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 reexec/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
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

```c
/* Get string from stdin */
char *gets(char *s)
{
    int c;
    char *dest = s;
    int gotchar = 0;

    while ((c = getchar()) != '
' && c != EOF) {
        *dest++ = c;
        gotchar = 1;
    }

    *dest++ = '\0';
    if (*c == EOF && !gotchar)
        return NULL;
    return s;
}
```

- Similar problems with other Unix functions
  - `strcpy`: Copies string of arbitrary length
  - `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification
Vulnerable buffer code

```c
/* 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

```bash
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);
}

Stack frame for caller

Return address
Saved %ebp
Saved %ebx

[3] [2] [1] [0]
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

Input = “123”

No Problem
Buffer overflow stack example

Stack frame for caller

Return address

Saved %ebp

Saved %ebx

[3][2][1][0]

buf

Input = “12345”

Stack frame for caller

Return address

Saved %ebp

Scratched

34 33 32 31

buf

Old %ebx gone!
Buffer overflow stack example

Input = “123456789”

Old %ebp and %ebx gone!

Caller cannot reference its local variables and parameters
Buffer overflow stack example

Input = “1234567891234”

Stack frame for caller

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

<table>
<thead>
<tr>
<th>Address</th>
<th>OPCODES</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>08048444:</td>
<td></td>
<td>&lt;echo&gt;</td>
</tr>
<tr>
<td>8048444:</td>
<td>55</td>
<td>push %ebp</td>
</tr>
<tr>
<td>8048445:</td>
<td>89 e5</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>8048447:</td>
<td>53</td>
<td>push %ebx</td>
</tr>
<tr>
<td>8048448:</td>
<td>83 ec 24</td>
<td>sub $0x24,%esp</td>
</tr>
<tr>
<td>804844b:</td>
<td>8d 5d f4</td>
<td>lea -0xc(%ebp),%ebx</td>
</tr>
<tr>
<td>804844e:</td>
<td>89 1c 24</td>
<td>mov %ebx,(%esp)</td>
</tr>
<tr>
<td>8048451:</td>
<td>e8 fa fe ff ff</td>
<td>call 8048350 <a href="mailto:gets@plt">gets@plt</a></td>
</tr>
<tr>
<td>8048456:</td>
<td>89 1c 24</td>
<td>mov %ebx,(%esp)</td>
</tr>
<tr>
<td>8048459:</td>
<td>e8 22 ff ff ff</td>
<td>call 8048380 <a href="mailto:puts@plt">puts@plt</a></td>
</tr>
<tr>
<td>804845e:</td>
<td>83 c4 24</td>
<td>add $0x24,%esp</td>
</tr>
<tr>
<td>8048461:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048462:</td>
<td>5d</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>8048463:</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>
Buffer overflow stack

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

Breakpoint 1, 0x0804844e in echo ()
(gdb) print /x *(unsigned *)&ebp
$1 = 0xbfffff428
(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 = 0xbfffff428
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 = 0xbfffff428
(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 = 0xbfffff0033
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 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.
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
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 2000

- **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;
}
```

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

```
fabianb@eleuthera:$ ~/stackAddress
local at 0x7fffff296f6cc
fabianb@eleuthera:$ ~/stackAddress
local at 0x7ffff764124fc
fabianb@eleuthera:$ ~/stackAddress
local at 0x7fffff48e4af
fabianb@eleuthera:$ ~/stackAddress
local at 0x7fffff4893664c
```
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 \texttt{-fno-stack-protector}

1. \texttt{echo:}
2. \texttt{pushl} \%ebp
3. \texttt{movl} \%esp, \%ebp
4. \texttt{pushl} \%ebx
5. \texttt{subl} $36, \%esp
6. \texttt{movl} \%gs:20, \%eax
7. \texttt{movl} \%eax, -12(\%ebp)
8. \texttt{xorl} \%eax, \%eax
9. \texttt{leal} \%eax, -20(\%ebp), \%ebx
10. \texttt{movl} \%ebx, (\%esp)
11. \texttt{call} \texttt{gets}
12. \texttt{movl} \%ebx, (\%esp)
13. \texttt{call} \texttt{puts}
14. \texttt{movl} -12(\%ebp), \%eax
15. \texttt{xorl} \%gs:20, \%eax
16. \texttt{je} \texttt{.L9}
17. \texttt{call} \texttt{__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
System-level protection

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


```c
#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

```c
/* 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

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

- Note size of pointers and effect of “long”
A simple example

Some assembly code differences

```c
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 \texttt{cmpq} and \texttt{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

```c
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 a specific order:

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

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