Working with Access Control Lists

ON THE LIST

IKO, Fotolia

The ancient Linux permission system is often insufficient for complex production environments. Access Control Lists offer a flexible alternative. **BY TIM SCHÜRMANN**

Alice appreciates the convenience of a PC-based electronic calendar, but to maintain her privacy, she has set strict permissions for her calendar file: She can add new appointments herself, but other members of her workgroup have read-only access. Others outside of her workgroup are not even allowed to look.

This configuration is fine at first, but one day Bob from another department agrees to collaborate with Alice. To allow this to happen, she has to give him access to her calendar data.

In this scenario, it is clear that the legacy Linux permission system has outlived its usefulness. To allow Bob to read the file with her calendar data, Alice can ask the administrator to move the new colleague into her own group, but this would allow Bob to view all the other documents produced by Alice's team. Another approach would be to temporarily set up a completely new user group with both Alice's and Bob's accounts as members. In this simple scenario, a temporary group might be an acceptable solution, but in a real-world enterprise environment, group management becomes far more complicated, and the habit of creating temporary groups on the fly can lead to too many groups with no good way of tracking them.

Access Control Lists, or ACLs for short, promise a solution. They add flexible access control to the legacy Unix permissions system, letting users add permissions for any group or users. Alice doesn't even need to talk to the administrator; she can simply put Bob on the list of authorized users and even specify default permissions for all new files.

Access Control Lists have been around for a while, and they are gradually becoming part of daily life in many production environments; however, the ACL security structure is still unfamiliar to many Linux users. In this article, I show you how to get started with ACLs in Linux.

Rotating Disks

If you plan to use ACLs, your filesystem must support extended attributes. Of the current crop of filesystems, Ext2, Ext3, Ext4, ReiserFS, JFS, and XFS all have ACL support. JFS and XFS support extended attributes by default; for all others, you need to stipulate the *acl* mount option to enable ACLs:

```
mount -o remount,acl,➥
defaults mount_point
```

Most current distributions set these parameters by default in */etc/fstab*:

```
/dev/hda1   /   ext3   acl,➥
user_xattr 1 1
```

For internal disks, you need not change anything, and ACLs also work over NFS as of NFSv3, assuming the server has a filesystem and operating system that support ACLs.

Kernel Issues

Besides the filesystem, the kernel also must support ACLs – after all, it is the kernel that finally grants or refuses access to a file. All current kernels in the 2.6 series have ACL support, and patches exist for the legacy 2.4.x kernel. The major distributions typically enable extended attributes for all of the filesystems mentioned previously, allowing users to start assigning permissions from scratch. To be sure, just enter the following command:

```
grep "XATTR\|POSIX_ACL" ➥
/boot/config-$(uname -r)
```

It should show two entries with *=y* if ACLs are supported. Ext2, for example, would show:

```
CONFIG_EXT2_FS_XATTR=y
CONFIG_EXT2_FS_POSIX_ACL=y
```

Otherwise, you have no alternative but to install a new kernel.

Band of Two

Besides filesystem and Linux kernel support, you also need a package with applications that display the ACLs for each file and modify them as needed. Most distributions include a package called *acl* for this purpose. Two of the programs it includes are particularly useful:

- *getfacl* displays the ACL for a file, and
- *setfacl* sets or changes the permissions for a file

Both tools rely on the *libattr* and *libacl* libraries, which many distros install by default.

POSIX and ACLs

You might stumble across the term POSIX (Portable Operating System Interface) ACLs on the Internet and in documentation. Although various drafts appeared at the end of the last century (POSIX 1003.1e, commonly referred to as POSIX.1e, and 1003.2c), for several reasons, the drafts were never approved. Most ACL implementations are still oriented on these drafts. To underline the close connection, many authors use the term POSIX ACLs [1].

History

To begin this study of ACLs, I'll take a quick look at the legacy Linux access control system. Normally in Linux, each filesystem object distinguishes between three different roles, which are called *classes* in POSIX terminology: the owner of the file (user), the group the file belongs to (group), and all other users (other). The owner specifies – for each of the three classes – whether the class can read, write to, or even execute the file. The familiar *ls -l* command displays these permissions as a cryptic list of letters:

```
-rw-r--r-- 1 alice ateam ➥
5410 7. Feb 11:21 calendar.cal
```

In this case, *alice* is the owner and her group is called *ateam*. *ls* appreciates read, write, and execute commissions to their first letters: *r*read, *w*rite, and e*x*ecute. Thus, for each class of users, a triplet of access permissions is used in *rwx* format. A dash (-) at any position means that the operation is forbidden.

Minimalism

Imagine an ACL as a piece of paper on which you jot a list of all other access permissions and the rights assigned to them. Linux staples the results onto the file and enforces the permissions on the list. In practice, Alice first checks which permissions are already assigned for her calendar. She uses *getfacl* to do so; the command outputs the ACL for a file. Because she has not yet added Bob, his list should be empty:

```
$ getfacl calendar.cal
# file: calendar.cal
# owner: alice
# group: ateam
user::rw-
group::r--
other::r--
```

For a list that should really be empty, this has quite a few entries.

First, *getfacl* repeats the file name, the owner of the file, and the group in the first three lines. The following lines each contain exactly one entry from the ACL; this is aptly known as an Access Control Entry (ACE). To retain downward compatibility with legacy systems, the ACL automatically maps existing permissions to entries in the list – this explains the three following lines in the above example. The first line shows the owner's permissions, the second shows permissions for the group, and the third shows those for all other users.

The entries thus precisely match what *ls -l* told us. Because these entries exist in any ACL, they are referred to as the minimal ACL. As soon as an entry is added, this is referred to as an extended ACL.

Setting ACEs

To grant Bob access to the calendar, Alice needs to set another entry for this ACL. The *setfacl* tool takes care of this:

```
setfacl -m user:bob:rw- ➥
calendar.cal
```

The parameters only appear cryptic at first glance: the above command line creates a new entry (*-m*) in the ACL for the *calendar.cal* file. Access is granted to a single *user* called *bob*. Bob can read and write to the file, but he can't execute it (*rw-*). *getfacl* outputs the resulting list:

```
$ getfacl calendar.cal
# file: calendar.cal
# owner: alice
# group: ateam
user::rw-
```

Windows and ACLs

Windows, as of XP, and the NT operating system family also support ACLs, but only with the NTFS filesystem. Although access to NTFS on Linux is typically possible, Linux has only limited support for its extended functionality, which unfortunately includes ACLs.

At least Samba supports ACLs, assuming the underlying system has support for extended permissions. Files stored on the Samba server keep their permissions as if they were stored on a normal NTFS drive. One problem remains: Because the ACLs use by Windows and Linux differ, Samba currently gives Windows users only a part of the functionality they are used to.

```
user:bob:rw-
group::r--
mask::rw--
other::r--
```

Each entry in an ACL follows the same pattern, starting with the type, which specifies to whom the following permissions were applied. This can be a single *user* or a whole *group*. A label follows the colon. This designates the name of the user or group the entry belongs to. In cases in which you do not need this name, you can simply omit it – you can see an example of this in the entries for standard permissions. The line ends with the familiar triplet of permissions.

Who Has Rights?

In the course of the project, Alice needs to grant the other members of Bob's group short-term access to her calendar. To do so, she simply adds an entry for *bteam*:

```
setfacl -m group:bteam:r-- ➥
calendar.cal
$ getfacl calendar.cal
# file: calendar.cal
# owner: alice
# group: ateam
user::rw-
user:bob:rw-
group::r--
group:bteam:r--
mask::rw--
other::r--
```

When read access for a file is requested, Linux simply checks permissions one after another.

First, the kernel checks to see whether the user has an entry, and if so, Linux applies the permissions defined by the entry. In the example here, Bob is granted read and write access to the calendar. In contrast, a personal entry does not exist for Carl, so Linux checks group permissions. Because Carl is a member of *bteam*, he is given read access. The kernel is unable to find a personal or group entry for the staff from accounts; in this case, standard Unix permissions apply.

Legacy

Unfortunately, *ls -l* does not display extended permissions. Instead, a single plus sign indicates the existence of extended permissions:

```
$ ls -l calendar.cal
-rw-r--r--+ 1 alice ateam ➥
5410 7. Feb 11:21 calendar.cal
```

Because the ACLs map standard Unix permissions, *setfacl* also replaces the good old *chmod* command. You simply have to change the entries. For example, the command

```
setfacl -m other::rw- ➥
calendar.cal
```

grants read and write access for the file to all other users, including accounts:

```
-rw-r--rw-+ 1 alice ➥
ateam 5410 7. ➥
Feb 11:21 calendar.cal
```

Every extended ACL contains a fairly ominous looking *mask* entry. The mask describes the maximum permissions the user can be granted.

If the mask defines a more restrictive permission set than is granted to a user in an ACE, the mask always takes priority. For example, Alice could have granted other members of Bob's group access to her calendar.

If she now wants to temporarily revoke these permissions, she would have to modify them for each member of the group; alternatively, she can just modify the mask:

```
setfacl -m mask::r-- ➥
calendar.cal
```

No matter what permissions the users have beforehand, from now on, they can only read the calendar. Of course, this applies to Bob too. Although he still has write permissions for the calendar, the mask takes priority, leaving him with just three permissions.

Under the hood, Linux performs a logical *AND* operation to calculate effective rights. To be able to read a file, the user or group thus needs read permissions and read permissions must exist in the mask.

To avoid administrators losing track, *getfacl* outputs the effective actual permissions for each user:

Listing 1: Creating a Default ACL

```
$ setfacl -m user:bob:rw-
projectfolder
$ setfacl -d --set user:bob:rw-
projectfolder
$ getfacl projectfolder
# file: projectfolder
# owner: alice
# group: ateam
user::rwx
user:bob:rw-
group::r-x
mask::rwx
other::r-x
default:user::rwx
default:user:bob:rw-
default:mask::rwx
default:other::r-x
```

Setfacl Investigated

setfacl has the following useful parameters: *-m* modifies or creates a new entry,

```
setfacl -m user:bob:r-- ➥
calendar.cal
```

which is deleted by *-x*:

```
setfacl -x user:bob ➥
calendar.cal
```

This only removes the specified entry, but it does not affect the group that Bob belongs to. *--set* removes all previous entries, setting only the new ones:

```
setfacl --set user:bob:r-- ➥
calendar.cal
```

Finally, the *-b* option empties the complete list. Additionally, setting the recursive parameter *-R* tells *setfacl* to work its way through the whole directory tree. At the same time, you can even set multiple comma-separated permissions:

```
setfacl -m user:bob:r--, ➥
group:cteam:rw- calendar.cal
```

Instead of user and group names, you can alternatively specify the UIDs and GIDs; *setfacl* will also accept permissions in a numeric format.

A couple of abbreviations apply in *setfacl*. Instead of *user*, you can simply say *u*. In similar fashion, the abbreviations *g*(roup), *m*(ask), *o*(ther) and *d*(efault) exist. It is also possible to leave out multiple subsequent dashes, as long as the command is unambiguous:

```
setfacl -m u:bob:r- ➥
calendar.cal
```

```
$ getfacl calendar.cal
# file: calendar.cal
# owner: alice
# group: ateam
user::rw-
user:bob:rw-     #effective:r--
group::r--
mask::r--
other::r--
```

Although Bob has write permissions, all he effectively keeps is read permission for the calendar.

Masks introduce another pitfall: *setfacl* autonomously changes the mask whenever you modify the permissions for user or group. If you misuse the mask, like Alice did in the example here, you have no alternative but to check its validity.

Standard

While she is working on a project, Alice creates a number of project files that Bob also needs to read. For each new document, Alice could theoretically modify the permissions manually. An easier option is to create directories with a default ACL. The subdirectories and files below the directory automatically inherit the default permissions. Directories have both an ACL of their own and the new default ACL of the parent directory. You can create a new default ACL by running *setfacl* (see Listing 1).

getfacl always lists standard permissions at the end. The format reflects the legacy entries, but each time starts with *default*. If you are only interested in the default entries, you can specify *getfacl -d*; alternatively, *getfacl -a* prevents the default entries from being displayed.

Because file access is handled through the kernel itself, legacy programs have no trouble working with extended permissions, which is not true of applications that manipulate file permissions, such as Konqueror or Nautilus. Konqueror version 3.5 or later can handle ACLs (as you can see in Figure 1), as can Nautilus 2.16 or newer.

Standard Unix commands such as *cp* or *mv* have been modified to handle ACLs. Loss-free copying or moving assumes that the target file system supports ACLs. If not, you only keep simple, legacy file permissions.

Programs that have not been modified to support ACLs simply change the standard permissions. One example of this is performing the backup. The legacy *tar* program does not keep ACLs. In this case, you should choose an alternative, such as Star tape archiver (*star*), which is included with many distributions. The command

```
star H=pax -acl -c -f ➥
backup.pax project_folder/
```

creates (*-c*) the archive (*-f*) *backup.pax* and stores the contents of the *project_folder* in it. The program uses the PAX file format. The command

```
star -acl -xp -acl -f ➥
backup.pax
```

unpacks the package created by the previous command.

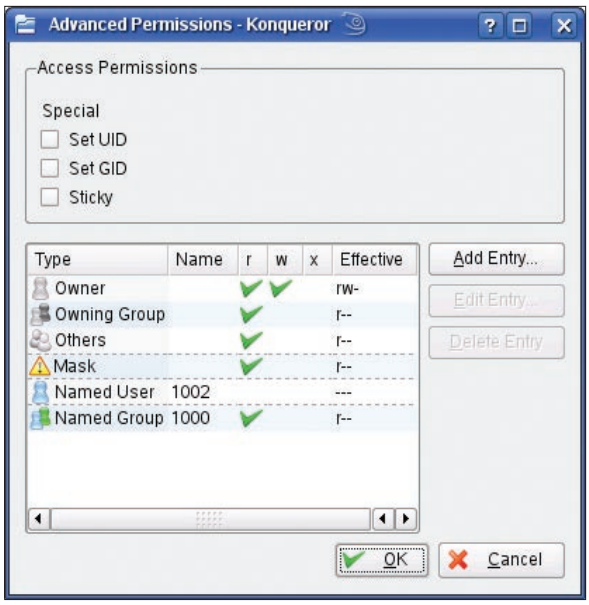

Figure 1: In KDE's Konqueror, the Extended permissions button in the File Properties dialog box takes you to this dialog, where you can conveniently modify ACL entries.

Collective

To create complex ACLs, you need to run *setfacl* multiple times, which is unwieldy and time consuming. Luckily, administrators can store the entries in a text file in the same format as *getfacl* output and then run *setfacl* with the *--set-file="file.txt"* parameter to restore the permissions. The use of - instead of a file name tells the tool to read the parameters from standard input. This means that you can move an ACL from one file to another:

```
getfacl file1 | setfacl ➥
--set-file=- file2
```

If you prefer to avoid exotic formats, I recommend a simple trick: You can tell *setfacl* to parse its parameters from a text file. The format precisely matches the output from *getfacl*, so it would seem to make sense to use *getfacl* to store all of the ACLs in a text file and then run *setfacl* later to restore them. To start, write the ACL output from *getfacl* to a text file:

```
getfacl -R --skip-➥
base ➥
directory/ > ➥
/backup.acl
```

This tells *getfacl* to change to *directory* and write the ACLs for all the objects it finds to a file called *backup.acl*. The *-R* parameter makes this process recursive. Then, you can store the ACL file with the actual content of the directory using your preferred packer. To restore the ACLs, run *setfacl*:

```
setfacl --restore=backup.acl
```

Conclusions

Access Control Lists offer extremely flexible permission management. At the same time, ACLs take some of the workload from the administrator's shoulders by shifting responsibility for access permissions to the file owner. Thanks to default ACLs and masks, the administrator still keeps control.

Data exchange remains a major problem. Because almost every operating system uses a different ACL variant, the various versions are typically incompatible or only partially convertible, so permissions are often lost in the conversion process. Administrators in heterogeneous environments therefore have to keep on their toes. ■

INFO

[1] POSIX ACL drafts: http://wt.xpilot.org/publications/posix.1e

[2] Konqueror: http://www.konqueror.org/

[3] Nautilus: http://www.gnome.org/projects/ /