Shifting the Odds

Writing (More) Secure Software

Steve Bellovin

908-582-5886

smb@research.att.com

AT&T Research

Murray Hill, NJ 07974

If our software is buggy, what does that say about its security?

—Robert H. Morris

What's the Problem?

- Software is buggy.
- Bugs in security-sensitive software are often security holes.
- On the whole, the profession does not know how to write correct code.
- But we can—and should—write software that is more correct, and more secure.

Principles of Software Engineering

- Simplicity is a virtue.
- If code is complex, you don't know if it's right (but it probably isn't).
- In a complex system, isolate security-critical modules. Make them (at least) simple and correct.

Interface Design

- Modules should have a clean, clear, precisely-defined interface.
- Reliance on global state is bad; it's too hard to know what's going on at any given time.
- Use of explicit parameters make input assumptions explicit. (It also makes multiple contexts possible, but that's not always good.)

Transitive Trust

- Programs have to trust any programs they invoke.
- ☞ Trusted programs should not invoke any untrustworthy programs.
- Similarly, they must ensure that untrusted programs are not invoked indirectly.

Race Conditions

- Permission-checking cannot be separated from access.
- Otherwise, the attacker can substitute a new object for the one that was checked.
- *Never* use the `access()` system call for security.
- It usually fails—but with race conditions, you only have to win once.

C++ Lite

- Using `class` definitions forces interface definition.
- Internal module structure hidden from the outside.
- Constructors and destructors (can) simplify storage management.
- Judicious use of `operator` definitions can simplify code structure.
- But full C++ is arguably too complex to use.

Case Study: Firewalls

- Firewalls are needed because most network applications on most computers are not trustable. That is, they may be buggy.
- Firewalls work because they minimize the amount of code that needs to be trusted.
- *If you don't run the code, it doesn't matter if it's buggy and insecure. Firewalls shed code to exploit this principle.*
- In short, firewalls are a network response to a software engineering problem.

Design Choices

- Eliminate `fingerd`, NFS, NIS, `rlogind`, etc.
- Restrict things like FTP.
- Deploy secure authentication.

Case Study: `gopherd`

- Information retrieval protocol.
- System sends back responses including file type and "selector string".
- The selector string—often a file name—is sent back to the server to request that document.
- Control files are used to force special actions, such as off-site links.
- But what if the user creates a bogus selector or control file?

Implementation Choices

Option 1: Let the daemon decide what files should be accessible from the outside.

- Parsing a file name is hard (though simpler on Unix than on many other systems).
- `gopherd` invokes other programs via the shell; what about shell metacharacters?
- What about file read permissions?

Option 2: Let the system do it.

- Use `chroot()`; let the system control the file system name space. (The basic `chroot()` code is 15 years old, quite small, and quite reliable.)
- No worries (from this perspective) about metacharacters; other files (in effect) aren't there.
- Rely on either `chroot()` or system file modes for read permissions.

Advantages of Option 2

- No complex, error-prone permission-checking code in the application.
- Easily accomodates different system-level permissions (i.e., military-style mandatory access controls).
- Simpler structure, more likely to be correct.

Improving the Solution

Problem: `chroot()` is a privileged operation; dare we give `gopherd` that much power?

Solution: Extremely early `chroot()` or outboard module that sets up the unprivileged environment before invoking `gopherd`.

Problem: Shell metacharacters can still be used to execute inappropriate commands.

Solution: Don't use the shell; it's too powerful. Execute commands explicitly.

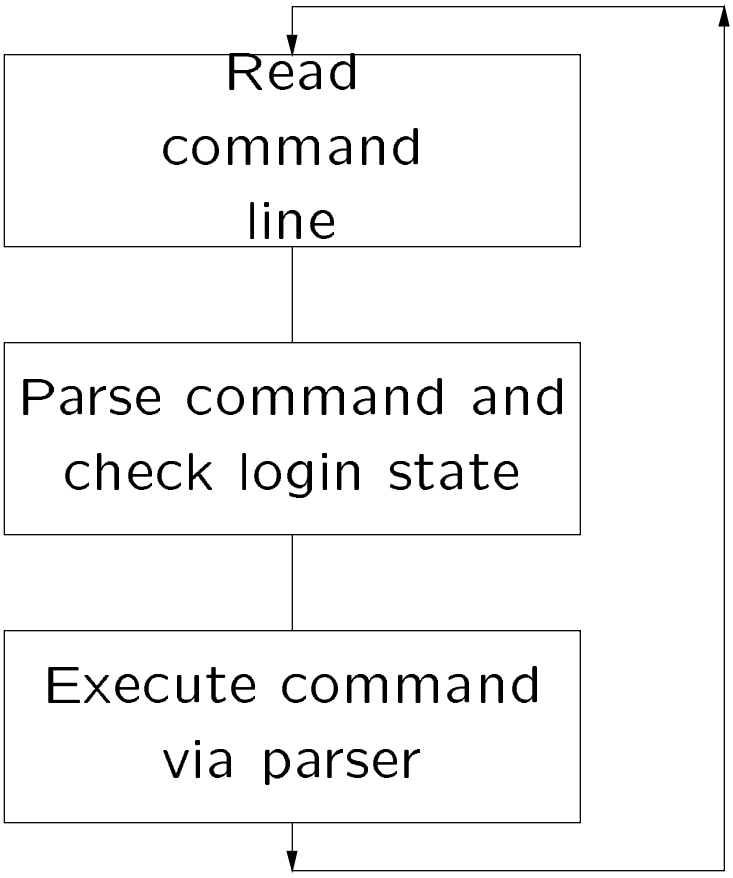
Problem: Shared `ftp`/`gopher` area can let users change the `gopherd` structure and/or cause execution of inappropriate commands.

Solution: Use the system permission mechanism so that `gopherd` *can't* access files written by `ftpd`.

Case Study: `ftpd`

- Current implementation has overly-general control structure. There is a lot of reliance on global state variables.
- There are too many privileged operations in the current code, which implies a need for a lot of bookkeeping.

`ftpcmd` Main Loop



Login Sequence

`USER` command:

Clear login state
Get user profile
Check for `anonymous`; set flag

`PASS` command:

If not anonymous, check password; on failure, clear state and exit.
Set directory, uid, from retrieved user profile.
If anonymous, use `chroot()` for access control.
Set logged-in flag.

Analysis

- Too much retained state: user profile, `anonymous` flag, logged-in flag.
- Twisty control structure: too much other code can be executed during login sequence.
- `chroot()` — the primary security mechanism for anonymous FTP — done quite late, and depends on all that saved state.

☞ There have been at least three separate bugs reported in the login process.

Solution

```
do {
    read command
    if (command == USER)
    print error message
    read next line
} while (TRUE)

if USER==ANONYMOUS
    chroot()
    set ANONYMOUS permissions
    read and ignore PASS command
else
    read PASS
    get user profile
    if !valid exit
    set user permissions
while (!EOF) {
    command processing
}
exit
```

Exercise: What would you change for a firewall-only `ftpd`?

Fixed Solution

```
do {
    read command
    if (command == USER) break
    print error message
    read next line
} while (TRUE)

if USER==ANONYMOUS
    chroot()
    set ANONYMOUS permissions
    read and ignore PASS command
else
    read PASS
    get user profile
    if !valid exit
    set user permissions
while (!EOF) {
    command processing
}
exit
```

With so little code, the bugs are (more) obvious.

Case Study: Remote Execution in `uucp`

- Currently executes only a few commands, under the `uucp` login.
- But spooled remote execution and file transfer are useful in general.
- Can we add such a capability to `uucp`?

Choice 1 — Let `uucp` do it

- Only `root` can switch to an arbitrary user account. Should `uucp` run as `root`?
- Command execution is only loosely coupled to remote machine identity; is the coupling secure enough that it can be trusted?

Choice 2 — Use an outboard privileged module

- Limit what has to be trusted. (N.B. `uucp` is a large, complex program with a long history of security bugs.)
- Use cryptographic authentication instead of a password.
- Example: Koenig's `asd` package, layered on top of `uucp`, has exactly 200 lines of privileged code, plus 274 lines to calculate a cryptographic checksum.
- Bonus: you can now use other transport mechanisms.

How `asd` Words

```
unseal -K /etc/keys/user | \
  instpkg | \
  mail user
```

Case Study: `rcp` and `rdist`

- `rcp` and `rdist` use the `rsh` protocol.
- The `rsh` protocol, apart from its reliance on names, requires that the client program be on a "privileged" port (whatever that is).
- Thus, `rcp` and `rdist` run as `root`.
- Both have a long history of security holes...

Solutions

1. Don't use the protocol directly; invoke the `rsh` command itself.
2. Invoke a small, trusted program that will set up the socket and pass back an open file descriptor (but watch out for race conditions).
3. Use a *real* authentication mechanism.

Case Study: Web Security

- The servers are huge, complex programs that do permission checking, parse file names, pass state around, switch uids, run user-written CGI scripts—and try to manage cryptographic keys, accept sensitive input, and manage access to restricted content.
- Clients are told what to do by the server, with choices ranging from the benign—display this picture or simple text—to redirections to other URLs, Postscript pictures, and, ultimately, Java. All the while, they must guard client keys, user identities, and the whole machine and network they're running on.
- The solution is left as an exercise for the profession.

What Have We Done?

- Rely on simple primitives (`chroot()`, operating system permission-checking, etc.).
- Explicit separation of access control from general complexity.
- Elimination of unnecessary code.

Authentication

- Many security problems are due to authentication failures.
- Such failures are often very difficult to detect, because the intruder appears to be a valid user.
- Most authentication failures come from bad assumptions about the environment.

Types of Authentication

Passwords An obsolete technology. Passwords can be guessed, given away, or collected by Trojan horses or eavesdroppers.

Name-based Relies on the trustworthiness of the network and on the integrity of the address-to-name mapping system.

Biometric Requires special-purpose hardware; gives probabilistic result; subject to biological disruptions (does a voiceprint work if the employee has a cold?).

Cryptographic Generally the best mechanism, but a great deal of care is required.

Cryptography

- Cryptography can be used for privacy and for authentication.
- The basic tools—conventional and public-key cryptography, secure hash functions, digital signatures, etc.—are generally used as the building blocks for *cryptographic protocols* such as Kerberos.
- But the design of these protocols is a tricky matter; errors abound. There are many published examples that simply aren't secure.
- Among the dangers: cut-and-paste attacks, replays, attacks on one party's clock, multiple concurrent sessions.

Sample Protocol Failure (by Needham and Abadi)

Message 1 $A \rightarrow S : A, B$

Message 2 $S \rightarrow A : CA, CB$

Message 3 $A \rightarrow B : CA, CB, \{\{K_{ab}, T_a\}_{K_a^{-1}}\}_{K_b}$

But B can resend parts of that message to C, pretending to be A:

Message 3′ $B \rightarrow C : CA, CC, \{\{K_{ab}, T_a\}_{K_a^{-1}}\}_{K_c}$

The quantity $\{K_{ab}, T_a\}_{K_a^{-1}}$ is treated as a black box.

General Rules for Cryptography

- Know your enemy—how strong must the cryptography be?
- Protocol failures are more dangerous, since once found they require less expertise to exploit.
- Dedicated cryptographic hardware is a Good Thing; general-purpose computers should not, in general, be entrusted with secret keys.

☞ But good cryptography is *not* a replacement for correct software.

Assurance—How Do We Know Our Code is Correct?

☞ We never *really* know.

- Formal certification of secure systems *does* address assurance; this is often overlooked by Feature Creatures.
- Example: "B2-compliant" means a few features were added; a B2 evaluation requires well-structured code, configuration management, etc.
- Newer security standards (i.e., the Canadian document) make assurance orthogonal to features.

What Should You Do?

- Use good software engineering practices.
- Analyze your assumptions.
- Build a connectivity matrix. Who is able to talk to whom? Who is *allowed* to? How do you enforce this?
- Rely on small, simple programs for security checking.
- But remember that computer security isn't everything.


HACKERS