# CSE 543: Computer Security

Fall 2024
Project 2: Encrypted File System
Due: 11:59 pm (eastern time), Oct. 11, 2024

September 28, 2024

## 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. Encrypted File System (EFS) is developed to prevent such leakages. 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, or if otherwise an adversary can read the file stored on the disk, its files are kept confidential. EFS has been implemented in a number of operating systems, such as Solaris, Windows NT, and Linux.

In this project, you are asked to implement a simulated version of EFS in Java. More specifically, you will need to implement several library functions, which simulates the functionalities of EFS.

### 1.1 Functionality Requirements

To simulate a file system, files are stored in blocks of fixed size. More specifically, a file will be stored in a directory which has the same name as the file. The file will be split to some certain length chunks and stored into different physical files. That is, a file named `/abc.txt` is stored in a directory `/abc.txt/`, which includes one or more physical files: `/abc.txt/0`, `/abc.text/1`, and so on, each physical file is of size exactly 1024 bytes, to simulate a disk block. You will need to implement the following functions:

`create(fileName, userName, password)` Creates a file that can be opened by someone who knows both user name and password. Both user name and password are ASCII strings of at most 128 bytes.

`findUser(fileName)` Returns the user name associated with the file.

`length(fileName, password)` Returns the length of the file, provided that the given password matches the one specified in creation. If it does not match, your code should throw an exception.

`read(fileName, sPos, length, password)` Returns the content of the file for the specified segment, provided that the given password matches. If it does not match, your code should throw an exception.

`write(fileName, sPos, content, password)` Writes content into the file at the specified position, provided that the password matches. If it does not match, the file should not be changed and your code should throw an exception.

`checkIntegrity(fileName, password)` Checks that the file has not been modified outside this interface. If someone has modified the file content or meta-data without using the write call, you should throw an exception.

### 1.2 Security Requirements

**Meta-data storage** You will need to store some meta-data for each file. In file systems, this is naturally stored in the i-node data structure. In this project, we require that meta-data are stored as part of the physical files. You will need to decide where to put such data (e.g. at the beginning of the physical files), and also what cryptographic operations need to be performed to the meta-data. Naturally the first physical file would contain some meta-data, however, you can also store meta-data in each of the physical files.

**User authentication** You need to ensure that if the password does not match, reading and writing is not allowed. You thus need to store something that is derived from the password; however, you should make it as difficult as possible for an adversary who attempts to recover the password from the stored information (perhaps using a dictionary attack).

**Encryption keys** 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, in order to reduce the amount of data encrypted under one key. This key needs to stored as part of the meta-data.

**Algorithm choice** You are required to use AES with 128-bit block size and 128-bit key size. The code for encrypting/decrypting one block (i.e. 128 bits) is provided. When you encrypt/decrypt data that are more than one block, you are required to use CTR, the Counter mode. You will need to decide how to generate the IV's (initial vectors). We assume that an adversary may read the content of the file stored on disk from time to time. In particular, a file may be written multiple times, and the adversary may observed the content on disk in between modifications. Your design and implementation should be secure against such an adversary. Your design should also hide the length of the file as much as possible. The number of physical files used for a file will leak some information about the file length; however, your design should not leak any additional information. That is, if files of length 1700 byes and 1800 bytes both need 2 physical files, then an adversary should not be able to tell which is the case.

**Message authentication** We want to detect unauthorized modification to the encrypted files. In particular, if the adversaries modify the file by directly accessing the disk containing EFS, we want to detect such modifications. Modification to other meta-data such as the user or file length should also be detected. Message Authentication Code (MAC) can help. You need to decide what specific algorithm to use, and how to combine encryption and MAC.

### 1.3 Efficiency Requirements

We also have the following two efficiency requirements.

**Storage** We want to minimize the number of physical files used for each file.

**Speed** We want minimize the number of physical files read or write for each read or write operation. That is, if an write operation changes only one byte in the file, we want accesses to as small number of physical files as possible, even if the file is very long. These two efficiency goals may be mutually conflicting. You need to choose a design that offers a balanced trade-off.

## 2 Project Tasks

You are asked to complete `EFS.java`. More specifically, you will need to implement the following functions. The functionality of the functions are described in Section 1.1. Please do NOT modify the other files. If there is indeed a need to modify them, please describe in detail in your report.

**You are not allowed to use encryption/decryption/hash/MAC functions in Java library. However, you can use the functions provided by us.**

1. `create(String fileName, String userName, String password)`
2. `String findUser(String fileName)`
3. `int length(String fileName, String password)`
4. `byte[] read(String fileName, int sPos, int length, String password)`
5. `void write(String fileName, int sPos, byte[] content, String password)`
6. `boolean checkIntegrity(String fileName, String password)`

In your implementation, if the password is incorrect, please throw `PasswordIncorrectException`. For other exceptions/errors, you can either throw an exception (other than `PasswordIncorrectException`), or handle it yourself.

## 3 Report

In your report, you should include the following:

### 3.1 Design Explanation

**Meta-data design** Describe the precise meta-data structure you store and where. For example, you should describe in which physical file and which location (e.g., in `/abc.txt/0`, bytes from 0 to 127 stores the user name, bytes from 128 to ... stores ...)

**User authentication** Describe how you stores password-related information and conduct user authentication.

**Encryption design** Describe how files are encrypted. How files are divided up, and how encryption is performed? Why your design ensures security even though the adversary can read each stored version on disk.

**Length hiding** Describe how your design hides the file length to the degree that is feasible without increasing the number of physical files needed.

**Message authentication** Describe how you implement message authentication, and in particular, how this interacts with encryption.

**Efficiency** Describe your design, and analyze the storage and speed efficiency of your design. Describe a design that offers maximum storage efficiency. Describe a design that offers maximum speed efficiency. Explain why you chose your particular design.

### 3.2 Pseudo-code

Provide pseudo-code for the functions `create`, `length`, `read`, `write`, and `checkIntegrity`. From your description, any crypto detail should be clear. Basically, it should be possible to check whether your implementation is correct without referring to your source code. You may want to describe how password is checked separately since it is used by several of the functions.

### 3.3 Design Variations

1. Suppose that the only write operation that could occur is to append at the end of the file. How would you change your design to achieve the best efficiency (storage and speed) without affecting security?
2. Suppose that we are concerned only with adversaries that steal the disks. That is, the adversary can read only one version of the the same file. How would you change your design to achieve the best efficiency.
3. Can you use the CBC mode? If yes, how would your design change, and analyze the efficiency of the resulting design. If no, why?
4. Can you use the ECB mode? If yes, how would your design change, and analyze the efficiency of the resulting design. If no, why?

## 4 Provided Code

Please get the code and create your own repository by following this link: `https://classroom.github.com/a/WwUNirTe`

You will find all the source code in `project2/src`. You can use JetBrains IntelliJ IDEA (or any other IDE/editor) to develop and execute your project. We provide a text editor which can be used to verify your design/implementation. To use the editor, simply launch `Editor.java`. We also provide a sample script named `Sample.java`. You are encouraged to start at reading the file. Please notice that in the sample, files are NOT encrypted. Also, there is NO efficiency optimization. You will need to design them yourself.

Here are the details of the sample script.

`create(fileName, userName, password)`

1. Creates a directory named as the name of the file you want to create.(e.g. `/abc.txt/`)
2. Creates the first block and use the whole block as meta-data. (`/abc.txt/0`)
3. Writes “0” into the first block. It means the file length is 0.
4. Writes user name into the first block.

`findUser(fileName)` Finds the user name from the first block. (`/abc.txt/0`)

`length(fileName, password)` Finds the length of file from the first block. (`/abc.txt/0`)

`read(fileName, sPos, length, password)`

1. Computes the block that contains the start position. Assume it is `n1`.
2. Computes the block that contains the end position. Assume it is `n2`.
3. Sequentially reads in the blocks from `n1` to `n2`(`/abc.txt/n1`, ... , `/abc.txt/n2`).
4. Gets the desired string.

`write(fileName, sPos, content, password)`

1. Computes the block that contains the start position. Assume it is `n1`.
2. Computes the block that contains the end position. Assume it is `n2`.
3. Sequentially writes into the blocks from `n1` to `n2`(`/abc.txt/n1`, ... , `/abc.txt/n2`).

4. Updates the first block (meta-data) for the length of file if needed.

`checkIntegrity(fileName, password)` Returns false (Not implemented).

We also provide some utility functions in `Utility.java`. Since the class `EFS` is derived from the class `Utility`, you can directly invoke them. The functions and parameters are listed here. Using these functions is encouraged but not required.

`set usernamePassword()` You can set/reset user name and password by invoking this function.

`readFromFile(File file)` The function will return all the content of file. It will throw an exception if the file cannot be read.

`saveToFile(String s, File file)` The function will write `s` to file (overwrite). It will throw an exception if the file cannot be written.

`setDir()` It will allow you to select a directory. The return value is the chosen directory. You can choose a new directory by typing the full path. If nothing is chosen, returns null.

`encryptAES(String plainText, String key)` The function will return AES encryption result of `plainText` using `key` as key.

`decryptAES(String cipherText, String key)` The function will return AES decryption result of `cipherText` using `key` as key.

`hashSHA256(String message)` The function will return SHA256 result of `message`.

`hashSHA384(String message)` The function will return SHA384 result of `message`.

`hashSHA512(String message)` The function will return SHA512 result of `message`.

`byteArray2String(byte[] array)` This function will help you to convert a byte array to a String. Please notice that if you want to convert a String to a byte array, `String.getBytes("UTF-8")` will work.

## 5 Deliverables

Please commit your changes relatively frequently so that we can see your work clearly. Do not forget to push changes to the remote repository in Github! You can get as low as zero points if we do not see any confirmation of your work. Do not attempt to falsify time stamps, this may be viewed as an academic integrity violation! Finally, include "report.pdf" in your Github repository containing answers to project questions in Sections 3.1, 3.2, and 3.3.

In your Canvas submission, please indicate the GIT commit number corresponding to the final submission. You should also specify your github username. The submission format in plain text

`<your_github_username>:GIT commit`

For example, mustbastani:936c332e7eb7feb5cc751d5966a1f67e8089d331

## 6 Grading

The assignment is worth 100 points total broken down as follows.

1. The report file with answers to questions in Sections 3.1, 3.2, and 3.3 (30 pts, 10 points each section).
2. Task 1 (10 pts), Task 2 (5 pts), Task 3 (10 pts), Task 4-5 (15 pts each).