**Buffer Overflows**

**CSE 351 Spring 2018**

**HOW THE HEARTBLEED BUG WORKS:**

**SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).**

User Meg wants these 6 letters: POTATO.

Secure connection using key 453853873420.

User Meg wants pages about 'iri games': Unlocking secure records with master key 513995733434.

**SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).**

User Meg wants these 6 letters: POTATO.

Secure connection using key 453853873424.

User Meg wants pages about 'iri games': Unlocking secure records with master key 513995733434.

**SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).**

User Meg wants these 6 letters: POTATO.

Secure connection using key 453853873424.

User Meg wants pages about 'iri games': Unlocking secure records with master key 513995733434.

**SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).**

User Meg wants these 500 letters: HAT.

Lucas requests the 'missed connections' page. Eve (administrator) wants to set server's master key to '1493500543'. Isabel wants pages about snakes but not too long. User Karen wants to change account password to 'isabelpass'.
Administrivia

- Midterm on Friday: Bring your student ID and sit with your section (see web page)
  - No note sheets allowed, but we will provide the reference sheet linked from the web page.
  - Remember that we have office hours today and Thursday, come ask if you have questions!
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
0x000000
0x400000
```

Diagram:

- Stack
- Heap
- Shared Libraries
- Data
- Instructions
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?

- 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: 12345678901234567890123
Segmentation Fault
Buffer Overflow Disassembly (buf-nsp)

echo:

```
00000000004005c6 <echo>:
  4005c6: 48 83 ec 18
  ...                          ...
  4005dc: e8 dd fe ff ff
  4005e1: 48 89 e7
  4005e4: e8 95 fe ff ff
  4005e9: 48 83 c4 18
  4005ed: c3

sub $0x18,%rsp
... calls printf ...
mov %rsp,%rdi
callq 4004c0 <gets@plt>
mov %rsp,%rdi
callq 400480 <puts@plt>
add $0x18,%rsp
retq
```

call_echo:

```
00000000004005ee <call_echo>:
  4005ee: 48 83 ec 08
  4005f2: b8 00 00 00 00
  4005f7: e8 ca ff ff ff
  4005fc: 48 83 c4 08
  400600: c3

sub $0x8,%rsp
mov $0x0,%eax
callq 4005c6 <echo>
add $0x8,%rsp
retq
```

return address
Buffer Overflow Stack

Before call to `gets`

Stack frame for `call_echo`

Return address (8 bytes)

16 bytes unused

`void echo()`

```c
char buf[8]; /* Way too small! */
gets(buf);
puts(buf);
```

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

Before 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>fc</td>
</tr>
</tbody>
</table>

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

Stack frame for call_echo

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

Note: Digit “\(N\)” is just 0x3\(N\) 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`

<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 00</td>
</tr>
<tr>
<td>34 33 32 31</td>
</tr>
<tr>
<td>30 39 38 37</td>
</tr>
<tr>
<td>36 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>
</tbody>
</table>

### Code Example

**void echo()**

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

**Call Stack**

```c
void echo()
{
    subq $24, %rsp
    ...
    movq %rsp, %rdi
    call gets
    ...
}
```

### Stack Frame for `call_echo`

- `buf ← %rsp`

### Output

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

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

After return from echo

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

0000000000400500 <deregister_tm_clones>:

| 400500:mov $0x60104f,%eax |
| 400505:push %rbp |
| 400506:sub $0x601048,%rax |
| 40050c:cmp $0xe,%rax |
| 400510:mov %rsp,%rbp |
| 400513:jbe 400530 |
| 400515:mov $0x0,%eax |
| 40051a:test %rax,%rax |
| 40051d:je 400530 |
| 40051f:pop %rbp |
| 400520:mov $0x601048,%edi |
| 400525:jmpq *%rax |
| 400527:nopw 0x0(%rax,%rax,1) |
| 40052e:nop |
| 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

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

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

![Diagram showing stack after call to `gets()`](image-url)
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)?
  - Talk to your neighbor!

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

```
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, ...

![Heartbeat - Normal usage](image)

![Heartbeat - Malicious usage](image)

---

By FenixFeather - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=32276981
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` =
  - `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:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>operand 1</th>
<th>operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>400638</td>
<td>sub</td>
<td>0x18, %rsp</td>
<td></td>
</tr>
<tr>
<td>40063c</td>
<td>mov</td>
<td>%fs:0x28, %rax</td>
<td></td>
</tr>
<tr>
<td>400645</td>
<td>mov</td>
<td>%rax, 0x8(%rsp)</td>
<td></td>
</tr>
<tr>
<td>40064a</td>
<td>xor</td>
<td>%eax, %eax</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>... call printf...</td>
<td></td>
</tr>
<tr>
<td>400656</td>
<td>mov</td>
<td>%rsp, %rdi</td>
<td></td>
</tr>
<tr>
<td>400659</td>
<td>callq</td>
<td>400530 <a href="mailto:gets@plt">gets@plt</a></td>
<td></td>
</tr>
<tr>
<td>40065e</td>
<td>mov</td>
<td>%rsp, %rdi</td>
<td></td>
</tr>
<tr>
<td>400661</td>
<td>callq</td>
<td>4004e0 <a href="mailto:puts@plt">puts@plt</a></td>
<td></td>
</tr>
<tr>
<td>400666</td>
<td>mov</td>
<td>0x8(%rsp), %rax</td>
<td></td>
</tr>
<tr>
<td>40066b</td>
<td>xor</td>
<td>%fs:0x28, %rax</td>
<td></td>
</tr>
<tr>
<td>400674</td>
<td>je</td>
<td>40067b &lt;echo+0x43&gt;</td>
<td></td>
</tr>
<tr>
<td>400676</td>
<td>callq</td>
<td>4004f0 <a href="mailto:__stack_chk_fail@plt">__stack_chk_fail@plt</a></td>
<td></td>
</tr>
<tr>
<td>40067b</td>
<td>add</td>
<td>0x18, %rsp</td>
<td></td>
</tr>
<tr>
<td>40067f</td>
<td>retq</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 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)
  - [3] [2] [1] [0]

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

- `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)

Input: 1234567
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”
Now you’re prepared for lab 3!
(one of the most fun labs, IMO)

You, after finishing lab 3
Good luck studying for the midterm!

- Remember to bring your student ID on Friday.
- We will have office hours today and Thursday, so come see us if you have questions!