Buffer overflows

- 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

Spring 2014
IA32/Linux Stack Frame

- **Current Stack Frame ("Top" to Bottom)**
  - “Argument build” area (parameters for function about to be called)
  - Local variables (if can’t be kept in registers)
  - Saved register context (when reusing registers)
  - Old frame pointer (for caller)

- **Caller’s Stack Frame**
  - Return address
    - How does call/ret change the stack?
  - Arguments for this call
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 ~2^{32}*

<table>
<thead>
<tr>
<th>Variable</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>$esp</td>
<td>0xfffffbc0d0</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>final malloc()</td>
<td>0x006be166</td>
</tr>
</tbody>
</table>

`malloc()` is dynamically linked; its address is determined at runtime.
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?

- Stack buffer overflow exploits!
Buffer Overflow in a nutshell

- Many classic Unix/Linux/C functions do not check argument sizes
- C does not check array bounds
- Allows overflowing (writing past the end of) buffers (arrays)
- Overflows of buffers on the stack overwrite interesting data
- Attackers just choose the right inputs
- Probably the most common type of security vulnerability
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

**080484f0  <echo>:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>80484f0:</td>
<td>push %ebp</td>
<td></td>
</tr>
<tr>
<td>80484f1:</td>
<td>mov %esp,%ebp</td>
<td></td>
</tr>
<tr>
<td>80484f3:</td>
<td>push %ebx</td>
<td></td>
</tr>
<tr>
<td>80484f4:</td>
<td>lea 0xfffffffff8(%ebp),%ebx</td>
<td></td>
</tr>
<tr>
<td>80484f7:</td>
<td>sub $0x14,%esp</td>
<td></td>
</tr>
<tr>
<td>80484fa:</td>
<td>mov %ebx,(%esp)</td>
<td></td>
</tr>
<tr>
<td>80484fd:</td>
<td>call 80484b0 &lt;gets&gt;</td>
<td></td>
</tr>
<tr>
<td>8048502:</td>
<td>mov %ebx,(%esp)</td>
<td></td>
</tr>
<tr>
<td>8048505:</td>
<td>call 8048394 <a href="mailto:puts@plt">puts@plt</a></td>
<td></td>
</tr>
<tr>
<td>804850a:</td>
<td>add $0x14,%esp</td>
<td></td>
</tr>
<tr>
<td>804850d:</td>
<td>pop %ebx</td>
<td></td>
</tr>
<tr>
<td>804850e:</td>
<td>leave</td>
<td></td>
</tr>
<tr>
<td>804850f:</td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>80485f2:</td>
<td>call 80484f0 &lt;echo&gt;</td>
<td></td>
</tr>
<tr>
<td>80485f7:</td>
<td>mov 0xfffffffff8(%ebp),%ebx</td>
<td></td>
</tr>
<tr>
<td>80485fa:</td>
<td>leave</td>
<td></td>
</tr>
<tr>
<td>80485fb:</td>
<td>xor %eax,%eax</td>
<td></td>
</tr>
<tr>
<td>80485fd:</td>
<td>ret</td>
<td></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

%ebp

/* 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

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, why?
What happens if input has one more byte?
Buffer Overflow Example #2

**Before call to gets**

Stack Frame for `main`

- `f7 85 04 08`
- `58 c6 ff ff`
- Saved `%ebx`
- `buf`
- `0xffffffffc630`

**Input “12345678”**

Stack Frame for `main`

- `f7 85 04 08`
- `58 c6 ff 00`
- `buf`
- `0xffffffffc630`

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
```

Saved `%ebx`

```
0xfffffc638
```

buf

```
0xfffffc630
```

Input “123456789ABC”

Stack Frame for `main`

```
0xfffffc658
```

```
0xfffffc638
```

buf

```
0xfffffc630
```

Return address corrupted

```
080485f2: call 80484f0 <echo>
080485f7: mov 0xffffffffc(%ebp),%ebx # Return Point
```

Hmmm, what can you do with it?
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

commandline facebook of the 80s!
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

Other ideas?
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