IBM Research

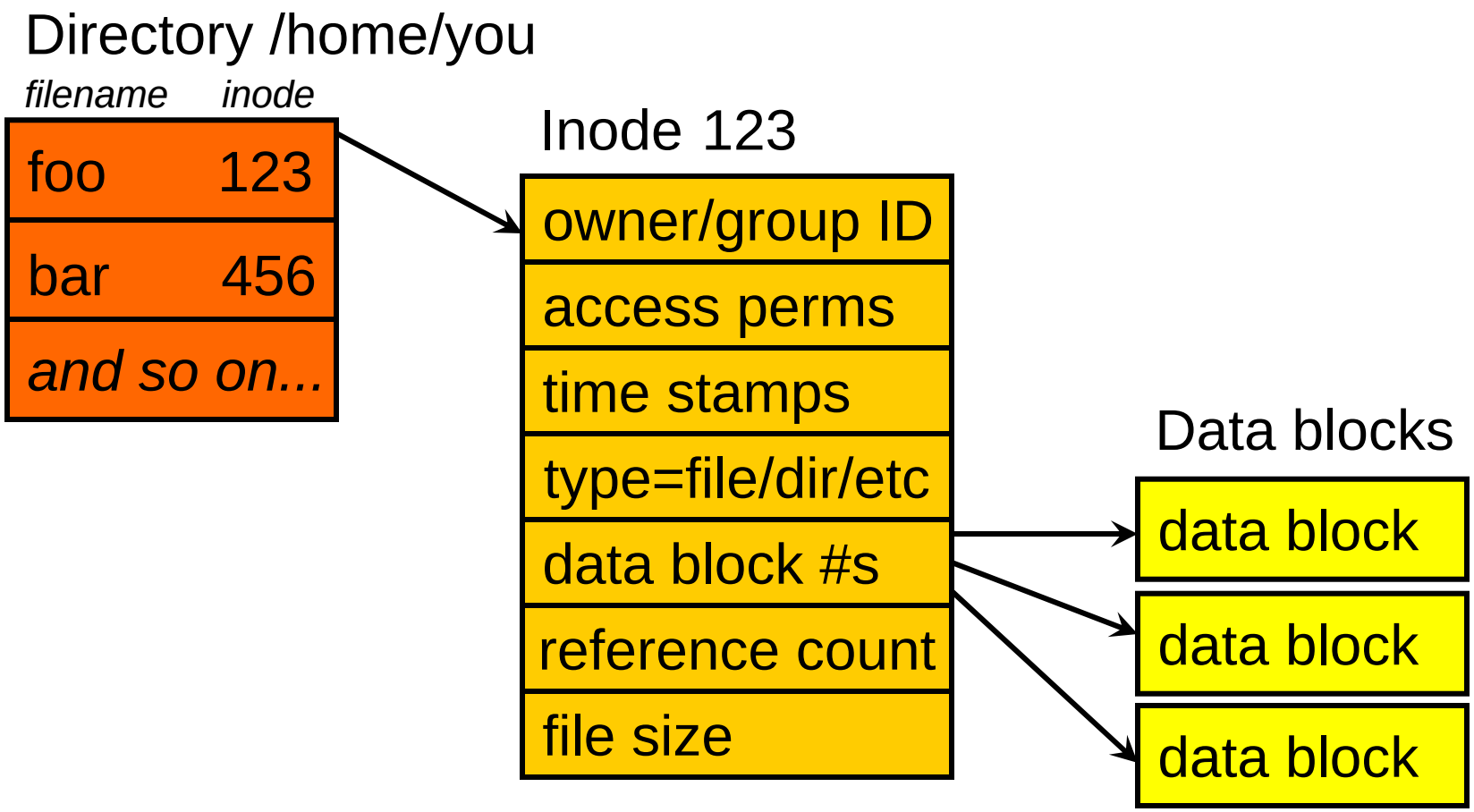
The broken file shredder
Programming traps and pitfalls

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

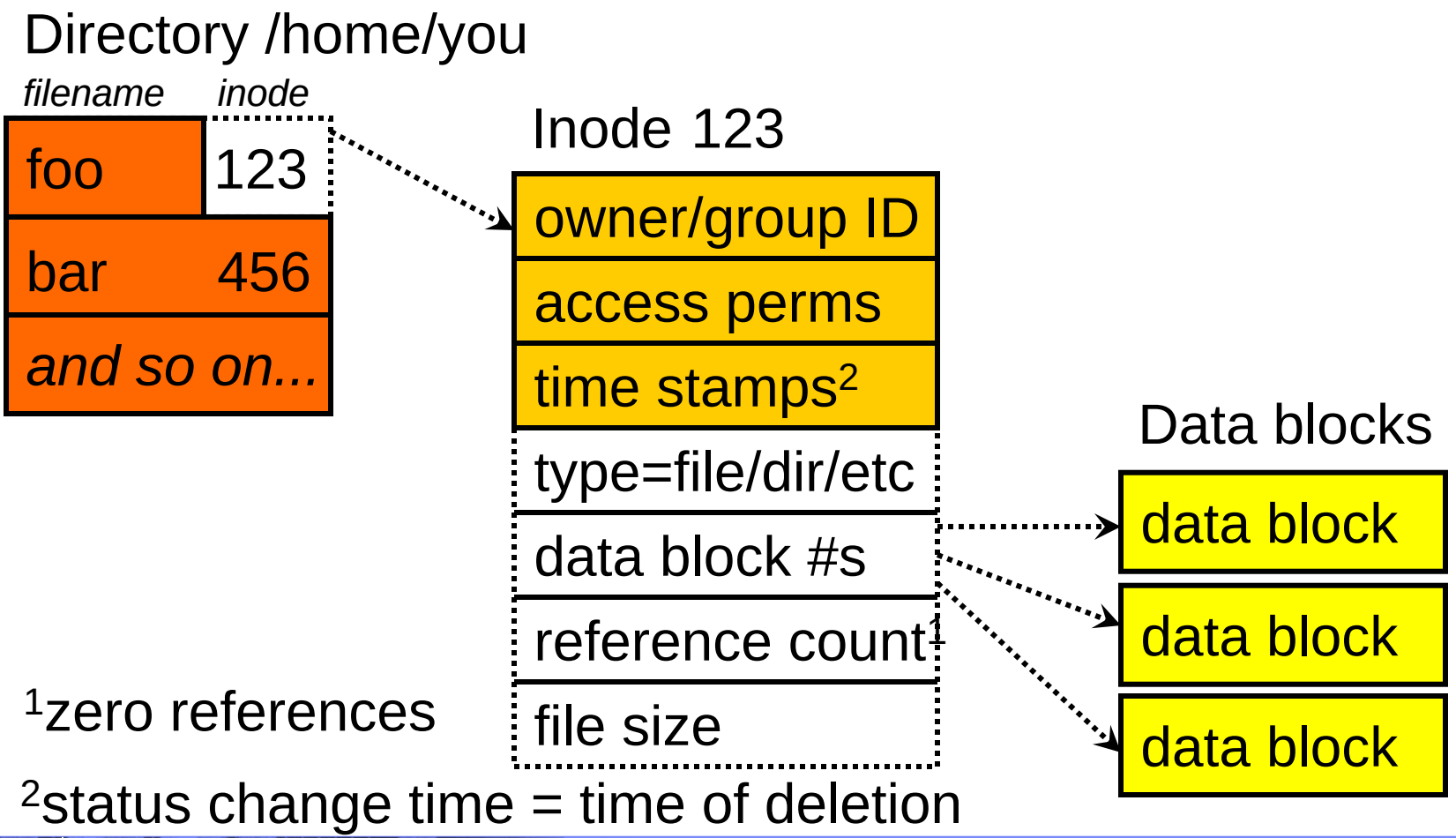
Overview

- What happens when a (UNIX) file is deleted.
- Magnetic disks remember overwritten data.
- How the file shredding program works.
- How the file shredding program failed to work.
- “Fixing” the file shredding program.
- Limitations of file shredding software.

UNIX file system architecture



Deleting a UNIX file destroys structure, not content



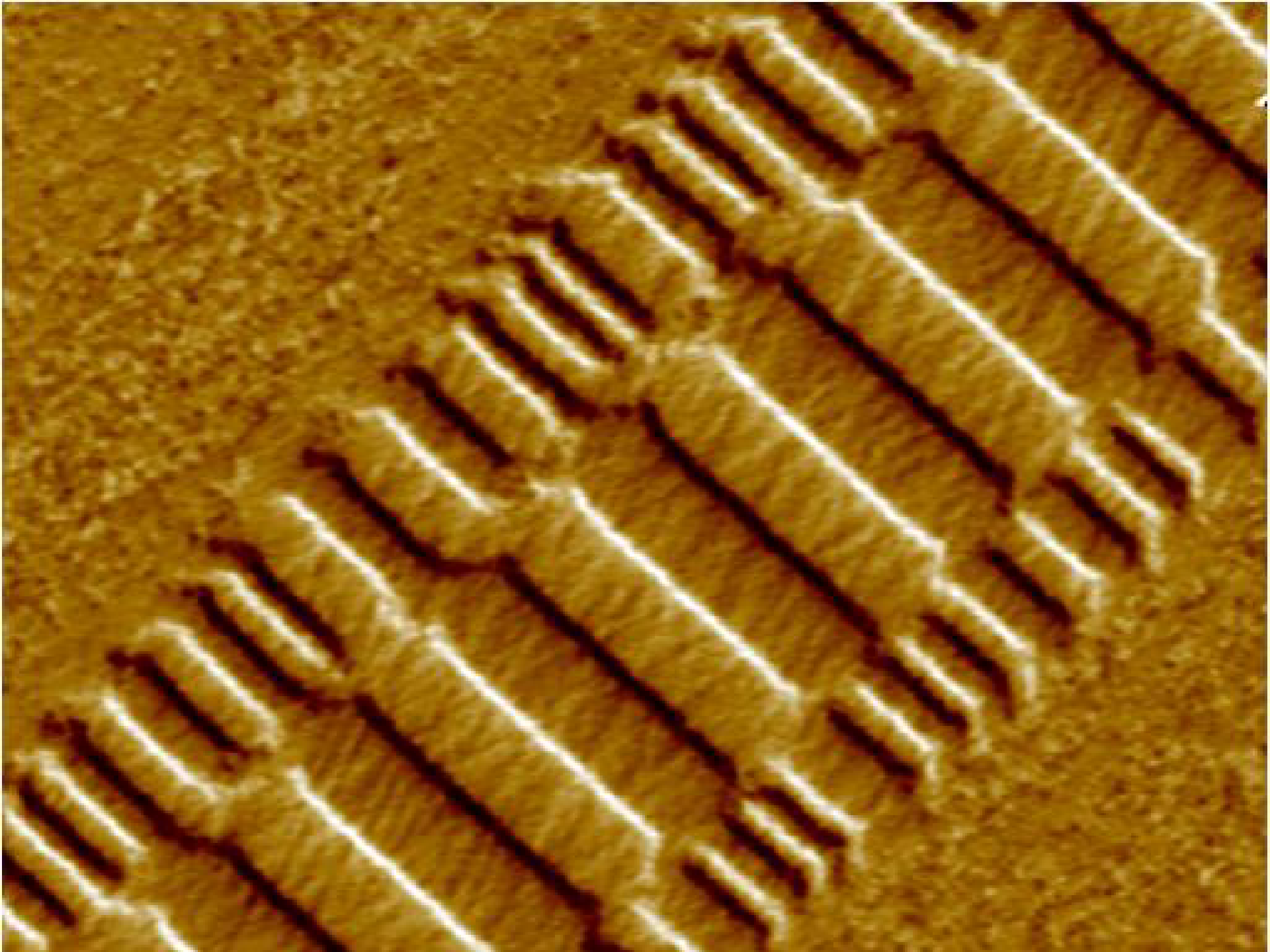
Persistence of deleted data

- Deleted file attributes and content persist in unallocated disk blocks.
- Overwritten data persists as tiny modulations on newer data.
- Information is digital, but storage is analog.

Peter Gutmann's papers: http://www.cryptoapps.com/~peter/usenix01.pdf

and http://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html

kool magnetic surface scan pix at http://www.veeco.com/



Avoiding data recovery with magnetic media

- Erase sensitive data before deleting it.
- To erase data, repeatedly reverse the direction of magnetization. Simplistically, write *1*, then *0*, etc.
- Data on magnetic disks is encoded to get higher capacity and reliability (MFM, RLL, PRML, ...). Optimal overwrite patterns depend on encoding.

mfm = modified frequency modulation; rll = run length limited;

prml = partial response maximum likelihood

File shredder pseudo code

```
/* Generic overwriting patterns. */
patterns = (10101010, 01010101,
        11001100, 00110011,
        11110000, 00001111,
        00000000, 11111111, random)

for each pattern
        overwrite file
remove file
```

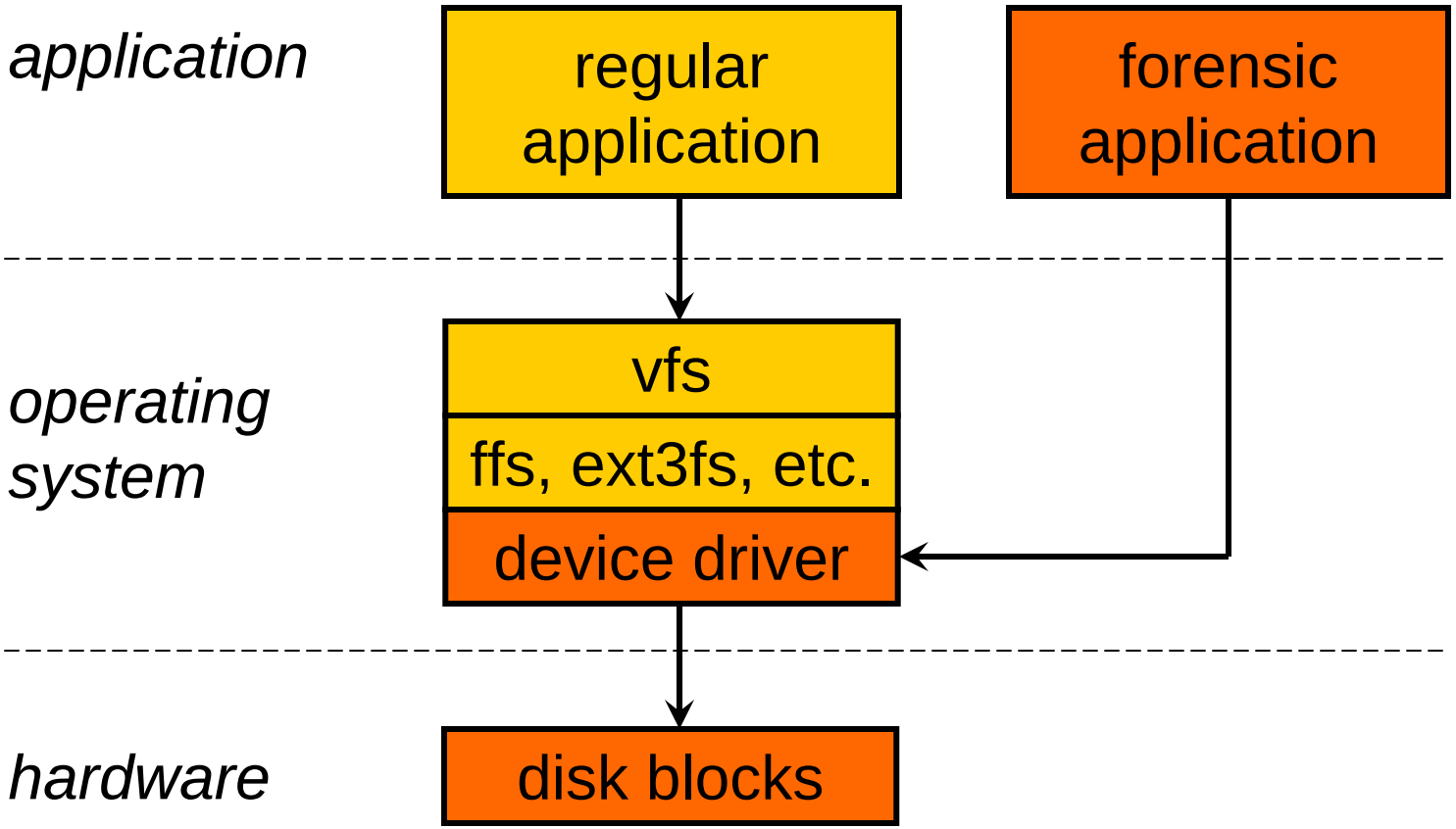
File shredder code, paraphrased

```
long overwrite(char *filename)
{
    FILE *fp;
    long count, file_size = filesize(filename);

   if ((fp = fopen(filename, “w”)) == NULL)
            /* error... */
    for (count = 0; count < file_size; count += BUFFER_SIZE)
            fwrite(buffer, BUFFER_SIZE, 1, fp);
    fclose(fp); /* XXX no error checking */

   return (count);
}
```

What can go wrong?

- The program fails to overwrite the target file content multiple times.
- The program fails to overwrite the target at all.
- The program overwrites something other than the target file content.
- Guess what :-).

Forensic tools to access (deleted) file information



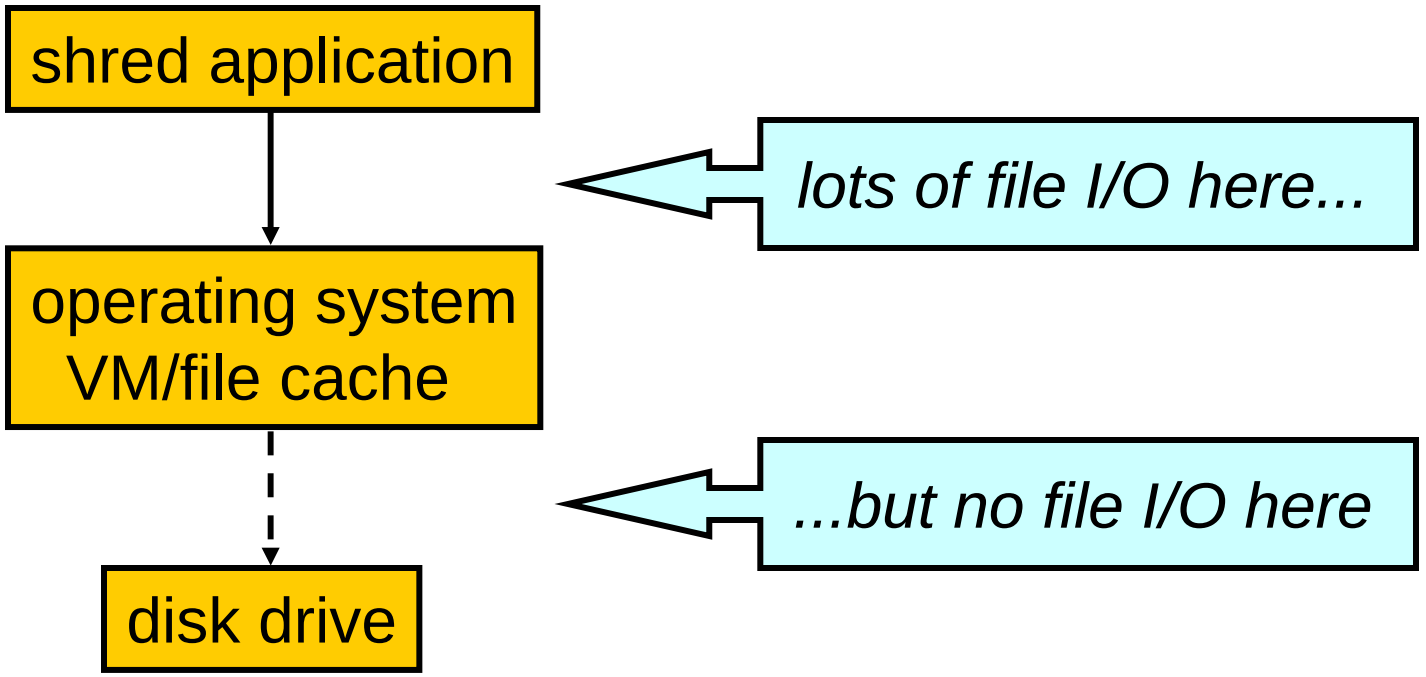
Coroner's Toolkit discovery

(Note: details are specific to the RedHat 6 implementation)

```
[root test]# ls -il shred.me                          list the file with its file number
1298547 -rw-rw-r--   1 jharlan  jharlan       17 Oct 10 08:25 shred.me
[root test]# icat /dev/hda5 1298547                   access the file by its file number
shred this puppy
[root test]# shred shred.me                           overwrite and delete the file
Are you sure you want to delete shred.me? y
1000 bytes have been overwritten.
The file shred.me has been destroyed!
[root test]# icat /dev/hda5 1298547                   access deleted file by its number
shred this puppy                                      the data is still there!

[root test]#
```

See: http://www.securityfocus.com/archive/1/138706 and follow-ups.

Delayed file system writes



File shredder problem #1
Failure to overwrite repeatedly

- Because of delayed writes, the shred program repeatedly overwrites the *in-memory* copy of the file, instead of the *on-disk* copy.

```
for each pattern
        overwrite file
```

File shredder problem #2
Failure to overwrite even once

- Because of delayed writes, the file system discards the *in-memory* updates when the file is deleted.
- The *on-disk* copy is never even updated!

```
for each pattern
        overwrite file
remove file
```

File shredder problem #3
Overwriting the wrong data

- The program may overwrite the wrong data blocks. *fopen(path,"w")* truncates the file to zero length, and the file system may allocate different blocks for the new data.

```
if ((fp = fopen(filename, "w")) == NULL)
        /* error... */
for (count = 0; count < file_size; count += BUFFER_SIZE)
        fwrite(buffer, BUFFER_SIZE, 1, fp);
fclose(fp); /* XXX no error checking */
```

“Fixing” the file shredder program

```
if ((fp = fopen(filename, “r+”)) == 0)          open for update, not truncate
        /* error... */
for (count = 0; count < file_size; count += BUFFER_SIZE)
        fwrite(buffer, BUFFER_SIZE, 1, fp);
if (fflush(fp) != 0)                            application buffer => kernel
        /* error... */
if (fsync(fileno(fp)) != 0)                     kernel buffer => disk
        /* error... */
if (fclose(fp) != 0)                            and only then close the file
        /* error... */
```

Limitations of file shredding

- Write caches in disk drives and/or disk controllers may ignore all but the last overwrite operation.
- Non-magnetic disks (flash, NVRAM) try to avoid overwriting the same bits repeatedly. Instead they create multiple copies of data.
- Not shredded: temporary copies from text editors, copies in printer queues, mail queues, swap files.
- Continued...

Limitations of file shredding (continued)

- File systems may relocate a file block when it is updated, to reduce file fragmentation.
- Disk drives relocate blocks that become marginal.
- Journaling file systems may create additional temporary copies of data (ext3fs: journal=data).
- Copy-on-write file systems (like Solaris ZFS) never overwrite a disk block that is “in use”.
- None of these limitations exist with file systems that encrypt each file with its own secret key.

Lessons learned

- An untold number of problems can hide in code that appears to be perfectly reasonable.
- Don’t assume, verify.
  - Optimizations in operating systems and in hardware may invalidate a program completely.
  - Examine raw disk blocks (network packets, etc.)
- Are we solving the right problem? Zero filling all free disk space (and all swap!) may be more effective.

IBM Research

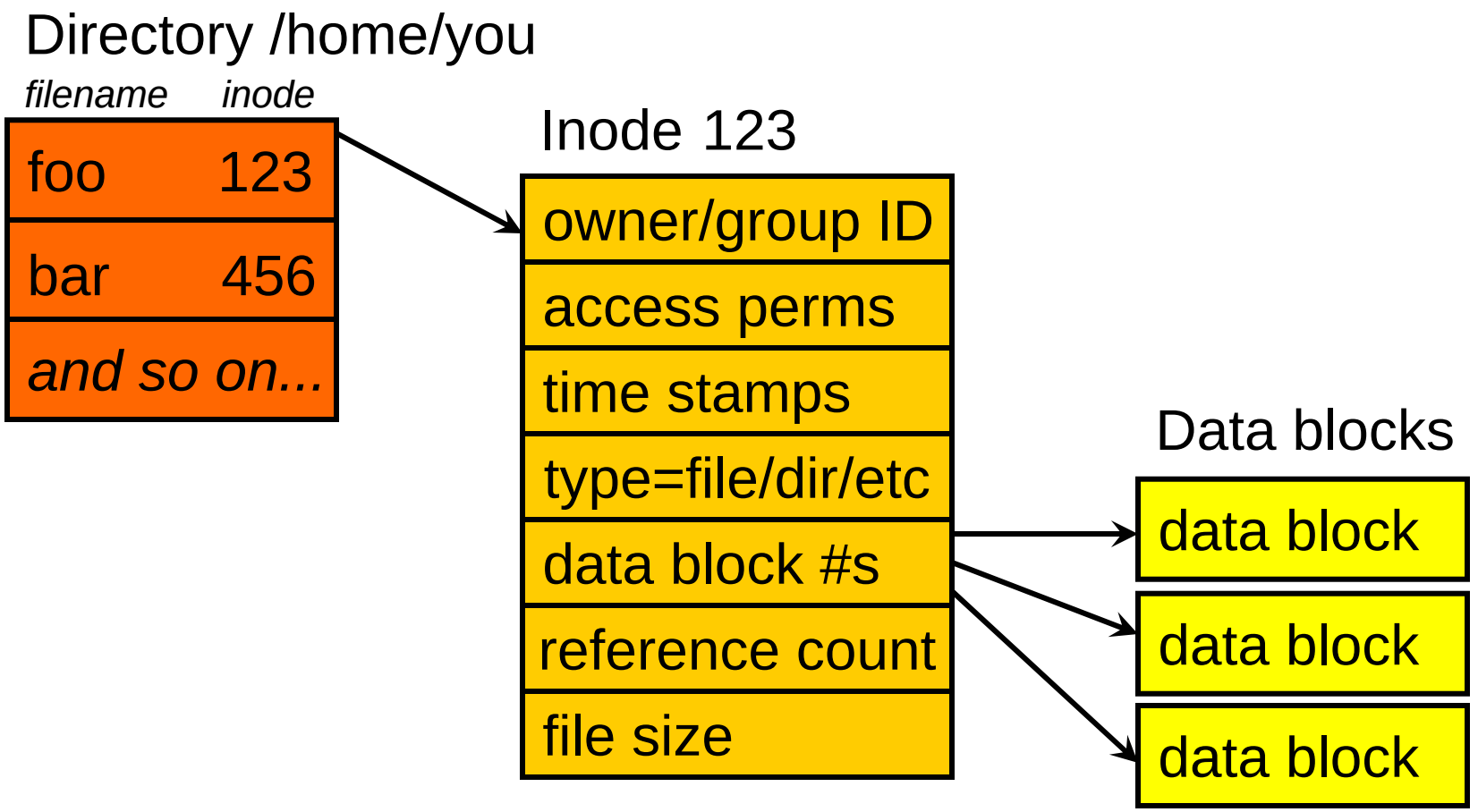
UNIX File system
Traps, pitfalls, and solutions

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

Overview

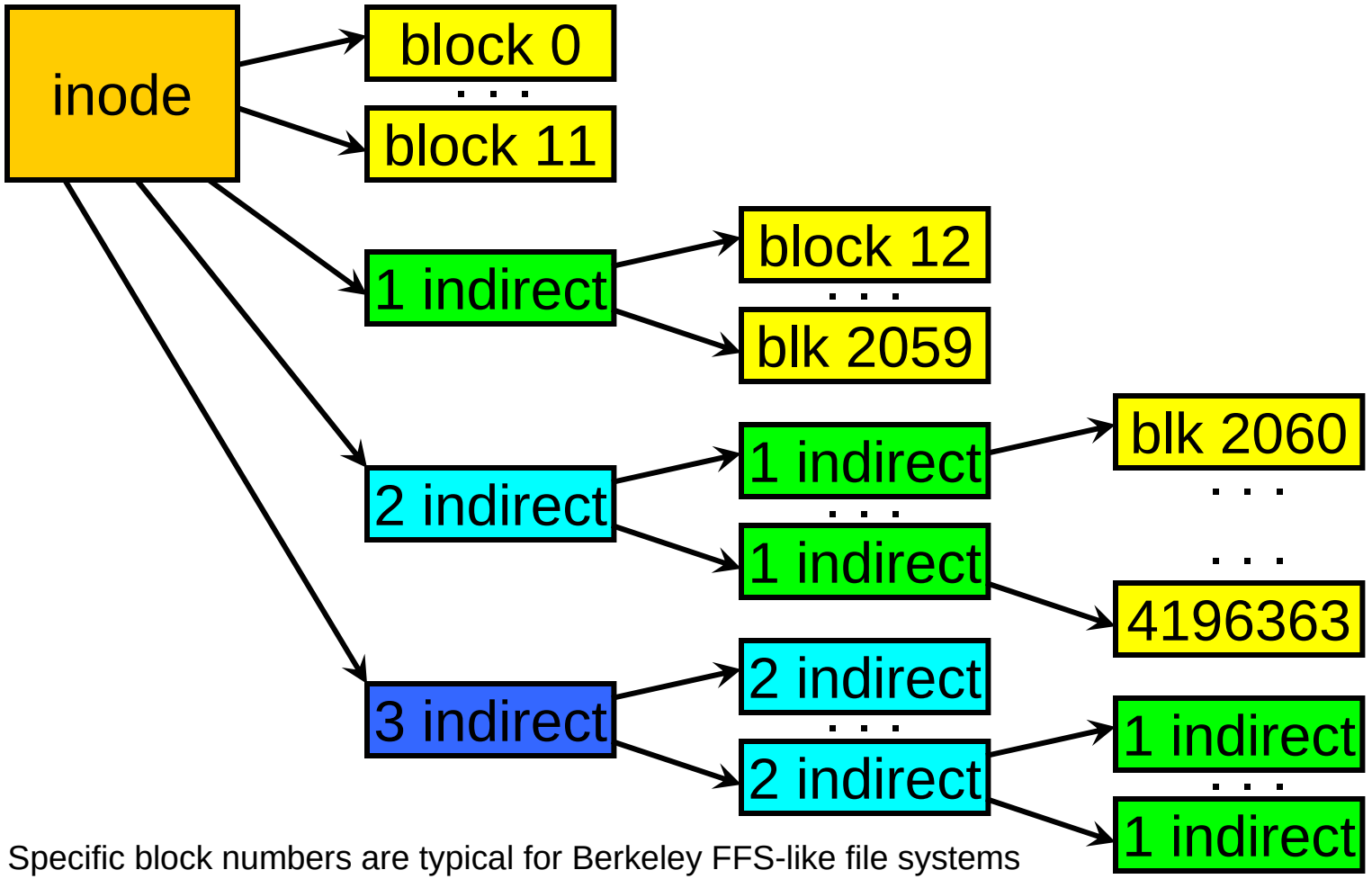
- UNIX file system architecture.
  - Features.
  - Gotchas (non-obvious consequences).
- Vulnerability case studies.
  - World-writable directories.
  - Race conditions.
  - Walking a hostile directory tree.
- Lessons learned.

UNIX file system architecture



Direct and indirect data blocks

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



Specific block numbers are typical for Berkeley FFS-like file systems

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

- 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

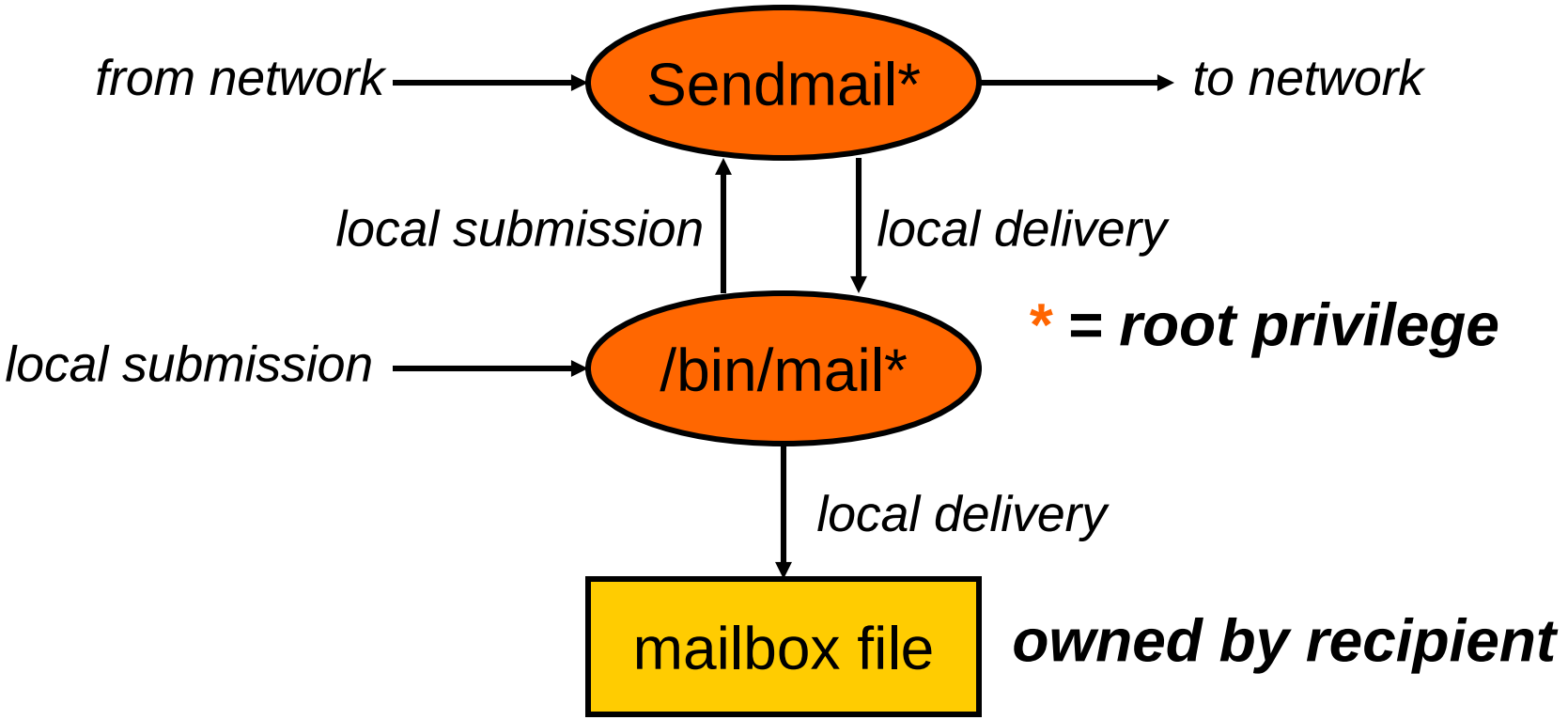
- Feature: access permission check happens on *open/execute* not *read/write*.
  - No general way to revoke access after a file is opened.
    - This also applies to non-file objects such as sockets.
  - Files must be created with correct permissions (as opposed to setting permissions after creating 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.

[1]Assuming that changing file ownership is a privileged operation.

[2]Historical Sendmail is privileged for other reasons (see part IV, Postfix).

/bin/mail delivery pseudocode

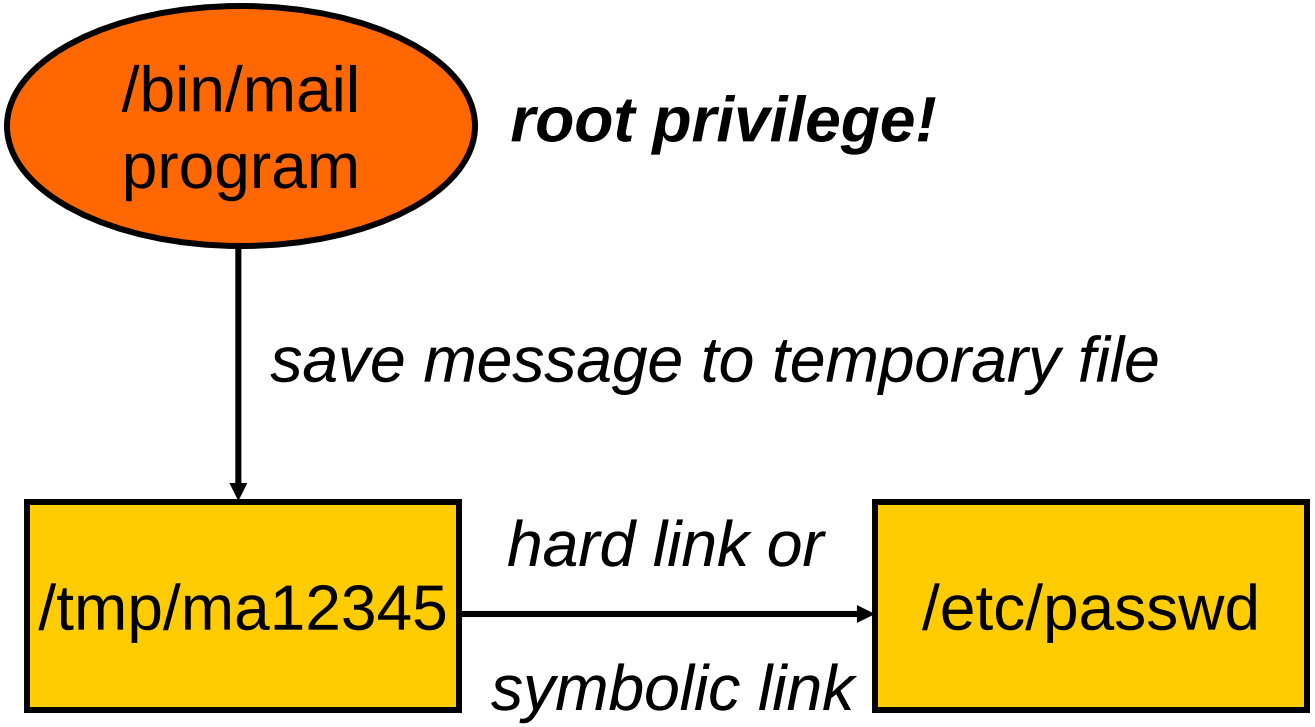
```
save message to temporary file
for each recipient
    lock recipient mailbox file
    append message to recipient mailbox file
    unlock recipient mailbox file
```

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

```
char    lettmp[ ] = "/tmp/maXXXXX";        /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?

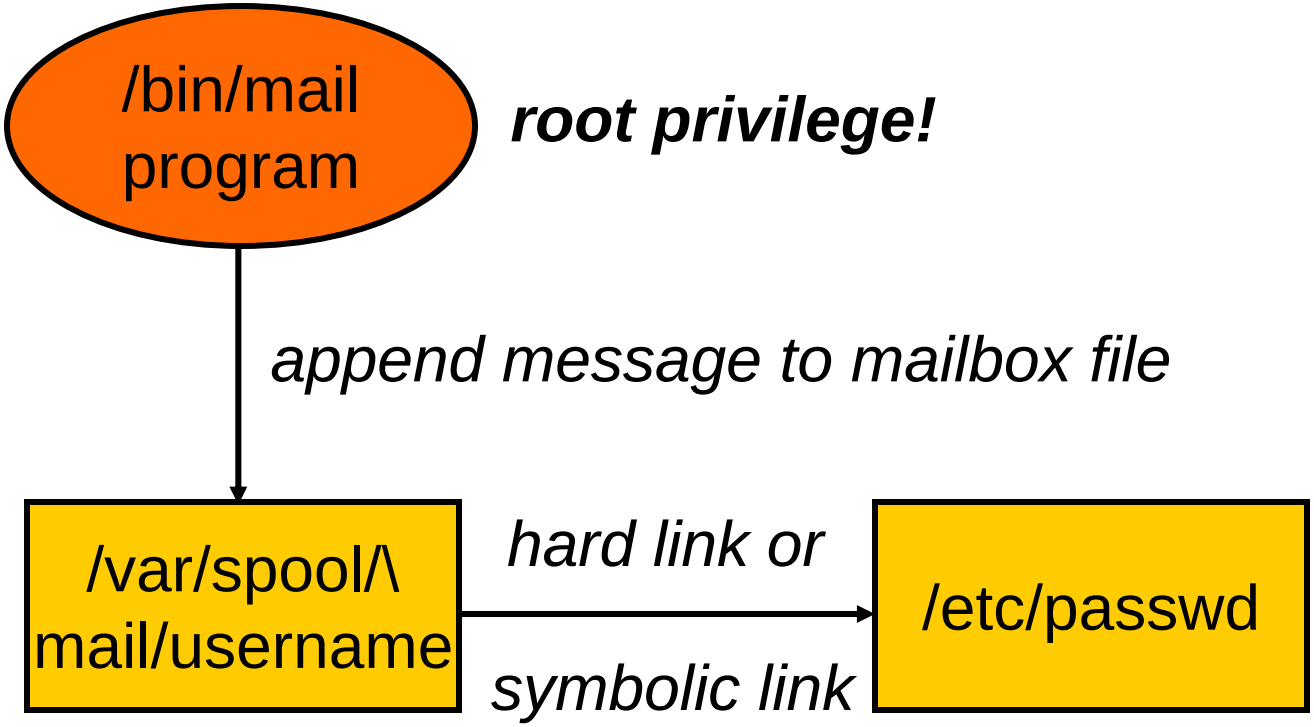


Step 2: append to mailbox file in world-writable directory

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

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 TCB[1] of a security system.
- The TCB may need updating whenever new software is installed on the system.

[1]TCB=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.

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 >= MAXPATHLEN.
  - Skip if pathname length >= MAXPATHLEN.
  - Pass the problem to the user of the result.
    - Use *chdir( )* to avoid system call failures within the tree walking program.
    - 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 1024[1].
- The UNIX *find* command changes directory *(chdir( ))* and leaves it to the user to handle long pathnames[2].
- Beware: changing directory can be dangerous when the directory tree is under control by an attacker.

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

[2]4.4BSD, 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.
- Experience teaches us to avoid what is broken and to build on the things that are good.

IBM Research

UNIX Setuid programming
Traps, pitfalls, and solutions

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

Overview

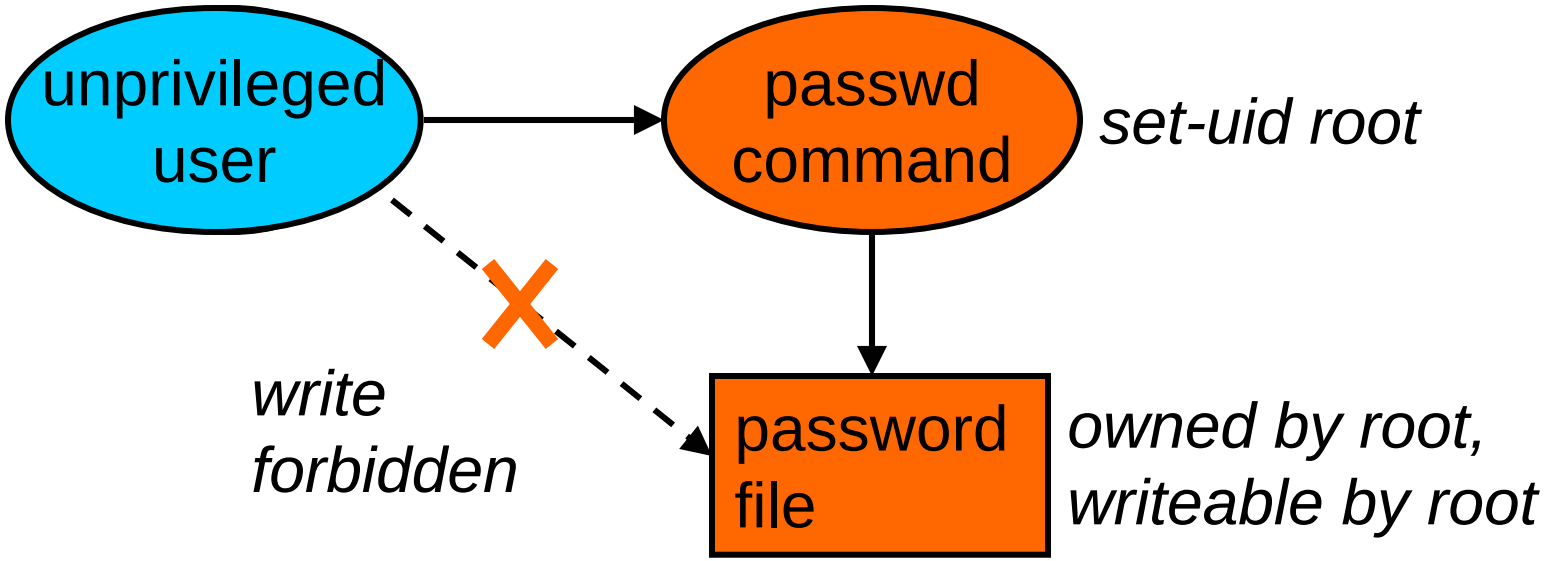
- The UNIX set-uid and set-gid mechanisms.
- Examples of vulnerabilities.
  - Inherited default file permissions.
  - Inherited process name.
  - Inherited open files.
  - Signal handlers.
- Bad and good alternatives.

Set-uid and set-gid in a nutshell

- Normally, a program file executes with the effective (user ID, group ID) of the process that invokes it[1].
- A set-uid (set-gid) file runs with the effective user ID (group ID) of the file owner[1]; i.e. with some or all of the owner's access privileges.
- Example: allow unprivileged users to update their own password file entry, without allowing them to update the password file directly.

[1]And with the auxiliary group IDs of the invoking process.

Set-uid example: controlled password file update



Getting privileges is easy, dropping them is hard

- Tricky to permanently drop set-uid privileges: different systems use different system calls[1]. With some old UNIX systems only set-uid root processes can permanently drop set-uid privileges.
- Similar problems exist with set-gid privileges.

[1]Hao Chen, David Wagner, Drew Dean, *Setuid Demystified*.

11th USENIX Security Symposium, San Francisco, 2002.

http://www.cs.berkeley.edu/~daw/papers/setuid-usenix02.pdf

Generic attacks on software (set-uid/gid or not)

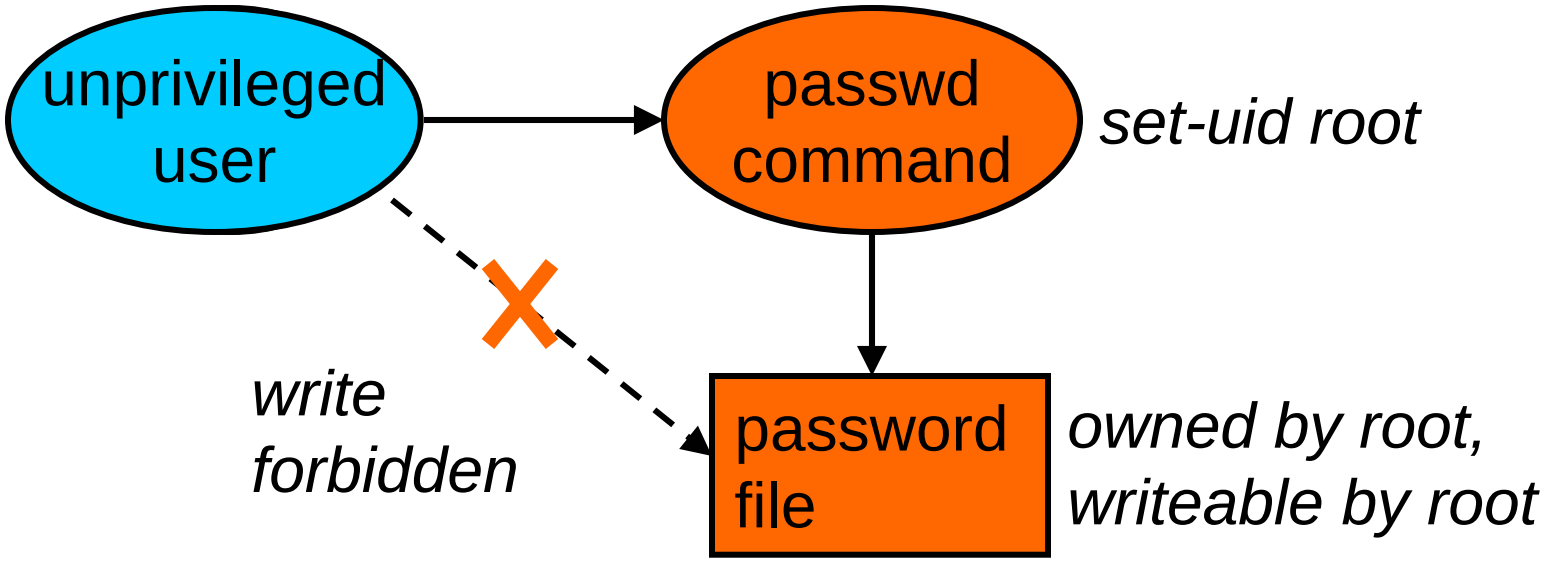
- Parsing errors
- Buffer overflows
- Race conditions
- Incorrect access permissions
- Weak authentication
- Trust without verification
- Resource starvation
- Timing attacks
- Poor encryption
- Poor key generation
- And so on...

Additional attack opportunities with set-uid/gid software due to process attribute inheritance

- command line + process name
- process environment
- open files (too many/too few)
- resource limits (file size etc.)
- umask (default file permission)
- process priority (race attacks)
- pending timers
- signal (enable/disable) mask
- current directory
- (root directory)
- child processes (!)
- POSIX session (signals)
- controlling terminal (signals)
- process group (signals)
- secondary group IDs
- (process ID)
- parent process ID
- attacks via /proc
- unsafe signal handlers (using the inherited real UID)

- **Set-uid case study: attacks via inherited default file access permissions**

Intended use: controlled password file update



UNIX passwd command purpose and implementation

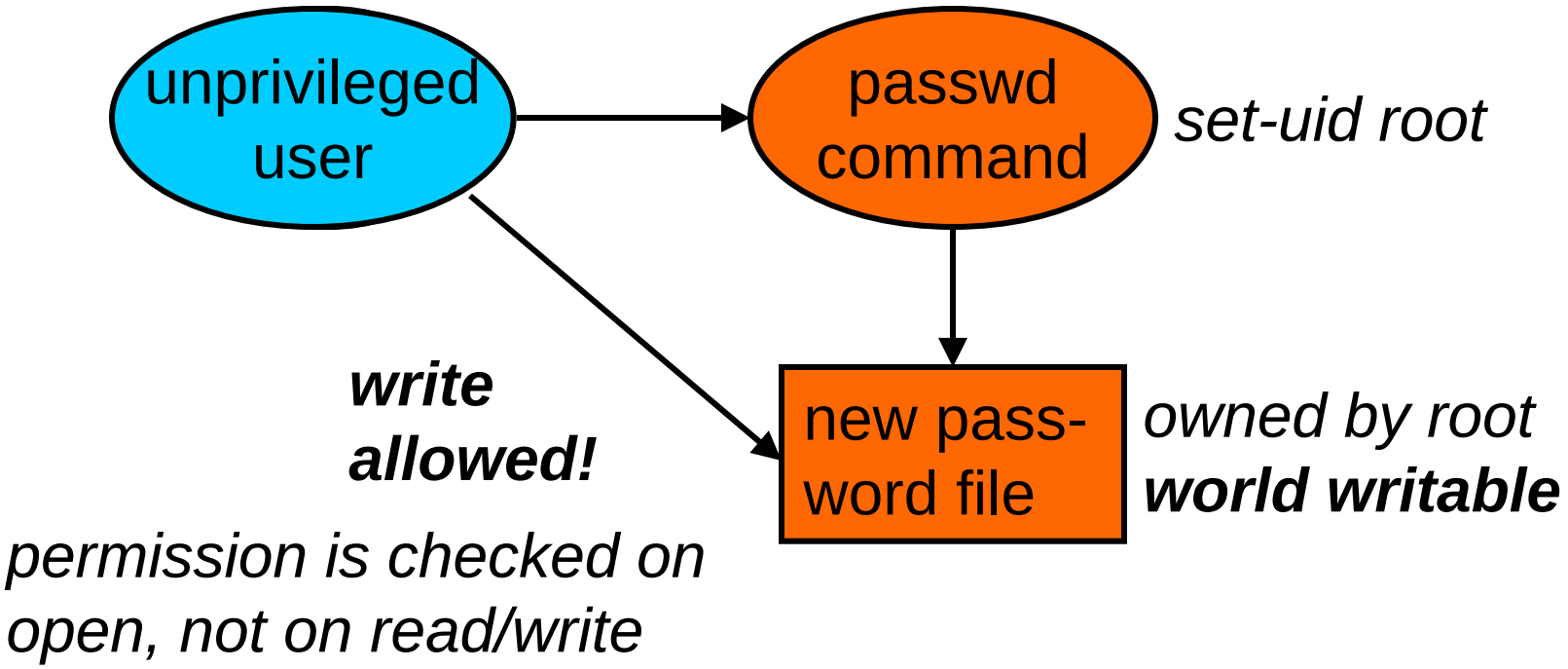
- Purpose: the *passwd* command is set-uid root, so that unprivileged users can make controlled changes to their own entry in the protected system password file.
- Pseudo code, ignoring file locking issues:
  - Sanity check the old and new passwords.
  - Copy current password file to new password file, replacing old password by new password.
  - Rename new password file to current password file.

Default access permission vulnerability

```
pwfile = fopen(SH_TMPFILE, "w");          create new password file
                                          with default access permissions
. . .other initialization. . .
chmod(SH_TMPFILE, 0600);                  forbid read/write access
                                          by “group” and “other”
. . .update the new password file. . .
if (fclose(pwfile))
    /* error... */
if (!err)
    rename(SH_TMPFILE, "/etc/shadow");    replace old password file
```

See: http://www.securityfocus.com/archive/1/138706 and follow-ups.

Default access permission exploit



Default access permissions exploit

- Invoke the *passwd* command with umask of zero. Optionally run the command at reduced priority.

```
$ umask 0                 default: world write permission
$ nice -20 passwd         make it run slower
```

- The *passwd* command will create the new password file with world write permissions:

```
pwfile = fopen(SH_TMPFILE, "w");  create rw-rw-rw- file
```

- Attack: open the new file for read/write access before the *passwd* command changes its access permissions.

Fixing the default access permissions vulnerability

- Reset default permissions to an appropriate value:

```
saved_mask = umask(077);    default: rw - - - - - - -
fp = fopen(pathname, “w”);
```

- Instead of *fopen(),* use a lower-level routine that creates the file with the right access permissions:

```
fd = open(pathname, O_CREAT | O_WRONLY, 0600);
fp = fdopen(fd, “w”);              see Postfix safe_open() function
```

- See also the “UNIX file system” segment on opening files in an untrusted directory.

- **Set-uid case study: attacks via inherited process name**

Process name attack intro

- Context: set-uid root security “access gate” programs log their executable file name for audit trail purposes.
  - Take the process name (the first command line element).
  - Search the PATH environment variable for a directory with a file that matches the process name.
  - Store the directory and file name in a buffer of MAXPATHLEN characters (typically, 1024 bytes).

Two process name exploits

- Both PATH and process name (first element of the command line) are under control by the attacker.

```
putenv("PATH=/bin:/usr/bin");          /* environment */
execl("/bin/su", "passwd", (char *) 0);   /* command line */
```

Executes */bin/su* with the process name *passwd*.

- The audit trail shows */bin/passwd* instead of */bin/su*.
- The process name can be up to 100kB-1MB long, overflowing the MAXPATHLEN pathname buffer.

Fixing the executable pathname lookup

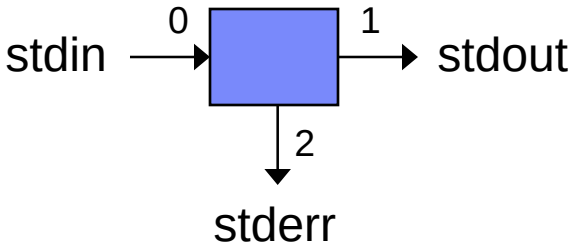
- On systems with the /proc pseudo file system:
  - FreeBSD: /proc/*pid*/file is symbolic link with the full pathname of the executable file[1].
  - Linux: /proc/*pid*/exe is symbolic link with the full pathname of the executable file.
  - Solaris: /proc/*pid*/as has the executable file name buried deep in the process address space (name can be displayed with, e.g., the *pmap* command)[2].

With too long pathname: [1]result = “unknown”; [2]result = relative pathname

- **Set-uid case study: attacks via inherited open files**

Standard open file environment

- Normally, each UNIX process runs with at least three open streams:
  - file number 0 (standard input).
  - file number 1 (standard output).
  - file number 2 (standard error)[1].

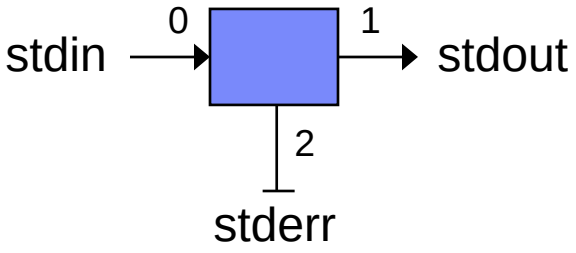


- Typically, all three streams are connected to the user's terminal.

[1]Error messages are written to file #2, so that they don't disappear when the standard output stream is directed to file or pipe.

What can go wrong?

- What happens if some of these streams are closed before the process is started? For example:
  - open: file number 0 (stdin).
  - open: file number 1 (stdout).
  - closed: file number 2 (stderr).



- The next file to be opened will be assigned file# 2.
- If this is the new password file, then user controlled error messages may end up in the password file!

TCP Wrapper / Postfix defense against open file problems

```
/*
 * To minimize confusion, make sure that the standard file descriptors
 * are open before opening anything else. XXX Work around for 44BSD
 * where fstat() can return EBADF on an open file descriptor.
 */
for (fd = 0; fd < 3; fd++)
    if (fstat(fd, &st) == -1
            && (close(fd), open("/dev/null", O_RDWR, 0)) != fd)
            msg_fatal("open /dev/null: %m");
```

Note: some BSD kernels force stdin/stdout/stderr to be open before process startup.

- Exploiting signal handlers in set-uid/gid software

Typical use of signal( ): clean up and terminate

```
void handler(int sig)
{
        printf("Interrupted!\n");          Unsafe!
        . . . clean up . . .
        exit(1);                           Unsafe!
}

int main(int argc, char **argv)
{
        signal(SIGINT, handler);           Control-C, or kill(2) call by process
        . . . normal processing. . .        with suitable real or effective UID
}
```

Signal handler exploits: memory corruption

```
void handler(int sig)
{
        printf("Interrupted!\n");          Unsafe!
        . . . clean up . . .
        exit(1);                           Unsafe!
}
```

- *printf( )* invokes *malloc( )*, which manages the heap.
- *printf( )* manages its own data structures and pointers.
- *exit( )* flushes standard I/O streams and invokes optional *atexit( )* application call-back routines.

Safe signal handler: set flag and do nothing else

```
void handler(int sig)
{
        got_signal = 1;
}
```

- Set a global variable in the signal handler, and examine the variable in the non-signal handler code.
- When setting a variable to implement some mutex, use a small enough data type such as *sig_atomic_t*. On a 32-bit machine, 64-bit updates are not atomic.

Safe signal handling - exit safely (from Postfix)

```
void handler(int sig)
{
        if (signal(SIGINT, SIG_IGN) != SIG_IGN) {       . . . only once . . .
                . . . clean up . . .
                _exit(1)                                . . . safe exit . . .
        }
}
```

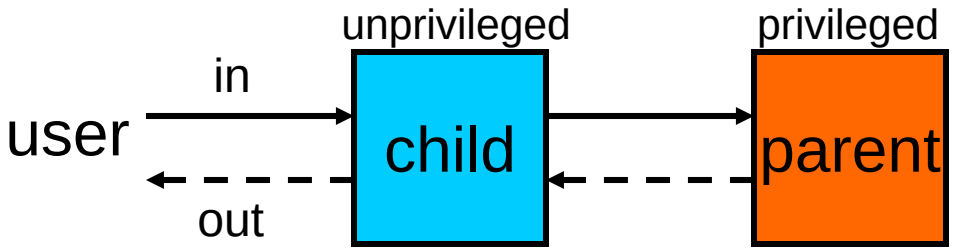
- *signal( )*, *sigaction( ), etc.* are atomic. The above code protects signal handlers against nested interrupts.
- *_exit( )* does not flush standard I/O streams and does not invoke *atexit( )* application call-back routines.

- Avoiding attacks on set-uid/gid commands

Bad alternative - parent/child

(from secure programming cookbook)

- The user invokes the privileged command.
- The command fork()s a child that drops privileges.
- All user input/output goes through the child process.

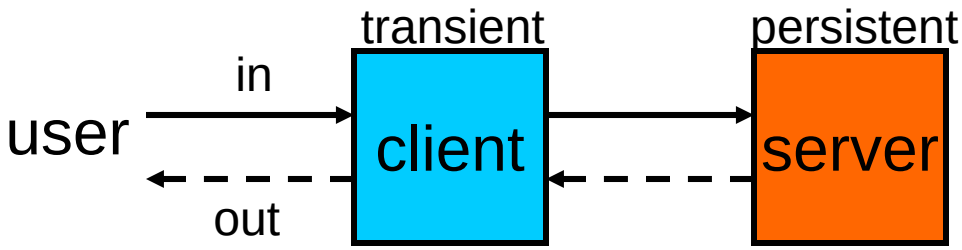


- Problem: the parent (and child) still inherit evil process attributes from the malicious user!

Good alternative - client/server

(Postfix MTA, version 1.0 and earlier)

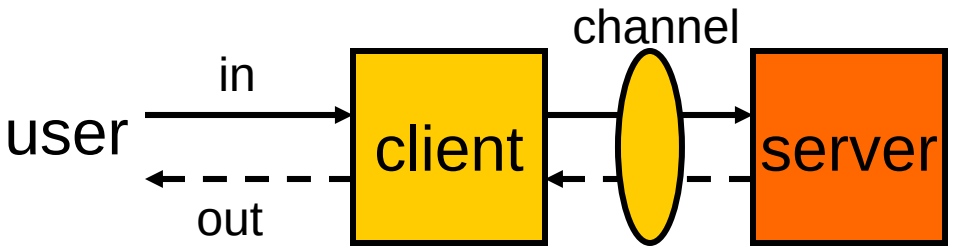
- Privileged process runs as persistent server.
- User process runs as transient client.



- The server inherits no process attributes from the user.
- Narrow protocol protects server against attack.

Better alternative: client/server, protected channel
(Postfix MTA, version 1.1 and later)

- Protected resource is managed by server process.
- Client-server channel requires group access privilege (ex: FIFO, UNIX-domain socket, or drop-off directory).
- Small set-gid client “guard” protects access to channel.



- Narrow protocol protects server against attack by compromised set-gid client “guard” process.

- **Final words**

Recap: extra attack opportunities with set-uid/gid commands due to process attribute inheritance

- command line + process name
- process environment
- open files (too many/too few)
- resource limits (file size etc.)
- umask (default file permission)
- process priority (race attacks)
- pending timers
- signal (enable/disable) mask
- current directory
- (root directory)
- child processes (!)
- POSIX session (signals)
- controlling terminal (signals)
- process group (signals)
- secondary group IDs
- (process ID)
- parent process ID
- attacks via /proc
- unsafe signal handlers (using the inherited real UID)

Set-uid/gid lessons learned

- Don’t use set-uid/set-gid unless absolutely necessary.
- If you think that set-uid is necessary, then you are probably mistaken.
- Instead of set-uid, try to use *set-gid* instead. Many access checks use the effective UID only. This limits the impact of group ID compromise.
- Defending against all *known* attack methods is not sufficient. New set-uid/set-gid attack opportunities arise as UNIX systems add new process attributes.

Setuid programming checklists and tips

- Garfinkel, Spafford, Schwartz: *Practical UNIX & Internet Security*. Chapter 16, *Secure Programming Techniques,* includes a checklist.
- Matt Bishop: *How to Write a Setuid Program*; *Robust Programming*; and other resources at: http://nob.cs.ucdavis.edu/~bishop/secprog/
- The BSDI setuid(7) manual page. http://www.homeport.org/~adam/setuid.7.html

IBM Research

The Postfix mail server
A secure programming example

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

Expectations before the first Postfix release...

[Postfix]: No experience yet, but I'd guess something like a wisened old man sitting on the porch outside the postoffice. Looks at everyone who passes by with deep suspicion, but turns out to be friendly and helpful once he realises you're not there to rob the place.

Article in alt.sysadmin.recovery

Overview

- Why write yet another UNIX mail system?
- Postfix architecture and implementation.
- Catching up on Sendmail, or how Postfix could grow 4x in size without becoming a bloated mess.
- The future of Postfix and other software as we know it.

- **Why (not) write yet another UNIX mail system**

New code, new bug opportunities

Code line counts for contemporary software:

- Windows/XP: 40 million; Vista 50+ million.
- Debian 2.2: 56 million; 3.1: 200+ million.
- Wietse's pre-Postfix average: 1 bug / 1000 lines[1].
- Postfix public release: 30k lines of opportunity[1,2].

[1]Not included: comment lines, or bugs found in development.

[2]Today: 97k lines of code.

CERT/CC UNIX mail advisories
(it's not just about Sendmail)

Bulletin	Software	Impact
CA-1988-01	Sendmail 5.58	run any command
CA-1990-01	SUN Sendmail	unknown
CA-1991-01	SUN /bin/mail	root shell
CA-1991-13	Ultrix /bin/mail	root shell
CA-1993-15	SUN Sendmail	write any file
CA-1993-16	Sendmail 8.6.3	run any command
CA-1994-12	Sendmail 8.6.7	root shell, r/w any file
CA-1995-02	/bin/mail	write any file

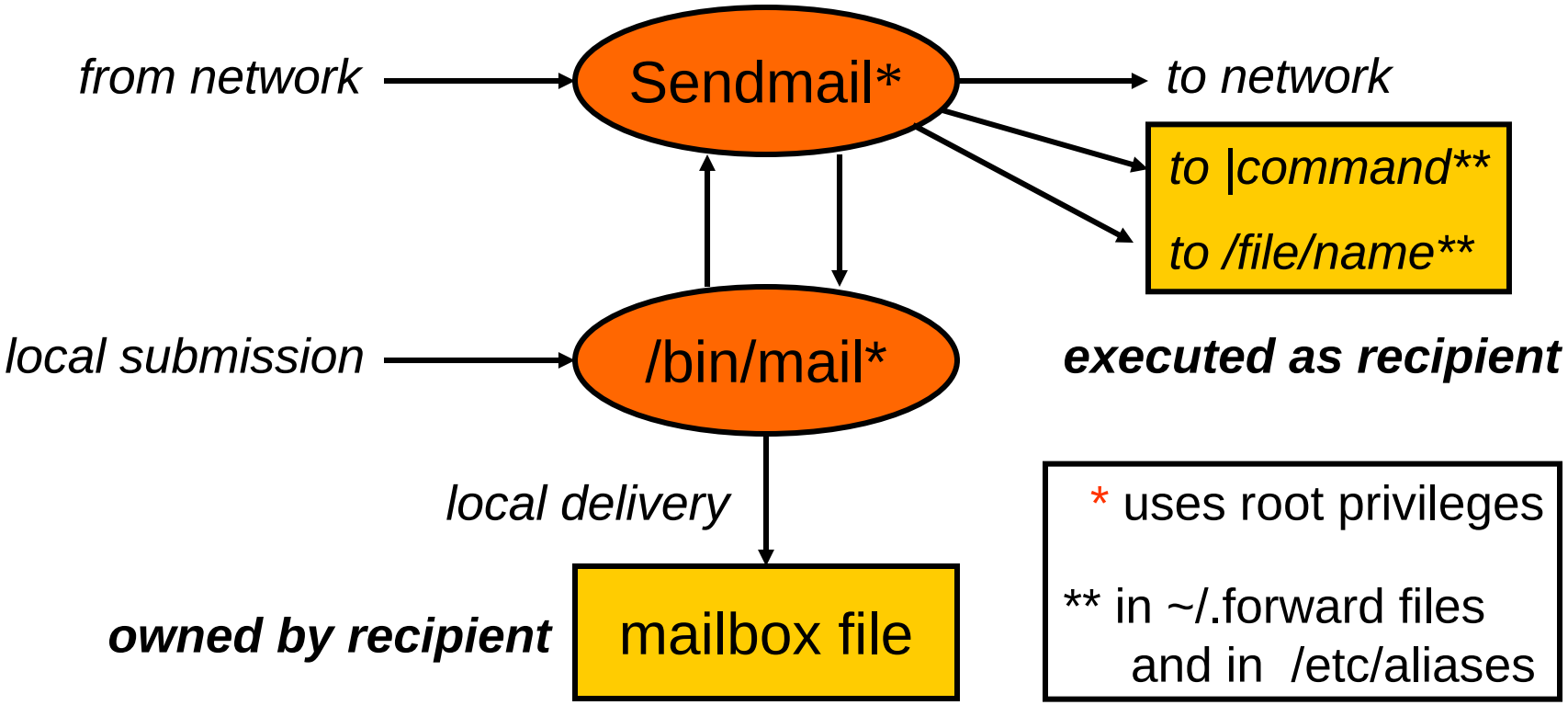
CERT/CC UNIX mail advisories

Bulletin	Software	Impact
CA-1995-05	Sendmail 8.6.9	any command, any file
CA-1995-08	Sendmail V5	any command, any file
CA-1995-11	SUN Sendmail	root shell
CA-1996-04	Sendmail 8.7.3	root shell
CA-1996-20	Sendmail 8.7.5	root shell, default uid
CA-1996-24	Sendmail 8.8.2	root shell
CA-1996-25	Sendmail 8.8.3	group id
CA-1997-05	Sendmail 8.8.4	root shell

CERT/CC UNIX mail advisories

Bulletin	Software	Impact
CA-2003-07	Sendmail 8.12.7	remote root privilege
CA-2003-12	Sendmail 8.12.8	remote root privilege
CA-2003-25	Sendmail 8.12.9	remote root privilege

Traditional UNIX mail delivery architecture



Root privileges in UNIX mail delivery

- Mailbox files are owned by individual users.
  - Therefore, */bin/mail* needs root privileges so that it can create / update user-owned mailbox files[1].
- "|*command*" and */file/name* destinations in aliases and in user-owned ~/.forward files.
  - Therefore, *sendmail* needs root privileges so that it can correctly impersonate recipients[2].

[1]Assuming that changing file ownership is a privileged operation.

[2]On UNIX systems, impersonation is always a privileged operation.

- **Postfix implementation - planning for failure**

Postfix primary goals
(It's not only about security)

- Compatibility: make transition easy.
- Wide deployment by giving it away.
- Performance: faster than the competition.
- Security: no root shells for random strangers.
- Flexibility: C is not an acceptable scripting language.
- Reliability: behave rationally under stress.
- Easy to configure: simple things should be easy.

Challenges: complexity

(How many balls can one juggle without dropping one)

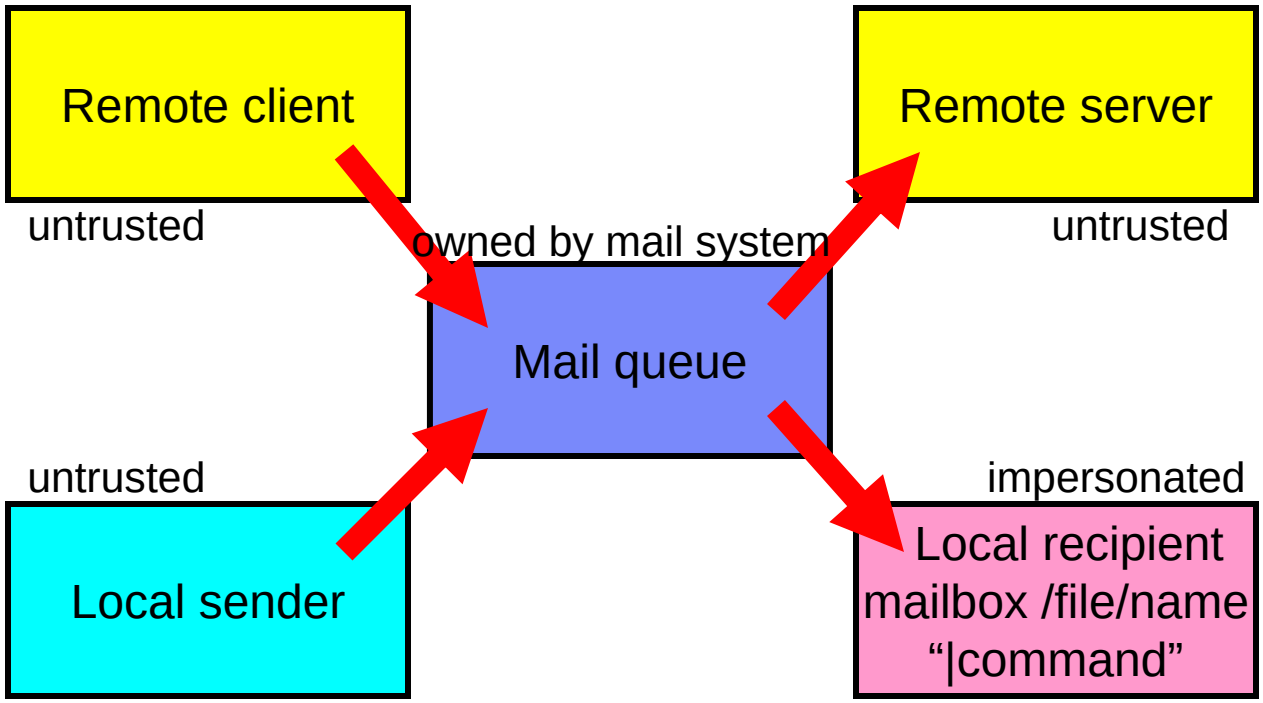
As we have learned, complexity != security.

- Multi-protocol: SMTP/DNS/TLS/LDAP/SQL/Milter.
- Broken implementations: clients, servers, proxies.
- Concurrent mailbox “database” access.
- Complex mail address syntax **<@x,@y:a%b@c>**.
- Queue management (thundering herd).
- SPAM and Virus control.
- Anti-spoofing: DKIM, SenderID, etc., etc.

Strategies: divide and conquer
(Juggle fewer balls, basically)

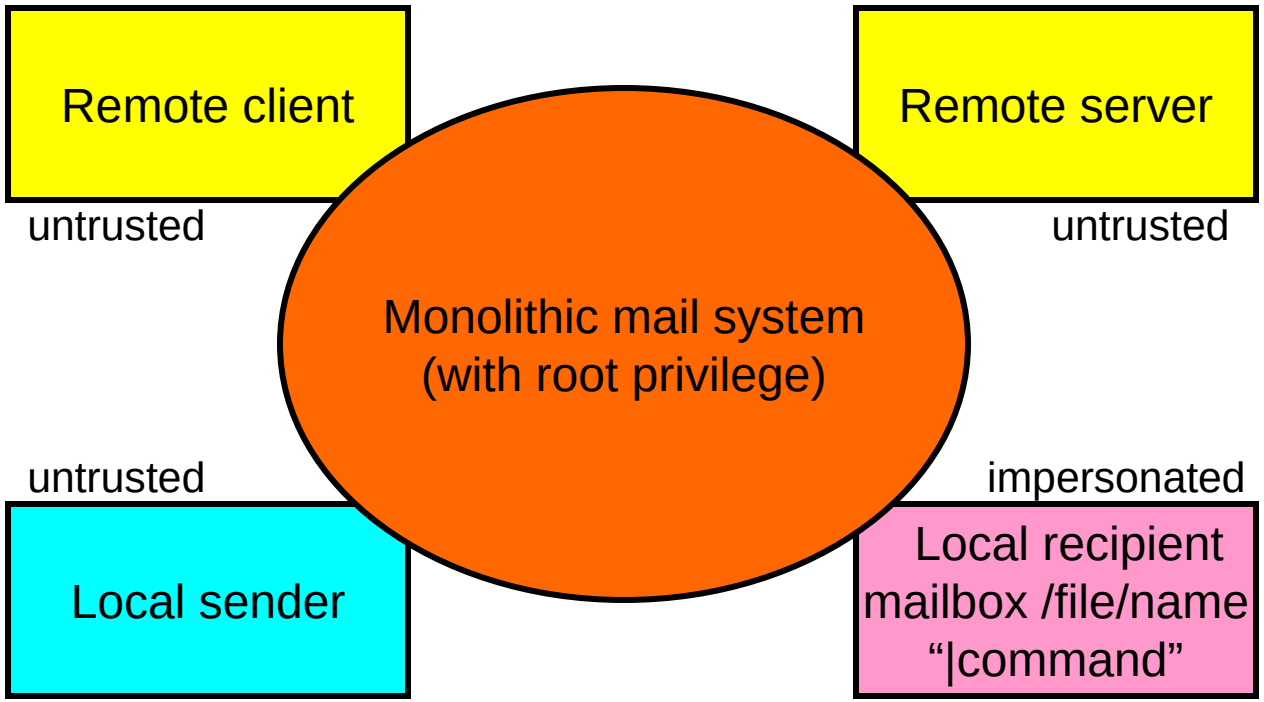
- Partitioned architecture (more on this next).
- More-or-less safe extension mechanisms:
  - Use SMTP or “pipe-to-command” for content inspection; let *other people* provide applications that do the work.
  - Simple SMTP access delegation protocol; let *other people* provide spf, greylist etc. applications.
  - Adopt Sendmail V8 Milter protocol; let *other people* provide anti-spoofing or content filter applications.
- More-or-less safe C programming API (example later).

UNIX mail systems cross (too) many privilege domains



Each arrow represents a privilege domain transition

Dangers of monolithic privileged MTAs: no damage control

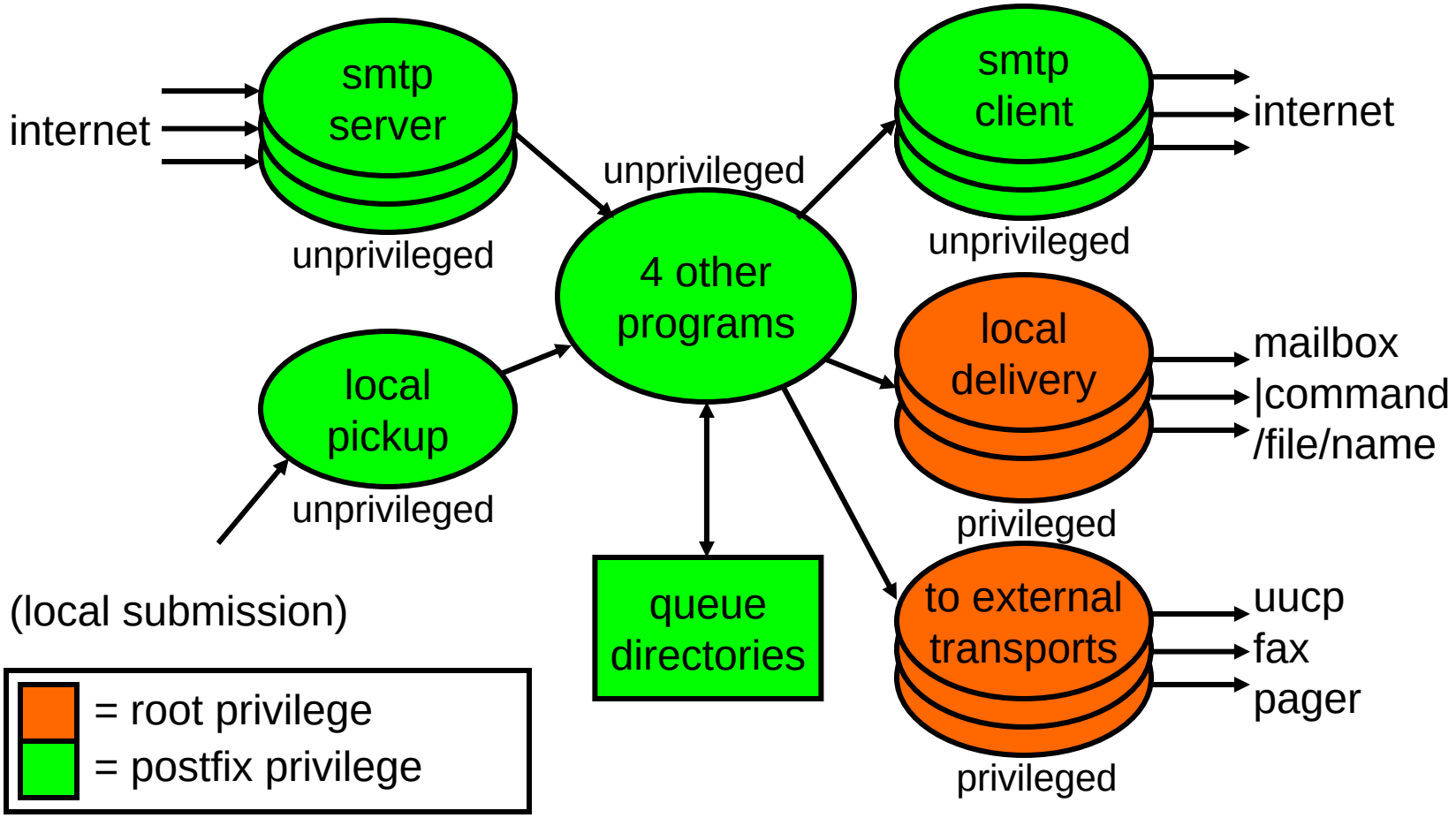


Dangers of monolithic privileged MTAs: no damage control

- One program touches all privilege domains.
  - Make one mistake and any remote client can execute any command, or read/write any file - with root privilege.
- No internal barriers:
  - Very convenient to implement.
  - Very convenient to break into.
- Postfix architecture prepares for failure, using multiple safety nets.

Postfix service-based architecture

(not shown: local submission, lmtp and qmqp protocols)



Postfix security principles

- Compartmentalize. Use one separate program per privilege domain boundary[1].
- Minimize privilege. Use system privilege only in programs that need to impersonate users. Many unprivileged daemons can run inside a *chroot()* jail.
- Do not trust queue file or IPC message content for sensitive decisions (e.g.: impersonation of recipients; command execution).
- Multi-layer defense of safety nets and sanity checks.

[1]Hidden privilege domain boundaries: DNS, LDAP, SQL, NIS, Netinfo, etc.

Low-level example - avoiding buffer overflow vulnerabilities

- 80-Column punch cards became obsolete years ago.
- Fixed-size buffers always have the wrong size: they are either too small, or too large.
- Exploit: “specially-crafted” input overwrites function call return address, function pointer, or some other critical information.
- Dynamic buffers are only part of the solution: they introduce new problems of their own.

Memory exhaustion attacks

- IBM web server: never-ending request.

  `forever { send “XXXXXX...” }`

- qmail 1.03 on contemporary platforms.
  - Never-ending request:

    `forever { send “XXXXXX....” }`

  - Never-ending recipient list:

    `forever { send “RCPT TO <address>\r\n” }`

- Impact: exhaust all virtual memory on the system; possibly crash other processes.

Dynamic buffers with safety nets

- Upper bounds on the sizes of object instances.
  - With SMTP, 2048-character input lines are sufficient. In other words, Postfix simulates larger punch cards.
- Upper bounds on the number of object instances.
- Plus some special handling for large items.
  - Limit the total length of each individual multi-line message header line (*To:*, *Received:,* etc.).
  - Don’t limit the length of message body lines; process them as chunks of 2048 bytes, one chunk at a time.

- **Catching up on Sendmail - the benefits of a security architecture**

Catching up on Sendmail

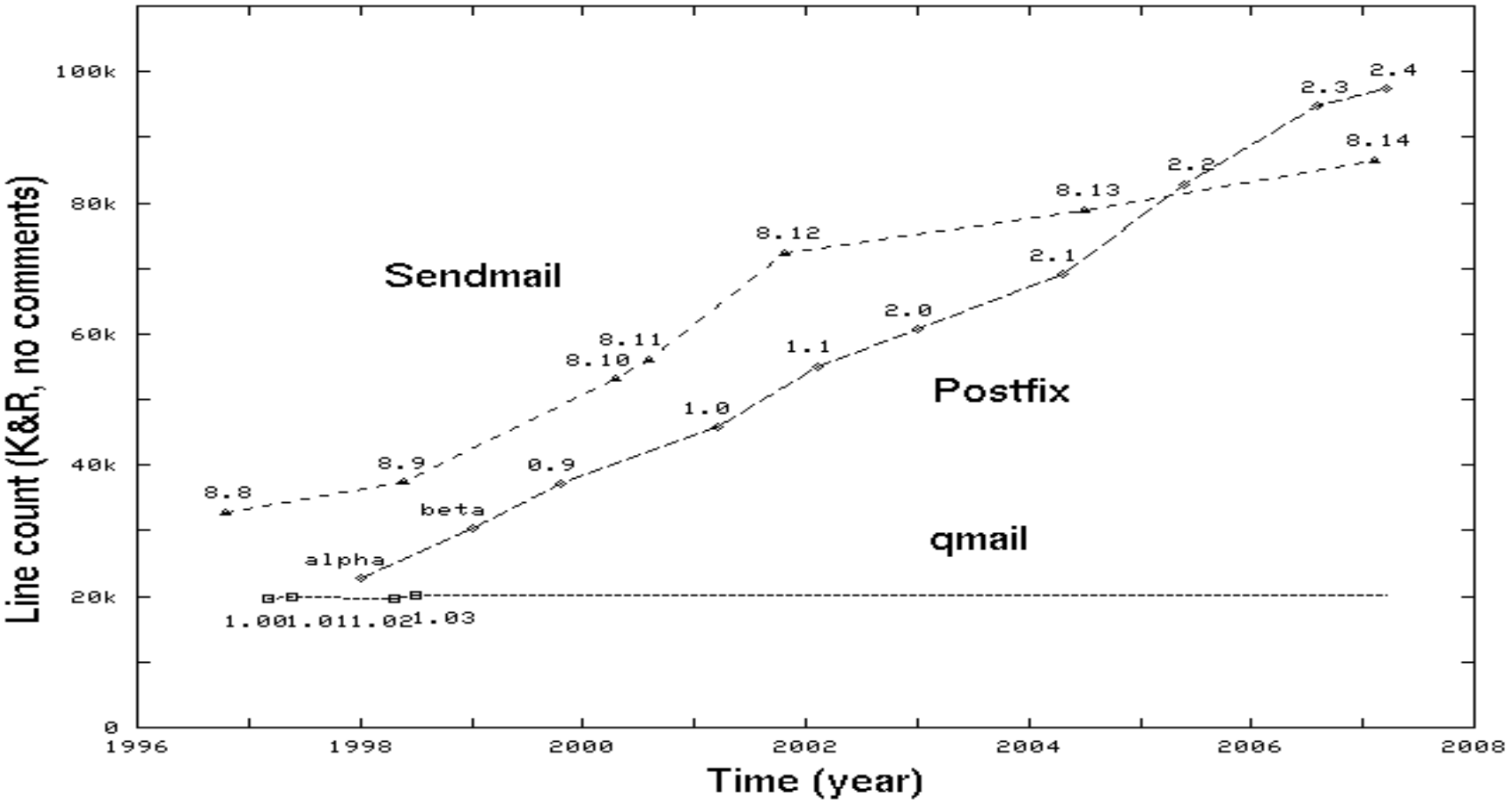
- How Postfix has grown in size, from a qmail[1]-like subset to a complete mail server.
- Where did all that code go?
- Why Postfix could grow 4x in size without becoming a bloated mess.
- Why writing Postfix code is like pregnancy.

[1]A direct competitor at the time of the first Postfix release.

How Postfix has grown in size

- Initial trigger: the Postfix 2.2 source tar/zip file was *larger* than the Sendmail 8.13 tar/zip file.
- Analyze eight years of Sendmail, Postfix, and qmail source code:
  - Strip comments (*shrinking* Postfix by 45% :-).
  - Format into the “Kernighan and Ritchie C” coding style (*expanding* qmail by 25% :-).
  - Delete repeating (mostly empty) lines.

MTA Source lines versus time



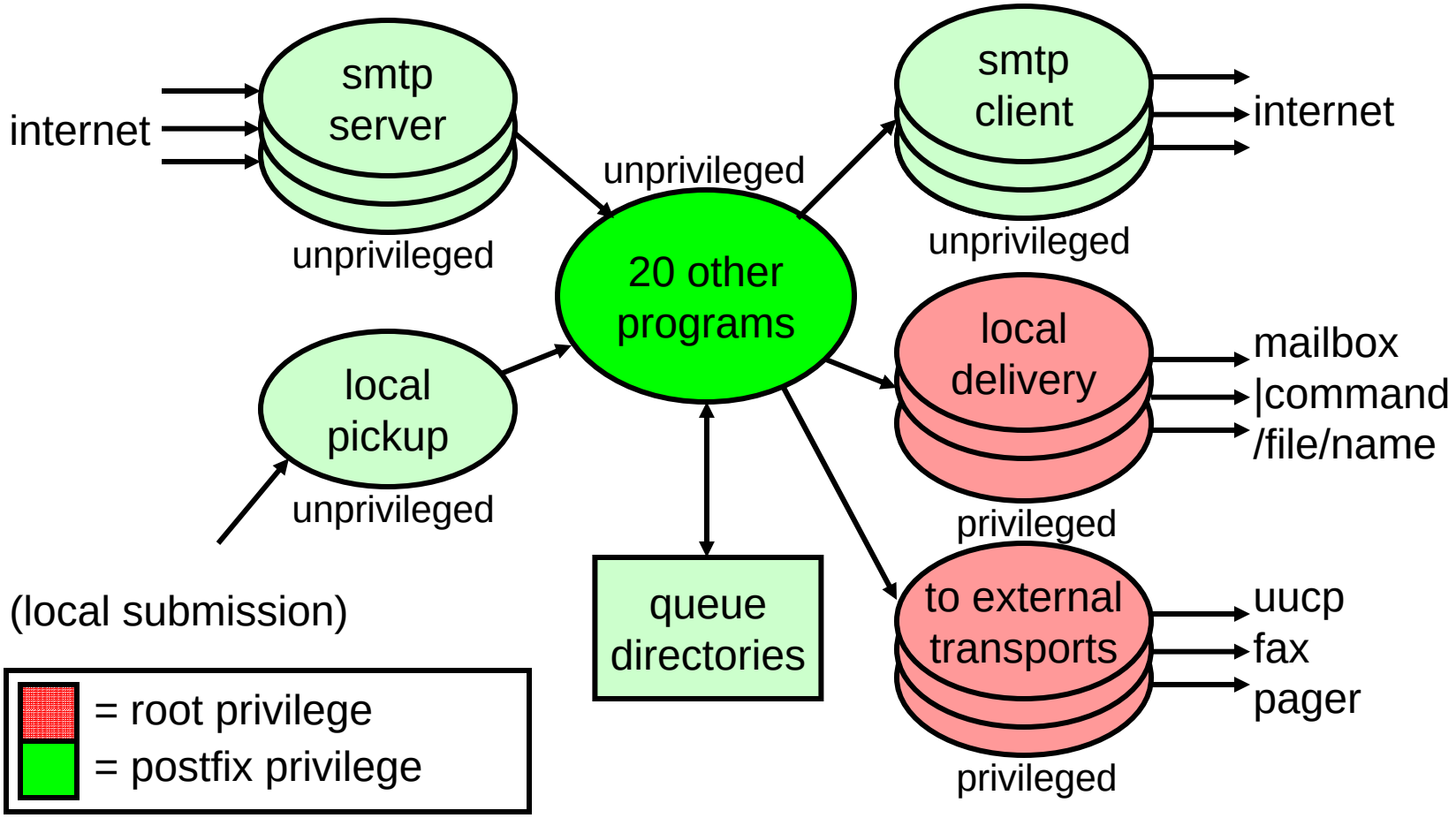
Where did all that code go?
(from Postfix alpha to Postfix 2.3)

- 4x Growth in size, 8400 lines/year, mostly same author.
- Small increase:
  - 1.3x Average program size (800 to 1100 lines).
- Medium increase:
  - 2.5x Program count (from 15 to 36).
- Large increase:
  - 4x Library code (from 13000 to 52000 lines).
- No increase: number of privileged programs.

Why Postfix could grow 4x and not become a bloated mess

- Typically a major Postfix feature is implemented by a new *server* process and a small amount of *client* code. Recent examples of servers:
  - flush(8) on-demand delivery cache.
  - scache(8) outbound connection cache.
  - tlsmgr(8) TLS session key and random number cache.
  - verify(8) email address verification probes and cache.
- This is not a coincidence. It is a benefit of the Postfix security architecture.

Postfix service-based architecture



Good news: the Postfix security architecture preserves integrity

- Normally, adding code to an already complex system makes it even more complex.
  - New code has unexpected interactions with already existing code, thus reducing over-all system integrity.
- The Postfix architecture *encourages* separation of functions into different, untrusting, processes.
  - Each new major Postfix feature is implemented as a separate server with its own simple protocol.
  - This separation minimizes interactions with already existing code, thus preserving system integrity.

Bad news: writing major Postfix feature is like pregnancy

- *Time*: throwing more people at the problem will not produce a faster result.
  - The typical time to complete a major feature is limited to 1-2 months. If it takes longer it gets snowed under by later developments. Postfix evolves in Internet time.
- *Size*: the result can have only a limited size.
  - With Postfix, a typical major feature takes about 1000 lines of code, which is close to the *average* size of a command or daemon program.

- **Conclusions and resources**

Future of Postfix

- Postfix >=2.3 is complete enough that I am no longer embarassed to recommend it to other people.
  - Built-in: TLS, SASL, MIME, IPv6, LDAP, SQL, DSN (Delivery Status Notification: *success*, *failed*, *etc*).
- Further extension via plug-in interfaces.
  - Domain Keys, DKIM, SenderID, SPF.
  - Non-Cyrus SASL authentication, content inspection.
  - Sendmail Milter applications, SMTP server access policy.
- Clean up internals, logging, hard-coded behavior.

Postfix author receives Sendmail Milter innovation award

MOUNTAIN VIEW, Calif. October 25th, 2006 Today at its 25 Years of Internet Mail celebration event, taking place at the Computer History Museum in Mountain View, California, Sendmail, Inc., the leading global provider of trusted messaging, announced the recipients of its inaugural Innovation Awards.

. . .

Wietse Venema, author, for his contribution of extending Milter functionality to the Postfix MTA.

http://www.sendmail.com/pdfs/pressreleases/Sendmail%20Innovation%20Awards_10%2025%2006_FINAL.pdf

Future of software as we know it

- It is becoming less and less likely that someone will write *from scratch* another full-featured
  - Postfix or Sendmail like MTA (100 kloc).
  - BSD/LINUX kernel (2-4 Mloc).
  - Web browser (Firefox: 2 Mloc),
  - Window system (X Windows: 2 Mloc).
  - Desktop suite (OpenOffice: 5 Mloc)
  - etc.
- Creationism loses, Evolutionism and ID rules:-)

Postfix Pointers

- The Postfix website at http://www.postfix.org/
- Richard Blum, *Postfix* (2001).
- Kyle Dent, *Postfix The Definitive Guide* (2003).
- Peer Heinlein, *Das Postfix Buch*, 2nd ed (2004).
- Ralf Hildebrandt, Patrick Koetter, *The Book of Postfix* (2005).
- Books or translations in Japanese, Chinese, Czech, other languages.