Buffer Overflows
CSE 351 Summer 2018

Instructor:
Justin Hsia

Teaching Assistants:
Josie Lee
Natalie Andreeva
Teagan Horkan

http://xkcd.com/163/
Administrivia

- Mid-quarter survey due tonight (7/20)
- Homework 3 due Monday (7/23)
- Lab 3 due next Friday (7/27)

- Midterm grades (out of 100) released
  - Solutions posted on website
  - Rubric and grades found on Gradescope
  - Regrade requests will be open until Sunday (7/22) @ 5 pm
    - Must include reason based on solutions and rubric
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: 0x000000
0x000000
0x400000
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 be called, 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?

pointer to start of an array
same as:
\*p = c;
p++;
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> ./buf-nsp
Enter string: 123456789012345678901234
Segmentation Fault
```
Disassembly (buf-nsp)

echo:

```
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
```

call_echo:

```
00000000004005ee <call_echo>:
  4005ee:  48 83 ec 08  sub  $0x8,%rsp
  4005f2:  b8 00 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
```
Buffer Overflow Stack

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
### Buffer Overflow Example

**Before call to gets**

<table>
<thead>
<tr>
<th>Stack frame for call_echo</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00 40 05 fc</td>
</tr>
</tbody>
</table>

16 bytes unused

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[3] [2] [1] [0]</td>
</tr>
</tbody>
</table>

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, %rdi

buf ← %rsp
Buffer Overflow Example #1

After call to gets

Stack frame for call_echo

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

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

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

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);
    ...
}

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

call_echo:

    ...
    4005f7: callq 4005c8 <echo>
    4005fc: add $0x8,%rsp
    ...

buf ← %rsp

unix> ./buf-nsp
Enter string: 123456789012345678901234
Segmentation Fault

Overflowed buffer and corrupted return pointer
Buffer Overflow Example #2

After return from echo

Stack frame for call_echo

```
%rsp
0 0 0 0
0 0 40 05
34 33 32 31
30 39 38 37
36 35 34 33
32 31 30 39
38 37 36 35
34 33 32 31
```

```
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
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 00 40 05 fc | 00 00 00 00 00 00 00 00 00 00 00 00 00 00
|-----------------------------|--------------------------------------------------

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
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

```c
/* 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` = 
  - 0x7ffd19d3f8ac
  - 0x7ffe8a462c2c
  - 0x7ffe927c905c
  - 0x7ffefd5c27dc
  - 0x7fffa0175afc

- Stack repositioned each time the 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 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 <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

- **Stack frame for call_echo**
- **Return address (8 bytes)**
- **Canary (8 bytes)**

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

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

**Input:** `1234567`

---

This is extra (non-testable) material
Roadmap

C:
```c
#define car *
c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:
```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();
```

Assembly language:
```assembly
get_mpg:
pushq %rbp
movq %rsp, %rbp
...
popq %rbp
ret
```

Machine code:
```
0111010000011000 1000110100000100000000101000100111000010110000011111101000011111
```

Computer system:
```
```

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Aside: Units and Prefixes

- Here focusing on large numbers (exponents > 0)
- Note that $10^3 \approx 2^{10}$
- SI prefixes are *ambiguous* if base 10 or 2
- IEC prefixes are *unambiguously* base 2

SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

<table>
<thead>
<tr>
<th>SI Size</th>
<th>Prefix</th>
<th>Symbol</th>
<th>IEC Size</th>
<th>Prefix</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^3</td>
<td>Kilo-</td>
<td>K</td>
<td>2^{10}</td>
<td>Kibi-</td>
<td>Ki</td>
</tr>
<tr>
<td>10^6</td>
<td>Mega-</td>
<td>M</td>
<td>2^{20}</td>
<td>Mebi-</td>
<td>Mi</td>
</tr>
<tr>
<td>10^9</td>
<td>Giga-</td>
<td>G</td>
<td>2^{30}</td>
<td>Gibi-</td>
<td>Gi</td>
</tr>
<tr>
<td>10^{12}</td>
<td>Tera-</td>
<td>T</td>
<td>2^{40}</td>
<td>Tebi-</td>
<td>Ti</td>
</tr>
<tr>
<td>10^{15}</td>
<td>Peta-</td>
<td>P</td>
<td>2^{50}</td>
<td>Pebi-</td>
<td>Pi</td>
</tr>
<tr>
<td>10^{18}</td>
<td>Exa-</td>
<td>E</td>
<td>2^{60}</td>
<td>Exbi-</td>
<td>Ei</td>
</tr>
<tr>
<td>10^{21}</td>
<td>Zetta-</td>
<td>Z</td>
<td>2^{70}</td>
<td>Zebi-</td>
<td>Zi</td>
</tr>
<tr>
<td>10^{24}</td>
<td>Yotta-</td>
<td>Y</td>
<td>2^{80}</td>
<td>Yobi-</td>
<td>Yi</td>
</tr>
</tbody>
</table>
How to Remember?

- Will be given to you on Final reference sheet

- Mnemonics
  - There unfortunately isn’t one well-accepted mnemonic
    - But that shouldn’t stop you from trying to come with one!
  - Killer Mechanical Giraffe Teaches Pet, Extinct Zebra to Yodel
  - Kirby Missed Ganondorf Terribly, Potentially Exterminating Zelda and Yoshi
  - xkcd: Karl Marx Gave The Proletariat Eleven Zeppelins, Yo
    - https://xkcd.com/992/
  - Post your best on Piazza!
How does execution time grow with SIZE?

```c
int array[SIZE];
int sum = 0;

for (int i = 0; i < 200000; i++) {
    for (int j = 0; j < SIZE; j++) {
        sum += array[j];
    }
}
```

Plot
Actual Data

![Graph showing the relationship between size and time. The graph has a linear trend line indicating a direct proportionality between the two variables.](image-url)