Mapping Between NFSv4 and Posix Draft ACLs
draft-ietf-nfsv4-acl-mapping-04

Status of this Memo

Copyright Notice

Abstract

A number of filesystems and applications support ACLs based on a withdrawn POSIX draft [2]. Those ACLs differ significantly from NFS version 4 (NFSv4) ACLs [1]. We describe how to translate between the two types of ACLs.

1. Introduction

Access Control Lists (ACLs) are used to specify fine-grained 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In the following sections we describe two ACL models: NFSv4 ACLs, and ACLs based on a withdrawn POSIX draft.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@".

The type may be 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. Assume a a requester asks for access, as 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, behaviour is undefined. If an access mask consisting of more than one bit is requested, it succeeds if and only if each bit in the mask is allowed.

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".

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 as follows:

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'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,

3. 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,

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

The above description omits one detail: in steps (2) and (3), the requested bits must be granted both by the matching ACE and by the ACL_MASK ACE. The ACL_MASK ACE thus limits the maximum permissions which may be granted by any ACL_USER or ACL_GROUP ACE, or by the ACL_GROUP_OBJ ACE.

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. We permit a server or client to use a different mapping, provided the mapping meets 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.)

For simplicity's sake, the translator may choose to handle the mask by first applying it to the USER, GROUP, and GROUP_OBJ ACEs, and then mapping the resulting ACL. However, that will result in an ACL from which it is impossible to determine the original value of the mask or of the masked USER, GROUP, and GROUP_OBJ bitmasks. If the resulting ACL is later translated back to a POSIX ACL, the translator will assume that the value of the mask is the union of the bitmasks permitted to any USER, GROUP, or GROUP_OBJ. If that would be incorrect, the original translation should not modify the bitmasks of the USER, GROUP, and GROUP_OBJ bitmasks, and should instead use additional DENY ACEs as necessary to give the effect of the mask. It should also arrange for the first GROUP@ ACE to be a DENY ACE whose bitmask is determined by the mask, allowing that ACE to be used to determine the original mask value.

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, 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, we 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 prepended 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, 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.

Finally, we enforce the POSIX mask ACE by prepending each ALLOW ACE for a named user, GROUP@, or named group, with a single DENY ACE whose entity and flags are the same as those for the corresponding ALLOW ACE, but whose bitmask is the inverse of the bitmask determined from the mask ACE, with the inverse calculated as described above. In the case of named users, these DENY aces may be coalesced with any existing prepended DENY aces. The DENY aces are omitted entirely if they would have no affect, or if the mask ACE has the same bitmask as the maximum of the affected ACEs. (With the one exception that if the POSIX ACL posesses exactly 4 ACEs, then a mask-derived DENY ace should be inserted before the GROUP@ ace, even if it would not otherwise be.)

Regardless of what scheme is used to represent the mask, the receiver will use the first GROUP@ DENY ace to determine the value of the mask (if it is different from the union of the bitmasks on the affected ACEs), and use the relevant ALLOWs to determine the pre-mask values of user and group ACEs.

The implementation may also choose to just mask out the bitmasks on the relevant ALLOW ACEs. This will produce a simpler ACL (in particular, an ACL that usually requires no DENY ACE's), at the expense of losing some ACL information after a chmod.

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 the equivalent POSIX ACL. It should also extract the mask correctly; as the mask doesn't affect the semantics of the NFSv4 ACL, and as there is more than one way the mask might be encoded, we require a convention for this. Specifically: we require that the mask be computed as the bitmask used on the first GROUP@ DENY ACE which precedes any GROUP@ allow ACE, unless no such DENY ACE exists, in which case the mask must be computed as the union of the bitmasks allowed to all named users, groups, and GROUP@ (where by the "bitmask allowed to" an entity we mean the maximum bitmask that the ACL would permit to any user matching the entity).

Implementations may vary in how they deal with NFSv4 ACLs that are not precisely semantically equivalent to any POSIX ACL. In particular they may return errors for such ACLs instead of attempting to map them. However, when possible without compromising security, they should attempt to be forgiving.

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."

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, and 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.

We require any server to 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 less permissive.

Given an NFSv4 ACL, first calculate the mask by taking the bitmask from the first GROUP@ DENY ACE from the original NFSv4 ACL, if it exists. After doing so, remove that DENY ACE, and clear the bits in its bitmask from any DENY ACE for a named user, group, or GROUP@ which precedes an ALLOW ACE for the same entity.

In the case where there is no such GROUP@ DENY ACE, continue through the rest of the algorithm and then calculate the mask as the union of the calculated permissions of all named users, group, and the GROUP_OBJ ACE.

Given an NFSv4 ACL, sort it into canonical order (OWNER@ ACEs first, then user ACEs, then GROUP@ ACEs, then group ACEs, then EVERYONE@ ACEs.) Also, sort the GROUP@ and group ACEs that all ALLOW ACEs precede all DENY ACEs. To do so, take advantage of the following observations:

1. If two consecutive ACEs are either both ALLOW ACEs, or both DENY ACEs, then we can swap their order without changing the effect of the ACL.

2. If it would be impossible for a single user to match both of the entities on two consecutive ACEs, then we can swap their order without changing the effect of the ACL.

3. If an ALLOW ACE is immediately followed by a DENY ACE, then swapping the order of the two ACEs will not make the ACL any more permissive.

4. If a DENY ACE is immediately followed by an ALLOW ACE, then swapping the order of the two ACEs will not make the ACL any more permissive, *if* we modify the bitmask on the ALLOW ACE by clearing any bits that are set in the DENY ACE.

The second observation is the trickiest: it may usually be safe to assume that two distinct user names cannot match the same user. An implementation with knowledge about group memberships or about the current value of the file owner might also use that information, but if it does so it will produce a translation that is no longer accurate after owners or group memberships change.

Fortunately, observations 1, 3, and 4 are sufficient to sort any ACL into canonical order, so a paranoid implementation can simply ignore number 2 completely, while an implementation willing to sacrifice some accuracy may choose to do something more complex.

Ensure that the resulting ACL posesses at least one each of OWNER@, GROUP@, and EVERYONE@ ACEs, by inserting an ALLOW ACE with a zero

bitmask if necessary in the correct position.

Next, for each entity, calculate a bitmask for that entity as follows: Starting with the first ACE for that entity (ignoring all previous ACEs), perform the NFSv4 ACL-checking algorithm for a user that is assumed to match the entity on every DENY ACE that a user matching the given entity might match, but is assumed to match only those entities on ALLOW ACEs that *any* user matching the current entity must match.

Finally, construct the POSIX ACL by translating NFSv4 entity names to uid's and gid's (and handling special entities in the obvious way), then assign a POSIX bitmask determined by the NFSv4 bitmask calculated in the previous step; the bitmask calculation should use 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.)

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 sorting ACEs, ALLOW ACEs can always be moved towards the start of the ACL, but a DENY ACE can be moved towards the start of the ACL only as long as we clear any of the DENY ACE's bitmask bits that are set in the intervening ALLOW ACEs.

2. When calculating the NFSv4 bitmask for each entity, err on the side of assuming that ALLOW ACEs apply and that DENY ACEs don't, with the one exception that when calculating the GROUP@ and named group bitmasks, ALLOW ACEs for groups other than the one under consideration should be ignored.

3. When mapping the NFSv4 bitmask to POSIX mode bits, err on the side of allowing access.

8. 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, and advertising this fact loudly in the server documentation. If this is considered insufficient, we should add to the NFSv4 protocol additional attributes necessary to advertise the server's limitations.

Note also that this ACL mapping 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.

9. 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: marius@citi.umich.edu

Intellectual Property Statement

Disclaimer of Validity

Copyright Statement

Acknowledgment

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