Implementing an advanced access control model on Linux

Aneesh Kumar K.V
IBM Linux Technology Center
aneesh.kumar@linux.vnet.ibm.com

Andreas Grünbacher
SUSE Labs, Novell
agruen@suse.de

Greg Banks
gnb@fmeh.org

Abstract

Traditional UNIX-like operating systems use a very simple mechanism for determining which processes get access to which files, which is mainly based on the file mode permission bits. Beyond that, modern UNIX-like operating systems also implement access control models based on Access Control Lists (ACLs), the most common being POSIX ACLs.

The ACL model implemented by the various versions of Windows is more powerful and complex than POSIX ACLs, and differs in several aspects. These differences create interoperability problems on both sides; in mixed-platform environments, this is perceived as a significant disadvantage for the UNIX side.

To address this issue, several UNIXes including Solaris and AIX started to support additional ACL models based on version 4 of the the Network File System (NFSv4) protocol specification. Apart from vendor-specific extensions on a limited number of file systems, Linux is lacking this support so far.

This paper discusses the rationale for and challenges involved in implementing a new ACL model for Linux which is designed to be compliant with the POSIX standard and compatible with POSIX ACLs, NFSv4 ACLs, and Windows ACLs. The authors' goal with this new model is to make Linux the better UNIX in modern, mixed-platform computing environments.

1 Introduction

File access control is concerned with determining which activities on file system objects (files, directories) a legitimate user is supposed to be permitted. It mediates attempts to access files, and allows or denies them based on administrative metadata attached to those files.

Linux has traditionally had a file access control model based on the traditional UNIX file mode model standardised by POSIX [7]. This model is proven, robust and simple. Beyond that, Linux implements the non-standard but widely deployed POSIX ACL model suitable for more complex permission scenarios, all within the bounds that the POSIX standard defines.

However, when a Linux system uses a remote filesystem access protocol like CIFS or NFSv4 to share files with a Microsoft Windows system, the mismatch between the Linux and Windows access control models poses a significant interoperability challenge. This is a very common and economically significant deployment scenario, and other UNIX-like systems (Solaris [10] and AIX [1]) have a solution to these problems.

In this paper, we discuss the challenges involved in implementing an advanced access control model on Linux which is designed to address these problems. The new model is based on the NFSv4 ACL model [6], with design elements from POSIX ACLs that ensure its compliance with the standard POSIX file permission model.

The NFSv4 ACL model is in turn based, somewhat more loosely, on the Windows ACL model [3]. This means the mapping between the new model and Windows ACLs, while not completely trivial, is at least predictable, understandable, and not lossy. This has the benefit of smoothing remote file access interoperability.

Another benefit is to provide Linux system administrators with an access control model for local filesystems which is finer-grained and more flexible than the traditional POSIX model. Some system administrators might also find the new model more familiar.

For compatibility and security, it is necessary to ensure that applications using the traditional POSIX file mode based security model still work when using a filesystem which implements the new ACL model. This results in a number of technical challenges whose solution we will describe.

2 File Permission Models

This section describes and compares the main file permission models in use today: the standard POSIX file permission model and the widely supported POSIX ACLs on the UNIX side, Windows ACLs on Windows, and NFSv4 ACLs, a hybrid between these two major approaches.

2.1 The POSIX File Permission Model

The traditional file permission model implemented by all UNIX-like operating systems including Linux follows the POSIX.1 standard [7]. The standard can be thought of as a "contract" between application programs and the operating system: POSIX.1 defines the mechanisms available to portable applications and specifies how compliant operating systems will react. Application programs can rely on the POSIX.1 behaviors; this is of major importance to system security.

POSIX.1 distinguishes between read (r), write (w), and execute/search (x) access. The read permission allows to read a file and directory, the write permission allows to write to a file and create and delete directory entries, and the search/execute permission allows to execute a file and access directory entries.

As explained in POSIX.1 Base Definitions, each file system object is associated with a user ID, group ID, and a file mode which includes three sets of file permission bits. A process that requests access to a file system object is classified into one of the three categories owner, group, and other depending on its effective user ID, effective group ID, and supplementary group IDs. This so-called file class determines which set of permissions determine if the requested access is granted. The access is granted if the set of permissions associated with the file class includes the permissions needed for the requested access, and otherwise denied. Figure 1 depicts this graphically.

To give an example, assume that a process tries to open a file for read access and the file permission bits as shown by the `ls` command are `rw-r-----`, granting read and write access to the owner class and read access to the group class and no access to the other class. The access is granted if the effective user ID of the process matches the user ID of the file, or if the effective group ID or any of the supplementary group IDs of the process match the group ID of the file; in all other cases, the access is denied.

The file permission bits are usually set so that the group class has the same or fewer permissions than the owner class, and the other class has the same or fewer permissions than the group class. However, other values like `-w-r-----`, which grants write access to the owner class and read access to the group class, can also be used. Because a process can only be in one file class at any one time, these file permission bits do not allow any process to get read and write access simultaneously.

While this model is flexible enough for a large number of real-world scenarios, the three permissions and three possible roles of processes can become a burden or be too limiting. For example, when people form an ad-hoc team which the operating system does not know about, they will have difficulties with sharing files in this team: the file permission model will not allow them to grant each other access to files without granting others outside this group access as well. The system administrator can help by creating a new group, but this administrative overhead is undesirable, and the number of groups can grow unreasonably large.

In awareness of these limitations, the POSIX.1 standard defines that the file group class may include other implementation-defined members, and allows additional and alternate file access control mechanisms:

- Additional file access control mechanisms may only further restrict the access permissions defined by the file permission bits.
- Alternate file access control mechanisms may restrict or extend the access permissions defined by the file permission bits. They must be enabled explicitly on a per-file basis (which implies that no alternate file access control mechanisms may be enabled for new files), and changing a file's permission bits with the `chmod` system call must disable them.

Many texts on the UNIX operating system describe the POSIX.1 file permission model in more detail including Advanced Programming in the UNIX(R) Environment [15].

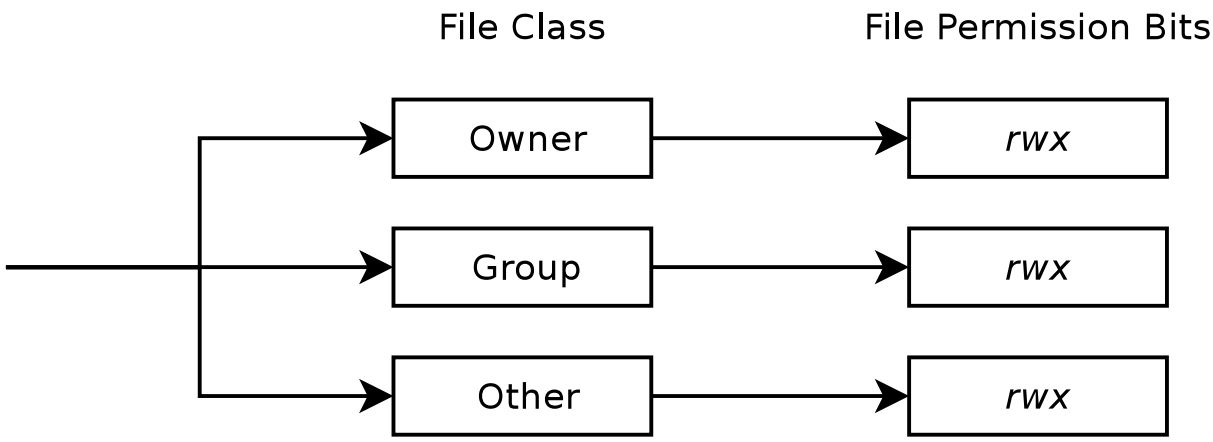

Figure 1: The POSIX.1 File Permission Model

2.2 POSIX.1e Access Control Lists

As we have seen in the previous section, the POSIX.1 file permission model only uses the file user ID and file group ID to distinguish between users; there is no way to grant permissions to additional users or groups. POSIX Access Control Lists (ACLs)[1] remove this restriction. Each file system object is associated with a list of Access Control Entries (ACEs), which define the permissions of the file owner ID, the file group ID, additional users and groups, and others.

In the usual POSIX ACL text form, the `user::` entry stands for the file user ID, the `group::` entry stands for the file group ID, and the `other::` entry stands for others. Further, `user:<name>:` entries stand for additional users and `group:<name>:` entries stand for additional groups with the specified names.

In POSIX.1 terms, POSIX ACLs are an additional file access control method. The Working Group has also made use of the provision that the file group class may include other implementation-defined members by assigning the additional user and group entries to this class. This raises the following questions:

1. As an additional file access control mechanism, POSIX ACLs may only further restrict the access permissions defined by the file permission bits. But since the additional user and group entries are members of the file group class, what if they grant permissions beyond the file group class permissions?

   The Working Group has answered this question by defining that the file group class permissions act as an upper bound or "mask" to the group class. An entry in the group class may include permissions which are not in the file group class permissions, but only permissions which are in the entry as well as in the file group class permissions are effective.

2. If the file group class permissions continue to define the permissions of the file group ID, how can additional users and groups be granted more permissions than the file group ID, since by definition of additional file access control mechanisms, the file group class cannot have permissions beyond the file group class permissions?

   This question has been answered by defining that the file group class permissions no longer define the file group ID permissions. Instead, in POSIX ACLs, the `group::` entry stands for the file group ID, and the new `mask::` entry stands for the file group class.[2]

   The file group ID entry remains a member of the file group class.

Figure 2 shows the relationship between file classes, ACEs, and the file permission bits: the file owner class contains exactly one ACE, the file group class contains

[1] POSIX.1e [8] was never ratified as a standard. The POSIX ACL implementations found on UNIX-like operating systems are based on drafts of the POSIX.1e Working Group.

[2] If an ACL contains no entries for additional users or groups, the group class only contains a single entry. In this case, the Working Group has defined that the `group::` entry shall continue to refer to the file group class permission bits and no `mask::` entry shall exist, resulting in the same behavior as without POSIX ACLs.

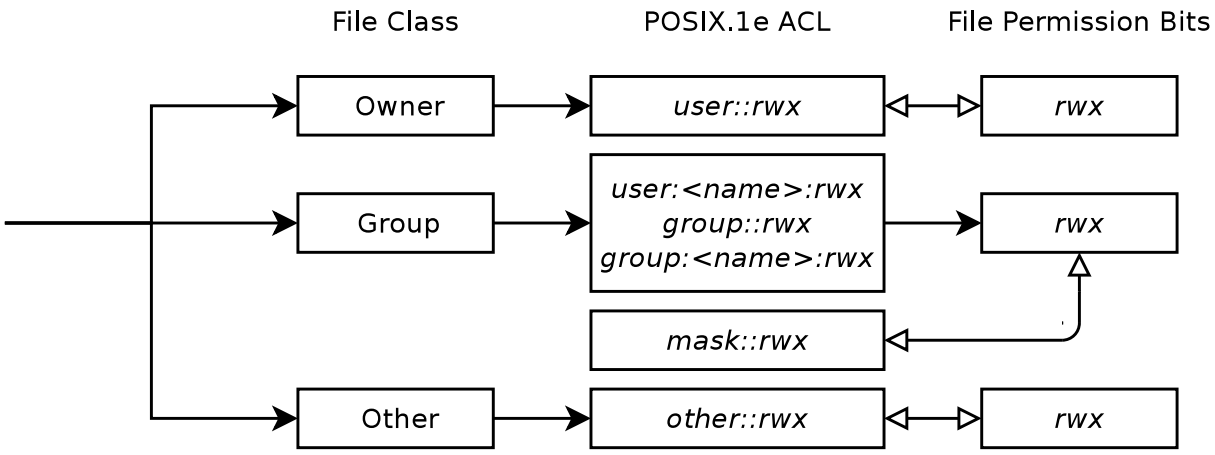

Figure 2: POSIX.1e Access Control Lists

one or more ACEs, and the file other class again contains exactly one ACE.

The open-headed arrows in Figure 2 show how ACEs and the file permission bits are kept in sync: per definition, the `user::` entry and owner class, `mask::` entry and group class, and `other::` entry and other class file permission bits are kept identical; changing the ACL changes the file permission bits and vice versa.

As an example, consider the following POSIX ACL as shown by the getfacl utility (line numbers added by hand):

```
1  # file: f
2  # owner: lisa
3  # group: users
4  user::rw-
5  user:joe:rwx       #effective:rw-
6  group::r-x         #effective:r--
7  mask::rw-
8  other::---
```

The first three lines indicate that the file is called `f`, the file user ID is `lisa`, and the file group ID is `users`. Line 4 shows that the owner, Lisa, has read and write access. Line 5 shows that Joe would have read, write, and execute access, but the file mask in line 7 forbids execute access, so Joe effectively only has read and write access. Line 6 shows that the group Users would have read and execute access, but effectively only has read access. Finally, line 8 shows that others have no access.

In addition to these "normal" ACLs, POSIX.1e also defines so-called default ACLs which have the same structure as "normal" ACLs. When a file system object is created in a directory which has a default ACL, the default ACL defines the initial value of the object's "normal" ACL and, if the new object is a directory, also the object's default ACL. Default ACLs have no effect after file creation.

2.3 Windows ACLs

Before Windows NT, Windows did not have an ACL model and permissions could only be attached to an entire exported directory tree (aka share). Other companies tried to fill this gap by inventing additional file permission schemes [14]. With the introduction of the NTFS filesystem in 1993, Microsoft introduced a new ACL based file permission model, commonly referred to as Windows ACLs or CIFS ACLs. Its key properties are:

- Windows ACLs are used to control access to a variety of OS objects in addition to filesystem objects, e.g. mutexes, semaphores, window system objects, and threads. Here we are concerned only with ACLs on filesystem objects.
- Controlled Windows objects have two ACLs, the DACL (Discretionary ACL) and SACL (System ACL). The SACL can only be edited by privileged users, and is used to implement logging of file accesses (or failures to access). Here we are concerned only with DACLs.
- Each filesystem object is owned by a Security Identifier (SID), a variable-length unique binary identifier which can refer to either a user or a group.

- Each Windows ACE contains a mask of 14 different permission bits, see Table 1 for a summary and File and Folder Permissions [2] on Microsoft TechNet for details.

- Windows ACEs are one of three types: Access-Denied (used in a DACL to explicitly deny access), Access-Allowed (used in a DACL to explicitly grant access), and System-Audit (used in a SACL to cause an entry to be made in the system security log). We shall refer to these by the short forms `DENY`, `ALLOW`, and `AUDIT` respectively.

- The order in which permissions are granted and denied in Windows ACLs matters. ACEs are processed from top to bottom until all requested permissions have been granted, or a requested permission has been explicitly denied.

- Each ACE contains a SID which identifies the user or group that the ACE refers to. There is no way to tell user from group ACEs. There are also some special SIDs with special semantics.

- The owner of a file can be explicitly mentioned in an ACE, and implicitly mentioned in an inherit-only ACE.[3] However, there is no way to construct an ACE that always applies to the file's current owner even if the owner is changed.[4]

- A Windows ACL entry can apply to the special SID `Everyone`, which includes the owner and all users and groups explicitly mentioned in other ACL entries. The POSIX.1 concept of file classes, where a process is classified into the owner, group, and other classes and cannot obtain permissions which go beyond the permissions of its class, does not exist, but `DENY` entries can be used to explicitly deny some permissions.

- Windows supports inheritance of permissions at file create time, and since Windows 2000, a feature called "Automatic Inheritance." With Automatic Inheritance, changes to the permissions of a directory can propagate to all files and directories below that directory. Create time inheritance and Automatic Inheritance use as a number of ACE flags (`INHERITED_ACE`, `INHERIT_ONLY_ACE`, `CONTAINER_INHERIT_ACE`, `OBJECT_INHERIT_ACE`, and `NO_PROPAGATE_INHERIT_ACE`).

[3] using the special *Creator Owner* SID

[4] not even using the special *Owner Rights* SID which appeared in Windows Vista [4]; such ACEs are actually disabled when the file's ownership changes.

2.4 NFSv4 ACLs

Up to version 3, the NFS protocol was mainly UNIX oriented, and its file permission model was limited to exposing POSIX file modes (in NFS terminology: the mode attribute) over the network. This created the implicit assumption of POSIX-like behavior.

Version 4 broke with the protocol's legacy and introduced a new ACL model based on Windows ACLs. The mode attribute was initially deprecated in favor of ACLs; more recent updates to the protocol re-endorsed the mode attribute and clarified some of the interactions between mode and ACLs (but some inconsistencies still remain).

The key properties of the NFSv4 ACL model are:

- NFSv4 [5] supports the same 14 permissions as Windows. NFSv4.1 [6] adds two additional permissions, `ACE4_WRITE_RETENTION` and `ACE4_WRITE_RETENTION_HOLD`, which have no equivalent in Windows or POSIX ACLs.

- The `ALLOW` and `DENY` ACE types are supported, and optionally also the `AUDIT` and `ALARM` types.

- The NFSv4 and Windows permission check algorithms are equivalent.

- NFSv4 uses principal strings modelled on Kerberos for identifying users and groups, e.g. `fred@example.com`. User and group ACEs are distinguised by an ACE flag.

- Each file system object is owned by a user, and has an owning group. The special `OWNER@` principal refers to the current owner, and the special `GROUP@` principal refers to the current owning group of a file, even when the owner or owning group changes.

- The special `EVERYONE@` principal refers to Everyone, which includes the owner and all users and groups explicitly mentioned in other ACL entries.

- NFSv4 recognizes that there is a relationship between the mode attribute and the ACL, but instead of connecting this to the POSIX file permission model and sticking to the same requirements, NFSv4.1 makes up its own special rules for updating the ACL when the mode changes, and vice versa. These rules are not fully compatible with POSIX.1 or POSIX.1e.
- NFSv4 supports inheritance of permissions at file create time. NFSv4.1 adds support for Automatic Inheritance.

2.5 ACL Model Differences

The various ACL models differ in a number of important details.

- POSIX1.e entries can only allow access, i.e. `ALLOW` semantics. Windows and NFSv4 entries can either `ALLOW` or `DENY` access. This provides considerably more expressive power (albeit at the cost of complexity).
- The order of evaluation of POSIX.1e entries is not significant, as all the entries are additive. In Windows and NFSv4 ACLs, entries can deny access, and thus their order is significant.
- The permission bits in POSIX.1e entries are quite simple, with only 3 bits defined. Windows has a total of 14 permission bits, most of which affect a smaller number of actions. NFSv4 follows Windows closely (even when the permission bits make no sense), but NFSv4.1 adds two more permission bits (`ACE4_WRITE_RETENTION_HOLD` and `ACE4_WRITE_RETENTION`) which have no equivalent anywhere else.
- POSIX.1e and NFSv4 ACEs have state which determine whether the ACE refers to a user or a group, because in POSIX these are strictly different namespaces. Windows ACEs contain only a SID, which might refer to either a user or a group.
- The POSIX.1e model reuses the POSIX.1 concept of process file classes, where a process is classified into the owner, group and other classes. The Windows model has no precise equivalent to any of these classes. A Windows ACE can apply to Everyone, but unlike the POSIX "other" class, that includes the owner and all users explicitly mentioned in other ACEs. The NFSv4 model compromises between the two, defining special `OWNER@` and `GROUP@` principals with the POSIX semantics, and `EVERYONE@` with the Windows semantics.
- The possible existance of `DENY` entries in Windows and NFSv4 models, and the differences in file class boundaries, also complicate the permission algorithm compared to POSIX.1e. The more complex algorithm must track both allowed and denied masks.
- The POSIX.1e model allows for inheritance of ACLs from a parent directory at the time when a filesystem object is created, using a separate "default ACL" stored on the parent directory. By contrast, the Windows, NFSv4 models combine the actual and default ACLs into one, with extra flags on each ACE to indicate whether it is to be inherited or not, and whether it is to be used only for inheritance.
- The POSIX.1e model has no explicit support for recursive modifications of ACLs on existing trees; like `chmod -R` this is expected to proceed entirely in a userspace utility which recurses over a tree and modifies ACLs. The Windows and NFSv4 models also rely on a userspace utility but have more complex ACE propagation algorithms (Automatic Inheritance) which need some extra bits stored on the ACL. The NFSv4 standard forgot these bits, and they only appear in NFSv4.1.
- In addition to the `ALLOW` and `DENY` types of entries, the Windows and NFSv4 models allow for `AUDIT` and `ALARM` entries. In Windows, these entries trigger system management side effects. In NFSv4, their meaning is undefined.
- The POSIX.1e model identifies users and groups using traditional UNIX user and group ID numbers. The Windows model uses SIDs. which are something like binary hierarchically scoped user and group IDs and provide a single namespace for both users and groups. The NFSv4 model uses principal strings modelled on Kerberos, e.g. `fred@example.com`.

3 Why We Need a New ACL Model

There are two main reasons why we consider new ACL model for Linux to be necessary.

Firstly, while POSIX.1e is the current default ACL model on Linux, and works well, its power and expressiveness is limited by its small set of permission bits and additive `ALLOW`-only semantics. Subtractive concepts like "grant read permission to all of the accounting department, but not to the trainees" are difficult to achieve and have to be painfully approximated. Windows can express these neatly and concisely, and we feel this will be useful to Linux system administrators.

Secondly, in a file serving scenario, there is a significant interface mismatch between POSIX ACLs and the ACL models expected or provided by Windows systems. This mismatch makes interoperability between Linux and Windows machines unnecessarily difficult, requiring the Linux-side software to perform complex, lossy, and potentially insecure mappings backwards and forwards between the models.

When files are available through different channels (e.g. Samba and NFS on the same Linux server) with different approaches to the ACL issue, there is the potential for users to be able to bypass intended access controls.

Even in the homogeneous case when a Linux client mounts a filesystem via NFSv4 from a Linux server, and both Linux systems understand POSIX ACLs, the ACL model enforced by the NFSv4 protocol requires two difficult mappings in order to transmit an ACL on the wire. The Linux NFS client presents ACLs using different formats and utilities than those used for the local filesystem on the Linux server, which leads to unnecessary confusion.

We need a solution which makes the ACL models of the client, the server and the protocol as similar as possible, so that mappings between them are much easier and safer. It should also provide a single point of ACL enforcement for all protocols and for local applications, and consistent management tools on the client and server.

4 Rich ACLs

To bring together the disparate models and address interoperability issues, we propose a new ACL mechanism for Linux called Rich-acl.

4.1 Design Principles

The proposed new ACL model uses NFSv4 ACLs at its core. Unlike NFSv4 and Windows ACLs, it identifies users and groups by their numeric UNIX IDs. This allows access decisions to be made for all Linux processes without having to translate identifiers to their local form first.

The `ALLOW` and `DENY` ACE types are supported; support for `AUDIT` and `ALARM` type ACEs might make sense to add in the future.

Rich-acl supports the same 14 permission bits as NFSv4 (three of which have a dual meaning and mnemonic for files and directories) plus the two additional write retention permissions of NFSv4.1. The permissions have the following meaning:

- `READ_DATA`, `WRITE_DATA`, `APPEND_DATA`: Read a file, modify a file, and modify a file by appending to it only.
- `LIST_DIRECTORY`, `ADD_FILE`, `ADD_SUBDIRECTORY`: List the contents of a directory, add files, and add subdirectories.
- `DELETE_CHILD`: Delete a file or subdirectory from a directory.
- `EXECUTE`: Execute a file, traverse a directory.
- `READ_ATTRIBUTES`: Read the stat information of a file or directory.
- `READ_ACL`: Read the ACL of a file or directory.
- `SYNCHRONIZE`: Synchronize with another thread by waiting on a file handle.
- `DELETE`: Delete a file or directory even without the `DELETE_CHILD` permission on the parent directory.
- `WRITE_ATTRIBUTES`: Set the access and modification times of a file or directory.
- `WRITE_ACL`: Set the ACL and POSIX file mode of a file or directory.
- `WRITE_OWNER`: Take ownership of a file or directory. Set the owning group of a file or directory to

the effective group ID or one of the supplementary group IDs.[5]

- READ_NAMED_ATTRS, WRITE_NAMED_ATTRS: Read and write Named Attributes. Named Attributes neither refer to Windows Alternate Data Streams nor to Linux Extended Attributes. These permissions will be stored, but have no further effect.

- WRITE_RETENTION, WRITE_RETENTION_HOLD: Set NFSv4.1 specific retention attributes. These permissions will be stored, but have no further effect.

Some of the Rich-acl permissions are a subset of a POSIX permission; others go beyond what the file permission bits can grant. Table 1 shows a complete mapping between Rich-ACL and POSIX permissions:

- The READ_DATA and LIST_DIRECTORY permissions map to the POSIX Read permission, the WRITE_DATA, APPEND_DATA, DELETE_CHILD, ADD_FILE, and ADD_SUBDIRECTORY permissions map to the POSIX Write permission, and the EXECUTE permission maps to the POSIX Execute/Search permission. These permissions fit the concept of an additional file access control mechanism.

- The READ_ATTRIBUTES, READ_ACL, and SYNCHRONIZE permissions are permissions which cannot be denied under POSIX. Denying these operations could cause problems with POSIX applications, so we always grant these permissions no matter what the ACL says (see section 5.4).

- The DELETE, WRITE_ATTRIBUTES, WRITE_ACL, and WRITE_OWNER permissions denote rights which go beyond the POSIX permissions, and the READ_NAMED_ATTRIBUTES, WRITE_NAMED_ATTRIBUTES, WRITE_RETENTION, and WRITE_RETENTION_HOLD permissions denote rights which have no equivalent in Linux. These eight permissions can only be enabled as part of an alternate file access control mechanism.

[5] Also see the setfsuid(2) and setfsgid(2) Linux manual pages.

In addition to defining how the permissions of the two models map onto each other, we need to define how processes are classified into the owner, group, and other classes; this determines which file permission bits affect which processes. We use the following rules analogously to POSIX ACLs:

1. Processes are in the owner class if their effective user ID matches the user ID of the file.

2. Processes are in the group class if they are not in the owner class and their effective group ID or one of the supplementary group IDs matches the group ID of the file, the effective user ID matches the user ID of an ACE, or the effective group ID or one of the supplementary group IDs matches the group ID of an ACE.

3. Processes are in the other class if they are not in the owner or group class.

Finally, POSIX requires that after creating a new file or changing a file's permission bits with the chmod system call, processes are not granted any permissions beyond the file permission bits of their file class. This requirement can be implemented in different ways:

1. The ACL can be replaced by an ACL which grants the equivalent of the file permission bits to the owner, the owning group, and others.[6]

2. The ACL can be changed so that it does not grant any permissions beyond the file permission bits. This may require removing permissions from ACEs. In addition, if the owner class has fewer permissions than the group class or the group class has fewer permissions than the other class, additional DENY ACEs may be needed.[7]

3. The ACL can be left unchanged; in this case, the access check algorithm must take both the ACL and the file permission bits into account, and only grant permissions which are granted by *both* mechanisms.

[6] NFSv4 ACLs on IBM GPFS and JFS2 do this, Sun/Oracle ZFS offers this as an option.

[7] SUN/Oracle ZFS tries to do this by default, but the documented behavior [10] does not always lead to the correct result.

Permission Bit	POSIX Mapping
READ_DATA (= LIST_DIRECTORY)	Read
WRITE_DATA (= ADD_FILE)	Write
APPEND_DATA (= ADD_SUBDIRECTORY)	Write
DELETE_CHILD	Write
EXECUTE	Execute/Search
READ_ATTRIBUTES	Always Allowed
READ_ACL	Always Allowed
SYNCHRONIZE	Always Allowed
DELETE	Alternate
WRITE_ATTRIBUTES	Alternate
WRITE_ACL	Alternate
WRITE_OWNER	Alternate
READ_NAMED_ATTRS	Alternate (No Effect)
WRITE_NAMED_ATTRS	Alternate (No Effect)
WRITE_RETENTION	Alternate (No Effect)
WRITE_RETENTION_HOLD	Alternate (No Effect)

Table 1: Mapping Between Rich-ACL and POSIX Permissions

We have chosen a variation of approach 3 for Rich-acls because it does not require complicated ACL manipulations, and is a consequent adaptation of the masking mechanism already found in POSIX ACLs, which has already proven itself.

The need for a variation to approach 3 becomes obvious when considering that the file permission bits are limited to the Read, Write, and Execute/Search permissions, and we would end up without a way to explicitly enable any of the alternate rich-acl permissions.

To get around this restriction, we introduce the new concept of file masks:

- Each file class (owner, group, and other) is associated with a file mask which contains a set of rich-acl permissions.
- When the file permission bits are changed, each file mask is set to the rich-acl permissions which correspond to the file permission bits of its class.
- The file masks can be changed explicitly to include alternate rich-acl permissions. Changing the file masks will also change the file permission bits.
- The access check algorithm grants an access if the rich-acl grants the access, and the file mask matching the process also includes the reuested rich-acl permisssions.

Figure 3 shows the relationship between file classes, the ACL, the file masks, and the file permission bits in rich-acls.

4.2 Specific Changes

To achieve the above principles, we made the following code changes.

- Modify the kernel VFS interface to allow filesystems optionally to exert more control over whether a process is allowed to create or delete filesystem objects. The rich-acl permission algorithm requires more information in these two cases than either the POSIX or POSIX.1e models do.
- Define a machine-independent binary encoding of a rich-acl ACL, and use the Linux extended attribute (xattr) mechanism to store encoded rich-acl ACLs on filesystem objects. The same encoding is used in the filesystem and on both the NFS client and server (which is not true of the current Linux NFS ACL code). The attribute used is `system.richacl`.
- Provide an in-kernel library for manipulating ACLs. This includes creating and destroying ACLs, performing permission checks, calculating a file mode from an ACL and applying a new mode

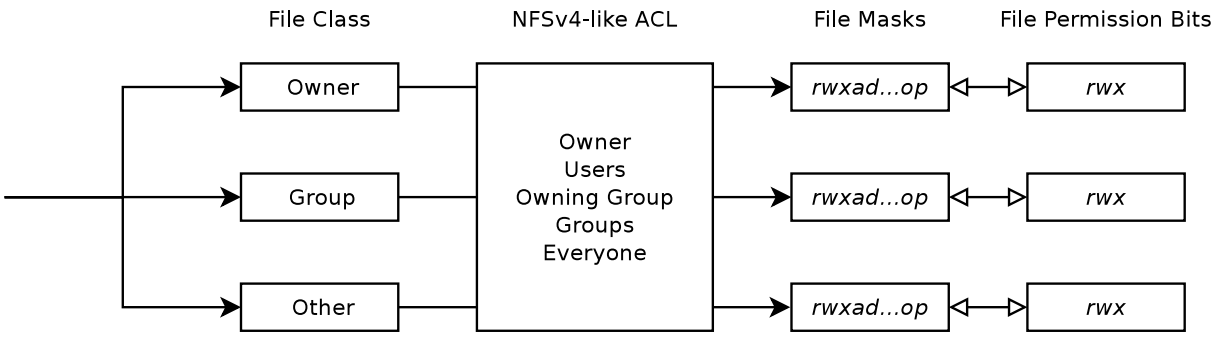

Figure 3: Rich Access Control Lists

to an ACL. and encoding an ACL to an xattr and decoding an ACL from a xattr.

- Use the kernel library to enhance the ext4 filesystem to store, retrieve and enforce the new permission model. A new ext4 superblock option `richacl`, settable with the `tune2fs` utility, is defined to control whether rich-acls are enabled.

- Use the kernel library to enhance the NFS client and server to store and retrieve rich-acl ACLs (enforcement is done in the server-side backing filesystem, not in NFS code). The server-side conversion between the rich-acl kernel in-memory format and the NFS wire format is much simpler than with POSIX.1e ACLs.

- Provide a userspace library for manipulating ACLs. It is similar to the kernel library, except that it does not provide a permission check algorithm.

- Use the userspace library to provide a command-line utility `setrichacl` to allow users to store, retrieve, and manipulate ACLs on files and directories (loosely equivalent to `chmod` and `ls`).

One of the advantages of this approach is consistency of use: the same tools can be used to examine and manipulate rich-acl ACLs on the NFS client and server, as well as for local applications.

Furthermore, with the rich-acl model ACLs are stored and enforced consistently and in one place: the server-side backing filesystem. This prevents users being able to evade access control by using different access techniques, such as logging into the server or using NFS instead of CIFS.

The code is available in two git repositories, kernel [12] and userspace [13]. There is also a patch for `tune2fs` [11].

5 Further Considerations

5.1 Standards

The text of the NFSv4 standard, as it applies to ACLs, has undergone several revisions and clarifications. The initial version appears to have been the result of a purely theoretical design exercise and not of implementation experience. Subsequent versions have had progressively fewer flaws and ambiguities, but some difficulties remain.

The behaviour of Windows ACLs is well documented by Microsoft. Sometimes the documentation is accurate; sometimes experiment is required to determine the true behaviour.

5.2 Multiple group entries

POSIX allows a user to be a member of multiple groups. The POSIX.1e model allows access if any one of the groups that the user belongs to, is allowed access. In contrast, rich-acl ACEs are processed in order. If the requested access mask bit matches the access mask bit present in a `DENY` ACE, then the access is denied. Each ACE is processed until all the bits of the requested access mask are allowed. This implies that when mapping a POSIX ACL to rich-acl we need to impose an ordering constraint on ACEs, such that ACEs which `ALLOW` access to any group must preceed ACEs which `DENY` access to any group.

5.3 OTHER vs EVERYONE@ ACEs

One of the major differences between the POSIX.1e model and the rich-acl model is that the rich-acl EVERYONE@ includes both user and group classes, whereas the POSIX.1e OTHER class excludes user and group classes. When mapping a POSIX.1e ACL to a rich-acl, the OTHER ACE will be mapped to a trailing ALLOW EVERYONE@ ACE, but to limit it's effect that ACE may need to be preceeded with one or more DENY ACEs which deny some access to specific groups.

5.4 Permissions which are always enabled

The NFSv4 ACL model has some permission bits which control actions which are always allowed under POSIX, such as ACE4_READ_ATTRIBUTES for reading file attributes ACE4_READ_ACL for reading the ACL itself. To limit impact on the Linux code (for example, by introducing new error cases in complex and critical code paths) and on existing POSIX applications we have chosen not to enforce these permission bits in the rich-acl model. In line with our design goals, ACEs which mention these permission bits will be accepted and stored, but the permission bits will have no effect.

One effect of this choice is that the NFS server will successfully complete a SETATTR operation which sets an ACL containing an ACE intended to DENY these permissions, despite not being able to accurately enforce the intended effect of the ACE. Such behaviour is prohibited by language in the NFSv4.1 RFC [6], which specifies that the NFS server must return the NFS4ERR_ATTRNOTSUPP error in this case.

Our experiments show that it is very easy for a user using the Windows permission editor GUI to set such an ACE, as an unintended side effect of the common idiom of denying another user the ability to read file data. This is due to the editor having "basic" and "advanced" modes; in the basic mode the user is presented with abstracted permissions like Read which are amalgams of multiple underlying permissions. So if our implementation were to strictly obey the RFC then Windows users would be unnecessarily inconvenienced and possibly Windows applications might be broken.

5.5 Sticky bit and capabilities

When describing the POSIX model above, we did not mention some of the more complex corners of the model. To meet our design goal of preserving expected POSIX behaviour, the rich-acl permission algorithm needs to take these into account.

The POSIX sticky(t) bit is used in the file mode of a directory to change the permission check for deleting files in the directory. It is usually employed to allow multiple users to share the /tmp directory in such a way that each user can delete only her own files. Such behaviour could be approximated with a well designed ACL on the /tmp directory; however our design goal meant that we need to preserve the behaviour of the sticky bit regardless of the presence of ACLs. A further reason is the security risk involved with perturbing the behaviour of /tmp.

POSIX defines capabilities CAP_DAC_OVERRIDE and CAP_DAC_READ_SEARCH which allow privileged processes to gain access regardless of the results of an access control check (with some limits). Another capability, CAP_FOWNER, allows privileged processes to gain access normally allowed only to the owner of an object and not subject to the POSIX DAC controls. Of course, CAP_FOWNER interacts with the implementation of the sticky bit.

5.6 Migration

Many filesystems using POSIX ACLs are already deployed. Therefore, in order to enable rich-acl on an existing Linux file system, we need to provide a mechanism for migrating the filesystem from existing POSIX.1e ACLs to rich-acl ACLs.

The current design has a two-step conversion process:

- In the first step, the kernel filesystem code constructs a temporary rich-acl ACL on the fly when an ACL on a filesystem object is required (for a permissions check or an xattr fetch), and the filesystem has the richacl option enabled with tunefs, and the object has no rich-acl ACL stored, and the object has a POSIX.1e ACL stored. Once the richacl option is enabled, objects in the filesystem appear to have both a POSIX.1e ACL and a functionally equivalent rich-acl ACL.
- The converted ACL is discarded after use, and not written back to the filesystem object. Hence we need a second step to complete the conversion: the

`setrichacl` utility reads the temporary rich-acl from the xattr and writes the same bytes back to the xattr, causing the filesystem to make the rich-acl permanent on disk. As a side effect of setting the rich-acl xattr, the kernel deletes the POSIX.1e ACL xattr.

The advantage of this technique is that the filesystem is immediately available with functioning rich-acls after the `richacl` option is enabled without requiring any modification of the on-disk metadata. This also allows experimentally enabling rich-acls on a filesystem in order to test application compatibility, enabling rich-acls on a readonly filesystem, and more easily fine-tuning the conversion algorithm used in the user-space utility used if needed.

Note that migrating a filesystem back from rich-acls to POSIX.1e ACLs is not supported once the rich-acls are made permanent on disk, as converting in that direction is usually lossy.

6 Open Issues / Future Work

Rich-acls are usable today by kernel developers and early adopters, but there is work remaining to be done.

- Use the userspace rich-acl library to enhance the Samba server to store and retrieve rich-acl ACLs (enforcement is done in the server-side backing filesystem, not in Samba code). The conversion between the rich-acl userspace in-memory format and the CIFS wire format is much simpler than with POSIX.1e ACLs. This could be based on the existing SGI [9] patch.
- Use the kernel rich-acl library to enhance the smbfs client to store and retrieve rich-acl ACLs.
- The `ls` program should indicate those filesystem objects which have rich-acls, for example by showing a + sign.
- The `find` program should be aware of ACLs, at the very least by providing a predicate which tests whether an object has a rich-acl. Even better would be predicates to test more subtle effects of rich-acls.
- The `setrichacl` utility should be updated to provide a convenient one-line command to perform the second step in the process of migrating a filesystem from POSIX.1e ACLs to rich-acl ACLs. In this mode it would traverse the filesystem tree, fetching the rich-acl xattr and setting it back again, causing the rich-acl ACL to become permanent on disk. Currently the program must be invoked twice per file or directory.
- The GNOME and KDE desktops need GUI applications written to allow users to display and edit rich-acls on filesystem objects without resorting to the command-line interface.
- The issue of user identity is not completely resolved. Linux, NFSv4 and Windows all use different unique identifiers for users. Rich-acls use Linux user ids, which require mapping to and from Windows SIDs or NFSv4 principals on demand. The mechanisms for such mappings can be awkward and slow. It may be useful to investigate storing SIDs and principals in the filesystem.
- It might be useful to implement Windows/NFSv4 SACLs and the `AUDIT` and `ALARM` ACE types.
- The Linux kernel NFSv4 server places a smaller limit on the maximum size of NFSv4 ACLs than does either the rich-acl implementation or the NFSv4 standard. Fixing this properly requires significant surgery to the NFSv4 XDR code.
- The `setrichacl` utility does not yet perform Automatic Inheritance.
- The POSIX API does not provide an interface for an application to atomically create a file or other filesystem object with a given set of extended attributes; this needs to be performed as two separate actions. Both the CIFS and NFSv4 protocols do however provide such an ability. This creates a race condition where a filesystem object may briefly have an unintended ACL and be less secure than the application expected. Such an interface could be provided to allow the Samba server to avoid the race.
- The POSIX `access` function and the NFSv4 `ACCESS` operation could be enhanced to allow a Rich-acl-aware application to test for more fine-grained access types.

- Windows Vista introduced a feature that allows to deny file owners the *Read Permissions* and *Change Permissions* permissions which they are otherwise implicitly granted [4]; support for this has not been implemented, yet.

7 Conclusion

The demand for improved interoperability in modern, mixed-platform environments is increasing over the years. On UNIX-like systems, the most widely available kind of ACLs is POSIX ACLs, and they will remain in that role for some time to come. Still, POSIX ACLs have proven unsuitable for addressing the interoperability challenges we are facing today.

In this paper we have discussed the many goals that a better, more interoperable ACL model should meet. The proposed new model meets those goals. A lot of work still remains to be done at all levels until end users will be able to reap the benefits, but all the key building blocks are there already.

8 Acknowledgements

The authors would like to thank IBM, Novell, and SGI for supporting this work and its predecessors at various times. We're also grateful for technical support from various members of the NFS Working Group, and the NFS and CIFS communities. David Disseldorp has reviewed this paper.

Legal Statement

References

[1] IBM Corp. Working with filesystems using NFSV4 ACLs. `http://www.ibm.com/developerworks/aix/library/au-filesys_NFSv4ACL/index%.html`.

[2] Microsoft Corp. File and folder permissions. `http://technet.microsoft.com/en-us/library/cc732880.aspx`.

[3] Microsoft Corp. How security descriptors and access control lists work. `http://technet.microsoft.com/en-us/library/cc781716.aspx`.

[4] Microsoft Corp. Security Identifiers (SIDs) new for Windows Vista. `http://technet.microsoft.com/en-us/library/cc749445%28WS.10%29.aspx`.

[5] IETF Network Working Group. RFC 3530: Network File System (NFS) version 4 protocol. `http://tools.ietf.org/html/rfc3530`.

[6] IETF Network Working Group. RFC 5661: Network File System (NFS) version 4 minor version 1 protocol. `http://tools.ietf.org/html/rfc5661`.

[7] The Open Group. Portable Operating System Interface. `http://www.unix.org/version3/`.

[8] The Open Group. What could have been IEEE 1003.1e/2c. `http://wt.tuxomania.net/publications/posix.1e/download.html`.

[9] Silicon Graphics Inc. Native NFSv4 ACLs for Linux XFS & NFS. `http://oss.sgi.com/projects/nfs/nfs4acl/`.

[10] Sun Microsystems Inc. Solaris ZFS administration guide. http://dlc.sun.com/pdf/819-5461/819-5461.pdf.

[11] Aneesh Kumar K.V. Rich-acl e2fsprogs patch. http://kernel.org/pub/linux/kernel/people/kvaneesh/richaclv1/e2fsprogs/%.

[12] Aneesh Kumar K.V. Rich-acl kernel git repo. http://git.kernel.org/?p=linux/kernel/git/kvaneesh/linux-richacl.git;a=%summary.

[13] Aneesh Kumar K.V. Rich-acl userspace git repo. http://git.kernel.org/?p=fs/acl/kvaneesh/acl.git;a=summary.

[14] Novell, Inc. Netware 6 trustee rights: How they work and what to do when it all goes wrong. http://support.novell.com/techcenter/articles/ana20030202.html.

[15] W. Richard Stevens. *Advanced Programming in the UNIX(R) Environment*. Addison-Wesley, June 1991.

Proceedings of the Linux Symposium

July 13th–16th, 2010
Ottawa, Ontario
Canada

Conference Organizers

Programme Committee

Proceedings Committee