Buffer Overflows
CSE 351 Autumn 2018

Instructor:
Justin Hsia

Teaching Assistants:
Akshat Aggarwal
Brian Dai
Kevin Bi
Sophie Tian
An Wang
Britt Henderson
Kory Watson
Teagan Horkan
Andrew Hu
James Shin
Riley Germundson

http://xkcd.com/804/
Administrivia

- Mid-quarter survey due tomorrow (11/1)
- Homework 3 due Friday (11/2)
- Lab 3 released today, due next Friday (11/9)

- Midterm grades (out of 100) to be released by Friday
  - Solutions posted on website
  - Rubric and grades will be found on Gradescope
  - Regrade requests will be open for a short time after grade release
Peer Instruction Question

- Minimize the size of the struct by re-ordering the vars

```c
struct old {
    int i;
    short s[3];
    char *c;
    float f;
};
```

```c
struct new {
    int i;
    float f;
    char *c;
    short s[3];
};
```

- What are the old and new sizes of the struct?

  ```
  sizeof(struct old) = \_\_\_ B
  sizeof(struct new) = \_\_\_ B
  ```

  A. 16 bytes  
  B. 22 bytes  
  C. 28 bytes  
  D. 32 bytes  
  E. We’re lost...

Vote on sizeof(struct old): http://PollEv.com/justinh
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

\[
\begin{align*}
0x00000000 &\quad \text{not drawn to scale} \\
0x00007fffffff &\quad \text{8 MiB} \\
0x400000 &\quad \text{OS stuff}
\end{align*}
\]
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?  

- `p1` → stack address
- `*p1` → heap address
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?

- End of file
- Pointer to start of an array (don't know size!)
- Newline
- Same as:
  ```c
  *p = c;
  p++;
  ```
- Reads character from input stream
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

**Stop condition looking for special characters**
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: 123456789012345
123456789012345
unix> ./buf-nsp
Enter string: 123456789012345
6
Segmentation Fault
unix> ./buf-nsp
Enter string: 123456789012345
67
Illegal instruction
```

The example code demonstrates a vulnerable buffer overflow. The `gets()` function reads input into a buffer that is too small, causing a buffer overflow. This results in an illegal instruction or segmentation fault when the code is executed.
Buffer Overflow Disassembly (buf-nsp)

echo:

```
0000000000400597 <echo>:
  400597: 48 83 ec 18
  ... calls printf ...
  4005aa: 48 8d 7c 24 08
  4005af: e8 d6 fe ff ff
  4005b4: 48 89 7c 24 08
  4005b9: e8 b2 fe ff ff
  4005be: 48 83 c4 18
  4005c2: c3
```

```
sub $0x18,%rsp
```

```
lea 0x8(%rsp),%rdi
```

```
callq 400480 <gets@plt>
```

```
lea 0x8(%rsp),%rdi
```

```
callq 4004a0 <puts@plt>
```

```
add $0x18,%rsp
retq
```

call_echo:

```
00000000004005c3 <call_echo>:
  4005c3: 48 83 ec 08
  4005c7: b8 00 00 00 00
  4005cc: e8 c6 ff ff ff
  4005d1: 48 83 c4 08
  4005d5: c3
```

```
sub $0x8,%rsp
```

```
mov $0x0,%eax
```

```
callq 400597 <echo>
```

```
add $0x8,%rsp
retq
```

```
return address
```

```
placed on stack
```
Buffer Overflow Stack

Before call to gets

Stack frame for call_echo

Return address (8 bytes)

8 bytes unused

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

8 bytes unused

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 d1</td>
</tr>
</tbody>
</table>

### 8 bytes unused


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

### 8 bytes unused

- `buf ← %rsp`

## `void echo()`

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

## `call_echo`

- `callq 400597 <echo>`
- `add $0x8,%rsp`
- `8 bytes unused`

## `buf`
Buffer Overflow Example #1

After call to `gets`

<table>
<thead>
<tr>
<th>Stack frame for call_echo</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00 35 34 33</td>
</tr>
<tr>
<td>32 31 30 39</td>
</tr>
<tr>
<td>38 37 36 35</td>
</tr>
<tr>
<td>34 33 32 31</td>
</tr>
<tr>
<td>8 bytes unused</td>
</tr>
</tbody>
</table>

Note: Digit “\(N\)” is just 0x3\(N\) in ASCII!

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

echo:
```c
subq $24, %rsp
...
leaq 8(%rsp), %rdi
call gets
...
```

`callq` 400597 `<echo>`

`add 8(%rdi), %rax`

`lea 8(%rdi), %rdx`

`mov 0x31, %rdx`

Note: Digit “\(N\)” is just 0x3\(N\) in ASCII!

`./buf-nsp`

```
Enter string: 123456789012345
123456789012345
Overflowed buffer, but did not corrupt state
```
Buffer Overflow Example #2

**After call to gets**

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

call_echo:

```assembly
4005cc: callq 400597 <echo>
4005d1: add $0x8,%rsp
...
```

unix> ./buf-nsp
Enter string: 1234567890123456
Illegal instruction

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

After return from echo

Stack frame for call_echo

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

%rsp

buf

8 bytes unused

```
00000000004004f0 <deregister_tm_clones>:
4004f0:  push   %rbp
4004f1:  mov $0x601040,%eax
4004f6:  cmp $0x601040,%rax
4004fc:  mov %rsp,%rbp
4004ff:  je     400518 ...
        pop    %rbp
400501:  mov $0x0,%edi
400506:  test %rax,%rax
400509:  je     400518
40050b:  pop   %rbp
40050c:  mov $0x601040,%edi
400511:  jmpq  *%rax
400513:  nopl  0x0(%rax,%rax,1)
400518:  pop   %rbp
400519:  retq
```

“Returns” to a byte that is not the beginning of an instruction, so program signals SIGILL, Illegal instruction.
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 `return`, will jump to exploit code

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

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

Stack after call to `gets()`

- High Addresses
- foo stack frame
- bar stack frame

- Low Addresses
- A B
- B starts here
- pad
- exploit code
- A(B)
- `%rsp`
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 x86-64 Linux)?

A. 27
B. 30
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](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)
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/10.1145/57373.57389) 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

```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 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

Stack after call to `gets()`

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:

400607: sub $0x18,%rsp
40060b: mov %fs:0x28,%rax # read canary value
400614: mov %rax,0x8(%rsp) # store canary on Stack
400619: xor %eax,%eax # erase canary from register

...  ... call printf ... 
400625: mov %rsp,%rdi
400628: callq 400510 <gets@plt>
40062d: mov %rsp,%rdi
400630: callq 4004d0 <puts@plt>
400635: mov 0x8(%rsp),%rax # read current canary on Stack
40063a: xor %fs:0x28,%rax # compare against original value
400643: jne 40064a <echo+0x43> # if unchanged, then return
400645: add $0x18,%rsp
400649: retq
40064a: callq 4004f0 <__stack_chk_fail@plt> # stack smashing detected

try:  diff buf-nsp.s bufs
# Setting Up Canary

## Before call to gets

<table>
<thead>
<tr>
<th>Stack frame for call_echo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return address (8 bytes)</td>
</tr>
<tr>
<td>Canary (8 bytes)</td>
</tr>
</tbody>
</table>

This is extra (non-testable) material

```c
/* 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

Stack frame for call_echo

Return address (8 bytes)

Canary (8 bytes)

00 37 36 35
34 33 32 31

Input: 1234567

/* 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
jne .L4 # if not same, FAIL

.L4: call __stack_chk_fail

This is extra (non-testable) material
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”