Buffer Overflows
CSE 410 Winter 2017

Instructor: Justin Hsia
Teaching Assistants: Kathryn Chan, Kevin Bi, Ryan Wong, Waylon Huang, Xinyu Sui
Administrivia

- Lab 2 & mid-quarter survey due *tonight*
- Lab 3 released today, due next Thursday (2/23)

**Midterm grades** (out of 46)
- **Mean:** 31.6 (69%), **Median:** 32.05, **Std Dev:** 6.89
- 1) Look at solutions, understand errors
- 2) Log in to Gradescope, look at detailed rubric items
- Regrade requests open until end of Thursday (2/16)
  - It is possible for your grade to go *down*
  - Make sure you submit separate requests for each portion of a question (*e.g.* Q4.1 and Q4.2) – these may go to different graders!

**Midterm Clobber Policy**
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

```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)
  - “Argument build” area (If callee needs to call another function -parameters for function about to call, if needed)
The Internet Worm

- These 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

- November 1988
  - Internet Worm attacks thousands of Internet hosts
  - Stack buffer overflow exploits!
Buffer Overflow in a nutshell

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

- Many Unix/Linux/C functions don’t check argument sizes
- C does not check array bounds
  - Allows overflowing (writing past the end) of buffers (arrays)
- 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
  - In particular, try to change the return address of the current procedure!
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();
}
```

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

echo:

```
00000000004005c8 <echo>:
  4005c8:  48 83 ec 18  # sub $0x18,%rsp
  ... calls printf ...
  4005de:  48 89 e7    # mov %rsp,%rdi
  4005e3:  e8 dd fe ff ff  # callq 4004c0 <gets@plt>
  4005e6:  e8 95 fe ff ff  # callq 400480 <puts@plt>
  4005eb:  48 83 c4 18  # add $0x18,%rsp
  4005ef:  c3
```

call_echo:

```
00000000004005f0 <call_echo>:
  4005f0:  48 83 ec 08  # sub $0x8,%rsp
  4005f4:  b8 00 00 00 00  # mov $0x0,%eax
  4005f9:  e8 ca ff ff ff  # callq 4005c8 <echo>
  4005fe:  48 83 c4 08  # add $0x8,%rsp
  400602:  c3  # retq
```
Buffer Overflow Stack

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

Before call to gets

Stack frame for call_echo

Return address (8 bytes)

16 bytes unused

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

Note: addresses increasing right-to-left, bottom-to-top

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

Before call to gets

Stack frame for call_echo

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>40</td>
<td>05</td>
<td>fe</td>
</tr>
</tbody>
</table>

16 bytes unused

[7][6][5][4][3][2][1][0]

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

call_echo:

... 4005f9: callq 4005c8 <echo>
     4005fe: add $0x8,%rsp
...

buf ← %rsp

echo:

subq $24, %rsp
...

movq %rsp, %rdi
call gets
...

[Image of stack frame and assembly code]
Buffer Overflow Example #1

After call to gets

Stack frame for call_echo

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>40</td>
<td>05</td>
<td>fe</td>
</tr>
<tr>
<td>00</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>

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

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

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

call_echo:

```
...
4005f9: callq 4005c8 <echo>
4005fe: add $0x8,%rsp
...
```

```
4005f9: callq 4005c8 <echo>
4005fe: add $0x8,%rsp
...
```

unix> ./buf-nsp
Enter string: 12345678901234567890123 01234567890123456789012

Overflowed buffer, but did not corrupt state

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

buf ← %rsp

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

call_echo:

. . .
4005f9: callq 4005c8 <echo>
4005fe: add $0x8,%rsp
. . .

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

Overflowed buffer and corrupted return pointer

unix> ./buf-nsp
Enter string: 12345678901234567890123
Segmentation Fault
Buffer Overflow Example #2 Explained

“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

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

void foo() {
    bar();
    A: ...
}
```

Stack after call to `gets()`

- `foo` stack frame
- `bar` stack frame
- `buf` starts here
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)
  - *Fun:* Nintendo hacks
    - Using glitches to rewrite code:  [https://www.youtube.com/watch?v=TqK-2jUQBUY](https://www.youtube.com/watch?v=TqK-2jUQBUY)
    - FlappyBird in Mario:  [https://www.youtube.com/watch?v=hB6eY73sLV0](https://www.youtube.com/watch?v=hB6eY73sLV0)

- You will learn some of the tricks in Lab 3
  - Hopefully to convince you to never leave such holes in your programs!!
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](https://doi.org/10.1145/55627.55643) 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, ...
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)?

<table>
<thead>
<tr>
<th>Previous stack frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00 40 05 fe</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>[0]</td>
</tr>
</tbody>
</table>

```
smash_me:
  subq $0x30, %rsp
  ...
  movq %rsp, %rdi
  call gets
  ...
```
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
    - 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`
  - 0x7ffd19d3f8ac
  - 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

```bash
unix> ./buf
Enter string: 12345678
12345678

unix> ./buf
Enter string: 123456789
*** stack smashing detected ***
```
Protected Buffer Disassembly (buf)

```
ProtectedBuffer Disassembly (buf)

...     ... call printf ...
...     ...

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 <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

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

Segment register (don’t worry about it)

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

buf ← %rsp

Stack frame for `call_echo`

Return address (8 bytes)

Canary (8 bytes)

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

Stack frame for `call_echo`

Return address (8 bytes)

Canary (8 bytes)

[3] [2] [1] [0]
Checking Canary

After call to gets

Stack frame for call_echo

Return address (8 bytes)

Canary (8 bytes)

00 37 36 35
34 33 32 31

Input: 1234567

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

```
...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: ...            
```

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”