

Abstract - One of the key aspects of a file system is to have an authorized access mechanism that only allows valid users to access the files. Encryption - Decryption techniques are widely used to improve these authentication checks. One such file system is eCryptfs. eCryptfs (the Enterprise Cryptographic File system) is a POSIX-compliant encrypted file system that provides advanced key management and policy features but leaves ACL support out. ACL support is extremely handy in file systems as it allows you to extend access controls to files and directories beyond the simple user/group/other ownership. In this report, we will describe an approach to implement ACL support for eCryptfs file system.

Index Terms - eCryptfs, ACL, File Systems, Name Hiding, encryption, decryption, Linux, Access Control Lists

I. INTRODUCTION

THE file system eCryptfs aims at providing advanced security mechanism for file systems using existing cryptographic technologies. It provides several policy features but does not provide Access Control List (ACL) support. Specifically, once eCryptfs has authenticated a key in the kernel's own keyring, then that key can be used by any user to decrypt files. Moreover, it encrypts only file data and not file names thus, leaving relevant information about the file openly available to all users. In this report, we discuss an approach to enhance the authentication and authorization techniques of eCryptfs so as to further restrict the access by implementing ACL support for the file system. We also incorporate the feature of File Name Hiding which restricts unauthorized users from viewing file names and other relevant information that they are not allowed to.

The rest of the report is organized as follows. Section II gives a background of eCryptfs and Access Control Lists. Section III gives an overview of the design we use to implement the ACL support mechanism for eCryptfs. The implementation of our approach is described in Section IV, followed by the evaluation in Section V. Concluding remarks appear in Section VI. Finally, we discuss the future work possibilities in Section VII.

II. BACKGROUND

Encryption is one of the most popular and reliable way to protect your data from unauthorized access. eCryptfs is one such POSIX-compliant enterprise-class stacked cryptographic filesystem for Linux. It is implemented through the FiST framework (3) for generating stacked filesystems. eCryptfs extends Cryptfs to provide advanced key management and policy features. It stores cryptographic metadata in the header of each file written, so that encrypted files can be copied between hosts; the file will be decryptable with the proper key, and there is no need to keep track of any additional

information aside from what is already in the encrypted file itself (4).

An access control list (ACL), with respect to a computer file system, is a list of permissions attached to an object. An ACL specifies which users or system processes are granted access to objects, as well as what operations are allowed on given objects. Each entry in a typical ACL specifies a subject and an operation.

A Filesystem ACL is a data structure (usually a table) containing entries that specify individual user or group rights to specific system objects such as programs, processes, or files. These entries are known as access control lists (ACLs) in the operating systems terminology. Each accessible object contains an identifier to its ACL.

The privileges or permissions determine specific access rights, such as whether a user can read from, write to, or execute an object. In some implementations an ACL can control whether or not a user, or group of users, may alter the ACL on an object (5).

III. DESIGN

1. Design Structure

As discussed above, the existing eCryptfs provides authentication and authorization techniques using encryption/decryption.

Figure 1 shows the existing eCryptfs arrangement. The User program residing in the User Land requests access to a file that is stored in the lower file system. The request is transferred via VFS to eCryptfs which then handles this request by verifying the authenticity of the user in the Basic Permission Check section. The user's credentials (password used to mount to eCryptfs) are verified with that of the file requested. Once verified, eCryptfs gets the file from the lower file system, decrypts the file and allows the user full access on that file. In other words, any user (whether he is the owner or not) can get full access to a file if he mounts the file system with the right key.

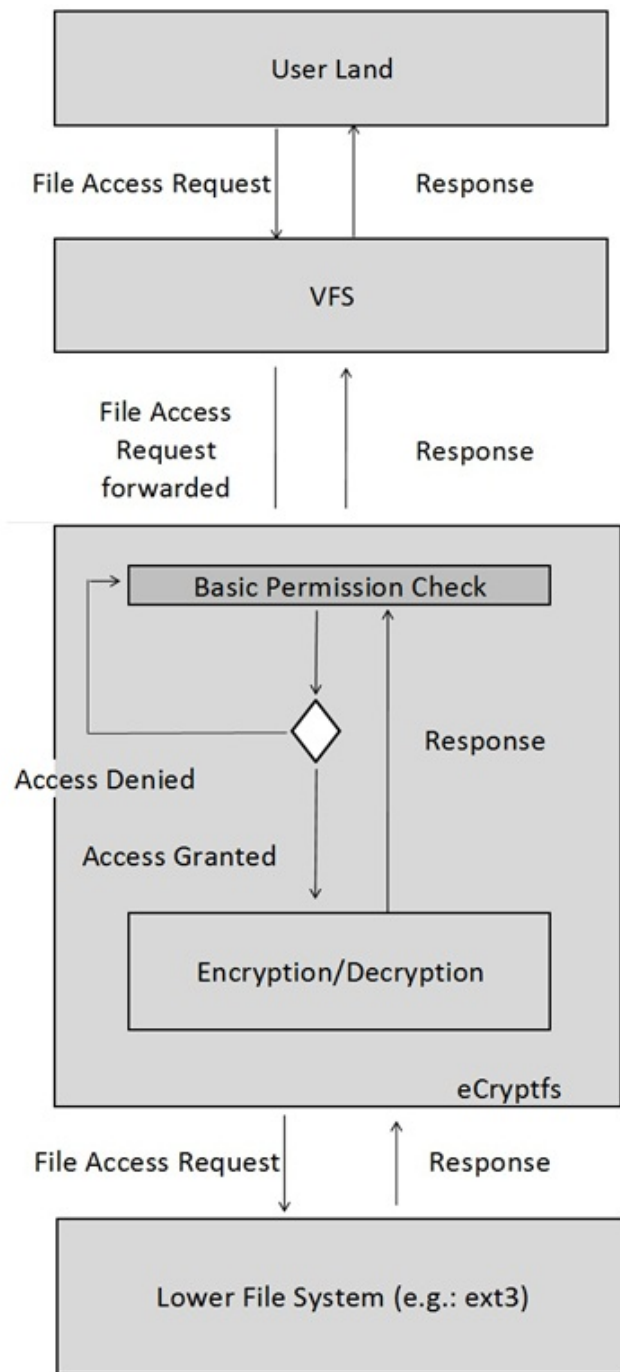


Figure 1: eCryptfs Layout

In Figure 2, we present an extension to the existing design by adding another layer of permission checks, namely Extended ACL Support. The main idea behind the extension is adding a new layer of permission checks that provides two new security features:

Support for Access Control Lists (ACLs): each of which can be a combination of any of the following:

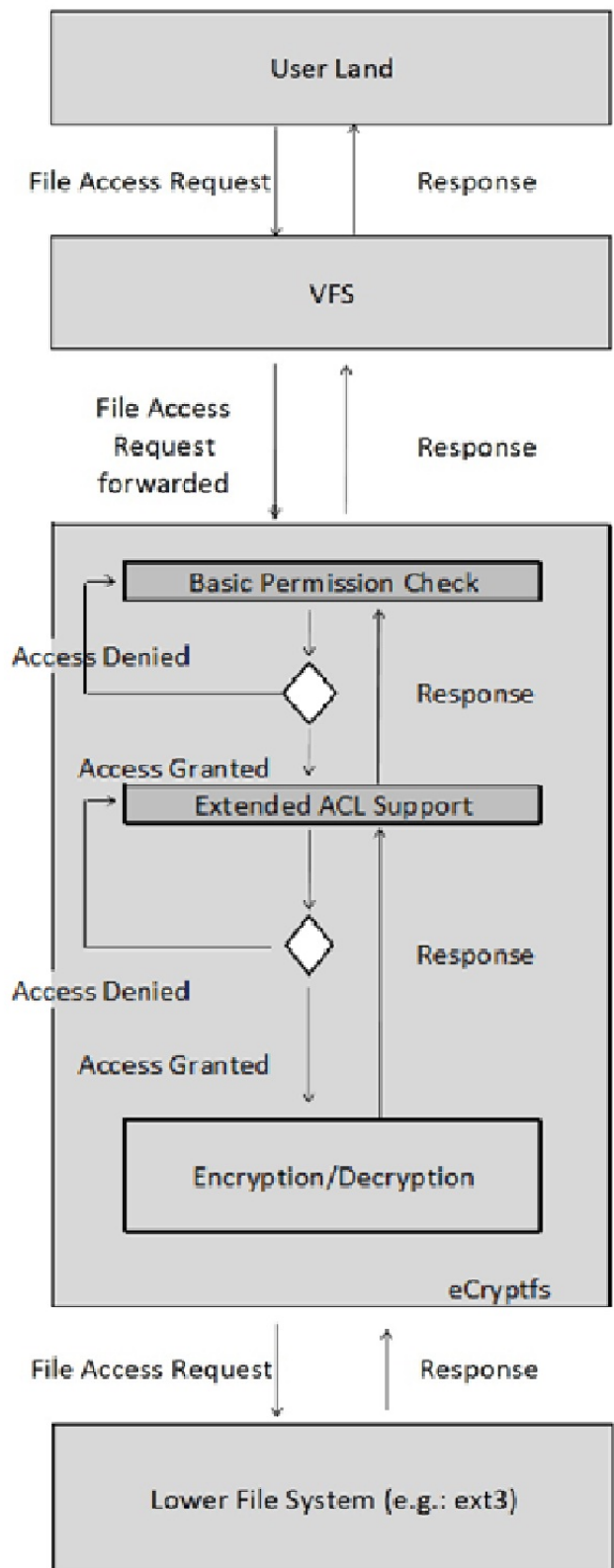


Figure 2: Extended eCryptfs Layout

UID, primary GID, PID or process name, SID (Session ID), or time-of-day.

File Name Hiding:

Hiding file names from the users that do not have permission. Unauthorized users do not have access the file; so it makes sense that they should not be able to know about the existence of the file as well. Thus, commands like "ls" do not display the results for files that the current user is not authorized to access.

2. Operations

Setting Access Control for a new file

Once the user mounts the file system with the right key*, he can set Access Controls for a file (if not already set) by using a simple user program. The user can set any combination of individual fields (UID, primary GID, PID or process name, SID (Session ID), or time-of-day) that he wants to in one Access Control.

Add new Access Controls for an existing file

Once the user mounts the file system with the right key and he further gets validated by the Extended ACL Permission Check Section, he can then add new Access Control for the file using the same user program. The new Access Control can be a combination of any atomic access control (e.g UID, primary GID, PID or process name, SID , or time-of-day) as in the above case.

3. Rules

Individual fields (UID, GID etc.) in a given ACL are a logical conjunction and ACL themselves are a disjunction. So, if you match any of the ACL, you allow. But to match an ACL, within it, all fields must match. For example, a file foobar.txt that has two Access Controls A & B can be accessed by the following only:

Let us assume,

$A : UID = x; GID = y; PID = z;$
 $B : UID = p; SID = q; PID = r; TOD = s;$

The above Access Controls are interpreted as $A \parallel B$ where,

$A = (UID = x \ \&\& \ GID = y \ \&\& \ PID = z)$
 $B = (UID = p \ \&\& \ SID = q \ \&\& \ PID = r \ \&\& \ TOD = s)$

In simpler words, any of the two Access Controls, A or B which themselves are an 'AND'ed combination of individual access control fields, is required to validate a user.

IV. IMPLEMENTATION

1. Support for Access Control Lists

We are using the ioctl to set all the ACL entries that the owner of the file wants to set for the file. We have defined the following flags for each ACL attribute:

$-U : uid$

$-P : pid$
 $-G : gid$
 $-S : sid$
 $-e : end \ time$
 $-s : start \ time$

The last two parameters specify time range within which the file can't be accessed by anybody. Both setting & checking of the ACL attributes are done at the eCryptfs layer.

To set and check the ACL at eCryptfs layer we have modified the following functions:

$eCryptfs_unlocked_ioctl$
 $eCryptfs_inode_permission$

Setting ACL attributes

$eCryptfs_unlocked_ioctl$ receives all the ACL attributes that the user wants to set for a particular file using 'acl_set' command.

User can set only one ACL at one time. We are storing the ACL's attributes and their value in $posix_acl_structure$. And we will cache this $posix_acl_structure$ in inode by using the set_cached_acl function.

We then store these ACL using $eCryptfs_setxattr$ function as the ACLs are stored at the same block along with the other extended attributes. So to distinguish the ACLs that we have set from other extended attributes , we are storing each ACL by its unique name. And the unique name that we are setting for each ACL is obtained by concatenating the string "user.ACL" and along with the time at the which that ACL was set by the user. We are using the function $do_gettimeofday$ to obtain the time stamp. This function gives the time in seconds elapsed from 1970. So this function will give the unique timestamp which we will append with the "user.ACL" string . And this concatenating string will give a unique name for each ACL. We are giving these unique name to each ACL so that when we will fetch the attributes from the disk then along with our ACLs, the other attributes like the extended attributes will also be fetched. So at the the time of checking, we will only compare the entries of the those ACLs whose names start with "user.ACL".

In a nut shell, the following three steps describe the entire for storing the ACL in cache and in the disk:

1. We are first caching the ACL in the inode using the set_cached_acl .
- 1.2 We then give a unique name to each ACL, a unique name is given by the combination of string "user.ACL" + the time given by $do_gettimeofday()$ function.
- 1.3 Store the ACL's using the $eCryptfs_setxattr$ function.

Checking ACL attributes

Now to check the ACL we will first probe the cache to get the ACL. We get the cached acl using the function

get_cached_acl. If the ACL is there in the cache, we will get it by function get_cached_acl which will return the acl if present in the cache and if acl is not cached then this function will return ACL_NOT_CACHED.

A function permission_acl() has been defined that will check the process attributes (like uid , gid , sid etc that the owner has set on the file) with each stored ACL for that file. And if all the attributes of any ACL matched with the current process attributes then that process is allowed to access that file. In case any ACL doesn't match, then we send the -EACCES error suggesting that the user is denied to access the file. This is done if the ACL is cached but if the ACL is not cached then we first call listxattr function that will give the name of all ACL attributes in the buffer with each name terminated with a null character. We then parse this buffer for each name of the ACL and fetch this ACL using function getxattr() function from the disk.

Once the ACL is fetched, we check the ACL attributes with the current process attributes using the function permission_grant() as described above.

In the permission grant function we are checking the attributes of each ACL using the current pointer that points to the current process. We match the stored ACL attribute with the current process attributes.

2. File Name Hiding

As described above, only authenticated users are allowed to access files. The aim here is to restrict the visibility of files from unauthenticated users. This is achieved by modifying the functionality of ecryptfs_readdir() function. We observed that everytime an 'ls' command is issued from user space, ecryptfs_readdir performs a read on the current directory. The ecryptfs_readdir function in turn calls ecryptfs_filldir to fill the contents (each file) of the directory into the results. When ecryptfs_filldir is called, it fetches the result from the lower filldir function. We intercept this call to the lower filldir and perform an ACL permission check on the file this function is called for. If the user passes the permission check, we allow filldir to return this result else it is blocked. Hence, only the files he is allowed to access are visible to the user [6].

The pseudo code of this functionality looks like this:

```
if this_file PASSES ACL permission check
{
    Call parent_directory->filldir()
}
else
{
    Skip filldir
}
```

3. User (Utility) Program

We have provided a user program that the user can run to set the ACL Attributes for a particular file in the following manner:

```
./acl_set -U 20 -P 32 -G 12 -S 10 -s 3 -e 5 -f file_name -D
DEFAULT
```

where,

- U specifies the attribute type UID
- 20 is the value for the UID
- P specifies PID,
- G specifies GID,
- S specifies SID,
- s & -e specify the time range,
- f 'file_name' specifies the name of the file for which we want to set the ACLs for
- D specifies the whether the file owner wants to specify the TYPE: DEFAULT or ACCESS TYPE for the ACLs.

V. TESTING & EVALUATION

1. Stress Test:

- Overview:
A file was created by root. Different users were then concurrently logged on to the system. Each user was given permission to access the file.
- Objective:
To test the robustness of the ACL permission checks. During a permission check of a single user, a significant number of memory allocations (Setting / Getting Cached ACLs) takes place. Running multiple users at the same time would increase the complexity and would be a good measure of the scalability of our code.
- Result: Passed.

2. Mixed Attributes:

- Overview:
User A has certain attributes (UID: X , GID: Y, PID: Z) different from user B (UID: A , GID: B, PID: C). In a situation where ACLs are set as a mixture of attributes of both User A and B, none of them should be able to access the file.
- Objective:
To verify the semantics of our ACL permission checking. Individual fields (UID, GID etc.) in a given ACL (for e.g. ACL1 = {UID: X , GID: B, PID: Z} and ACL2 = {UID: A, GID: Y, PID: C}) are logically ANDed (conjunction) and ACL themselves (ACL1 and ACL2) are logically ORed (disjunction).
- Result: Passed.

3. Name Hiding:

- Overview:
A particular user should be able to only see the files he/she has access to. Consider a simple statement like [ls]. The user would only be displayed the files for which he/she is the owner or has been granted access through a certain ACL.

- Objective:
To test whether a user who logs in with his key gets to see the files that he created with the same key or not. All other files that were created with the other key or that do not allow permission as per the ACLs should not be visible.

- Result: Passed

4. Unit Testing:

Independent tests were run for both ACL Permission Check and File Name Hiding:

ACL Permission Check

- Overview:
Before integration, the individual functionality of ACL Permission Check should run fine that is, the extended ACL Support should run as expected independent of the File Name Hiding feature.
- Objective:
To test the functionality without the File Name hiding feature. The extended ACLs should be supported just like described in the above sections i.e. Access Control Lists verify a user's authentication though the files created using other keys are still valid.

- Result: Passed

File Name Hiding

- Overview:
Before integration, the File Name Hiding feature should work independent of the ACL support mechanism without any issues.
- Objective:
To test File Name hiding feature without ACL support where in the regular eCryptfs functionality is working and the user that has logged in with Key A should not be able to view the file names of files created by Key B.
- Result: Passed

VI.

CONCLUSION

We have extended the existing security mechanism of eCryptfs from just a passkey validation system to a per file Access Control List support. We also added another important security feature of File Name Hiding. The entire project has been implemented in the existing eCryptfs file system and no other file systems neither lower (EXT3) nor upper (VFS) were modified thus developing a new version of the existing stackable file system which can be used on user's discretion. We have performed extensive tests (both customized and generic) on the new file system and it passed all of them cleanly.

VII.

FUTURE WORK

We have implemented a simple ACL support to the existing eCryptfs file system. The project can be further extended to enforce binary checksums that is, only binaries which have a matched (secure) checksum are allowed to execute [7].

REFERENCES

1. Naveen Kumar , Jonathan Misurda , Bruce R. Childers , Mary Lou Soffa, "FIST: A Framework for Instrumentation in Software Dynamic Translators", <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.3.6711>
2. "Linux man pages", <http://linux.die.net/man/7/ecryptfs>
3. "Access control list", http://en.wikipedia.org/wiki/Access_control_list
4. Andreas Moog, "eCryptfs - Enterprise Cryptographic Filesystem", <http://ecryptfs.sourceforge.net/ecryptfs-faq.html>
5. Vincent Danen, "Learn to use extended filesystem ACLs", http://articles.techrepublic.com.com/5100-10878_11-6091748.html
6. Erez Zadok, "Writing Stackable Filesystems", <http://www.ee.ryerson.ca/~courses/coe518/LinuxJournal/elj2003-109-stackablefilesystems.pdf>
7. CSE 506 Fall 2010 Courseware & Assignments, Stony Brook University, <http://www.cs.sunysb.edu/~ezk/cse506-f10>