CSCE-110 Fall 2019: Term Project

The Safe Box

Phase 1

The Building Block

Assigned: Thursday, November 7, 2019 in Class

Due Date: Thursday, November 14, 2019

Overview

In this phase of the project your group is required to design and implement the most basic building block of the Safe Box. Essentially, you will need to implement the **import** and **export** functionalities which will entail mainly designing and implementing the shredding, merging, encryption, and decryption functions. You are required to build a parameterized program that can do one of two functions, import or export, based on its parameters. In this phase, you are required to utilize the **crypto++** library to perform encryption and decryption on your files content. Finally, the result of this phase is considered the building block of the next phase and you cannot proceed with phase 2 unless you are fully and completely done with this phase.

Details

In this phase, you will implement your first Safe Box version which is relatively simple and basic. The Safe Box in this phase should provide two main functionalities, import and export. The import functionality should take a file located in your file system and store it into the Safe Box. The import should shred the file and encrypt its content. The command line parameters of the import functionality are:

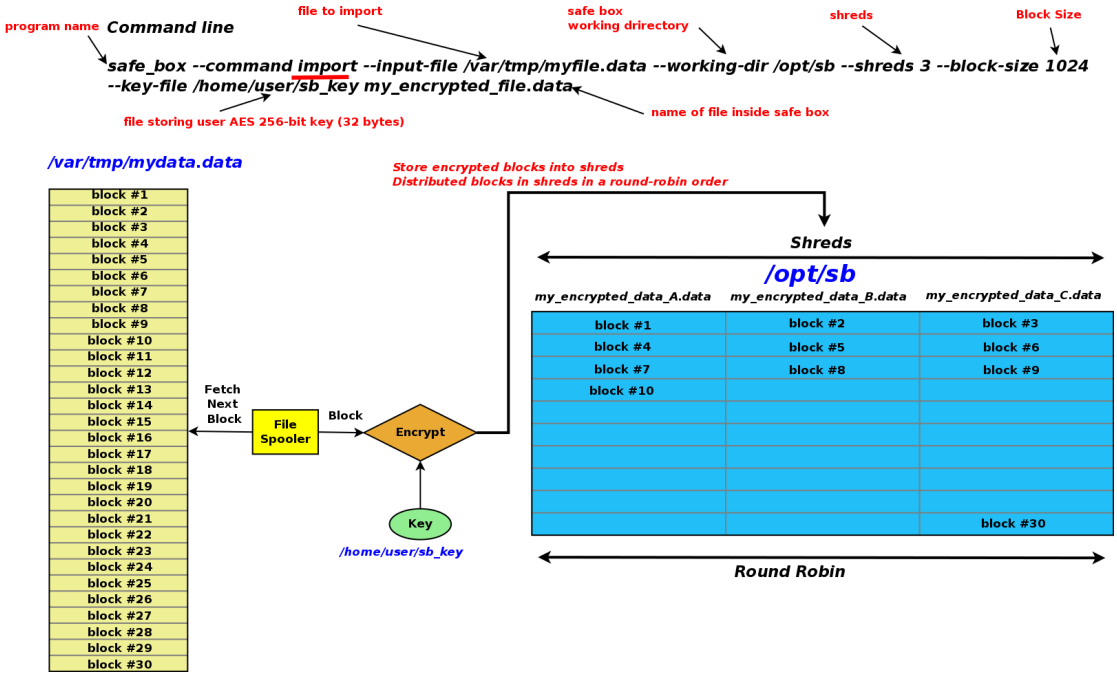
- Full path **file name** of the file to be stored in the Safe Box.
- Full path Safe Box **working area**.
- Number of **shreds**.
- **Block Size** in KB.
- **Encryption key**.
- Name to be used to store the file inside the Safe Box.

The **file name** should be a valid name of a file located on your file system. The **working area** should be a valid path on your file system where the Safe Box should use to store the shredded encrypted version of the target file. The file should be stored in different files we call them **shreds**. Each shred contains a

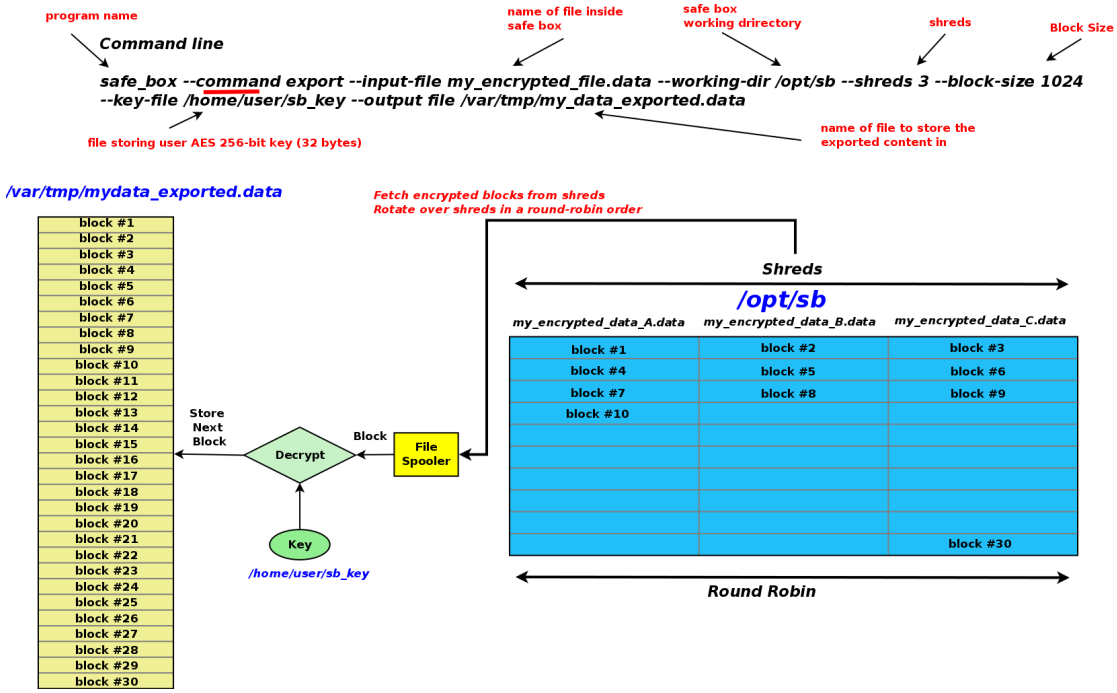
part of the original file. Since the original file can be relatively huge, we might not be able to load a whole shred and encrypt it in one go due to memory limitations, consequently we read blocks of fixed size, encrypt them one by one, and store them in their corresponding shred. The **block size** defines the number of kilo bytes that will be processed at a time. Of course the last block might not fulfill the required size and hence it will be smaller than the rest of the blocks and you will need to handle that. In your implementation in this specific phase, which will essentially change in subsequent phases, you will need to store the encrypted blocks in a **round-robin** order. Finally, you will use the **encryption key**, provided by the user, to encrypt each block before storing it in its corresponding shred; what will be stored is the encrypted version of the block.

For encryption, you are required to use crypto++ library to apply the **AES** (Advanced Encryption Standard) algorithm to securely encrypt the content of each block. AES comes in different versions identified by the key size, typically 128-bit (16 bytes), 192-bit (24 bytes), and 256-bit (32 bytes). The larger the key size the more secure the encryption and hence you will use the 256-bit version which will entail providing a key of size 32 bytes. Providing such large key on the command line may be cumbersome and not handy so you can store your key in a text file and provide the name of the text file as a command parameter to your program.

To be able to illustrate the way the Safe Box import functionality works, lets take a practical example. Consider we have a file that is 30 MB that we would like to store in the safe box and it is located at ***/var/tmp/myfile.data***, and that the block size is 1024 (1024*1024 bytes = 1048576 bytes = 1 MB). The working area of the Safe Box will be located at ***/opt/sb/***, your key is stored in ***/home/user/sb_key***, the number of shreds is 3, and finally the name that will be used to store the file in the Safe Box is ***my_encrypted_file.data***. The following diagram illustrates the mechanics of the Safe Box export operation.



The export operation works exactly in the reverse order. The following diagram illustrates the mechanics of the Safe Box export operation.



Finally, you need to handle all errors and report them to the user. You are required to use exception handling to achieve that and to build an exception handling hierarchy to be able to extend the handling of exceptions and error handling as functionalities are added.

What to submit

1. All your code.
2. Your code should be split among header files (.h) and source files (.cpp), and all your implementation need to be in the source files. You need to have a very good reason for including implementation in header files, and an explanation of such motives need to be included in your report.
3. Very detailed in-code documentation.
4. A modular Makefile hierarchy.
5. A report describing how to build the software using g++, including all design decisions taken and their alternatives, and explaining anything unusual to the teaching assistant. The report should be in PDF format.
6. Contributions files; each group member will provide a file named after her/his name explaining what she/he did and her/his contribution in this project phase.
7. Do not submit any object, or executable files. Any file uploaded and can be reproduced will result in grade deductions.

How to submit:

First of all, you will need to commit your code to your GIT repository. You will need to upload a text file named phase1_git_commit_id.txt, that includes your GIT commit ID to blackboard. The commit ID submitted by your group on blackboard will be used by the TA to grab your code which will be considered and graded, so make sure to include the correct version commit id. The TA will compare the commit ID date on the GIT repository with the deadline to know if the code was committed on the GIT repository before or after the deadline.

Moreover, you are required also to compress all your work: source code, report, readme file, and any extra information into a zip archive. You should name your archive in the specific format <Section_ID>_<Team_ID>_Term_Project_F19_Phase1.zip. Finally, upload your archive to blackboard.

If you were able to package your work into a docker environment, which is optional, you can mention your docker repository in your readme file, but it is a must to upload all your source code assignment material to blackboard and commit you code as well to your GIT repository.

This is a group project and you should elect a group captain among your group members, and all the submissions are expected to be performed through her/his blackboard account. IMPORTANT: EVERY GROUP SHOULD SUBMIT THROUGH THE GROUP CAPTAIN ONLY.

IMPORTANT: You will have to present your work and run a demo to the TA and/or the Professor in order to get your grade.

Grading

This project phase is worth 25% of the overall project grade. This phase will be graded on a 100% grade scale, and then will be scaled down to the 25% its worth. The grading of this phase will be broken down according to the following criteria:

a. Safe Box Program — 80%
    i. Functionally running — 50%
    ii. Code Quality — 15%
    iii. In-Line Documentation — 15%
b. Report — 20%

The work will be evaluated based on the correct execution of the safe box program based on the functionalities stated in this phase, the code correctness and neatness, design decision aspects, and how much you have utilized the concepts of abstraction, encapsulation, modularity, and hierarchy. It is very important to highlight that although in-code documentation is worth 10%, yet missing or unclear documentation might result in more grade deductions if the grader cannot verify the correctness of your code and logic, and your provided in-line documentation were not sufficient to clarify such matters.

Delays

You have up to 3 calendar days of delay after which the corresponding phase will not be accepted and your grade in that case will be ZERO. For every day (of the 3 allowed days per phase), a penalty of 10% will be deducted from the total grade.