



IBM Research

UNIX File system

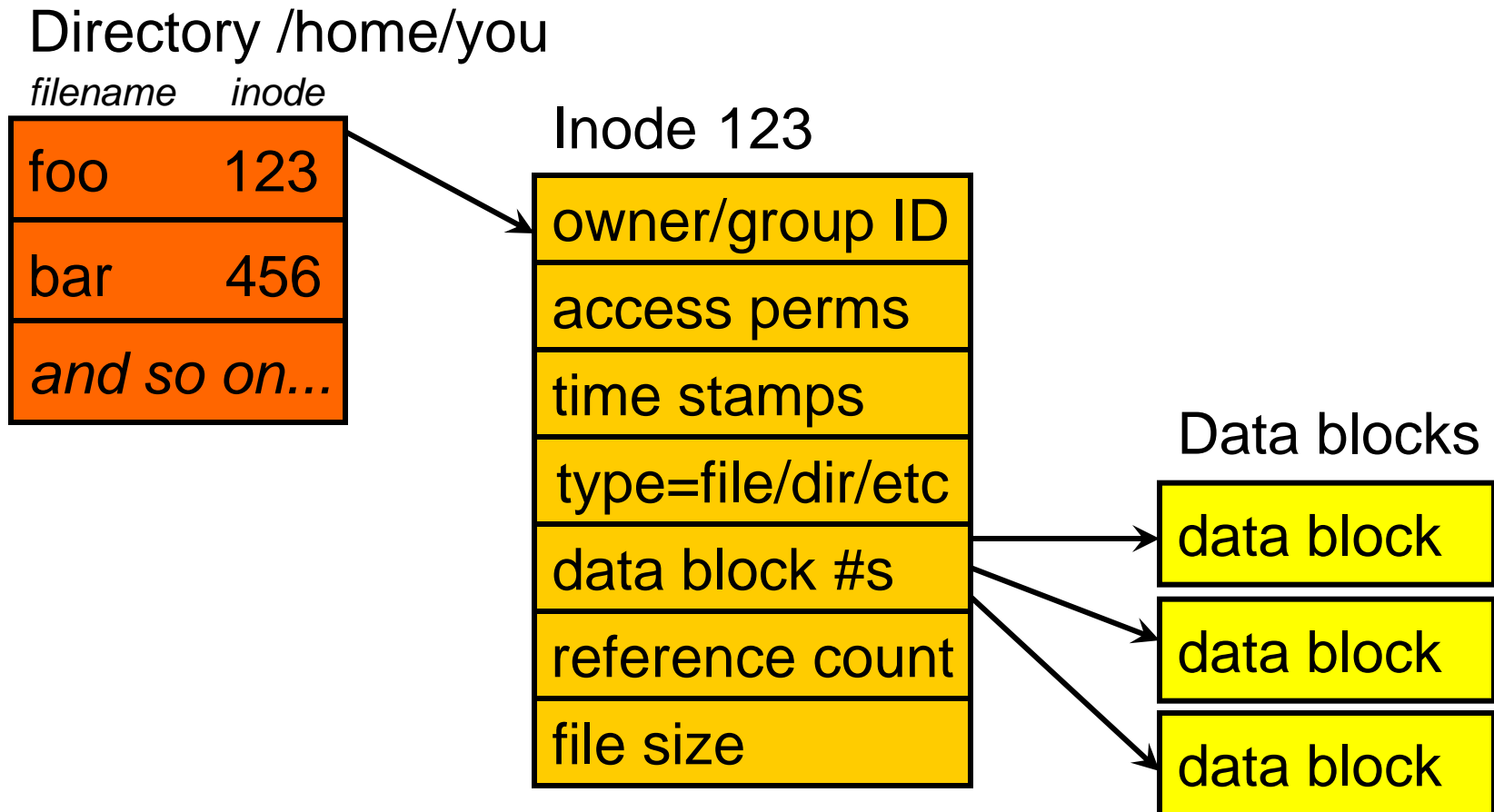
Traps, pitfalls, and solutions

Wietse Venema
IBM T.J.Watson Research Center
Hawthorne, NY, USA

Overview

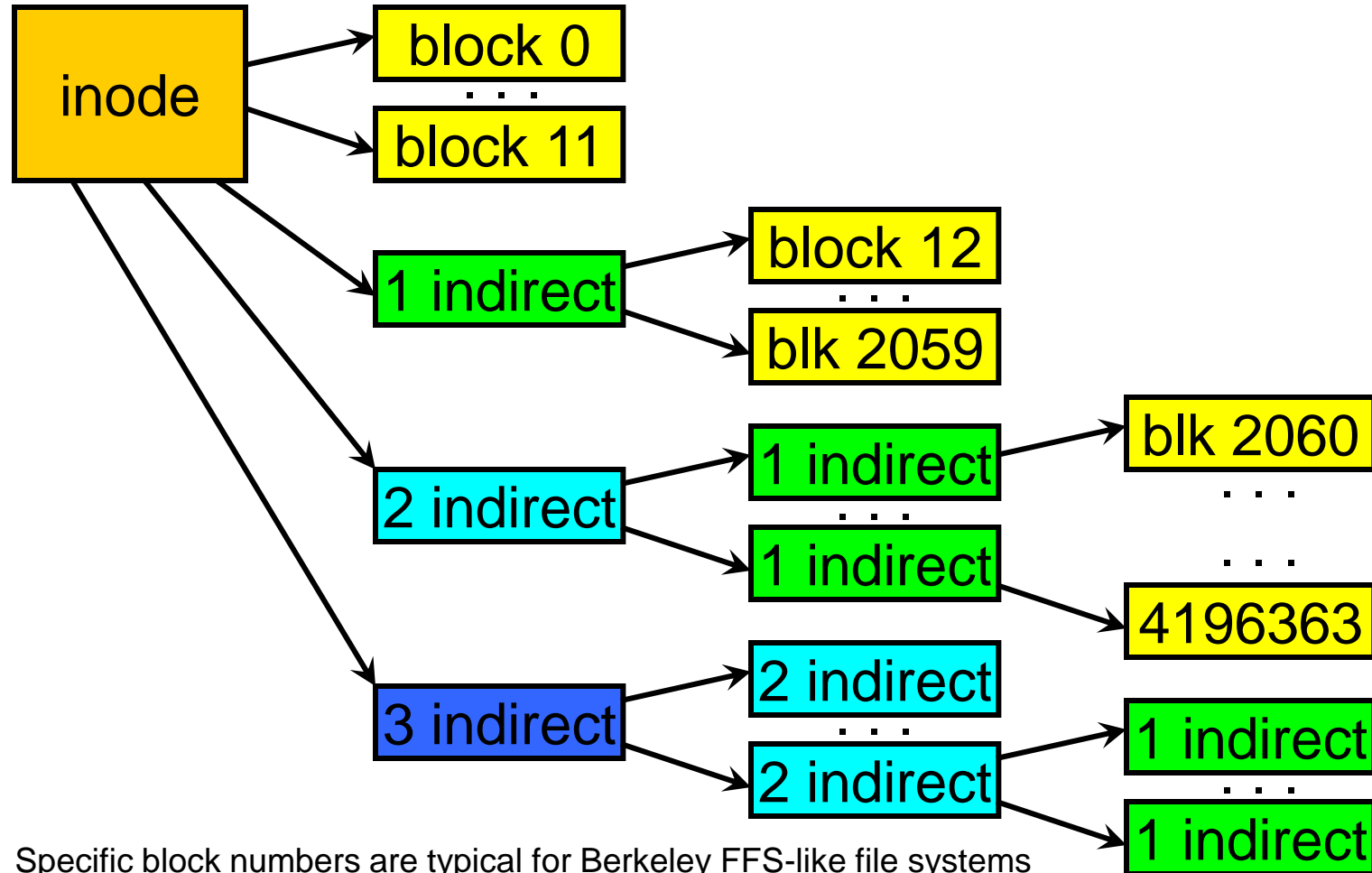
- UNIX file system architecture.
 - Features.
 - Gotchas (non-obvious consequences).
- Vulnerability case studies (not UNIX specific).
 - World-writable directories.
 - Race conditions.
 - Walking a hostile directory tree.
- Lessons learned.

UNIX file system architecture



Direct and indirect data blocks

(the truth, the whole truth, and nothing but the truth)



UNIX file system features (gotchas will be next)

- Separation of file name, file attributes, and file data blocks.
- Names may contain any character except “/” or null.
- Shared name space for files, directories, FIFOs, sockets, and device drivers such as */dev/mem* or */dev/ttya* (“everything is a file”).
- Permission check on *open/execute*, not *read/write*.
- Files can have holes (regions without data blocks).

UNIX file system gotchas

(unexpected consequences are bad for security)

- Feature: separation of file name, file attributes, and file data blocks.
 - *Multiple names* per file system object (multiple directory entries referring to the same file attribute block). Also known as multiple *hard links*.
 - Opportunities for name aliasing problems.
 - *Zero names* per file system object (when a file is deleted, the attributes and storage survive until the file is closed or stops executing).
 - A deleted file may not go away immediately.

UNIX file system gotchas

- Symbolic links provide another aliasing mechanism (a symbolic link provides a substitute pathname).
- Feature: a file name may contain any character except for “/” or null.
 - Beware of file names containing space, newline, quotes, other control characters, and so on.
 - Many UNIX systems allow ASCII codes > 127 , causing surprises with signed characters.
 - Example: `isalpha()` etc. table lookup with negative array index.

UNIX file system gotchas

- Feature: shared name space for files, directories, FIFOs, sockets, and device drivers such as */dev/mem* or */dev/tty01* (“everything is a file”).
 - The `open()` call may cause unexpected results (like blocking the program) when opening a non-file object.
 - Example: opening a FIFO or a serial port device driver.
 - Reading a non-file object such as */dev/mem* may lock up systems with memory-mapped hardware.
 - Example: reading device control registers.

UNIX file system gotchas

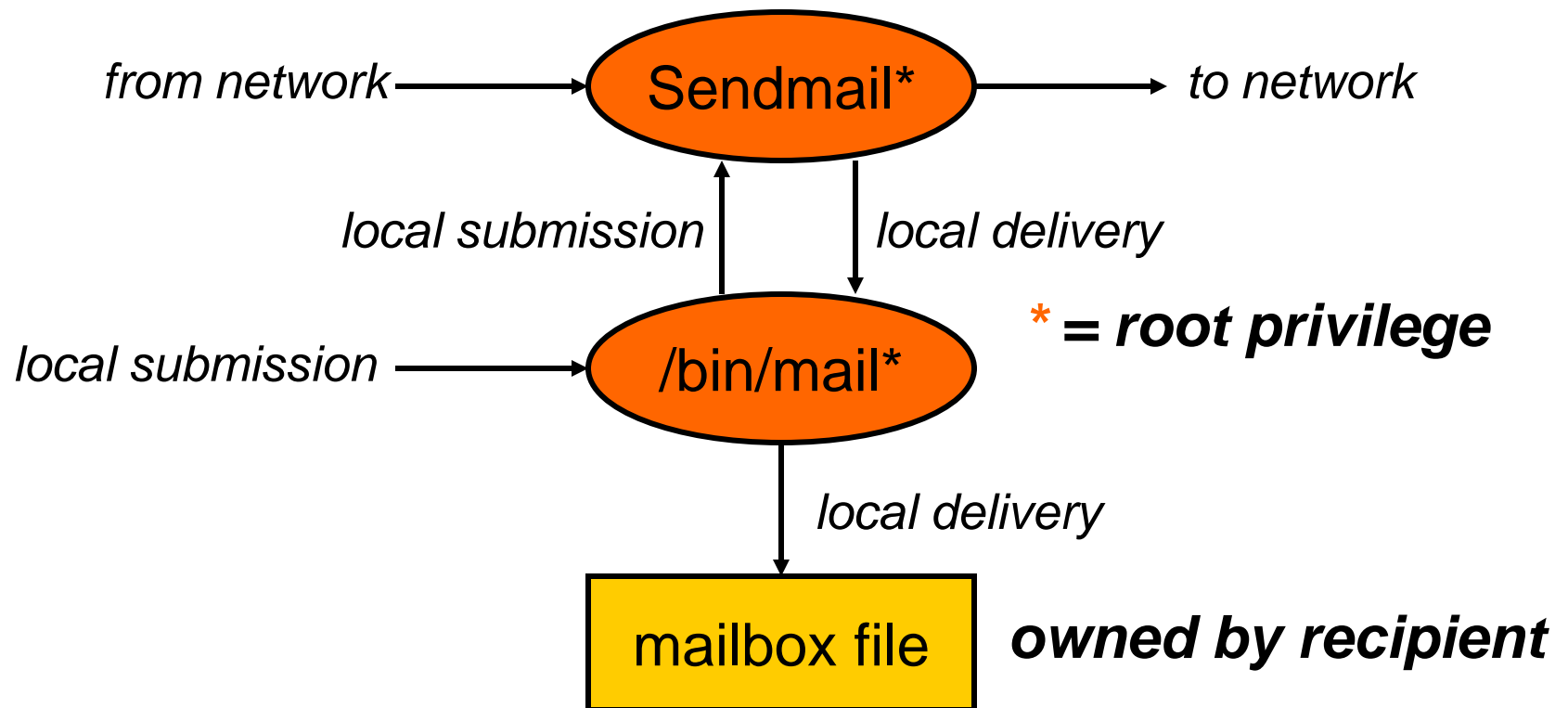
- Feature: access permission check happens on *open/execute* not *read/write*.
 - No general way to revoke access after a file is opened.
 - This also applies to non-file objects such as sockets.
 - Files must be created with correct permissions (as opposed to setting permissions after creating them).
- Feature: files can have holes (regions without data blocks; these read as blocks of null bytes).
 - The copy of a file can occupy more disk space than the original file.

- **File system case study: The evils of world-writable directories**

Overview

- Traditional UNIX mail delivery architecture.
- Multiple security problems caused by world-writable directories.
- Plugging the holes that result from bad design.
- “Solutions” introduce new problems.
- Fixing the problem requires changing the design.

Traditional UNIX mail delivery architecture



Traditional UNIX mail delivery architecture

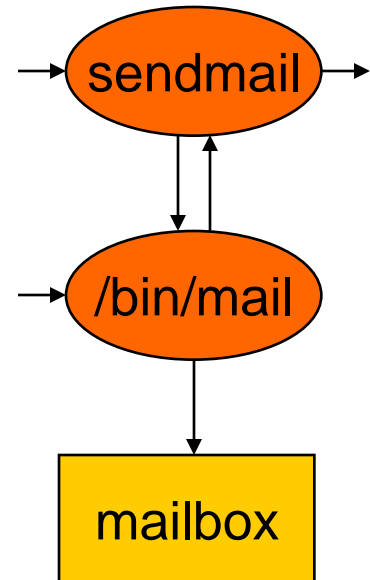
- Mailbox files are typically named */var/mail/username*.
- Mailbox files are owned by individual users.
 - Therefore, */bin/mail* needs root privileges so that it can create and update user-owned mailbox files^{1,2}.
- Mail reader programs are unprivileged.
 - Therefore, the */var/mail* mailbox directory needs to be world writable so that mail reader software can create */var/mail/username.lock* files.

¹Assuming that changing file ownership is a privileged operation.

²Historical Sendmail is privileged for more reasons (see part IV, Postfix).

/bin/mail delivery pseudocode

save message to temporary file
for each recipient
 lock recipient mailbox file
 append message to recipient mailbox file
 unlock recipient mailbox file

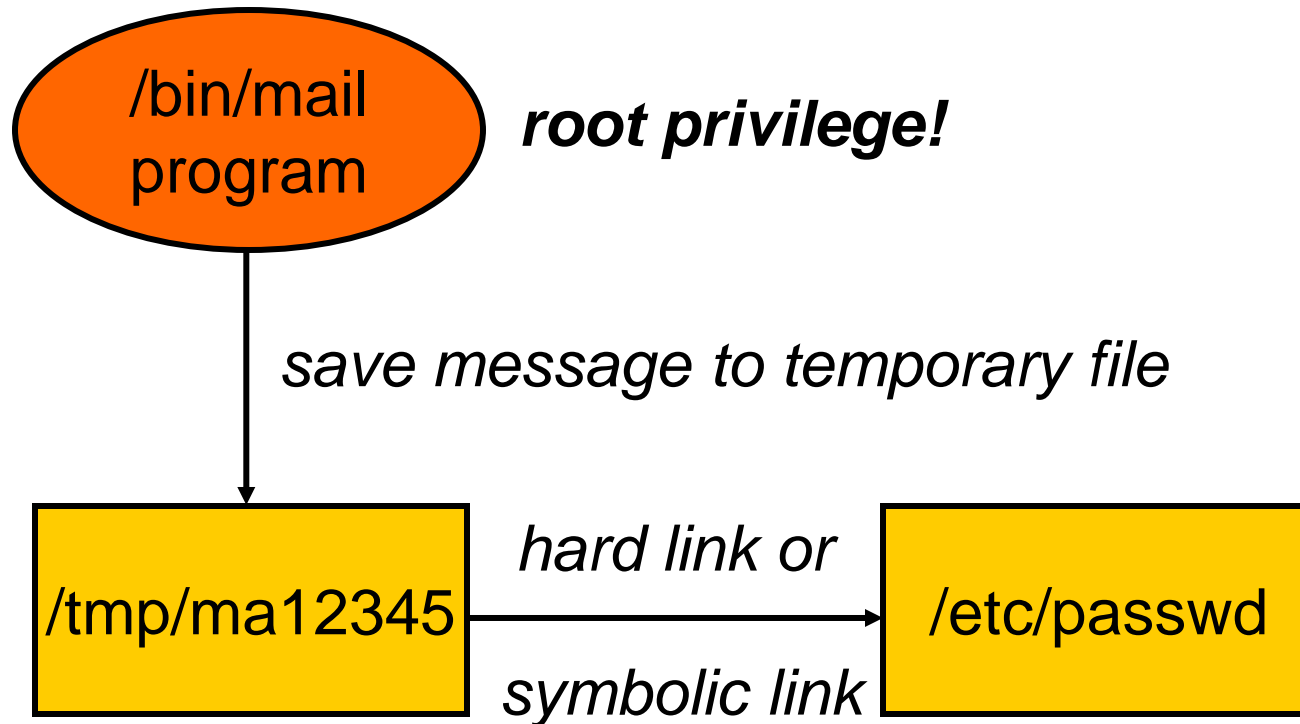


Step 1: save message to temporary file in world-writable directory

```
char  lettmp[ ] = "/tmp/maXXXXXX";    /tmp is world-writable
...
main(argc, argv)
char **argv;
{
    ...
    mktemp(lettmp);                    replace X's by some unique string
    unlink(lettmp);                   lame defense against attack
    ...                                window of vulnerability here
    tmpf = fopen(lettmp, "w");        maybe open the right file, maybe not?
}
```

From file bin/mail.c in archive .../4BSD/Distributions/4.2BSD/src.tar.gz

What can go wrong?

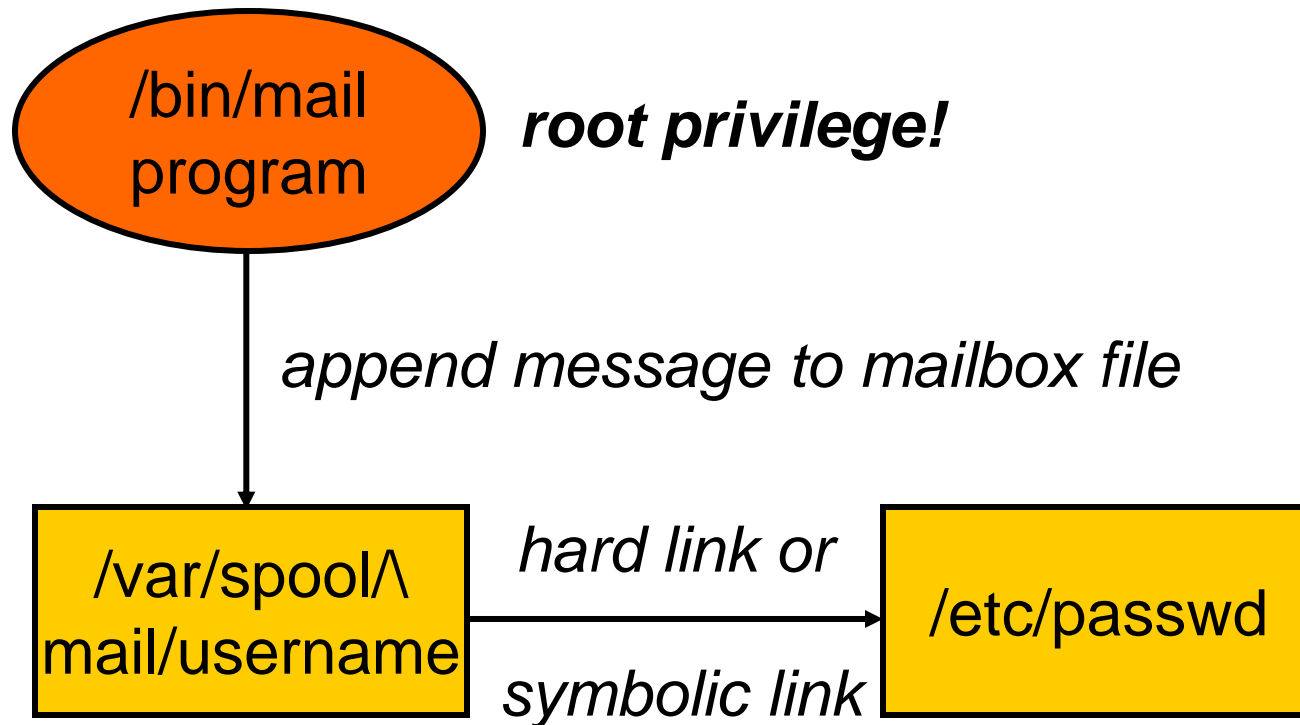


Step 2: append message to recipient mailbox file in world-writable directory

| | |
|--|---|
| <code>if (!safefile(file))</code> | <i>lame defense against attack with symbolic</i> |
| <code> return(0);</code> | <i>links or with multiple hard links</i> |
| <code>lock(file);</code> | <i>window of vulnerability here</i> |
| <code>malf = fopen(file, "a");</code> | <i>maybe open the right file, maybe not?</i> |
| <code>...</code> | <i>window of opportunity here</i> |
| <code>chown(file, pw->pw_uid, pw->pw_gid);</code> | <i>cool :-)</i> |
| <code>...</code> | |
| <code>copylet(n, malf, ORDINARY);</code> | <i>append message</i> |
| <code>fclose(malf);</code> | <i>XXX no error checking</i> |
| <code>...</code> | |
| <code>unlock();</code> | |
| <code>return(1);</code> | |

From file bin/mail.c in archive ../4BSD/Distributions/4.2BSD/src.tar.gz

What can go wrong?



Painless to safely create file in unsafe directory

- For example, to save the message to temporary file:

```
if ((fd = open(path, O_RDWR | O_CREAT | O_EXCL, 0600)) < 0)
    /* error... */
```

- Will not follow symbolic links to other files.
- Will not open an existing (hard link to) file.
- More convenient: *mkstemp()* creates a unique name and creates the file using the above technique.

Painful to open existing file in unsafe directory

(from Postfix MTA)

```
if ((fd = open(path, O_APPEND | O_WRONLY, 0)) < 0)           will follow symlink
    /* error: open failed */
if (fstat(fd, &fstat_st) < 0)                               get open file attributes
    /* error: cannot get open file attributes */
if (!S_ISREG(fstat_st.st_mode)                              check file type
    /* error: not a regular file */
if (fstat_st.st_nlink != 1)                                 check hard link count
    /* error: file has the wrong number of hard links */
if (lstat(path, &lstat_st) < 0                             won't follow symlink
    || lstat_st.st_dev != fstat_st.st_dev || lstat_st.st_ino != fstat_st.st_ino)
    /* error: file was removed or replaced */
```

Plugging /bin/mail like vulnerabilities with world-writable directories

- Create files with `open(.O_CREAT | O_EXCL. .)`. This protects against symlink/hardlink attacks.
 - Use `mkstemp()` to open a temporary file and to generate a unique file name at the same time.
- To open an existing file, compare `open()+fstat()` file attributes with `lstat()` file attributes. This will expose symbolic link aliasing attacks.
 - See also: the Postfix `safe_open()` routine.

“Solutions” introduce new problems

- Widely adopted remedy: group (not world) writable */var/mail* mailbox directory.
- Unfortunately, this introduces its own set of problems.
 - All mail reader programs need extra privileges to create */var/mail/username.lock* files.
 - All mail reader programs are now part of the defense (instead of only the */bin/mail* delivery program). That is a lot more code than just */bin/mail*.
- Thus, */bin/mail* still needs to defend against attack.

Lessons learned

- World-writable directories are the root of a lot of evil. They are to be avoided at all cost.
- Retrofitting security into a broken design rarely produces a good result.
- A proper solution addresses the underlying problem and changes the mail delivery model. This of course introduces incompatibility.

- **File system case study: The broken tree walker**

Overview

- Purpose of the privileged tree walking program.
- Buffer overflow problem due to mistaken assumptions about the maximal pathname length.
- There is no silver bullet. Long pathnames are a pain to deal with no matter what one does.
- How other programmers dealt with the problem.

Tree walker purpose

- Walk down a directory tree and examine the attributes of all files.
- This program is run while configuring the TCB¹ of a security system.
- The TCB may need updating whenever new software is installed on the system.

¹TCB=Trusted Computing Base, responsible for enforcing security policy.

What can go wrong?

- Get into trouble by following symbolic links so that you end up in an unexpected place.
- Get into trouble with non-file objects (like those in the */proc* or */dev* directories). This is “fixed” by blacklisting portions of the file system name space.
- Get into trouble with deeply nested directory trees.
- And more

Tree walker main loop

```
static void dir_list(char* dir_name, [other arguments omitted...])
{
    ...
    char    file_name[MAXPATHLEN];
    ...
    for (each entry in directory dirname) {
        sprintf(file_name, "%s/%s", dir_name, entry->d_name);
        if (file_name resolves to a directory)
            dir_list(file_name, [other arguments omitted...]);
    }
}
```

Note: MAXPATHLEN (typically: 1024) is the maximal pathname length accepted by system calls such as *open()*, *chdir()*, *remove()*, etc.

Tree walker vulnerability

- Buffer overflow in a security configuration tool! Real pathnames *can* exceed the MAXPATHLEN limit of system calls such as *open()*, *chdir()*, etc.
- Possible remedies:
 - Abort if pathname length \geq MAXPATHLEN.
 - Skip if pathname length \geq MAXPATHLEN.
 - Pass the problem to the user of the result.
 - Use *chdir()* to avoid system call failures within the tree walking program (requires using *fchdir()* to get back).
 - Use a variable length result buffer to avoid buffer overflows.

What did other programmers do?

- The UNIX *tar* (tape archive) format cannot store files with pathnames longer than 1024¹.
- The UNIX *find* command changes directory (*chdir()*) and leaves it to the user to handle long pathnames².
- Beware: changing directory can be dangerous when the directory tree is under control by an attacker.

¹See: Elizabeth Zwicky, *Torture-testing Backup and Archive Programs*.

²4.BSD, Solaris, Linux.

UNIX file system lessons learned

- Exercise extreme caution when doing anything in an untrusted directory or directory tree:
 - Creating a file. Hard/symlink attacks.
 - Open existing file. Hard/symlink attacks; non-files.
 - Reading a file. Non-file objects (FIFO, device, etc).
 - Removing a file. Hard/symlink attacks.
 - Manipulating file names. Spaces, control chars, ...
 - Changing directory. Where will you go today?

UNIX Lessons learned

- UNIX has been around for 35+ years. Its strengths and weaknesses are relatively well understood.
- As with many systems, shortcomings are the unintended result from decisions made long ago.
- Each fix will break at least one legitimate application.
- Experience teaches us to avoid what is broken and to build on the things that are good.