# CS 6V81-05: System Security and Malicious Code Analysis
# Buffer Overflow, Integer and Heap Overflow

Zhiqiang Lin

Department of Computer Science
University of Texas at Dallas

April $11^{th}$, 2012

# Outline

# Outline

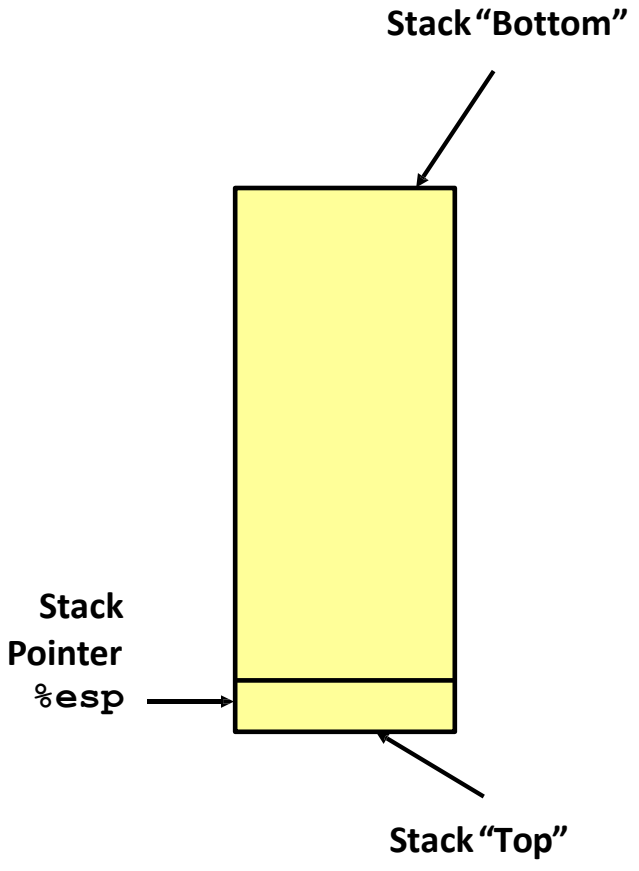
# IA32 Stack

Region of memory managed with stack discipline

Grows toward lower addresses

Register %esp indicates lowest stack address

address of top element

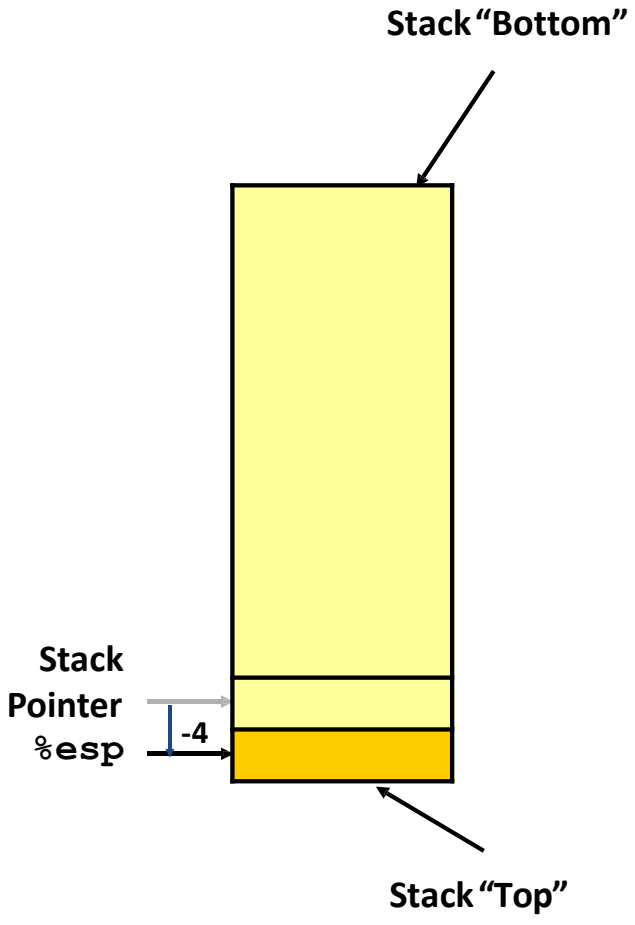


# IA32 Stack Pushing

## Pushing

`pushl src`

Fetch operand at Src
Decrement %esp by 4
Write operand at address given by %esp

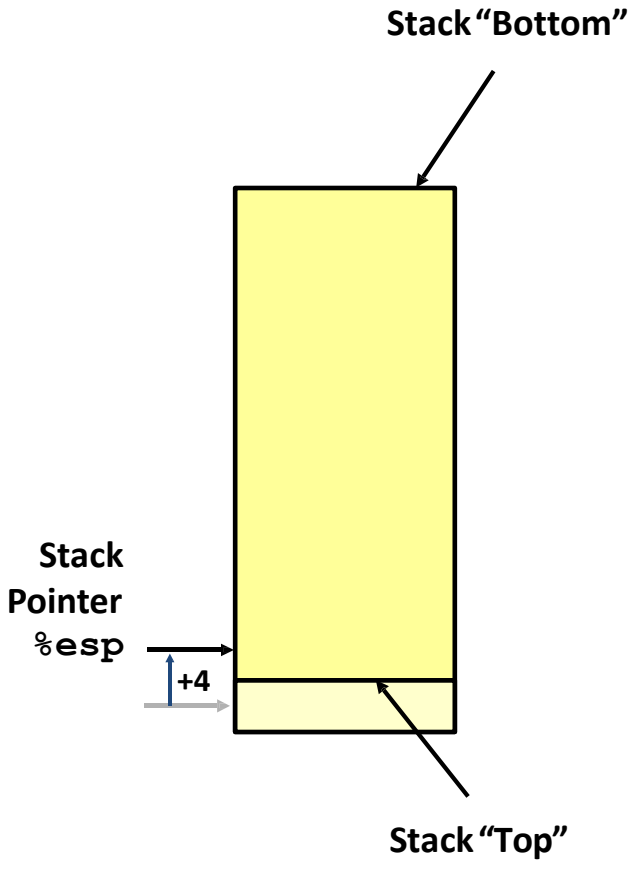


# IA32 Stack Popping

## Popping

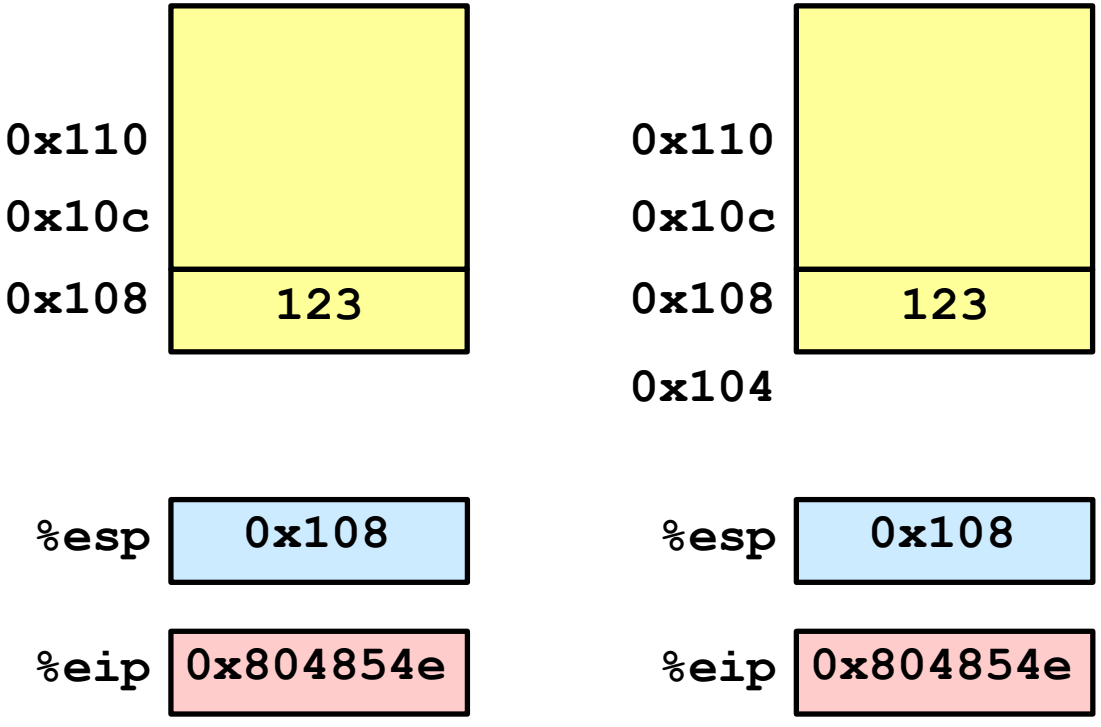
`popl dest`

Read operand at address given by %esp
Increment %esp by 4
Write to Dest



# Procedure Call Example

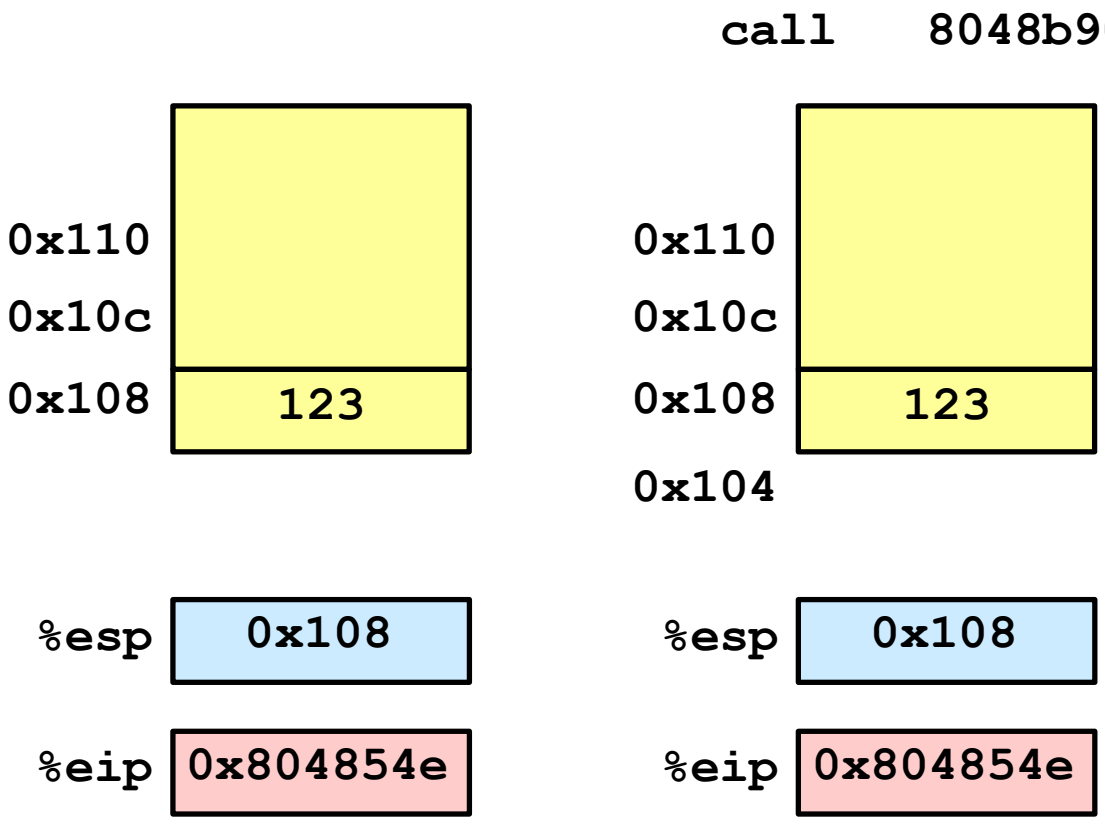
```
804854e: e8 3d 06 00 00   call   8048b90 <main>
8048553: 50               pushl  %eax
```




`%eip` is program counter

# Procedure Call Example

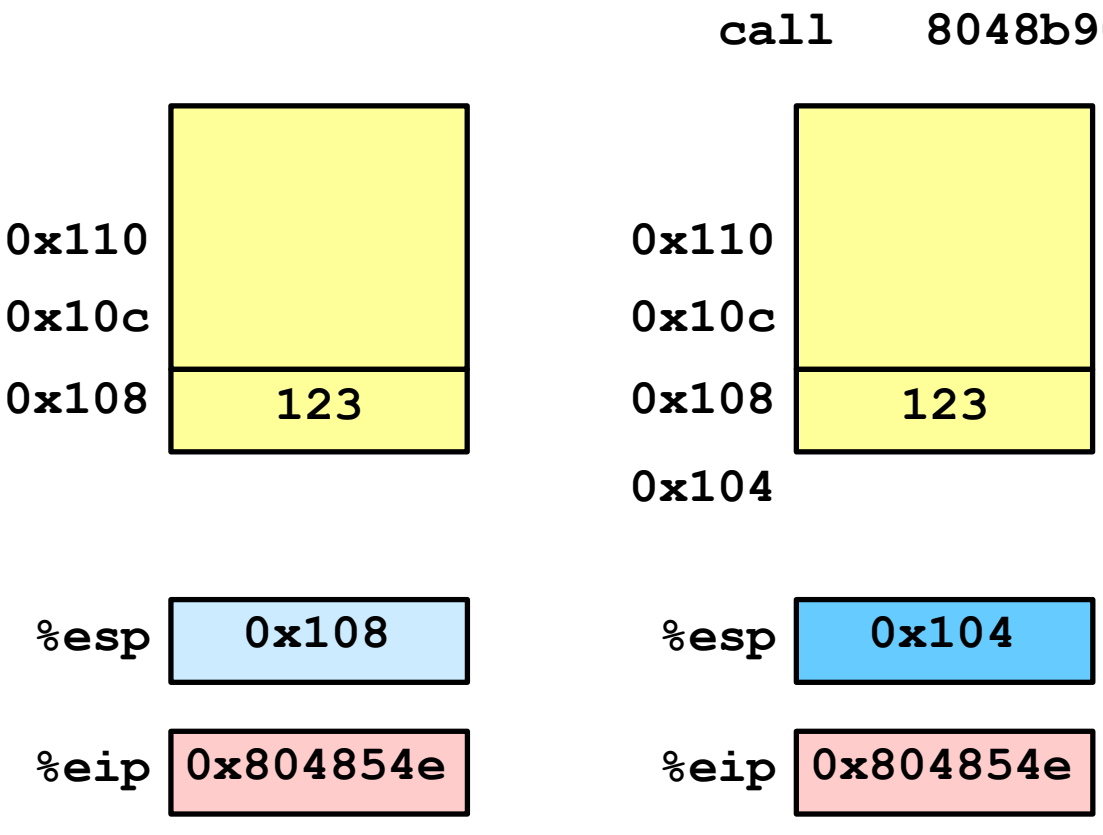
```
804854e: e8 3d 06 00 00   call   8048b90 <main>
8048553: 50               pushl  %eax
```




%eip is program counter

# Procedure Call Example

```
804854e: e8 3d 06 00 00   call   8048b90 <main>
8048553: 50               pushl  %eax
```

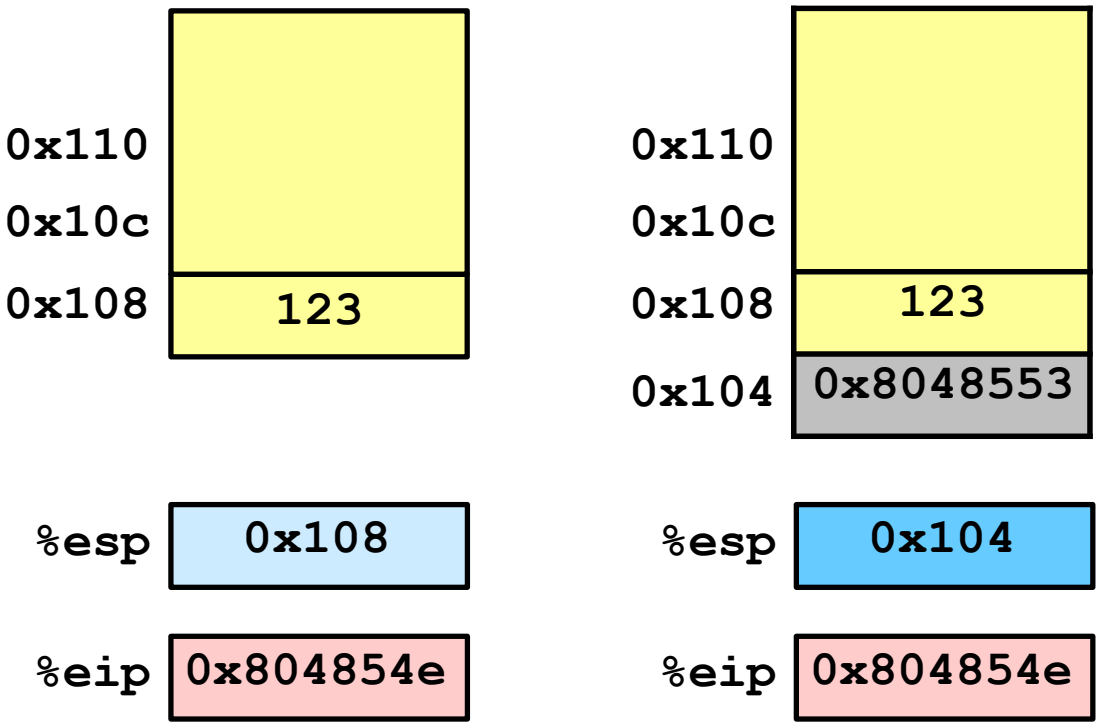



%eip is program counter

# Procedure Call Example

```
804854e: e8 3d 06 00 00   call   8048b90 <main>
8048553: 50               pushl  %eax
```

**call 8048b90**

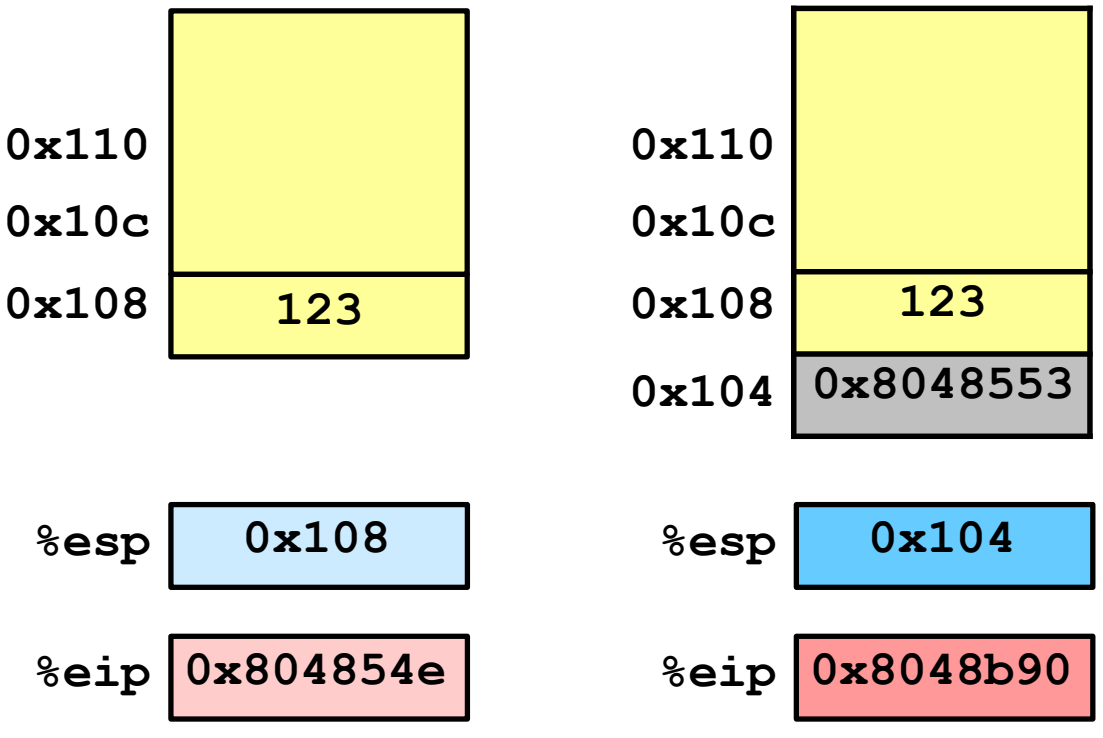



**%eip is program counter**

# Procedure Call Example

```
804854e: e8 3d 06 00 00   call   8048b90 <main>
8048553: 50               pushl  %eax
```

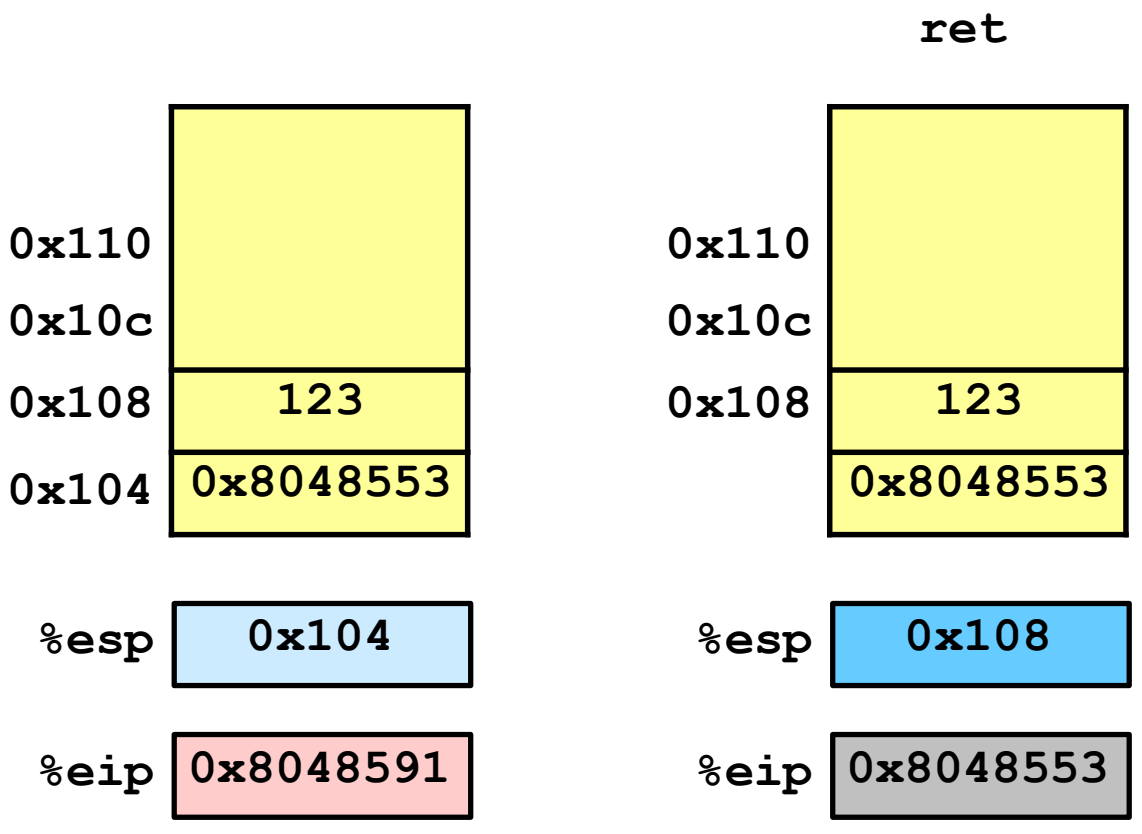
**call 8048b90**




**%eip is program counter**

# Procedure Return Example

```
8048591: c3                    ret
```




`%eip` is program counter

# Call Chain Example

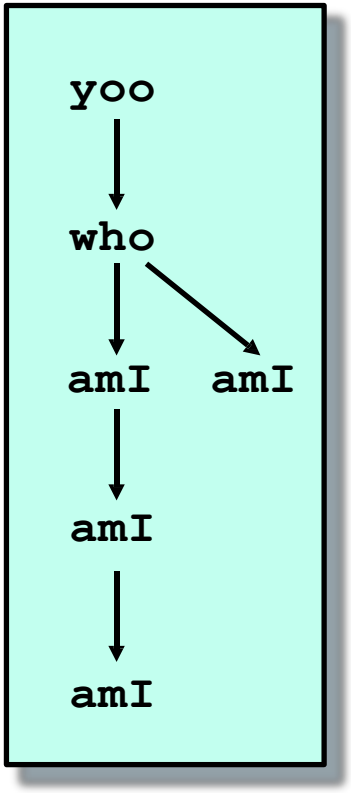
- **Code Structure**

```
yoo(...)
{
    •
    •
    who();
    •
    •
}
```

```
who(...)
{
    • • •
    amI();
    • • •
    amI();
    • • •
}
```

```
amI(...)
{
    •
    •
    amI();
    •
    •
}
```

**Call Chain**




- **Procedure `amI` recursive**

# Call Chain Example

Contents

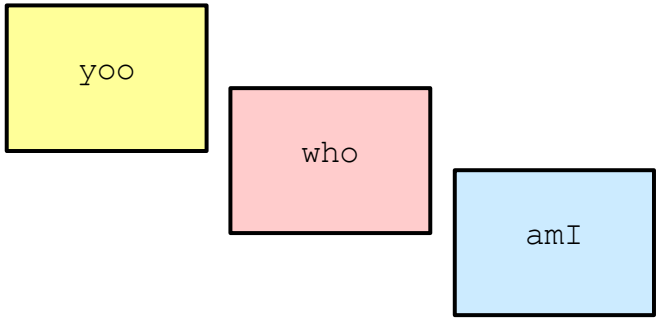
- Local variables
- Return information
- Temporary space

Management

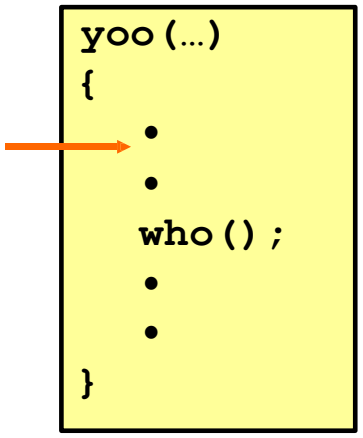
- Space allocated when enter procedure
  - "Set-up" code
- Deallocated when return
  - "Finish" code

Pointers

- Stack pointer **%esp** indicates stack top
- Frame pointer **%ebp** indicates start of current frame

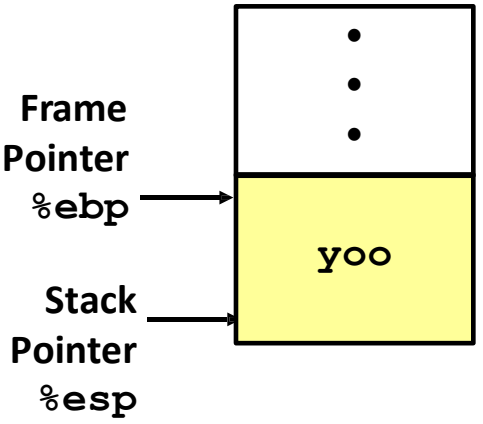


# Stack Operation

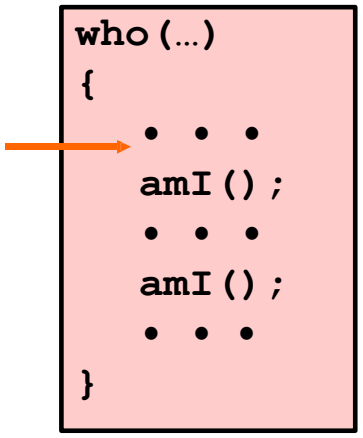



Call Chain

yoo



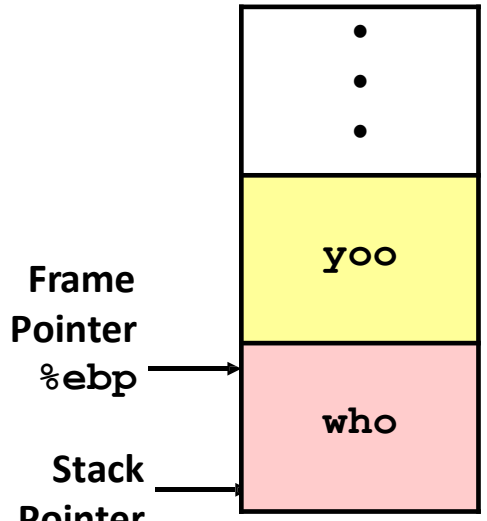
# Stack Operation




Call Chain

yoo

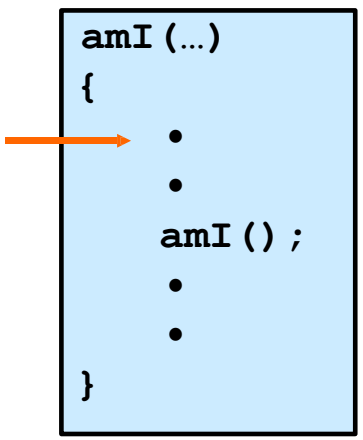
who

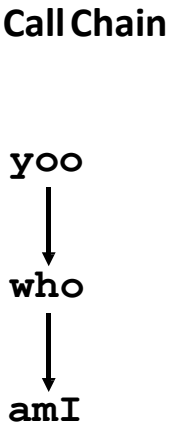


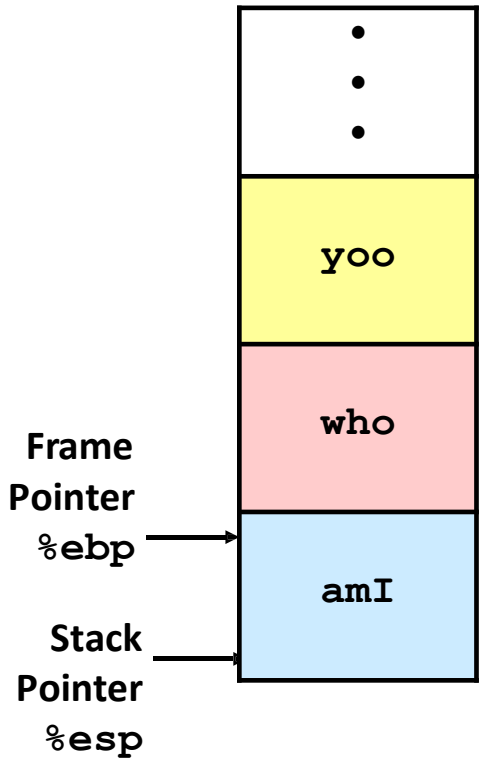

Frame Pointer %ebp

Stack Pointer %esp

# Stack Operation



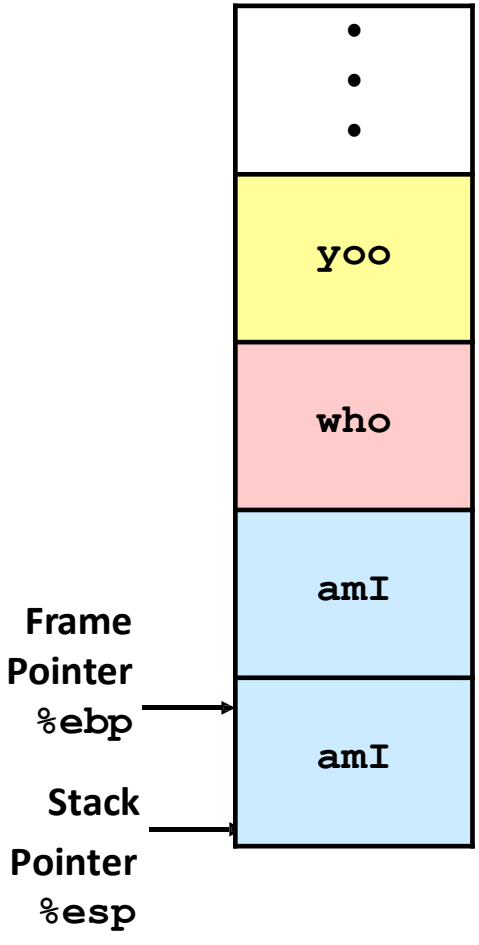



# Stack Operation

```
amI(...)
{
    •
    •
    amI();
    •
    •
}
```

Call Chain

yoo
↓
who
↓
amI
↓
amI

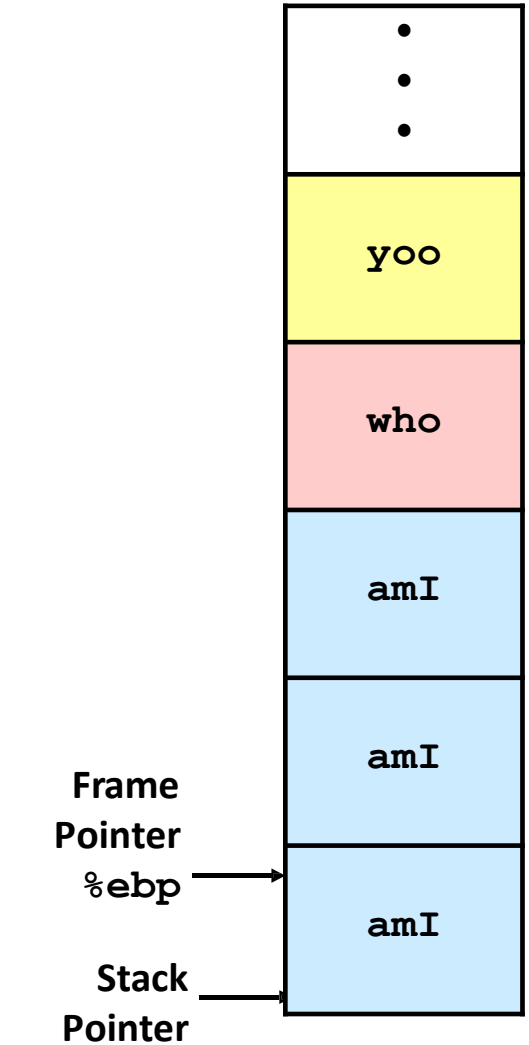


# Stack Operation

```
amI(...)
{
    •
    •
    amI();
    •
    •
}
```

Call Chain

yoo
who
amI
amI
amI

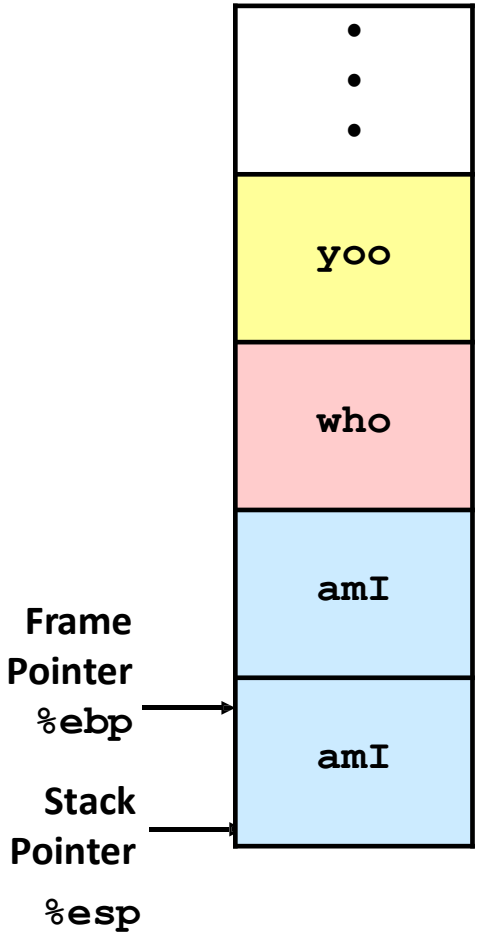


# Stack Operation

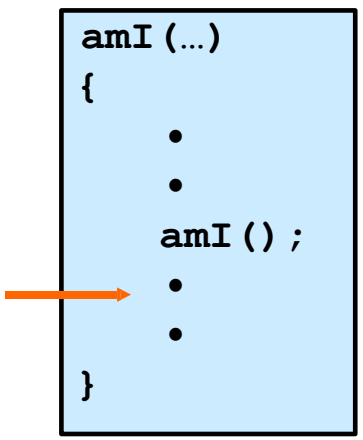
```
amI(...)
{
    •
    •
    amI();
    •
    •
}
```

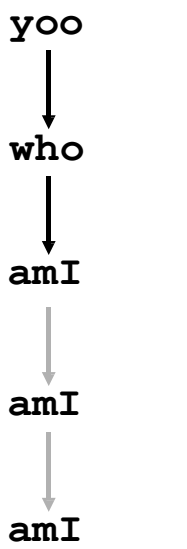
Call Chain

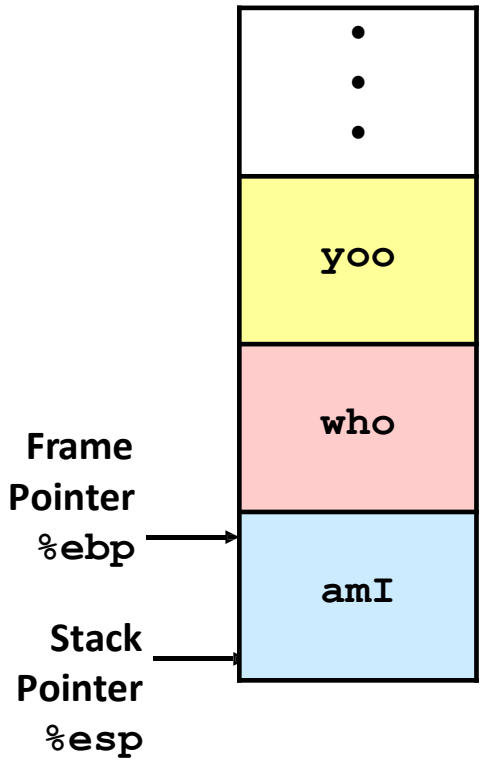
yoo
↓
who
↓
amI
↓
amI
↓
amI



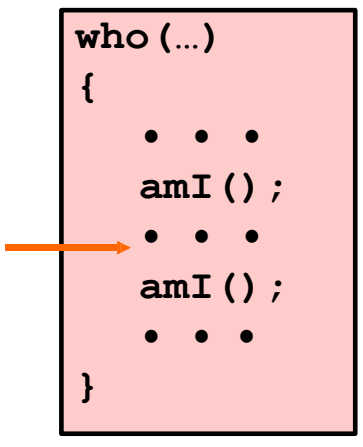
# Stack Operation



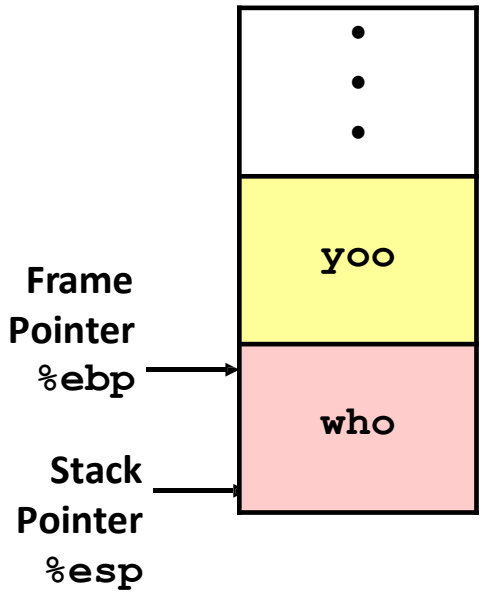



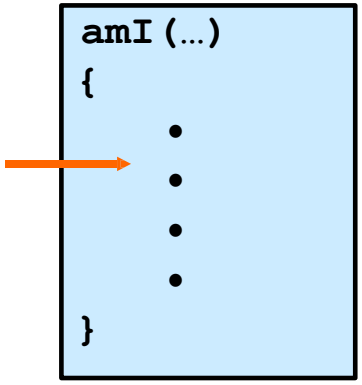
# Stack Operation

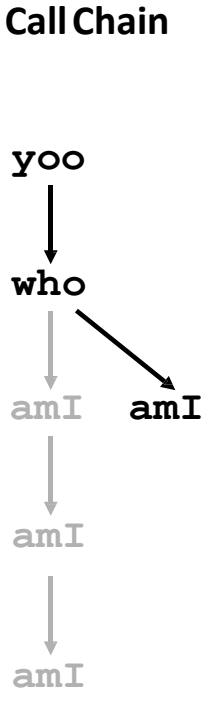


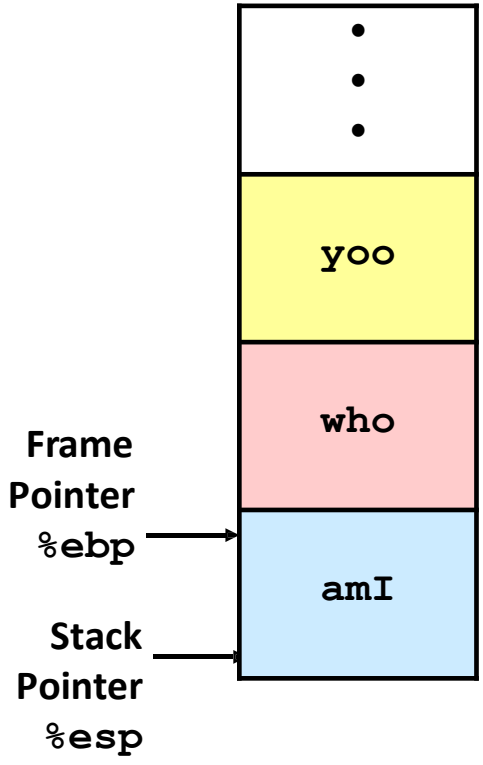




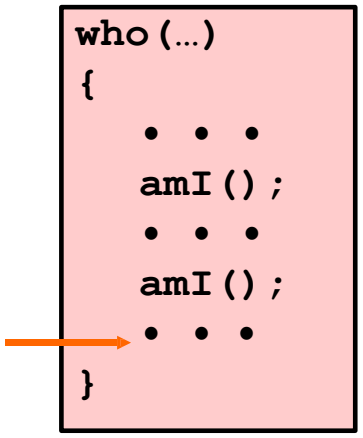
# Stack Operation

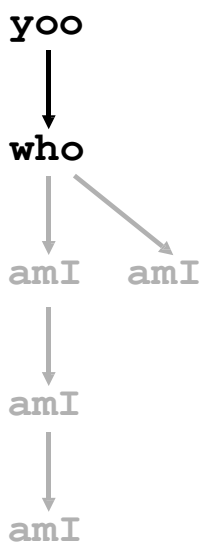


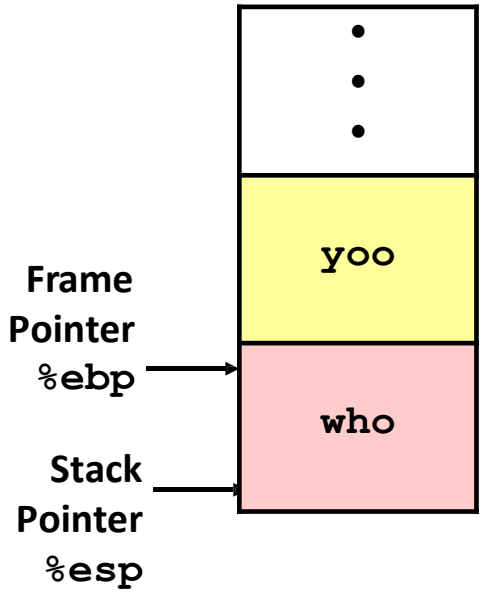



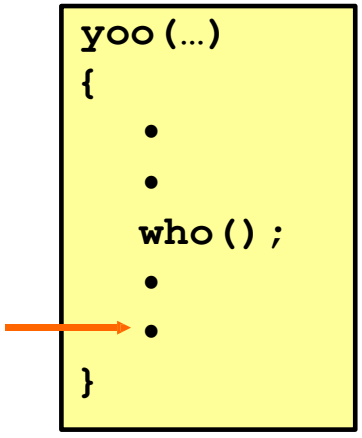
# Stack Operation

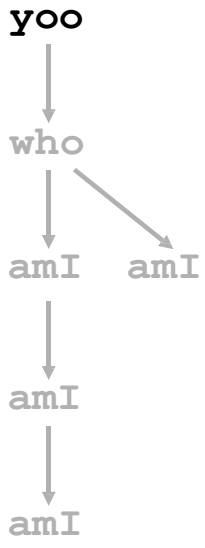


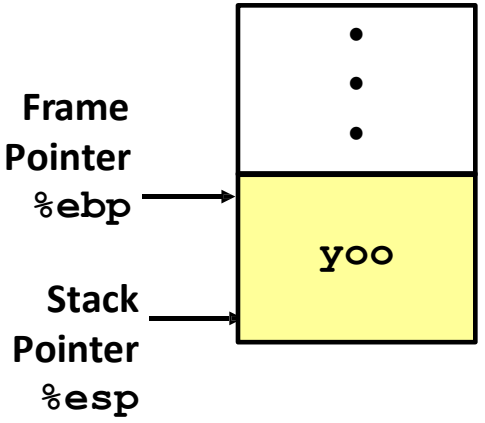



# Stack Operation

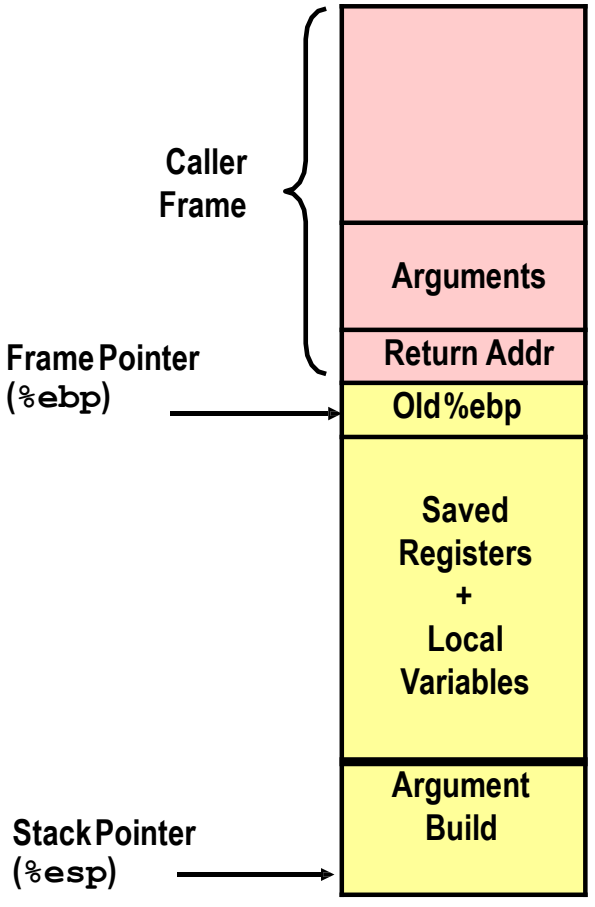


# IA32/Linux Stack Frame

- Current Stack Frame ("Top" to Bottom)
  - Parameters for function about to call
    - "Argument build"
  - Local variables
    - If can't keep in registers
  - Saved register context
    Old frame pointer
- Caller Stack Frame
  - Return address
    - Pushed by **call** instruction
  - Arguments for this call



# IA32 Linux Memory Layout

Stack

Runtime stack (8MB limit)
E. g., local variables

Heap

Dynamically allocated storage
When call malloc(), calloc(), new()

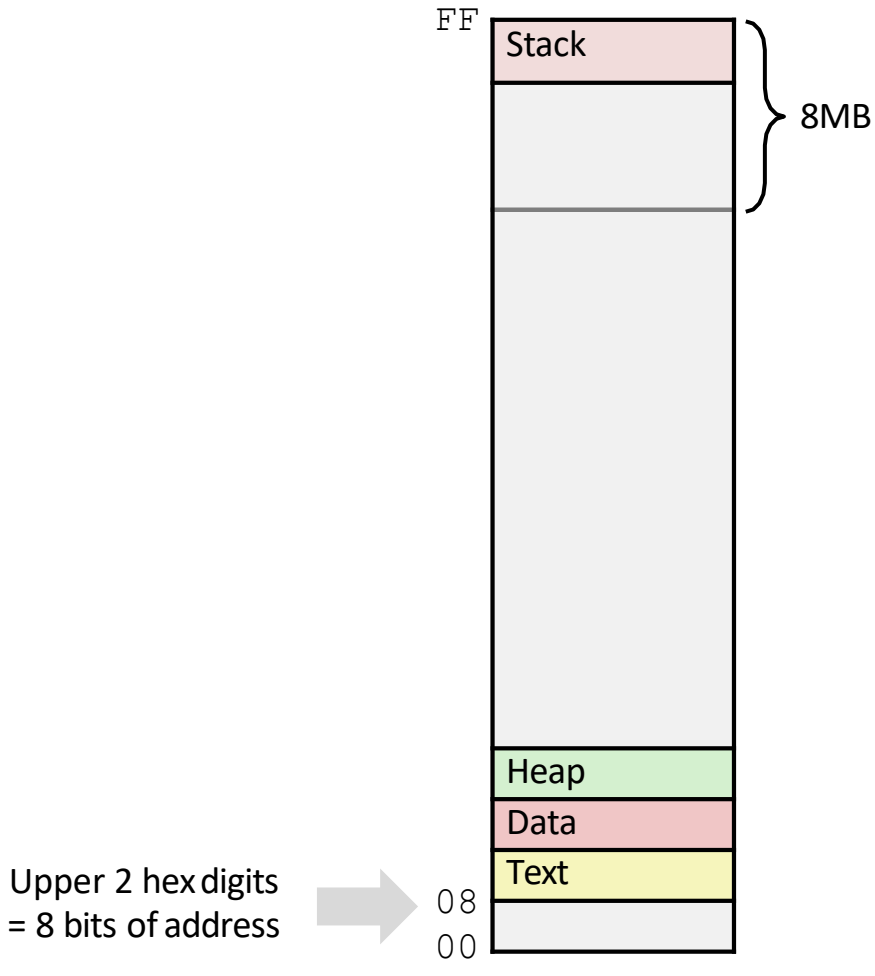
Data

Statically allocated data
E.g., arrays & strings declared in code

Text

Executable machine instructions

Read-only



# Memory Allocation Example

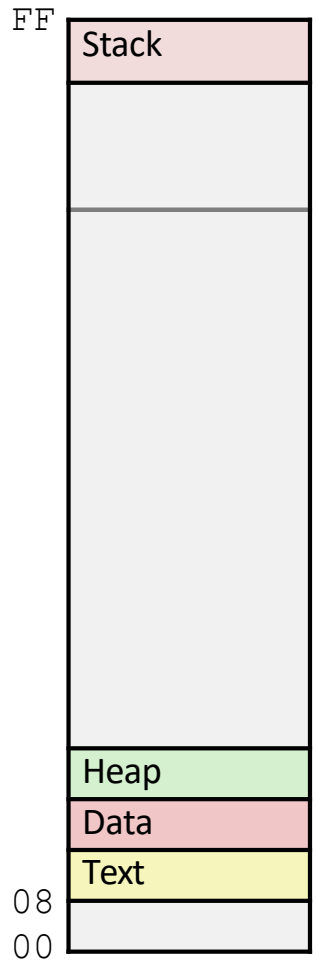
```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB*/

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?

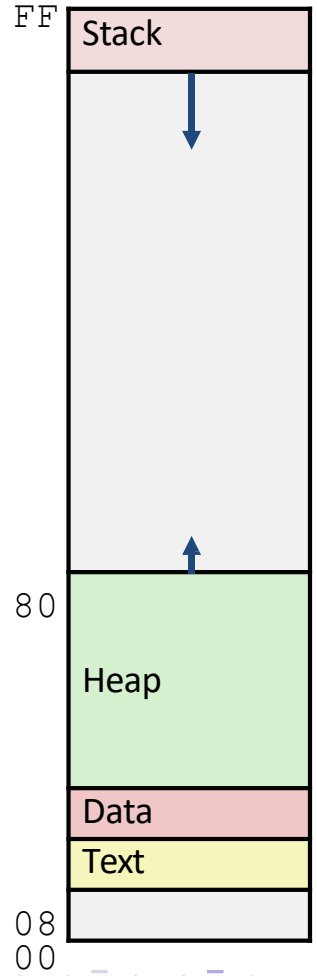


# IA32 Example Addresses

*address range ~$2^{32}$*

| | |
|---|---|
| `$esp` | `0xffffbcd0` |
| `p3` | `0x65586008` |
| `p1` | `0x55585008` |
| `p4` | `0x1904a110` |
| `p2` | `0x1904a008` |
| `&p2` | `0x18049760` |
| `&beyond` | `0x08049744` |
| `big_array` | `0x18049780` |
| `huge_array` | `0x08049760` |
| `main()` | |
| `useless()` | `0x08049744` |
| final `malloc()` | `0x006be166` |

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



# Outline

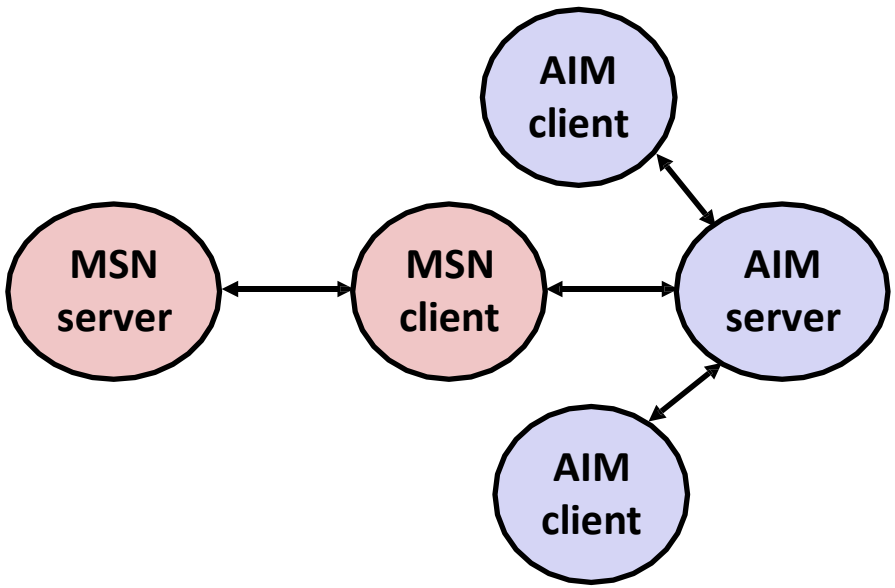
# Internet Worm and IM War

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



# 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.

# 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

# 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
```

# Buffer Overflow Disassembly

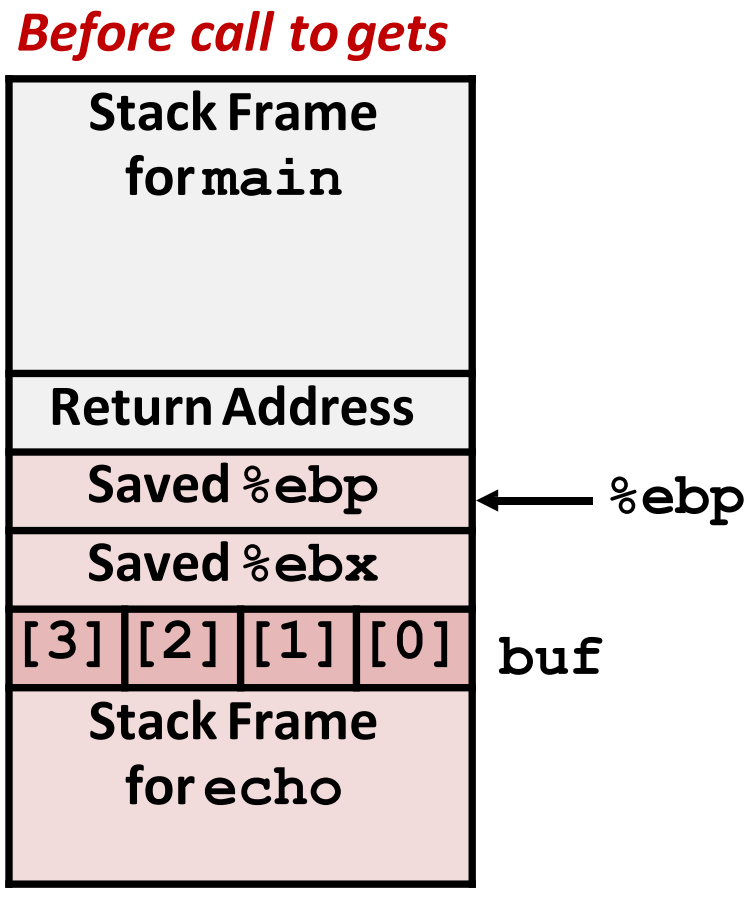
## echo

```
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
```

# Buffer Overflow Stack

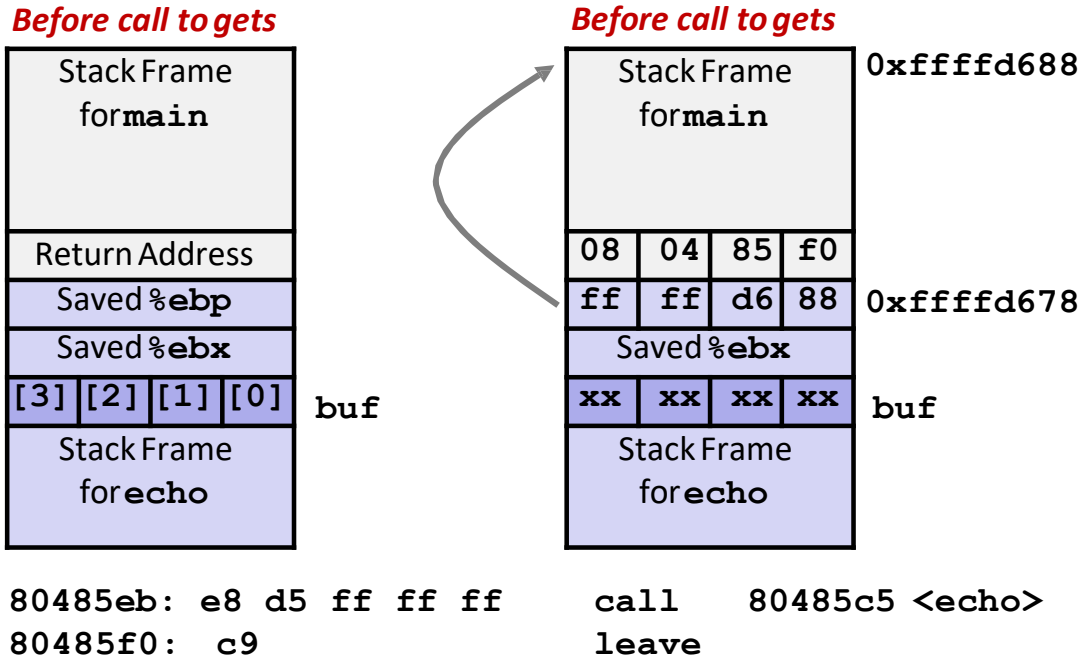



```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

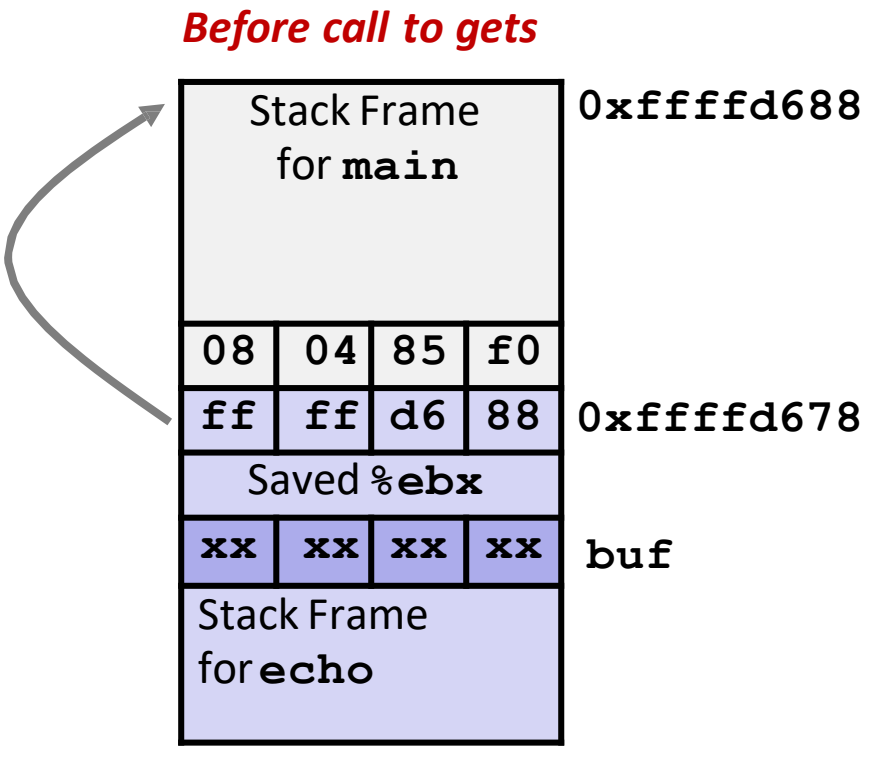
```
echo:
pushl %ebp # Save %ebp on stack
movl  %esp, %ebp
pushl %ebx # Save %ebx
subl  $20, %esp # Allocate stack space
leal -8(%ebp),%ebx # Compute buf as %ebp-8
movl  %ebx, (%esp) # Push buf on stack
call  gets # Call gets
. . .
```

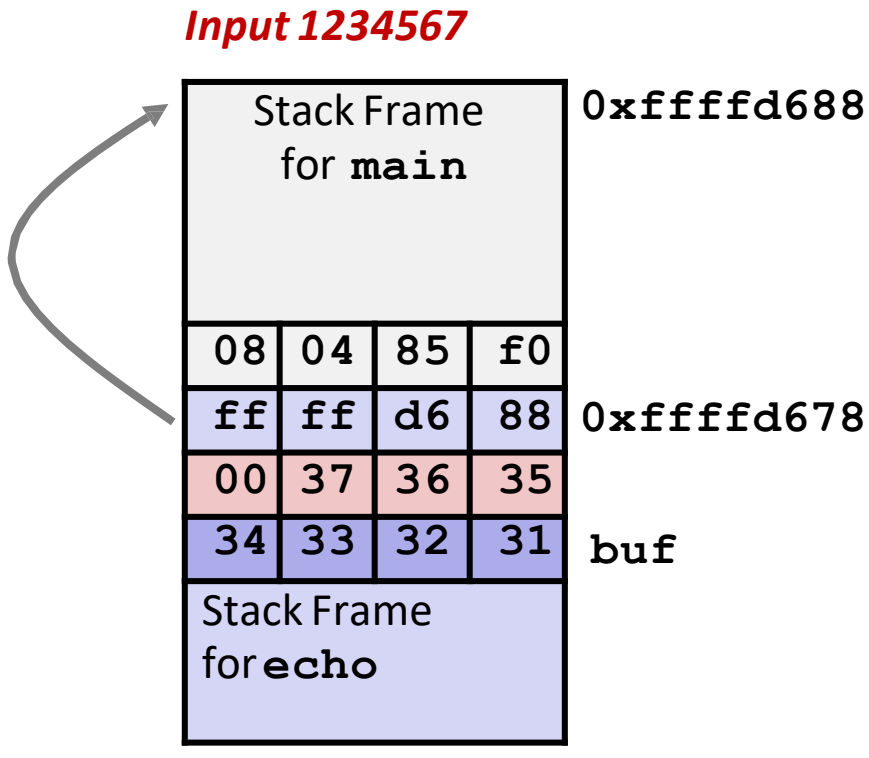
# Buffer Overflow Stack Example

```
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
```



# Buffer Overflow Stack Example #1

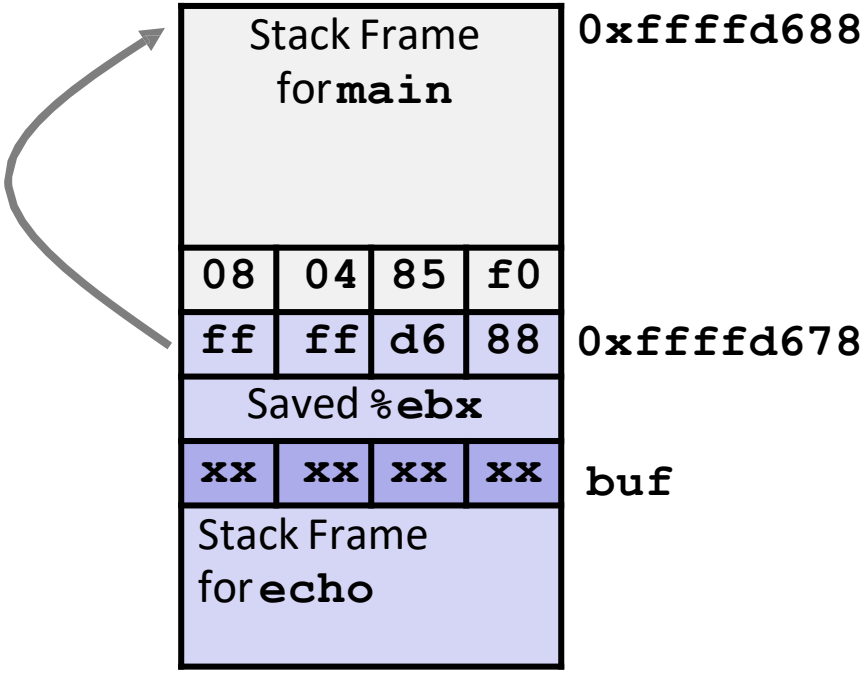


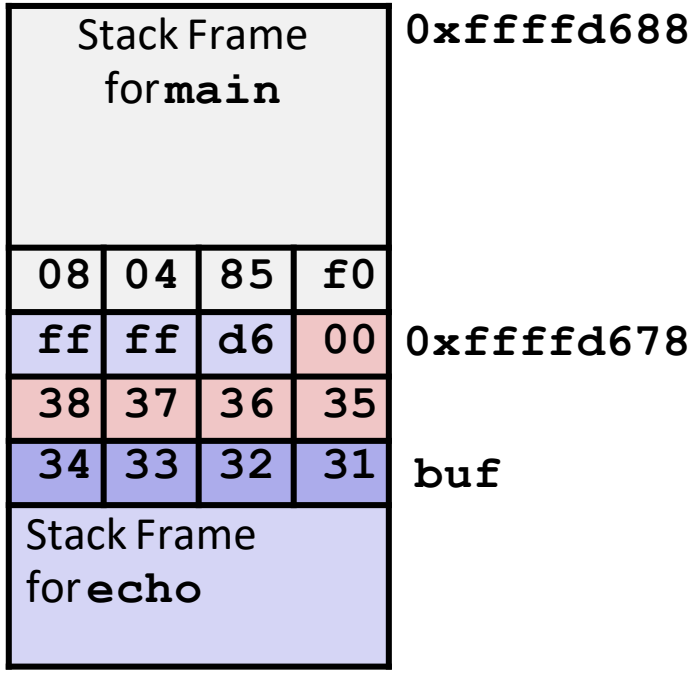

**Overflow buf, and corrupt %ebx, but no problem**

# Buffer Overflow Stack Example #2

***Before call to gets***




***Input 12345678***

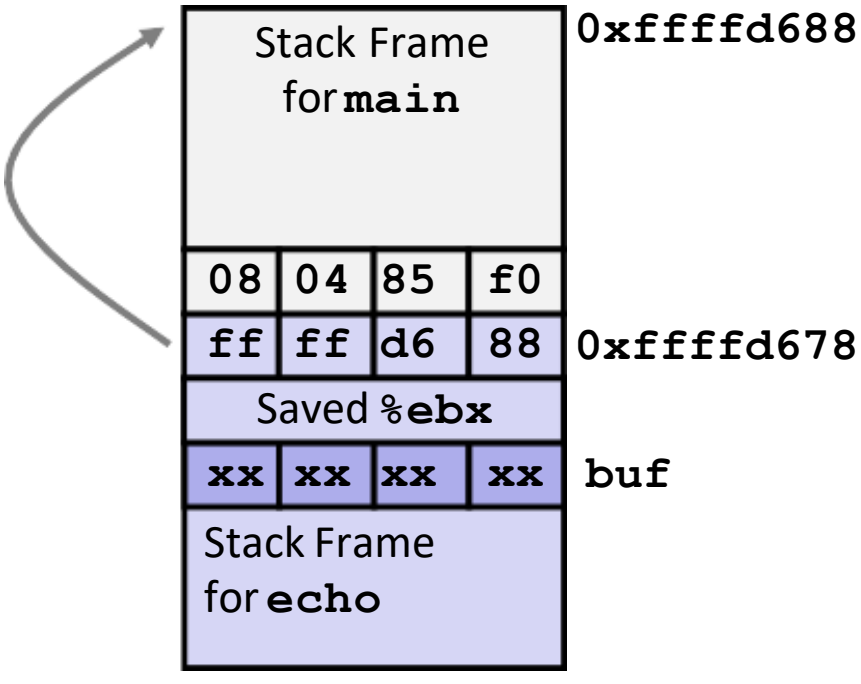



**Base pointer corrupted**

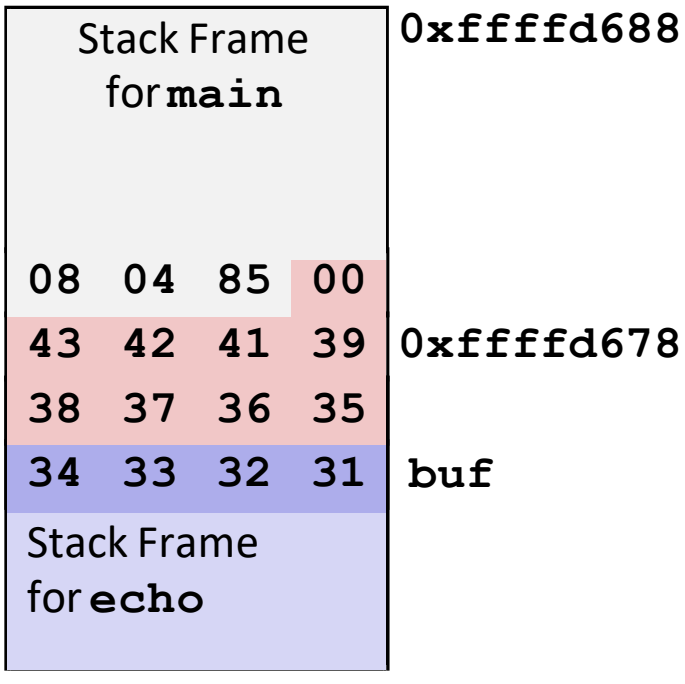
```
. . .
80485eb:    e8 d5 ff ff ff    call   80485c5 <echo>
80485f0:    c9                leave    # Set %ebp to corrupted value
80485f1:    c3                ret
```

# Buffer Overflow Stack Example #3

***Before call to gets***




***Input 123456789!"#***




**Return address corrupted**

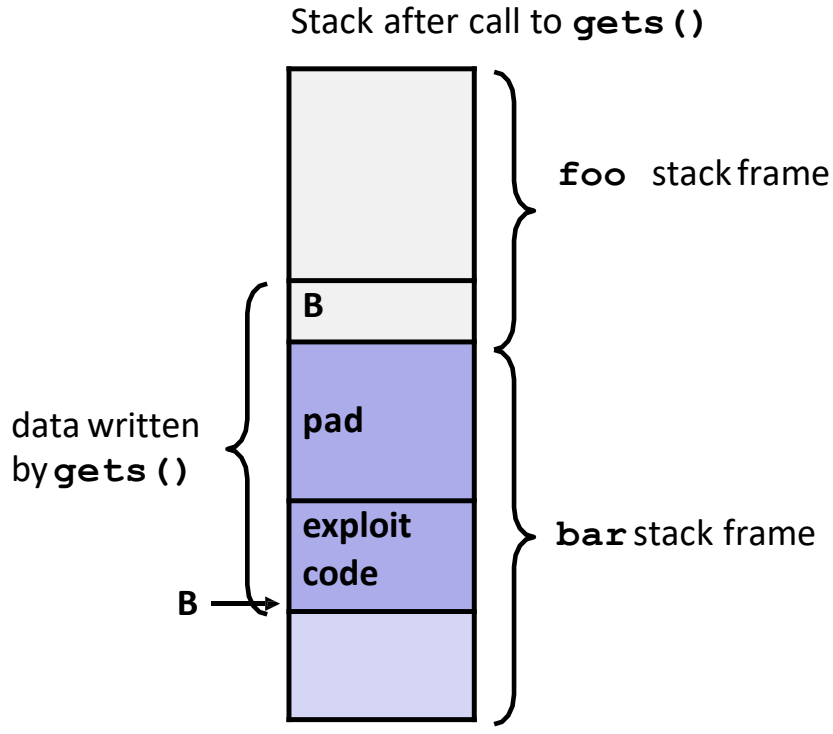
```
80485eb:    e8 d5 ff ff ff    call   80485c5 <echo>
80485f0:    c9                leave    # Desired return point
```

# Buffer Overflow Stack Example #3

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

return address A

```
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

# 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 (fingered) used **gets()** to read the argument sent by the client:
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.

# Exploits Based on Buffer Overflows

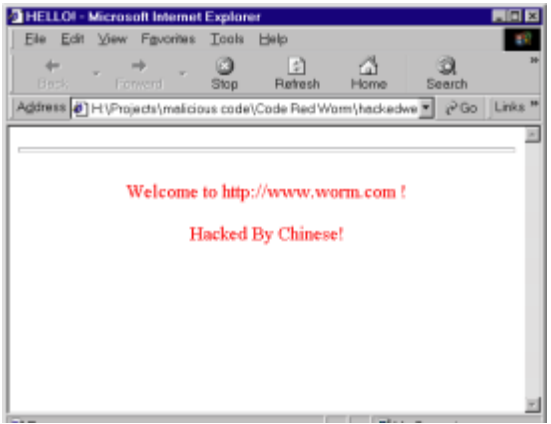
Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines

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.

# 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



# Avoiding Overflow Vulnerability

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

# 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
  - X86-64 added 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
```

# 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 ***
```

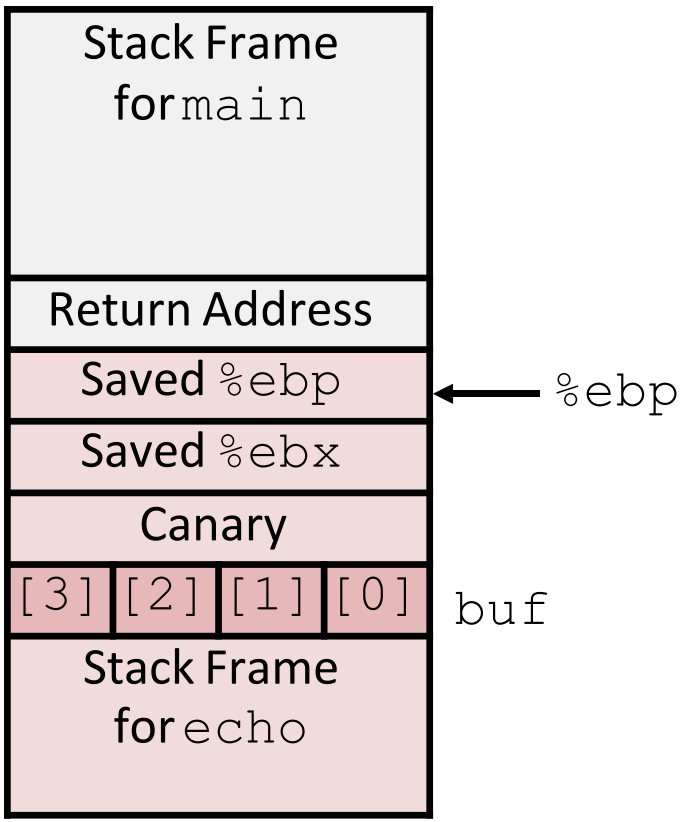
# Protected Buffer Disassembly

## 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
```

# Setting Up Canary

*Before call to gets*

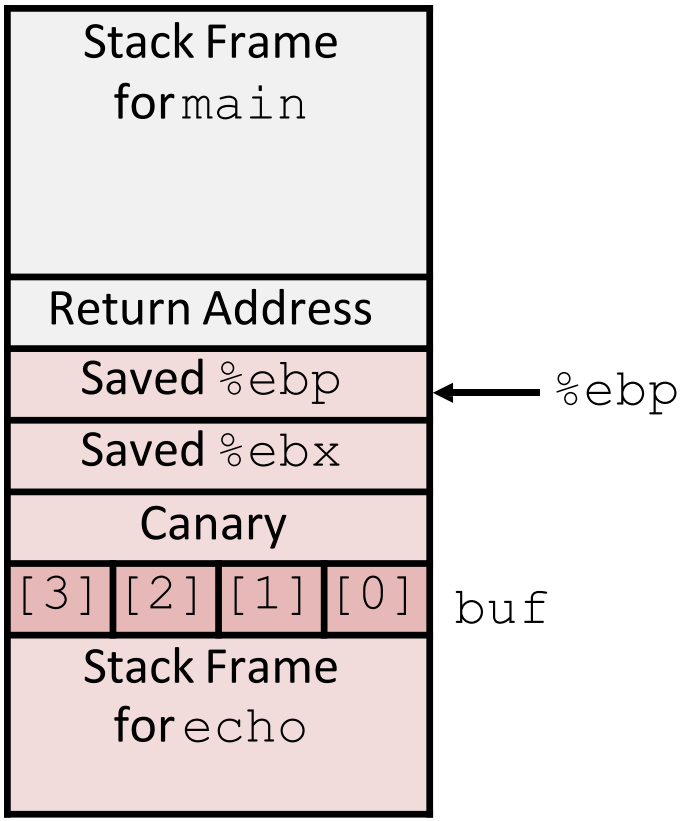



```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

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

# Checking Canary

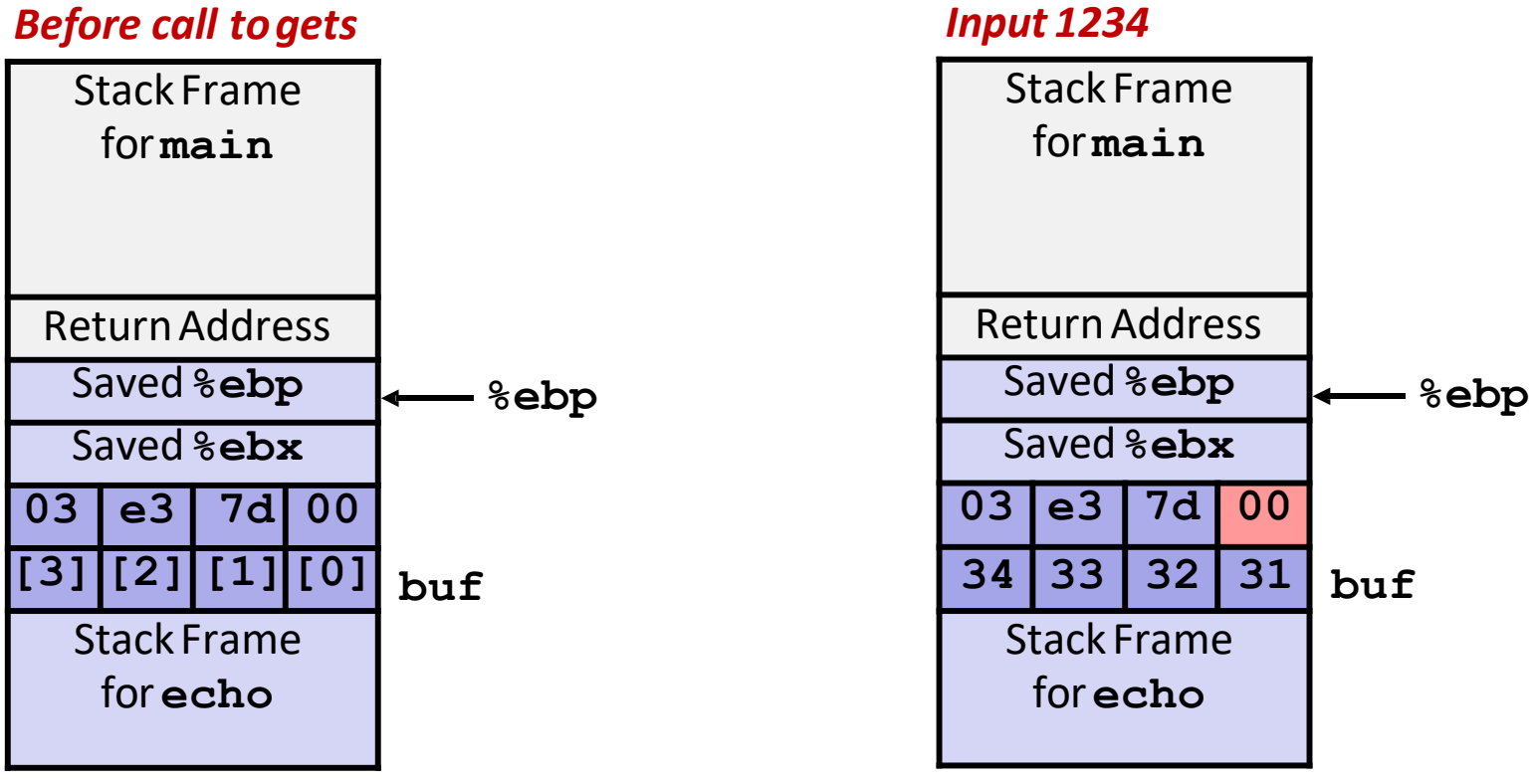
*Before call to gets*




```
/* 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:
. . .
```

# Canary Example

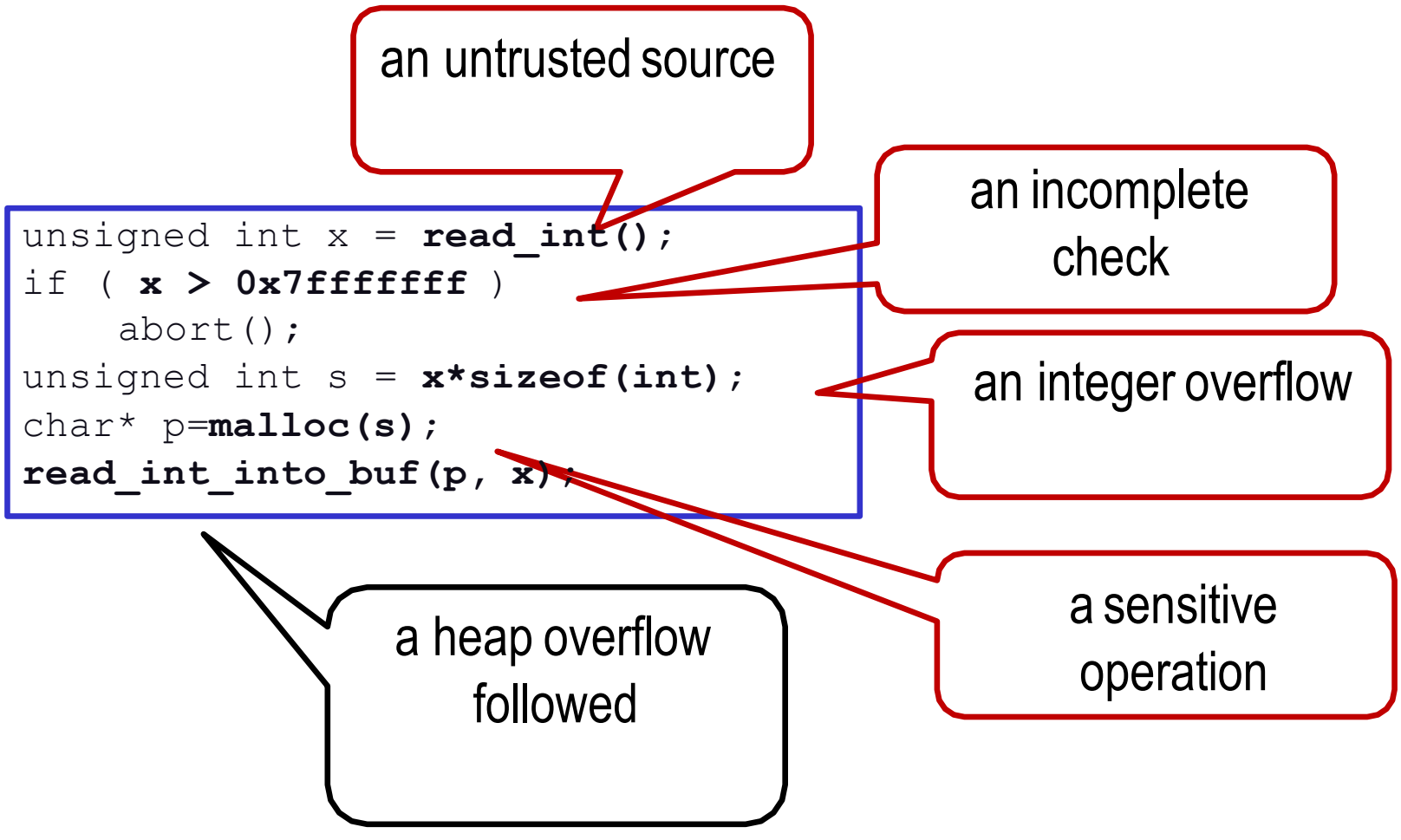



```
(gdb) break echo
(gdb) run
(gdb) stepi 3
(gdb) print /x *((unsigned *) $ebp - 2)
$1 = 0x3e37d00
```

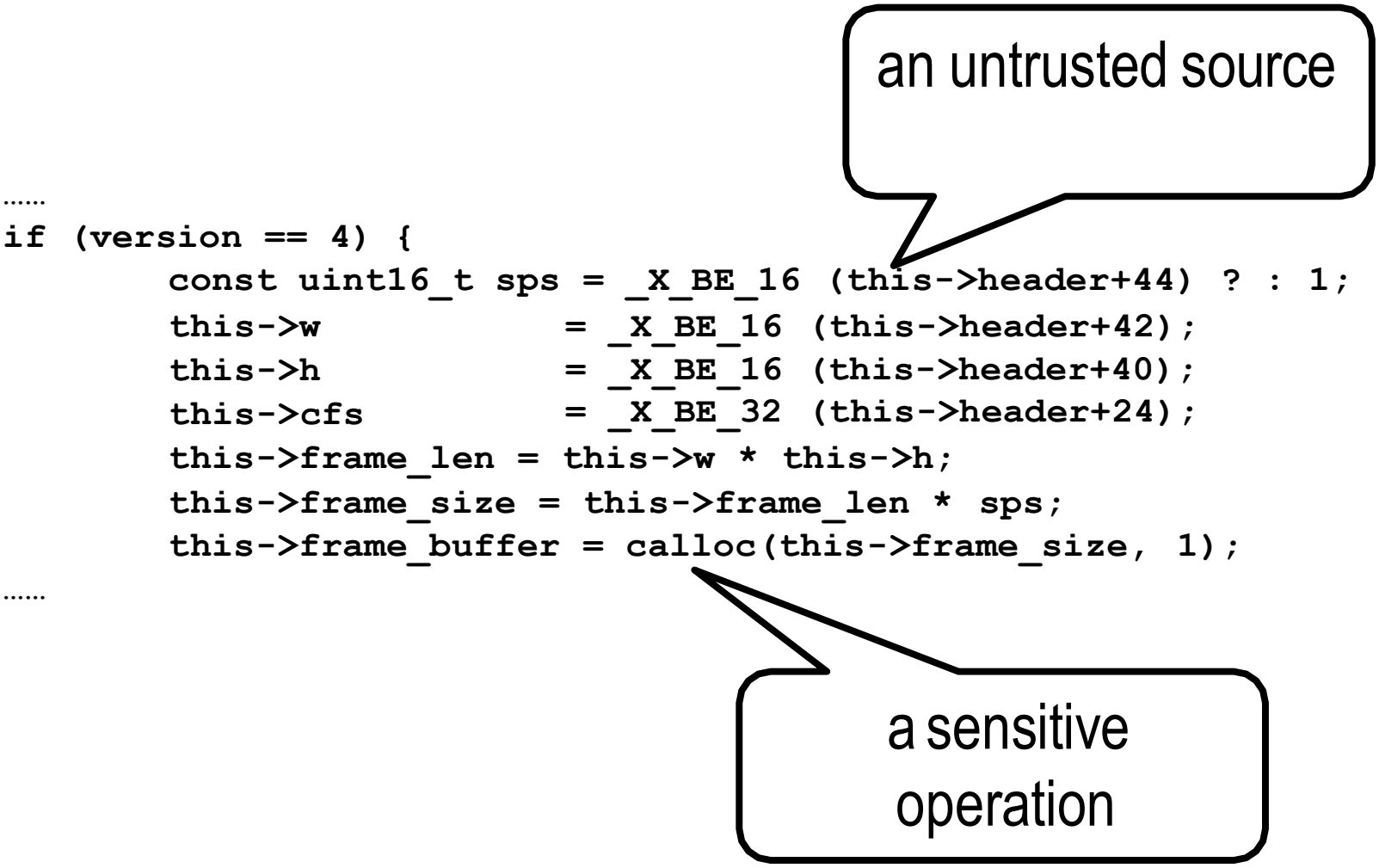
**Benign corruption!**
**(allows programmers to make silent off-by-one errors)**

# Outline

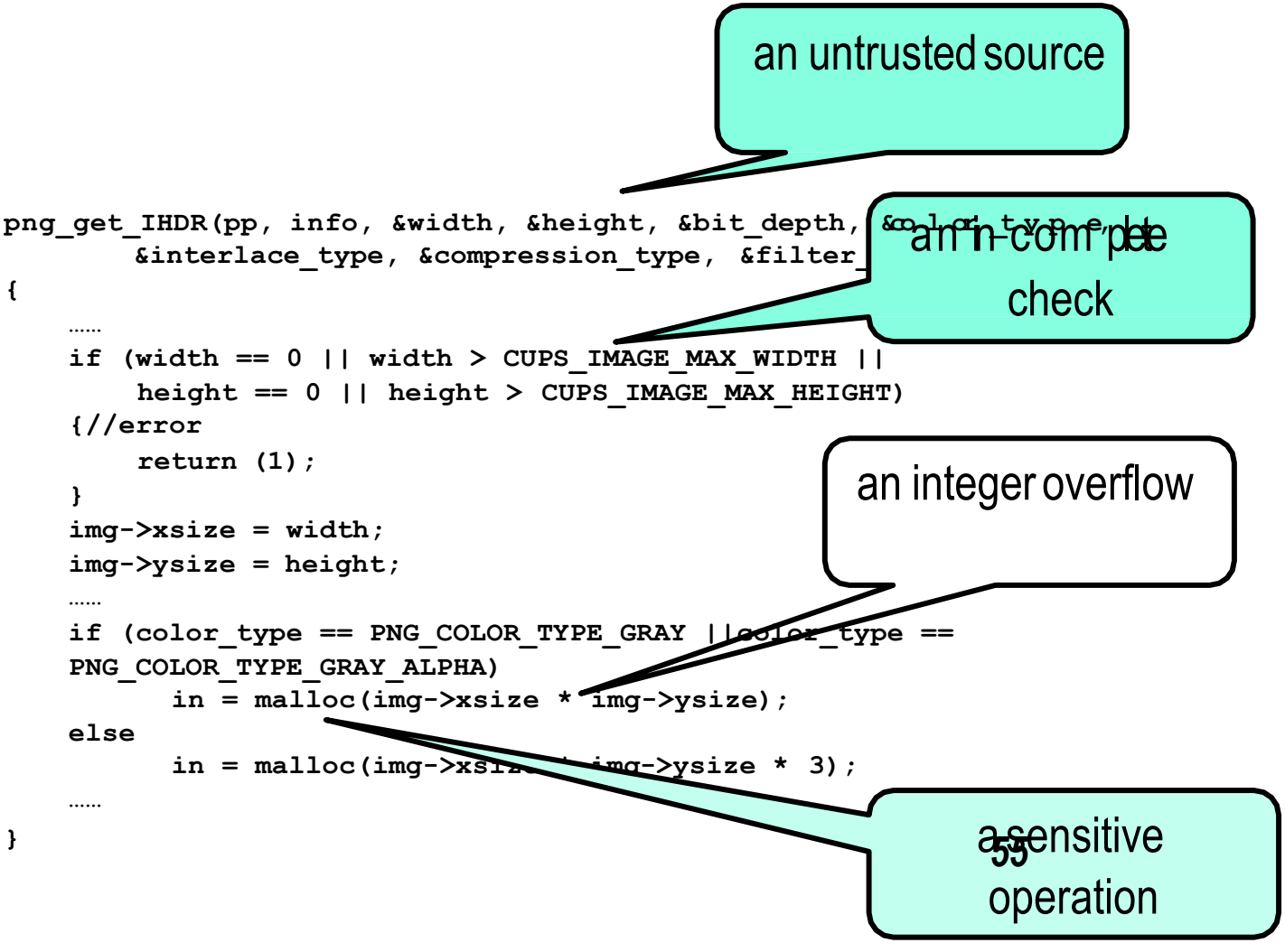
# What are the common features of integer overflow vulnerabilities?



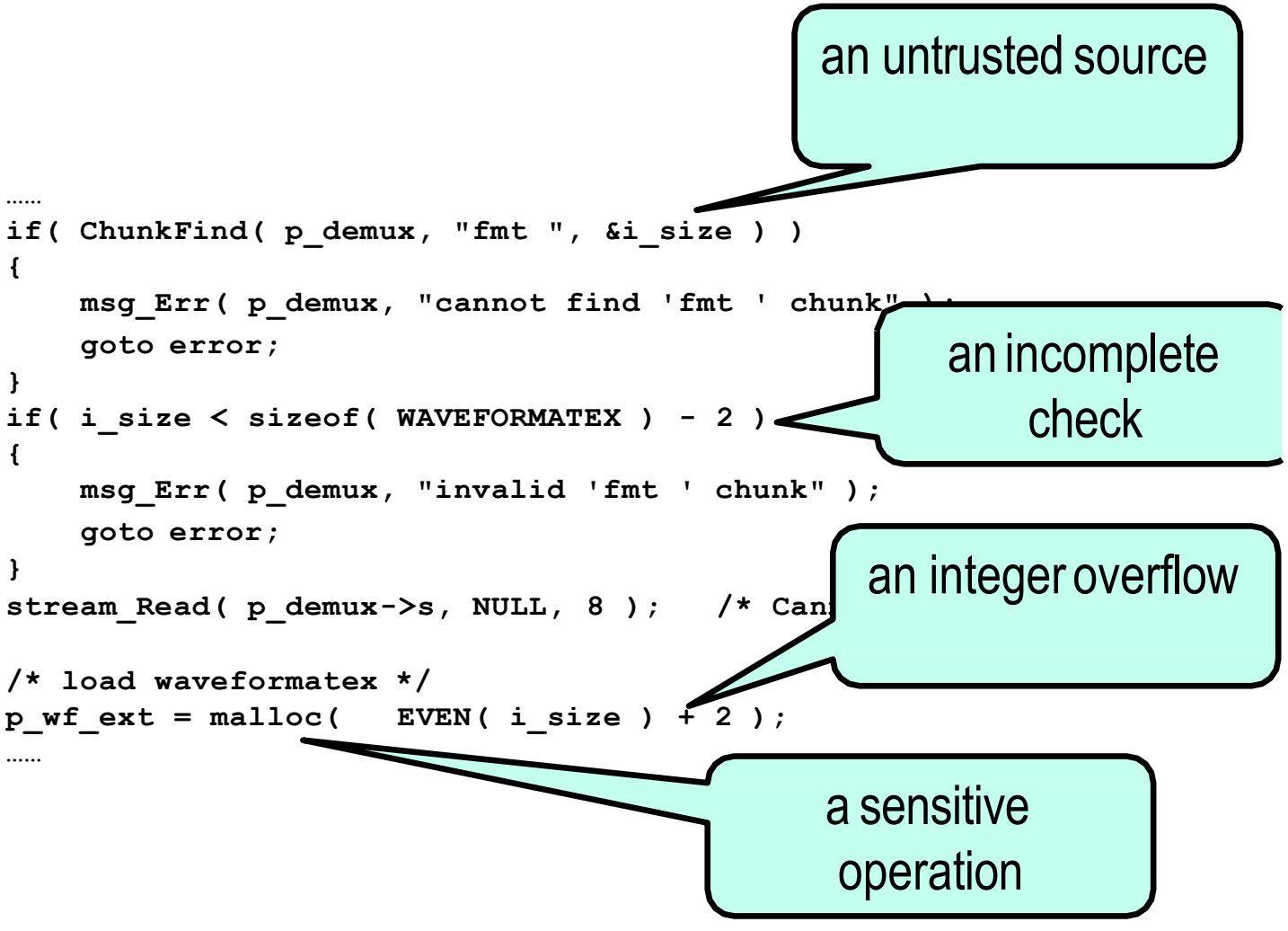
# CVE-2008-5238(Xine)



# CVE-2008-1722(CUPS)



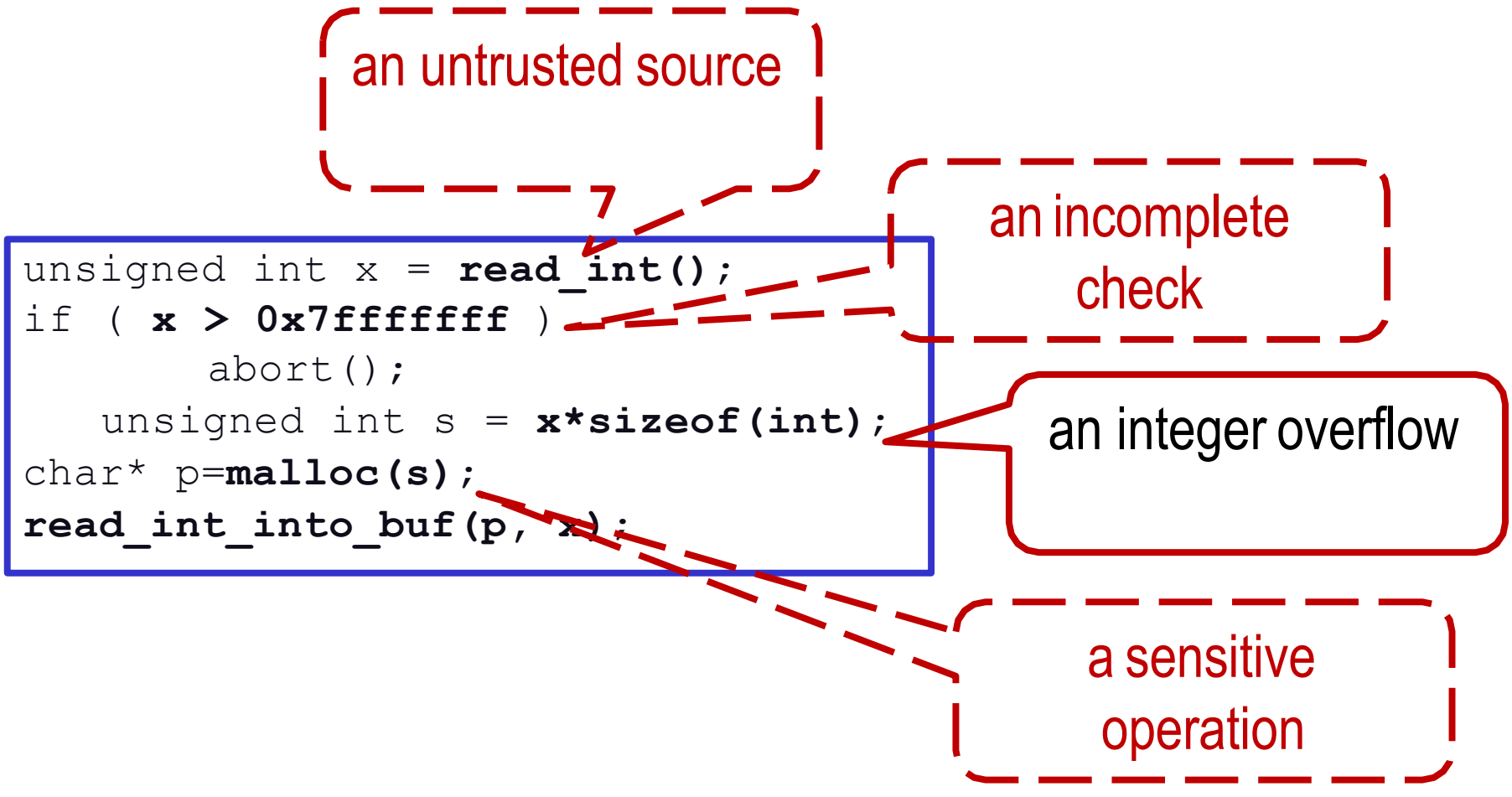
# CVE-2008-2430(VLC)




```
.....
if( ChunkFind( p_demux, "fmt ", &i_size ) )
{
    msg_Err( p_demux, "cannot find 'fmt ' chunk" );
    goto error;
}
if( i_size < sizeof( WAVEFORMATEX ) - 2 )
{
    msg_Err( p_demux, "invalid 'fmt ' chunk" );
    goto error;
}
stream_Read( p_demux->s, NULL, 8 );   /* Can

/* load waveformatex */
p_wf_ext = malloc(   EVEN( i_size ) + 2 );
.....
```

# What's the essential feature of integer overflow vulnerabilities?



# Outline

# Summary

## Software vulnerabilities

1. Cannot be avoided (software complexity)
2. Memory bugs (buffer overflow, integer overflow) are dangerous
3. Tons of research has been carried out to stop memory bugs
4. Stack, and integer and heap overflow can be stopped.

# References

http://en.wikipedia.org/wiki/Buffer_overflow Smashing the Stack for Fun and Profit by Aleph One

A Comparison of Buffer Overflow Prevention Implementations and Weaknesses (Blackhat 2004)

http://www.cs.cmu.edu/afs/cs/academic/class/15213-f10/www/lectures/08-machine-advanced.pdf

...