Buffer Overflows

Address space layout, the stack discipline, + C’s lack of bounds-checking = HUGE PROBLEM

Outline
Goal: how the stack + lack of bounds checking make C program vulnerable to a certain (serious!) type of security vulnerability

- Understanding buffer overflows
- Refresher on memory layout
- C library function: gets
- gets + echo buffer overflow example
- Simplified security exploit example
- Buffer overflows in the wild
- When this is a problem
- Real-world implications
- Unit summary

getaddrinfo()

x86-64 Linux memory layout

C: String library code

C standard library function gets()

What could go wrong when using this code?

Same problem in many C library functions:
- strcpy: Copies string of arbitrary length
- scanf, fscanf, sscanf, when given %s conversion specification
C: Vulnerable buffer code using gets (...)

```c
int main() {
    printf("Type a string:");
    echo();
    return 0;
}
```

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

These two lines of code introduce a vulnerability!

Vulnerable buffer code using gets:

```
00000000004006cf <echo>:
4006cf: 48 83 ec 18 sub $24,%rsp
4006e1: e8 a5 ff ff ff callq 400680 <gets>
4006e6: 48 89 a7 mov %rsp,%rdi
4006e9: a5 ff ff ff callq 400680 <gets>
4006ed: 48 89 a7 mov %rdi,%rdx
4006f0: 48 89 ef cmp %rdx,%ecx
4006f3: 75 04 jae 4006f8
4006f7: 48 83 c4 08 add $0x8,%rsp
4006f9: c3 retq
```

```c
void echo() {
    char buf[4];
    gets(buf);
}
```

Buffer overflow example: before input

Before call to gets

```
00 00 00 00 00 40 06 16
20 bytes unused
buf ← %rsp
```

Return Address

Call stack:

```
Return Address
call_echo:
        4006f1: callq 4006cf <echo>
        4006f6: add $0x8,%rsp...
```

Stack frame for `call_echo`

```
00 00 00 00 00 40 06 16
buf ← %rsp
```

```
00 00 00 00 32 31 30 39 38 37 36
buf ← %rsp
```

```
35 34 33 32 31 30 39 38 37 36
```

Null Terminator

Buffer overflow example: input #1

After call to gets

```
00 00 00 00 00 40 06 16
buf ← %rsp
```

Call stack:

```
Return Address
call_echo:
        4006f1: callq 4006cf <echo>
        4006f6: add $0x8,%rsp...
```

Stack frame for `call_echo`

```
00 00 00 00 32 31 30 39 38 37 36
buf ← %rsp
```

```
35 34 33 32 31 30 39 38 37 36
```

Null Terminator

Overflowed buffer, but did not corrupt state
Buffer overflow example: input #2

After call to gets

Stack frame for `call_echo`

Return Address

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

Null Terminator

```
buf⟵%rsp
```

Works because: "Returns" to unrelated code, despite what the C code had!
Lots of things happen, without modifying critical state
Eventually executes `retq` back to `main`

Exploiting buffer overflows

Stack after call to `gets`

```
... return address A
```

Data written by `gets`

```
pad
```

Exploit code

```
buf
```

Bar stack frame
Simplified exploit example (no padding)

```c
#include <stdio.h>

void delete_all_files() {
    // ... users shouldn't be able to call this
}
void read_input() {
    char buf[8];
    gets(buf);
}
int main() {
    read_input();

    read_input:
        401126: subq $8, %rsp
        40112a: leaq (%rsp), %rdi
        40112f: movl $0, %eax
        401134: call gets
        401139: addq $24, %rsp
        40113d: ret

    delete_all_files:
        40003e: call evil

    main:
        ...    ...
        400048: call read_input
        40004d: addq $8, %rsp
        400051: ret
```

Update the stack and registers diagram to the state at the red line.

Discuss: how long would the user input on standard in need for a buffer overflow attack? What address would we want to appear, and where, to delete all files?
Exploiting buffer overflows: when is this a problem?

We could construct x86 code to mess up our own programs call stack

But, we trust our own code to not!

The problem: allowing user input (untrusted source) to potentially corrupt the stack

Combination of: untrusted input, code that does not enforce bounds

gets(input); strcpy(input, ...); scanf(input, ...); etc

Exploits in the wild

Buffer overflow bugs allow remote attackers to execute arbitrary code on machines running vulnerable software.

1988: Internet worm

Early versions of the finger server daemon (fingerd) used `gets()` to read the argument sent by the client:

```
finger somebody@cs.wellesley.edu
```

Attack by sending phony argument:

```
finger "exploit-code padding new-return-address"
```

... Still happening

```
getaddrsinfo()
```

Heartbleed (2014)

Buffer over-read in OpenSSL

- Widely used encryption library (https)
- "Heartbeat" packet
  - Specifies length of message
  - Server echoes that much back
  - Library just "trusted" this length
  - Allowed attackers to read contents of memory anywhere they wanted
- ~17% of Internet affected
  - "Catastrophic"
- Github, Yahoo, Stack Overflow, Amazon AWS, ...

Avoiding overrun vulnerabilities

1. Use a memory-safe language (not C)!

2. If you have to use C, use library functions 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 `has` where `n` is a suitable integer

Other ideas?
System-level protections

Available in modern OSs/compilers/hardware
(We disabled these for buffer assignment.)

1. Randomize stack base, maybe frame padding
2. Detect stack corruption
   save and check stack "canary" values
3. Non-executable memory segments
   stack, heap, data, ... everything except text
   hardware support

Helpful, not foolproof!
Return-oriented programming, over-reads, etc.

Conclusion of unit: Hardware-Software Interface (ISA)

Lectures
(building on everything from HW)
- Programming with Memory
- x86 Basics
- x86 Control Flow
- x86 Procedures, Call Stack
- Representing Data Structures
- Buffer Overflows

Labs
(building on everything from HW)
- 7: Pointers in C
- 8: x86 Assembly
- 9: x86 Stack
- 10: Data structures in memory
- 11: Buffer overflows (less)

Topics
- C programming: pointers, dereferencing, arrays,
cursor-style programming, using malloc
- x86: instruction set architecture, machine code,
assembly language, reading/writing x86, basic
program translation
- Procedures and the call stack, data layout,
security implication

Assignments
- Pointers
- x86
- Buffer (less)

Mid-semester exam 2: ISA
November 16
(1 week from today)