# Encrypted File System Lab

## 1 Overview

In a traditional file system, files are usually stored on disks unencrypted. When the disks are stolen by someone, contents of those files can be easily recovered by the malicious people. To protect files even when the disks are stolen, we can use encryption tools to encrypt files. For example, we can use "pgp" command to encrypt files. However, this is quite inconvenient; users need to decrypt a file before editing the file, and then remember to encrypt it afterward. It will be better if encryption and decryption can be transparent to users. Encrypted File System (EFS) is developed for such a purpose, and it has been implemented in a number of operating systems, such as `Solaris`, `Windows NT`, and `Linux`.

## 2 Lab Task

In an EFS, files on disks are all encrypted, nobody can decrypt the files without knowing the required secret. Therefore, even if a EFS disk is stolen, its files are kept confidential.

### 2.1 Transparency

The most important feature of EFS is *transparency*. Namely, when legitimate users use the files in EFS, the users do not need to conduct encryption/decryption explicitly; encryption/decryption is conducted automatically by the file system. This distinguishes EFS from normal file-encryption programs.

More importantly, EFS should also be transparent to applications. Any application that work in a traditional file system should still work properly in EFS. When users read a file (encrypted) using a normal editor software, EFS will automatically decrypt the file contents before giving them to the software; similarly, EFS will automatically encrypt the file contents when users write to a file. All these happen on the fly, neither users nor the editor software should be aware of the encryption/decryption process. For example, if users use "`cat`" to look at the contents of a file, `cat` will display the decrypted contents; the decryption is transparently conducted by the EFS. If users use "`vi`" to edit a file, every time they issue a "`save`" command, the contents of the file should be encrypted and then saved to the disk; the encryption is also transparently conducted by the EFS. There is no need to modify application programs.

In this lab, your task is to design and implement an EFS for `Minix`. This lab is a comprehensive lab; it integrates a number of security principles, including encryption, key management, authentication, and access control.

### 2.2 Key Management

**(a) Key storage dilemma.** In an EFS, we can choose to use one single key to encrypt all the files in the encrypted file system; or we can choose to encrypt each file using a different key. In this lab, we choose the

latter approach; we call this approach the *per-file-key* approach. Obviously, these keys cannot be stored on the disk in plaintext; otherwise, adversaries can find those keys after they have stolen the disk. On the other hand, we cannot ask users to type each of those keys every time they try to access a file, because no user can remember all these keys. This is a dilemma that you have to solve in your EFS design.

**(b) Where to store key-related information.** A number of places can be used to store key-related information. One of the places is the i-node data structure. However, i-node does not provide enough space to store extra information that you need. There are two difference approaches to solve this problem, one requires a modification of i-node, and the other redefines a field of i-node. Please see Section 4.1 for details. Another place that can be used to store key-related information is the *superblock*. Please see Section 4.2 for details.

**(c) Authentication:** Users must be authenticated before he can access the EFS. This authentication is not to authenticate users per se; instead, its focus is to ensure that users provide the correct key information. Without the authentication, a user who types a wrong key might corrupt an encrypted file, if such a key is directly or indirectly used for encrypting/decryption files.

Depending on your design, authentication can be conducted in different ways. One way is to just authenticate the `root`, who initially sets up the EFS; another way is to authenticate each user. Regardless of what approach you take, authentication must be kept at minimum: no user is going to like your EFS if you ask users to authenticate themselves too frequently. You have to balance the security and usability of your system. Another authentication issue is where and how to store the authentication information.

**(d) Miscellaneous issues:** There are a number of other issues that you need to consider in your design:

- **File sharing:** Does your implementation support `group` concept in `Unix`? Namely, if a file is accessible by a group, can group member still be able to access the file in EFS?

- **Key update:** If keys need to be updated, how can your system support this functionality? Although you do not need to implement this functionality in this lab, you need to discuss in your report how your system can be extended to support this functionality.

## 2.3 Encryption Algorithm

We assume that AES algorithm (a 128-bit block cipher) is used for encryption and decryption. AES's key size can be 128 bits, 192 bits, or 256 bits, and you can choose one to support in your EFS implementation. The code given in `aes.c` is for encrypting/decrypting one block (i.e. 128 bits), so if you need to encrypt/decrypt data that are more than one block, you need to use a specific AES mode, such as ECB (Electronic Code Book), CBC (Cipher Block Chaining), etc. You can decide which mode to use, but you need to justify your design decision in your report.

Since AES is a 128-bit block cipher, it requires that data must be encrypted as a data chunk of 16 bytes. If the data (in particular, the last block of a file) is not a multiple of 16, we need to pad the data. Will this increase the length of your file? How do you make sure that the padded data is not seen by users?

To use AES, you should install the `libcrypt` library in your `Minix` system. The installation manual is available on the web site of this lab. This library includes both encryption and one-way hashing.

## 3   EFS Setup

Modifying a file system can be very risky. You could end up loosing all data; restoring the old boot image wont help if your file system is messed up. A good way to avoid these troubles is to have an extra hard disk at your discretion. You can always reformat this hard disk when things go wrong. Of course, you do not need a physical hard disk in Vmware, you can use a virtual one. Here are the steps on how to create a virtual hard disk, how to build a file system on the disk, and how to mount and use the file system:

1. Goto the settings page of your virtual machine and add a hard drive.

   (a) Right click on your VM's tab and select **settings** from the menu.

   (b) Click on the **Add** button on the **Hardware** tab.

   (c) Select **Hard Disk** from the popup window and select default options(already highlighted) in the consecutive steps.

   (d) A preallocated hard disk of size 100 MB should be sufficient for our case.

2. Restart Minix.

3. The virtual device would be allocated a device number. If `/dev/c0d0` is your current disk then most likely `/dev/c0d1` would be your new hard drive. Hard drives have name of the form `/dev/cXdXpXsX` where **d** signifies the disk number and **p** signifies the partition number. Assuming that you had just one hard disk earlier (disk 0), your new hard disk number will be 1, hence the name `/dev/c0d1`.

4. `# mkfs /dev/c0d1` : Make a normal Minix file system on the new device. A file system begins with a **boot block**, whose size is fixed at 1024 bytes. It contains an executable code to begin the process of loading the OS. It is not used once the system has booted. The **super block** follows the boot block and contains the information describing the layout of the file system. The `mkfs` command plugs information into this super block. For example, the block size to be used and the MAGIC number used to identify the file system. Since Minix3 supports multiple file systems, the MAGIC number is used to differentiate between different File systems. You would need to modify the `mkfs` command if you are developing a new file system type.

5. `# mkdir /MFS:` Create a directory for mounting the new file system.

6. `# mount /dev/c0d1 /MFS:` Mount the file system onto the `/MFS` directory . The above command performs the following steps for a successful file system mount:

   (a) Set the *mounted on* flag on the in-memory copy of the inode of /MFS. This flag means that another file system is mounted on `/MFS`.

   (b) Load the super block of `/dev/c0d1` onto the super block table. The system maintains a table of the superblocks of all the file systems that have been recently mounted (even if they are unmounted).

   (c) Change the value of *inode-mounted-upon* field of super block entry of `/dev/c0d1` in the super block table to point to `/MFS`.

   When you try to access the a file on the newly mounted file system, say `# cat /MFS/file`. The following steps takes place:

   (a) The system first looks up `/MFS` inode in the root directory.

(b) It finds the *mounted on* flag set. It then searches the super block table for superblocks with *inode-mounted-upon* pointing to the inode of /MFS.

(c) It then jumps to the root of this mounted file system. The *inode-for-the-root-of-mounted-fs* field of the super block points to the root inode of the mounted file system.

(d) It then looks for the file inode on this file system.

If you have come this far then your basic setup is done. All modification will be implemented on this new hard disk.

# 4 Design and Implementation issues

## 4.1 Store extra information in i-node

There are two different ways to use i-node to store extra information for EFS:

- **Without modifying i-node:**

  The disk inode for the version 2 and 3 of Minix file system is represented by the following structure:

  ```
  typedef struct {     /* V2.x disk inode */
    mode_t d2_mode;    /* file type, protection, etc. */
    u16_t d2_nlinks;    /* how many links to this file. HACK! */
    uid_t d2_uid;      /* user id of the file's owner. */
    u16_t d2_gid;      /* group number HACK! */
    off_t d2_size;     /* current file size in bytes */
    time_t d2_atime;    /* when was file data last accessed */
    time_t d2_mtime;    /* when was file data last changed */
    time_t d2_ctime;    /* when was inode data last changed */
    zone_t d2_zone[V2_NR_TZONES]; /* block nums for direct,
                                    ind, and dbl ind */
  } d2_inode;
  ```

  The last zone (i.e., d2_zone[V2_NR_TZONES-1]) is unused (it can be used for triple indirect zone, which is needed only for very large files). We can use this entry to store our extra information. However, this entry has only 32 bits. To store information that is more than 32 bits, we need to allocate another disk block to store that information, and store the address of that block in this zone entry. Please refer to the document [1] for instructions.

- **Modifying i-node:** Another approach is to modify the i-node data structure, and add a new entry to it. This can be done by introducing a character array to store the information you want in the inode structure. If you do this, you are changing the file system type. A number of issues need to be taken care of:

  1. You need to be sure that your inodes are still aligned to the disk blocks. Namely, the size of disk block (1024 bytes) has to be a multiple of the size of inode (the original inode size is 64 bytes).
  2. Changing the inode essentially means that we are creating a new file system. A number of changes need to be made in the operating system, so the OS can support this new file system. Please refer to the document [2] for details.
  3. Defining a new file system allows the EFS to co-exist with the other existing file systems. This gives you the flexibility to extend it in any way you like without touching other file systems.

## 4.2   Store extra information in superblock

The superblock contains information necessary to identify file systems. Each file system has its own superblock. File system specific information can be stored here. For example, you can store the information specific to your EFS in the super block. Unlike the modification of inodes, modification/additions to the superblock is quite straightforward.

## 4.3   Modifying EFS

In `Minix`, the `do_read()` and `do_write()` procedures perform the read and write operations, respectively. Due to the similarity in these operations, both these procedures call `read_write()`, which calls `rw_chunk()`, to read data from the block cache to the user space. Somewhere down the procedure call hierarchy `rw_block()` is invoked to read a block of data from the disk and load it to the block cache. This means that we can implement the encryption/decryption operation in two places:

1. Decrypt a block from the in-memory block cache before passing it to the user space, and encrypt a block while copying it from the user space to the cache. The changes need to be made in `rw_chunk()` for this approach.

2. Decrypt a block while loading the block cache from the disk and encrypt while writing it back.

The first approach is easier as you already have inode pointing to the block (hence its superblock information and the key you might have stored in the inode). The following snippet from `rw_chunk()` illustrates the read/write operations to and from the block cache:

```
if (rw_flag == READING) {
/* Copy a chunk from the block buffer to user space. */

================================================
DECRYPT THE BUFFER TO BE COPIED TO USER SPACE
================================================

r = sys_vircopy(FS_PROC_NR, D, (phys_bytes) (bp->b_data+off),
    usr, seg, (phys_bytes) buff,
    (phys_bytes) chunk);

================================================
ENCRYPT THE BUFFER IN THE CACHE BACK AFTER COPYING
================================================

} else {
/* Copy a chunk from user space to the block buffer. */
r = sys_vircopy(usr, seg, (phys_bytes) buff,
    FS_PROC_NR, D, (phys_bytes) (bp->b_data+off),
    (phys_bytes) chunk);

==================================
ENCRYPT THE BUFFER IN THE CACHE
==================================
```

```
bp->b_dirt = DIRTY;
}
```

You can use the hints provided in the above code to perform the encryption/decryption operations. However there might be other issues that need to be taken care of in `rw_chunk()`.

## 5  Suggestions

1. READ the system call implementation manual supplied by your TA.

2. READ Chapter 5 of the Minix book [3].

3. MODULARIZE your design and implementation. This project can be modularized into 3 distinct stages: file system modification, encryption (and decryption), and key management. File system modification should be drive by the design of your key management.

4. DO NOT leave memory leaks and dangling pointers any where in your code.

5. FOLLOW incremental development strategy. Compile the kernel at every stage and test your changes. Put `printf` statements in your code to trace the kernel code. Even while writing small benign functions, compile and test your code to see the effect. It pays to be paranoid: you don't want your code to fail during the demo, which does happen if there is a memory leak that leads to a race condition.

6. USE `/var/log/messages`, which stores the sartup messages. You can refer to it if the screen scrolls too fast.

7. KEEP a copy of the original image in your home directory. You can revert to it if something fails.

8. USE the snapshot feature of Vmware as version control. Take a snapshot if a feature is completely implemented. It is easier to revert to a snapshot rather than finding the code snippet to delete.

9. USE the right image. The image tracker of Minix is buggy. To be sure that you are using the right image, please follow these steps:

    (a) `# halt`
    (b) `d0d0s0>ls /boot/image`   /* List all the images present */
    (c) `d0p0s0>image=/boot/image/3.1.2arXX`   /* XX is the latest revision number */
    (d) `d0p0s0>boot`

10. DO NOT try to do this project in one sitting. You are supposed to do it in 3-4 weeks. Spread out the work. Late night coding introduces more errors.

11. DO NOT do this on a real hard disk. You will be risking data corruption.

# 6 Testing your implementation

You are free to design your own implementation. A sample implementation might look like the following:

1. `# mkfs -e /dev/c0d1` /* Format `/dev/c0d1` as an EFS */
   `EFS login:` ← Password used for authenticating the user

2. `# mount -e /dev/c0d1 /MFS` /* Mount EFS `/dev/c0d1` on /MFS */
   `EFS login:` ← Enter the password associated with the given EFS. If the password is wrong, the FS should not be mounted.

3. Copy a file from your drive to `/MFS`. It will be in clear text when you read it.

4. To demonstrate that encryption/decryption process is working, comment out the authentication procedure and recompile the kernel. Then mount the file system and try reading the file. It should NOT be in clear text.

# 7 Submission and Demonstration

You should submit a detailed lab report to describe your design and implementation. You should also describe how you test the functionalities and security of your system. You also need to demonstrate your system to us. Please sign up a demonstration time slot with the TA. Please take the following into consideration when you prepare for demonstraiton:

- The total time of the demo will be 15 minutes, no more additional time would be given. So prepare your demonstration so you can cover the important features.

- You are entirely responsible for showing the demo. We will NOT even touch the keyboard during the demonstration; so you should not depend on us to test your system. If you fail to demo some important features of your system, we will assume that your system does not have those features.

- You need to practice before you come to the demonstration. If the system crashes or anything goes wrong, it is your own fault. We will not debug your problems, nor give you extra time for it.

- During the demo, you should consider yourself as salesmen, and you want to sell your system to us. You are given 15 minutes to show us how good your system is. So think about your sales strategies. If you have implemented a great system, but fail to show us how good it is, you are not likely to get a good grade.

- Do turn off the messages your system prints out for debugging purposes. Those messages should not appear in a demonstration.

# References

[1] Karthick Jayaman. How to manipulate the Inode data structure. *Available from our web page*.

[2] Sridhar Iyer. Defining a new file system in `Minix` 3. *Available from our web page*.

[3] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems Design and Implementation*. Prentice Hall, 3rd edition, 2006.