The Postfix mail server as 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.

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: 95k 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

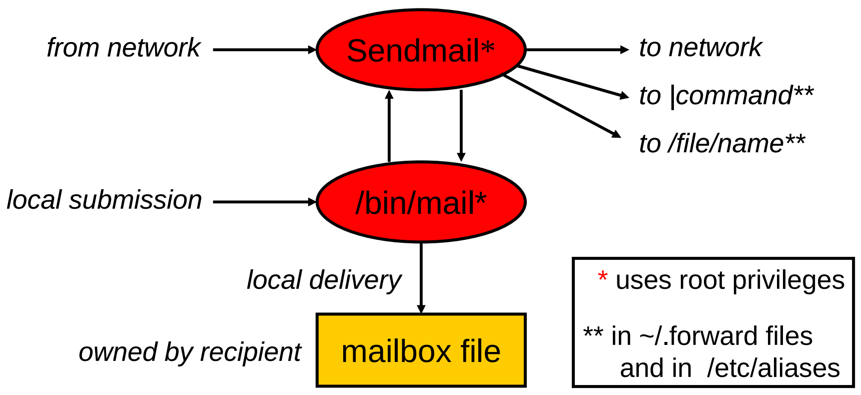
CERT/CC UNIX mail advisories

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.

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

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.

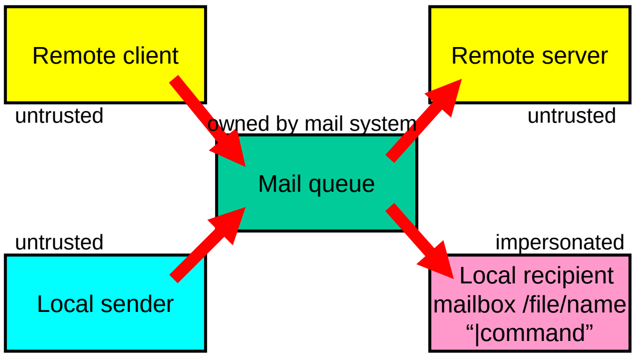
Challenges: complexity

(How many balls can one juggle without messing up)

- Multi-protocol: SMTP, DNS, 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.

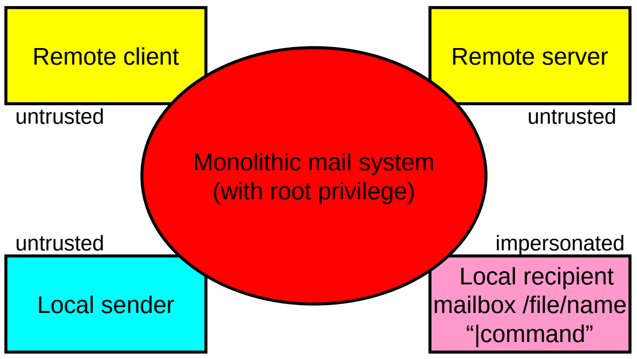
And as we have learned, complexity and security do not go together well.

UNIX mail systems cross (too) many privilege domains



Each arrow represents a privilege domain transition

Dangers of monolithic privileged MTAs: no damage control

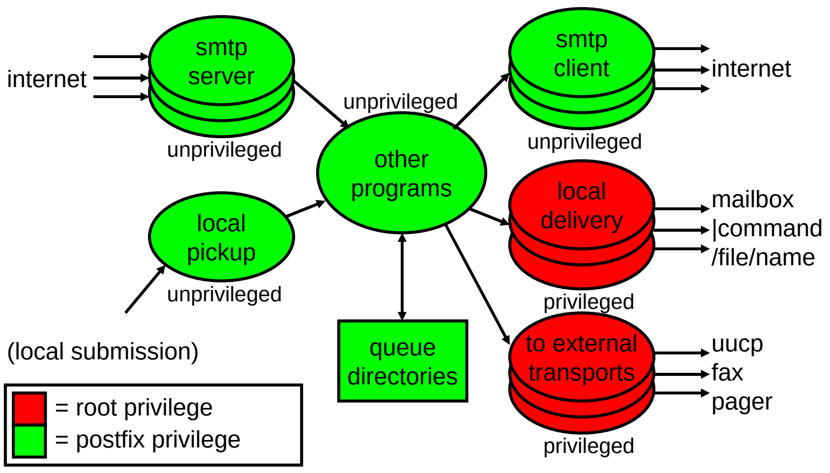

Remote client
untrusted
Remote server
untrusted
Monolithic mail system (with root privilege)
untrusted
Local sender
impersonated
Local recipient mailbox /file/name "|command"

Dangers of monolithic privileged MTAs: no damage control

- One program touches all privilege domains.
  - Make one mistake and a remote client can execute any command, or can read and write any file - with root privilege.
- No internal barriers:
  - Very convenient to implement.
  - Very convenient to break into.

Postfix service-based architecture

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



Main Postfix security guiding 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 files and IPC messages for security sensitive decisions (like: impersonation of recipients).

[1]Hidden privilege domain boundaries may result from interactions with DNS, LDAP, MySQL, PostgreSQL, NIS, NETINFO, etc.

Low-level example - avoiding buffer overflow vulnerabilities

- 80-Column punch cards went out of fashion years ago.
- Fixed-size buffers often have the wrong size: they are either too small, or too large.
- “specially-crafted” input overwrites function call return address, function pointer, or other critical information.
- Dynamic buffers are only part of the solution, because 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. Basically, Postfix uses larger punch cards.
- Upper bounds on the number of object instances.
- Plus some special handling for large items.
  - Limit the total length of multi-line message header lines (To:, Received: etc.).
  - Don’t limit the length of message body lines; process them as chunks of 2048 characters, one at a time.

Catching up on Sendmail

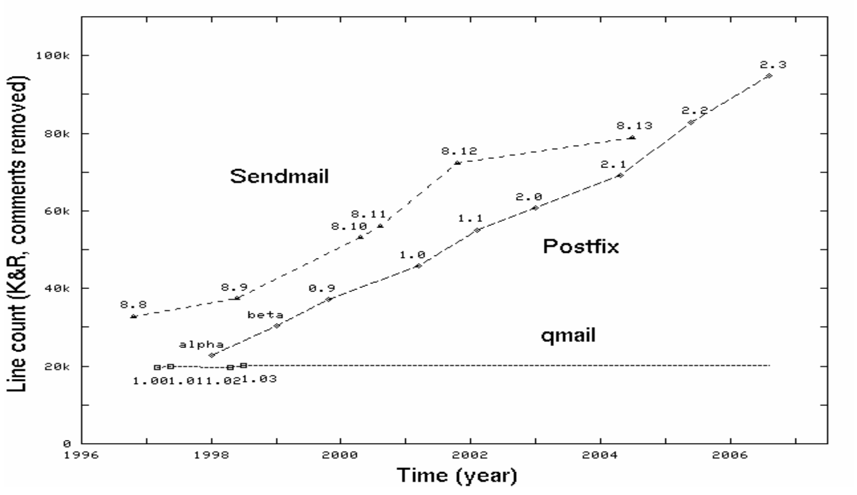
- How Postfix has grown in size, from a qmail[1]-like subset to a complete mail server.
- Where did all that code go?
- Why Postfix could grow 4x in size without becoming a bloated mess.
- Why writing Postfix code is like pregnancy.

[1]A direct competitor at the time of first 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 (*reducing* Postfix by 45% :-).
  - Format into the "Kernighan and Ritchie C" coding style (*expanding* qmail by 25% :-).
  - Delete repeating (mostly empty) lines.

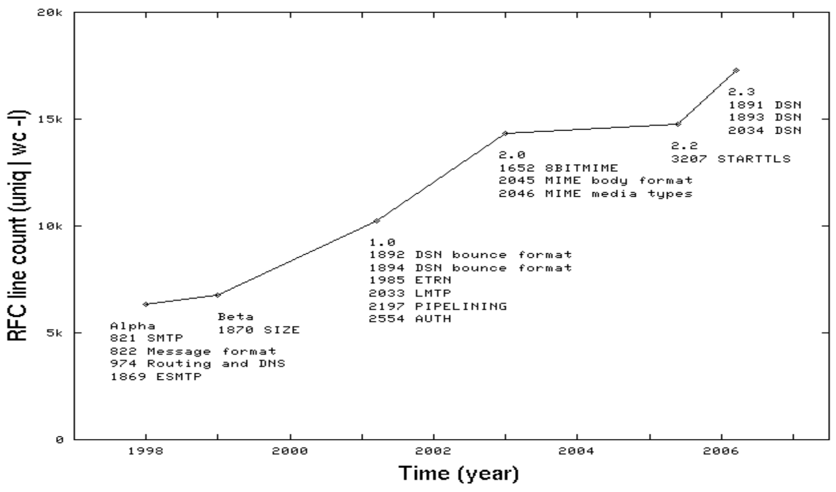
MTA Source lines versus time



Where did all that code go?
(Lies, damned lies, and statistics)

- 4x Growth in size, 8400 lines a year, 23 lines each calendar day, most but not all by the same person.
- Small increase:
  - 1.3x Average program size (800 to 1100 lines).
- Large increase:
  - 4x Library code (from 13000 to 52000 lines).
  - 2.5x Command/daemon count (from 15 to 36).
- No increase: number of system privileged programs.

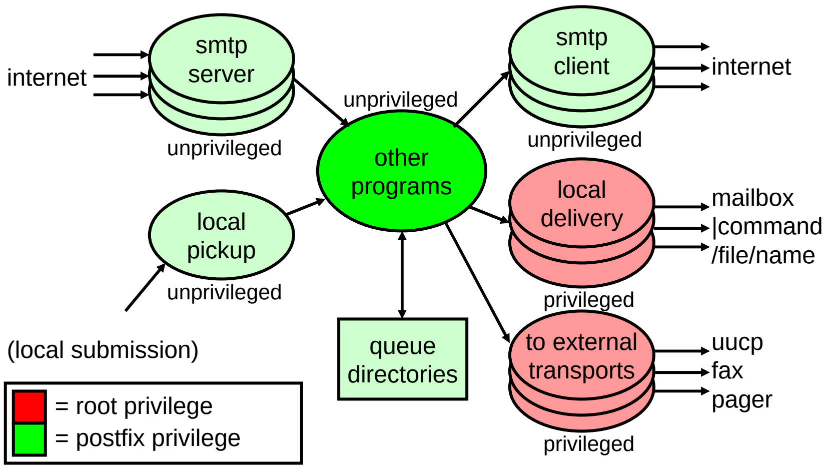
Postfix RFC lines versus time



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:
  - flush(8) controls on demand delivery.
  - tlsmgr(8) controls the TLS(SSL) session key cache.
  - verify(8) controls email address verification probes.
  - anvil(8) controls inbound connection/rate limits.
  - scache(8) controls outbound connection cache.
- This is not a coincidence. It is a benefit of the Postfix 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.
  - Implementing each new major Postfix feature with a separate program minimizes interactions with already existing code, thus preserving over-all system integrity.

Bad news: writing major Postfix feature is like pregnancy

- *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.
- *Time*: throwing more people at the problem will not get you 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.

Conclusions and Resources

Lessons learned: UNIX/C

- Neither UNIX nor C were designed with security as a major goal. Implementing “secure” software in such an environment is an exercise in:
  - Eliminating the many unsafe mechanisms.
  - Hardening the few remaining mechanisms.
- Regardless of environment, UNIX, Win32, JAVA:
  - Be liberal with sanity checks and safety nets.
  - Be prepared for the unexpected. Never assume.

Future of software as we know it

- It is becoming less and less likely that someone will write another full-featured Postfix or Sendmail MTA *from scratch* (100 kloc).
- It is becoming even less likely that someone will write another full-featured BSD or LINUX kernel *from scratch* (2-4 Mloc).
- ..or a full-featured web browser (Firefox: 2 Mloc),
- ..or another window system (X Windows: 2 Mloc).
- ..or a desktop suite (OpenOffice: 5 Mloc), etc.
- Creationism versus evolutionism.

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 in Japanese, Chinese, other languages.

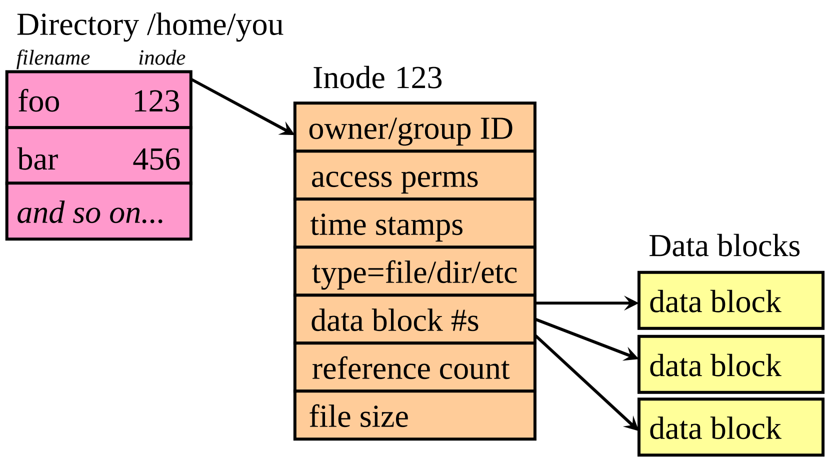
Secure Programming Traps and Pitfalls – The Broken File Shredder

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

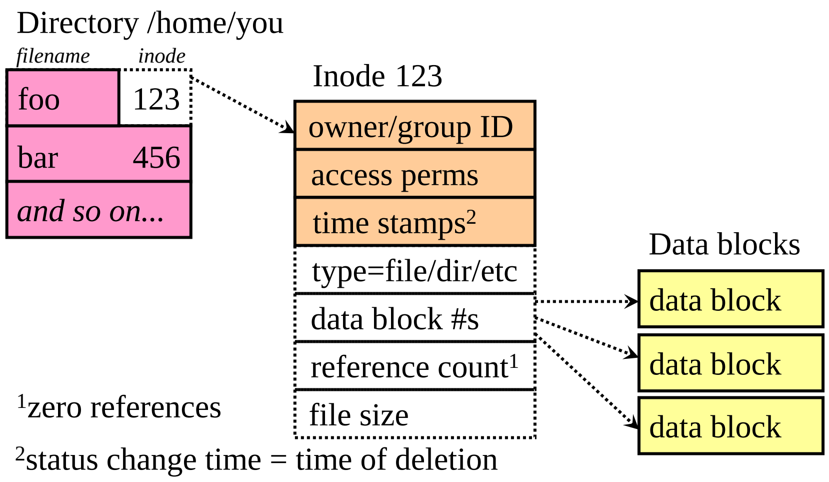
Overview

- 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

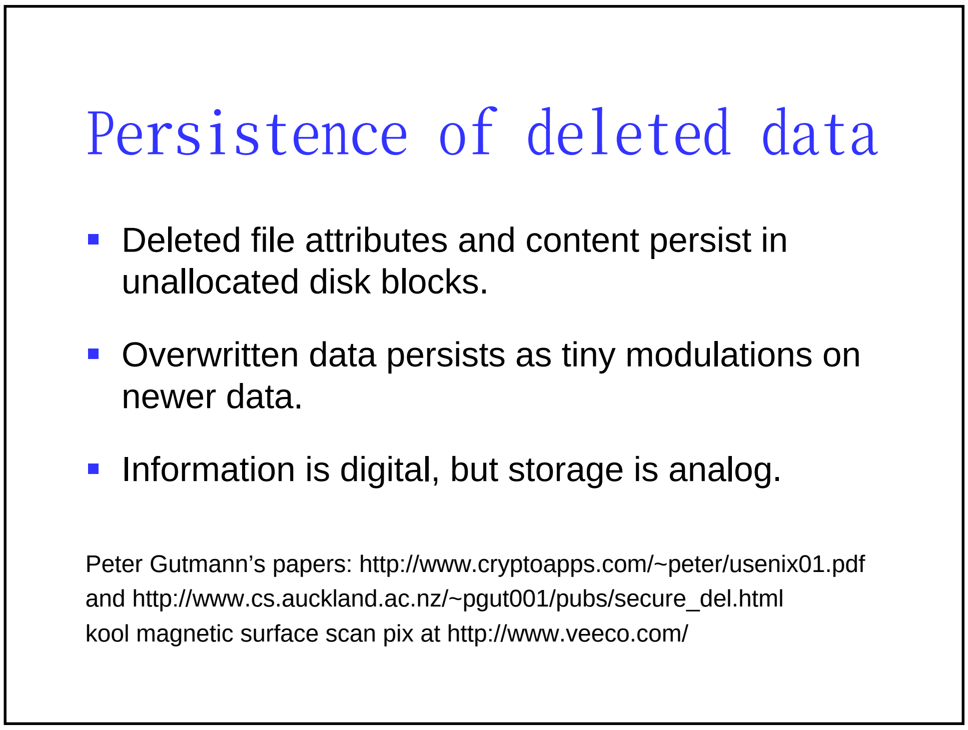


Deleting a (UNIX) file destroys structure not content



[1]zero references

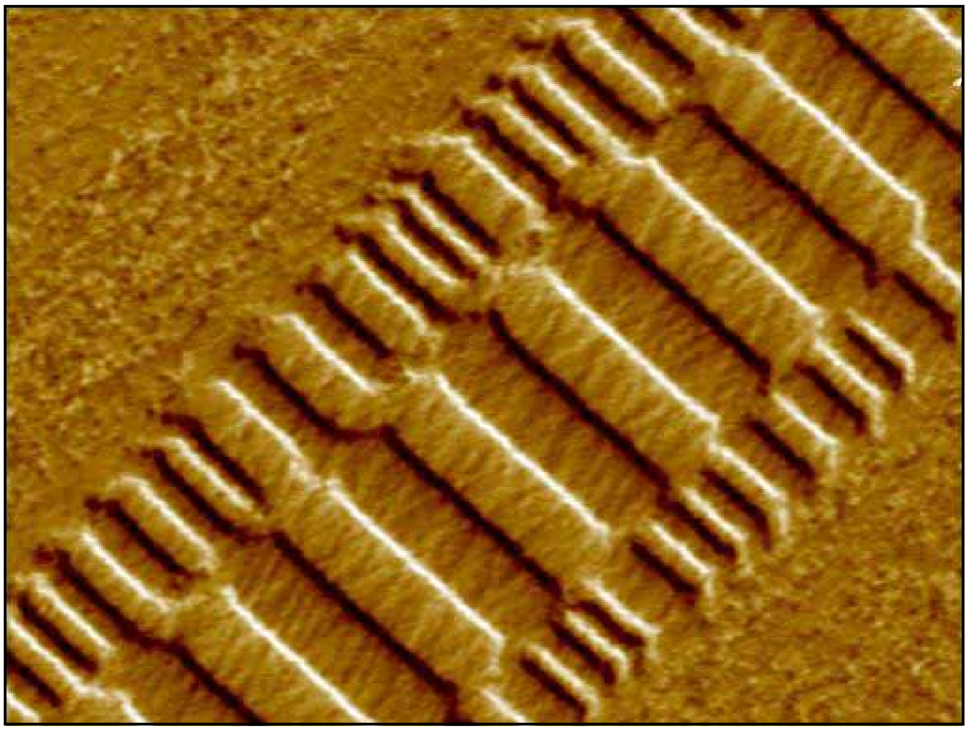
[2]status change time = time of deletion



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 from magnetic media

- Erase sensitive data before deleting it.
- To erase, 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 with pattern
remove file
```

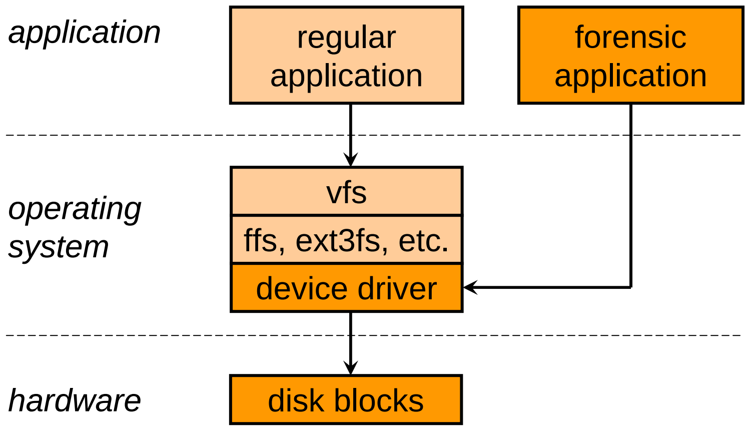
File shredder code, paraphrased

```
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



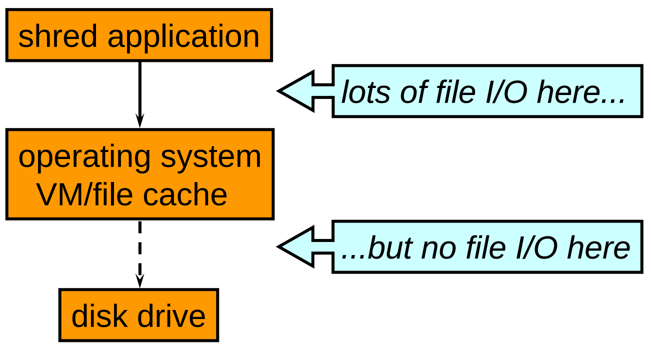
Coroner's Toolkit discovery

(Note: details are specific to RedHat 6 Ext2fs)

```
[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 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 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+”)) == NULL)      open for update
        /* 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 and instead create multiple copies of data.
- Not shredded: temporary copies from text editors, printer spoolers, mail clients; swap files.
- But wait, there is more...

Limitations of file shredding

- The file system may relocate a file block when it is updated, to reduce file fragmentation.
- 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 problems exist with file systems that encrypt each file with its own encryption 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 invalidate the program completely.
  - Examine raw disk blocks (network packets, etc.)
- Are we solving the right problem? Zero filling free disk space (and all swap!) may be more effective.