Machine-Level Programming: Buffer Overflow

CMPU 224 – Computer Organization
Jason Waterman
x86-64 Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)
  - E.g., local variables

- **Heap**
  - Dynamically allocated as needed
  - When `malloc()` is called

- **Data**
  - Statically allocated data
  - E.g., global vars, static vars, string constants

- **Text / Shared Libraries**
  - Executable machine instructions
  - Read-only
String Library Code

- Implementation of Linux function `gets()`
  - `gets()` reads a line from `stdin` into the buffer pointed to by `s` until either a terminating newline or EOF, which it replaces with a null byte
  - No way to specify limit on number of characters to read
- Similar problems with other library functions
  - `strcpy`, `strcat`: Copy strings of arbitrary length
  - `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification

```c
/* Get string from stdin */
char *gets(char *s){
    char *p = s;
    int c = getchar();
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return s;
}
```
NAME

gets - get a string from standard input (DEPRECATED)

SYNOPSIS

#include <stdio.h>
char *gets(char *s);

DESCRIPTION

Never use this function.

gets() reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with a null byte ('\0'). No check for buffer overrun is performed (see BUGS below).

RETURN VALUE

gets() returns s on success, and NULL on error or when end of file occurs while no characters have been read. However, given the lack of buffer overrun checking, there can be no guarantees that the function will even return.

BUGS

Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.
Vulnerable Buffer Code

```c
#include <stdio.h>

// Compile with:
// gcc -Og -fno-stack-protector -o bufdemo bufdemo.c

void echo(void) {  
    char buf[4];  // Way too small!!!
    gets(buf);
    puts(buf);
}

void call_echo(void) {  
    echo();
}

void main(void) {  
    puts('Type a string:');
    call_echo();
}
```

Linux>./bufdemo  
Type a string: 012  
012  

Linux>./bufdemo  
Type a string: 0123456789012345678901234  
Segmentation fault  

btw, how big is big enough?
Such problems are a BIG deal

• Generally called a “buffer overflow”
  • when exceeding the memory size allocated for an array

• Why a big deal?
  • It’s the #1 technical cause of security vulnerabilities

• Most common form
  • Unchecked lengths on string inputs
  • Particularly for bounded character arrays on the stack
    • sometimes referred to as stack smashing
### Buffer Overflow Disassembly

```c
#include <stdio.h>

// Compile with:
// gcc -Og -fno-stack-protector -o \ bufdemo bufdemo.c

void echo(void) {
    char buf[4]; // Way too small!!!
    gets(buf);
    puts(buf);
}

void call_echo(void) {
    echo();
}

void main(void) {
    puts("Type a string:");
    call_echo();
}
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Op Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>400614:</td>
<td>sub</td>
<td>48 83 ec 18</td>
<td>sub $0x18,%rsp</td>
</tr>
<tr>
<td>400618:</td>
<td>mov</td>
<td>48 89 e7</td>
<td>mov %rsp,%rdi</td>
</tr>
<tr>
<td>40061b:</td>
<td>callq</td>
<td>e8 a6 ff ff ff</td>
<td>callq 4005c6 &lt;gets&gt;</td>
</tr>
<tr>
<td>400620:</td>
<td>mov</td>
<td>48 89 e7</td>
<td>mov %rsp,%rdi</td>
</tr>
<tr>
<td>400623:</td>
<td>callq</td>
<td>e8 68 fe ff ff</td>
<td>callq 400490 <a href="mailto:puts@plt">puts@plt</a></td>
</tr>
<tr>
<td>400628:</td>
<td>add</td>
<td>48 83 c4 18</td>
<td>add $0x18,%rsp</td>
</tr>
<tr>
<td>40062c:</td>
<td>retq</td>
<td>c3</td>
<td></td>
</tr>
<tr>
<td>40062d:</td>
<td>sub</td>
<td>48 83 ec 8</td>
<td>sub $0x8,%rsp</td>
</tr>
<tr>
<td>400631:</td>
<td>callq</td>
<td>e8 de ff ff ff</td>
<td>callq 400614 &lt;echo&gt;</td>
</tr>
<tr>
<td>400636:</td>
<td>add</td>
<td>48 83 c4 8</td>
<td>add $0x8,%rsp</td>
</tr>
<tr>
<td>40063a:</td>
<td>retq</td>
<td>c3</td>
<td></td>
</tr>
<tr>
<td>40063b:</td>
<td>sub</td>
<td>48 83 ec 8</td>
<td>sub $0x8,%rsp</td>
</tr>
<tr>
<td>40063f:</td>
<td>mov</td>
<td>bf e4 06 40 00</td>
<td>mov $0x4006e4,%edi</td>
</tr>
<tr>
<td>400644:</td>
<td>callq</td>
<td>e8 47 fe ff ff</td>
<td>callq 400490 <a href="mailto:puts@plt">puts@plt</a></td>
</tr>
<tr>
<td>400649:</td>
<td>callq</td>
<td>e8 df ff ff ff</td>
<td>callq 40062d &lt;call_echo&gt;</td>
</tr>
<tr>
<td>40064e:</td>
<td>add</td>
<td>48 83 c4 8</td>
<td>add $0x8,%rsp</td>
</tr>
<tr>
<td>400652:</td>
<td>retq</td>
<td>c3</td>
<td></td>
</tr>
</tbody>
</table>
```
Buffer Overflow Stack

Stack before call to gets

Stack Frame for call_echo

Return Address (8 bytes)

void call_echo(void) {
    echo();
}

void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
**Buffer Overflow Stack**

Stack Frame for call_echo

Stack after call to gets

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

```
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

Linux> ./bufdemo
Type a string:
123
123

```
0000000000400614 <echo>:
  400614:  48 83 ec 18         sub $0x18,%rsp
  400618:  48 89 e7           mov %rsp,%rdi
  40061b:  e8 a6 ff ff ff     callq 4005c6 <gets>
  400620:  48 89 e7           mov %rsp,%rdi
  400623:  e8 68 fe ff ff     callq 400490 <puts@plt>
  400628:  48 83 c4 18       add $0x18,%rsp
  40062c:  c3               retq
```

---

Linux> ./bufdemo
Type a string:
123
123
Buffer Overflow Stack

Stack after call to gets

Stack Frame for call_echo

Stack after call to gets

0000000000400614 <echo>:
400614: 48 83 ec 18 sub $0x18,%rsp
400618: 48 89 e7 mov %rsp,%rdi
40061b: e8 a6 ff ff ff callq 4005c6 <gets>
400620: 48 83 c4 18 add $0x18,%rsp
40062c: c3 retq

void call_echo(void) {
    echo();
}

void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}

Linux>.bufdemo
Type a string:
12345678901234567890123
12345678901234567890123

Overflowed buffer but didn’t corrupt return address
Buffer Overflow Stack

Stack after call to gets

Stack Frame for call_echo

0000000000400614 <echo>:
400614: 48 83 ec 18 sub $0x18,%rsp
400618: 48 89 e7 mov %rsp,%rdi
40061b: e8 a6 ff ff ff callq 4005c6 <gets>
400620: 48 89 e7 mov %rsp,%rdi
400623: e8 68 fe ff ff callq 400490 <puts@plt>
400628: 48 83 c4 18 add $0x18,%rsp
40062c: c3 retq

void call_echo(void) {
    echo();
}

void echo() {
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}

Overflowed buffer and corrupted the return address

Linux>./bufdemo
Type a string:
123456789012345678901234
Segmentation fault

10/12/2018

CMPU 224 -- Computer Organization
Where did we go?

Stack after call to gets

Stack Frame for call_echo

<table>
<thead>
<tr>
<th>00</th>
<th>00</th>
<th>00</th>
<th>00</th>
<th>00</th>
<th>40</th>
<th>06</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
<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>
<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>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
</tr>
</tbody>
</table>

program crashes here

We now have the ability to hijack the program by jumping to arbitrary code anywhere in the program!
Code Injection Attacks

- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When Q executes ret, will jump to exploit code

```
void P()
{
    Q();
    ...
}

int Q()
{
    char buf[64];
    gets(buf);
    ...
    return ...;
}
```
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)
  • “IM wars” (1999)
  • Twilight hack on Wii (2000s)
  • ... and many, many more

• You will learn some of the tricks in attacklab
  • 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 jawaterman@mote.cs.vassar.edu`
  • Worm attacked fingerd server by sending phony argument:
    • `finger “exploit-code padding new-return-address”`
    • exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

• Once on a machine, 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...
  • and CERT was formed...
OK, what to do about buffer overflow attacks

• Avoid overflow vulnerabilities

• Employ system-level protections

• Have compiler use “stack canaries”

• Lets talk about each...
1. Avoid Overflow Vulnerabilities in Code (!)

- For example, use library routines that limit string lengths
  - \texttt{fgets} instead of \texttt{gets}
  - \texttt{strncpy} instead of \texttt{strcpy}
  - Don’t use \texttt{scanf} with \texttt{\%s} conversion specification
    - Use \texttt{fgets} to read the string
    - Or use \texttt{\%ns} where \texttt{n} is a suitable integer

```c
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```
2. System-Level Protections can help

- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
  - Makes it difficult for hacker to predict beginning of inserted code
2. System-Level Protections can help

- Nonexecutable 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

Any attempt to execute this code will fail
3. Stack Canaries can help

• Idea
  • Place special value ("canary") on stack just beyond buffer
  • Check for corruption before exiting function

• GCC Implementation
  • `-fstack-protector`
  • Now the default (disabled earlier)

```
Linux>./bufdemo-sp
Type a string:
0123456
0123456

Linux>./bufdemo-sp
Type a string:
01234567
*** buffer overflow detected ***: ./bufdemo-sp terminated
```
Protected Buffer Disassembly

echo:

40072f:  sub    $0x18,%rsp
400733:  mov    %fs:0x28,%rax  # Retrieve canary
40073c:  mov    %rax,0x8(%rsp) # Store on stack
400741:  xor    %eax,%eax      # Zero out Register
400743:  mov    %rsp,%rdi
400746:  callq  4006e0 <gets>
40074b:  mov    %rsp,%rdi
40074e:  callq  400570 <puts@plt>
400753:  mov    0x8(%rsp),%rax # Retrieve canary
400758:  xor    %fs:0x28,%rax  # Compare to stored value
400761:  je     400768 <echo+0x39>
400763:  callq  400580 <__stack_chk_fail@plt> # Stack bad!
400768:  add    $0x18,%rsp
40076c:  retq
Setting Up Canary

Stack before call to gets

Stack Frame for call_echo

- Return Address (8 bytes)
- Canary (8 bytes)

Stack before call to gets

void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}

/* Echo Line */

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

buf ← %rsp
Checking the Canary

/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}

Stack before call to gets

Stack Frame for
call_echo

Return Address (8 bytes)

Canary (8 bytes)

buf ← %rsp

Stack before call to gets

void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}

Stack Frame for
call_echo

Return Address (8 bytes)

Canary (8 bytes)

buf ← %rsp

movq 8(%rsp), %rax  # Retrieve from stack
xorq %fs:40, %rax  # Compare to canary
je .L6
# If same, OK

call __stack_chk_fail  # FAIL

.L6:

buf ← %rsp

movq 8(%rsp), %rax  # Retrieve from stack
xorq %fs:40, %rax  # Compare to canary
je .L6
# If same, OK

call __stack_chk_fail  # FAIL

.L6:
Return-Oriented Programming Attacks

• Challenge (for hackers)
  • Stack randomization makes it hard to predict buffer location
  • Marking stack non-executable makes it hard to insert binary code

• Alternative Strategy
  • Use existing code
    • E.g., library code from stdlib
  • String together fragments to achieve overall desired outcome
  • *Does not overcome stack canaries*

• Construct program from *gadgets*
  • Sequence of instructions ending in `ret`
    • Encoded by single byte `0xc3`
  • Code positions fixed from run to run
  • Code is executable
Gadget Example #1

```c
long ab_plus_c(long a, long b, long c) {
    return a*b + c;
}
```

Gadget address = 0x4004d4

- Use tail end of existing functions
Gadget Example #2

```c
void setval(unsigned *p) {
    *p = 3347663060u;
}
```

- Encodes `movq %rax, %rdi`
- `rdi ← rax`
- Gadget address = 0x4004dc

• Repurpose byte codes
ROP Execution

- Trigger with `ret` instruction
  - Will start executing Gadget 1
- Final `ret` in each gadget will start next one