Buffer Overflows

Blase Ur, David Cash, Ben Zhao
UChicago CMSC 23200/33250
(Slides partially borrowed from Michelle Mazurek, Mike Hicks, and Dave Levin at UMD)

THE UNIVERSITY OF CHICAGO

UCHICAGO SUPER GROUP

What is a buffer overflow?

"The software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer." (NIST/CWE)

What is a buffer overflow?

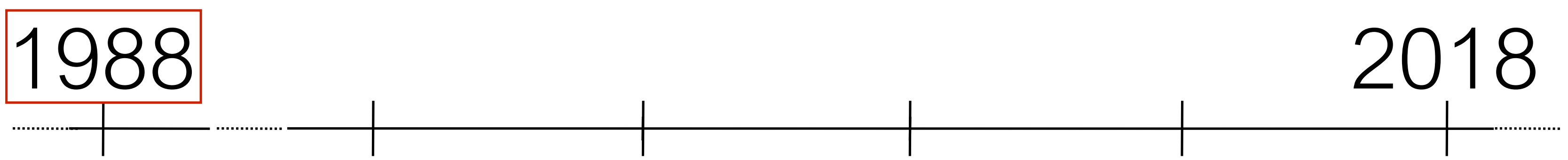
- A **low-level** bug, typically in **C/C++**
- Causes a crash if accidentally triggered
- If maliciously triggered, can be **much worse**
  - **Steal** private info
  - **Corrupt** important info
  - **Run** arbitrary code

Critical systems in C/C++

- Most **OS kernels** and utilities
  - X windows server, shell
- Many **high-performance servers**
  - Microsoft IIS, Apache httpd, nginx
  - Microsoft SQL server, MySQL, redis, memcached
- Many **embedded systems**
  - Mars rover, industrial control systems, automobiles, healthcare devices

History of buffer overflows

The harm has been substantial



- **Morris worm**
  - Propagated across machines (too aggressively, thanks to a bug)
  - One way it propagated was a **buffer overflow** attack against a vulnerable version of `fingerd` on VAXes
    - Sent a special string to the finger daemon, which caused it to execute code that created a new worm copy
    - Didn't check OS: caused Suns running BSD to crash
  - End result: $10-100M in damages, probation, community service

Slashdot

Channels

stories

submissions

popular

blog

ask slashdot

book reviews

games

idle

yro

technology

23-Year-Old X11 Server Security Vulnerability Discovered

Posted by **Unknown Lamer** on Wednesday January 08, 2014 @10:11
from the stack-smashing-for-fun-and-profit dept.

An anonymous reader writes

> "The recent report of X11/X.Org security in bad shape rings more truth today. The X.Org Foundation announced today that they've found a X11 security issue that dates back to 1991. The issue is a possible stack buffer overflow that could lead to privilege escalation to root and affects all versions of the X Server back to X11R5. After the vulnerability being in the code-base for 23 years, it was finally uncovered via the automated cppcheck static analysis utility."

There's a `scanf` used when loading BDF fonts that can overflow using a carefully crafted font. Watch out for those obsolete early-90s bitmap fonts.

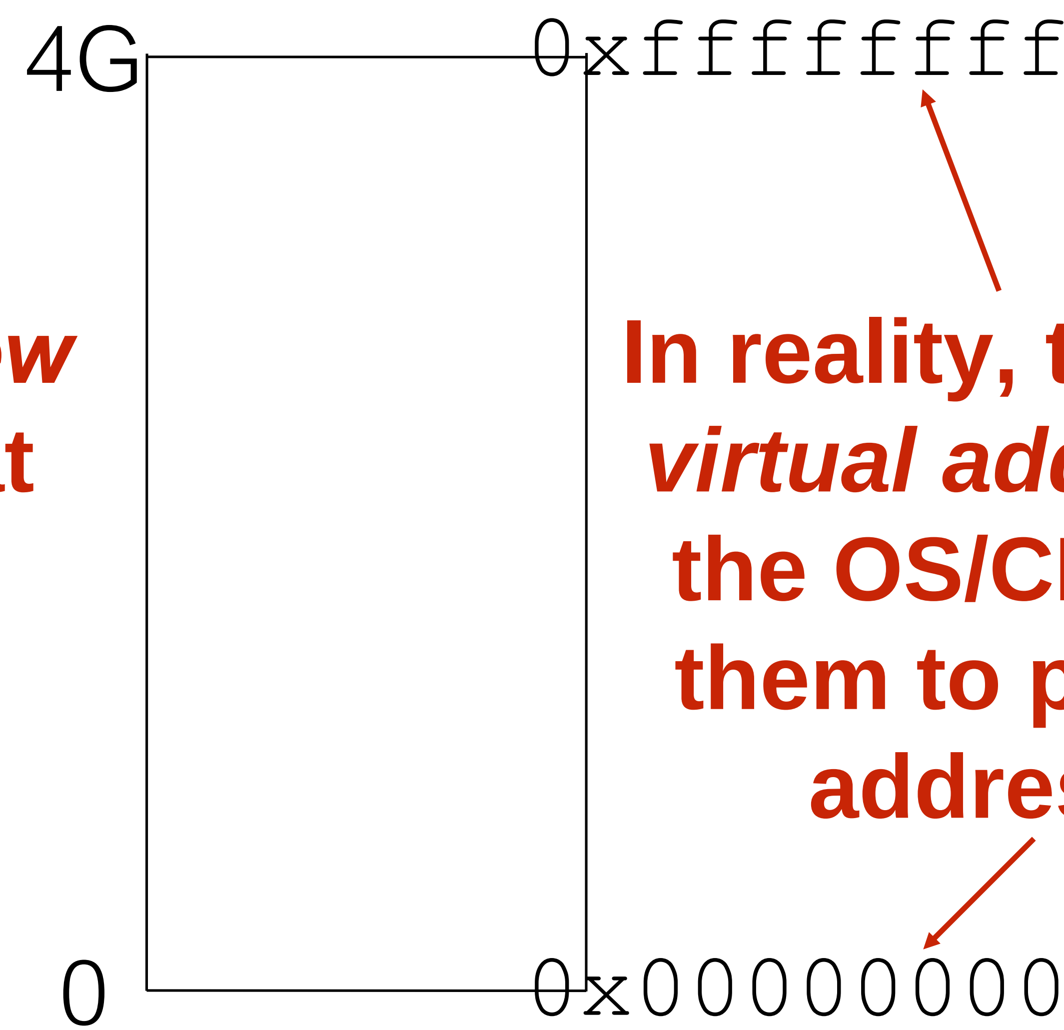
Note about terminology

- 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

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/C process model
    - Similar to other operating systems

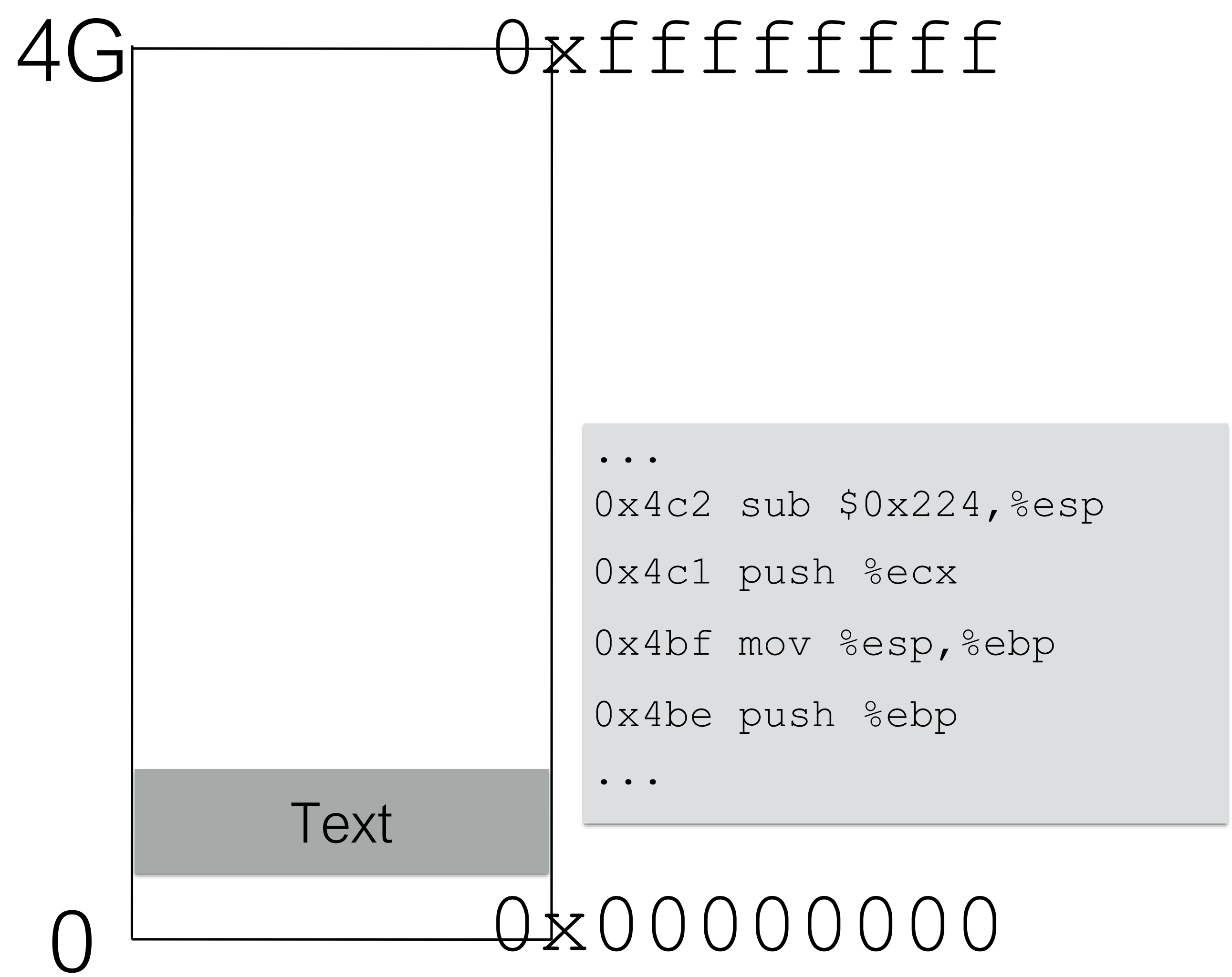
All programs stored in memory



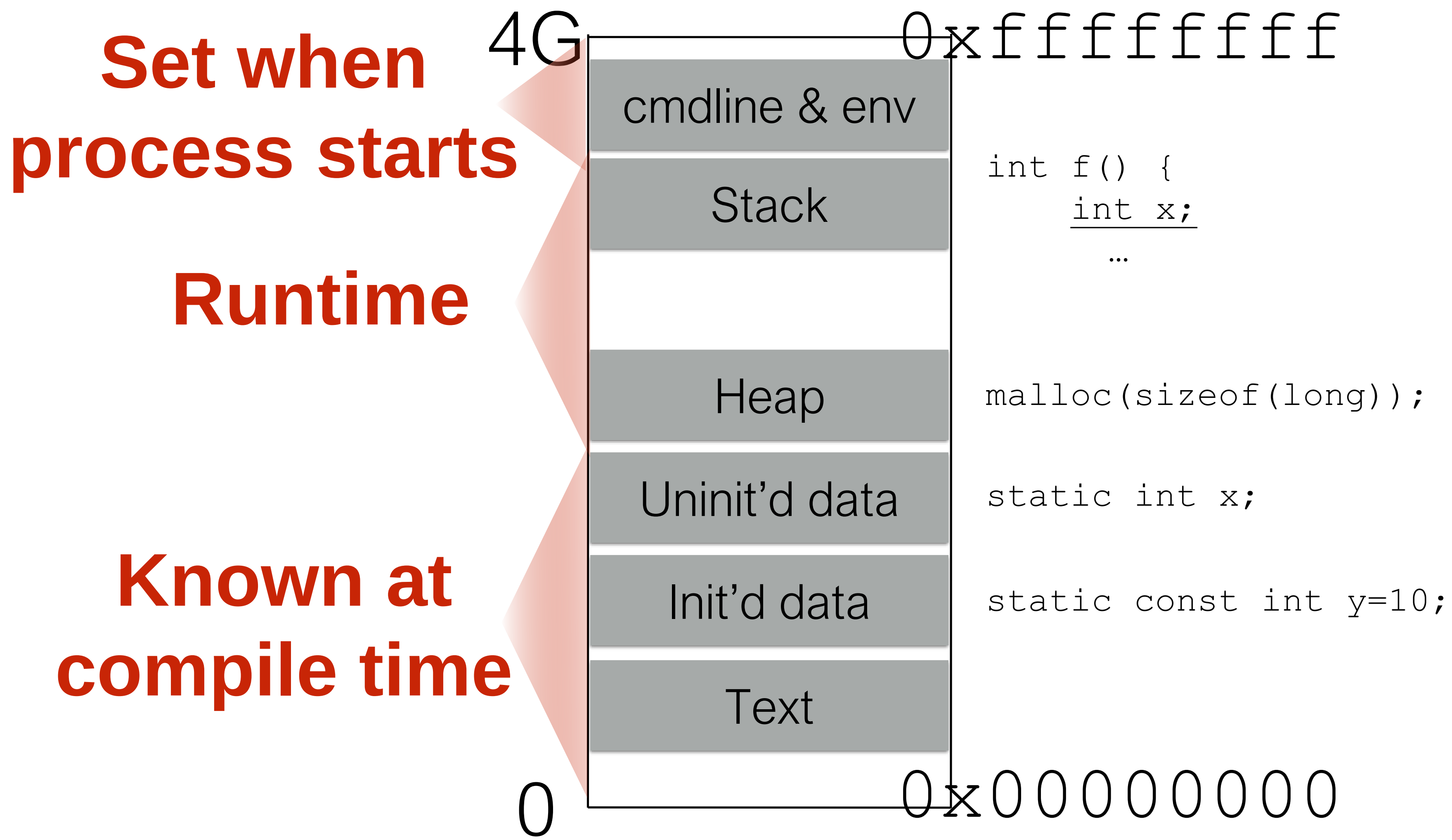
The *process's view* of memory is that it owns all of it

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

Program **instructions** are in memory



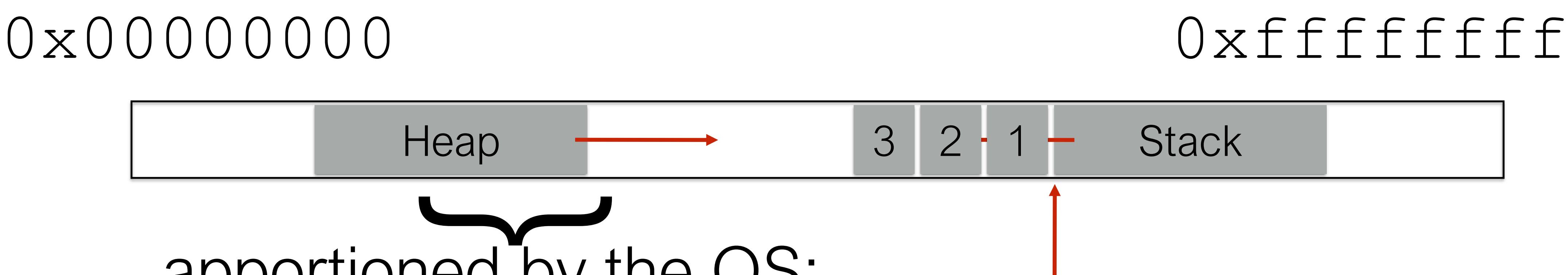
Location of data areas



Memory allocation

Stack and heap grow in opposite directions

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



apportioned by the OS;
managed in-process
by **`malloc`**

Stack
pointer

```
push 1
push 2
push 3

return
```

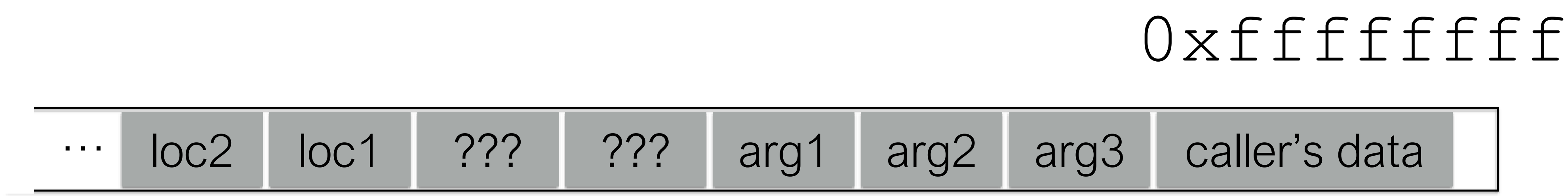
Focusing on the stack for now

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;
    ...
}
```



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

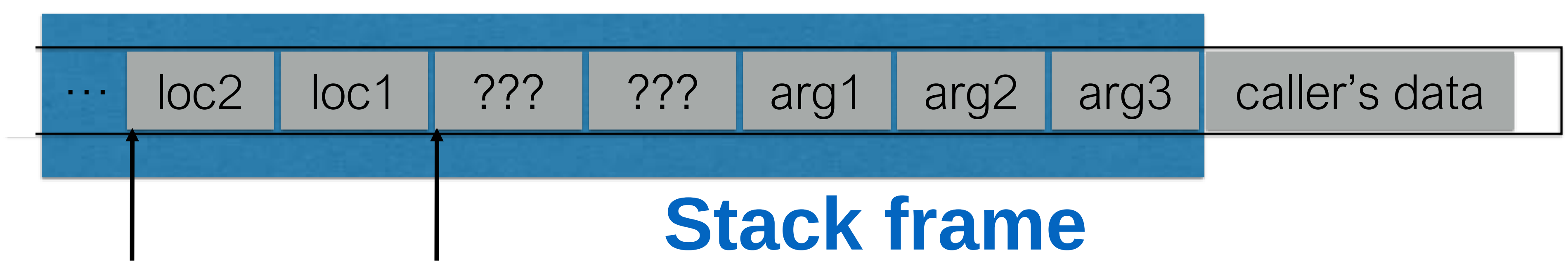
The local variable allocation is ultimately up to the compiler: Variables could be allocated in any order, or not allocated at all and stored only in registers, depending on the optimization level used.

Accessing variables

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

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

`0xffffffff`



`0xbffff323` `%ebp`

Frame pointer

Stack frame
for `func`

Can't know absolute address at compile time

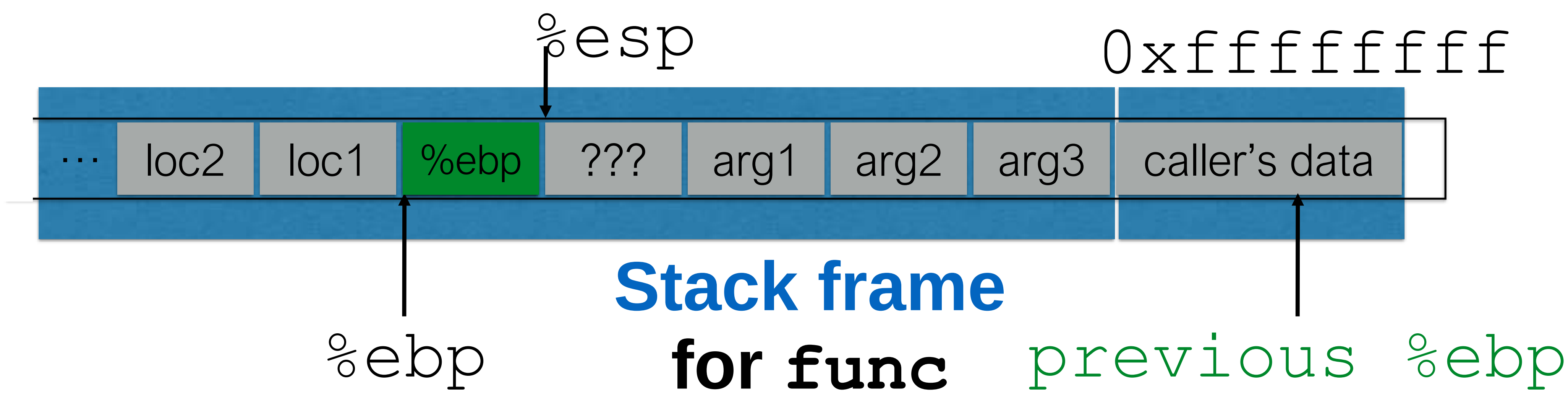
But can know the **relative** address

- **`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`

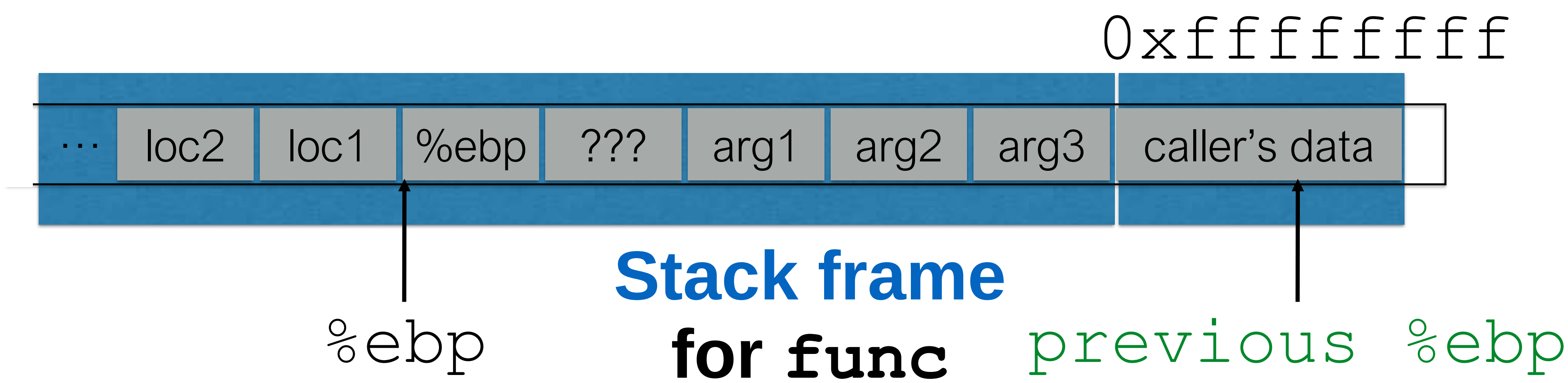
Set `%ebp` to current `%esp`

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

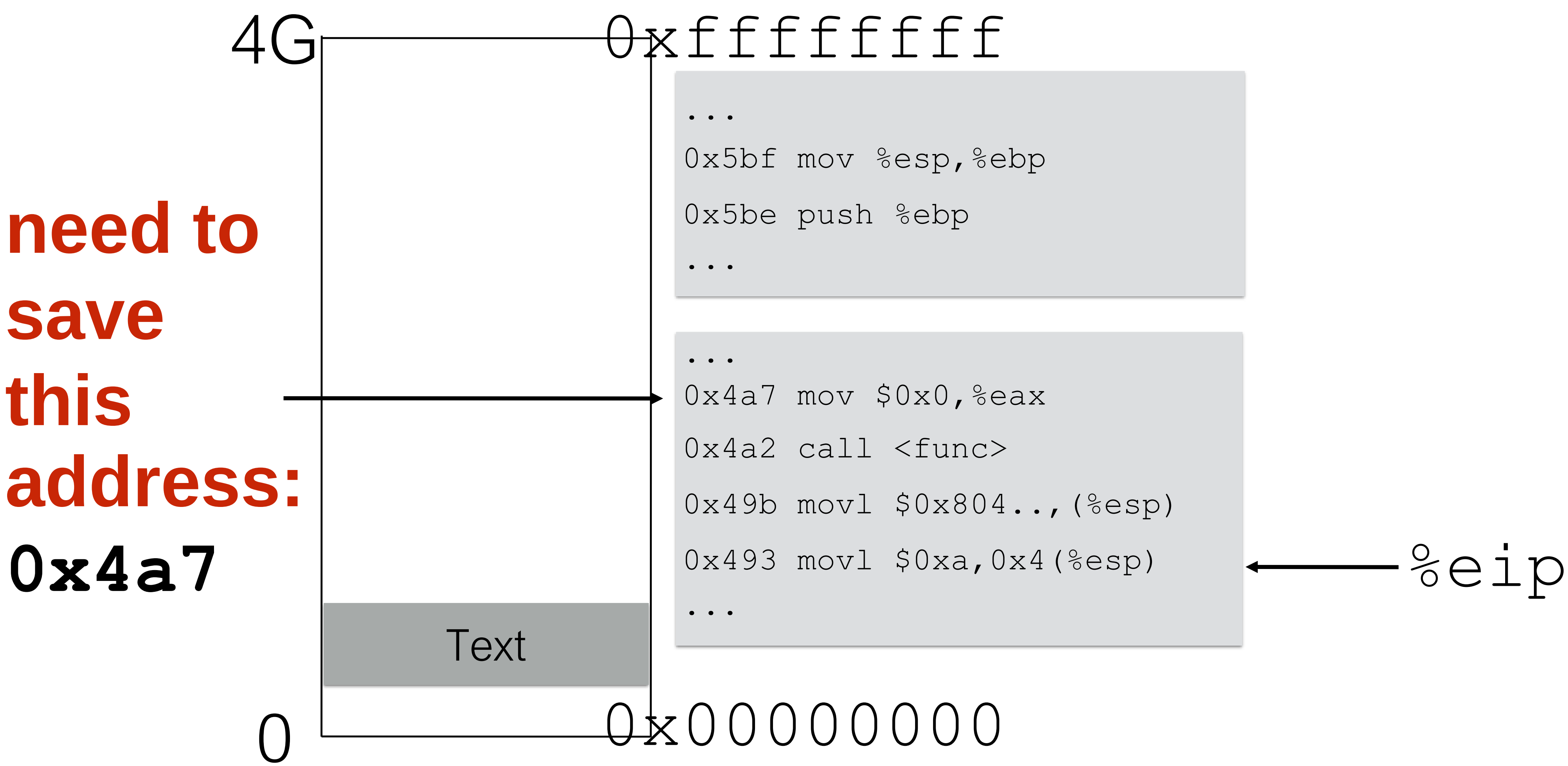
Returning from functions

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

Q: How do we resume here?



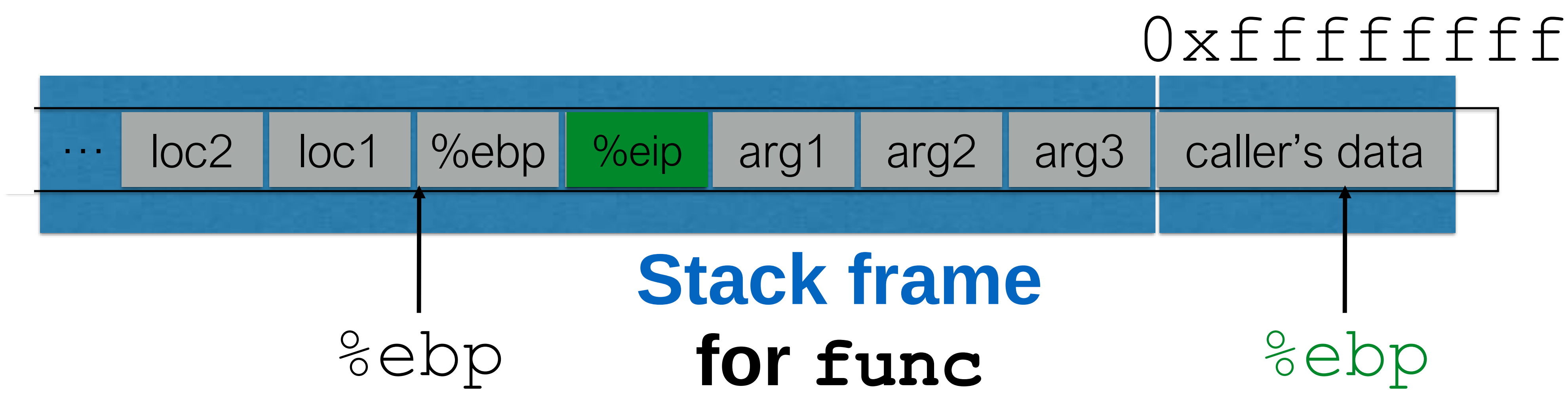
Instructions in memory



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

Buffer overflows from 10,000 ft

- **Buffer** =
  - Contiguous memory associated with a variable or field
  - Common in C
    - All strings are NULL-terminated arrays of chars
- **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);
    ...
}
```

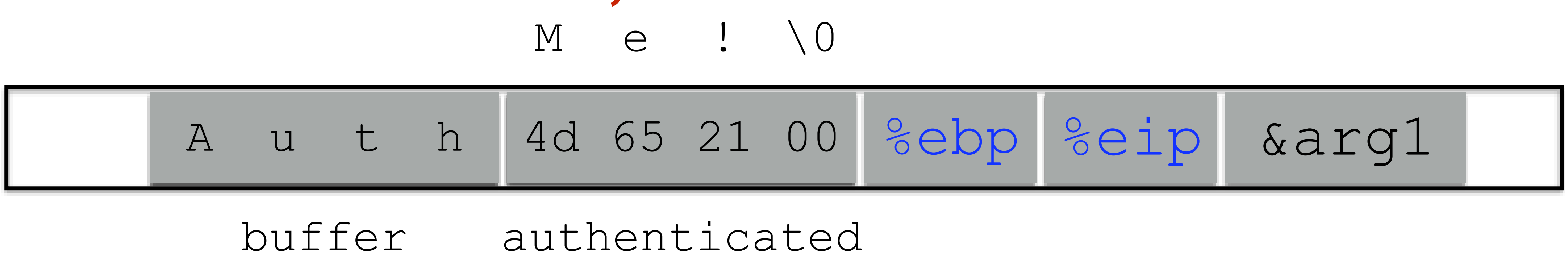
Upon return, sets `%ebp` to `0x0021654d`

SEGFAULT

Security-relevant outcome

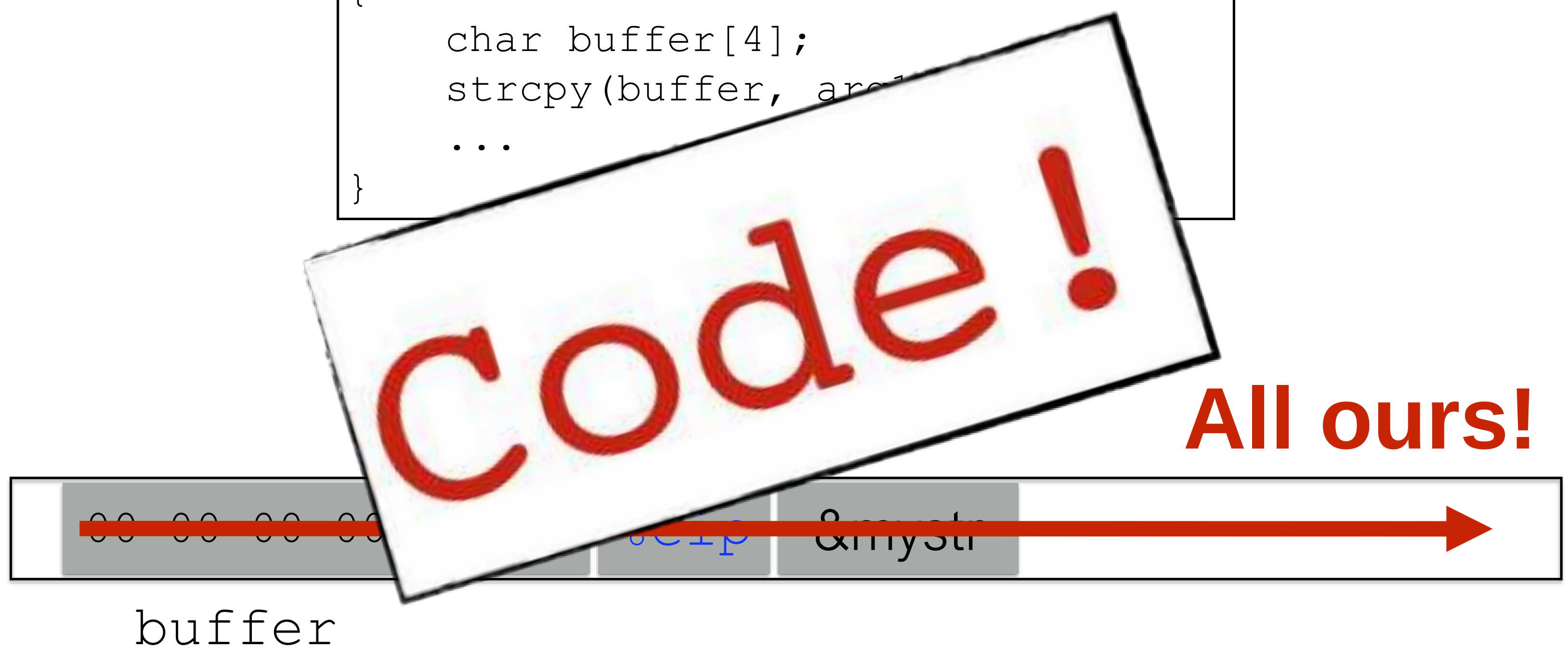
```
void func(char *arg1)
{
     int authenticated = 0;
     char buffer[4];
     strcpy(buffer, arg1);
     if(authenticated) { ...
}

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

Code still runs; user now ‘authenticated’



Could it be worse?

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



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

Aside: User-supplied strings

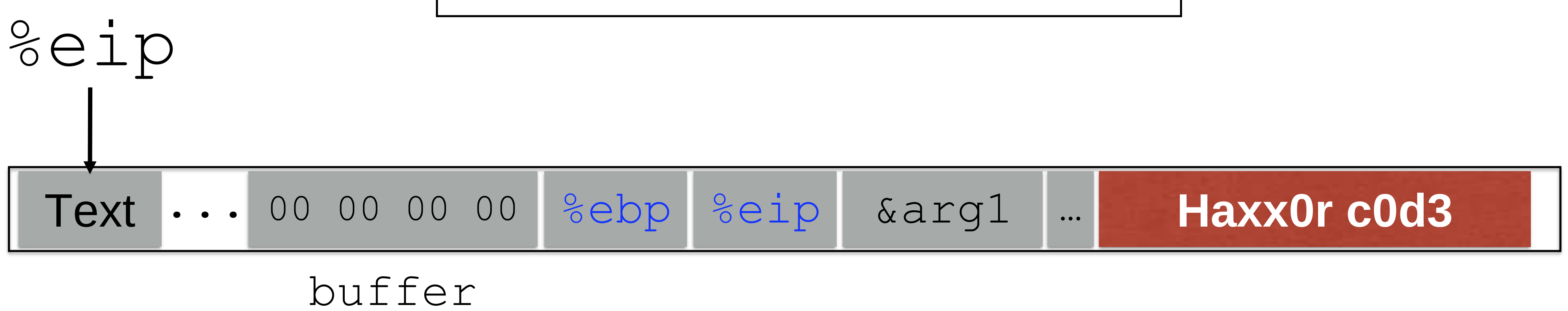
- These examples provide their own strings
- In reality strings come **from *users*** in myriad ways
  - Text input, packets, environment variables, file input…
- **Validating assumptions** about user input is critical!
  - We will discuss it later, and throughout the course

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);
    ...
}
```



(1) Load my own code into memory

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

Challenge 1
Loading code into memory

- It **must be 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**

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"
...
```

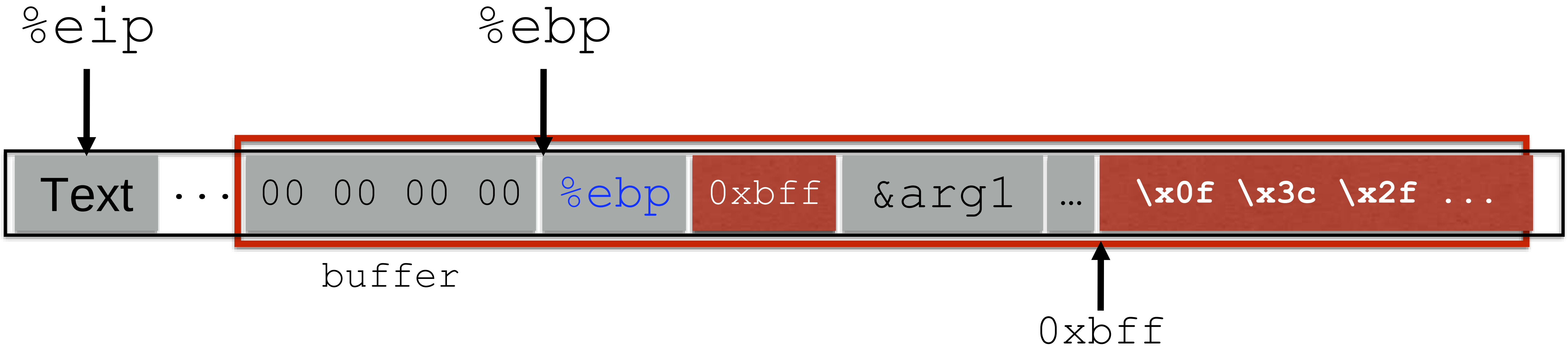
(Part of) your input

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

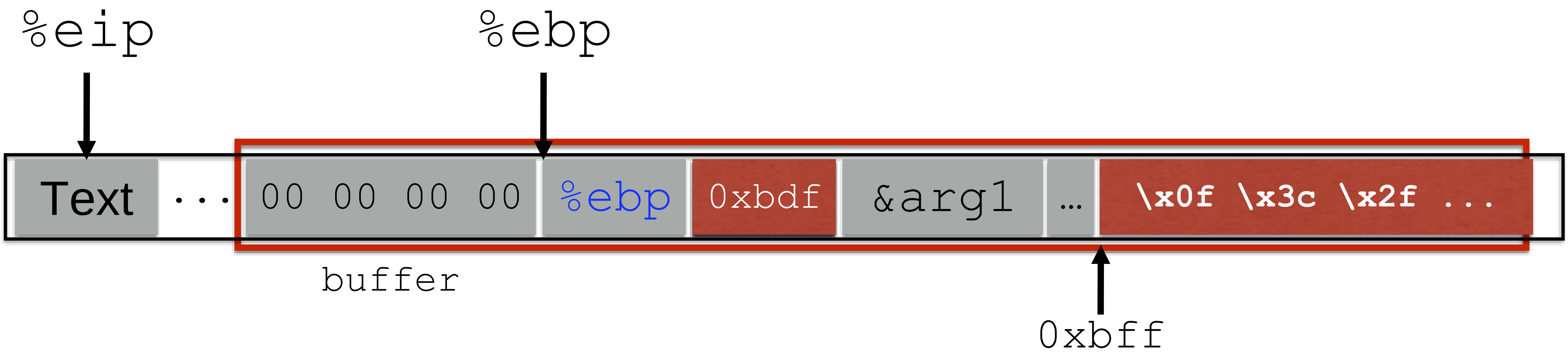
Hijacking the saved %eip



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

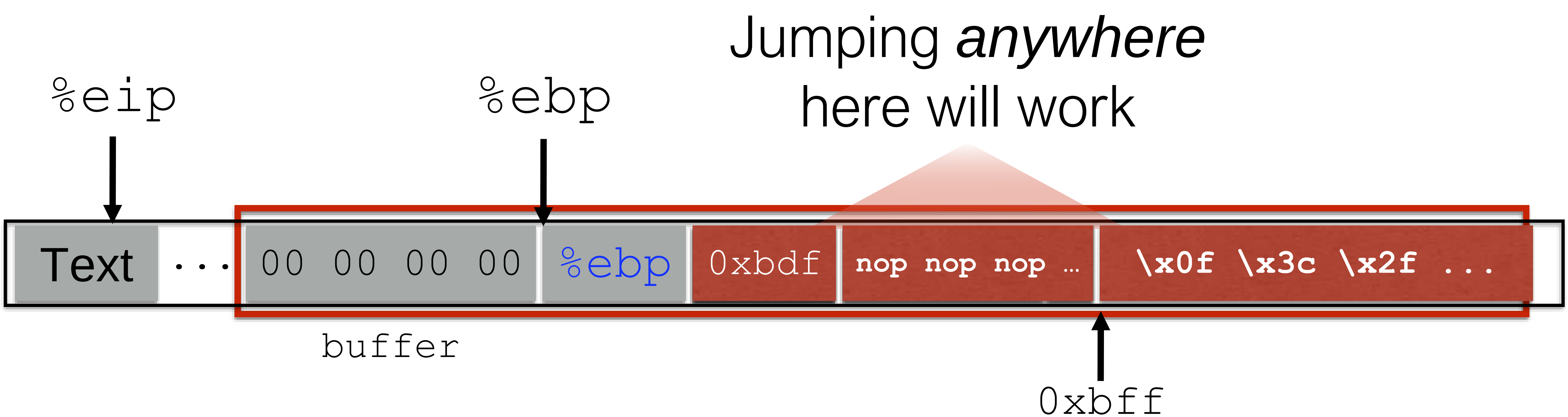
- If we don't have access to the code, we don't know how far the buffer is from the saved `%ebp`

- 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

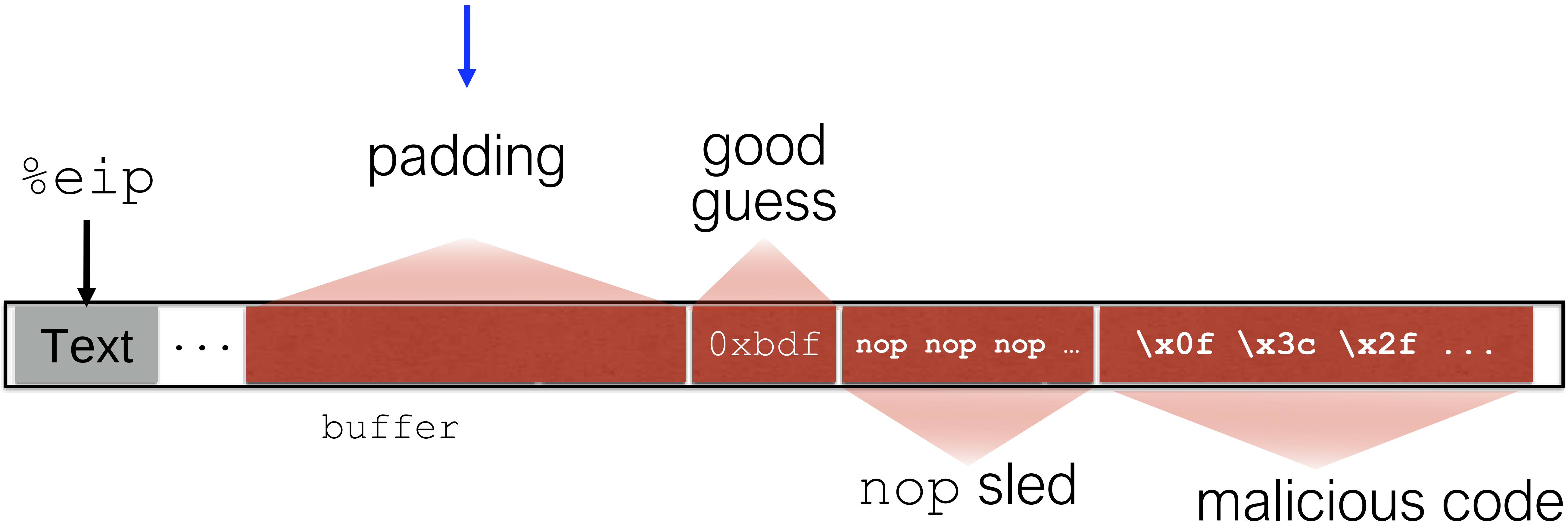
`nop` is a single-byte no-op instruction
(just moves to the next instruction)



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



Heap overflow

- Stack smashing overflows a stack-allocated buffer
- You can also **overflow a buffer** allocated by `malloc`, which resides on the **heap**
- Overflow into:
  - the C++ object *vtable*
  - adjacent objects
  - heap metadata

Integer overflow

```
void vulnerable()
{
  char *response;
  int nresp = packet_get_int();
    if (nresp > 0) {
     response = malloc(nresp*sizeof(char*));
     for (i = 0; i < nresp; i++)
      response[i] = packet_get_string(NULL);
    }
```

- What if we set `nresp` =1,073,741,824?
  - Assume `sizeof(char*)` = 4
- The `for` loop now creates an overflow! (int_max is 2,147,483,647)

Integer overflow

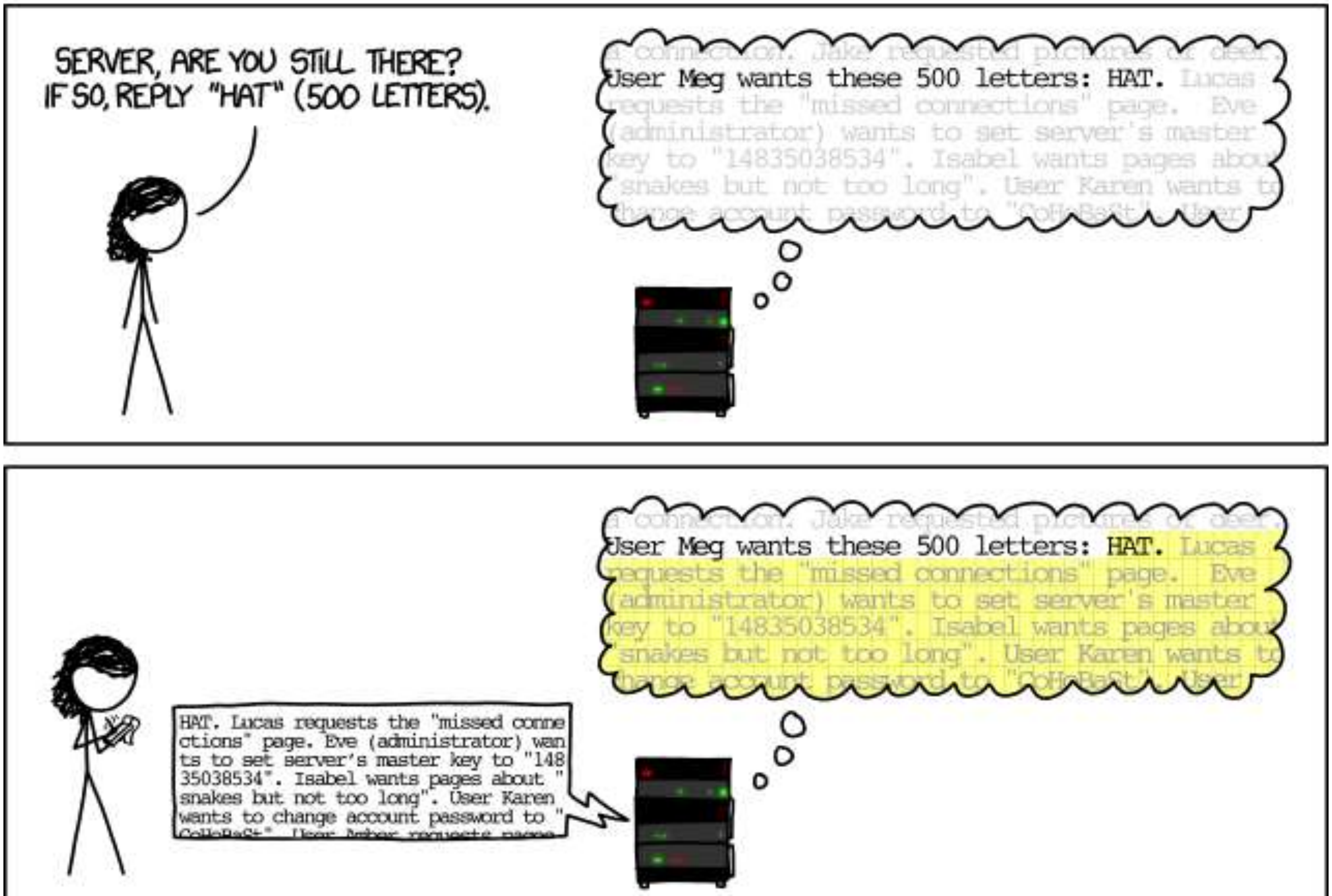
```
void vulnerable()
{
  char *response;
  int nresp = packet_get_int();
    if (nresp > 0) {
     response = malloc(nresp*sizeof(char*));
     for (i = 0; i < nresp; i++)
      response[i] = packet_get_string(NULL);
    }
```

- What if we set `nresp` =1,073,741,824?
  - Assume `sizeof(char*)` = 4
- The `for` loop now creates an overflow! (int_max is 2,147,483,647)

Read overflow

- Rather than permitting writing past the end of a buffer, a bug could permit **reading past the end**
- Might **leak secret information**

Heartbleed


SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
User Meg wants these 500 letters: HAT. Lucas
requests the "missed connections" page. Eve
(administrator) wants to set server's master
key to "14835038534". Isabel wants pages about
snakes but not too long". User Karen wants to
User Meg wants these 500 letters: HAT. Lucas
requests the "missed connections" page. Eve
(administrator) wants to set server's master
key to "14835038534". Isabel wants pages about
snakes but not too long". User Karen wants to
HAT. Lucas requests the "missed conne
ctions" page. Eve (administrator) wan
ts to set server's master key to "148
35038534". Isabel wants pages about "
snakes but not too long". User Karen
wants to change account password to "

Defenses

Attack commonalities

1. The attacker is able to **control some data** that is used by the program

2. The use of that data permits **unintentional access to some memory area** in the program
   - Past a buffer
   - To arbitrary positions on the stack / in the heap

How to get memory safety?

- The easiest way to avoid all of these vulnerabilities is to use a memory-safe language

- Modern languages are memory safe
  - Java, Python, C#, Ruby
  - Haskell, Scala, Go, Objective Caml, Rust

- In fact, these languages are **type safe,** which is even better (more on this shortly)

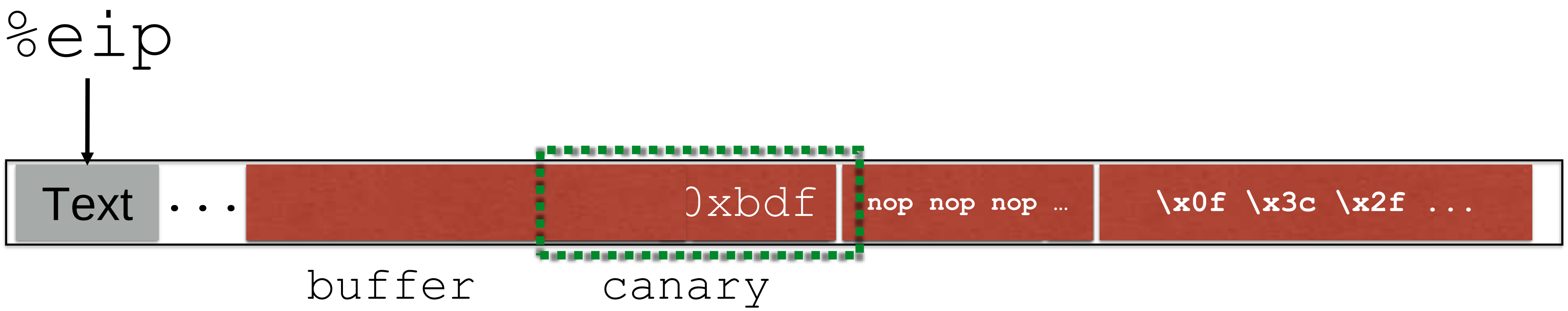
Detecting overflows with canaries

19th century coal mine integrity

- Is the mine safe?
- Dunno; bring in a canary
- If it dies, abort!

We can do the same for stack integrity!

Detecting overflows with canaries



Check canary just before every function return.

Not the expected value: abort!

What value should the canary have?

Canary values

1. Terminator canaries (CR, LF, NUL (i.e., 0), -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

Avoiding exploitation

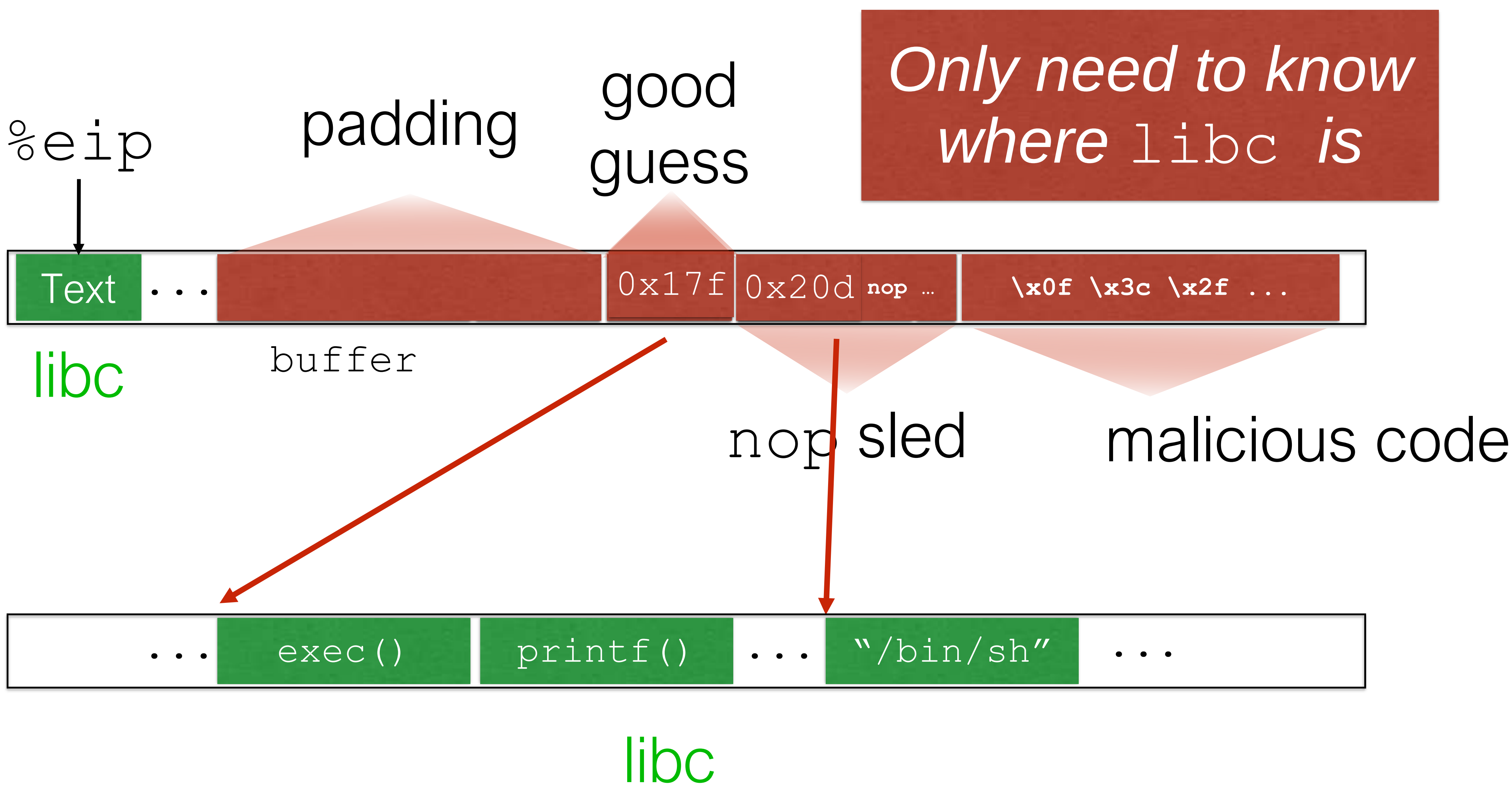
Recall the steps of a stack smashing attack:

- Putting attacker code into memory
  Defense: Stack Canaries
- Getting `%eip` to point to an address you specify
- Finding the correct address

How can we make these attack steps more difficult?

- Goal: Don’t run attacker code
- Defense: Make stack non-executable
    - Try to jump to attacker shellcode in the stack, panic instead

Return-to-libc



Avoiding exploitation

Recall the steps of a stack smashing attack:

- Putting attacker code into memory

  Defense: Stack Canaries

- Getting `%eip` to point to address you specify

  Defense: Non-executable stack (kind of)

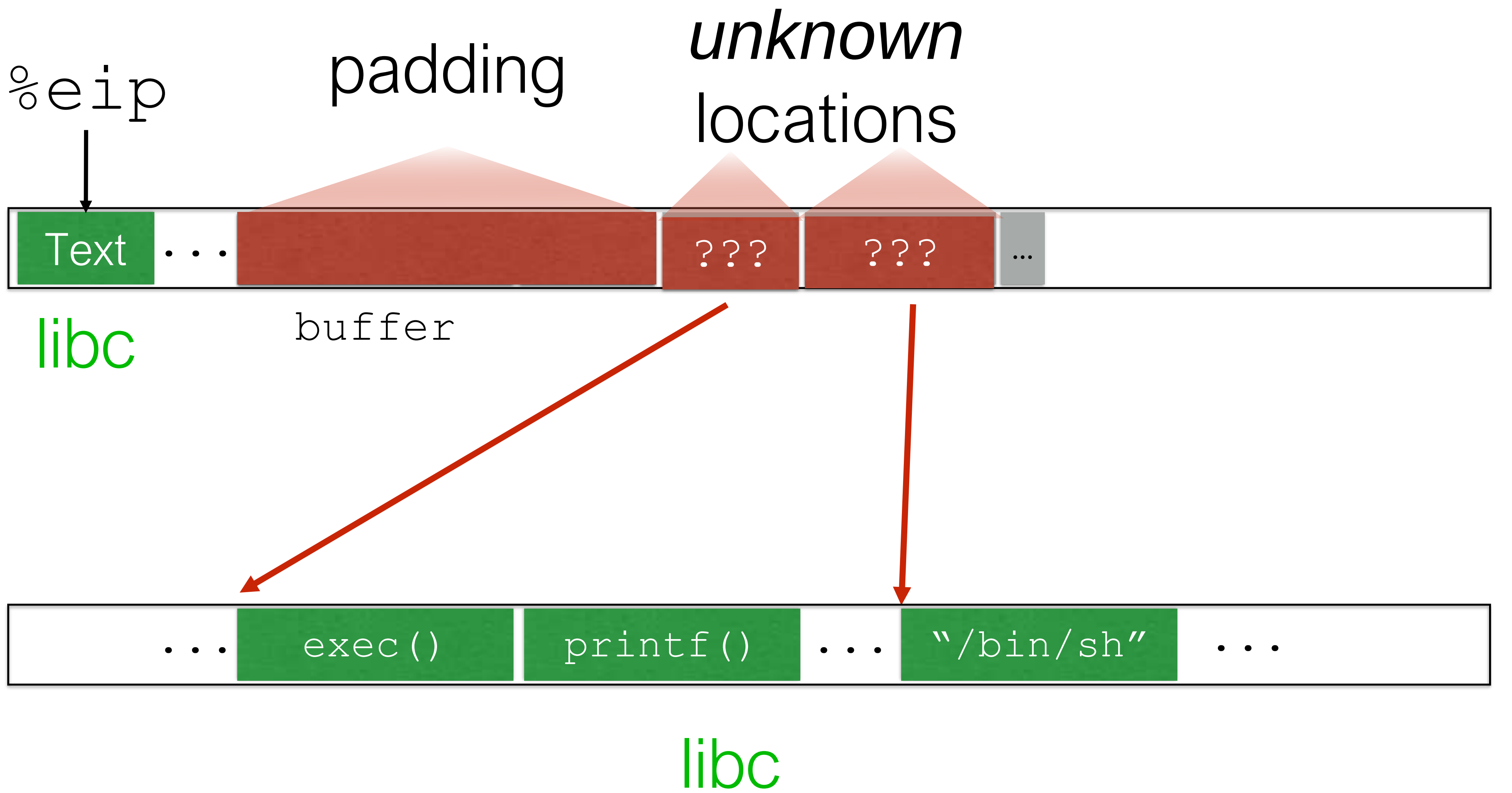
- Finding the correct address

How can we make these attack steps more difficult?

Address-space layout randomization (ASLR)

- Randomly place some elements in memory
- Make it hard to find libC functions
- Make it hard to guess where stack (shellcode) is

Return-to-libc, thwarted



Return-oriented Programming

- Idea: rather than use a single (libc) function to run your shellcode, **string together pieces of existing code, called *gadgets***, to do it instead

- Challenges
  - **Find the gadgets** you need
  - **String them together**