CS194-3/CS16x
Introduction to Systems

Lecture 19

Software Flaws

October 31, 2007

Prof. Anthony D. Joseph
http://www.cs.berkeley.edu/~adj/cs16x

Goals for Today

- Software distribution – access control, authorization, involuntary installation
- Enforcement
- Software security
  - Can have perfect design, specification, algorithms, but still have implementation vulnerabilities!
- Examine common implementation flaws in C
- Implementation flaws can occur with improper use of language, libraries, OS, or app logic

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Slides courtesy of Kubiatowicz, AJ Shankar, George Necula, Alex Aiken, Eric Brewer, Ras Bodik, Ion Stoica, Doug Tygar, and David Wagner.

How fine-grained should access control be?

- Example of the problem:
  - You buy a copy of a new game from "Joe's Game World"
  - It runs with your userid and deletes all your files!!
- How can you prevent this?
  - Create a *games* userid with no write privileges (like Unix 'nobody')
  - What if the game needs to write out a file recording scores?
    - » Give it write privileges to one file (or dir) to *games* userid
  - But what about non-game programs, such as Quicken?
    - » Create a *quicken* userid to prevent access to non-quicken-related files
  - But – how to get this right??? Pretty complex…

Authorization Continued

- Principle of least privilege: programs, users, and systems should get only enough privileges to perform their tasks
  - Very hard to do in practice
    - » How do you figure out what the minimum set of privileges is needed to run your programs?
  - People often run at higher privilege then necessary
    - » Such as the "administrator" privilege under windows
- One solution: Signed Software
  - Only use software from sources that you trust, thereby dealing with the problem by means of authentication
  - Fine for big, established firms such as Microsoft, since they can make their signing keys well known and people trust them
    - » Actually, not always fine: recently, one of Microsoft's signing keys was compromised, leading to malicious software that looked valid
  - What about new startups?
    - » Who "validates" them?
    - » How easy is it to fool them?

Pre-Installed Software

- Can I really trust software installed by computer maker?
- No! Most major computer manufacturers have shipped computers with viruses
  - How? Forgot to update virus scanner on "gold" master PC
- Software companies, PR firms, and others routinely release software that contains viruses

Involuntary Installation

- What about software loaded without your consent?
  - Macros attached to documents (such as Microsoft Word)
  - Active X controls (programs on web sites with potential access to whole machine)
  - Spyware included with normal products
- Active X controls can have access to the local machine
  - Install software/Launch programs
- Sony Spyware (October 2005)
  - About 50 recent CDs from Sony automatically install software when you played them on Windows machines
    - » Called XCP (Extended Copy Protection)
    - » Modify operating system to prevent more than 3 copies and to prevent peer-to-peer sharing
  - Side Effects:
    - » Reporting of private information to Sony
    - » Hiding of generic file names of form $sys_xxx; easy for other virus writers to exploit
    - » Hard to remove (crashes machine if not done carefully)
  - Vendors of virus protection software declare it spyware
    - » Computer Associates, Symantec, even Microsoft

Enforcement

- Enforcer checks passwords, ACLs, etc
  - Makes sure the only authorized actions take place
  - Bugs in enforcer⇒things for malicious users to exploit
- In UNIX, superuser can do anything
  - Because of coarse-grained access control, lots of stuff has to run as superuser in order to work
  - If there is a bug in any one of these programs, you lose!
- Paradox
  - Bullet-proof enforcer
    - » Only known way is to make enforcer as small as possible
    - » Easier to make correct, but simple-minded protection model
  - Fancy protection
    - » Tries to adhere to principle of least privilege
    - » Really hard to get right
- Same argument for Java or C++: What do you make private vs public?
  - Hard to make sure that code is usable but only necessary modules are public
  - Pick something in middle? Get bugs and weak protection!

State of the World

- State of the World in Security
  - Authentication: Encryption
    - » But almost no one encrypts or has public key identity
  - Authorization: Access Control
    - » But many systems only provide very coarse-grained access
    - » In UNIX, need to turn off protection to enable sharing
  - Enforcement: Kernel mode
    - » Hard to write a million line program without bugs
    - » Any bug is a potential security loophole!
- Some types of security problems
  - Abuse of privilege
    - » If the superuser is evil, we're all in trouble/can't do anything
    - » What if sysop in charge of instructional resources went crazy and deleted everybody's files (and backups)???
  - Imposter: Pretend to be someone else
    - » Example: in unix, can set up an .rhosts file to allow logins from one machine to another without retyping password
    - » Allows "rsh" command to do an operation on a remote node
    - » Result: send rsh request, pretending to be from trusted user→install .rhosts file granting you access

Some Security Problems

- Virus:
  - A piece of code that attaches itself to a program or file so it can spread from one computer to another, leaving infections as it travels
  - Most attached to executable files, so don't get activated until the file is actually executed
  - Once caught, can hide in boot tracks, other files, OS
- Worm:
  - Similar to a virus, but capable of traveling on its own
  - Takes advantage of file or information transport features
  - Because it can replicate itself, your computer might send out hundreds or thousands of copies of itself
- Trojan Horse:
  - Named after huge wooden horse in Greek mythology given as gift to enemy; contained army inside
  - At first glance appears to be useful software but does damage once installed or run on your computer

Buffer Overrun Vulnerabilities

- Most common class of implementation flaw
- C is basically a portable assembler
  - 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

Administrivia

- Project 2 code due Thursday 11/1
- Midterm 2 Exam:
  - Thursday 11/8 5:30-7pm, 405 Soda
  - We'll provide pizza and drinks

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 (81^{st} set non-zero)
  - Makes `authenticated` flag true!
  - Attacker gains access: security breach!

More Serious Exploit Example

```
char buf[80];
int (*fnptr)();
...
```

- 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 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 "Owned"
- Buffer overrun vulnerabilities and malicious code injection attacks are primary/favorite method used by worm writers

Buffer Exploit History

- How likely are the conditions required to exploit buffer overruns? Actually fairly rare…
- But, first Internet worm (Morris worm) spread using several attacks
  - One used buffer overrun to overwrite authenticated flag in `in.fingerd`
- Technique now exploited by many network attacks
  - Anytime input comes from network request and is not checked for size
  - Code executes with same privileges as running pgm
- How to prevent?
  - Don't code this way! (ok, wishful thinking)
  - New mode bits in Intel, AMD, and Sun processors
    - » Put in page table; says "don't execute code in this page"
- 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) reg points to next machine instruction to execute
- 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 (use previous techniques)
- 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

- Techniques for when:
  - Malicious code gets stored at unknown location
  - 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

Buffer Overrun Summary

- Historically, many security researchers have underestimated opportunities for obscure and sophisticated attacks
  - Very easy mistake to make…
- Lesson learned:
  - If your program has a buffer overrun bug, assume that the bug is exploitable and an attacker can take control of program
- Buffer overruns are bad stuff – you don't want them in your programs!
  - Some automated solutions – dynamic memory layout

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…

BREAK

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 bugs, like buffer overruns, are nasty business

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

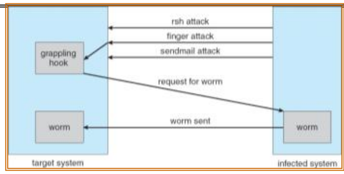
```
size_t len = read_int_from_network();
char *buf;
buf = malloc(len+5);
read(fd, buf, len);
...
```

- 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

Many More Vulnerabilities…

- We've only scratched the surface!
  - These are the most prevalent examples
- If it makes you just a bit more cautious about how you write code, good!
- Many real-world examples…

The Morris Internet Worm



- Internet worm (Self-reproducing)
  - Author Robert Morris, a first-year Cornell grad student
  - Launched close of Workday on November 2, 1988
  - Within a few hours of release, it consumed resources to the point of bringing down infected machines
- Techniques
  - Exploited UNIX networking features (remote access)
  - Bugs in *finger* (buffer overflow) and *sendmail* programs (debug mode allowed remote login)
  - Dictionary lookup-based password cracking
  - Grappling hook program uploaded main worm program

Some other Attacks

- Trojan Horse Example: Fake Login
  - Construct a program that looks like normal login program
  - Gives "login:" and "password:" prompts
    - » You type information, it sends password to someone, then either logs you in or says "Permission Denied" and exits
  - In Windows, the "ctrl-alt-delete" sequence is supposed to be really hard to change, so you "know" that you are getting official login program
- Is SONY XCP a Trojan horse?
- Salami attack: Slicing things a little at a time
  - Steal or corrupt something a little bit at a time
  - E.g.: What happens to partial pennies from bank interest?
    - » Bank keeps them! Hacker re-programmed system so that partial pennies would go into his account.
    - » Doesn't seem like much, but if you are large bank can be millions of dollars
- Eavesdropping attack
  - Tap into network and see everything typed
  - Catch passwords, etc
  - Lesson: never use unencrypted communication!

Ken Thompson's self-replicating program

- Bury Trojan horse in binaries, so no evidence in source
  - Replicates itself to every UNIX system in the world and even to new UNIX's on new platforms. No visible sign.
  - Gave Ken Thompson ability to log into any UNIX system
- Two steps: Make it possible (easy); Hide it (tricky)
- Step 1: Modify login.c

```
A: if (name == "ken")
       don't check password
       log in as root
```

  - Easy to do but pretty blatant! Anyone looking will see.
- Step 2: Modify C compiler
  - Instead of putting code in login.c, put in compiler:

```
B: if see trigger1
       insert A into input stream
```

  - Whenever compiler sees trigger1 (say /*gobbledygook*/), puts A into input stream of compiler
  - Now, don't need A in login.c, just need trigger1

Self Replicating Program Continued

- Step 3: Modify compiler source code:

```
C: if see trigger2
       insert B+C into input stream
```

  - Now compile this new C compiler to produce binary
- Step 4: Self-replicating code!
  - Simply remove statement C in compiler source code and place "trigger2" into source instead
    - » As long as existing C compiler is used to recompile the C compiler, the code will stay into the C compiler and will compile back door into login.c
    - » But no one can see this from source code!
- When porting to new machine/architecture, use existing C compiler to generate cross-compiler
  - Code will migrate to new architecture!
- Lesson: never underestimate the cleverness of computer hackers for hiding things!

Conclusion

- Attackers will exploit any and all flaws!
  - Buffer overruns, format string usage errors, implicit casting, TOCTTOU, …
- Buffer overrun attack: exploit bug to execute code
- Format string attack: exploit bug in printf, fprintf, sprintf
- Implicit casting attack: exploit missing cast statement
- Self-modifying code can be used for nearly undetectable attacks