CSU34031 – ADVANCED TELECOMMUNICATIONS

ASSIGNMENT 2 – SECURING THE CLOUD

DOCUMENTATION

Name: Claire Cassidy

Student Number: 16325301

CONTENTS

PROBLEM STATEMENT:

The goal of this assignment was to implement a secure cloud storage application for a cloud storage service of our choice. For my implementation, I chose to use Google Drive. The application should provide users of the associated 'secure cloud storage group' the ability to upload encrypted files to the service, as well as download any encrypted file another cloud storage group member has uploaded and be able to read it in plain-text form. To anyone else viewing the file, the file should appear to be encrypted and its contents must not be salvageable to anyone not registered with the group.

In addition, the solution for representing the secure cloud storage group should provide the ability to add and remove users from the group. While encryption alone can prevent snoopers on the network from reading the contents of communications, the file contents must be further protected from attacks through the use of public key certificates, which are used to verify the identity of the sender of a communication.

In my instance, I interpreted the scenario as being something like the following: an organisation shares a communal Google Drive account, but certain departments deal with confidential documents that should only be accessible to those in that department. Thus, the 'cloud group' for a user is defined to be a group to be those users registered with the CAM, and would define members of that department All other employees with access to the Google Drive account should only see encrypted files for any files uploaded by employees from that department.

THEORY

Communication between users on a network involves, at its core, the transmission of bytes across a medium from one endpoint to another. Modern networking can facilitate communications between distant users via the internet, and as such, messages between users pass along any number of intermediate networks on their way to their destination. This means that there is ample opportunity for communications to be intercepted and read by a malicious third party, or modified along the way by some middleman to either obscure the true identity of the sender or simply change the meaning of the message being sent. It is for these reasons that telecommunications must be secured using a suite of cryptographic methods, to ensure the integrity and confidentiality of the transmission.

The confidentiality of message can be achieved by encrypting it in such a way that only the sender and receiver know the precise series of functions to apply to the encrypted message (*ciphertext*) to retrieve the original, unencrypted message (*plaintext*). In my implementation, I used both symmetric and asymmetric key encryption. Symmetric key encryption is achieved by sharing the same cryptographic key among all users involved in a communication. This key must be kept secret from outside parties. This method of encryption has the downside that the key must be transferred to all parties by secure means, oftentimes physically, due to the inherent risks in sharing the key itself across and unsecure network. Asymmetric key encryption involves a pair of keys for each party in the communication; a public key and a private key. The keys are generated in such a way that the holder can encrypt a message with one key and decrypt it with the other. In practice, one of these keys is made publicly available and one is kept private to the party generating the keys (hence *public* and *private* keys). Another party may encrypt a message with this public key, knowing that only the only person who can undo this encryption is the party with the corresponding private key.

Confidentiality isn't enough to guarantee secure communication, however, since the message may be intercepted by a middleman and be tampered with. To overcome this, *message digests* are used, which is a hash of the bytes of a message along with the private key of the sender (which is only linked to one user in the network as it is private), and can be undone and verified by using the associated users public key, which is known to the recipient. In this way, the integrity of the message and its sender can be verified. This is also called *signing*, and is the equivalent of a *public key certificate* in contexts where it is not appropriate to go through a Certificate

Authority to obtain a formal certificate. This was achieved in my implementation by pairing asymmetric encryption via RSA with SHA256 hashing and PSS for padding.

DESIGN AND SETUP

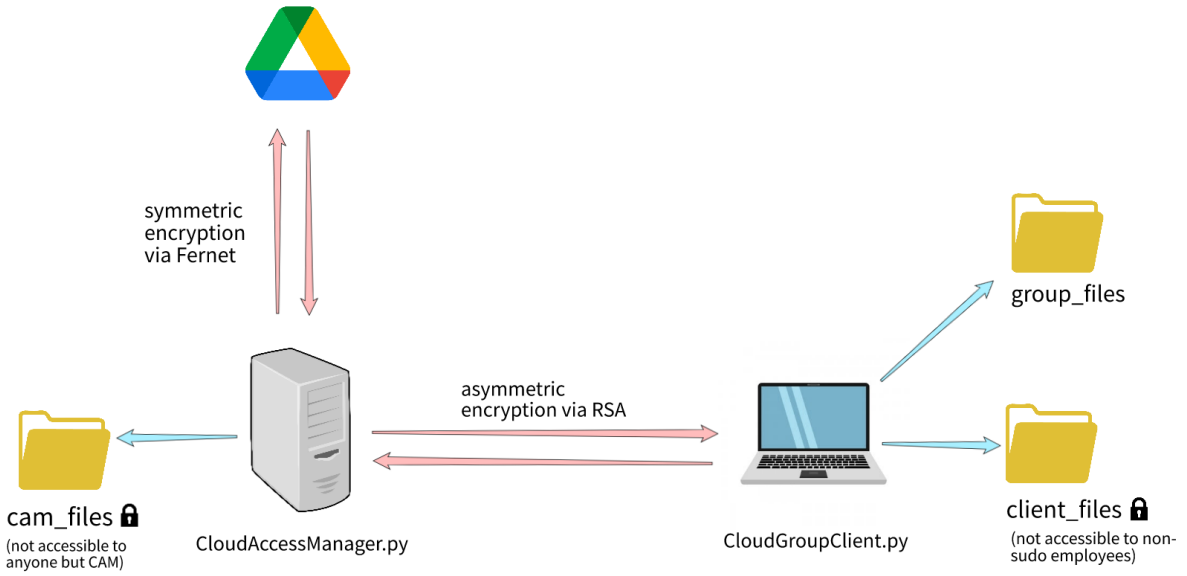
SOURCE FILES:

My implementation was completed in Python and consists of two scripts; `CloudAccessManager.py` and `CloudGroupClient.py`, henceforth referred to as '*CAM*' and '*client*' respectively. Both scripts are set up to communicate via sockets, and the intention is that CAM would be placed on some central server and access to its related folders would not be available to employees, even those in the cloud group. The CAM is the gatekeeper for access to the Google Drive account for those in the cloud group. The Client would serve as the interface through which members of the secure cloud storage group can interface with the CAM, and would transmit requests for file uploads and downloads from some local machine to the server on which the CAM is located.

ENCRYPTION & KEY MANAGEMENT:

As stated, my implementation uses both symmetric and asymmetric encryption. The communication channel between the CAM and Cloud is encrypted symmetrically (using a Python library called Fernet), whereas the communication channel between the CAM and Client is asymmetric (using Python's cryptography module; specifically the sections pertaining to RSA). The cloud channel can safely be encrypted using symmetric keys as no key exchange needs to take place, since the cloud endpoint never actually decrypts what it receives; just uploads the encrypted files as they are. The client and CAM interface via RSA, exchanging plaintext public keys on their initial communication which are saved locally for the next time they interact. The asymmetric communication is has its integrity preserved via the use of signing and verification, which allow the recipient to verify the identity of the sender.

This approach leads to a very simple key management system that means if the confidentiality of the local cam_files and client_files folders are respected keys never have to be refreshed. The symmetric key used for actually encrypting the file uploaded to Google Drive is never exposed to anyone and is only every visible to the CAM. The CAM itself can only be interfaced with by a cloud group user/admin via the client. The CAM and Client use their associated public and private keys to pass data, and the end user of the client never sees or interacts with the client's private key. This means that even if a user leaves the group, the integrity of the files on the cloud will not be compromised as this user never had access to the key used to encrypt the end file. If the worst case scenario occurs and it is suspected that a user managed to access the client_files folder, the only communication channel that would be compromised is the one between the client and CAM, and can be fixed by simply refreshing the client's keys, leaving those files encrypted on the cloud untouched. The following diagram shows the flow of data:



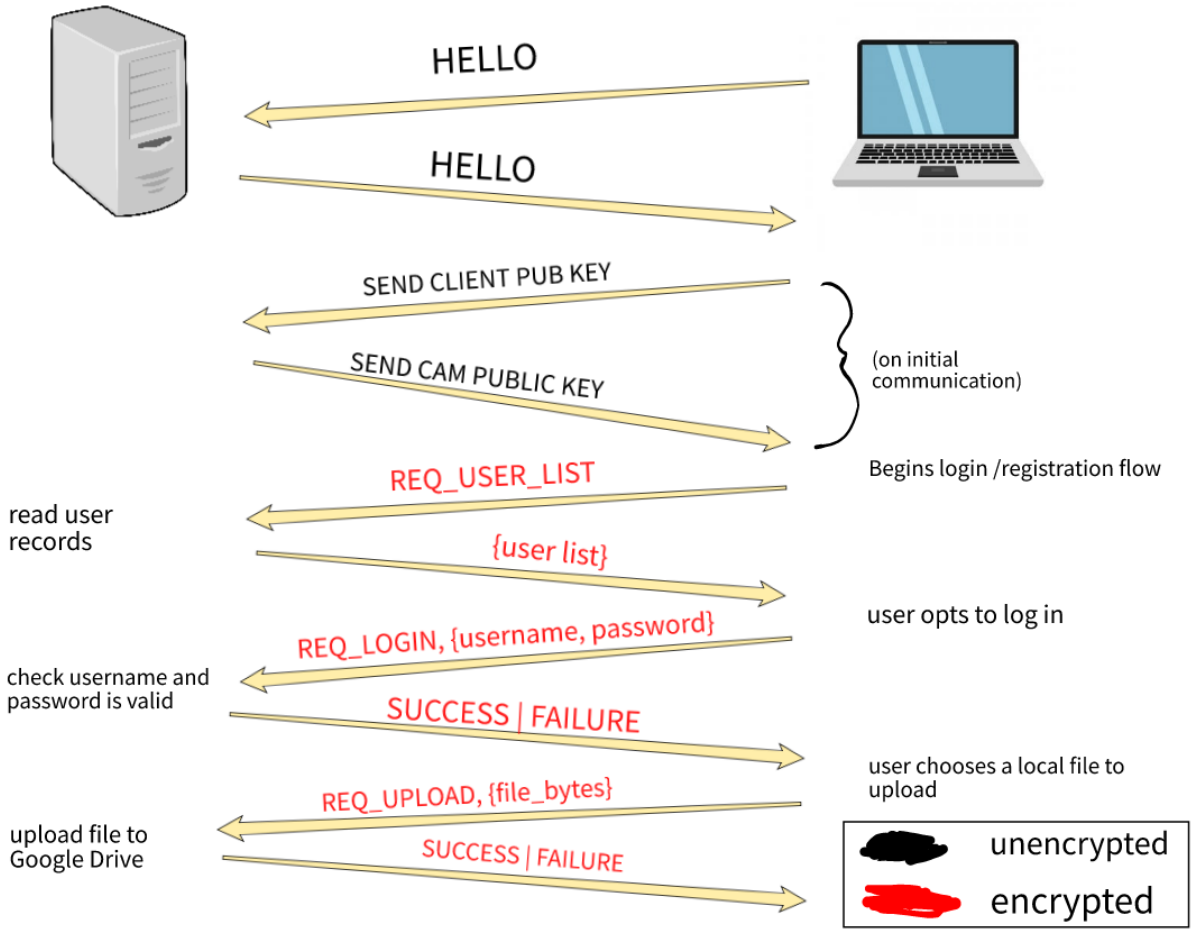
The persistent data for the program is organised in the above manner. group_files is the folder that the users of the client can access directly to place files for upload and retrieve files that have been downloaded. Each user has their own subdirectory in this folder under their username. client_files contains information for the CloudGroupClient.py script, specifically the public and private keys for the client and the public key of the CAM. This folder shouldn't be accessed by any employees, as it contains the private key for communication between the client and CAM. cam_files is used to hold keys, Google Drive authentication information, information about registered users and a stage folder which is used to temporarily hold files as they are uploaded to the cloud. This folder is only intended to be accessible to the CAM.

PACKAGES USED:

RSA: https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/

Symmetric Key Encryption: https://cryptography.io/en/latest/fernet/

CAM-CLIENT COMMUNICATION PROTOCOL:



The CAM and Client interact via sockets, and use a communication protocol based on requests and responses. There are many request types illustrated in the protocol to cover the problem statement, and are listed as constants at the top of both files. The protocol allows the CAM to interpret what is being asked of it, what it expects to receive next, and respond appropriately to requests.

Initially, communication begins with a HELLO message exchange. If this is the first time the CAM and client are communicating, a public key exchange takes place. This is unencrypted but unproblematic because the public key is assumed to be publicly available. Once keys have been exchanged all further communication is encrypted.

After contact has been established, the user list is sent to assist the client in servicing log-in/registration requests. The client enters a loop of servicing user requests, passing them on the CAM where appropriate. The CAM also enters a loop of receiving requests over the socket and handling them appropriately on its end. This

continues until the user exits the client, causing the socket connection to be closed and communication between the client and CAM to be terminated.

Cloud Storage Solution: Google Drive:

As noted, my implementation uses Google Drive as the cloud storage solution. It is through the associated API that requests to upload, download, and delete files from the service are made. You can read more about this API here: https://developers.google.com/drive/api/v3/about-sdk

User Management:

The problem statement specifies that there must be an ability to add and remove users from the secure cloud group. As the cloud group is defined to be those with accounts on the CAM, adding and removing users is equivalent to adding or removing their associated accounts.

Adding a user to the group is as simple as having them register an account through the client. CAM accounts come in two privilege levels, user and admin. When it is first launched, the CAM will deploy a default admin account, sysadmin, with the password adminpw. Only admins may remove user accounts. They may also demote other admins and promote users to admin privileges. It is recommended that, once deployed, the user registers a new user account, uses the default admin account to promote that user to admin privileges, and deletes the default admin, since its password is known.

Setup:

To setup, the user should create a folder with the two python files given in the appendix. The setup of all other requisite files and folders is done automatically.

Run CloudAccessManager.py to set up requisite folders. On the first run, it will complain that it can't find credentials.json required for authenticating access to Google Drive. Follow the instructions here to obtain the credentials.json file and save it in the folder ``cam_files/gdrive/`` that was just created on the initial run of `` CloudAccessManager.py``.

On the next run you will be prompted to connect your Google account to the app (this may require your email to be added to a list of authenticated users so please let me know if this is the case and I can add your email). Keep clicking proceed until you receive a message saying the authentication flow has been completed. This will save a token to the local Gdrive files so that you may proceed to interact remotely with the attached google account.

Navigate back to the console and the CAM will notify you that it is now listening for requests from the client. You may now launch the client class and begin communicating with the CAM.

IMPLEMENTATION:

A full code listing without breaks can be found in the Appendix

CLOUDACCESSMANAGER.PY

```
import io
import os.path

from cryptography.exceptions import InvalidSignature
from cryptography.hazmat.backends import default_backend
from google.oauth2.credentials import Credentials
from google.auth.transport.requests import Request
from google_auth_oauthlib.flow import InstalledAppFlow
from googleapiclient.discovery import build
from googleapiclient.http import MediaFileUpload, MediaIoBaseDownload

from multiprocessing.connection import Listener
from apiclient import errors

from cryptography.fernet import Fernet
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.primitives.asymmetric import rsa, padding

# What GDrive permissions we're requiring:
SCOPES = ['https://www.googleapis.com/auth/drive']
drive_service = None

# Constants
REGISTER = 0
LOG_IN = 1

CLOUD = 0
CLIENT = 1

ENCRYPT = 0
DECRYPT = 1

AS_BYTES = 0
AS_STR = 1

USER = 0
ADMIN = 1

USERS_PATH = r"cam_files\users"
ADMINS_PATH = r"cam_files\admins"

# Communication protocol between CAM and Client:
HELLO = "hello"
REQ_USER_LIST = "userlist"
REQ_REGISTER = "register"
REQ_CLOSE = "close"
REQ_LOGIN = "login"
REQ_DOWNLOAD = "download"
REQ_UPLOAD = "upload"
REQ_CLOUD_FILES = "files"
```

```
REQ_DEL_USER = "deluser"
REQ_MK_ADMIN = "admin"
REQ_RM_ADMIN = "demote"
REQ_DEL_FILE = "delfile"
OK = "ok"
SUCCESS = "success"  # request completed successfully
FAILURE = "failure"  # something went wrong

# dynamic data structure for keeping a list of usernames in memory
user_usernames = []
admin_usernames = []
cloud_filenames = {}

symmetric_key_cloud = None
public_key_client = None
private_key = None
```

There are many constants required for the program, particularly to distinguish between intended use cases of functions and for defining the communication protocol between the CAM and the client. In addition, local data structures to hold data found by analysing files are loaded for efficiency. Global variables are declared for the appropriate keys so they can be accessed easily throughout the program.

```
def main():
    global user_usernames, admin_usernames, \
        symmetric_key_cloud, drive_service, cloud_filenames, public_key_client,
private_key

    # perform one-time directory setups
    perform_dir_setup()

    # authorise self to upload/download from associated GDrive account
    drive_service = perform_cloud_auth()

    # perform one-time public/private key generation or load existing keys
    load_asymm_keys()

    # get the Fernet key for communication between program and cloud
    symmetric_key_cloud = Fernet(load_key(CLOUD))

    # initialise list of usernames (one time file-read)
    user_usernames, admin_usernames = load_usernames(USERS_PATH, ADMINS_PATH)
    # initialise list of gdrive files
    load_cloud_file_list()

    print(f'CloudAccessManager ready to service requests ...')

    # listen for communication from cloud group client
    address = ('localhost', 6000)
    listener = Listener(address, authkey=b'cloud_group')

    # accept connection via socket
    conn = listener.accept()
    print(f'connection accepted from {listener.last_accepted}')

    # client initiate comms with a plaintext HELLO exchange:

    proceed = True
```

```
    msg = conn.recv()
    if msg == HELLO:

        # respond with plaintext hello
        conn.send(HELLO)

        # If first time communicating with client, exchange pub keys in plaintext
        cur_dir = os.path.dirname(os.path.realpath(__file__))
        path_to_client_pubkey = os.path.join(cur_dir,
f'cam_files\\keys\\client_pubkey.pem')

        if not os.path.exists(path_to_client_pubkey):
            perform_key_exchange(conn, cur_dir, path_to_client_pubkey)

        # further comms will be encrypted, so load client's pub key
        load_client_pub_key()

    else:
        proceed = False
        conn.close()  # terminate connection; protocol not being followed

    if proceed is True:

        while True:  # repeatedly service requests from client until they close

            # keys exchanged, can now engage in encrypted comms
            close = False

            while close is False:

                req = decrypt_from_src(conn, AS_STR)

                if req == REQ_USER_LIST:
                    send_user_list(conn)
                elif req == REQ_REGISTER:
                    process_registration(conn)
                elif req == REQ_LOGIN:
                    process_login(conn)
                elif req == REQ_DOWNLOAD:
                    process_download(conn)
                elif req == REQ_CLOUD_FILES:
                    send_file_list(conn)
                elif req == REQ_UPLOAD:
                    handle_upload(conn)
                elif req == REQ_DEL_USER:
                    handle_user_deletion(conn)
                elif req == REQ_MK_ADMIN:
                    handle_user_promotion(conn)
                elif req == REQ_RM_ADMIN:
                    handle_admin_demotion(conn)
                elif req == REQ_DEL_FILE:
                    handle_delete_file(conn)
                elif req == REQ_CLOSE:

                    encrypt_and_send(conn, OK)
                    print(f'Closing connection ... ')
                    close = True
```

```
            conn.close()
            break

    listener.close()
```

The main function begins by checking for one-time setups that need to be performed, such as generating directories or authenticating with the cloud. The keys are loaded from persistent storage and the local data structures are populated. Once this setup has been performed, the CAM creates a Listener object, which is based on sockets, and binds it to port 6000. Once a client attempts to connect, the Listener accepts the connection and returns a conn object through which data is exchanged. The communication protocol is then followed, where each end says HELLO, followed by public key exchanges if they haven't already been exchanged. Then, all further communication is encrypted asymmetrically, and the CAM enters a loop in which it receives requests based on the defined protocol and calls the appropriate handlers to deal with them. When the client wants to close the connection, it sends REQ_CLOSE and this closes the socket, breaks the loop and exits the program.

```
def perform_key_exchange(conn, cur_dir, save_path):
    # receive public key
    client_pubkey = conn.recv()
    print(client_pubkey)

    # store key
    with open(save_path, 'wb') as file:
        file.write(client_pubkey)
        file.close()

    # respond with own pubkey
    path_to_cam_pubkey = os.path.join(cur_dir, f'cam_files\\keys\\public_key.pem')

    with open(path_to_cam_pubkey, 'rb') as file:
        file_bytes = file.read()
        conn.send(file_bytes)
```

The key exchange is defined as the client sending its public key and the CAM responding with its public key. Both sides write the key they receive for future communications.

```
def load_asymm_keys():
    global private_key

    cur_dir = os.path.dirname(os.path.realpath(__file__))
    path_to_keys = os.path.join(cur_dir, f'cam_files\\keys')

    if not os.path.exists(path_to_keys):    # need to generate them
        print(f'Performing one-time key generation ... ')
        # make it
        os.mkdir(path_to_keys)

        # generate public and private keys
        cam_priv_key = rsa.generate_private_key(public_exponent=65537,
key_size=4096)
        cam_public_key = cam_priv_key.public_key()

        # store the keys
        pem_priv = cam_priv_key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.PKCS8,
            encryption_algorithm=serialization.NoEncryption()
        )
```

```
        # save local variable for priv key
        private_key = pem_priv

        save_loc = os.path.join(path_to_keys, 'private_key.pem')
        with open(save_loc, 'wb') as pem_file:
            pem_file.write(pem_priv)

        pem_pub = cam_public_key.public_bytes(encoding=serialization.Encoding.PEM,

format=serialization.PublicFormat.SubjectPublicKeyInfo)

        save_loc = os.path.join(path_to_keys, 'public_key.pem')
        with open(save_loc, 'wb') as pem_file:
            pem_file.write(pem_pub)
    else:
        print(f'Loading asymm keys ... ')

        path_to_priv_key = os.path.join(path_to_keys, f'private_key.pem')

        # retrieve private key
        with open(path_to_priv_key, "rb") as key_file:
            private_key = serialization.load_pem_private_key(
                key_file.read(),
                password=None,
                backend=default_backend()
            )
```

This function is called during the setup phase to load the requisite keys from the disk into memory. It first checks whether those keys exists and if not generates a new public and private key for the CAM. The CAM only needs to have access to its own private key after the key exchange takes place, so that is the only key that is loaded if the keys are found to already exist.

```
def load_client_pub_key():
    global public_key_client

    cur_dir = os.path.dirname(os.path.realpath(__file__))
    path_to_client_pubkey = os.path.join(cur_dir,
f'cam_files\\keys\\client_pubkey.pem')

    with open(path_to_client_pubkey, "rb") as key_file:
        public_key_client = serialization.load_pem_public_key(
            key_file.read(),
            backend=default_backend()
        )
```

Simple helper function to load the client's public key from memory. Will only be called after the key exchange has taken place.

```
def perform_cloud_auth():
    # Load access token or creates one if DNE
    creds = None

    # Load credentials if they exist (i.e. the authorisation set up has been run
already)
    if os.path.exists('cam_files\\gdrive\\token.json'):
```

```
        creds =
Credentials.from_authorized_user_file('cam_files\\gdrive\\token.json', SCOPES)

    # If they don't exist, create
    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            creds.refresh(Request())
        else:
            flow = InstalledAppFlow.from_client_secrets_file(
                'cam_files\\gdrive\\credentials.json', SCOPES)
            creds = flow.run_local_server(port=0)
        # Save the credentials for the next run
        with open('cam_files\\gdrive\\token.json', 'w') as token:
            token.write(creds.to_json())

    # return reference to drive service object that will handle upload/download
requests
    return build('drive', 'v3', credentials=creds)
```

This function is used to perform first time authentication of the associated Google Drive account using the credentials.json file discussed in the setup. This process will cause a token.json file to be written which is used on future sessions to authenticate the user without them having to authorise again. Finally, it returns an object representing the Drive service through which calls to upload, download, and delete files are made.

```
def load_key(type):
    cwd = os.getcwd()
    key = None

    # Either fetch saved fernet key from previous session or generate if DNE
    if type == CLOUD:
        path_to_key = f'{cwd}\\cam_files\\fernet_cloud.key'
    elif type == CLIENT:
        path_to_key = f'{cwd}\\cam_files\\fernet_client.key'

    if os.path.exists(path_to_key) is False:  # key dne yet; create
        # generate symmetric key for communication with cloud/client
        print(f'Writing New {"Cloud" if type == CLOUD else "Client"} Symmetric
Key ...')
        key = Fernet.generate_key()

        # save the key
        with open(path_to_key, 'wb') as key_file:
            key_file.write(key)
            key_file.close()

        # Save an additional copy for the client to use
        if type == CLIENT:
            path_to_client = f'{cwd}\\client_files\\fernet_client.key'

            # save the key
            with open(path_to_client, 'wb') as key_file:
                key_file.write(key)
                key_file.close()

    if key is None:  # will be none if it already existed; load
        # load symmetric key
        with open(path_to_key, 'rb') as key_file:
            key = key_file.read()
```

```
    return key
```

This function is used to generate a symmetric key for the cloud if it does not exist, and load the symmetric key for the cloud. There is some extra code here relating to a client symmetric key which was from an earlier iteration where all communications were done via symmetric key encryption. A clear downside of this is seen where the encapsulation between client_files and cam_files needed to be broken in order to essentially exchange the symmetric key with the client by saving it to the client_files folder. This is no longer the case with the use of asymmetric key encryption.

```
# fetch list of usernames for dynamic storage
def load_usernames(users_path, admins_path):
    users = []
    admins = []

    cur_dir = os.path.dirname(os.path.realpath(__file__))

    full_path_users = os.path.join(cur_dir, users_path)
    users = [os.path.splitext(file)[0] for file in os.listdir(full_path_users)]

    full_path_admins = os.path.join(cur_dir, admins_path)
    admins = [os.path.splitext(file)[0] for file in os.listdir(full_path_admins)]

    return users, admins
```

Helps load the dynamic list of usernames from the cam_files system to limit the number of file reads. This is used to authenticate logins and registrations.

```
# get a list of files from the google drive service
def load_cloud_file_list():
    global cloud_filenames

    results = drive_service.files().list().execute()
    items = results.get('files', [])

    if not items:
        print('No files found.')
    else:
        print('Files:')
        for item in items:
            print(u'{0} ({1})'.format(item['name'], item['id']))

    # List of filename : id pairs
    for item in items:
        cloud_filenames[item['name']] = item['id']

    print(cloud_filenames)
```

Again, a one time call to the Google Drive service to efficiently populate a dynamic list of files on the cloud. This is used to authenticate requests for uploads, downloads and deletions.

```
def send_user_list(conn):

    encoded_list = " ".join(user_usernames)
    encoded_list += "|"
    encoded_list += " ".join(admin_usernames)
```

```
    encrypt_and_send(conn, encoded_list)
```

Sends the dynamic list of users to the client so they can perform client-side authentications of registrations/logins, for example, checking that a username is not already taken without clogging the network with requests to CAM.

```
def process_registration(conn):
    encrypt_and_send(conn, OK)

    registration_details = decrypt_from_src(conn, AS_STR)  # in form
"username|password"
    registration_details = registration_details.split("|")
    username = registration_details[0]
    password = registration_details[1]
    print(f'Preparing to register user \'{username}\' ... ')

    # write the new user details to CAM files:
    cur_dir = os.path.dirname(os.path.realpath(__file__))
    full_path_users = os.path.join(cur_dir, USERS_PATH)
    path_to_new_file = os.path.join(full_path_users, f'{username}.txt')

    with open(path_to_new_file, 'w') as new_user_file:
        new_user_file.write(f'{password}')
        new_user_file.close()

    # add to dynamic data structure representing usernames:
    user_usernames.append(username)

    # conn.send(SUCCESS)
    encrypt_and_send(conn, SUCCESS)
    print(f'Successfully created registration record for \'{username}\'')
```

Accepts a username and password and creates the user record in cam_files. User records are saved with the username as the filename and their password as the contents.

```
def handle_user_promotion(conn):
    global user_usernames, admin_usernames

    encrypt_and_send(conn, OK)
    username = decrypt_from_src(conn, AS_STR)

    cur_dir = os.path.dirname(os.path.realpath(__file__))
    # get cur path to user record:
    full_path_to_user_record = os.path.join(cur_dir,
f'cam_files\\users\\{username}.txt')

    # get dest path:
    dest_path = os.path.join(cur_dir, f'cam_files\\admins\\{username}.txt')

    # move the record
    os.rename(full_path_to_user_record, dest_path)

    # update dynamic data structures holding usernames
    user_usernames.remove(username)
    admin_usernames.append(username)
```

```
    encrypt_and_send(conn, SUCCESS)
```

Promotes a user to admin status by moving its record into the admins folder and updating the dynamic data structure representing users and admins.

```
def handle_admin_demotion(conn):
    global user_usernames, admin_usernames

    encrypt_and_send(conn, OK)
    username = decrypt_from_src(conn, AS_STR)

    cur_dir = os.path.dirname(os.path.realpath(__file__))
    # get cur path to admin record:
    full_path_to_admin_record = os.path.join(cur_dir,
f'cam_files\\admins\\{username}.txt')

    # get dest path:
    dest_path = os.path.join(cur_dir, f'cam_files\\users\\{username}.txt')

    # move the record
    os.rename(full_path_to_admin_record, dest_path)

    # update dynamic data structures holding usernames
    admin_usernames.remove(username)
    user_usernames.append(username)

    encrypt_and_send(conn, SUCCESS)
```

Same principle as previous function, but demotes and admin to a user.

```
def process_login(conn):
    # acknowledge login request
    encrypt_and_send(conn, OK)

    login_details = decrypt_from_src(conn, AS_STR)  # in form
'[username]|[password]'
    login_details = login_details.split("|")
    print(f'Login Details: {login_details}')
    username = login_details[0]
    password = login_details[1]

    user_type = None
    print(f'Users: {user_usernames}')
    print(f'Admins: {admin_usernames}')

    if username in user_usernames:
        user_type = USER
    elif username in admin_usernames:
        user_type = ADMIN

    if user_type is not None:

        # get user's password from storage
        cur_dir = os.path.dirname(os.path.realpath(__file__))

        if user_type == USER:
            full_path = os.path.join(cur_dir, USERS_PATH)
        else:  # admin
            full_path = os.path.join(cur_dir, ADMINS_PATH)
```

```
        path_user_record = os.path.join(full_path, f'{username}.txt')
        print(path_user_record)

        with open(path_user_record, 'rb') as user_file:
            # receive string representation from saved bytes
            true_password = ''.join(user_file.readline().decode('utf-8').split())

        # if passed pw matches registration pw
        if password == true_password:
            encrypt_and_send(conn, SUCCESS)
        else:
            encrypt_and_send(conn, FAILURE)

    else:
        # user dne
        encrypt_and_send(conn, FAILURE)
```

Performs validation on a login by checking the passed username and password against the user record. If the user record exists (either in admins or users) and the password matches the saved password, the login is deemed a success. Otherwise it is a failure and the client blocks access.

```
def process_download(conn):
    global cloud_filenames

    encrypt_and_send(conn, OK)

    target_file = decrypt_from_src(conn, AS_STR)
    print(target_file)
    print(cloud_filenames)

    if target_file in cloud_filenames:
        # query Gdrive for file
        request =
drive_service.files().get_media(fileId=cloud_filenames[target_file])

        file_handler = io.BytesIO()
        downloader = MediaIoBaseDownload(file_handler, request)
        done = False
        while done is False:
            status, done = downloader.next_chunk()
            print(f'Fetching ... {int(status.progress() * 100)}%.')

        # file bytes in file_handler; will have been encrypted with cloud symm key
        #   decrypt; re-encrypt with client symm key and send
        decrypted_bytes = symmetric_key_cloud.decrypt(file_handler.getvalue())
        encrypt_and_send(conn, decrypted_bytes)

    else:
        encrypt_and_send(conn, FAILURE)
```

Accepts a filename, verifies its existence in the cloud and queries the drive_service object for the file. The file retrieved from the cloud will be encrypted with the cloud_public_key, so we decrypt it first to plaintext. Then we re-encrypt it using the client's public key as we send so that the user gets the plaintext file once they apply their private key.

```
def encrypt_and_send(conn, msg):
    global public_key_client, private_key
```

```
    # convert msg to bytes
    if isinstance(msg, str):
        msg = str.encode(msg)

    # encrypt with client's public key
    ciphertext = public_key_client.encrypt(
        msg,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    # generate the message digest
    signature = private_key.sign(
        ciphertext,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )

    # print(f'\tSending {msg}; ciphertext: {ciphertext}')
    conn.send(ciphertext)

    # print(f'\t\tSending signature: {signature}')
    conn.send(signature)
```

This function is called any time a message is sent to the client (apart from during the plaintext key exchange). This function is what implements the asymmetric RSA encryption, by encrypting it with the client's public key. The client waits for two messages whenever it is expecting a response. First the encrypted message is sent, followed by the message digest, here called *signature*. The first message is just the encrypted text and the second is used to verify the integrity of the message on the client's side.

```
# decrypts a message from the client encrypted using CAM's pub key by using the
CAM's private key
def decrypt_from_src(conn, as_what):
    global private_key, public_key_client

    ciphertext = conn.recv()
    signature = conn.recv()

    try:
        verified_ciphertext = public_key_client.verify(
            signature,
            ciphertext,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )

        # will throw exception if signature not valid
        #   otherwise decrypt
```

```
        plaintext = private_key.decrypt(
            ciphertext,
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )

        # cast as string if requested
        if as_what == AS_STR:
            plaintext = plaintext.decode("utf-8")

        # print(f'\tReceived {plaintext}; ciphertext: {ciphertext}')
        return plaintext

    except InvalidSignature:
        print(f'Invalid signature on msg')

    return None
```

This function is called whenever the CAM is expecting a response from the client. Again, two messages are sent, the first of which being the encrypted message and the second being the message digest. First the message integrity is checked by verifying the digest. If ever this doesn't match, an exception is thrown and the encrypted message is unacknowledged. If no exception is thrown, we are safe to decrypt the message and return it to the caller. An additional parameter is passed to determine what form the caller function wants the message in – raw bytes (for files) or a string (for protocol constants).

```
def handle_delete_file(conn):
    global cloud_filenames

    encrypt_and_send(conn, OK)
    target_file = decrypt_from_src(conn, AS_STR)

    # get associated GDrive id:
    file_id = cloud_filenames[target_file]

    delete_success = False
    # call on drive service to delete the file:
    try:
        drive_service.files().delete(fileId=file_id).execute()
        delete_success = True
    except errors.HttpError as error:
        print(f'ERROR OCCURRED DELETING FILE {target_file}:{file_id}\n\t{error}')

    if delete_success:
        # remove from local data structure
        cloud_filenames.pop(target_file)

        # report success to client
        encrypt_and_send(conn, SUCCESS)
    else:
        encrypt_and_send(conn, FAILURE)
```

Receives the name of a target file to delete, checks its existence in the cloud, and if it exists, uses the ID Google Drive associates with that file to delete it from the service.

```
def send_file_list(conn):
    global cloud_filenames

    encoded_file_list = "|".join(cloud_filenames.keys())

    encrypt_and_send(conn, encoded_file_list)
```

Sends the list of files on the cloud to the client. Used for verification on their end before sending requests.

```
def handle_upload(conn):
    global drive_service

    encrypt_and_send(conn, OK)

    # response will be filename followed by filebytes
    file_name = decrypt_from_src(conn, AS_STR)
    encrypt_and_send(conn, OK)
    file_bytes = decrypt_from_src(conn, AS_BYTES)

    # encrypt the file using CAM's cloud symm key
    encrypted_file_bytes = symmetric_key_cloud.encrypt(file_bytes)

    # upload to cloud
    #   first need to save to staging area

    cur_dir = os.path.dirname(os.path.realpath(__file__))
    rel_path = f'cam_files\\stage\\{file_name}'
    path_to_stage_file = os.path.join(cur_dir, rel_path)

    with open(path_to_stage_file, 'wb') as stage_file:
        stage_file.write(encrypted_file_bytes)
        stage_file.close()

    # now upload it
    file_metadata = {'name': file_name}
    to_upload = MediaFileUpload(rel_path, resumable=True)
    file = drive_service.files().create(body=file_metadata,
                                        media_body=to_upload,
                                        fields='id').execute()
    to_upload = None

    # save uploaded file data to dynamic data structure
    cloud_filenames[file_name] = file.get('id')

    # remove file from staging area
    os.remove(rel_path)

    # tell client it was successful
    encrypt_and_send(conn, SUCCESS)
```

Takes a file name followed by the raw file bytes to upload. Google Drive's API needs a pointer to an actual file to work, so first the file is saved in the stage folder so it can be pointed to. Then it uses the drive_service object to upload the file, gets the file ID back and adds it to the dynamic list of files. Finally, the file is removed from the stage.

```
# one-time setup of cam_files directory structure
def perform_dir_setup():
    cur_dir = os.path.dirname(os.path.realpath(__file__))
```

```
    path_to_cam_files = os.path.join(cur_dir, f'cam_files')

    if not os.path.exists(path_to_cam_files):
        print(f'Creating CAM files @ {path_to_cam_files} ... ')
        os.mkdir(path_to_cam_files)

        path_to_stage = os.path.join(path_to_cam_files, f'stage')
        print(f'Setting up stage @ {path_to_stage} ... ')
        os.mkdir(path_to_stage)

        path_to_users = os.path.join(path_to_cam_files, f'users')
        print(f'Setting up user records @ {path_to_users} ... ')
        os.mkdir(path_to_users)

        path_to_admins = os.path.join(path_to_cam_files, f'admins')
        print(f'Setting up admin records @ {path_to_admins} ... ')
        os.mkdir(path_to_admins)

        # create default admin account
        path_to_default_admin = os.path.join(path_to_admins, f'sysadmin.txt')
        with open(path_to_default_admin, 'w') as default_admin:
            default_admin.write('adminpw')
            default_admin.close()

        path_to_gdrive = os.path.join(path_to_cam_files, f'cam_files/gdrive')
        print(f'Setting up GDrive folder @ {path_to_gdrive}')
        os.mkdir(path_to_gdrive)
        # make credentials.json here:
        with open(os.path.join(path_to_gdrive,
f'cam_files/gdrive/credentials.json'), 'w') as f:
            f.close()
```

Called on first-time startup to generate the necessary cam_files directory structure. Also creates default admin account.

```
def handle_user_deletion(conn):
    encrypt_and_send(conn, OK)

    username = decrypt_from_src(conn, AS_STR)

    rel_path = None
    if username in admin_usernames:
        # also delete record in users dynamic data structure:
        admin_usernames.remove(username)
        rel_path = os.path.join(ADMINS_PATH, f'{username}.txt')
    elif username in user_usernames:
        user_usernames.remove(username)
        rel_path = os.path.join(USERS_PATH, f'{username}.txt')
    else:  # user doesn't seem to exist :/
        encrypt_and_send(conn, FAILURE)

    if rel_path is not None:
        # delete user record file
        cur_dir = os.path.dirname(os.path.realpath(__file__))
        target_file = os.path.join(cur_dir, rel_path)

        os.remove(target_file)

        encrypt_and_send(conn, SUCCESS)
```

Deletes the given user record from the appropriate folder based on its privilege level and removes from related dynamic data structures.

CLOUDGROUPCLIENT.PY

```
import os
import re

from multiprocessing.connection import Client

from shutil import rmtree

from cryptography.exceptions import InvalidSignature
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.primitives.asymmetric import rsa, padding

USER_PRIVILEGE = 0
ADMIN_PRIVILEGE = 1

AS_BYTES = 0
AS_STR = 1

REGISTER = 'r'
LOG_IN = 'l'
QUIT = 'q'

PROTOCOL_EX = "Unexpected communication protocol error"

REGEX_VALID_USERNAME = '^[A-Za-z0-9_-]*$'
REGEX_VALID_PASSWORD = r'[A-Za-z0-9@#$%^&+=]{6,}'

# Communication protocol between CAM and Client:
HELLO = "hello"
REQ_USER_LIST = "userlist"
REQ_REGISTER = "register"
REQ_CLOSE = "close"
REQ_LOGIN = "login"
REQ_DOWNLOAD = "download"
REQ_UPLOAD = "upload"
REQ_CLOUD_FILES = "files"
REQ_DEL_USER = "deluser"
REQ_MK_ADMIN = "admin"
REQ_RM_ADMIN = "demote"
REQ_DEL_FILE = "delfile"
OK = "ok"
SUCCESS = "success"  # request completed successfully
FAILURE = "failure"  # something went wrong

# dynamic data structure for keeping a list of usernames in memory
users = []
admins = []
cloud_files = []
```

```
public_key_cam = None
private_key = None
```

Constants, globals and dynamic data structures created for almost identical purposes as in CAM.

```
def main():
    global users, admins, cloud_files, public_key_cam, private_key

    # perform one-time setup of client_files and keys where applicable
    perform_initial_setup()

    # connect to CAM
    address = ('localhost', 6000)
    conn = Client(address, authkey=b'cloud_group')

    # initialise communication with CAM
    #   send plaintext HELLO msg
    conn.send(HELLO)
    res = conn.recv()
    if res == HELLO:

        # if haven't got a record of CAM's pubkey, keys haven't been exchanged
yet.
        cur_dir = os.path.dirname(os.path.realpath(__file__))
        path_to_cam_pubkey = os.path.join(cur_dir,
f'client_files\\cam_pubkey.pem')

        if not os.path.exists(path_to_cam_pubkey):
            perform_key_exchange(conn, cur_dir, path_to_cam_pubkey)

        # now load keys
        load_keys()

        # begin registration/login flow:
        encrypt_and_send(conn, REQ_USER_LIST)
        user_list_string = decrypt_from_src(conn, AS_STR)
        users, admins = extract_usernames(user_list_string)

        # get filenames from cloud
        encrypt_and_send(conn, REQ_CLOUD_FILES)
        res = decrypt_from_src(conn, AS_STR)
        cloud_files = res.split("|")

        prompt_for_login(users, admins, conn)

        # ^ returns when user hits QUIT, so close connection with CAM and exit
        encrypt_and_send(conn, REQ_CLOSE)
    else:
        raise Exception(PROTOCOL_EX)
```

Performs the initial first-time setup steps where required and initiates communication with the CAM via its connection object. As per the protocol, it first sends HELLO to CAM, and performs a key exchange. Next, its private key is loaded and it begins its registration/login flow by asking for a list of users from the CAM. This is followed by asking for a list of filenames on the cloud. Both of these lists are used for validating requests entered by the user. Next it calls the prompt_for_login function, which loops asking for logins and servicing the requests

of users. When the user exits this menu they are considered to have logged out, and thus once this function returns, the client sends a final request to close communications with the CAM.

```
def load_keys():
    global private_key, public_key_cam

    # load client's private key for decryption
    cur_dir = os.path.dirname(os.path.realpath(__file__))
    path_to_priv_key = os.path.join(cur_dir, f'client_files\\private_key.pem')

    with open(path_to_priv_key, "rb") as key_file:
        private_key = serialization.load_pem_private_key(
            key_file.read(),
            password=None,
            backend=default_backend()
        )

    # load CAM's public key for encryption
    path_to_cam_pubkey = os.path.join(cur_dir, f'client_files\\cam_pubkey.pem')

    with open(path_to_cam_pubkey, "rb") as key_file:
        public_key_cam = serialization.load_pem_public_key(
            key_file.read(),
            backend=default_backend()
        )
```

This function loads the private key for the client and the public key of CAM. It is called only after key exchange has taken place.

```
def perform_key_exchange(conn, cur_dir, save_path):

    print(f'Performing one-time public key exchange ... ')
    # send client's pubkey to CAM
    path_to_client_pubkey = os.path.join(cur_dir, f'client_files\\public_key.pem')

    with open(path_to_client_pubkey, 'rb') as file:
        file_bytes = file.read()
        conn.send(file_bytes)

    # CAM responds with own PKey
    cam_pubkey = conn.recv()

    # save it
    with open(save_path, 'wb') as file:
        file.write(cam_pubkey)
        file.close()
```

Discussed in CAM.

```
def perform_initial_setup():
    cur_dir = os.path.dirname(os.path.realpath(__file__))
    path_client_files = os.path.join(cur_dir, f'client_files')

    if not os.path.exists(path_client_files):
        os.mkdir(path_client_files)
        print(f'Created \'{path_client_files}\'')
```

```
        print(f'Generating keys ...')
        # create client's asymm keys
        client_priv_key = rsa.generate_private_key(public_exponent=65537,
key_size=4096)
        client_public_key = client_priv_key.public_key()

        # store the keys
        pem_priv = client_priv_key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.PKCS8,
            encryption_algorithm=serialization.NoEncryption()
        )

        save_loc = os.path.join(path_client_files, 'private_key.pem')
        with open(save_loc, 'wb') as pem_file:
            pem_file.write(pem_priv)

        pem_pub =
client_public_key.public_bytes(encoding=serialization.Encoding.PEM,

format=serialization.PublicFormat.SubjectPublicKeyInfo)

        save_loc = os.path.join(path_client_files, 'public_key.pem')
        with open(save_loc, 'wb') as pem_file:
            pem_file.write(pem_pub)
```

Like with the CAM, the client performs some initial one-time directory setups before servicing any user requests. It is also within this function that the client's asymmetric keys will be generated and saved.

```
# decode user list received by CAM
def extract_usernames(user_list_string):

    # divide into users and admins
    split_list = user_list_string.split("|")
    user_usernames = split_list[0].split(" ")
    admin_usernames = split_list[1].split(" ")

    return user_usernames, admin_usernames
```

Parses out the string representation of users and admins lists sent by the client into two lists.

```
def prompt_for_login(users, admins, conn):
    login_success = False

    while not login_success:

        print('\n')

        privilege_level = None
        option = input(
            f'Welcome to SecuringTheCloud. [R]egister or [L]og in? ([Q] to
exit)\n-----------------\n').lower()

        if option == REGISTER:

            res = register_new_user(conn, users + admins)

        elif option == LOG_IN:
```

```
            handle_log_in(conn)

        elif option == QUIT:
            encrypt_and_send(conn, REQ_CLOSE)

            res = decrypt_from_src(conn, AS_STR)
            if res == OK:
                print(f'Goodbye!')
                break
            else:
                raise Exception(PROTOCOL_EX)

        else:
            print(f'Not a valid option. Please try again.')
```

The base menu for user interaction. Repeatedly loops to ask for an option to login, register or quit via stdin and calls the appropriate handler function.

```
def register_new_user(conn, exclusion_list):
    valid_username = False
    valid_password = False
    go_back = False

    username = None

    while valid_username is False:
        username = input(f'Preparing to register a new user... ([B] to go back)\n
Username: ').lower()

        if username == 'b':
            go_back = True
            break
        elif 6 <= len(username) <= 15 and re.match(REGEX_VALID_USERNAME,
username):
            if username in exclusion_list:
                print(f' Sorry, that username has been taken.\n')
            else:
                valid_username = True
        else:
            print(
                f' Please enter a username between 6 and 15 characters containing
only letters, numbers, -, and/or _\n')

    if go_back is False:
        while valid_password is False:
            password = input(' Password: ')

            if 6 <= len(password) <= 15 and re.match(REGEX_VALID_PASSWORD,
password):
                valid_password = True
            else:
                print(f' Please enter a password between 6 and 15 characters
containing only letters, numbers, '
                      f'and/or the following special characters: @ # $ % ^ & +
=\n')

        if username is not None:
            encrypt_and_send(conn, REQ_REGISTER)
```

```
            res = decrypt_from_src(conn, AS_STR)
            if res == OK:
                encrypt_and_send(conn, f'{username}|{password}')
                users.append(username)
            else:
                raise Exception(PROTOCOL_EX)

            res = decrypt_from_src(conn, AS_STR)
            if res == SUCCESS:
                validate_group_folder(username)
                print(f'Registered user \'{username}\' successfully. Please
proceed to log in.')

            return True
    else:
        raise Exception(PROTOCOL_EX)

    return False
```

Registers a new user by taking in a username and password, checking that they are valid entries and that the username is not already taken, and passes them on to the CAM to register. If its successful, creates a new local folder for the user's files in group_folders/<username>. Reports the success/failure of the operation to the user.

```
def handle_log_in(conn):
    go_back = False

    while go_back is False:

        print(f'\nEnter your login details ([B] to go back)')

        username = input(f' Username: ').lower()
        if username == 'b':
            break

        password = input(f' Password: ')

        # send to CAM to authenticate:
        # send request to log in
        encrypt_and_send(conn, REQ_LOGIN)

        # await OK from CAM
        res = decrypt_from_src(conn, AS_STR)
        if res == OK:
            # proceed to send login details for verification
            encrypt_and_send(conn, f'{username}|{password}')

            # response is either SUCCESS if verifiable or FAIL if not
            res = decrypt_from_src(conn, AS_STR)
            if res == SUCCESS:

                # successful login
                print(f'\nLogin Successful. Welcome {username}.')

                if username in users:
                    validate_group_folder(username)
                    handle_user(conn, username)
                elif username in admins:
                    validate_group_folder(username)
                    handle_admin(conn, username)
```

```
                else:
                    raise Exception(f'Something\'s gone terribly wrong :(')

            elif res == FAILURE:
                # unsuccessful login; repeat loop
                print(f'Login unsuccessful. Please try again.')
            else:
                raise Exception(PROTOCOL_EX)

        else:
            raise Exception(PROTOCOL_EX)
```

Takes in a username and password and sends to CAM to authenticate. If it is a success, it allows the user to log in, ensures their group_folders/<username>/uploads folder is created and then sends them to the appropriate interface based on whether they are an admin or a user. If it is unsuccessful, the input loops until the user successfully logs in or quits.

```
# creates \group_files\{username}\uploads if dne
def validate_group_folder(username):
    cur_dir = os.path.dirname(os.path.realpath(__file__))
    full_path = os.path.join(cur_dir, f'group_files\\{username}\\uploads')

    if not os.path.exists(full_path):
        os.makedirs(full_path)
```

Creates the user's group_folders/<username>/uploads folder is it does not exist. This is where the client will look for files to upload.

```
def handle_user(conn, username):
    global cloud_files

    keep_going = True

    while keep_going is True:

        valid_option = False

        while valid_option is False:
            option = input(f'Do you wish to [U]pload or [D]ownload a file? ([B]ack
to logout)\n').lower()

            if option == 'd':

                handle_download(conn, username)

            elif option == 'u':

                handle_upload(conn, username)

            elif option == 'b':

                print(f'Logging out ...')
                valid_option = True
                keep_going = False

            else:
                print(f'Not a valid option.')
```

Presents the menu options for a user account and calls the appropriate handler for the user's choice.

```
def handle_download(conn, username):
    valid_option = True

    has_downloaded_smth = False

    while has_downloaded_smth is False:

        option = input(
            f'Choose one of the following options:'
            f'\n\t[L]: List the files currently available on the cloud'
            f'\n\t[<filename.ext>]: Download a file'
            f'\n\t[B]: Return to previous menu\n')

        if option == 'l' or option == 'L':

            print(f'\nFiles available for download:')
            for file in cloud_files:
                print(f' {file}')
            print()

        elif option == 'b' or option == 'B':

            valid_option = True
            break

        else:  # file request

            if option in cloud_files:
                has_downloaded_smth = True

                # first notify that local file will be overwritten if required
                cur_dir = os.path.dirname(os.path.realpath(__file__))
                path_to_dl = os.path.join(cur_dir,
f'group_files\\{username}\\downloads\\{option}')

                proceed = True

                if os.path.exists(path_to_dl):  # i.e. already file in user's dl
folder with same name

                    valid_option = False

                    while valid_option is False:
                        option2 = input(f'Proceeding will overwrite local file
\'{option}\' in your downloads folder. '
                                        f'Proceed? [Y/N]\n').lower()

                        if option2 == 'y':
                            valid_option = True
                        elif option2 == 'n':
                            valid_option = True
                            proceed = False
                            print(f'Cancelling operation ... ')
                        else:
                            print(f'Not a recognised option. Please enter [Y/N]')

                if proceed is True:
```

```
                    result = request_download(conn, option, username)

            else:
                print(f'That file does not exist. Enter [L] to see a list of files
available to download.')
```

Called when a user opts to download something. Let's the user see a list of files on the cloud available for download and accepts a filename. It checks the user's downloads folder to see if a file with the same name exists and if so, prompts them to see if they are okay with it being overwritten. It then calls the request_download function to send the request to the CAM and save it.

```
def handle_upload(conn, username):

    valid_filename = False

    while valid_filename is False:

        # get filenames in user's uploads folder
        cur_dir = os.path.dirname(os.path.realpath(__file__))
        path_to_uploads = os.path.join(cur_dir,
f'group_files\\{username}\\uploads')

        file_names = None

        if os.path.exists(path_to_uploads):
            print(f'Files available for upload:')
            file_names = os.listdir(path_to_uploads)
            for file in file_names:
                print(f'\t{file}')
            print()
        else:
            raise Exception(f'Uploads folder doesn\'t exist for user {username}')

        file_name = input('\nPlease enter the name of the file you wish to upload.
([B]ack to return)\n')

        if file_name == 'b' or file_name == 'B':
            valid_filename = True
        elif file_name in file_names:

            proceed = True

            if file_name in cloud_files:
                valid_option = False

                while valid_option is False:
                    overwrite = input(f'File with this name already uploaded.
Overwrite? [Y/N]\n').lower()
                    if overwrite == 'y':
                        valid_option = True
                        proceed = True

                        # Delete the old file
                        encrypt_and_send(conn, REQ_DEL_FILE)
                        res = decrypt_from_src(conn, AS_STR)
                        if res == OK:
                            encrypt_and_send(conn, file_name)
```

```
                            res = decrypt_from_src(conn, REQ_DEL_FILE)
                            if res == FAILURE:
                                print(f'Something went wrong deleting cloud file.
Aborting ...')
                                proceed = False
                        else:
                            raise Exception(PROTOCOL_EX)
                    elif overwrite == 'n':
                        valid_option = True
                        proceed = False
                        print(f'Operation cancelled.')
                    else:
                        print(f'Invalid option. Please enter [Y/N].')

            if proceed:
                upload_file_path = os.path.join(path_to_uploads, file_name)
                print(f'Uploading \'{file_name}\' ... ')

                file_bytes_unencrypted = None
                with open(upload_file_path, 'rb') as upload_file:
                    file_bytes_unencrypted = upload_file.read()

                encrypt_and_send(conn, REQ_UPLOAD)
                res = decrypt_from_src(conn, AS_STR)

                if res == OK:
                    # proceed to send encrypted file bytes
                    encrypt_and_send(conn, file_name)

                    res = decrypt_from_src(conn, AS_STR)
                    if res == OK:
                        encrypt_and_send(conn, file_bytes_unencrypted)
                    else:
                        raise Exception(PROTOCOL_EX)

                    res = decrypt_from_src(conn, AS_STR)
                    if res == SUCCESS:
                        # add file to local list of files available on cloud
                        cloud_files.append(file_name)

                        print(f'File uploaded to cloud successfully.')
                    else:
                        print(f'Unexpected error uploading file to cloud')

                else:
                    raise Exception(PROTOCOL_EX)

        else:
            print(f'Couldn\'t find file \'{file_name}\'. Please ensure this file
is '
                  f'located at {path_to_uploads}')
```

Scans the user's uploads folder for files to upload. Asks them to input a file name, and if it already exists in the cloud prompts them to ask if they're okay with the remote file being overwritten. If so, the old file is deleted and the new file is send to the CAM to upload to the cloud. Reports the results of the operation to the user.

```
def handle_admin(conn, username):
```

```
    keep_going = True

    while keep_going:
        option = input(f'Please choose an option:'
                       f'\n\t[U]:\tUpload a file'
                       f'\n\t[D]:\tDownload a file'
                       f'\n\t[M]:\tManage cloud group'
                       f'\n\t[B]:\tLog out\n').lower()

        if option == 'u':
            handle_upload(conn, username)
        elif option == 'd':
            handle_download(conn, username)
        elif option == 'm':
            manage_cloud_group(conn, username)
        elif option == 'b':
            keep_going = False
```

Presents the admin users with an expanded menu that allows them the manage the cloud group.

```
def manage_cloud_group(conn, username):
    keep_going = True

    while keep_going:
        option = input(f'Please choose a user management option:'
                       f'\n[U]:\tDelete a user'
                       f'\n[F]:\tDelete a cloud file'
                       f'\n[P]:\tPromote a user to admin'
                       f'\n[D]:\tDemote an admin'
                       f'\n[B]:\tGo back\n').lower()

        if option == 'u':
            handle_delete_user(conn)
        elif option == 'f':
            handle_file_deletion(conn)
        elif option == 'd':
            handle_admin_demotion(conn)
        elif option == 'p':
            handle_user_promotion(conn)
        elif option == 'b':
            keep_going = False
        else:
            print(f'Option not recognised. Please try again.\n')
```

Another menu presented to admin users when they opt to manage the cloud group.

```
def handle_file_deletion(conn):
    global cloud_files

    keep_going = True

    while keep_going is True:
        print(f'Files currently stored on cloud:')
        for file in cloud_files:
            print(f'\t{file}')
```

```
        file_name = input(f'Please enter the name of the file you wish to delete:
([B] to go back)\n')

        if file_name.lower() == 'b':
            keep_going = False
        elif file_name in cloud_files:
            # send request to CAM to delete file
            encrypt_and_send(conn, REQ_DEL_FILE)
            res = decrypt_from_src(conn, AS_STR)

            if res == OK:

                # send filename to delete
                encrypt_and_send(conn, file_name)
                res = decrypt_from_src(conn, AS_STR)

                if res == SUCCESS:
                    print(f'File \'{file_name}\' successfully removed from cloud
group storage')

                    # remove reference in local data structure
                    cloud_files.remove(file_name)
                else:
                    print(f'Something went wrong deleting that file :/')

            else:
                raise Exception(PROTOCOL_EX)

        else:
            print(f'Invalid file name. Please try again.')
```

Can be used by an admin to delete a file on the cloud. Checks if the file exists in the cloud, and if so, sends request to CAM to delete and removes from local data structure.

```
def handle_admin_demotion(conn):
    global users, admins

    valid_username = False

    while valid_username is False:

        print(f'Admins available for demotion:')
        for admin in admins:
            print(f'\t{admin}')

        target_admin = input(f'\nPlease enter the name of the admin you wish to
demote. ([B]ack to go back)\n')

        if target_admin.lower() == 'b':
            valid_username = True  # break
        elif target_admin in admins:
            # Move local record into users list
            admins.remove(target_admin)
            users.append(target_admin)

            # Send CAM a message to move user record into users directory
            encrypt_and_send(conn, REQ_RM_ADMIN)
            res = decrypt_from_src(conn, AS_STR)
```

```
            if res == OK:
                # send username to demote:
                encrypt_and_send(conn, target_admin)
                res = decrypt_from_src(conn, AS_STR)

                if res == SUCCESS:
                    print(f'User {target_admin} successfully demoted to user')
                    print(f'\tUSERS: {users}')
                    print(f'\tADMINS: {admins}')
                else:  # FAILURE
                    print(f'Unexpected error occurred when promoting user in
CAM :/')
            else:
                raise Exception(PROTOCOL_EX)

        elif target_admin in users:
            print(f'That user is not an admin!')
        else:  # invalid username
            print(f'That user does not exist. Please try again.')
```

Can be used by an admin to demote another admin to user. Checks the passed username represents an admin and if so sends a demotion request to CAM and updates the local data structures.

```
def handle_user_promotion(conn):
    global users, admins

    valid_username = False

    while valid_username is False:

        print(f'Users available for promotion:')
        for user in users:
            print(f'\t{user}')

        target_user = input(f'\nPlease enter the name of the user you wish to
promote to admin. ([B]ack to go back)\n')

        if target_user.lower() == 'b':
            valid_username = True  # break
        elif target_user in users:
            # Move local record into admins list
            users.remove(target_user)
            admins.append(target_user)

            # Send CAM a message to move user record into admin directory
            encrypt_and_send(conn, REQ_MK_ADMIN)
            res = decrypt_from_src(conn, AS_STR)

            if res == OK:
                # send username to promote:
                encrypt_and_send(conn, target_user)
                res = decrypt_from_src(conn, AS_STR)

                if res == SUCCESS:
                    print(f'User {target_user} successfully promoted to admin')
                    print(f'\tUSERS: {users}')
                    print(f'\tADMINS: {admins}')
```

```
                else:  # FAILURE
                    print(f'Unexpected error occurred when promoting user in
CAM :/')
            else:
                raise Exception(PROTOCOL_EX)
        elif target_user in admins:
            print(f'That user is already an admin!')
        else:  # invalid username
            print(f'That user does not exist. Please try again.')
```

Same as previous but for promoting a user to admin.

```
def handle_delete_user(conn):

    global users, admins

    all_users = users + admins
    print(all_users)

    valid_username = False

    while valid_username is False:
        target_user = input(f'Enter a username to delete. Enter [L] to view a list
of registered users. '
                            f'Enter [B] to go back.\n')

        if target_user == 'L' or target_user == 'l':
            for user in all_users:
                print(f'\t{user}')
        elif target_user == 'B' or target_user == 'b':
            valid_username = True
        elif target_user in all_users:
            # send req to delete to cam:
            encrypt_and_send(conn, REQ_DEL_USER)
            res = decrypt_from_src(conn, AS_STR)

            if res == OK:
                # send the username
                encrypt_and_send(conn, target_user)
                res = decrypt_from_src(conn, AS_STR)

                if res == SUCCESS:
                    print(f'User {target_user} deleted from CAM records ... ')
                else:
                    print(f'Couldn\'t find record for user {target_user} in
CAM :/')
            else:
                raise Exception(PROTOCOL_EX)

            # delete from local group records:
            cur_dir = os.path.dirname(os.path.realpath(__file__))
            group_files_target = os.path.join(cur_dir,
f'group_files\\{target_user}')
            print(group_files_target)

            rmtree(group_files_target)
```

Allows an admin to delete a user provided that that user exists. Sends a request to the CAM and removes the user's group_files records and references to the user in local data structures.

```
def encrypt_and_send(conn, msg):
    global public_key_cam

    # convert msg to bytes
    if isinstance(msg, str):
        msg = str.encode(msg)

    ciphertext = public_key_cam.encrypt(
        msg,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    # generate message digest
    signature = private_key.sign(
        ciphertext,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )

    # print(f'\tSending {msg}; ciphertext: {ciphertext}')
    conn.send(ciphertext)

    # print(f'\t\tSending signature: {signature}')
    conn.send(signature)
```

Same implementation as in CAM.

```
def decrypt_from_src(conn, as_what):
    global private_key, public_key_cam

    ciphertext = conn.recv()
    signature = conn.recv()

    try:
        verified_ciphertext = public_key_cam.verify(
            signature,
            ciphertext,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )

        # will throw exception if signature not valid
        #   otherwise decrypt
        plaintext = private_key.decrypt(
            ciphertext,
```

```
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )

        # cast as string if requested
        if as_what == AS_STR:
            plaintext = plaintext.decode("utf-8")

        # print(f'\tReceived {plaintext}; ciphertext: {ciphertext}')
        return plaintext

    except InvalidSignature:
        print(f'Invalid signature on msg')

    return None
```

Same implementation as in CAM.

```
def request_download(conn, filename, username):
    encrypt_and_send(conn, REQ_DOWNLOAD)
    res = decrypt_from_src(conn, AS_STR)

    if res == OK:
        # submit request for file in form [filename.ext]
        encrypt_and_send(conn, filename)

        res = decrypt_from_src(conn, AS_BYTES)

        if res == str.encode(FAILURE):  # file not found on cloud
            print(f'Unexpected error finding file in cloud.')
        else:
            write_to_user_directory(res, username, filename)
            print(f'File download successful. You file has been saved to
\'group_files\\{username}\\downloads\'.')

    else:
        raise Exception(PROTOCOL_EX)
```

Sends a request to the CAM to transfer the named file and calls a function to write the file to the logged-in user's group_files downloads folder.

```
def write_to_user_directory(file_bytes, username, filename):
    # files that are downloaded are written to group_files/<username>/downloads/
    cur_dir = os.path.dirname(os.path.realpath(__file__))

    path_to_user_dir = os.path.join(cur_dir,
f'group_files\\{username}\\downloads')

    # make parent directories if they don't already exist
    if not os.path.exists(path_to_user_dir):
        os.makedirs(path_to_user_dir)

    path_to_target_file = os.path.join(path_to_user_dir, filename)

    with open(path_to_target_file, 'wb') as target_file:
```

```
        target_file.write(file_bytes)
        target_file.close()
```

Writes the named file and its bytes to the logged-in user's downloads folder, making the downloads folder if it doesn't already exist.

APPENDIX:

Full code listing without breaks:

CLOUDACCESSMANAGER.PY

```
import io
import os.path

from cryptography.exceptions import InvalidSignature
from cryptography.hazmat.backends import default_backend
from google.oauth2.credentials import Credentials
from google.auth.transport.requests import Request
from google_auth_oauthlib.flow import InstalledAppFlow
from googleapiclient.discovery import build
from googleapiclient.http import MediaFileUpload, MediaIoBaseDownload

from multiprocessing.connection import Listener
from apiclient import errors

from cryptography.fernet import Fernet
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.primitives.asymmetric import rsa, padding

# What GDrive permissions we're requiring:
SCOPES = ['https://www.googleapis.com/auth/drive']
drive_service = None

# Constants
REGISTER = 0
LOG_IN = 1

CLOUD = 0
CLIENT = 1

ENCRYPT = 0
DECRYPT = 1

AS_BYTES = 0
AS_STR = 1

USER = 0
ADMIN = 1

USERS_PATH = r"cam_files\users"
ADMINS_PATH = r"cam_files\admins"

# Communication protocol between CAM and Client:
HELLO = "hello"
REQ_USER_LIST = "userlist"
REQ_REGISTER = "register"
REQ_CLOSE = "close"
```

```
REQ_LOGIN = "login"
REQ_DOWNLOAD = "download"
REQ_UPLOAD = "upload"
REQ_CLOUD_FILES = "files"
REQ_DEL_USER = "deluser"
REQ_MK_ADMIN = "admin"
REQ_RM_ADMIN = "demote"
REQ_DEL_FILE = "delfile"
OK = "ok"
SUCCESS = "success"  # request completed successfully
FAILURE = "failure"  # something went wrong

# dynamic data structure for keeping a list of usernames in memory
user_usernames = []
admin_usernames = []
cloud_filenames = {}

symmetric_key_cloud = None
public_key_client = None
private_key = None


def main():
    global user_usernames, admin_usernames, \
        symmetric_key_cloud, drive_service, cloud_filenames, public_key_client,
private_key

    # perform one-time directory setups
    perform_dir_setup()

    # authorise self to upload/download from associated GDrive account
    drive_service = perform_cloud_auth()

    # perform one-time public/private key generation or load existing keys
    load_asymm_keys()

    # get the Fernet key for communication between program and cloud
    symmetric_key_cloud = Fernet(load_key(CLOUD))

    # initialise list of usernames (one time file-read)
    user_usernames, admin_usernames = load_usernames(USERS_PATH, ADMINS_PATH)
    # initialise list of gdrive files
    load_cloud_file_list()

    print(f'CloudAccessManager ready to service requests ...')

    # listen for communication from cloud group client
    address = ('localhost', 6000)
    listener = Listener(address, authkey=b'cloud_group')

    # accept connection via socket
    conn = listener.accept()
    print(f'connection accepted from {listener.last_accepted}')

    # client initiate comms with a plaintext HELLO exchange:

    proceed = True

    msg = conn.recv()
```

```
    if msg == HELLO:

        # respond with plaintext hello
        conn.send(HELLO)

        # If first time communicating with client, exchange pub keys in plaintext
        cur_dir = os.path.dirname(os.path.realpath(__file__))
        path_to_client_pubkey = os.path.join(cur_dir,
f'cam_files\\keys\\client_pubkey.pem')

        if not os.path.exists(path_to_client_pubkey):
            perform_key_exchange(conn, cur_dir, path_to_client_pubkey)

        # further comms will be encrypted, so load client's pub key
        load_client_pub_key()

    else:
        proceed = False
        conn.close()  # terminate connection; protocol not being followed

    if proceed is True:

        while True:  # repeatedly service requests from client until they close

            # keys exchanged, can now engage in encrypted comms
            close = False

            while close is False:

                req = decrypt_from_src(conn, AS_STR)

                if req == REQ_USER_LIST:
                    send_user_list(conn)
                elif req == REQ_REGISTER:
                    process_registration(conn)
                elif req == REQ_LOGIN:
                    process_login(conn)
                elif req == REQ_DOWNLOAD:
                    process_download(conn)
                elif req == REQ_CLOUD_FILES:
                    send_file_list(conn)
                elif req == REQ_UPLOAD:
                    handle_upload(conn)
                elif req == REQ_DEL_USER:
                    handle_user_deletion(conn)
                elif req == REQ_MK_ADMIN:
                    handle_user_promotion(conn)
                elif req == REQ_RM_ADMIN:
                    handle_admin_demotion(conn)
                elif req == REQ_DEL_FILE:
                    handle_delete_file(conn)
                elif req == REQ_CLOSE:

                    encrypt_and_send(conn, OK)
                    print(f'Closing connection ... ')
                    close = True

            conn.close()
            break
```

```
    listener.close()


def perform_key_exchange(conn, cur_dir, save_path):
    # receive public key
    client_pubkey = conn.recv()
    print(client_pubkey)

    # store key
    with open(save_path, 'wb') as file:
        file.write(client_pubkey)
        file.close()

    # respond with own pubkey
    path_to_cam_pubkey = os.path.join(cur_dir, f'cam_files\\keys\\public_key.pem')

    with open(path_to_cam_pubkey, 'rb') as file:
        file_bytes = file.read()
        conn.send(file_bytes)


def load_asymm_keys():
    global private_key

    cur_dir = os.path.dirname(os.path.realpath(__file__))
    path_to_keys = os.path.join(cur_dir, f'cam_files\\keys')

    if not os.path.exists(path_to_keys):    # need to generate them
        print(f'Performing one-time key generation ... ')
        # make it
        os.mkdir(path_to_keys)

        # generate public and private keys
        cam_priv_key = rsa.generate_private_key(public_exponent=65537,
key_size=4096)
        cam_public_key = cam_priv_key.public_key()

        # store the keys
        pem_priv = cam_priv_key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.PKCS8,
            encryption_algorithm=serialization.NoEncryption()
        )

        # save local variable for priv key
        private_key = pem_priv

        save_loc = os.path.join(path_to_keys, 'private_key.pem')
        with open(save_loc, 'wb') as pem_file:
            pem_file.write(pem_priv)

        pem_pub = cam_public_key.public_bytes(encoding=serialization.Encoding.PEM,

format=serialization.PublicFormat.SubjectPublicKeyInfo)

        save_loc = os.path.join(path_to_keys, 'public_key.pem')
        with open(save_loc, 'wb') as pem_file:
            pem_file.write(pem_pub)
```

```
    else:
        print(f'Loading asymm keys ... ')

        path_to_priv_key = os.path.join(path_to_keys, f'private_key.pem')

        # retrieve private key
        with open(path_to_priv_key, "rb") as key_file:
            private_key = serialization.load_pem_private_key(
                key_file.read(),
                password=None,
                backend=default_backend()
            )


def load_client_pub_key():
    global public_key_client

    cur_dir = os.path.dirname(os.path.realpath(__file__))
    path_to_client_pubkey = os.path.join(cur_dir,
f'cam_files\\keys\\client_pubkey.pem')

    with open(path_to_client_pubkey, "rb") as key_file:
        public_key_client = serialization.load_pem_public_key(
            key_file.read(),
            backend=default_backend()
        )


def perform_cloud_auth():
    # Load access token or creates one if DNE
    creds = None

    # Load credentials if they exist (i.e. the authorisation set up has been run
already)
    if os.path.exists('cam_files\\gdrive\\token.json'):
        creds =
Credentials.from_authorized_user_file('cam_files\\gdrive\\token.json', SCOPES)

    # If they don't exist, create
    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            creds.refresh(Request())
        else:
            flow = InstalledAppFlow.from_client_secrets_file(
                'cam_files\\gdrive\\credentials.json', SCOPES)
            creds = flow.run_local_server(port=0)
        # Save the credentials for the next run
        with open('cam_files\\gdrive\\token.json', 'w') as token:
            token.write(creds.to_json())

    # return reference to drive service object that will handle upload/download
requests
    return build('drive', 'v3', credentials=creds)


def load_key(type):
    cwd = os.getcwd()
    key = None
```

```
    # Either fetch saved fernet key from previous session or generate if DNE
    if type == CLOUD:
        path_to_key = f'{cwd}\\cam_files\\fernet_cloud.key'
    elif type == CLIENT:
        path_to_key = f'{cwd}\\cam_files\\fernet_client.key'

    if os.path.exists(path_to_key) is False:  # key dne yet; create
        # generate symmetric key for communication with cloud/client
        print(f'Writing New {"Cloud" if type == CLOUD else "Client"} Symmetric
Key ...')
        key = Fernet.generate_key()

        # save the key
        with open(path_to_key, 'wb') as key_file:
            key_file.write(key)
            key_file.close()

        # Save an additional copy for the client to use
        if type == CLIENT:
            path_to_client = f'{cwd}\\client_files\\fernet_client.key'

            # save the key
            with open(path_to_client, 'wb') as key_file:
                key_file.write(key)
                key_file.close()

    if key is None:  # will be none if it already existed; load
        # load symmetric key
        with open(path_to_key, 'rb') as key_file:
            key = key_file.read()

    return key


# fetch list of usernames for dynamic storage
def load_usernames(users_path, admins_path):
    users = []
    admins = []

    cur_dir = os.path.dirname(os.path.realpath(__file__))

    full_path_users = os.path.join(cur_dir, users_path)
    users = [os.path.splitext(file)[0] for file in os.listdir(full_path_users)]

    full_path_admins = os.path.join(cur_dir, admins_path)
    admins = [os.path.splitext(file)[0] for file in os.listdir(full_path_admins)]

    return users, admins


# get a list of files from the google drive service
def load_cloud_file_list():
    global cloud_filenames

    results = drive_service.files().list().execute()
    items = results.get('files', [])

    if not items:
        print('No files found.')
```

```
    else:
        print('Files:')
        for item in items:
            print(u'{0} ({1})'.format(item['name'], item['id']))

    # List of filename : id pairs
    for item in items:
        cloud_filenames[item['name']] = item['id']

    print(cloud_filenames)


# Client request handlers:
def send_user_list(conn):

    encoded_list = " ".join(user_usernames)
    encoded_list += "|"
    encoded_list += " ".join(admin_usernames)

    encrypt_and_send(conn, encoded_list)


def process_registration(conn):
    encrypt_and_send(conn, OK)

    registration_details = decrypt_from_src(conn, AS_STR)  # in form
"username|password"
    registration_details = registration_details.split("|")
    username = registration_details[0]
    password = registration_details[1]
    print(f'Preparing to register user \'{username}\' ... ')

    # write the new user details to CAM files:
    cur_dir = os.path.dirname(os.path.realpath(__file__))
    full_path_users = os.path.join(cur_dir, USERS_PATH)
    path_to_new_file = os.path.join(full_path_users, f'{username}.txt')

    with open(path_to_new_file, 'w') as new_user_file:
        new_user_file.write(f'{password}')
        new_user_file.close()

    # add to dynamic data structure representing usernames:
    user_usernames.append(username)

    # conn.send(SUCCESS)
    encrypt_and_send(conn, SUCCESS)
    print(f'Successfully created registration record for \'{username}\'')


def handle_user_promotion(conn):
    global user_usernames, admin_usernames

    encrypt_and_send(conn, OK)
    username = decrypt_from_src(conn, AS_STR)

    cur_dir = os.path.dirname(os.path.realpath(__file__))
    # get cur path to user record:
    full_path_to_user_record = os.path.join(cur_dir,
f'cam_files\\users\\{username}.txt')
```

```
    # get dest path:
    dest_path = os.path.join(cur_dir, f'cam_files\\admins\\{username}.txt')

    # move the record
    os.rename(full_path_to_user_record, dest_path)

    # update dynamic data structures holding usernames
    user_usernames.remove(username)
    admin_usernames.append(username)

    encrypt_and_send(conn, SUCCESS)


def handle_admin_demotion(conn):
    global user_usernames, admin_usernames

    encrypt_and_send(conn, OK)
    username = decrypt_from_src(conn, AS_STR)

    cur_dir = os.path.dirname(os.path.realpath(__file__))
    # get cur path to admin record:
    full_path_to_admin_record = os.path.join(cur_dir,
f'cam_files\\admins\\{username}.txt')

    # get dest path:
    dest_path = os.path.join(cur_dir, f'cam_files\\users\\{username}.txt')

    # move the record
    os.rename(full_path_to_admin_record, dest_path)

    # update dynamic data structures holding usernames
    admin_usernames.remove(username)
    user_usernames.append(username)

    encrypt_and_send(conn, SUCCESS)


def process_login(conn):
    # acknowledge login request
    encrypt_and_send(conn, OK)

    login_details = decrypt_from_src(conn, AS_STR)  # in form
'[username]|[password]'
    login_details = login_details.split("|")
    print(f'Login Details: {login_details}')
    username = login_details[0]
    password = login_details[1]

    user_type = None
    print(f'Users: {user_usernames}')
    print(f'Admins: {admin_usernames}')

    if username in user_usernames:
        user_type = USER
    elif username in admin_usernames:
        user_type = ADMIN

    if user_type is not None:
```

```
        # get user's password from storage
        cur_dir = os.path.dirname(os.path.realpath(__file__))

        if user_type == USER:
            full_path = os.path.join(cur_dir, USERS_PATH)
        else:  # admin
            full_path = os.path.join(cur_dir, ADMINS_PATH)

        path_user_record = os.path.join(full_path, f'{username}.txt')
        print(path_user_record)

        with open(path_user_record, 'rb') as user_file:
            # receive string representation from saved bytes
            true_password = ''.join(user_file.readline().decode('utf-8').split())

        # if passed pw matches registration pw
        if password == true_password:
            encrypt_and_send(conn, SUCCESS)
        else:
            encrypt_and_send(conn, FAILURE)

    else:
        # user dne
        encrypt_and_send(conn, FAILURE)


def process_download(conn):
    global cloud_filenames

    encrypt_and_send(conn, OK)

    target_file = decrypt_from_src(conn, AS_STR)
    print(target_file)
    print(cloud_filenames)

    if target_file in cloud_filenames:
        # query Gdrive for file
        request =
drive_service.files().get_media(fileId=cloud_filenames[target_file])

        file_handler = io.BytesIO()
        downloader = MediaIoBaseDownload(file_handler, request)
        done = False
        while done is False:
            status, done = downloader.next_chunk()
            print(f'Fetching ... {int(status.progress() * 100)}%.')

        # file bytes in file_handler; will have been encrypted with cloud symm key
        #   decrypt; re-encrypt with client symm key and send
        decrypted_bytes = symmetric_key_cloud.decrypt(file_handler.getvalue())
        encrypt_and_send(conn, decrypted_bytes)

    else:
        encrypt_and_send(conn, FAILURE)


def encrypt_and_send(conn, msg):
    global public_key_client, private_key
```

```
    # convert msg to bytes
    if isinstance(msg, str):
        msg = str.encode(msg)

    # encrypt with client's public key
    ciphertext = public_key_client.encrypt(
        msg,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    # generate the message digest
    signature = private_key.sign(
        ciphertext,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )

    # print(f'\tSending {msg}; ciphertext: {ciphertext}')
    conn.send(ciphertext)

    # print(f'\t\tSending signature: {signature}')
    conn.send(signature)


# decrypts a message from the client encrypted using CAM's pub key by using the
CAM's private key
def decrypt_from_src(conn, as_what):
    global private_key, public_key_client

    ciphertext = conn.recv()
    signature = conn.recv()

    try:
        verified_ciphertext = public_key_client.verify(
            signature,
            ciphertext,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )

        # will throw exception if signature not valid
        #   otherwise decrypt
        plaintext = private_key.decrypt(
            ciphertext,
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None
```

```
            )
        )

        # cast as string if requested
        if as_what == AS_STR:
            plaintext = plaintext.decode("utf-8")

        # print(f'\tReceived {plaintext}; ciphertext: {ciphertext}')
        return plaintext

    except InvalidSignature:
        print(f'Invalid signature on msg')

    return None


def handle_delete_file(conn):
    global cloud_filenames

    encrypt_and_send(conn, OK)
    target_file = decrypt_from_src(conn, AS_STR)

    # get associated GDrive id:
    file_id = cloud_filenames[target_file]

    delete_success = False
    # call on drive service to delete the file:
    try:
        drive_service.files().delete(fileId=file_id).execute()
        delete_success = True
    except errors.HttpError as error:
        print(f'ERROR OCCURRED DELETING FILE {target_file}:{file_id}\n\t{error}')

    if delete_success:
        # remove from local data structure
        cloud_filenames.pop(target_file)

        # report success to client
        encrypt_and_send(conn, SUCCESS)
    else:
        encrypt_and_send(conn, FAILURE)


def send_file_list(conn):
    global cloud_filenames

    encoded_file_list = "|".join(cloud_filenames.keys())

    encrypt_and_send(conn, encoded_file_list)


def handle_upload(conn):
    global drive_service

    encrypt_and_send(conn, OK)

    # response will be filename followed by filebytes
    file_name = decrypt_from_src(conn, AS_STR)
    encrypt_and_send(conn, OK)
```

```
    file_bytes = decrypt_from_src(conn, AS_BYTES)

    # encrypt the file using CAM's cloud symm key
    encrypted_file_bytes = symmetric_key_cloud.encrypt(file_bytes)

    # upload to cloud
    #   first need to save to staging area

    cur_dir = os.path.dirname(os.path.realpath(__file__))
    rel_path = f'cam_files\\stage\\{file_name}'
    path_to_stage_file = os.path.join(cur_dir, rel_path)

    with open(path_to_stage_file, 'wb') as stage_file:
        stage_file.write(encrypted_file_bytes)
        stage_file.close()

    # now upload it
    file_metadata = {'name': file_name}
    to_upload = MediaFileUpload(rel_path, resumable=True)
    file = drive_service.files().create(body=file_metadata,
                                        media_body=to_upload,
                                        fields='id').execute()
    to_upload = None

    # save uploaded file data to dynamic data structure
    cloud_filenames[file_name] = file.get('id')

    # remove file from staging area
    os.remove(rel_path)

    # tell client it was successful
    encrypt_and_send(conn, SUCCESS)


# one-time setup of cam_files directory structure
def perform_dir_setup():
    cur_dir = os.path.dirname(os.path.realpath(__file__))
    path_to_cam_files = os.path.join(cur_dir, f'cam_files')

    if not os.path.exists(path_to_cam_files):
        print(f'Creating CAM files @ {path_to_cam_files} ... ')
        os.mkdir(path_to_cam_files)

        path_to_stage = os.path.join(path_to_cam_files, f'stage')
        print(f'Setting up stage @ {path_to_stage} ... ')
        os.mkdir(path_to_stage)

        path_to_users = os.path.join(path_to_cam_files, f'users')
        print(f'Setting up user records @ {path_to_users} ... ')
        os.mkdir(path_to_users)

        path_to_admins = os.path.join(path_to_cam_files, f'admins')
        print(f'Setting up admin records @ {path_to_admins} ... ')
        os.mkdir(path_to_admins)

        # create default admin account
        path_to_default_admin = os.path.join(path_to_admins, f'sysadmin.txt')
        with open(path_to_default_admin, 'w') as default_admin:
            default_admin.write('adminpw')
```

```
            default_admin.close()

        path_to_gdrive = os.path.join(path_to_cam_files, f'cam_files/gdrive')
        print(f'Setting up GDrive folder @ {path_to_gdrive}')
        os.mkdir(path_to_gdrive)
        # make credentials.json here:
        with open(os.path.join(path_to_gdrive,
f'cam_files/gdrive/credentials.json'), 'w') as f:
            f.close()


def handle_user_deletion(conn):
    encrypt_and_send(conn, OK)

    username = decrypt_from_src(conn, AS_STR)

    rel_path = None
    if username in admin_usernames:
        # also delete record in users dynamic data structure:
        admin_usernames.remove(username)
        rel_path = os.path.join(ADMINS_PATH, f'{username}.txt')
    elif username in user_usernames:
        user_usernames.remove(username)
        rel_path = os.path.join(USERS_PATH, f'{username}.txt')
    else:  # user doesn't seem to exist :/
        encrypt_and_send(conn, FAILURE)

    if rel_path is not None:
        # delete user record file
        cur_dir = os.path.dirname(os.path.realpath(__file__))
        target_file = os.path.join(cur_dir, rel_path)

        os.remove(target_file)

        encrypt_and_send(conn, SUCCESS)


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

CLOUDGROUPCLIENT.PY

```
import os
import re

from multiprocessing.connection import Client

from shutil import rmtree

from cryptography.exceptions import InvalidSignature
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.primitives.asymmetric import rsa, padding

USER_PRIVILEGE = 0
ADMIN_PRIVILEGE = 1
```

```
AS_BYTES = 0
AS_STR = 1

REGISTER = 'r'
LOG_IN = 'l'
QUIT = 'q'

PROTOCOL_EX = "Unexpected communication protocol error"

REGEX_VALID_USERNAME = '^[A-Za-z0-9_-]*$'
REGEX_VALID_PASSWORD = r'[A-Za-z0-9@#$%^&+=]{6,}'

# Communication protocol between CAM and Client:
HELLO = "hello"
REQ_USER_LIST = "userlist"
REQ_REGISTER = "register"
REQ_CLOSE = "close"
REQ_LOGIN = "login"
REQ_DOWNLOAD = "download"
REQ_UPLOAD = "upload"
REQ_CLOUD_FILES = "files"
REQ_DEL_USER = "deluser"
REQ_MK_ADMIN = "admin"
REQ_RM_ADMIN = "demote"
REQ_DEL_FILE = "delfile"
OK = "ok"
SUCCESS = "success"  # request completed successfully
FAILURE = "failure"  # something went wrong

# dynamic data structure for keeping a list of usernames in memory
users = []
admins = []
cloud_files = []

public_key_cam = None
private_key = None


def main():
    global users, admins, cloud_files, public_key_cam, private_key

    # perform one-time setup of client_files and keys where applicable
    perform_initial_setup()

    # connect to CAM
    address = ('localhost', 6000)
    conn = Client(address, authkey=b'cloud_group')

    # initialise communication with CAM
    #   send plaintext HELLO msg
    conn.send(HELLO)
    res = conn.recv()
    if res == HELLO:

        # if haven't got a record of CAM's pubkey, keys haven't been exchanged
yet.
        cur_dir = os.path.dirname(os.path.realpath(__file__))
        path_to_cam_pubkey = os.path.join(cur_dir,
f'client_files\\cam_pubkey.pem')
```

```
        if not os.path.exists(path_to_cam_pubkey):
            perform_key_exchange(conn, cur_dir, path_to_cam_pubkey)

        # now load keys
        load_keys()

        # begin registration/login flow:
        encrypt_and_send(conn, REQ_USER_LIST)
        user_list_string = decrypt_from_src(conn, AS_STR)
        users, admins = extract_usernames(user_list_string)

        # get filenames from cloud
        encrypt_and_send(conn, REQ_CLOUD_FILES)
        res = decrypt_from_src(conn, AS_STR)
        cloud_files = res.split("|")

        prompt_for_login(users, admins, conn)

        # ^ returns when user hits QUIT, so close connection with CAM and exit
        encrypt_and_send(conn, REQ_CLOSE)
    else:
        raise Exception(PROTOCOL_EX)


def load_keys():
    global private_key, public_key_cam

    # load client's private key for decryption
    cur_dir = os.path.dirname(os.path.realpath(__file__))
    path_to_priv_key = os.path.join(cur_dir, f'client_files\\private_key.pem')

    with open(path_to_priv_key, "rb") as key_file:
        private_key = serialization.load_pem_private_key(
            key_file.read(),
            password=None,
            backend=default_backend()
        )

    # load CAM's public key for encryption
    path_to_cam_pubkey = os.path.join(cur_dir, f'client_files\\cam_pubkey.pem')

    with open(path_to_cam_pubkey, "rb") as key_file:
        public_key_cam = serialization.load_pem_public_key(
            key_file.read(),
            backend=default_backend()
        )


def perform_key_exchange(conn, cur_dir, save_path):

    print(f'Performing one-time public key exchange ... ')
    # send client's pubkey to CAM
    path_to_client_pubkey = os.path.join(cur_dir, f'client_files\\public_key.pem')

    with open(path_to_client_pubkey, 'rb') as file:
        file_bytes = file.read()
        conn.send(file_bytes)
```

```
    # CAM responds with own PKey
    cam_pubkey = conn.recv()

    # save it
    with open(save_path, 'wb') as file:
        file.write(cam_pubkey)
        file.close()


def perform_initial_setup():
    cur_dir = os.path.dirname(os.path.realpath(__file__))
    path_client_files = os.path.join(cur_dir, f'client_files')

    if not os.path.exists(path_client_files):
        os.mkdir(path_client_files)
        print(f'Created \'{path_client_files}\'')

        print(f'Generating keys ...')
        # create client's asymm keys
        client_priv_key = rsa.generate_private_key(public_exponent=65537,
key_size=4096)
        client_public_key = client_priv_key.public_key()

        # store the keys
        pem_priv = client_priv_key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.PKCS8,
            encryption_algorithm=serialization.NoEncryption()
        )

        save_loc = os.path.join(path_client_files, 'private_key.pem')
        with open(save_loc, 'wb') as pem_file:
            pem_file.write(pem_priv)

        pem_pub =
client_public_key.public_bytes(encoding=serialization.Encoding.PEM,

format=serialization.PublicFormat.SubjectPublicKeyInfo)

        save_loc = os.path.join(path_client_files, 'public_key.pem')
        with open(save_loc, 'wb') as pem_file:
            pem_file.write(pem_pub)


# decode user list received by CAM
def extract_usernames(user_list_string):

    # divide into users and admins
    split_list = user_list_string.split("|")
    user_usernames = split_list[0].split(" ")
    admin_usernames = split_list[1].split(" ")

    return user_usernames, admin_usernames


def prompt_for_login(users, admins, conn):
    login_success = False

    while not login_success:
```

```
        print('\n')

        privilege_level = None
        option = input(
            f'Welcome to SecuringTheCloud. [R]egister or [L]og in? ([Q] to
exit)\n-----------------\n').lower()

        if option == REGISTER:

            res = register_new_user(conn, users + admins)

        elif option == LOG_IN:
            handle_log_in(conn)

        elif option == QUIT:
            encrypt_and_send(conn, REQ_CLOSE)

            res = decrypt_from_src(conn, AS_STR)
            if res == OK:
                print(f'Goodbye!')
                break
            else:
                raise Exception(PROTOCOL_EX)

        else:
            print(f'Not a valid option. Please try again.')


def register_new_user(conn, exclusion_list):
    valid_username = False
    valid_password = False
    go_back = False

    username = None

    while valid_username is False:
        username = input(f'Preparing to register a new user... ([B] to go back)\n
Username: ').lower()

        if username == 'b':
            go_back = True
            break
        elif 6 <= len(username) <= 15 and re.match(REGEX_VALID_USERNAME,
username):
            if username in exclusion_list:
                print(f' Sorry, that username has been taken.\n')
            else:
                valid_username = True
        else:
            print(
                f' Please enter a username between 6 and 15 characters containing
only letters, numbers, -, and/or _\n')

    if go_back is False:
        while valid_password is False:
            password = input(' Password: ')
```

```
            if 6 <= len(password) <= 15 and re.match(REGEX_VALID_PASSWORD,
password):
                valid_password = True
            else:
                print(f' Please enter a password between 6 and 15 characters
containing only letters, numbers, '
                      f'and/or the following special characters: @ # $ % ^ & +
=\n')

        if username is not None:
            encrypt_and_send(conn, REQ_REGISTER)

            res = decrypt_from_src(conn, AS_STR)
            if res == OK:
                encrypt_and_send(conn, f'{username}|{password}')
                users.append(username)
            else:
                raise Exception(PROTOCOL_EX)

            res = decrypt_from_src(conn, AS_STR)
            if res == SUCCESS:
                validate_group_folder(username)
                print(f'Registered user \'{username}\' successfully. Please
proceed to log in.')

            return True
    else:
        raise Exception(PROTOCOL_EX)

    return False


def handle_log_in(conn):
    go_back = False

    while go_back is False:

        print(f'\nEnter your login details ([B] to go back)')

        username = input(f' Username: ').lower()
        if username == 'b':
            break

        password = input(f' Password: ')

        # send to CAM to authenticate:
        # send request to log in
        encrypt_and_send(conn, REQ_LOGIN)

        # await OK from CAM
        res = decrypt_from_src(conn, AS_STR)
        if res == OK:
            # proceed to send login details for verification
            encrypt_and_send(conn, f'{username}|{password}')

            # response is either SUCCESS if verifiable or FAIL if not
            res = decrypt_from_src(conn, AS_STR)
            if res == SUCCESS:
```

```
                # successful login
                print(f'\nLogin Successful. Welcome {username}.')

                if username in users:
                    validate_group_folder(username)
                    handle_user(conn, username)
                elif username in admins:
                    validate_group_folder(username)
                    handle_admin(conn, username)
                else:
                    raise Exception(f'Something\'s gone terribly wrong :(')

            elif res == FAILURE:
                # unsuccessful login; repeat loop
                print(f'Login unsuccessful. Please try again.')
            else:
                raise Exception(PROTOCOL_EX)

        else:
            raise Exception(PROTOCOL_EX)


# creates \group_files\{username}\uploads if dne
def validate_group_folder(username):
    cur_dir = os.path.dirname(os.path.realpath(__file__))
    full_path = os.path.join(cur_dir, f'group_files\\{username}\\uploads')

    if not os.path.exists(full_path):
        os.makedirs(full_path)


def handle_user(conn, username):
    global cloud_files

    keep_going = True

    while keep_going is True:

        valid_option = False

        while valid_option is False:
            option = input(f'Do you wish to [U]pload or [D]ownload a file? ([B]ack
to logout)\n').lower()

            if option == 'd':

                handle_download(conn, username)

            elif option == 'u':

                handle_upload(conn, username)

            elif option == 'b':

                print(f'Logging out ...')
                valid_option = True
                keep_going = False

            else:
```

```
                print(f'Not a valid option.')


def handle_download(conn, username):
    valid_option = True

    has_downloaded_smth = False

    while has_downloaded_smth is False:

        option = input(
            f'Choose one of the following options:'
            f'\n\t[L]: List the files currently available on the cloud'
            f'\n\t[<filename.ext>]: Download a file'
            f'\n\t[B]: Return to previous menu\n')

        if option == 'l' or option == 'L':

            print(f'\nFiles available for download:')
            for file in cloud_files:
                print(f' {file}')
            print()

        elif option == 'b' or option == 'B':

            valid_option = True
            break

        else:  # file request

            if option in cloud_files:
                has_downloaded_smth = True

                # first notify that local file will be overwritten if required
                cur_dir = os.path.dirname(os.path.realpath(__file__))
                path_to_dl = os.path.join(cur_dir,
f'group_files\\{username}\\downloads\\{option}')

                proceed = True

                if os.path.exists(path_to_dl):  # i.e. already file in user's dl
folder with same name

                    valid_option = False

                    while valid_option is False:
                        option2 = input(f'Proceeding will overwrite local file
\'{option}\' in your downloads folder. '
                                        f'Proceed? [Y/N]\n').lower()

                        if option2 == 'y':
                            valid_option = True
                        elif option2 == 'n':
                            valid_option = True
                            proceed = False
                            print(f'Cancelling operation ... ')
                        else:
                            print(f'Not a recognised option. Please enter [Y/N]')
```

```
                if proceed is True:
                    result = request_download(conn, option, username)

            else:
                print(f'That file does not exist. Enter [L] to see a list of files
available to download.')


def handle_upload(conn, username):

    valid_filename = False

    while valid_filename is False:

        # get filenames in user's uploads folder
        cur_dir = os.path.dirname(os.path.realpath(__file__))
        path_to_uploads = os.path.join(cur_dir,
f'group_files\\{username}\\uploads')

        file_names = None

        if os.path.exists(path_to_uploads):
            print(f'Files available for upload:')
            file_names = os.listdir(path_to_uploads)
            for file in file_names:
                print(f'\t{file}')
            print()
        else:
            raise Exception(f'Uploads folder doesn\'t exist for user {username}')

        file_name = input('\nPlease enter the name of the file you wish to upload.
([B]ack to return)\n')

        if file_name == 'b' or file_name == 'B':
            valid_filename = True
        elif file_name in file_names:

            proceed = True

            if file_name in cloud_files:
                valid_option = False

                while valid_option is False:
                    overwrite = input(f'File with this name already uploaded.
Overwrite? [Y/N]\n').lower()
                    if overwrite == 'y':
                        valid_option = True
                        proceed = True

                        # Delete the old file
                        encrypt_and_send(conn, REQ_DEL_FILE)
                        res = decrypt_from_src(conn, AS_STR)
                        if res == OK:
                            encrypt_and_send(conn, file_name)

                            res = decrypt_from_src(conn, REQ_DEL_FILE)
                            if res == FAILURE:
                                print(f'Something went wrong deleting cloud file.
Aborting ...')
```

```
                                proceed = False
                            else:
                                raise Exception(PROTOCOL_EX)
                        elif overwrite == 'n':
                            valid_option = True
                            proceed = False
                            print(f'Operation cancelled.')
                        else:
                            print(f'Invalid option. Please enter [Y/N].')

                if proceed:
                    upload_file_path = os.path.join(path_to_uploads, file_name)
                    print(f'Uploading \'{file_name}\' ... ')

                    file_bytes_unencrypted = None
                    with open(upload_file_path, 'rb') as upload_file:
                        file_bytes_unencrypted = upload_file.read()

                    encrypt_and_send(conn, REQ_UPLOAD)
                    res = decrypt_from_src(conn, AS_STR)

                    if res == OK:
                        # proceed to send encrypted file bytes
                        encrypt_and_send(conn, file_name)

                        res = decrypt_from_src(conn, AS_STR)
                        if res == OK:
                            encrypt_and_send(conn, file_bytes_unencrypted)
                        else:
                            raise Exception(PROTOCOL_EX)

                        res = decrypt_from_src(conn, AS_STR)
                        if res == SUCCESS:
                            # add file to local list of files available on cloud
                            cloud_files.append(file_name)

                            print(f'File uploaded to cloud successfully.')
                        else:
                            print(f'Unexpected error uploading file to cloud')

                    else:
                        raise Exception(PROTOCOL_EX)

            else:
                print(f'Couldn\'t find file \'{file_name}\'. Please ensure this file
is '
                      f'located at {path_to_uploads}')


def handle_admin(conn, username):
    keep_going = True

    while keep_going:
        option = input(f'Please choose an option:'
                       f'\n\t[U]:\tUpload a file'
                       f'\n\t[D]:\tDownload a file'
                       f'\n\t[M]:\tManage cloud group'
                       f'\n\t[B]:\tLog out\n').lower()
```

```
        if option == 'u':
            handle_upload(conn, username)
        elif option == 'd':
            handle_download(conn, username)
        elif option == 'm':
            manage_cloud_group(conn, username)
        elif option == 'b':
            keep_going = False


def manage_cloud_group(conn, username):
    keep_going = True

    while keep_going:
        option = input(f'Please choose a user management option:'
                       f'\n[U]:\tDelete a user'
                       f'\n[F]:\tDelete a cloud file'
                       f'\n[P]:\tPromote a user to admin'
                       f'\n[D]:\tDemote an admin'
                       f'\n[B]:\tGo back\n').lower()

        if option == 'u':
            handle_delete_user(conn)
        elif option == 'f':
            handle_file_deletion(conn)
        elif option == 'd':
            handle_admin_demotion(conn)
        elif option == 'p':
            handle_user_promotion(conn)
        elif option == 'b':
            keep_going = False
        else:
            print(f'Option not recognised. Please try again.\n')


def handle_file_deletion(conn):
    global cloud_files

    keep_going = True

    while keep_going is True:
        print(f'Files currently stored on cloud:')
        for file in cloud_files:
            print(f'\t{file}')

        file_name = input(f'Please enter the name of the file you wish to delete:
([B] to go back)\n')

        if file_name.lower() == 'b':
            keep_going = False
        elif file_name in cloud_files:
            # send request to CAM to delete file
            encrypt_and_send(conn, REQ_DEL_FILE)
            res = decrypt_from_src(conn, AS_STR)

            if res == OK:

                # send filename to delete
                encrypt_and_send(conn, file_name)
```

```
                res = decrypt_from_src(conn, AS_STR)

                if res == SUCCESS:
                    print(f'File \'{file_name}\' successfully removed from cloud
group storage')

                    # remove reference in local data structure
                    cloud_files.remove(file_name)
                else:
                    print(f'Something went wrong deleting that file :/')

            else:
                raise Exception(PROTOCOL_EX)

        else:
            print(f'Invalid file name. Please try again.')


def handle_admin_demotion(conn):
    global users, admins

    valid_username = False

    while valid_username is False:

        print(f'Admins available for demotion:')
        for admin in admins:
            print(f'\t{admin}')

        target_admin = input(f'\nPlease enter the name of the admin you wish to
demote. ([B]ack to go back)\n')

        if target_admin.lower() == 'b':
            valid_username = True  # break
        elif target_admin in admins:
            # Move local record into users list
            admins.remove(target_admin)
            users.append(target_admin)

            # Send CAM a message to move user record into users directory
            encrypt_and_send(conn, REQ_RM_ADMIN)
            res = decrypt_from_src(conn, AS_STR)

            if res == OK:
                # send username to demote:
                encrypt_and_send(conn, target_admin)
                res = decrypt_from_src(conn, AS_STR)

                if res == SUCCESS:
                    print(f'User {target_admin} successfully demoted to user')
                    print(f'\tUSERS: {users}')
                    print(f'\tADMINS: {admins}')
                else:  # FAILURE
                    print(f'Unexpected error occurred when promoting user in
CAM :/')
            else:
                raise Exception(PROTOCOL_EX)

        elif target_admin in users:
```

```
            print(f'That user is not an admin!')
        else:  # invalid username
            print(f'That user does not exist. Please try again.')


def handle_user_promotion(conn):
    global users, admins

    valid_username = False

    while valid_username is False:

        print(f'Users available for promotion:')
        for user in users:
            print(f'\t{user}')

        target_user = input(f'\nPlease enter the name of the user you wish to
promote to admin. ([B]ack to go back)\n')

        if target_user.lower() == 'b':
            valid_username = True  # break
        elif target_user in users:
            # Move local record into admins list
            users.remove(target_user)
            admins.append(target_user)

            # Send CAM a message to move user record into admin directory
            encrypt_and_send(conn, REQ_MK_ADMIN)
            res = decrypt_from_src(conn, AS_STR)

            if res == OK:
                # send username to promote:
                encrypt_and_send(conn, target_user)
                res = decrypt_from_src(conn, AS_STR)

                if res == SUCCESS:
                    print(f'User {target_user} successfully promoted to admin')
                    print(f'\tUSERS: {users}')
                    print(f'\tADMINS: {admins}')
                else:  # FAILURE
                    print(f'Unexpected error occurred when promoting user in
CAM :/')
            else:
                raise Exception(PROTOCOL_EX)
        elif target_user in admins:
            print(f'That user is already an admin!')
        else:  # invalid username
            print(f'That user does not exist. Please try again.')


def handle_delete_user(conn):

    global users, admins

    all_users = users + admins
    print(all_users)

    valid_username = False
```

```
    while valid_username is False:
        target_user = input(f'Enter a username to delete. Enter [L] to view a list
of registered users. '
                            f'Enter [B] to go back.\n')

        if target_user == 'L' or target_user == 'l':
            for user in all_users:
                print(f'\t{user}')
        elif target_user == 'B' or target_user == 'b':
            valid_username = True
        elif target_user in all_users:
            # send req to delete to cam:
            encrypt_and_send(conn, REQ_DEL_USER)
            res = decrypt_from_src(conn, AS_STR)

            if res == OK:
                # send the username
                encrypt_and_send(conn, target_user)
                res = decrypt_from_src(conn, AS_STR)

                if res == SUCCESS:
                    print(f'User {target_user} deleted from CAM records ... ')
                else:
                    print(f'Couldn\'t find record for user {target_user} in
CAM :/')
            else:
                raise Exception(PROTOCOL_EX)

            # delete from local group records:
            cur_dir = os.path.dirname(os.path.realpath(__file__))
            group_files_target = os.path.join(cur_dir,
f'group_files\\{target_user}')
            print(group_files_target)

            rmtree(group_files_target)


def encrypt_and_send(conn, msg):
    global public_key_cam

    # convert msg to bytes
    if isinstance(msg, str):
        msg = str.encode(msg)

    ciphertext = public_key_cam.encrypt(
        msg,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    # generate message digest
    signature = private_key.sign(
        ciphertext,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
```

```
        ),
        hashes.SHA256()
    )

    # print(f'\tSending {msg}; ciphertext: {ciphertext}')
    conn.send(ciphertext)

    # print(f'\t\tSending signature: {signature}')
    conn.send(signature)


def decrypt_from_src(conn, as_what):
    global private_key, public_key_cam

    ciphertext = conn.recv()
    signature = conn.recv()

    try:
        verified_ciphertext = public_key_cam.verify(
            signature,
            ciphertext,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )

        # will throw exception if signature not valid
        #   otherwise decrypt
        plaintext = private_key.decrypt(
            ciphertext,
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )

        # cast as string if requested
        if as_what == AS_STR:
            plaintext = plaintext.decode("utf-8")

        # print(f'\tReceived {plaintext}; ciphertext: {ciphertext}')
        return plaintext

    except InvalidSignature:
        print(f'Invalid signature on msg')

    return None


def request_download(conn, filename, username):
    encrypt_and_send(conn, REQ_DOWNLOAD)
    res = decrypt_from_src(conn, AS_STR)

    if res == OK:
        # submit request for file in form [filename.ext]
        encrypt_and_send(conn, filename)
```

```
        res = decrypt_from_src(conn, AS_BYTES)

        if res == str.encode(FAILURE):  # file not found on cloud
            print(f'Unexpected error finding file in cloud.')
        else:
            write_to_user_directory(res, username, filename)
            print(f'File download successful. You file has been saved to
\'group_files\\{username}\\downloads\'.')

    else:
        raise Exception(PROTOCOL_EX)


def write_to_user_directory(file_bytes, username, filename):
    # files that are downloaded are written to group_files/<username>/downloads/
    cur_dir = os.path.dirname(os.path.realpath(__file__))

    path_to_user_dir = os.path.join(cur_dir,
f'group_files\\{username}\\downloads')

    # make parent directories if they don't already exist
    if not os.path.exists(path_to_user_dir):
        os.makedirs(path_to_user_dir)

    path_to_target_file = os.path.join(path_to_user_dir, filename)

    with open(path_to_target_file, 'wb') as target_file:
        target_file.write(file_bytes)
        target_file.close()


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