CS432 – Computer and Network Security
Spring 2024-2025 – In-lab assignment

- No collaboration is allowed. You are not allowed to ask and get help from your classmates. Any such activity will directly result in failure in this lab.
- All cell phones must be totally switched off if you are not using them as a modem.
- Any type of online communication via email, DM, WhatsApp, etc. with another human being will be treated as plagiarism.
- IP-sharing is strictly prohibited, you should connect to VPN using your own SU account credentials only.

Your task

Your task in this lab is to implement the **client-side** of a secure sequential communication protocol by using Python sockets and the cryptography library introduced in the labs. IP and port number of the server will be provided by your TAs on the board or on Zoom chat and your client will connect to it. You do not need to do anything for the server part. You will only implement the client part. After the connection phase, you are expected to follow the steps below. Your solution will not have any graphical user interface (GUI) so that you can hardcode the IP and the port number in the code (put them as variables inside the code) for connecting.

Since the main aim of the lab is to help gain practical experience with cryptographic operations, you may or may not use try-catch blocks (they will not be considered as part of the grading). However, it would be for your own sake if you use them to see the failures in connections and operations. **Now, start by extracting In-lab-exercise.zip and open the project inside it.** The project contains the initial part of your application. The keys and the plaintexts that will be used during the exercise are loaded automatically by reading the corresponding text files located under the `keys` folder. The project code also contains the helper functions that are ready to be used. In addition, the initial connection code to the server is provided. Please set the IP address and the port number accordingly and then you can connect. Do not forget that if you are connecting from outside the campus you need to have a VPN connection to connect successfully.

The scenario to be implemented is described below. Since you need to implement your solution step by step, an error in a step may affect the next one. Therefore, **the server will send you a plaintext message with a byte size of at most 64 if you send an incorrect or unexpected message to the server.** After each step, you can check if you get a plaintext error message or not by using the socket receive function. Note that this is just for failure cases (except for the last

success message to be received). If your client is acting properly you are not going to receive a plaintext error message but rather the expected message indicated in the corresponding step.

When receiving a message from the server you can take the substring of the bytes object as (if the size of the bytes object to be received is at most 64):

```
incomingMessage = clientSocket.recv(64)
first_10_bytes = incomingMessage[:10]
remaining_bytes = incomingMessage [10:]
```

Be careful that the bytes object size to be received is not always 64 or below. They are specified in the respective steps. Read carefully.

Some other conversions you can use:

String to Bytes Object → `bytes_object = my_string.encode()`
Bytes Object to String → `my_string = bytes_object.decode()`
Bytes Object to Hex String → `hex_string = bytes_object.hex()`

Keep in mind that cryptographic functions require byte objects as input. Therefore, if your input is in string format, ensure you convert it appropriately before passing it to the corresponding cryptographic function.

After the connection is established, the scenario that you will implement is as follows:

1) Before you (client) send anything to the server, it will send you an encrypted integer identifier number (encrypted token). You should receive it using the client socket and decrypt using **AES-128 in CBC mode** with the 128-bit symmetric key and the IV given in the first and second line of the **AES128key_IV.txt file respectively (loaded in the code already, you don't need to read it)**. For more information on using AES-128 in CBC mode, please visit https://pycryptodome.readthedocs.io/en/latest/src/cipher/classic.html#cbc-mode. Note that the BlockSize and the FeedbackSize are 128-bit for AES. When receiving through the socket, you can assume that the size of the bytes object to be received is 16 bytes. After the decryption, the resulting 4-digit token (do not store it as an integer, keep it as a bytes object) will be used in step 2.

2) Compute the hash of the token bytes object using the hash function SHA256.

   Example:
   - Token: `b"6357"`

- Compute the hash of the token string: `SHA256(b"6357")`

3) Using the hash digest computed in Step 2 as a 256-bit key, generate the HMAC of your full name **(without any Turkish characters). The hash algorithm to be used in HMAC generation is SHA512.** Then, you will concatenate the HMAC value as a **hexadecimal string** to your full name using the delimiter colon (':') and send it to the server through the socket. Do not forget to convert the resulting string to a bytes object before sending.

   Message to send → Name Surname:HexStringHMAC

4) If the HMAC is verified by the server, it will send the client a ciphertext message encrypted with the RSA cryptosystem. You should receive (through the socket) and decrypt it using the public/private key pair which is given in **RSA2048key.txt** file (already imported). When receiving through socket, you can assume that the length of the bytes object to be received is 256 bytes. After the decryption, store the decrypted result as a bytes object to be used in step 5.

5) The message obtained in step 4 consists of two parts: **256 bits AES** encryption key and **128 bits IV**. The encryption key is the bytes object elements [0…31] while the IV is going to be the bytes object elements [32…47]. Use this encryption key and the IV to encrypt the plaintext given in the file **plaintext.txt** (already imported in the project) **with AES-256 in CFB mode.**

6) After the encryption, sign the ciphertext using the RSA cryptosystem. The public/private key pair that is going to be used for signature generation is the same 2048-bit key given in the **RSA2048key.pem** file. It is already imported in the project provided in the In-lab-exercise.zip file. Also, the hash algorithm to be used in signing is **SHA256**. Then, you will concatenate the **hexadecimal string representation** of the generated signature to the **hexadecimal string representation** of the ciphertext using the delimiter colon (':') and send it to the server through the socket.

   Message to send → HexStringCiphertext:HexStringSignature

7) If the signature is verified and the ciphertext is decrypted correctly by the server, it will send you a plaintext message as a response indicating whether your application was successful or not. Print this message onto the console. The socket must be closed automatically after you get this message.

After you get the success message from the server, the steps that you should follow are:

1) Check whether your full name is in the successful attempts table (Webpage will be indicated by your TAs).
2) Do not forget to submit your application to the assignment “In-lab Exercise” in a zip file on SuCourse+ under the Lab Materials tab. (Name your zip file as **"yourSuNetusername_lastname_othernames.zip"**)

Please follow the steps carefully and verify whether your attempt was successful (check your name on the successful attempts webpage). Also, do not forget to submit your client application. Otherwise, we cannot grade your work!