Network Working Group S. Falkner
Internet-Draft L. Week
Expires: August 19, 2006 Sun Microsystems, Inc.
February 15, 2006

NFS Version 4 ACLs
draft-ietf-nfsv4-acls-00.txt

Status of this Memo

Copyright Notice

Abstract

NFS version 4 (specified in RFC 3530) Access Control Lists (ACLs) provide more fine grained control than previous file permission models, but before the full benefit of the model can be exploited, some changes and clarifications must be made. This document will describe the details that implementors should consider in order to allow implementations to function and interoperate better.

Table of Contents

1. Requirements notation 3
2. Security Considerations 4
3. Introduction 5
4. Syntax for the Representation of ACLs 6
5. Interaction Between Mode and ACL 7
5.1. What should happen to the mode if a SETATTR of ACL is done? 7
5.2. How should a mode given in the arguments to CREATE or OPEN affect an inherited ACL? 11
5.3. What should happen to an existing ACL if a mode is applied to the file/directory? 12
5.4. What should happen if both mode and ACL are given to SETATTR? 17
5.4.1. Client Side Recommendations 17
5.4.2. Server Side Recommendations 18
6. Deficiencies in a Mode Representation of an ACL 19
7. Access Control Semantics 20
8. Inheritance and turning it off 21
9. EVERYONE@: What does it mean? 22
10. Access Mask Bit Discussion 23
11. Append Only Behavior 27
12. ACE4_DELETE/ACE4_DELETE_CHILD Behavior 28
13. ACE4_ADD_FILE and ACE4_ADD_SUBDIRECTORY 29
14. POSIX Considerations 30
14.1. Background Information 30
14.2. Additional and Alternate File Access Control Mechanisms . 30
14.3. NFSv4 ACLs vs. POSIX-draft ACLs 30
14.4. NFSv4 ACLs vs. POSIX 32
14.5. umask Considerations 33
15. Normative References 33
Authors' Addresses 34
Intellectual Property and Copyright Statements 35

1. Requirements notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Security Considerations

None.

3. Introduction

Readers of this document are expected to have basic knowledge of the Network File System (NFS) version 4 protocol [RFC3530]. Familiarity with the NFS version 4 protocol will provide the correct context for the issues discussed in this document.

The NFS version 4 protocol defines a standard Access Control List (ACL) model, which to our knowledge, is the first approved standard for ACLs. Prior attempts have been made to standardize ACL models, but none have succeeded. Therefore, there has been a proliferation of ACL models in the computer industry. These multiple models make it close to impossible to interoperate between the wide array of vendors.

NFS version 4 ACLs attempt to bridge the gap between the different vendors, allowing them to interoperate. Implementations have been suffering from differing interpretations of the standard. This document attempts to clarify some of the pieces of the NFS version 4 ACL model with the hope that vendors will be able to agree on semantics which will lead to increased ability to interoperate.

4. Syntax for the Representation of ACLs

Throughout the document the following abbreviations will be used for the ACE type:

ALLOW (short for ACE4_ACCESS_ALLOWED_ACE_TYPE)

DENY (short for ACE4_ACCESS_DENIED_ACE_TYPE)

AUDIT (short for ACE4_SYSTEM_AUDIT_ACE_TYPE)

ALARM (short for ACE4_SYSTEM_ALARM_ACE_TYPE)

Representation of ACLs in this document will be of the form:

```
<Field #1>:<Field #2>:<Field #3>:<Field #4>
```

Field #1 is the "ACE who" and example values are:
OWNER@, GROUP@, lisagab@sun.com, samf@sun.com

Field #2 are the "access mask bits" separated with "/" characters and example values are:
ACE4_READ_DATA/ACE4_WRITE_DATA

Field #3 are the "ACE flags" and example values are:
ACE4_FILE_INHERIT_ACE, ACE4_IDENTIFIER_GROUP

Field #4 is the "ACE type" and example values are:
ALLOW, DENY

5. Interaction Between Mode and ACL

For client and server implementors alike, a misunderstanding of the interactions between the mode and ACL file attributes is likely to be a cause of problems.

The relationship between NFS version 4 modes and ACLs is difficult, but not impossible to specify. Contributing to this is the fact that while there are portions of the mode that ACLs don't specify, it is impossible for a mode to represent all of the information in an ACL.

The NFS version 4 mode attribute is based on the UNIX mode bits. Some information that is traditionally associated with the UNIX mode bits, "setuid"/"setgid"/"sticky" (MODE4_SUID/MODE4_SGID/MODE4_SVTX), are not defined to be part of the ACL. Other information that can also affect file permissions, such as the file's owner and owning group are also not defined to be part of the ACL. In other words, from looking at an ACL, a user will be unable to tell who the owner and owning group of a file is. This is because of the special identifiers, OWNER@ and GROUP@.

This section will attempt to answer the following questions:

1. What should happen to the mode if a SETATTR of ACL is done?

2. How should a mode given to CREATE or OPEN affect an inherited ACL?

3. What should happen to an existing ACL if a mode is applied to the file/directory?

4. What should happen if both mode and ACL are given to SETATTR?

5.1. What should happen to the mode if a SETATTR of ACL is done?

Keeping the mode and ACL attributes synchronized is important, but as mentioned previously, the mode cannot possibly represent all of the information in the ACL.

Still, the mode should be modified to represent the access as accurately as possible. The mode is not guaranteed to be accurate and could potentially be more restrictive than the access that would actually be given by the ACL (for more discussion on this topic, see Section 6). Because of this, client implementations are not recommended to not do their own access checks based on the mode of a file. For further information on access checking, see Section 7.

The general algorithm to assign a new mode attribute to an object

based on a new ACL being set is:

1. Walk through the ACEs in order, looking for ACEs with a "who" value of OWNER@, GROUP@, or EVERYONE@.

2. It is understood that ACEs with a "who" value of OWNER@ affect the *USR bits of the mode, GROUP@ affect *GRP bits, and EVERYONE@ affect *USR, *GRP, and *OTH bits (For more discussion on the EVERYONE@ special identifier see Section 9).

3. If such an ACE specifies ALLOW or DENY for ACE4_READ_DATA, ACE4_WRITE_DATA, or ACE4_EXECUTE, and the mode bits affected have not been determined yet, set them to one (if ALLOW) or zero (if DENY).

4. Upon completion, any mode bits as yet undetermined have a value of zero.

This pseudocode more precisely describes the algorithm:

```
/* octal constants for the mode bits */

RUSR = 0400
WUSR = 0200
XUSR = 0100
RGRP = 0040
WGRP = 0020
XGRP = 0010
ROTH = 0004
WOTH = 0002
XOTH = 0001

/*
 * old_mode represents the previous value
 * of the mode of the object.
 */

mode_t mode = 0, seen = 0;
for each ACE a {
    if a.type is ALLOW or DENY and
    ACE4_INHERIT_ONLY_ACE is not set in a.flags {
        if a.who is OWNER@ {
            if ((a.mask & ACE4_READ_DATA) &&
                (! (seen & RUSR))) {
                    seen |= RUSR;
                    if a.type is ALLOW {
                        mode |= RUSR;
                    }
```

```
                   }
                   if ((a.mask & ACE4_WRITE_DATA) &&
                       (! (seen & WUSR))) {
                           seen |= WUSR;
                           if a.type is ALLOW {
                               mode |= WUSR;
                           }
                   }
                   if ((a.mask & ACE4_EXECUTE) &&
                       (! (seen & XUSR))) {
                           seen |= XUSR;
                           if a.type is ALLOW {
                               mode |= XUSR;
                           }
                   }
               } else if a.who is GROUP@ {
                   if ((a.mask & ACE4_READ_DATA) &&
                       (! (seen & RGRP))) {
                           seen |= RGRP;
                           if a.type is ALLOW {
                               mode |= RGRP;
                           }
                   }
                   if ((a.mask & ACE4_WRITE_DATA) &&
                       (! (seen & WGRP))) {
                           seen |= WGRP;
                           if a.type is ALLOW {
                               mode |= WGRP;
                           }
                   }
                   if ((a.mask & ACE4_EXECUTE) &&
                       (! (seen & XGRP))) {
                           seen |= XGRP;
                           if a.type is ALLOW {
                               mode |= XGRP;
                           }
                   }
               } else if a.who is EVERYONE@ {
                   if (a.mask & ACE4_READ_DATA) {
                       if ! (seen & RUSR) {
                           seen |= RUSR;
                           if a.type is ALLOW {
                               mode |= RUSR;
                           }
                       }
                       if ! (seen & RGRP) {
                           seen |= RGRP;
                           if a.type is ALLOW {
```

```
                    mode |= RGRP;
                }
            }
            if ! (seen & ROTH) {
                seen |= ROTH;
                if a.type is ALLOW {
                    mode |= ROTH;
                }
            }
        }
        if (a.mask & ACE4_WRITE_DATA) {
            if ! (seen & WUSR) {
                seen |= WUSR;
                if a.type is ALLOW {
                    mode |= WUSR;
                }
            }
            if ! (seen & WGRP) {
                seen |= WGRP;
                if a.type is ALLOW {
                    mode |= WGRP;
                }
            }
            if ! (seen & WOTH) {
                seen |= WOTH;
                if a.type is ALLOW {
                    mode |= WOTH;
                }
            }
        }
        if (a.mask & ACE4_EXECUTE) {
            if ! (seen & XUSR) {
                seen |= XUSR;
                if a.type is ALLOW {
                    mode |= XUSR;
                }
            }
            if ! (seen & XGRP) {
                seen |= XGRP;
                if a.type is ALLOW {
                    mode |= XGRP;
                }
            }
            if ! (seen & XOTH) {
                seen |= XOTH;
                if a.type is ALLOW {
                    mode |= XOTH;
                }
```

```
                        }
                    }
                }
            }
        }
        return mode | (old_mode & (SUID | SGID | SVTX))
```

5.2. How should a mode given in the arguments to CREATE or OPEN affect an inherited ACL?

The goal of implementing ACL inheritance is for newly created objects to inherit the ACLs they were intended to inherit, but without disregarding the mode that is given with the arguments to the CREATE or OPEN operations. The general algorithm is as follows:

1. Form an ACL on the newly created object that is the concatenation of all inheritable ACEs from its parent directory. Note that there may be zero inheritable ACEs; thus, an object may start with an empty ACL.

   This is self explanatory. If, for example, a new non-directory file is being created, ACEs with the flag of ACE4_FILE_INHERIT_ACE will be considered inheritable.

2. For each ACE in the new ACL, adjust its flags if necessary, and possibly create two ACEs in place of one.

   This will be discussed in detail below.

3. Apply the algorithm for applying a mode to a file/directory with an existing ACL on the new object as described in Section 5.3, using the mode that is to be used for file creation.

   This ensures that the mode is honored.

Step 2 above is necessary to honor the intent of the inheritance-related flags. It also is intended to preserve information about the original inheritable ACEs in the case that they will be modified by other steps. Paragraph 2 is detailed in the following algorithm:

1. If the ACE4_NO_PROPAGATE_INHERIT_ACE is set, or the type of the file is something other than "directory", then clear the following flags:

      ACE4_NO_PROPAGATE_INHERIT_ACE

      ACE4_FILE_INHERIT_ACE

ACE4_DIRECTORY_INHERIT_ACE

ACE4_INHERIT_ONLY_ACE

Continue on to the next ACE.

2. If the type of file is "directory" and ACE4_FILE_INHERIT_ACE is set and ACE4_DIRECTORY_INHERIT_ACE is NOT set, then we ensure that ACE4_INHERIT_ONLY_ACE is set. Continue on to the next ACE. Otherwise:

3. If the type of the ACE is neither ALLOW nor DENY, then continue on to the next ACE.

4. Copy the original ACE into a second, adjacent ACE.

5. On the first ACE, ensure that ACE4_INHERIT_ONLY_ACE is set.

6. On the second ACE, clear the following flags:

   ACE4_NO_PROPAGATE_INHERIT_ACE

   ACE4_FILE_INHERIT_ACE

   ACE4_DIRECTORY_INHERIT_ACE

   ACE4_INHERIT_ONLY_ACE

7. On the second ACE, if the type field is ALLOW, an implementation MAY clear the following mask bits:

   ACE4_WRITE_ACL

   ACE4_WRITE_OWNER

5.3. What should happen to an existing ACL if a mode is applied to the file/directory?

An existing ACL can mean two things in this context. One, that a file/directory already exists and it has an ACL. Two, that a directory has inheritable ACEs that will make up the ACL for any new files or directories created therein.

The high-level goal of the behavior when a mode is set on a file with an existing ACL is to take the new mode into account, without needing to disregard a pre-existing ACL.

When a mode is applied to an object, e.g. via SETATTR or CREATE/OPEN,

the ACL must be modified to accommodate the mode.

1. The ACL is traversed, one ACE at a time. For each ACE:

   1. If the type of the ACE is neither ALLOW nor DENY, the ACE is left unchanged. Continue to the next ACE.

   2. If the ACE4_INHERIT_ONLY_ACE flag is set on the ACE, it is left unchanged. Continue to the next ACE.

   3. If either or both of ACE4_FILE_INHERIT_ACE or ACE4_DIRECTORY_INHERIT_ACE are set:

      1. A copy of the ACE is made, and placed in the ACL immediately following the current ACE.

      2. In the first ACE, the flag ACE4_INHERIT_ONLY_ACE is set.

      3. In the second ACE, the following flags are cleared:

         ACE4_FILE_INHERIT_ACE

         ACE4_DIRECTORY_INHERIT_ACE

         ACE4_NO_PROPAGATE_INHERIT_ACE

      The algorithm continues on with the second ACE.

   4. If the "who" field is one of the following:

      OWNER@

      GROUP@

      EVERYONE@

      then the following mask bits are cleared:

      ACE4_READ_DATA

      ACE4_LIST_DIRECTORY

      ACE4_WRITE_DATA

      ACE4_ADD_FILE

      ACE4_APPEND_DATA

ACE4_ADD_SUBDIRECTORY

ACE4_EXECUTE

At this point, we proceed to the next ACE.

5. Otherwise, if the "who" field did not match one of OWNER@, GROUP@, or EVERYONE@, the following steps SHOULD be performed.

   1. If the type of the ACE is ALLOW, we check the preceding ACE (if any). If it does not meet all of the following criteria:

      1. The type field is DENY.

      2. The who field is the same as the current ACE.

      3. The flag bit ACE4_IDENTIFIER_GROUP is the same as it is in the current ACE, and no other flag bits are set.

      4. The mask bits are a subset of the mask bits of the current ACE, and are also a subset of the following:

         ACE4_READ_DATA

         ACE4_LIST_DIRECTORY

         ACE4_WRITE_DATA

         ACE4_ADD_FILE

         ACE4_APPEND_DATA

         ACE4_ADD_SUBDIRECTORY

         ACE4_EXECUTE

      then an ACE of type DENY, with a who equal to the current ACE, flag bits equal to (<current-ACE-flags> & ACE4_IDENTIFIER_GROUP), and no mask bits, is prepended.

   2. The following modifications are made to the prepended ACE. The intent is to mask the following ACE to disallow ACE4_READ_DATA, ACE4_WRITE_DATA, ACE4_APPEND_DATA, or ACE4_EXECUTE, based upon the group permissions of the new mode. As a special case, if the ACE matches the current

owner of the file, the owner bits are used, rather than the group bits. This is reflected in the algorithm below.

Let there be three bits defined:

```
#define READ    04
#define WRITE   02
#define EXEC    01
```

Let "amode" be the new mode, right-shifted three bits, in order to have the group permission bits placed in the three low order bits of amode, i.e. amode = mode >> 3

If ACE4_IDENTIFIER_GROUP is not set in the flags, and the "who" field of the ACE matches the owner of the file, we shift amode three more bits, in order to have the owner permission bits placed in the three low order bits of amode:

amode = amode >> 3

amode is now used as follows:

```
If ACE4_READ_DATA is set on the current ACE:
   If READ is set on amode:
       ACE4_READ_DATA is cleared on the prepended ACE
   else:
       ACE4_READ_DATA is set on the prepended ACE

If ACE4_WRITE_DATA is set on the current ACE:
   If WRITE is set on amode:
       ACE4_WRITE_DATA is cleared on the prepended ACE
   else:
       ACE4_WRITE_DATA is set on the prepended ACE

If ACE4_APPEND_DATA is set on the current ACE:
   If WRITE is set on amode:
       ACE4_APPEND_DATA is cleared on the prepended ACE
   else:
       ACE4_APPEND_DATA is set on the prepended ACE

If ACE4_EXECUTE is set on the current ACE:
   If EXEC is set on amode:
       ACE4_EXECUTE is cleared on the prepended ACE
   else:
```

ACE4_EXECUTE is set on the prepended ACE

3. To conform with POSIX, and prevent cases where the owner of the file is given permissions via an explicit group (i.e. alternate permissions are not disabled following a chmod), we implement the following step.

```
If ACE4_IDENTIFIER_GROUP is set in the flags field of
the ALLOW ACE:
    Let "mode" be the mode that we are chmoding to:
        extramode = (mode >> 3) & 07
        ownermode = mode >> 6
        extramode &= ~ownermode
    If extramode is not zero:
        If extramode & READ:
            Clear ACE4_READ_DATA in both the
            prepended DENY ACE and the ALLOW ACE
        If extramode & WRITE:
            Clear ACE4_WRITE_DATA and ACE_APPEND_DATA in both
            the prepended DENY ACE and the ALLOW ACE
        If extramode & EXEC:
            Clear ACE4_EXECUTE in both the prepended DENY
            ACE and the ALLOW ACE
```

2. If there are at least six ACEs, the final six ACEs are examined. If they are not equal to the following ACEs:

```
A1) OWNER@:::DENY
A2) OWNER@:ACE4_WRITE_ACL/ACE4_WRITE_OWNER/
    ACE4_WRITE_ATTRIBUTES/ACE4_WRITE_NAMED_ATTRIBUTES::ALLOW
A3) GROUP@::ACE4_IDENTIFIER_GROUP:DENY
A4) GROUP@::ACE4_IDENTIFIER_GROUP:ALLOW
A5) EVERYONE@:ACE4_WRITE_ACL/ACE4_WRITE_OWNER/
    ACE4_WRITE_ATTRIBUTES/ACE4_WRITE_NAMED_ATTRIBUTES::DENY
A6) EVERYONE@:ACE4_READ_ACL/ACE4_READ_ATTRIBUTES/
    ACE4_READ_NAMED_ATTRIBUTES/ACE4_SYNCHRONIZE::ALLOW
```

Then six ACEs matching the above are appended.

3. The final six ACEs are adjusted according to the incoming mode.

```
/* octal constants for the mode bits */

RUSR = 0400
WUSR = 0200
XUSR = 0100
RGRP = 0040
WGRP = 0020
XGRP = 0010
ROTH = 0004
WOTH = 0002
XOTH = 0001

If RUSR is set: set ACE4_READ_DATA in A2
    else: set ACE4_READ_DATA in A1
If WUSR is set: set ACE4_WRITE_DATA and ACE4_APPEND_DATA in A2
    else: set ACE4_WRITE_DATA and ACE4_APPEND_DATA in A1
If XUSR is set: set ACE4_EXECUTE in A2
    else: set ACE4_EXECUTE in A1
If RGRP is set: set ACE4_READ_DATA in A4
    else: set ACE4_READ_DATA in A3
If WGRP is set: set ACE4_WRITE_DATA and ACE4_APPEND_DATA in A4
    else: set ACE4_WRITE_DATA and ACE4_APPEND_DATA in A3
If XGRP is set: set ACE4_EXECUTE in A4
    else: set ACE4_EXECUTE in A3
If ROTH is set: set ACE4_READ_DATA in A6
    else: set ACE4_READ_DATA in A5
If WOTH is set: set ACE4_WRITE_DATA and ACE4_APPEND_DATA in A6
    else: set ACE4_WRITE_DATA and ACE4_APPEND_DATA in A5
If XOTH is set: set ACE4_EXECUTE in A6
    else: set ACE4_EXECUTE in A5
```

5.4. What should happen if both mode and ACL are given to SETATTR?

The only reason that a mode and ACL should be set in the same SETATTR is if the user wants to set the SUID, SGID and SVTX bits along with setting the permissions by means of an ACL. There is still no way to enforce which order the attributes will be set in, and it is likely that different orders of operations will produce different results.

In the long run, the best solution would be the ability to set SUID, SGID and SVTX bits independent of the mode, but since we don't have this ability in NFS version 4.0, this is what we recommend.

5.4.1. Client Side Recommendations

If an application needs to enforce a certain behavior, it is recommended that the client implementations set mode and ACL in separate SETATTR requests. This will produce consistent and expected

results.

If an application wants to set SUID, SGID and SVTX bits and an ACL, we recommend:

In the first SETATTR, set the mode with SUID, SGID and SVTX bits as desired and all other bits with a value of 0.

In a following SETATTR (preferably in the same COMPOUND) set the ACL.

5.4.2. Server Side Recommendations

If both mode and ACL are given to SETATTR, server implementations should verify that the mode and ACL don't conflict, i.e. the mode computed from the given ACL must be the same as the given mode, excluding the SUID, SGID and SVTX bits. The algorithm for assigning a new mode based on the ACL can be used. This is described in Section 5.1. If a server receives a request to set both mode and ACL, but the two conflict, the server should return NFS4ERR_INVAL.

6. Deficiencies in a Mode Representation of an ACL

As mentioned in Section 5.1, the representation of the mode is deterministic, but not guaranteed to be accurate. The mode bits potentially convey a more restrictive permission than what will actually be granted via the ACL.

Given the following ACL of two ACEs:

```
GROUP@:ACE4_READ_DATA/ACE4_WRITE_DATA/ACE4_EXECUTE:
    ACE4_IDENTIFIER_GROUP:ALLOW
EVERYONE@:ACE4_READ_DATA/ACE4_WRITE_DATA/ACE4_EXECUTE::DENY
```

we would compute a mode of 0070. However, it is possible, even likely, that the owner might be a member of the object's owning group, and thus, the owner would be granted read, write, and execute access to the object. This would conflict with the mode of 0070, where an owner would be denied this access.

The only way to overcome this deficiency would be to determine whether the object's owner is a member of the object's owning group. This is difficult, but worse, on a POSIX or any UNIX-like system, it is a process' membership in a group that is important, not a user's. Thus, any fixed mode intended to represent the above ACL can be incorrect.

Example: administrative databases (possibly /etc/passwd and /etc/group) indicate that the user "bob" is a member of the group "staff". An object has the ACL given above, is owned by "bob", and has an owning group of "staff". User "bob" has logged in to the system, and thus processes have been created owned by "bob" and having membership in group "staff".

A mode representation of the above ACL could thus be 0770, due to user "bob" having membership in group "staff". Now, the administrative databases are changed, such that user "bob" is no longer in group "staff". User "bob" logs in to the system again, and thus more processes are created, this time owned by "bob" but NOT in group "staff".

A mode of 0770 is inaccurate for processes not belonging to group "staff". But even if the mode of the file were proactively changed to 0070 at the time the group database was edited, mode 0070 would be inaccurate for the pre-existing processes owned by user "bob" and having membership in group "staff".

7. Access Control Semantics

The NFS version 4 specification [RFC3530] defines how an ACL is interpreted, and it states that the access is undefined if you get through the entire ACL and haven't encountered an ALLOW or DENY ACE for the requester.

We now recommend that if you fall through the ACL, access is denied. This allows the behavior to be clearly defined, and consistent across implementations. In fact, there is precedence for this behavior in current implementations.

It is convenient that [RFC3530] gave implementations flexibility by leaving the access undefined, but the flexibility is still present given that there have always been security policies independent of file permissions. Servers can have other security policies in place, and in those cases, access will be decided outside of what is defined in the ACL.

Examples of security policies that can be in place outside of what is defined (or not defined) in the ACL are:

1. The owner of the file will always be granted ACE4_WRITE_ACL and ACE4_READ_ACL permissions.

2. The ACL may say that an entity is to be granted ACE4_WRITE_DATA permission, but the file system is mounted read only.

For multiple reasons, including the one listed above, client implementations are recommended not to do their own access checking. All access checking should be done on the server.

8. Inheritance and turning it off

The inheritance of access permissions may be problematic if a user cannot prevent their file from inheriting unwanted permissions. For example, a user, "samf", sets up a shared project directory to be used by everyone working on Project Foo. "lisagab" is a part of Project Foo, but is working on something that should not be seen by anyone else. How can "lisagab" make sure that any new files that she creates in this shared project directory do not inherit anything that could compromise the security of her work?

More relevant to the implementors of NFS version 4 clients and servers is the question of how to communicate the fact that user, "lisagab", doesn't want any permissions to be inherited to her newly created file or directory.

To do this, implementors should standardize on what the behavior of CREATE and OPEN must be if:

1. just mode is given

   In this case, inheritance will take place, but the mode will be applied to the inherited ACL as described in Section 5.1, thereby modifying the ACL.

2. just ACL is given

   In this case, inheritance will not take place, and the ACL as defined in the CREATE or OPEN will be set without modification.

3. both mode and ACL are given

   In this case, implementors should verify that the mode and ACL don't conflict, i.e. the mode computed from the given ACL must be the same as the given mode. The algorithm for assigning a new mode based on the ACL can be used. This is described in Section 5.1) If a server receives a request to set both mode and ACL, but the two conflict, the server should return NFS4ERR_INVAL. If the mode and ACL don't conflict, inheritance will not take place and both, the mode and ACL, will be set without modification.

4. neither mode nor ACL are given

   In this case, inheritance will take place and no modifications to the ACL will happen. It is worth noting that if no inheritable ACEs exist on the parent directory, the file will be created with an empty ACL, thus granting no accesses.

9. EVERYONE@: What does it mean?

The NFS version 4 specification [RFC3530] refers to the "EVERYONE@" special identifier as meaning "The world". This is confusing because there are a couple of different ways to interpret the wording. These different interpretations are problematic and it would be advantageous for implementors to standardize on a single meaning.

The different interpretations are as follows:

1. "EVERYONE@" is equivalent to the UNIX "other" entity, which by definition does not include the owner or owning group of the file.

2. "EVERYONE@" literally means everyone, including the file's owner and owning group.

The goal of standardizing on what "EVERYONE@" means may be best expressed from a user's point of view. A user of NFS version 4 ACLs should expect that setting an ACL such as the following will have the same affect regardless of what vendor's implementation they are using.

EVERYONE@:ACE4_READ_DATA::ALLOW

Examples of how the different interpretations could cause different behaviors are as follows:

If we take interpretation #1 where "EVERYONE@" is equivalent to the UNIX "other" entity and the owner or a user in the owning group attempt to access the file for reading, they would be denied. This is because we fell through the ACL and didn't find any entries specifying the permissions for OWNER@ and GROUP@ (see Section 7).

If we take interpretation #2 where "EVERYONE@" is literally everyone, including the owner and owning group, and the owner or a user in the owning group attempt to access the file for reading, they would be allowed.

The first interpretation is understandable, but does not follow the intent of the special identifier. Therefore, it is recommended that implementors use "EVERYONE@" to mean literally everyone.

10. Access Mask Bit Discussion

The purpose of this section is to clarify the meaning of the different access mask bits. This will state the operations and a description of what the access mask bit controls. A major portion of the descriptions were taken from [RFC3530]. The following is a list of access mask bits that can be set on an ACE:

```
ACE4_READ_DATA
    Operation(s) affected:
        READ
        OPEN
    Discussion:
        Permission to read the data of the file.

ACE4_LIST_DIRECTORY
    Operation(s) affected:
        READDIR
    Discussion:
        Permission to list the contents of a directory.

ACE4_WRITE_DATA
    Operation(s) affected:
        WRITE
        OPEN
    Discussion:
        Permission to modify a file's data anywhere in
        the file's offset range.  This includes the
        ability to write to any arbitrary offset and as
        a result to grow the file.

ACE4_ADD_FILE
    Operation(s) affected:
        CREATE
        OPEN
    Discussion:
        Permission to add a new file in a directory.
        The CREATE operation is affected when
        nfs_ftype4 is NF4LNK, NF4BLK, NF4CHR, NF4SOCK,
        or NF4FIFO. (NF4DIR is not listed because it is
        covered by ACE4_ADD_SUBDIRECTORY.) OPEN is
        affected when used to create a regular file.

ACE4_APPEND_DATA
    Operation(s) affected:
        WRITE
        OPEN
    Discussion:
```

```
                The ability to modify a file's data, but only
                starting at EOF.  See Section 11 for further
                discussion on the relationship between
                ACE4_APPEND_DATA and ACE4_WRITE_DATA.

        ACE4_ADD_SUBDIRECTORY
            Operation(s) affected:
                CREATE
            Discussion:
                Permission to create a subdirectory in a
                directory.  The CREATE operation is affected
                when nfs_ftype4 is NF4DIR.

        ACE4_READ_NAMED_ATTRS
            Operation(s) affected:
                OPENATTR
            Discussion:
                Permission to lookup the named attributes directory.
                OPENATTR is affected when it is not used to
                create a named attribute directory.  This is
                when 1.) createdir is TRUE, but a named
                attribute directory already exists, or 2.)
                createdir is FALSE.

        ACE4_WRITE_NAMED_ATTRS
            Operation(s) affected:
                OPENATTR
            Discussion:
                Permission to create a named attribute directory.
                OPENATTR is affected when it is used to create
                a named attribute directory.  This is when
                createdir is TRUE and no named attribute
                directory exists.  The ability to check whether
                or not a named attribute directory exists
                depends on the ability to look it up,
                therefore, users also need the
                ACE4_READ_NAMED_ATTRS permission in order to
                create a named attribute directory.

        ACE4_EXECUTE
            Operation(s) affected:
                LOOKUP
            Discussion:
                Permission to execute a file or traverse/search
                a directory.

        ACE4_DELETE_CHILD
            Operation(s) affected:
```

```
            REMOVE
        Discussion:
            Permission to delete a file or directory within
            a directory.

    ACE4_READ_ATTRIBUTES
        Operation(s) affected:
            GETATTR of file system object attributes
        Discussion:
            The ability to read basic attributes (non-ACLs)
            of a file.

    ACE4_WRITE_ATTRIBUTES
        Operation(s) affected:
            SETATTR of time_access_set, time_backup,
            time_create, time_modify_set
        Discussion:
            Permission to change the times associated with
            a file or directory to an arbitrary value.

    ACE4_DELETE
        Operation(s) affected:
            REMOVE
        Discussion:
            Permission to delete the file or directory.

    ACE4_READ_ACL
        Operation(s) affected:
            GETATTR of acl
        Discussion:
            Permission to read the ACL.

    ACE4_WRITE_ACL
        Operation(s) affected:
            SETATTR of acl and mode
        Discussion:
            Permission to write the acl and mode attributes.

    ACE4_WRITE_OWNER
        Operation(s) affected:
            SETATTR of owner and owner_group
        Discussions:
            Permission to write the owner and owner_group
            attributes. On UNIX systems, this is the
            ability to execute chown or chgrp.

    ACE4_SYNCHRONIZE
        Operation(s) affected:
```

NONE

Discussion:

Permission to access file locally at the server with synchronized reads and writes.

11. Append Only Behavior

The NFS version 4 ACL model allows for the notion of append-only files, by allowing ACE4_APPEND_DATA and denying ACE4_WRITE_DATA to the same user or group.

If a file has an ACL such as the one described above and a WRITE request is made for somewhere other than EOF, the server SHOULD return NFS4ERR_ACCESS.

12. ACE4_DELETE/ACE4_DELETE_CHILD Behavior

There are two separate access mask bits that govern the ability to delete a file: ACE4_DELETE and ACE4_DELETE_CHILD. ACE4_DELETE is intended to be specified by the ACL for the object to be deleted, and ACE4_DELETE_CHILD is intended to be specified by the ACL of the parent directory.

In addition to ACE4_DELETE and ACE4_DELETE_CHILD, many systems also consider the "sticky bit" (MODE4_SVTX) and the appropriate "write" mode bit when determining whether to allow a file to be deleted. The mode bit for write corresponds to ACE4_WRITE_DATA, which is the same physical bit as ACE4_ADD_FILE. Therefore, ACE4_ADD_FILE can come into play when determining permission to delete.

In the algorithm below, the strategy is that ACE4_DELETE and ACE4_DELETE_CHILD take precedence over the sticky bit, and the sticky bit takes precedence over the "write" mode bits (reflected in ACE4_ADD_FILE).

Server implementations SHOULD grant or deny permission to delete based on the following algorithm.

```
if ACE4_EXECUTE is denied by the parent directory ACL:
    deny delete
else if ACE4_EXECUTE is unspecified by the parent directory ACL:
    deny delete
else if ACE4_DELETE is allowed by the target object ACL:
    allow delete
else if ACE4_DELETE_CHILD is allowed by the parent directory ACL:
    allow delete
else if ACE4_DELETE_CHILD is denied by the parent directory ACL:
    deny delete
else if ACE4_ADD_FILE is allowed by the parent directory ACL:
    if MODE4_SVTX is set for the parent directory:
        if the principal owns the parent directory OR
            the principal owns the target object OR
            ACE4_WRITE_DATA is allowed by the target object ACL:
                allow delete
            else:
                deny delete
    else:
        allow delete
else:
    deny delete
```

13. ACE4_ADD_FILE and ACE4_ADD_SUBDIRECTORY

As specified in Section 10, the permission granted by ACE4_WRITE_DATA is a superset of the permission granted by ACE4_APPEND_DATA. With directories, ACE4_WRITE_DATA is analogous to ACE4_ADD_FILE, and ACE4_APPEND_DATA is analogous to ACE4_ADD_SUBDIRECTORY.

A question this raises is whether ACE4_ADD_FILE is a superset of ACE4_ADD_SUBDIRECTORY. In other words, does the granting of ACE4_ADD_FILE imply the permission to create a subdirectory?

It is proposed that ACE4_ADD_FILE does not imply ACE4_ADD_SUBDIRECTORY.

14. POSIX Considerations

Disclaimer: This section is relevant to platforms with requirements to be POSIX compliant. These platforms are typically UNIX based, and the following discussion will be heavily biased toward those platforms.

14.1. Background Information

The standard POSIX (See [POSIX]) file access control mechanism uses the file permission bits contained in the file mode. The file permission bits are used to determine whether a process has read, write or execute/search permission to a file based on which class the process is in; file owner, file group or file other class.

A process is in the file owner class if the effective user ID of the process matches the user ID of the file. A process is in the file group class if the process is not in the file owner class and if the effective group ID or one of the supplementary group IDs of the process matches the group ID associated with the file. A process is in the file other class if the effective user ID of the process is not in the file owner class or the file group class.

The POSIX spec says that a process is in one and only one class, therefore, the access permissions that exist for that class are the only ones that will be considered when doing access checks.

14.2. Additional and Alternate File Access Control Mechanisms

In addition to the standard file access control mechanism, the POSIX spec allows for additional and alternate file access control mechanisms. According to Section 4.4 of the POSIX spec, a file can have one or both mechanisms in place.

The additional file access control mechanism is defined to be layered upon the file permission bits, but they can only further restrict the standard file access control mechanism. The alternate file access control mechanism is defined to be independent of the file permission bits and which if enabled on a file may either restrict or extend the permissions of a given user. Another major distinction between the additional and alternate file access control mechanisms is that any alternate mechanism must be disabled after the file permission bits are changed with a chmod. Additional mechanisms do not need to be disabled when a chmod is done.

14.3. NFSv4 ACLs vs. POSIX-draft ACLs

The goal of both ACL models is similar in that we want to be able to

give the owner of the file more fine grained access control than is available with the file permission bits. Much like POSIX draft ACLs, NFSv4 ACLs are made up of multiple Access Control Entries (ACEs), but the similarities don't go much further.

One major difference between POSIX draft and NFSv4 is that NFSv4 ACLs have two types of ACEs that play a role in access checking. Those two types are ALLOW and DENY. One important thing to note about this distinction is that in POSIX draft ACLs, a single entry defines what is allowed and also what is denied for the user or group that the entry applies to. NFSv4 ACLs separate what is allowed and what is denied by having the distinct ALLOW and DENY types of ACEs. The importance of this is that a user shouldn't infer from any single NFSv4 ACE that defines some set of permissions whether or not the permissions that weren't defined in that ACE are allowed or denied. This leads us to have to look at the ACL as a whole in order to determine a user's access.

For instance, in the following example, the first "bob@sun.com" entry defines the capability for user "bob@sun.com" regarding ACE4_READ_DATA, but you cannot infer from that entry whether "bob@sun.com" is granted ACE4_WRITE_DATA or not. One must continue through the ACL to see what the other capabilities are.

```
bob@sun.com:ACE4_READ_DATA::ALLOW

bob@sun.com:ACE4_WRITE_DATA::ALLOW

GROUP@:ACE4_EXECUTE:ACE4_IDENTIFIER_GROUP:DENY

bob@sun.com:ACE4_EXECUTE::ALLOW
```

This example also illustrates a couple of other differences between the two models. The first difference to be noted is that the ordering of the ACEs is different from what is typical with POSIX-draft ACLs. POSIX-draft ACLs have a defined ordering of the ACEs which is as follows: owner, supplemental users, owning group, supplemental groups, and other. This ordering is maintained by the kernel and cannot be changed by the user. With NFSv4 ACLs, there is no rigid order of the ACEs and the order is user defined. The second difference that this example illustrates is that there is no MASK_OBJ or mask entry in this ACL. This is because NFSv4 ACLs have no notion of a mask.

NFSv4 ACLs go beyond the ability to define access with regard to the standard read, write and execute/search permissions and allows the user to set ACLs to define file control permissions (i.e. - the ability to allow or deny a user the ability to write the ACL or

change the owner). The model also allows inheriting both standard and file control permissions to newly created files.

14.4. NFSv4 ACLs vs. POSIX

In various other vendors' implementations of NFSv4 ACLs, they have taken a chmod to mean the setting of a six member ACL, therefore throwing away the ACL and replacing it with six ACEs which reflect the mode. The user feedback on this behavior has been unfavorable. Some have gone as far as to say that it is unacceptable. We presume the dissatisfaction comes from a user spending time crafting an ACL only to get it stomped by a later chmod. Considering how many applications use chmod, we should not follow this behavior. In addition, security problems can arise if any explicit DENY ACEs are automatically removed as the result of a chmod (as shown in the example below).

We believe it is a requirement to preserve an object's ACL upon chmod.

A DENY type of ACE is considered to be an additional file access control mechanism, since it can only further restrict permissions. By categorizing DENY ACEs as additional, we have the ability to be able to keep the DENY ACEs without modification, except for on the abstract entities "OWNER@", "GROUP@" and "EVERYONE@" (see section Section 5.3 for further explanation).

The importance of classifying DENY ACEs as a additional file access control mechanism is best shown in the following example:

Suppose we have a file with the following ACL:

```
www@sun.com:ACE4_READ_DATA::DENY

OWNER@:<arbitrary mask>::DENY

OWNER@:<arbitrary mask>::ALLOW

GROUP@:<arbitrary mask>:ACE4_IDENTIFIER_GROUP:DENY

GROUP@:<arbitrary mask>:ACE4_IDENTIFIER_GROUP:ALLOW

EVERYONE@:<arbitrary mask>::DENY

EVERYONE@:<arbitrary mask>::ALLOW
```

We require the ability to keep the "www@sun.com:ACE4_READ_DATA::DENY" ACE in the event of a chmod so that "www@sun.com"'s permissions do

not get elevated by the deletion of the ACE upon execution of chmod.

The ALLOW type of ACE is considered to be an alternate file access control mechanism because it can further extend the permissions of a user. As previously mentioned, POSIX states that alternate mechanisms must be disabled at the time of chmod. This is different from requiring the deletion of any alternate mechanisms, and allows us to preserve the ACL. See Paragraph 1.5 in Section 5.3.

14.5. umask Considerations

The umask is used by UNIX operating system users to affect or mask down the file permission bits of newly created files. umask is not part of the NFS version 4 protocol. Instead, it is an entirely client-side concept.

umask can be briefly described as an attribute of a process which is a set of bits that are not to be set in the mode bits of a newly created file. If a process creates a new file via the open() system call, with an octal mode of 0777, and the process has a umask of 0022, the resulting file would have an octal mode attribute of 0755.

On client implementations that implement the concept of a umask, NFSv4 client implementations SHOULD apply the umask on newly created files, whether or not a newly created file will be affected by inheritable ACEs in the parent directory.

15. Normative References

[POSIX] "The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition", IEEE STD. 1003.1, January 2004.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3530] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and D. Noveck, "Network File System (NFS) version 4 Protocol", RFC 3530, STD 1, April 2003.

Authors' Addresses

Sam Falkner
Sun Microsystems, Inc.
500 Eldorado Blvd.
MS: UBRM05-171
Broomfield, CO 80021
USA

Email: sam.falkner@sun.com

Lisa Week
Sun Microsystems, Inc.
500 Eldorado Blvd.
MS: UBRM05-171
Broomfield, CO 80021
USA

Email: lisa.week@sun.com

Acknowledgment

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