Last time

We kicked off
our 1st section:
Software Security

By launching
Buffer overflows
and other memory safety vulnerabilities

- Buffer overflow fundamentals

This time

We will continue

Buffer overflows

and other memory safety vulnerabilities

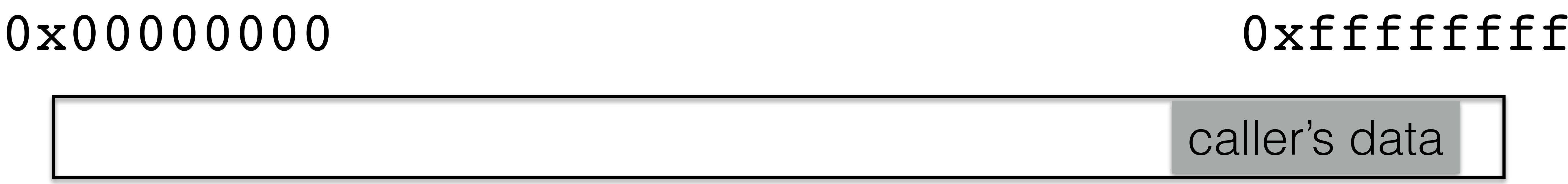
By looking at

Overflow Atks/Defs

- Finish stack layout review
- Attack!
- Everything you've always wanted to know about `gdb` but were too afraid to ask
- Overflow defenses

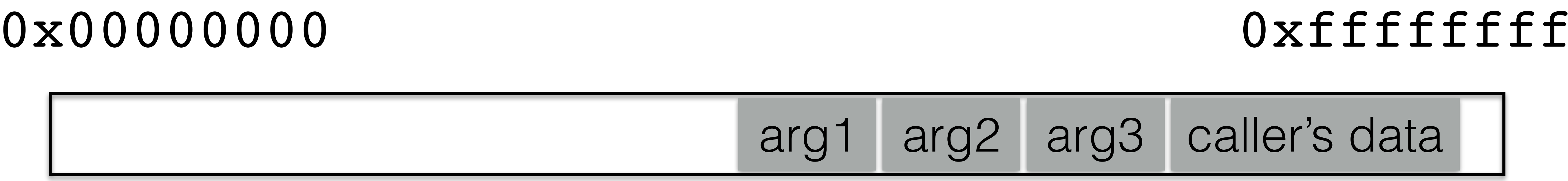
Stack layout when calling functions

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



Stack layout when calling functions

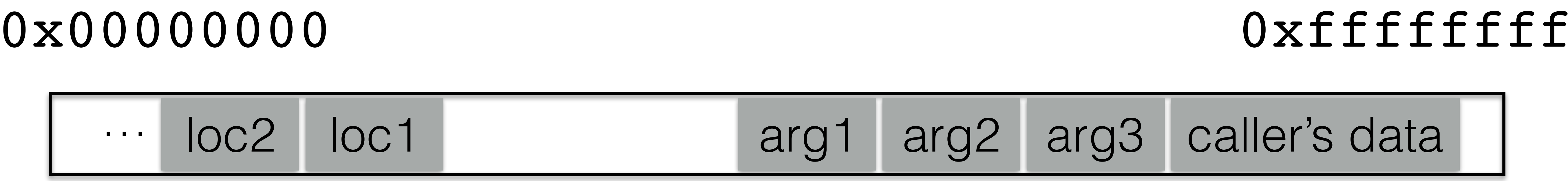
```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
}
```



Arguments pushed in reverse order of code

Stack layout when calling functions

```
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 functions

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

`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 functions

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

Two 4B values between the arguments and the local variables

0x00000000 0xffffffff

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

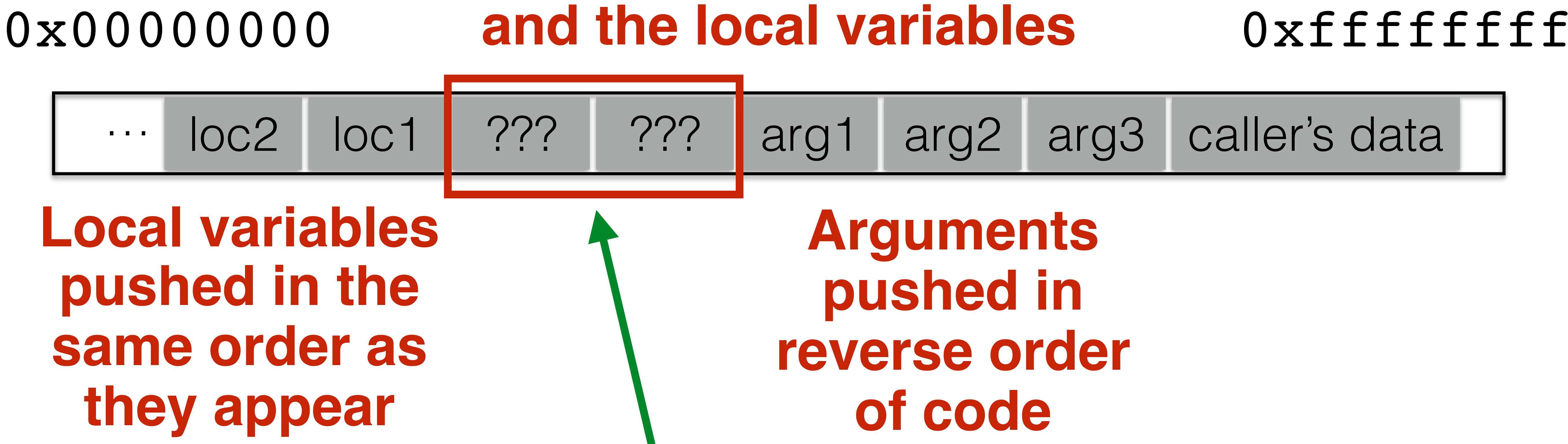
Local variables pushed in the same order as they appear in the code

Arguments pushed in reverse order of code

Stack layout when calling functions

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

Two 4B values between the arguments and the local variables



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

Arguments pushed in reverse order of code

Let's explore (and attack and defend) them

Accessing variables

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

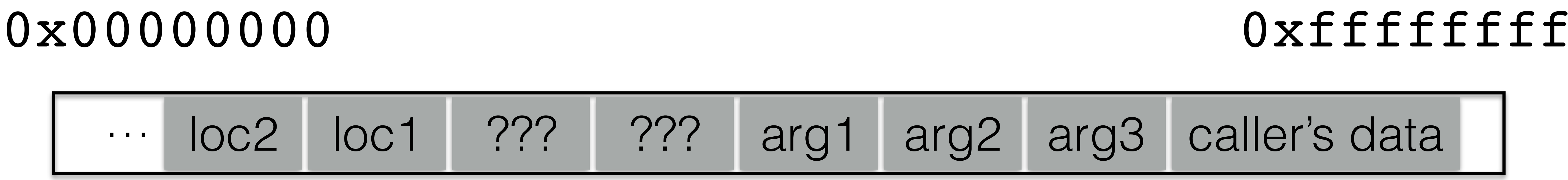
0x00000000 0xffffffff

···	loc2	loc1	???	???	arg1	arg2	arg3	caller’s data

Accessing variables

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

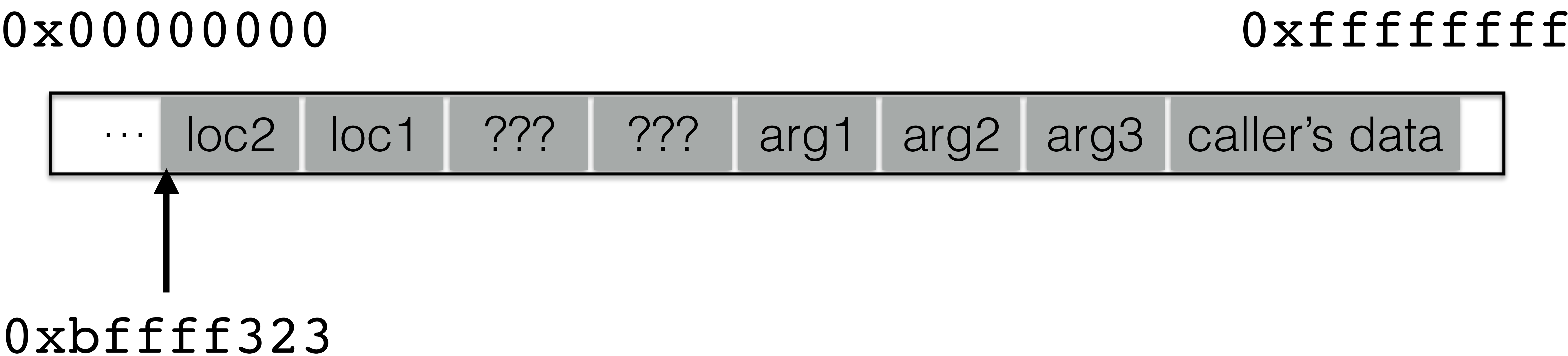
Q: Where is (this) `loc2`?



Accessing variables

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

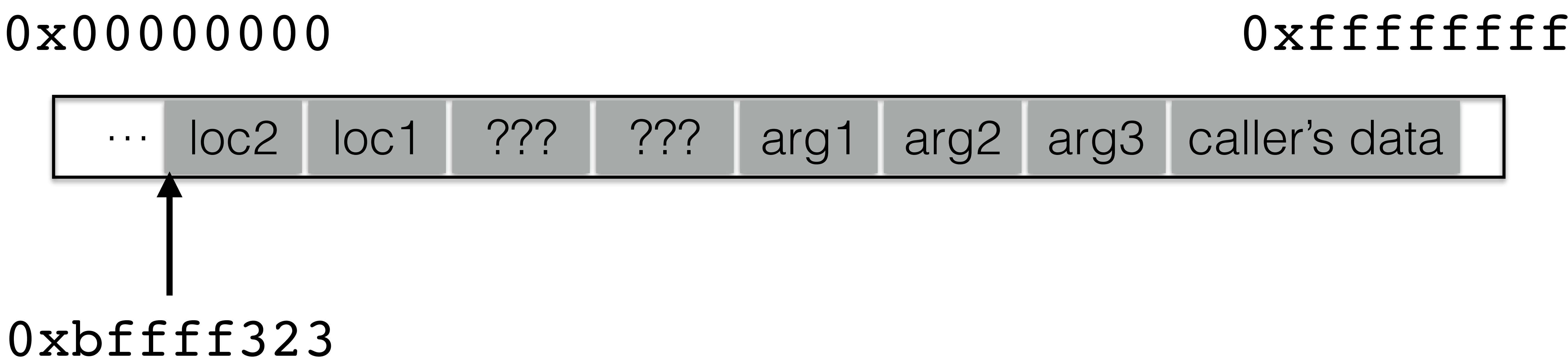
Q: Where is (this) `loc2`?



Accessing variables

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

Q: Where is (this) `loc2`?

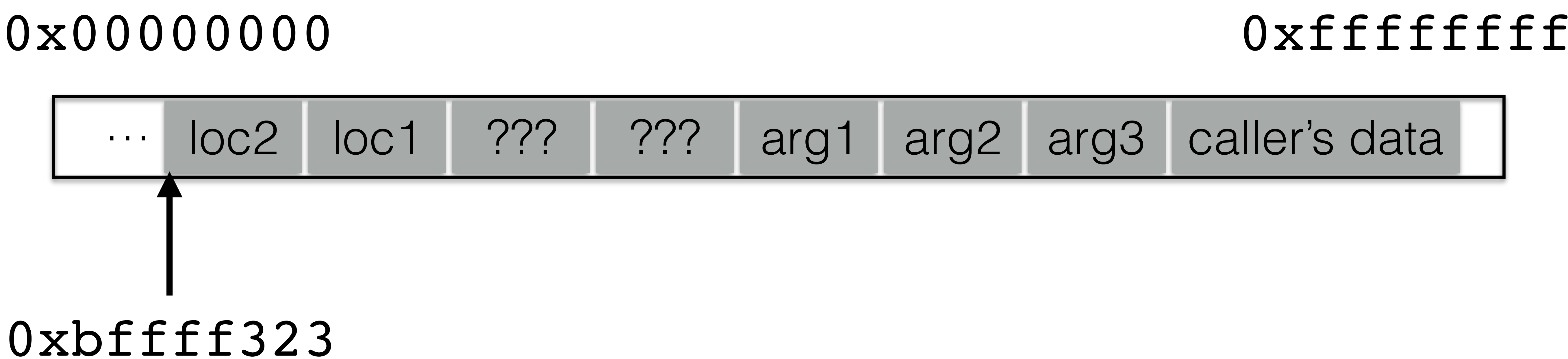


Undecidable at compile time

Accessing variables

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

Q: Where is (this) `loc2`?



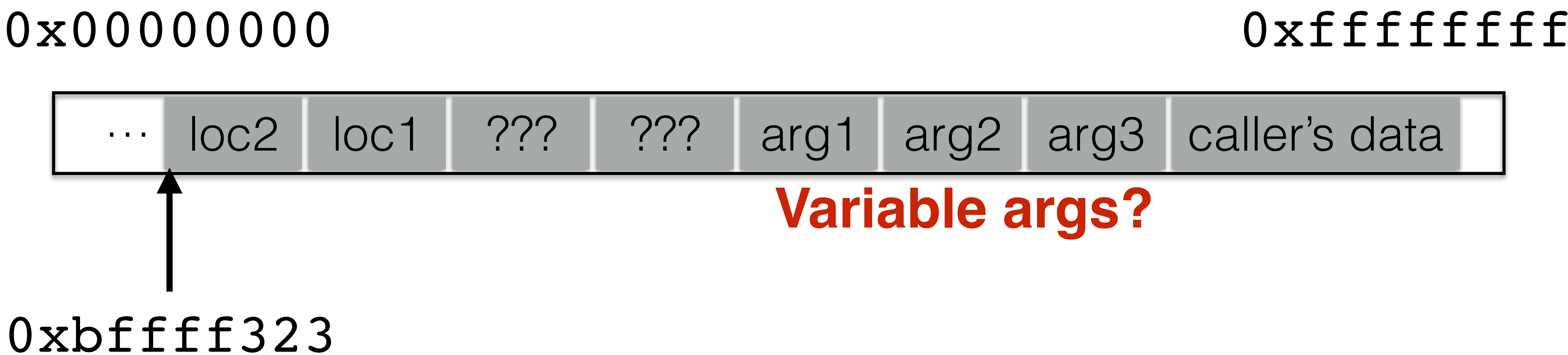
Undecidable at compile time

- 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`?

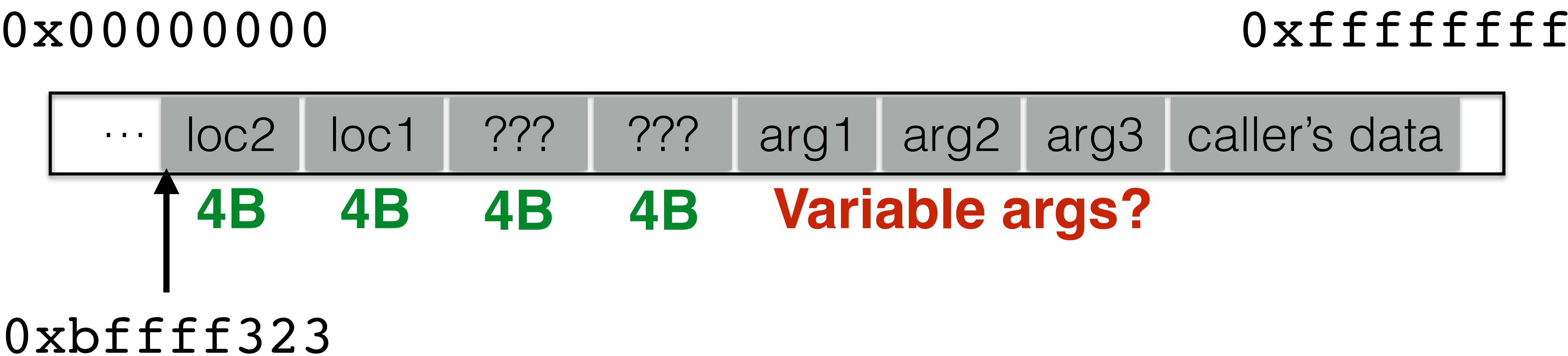


Undecidable at compile time

- 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?
    ...
}
```



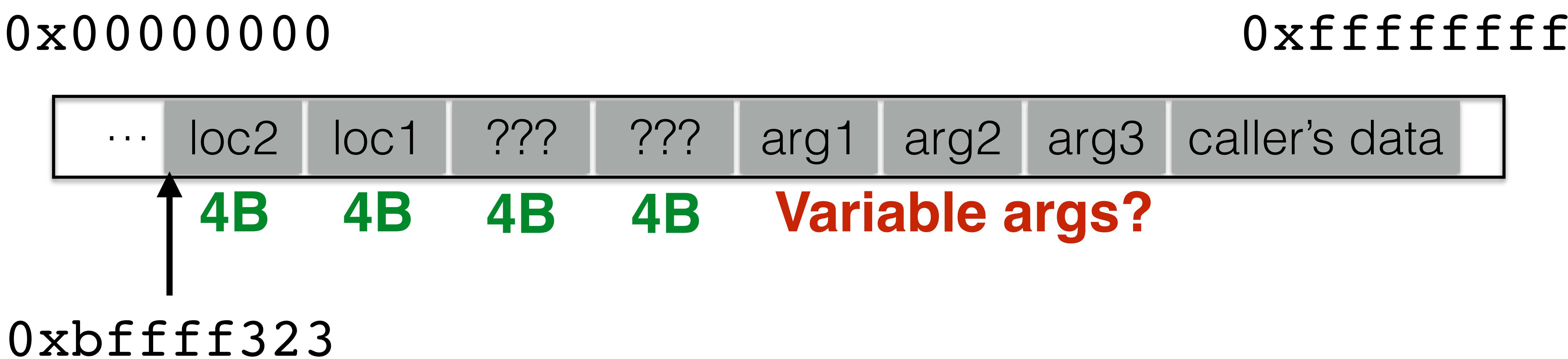
Undecidable at compile time

- 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`?



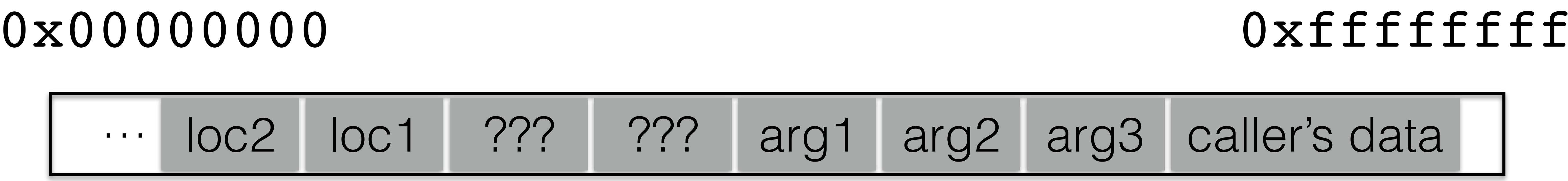
Undecidable at compile time

- 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`?

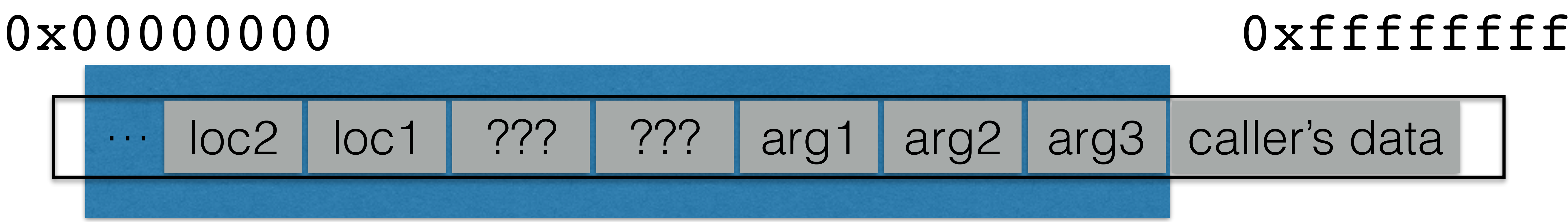


- I don't know where loc2 is,
- and I don't know how many args
- *but* loc2 is *always* 8B before "???"s

Accessing variables

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

Q: Where is (this) `loc2`?



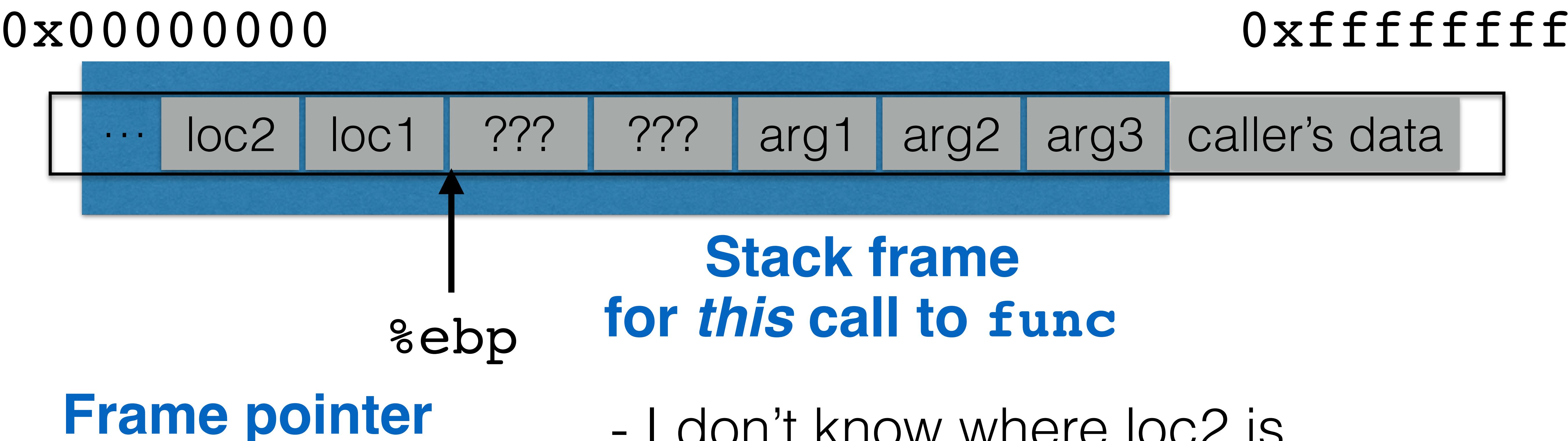
Stack frame
for *this* call to `func`

- 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`?



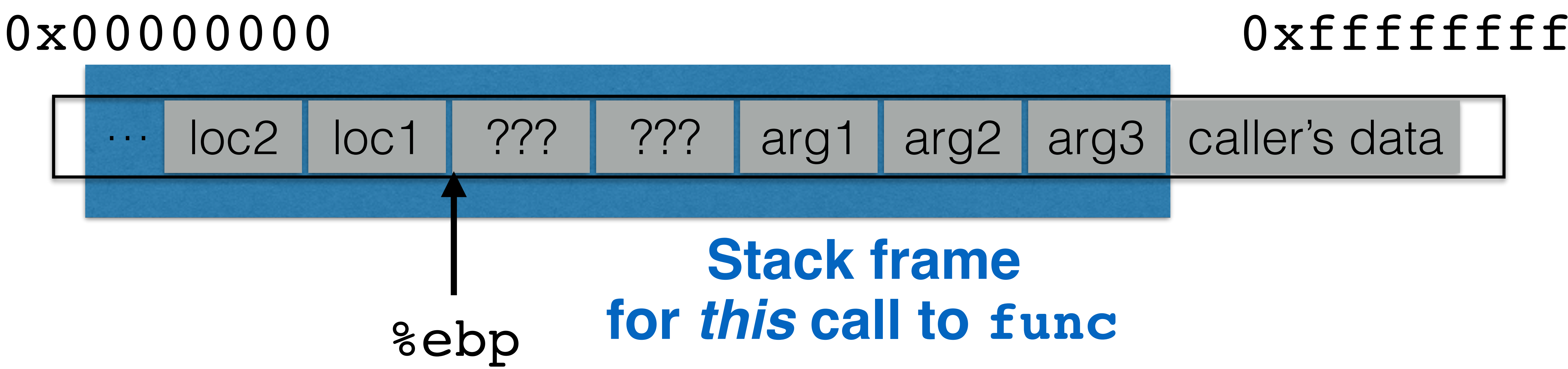
- I don't know where loc2 is,
- and I don't know how many args
- *but* loc2 is *always* 8B before "???"s

Accessing variables

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

Q: Where is (this) `loc2`?

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

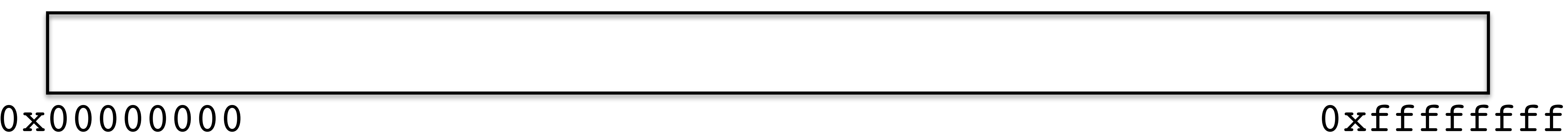
`(%ebp)` The value at memory address **`%ebp`**
(like dereferencing a pointer)

`0x00000000` `0xffffffff`

Notation

`0xbfff03b8` **`%ebp`** A memory address

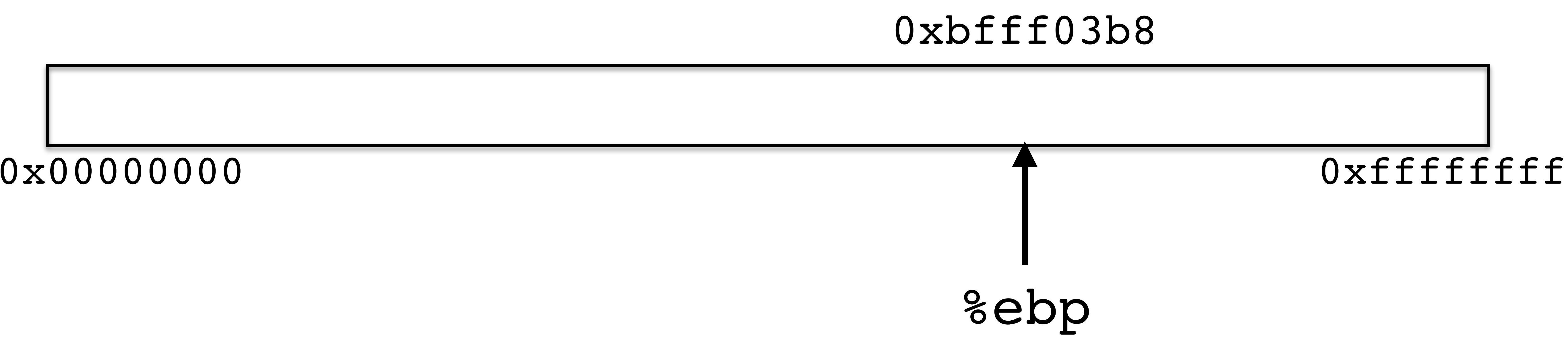
`(%ebp)` The value at memory address **`%ebp`**
(like dereferencing a pointer)



Notation

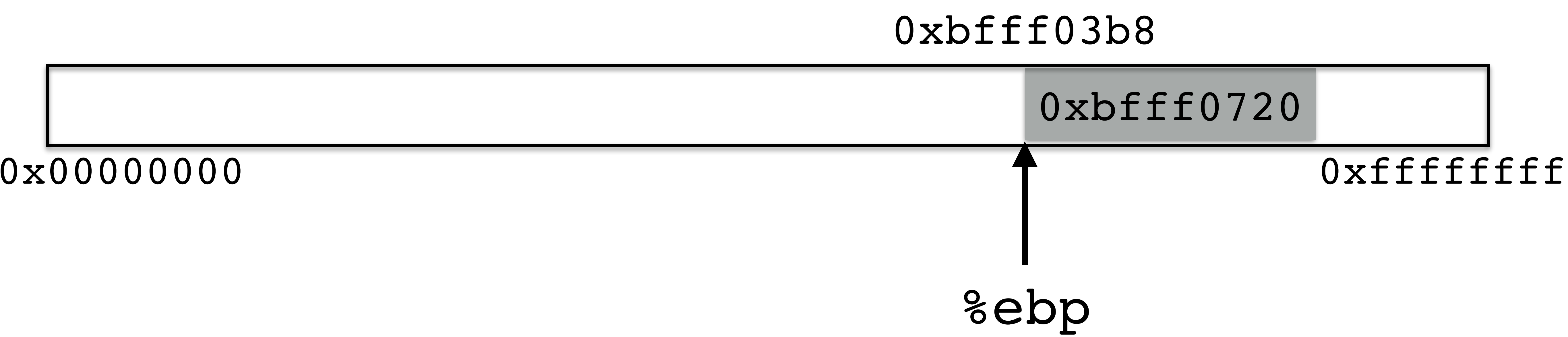
`0xbfff03b8` **`%ebp`** A memory address

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



Notation

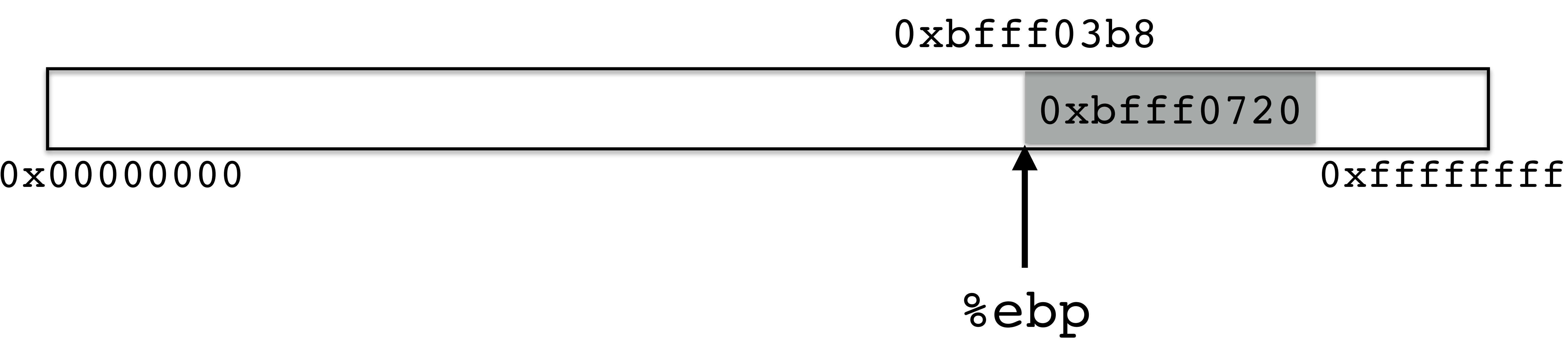
`0xbfff03b8`	**`%ebp`**	A memory address
`0xbfff0720`	**`(%ebp)`**	The value at memory address **`%ebp`** (like dereferencing a pointer)



Notation

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

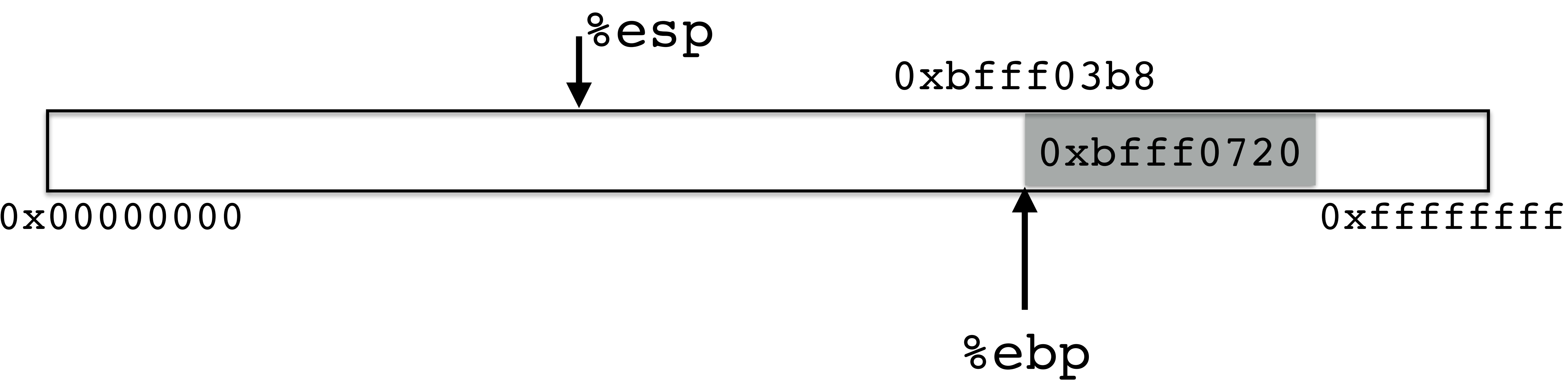
`pushl %ebp`



Notation

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

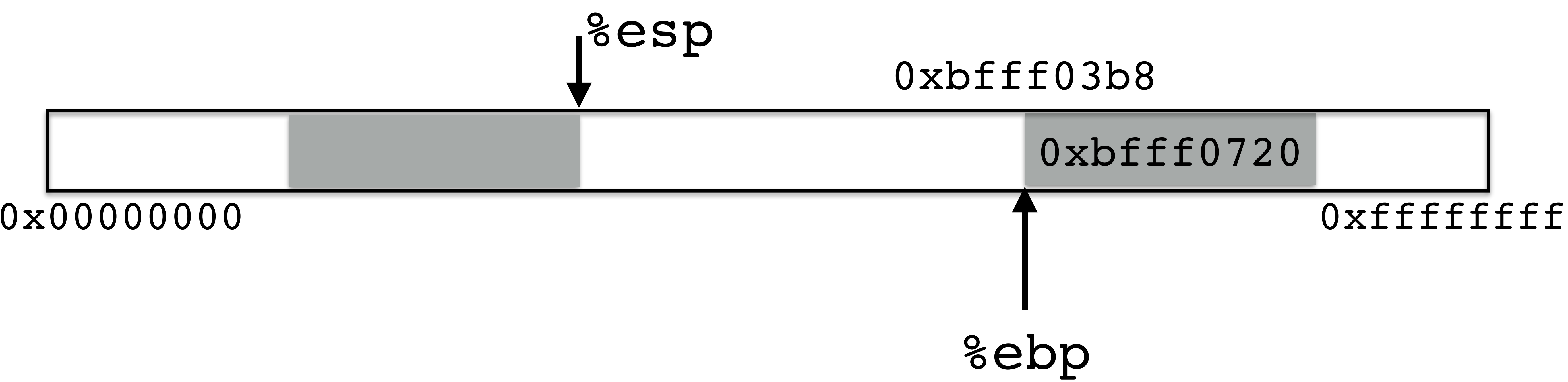
`pushl %ebp`



Notation

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

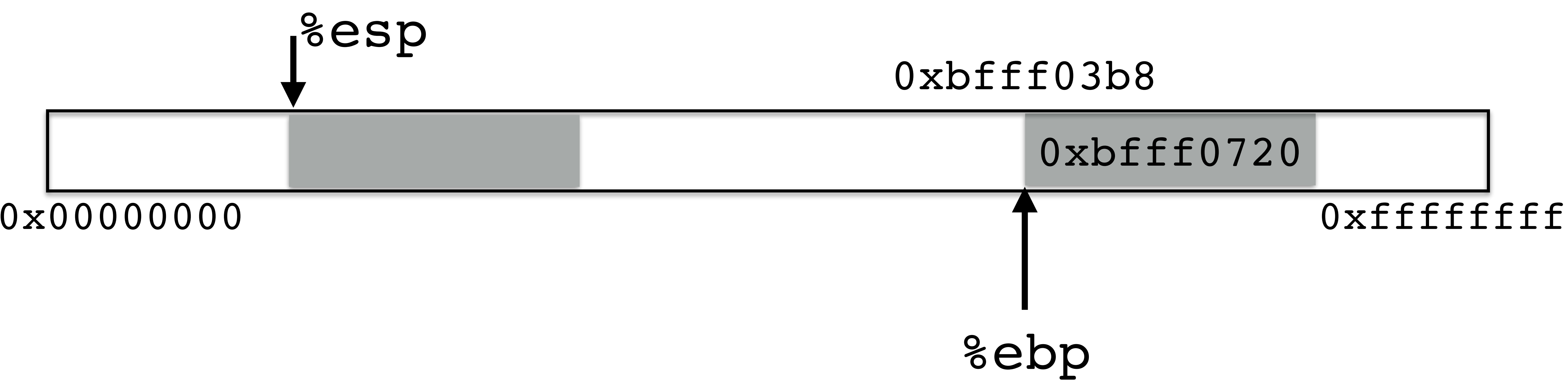
`pushl %ebp`



Notation

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

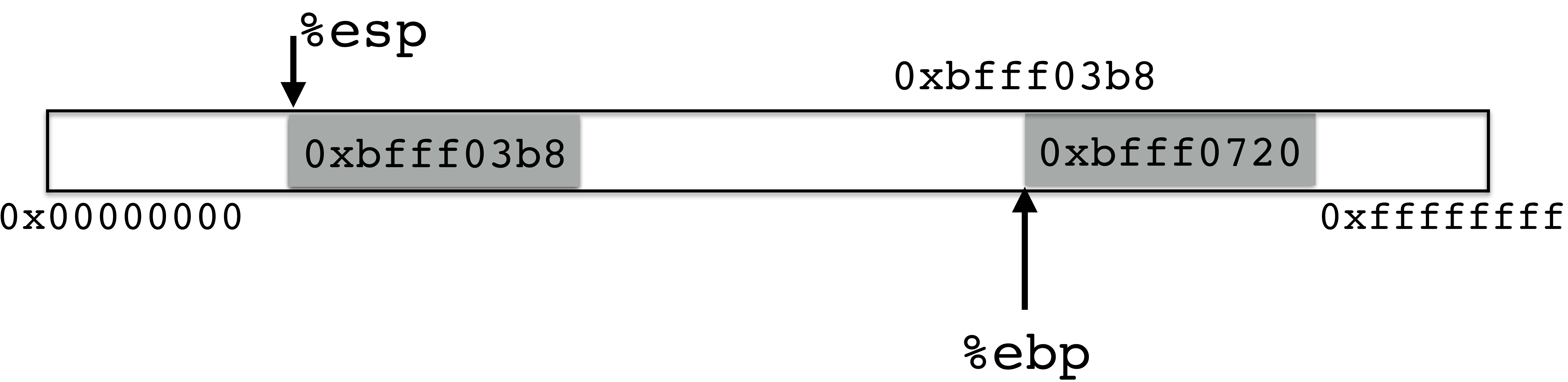
`pushl %ebp`



Notation

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

`pushl %ebp`

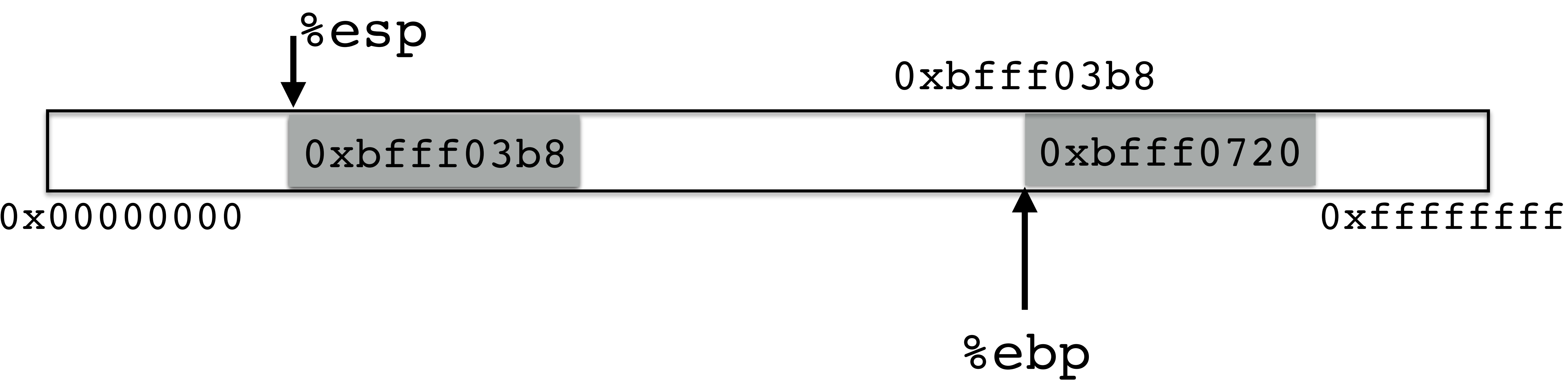


Notation

`0xbfff03b8` `%ebp` A memory address

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

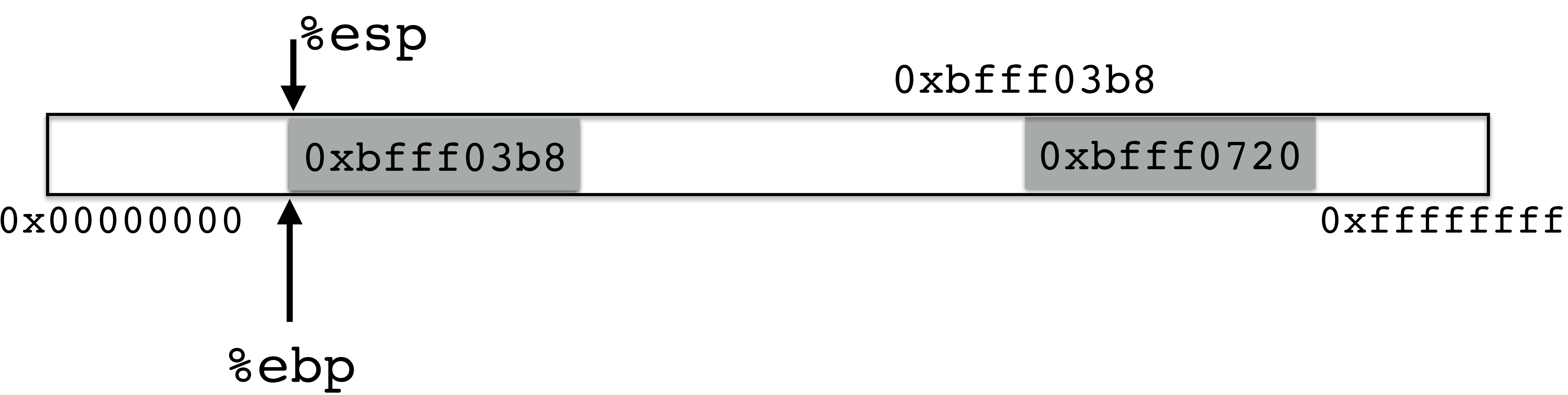
```
pushl %ebp
movl  %esp %ebp /* %ebp = %esp */
```



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 */
```

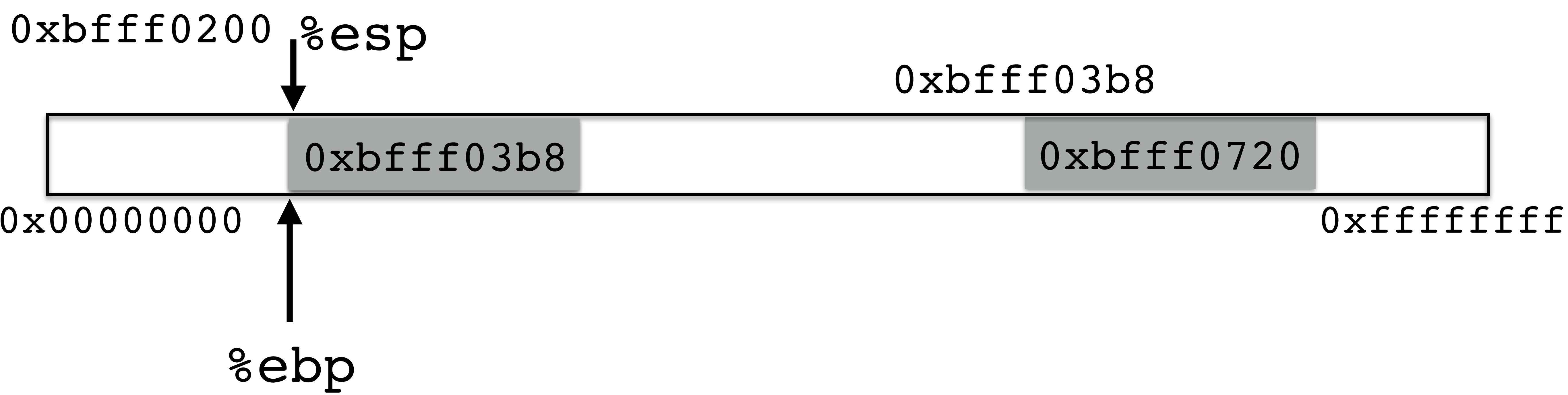


Notation

`0xbfff03b8` `%ebp` A memory address

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

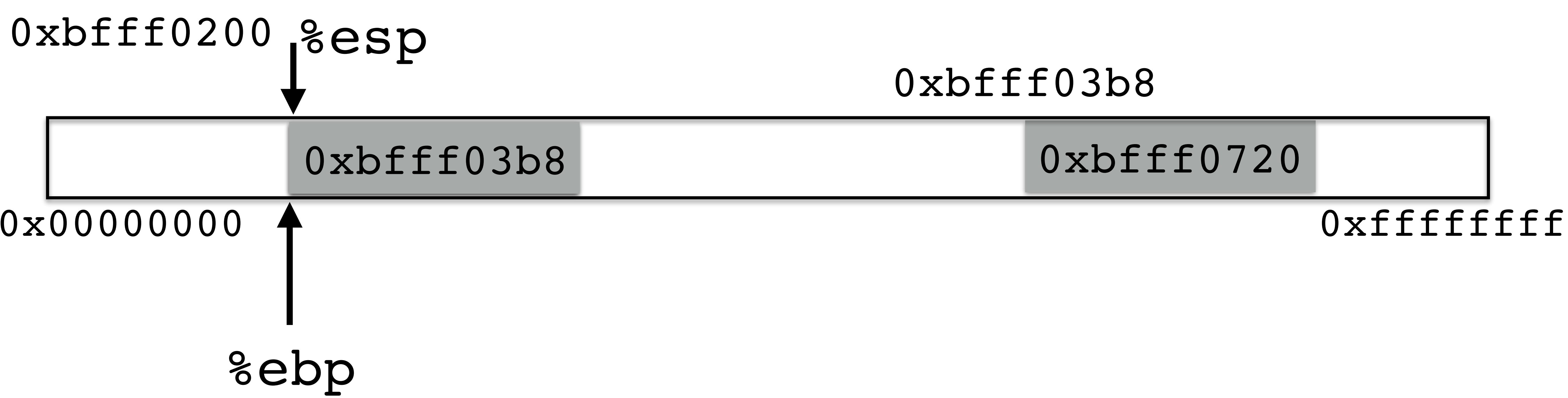
```
pushl %ebp
movl  %esp %ebp  /* %ebp = %esp */
```



Notation

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

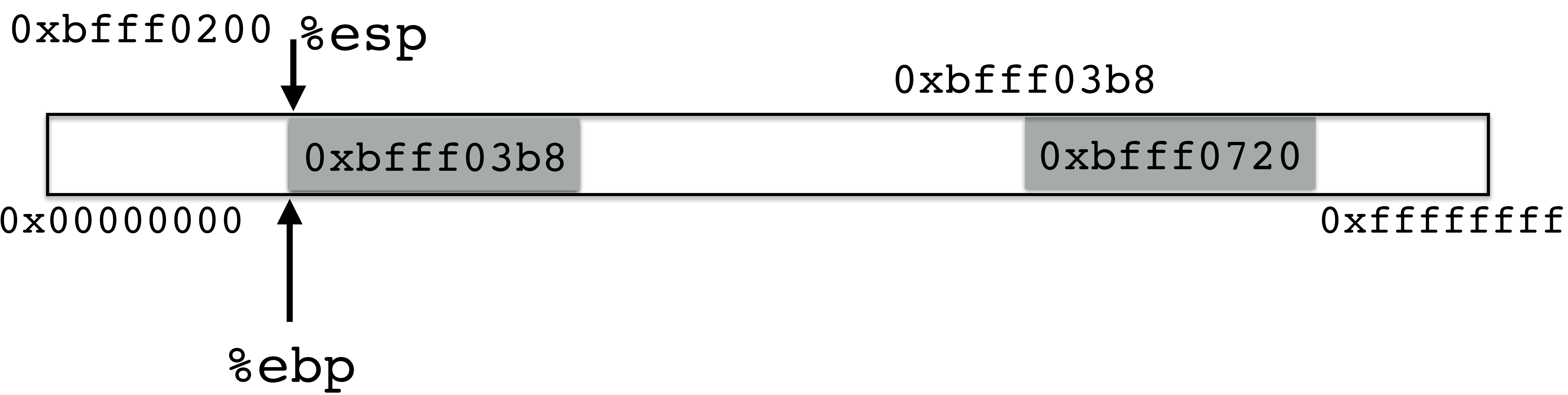
```
pushl %ebp
movl  %esp %ebp  /* %ebp = %esp */
```



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 */
```

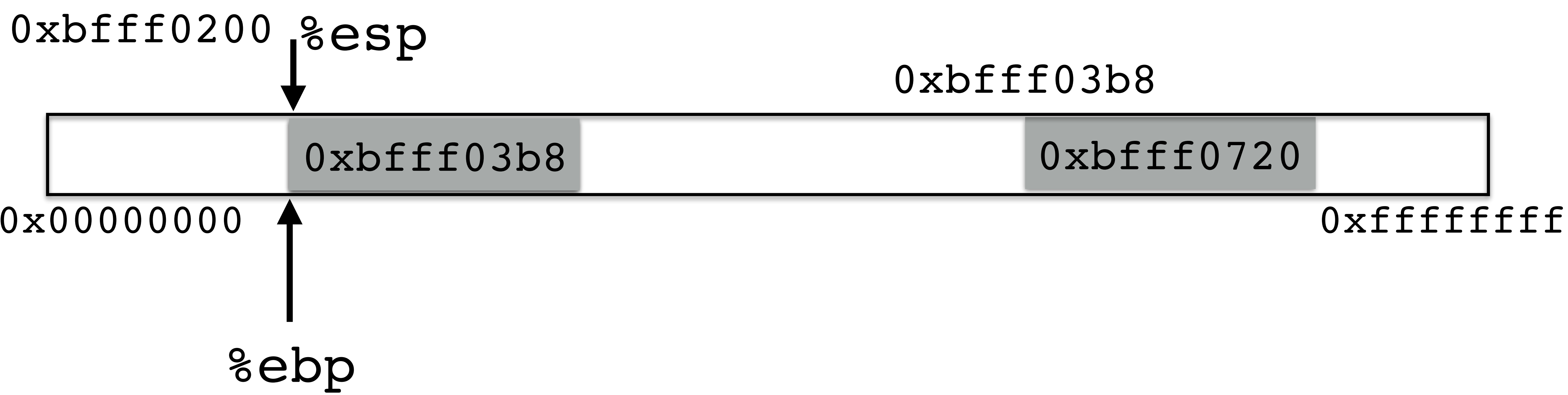


Notation

~~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) */
```

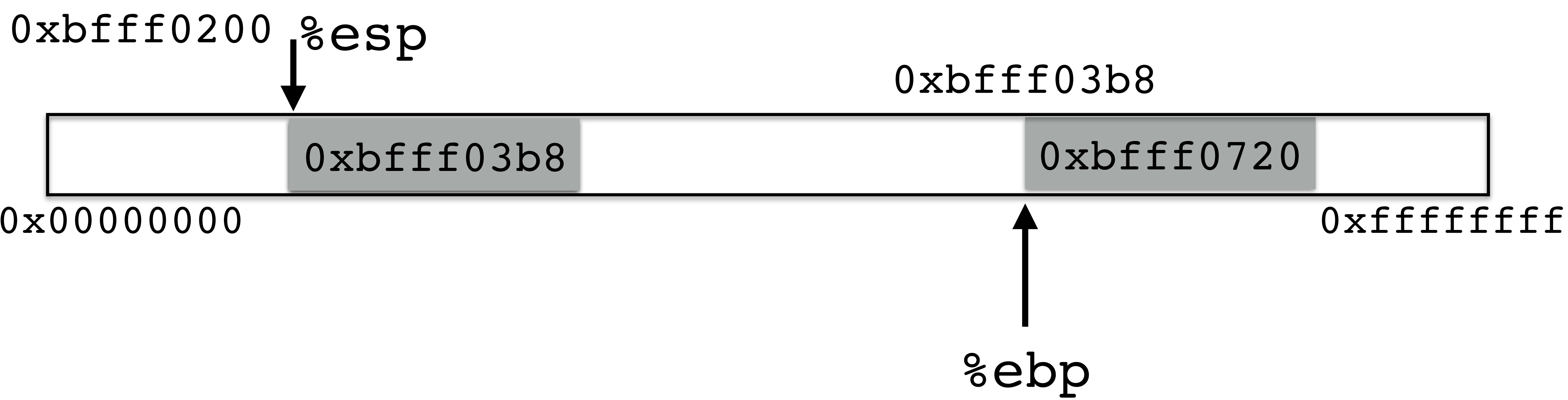


Notation

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

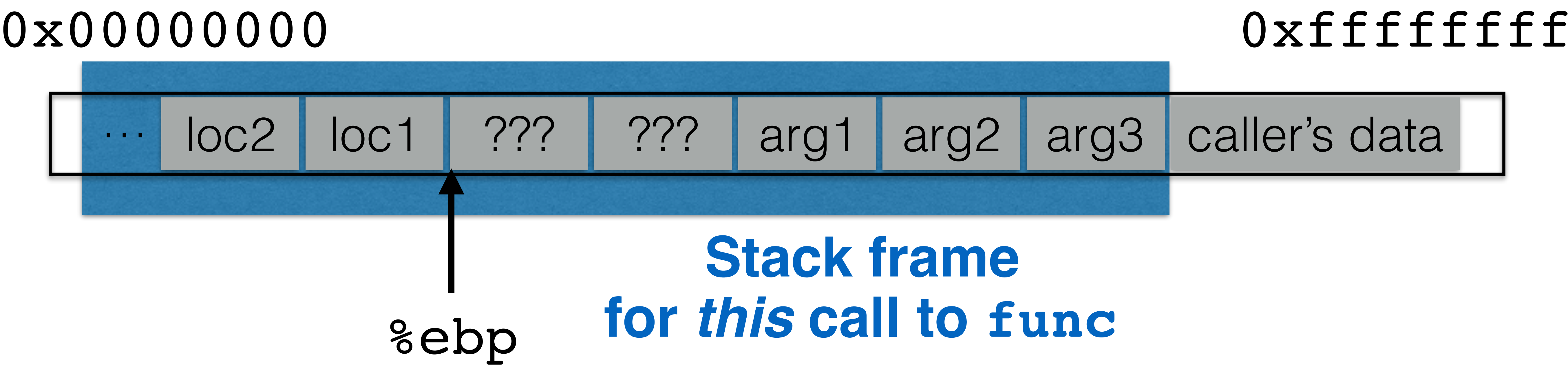
~~0xbfff0720~~
0xbfff03b8 `(%ebp)` The value at memory address `%ebp`
(like dereferencing a pointer)

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



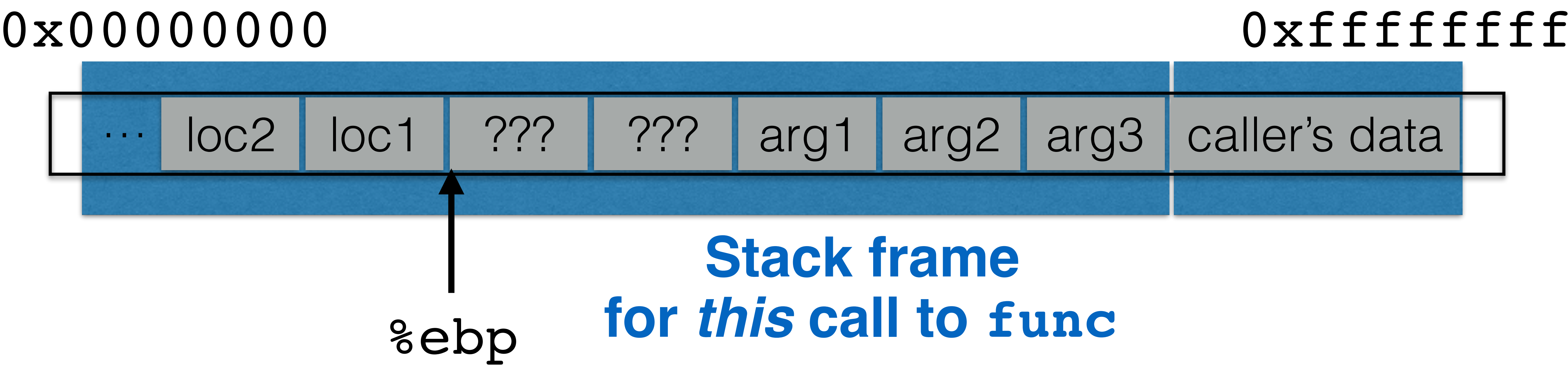
Returning from functions

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



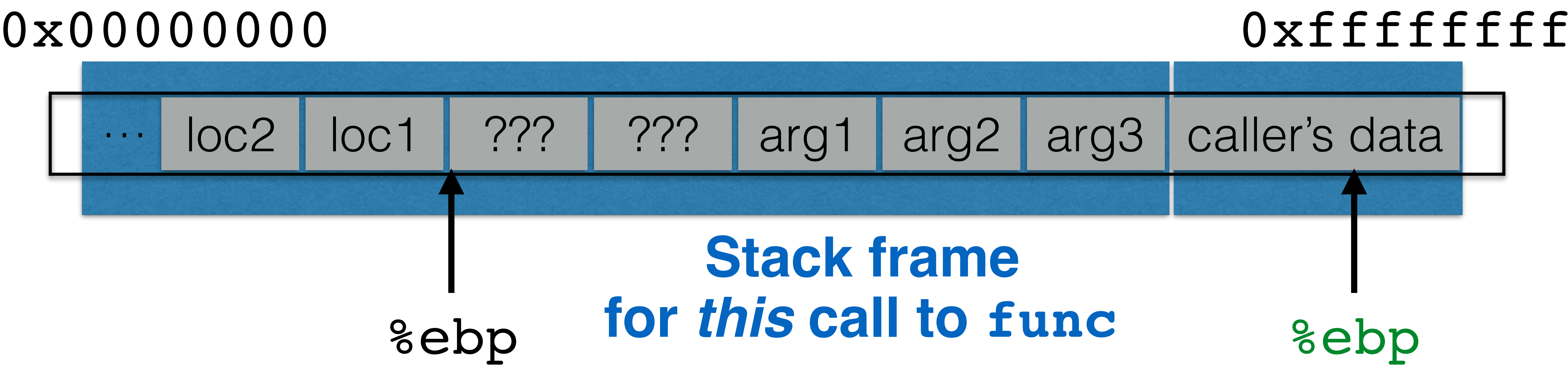
Returning from functions

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



Returning from functions

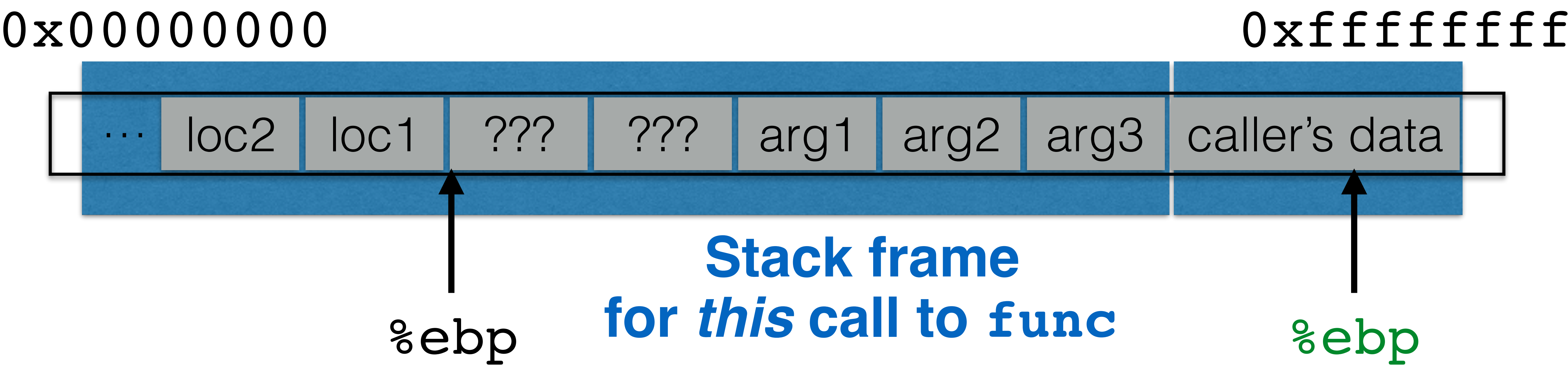
```
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}
```



Returning from functions

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

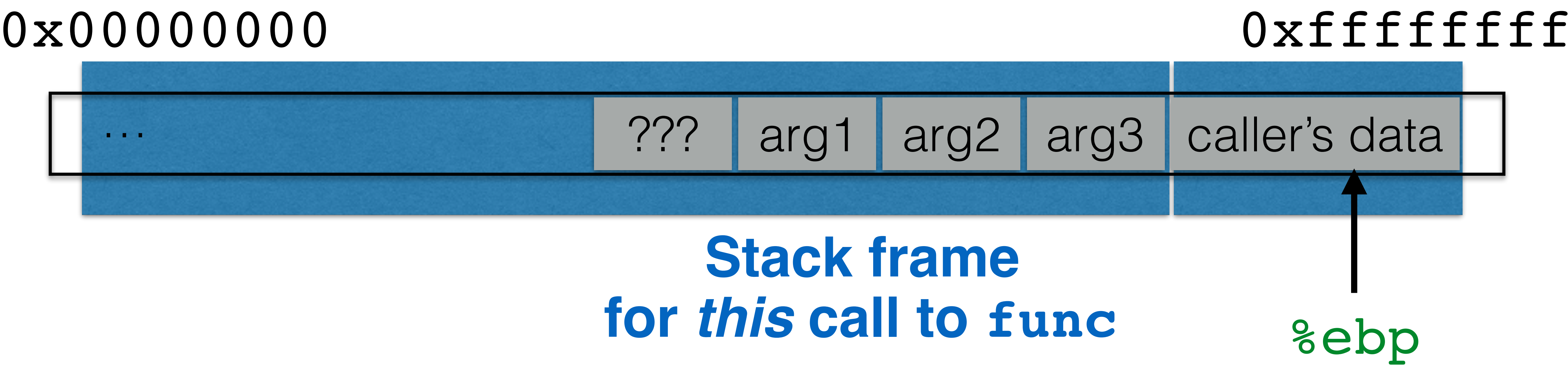
Q: How do we restore %ebp?



Returning from functions

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

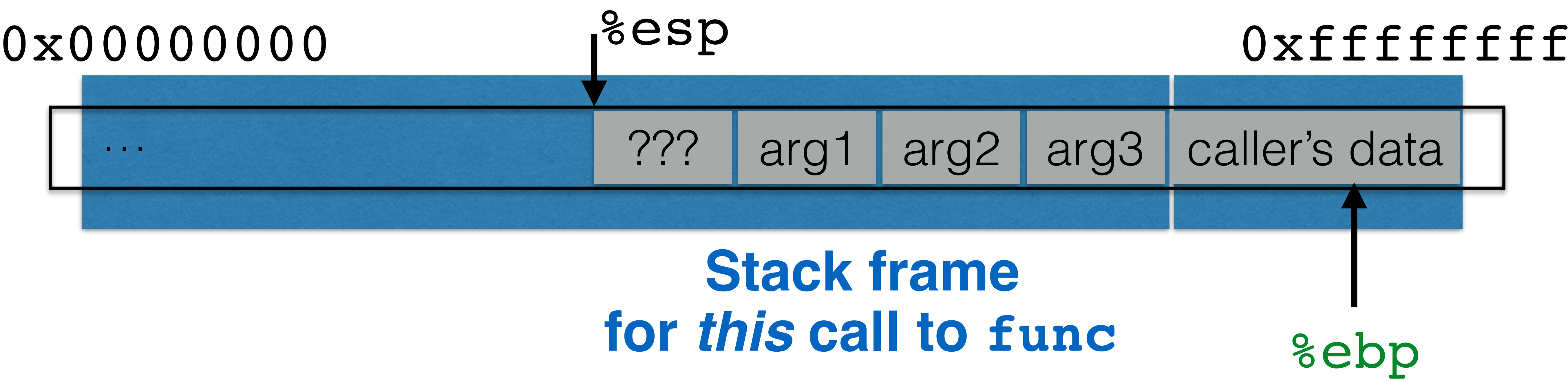
Q: How do we restore %ebp?



Returning from functions

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

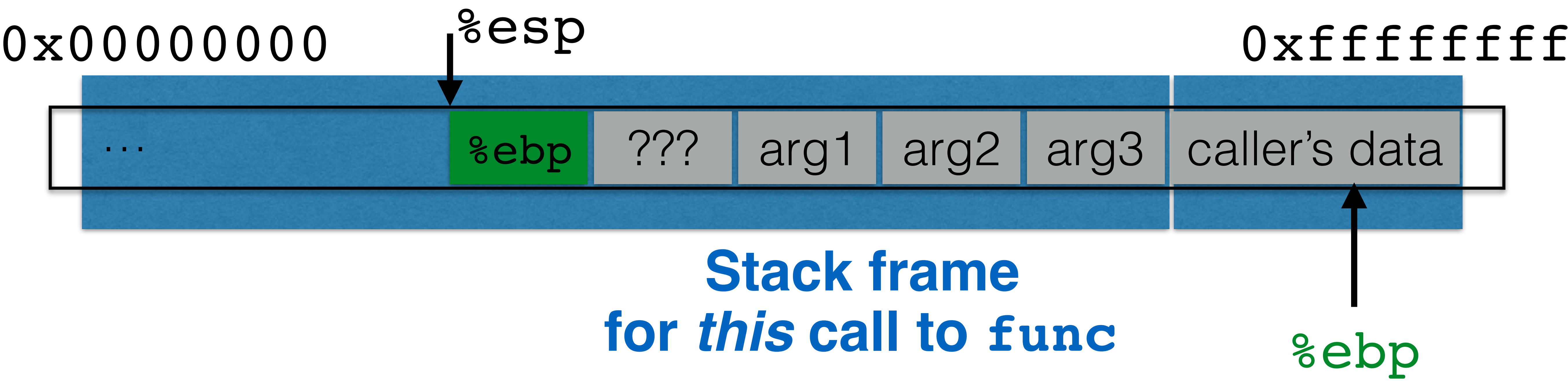
Q: How do we restore %ebp?



Returning from functions

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

Q: How do we restore %ebp?

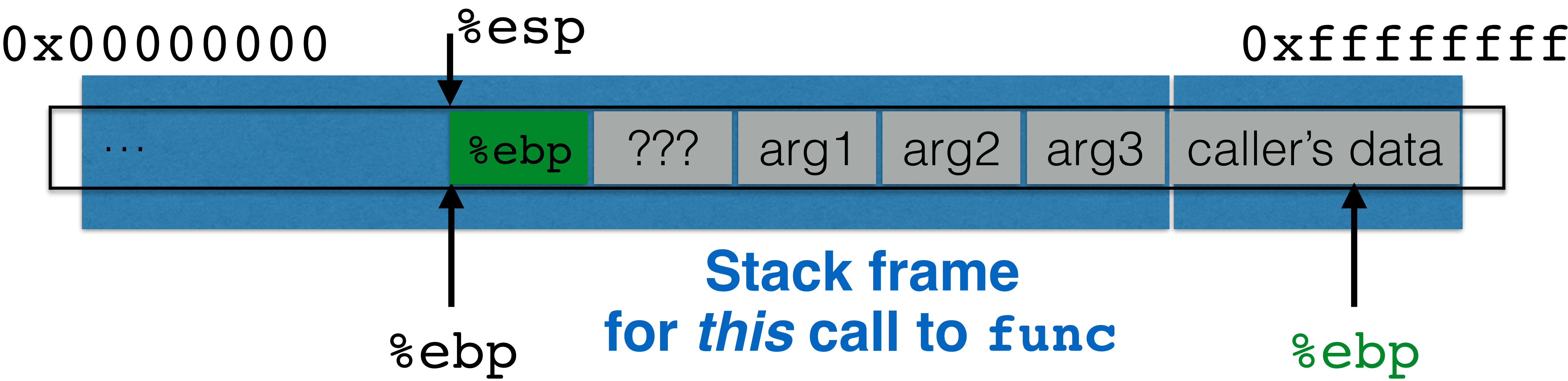


1. Push `%ebp` before `locals`

Returning from functions

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

Q: How do we restore %ebp?

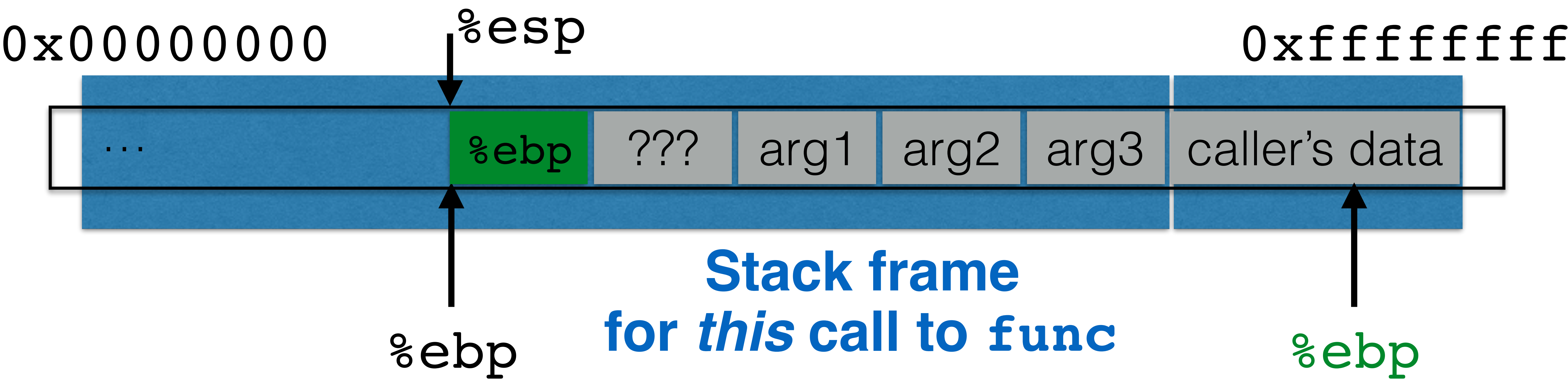


1. **Push `%ebp` before `locals`**
2. **Set `%ebp` to current `%esp`**

Returning from functions

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

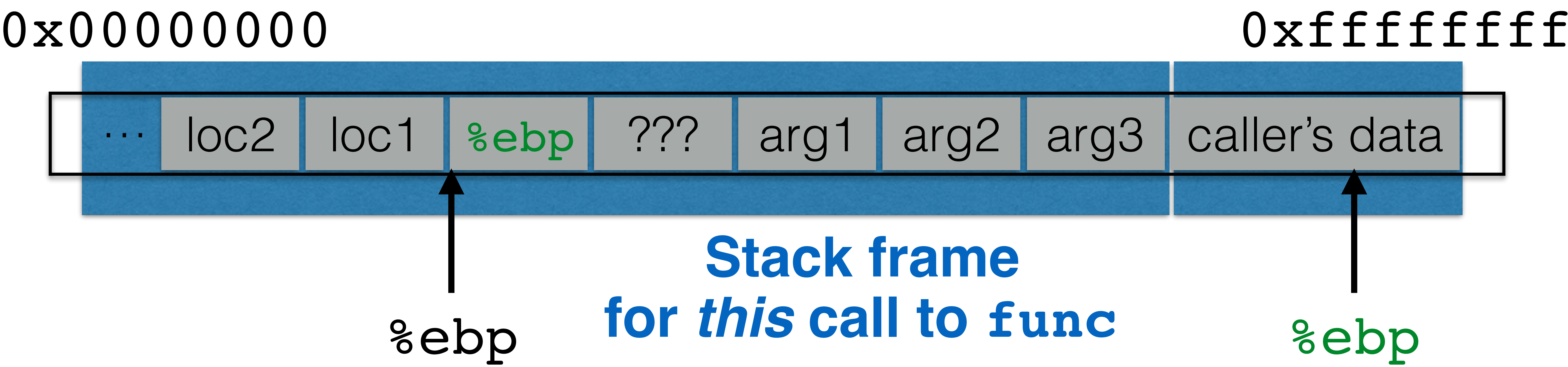
Q: How do we restore %ebp?



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

Returning from functions

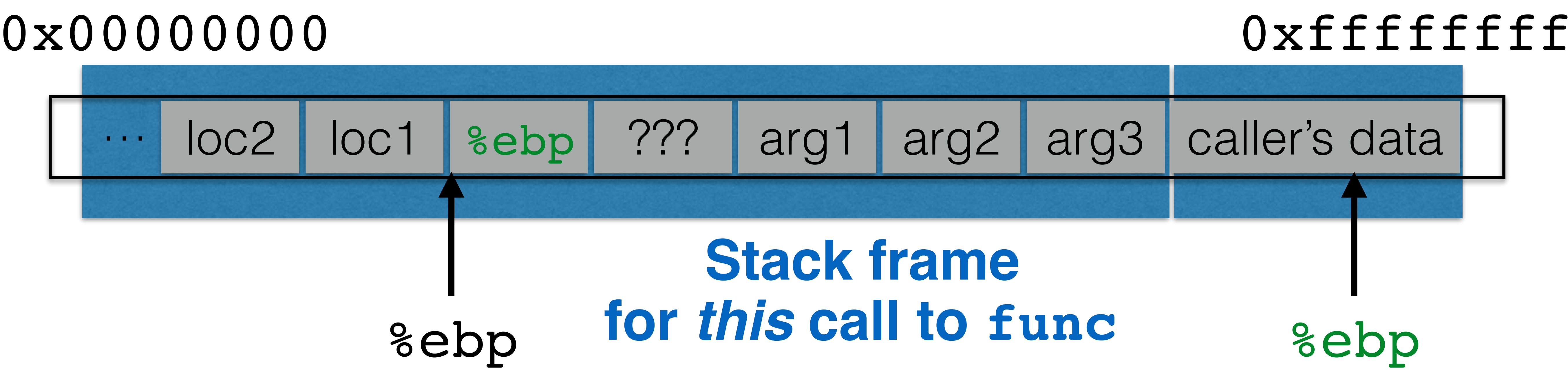
```
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}
```



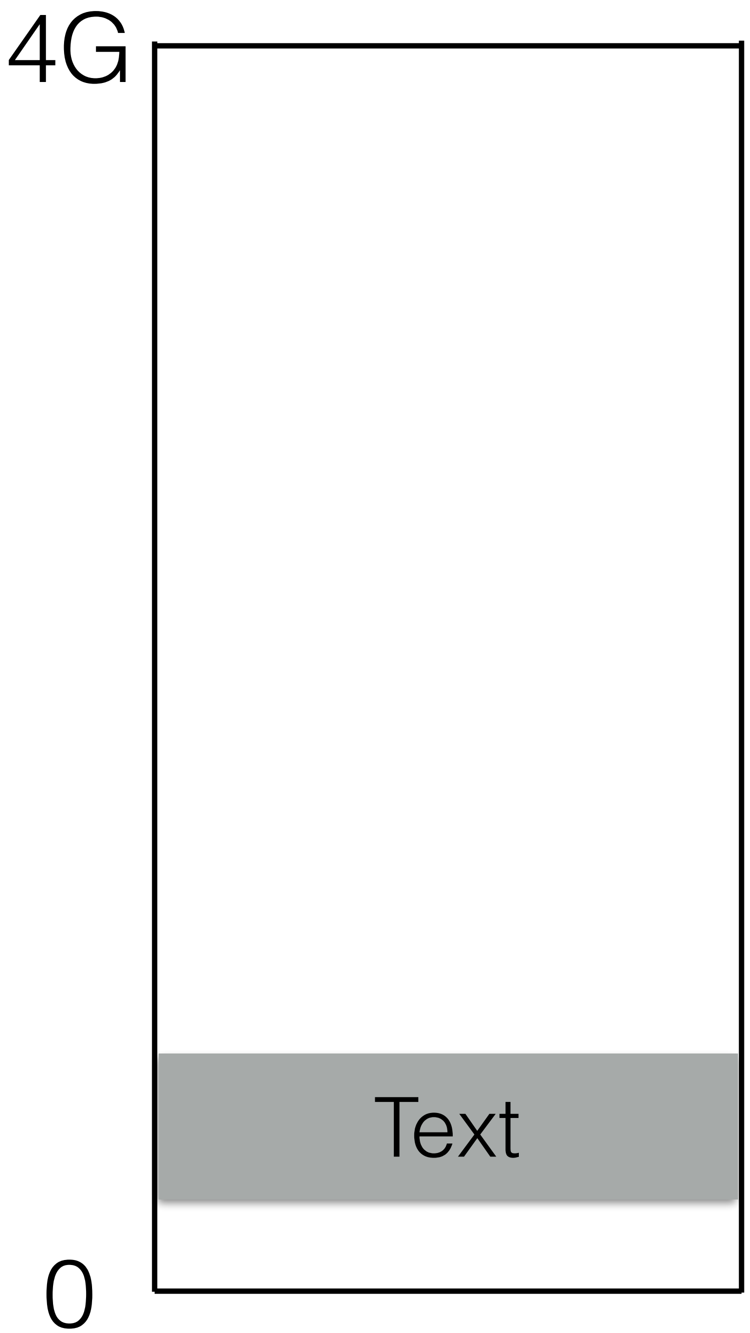
Returning from functions

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

Q: How do we resume here?



The instructions themselves are in memory

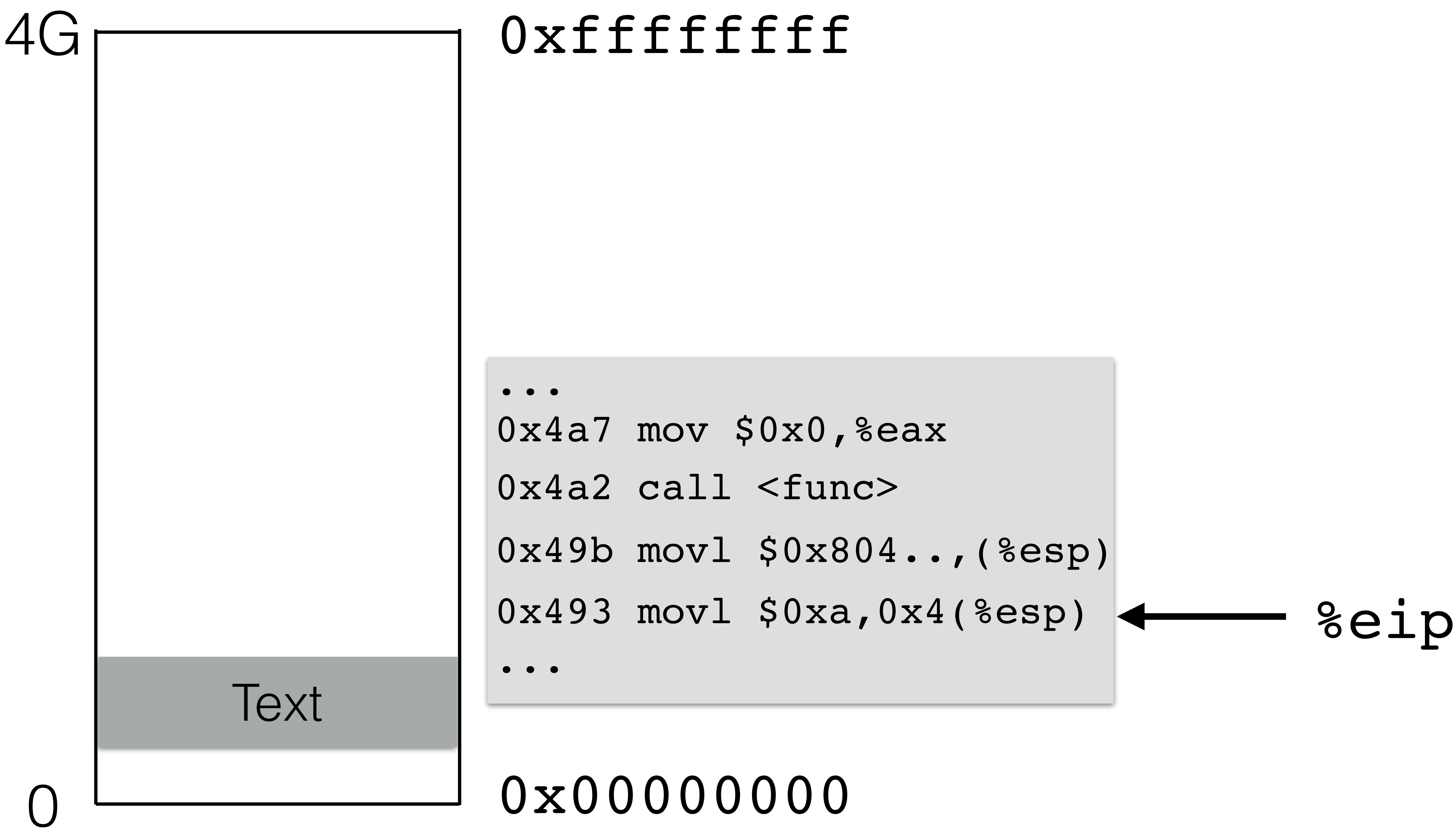


0xffffffff

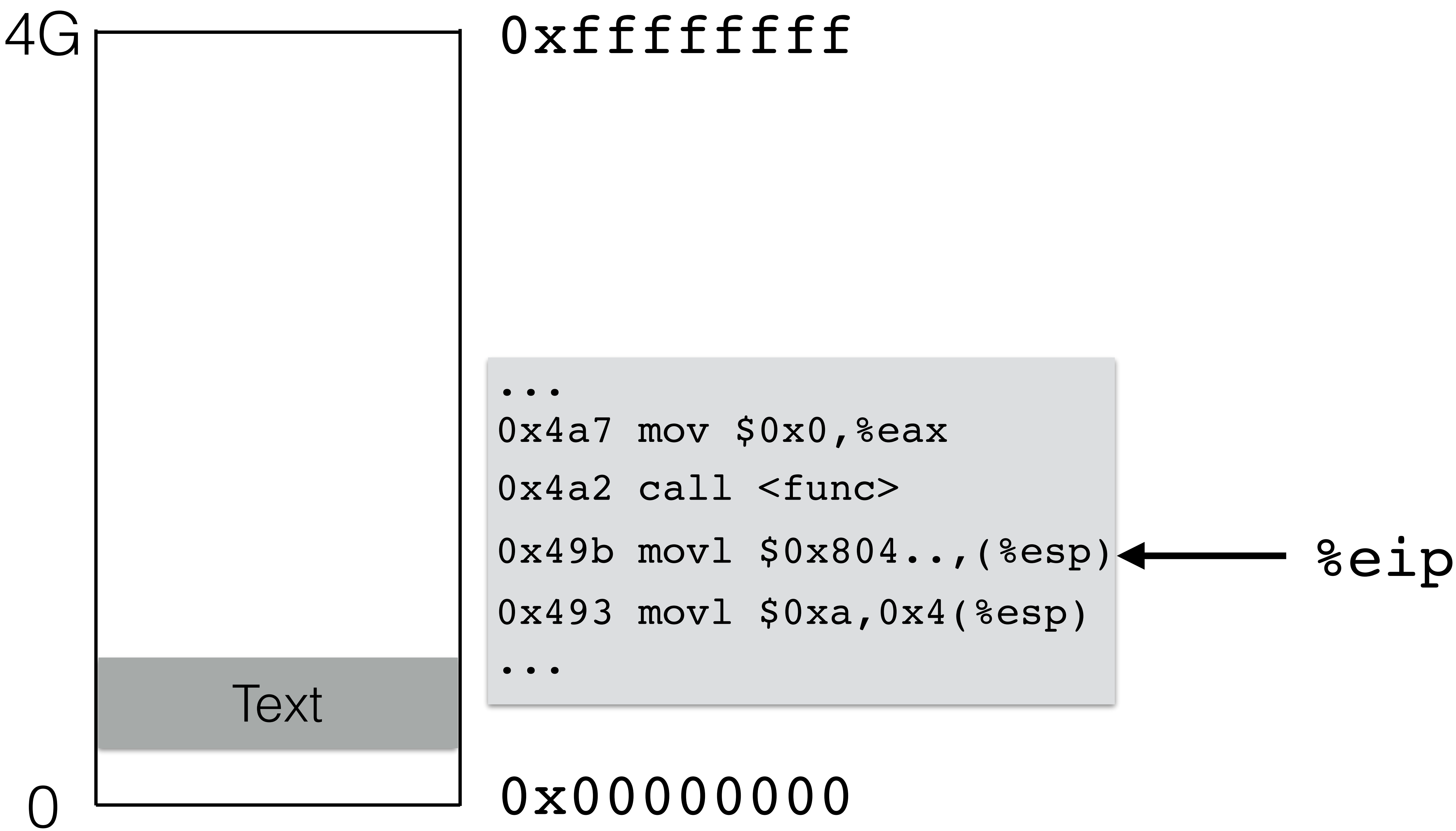
```
...
0x4a7 mov $0x0,%eax
0x4a2 call <func>
0x49b movl $0x804..,(%esp)
0x493 movl $0xa,0x4(%esp)
...
```

0x00000000

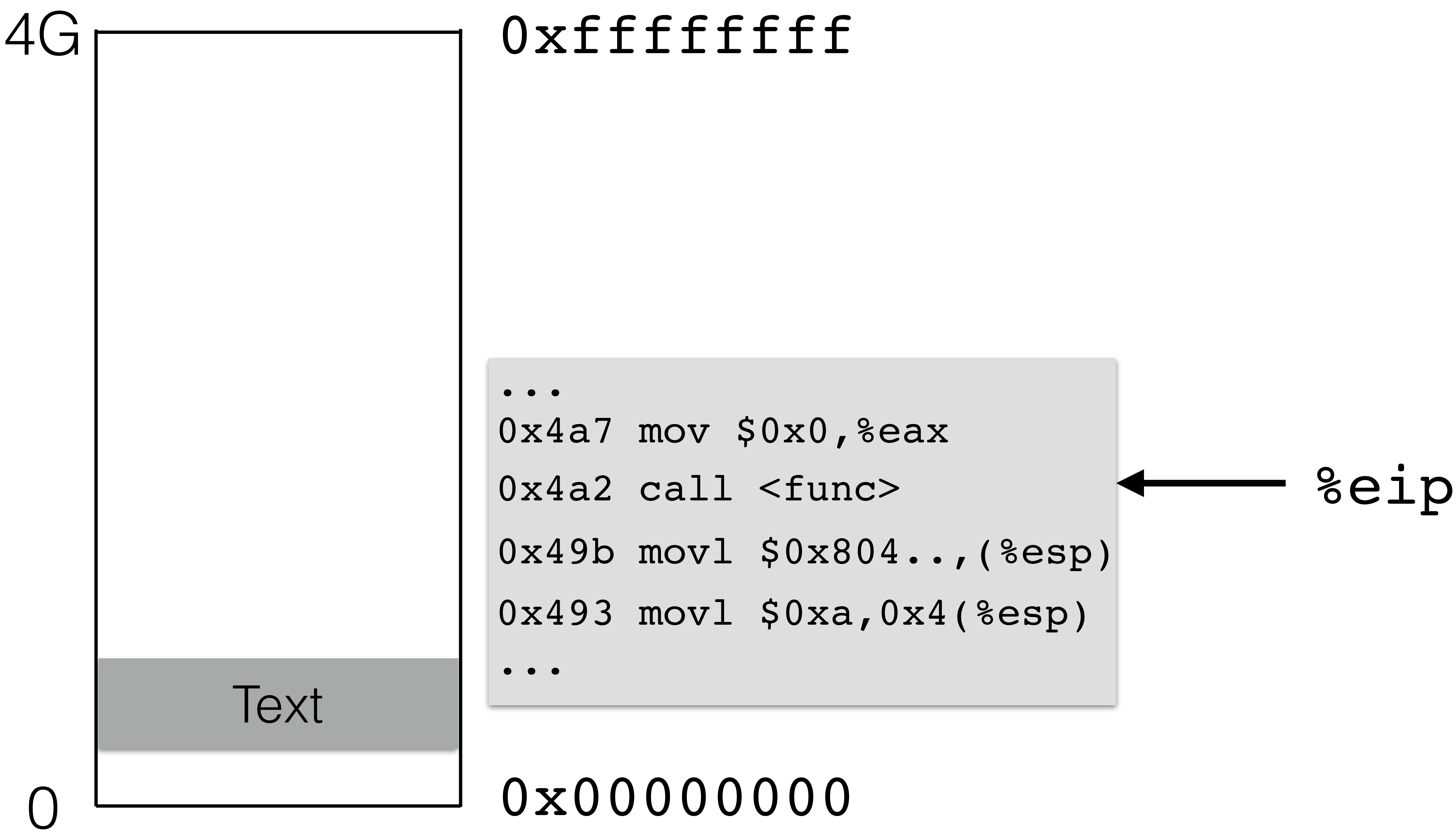
The instructions themselves are in memory



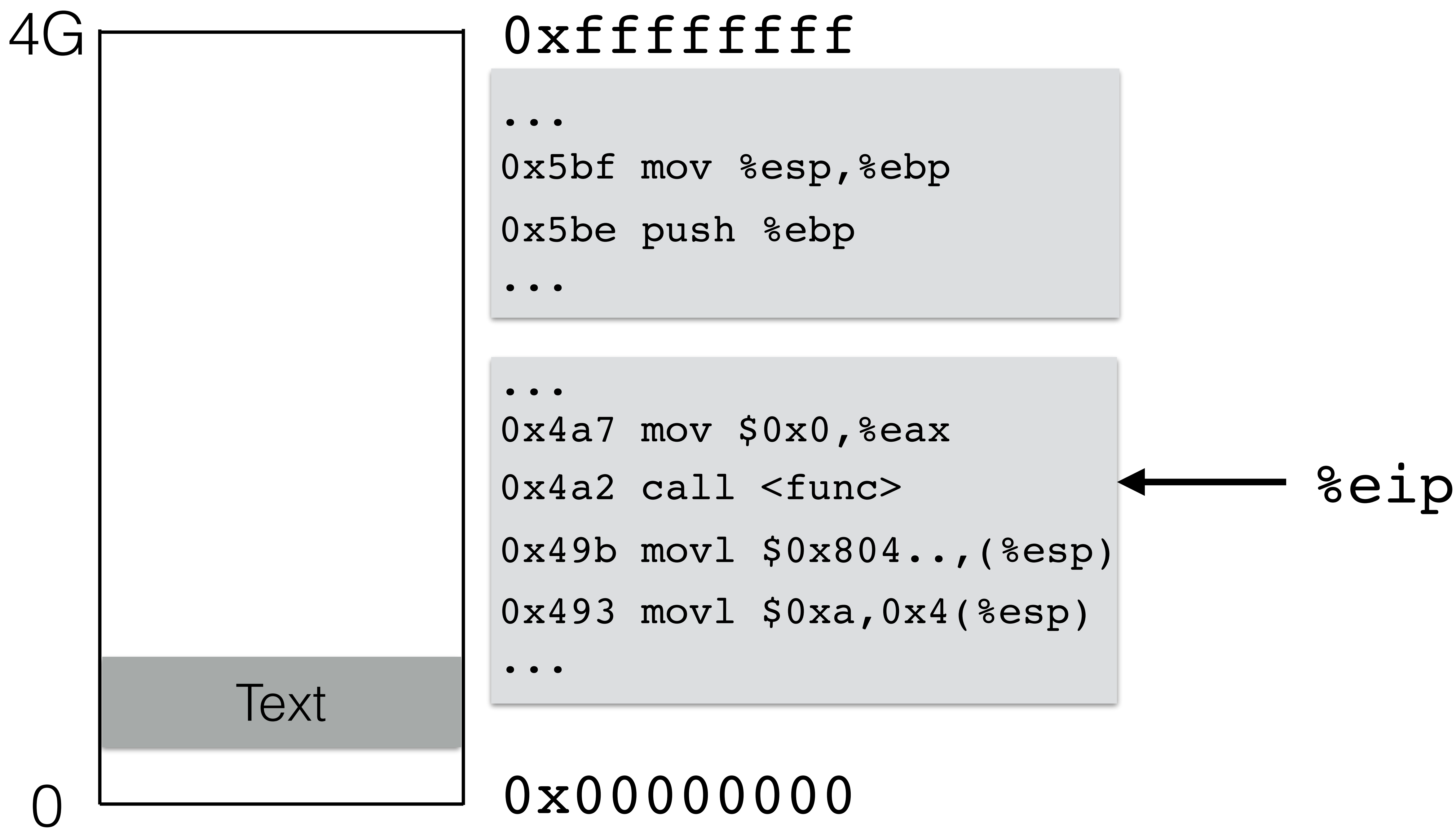
The instructions themselves are in memory



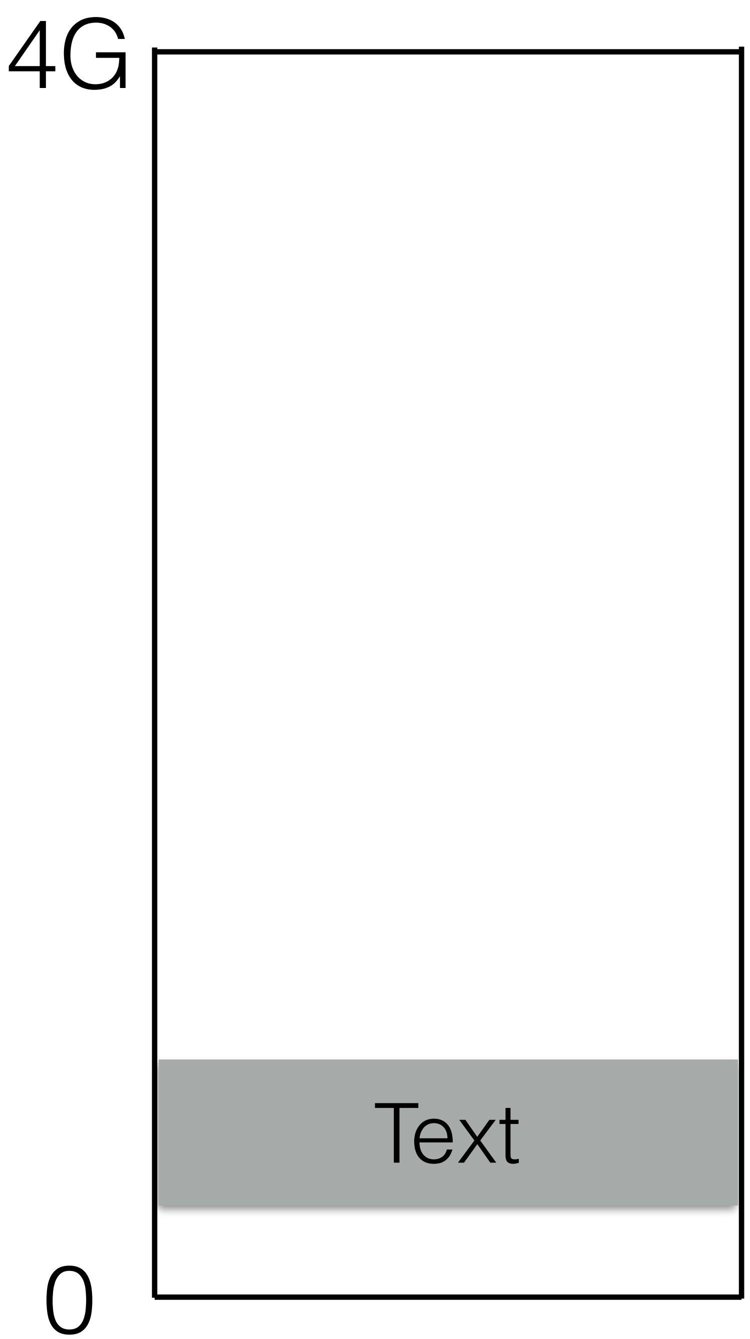
The instructions themselves are in memory



The instructions themselves are in memory



The instructions themselves are in memory



0xffffffff

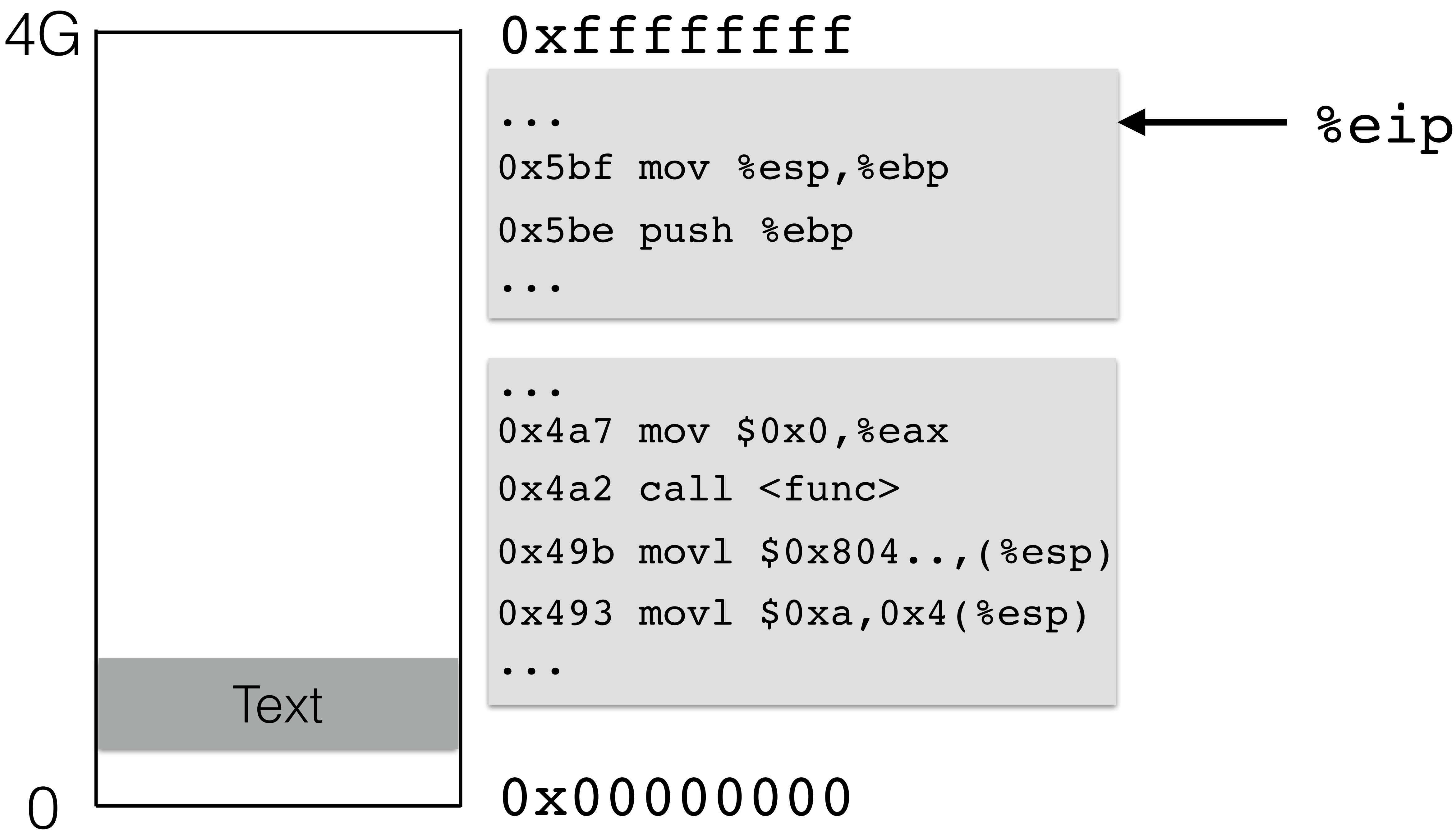
```
...
0x5bf mov %esp,%ebp
0x5be push %ebp
...
```

← %eip

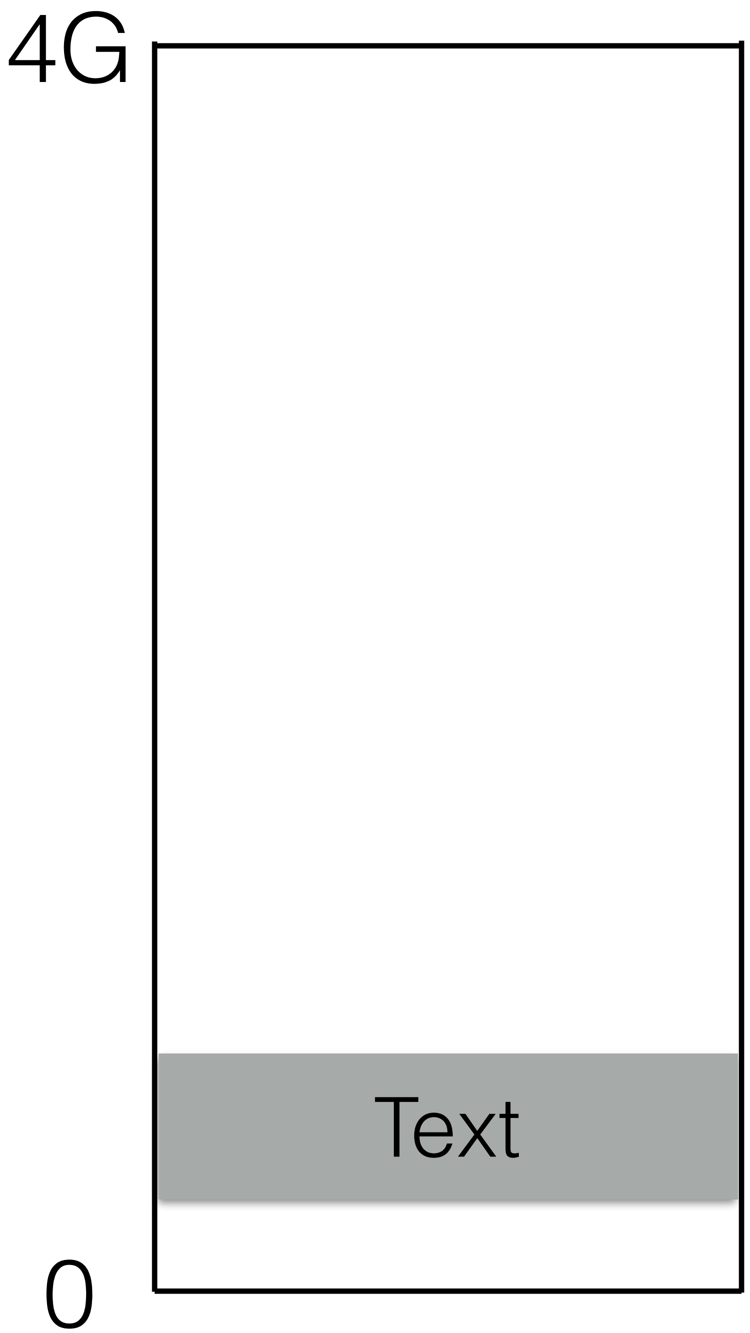
```
...
0x4a7 mov $0x0,%eax
0x4a2 call <func>
0x49b movl $0x804..,(%esp)
0x493 movl $0xa,0x4(%esp)
...
```

0x00000000

The instructions themselves are in memory



The instructions themselves are in memory

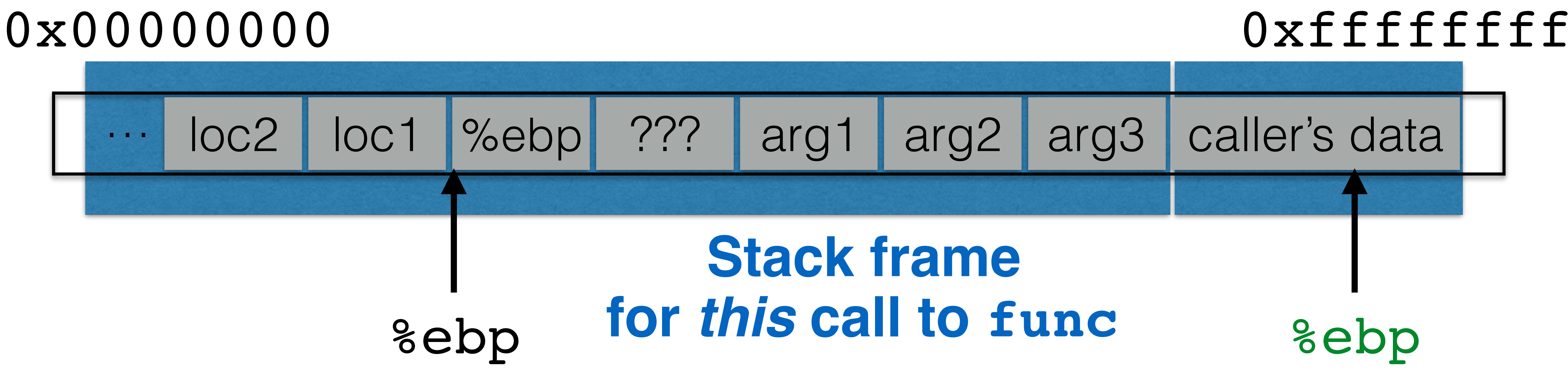



```
0xffffffff

...
0x5bf mov %esp,%ebp
0x5be push %ebp
...

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

0x00000000
```

Returning from functions

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

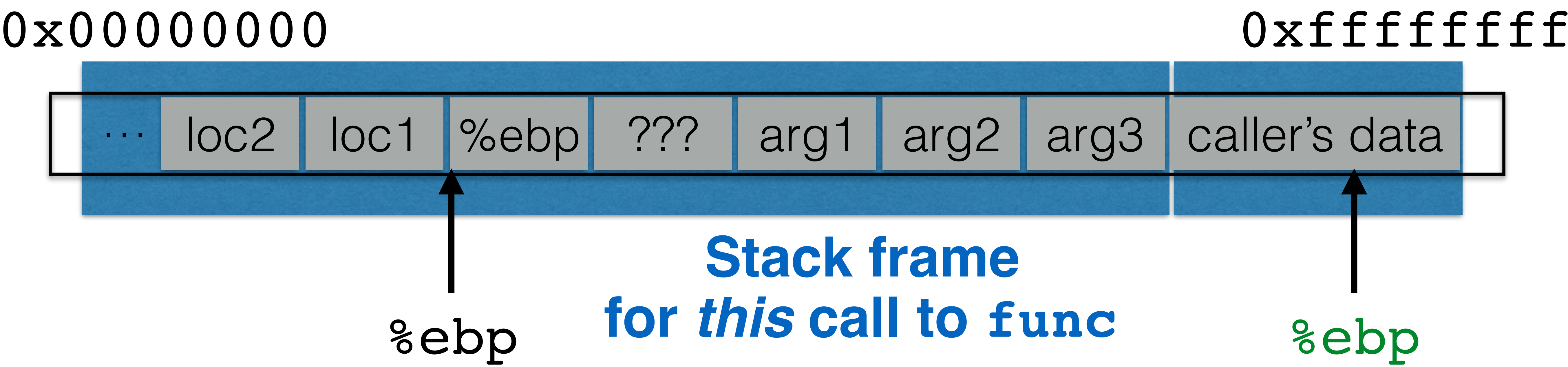
Q: How do we resume here?



Returning from functions

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

Q: How do we resume here?



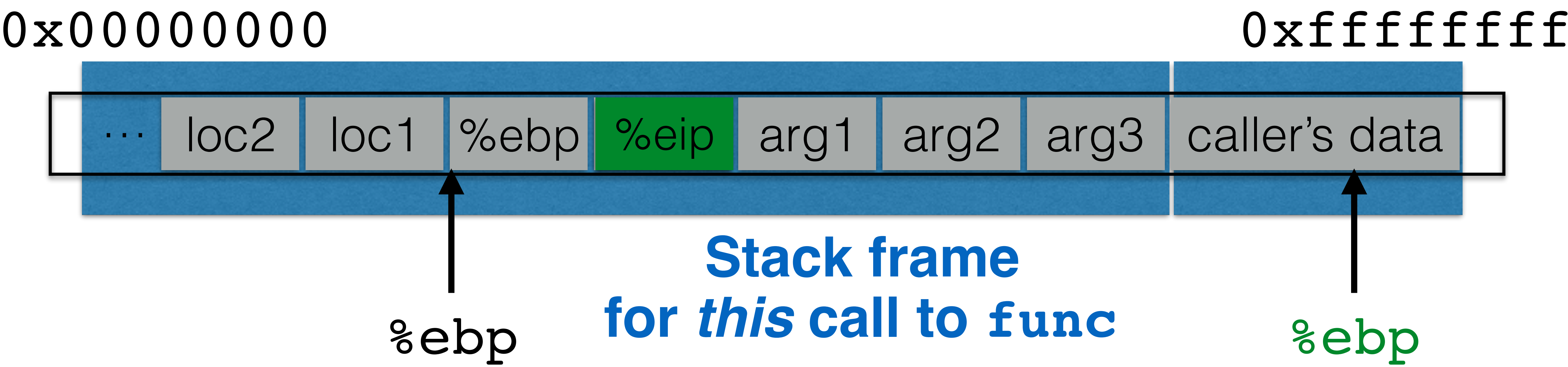
Stack frame for *this* call to `func`

Push next `%eip` before `call`

Returning from functions

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

Q: How do we resume here?

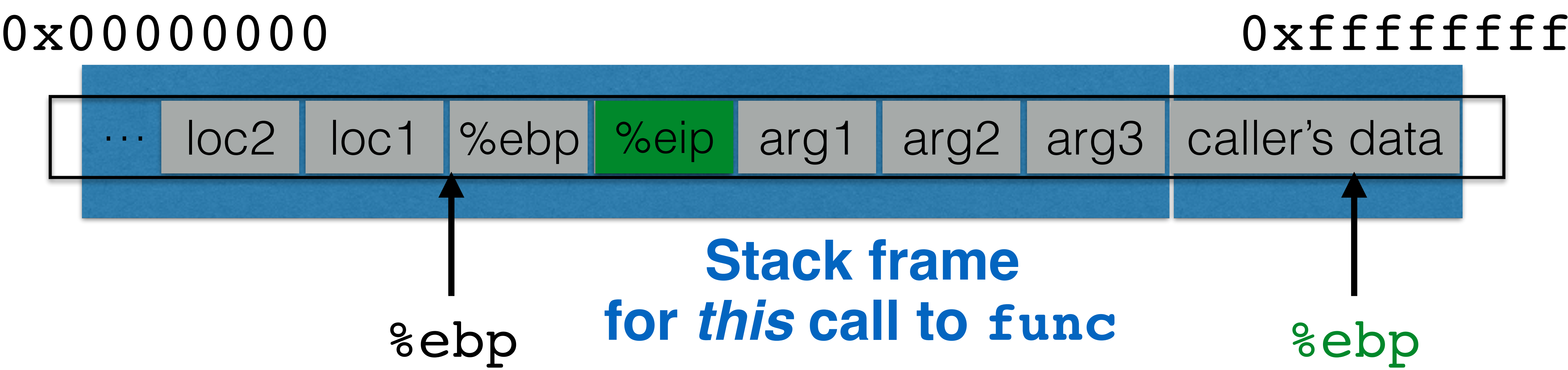


Push next `%eip` before `call`

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 and functions: Summary

Stack and 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 and 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 and functions: Summary

Calling function:

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

Called function:

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

Returning function:

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

Buffer overflows

Buffer overflows from 10,000 ft

- 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

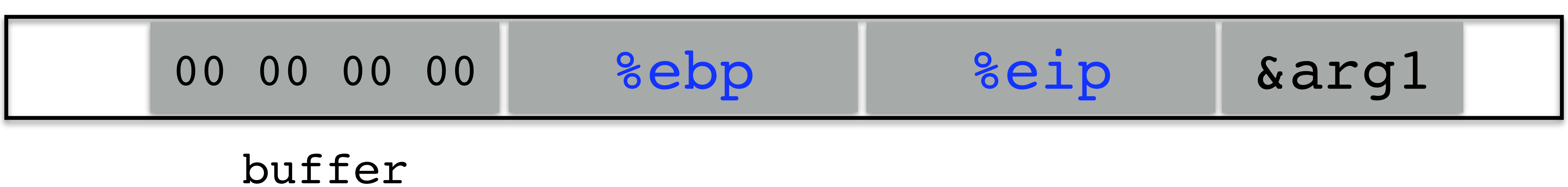
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

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);
    ...
}
```

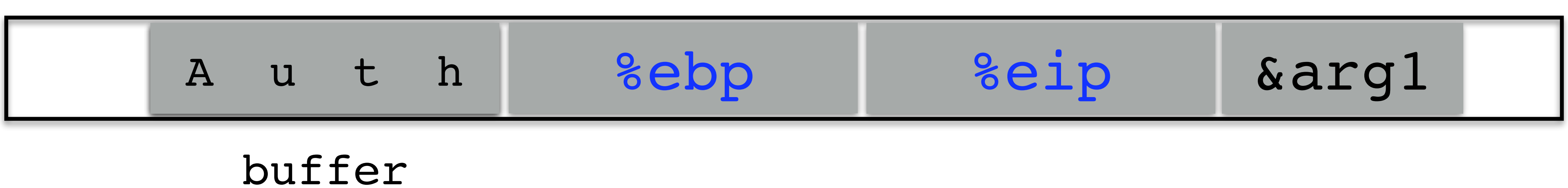
A buffer overflow example

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

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



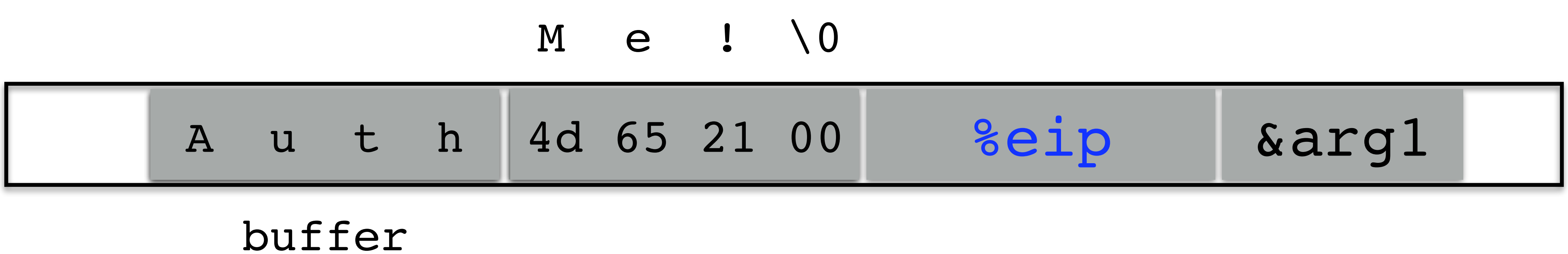
A buffer overflow example

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

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



A buffer overflow example

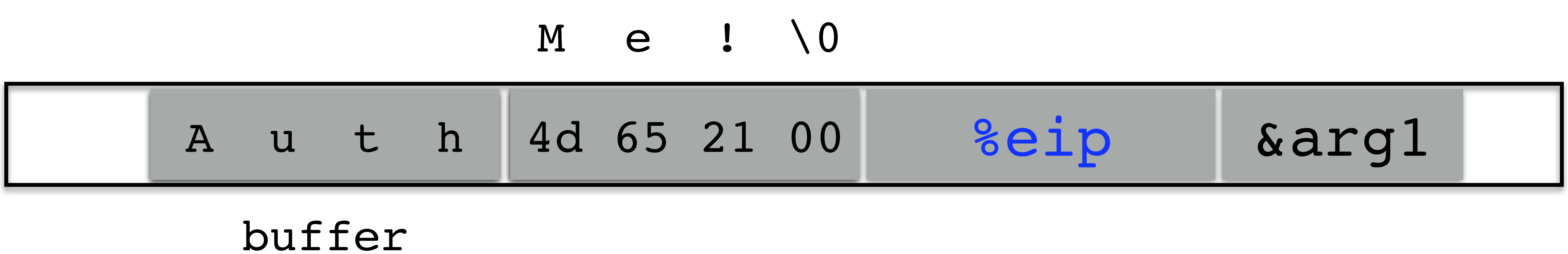
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

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);
    ...
}
```

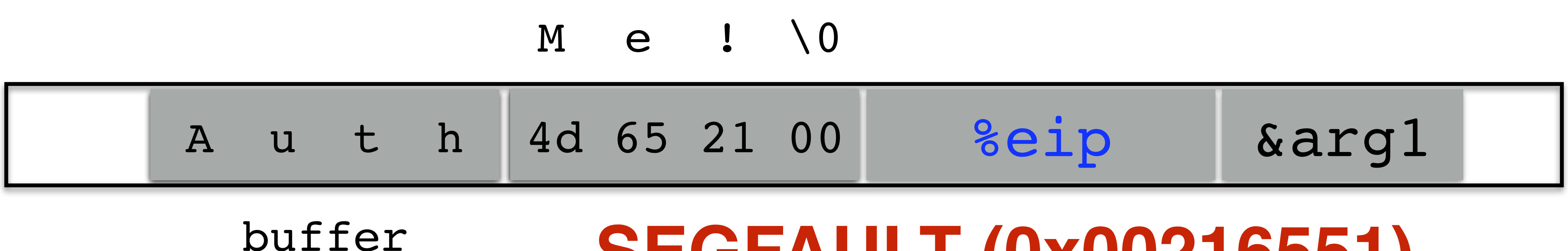
Upon return, sets `%ebp` to `0x0021654d`



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`



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

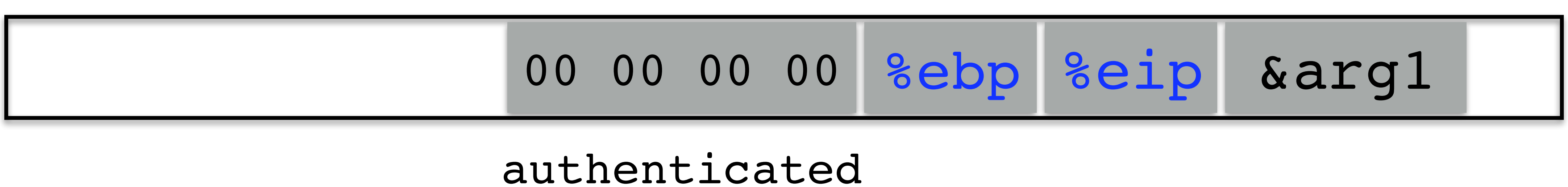
```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

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);
    ...
}
```

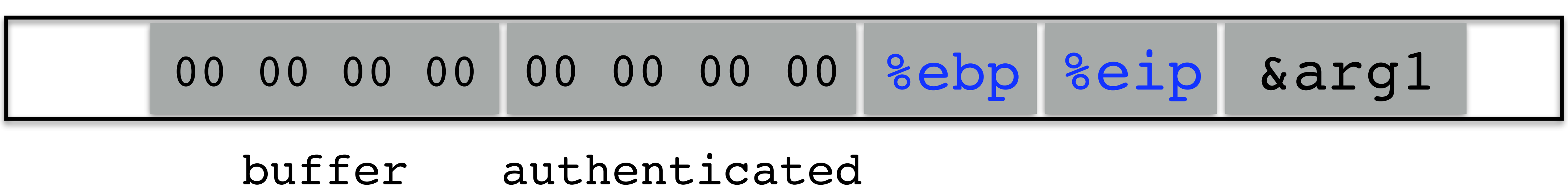
A buffer overflow example

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

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



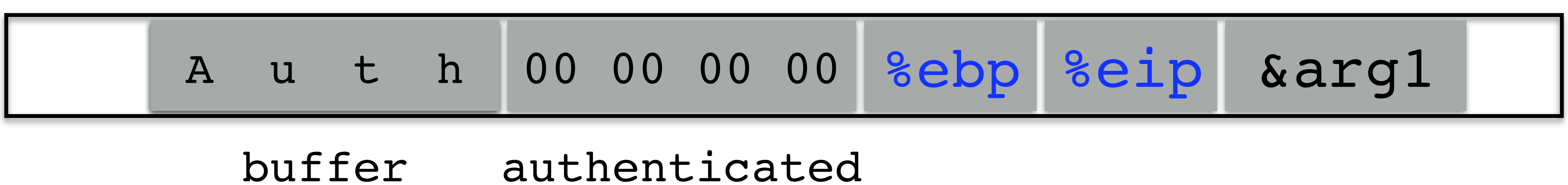
A buffer overflow example

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

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



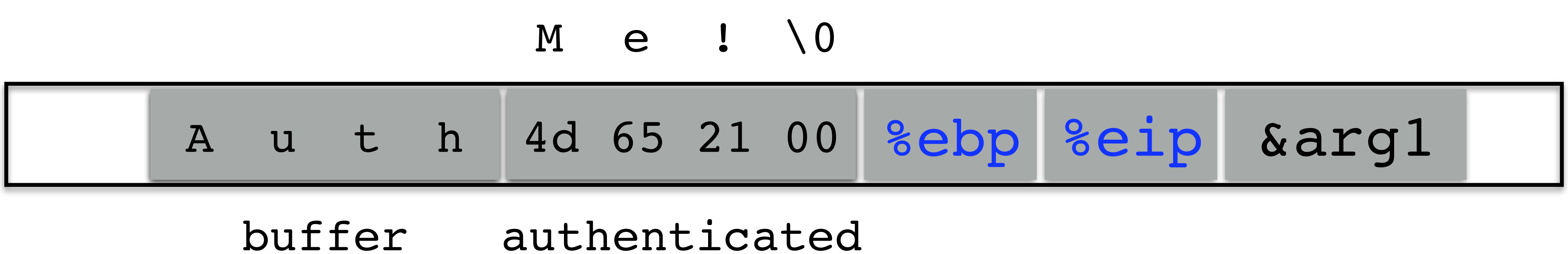
A buffer overflow example

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

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



A buffer overflow example

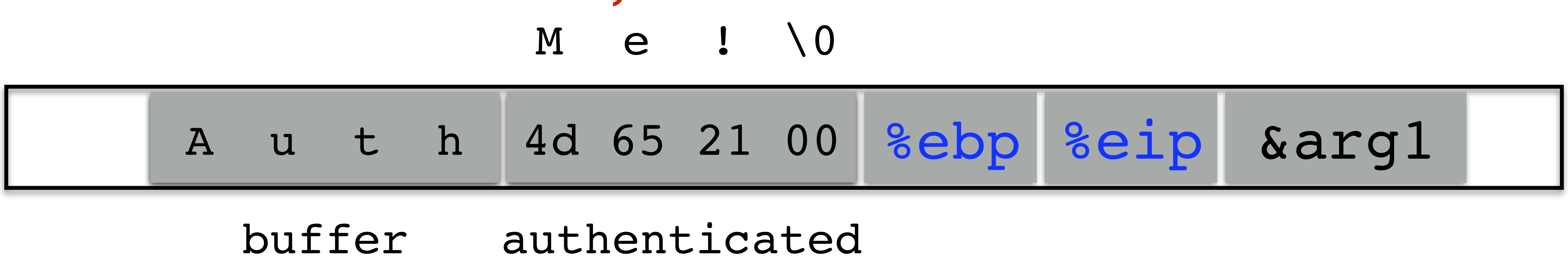
```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

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



A buffer overflow example

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

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

Code still runs; user now 'authenticated'



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

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

```
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

AA 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

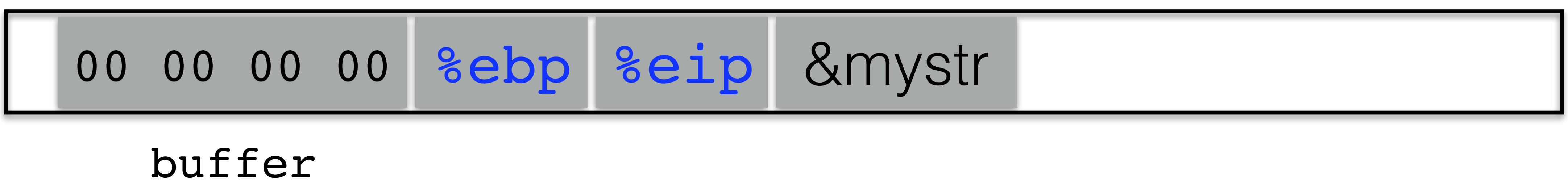
The vulnerability in IE is an invalid pointer reference, Microsoft said in security advisory 979352, which it issued on Thursday. Under certain conditions, the invalid pointer can be accessed after an object is deleted, the advisory states. In specially crafted attacks, like the ones launched against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

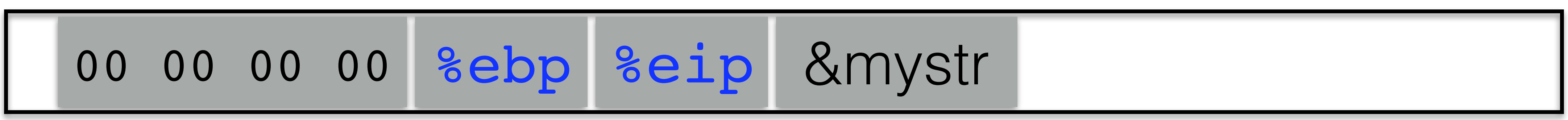
What’s the worst that could happen?

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



What's the worst that could happen?

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

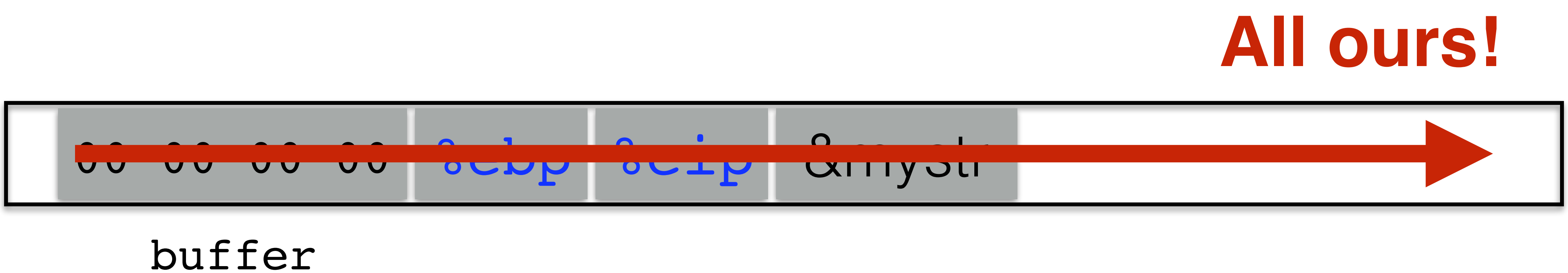


buffer

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

What’s the worst that could happen?

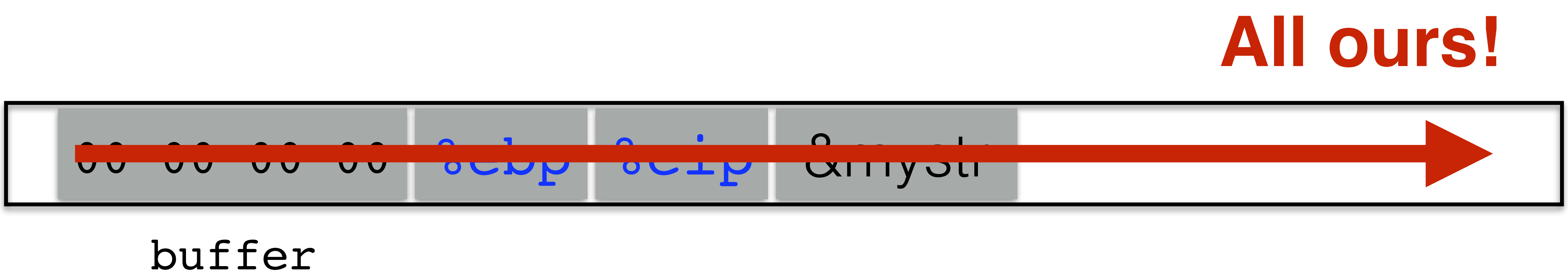
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```



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

What's the worst that could 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 and functions: Summary

Stack and functions: Summary

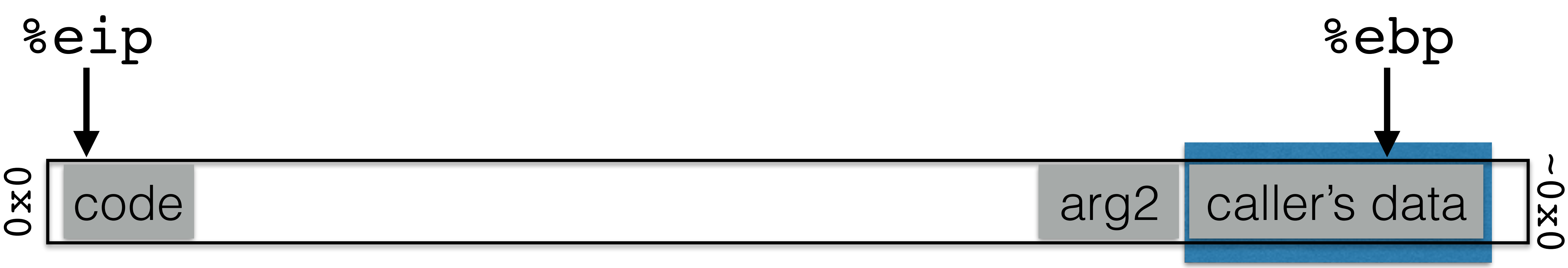
Calling function:

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 and 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 and 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 and functions: Summary

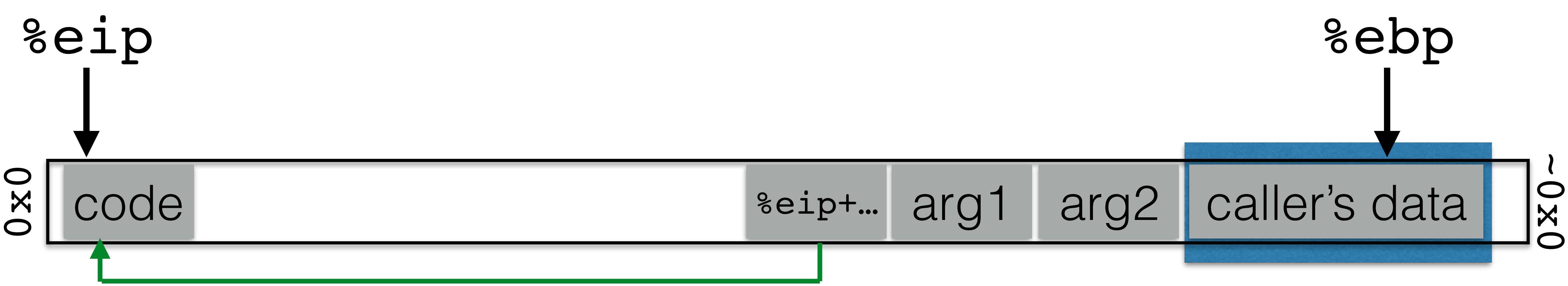
Calling function:

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 and functions: Summary

Calling function:

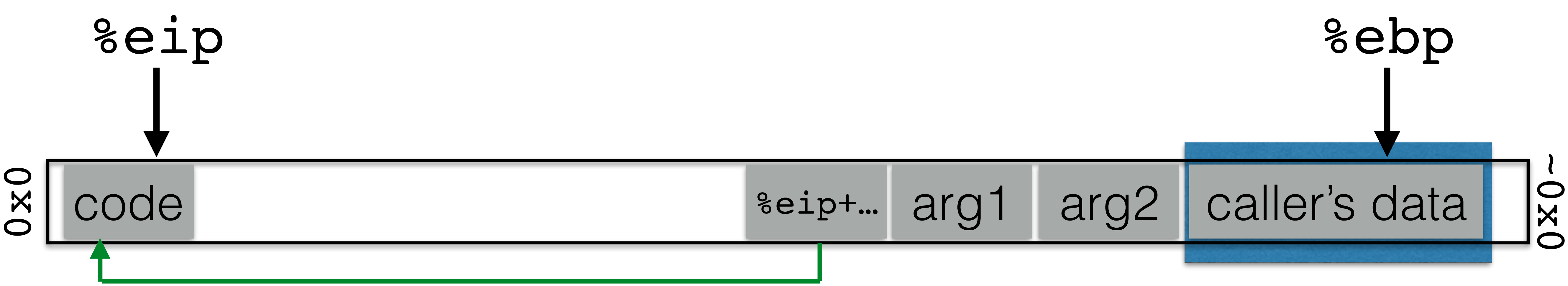
1. **Push arguments** of the function you're calling onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**



Stack and 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**



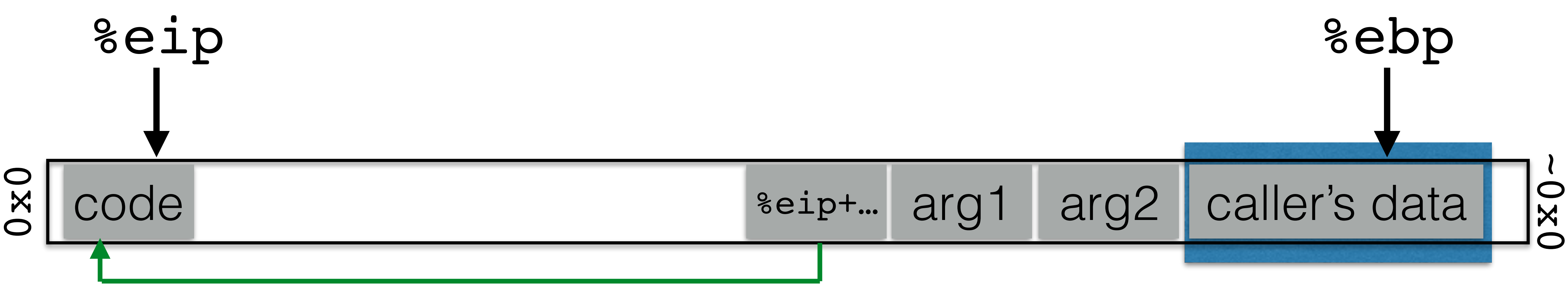
Stack and functions: Summary

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



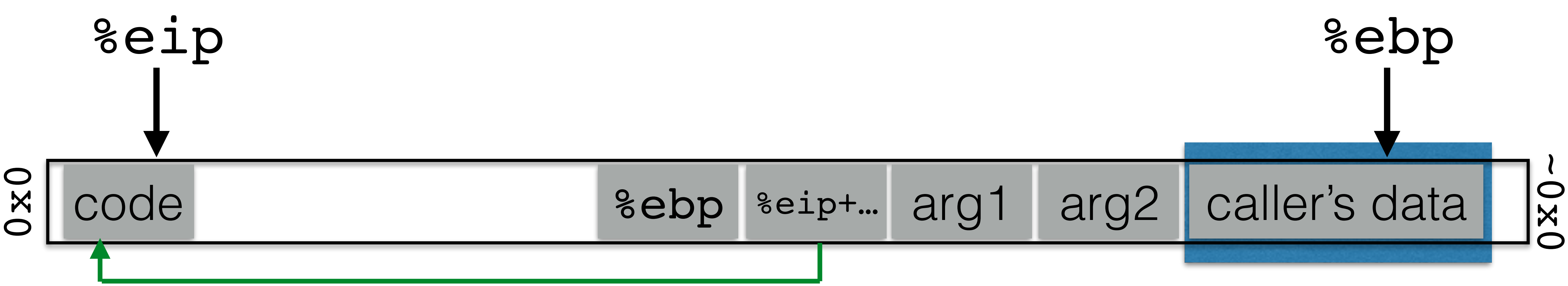
Stack and functions: Summary

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



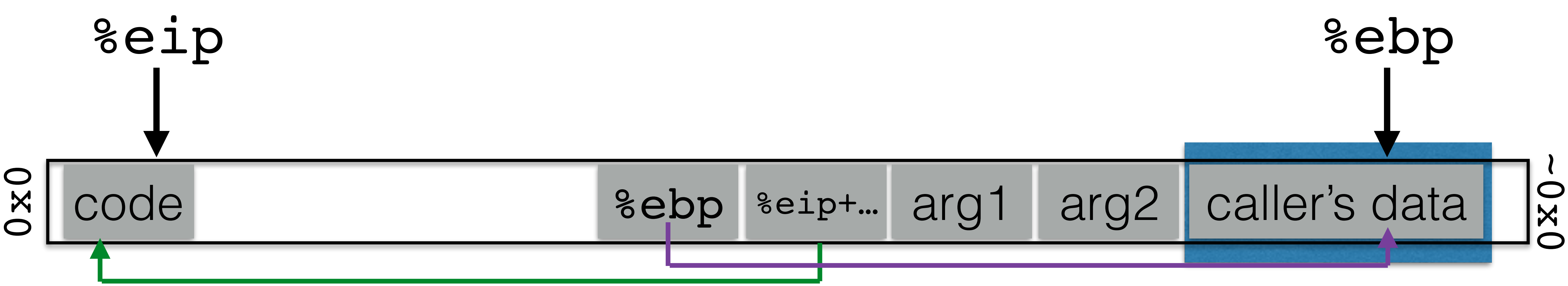
Stack and functions: Summary

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



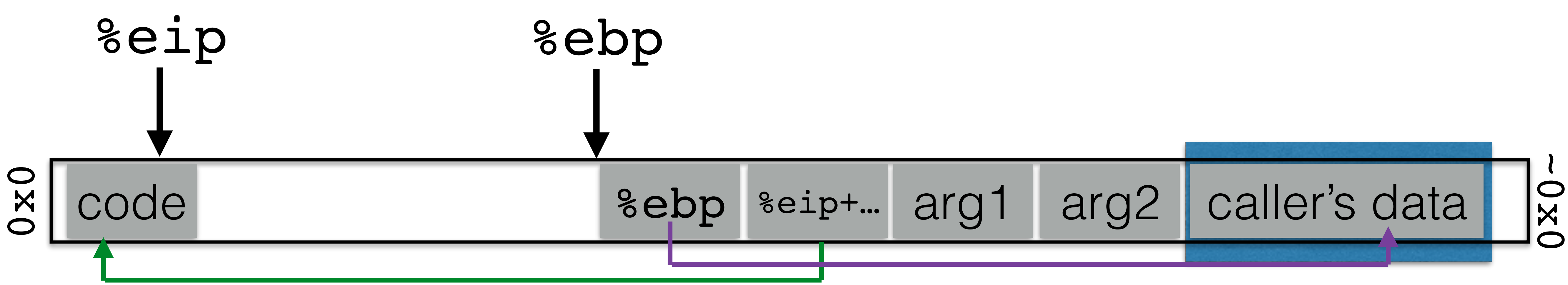
Stack and functions: Summary

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



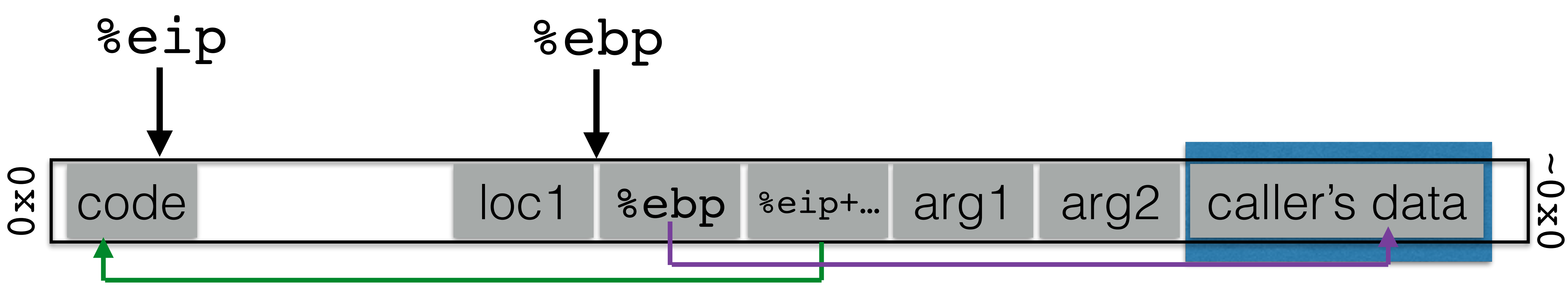
Stack and functions: Summary

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



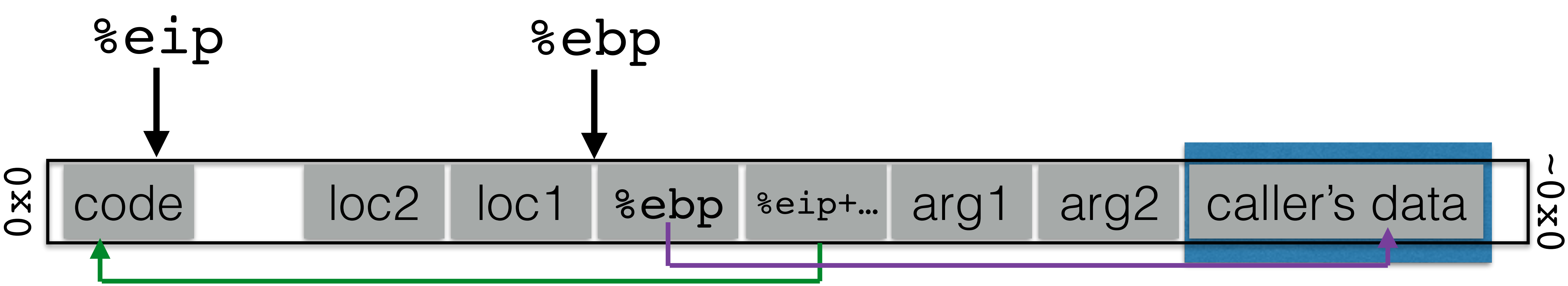
Stack and functions: Summary

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 and functions: Summary

Calling function:

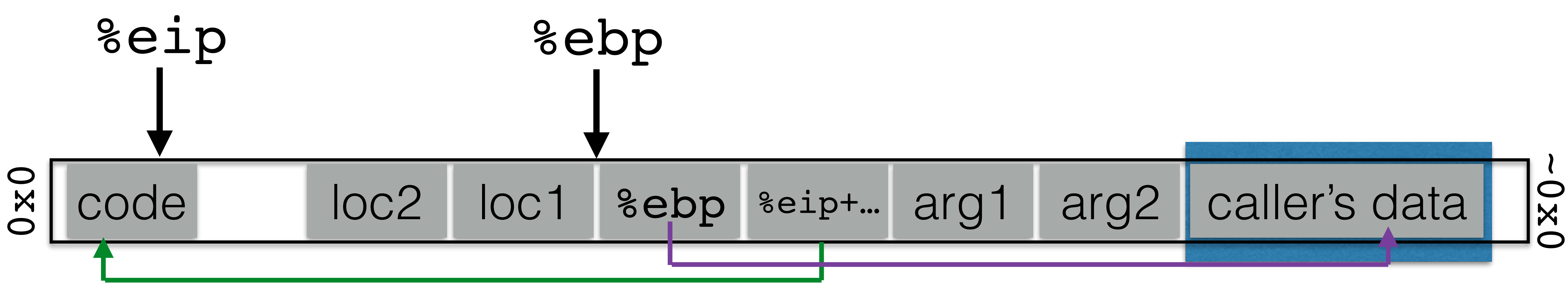
1. **Push arguments** of the function you're calling onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

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 and functions: Summary

Calling function:

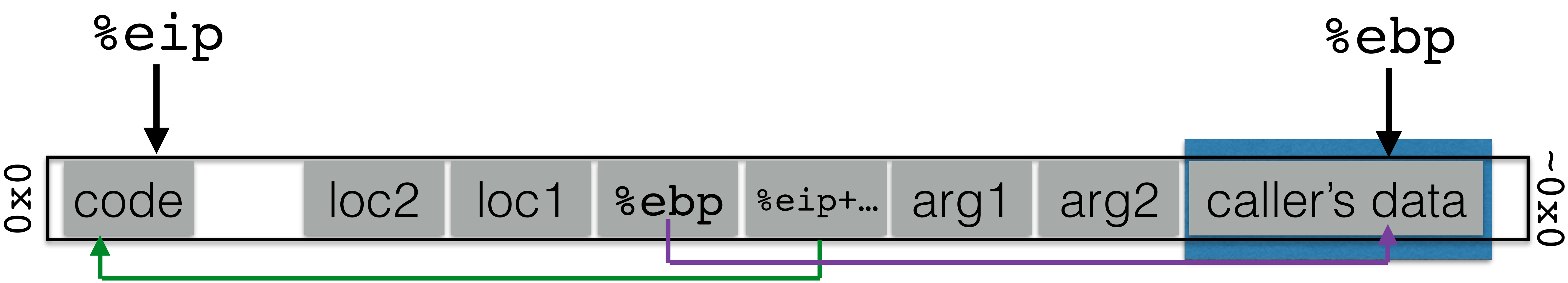
1. **Push arguments** of the function you're calling onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

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 and functions: Summary

Calling function:

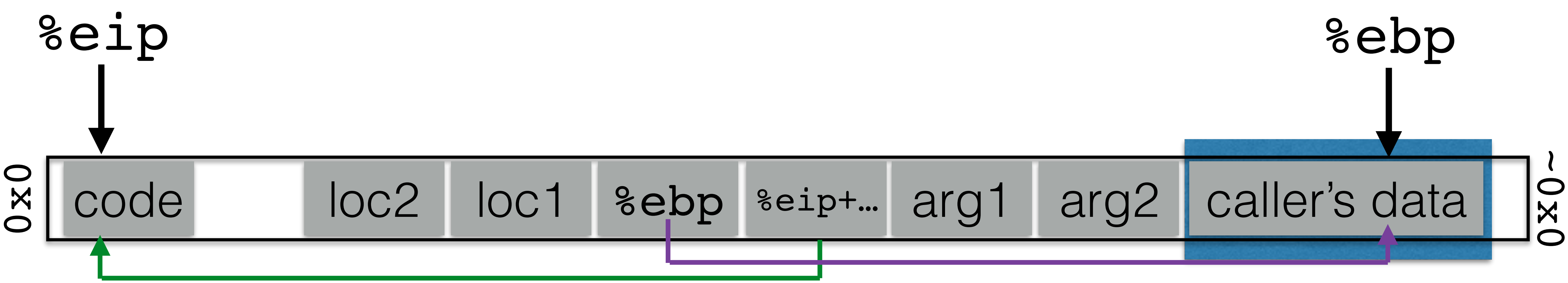
1. **Push arguments** of the function you're calling onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: the current %eip + (some amount)
3. **Jump to the address of the function you are calling**

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:

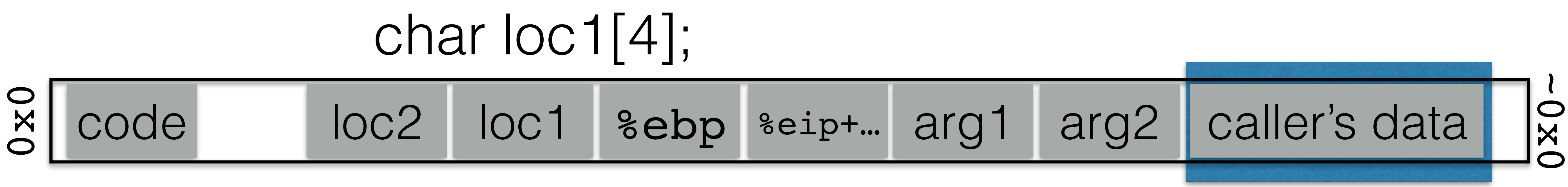
7. **Reset the previous stack frame**: %ebp = (%ebp)
8. **Jump back to return address**: %eip = 4(%ebp)



gdb: your new best friend

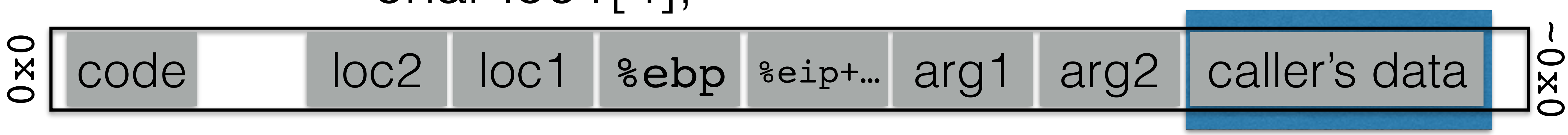
`i f`	Show **i**nfo about the current **f**rame (prev. frame, locals/args, %ebp/%eip)
`i r`	Show **info** about **reg**isters (%ebp, %eip, %esp, etc.)
`x/<n> <addr>`	E**x**amine <n> bytes of memory starting at address <addr>
`b <function>` `s`	Set a **b**reakpoint at <function> **s**tep through execution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

Buffer overflow



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

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

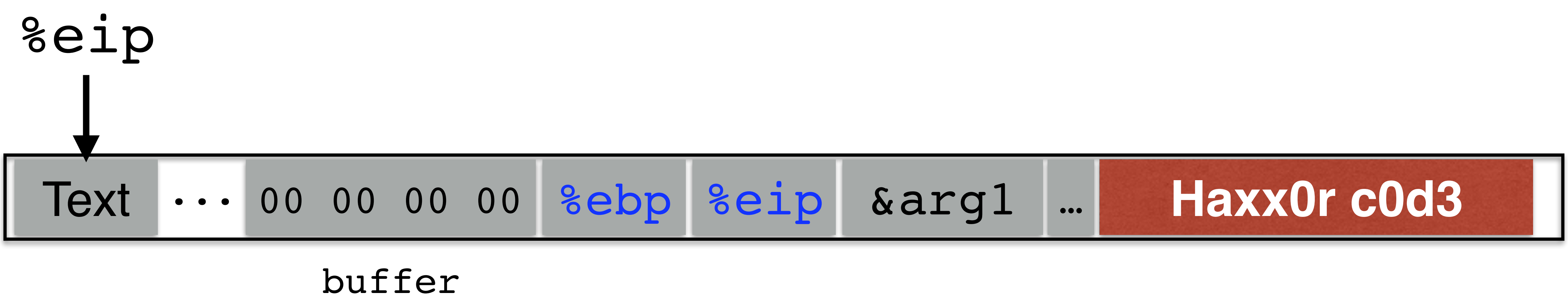
High-level idea

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

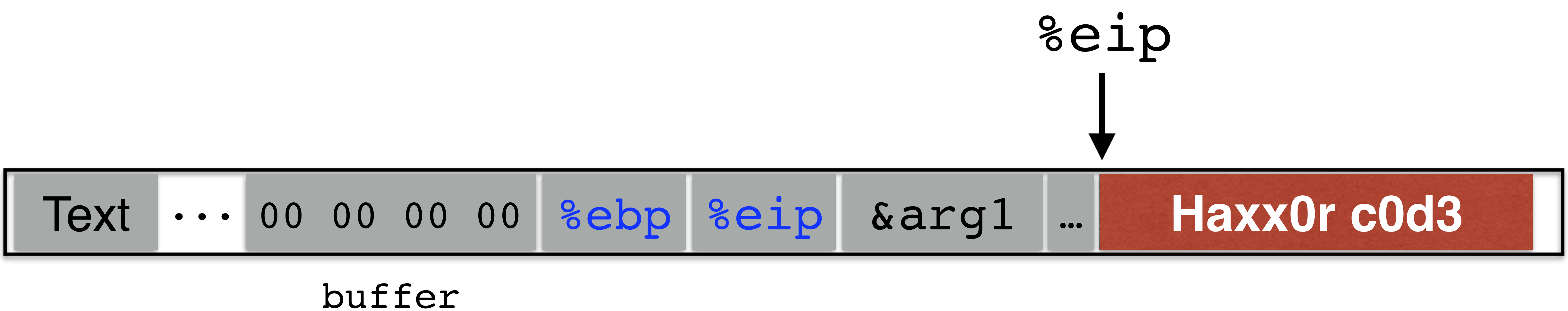


(1) Load our own code into memory

(2) Somehow get `%eip` to point to it

High-level idea

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

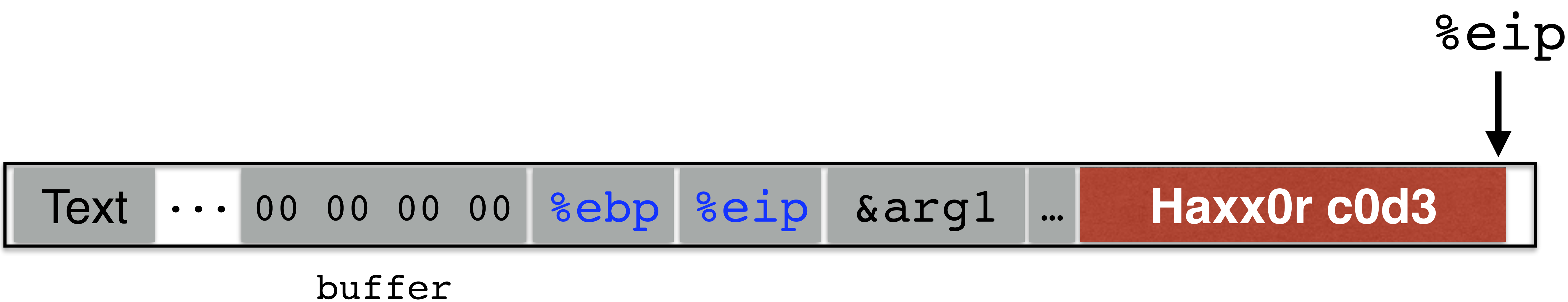


(1) Load our own code into memory

(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

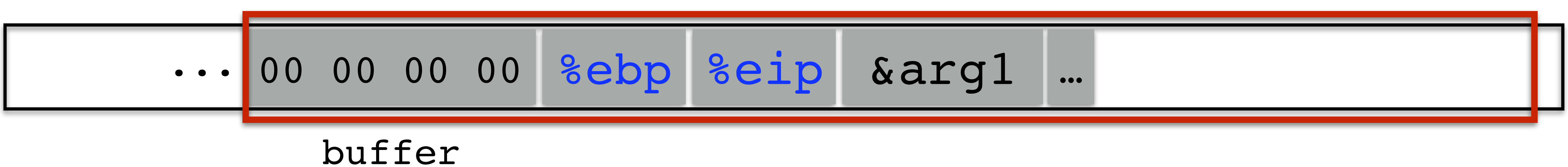
- ***All we can do is write to memory from `buffer` onward***
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running

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

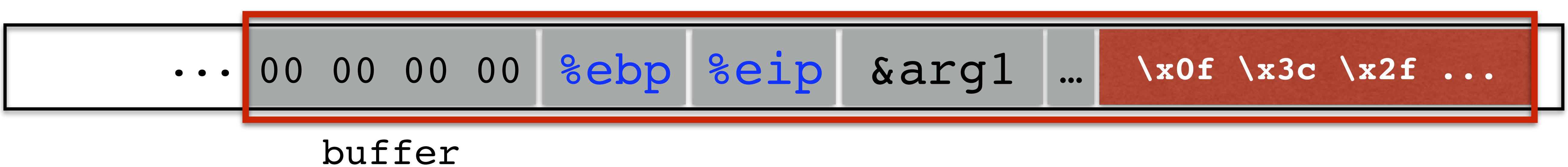


Thoughts?

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

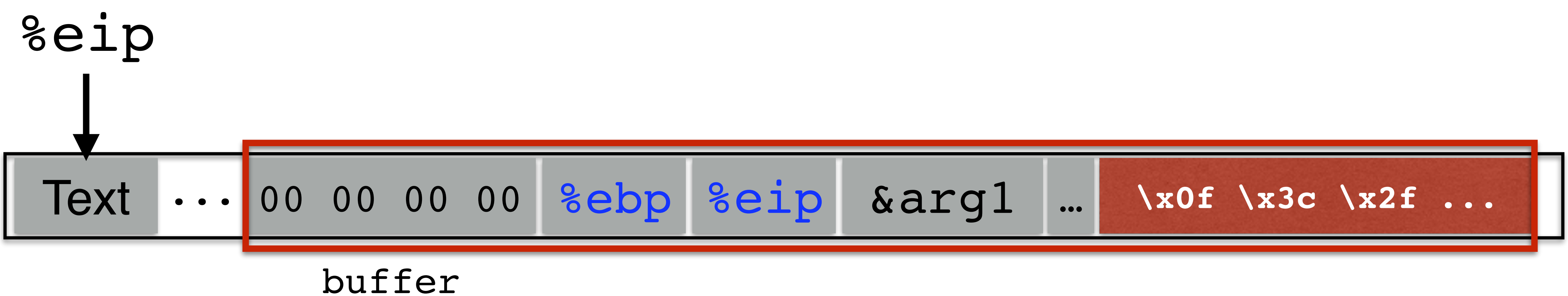


Thoughts?

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

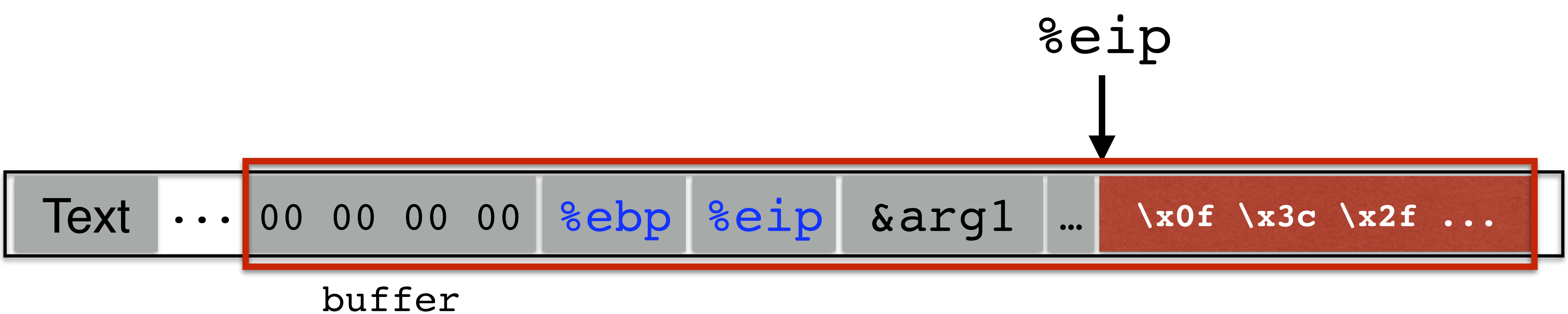


Thoughts?

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

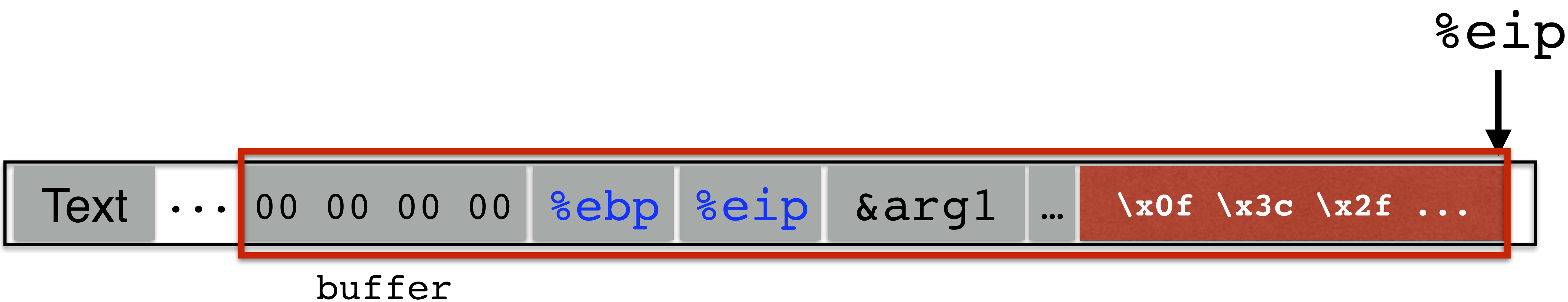


Thoughts?

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 and 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**

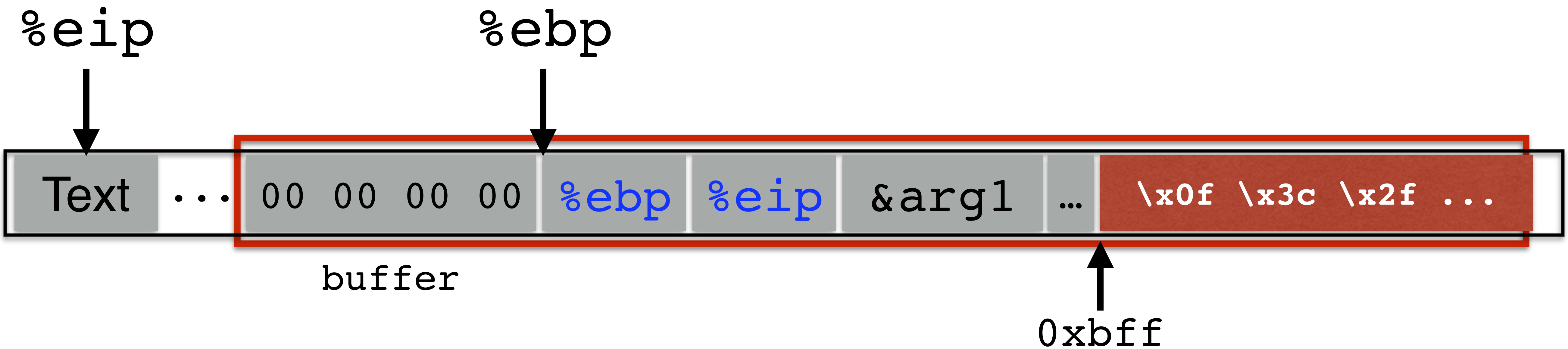
Called function:

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

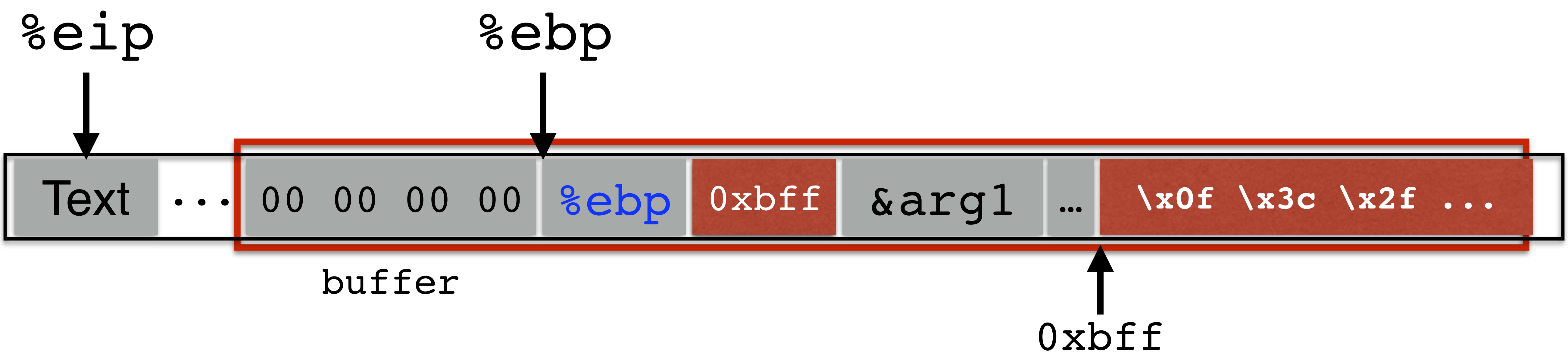
Returning function:

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

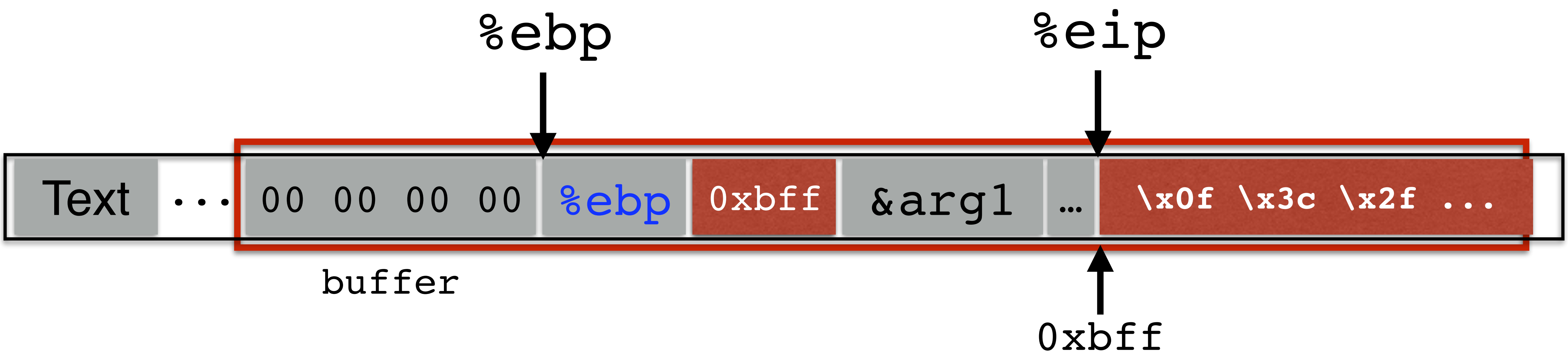
Hijacking the saved `%eip`



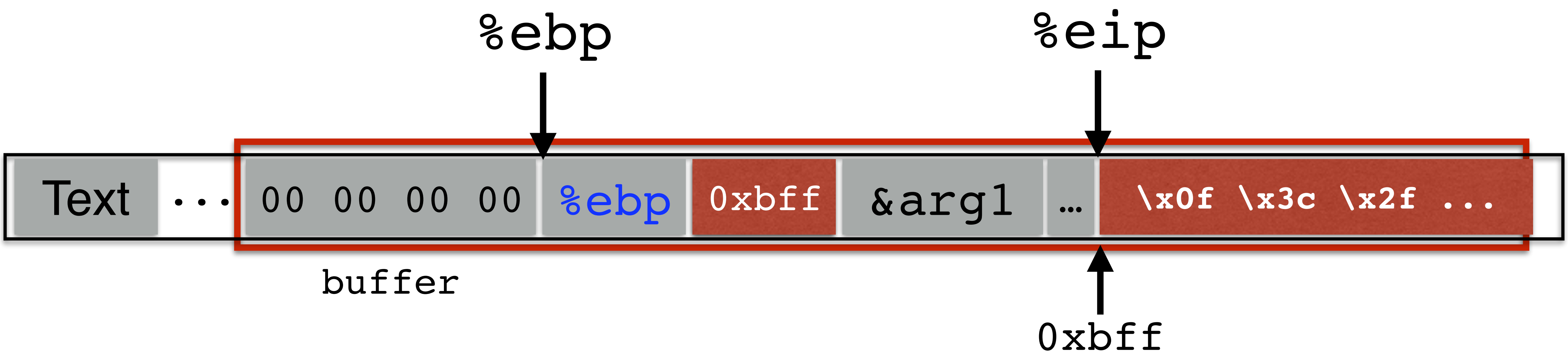
Hijacking the saved `%eip`



Hijacking the saved `%eip`



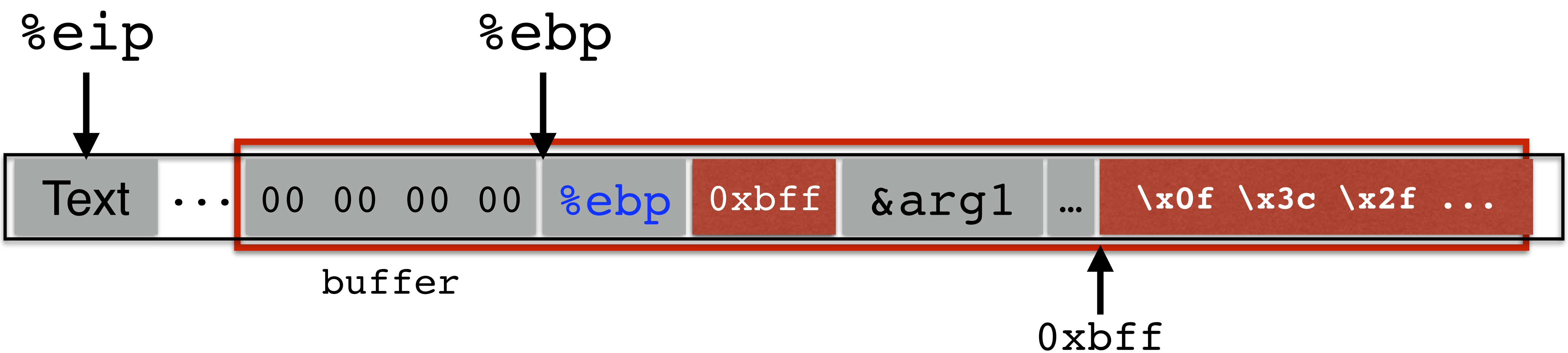
Hijacking the saved `%eip`



But how do we know the address?

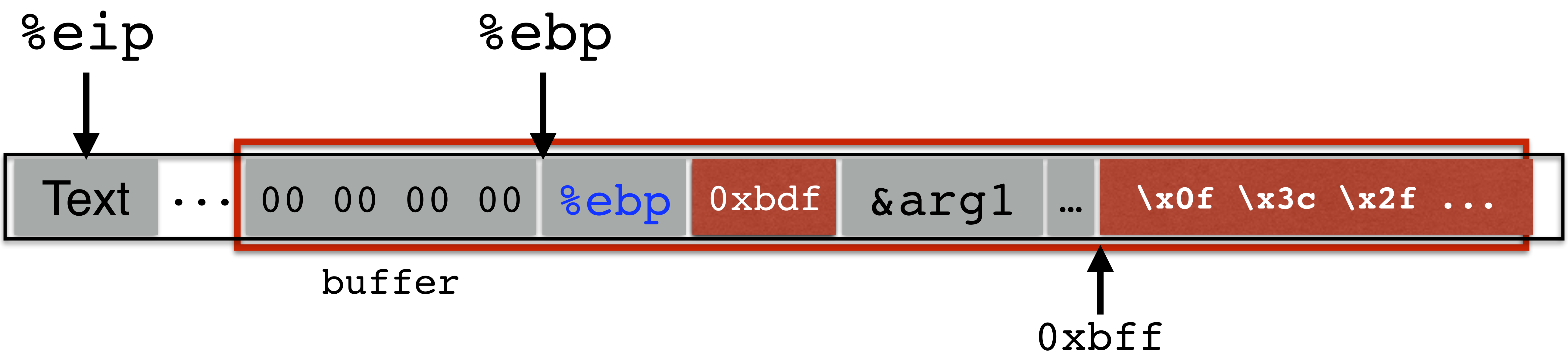
Hijacking the saved `%eip`

What if we are wrong?



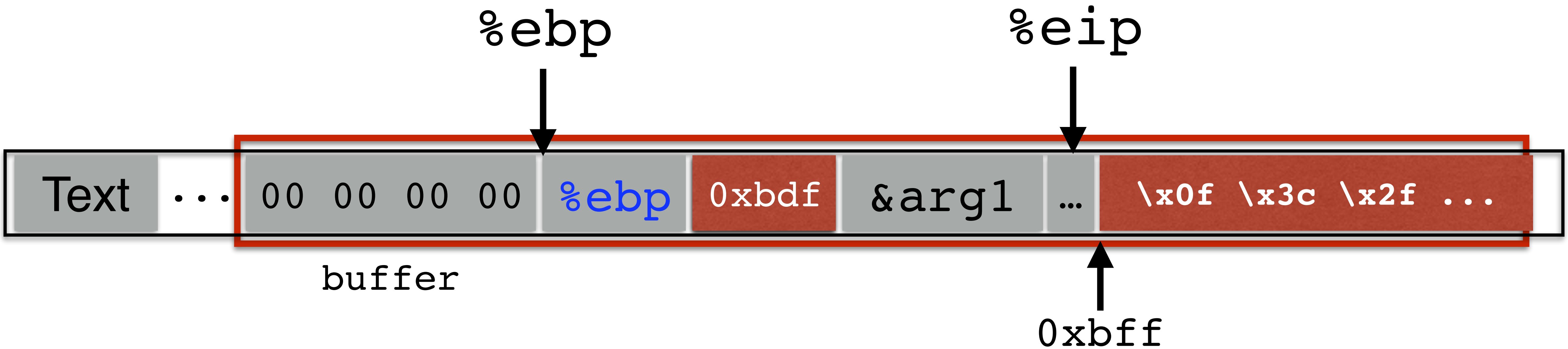
Hijacking the saved `%eip`

What if we are wrong?



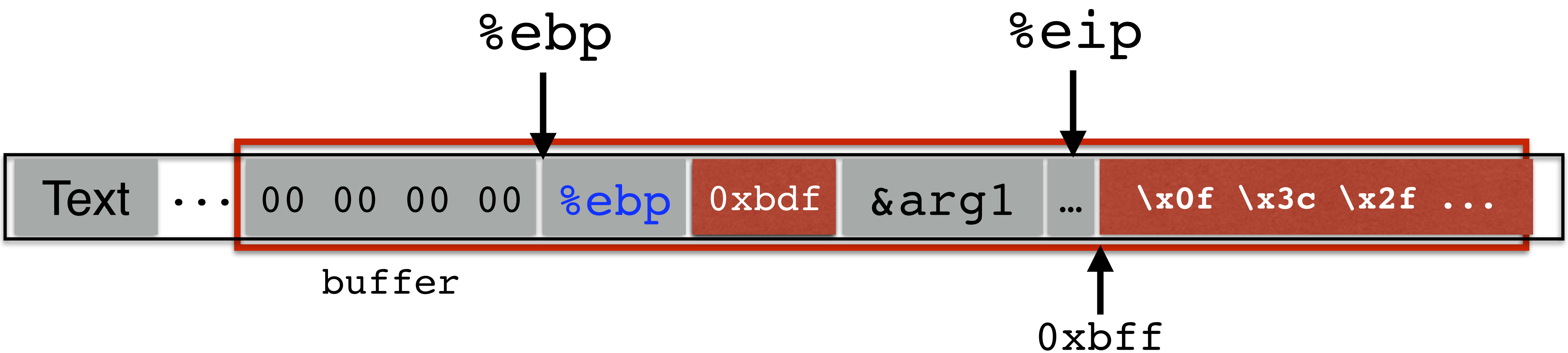
Hijacking the saved `%eip`

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`

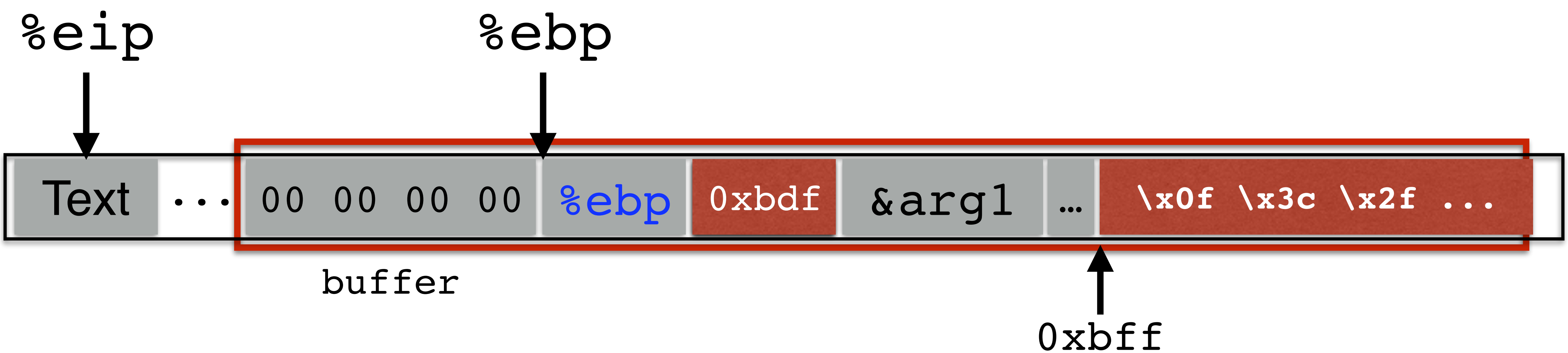
- One approach: just try a lot of different values!

- 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)

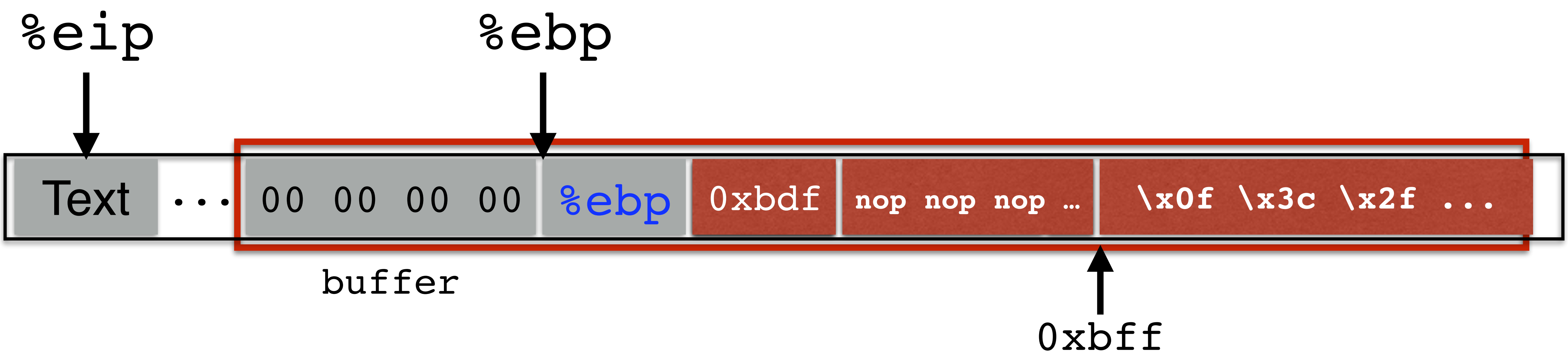
Improving our chances: `nop` sleds

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



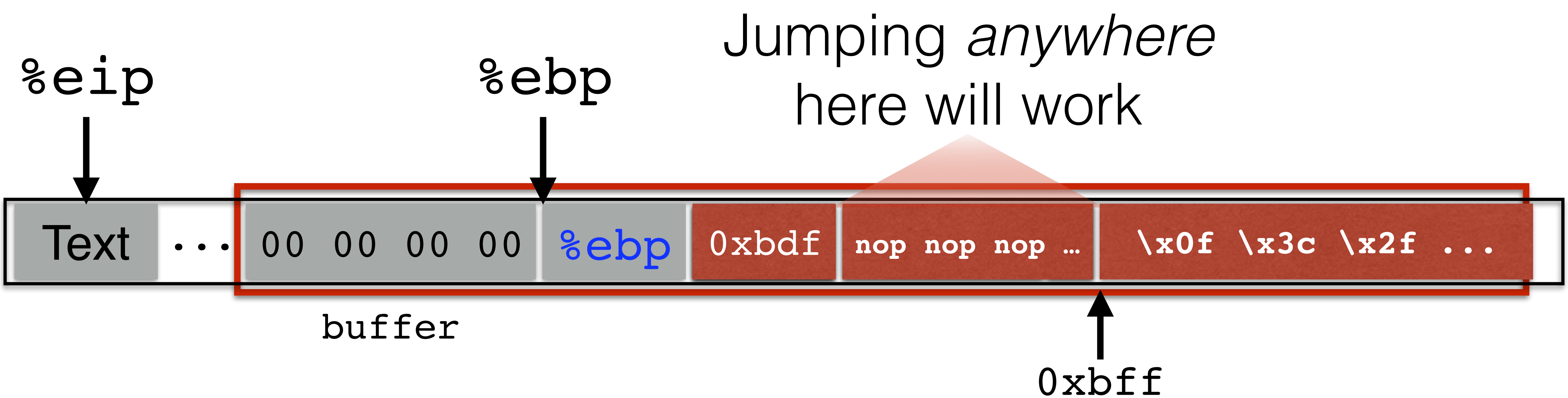
Improving our chances: `nop` sleds

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



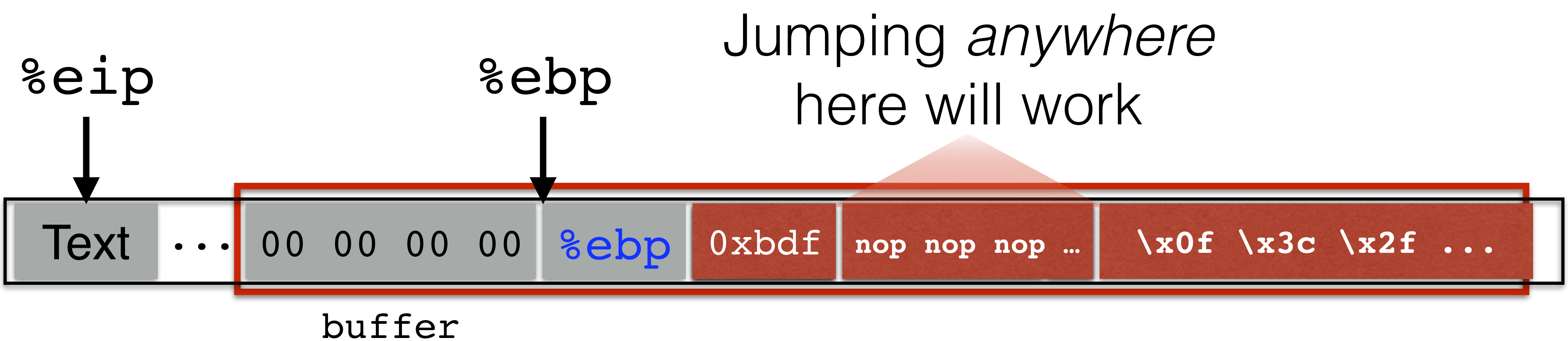
Improving our chances: nop sleds

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



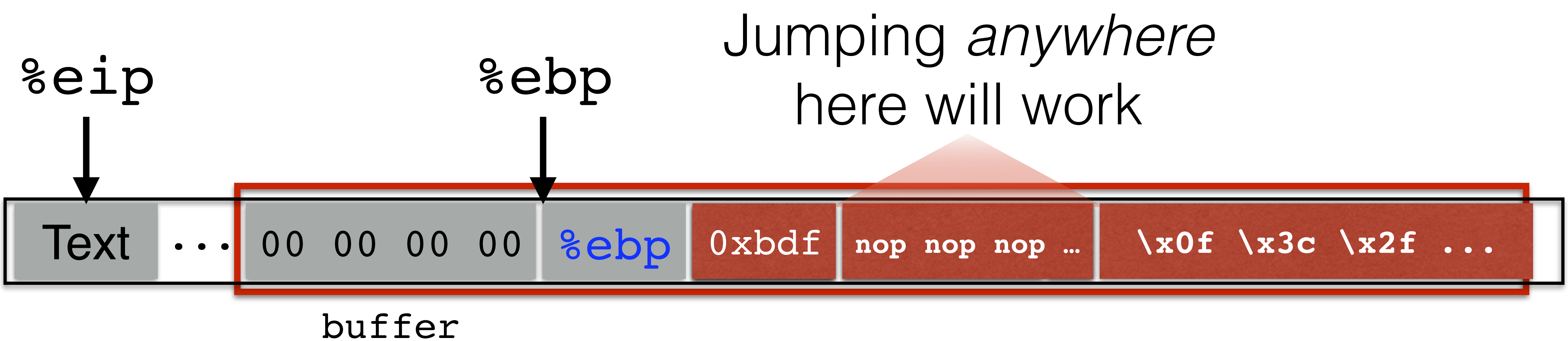
Improving our chances: `nop` sleds

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



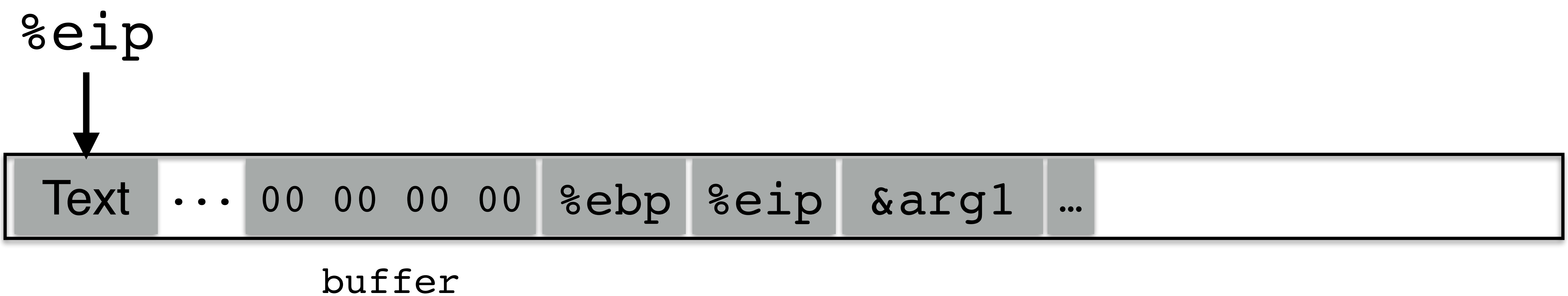
Improving our chances: `nop` sleds

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

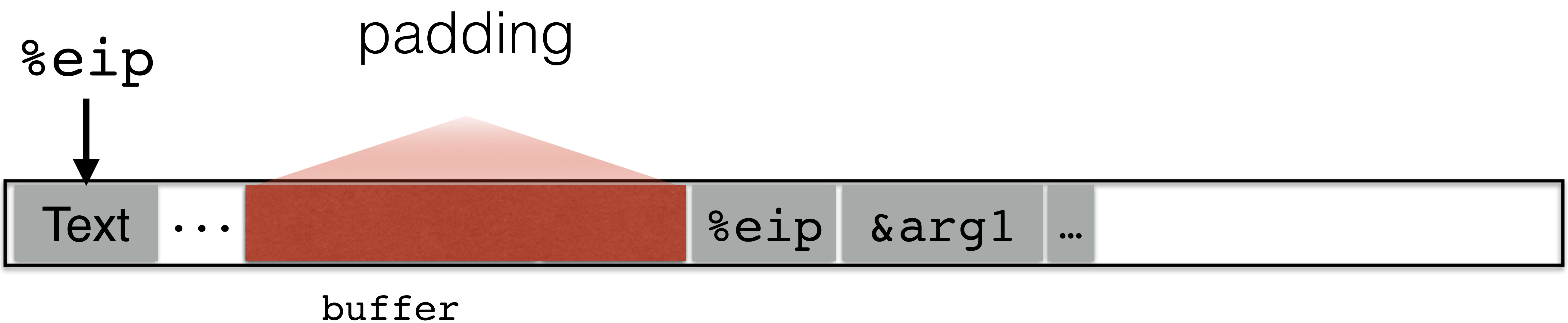


Now we improve our chances of guessing by a factor of #nops

Putting it all together



Putting it all together

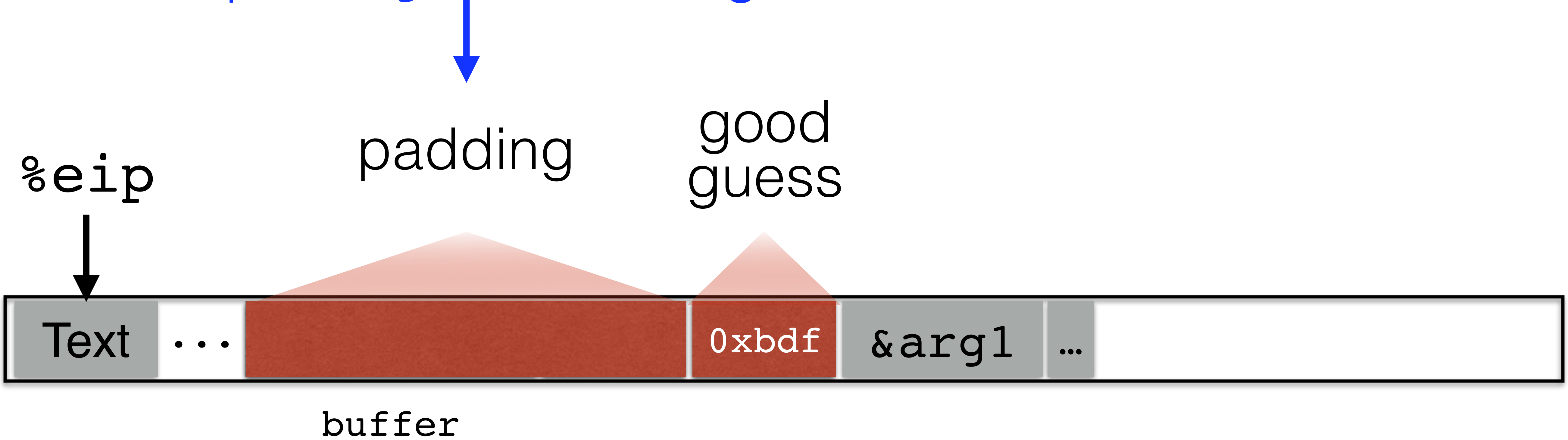


Putting it all together

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

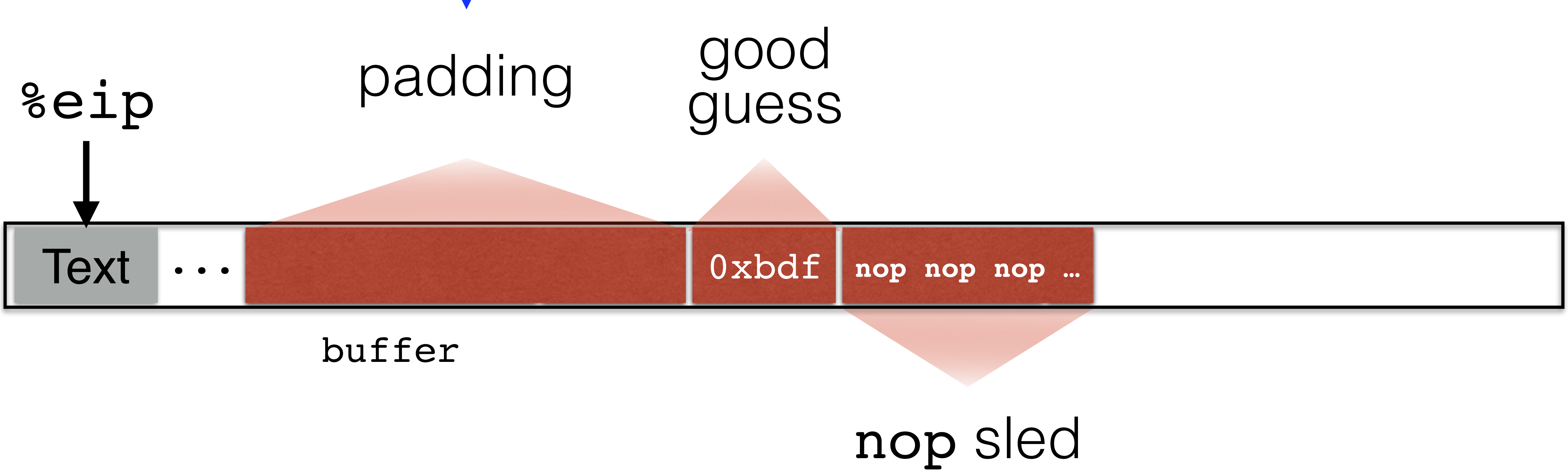
Putting it all together

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



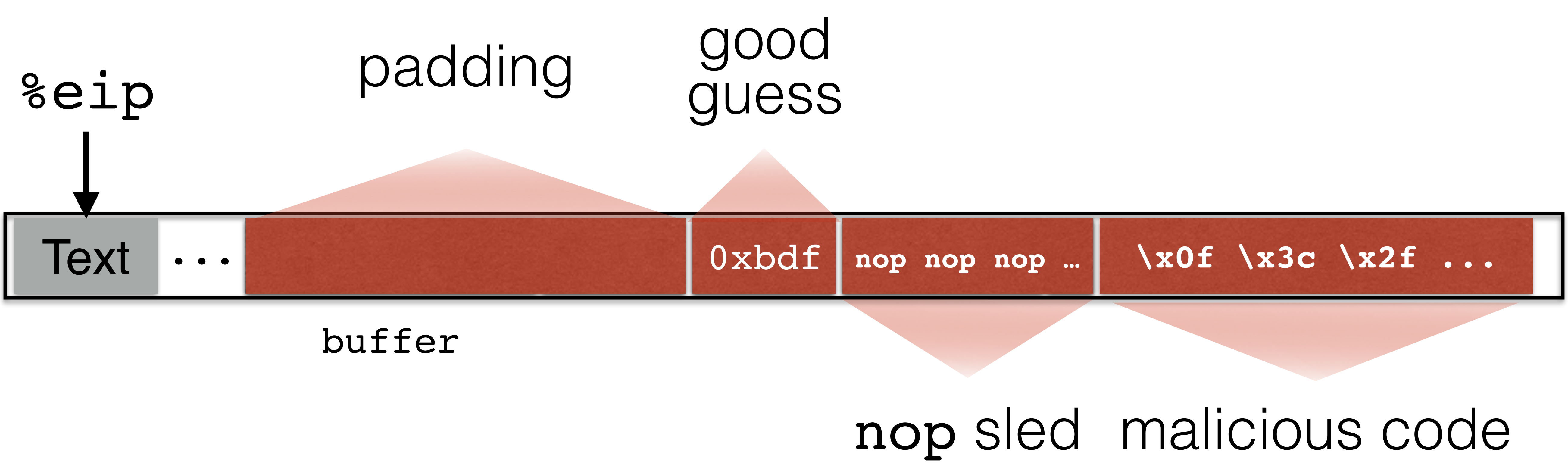
Putting it all together

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



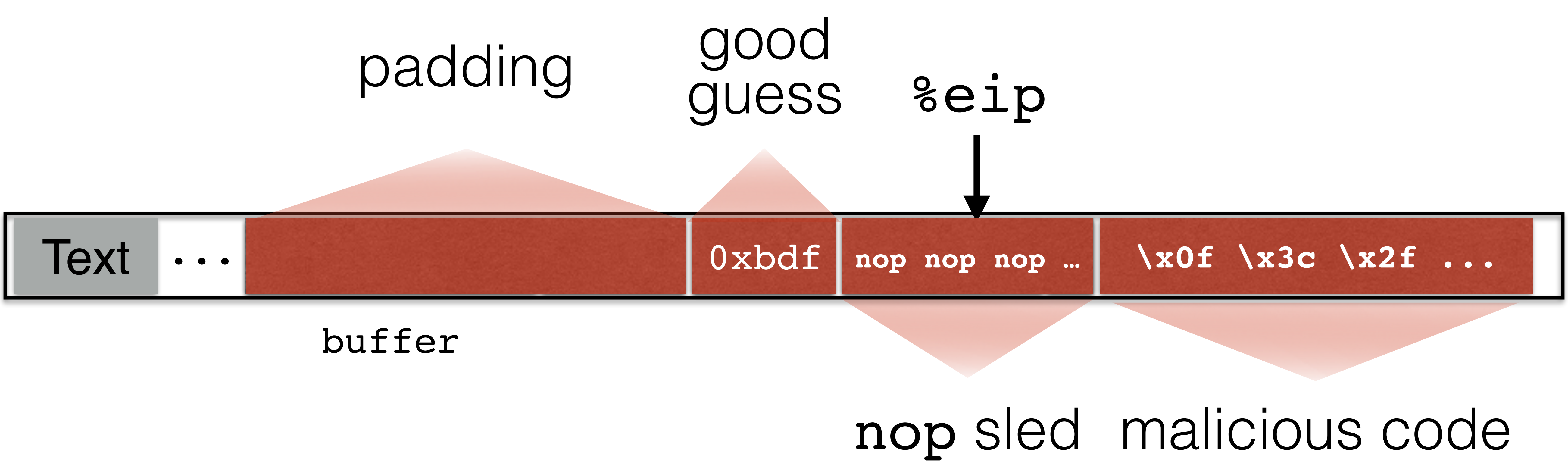
Putting it all together

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



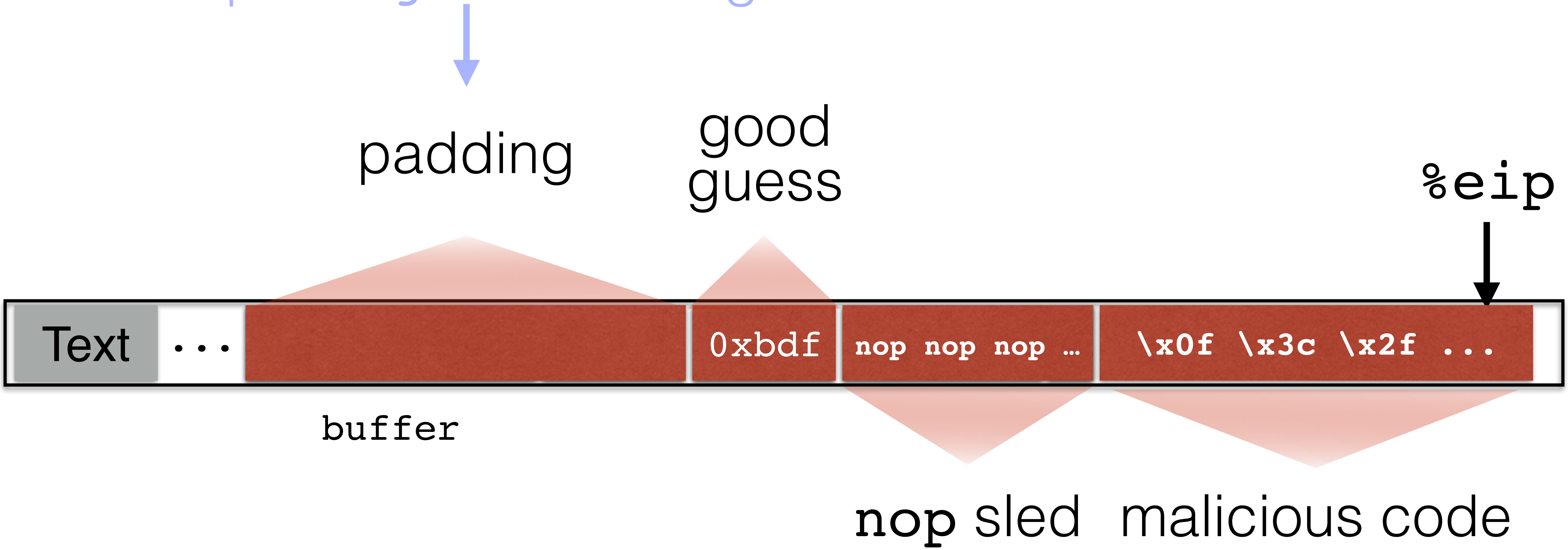
Putting it all together

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



Putting it all together

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



Project one
has been posted

Due 2 weeks from today (11:59pm Thursday Feb 18)

Defenses

Recall our challenges

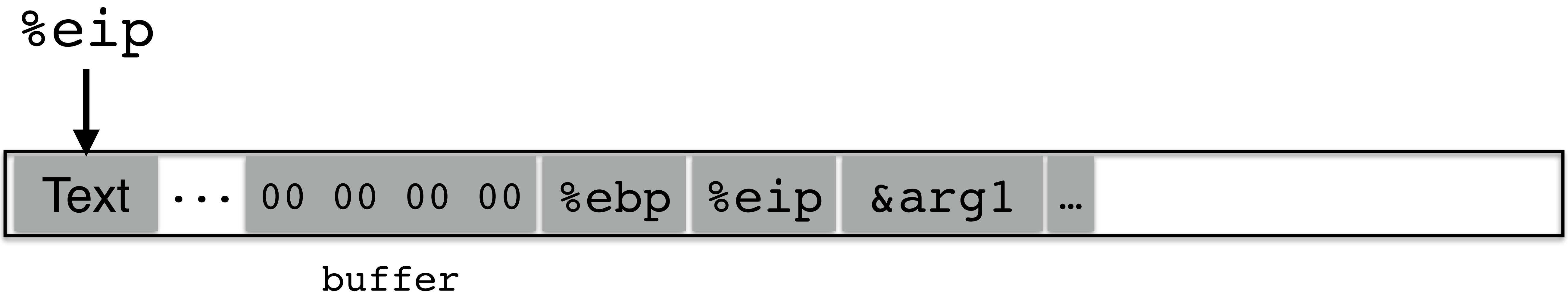
How can we make these even more difficult?

- Putting code into the memory (no zeroes)

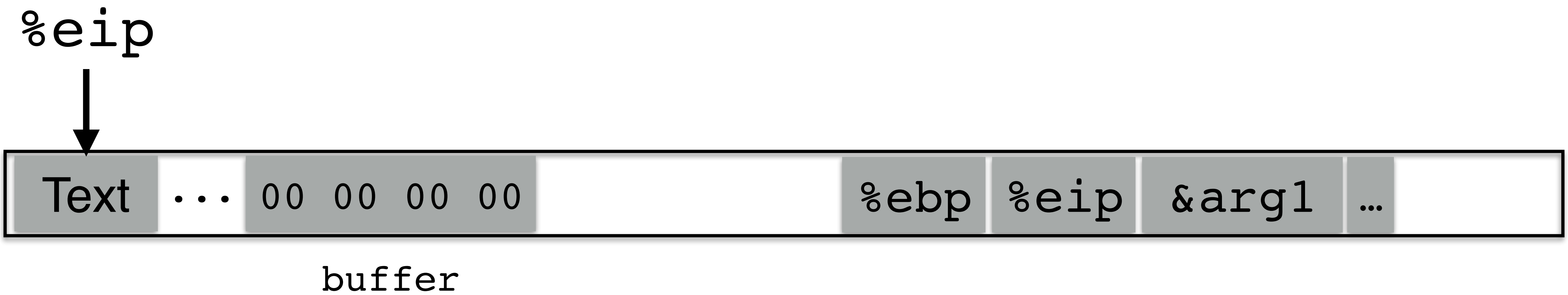
- Getting %eip to point to our code (dist buff to stored eip)

- Finding the return address (guess the raw addr)

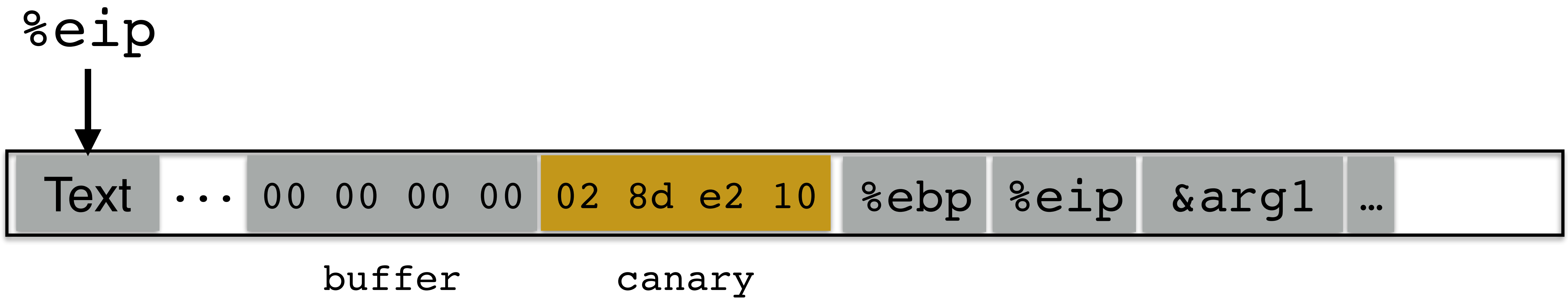
Detecting overflows with canaries



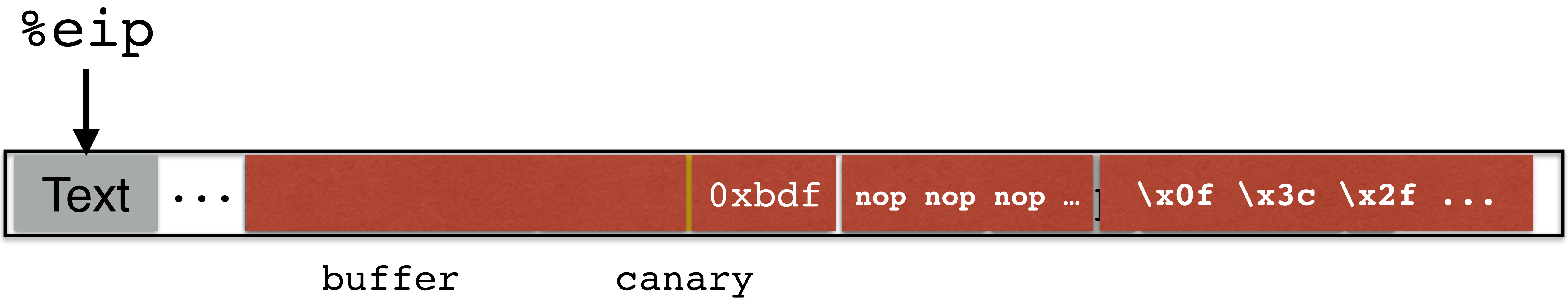
Detecting overflows with canaries



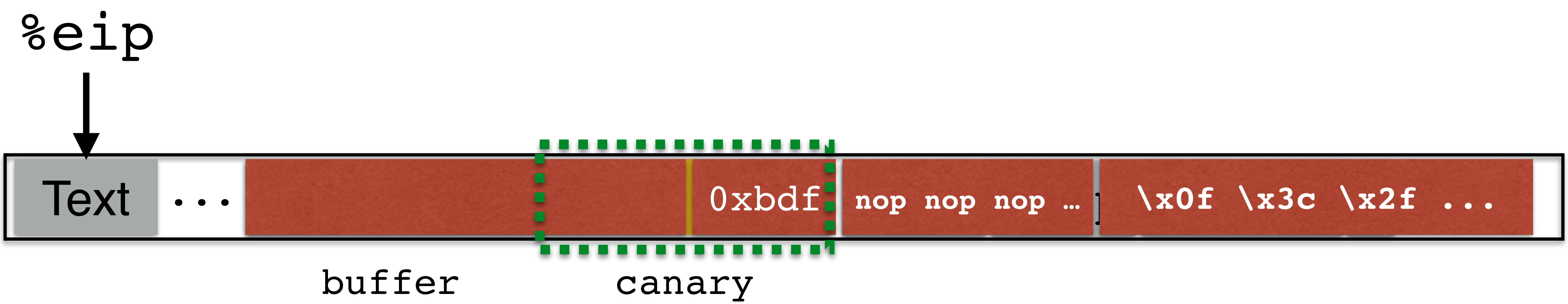
Detecting overflows with canaries



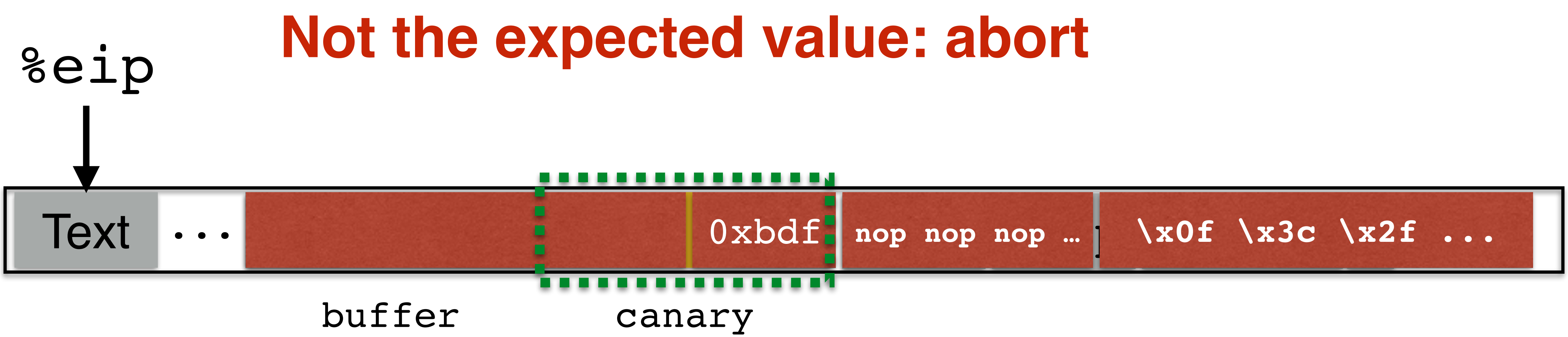
Detecting overflows with canaries



Detecting overflows with canaries

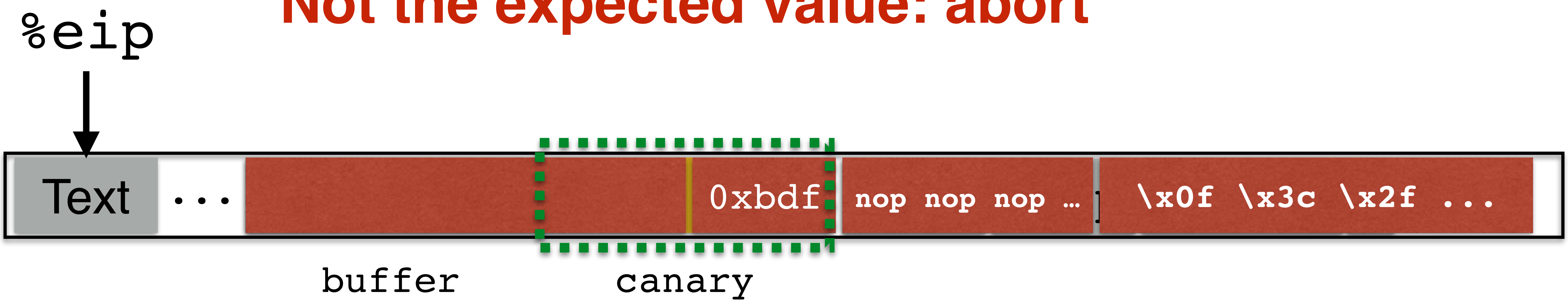


Detecting overflows with canaries



Detecting overflows with canaries

Not the expected value: abort



What value should the canary have?

Canary values

From StackGuard [Wagle & Cowan]

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

Recall our challenges

How can we make these even more difficult?

- Putting code into the memory (no zeroes)
  - Option: Make this detectable with canaries

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

- Finding the return address (guess the raw addr)

More next time…