CS 194-1 (CS 161) Computer Security

Lecture 13

Software security; Common implementation flaws; Principles

October 16, 2006
Prof. Anthony D. Joseph
http://cs161.org/

Goals for Today

- Next 3 lectures are about software security
  - Can have perfect design, specification, algos, 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
- Principles for building secure systems
  - Trusted computing base (TCB)
  - Three Cryptographic principles
  - 13 other security principles

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

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)();
...
```
- 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` (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) 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!

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

Administrivia

- Project #1 code and docs due today
  - Don't use up all your slip days
- Homework #2 due 10/27 (posted today)

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

Non-Language-Specific Vulnerabilities

```
int openfile(char *path) {
    struct stat s;
    if (stat(path, &s) < 0)
        return -1;
    if (!S_ISRREG(s.st_mode)) {
        error("only regular files allowed!");
        return -1;
    }
    return open(path, O_RDONLY);
}
```

- Code to open only regular files
  - Not symlink, directory, nor special device
- On Unix, uses `stat()` call to extract file's meta-data
- Then, uses `open()` call to open the file

The Flaw?

- Code assumes FS is unchanged between `stat()` and `open()` calls – Never assume anything…
- An attacker could change file referred to by `path` in between `stat()` and `open()`
  - From regular file to another kind
  - Bypasses the check in the code!
  - If check was a security check, attacker can subvert system security
- Time-Of-Check To Time-Of-Use (TOCTTOU) vulnerability
  - Meaning of `path` changed from time it is checked (`stat()`) and time it is used (`open()`)

TOCTTOU Vulnerability

- In Unix, often occurs with filesystem calls because system calls are not atomic
- But, TOCTTOU vulnerabilities can arise anywhere there is mutable state shared between two or more entities
  - Example: multi-threaded Java servlets and applications are at risk for TOCTTOU

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!
- In future lectures, we'll discuss how to prevent (or reduce the likelihood) of these kinds of flaws, and to improve the odds of surviving any flaws that do creep in

Principles of Secure Software

- Let's explore some principles for building secure systems
  - Trusted Computing Base & several principles
- These principles are neither necessary nor sufficient to ensure a secure system design, but they are often very helpful
- Goal is to explore what you can do at design time to improve security
  - How to choose an architecture that helps reduce likelihood of system flaws (or increases survival rate)
- Next lecture: what to do at implementation time

The Trusted Computing Base (TCB)

- *Trusted Component*:
  - A system part we rely upon to operate correctly for system security
  - (A part that can violate our security goals)
- *Trustworthy components*:
  - System parts that we're justified in trusting (assume correct operation)
- In Unix, the super-user (root) is trusted
  - Hopefully they are also trustworthy…
- *Trusted Computing Base*:
  - System portion(s) that must operate correctly for system security goals to be assured

TCB Definition

- We rely on every component in TCB working correctly
- Anything outside isn't relied upon
  - Can't defeat system's security goals even if it misbehaves or is malicious
- TCB definition:
  - Must be large enough so that nothing outside the TCB can violate security

TCB Example

- Security goal: only authorized users allowed to log into my system using SSH
- What is the TCB?
  - TCB includes SSH daemon (it makes authentication and authorization decisions)
  - If sshd has a bug (buf overrun) or was maliciously reprogrammed (backdoor), it can violate security goal by allowing unauthorized access
  - TCB also includes OS (can tamper with sshd's operation and address space)
  - TCB also includes CPU (rely on it to execute sshd correctly)

TCB Example (continued)

- What about a web browser application on the same machine? Is it in the TCB?
- Hopefully not!
  - OS is supposed to protect sshd from other unprivileged applications
- Another ex.: network perimeter firewall
  - Enforces security goal that only authorized connections are permitted into internal net
- In this example, the firewall is the TCB for this security goal

TCB as Reference Monitor

- There's always a mechanism responsible for enforcing an access control policy
  - Recall firewall lecture: this mechanism is a *Reference Monitor*
- Reference monitor is the TCB for security goal of ensuring access control policy
  - A reference monitor is just a TCB specialized for access control
- Recall: three guiding principles for reference monitor
  - *Unbypassable*, *Tamper-resistant*, and *Verifiable*

TCB as a Reference Monitor

- *Unbypassable*:
  - No way to bypass the TCB and breach security
- *Tamper-resistant*:
  - TCB protected from tampering by anyone else
    - » Other system parts (outside TCB) shouldn't be able to modify TCB's code or state
  - The integrity of the TCB must be maintained
- *Verifiable*:
  - Should be possible to verify TCB correctness
    - » Means TCB should be as simple as possible (beyond the state of the art to verify complex subsystems)

BREAK

Why Keep the TCB Simple and Small?

- Good practice!
  - Less code you write, less chances to make mistakes or introduces implementation flaws
- Industry standard error rates are 1–5 defects per thousand Lines of Code (kLoC)
  - TCB containing 1 kLoC might have 1–5 defects
  - 100 kLoC TCB might have 100–500 defects!
  - (Windows XP is about 40,000 kLoC of TCB!!)
    - » Almost all of which is the TCB
- Lesson:
  - Shed code and design system so as much code can be moved outside the TCB as possible

TCBs: What are They Good for?

- Is the TCB concept just an esoteric idea?
  - No, it is a very powerful and pragmatic idea
  - TCB allows primitive, yet effective modularity
- Separates system into two parts: security-critical (TCB) and everything else
- Building secure and correct systems is hard!
  - More pieces makes security assurance harder
  - Only parts in TCB must be correct for system security -> focus efforts where they matter
  - Making TCB small gives us better odds of ending up with a secure system

Ex: Email Retention for National Archives

- National Archives chartered with saving a copy of every email ever sent by government officials
  - Security Goal: Ensure that saved records cannot be deleted or destroyed
  - Someone being investigated might try to destroy embarrassing or incriminating archived documents
- We need an "append-only" document storage system
  - How can we do it?

A Possible Approach

- Augment email program on every desktop computer to save a copy of all emails to a special directory on that computer
  - What's the TCB for this approach?
    - » TCB includes *every* copy of email application on *every* government machine
    - » Also OS, all privileged SW, and sys admins
- That's an awfully large TCB!
  - Unlikely that everything in TCB works correctly
- Also, any sys admin can delete files from the special directory after the fact
- We'd better find a better solution!!

Another Approach

- Set up a high-speed networked printer
  - An email is "collected" when it is printed
  - Printer room is locked to prevent tampering
  - What's the TCB in this system?
    - » TCB includes room's physical security
    - » Also includes the printer
- Suppose we add a ratchet to paper spool so that it can only rotate forward
  - Don't need to trust the rest of the printer
- Wow!
  - TCB is only this ratchet, and room's physical security, nothing else!
- But, our approach uses a *lot* of paper!

An All-Electronic Approach

- Networked PC running special server SW
  - Accepts email msgs and adds them its local FS
  - FS carefully implemented to provide write-once semantics: once a file is created, it can never be overwritten or deleted
  - Packet filter blocks all non-email connections
- What's in the TCB now?
  - Server PC/app/OS/FS, privileged apps on PC, packet FW, PC's sys admins, room's physical security, …
- TCB is bigger than with a printer, but smaller than all machines approach's TCB
- I think you've earned your consulting fee

TCB Principles Summary

- Know what is in the TCB
  - Design your system so that the TCB is clearly identifiable
- Try to make the TCB as unbypassable, tamper-resistant, and verifiable as possible
- Keep It Simple, Stupid (KISS)
  - The simpler the TCB, the greater the chances you can get it right
- Decompose for security
  - Choose a system decomposition/modularization based on simple/clear TCB
    - » Not just functionality or performance grounds

Three Cryptographic Principles

- Three principles widely accepted in crypto community that seem useful in computer security
  - Conservative Design
  - Kerkhoff's Principle
  - Proactively Study Attacks

1. *Conservative Design*

- *Systems should be evaluated according to worst plausible security failure, under assumptions favorable to attacker*
  - Doug Gwyn came up with this formulation
- If you find such circumstance where the system can be rendered insecure, then you should seek a more secure system

2. *Kerkhoff's Principle*

- Cryptosystems should remain secure even when the attacker knows all internal details of the system
- The key should be the only thing that must be kept secret
- If your secrets are leaked, it is a lot easier to change the key than to change the algorithm

3. *Proactively Study Attacks*

- We must devote considerable effort to trying to break our own systems
  - How we can gain confidence in their security
- Other reasons:
  - In security game, attacker gets last move
  - Very costly if a security hole is discovered after wide system deployment
- Pays to try to identify attacks before bad guys find them
  - Gives us lead time to close security holes before they are exploited in the wild

Principles for Secure Systems

- General principles for secure system design
  - Many drawn from a classic 1970s paper by Saltzer and Schroeder
- 1. *Security is Economics*
  - No system is 100% secure against all attacks
    - » Only need to resist a certain level of attack
    - » No point buying a $10K firewall to protect $1K worth of trade secrets
  - Often helpful to quantify level of effort an attacker would expend to break the system.
  - Adi Shamir once wrote, "There are no secure systems, only degrees of insecurity"
    - » A lot of the science of computer security comes in measuring the degree of insecurity

Economics Analogy

- Safes come with a security level rating
- Consumer-grade safe:
  - Rated to resist attack for up to 5 minutes by anyone without tools
- High-end safe might be rated TL-30
  - Secure against burglar with safecracking tools and less than 30 minutes access
  - We can hire security guards with a less than 30 minute response time to any intrusion

Corollary of This Principle

- Focus your energy on securing weakest links
  - Security is like a chain: it is only as secure as the weakest link
  - Attackers follow the path of least resistance, and will attack system at its weakest point
- No point in putting an expensive high-end deadbolt on a screen door
  - Attacker isn't going to bother trying to pick the lock when he can just rip out the screen and step through!

2. *Least Privilege*

- Minimize how much privilege you give each program and system component
  - Only give a program the minimum access privileges it legitimately needs to do its job
- Least privilege is a powerful approach
  - Doesn't reduce failure probability, but can reduce expected cost of failures
- Less privilege a program has, less harm it can do if it goes awry or runs amok
  - Computer-age version of shipbuilder's notion of "watertight compartments":
    - » Even if one compartment is breached, we minimize damage to rest of system's integrity

Principle of Least Privilege Examples

- Can help reduce damage caused by buffer overruns or other program vulnerabilities
  - Intruder gains all the program's privileges
  - Fewer privileges a program has, less harm done if it is compromised
- How is Unix in terms of least privilege?
  - Answer: Pretty lousy!
  - Programs gets all privileges of invoking users
  - I edit a file and editor receives all my user account's privileges (read, modify, delete)
- Strictly speaking editor only needs access to file being edited to get job done

Principle of Least Privilege Examples

- How is Windows in terms of least privilege?
  - Answer: Just as lousy!
  - Arguably worse, as many users run as Administrator and many Windows programs require Administrator access to run
- Every program receives total power over the whole computer!!
- Microsoft's security team recognizes this risk
  - Advice: Use limited privilege account and "Run As…"