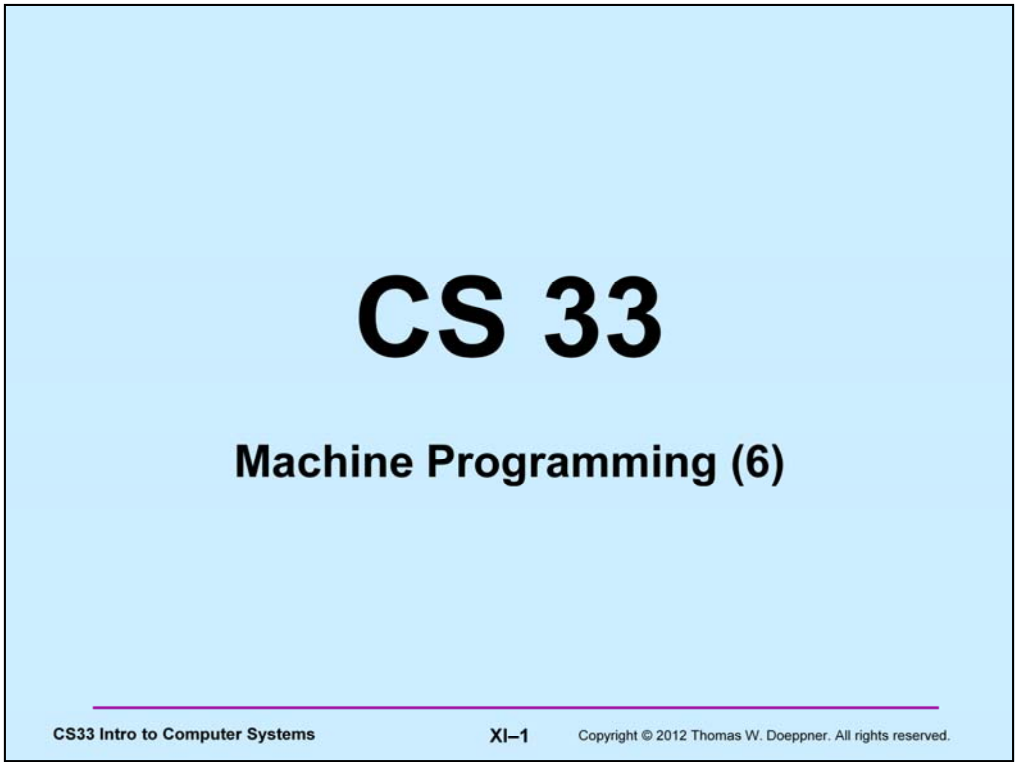
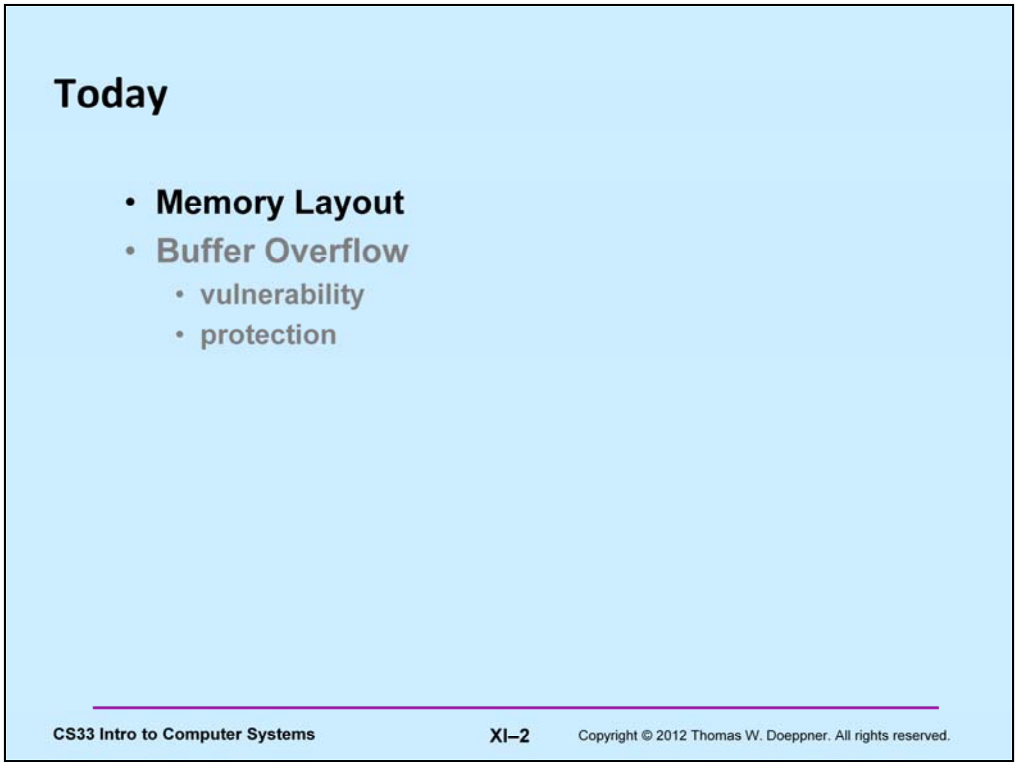


CS 33

Machine Programming (6)

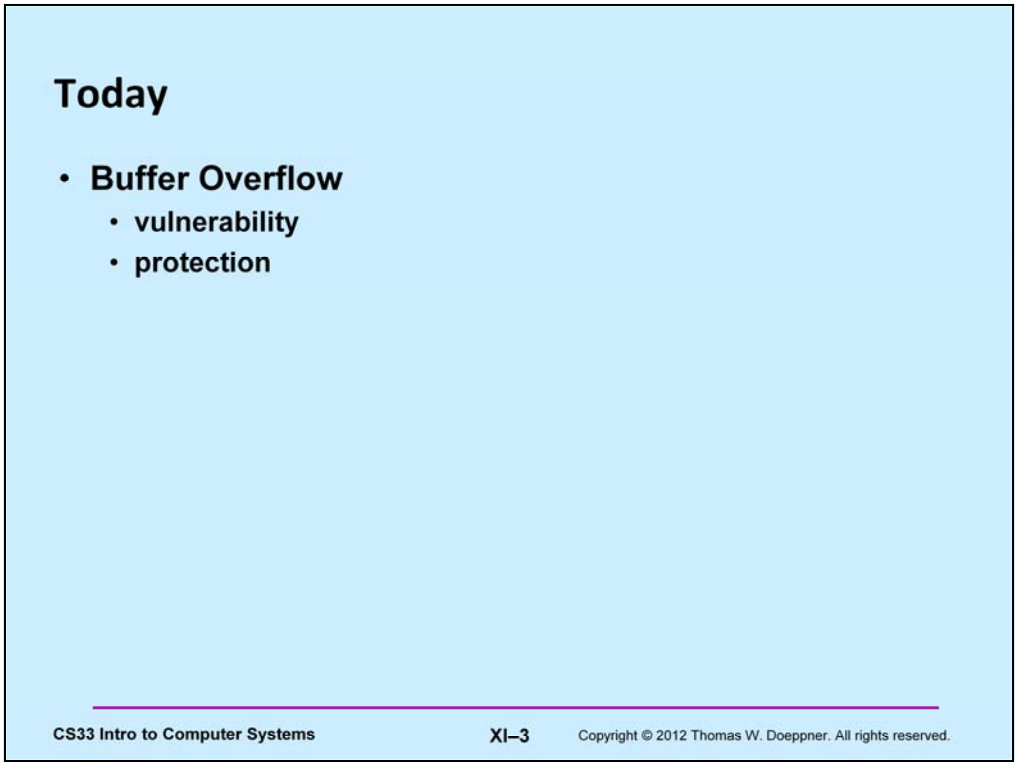
Most of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective,” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.



Today

- **Memory Layout**
- Buffer Overflow
  - vulnerability
  - protection

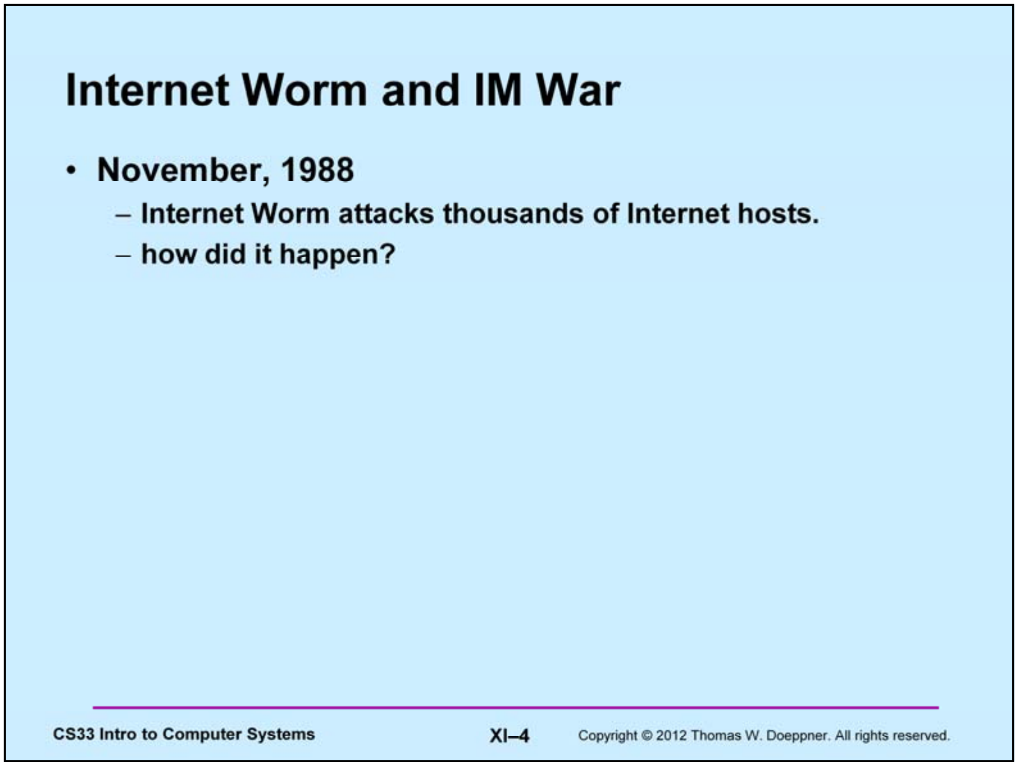
Supplied by CMU.



Today

- Buffer Overflow
  - vulnerability
  - protection

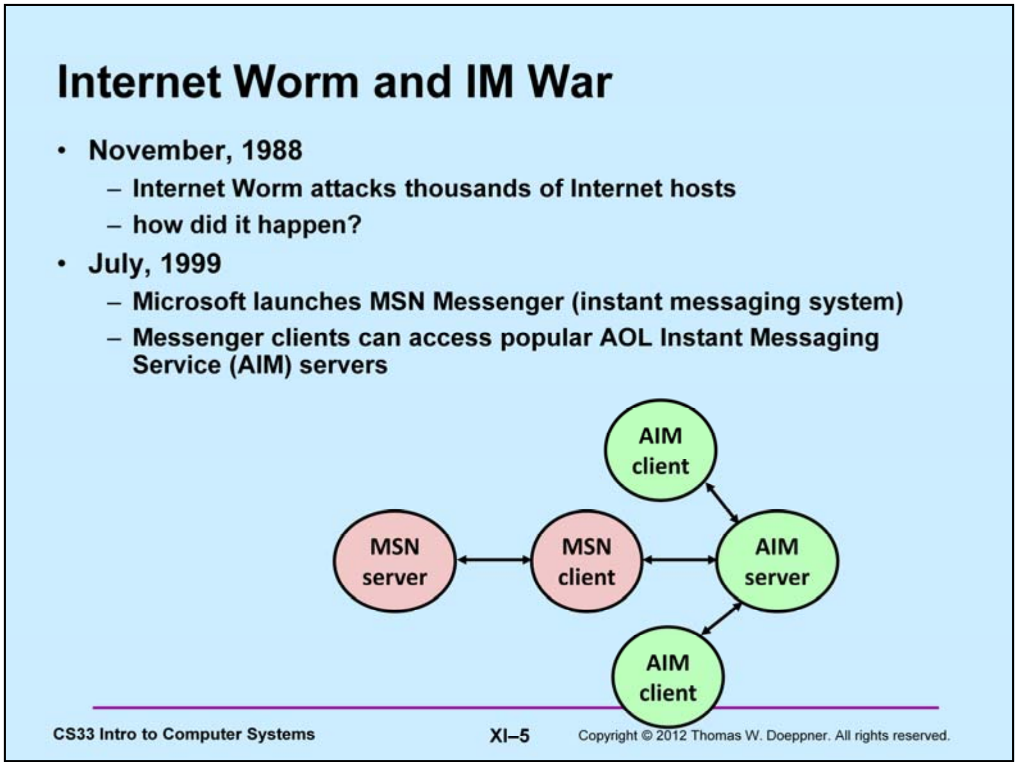
Supplied by CMU.



Internet Worm and IM War

- November, 1988
  - Internet Worm attacks thousands of Internet hosts.
  - how did it happen?

Supplied by CMU.

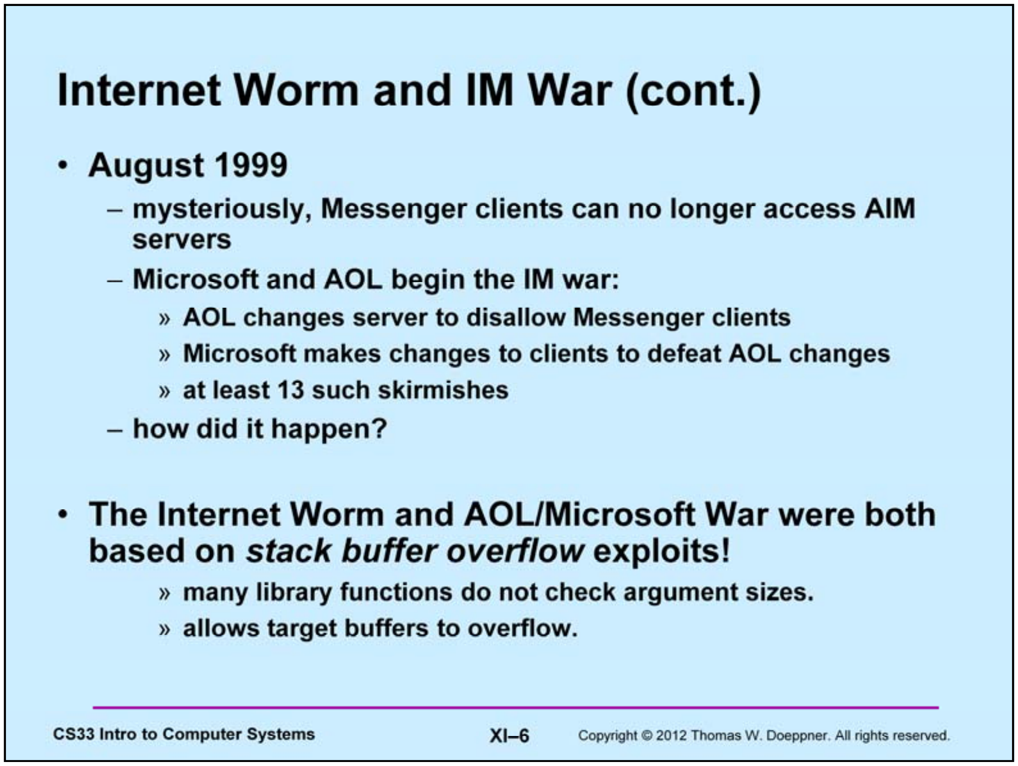


Internet Worm and IM War

- November, 1988
  - Internet Worm attacks thousands of Internet hosts
  - how did it happen?
- July, 1999
  - Microsoft launches MSN Messenger (instant messaging system)
  - Messenger clients can access popular AOL Instant Messaging Service (AIM) servers

AIM client
MSN server
MSN client
AIM server
AIM client

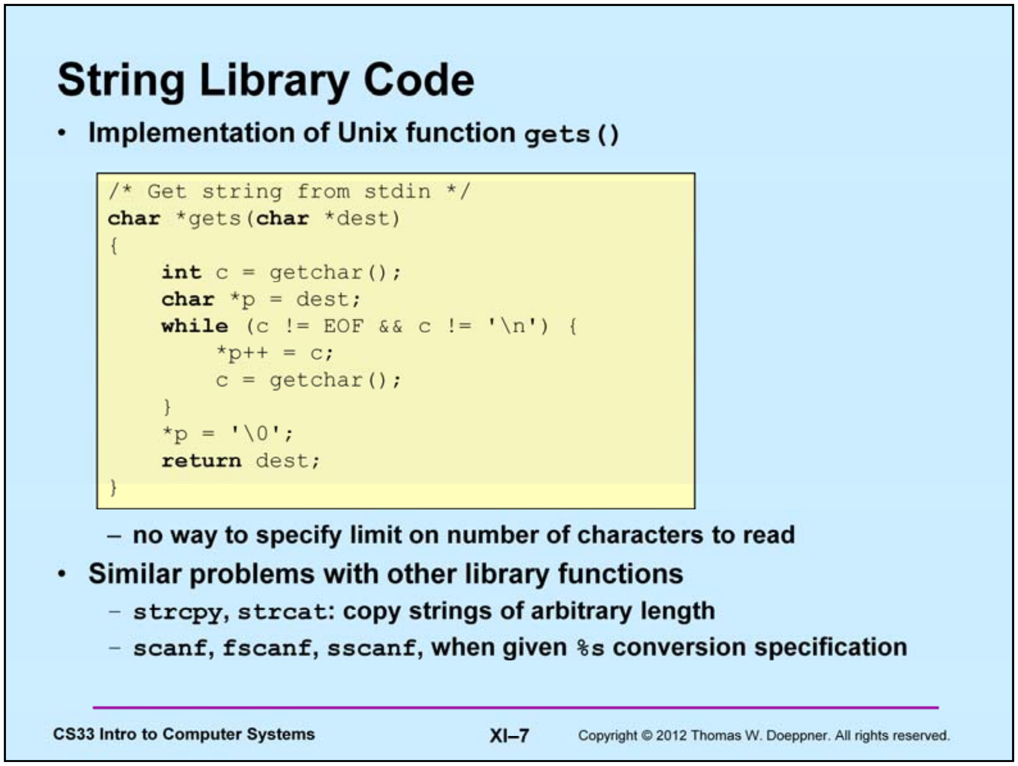
Supplied by CMU.



Internet Worm and IM War (cont.)

- August 1999
  - mysteriously, Messenger clients can no longer access AIM servers
  - Microsoft and AOL begin the IM war:
    - » AOL changes server to disallow Messenger clients
    - » Microsoft makes changes to clients to defeat AOL changes
    - » at least 13 such skirmishes
  - how did it happen?
- The Internet Worm and AOL/Microsoft War were both based on *stack buffer overflow* exploits!
    - » many library functions do not check argument sizes.
    - » allows target buffers to overflow.

Supplied by CMU.



String Library Code

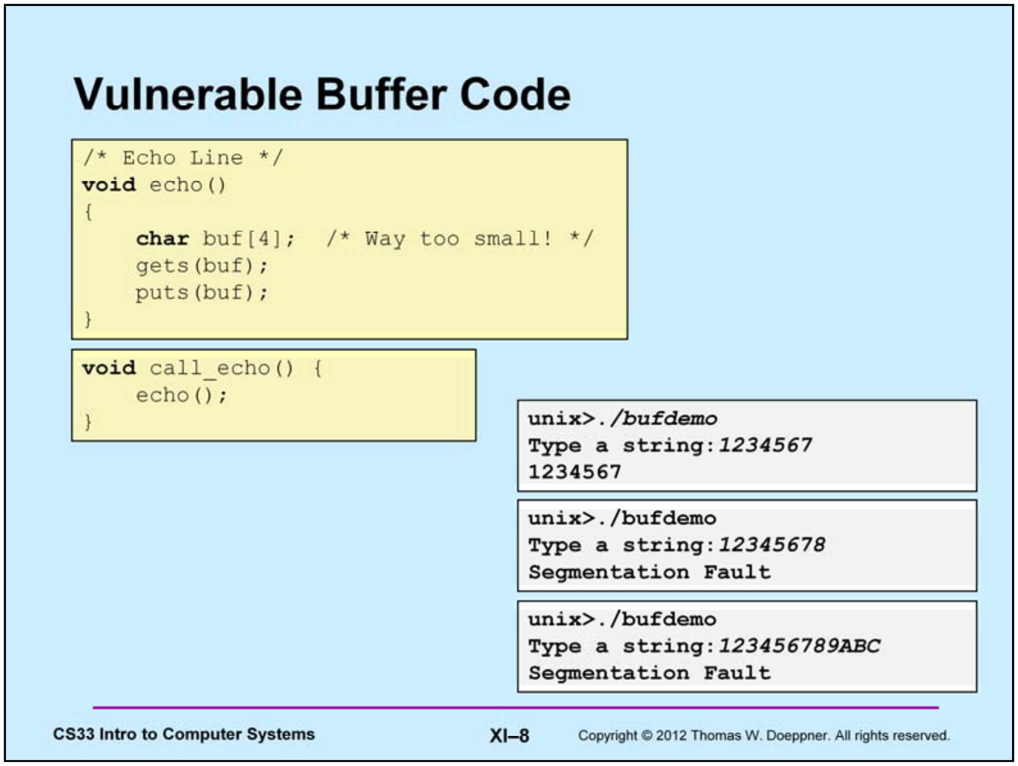
- Implementation of Unix function `gets()`

```
/* 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 library functions
  - `strcpy`, `strcat`: copy strings of arbitrary length
  - `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification

Supplied by CMU.



Vulnerable Buffer Code

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

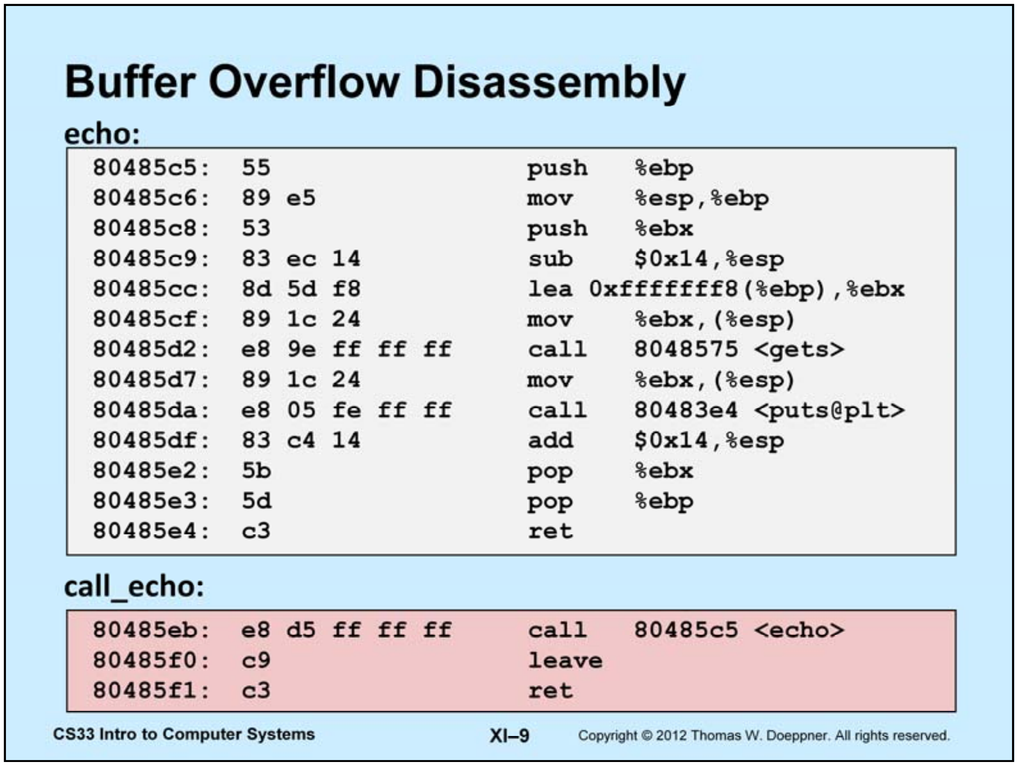
```
void call_echo() {
    echo();
}
```

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

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

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

Supplied by CMU.



Buffer Overflow Disassembly

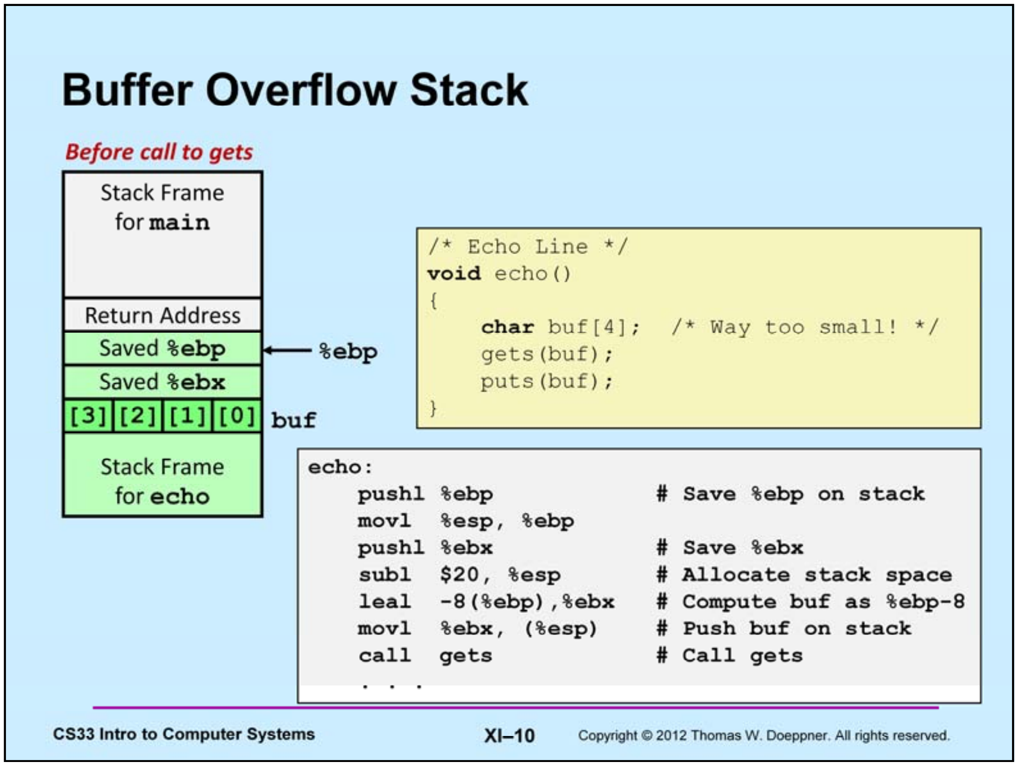
echo:

```
 80485c5:  55                    push   %ebp
 80485c6:  89 e5                 mov    %esp,%ebp
 80485c8:  53                    push   %ebx
 80485c9:  83 ec 14              sub    $0x14,%esp
 80485cc:  8d 5d f8              lea 0xfffffff8(%ebp),%ebx
 80485cf:  89 1c 24              mov    %ebx,(%esp)
 80485d2:  e8 9e ff ff ff        call   8048575 <gets>
 80485d7:  89 1c 24              mov    %ebx,(%esp)
 80485da:  e8 05 fe ff ff        call   80483e4 <puts@plt>
 80485df:  83 c4 14              add    $0x14,%esp
 80485e2:  5b                    pop    %ebx
 80485e3:  5d                    pop    %ebp
 80485e4:  c3                    ret
```

call_echo:

```
 80485eb:  e8 d5 ff ff ff        call   80485c5 <echo>
 80485f0:  c9                    leave
 80485f1:  c3                    ret
```

Supplied by CMU.

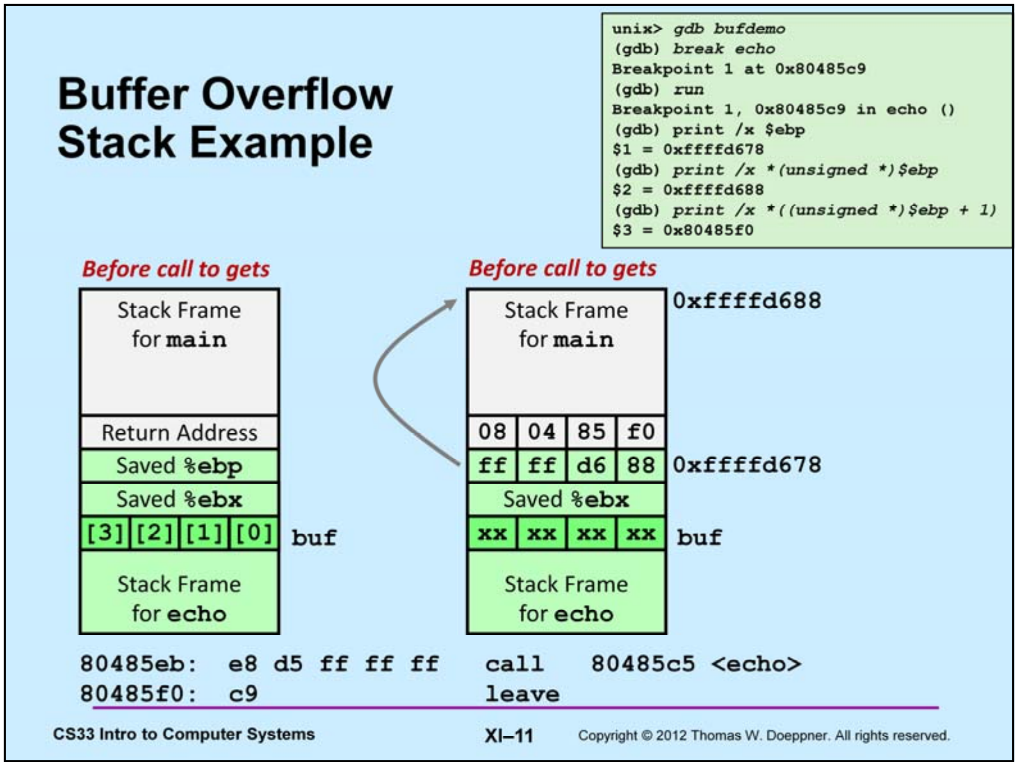


Buffer Overflow Stack

```
/* 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
    subl  $20, %esp         # Allocate stack space
    leal  -8(%ebp),%ebx     # Compute buf as %ebp-8
    movl  %ebx, (%esp)      # Push buf on stack
    call  gets              # Call gets
    . . .
```

Supplied by CMU.



Buffer Overflow Stack Example

```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x80485c9
(gdb) run
Breakpoint 1, 0x80485c9 in echo ()
(gdb) print /x $ebp
$1 = 0xffffd678
(gdb) print /x *(unsigned *)$ebp
$2 = 0xffffd688
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x80485f0
```

Before call to gets

Stack Frame for **main**

Return Address

Saved **%ebp**

Saved **%ebx**

[3] [2] [1] [0] **buf**

Stack Frame for **echo**

Before call to gets

Stack Frame for **main** **0xffffd688**

08 04 85 f0

ff ff d6 88 **0xffffd678**

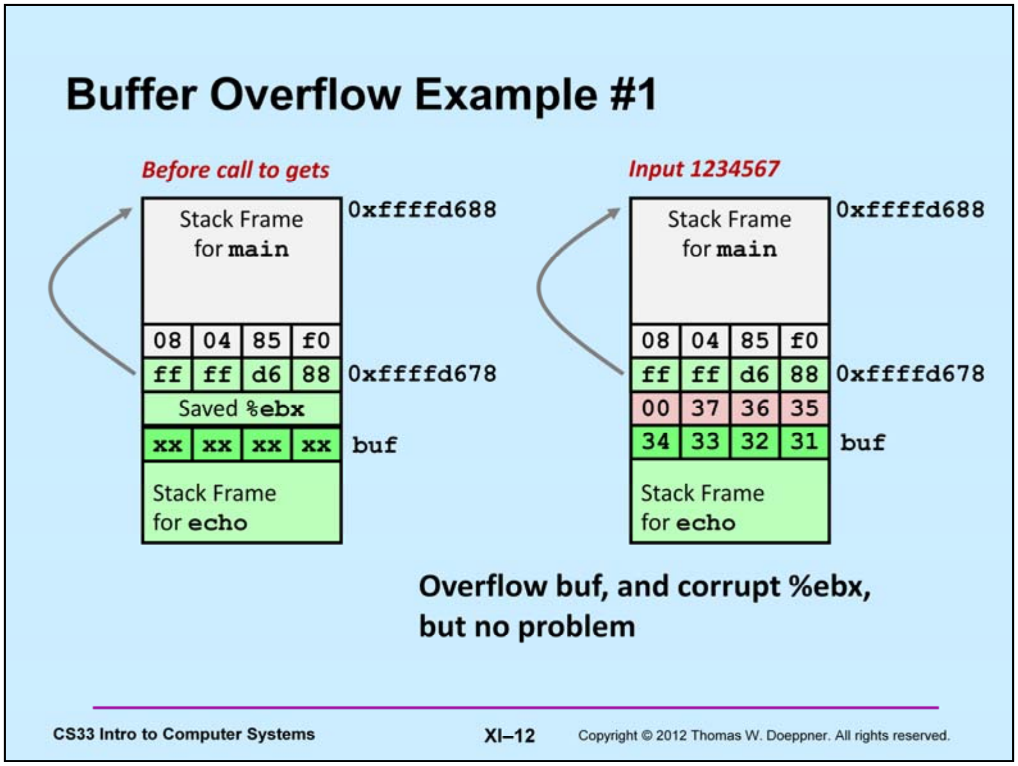
Saved **%ebx**

xx xx xx xx **buf**

Stack Frame for **echo**

```
80485eb:  e8 d5 ff ff ff    call   80485c5 <echo>
80485f0:  c9                leave
```

Supplied by CMU.



Buffer Overflow Example #1

Before call to gets

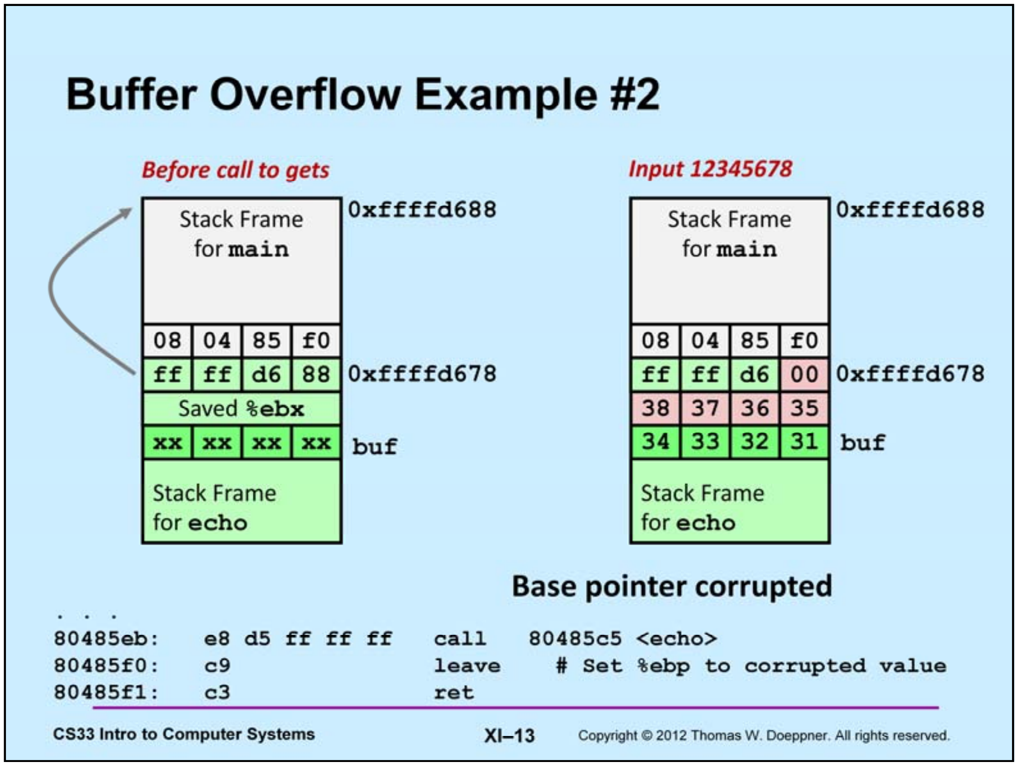
Stack Frame for **main**				0xffffd688
08	04	85	f0	
ff	ff	d6	88	0xffffd678
Saved %ebx				
xx	xx	xx	xx	buf
Stack Frame for **echo**				

Input 1234567

Stack Frame for **main**				0xffffd688
08	04	85	f0	
ff	ff	d6	88	0xffffd678
00	37	36	35	
34	33	32	31	buf
Stack Frame for **echo**				

Overflow buf, and corrupt %ebx, but no problem

Supplied by CMU.



Buffer Overflow Example #2

Before call to gets

Stack Frame for **main**				0xffffd688
08	04	85	f0	
ff	ff	d6	88	0xffffd678
Saved %ebx				
xx	xx	xx	xx	buf
Stack Frame for **echo**				

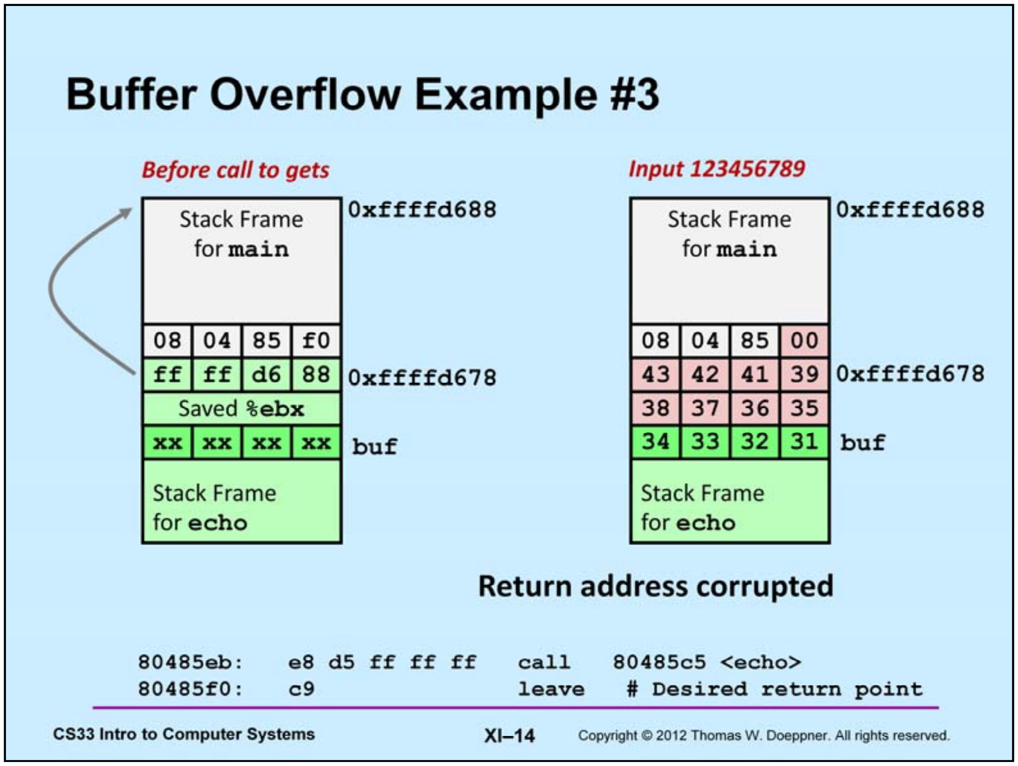
Input 12345678

Stack Frame for **main**				0xffffd688
08	04	85	f0	
ff	ff	d6	00	0xffffd678
38	37	36	35	
34	33	32	31	buf
Stack Frame for **echo**				

Base pointer corrupted

```
. . .
80485eb:    e8 d5 ff ff ff    call   80485c5 <echo>
80485f0:    c9                leave    # Set %ebp to corrupted value
80485f1:    c3                ret
```

Supplied by CMU.



Buffer Overflow Example #3

Before call to gets

Stack Frame for **main** 0xffffd688

08 04 85 f0

ff ff d6 88 0xffffd678

Saved %ebx

xx xx xx xx buf

Stack Frame for **echo**

Input 123456789

Stack Frame for **main** 0xffffd688

08 04 85 00

43 42 41 39 0xffffd678

38 37 36 35

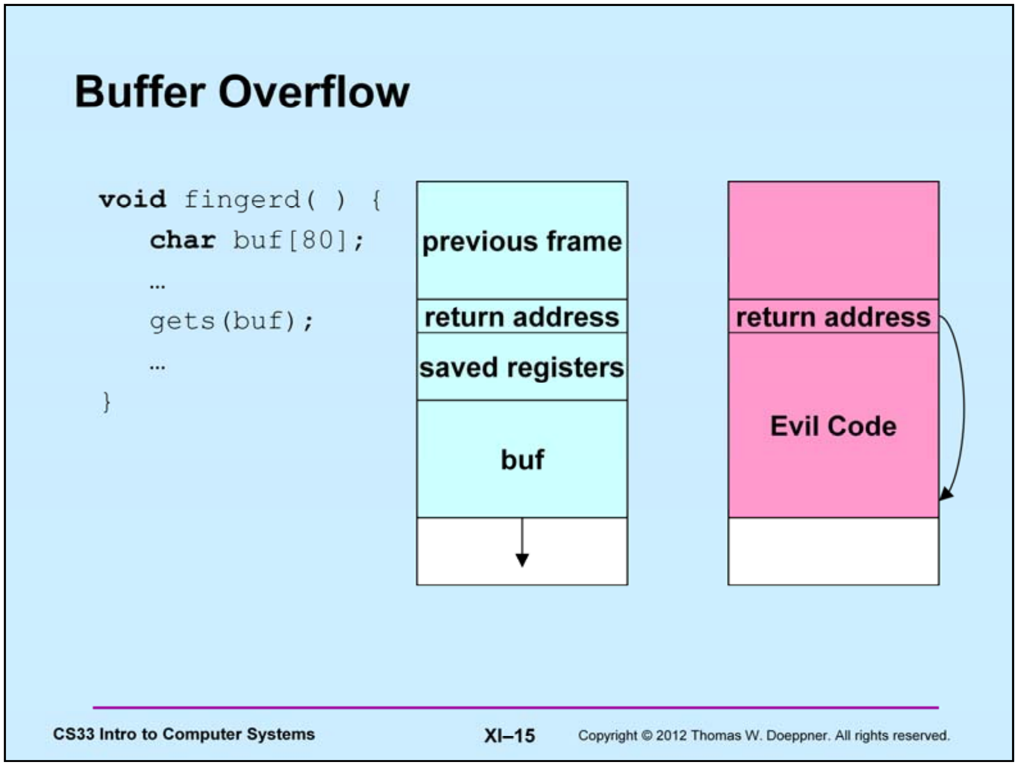
34 33 32 31 buf

Stack Frame for **echo**

Return address corrupted

```
80485eb:    e8 d5 ff ff ff    call    80485c5 <echo>
80485f0:    c9                leave    # Desired return point
```

Supplied by CMU.



Buffer Overflow

```
void fingerd( ) {
    char buf[80];
    …
    gets(buf);
    …
}
```

previous frame

return address

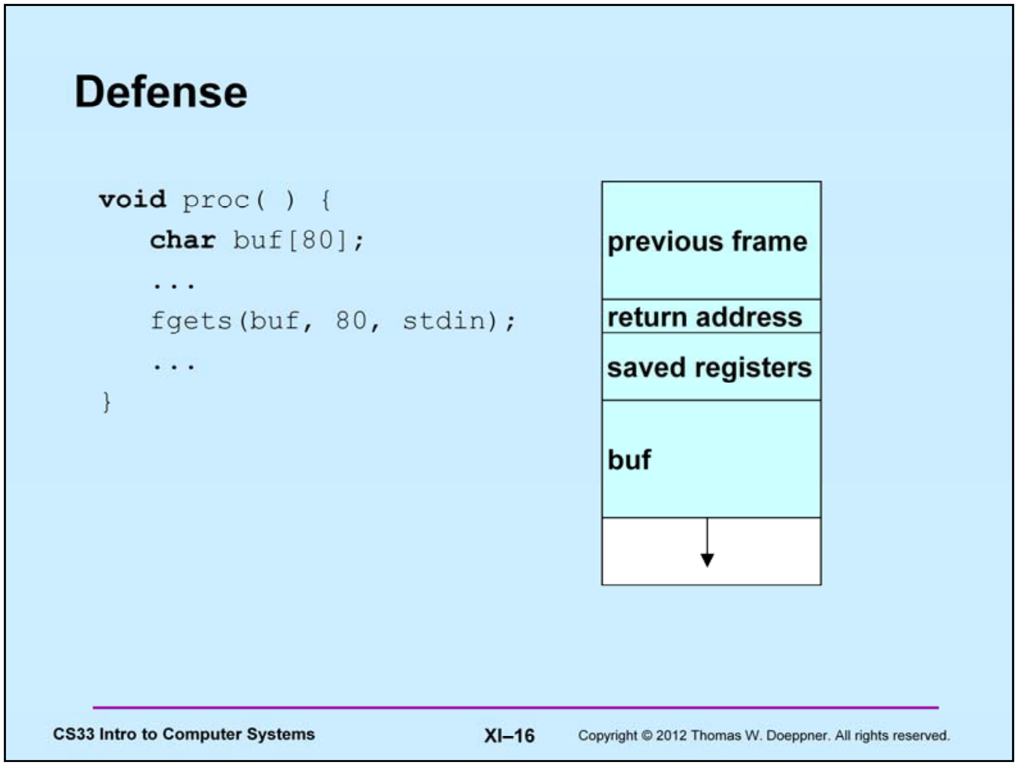
saved registers

buf

return address

Evil Code

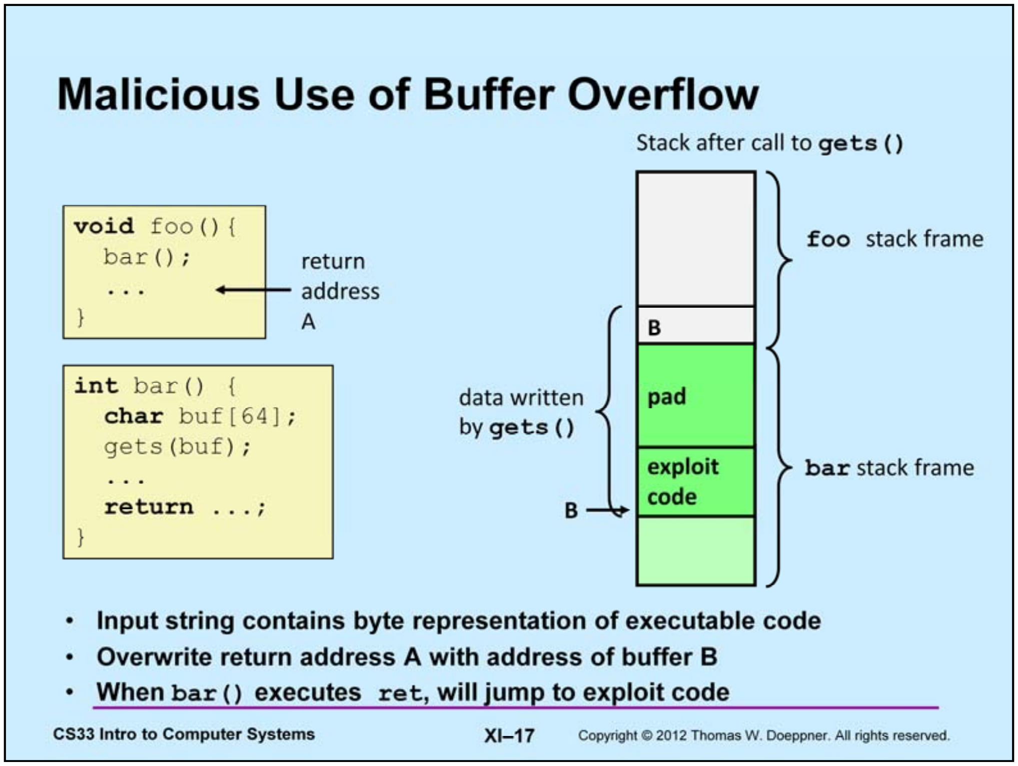
Programs susceptible to buffer-overflow attacks are amazingly common and thus such attacks are probably the most common of the bug-exploitation techniques. Even drivers for network interface devices have such problems, making machines vulnerable to attacks by maliciously created packets.



Defense

```
void proc( ) {
   char buf[80];
   ...
   fgets(buf, 80, stdin);
   ...
}
```

previous frame
return address
saved registers
buf



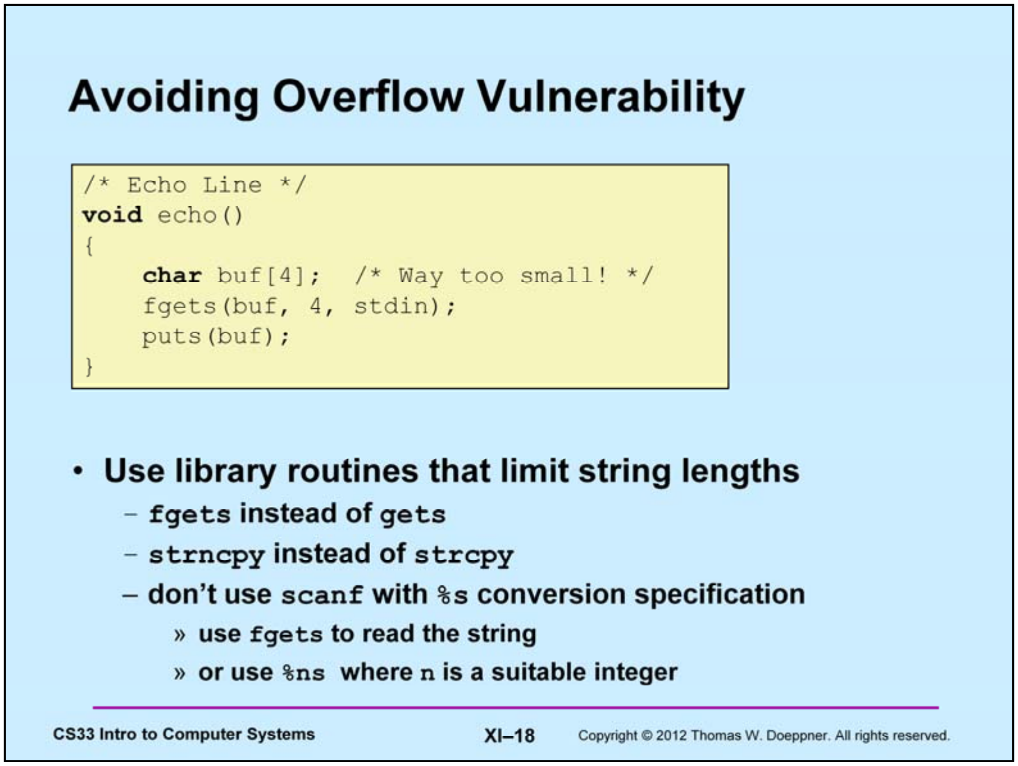
Malicious Use of Buffer Overflow

```
void foo(){
  bar();
  ...
}
```

```
int bar() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When bar() executes ret, will jump to exploit code

Supplied by CMU.

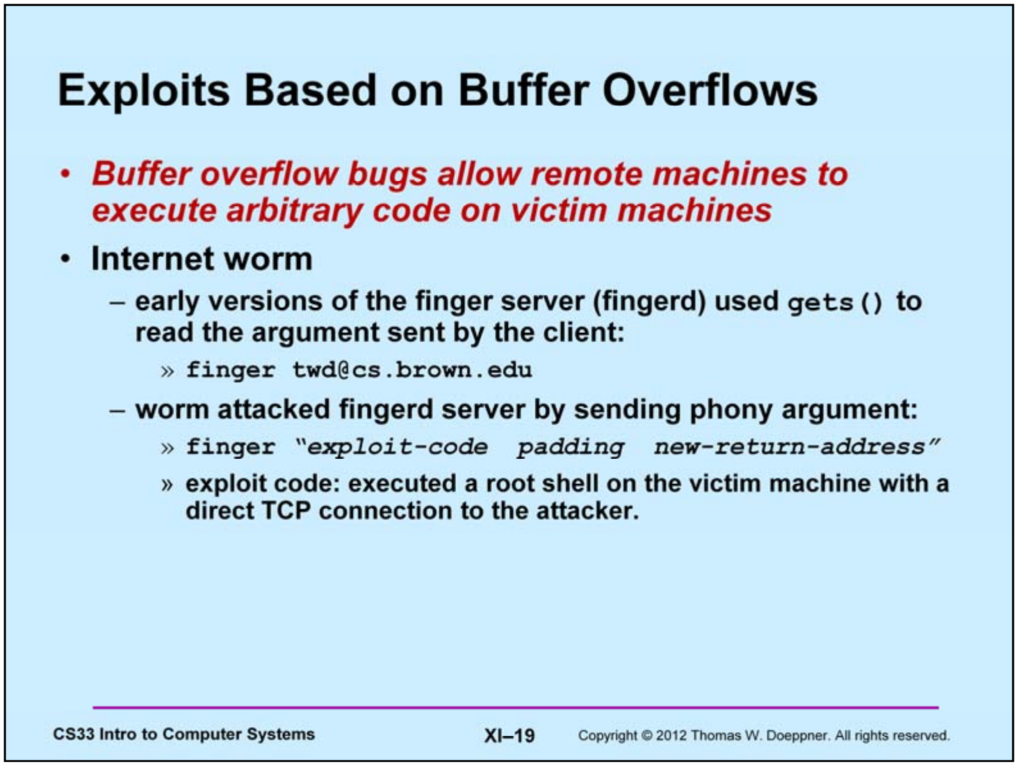


Avoiding Overflow Vulnerability

```
/* 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`
  - `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

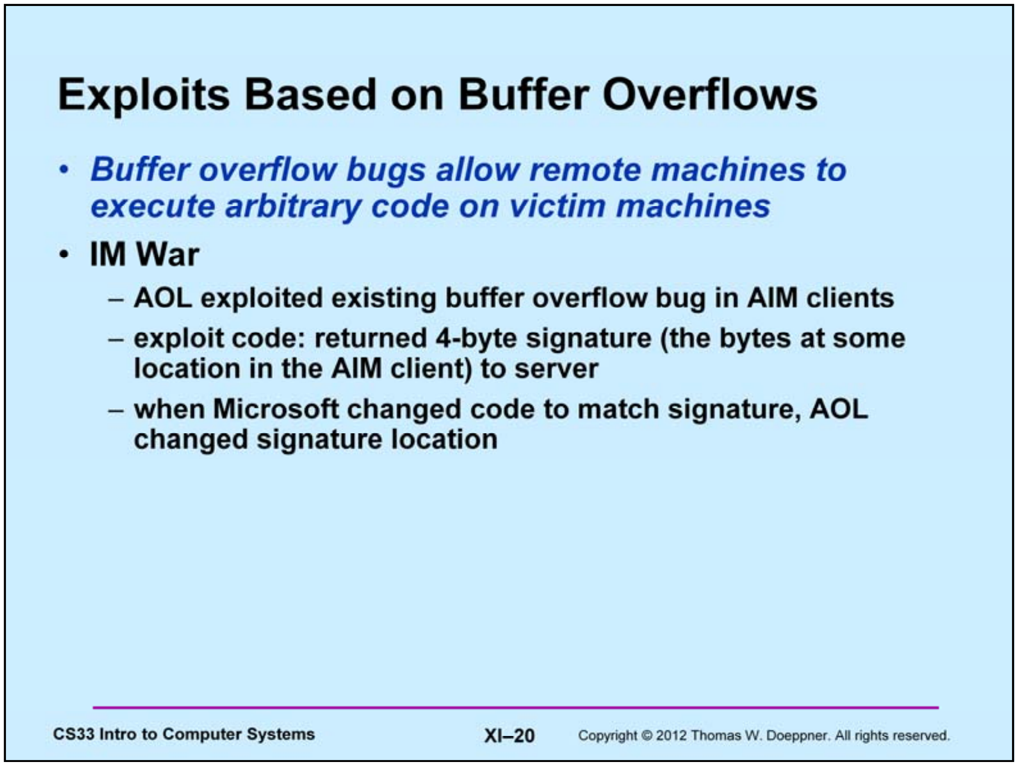
Supplied by CMU.



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 twd@cs.brown.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.**

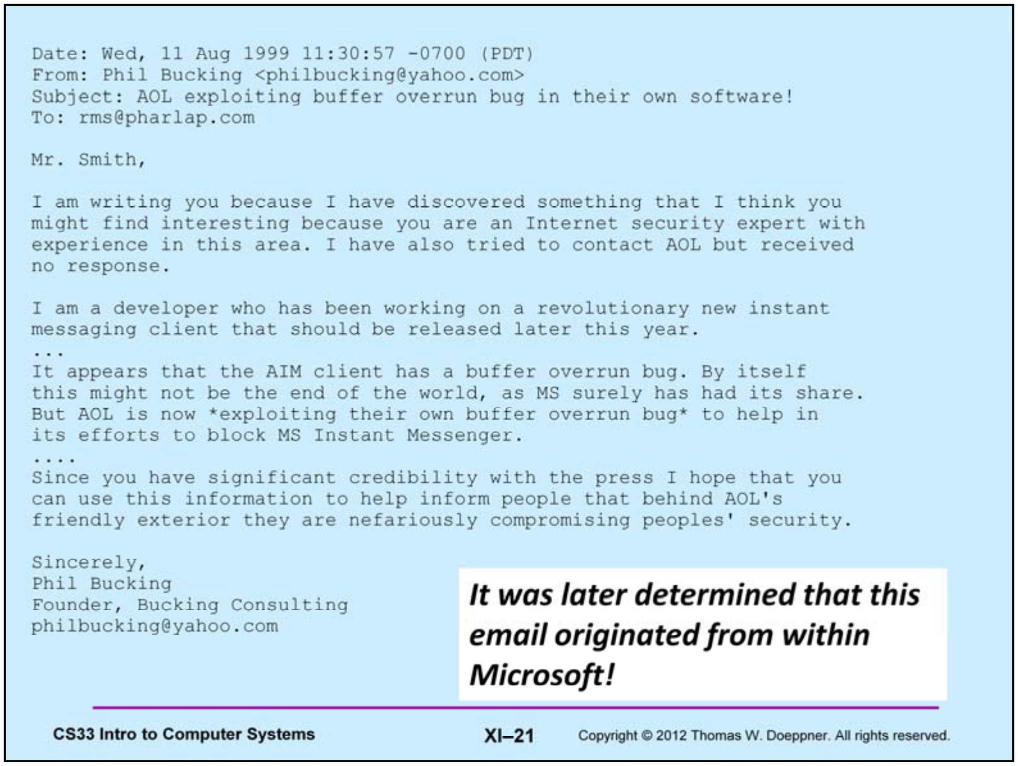
Supplied by CMU.



Exploits Based on Buffer Overflows

- ***Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines***
- **IM War**
  - **AOL exploited existing buffer overflow bug in AIM clients**
  - **exploit code: returned 4-byte signature (the bytes at some location in the AIM client) to server**
  - **when Microsoft changed code to match signature, AOL changed signature location**

Supplied by CMU.



Date: Wed, 11 Aug 1999 11:30:57 -0700 (PDT)
From: Phil Bucking <philbucking@yahoo.com>
Subject: AOL exploiting buffer overrun bug in their own software!
To: rms@pharlap.com

Mr. Smith,

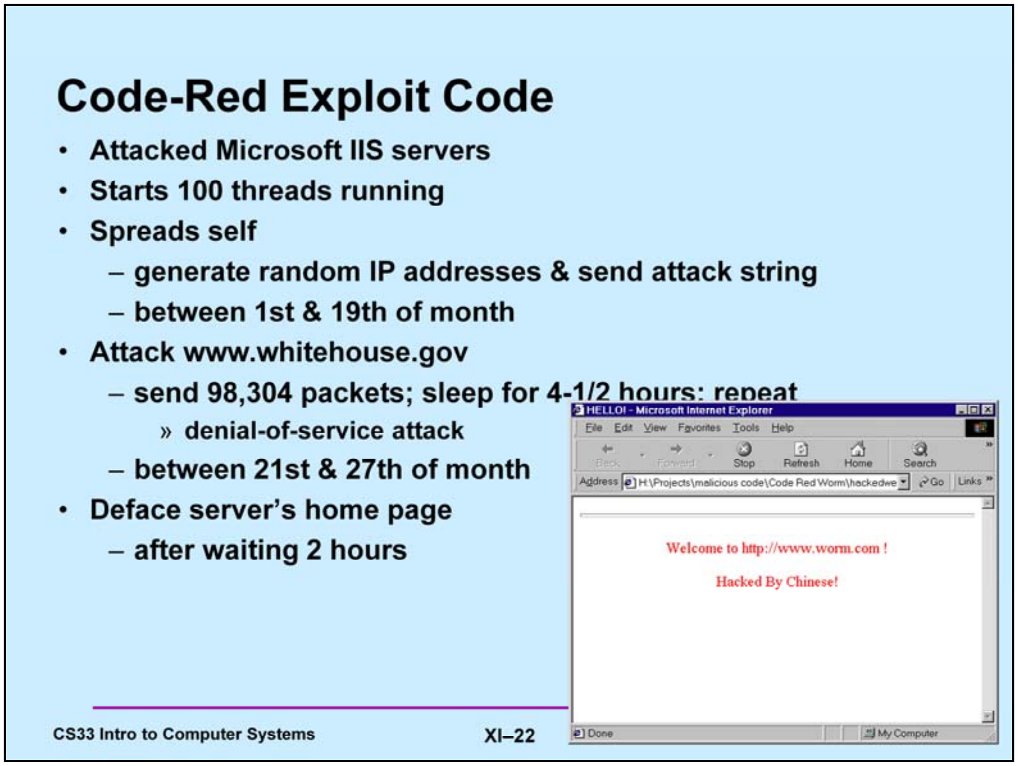
I am writing you because I have discovered something that I think you might find interesting because you are an Internet security expert with experience in this area. I have also tried to contact AOL but received no response.

I am a developer who has been working on a revolutionary new instant messaging client that should be released later this year.
...
It appears that the AIM client has a buffer overrun bug. By itself this might not be the end of the world, as MS surely has had its share. But AOL is now *exploiting their own buffer overrun bug* to help in its efforts to block MS Instant Messenger.
....
Since you have significant credibility with the press I hope that you can use this information to help inform people that behind AOL's friendly exterior they are nefariously compromising peoples' security.

Sincerely,
Phil Bucking
Founder, Bucking Consulting
philbucking@yahoo.com

It was later determined that this email originated from within Microsoft!

Supplied by CMU.



Code-Red Exploit Code

- Attacked Microsoft IIS servers
- Starts 100 threads running
- Spreads 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

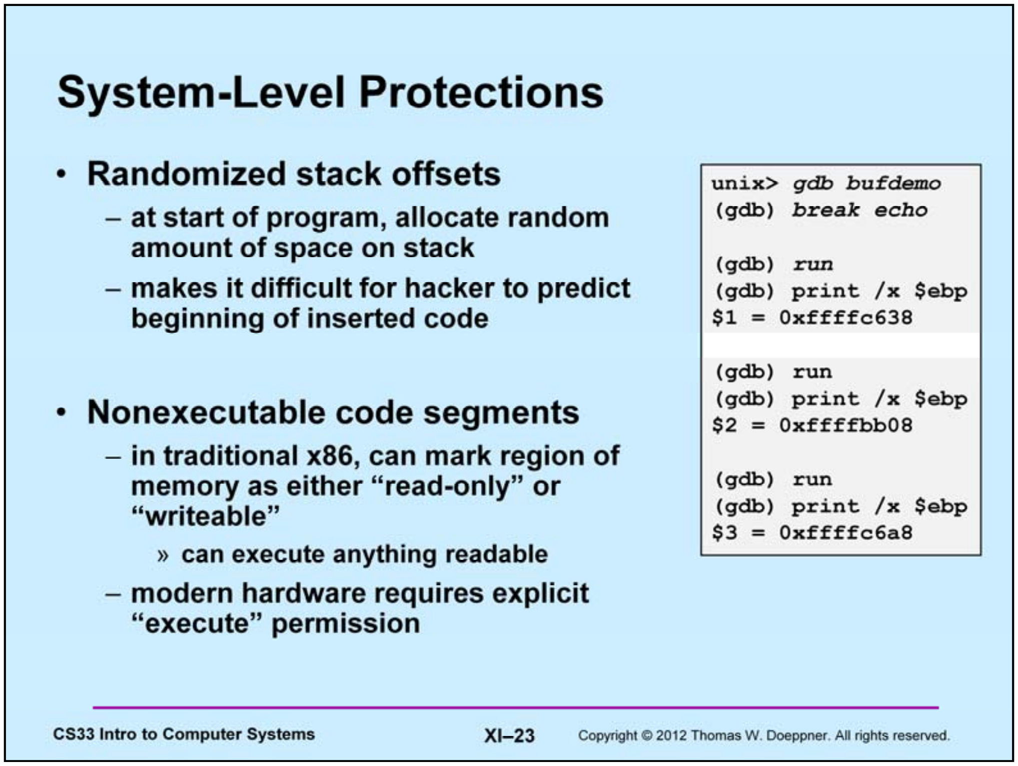
HELLO! - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Forward
Stop
Refresh
Home
Search
Address
H:\Projects\malicious code\Code Red Worm\hackedwe
Go
Links
Done
My Computer

Welcome to http://www.worm.com !

Hacked By Chinese!

Supplied by CMU.

A detailed description of the exploit can be found at http://www.eeye.com/Resources/Security-Center/Research/Security-Advisories/AL20010717.



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
  - in traditional x86, can mark region of memory as either “read-only” or “writeable”
    - » can execute anything readable
  - modern hardware requires explicit “execute” permission

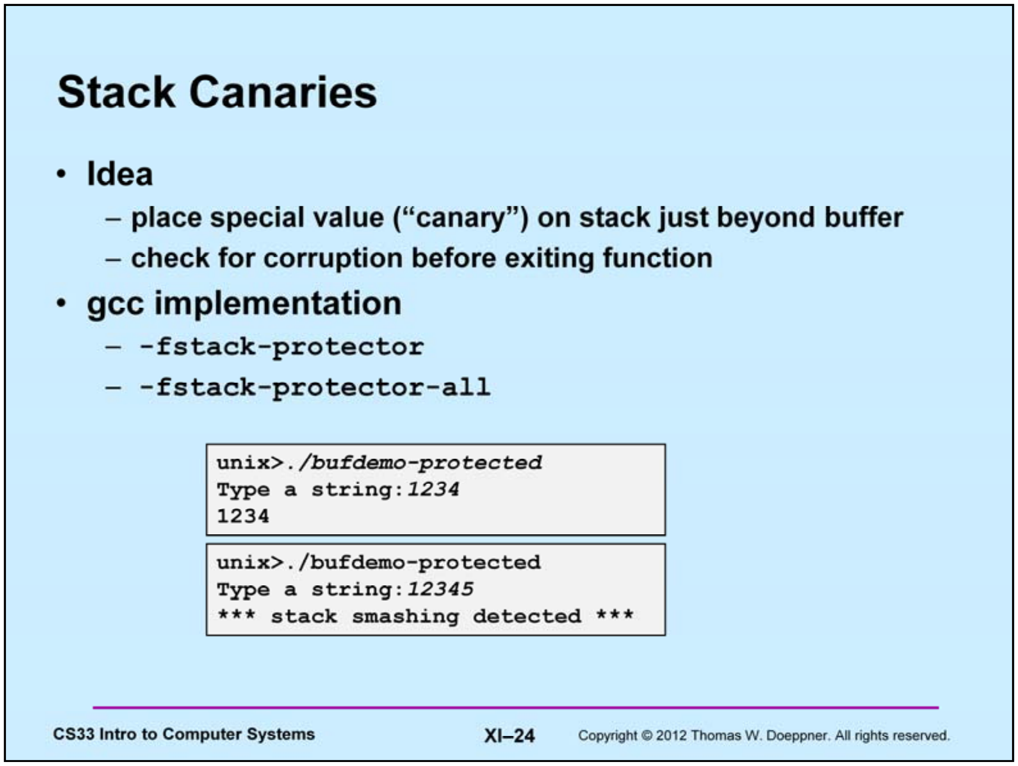
```
unix> gdb bufdemo
(gdb) break echo

(gdb) run
(gdb) print /x $ebp
$1 = 0xffffc638

(gdb) run
(gdb) print /x $ebp
$2 = 0xffffbb08

(gdb) run
(gdb) print /x $ebp
$3 = 0xffffc6a8
```

Supplied by CMU.



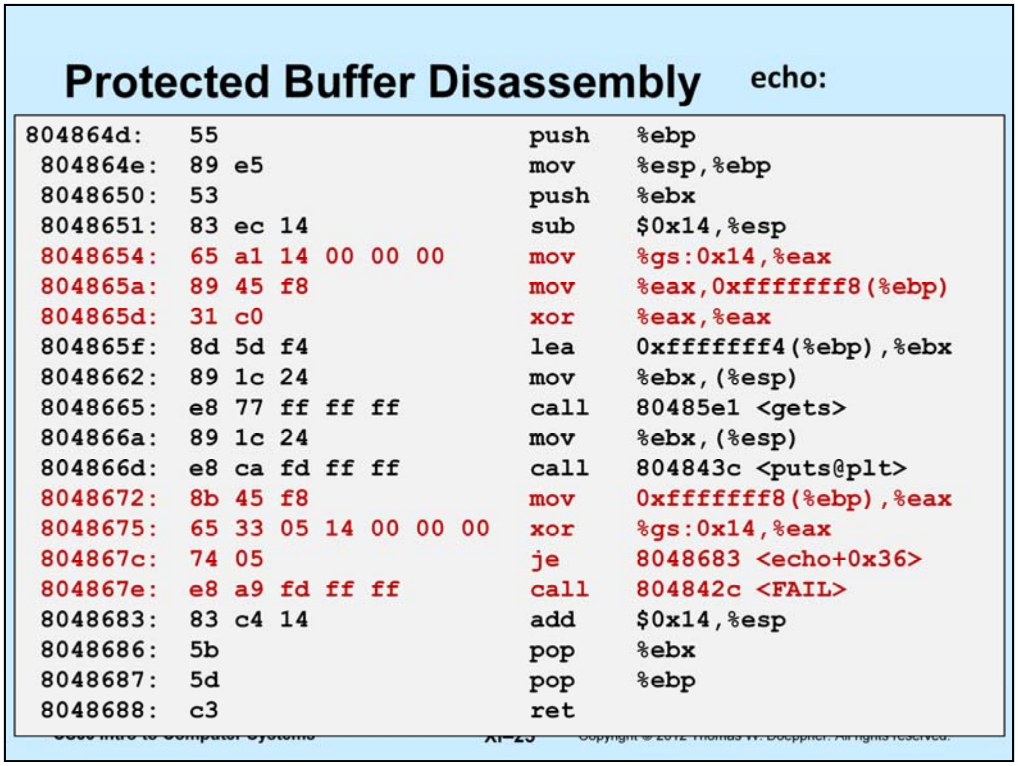
Stack Canaries

- Idea
  - place special value (“canary”) on stack just beyond buffer
  - check for corruption before exiting function
- gcc implementation
  - `-fstack-protector`
  - `-fstack-protector-all`

```
unix>./bufdemo-protected
Type a string:1234
1234
```

```
unix>./bufdemo-protected
Type a string:12345
*** stack smashing detected ***
```

Supplied by CMU.



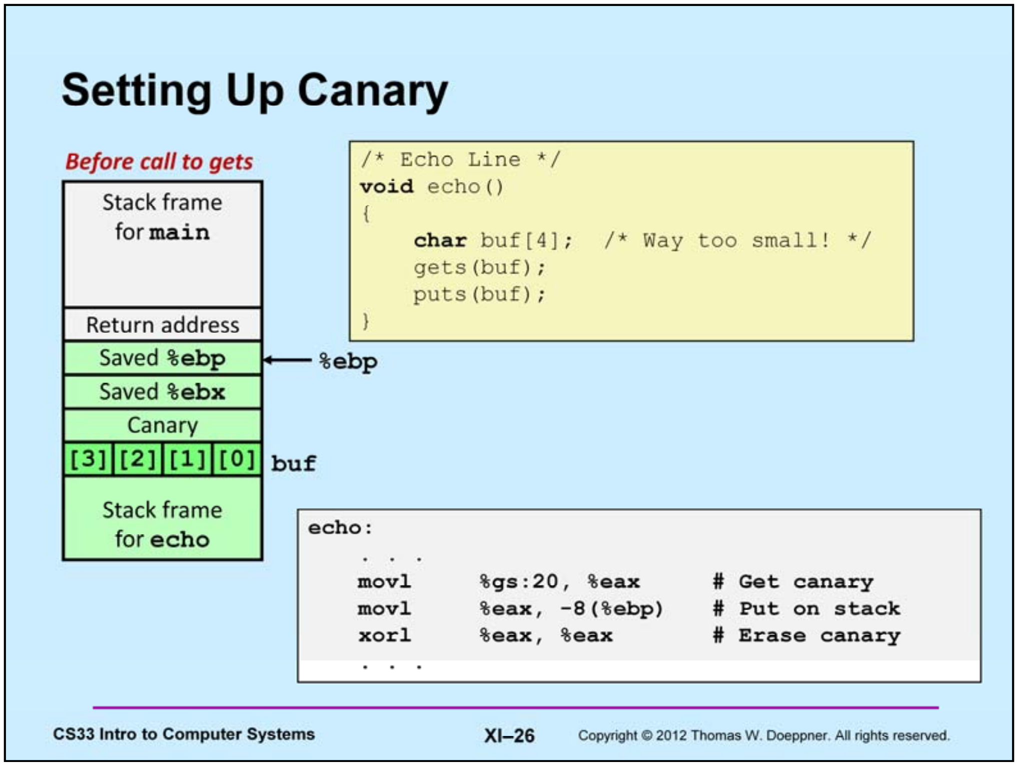
Protected Buffer Disassembly

echo:

```
804864d:   55                      push   %ebp
 804864e:  89 e5                   mov    %esp,%ebp
 8048650:  53                      push   %ebx
 8048651:  83 ec 14                sub    $0x14,%esp
 8048654:  65 a1 14 00 00 00       mov    %gs:0x14,%eax
 804865a:  89 45 f8                mov    %eax,0xfffffff8(%ebp)
 804865d:  31 c0                   xor    %eax,%eax
 804865f:  8d 5d f4                lea    0xfffffff4(%ebp),%ebx
 8048662:  89 1c 24                mov    %ebx,(%esp)
 8048665:  e8 77 ff ff ff          call   80485e1 <gets>
 804866a:  89 1c 24                mov    %ebx,(%esp)
 804866d:  e8 ca fd ff ff          call   804843c <puts@plt>
 8048672:  8b 45 f8                mov    0xfffffff8(%ebp),%eax
 8048675:  65 33 05 14 00 00 00    xor    %gs:0x14,%eax
 804867c:  74 05                   je     8048683 <echo+0x36>
 804867e:  e8 a9 fd ff ff          call   804842c <FAIL>
 8048683:  83 c4 14                add    $0x14,%esp
 8048686:  5b                      pop    %ebx
 8048687:  5d                      pop    %ebp
 8048688:  c3                      ret
```

Supplied by CMU.

The operand “%gs:0x14” requires some explanation, as it uses features we haven’t previously discussed. gs is one of a few “segment registers,” which refer to other areas of memory. They are generally not used, being a relic of the early days of the x86 architecture before virtual-memory support was added. You can think of it as an area where global variables (accessible from anywhere) may be stored and made read-only. It’s used here to store the “canary” values.



Setting Up Canary

Before call to gets

Stack frame for **main**

Return address

Saved **%ebp** ← **%ebp**

Saved **%ebx**

Canary

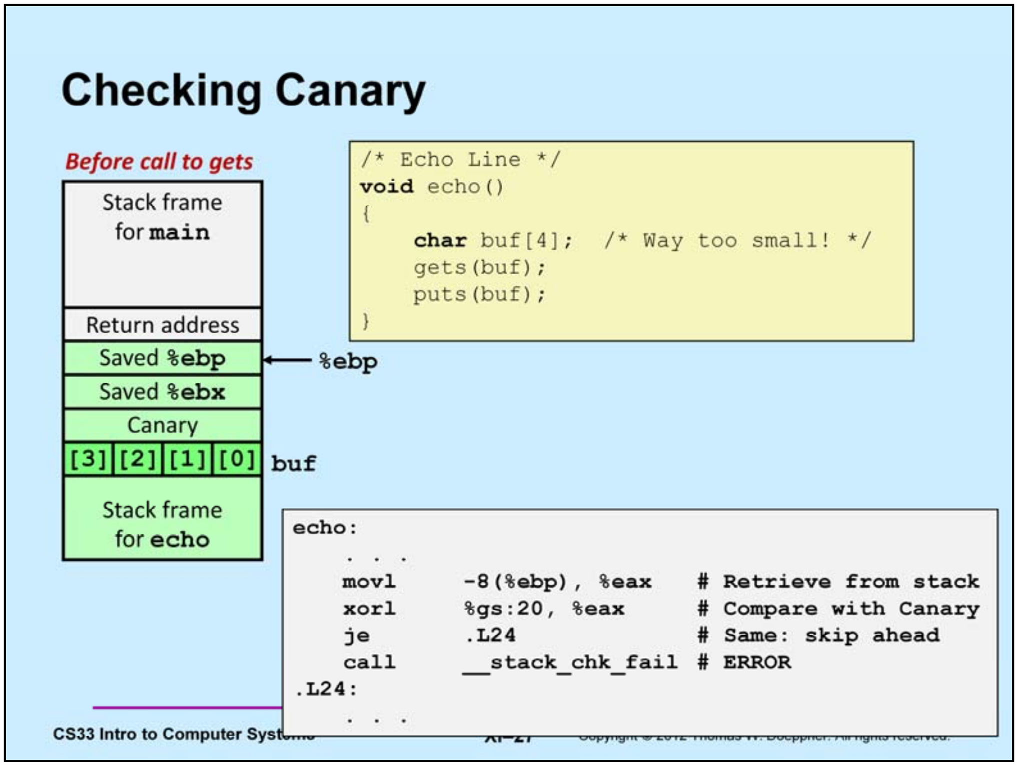
[3] [2] [1] [0] **buf**

Stack frame for **echo**

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

```
echo:
    . . .
    movl    %gs:20, %eax      # Get canary
    movl    %eax, -8(%ebp)    # Put on stack
    xorl    %eax, %eax        # Erase canary
    . . .
```

Supplied by CMU.



Checking Canary

Before call to gets

Stack frame for **main**

Return address

Saved **%ebp** ← **%ebp**

Saved **%ebx**

Canary

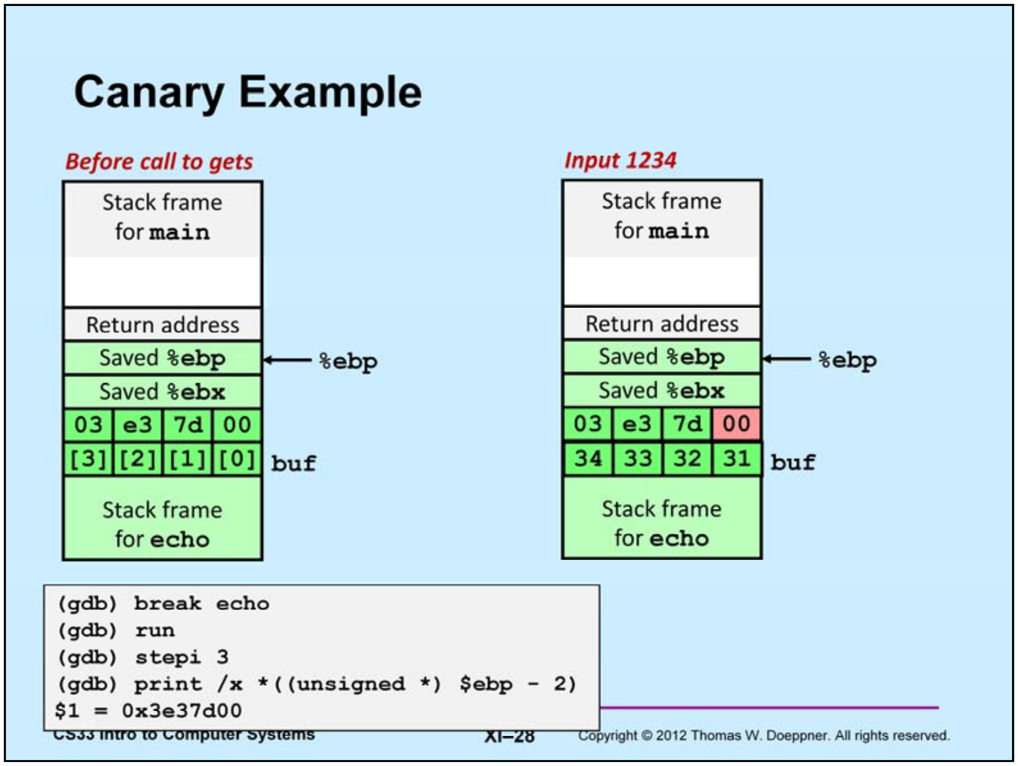
[3] [2] [1] [0] **buf**

Stack frame for **echo**

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

```
echo:
    . . .
    movl     -8(%ebp), %eax    # Retrieve from stack
    xorl     %gs:20, %eax      # Compare with Canary
    je       .L24              # Same: skip ahead
    call     __stack_chk_fail  # ERROR
.L24:
    . . .
```

Supplied by CMU.



Canary Example

Before call to gets

Stack frame for **main**

Return address

Saved %**ebp** ← %**ebp**

Saved %**ebx**

03 | e3 | 7d | 00

[3] | [2] | [1] | [0] **buf**

Stack frame for **echo**

Input 1234

Stack frame for **main**

Return address

Saved %**ebp** ← %**ebp**

Saved %**ebx**

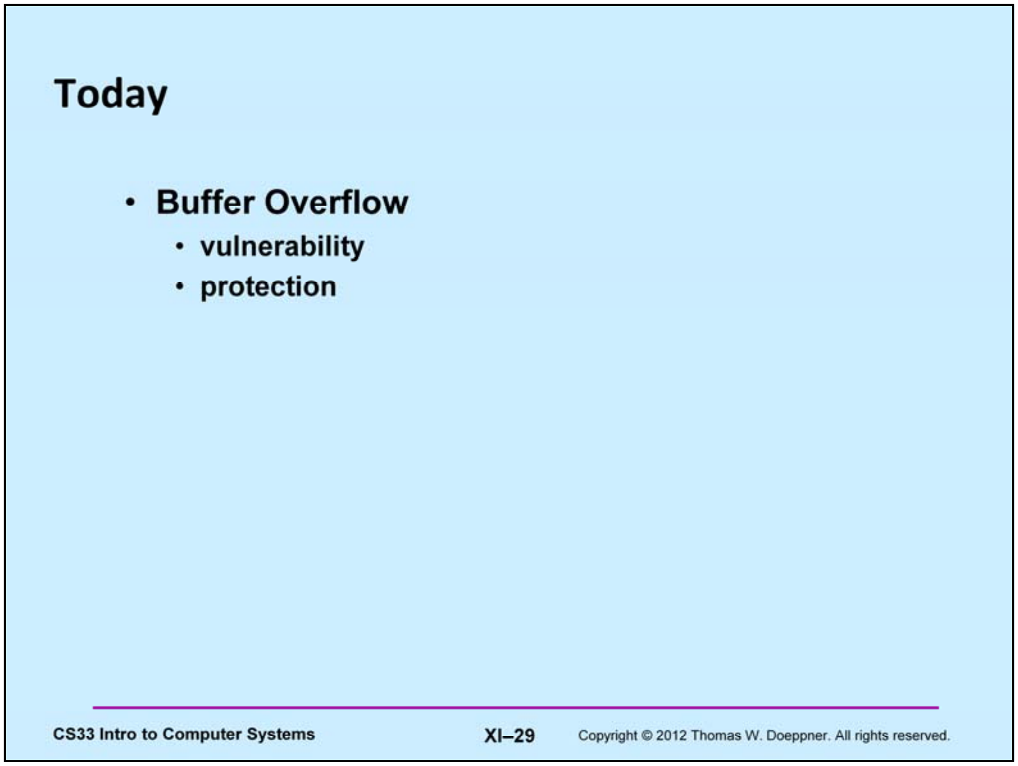
03 | e3 | 7d | 00

34 | 33 | 32 | 31 **buf**

Stack frame for **echo**

```
(gdb) break echo
(gdb) run
(gdb) stepi 3
(gdb) print /x *((unsigned *) $ebp - 2)
$1 = 0x3e37d00
```

Supplied by CMU.



Today

- Buffer Overflow
  - vulnerability
  - protection

Supplied by CMU.