eCryptfs: UID Based Authorization

Arvind Chaudhary and Bharat Singh
Stony Brook University

Abstract

eCryptfs is a POSIX-compliant **e**nterprise **crypt**ographic **f**ile **s**ystem for Linux that stacks on top of an existing file system. Currently eCryptfs prevents an attacker from accessing the file contents outside the context of trusted host environment. It does not impose any additional restrictions on who can access the files, deferring to the standard user/group/other permissions, capabilities, and so forth to regulate access to the files. However it is not easy to deny access to root user via standard access control mechanisms. This paper describes the design to support UID based access in eCryptfs file system. We propose an additional access restriction to disallow any non-authorized user.

1 Introduction

eCryptfs [5] is a POSIX-compliant enterprise-class stacked cryptographic file system for Linux. It is derived from Erez Zadok's Cryptfs [8], implemented through the FiST framework for generating stacked file systems. eCryptfs is a native Linux file system. It builds as a stand-alone kernel module for the Linux kernel; there is no need to apply any kernel patches. It is available in the mainline Linux Kernel as of 2.6.19. eCryptfs is widely used, as the basis for Ubuntu's Encrypted Home Directory, natively within Google's ChromeOS, and transparently embedded in several network attached storage (NAS) devices.

eCryptfs stores cryptographic meta data in the header of each file, so that encrypted files can be copied between hosts. The file will be decrypted only if a valid key is produced. There is no need to keep track of any additional information aside from what is already in the encrypted file itself.

eCryptfs simply requires that a File Encryption Key (FEK) be associated with any given inode in order to decrypt the contents of the file on disk. This prevents an attacker from accessing the file contents outside the context of the trusted host environment. For instance, storage media is lost or stolen. This is the only type of unauthorized access that eCryptfs is intended to prevent.

But in a multiuser environment, once a user have access to the encrypted data, there is no prevention from another user accessing the data even if that user does not have valid key. To tackle this problem there are many access control mechanisms, but it is very difficult to deny access to a root user via access control mechanisms. Thus all the users with sudo permission can also access the data that a user does not want to share. eCryptfs offers no additional access control functions other than what is already implemented via standard POSIX file permissions, access control mechanisms (capabilities, SE Linux) and so forth.

We have introduced a policy based authentication mechanism inside eCryptfs. It is an additional check to prevent unauthorized users inside a trusted host environment from accessing the encrypted data, even root user is not allowed. Based on our experimental results we see that this mechanism does not add much overhead in the current performance of eCryptfs.

The rest of this document is organized as follows. Section 2 describes the background and current limitations of eCryptfs. Section 3 describes our proposed design. Section 4 describes evaluation plan and results. Section 5 describes related work. Section 6 describes how the solution can be extended for different other methods, conclusion and future work.

2 Background

eCryptfs protects data confidentiality even when an unauthorized agent gains access to the host machine. A secret paraphrase is used to control access to the file contents. Properties like file name, size and other associated metadata are left unencrypted. Crypto information for each file is stored as crypt header following the rfc2240 [2] format along with eCryptfs marker in the file itself, so files are portable in the event of a system or disk failure. Even host and persistent storage is not trusted here.

Current Limitations.

- Any user can decrypt the file, if a valid key is produced. Once the file is decrypted, anyone on that host can read or write based on Unix permissions. Such behavior is not suitable for a multi-user or file sharing environment.

- Anyone with file permissions can delete the file. Even if the file is encrypted any user with Unix permissions can modify the file, it may corrupt the crypto headers, hence the file cannot be decrypted.

- Access revocation does not work without unmounting the eCryptfs file system, making it a disruptive operation.

- There is no support for key expiry in eCryptfs. eCryptfs can leverage the key expiration feature of Linux kernel [3].

3 Design

Threat Model. Multi user systems are prevalent in an enterprise environment. If eCryptfs is deployed, it can stop outsider attack, but it is still prone to insider attack. For e.g., a user u_1 has illegally gained access to trusted system and can now access a mounted eCryptfs directory, which was intended to be used by other user. Similarly root user can access data belonging to others. In an ideal case no user other than the file owner should be allowed access to the encrypted data.

We aim to add policy based file encryption scheme that prevents illegal access to encrypted file in case of a multi user environment. Our design goals are as follows.

- **UID based Authorization** eCryptfs should allow a set of authorized users to access and transparently encrypt or decrypt files. eCryptfs should allow administrators to add or update a user to the set of authorized users.

- **Revoke Access** eCryptfs should be able to revoke access for a user legally, but also making sure that there must be at least 1 valid user per file (do not revoke all of them).

eCryptfs before changes. The Figure 1 shows existing functionality of eCryptfs. eCryptfs is a stacked file system and has mapping to VFS objects and operations.

VFS Objects. eCryptfs maintains a reference between objects of lower file system and objects in eCryptfs file system. The reference is maintained via a set of objects like

- *private_data* of file object
- *u.generic_ip* of inode object
- *d_fsdata* of dentry object
- *s_fs_info* of superblock object

The *inode u.generic_ip* have pointer to *struct ecryptfs_crypt_stat* which contains file crypto header information. The crypto header is stored persistently along with file data on disk.

VFS Operations. At mount time, eCryptfs-utils generates an authentication token for the passphrase specified by the user. These tokens are stored in Linux kernel keyring and are used to setup crypto context for the eCryptfs files. VFS operations from all the users just try to validate the file crypt headers and the authentication token. If the key identifier in the header is matched against the mount-wide key identifier, the request is passed to the lower file system to perform actual IO. We can see here that there is no support for per user access check , access revoke and expiry in eCryptfs kernel. We have tried to address some of these limitations in our design.

```
struct ecryptfs_allowed_list {
  uid_t a_uid[4];   /*allowed UID*/
  uid_t a_gid[4];   /*allowed GID*/
  uid_t a_suid[4];  /*saved UID*/
  uid_t a_sgid[4];  /*saved GID*/
  uid_t a_euid[4];  /*effective UID*/
  uid_t a_egid[4];  /*effective GID*/
};
```

Listing 1: ecryptfs_allowed_list structure

eCryptfs after changes. eCryptfs stores the mount wide options in *struct ecryptfs_mount_crypt_stat*, see Listing 2. That is stored via private field of superblock and every encrypted file has this structure written in the crypto header. We have tried to leverage this structure to achieve our design goal.

The Figure 2 shows the functionality with our proposed changes to eCryptfs.

Check UID Module. We have added a UID based security enforcement module to eCryptfs kernel. This module is responsible to handle the per-user metadata and access enforcements as well as file access policy management activities. We added a new field in *ecryptfs_mount_crypt_stat* structure named as *ecryptfs_allowed_list*, see Listing 1. This field represents the set of authorized users who have access to files. This field is populated at mount time and is configurable via a set of IOCTL commands. First member denotes admin role, for e.g., a_uid[0], a_gid[0] are designated admin user/group and so on. Admin roles are non-revocable but they can grant/revoke access to other users/groups. As of now we are just using *a_uid* list, but other fields can also be used to enforce restriction policies.

eCryptfs Operations for UID checks. For every file operation VFS checks for inode permissions, which is passed via *ecryptfs_permission* function. In *ecryptfs_permission* we have added a check if the request is from one of the authorized users, see Listing 3. If the user is not present in the set of authorized users, then return permission denied error.

IOCTL Interface. We have added a set of ioctl commands to eCryptfs kernel, which can be used by the eCryptfs administrator to perform user policy management. Administrative role is assigned to an user with uid 1234. As of now admin role is fixed and non-revocable. At the time of eCryptfs mount, admin user is added to the list of allowed users. Admin user can use the ioctl interface to grant/revoke eCryptfs access to an user. Access to

```
struct ecryptfs_mount_crypt_stat {
  u32 flags;
  struct list_head global_auth_tok_list;
  struct mutex global_auth_tok_list_mutex;
  size_t global_default_cipher_key_size;
  size_t global_default_fn_cipher_key_bytes;
  unsigned char global_default_cipher_name[ECRYPTFS_MAX_CIPHER_NAME_SIZE + 1];
  unsigned char global_default_fn_cipher_name[ECRYPTFS_MAX_CIPHER_NAME_SIZE + 1];
  char global_default_fnek_sig[ECRYPTFS_SIG_SIZE_HEX + 1];
  struct ecryptfs_allowed_list alist;
};
```

Listing 2: ecryptfs_mount_crypt_stat structure

```
static int
ecryptfs_permission(struct inode *inode, int mask)
{
  struct ecryptfs_mount_crypt_stat *mount_crypt_stat;
  struct ecryptfs_allowed_list *alist;
  uid_t cuid;
  mount_crypt_stat = &ecryptfs_superblock_to_private(inode->i_sb)->mount_crypt_stat;
  cuid = current_uid().val;
  if (mount_crypt_stat) {
    alist = &mount_crypt_stat->alist;
    if (cuid != alist->a_uid[0] && cuid != alist->a_uid[1] &&
        cuid != alist->a_uid[2] && cuid != alist->a_uid[3]) {
      return -EPERM;
    }
  }
  return inode_permission(ecryptfs_inode_to_lower(inode), mask);
}
```

Listing 3: ecryptfs_permission function

eCryptfs ioctl commands is restricted to admin user only. It returns permission denied if the command is raised by a non-admin user.

- *ecryptfs_list_users()* used to get the current allowed users for an eCryptfs mount. This command returns the list of allowed UIDs. UID of admin user is not listed for security purposes.
  Usage: *ecryptfs_list_users mount_dir*
- *ecryptfs_allow_user()* used to add an user to the set of allowed users for an eCryptfs mount. This command takes an UID and tries to add it to the list of allowed users. If the list already have the max number of allowed users, it returns *EUSERS*. For our experiment we have set the size of allowed users list to *3*. It does takes care of already existing user in the allowed list to avoid duplicates.
  Usage: *ecryptfs_allow_user mount_dir UID*
- *ecryptfs_revoke_user()* used to revoke an user's access to an eCryptfs mount. This command takes an UID and tries to remove it from the list of allowed users. If the user exists in the allowed user list, it removes the UID from the allowed list, else just return.
  Usage: *ecryptfs_revoke_user mount_dir UID*

These IOCTL commands land into eCryptfs kernel, and update the structure *ecryptfs_allowed_list* from *mount_crypt_stat* see Listing 2. There is a possible race condition between ioctl updating *ecryptfs_allowed_list* and *ecryptfs_permisssion* reading the list to determine file access for a user. Its not a disruptive scenario, in worst case we may transiently allow an operation from a revoked user. We are planning to fix this in future using a lock or RCU. *mount_crypt_stats* is not persistent across mount/reboot as its container super block itself is not persistent. So we plan to save the structure *ecryptfs_allowed_list* to a persistent location and read it on a remount. So a flexible user access management scheme can be deployed using these IOCTL commands.

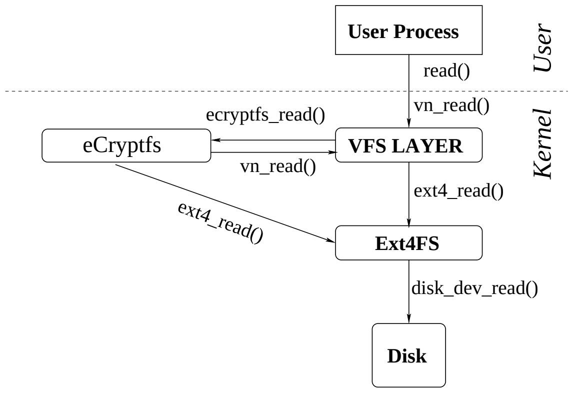

Figure 1: eCryptfs before changes

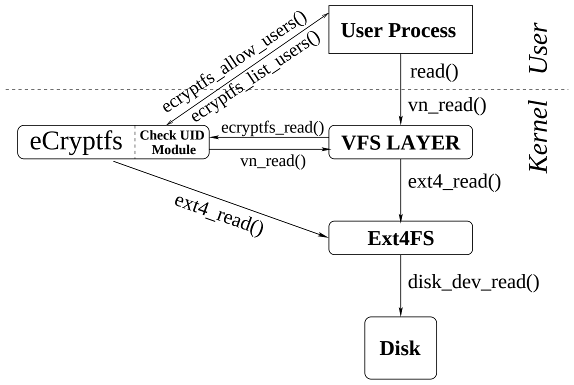

Figure 2: eCryptfs after changes

	u_1		u_2		u_3		**root**	
	with change	without changes	with changes	without changes	with changes	without changes	with changes	without changes
create file	Y	Y	N	Y	N	Y	N	Y
create dir	Y	Y	N	Y	N	Y	N	Y
remove file	Y	Y	N	Y	N	Y	N	Y
remove dir	Y	Y	N	Y	N	Y	N	Y
create symlink	Y	Y	N	Y	N	Y	N	Y
read symlink	Y	Y	N	Y	N	Y	N	Y
write symlink	Y	Y	N	Y	N	Y	N	Y
create hardlink	Y	Y	N	Y	N	Y	N	Y
write hardlink	Y	Y	N	Y	N	Y	N	Y
stat	Y	Y	N	Y	N	Y	N	Y
change dir	Y	Y	N	Y	N	Y	N	Y
read file	Y	Y	N	Y	N	Y	N	Y
write file	Y	Y	N	Y	N	Y	N	Y
create tar	Y	Y	N	Y	N	Y	N	Y
untar	Y	Y	N	Y	N	Y	N	Y
make	Y	Y	N	Y	N	Y	N	Y
rename	Y	Y	N	Y	N	Y	N	Y

Table 1: Results of different file system operations for different users, with and without the changes.

4 Evaluation

Evaluation Goals. We aimed to provide a correct working solution for eCryptfs with minimal performance overheads.

- **Correctness**
  Only the users that are allowed to access, should be able to access the file contents. Other users should get a permission denied error irrespective of their privilege level.

  The user who is assigned the administrative role should be able to add or revoke permissions to other users in the system.

  Users without administrative privilege should not be able to change file access policy or gain illegal access.

- **Performance**
  Performance of underlying file system should not suffer a penalty due to this extra security enforcement in eCryptfs.

  We keep the policy management separate from data path, so there is no overhead from management tasks on system performance.

- **Regression testing**
  This change should not break any existing functionality in eCryptfs as well as the underlying file system.

  We have tested our changes with basic file operations like create file, directory, symlink, delete, rename, untar, kernel compilation. We also have run more comprehensive tests such as *XFSTESTS* [7] and eCryptfs-utils test scripts.

Evaluation Plan. We have run a set of testing tools along with our manually written tests. The tools are *XFSTESTS* and test scripts provided in eCriptfs-utils. We have run these tools on both modified and unmodified eCryptfs kernel for different users with different privilege level. We have tried to run the maximum number of tests from these tools, in case of a test failure we rerun

	root		**user1**		**root_notallowed**	
	with changes	without changes	with changes	without changes	with changes	without changes
xfstests	56(68)	56(68)	32(68)	32(68)	0(68)	-
eCryptfs-tests	25(25)	25(25)	22(25)	22(25)	0(25)	-

Table 2: Number of passed tests and total tests for XFSTEST *and* eCryptfs-tests *for different users, with and without the changes.*

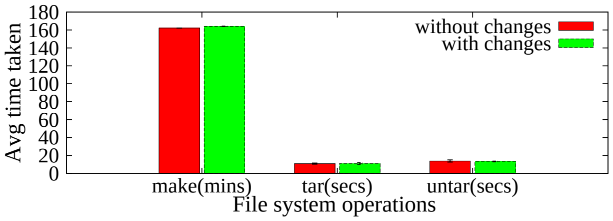

Figure 3: eCryptfs performance comparison with and without our changes for make, tar, untar operations.

it in isolation and identify the root cause of failure. Test cases from these tools are sufficient, as they test for the functionality and regression of the file system.

Experimental Setup. We have used a virtual machine with two Intel®Xeon™dual-core 2.40GHz CPUs, and 8GB RAM. The machine ran Ubuntu 14.04.1 with a vanilla 3.19.5 Linux Kernel. We chose Ubuntu because it is freely available and the package eCryptfs-utils is installed by default. For our experiments, we have 4 users u_1, u_2, u_3 and a root user. Only user u_1 is authorized to access encrypted files. User u_2 is placed in sudoer list, u_3 is a normal user. All users have intention to access the encrypted data, so we have 3 type of attackers and one authorized user.

Results. We ran basic file system operation for all four users mentioned above on the setup described. We made changes to Linux Kernel 3.19.5. Table 1 describes the results for all the file systems operations we ran. Column 1 describes the file system operation. Column 2,3,4 shows results for users u_1, u_2, u_3 and root user respectively with and without our changes in Linux kernel. **Y** indicates that the operation ran successfully. **N** indicates that the user was denied to run the corresponding commands inside the eCryptfs mount point. We also ran some performance tests, such as make a Linux Kernel, tar/untar the source tree of Linux Kernel. Figure 3 shows that there is no significant performance overhead due to our changes. The overhead for make workload is *1.17%* and for tar/untar is *1.66%*.

We ran *XFSTESTS* test-suite on Linux-3.19.5 vanilla kernel for *eCryptfs* file system. We used *FSL* git repository of *XFSTESTS* for our tests. We modified the test-suite to support *eCryptfs* file system. These tests can be categorized in two categories, 1, without changes and 2, with changes in eCryptfs kernel code. For category 1 we ran the test-suite for root user and a non-root user. For category 2 we ran the test-suite for an allowed non-root user, and root user. Then we added the root user to allowed users list and ran the test-suite again. We compared the test results for both categories for each corresponding user. We find there is no difference in the results. We see that for category 2, when root user was not allowed to access eCryptfs file system, all tests failed.

We also found new interesting bugs with eCryptfs file system.

- **noatime mount option:** If the eCryptfs file system is mounted with noatime mount option, the access times for any file should never change. But we see that there is no difference between if file system is mounted with noatime option or not. The access time for files changes in both the cases.

- **Direct IO:** eCryptfs does not support direct IO. So all the tests for direct IO failed.

- **File access timestamp Epoch:** Create a file in eCryptfs directory with creation time before epoch. Check the time of last access as seconds since Epoch, it should be a negative value. Now flush the cache by remounting the file system. Now if we check again for the time of last access as seconds since Epoch, it should still be negative, but that is not the case here.

All the *XFSTESTS* that failed can be assigned to the issues mentioned above.

We also ran the default eCryptfs-test script found in eCryptfs utils. These tests include various file systems tests, such as file read, write, concurrent access, inode races, symlinks, file truncate. These tests also include tests for already identified bugs in eCryptfs. These tests ensures that we did not break anything that was working before our changes. For both the categories, we did not find any difference in the results for all the tests. As expected for any non-allowed user all the tests in the script failed.

Table 2 shows the number of tests that passed from total number of tests for both the test-suites for various users. We examined the failed tests and identified why some of *XFSTESTS* failed. The numbers show that there was nothing broken due to our changes. Since many tests requires root permissions as these are kernel level tests, the amount of tests that failed for non-root user were considerably higher than that of root user.

IOCTL interface is restricted to admin user. For our experiments we hard coded a UID 1234 as admin user in eCryptfs kernel. We created a user with UID 1234 using adduser(1) utility. At mount time only admin user is added to the list of allowed users. Root can mount the file system but cannot access the files within the mount point, as it is not part of allowed list. At this point only admin user can perform file operation and user management activity. Once the admin added a user to the set of allowed users, that user can perform file operation. Still user management operations like list_users/allow_user/revoke_user is restricted to admin user.

We tested IOCTL interface via multiple users like admin, user1, user2, root. We observed that only admin user was able to successfully run the IOCTL commands, other users including root got permission denied error. Since admin role is non-revocable, even the admin user cannot revoke itself. Admin user cannot add any users to the allowed list once the list was full, too many users error is returned. We verified that the file operation path was being properly affected by the changes done via IOCTL interface.

5 Related Work

File based encryption are popular and have been deployed widely. Matt Blaze designed Cryptographic File System (CFS) pushes encryption services into the file system itself [1]. CFS supports secure storage at the system level through a standard Unix file system interface to encrypted files. Users associate a cryptographic key with the directories they wish to protect. Files in these directories (as well as their pathname components) are transparently encrypted and decrypted with the specified key without further user intervention; plain text is never stored on a disk or sent to a remote file server. CFS can use any available file system for its underlying storage without modification, including remote file servers such as NFS. System management functions, such as file backup, work in a normal manner and without knowledge of the key.

EncFS is a user-space stackable cryptographic filesystem similar to eCryptfs, and aims to secure data with the minimum hassle [4]. It uses FUSE to mount an encrypted directory onto another directory specified by the user. It does not use a loopback system like some other comparable systems such as TrueCrypt and dm-crypt.

Existing cryptographic file systems for Unix do not take into account that sensitive data must often be shared with other users, but still kept secret. By design, the only one who has access to the secret data is the person who encrypted it and therefore knows the encryption key or password. This paper presents a kernel driver for a new encrypted file system, called Fairly Secure File System (FSFS), which provides mechanisms for user management and access control for encrypted files [6]. The driver has been specifically designed with multi user systems in mind. FSFS also tries to prevent unintentional transfer of sensitive data to unencrypted file systems, where it would be stored in plain text.

6 Conclusions

We have introduced a UID based authentication mechanism inside eCryptfs. It is an additional check to prevent unauthorized users inside a trusted host environment from accessing the encrypted data, even root user is not allowed. Based on our results we see that this mechanism works for all type of users irrespective of their privilege level and does not add much overhead in the performance of eCryptfs. This policy based access control can be extended to other parameters like process id, tty, application.

Future Work. We plan to integrate role based access control (RBAC) with flexible number of admin and users using the system. We will extend the access policies employing restrictions based on pid, sid, process name, etc. To address the problem of non-persistent super block, we plan to add a policy file for eCryptfs to make the *ecryptfs_allowed_list* persistent across reboot/remounts. As of now Linux kernel keyring supports key expiration, but eCryptfs does not. We plan to add the key expiration support to eCryptfs.

Acknowledgments. We thank Professor Erez Zadok for guidance and rigorous design reviews. We also thank Ming Chen for helping with the *XFSTESTS*.

References

[1] M. Blaze. A cryptographic file system for unix. In *Proceedings of ACM CCS Conference (93)*, Nov 1993.

[2] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. Openpgp message format. RFC 2240, March 1998.

[3] J. Edge. Kernel key management. *Linux weekly news*, 2006.

[4] Valient Gough. Encfs: Encrypted filesystem module for linux. *Arch Linux*, 2008.

[5] Mike Halcrow. ecryptfs: a stacked cryptographic filesystem. *Linux Journal*, 2007(156):2, 2007.

[6] S. Ludwig and W. Kalfa. File system encryption with integrated user management. *ACM SIGOPS Operating Systems Review*, 35:88–93, October 2001.

[7] Xfs. http://xfs.org/index.php/Getting_the_latest_source_code#XFS_tests.

[8] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science Department, Columbia University, 1998.