Software security; Common implementation flaws; Principles

Dawn Song

dawnsong@cs.berkeley.edu

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**

Preventing overflow attacks

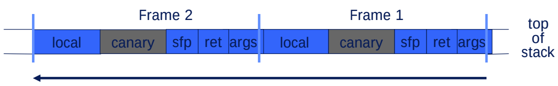
- **Main problem:**
  - **strcpy(), strcat(), sprintf() have no range checking.**
  - **"Safe" versions strncpy(), strncat() are misleading**
    - **strncpy() may leave buffer unterminated.**
    - **strncpy(), strncat() encourage off by 1 bugs.**
- **Defenses:**
  - **Type safe languages (Java, ML). Legacy code?**
  - **Mark stack as non-execute. Random stack location.**
  - **Static source code analysis.**
  - **Run time checking: StackGuard, Libsafe, SafeC, (Purify).**
  - **Many more …**

Marking stack as non-execute

- **Basic stack exploit can be prevented by marking stack segment as non-executable.**
  - **NX-bit on AMD Athlon 64, XD-bit on Intel P4 "Prescott".**
    - **NX bit in every Page Table Entry (PTE)**
  - **Support in SP2. Code patches exist for Linux, Solaris.**
- **Limitations:**
  - **Does not defend against `return-to-libc' exploit.**
    - **Overflow sets ret-addr to address of libc function.**
  - **Does not block more general overflow exploits:**
    - **Overflow on heap: overflow buffer next to func pointer.**
  - **Some apps need executable stack (e.g. LISP interpreters).**
  ```

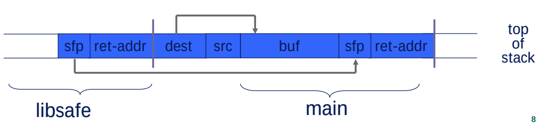
Run time checking: StackGuard

- **Many many run-time checking techniques …**
  - **Here, only discuss methods relevant to overflow protection.**
- **Solutions 1: StackGuard (WireX)**
  - **Run time tests for stack integrity.**
  - **Embed "canaries" in stack frames and verify their integrity prior to function return.**



Run time checking: Libsafe

- **Solutions 2: Libsafe (Avaya Labs)**
  - **Dynamically loaded library.**
  - **Intercepts calls to strcpy (dest, src)**
    - **Validates sufficient space in current stack frame:**
      |frame-pointer – dest| > strlen(src)
    - **If so, does strcpy.**
      Otherwise, terminates application.



More methods …

- **StackShield**
  - **At function prologue, copy return address RET and SFP to "safe" location (beginning of data segment)**
  - **Upon return, check that RET and SFP is equal to copy.**
  - **Implemented as assembler file processor (GCC)**
- **Randomization:**
  - **PaX ASLR: Randomize location of libc.**
    - **Attacker cannot jump directly to exec function.**
  - **Instruction Set Randomization (ISR)**
    - **Attacker cannot execute its own code.**

Non-Language-Specific Vulnerabilities

- ```
  int openfile(char *path) {
      struct stat s;
      if (stat(path, &s) < 0)
          return -1;
      if (!S_ISREG(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

Administrivia

- Office hour this week moved to Thu 4pm.
- From next week on, office hour moved to Wed 5pm.

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

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

TCB Principles Summary

- Know what is in the TCB
  - Design your system so that the TCB is clearly identifiable
- 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***
- **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…"