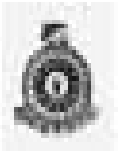

Mapping POSIX ACLs to CIFS Security Descriptors

A dissertation submitted for the Degree of Master of Computer Science

M.M. Nanayakkara
University of Colombo School of Computing
February, 2010

Declaration

The Dissertation is my original work and has not been submitted previously for a degree at this or any other university/institute. To the best of my knowledge it does not contain any material published or written by another person, except as acknowledge in the text.

Author's name: M. Mahesh Nanayakkara Date ……

Signature …….

This is to certify that this dissertation is based on the work of Mr M. Mahesh Nanayakkara under my supervision. The dissertation has been prepared according to the format stipulated and is of acceptable standard.

Certified by

Supervisor Name: Chamath Keppitiyagama Date ……...

Signature ……..

Abstract

Linux kernel "2.6" introduced a new file system driver cifs.ko which is a network file system, which provides access to files and directories on another computer as if they were on a locally attached disk. It can be used to mount Windows and Samba file systems and run demanding applications from those mounts (i.e. cifs.ko driver works like the client to those CIFS services points). Its security consideration in Access Control List (ACL) are checked at client-end. This can result in strange behaviours like loss of its semantics due to improper mapping between its security descriptors to UNIX extensions.

File permission bits are not enough to cover all mask flags that an ACL may grant. Mask flags such as WRITE_OWNER go beyond Read, Write, and Execute. CIFS specification defines an access control list model that is close to how access control lists work on Windows. A possible mapping to UNIX file permissions should be extracted from the ACL model.

POSIX compliant systems that implement ACL permission models must do so within the extension mechanisms that POSIX allows. As a result, it will not violate assumptions that applications may rightfully make about the system behaviour. The CIFS specification does not specify in sufficient detail how CIFS ACLs map to POSIX. Also it is not fully consistent with the extension mechanisms that POSIX defines. The purpose of this project is to clarify the relationship between CIFS ACLs and POSIX, and to detail how CIFS ACLs can be implemented correctly on POSIX compliant systems. Another goal of the project is to define the additional concepts and mechanisms needed to achieve a functional and correct POSIX compliant system.

Acknowledgement

I would like to thank to my supervisor, Dr Chamath Keppitiyagama, whose encouragement, guidance and support from the initial to the final level enabled me to develop an understanding of the subject.

Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of the project, especially Steve French, Andreas Grünbacher for there enormous support.

List of acronyms

VFS virtual file system.

CIFS common internet files system.

EA extended attributes

ACL access control list

POSIX Portable Operating System Interface

NFS network file systems

rwx read write execution

SID security id.

RID Relative id

UID user id

GID group id

SFU services for UNIX

Table of Content

List of Tables

List of Figures

Chapter 1– Introduction

Organizations today are using heterogeneous computing infrastructures that include machines running Linux, UNIX, as well as Microsoft operating systems. This raises the problem of sharing files across operating system boundaries, and controlling access to shared files. UNIX based systems use the traditional POSIX file permission model The Institute of Electrical and Electronics Engineers (IEEE) (2001) also known as 1003.1 for access control, which basically uses the file permission bits in the file mode to define access. Most current UNIX systems augment the traditional POSIX model with POSIX-draft ACLs IEEE (2004) also known as 1003.1e for defining advanced scenarios.

The ACL model that Windows uses is sufficiently different from POSIX-draft ACLs, and mapping between the two systems leads to various user-observable mapping artefacts, which can be very frustrating to users. It is desirable to avoid such unexpected permission granting and strange behaviours while mapping Windows ACL to POSIX-draft ACLs as much as possible.

The CIFS specification defines its own access control list model, including ACL semantics. The model chosen is much closer to Windows ACLs than to how the traditional UNIX permission model works. In addition to that, CIFS also supports the mode attribute, which corresponds with the UNIX file mode. POSIX compliant systems that implement extended permission models must do so within the extension mechanisms that POSIX allows in order not to violate assumptions that applications may rightfully make about the system behaviour (Gruenbacher, 2006). Violating these assumptions would lead to potential security threats. The definitions in the CIFS specification while sufficient for specifying the interactions with systems that implement the traditional UNIX permission model are not fully consistent with the extension mechanisms that POSIX defines.

Permission models

IEEE (2001) Traditional UNIX ACL permission Model

- Three classes of users (owner,group,other)
- Three permission per class (read write execute)
- Specifies in the posix 1003.1 standard
- Every unix like operating system has them
- Supported by NFS and other protocols.

IEEE (2004) Posix Draft ACLs Permission Model

- Entries for owner, owning group others
- Entries for additional users and groups
- Still read write execute only, but per entry
- POSIX 1003.1e working group
- Well integrated with POSIX permission model

CIFS ACLs permission model.

- Many Permissions
- Allow and Deny entries
- More complicated inheritance scheme
- Procedural rather than declarative
- Introduced with Windows NT/NTFS
- Used in the CIFS protocol

The 2.6 version of the Linux kernel introduced a new file system driver cifs.ko (French , 2007) to mount to Windows and Samba servers and run demanding applications from those mounts. It provides access to files and directories on another computer as if they were on a locally attached disk. Although commonly used to mount to non-Linux servers, such as Windows, the cifs virtual file system client (i.e. cifs vfs) is optimized for Samba and servers which implement the most of the CIFS POSIX extensions. Also Supporting these different environment is really important, different access control list models is one of the problems run in to these areas.

This project investigates the security aspect of this new file (i.e. cifs.ko) system in context of access control list. And map CIFS ACL to definition found in IEEE (2004) POSIX-draft ACLs also know as 1003.1e. By doing that we are trying to preserve the ACL information by right conversions. It should not add any additional permission or remove existing permissions. It is a real challenging to come up such algorithms because Windows ACL model (CIFS) and POSIX ACL model differ in many ways. This Project attempt to minimize the gap between those two entitles. There needs to analyse on the POSIX standards (found in Appendix) and the windows ACL model. Those findings are analysed and transform the findings in to algorithms for the ACL mapping found it the design section of this document. And the Linux CIFS file system architecture and its design for support ACL mapping listed under the same section of this document. Finally, the purpose of this document is to clarify the relationship between CIFS ACLs and POSIX, and to detail how CIFS ACLs can be implemented correctly on POSIX compliant systems. Mainly the first part of this document discusses how CIFS ACLs relate to POSIX, and the second part describes algorithms which can be used to implement CIFS ACLs in a POSIX compliant way. All of these will make Linux a much better file server in Windows and mixed environments.

Chapter 2– Literature Survey

Introduction

The Common Internet File System (CIFS), also known as Server Message Block (SMB) [WWW 01], is a network protocol whose most common use is sharing files on a Local Area Network (LAN). [WWW 02]The protocol allows a client to manipulate files just as if they were on the local computer. Operations such as read, write, create, delete, and rename are all supported – the only difference being that the files are not on the local computer and are actually on a remote server. The CIFS protocol works by sending packets from the client to the server. Each packet is typically a basic request of some kind, such as open file, close file, or read file. The server then receives the packet, checks to see if the request is legal, [WWW 03] verifies the client has the appropriate file permissions, and finally executes the request and returns a response packet to the client. The client then parses the response packet and can determine whether or not the initial request was successful. CIFS is a fairly high-level network protocol. In the OSI model, it is probably best described at the Application/Presentation layer.

Barry Feigenbaum originally designed SMB at IBM [WWW 04]. After, Microsoft has made considerable modifications to the version used most commonly. The new name first appeared around 1996/97 when Microsoft submitted draft CIFS specifications to the **I**nternet **E**ngineering **T**ask **F**orce (IETF).

The CIFS protocol is most commonly used with Microsoft operating systems [WWW 05]. Windows for Workgroups was the first Microsoft operating system to use CIFS, and each Microsoft operating system since then has been able to function as both a CIFS server and client. Microsoft operating systems use CIFS for remote file operations, authentication, and remote printer services. It would be fair to say the core of native Microsoft networking is built around its CIFS services.

Because of Microsoft's large corporate and home user base, the CIFS protocol is found virtually everywhere. Many of UNIX type operating system implements a CIFS

client/server via the Samba program. Apple computers also have CIFS clients and servers available, which might make CIFS the most common protocol for file sharing available.

Because of the importance of the SMB protocol in interacting with the widespread of platform,[WWW 04] the Samba project originated with the aim of reverse engineering and providing a free implementation of a compatible SMB client and server for use with non-Microsoft operating systems. Since 1991, they have been gathering information and implementing their own CIFS server, called Samba. Samba is published as Open Source under the terms of the GNU General Public License. Samba Team members has shared the knowledge they gain via the World Wide Web. Samba is included with most distributions of Linux, and several commercial UNIX flavours as well.

The POSIX 1003.1e/1003.2c

POSIX is the Portable Operating System Interface [WWW 06], the open operating interface standard accepted world-wide. [WWW 07] It is produced by IEEE and recognized by ISO and ANSI. POSIX support assures code portability between systems, i.e. only if features are identical across many platforms, developers will widely use them in their software. That is the goal of POSIX Extensions to POSIX.1: talk about Access Control Lists (ACL), Audit, Capability, Mandatory Access Control (MAC), and Information Labelling.

However standardizing all these diverse areas was too ambitious a goal. [WWW 08] In January 1998, sponsorship for this work was withdrawn. While some parts of the documents produced by the working group until then were already of high quality, the overall works were not ready for publication as standards. It was decided that draft POSIX 1003.1e 17, the last version of the documents the working group had produced, should be made available to the public.

Several UNIX system vendors have implemented various parts of the security extensions, augmented by vendor-specific extensions. The resulting versions of their operating systems have often been labelled ``trusted'' operating systems, e.g., Trusted Solaris, Trusted Irix, Trusted AIX. Some of these ``trusted'' features have later been incorporated into the vendors' main operating systems.

ACLs are supported on different file system types on almost all UNIX-like systems nowadays. Some of these implementations are compatible with draft 17 of the specification, while others are based on older drafts. Unfortunately, this has resulted in a number of subtle differences among the different implementations.

User and Group Accounts

In Windows accounts are identified by a name and a globally unique ID known as a Security Identifier or SID [WWW 09]. All user and group accounts share a common namespace - no two accounts can have the same SID. A SID is represented by a large variable length number consisting of a DomainSID part and a Relative ID (RID) part. The DomainSID identifies a specific Windows domain. The RID part is a unique small number (only 32 bits) which is used to identify a specific group or user member in a domain. Windows has many domains, a system local domain and many global trusted domains. The DomainSid is a sufficiently large number that all SIDs can be unique across all domains in the local network. There is some special well-defined SIDs that is members of the special domain and are universal across all Windows systems (Windows NT, Windows 2000, Windows XP and Windows 2003). The SYSTEM user account, the World group account and the Administrators group account are examples of these well-known SIDs. In Windows, accounts are uniquely identified by their SID. Furthermore, within each domain the user and group accounts share the same namespace.

In UNIX, user accounts and group accounts are represented by a name and a numerical identifier (ID) [WWW 10]. These accounts are in separate name spaces so that the same name can be used as a user account and a group account. Additionally, a User ID (UID) and Group ID (GID) could have the same numeric value. User and group IDs are distinguished from each other by the context in which they are used.

User and group accounts are used for identification and ownership of files. Files on Windows have an owner and a primary group just like on UNIX. However, on Windows, because user and group accounts share the same namespace, it is possible for the owner of a file to be a group account, and the file’s primary group could be a user account.

Windows File Security

Every file and directory in the Windows NTFS file system has security information associated with it [WWW 09], including an owner security identifier (owner SID), a group security identifier (group SID), a Discretionary Access Control List (DACL), and a set of file attribute flags like ATTRIB_READONLY and ATTRIB_SYSTEM.

The owner SID represents the owner of the file and the owner is always permitted to change or update any part of the file's attributes including the DACL The group SID is not used in the Windows environment but is required in SFU's UNIX environments. The DACL consists of zero or more individual Access Control Entries (ACEs). Each ACE contains a SID, an access mask, and some flags that control the inheritance of ACEs. These inheritance flags control how ACEs are propagated from a directory to the files and subdirectories contained within it. Also access-allowed ACE grants the access rights specified in the ACE's access mask and the access-denied ACE denies the access rights. The access-denied ACE is enforced regardless of any access-allowed ACE.

UNIX File Security

The file access control mechanism used in UNIX system is based on file permission bits. Associated with each file or directory is a user ID (i.e. the owner), a group ID and a set of twelve bits known as the permission bits [WWW 11]. Three of the bits specify the access by the file's owner, three bits specify the access allowed to users that are members of the file's group, and the last three bits control access to everyone else (others). Each set of three permission bits consists of a read permission (r), a write permission (w), and an execute permission (x). For normal files, these permissions are straightforward. For directories, write permissions dictate whether or not users can create or delete subdirectories and files.

Besides the standard nine permission bits, there are three bits with special meaning called the setuid bit, setgid bit and "sticky" bit. The setuid and setgid permissions only apply to executable programs (binaries and/or scripts). When the setuid bit is set the program (file)

is executed with the user ID of the owner of the file and NOT the user ID of the user that is accessing the file. [WWW 09] For example, a user *joe* may assign the setuid permissions on one of his program files. Another user *john* may logon and run *joe's* program. Since the setuid bit is set, the program will run as if *joe* has executed it. Similarly, when the setgid bit is set the program (file) is executed with the ID of the group of the file and NOT the group ID of the user that is accessing the file. The sticky bit is usually reserved for use with directories. When this bit is set, files or subdirectories in the directory can only be deleted by the owner of the file or subdirectory. When this bit is not set, anyone with write permission on the directory can remove files or subdirectories. This is used so that users who share access to a directory cannot delete each others' files

Status of ACLs on Linux

Patches that implement POSIX 1003.1e draft 17 ACLs have been available for various versions of Linux for several years now [WWW 08]. They were added to version 2.5.46 of the Linux kernel in November 2002. SuSE and the United Linux consortium have integrated the 2.4 kernel ACL patches earlier than others, their current products cater the most complete ACL support available for Linux. The Linux *getfacl* and *setfacl* command line utilities do not strictly follow POSIX 1003.2c draft 17, [WWW 07] which shows mostly in the way they handle default ACLs.

How ACLs Work

The traditional POSIX file system object permission model defines three classes of users called owner, group, and other. [WWW 08] Each of these classes is associated with a set of permissions. The permissions defined are read (r), write (w), and execute (x). In this model, the *owner class* permissions define the access privileges of the file owner, the *group class* permissions define the access privileges of the owning group, and the *other class* permissions define the access privileges of all users that are not in one of these two classes.

An ACL consists of a set of entries. The permissions of each file system object have an

ACL representation [WWW 12]. Each of the three classes of users is represented by an ACL entry. Permissions for additional users or groups occupy additional ACL entries.

Below table shows the defined entry types and their text forms. Each of these entries consists of a type, a qualifier that specifies to which user or group the entry applies, and a set of permissions. The qualifier is undefined for entries that require no qualification.

ACLs equivalent with the file mode permission bits are called *minimal* ACLs. They have three ACL entries. ACLs with more than the three entries are called *extended* ACLs. Extended ACLs also contain a mask entry and may contain any number of named user and named group entries [WWW 08] Table 2-1.

Entry type	**Text form**
Owner	user::*rwx*
Named user	user:*name*:*rwx*
Owning group	group::*rwx*
Named group	group:*name*:*rwx*
Mask	mask::*rwx*
Others	other::*rwx*

Table 2-1 Types of ACL Entries

These named group and named user entries are assigned to the *group class*, which already contains the owning group entry. Different from the POSIX.1 permission model, the group class may now contain ACL entries with different permission sets, so the group class permissions alone are no longer sufficient to represent all the detailed permissions of all ACL entries it contains. Therefore, the meaning of the group class permissions is redefined: under their new semantics, they represent an upper bound of the permissions that any entry in the group class will grant.

In minimal ACLs, the group class permissions are identical to the owning group permissions. In extended ACLs, the group class may contain entries for additional users or groups. This result in a problem: some of these additional entries may contain permissions that are not contained in the owning group entry, so the owning group entry permissions may differ from the group class permissions.

This problem is solved by the virtue of the mask entry. With minimal ACLs, the group class permissions map to the owning group entry permissions. With extended ACLs, the group class permissions map to the mask entry permissions, whereas the owning group entry still defines the owning group permissions. The mapping of the group class permissions is no longer constant. Figure 2-1 [WWW 08] shows these two cases.

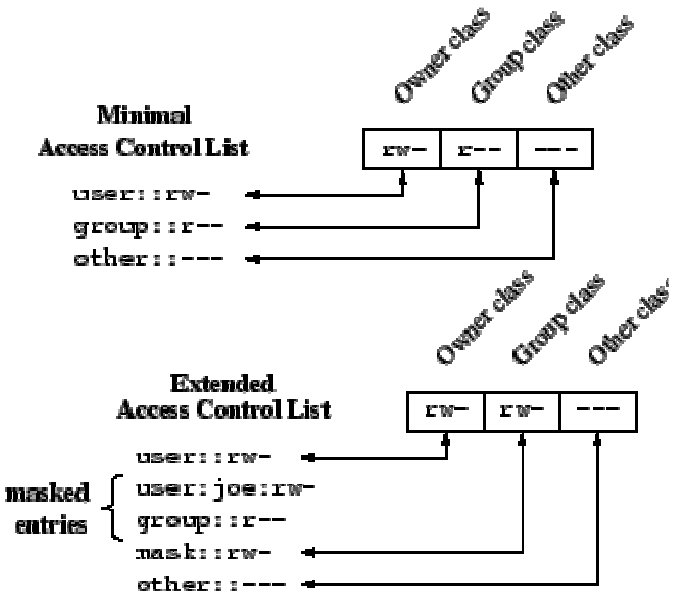

Figure 2-1Minimum and extended ACL

When an application changes any of the owner, group, or other class permissions (e.g., via the *chmod* command), the corresponding ACL entry changes as well. Likewise, when an application changes the permissions of an ACL entry that maps to one of the user classes, the permissions of the class change.

The group class permissions represent the upper bound of the permissions granted by any entry in the group class. With minimal ACLs this is trivially the case. With extended ACLs, this is implemented by masking permissions (hence the name of the mask entry): permissions in entries that are a member of the group class which are also present in the mask entry are effective. Permissions that are absent in the mask entry are masked and thus do not take effect [WWW 08] (Table 2-2).

Entry type	**Text form**	**Permissions**
Named user	user:joe:r-x	r-x
Mask	mask::rw-	rw-
Effective permissions		r-

Table 2-2 Effective permissions

The owner and other entries are not in the group class. [WWW 09] Their permissions are always effective and never masked. There are two types of ACLs POSIX system maintains one is ACLs that define the current access permissions of file system objects. This type is called *access* ACL. A second type called *default* ACL is also defined. They define the permissions a file system object inherits from its parent directory at the time of its creation. Only directories can be associated with default ACLs. Default ACLs for non-directories would be of no use, because no other file system objects can be created inside non-directories. Default ACLs are not directly involve in access checks.

When a directory is created inside a directory that has a default ACL, the new directory inherits the parent directory's default ACL both as its access ACL and default ACL. Objects that are not directories inherit the default ACL of the parent directory as their access ACL only.

The permissions of inherited access ACLs are further modified by the *mode* parameter that each system call creating file system objects has. The *mode* parameter contains nine permission bits that stand for the permissions of the owner, group, and other class permissions. The effective permissions of each class are set to the intersection of the permissions defined for this class in the ACL and specified in the *mode* parameter.

If the parent directory has no default ACL, the permissions of the new file are determined as defined in POSIX.1. The effective permissions are set to the permissions defined in the *mode* parameter, minus the permissions set in the current *umask*.

Access Check Algorithm

A process requests access to a file system object. [WWW 13] Two steps are performed. Step one selects the ACL entry that most closely matches the requesting process. The ACL entries are looked at in the following order: owner, named users, (owning or named) groups, others. Only a single entry determines access. Step two checks if the matching entry contains sufficient permissions.

A process can be a member in more than one group, so more than one group entry can match. If any of these matching group entries contain the requested permissions, one that contains the requested permissions is picked (the result is the same no matter which entry is picked). If none of the matching group entries contains the requested permissions, access will be denied no matter which entry is picked.

The access check algorithm can be described in pseudo-code as follows [WWW 12].

If

the user ID of the process is the owner, the owner entry determines access

else if

the user ID of the process matches the qualifier in one of the named user entries, this entry determines access

else if

one of the group IDs of the process matches the owning group and the owning

group entry contains the requested permissions, this entry determines access

else if

one of the group IDs of the process matches the qualifier of one of the named group entries and this entry contains the requested permissions, this entry determines access

else if

one of the group IDs of the process matches the owning group or any of the named group entries, but neither the owning group entry nor any of the matching named group entries contains the requested permissions, this determines that access is denied

else

the other entry determines access.

If

the matching entry resulting from this selection is the owner or other entry and it contains the requested permissions, access is granted

else if

the matching entry is a named user, owning group, or named group entry and this entry contains the requested permissions and the mask entry also contains the requested permissions (or there is no mask entry), access is granted

else

access is denied

Implementation and design of ACL

The design of how ACLs passed between user space and the kernel, and inside the kernel, between the virtual file system (VFS) and the low-level file system layer [WWW 14]. FreeBSD, Solaris, Irix, and HP-UX all have separate ACL system calls. Linux does not have ACL system calls. [WWW 08] Instead, ACL's are passed between the kernel and user space as EAs (Extended attributes). This reduces the number of system interfaces, but

with the same number of end operations. While the ACL system calls provide a more explicit system interface, the EA interface is easier to adapt to future requirements, such as non-numerical identifiers for users and groups in ACL entries. The rationale for using separate ACL system calls in FreeBSD was that some file systems support EAs but not ACLs, and some file systems support ACLs but not EAs, so EAs are treated as pure binary data. EAs and ACLs only become related inside a file system. The rationale for the Linux design was to provide access to all meta data pertinent to a file system object through the same interface. Different classes of attributes that are recognized by name are reserved for system objects such as ACLs.

The access ACL of a file system object is accessed for every access decision that involves that object. Access checking is performed on the whole path from the namespace root to the file in question. It is important that ACL access checks are efficient. To avoid frequently looking up ACL attributes and converting them from the machine-independent attribute representation to a machine-specific representation, the Ext2, Ext3, JFS, and ReiserFS [WWW 08] implementations cache the machine-specific ACL representations. This is done in addition to the normal file system caching mechanisms, which use the page cache, the buffer cache, or both. XFS does not use this additional layer of caching.

Samba and ACLs

Microsoft Windows supports ACLs on its NTFS file system, and in its Common Internet File System (CIFS) protocol, which formerly has been known as the Server Message Block (SMB) protocol [WWW 08]. CIFS is used to offer file and print services over a network. Samba is an Open Source implementation of CIFS. It is used to offer UNIX file and print services to Windows users. Samba allows POSIX ACLs to be manipulated from Windows. This feature adds a new quality of interoperability between UNIX and Windows.

The ACL model of Windows differs from the POSIX ACL model in a number of ways, so it is not possible to offer entirely seamless integration. The most significant differences

between these two kinds of ACLs are:

- Windows ACLs support over ten different permissions for each entry in an ACL, including things such as append and delete, change permissions, take ownership, and change ownership. Current implementations of POSIX.1 ACLs only support read, write, and execute permissions.

- In the POSIX permission check algorithm, the most significant ACL entry defines the permissions a process is granted, so more detailed permissions are constructed by adding more closely matching ACL entries when needed. In the Windows ACL model, permissions are cumulative, so permissions that would otherwise be granted can only be restricted by DENY ACL entries.

- POSIX ACLs do not support ACL entries that deny permissions. A user can be denied permissions be creating an ACL entry that specifically matches the user.

- Windows ACLs have had an inheritance model that was similar to the POSIX ACL model. Since Windows 2000, Microsoft uses a dynamic inheritance model that allows permissions to propagate down the directory hierarchy when permissions of parent directories are modified. POSIX ACLs are inherited at file create time only.

- In the POSIX ACL model, access and default ACLs are orthogonal concepts. In the Windows ACL model, several different flags in each ACL entry control when and how this entry is inherited by container and non-container objects.

- Windows ACLs have different concepts of how permissions are defined for the file owner and owning group. The owning group concept has only been added with Windows 2000. This leads to different results if file ownership changes.

- POSIX ACLs have entries for the owner and the owning group both in the access ACL and in the default ACL. At the time of checking access to an object, these

entries are associated with the current owner and the owning group of that object. Windows ACLs support two pseudo groups called Creator Owner and Creator Group that serve a similar purpose for inheritable permissions, but do not allow these pseudo groups for entries that define access. When an object inherits permissions, those abstract entries are converted to entries for a specific user and group.

- The permissions in the POSIX access ACL are mapped to Windows access permissions. The permissions in the POSIX default ACL are mapped to Windows inheritable permissions.

- Minimal POSIX ACLs consist of three ACL entries defining the permissions for the owner, owning group, and others. These entries are required. Windows ACLs may contain any number of entries including zero. If one of the POSIX ACL entries contains no permissions and omitting the entry does not result in a loss of information, the entry is hidden from Windows clients. If a Windows client sets an ACL in which required entries are missing, the permissions of that entry are cleared in the corresponding POSIX ACL.

- The mask entry in POSIX ACLs has no correspondence in Windows ACLs. If permissions in a POSIX ACL are ineffective because they are masked and such an ACL is modified via CIFS, those masked permissions are removed from the ACL.

- Because Windows ACLs only support the Creator Owner and Creator Group pseudo groups for inheritable permissions, owner and owning group entries in a default ACL are mapped to those pseudo groups. For access ACLs, these entries are mapped to named entries for the current owner and the current owning group.

- If an access ACL contains named ACL entries for the owner or owning group, the permissions defined in such entries are not effective unless file ownership changes, so such named entries are ignored. When an ACL is set by Samba that contains

Creator Owner or Creator Group entries, these entries are given precedence over named entries for the current owner and owning group, respectively.

- POSIX access ACL and default ACL entries that define the same permissions are mapped to a Windows ACL entry that is flagged as defining both access and inheritable permissions.

Conclusion

On UNIX-like systems, it is easier to work around problems than on other popular systems, but these workarounds cause complexity and may contain bugs. It may be better to solve some of the existing problems at their root [WWW 08]. All extensions must be designed carefully to simplify the integration with existing systems like Windows/CIFS.

The UNIX way of identifying users and groups by numeric IDs is a problem in large networks [WWW 09]. Like the whole POSIX.1 permission model, current implementations of POSIX.1e ACLs are based on these unique IDs. Maintaining central user and group databases becomes increasingly difficult with increasing network size.

In CIFS, users and groups are identified by globally unique security identifers (SIDs). Processes have a number of SIDs, which determine their privileges. CIFS ACLs may contain SIDs from different domains.

Current implementations of POSIX ACLs only support numeric user or group identifiers within the local domain. Allowing non-local identifiers in ACLs seems possible but difficult. A consequent implementation would require substantial changes to the process model [WWW 08]. At a minimum, in addition to non-local user and group identifiers in ACL entries, file ownership and group ownership for non-local users and groups would have to be supported.

Chapter 3– Requirements and analysis

Windows to UNIX Mapping

In Windows, file access permission is controlled by the DACL associated with each file This DACL can specify access rights to many different users and/or groups [WWW 09]. But in a UNIX system environment, there are only three classes of users and only three permission bits for each class. When UNIX application requests file permission information, a mapping from the DACL to these nine permission bits must be performed. This mapping starts by searching through all the ACEs in the DACL looking for the access-allowed and access-denied ACEs and then examining only the ACEs that contain either the file's owner SID, the file's group SID or one of the WorldSID or the AuthenticatedUsers-SIDs. For each of these selected ACEs, the access rights in the ACE are collected and saved into one or more of three different collections: the "owner" collection, the "group" collection and the "other" collection. These collections correspond to the three UNIX file permission classes and are used at the end when mapping access rights into permission bits. Each of these collections contains a set of granted rights and a set of denied rights to maintain the distinctions between the two different types of ACEs. When an access-denied ACE is found, the access rights are added to the set of denied rights only if the access right is not already present in the set of granted rights. For an access-allowed ACE, the access rights are added to the set of granted rights only if this access right is not already present in the set of denied rights. The order of ACEs in the DACL is important because it is the first occurrence of the access right that determines whether the effective resultant access right is allowed or denied.

The access rights from the file's owner SID are saved into the "owner" collection and the access rights from the file's group SID are saved into the "group" collection. Note that if the file owner and file group SIDs are the same, then the access rights are saved in both the owner" and "group" collections. Once all the ACEs have been scanned, then the set of granted rights in each of these collections are used to set the nine UNIX file permission bits.

Representation of ACLs

The representation used for CIFS ACLs in this document consists of one ACL entry per line, which contains a colon separated list of the individual who value, access mask value, flag, and type fields.

- The who field is represented by a id (e.g., uid, gid).
- The access mask field is represented by a list of slash separated access mask names (e.g., READ_DATA/WRITE_ACL/EXECUTE).
- The flag field is represented by a list of slash separated flag names (e.g., FILE_INHERIT_ACE/IDENTIFIER_GROUP).
- The type field is represented as ALLOW, DENY, AUDIT, or ALARM.

The Portable Operating System Interface (POSIX) standard defines the framework within which ACLs may grant or deny permissions on POSIX compliant systems this can find in IEEE (2004) 1003.1e document. POSIX defines extension mechanisms, and the conditions that these extension mechanisms must fulfils in order to comply. We must determine which aspects of ACLs map onto which POSIX concepts, and how ACLs can extend the POSIX permission model without violating POSIX requirements and expectations.

For reference, Appendix II (POSIX Definitions) repeats the definitions of IEEE (2004) 1003.1e document, which are relevant to file permissions.

Mapping Principals/Entry-tag type to File Classes

POSIX defines that the owner, group, and other file permission bits in the file mode are to be used with the corresponding file classes of processes. In order to apply this definition to files with CIFS ACLs, we define the following mapping of principals to file classes:

- Entry tag that match the owner attribute (and thus match ACL entries with a who id value of OWNER ids) are in the file owner class.
- Entry tag that are not in the file owner class and match the owner group attribute (and thus match ACL entries with a who value of GROUP ids), or match one or more ACL entries with a who value other than EVERYONE ids, are in the file group class.
- Entry tag that are not in the file owner or file group class (and thus only match ACL entries with a who value of EVERYONE ids) are in the file other class.

Synchronization between the Mode and ACL Attributes

Entry tag types are associated with a set of Read, Write, and Execute permissions by their file class. This set of permissions defines an upper bound to the permissions that may be granted, and ACLs may further restrict these permissions. POSIX requires that for new files and after a setting the file mode, no more permissions than allowed by the file permission bits must be granted by the ACL.

Ensuring that an ACL does not granted more permissions than the file mode allows may require removing excess access mask flags from ACL entries, inserting DENY entries at certain positions in the ACL, etc. Goal for such an algorithm is that the result of applying a file mode to an ACL should allow as many of the original permissions as possible, without exceeding the file mask.

Setting ACLs should update the mode attribute so that it reflects the permissions that the ACL grants as closely as possible. The file permission bits are not enough to cover all mask flags that an ACL may grant, because mask flags such as WRITE_OWNER go beyond Read, Write, and Execute. Nevertheless, the computed mode should reflect which mask flags that are covered by Read, Write, and Execute are granted to which file classes.

We allow the mode to be a superset of the permissions that are actually granted because computing the precise maximum is difficult, and POSIX allows implementations to grant less permission to processes than the file permission bits would imply.
It is a common misbelieve to assume that the group file permission bits reflect the permissions of the owning group: in fact, the group file permission bits correspond with the permissions of the file group class. The file group class also includes principals that do not match the owning group. This is also true with the user class.

Disabling Access Mask Flags Using Masking

A server that wishes to implement the mode attribute in a POSIX compliant must, when the file mode is set, ensure that no process is granted more permissions than allowed by the file mode itself. In the process, much or all of the information present in the original ACL can be lost. This is often undesirable. Traditional POSIX file systems have the property that restoring a mode to a previous value will restore all permissions to the previous value, and some applications may depend on this property.

Therefore, many file systems that support both ACLs and mode bits implement them in such a way that setting a more restrictive mode and then restoring the original mode will also restore as much of the original ACL as possible. File systems do this by storing a "mask" which is independent from the rest of the ACL, and modifying only the mask when the file mode is set. This allows the file system to enforce restricted permissions without having to modify the original ACL Each mask defines the maximum set of mask flags that a principal in the respective class may be granted, much like the file mode. Setting the file mode sets the file masks, and vice versa.

Mapping Between Access Mask Flags and Permissions

CIFS ACLs offer an extended set of access mask flags which goes beyond what can be expressed with the POSIX read, write, and execute permissions. Some of these mask flags are always allowed under POSIX, some correspond to POSIX permission or are subset of POSIX permission, and some have no POSIX equivalent [WWW 15] Table 3-1.

Access Mask Flag	**Classification**
READ_DATA	Read
LIST_DIRECTORY	Read
WRITE_DATA	Write
ADD_FILE	Write
APPEND_DATA	Write
ADD_SUBDIRECTORY	Write
READ_NAMED_ATTRS	Implementation Defined
WRITE_NAMED_ATTRS	Implementation Defined
EXECUTE	Execute
DELETE_CHILD	Write
READ_ATTRIBUTES	Always Allowed
WRITE_ATTRIBUTES	Alternate
DELETE	Alternate
READ_ACL	Always Allowed
WRITE_ACL	Alternate
WRITE_OWNER	Alternate
SYNCHRONIZE	Does Not Apply

Table 3-1 Mapping between Access Mask Flags and Permissions

Mask flags classified as Read, Write, or Execute are equivalent to or a subset of the respective permission. Mask flags classified as Alternate have no equivalent in POSIX, and go beyond Read, Write, and Execute. Mask flags classified as Always Allowed are always allowed under POSIX.

The READ_NAMED_ATTRS and WRITE_NAMED_ATTRS flags are classified as Implementation Defined. Depending on how access to named attributes is controlled, these mask flags may either have no equivalent in POSIX, or they may be subsets of the Read and Write permissions, respectively. If other means such as file ownership determine access, the two flags should be classified as Alternate.

The SYNCHRONIZE is classified as Does Not Apply. It defines the permission to access a file locally at the server with synchronized reads and writes. Synchronized reads do not have a correspondence in POSIX.

A mode SETATTR must disable mask flags classified as Alternate, and must only keep mask flags classified as Read, Write, and Execute enabled if the file class corresponding with the entry tag includes the respective permissions. It is not required to disable mask flags classified as Always Allowed or Does Not Apply.

An ACL SETATTR should compute an upper bound of the mask flags granted to each file class of principals. For each file mask in this upper bound that is classified as Read, Write, or Execute, the corresponding permission should be set in the corresponding set of permissions in the file mask.

Chapter 4– Design and Implementation

Linux file systems based by Virtual file system (VFS) (Daniel, 2002), which is an abstraction layer [WWW 16]. On top of VFS more concrete file system can work. The purpose of a VFS is to allow for client applications to access different types of concrete file systems in a uniform way. A VFS can for example be used to access local and network storage devices transparently without the client application noticing the difference [WWW 17]. Or it can be used to bridge the differences in Windows, Mac OS and UNIX file systems, so that applications could access files on local file systems of those types without having to know what type of file system they are accessing.

Extended file attributes is a file system feature that enables users to associate computer files with metadata not interpreted by the file system, whereas regular attributes have a purpose strictly defined by the file system such as file permission or records of creation and modification times. Typical uses can be storing the author of a document, the character encoding of a plain-text document, ACLs or a checksum

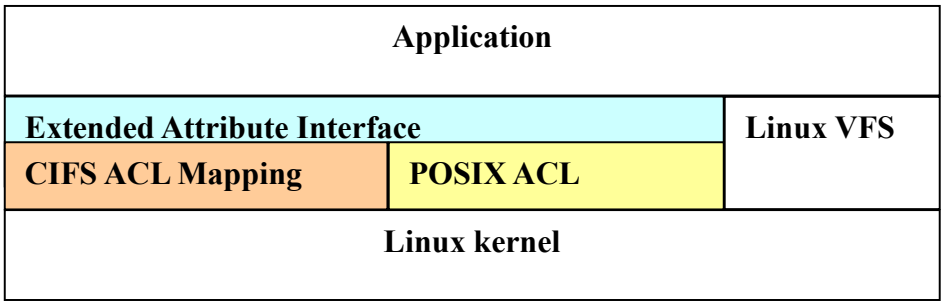

Figure 4-1 VFS ACL mapping

We select this extend attribute to store and check or the ACL when cifs used to working with a windows or samba share. To work with ACLs we need to store them in a data structure and group them accordingly by the convenience of usability and the maintainability. We select the below structures to hold the data.

Following data structures use to contain the CIFS windows security descriptors.

```
struct cifs_ntsd {
        int revision; /* revision level */
        int type;
        int osidoffset;   // owner security id.
        int gsidoffset;   // group security id.
        int sacloffset;   // system access control list
        int dacloffset;   // discretionary access control list.
 }
```

Note what is DACL and what is SACL can found from Appendix.

Following data structure contains CIFS security id data.

```
 struct cifs_sid {
        int revision; /* revision level */
        int num_subauth;
        int authority[7];
        int sub_auth[6];  /* sub_auth[num_subauth] */
}
```

Following data structure contains the CIFS acl control list data.

```
 struct cifs_acl {
        int revision; /* revision level */
        int size;
        int num_aces;
}
```

Following data structure contains the access control entries.

```
struct cifs_ace {
        int type;     //access type i.e. access allow or access denies.
        int flags;
        int size;
        int access_req;
        struct cifs_sid sid;   /* i.e. user or group who gets these permissions */
```

Design Overview

For each of the ACEs in the file's DACL, ignore the ACE if it is not an access-denied or access-allowed type. If the ACE contains the Authenticated Users SID then add the allowed or denied access right bits into the "owner", "group" and "other" collections. If this is an access-denied ACE, then add each access right to the set of denied rights in each collection but only if the access right is not already present in the set of granted rights in that collection. Similarly If this is an access-allowed ACE, then add each access right to the set of granted rights in each collection but only if the access right is not already present in the set of denied rights in that collection.

- If the ACE contains the file's group SID, then save the access rights in the "group" collection as appropriate in the corresponding set of granted or denied rights
- If the ACE contains the file's owner SID, then save the access rights in the "owner" collection as appropriate in the corresponding set of granted or denied rights.

Once all the granted Windows access right bits have been collected, then the UNIX permission bits are assembled. For each class, if the Read_Data bit is granted, then the corresponding "r" permission bit is set. If Write_Data access rights are granted then the "w" permission bit is set. And finally, if the Execute access right is granted, then the "x" permission bit is set.

It is desirable to set the permission bit accordingly for the access flags when getting the inode information. So it is cater at the all the regular operations such as makedir , readfile, and revalidate etc.

Getting the inode information need to be retrieve with the translated CIFS ACL (similar to NTFS ACL) for a file, into mode bits of the Linux extension (i.e. cifs,ko). To do this we need to extract the ACL details from the extended attributes from the inode operation table (i.e getxattr()).

And If we can retrieve the ACL, we can retrive the Access Control Entries (ACE). By retrieving the extended attribute ACL we should able to get type cifs_ntsd of data block and that data block should contains two type of cifs_sid data blocks for each type (i.e. owner, and group) also type cifs_acl DACL which contains the list of ACEs. After getting the list of ACE's we can work on to map the access mask
flag to file mode bits to meet our objectives.

ACE mapping Algorithm

```
if(dacl==null)
        give all the permission. // inode->i_mode |= S_IRWXUGO;
Else
if(num_ACL > 0)
        set initial masks.
         umode_t user_mask = S_IRWXU; // 00700
         umode_t group_mask = S_IRWXG; // 00070
         umode_t other_mask = S_IRWXO; // 00007
For each ace entry Do

if (ownerId){

  Update file mode with access flag accordance with ALLOW or DENIE entries

}

if (goupId){

  Update file mode with access flag accordance with ALLOW or DENIE entries

}

if (other){

  Update file mode with access flag accordance with ALLOW or DENIE entries

}

Ends do.
```

Algorithm to map ALLOW and DENIE entries to mode bits.

The order of ACEs is important. The canonical order is to begin with DENY entries followed by ALLOW, this is expect to given from the CIFS windows share.

```
if(ace_flag == ACCESS_DENIED)
{
 if (flags & GENERIC_ALL)
                    pbits_to_set &= ~S_IRWXUGO;

              if ((flags & GENERIC_WRITE)
                    pbits_to_set &= ~S_IWUGO;
               if ((flags & GENERIC_READ))
                    pbits_to_set &= ~S_IRUGO;
             if ((flags & GENERIC_EXECUTE))
                    pbits_to_set &= ~S_IXUGO;
} else
 /* else ACCESS_ALLOWED type */

      if (flags & GENERIC_ALL) {
            *pmode |= (S_IRWXUGO & (*pbits_to_set));

             return;
      }
       if (flags & GENERIC_WRITE)
            inode_pmode |= (S_IWUGO & (*pbits_to_set));
       if (flags & GENERIC_READ)
             inode_pmode |= (S_IRUGO & (*pbits_to_set));
      if (flags & GENERIC_EXECUTE)
            inode_pmode |= (S_IXUGO & (*pbits_to_set));
}
```

Extended Attributes

To do this we have use the ACL details from the extended attributes from the inode operation table (i.e. getxattr()). This is the high level design of targeted solution. All the ACL information's are passed via the Extended Attributed Interface. So after the CIFS permissions are defined, POSIX applications will be governed by the CIFS ACL model, but it will only see traditional POSIX or POSIX ACL's. As we earlier discuss ACLs are pieces of information of variable length that are associated with file system objects. Dedicated strategies for storing ACLs on file systems might be devised. Each shadow inode stores an ACL in its data blocks. Multiple files with the same ACL may point to the same shadow inode.

Because other kernel and user space extensions in addition to ACLs benefit from being able to associate pieces of information with files, Linux and most other UNIX-like operating systems implement a more general mechanism called Extended Attributes (EAs) [WWW 18]. On these systems, ACLs are implemented as EAs. Extended attributes are name and value pairs associated permanently with file system objects, similar to the environment variables of a process. The EA system calls used as the interface between user space and the kernel copy the attribute names and values between the user and kernel address spaces.

At the file system level, the obvious and straight-forward approach to implement EAs is to create an additional directory for each file system object that has EAs and to create one file for each extended attribute that has the attribute's name and contains the attribute's value. Because on most file systems allocating an additional directory plus one or more files requires several disk blocks, such a simple implementation would consume a lot of space, and it would not perform very well because of the time needed to access all these disk blocks. Therefore, most file systems use different mechanisms for storing EAs [WWW 08].

This is the high level design of targeted solution. All the ACL information's are passed via the Extended Attributed Interface. So after the CIFS permissions are defined, POSIX applications will be governed by the CIFS ACL model, but it will only see traditional POSIX or POSIX ACL's.

Chapter 5– Testing

A CIFS windows share needs to mount by this file system driver and investigate the acl control lists and its access control entries at the windows share. Also check the corresponding acls at the linux end. We can test this with the getfacl utility found at the linux.

Also invoke the inode operations like make directory and rename etc. And to see if there the any acl lost or changes due to that. And those changes need to be grant no more permissions that the its original ACL intend to give.

The ACL mapping algorithms that presented here tries to get the cifs acl security id get the windows cifs sids and map them to Linux inode mode bit values with the correct value for each owner group and others with its read write execution permissions.

Investigate the functionality by mount a windows share by using cifs.ko and setting the permission at the windows share file with appropriate allow and deny entries. And try to read the permissions with the getfacl utility from the Linux side.

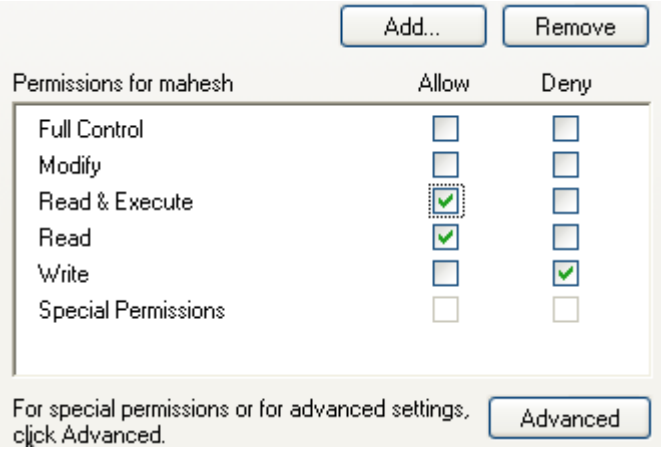

Figure 5-1 windows permission setting

User should be able to observe the getfsacl results with appropriate entries.

```
# file: test
# owner: mahesh
# group: mahesh
user::r-x
group::r--
other::r--
```

EA and ACL Performance

Since ACLs define a more sophisticated discretionary access control mechanism, they have an influence on all access decisions for file system objects. It is interesting to compare the time it takes to perform an access decision with and without ACLs.

SuSE Novell has conducted some good research on this EA and its behavious in context of ACLs .Measurements were performed on a PC running SuSE Linux 8.2, with the SuSE 2.4.20 kernel. The machine has an AMD Athlon processor clocked at 1.1 GHz and 512 MiB of RAM. The disk used was a 30 GB IBM Ultra ATA 100 hard drive with 7200 RPM, an average seek time of 9.8 ms, and 2 MiB of on-disk cache. The Ext2, Ext3, Reiserfs, and JFS file systems were created with default options on an 8 GiB partition. On XFS, to compare EAs that are stored in inodes and EAs that are stored externally, file systems with inode sizes of 256 bytes and 512 bytes were used. These file systems are labeled XFS-256 and XFS-512, respectively.

To exclude the time for loading the file's inode into the cache, a stat system call was performed before checking access. The time taken for the stat system call is not shown. The first access to the access ACL of a file may require one or more disk accesses, which are several orders of magnitude slower than accessing the cache. The actual times these disk accesses take vary widely depend on the disk speed and on the relative locations of the disk blocks that are accessed. The function used for measuring time has a resolution of 1 microsecond. In the ACL case, the file that is checked has a five-entry access ACL.

Chapter 6– Conclusion

Operating systems offer relatively different implementations in many aspects, and different access control lists are one of the problems we face while communicating with these systems. CIFS got lot more permissions other that read right execute, Like permission to delete a specific file, permission to change specific permissions, it is really complex frame work they specify. In POSIX sid we have only Allow permissions but in CIFS side we have Allow and Deny permissions. And also the two different ACL inheritance models.

It will take quite a lot of effort to analyze all these CIFS permissions and mapping them to POSIX such that the semantics are the same. In this project what we tried to archive is the analysis of the CIFS side and the POSIX side and what are the permissions can be map without any major security threats. We took a one of its mapping in this project work and implementation it on new file system driver at the Linux box (i.e. cifs.ko).

There are still other concerns that need to work, UNIX way of identifying users and groups by numeric IDs is a problem in large networks. Like the whole IEEE (2004) POSIX.1 permission model, current implementations of POSIX.1e ACLs are based on these unique IDs. Maintaining central user and group databases becomes increasingly difficult with increasing network size. In CIFS, users and groups are identified by globally unique security identifiers (SIDs). Processes have a number of SIDs, which determine their privileges. CIFS ACLs may contain SIDs from different domains.
Current implementations of POSIX ACLs only support numeric user or group identifiers within the local domain. Allowing non-local identifiers in ACLs seems possible but difficult. A consequent implementation would require substantial changes to the process model. At a minimum, in addition to non-local user and group identifiers in ACL entries, file ownership and group ownership for non-local users and groups would have to be supported.

References

[1] Applications may rightfully make about the system behaviour Gruenbacher A. (2004). *NFSv4 ACLs in POSIX*

[2] Introduced a new file system driver cifs.ko French .S (2007). Linux CIFS Client Guide PP 1-3

[3] Linux file systems based by Virtual file system Daniel .P (2002).*Understanding the Linux Kernel O'Reilly Media* pp 303-305

[4] UNIX based systems use the traditional POSIX file permission model The Institute of Electrical and Electronics Engineers (IEEE). (2004) *.posix_1003.1e-990310* pp 35-38

Web references

[WWW 01] http://www.samba.org/cifs/ [2008/12/23]

[WWW 02] http://www.snia.org/tech_activities/CIFS/ [2009/01/15]

[WWW 03] http://www.codefx.com/CIFS_Explained.htm [2009/02/23]

[WWW 04] http://en.wikipedia.org/wiki/Server_Message_Block [2009/02/10]

[WWW 05] http://ubiqx.org/cifs/Intro.html [2009/02/10]

[WWW 06] http://www.lynuxworks.com/products/posix/posix.php3 [2009/05/09]

[WWW 07] http://www.suse.de/~agruen/acl/posix/posix.html [2009/04/23]

[WWW 08] http://www.suse.de/~agruen/acl/linux-acls/online/ [2009/04/19]

[WWW 09] http://technet.microsoft.com/en-us/library/bb463216.aspx [2009/07/29]

[WWW 10] http://www.cs.unc.edu/cgi-bin/howto?howto=linux-posix-acls [2010/01/23]

[WWW 11] http://www.topbits.com/unix-file-permissions.html [2009/12/13]

[WWW 12] http://linuxmanpages.com/man5/acl.5.php [20010/01/11]

[WWW 13] http://www-uxsup.csx.cam.ac.uk/pub/doc/suse/suse9.3/suselinux-adminguide_en/sec.acls.handle.html [2010/02/13]

[WWW 14] http://www.softpanorama.org/Access_control/acl.shtml [2009/12/29]

[WWW 15] http://www.suse.de/~agruen/nfs4acl/draft-gruenbacher-nfsv4-acls-in-posix-00.html [2009/12/23]

[WWW 16] http://en.wikipedia.org/wiki/Virtual_file_system [2009/12/23]

[WWW 17] http://en.allexperts.com/e/v/vi/virtual_file_system.htm [2009/12/23]

[WWW 18] http://www.pcguide.com/ref/hdd/file/ntfs/filesAttr-c.html [2009/12/24]

Appendix

Appendix I– An overview of CIFS Architecture

Overview

The cifs.ko module is a Linux virtual file system module (French, 2007). It exports a set of entry points to the kernel. The kernel communicates with userspace via libc (the operating system runtime library). The cifs module is similar to a dynamically loadable device driver, and typically is loaded implicitly by any attempt to mount with type cifs ("mount -t cifs") although it can be loaded explicitly via modprobe or insmod. The cifs module can be unloaded via rmmod (or shutdown of the system).

Mount

When the mount utility is invoked, it searches for a mount helper with a matching name to the file system type (mount.fstype) which in our case is /sbin/mount.cifs (similarly on umount /sbin/umount.cifs would be invoked, if found, but umount.cifs is not necessary for most 'situations'). The small helper mount.cifs lightly parses the cifs specific mount options (most importantly translating the host name of the target server into an ip address) and then invokes do_mount in libc (the main operating system runtime library) which crosses into the kernel address space and into the virtual file system mapping layer of the kernel (vfs). The vfs layer of the kernel mediates between libc and a file system (such as the cifs module). The file system sets up function pointers in the key objects in order to facilitate this mapping.

Filesystem objects

File systems manage the following objects, most of which export function pointers (which cifs sets up when the object is instantiated in order to point to cifs helper functions)

1. dentries: file names and directory names in the file system namespace. A file on disk may have multiple names.
2. inodes: the metadata such as timestamps, mode and attributes which describe a file on disk pages in the page cache: the data in a file
3. file structs: the information about an open file instance, e.g. whether it was opened for read or write or both
4. superblock: the information about a unique server resource which is mounted by this client (the vfs mount includes the information regarding the local path it is mounted). A superblock may be mounted over more than one local directory.

CIFS specific objects

Most of the important CIFS structures are defined in fs/cifs/cifsglob.h which includes more detailed structure definitions. During the first mount to a server (target with a unique ip address), the cifs module creates a tcp socket and a corresponding cifs TCP Server Info structure. For each unique user name (mounts can be made more than once to the same server/share with different credentials) mounted to that server, a cifsSesInfo structure is created. For each unique share (\\servername\sharename) that is mounted, (each unique exported resource on some server that is mounted by this client) a cifsTconInfo structure is created ("Tcon " stands for the SMB/CIFS term "tree connection"). Each inode that is accessed (looked up e.g. via stat) causes a cifInodeInfo struct to be created and linked in with the inode structure, and similarly opening a file causes a cifsFileInfo struct to be created an linked in with the file system specific area of the file object.

CIFS source code

The CIFS kernel source code and header files are located in the fs/cifs directory of the kernel. The files can be categorized as follows.

1. Network protocol definition (mostly SMB/CIFS on-the-wire data formats): cifspdu.h, ntlmssp.h, rfc1002pdu.h
2. the modules internal data structure definitions, cifsglob.h
3. Worker functions which handle SMB/CIFS protocol implementation: cifssmb.c

4. Linux VFS specific mappings to the worker functions: inode.c, file.c, link.c, readdir.c
5. Mount (and new session) handling: cifsfs.c, connect.c and sess.c
6. encryption routines: cifsencrypt.c, smbencrypt.c, md5.c, md4.c, smbdes.c
7. misc helper funtions: misc.c, netmisc.c, cifs_debug.c
8. functions which manage sending SMB as tcp data (transport.c)

The cifs mount helper, mount.cifs, is located in Samba version 3 source code in source/client/mount.cifs.c

Appendix II– POSIX Definitions

For reference, we have included the definitions related to file permissions from the Definitions volume of POSIX [2] (Institute of Electrical and Electronics Engineers, "Information Technology - Portable Operating System Interface (POSIX) - Base Definitions," December 2004.) here.

File Permission Bits

> Information about a file that is used, along with other information, to determine whether a process has read, write, or execute/search permission to a file. The bits are divided into three parts: owner, group, and other. Each part is used with the corresponding file class of processes. These bits are contained in the file mode.

File Owner Class

> The property of a file indicating access permissions for a process related to the user identification of a process. A process is in the file owner class of a file, if the effective user ID of the process matches the user ID of the file.

File Group Class

> The property of a file indicating access permissions for a process related to the group identification of a process. A process is in the file group class of a file, 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. Other members of the class may be implementation-defined.

File Other Class

The property of a file indicating access permissions for a process related to the user and group identification of a process. A process is in the file other class of a file, if the process is not in the file owner class or file group class.

Additional File Access Control Mechanism

An implementation-defined mechanism that is layered upon the access control mechanisms defined in [2] (Institute of Electrical and Electronics Engineers, “Information Technology - Portable Operating System Interface (POSIX) - Base Definitions,” December 2004.), but does not grant permissions beyond those defined in [2] (Institute of Electrical and Electronics Engineers, “Information Technology - Portable Operating System Interface (POSIX) - Base Definitions,” December 2004.), although they may further restrict them.

Alternate File Access Control Mechanism

An implementation-defined mechanism that is independent of the access control mechanisms defined in [2] (Institute of Electrical and Electronics Engineers, “Information Technology - Portable Operating System Interface (POSIX) - Base Definitions,” December 2004.), and which if enabled on a file, may either restrict or extend the permissions of a given user. IEEE Std 1003.1-2001 defines when such mechanisms can be enabled and when they are disabled.

File Access Permissions

- The standard file access control mechanism uses the file permission bits, as described below.

- Implementations may provide additional or alternate file access control mechanisms, or both. An additional access control mechanism shall only further restrict the access permissions defined by the file permission bits. An alternate file access control mechanism shall:

- Specify file permission bits for the file owner class, file group class, and file other class of that file, corresponding to the access permissions.

- Be enabled only by an explicit user action, on a per-file basis, by the file owner or a user with the appropriate privilege.

- Be disabled for a file after the file permission bits are changed for that file with chmod(). The disabling of the alternate mechanism need not disable any additional mechanisms supported by an implementation.

- Whenever a process requests for file access permission for read, write, or execute/search, if access is not denied by any additional mechanism, access shall be determined as follows:

- If a process has the appropriate privilege:

    - If read, write, or directory search permission is requested, access shall be granted.

    - If execute permission is requested, access shall be granted if execute permission is granted to at least one user by the file permission bits or by an alternate access control mechanism; otherwise, access shall be denied.

- Otherwise:

    - The file permission bits of a file contain read, write, and execute/search permissions for the file owner class, file group class, and file other class.

    - Access shall be granted if an alternate access control mechanism is not enabled and the requested access permission bit is set for the class (file owner class, file group class, or file other class) to which the process belongs, or if an alternate access control mechanism is

enabled and it allows the requested access; otherwise, access shall be denied.

Appendix III– MSDN Definitions.

DACL

A discretionary access control list (DACL) identifies the trustees that are allowed or denied access to a securable object. When a process tries to access a securable object, the system checks the ACEs in the object's DACL to determine whether to grant access to it. If the object does not have a DACL, the system grants full access to everyone. If the object's DACL has no ACEs, the system denies all attempts to access the object because the DACL does not allow any access rights. The system checks the ACEs in sequence until it finds one or more ACEs that allow all the requested access rights, or until any of the requested access rights are denied.

SACL

A system access control list (SACL) enables administrators to log attempts to access a secured object. Each ACE specifies the types of access attempts by a specified trustee that cause the system to generate a record in the security event log. An ACE in a SACL can generate audit records when an access attempt fails, when it succeeds, or both. In future releases, a SACL will also be able to raise an alarm when an unauthorized user attempts to gain access to an object.

Appendix IV– Building the Linux kernel module.

cifs,ko can be built as a kernel module. With 2.6 Linux kernel there is new build module call name as kbuid, which will allows the file system to be built as a loadable module.

Assume you are at the directory where cifs source files reside.
Make – C /lib/user/linux-kerenel-header/build/ -M=`pwd'

Further details of how this build work, can be found at the Linux documentation
(I.e. Linux kernel source/Documentation/Modules.txt)
Output binary file cifs.ko can be loaded as the kernel module using the insmod or modprob utilities and check it executing ls /proc/fs to list down the all available file systems.