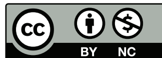
Complexity and Security



An Early Warning

"Finally, although the subject is not a pleasant one, I must mention PL/I, a programming language for which the defining documentation is of a frightening size and complexity. Using PL/I must be like flying a plane with 7,000 buttons, switches, and handles to manipulate in the cockpit. I absolutely fail to see how we can keep our growing programs firmly within our intellectual grip when by its sheer baroqueness the programming language—our basic tool, mind you!—already escapes our intellectual control. . . .
"When FORTRAN has been called an infantile disorder, full PL/I, with its growth characteristics of a dangerous tumor, could turn out to be a fatal disease."

Edsger W. Dijkstra, 1972

Complexity is Bad

- We've known since the beginning of computers that it's hard to write correct code
- We've known for decades that complexity leads to buggy code.
- Why?

Perfection

> First, one must perform perfectly. The computer resembles the magic of legend in this respect, too. If one character, one pause, of the incantation is not strictly in proper form, the magic doesn't work.
>
> Frederick P. Brooks, Jr.
> *The Mythical Man-Month*

- Code has to be (nearly) perfect to work
- The more complex it is, the harder it is to grasp all of it, end to see the interactions between the different pieces
- In theory, code's mental complexity is $O(n^2)$ in the number of lines of code
- The goal of modularization techniques is to cut that to $O(m(\frac{n}{m})^2 + (m\epsilon)^2)$: code in each module interacts only within the module, plus APIs to other modules

Security

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

Robert H. Morris

Buggy Code is Insecure Code

- Bugs are just as likely in security-sensitive code as in “ordinary” application code
- Example: an open source Yubikey server has a really bad SQL injection attack
- Security-sensitive code has to be correct, or it might be insecure
- How?

The Fundamental Problem

- The real issue: interaction
- To be secure, a program must minimize interactions with the outside
- All interactions must be controlled

Relative Attack Surface Quotient

- RASQ: Relative Attack Surface Quotient
- Microsoft metric of how vulnerable an application is
- Roughly speaking, it measures how many input channels it has
- Must reduce RASQ

Not All Channels Are Equal

- Some channels are easier to exploit
- Some are more accessible to attackers
- Some have a bad track record

RASQ Examples

- Weak ACLs on shared files: .9—names are generally known; easy to attack remotely
- Weak ACLs on local files: .2—only useful to attacker after initial compromise
- Open sockets: 1.0—potential target

Note Well: *Relative*

- We cannot assign an absolute value to attack surface
- We can compare two different alternatives
- In other words, we do not say “this is insecure”; rather, we say “this is *less* secure”

Note Well: *Attack Surface*

- We are also not measuring code correctness
- Rather, we are measuring how many points an attacker can try to exploit
- RASQ says nothing about whether, say, socket-handling code is correct or not; rather, it says “Danger: here is socket code”
- We can compare two programs to see which has fewer danger points
- It also points us at areas of code that demand more scrutiny and more testing

Reducing RASQ: A Management Issue

- RASQ is a tool; you have to use it properly
- Example: Microsoft decreed that the RASQ of a subsystem could not go up
- The security group reviewed all code and had the authority to block *anything* from shipping
- Security is partially a *management* problem

Security and Complexity

- Complex code is buggy and hence insecure
- We thus have four challenges
  1. To the extent possible, eliminate complexity
  2. Protecting the unavoidably complex (i.e., buggy) application code from attackers
  3. Presenting a simple interface to the world
  4. Ensuring that our security code is simple

Reducing Complexity

Rule 1 Follow standard advice on good programming, modularity, etc.

Reducing Complexity

Rule 1 Follow standard advice on good programming, modularity, etc.

Rule 2 There is no Rule 2

Living with Complexity

- There are some unavoidably complex programs—there is no way to build a simple web browser for today's world
- (Personally, I think the web took a very dark turn with some of that complexity, but I was outvoted)
- Strategy: security boundaries between some modules: isolate complex code!

Example: Web Browsers

- Rendering HTML is inherently complex and risky: HTML comes from the enemy
- JavaScript is even worse
- But: accepting user clicks keystrokes is not sensitive
- Copying a pixel string to the display is not complex
- So: let that guide your modularization

First Cut: Web Browser Design

- Process HTML in a separate process
- Probably handle JavaScript in yet another process
- Do the user interface in a third process
- Have a simple message-passing interface between the processes
- Why? Because processes are a security boundary; one process cannot (to a first approximation) read or modify another process' memory

Strengthening the Design

- Sandbox the risky processes
- Why? To protect the operating system (and hence user files) if the complex code is buggy and insecure
- All current operating systems support some form of sandboxing

More Security Boundaries

- Web sites don’t trust each other
- You also don’t want user cookies leaking
- Have a process per site visited
- (It’s more complex than that; see the reading)

Proper Modularization: The 4.3BSD FTP Daemon (1986)

- (Not interesting by itself, but it's a good example.)
- Implements the standard File Transfer Protocol
- Input defined by RFC 959; no ability to change it
- Small enough to understand; large enough to provide examples, good and bad. . .

The FTP Protocol

- Download and upload files
- Sequence of simple, 3- and 4-letter commands
- Commands have zero or one operands
- Responses prefixed by 3-digit result code
- Must support *anonymous ftp* — unauthenticated access to restricted set of resources
- Alternatively, permit login with username and password

Sample FTP Session

```
$ ftp ftp.netbsd.org
220 ftp.NetBSD.org FTP server (NetBSD-ftpd 20040809) ready.
USER anonymous
331 Guest login ok, type your name as password.
PASS anything
230 Guest login ok, access restrictions apply.
LIST
150 Opening ASCII mode data connection for '/bin/ls'.
     (data transferred on separate TCP connection)
226 Transfer complete.
FBAR
500 'FBAR': command not understood
```

Things to Notice

- USER and PASS are separate commands
- 331 indicates only one command can follow: PASS (rename also uses a 300-class reply)
- 200-class replies indicate success
- 100-class replies are intermediate states
- 400- and 500-class replies are temporary and permanent failures

The Structure of FTPD

- Read a command line at a time
- Parse the line
- Before executing most commands, see if the user is logged in—some commands are legal before login
- Use flag and state variables for multi-command sequences such as USER/PASS and RNFR/RNTO
- Use `chroot()` to contain anonymous FTP users

Consider This Command Sequence

```
USER anonymous
CWD ~root
PASS anything
```

How is it processed?

Processing USER

- Set the anonymous login flag
- Retrieve the anonymous entry from `/etc/passwd`
- This will be needed for its home directory and uid

Processing PASS

- Check the anonymous login flag
- If set, accept any password; otherwise, check the password against the retrieved `/etc/passwd` entry
- Do "login" processing: setuid to that user, `chdir()` to the home directory
- If anonymous login, do a `chroot()` before giving up root privileges
- But there's a problem in the parser...

What's Wrong with This Parser?

- The legal sequence is

  USER
  PASS
  session commands

- ftpd's parser treats all commands the same, including USER and PASS
- This is a recipe for trouble...

The Fatal Sequence

```
USER anonymous
CWD ~root
PASS xxx
```

The USER command read in the `/etc/passwd` data for the anonymous login.

The Fatal Sequence

```
USER anonymous
CWD ~root
PASS xxx
```

The USER command read in the `/etc/passwd` data for the anonymous login. When parsing the `CWD` command, the `˜root` operand was parsed—and the profile entry for `root` retrieved—before the command was rejected as untimely.

The Fatal Sequence

```
USER anonymous
CWD ~root
PASS xxx
```

The USER command read in the `/etc/passwd` data for the anonymous login. When parsing the `CWD` command, the `˜root` operand was parsed—and the profile entry for `root` retrieved—before the command was rejected as untimely.

The PASS command completed the login sequence—as root. . .

The Heart of the Problem

- The security of the code depended utterly on setting and checking state variables
- The flaw was that other commands could change other state
- The programmer had to keep track of all of those state changes—but didn't get it right

Solution 1

- All commands are not equal!
- USER can be followed only by PASS; no other commands are valid until after logging in
- Why should the parser accept anything else?
- In particular: *do not* even try to parse other commands until after login

Solution 1

```
def dologin():
    while True:
        ubuf = readinput()
        if ubuf[0:3] != "USER": continue
        pbuf = readinput()
        if pbuf[0:3] != "PASS": continue
        if checklogin(ubuf, pbuf): return
```

Solution 2

- Put the login code into a separate program
- When it's done executing successfully, and after it's set up the proper UID, directory, etc., `exec()` the program that handles the rest of FTP
- More secure—*all* the first program can do is log somone in
- The rest of the code simply doesn't exist
- Another advantage: to log someone in, you have to change UID, which requires root privileges
- This way, the bulk of the FTP daemon does not require any privileges

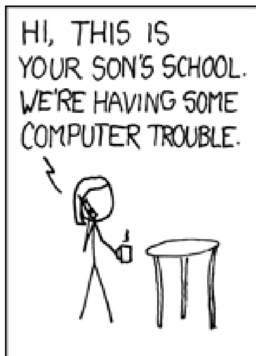
Reducing Complexity

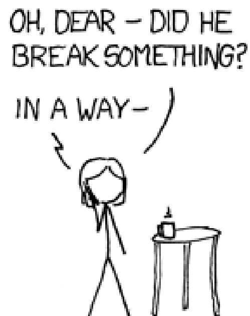
- Both solutions drastically reduce the complexity of the login sequence
- More precisely, they reduce its attack surface
- Since the login sequence requires privileges and the rest doesn’t, we want to make it utterly correct
- Complex code is more likely to be buggy. . .

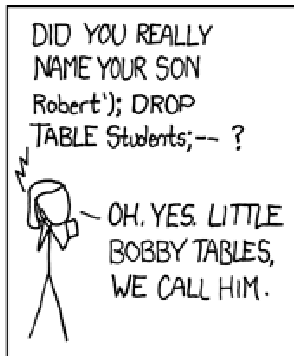
Guards

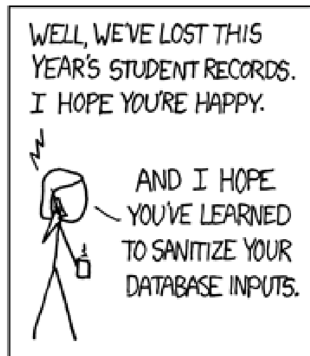
- Sometimes, it’s possible to put “guard” modules in front of complex code
- Guards sanitize inputs, limit string lengths, etc.
- These can be buggy, too, of course—but formal specifications help
- Lexical analyzer generators, parser generators, etc., are your friend

What Went Wrong Here?









(From http://xkcd.com/327/)

SQL Injection Attacks

- Suppose a program is querying an SQL database based on a userID and query string:

```
sprintf(buf, "select where user=\"%s\" && query="%s\"", uname, query);
```

- What if query is

```
foo" || user="root
```

- The actual command passed to SQL is

```
select where user="uname" && query = "foo" || user="root"
```

- This will retrieve records it shouldn't have

Input Sanitization?

- Simple answer: the student's name wasn't processed properly
- A name with quotes should have been rejected, or the quote mark should have been escaped
- Input sanitization is a good idea—but robust design is better

Input Sanitization is Hard!

- What characters should you strip?
- Remember names like O’Brien, with a quote mark
- Remember Unicode
- Remember that Windows uses \ as a pathname separator while Linux uses /
- Remember that there’s a Unicode character that looks like a /
- Etc.

Interface Design

- The deeper problem was the interface between the input module and the database
- The program rendered it as a command string, necessitating a parsing operation
- A better answer: use SQL stored procedures
- No need for parsing!

Cheswick and Bellovin

"To paraphrase Einstein: make your security arrangements as simple as possible, but no simpler. Complex things are harder to understand, audit, explain, and get right. Try to distill the security portions down to simple, easy pieces."

How Do We Design Security-Sensitive Code?

- First and foremost: avoid complexity
- Second: modularization
- Third: proper interfaces
- In other words: the same basic principles, but here especially we want to be guided by execution environments

A Look Back at Our Authentication Design

Developers

1. MFA use should be required, including for social network admins
2. U2F is probably the best choice
3. Internal, locked-down database
4. Recovery via management chain and overnight shipping

Social Network Users

1. MFA should be available
2. U2F support is needed for employees; TOTP with soft tokens is more accessible to most users
3. Separate database for authentication only
4. Recovery via email, plus password for token loss

Why Separate Authentication Databases?

- Simplicity of code: no need for as many conditionals
- Separation of modules: one module does employee authentication; another does user authentication
- Isolation between modules: no way for the user authentication module to grant employee privileges; that code simply does not exist in that module

Protecting the Data

- Also: the user authentication module has no access to the employee authentication database, which is more sensitive
- How do we protect either authentication database from the its authentication module?
- Put the database on a separate server?
- Advantages and disadvantages—how do we analyze it?

Separating the Database

Advantages

- If the code is buggy and insecure, the database isn't directly reachable
- The database can be centralized, while login is distributed (but is that a good idea?)

Disadvantages

- We need another machine (probably a minor issue)
- We need another interface
- There is extra code, and perhaps extra complexity, to sending queries and receiving responses
- There is also a new failure mode: the authentication database isn't reachable

How Do We Analyze This?

- Execution environment: with separate machines, harder for an attacker to reach more data
- But: what is the interface like?
- If it's SQL-like—`select where user="foo"`—the attacker can dump the database or iterate through it
- We need a better interface: `isvalid(user, pw, MFA)`
- Note the difference: it's a semantic interface that enforces the separation of execution environments
- The server might even be able to do rate-limiting if each login server has its own credentials to access the database

What’s the Answer?

- It depends!
- We are trading complexity for assurance
- The exact answer will vary, depending on the threat environment—how likely is it that the login server will be hacked?—and the complexity of the actual interface design

Implementation Issues

- If your coding environment has a good, simple way to pass complex parameters safely, that reduces code complexity
- Examples: Python's `pickle` module; JSON encoder/decoders, some implementations of Remote Procedure Calls (RPC)
- The *library* may be more complex—but your code will be much simpler
- (Do you trust the library vendor?)
- A good mechanism makes separated databases more attractive

TLS Encryption

- Conceptually simple to set up
- Server: do crazy cryptographic handshakes, send client certificate chain plus *something* signed
- Client: verify signature, verify certificate chain, verify certificate validity, verify that the certificate contains the name you wanted to connnect to
- So why do so many apps get this wrong?

OpenSSL

- OpenSSL does many, many things
- There are many options, e.g., the list of symmetric ciphers accepted, the list of asymmetric ciphers, the list of hash functions, the key lengths, and more
- There are different over-the-wire encodings, BER and DER
- OpenSSL provides low-level routines for all of this, but doesn’t have the right high-level routines
- Consequence: programmers omit some validation steps
- We need a simpler high-level API

Other API Considerations

- *Must* protect keys—applications should not handle them
- Conclusion: do not provide any API to export keys, only to do things like encrypt, decrypt, verify, etc.
- Sometimes, though, we need to move keys around
- Answer: an API to “wrap” keys by encrypting them with another key
- This creates complexity—but it is necessary complexity, to preserve the proper execution environment

Language Protections

- Object-oriented languages are good for hiding interface details
- Example: C++ classes have public and private members
- This is not a strong security measure—injected machine code can get at private data—so what is it good for?
- But: it keeps the programers from doing bad things
- And: it allows for a future, more secure implementation—perhaps use an HSM?—if circumstances demand
- But: the real benefit is reduced code complexity

APIs

- API design is crucial
- It's also difficult—it's too easy to allow too much flexibility
- You can provide high-level routines that take the place of many low-level calls—but if the low-level routines are there, someone *will* use them
- Best guidelines: use good taste, and don't supply unnecessary options
- Yes, it's hard

Complexity

- Complexity is the enemy of security
- Reducing complexity was one of the original motivations for firewalls. In 1994, Bill Cheswick and I wrote

  > *Corollary 3.1 (Fundamental Theorem of Firewalls) Most hosts cannot meet our requirements: they run too many programs that are too large. Therefore, the only solution is to isolate them behind a firewall if you wish to run any programs at all.*

- In those days, of course, firewalls were small and simple—and that's no longer true. . .

Questions?

(Great egret, Central Park, April 1, 2019)