Assignment 3
TCP Secure Text and Picture File Transfer Application [Part II]
(Due: Midnight on Saturday, December 16th, 2023)

Assignment Setup:

Create a project in Eclipse called `A3`. Download the file `A3.zip` from E-Learning. Extract the zipped files into the `A3` project. You will notice that there are the following files:

1- `A3_test.py`: This is your testing file. This file uses multithreading which starts the server and clients on different threads.
2- There are two output files: `A3_output_client.txt` and `A3_output.server.txt`. Your output should exactly match the given output. Do not edit these files.
3- There are two files to store your output: `A3_student_client.txt` and `A3_student.server.txt`. The testing file will tunnel all of your output to these two files. You do not need to open or close these files.
4- There is one input file called `t1.txt`. There is also `t1_copy.txt` which is empty. When you download the file, it will be stored here.
5- `utilities.py` contains some useful utility functions.
6- `sever_private.txt` and `server_public.txt`: These two files represent the RSA keys for the server.
7- `Passwords.pwl` is a binary file that stores password hashes.

Then copy your `A2.py` file from Assignment 2. Rename the file to `A3.py`. Make sure your name is correctly listed on the top of the file. In this assignment, you are going to edit your A2 solution.

Make sure that all your files are on the same directory level within the project.

Note: If you were unable to complete A2, then you can borrow the solution of one of the students who got it write. In this case, you need to have the following comment on top of the file:

I borrowed the A2 solution from student <name>

Failing to do the above is considered plagiarism and will be treated with strictest measures.

Submission:

When you are done, you only need to submit one file: `A3.py` . Do not submit any other file. **No late submissions are accepted for this assignment**.

Assignment Policies:

1- You can only import the following Python modules:
   a. `socket`
   b. `cryptography`
   c. `utilities`
   d. `hashlib`
2- You are not allowed to use `print` statements. Similar to A2, you need to use the write function to print your output to the designated files.
3- You can write your own private functions. However, you can't have more than three private functions. Every private function should start with an underscore and should be placed at the bottom of the `A2.py` file.

Overview:

In Part I, you have implemented the "Secure Text and Picture File Transfer Protocol (STP). In this assignment, you will enhance that implementation with security measures.

You will work on providing confidentiality, integrity and authentication. Also, you will implement some measures to ensure availability. Several attacks will be highlighted, and you would need to develop counter measures to protect your system.

Changes to Protocol:

The client/server responses should be changed to the following:

Old Value	New Value
<configuration_complete>	<config_done>
<configuration_approved>	<config_valid>
#10:ILLEGAL_COMMAND#	#10:BAD_CMD#
#20:COMMAND_MISSING#	#20:CMD_MISS#
#30:BAD_CONFIGURATION#	#30:BAD_CONFIG#

You will notice that the new values are always 15 characters or less. This will simplify our implementation.

Apply the above changes before proceeding to the next step.

Security Protocol:

The STP protocol achieves security through the implementation of five measures:

1- Signup: In this step, the client and server build trust. The client creates and sends a login password, and the server shares its public key.
2- Authentication (login): The client sends an encrypted message containing its password. The server authenticates the password and either accepts or denies the login.
3- Key Distribution: After the client is authenticated, the client shares with the server a symmetric key. This is done using RSA public key cryptography.
4- Confidential Communication: Once the symmetric key is securely shared, the client and server communicate using encrypted messages.
5- Integrity: The integrity of the downloaded file is verified using hash functions.

Signup:

We will assume that clients have previously registered with the server through a *signup* process.

During the signup, the server sent its public key to each of the clients. Each client stores the server's public key locally in `server_public.txt` file. We will assume that this file is kept integrate, and no adversary changed or will change its contents.

The server stores its own private key in the file: `server_private.txt`. Assume that this file is stored in a secure way and only the server can access it.

During the signup process, each client created a login password. Each client stores its login password securely and does not share it with anyone. Even the server does not have a copy of that password. However, the server stores password hashes for all registered clients in the file: `'passwords.pwl'`. The file is structured in blocks of 48 bytes, where the first 32 bytes is the hash digest for the password, while the last 16 bytes is a *salt*.

Note that in real-life settings the passwords.txt and server_private.txt files should be stored with extra security measures like encryption and salting. However, to

simplify the implementation in this assignment, we will assume that these files are stored unencrypted.

Other than understanding the above process, you do not need to apply any changes to your code to accommodate the signup process.

Key Management at the Client Side:

Each client maintains a tuple containing:

1- `login_password` (str): A strong password containing exactly 15 characters
2- `Symm_key` (bytes): A *Fernet* key which is 44 bytes long.

You do not have to generate the above keys. They will be given to you.

You need to change the `stp_client` prototype from:

```
stp_client(outfile, server, filename=None, commands=None)
```

To the following:

```
stp_client(outfile, server, keys, filename=None, commands=None)
```

Authentication:

Once a client has completed the TCP 3-way handshake, i.e., has successfully connected, it needs to authenticate itself with the server using the following authentication process:

Step 1: Client sends login token

The client encrypts its login password, which is the first element in the tuple keys, using the server's RSA public key. This message is called the *login token*. Since the login password is always 15 characters, the length of this token is always 128 bytes.

Instead of implementing your own RSA scheme, the utilities file defines for you a class called RSA. You would need to load the server's public key then encrypt the login password using the key. Study the docstring of the implemented methods and use the appropriate methods to achieve the above.

Step 2: Client Sends the Login Token

Once the login token is prepared, the client needs to send it to the server. The message is in bytes, so no encoding is required.

You would need to define a new function called:

```
login( outfile, client_socket, login_password)
```

such that:

- `outfile`: is the name of the output file that will be used by the write function.
- `client_socket`: a reference to the client socket object.
- `login_password`: the login password which is the first element in the `keys` tuple.

You would need to edit your `stp_client` function such that after it successfully connects and before it sends the commands, it should call the `login` function.

The `login` function performs the following:

1- Loads the server's public key
2- Encrypts the `login_password` using the above public key.
3- Send the login token to the server.
4- Wait for the server's `response` which is always 13 bytes.
5- Return the `response` to the `stp_client` function.

Step 3: Sever Authenticates the Login Token

Upon the successful connection of the client, the server passes the client's connection to the `handle_client` function (this was done in A2). The first thing that should be done in the `handle_client` is to authenticate the user. This should be done using the function: `authenticate_client(outfile,connection)` which performs the following:

1- It receives the login token from the client which is always 128 bytes.
2- Loads the server's private key
3- Decrypts the login token using the above key. The result is the login password.
4- Hash the login password
5- Check if the hash exists in the `'passwords.pwl'` file.
6- If the given password hash matches one of the passwords in the file, the authentication is successful. Otherwise, the authentication fails.

The `utilities` file defines a class called `Password`. You can use the static method `Password.hash_password` to get the hash digest of any string. Remember, the passwords file is a binary file that contains blocks of 48 bytes. For each of these blocks, the first 32 bytes are for the password hash digest and the last 16 bytes are for the password salt. You would need to pass both to the `hash_password` method.

Step 4: Complete Authentication

If the client is authenticated, the server sends the message: `<user_authen>`. If the client is not authenticated, the server sends the message: `<user_denied>`. This needs to be done inside the `handle_client` function. Note that the above response is not encrypted and it is always 13 bytes.

If the server sends `<user_denied>`, it closes the client's connection and returns, i.e., waits to accept another client. At the client side, if it receives `<user_denied>` it also closes the connection and exits.

If the server sends `<user_authen>`, then it should keep the client's connection open waiting for the next message. At the client side, if it receives `<user_authen>` then it should move to the next step, which is sharing the symmetric key.

At this stage, it would be best to run `task1()` in the testing file. This testing scenario focuses on failed user authentication.

Secure Key Distribution:

In order for the client and server to communicate in a confidential manner, they need to exchange a symmetric key in a secure manner.

The symmetric key is a `Fernet` key of type *bytes* and of size 44 bytes. The `Fernet` module is defined under the `cryptography` library. Documentation of this library is available at: https://pypi.org/project/cryptography/ and you would need to use pip to install it.

Each client maintains a unique symmetric key. You do not need to worry about generating this key. The symmetric key is provided as the second element in the tuple `keys`, which is an argument passed to the `stp_client` function.

In order to securely exchange this key, the RSA scheme will be used. Assuming that the user has been authenticated, below are the steps of how to proceed:

1- The client encrypts `symm_key` using the server's public key. This message is exactly 128 bytes.
2- The server receives the key and decrypt it using its private key. The server does not perform any validation to the received key.

3- The server encrypts the response: `<key_received>` using the received symmetric key and sends it to the client. The length of this message is going to be exactly 100 bytes.
4- The client receives and decrypts the response. If it is `<key_received>`, it starts sending commands. Otherwise, it disconnects.

To achieve the above, at the client side, implement the following function:
`send_symm_key(outfile, client_socket,symm_key)`

which performs the following:

1- Load the server's public key.
2- Encrypt the symmetric key using the above public key. This is called the `symm_token` and is 128 bytes long.
3- Send the `symm_token` to the server
4- Waits for the server's response which is 100 bytes
5- Create a `Fernet` object using the symmetric key
6- Decrypts the server's response using the `Fernet` object.
7- If the response is `<key_received>` the function returns `True`, otherwise, the function returns `False`.

The above function should be called by the `stp_client` function.

At the server's side, implement the following function:
`get_symm_key(outfile, connection)`

which performs the following:

1- Receives the `symm_token` from the client which is 128 bytes.
2- Loads the server's private key
3- Decrypts the received `symm_token` using the above private key
4- Create a `Fernet` object using the symmetric key
5- Encrypts the message `<key_received>` using the Fernet object. The output is a 100 bytes message.
6- Send the above encrypted message to the client
7- Return the symmetric key to the calling function which is `handle_client`.

Confidential Communication:

Once the client has been authenticated and has securely shared a symmetric key with the server, both parties would be ready to communicate in a confidential

manner. This is going to be done by encrypting messages using the shared Fernet symmetric key.

For simplification purposes, encryption would be applied only to the commands and responses, but not to the actual file contents. This means, after authentication, everything will be encrypted except the file contents.

To further simplify the process, assume that the commands and responses will be always less than 16 bytes. This guarantees that the length of the encrypted messages is always exactly 100 bytes.

To allow the above, you need to modify the implementation of some functions.

At the client side, change the `send_commands` function to be:

```
send_commands(outfile, client_socket, commands, symm_key)
```

The function should encrypt each command using the given symmetric key before sending it.

Also change the function get_config_response to be:

```
get_config_response(outfile, client_socket , symm_key)
```

the received message needs to be decrypted.

At the server's side, change the `rceive_commands` function to be:

```
receive_commands(outfile, connection, symm_key)
```

The function should decrypts the received messages and encrypts the responses.

Data Integrity:

To ensure that the sent file is received without changes, hash functions will be used to verify the integrity of the file.

Start by implementing the function:

```
compute_file_hash(filename)
```

The function takes a filename which is assumed to be a text file. The function reads the contents of the file, removes all whitespace characters then computes the MD5 hash digest of the contents.

Use the function `md5` which is defined in the built-in Python library `hashlib` to compute the hash digest.

The md5 hash digest is always 128 bits, i.e., 16 bytes.

At the client side, inside the `upload_file` function and before uploading the file contents, send the hash digest then send the file contents.

At the server side, inside the `download_file` function assume that the first 16 bytes relate to the hash digest and the remaining bytes are the file contents.

After completing the download, compute the hash digest for the downloaded file and compare it with the received hash value. If they match, then integrity is verified.

Good luck!