SecureFS: A Password-Protected Encrypted Filesystem Using FUSE and AES-256

1. Introduction

In this project, I developed a simple encrypted filesystem named SecureFS using the FUSE (Filesystem in Userspace) interface on Linux. SecureFS encrypts and decrypts file contents transparently using AES-256 encryption with a password-derived key. This approach enhances data confidentiality by ensuring that stored files are encrypted on disk, protecting them from unauthorized access.

2. Methodology

- ### Initialization and Key Derivation

  Upon launch, SecureFS prompts the user for a password. This password is used to derive a 256-bit AES key using PBKDF2 with SHA-256 and a fixed salt. The derived key is stored in memory and used for all encryption and decryption operations.

```c
// Derive AES key from password using PBKDF2 with a fixed salt
void get_password_and_derive_key() {
    char password[128];
    printf("Enter your Password: ");
    fflush(stdout);

    // Disable echo for password input
    struct termios oldt, newt;
    tcgetattr(STDIN_FILENO, &oldt);
    newt = oldt;
    newt.c_lflag &= ~ECHO;
    tcsetattr(STDIN_FILENO, TCSANOW, &newt);
    fgets(password, sizeof(password), stdin);
    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
    printf("\n");

    password[strcspn(password, "\n")] = 0; // Remove trailing newline

    // Fixed salt for key derivation
    unsigned char salt[8] = "e0eb3621661ceddd";
    if (!PKCS5_PBKDF2_HMAC(password, strlen(password), salt, sizeof(salt),
                           10000, EVP_sha256(), sizeof(aes_key), aes_key)) {
        fprintf(stderr, "PBKDF2 error\n");
        exit(1);
    }
```

- ### File Path Mapping

  All files are stored in a designated backend directory with the .sec extension to distinguish them as encrypted files. This mapping is handled by the get_backend_path() function.

```c
// Construct the full backend file path by appending ".sec" extension
void get_backend_path(const char *path, char *fullpath) {
    snprintf(fullpath, 512, "%s%s.sec", BACKEND_DIR, path);
}
```

- **File Encryption and Decryption**

  SecureFS uses OpenSSL's EVP interface to perform AES-256-CBC encryption and decryption. A fixed IV is used for demonstration purposes.

  Encyption:

```c
// Encrypt plaintext with AES-256-CBC using derived key and fixed IV
int encrypt(unsigned char *plaintext, int plaintext_len, unsigned char *ciphertext) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    unsigned char iv[16] = "1234567890123456"; // Fixed IV (for demonstration only)

    int len, ciphertext_len;

    EVP_EncryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, aes_key, iv);
    EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len);
    ciphertext_len = len;

    EVP_EncryptFinal_ex(ctx, ciphertext + len, &len);
    ciphertext_len += len;

    EVP_CIPHER_CTX_free(ctx);
    return ciphertext_len;
}
```

  Decryption:

```c
// Decrypt ciphertext with AES-256-CBC using derived key and fixed IV
int decrypt(unsigned char *ciphertext, int ciphertext_len, unsigned char *plaintext) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    unsigned char iv[16] = "1234567890123456";

    int len, plaintext_len;

    EVP_DecryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, aes_key, iv);
    EVP_DecryptUpdate(ctx, plaintext, &len, ciphertext, ciphertext_len);
    plaintext_len = len;

    EVP_DecryptFinal_ex(ctx, plaintext + len, &len);
    plaintext_len += len;

    EVP_CIPHER_CTX_free(ctx);
    return plaintext_len;
}
```

- **Main Function and Mounting**

  The main function creates the backend directory if it doesn't exist, prompts for the password, and starts the FUSE main loop.

```c
// Main entry point: create backend directory if needed, get password, then start FUSE
int main(int argc, char *argv[]) {

    mkdir(BACKEND_DIR, 0700);

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <mountpoint>\n", argv[0]);
        return 1;
    }

    get_password_and_derive_key();

    return fuse_main(argc, argv, &securefs_oper, NULL);
}
```

3. How to Use

- ### Prerequisites

  To use SecureFS, the following dependencies must be installed:
  FUSE 3 (libfuse3-dev)
  OpenSSL development libraries (libssl-dev)
  A Linux environment

  To install the required packages on Ubuntu:
  sudo apt update
  sudo apt install libfuse3-dev libssl-dev build-essential

- ### Compiling the Filesystem

  To build the securefs binary from the source code:
  gcc `pkg-config fuse3 --cflags` securefs.c -o securefs `pkg-config fuse3 --libs` -lcrypto
  This command compiles the code with proper FUSE and OpenSSL flags and links the required libraries.

- ### Running and Mounting SecureFS

  First, create a directory that will act as the mount point:
  mkdir ~/secure_mount

  Then, run SecureFS with the mount point:
  ./securefs ~/secure_mount

  You will be prompted to enter a password:
  Enter your Password:

  This password is used to derive a 256-bit AES key which is kept in memory and used for all cryptographic operations during that session.

- ### Using the FileSystem

Once mounted, SecureFS behaves like a normal directory. You can perform typical file operations such as:
cd ~/secure_mount
echo "Hello, SecureFS!" > example.txt
cat example.txt
ls
rm example.txt

Behind the scenes:
When you write to a file like example.txt, SecureFS encrypts the contents and saves it in a hidden backend directory: /tmp/securefs_backend.
The file is stored with an added .sec extension, like file1.txt.sec.
When you read the file (e.g., with cat), SecureFS decrypts it in memory and shows the original content.
So, although you see normal filenames and contents in ~/secure_mount, everything stored on disk is encrypted and unreadable without the password.

- **Unmounting the Filesystem**

When you're done, unmount the filesystem to stop SecureFS:
fusermount3 -u ~/secure_mount
This cleanly shuts down the process and removes the virtual filesystem from the mount point.

4. Testing and Observation

Step 1: Mount SecureFS and Create a File
./securefs ~/secure_mount
Password entered: 1234

echo "Sensitive Info" > ~/secure_mount/data.txt
This creates an encrypted file: /tmp/securefs_backend/data.txt.sec

Step 2: Unmount the Filesystem
fusermount3 -u ~/secure_mount
This flushes the session and clears the encryption key from memory.

Step 3: Remount SecureFS with a Different Password
./securefs ~/secure_mount
Password entered: 4321

cat ~/secure_mount/data.txt
SecureFS attempts to decrypt data.txt.sec using the wrong key.
Since the key doesn't match, OpenSSL decryption fails.
The output is either garbage data or nothing at all.

5. Conclusion

Working on SecureFS helped me understand how filesystems and encryption can work together to protect user data. By using FUSE, I was able to build a virtual filesystem that feels just like a normal folder but secretly encrypts everything behind the scenes. The integration with OpenSSL allowed me to use strong AES-256 encryption, and the password system ensures that only someone with the right password can read the files.

One of the most interesting parts was seeing how file operations like reading and writing could be intercepted and transformed — in this case, to perform encryption and decryption automatically. It really showed me how powerful and flexible user-space filesystems can be.
Through testing, I confirmed that files created with one password couldn't be read with another, which means the encryption is working. Even though the system is simplified (like using a fixed IV and only handling small files), it's still a good demonstration of how you can build real security features on top of basic OS concepts.

Overall, this project gave me hands-on experience with file handling, memory management, and cryptography — all within the context of operating systems.