Software Security

Software Security

- Software security is a kind of computer security that focuses on the **secure design and implementation of software**
  - Using the best languages, tools, methods
- ***Focus*** *of study:*

the code

- By contrast: Many popular approaches to security treat software as a *black box* (ignoring the code)
  - OS security, anti-virus, firewalls, etc.

Why Software Security?

Firewalls and anti-virus are like building walls around a weak interior

Attackers often can bypass outer defenses to attack weaknesses within

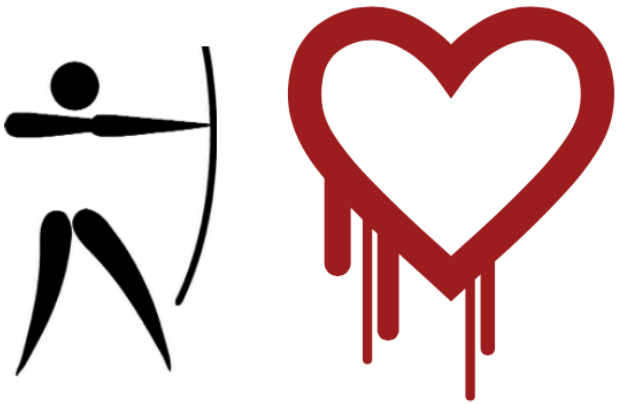
Software Security aims to address weaknesses directly

Ex: Heartbleed

- SSL/TLS is a core **protocol** for **encrypted communications** used by the web
- Heartbleed is a **bug** in the commonly used **OpenSSL** implementation of SSL/TLS, v1.0.1 - 1.0.1f
  - Discovered in March 2014, it has been in released code since March 2012 (**2 years old!**)
- A carefully crafted packet causes OpenSSL to read and return portions of a vulnerable server's memory
  - Leaking passwords, keys, and other private information

Heartbleed, meet SoftSec

- **Black box security is incomplete against Heartbleed exploits**
  - Issue is not at the level of system calls or deposited files: nothing the OS or antivirus can do
  - Basic attack packets could be blocked by IDS, but
    - "Packet chunking" may bypass basic filters
    - Exfiltrated data on the encrypted channel; invisible to forensics
- **Software security** methods attack the **source** of the problem: **the buggy code**



Low-level Vulnerabilities

- Programs written in **C and C++** are susceptible a variety of dangerous **vulnerabilities**
  - **Buffer overflows**
    - On the stack
    - On the heap
    - Due to integer overflow
    - Over-writing and over-reading
  - **Format string mismatches**
  - **Dangling pointer dereferences**

Attacks

- *Stack smashing*
- *Format string attack*
- *Stale memory access*
- *Return-oriented Programming (ROP)*

- All **violations** of **memory safety**
  - Accesses to memory via pointers that don't *own* that memory

Ensuring Memory Safety

- The easiest way to avoid these vulnerabilities is to **use a memory-safe programming language**
  - Better still: a ***type*-safe** language
- For C/C++, use **automated defenses**
  - *Stack canaries*
  - *Non-executable data (aka W+X or DEP)*
  - *Address space layout randomization (ASLR)*
  - *Memory-safety enforcement (e.g., SoftBound)*
  - *Control-flow Integrity (CFI)*
- and **safe programming patterns and libraries**
  - *Key idea*: **validate untrusted input**

Securing the WWW

- Cybersecurity battles rage on the **world wide web**
- There are new **vulnerabilities** and **attacks**
  - *SQL injection*
  - *Cross-site scripting (XSS)*
  - *Cross-site request forgery (CSRF)*
  - *Session hijacking*

- The **defenses** have a **similar theme**
  - Careful who/what you trust: **Validate input**
  - Reduce the possible damage, make exploitation harder

Low-level security

or

C and the infamous **buffer overflow**

What is a buffer overflow?

- A buffer overflow is a **bug** that affects low-level code, typically in **C** and **C++**, with **significant security implications**
- **Normally**, a program with this bug will simply **crash**
- But an **attacker** can alter the situations that cause the program to **do much worse**
  - **Steal** private information (e.g., Heartbleed)
  - **Corrupt** valuable information
  - **Run code** of the attacker's choice

Why study them?

- Buffer overflows are still **relevant** today
  - C and C++ are still popular
  - Buffer overflows still occur with regularity
- They have a **long history**
  - Many different approaches developed to defend against them, and bugs like them
- They share **common features with other bugs** that we will study
  - In **how the attack works**
  - In **how to defend against it**

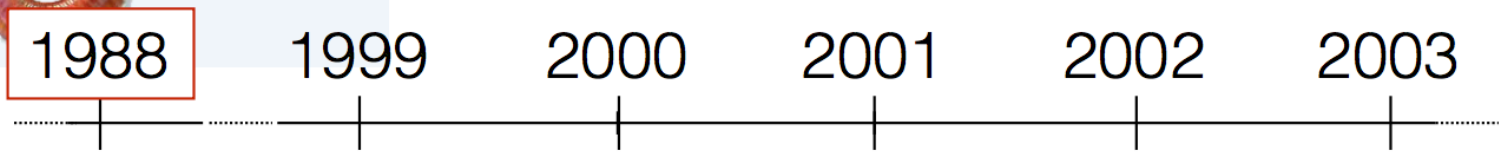
Critical systems in C/C++

- 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
- Many **embedded systems**
  - Mars rover, industrial control systems, automobiles

A successful attack on these systems is particularly dangerous!

History of buffer overflows

The harm has been substantial

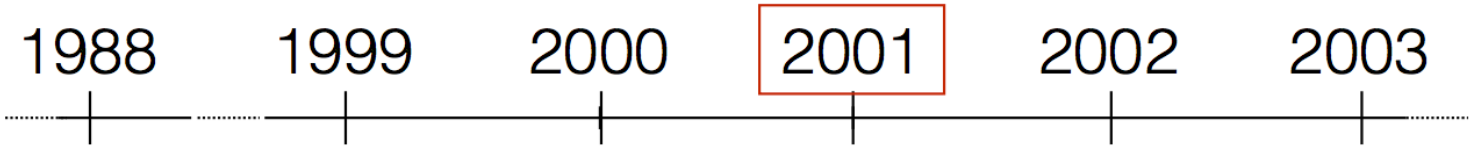


- **Morris worm**
  - Propagated across machines (too aggressively, thanks to a bug)
  - One way it propagated was a **buffer overflow** attack against a vulnerable version of `fingerd` on VAXes
    - Sent a special string to the finger daemon, which caused it to execute code that created a new worm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

Morris now a professor at MIT

History of buffer overflows

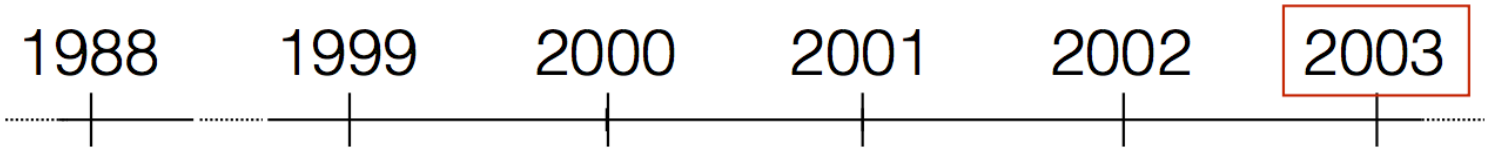
The harm has been substantial



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

History of buffer overflows

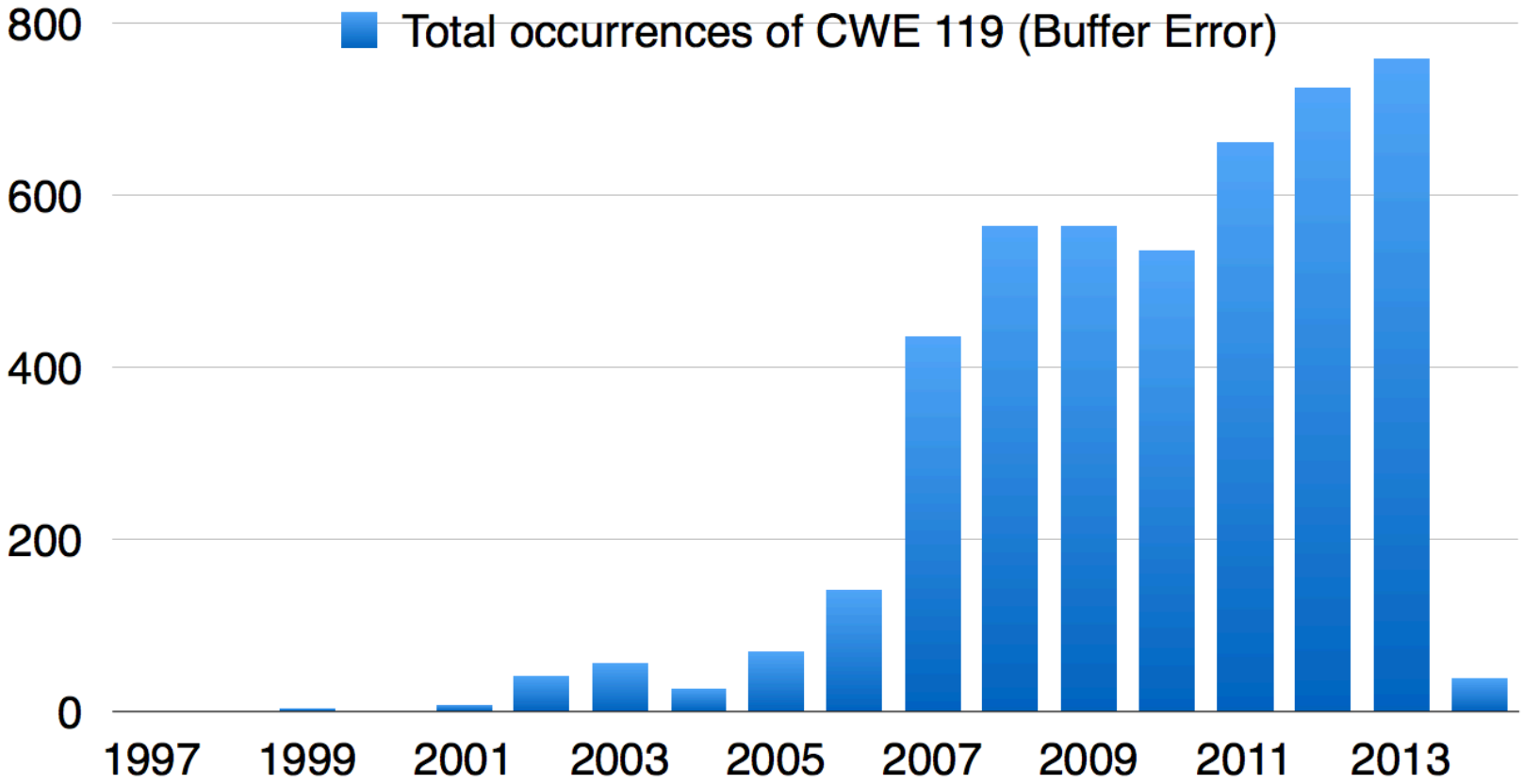
The harm has been substantial



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


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.

Trends



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

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

Memory layout

All programs are stored in memory

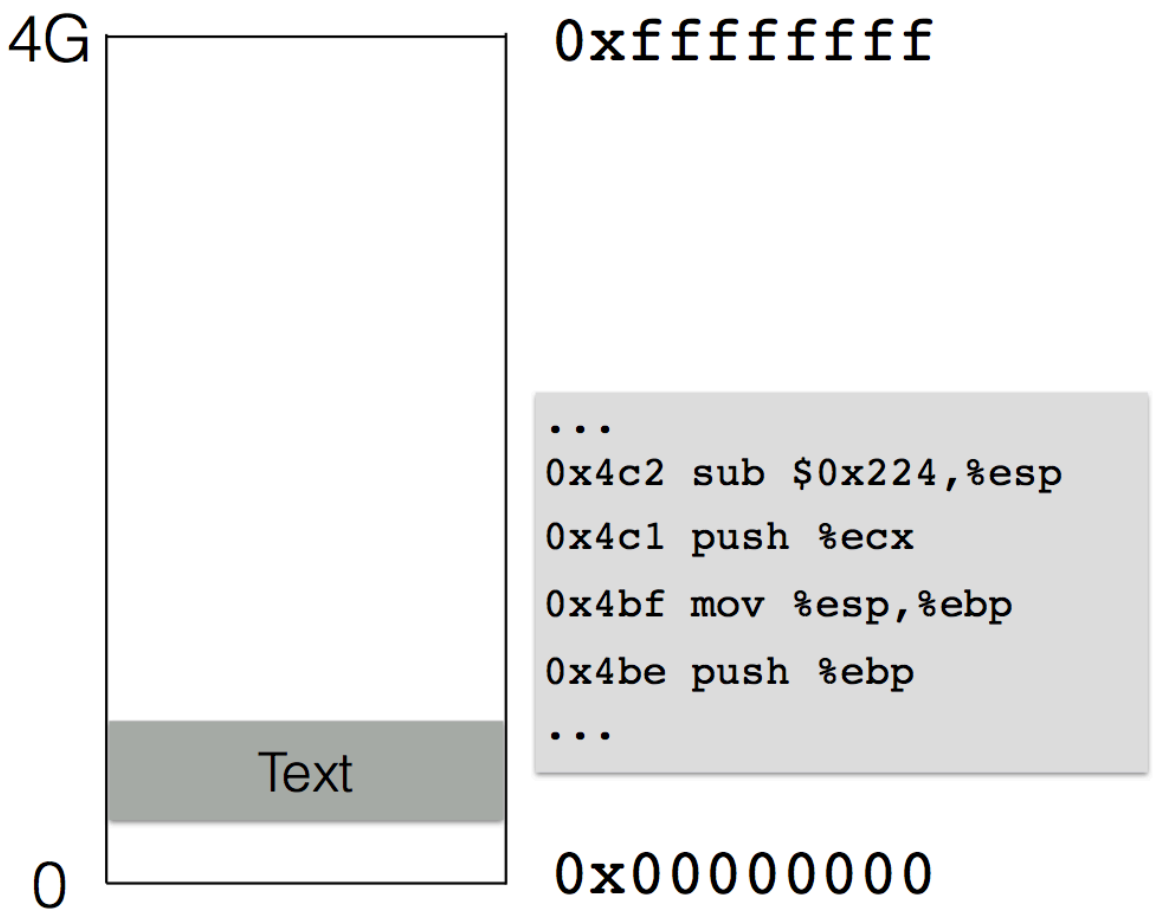
The *process's view* of memory is that it owns all of it

0xffffffff

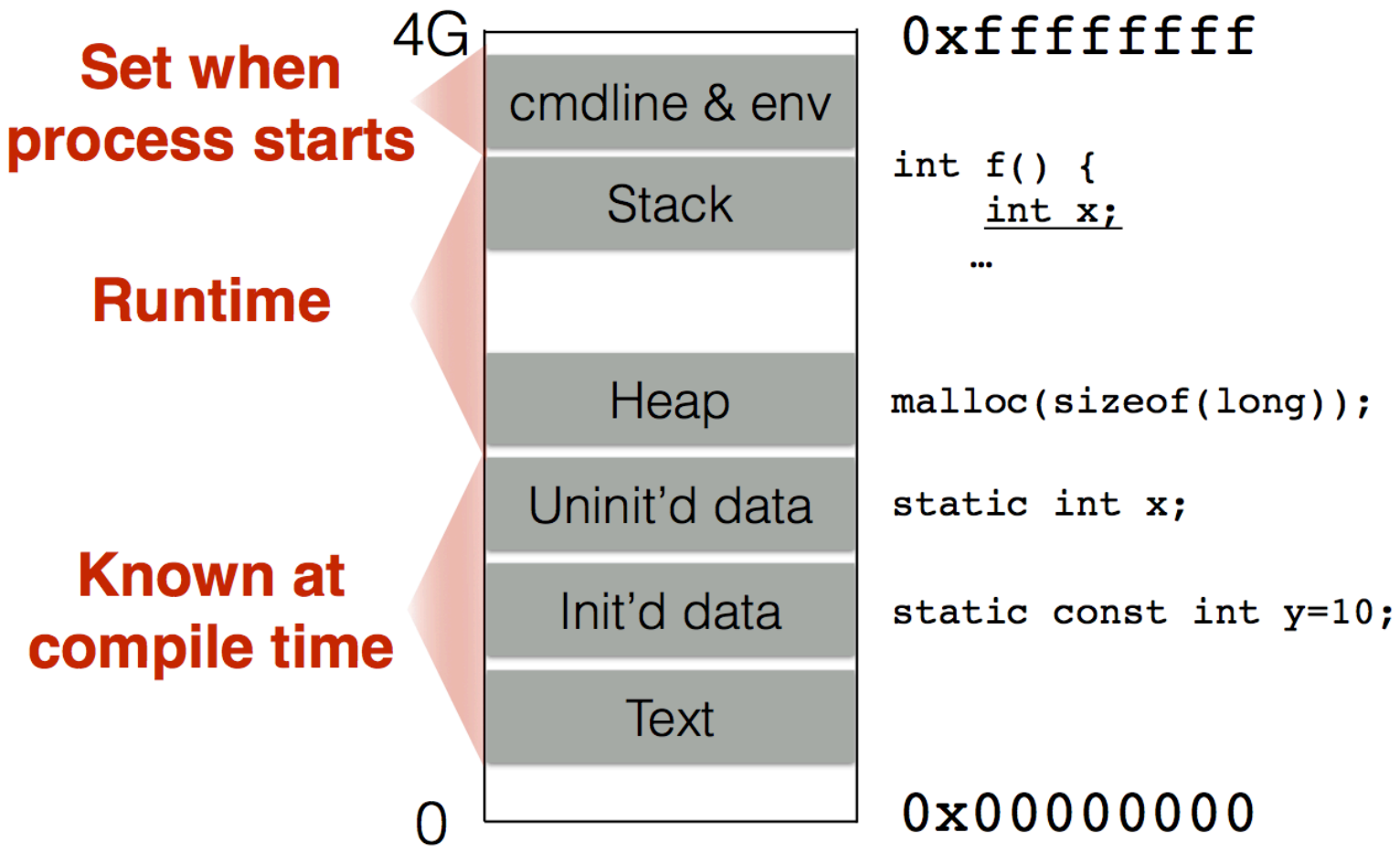
In reality, these are *virtual addresses*; the OS/CPU map them to physical addresses

0x00000000

The instructions themselves are in memory



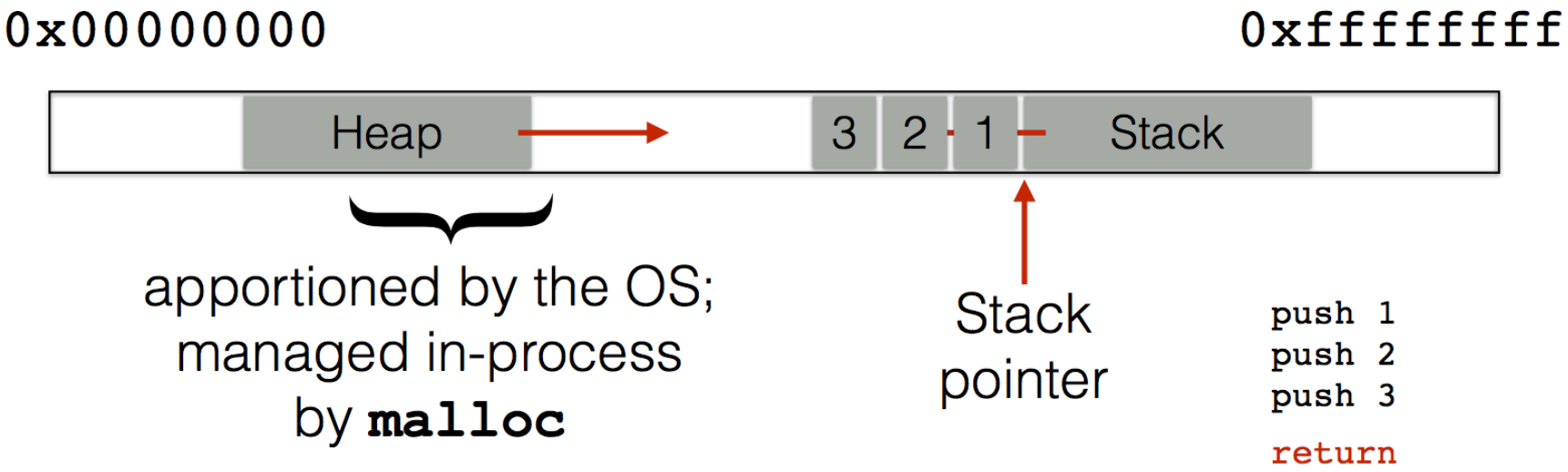
Location of data areas



Memory allocation

Stack and heap grow in opposite directions

Compiler emits instructions
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

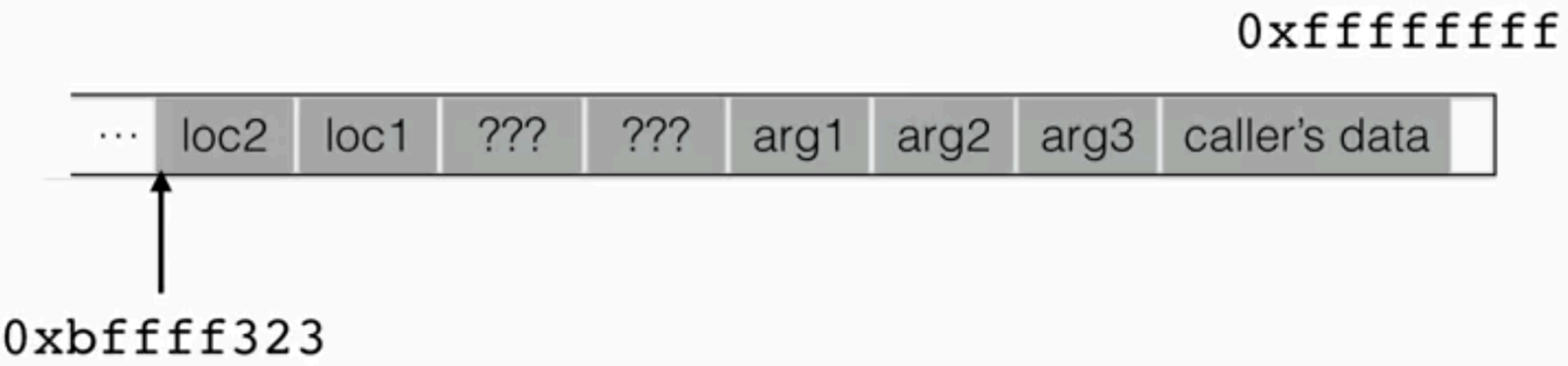
Arguments pushed in reverse order of code

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`?



Can't know absolute address at compile time

But can know the **relative** address

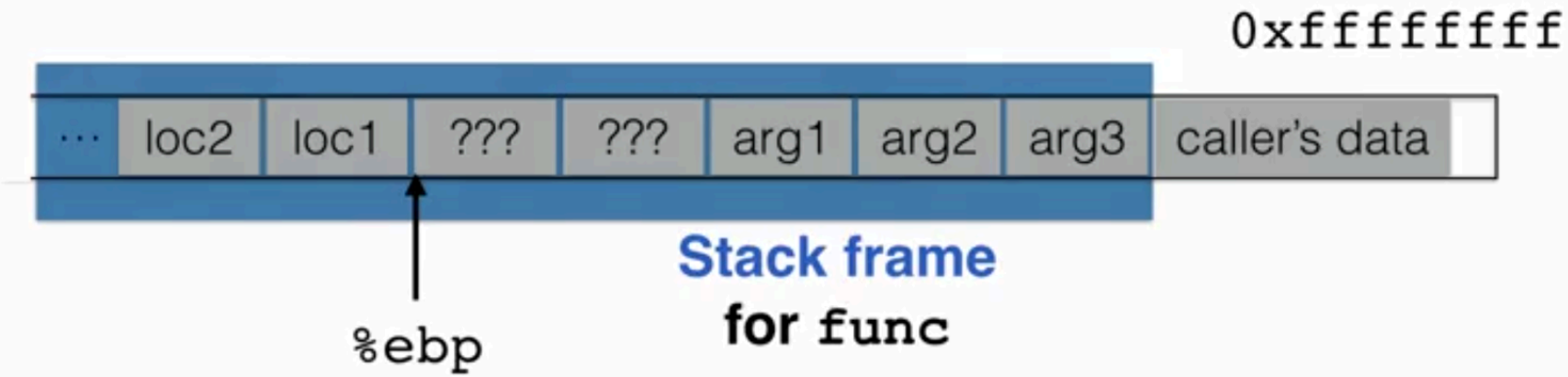
- **`loc2`** is always 8B before ???s

Accessing variables

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

Q: Where is (this) `loc2`?

A: -8(%ebp)



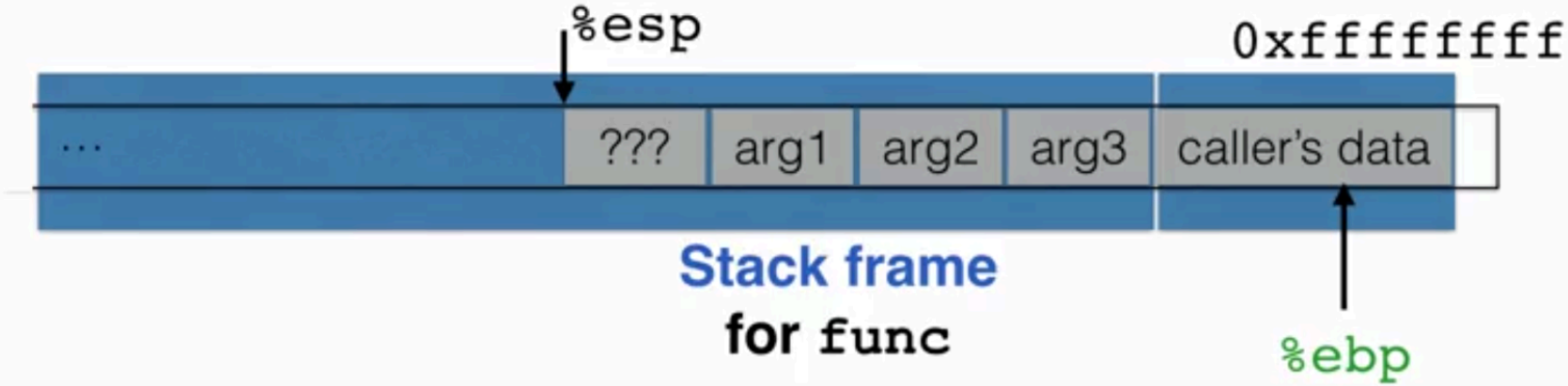
But can know the **relative** address

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

Returning from functions

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

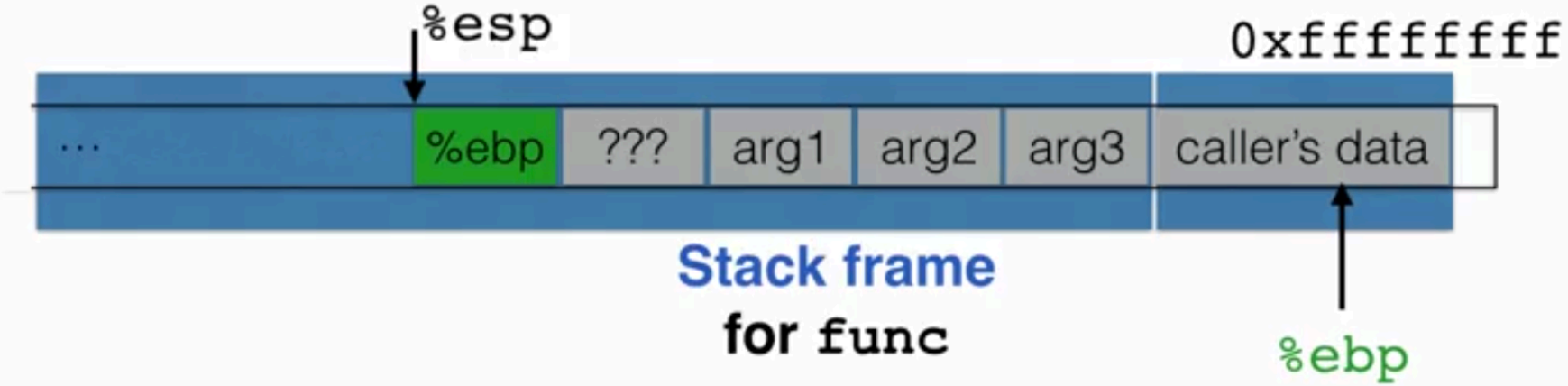
Q: How do we restore %ebp?



Returning from functions

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

Q: How do we restore %ebp?

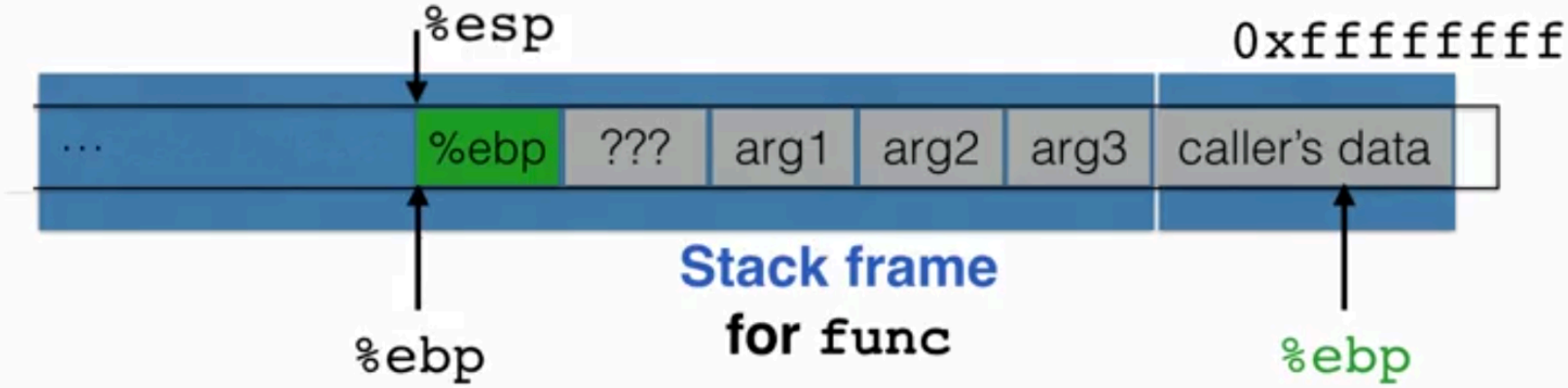


Push `%ebp` before `locals`

Returning from functions

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

Q: How do we restore %ebp?



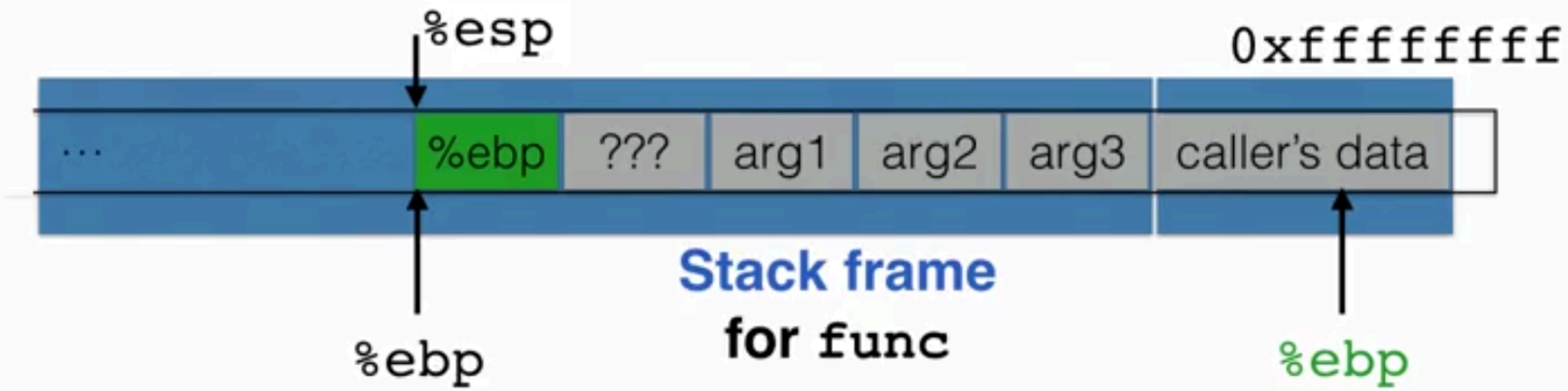
Push `%ebp` before `locals`

Set %ebp to current (%esp)

Returning from functions

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

Q: How do we restore %ebp?

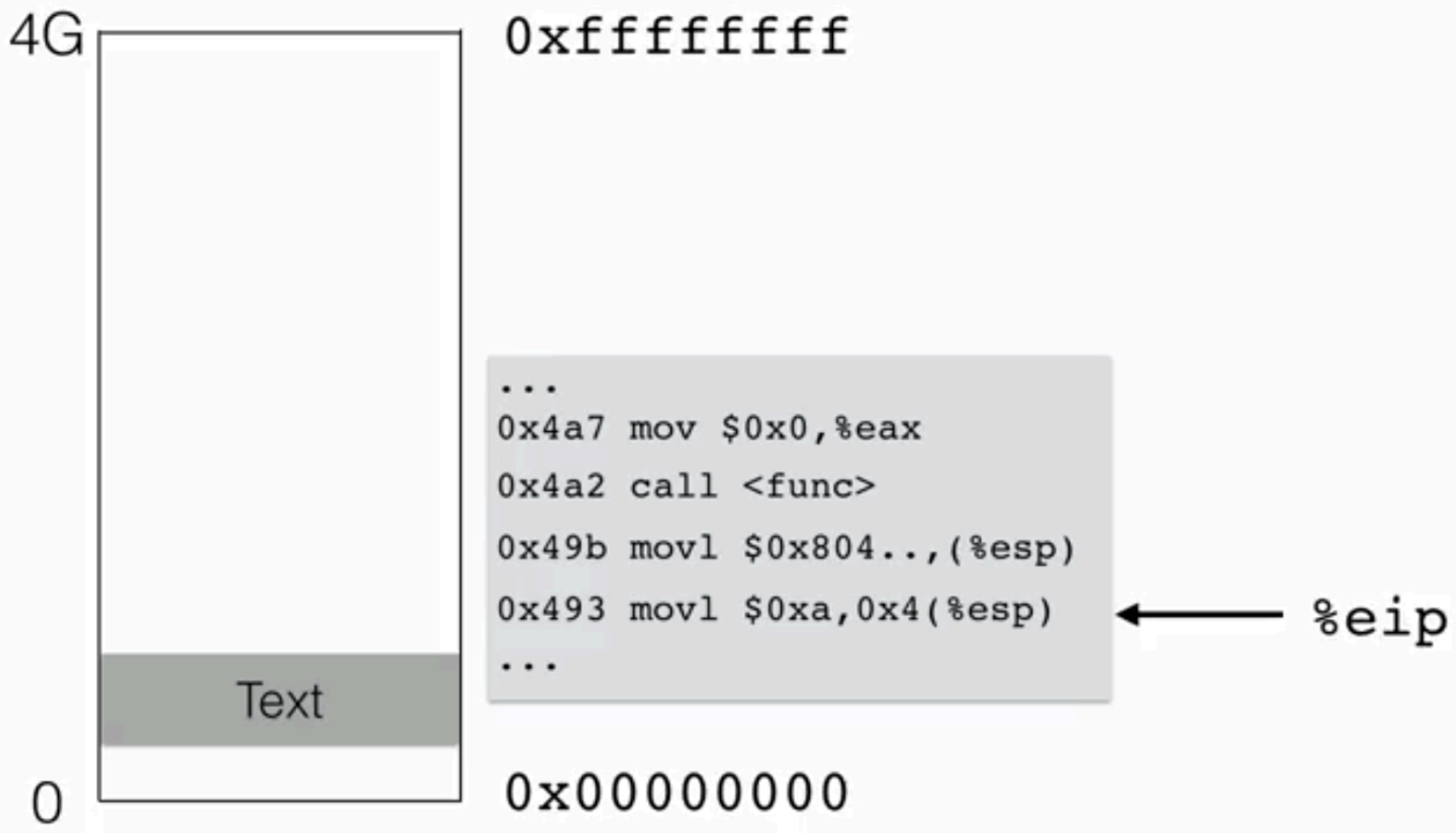


Push `%ebp` before `locals`

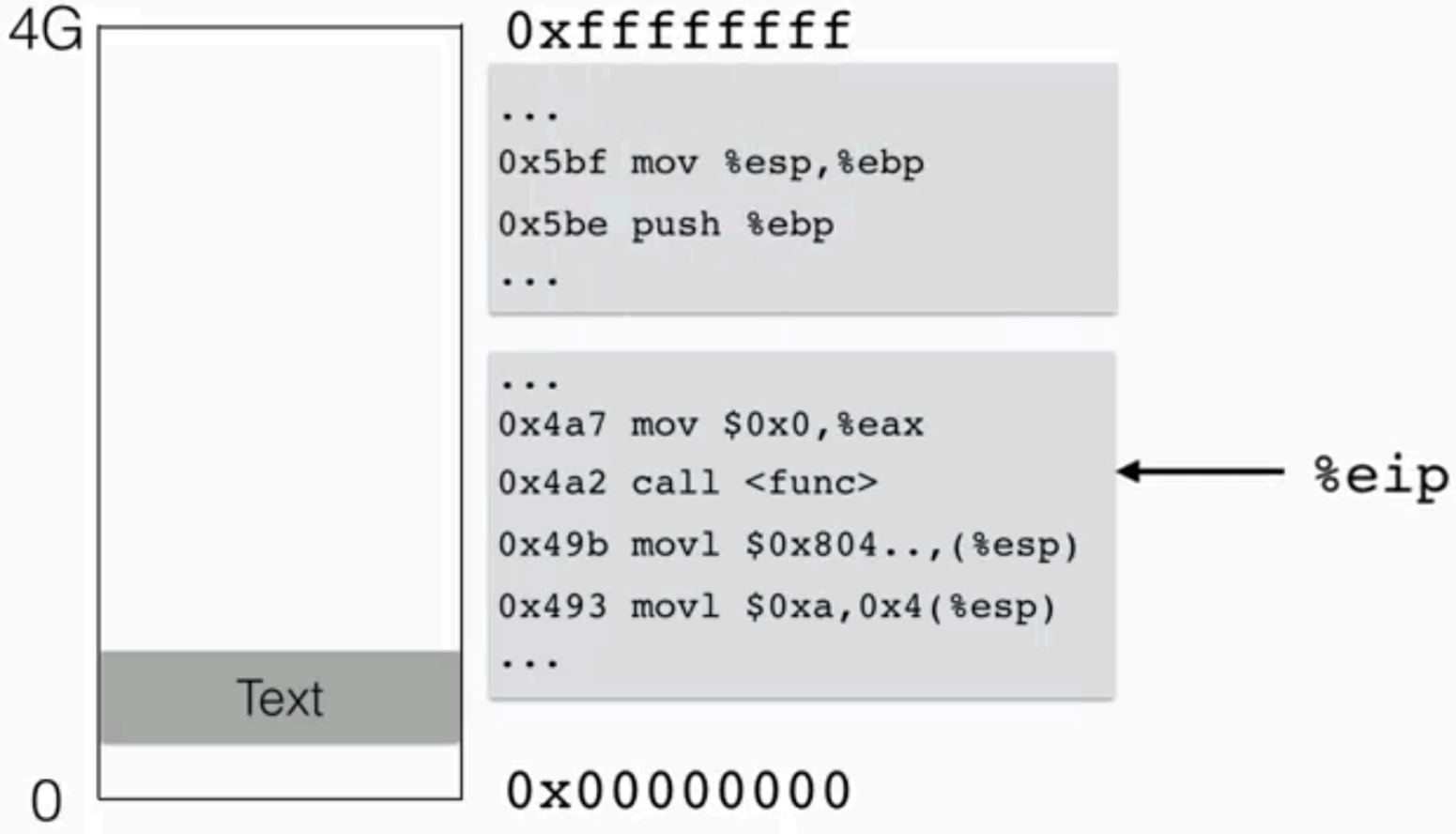
Set %ebp to current (%esp)

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

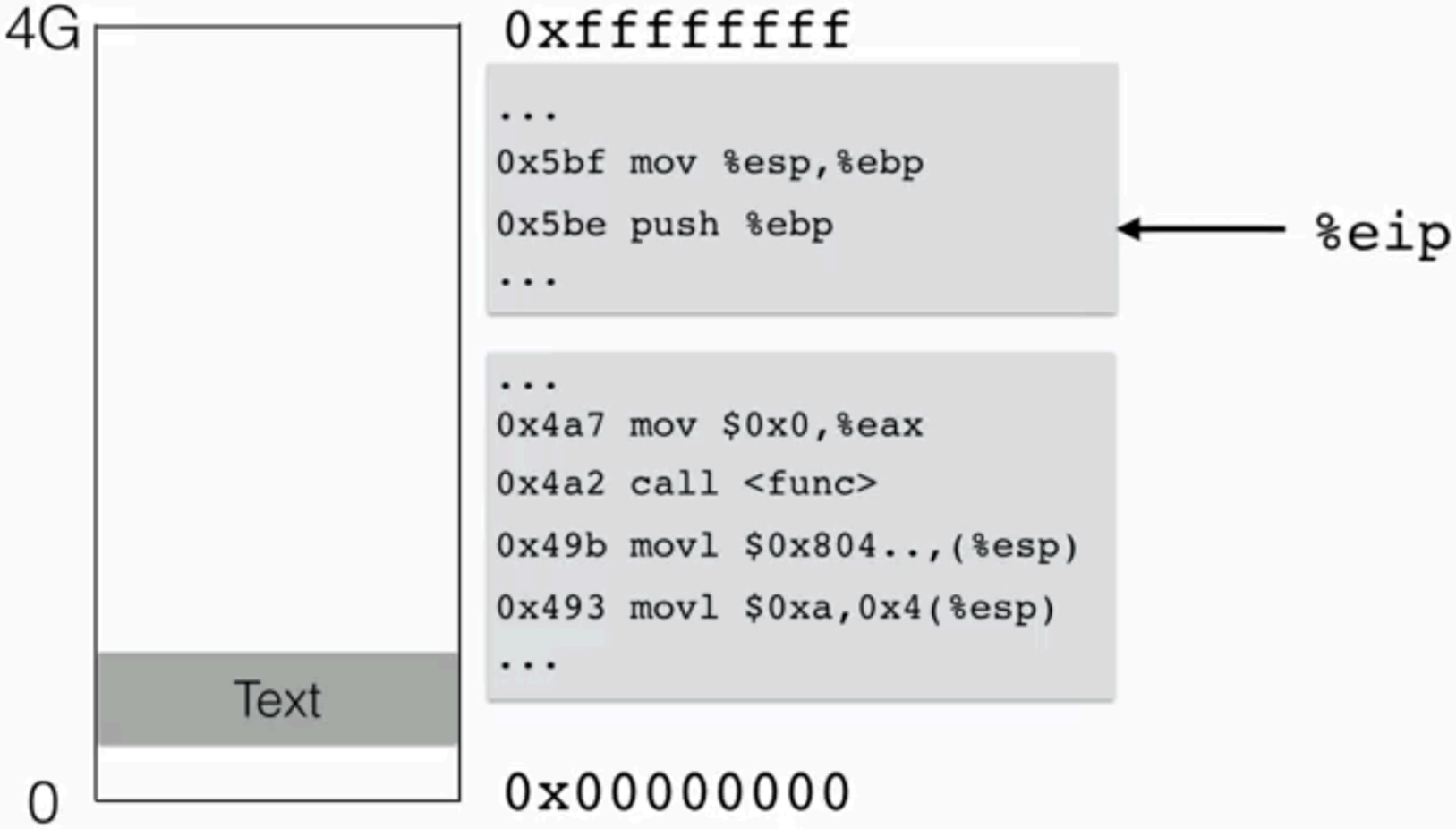
Instructions in memory



Instructions in memory



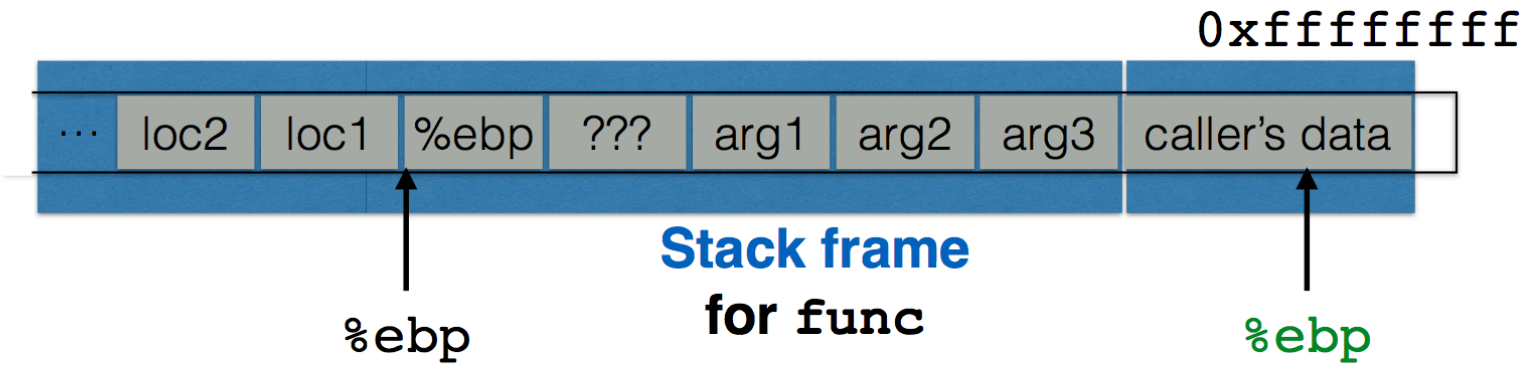
Instructions in memory



Returning from functions

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

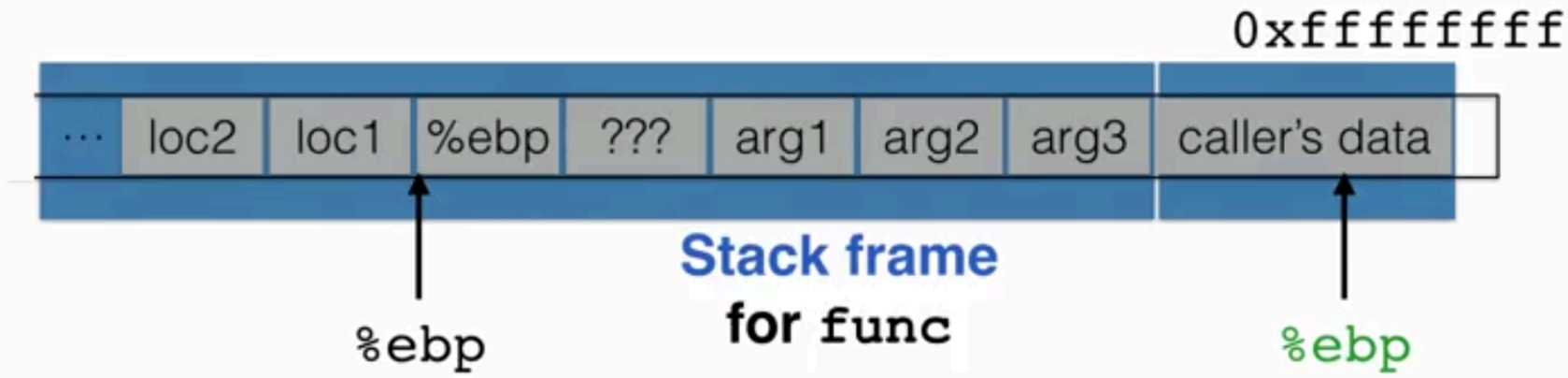
Q: How do we resume here?



Returning from functions

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

Q: How do we resume here?

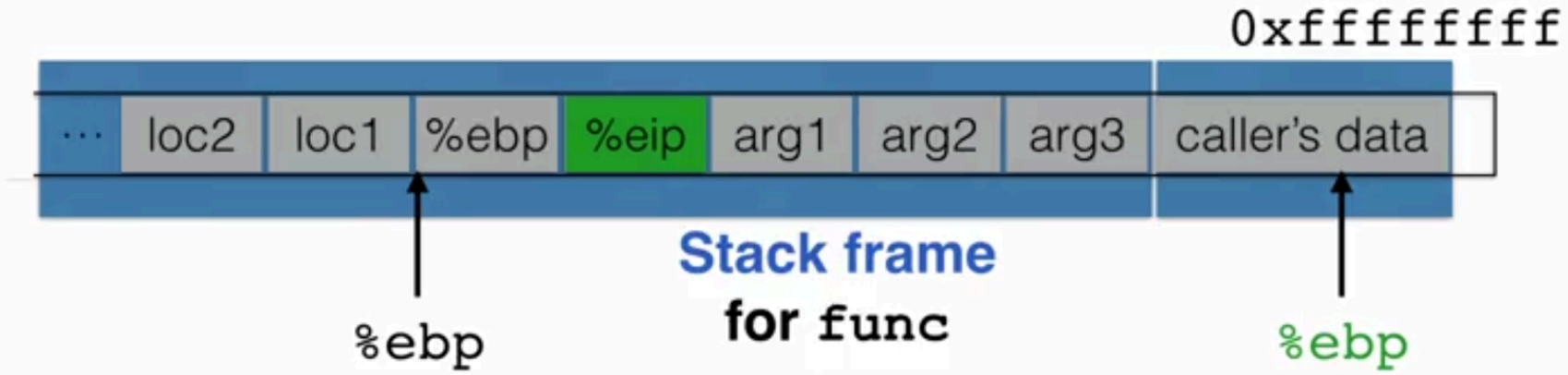


Push next `%eip` before `call`

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** (%ebp) to where the end of the stack is right now (%esp)
6. **Push local variables** onto the stack

Returning function:

7. **Reset the previous stack frame**: %esp = %ebp, %ebp = (%ebp)
8. **Jump back to return address**: %eip = 4(%esp)

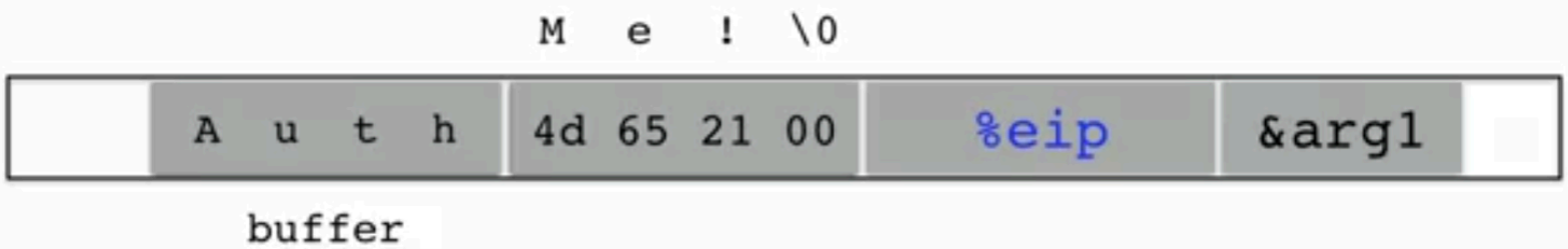
Buffer overflows

Benign outcome

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

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

Benign outcome

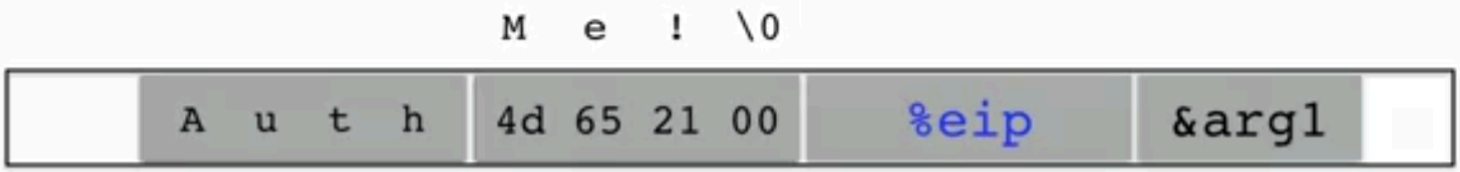
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

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



Benign outcome

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

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

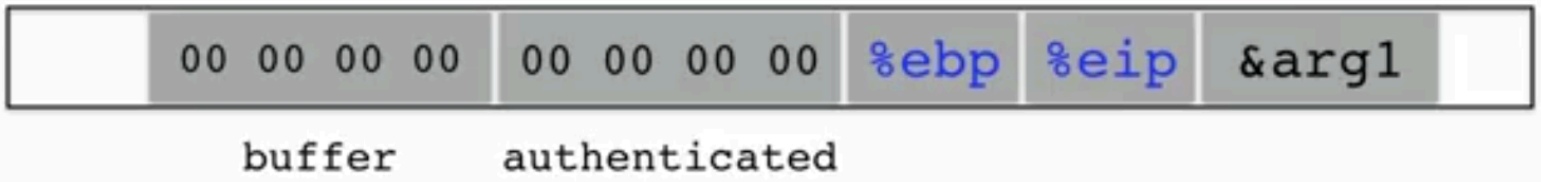
Upon return, sets %ebp to 0x0021654d



SEGFAULT (0x00216551) (during subsequent access)

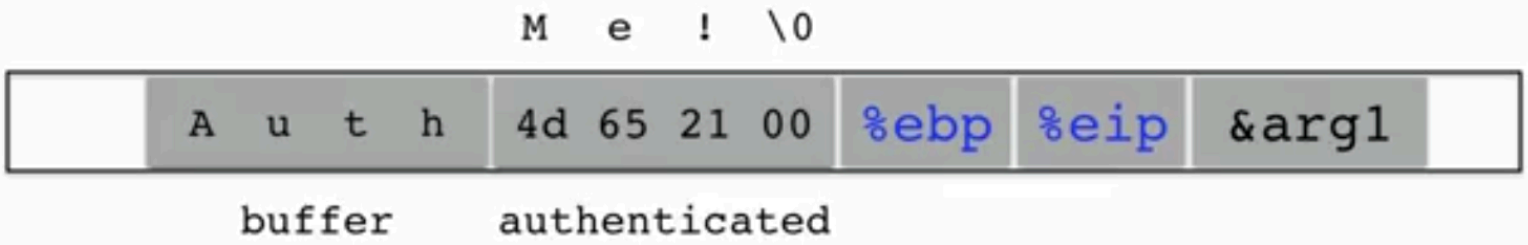
Security-relevant outcome

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

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



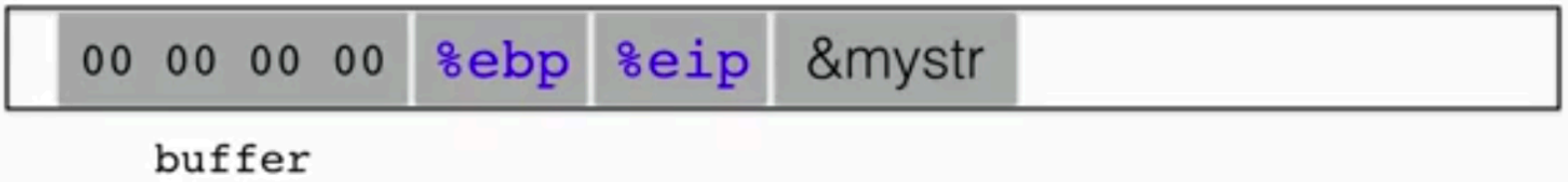
Security-relevant outcome

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

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



Could it be worse?

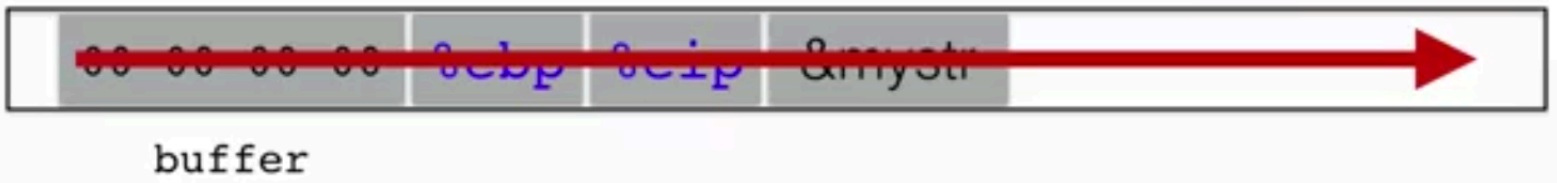
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```



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

Could it be worse?

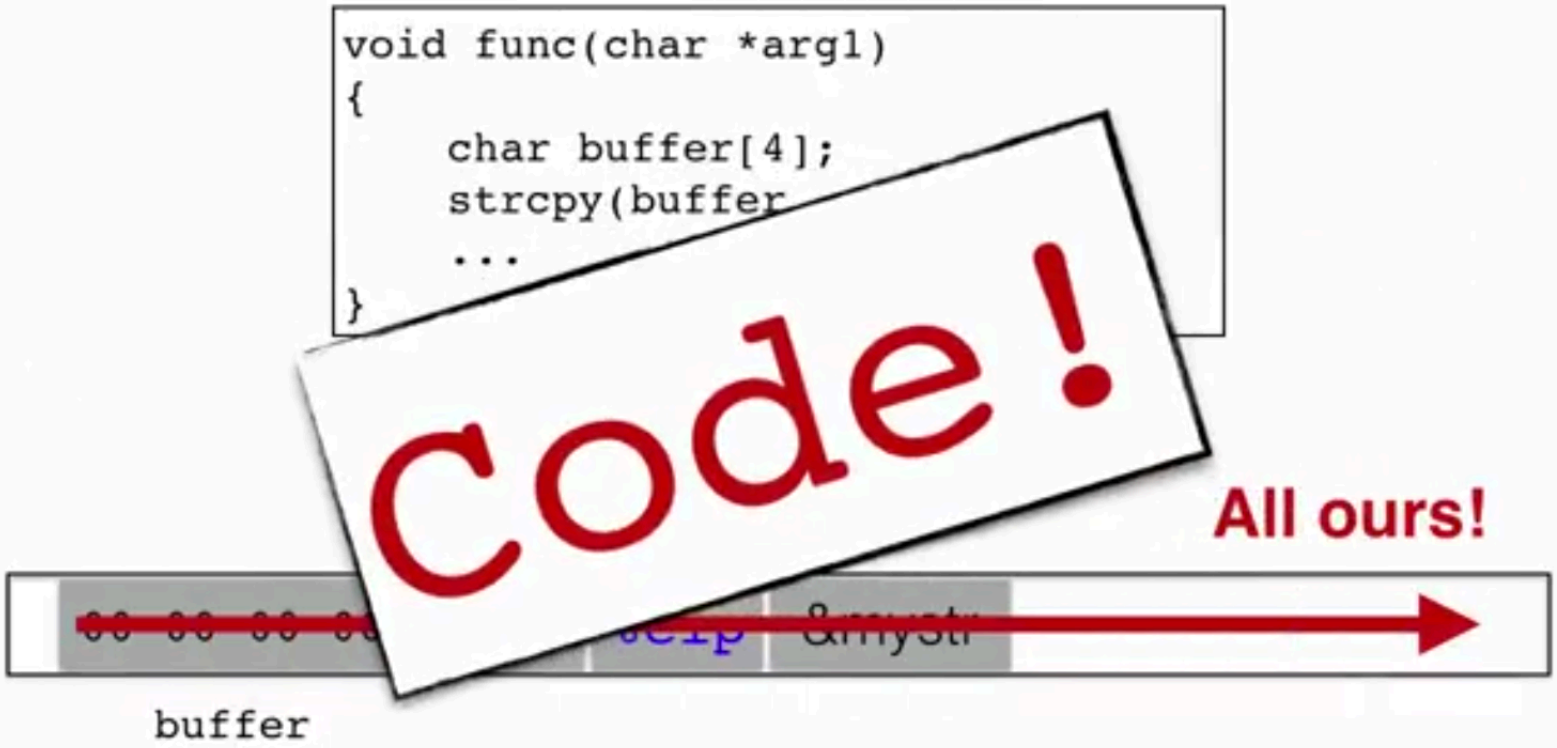
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```



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

What could you write to memory to wreak havoc?

Could it be worse?

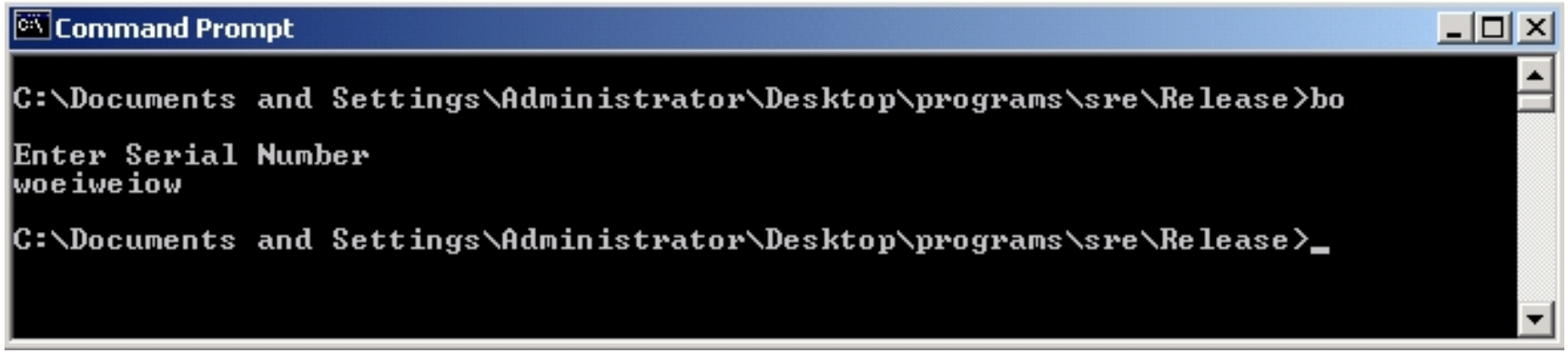


`strcpy` will let you write as much as you want (til a '\0')

What could you write to memory to wreak havoc?

Stack Smashing Example

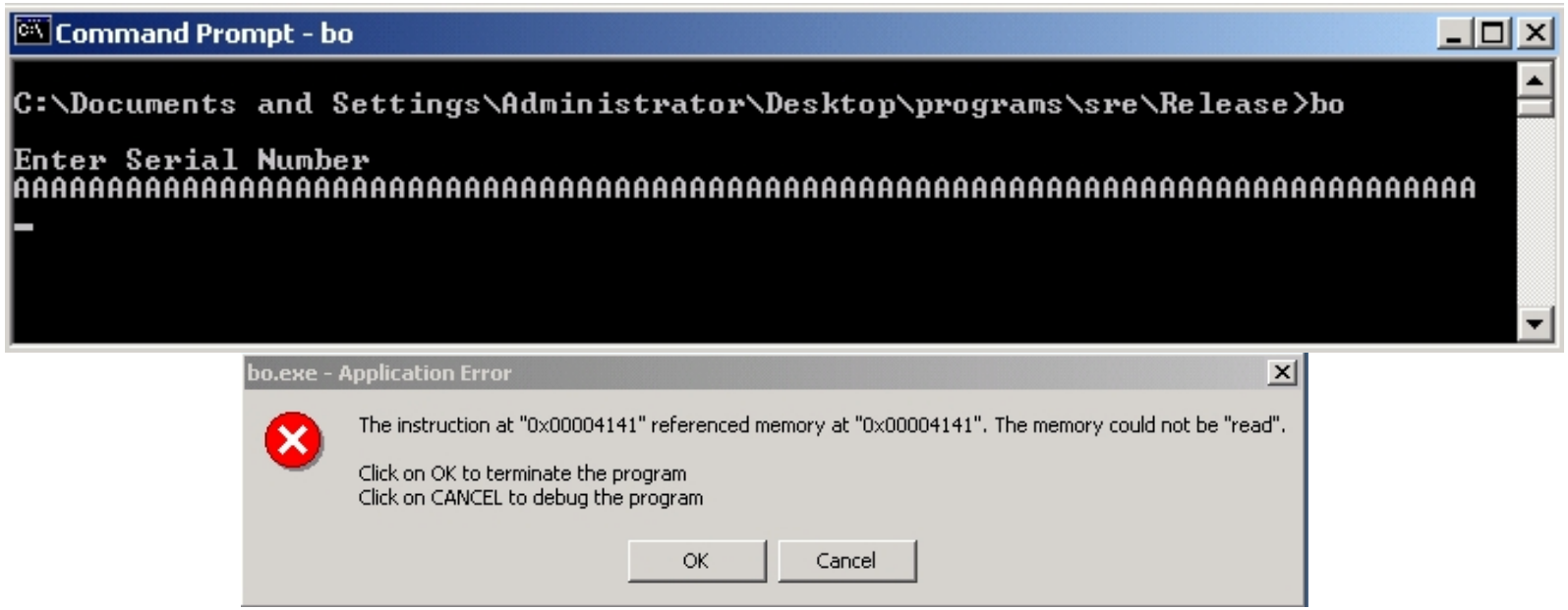
- Program asks for a serial number that the attacker does not know
- Attacker does **not** have source code
- Attacker does have the executable (exe)



- Program quits on incorrect serial number

Buffer Overflow Present?

- By trial and error, attacker discovers apparent buffer overflow



- Note that 0x41 is ASCII for "A"
- Looks like **ret** overwritten by 2 bytes!

Disassemble Code

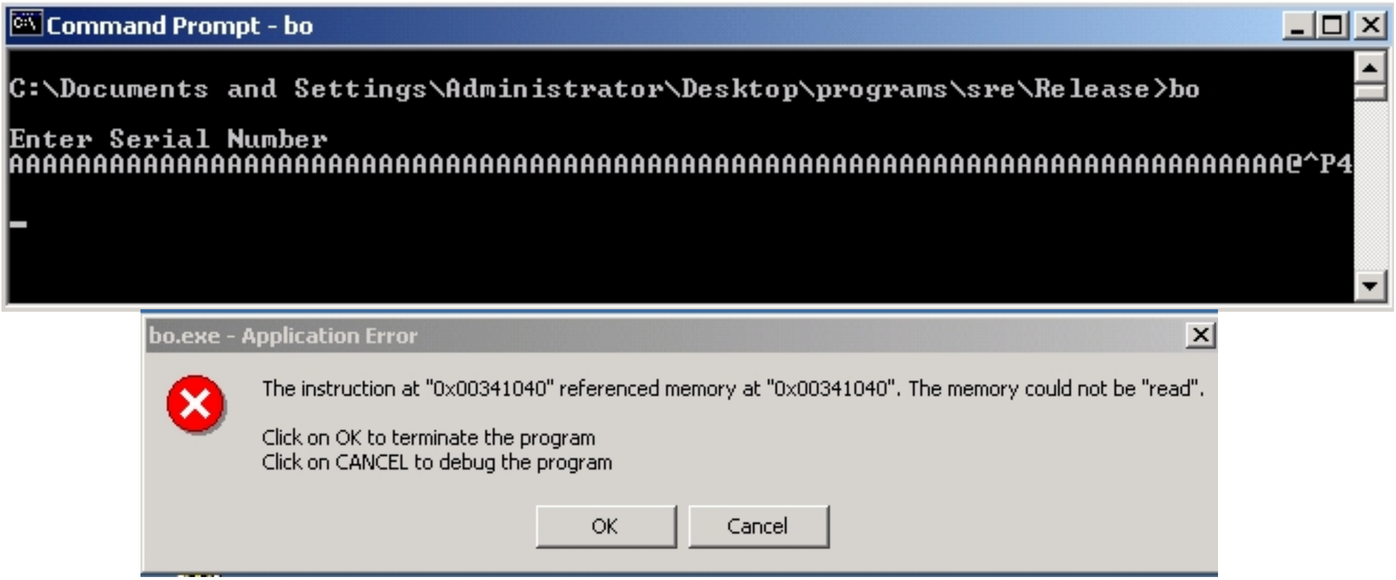
- Next, disassemble bo.exe to find

```
.text:00401000
.text:00401000                sub     esp, 1Ch
.text:00401003                push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008                call    sub_40109F
.text:0040100D                lea     eax, [esp+20h+var_1C]
.text:00401011                push    eax
.text:00401012                push    offset aS       ; "%s"
.text:00401017                call    sub_401088
.text:0040101C                push    8
.text:0040101E                lea     ecx, [esp+2Ch+var_1C]
.text:00401022                push    offset aS123n456 ; "S123N456"
.text:00401027                push    ecx
.text:00401028                call    sub_401050
.text:0040102D                add     esp, 18h
.text:00401030                test    eax, eax
.text:00401032                jnz     short loc_401041
.text:00401034                push    offset aSerialNumberIs ; "Serial number is correct.\n"
.text:00401039                call    sub_40109F
.text:0040103E                add     esp, 4
```

- The goal is to exploit buffer overflow to jump to address 0x401034

Buffer Overflow Attack

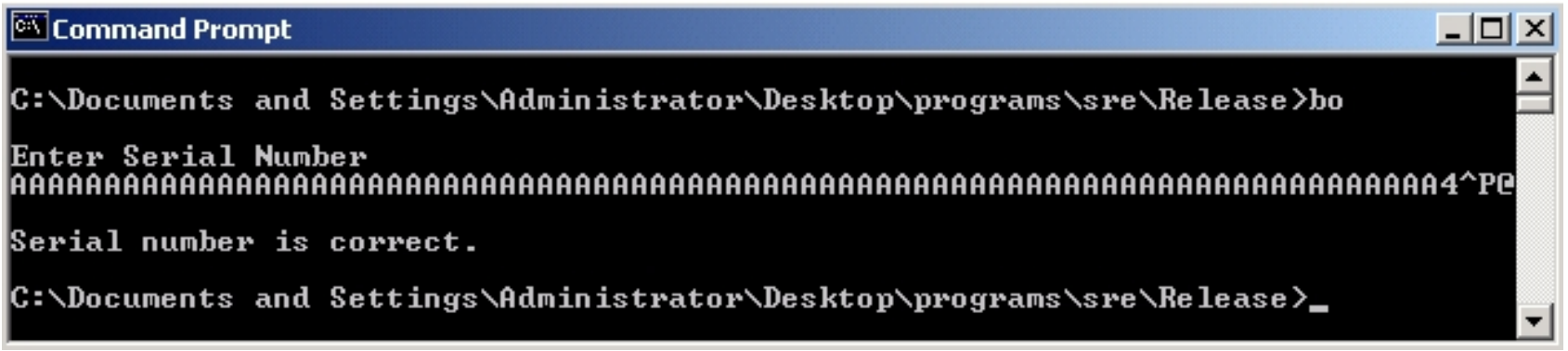
- Find that, in ASCII, 0x401034 is "@^P4"



- ❑ Byte order is reversed? Why?
- ❑ X86 processors are "little-endian"

Overflow Attack, Take 2

- Reverse the byte order to "4^P@" and…



- Success! We've bypassed serial number check by exploiting a buffer overflow
- What just happened?
  - Overwrote return address on the stack

Buffer Overflow

- Attacker did **not** require access to the source code
- Only tool used was a disassembler to determine address to jump to
- Find desired address by trial and error?
  - Necessary if attacker does not have exe
  - For example, a remote attack

Source Code

- Source code for buffer overflow example

- Flaw easily found by attacker…
- **…without access to source code!**

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

main()
{
    char in[75];

    printf("\nEnter Serial Number\n");

    scanf("%s", in);

    if(!strncmp(in, "S123N456", 8))
    {
       printf("Serial number is correct.\n");
    }
}
```