Today

- C operators and their precedence
- Memory layout
- Buffer overflow, worms, and viruses

Operator Preference in C (16 levels)

Operators	*Associativity*	
`()` `[]` `->` `.` (postfix versions of `++` `--`)	left to right	16
(prefix versions of `++` `--`) `sizeof`	right to left	15
`!` `~` (unary versions of `+` `-` `&` `*`)	right to left	15
`(type)`	right to left	14
`*` `/` `%`	left to right	13
`+` `-`	left to right	12
`<<` `>>`	left to right	11
`<` `<=` `>` `>=`	left to right	10
`==` `!=`	left to right	9
`&`	left to right	8
`^`	left to right	7
`|`	left to right	6
`&&`	left to right	5
`||`	left to right	4
`?:`	right to left	3
`=` `+=` `-=` `*=` `/=` `%=` `&=` `^=` `!=` `<<=` `>>=`	right to left	2
`,`	left to right	1

++ and --

- Unary increment(++)/decrement(--) operators
  - Prefix (to left, before): --x   decrement first, then use
  - Postfix (to right, after): x++   use first, then increment

```
x = 3;
y = x++;   // y gets 3, then x incremented to 4
z = --x;   // x decremented to 3, then z gets 3
           // x, y, and z all are 3 at end
```

```
int j;
int ni = n*i;
double *rowp = a+ni;
for (j = 0; j < n; j++)
    {*rowp = b[j]; rowp++;}
```

```
int j;
int ni = n*i;
double *rowp = a+ni;
for (j = 0; j < n; j++)
    *rowp++ = b[j];
```


Precedence Examples

```
a*b+c

a-b+c

sizeof(int)*p

*p->q

*x++

a+=b++

a++b

a+++b

a++++b
```

Precedence Examples

`a*b+c`	`(a*b)+c`
`a-b+c`	`(a-b)+c`
`sizeof(int)*p`	`(sizeof(int))*p`
`*p->q`	`*(p->q)`
`*x++`	`*(x++)` not `(*x)++` but increment after use
`a+=b++`	`a+=(b++)` but increment after use
`a++b`	`a+(+b)`
`a+++b`	`(a++)+b` not `a+(++b)` but increment after use
`a++++b`	`(a++)+(+b)` but increment after use

C Pointer Declarations

```
int *p

int *p[13]

int *(p[13])

int **p

int (*p)[13]

int *f()

int (*f)()
```

C Pointer Declarations (Check out guide)

`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

University of Washington

Avoiding Complex Declarations

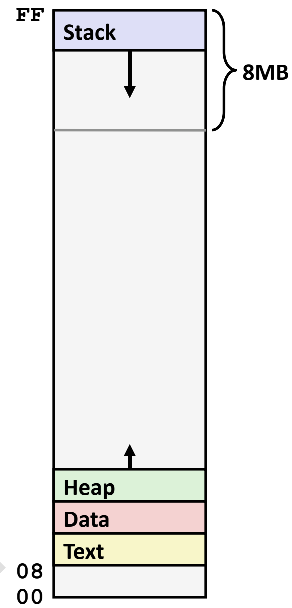
- **Use `typedef` to build up the declaration**

- `int (*(*x[3])())[5];`
  - `x` is an array of 3 elements, each of which is a pointer to a function returning an array of 5 ints
  - `typedef int fiveints[5];`
  - `typedef fiveints* p5i;`
  - `typedef p5i (*f_of_p5is)();`
  - `f_of_p5is x[3];`

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

not drawn to scale



Upper 2 hex digits
= 8 bits of address

Memory Allocation Example

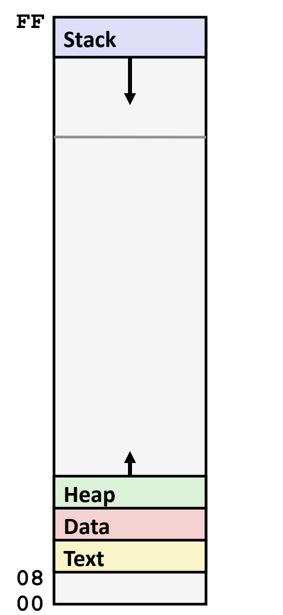
not drawn to scale

```
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?

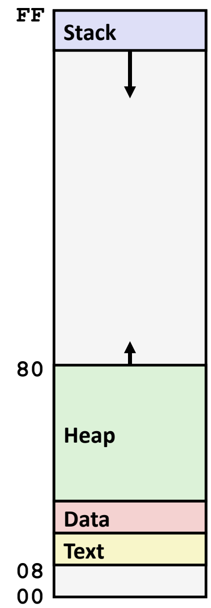
IA32 Example Addresses

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

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

not drawn to scale



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()`**

```
/* 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
```

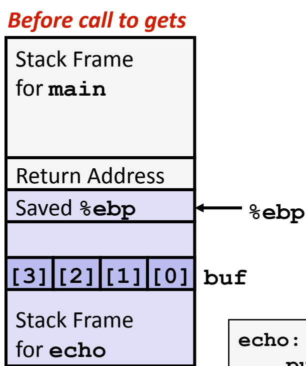
```
unix>./bufdemo
Type a string:123456789ABC
Segmentation Fault
```


Buffer Overflow Disassembly

```
080484f0 <echo>:
 80484f0:   55                  push   %ebp
 80484f1:   89 e5               mov    %esp,%ebp
 80484f3:   53                  push   %ebx
 80484f4:   8d 5d f8            lea    0xfffffff8(%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 0xfffffffc(%ebp),%ebx
 80485fa:   c9                  leave
 80485fb:   31 c0               xor    %eax,%eax
 80485fd:   c3                  ret
```

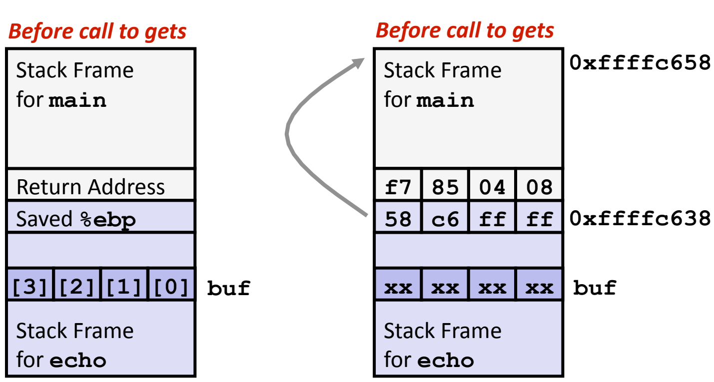
Buffer Overflow Stack




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

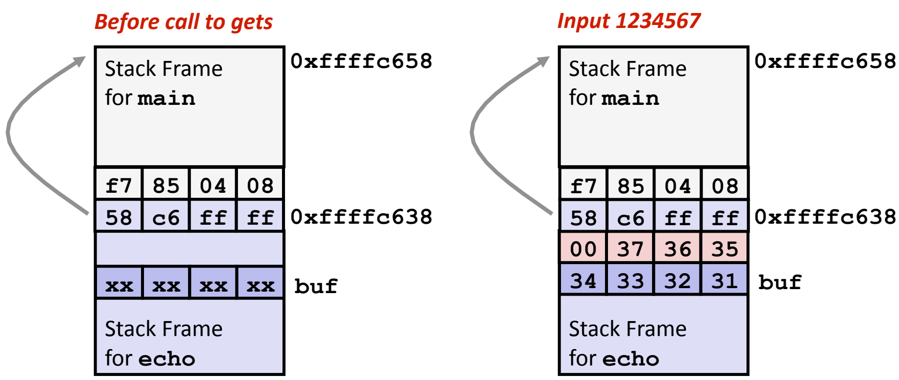
```
echo:
    pushl %ebp              # Save %ebp on stack
    movl  %esp, %ebp
    pushl %ebx              # Save %ebx
    leal  -8(%ebp),%ebx     # Compute buf as %ebp-8
    subl  $20, %esp         # Allocate stack space
    movl  %ebx, (%esp)      # Push buf addr on stack
    call  gets          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


Buffer Overflow Stack Example



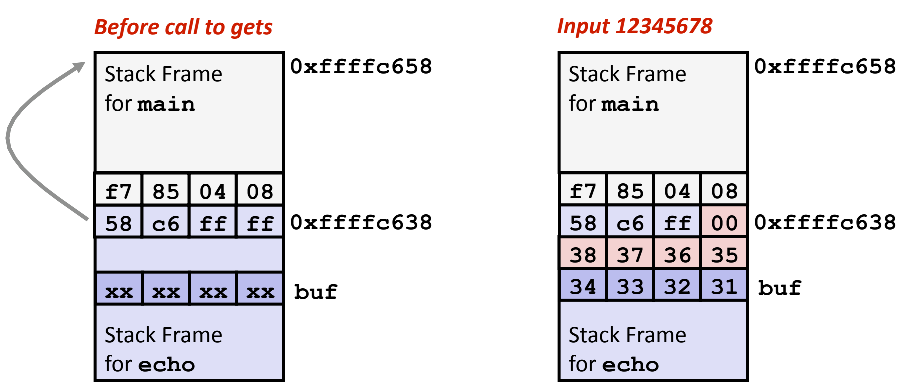

```
80485f2:call 80484f0 <echo>
80485f7:mov  0xfffffffc(%ebp),%ebx # Return Point
```

Buffer Overflow Example #1



Overflow buf, but no problem

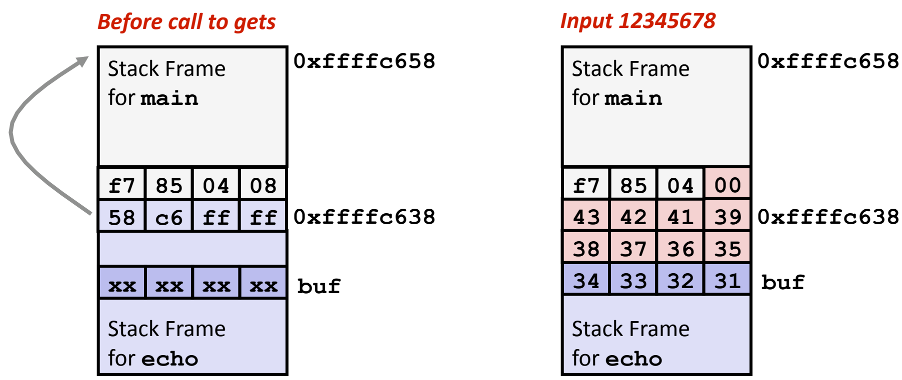
Buffer Overflow Example #2



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



Return address corrupted

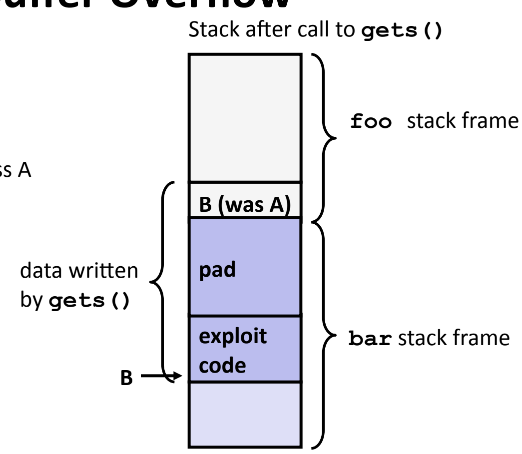
```
80485f2:  call 80484f0 <echo>
80485f7:  mov  0xfffffffc(%ebp),%ebx # Return Point
```


Malicious Use of Buffer Overflow

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

return address A

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



- **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.

Exploits Based on Buffer Overflows

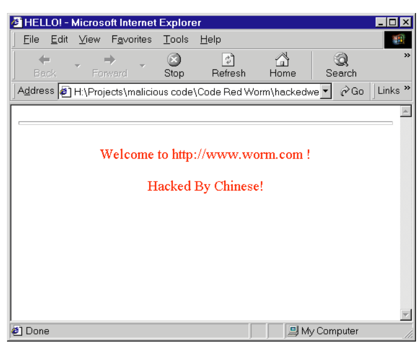
- ***Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines***
- **IM War**
  - AOL exploited existing buffer overflow bug in AIM clients
  - exploit code: returned 4-byte signature (the bytes at some location in the AIM client) to server.
  - When Microsoft changed code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

Code Red 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

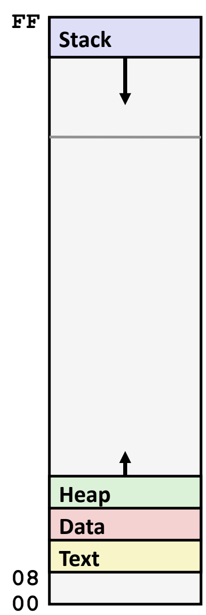
```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small!
*/
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- **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

not drawn to scale



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**