The Hardware/Software Interface
CSE351 Winter 2013

Buffer Overflow
Buffer Overflow

- Buffer overflows are possible because C doesn’t check array boundaries
- Buffer overflows are *dangerous* because buffers for user input are often stored on the stack
  - Probably the most common type of security vulnerability

- Today we’ll go over:
  - Address space layout
  - Input buffers on the stack
  - Overflowing buffers and injecting code
  - Defenses against buffer overflows
IA32 Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)

- **Heap**
  - Dynamically allocated storage
  - Allocated by `malloc()`, `calloc()`, `new()`

- **Data**
  - Statically allocated data
    - Read-only: string literals
    - Read/write: global arrays and variables

- **Text**
  - Executable machine instructions
  - Read-only

Upper 2 hex digits = 8 bits of address
Memory Allocation Example

```c
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 ... */
}
```

Where does everything go?
IA32 Example Addresses

address range \( \sim 2^{32} \)

\[
\begin{align*}
\$\text{esp} & \quad 0xFFFFFBCD0 \\
\text{p3} & \quad 0x65586008 \\
\text{p1} & \quad 0x55585008 \\
\text{p4} & \quad 0x1904A110 \\
\text{p2} & \quad 0x1904A008 \\
\&\text{p2} & \quad 0x18049760 \\
\text{beyond} & \quad 0x08049744 \\
\text{big\_array} & \quad 0x18049780 \\
\text{huge\_array} & \quad 0x08049760 \\
\text{main()} & \quad 0x080483C6 \\
\text{useless()} & \quad 0x08049744 \\
\text{final\_malloc()} & \quad 0x006BE166
\end{align*}
\]

\texttt{malloc()} is dynamically linked
address determined at runtime

not drawn to scale
Internet Worm

- These characteristics of the traditional IA32 Linux memory layout provide opportunities for malicious programs
  - Stack grows “backwards” in memory
  - Data and instructions both stored in the same memory

- November, 1988
  - Internet Worm attacks thousands of Internet hosts.
  - How did it happen?

- The Internet Worm was 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()`

```c
/* 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;
}
```

- What could go wrong in this code?
String Library Code

- Implementation of Unix function `gets()`

```c
/* 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;
}
```

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

- 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;
}
```

```
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>80484f0</td>
<td>55</td>
<td><code>push %ebp</code></td>
</tr>
<tr>
<td>80484f1</td>
<td>89 e5</td>
<td><code>mov %esp,%ebp</code></td>
</tr>
<tr>
<td>80484f3</td>
<td>53</td>
<td><code>push %ebx</code></td>
</tr>
<tr>
<td>80484f4</td>
<td>8d 5d f8</td>
<td><code>lea 0xfffffffff8(%ebp),%ebx</code></td>
</tr>
<tr>
<td>80484f7</td>
<td>83 ec 14</td>
<td><code>sub $0x14,%esp</code></td>
</tr>
<tr>
<td>80484fa</td>
<td>89 1c 24</td>
<td><code>mov %ebx,(%esp)</code></td>
</tr>
<tr>
<td>80484fd</td>
<td>e8 ae ff ff ff</td>
<td><code>call 80484b0 &lt;gets&gt;</code></td>
</tr>
<tr>
<td>8048502</td>
<td>89 1c 24</td>
<td><code>mov %ebx,(%esp)</code></td>
</tr>
<tr>
<td>8048505</td>
<td>e8 8a fe ff ff</td>
<td><code>call 8048394 &lt;puts@plt&gt;</code></td>
</tr>
<tr>
<td>804850a</td>
<td>83 c4 14</td>
<td><code>add $0x14,%esp</code></td>
</tr>
<tr>
<td>804850d</td>
<td>5b</td>
<td><code>pop %ebx</code></td>
</tr>
<tr>
<td>804850e</td>
<td>c9</td>
<td><code>leave</code></td>
</tr>
<tr>
<td>804850f</td>
<td>c3</td>
<td><code>ret</code></td>
</tr>
<tr>
<td>80485f2</td>
<td>e8 f9 fe ff ff</td>
<td><code>call 80484f0 &lt;echo&gt;</code></td>
</tr>
<tr>
<td>80485f7</td>
<td>8b 5d fc</td>
<td><code>mov 0xfffffffffc(%ebp),%ebx</code></td>
</tr>
<tr>
<td>80485fa</td>
<td>c9</td>
<td><code>leave</code></td>
</tr>
<tr>
<td>80485fb</td>
<td>31 c0</td>
<td><code>xor %eax,%eax</code></td>
</tr>
<tr>
<td>80485fd</td>
<td>c3</td>
<td><code>ret</code></td>
</tr>
</tbody>
</table>
Buffer Overflow Stack

Before call to gets

Stack Frame for main

Return Address
Saved %ebp
Saved %ebx
[3][2][1][0]
buf

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

echo:
    pushl %ebp  # Save %ebp on stack
    movl %esp, %ebp
    pushl %ebx  # Save %ebx
    leal -8(%ebp),%ebx  # Compute buf as %ebp-8
    subl $20, %esp  # Allocate stack space
    movl %ebx, (%esp)  # Push buf addr on stack
    call gets  # Call gets
    . . .
Buffer Overflow Stack Example

Before call to gets

Stack Frame for `main`

Return Address
Saved `%ebp`
Saved `%ebx`

buf

Before call to gets

Stack Frame for `main`

0xffffffffc658

f7 85 04 08
58 c6 ff ff

Saved `%ebx`

buf

0xffffffffc638

0xffffffffc630

80485f2: call 80484f0 <echo>
80485f7: mov 0xffffffffc(%ebp),%ebx # Return Point
Buffer Overflow Example #1

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

Before call to `gets`

Stack Frame for `main`

0xfffffc658

f7 85 04 08
58 c6 ff ff
Saved %ebx
xx xx xx xx

0xfffffc638

buf

0xfffffc630

Input 12345678

Stack Frame for `main`

0xfffffc658

f7 85 04 08
58 c6 ff 00
38 37 36 35
34 33 32 31
buf

Frame pointer corrupted

... 804850a:  83 c4 14  add $0x14,%esp  # deallocate space
804850d:  5b  pop %ebx  # restore %ebx
804850e:  c9  leave  # movl %ebp, %esp; popl %ebp
804850f:  c3  ret  # Return
Buffer Overflow Example #3

Before call to `gets`

Stack Frame for `main`

0xfffffc658

f7 85 04 08
58 c6 ff ff

Saved %ebx

xx xx xx xx

buf

0xfffffc630

Input 123456789ABC

Stack Frame for `main`

0xfffffc658

f7 85 04 00
43 42 41 39

38 37 36 35

34 33 32 31

buf

0xfffffc630

Return address corrupted

080485f2: call 80484f0 <echo>
080485f7: mov 0xffffffffc(%ebp),%ebx # Return Point
Malicious Use of Buffer Overflow

- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer (need to know B)
- When `bar()` executes `ret`, will jump to exploit code (instead of A)
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
Avoiding Overflow Vulnerability

Use library routines that limit string lengths

- `fgets` instead of `gets` (second argument to `fgets` sets limit)
- `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

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

- **Randomized stack offsets**
  - At start of program, allocate random amount of space on stack
  - Makes it difficult for exploit to predict beginning of inserted code

- **Use techniques to detect stack corruption**

- **Nonexecutable code segments**
  - Only allow code to execute from “text” sections of memory
  - Do NOT execute code in stack, data, or heap regions
  - Hardware support needed