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

Not drawn to scale
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 a 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.
#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();
}

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
#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();
}
Buffer Overflow Stack

Stack before call to gets

Stack Frame for call_echo

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

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);
}

Linux> ./bufdemo
Type a string:
123
123

2/21/2024
CMPU 224 -- Computer Organization

increasing memory addresses
Buffer Overflow Stack

Stack after call to gets

Stack Frame for call_echo

00 00 00 00 00 00 00 00 00 00 40 06 36
00 33 32 31 30 39 38 37
36 35 34 33 32 31 30 39
38 37 36 35 34 33 32 31

increasing memory addresses

void call_echo(void) {
    echo();
}

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

increasing memory addresses

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

20 bytes unused
buf ← %rsp

Overflowed buffer but didn’t corrupt return address

Stack after call to gets

Stack Frame for call_echo

increasing memory addresses

Linux>./bufdemo
Type a string:
12345678901234567890123
12345678901234567890123

2/21/2024
CMPU 224 -- Computer Organization
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);
}

increasing memory addresses

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

20 bytes unused
buf %rsp

increasing memory addresses

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

Overflowed buffer and corrupted the return address
Where did we go?

Actual assembly code near address 0x400600

Stack after call to gets

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 these tricks
  - Hopefully to convince you to never leave such holes in your programs!!
OK, what to do about buffer overflow attacks

• Avoid overflow vulnerabilities

• Employ system-level protections

• Have compiler use “stack canaries”

• Let's talk about each of these...
1. Avoid Overflow Vulnerabilities in Code (!)

• For example, use library routines that limit string lengths
  • fgets instead of gets
  • 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

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

- Randomized stack offsets (ASLR)
  - 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

- 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

Any attempt to execute this code will fail.
Return-Oriented Programming (ROP) 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

• Construct program from gadgets
  • Sequence of instructions ending in \texttt{ret}
    • Encoded by single byte \texttt{0xc3}
  • Code positions are fixed from run to run (until recently)
  • Code is executable
Gadget Example #1

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

```
00000000004004d0 <ab_plus_c>:
4004d0:  48 0f af fe  imul %rsi,%rdi
4004d4:  48 8d 04 17  lea (%rdi,%rdx,1),%rax
4004d8:  c3           retq
```

```
rax ← rdi + rdx
Gadget address = 0x4004d4
```

- Use tail end of existing functions
Gadget Example #2

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

Encodes `movq %rax, %rdi`

<setval>:
4004d9: c7 07 d4 48 89 c7  movl $0xc78948d4,(%rdi)
4004df: c3       retq

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
3. Stack Canaries can help

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

• GCC Implementation
  • \texttt{-fstack-protector}
  • Now the default (disabled earlier)

\begin{verbatim}
Linux>./bufdemo-sp
Type a string:
0123456
0123456

Linux>./bufdemo-sp
Type a string:
01234567

*** buffer overflow detected ***: ./bufdemo-sp terminated
\end{verbatim}
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>  # Zero when equal
400763:  callq  400580 <__stack_chk_fail@plt> # Stack bad!
400768:  add  $0x18,%rsp
40076c:  retq
```

Stack before call to gets

Stack Frame for call_echo

Return Address (8 bytes)

Canary (8 bytes)

buf ← %rsp
Summary of Buffer Overflow Attacks

• One of the leading causes of exploits for hackers!
• Fixes
  • Write in languages that don’t allow buffer overflows
    • But your code may still call a library in another language with a buffer overflow
  • ASLR makes it harder to predict stack addresses
    • But is not foolproof
  • Marking the stack as non-executable prevents code injection attacks
    • But not ROP attacks
  • Stack Canaries will prevent both code injection and ROP attacks
    • But there are still millions of lines of code out there without these protections