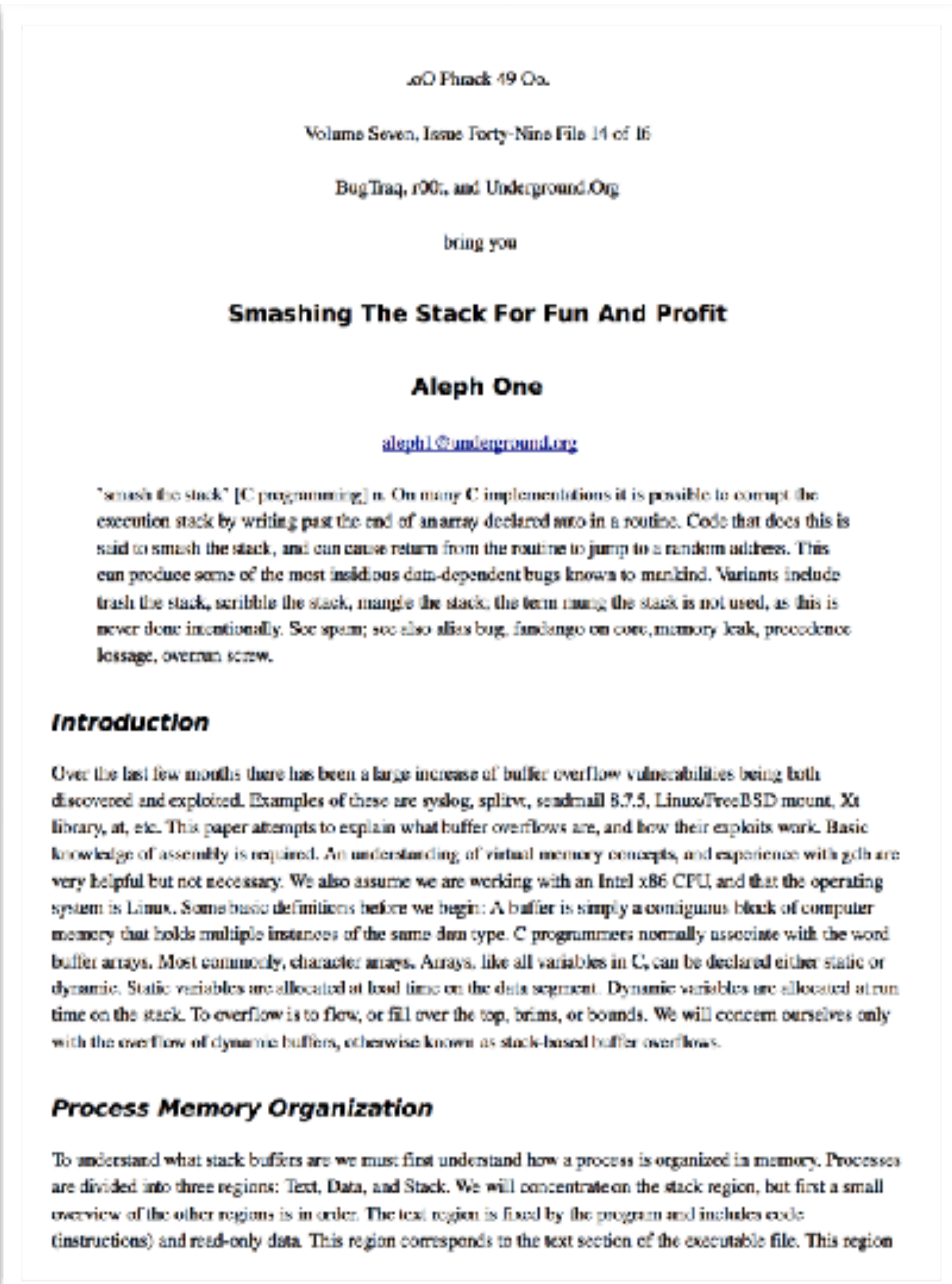
CLASSIC
MEMORY ATKS & DEFS

CMSC 414

JAN 30 2018

TODAY'S RESOURCES



.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine File 14 of 16

BugTraq, r00t, and Underground.Org

bring you

Smashing The Stack For Fun And Profit

Aleph One

aleph1@underground.org

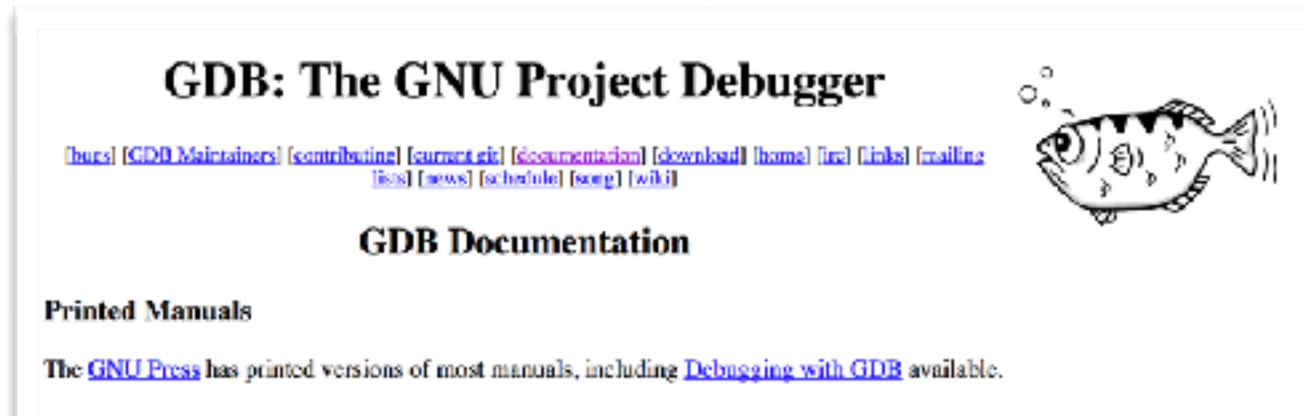
'smash the stack' [C programming] n. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce 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

To understand what stack buffers are we must first understand how a process is organized in memory. Processes are divided into three regions: Text, Data, and Stack. We will concentrate on the stack region, but first a small overview of the other regions is in order. The text region is fixed by the program and includes code (instructions) and read-only data. This region corresponds to the text section of the executable file. This region



GDB: The GNU Project Debugger

[bugs] [GDB Maintainers] [contributing] [current git] [documentation] [download] [home] [irc] [links] [mailing lists] [news] [schedule] [song] [wiki]

GDB Documentation

Printed Manuals

The GNU Press has printed versions of most manuals, including Debugging with GDB available.

Online GDB manuals

Document

GDB Use

Des

Tra

GDB Inte

Tec

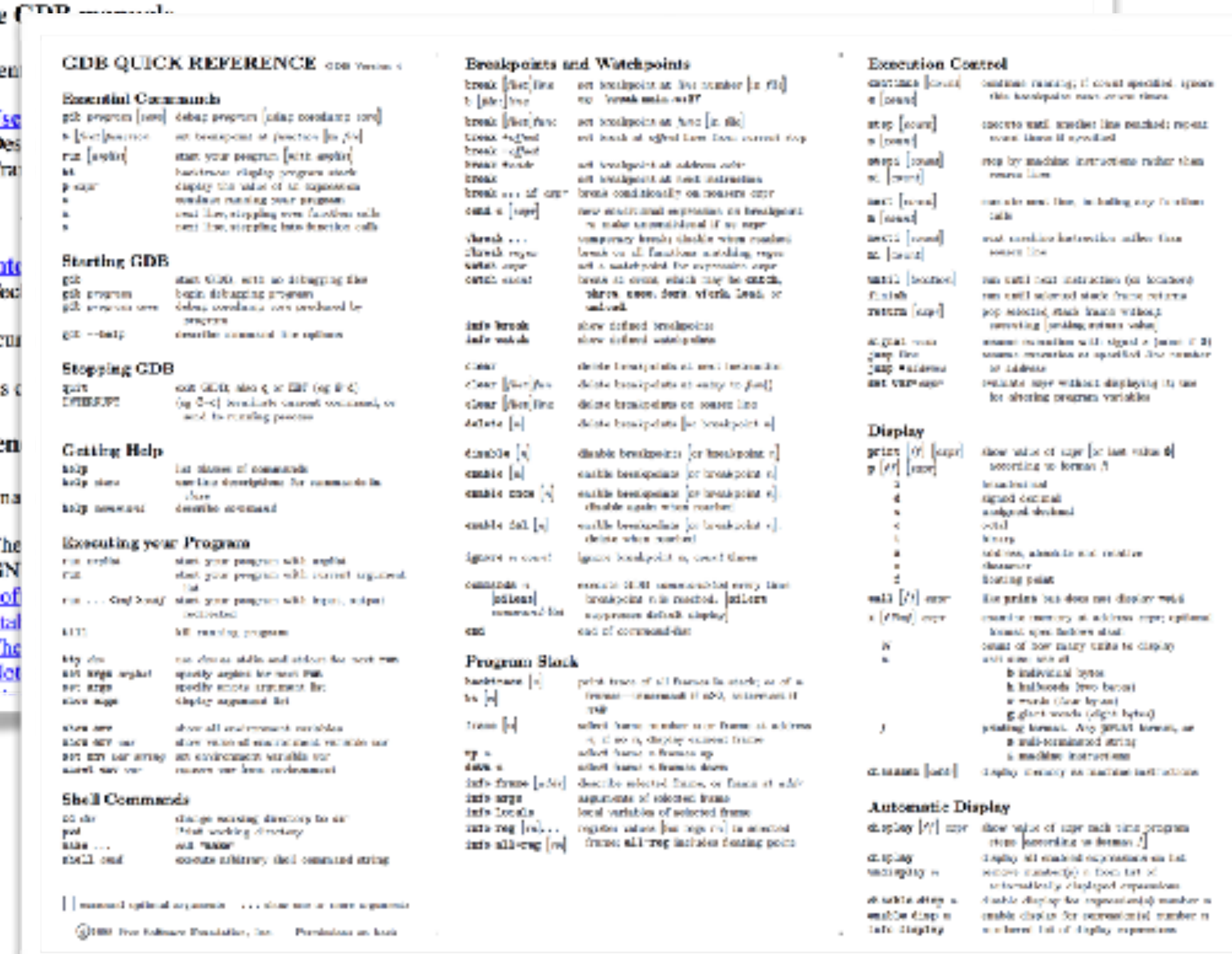
The docu

Versions

Referen

Additiona

- The GN Sof
- Sta
- The
- Not



GDB QUICK REFERENCE

Essential Commands

Starting GDB

Stopping GDB

Getting Help

Executing your Program

Shell Commands

Breakpoints and Watchpoints

Program Stack

Execution Control

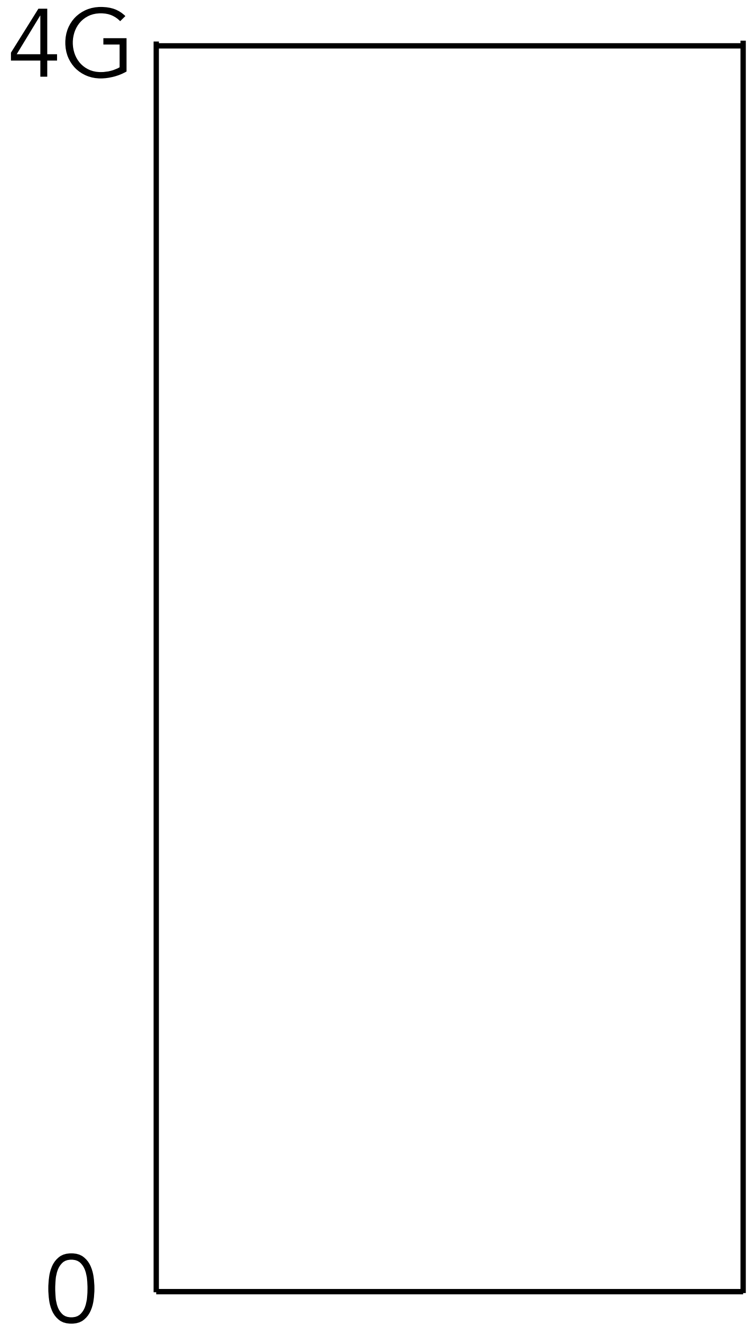
Display

Automatic Display

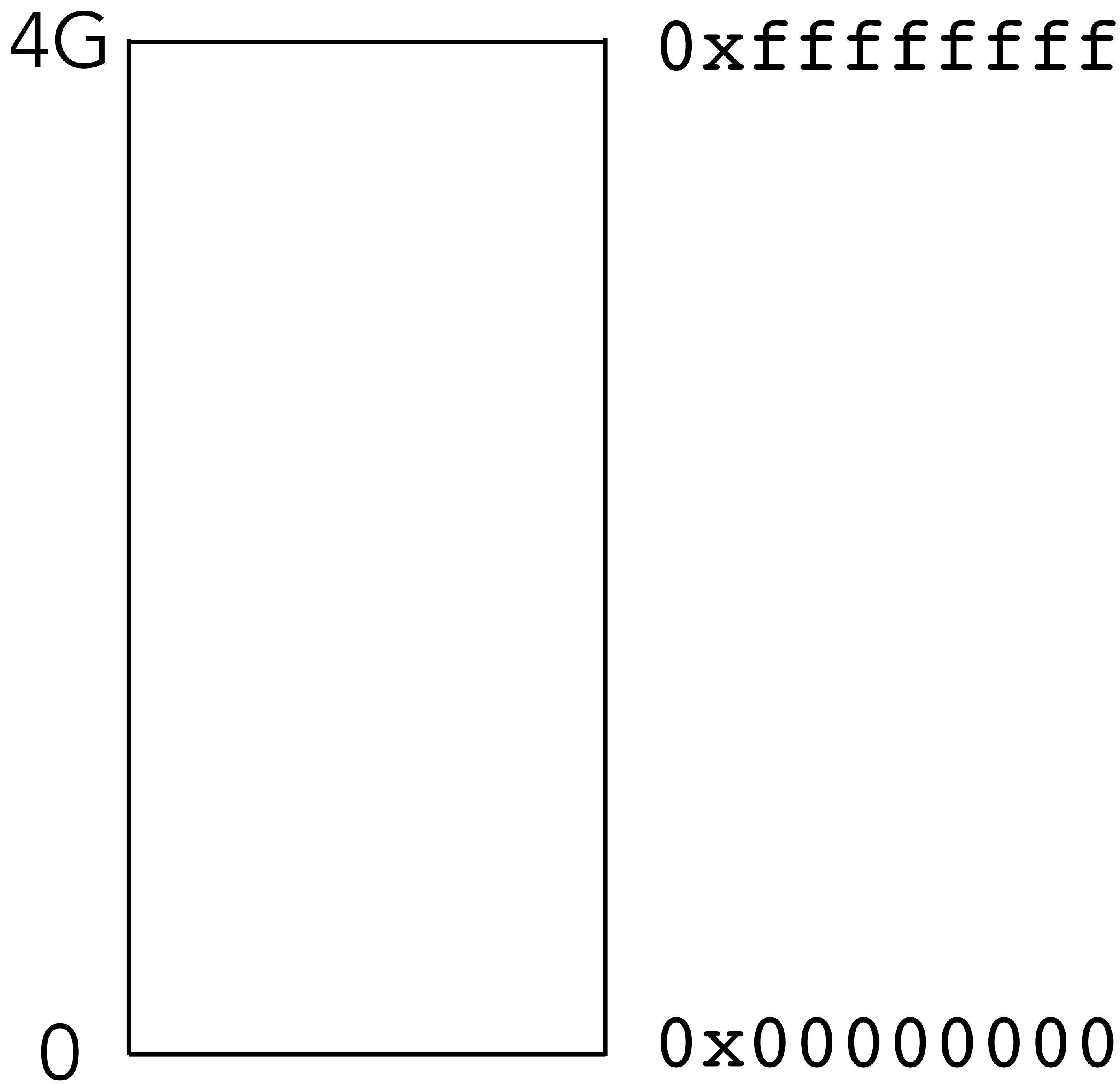
REFRESHER

- 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

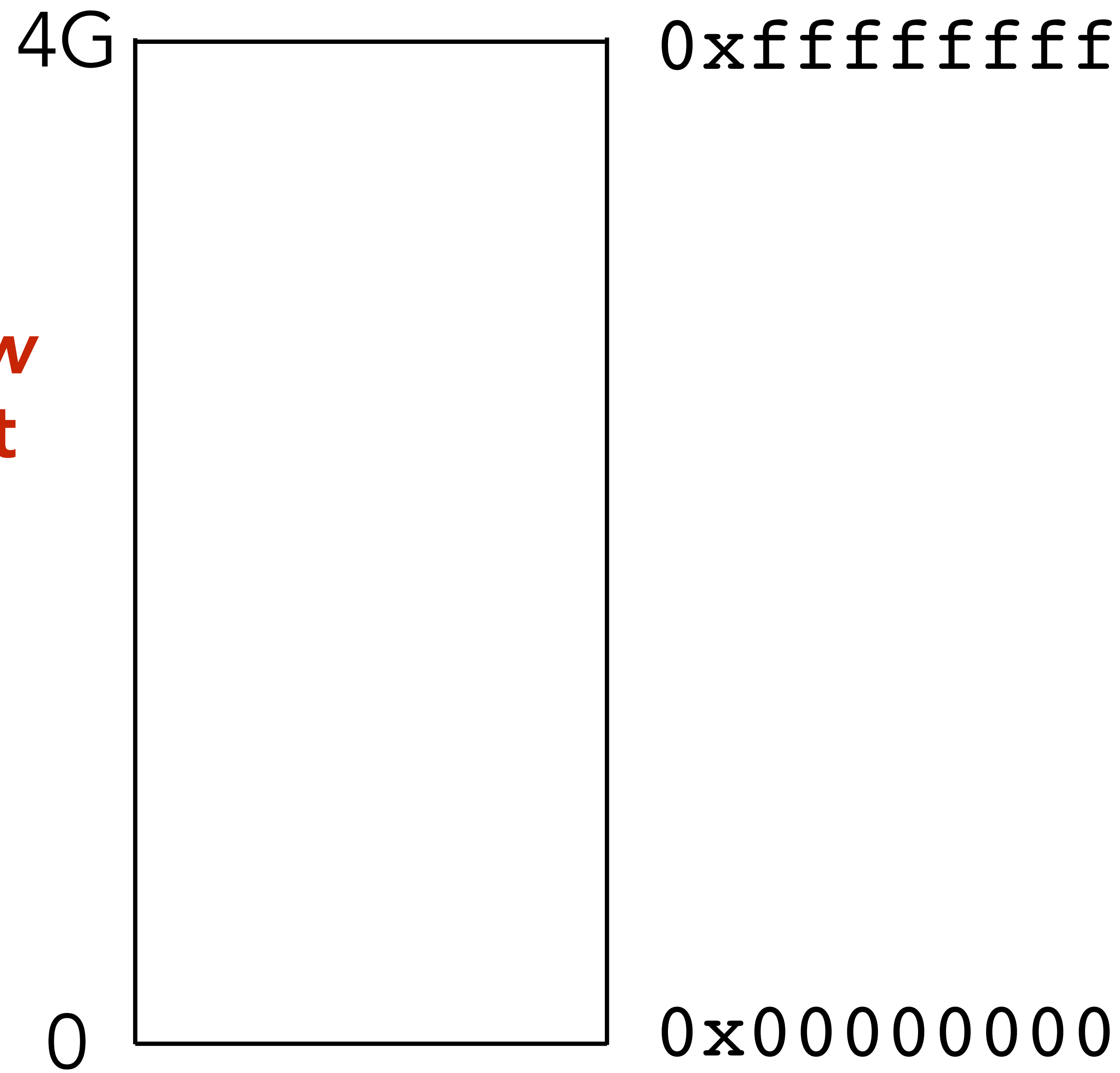
ALL PROGRAMS ARE STORED IN MEMORY



ALL PROGRAMS ARE STORED IN MEMORY



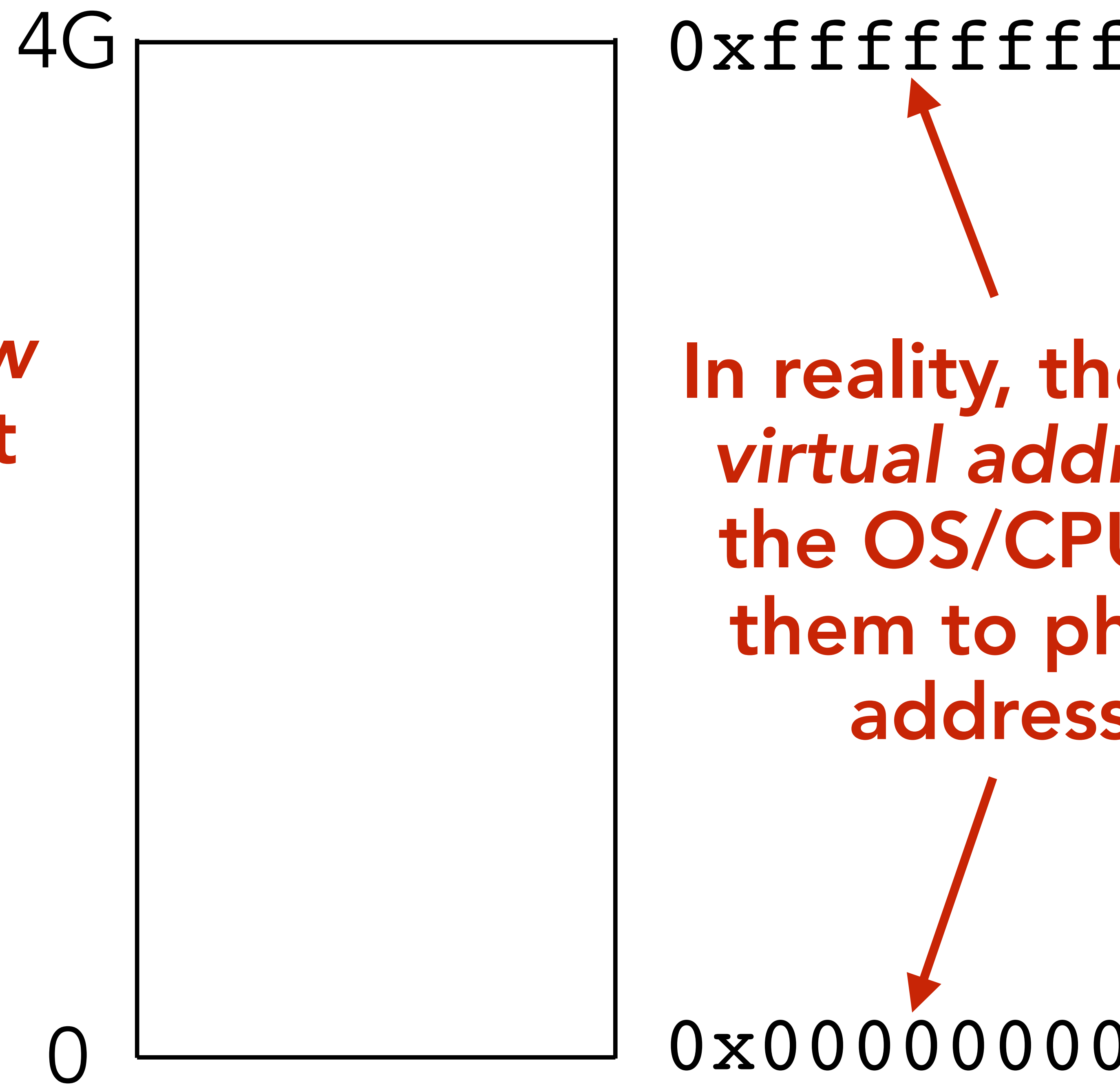
ALL PROGRAMS ARE STORED IN MEMORY



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

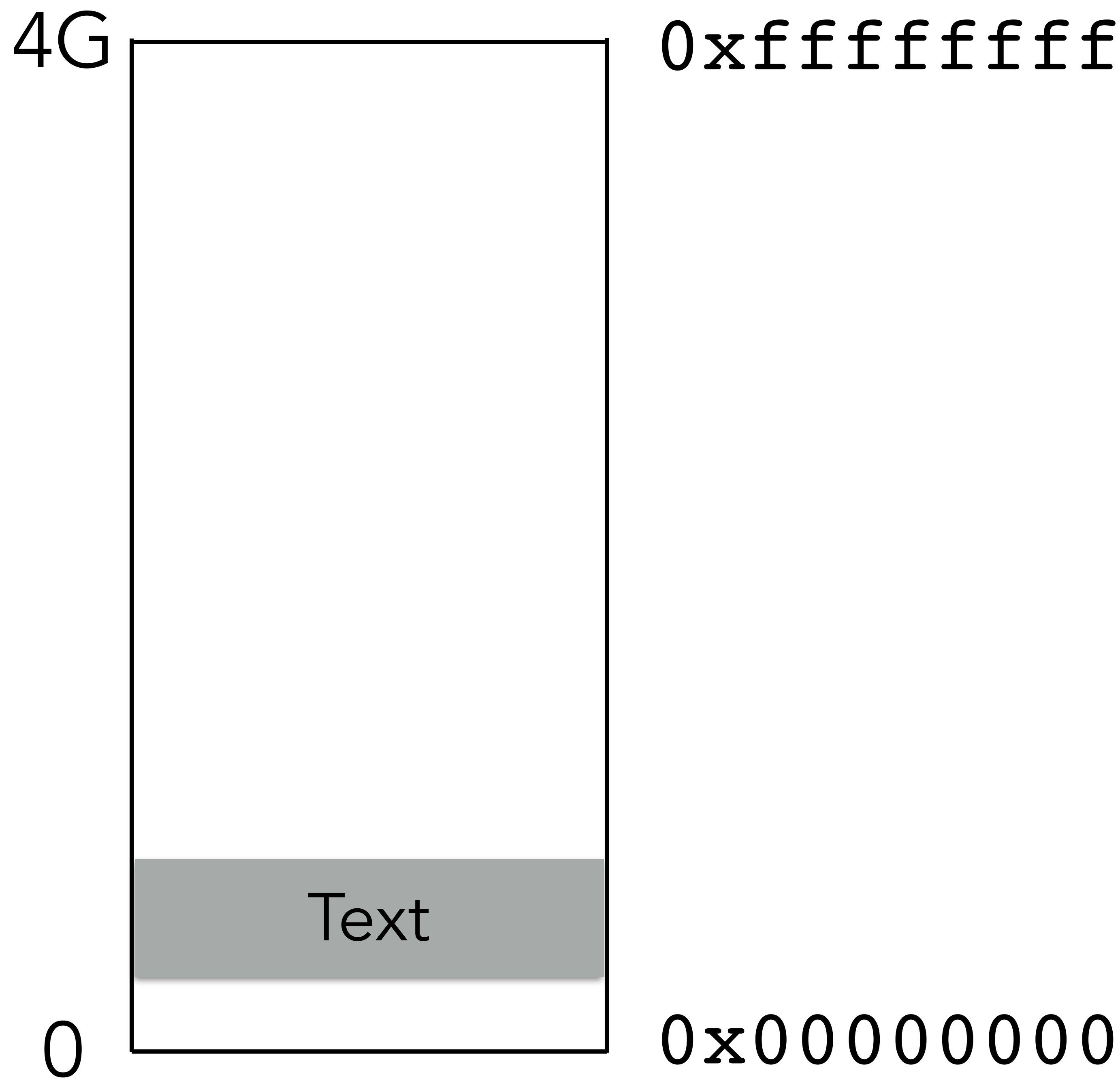
ALL PROGRAMS ARE STORED IN MEMORY

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

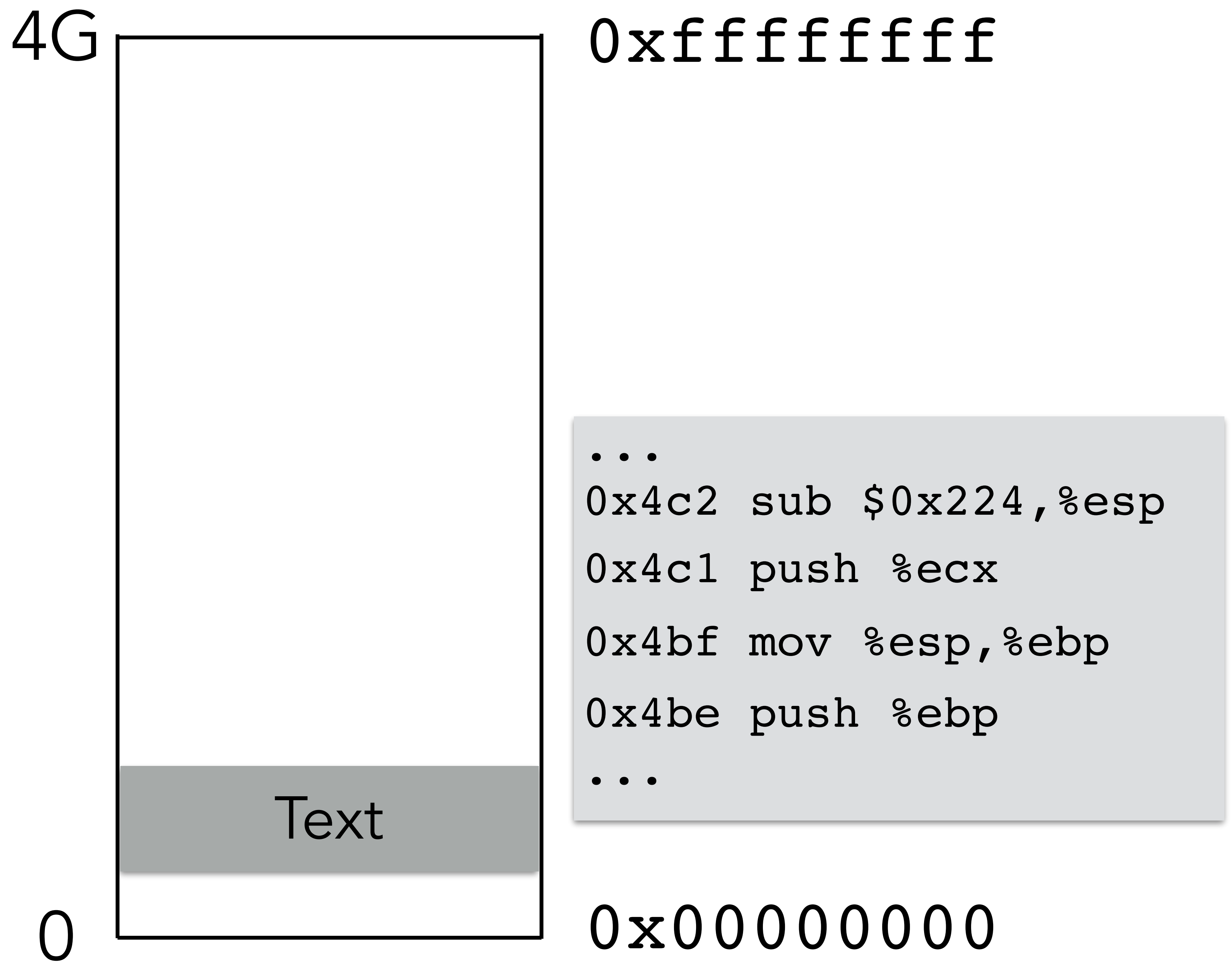


In reality, these are *virtual addresses*; the OS/CPU map them to physical addresses

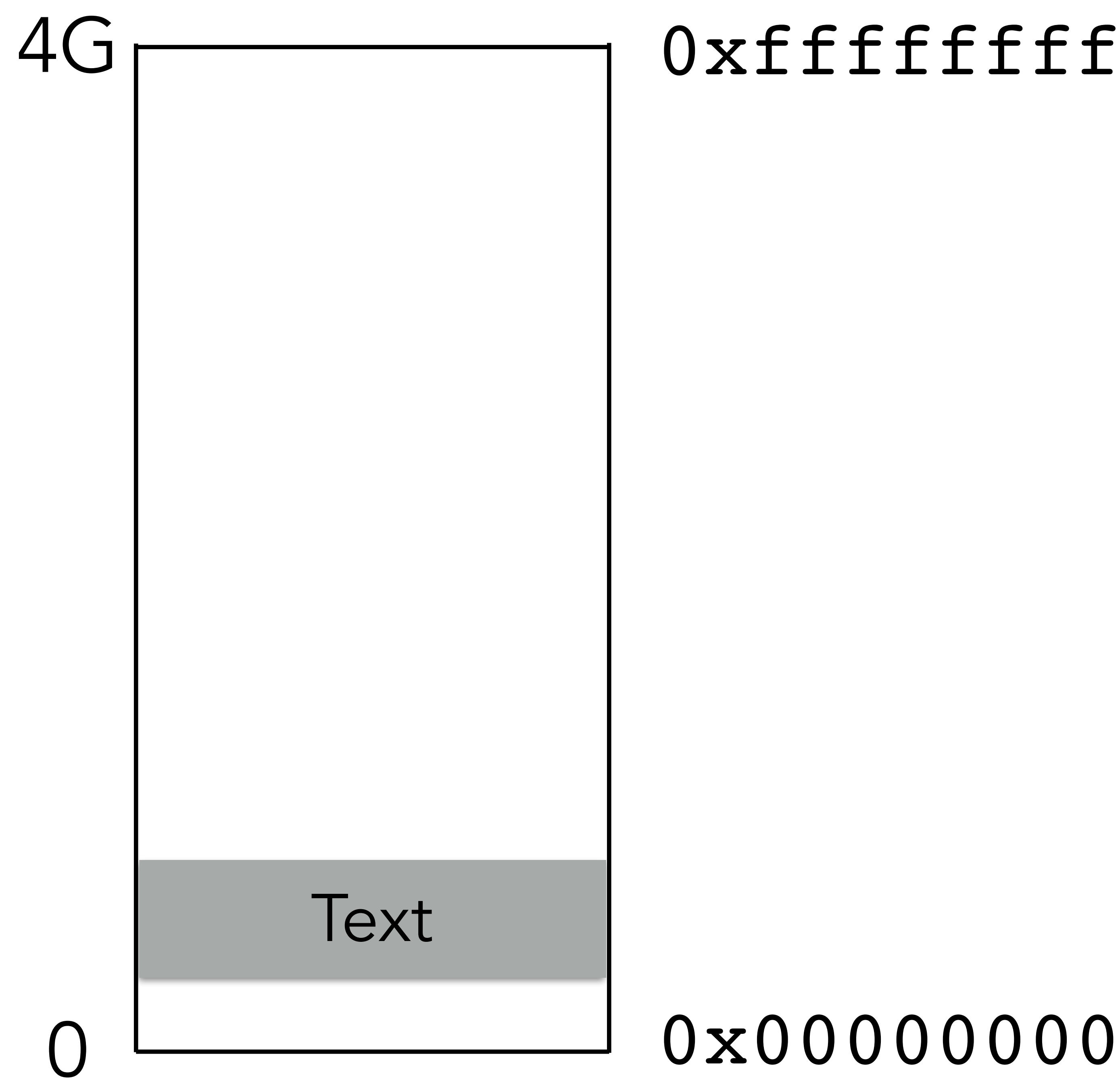
THE INSTRUCTIONS THEMSELVES ARE STORED IN MEMORY



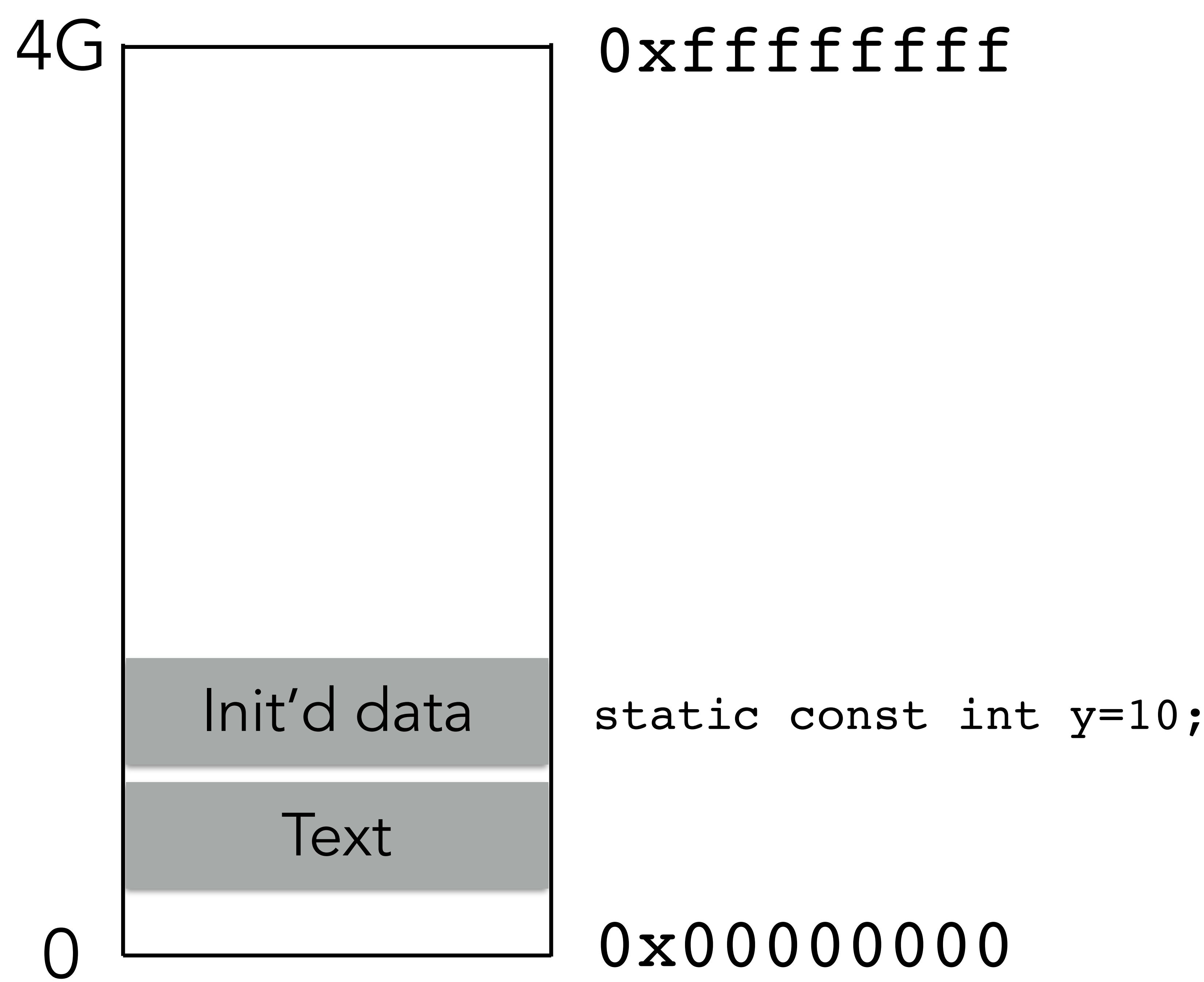
THE INSTRUCTIONS THEMSELVES ARE STORED IN MEMORY



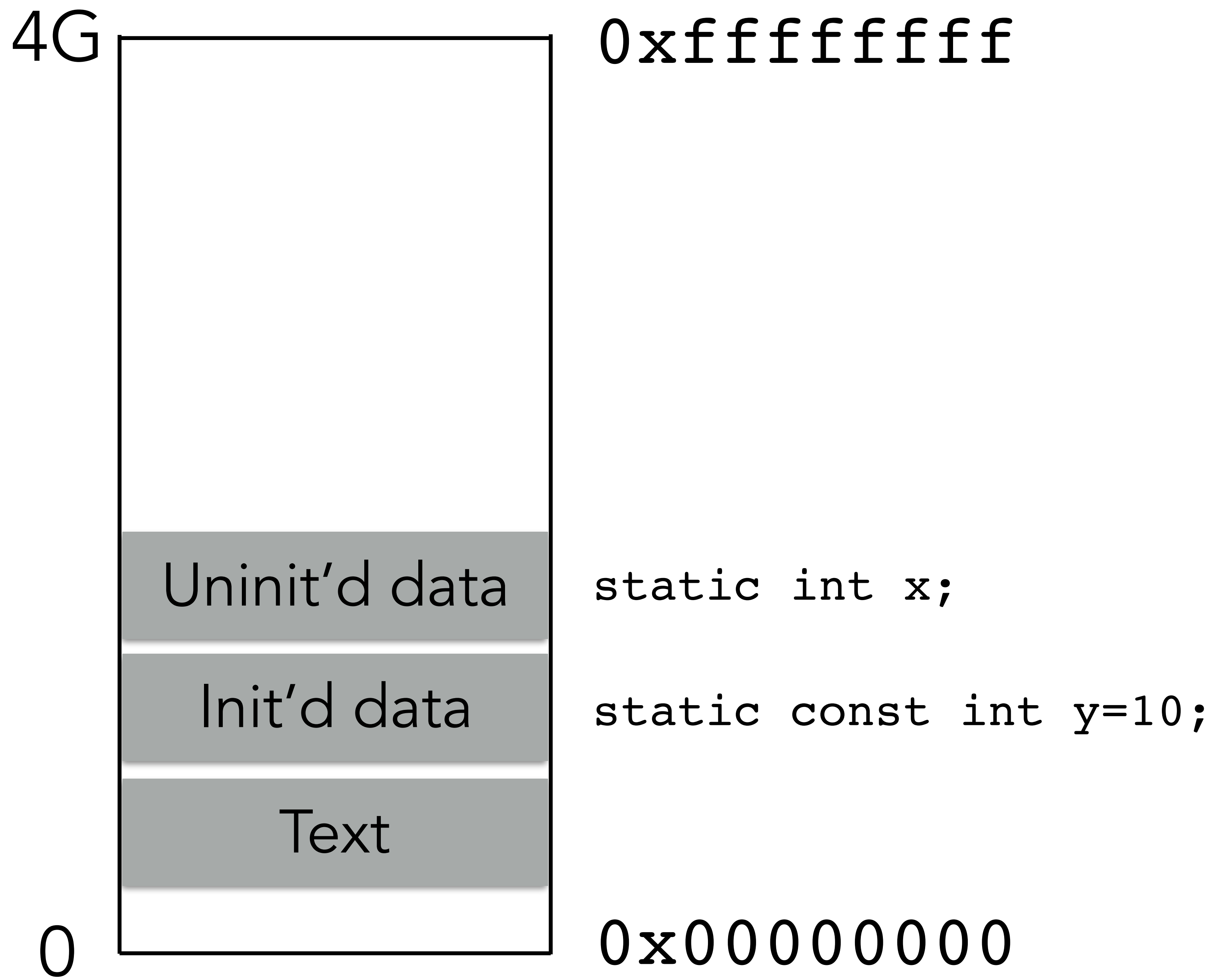
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



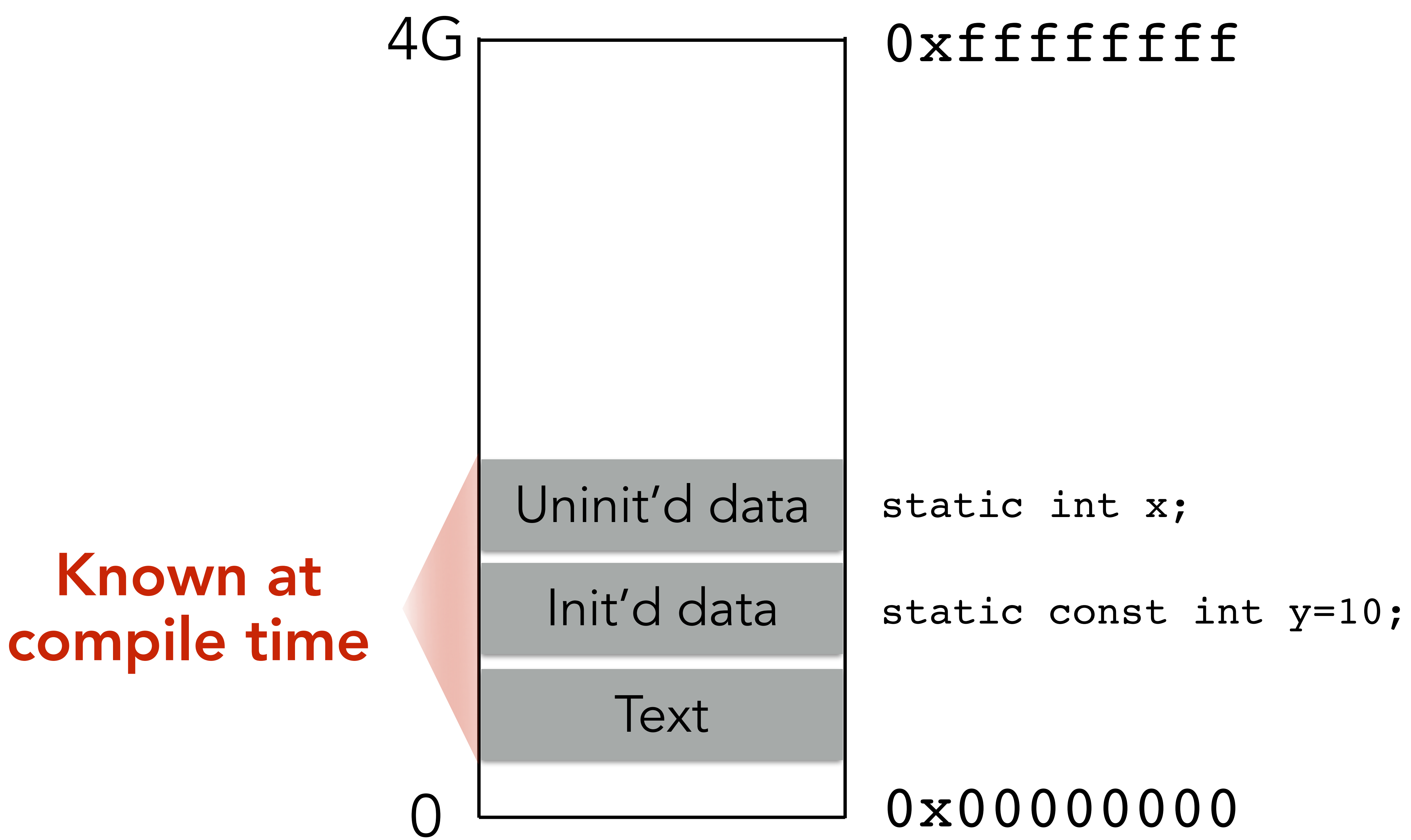
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



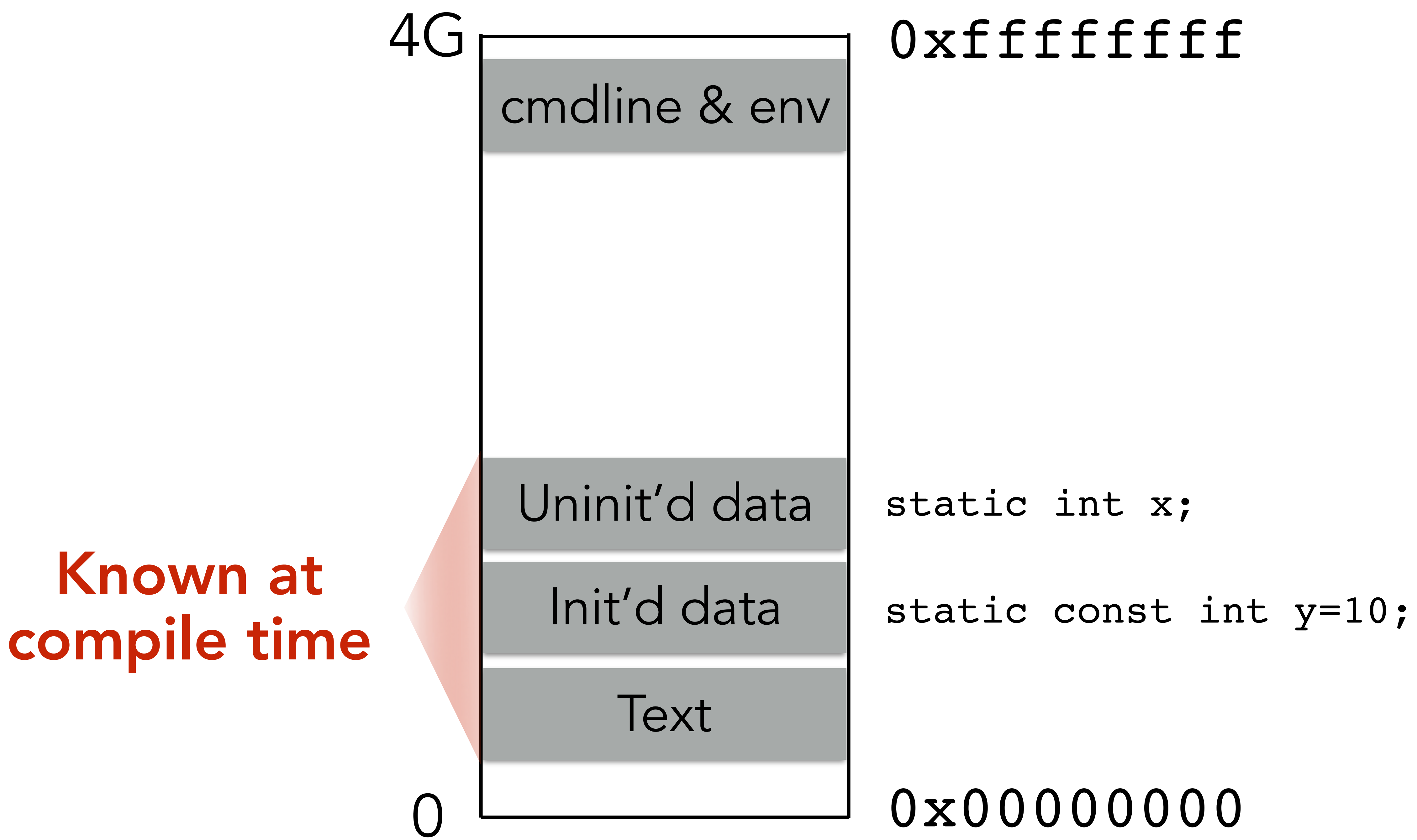
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



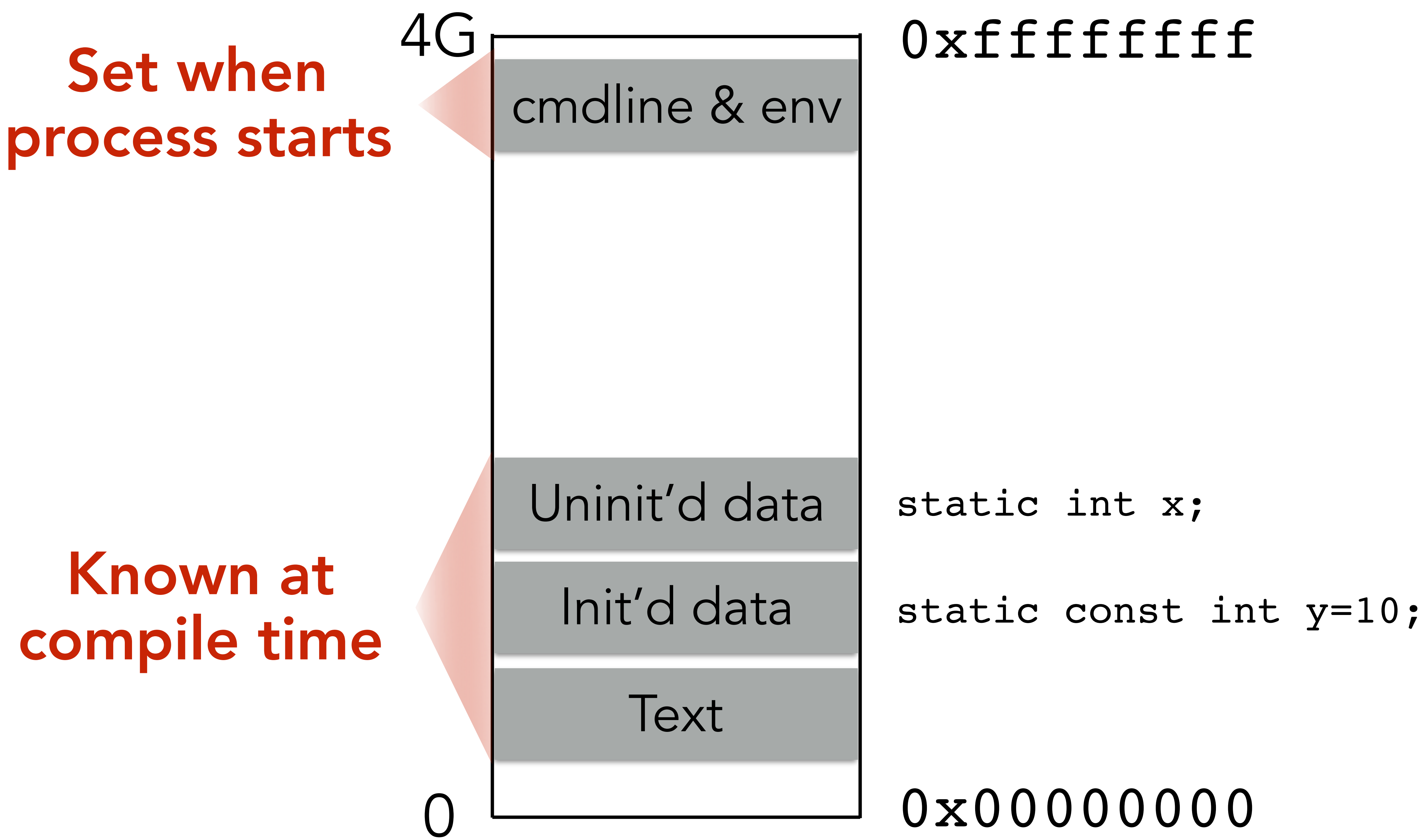
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



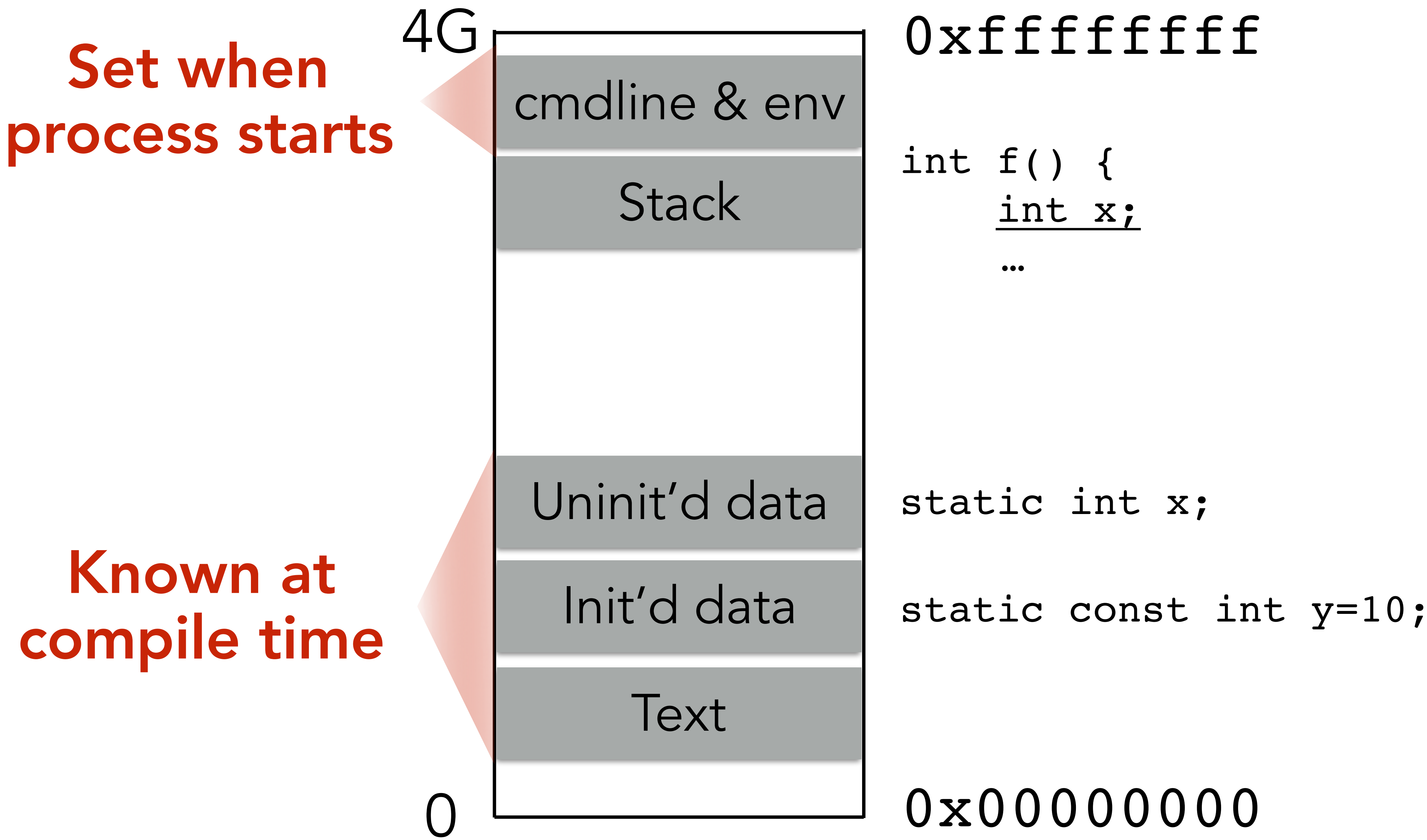
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



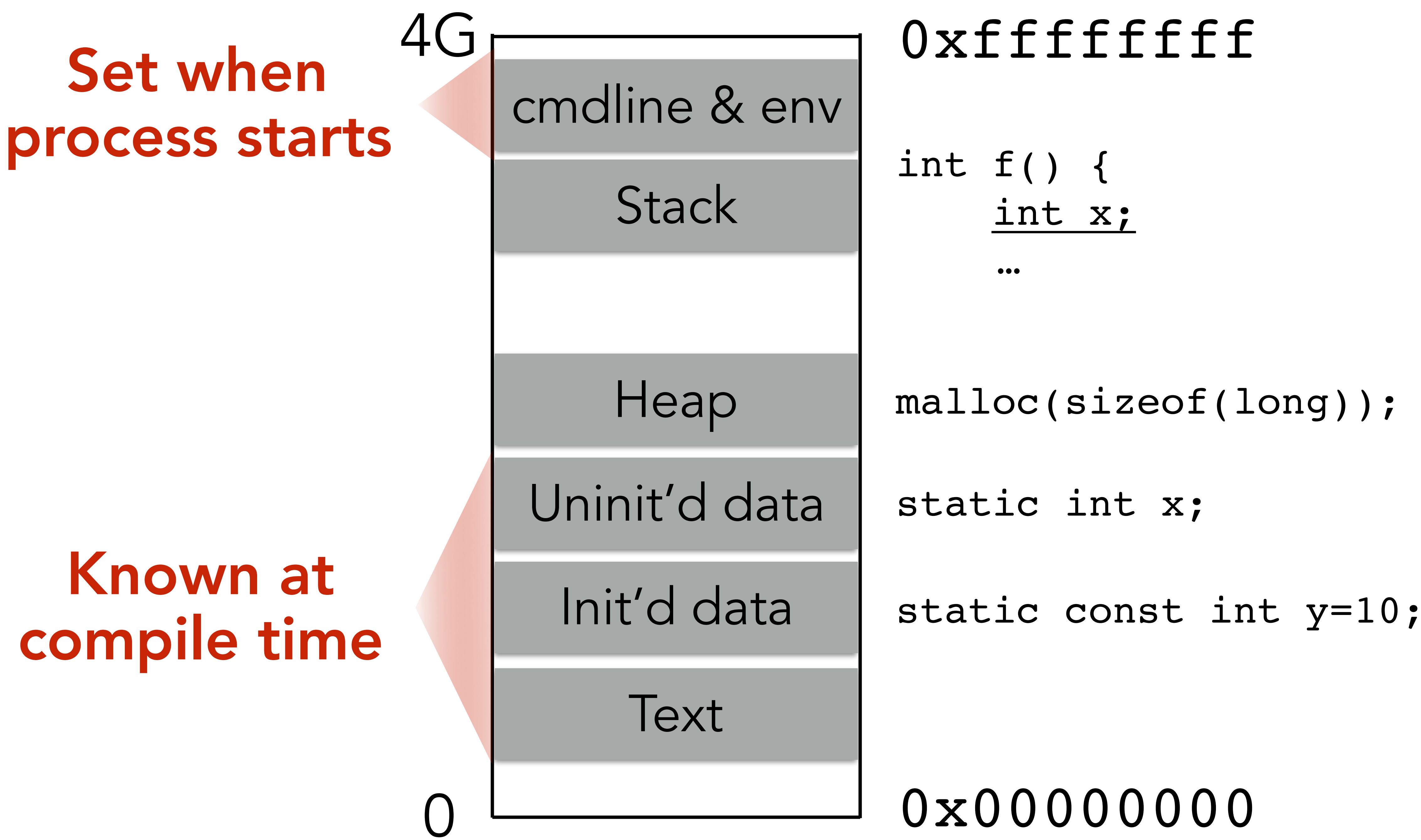
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



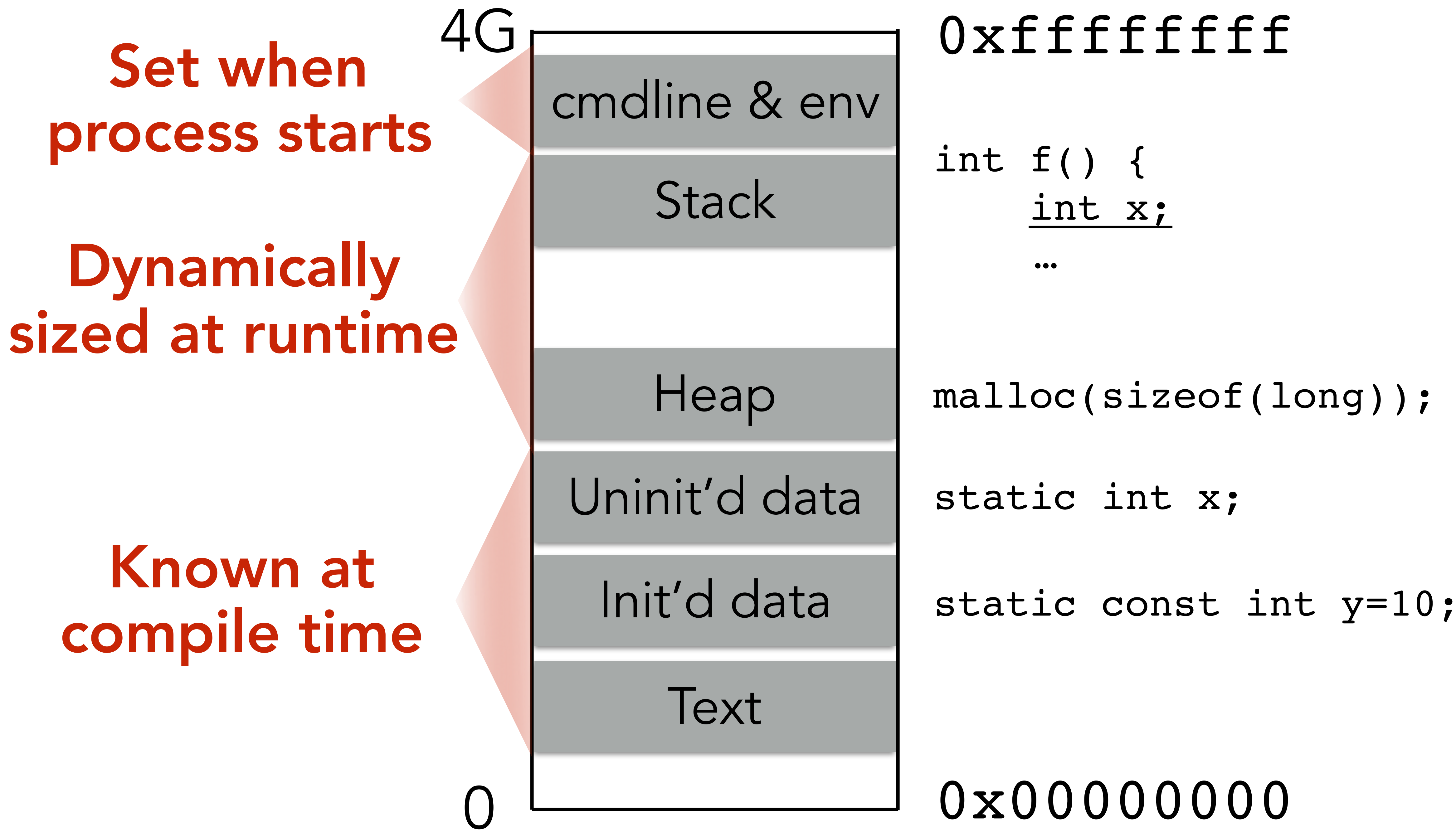
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



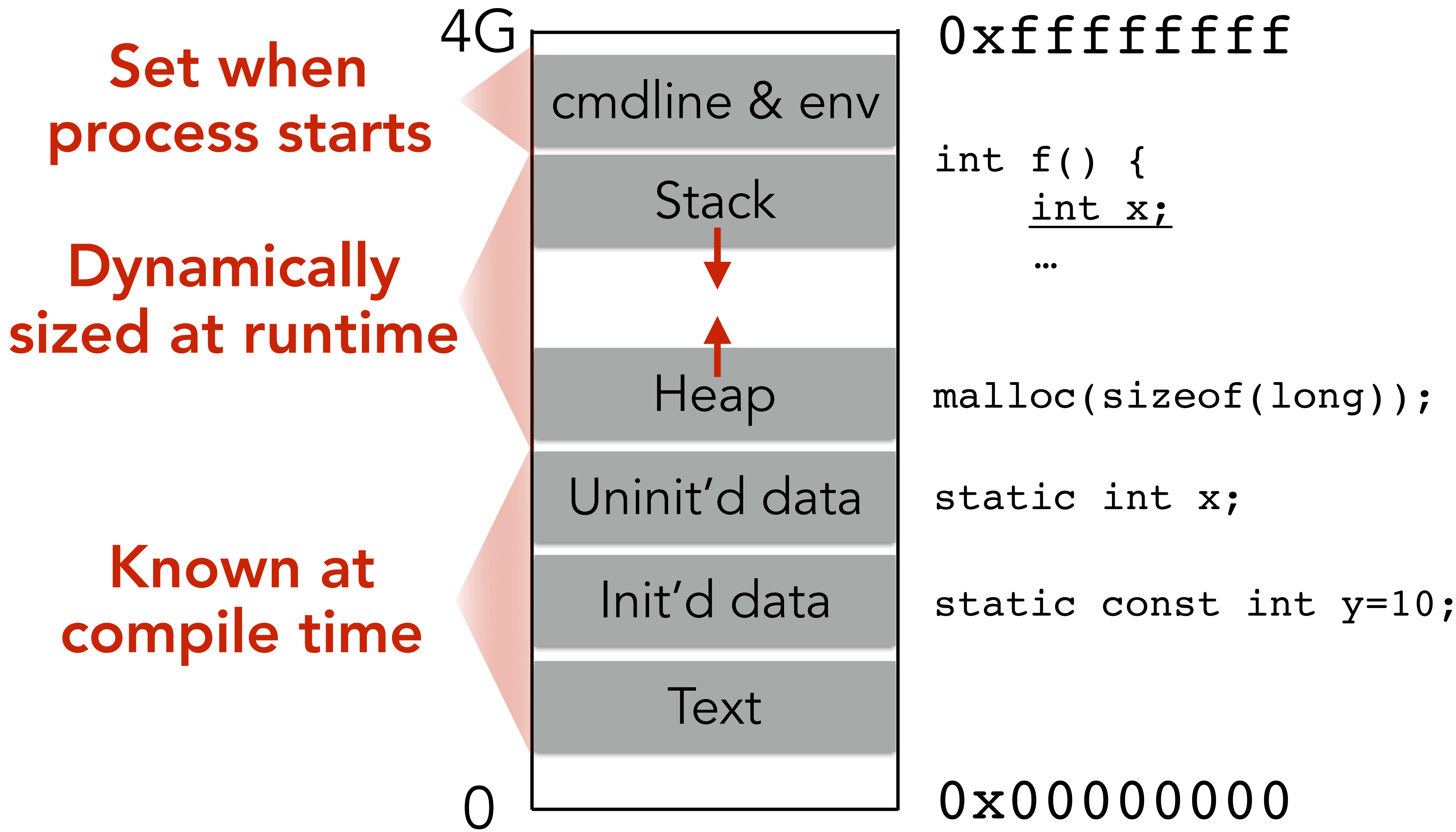
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



DATA'S LOCATION DEPENDS ON HOW IT'S CREATED

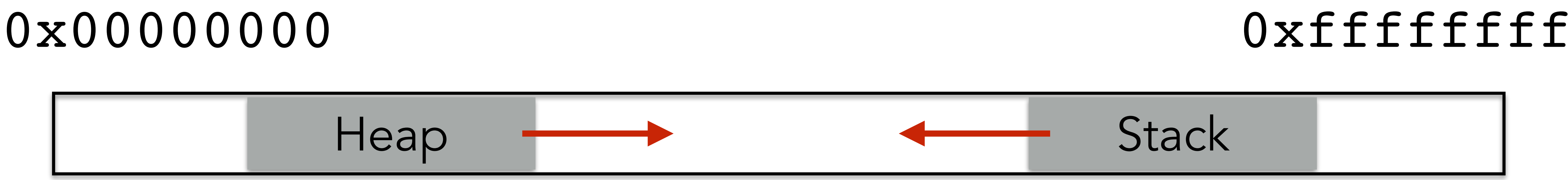


DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

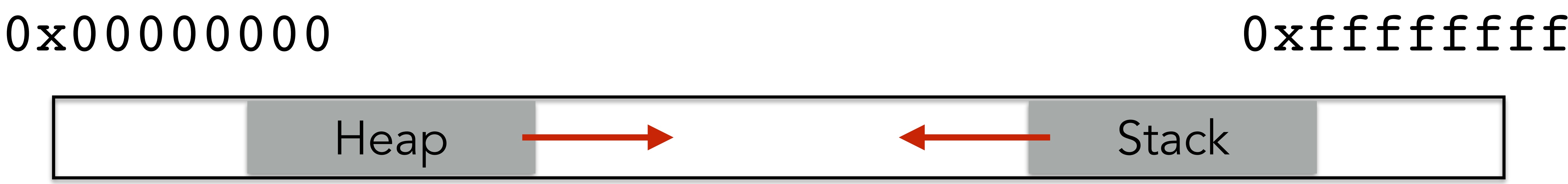
Stack and heap grow in opposite directions



WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

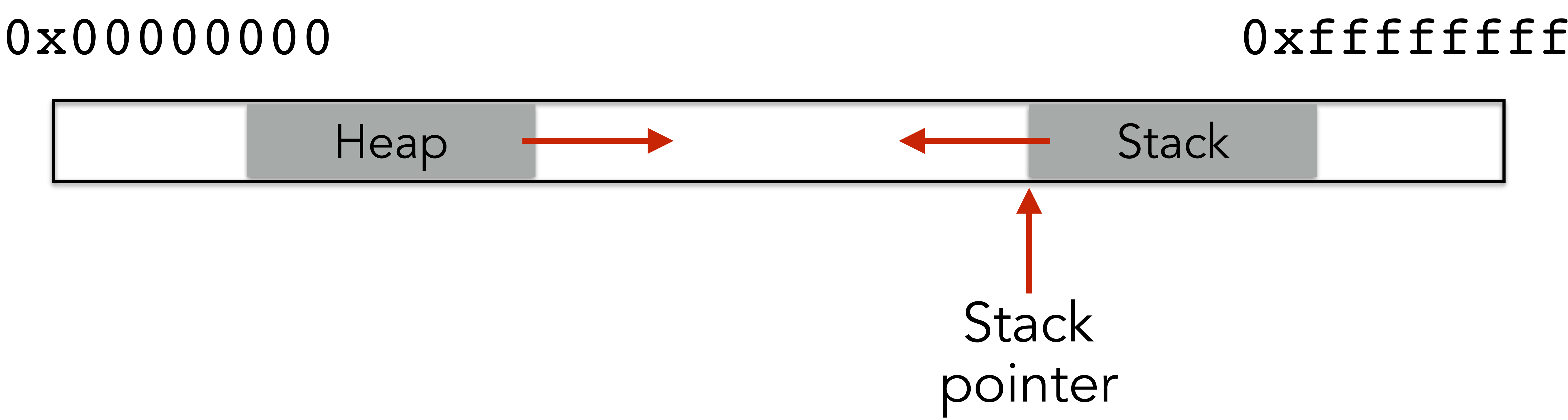
Compiler provides instructions that adjusts the size of the stack at runtime



WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

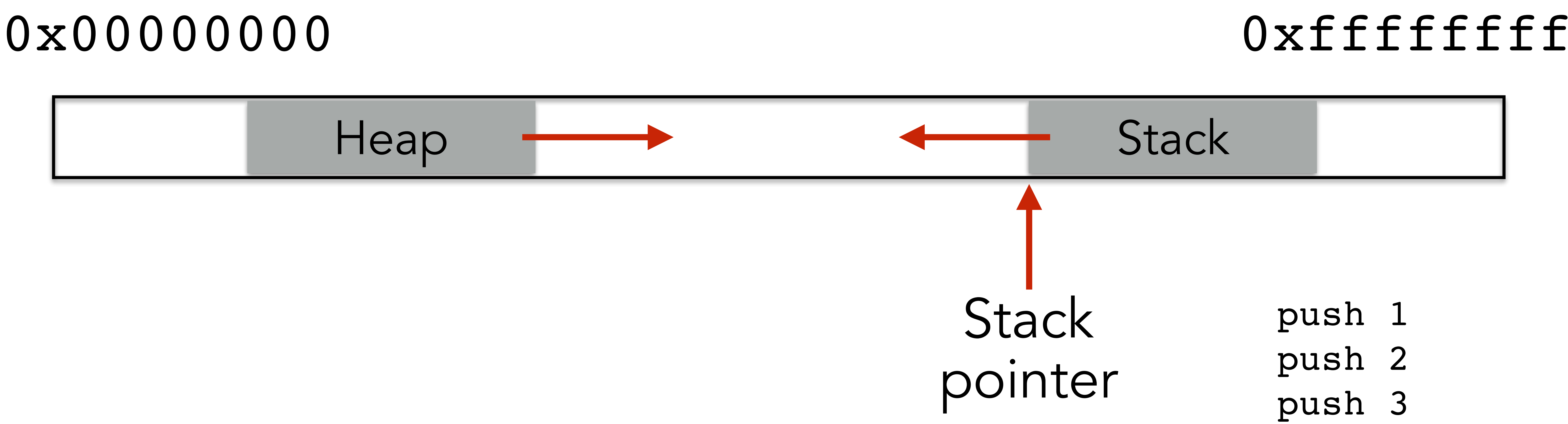
Compiler provides instructions that adjusts the size of the stack at runtime



WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

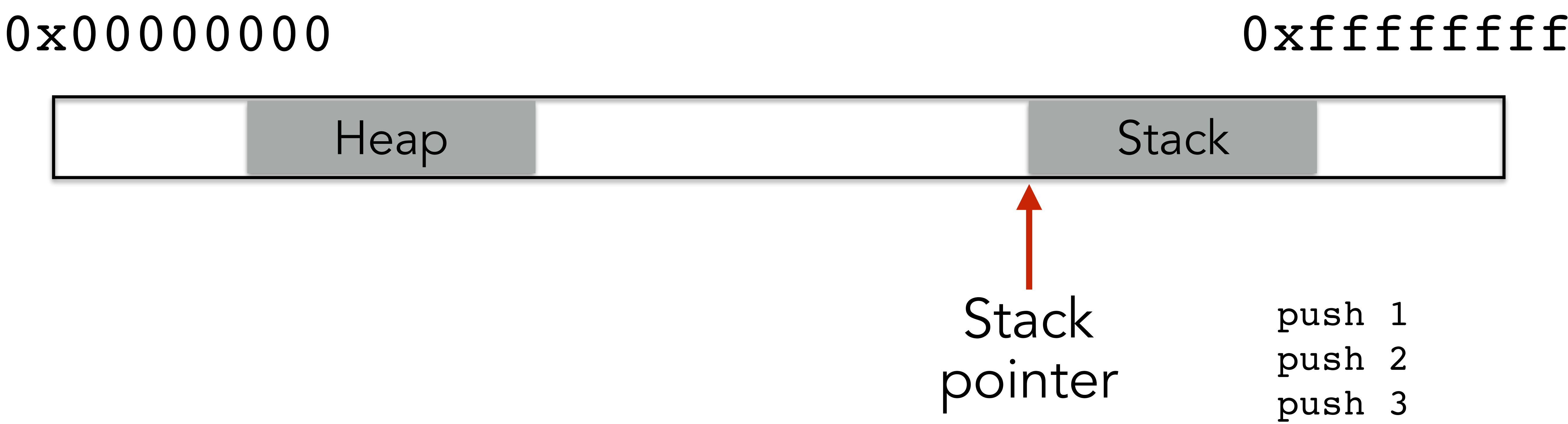
Compiler provides instructions that adjusts the size of the stack at runtime



WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

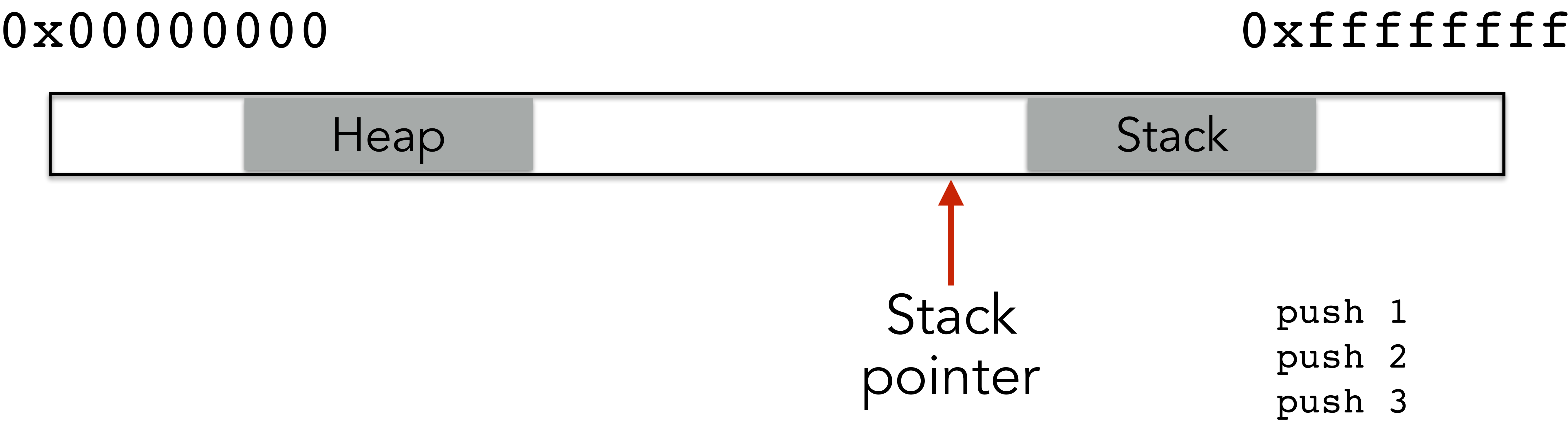
Compiler provides instructions that adjusts the size of the stack at runtime



WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

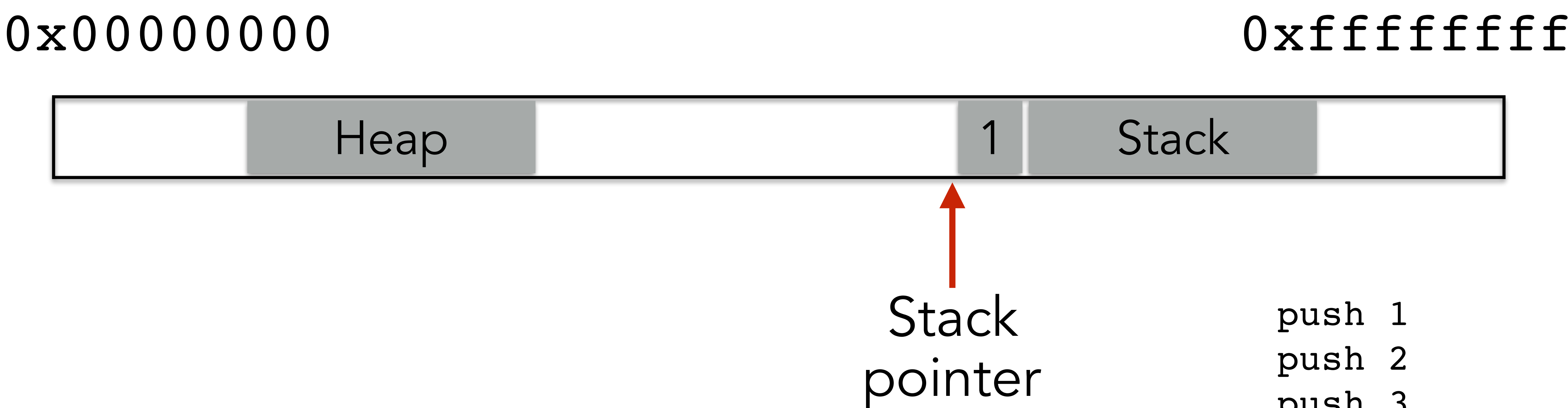
Compiler provides instructions that adjusts the size of the stack at runtime



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

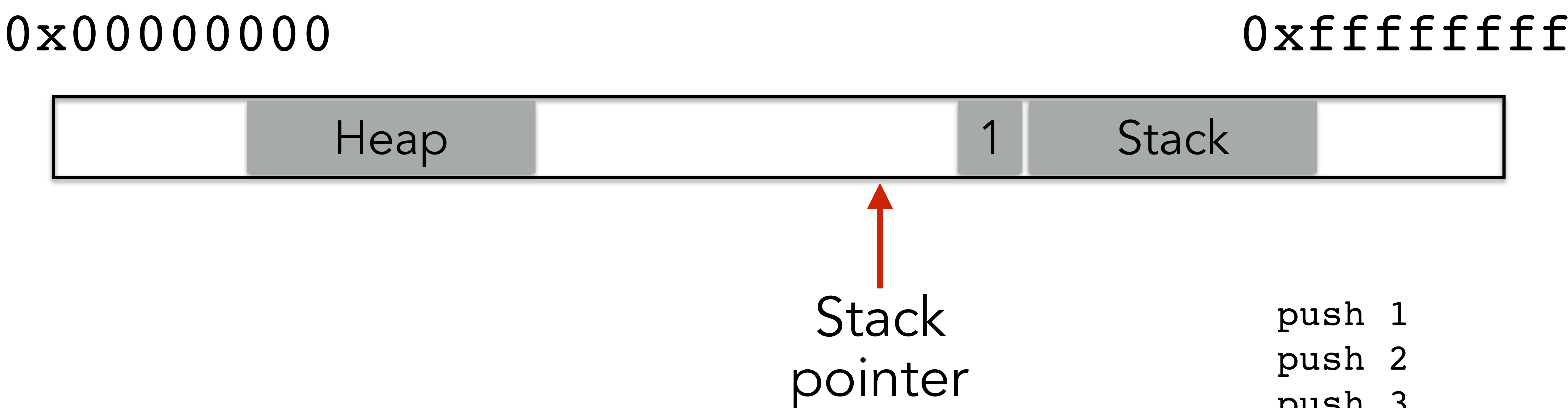



```
push 1
push 2
push 3
```

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

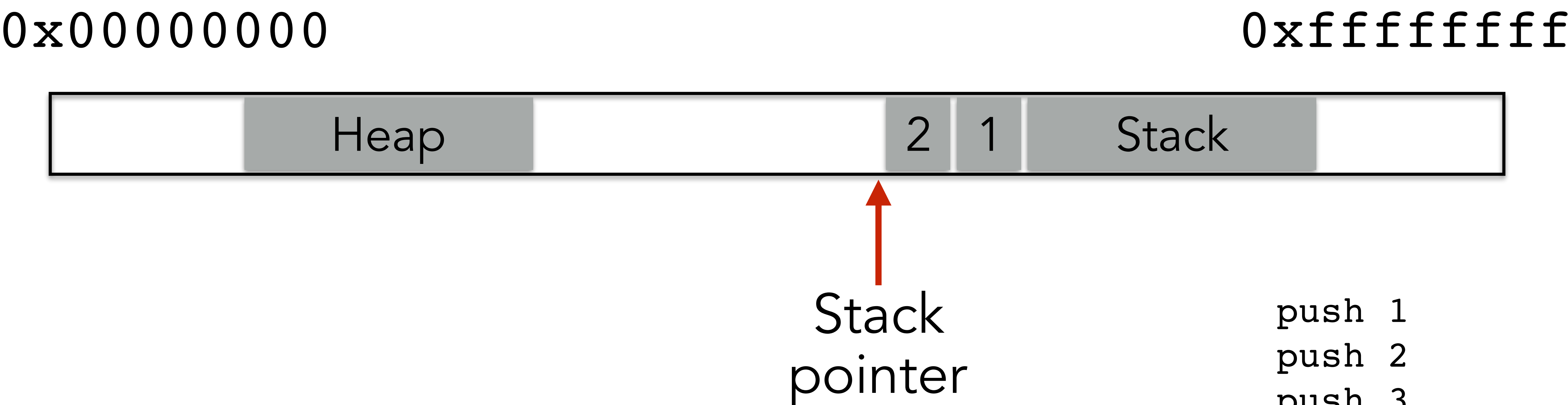



```
push 1
push 2
push 3
```

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

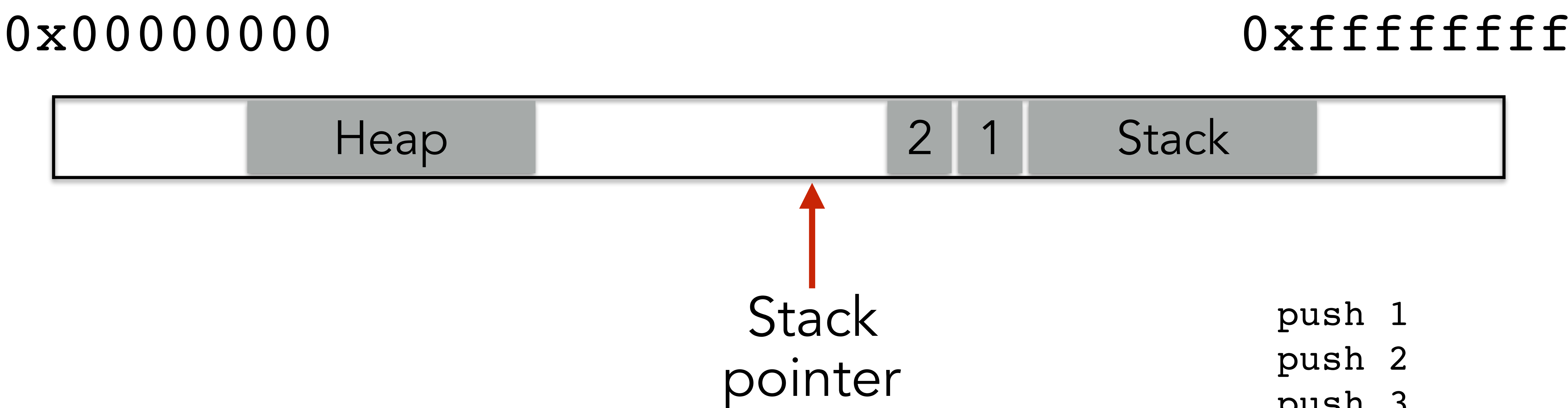



```
push 1
push 2
push 3
```

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

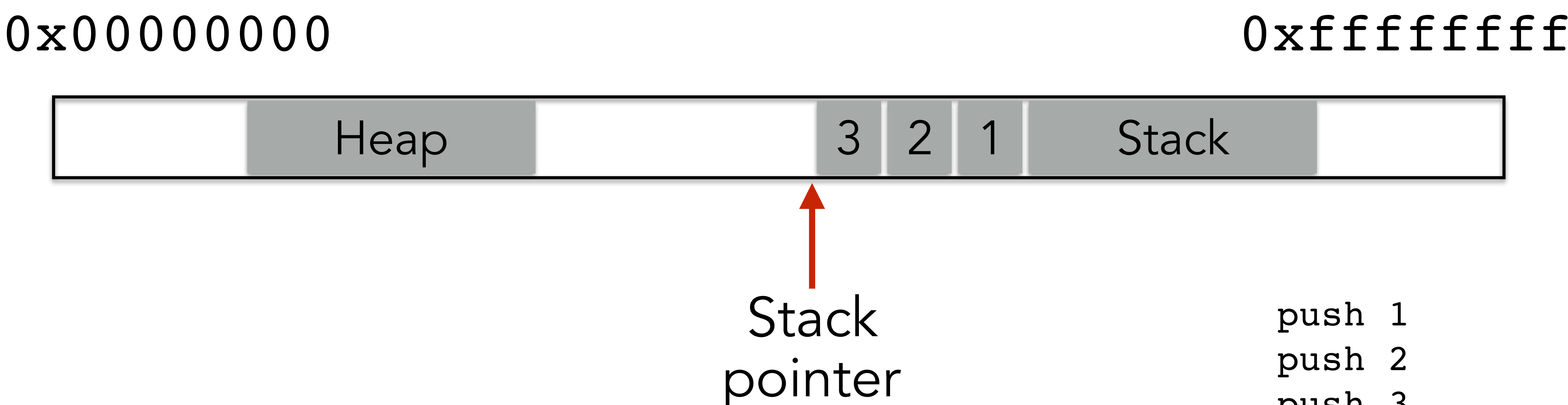



```
push 1
push 2
push 3
```

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

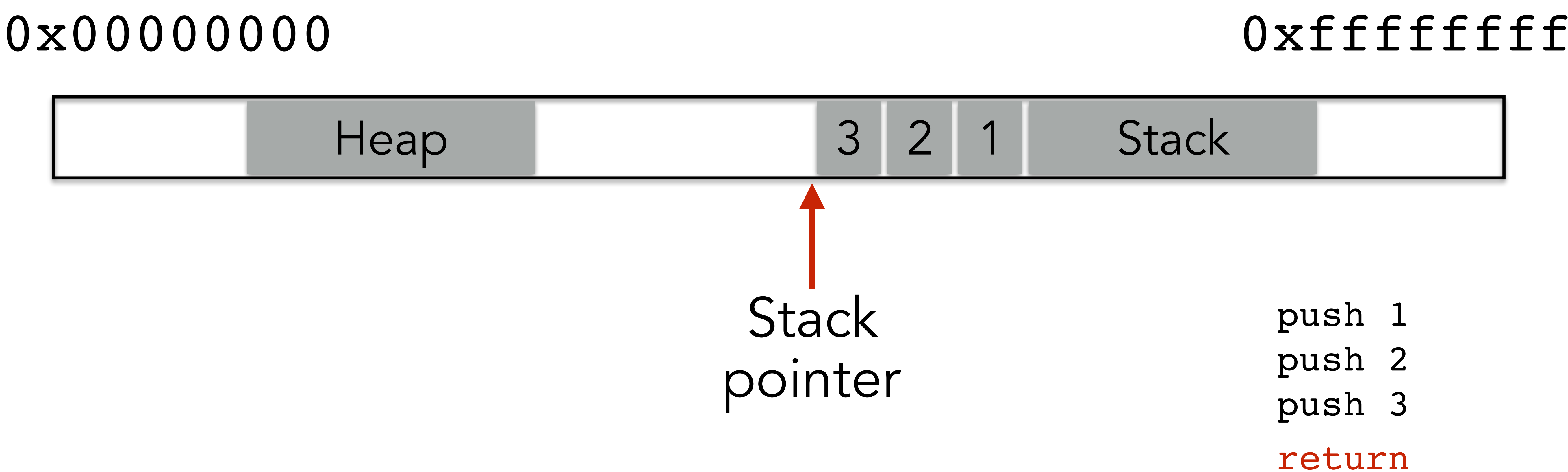



```
push 1
push 2
push 3
```

WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

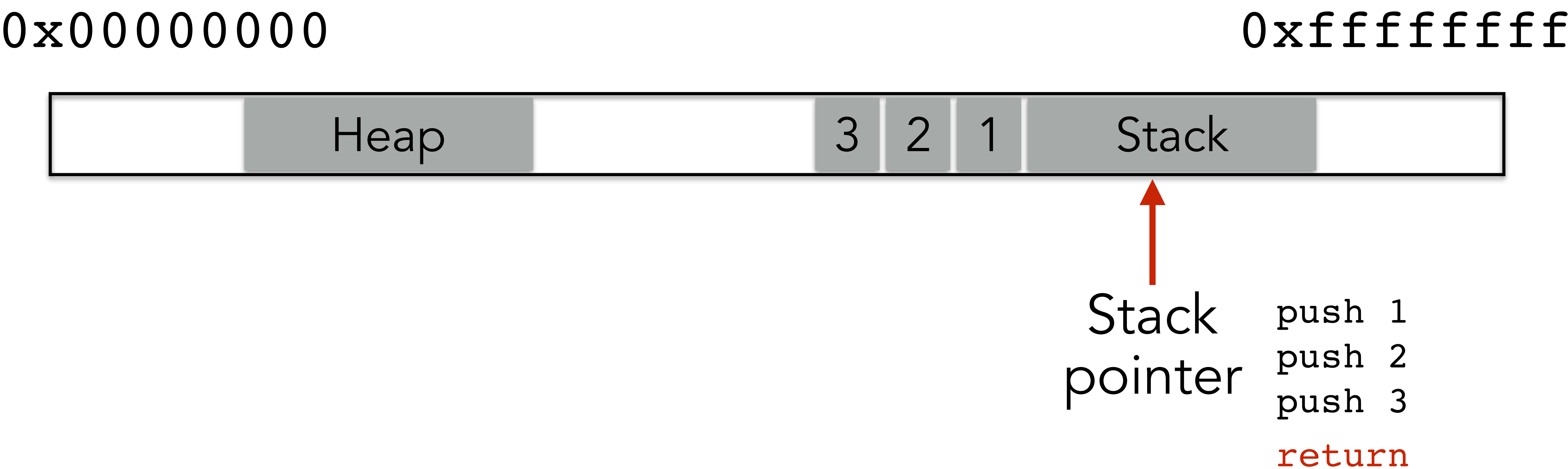
Compiler provides instructions that adjusts the size of the stack at runtime



WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

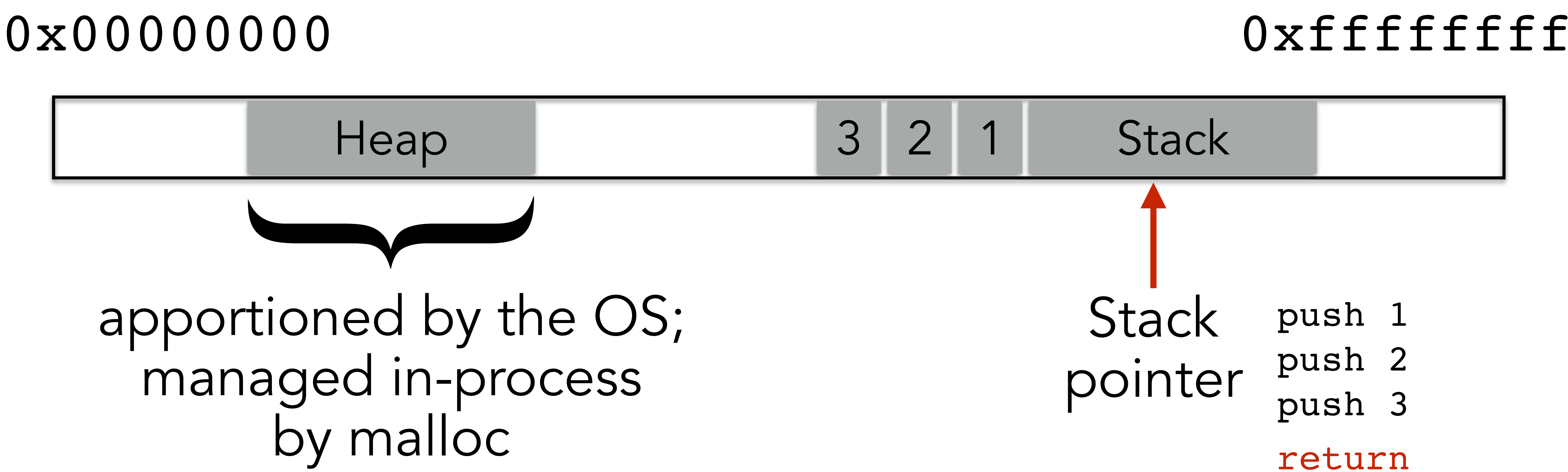
Compiler provides instructions that adjusts the size of the stack at runtime



WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

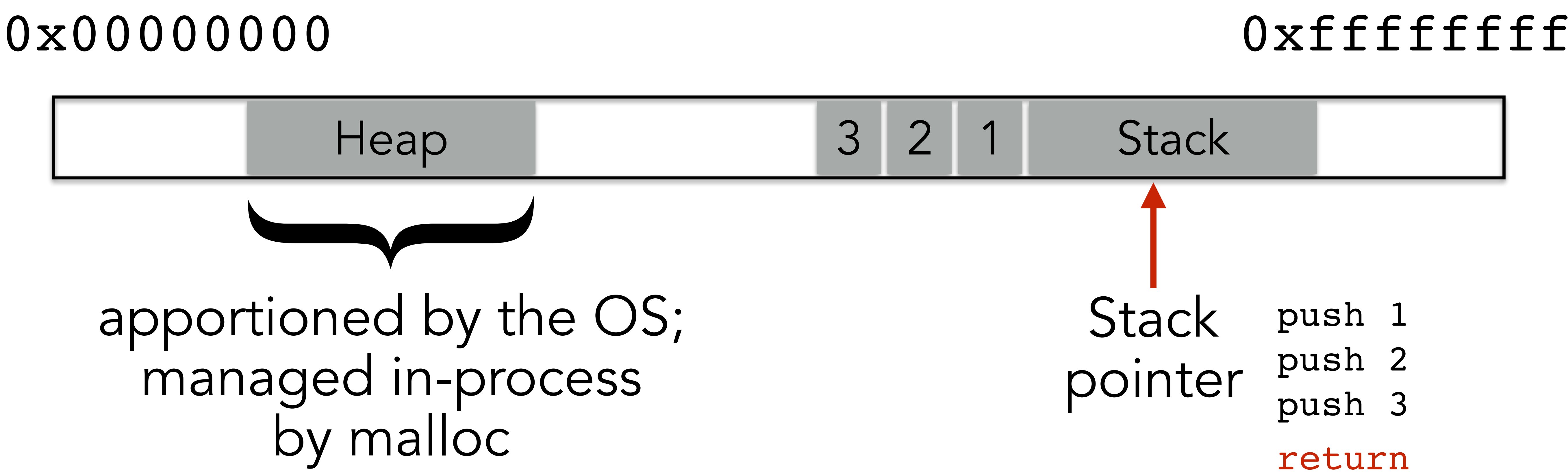
Compiler provides instructions that adjusts the size of the stack at runtime



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

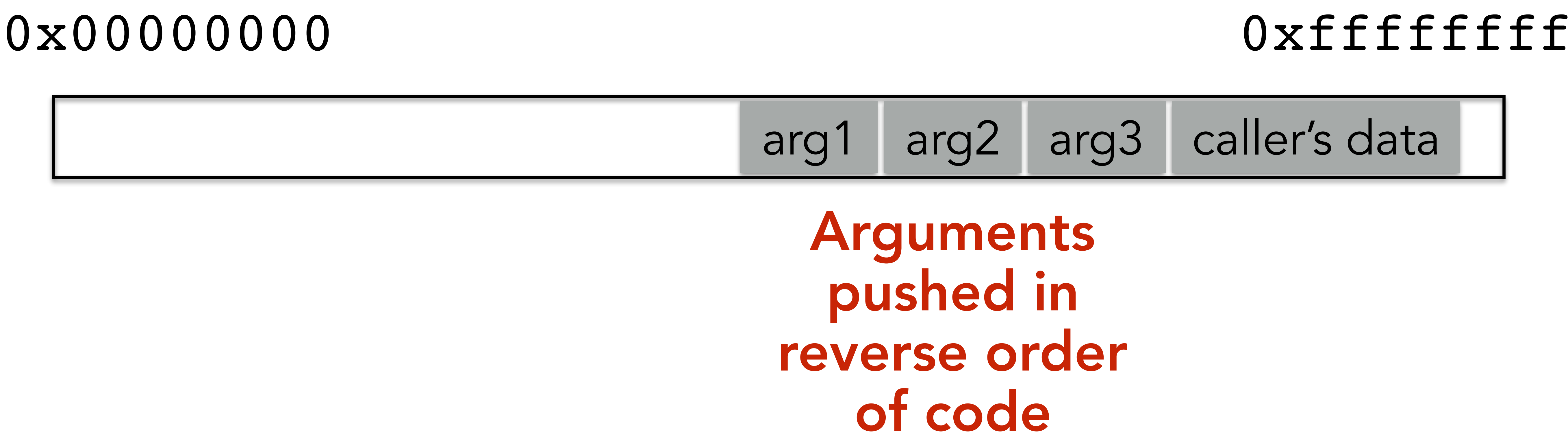
```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```

0x00000000 0xffffffff

caller's data

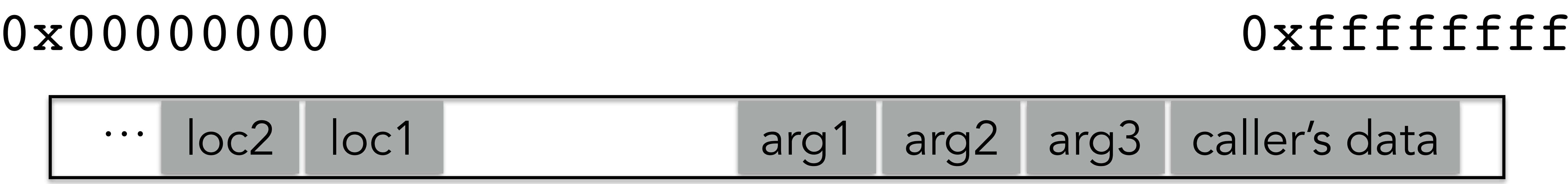
STACK LAYOUT WHEN CALLING FUNCTION

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



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;
     ...
}
```

0x00000000 0xffffffff

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

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

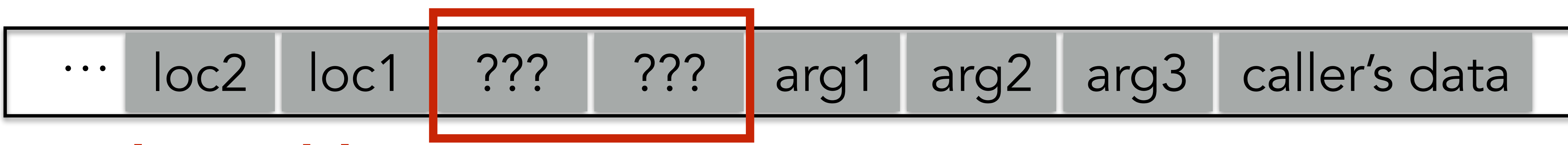
Arguments pushed in reverse order of code

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

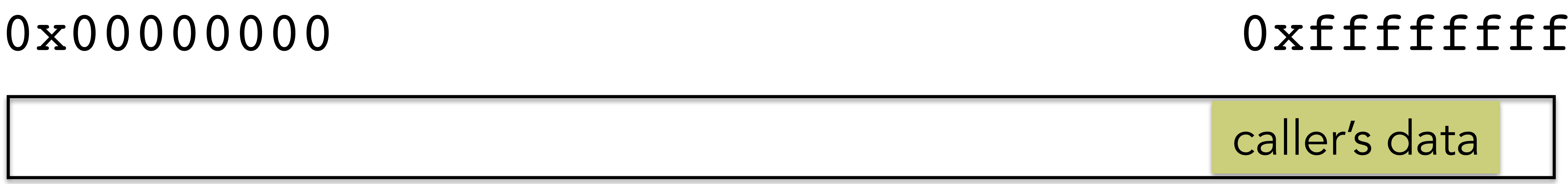


Local variables pushed in the same order as they appear in the code

Arguments pushed in reverse order of code

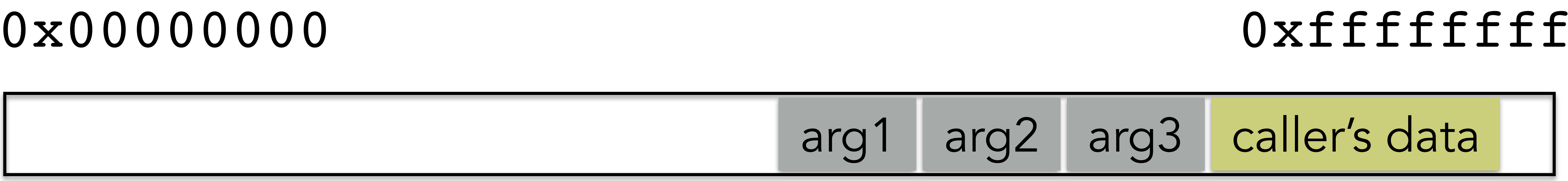
STACK LAYOUT WHEN CALLING FUNCTION

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



STACK LAYOUT WHEN CALLING FUNCTION

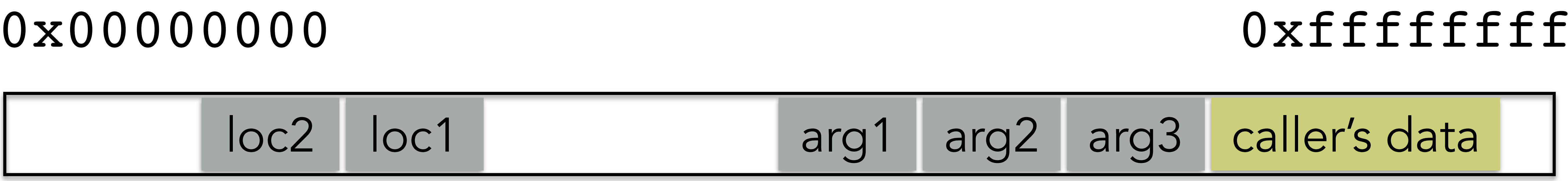
```c
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```



Arguments pushed in reverse order of code

STACK LAYOUT WHEN CALLING FUNCTION

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

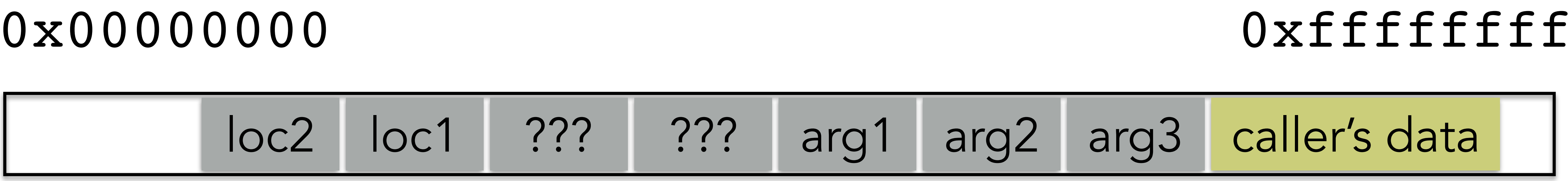


Local variables pushed in the same order as they appear in the code

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

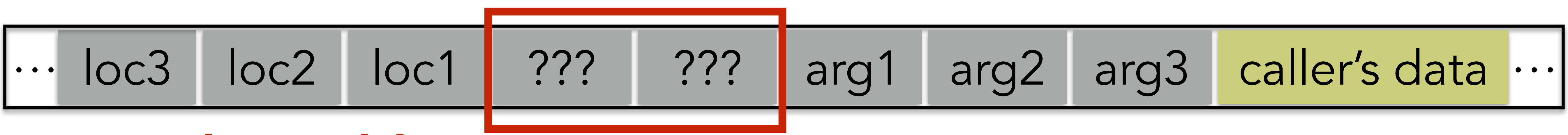
Arguments pushed in reverse order of code

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

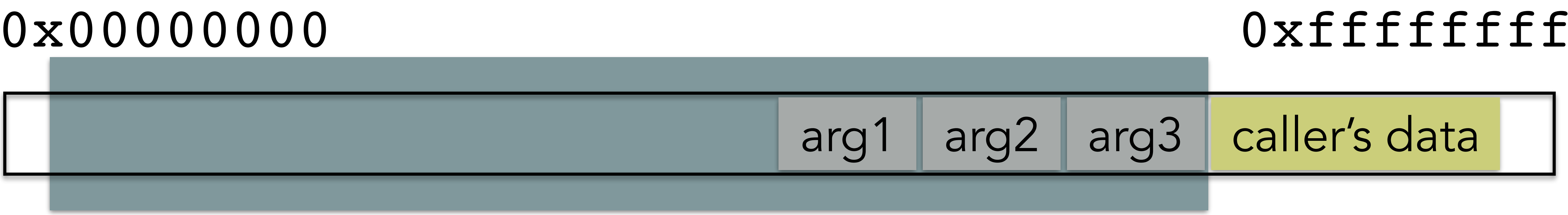
Arguments pushed in reverse order of code

STACK FRAMES

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

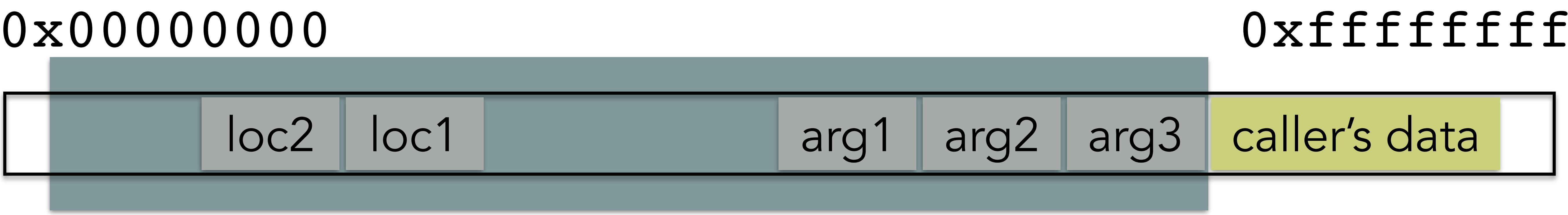
STACK FRAMES

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



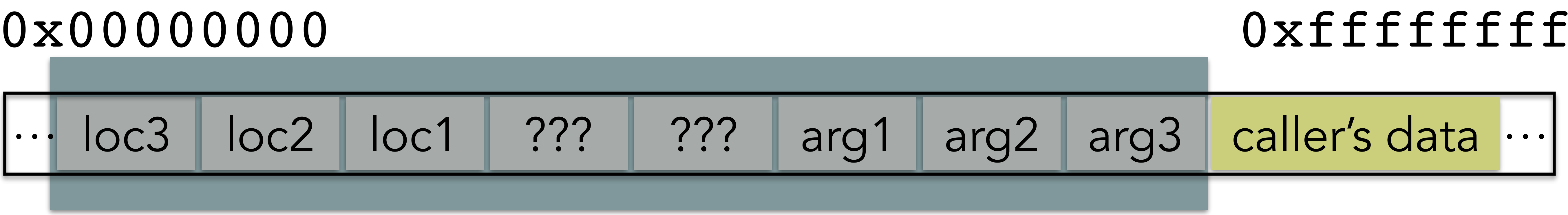
STACK FRAMES

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



STACK FRAMES

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



The part of the stack corresponding to
this particular invocation of
this particular function

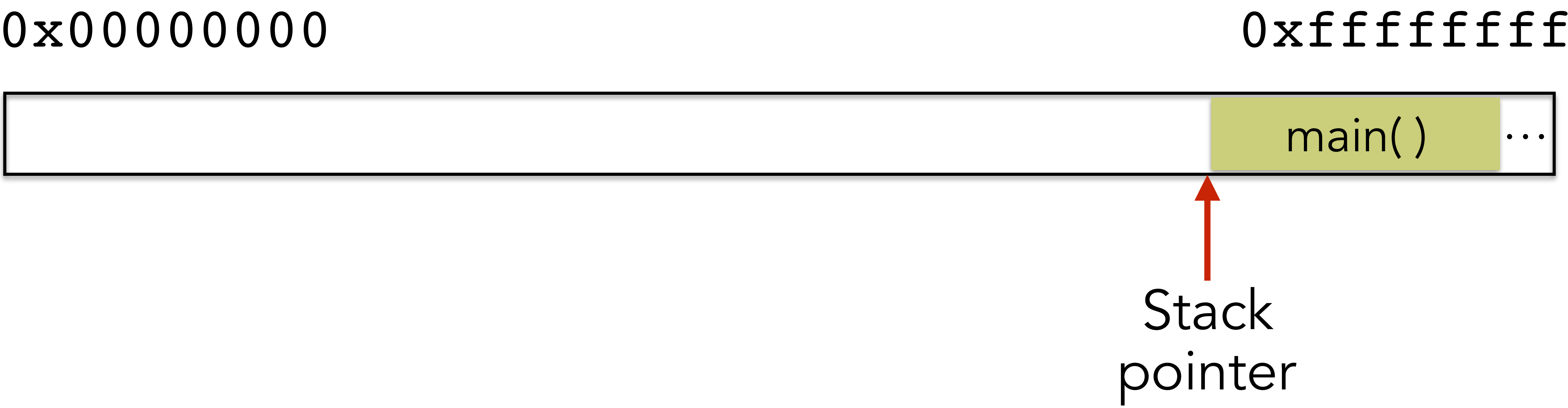
STACK FRAMES

```
void main() { countUp(3); }

void countUp(int n) {
   if(n > 1)
      countUp(n-1);
   printf(“%d\n”, n);
}
```

0x00000000 0xffffffff

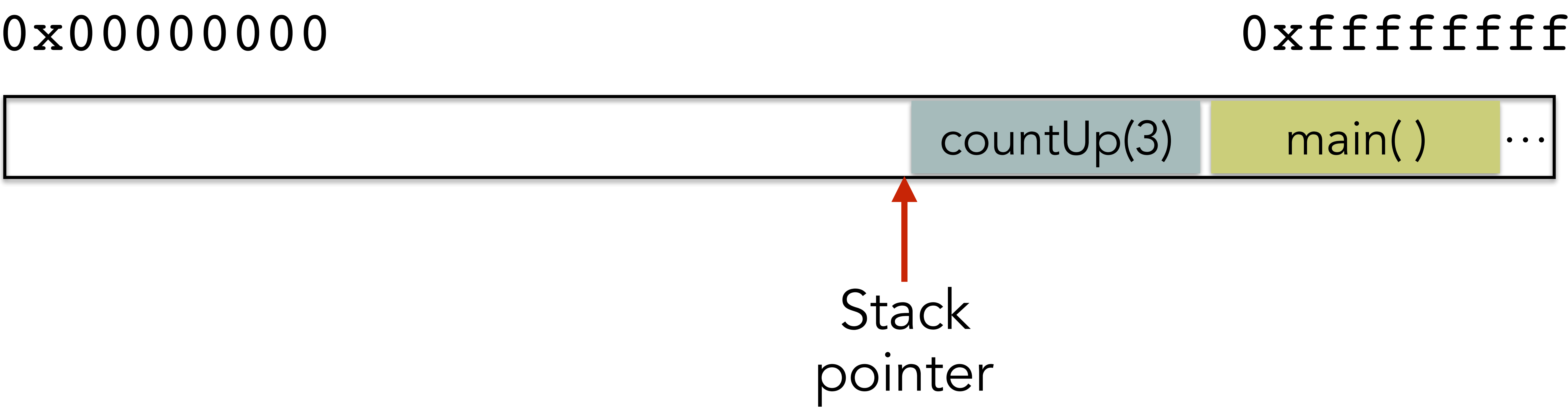
STACK FRAMES

```
void main() { countUp(3); }

void countUp(int n) {
   if(n > 1)
      countUp(n-1);
   printf("%d\n", n);
}
```



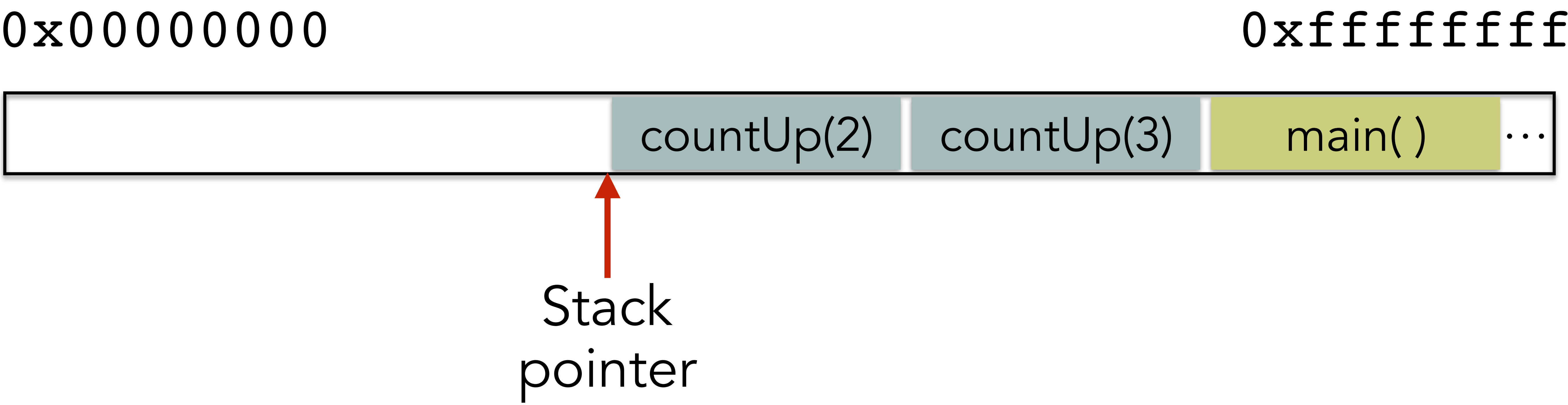
STACK FRAMES

```
void main() { countUp(3); }

void countUp(int n) {
   if(n > 1)
      countUp(n-1);
   printf("%d\n", n);
}
```



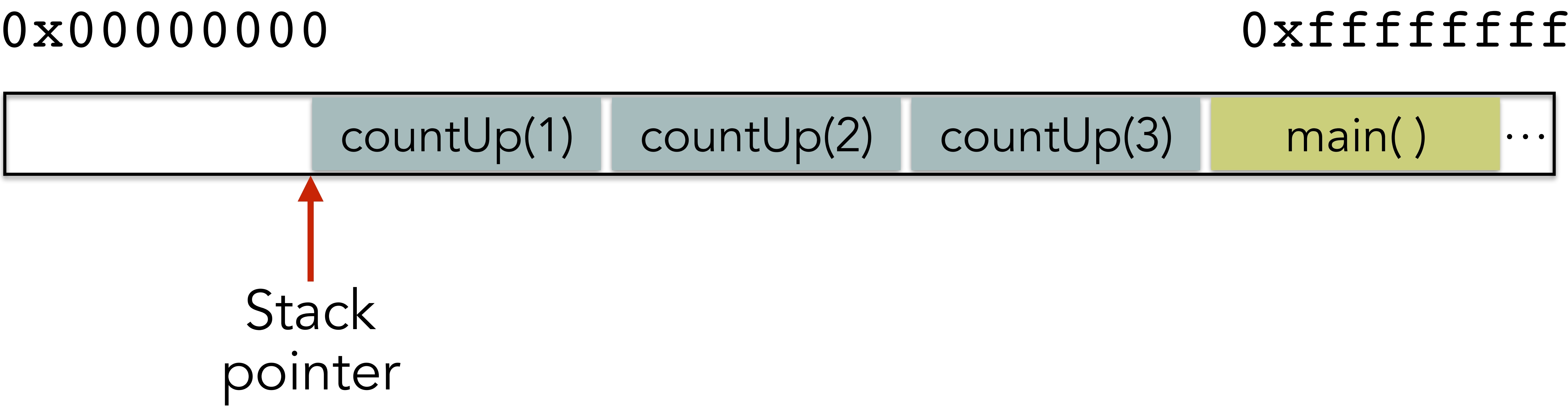
STACK FRAMES

```
void main() { countUp(3); }

void countUp(int n) {
   if(n > 1)
      countUp(n-1);
   printf("%d\n", n);
}
```



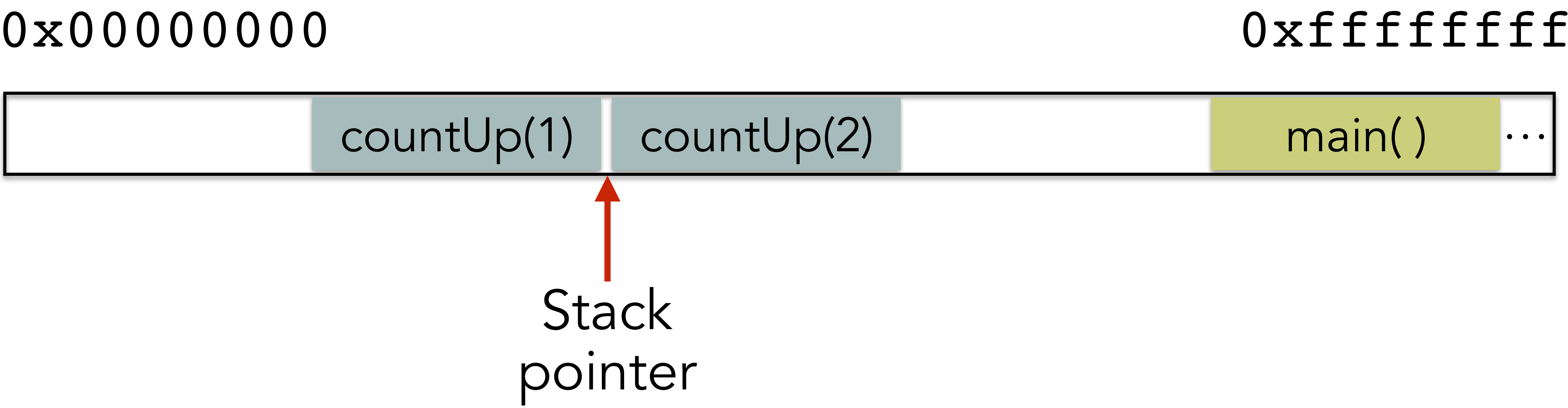
STACK FRAMES

```c
void main() { countUp(3); }

void countUp(int n) {
   if(n > 1)
      countUp(n-1);
   printf("%d\n", n);
}
```



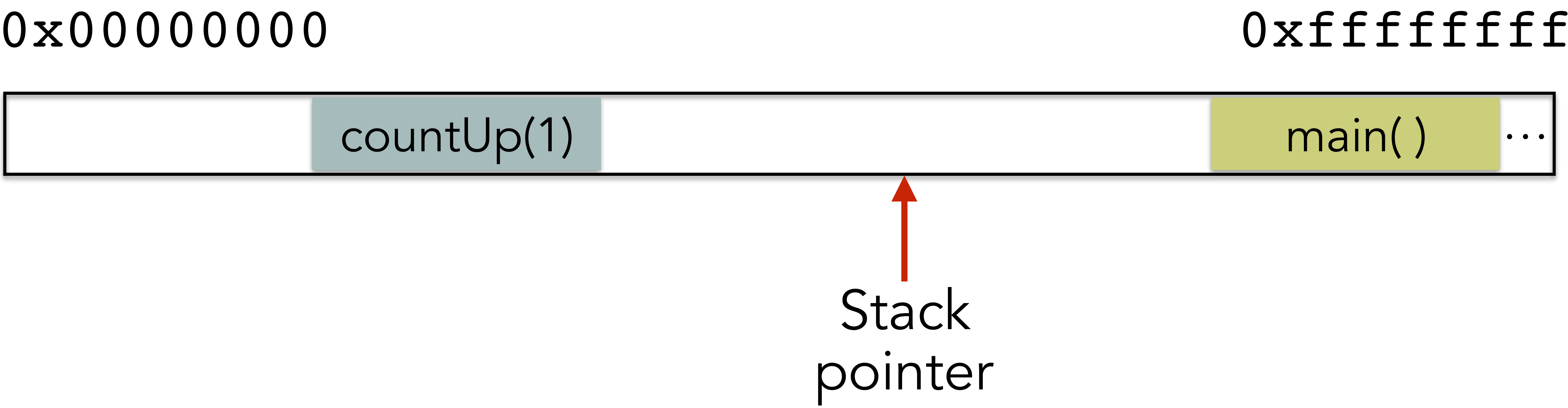
STACK FRAMES

```
void main() { countUp(3); }

void countUp(int n) {
   if(n > 1)
      countUp(n-1);
   printf("%d\n", n);
}
```



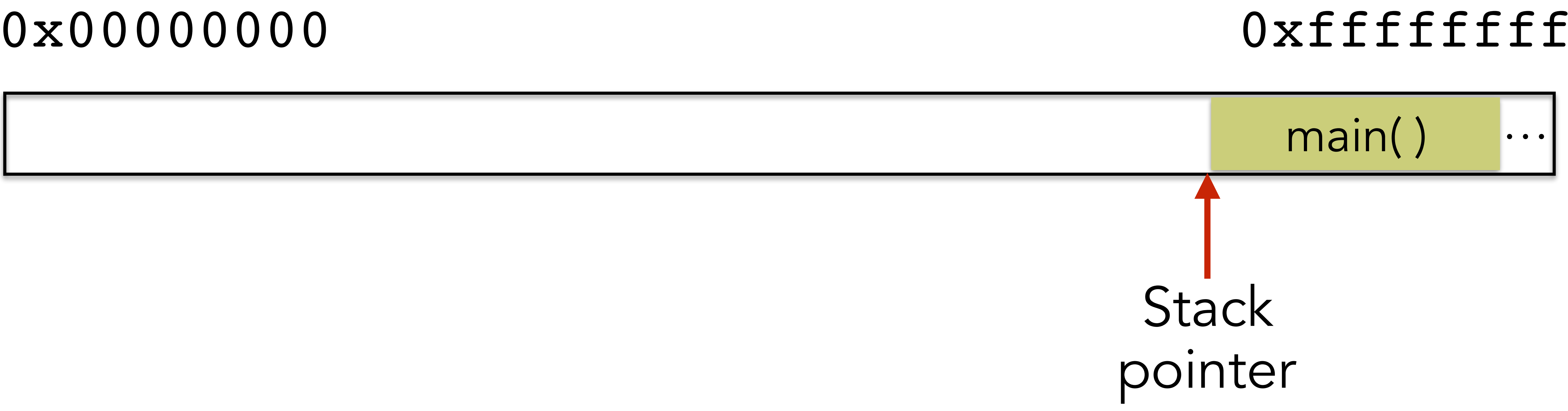
STACK FRAMES

```
void main() { countUp(3); }

void countUp(int n) {
   if(n > 1)
      countUp(n-1);
   printf("%d\n", n);
}
```



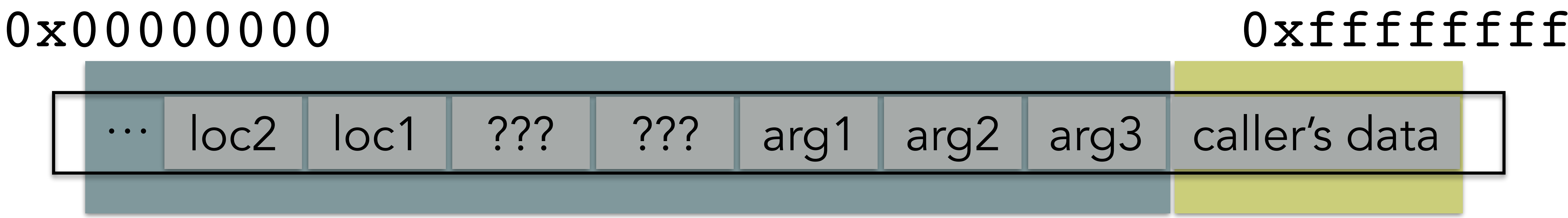
STACK FRAMES

```
void main() { countUp(3); }

void countUp(int n) {
   if(n > 1)
      countUp(n-1);
   printf("%d\n", n);
}
```



ACCESSING VARIABLES

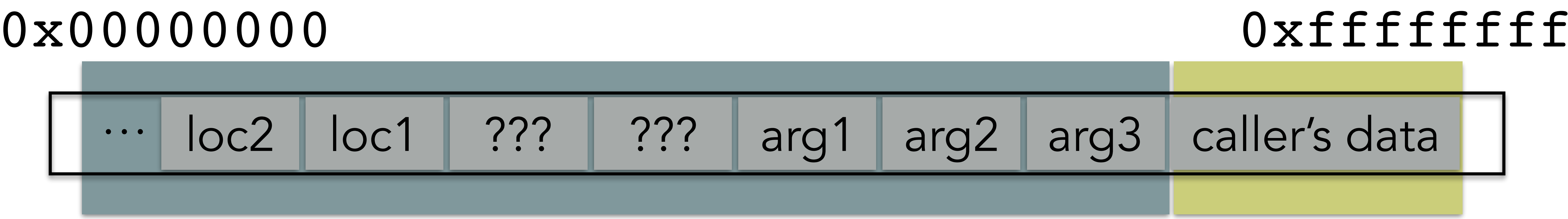
```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    loc2++;
}
```



ACCESSING VARIABLES

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

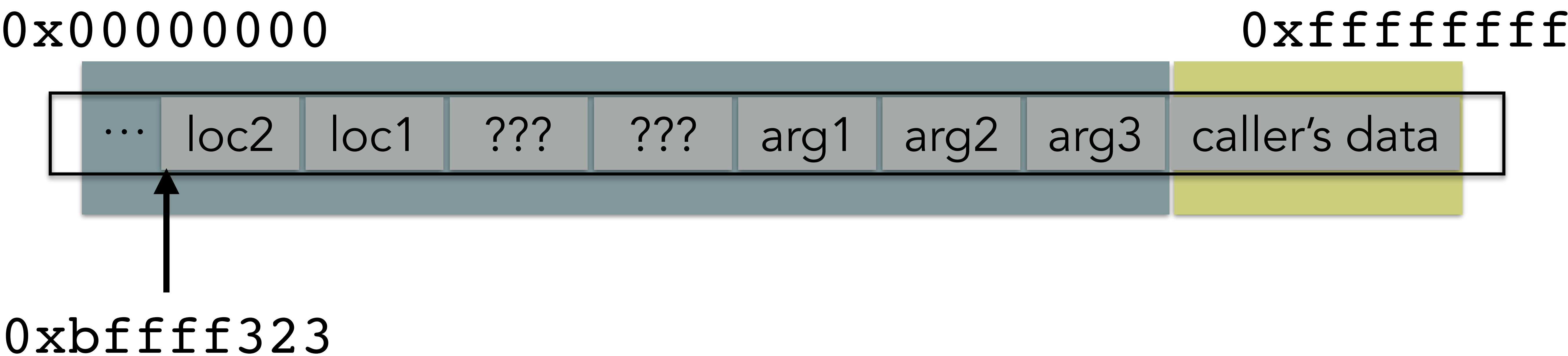
Q: Where is (this) loc2?



ACCESSING VARIABLES

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

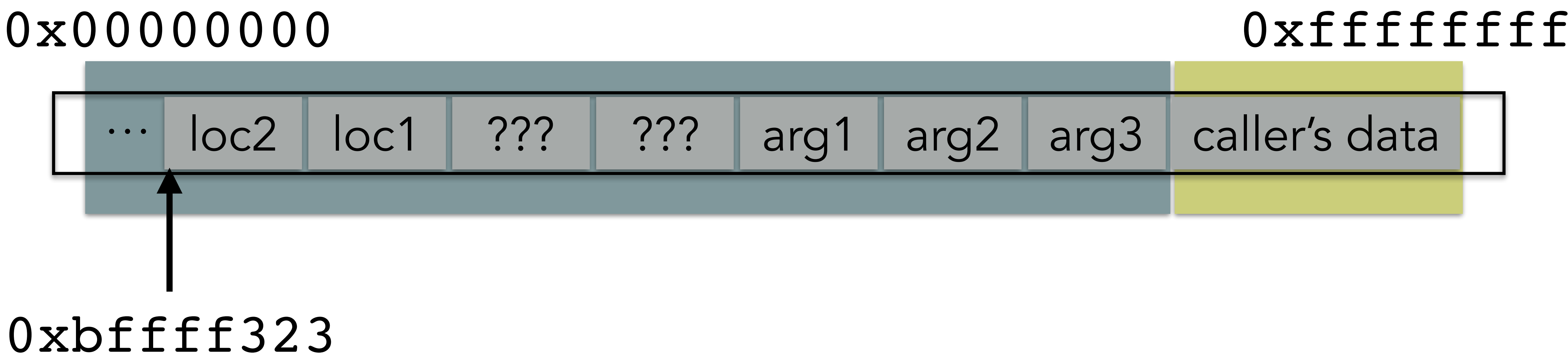
Q: Where is (this) loc2?



ACCESSING VARIABLES

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

Q: Where is (this) loc2?

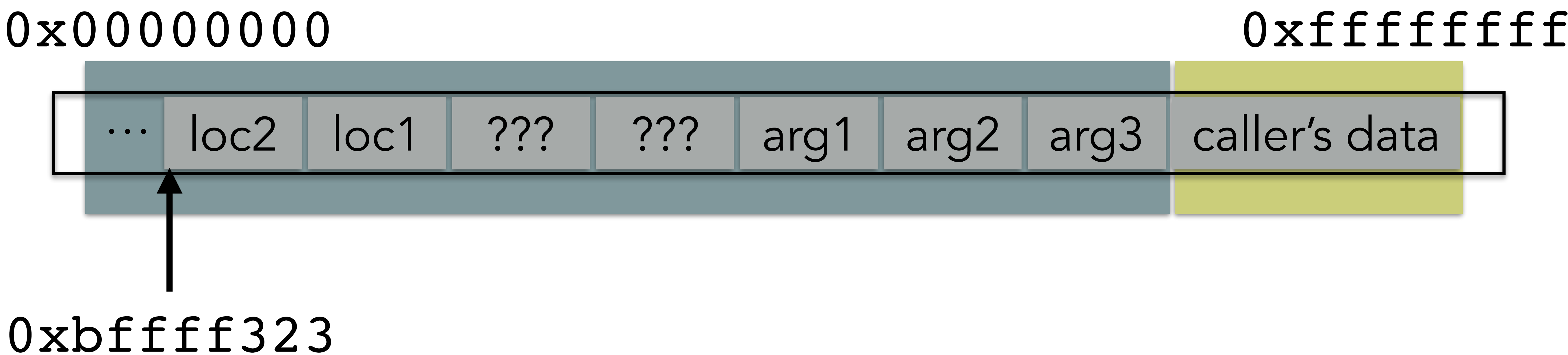


Undecidable at compile time

ACCESSING VARIABLES

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

Q: Where is (this) loc2?



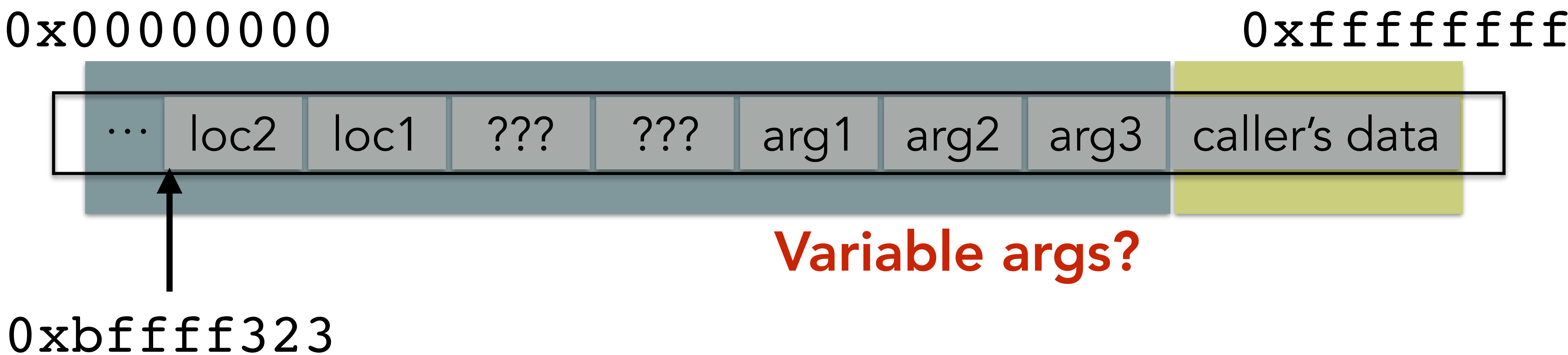
Undecidable at compile time

- 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?



Variable args?

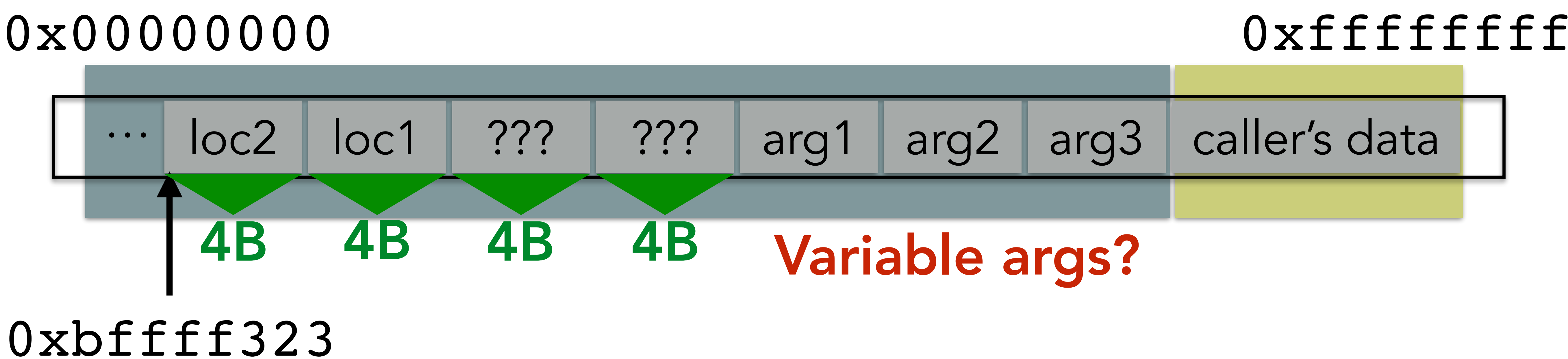
Undecidable at compile time

- 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?



Variable args?

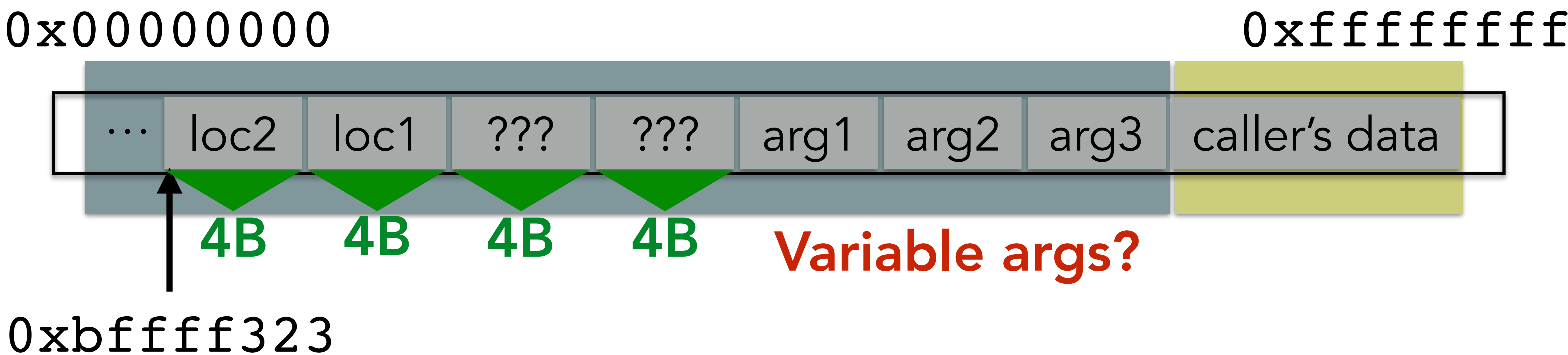
Undecidable at compile time

- 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?



Variable args?

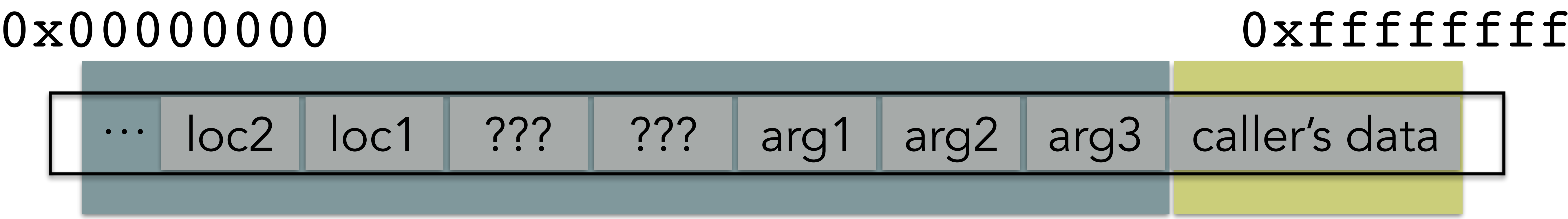
Undecidable at compile time

- 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?

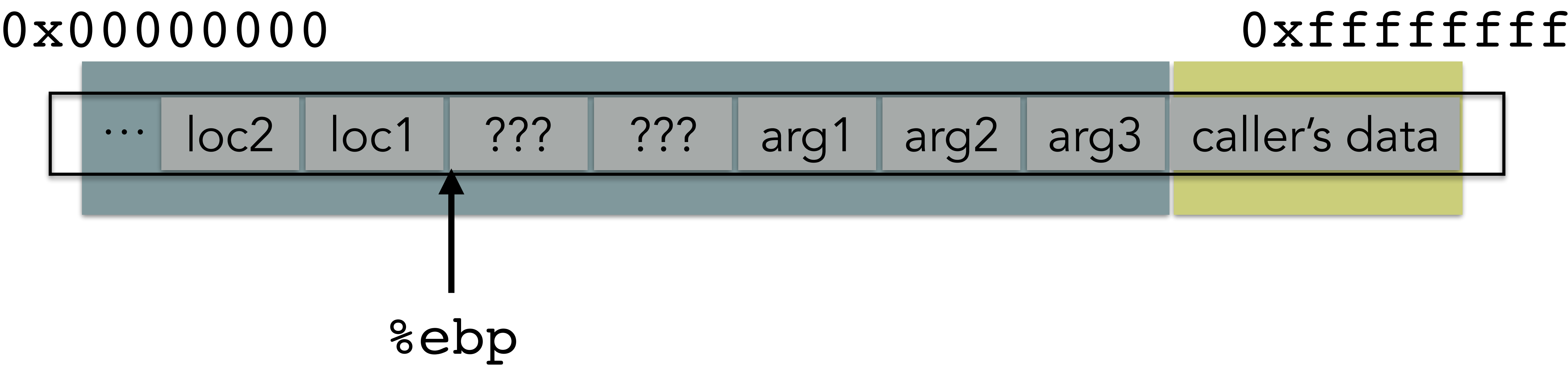


- I don't know where loc2 is,
- and I don't know how many args
- *but* loc2 is *always* 8B before "???"s

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

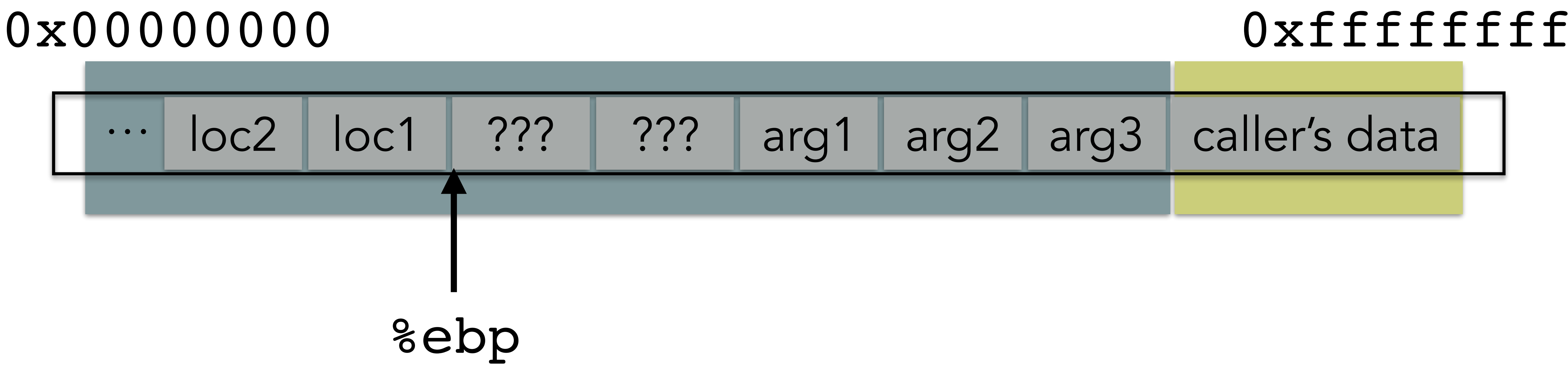
- I don't know where loc2 is,
- and I don't know how many args
- *but* loc2 is *always* 8B before "???"s

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

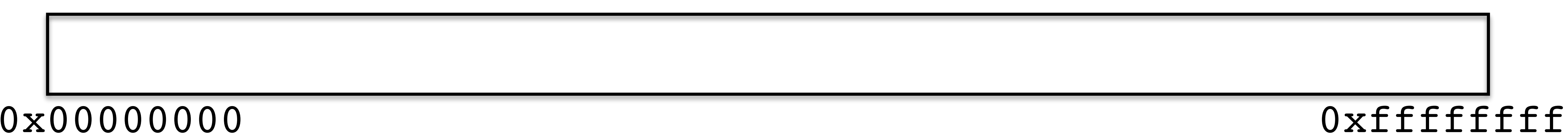
- I don't know where loc2 is,
- and I don't know how many args
- *but* loc2 is *always* 8B before "???"s

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)



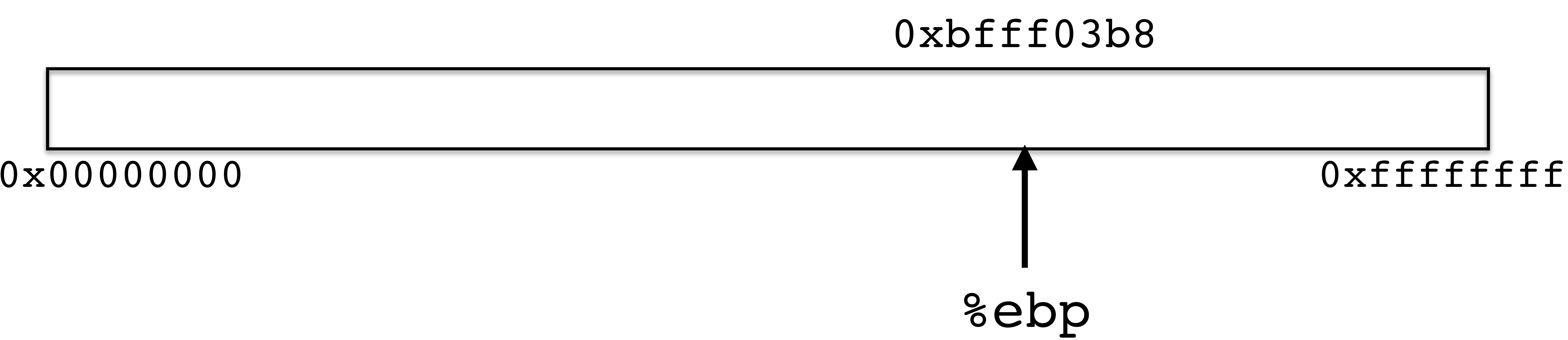
NOTATION

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

`0x00000000` `0xffffffff`

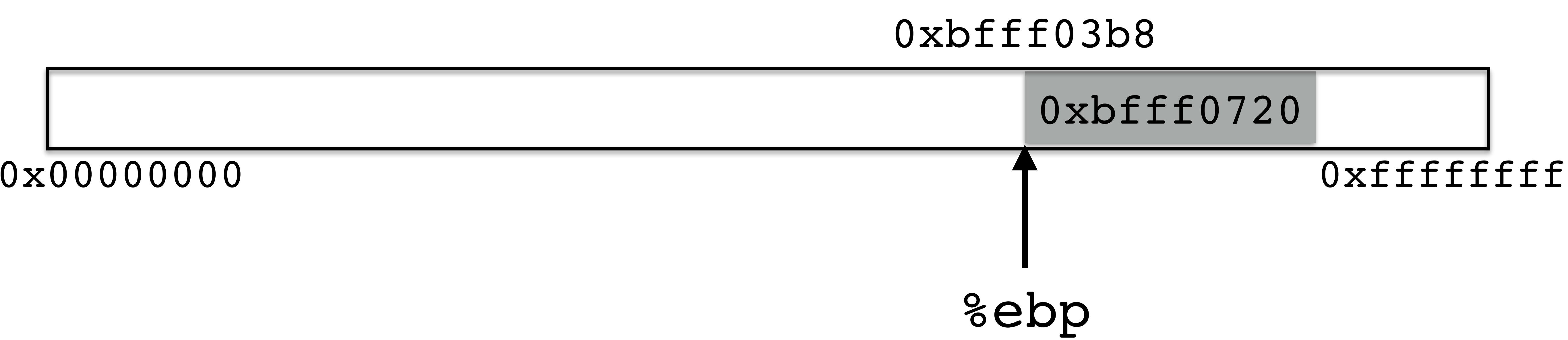
NOTATION

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



NOTATION

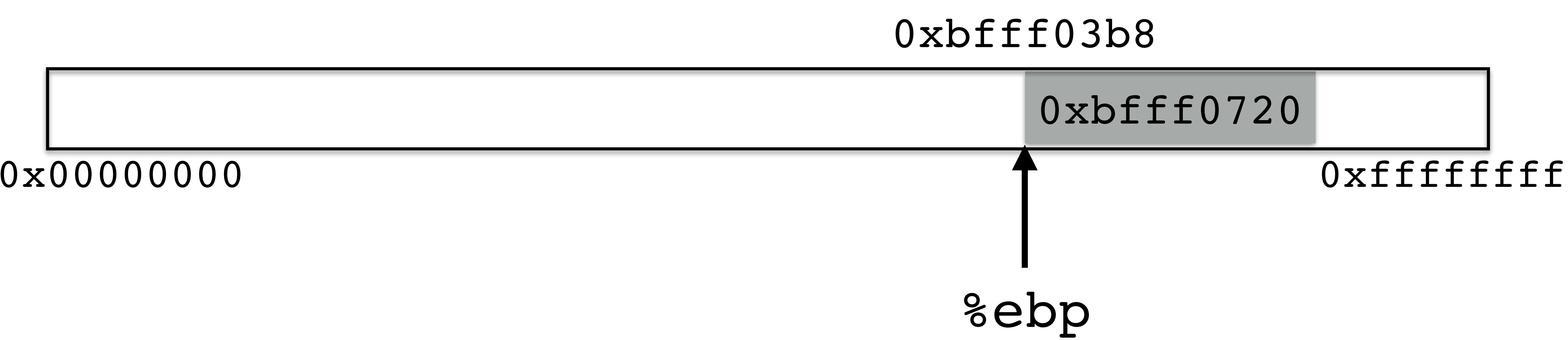
`0xbfff03b8`	`%ebp`	A memory address
`0xbfff0720`	`(%ebp)`	The value at memory address %ebp (like dereferencing a pointer)



NOTATION

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

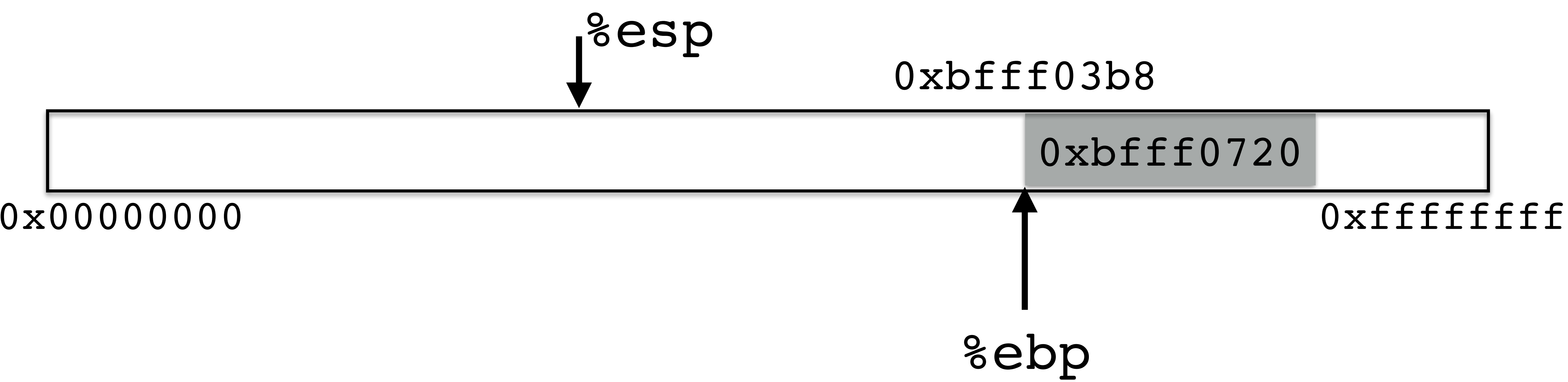
`pushl %ebp`



NOTATION

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

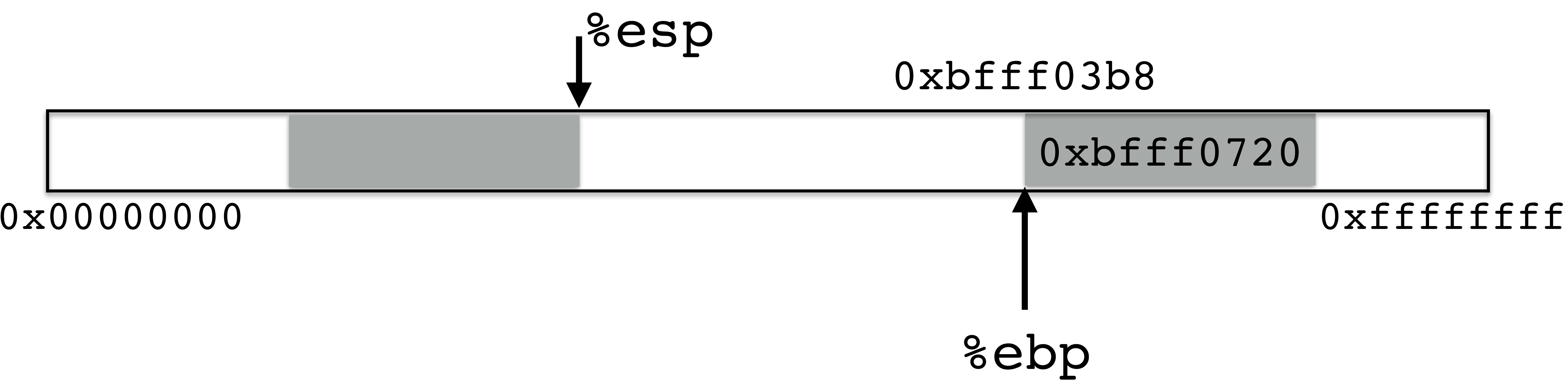
```
pushl %ebp
```



NOTATION

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

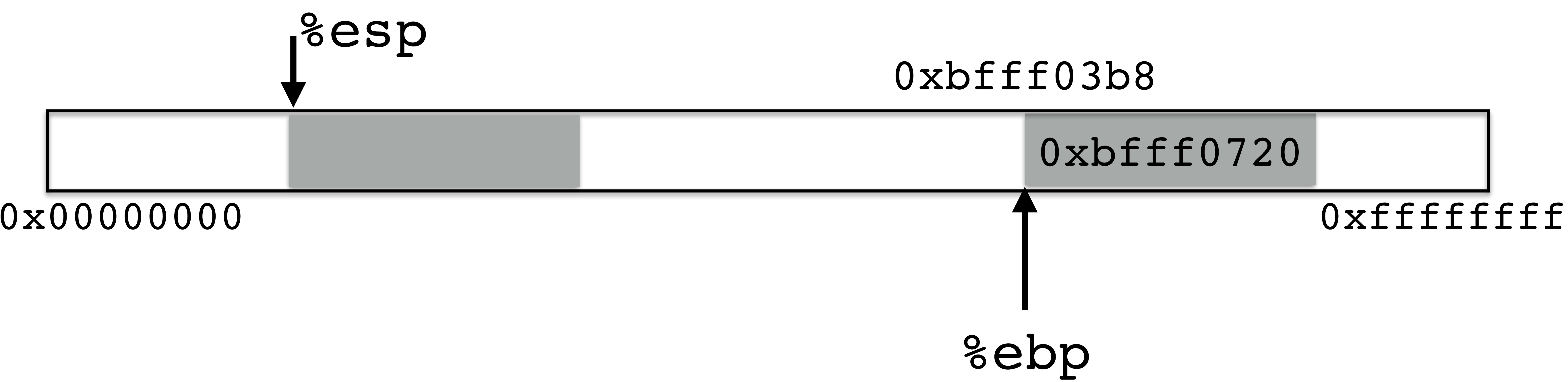
```
pushl %ebp
```



NOTATION

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

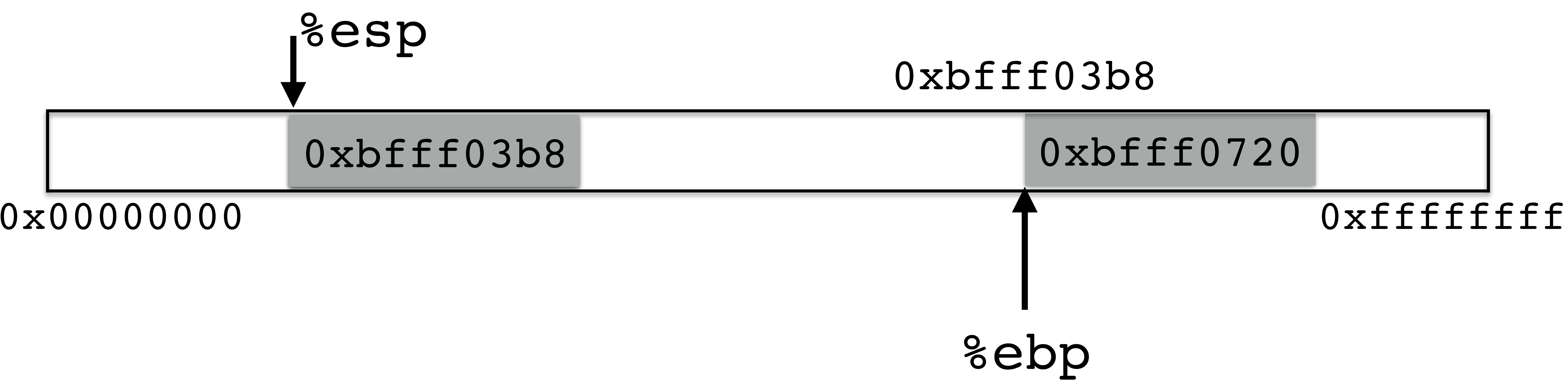
`pushl %ebp`



NOTATION

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

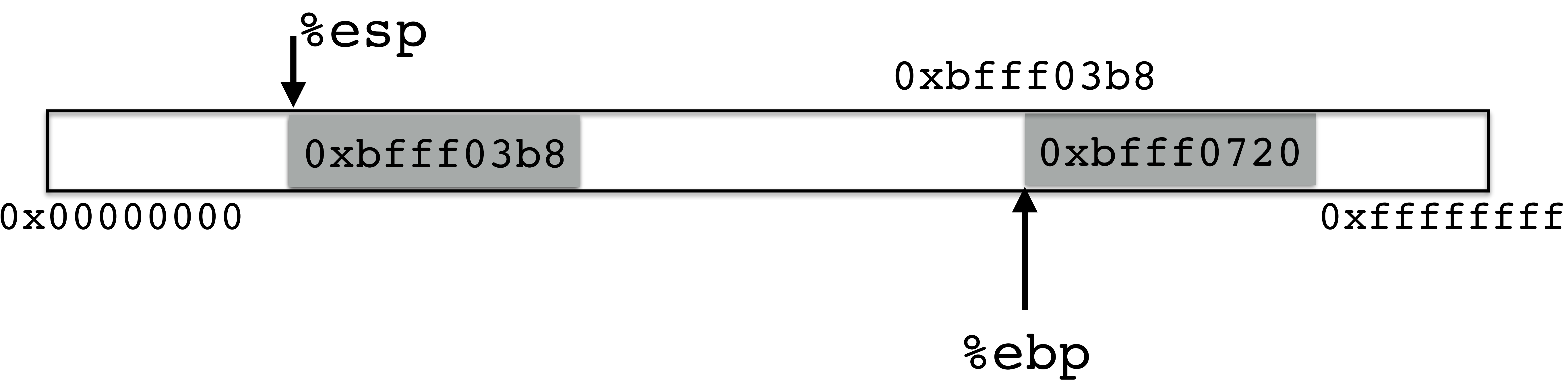
`pushl %ebp`



NOTATION

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

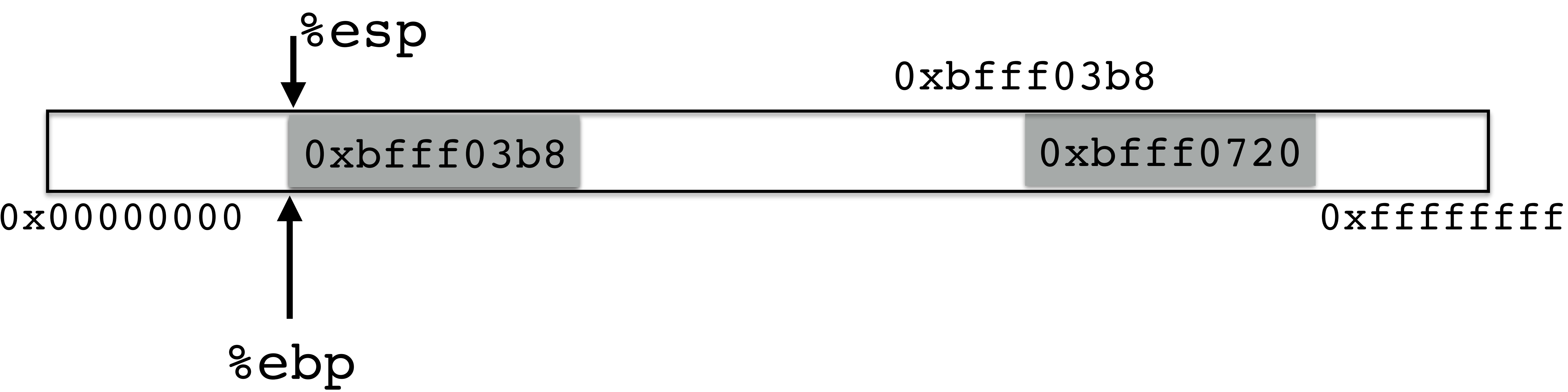
```
pushl %ebp
movl  %esp %ebp  /* %ebp = %esp */
```



NOTATION

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

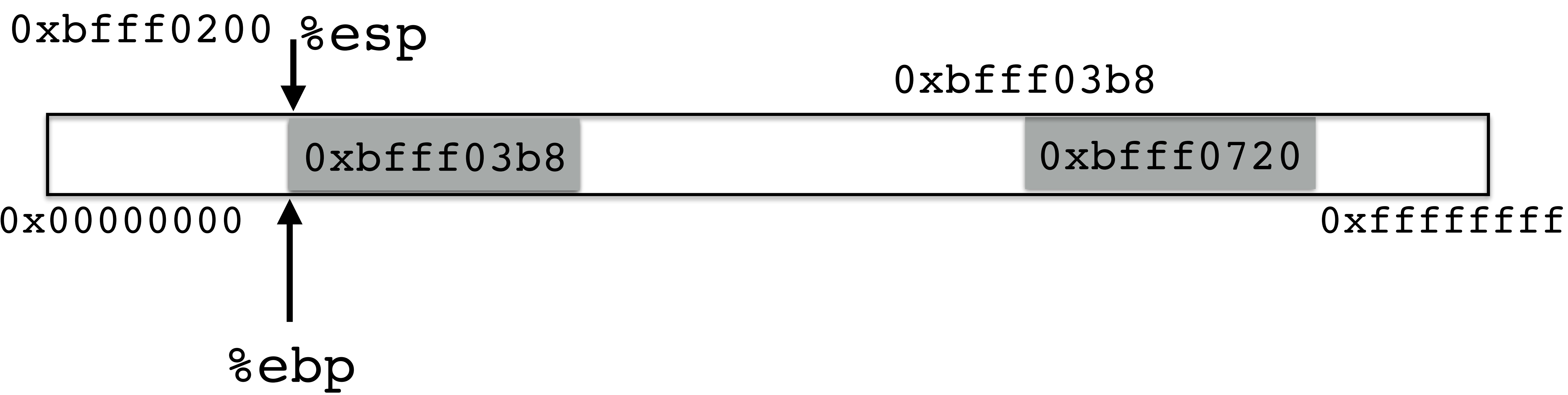
```
pushl %ebp
movl  %esp %ebp /* %ebp = %esp */
```



NOTATION

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

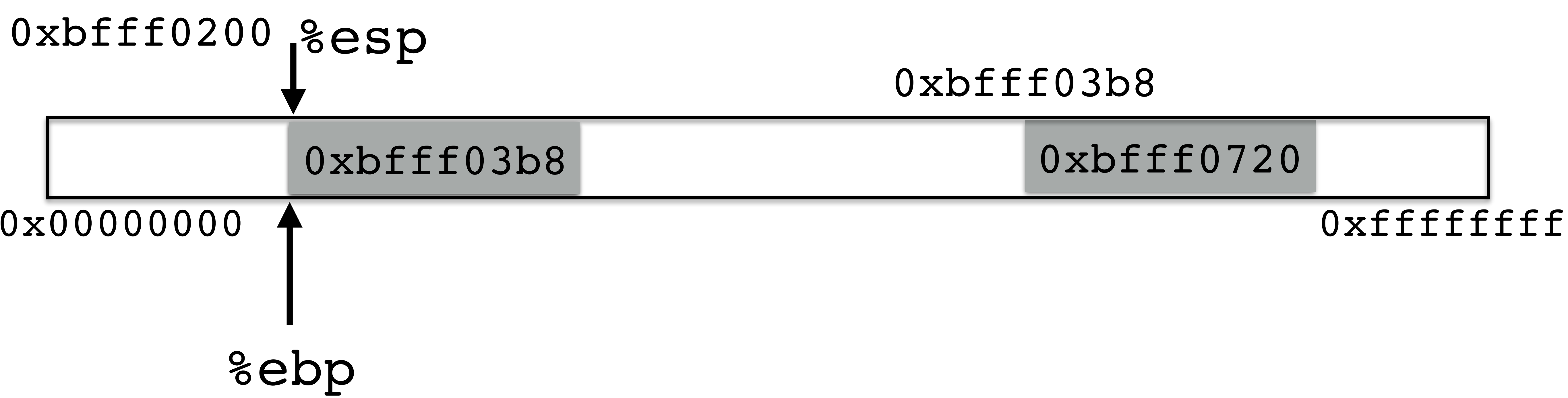
```
pushl %ebp
movl  %esp %ebp /* %ebp = %esp */
```



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 */
```

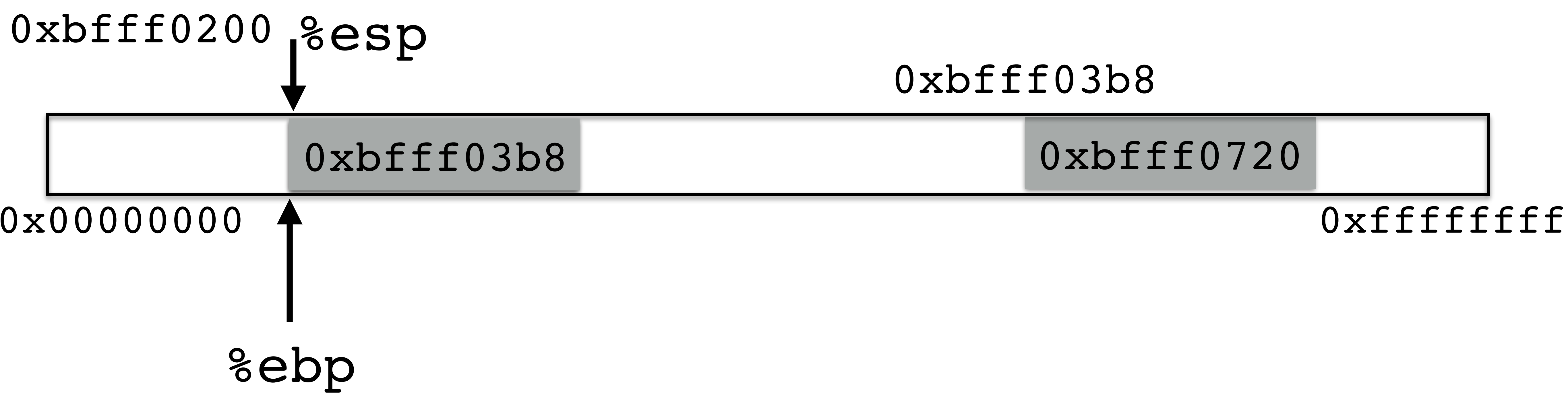


NOTATION

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

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

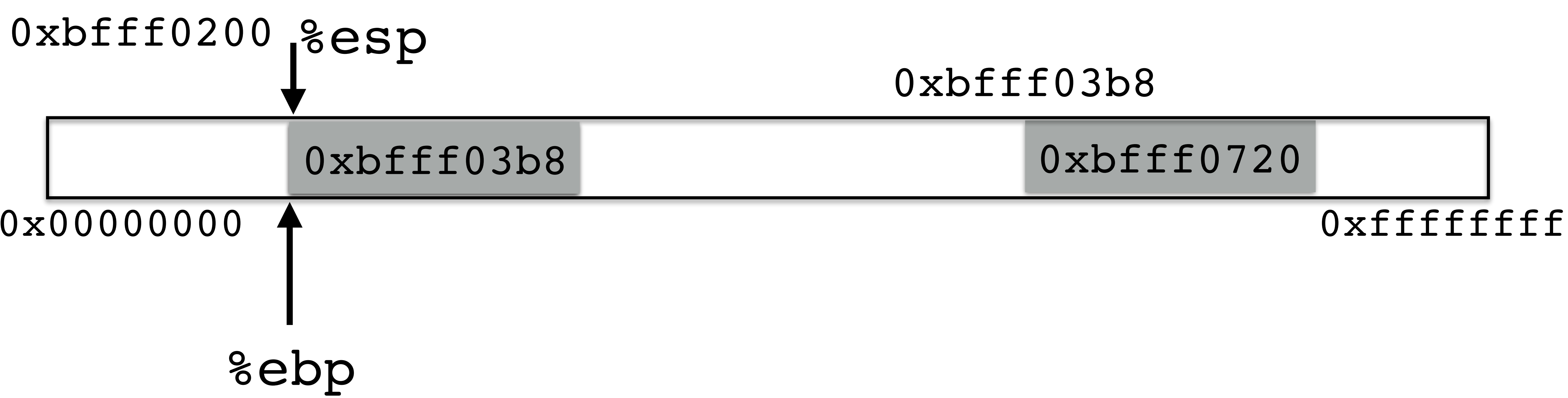
```
pushl %ebp
movl  %esp %ebp   /* %ebp = %esp */
```



NOTATION

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

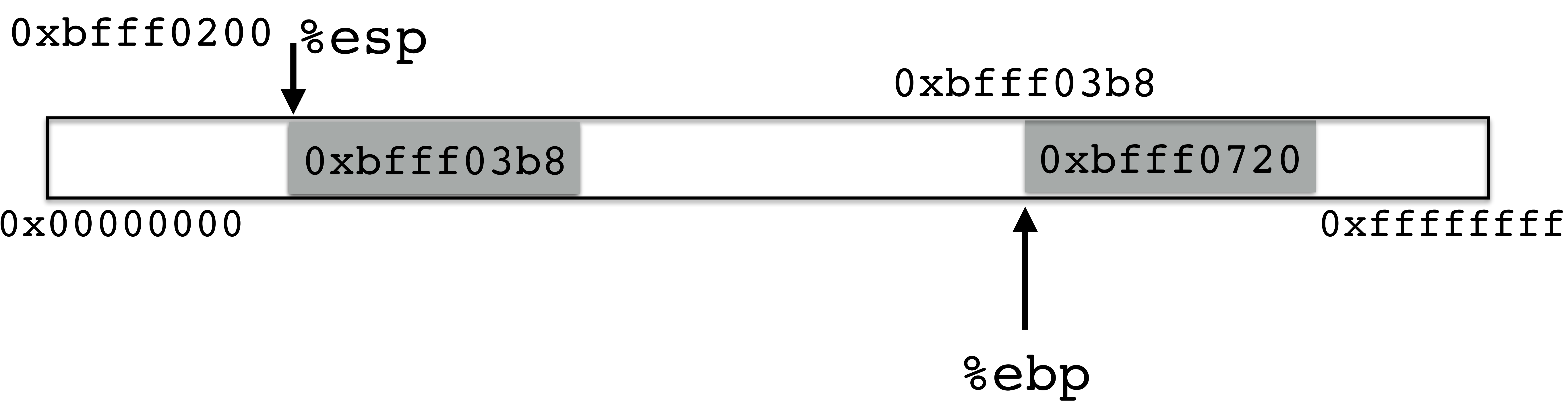
```
pushl %ebp
movl  %esp %ebp    /* %ebp = %esp */
movl  (%ebp) %ebp  /* %ebp = (%ebp) */
```



NOTATION

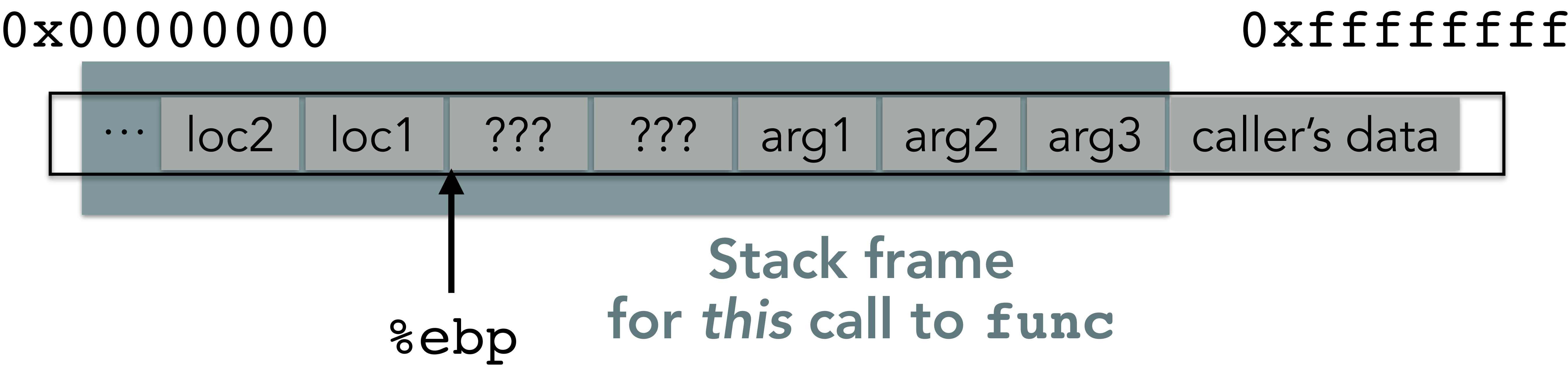
~~0xbfff03b8~~ 0xbfff0200	%ebp	A memory address
~~0xbfff0720~~ 0xbfff03b8	(%ebp)	The value at memory address %ebp (like dereferencing a pointer)

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



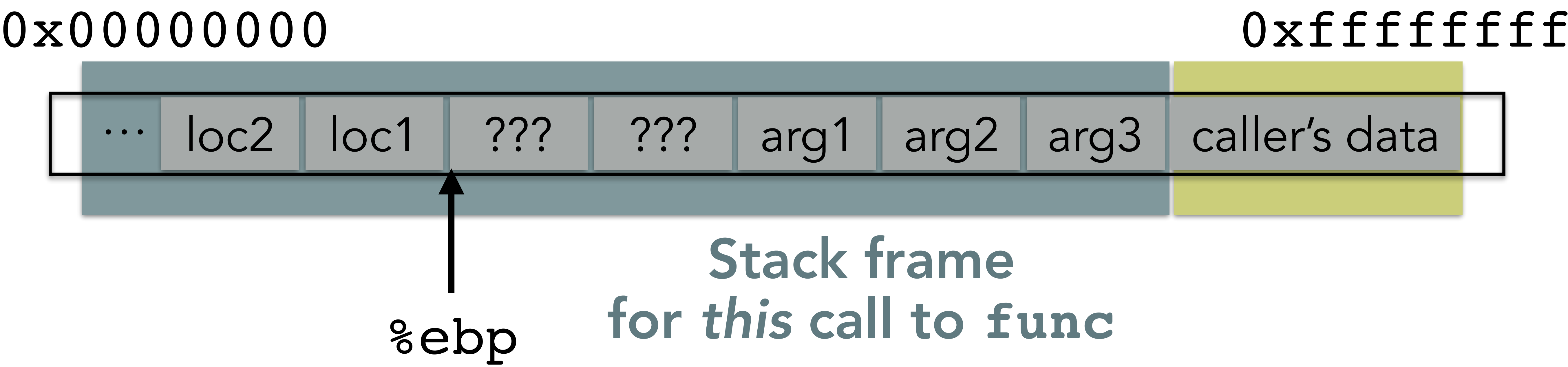
RETURNING FROM FUNCTIONS

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



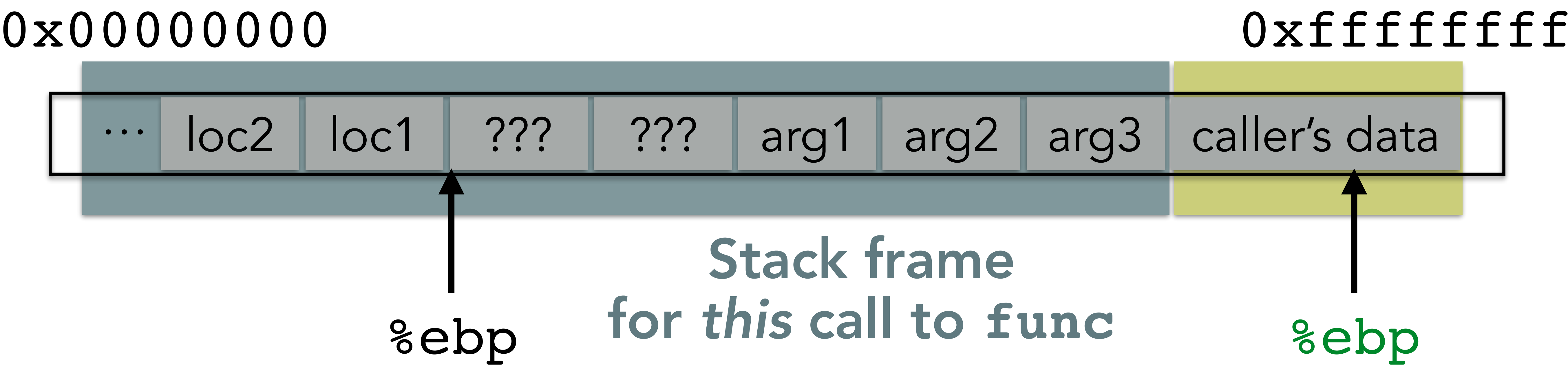
RETURNING FROM FUNCTIONS

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



RETURNING FROM FUNCTIONS

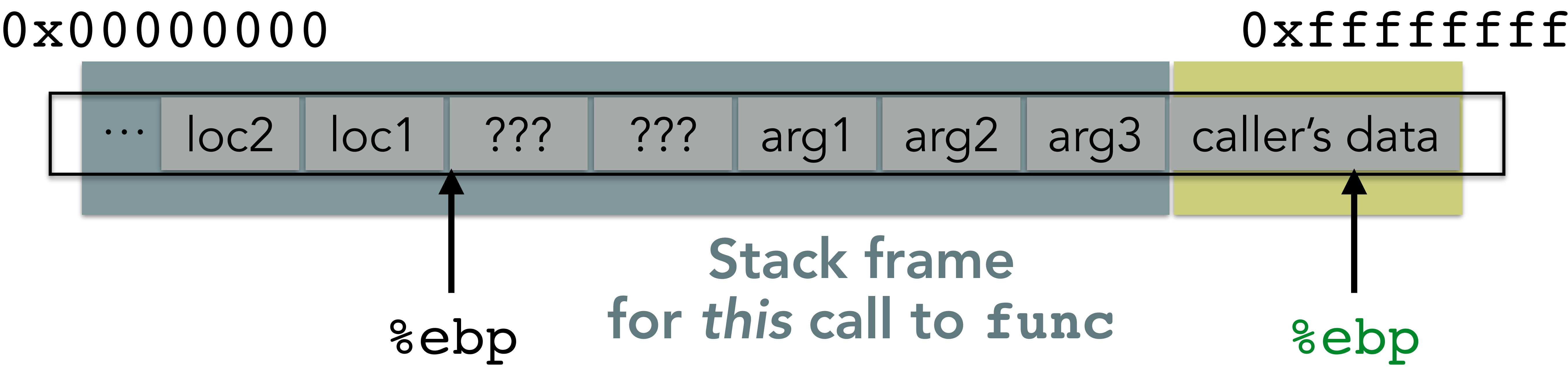
```
int main()
{
     ...
     func("Hey", 10, -3);
     ...
}
```



RETURNING FROM FUNCTIONS

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

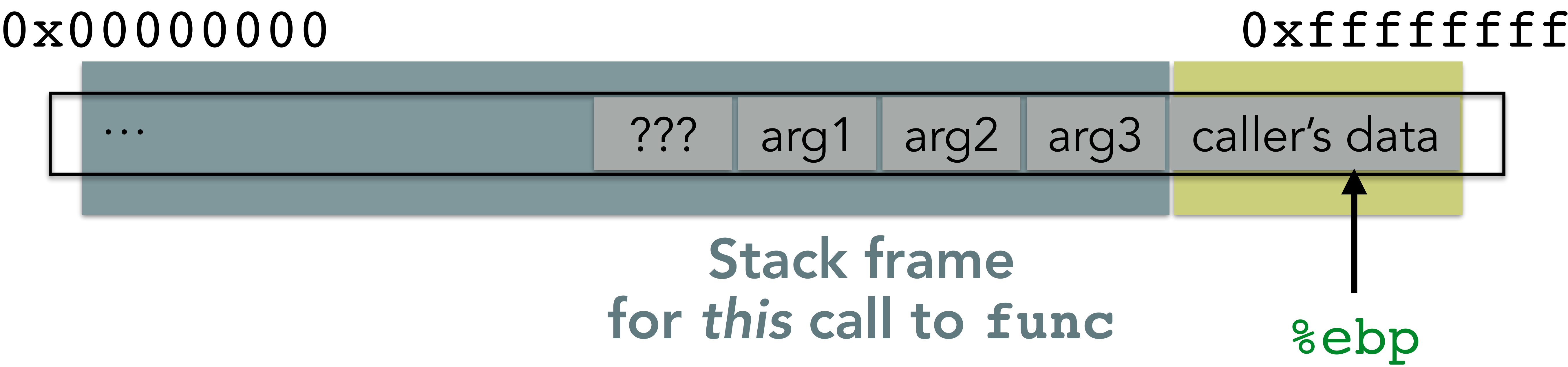
Q: How do we restore %ebp?



RETURNING FROM FUNCTIONS

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

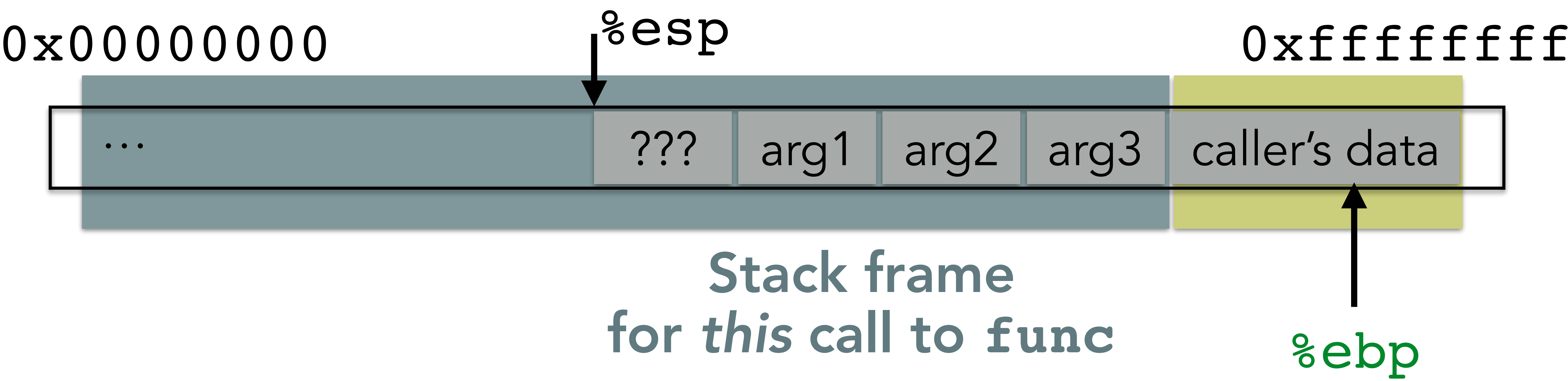
Q: How do we restore %ebp?



RETURNING FROM FUNCTIONS

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

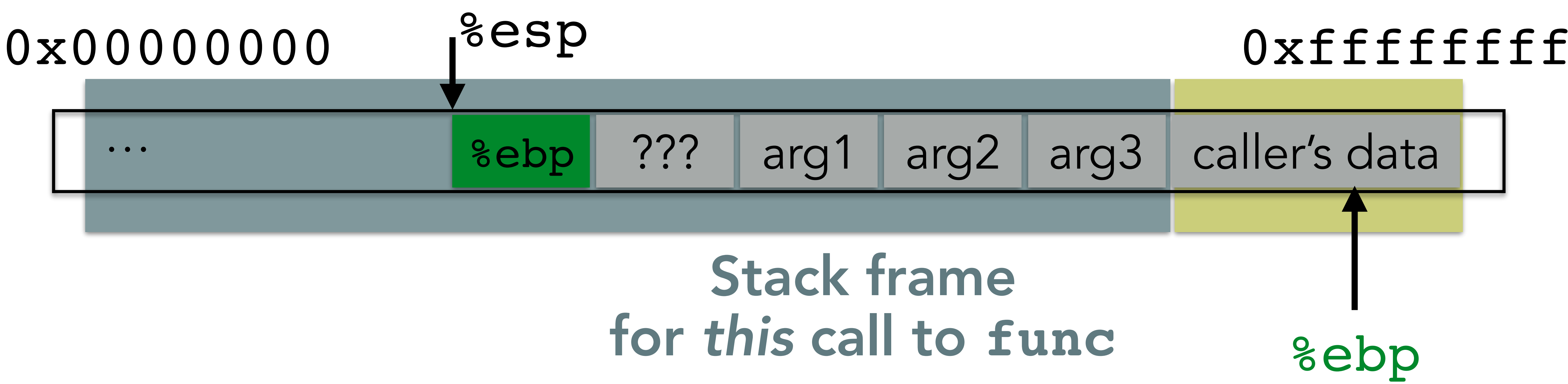
Q: How do we restore %ebp?



RETURNING FROM FUNCTIONS

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

Q: How do we restore %ebp?

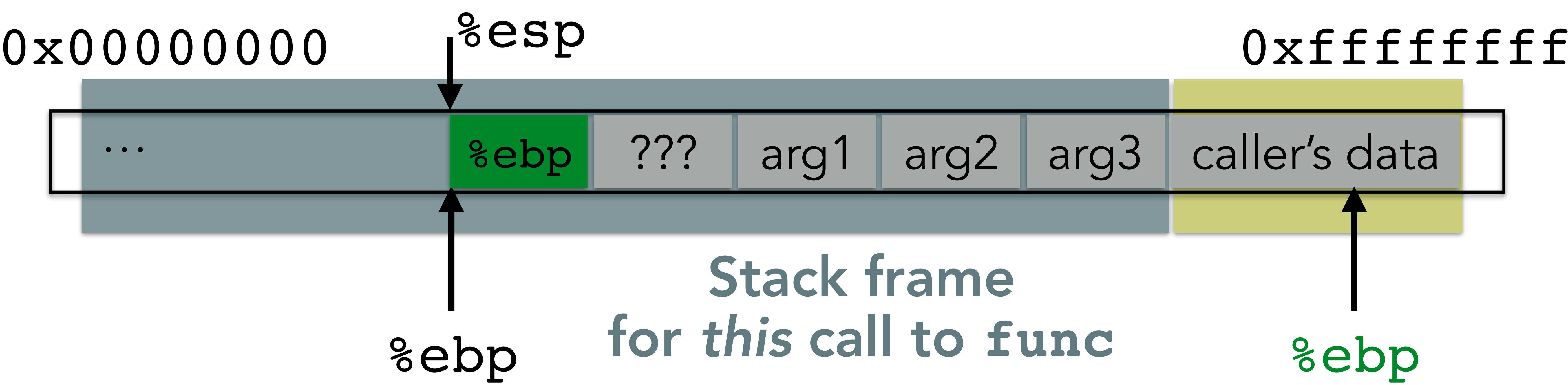


1. Push %ebp before locals

RETURNING FROM FUNCTIONS

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

Q: How do we restore %ebp?

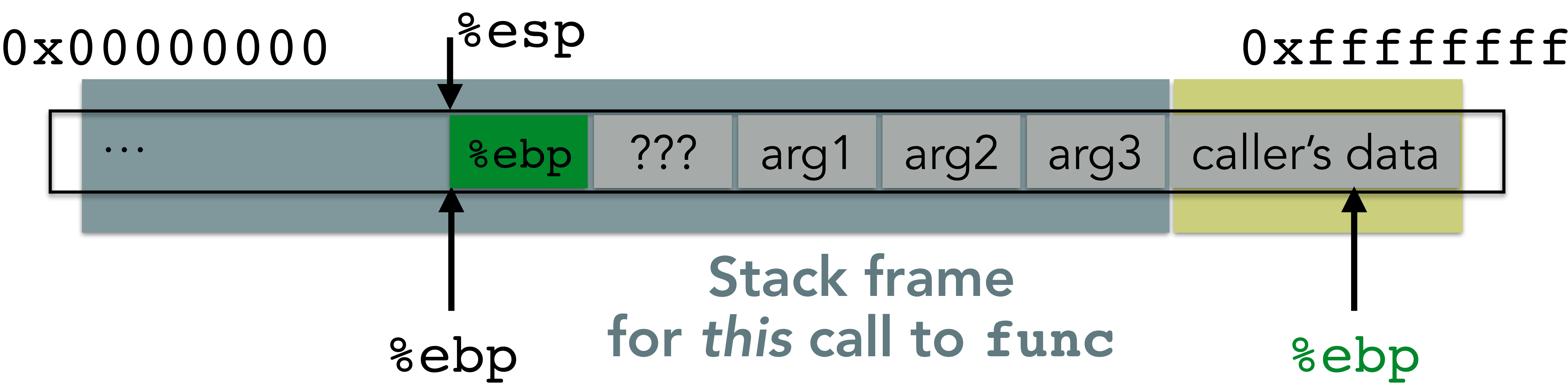


1. Push %ebp before locals
2. Set %ebp to current %esp

RETURNING FROM FUNCTIONS

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

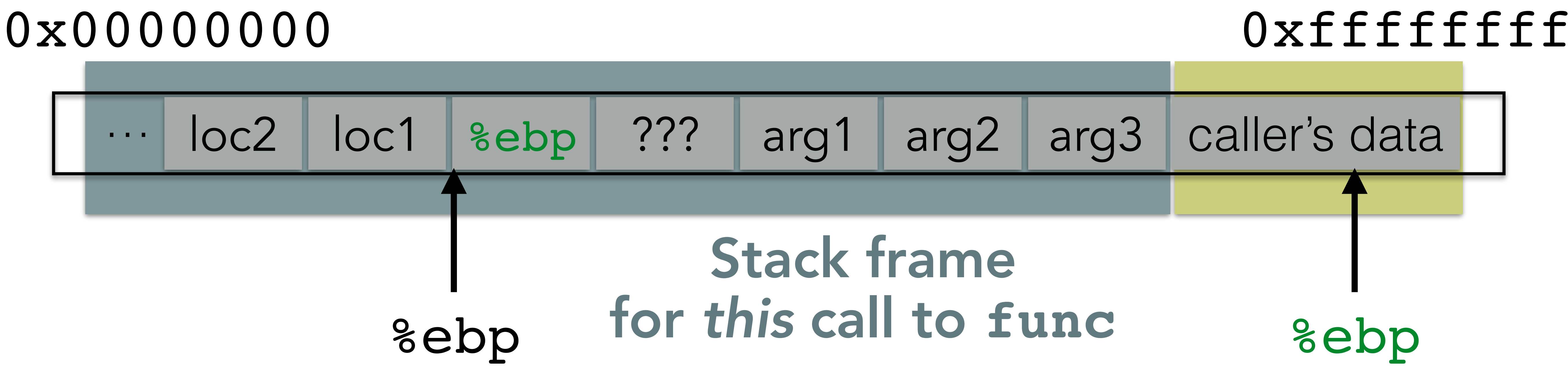
Q: How do we restore %ebp?



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

RETURNING FROM FUNCTIONS

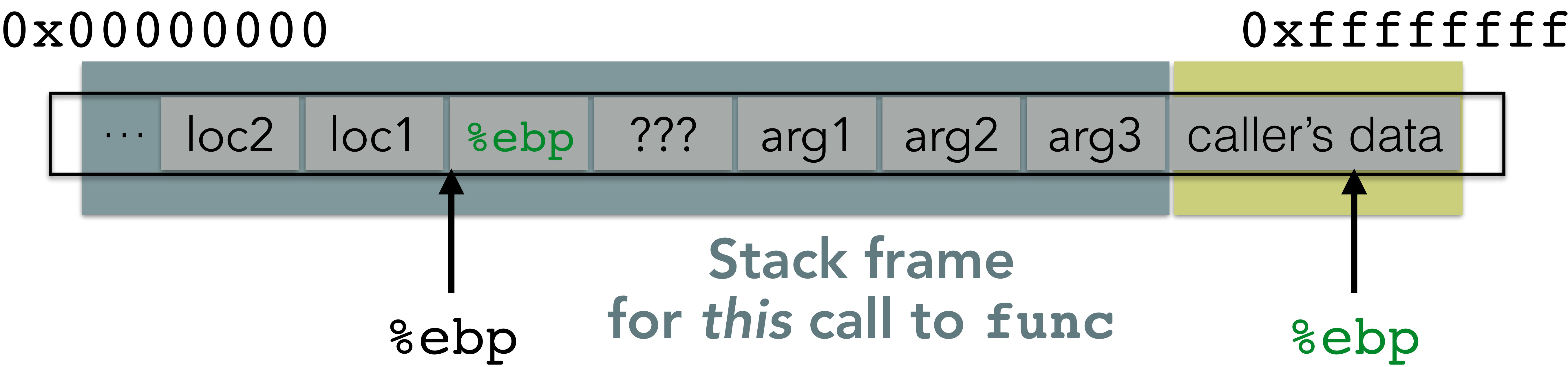
```
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}
```



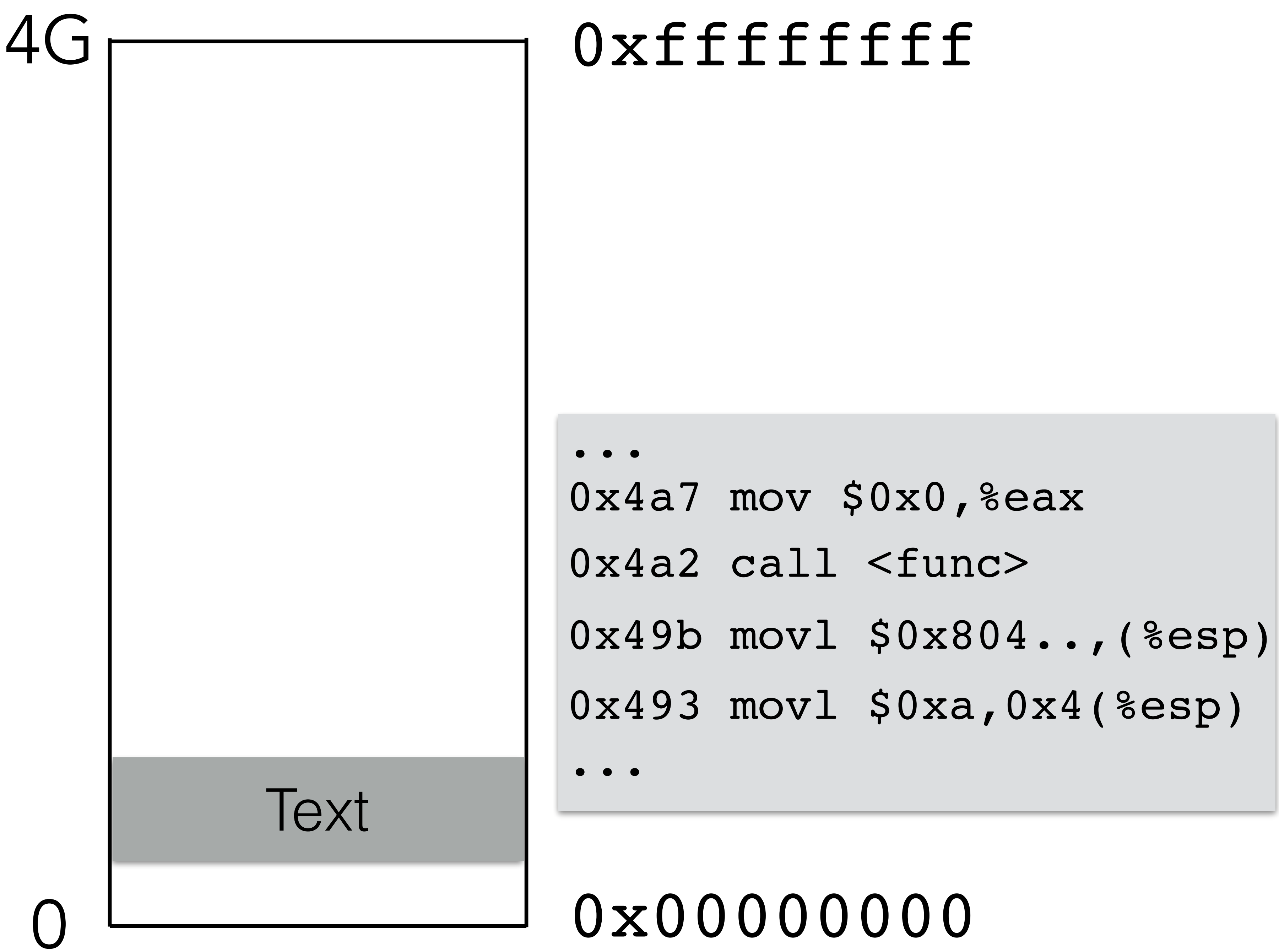
RETURNING FROM FUNCTIONS

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

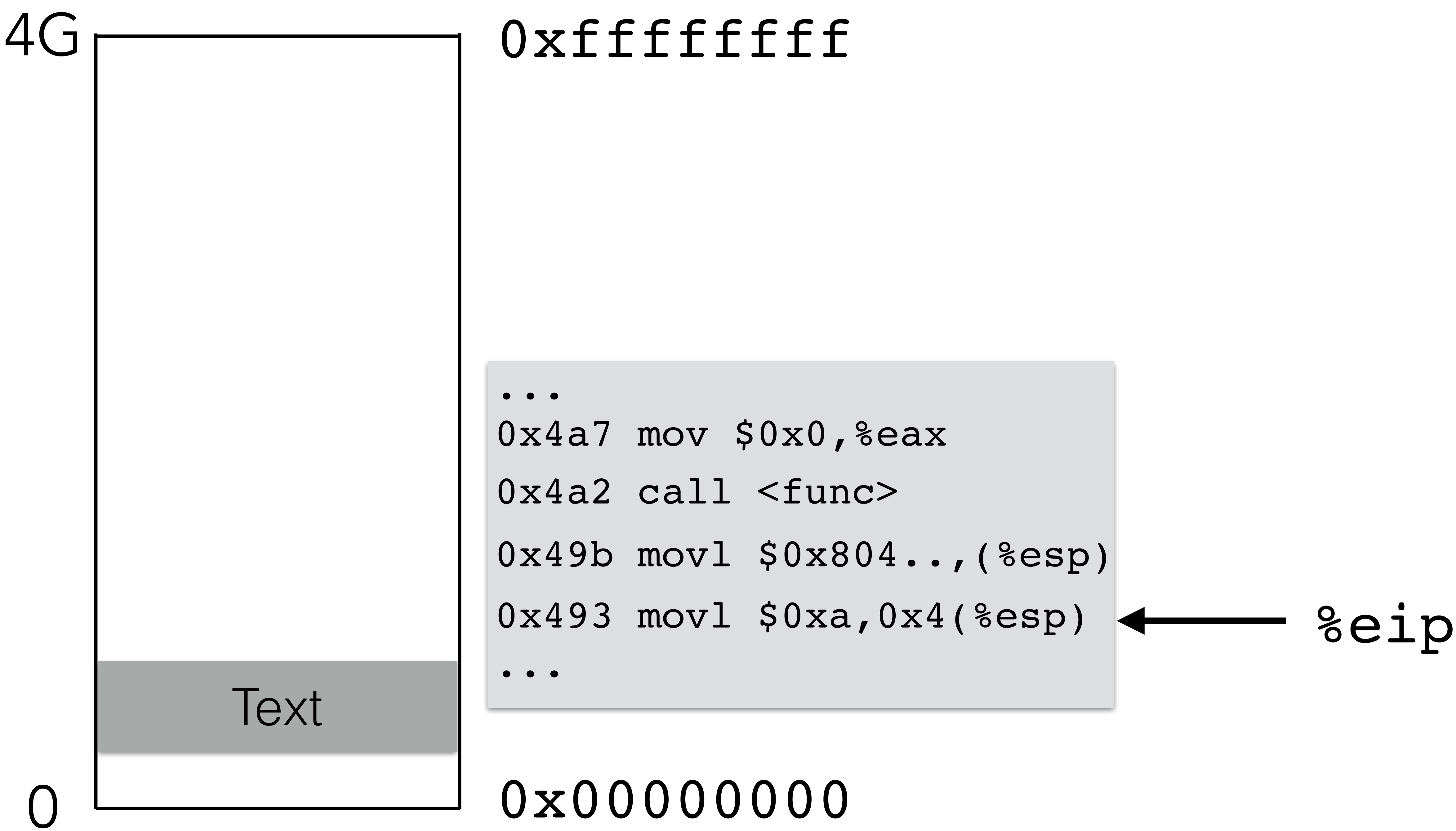
Q: How do we resume here?



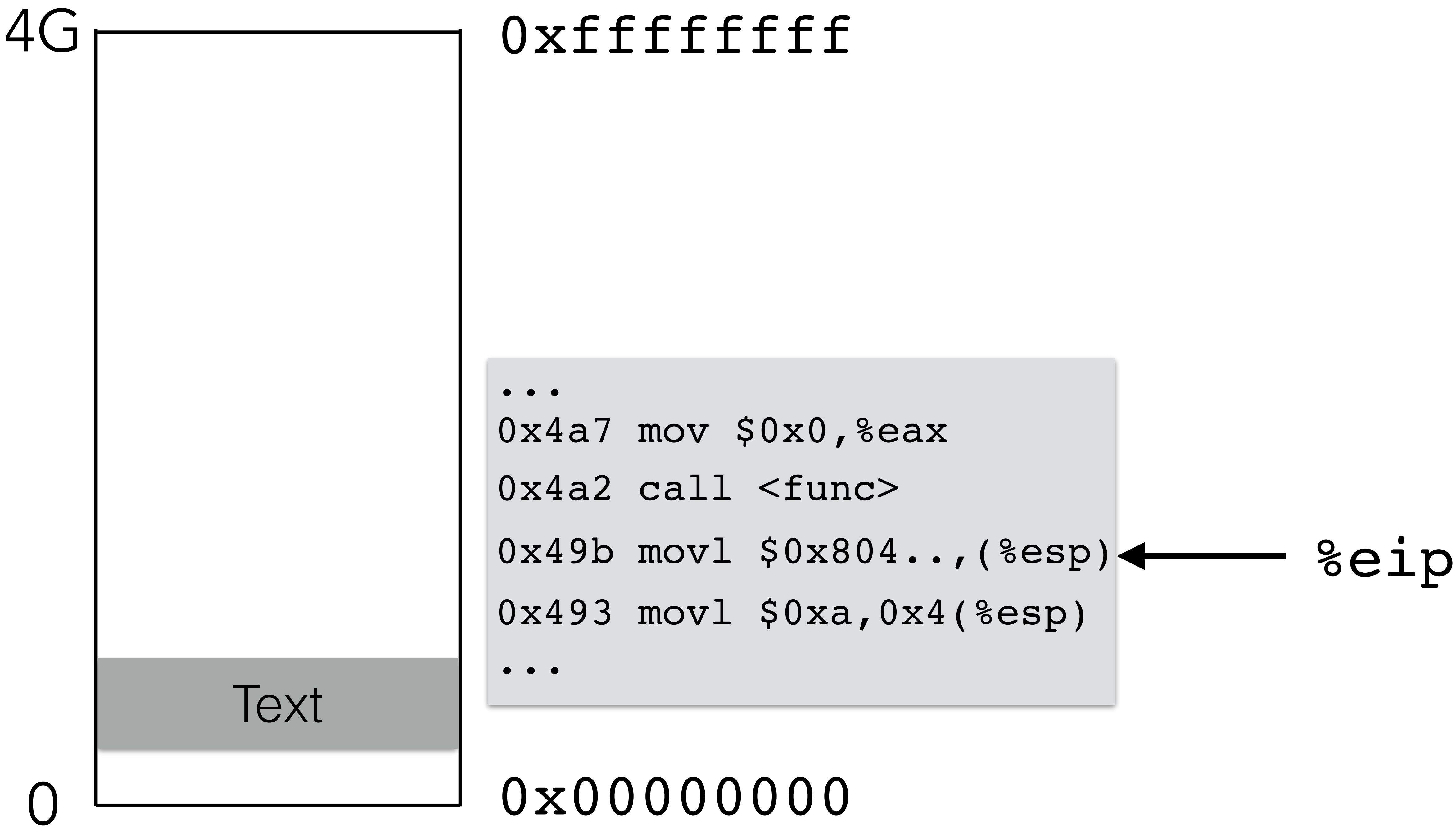
INSTRUCTIONS THEMSELVES ARE IN MEMORY



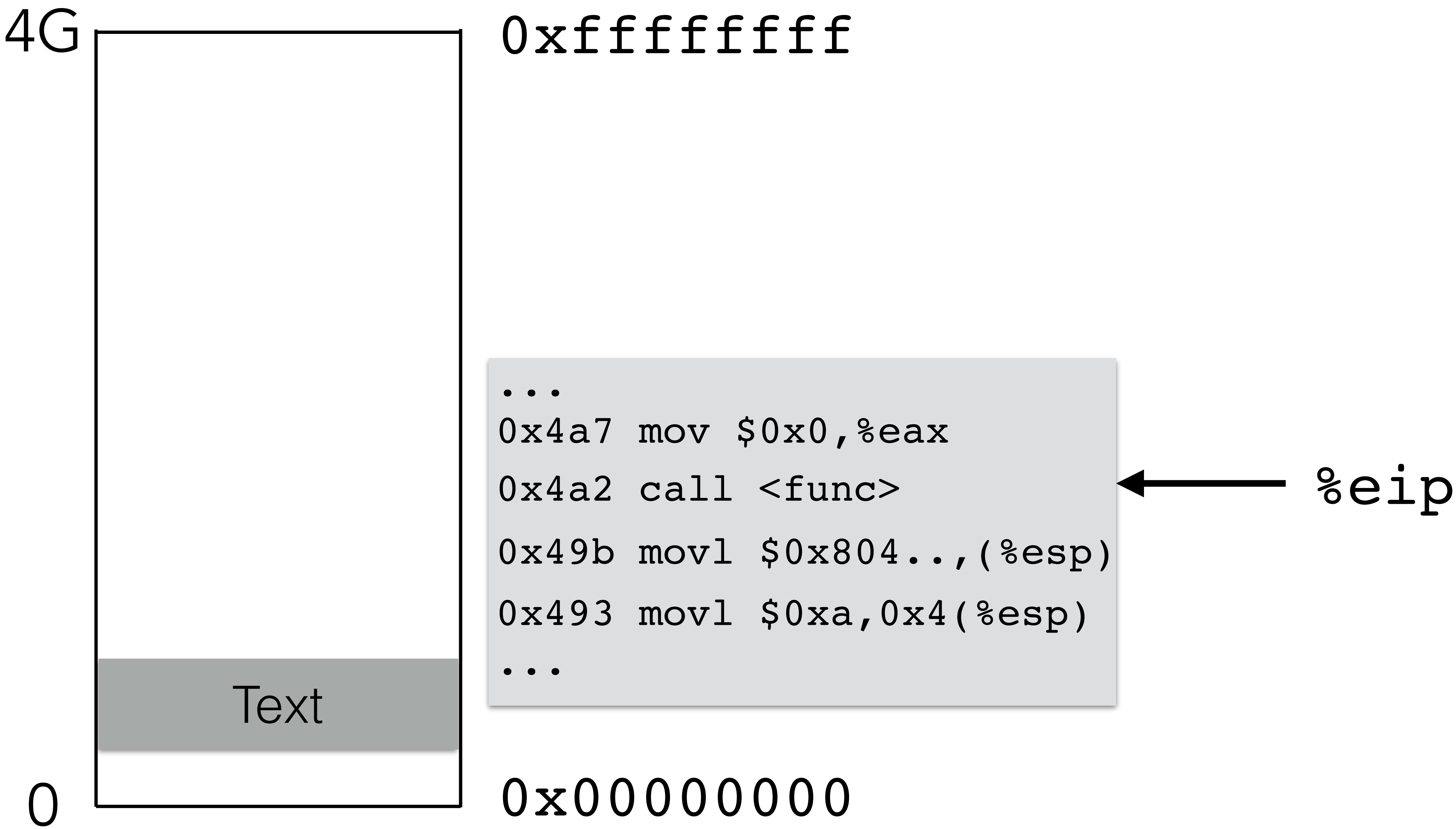
INSTRUCTIONS THEMSELVES ARE IN MEMORY



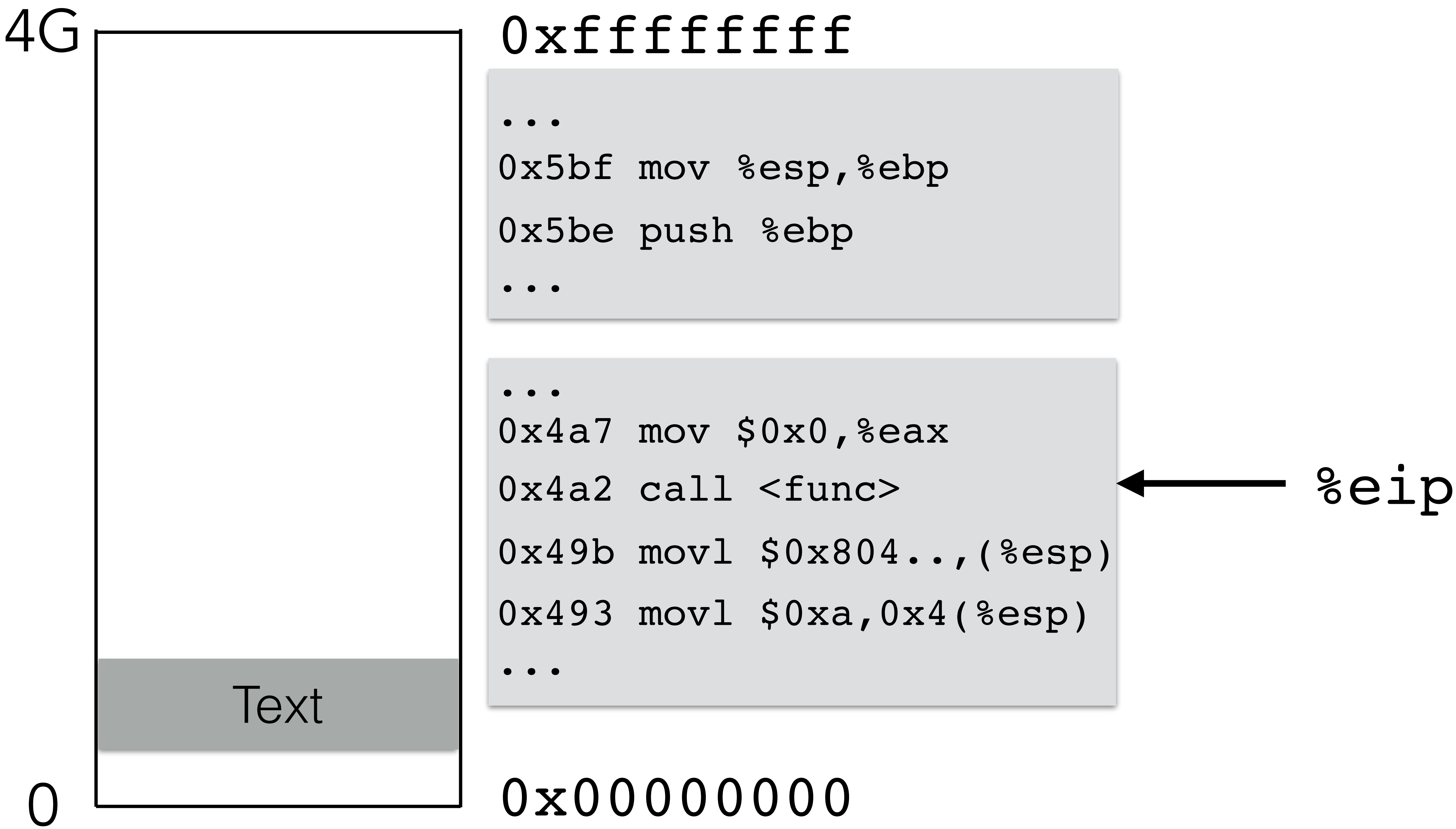
INSTRUCTIONS THEMSELVES ARE IN MEMORY



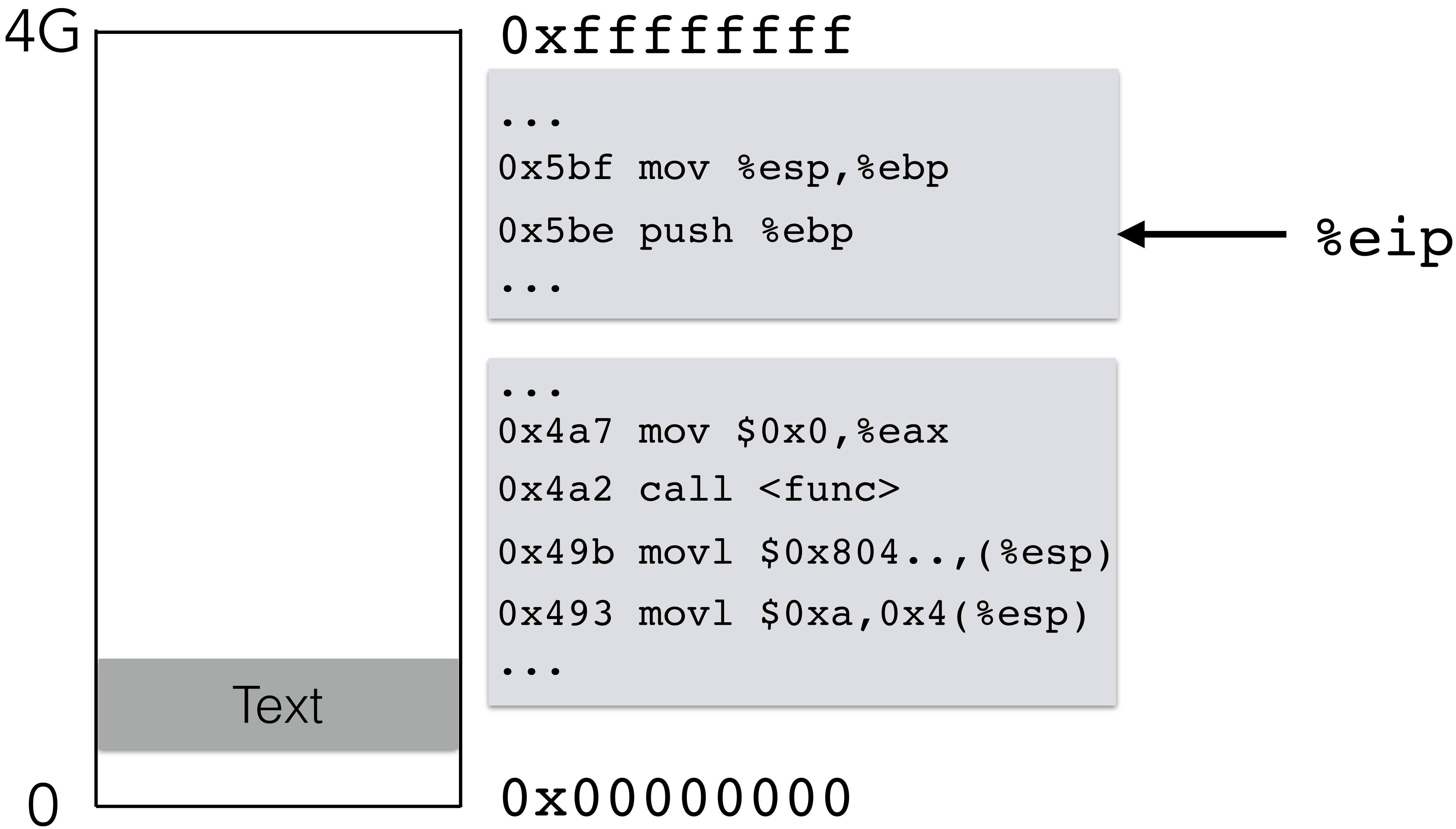
INSTRUCTIONS THEMSELVES ARE IN MEMORY



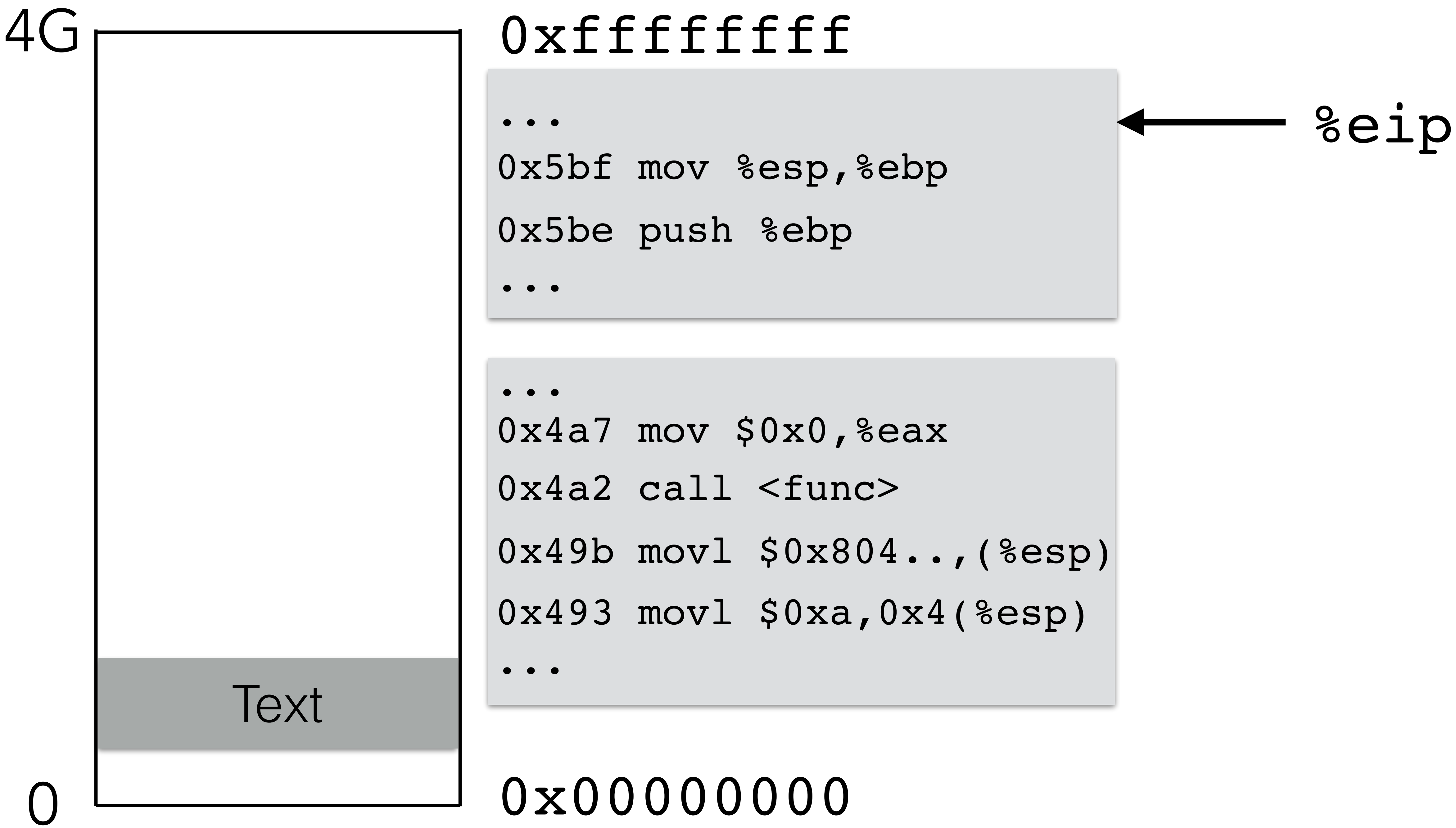
INSTRUCTIONS THEMSELVES ARE IN MEMORY



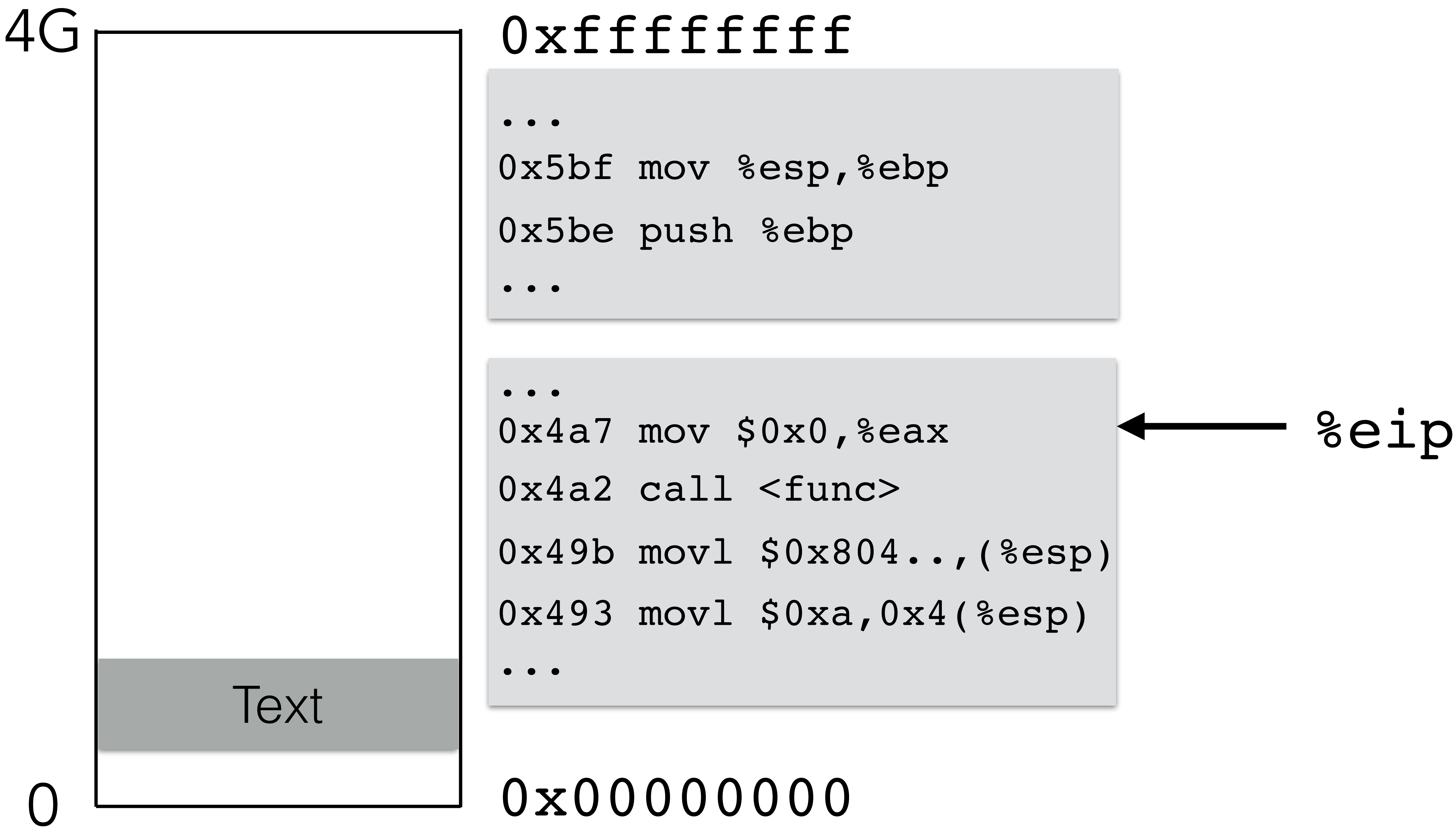
INSTRUCTIONS THEMSELVES ARE IN MEMORY



INSTRUCTIONS THEMSELVES ARE IN MEMORY



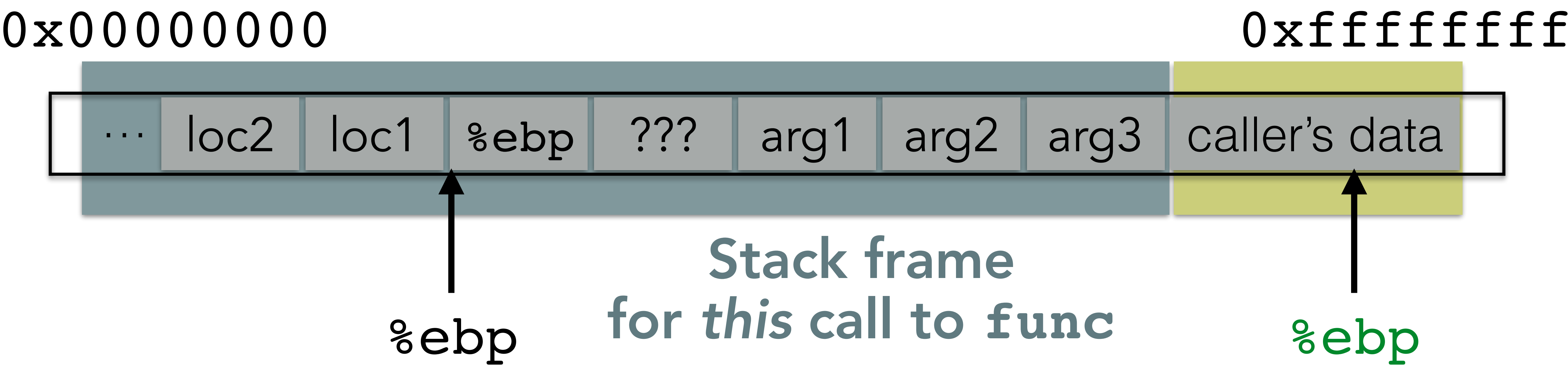
INSTRUCTIONS THEMSELVES ARE IN MEMORY



RETURNING FROM FUNCTIONS

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

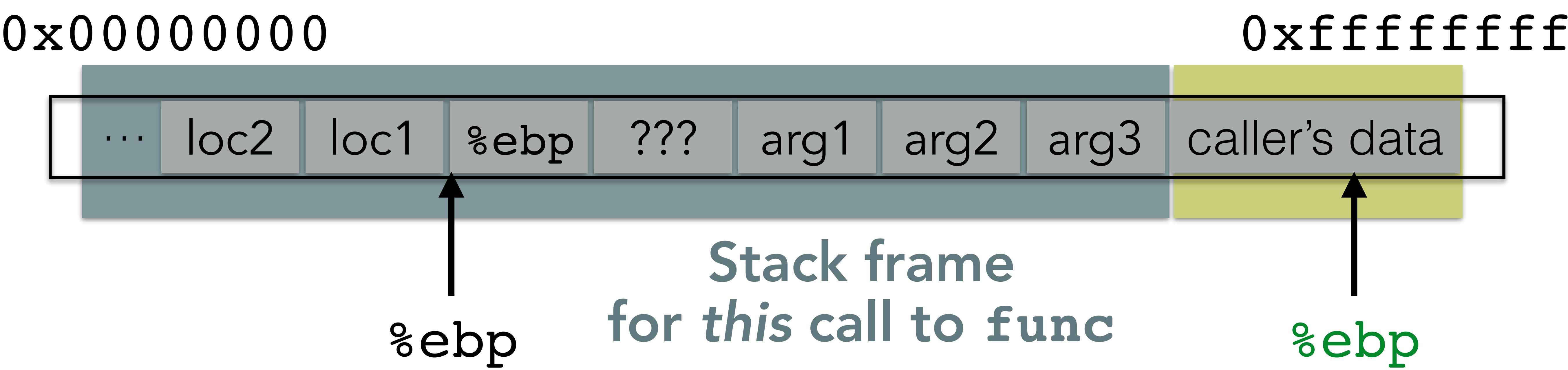
Q: How do we resume here?



RETURNING FROM FUNCTIONS

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

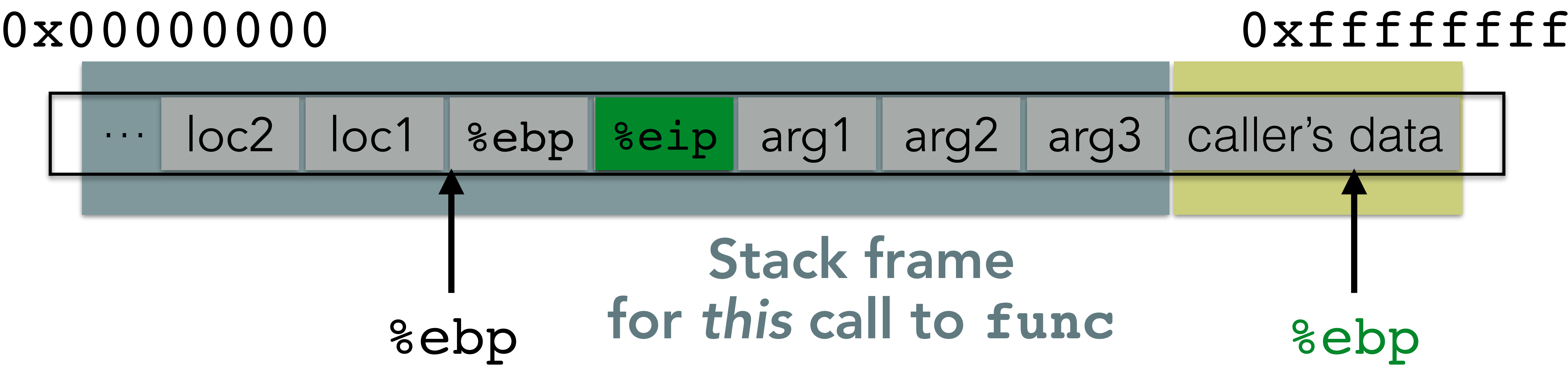
Q: How do we resume here?



Push next %eip before call

RETURNING FROM FUNCTIONS

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

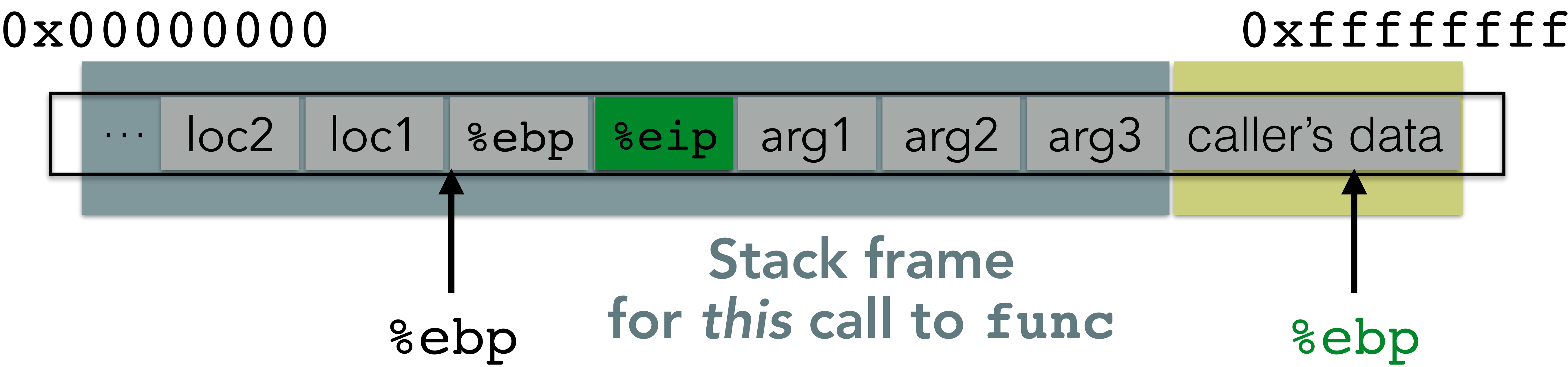


Push next %eip before call

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

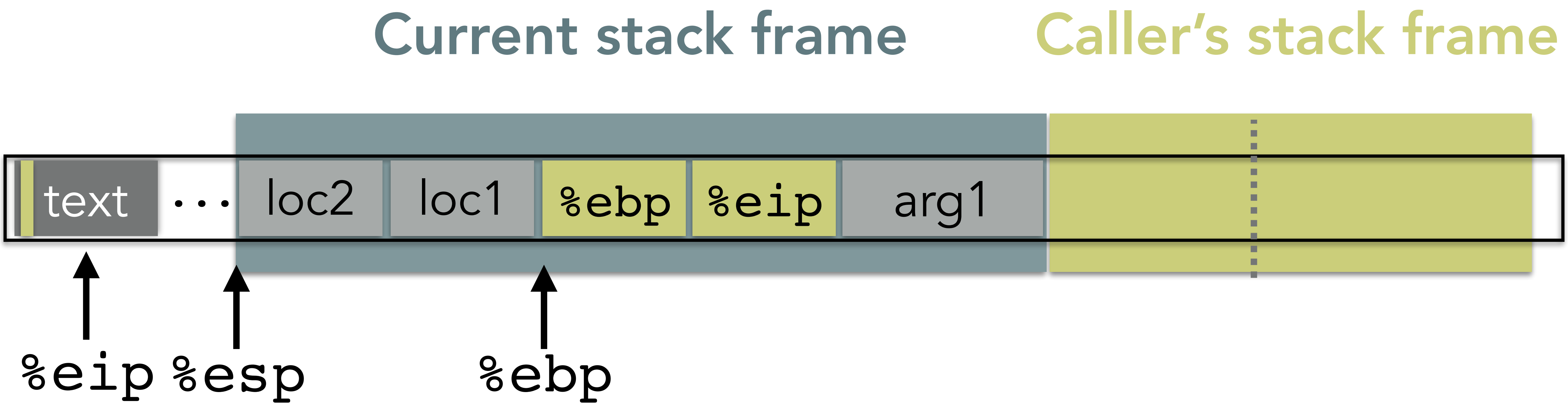
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



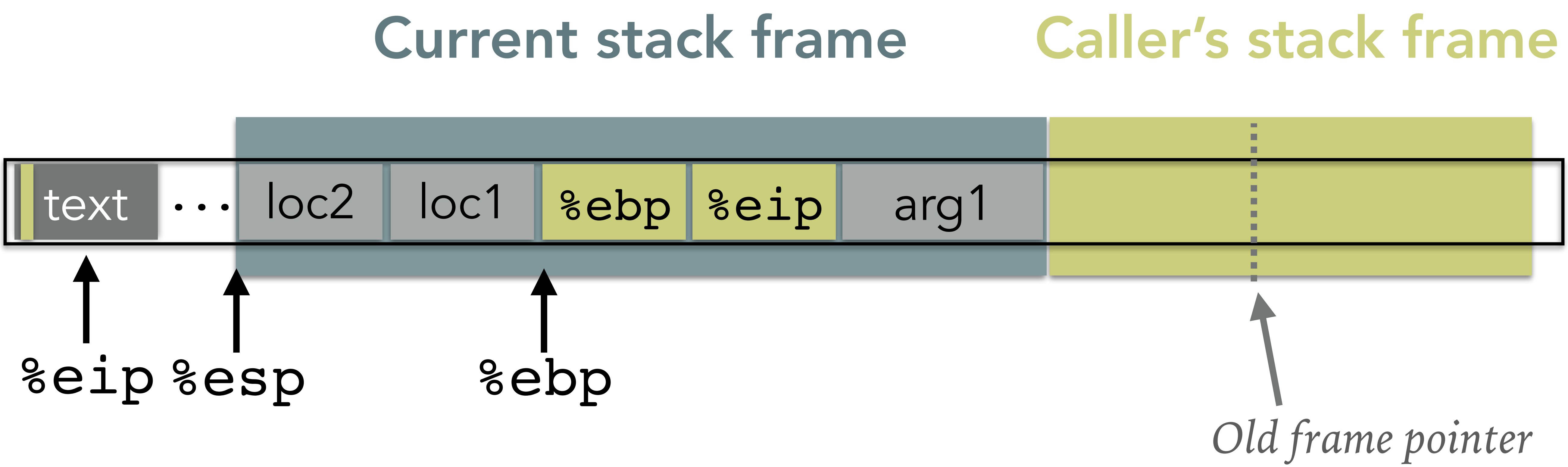
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



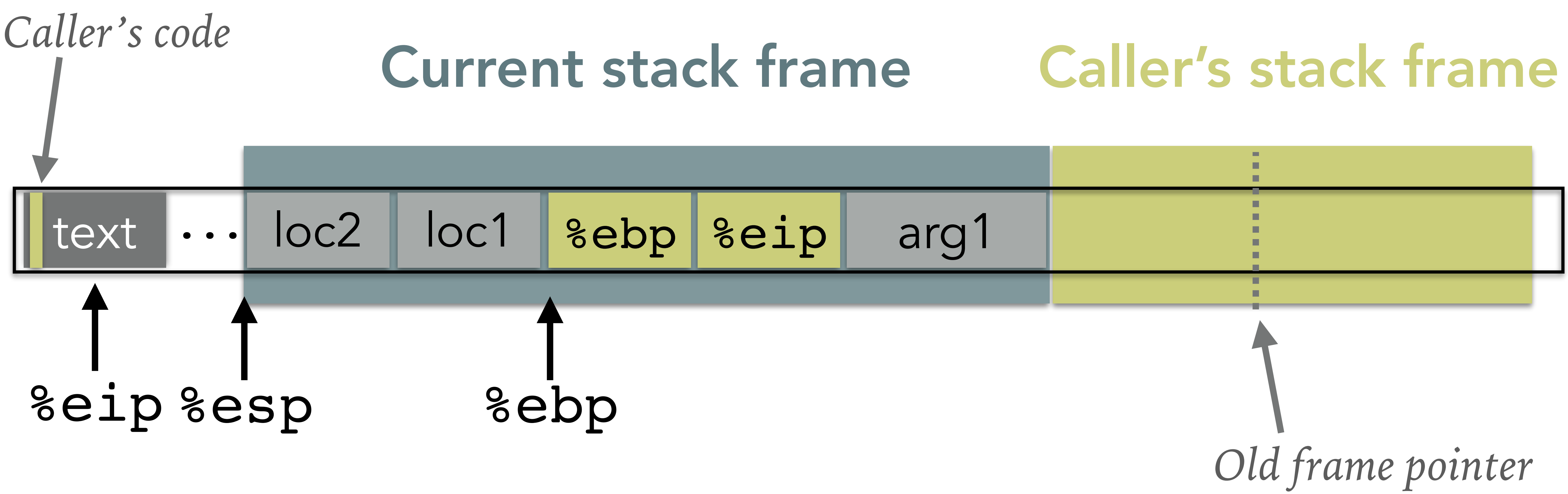
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



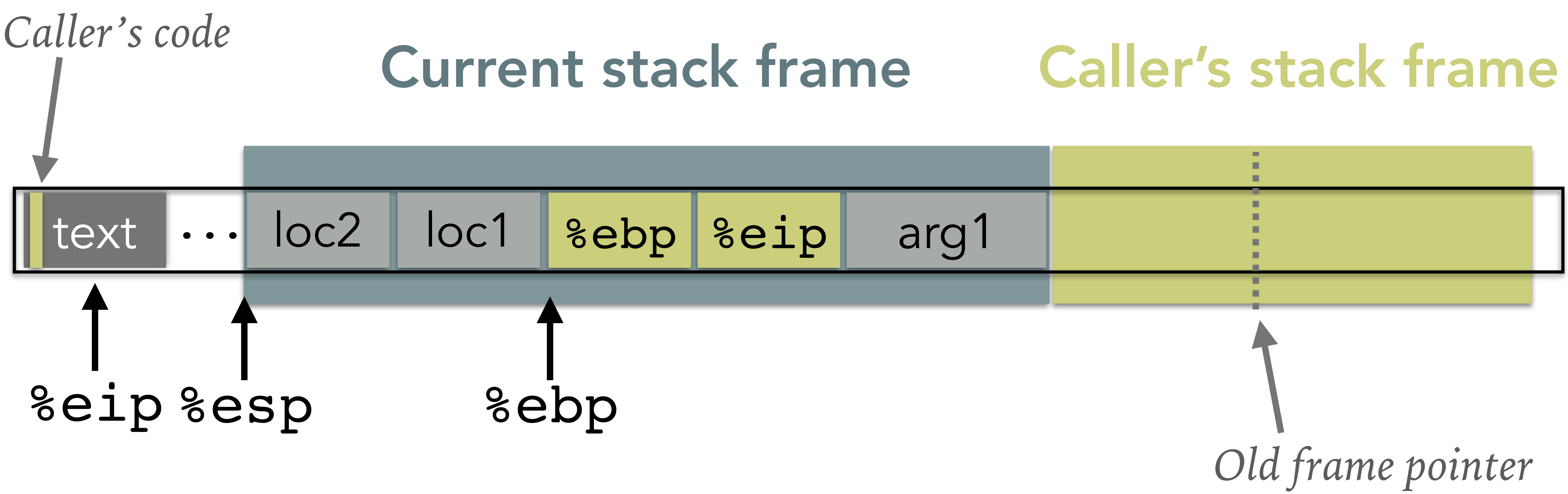
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



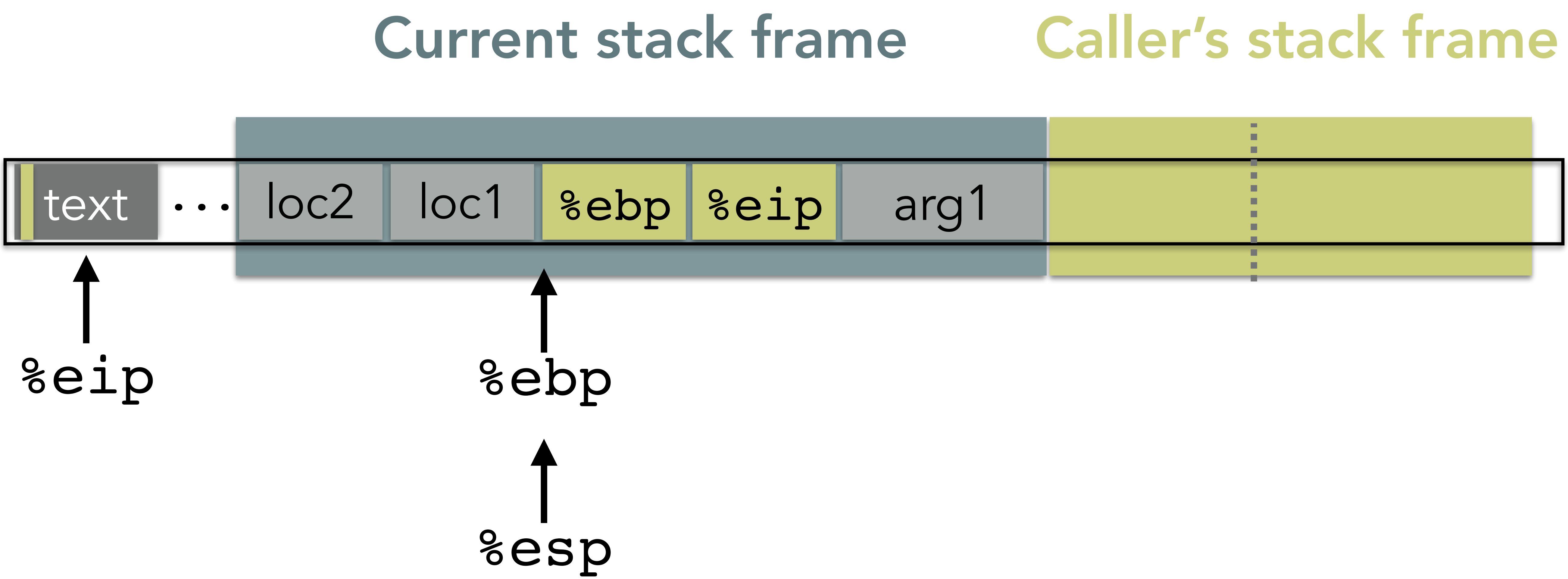
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



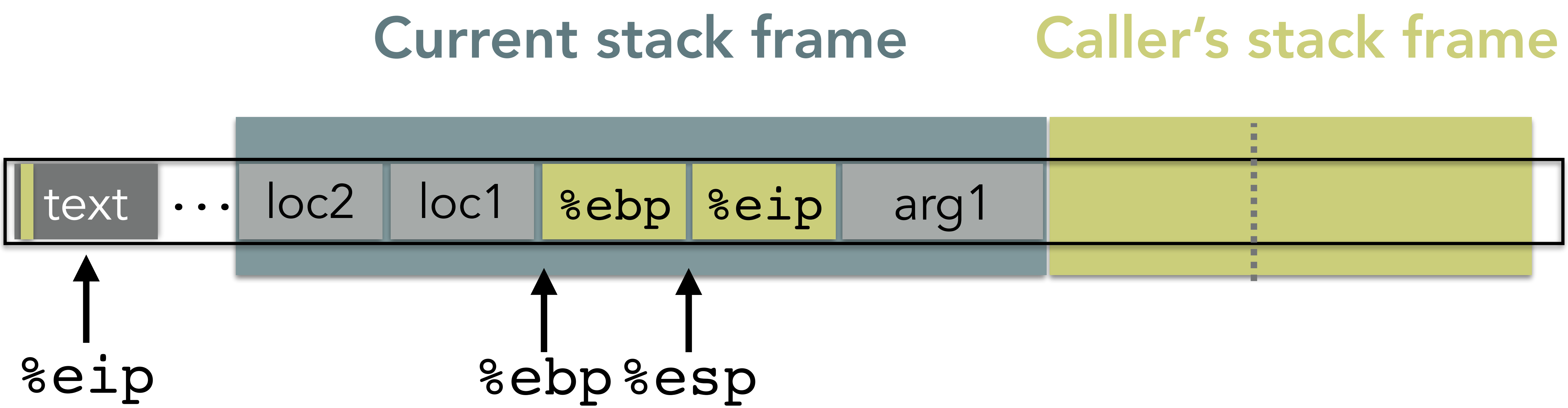
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



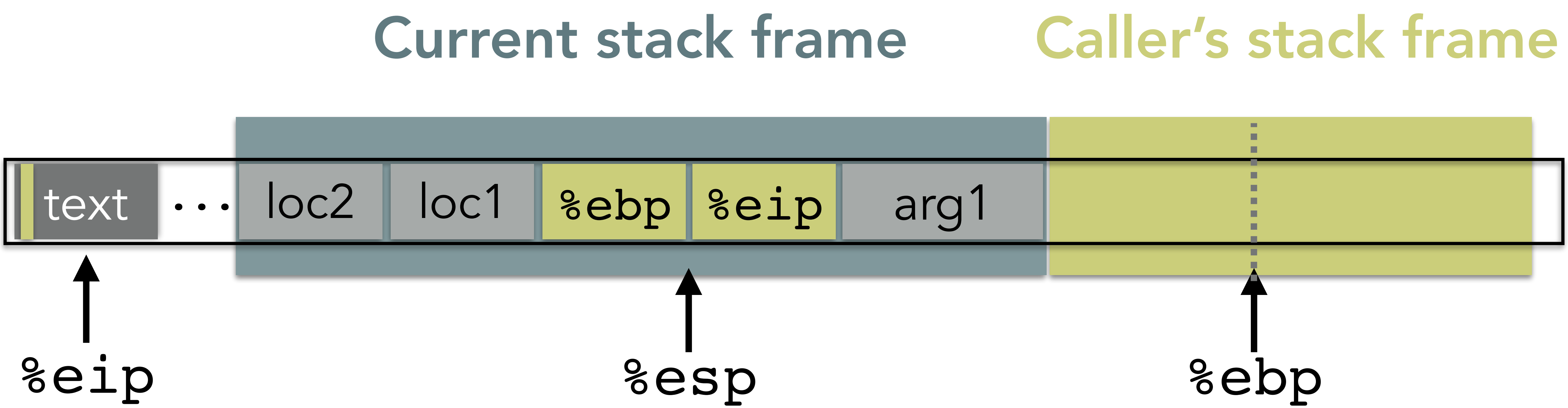
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



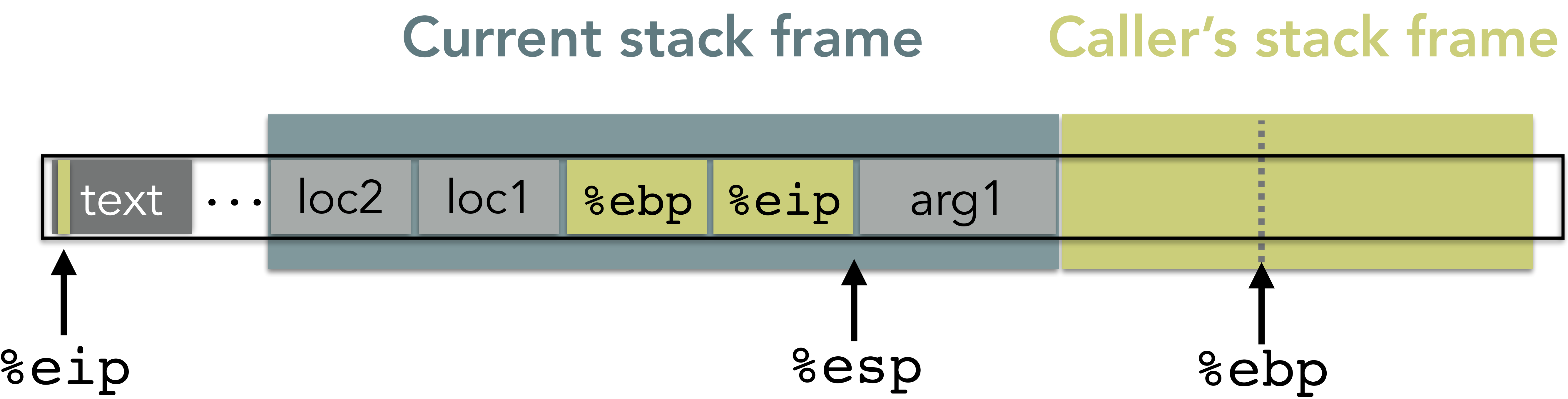
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



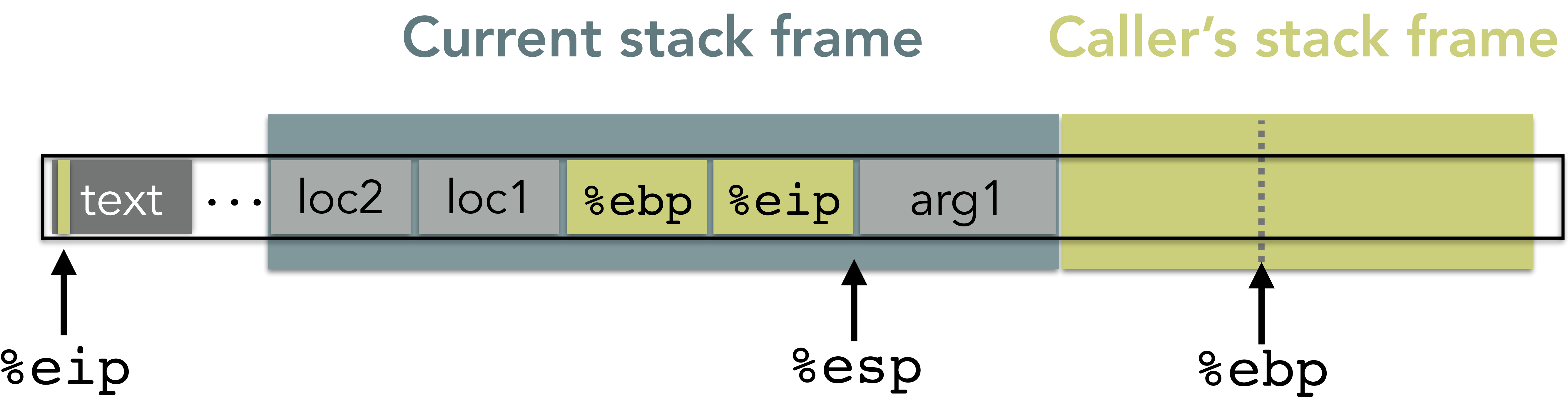
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



The next instruction is to "remove" the arguments off the stack

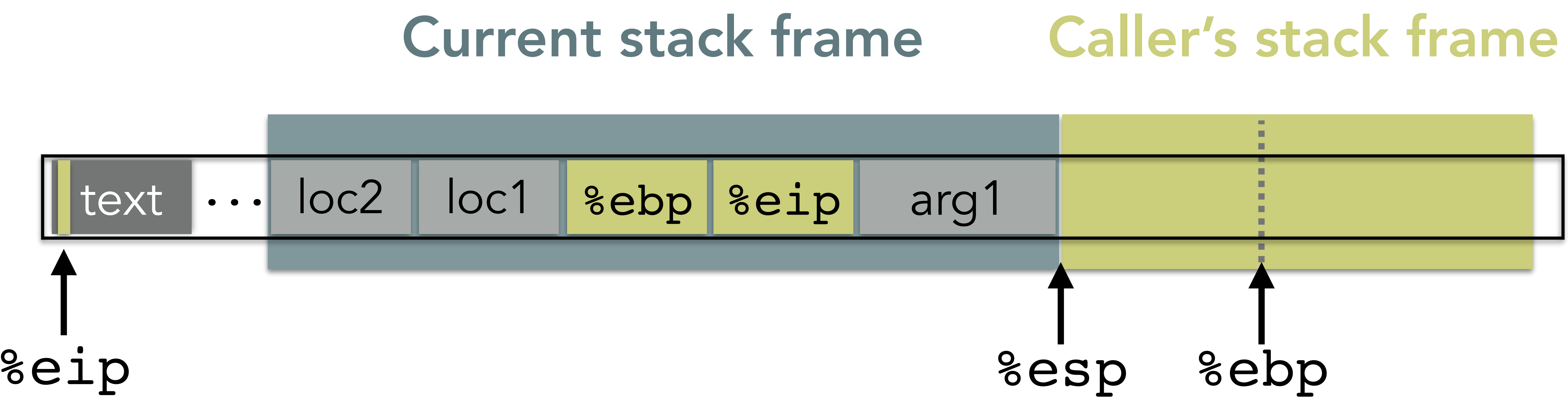
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



The next instruction is to "remove" the arguments off the stack

And now we're back where we started

STACK & FUNCTIONS: SUMMARY

STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

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: e.g., %eip + 2
3. **Jump to the function's address**

STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

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: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

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: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip

STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

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: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip

Calling function (after return):

9. **Remove the arguments** off of the stack: %esp = %esp + *number of bytes of args*

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

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

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

A BUFFER OVERFLOW EXAMPLE

```c
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

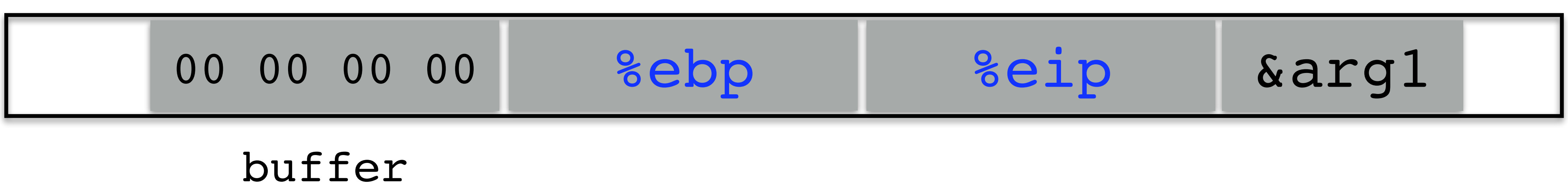
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

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

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

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



A BUFFER OVERFLOW EXAMPLE

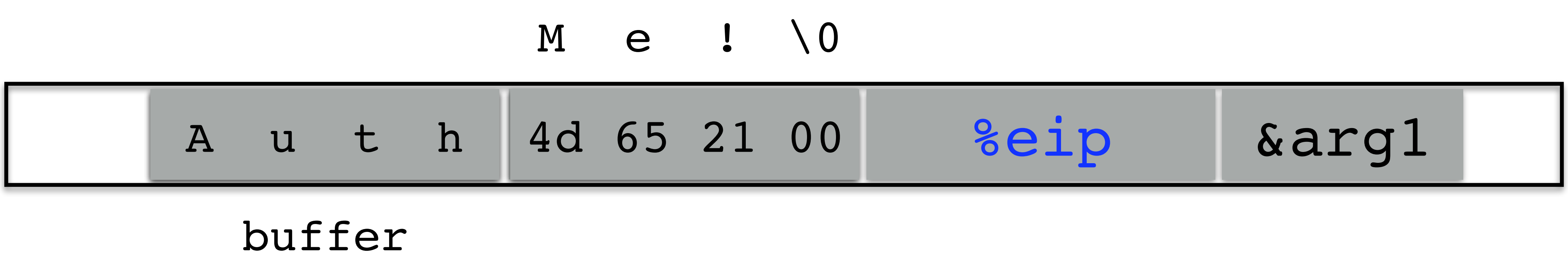
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

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

	A u t h	%ebp	%eip	&arg1	

buffer

A BUFFER OVERFLOW EXAMPLE

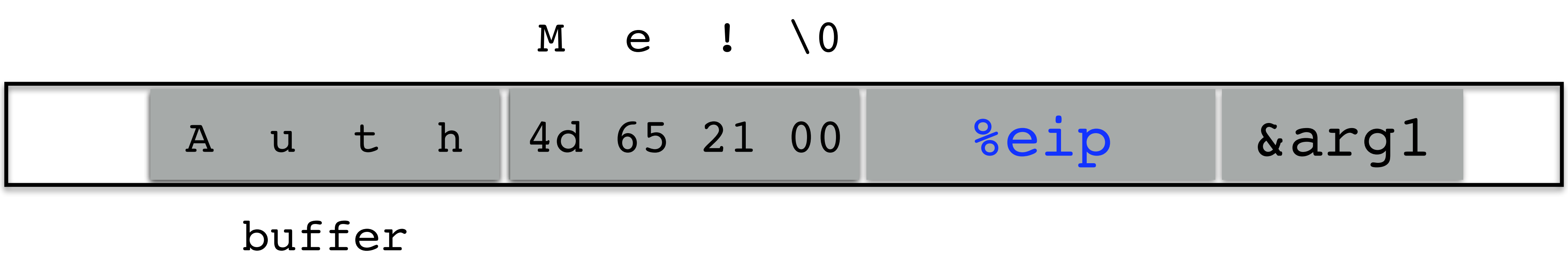
```c
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

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);
    ...
}
```

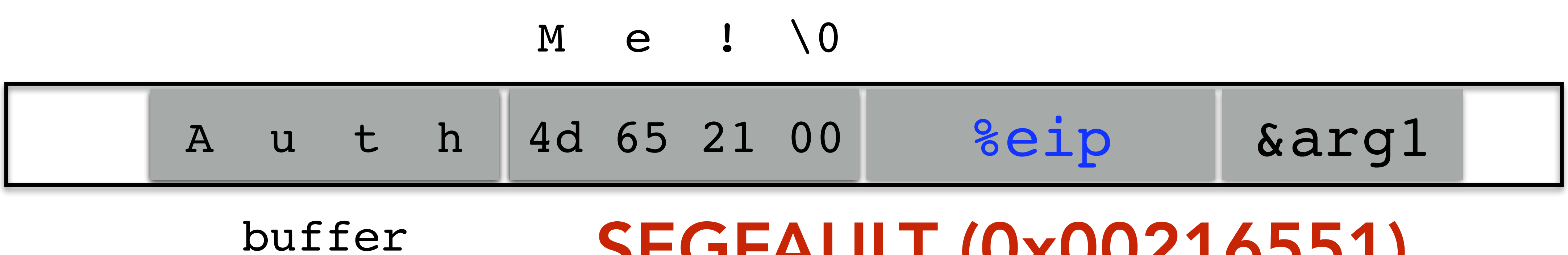
Upon return, sets %ebp to 0x0021654d



A BUFFER OVERFLOW EXAMPLE

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

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

Upon return, sets %ebp to 0x0021654d



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);
    ...
}
```

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

```c
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				

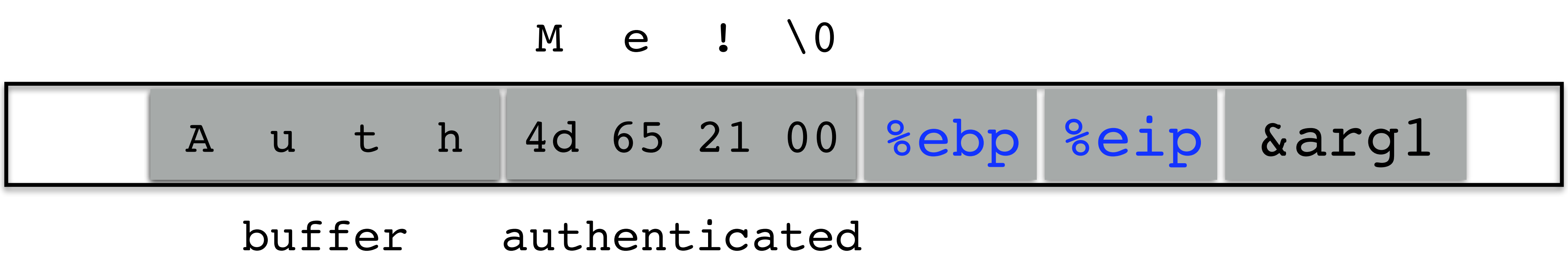
A BUFFER OVERFLOW EXAMPLE

```
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

```c
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

```c
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);
}
```

```c
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

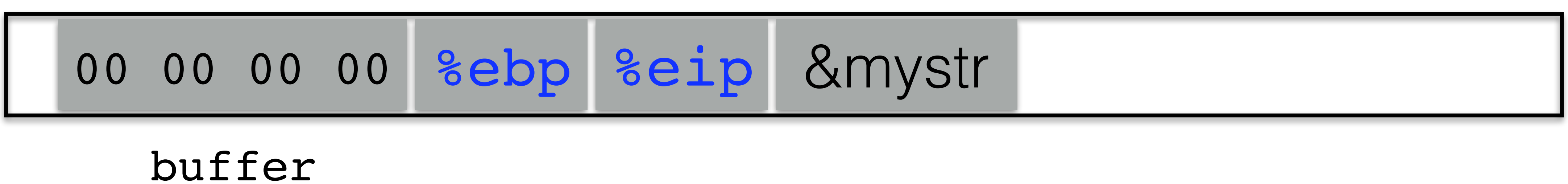
The vulnerability in IE is an invalid pointer reference, Microsoft said in security advisory 979352, which it issued on Thursday. Under certain conditions, the invalid pointer can be accessed after an object is deleted, the advisory states. In specially crafted attacks, like the ones launched against 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…

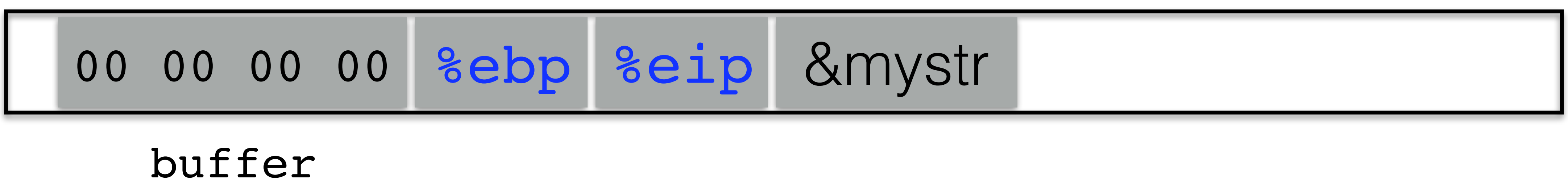
WHAT'S THE WORST THAT CAN HAPPEN?

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



WHAT'S THE WORST THAT CAN HAPPEN?

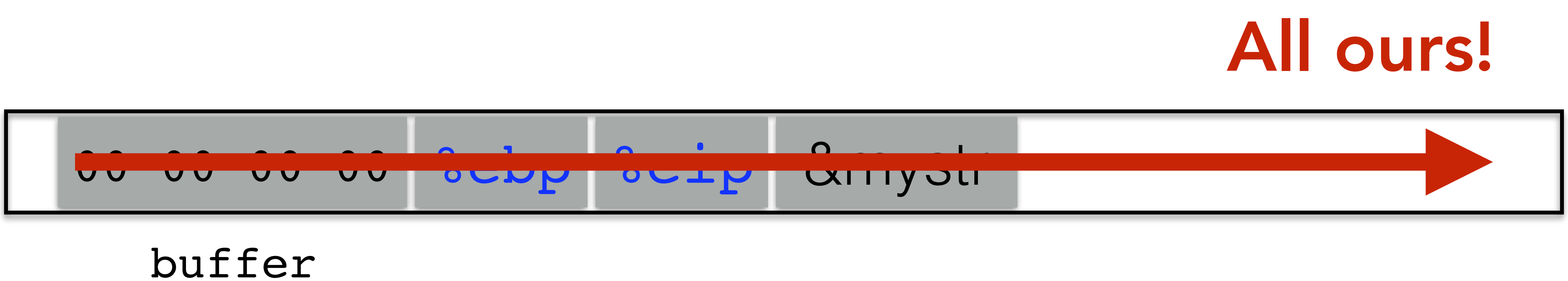
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```



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

WHAT'S THE WORST THAT CAN HAPPEN?

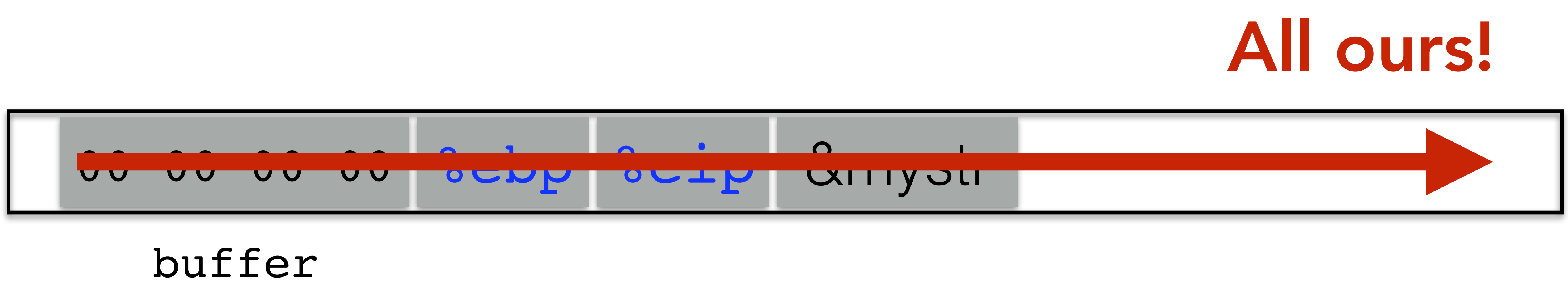
```
void func(char *arg1)
{
     char buffer[4];
     strcpy(buffer, arg1);
     ...
}
```



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

WHAT'S THE WORST THAT CAN HAPPEN?

```c
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

Calling function (before calling):

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: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip

STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

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: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip

STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

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: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip

STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

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: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip

STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

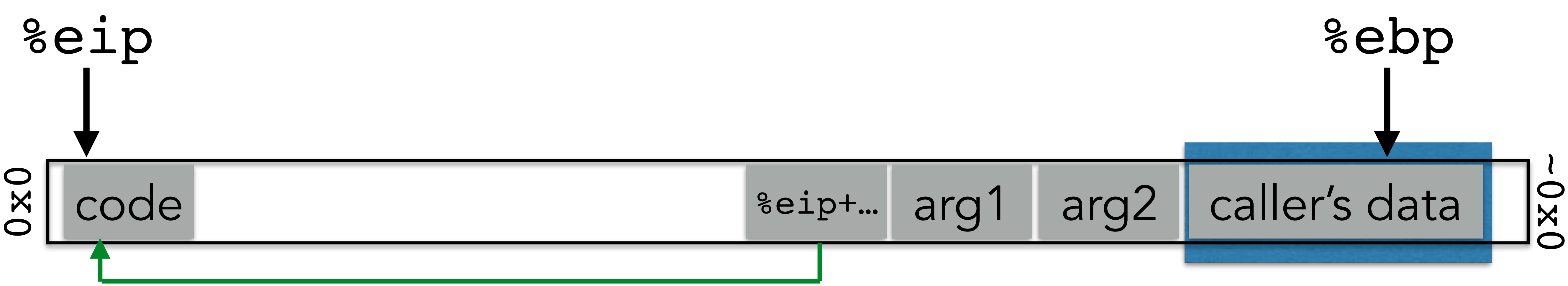
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip



STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

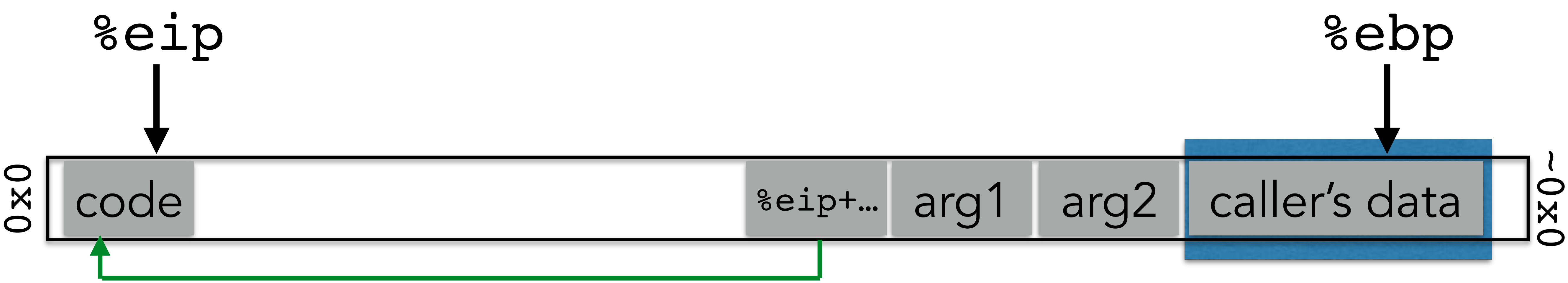
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip



STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

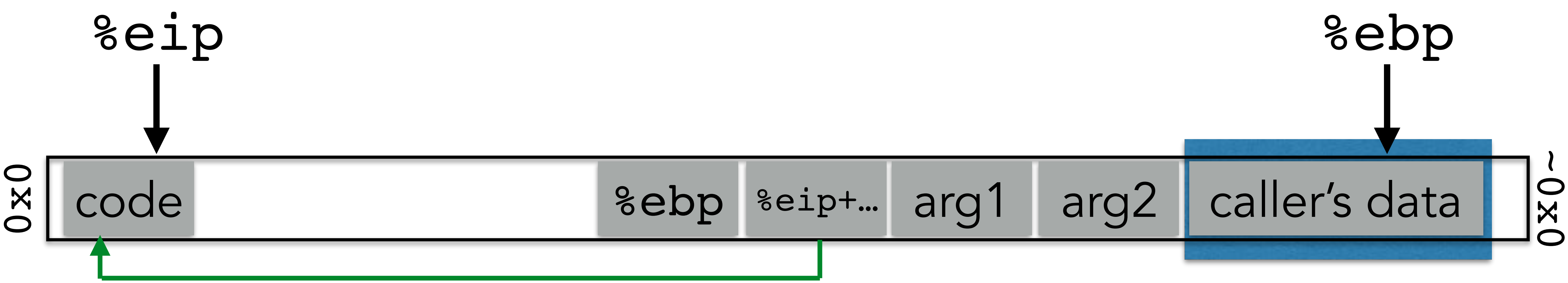
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip



STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

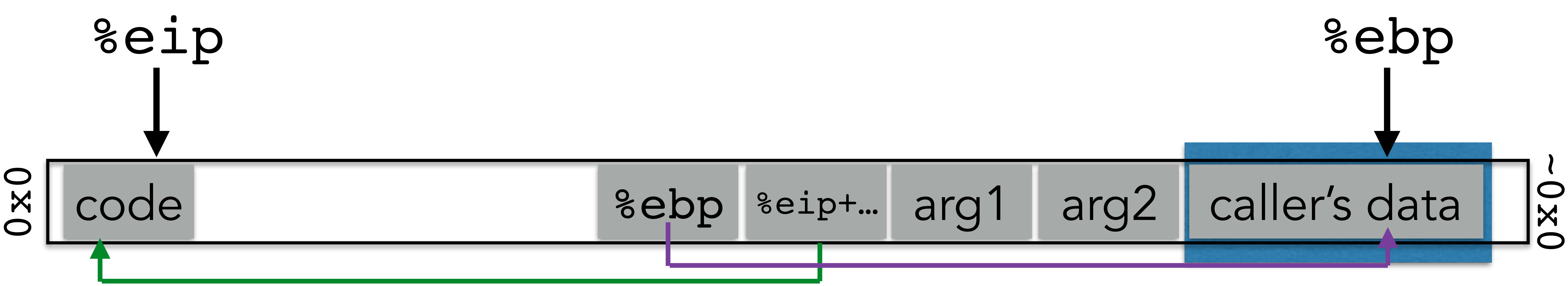
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip



STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

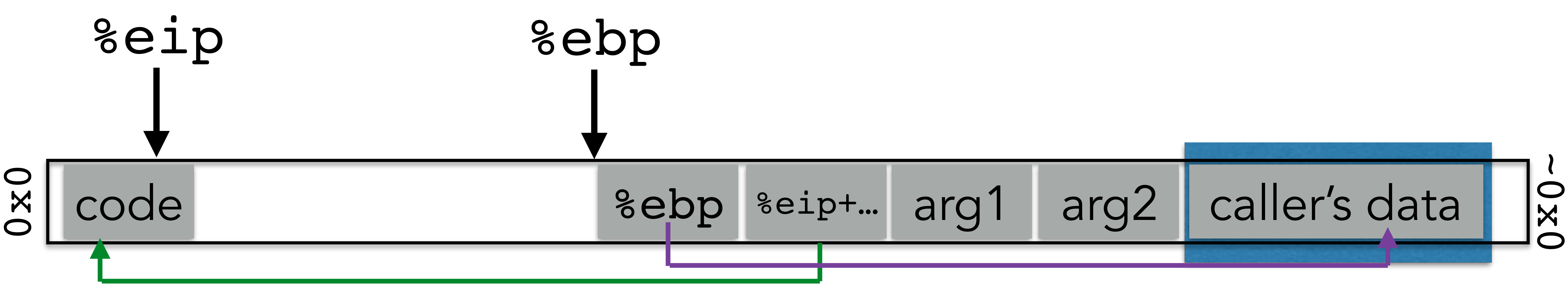
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip



STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

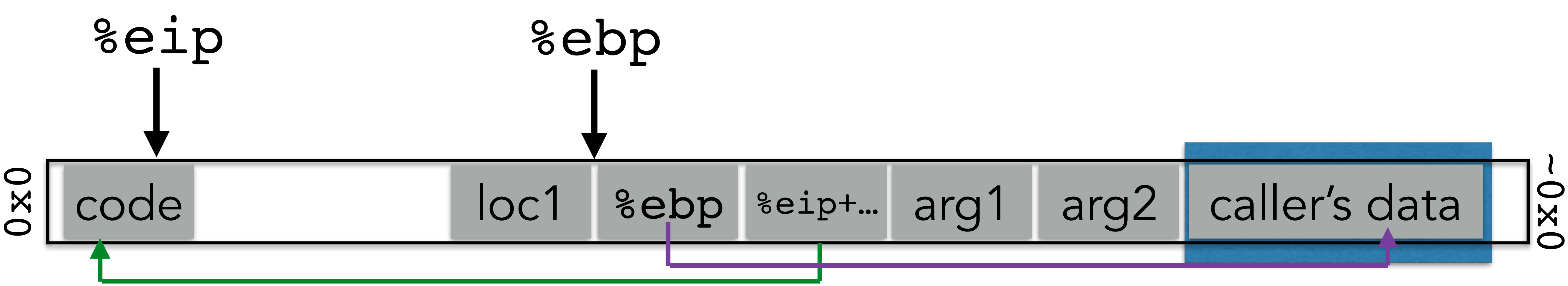
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip



STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

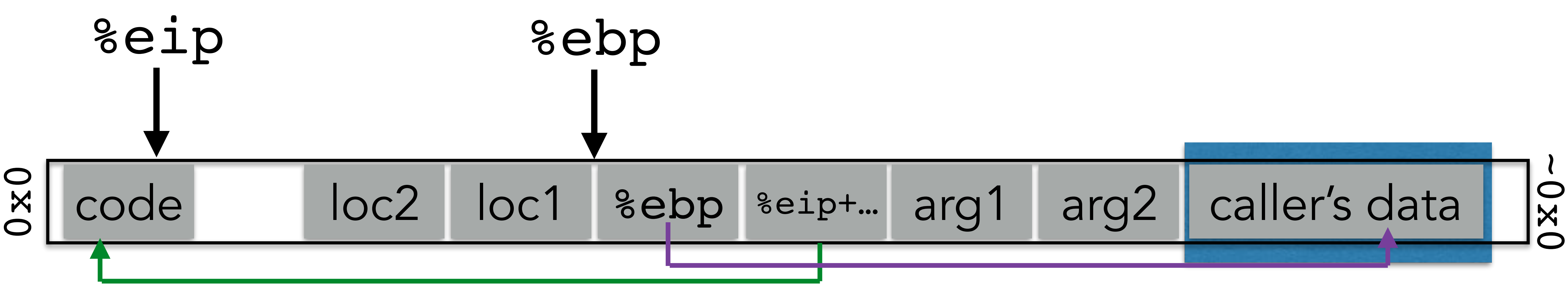
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip



STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

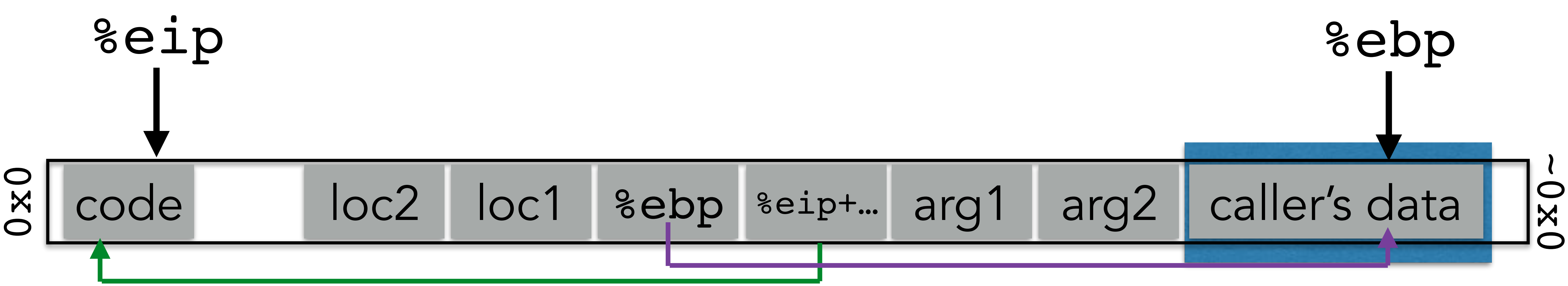
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip



STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

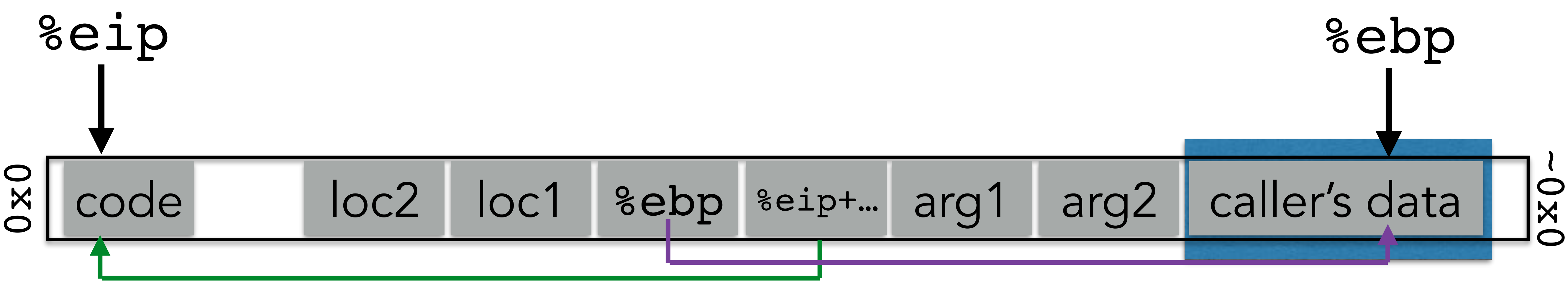
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip



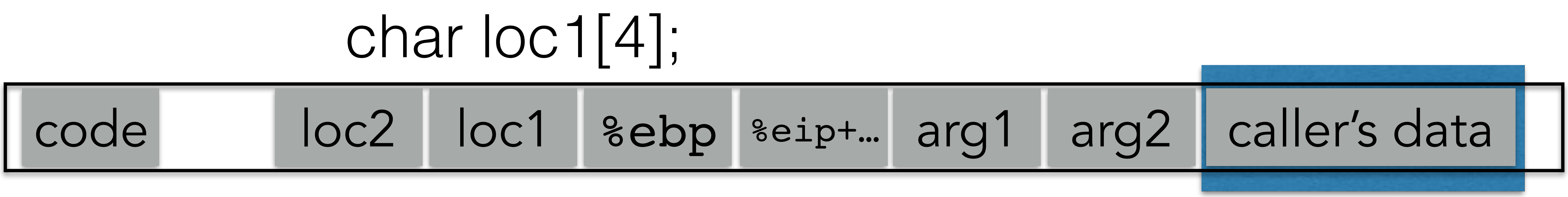
GDB: YOUR NEW BEST FRIEND

`run <input>`	Run the program with `input` as the command-line arguments
`print <var>` (or just "`p <var>`")	Print the value of variable `var` (Can also do some operations: `p &x`)
`b <function>`	Set a breakpoint at `function`
`s` `c`	**s**tep through execution (into calls) **c**ontinue execution (no more stepping)

GDB: YOUR NEW BEST FRIEND

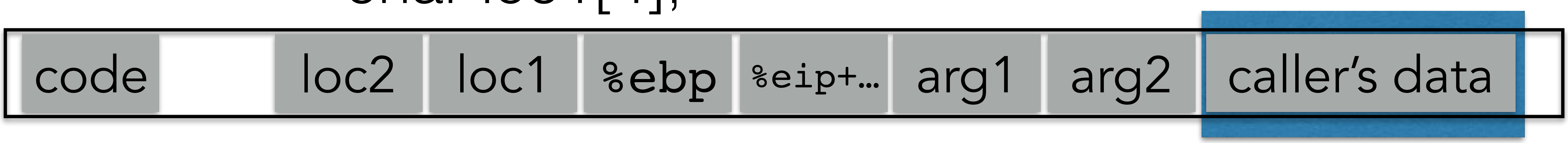
`info frame` (or just "`i f`")	Show **i**nfo about the current **f**rame (prev. frame, locals/args, %ebp/%eip)
`info reg` (or just "`i r`")	Show **info** about **reg**isters (%ebp, %eip, %esp, etc.)
`x/<n> <addr>`	E**x**amine <n> bytes of memory starting at address <addr>

BUFFER OVERFLOW



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!")

char loc1[4];


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

etc.

BUFFER OVERFLOW

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

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

char loc1[4];


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

CODE
INJECTION

HIGH-LEVEL IDEA

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

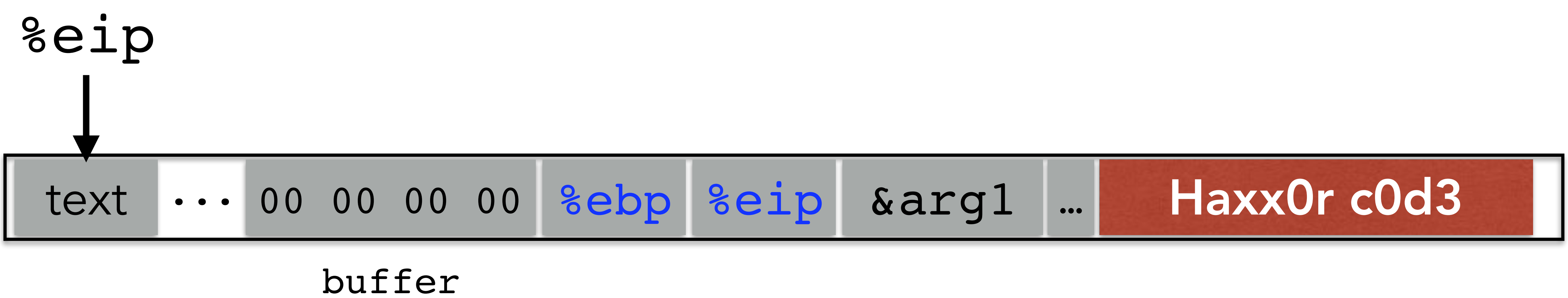
HIGH-LEVEL IDEA

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

(1) Load our own code into memory

HIGH-LEVEL IDEA

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

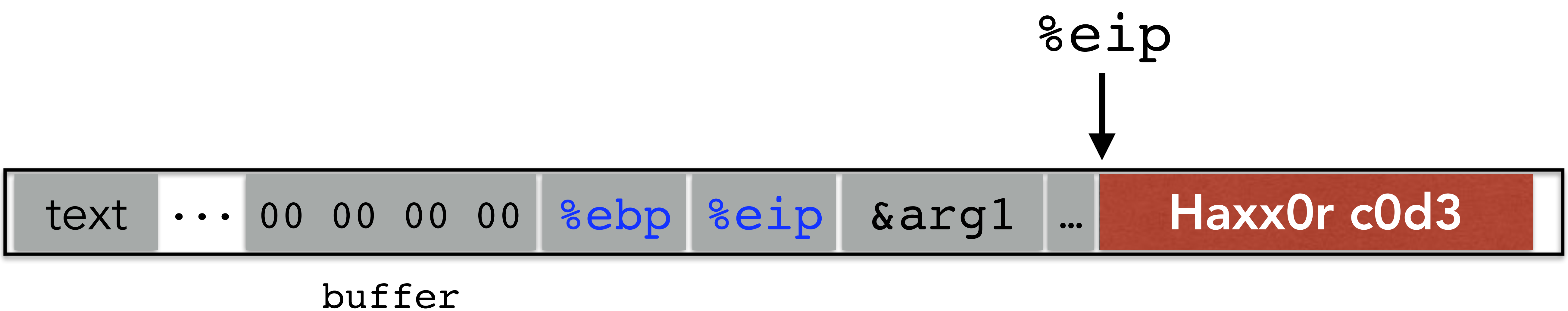


(1) Load our own code into memory

(2) Somehow get %eip to point to it

HIGH-LEVEL IDEA

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

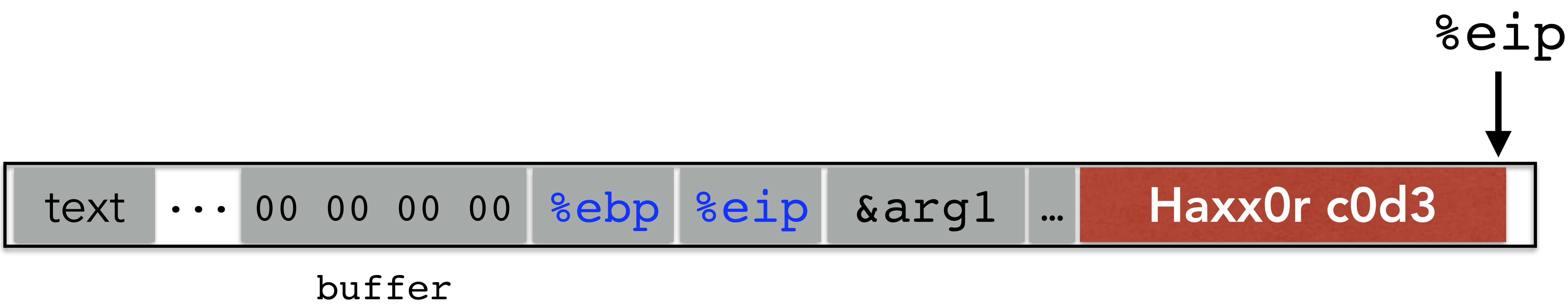


(1) Load our own code into memory

(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

```
#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

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
...
```

```
“\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

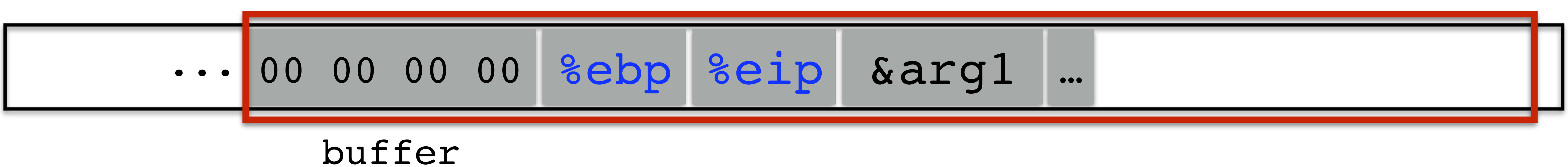
CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

- ***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

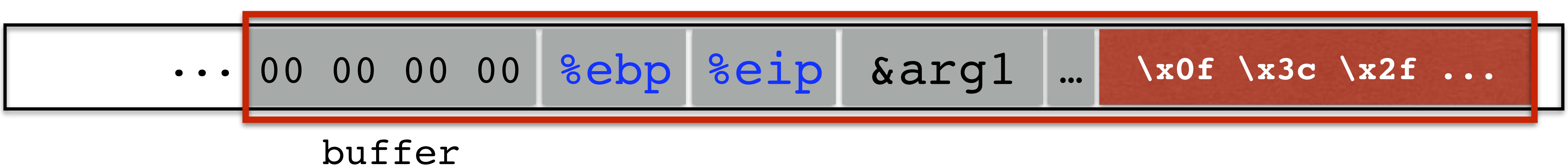
- ***All we can do is write to memory from buffer onward***
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running



Thoughts?

CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

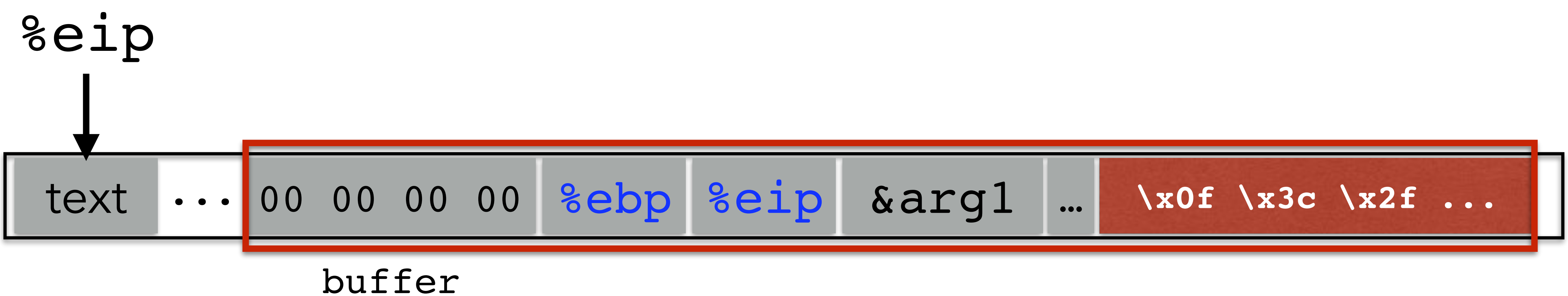
- ***All we can do is write to memory from buffer onward***
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running



Thoughts?

CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

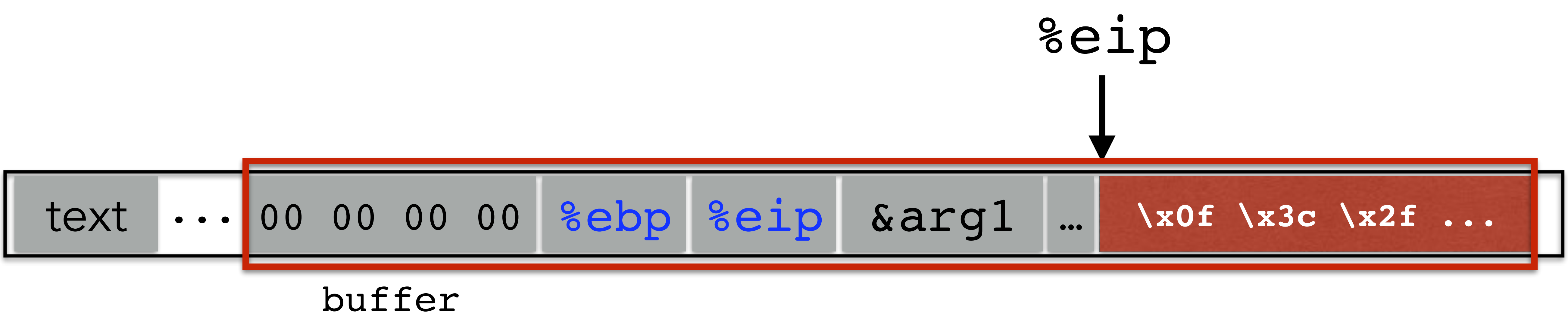
- ***All we can do is write to memory from buffer onward***
    - With this alone we want to get it to jump to our code
    - We have to use whatever code is already running



Thoughts?

CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

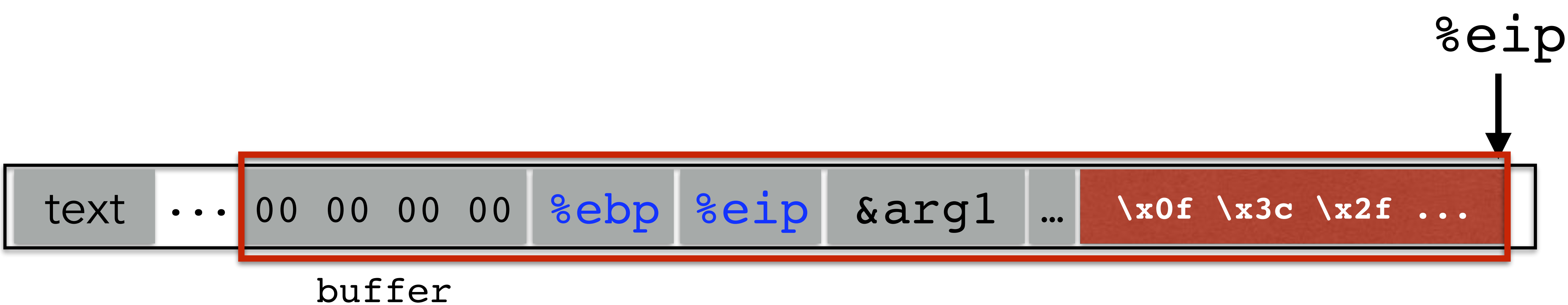
- ***All we can do is write to memory from buffer onward***
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running



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 (before calling):

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: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

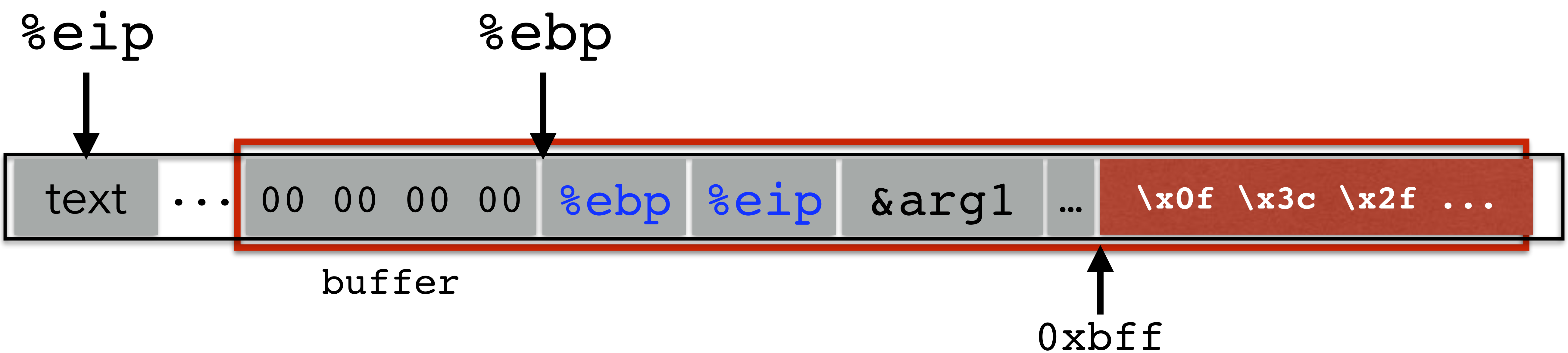
Called function (when returning):

7. **Reset the previous stack frame**: %esp = %ebp; pop %ebp
8. **Jump back to return address**: pop %eip

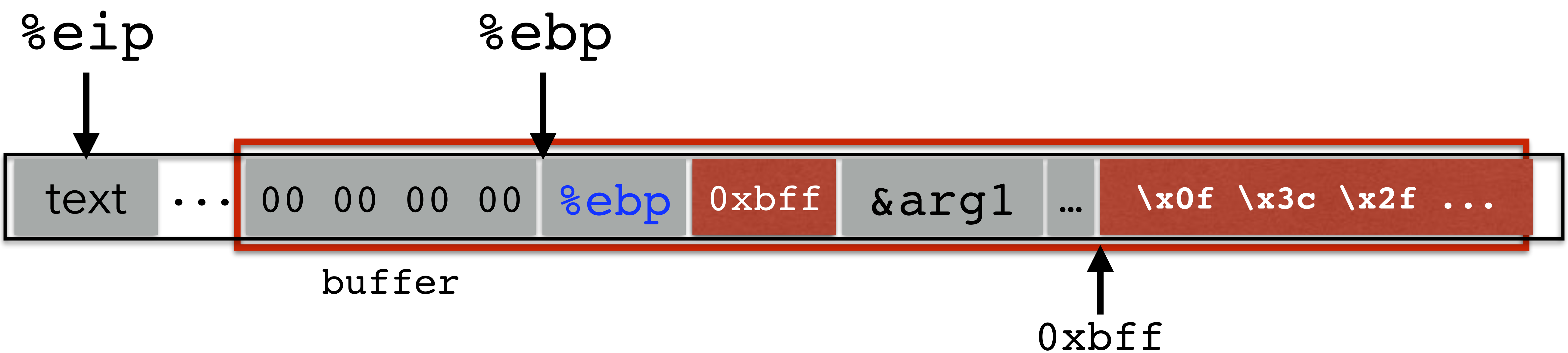
Calling function (after return):

9. **Remove the arguments** off of the stack: %esp = %esp + *number of bytes of args*

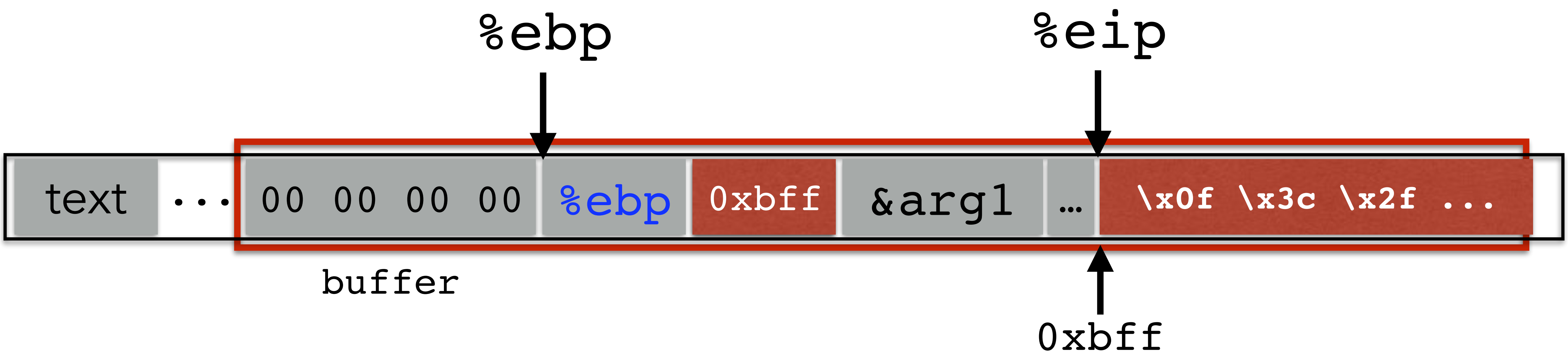
HIJACKING THE SAVED %EIP



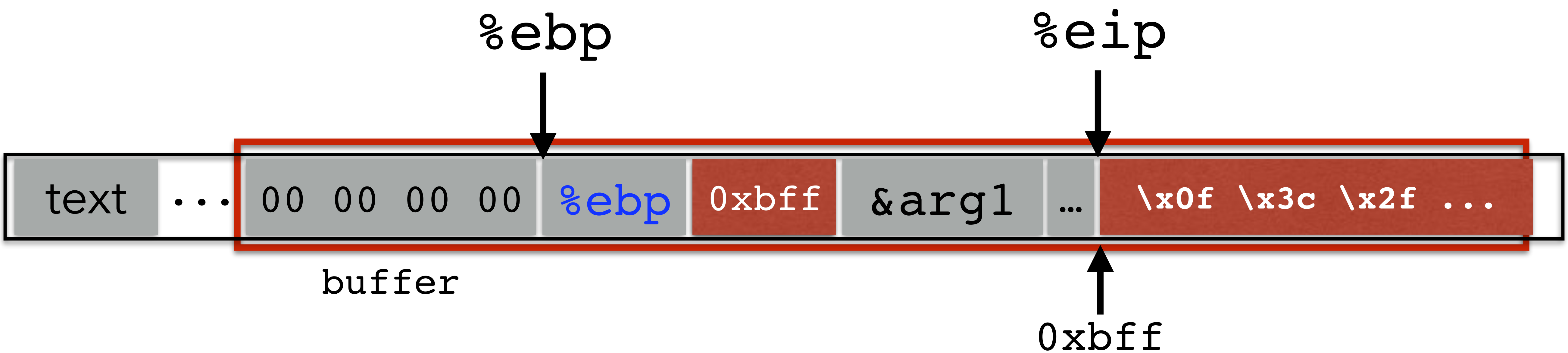
HIJACKING THE SAVED %EIP



HIJACKING THE SAVED %EIP



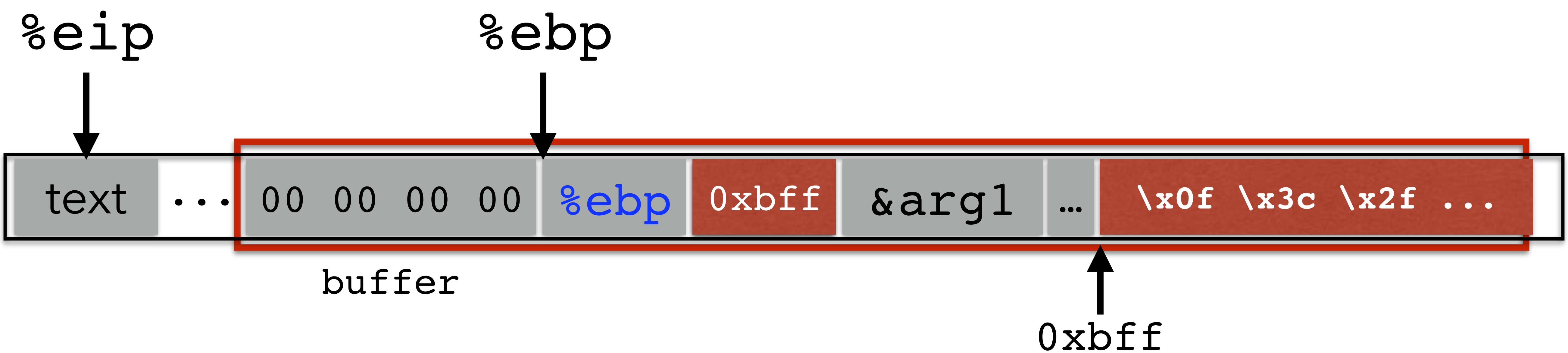
HIJACKING THE SAVED %EIP



But how do we know the address?

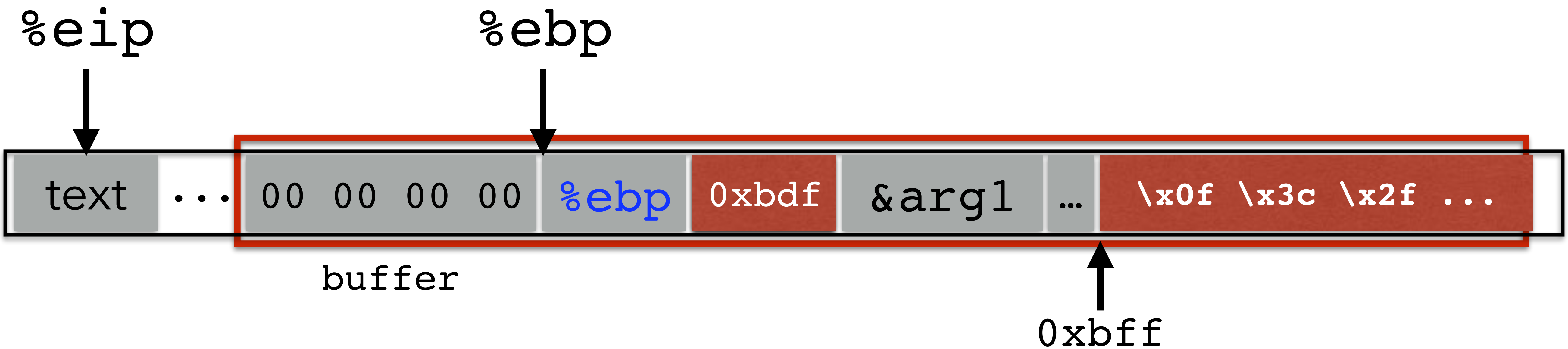
HIJACKING THE SAVED %EIP

What if we are wrong?



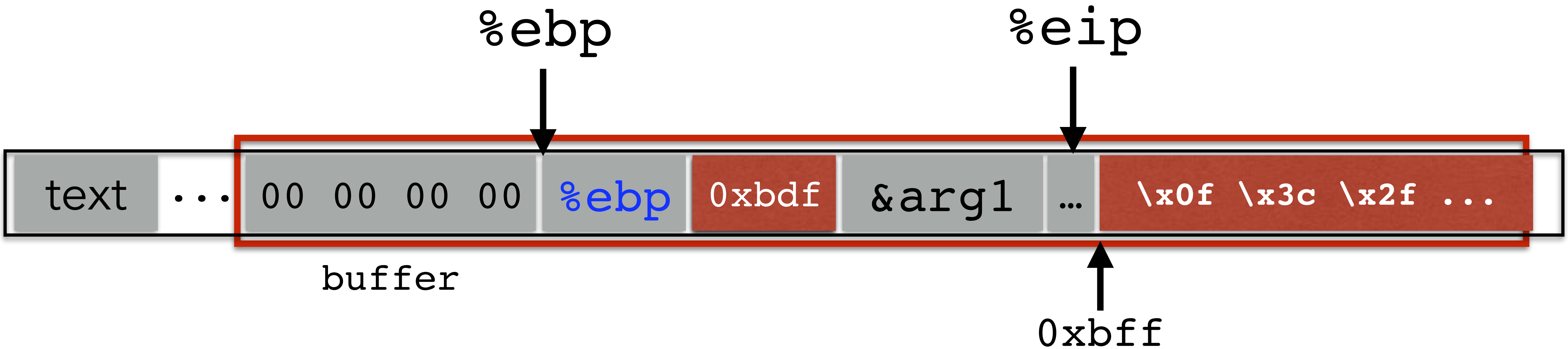
HIJACKING THE SAVED %EIP

What if we are wrong?



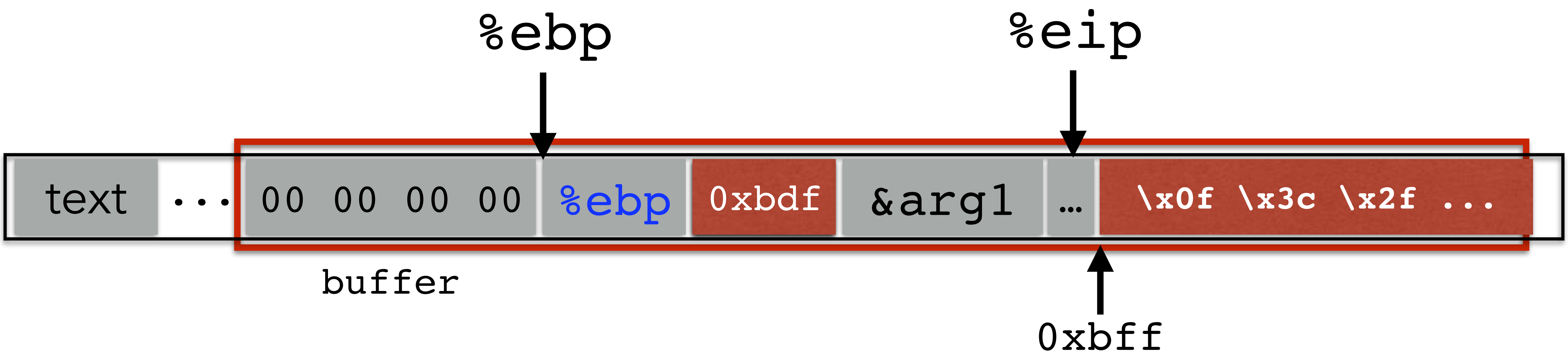
HIJACKING THE SAVED %EIP

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

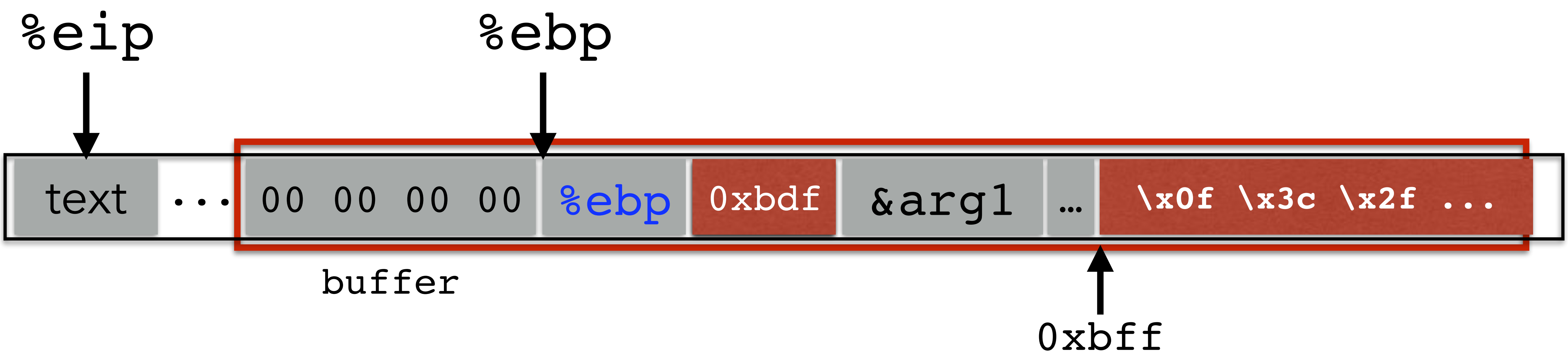
CHALLENGE 3: FINDING THE RETURN ADDRESS

- 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

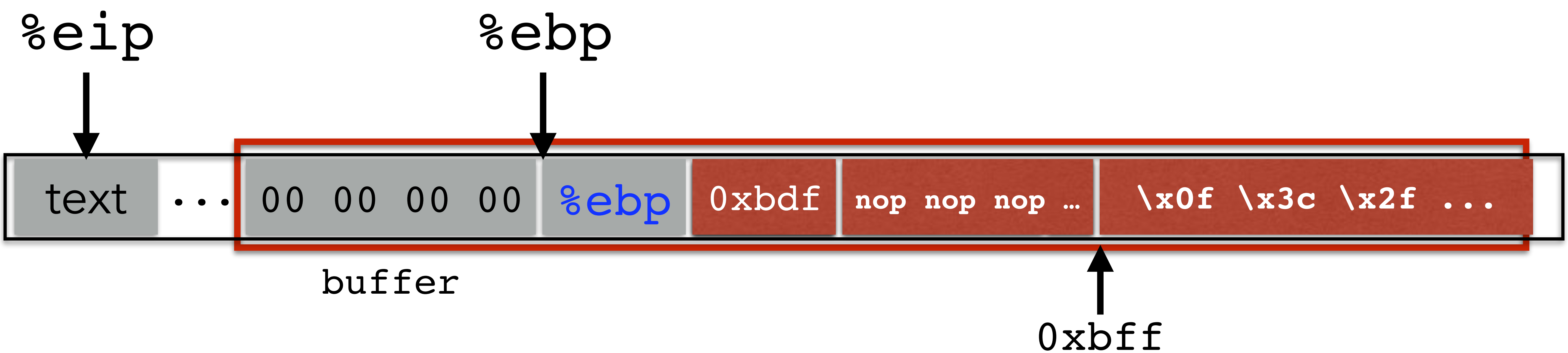
nop is a single-byte instruction

(just moves to the next instruction)



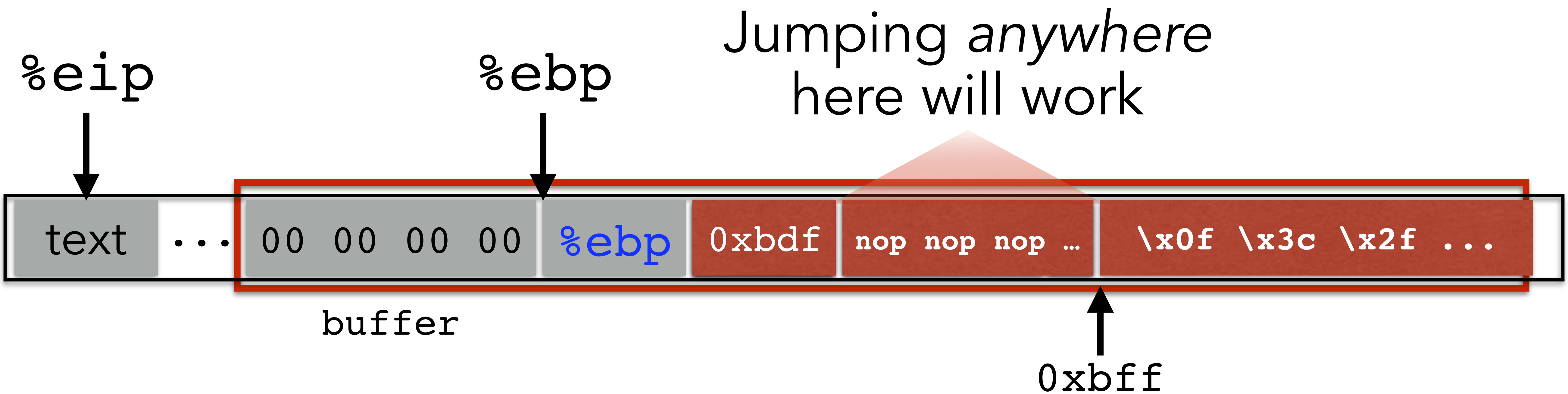
IMPROVING OUR CHANCES: NOP SLEDS

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



IMPROVING OUR CHANCES: NOP SLEDS

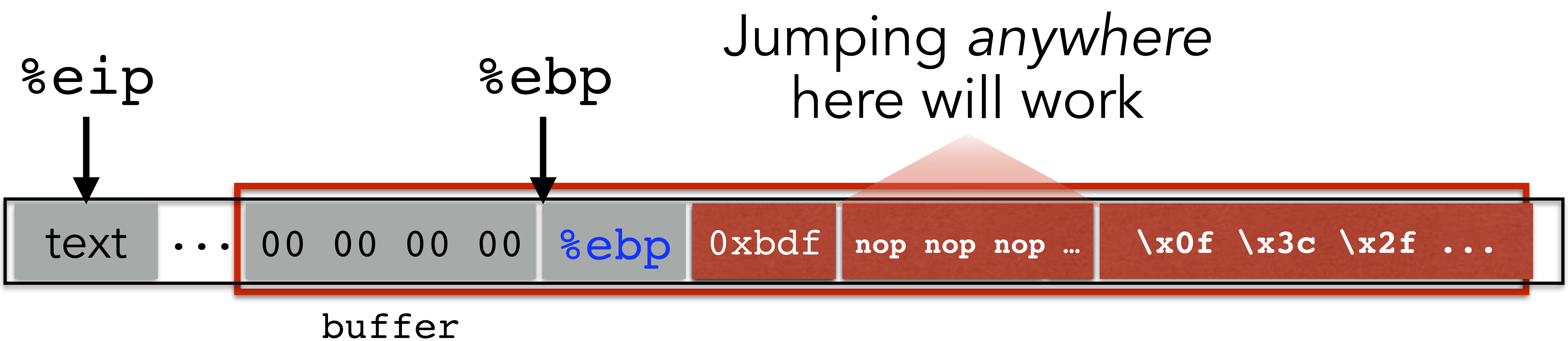
nop is a single-byte instruction
(just moves to the next instruction)



IMPROVING OUR CHANCES: NOP SLEDS

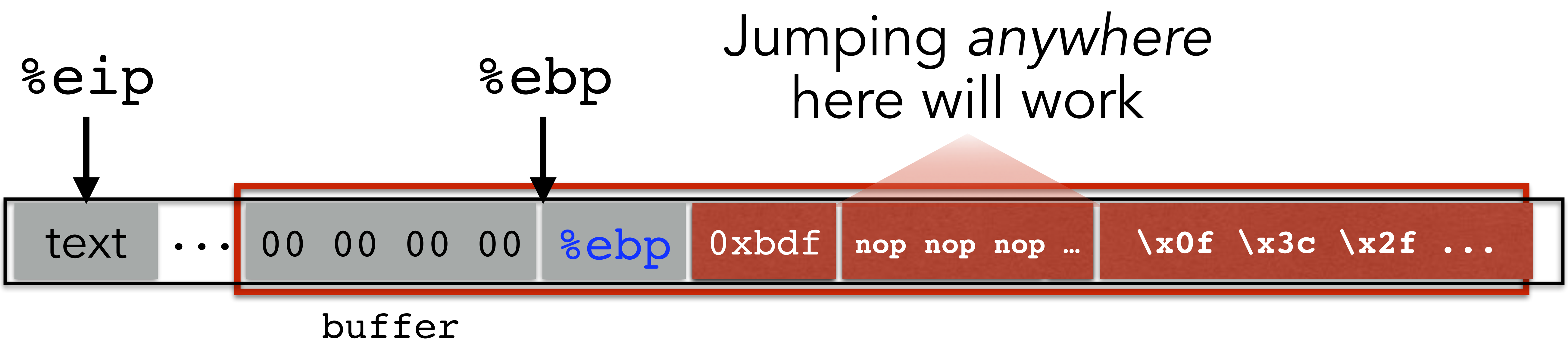
nop is a single-byte instruction

(just moves to the next instruction)



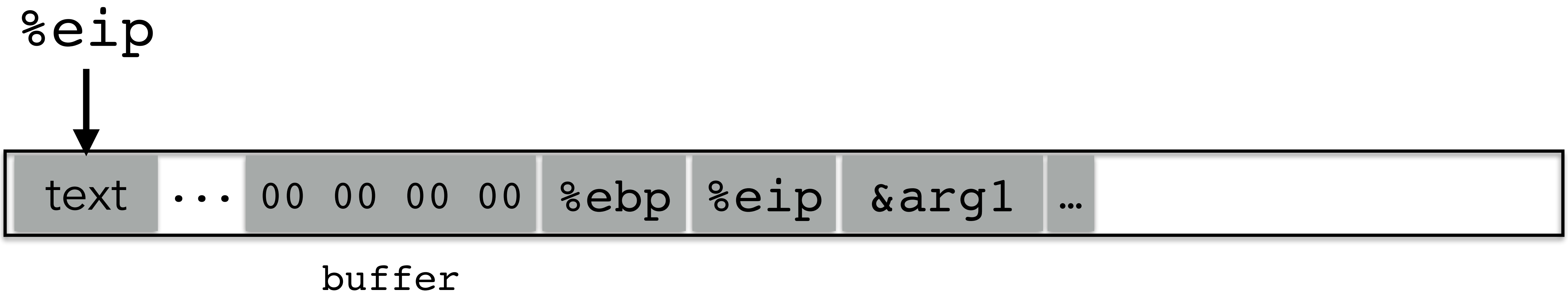
IMPROVING OUR CHANCES: NOP SLEDS

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



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

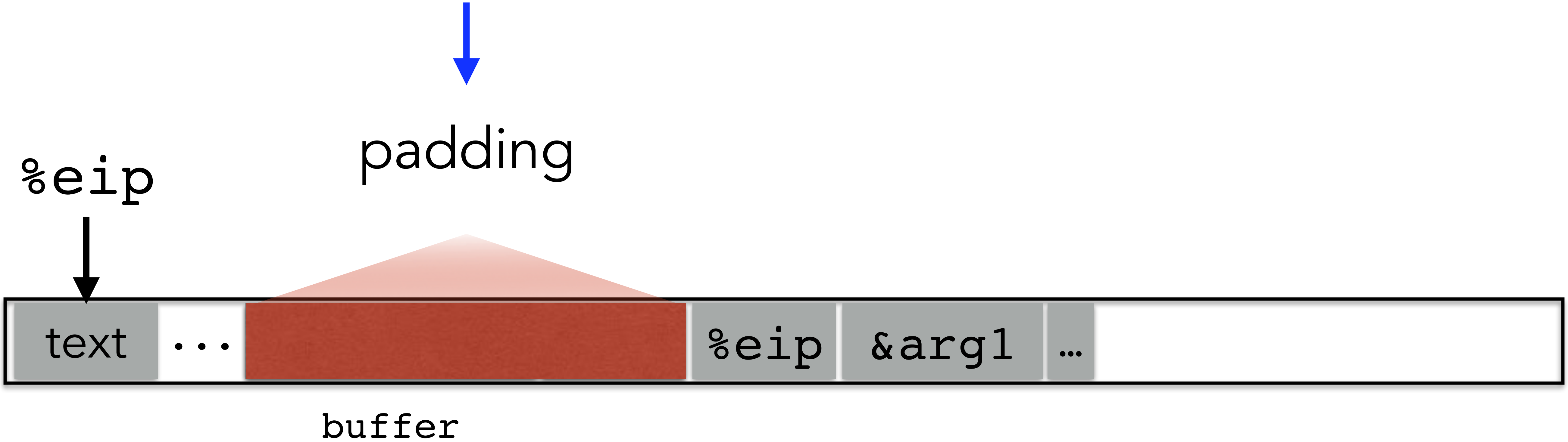
BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER



BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

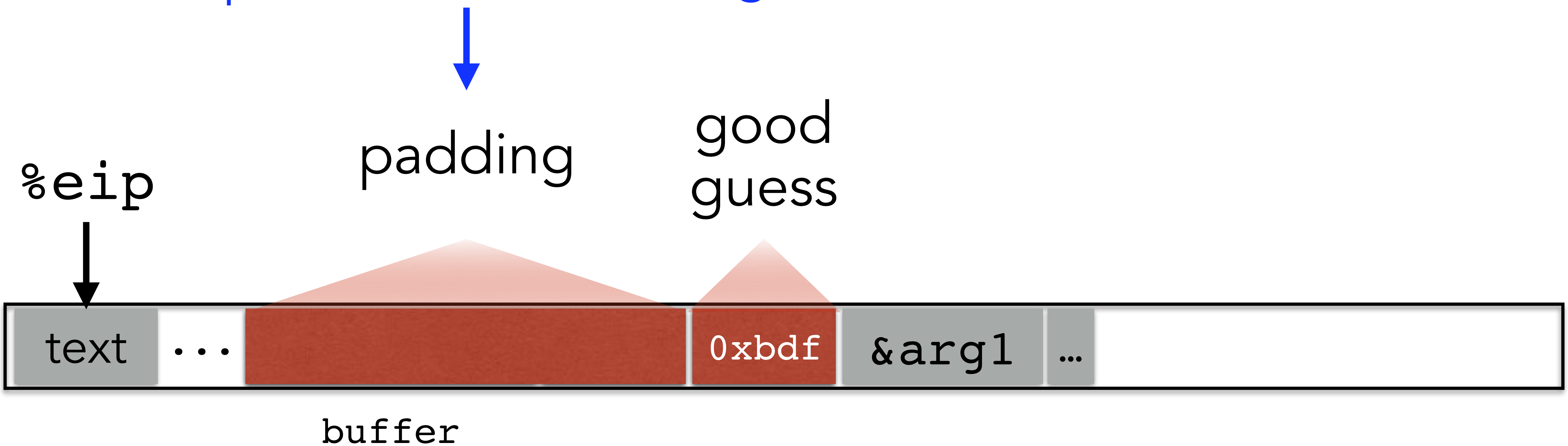
BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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



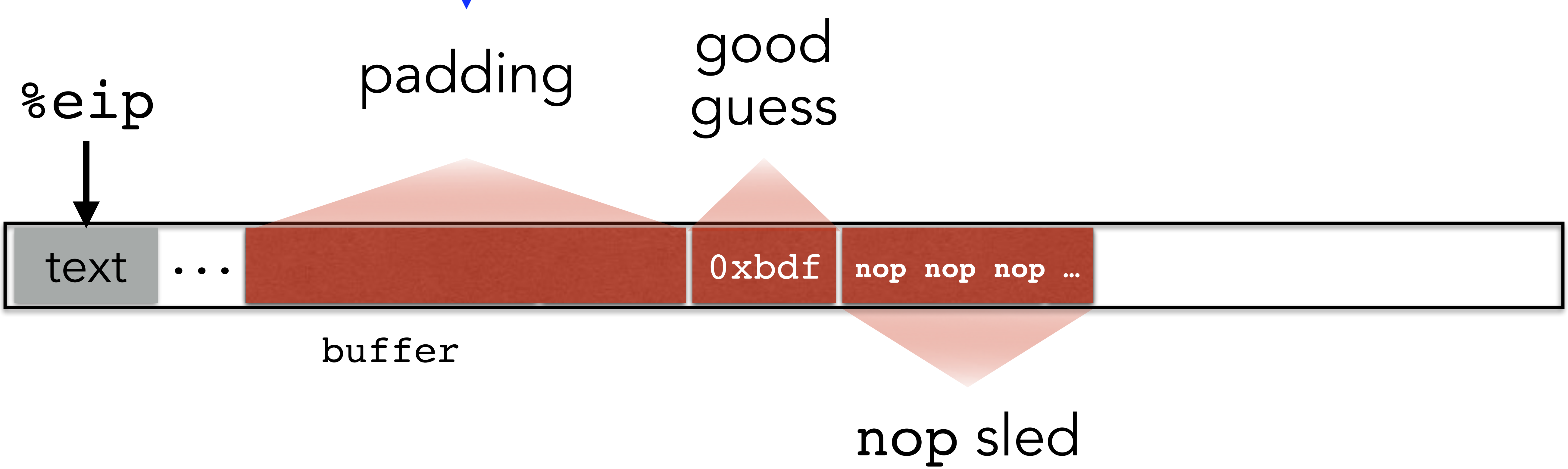
BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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



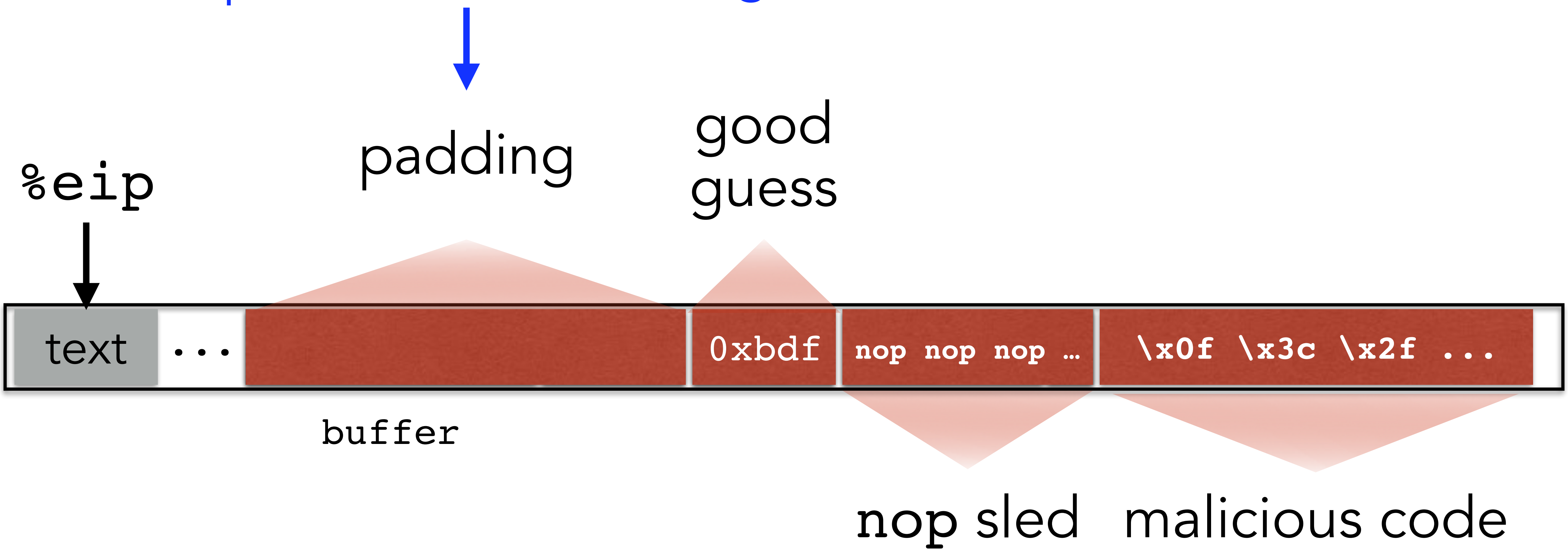
BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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



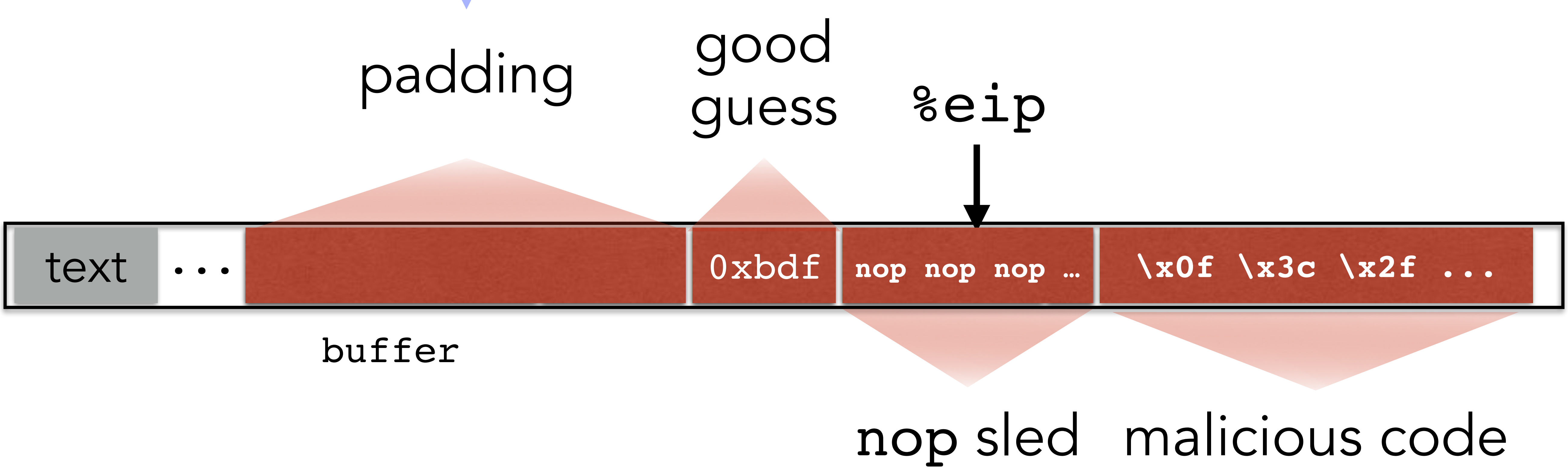
BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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.