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**

C and C++ still very popular

Language Rank		Types	Spectrum Ranking
1.	Java		100.0
2.	C		99.2
3.	C++		95.5
4.	Python		93.4
5.	C#		92.2
6.	PHP		84.6
7.	Javascript		84.3
8.	Ruby		78.6
9.	R		74.0
10.	MATLAB		72.6

http://spectrum.ieee.org/static/interactive-the-top-programming-languages

Critical systems in C/C++

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

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

History of buffer overflows

The harm has been substantial

1988 1999 2000 **[2001]** 2002 2003

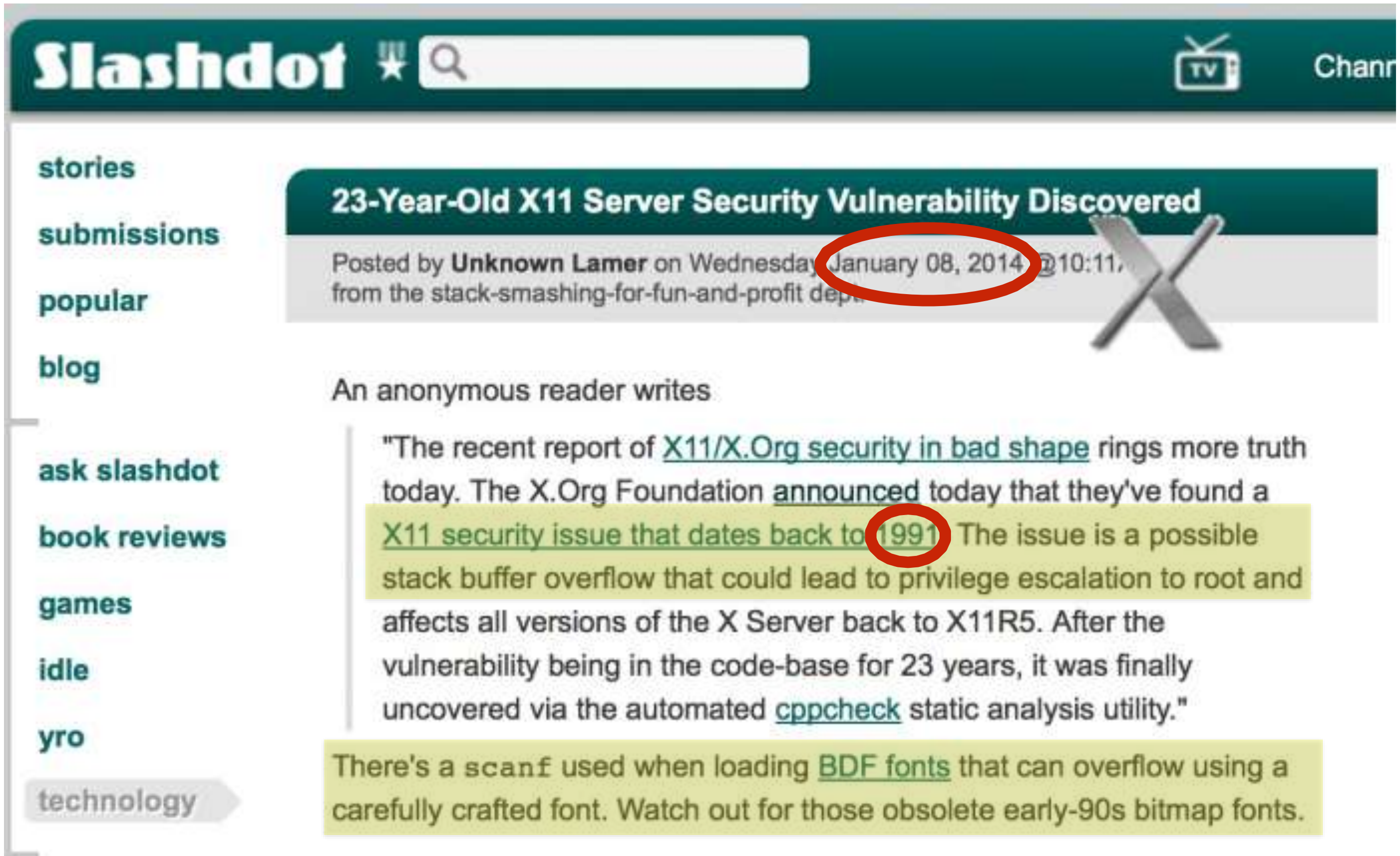
- **CodeRed**
  - Exploited an overflow in the MS-IIS server
  - 300,000 machines infected in 14 hours

History of buffer overflows

The harm has been substantial

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

Chann

stories

submissions

popular

blog

ask slashdot

book reviews

games

idle

yro

technology

23-Year-Old X11 Server Security Vulnerability Discovered

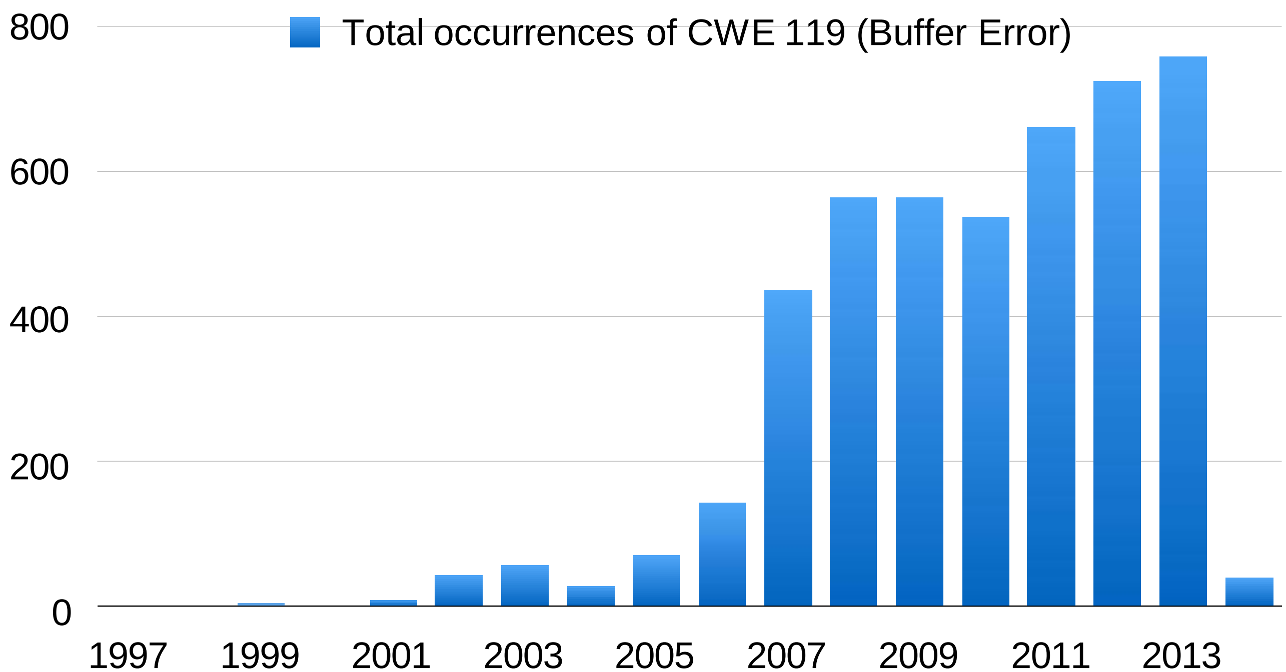
Posted by **Unknown Lamer** on Wednesday January 08, 2014 @10:11
from the stack-smashing-for-fun-and-profit dept.

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

http://cwe.mitre.org/top25/

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)

What we'll do

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

Analyzing security requires a whole-systems view

Note about terminology

- I use the term **buffer overflow** to mean ***any access of a buffer outside of its allotted bounds***
  - Could be an over-*read*, or an over-*write*
  - Could be during *iteration* ("running off the end") or by *direct access* (e.g., by pointer arithmetic)
  - Out-of-bounds access could be to addresses that *precede* or *follow* the buffer
- **Others sometimes use different terms**
  - They might reserve buffer overflow to refer only to actions that write beyond the bounds of a buffer
    - Contrast with terms *buffer underflow* (write prior to the start), *buffer overread* (read past the end), *out-of-bounds access*, etc.

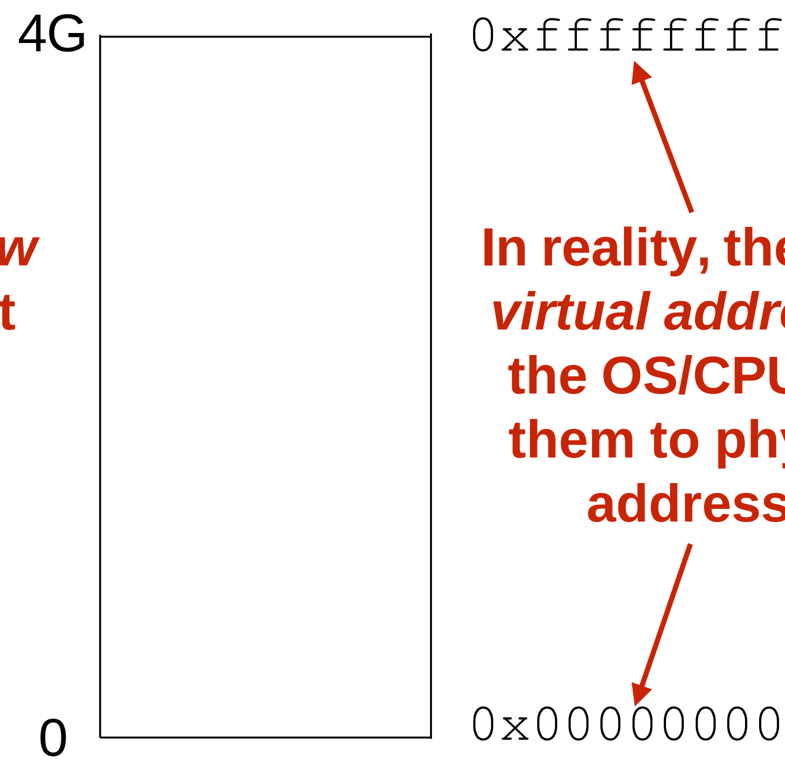
Memory layout

Memory Layout Refresher

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

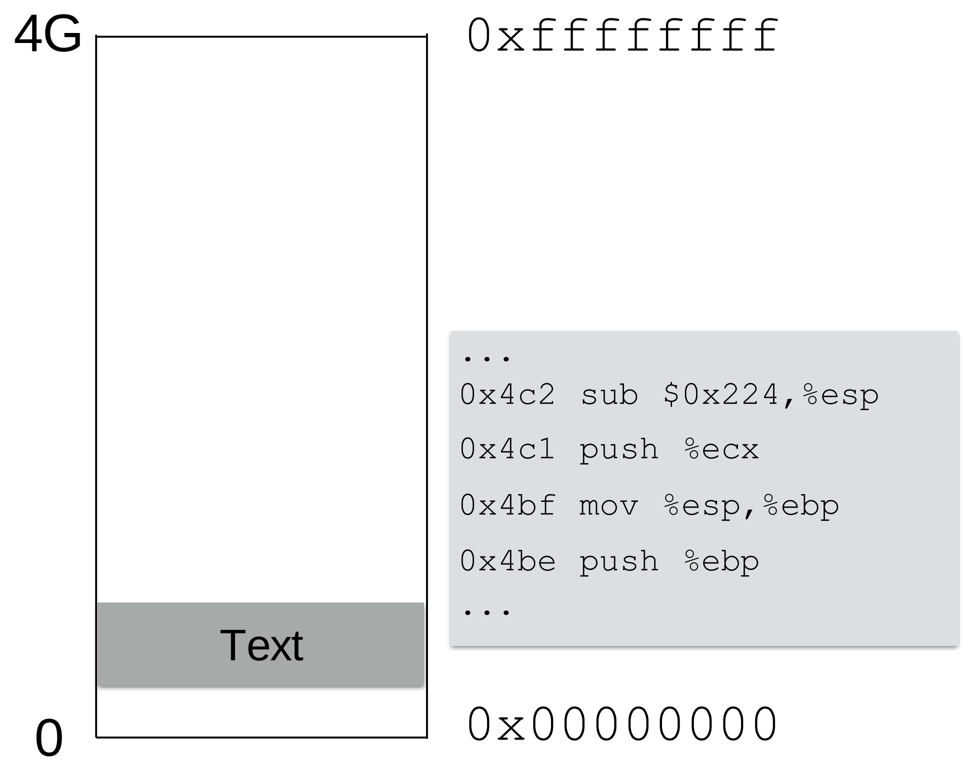
All programs are stored in memory

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

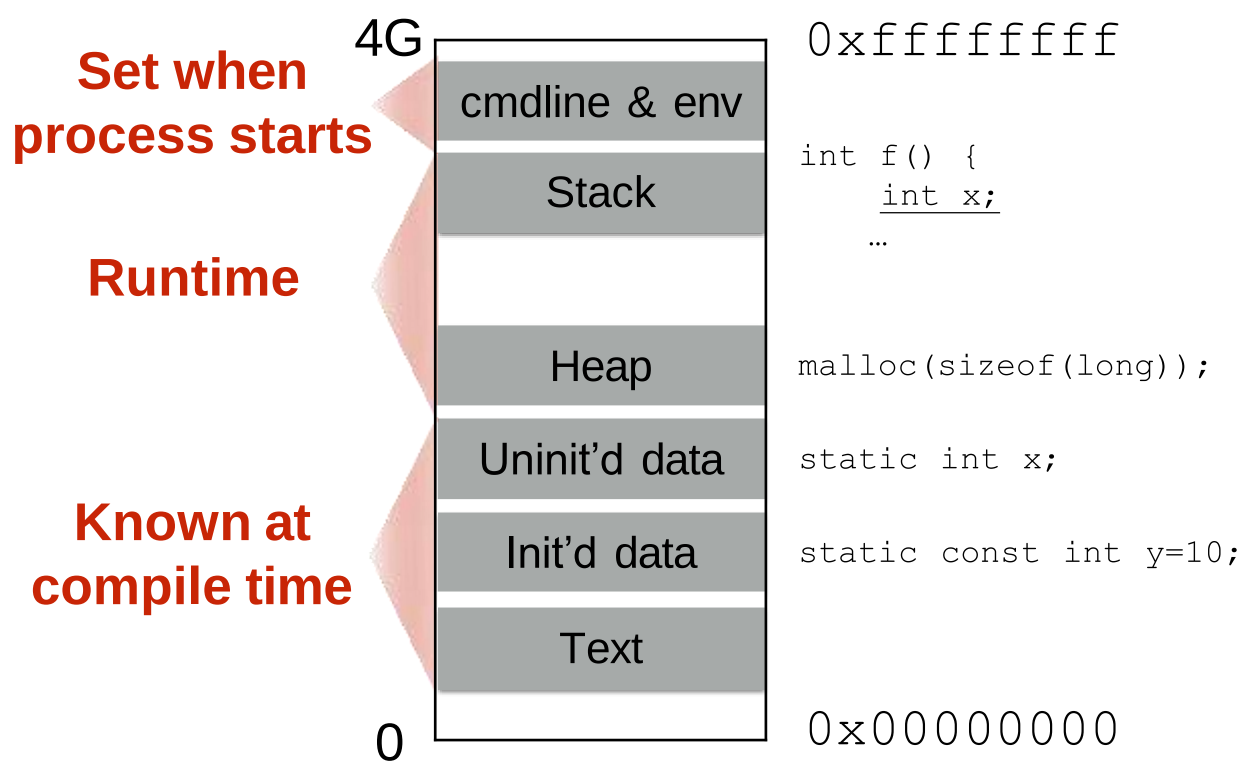


In reality, these are *virtual addresses*; the OS/CPU map them to physical addresses

The instructions themselves are in memory



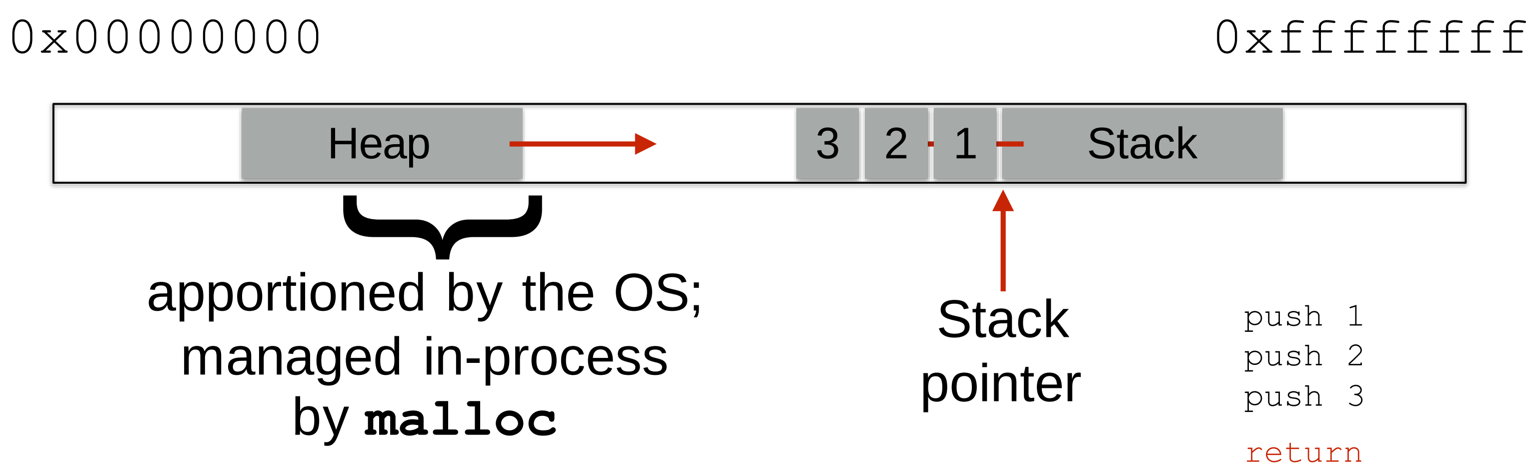
Location of data areas



Memory allocation

Stack and heap grow in opposite directions

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



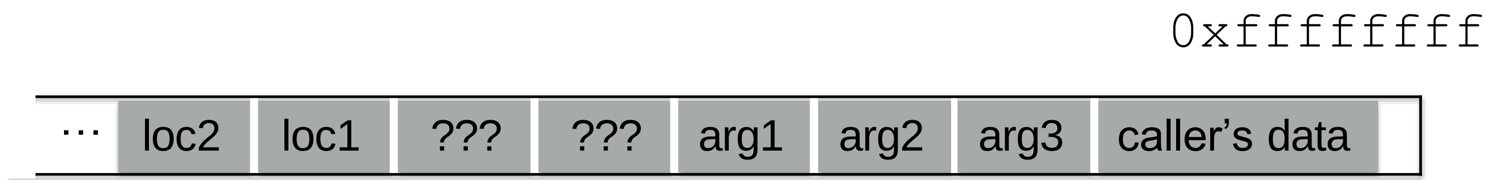
Focusing on the stack for now

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;
    ...
}
```



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

Arguments pushed in reverse order of code

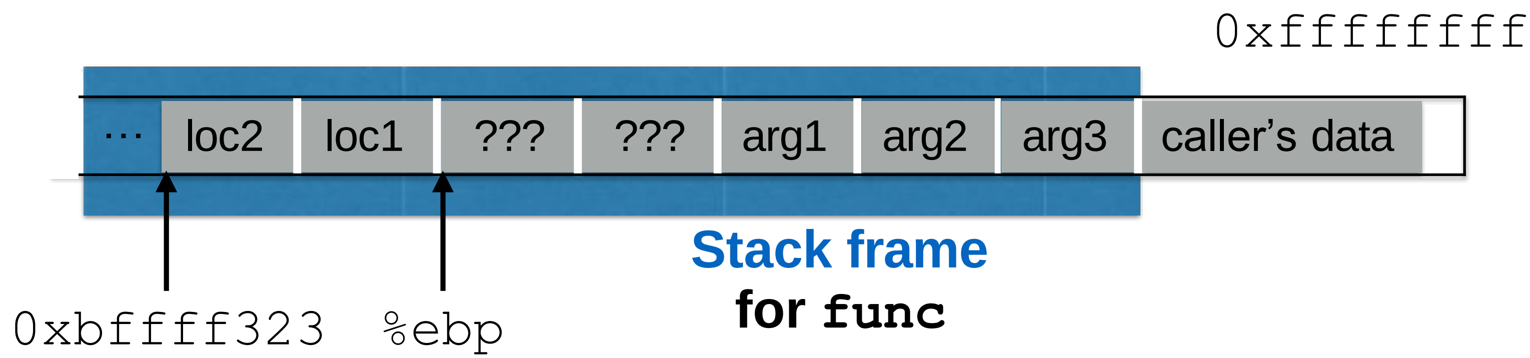
The local variable allocation is ultimately up to the compiler: Variables could be allocated in any order, or not allocated at all and stored only in registers, depending on the optimization level used.

Accessing variables

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

Q: Where is (this) `loc2`?

A: -8(%ebp)



Frame pointer

Can't know absolute address at compile time

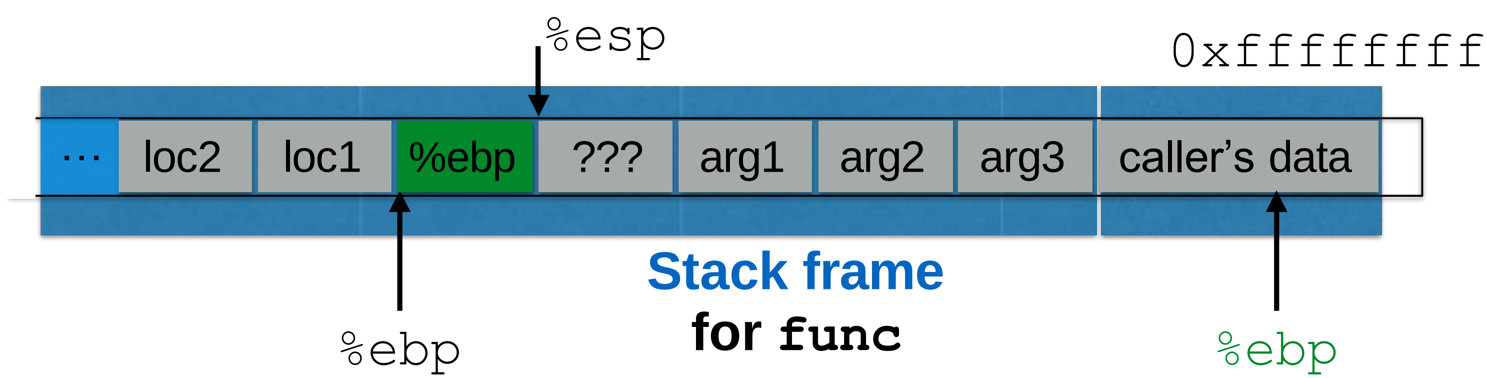
But can know the **relative** address

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

Returning from functions

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

Q: How do we restore %ebp?



Push `%ebp` before `locals`

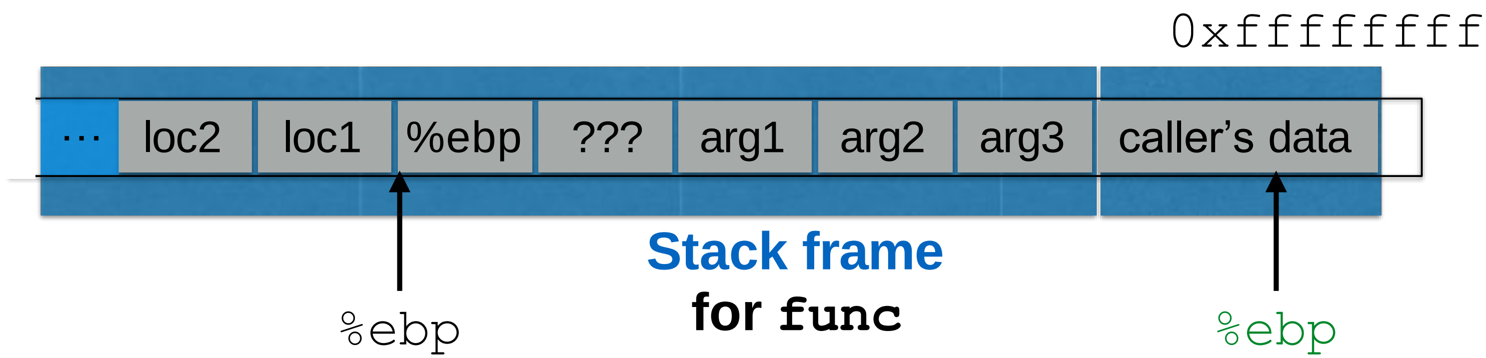
Set %ebp to current (%esp)

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

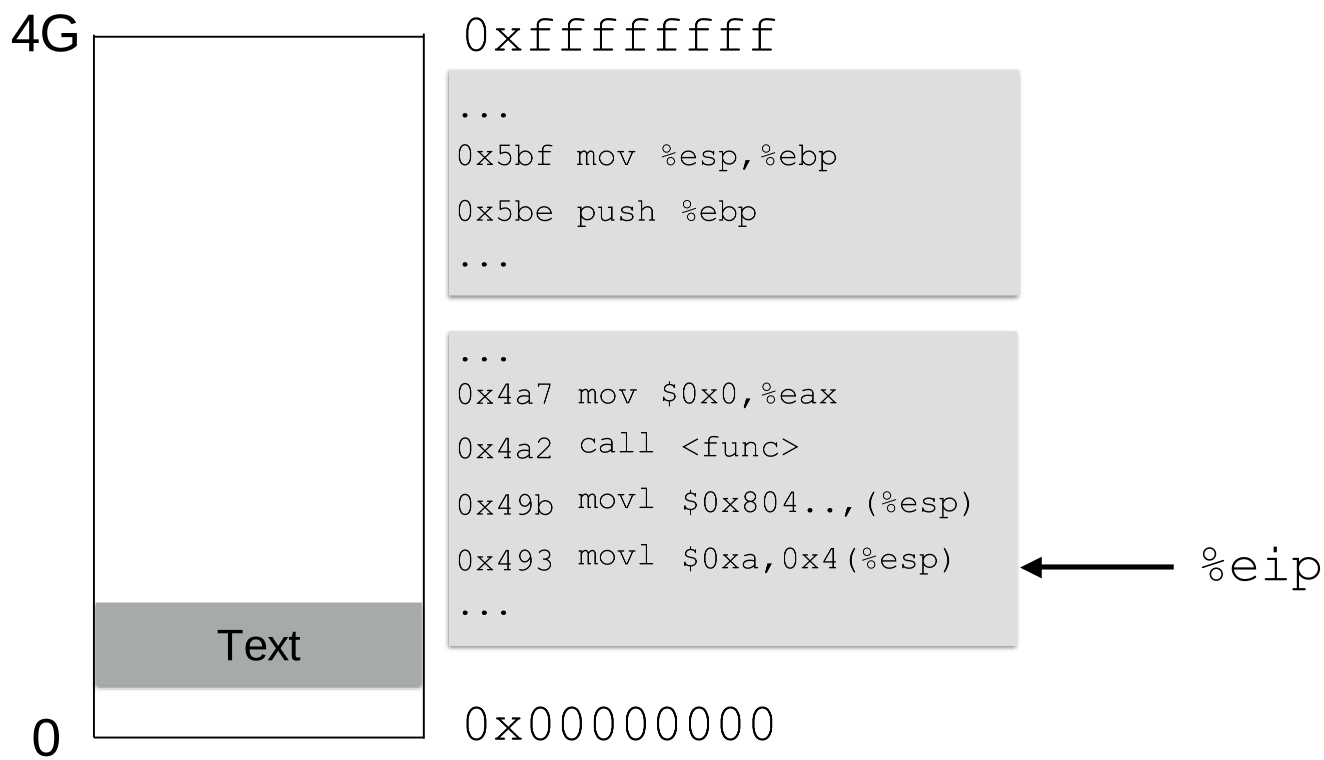
Returning from functions

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

Q: How do we resume here?



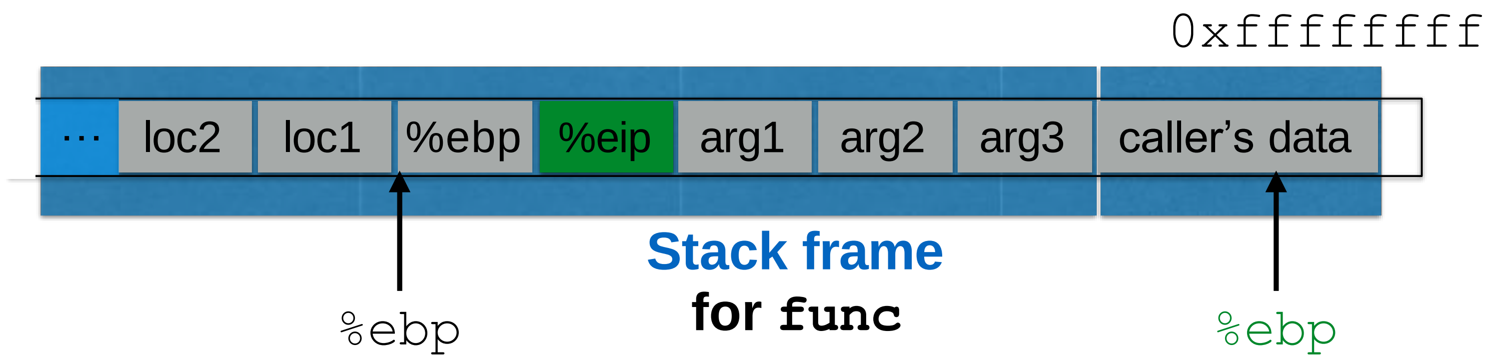
Instructions in memory



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** (%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

Buffer overflows from 10,000 ft

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

Benign outcome

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

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

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

M e ! \0

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

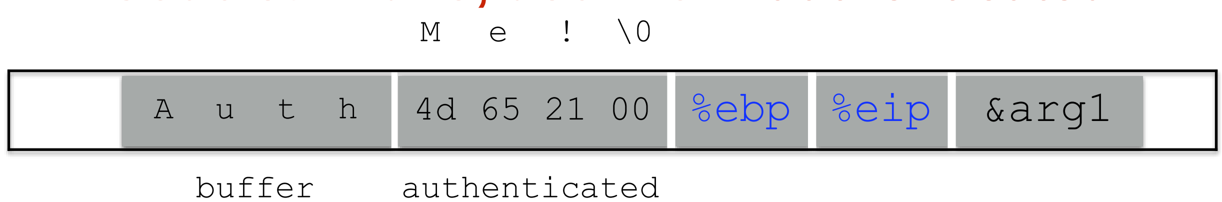
buffer

SEGFAULT (0x00216551) (during subsequent access)

Security-relevant outcome

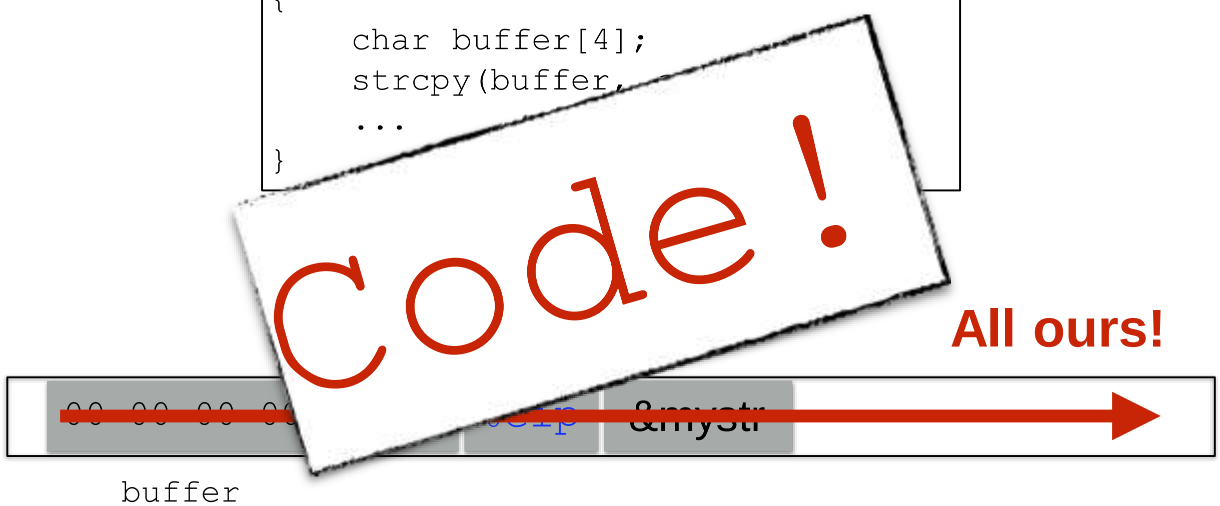
```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

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

Code still runs; user now 'authenticated'



Could it be worse?

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



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

What could you write to memory to wreak havoc?

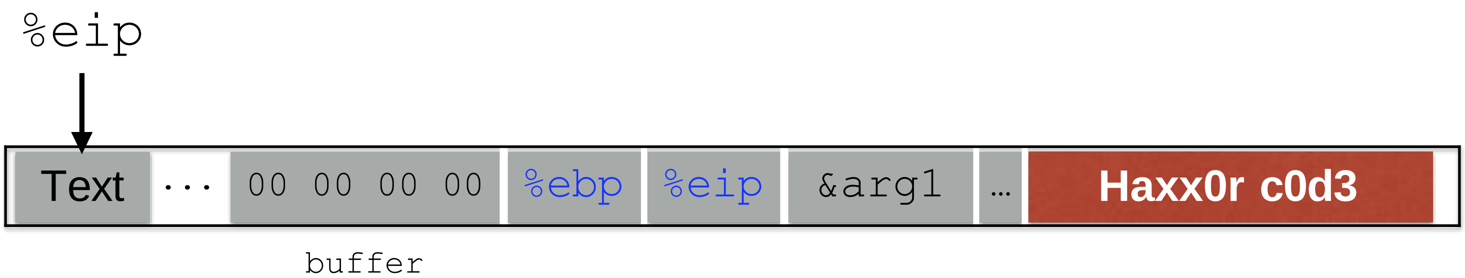
Aside: User-supplied strings

- These examples provide their own strings
- In reality **strings** come **from *users*** in myriad aways
  - **Text** input
  - **Packets**
  - **Environment variables**
  - **File** input…
- **Validating assumptions** about **user input** is extremely **important**
  - We will discuss it later, and throughout the course

Code injection

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 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 use the loader** (we're injecting)

What code to run?

- Goal: **general-purpose shell**
  - Command-line prompt that gives attacker **general access to the system**
- The code to launch a shell is called **shellcode**

Shellcode

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

Assembly

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

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

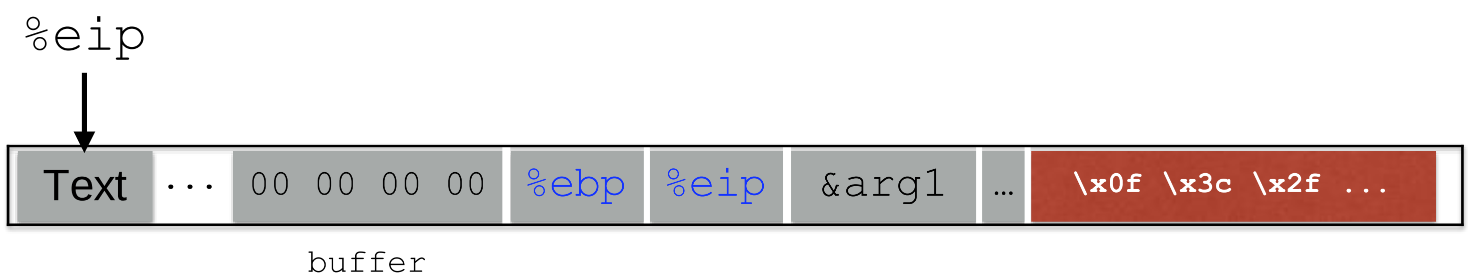
Machine code

(Part of) your input

Challenge 2

Getting injected code to run

- We can't insert a "jump into my code" instruction
- We don't know precisely where our code is



Recall

Memory layout 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**

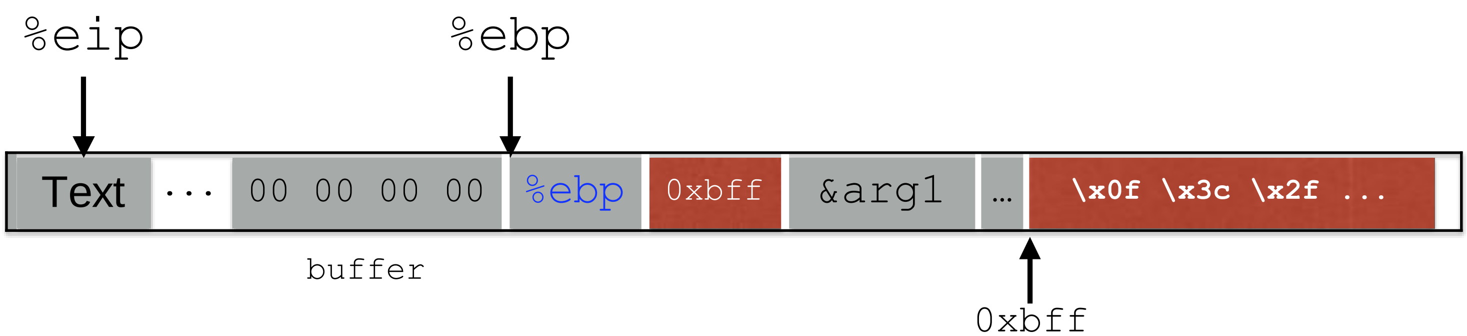
Called function:

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)

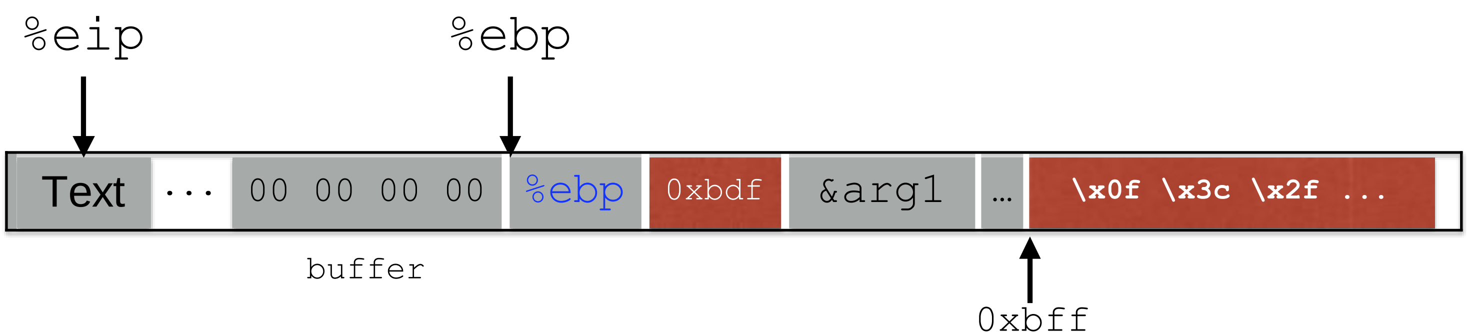
Hijacking the saved `%eip`



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

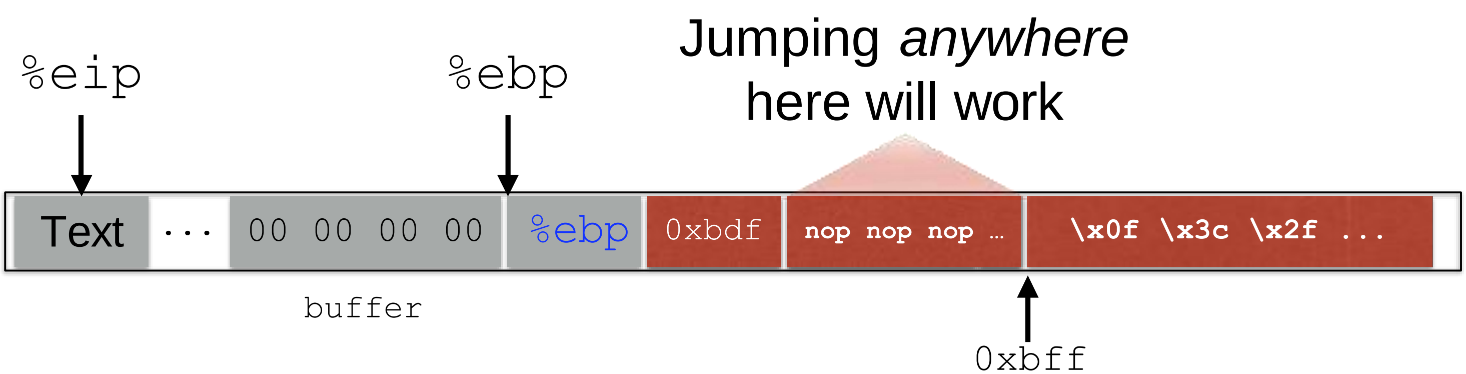
- If we don’t have access to the code, we don’t know how far the buffer is from the saved `%ebp`

- One approach: 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

- Without address randomization (discussed later):
  - The **stack always starts** from the same **fixed address**
  - The stack will grow, but usually it **doesn’t grow very deeply** (unless the code is heavily recursive)

Improving our chances: `nop` sleds

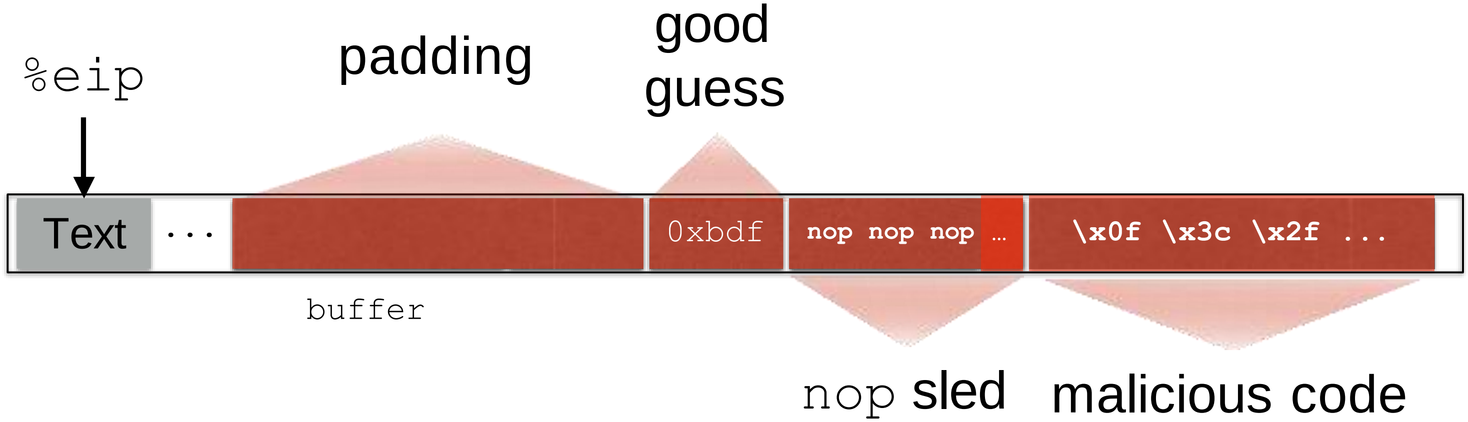
`nop` is a single-byte instruction
(just moves to the next instruction)



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

Putting it all together

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



Other memory exploits

Other attacks

- The code injection attack we have just considered is called **stack smashing**
  - The term was coined by *Aleph One* in 1996
- Constitutes an **integrity violation**, and arguably a **violation of availability**
- Other attacks exploit bugs with buffers, too

Heap overflow

- Stack smashing overflows a stack allocated buffer
- You can also **overflow a buffer** allocated by `malloc`, which resides on the **heap**

Heap overflow

```
typedef struct _vulnerable_struct {
  char buff[MAX_LEN];
  int (*cmp)(char*,char*);
} vulnerable;

int foo(vulnerable* s, char* one, char* two)
{
 strcpy( s->buff, one );
 strcat( s->buff, two );
 return s->cmp( s->buff, "file://foobar" );
}
```

copy `one` *into* `buff`

copy `two` *into* `buff`

must have `strlen(one)+strlen(two) < MAX_LEN`

or we overwrite **`s->cmp`**

Heap overflow variants

- **Overflow into the C++ object *vtable***
  - C++ objects (that contain virtual functions) are represented using a *vtable*, which contains pointers to the object's methods
  - This table is analogous to `s->cmp` in our previous example, and a similar sort of attack will work
- **Overflow into adjacent objects**
  - Where buff is not collocated with a function pointer, but is allocated near one on the heap
- **Overflow heap metadata**
  - Hidden header just before the pointer returned by malloc
  - Flow into that header to corrupt the heap itself
    - Malloc implementation to do your dirty work for you!

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);
  }
```

HUGE

Wrap-around

Overflow

- If we set `nresp` to 1073741824 and `sizeof(char*)` is 4
- then `nresp*sizeof(char*)` overflows to become 0
- subsequent writes to allocated `response` overflow it

Corrupting data

- The attacks we have shown so far **affect code**
  - *Return addresses* and *function pointers*

- But attackers can **overflow data** as well, to
  - **Modify a secret key** to be one known to the attacker, to be able to decrypt future intercepted messages
  - **Modify state variables** to bypass authorization checks (earlier example with `authenticated` flag)
  - **Modify interpreted strings** used as part of commands
    - E.g., to facilitate SQL injection, discussed later in the course

Read overflow

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

Read overflow

```
int main() {
  char buf[100], *p;
  int i, len;
  while (1) {
   p = fgets(buf,sizeof(buf),stdin);
   if (p == NULL) return 0;
   len = atoi(p);
   p = fgets(buf,sizeof(buf),stdin);
   if (p == NULL) return 0;
   for (i=0; i<len; i++)
    if (!iscntrl(buf[i])) putchar(buf[i]);
    else putchar('.');
   printf("\n");
 }}
```

Read integer

Read message

Echo back (partial) message

May exceed actual message length!

Sample transcript

```
% ./echo-server
24
every good boy does fine
ECHO: |every good boy does fine|
10
hello there
ECHO: |hello ther|
25
hello
ECHO: |hello..here..y does fine.|
```

OK: input length < buffer size

BAD: length > size !

leaked data

Heartbleed

- The **Heartbleed bug** was a read overflow in exactly this style
- The SSL server should accept a “heartbeat” message that it echoes back
- The heartbeat message specifies the length of its echo-back portion, but the buggy SSL **software did not check the length was accurate**
- Thus, an attacker could request a longer length, and **read past the contents of the buffer**
  - Leaking passwords, crypto keys, …

Stale memory

- A **dangling pointer bug** occurs when a pointer is freed, but the program continues to use it
- An attacker can **arrange for the freed memory to be reallocated** and under his control
  - When the dangling pointer is dereferenced, it will access attacker-controlled data

```
struct foo { int (*cmp)(char*,char*); };

struct foo *p = malloc(…);
free(p);
...
q = malloc(…) //reuses memory
*q = 0xdeadbeef; //attacker control
...
p->cmp(“hello”,”hello”); //dangling ptr
```

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 f… et Explorer (IE) flaw that led to cyberattacks on Google and its corporat… s.

The zero-day attack that exploited IE is part … are that is keeping researchers very busy.

"We're discovering th… te basis, and we've seen about a dozen files dropped … Alperovitch, vice president of research at McAfee Labs, told Tech…

Dangling pointer dereference!

The attack… 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.

Format string vulnerabilities

Formatted I/O

- C’s printf family supports formatted I/O

```
void print_record(int age, char *name)
{
  printf(“Name: %s\tAge: %d\n”,name,age);
}
```

- Format specifiers
  - Position in string indicates stack argument to print
  - Kind of specifier indicates type of the argument
    - %s = string
    - %d = integer
    - etc.

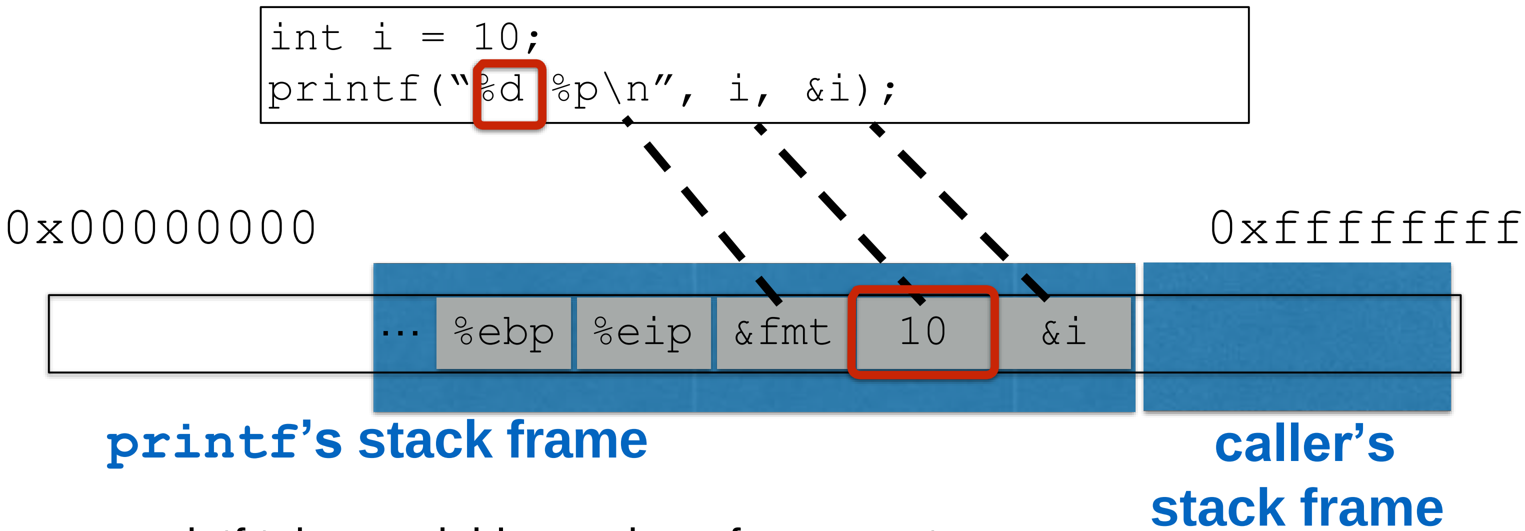
What’s the difference?

```
void safe()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(“%s”,buf);
}
```

```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

Attacker controls the format string

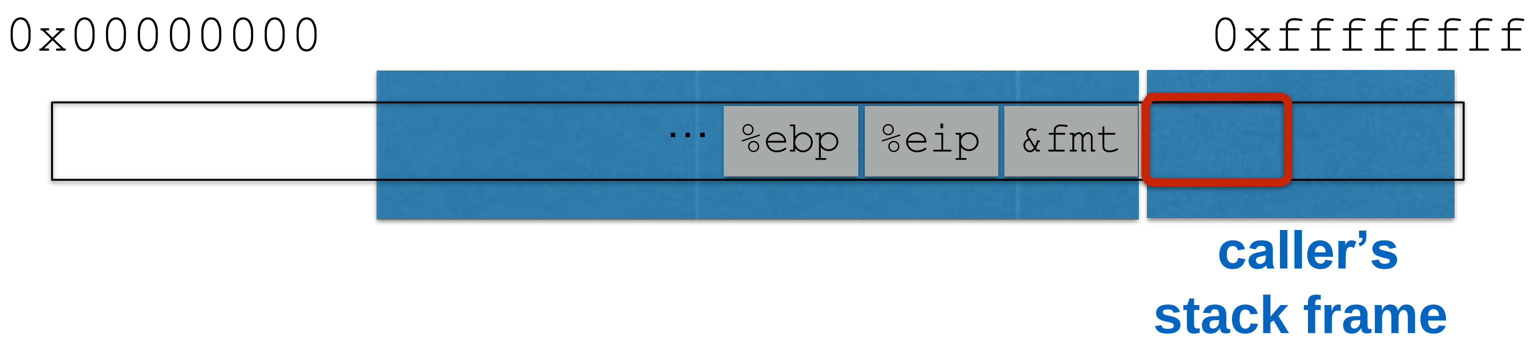
`printf` implementation



- printf takes variable number of arguments
- printf pays no mind to where the stack frame "ends"
- It presumes that you called it with (at least) as many arguments as specified in the format string

```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

“%d %x"



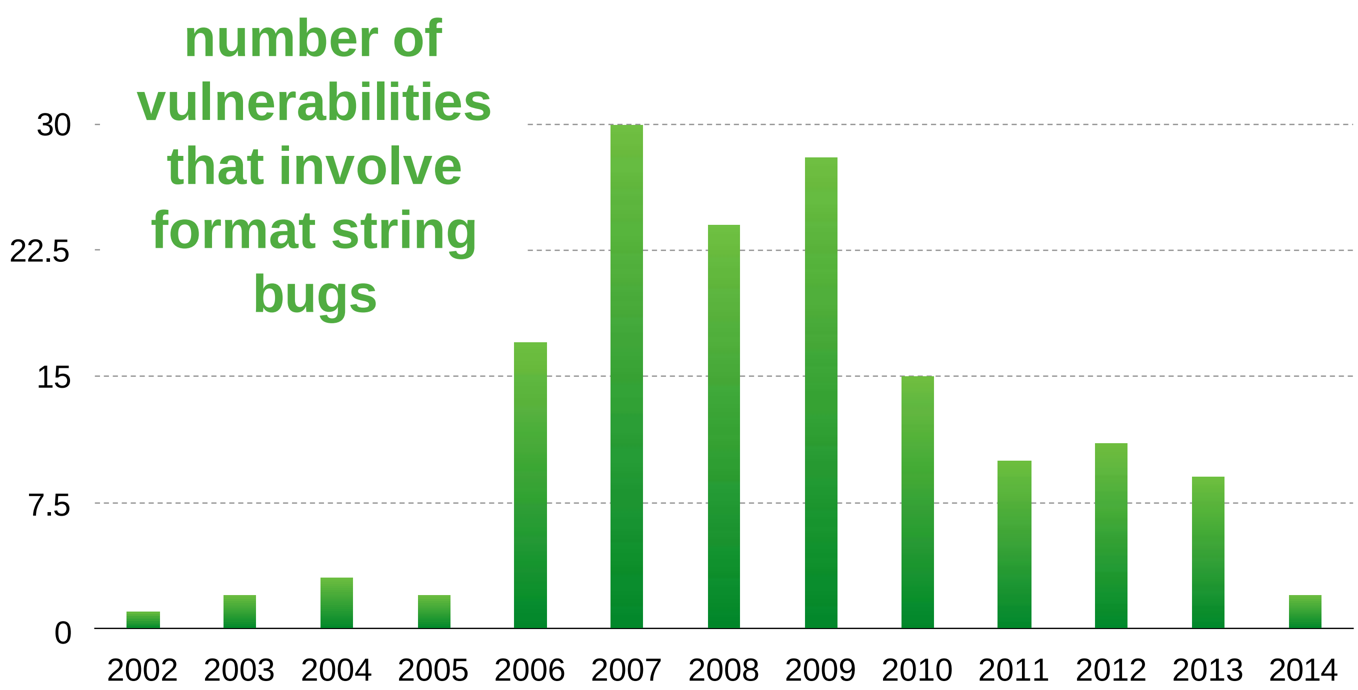
Format string vulnerabilities

- `printf("100% dave");`
  - Prints stack entry 4 byes above saved %eip
- `printf("%s");`
  - Prints bytes *pointed to* by that stack entry
- `printf("%d %d %d %d …");`
  - Prints a series of stack entries as integers
- `printf("%08x %08x %08x %08x …");`
  - Same, but nicely formatted hex
- `printf("100% no way!")`
  - ***WRITES*** the number 3 to address pointed to by stack entry

Why is this a buffer overflow?

- We should think of this as a buffer overflow in the sense that
  - The stack itself can be viewed as a kind of buffer
  - The size of that buffer is determined by the number and size of the arguments passed to a function
- Providing a bogus format string thus induces the program to overflow that “buffer”

Vulnerability prevalence



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

Time to switch hats

We have seen many styles of attack

What can be done to defend against them?