Memory Safety and Buffer Overflows

(with material from Mike Hicks, Dave Levin and Michelle Mazurek)

Today’s agenda

- Why care about buffer overflows?
- Memory layout refresher
- Overflows and how they work

What is a buffer overflow?

- A **low-level** bug, typically in **C/C++**
  - Significant security implications!

- If accidentally triggered, causes a crash

- If maliciously triggered, can be **much worse**
  - **Steal** private info
  - **Corrupt** important info
  - **Run** arbitrary code

C and C++ still very popular

Language Rank	Types	Spectrum Ranking
1. Python		100.0
2. C++		98.3
3. C		98.3
4. Java		97.2
5. C#		92.7
6. R		82.8
7. PHP		82.7
8. JavaScript		82.6
9. Go		76.4
10. Assembly		74.2
11. Matlab		72.8
12. Scala		72.1
13. Ruby		71.3
14. HTML		70.5

https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages

Critical systems in C/C++

- Most **OS kernels** and utilities
  - fingerd, X windows server, shell
- Many **high-performance servers**
  - Microsoft IIS, Apache httpd, nginx
  - Microsoft SQL server, MySQL, redis, memcached
- Ma

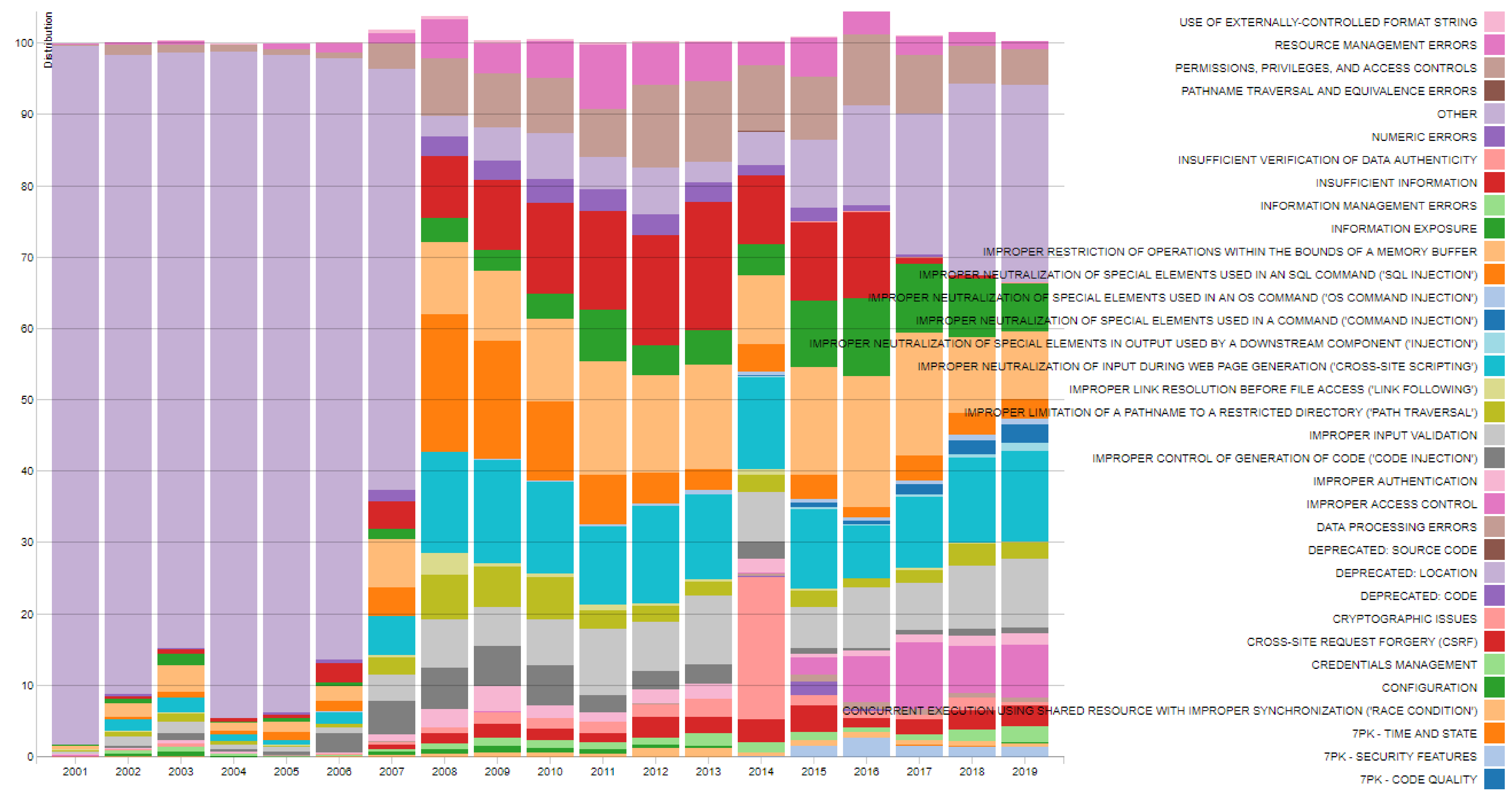
A successful attack on these systems is particularly dangerous!

  - Mars rover, industrial control systems, automobiles, healthcare devices, IoT

Trends

Relative Vulnerability Type Totals By Year

The vulnerabilties in the NVD are assigned a CWE based on a slice of the total CWE Dictionary. The visualization below shows a stacked bar graph of the total number of vulnerabilities assigned a CWE for each year. It is possible (although not common) that a vulnerability has multiple CWEs assigned.



https://nvd.nist.gov/vuln/visualizations/cwe-over-time

History of Buffer Overflows

- Morris Worm (1988)
  - First internet worm
  - Spread across Unix Machines
- Code Red (2001)
  - Vulnerability in Microsoft Internet Information Services (for hosting web applications)
  - DDoS attack on White House’s servers
- SQL Slammer (2003)
  - Vulnerability in Microsoft SQL Server 2000.
  - Worm spread across more than 250,000 computers and caused a massive internet outage

Recent Examples

Netgear Zero-Day Allows Full Takeover of Dozens of Router Models

June 19, 2020 / 9:05 am

An unpatched vulnerability in the web server of device firmware gives attackers root privileges, researchers said.

UPDATED

Researchers this week said they discovered an unpatched, zero-day vulnerability in firmware for Netgear routers that put 79 device models at risk for full takeover, they said.

Microsoft Releases Emergency Windows Patch for Malicious Image Attack

By Ryan Whitwam on July 1, 2020 at 3:33 pm | 0 Comments

Microsoft rolls out patches to Windows 10 on a more or less regular schedule these days, but it wastes no time when there's a flaw that could put users at risk. The company is

Windows 10 and Windows
files that let them execute
ng this flaw is a bit unusual,

ıside the Windows Codecs
This component contains the necessary software to decode and render many
t image and video formats in Windows. By causing a buffer overflow with
ıed image files, the attacker can "trick" the computer into leaking important data
ning code hidden in the image files.

VLC Media Player Fixes Critical Buffer Overflow RCE Bug

By Bill Toulas - June 17, 2020

Users of the VLC (Video Lan Client) media player, one of the most widely used, powerful, and versatile media players out there, are advised to apply the latest available update, which fixes a nasty bug. Carrying the identifier CVE-2020-13428, this vulnerability allows a malicious remote actor to either crash the media player or carry out arbitrary code execution with the privileged of the victimized user. When combined with additional

What we’ll do

- Understand how these attacks work, and how to defend against them
- These require knowledge about:
  - The compiler
  - The OS
  - The architecture

Analyzing security requires a whole-systems view

Note about terminology

- We will use **buffer overflow** to mean ***any access of a buffer outside of its allotted bounds***
  - An over-*read*, or an over-*write*
  - During *iteration* (“running off the end”) or by *direct access*
  - Could be to addresses that *precede* or *follow* the buffer

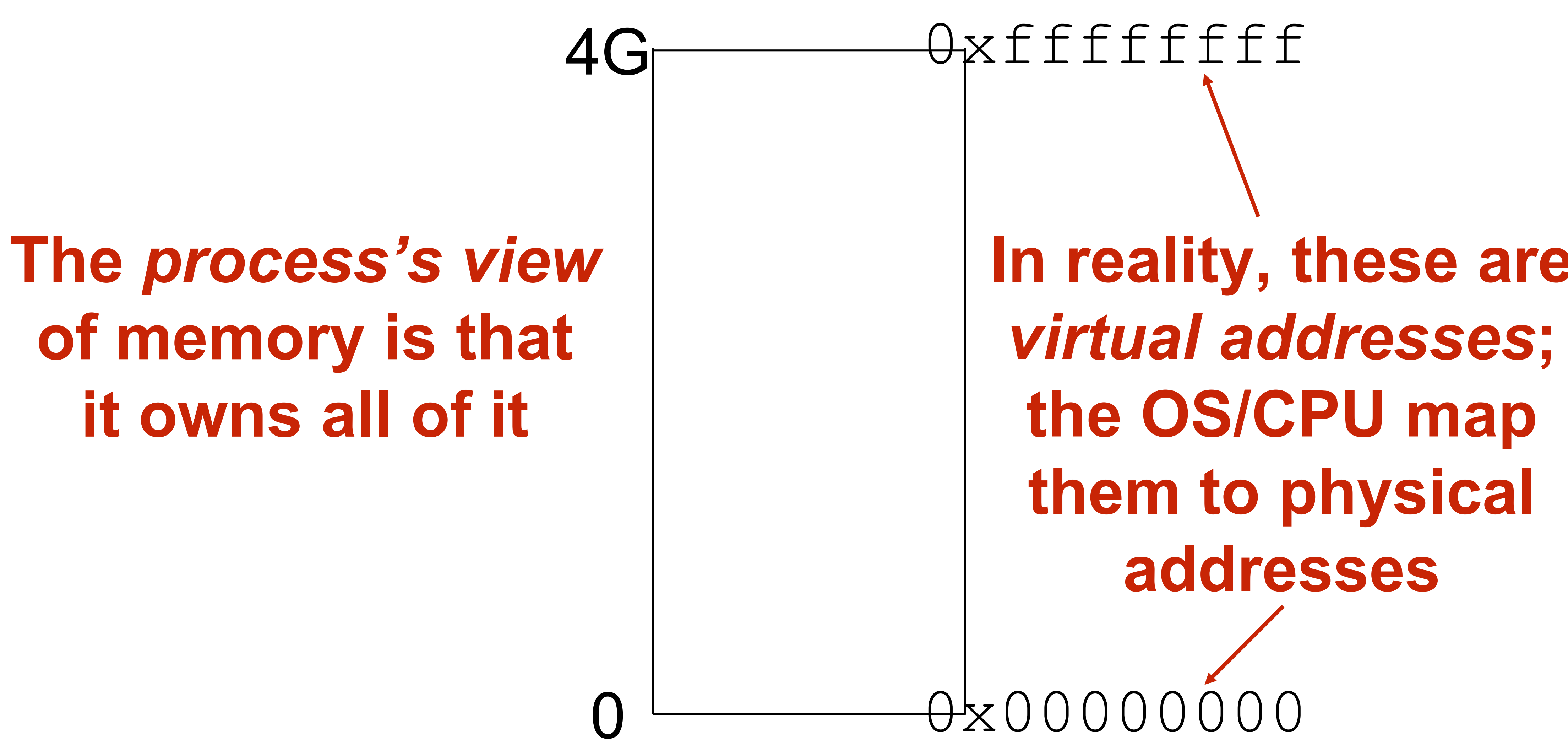
http://www.williesimpson.com/wp-content/uploads/2011/03/memory-lane.jpg

Memory layout

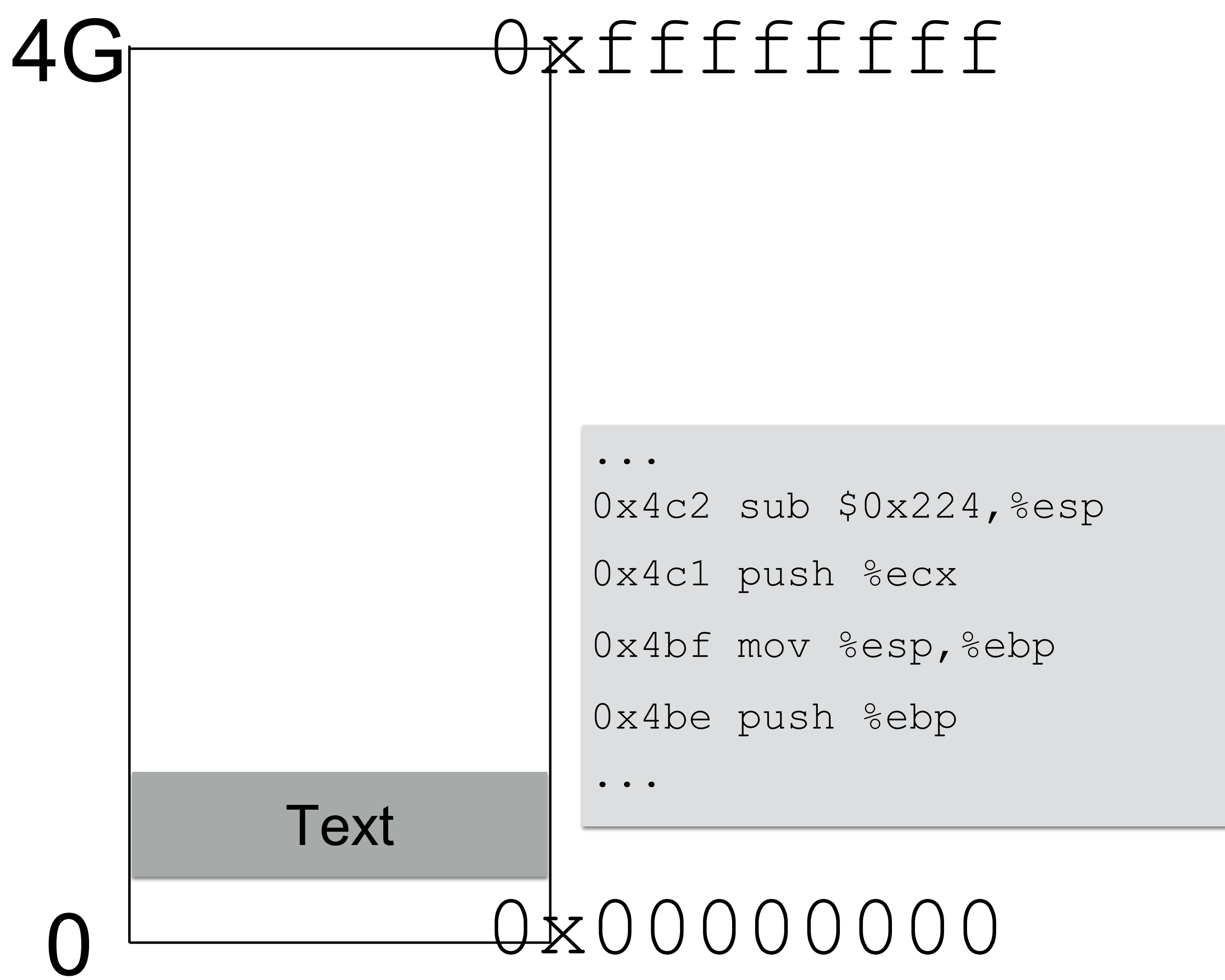
Memory Layout Refresher

- 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

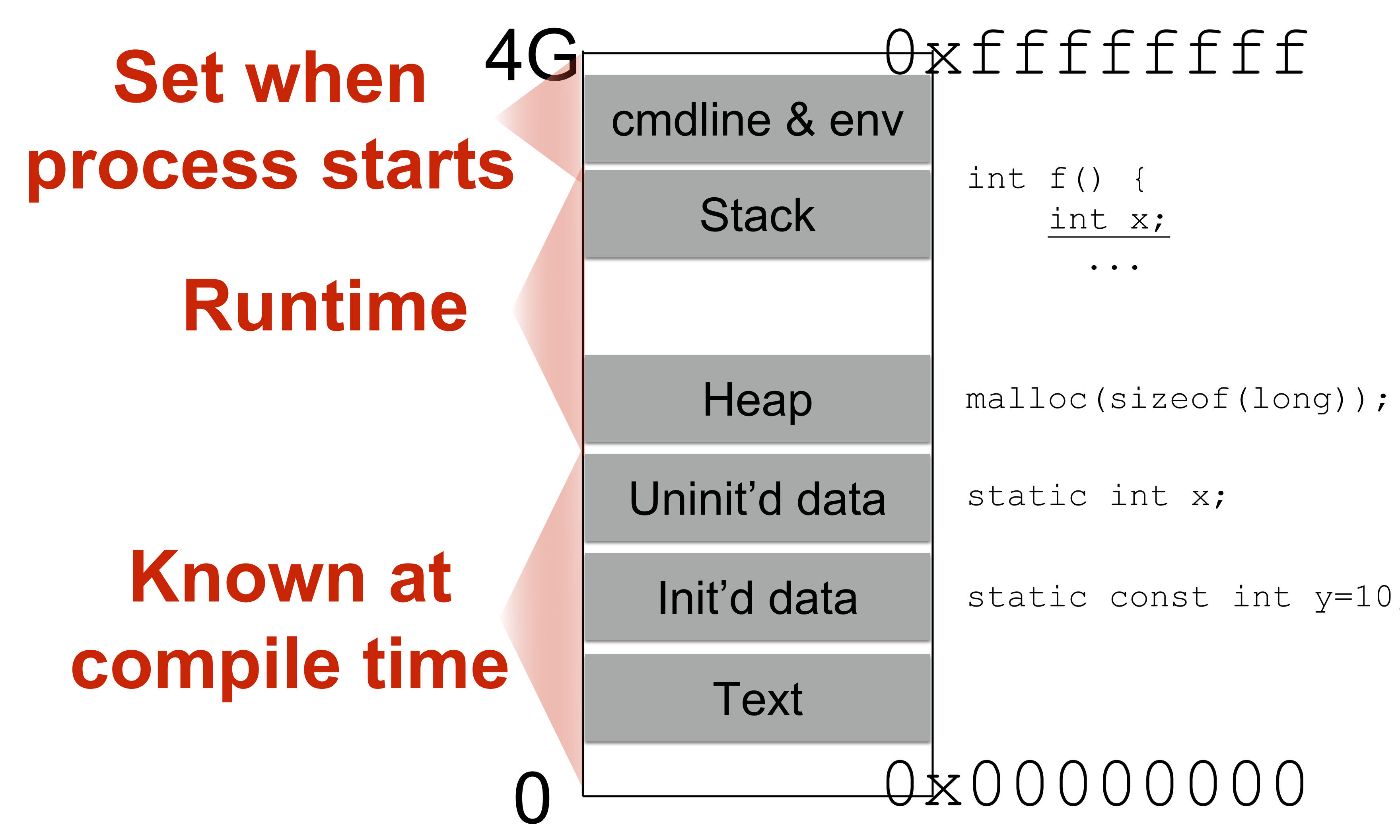
All programs stored in memory



Program **instructions** are in memory



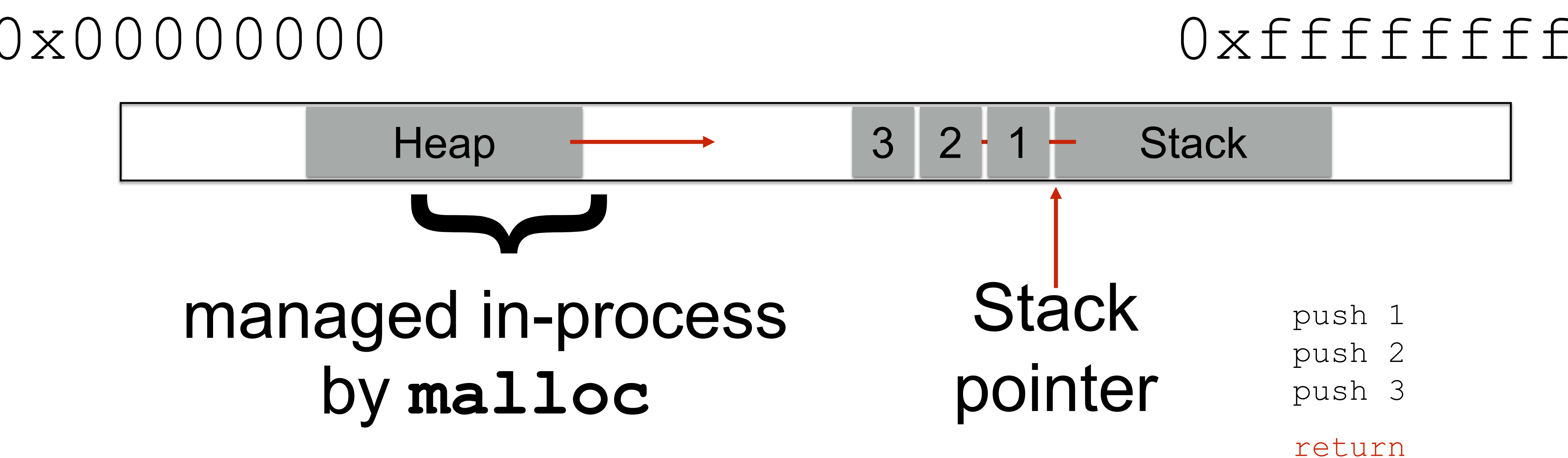
Location of data areas



Memory allocation

Stack and heap grow in opposite directions

Compiler emits instructions to
adjust the size of the stack at run-time



Focusing on the stack for now

Stack and function calls

- What happens when we **call** a function?
  - What data needs to be stored?
  - Where does it go?
- What happens when we **return** from a function?
  - What data needs to be *restored*?
  - Where does it come from?

Basic stack layout

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

0xffffffff

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

Happens during callee

Arguments pushed in reverse order of code

Happens during caller

The local variable allocation is ultimately up to the compiler: Variables could be allocated in any order, or not allocated at all and stored only in registers, depending on the optimization level used.

Accessing variables

```
void func(char *arg1, int arg2, int arg3)
{
    ...
    loc2++;
    ...
}
```

Q: Where is (this) `loc2`?
A: -8(%ebp)

0xffffffff

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

0xbffff323 %ebp

Frame pointer

Stack frame for `func`

Can't know absolute address at compile time

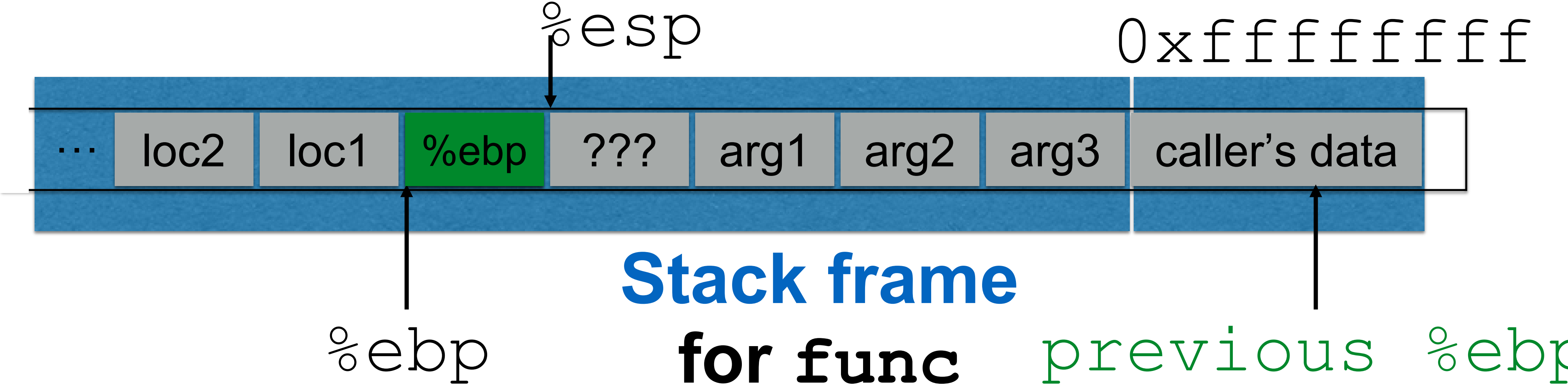
But can know the **relative** address

- `loc2` is always 8B before ???s

Returning from functions

Q: How do we restore previous %ebp?

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



Push current `%ebp` before `locals`

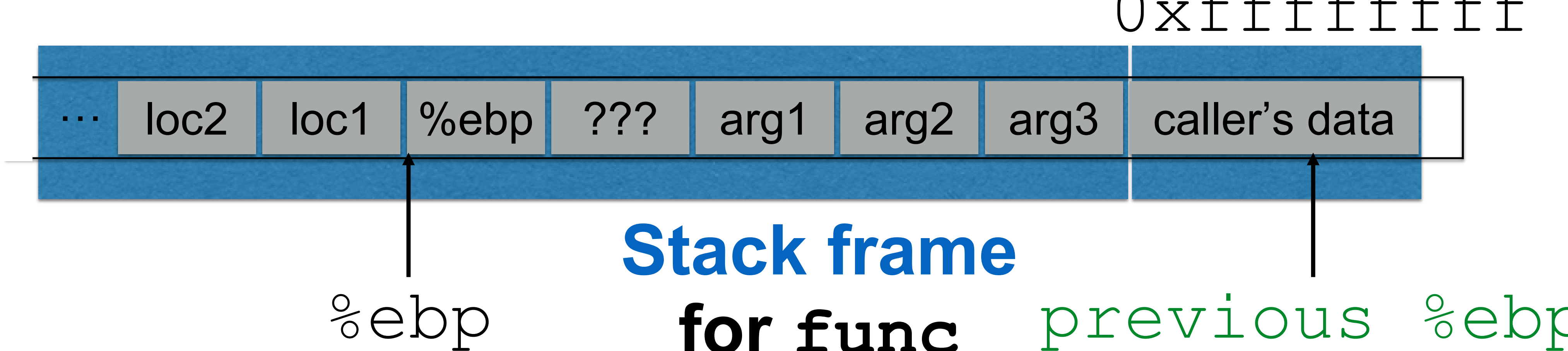
Set `%ebp` to current `%esp`

Set `%ebp` `to(%ebp)` at return

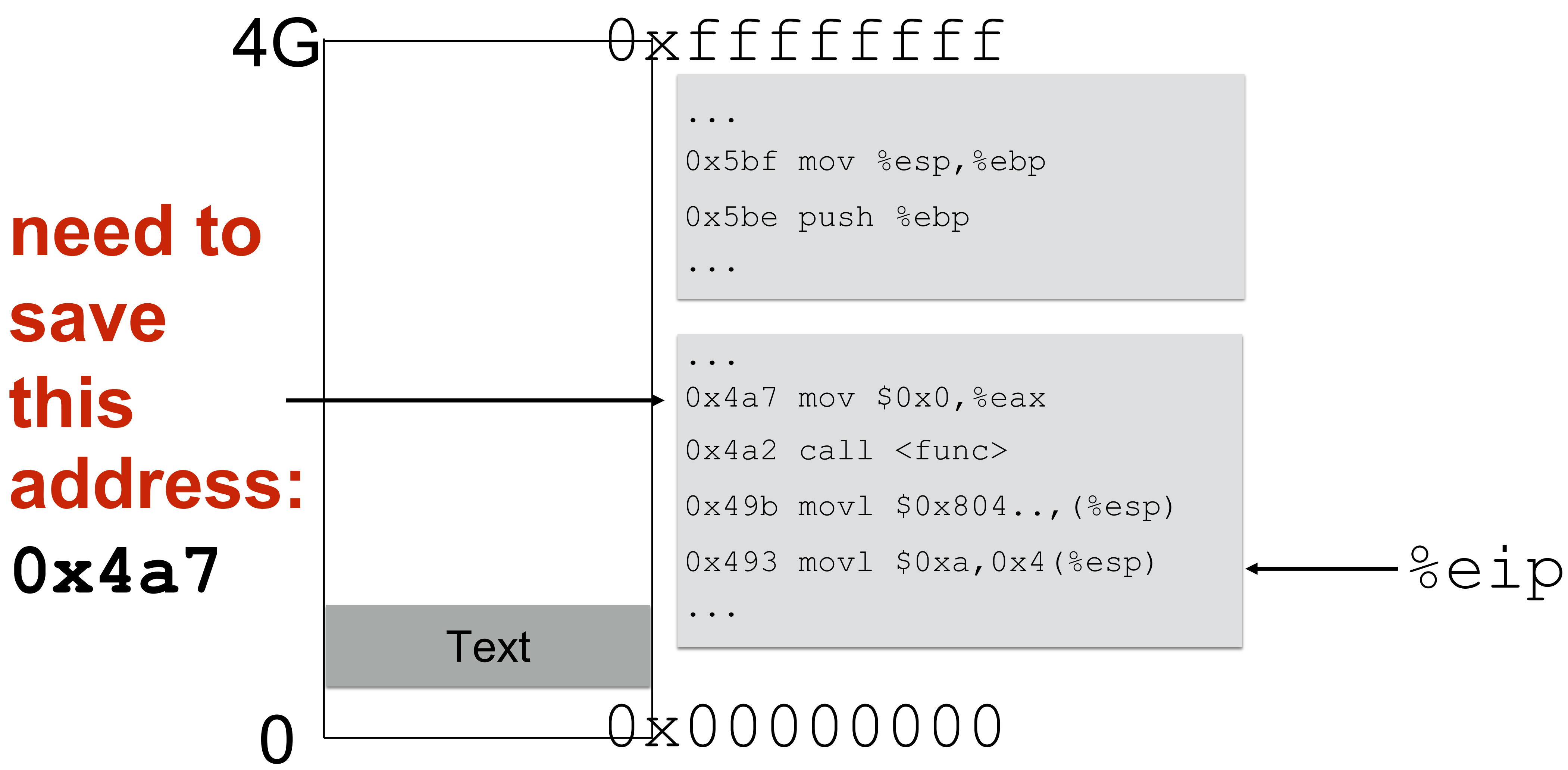
Returning from functions

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

Q: How do we resume here?



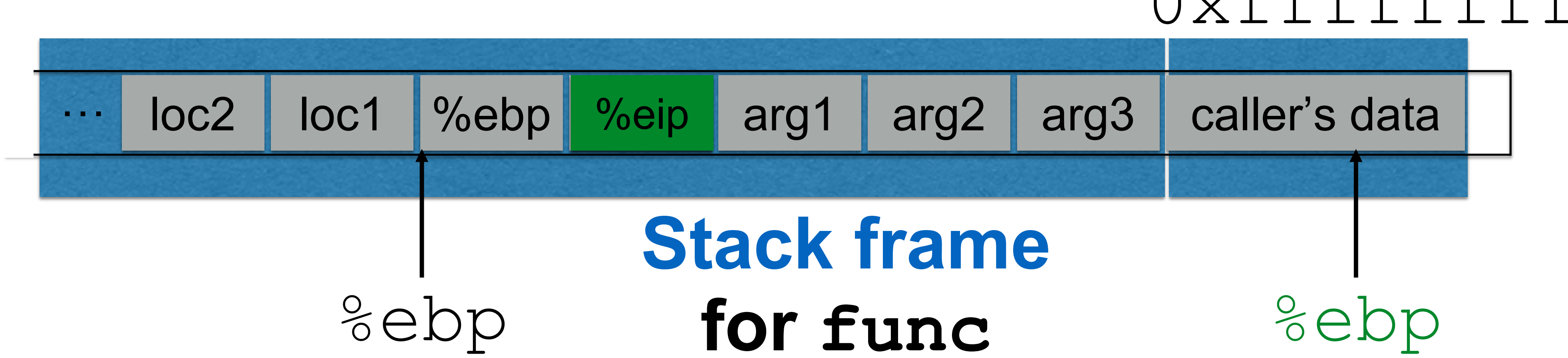
Instructions in memory



Returning from functions

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

Q: How do we resume here?



Set `%eip to 4(%ebp)` at return

Push next `%eip` before `call`

Stack and functions: Summary

Calling function:

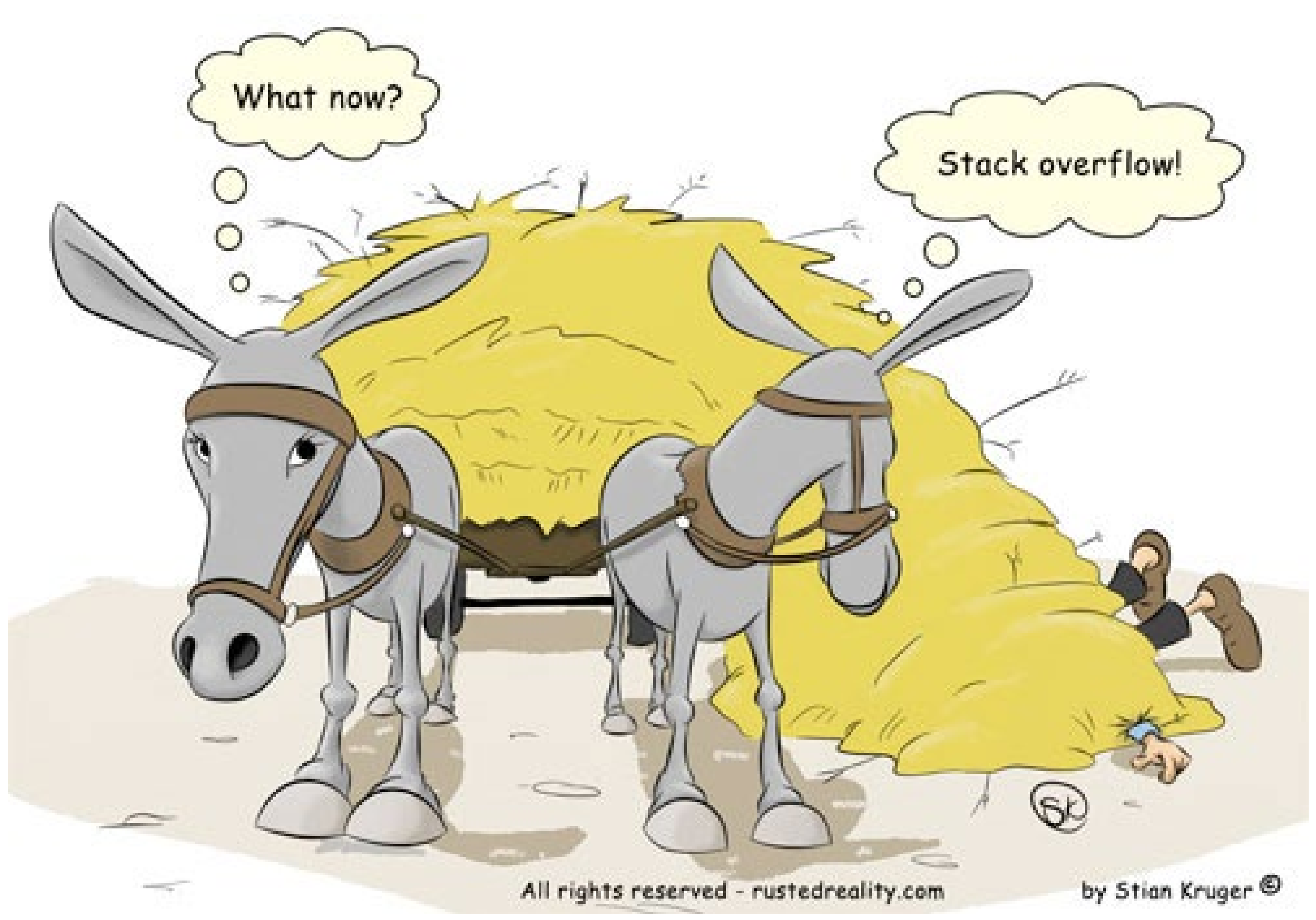
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you
3. **Jump** to the function's address

Called function:

4. **Push the old frame pointer** onto the stack: %ebp
5. **Set frame pointer** to where the end of the stack is right now: %ebp = %esp
6. **Push local variables** onto the stack

Returning from function:

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



http://rustedreality.com/stack-overflow/

Buffer overflows

Buffer overflows from 10,000 ft

- **Buffer =**
  - Contiguous memory associated with a variable or field
  - Common in C
    - All strings are (NUL-terminated) arrays of `char`'s
- **Overflow =**
  - Put more into the buffer than it can hold
- Where does the overflowing data **go**?
  - Well, now that you are experts in memory layouts…

Benign outcome

```
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)

Security-relevant outcome

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

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

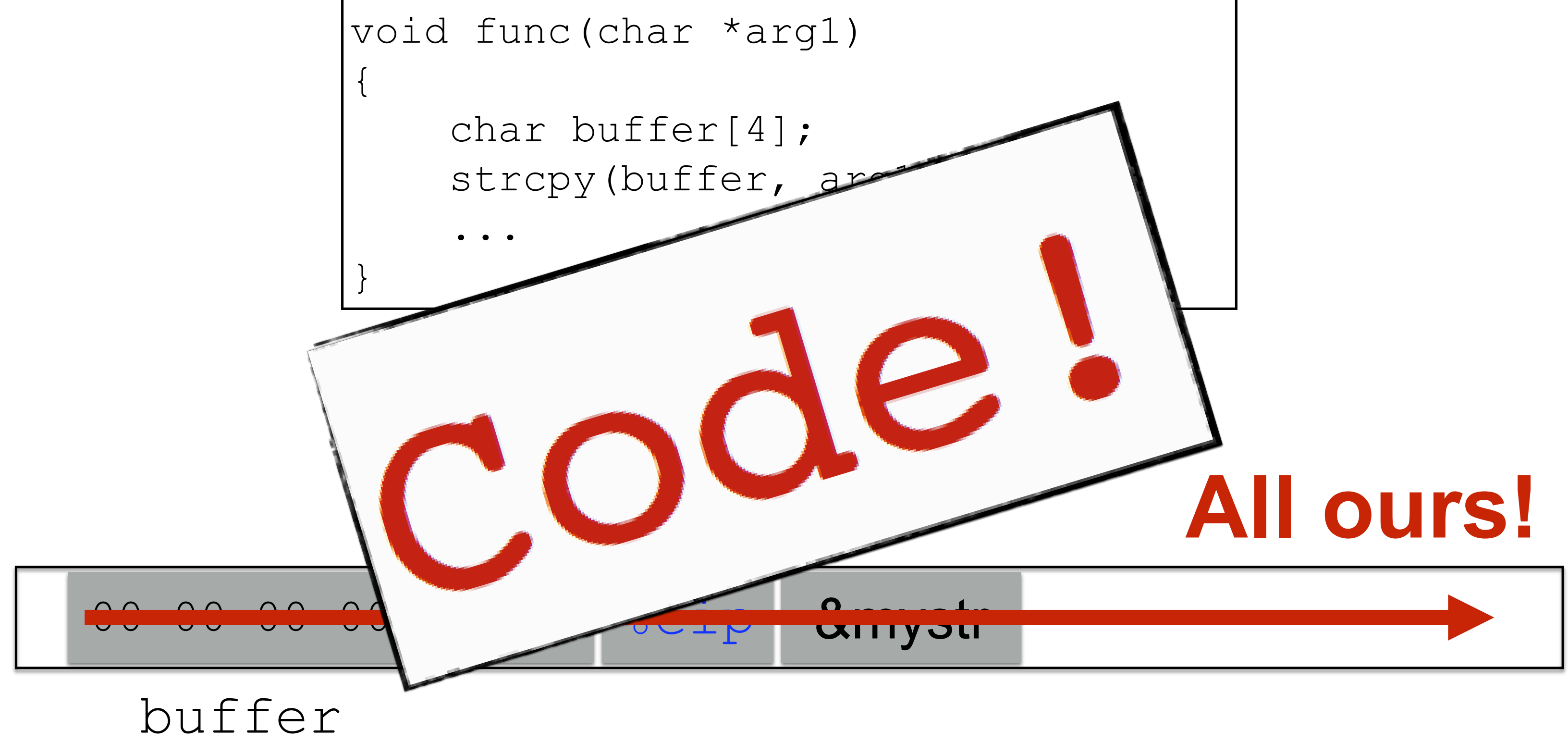
Code still runs; user now ‘authenticated’

M e ! \0

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

buffer authenticated

Could it be worse?



`strcpy` will let you write as much as you want (til a '\0')
What could you write to memory to wreak havoc?

Aside: User-supplied strings

- These examples provide their own strings
- In reality strings come **from *users*** in myriad ways
    - Text input, packets, environment variables, file input…
- **Validating assumptions** about user input is critical!
    - We will discuss it later, and throughout the course

Code Injection

http://images.clipartpanda.com/injection-clipart-color-medical-injection-21941863.jpg

Code Injection: Main idea

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

%eip

Text	...	00 00 00 00	%ebp	%eip	&arg1	...	Haxx0r c0d3

buffer

(1) Load my own code into memory

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

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 to write assembly to never contain a full zero byte?
  - It **can’t use the loader** (we’re injecting)
    - How to find addresses we need?

What code to run?

- One goal: **general-purpose shell**
  - Command-line prompt that gives attacker **general access to the system**
- The code to launch a shell is called **shellcode**
- Other stuff you could do?

Shellcode

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

filename argv envp

xor to avoid zero byte

Assembly

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

Machine code

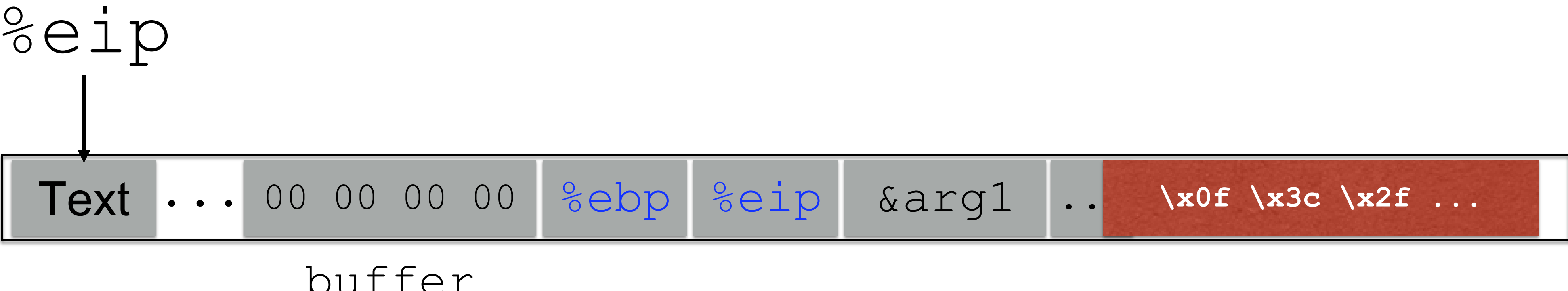
```
"\x31\xc0"
"\x50"
"\x68" "//sh"
"\x68" "/bin"
"\x89\xe3"
"\x50"
...
```

(Part of) your input

Challenge 2

Getting injected code to run

- We have code somewhere in memory
  - We don't know precisely where
- We need to move %eip to point at it



Recall

Stack and functions: Summary

Calling function:

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

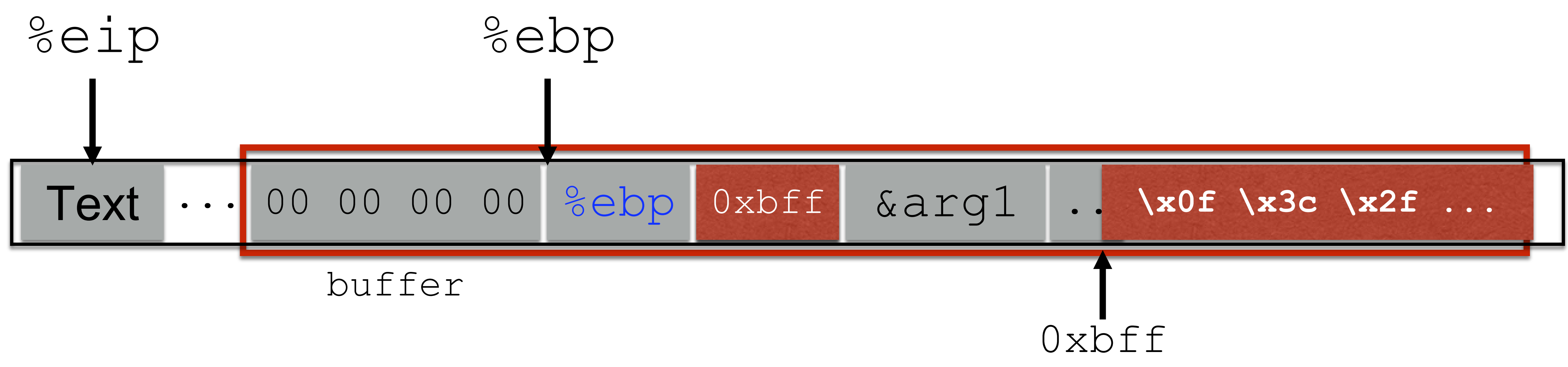
Called function:

4. **Push the old frame pointer** onto the stack: %ebp
5. **Set frame pointer** to where the end of the stack is right now: %ebp = %esp
6. **Push local variables** onto the stack

Returning from function:

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

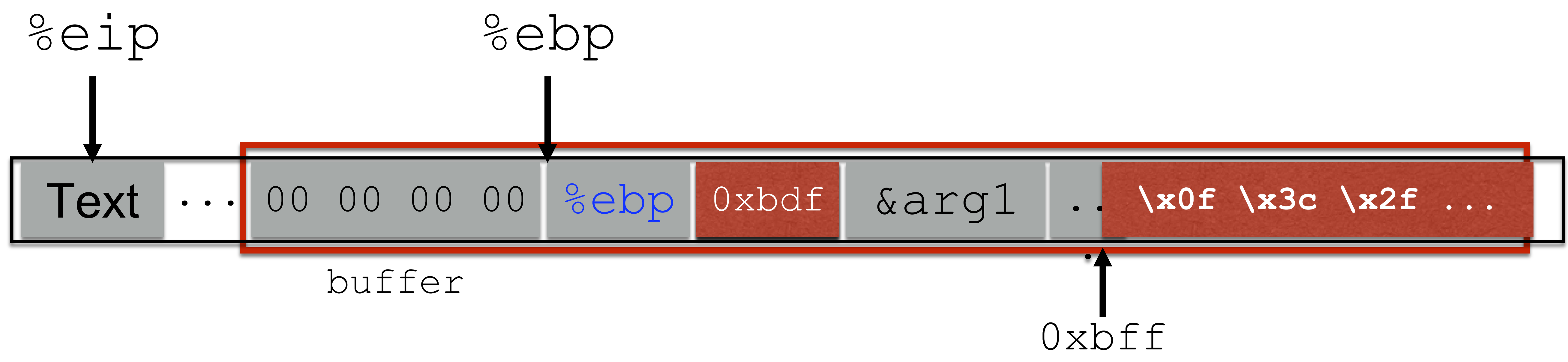
Hijacking the saved `%eip`



But how do we know the address?

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

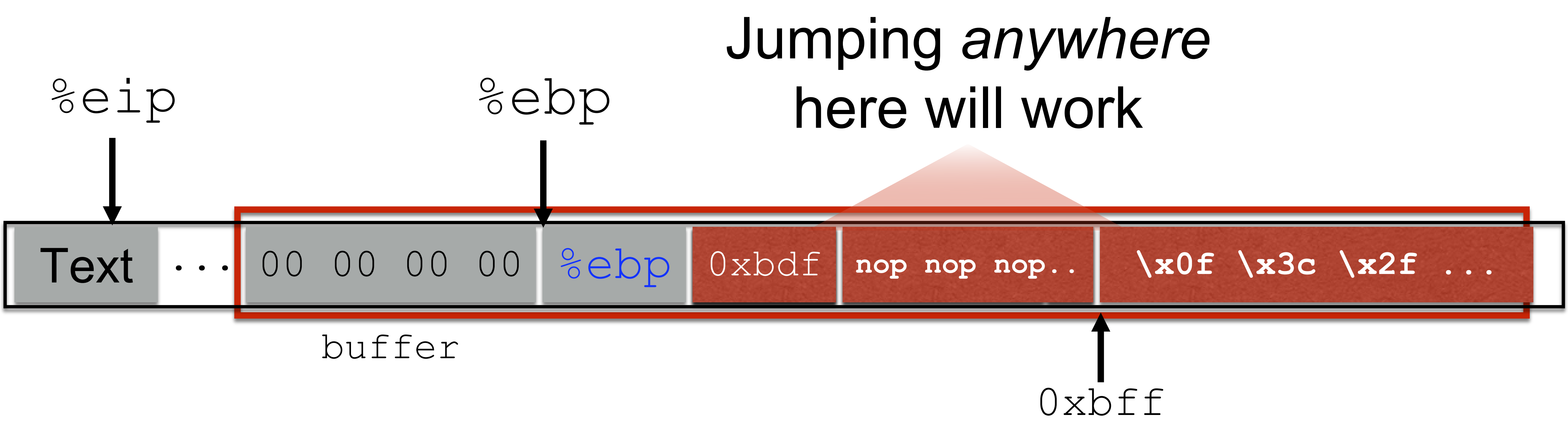
- If we don't have access to the code, we don't know how far the buffer is from the saved `%ebp`

- One approach: 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

- Without address randomization (discussed later):
  - Stack **always** starts from the same **fixed address**
  - 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 no-op instruction
(just moves to the next instruction)



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

Putting it all together

Fill in the space between the target buffer and the `%eip` to overwrite