CS 194-1 (CS 161)
Computer Security

Lecture 15

Software security (defensive programming)

October 23, 2006
Prof. Anthony D. Joseph
http://cs161.org/

Review: Defensive Programming

- Like defensive driving, but for code:
  - Avoid depending on others, so that if they do something unexpected, you won't crash – survive unexpected behavior
- Software engineering focuses on functionality:
  - Given correct inputs, code produces useful/correct outputs
- Security cares about what happens when program is given invalid or unexpected inputs:
  - Shouldn't crash, cause undesirable side-effects, or produce dangerous outputs for bad inputs
- Defensive programming
  - Apply idea at every interface or security perimeter
    - » So each module remains robust even if all others misbehave
- General strategy
  - Assume attacker controls module's inputs, make sure nothing terrible happens

Goals for Today

- Defensive programming techniques to avoid security holes when writing code
  - Several good practices
  - Lots of overlap with software engineering and general software quality, but security places heavier demands
- Isolation
  - Software techniques for keeping suspect programs from affecting other apps or the OS
    - » Separate program modules
    - » System call interposition
    - » Virtual Machines

Some General Advice

- *1. Check for error conditions*
  - Check rv's, error paths, exception handling
  - Always safe to use ***fail-stop*** behavior
- 2. *Validate All Inputs*
  - Sanity-check all inputs from rest of program
  - Treat external inputs (could be from adversary) with particular caution
  - Check that the input looks reasonable
  - Be conservative
    - » Better to limit inputs to expected values (might cause some loss of functionality) than to liberally allow all (might permit unexpected security holes)

What's Wrong with this Code?

```
char *username = getenv("USER");
char *buf = malloc(strlen(username)+6);
sprintf(buf, "mail %s", username);
FILE *f = popen(buf, "r");
fprintf(f, "Hi.\n");
fclose(f);
```

- Answer: If attacker controls `USER` environment variable, then could arrange for its value to be something like "`adj; /bin/rm -rf $HOME`"
  - `popen()` passes its input to shell for execution, and shell will execute command "`mail adj`" followed by "`/bin/rm -rf $HOME`"
- Solution: validate that username looks reasonable
  - If attacker can control other env vars (e.g., PATH), then could cause wrong `mail` command to be invoked ? have to validate whole environment!

Advice: 3. *Whitelist, Don't Blacklist*

- Common mistake:
  - When validating input from an untrusted source, trying to enumerate bad inputs and block them
  - Don't do that! Why?
  - Known as *blacklisting* (analogous to default-allow policy)
  - Can overlook some patterns of dangerous inputs
- Instead, use *whitelist* of known-good types of inputs, and block anything else
  - Default-deny policy (much safer)

Whitelisting Example

- Check a username using a regular expression:
  - [a-z][a-z0-9]*
  -
```
char *validate_username(char *u) {
        char *p;
        if (!u || *u < 'a' || *u > 'z')
            die();
        for (p=u+1; *p; p++)
            if ((*p < '0' || *p > '9') &&
                (*p < 'a' || *p > 'z'))
                die();
        return u;
    }
```
- Use with appropriate error-checking before using a user-supplied username

More Advice

- 4. *Don't crash or enter infinite loops, Don't corrupt memory*
  - Regardless of received inputs – NO abnormal termination, infinite loops, internal state corruption, control flow hijacks
  - Explicitly validate all inputs and avoid memory leaks
  - Defend against DoS attacks:
    - » Attacker supplies inputs that lead to worst-case performance (hashtable with O(1) expected, but O(n) worst case lookup)

More Advice

- 5. *Beware of integer overflow*
  - Integer overflow often violates programmer's mental model and leads to unexpected (undesired) behavior
- 6. *Check exception-safety of the code*
  - Explicitly (programmer) thrown and implicitly (platform) thrown exceptions
  - Verify that your code doesn't throw runtime exceptions (null ptr deref, div 0,…)
  - Less restrictively, check that all such exceptions are handled and will propagate across module boundaries

Famous Example: Ariane 5

- Ariane 4 flight control sw written in Ada
  - Same software reused for more powerful Ariane 5
- Ariane 5 blew up shortly after first launch
  - Cause: uncaught integer overflow exception caused software to terminate abruptly…
- 16-bit reg: flight trajectory's horizontal velocity
  - Ariane 4 – verified range of physically possible flight trajectories could not overflow variable, so no need for exception handler…
- Ariane 5's rocket engine was more powerful, causing larger horizontal velocity to be stored into register triggering overflow…
  - Losses of around $500 million

Multiple Clients

- Module *M* supports multiple *clients*
  - Must defend itself against malicious clients
  - Isolate malicious clients from each other
- *M* may in turn invoke other utility modules
  - Same requirements apply…
- Exception: *M* computes a pure function (no internal state or I/O)
  - One client can't disrupt another or corrupt *M*'s state
  - Thus, functional programming simplifies defensive programming task

Pre-Condition Choices

- Use precondition and leave it to caller to ensure it is true
- Or, explicitly check for ourselves that condition holds (and abort if it doesn't)
- How should we decide between these two strategies (for externally invoked fcns)?
  - Use documented preconditions to express intended contract
  - Use explicit checking for anything that could corrupt our internal state, cause us to crash, or disrupt other clients
- Don't need to worry as much about internal helper functions

Security Choices for Languages

- *Pick tools that you know well*
  - Many security bugs caused by insufficient familiarity with obscure corner cases in language, libraries, or programming env.
  - Read and understand formal language spec
- *Pick a prog. platform designed for safety*
  - >50% of security holes in C code related to absence of bounds-checking in C
  - Choose strong type checking and automatic: array/ptr bounds-checking, memory mgmt, and uninitialized variables
  - Assembly language is a poor choice (so easy to make devastating mistakes)
    - » Use only when absolutely necessary (like C and C++?)
  - Type-safe languages (Java, C#, Ada, ML) have many security advantages

Dealing with Insecure Languages

- Can't always choose the language based on security…
  - Other considerations may dominate
  - Or, may be forced to maintain legacy code
- Need to be extra careful
  - Avoid obscure corners of language
  - If no automatic bounds-checking, consider inserting manual bounds-checks
  - Consider writing code so you can prove that out-of-bounds accesses are impossible

C-Specific Advice

- *Avoid buffer overruns*
  - Prove no mem access (array, ptr deref, structure) can overflow bounds
  - Make all preconditions, loop invariants, and object invariants for this explicit in code
- *Avoid undefined behavior*
  - Used frequently in the C standard
    - » Many primitives have implicit preconditions
    - » `a[i]` is undefined if `i` is out of bounds
  - Can be used to hijack program control
- *Get familiar with the C standard*
  - Textbooks, man pages, and informal guides occasionally get things wrong

Administrivia

- Journal due by midnight tonight
- Homework #2 due Friday October 27th
- Midterm #2 is November 6th in class
- Project #2 will be posted later this week
- No office hours for Prof. Joseph *next* week: 10/30 and 11/2

Security is an Ongoing Process

- Integrate into all phases of system development lifecycle
  - Requirements analysis, Design
  - Implementation, Testing
  - Quality assurance, Bug fixing
  - Maintenance
- Steps:
  - Test code thoroughly
  - Use code reviews to cross-check each other
  - Evaluate the cause of all bugs found

Pre-Deployment: Test Code Thoroughly

- Testing can help eliminate (security) bugs
- Test corner cases: long/unusual/8-bit strings
  - Strings containing format specifiers (%s) and newlines, and other unexpected values
- Analyze manuals and documentation
  - If manual says input should be of a particular form, construct counter test cases
- Use unit tests to stress boundary conditions
  - 0, 1, -1, 2^{31}-1, -2^{31} are fun to try
  - Try inputs with unusual pointer aliasing or pointing to overlapping memory regions
- Automate tests and run them nightly

More Process

- *Use code reviews to cross-check each other*
  - We're all fallible – use another perspective to find defects we've missed
  - Easy to make implicit assumptions without realizing it – original programmer will make same erroneous assumption when reviewing their own code
- Code reviews keep us honest and motivated
  - Don't want to be embarrassed in front of peers

More Process

- *Evaluate the cause of all bugs found*
  - What to do when you find a security bug?
  - Fix it first, then follow several steps
- 1. Generate regression test that triggers the security hole and add to test suite
- 2. Check whether there are similar bugs elsewhere in the codebase
  - Document pitfall or coding pattern that causes this bug, so others can learn from it
- 3. Consider how to prevent similar bugs from being introduced in the future

Security Bugs

- Have to fix the root cause that creates conditions for security bugs to be introduced
- Periodically investigate security bug root causes
  - Are there adequate resources for security?
  - Is security adequately prioritized?
  - Was the design well-chosen?
  - Are you using the right tools for the job?
  - Are deadlines too tight?
  - Does it indicate some weakness in the process?
  - Do engineers need more training on security?
  - Should you be doing more testing, more code reviews, something else?

Isolation

- An isolated program can't affect other programs on the system
  - Isolation is related to topics we've seen before (access control)
    - » Access control enforces some security policy (a means to an end), whereas isolation is a security goal (the end itself)
- Related to VM and memory protection
  - Virtual memory only isolates memory between processes – doesn't prevent other kinds of influence (opening an IPC pipe from one process to another)
- Want to isolate against *all* influences, so memory protection alone is not enough

Isolation Examples

- You find a cool program that draws dancing hamsters on the screen
  - You want to download and try it but don't know if you can trust the developer
- Want to display an emailed MS Word file
  - Don't want my PC infected with a macro virus
- These are *sandboxing* problems
  - Run software in an isolated env. – can't harm rest of the machine even if it is malicious
- Designing a complicated software application
  - Following principle of least privilege, decompose it into multiple *isolated* pieces

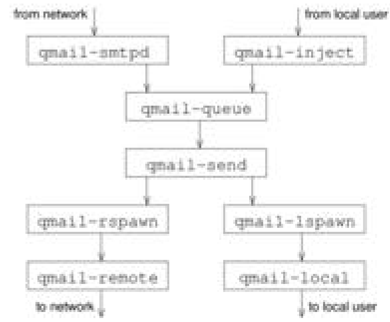
Decomposing Software for Security

- Replacing a popular mail application: `sendmail` (written by EECS staffer Eric Allman)
  - Large (100K LoC), monolithic, runs as root, and plagued by security problems
- `qmail` secure mailer (2nd most popular mailer)
  - Written by Dan Bernstein
  - He offered a $500 prize in 1997 to first person to find a serious security hole
    - » The $500 still remains unclaimed…
- Let's see why…

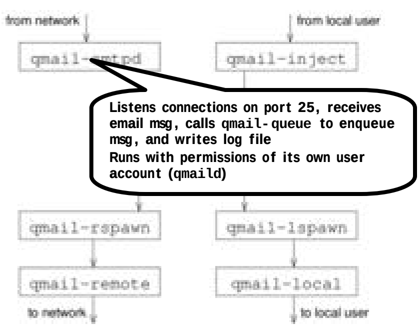
What Does a Mail Daemon Do?

- **Receives incoming email via port 25**
  - **Has to listen for connections to port 25**
- **Receives email submissions from other programs on this host**
  - **Has to be prepared to be invoked by other programs who want to submit mail for transmission elsewhere**
- **When it receives an email message**
  - **Queues the message, determines where to route it (locally delivery to a user or forwarded to another host)**

Qmail Internals

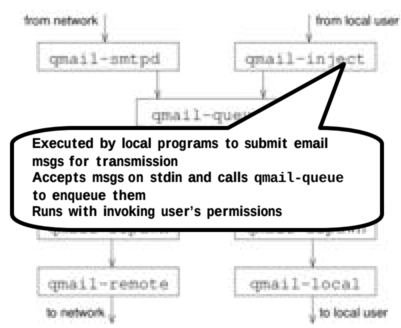


Qmail Internals



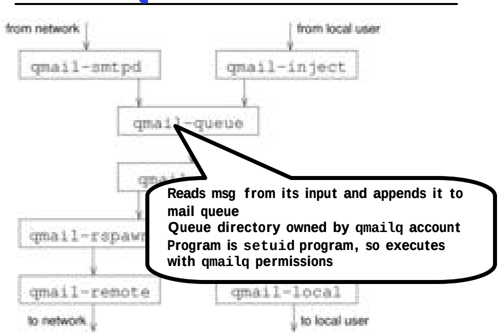
Listens connections on port 25, receives email msg, calls `qmail-queue` to enqueue msg, and writes log file
Runs with permissions of its own user account (`qmaild`)

Qmail Internals



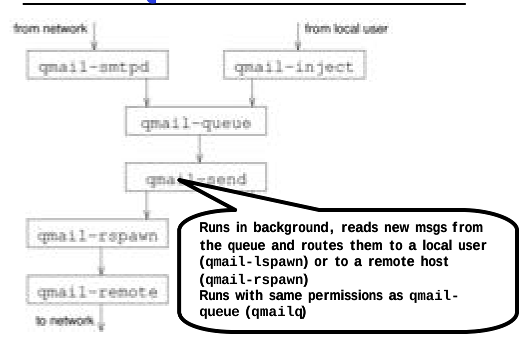
Executed by local programs to submit email msgs for transmission
Accepts msgs on stdin and calls `qmail-queue` to enqueue them
Runs with invoking user's permissions

Qmail Internals



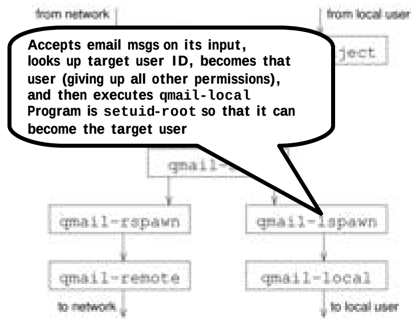
Reads msg from its input and appends it to mail queue
Queue directory owned by `qmailq` account
Program is `setuid` program, so executes with `qmailq` permissions

Qmail Internals

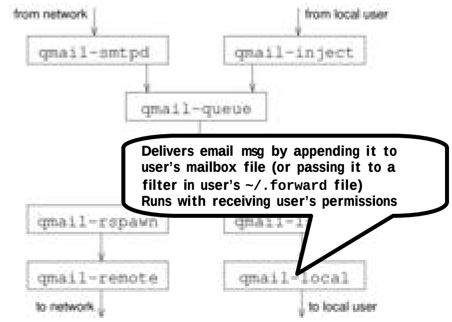


Runs in background, reads new msgs from the queue and routes them to a local user (`qmail-lspawn`) or to a remote host (`qmail-rspawn`)
Runs with same permissions as `qmail-queue` (`qmailq`)

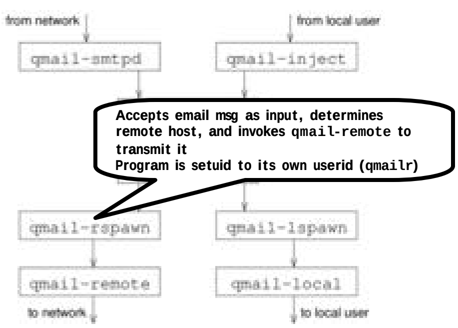
Qmail Internals



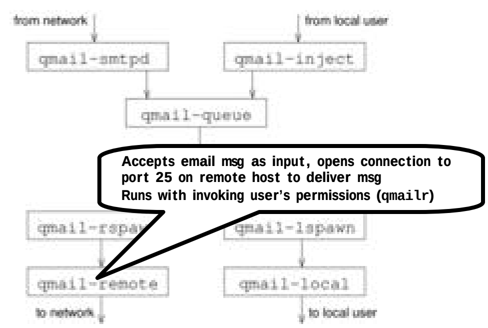
Qmail Internals



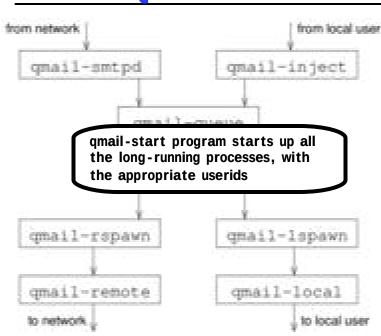
Qmail Internals



Qmail Internals



Qmail Internals



Why Use So Many Programs?

- Minimizes amount of code running as root!
  - Only `qmail-start` and `qmail-lspawn`
    - » Principle of least privilege (vs `sendmail`)
- Reduces amount of security-critical code
  - Only local users can invoke `qmail-inject`
- Separates logically different functions into mutually distrusting programs
  - No program trusts data from the others
  - Security holes do not give root access*
  - OS prevents tampering with executable
- Each program is extremely simple
  - `qmail-send` is 1600 loCC (others < 800)

Isolation and Controlled Sharing

- Pure isolation is usually too strict
- Isolation is analogous to the *deny-all* starting point of a default-deny policy
- Controlled sharing allows limited escape routes out of the sandbox
  - Useful interaction without exposure to attack?
  - `qmail`: controlled sharing between qmail programs through explicit communication channels

BREAK

Example Decomposition for Security

- Design web service to convert files from one format to another (MP3 to OGG)
  - Accepts port 80 connections, translates file into new format, and sends back results
- Break into two pieces:
  - *Master process* receives file, invokes slave process with data, and returns slave's output
  - Slave process takes in byte array, transforms MP3 into OGG format, and outputs OGG data
    - » Deterministic function of its input – no permissions needed at all – can be sandboxed
    - » Buggy MP3-to-OGG code can only return incorrect OGG files – can't harm our machine

Web Browser Example

- Web browser needs to decompress a received file
  - Decompression program is complex and you're not 100% sure you trust it
- How do you structure your application to minimize trust in the decompressor?
  - 2002: discovered that `zlib` libraries from 2/98 – 3/02 were vulnerable to code injection exploit
- Unix and Windows don't make it very easy to get the necessary kind of isolation
  - Many apps where sandboxing and isolation would be very useful

Access Control

- Secure sandbox must be inescapable
  - How to enforce isolation guarantees?
- Easiest solution – create new user account
  - Install and run sandboxed program in account
  - Uses OS's access control mechanisms
- Problem: OS is focused on protecting file access
  - Program can create connections or run servers*
    - » "default allow" policy for network connections…
  - Many files are world-readable ("default allow")
  - Pgm can attack other machines, send spam,…
    - » Machines behind my FW are vulnerable!
    - » Might steal /etc/passwd file and email it

Qmail's Strategy

- `qmail` uses OS to build its sandbox
  - Its isolation guarantees are actually slightly weaker than mentioned before…
- Intruder who gains control of a qmail program isn't entirely isolated
  - Can attack other hosts on same intranet
  - A limitation of `qmail`'s isolation strategy
- Difficult to do better while remaining portable

System Call Interposition

- *Interposition* on the system call interface
  - Place a sandbox enforcer between sandboxed application and OS
- Mediates all system call requests:
  - App's syscalls are re-directed to enforcer
  - Enforcer approves or denies syscall request based on the arguments
    - » Extends OS's access control policy without modifying the OS itself
- Example Policy – MP3-to-OGG
  - Pure computation, nothing else
  - Deny all system calls except receiving input (fd0) and producing output (fd1)

Another Example Policy

- Adobe Acrobat PDF viewer on Linux:
  - Allow `connect()` to port 6000 on localhost to open X windows
  - Allow `open()` or manipulate of files under `~/.acrobat` for its preferences
  - Allow any calls to `read()` or `write()` since they're only useful on open file descriptors
- But, many other items needed (file to view?)
  - Loads dynamic libraries (`open()` and `mmap()`)
  - Uses `/usr/lib/locale` to determine language
  - Uses signals and threads (need to apply syscall interposition to spawned processes…)
- Sandboxing policy is surprisingly complex!!

Subtle Interposition Pitfalls

- Very easy risk of TOCTTOU vulnerabilities
- Examples:
  - `open()` syscall's first arg is ptr to filename and malicious program could change it after enforcer's check but before OS executes `open()`
    - » Solution: OS copies filename into kernel memory then to enforcer
  - Calls like `open("foo")` rely on current directory and, in a multi-threaded / processor environment, program could change working directory
    - » Solution: accurately maintain *shadow state*

Shadow State

- OS maintains state for each running process (e.g., current working dir)
- For security, enforcer maintains its own copy of state
  - If the copies get out of sync, enforcer may allow prohibited system calls to proceed
  - Hard to interpose a reference monitor on an interface where a call's meaning depends on state not exposed in call's args
- Alternative: *Virtualize* and emulate OS
  - Sandboxed application thinks it is running on a real OS, but actually running on enforcer's emulated OS

System Call Interposition Summary

- Lots of research into syscall interposition
  - I've omitted many interesting details
- For more information, read about tools such as Systrace, Janus, and Ostia
- Question for thought:
  - How could you use system call interposition to make `qmail` more bullet-proof?

Physical Isolation

- Run sandboxed app on a physically isolated machine
  - When done, reboot and reformat machine and reuse it for another sandboxed app
- A good way to achieve isolation
  - Can be pretty confident that nothing can escape the sandbox (especially if machine doesn't have any network interfaces)
  - But, very expensive!
- Approach used in military domains
  - Need access to Internet and SIPRNET
  - Give each analyst two separate machines
  - Could we use virtual machines instead?

Virtual Machines

- If real machines are too expensive, use a *virtual machine* instead
- A virtual machine is a software app that emulates a physically separate machine
  - Examples: VMWare, Virtual PC, QEmu, Bochs
- How does a virtual machine work?
- Consider an x86 emulator program:
  - Takes in an x86 binary and interprets the instructions *entirely in software*
  - Maintains (in SW) the emulated state of an x86 CPU and emulates behavior of physical devices

Virtual Machines Internals

- x86 program running on VMWare on Linux
  - VMWare creates 100MB file on Linux FS to store emulated 100MB hard drive
  - Translates program's reads/writes into Linux read/write syscalls to 100MB file
    - » Same for writes to screen
  - Big slowdown, but tricks eliminate most overhead
- One physical machine can simulate dozens
  - » Benefits of physical isolation without HW
  - » "Bad" programs unable to change long-term state
- Virtualization is a powerful technique
  - Like VM and syscall interposition, virtual machines work by virtualizing the HW interface

Interpreted Code

- Virtual machines illustrate how interpreted languages can be used for sandboxing
  - Interpreter is a loop repeatedly decoding and executing a sequence of instructions
- Simplest example: combinatorial circuits
  - Can implement any stateless deterministic computation as a combinatorial circuit
  - Given boolean function $f{:}\{0,1\}^n ? \{0,1\}$, find a combinatorial circuit that computes f
    - » f is deterministic and side-effect-free
    - » A network of AND/OR/NOT gates with n inputs, 1 output, and no cycles or memory
  - Easy to sandbox – evaluate circuit in sw

Interpreted Code Example

- You write an extensible spam-filtering app
  - Your friend Sam creates a program that takes an email as input and classifies it either as "spam" or "not spam"
  - If you don't trust Sam, you can't run his program – might be a Trojan horse!
- Express Sam's program as boolean function *f*
  - Takes an email (a bit-string) as input and produces a boolean output
- Solution: Sam expresses his program as a combinatorial circuit
  - Malicious filter can't leak email contents
  - Worst case: causes wrong filtering decisions

Boolean Circuits Interpreter

- Use simple NAND gates to express arbitrary combinatorial logic
- Emulate circuit using very simple CPU:
  - Store each value in circuit in a register
  - Each instruction reads inputs from two specified registers, computes their NAND, and stores result to third register
    - » `NAND r1037, r27, r45` computes NAND of bit in register r27 and bit in r45, storing result in r1037
  - Interpreter only takes a few lines of code
- But, circuits aren't very friendly/flexible
  - Apply same principles to an interpreter

Secure Interpreter

- Design language so it is impossible to express operations that would violate sandboxing policy
  - Ex: no way to do I/O or R/W outside program's address space
- Example: Berkeley Packet Filter
  - Interpreted language for expressing packet filters that can be downloaded into the kernel
  - Language prevents writers from expressing harmful programs
  - Ex: can't write non-terminating loops because no backward jumps are allowed

Summary

- Defensive programming won't prevent bugs or security problems
  - But, it can help contain the damage
- Testing the uncommon is critical
- Several programming techniques for avoiding or handling problems
- Use isolation techniques for untrusted code
  - Module decomposition
  - System call interposition
  - Virtual machines and secure interpreters