Native eCryptfs Access Control List on Linux

Chenyang Liu, Yu Zhang, Yangting Zhang
Stony Brook University

Abstract

For most file systems such as EXT3 and EXT4, ACL(Access Control List) is used to control the accessibility of a file to specific users. ACL allows us to set a list of access controls, each of which can be a combination of user id, group id, process id, session id and time-of-day. Only the users that meet with these requirements have access to the file. Also, for invalid users, these files should not even be visible to them. However in eCryptfs, there is no such access authorization mechanism. In this report, we describe an approach to support ACL for eCryptfs and hide inaccessible files from user.

1 Introduction

For security, it is very import for a file system to support access validation mechanisms. In specific situations, we should only allow authorized users to access a file. In some file systems, e.g. EXT3, EXT4, this is implemented through ACL. Only users that passed the ACL permission checking are able to read, write or execute the file. eCryptfs uses encryption and decryption to enhance security, but leaves ACL out. ACL is necessary for eCryptfs because it allows us to set detailed access controls other than the traditional 9-bits permission mode. Also, it is reasonable that invalid users should not even know the existence of the file. This is supported through the file name hiding mechanism.

The rest of this report is organized as follows. Section 2 gives an overview of eCryptfs and ACL. Section 3 introduces how we designed to implement ACL for eCryptfs and hide inaccessible files from invalid users. The detailed implementation is described in Section 4. The testing and evaluation is presented in Section 5. Conclusion appears in Section 6. In the end, we discussed the future work in Section 7.

2 Background

Access Control List (ACL) An access control list (ACL), is a list of permissions attached to an object. An ACL consists of a set of ACL entries. An ACL entry specifies the access permissions on the associated object for an individual user or a group of users as a combination of read, write and search/execute permissions. Each file system object is associated with a list of Access Control Entries (ACEs), which define the permissions of the file owner ID, the file group ID, additional users and groups, and others. An ACL entry contains an entry tag type, an optional entry tag qualifier, and a set of permissions. A valid ACL contains exactly one entry with each of the ACL_USER, ACL_GROUP_OBJ, and ACL_OTHER tag types.

A process may request read, write, or execute/search access to a file object protected by an ACL. This request would be dealt with by an access check algorithm.

eCryptfs eCryptfs is a cryptographic lesystem for Linux that stacks on top of existing filesystems. It integrates file encryption into the file system. When eCryptfs is mounted onto an other file system, we can create and access files through eCryptfs. Files are encrypted and stored persistently on the lower file system, when accessed with correct passcode, files would be decrypted and become readable through the upper file system eCryptfs(2). In eCryptfs, cryptographic metadata is stored as preamble in thehead of each file, thus encrypted files can be moved or copied between devices(3).

3 Design

System security and flexibility are among the most desirable features of a cryptographic file system. Traditional file permissions are not enough to support more complex semantics. For example, the system administrator may wish to distribute specific access privileges to users or choose a time period exclusively for private housekeeping. Finer access controls also improve security by limiting access from specified processes and Linux sessions and even control the file name visibility.

Our design goals are summarized below:

- Provide finer access control for eCryptfs and support higher level semantics.
- The extended system must be secure, i.e. it is coherent to the ACLs semantics.
- The system must be convenient for users and administrators to perform access control operations.
- The administrative and performance overhead should be small.

We reached our goals by implementing four new system components:

(1) Extended ACLs semantics

(2) ACLs Persistent Storage

(3) ACLs Access check algorithm

(4) File name hiding.

Tag	Id	Semantic
ACL_PROCESS	PID	Allow access from process with specific PID.
ACL_SESSION	SID	Allow access from session with specific SID.
ACL_TIME	Time period	Allow access in specific time periods of a day(0 to 23)

Table 1: Extended ACL

3.1 Extended POSIX ACLs Semantics

As we have seen in the background section, the POSIX.1e ACLs extended the basic file permission model by introducing a list of ACEs associated with each file system objects. Standard ACEs supported the following formats:

<tag>: USER,GROUP,MASK,OTHER

<perm>: File permission(rwx)

<id>: Id associated with the tag

We observed that the id field is associated with the tag. So, we extended the ACLs semantics by introducing three new tags: ACL_PROCESS, ACL_SESSION, ACL_TIME. The semantic of the new tags is listed in table 1.

For the time tag, we stored the time period information in the id field. Similarly, it is flexible for adding other semantics in this way.

3.2 ACLs Persistent Storage

In modern Linux, ACLs are implemented on top of extended attributes and ACLs xattrs are integrated into the inode permission checks. We added a layer of indirection for ACL between eCryptfs and underlying lower file system. Figure 1 provided an overview of the whole process of setting the ACL and the persistent storing process.

Initialization. The userland program first assembled input to compatible structure and passed it through IOCTL to kernel space. We then converted it to eCryptfs ACLs structure which is compatible to standard Posix acl structure.

ACLs to on-disk format. For the underlying file system to persistently store the ACLs, we converted the set of ACLs to the on-disk format. Unique xattr name To distinguish our extended attributes for ACLs from other extended attributes, we assigned a unique name for each ACLs attributes. The name consisted of the prefix user.ACL concatenated with the current time in seconds. We limited the length of the xattr name to 21 characters long.

Storing to disk. After we obtained the unique name and converted ACLs to on-disk format for extended attributes, we eventually called the setxattr function of the lower file system to store the ACLs to disk and associated it to lower files.

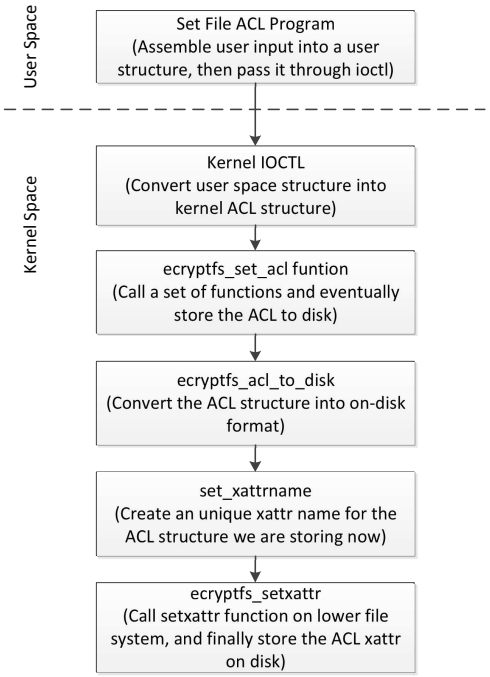

Figure 1: Set ACL to persistent storage

3.3 ACLs Access Check Algorithm

Access check was done at the eCryptfs inode permission check function. Figure 2 gives a high-level overview of the access checking process.

We first fetched the names of all extended attributes and parsed the name list to extract the name of every ACLs attributes. At this time, we fetched the real identities of the ACLs extended attributes from the lower file system. We then checked against every ACLs to grant permission.

ACL Checking. When an access check is performed, every entries are tested against the corresponding effective ID. For example, if the tag of an entry is ACL_TIME, we first get the current time and judge if it is inside the predefined time period obtained from the id field.

3.4 File Name Visibility

As discussed above, unauthorized users does not have access to the files. So it is reasonable that these invalid users should not even know the existence of the files. Therefore, when typing ls command, every user could only see the files that they have the right to access.

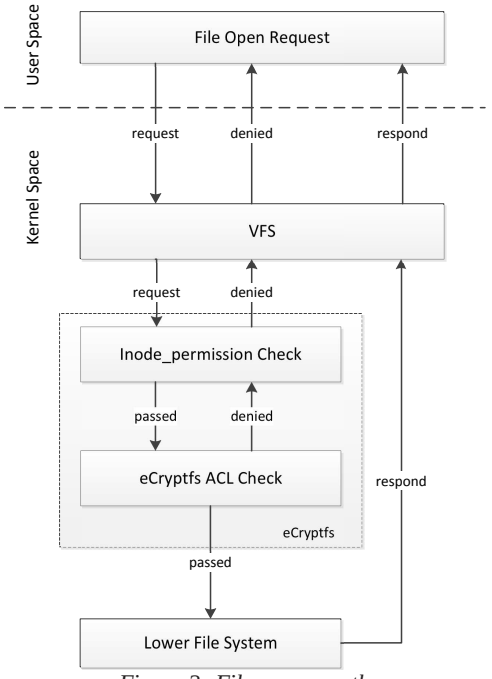

Figure 2: File access path

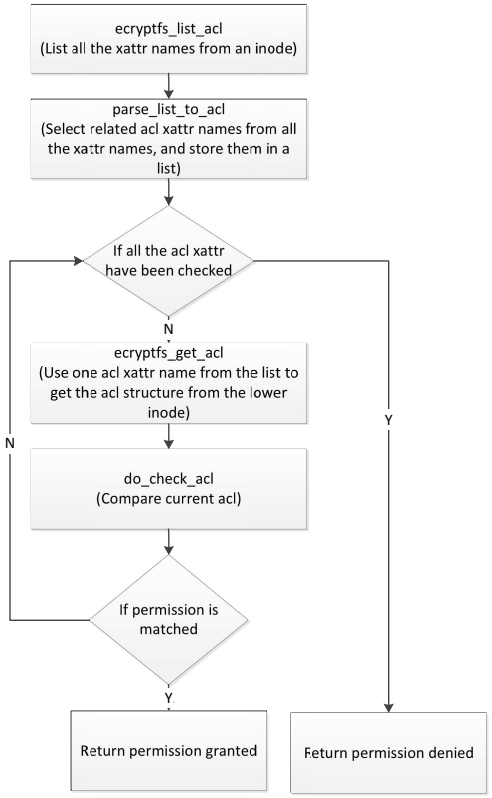

Figure 3: Check Permission Detail

4 Implementation

4.1 Kernel Data structure

Each ACL is stored in a struct posix_acl, which is defined in posix_acl.h: struct posix_acl {
union { atomic_t a_refcount;
struct rcu_head a_rcu;
};
unsigned int a_count;
struct posix_acl_entry a_entries[0];
}; An ACL contains multiple ACL entries, which are stored in the array a_entries. The number of ACL entries is recorded in a_count. For ACL entry, it is stored in the struct posix_acl_entry, which is defined as follows:

```
struct posix_acl_entry {
short e_tag;
unsigned short e_perm;
unsigned int e_id;
};
```

In this structure, e_tag can be ACL_USER, ACL_GROUP, etc. To support process_id, session_id and time interval control, we added three new tags: ACL_PROCESS, ACL_SESSION and ACL_TIME. e_id specifies the allowed id associated to e_tag. e_perm defines the permission that should be granted to authorized users.

4.2 Userland Data structure

The user space program is used to set ACL list to a particular file in eCryptfs. In user_acl.h file, there are two structures which are posix_acl_entry and posix_acl. These two structures are basically identical to the ACL structure in Linux kernel.

```
struct posix_acl_entry {
short e_tag;
unsigned short e_perm;
unsigned int e_id;
};
struct posix_acl {
unsigned int a_count;
struct posix_acl_entry a_entries[10];
};
```

Notice the user level posix_acl structure does not include the union field, because here the structure is only used to pass the value into the kernel. Also, the posix_acl structure at user level have a list of posix_acl_entry whose maximum length is 10, because normally the user is not allowed to set more than 10 ACL entries at a time.

The user level program first took the users input and stored them into the user level posix_acl and posix_acl_entry structure. After assembling these structures, we then passed the structures into kernel via ioctl function, and left the kernel side operations to complete the task.

In user program, except some regular error checking, it also checked if the current user is the owner of the file. If not, current user cannot use this program to set ACL to this file.

4.3 ACL Access Check Algorithm

Before accessing to a file, permission check should be performed. In ecryptfs_permission, we call inode_permission to check the 9-bit permission mode, then call ecryptfs_check_acl to do the ACL checking. Permission is granted to the user only if both the two checks are passed. The details of the access check algorithm is shown in Figure 3.

In ecryptfs_check_acl, a list of all the names of extended attributes attached to an inode is obtained through ecryptfs_list_acl. We eliminate useless names from the list by parse_list_to_acl, because all we need are the names that begin with user.ACL_.

Now we have got a list of all the ACL names. For each name, we fetch the ACL from extended attributes using ecryptfs_getxattr, and perform permission checking through do_check_acl. During the checking, an ACL is met only if all of its ACL entries are met. If any of the ACL is met, permission will be granted.

4.4 File Name Visibility

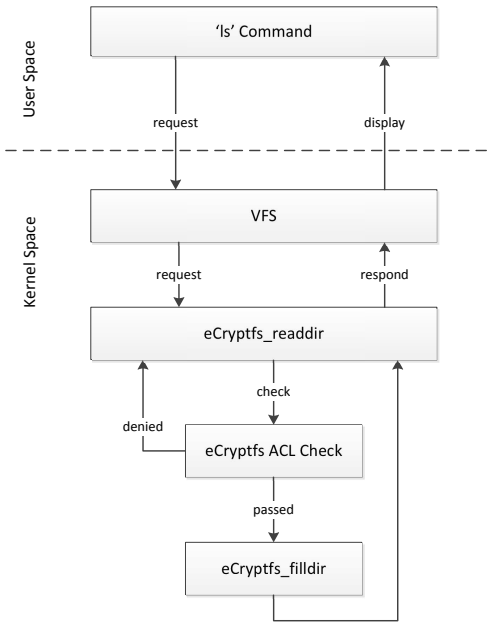

Figure 4: File name hiding

We further utilized ACLs to support filename hiding from unprivileged users. We intercepted the filldir function of eCryptfs and checked the permission there. If the user cannot pass the permission check, we left the dirent buffer unfilled. Therefore, files are only visible to users who are granted permission to access them. The function is shown in Figure 4.

5 Evaluation

We conducted different types of test to ensure we reached our goal. The usage of our userland program is as follows:

setacl -u UID -g GID -s SID -p -PID -t start end -m rwe filename

UID. We created three users as follows:
root uid=0, test1 uid=2302, test2 uid=2303.
ACL of the test file:
-u 2302 -m rw
Expected result:
User test1 can see the test file, and have read and write permission on this file. Root and test2 cannot see the test file and have no permission on this file.
Result:
Passed

GID. Three users:
root gid=0, test1 gid=100, test2 gid=2303.
ACL of the test file:
-g 100 -m rw
Expected result:
User test1 can see the test file, and have read and write permission on this file. Root and test2 cannot see the test file and have no permission on this file.
Result:
Passed

Time. ACL of the test file:
-t 10 20 -m rw
Expected result:
User will have access to the file between 10 am to 20 pm. Anytime out of this time frame, file cannot be access.
Result:
Passed

SID. ACL of the test file:
-s 2060 -m rw
Expected result:
Only user with current session ID 2060 will have access to this file, otherwise permission will be denied.
Result:
Passed.

Combinations. ACL of the test file:
-u 2302 -g 100 -s 2060 -t 10 22 -m rw
Expected result:
Only user with exactly the same values will have access to the test file. If any field fail to match this ACL, permission will be denied.
Result:
Passed.

Multiple ACLs. ACLs of the test file:
ACL 1: -u 2401 -g 100 -s 2061 -t 10 22 -m rw
ACL 2: -u 2302 -g 100 -s 2060 -t 10 22 -m rw
ACL 3: -u 0 -g 0 -s 2065 -t 22 24 -m rw
Expected result:
If a user matches any one of the ACLs listed above, permission will be granted.

Result:
Passed.

File operation overhead. In order to test the overhead introduced by ACL checking, we used time command to measure the execution time of cat <filename>. When ACL was not enabled, the time taken to cat a file was 0.004s, while the time was 0.003s if ACL was enabled. The overhead introduced by ACL is not too much.

6 Conclusions

In this project, we have implemented ACL support for eCryptfs. eCryptfs was only able to support the user authorization mechanism, which is not enough for some specific situations. ACL enables us to allow only a set of users to access a file. We also implemented the file name hiding policy, which hides inaccessible files from invalid users. All of these are implemented through wrapping the existing eCryptfs, thus no eCryptfs routine is changed. According to the tests, the above mechanisms work well without bugs.

Future Work. The ACL we have implemented in this project can be extended to support more operations, such as modify or remove an ACL entry from a file. Also, binary checksum can be implemented, that is, only binaries which have a matched (secure) checksum are allowed to execute[4].

Another possible extension is to introduce DENY semantics as NFSv4 ACLs do. This will provide even more complex semantics.

References

[1] ACL manual in Linux Kernel.

[2] Michael Austin Halcrow, eCryptfs: An Enterprise-class Encrypted Filesystem for Linux.

[3] https://launchpad.net/ecryptfs.

[4] http://www.cs.sunysb.edu/ ezk/cse506-s12/hw3.txt