Program Structure I

Program Structure

- We've seen that program bugs are a major contributor to security problems
- We can't build bug-free software
- Can we build bug-*resistant* software?
- Let's look at a few examples, good and bad

What Are Our Goals?

- What makes software “bug-*resistant*”?
- We want to do two things:

☞ Minimize the chances that a bug will occur

☞ Minimize the consequences of any bugs that do occur

Minimizing the Chances of a Bug

- Keep the program small and simple
- Keep it well-structured
- Use proper modularization
- In other words, use all the tools we learn about in “how to program well” courses
- *This is the single most important thing we can do for security*

Minimizing the Impact of Bugs

- If we can't make the whole program bug-free, try to isolate the security-critical sections
- Use strong isolation between the security-critical sections and the rest
- Use strong confinement to isolate the non-critical sections from the rest of the system

Isolation and Confinement

- Forms of isolation:
  - Separate computer
  - Privilege separation
  - Process boundary
  - External program
  - C++ or Java class
  - C file
- We already discussed confinement
- Let's look at a real-world example

The 4.3BSD FTP Daemon (1986)

- Implements the standard File Transfer Protocol
- Input defined by RFC 959; no ability to change it
- Ancestor to many of today's FTP daemons, but smaller: 2600 lines of code versus 5000 (Redhat) or 8600 (NetBSD)
- 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
- Use a YACC grammar to parse input
- Add logged-in check as part of the grammar
- Use flag and state variables for multi-command sequences such as USER/PASS and RNFR/RNTO
- Use `chroot()` to contain anonymous FTP users

What is a YACC Grammar?

- Formal grammar to specify input syntax:

```
cmd : USER SP username CRLF
    | PASS SP password CRLF
    | CWD SP pathname CRLF
    | ...
```

- Parser-generator reads the grammar and generates C code to "recognize" the grammar
- C code sequences can be associated with each rule
- Code is executed when that rule is recognized by the parser

A Simpler Sample Grammar

```
expr : NUMBER operator NUMBER;

operator : PLUS | MINUS | STAR | SLASH;
```

The *terminal* symbols NUMBER, PLUS, MINUS, etc., are recognized by a *lexical analyzer*.

Are Parser Generators Good?

- That is, do they help security?
- Absolutely!
- By definition, a formal grammar specifies the *precise* input accepted; we've already seen that poor input specification can lead to security problems
- But — buffer overflows are more likely the result of lexical analysis, such as recognizing command names and pathnames
- You could use a lexical analyzer generator. . .

The Issues Here

- The over-the-wire protocol is quite simple; a formal grammar is probably overkill here
- The protocol is a poor match for the grammar implemented
- This is an implementation problem, not a problem with the concept of using formal grammars

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 grammar. . .

What's Wrong with This Grammar?

- The legal sequence is

  USER
  PASS
  session commands

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

A Closer Look at the Actual YACC Grammar

```
cmd   : USER SP username CRLF
      | PASS SP password CRLF
      | CWD check_login SP pathname CRLF
      | ...
```

The **`check_login`** is a pseudo-rule; it's just a hook for some C code that checks the logged-in flag

What's a "pathname"?

```
pathname : STRING
  = {
        if ($1 && strncmp((char *) $1, "~", 1) == 0) {
                $$ = (int)*glob((char *) $1);
                if (globerr != NULL) {
                        reply(550, globerr);
                        $$ = NULL;
                }
                free((char *) $1);
         } else
                $$ = $1;
    }
    ;
```

$1 is a pointer to a character string containing the filename.

What Does the Code Do?

- Only executed if the grammar rule is matched
- If the first character is `~`, it tries to do home directory expansion.
- That is, it replaces `~smb` by `/home/smb`
- To do that, `glob()` looks up `smb`'s record in `/etc/passwd`
- Hmm — two different routines are retrieving `/etc/passwd` entries. I wonder if that could cause trouble. . .

The Evils of Static Buffers

- Note the following text from the **`getpwnam()`** man page:

  > The return value may point to static area, and may be overwritten by subsequent calls.

- Processing "USER anonymous" calls

```
getpwnam("anonymous");
```

- Processing "**`~root`**" calls

```
getpwnam("root");
```

- The second call overwrites the buffer used by the first call

The Final Code Sequence

```
pw = getpwnam("anonymous");
if (user == "anonymous") guest = 1;
...
globpw = getpwnam("root");
...
if (!guest) { check password }
chdir(pw->pw_dir);
if (guest) chroot(pw->pw_dir);
setuid(pw->pw_uid);
```

In other words, it will do **`chdir("/")`**, **`chroot("/")`**, and **`setuid(0)`**!
Oops...

What Went Wrong

- The immediate problem is that the programmer forgot the semantics of **`getpwnam()`**
- The obvious — and implemented — fix was to save and restore the buffer before calling **`glob()`**
- But that ignores the real issue: improper modularization

Designing A Better Grammar

- All commands are not equal!
- USER can be followed only by PASS; no other commands are valid until after logging in
- Why should the grammar accept anything else?

A Better Grammar

```
ftpsess : user pass cmdseq
user : USER SP username CRLF
pass : PASS SP password CRLF
cmdseq : cmd | cmdseq cmd

cmd : CWD SP pathname CRLF
    | ...
```

This *forces* a login before any other code can executed

More Generally

- Any code can have a security vulnerability
- By limiting the code that can possible be executed as root, we limit our exposure
- We then use strong confinement mechanisms — **`setuid()`** and **`chroot()`** — to make the bulk of the code much less dangerous

Let's Take it Further

- How do we *know* that no other code will be executed before login is complete?
- Do we have sufficient confidence in YACC — and our understanding of it — to be *certain* that nothing else can be executed?
- Let's isolate things further

Breaking Up FTPD

- Split ftpd into two programs
- The first handles login: checking for anonymous ftp, validating the password if needed, doing the **`setuid()`** and **`chroot()`**
- It then **`exec()`**s the other program
- This second program, which always executes unprivileged, handles the bulk of the protocol
- The resulting ftpd is *smaller* — and the privileged section is only about 125 lines
- We can now have an even simpler anonymous-only login program for sites that don't offer full ftp

There's Another Problem...

- Under certain circumstances, ftpd is supposed to use TCP port 20
- Only root can bind to a low-numbered port
- If we've irrevocably given up root privileges in the login program, how can we do this?

How Did it Ever Work?

- Traditional ftpd has a login procedure, too
- It doesn't irrevocably give up root, it uses **`seteuid()`** instead
- It temporarily regains its privileges before binding to port 20, then release them
- Ugly, dangerous, and creates risks in case of buffer overflows or the like

Splitting Out Port 20

- Write a small setUID program that binds an open socket to port 20
- Create a socket, **`fork/exec()`** invoke this program
- When it returns, you have a socket; connect to the proper place
- That program is a bit tricky, because it has to verify that it's only invoked by ftpd
- It's still quite small, and it's better than uid-swapping

What Have We Done?

- We've divided the program into privileged and unprivileged sections
- We used strong isolation between the sections
- By getting rid of various flags, we simplified the program logic

It's Really Helped

- There have been other ftpd vulnerabilities in the login code — see, for one example,
  `http://www.cert.org/advisories/CA-1993-06.html`
- Splitting out the port 20 access may prevent the race condition attacks described in
  `http://www.cert.org/advisories/CA-1997-16.html`
- The best way to win is to get out of the game!

rsh/rlogin/rcp

- For various complex reasons, **`rsh`**, **`rlogin`**, and **`rcp`** need to set network connections as **`root`**
- (They're horribly insecure for network reasons, but we won't discuss that here.)
- How should that be done? Make them setUID?

SetUID Root?

- No — requires trusting complex code.
- (`rcp` is especially problematic.)
- No — leaves facility unavailable to other applications
- Yes — avoid extra data copy?

Strategies

- We only need `root` privileges to set up the connection
- Use an external program for that
- Pass an open file descriptor back?
- Or fork and pay the price of extra data copies?
- Both work — avoid privilege in the large program

Again...

- We separated out the privileged part
- We gained flexibility
- We increased security

Building a Mailer

- Use privilege separation (AKA confinement)
- Run part of the mailer sandboxed
- What goes where?

Normal Level

- User interface
- Sending mail
- ☞ You usually send through a trusted site
- Receiving mail
- Is that right? What about data-driven attacks?
- Saving mail

Sandboxed

- Opening attachments
- ☞ What about saved attachments?
- URLs: must have the *browser* sandbox them
- ☞ Can't rely on the sender; t's trivial to forge the sender of an email message
- Note: you never really know what a URL is. Click on

  `https://www.cs.columbia.edu/~smb/SMBlog-in-PDF.pdf`

  —it's *not* going to be PDF
- Mental homework: suppose the mailer can encrypt, decrypt, and digitally sign email. What privilege level should be used for those operations?