

Today

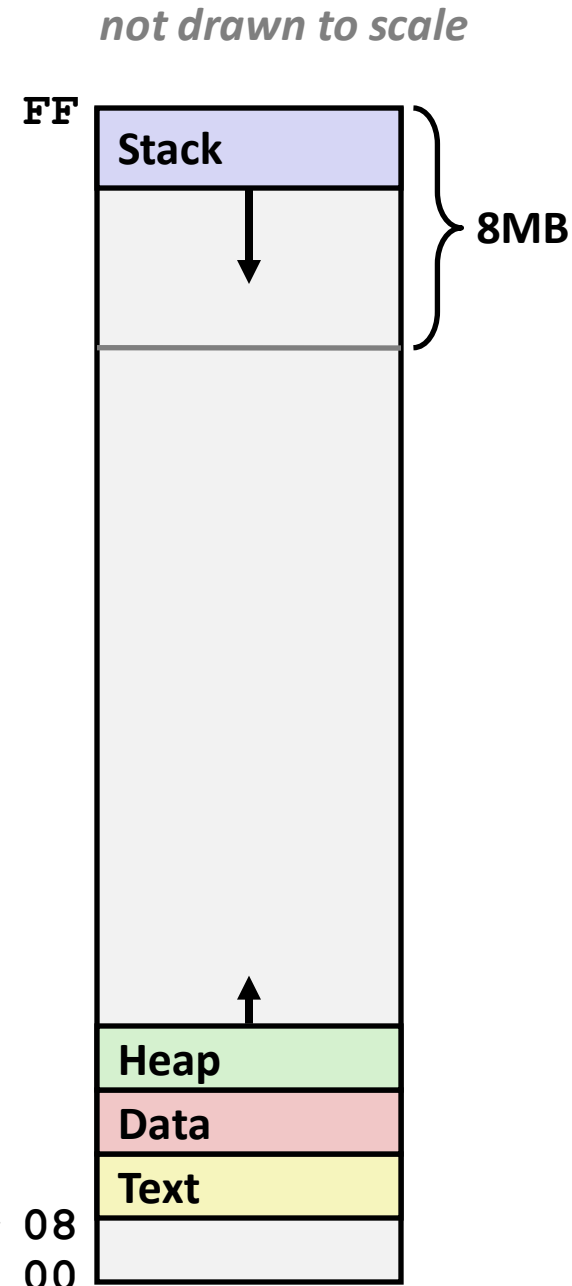
- **Memory layout**
- **Buffer overflow, worms, and viruses**

IA32 Linux Memory Layout

- **Stack**
 - Runtime stack (8MB limit)
- **Heap**
 - Dynamically allocated storage
 - When call `malloc()`, `calloc()`, `new()`
- **Data**
 - Statically allocated data
 - E.g., arrays & strings declared in code
- **Text**
 - Executable machine instructions
 - Read-only

Upper 2 hex digits
= 8 bits of address

08
00



Memory Allocation Example

```
char big_array[1<<24]; /* 16 MB */
char huge_array[1<<28]; /* 256 MB */

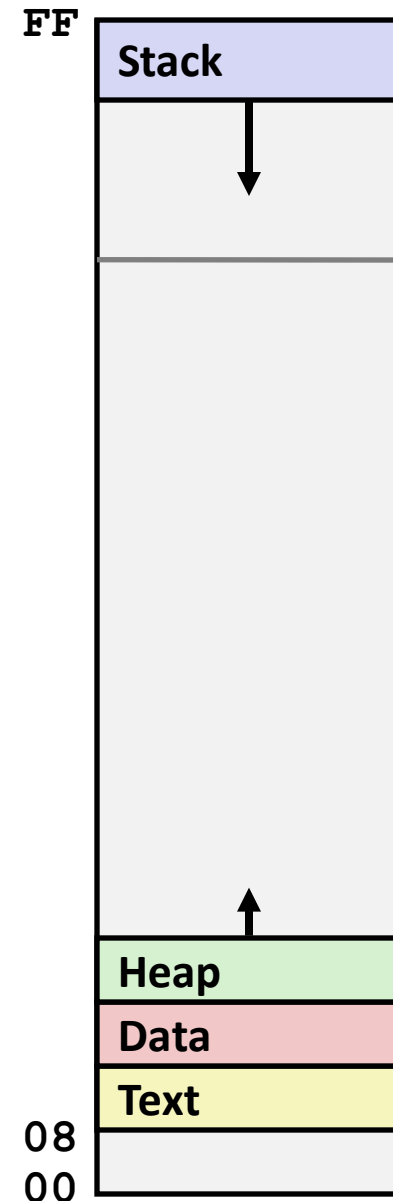
int beyond;
char *p1, *p2, *p3, *p4;

int useless() { return 0; }

int main()
{
    p1 = malloc(1 << 28); /* 256 MB */
    p2 = malloc(1 << 8); /* 256 B */
    p3 = malloc(1 << 28); /* 256 MB */
    p4 = malloc(1 << 8); /* 256 B */
    /* Some print statements ... */
}
```

Where does everything go?

not drawn to scale



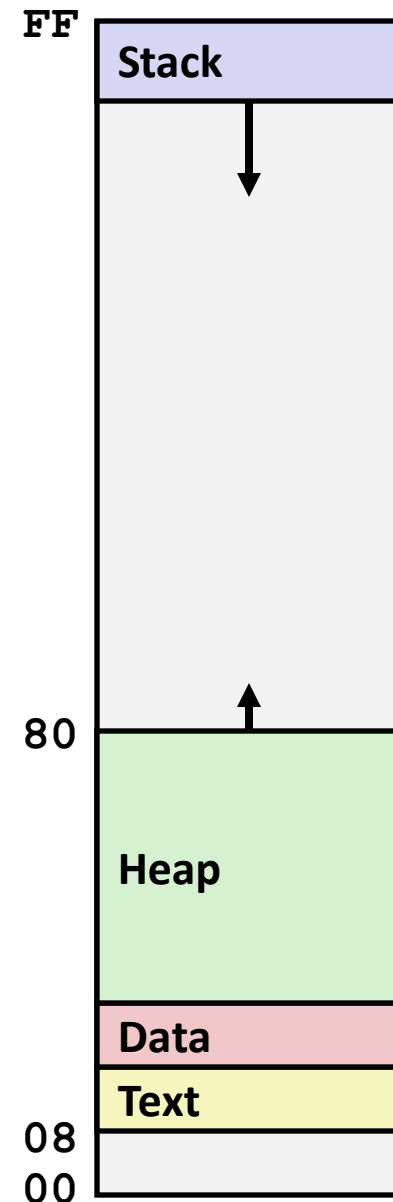
IA32 Example Addresses

address range $\sim 2^{32}$

<code>\$esp</code>	<code>0xffffbcd0</code>
<code>p3</code>	<code>0x65586008</code>
<code>p1</code>	<code>0x55585008</code>
<code>p4</code>	<code>0x1904a110</code>
<code>p2</code>	<code>0x1904a008</code>
<code>&p2</code>	<code>0x18049760</code>
<code>beyond</code>	<code>0x08049744</code>
<code>big_array</code>	<code>0x18049780</code>
<code>huge_array</code>	<code>0x08049760</code>
<code>main()</code>	<code>0x080483c6</code>
<code>useless()</code>	<code>0x08049744</code>
<code>final malloc()</code>	<code>0x006be166</code>

`malloc()` is dynamically linked
address determined at runtime

not drawn to scale



Internet Worm

■ November, 1988

- Internet Worm attacks thousands of Internet hosts.
- How did it happen?

Internet Worm

■ November, 1988

- Internet Worm attacks thousands of Internet hosts.
- How did it happen?

■ The Internet Worm was based on *stack buffer overflow* exploits!

- many Unix functions do not check argument sizes
- allows target buffers to overflow

String Library Code

■ Implementation of Unix function gets ()

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

■ Anything interesting?

String Library Code

■ Implementation of Unix function `gets()`

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

- No way to specify limit on number of characters to read

■ Similar problems with other Unix functions

- **strcpy**: Copies string of arbitrary length
- **scanf**, **fscanf**, **sscanf**, when given **%s** conversion specification

Vulnerable Buffer Code

```
/* Echo Line */  
void echo()  
{  
    char buf[4]; /* Way too small! */  
    gets(buf);  
    puts(buf);  
}
```

```
int main()  
{  
    printf("Type a string:");  
    echo();  
    return 0;  
}
```

```
unix>./bufdemo  
Type a string:1234567  
1234567
```

```
unix>./bufdemo  
Type a string:12345678  
Segmentation Fault
```

```
unix>./bufdemo  
Type a string:123456789ABC  
Segmentation Fault
```

Buffer Overflow Disassembly

080484f0 <echo>:

```

80484f0: 55          push    %ebp
80484f1: 89 e5       mov     %esp,%ebp
80484f3: 53          push    %ebx
80484f4: 8d 5d f8    lea     0xffffffff8(%ebp),%ebx
80484f7: 83 ec 14    sub     $0x14,%esp
80484fa: 89 1c 24    mov     %ebx, (%esp)
80484fd: e8 ae ff ff ff call    80484b0 <gets>
8048502: 89 1c 24    mov     %ebx, (%esp)
8048505: e8 8a fe ff ff call    8048394 <puts@plt>
804850a: 83 c4 14    add     $0x14,%esp
804850d: 5b          pop     %ebx
804850e: c9          leave
804850f: c3          ret

```

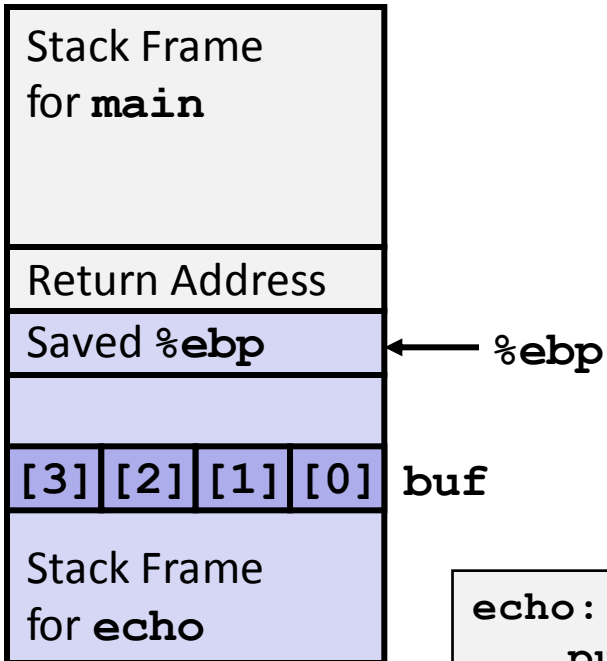
```

80485f2: e8 f9 fe ff ff call    80484f0 <echo>
80485f7: 8b 5d fc    mov     0xfffffffffc(%ebp),%ebx
80485fa: c9          leave
80485fb: 31 c0       xor     %eax,%eax
80485fd: c3          ret

```

Buffer Overflow Stack

Before call to gets



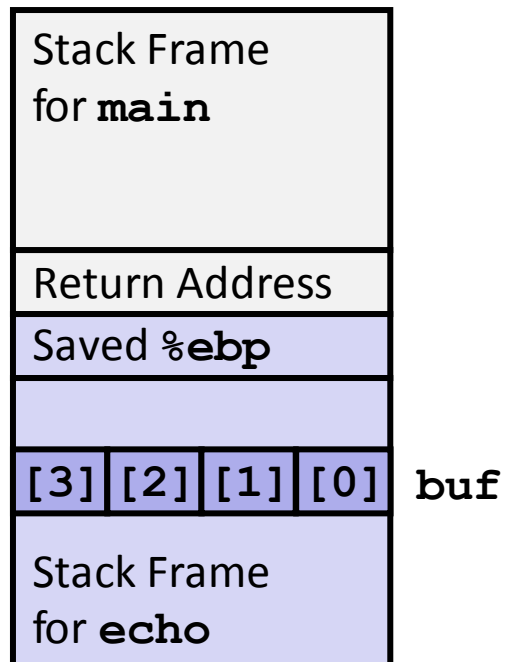
```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushl %ebp                # Save %ebp on stack
    movl %esp, %ebp
    pushl %ebx                # Save %ebx
    leal -8(%ebp), %ebx       # Compute buf as %ebp-8
    subl $20, %esp            # Allocate stack space
    movl %ebx, (%esp)         # Push buf addr on
stack
    call gets                  # Call gets
```

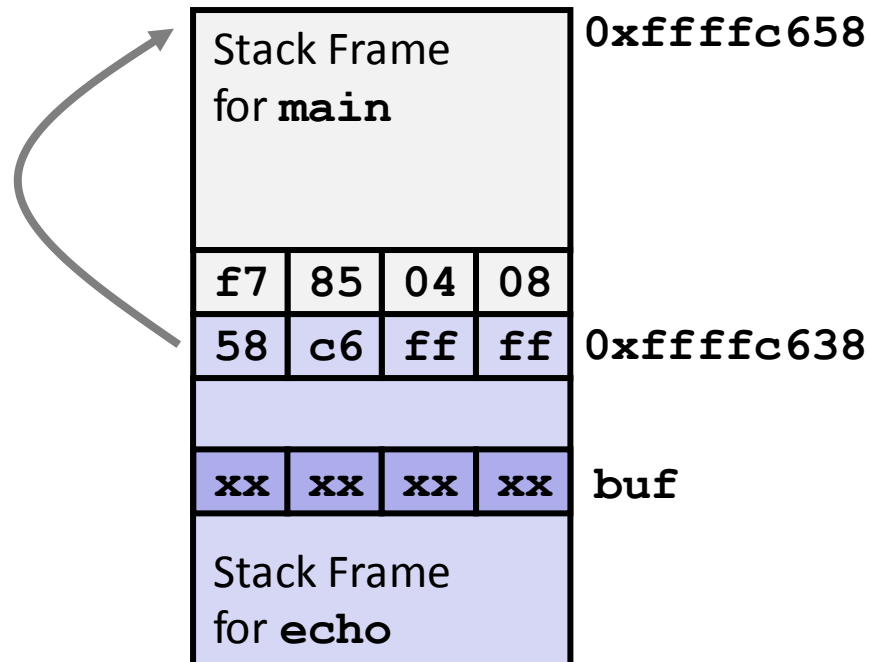
. . .

Buffer Overflow Stack Example

Before call to gets



Before call to gets

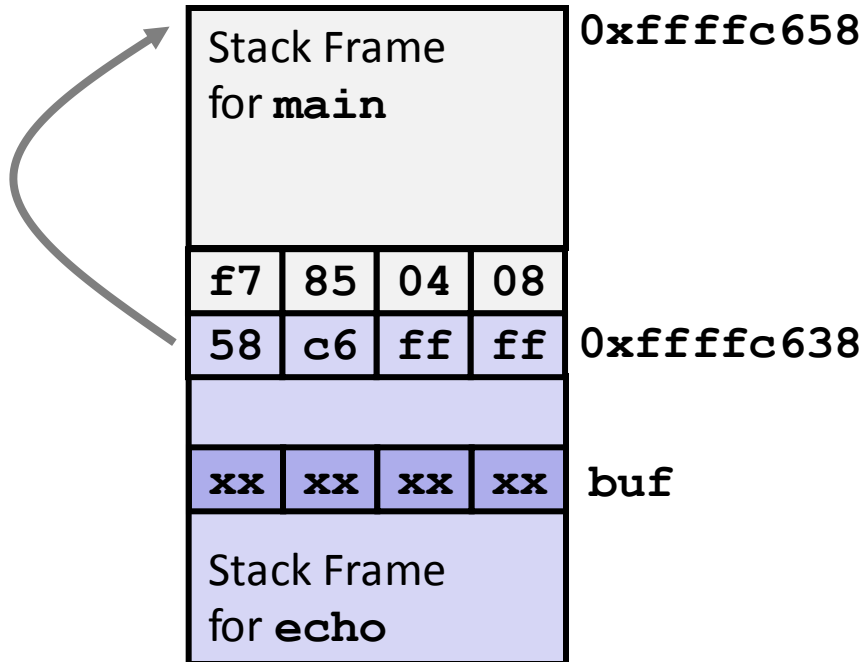


```
80485f2: call 80484f0 <echo>
```

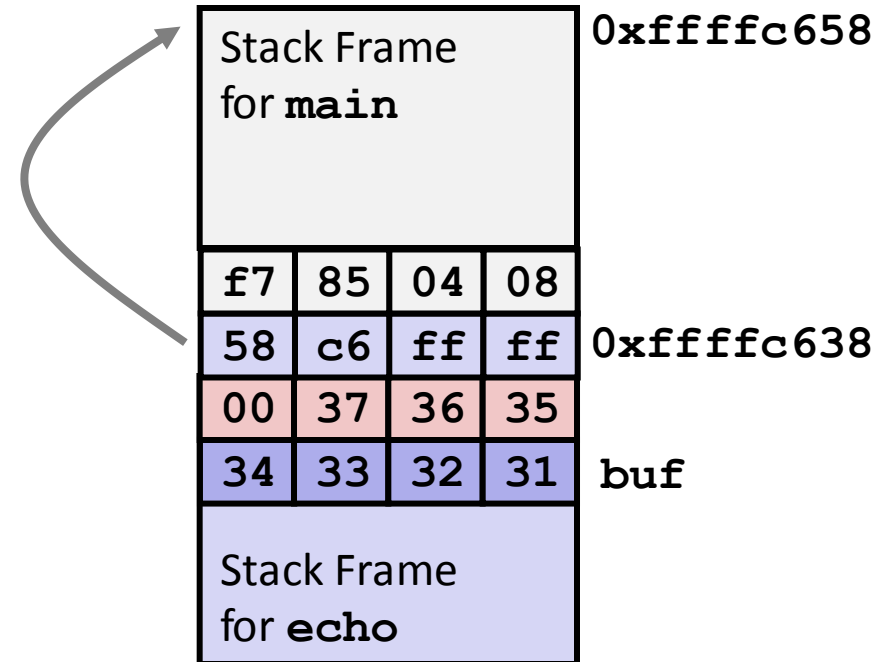
```
80485f7: mov 0xffffffffc(%ebp), %ebx # Return Point
```

Buffer Overflow Example #1

Before call to gets



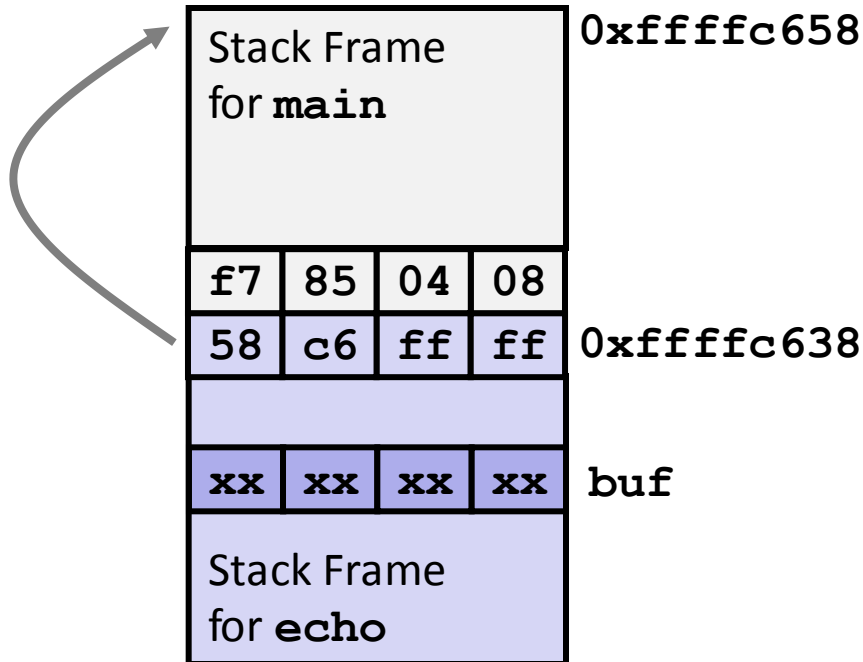
Input 1234567



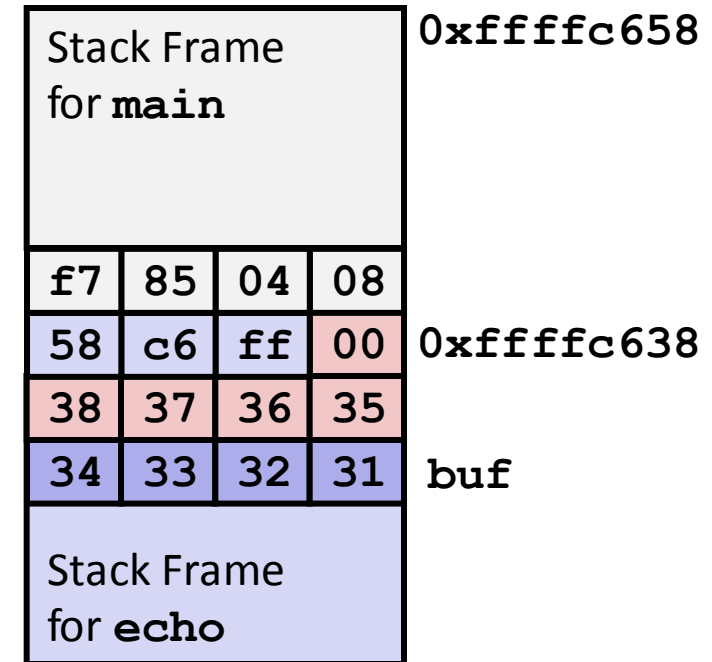
Overflow buf, but no problem

Buffer Overflow Example #2

Before call to gets



Input 12345678



Base pointer corrupted

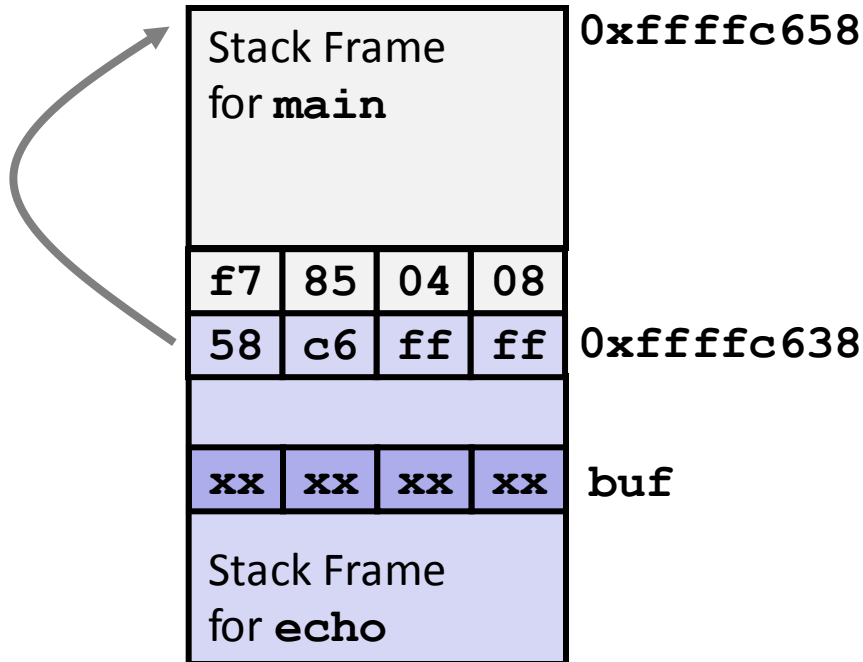
```

. . .
804850a: 83 c4 14  add    $0x14,%esp  # deallocate space
804850d: 5b         pop     %ebx      # restore %ebx
804850e: c9         leave   # movl %ebp, %esp; popl %ebp
804850f: c3         ret      # Return

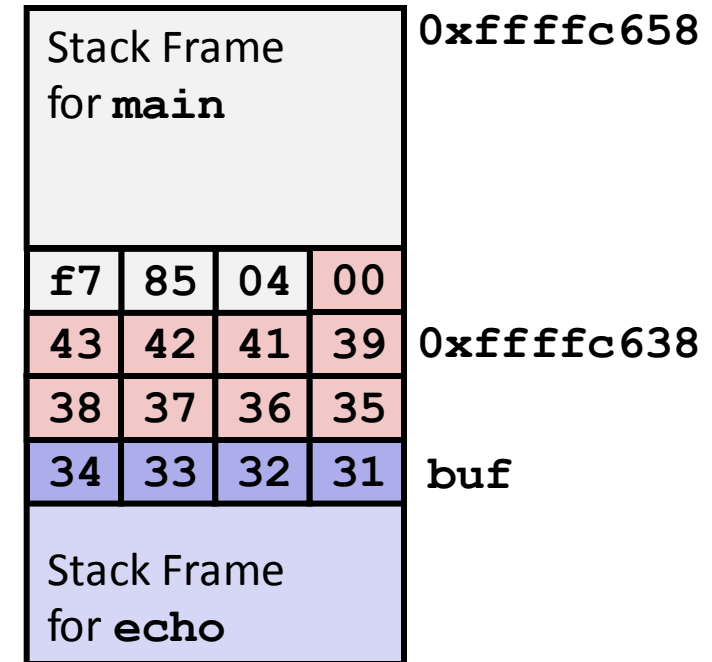
```

Buffer Overflow Example #3

Before call to gets



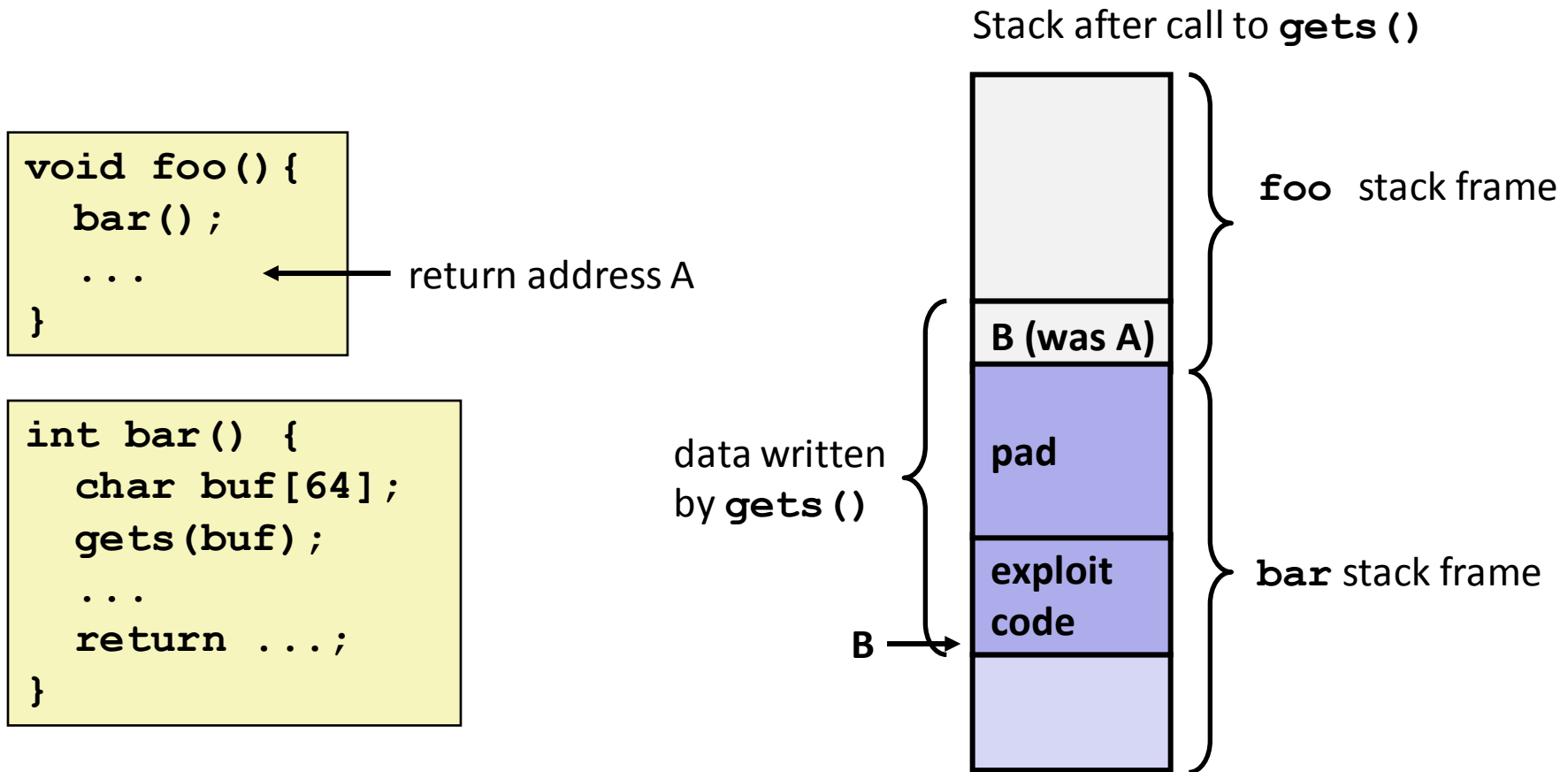
Input 123456789ABC



Return address corrupted

```
80485f2: call 80484f0 <echo>
80485f7: mov 0xffffffffc(%ebp),%ebx # Return Point
```

Malicious Use of Buffer Overflow



- Input string contains byte representation of executable code
- Stack frame must be big enough to hold exploit code
- Overwrite return address with address of buffer (need to know B)
- When `bar()` executes `ret`, will jump to exploit code (instead of A)

Exploits Based on Buffer Overflows

- *Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines*
- **Internet worm**
 - Early versions of the finger server (fingerd) used `gets ()` to read the argument sent by the client:
 - `finger droh@cs.cmu.edu`
 - Worm attacked fingerd server by sending phony argument:
 - `finger "exploit-code padding new-return-address"`
 - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

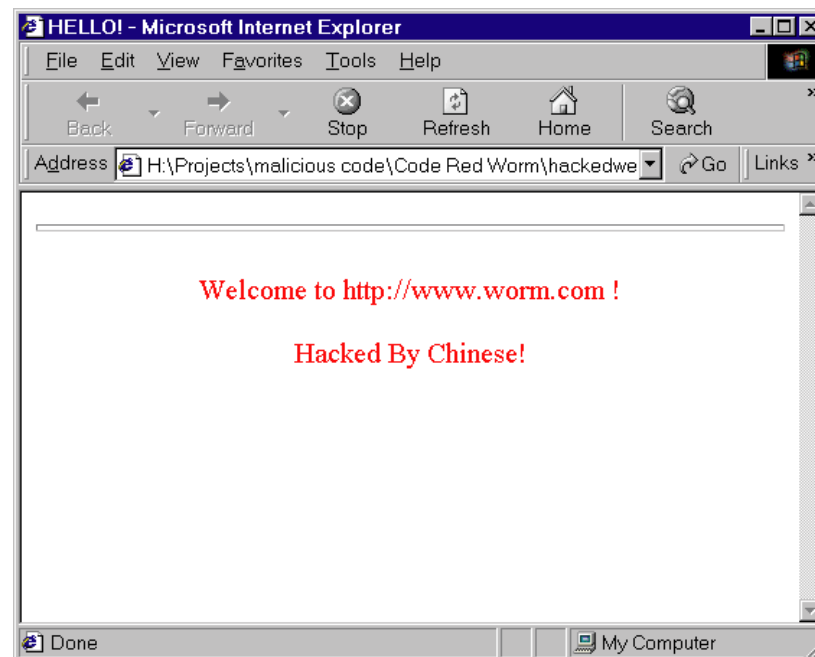
Code Red Worm

■ History

- June 18, 2001. Microsoft announces buffer overflow vulnerability in IIS Internet server
- July 19, 2001. over 250,000 machines infected by new virus in 9 hours
- White house must change its IP address. Pentagon shut down public WWW servers for day

Code Red Exploit Code

- Starts 100 threads running
- Spread self
 - Generate random IP addresses & send attack string
 - Between 1st & 19th of month
- Attack **www.whitehouse.gov**
 - Send 98,304 packets; sleep for 4-1/2 hours; repeat
 - Denial of service attack
 - Between 21st & 27th of month
- Deface server's home page
 - After waiting 2 hours
- Later versions even more aggressive
- And it goes on still...



Avoiding Overflow Vulnerability

```
/* Echo Line */  
void echo()  
{  
    char buf[4]; /* Way too small!  
*/  
    fgets(buf, 4, stdin);  
    puts(buf);  
}
```

■ Use library routines that limit string lengths

- **fgets** instead of **gets** (second argument to fgets sets limit)
- **strncpy** instead of **strcpy**
- Don't use **scanf** with **%s** conversion specification
 - Use **fgets** to read the string
 - Or use **%ns** where **n** is a suitable integer

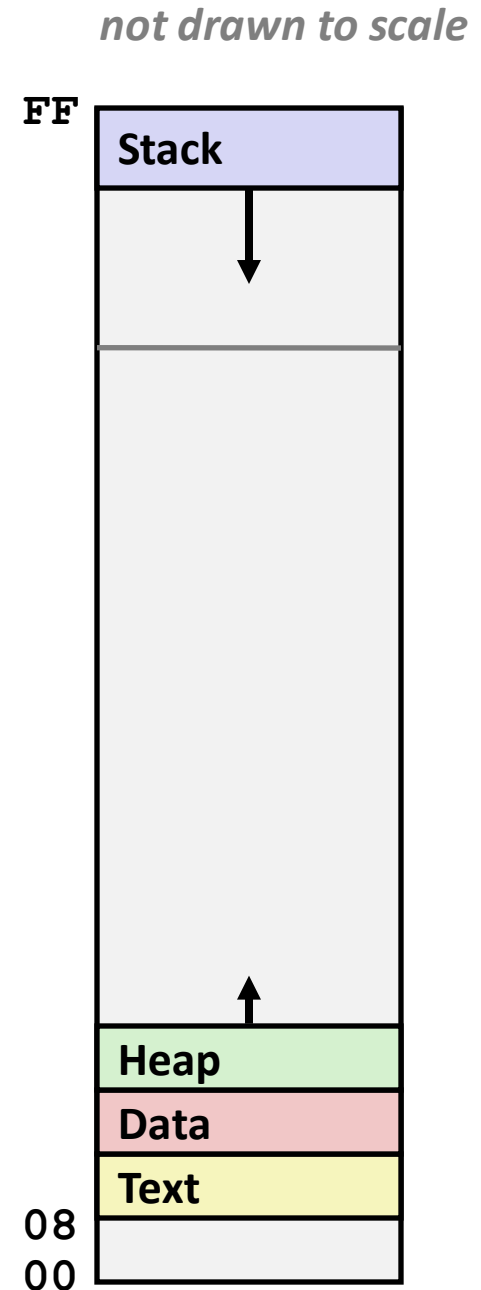
System-Level Protections

■ Randomized stack offsets

- At start of program, allocate random amount of space on stack
- Makes it difficult for hacker to predict beginning of inserted code

■ Nonexecutable code segments

- Only allow code to execute from “text” sections of memory
- Do NOT execute code in stack, data, or heap regions
- Hardware support



Worms and Viruses

- **Worm: A program that**
 - Can run by itself
 - Can propagate a fully working version of itself to other computers

- **Virus: Code that**
 - Adds itself to other programs
 - Cannot run independently

- **Both are (usually) designed to spread among computers and to wreak havoc (and, these days, profit\$\$\$)**