Machine-Level Prog. V – Miscellaneous Topics

Today

- Buffer overflow
- Floating point code

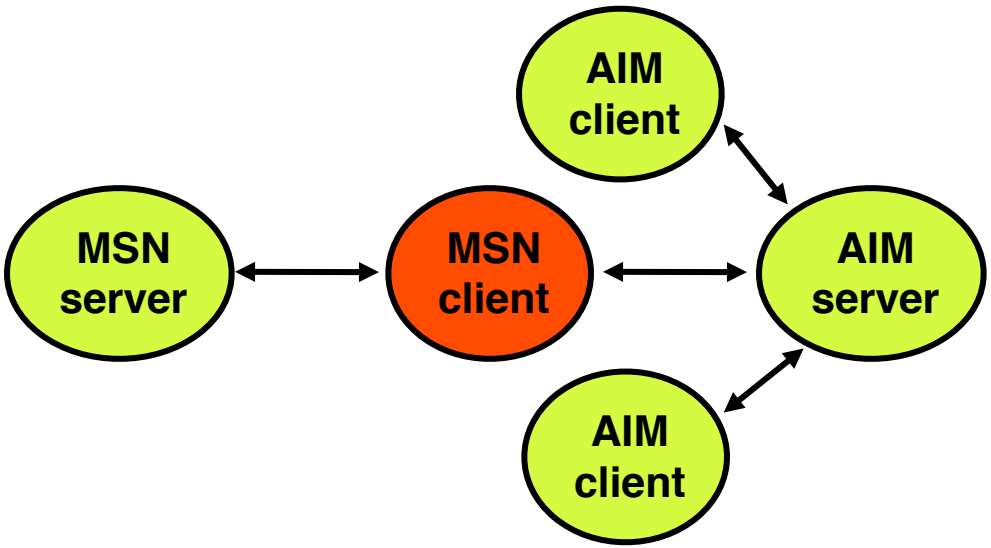
Next time

- Memory

Chris Riesbeck, Spring 2010
Original: Fabian Bustamante

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;
}
```

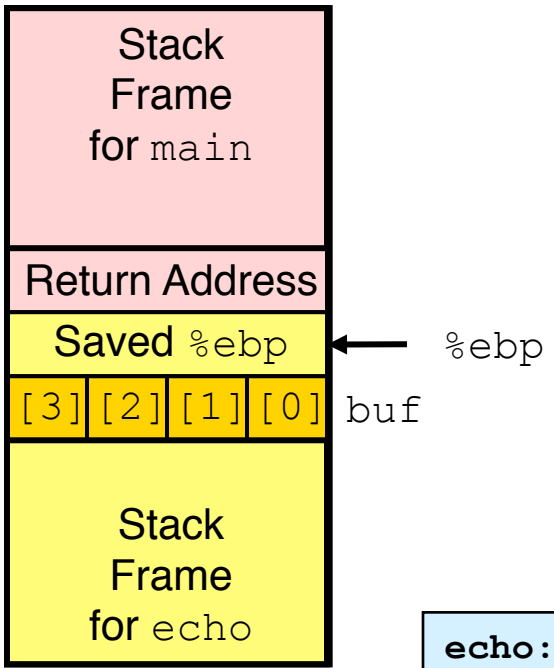
Buffer overflow executions

```
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

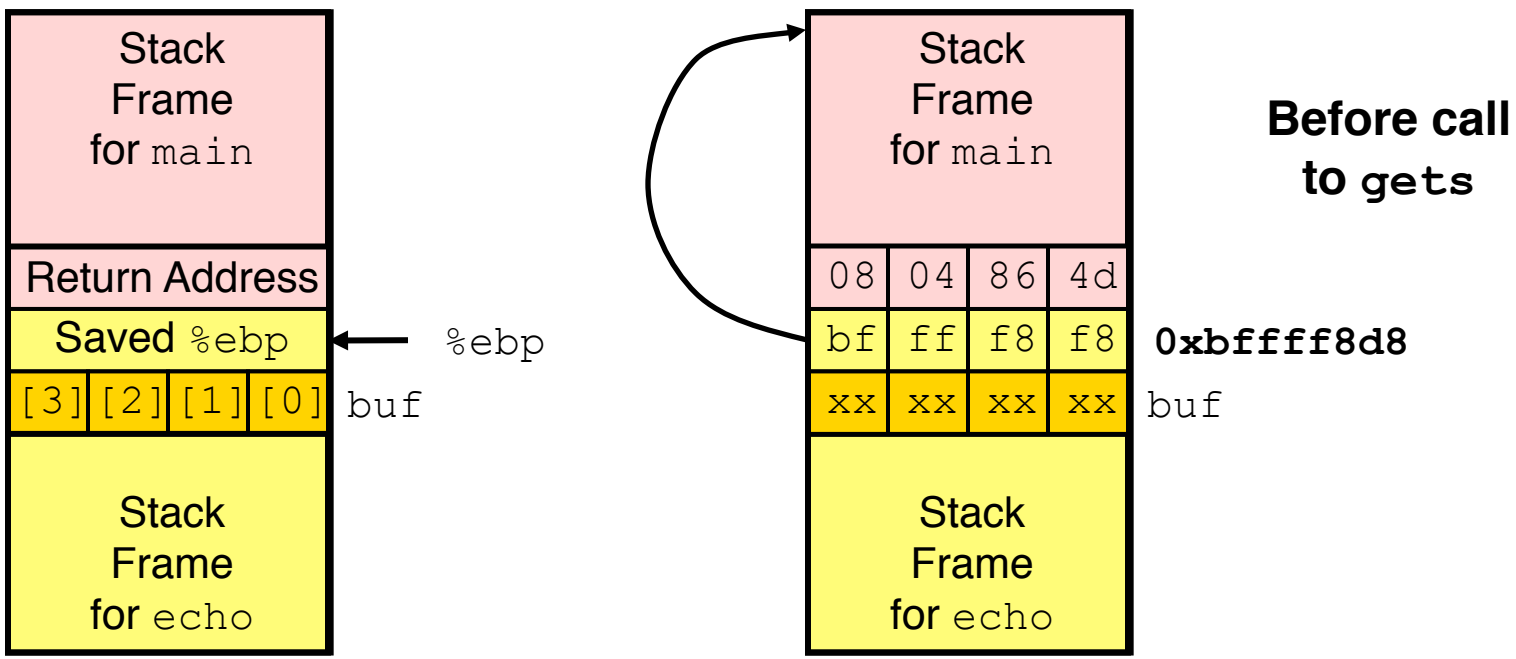



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
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
    . . .
```

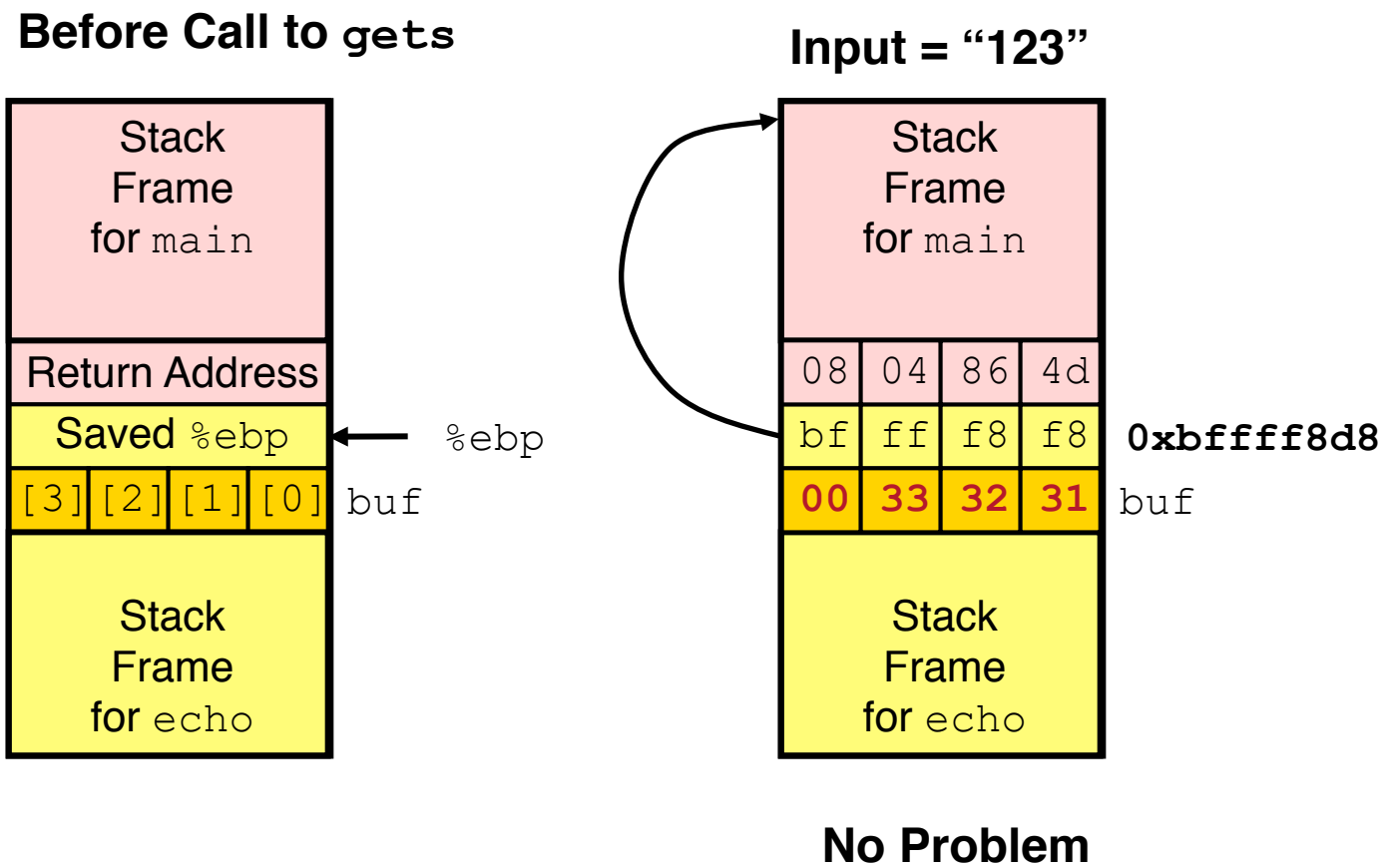
Buffer overflow stack example

```
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
```

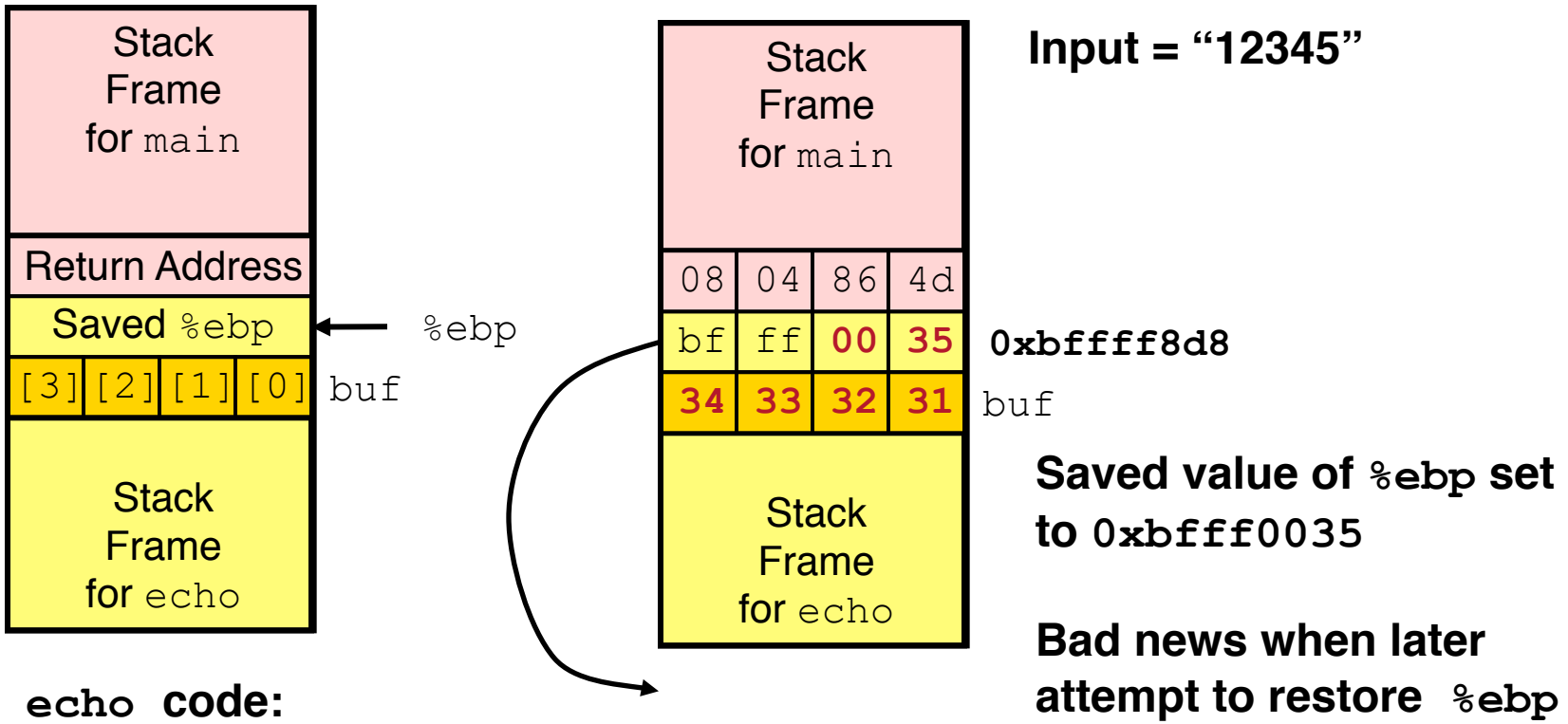



```
8048648:  call 804857c <echo>
804864d:  mov  0xffffffe8(%ebp),%ebx # Return Point
```

Buffer overflow example #1



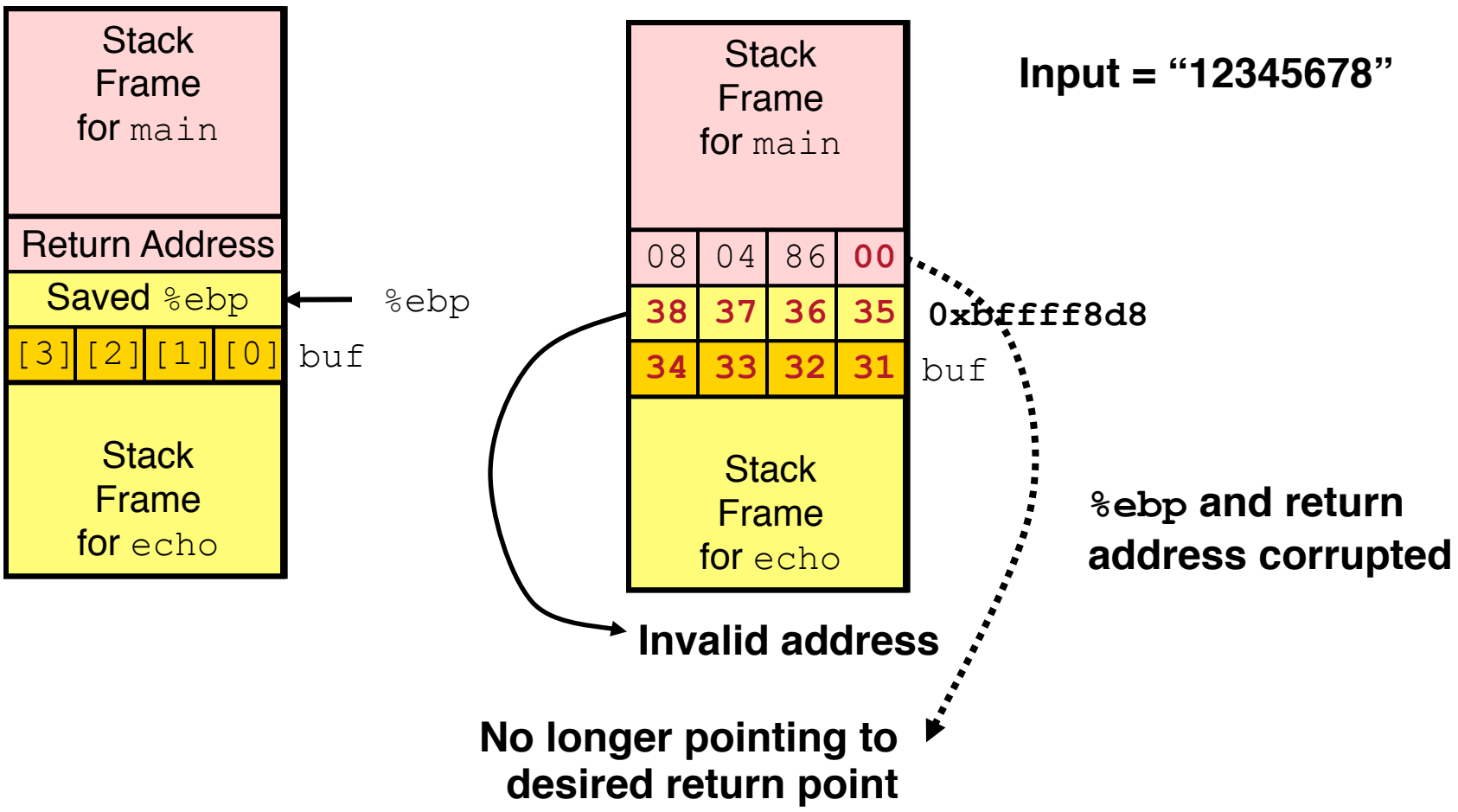
Buffer overflow stack example #2



echo code:

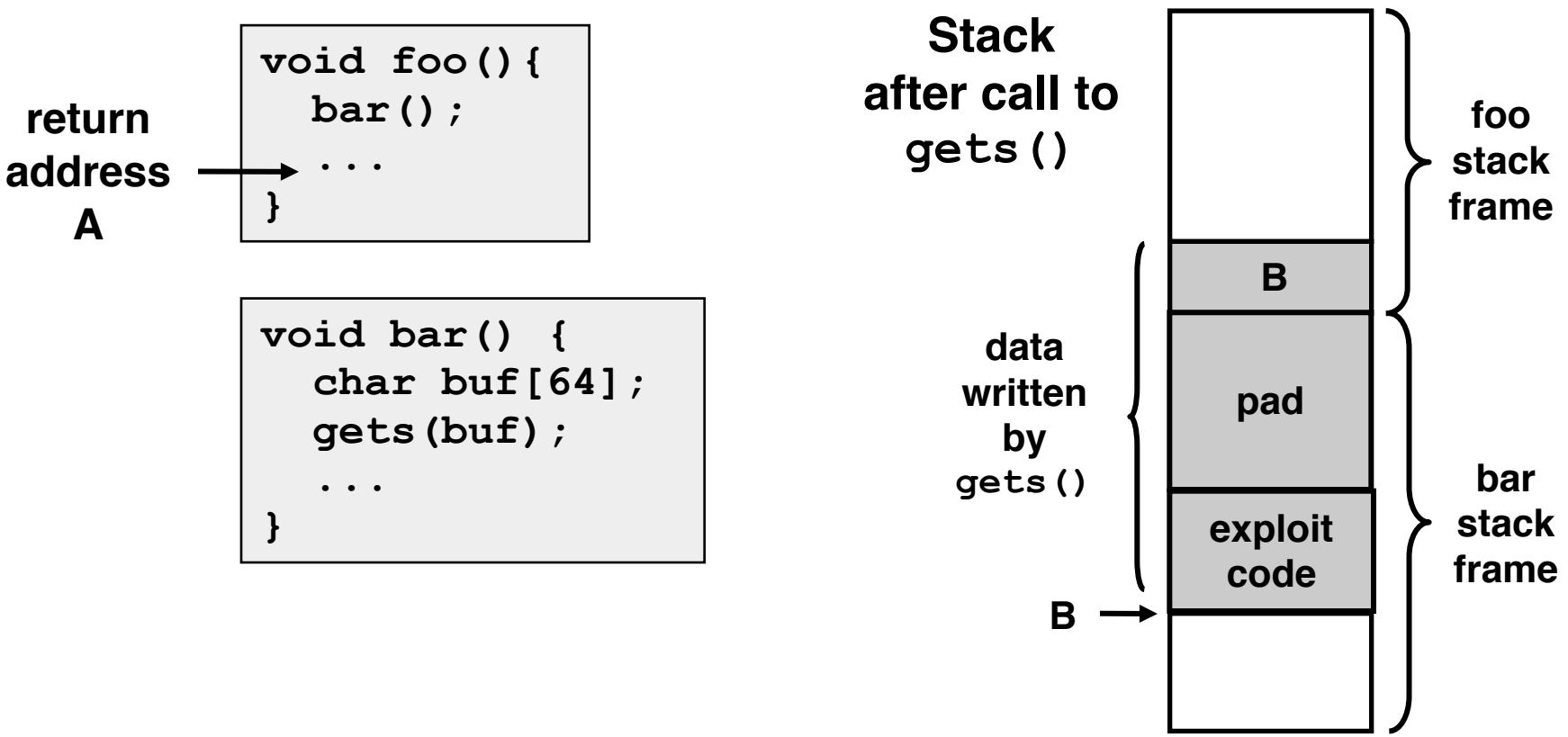
```
8048592:  push   %ebx
8048593:  call   80483e4 <_init+0x50>  # gets
8048598:  mov    0xfffffff8(%ebp),%ebx
804859b:  mov    %ebp,%esp
804859d:  pop    %ebp     # %ebp gets set to invalid value
804859e:  ret
```

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

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.

Email from a supposed consultant

```
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
```

Later determined to be from MS

Avoiding overflow vulnerability

- 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

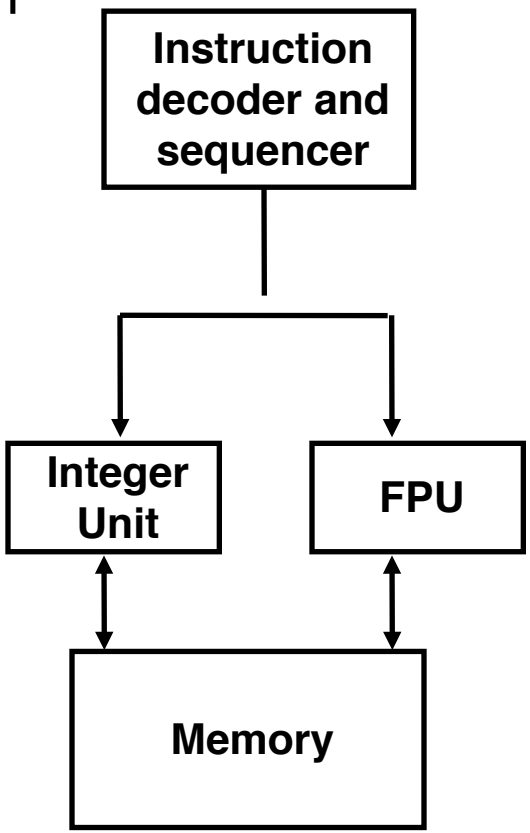
```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

IA32 floating point

- Note: the Floating Point textbook material is available as a “web-aside” at the textbook site.
- Book home page:
  - http://csapp.cs.cmu.edu/
- Web asides:
  - http://csapp.cs.cmu.edu/public/waside.html
- Floating point aside
  - http://csapp.cs.cmu.edu/public/waside/waside-x87.pdf

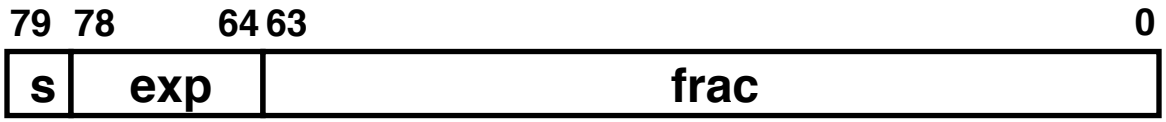
IA32 floating point

- 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



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 & formats
  - ~50 basic instruction types
  - load, store, add, multiply
  - sin, cos, tan, arctan, and log!
- Sample instructions:

Instruction	Effect	Description
`fldz`	push 0.0	Load zero
`flds Addr`	push M[Addr]	Load single precision real
`fmuls Addr`	%st(0) <- %st(0)*M[Addr]	Multiply
`faddp`	%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

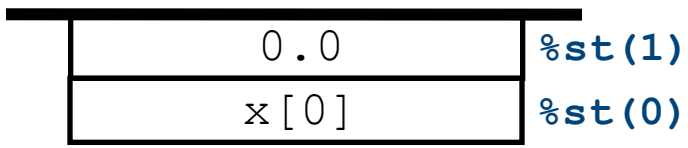
Initialization

%ebx=&x
%ecx=&y

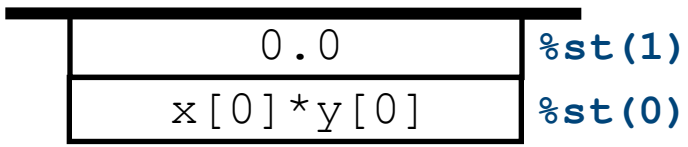
1. `fldz`

Iteration 0

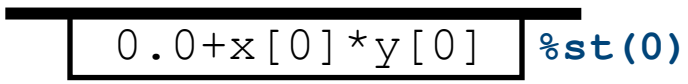
2. `flds (%ebx,%eax,4)`



3. `fmuls (%ecx,%eax,4)`

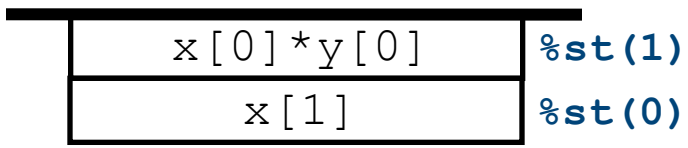


4. `faddp`

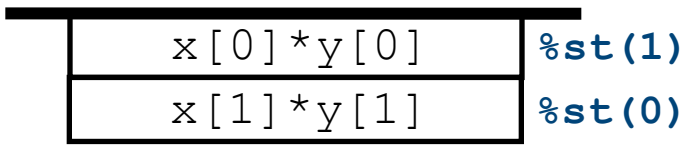


Iteration 1

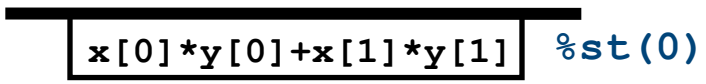
5. `flds (%ebx,%eax,4)`



6. `fmuls (%ecx,%eax,4)`



7. `faddp`



Final observations

- 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