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

Where does everything go?

### 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 ... */
}
```
IA32 Example Addresses

address range ~2^32

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$esp</td>
<td>0xffffbcd0</td>
</tr>
<tr>
<td>p3</td>
<td>0x65586008</td>
</tr>
<tr>
<td>p1</td>
<td>0x55585008</td>
</tr>
<tr>
<td>p4</td>
<td>0x1904a110</td>
</tr>
<tr>
<td>p2</td>
<td>0x1904a008</td>
</tr>
<tr>
<td>&amp;p2</td>
<td>0x18049760</td>
</tr>
<tr>
<td>beyond</td>
<td>0x08049744</td>
</tr>
<tr>
<td>big_array</td>
<td>0x18049780</td>
</tr>
<tr>
<td>huge_array</td>
<td>0x08049760</td>
</tr>
<tr>
<td>main()</td>
<td>0x080483c6</td>
</tr>
<tr>
<td>useless()</td>
<td>0x08049744</td>
</tr>
<tr>
<td>malloc()</td>
<td>0x006be166</td>
</tr>
</tbody>
</table>

malloc() is dynamically linked.
Address determined at runtime.

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?

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;
}
```

- 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 a conversion specification
Vulnerable Buffer Code

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

```c
int main()
{
    printf("Type a string: ");
    echo();
    return 0;
}
```

/* Echo Line */

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

unix> ./bufdemo
Type a string: 1234567
1234567
unix>

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

unix> ./bufdemo
Type a string: 123456789ABC
Segmentation Fault
9

Buffer Overflow Stack

Before call to gets

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

```
int main()
{
    printf("Type a string: ");
    echo();
    return 0;
}
```

Before call to gets

```
char buf[4]; /* Way too small! */
gets(buf);
puts(buf);
```

```c
//echo:
pushl %ebp
# Save %ebp on stack
movl %esp, %ebp
pushl %ebx
# Save %ebx
lea -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

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

```
int main()
{
    printf("Type a string: ");
    echo();
    return 0;
}
```

```
char buf[4]; /* Way too small! */
gets(buf);
puts(buf);
```

```
//echo:
pushl %ebp
# Save %ebp on stack
movl %esp, %ebp
pushl %ebx
# Save %ebx
lea -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 Example #1

Before call to gets

Input 1234567

Stack Frame for main
0xffffc658

Buf
0xffffc6638

Saved %ebx
xx xx xx xx

0xffffc630

Overflow buf, and corrupt saved %ebx, but no problem

Buffer Overflow Example #2

Before call to gets

Input 1234567

Stack Frame for main
0xffffc658

Buf
0xffffc6638

Saved %ebx
xx xx xx xx

0xffffc630

Frame pointer corrupted

804850a: 83 c4 14 add $0x14,%esp # deallocate space
804850b: 5b pop %ebx # restore %ebx
804850c: c9 leave # movl %ebp, %esp; popl %ebp
804850d: c3 ret # Return

Buffer Overflow Example #3

Before call to gets

Input 123456789ABC

Stack Frame for main
0xffffc658

Buf
0xffffc6638

Saved %ebx
xx xx xx xx

0xffffc630

Return address corrupted

080485f2: call 8048ef0 <echo>
080485f7: mov 0xfffffc(%ebp),%ebx # Return Point

Malicious Use of Buffer Overflow

Stack after call to gets()

foo stack frame

pad

exploit code

Return address A

int bar() {
    char buf[64];
    gets(buf);
    ...
    return ...;
}

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

- 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

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

- **Use library routines that limit string lengths**
  - Use `fgets` instead of `gets` (second argument to `fgets` sets limit)
  - Use `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

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