

# 15-213

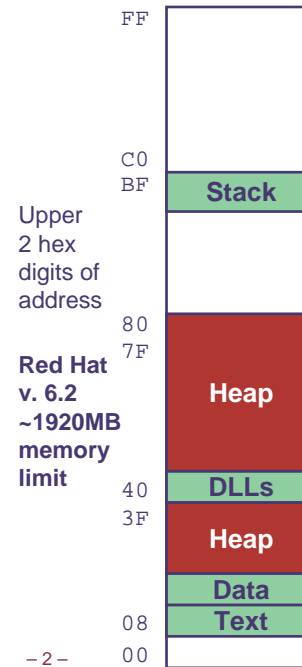
"The course that gives CMU its Zip!"

## Machine-Level Programming V: Miscellaneous Topics Sept. 23, 2003

### Topics

- Linux Memory Layout
- Understanding Pointers
- Buffer Overflow
- Floating Point Code

## Linux Memory Layout



### Stack

- Runtime stack (8MB limit)

### Heap

- Dynamically allocated storage
- When call malloc, calloc, new

### DLLs

- Dynamically Linked Libraries
- Library routines (e.g., printf, malloc)
- Linked into object code when first executed

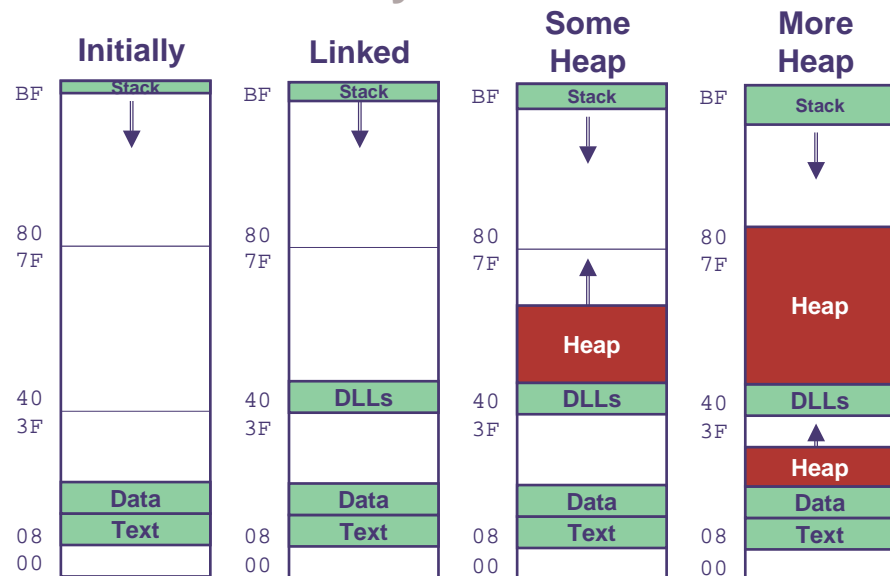
### Data

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

### Text

- Executable machine instructions
- Read-only

## Linux Memory Allocation



## Text & Stack Example

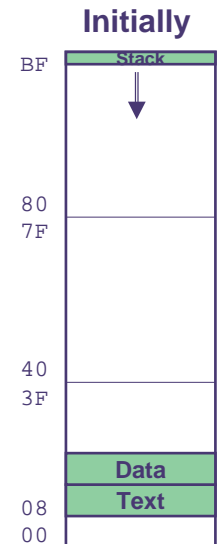
```
(gdb) break main
(gdb) run
Breakpoint 1, 0x804856f in main ()
(gdb) print $esp
$3 = (void *) 0xbfffc78
```

### Main

- Address 0x804856f should be read 0x0804856f

### Stack

- Address 0xbfffc78



# Dynamic Linking Example

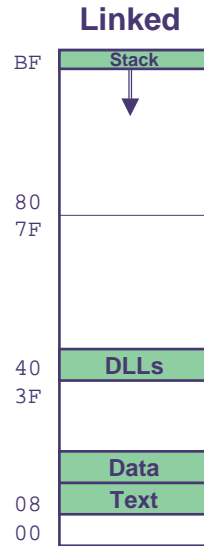
```
(gdb) print malloc
$1 = {<text variable, no debug info>}
0x8048454 <malloc>
(gdb) run
Program exited normally.
(gdb) print malloc
$2 = {void *(unsigned int)}
0x40006240 <malloc>
```

## Initially

- Code in text segment that invokes dynamic linker
- Address 0x8048454 should be read 0x08048454

## Final

- Code in DLL region



# Memory Allocation Example

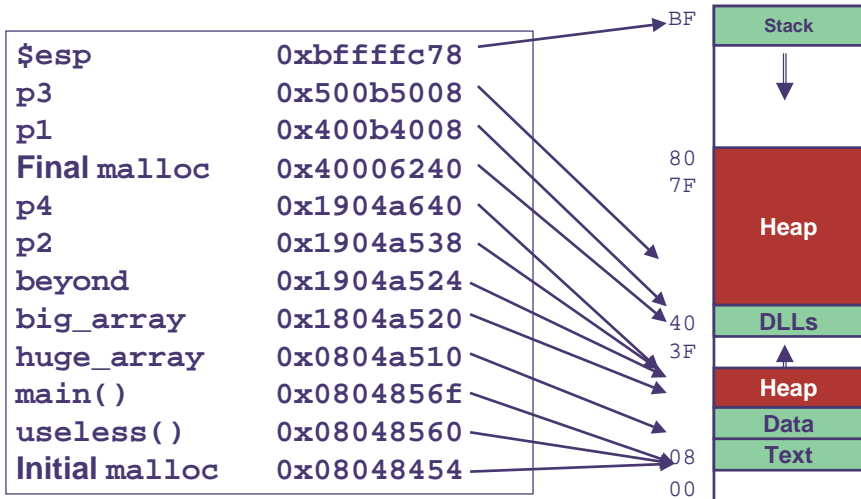
```
char big_array[1<<24]; /* 16 MB */
char huge_array[1<<28]; /* 256 MB */

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 ... */
}
```

# Example Addresses



# C operators

## Operators

```
() [] -> .
! ~ ++ -- + - * & (type) sizeof
* / %
+ -
<< >>
< <= > >=
== !=
&
^
|
&&
||
?:
= += -= *= /= %= &= ^= != <<= >>=
,
```

## Associativity

```
left to right
right to left
left to right
left to right
left to right
left to right
left to right
left to right
left to right
right to left
right to left
left to right
```

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

## C pointer declarations

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

- 9 -

15-213, F'03

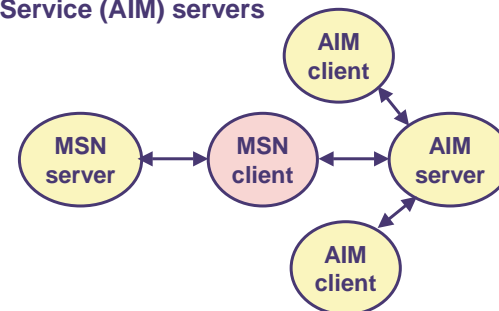
## Internet Worm and IM War

### 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



- 10 -

15-213, F'03

## 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.

- 11 -

15-213, F'03

## 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

- 12 -

15-213, F'03

# 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;
}
```

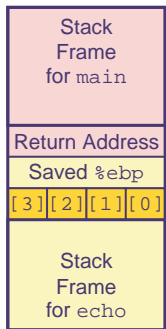
# Buffer Overflow Executions

```
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

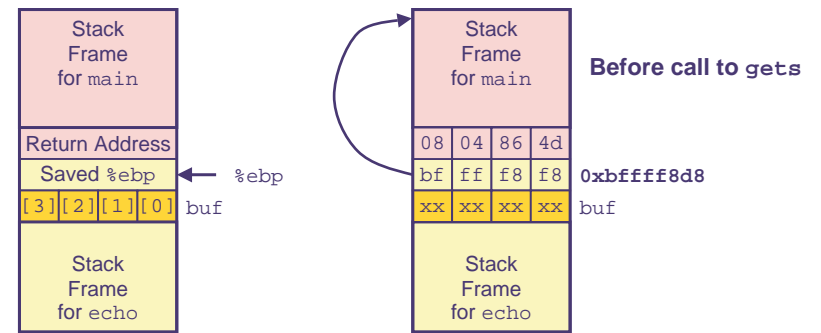


```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushl %ebp          # Save %ebp on stack
    movl %esp,%ebp
    subl $20,%esp      # Allocate space on stack
    pushl %ebx         # Save %ebx
    addl $-12,%esp     # Allocate space on stack
    leal -4(%ebp),%ebx # Compute buf as %ebp-4
    pushl %ebx         # Push buf on stack
    call gets          # Call gets
    . . .
```

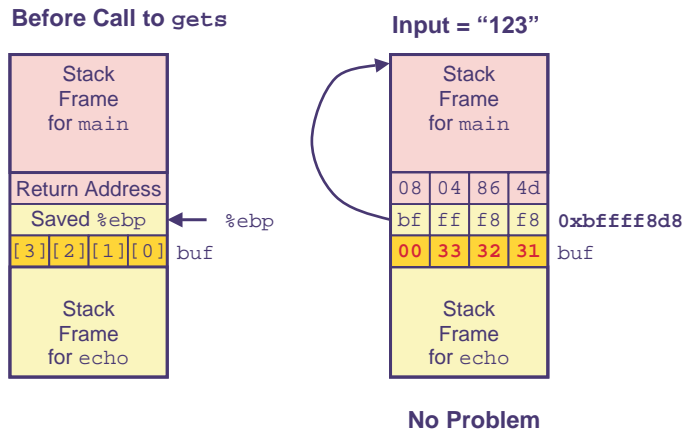
# Buffer Overflow Stack Example

```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x *(unsigned *)$ebp
$1 = 0xbffff8f8
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x804864d
```

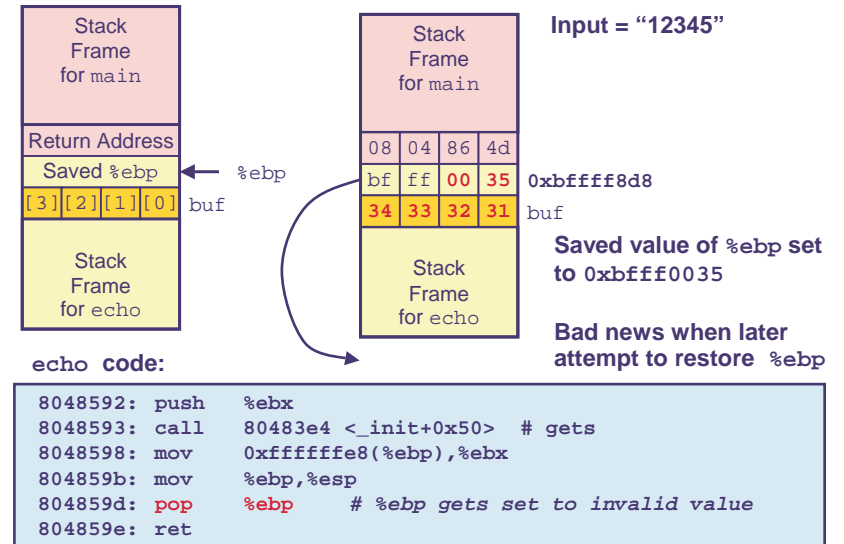


```
8048648: call 804857c <echo>
804864d: mov 0xfffffe8(%ebp),%ebx # Return Point
```

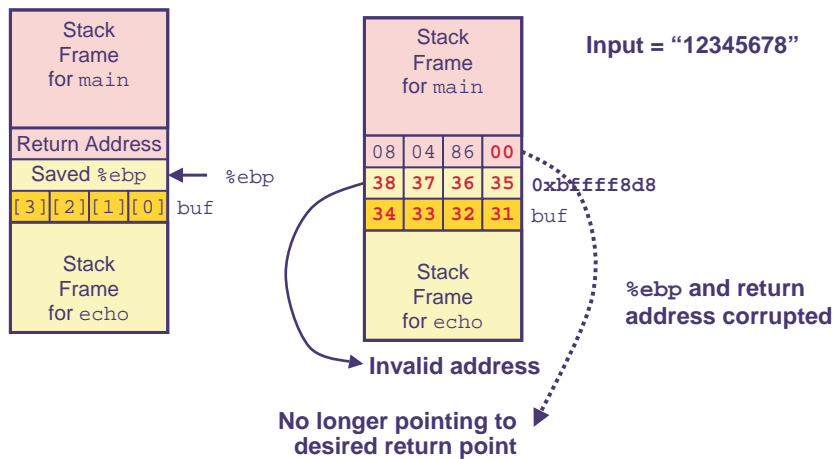
# Buffer Overflow Example #1



# Buffer Overflow Stack Example #2



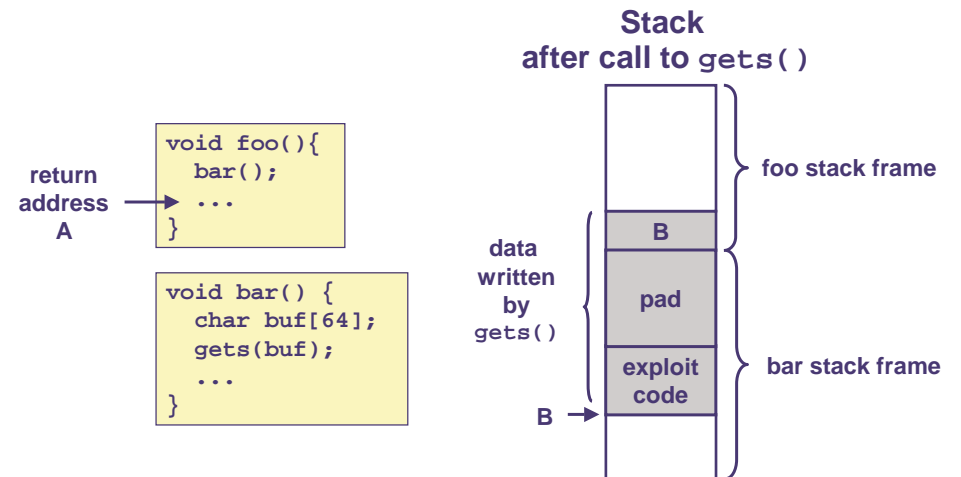
# Buffer Overflow Stack Example #3



```

8048648: call 804857c <echo>
804864d: mov  0xffffffe8(%ebp),%ebx # Return Point
    
```

# Malicious Use of Buffer Overflow

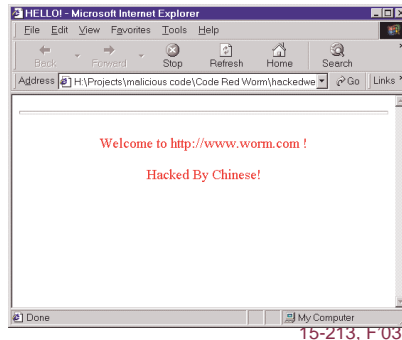


- Input string contains byte representation of executable code
- Overwrite return address with address of buffer
- When bar() executes ret, will jump to exploit code



# 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](http://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



- 25 -

15-213, F'03

# Code Red Effects

## Later Version Even More Malicious

- Code Red II
- As of April, 2002, over 18,000 machines infected
- Still spreading

## Paved Way for NIMDA

- Variety of propagation methods
- One was to exploit vulnerabilities left behind by Code Red II

- 26 -

15-213, F'03

# 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

- 27 -

15-213, F'03

# IA32 Floating Point

## History

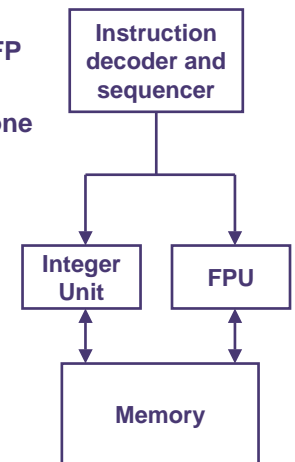
- 8086: first computer to implement IEEE FP
  - separate 8087 FPU (floating point unit)
- 486: merged FPU and Integer Unit onto one chip

## Summary

- Hardware to add, multiply, and divide
- Floating point data registers
- Various control & status registers

## Floating Point Formats

- single precision (C `float`): 32 bits
- double precision (C `double`): 64 bits
- extended precision (C `long double`): 80 bits



- 28 -

15-213, F'03

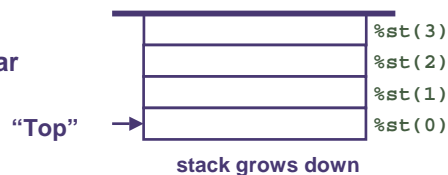
# FPU Data Register Stack

## FPU register format (extended precision)



## FPU registers

- 8 registers
- Logically forms shallow stack
- Top called %st(0)
- When push too many, bottom values disappear



# FPU instructions

## Large number of floating point instructions and formats

- ~50 basic instruction types
- load, store, add, multiply
- sin, cos, tan, arctan, and log!

## Sample instructions:

Instruction	Effect	Description
<code>fldz</code>	push 0.0	Load zero
<code>flds Addr</code>	push M[Addr]	Load single precision real
<code>fmuls Addr</code>	%st(0) <- %st(0)*M[Addr]	Multiply
<code>faddp</code>	%st(1) <- %st(0)+%st(1); pop	Add and pop

# Floating Point Code Example

## Compute Inner Product of Two Vectors

- Single precision arithmetic
- Common computation

```
float ipf (float x[],
          float y[],
          int n)
{
    int i;
    float result = 0.0;

    for (i = 0; i < n; i++) {
        result += x[i] * y[i];
    }
    return result;
}
```

```
pushl %ebp                # setup
movl %esp,%ebp
pushl %ebx

movl 8(%ebp),%ebx        # %ebx=&x
movl 12(%ebp),%ecx       # %ecx=&y
movl 16(%ebp),%edx       # %edx=n
fldz                    # push +0.0
xorl %eax,%eax          # i=0
cmpl %edx,%eax          # if i>=n done
jge .L3

.L5:
flds (%ebx,%eax,4)       # push x[i]
fmuls (%ecx,%eax,4)     # st(0)*=y[i]
faddp                    # st(1)+=st(0); pop
incl %eax                # i++
cmpl %edx,%eax          # if i<n repeat
jl .L5

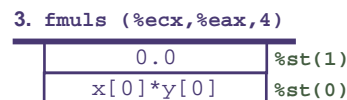
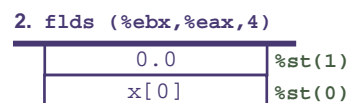
.L3:
movl -4(%ebp),%ebx      # finish
movl %ebp, %esp
popl %ebp
ret                      # st(0) = result
```

# Inner Product Stack Trace

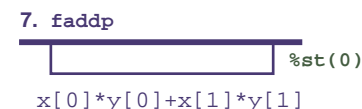
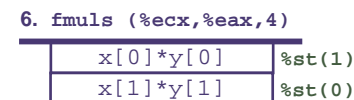
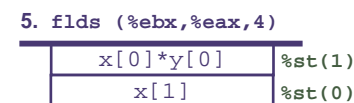
## Initialization



## Iteration 0



## Iteration 1





# 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

## IA32 Floating Point

- Strange “shallow stack” architecture