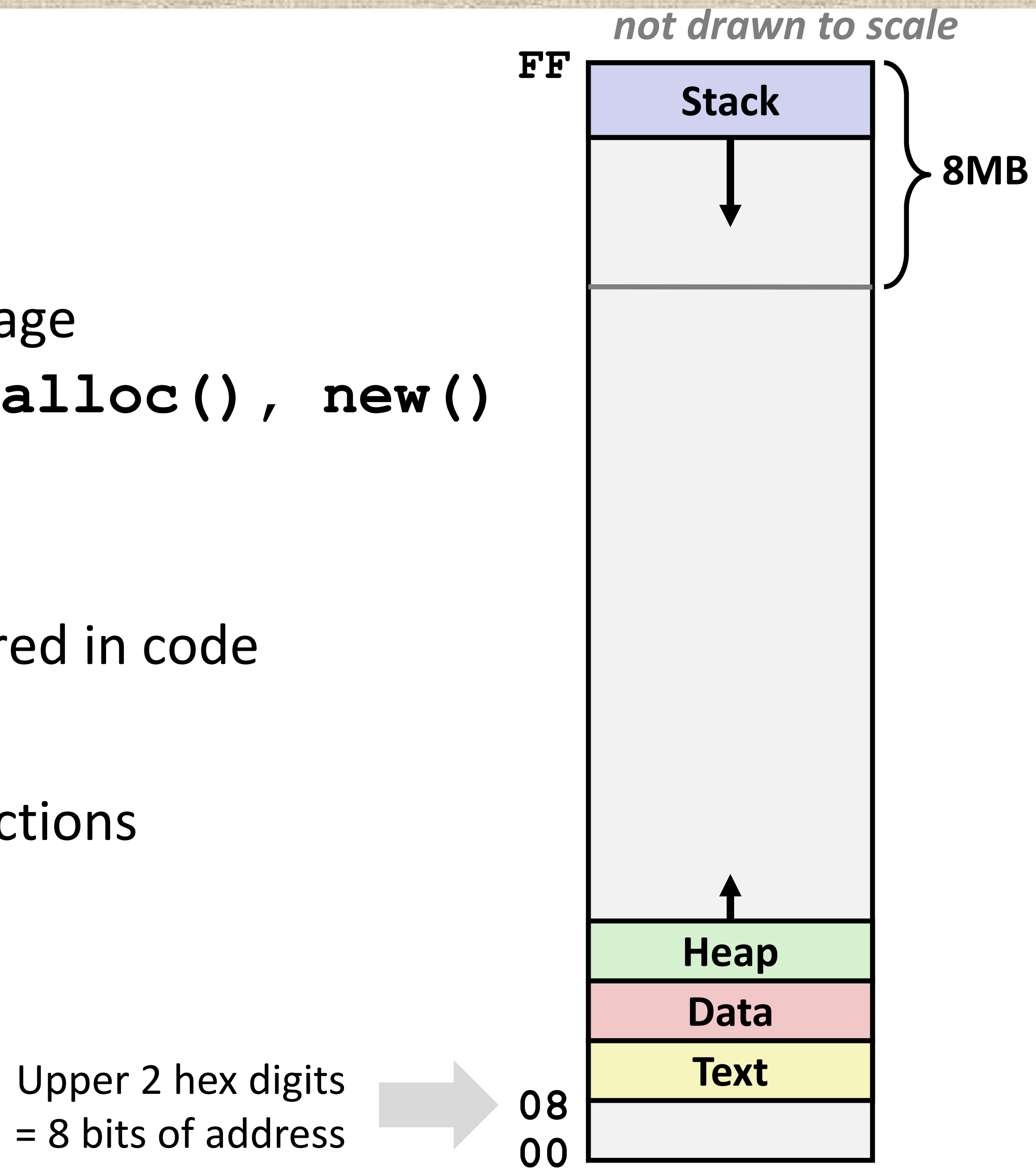
Machine-Level Programming: Miscellaneous Topics

Chapter 3 of B&O

Some notes adopted from Bryant and O'Hallaron

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



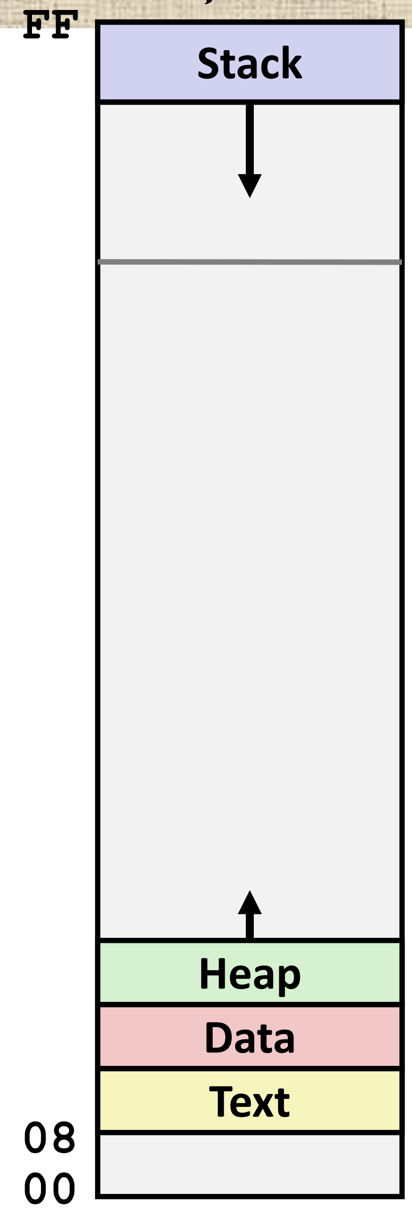
Memory Allocation Example

```
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?

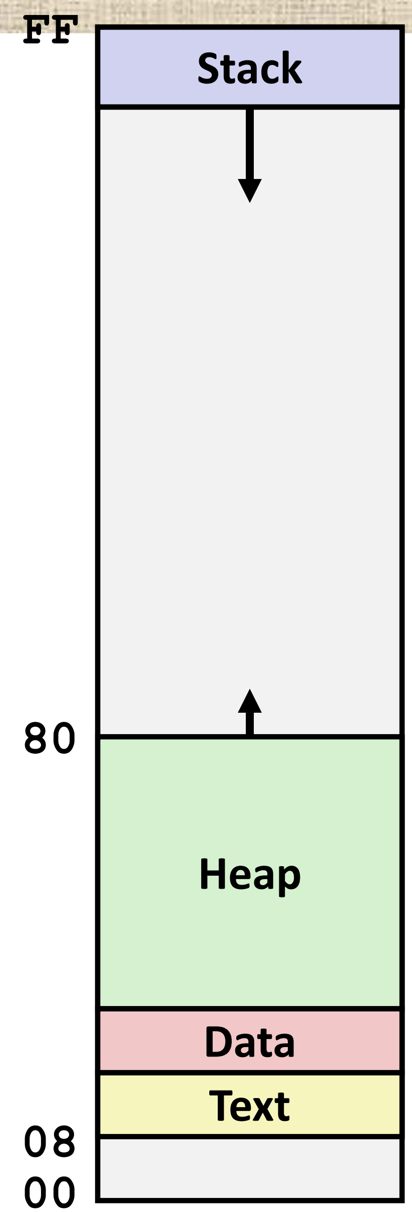
The pointers point to stuff in the heap, but they themselves are in data

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()`	`0x080483c6`
`useless()`	`0x08049744`
final `malloc()`	`0x006be166`

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

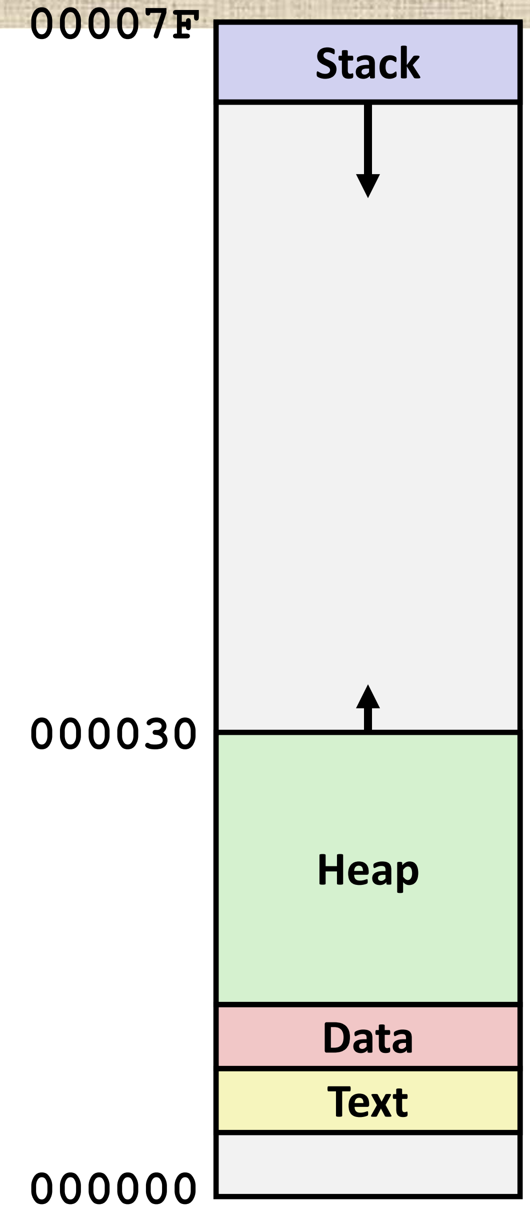


x86-64 Example Addresses

address range ~2^{47}

`$rsp`	`0x7ffffff8d1f8`
`p3`	`0x2aaabaadd010`
`p1`	`0x2aaaaaadc010`
`p4`	`0x000011501120`
`p2`	`0x000011501010`
`&p2`	`0x000010500a60`
`beyond`	`0x000000500a44`
`big_array`	`0x000010500a80`
`huge_array`	`0x000000500a50`
`main()`	`0x000000400510`
`useless()`	`0x000000400500`
final `malloc()`	`0x00386ae6a170`

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



C operators

<table>
<tr><th>Operators</th><th>Associativity</th></tr>
<tr><td>()   []   ->   .</td><td>left to right</td></tr>
<tr><td>!   ~   ++   --   +   -   *   & (type) sizeof</td><td>right to left</td></tr>
<tr><td>*   /   %</td><td>left to right</td></tr>
<tr><td>+   -</td><td>left to right</td></tr>
<tr><td><<   >></td><td>left to right</td></tr>
<tr><td><   <=   >   >=</td><td>left to right</td></tr>
<tr><td>==   !=</td><td>left to right</td></tr>
<tr><td>&</td><td>left to right</td></tr>
<tr><td>^</td><td>left to right</td></tr>
<tr><td>|</td><td>left to right</td></tr>
<tr><td>&&</td><td>left to right</td></tr>
<tr><td>||</td><td>left to right</td></tr>
<tr><td>?:</td><td>right to left</td></tr>
<tr><td>= += -= *= /= %= &= ^= != <<= >>=</td><td>right to left</td></tr>
<tr><td>,</td><td>left to right</td></tr>
</table>

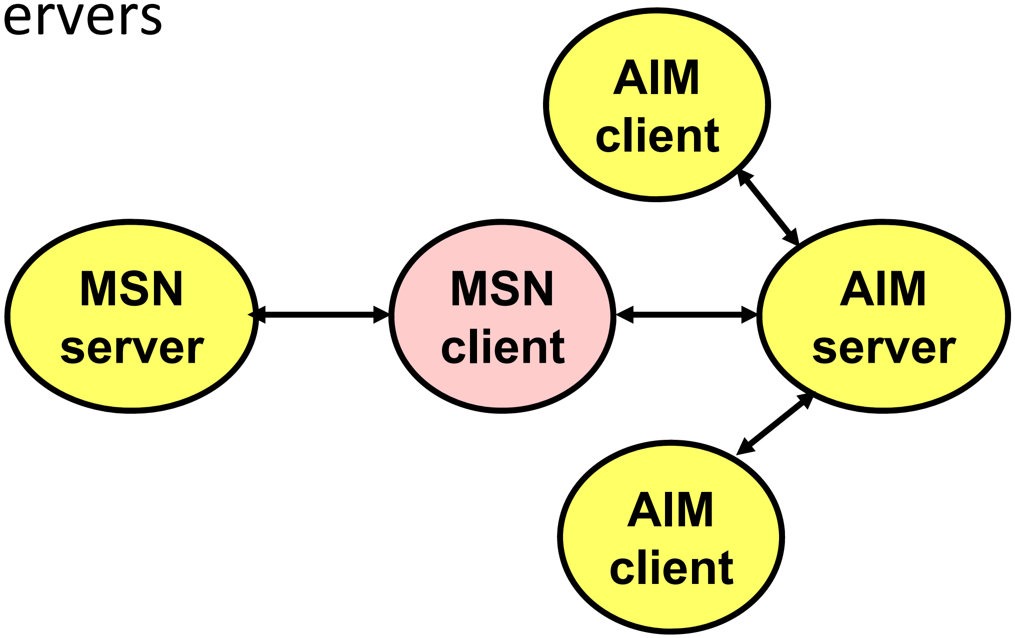
Note: Unary +, –, and * have higher precedence than binary forms

C pointer declarations

`int *p`	p is a pointer to int
`int *p[13]`	p is an array[13] of pointer to int
`int *(p[13])`	p is an array[13] of pointer to int
`int **p`	p is a pointer to a pointer to an int
`int (*p)[13]`	p is a pointer to an array[13] of int
`int *f()`	f is a function returning a pointer to int
`int (*f)()`	f is a pointer to a function returning int
`int (*(*f())[13])()`	f is a function returning ptr to an array[13] of pointers to functions returning int
`int (*(*x[3])())[5]`	x is an array[3] of pointers to functions returning pointers to array[5] of ints

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

String Library Code

- Implementation of Unix function `gets`
  - No way to specify limit on number of characters to read

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

- 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;
}
```

Buffer Overflow Executions

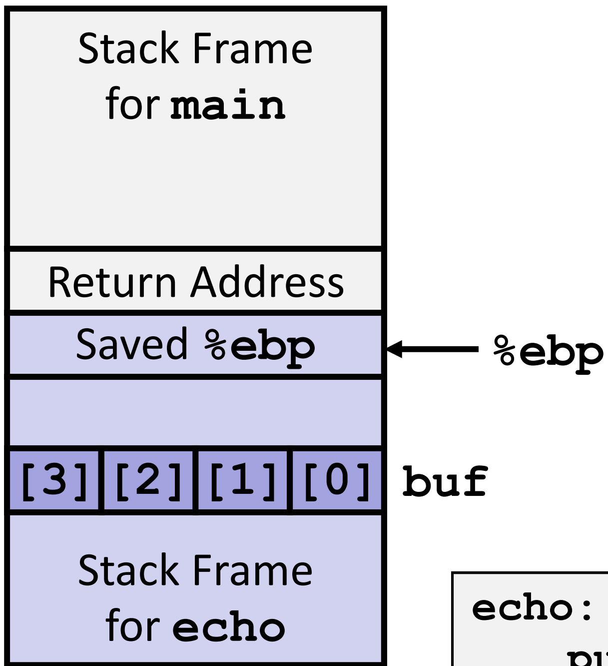
```
unix>./bufdemo
Type a string:123
123
```

```
unix>./bufdemo
Type a string:12345
Segmentation Fault
```

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

Buffer Overflow Stack

Before call to gets

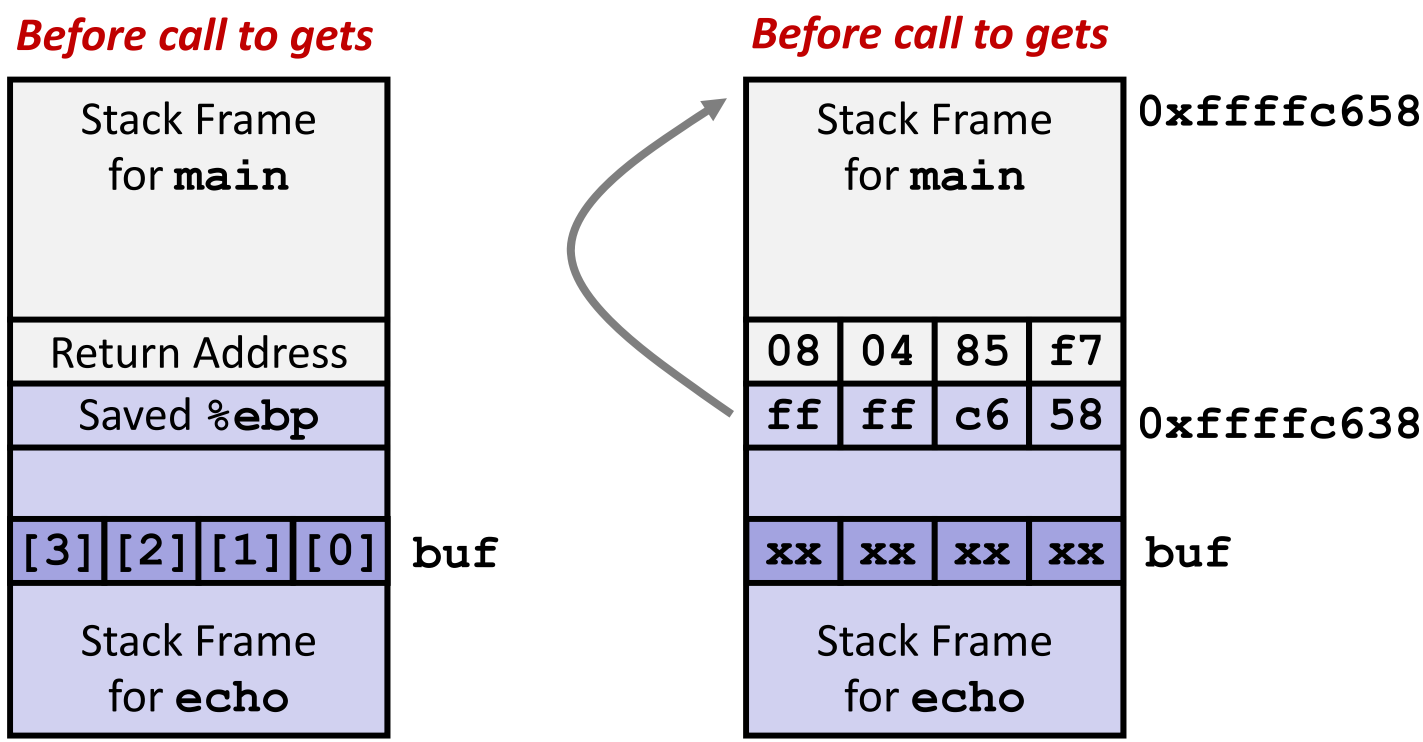



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
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 on stack
    call  gets             # Call gets
    . . .
```

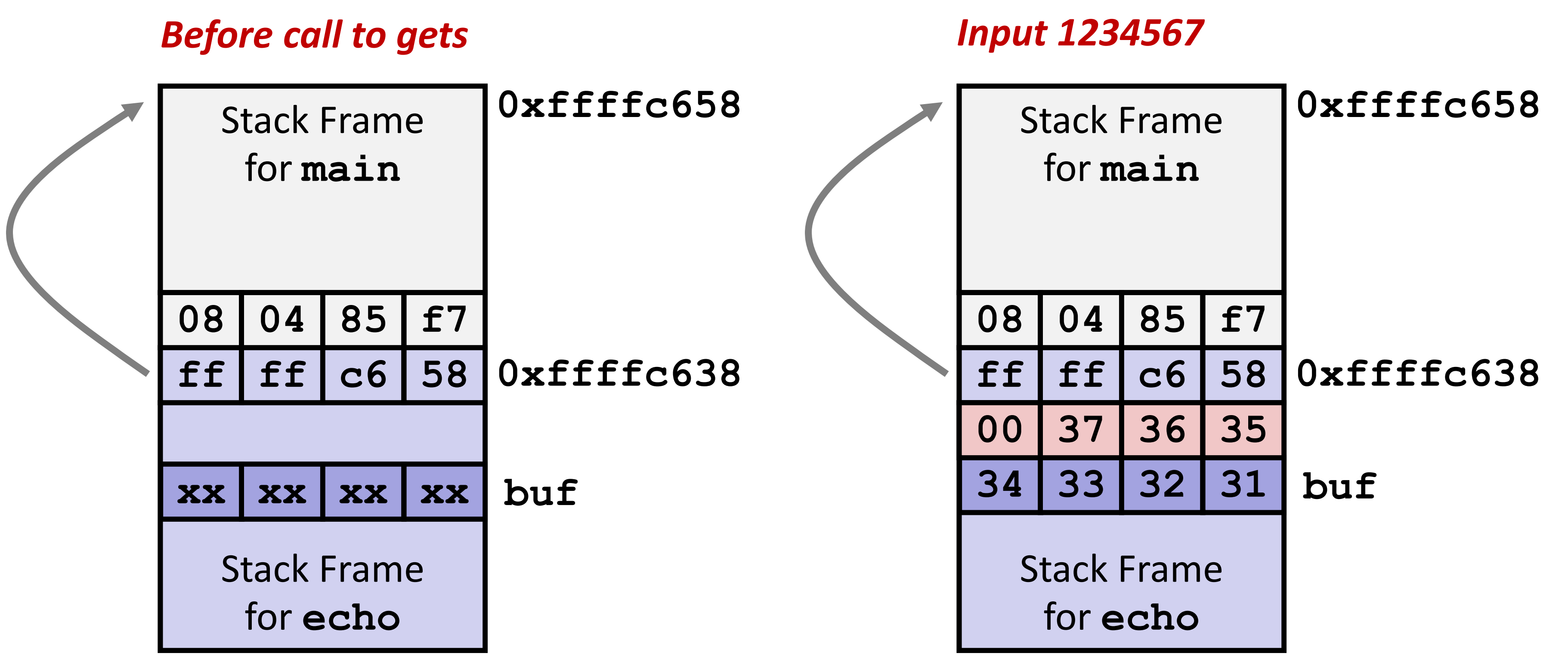
Buffer Overflow Stack Example

```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x $ebp
$1 = 0xffffc638
(gdb) print /x *(unsigned *)$ebp
$2 = 0xffffc658
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x80485f7
```




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

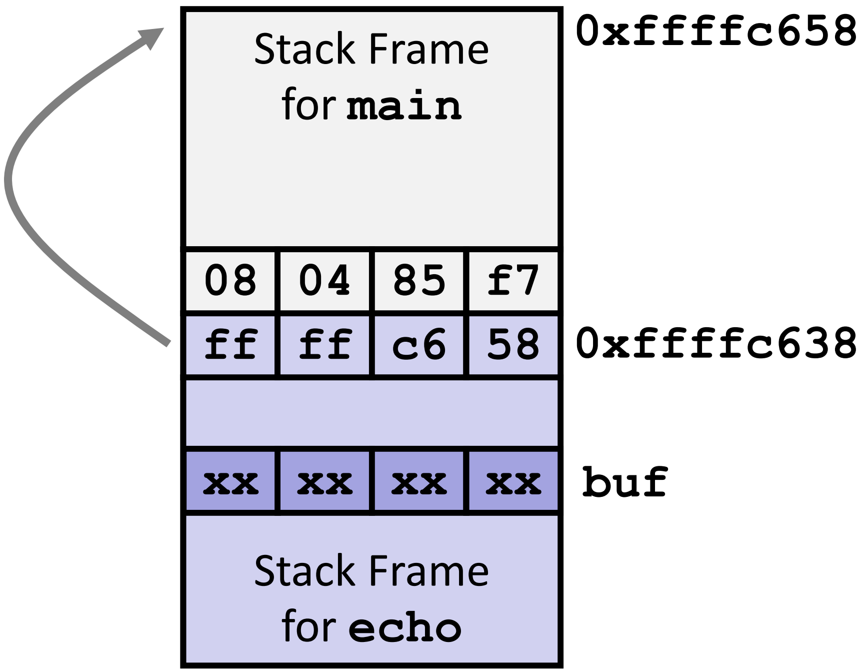
Buffer Overflow Example # 1



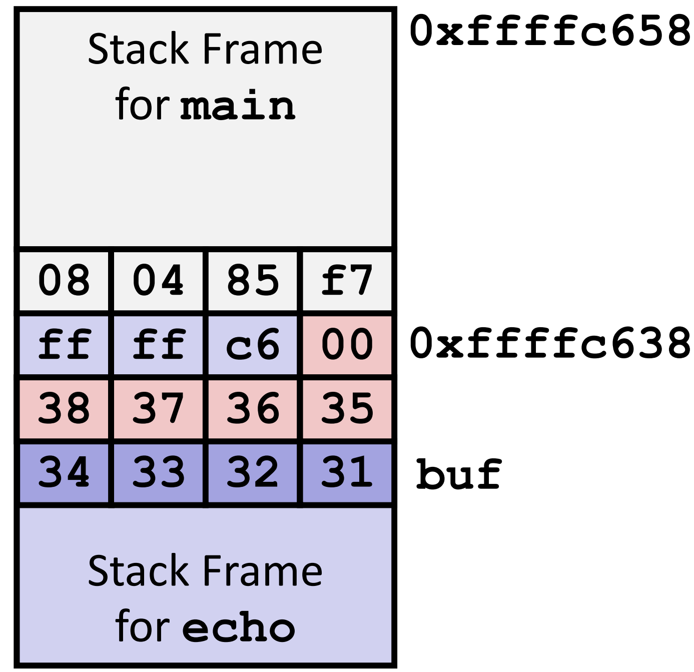
Overflow buf, but no problem

Buffer Overflow Example #2

Before call to gets



Input 12345678

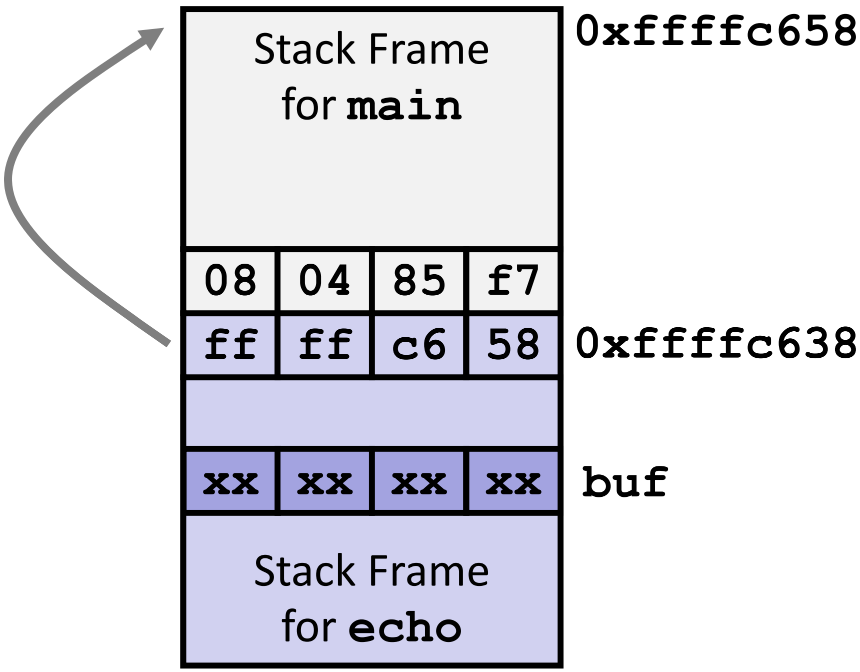


Base pointer corrupted

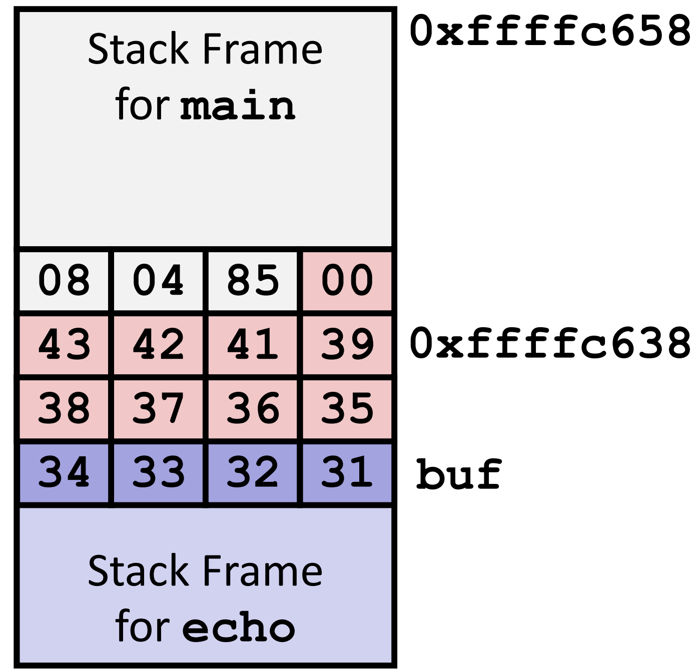
```
. . .
804850a: 83 c4 14   add    $0x14,%esp   # deallocate space
804850d: 5b         pop    %ebx         # restore %ebx
804850e: c9         leave               # movl %ebp, %esp; popl %ebp
804850f: c3         ret                 # Return
```

Buffer Overflow Example #3

Before call to gets



Input 12345678



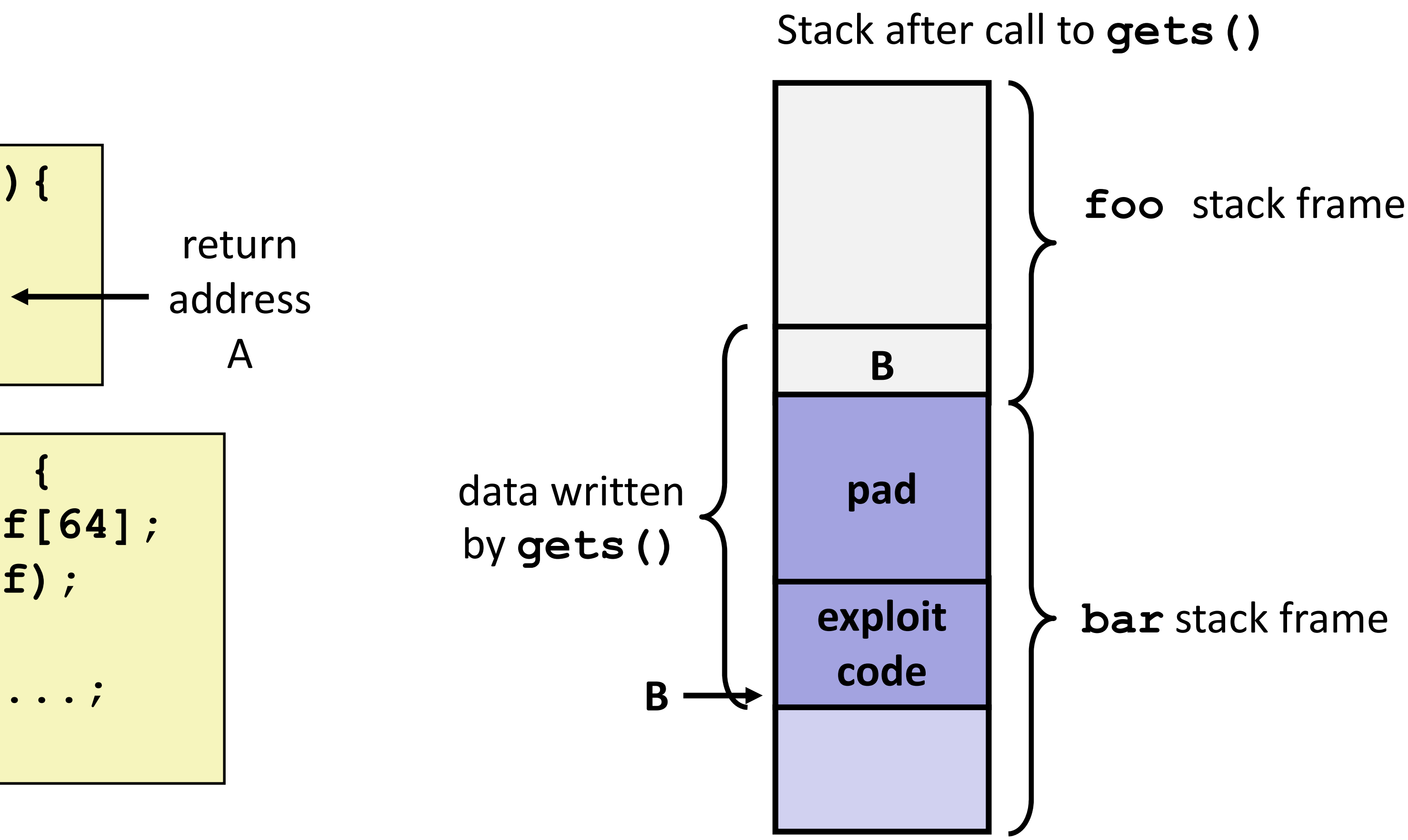
Return address corrupted

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

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 with address of buffer
- 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 (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.

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.

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!

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

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

```
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
```

- Nonexecutable code segments
  - In traditional x86, can mark region of memory as either “read-only” or “writeable”
    - Can execute anything readable
  - Add explicit “execute” permission

Final Observations

- Memory Layout
  - OS/machine dependent (including kernel version)
  - Basic partitioning: stack/data/text/heap/DLL found in most machines
- Type Declarations in C
  - Notation obscure, but very systematic
- Working with Strange Code
  - Important to analyze nonstandard cases
    - E.g., what happens when stack corrupted due to buffer overflow
  - Helps to step through with GDB