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

Hex Address: 0x00007FFFFFFF

Hex Address: 0x400000

Hex Address: 0x000000
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

```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?
Reminder: x86-64/Linux Stack Frame

- **Caller’s Stack Frame**
  - Arguments (if > 6 args) for this call

- **Current/ Callee Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Old frame pointer (optional)
  - Saved register context
    (when reusing registers)
  - Local variables
    (if can’t be kept in registers)
  - “Argument build” area
    (If callee needs to call another
    function - parameters for function
    about to call, if needed)
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)
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

- 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 to a dst
  - `scanf`, `fscanf`, `sscanf`, when given `%s` specifier
Vulnerable Buffer Code

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

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

```
unix> ./buf-nsp
Enter string: 12345678901234567890123
12345678901234567890123
unix>
unix> ./buf-nsp
Enter string: 123456789012345678901234
Segmentation Fault
```
Buffer Overflow Disassembly (buf-nsp)

**echo:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4005c6</td>
<td>sub $0x18,%rsp</td>
<td>sub $0x18,%rsp</td>
</tr>
<tr>
<td>4005d9</td>
<td>mov %rsp,%rdi</td>
<td>mov %rsp,%rdi</td>
</tr>
<tr>
<td>4005dc</td>
<td>callq 4004c0 <a href="mailto:gets@plt">gets@plt</a></td>
<td>callq 4004c0 <a href="mailto:gets@plt">gets@plt</a></td>
</tr>
<tr>
<td>4005e1</td>
<td>mov %rsp,%rdi</td>
<td>mov %rsp,%rdi</td>
</tr>
<tr>
<td>4005e4</td>
<td>callq 400480 <a href="mailto:puts@plt">puts@plt</a></td>
<td>callq 400480 <a href="mailto:puts@plt">puts@plt</a></td>
</tr>
<tr>
<td>4005e9</td>
<td>add $0x18,%rsp</td>
<td>add $0x18,%rsp</td>
</tr>
<tr>
<td>4005ed</td>
<td>retq</td>
<td>retq</td>
</tr>
</tbody>
</table>

**call_echo:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4005ee</td>
<td>sub $0x8,%rsp</td>
<td>sub $0x8,%rsp</td>
</tr>
<tr>
<td>4005f2</td>
<td>mov $0x0,%eax</td>
<td>mov $0x0,%eax</td>
</tr>
<tr>
<td>4005f7</td>
<td>callq 4005c6 &lt;echo&gt;</td>
<td>callq 4005c6 &lt;echo&gt;</td>
</tr>
<tr>
<td>4005fc</td>
<td>add $0x8,%rsp</td>
<td>add $0x8,%rsp</td>
</tr>
<tr>
<td>400600</td>
<td>retq</td>
<td>retq</td>
</tr>
</tbody>
</table>
Buffer Overflow Stack

**Before call to gets**

Stack frame for call_echo

Return address (8 bytes)

16 bytes unused

- [3] [2] [1] [0]

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

```
echo:    subq $24, %rsp
...      
movq %rsp, %rdi
call gets
...  
```

`buf ← %rsp`

**Note:** addresses increasing right-to-left, bottom-to-top
Buffer Overflow Example

Before call to gets

Stack frame for call_echo

00 00 00 00
00 40 05 fc

16 bytes unused

[3] [2] [1] [0]

void echo()
{
    char buf[8];
    gets(buf);
    ...
}

echo:
    subq $24, %rsp
    ...
    movq %rsp, %rdi
    call gets
    ...

call_echo:

    ...
    4005f7: callq 4005c6 <echo>
    4005fc: add $0x8,%rsp
    ...

buf ← %rsp
Buffer Overflow Example #1

After call to gets

void echo()
{
    char buf[8];
    gets(buf);
    ... 
}

echo:
    subq $24, %rsp
    ...
    movq %rsp, %rdi
    call gets
    ...

call_echo:
    ...
    4005f7: callq 4005c6 <echo>
    4005fc: add $0x8, %rsp
    ...

buf ← %rsp

Note: Digit “N” is just 0x3N in ASCII!

unix> ./buf-nsp
Enter string: 12345678901234567890123
12345678901234567890123
Overflowed buffer, but did not corrupt state
Buffer Overflow Example #2

After call to gets

Stack frame for call_echo

<table>
<thead>
<tr>
<th>00</th>
<th>00</th>
<th>00</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>40</td>
<td>05</td>
<td>00</td>
</tr>
<tr>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
</tr>
<tr>
<td>30</td>
<td>39</td>
<td>38</td>
<td>37</td>
</tr>
<tr>
<td>36</td>
<td>35</td>
<td>34</td>
<td>33</td>
</tr>
<tr>
<td>32</td>
<td>31</td>
<td>30</td>
<td>39</td>
</tr>
<tr>
<td>38</td>
<td>37</td>
<td>36</td>
<td>35</td>
</tr>
<tr>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
</tr>
</tbody>
</table>

void echo()
{
    char buf[8];
    gets(buf);
    ...
}

call Echo:

unix> ./buf-nsp
Enter string: 123456789012345678901234

Segmentation Fault

Overflowed buffer and corrupted return pointer
Buffer Overflow Example #2 Explained

After return from echo

Stack frame for call_echo

00 00 00 00
00 40 05 00
34 33 32 31
30 39 38 37
36 35 34 33
32 31 30 39
38 37 36 35
34 33 32 31

buf ← %rsp

0000000000400500 <deregister_tm_clones>:
400500:    mov    $0x60104f,%eax
400505:    push   %rbp
400506:    sub    $0x601048,%rax
400510:    mov    %rsp,%rbp
400513:    jbe   400530
400515:    mov    $0x601048,%edi
400520:    mov    $0x601048,%edi
400525:    jmpq   *%rax
400527:    nopw   0x0(%rax,%rax,1)
400530:    pop    %rbp
400531:    retq

“Returns” to unrelated code, but continues!
Eventually segfaults on retq of deregister_tm_clones.
Malicious Use of Buffer Overflow: Code Injection Attacks

- 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
Peer Instruction Question

- smash_me is vulnerable to stack smashing!
- What is the minimum number of characters that gets must read in order for us to change the return address to a stack address (in Linux)?

Previous stack frame

```
    00 00 00 00
    00 40 05 fe
    ...  
    [0]
```

smash_me:
```
  subq $0x30, %rsp
  ... 
  movq %rsp, %rdi
  call gets
  ... 
```

A. 33
B. 36
C. 51
D. 54
E. We’re lost...
Exploits Based on Buffer Overflows

- Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines
- Distressingly common in real programs
  - Programmers keep making the same mistakes 😞
  - Recent measures make these attacks much more difficult
- Examples across the decades
  - Original “Internet worm” (1988)
  - Still happens!!
    - Heartbleed (2014, affected 17% of servers)
    - Cloudbleed (2017)
  - Fun: Nintendo hacks
    - Using glitches to rewrite code: https://www.youtube.com/watch?v=TqK-2JUQBUY
    - FlappyBird in Mario: https://www.youtube.com/watch?v=hB6eY73sLV0
Example: the original Internet worm (1988)

- Exploited a few vulnerabilities to spread
  - 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 with phony argument:
    - `finger "exploit-code padding new-return-addr"`
    - Exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker

- Scanned for other machines to attack
  - Invaded ~6000 computers in hours (10% of the Internet)
    - see June 1989 article in Comm. of the ACM
  - The young author of the worm was prosecuted...
Heartbleed (2014)

- Buffer over-read in OpenSSL
  - Open source security library
  - Bug in a small range of versions

- “Heartbeat” packet
  - Specifies length of message
  - Server echoes it back
  - Library just “trusted” this length
  - Allowed attackers to read contents of memory anywhere they wanted

- Est. 17% of Internet affected
  - “Catastrophic”
  - Github, Yahoo, Stack Overflow, Amazon AWS, ...
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

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

- 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
2) System-Level Protections

- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
    - Addresses will vary from one run to another
  - Makes it difficult for hacker to predict beginning of inserted code

- Example: Code from Slide 6 executed 5 times; address of variable `local` =
  - 0x7fffd19d3f8ac
  - 0x7ffe8a462c2c
  - 0x7ffe927c905c
  - 0x7ffefd5c27dc
  - 0x7fffa0175afc

- Stack repositioned each time program executes
2) System-Level Protections

- **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

Any attempt to execute this code will fail
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

```
unix> ./buf
Enter string: 12345678
12345678
unix> ./buf
Enter string: 123456789
*** stack smashing detected ***
```
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    %rsp,%rdi
400659:  callq  400530 <gets@plt>
40065e:  mov    %rsp,%rdi
400661:  callq  4004e0 ...
400668:  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

Stack frame for call_echo

Return address (8 bytes)

Canary (8 bytes)

[3] [2] [1] [0]

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

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

buf ← %rsp
Checking Canary

**After call to gets**

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

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

Input: 1234567

Stack frame for call_echo

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

buf ← %rsp
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”