EXT4 Encryption Design Document (public version)

Authors: Michael Halcrow <mhalcrow@google.com> (Project Lead), Uday Savagaonkar <savagaon@google.com> (Filename Encryption), Ted Ts'o <tytso@google.com> (EXT4 Maintainer), Ildar Muslukhov <muslukhovi@gmail.com> (Contributor)

For External Publication

Due to people accidentally adding comments, comments have been disabled; if you would like to make comments, send a request to tytso@google.com.

Adversarial Model

EXT4 encryption will currently focus exclusively on attacks against file content (not metadata) confidentiality under a single point-in-time permanent offline compromise of the block device

content. EXT4 encryption in its current form does not protect the confidentiality of file metadata, including the file sizes and permissions.

EXT4 encryption is not currently resilient in the face of an adversary who is able to manipulate the offline block device content prior to the authorized user later performing EXT4 file system I/O on said content. In that scenario the user can have no expectations regarding either data integrity or confidentiality.

We are not currently planning on attempting any mitigations against timing attacks. We recognize that these are important to address, but mitigations will require substantial work in the Linux kernel Crypto API. Addressing timing attacks against users of the Crypto API is out of scope for this effort.

Encryption Modes

We will encrypt pages in EXT4 inode mappings using a modular encryption interface, and so code changes for new encryption modes will be relatively straightforward as encryption standards evolve in the future. We expect this will happen as results from CAESAR (Competition for Authenticated Encryption) come to be standardized.

The cipher for file contents will be AES-256-XTS (no integrity) and the cipher for file names will be AES-256-CBC (no integrity). A discussion of the file name encryption mode will come later in this document under the section “Filename Encryption”.

Data Encryption Mode

We will generate a unique 512-bit data encryption key for each inode and will use it directly to encrypt and decrypt on a per-page basis. The IV (tweak value) for each page will be the logical page offset (page->index) within the inode mapping.

This mode is appropriate to defend data confidentiality in the event of a single point-in-time permanent offline compromise of the file system contents. Data integrity and confidentiality may be compromised if a file system is mounted and used after it has been subject to an offline attack.

Data Path Modifications

We will modify the write path with bounce pages and will encrypt to those pages. We will schedule the content of the bounce pages for writes through the block layer.

We will modify the read path with a completion callback. The completion callback will decrypt in place.

We will tolerate the exposure of zero pages in mappings.

Cryptographic Key Management

Proliferation

Efforts shall be made to limit the proliferation and longevity of secret key material in memory, both userspace and kernel. When data containers with keys are released, the contents will be wiped. This can help thwart some attacks that involve leaking kernel memory, such as through vulnerabilities in the syscall interface.

However we recognize that we lack hardware support on the majority of general purpose computing devices to effectively protect against more sophisticated attacks against memory contents of a system that is running or is hibernated.

Kernel Keyring

EXT4 will use “logon” type keys that are accessible in a keyring (see *Documentation/security/keys.txt*) for any given process performing I/O on an encrypted part of an EXT4 mount. There may be several keys available in the keyring. Each key will have a unique descriptor that is derived from the key itself. The protectors for files and directories contain descriptors for keys used to derive content encryption keys and file name encryption keys.

Protectors

For each directory and file we will write a protector into a hidden xattr in the “security” namespace. For volumes formatted with 256-byte inode sizes, we are constrained on how much data we can write to the xattr before we “spill over” the 256-byte inode limit, which has a negative impact on disk space usage and performance. We elect to include a minimal encryption key feature set for the present release in order to work within a 256-byte inode size constraint.

Our present proposal allows for a single key to protect any given directory or file.

Protector Format

```
/**
 * Protector format:
 *  1 byte: Protector format (0 = this version)
 *  1 byte: File contents encryption mode
 *  1 byte: File names encryption mode
 *  1 byte: Reserved
 *  8 bytes: Master Key descriptor
 *  16 bytes: Encryption Key derivation nonce
 */
```

Size	Name	Description
1 byte	Format	Currently 0
1 byte	Contents encryption mode	0 = AES-256-XTS
1 byte	Name encryption mode	0 = AES-256-CBC
1 byte	Reserved	Reserved
8 bytes	Master key descriptor	TRUNC(8, SHA512(SHA512(MK)))
16 bytes	Encryption key derivation nonce	Per-inode nonce chosen uniformly at random

The per-file encryption key will be derived from the Master Key and the derivation nonce by performing an AES-128-ECB encryption of the 64-byte Master Key with the nonce as the AES key.

Encryption Policy

The encryption policy instructs EXT4 how to protect new files and directories. The user sets the encryption policy after mounting by issuing an ioctl on the target top-most directory.

If an encryption context exists on the directory at the time that the user attempts to set the policy, EXT4 will validate that the master key in the policy is the same as the one in the context and will reject the policy if not.

During lookup, EXT4 will validate that the context on each subdirectory is the same as the context of the parent directory. If it isn't, then EXT4 will behave as if the encryption key is not available.

When the encryption key is not available, the directory contents will be encrypted file names. Attempts to read the files will result in an access denied error.

If an encryption context does not exist on the directory at the time that the user attempts to set the policy, EXT4 will obtain the master key that the policy describes and generate an encryption context for the directory. EXT4 will only allow encryption context creation on empty directories.

The policy contains the encryption modes and the Master Key descriptor.

```
struct ext4_encryption_policy {
  char version;
  char contents_encryption_mode;
  char filenames_encryption_mode;
```

```
  char master_key_descriptor[8];
} __attribute__((__packed__));
```

On ext4_create(), the newly created file or directory inode will inherit the immediate parent directory encryption context for the associated hard link.

If the encryption context generation operation fails -- for instance, if an encryption key isn't available -- the file open or mmap operation will return -EACCES.

Once a policy is set, one can only change it if the directory is empty or the file is 0 bytes in length.

Filename Encryption

Usage Model Requirements

Following usage-model requirements are identified.

1. Only the users that have access to the directory encryption key are allowed to create new directory entries in the directory.
2. Any user that has DAC permission to access an encrypted directory should be able to list the contents of the directory, irrespective of whether the user has access to directory encryption key.
    a. Users with access to the directory encryption key must see the filenames in clear-text form--the form in which the name was specified when the directory entry was created.
    b. The filenames displayed to users without access to the directory-encryption key must be encoded in such a way that they leak minimal information about the plaintext form of the filename.
    c. Directory listing with or without directory-encryption key must be POSIX compliant.
        i. Among other things, this implies that a displayed filename must uniquely identify a directory entry.
    d. Filenames listed with or without directory-encryption key must be legal EXT4 filenames.
        i. They must be 255 or fewer characters long.
        ii. They must not contain the '\0' and '/' characters.
3. Users who have access to the directory-encryption key must be able to perform all operations on the directory that are allowed by the DAC policy for directory.
4. Users who do not have access to the directory-encryption key are only allowed to list the contents of the directory (in encoded form), and delete files from the directory, if these operations are allowed by the DAC policy. Any other operations permitted by the DAC policy (such as creating new files in the directory) are disallowed if the user does not have access to the directory-encryption key.

Background on EXT4 Directories

An EXT4 directory consists of multiple directory entries. Table 1 shows the format of each directory entry.

Table 1: Format of an EXT4 Directory Entry

Offset	Size	Name	Description
0x0	__le32	inode	Number of the inode that this directory entry points to.
0x4	__le16	rec_len	Length of this directory entry.
0x6	__u8	name_len	Length of the file name.
0x7	__u8	file_type	File type code, one of: 0x0 Unknown. 0x1 Regular file. 0x2 Directory. 0x3 Character device file. 0x4 Block device file. 0x5 FIFO. 0x6 Socket. 0x7 Symbolic link.
0x8	char[]	name	File name. The size of this field is specified by the name_len field described above.

Each directory entry has a filename that is unique across that particular directory. A filename consists of a string of one or more ASCII characters. The number of ASCII characters in the filename is specified by the name_len field in the directory entry. Since the name_len field is one byte in size, a file name can at most be 255 bytes long. A file name is allowed to have any ASCII characters, except for the '/' and '\0' characters. Additionally, the filenames "." and ".." are reserved for the current and parent directories, and consequently, no other entries in the directory are allowed to have these names.

To enable efficient search of filenames in a directory, EXT4 supports “indexed” directories. An indexed directory (a.k.a. DX_DIR) is arranged as a “hash tree.” The hash tree is traversed using a 32-bit hash of the filename that is being looked up in the directory.

Each node in the hash tree consists of a single logical block in the directory file. The tree is two levels deep. The nodes at depth level of one are called the index nodes, while the nodes at the depth level of two are called leaf nodes.

Each leaf node contains several directory entries, each with a format as shown in Table 1.

Several leaf nodes share an index node as a parent node. The index nodes do not contain any directory entries. Instead, these nodes contain a table that allows locating a leaf node containing a particular filename, based on the 32-bit hash of the filename. Each entry in the table has format as shown in Table 2.

Table 2: Format of Table Entry in an Index Node

Offset	Type	Name	Description
0x0	__le32	hash	Starting hash value.
0x4	__le32	block	Logical block number within the directory file.

The root node of the hash tree, which is always the logical block 0 in the directory file, contains exactly two directory entries, “.” and “..”. In addition to these directory entries, it also contains a table that allows locating an index node that is parent to the node containing a given filename, based on the 32-bit hash of the filename.

Thus, given a filename, the directory entry containing that filename is looked up in a DX_DIR as follows. If the filename is “.” or “..”, then the directory entry is located in the logical block 0 of the directory file. For all other files, the lookup logic computes the 32-bit hash of the filename, and based on the lookup table located in the root node, it locates the logical block that corresponds to the index node responsible for that filename. The logic then reads the lookup table from the index block, and based on this table determines the logical block that corresponds to the leaf node that may contain the filename. It then reads the leaf-node block, and exhaustively searches that block for locating the directory entry.

It should be noted that, while DX_DIR is the preferred directory format for EXT4, the filesystem also supports non-indexed directories. A non-indexed directory is simply a directory file containing a list of directory entries. Filenames are searched in such directories exhaustively, one logical block at a time. Small directories (ones that fit in a single filesystem block) are stored as non-indexed directories.

Finally, even for indexed directories, the filesystem may fall back to exhaustive search, if it determines that the index for the directory is corrupted.

Crypto Scheme

Name Domains

In the EXT4 filesystem, names serve three purposes:

1. **Purpose 1**: File names are used by users as mnemonics to uniquely identify files in the directory.
2. **Purpose 2**: Names are stored on the disk in directory entries, and are used to locate directory entries corresponding to the files.
3. **Purpose 3**: Hashes of names are used for accessing the directory indexing structure (HTree).

Nominally, the EXT4 filesystem uses the same ASCII character string for all three purposes. However, to meet the usage requirements identified above, we split the three purposes, and use different ASCII character strings for each purpose.

For the ease of discussion, we use following terminology. The ASCII character strings used for Purpose 1 are called the user-domain filenames. These are the names used by user processes to refer to the files in the directory. The ASCII character strings used for Purpose 2 are called the disk-domain file names. These are the character strings actually stored in the directory entries on the disk. Finally, the ASCII strings used for Purpose 3 are called the HTree-domain file names.

To enable the various POSIX-compliant directory operations, we need to define how a filename is transformed/compared across these three domains. These transformations must comply with the usage requirements identified above.

High-level Design

To describe the high-level picture of the crypto scheme, we start from the user-domain filename for a user that has access to the directory-encryption key. When such a user creates a directory entry in the encrypted directory, we encrypt the user-domain filename to generate the disk-domain filename, and store the encrypted filename in the directory entry. When a user that has access to the directory-encryption key reads the directory entry, we decrypt the disk-domain filename stored in the directory entry to generate the user-domain filename, and return that to the user.

A user that does not have access to the directory-encryption key is not allowed to create new directory entries in the directory. However, such a user is allowed to read the existing directory entries from an encrypted directory. When such a user reads a directory entry, we compute a cryptographic hash of the disk-domain filename, encode the hash using a set of printable characters, and return the encoded hash to the user as the filename corresponding to the directory entry.

Finally, irrespective of whether user does or does not have access to the directory-encryption key, the filesystem must be able to compute the HTree-access hash for the particular file. This hash is used for searching for files in a directory, as well as for completing the readdir operation. Since the layout of the directory index tree is based on the HTree-access hash, it is important that a given directory entry always result in a fixed HTree-access hash, irrespective of whether the directory-encryption key is available or not. Additionally, from the point-of-view of a user that does not have access to the directory-encryption key, the encoded cryptographic hash of the filename's ciphertext acts as a filename, and such a user may perform a directory search using the encoded representation. Consequently, the filesystem must be able to compute the HTree-access hash correctly based on the encoded cryptographic hash of the filename's ciphertext.

To this end, we modify the EXT4's HTree-access hash computation as follows. Recall that EXT4 computes the HTree-access hash by passing the filename as an argument to the ext4_dirhash() function. For encrypted directories, instead of passing the unencrypted filename to the ext4_dirhash() function, we pass the encoded cryptographic hash of the filename's ciphertext, and use the final result as the HTree-access hash. This new definition of the HTree-access hash allows the filesystem to compute the HTree-access hash either from the plaintext filename (when the directory-encryption key is available), the encrypted filename (irrespective of whether the directory-encryption key is available), or the encoded cryptographic hash of the encrypted filename (irrespective of whether the directory-encryption key is available).

Figure 1 shows these transformations pictorially.

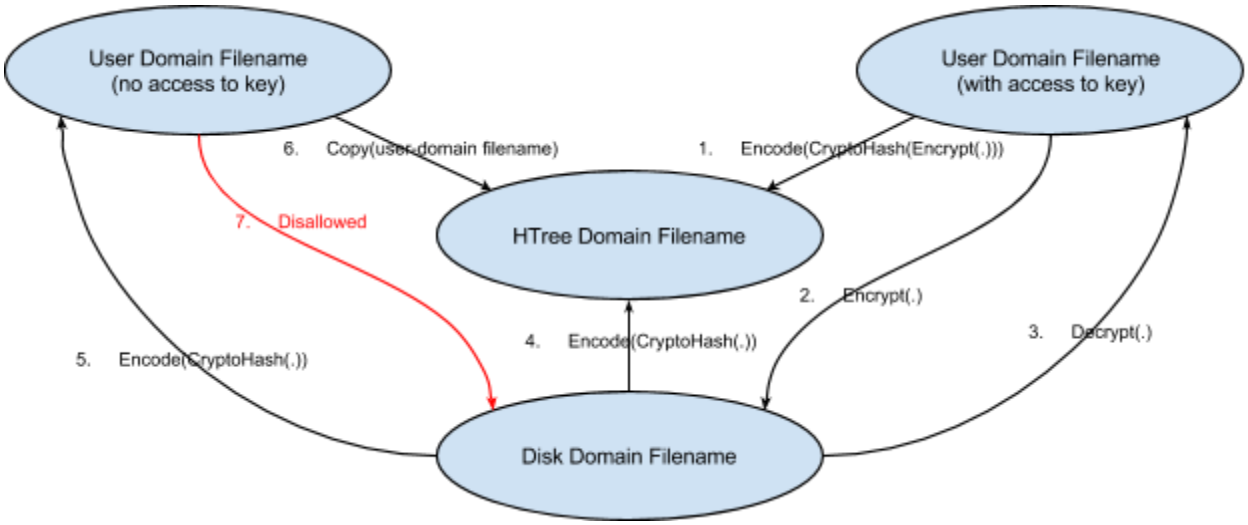

Figure 1: Filename Transformation Between Various Domains

As a part of the search operation, the filesystem is required to compare a user-domain filename against a disk-domain filename. For an encrypted directory, this comparison needs to be more than just a string comparison. More specifically, to compare a user-domain filename against a disk-domain filename, we first convert the disk-domain filename to a user-domain filename, and then perform a string match between the two.

Filename Encryption Crypto Definition

The various crypto operations are performed as a part of transforming the filenames between the various domains. As shown in Figure 1, four such operations are necessary--Encrypt, Decrypt, Crypto Hash, and Encode.

Encryption

Then encryption operation operation is used in arcs 1 and 2 in Figure 1.

Filenames are encrypted using AES-CTS mode of operation.

Decryption

The decrypt operation is used in arc 3 of Figure 1.

Decryption is reverse process of encryption. Consequently, encrypted filenames are simply decrypted using AES-CTS mode of operation.

Crypto Hash

The CryptoHash operation is used in arcs 1, 4, and 5 of Figure 1. For the CryptoHash operation, we simply use the SHA256 as the hash function.

Encoding

The encode operation is used in arcs 1, 4, and 5 of Figure 1. The encode operation is very similar to Base64 encoding. However, to make sure that an encoded name is a legal EXT4 name, the character set used for encoding is a-zA-Z0-9+=. The encoding function maps three bytes of the input string (hash to be encoded) to 4 characters in the above range.

Symlink Encryption

Symlink encryption is very similar to filename encryption. The contents of symlink are encrypted using AES-CTS mode of operation. If a user that has access to the symlink-encryption key attempts to read/dereference the symlink, the contents of the symlink are decrypted and returned to the user. On the other hand, if a user that does not have access to the symlink-encryption key attempts to read/dereference the symlink, encoded crypto hash of the encrypted symlink is returned. All the crypto algorithms used in these operations are exactly same as those used in filename encryption. However, there are three salient differences between filename encryption and symlink encryption.

1. Filenames reside in the directory file, whereas a symlink is allocated its own inode. The path to which the symlink points is stored in logical block zero belonging to the symlink's

inode. Consequently, symlink gets its own encryption key, and encryption-key protector. Symlink inherits the encryption policy from the directory in which the symlink is created. However, at a later time, if the symlink is moved to a new directory, its policy may diverge from the policy of the directory in which the symlink resides.
2. The maximum length of a symlink path is one filesystem block, instead of the 255-byte limit on filename. Since the filesystem blocks are always an integral multiple of 16 bytes, no special handling of any symlink length is required.
3. Symlinks do not have any notion of HTree, and consequently, no mappings from user-domain or disk-domain to HTree domain are required.

Test Plan

We will validate the correctness of the read and write path modifications using xfstests, fsstress, and kernel builds.

We will validate the encrypted state of the inodes by mounting without encryption enabled and verifying the ciphertext written to disk against the results using the same encryption mode and keys with OpenSSL.