CS 6324: Information Security

Project 1 - Encrypted File System

Instructions from the TA:

- Your code will be evaluated using a test script. Please make sure it can be built using the tool in our server at `csa-chk22b.utdallas.edu`.
- Each student will use their account in the server. You can login to the server via SSH. The username is your NetID and the temporary password is your UTD ID. When you login for the first time, it will require you to set up a new password.
- Use the UTD VPN if you connect to the server from outside the campus: `https://oit.utdallas.edu/howto/vpn/`.
- In the server, `~/proj1_test_build.tar.gz` contains `run_test_build.sh` to help you check your `EFS.java` against a simple test case.
- Please put all your implementation into the `EFS.java` file. Deduction will apply if your program needs extra setup to build.
- The written portion of the project must be typed. Using Latex is recommended, but not required. The submitted document must be a PDF (no doc or docx are allowed).
- Contact the TA if you have any questions.

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 into trunks 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(file_name, user_name, password) Create 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(file_name) Return the user_name associated with the file.

length(file_name, password) Return 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(file_name, starting_position, length, password) Return 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(file_name, starting_position, content, password) Write 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.

cut(file_name, length, password) Cut the file to be the specified length, provided that the password matches. If it does not match, no change should occur and your code should throw an exception.

check_integrity(file_name, password) Check 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, return False; otherwise, return True, provided that the given password matches. If it does not match, your code 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 other 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.

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 blocks, you are required to use CTR, the Counter mode. You will need to decide how to generate the IV's (initial vectors). For encryption, you can treat a file as a message, or treat a chunk (stored in one physical file) as a message, or choose some other design.

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 between modifications. Your design and implementation should be secure against such an adversary.

You 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 bytes 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 accessed for each read or write operation. That is, if an write operation changes only one byte in the file, we want to access as small a 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 tradeoff.

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 Sec 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 file_nameString user_name, String password)
2. String findUser(String file_name)
3. int length(String file_name, String password)
4. byte[] read(String file_name, int starting_position, int length, String password)
5. void write(String file_name, int starting_position, byte[] content, String password)
6. void cut(String file_name, int length, String password)
7. boolean check_integrity(String file_name, 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 store 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 hide 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, check_integrity, cut**. 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 Submission and Grading

Please submit your “EFS.java” along with your report.
Student may create extra source files but must not update the given source files except “EFS.java.”
If you have added new files, please submit all the source files including them.
Turn off debug output or message alert window in the submitted version.
You will be graded on the following criteria:

Functionality Correctness (50 pts) Your code should implement the read/write/cut etc., correctly, so that a sequence of calls will return correct results.

Design Correctness (30 pts) To what degree does your design/implementation satisfy the specifications? Are there security vulnerabilities? Etc.

Design Variations (20 pts)

5 Project Instruction

You will find all the source code in Project1/src.
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 program named “Sample.java”. You are encouraged to start by 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.
Please feel free to use your own program to verify your implementation if you want.
Here are the details of the sample program.

create(file_name, user_name, password) The detailed steps are

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

findUser(file_name) Find the user_name from the first block (/abc.txt/0).

length(file_name, password) Find the length of file from the first block (/abc.txt/0).

read(file_name, starting_position, length, password) The detailed steps are

1. Compute the block that contains the start position. Assume it is n_1.
2. Compute the block that contains the end position. Assume it is n_2.
3. Sequentially read in the blocks from n_1 to n_2 (/abc.txt/n_1, /abc.txt/(n_1 + 1), ... , /abc.txt/n_2).
4. Get the desired string.

write(file_name, starting_position, content, password) The detailed steps are

1. Compute the block that contains the start position. Assume it is n_1
2. Compute the block that contains the end position. Assume it is n_2.
3. Sequentially write into the blocks from n_1 to n_2 (/abc.txt/n_1, /abc.txt/(n_1 + 1), ... , /abc.txt/n_2).
4. Update the first block(meta data) for the length of file if needed.

cut(file_name, length, password) The detailed steps are

1. Find the number of block needed. Assume it is n.
2. Envoke $write$ to update block n (/abc.txt/$(n + 1)$).
3. Remove the redundant blocks if necessary.
4. Update the first block(meta data) for the length of file.

check_integrity(file_name, password) N/A

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.

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

byte[] read_from_file(File file) The function will return all the content of $file$ as a byte array. So $file$ can be a binary file. It will throw an exception if the file cannot be read.

void save_to_file(byte[] s, File file) The function will write s to a binary file $file$(overwrite). It will throw an exception if the file cannot be written.

File set_dir() 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, return null.

byte[] encrypt_AES(byte[] plainText, byte[] key) The function will return AES encryption result of $plainText$ using key as key.

byte[] decript_AES(byte[] cipherText, byte[] key) The function will return AES decryption result of $cipherText$ using key as key.

byte[] hash_SHA256(byte[] message) The function will return SHA256 result of $message$.

byte[] hash_SHA384(byte[] message) The function will return SHA384 result of $message$.

byte[] hash_SHA512(byte[] message) The function will return SHA512 result of $message$.

byte[] secureRandomNumber(int randomNumberLength) The function will return a random number vector with length $randomNumberLength$. You can assume this is a secure random number generator.