eCryptfs v0.1 Design Document

Michael A. Halcrow

March 24, 2006

Contents

1 Introduction

This document details the design for eCryptfs[1]. eCryptfs is a POSIX-compliant enterprise-class stacked cryptographic filesystem for Linux. It is derived from Erez Zadok's Cryptfs, implemented through the FiST framework for generating stacked filesystems. eCryptfs 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.

[1] To obtain eCryptfs, visit `http://ecryptfs.sf.net`

eCryptfs is a native Linux filesystem. It builds as a stand-alone kernel module for the Linux kernel[2]; there is no need to apply any kernel patches.

The developers are implementing eCryptfs features on a staged basis. The first stage (version 0.1) includes mount-wide passphrase support and data confidentiality enforcement. The second stage (version 0.2) will include mount-wide public key support and data integrity enforcement. The third stage (version 0.3) will include per-file policy support. This document provides a technical description of the eCryptfs filesystem release version 0.1. eCryptfs version 0.1 is now complete, and the developers are recommending that eCryptfs be merged into the mainline Linux kernel.

Michael Halcrow has published two papers covering eCryptfs at the Ottawa Linux Symposium (2004 and 2005)[3]. These papers provide a high-level overview of eCryptfs, along with extensive discussion of various topics relating to filesystem security in Linux.

2 Threat Model

eCryptfs version 0.1 protects data confidentiality in the event that an unauthorized agent gains access to the data in a context that is outside the control of the host operating environment. A secret passphrase predicates access to the unencrypted contents of each individual file object. An agent without the passphrase secret associated with any given file (see Section 4.1) should not be able to discern any strategic information about the contents of any given encrypted file, aside from what can be deduced from the file name, the file size, or other metadata associated with the file. It should be about as difficult to attack an encrypted eCryptfs file as it is to attack a file encrypted by GnuPG (using the same cipher, key, etc.).

No intermediate state of the file on disk should be more easily attacked than the final state of the file on disk; in the event of a system error or power failure during an eCryptfs operation, no partially written content should weaken the file's confidentiality. Attackers should not be able to detect via a watermarking attack whether an eCryptfs user is storing any particular plaintext. We assume that an attacker potentially has access to every intermediate state of an encrypted file on secondary storage.

eCryptfs offers no additional access control functions other than what is already implementable via standard POSIX file permissions, Mandatory Access Control mechanisms (i.e., SE Linux), and so forth. Release 0.1 does not include integrity verification; that feature will be included in a later release.

[2]Release 0.1.5 of eCryptfs requires the Linux kernel version 2.6.16.

[3]See `http://www.linuxsymposium.org/2006/proceedings.php`. The eCryptfs paper is on page 209 of the first of the two halves of the proceedings document.

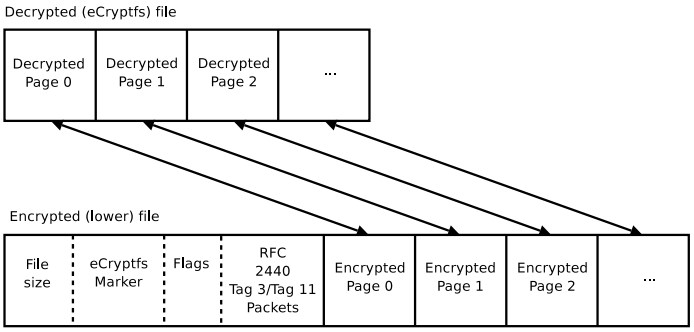

Figure 1: Relationship between an eCryptfs file and its instantiation in the lower filesystem.

3 Functional Overview

eCryptfs is a stacked filesystem that is implemented natively in the Linux kernel VFS. Since eCryptfs is stacked, it does not write directly into a block device. Instead, it mounts on top of a directory in a *lower* filesystem. Most any POSIX-compliant filesystem can act as a lower filesystem; EXT2, EXT3, and JFS are known to work with eCryptfs. Objects in the eCryptfs filesystem, including *inode*, *dentry*, and *file* objects, correlate in a one-to-one basis with the objects in the lower filesystem.

eCryptfs encrypts and decrypts the contents of the file; release 0.1 passes through other attributes of the file unencrypted, such as the file size, the file name, the access permissions, the timestamp, and the extended attributes. Directory contents are also passed through unencrypted. Figure 1 illustrates the relationship between a decrypted eCryptfs file and an encrypted file on the lower filesystem. Section 4.3 details the header contents.

eCryptfs is derived from Cryptfs[2], which is part of the FiST framework developed and maintained by Erez Zadok[3].

3.1 VFS Objects

eCryptfs maintains the reference between the objects in the eCryptfs filesystem and the objects in the lower filesystem. The references to the lower filesystem objects are maintained from eCryptfs via (1) the file object's *private_data* pointer, (2) the inode object's *u.generic_ip* pointer, (3) the dentry object's *d_fsdata* pointer, and (4) the superblock object's *s_fs_info* pointer. The pointers for the eCryptfs dentry, file, and superblock objects only reference the corresponding lower filesystem objects.

The inode *u.generic_ip* pointer references a data structure that contains state information for cryptographic operations and a reference to the lower inode object. The *ecryptfs_crypt_stat* structure is the inode cryptographic state

structure; the contents of this struct are given in Figure 2. eCryptfs fills in the *ecryptfs_crypt_stat* struct from information stored in the header region of the lower file (for existing files) or from the mount-wide policy (for newly created files).

3.2 VFS Operations

3.2.1 Mount

At mount time, a helper application generates an authentication token for the passphrase specified by the user. eCryptfs uses the keyring support in the Linux kernel to store the authentication token in the user's session keyring. A mount parameter contains the identifier for this authentication token. eCryptfs retrieves the authentication token from the session keyring using this identifier. It then uses the contents of the authentication token to set up the cryptographic context for newly created files. It also uses the contents of the authentication token to access the contents of previously created files.

3.2.2 File Open

The file format for the lower file is covered in this paper in Section 4.3.

When an existing file is opened in eCryptfs, eCryptfs opens the lower file and reads in the header. The existence of an eCryptfs marker is verified, the flags are parsed, and then the packet set is parsed.

The key identifier contained in the header is matched against the mount-wide key identifier specified at mount time. If eCryptfs cannot match the key identifier with the one specified at mount time, the open fails with a -EIO error code. eCryptfs generates a root initialization vector by taking the MD5 sum of the file encryption key; the root IV is the first N bytes of that MD5 sum, where N is the number of bytes constituting an initialization vector for the cipher being used for the file (it is worth noting that known plaintext attacks against the MD5 hash algorithm do not affect the security of eCryptfs, since eCryptfs only hashes secret values).

While processing the header information, eCryptfs modifies the *ecryptfs_crypt_stat* struct associated with the eCryptfs inode object. The modifications to the *ecryptfs_crypt_stat* structure include:

- Setting various flags, such as *ECRYPTFS_ENCRYPTED*.
- Writing the inode file encryption key.
- Writing the cipher name.
- Writing the root initialization vector.
- Filling in the array of authentication token signatures for the authentication tokens associated with the inode.
- Setting the number of header pages.

- Setting the extent[4] size.

eCryptfs later uses this information when performing VFS operations.

When a file is opened that does not yet exist, the *ecryptfs_crypt_stat* structure is initialized according to the mount-wide policy for release 0.1. This information is used to generate and write the file header prior to any further VFS operations:

- The file is encrypted.
- The cipher is AES-128.
- The root IV is the MD5 hash of the session key.
- The only authentication token associated with the file is the mount-wide passphrase specified at mount time.
- There is one header page.
- The extent size is equal to the kernel's configured page size.

Once the *ecryptfs_crypt_stat* structure is filled in, eCryptfs initializes the kernel crypto API cryptographic context for the inode. The cryptographic context is initialized in CBC mode and is used in all subsequent page reads and writes.

3.2.3 Page Read

Reads can only occur on an open file, and a file can only be opened if an applicable authentication token exists in the user's session keyring at the time that the VFS syscall that effectively opens the file takes place.

On a page read, the eCryptfs page index is interpolated into the corresponding lower page index, taking into account the header page in the file. eCryptfs derives the initialization vector for the given page index by concatenating the ASCII text representation of the page offset to the root initialization vector bytes for the inode and taking the MD5 sum of that string.

eCryptfs then reads in the encrypted page from the lower file and decrypts the page. eCryptfs first sets up the cryptographic structures to perform the decryption. It then makes the call to the kernel crypto API to perform the decryption for the page (in release 0.1, an extent is equivalent to a page). This decrypted page is what results from the VFS page read syscall.

3.2.4 Page Write

On a page write, eCryptfs performs a similar set of operations that occur on a page read (see Section 3.2.3), only the data is encrypted rather than decrypted. The lower index is interpolated, the initialization vector is derived, the page is encrypted with the file encryption key via the kernel crypto API, and the encrypted page is written out to the lower file.

[4] An extent is a contiguous region of CBC-encrypted data in the file.

3.2.5 File Truncation

When a file is either truncated to a smaller size or extended to a larger size, eCryptfs updates the filesize field (the first 8 bytes of the lower file) accordingly. When seeking past the end of the file, eCryptfs writes encrypted strings of zero's between the previous end of the file and the new end of the file.

3.2.6 File Close

In eCryptfs release 0.1, the packet set in the header never changes after the file is initially created. When a file is no longer being accessed, the kernel VFS frees its associated file, dentry, and inode objects according to the standard resource deallocation process in the VFS; eCryptfs does not perform any further cryptographic operations on the file.

4 Cryptographic Properties

4.1 Key Management

RFC 2440 (OpenPGP)[1] heavily influences the design of eCryptfs, although deviations from the RFC are necessary to support random access in a filesystem. eCryptfs stores RFC 2440-compatible packets in the header for each file. Packet types used include Tag 3 (passphrase) and Tag 11 (literal). Each file has a unique *file encryption key* associated with it; the file encryption key acts as a symmetric key to encrypt and decrypt the file contents[5]. eCryptfs generates that file encryption key via the Linux kernel *get_random_bytes()* function call at the time that a file is created. The length of the file encryption key is dependent upon the cipher being used. By default, eCryptfs selects AES-128[6].

Active eCryptfs inodes contain cryptographic contexts, with one unique context per unique inode. This context exists in a data structure that contains such things as the file encryption key, the cipher name, the root initialization vector, signatures of authentication tokens associated with the inode, various flags indicating inode cryptographic properties, pointers to crypto API structs, and so forth. The *ecryptfs_crypt_stat* struct definition is in the *ecryptfs_kernel.h* header file and is comprised of the elements in Figure 2.

The file encryption key is encrypted and stored in the first extent of the *lower* (encrypted) file. The file encryption key is encrypted with the authentication token's key that encrypts the file encryption key. Authentication token types reflect the encryption mechanism. There is one "global" *passphrase* authentication token that eCryptfs generates at mount time from the user's specified passphrase. Conversion of a passphrase into a key follows the S2K process as described in RFC 2440, in that the passphrase is concatenated with a salt; that data block is then iteratively MD5-hashed 65,536 times to generate the key that encrypts the file encryption key.

[5] Note that the *file encryption key.* is analogous to the *session key* referenced in RFC 2440

[6] Later versions of eCryptfs will allow the user to select the cipher and key length.

Name	*Type*	*Description*
lock	Mutex	Mutex for crypt stat object
root_iv	Byte Array	The root initialization vector
iv	Byte Array	The current cached initialization vector
key	Byte Array	The file encryption key
cipher	Byte Array	Kernel crypto API cipher description string
keysig	Byte Array	Signature for authentication token associated with the inode
flags	Bit vector	Status flags (encrypted, etc.)
iv_bytes	Integer	Length of IV
num_header_pages	Integer	Number of header pages for lower file
extent_size	Integer	Number of bytes in an extent
key_size_bits	Integer	Length of file encryption key in bits
tfm	Crypto API Context	Bulk data crypto context
md5_tfm	Crypto API Context	MD5 crypto context

Figure 2: Contents of cryptographic stat structure (in kernel) for eCryptfs inode

eCryptfs stores authentication tokens in the user's session keyring (a component of the Linux kernel keyring service). Helper scripts place the authentication token containing the mount-wide passphrase into the user session keyring at mount time.

When eCryptfs opens an encrypted file, it attempts to match the authentication token contained in the header of the file against the instantiated authentication token for the mount point. If the authentication token for the mount point matches the authentication token in the header of the file, then it uses that instantiated authentication token to decrypt the file encryption key that is used to encrypt and decrypt the file contents on page write and read operations.

4.2 Cryptographic Confidentiality Enforcement

eCryptfs enforces the confidentiality of the data that is outside the control of the host operating environment by encrypting the contents of the file objects containing the data. eCryptfs utilizes the Linux kernel cryptographic API to perform the encryption and decryption of the contents of its files over subregions known as *extents*.

In release 0.1, the length of each extent is fixed to the page size (typically $4,096$ bytes[7]). Since each file encrypted by eCryptfs contains a header page, the encrypted file in the lower filesystem will always be one page larger than the

[7]Release 0.1 does not support moving files between hosts with kernels of differing page

unencrypted file delivered by eCryptfs; eCryptfs transparently maps the page indices between the eCryptfs file and the lower file on read and write operations. Each extent is independently encrypted in CBC mode.

eCryptfs derives the initialization vector (IV) for each extent from a *root initialization vector* that is unique for each file. The root IV is a subset of the MD5 hash of the file encryption key for the file. The extent IV derivation process entails taking the MD5 sum of the secret root IV concatenated with the ASCII decimal characters representing the extent index.

When a *readpage()* request comes through as the result of a VFS syscall, eCryptfs will interpolate the page index to find the corresponding extent in the lower (encrypted) file. eCryptfs reads this extent in and then decrypts it; each extent is encrypted with whatever cipher that eCryptfs selected for the file at the time the file was created (in release 0.1, this defaults to the AES-128 cipher). Each extent region is independent of the other extent regions; they are not chained in any way.

When a *writepage()* request comes through as a result of a VFS syscall, eCryptfs will read the target extent from the lower file using the process described in the prior paragraph. The data on that page is modified according to the write request. The entire (modified) page is re-encrypted (again, in CBC mode) with the same IV and key that were used to originally encrypt the page; the newly encrypted page is then written out to the lower file.

Future releases will include support for integrity verification.

4.3 File Format

This release only supports a mount-wide passphrase, and so the packet set consists only of a single Tag 3 followed by a single Tag 11 packet. These packets store the encrypted file encryption key and adhere to the specification given in RFC 2440.

The first 20 bytes consist of the file size, the eCryptfs marker, and a set of status flags. From byte 20 on, only RFC 2440-compliant packets are valid.

```
Page 0:
  Octets 0-7:        Unencrypted file size
  Octets 8-15:       eCryptfs special marker
  Octets 16-19:      Flags
   Octet 16:         File format version number (between 0 and 255)
   Octets 17-18:     Reserved
   Octet 19:         Bit 1 (lsb): Reserved
                     Bit 2: Encrypted?
                     Bits 3-8: Reserved
  Octet  20:         Begin RFC 2440 authentication token packet set
Page 1:
  Extent 0 (CBC encrypted)
Page 2:
  Extent 1 (CBC encrypted)
...
```

In the RFC 2440 packet set, each Tag 3 (passphrase) packet is immediately followed by a Tag 11 (literal) packet containing the identifier for the passphrase

sizes.

in the Tag 3 packet. This identifier is formed by hashing the key that is generated from the passphrase in the String-to-Key (S2K) operation. Release 0.1 only supports one Tag 3/Tag 11 pair, which correlates with the mount-wide passphrase.

4.3.1 Marker

The eCryptfs marker for each file is formed by generating a 32-bit random number (X) and writing it immediately after the 8-byte file size at the head of the lower file. The hexadecimal value[8] $0x3c81b7f5$ is XOR'd with the random value ($Y = 0x3c81b7f5 \otimes X$), and the result is written immediately after the random number.

4.4 Deployment Considerations

eCryptfs is concerned with protecting the confidentiality of data on secondary storage that is outside the control of a trusted host environment. eCryptfs operates on the VFS layer, and so it will not encrypt data written to the swap secondary storage. It is recommended that the user employ dm-crypt[9] to encrypt the swap space on a machine where sensitive data may be loaded into memory at some point.

Selection of a passphrase should follow standard strong passphrase practices. eCryptfs ships with various helper applications in the misc/ directory; use whatever tools are convenient for you to generate a strong passphrase string. The user should store the string in a secure place and use that as the passphrase when prompted.

4.5 Cryptographic Summary

The key design components for eCryptfs realease 0.1 are:

- Header page contains plaintext file size, eCryptfs marker, version, flags, and RFC 2440 packets.
- A mount-wide passphrase is stored in the user session keyring in the form of an authentication token.
- Each file has a unique randomly-generate file encryption key. The file encryption key is encrypted and stored in the file header as a Tag 3 packet as defined by RFC 2440.
- The authentication token identifier, which is stored in the Tag 11 packet following the Tag 3 packet, is formed by taking the hash of the key that encrypts the file encryption key.

[8] This value is arbitrary.
[9] See `http://www.saout.de/misc/dm-crypt/`

  - The key that encrypts the file encryption key is generated according to the S2K mechanism described in RFC 2440.

- Page-size extents are encrypted with the default cipher (AES-128) in CBC mode.
- Each file's root initialization vector is the MD5 sum of the file encryption key for the file.
- The initialization vector for each extent is generated by concatenating the root IV and the ASCII representation of the page index and taking the MD5 sum of that string.

References

[1] J. Callas, L. Donnerhacke, H. Finney, R. Thayer, "OpenPGP Message Format," RFC 2440, Internet Engineering Task Force, Network Working Group, Nov. 1998, `http://www.ietf.org/rfc/rfc2440.txt`; accessed March 13, 2006.

[2] 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.

[3] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. To appear in USENIX Conf. Proc., June 2000.