

Encrypted file system for Unix based operating systems

Assignment proposal

Abstract

Many users pay attention to their online security such as providing strong passwords, installing firewall and anti-virus software. However, many users also overlook the importance of offline on device security. A thief who can steal a storage device from a computer can easily connect the device and extract any information available on it. This can be even achieved without even stealing any hardware: An attacker can connect a bootable USB drive to the victim's computer, boot from it to some live operating system, mount the victim's main storage device and gain full access to it. Even without knowing the user's password to the main operating system.

To protect those kinds of attacks, a full disk encryption is required.

Full disk encryption

A full disk encryption forces any file or folder to be fully encrypted before physically saving it on the device. Naturally, any file or folder which to be read must be decrypted before. The encryption works with a symmetrical key which only the user knows.

This method of encryption makes sure that while the drive is unmounted and unused, it is fully encrypted and anyone without the key cannot read any contents. Today's processors have built in encryption instruction sets which allows on-the-fly seamless and fast encryption and decryptions of files.

Current solutions

Today, every leading operating system offers a built-in disk encryption solution:

- **Microsoft Windows** Bitlocker offers full disk encryption for professional and enterprise versions of Windows.
<https://docs.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview>
- **Apple macOS** FileVault offers full disk encryption for every modern macOS version.
<https://support.apple.com/en-us/HT208344>

- **Linux's** fscrypt is a directory based encryption integrated into ext4 file system.
<https://www.kernel.org/doc/html/v4.18/filesystems/fscrypt.html>

In addition, there are many third-party solutions available.

Exercise definition

Write a secure way of storing files and folders on a storage device.

Requirements:

- The data must be encrypted when stored in the secure area.
- When reading the data, it will be decrypted.
- When the user disconnects from the device or relocates the device to a different computer, the secure area must not be readable without the encryption key.
- Use a strong encryption algorithm. The key should be determined and known by the user.
- The data will **never** be stored unencrypted in the secure area. This means when reading the data, a decrypted copy of the data is sent to the user.

Solution

The user will allocate an empty folder on this current storage device to store the secure data (the data folder). A new folder will be mounted as a new file system which called *secfs* (The working folder). The working folder is a virtual folder mounted for the user and every system call in this folder (like create, rmdir, unlink) will call the FUSE API and execute the code in *secfs*. Every file which will be written into the *secfs* working folder will be encrypted on the fly using 128-bit AES encryption and stored in the data folder. Similarly, every read operation in the working folder

When the user first launches the application, he will be prompted create a new secure area and a secret key. This will initialize *secfs* database and will create the mount point in the system which will act as a working folder.

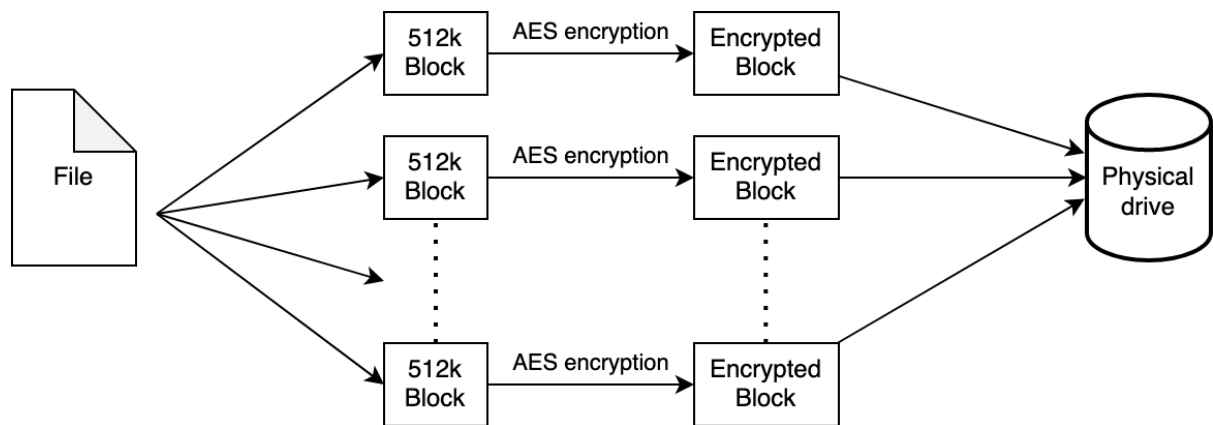
Implementation details

The heart of the software is the implementation of FUSE API which can be found in "filesystem.c" file. FUSE takes care of mounting a new folder and calling the API implementation in the software.

Blocks

Large files can take a lot of time and memory to encrypt and decrypt. In addition, FUSE can issue a write command on a large file but within a very tiny range. In order to update a large encrypted file, even one byte, will require full decryption and encryption of the file.

Therefore, every file will be split into 512k size chunks called "Blocks". Every block will be encrypted separately and stored on the physical drive:



With this method, large file can be partially updated without encryption and decrypting the whole file. Another advantage we get by using blocks is to hide the actual file sizes in the filesystem.

Database

To manage all the files and blocks the software keeps a database that maps all blocks to file names and sizes. It also creates a file and folder records. The database is stored encrypted on the physical drive along the blocks.

The database is implemented by a simple array with linear searches. I chose this implementation to speed up development.

Source files

- **Indexdb.c, blockdb.c:** Manages the database index of the file system. Operations to insert, remove and create new blocks or file entries. As well as loading the database from file and saving to file.
- **encryption.c:** Provides encryption methods. Uses openssl library.
- **Passwordinput.c:** Provides methods for secure password entry for the user.
- **Filesystem.c:** The implementation of the filesystem and the FUSE API.
- **Secfs.c:** The main struct that collects all information about the file system and some helper methods for.
- **Utilities.c** – a Collection of helper methods, macros and typedefs that are used across the whole project.

Building and running

Requirements

- Unix based OS (macOS is not supported).
- FUSE3 installed. (apt/yum install fuse3)

Building

Run “make” in the terminal inside the root folder of the source files to build the project. The output binary file called “secfs”.

Running

“./secfs <Data folder> <Work folder>”

The work folder is the mount path where all the files and folders accessible to the user. The data folder is the folder where all the physical blocks will be store.

Scenario to test

1. Create two folders:
 - a. `/home/$USER/Desktop/work_folder`
 - b. `/home/$USER/Desktop/secure_data`
2. Launch secfs: `./secfs /home/$USER/Desktop/secure_data /home/$USER/Desktop/secure_data`
3. Open and navigate to `/home/$USER/Desktop/secure_data` folder
4. Drop files and/or folders to `/home/$USER/Desktop/secure_data`
5. Unmount `/home/$USER/Desktop/secure_data` or stop the software
6. Check `/home/$USER/Desktop/secure_data` folder to find all the encrypted data.
7. Relaunch the software with the same parameters like in step 2
8. Navigate to `/home/$USER/Desktop/secure_data`
9. Open and read the decrypted data.

Security implications

Secfs was designed to have the following security features:

- All files are stored encrypted using AES CBC mode with 128 bit key.
- The database is also stored encrypted on the physical drive and loaded to the memory when the application is running.
- Every file is split to blocks with a random name. When reading the encrypted data folder, it is impossible to know the number of files, folder, their names or their sizes.
- Every block has a unique IV to prevent similar files have the same first encrypted bits.
- The user's password is hashed with salt and the result of that is used as the encryption and the decryption key for the file system.
- Neither user's password nor the hashed key is stored on the physical drive. It kept in memory while the application is running.

Limitations

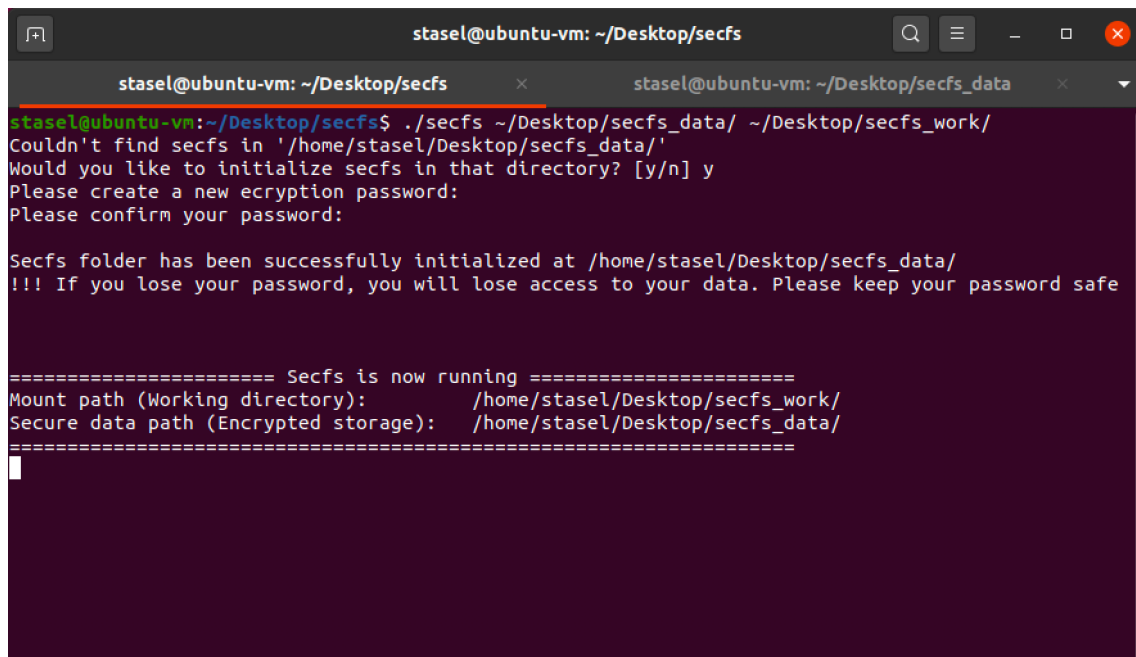
Because of time constraints there are few limitations to the software.

- The software compiles on macOS however FUSE isn't working correctly
- The index and block databases are very inefficient. Both use a simple linear search for simplicity.
- Because of the inefficient database, the more files there are in the filesystem, the slower it works. So large folders will have poor performance.
- Symlinks and hard links were not implemented.

Possible improvements

- Introducing a proper SQLite database to efficiently store and query all blocks and files. This will require to have the database encrypted as well and there are a couple of solutions available.
- Implementing the rest of the FUSE API and the file system to have more feature available.

Screenshots



A terminal window titled 'stasel@ubuntu-vm: ~/Desktop/secfs' with two tabs. The active tab shows the command `./secfs ~/Desktop/secfs_data/ ~/Desktop/secfs_work/` being executed. The output indicates that secfs was not found in the specified path, prompting the user to initialize it. The user confirms with 'y', creates a password, and confirms it. The terminal then displays the successful initialization of secfs at `/home/stasel/Desktop/secfs_data/` and provides the mount and secure data paths.

```
stasel@ubuntu-vm: ~/Desktop/secfs
stasel@ubuntu-vm: ~/Desktop/secfs_data
stasel@ubuntu-vm:~/Desktop/secfs$ ./secfs ~/Desktop/secfs_data/ ~/Desktop/secfs_work/
Couldn't find secfs in '/home/stasel/Desktop/secfs_data/'
Would you like to initialize secfs in that directory? [y/n] y
Please create a new encryption password:
Please confirm your password:

Secfs folder has been successfully initialized at /home/stasel/Desktop/secfs_data/
!!! If you lose your password, you will lose access to your data. Please keep your password safe

===== Secfs is now running =====
Mount path (Working directory):      /home/stasel/Desktop/secfs_work/
Secure data path (Encrypted storage): /home/stasel/Desktop/secfs_data/
=====
```

Figure 1 - Running secfs for the first time

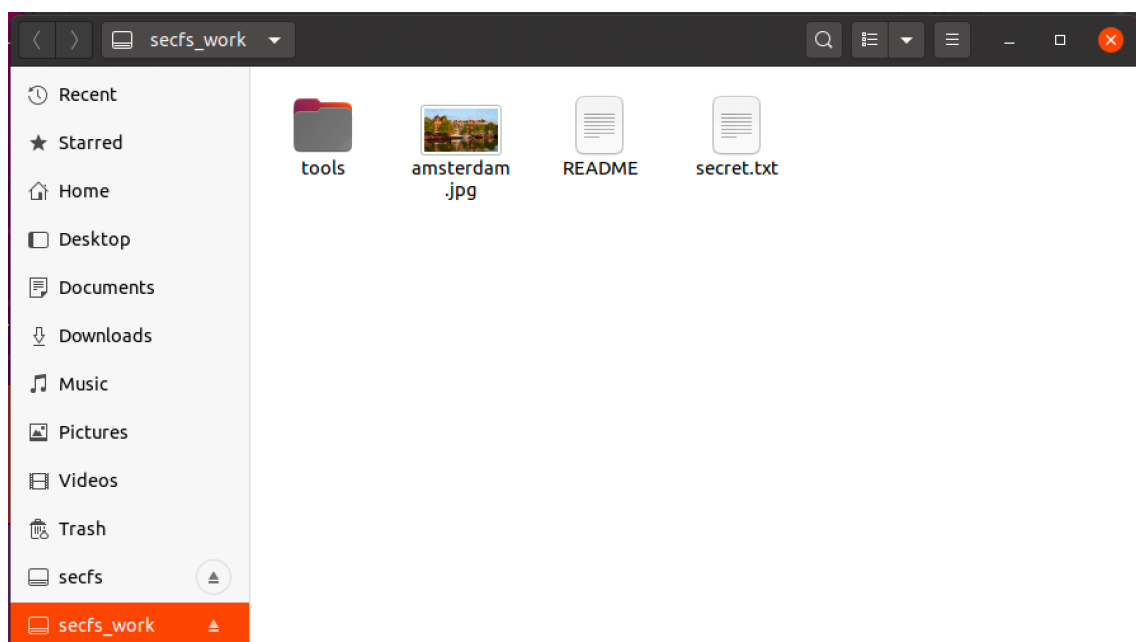


Figure 2 - Decrypted work folder with secfs running

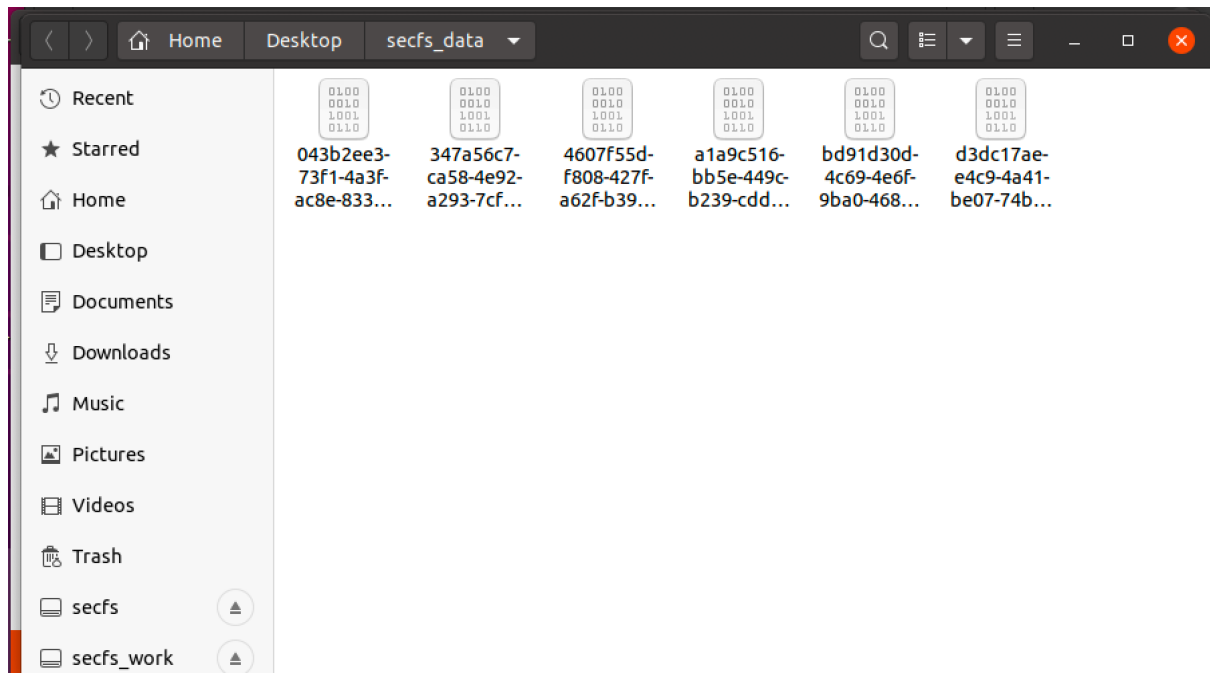


Figure 3 - The physical files stored in the data folder

Resources

Fuse

<https://github.com/libfuse/libfuse>

FUSE (Filesystem in Userspace) is an interface for userspace programs to export a filesystem to the Linux kernel.

A simple example how to interface with Fuse can be found in the following link:

<https://github.com/libfuse/libfuse/blob/master/example/hello.c>

OpenSSL

OpenSSL is a popular encryption library. It will be used for encryption and decryption of data. The usage will be similar to the following example:

https://wiki.openssl.org/index.php/EVP_Symmetric_Encryption_and_Decryption