Course: CS3031 Advanced Telecommunications
Title: Assignment 2 - Securing the Cloud
Name: Leong Kai Ler
Student Number: 15334636
Date: April 8, 2019

Introduction:

Purpose:

The aim of this project is to develop a secure cloud storage application for Dropbox, Box, Google Drive, Office365 etc. For example, the application should be able to secure all files that are uploaded to the cloud, such that only people that are part of your “Secure Cloud Storage Group” will be able to decrypt your uploaded files. To all other users the files will be encrypted.

A suitable key management system will be designed and implemented for the application that will allow files to be shared securely, and users to be added and removed from “Secure Cloud Storage Group”. The application can be set up on a desktop or mobile platform and make use of any open source cryptographic libraries.

Design

The implementation of this assignment can be divided into mainly two components: User Interface and a Cloud Storage Group. These components make use of different modules that consists of helper functions to execute their tasks.

The highlight of the design revolves around how to transmit symmetric keys to valid users in the group securely. The symmetric keys used for encryption when uploading files and decryption when downloading. One critical assumption that is took on during the development of the application is such that the activities of the cloud storage group is always being peeked on and the contents in it is always exposed to outsiders.

As a result, the contents in the cloud storage has to be encrypted for data protection and can only be decrypted by users who have access to the symmetric keys. To achieve this, a symmetric key has to shared securely among the users. The initiative taken here is to have the users send their public keys to the cloud storage group, so that it can be used to encrypt the symmetric key during transmission.

For this assignment, the target cloud storage is Google Drive. Further and more circumstantial explanation of the modus operandi will be elaborated below.

User Interface

Implementation:

The user interface allows every user to log in using their user names and create their own unique asymmetrical keys. The asymmetrical keys of every user is stored, so that it can loaded the next time the same users access the user interface. Each asymmetrical key pair consist of a public key and a private key. The public key is sent along with the user name and the user port number to the cloud drive group as authentication terms to request for the symmetric key used for encryption and decryption of files in the group. If the user is not listed as one of legitimate members in the group, no symmetrical keys will be sent in response and an illegal trespassing exception will be reported. On the other hand, if the request is authorized, the public key sent will be utilized to encrypt the symmetric key and the package will then be transmitted back as response.

Subsequently, upon attaining the encrypted symmetric key from the cloud storage group, the private key in the key pair is then used to decrypt the encrypted symmetric key. To summarise, the symmetric key ensures the overall protection of the files in the cloud storage while its sanctity is assured by the asymmetrical key pairs approach used during transmissions. All users have to continuously request for the symmetrical keys from the cloud storage group to ensure they have the most updated version key in case of renewals being carried out due to a member being ousted.

A local GoogleDrive web server is then set up to handle authentication to the target folder in the cloud drive, before creating a GoogleDrive instance to enable any alterations to be done between local and the cloud drive. This allow the users to upload and download files from the cloud storage. During uploading, the symmetrical keys are used to encrypt the file contents before being written to the new file. Similarly, when downloading, the file contents read will be decrypted using the same key and written to file before being saved to the downloads folder. The application always assume that the visibility of the cloud storage is always compromised, so even if a non-member user managed to gain access to the files on the drive, he or she won't be able to read the file contents since he or she doesn't have access to the symmetric keys used in encryption and decryption.

The user class make use of the Key Manager module to create, save and load the users' corresponding keys, while all the encryption and decryption work are facilitated by importing the Encryptor module. Moreover, a socket is set up to relay requests to the cloud storage group whereas a listener is initialize to receive responses.

Code:

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
Created on Thu Apr  4 09:11:25 2019

@author: Adamlkl
"""
import os
import sys
import random
import Encryptor
import KeySaver
from multiprocessing.connection import Client
from multiprocessing.connection import Listener
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive

folder_Id = ’17oua44SP5sR6E_g_h3a9Ua5qjHqAFvFy’

’’’
    - try to establish connection with CloudGroup
    - send over public key for symmetrical key encryption
    - get encrypted symmetrical key from CloudGroup
    - if user is not in the userlist, encrypted symmetrical key won’t be sent
’’’
def retrieve_symmetrical_key(username, address, port, group_address,
   group_listener, user_key):
    ser_key = KeySaver.serialize_key(user_key)
    conn = Client(address, authkey=b’secret password’)
    conn.send([username, port, ser_key])
    conn.close()

    group_connection = group_listener.accept()

    encryption_key = b"error"
    try:
        encryption_key = group_connection.recv()
    except:
        pass
    group_connection.close()
    return KeySaver.generate_symmetric_key(user_key, encryption_key)

# download file from drive folder with key passed and save it to downloads
def retrieve_file(symmetric_key, filename, drive):
    file_list = drive.ListFile({’q’: "’17oua44SP5sR6E_g_h3a9Ua5qjHqAFvFy’ in
       parents and trashed=false"}).GetList()
    for file1 in file_list:
        # find file to be downloaded
        if file1["title"] == filename:
            encrypted_text = file1.GetContentString()
            decrypted_text = Encryptor.decrypt(encrypted_text.encode(),
               symmetric_key,)

            # printing downloaded text to check results
            print(decrypted_text)
            # save files in downloads
            with open (os.path.join("downloads",filename),'wb') as d_file:
                d_file.write(decrypted_text)
                d_file.close()

# encrypt file with key passed and upload it to drive folder
def upload_file(symmetric_key, filename, drive):
    u_file = drive.CreateFile({"parents": [{"kind": "drive#fileLink", "id": "
       17oua44SP5sR6E_g_h3a9Ua5qjHqAFvFy"}],'title':filename})
    with open (os.path.join("testfiles",filename),'rb') as uploadfile:
        plain_text = uploadfile.read()
        encrypted_text = Encryptor.encrypt(plain_text, symmetric_key)
        u_file.SetContentString(encrypted_text.decode())
        uploadfile.close()
    u_file.Upload()

# print usage of the user page
def usage():
    print "Command List: \n1. upload <file> \n2. download <file> \n3. quit\n"

# print list of files in drive folder
def print_fileList(drive):
    # Auto-iterate through all files that matches this query
    file_list = drive.ListFile({'q': "'17oua44SP5sR6E_g_h3a9Ua5qjHqAFvFy' in
       parents and trashed=false"}).GetList()
    for file1 in file_list:
        print('title: %s, id: %s' % (file1['title'], file1['id']))

def main():
    # get username
    username = raw_input("Enter username?\n")

    '''
        - try to load user's key using username
        - if found, load the key from file
        - otherwise, create a new one and save it in files
    '''
    try:
        user_key = KeySaver.load_key(username)
    except:
        user_key = Encryptor.generate_private_key()
        KeySaver.save_key(username, user_key)

    '''
        - sets up local Google webserver to automatically receive
          authentication code from user and authorizes by itself.
    '''
    gauth = GoogleAuth()
    # Creates local webserver and auto handles authentication.
    gauth.LocalWebserverAuth()
    # Create GoogleDrive instance with authenticated GoogleAuth instance.
    drive = GoogleDrive(gauth)

    '''
        - attempting to establish connection with CloudGroup to get symmetric
          key for encryption of files
    '''
    address = ('localhost', 6000)
    port = random.randint(6001,7000)
    group_address = ('localhost', port)      # family is deduced to be 'AF_INET
       '
    group_listener = Listener(group_address, authkey=b'secret password')
    sym_key = retrieve_symmetrical_key(username, address, port, group_address,
         group_listener, user_key)
    running = True

    while running:
        inputs = raw_input("How can I help you?\n")

        # requests for symmetrical key in case it is changed
        sym_key = retrieve_symmetrical_key(username, address, port,
           group_address, group_listener, user_key)

        # handles instructions from users
        argv = inputs.split(' ')
        if len(argv)>2:
            print "Usage: python ex.py "
            sys.exit(1)
        else:
            command = argv[0]

            if command == "upload":
                filename = argv[1]
                upload_file(sym_key, filename, drive)
            elif command == "download":
                filename = argv[1]
                retrieve_file(sym_key, filename, drive)
            elif command == "quit":
                running = False
                print "Goodbye"
            else:
                usage()

if __name__ == '__main__':
    main()
```

Cloud Storage Group

Implementation:

The cloud storage group is mainly responsible for the management of the operations of the cloud storage such as adding or removing users from the members list, re-encrypting all the files on the drive folder when a user is evicted, listen to requests from users that are either members or non-members and relay the appropriate response message.

Upon activation, the cloud storage group will attempt to load a stored list into a set data structure which contains the user names of registered members. This is later used to validate the access of symmetrical keys when requested by users. Then, the symmetrical keys are loaded from the local data storage or created if no copy exists and stored for future usage and reference.

The cloud storage group employs the functionality of listener and clients from the multiprocessing library in Python. Multiprocessing is a package that supports spawning processes using an API similar to the threading module. The multiprocessing package offers both local and remote concurrency to the design. A listener, which is a wrapper for a bound socket is used to 'listen' for connections established by sockets from users. An authentication key is also used as the secret key here for an HMAC-based authentication challenge. This ensures that the connection can only be carried by users through the specified interface.

While active. the listener will attempt to receive requests from users through the connections established. These requests come in the form of a list containing the user names, address of user's listener and a public key. The cloud storage group will then respond by validating the user's membership using the member list. A illegal access error exception will be raised if a non-member attempts to request for the symmetrical key. On the contrary, if a member requests is authorised, a socket, i.e. Client will be used instead to set up a connection to the member's listener. The connection will serve as a pathway to transmit a symmetric key encrypted using the received public key back to the user.

For this component, the main concern that has to be take into account is a situation where a member is evicted, since adding a new member only involves the member having access to the symmetric key the next time he applies for it. In situation where a member is removed from the list, it is essential to have counter measures designated for it. There are two alternatives available for this complexity. Either the Cloud Storage Group could broadcast the new symmetric key to every members in the user list or it could just wait for an incoming request for symmetric keys from its group members and send the new ones during the transmissions. In this context, the latter is selected to as there would be no need for new symmetrical keys if no upload or download activities are carried out by the users, hence unnecessary broadcasting can be avoided, thus reducing the complexity of the routine algorithm.

Knowing that, the only protocols needed when a member is removed from the group are as follows:

- Decrypt all the files in the corresponding cloud drive.
- Create a new symmetric key and save the key.
- Encrypt all the files in the corresponding cloud drive.

The mentioned steps are crucial and requisite to avoid the removed member from having access to the drive and being able to read the contents of the uploaded files on the drive.

Code:

```python
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
Created on Fri Apr  5 13:19:19 2019

@author: Adamlkl
"""
import os
import pickle
import threading
import KeySaver
import DriveManager
import Encryptor
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from multiprocessing.connection import Client
from multiprocessing.connection import Listener


class CloudGroup(threading.Thread):

    def __init__(self, users, lock, listener, sym_key):
        super(CloudGroup,self).__init__()
        self.users  = users
        self.lock = lock
        self.listener = listener
        self.key = sym_key
        self.running = True

    # change symmetric key of the CloudGroup
    def change_key(self,new_key):
        self.key = new_key

    # shut down the CloudGroup
    def close(self):
        self.running = False

    '''
        - continously listens to users
        - get public key from valid users
        - encrypt symmetric key with corresponding public key
        - send encryted symmetric key back to the user
    '''
    def run(self):
        while True:
            connection = self.listener.accept()
            message = []
            try:
                message = connection.recv()
            except:
                pass
            # print(msg)
            if len(message)==3:
                self.lock.acquire()
                if message[0] in self.users:
                    client_address = ('localhost', message[1])
                    response = Client(client_address, authkey=b'secret
                       password')

                    public_key = KeySaver.load_public_key(message[2])
                    encrypted_key = Encryptor.encrypt_symmetric_key(public_key
                       , self.key)
                    response.send(encrypted_key)
                    response.close()
                else:
                    print('Unauthorised access by %s' % (message[0]))
                self.lock.release()

            connection.close()


# print commands in usage list
def usage():
    print "Command List: \n1. add <username> \n2. remove <username> \n4. list
       \n4. quit\n"

'''
    - decrypt all the files in drive folder using old symmetric key
    - create new symmtric key and save it
    - encrypt all the files in drive folder using new symmetric key
'''
def reset(key, drive, client_handler, sym_key_filename):
    folder = drive.ListFile({'q': "'17oua44SP5sR6E_g_h3a9Ua5qjHqAFvFy' in
       parents and trashed=false"}).GetList()
    DriveManager.decrypt_all_files(key, folder)
    new_key = Encryptor.generate_key()
    with open(os.path.join("SymmetricKey",sym_key_filename), 'wb') as key_file
       :
        key_file.write(new_key)
    client_handler.change_key(new_key)
    DriveManager.encrypt_all_files(new_key, folder)

def main():
    filename = 'UserList'
    # load users in CloudGroup
    try:
        infile = open(filename,'rb')
        user_list = pickle.load(infile)
        infile.close()
    except IOError:
        print "Could not read file:", filename
        user_list = set()

    # load Symmetric key
    sym_key_filename = 'Symmetric_Key.txt'
    try:
        with open(os.path.join("SymmetricKey",sym_key_filename), 'rb') as
           key_file:
            sym_key = key_file.read()
    except:
        print "Could find symmetric key file:", filename
        sym_key = Encryptor.generate_key()
        with open(os.path.join("SymmetricKey",sym_key_filename), 'wb') as
           key_file:
            key_file.write(sym_key)

    '''
        - sets up local Google webserver to automatically receive
          authentication code from user and authorizes by itself.
    '''
    gauth = GoogleAuth()
    # Creates local webserver and auto handles authentication.
    gauth.LocalWebserverAuth()
    # Create GoogleDrive instance with authenticated GoogleAuth instance.
    drive = GoogleDrive(gauth)

    '''
        - Create listener to receive requests from users
        - attempting to establish connection with users to send symmetric
          key for encryption of files
    '''
    address = ('localhost', 6000)     # family is deduced to be 'AF_INET'
    listener = Listener(address, authkey=b'secret password')
    lock = threading.Lock()
    client_handler = CloudGroup(user_list, lock, listener, sym_key)
    client_handler.start()
    running = True

    while running:
        command = raw_input('How may I help you?\n')
        argv = command.split(' ')
        # use lock to prevent anyone contacting CloudGroup at during
           management
        lock.acquire()

        # add user to list
        if argv[0] == 'add':
            user_list.add(argv[1])

        # remove user from list
        elif argv[0] == 'remove':
            user_list.remove(argv[1])
            reset(sym_key, drive, client_handler, sym_key_filename)

        # shut down CloudGroup
        elif argv[0] == 'quit':
            #save user files
            users = open('UserList','wb')
            pickle.dump(user_list,users)
            users.close()
            client_handler.close()
            running = False
            print 'Goodbye'

        # print valid users
        elif argv[0] == 'list':
            print 'Users:'
            for x in user_list:
                print(x)

        # print usage
        else:
            usage()
        lock.release()


if __name__ == '__main__':
    main()
```

Modules

Since a lot of subroutines that leverage available cryptography libraries are shared by both classes mentioned above, modules are created for different purposes to provide helper functions while significantly reducing the need to import corresponding cryptographic libraries in every classes aside from repeating the same chunk of code.

Encryptor

Implementation

The Encryptor is set up to process all the encryption and decryption of the files passed. This module uses Fernet since it guarantees that a text encrypted by it cannot be manipulated or read without the specific passed key. Fernet is known for its symmetric authenticated cryptography implementation(also known as “secret key”).

Helper functions included in this module are:

- generating a fresh Fernet key.
- generating a private, 2048 bits long RSA key which unlike keys used in symmetric cryptography, has a complex internal structure with specific mathematical properties.
- encrypt a plain text using the passed key and return the encrypted version
- decrypt an encrypted text using the passed key and return the decrypted version

Code:

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
Created on Fri Mar 29 14:35:29 2019

@author: Adamlkl
"""
'''
    https://docs.python-guide.org/scenarios/crypto/
'''
from cryptography.fernet import Fernet
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes

# generate normal key
def generate_key():
    key = Fernet.generate_key()
    return key
```

```
# generates a new RSA private key using the provided backend with 2048 bits
def generate_private_key():
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
        backend=default_backend())
    return private_key

# encrypt symmetric key using public key passed
def encrypt_symmetric_key(public_key, sym_key):
    encrypted_key = public_key.encrypt(
                        sym_key,
                        padding.OAEP(
                            mgf=padding.MGF1(algorithm=hashes.SHA256()),
                            algorithm=hashes.SHA256(),
                            label=None
                        )
                    )
    return encrypted_key

# encrypt plain text using key passed
def encrypt(plain_text, key):
    cipher_suite = Fernet(key)
    return cipher_suite.encrypt(plain_text)

# decrypt encrypted text using key passed
def decrypt(encrypted_text, key):
    cipher_suite = Fernet(key)
    return cipher_suite.decrypt(encrypted_text)

# -----------------testing functions--------------------------------------
def encryptt(filename, key):
    cipher_suite = Fernet(key)

    plain_file = open(filename, "rb")
    plain_text = plain_file.read()
    plain_file.close()

    encrypted_text = cipher_suite.encrypt(plain_text)
    encrypted_file = open(filename+".aes", "wb+")
    encrypted_file.write(encrypted_text)
    encrypted_file.close()

def decryptt(filename, key):
    cipher_suite = Fernet(key)

    ’’’
    with open(os.path.join("EncryptedFiles",filename), ’rb’) as encrypted_file
        :
        encrypted_text = encrypted_file.read()
        encrypted_file.close()
    ’’’
    encrypted_file = open(filename+".aes", "rb")
    encrypted_text = encrypted_file.read()
    encrypted_file.close()

    decrypted_text = cipher_suite.decrypt(encrypted_text)
    if ’/’ in filename:
        index = filename.rfind('/')
        format_name = filename[0:index]
        format_name = format_name + "/decrypted_"
        format_name = format_name + filename[index+1:]
        decrypted_file = open(format_name, "wb+")
    else:
        decrypted_file = open("decrypted_"+filename, "wb+")
    decrypted_file.write(decrypted_text)
    decrypted_file.close()

def main():
    key = generate_key()
    encrypt("testfiles/COPYLIST.txt",key)
    decrypt("testfiles/COPYLIST.txt",key)

if __name__ == '__main__':
    main()
```

Drive Manager

Implementation

The Drive Manager is a simple module that provides simple functionalities such as encrypting, decrypting, listing and deleting all the files in the corresponding folder. This module is built upon the Encryptor.

Code:

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
Created on Sat Apr  6 21:21:28 2019

@author: Adamlkl
"""
import Encryptor
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive

# encrypt all the files in the drive folder usng key passed
def encrypt_all_files(key, folder):
    for file1 in folder:
        plain_text = file1.GetContentString()
        encrypted_text = Encryptor.encrypt(plain_text.encode(), key)
        file1.SetContentString(encrypted_text.decode())
        file1.Upload()
        print(encrypted_text)

# decrypt all the files in the drive folder usng key passed
def decrypt_all_files(key, folder):
    for file1 in folder:
        print('title: %s, id: %s' % (file1['title'], file1['id']))
        encrypted_text = file1.GetContentString()
        decrypted_text = Encryptor.decrypt(encrypted_text.encode(), key)
        file1.SetContentString(decrypted_text.decode())
        file1.Upload()
        print(decrypted_text)

# list all files in the drive folder
def list_all_files(folder):
    for file1 in folder:
        print('title: %s, id: %s' % (file1['title'], file1['id']))

# clean the drive folder by deleting all files in it
def delete_all_files(folder):
    for file1 in folder:
        print('Deleting file... title: %s, id: %s' % (file1['title'], file1['
            id']))
        file1.Delete()

def main():
    gauth = GoogleAuth()
    gauth.LocalWebserverAuth() # Creates local webserver and auto handles
        authentication.
    # Create GoogleDrive instance with authenticated GoogleAuth instance.
    drive = GoogleDrive(gauth)

    key = Encryptor.generate_key()
    folder = drive.ListFile({'q': "'17oua44SP5sR6E_g_h3a9Ua5qjHqAFvFy' in
        parents and trashed=false"}).GetList()

    #testing if I can get the encryption and decryption properly
    encrypt_all_files(key, folder)
    decrypt_all_files(key, folder)
    list_all_files(folder)
    delete_all_files(folder)

if __name__ == '__main__':
    main()
```

Key Manager

Implementation

The Key Saver is a module responsible for key management. It takes advantage of the subroutines available in the Cryptography libraries to implement its functions:

- saving private key by converting it into bytes form using serialization and writing the contents onto a file.
- load private key by reading the byte form contents of the passed file.
- Extracting the public part of a key in bytes.
- Generate a symmetric key using SHA-256, a cryptographic secure hash algorithm

Code:

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
Created on Fri Mar 29 15:51:42 2019

@author: Adamlkl
"""
'''
    - https://www.datacamp.com/community/tutorials/pickle-python-tutorial#what
    - https://security.stackexchange.com/questions/12332/where-to-store-a-
       server-side-encryption-key
    - https://pypi.org/project/pyAesCrypt/
'''
import os
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes

def save_key(username, private_key):
    '''
    pem = private_key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.PKCS8,
            encryption_algorithm=serialization.BestAvailableEncryption(b'
               mypassword')
    )
    '''
    pem = private_key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.PKCS8,
            encryption_algorithm=serialization.NoEncryption()
    )
    with open(os.path.join("Users",username), 'wb') as key_file:
        key_file.write(pem)
```

```
def load_key(username):
    with open(os.path.join("Users",username), ’rb’) as key_file:
        private_key = serialization.load_pem_private_key(
            key_file.read(),
            password=None,
            backend=default_backend()
        )
    return private_key

# serialize public key
def serialize_key(private_key):
    public_key = private_key.public_key()
    pem = public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo
        )
    return pem

def generate_symmetric_key(private_key, encryption_key):
    symkey = private_key.decrypt(
                encryption_key,
                padding.OAEP(
                    mgf=padding.MGF1(algorithm=hashes.SHA256()),
                    algorithm=hashes.SHA256(),
                    label=None
                )
            )
    return symkey

def load_public_key(key):
    return serialization.load_pem_public_key(key,backend=default_backend())
```

Improvements:

In long run, a graphical user interface version of the application can be developed to provide more comprehensive usage and systematic interactive usage to users.