16308185 Shaun Jose

CS3031 Documentation - Project 2

Introduction

The program I created consists of 6 source files and contains the following main features for file sharing on Google Drive:

1. Generate a symmetric key
2. Encrypt and upload files to drive using the generated symmetric key
3. Decrypt and download files from drive using the symmetric key
4. Add and remove users to the Secure cloud Storage
5. Login of a user using username and password, with details saved to file
6. Generation of private and public key pair for each user except for admin, and save private key to file for each user
7. Admin sending symmetric key to another user by encrypting it with their public keys so only they can read the key, for reach user
8. New symmetric key generation by admin, private_public key pair generation for all users, and re-encryption of all files on the drive once a user is removed, for security. (Note that the new symmetric key is encrypted and resent to all users in this case)

All these features are implemented in different parts and files of the program. In this document, I will explain the high level details of my program, file by file, and how the different features are implemented in the relevant files. I will start with KeyManagementSystem.py, as it's utility and concepts needs to be solidified before I can move on to the explanation of the other source files in my program, which depend on it greatly.
If needed, the code of all the files, along with the README (containing instructions on how to run the program), can be found at the end of this document.
Please move on to the next page to find the explanation of these files and features.

KeyManagementSystem.py

This file is significantly important, but what it does is surprisingly simple. It contains a class KMS (short for KeyManagementSystem) which is concerned with 2 aspects - Key + fernet initialisation, and returning the symmetric key to a user.

<u>How does Key + fernet initialisation work?</u>
When an instance of this class is created, the class begins to initialise the symmetric key and fernet, using the symmetric key. It first looks for the symmetric key in the Admin's folder, and if it doesn't exist, a new symmetric key is generated and saved to a file in the Admin's folder. This prevents the KMS from repeatedly generating a new symmetric key repeatedly by just simply reading it from the file for the next time the program runs! This covers the first aspect that this file deals with. Note that the generation of a new symmetric key is a method in this file (generate_new_key), which calls another method in GroupHandler to share the new key with all the other users, and is very important for security when a user is removed (stay tuned, explained in GroupHandler.py)

<u>How does returning a symmetric key to a user work?</u>
This aspect is covered by the getKey() method, and all it does is return the symmetric key to the user, based on the username passed to it. If the user is "admin", it could read the symmetric key straightaway from the file in Admin's folder, where the symmetric key is saved. However, if it's another user, my program ensures that the symmetric key is saved to that user's folder in a file, after it's been encrypted with the user's public key. In this case, the method reads the user's private key which is saved in a file inside the user's folder, reads the encrypted symmetric key from the other file in the user's folder, decrypts the symmetric key using the private key and returns the decrypted symmetric key

This is all the file is concerned with and is extremely simple, as promised, but you can see why it is so important due to how often the symmetric key is used, and how dependent the program is on the symmetric key. As this file also initialises the Admin's folder, this could very well be the starting point of my program, and it is (in a way), as a KMS object instantiation is the first thing the main does.
Note that the fernet is just an object based on the key, used for cryptography. Now that we've got the key and fernet out of the way, we can move on to FileSharing, which could very well be the most complex file of my program, because of it's length

FileSharing.py

As the name suggests, this file is concerned with user authentication for google drive, a whole lot of file sharing and "resetting the drive". The reason this file is huge is because of the many error checks and handling that has to be done before, during or after a file is shared. It contains the class GoogleDriveAccess.

Authentication

When an object of this class is instantiated, the constructor opens up a Local web server in order to ask the user to log in. Once a user logs in, it's details are saved to the file authCookie.json, due to the file settings.yaml, whose main purpose is to save details once logged in to save the user trouble from having to constantly authenticate him/herself every time the user logs in or every time the program runs. Note that the GoogleAPI credentials are needed for this authentication to work, and in the case of my program, all users use this same file for authentication as my program runs locally of course. Note that the constructor also takes the username of the usr seeking access and creates a folder for that user's file-keeping, if it doesn't already exist

General File Sharing

The GoogleDriveAccess class has the upload_file and download_file methods, out of which both need the filename to be uploaded/downloaded, and the fernet from the KMS class for encryption and decryption.
The upload_file method first checks if the specified file exists in the relevant user's folder, and if not, it fails to upload the file. It then checks if there's a shared folder up on the drive, and if not, it creates a shared folder (shared between all the users in the group). It then checks whether a file with the same filename is already up on the drive. This is extremely important, as Google drive actually allows multiple files with the same filename to be stored in the same folder, but because that would cause confusion for the users in the group, I forcefully decline uploading a file with the same filename. This reduces the confusion and also ensures that the same file isn't uploaded multiple times, unnecessarily. If all is good (file exists in user's folder and no file with the same name up on the drive), I read the contents of the file saved locally, create an encrypted version of that file, upload it to the drive as filename.encrypt to show that it's encrypted, and delete the local encrypted version. (Note that encryption-decryption is done using the fernet from KMS)

The download_file method normalises the filename given to it, as needed, and checks if the shared folder is on the drive. If it exists, it checks if the file specified is on the drive. If either of these conditions fail, the method stops. However, if all is good, the method downloads the contents of the encrypted file from the drive, decrypts the contents and saves a file with the same filename (without the .encrypt) to the user's folder, where the user can see the file in plaintext (or it's original form)
upload_file and download_file return the encrypted and decrypted filename of the file respectively, which is then printed out, so that the user knows where to find the encrypted file (on the drive) or decrypted file (in the user's local folder).

The hard part of file sharing is done, and the other half of it is just dealing with the user. The startSharing() method is called from GroupHandler (explained later), and asks a user to give it commands for file sharing. It accepts the "upload filename", "download filename" user input, and if valid, it lets the upload_file and download_file methods deal with it. Additionally, only if the user is admin, it also allows the user to add or remove users from the group, by just providing the username of the user. The method checks if the user is already in the group in case of add and whether it's not in the group in case of remove. If these issues don't exist, the method simply adds the specified user or removes the user from the group. Note that adding a user just adds the username and nothing else (no public_private key pair, no encrypted symmetric key saving, and no password initialisation), as this is dealt with once a user logs in (as explained in the GroupHandler file - see later). This method continuously accepts commands from the user until the user enters e to exit this loop. This method returns True if an existing user is removed, and false otherwise, as, if an old user is removed, the drive and keys need to be reset, for security purposes! That's all for file sharing, and we can move on to understand what "resetting the drive" actually means

Resetting the drive

The method resetDrive() is called when a user is removed, by the GroupHandler file. Recall that the previous class returns True or false to indicate whether an existing user has been removed, and if GroupHandler gets a True from that method, it calls this one to reset the drive. Resetting the drive consists of 3 main operations :

1. Downloading and Decrypting all the files from the drive (using the download_file method for all files on the drive), saving it locally and then deleting the files on the drive

2. Generating a new key using the method generate_new_key in the KMS class (which saves the key to the file in Admin's folder, and in turn calls the method in GroupHandler to make the all the existing users regenerate their private_public key pair and save the new symmetric key encrypted with the newly generated public key of the user, in the user's folder)
3. Uploading and encrypting all the locally saved files with the newly generated symmetric key, to the drive

These 3 main operations carried out by the reset_drive method is extremely important when a user is removed (that's why it's called only when a user is removed). This is because if the user removed had stored all the public_private key pairs of all the users and the encrypted symmetric key, it could decrypt the symmetric key using the user's private key and get the symmetric key in plaintext. This in turn would allow the user to decrypt the files from the drive! Even if the user didn't have the public_private key pairs of all users, and previously saved the symmetric key in plaintext, which could be retrieved from the Admin's folder, it would have the symmetric key and be able to read all the files on the drive. Thus, changing the public_private key pairs of all users, and the symmetric key, and re-encrypting all the files which were on the drive with the new symmetric key, ensures that the user will not be able to read any of the files or the symmetric key for that matter, and thus provides that additional solid security which I felt was necessary to implement in my program.

Note that the users' private_public key pair only exist so that the admin can securely share the symmetric key with each user by encrypting it with their respective public keys. Since the admin saves the symmetric key in plaintext in it's folder, it already has the symmetric key and thus doesn't need a public_private key pair

We've covered every aspect of what this file does, and now we can move on to the GroupHandler file, which is relatively simple (when compared to this file). Although slightly complex, this is the last complex file of my program, as you will see that the others are extremely simple and short. So let's move on to cover up the last complex part of my program

GroupHandler.py

As the name would imply, this file deals with handling the secure cloud storage group. It accepts a user log in, initialises what's needed for a user, saves the user's details to the files and changes the keys of each user if needed (when a user is removed)

Accepting a user login

Recall that adding a user just adds the username of that user (and nothing else). Existing users, however, must already have their own folder, a public_private key pair, an encrypted symmetric key saved to the folder and their passwords, of course. To distinguish new and existing (or old) users between each other, all the new users' usernames are saved to a new_users file while the existing or old users usernames and passwords are saved to the main 'Users.txt' file. At login, if the method is provided a new username, it asks the user to set and confirm a password, and if successful, the user is removed from new_users and saved in the 'Users.txt' file along with it's password. If the method is provided with an old username however, it simple asks for the password and grants access to the user accordingly. This method keeps going on until someone logs in successfully, and is the "doorway" to the rest of the program. After a successful login, this method initialises what's needed for the user, gets the symmetric key and fernet from the user using the getKey() method in KeyManagementSystem (KMS) and then calls startSharing() in the FileSharing.py file so that the user can begin to give upload and download commands, and in case of the admin, add user and remove user commands as well.

As the secure group can be altered by the add and remove user commands, the files are saved using the save methods called in this class, and if True is received from startSharing() (indicating that an existing user has been removed), the user's folder is deleted and resetDrive() is called to reset all the keys as explained earlier.

Initialises what's needed for a user

If the user is new, it doesn't have a folder, a private_public key pair, or even the symmetric key. A GoogleDriveAccess instance is created for the user logged in, which grants the user access to the drive and creates a folder for the user. Then, using the RSA algorithm in the cryptography python library, a private_public key pair is initialised for the user, and the private key is serialised and saved, as the public key is easily derivable from the private key using the program and the library. Lastly, the user gets the symmetric key,

encrypts it using it's public key and saves it in a file in it's folder. It then goes on to do the initialisation need for an existing user, as after this point, it is an existing or old user (since it's password, folder, key pair and encrypted symmetric key have been saved)
The initialisation of an old user consist of 3 trivial steps :- getting the decrypted symmetric key using the getKey() method in KMS, creating a fernet object using this key and returning this fernet. This is needed as the acceptUser method accepts the fernet from this method after the user login, and then needs to pass it on to the startSharing method (in FileSharing) in order for file decryption and encryption which uses the fernet.
That is all the initialisation needed, and to summarise it:
Each user ends up with it's own folder, password, private_public key pair and encrypted symmetric key, and the user then gets the symmetric key in plaintext and returns the fernet of that key to be used by encryption-decryption in FileSharing

<u>Saves the user details to files</u>
Now that I've explained the distinction between new and old users, this should be very simple to explain. save_old_users is a method that takes the dictionary of old/existing users' usernames and passwords, and saves it in a certain format in the 'Users.txt' file. save_new_users is another method that just takes an array of the new uninitialised users and saves it to another file in a particular format. The format is required in order to read the info from the files properly, and separate users and passwords from each other, for which I have two other dedicated methods, getCurrUsersPass which returns a dictionary of the usernames and passwords by reading 'Users.txt' and getNewUsers which returns an array of the new usernames by reading from the newUsers.txt file. All 4 of these methods use readFile() and saveFile() in FileFunctionalities.py, which is trivial and explained later.

<u>Changes the keys</u>
When a user is removed, the keys should change, as explained earlier. Recall that the initialisation of a new user involved creating a folder if it doesn't exist, generating a private_public key pair and getting the symmetric key. The method change_all_keys() is called from generate_new_keys, after a new key is generated. change_all_keys just gets all existing users who are supposed to have the encrypted key, and calls the initaliseNewUsers() method for each old/existing user, so that the private_public key pairs are regenerated, the private key file is overwritten, the new symmetric ket is retrieved and the encrypted new symmetric key with the new public key is saved to the user's file as

well, in place of the previous encrypted symmetric key. This method does a small part of the work of “resetting the drive”, and is placed in this group, as it has to do with the old/existing users and their keys.

We’ve finally covered up the last important part of our program, and we’re done with the 3rd source file. The other 3 source files are trivial and extremely simple, and might not even need an explanation, but you can find their explanations below if needed.

main.py

This file has one function (main) and consists of 3 lines.
1st line - Creating an instance of KMS in order to generate a key and admin’s folder, in case they don’t exist (when the program runs for the first time ever)
2nd line - calling acceptUser() in GroupHandler to accept a user login and carry on with file sharing
3rd line - Printing a “Goodbye” message when the acceptUser method is done (which is when the user is done sharing files)

That’s it for main, moving on to FileFunctionalities

FileFunctionalities.py

This file consists of two method.:
readFile - Takes a filename, reads all it contents, returns the contents.
saveFile - Takes a filename and contents, creates/overwrites the file with the contents

The last file is ‘constants.py’ and has no functionalities really, except for constant declarations, and so there’s no explanation for this. However, the source code of this file can be found in the appendix, just like the rest of these files

Limitations

My program works for the most part and beyond, as it covers cases with similar filenames on google drive and "resetting the drive" for additional security, and I'm quite happy with my implementation. However, there's always scope for improvement, and some of them are to do with dealing with limitations, some of which are listed below:

1. Multiple folders with the same name of Google drive might be a problem, as the program won't be able to differentiate between the files, but my program will still work. This shouldn't be much of a problem as the DRIVE_FOLDER name is too specific and set to SecureCloudStorage, so it is unlikely that a folder would have the same name in the same drive.

2. The password for new users are set by the users for the first time when the user enters his/her username. If an intruder guesses the username between the time when the new user is added and time when the new user logs in to set his/her password, the intruder will be given the opportunity to set the password for the new user thus gaining access to the SecureCloudStorage.

3. If the intruder gains access to local files, intruder can see all the user's passwords, which is extremely dangerous, but since it's a local assignment, we assume that they do not have local copies of the files that admin handles. In a real life scenario, each user would have only their own files locally, and only the admin's machine or a database would have the user-password list

4. A file to be uploaded has to be in the user's folder -> (I don't really consider this as a limitation but as an improvement in file organisation, because that was the main purpose of creating folders for each user)

5. One client_secrets.json file is used, whereas users should have their own client_secrets files. I can't help it in this case or do much about it as this program is a local simulation (people would use their own client_sercets.json file in the real world).

Please refer to the next pages for the code.

“Just Show Me The Code”

KeyManagementSystem.py

```
# author: ShaunJose (@Github)
# File description: Contains the KeyManagementSystem class

# Imports
import os
from constants import ADMIN_FOLDER, SHARED_KEY_FILE, PRIV_KEY_FILE
from FileFunctionalities import readFile, saveFile
from cryptography.fernet import Fernet
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes


# key management system class
class KMS:

    # Constructor
    def __init__(self):
        self.__init_key_fernet__() # initialise self.key and self.fernet


    # Initialises a new key if it doesn't exist
    def __init_key_fernet__(self):
        """
        Initialises a new key if it doesn't exist

        return: None
        """
        # read key if key exists else generate a new one
        if os.path.exists(ADMIN_FOLDER + "/" + SHARED_KEY_FILE):
            self.key = readFile(ADMIN_FOLDER + "/" + SHARED_KEY_FILE)
            self.fernet = Fernet(self.key) # create fernet obj
        else:
            self.__generate_new_key__()


    # Generate and save a key
    def __generate_new_key__(self):
        """
        Generates a new key (and fernet obj) and saves it to the file

        return: None
        """

        # generate a key
        self.key = Fernet.generate_key()

        # create admin's directory if it doesn't exist
        if not os.path.isdir(ADMIN_FOLDER):
            os.mkdir(ADMIN_FOLDER)

        # save key to appropriate file in admin's directory
        saveFile(ADMIN_FOLDER + "/" + SHARED_KEY_FILE, self.key)

        # create fernet object
        self.fernet = Fernet(self.key) # create fernet obj

        # Broadcast the new key to everyone (method in group handler class)
        change_all_keys()


    # Returns the symmetric key of the group
    @staticmethod
    def getKey(username):
        """
        Returns the symmetric key to the caller

        param username: User's who's folder has to be dealt with and read into

        return: Symmetric key
        """

        sym_key = None
        if username != "admin":
            # Read private key of user
            folder_name = username + "_files/"
            filepath = folder_name + PRIV_KEY_FILE
            priv_key = serialization.load_pem_private_key(readFile(filepath), password = None,
backend = default_backend())

            # Read and decrypt the encrypted symmetric key
            filepath = folder_name + SHARED_KEY_FILE
            sym_key = priv_key.decrypt(readFile(filepath), padding.OAEP(mgf =
padding.MGF1(algorithm = hashes.SHA256()), algorithm = hashes.SHA256(), label =
None))
        else: # if it's the admin, it's already stored in plaintext
            sym_key = readFile(ADMIN_FOLDER + "/" + SHARED_KEY_FILE)

        return sym_key

# place here due to cyclic imports
from GroupHandler import change_all_keys
```

FileSharing.py

```
# author: ShaunJose (@Github)
# File description: Contains class for google drive access and authentication, and
methods for file sharing purposes

# Imports
import os
import shutil
from constants import ENCR_EXTENSION, DRIVE_FOLDER, DRIVE_ROOT_ID,
INSTR_UP, INSTR_DOWN, INSTR_ADD, INSTR_REM, INSTR_EXIT, ADMIN_FOLDER
from FileFunctionalities import readFile, saveFile
from KeyManagementSystem import KMS
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive



# Google drive access class
class GoogleDriveAccess:

    # Constructor
    def __init__(self, username):
        self.googleAuth = GoogleAuth() # init authentication obj
        self.googleAuth.LocalWebserverAuth() # ask user to complete auth, if not already
completed
        self.drive = GoogleDrive(self.googleAuth) # init drive obj
        # Create user-files folder here, where all the user's files would be saved (if it doesn't
already exist)
        self.user_folder = username + "_files"
        if not os.path.isdir(self.user_folder):
            os.mkdir(self.user_folder)



    # Uploads a file after encrypting it
    def upload_file(self, filename, fernet):
        """
        Saves and uploads an encrypted version of the file named filename, if the file exists
locally and no duplicate is on the drive

        param filename: Name of the file to be uploaded
        param fernet: The fernet object created in KeyManagementSystem class

        return: Filename of encrypted file, if file uploaded successfully, else None
        """

        # add path to filename if needed
        prevFilename = filename
        filename = self.normaliseFilename(filename)

        # return None if file doesn't exist
        if not os.path.exists(filename):
            print("Error 404: File specified not found")
            return None

        # get the drive's main folder ID
        folderID = self._get_file_ID_(DRIVE_FOLDER, DRIVE_ROOT_ID)

        # Create and share folder on drive if it doesnt exist, and get folderID
        if folderID == None:
            folderID = self.__create_share_folder__()
            print("Created shared folder on drive")

        # Check if file with identical name is already on the drive
        encryptedFilename = prevFilename + ENCR_EXTENSION
        encryptedFileID = self._get_file_ID_(encryptedFilename, folderID)
        if encryptedFileID != None: # implies there's a file identically named
            print("This file is already up on the drive!")
            return None

        # Save encrypted file locally
        plain_text = readFile(filename) # getting file contents
        cipher_text = GoogleDriveAccess._encrypt_(plain_text, fernet)
        normEncryptedFilename = encryptedFilename
        normEncryptedFilename = self.normaliseFilename(normEncryptedFilename)
        saveFile(normEncryptedFilename, cipher_text) # Saving file locally

        # Upload encrypted file to google drive folder
        fileToUpload = self.drive.CreateFile({"title": encryptedFilename, "parents": [{"id":
folderID}]})
        fileToUpload.SetContentFile(normEncryptedFilename) # set file contents
        fileToUpload.Upload()

        # Delete local version of encrypted file here
        os.remove(normEncryptedFilename)

        return encryptedFilename



    # Downloads a file and saves the decrypted version of it
    def download_file(self, filename, fernet):
        """
        Downloads a file named filename from the drive and saves the decrypted version, if
the file exists

        param filename: Name of the file to be downloaded
        param fernet: The fernet object created in KeyManagementSystem class

        return: Filename of decrypted file, if file downloaded successfully, else None
        """

        # Add encrypt extension if it's not in filename
        if ENCR_EXTENSION not in filename:
            filename = filename + ENCR_EXTENSION

        # If drive's shared folder doesn't exist, return None
        folderID = self._get_file_ID_(DRIVE_FOLDER, DRIVE_ROOT_ID)
        if folderID == None:
            print("Error 404: Shared folder not found on drive")
            return None

        # Return None if file doesnt exist on drive
        fileID = self._get_file_ID_(filename, folderID)
        if fileID == None:
            print("Error 404: File specified does not exist on the drive")
            return None

        # Download encrypted file contents from google drive
        downloadedFile = self.drive.CreateFile({'id': fileID})
        cipher_text = downloadedFile.GetContentString()

        # Get decrypted file data and save decrypted file locally
        plain_text = GoogleDriveAccess._decrypt_(cipher_text.encode('utf-8'), fernet)
        filename = self.normaliseFilename(filename)
        decryptedFilename = filename[ : -len(ENCR_EXTENSION)]
        saveFile(decryptedFilename, plain_text)

        return decryptedFilename



    # Gets the file ID of the file specified in the specified parents
    def _get_file_ID_(self, filename, parents):
        """
        Returns the fileID if it exists on the drive in the parents folder

        param filename: name of file to be found
        param parents: Directory in which filename is to be found

        return: id of the file if found in parents, else None
        """
        fileID = None

        # Iterate through files & folders in specified parent, until file/folder needed is found
        file_list = self.drive.ListFile({'q': "'%s' in parents and trashed=false" %
(parents)}).GetList()
        for file in file_list:
            if file['title'] == filename:
                fileID = file['id']
                break

        return fileID



    # creates and shares the shared drive folder
    def __create_share_folder__(self):
        """
        Creates and shares the shared google drive folder to all users in the group

        return: ID of the shared folder on the drive
        """

        folder = self.drive.CreateFile({'title': DRIVE_FOLDER, "mimeType": "application/
vnd.google-apps.folder"})
        folder.Upload()

        return folder["id"]



    # Adds folder path to filename if it doesn't already exist
    def normaliseFilename(self, filename):
        """
        Adds folder path to filename if filename doesn't already have it

        param filename: Filename to normalise
        return: Normalized file name
        """

        if self.user_folder not in filename:
            filename = self.user_folder + "/" + filename

        return filename


    # Encrypts file
    @staticmethod
    def _encrypt_(plain_text, fernet):
        """
        Encrypts the plain_text passed to it

        param plain_text: Data to be encrypted
        param fernet: The fernet object created in KeyManagementSystem class

        return: The encrypted plain_text
        """

        cipher_text = fernet.encrypt(plain_text)
        return cipher_text


    # Decrypts a file
    @staticmethod
    def _decrypt_(cipher_text, fernet):
        """
        Decrypts the cipher_text passed to it

        param cipher_text: Data to be decrypted
        param fernet: The fernet object created in KeyManagementSystem class

        return: The decrypted cipher_text
        """

        plain_text = fernet.decrypt(cipher_text)
        return plain_text



    # Simulates file sharing for a user
    @staticmethod
    def startSharing(username, fernet, users_pass, new_users):
        """
        Allows a user to upload or donwload files. Is user is admin, also allows user to add
or remove users from the group

        param username: username of user uploading/downloading files
        fernet: fernet of symmetric key of group
        users_pass: Dictionary of old users and their passwords
        new_users: An array of the uninitialised new users

        return: True if old user removed, False otherwise
        """

        GoogleDriveAccess.printInstructions(username == "admin") #print instructions to the
user

        # accept user input
        removed_old = False
        old_users = users_pass['users']
        passwords = users_pass['passwords']
        exit = False
        driveAccess = GoogleDriveAccess(username)
        while not exit:
            userIn = raw_input(INSTR_EXIT)
            if userIn == "e":
                exit = True
```

```
elif len(userIn) > 7 and userIn[0:7] == "upload ": # upload command
    filename = driveAccess.upload_file(userIn[7:], fernet)
    if filename != None:
        print("\n\nEncrypted version in: " + filename + " on the drive")
    else:
        print("\n\nUpload failed.")

elif len(userIn) > 9 and userIn[0:9] == "download ": # download
    filename = driveAccess.download_file(userIn[9:], fernet)
    if filename != None:
        print("\n\nDecrypted version saved in: " + filename)
    else:
        print("\n\nDonwload failed.")

else: # still could be add, remove, or neither
    if username == "admin": # for admin add and remove exists
        if len(userIn) > 4 and userIn[0:4] == "add ": # add
            user = userIn[4:]
            if user in old_users or user in new_users:
                print(user + " is already a part of the group.")
            else:
                new_users.append(user)
                print(user + " successfully added!")

        elif len(userIn) > 7 and userIn[0:7] == "remove ": # remove
            user = userIn[7:]
            if user in old_users:
                if user == "admin":
                    print("You can't remove yourself!")
                else:
                    index = old_users.index(user)
                    del old_users[index]
                    del passwords[index]
                    shutil.rmtree(user + "_files") # del folder
                    removed_old = True
                    print(user + " has been kicked out!")
                elif user in new_users:
                    new_users.remove(user)
                    print(user + " has been kicked out!")
                else:
                    print(user + " is not a part of the group.")

            else:
                print("Invalid input. Please try again")
        else:
            print("Invalid input. Please try again")

    return removed_old


# Prints the instructions for file sharing
@staticmethod
def printInstructions(adminInstructions):
    """
    Prints the instructions for filesharing

    param adminInstructions: True to print add and remove user messages, keep False
otherwise

    return: None
    """

    if adminInstructions:
        print(INSTR_ADD)
        print(INSTR_REM)

    print(INSTR_UP)
    print(INSTR_DOWN)
# Changes the symmetric key, and re - encrypts all files on the drive
@staticmethod
def resetDrive():
    """
    Downloads and decrypts all files from the drive, and saves them to admin's folder.
Deletes all files on the drive, changes the symmetric key and in turn, key of all users, and
re-encrypts all donwloaded files using the new key

    return: None
    """

    # init two main instances
    kms = KMS()
    driveAccess = GoogleDriveAccess("admin") # get admin's drive access

    # download and then delete all files, decrypt them and save them to admin's folder
    filenames = []
    folderID = driveAccess._get_file_ID_(DRIVE_FOLDER, DRIVE_ROOT_ID)
    file_list = driveAccess.drive.ListFile({'q': "'%s' in parents and trashed=false" %
(folderID)}).GetList()
    for file in file_list:
        filenames.append(driveAccess.download_file(file['title'], kms.fernet))
        file.Delete()

    # generate new key and change it for all users' files
    kms.__generate_new_key__()

    # upload encrypted version of all files (using new key and fernet)
    for file in filenames:
        file = file[len(ADMIN_FOLDER) + 1 :] #cut out the 'admin_files/'
        driveAccess.upload_file(file, kms.fernet)
```

GroupHandler.py

```
# author: ShaunJose
# File description: Handles the group (including accepting user login, user additon and
deletion etc)



# Imports
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes
from cryptography.fernet import Fernet
import os
from constants import USERS_FILE, N_USERS_FILE, USER_DELIM,
USER_PASS_DELIM, PRIV_KEY_FILE, SHARED_KEY_FILE, ADMIN_PASS
from FileSharing import GoogleDriveAccess
from FileFunctionalities import readFile, saveFile
from KeyManagementSystem import KMS

# Accepts a user log in
def acceptUser():
    """
    Accepts a user login, then calls file sharing method to share files

    return: None
    """

    # get current users (with passwords) and new users
    users_pass = _getCurrUsersPass_()
    new_users = _getNewUsers_()

    # Accept valid and verified username and password
    username = ""
    fernet = None
    while True:
        username = raw_input("Username: ")
        if username in users_pass['users']: # if old usr
            index = users_pass['users'].index(username)
            password = raw_input("Password: ")
            if password == (users_pass['passwords'])[index]:
                print("Login successful!\n")
                fernet = handle_old_user(username)
                break
            else:
                print("Incorrect details. Please try again.\n") # Wrong password
        elif username in new_users: # if new user
            s_password = raw_input("Set password: ")
            c_password = raw_input("Confirm password: ")
            if s_password == c_password:
                users_pass['users'].append(username) # add user details to curr users dict
                users_pass['passwords'].append(s_password)
                new_users.remove(username) # not a new user anymore
                print("Password saved and login successful!\n")
                fernet = handle_new_user(username)
                break
            else:
                print("Passwords don't match. Login failed.\n")
        else: # non-existent user
            print("User specified is not currently in the group.\n")

    # start file sharing
    removed_old = GoogleDriveAccess.startSharing(username, fernet, users_pass,
new_users)

    # Save users and new users
    save_old_users(users_pass)
    save_new_users(new_users)
    if removed_old:
        print("Changing keys and re-encrypting all files...")
        GoogleDriveAccess.resetDrive()
        print("Re-encryption successful.")


# Reads current user's and passwords file and return a dict with users and pass
def _getCurrUsersPass_():
    """
    Gets the current users and passwords and returns a dictionary with the users and
passwords. If file doesn't exist, it creates it with admin's details

    return: dict with users array (accesed by 'users' key) and passwords array (accessed
by 'passwords' key)
    """

    # if file doesn't exist, create file with admin details
    if not os.path.exists(USERS_FILE):
        ret = {'users' :["admin"], 'passwords': [ADMIN_PASS]}
        saveFile(USERS_FILE, "admin" + USER_PASS_DELIM + ADMIN_PASS +
USER_DELIM)
        return ret

    # get users file contents
    curr_users_contents = readFile(USERS_FILE)
    users_passwords = curr_users_contents.split(USER_DELIM)
    ret = {'users': [], 'passwords': []} # init users and pass dict

    # separate users from passwords
    for userPass in users_passwords:
        if not userPass: # if string is empty we're done
            break
        tmpArr = userPass.split(USER_PASS_DELIM)
        ret['users'].append(tmpArr[0])
        ret['passwords'].append(tmpArr[1])
    return ret



# Reads the new user file and returns array with new usernames
def _getNewUsers_():
    """
    Reads contents of the new user files and gives back an array with new usernames.
Create empty file if file doesn't exist

    return: Array with new usernames
    """

    # Create emtpy file if it doesn't exist
    if not os.path.exists(N_USERS_FILE):
        saveFile(N_USERS_FILE, "")
        return []

    # get new user's usernames
    new_users_contents = readFile(N_USERS_FILE)
    new_users = new_users_contents.split(USER_DELIM)

    return new_users[:-1] # ignore last empty element



# Does the initialisation for the new user
def handle_new_user(username):
    """
    Initialisation of a new user. Creates new folder for user files, generates private and
public key and saves serialized private key, and also saves the encrypted symmetric key
to user's folder, to be decrypted by the user's private key

    param username: username of the new user to be initialised

    return: the fernet
    """

    driveAccess = GoogleDriveAccess(username) # This creates a folder for the user, and
also gives access to the drive

    # key pair generation and verification
    priv_key = rsa.generate_private_key(public_exponent = 65537, key_size = 2048,
backend = default_backend())
    pub_key = priv_key.public_key()

    # Serialization of private key for file storage
    priv_bytes = priv_key.private_bytes(encoding = serialization.Encoding.PEM, format =
serialization.PrivateFormat.TraditionalOpenSSL, encryption_algorithm =
serialization.NoEncryption() )

    # Save serialized private key to user's folder in plaintext
    folder_name = username + "_files/"
    filepath = folder_name + PRIV_KEY_FILE
    saveFile(filepath, priv_bytes)

    # encrypt (using user's pubkey) and save symmetric key to user's folder
    kms = KMS()
    encrypted_sym_key = pub_key.encrypt(kms.key, padding.OAEP(mgf =
padding.MGF1(algorithm = hashes.SHA256()), algorithm = hashes.SHA256(), label =
None))
    filepath = folder_name + SHARED_KEY_FILE
    saveFile(filepath, encrypted_sym_key)

    # do what you do for the old user anywyay, and return the fernet it returns
    return handle_old_user(username)



# Gets symmetric key, creates a fernet and returns it
def handle_old_user(username):
    """
    Gets symmetric key using method in KMS. Initialises fernet based on the symmetric
key

    param username: username of the old user being handled

    return: fernet
    """

    sym_key = KMS.getKey(username) # get sym key

    fernet = Fernet(sym_key) # create a fernet obj

    return fernet


# Saves all the old users in USERS_FILE
def save_old_users(users_pass):
    """
    Saves the old users to the relevant file, in proper format

    param users_pass: Dictionary containg arrays of old users and passwords

    return: None
    """

    # make content using dictionary and delimiters
    content = ""
    for i in range(len(users_pass['users'])):
        content += users_pass['users'][i] + USER_PASS_DELIM
        content += users_pass['passwords'][i] + USER_DELIM

    # save the file
    saveFile(USERS_FILE, content)
```

```
# Saves all the new users to N_USERS_FILE
def save_new_users(users):
    """
    Saves the new users to the relevant file, in proper format

    param users: Array of new users

    return: None
    """

    # make content using array and delimiters
    content = ""
    for i in range(len(users)):
        content += users[i] + USER_DELIM

    # save the file
    saveFile(N_USERS_FILE, content)


# Changes the private and public key of all (old) users except for admin, and also saves
the encrypted version of the NEW or CURRENT symmetric key to the user's files
def change_all_keys():
    """
    Changes the private and public key of all (old) users except for admin, and also saves
the encrypted version of the NEW or CURRENT symmetric key to the user's files

    return: None
    """

    # reinitialise pirv_pub key pair for all users and save new sym key
    users_pass = _getCurrUsersPass_()
    users = users_pass['users']
    for user in users:
        if user != "admin":
            handle_new_user(user)
```

FileFunctionalities.py

```
# Returns the contents of a file
def readFile(filename):
    """
    Reads and returns contents of a file named with the passed filename

    param filename: Name of the file to be read

    return: Contents of file if exists, else None
    """

    # Read file contents
    fileIn = open(filename, "r")
    contents = fileIn.read()
    fileIn.close()

    # return file contents
    return contents

# Creates and saves a file with contents
def saveFile(filename, contents):
    """
    Saves a file with the name in filename, and adds the passed contents to it

    param filename: Name of the file to be created/overwritten
    param contents: Contents that have to be saved to the file

    return: None
    """

    # Create/overwrite file with contents in it
    fileOut = open(filename, "w")
    fileOut.write(contents)
    fileOut.close()
```

main.py

```
# author: ShaunJose (@Github)
# File description: Run this to run project.

# Imports
from KeyManagementSystem import KMS
from FileSharing import GoogleDriveAccess
from GroupHandler import acceptUser



# Main method
if __name__ == "__main__":

    kms = KMS() # to initialise shared key incase it hasnt been initialised

    acceptUser() # Call method to accept user login

    print("Goodbye!")
```

constants.py

```
# Name of the admin's folder
ADMIN_FOLDER = "admin_files"

# File name where the symmetric key is saved
SHARED_KEY_FILE = "Shared Key"

# Make encrypted file and descrypted extensions
ENCR_EXTENSION = ".encrypt"

# Google drive folder name where all encrypted files will be saved online
DRIVE_FOLDER = "Secure Cloud Storage"

# Id of the root folder on drive
DRIVE_ROOT_ID = "root"

# Name of file where User names and passwords are saved
USERS_FILE = "Users.txt"

# Name of file where new users are saved
N_USERS_FILE = "newUsers.txt"

# Delimiter between users in files
USER_DELIM = "\n\n"

# Delimiter between user and user's password in files
USER_PASS_DELIM = "\n"

# admin's default password
ADMIN_PASS = "hardToGuess"

# Private key file name for each user
PRIV_KEY_FILE = "Private Key"
```

```
# Instructions for user input (upload, download, add user, remove user)
INSTR_UP = "To upload a file to the drive, type upload filename (with extension). Make
sure the file is in your user folder"
INSTR_DOWN = "To download a file from the drive, type download filename (with
extension)"
INSTR_ADD = "To add a user, type add username"
INSTR_REM = "To remove a user, type remove username"
INSTR_EXIT = "Enter e to exit, or continue file sharing\n"
```

README.md

~~~
# Secure_Cloud_Storage_App

### Note - In progress

### Description
An implementation of a secure cloud storage application for Google drive



### Run

**Attention!!**
Make sure you have your *'client_secrets.json'* file for the Google Drive API, containing your client_id, client secret key, etc

```
python main.py
```



### Dependencies
1. cryptography
```
pip install cryptography
```

2. pydrive
```
pip install pydrive
```
~~~