A Linux Implementation of Windows ACLs

William Mahoney, James Harr

College of Information Science and Technology, University of Nebraska at Omaha

Summary
The file protection methods used within the Linux operating system are based on the traditional flags associated with old-style UNIX systems. The Access Control List (ACL) permissions utilized in more recent additions of Linux are constructed on top of these original permissions, and the result is a confusing blend of the old style and new style file protection mechanisms. When permission settings are confusing to the user, incorrect permission settings are more likely; this leads to vulnerabilities in systems which can then be taken advantage of by adversaries. The various Windows operating systems use a more simplistic ACL method for file permission checks, and the authors describe an implementation of these permissions into the Linux file system.

Key words:
Linux, Access Control Lists, File Security

1. Introduction

Computer security is often described in terms of objects *O* being protected while subjects *S* attempt to access them. A simple protection mechanism was first proposed by Lampson [8] and later clarified by Graham and also Denning [5][6], and recently again by Bishop [4]. The mechanism is the Access Control Matrix (ACM), a table of *S* rows of *O* columns, where each entry [*s*][*o*] in the table describes the permissions available to subject *s* on that particular object *o*. These permissions include the ability to read the object, write to the object, execute the object, and others as necessary by the characteristics of that object.

A difficulty with the ACM model of software security is that it does not scale well in real life. A computer system might have tens of thousands of objects and as many users, requiring an extremely large $|S| \times |O|$ matrix representation. The access control matrix is good from a theoretical standpoint but cumbersome in practice. A better solution is to distribute the workload, so that each object maintains its own concept of the users that are authorized to use that object, and in what modes. This is an Access Control List (ACL); each object is associated with a list of the users for that object, and an explicit statement of what that users permissions consist of. This is, in effect, storing the ACM "a column at a time" across various objects.

Anderson characterizes Access Control as "… the traditional center of gravity of computer security. It is where security engineering meets computer science. Its function is to control which principals (persons, processes, machines, …) have access to which resources in the system – which files they can read, which programs they can execute, how they share data with other principals, and so on" [1].

ACLs have a number of advantages, including the fact that they are oriented towards data and can be maintained by the owners of that data. People thus feel in control of their own destiny. The ACL concept is also relatively simple to implement, where each file has an area set aside for the permissions on the file. The algorithm for checking these permissions (with the Linux exception we describe below) is correspondingly simple to implement and to verify. Of course these advantages come at a price, namely that the operating system must scan the ACL at each initial object access, inherently slower than accessing a table entry in an ACM. Another difficulty is the expense involved should one desire to ask what files are accessible to a certain subject *s* – the data is simply not maintained in this manner. The case of an employee leaving or being reassigned requires a search of all files should we desire to remove all of the user permissions.

Regardless, the advantage of ACLs can not be denied – it can be a very simple but secure protection mechanism if employed properly. For these reasons we wish to add ACLs to the popular Linux system. Readers familiar with recent Linux distributions will realize that Linux already has ACLs; but the implementation and the requirement for backward compatibility nullifies one advantage of ACLs, namely their simplicity. We thus describe an ongoing implementation of simpler ACLs on top of existing Linux file systems.

Section two of this paper provides the reader with the background in ACL implementations on both the Windows and Linux platforms. Note that other models, Cisco for instance [10], are not described, as they are not germane to the task we wish to accomplish. Section three

Manuscript received July 5, 2010
Manuscript revised July 20, 2010

details the methods we are using to implement the different ACL scheme in the Linux EXT3 file system. Section four presents out conclusions. Note that we have not included a section on prior work, since the majority of this work has been on the ACL scheme we are replacing.

2. An Overview of ACL Models

The two models we review here are the Windows model and the Linux model. In the case of the former, it is a very straightforward method for determining the access rights of a given subject against an object – a linear search of the ACL is used to make the case. For Linux, though, the need for backwards compatibility has a large impact on the implementation algorithm and thus the lack of simplicity when using this method.

2.1 Access Control Lists on Windows

The Windows style Access Control Lists are made up of a set of Access Control Entries (ACEs). Each ACE determines the access rights which are allowed, denied, or audited for the user listed in the entry. The Windows ACL model must be backwards compatible with older versions of the operating system and for this reason, ACLs are not strictly required on files. Older "FAT" file systems, for instance, do not support the ACL model at all. As a result, when a process tries to access a file, the system first checks whether the type of the file system, and then if necessary checks the entries in the ACL to determine whether the appropriate permissions are available. If the file has no ACL, then full access to the file is the result. This maintains the backward compatibility to older file systems. If the file system supports ACLs and a file actually has an ACL but the ACL contains no ACEs (the ACL is empty), then the system denies all attempts to access the file. In the normal case, where the ACL actually contains entries, the system checks each ACE in turn until it finds one or more ACEs that allow the requested access rights, or until any of the rights are denied.

Windows also contains two types of ACLs, a Discretionary ACL or DACL, and a System ACL or SACL. The SACLs are used primarily for auditing and logging of security related events, and are not discussed extensively here; nor are they a part of our Linux implementation at this time. For this reason when we refer to ACLs in this paper we are referring to DACLs. Similarly, Windows documentation refers to ACLs in the context of "securable objects". We will simply call these files.

Figure one shows an example of the Windows scheme. In the figure, two processes (threads) attempt to access a file. The ACL is shown in the figure and we note that user Andrew is specifically denied access, regardless of what groups he is in. In the case of Andrew, ACE one is checked and the user is immediately denied. Jane requests write access. She is checked against ACE one, which does not apply, and then against ACE two and is allowed write access because she is a member of Group A. Since access was not denied, ACE three is also checked and Jane gains read and execute permissions as well.

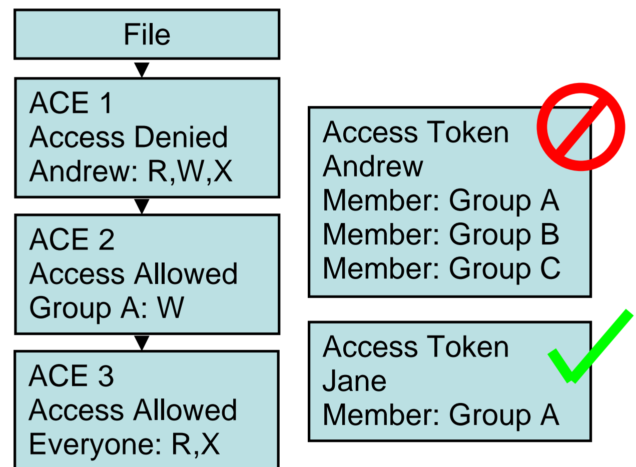

Fig. 1 Windows ACL Example

The Windows model stops checking the ACEs in the list once the required access is explicitly known. This implies that the order in which the ACEs are listed in the ACL is important; swapping numbers one and three in the above figure would allow access to Andrew, for instance.

Conceptually the Windows ACL model is elegant in its simplicity and operates the way people would assume that ACLs "should" operate.

2.2 Access Control Lists on Linux

Like the Windows Model, the Linux model must maintain a certain level of backwards compatibility so that legacy systems operate correctly. In this case the inode-based UNIX permissions are utilized, as is described in many references such as originally by Ritchie and Thompson [12] and in their patent [14]. In this scheme a file has three levels of permissions: "user", "group", and "other". Each of these, in turn contains "read", "write", and "execute" permissions which are abbreviated as "rwx". This scheme was implemented originally by reserving nine bits in the flags of the inode entry for the file, with the other bits in the flags indicating directories, character or block devices,

or other uses. A permission setting of "rwx-rw-r--" indicates all "read", "write", and "execute" permissions for the user, "read", and "write" for users in the same group, and "read" permission for anyone else. These are the permissions commonly set by utilities such as *chmod* and settings like *umask*.

Although this model is very simple to implement, difficulties arise in complex settings where different individuals must be allowed different permissions on the files. Suppose Al, Betty, and Charles all require different rights to a file, but the file is owned by Debbie. Setting the "group" and "other" permissions may not cover the requirements. Adding additional users only exacerbates the problem. System administrators have managed to find workarounds for the model's limitations, but some of these workarounds require non-obvious group setups that may not reflect organizational structures. An additional difficulty is that only the root user can create groups or change group membership. Thus, maintaining the correct permissions under these circumstances proves to be difficult. A common occurrence on today's Linux systems, for instance, is to have the user ID number and the group ID number identical, thus creating groups of one; the "other" category is thus the only setting if any importance.

The Linux process for file permission checking is summarized in the following steps. Here we will use "user", "group", and "other" (with quotation marks) to refer to the inode-based UNIX style permissions on the file, while named user and named group refer to ACL entries for the file:

1. If the user ID of the process is the "user", then use the "user" entry in the inode-based flags to determine the access.
2. Otherwise, if the user ID of the process matches one of the named user entries, this entry determines the access.
3. Otherwise, if one of the group IDs of the process matches the "group" of the file, then use the "group" entry to determine the access.
4. Otherwise, if one of the group IDs of the process matches one of the named group entries, and if an entry contains the needed permissions, then this entry determines the access.
5. Otherwise, if one of the group IDs of the process matches one of the named group entries, but none of the entries has the necessary permissions, then the access is denied.
6. Otherwise, use the "other" setting on the file.

Note in step four that there might be several groups associated with a process, and several groups listed in the ACL, so it is necessary to state the rule as above. Also, Linux ACLs contain a mask entry which is utilized within the group entries. This mask is and-ed with the permissions so that, regardless of other settings, one can turn off certain permissions by masking them with zero.

2.3 Motivation for Research

Now that we understand the basic differences between the two models, we desire to implement the Windows model onto the Linux file system. This is primarily motivated by these factors:

- We feel that the Windows ACL methods accurately reflect how users "think" about ACL control on a file.
- We feel that the Linux access control list algorithm is sufficiently complex that many system administrators, novice or experienced, may make mistakes.
- These mistakes lead to the potential for information leakage, as well as potentially allowing unauthorized persons execute permissions on files that they should not have access to.

ACLs on Linux already exist, as noted above. However we feel that they are overly complex, and also note that others also feel this way, as evidenced for example by the ACL simplifications added to the Solaris system [13] and to HP-UX [7]. Further, different versions of NFS have slight quirks when dealing with ACLs (see for example [11]). We thus turn our attention to the implementation of a simpler ACL model on the Linux system, one which follows the simpler Windows style.

3. Implementation

In order to understand the methods used for the implementation of our ACL scheme, we first introduce a general overview of how the Linux file system is organized and the layers contained within it. Referring to figure two, various applications running on a Linux system make operating system calls through the application programming interface (API). These are traps into the Kernel code for the Linux system, and the type of the request and the appropriate parameters are decoded and examined. The key data structures important for our implementation are the directory entry structures, or "dentries", and the file information node entries or "inodes". Each is cached in the file system code for improved performance.

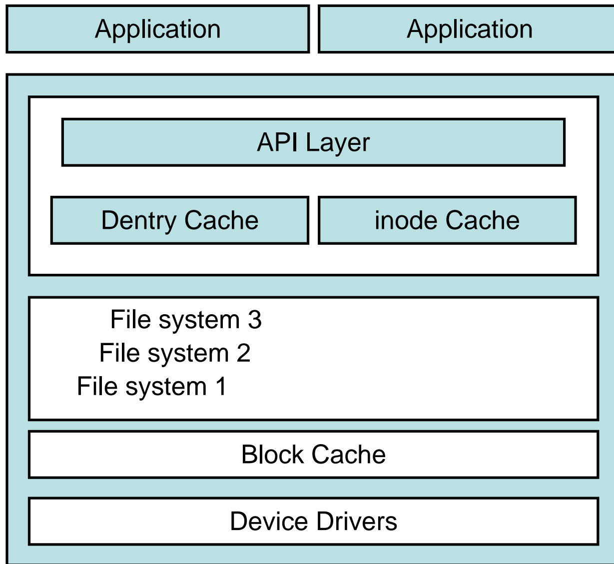

Fig. 2 File System Overall Arrangement

A key consideration when examining figure two is that the dentry and inode caches and data structures exist on top of, and independently of, the underlying file system. This gives the Linux file system a certain amount of independence over the actual file system structure below; for example, a file system for a CD-ROM may operate in a different manner than a block oriented file system on a standard hard drive, but to the upper layers and the API it appears identical. For these reasons the top level API and cache is a Virtual File System (VFS) layer, and programs actually interact with the VFS in Linux – not the individual file systems as contained on the disks.

There is a long history of documentation available on the usage and information contained in the inode. Good early references include texts by Bach [3] and also Andleigh [2]. To summarize, though, the main role of the inode is to contain the information necessary to locate the data belonging to a certain file, and also to maintain other sundry information such as timestamps. The inode is also the location where the association between a file and its owner is made, since the inode structure contains the “user” and “group” numbers for the file as described above, the protection (file mode), the size of the file, and the dates of last access, creation and last modification. When an inode is read into the cache, additional information is also maintained for the data structure, including its serial number, and a pointer to the “superblock” of the file system containing the file. There are also pointers to the dentries for this file. Further complicating the matter, there is a difference between an inode in the cache and an inode in a particular file system. Early versions of Linux had a C language union containing all variants of the inode belonging to the different file system types. New versions translate back and forth between a file system inode and a VFS inode, making the upper levels of the file system code operate independently relative to the actual inode on the disk.

In a Linux file system, the mapping from a file name to the data in the file is by associating a directory entry name with an inode number. The inode contains the data necessary for maintaining the file, while the name of the file is only used to map to the inode. It is possible (and frequently occurs) that more than one file name is associated with the same file contents.

Within the Linux system, however, the name association is by “dentry”. Specifically, a dentry contains a pointer to the associated inode in the inode cache, a pointer to the parent dentry in the dentry cache, the name component in the file name, and other information such as a pointer to the meta data for the file system. At the level of the dentry, operations exist to manipulate each of these structures. The operations include dentry comparisons, deleting and/or releasing a cache entry, and associating the dentry with the inode. The name component in the dentry is the portion of the file name that is contained within this dentry and is a part of a potentially larger qualified filename.

This overall scheme is depicted in figure three, where we show the relationship for various data structures involved in the file “/home/jharr/file”, including the linked list of dentry structures in the dentry cache, each of which is pointing at the appropriate inodes in the inode cache.

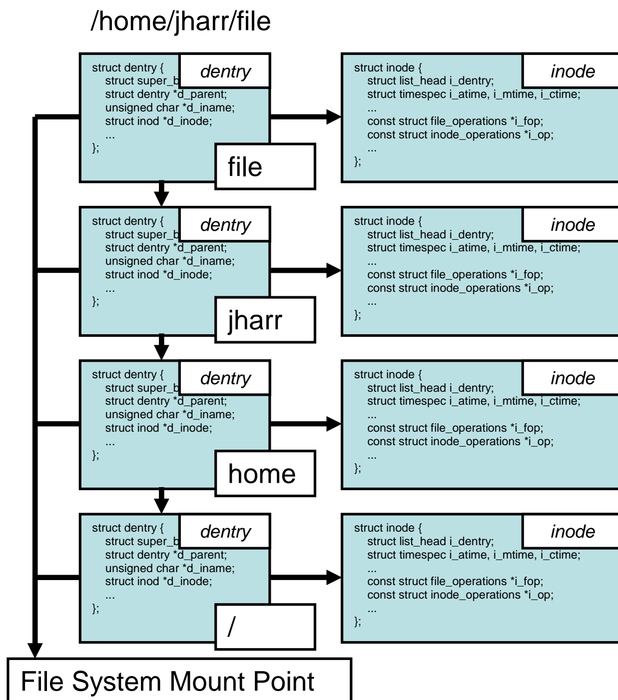

Fig. 3 Relationship Between Dentry and Inode

In figure three, inodes exist for each of the components of the file; the inode for "/" controls the root directory, "home" is a directory, and so on. The flags in the inode determine what type of entity it is, whether it is a directory, a FIFO, a device, or one of several other types of entities. The lack of any of these flags being set indicates an ordinary file, and we handle access control lists on these.

We next consider one specific file system within the Linux world: EXT3. This is the third version of the extended file system in the Linux community and has been in place since 2002. Many if not most Linux distributions utilize EXT3 as the default file system, although the user does have the option of selecting an alternative. It is thus a good candidate for our ACL research and we assume that an EXT3 file system is available for our implementation.

Each inode in a Linux EXT3 file system contains an area available for additional information; this extra data area is the "extended attribute" area of the inode entry as it is on disk. When an EXT3 file system is available and an inode needs to be cached for the VFS level, the extended attributes are copied into the cache as well as the other inode data. Note that any file system can implement extended attributes; all that is necessary is to implement "get", "set", "list", etc. for the extended attributes. It is necessary to implement these in each file system, of course, but it is much simpler at the higher levels because the "where" and "how" is already managed, excepting any decisions regarding how you want to translate in-memory structures to on-disk structures. The VFS puts no requirements on what extended attributes a FS has to support. The file system just returns the appropriate error if the needed attribute is not handled. The extended attribute area is what we take advantage of and use to hold our ACL data.

Extended attributes need space on the media, of course. In the best case scenario, the attributes are stored in the inode and no extra space is used. In the worst case, EXT4 for example, another block is specifically set aside for extended attributes. So the requirement is approximately one block (typically 4k) minus some overhead for the common case. The ACL entry structures are eight bytes, so we have the capability for approximately 500 of them on a single inode. POSIX ACLs have the same restriction.

The ACL entries overlay a structure containing the user ID and permissions necessary for the Windows-style ACLs. These include information similar to that which is detailed in figure one above and includes a flag indicating whether this entry is for permissions that are allowed or denied, the user ID in question, and the permissions pertaining to this entry.

Next we alter certain operating system calls within the VFS layer; fore example, "may_open", "may_delete", "may_creat" in "namei.c", but all functions at the VFS layer that involve access to the file need to be modified. Each of these is altered to call a new function as part of the permission processing. This function then scans the ACL data in the extended attributes and determines the permissions pertinent to the file. Scanning the ACL entries during the traversal of a filename is necessary, and this code is also scattered in much of the "namei.c" file, particularly functions such as "do_path_lookup".

This functionality is recursive. Referring back to figure three, suppose that Jane wishes to write the file /home/jharr/file and that the ACL appropriate for this file is the same as we saw previously in figure one. Then, just like before, the access should be allowed because Jane is in Group A. However, our ACL implementation also checks the parent directory ACL for /home/jharr, and then checks /home, finally stopping after checking the file system root. Since the dentry tree is a view of the filesystem with path information, this is accomplished with a simple while loop that retrieves ACL entries and merges them together to form an effective ACL for that dentry.

One remaining issue is Linux file system links. A link in a Linux file system is a case where more than one file name refers to the same file contents. Recall that a file name is simply a mapping from the string that we see over to the inode number on the file system, and that the inode contains the permissions and the data as to where the file contents are stored. In our system the inode on the disk also contains the ACL entries. In UNIX terminology there are two types of links: hard links, and soft links. The latter is of little consequence to our scheme; in a soft link, the system follows the path for the file, and when the inode is finally reached it indicates a new path to the file which is actually to be used. For instance, /home/bmahoney/data could be a soft link to /home/jharr/file. A request to open the former follows all of the usual permission checks (including ours), and upon the realization that this is a soft link, the system then follows all of the permission checks to /home/jharr/file. Incidentally, note that this could also be a soft link to yet another file, and that there is no need whatsoever for any of these to be on the same file system. Windows users will recognize this as akin to a "shortcut" on the Windows file system.

In contrast, a hard link is an issue for our ACL scheme and must be dealt with. A hard link is a simple concept: two or mode file names refer to the same inode on the disk. For instance, imagine that /home/bmahoney/data and /home/jharr/file are both referring to inode 356. In this case the inode, which contains a "link count", would have a count of two. Hard links are quite common in UNIX file systems because this is how directories link back to their parents. The ".." entry in a directory is a hard link to the inode of the parent, and this link is set up when the directory is created.

The difficulty for hard links in our ACL scheme is that I do not need permissions on a file in order to create a link to the file. For instance I can create a link /home/bmahoney/link to the /etc/shadow file, containing the encrypted Linux passwords, even though I (hopefully) do not have read access to the file. Now, opening /home/bmahoney/link will have us consider the ACL entries on this path when we should be considering the ACL entries on /etc/shadow. Our tentative solution for this problem is that we intercept the link system call and disallow the link if the user issuing the call is not the owner of the original file. We are currently testing to determine whether this solution solves all cases and/or causes any lack of functionality in existing packages.

4. Conclusions

The Windows style Access Control Lists are much simpler conceptually than the Linux/POSIX model, and turn out to be relatively easy to add to a stock Linux distribution. Our implementation currently requires the use of the EXT3 file system so that we have a convenient place to store the ACL data, but this is not a severe restriction as most systems already use EXT3; any file system with extended attributes should be a candidate for our ACL scheme.

Currently we have verified that the overall approach operates as we have planned, and we are in the process of removing inefficient debugging software so as to make a production Linux system for further testing. Once this is accomplished we will turn our attention to determining what, if anything, functions incorrectly due to the modifications in our hard link requirements. Subsequent to this we will determine the appropriate steps to bring in outside testers and what procedures are best for releasing our changes to the public domain. Ultimately the success of the research will be simple to measure at that point – will the software be used?

Acknowledgments

This research is partially funded by Department of Defense (DoD)/Air Force Office of Scientific Research (AFOSR), NSF Award Number FA9550-07-1-0499, under the title "High Assurance Software".

References

[1] Anderson, Ross, "Security Engineering: A Guide to Building Dependable Distributed Systems", 2ed, Wiley, 2008.
[2] Andleigh, Prabhat, "UNIX System Architecture", Prentice Hall, 1990.
[3] Bach, Maurice, "The Design of the UNIX Operating System", Prentice Hall, 1986.
[4] Bishop, Matt, "Computer Security – Art and Science", Pearson Education, 2003.
[5] Denning, Peter, "Third Generation Computer Systems", ACM Computing Surveys, Vol. 3 No. 4, pp 175-216, December 1971.
[6] Graham, G. Scott, and Denning, Peter, "Protection – Principles and Practice", Proceedings of the Spring Joint Computer Conference, Atlantic City, 1972, pp 417-430.
[7] HP-UX, "ACL – Introduction to Access Control Lists" Reference Vol. 5, http://docs.hp.com/en/B2355-90684/acl.5.html
[8] Lampson, Butler, "Protection", Operating Systems Review, Vol. 8 No. 1, pp 18-24, Jan 1974.
[9] Microsoft, http://msdn.microsoft.com/en-us/library/aa446683(VS.85).aspx
[10] Morrissey, Peter, "Demystifying Cisco Access Control Lists", Network Computing, http://www.networkcomputing.com/907/907ws1.html
[11] Red Hat bug report 454072, "cp and chmod don't respect NFSv4 ACLs", https://bugzilla.redhat.com/

[12] Ritchie, D. M., Thompson, K., "The UNIX Time-Sharing System", Bell System Technical Journal 57 no. 6, part 2 (July-August 1978) and available via http://cm.bell-labs.com/cm/cs/who/dmr/cacm.html
[13] Sun, "Solaris ZFS Administration Guide", http://dlc.sun.com/pdf/819-5461/819-5461.pdf
[14] U.S. Patent 4,135,240 available at http://www.google.com/patents/about?id=HuA4AAAAEBAJ&dq=4135240

William R. Mahoney is an Assistant Professor and Graduate Faculty at the University of Nebraska at Omaha Peter Kiewit Institute, and is the Director of the Nebraska University Center for Information Assurance (NUCIA). He has been actively developing information assurance curricula in order to enhance the degree program in the technological areas. He is a recipient of the College of IS&T 2008 Alumni Outstanding Teaching Award, and has assisted the Omaha Public Schools "A+ Excellence in Education" program by providing electronics demonstrations during the summer at Omaha elementary schools. Prior to the Kiewit Institute Dr. Mahoney worked for 20+ years in the computer design industry, specifically in the areas of embedded computing and real-time operating systems. During this time he was also on the part time faculty of the University of Nebraska at Omaha.

James Harr received the B.S. degree in Computer Science from the University of Nebraska at Omaha, where he is also currently a graduate student. In addition Mr. Harr is the University of Nebraska at Omaha Network Engineer.