Common Implementation Flaws

Dawn Song
dawnsong@cs.berkeley.edu

Goals for Today

- Next few lectures are about software security
  - Can have perfect design, specification, algorithms, but still have implementation vulnerabilities!
- Examine common implementation flaws
  - Many security-critical apps use C, and C has peculiar pitfalls
- Implementation flaws can occur with improper use of language, libraries, OS, or app logic
- Real goal:
  - Put on the attacker's hat: how to exploit a vulnerable program for fun & profit!

Simple Example

- ```
  char buf[80];
  void vulnerable() {
        gets(buf);
  }
  ```
- `gets()` reads all input bytes available on `stdin`, and stores them into `buf[]`
- What if input has more than 80 bytes?
  - gets() writes past end of `buf`, overwriting some other part of memory
  - This is a bug!
- Results?
  - Program crash/core-dump?
  - Much worse consequences possible…
  ```

Modified Example

```
char buf[80];
int authenticated = 0;
void vulnerable() {
    gets(buf);
}
```

- A `login` routine sets `authenticated` flag only if user proves knowledge of password
- What's the risk?
  - `authenticated` stored immediately after `buf`
  - Attacker "writes" data after end of `buf`
- Attacker supplies 81 bytes (81st set non-zero)
  - Makes `authenticated` flag true!
  - Attacker gains access: security breach!

More Serious Exploit Example

```
char buf[80];
int (*fnptr)();
void vulnerable() {
    gets(buf);
}
```

- Function pointer `fnptr` invoked elsewhere
- What can attacker do?
  - Can overwrite `fnptr` with any address, redirecting program execution!
- Crafty attacker:
  - Input contains malicious machine instructions, followed by pointer to overwrite `fnptr`
  - When `fnptr` is next invoked, flow of control re-directed to malicious code
- This is a *malicious code injection* attack

Buffer Overrun Vulnerabilities

- Most common class of implementation flaw (used to be)
  - Web application implementation flaw is taking over
- C does not guarantee type safety
  - Programmer exposed to bare machine
  - No bounds-checking for array or pointer accesses
- *Buffer overrun* (or *buffer overflow*) vulnerabilities
  - Out-of-bounds memory accesses used to corrupt program's intended behavior

Buffer Overrun Exploits

- **Demonstrate how adversaries might be able to use a buffer overrun bug to seize control**
  - **This is very bad!**
- **Consider: web server receives requests from clients and processes them**
  - **With a buffer overrun in the code, malicious client could seize control of server process**
  - **If server is running as root, attacker gains root access and can leave a backdoor**
    - **System has been "0wned"**
- **Buffer overrun vulnerabilities and malicious code injection attacks are primary/favorite method used by worm writers**

Buffer Overflow Exploit History

- **First Internet worm (Morris worm) spread using several attacks**
  - **One used buffer overrun to overwrite authenticated flag in `in.fingerd` (network finger daemon)**
- **Attackers have discovered much more effective methods of malicious code injection…**

C Program Memory Layout

- **Text region (program's executable code)**
- **Heap, (dynamically allocated data)**
  - **Grows/shrinks as objects allocated/freed**
- **Stack (local variable storage)**
  - **Grows/shrinks with function calls/returns**

text region	heap … stack
0x00…0	0xFF…F

- **Function call pushes new stack frame on stack**
  - **Frame includes space for function's local vars**
  - **Intel (x86) machines stack grows "down"**
  - **Stack pointer (SP) reg points to current frame**
  - **Stack extends from SP to the end of memory**

C Program Execution

- Instruction pointer (IP) register points to next machine instruction to execute
- Caller sets up arguments on stack
- Procedure call instruction:
  - Pushes current IP onto stack (return addr)
  - Jumps to beginning of function being called
- Compiler inserts prologue into each function
  - Pushes current SP value of SP onto stack
  - Allocates stack space for local variables by decrementing SP by appropriate amount
- Function return:
  - Old SP and return address retrieved from stack, and stack frame popped from stack
  - Execution continues from return address

Stack Smashing Attack

- ```
  void vulnerable() {
      char buf[80];
      gets(buf);
  }
  ```
- When `vulnerable()` is called, stack frame is pushed onto stack

buf	saved SP	ret addr	caller's stack frame	…

- Given "too-long" input, saved SP and return addr will be overwritten
- This is the stack smashing attack!

Stack Smashing Attack

- First, attacker stashes malicious code sequence somewhere in program's address space
- Next, attacker provides carefully-chosen 88-byte sequence
  - Last four bytes chosen to hold code's address overwrite saved return address
- When `vulnerable()` returns, CPU loads attacker's return addr – handing control over to attacker's malicious code
- Stack smashing exploit reference:
  - "Smashing the Stack for Fun and Profit," written by Aleph One in November 1996
  ```

Buffer Overrun Summary

- **Attackers developed techniques for when:**
  - **Buffer stored on the heap instead of on stack**
  - **Can only overflow buffer by one byte**
  - **Characters written to buffer are limited (e.g., only uppercase characters)**
  - **…**
- **Exploiting buffer overruns appears mysterious, complex, or incredibly hard to exploit**
  - **Reality – it is none of the above!**
- **Worms exploit these bugs all the time**
  - **Code Red II compromised 250K machines by exploiting IIS buffer overrun**

Format String Vulnerabilities

```
void vulnerable() {
  char buf[80];
  if (fgets(buf, sizeof buf, stdin) == NULL)
    return;
  printf(buf);
}
```

- **Do you see the bug?**
- **Last line should be `printf("%s", buf)`**
  - **If `buf` contains "`%`" chars, `printf()` will look for non-existent args, and may crash or core-dump trying to chase missing pointers**
- **Reality is worse…**

Attack Examples

- **Attacker can learn about function's stack frame contents if they can see what's printed**
  - **Use string "`%x:%x`" to see the first two words of stack memory**
- **What does this string ("`%x:%x:%s`") do?**
  - **Prints first two words of stack memory**
  - **Treats next stack memory word as memory addr and prints everything until first `'\0'`**
- **Where does that last word of stack memory come from?**
  - **Somewhere in `printf()`'s stack frame or, given enough `%x` specifiers to walk past end of `printf()`'s stack frame, comes from somewhere in `vulnerable()`'s stack frame**

A Further Refinement

- `buf` is stored in vulnerable()'s stack frame
  - Attacker controls `buf`'s contents and, thus, part of `vulnerable()`'s stack frame
  - Where `%s` specifier gets its memory addr!
- Attacker stores addr in `buf`, then when `%s` reads a word from stack to get an addr, it receives the addr they put there for it…
  - Exploit: `"\x04\x03\x02\x01:%x:%x:%x:%x:%s"`
  - Attacker arranges right number of `%x`'s, so addr is read from first word of `buf` (contains `0x01020304`)
  - Attacker can read any memory in victim's address space – crypto keys, passwords…

Yet More Troubles…

- Even worse attacks possible!
  - *If the victim has a format string bug*
- Use obscure format specifier (`%n`) to write any value to any address in the victim's memory
- Enables attackers to mount malicious code injection attacks
  - Introduce code anywhere into victim's memory
  - Use format string bug to overwrite return address on stack (or a function pointer) with pointer to malicious code

Format String Bug Summary

- Any program that contains a format string bug can be exploited by an attacker
  - Gains control of victim's program and all privileges it has on the target system
- Format string bug, like buffer overruns, are nasty business

Administravia

- Group partner sign-up
  - Use newsgroup to find partner
- HW1 graded

Mean:	42.4
Standard deviation:	12.3
Minimum:	6.0
1st quartile:	35.0
2nd quartile (median):	48.0
3rd quartile:	53.0
Maximum:	55.0

Another Vulnerability

-
```
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > sizeof buf) {
        error("length too large, nice try!");
        return;
    }
    memcpy(buf, p, len);
}
```
- What's wrong with this code?
- Hint – `memcpy()` prototype:
  - `void *memcpy(void *dest, const void *src, size_t n);`
- Definition of `size_t`: `typedef unsigned int size_t;`
- Do you see it now?

Implicit Casting Bug

- Attacker provides a negative value for `len`
  - `if` won't notice anything wrong
  - Execute `memcpy()` with negative third arg
  - Third arg is implicitly cast to an `unsigned int`, and becomes a very large positive int
  - `memcpy()` copies huge amount of memory into `buf`, yielding a buffer overrun!
- A signed/unsigned or an implicit casting bug
  - Very nasty – hard to spot
- C compiler doesn't warn about type mismatch between `signed int` and `unsigned int`
  - Silently inserts an implicit cast

Another Example

- ```
  size_t len = read_int_from_network();
  char *buf;
  buf = malloc(len+5);
  read(fd, buf, len);
  ...
  ```
- **What's wrong with this code?**
  - **No buffer overrun problems (5 spare bytes)**
  - **No sign problems (all ints are unsigned)**
- **But, `len+5` can overflow if `len` is too large**
  - **If `len = 0xFFFFFFFF`, then `len+5` is 4**
  - **Allocate 4-byte buffer then read a lot more than 4 bytes into it: classic buffer overrun!**
- **You have to know programming language's semantics very well to avoid all the pitfalls**
  ```