
بسم الله الرحمن الرحيم

Fundamentals of Secure Computing

Module II: Software Security

Ali Shakiba

Vali-e-Asr University of Rafsanjan

ali.shakiba@vru.ac.ir

Fall 2017

Low-level security: Attacks and exploits

Required Readings

- Common vulnerabilities guide for C programmers
  - Take note of the unsafe C library functions listed here, and how they are the source of buffer overflow vulnerabilities.
  - https://security.web.cern.ch/security/recommendations/en/codetools/c.shtml
- Memory layout
  - Explains a C program's memory layout
  - http://www.geeksforgeeks.org/memory-layout-of-c-program/

Low level security or **C** & the infamous **buffer overflow**

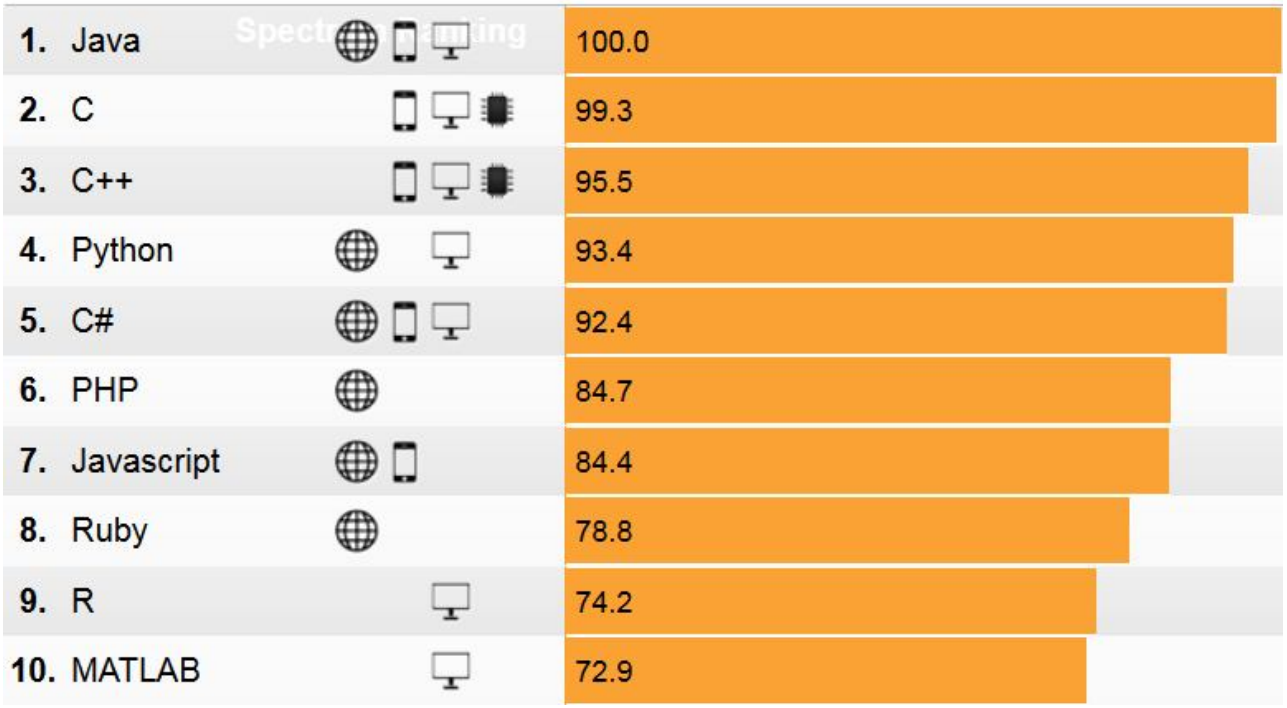
What is buffer overflow?

- 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 & C++ are still very popular!



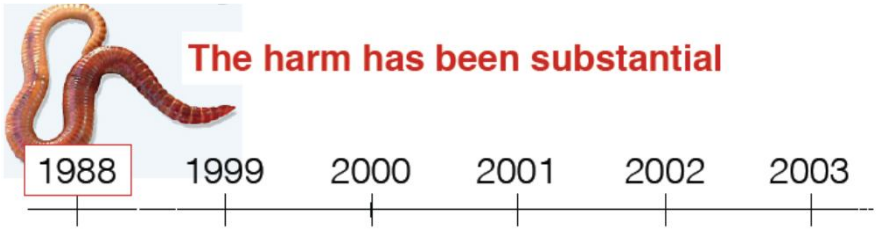
https://spectrum.ieee.org/static/interactive-the-top-programming-languages

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

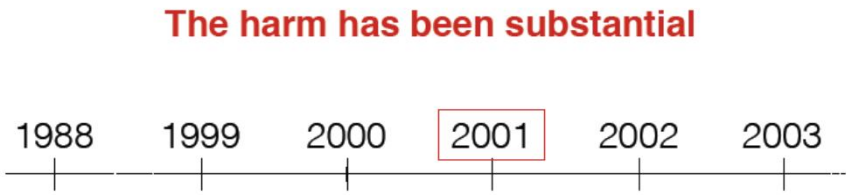


Morris worm

- 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

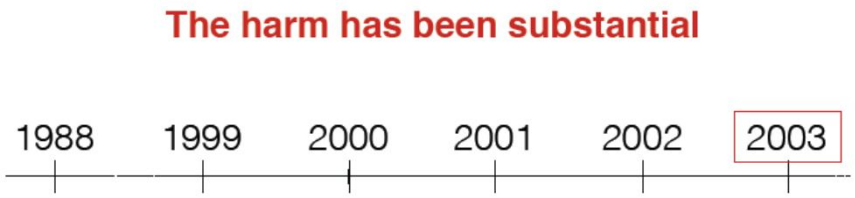
History of buffer overflows



CodeRed

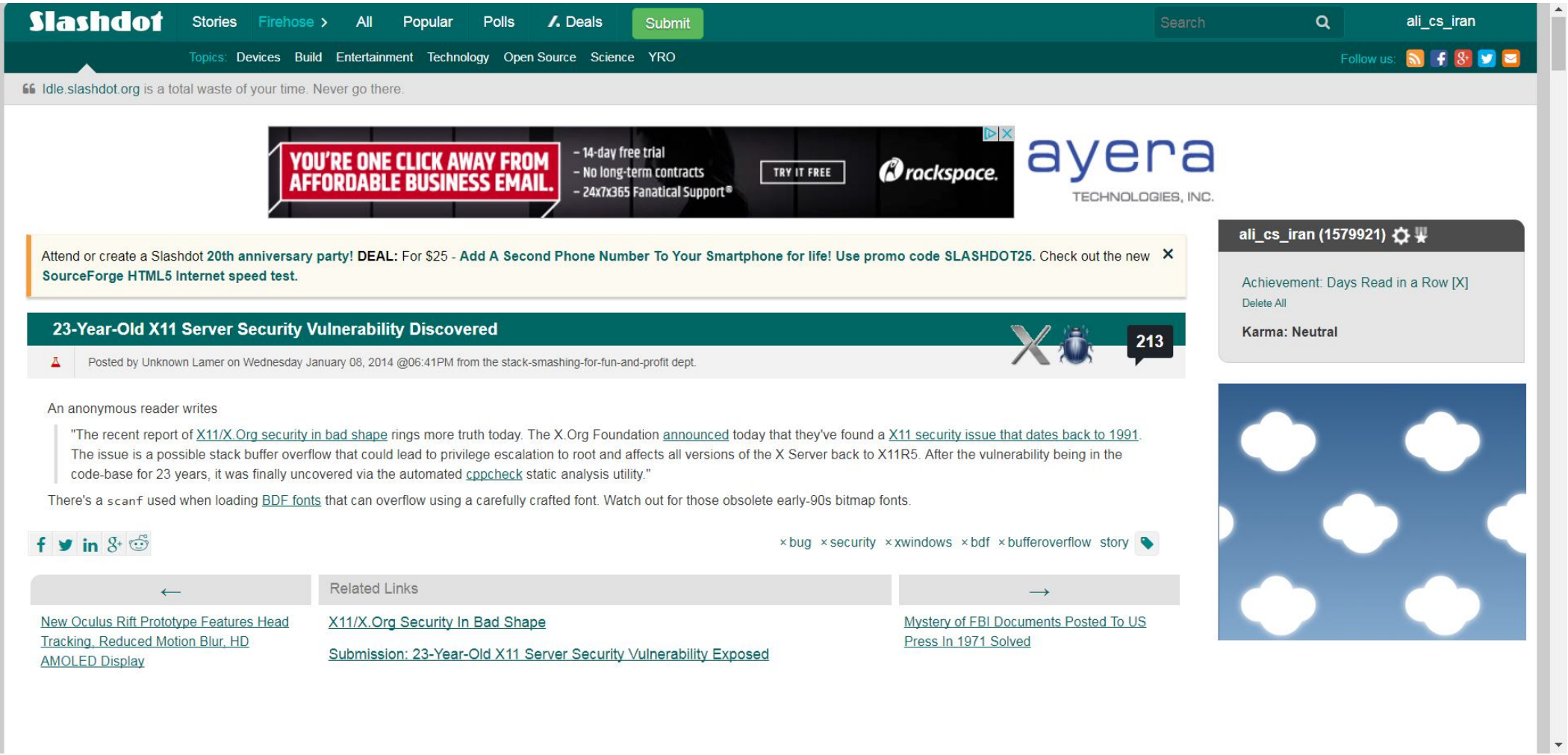
- Exploited an overflow in the MS-IIS server
- 300,000 machines infected in 14 hours

History of buffer overflows

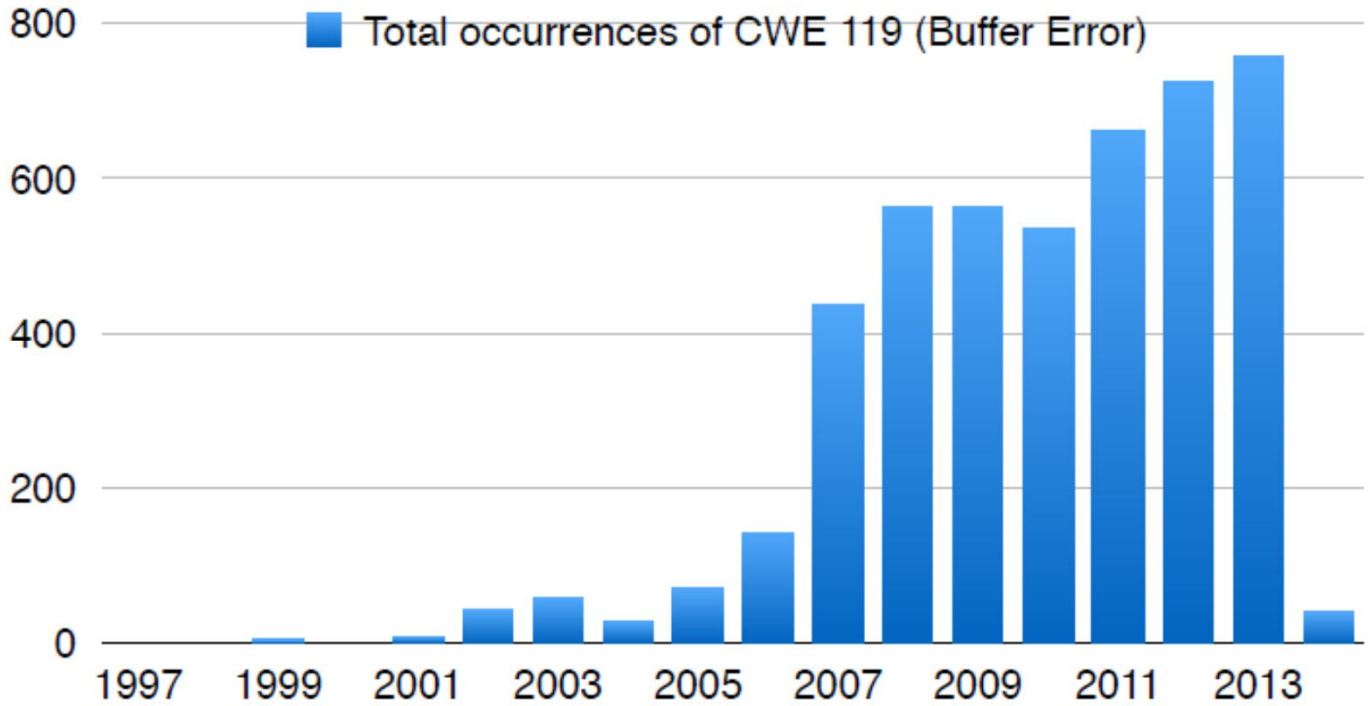


SQL Slammer

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



https://tech.slashdot.org/story/14/01/08/1421235/23-year-old-x11-server-security-vulnerability-discovered



https://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)
[13]	69.3	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]	68.5	CWE-494	Download of Code Without Integrity Check

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

What we'll do?

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

Analyzing security requires a whole-systems view.

Note about terminology

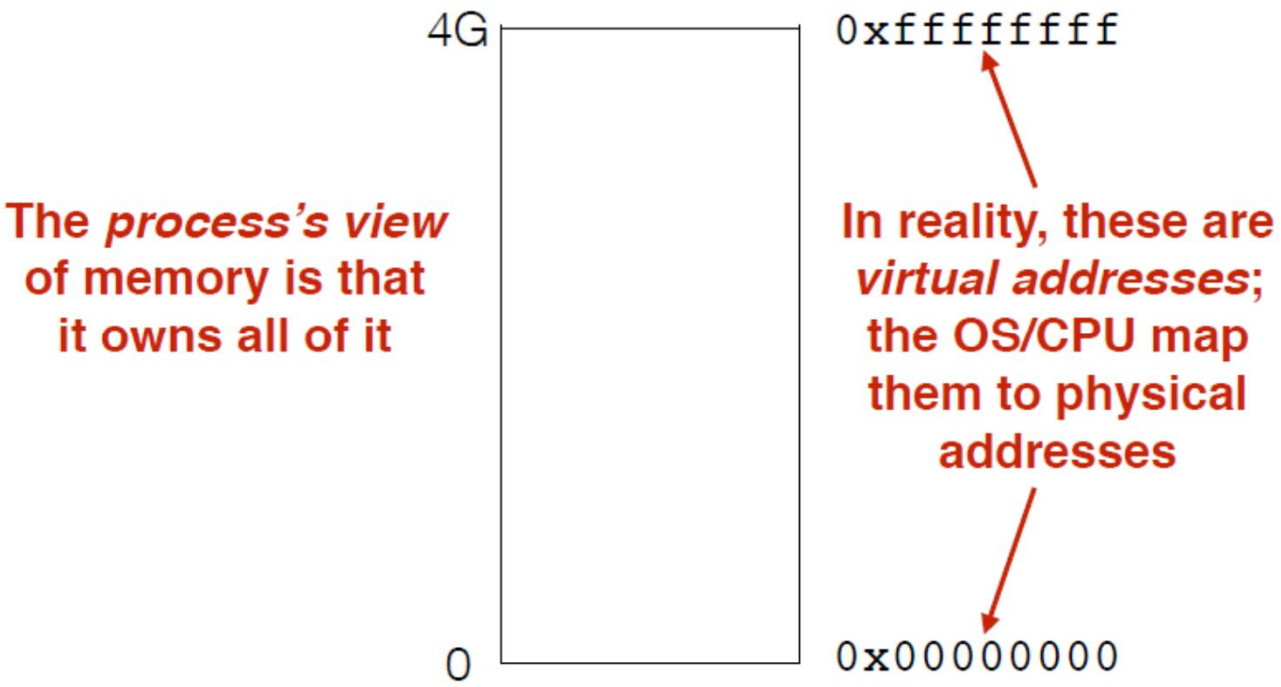
- We use the term **buffer overflow** to mean ***any access of a buffer outside of its allotted bounds***
  - Could be an over-*read*, or an over-*write*
  - Could be during *iteration* ("running off the end") or by *direct access* (e.g., by pointer arithmetic)
  - Out-of-bounds access could be to addresses 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

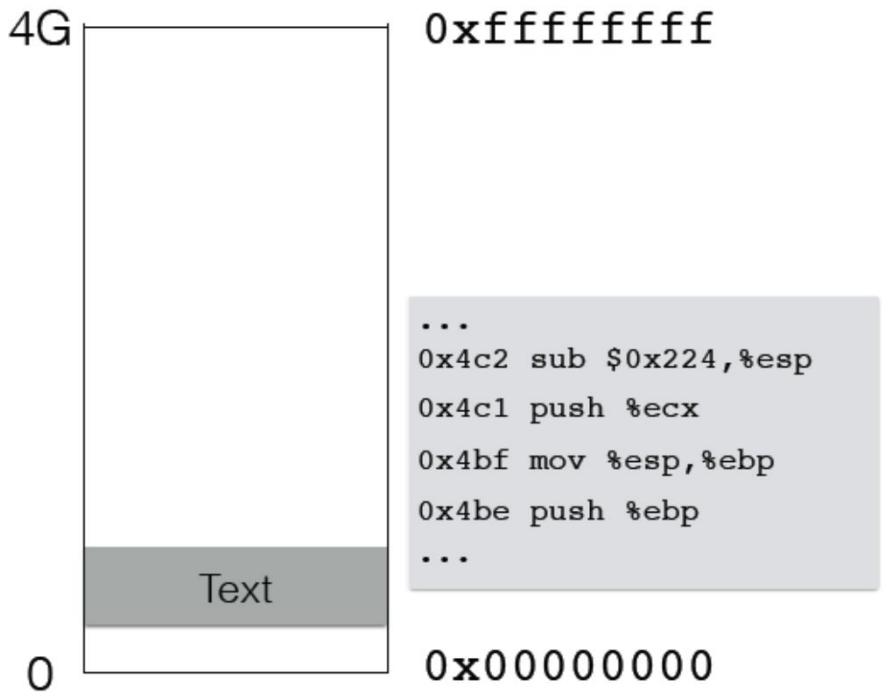
Memory layout refresher

- 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

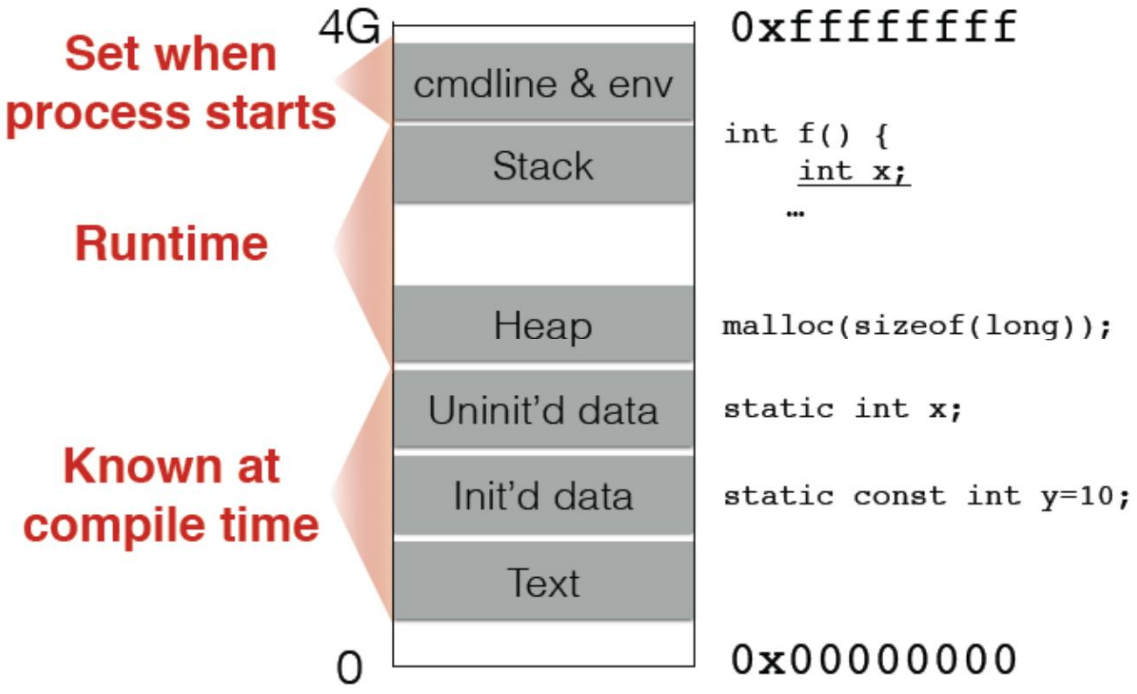
All programs are stored in memory



The instructions themselves are in memory



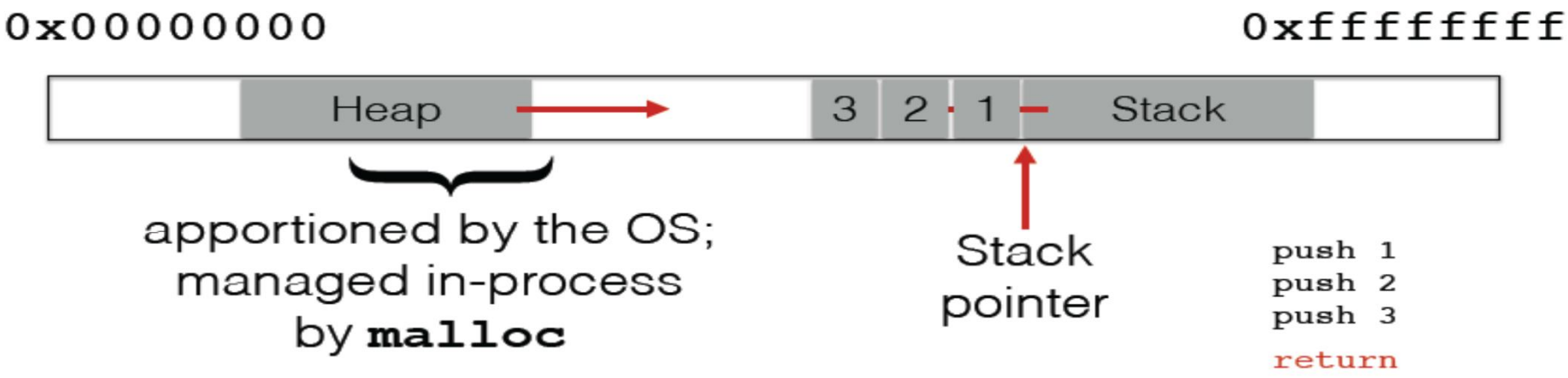
Location of data areas



Memory allocation

Stack and heap grow in opposite directions

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



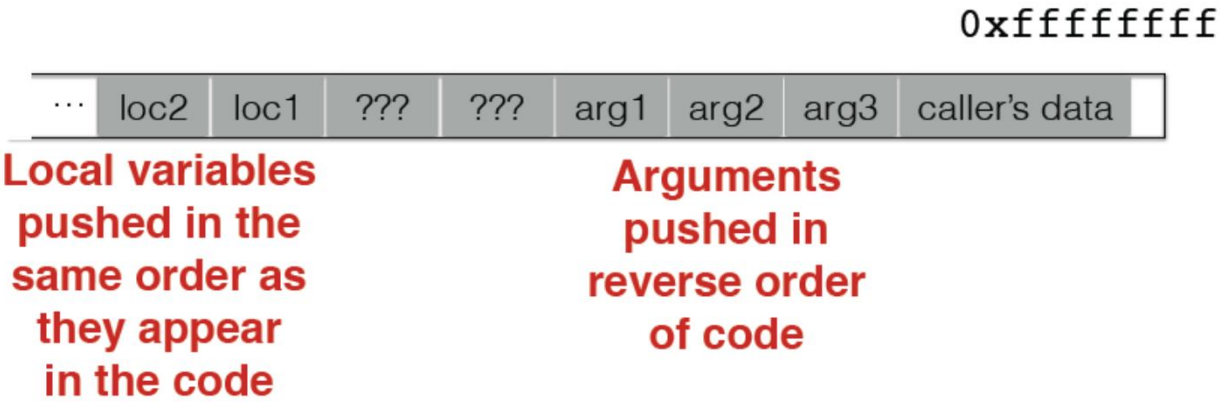
Focusing on the stack for now

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

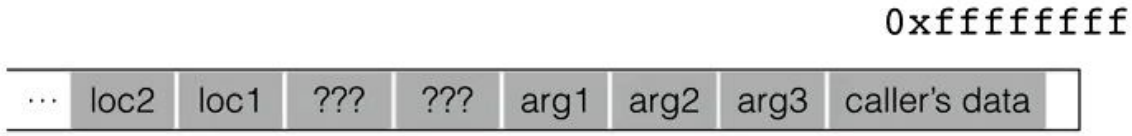
```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    ...
}
```



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

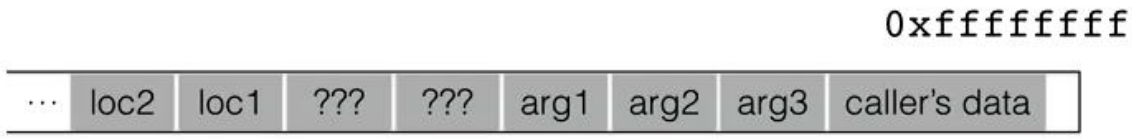
```
void func(char *arg1, int arg2, int arg3)
{
    ...
    loc2++;
    ...
}
```



Accessing variables

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

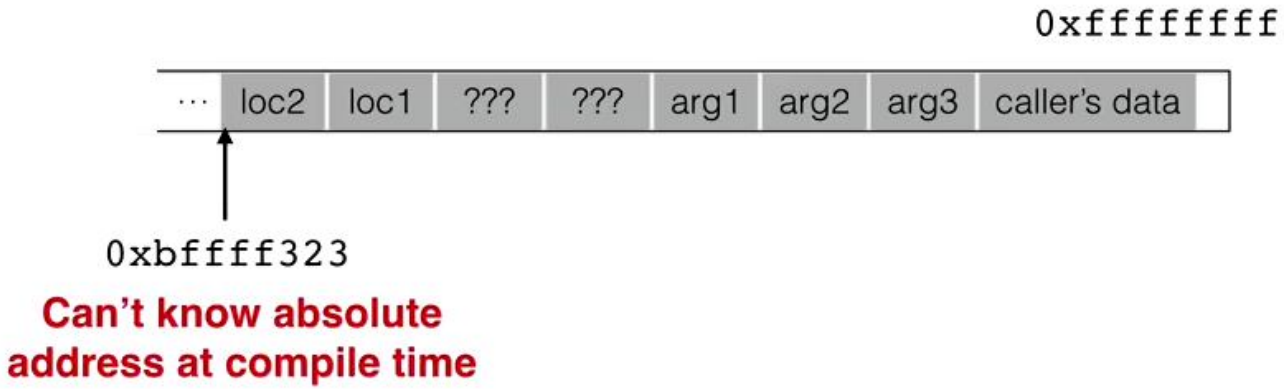
Q: Where is (this) `loc2`?



Accessing variables

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

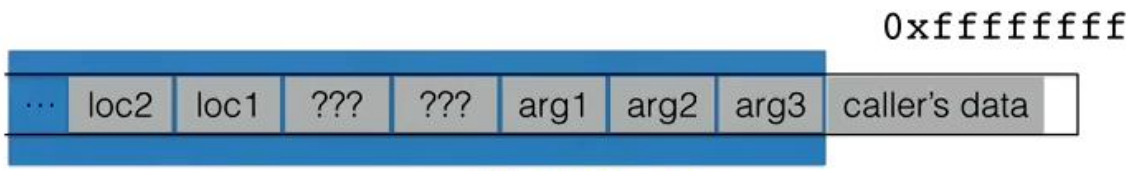
Q: Where is (this) `loc2`?



Accessing variables

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

Q: Where is (this) `loc2`?



Stack frame for `func`

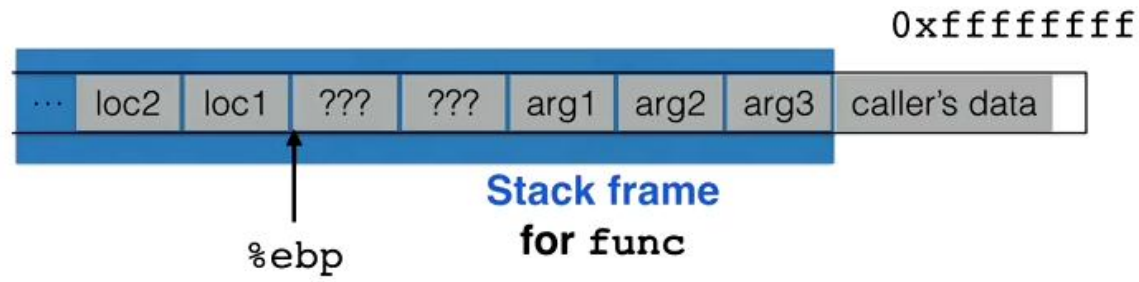
But can know the **relative** address

- **`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)

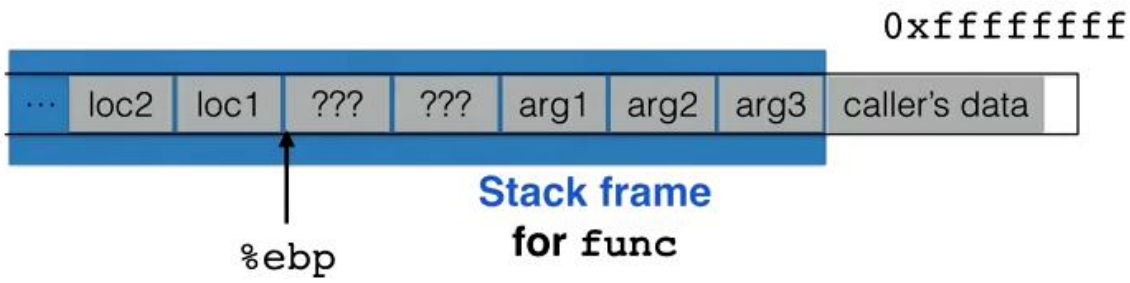


But can know the **relative** address

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

Returning from functions

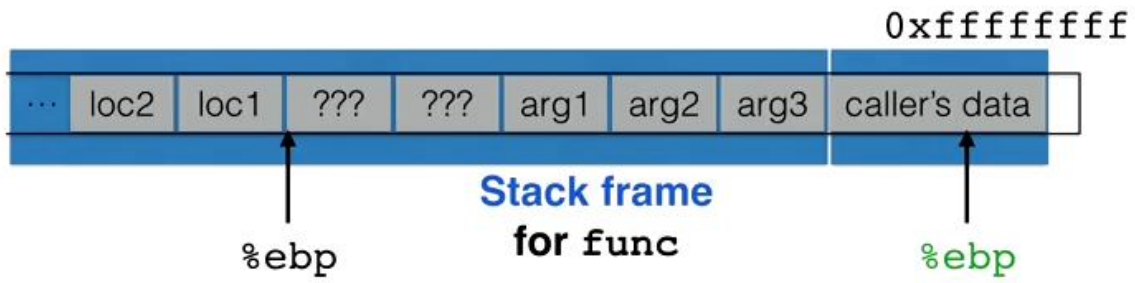
```
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}
```



Returning from functions

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

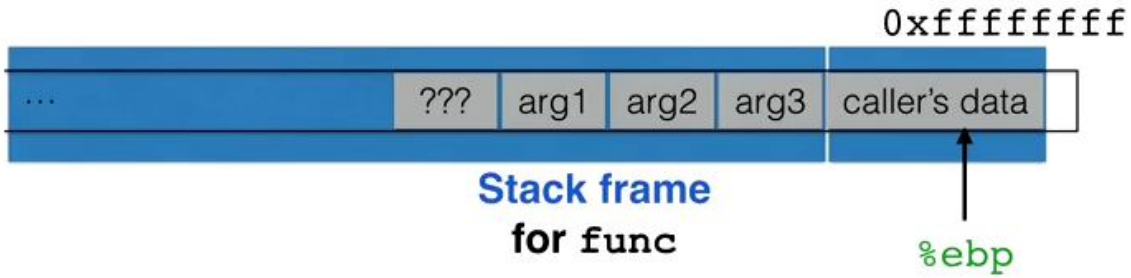
Q: How do we restore %ebp?



Returning from functions

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

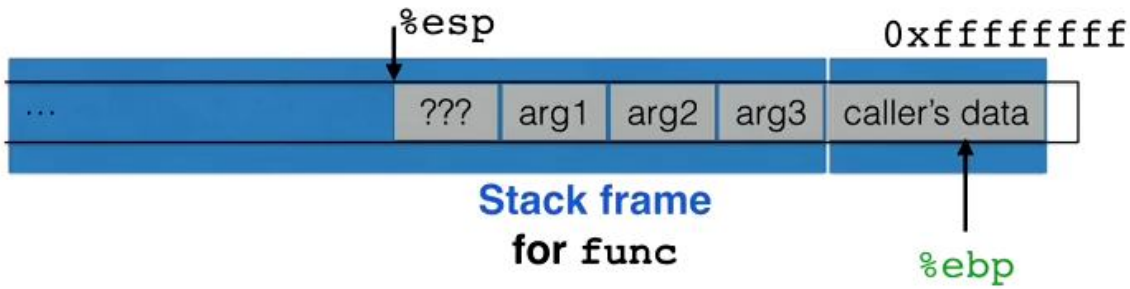
Q: How do we restore %ebp?



Returning from functions

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

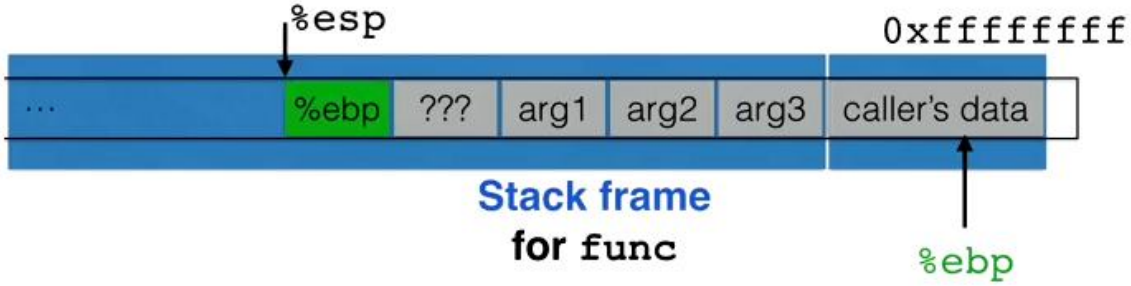
Q: How do we restore %ebp?



Returning from functions

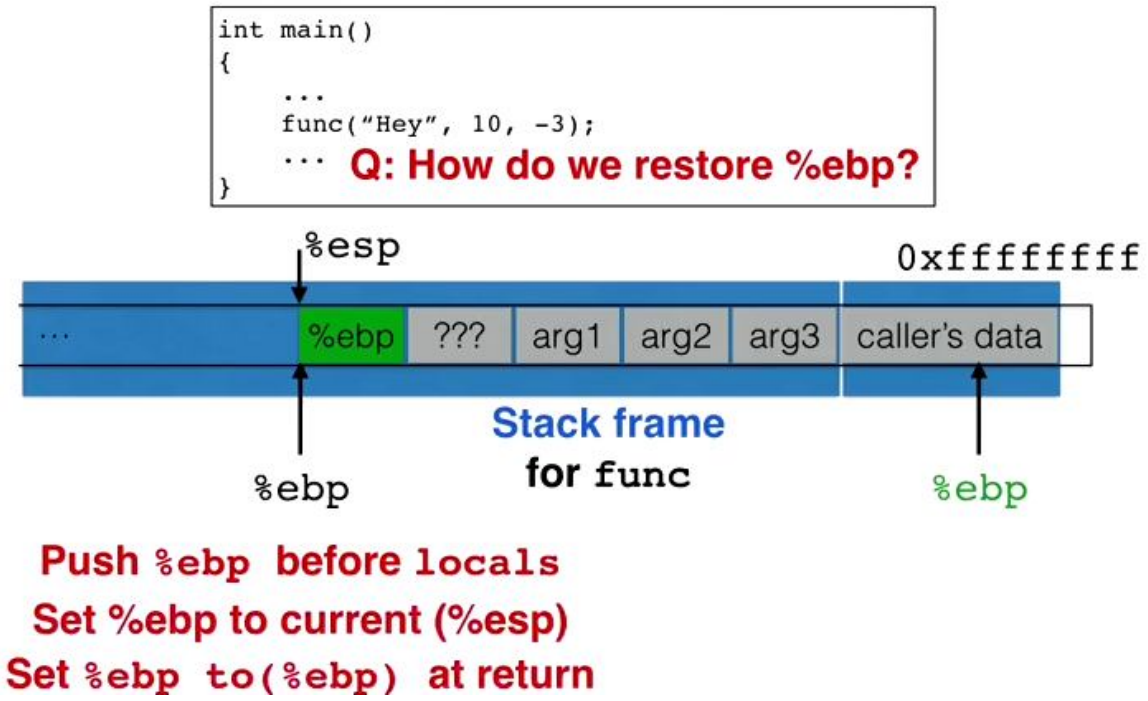
```
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}
```

Q: How do we restore %ebp?



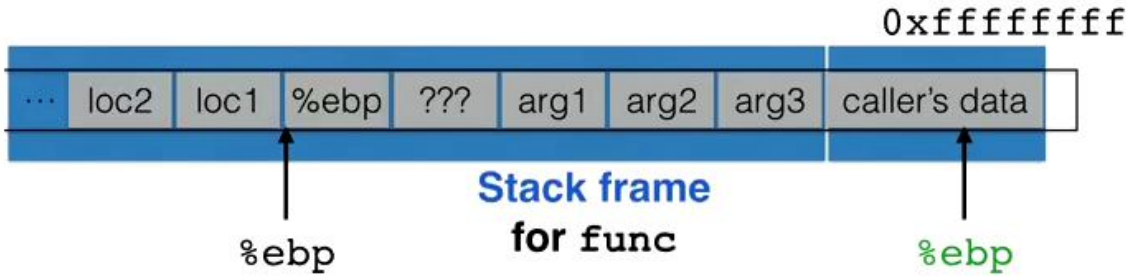
Push `%ebp` before `locals`

Returning from functions



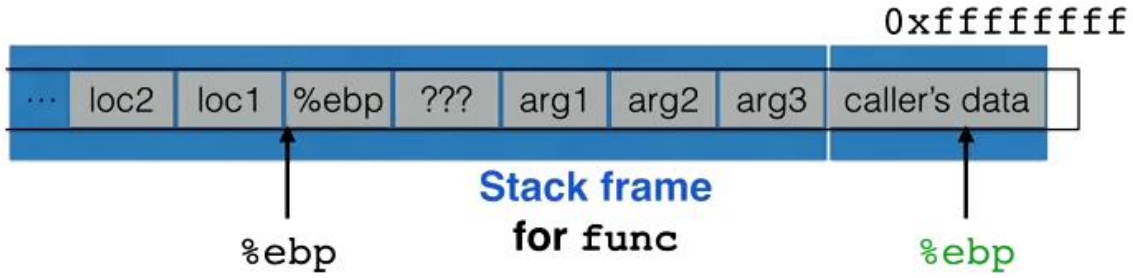
Returning from functions

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

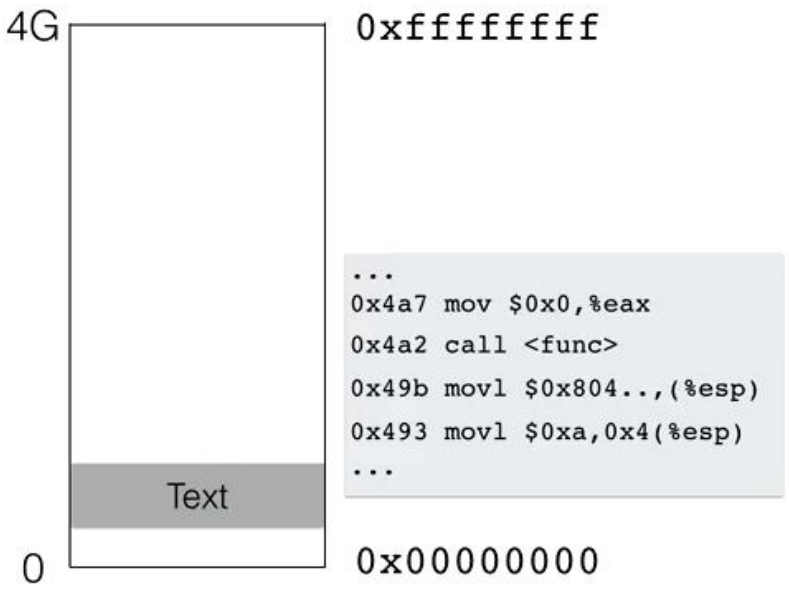


Returning from functions

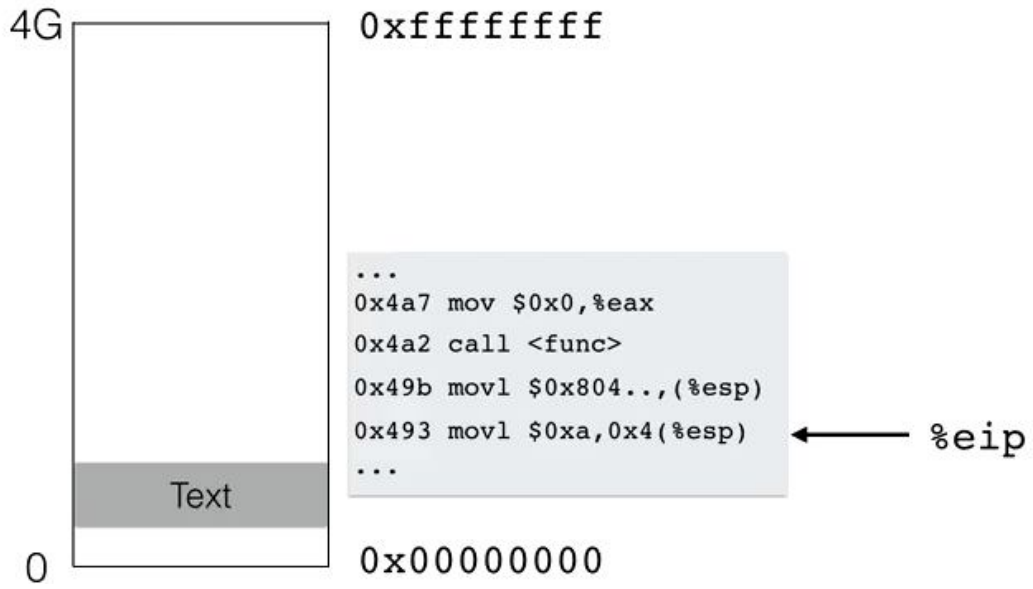
```
int main()
{
    ...
    func("Hey", 10, -3);
    ... Q: How do we resume here?
}
```



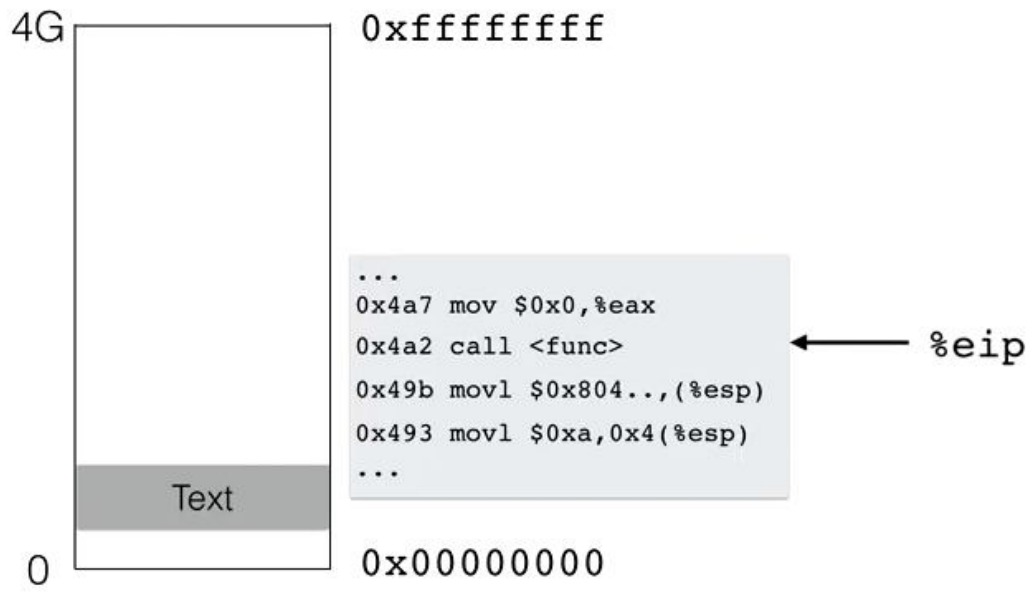
Instructions in memory



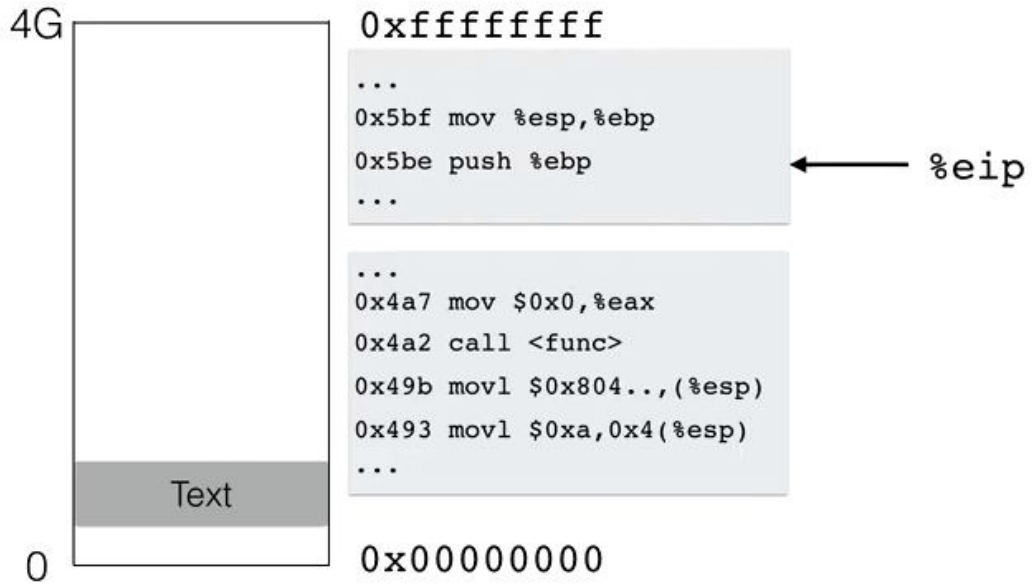
Instructions in memory



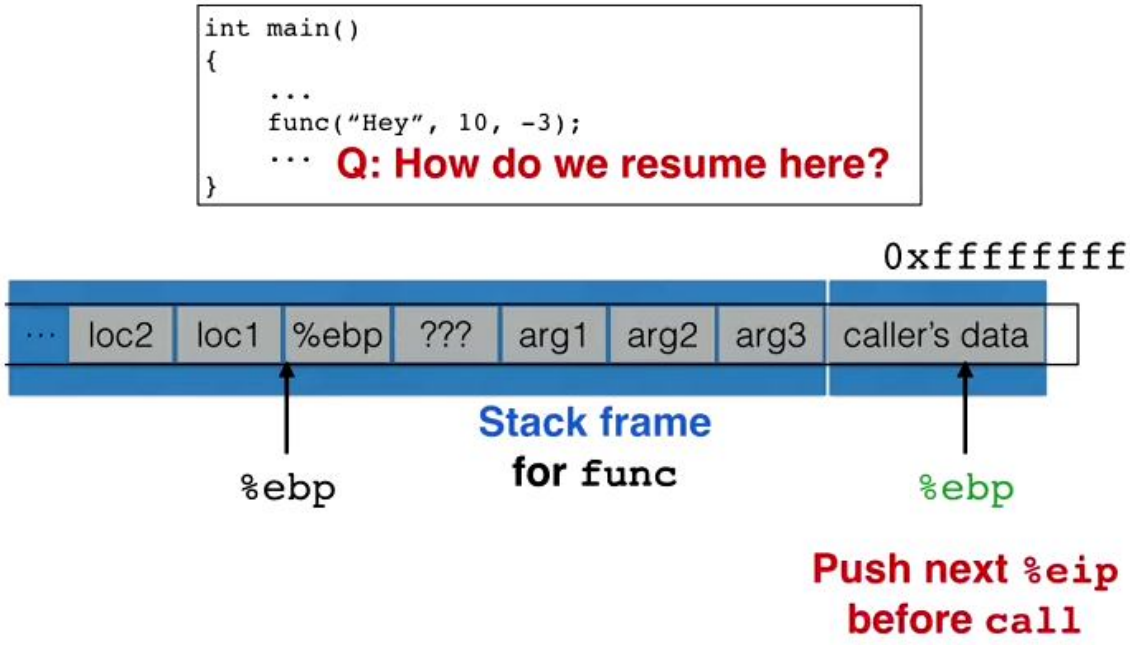
Instructions in memory



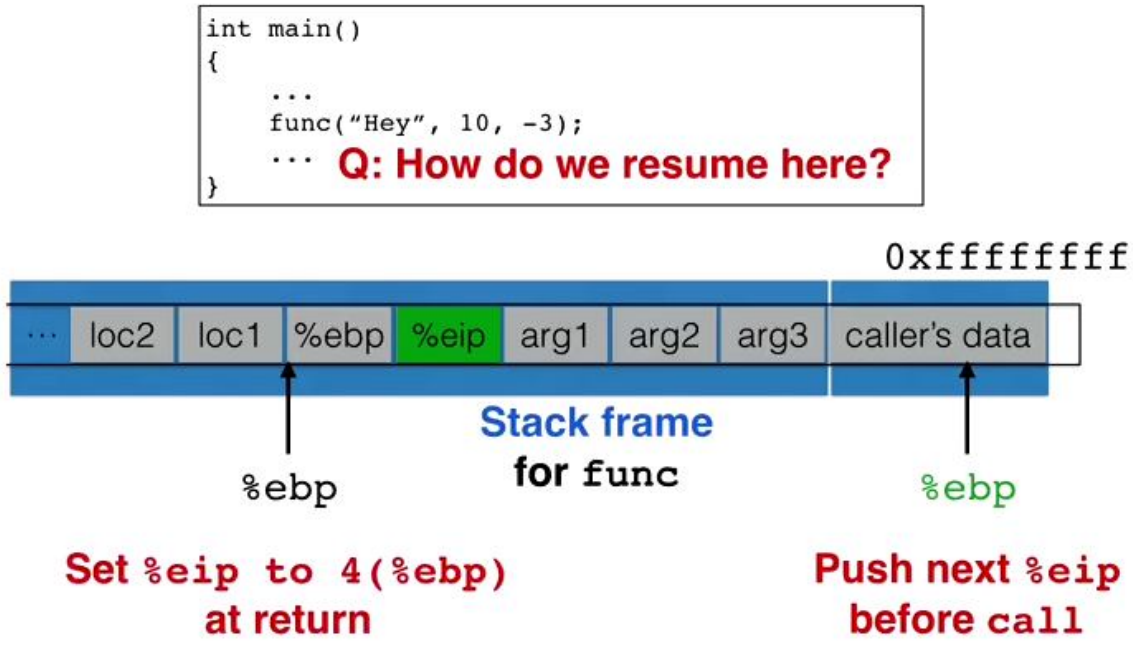
Instructions in memory



How do we resume here?



Instructions in memory



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:

4. **Reset the previous stack frame**: %esp = %ebp, %ebp = (%ebp)
5. **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

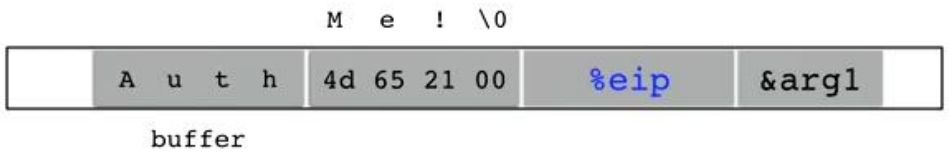
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

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

Benign outcome

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

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

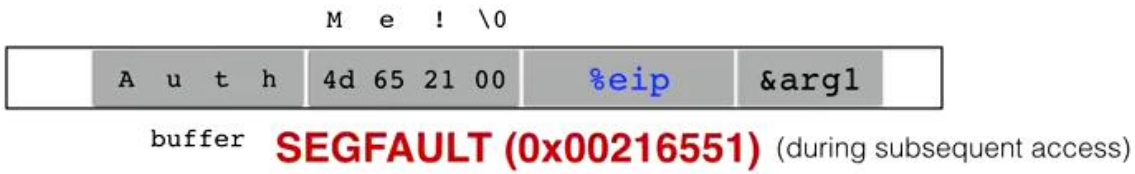


Benign outcome

```
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

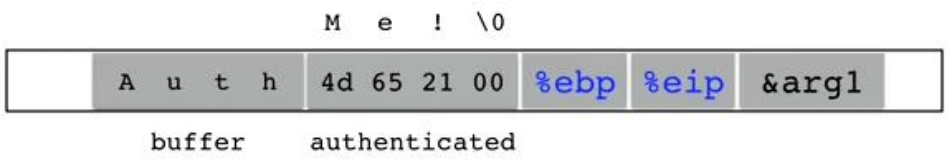
```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

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);
    ...
}
```

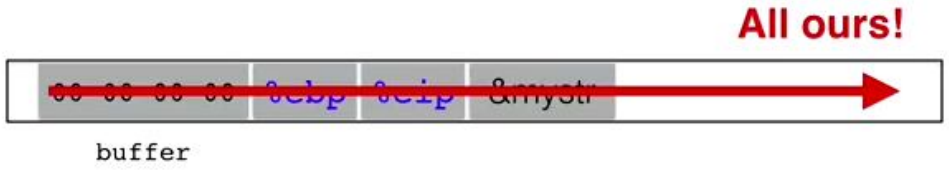


Could it be worse?

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

Could it be worse?

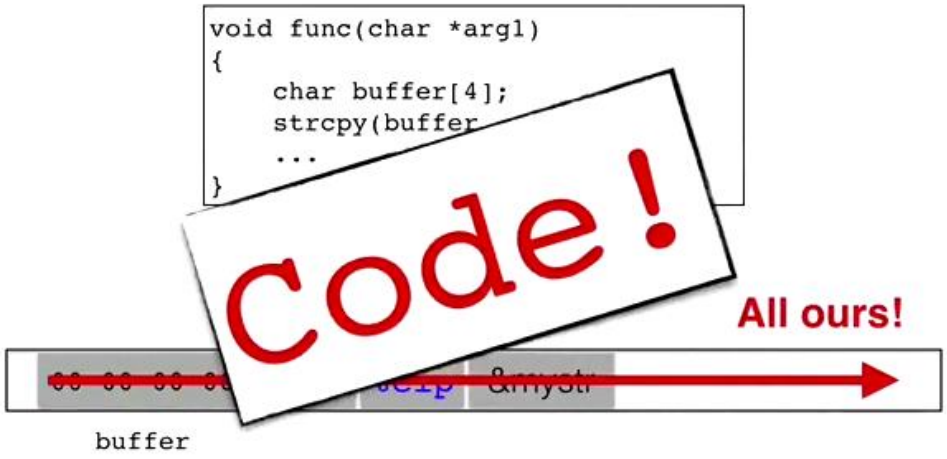
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```



`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?

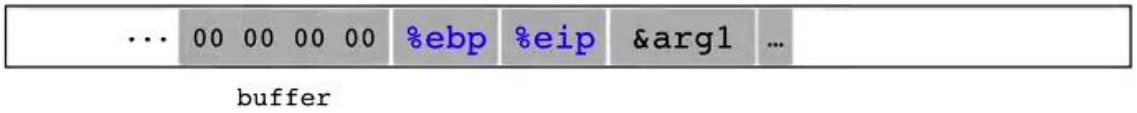
Aside: User-supplied strings

- 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);
    ...
}
```



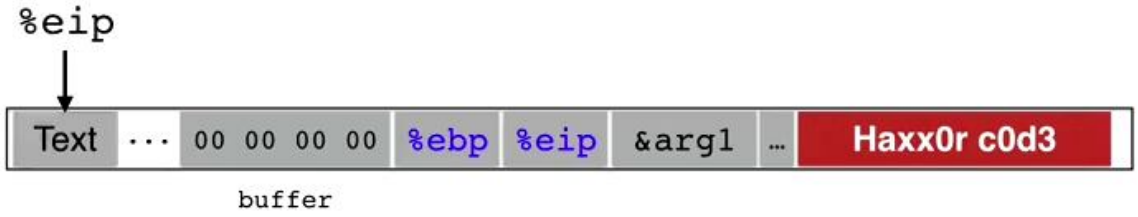
Code Injection: Main idea

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

(1) Load my own code into memory

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

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**

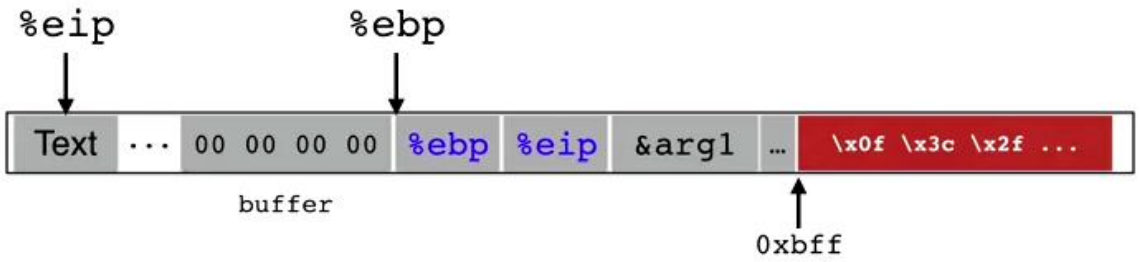
Called function:

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

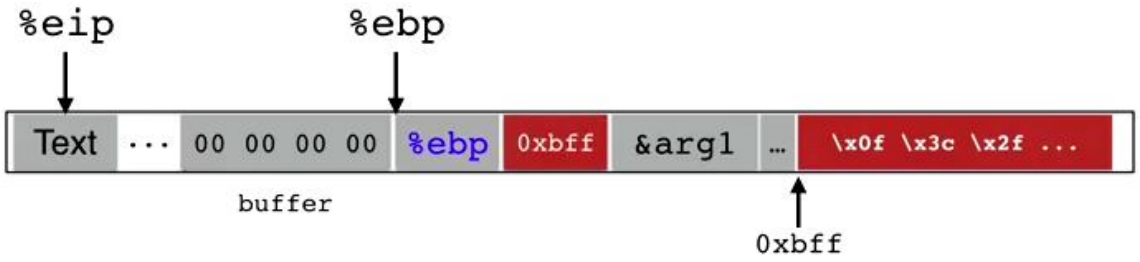
Returning function:

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

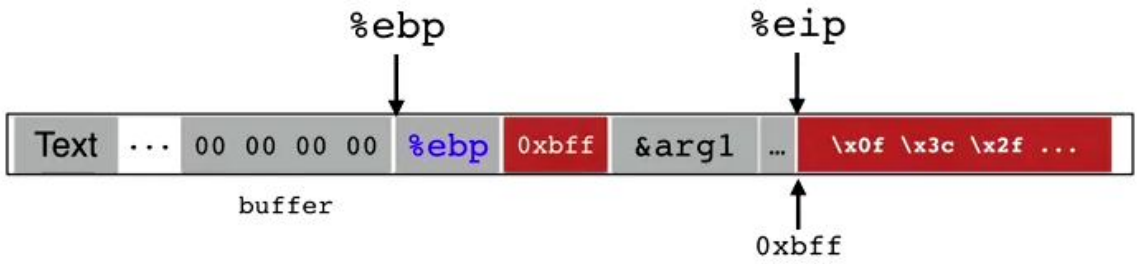
Hijacking the saved %eip



Hijacking the saved %eip



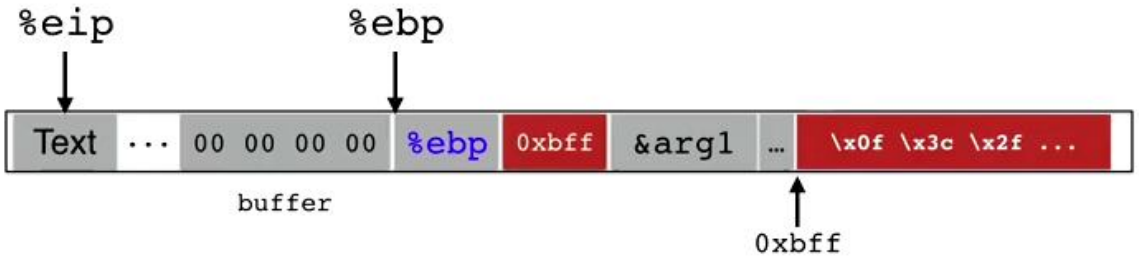
Hijacking the saved %eip



But how do we know the address?

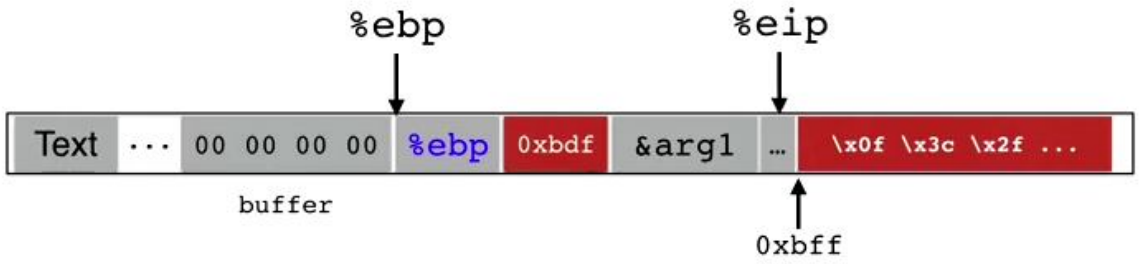
Hijacking the saved %eip

What if we are wrong?



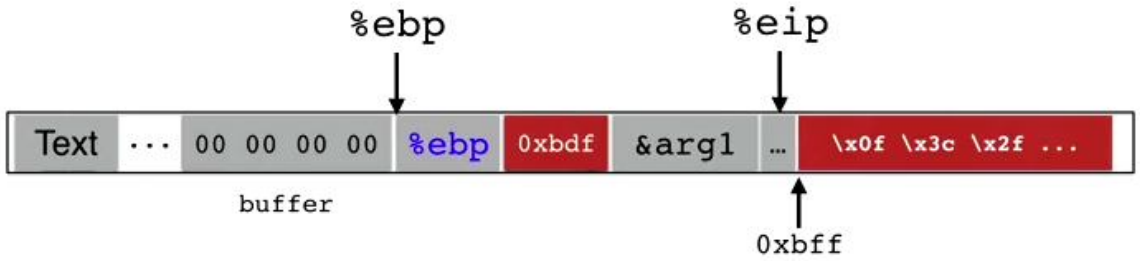
Hijacking the saved %eip

What if we are wrong?



Hijacking the saved %eip

What if we are wrong?

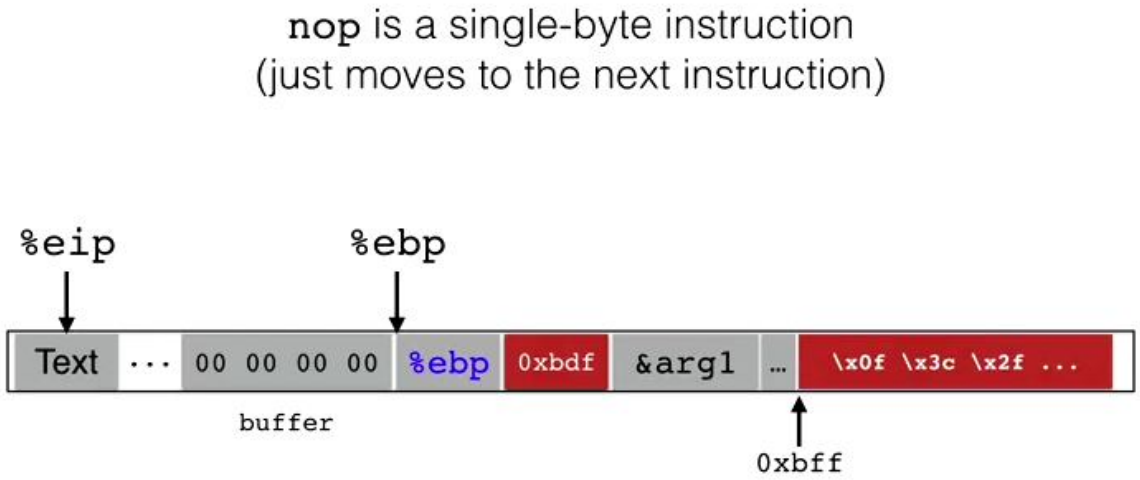


This is most likely data, so the CPU will panic (Invalid Instruction)

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)

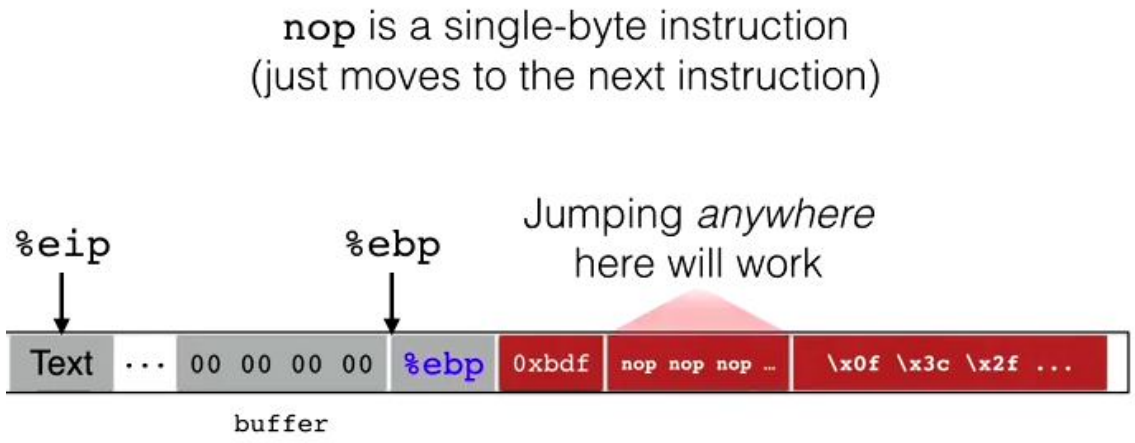
Improving our chances: nop sleds



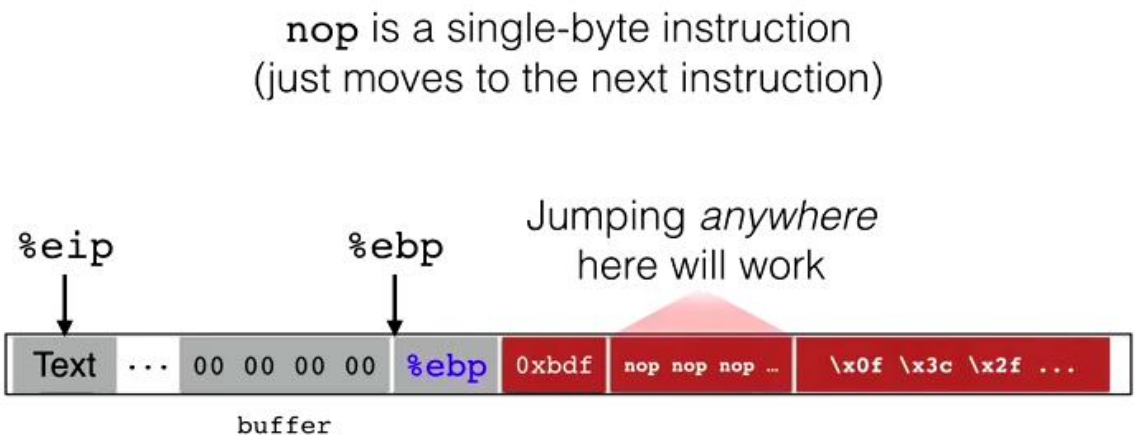
Improving our chances: nop sleds

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

Improving our chances: nop sleds

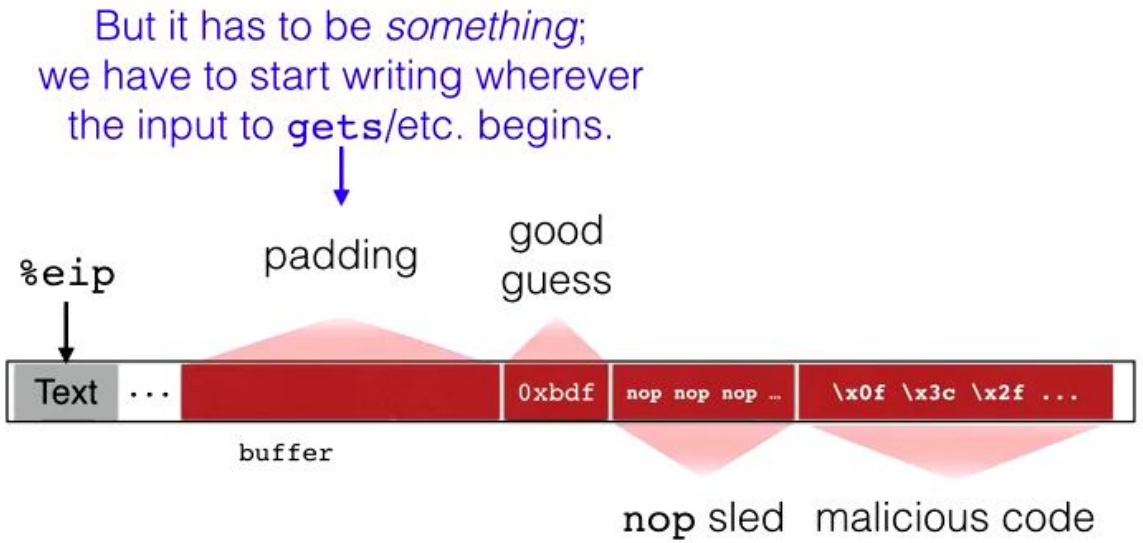


Improving our chances: nop sleds



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

Putting it all together



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);
  }
```

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

- If we set **nresp** to 1073741824 and **sizeof(char*)** is 4

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

- If we set `nresp` to 1073741824 and `sizeof(char*)` is 4
- then `nresp*sizeof(char*)` overflows to become 0

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 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 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

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

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");
 }}
```

May exceed actual message length!

Read integer

Read message

Echo back (partial) message

Sample transcript

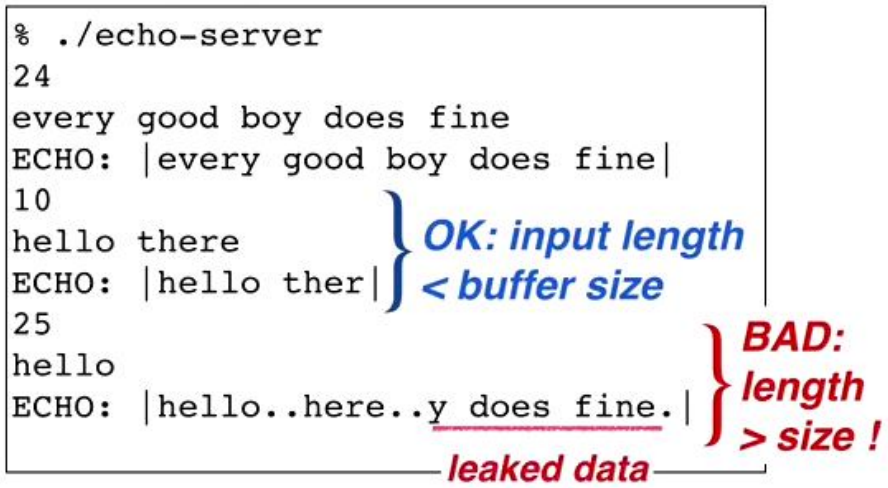
```
% ./echo-server
24
every good boy does fine
ECHO: |every good boy does fine|
```

Sample transcript

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

OK: input length < buffer size

Sample transcript



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
```

TechNewsWorld > Security | **Next Article in Security**

IE's Role in the Google-China War

By Richard Adhikari
Jan 15, 2010 12:25 PM PT

Print
Email

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.

https://www.technewsworld.com/story/69121.html

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);
}
```

What's the difference?

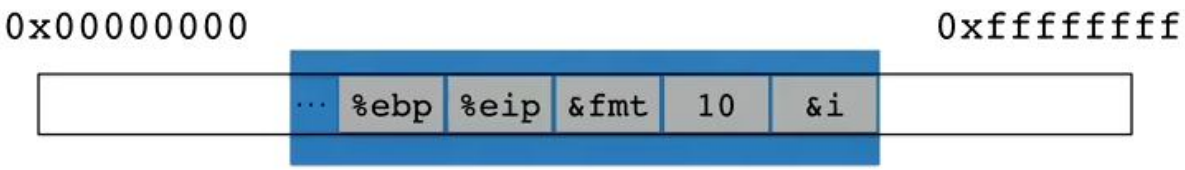
```
void safe()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf("%s",buf);
}
```

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

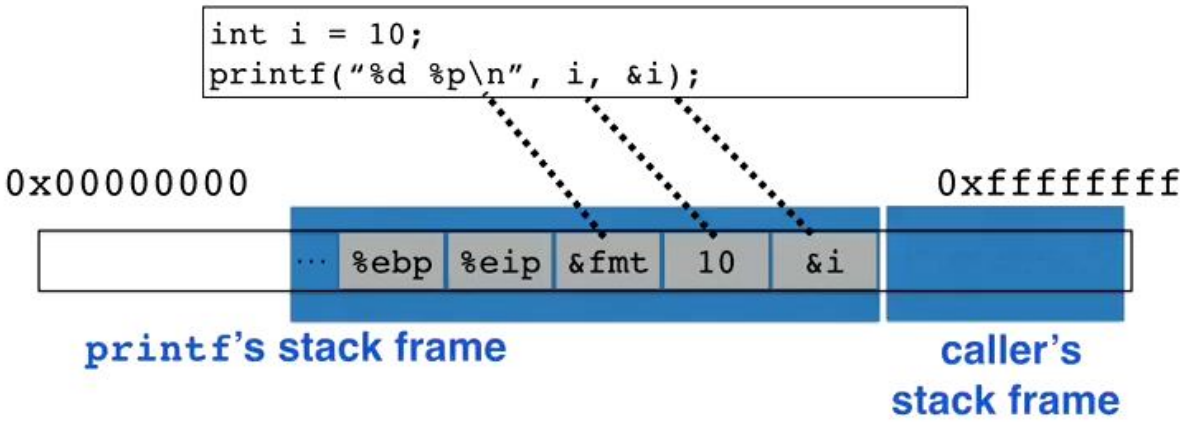
Attacker controls the format string

printf implementation

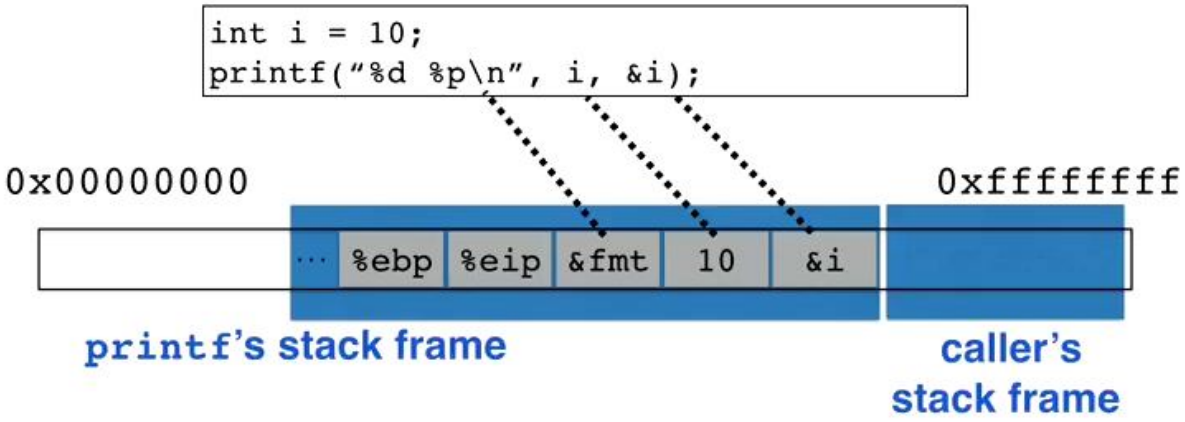
```
int i = 10;
printf("%d %p\n", i, &i);
```



printf implementation

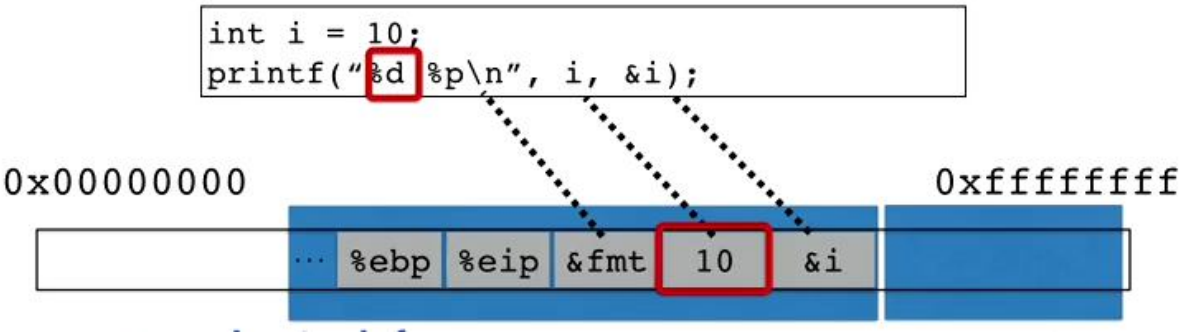


printf implementation



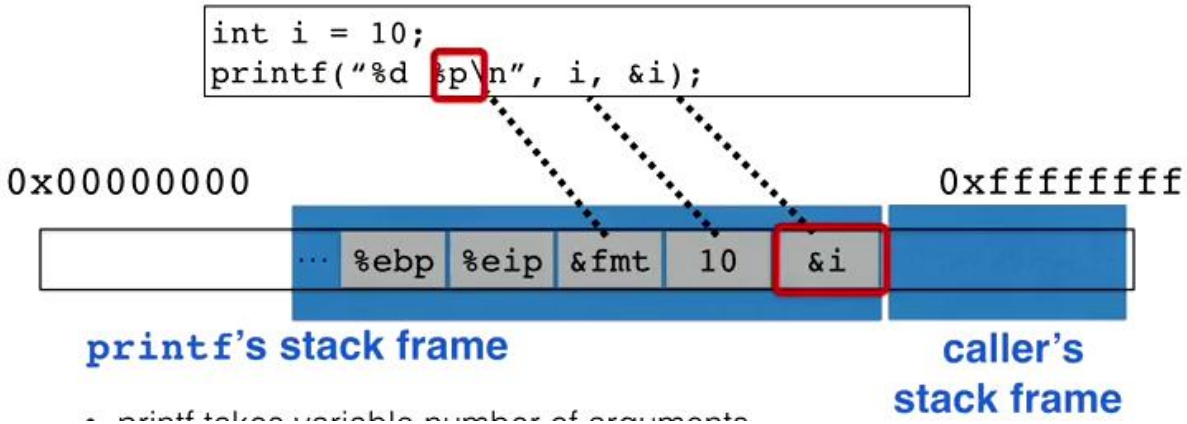
- printf takes variable number of arguments
- printf pays no mind to where the stack frame "ends"
- It presumes that you called it with (at least) as many arguments as specified in the format string

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

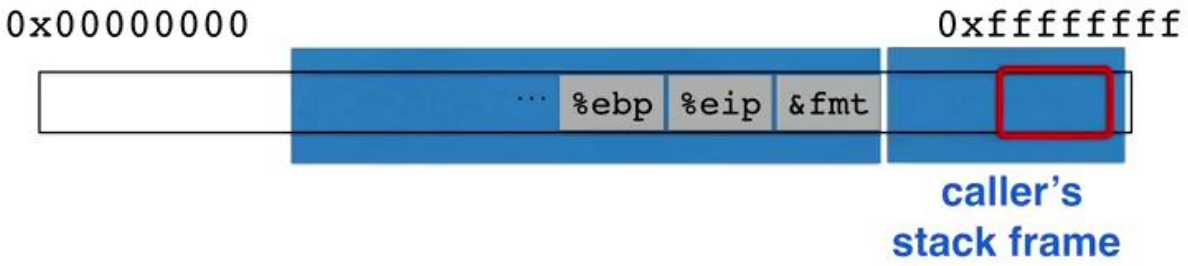
printf implementation



- 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"



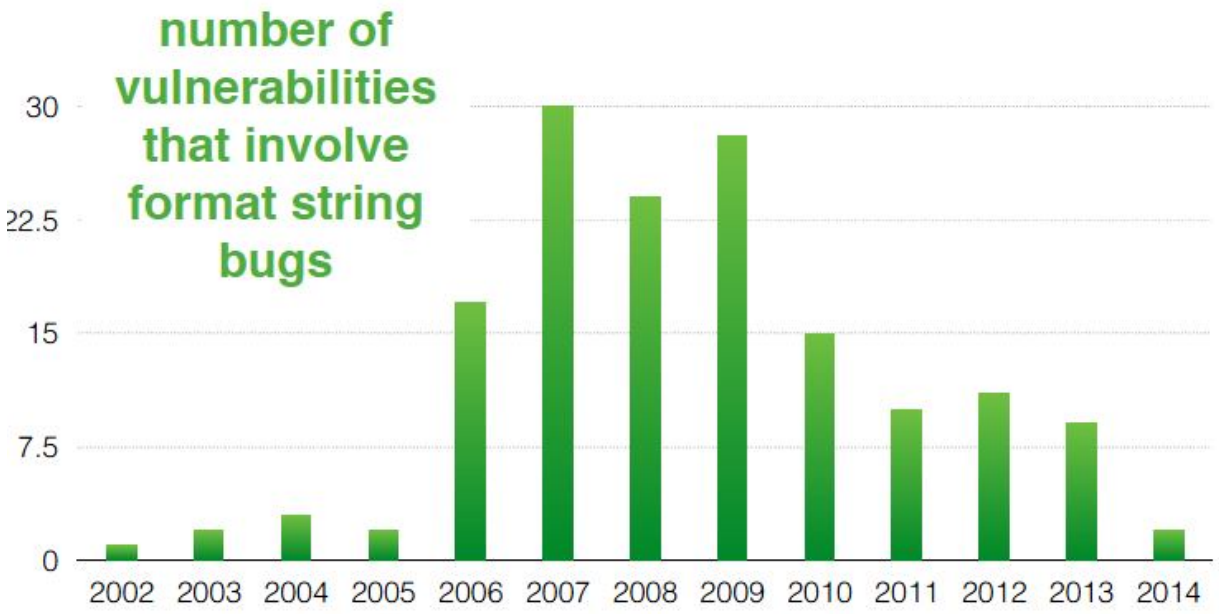
Format string vulnerabilities

- printf("100% desktop");
  - 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?

Stepping back

What do these attacks have in common?!

1. The **attacker** is able to **control some data** that is used by the program
2. The use of that data **permits unintentional access to some memory area** in the program
   - past a buffer
   - to arbitrary positions on the stack

Outline

- **Memory safety** and **type safety!**
  - Properties that, if satisfied, ensure an application is immune to memory attacks
- Automatic defenses
  - **Stack canaries!**
  - Address space layout randomization (**ASLR**)
- Return-oriented programming (**ROP**) attack
  - How Control Flow Integrity (**CFI**) can defeat it
- **Secure coding**

Memory Safety

Low-level attacks enabled by a lack of **Memory Safety**

A memory safe program execution:

1. only **creates pointers** through **standard means**
   - p = malloc(...), or p = &x, or p = &buf[5], etc.

- 2. only uses a pointer to **access memory** that **"belongs" to that pointer**!

Combines two ideas:

temporal safety and **spatial safety**

Spatial safety

- View pointers as triples (**p**,***b***,***e***)
  - **p** is the actual pointer
  - ***b*** is the base of the memory region it may access
  - ***e*** is the extent (bounds) of that region

- **Access allowed** *iff* ***b*** ≤ **p** ≤ **e**-sizeof(typeof(**p**))

- Operations:
  - Pointer arithmetic increments **p**, leaves ***b*** and ***e*** alone
  - Using &: ***e*** determined by size of original type

Examples

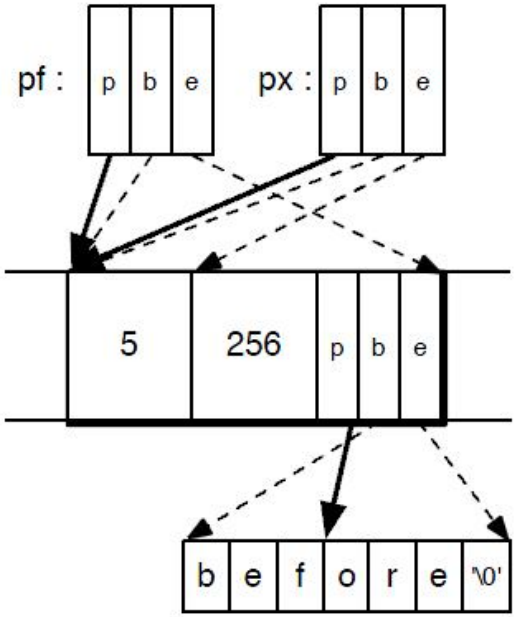
```
struct foo {
 char buf[4];
 int x;
};
```

```
int x;          // assume sizeof(int)=4
int *y = &x;    // p = &x, b = &x, e = &x+4
int *z = y+1;   // p = &x+4, b = &x, e = &x+4
*y = 3;         // OK: &x ≤ &x ≤ (&x+4)-4
*z = 3;         // Bad: &x ≤ &x+4 ≰ (&x+4)-4
```

```
struct foo f = { "cat", 5 };
char *y = &f.buf; // p = b = &f.buf, e = &f.buf+4
y[3] = 's';    // OK: p = &f.buf+3 ≤ (&f.buf+4)-1
y[4] = 'y';    // Bad: p = &f.buf+4 ≰ (&f.buf+4)-1
```

Visualized example

```
struct foo {
  int x;
  int y;
  char *pc;
};
struct foo *pf = malloc(...);
pf->x = 5;
pf->y = 256;
pf->pc = "before";
pf->pc += 3;
int *px = &pf->x;
```



No buffer overflows

- A buffer overflow violates spatial safety

```
void copy(char *src, char *dst, int len)
{
  int i;
  for (i=0;i<len;i++) {
    *dst = *src;
    src++;
    dst++;
  }
}
```

- Overrunning the bounds of the source and/or destination buffers implies either src or dst is illegal

No format string attacks

- The call to printf dereferences illegal pointers

```
char *buf = "%d %d %d\n";
printf(buf);
```

  - View the stack as a buffer defined by the number and types of the arguments it provides
  - The extra format specifiers construct pointers beyond the end of this buffer and dereference them
- Essentially a kind of buffer overflow

Temporal safety

- A **temporal safety violation** occurs when trying to **access undefined memory!**
  - Spatial safety assures it was to a legal region
  - Temporal safety assures that region is still in play
- Memory regions either **defined** or **undefined**
  - Defined means allocated (and active)
  - Undefined means unallocated, uninitialized, or deallocated
- Pretend memory is infinitely large (we never reuse it)

No dangling pointers

- Accessing a freed pointer violates temporal safety

```
int *p = malloc(sizeof(int));
*p = 5;
free(p);
printf("%d\n",*p); // violation
```

The memory dereferenced no longer belongs to p.

- Accessing uninitialized pointers is similarly not OK:

```
int *p;
*p = 5; // violation
```

Integer overflows?

- Allowed as long as they are not used to manufacture an illegal pointer

```
int f() {
  unsigned short x = 65535;
  x++; // overflows to become 0
  printf("%d\n",x); // memory safe
  char *p = malloc(x); // size-0 buffer!
  p[1] = 'a'; // violation
}
```

- Integer overflows often enable buffer overflows
  - happens often enough we think of them independently

For more on memory safety, see http://www.pl-enthusiast.net/2014/07/21/memory-safety/

Most languages memory safe

- The easiest way to avoid all of these vulnerabilities is to **use a memory safe language**
- Modern languages are memory safe
  - Java, Python, C#, Ruby
  - Haskell, Scala, Go, Objective Caml, Rust

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

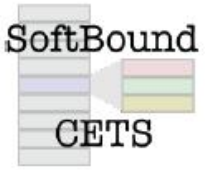
Memory safety for C

- **C/C++ here to stay**. While not memory safe, you can write memory safe programs with them
  - The problem is that there is no guarantee
- Compilers could add **code to check for violations**
  - An out-of-bounds access would result in an immediate failure, like an *ArrayBoundsException* in Java
- This idea has been around for more than 20 years. **Performance has been the limiting factor**
  - Work by Jones and Kelly in 1997 adds 12x overhead
  - Valgrind memcheck adds 17x overhead

Progress

- Research has been **closing the gap!**
  - **CCured** (2004), 1.5x slowdown
    - But no checking in libraries
    - Compiler rejects many safe programs
  - **Softbound/CETS** (2010): 2.16x slowdown
    - Complete checking
    - Highly flexible
  - **Intel MPX** hardware
    - Hardware support to make checking faster

ccured



https://software.intel.com/en-us/blogs/2013/07/22/intel-memoryprotection-extensions-intel-mpx-support-in-the-gnu-toolchain

Type Safety

Type safety

- Each object is ascribed a **type** (int, pointer to int, pointer to function), and
- Operations on the object are always *compatible* with the object's type
  - Type safe programs do not "go wrong" at run-time
  - **Type safety** is **stronger** than memory safety

```
int (*cmp)(char*,char*);
int *p = (int*)malloc(sizeof(int));
*p = 1;
cmp = (int (*)(char*,char*))p;
cmp("hello","bye"); // crash!
```

Memory safe, but not type safe

Dynamically Typed Languages

- **Dynamically typed languages**, like Ruby and Python, which do not require declarations that identify types, can be viewed as **type safe** as well
- Each **object** has **one type: Dynamic**
  - Each operation on a Dynamic object is permitted, but *may be unimplemented*
  - In this case, it *throws an exception*

Well-defined (but unfortunate)

Enforce invariants

- Types really show their strength by **enforcing invariants** in the program
- Notable here is the enforcement of **abstract types**, which characterize modules that keep their **representation hidden** from clients
  - As such, we can reason more confidently about their **isolation** from the rest of the program

For more on type safety, see http://www.pl-enthusiast.net/2014/08/05/type-safety/

Types for Security

- **Type-enforced invariants can relate directly to security properties!**
  - By expressing stronger invariants about data's privacy and integrity, which the type checker then enforces
- **Example: Java with Information Flow (JIF)**

```
int{Alice→Bob} x;
int{Alice→Bob, Chuck} y;
x = y; //OK: policy on x is stronger
y = x; //BAD: policy on y is not
       //as strong as x
```

Types have *security labels*

Labels define what information flows allowed

http://www.cs.cornell.edu/jif

Why not type safety?

- C/C++ often chosen **for performance** reasons
  - Manual memory management
  - Tight control over object layouts
  - Interaction with low-level hardware
- **Typical enforcement** of type safety is **expensive**
  - **Garbage collection** avoids temporal violations
    - Can be as fast as malloc/free, but often uses much more memory
  - **Bounds** and **null-pointer checks** avoid spatial violations
  - **Hiding representation** may **inhibit optimization**
    - Many C-style casts, pointer arithmetic, & operator, not allowed

Not the end of the story

- **New languages** aiming to **provide similar features** to C/C++ while **remaining type safe!**
  - Google's **Go**
  - Mozilla's **Rust**
  - Apple's **Swift**
- **Most applications do not need C/C++!**
  - Or the risks that come with it

These languages may be the future of low-level programming

Avoiding exploitation

Other defensive strategies

Until C is memory safe, what can we do?

Make the bug harder to exploit

- Examine necessary steps for exploitation, make one or more of them difficult, or impossible

Avoid the bug entirely

- Secure coding practices
- Advanced code review and testing
  - E.g., program analysis, penetrating testing (fuzzing)

Strategies are complementary: Try to avoid bugs, but add protection if some slip through the cracks

Avoiding exploitation

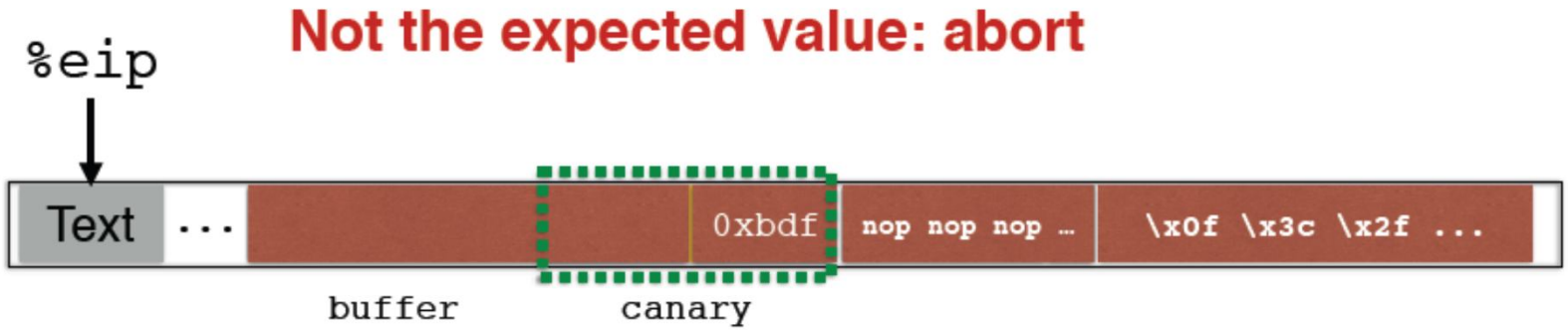
Recall the steps of a stack smashing attack:

- Putting attacker code into the memory (no zeroes)
- Getting %eip to point to (and run) attacker code
- Finding the return address (guess the raw addr)

How can we make these attack steps more difficult?

- **Best case**: Complicate exploitation by changing the **libraries**, **compiler** and/or **operating system**
  - Then we don't have to change the application code
  - *Fix is in the architectural design, not the code*

Detecting overflows with canaries



What value should the canary have?

Canary values
From StackGuard [Wagle & Cowan]

1. Terminator canaries (CR, LF, NUL (i.e., 0), -1)
   - Leverages the fact that scanf etc. don't allow these
2. Random canaries
   - Write a new random value @ each process start
   - Save the real value somewhere in memory
   - Must write-protect the stored value
3. Random XOR canaries
   - Same as random canaries
   - But store canary XOR some control info, instead

Recall our challenges

- Putting code into the memory (no zeroes)
  - **Defense**: Make this detectable with **canaries**
- Getting %eip to point to (and run) attacker code
  - **Defense**: **Make stack (and heap) non-executable**
  - **Defense**: **Use Address-space Layout Randomization**
- Finding the return address (guess the raw addr)
  - **Defense**: **Use Address-space Layout Randomization**

So: even if canaries could be bypassed, no code loaded by the attacker can be executed (will panic)

Randomly place standard libraries and other elements in memory, making them harder to guess

ASLR today

- **Available on modern operating systems**
  - Available on Linux in 2004, and adoption on other systems came slowly afterwards; **most by 2011**
- Caveats:
  - **Only shifts the offset** of memory areas
    - Not locations within those areas
  - **May not apply to program code**, just libraries
  - **Need sufficient randomness**, or can brute force
    - 32-bit systems typically offer 16 bits = 65536 possible starting positions; sometimes 20 bits. Shacham demonstrated a brute force attack could defeat such randomness in 216 seconds (on 2004 hardware)
  - **64-bit systems more promising**, e.g., 40 bits possible

Return Oriented Programming (ROP)

Cat and mouse

- **Defense: Make stack/heap nonexecutable** to prevent injection of code
  - **Attack response: Jump/return to libc**
- **Defense: Hide the address of desired libc code** or **return address** using ASLR
  - **Attack response: Brute force search** (for 32-bit systems) or **information leak** (format string vulnerability)
- **Defense: Avoid using libc code entirely** and use code in the program text instead
  - **Attack response:** Construct needed functionality using **return oriented programming (ROP)**

Return-oriented Programming

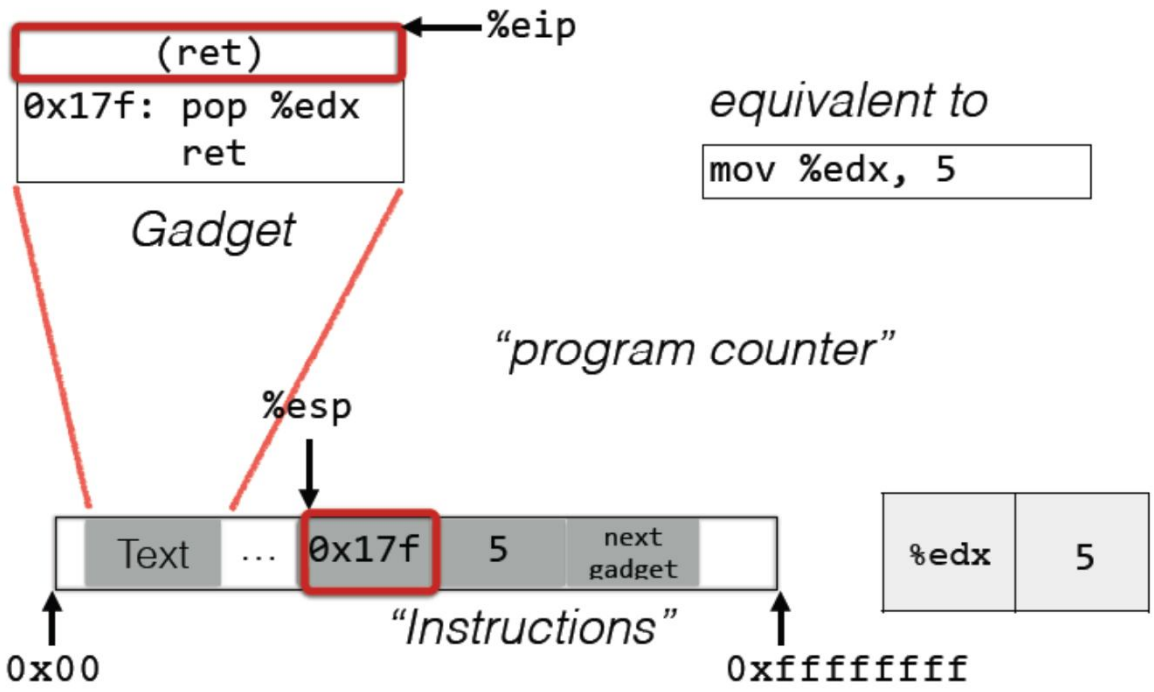
- Introduced by Hovav Shacham in 2007
- *The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)*, CCS'07
- Idea: rather than use a single (libc) function to run your shellcode, **string together pieces of existing code, called *gadgets***, to do it instead

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

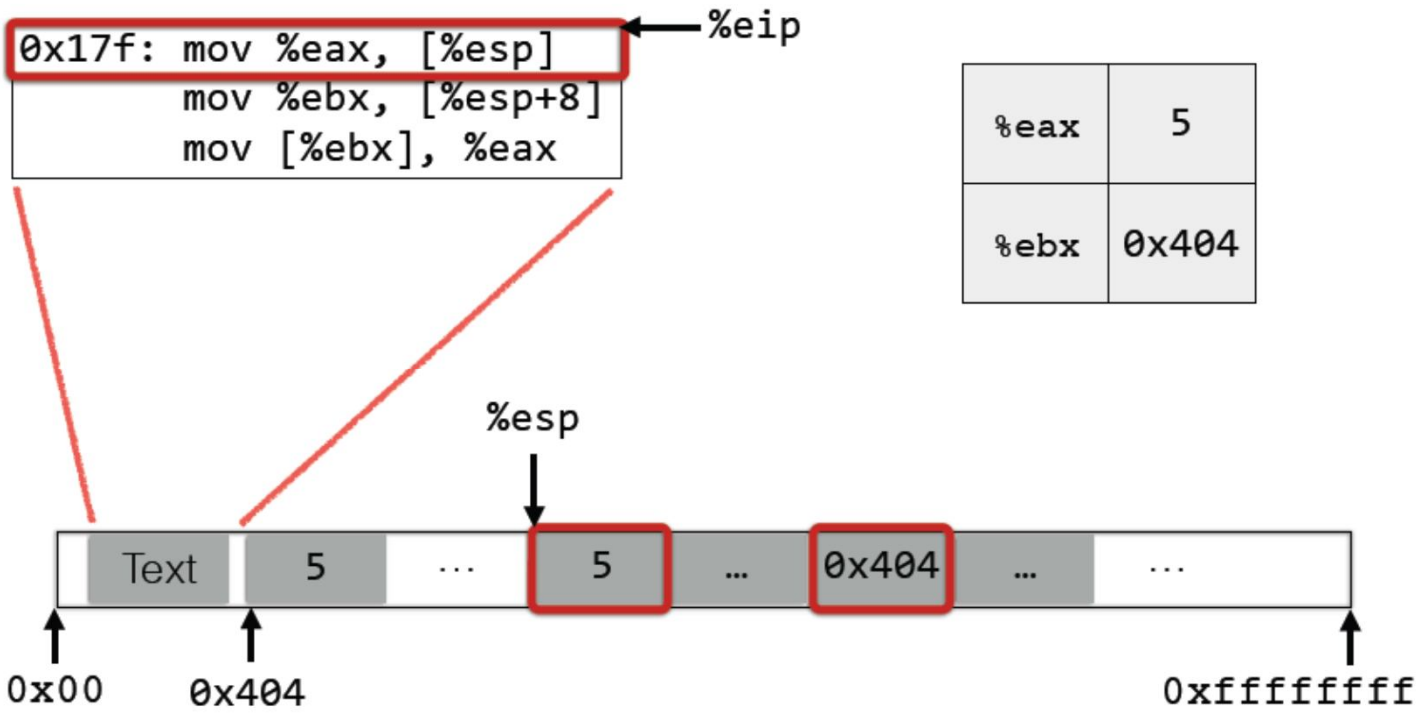
Approach

- Gadgets are instruction groups that end with ret
- Stack serves as the code
  - %esp = program counter
  - Gadgets invoked via ret instruction
  - Gadgets get their arguments via pop, etc.
    - Also on the stack

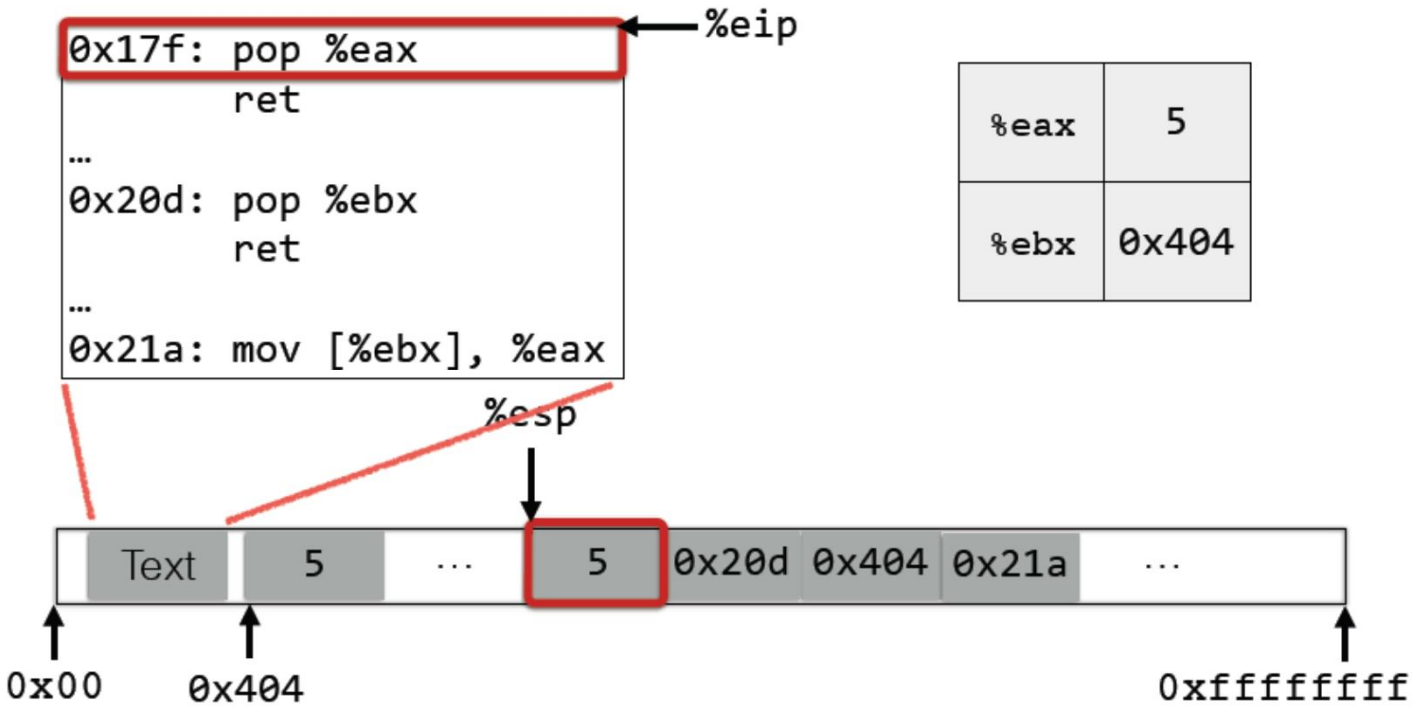
Simple example



Code Sequence



Equivalent ROP Sequence



Whence the gadgets?

- How can we find gadgets to construct an exploit?
  - **Automate a search of the target binary for gadgets** (look for ret instructions, work backwards)
    - Cf. https://github.com/0vercl0k/rp
- Are there sufficient gadgets to do anything interesting?
  - Yes: Shacham found that for significant codebases (e.g., libc), **gadgets are Turing complete**
    - Especially true on x86's dense instruction set
  - Schwartz et al (USENIX Security '11) have automated gadget shellcode creation, though not needing/requiring Turing completeness

Blind ROP

- **Defense: Randomizing the location of the code** (by compiling for position independence) on a 64-bit machine makes attacks very difficult
  - Recent, published attacks are often for 32-bit versions of executables
- **Attack response: Blind ROP**
  - If server restarts on a crash, but does not re-randomize:
    1. Read the stack to **leak canaries and a return address**
    2. Find gadgets (at run-time) to **effect call to write**
    3. **Dump binary to find gadgets for shellcode**

http://www.scs.stanford.edu/brop/

Defeat!

- The blind ROP team was able to **completely automatically**, only **through remote interactions**, develop a **remote code exploit for nginx**, a popular web server
  - The exploit was carried out on a 64-bit executable with full stack canaries and randomization
- Conclusion: **give an inch, and they take a mile?**
- Put another way: **Memory safety is really useful!**

Secure Coding