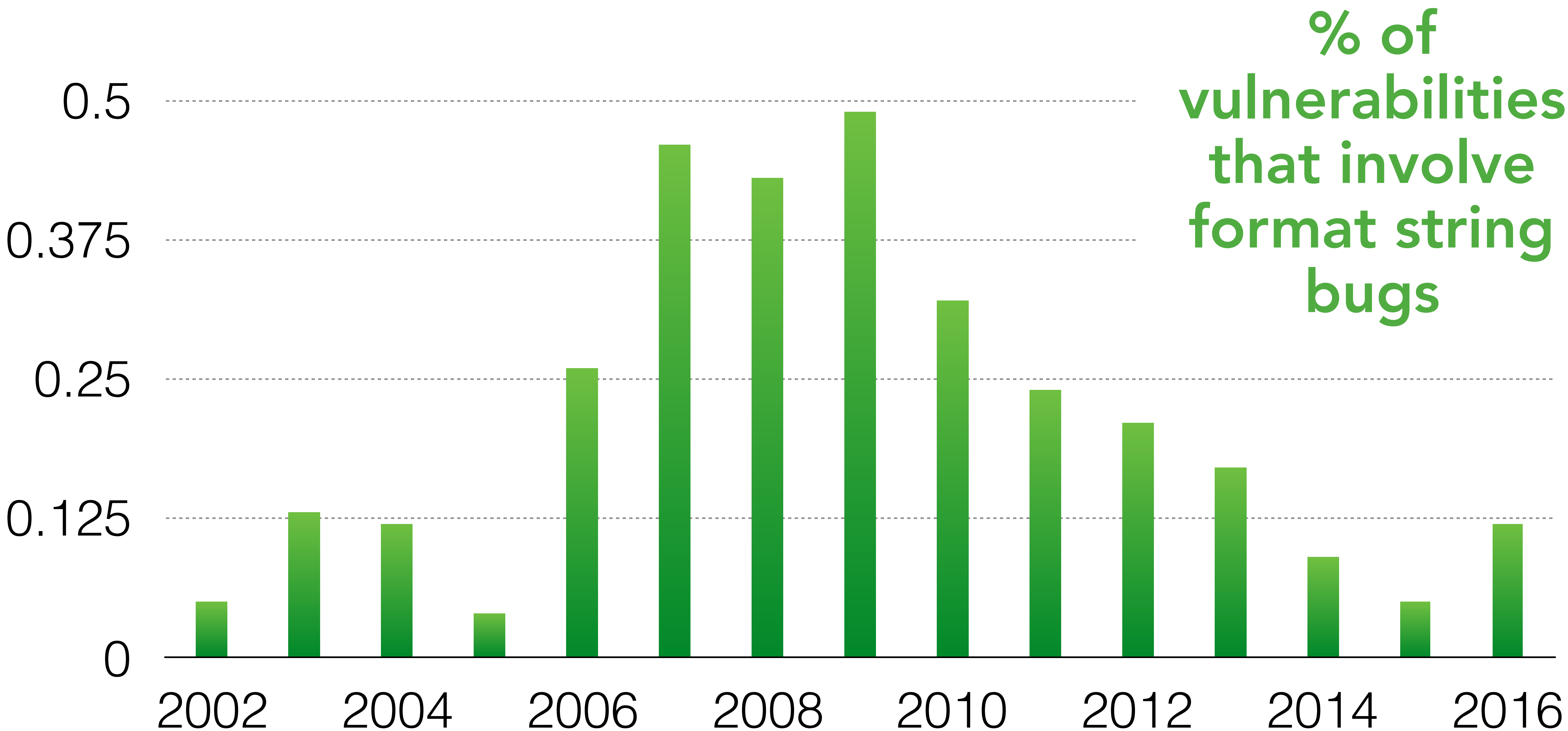
FORMAT STRING VULNERABILITIES

- `printf(“100% dml”);`
  - Prints stack entry 4 byes above saved %eip
- `printf(“%s”);`
  - Prints bytes *pointed to* by that stack entry
- `printf(“%d %d %d %d …”);`
  - Prints a series of stack entries as integers
- `printf(“%08x %08x %08x %08x …”);`
  - Same, but nicely formatted hex
- `printf(“100% no way!”)`

FORMAT STRING VULNERABILITIES

- `printf("100% dml");`
  - Prints stack entry 4 byes above saved %eip
- `printf("%s");`
  - Prints bytes *pointed to* by that stack entry
- `printf("%d %d %d %d …");`
  - Prints a series of stack entries as integers
- `printf("%08x %08x %08x %08x …");`
  - Same, but nicely formatted hex
- `printf("100% no way!")`
  - ***WRITES*** the number 3 to address pointed to by stack entry

FORMAT STRING PREVALENCE



http://web.nvd.nist.gov/view/vuln/statistics

WHAT'S WRONG WITH THIS CODE?

```
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if(len > BUF_SIZE) {
        printf("Too large\n");
        return;
    }
    memcpy(buf, p, len);
}
```

WHAT'S WRONG WITH THIS CODE?

```
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if(len > BUF_SIZE) {
        printf("Too large\n");
        return;
    }
    memcpy(buf, p, len);
}
```

```
void *memcpy(void *dest, const void *src, size_t n);
```

WHAT'S WRONG WITH THIS CODE?

```
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if(len > BUF_SIZE) {
        printf("Too large\n");
        return;
    }
    memcpy(buf, p, len);
}
```

```
void *memcpy(void *dest, const void *src, size_t n);
```

```
typedef unsigned int size_t;
```

WHAT'S WRONG WITH THIS CODE?

```
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{   Negative
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if(len > BUF_SIZE) {
        printf("Too large\n");
        return;
    }
    memcpy(buf, p, len);
}
```

```
void *memcpy(void *dest, const void *src, size_t n);
typedef unsigned int size_t;
```

WHAT'S WRONG WITH THIS CODE?

```
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{   Negative
    int len = read_int_from_network();
    char *p = read_string_from_network();
 Ok if(len > BUF_SIZE) {
        printf(“Too large\n”);
        return;
    }
    memcpy(buf, p, len);
}
```

```
void *memcpy(void *dest, const void *src, size_t n);
```

```
typedef unsigned int size_t;
```

WHAT'S WRONG WITH THIS CODE?

```
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if(len > BUF_SIZE) {
        printf(“Too large\n”);
        return;
    }
    memcpy(buf, p, len);
}
```

Negative

Ok

Implicit cast to unsigned

```
void *memcpy(void *dest, const void *src, size_t n);
typedef unsigned int size_t;
```

INTEGER OVERFLOW VULNERABILITIES

WHAT'S WRONG WITH THIS CODE?

```
void vulnerable()
{
    size_t len;
    char *buf;

    len = read_int_from_network();
    buf = malloc(len + 5);
    read(fd, buf, len);
    ...
}
```

WHAT'S WRONG WITH THIS CODE?

```
void vulnerable()
{
    size_t len;
    char *buf;
```

HUGE

```
    len = read_int_from_network();
    buf = malloc(len + 5);
    read(fd, buf, len);
    ...
}
```

WHAT'S WRONG WITH THIS CODE?

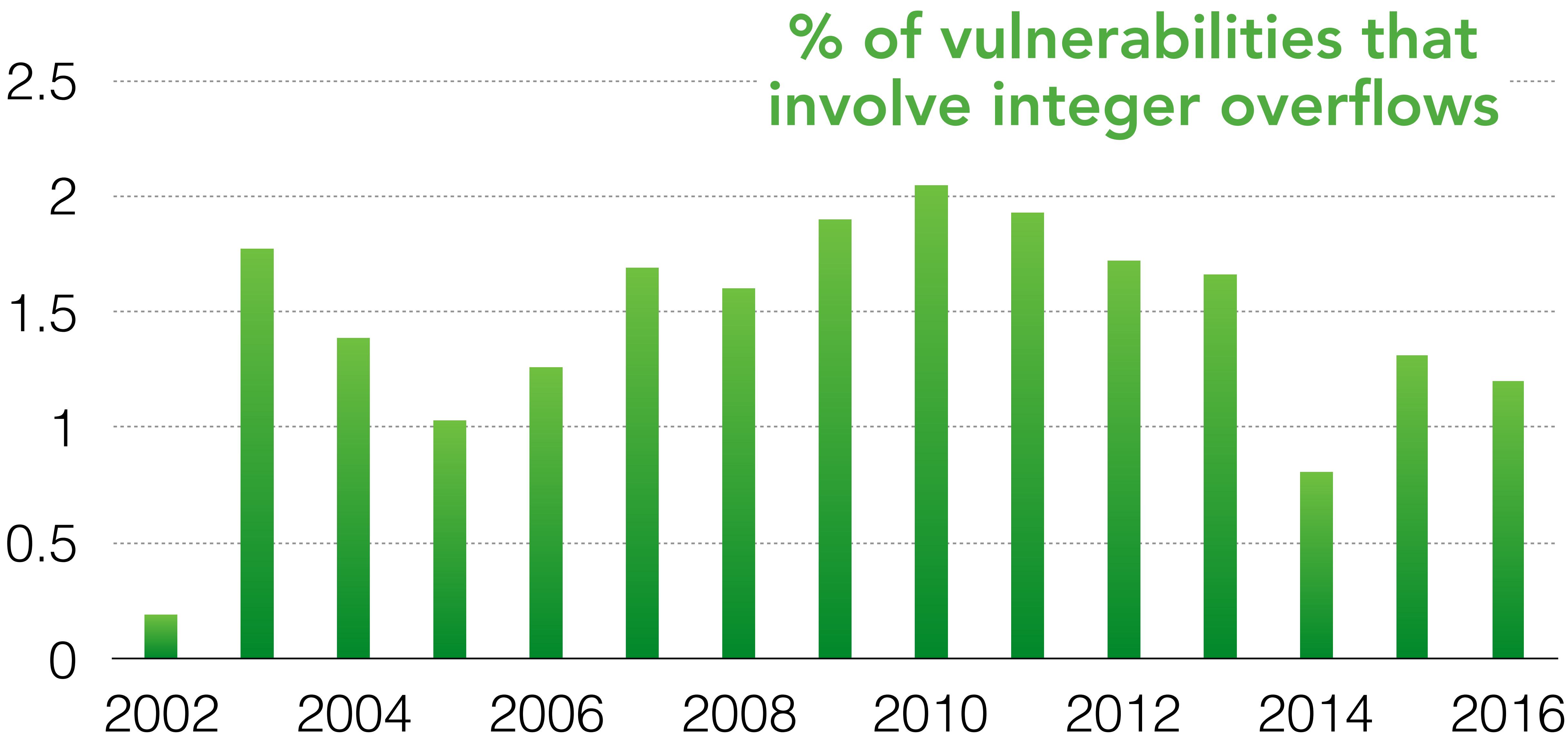
```
void vulnerable()
{
    size_t len;
    char *buf;
  HUGE
    len = read_int_from_network();
    buf = malloc(len + 5); Wrap-around
    read(fd, buf, len);
    ...
}
```

WHAT'S WRONG WITH THIS CODE?

```
void vulnerable()
{
    size_t len;
    char *buf;
  HUGE
    len = read_int_from_network();
    buf = malloc(len + 5); Wrap-around
    read(fd, buf, len);
    ...
}
```

Takeaway: You have to know the semantics of your programming language to avoid these errors

INTEGER OVERFLOW PREVALENCE



http://web.nvd.nist.gov/view/vuln/statistics