Machine-Level Prog. V – Miscellaneous Topics

Today

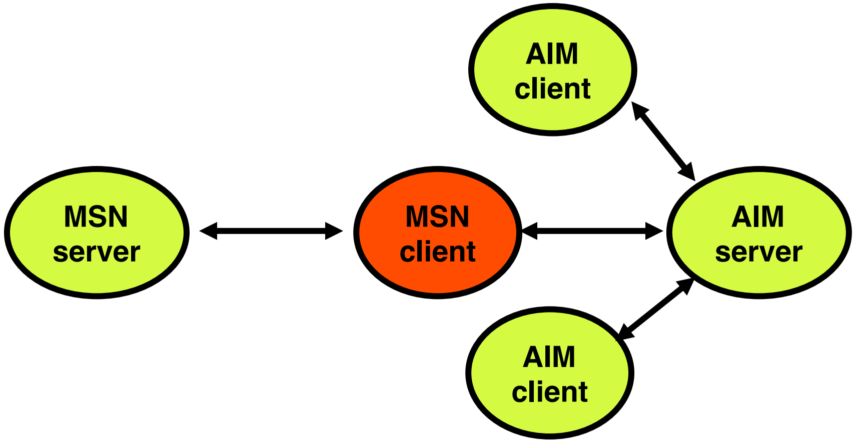
- Buffer overflow
- Extending IA32 to 64 bits

Next time

- Memory

Internet worm and IM war

- November, 1988
  – Internet Worm attacks thousands of Internet hosts.
  – How did it happen? Three ways to spread
    - Copy itself into trusted hosts through rexec/rsh
    - Use sendmail to propagate, through a hole in its debug mode
    - *And the most effective?*
- July, 1999
  – Microsoft launches MSN Messenger (IM 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 *s)
{
    int c;
    char *dest = s;
    int gotchar = 0;
    while ((c = getchar()) != ‘\n’ && c != EOF) {
        *dest++ = c;
        gotchar = 1;
    }
    *dest++ = '\0';
    if *c == EOF && !gotchar)
     return NULL;
    return s;
}
```

No bounds checking!

- 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;
}
```

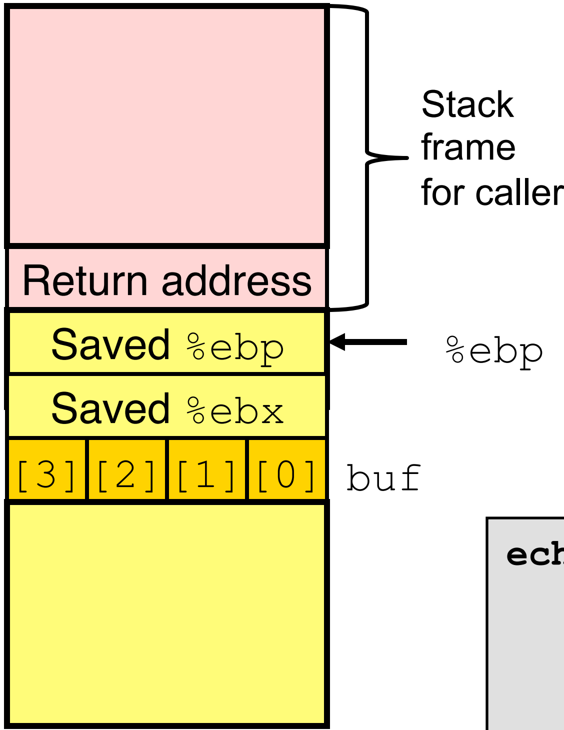
Buffer overflow executions

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

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

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

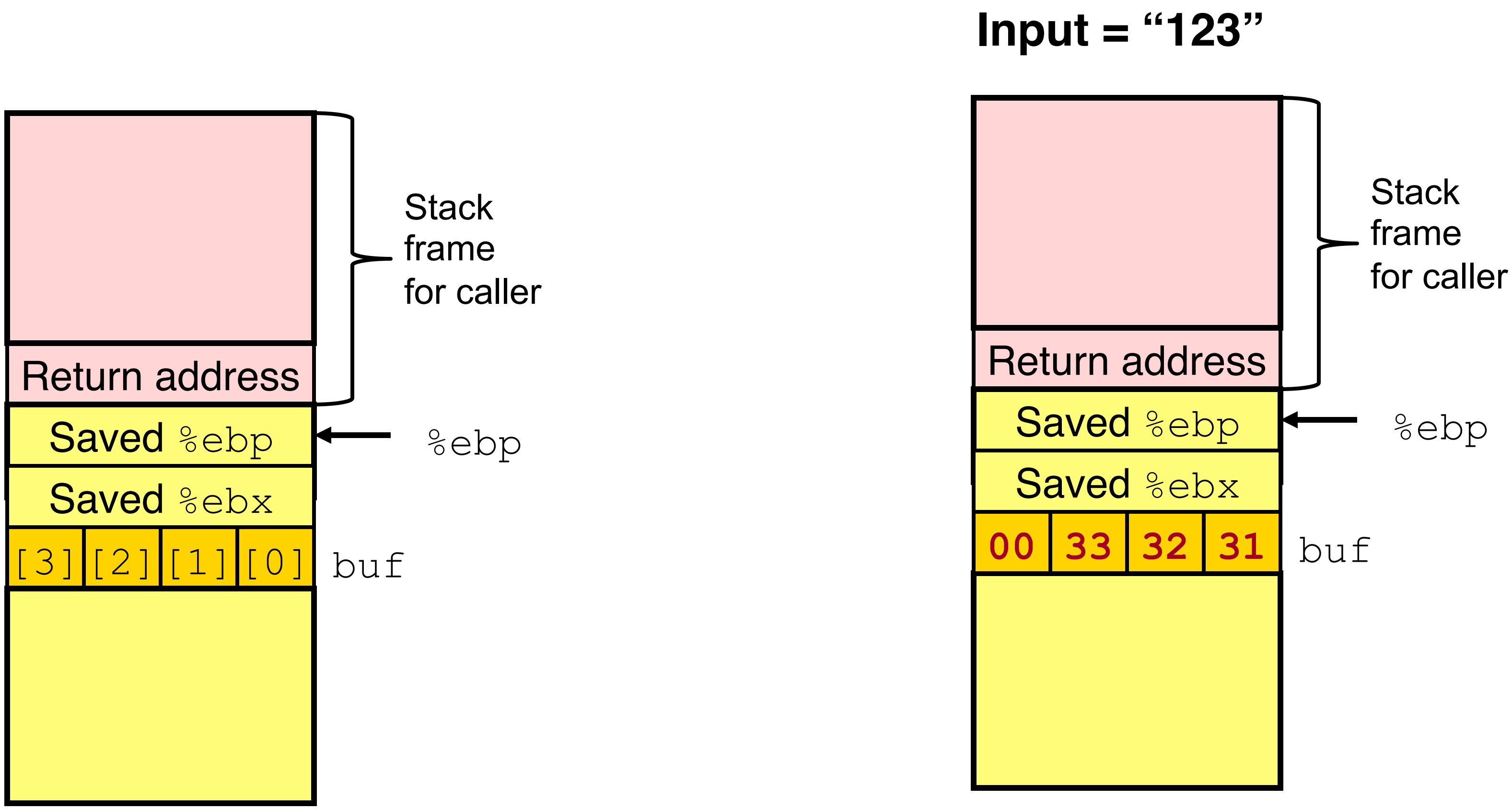
Buffer overflow stack




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

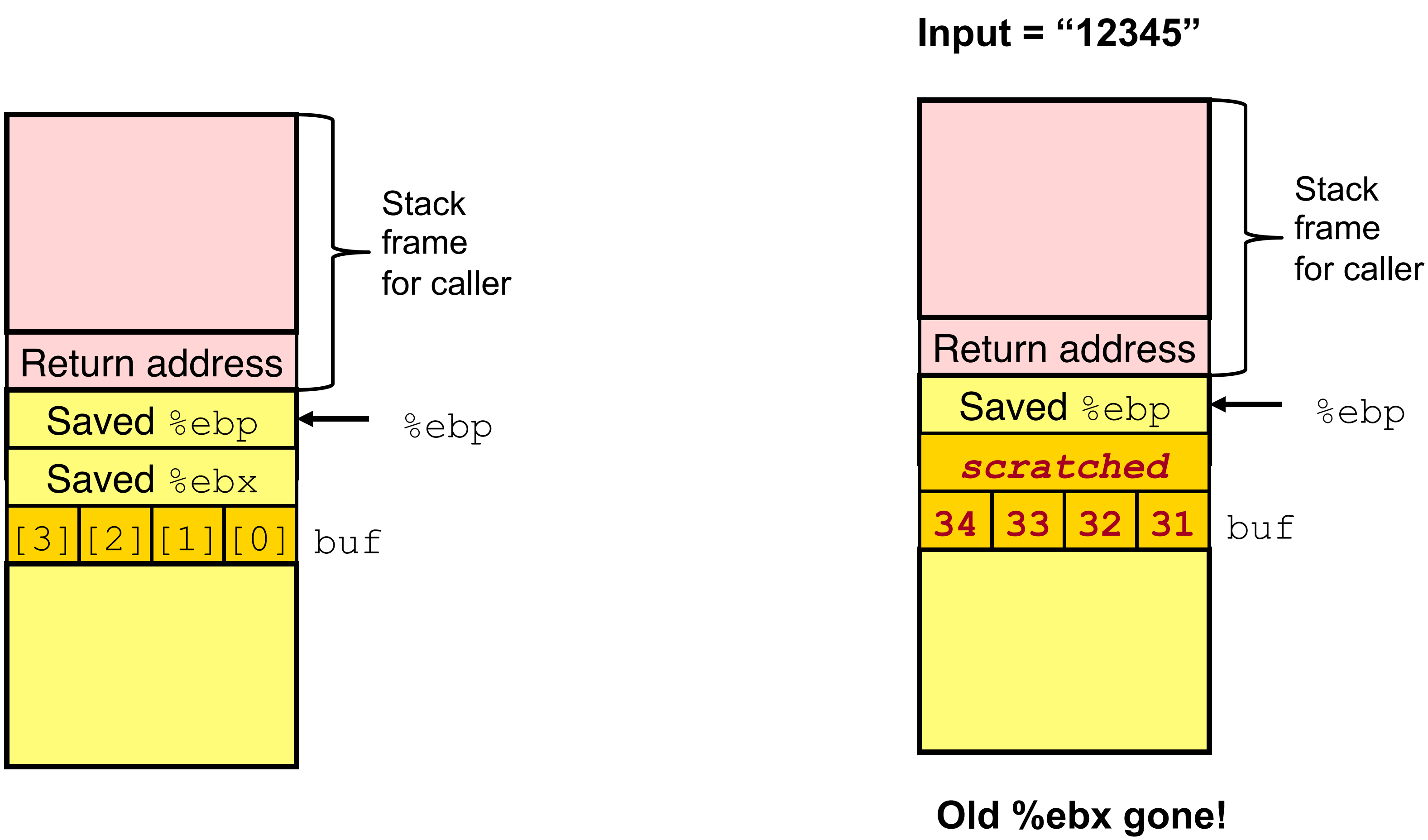
```
echo:
    pushl   %ebp                # Save %ebp on stack
    movl    %esp,%ebp           # Set new frame pointer
    pushl   $ebx                # Save %ebx
    subl    $20,%esp            # Allocate space on stack
    leal    -12(%ebp), %ebx     # Comp buf as %ebp-12
    movl    %ebx, (%esp)        # Store buf at stack top
    call    gets                # Call gets
    movl    %ebx, (%esp)        # Store buf at stack top
    call    puts                # Call puts
    addl    $20, %esp           # Deallocate 20B
    popl    %ebx                # Restore %ebx
    popl    %ebp                # Restore %ebp
    ret                         # Return
```

Buffer overflow stack example



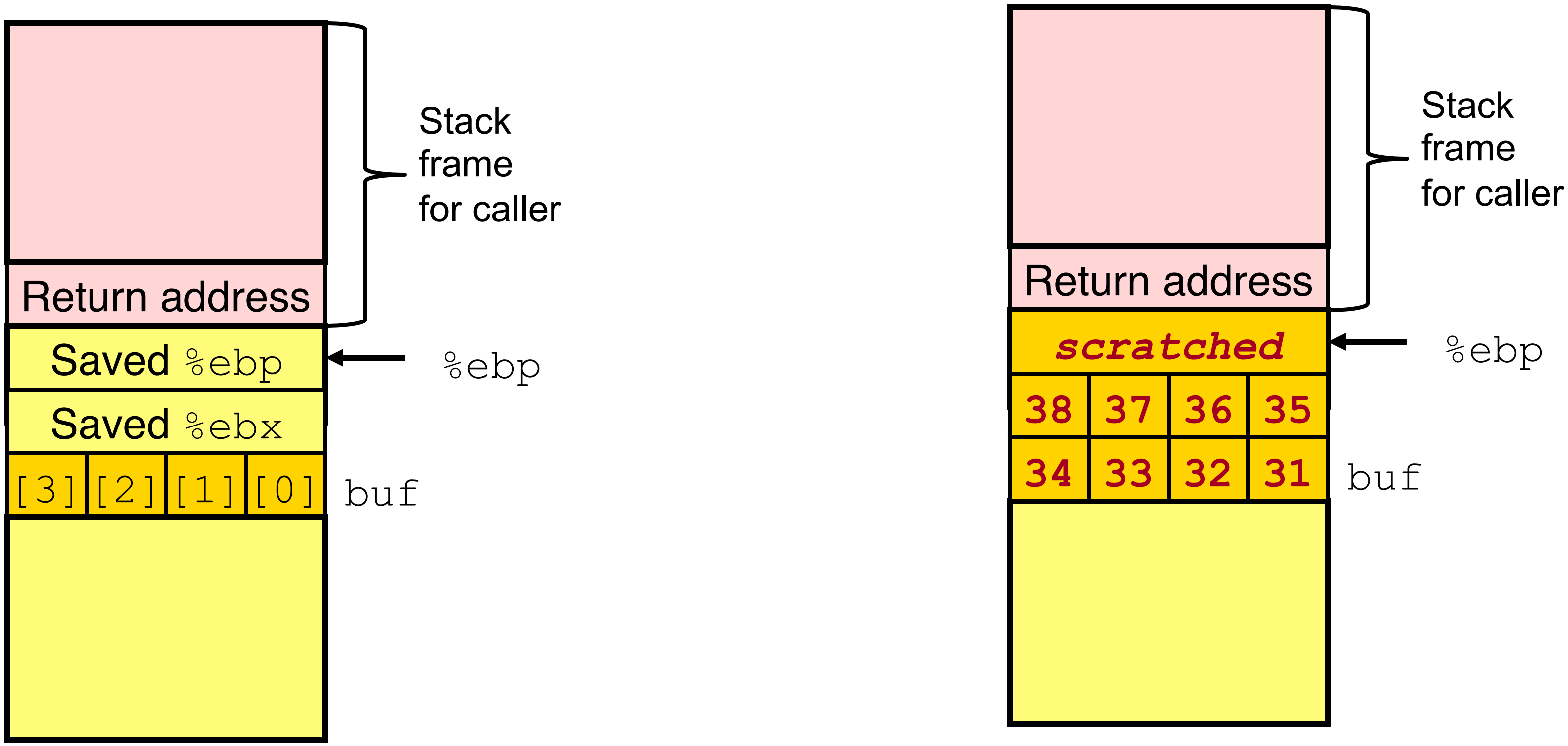
No Problem

Buffer overflow stack example



Buffer overflow stack example

Input = "123456789"

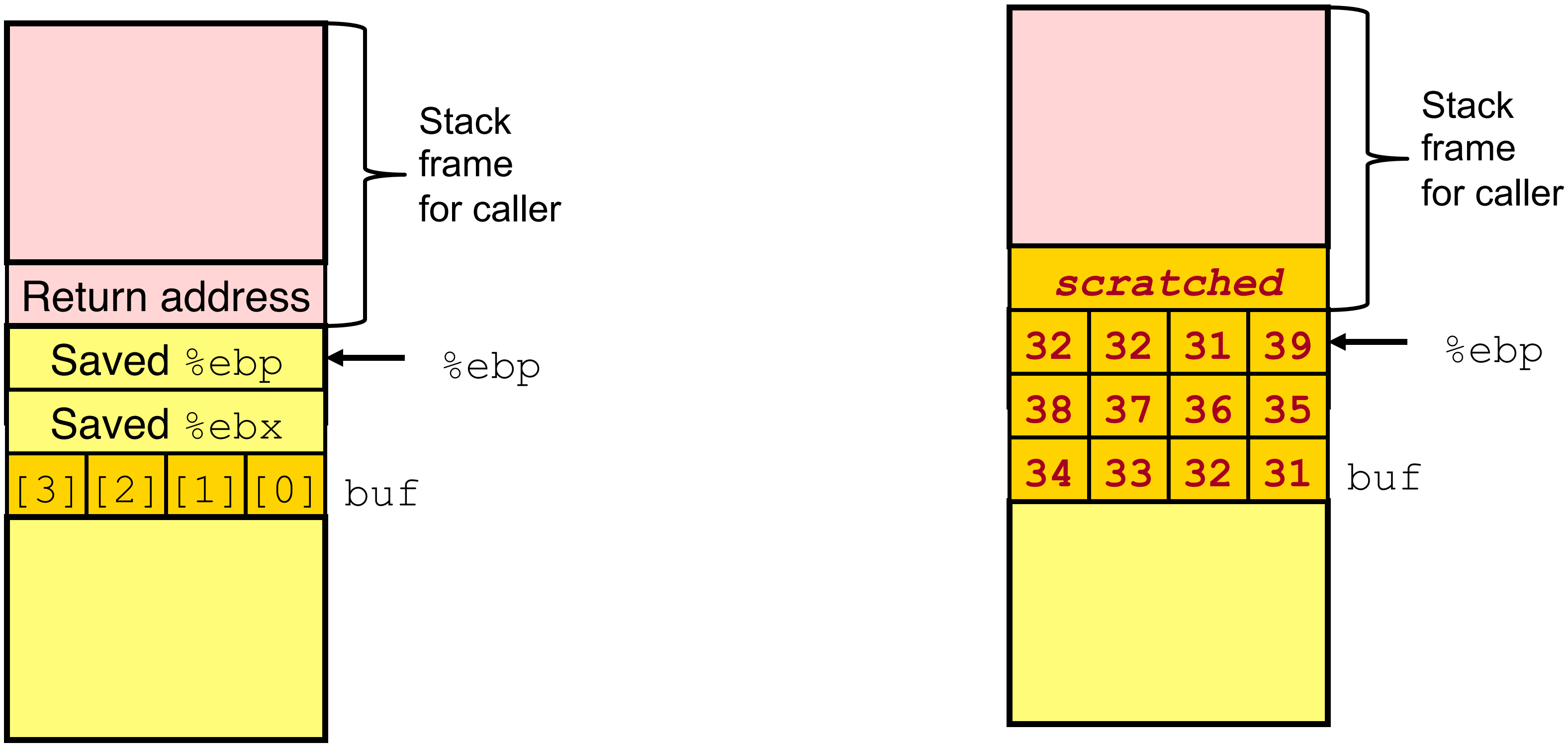


Old %ebx and %ebp gone!

Caller cannot reference its local variables and parameters

Buffer overflow stack example

Input = “1234567891234”



Old %ebx, %ebp and return address gone!

Return to where?

Buffer overflow stack

```
$ gcc -O1 -D_FORTIFY_SOURCE=0 -fno-stack-protector -o bufdemo
echo.c bufdemo.c
```

```
$ gdb bufdemo)
...done.
(gdb) break *0x804844e
Breakpoint 1 at 0x804844e
(gdb) break *0x8048456
Breakpoint 2 at 0x8048456
```

```
% objdump -d bufdemo
```

```
08048444 <echo>:
 8048444:    55                push   %ebp
 8048445:    89 e5             mov    %esp,%ebp
 8048447:    53                push   %ebx
 8048448:    83 ec 24          sub    $0x24,%esp
 804844b:    8d 5d f4          lea    -0xc(%ebp),%ebx
 804844e:    89 1c 24          mov    %ebx,(%esp)
 8048451:    e8 fa fe ff ff    call   8048350 <gets@plt>
 8048456:    89 1c 24          mov    %ebx,(%esp)
 8048459:    e8 22 ff ff ff    call   8048380 <puts@plt>
 804845e:    83 c4 24          add    $0x24,%esp
 8048461:    5b                pop    %ebx
 8048462:    5d                pop    %ebp
 8048463:    c3                ret
```

Buffer overflow stack

```
(gdb) run
Starting program: /home/fabianb/eecs213/bufdemo

Breakpoint 1, 0x0804844e in echo ()
(gdb) print /x *(unsigned *)$ebp
$1 = 0xbffff428
(gdb) n
Single stepping until exit from function echo,
which has no line number information.
Type a string: 123

Breakpoint 2, 0x08048456 in echo ()
(gdb) print /x *(unsigned *)$ebp
$2 = 0xbffff428
```

Buffer overflow stack

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/fabianb/eecs213/bufdemo

Breakpoint 1, 0x0804844e in echo ()
(gdb) print /x *(unsigned *)$ebp
$3 = 0xbffff428
(gdb) n
Single stepping until exit from function echo,
which has no line number information.
Type a string: 1234567890123

Breakpoint 2, 0x08048456 in echo ()
(gdb) print /x *(unsigned *)$ebp
$4 = 0xbfff0033
```

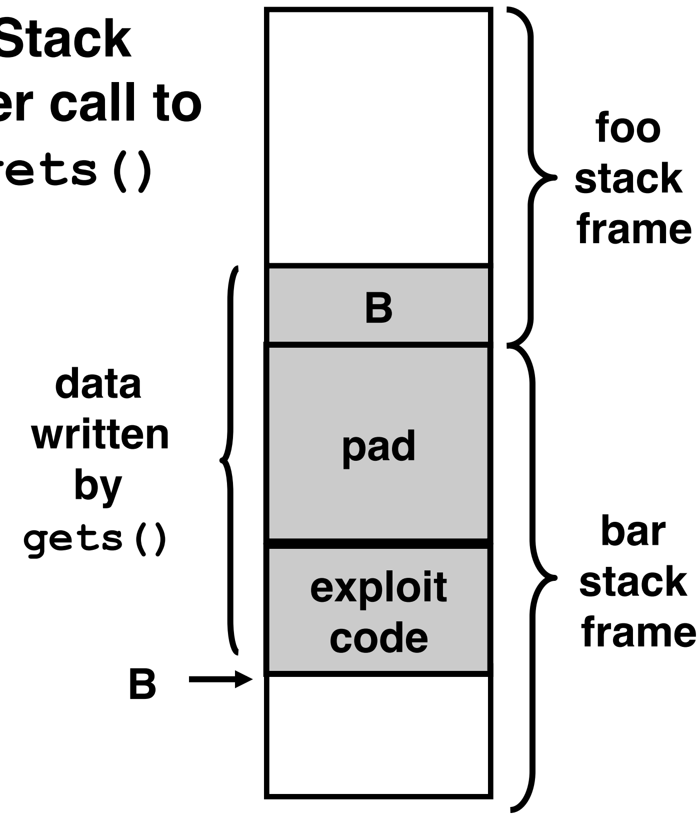
Malicious use of buffer overflow

return address A

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

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

Stack after call to `gets()`



- 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 fabianb@cc.gatech.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

AUGUST 30, 1999 VOLUME 21, ISSUE 35 Founded in 1978

ENTERPRISE NETWORKING

Phony e-mail heats up Microsoft, AOL battle

IM war gets interesting when security expert receives message from so-called Microsoft employee

By Nancy Weil

RICHARD SMITH quickly realized that he had been subjected to e-mail fakery when he discovered that the "consultant" writing him to accuse America Online (AOL) of breaching its subscriber's instant-messaging security was apparently someone from Microsoft.

AOL and Microsoft have been duking it out over instant messaging in a public dispute for several weeks, beginning with Microsoft's announcement that it was releasing a version of its Messenger software that interoperates with AOL Instant Messenger (AIM). AOL didn't then, and doesn't now, like that idea, because it requires Microsoft's software to access AOL servers and subscriber passwords. AOL has been creating barricades to keep Microsoft out, but Microsoft continues to find ways around the barriers.

Smith, who is president of Phar Lap Software, in Cambridge, Mass., has been a vocal critic of Microsoft's security and privacy practices.

Earlier this year Smith uncovered a security breach when he found that the Windows 98 operating system was allegedly gathering user registration information and sending it to the Microsoft headquarters in Redmond, Wash.

"It's kind of strange," Smith said regarding the e-mail incident.

According to Smith the e-mail claimed that AOL is sending "code snippets" that execute programs on AOL subscribers' PCs to keep out other instant messaging software. The code snippets are continually rewritten so that it doesn't matter how many times Microsoft — or anyone else — tries a backdoor method to access AIM, the AOL code won't allow access.

"AOL considers this an extremely valuable little trick, but this is something virus writers do, not reputable software companies," Smith said.

What particularly troubles Smith is that AOL allegedly is using buffer overflows to execute the programs on subscriber PCs and it's "tricky to get those right."

AOL IS DEFENDING ITS *instant-messaging territory, until now a unique part of its service provider business, from encroachment by Microsoft.*

A buffer overflow is ordinarily caused by a programming error and will generally cause a software program to crash. When caused intentionally, it is a way to violate security and force a software program to execute foreign code. Use of buffer overflows is common practice among hackers, crackers, and virus writers.

If AOL is using buffer overflows to keep instant messaging software other than AIM at bay it "could set up a situation where a few hundred thousand computers crash simultaneously," Smith said.

AOL denies that it is doing anything that might breach the securi-

mail," said Tricia Primrose, an AOL spokeswoman. "History has shown us lessons about Microsoft. This is par for the course."

Microsoft is continuing to investigate the incident but the software giant has not yet identified the e-mail sender.

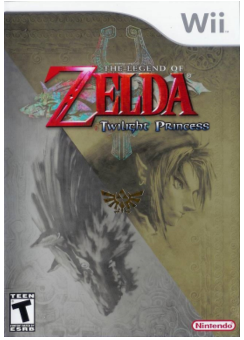
"It's very possible that it was someone from Microsoft. If it was, that's behavior that we don't condone," said Tom Pilla, a Microsoft spokesman.

While the identity of the e-mail author has not been determined, that person's conduct "should not excuse the behavior of AOL," Pilla said. "The real issue here is that AOL is putting 40 million users at risk," he said and added that, while AOL claims it wants AIM to interoperate with other instant messag-

Later determined to be from MS

Other exploits based on buffer overflows

- Code red worm 2001
  – Exploiting vulnerability in Microsoft's Internet Information Service 2001
  – On July 19, 2001 – 359,000 infected hosts
- SQL Slammer worm 2003
  – Same with Microsoft's SQL Server 200
- Hacks to run unofficial software in Xbox, PS2 and Wii without needing hardware modification 2003
  – Twilight hack – exploiting buffer overflow (in *The legend of Zelda: Twilight Princess*) in the Wii



System-level protection

- Stack randomization
  - At start of program, allocate random amount of stack space
  - Makes it difficult to predict beginning of inserted code

```
#include <stdio.h>

int main()
{
  int local;
  printf("local at %p\n", &local);
  return 0;
}
```

```
fabianb@eleuthera:~$ ./stackAddress
local at 0x7ffff296f6cc
fabianb@eleuthera:~$ ./stackAddress
local at 0x7fff764124fc
fabianb@eleuthera:~$ ./stackAddress
local at 0x7fffe48e4afc
fabianb@eleuthera:~$ ./stackAddress
local at 0x7fff4893664c
```

  - Brute force solution – “nop sled” – keep adding nop before the exploit code

System-level protection

- Stack corruption detection
  - Detect when there has been an out-of-bound write
  - Store a canary value (randomly generated) in stack frame between any local buffer and rest of the stack
  - To run overflow example, compile with **`-fno-stack-protector`**

```
echo:
            pushl   %ebp
            movl    %esp, %ebp
            pushl   %ebx
            subl    $36, %esp
            movl    %gs:20, %eax
            movl    %eax, -12(%ebp)
            xorl    %eax, %eax
            leal    -20(%ebp), %ebx
            movl    %ebx, (%esp)
            call    gets
            movl    %ebx, (%esp)
            call    puts
            movl    -12(%ebp), %eax
            xorl    %gs:20, %eax
            je      .L9
            call    __stack_chk_fail
```

Read value from a special, read-only segment in memory

Store it on the stack at offset -12 from `%ebp`

Check the canary is fine using xorl (0) if the two values are identical

System-level protection

- `gcc -O1 -S -D_FORTIFY_SOURCE=0 echo.c`
- `gcc -O1 -S -D_FORTIFY_SOURCE=1 echo.c`

```
echo:
    pushl    %ebp
    movl     %esp, %ebp
    pushl    %ebx
    subl     $36, %esp
    leal     -12(%ebp), %ebx
    movl     %ebx, (%esp)
    call     gets
    movl     %ebx, (%esp)
    call     puts
    addl     $36, %esp
    popl     %ebx
    popl     %ebp
    ret
```

```
echo:
    pushl    %ebp
    movl     %esp, %ebp
    pushl    %ebx
    subl     $36, %esp
    movl     $4, 4(%esp)
    leal     -12(%ebp), %ebx
    movl     %ebx, (%esp)
    call     __gets_chk
    movl     %ebx, (%esp)
    call     puts
    addl     $36, %esp
    popl     %ebx
    popl     %ebp
    ret
```

System-level protection

- Part of Stack Smashing Protector (SSP)
  - A debugging/security extension for GCC

http://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html

```
#undef gets

char *
__gets_chk(char * __restrict buf, size_t slen)
{
  char *abuf;
  size_t len;

  if (slen >= (size_t)INT_MAX) return gets(buf);
  if ((abuf = malloc(slen + 1)) == NULL)
    return gets(buf);
  if (fgets(abuf, (int)(slen + 1), stdin) == NULL)
    return NULL;

  len = strlen(abuf);
  if (len > 0 && abuf[len - 1] == '\n')
    --len;
  if (len >= slen) __chk_fail();

  (void)memcpy(buf, abuf, len);
  buf[len] = '\0';
  free(abuf);
  return buf;
}
```

In the current implementation mem{cpy,pcpy,move,set}, st{r,p,nc}py, str{,n}cat, {,v}s{,n}printf and gets functions are checked this way

System-level protection

- Limiting executable code regions
  - Virtual memory is divided into pages
  - Each page can be assigned a read/write/execute control
  - x86 merged read and execute into a single 1-bit flag
  - Since stack has to be readable → executable
  - Now, AMD and Intel after, add executable space protection
    - A NX (for “No eXecute”) bit in the page table

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);
}
```

x86-64: Extending IA32 to 64 bits

- New hardware capacities but same instruction set!
  - 32-bit word size is limiting – only 4GB virtual address space
    - A serious problem for applications working on large data-sets e.g. data-mining, scientific computing
- Need larger word size – next logical: 64b
  - DEC Alpha 1992
  - Sun Microsystems 1995
- The price of backward compatibility
  - Intel & Hewlett-Packard 2001
    - IA64 – a totally new instruction set
  - AMD 2003
    - x86-64 – evolution of Intel IA32 instruction set to 64b; fully backward compatibility
    - AMD took over and forced Intel to backtrack
    - Intel now offers Pentium 4 Xeon

x86-64 overview

- Pointers and long integers are 64b; integer operations support 8 (b), 16 (w), 32 (l), 64 (q) bits data types
- Set of general purpose regs expanded to 16 (from 8)
- Much of program state is held in these registers, including up to 6 integer and pointer procedure arguments
- Conditional operations implemented as conditional moves
- Floating point operations implemented using register-oriented instructions rather than stack-based ones

Data types

- Note pointers (now potentially given access to 2^{64} bytes) and long integers

C dec	Intel	Suffix	X86-64 size	IA32 size
`char`	Byte	`b`	1	1
`short`	Word	`w`	2	2
`int`	Double word	`l`	4	4
`long int`	Quad word	`q`	8	4
`long long int`	Quad word	`q`	8	8
`char *`	Quad word	`q`	8	4
`float`	Single prec	`s`	4	4
`double`	Double prec	`d`	8	8
`long double`	Extended prec	`t`	10/16	10/12

- Note size of pointers and effect of “long”

A simple example

- Some assembly code differences

```
long int simple_l (long int*xp, long int y)
{
  long int t = *xp + y;
  *xp = t;
  return t;
}
```

% gcc -O1 -S -m32 simple.c

```
1. simple_l:
2.    pushl    %ebp
3.    movl     %esp, %ebp
4.    movl     8(%ebp), %edx
5.    movl     12(%ebp), %eax
6.    addl     (%edx), %eax
7.    movl     %eax, (%edx)
8.    pop      %ebp
9.    ret
```

% gcc -O1 -S -m64 simple.c

```
1. simple_l:
2.    movq     %rsi, %rax
3.    addq     (%rdi), %rax
4.    movq     %rax, (%rdi)
5.    ret
```

`movq` instead of `movl`

No stack frame, arguments passed in registers

Return value in %rax

Accessing information

- Summary of changes to registers
  - Double number of registers to 16
  - All registers are 64b long
    - Extended `%rax, %rcx, %rdx, %rbx, %rsi, %rdi, %rsp, %rbp`
    - New `%r8-%r15`
  - Low-order 32, 16 and 8 bits of each register can be accessed directly (e.g. `%eax, %ax, %al`)
  - For backward compatibility, the second byte of `%rax, %rcx, %rdx, and %rbx` can be accessed directly (e.g. `%ah`)
- Same addressing forms plus a PC-relative (pc is in `%rip`) operand addressing mode

```
add 0x200ad1(%rip), %rax
```

Arithmetic instructions and control

- To each arithmetic instruction class seen, add instructions that operate on quad words with suffix `q`

```
addq %rdi, %rsi
```

- GCC must carefully chose operations when mixing operands of different sizes
- For control, add `cmpq` and `testq` to compare and test quad words

Procedures in x86-64

- Some highlights
  - Up to the first 6 arguments are passed via registers
  - `callq` stores a 64-bit return address in the stack
  - Many functions don't even need a stack frame
  - Functions can access storage on the stack up to 128 bytes beyond current stack pointer value; this is so you can store information there without altering the stack pointer
  - No frame pointer; references are made relative to stack pointer
  - There are also a few (6) callee-save registers and only two caller-save (`%r10` and `%r11`, you can also use argument passing registers when there are <6 arguments)

Argument passing

- Up to 6 integral arguments can be passed via regs
- The rest using the stack

```
void proc(long a1, long *a1p,
     int a2, int *a2p,
     short a3, short *a3p,
     char a4, char *a4p)
{
     *a1p += a1;
     *a2p += a2;
     *a3p += a3;
     *a4p += a4;
}
```

Registers are used in an specific order

Oper. size/ Argument #	1	2	3	4	5	6
64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
32	%edi	%esi	%edx	%ecx	%r8d	%r9d
16	%di	%si	%dx	%cx	%r8w	%r9w
8	%dil	%sil	%dl	%cl	%r8b	%r9b

```
Proc:
     movq    16(%rsp), %r10    # Fetch a4p (64b)
     addq    %rdi, (%rsi)      # *a1p += a1 (64b)
     addl    %edx, (%rcx)      # *a2p += a2 (32b)
     addw    %r8w, (%r9)       # *a3p += a3 (16b)
     movzbl  8(%rsp), %eax     # Fetch a4 (8b)
     addb    %al, (%r10)       # *a4p += a4 (8b)
     ret
```

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
- Thanks to AMD, x86 has caught up with RISC from early 1980s!
- Moving from 32b to 64b, more memory needed for pointers; of course
- Nevertheless, 64b operating systems and applications will become commonplace