Buffer Overflows
CSE 351 Autumn 2017

Instructor: Justin Hsia
Teaching Assistants: Lucas Wotton Michael Zhang Parker DeWilde Ryan Wong Sam Gehman Sam Wolfson Savanna Yee Vinny Palaniappan

Administrivia
- Mid-quarter survey due tomorrow (11/2)
- Homework 3 due Friday (11/3)
- Lab 3 released today, due next Thursday (11/9)
- Midterm grades (out of 50) to be released by Saturday
  - Solutions posted on website
  - Rubric and grades will be found on Gradescope
  - Regrade requests will be open for a short time after grade release

Buffer Overflows
- Address space layout (more details!)
- Input buffers on the stack
- Overflowing buffers and injecting code
- Defenses against buffer overflows

Review: General Memory Layout
- Stack
  - Local variables (procedure context)
- Heap
  - Dynamically allocated as needed
    - malloc(), calloc(), new, ...
- Statically allocated Data
  - Read/write: global variables (Static Data)
  - Read-only: string literals (Literals)
- Code/Instructions
  - Executable machine instructions
  - Read-only

x86-64 Linux Memory Layout
- Stack
  - Runtime stack has 8 MiB limit
- Heap
  - Dynamically allocated as needed
    - malloc(), calloc(), new, ...
- Statically allocated data (Data)
  - Read-only: string literals
  - Read/write: global arrays and variables
- Code / Shared Libraries
  - Executable machine instructions
  - Read-only

Memory Allocation Example
```c
char big_array[1L<<24]; /* 16 MB */
char huge_array[1L<<31]; /* 2 GB */
int global = 0;
int useless() { return 0; }
int main()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8); /* 256 B */
    p3 = malloc(1L << 32); /* 4 GB */
    p4 = malloc(1L << 8); /* 256 B */
    /* Some print statements ... */
}
```

Where does everything go?
Memory Allocation Example

```
char big_array[1L<<24]; /* 16 MB */
char huge_array[1L<<31]; /* 2 GB */
int global = 0;
int useless() { return 0; }
int main()
{
  void *p1, *p2, *p3, *p4;
  int local = 0;
  p1 = malloc(1L << 28); /* 256 MB */
  p2 = malloc(1L << 8); /* 256 B */
  p3 = malloc(1L << 32); /* 4 GB */
  p4 = malloc(1L << 8); /* 256 B */
  /* Some print statements ... */
}
```

Buffer Overflow in a Nutshell

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

- C does not check array bounds
  - Many Unix/Linux/C functions don’t check argument sizes
  - Allows overflowing (writing past the end) of buffers (arrays)

String Library Code

Implementation of Unix function `gets()`

```
/* Get string from stdin */
int c = getchar();
char *p = dest;
while (c != EOF && c != '
') {
  *p++ = c;
  c = getchar();
}  
*p = '\0';
return dest;
```

String Library Code

```
/* Get string from stdin */
int c = getchar();
char *p = dest;
while (c != EOF && c != 'n') {
  *p++ = c;
  c = getchar();
}  
*p = '\0';
return dest;
```

Buffer Overflow in a Nutshell

- Buffer overflows on the stack can overwrite “interesting” data
- Attackers just choose the right inputs

- Simplest form (sometimes called “stack smashing”)
  - Unchecked length on string input into bounded array causes overwriting of stack data
  - Try to change the return address of the current procedure

- Why is this a big deal?
  - It is (was?) the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance

String Library Code

- No way to specify limit on number of characters to read
- Similar problems with other Unix functions:
  - `strcpy`: Copies string of arbitrary length to a dst
  - `scanf`, `fscanf`, `sscanf`, when given %s specifier
Vulnerable Buffer Code

```c
void call_echo() {
    echo();
}

/* Echo Line */
void echo() {
    char buf[8]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

unix> ./buf-nsp
Enter string:
12345678901234567890123
12345678901234567890123
unix>
```
```
```c
void call_echo() {
    echo();
}
```

Buffer Overflow Disassembly (buf-nsp)

```
00000000004005c6 <echo>:
4005c6:  48 83 ec 18
        sub $0x18,%rsp
... calls printf ...
4005d9:  48 89 e7
        mov %rsp,%rdi
4005dc:  e8 dd fe ff ff
        callq 4004c0 <gets@plt>
4005e1:  48 89 e7
        mov %rsp,%rdi
4005e4:  e8 95 fe ff ff
        callq 400480 <puts@plt>
4005e9:  48 83 c4 18
        add $0x18,%rsp
4005ed:  c3
        retq
```

```c
void call_echo() {
    echo();
}
```

Buffer Overflow Example

Before call to gets

```
00000000004005ee <call_echo>:
4005ee:  48 83 ec 08
        sub $0x8,%rsp
4005f2:  b8 00 00 00 00
        mov $0x0,%eax
4005f7:  e8 ca ff ff ff
        callq 4005c6 <echo>
4005fc:  48 83 c4 08
        add $0x8,%rsp
400600:  c3
        retq
```

After call to gets

```
00000000004005f7: callq 4005c6 <echo>
4005f7:  e8 ca ff ff ff
        callq 4005c6 <echo>
4005fc:  48 83 c4 08
        add $0x8,%rsp
400600:  c3
        retq
```
Buffer Overflow Example #2 Explained

Malicious Use of Buffer Overflow: Code Injection Attacks

Peer Instruction Question

Example: the original Internet worm (1988)

Heartbleed (2014)
Dealing with buffer overflow attacks

1) Avoid overflow vulnerabilities
2) Employ system-level protections
3) Have compiler use “stack canaries”

1) Avoid Overflow Vulnerabilities in Code

- Use library routines that limit string lengths
  - fgets instead of gets (2nd 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[8];  /* Way too small! */
    fgets(buf, 8, stdin);
    puts(buf);
}
```

2) System-Level Protections

- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
  - Makes it difficult for hacker to predict beginning of inserted code
  - Example: Code from Slide 6 executed 5 times, address of variable local =
    - 0x7ffd19d3f8ac
    - 0x7ffe8a462c2c
    - 0x7ffe927c905c
    - 0x7ffefd5c27dc
    - 0x7fffa0175afc
  - Stack repositioned each time program executes

- Non-executable 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
  - Stack marked as non-executable
    - Do NOT execute code in Stack, Static Data, or Heap regions
    - Hardware support needed

3) Stack Canaries

- Basic Idea: place special value (“canary”) on stack just beyond buffer
  - Secret value known only to compiler
  - “After” buffer but before return address
  - Check for corruption before exiting function
- GCC implementation (now default)
  - -fstack-protector
  - Code back on Slide 14 (buf-nsp) compiled with -fno-stack-protector flag

```
```

**Protected Buffer Disassembly (buf£)**

```
echo:
    400638:  sub    $0x18,%rsp
    40063c:  mov    %fs:0x28,%rax
    400645:  mov    %rax,0x8(%rsp)
    40064a:  xor    %eax,%eax
    ...     ... call printf ...
    400656:  mov    %esp,%rax
    400659:  callq  400530 <gets@plt>
    40065e:  mov    %esp,%rdi
    400661:  callq  4004e0 <puts@plt>
    400666:  mov    0x8(%rsp),%rax
    40066b:  xor    %fs:0x28,%rax
    400674:  je     40067b <echo+0x43>
    400676:  callq  4004f0 <__stack_chk_fail@plt>
    40067b:  add    $0x18,%rsp
    40067f:  retq
```
### Setting Up Canary

**Before call to gets**

```c
/* Echo Line */
void echo()
{
    char buf[8]; /* Way too small! */
    gets(buf);
    puts(buf);  
}
```

**Stack frame for call_echo**

- Return address (8 bytes)
- Canary (8 bytes)

```asm
echos:  
...  
movq %fs:40, %rax  # Get canary  
movq %rax, 8(%rsp)  # Place on stack  
xorl %eax, %eax  # Erase canary  
...  
```

**Segment register (don’t worry about it)**

<table>
<thead>
<tr>
<th>buf</th>
<th>%rsp</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>37</td>
<td>36</td>
</tr>
<tr>
<td>35</td>
<td>34</td>
<td>33</td>
</tr>
<tr>
<td>32</td>
<td>31</td>
<td>30</td>
</tr>
</tbody>
</table>

**After call to gets**

**Input:** 1234567

### Checking Canary

**Stack frame for call_echo**

```c
/* Echo Line */
void echo()
{
    char buf[8]; /* Way too small! */
    gets(buf);
    puts(buf);  
}
```

**Stack frame for call_echo**

- Return address (8 bytes)
- Canary (8 bytes)

```asm
echos:  
...  
movq %fs:40, %rax  # retrieve from Stack  
movq %rax, 8(%rsp)  # compare to canary  
je .L2  # if same, OK  
call __stack_chk_fail  # else, FAIL  
.L6:  
...  
```

### Summary

1. **Avoid overflow vulnerabilities**
   - Use library routines that limit string lengths

2. **Employ system-level protections**
   - Randomized Stack offsets
   - Code on the Stack is not executable

3. **Have compiler use “stack canaries”**