Network Working Group
Internet-Draft
Expires: February 23, 2007

M. Eriksen
J. Fields
CITI

August 22, 2006

Mapping Between NFSv4 and Posix Draft ACLs
draft-ietf-nfsv4-acl-mapping-05

Abstract

A number of filesystems and applications support ACLs based on a withdrawn POSIX draft [2]. Those ACLs differ significantly from NFS version 4 ACLs [1]. We describe how to translate between the two types of ACLs.

1. Introduction

Access Control Lists (ACLs) are used to specify access rights to file system objects. An ACL is a list of Access Control Entries (ACEs), each specifying an entity (such as a user) and some level of access for that entity.

In the following sections describe NFSv4 ACLs and ACLs based on a withdrawn POSIX draft. We will refer to the latter as "POSIX ACLs". Since NFSv4 ACLs are more fine-grained than POSIX ACLs, it is not possible in general to map an arbitrary NFSv4 ACL to a POSIX ACL with the same semantics. However, it is possible to map any POSIX ACL to a NFSv4 ACL with nearly identical semantics, and it is possible to map any NFSv4 ACL to a POSIX ACL in a way that preserves certain guarantees. We will explain how to do this, and give guidelines for clients and servers performing such translation.

2. NFSv4 ACLs

An NFSv4 ACL is an ordered sequence of ACEs, each having an entity, a type, some flags, and an access mask.

The entity may be the name of a user or group, or may be one of a small set of special entities. Among the special entities are "OWNER@" (the current owner of the file), "GROUP@" (the group associated with the file), and "EVERYONE@".

An ACL may have a "type" of ALLOW or DENY. (AUDIT or ALARM are also allowed, but they are not relevant to our discussion).

The access mask has 14 separate bits, including bits to control read, write, execute, append, ACL modification, file owner modification, etc.; consult [1] for the full list.

Of the flags, four are relevant here. The ACE4_IDENTIFIER_GROUP flag is used to indicate that the entity name is the name of a group. The other three concern inheritance: ACE4_DIRECTORY_INHERIT_ACE indicates that the ACE should be added to new subdirectories of the directory; ACE4_FILE_INHERIT_ACE does the same for new files; and ACE4_INHERIT_ONLY indicates that the ACE should be ignored when determining access to the directory itself.

The NFSv4 ACL permission-checking algorithm is straightforward. First, assume a requester asks for access specified by a single bit in the access bitmask. We allow the access if the first ACE in the ACL that matches the requester and that has that bit set is an ALLOW ACE, and we deny the access if the first such ACE is a DENY ACE. If no matching ACE has the bit in question set, access is normally denied.

If a requester asks for access requiring multiple bits from the access bitmask simutaneously, then we allow the access if and only if each bit in the requested bitmask would be allowed individually.

We refer the reader to [1] for further details.

3. POSIX ACLs

A number of operating systems implement ACLs based on the withdrawn POSIX 1003.1e/1003.2c Draft Standard 17 [2]. We will refer to such ACLs as "POSIX ACLs", though they are not part of any published POSIX standard.

POSIX ACLs use access masks with only the traditional "read", "write", and "execute" bits. Each ACE in a POSIX ACL is one of five types: ACL_USER_OBJ, ACL_USER, ACL_GROUP_OBJ, ACL_GROUP, ACL_MASK, and ACL_OTHER. Each ACL_USER ACE has a uid associated with it, and each ACL_GROUP ACE has a gid associated with it. Every POSIX ACL must have exactly one ACL_USER_OBJ, ACL_GROUP_OBJ, and ACL_OTHER ACE, and at most one ACL_MASK ACE. The ACL_MASK ACE is required if the ACL has any ACL_USER or ACL_GROUP ACEs. There may not be two ACL_USER ACEs with the same uid, and there may not be two ACL_GROUP ACEs with the same gid.

Given a POSIX ACL and a requester asking for access, permission is determined by consulting the ACEs in the order ACL_USER_OBJ, ACL_USER, ACL_GROUP_OBJ, ACL_GROUP, ACL_OTHER, and allowing or denying access based on the first ACE encountered that the requester matches, except that we never allow the ACL_USER, ACL_OWNER_OBJ, or ACL_GROUP objects to grant more than the ACL_MASK object does, and in the case of ACL_GROUP_OBJ and ACL_GROUP ACEs, we allow access if any one of those ACEs allows access.

In more detail:

1. If the requester is the file owner, then allow or deny access depending on whether the ACL_USER_OBJ ACE allows or denies it. Otherwise,

2. if the requester matches the file's group, and the ACL mask ACE would deny the requested access, then skip to step 5. Otherwise,

3. if the requester's uid matches the uid of one of the ACL_USER ACEs, then allow or deny access depending on whether the ACL_USER_OBJ ACE allows or denies it. Otherwise,

4. Consider the set of all ACL_GROUP ACEs whose gid the requester is a member of. Add to that set the ACL_GROUP_OBJ ACE, if the requester is also a member of the file's group. Allow access if any ACE in the resulting set allows access. If the set of matching ACEs is nonempty, and none allow access, then deny access. Otherwise, if the set of matching ACEs is empty,

5. if the requester's access mask is allowed by the ACL_OTHER ACE, then grant access. Otherwise, deny access.

Each file may have a single POSIX ACL associated with it, used to determine access to that file. Directories, however, may have two ACLs: one, the "access ACL", used to determine access to the directory, and one, the "default ACL", used only as the ACL to be inherited by newly created objects in the directory.

4. Ordering of NFSv4 and POSIX ACLs

POSIX ACLs are unordered--the order in which the POSIX access-checking algorithm considers the entries is determined entirely by the type of the entries, so the entries don't need to be kept in any particular order.

By contrast, the meaning of an NFSv4 ACL can be dramatically changed by modifying the order that the entries are listed in.

In the following, we will say that an NFSv4 ACL is in the "canonical order" if its entries are ordered in the order that the POSIX algorithm would consider them. That is, with all OWNER@ entries first, followed by user entries, followed by GROUP@ entries, followed by group entries, with all EVERYONE@ entries at the end.

5. A Minor Eccentrity of POSIX ACLs

We will see below that it is possible to find an NFSv4 ACL with precisely the same effect as any given POSIX ACL, with one extremely minor exception: if a requester that is a member of more than one group listed in the ACL requests multiple bits simultaneously, the POSIX algorithm requires all of the bits to be granted simultaneously by one of the group ACEs. Thus a POSIX ACL such as

```
ACL_USER_OBJ: ---
ACL_GROUP_OBJ: ---
g1: r--
g2: -w-
ACL_MASK: rw-
ACL_OTHER: ---
```

will prevent a user that is a member of groups g1 and g2 from opening a file for both read and write, even though read and write would be individually permitted.

The NFSv4 ACL permission-checking algorithm has the property that it permits a group of bits whenever it would permit each bit individually, so it is impossible to mimic this behaviour with an NFSv4 ACL.

6. Mapping POSIX ACLs to NFSv4 ACLs

6.1. Requirements

In the next section we give an example of a mapping of POSIX ACLs into NFSv4 ACLs. A server or client may use a different mapping, but the mapping should meet the following requirements:

It must map the POSIX ACL to an NFSv4 ACL with identical access semantics, ignoring the minor exception described in the previous section.

It must map the read mode bit to ACE4_READ_DATA, the write bit to ACE4_WRITE_DATA and ACE4_APPEND_DATA (and ACE4_DELETE_CHILD for directories), and the EXECUTE bit to ACE4_EXECUTE. It should also allow ACE4_READ_ACL, ACE4_READ_ATTRIBUTES, and ACE4_SYNCHRONIZE unconditionally, and allow ACE4_WRITE_ACL and ACE4_WRITE_ATTRIBUTES to the owner. The handling of other NFSv4 mode bits may depend on the implementation, but it is preferable to leave them unused.

It should avoid using DENY ACEs. If DENY ACEs are required, it should attempt to place them at the beginning. (This is not always possible.)

The resulting NFSv4 ACL must take into account the mask ACE, by ensuring that it does not give the group file owner or any users or groups named in the ACL more permissions than permitted by the mask. It would also be possible to specify a mapping that encoded the mask in such a way that the original value of the mask could be recovered by someone that knew the ACL was produced by our algorithm. However, the added complexity and fragility of such a mapping is not worth the small benefit of preserving the mask information, so we do not attempt that here.

6.2. Example POSIX->NFSv4 Mapping

We now describe an algorithm which maps any POSIX ACL to an NFSv4 ACL with the same semantics, meeting the above requirements.

First, modify all ACL_USER, ACL_GROUP, and ACL_GROUP_OBJ ACEs by removing any permissions not granted by the mask ACE. The mask ACE may then be ignored for the rest of this process.

Translate the uid's and gid's on the ACL_USER and ACL_GROUP ACEs into NFSv4 names, using directory services, etc., as appropriate, and translate ACL_USER_OBJ, ACL_GROUP_OBJ, and ACL_OTHER to the special NFSv4 names "OWNER@", "GROUP@", and "EVERYONE@", respectively.

Next, map each POSIX ACE (excepting any mask ACE) in the given POSIX ACL to an NFSv4 ALLOW ACE with an entity determined as above, and with a bitmask determined from the permission bits on the POSIX ACE as follows:

1. If the read bit is set in the POSIX ACE, then set ACE4_READ_DATA.

2. If the write bit is set in the POSIX ACE, then set ACE4_WRITE_DATA and ACE4_APPEND_DATA. If the object carrying the ACL is a directory, set ACE4_DELETE_CHILD as well.

3. If the execute bit is set in the POSIX ACE, then set ACE4_EXECUTE.

4. Set ACE4_READ_ACL, ACE4_READ_ATTRIBUTES, and ACE4_SYNCHRONIZE unconditionally.

5. If the ACE is for the special "OWNER@" entity, set ACE4_WRITE_ACL and ACE4_WRITE_ATTRIBUTES.

6. Clear all other bits in the NFSv4 bitmask.

In addition, set the GROUP flag in each ACE which corresponds to a named group (but not in the GROUP@ ACE, or any of the other special entity ACEs).

At this point, we've replaced the POSIX ACL by an NFSv4 ACL with the same number of ACEs (ignoring any mask ACE), all of them ALLOW ACEs.

Order this NFSv4 ACL in the canonical order: OWNER@, users, GROUP@, groups, then EVERYONE@.

If the bitmasks in the resulting ACEs are non-increasing (so no ACE allows a bit not allowed by a previous ACE), then we can skip the next step.

Otherwise, we need to insert additional DENY ACE's to emulate the first-match semantics of the POSIX ACL permission-checking algorithm:

1. If an ACL_USER_OBJ, ACL_OTHER, or ACL_USER ACE fails to grant some permissions that are granted later in the ACL, then that ACE must be preceded by a single DENY ACE. The DENY ACE should have the same entity and flags as the corresponding ALLOW ACE, but the bitmask on the DENY ACE should be the bitwise NOT of the bitmask on the ALLOW ACE, except that the ACE4_WRITE_OWNER, ACE4_DELETE, ACE4_READ_NAMED_ATTRIBUTES, and ACE4_WRITE_NAMED_ATTRIBUTES bits should be cleared, and the ACE4_DELETE_CHILD bit should be cleared on non-directories. (Also, in the xdr-encoded ACL that

is transmitted, all bits not defined in the protocol should be cleared.)

2. All of the ACL_GROUP_OBJ and ACL_GROUP ACEs are consulted by the POSIX algorithm before determining permissions. To emulate this behaviour, instead of adding a single DENY before corresponding GROUP@ or named group ACEs, we insert a list of DENY ACEs after the list of GROUP@ and named group ACEs. Each DENY ACE is determined from its corresponding ALLOW ACE exactly as in the previous step. As before, these DENY ACEs should only be added when they are necessitated by an ACE that is less permissive than the final EVERYONE@ ACE.

On directories with default ACLs, we translate the default ACL as above, but set the ACE4_INHERIT_ONLY_ACE, ACE4_DIRECTORY_INHERIT_ACE, and ACE4_FILE_INHERIT_ACE flags on every ACE in the resulting ACL. On directories with both default and access ACLs, we translate the two ACLs and then concatenate them. The order of the concatenation is unimportant.

7. Mapping NFSv4 ACLs to POSIX ACLs

7.1. Requirements

Any mapping of NFSv4 ACLs to POSIX ACLs must map any NFSv4 ACL that is semantically equivalent to a POSIX ACL (with the exception of the "minor inaccuracy" mentioned above) to an equivalent POSIX ACL.

However, a more difficult problem is presented by NFSv4 ACLs that are not precisely equivalant to any POSIX ACL.

The only way that the NFSv4 protocol gives servers to indicate that they support only a subset of the ACL model is the "aclsupport" attribute, which allows a server to advertise that it only supports certain ACE types. This allows a server to report that it only supports ALLOW ACEs, or that it does not support AUDIT or ALARM ACEs (which will be the case for most servers with only POSIX ACLs). But it does not give a way to claim support for more complex subsets of the ACL model.

While it is possible for a server to reject any ACLs that do not fit its ACL model, this places a large burden on clients and users, since the server has no way to explain why it rejected a particular ACL. Therefore, it is preferable to be more forgiving, whenever that is possible without compromising security, and to limit any restrictions to those that are easily documented and verified by users.

The language of [1] allows a server some flexibility in handling ACLs that it cannot enforce completely accurately, as long as it adheres to "the guiding principle... that the server must not accept ACLs that appear to make [a file] more secure than it really is."

ACLs with arbitrary sequences of ALLOWs and DENYs may be particularly troublesome; but note that an NFSv4 ACL consisting entirely of ALLOW ACLs can always be transformed into a POSIX-equivalent ACL by first sorting it into the canonical order, then inserting DENY ACEs as necessary to ensure POSIX first-match semantics. Since inserting DENY ACEs can only restrict access, it is safe for a server to do this.

Therefore servers should accept, at least, any NFSv4 ACL that consists entirely of ALLOW ACLs.

Clients should also be at least as forgiving, to promote interoperability when heterogeneous clients share files.

7.2. Example NFSv4->POSIX Mapping

We now give an example of an algorithm that meets the above requirements. We assume it is to be used by a server mapping client-provided NFSv4 ACLs to POSIX ACLs it can store in its filesystem, so the translation errs on the side of making the ACL more restrictive.

In fact, if we ignore some loss of information in the mask ACE, this mapping takes an NFSv4 ACL to the unique most permissive POSIX ACL that is no more permissive than the given NFSv4 ACL.

Before starting, if the ACL in question is for a directory, we split it into two ACLs, one purely effective and one purely inherited, as follows:

1. ACEs with no inheritance flags are put in the purely effective ACL.

2. Aces with FILE_INHERIT and DIRECTORY_INHERIT both set are put in both the effective and the inherited ACL

3. Aces with FILE_INHERIT, DIRECTORY_INHERIT, and INHERIT_ONLY all set are put only in the inherited ACL.

Other combinations of ineritance flags may be rejected or silently modified to one of the above.

The main algorithm that follows is then performed on each ACL, with one used to set the effictive ACL, and one the default ACL.

First, we calculate the OTHER mode as follows:

1. Initialize the bitmasks other_allow and other_deny both to zero.

2. For each ACE in the ACL, starting from the top:

    1. If the ACE is not an EVERYONE@ ACE, ignore it and move to the next ACE.

    2. If the ACE is an EVERYONE@ ALLOW ACE, then add to other_allow any bits set in this ACE but not set in other_deny.

    3. If the ACE is an EVERYONE@ DENY ACE, then add to other_deny any bits set in this ACE but not set in other_allow.

3. Discard other_deny. Set the USER_OBJ mask from other_allow using the inverse of the mapping described previously in the POSIX-to-NFSv4 mapping, erring on the side of denying bits if it cannot

determine a sensible mapping. However, if certain bits simply cannot be mapped in a reasonable way to mode bits, the server may simply ignore them rather than returning an error. (For example, the server should deny write if either ACE4_WRITE_DATA or ACE4_APPEND_DATA are denied. But it may choose to ignore ACE4_READ_ATTRIBUTES entirely; though in that case it may at least want to treat specially the case where such bits are explicitly denied by some DENY ACE.)

Note that the bits determined above are exactly the maximum bits that will always be permitted to a user that doesn't match the file owner or group, or any of the named owners or groups. Thus this choice of the OTHER mode is exactly the maximum choice we can safely make.

Next we calculate the GROUP_OBJ and GROUP masks.

1. Initialize to zero an allow and deny bitmask for each GROUP_OBJ and for each GROUP mask.

2. For each ACE in the ACL, starting from the top:

    1. If the ACE is an OWNER@ or named user ACE, ignore it and move to the next ACE.

    2. If the ACE is an EVERYONE@ ALLOW ACE, then, for each GROUP or GROUP_OBJ allow mask, set the bits allowed in the EVERYONE ACE but not already in this GROUP or GROUP_OBJ's deny mask.

    3. If the ACE is an EVERYONE@ DENY ACE, then, for each GROUP or GROUP_OBJ deny mask, set the bits denied in the EVERYONE ACE but not already allowed in this GROUP or GROUP_OBJ's deny mask.

    4. If the ACE is a GROUP or GROUP@ ALLOW ACE, then set the allow bits in the corresponding GROUP or GROUP_OBJ allow mask that are allowed by this ACE but not already denied by the corresponding GROUP or GROUP_OBJ deny mask.

    5. If the ACE is a GROUP or GROUP@ DENY ACE, then set the deny bits in the corresponding GROUP or GROUP_OBJ deny mask that are denied by this ACE but not already allowed by the corresponding GROUP or GROUP_OBJ allow mask. Call the resulting deny mask "m". In each GROUP or GROUP_OBJ deny mask, set every bit that is in m and not already in that GROUP or GROUP_OBJ allow mask.

3. Having calculated allow and deny masks for GROUP_OBJ and each GROUP, we now set the corresponding modes from the allow masks as we did in the last step of the USER_OBJ mask calculation above.

Note that the bits thus determined for a group are exactly the maximum bits that will always be permitted to a user that matches the group in question, and that is denied any bits that could be denied by matching other groups, without out being allowed bits by matching any such groups. This is the most permissive mode we can choose that will never permit more permissions than the original NFSv4 ACL, for any possible choice of group memberships.

An implementation with special knowledge about the current gowning group or about group memberships may choose to use that knowledge to calculate a more permissive mode. However, doing so may render resulting POSIX ACL inaccurate after the owning group changes, or after any group memberships change.

Next, we calculate USER modes by first calculating allow and deny masks for each USER as above, this time assuming we are a user that does not match the file owner, that matches no user except for the one user under consideration, and that matches groups only when they would deny some permissions that they have not allowed yet. (To ensure this last step it will also be necessary to maintain group allow and deny ACEs, as we did in the previous calculation.) We omit the detailed steps, which are similar. Again, the implementation may choose to use special knowledge about group memberships at the risk of increased complexity and of loss of some accuracy.

Next, we calculate the USER_OBJ mode by calculating allow and deny masks for a user that matches the file owner and any user or group that denies bits that it does not first allow.

Finally, if the resulting ACL has any named user or group ACEs, add a mask ACE with bitmask equal to the union of the calculated permissions of all named users, group, and the GROUP_OBJ ACE.

The resulting mapping errs on the side of creating a more restrictive ACE. However it can be modified to produce a mapping that errs on the side of permissiveness, for the purposes of translating a server-provided NFSv4 ACL to a POSIX ACL to present to a user or application, as follows:

1. When performing the final mapping from the allow bitmask to a mode, we instead using a mapping that errs on the side of permissiveness; for example, we allow write permissions even if only one of WRITE_DATA, APPEND_DATA, or (in the case of directories) DELETE_CHILD is allowed.

2. Wherever in the above we pessimistically assume that a user will match any entity that has permissions denied to it before they are first allowed, we instead assume that the user will match any entity that has permissions allowed to it before they are first denied.

Once again, the resulting mapping may be seen to produce the unique (up to choice of mask) POSIX ACL which is the most restrictive among all POSIX ACLs no more restrictive than the given NFSv4 ACL.

Note that the above algorithms may be optimized in a number of ways: for example, although they are described in terms of multiple passes, it will be simpler and more efficient to calculate the entire POSIX ACL in a single pass.

8. Backwards Compatibility

Previous versions of this document recommended a different POSIX->NFSv4 mapping, which enforces POSIX semantics by inserting DENYs into the ACL even when those DENY's would have no effect, and which represents the POSIX mask ACE using additional DENYs. The resulting ACLs are overly complex and create problems for Windows clients, because the default Windows ACL editor prefers to order DENYs before ALLOWs.

The NFSv4 to POSIX mapping we describe in this document can accept the NFSv4 ACLs produced by the old mapping.

However, previous versions of this document also recommended accepting only NFSv4 ACLs that were precisely those produced by the old POSIX->NFSv4 mapping; therefore, existing implementations of that recommendation will reject the NFSv4 ACLs produced by the newer mapping.

We strongly recommend fixing implementations to accept a wider range of NFSv4 ACLs. However, we briefly document the old mapping here in case that is impossible:

Names, bitmasks, and flags are determined as in the the current mapping.

Whenever the following instructions requiring taking "the complement" of an NFSv4 bitmask, do so as follows: first, take the bitwise NOT of the bitmask. Then clear the ACE4_WRITE_OWNER, ACE4_DELETE, ACE4_READ_NAMED_ATTRIBUTES, and ACE4_WRITE_NAMED_ATTRIBUTES bits. Also, clear the ACE4_DELETE_CHILD bit on non-directories, and clear any bits not defined in the protocol.

Create one ALLOW ACE for each entity (OWNER@, GROUP@, and EVERYONE@, and each user and group named in the given POSIX ACL). After each OWNER@, EVERYONE@, and named user ACE, append a DENY ACE with the same entity and flags as the corresponding ALLOW ACE, but with bitmask set to the complement (as defined above) of the ALLOW ACE.

Do the same for each GROUP@ and named group ACE, but instead of inserting each new DENY ACE after the corresponding ALLOW ACE, insert all of the DENY ACEs at the end of the list of GROUP@ and named group ACEs, in the same order that the GROUP@ and named group ALLOW ACEs occur in.

Finally, prepend each GROUP@, named user, and named group ACE by a single DENY whose entity and flags are the same as the corresponding ALLOW, but whose bitmask is the complement (as defined above) of the

bitmask determined from the mask ACE in the given POSIX ACL. Skip this step if the given POSIX ACL has no mask ACE.

9. Security Considerations

Any automatic mapping from one ACL model to another must provide guarantees as to how the mapping affects the meaning of ACLs, or risk misleading users about the permissions set on filesystem objects. For this reason, caution is recommended when implementing this mapping. It is better to return errors than to break any such guarantees.

That said, there may be cases where small losses in accuracy can avoid dramatic interoperability and usability problems; as long as the losses in accuracy are clearly documented, these tradeoffs may be found acceptable.

For example, a server unable to support all of the NFSv4 mode bits does not have a way to communicate its exact limitations to clients, so clients (and users) may be unable to recover from such errors. For this reason we recommend ignoring bitmask bits that the server is completely unable to map to mode bits, at least when no ACE explicitly contradicts the server's default behavior. If this is considered insufficient, we should add to the NFSv4 protocol additional attributes necessary to advertise the server's limitations.

Note also that any ACL mapping also requires mapping between NFSv4 usernames and local id's. When the mapping of id's depends on remote services, the method used for the mapping must be at least as secure as the method used to set or get ACLs.

10. References

[1] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and D. Noveck, "Network File System (NFS) version 4 Protocol", RFC 3530, April 2003.

[2] Institute of Electrical and Electronics Engineers, Inc., "IEEE Draft P1003.1e", October 1997, <http://wt.xpilot.org/publications/posix.1e/download.html>.

Authors' Addresses

Marius Aamodt Eriksen
U. of Michigan Center for Information Technology Integration

Email: marius@citi.umich.edu

J. Bruce Fields
U. of Michigan Center for Information Technology Integration

Email: bfields@citi.umich.edu

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.