BUFFER OVERFLOW

Spring 2014
Euiseong Seo
(euiseong@gmail.com)
November, 1988

- Internet Worm attacks thousands of Internet hosts.
- How did it happen?
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
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 library functions do not check argument sizes.
- allows target buffers to overflow.
Implementation of Unix function \texttt{gets()}

\begin{verbatim}
/* 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;
}
\end{verbatim}

- No way to specify limit on number of characters to read

Similar problems with other library functions

- \texttt{strcpy, strcat}: Copy strings of arbitrary length
- \texttt{scanf, fscanf, sscanf}, when given \%s conversion specification
/* Echo Line */  
void echo()  
{   
    char buf[4];  /* Way too small! */  
    gets(buf);  
    puts(buf);  
}  

void call_echo() {  
    echo();  
}  

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

echo:

80485c5:  55                  push   %ebp
80485c6:  89 e5               mov    %esp,%ebp
80485c8:  53                  push   %ebx
80485c9:  83 ec 14            sub    $0x14,%esp
80485cc:  8d 5d f8            lea  0xfffffffff8(%ebp),%ebx
80485cf:  89 1c 24            mov    %ebx,(%esp)
80485d2:  e8 9e ff ff ff      call   8048575 <gets>
80485d7:  89 1c 24            mov    %ebx,(%esp)
80485da:  e8 05 fe ff ff      call   80483e4 <puts@plt>
80485df:  83 c4 14            add    $0x14,%esp
80485e2:  5b                  pop    %ebx
80485e3:  5d                  pop    %ebp
80485e4:  c3                  ret

call_echo:

80485eb:  e8 d5 ff ff ff ff  call   80485c5 <echo>
80485f0:  c9                  leave
80485f1:  c3                  ret
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}

Before call to gets

Stack Frame for main

<table>
<thead>
<tr>
<th>Return Address</th>
<th>Saved %ebp</th>
<th>Saved %ebx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[3] [2] [1] [0]</td>
<td></td>
</tr>
</tbody>
</table>

Stack Frame for echo

echo:
    pushl %ebp               # Save %ebp on stack
    movl %esp, %ebp          # Save %esp
    pushl %ebx               # Save %ebx
    subl $20, %esp           # Allocate stack space
    leal -8(%ebp),%ebx      # Compute buf as %ebp-8
    movl %ebx, (%esp)        # Push buf on stack
    call gets               # Call gets
    ...
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x80485c9
(gdb) run
Breakpoint 1, 0x80485c9 in echo ()
(gdb) print /x $ebp
$1 = 0xfffffd678
(gdb) print /x *(unsigned *)&ebp
$2 = 0xfffffd688
(gdb) print /x *((unsigned *)&ebp + 1)
$3 = 0x80485f0

Before call to gets

Stack Frame for main

Return Address
Saved %ebp
Saved %ebx
[3][2][1][0]
Stack Frame for echo

Before call to gets

Stack Frame for main

0x08 0x04 0x85 0xf0
0xff 0xff 0xd6 0x88
Saved %ebx
buf
Stack Frame for echo

0xfffffd688
0xfffffd678
buf

80485eb: e8 d5 ff ff ff call 80485c5 <echo>
80485f0: c9 leave
Buffer Overflow Example #1

Before call to `gets`

- Stack Frame for `main`
  - Saved `%ebx`
  - Buf

- Stack Frame for `echo`

Input 1234567

- Stack Frame for `main`
  - Buf

Overflow buf, and corrupt `%ebx`, but no problem
**Buffer Overflow Example #2**

*Before call to gets*

- Stack Frame for `main`
  - 08 04 85 f0
  - ff ff d6 88
  - Saved %ebx
  - xx xx xx xx

Stack Frame for `echo`

- Stack Frame for `main`
  - 08 04 85 f0
  - ff ff d6 88
  - Saved %ebx
  - 38 37 36 35
  - 34 33 32 31

Input 12345678

Base pointer corrupted

```
80485eb: e8 d5 ff ff ff    call 80485c5 <echo>
80485f0: c9                leave    # Set %ebp to corrupted value
80485f1: c3                ret
```
Buffer Overflow Example #3

Before call to gets

Stack Frame for main

0xffffd678
08 04 85 f0
ff ff d6 88
Saved %ebx
xx xx xx xx
buf

Stack Frame for echo

Input 123456789

Stack Frame for main

0xffffd688
08 04 85 00
43 42 41 39
38 37 36 35
34 33 32 31
buf

Return address corrupted

80485eb: e8 d5 ff ff ff call 80485c5 <echo>
80485f0: c9 leave # Desired return point
Input string contains byte representation of executable code

Overwrite return address A with address of buffer B

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.
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

It was later determined that this email originated from within Microsoft!
**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
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
  - Or use `%ns` where `n` is a suitable integer
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
- In traditional x86, can mark region of memory as either “read-only” or “writeable”
  - Can execute anything readable
- X86-64 added explicit “execute” permission

```plaintext
unix> gdb bufdemo
(gdb) break echo
(gdb) run
(gdb) print /x $ebp
$1 = 0xffffffffc638
(gdb) run
(gdb) print /x $ebp
$2 = 0xffffffffbb08
(gdb) run
(gdb) print /x $ebp
$3 = 0xffffffffc6a8
```
**Stack Canaries**

- **Idea**
  - Place special value ("canary") on stack just beyond buffer
  - Check for corruption before exiting function

- **GCC Implementation**
  - `-fstack-protector`
  - `-fstack-protector-all`

```
unix>./bufdemo-protected
Type a string:1234
1234

unix>./bufdemo-protected
Type a string:12345
*** stack smashing detected ***
```
## Protected Buffer Disassembly

**echo:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Mnemonic</th>
<th>804864d: 55</th>
<th>push %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>804864e: 89 e5</td>
<td>mov %esp,%ebp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8048650: 53</td>
<td>push %ebx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8048651: 83 ec 14</td>
<td>sub $0x14,%esp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8048654: 65 a1 14 00 00 00</td>
<td>mov %gs:0x14,%eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>804865a: 89 45 f8</td>
<td>mov %eax,0xfffffffff8(%ebp)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>804865d: 31 c0</td>
<td>xor %eax,%eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>804865f: 8d 5d f4</td>
<td>lea 0xfffffffff4(%ebp),%ebx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8048662: 89 1c 24</td>
<td>mov %ebx,(%esp)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8048665: e8 77 ff ff ff</td>
<td>call 80485e1 &lt;gets&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>804866a: 89 1c 24</td>
<td>mov %ebx,(%esp)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>804866d: e8 ca fd ff ff</td>
<td>call 804843c <a href="mailto:puts@plt">puts@plt</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8048672: 8b 45 f8</td>
<td>mov 0xfffffffff8(%ebp),%eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8048675: 65 33 05 14 00 00 00</td>
<td>xor %gs:0x14,%eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>804867c: 74 05</td>
<td>je 8048683 &lt;echo+0x36&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>804867e: e8 a9 fd ff ff</td>
<td>call 804842c &lt;FAIL&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8048683: 83 c4 14</td>
<td>add $0x14,%esp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8048686: 5b</td>
<td>pop %ebx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8048687: 5d</td>
<td>pop %ebp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8048688: c3</td>
<td>ret</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}

Before call to gets

Stack Frame for main

Return Address
Saved %ebp
Saved %ebx
Canary
[3][2][1][0]
Stack Frame for echo

echo:
    ..
    movl %gs:20, %eax  # Get canary
    movl %eax, -8(%ebp) # Put on stack
    xorl %eax, %eax    # Erase canary
    ..
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}

Before call to gets

Stack Frame for main

Return Address
Saved %ebp
Saved %ebx
Canary
[3][2][1][0]
Stack Frame for echo

Before call to gets

Stack Frame for main

Return Address
Saved %ebp
Saved %ebx
Canary
[3][2][1][0]
Stack Frame for echo

.movl  -8(%ebp), %eax   # Retrieve from stack
.xorl  %gs:20, %eax    # Compare with Canary
    .L24       # Same: skip ahead
    .L24:
        . . .
(gdb) break echo
(gdb) run
(gdb) steipi 3
(gdb) print /x *((unsigned *) $ebp - 2)
$1 = 0x3e37d00

Benign corruption!
(allowing programmers to make silent off-by-one errors)
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
  - Add itself to other programs
  - Cannot run independently

- Both are (usually) designed to spread among computers and to wreak havoc