Secure Programming II

"I'm paranoid, but am I paranoid enough?"

Special Techniques for Secure Programs

- Buffer overflows are bad in any case

- Some problems are only a risk for security-sensitive programs

- But what is a "security-sensitive program"?

- A security-sensitive program is one that runs with one set of permissions and accepts input from somone with different (especially lesser) permissions

- Includes most network servers and setUID programs, and many system daemons

SetUID Programs Are More Sensitive

- Anyone on the local machine can invoke them

- Many environmental influences that can be controlled by the invoker

- On the other hand, network daemons can be accessed remotely

Macro Injection Attacks

- Suppose a program is querying an SQL database based on valid 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

- Stored SQL procedures are much safer

What Was Wrong with That Slide?

Did You Notice?

- I wrote `sprintf` instead of `snprintf`

- I was mostly trying to save room on a complex slide

- I was also curious to see who'd notice...

More Generally

- If you invoke an external program, be aware of its parsing rules

- Especially serious for languages like Shell, Perl, and Python, where data can be converted to statements and executed

- Example: what delimits different arguments to the shell?

- Blank, tab, newline? Why?

SQL Injection Attacks

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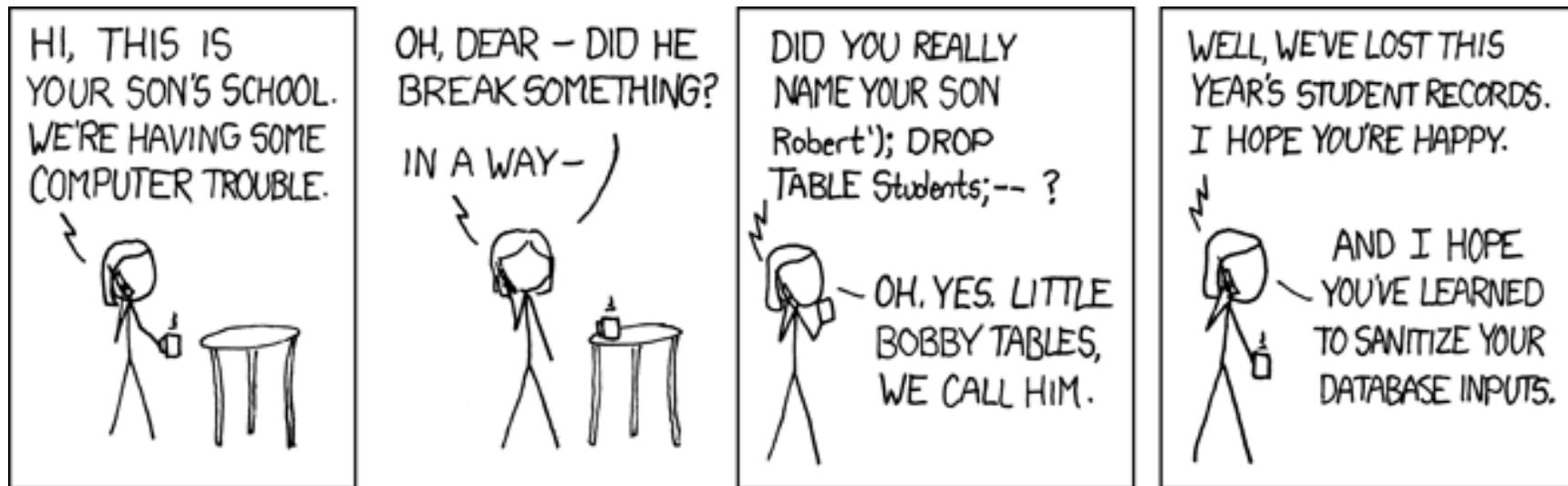
IFS

- The shell variable IFS lists the delimiters used when parsing command lines

- If you can change it, you can control the shell's parsing

- (The exact effects are subtle, because of the risks of just accepting it blindly — know your semantics!)

Other Sensitive Environment Variables

- **PATH** Search path for finding commands

  - If "**.**" is first,, you'll execute a command in the current directory. What if it's booby-trapped?

  - Secure programs should always use absolute paths or reset **PATH**

- **ENV** With some shells, a file to execute on startup

- **LD_LIBRARY_PATH** The search path for shared libraries

- **LD_PRELOAD** Extra modules loaded at runtime

Some of these are disabled for setUID programs, to minimize the risks

Search Paths

- What directories do programs come from? Components of programs?

- (Important for correctness as well as security — complex issue for Multics, in the 1960s.)

- Choices: program specification, user specification, system directories, current directory, location of base program, location of data file, probably more

File Descriptors

- Normally, file descriptor 0 is stdin, 1 is stdout, and 2 is stderr

- The `open()` system call allocates the first available file descriptor, starting from 0

- Suppose you close fd 1, then invoke a setUID program that will open some sensitive file for output

- Anything it prints to stdout will overwrite that file

- Similar tricks for fd 0

Some Other Inherited Attributes

current directory

root directory   *see* `chroot()`

resource limits   *see* `getrlimit()`

umask

timers   *see* `getitimer()`

signal mask

open files   *See the FIOCLEX option to* `ioctl`

Current uid

Effective uid

Process Creation on Windows

- The **CreateProcess** call creates processes on Windows

- Executing a new program is part of the process creation mechanism

- 10 parameters control the program to be executed, window creation, priority, security attributes, file inheritance, and much more

- The Windows call does more for you, but is it simpler?

- Do programmers have a better understanding of what is inherited, and the implications of those things?

Why Do These Matter?

- Will such a program misbehave?

- Will it core dump after having read a sensitive file? (Some systems prevent core dumps of setUID programs.)

- If the program terminates prematurely, will it leave some crucial resource locked?

Access Control

- Some privileged programs need to read or write user-specified files

- Example: local mailer, as we saw last week

- Other examples: web server (remote), lpr (setUID)

- Very tricky...

Remote Access Control

- Don't want to offer all system files to, say, web users

- Operating system doens't help — too many files are world-readable

- Web server must implement its own access control

- Several different levels

Filename Parsing

- User supplies pathname; application must check for validity

- Administrator specifies list of accessible files and/or directories

- Sometimes, wildcards — *, ?, and more — are permitted

- Application must *parse* supplied filename

- Remarkably difficult

The " .. " Problem

- Attackers try to get at other files

- Simplest attack: put `..` in the path

- `http://example.com/../../../../etc/passwd`

- The `..` can occur later:

- `http://example.com/a/b/../../../../etc/passwd`

- If directory **/dir** is legal, what about **/dir/../dir/file**? Do you want to count levels?

- Watch out for **/dir///../../file** — replicated **/**'s counts as a single one

- Note that **/foo..bar/bletch** is legal

Application Syntax Issues

- Applications can have their own weird syntax

- Example: in URLs, %xx can specify two hex digits for the character. `%2F` is the same as `/`

- When is that expanded?

- How is `/foo%2F..%2Fetc/passwd` processed?

Unicode

- Standard for representing (virtually) all of the world's scripts
  - ☞ There are proposals for Klingon and Tengwar ("Elvish") codepoints

- *Many* problems!

- Some symbols look the same, but have different values: ordinary **/** — technically called "solidus" — is U+002F, but U+2044, "fraction slash", looks the same

- "Combining characters" and "grapheme joiners" make life even more complicated. Thus, **á** can be U+00C1 or the two-character sequence U+0041,U+0301

- Comparison rules have to be application-dependent — and watch out for false visual equivalences; these have already been used for attacks, especially with Cyrillic domain names

Cyrillic Homograph Attack on "Paypal"

Glyph	Unicode value in Cyrillic
P	U+0420
a	U+0430
y	U+0443
p	U+0440
a	U+0430
l	U+006C (ASCII)

Operating Systems Don't Have Such Problems

- Conceptually, you're trying to permit certain subtrees.

- The application is trying to map a string into a subtree

- The OS has one mapping function; the application has another

- The OS doesn't care about the tree structure for access control; it uses its own mechanisms

- The OS stores permissions with the data; no separate parse is needed

File Access by SetUID Programs

- Some commands — `lpr`, for example — need to write to restricted places, but also read users' files

- Need permissions to write to spool directory; need user permissions to read users' files

- How can this be done?

First Attempt: Access() System Call

```
if (access(file, R_OK) == 0) {
    fd = open(file, O_RDONLY);
    ret = read(fd, buf, sizeof buf);
    ....
}
else {
    perror(file);
    return -1;
}
```

What's wrong?

Several Problems

- Only useful if setUID root – other UIDs can't open read-protected files.

- (I didn't check the return code on the `open()` call. . .)

- Race conditions

- Generic name: TOCTTOU (Time of Check to Time of Use)

Race Conditions

- There is a window between the `access()` call and the `open()` call

- The attack program can create a link to a readable file, invoke `lpr` in the background, then remove the link and replace it with a link to a protected file

- The probability of success is low but not zero — and the attacker only has to win once

Temporary Files

- The same attack can happen on files in `/tmp`

- The standard C library subroutine `mktemp()` is vulnerable to this

- Alternatives: `mkstemp()` or `mktemp()` with the `O_CREAT | O_EXCL` flags to `open()`

- Caution: if `open()` is used that way, generate a new template if EEXIST is returned

Shedding SetUID

- A setUID program can give up and then regain its setUID status:

```
save_uid = geteuid();
seteuid(getuid());
fd = open(file, O_RDONLY);
seteuid(save_uid);
```

- Better alternative: run unprivileged most of the time, but assume setUID status only when doing privileged operations

☞ But — watch for SIGINT, buffer overflows; injected code can reassume privileges, too

Lock Directories

- Have a parent directory that's mode 700, and a 777 subdirectory

- While privileged, do a `chdir()` to the subdirectory

- Give up privileges; write files in this subdirectory

Use a Subprocess

- Fork, and have a non-privileged subprocess open the user's files

- Option 1: copy the file contents to the parent process over a pipe — safe but slow

- Option 2: send the *file descriptor* via `sendmsg()`/`recvmsg()` over a Unix-domain socket

Issues with Message-Passing Systems

- File-opening permissions

- Authentication

- Other issues?

Opening Files

- How does the server open a private file? Two ways. . .

- The client opens the file and passes the open file descriptor

- The client sends some sort of access right — a *capability* — to the server

Authentication

- Who is allowed to send messages to the server?

- How does the server know the client's identity?

- Two solutions: support from the OS or cryptographic authentication

Other Issues?

- The buggy code problem doesn't go away

- It's very similar to the network security problem; it hasn't been solved, either

The Fundamental Problem

- The real issue: interaction

- To be secure, a program must minimize interactions with the outside

- All interactions must be controlled

RASQ

- 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

Generic Defenses

- Better OS

- What's a secure OS? *One that makes it easy to write security-sensitive programs*

- Most don't qualify. . .

Minimize Chances for Mistakes

- Eliminate unnecessary interactions

- Example: per-process or per-user `/tmp`

- Avoid error-prone primitives (i.e., minimize the chances of *comprehension* mistakes)

- Tight specification of input and environment — and check that it's all true