The journal of Apple technology.

Volume Number: 19 (2003)
Issue Number: 6
Column Tag: Section 7

Section 7

File Mode Idioms

by Rich Morin

Which modes are used for what?

Each Mac OS X file system node has a 16-bit mode word, as described in chmod(1,2) and stat(2). This word specifies the node's type, what access modes are allowed, and some specialized handling. 16 bits provides 64 K possible variations, but only a relatively small number of "file mode idioms" are found with any frequency. By learning some of these idioms, you can make your system more secure and understand existing configuration decisions.

File Types and Modes

Before we get into the idioms, however, let's review the basics of file types and modes. The file system recognizes many types of "files", including a number of things (e.g., directories) that aren't really files, at all. The most common file type, however, is the "regular file", followed by the "directory" and the "symbolic link" (aka symlink(7)).

There are dozens of device files (see mknod(1,2,8) for details), but they are almost always segregated into the /dev directory. Sockets and named pipes can be used to enable interprocess communication between arbitrary processes (see mkfifo(2) for details). Finally, the "whiteout" type is used for the (ever-experimental) Union File System, described in FreeBSD's mount_union(8) man page.

The node's type is specified by the top four bits in the mode word, available via the stat(2) system call:

```
0160000  whiteout
0140000  socket
0120000  symbolic link
0100000  regular file
0060000  block special device
0040000  directory
0020000  character special device
0010000  named pipe (fifo)
```

The next three bits specify some forms of "special handling". If the node is a regular file, the bits are interpreted as follows:

```
0004000  set user  id on execution
0002000  set group id on execution
0001000  save swapped text, even after use
```

The first two bits are described in the setuid(2) man page. Briefly, they allow a program to run with the permissions of

its owner (or group), rather than those of the user who started it. This is used to provide controlled access to elevated privileges in (carefully constructed!) system commands.

You can get a list of your system's setuid and setgid commands with the following C-shell command (use whereis(1) or which(1) to find the full path name of a specific command). In the output below, df is setgid to "operator" and rcp is setuid to "root":

```
% ls -l /{,usr/}{,*/}{,s}bin | grep '[r-]-s'
-r-xr-sr-x  1 root  operator  ...  df
-r-sr-xr-x  1 root  wheel     ...  rcp
...
```

The third bit, described in sticky(8), tells the system to retain the read-only parts of a program's image in memory, after the program has terminated. This can be used to reduce the start-up time for frequently-run programs. Whether your OS honors the request is, of course, up to the vendor (:-).'

If the node is a directory and the system is SysV-ish (e.g., Red Hat Linux), the setgid bit may be interpreted as forcing "BSDish" behavior in setting the group for a new file. That is, a new file will get the enclosing directory's group, rather than the user's. On BSDish systems (e.g., OSX), this bit has no effect.

Sticky directories are a bit more complex; here's a snippet from the manual: "A file in a sticky directory may only be removed or renamed by a user if the user has write permission for the directory and the user is the owner of the file, the owner of the directory, or the super-user. This feature is usefully applied to directories such as /tmp, which must be publicly writable but should deny users the license to arbitrarily delete or rename each others' files."

The bottom nine bits are divided into three sets of permissions (for the file's owner, members of the file's group, and everyone else); each set specifies read, write, and execute permission:

```
0000400  read  permission, owner
0000200  write permission, owner
0000100  execute/search permission, owner
0000040  read permission, group
...
```

The meanings of read, write, and execute are a bit strained, when it comes to directories. Read permission allows the user to "read" the directory, looking for file names, etc. Write permission allows the user to "write" the directory, creating or removing files, etc. Finally, execute permission allows the user to access an item contained within the directory.

File Idioms

Most files are readable and writable by their owners. If nobody else is expected to access the file, no other permissions are needed. However, it is common to allow group access, as well:

```
% touch 0600 0660
% chmod 0600 0600
% chmod 0660 0660
% ls -l 0*
-rw-------  ...  0600
-rw-rw----  ...  0660
```

Obviously, executable files need to have the appropriate "execute" bits set. Less obviously, the "read" bit must be set for scripts (so the interpreter can read them!). In practice, even binary executables tend to have read access turned

on; for one thing, this allows debuggers to inspect the binary.

Distributed executables often have write access turned off. This seems like a good idea, because it reduces the chance of inappropriate modification. An inspection of /usr/bin on my OSX system, however, shows that this practice isn't universal:

```
-rwxr-xr-x  ...  cscope
-r-xr-xr-x  ...  ctags
```

System-wide files, such as the executables in bin directories, generally need to be accessible by everyone on the system. Many system control files also need universal read access:

```
-r--r--r--  ...  /etc/crontab
```

sudo(8) allows any command to be run as if by any specified user, assuming that the actual user can supply the required password. Unfortunately, this requires passwords to be handed out, remembered, guarded, etc. Fortunately, the file system provides an elegant solution:

```
-rwsr-x---  1 root  wheel  ...  so
```

The "so" command (at least our version :-) gives root privileges to anyone who is in group wheel. If arguments are given, they are run as a command line; otherwise, the user is given a root-enabled shell.

Directory Idioms

The mode bits for home directories should keep each user's files safe from casual inspection (let alone modification). Depending on the environment, and your own level of paranoia, one of the following is probably appropriate:

```
drwx------  ...  abc  grp1  ...
drwx--x---  ...  def  grp2  ...
drwxr-x---  ...  ghi  grp3  ...
drwxr-x--x  ...  jkl  grp4  ...
drwxr-xr-x  ...  mno  grp5  ...
```

User abc doesn't want anyone else doing anything with his files. User def is willing to let members of group grp2 access files, but only if they know the file's name (removing read access from directories turns off ls access, wild cards, etc.). User ghi seems to trust his group fairly well, but still doesn't want them creating files in his home directory.

Users jkl and mno trust everyone on their computer as much as users def and ghi trust members of their groups. Because security tends to be antithetical to convenience, user ghi has the least problems sharing files, etc. For what it's worth, I use mno's mode on my desktop machine and jkl's mode on my server account.

Allowing directory execute access by others can be quite useful. Let's say that you want to set up a "drop box" where other users can leave files. Anyone should be able to drop stuff off, but only you should be able to look into the box, retrieve files, or (gasp!) remove files. Here's how:

```
% chmod 1733 drop_box
% ls -d drop_box
drwx-wx-wt  ... drop_box
```

As the owner, I am allowed to do anything to the directory. Others (including members of my group) are allowed to access and even create files in the directory, but they are not allowed to list its contents. Finally, the "sticky" bit

(described above), keeps anyone but me from removing files from the directory.

I encourage you to set up a "testbed" directory and try out different modes within it. Try out different combinations to see how they might be useful; all of this will pay off some day when you're trying to figure out an obscure "file not found" or "permission denied" error message!

Rich Morin has been using computers since 1970, Unix since 1983, and Mac-based Unix since 1986 (when he helped Apple create A/UX 1.0). When he isn't writing this column, Rich runs Prime Time Freeware (www.ptf.com), a publisher of books and CD-ROMs for the Free and Open Source software community. Feel free to write to Rich at rdm@ptf.com.