Securing Unix Filesystems - When Good Permissions Go Bad

Introduction

Unix has a very elegant and flexible permission system at the heart of its filesystem security. These permissions allow and/or disallow access to each file on the system by various users and groups. Unfortunately, this system is often misused in such a manner that parts of the system become insecure, vulnerable to both malicious and accidental damage.

The permission system revolves around three major components: mode, owner, and group. Each file on the system has all three of these elements. And each file, being present for a unique purpose, has individual needs. Not all files are alike, nor should they all have the same permissions. To blanketly apply one set of permissions to many files out of "convenience" is to undermine the very purpose of the permission system--keeping files as secure as possible. Yet this is exactly what happens on all too many systems that are deployed.

Whether you are an administrator at some level, or an end-user that "just works there", let's take a few minutes to explore the permission system, and how it is best used, as well as the mistakes not to make.

How It's Designed To Work

The first thing to consider about files under unix systems is who owns them. There are two relevant settings in this case, the owner and group. The owner is a username. The group is a group name, which resolves (via /etc/group) to a list of usernames, any one of which will test as true when that group name is checked. Groups will make more sense after we've discussed the basics of modes.

The -real- power of the system lies in the mode. There are two ways to deal with modes on a system level via the 'chmod' program--absolute and symbolic. Let's look first at the absolute modes.

Absolute modes are specified in octal notation, and are bitmasks. Each digit place represents three bits of a two-byte mode, and can be set to a bitwise OR of potential values. The rest of the two bytes, in case you're counting bits, is used by the system for information about the type of file, and is not germaine to chmod(1).

Basically, each place can be set from 0-7. Only 4, 2, and 1 are standalone settings. The remaining values are the results of bitwise OR's: 2+4 gives you 6, for instance. In code, it's a bitwise OR operation. Mathematically, you simply add the values for the bits you wish to set.

Let's break down the bit setting functionality, remembering that there are four relevant places that we deal with when setting modes. We'll handle the leading place last, since it has some special functionality and is more complex than the others.

The last three bit settings each correspond to owner, group, and others. Others would be any user that is not the owner, and does not belong to the group.

In each case, the following values have the following functionality:

4	-	Read permission granted
2	-	Write permission granted
1	-	Execute permission granted
0	-	No permission granted

For files, the execute bit is exactly what it means; the file may be executed as a runnable program. In general, if it is a binary that runs under your platform, it is executed. If it is a script or other file, it is fed to an interpreter. A **#!/path/to/interpreter** line at the top of a script script is preferable, but if a file lacks one the file will be fed to the default shell interpreter.

For directories, the execute bit signifies that the directory may be searched *through*, but not directly read. For instance, a directory with the execute bit set, but the read bit not set would allow you to access specific files under that directory by name, but you could not obtain a list of files in that directory with **ls** or any other program. An excellent demonstration of the usefulness of this is the idea of giving users access to a program by telling them what file to execute, but not letting them see other files in the same directory that you may not want them to. They can only use what they know about. If you've ever logged into an FTP cite and been curious enough to change directories into perhaps **/bin** but gotten an empty listing back, this is precisely the methodology used.

The leading bit setting in a mode implements special functionality:

- If an executable has a leading place set to 4, when the program is executed, it is executed as the owner of the program, not the user calling it. This is referred to as SUID.
- If an executable is set to group execute mode (in any combination of logical ORs), and the leading place is set to 2, the executable is run with as if it were a member of the group that owns the file, rather than as the group of the calling user. This is referred to as SGID.
- If an executable is set to a non-execute group mode, and the leading place is set to 2, the file is mandatorily locked while any program accesses it, meaning that other programs may not read/write from or to that file. If the node is a directory, any files or subdirectories created there are created with the group ID of the parent directory with these modes set, rather than that of the calling process.
- If a directory is set with a leading place of 1, any user that would ordinarily be able to create a file or directory under it according to the rest of the permissions may do so--*but*, no other user may remove those files or directories. Only the owner of a file or directory (or the superuser) may remove a file from a directory with this bit set. This is referred to as the "sticky bit".
- It is possible to combine SUID and SGID (6), as well as SGID and the "sticky bit" (3). It is not possible to enable mandatory locking and SGID at the same time.
- It should be noted that the "mandatory locking" functionality appears to be system-dependant, and may not be available on your platform. Please consult your chmod(1) documentation for further details.

Symbolic modes are simply a mask representation of the absolute modes:

```
 u  g  o    (user, group, others)
rwxrwxrwx
```

In the case of the first place from absolute notation, 4 will change the user execute position to read 's'. A setting of 2 will do the same thing at the group execute position. A setting of 1 will change the execute position for others to read as 't'. This last only works for directories, as the "sticky bit" has been depricated for files.

You can read more on using symbolic modes with 'man chmod'.

Permissions are evaluated left to right. This lends itself to some interesting situations that can be beneficial for security purposes. Take an example of a file with mode 0705, owner 'fairlite', group 'restricted'. This file may be read from, written to, or executed by 'fairlite', and read from and executed by all other users--*except* any user that belongs to the group 'restricted'. Because of the way permissions are evaluated in order, left to right, if you belong to that group, the evaluation for you ends there, and does not proceed on to "others". (It is worth noting for those that use a program like MS Internet Explorer to FTP files and change their modes that IE's verbiage of "All Users" really translates to "others" and is a complete misnomer.) Many people do not understand the evaluation order, and assume that the "others" setting will override user and group settings. This is not the case.

Now that we have an overview of how permissions work (full details are available via 'man chmod' on unix systems), let's explore some of the foibles that create holes in security, and weakened security.

Executable Data

Put simply, there is NO reason for a data file to be executable. If a file is not meant to be run as a program, it does not need, nor should it have an executable bit set.

Setting an executable bit on a program lets people run a program arbitrarily, without taking the extra step of calling a shell interpreter, at a minimum.

For instance, if I had a text file (like this essay) which contained an example of a command sequence that was dangerous, such as:

```
;ls / ; rm -rf /home ;
```

...anyone that happened to type the name of the file as a command would end up triggering a removal of the **/home** filesystem if they happened to have permission to do so. Ordinarily, this example would require you to be root or the owner of **/home** and its contents to do any damage, but as we see later, this is not always the case. Even if others cannot execute a file, if the owner has execute permissions and did this, they would wipe out any files they had permission to remove.

One should also keep in mind that a file need not be set to executable to be executed. Anyone with permission to read the file may attempt to execute it by feeding it manually to an interpreter. A Bourne shell script with mode 0744 may still be executed by anyone on the system by using the command **/bin/sh** . You may wish to lock down read ability on scripts that you don't want others to run.

One other thing that should be noted is that shell scripts *require* read permission to run. While binaries will run with only the execute bit set, an interpreter needs both the execute -and- read bits set, because it must first read the file.

World-Everything Directories

There is nothing quite so insecure as a directory with the mode 0777 set, except perhaps leaving your keys in the ignition of your car, the doors unlocked, and putting a sign on the dash saying, "Steal me!"

With the writeable bit set for "others", *anyone* may add or remove files to a directory, whether they own it or not.

Similarly, group write permission may be dangerous if used incorrectly in the context of a general group. If all users are in group "users", there is generally no excuse or reason for having 077x permissions on a directory, as anyone in the group can remove anyone else's files.

Spool directories where you want multiple users to write files should use the "sticky-bit", or a leading '1' in octal, such as 1777, or in the case of group access, 1775 (or stricter, depending on what the "others" need access to). This allows anyone (either in the group, in group-only context, or anyone at all in world-writeable context) to create a file in that directory, but ONLY the owner of the file may remove it.

World-Everything and World-Writeable Files

Even assuming you have your directories secure, your files should be secured as well. Assuming a mode 0666 on a file inside a directory that is moded 0755, anyone may actually open the file and write arbitrary data to it, or even wipe out segments of it, despite the fact that they cannot create new files in that directory. You don't need to have write permission to a directory to affect an insecure file inside that directory.

Excessive SUID/SGID Privileges

In these days of hightened security awareness, many vendors are trying very hard to (wisely) limit the number of programs that run SUID and/or SGID. Not all SUID/SGID files are "bad". For example, most mail transport agents run SGID at the least.

The most potentially serious SUID programs are those owned by root. If a program is owned by a user that has no other privileges on the system, it is far safer if it breaks, because it can only damage files which pertain to it (assuming a properly designed system).

The steps to keeping things as secure as possible are:

- Make sure the program really needs SUID or SGID permissions to function properly.
- Make the owner "isolated" if at all possible. It is easy to just set root ownership and SUID on a program so that it can write to anywhere. But if that program is ever vulnerable to an exploit, it could then enable a write to (or delete from!) anywhere that root can. In short, root's permissions would be gained.
- To keep this risk to a minimum, if you can run a web server as 'nobody' or 'www', a separate user who is isolated and really has no privileges anywhere outside its own subsystem, you should do so. The same applies to groups and SGID modes.
- If you are the developer of the program, ensure that the program drops its elevated privileges at every opportunity, especially before operations such as system(), fork(), and exec(). The less time a program spends with elevated privileges, the less chance there is for it to be exploited if a bug is discovered.

Summary

Perhaps the simplest way of summing up permission management technique is, "Be paranoid." If you want a secure system, tighten things to the most stringent modes possible while maintaining usability. Use the permission system to its fullest potential to keep things hardened.

Remember, not all attacks are external. But external attacks can lead to local attacks. And local attacks can be very costly, even if the attack is by a non-root account, if your permissions are lax.

Tools

One could use a variety of methods for scanning for various modes. The **ls -la** command is one of the most useful. You can spend a long time looking at and tweaking your filesystem contents to be more secure.

I have written a utility that looks for the "most dangerous" modes that require close observation (SGID/SUID), or outright fixing ("group" and/or "other" write modes either alone or in concert with other modes). This tool is written in Perl version 5, and generates full reports. It will even email them to you, if you have a usable SMTP server at your disposal. It tracks files it has already found, so if you set it up to run every evening on your system and point it at **/home** or wherever your users keep files, you can run selected audits and see dangerous changes as they appear, but not be bothered with the rest. You may, of course, request a full audit report at any time. With multiple, selectable output formats, you can choose the output that's right for you. You also get the source code, so you can add or remove scan modes if you wish.

If you wish to acquire this program, the Fairlight Permission Analysis Assistant, please click here to license it. The license cost is $59.95 per system.

As titled, the program is an *assistant*. It will report its findings. Not all of the results are bad, or require fixing. It is the responsibility of the person doing the analysis to recognize the import of what results are given to them. The analysis process still takes time and effort, but this tool does the data collection for you in a convenient manner.

System requirements for **flpaa** are:

- Perl 5.6 or higher
- Getopt::Long