Backup and restore of encrypted files HLD

Revision 0.3

January 26th, 2023

Revision History

The following is a chronological history of changes made to this document.

Revision	Date	Reason for change	Author
0.1	16 August 2022	Initial version	Sebastien Buisson
0.2	21 October 2022	Comments from Andreas Dilger integrated.	Sebastien Buisson
0.3	26 January 2023	Update security.encdata xattr format	Sebastien Buisson

Table of Contents

Introduction

Lustre's client-side encryption provides a special directory for users to safely store sensitive files. The purpose of this feature is to protect data in transit between clients and servers and protect data at rest. Client-side encryption uses the `fscrypt` library with v2 encryption policies, which allow filesystems to support transparent encryption of file and directories.

Client-side encryption encrypts all file data and file names on the clients, meaning only encrypted data will be sent over the network and stored on the target file system. Without the correct encryption key, regular files cannot be opened or truncated, yet file metadata (such as timestamps and file ownership) may be read. Directories may be listed, in which case the filenames will be listed in an encoded form derived from their ciphertext. Client-side Encryption originally landed to Lustre 2.14.0, with content only encryption support. File name encryption originally landed to Lustre 2.15.0.

Requirements Description

The need for backup and restore of encrypted files arose, just like we would do for normal files. While backup and restore at the backend device level is always an option and works independently of OST and MDT content, the situation differs with other backup/restore strategies.

The use cases we would like to support are:

- Backup and restore at the backend file system level, with OSTs and MDTs hosting Lustre encrypted files;
- Backup and restore of encrypted files at the Lustre client level.

By extension, we would like to be able to use Lustre/HSM on encrypted files, and also have some capabilities like moving Lustre encrypted files between file systems without decrypting and then re-encrypting.

The core principle we need to retain and apply to all these backup/restore methods is that we must not make any clear text copy of encrypted files. This means backup/restore must be carried out without the encryption key.

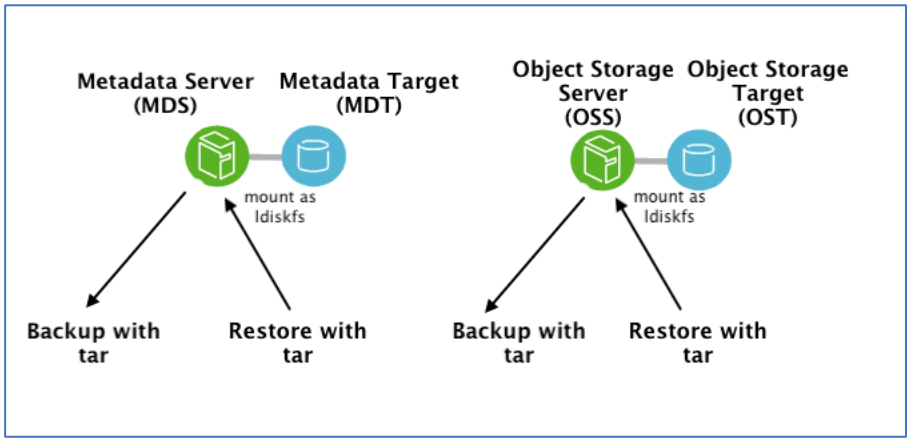
The first challenge we have to address is to get access to raw encrypted files without the encryption key. By design, `fscrypt` does not allow such kind of access, and file systems in general would not let read or write files flagged as encrypted if the encryption key is not provided. A workaround must be found for access to both raw encrypted content, and raw encrypted names.

The second challenge is to determine the clear text size of encrypted files. Without the encryption key, the apparent size corresponds to raw content. This is fine as we want to backup the whole raw content, but the clear text file size must be backed up as well in order to properly restore encrypted files later on. This information cannot be inferred by any other means.

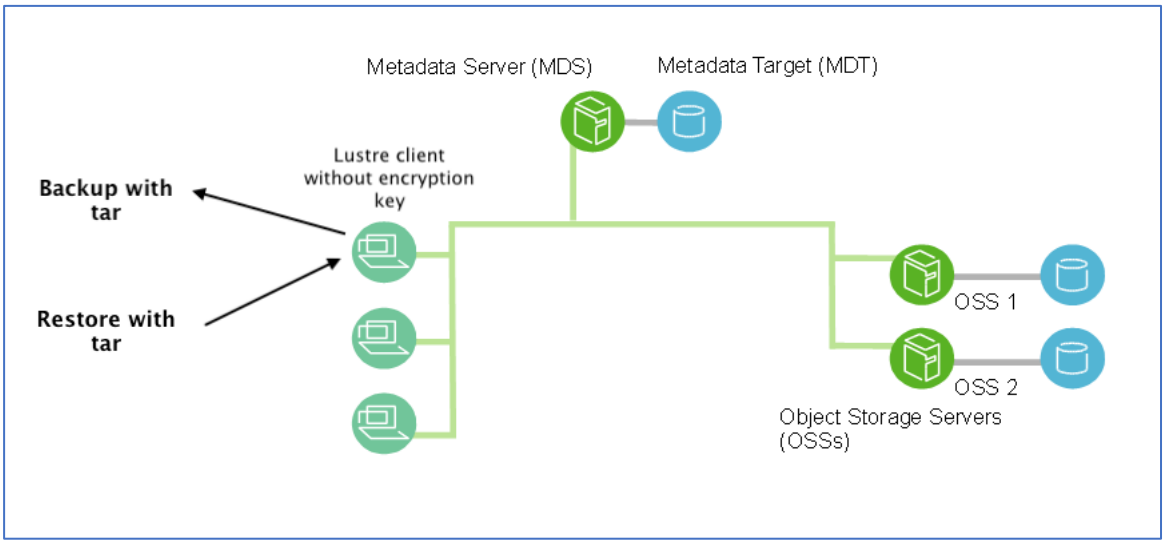
The third challenge is to get access to the encryption context of files and directories. By design, fscrypt does not expose this information, internally stored as an extended attribute

but with no associated handler. However, making a backup of the encryption context is crucial because it gives access to the per-file key used to actually encrypt file content.
It is also a non-trivial operation to restore the encryption context. Indeed, `fscrypt` imposes that an encryption context can only be set on a new file or an existing but empty directory.

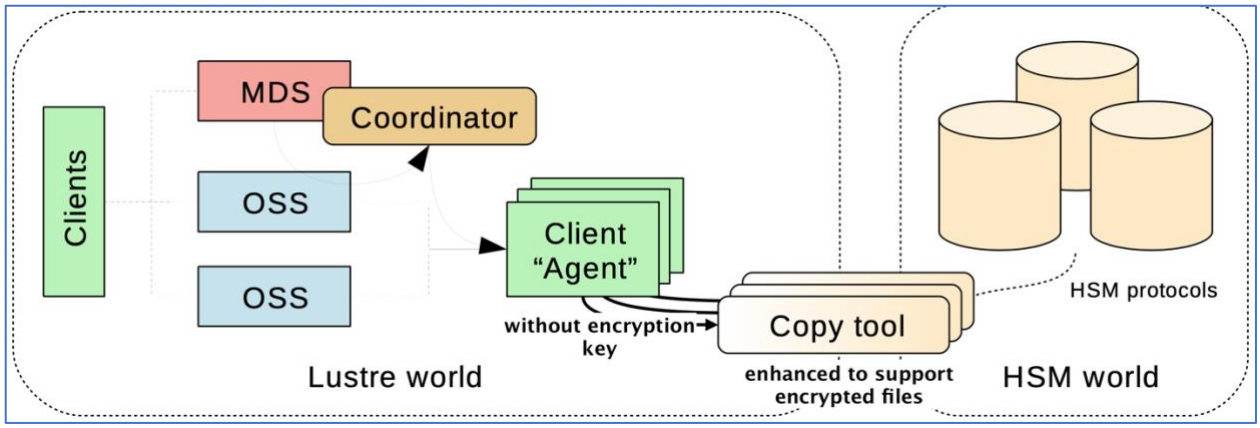
High-Level Diagrams



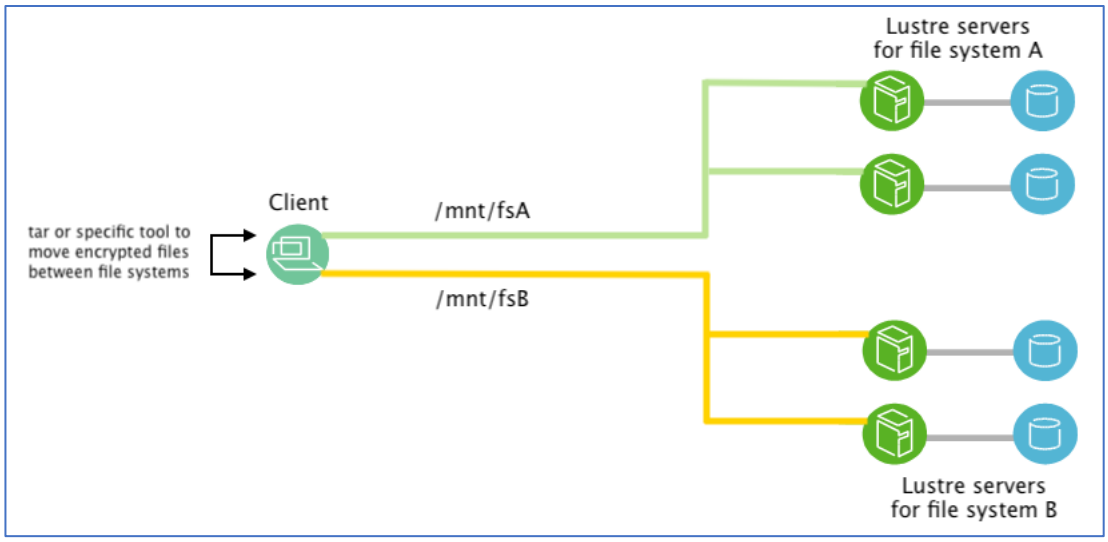
Backup/restore at backend file system level



Backup/restore at Lustre client level



Lustre/HSM on encrypted files



Encrypted files movement between file systems without decrypt/reencrypt

Data Flow

The common interface to all use cases is going to be a special extended attribute named `security.encdata`, containing:

- encoding method used for binary data.

Assume name can be up to 255 chars.

- number of bytes that the clear text file data is smaller than the reported file size.

This value is at most 4095 (set to 0 for dirs).

- encryption context

40 bytes for v2 encryption context.

- encrypted name

256 bytes max

In order to improve portability if we need to change the on-disk format in the future, the content of the `security.encdata` xattr is expressed as ASCII text with a "key: value" YAML format. As encryption context and encrypted file name are binary, they need to be encoded. So the content of the `security.encdata` xattr is:

```
{ encoding: base64url, delta: 3012, enc_ctx: YWJjZGVmZ2hpamtsbW
  5vcHFyc3R1dnd4eXphYmNkZWZnaGlqa2xtbg==, enc_name: ZmlsZXdpdGh2Z
  XJ5bG9uZ25hbWVmaWxld2l0aHZlcnlsb25nbmFtZWZpbGV3aXRodmVyeWxvbmdu
  YW1lZmlsZXdpdGh== }
```

Because base64 encoding has a 33% overhead, this gives us a string of at most 699 characters.

Note that in case an encrypted file's name is not encrypted (null mode for name encryption), the content of the `enc_name` field is the encoding of the not-encrypted name.

This extended attribute is not shown when listing xattrs, it is only expose when fetched explicitly.

Backup/restore at backend file system level

Backup phase

Lustre Operations Manual describes in section 18.3 "Backing Up an OST or MDT (Backend File System Level)" the way to proceed in the standard case. This procedure basically consists in mounting MDT or OST targets as `ldiskfs`, and then rely on the `tar` utility to create a backup. In this scenario, tar is accessing the mounted target without the encryption key.

Lustre encrypted files must have the proper encryption flag set at the `ldiskfs` level, so that it is possible for `tar` to identify them. We propose to modify the tar utility to make it "Lustre encryption aware". When detecting `ldiskfs` encrypted files, `tar` needs to explicitly fetch the `security.encdata` extended attribute, and store it along with the file. Fetching this extended attribute will internally trigger in `ldiskfs` a mechanism responsible for gathering the required information. Tar also needs to specify the `O_FILE_ENC | O_DIRECT` flags to read raw data without the encryption key.

On an MDT target, file size is not relevant, so the corresponding field in the `security.encdata` extended attribute will be set to 0. Encryption context needs to be included, as well as the raw encrypted name. The name given to the backed-up file is going to be the critical-encoding of the encrypted file name.

On an OST target, file size is a valuable information, as well as the "dummy" fscrypt encryption context set. But the encrypted name is meaningless here, so the `enc_name` field is set to the encoding of the OST object.

Restore phase

Lustre Operations Manual describes in section 18.4 "Restoring a File-Level Backup" the way to proceed in the standard case. This procedure basically consists in mounting MDT or OST targets as `ldiskfs`, and then rely on the tar utility to extract a previously created tarball to the ldiskfs file system. In this scenario, tar is accessing the mounted target without the encryption key.

When tar restores the `security.encdata` extended attribute, this will internally trigger in `ldiskfs` a mechanism responsible for extracting the required information, and setting it accordingly. `tar` also needs to specify the `O_FILE_ENC | O_DIRECT` flags to write raw data without the encryption key.

To create a valid ldiskfs file with proper encryption context and encrypted name, we can imagine a mechanism where the file with the critical-encoded name is created with the O_TMPFILE flag. That would allow setting the `security.encdata` extended attribute on this invisible file, before atomically linking it to the namespace with the correct encrypted name.

Backup/restore at Lustre client level

The method to backup and restore files from a Lustre client also leverages the `tar` utility. Similarly to the backup/restore use case at the backend file system level detailed above, we are considering the use of a "Lustre encryption aware" `tar`. In this scenario, `tar` is accessing the Lustre file system from a client without the encryption key, in order to avoid making a clear text copy of encrypted files.

Backup phase

When detecting Lustre encrypted files, `tar` needs to explicitly fetch the `security.encdata` extended attribute, and store it along with the file. Fetching this extended attribute will internally trigger in llite a mechanism responsible for gathering the required information. Tar also needs to specify the `O_FILE_ENC | O_DIRECT` flags to read raw data without the encryption key. The name of the backed-up file is the encoded+digested form returned by fscrypt.

Restore phase

The tar utility is used to extract a previously created tarball to the Lustre file system. When tar restores the `security.encdata` extended attribute, this will internally trigger in llite a mechanism responsible for extracting the required information, and setting it accordingly. Tar also needs to specify the O_FILE_ENC | O_DIRECT flags to write raw data without the encryption key.

tar will use truncate to set the correct clear text size on restored encrypted files. That will need to be an encryption-key free truncate, implemented in llite to just set the size.

To create a valid encrypted file from client side with proper encryption context and encrypted name, we can imagine a mechanism where the file with the encoded+digested name is created from client side with the O_TMPFILE flag. That would allow setting the `security.encdata` extended attribute on this invisible file, before atomically linking it to the namespace with the correct encrypted name.

Lustre/HSM on encrypted files

Lustre/HSM can work with different copy tools, depending on the nature of the HSM being interfaced. Below we consider the use of the POSIX copytool provided with Lustre, `lhsmtool_posix`, that will need to be modified to properly handle encrypted files. In this scenario, `lhsmtool_posix` is accessing the Lustre file system from a client without the encryption key, in order to avoid making a clear text copy of encrypted files.

Backup phase

When detecting Lustre encrypted files, `lhsmtool_posix` needs to explicitly fetch the `security.encdata` extended attribute, and store it along with the file in the HSM. Fetching this extended attribute will internally trigger in llite a mechanism responsible for gathering the required information. `lhsmtool_posix` also needs to specify the `O_FILE_ENC | O_DIRECT` flags to read raw data without the encryption key.

In the "normal" HSM case, archived and released files still have their inode on the MDT. The action to archive just reads raw data and writes it to HSM. This is a file content operation only. Encryption context and file name do not need to be handled, they are kept on the MDT inode.

In the Disaster Recovery HSM case, the whole files can be recreated from HSM. So this needs careful backup of encryption context and raw encrypted file name along with clear text file size and raw encrypted content.

Restore phase

When `lhsmtool_posix` restores the `security.encdata` extended attribute, this will internally trigger in llite a mechanism responsible for extracting the required information, and setting it accordingly. `lhsmtool_posix` also needs to specify the `O_FILE_ENC | O_DIRECT` flags to write raw data without the encryption key.

In the "normal" HSM case, archived and released files still have their inode on the MDT. The action to restore just fetches raw data from HSM and writes it into the Lustre file. This is a file content operation only. Encryption context and file name do not need to be handled, they are already correct on the MDT inode.

In the Disaster Recovery HSM case, the whole file is recreated from HSM. The import action creates the inode on the MDT, in released state. This needs to properly set the encryption context and raw encrypted name from the `security.encdata` extended attribute. To do so, we can imagine a mechanism where the file with the encoded+digested name is created from client side with the O_TMPFILE flag. That would allow setting the `security.encdata` extended attribute on this invisible file, before atomically linking it to the namespace with the correct encrypted name. The restore action then fetches raw data from HSM and writes it into the Lustre file. This also needs to use truncate to set the correct clear text size on the restored encrypted file. That will need to be an encryption-key free truncate, implemented in llite to just set the size.

Encrypted files movement between file systems without decrypt/encrypt

Copying or moving encrypted files between file systems is normally possible only with the encryption key. Reading on one end triggers decryption, and then writing on the other end consists in re-encrypting. The per-file encryption keys on both ends are generated independently so necessarily differ.

To copy or move encrypted files between file systems without decrypting and then re-encrypting, one possible scenario is to leverage the enhanced tar utility described in section "Backup/restore at Lustre client level", used without the encryption key. For this to work, the `tar` backup/restore must be done with the topmost encrypted directory, as all other encryption keys are inherited from there.

Another possibility would be to use a dedicated `lfs` command, see below.

Tool for development/testing

To ease development and testing, we propose to create a new `lfs` sub-command. The purpose of this new command is to manually backup and restore encrypted files, and leverage all Lustre internal mechanisms previously mentioned. The point is not having to modify existing tools like `tar` and `lhsmtool_posix`, but to create a simple command that we completely control.

```
lfs fscrypt read <path to Lustre file> -d <external dir>
lfs fscrypt write <path to backed up file> -d <dir>
```

- `read` action

Reads file at `path`, and writes into `dir`, with all xattrs plus `security.encdata` extended attribute if encrypted. Name of output file is no-key encoded name if encrypted.

- `write` action

Writes file from `path` in Lustre directory `dir`. Restores all xattrs, and internally sets encryption context, name and size from `security.encdata` extended attribute.

Security implications

Doing backup and restore of encrypted files must not compromise their security. This is the reason why we want to carry out these operations without the encryption key. It avoids making a clear text copy of encrypted files.

The `security.encdata` extended attribute contains the encryption context of the file or directory. This has a 16-byte nonce (per-file random value) that is used along with the master key to derive the per-file key thanks to a KDF function. But the master key is not stored in Lustre, so it is not backed up as part of the scenarios described above, which makes the backup of the raw encrypted files safe.

The process of restoring encrypted files must not change the encryption context associated with the files. In particular, setting an encryption context on a file must be possible only once, when the file is restored. And the newly introduced capability of restoring encrypted files must not give the ability to set an arbitrary encryption context on files.

Documentation

The new `lfs` command needs to be documented, with a dedicated man page and in the Lustre Operations Manual.

Sections 18.3 “Backing Up an OST or MDT (Backend File System Level)” and 18.4 “Restoring a File-Level Backup” of the Lustre Operations Manual must be updated to mention the capability to backup/restore encrypted files.