CS432/532 Computer and Network Security Spring 2025 - Term Project

Secure Peer-to-Peer Communication Application

Project Step 1 Due: May 8, 2025, Thursday, 23:55 (to be submitted to SUCourse)

Project Step 2 Due: May 15, 2025, Thursday, 23:55 (to be submitted to SUCourse)

Project Step 3 Due: May 22, 2025, Thursday, 23:55 (to be submitted to SUCourse)

Demos: Time and Schedule will be determined later

Submission and Rules

- **CS432 students** can work in groups of at most 2 people in all three project steps. We will collect group information before step 1, but we will not group people ourselves; if you cannot find a group, you can do the project alone. The project requirements will **not** change with the group size.
- **CS532 students** have to work **individually** on the project.
- Equal distribution of the work among the group members is essential.
- All group members should appear at the demos.
- All group members should submit the complete project in each step to SUCourse+. No email submissions, please. Make sure that all group members submit the same version. If a group member does not submit, we will assume that this person has been kicked out of the group and he/she will receive zero. On the other hand, if you allow a group member to submit the same code as you, then you acknowledge his/her contribution. Consequently, the group members who submit step 1 together should continue to work together for steps 2 and 3 as well.
- The submitted code will be used during the demo. No modification on the submitted code will be allowed.
- Submission steps:
    - Create a folder named ***XXXXX_Surname_Name***, where *XXXXX* is your SUNet ID. Put your *source files inside*.
    - Compress your *XXXXX_Lastname_Name* folder **using ZIP**.
- You will be invited for a demonstration of your work. Date and other details about the demo will be announced later.
- For each step, you are allowed to submit late with a penalty. The maximum possible grade you can receive for the given step decreases based on when you submit it. Below, you can find the grading information for late submissions for each step:
    - Step 1:

        - Submitted 1 day late (i.e., May 9): 90%
        - Submitted by Step 2's deadline (i.e., May 15): 70%
        - Submitted by Step 3's deadline (i.e., May 22): 50%
    - Step 2:
        - Submitted 1 day late (i.e., May 16): 90%
        - Submitted by Step 3's deadline (i.e., May 22): 70%
    - Step 3:
        - Submitted 1 day late (i.e., May 23): 90%

Programming Rules

Your application should have a graphical user interface (GUI). **It is not a console application!**

Your code should be clearly commented. This may affect up to 5% of your grade.

Your program should be portable. It should not require any dependencies specific to your computer. We will download, compile, and run it. If it does not run, it means that your program is not running. So, do test your program before submission.

In your application, when a window is closed, **all threads related to this window should be terminated.**

Introduction

In this project, you will implement a client-side application that establishes *secure* and *authenticated* peer-to-peer communication. You will communicate with a server that acts as an authentication server and a key distribution center, which will be provided by us.

The *Client* module:

- ➢ Registers and authenticates with the server.
- ➢ Connects to the server to establish a master key for secure communication.
- ➢ Uses the master key to obtain an integrity key from the server, which will be used for session key generation.
- ➢ Establishes a session key for communication with another client.
- ➢ Generates the necessary cryptographic materials from the session key for peer-to-peer communication.
- ➢ Engages in peer-to-peer communication with the other client.

Roughly, the enrollment of the clients and client authentication to the server constitute the first step of the project. The integrity key distribution is the second step, and the rest is the third step of the project.

You should design and implement a user-friendly and functional graphical user interface (GUI) for your client program. In three of the steps, all activities and data generated by the server and the clients should be reported in text fields on their GUIs. These include (but are not limited to):

- RSA public and private keys in hexadecimal format,
- AES keys and IVs in hexadecimal format,
- Random challenges and responses calculated for the challenge-response protocol runs in hexadecimal format,
- Digital signatures in hexadecimal format,
- Verification results (digital signatures, HMAC verifications, etc.),
- Message transfer operations and message texts,
- HMAC values, session keys, etc. in hexadecimal format,
- And all other details that are needed to follow the flow of execution. We cannot explain all the details of the things that need to be shown on the GUI within the project document; thus, you may take this as the golden rule: "**if we cannot follow what is going on, we cannot grade**". And please do not ask us if you should display this or that; please use your own reasoning, thinking of the golden rule above. Another related rule is that we will not follow the execution of your code in the debug console; we will grade using whatever you put on the GUIs.

Ideally, your application should feature two separate windows: a **Message Window** and a **Debug Window**. The Message Window should display the protocol-level messages exchanged during normal operation, such as client communications and protocol status updates. In contrast, the Debug Window should provide a more detailed insight into the internal operations of your application. This includes information such as ciphertexts received, keys and IVs used for decryption, decrypted plaintext, HMAC values, verification results, and any other relevant data. The Debug Window is essential for understanding the full execution flow of your program. Remember, we will evaluate your project solely based on the information presented in these GUI windows — **not** the debug console — so ensure that all critical details are clearly displayed within the GUI.

Project Step 1 (Due: May 8, 2025, Thursday, 23:55)

In the first step of the project, the client performs *enrollment*, *user authentication, disconnection,* and *deletion* operations. You will implement enrollment and authentication

protocols between the client and the server. The initial contact of each client with the server is the *enrollment* process. Only the enrolled users (clients) can be authenticated by the server; this authentication is done via a challenge-response protocol. Enrollment, authentication, and disconnection processes are explained in the following subsections.

We will be providing you with two RSA-2048 public-private key pairs in the project pack. These key pairs will be used by the server; one for encryption/decryption and one for signing/verification purposes.

Enrollment and Authentication Phase

Before doing anything, a client has to enroll and authenticate itself to the server in a secure way if he/she has not done it already. In this phase, you will implement a secure login protocol for client authentication to the server. This will be achieved through a challenge-response process where the server sends a unique code to the client's registered email address, and the client must then provide this code to complete the authentication.

Before making the connection to the server, the client program should load the server's public keys (one for encryption and the other for signature verification) from the file system (these keys are also provided in the project pack). Then, the user enters the IP address and the port number of the server (provided below).

IP address: harpoon1.sabanciuniv.edu (10.3.0.239)
Port: 9999

The client should connect to the server using the IP address and the port number entered. If the connection is not successful, it should show an error message and ask the user to enter the port number and IP address information again. If the connection is successful, the user can start communicating with the server.

To initiate the authentication process with the server, the client must first send the string "auth" to the server. Upon receiving this message, the server responds with a message stating "Successfully starting auth flow.", along with a signature generated using the server's RSA private key. The signature and the message are attached together (i.e., concatenated) in the format <signature><message> (i.e., $\mathbf{RSA_s(M) \,||\, M}$). The client must verify the authenticity of this message by validating the signature.

In this project, each student is allowed to enroll/authenticate up to five times using the same student ID. To track each enrollment attempt, the client must create a unique username for each session. After successfully verifying the server's signed message, the client proceeds by sending their 5-digit student ID and the generated username to the server in the format <ID><username>. If the provided student ID is invalid, the server responds with an error message signed in the same <signature><message> format,

indicating the nature of the error. If the username is duplicated again, the server returns an error message. If the student ID and username are valid, the server records the ID and the username. Then, it sends a success message, again signed in the same format, confirming that the enrollment has been completed successfully.

In addition to the success message, the server also sends an email to the client's registered Sabancı University email address. This email contains the student's ID and username, along with a randomly generated six-digit code.

Once the client receives a signed response from the server, it must first verify the server's signature using the corresponding public key. If the verification is successful, the client examines the content of the response to determine whether it indicates an "error" or "success". If an "error" response is received, the client may attempt authentication again by starting the "authentication step" again. If a "success" response is received, the client must check its registered email account to retrieve the code sent by the server and proceed with the next steps of the authentication process using this code.

After receiving the code, the client initiates the code verification process. To begin, the client first sends the string "code" to the server (not the 6-digit code itself, but a string that says "code"). Upon receiving this message, the server responds with either a success or error message, signed using the server's RSA private key. After validating the server's signature and success message, the client proceeds with the next steps.

The client computes the SHA-512 hash of the received code. It then generates random 32-bytes. The most significant 16-bytes (i.e., upper half 128-bit) will be the Master key (K_M), and the remaining 16-bytes (i.e., lower half 128-bit) will be the IV. In other words, both AES key and IV will be 128 bits." Then the client should encrypt this 32-byte key-iv pair using the server's RSA public encryption key. Afterward, the client concatenates the hash value, the encrypted **(K_M || IV)**, the student ID and username into a single message in the format <hash of the code><**RSA_E (K_M || IV)**><student ID><username>, and sends it to the server.

Please note that you should store the K_M and IV values that you generated for your account, as you will be using these values again to communicate with the server in the following steps of this project.

The server checks whether the provided hash of the code matches the expected value. If the hash is correct, the server sends a positive acknowledgment message, such as "Authentication Successful", signed with its RSA signature key. If the hash does not match, the server responds with a negative acknowledgement message, such as "Authentication Unsuccessful", also signed accordingly.

Upon receiving and validating a positive acknowledgment, the client is considered successfully authenticated.

When the client verifies the signature on the acknowledgment, he/she makes sure that the sender of this acknowledgment message is the valid server. If the response is verified, then the client checks whether the response is "error" or "success". If it is a failed verification, the client may try again.

Disconnection

A client may disconnect from the server by pressing a disconnect button or by closing the window. After disconnection, the same user may want to log in again by running a brand-new enrollment and authentication phase.

If the server disconnects, the connected clients must understand this and get disconnected as well. Later, when the server opens again, the clients can make fresh connections.

The client application must not crash while handling disconnections.

Deletion

Since each student is limited to a maximum of five accounts, you may want to delete an existing account if you have lost your master key generated for that account, or if you want to free up a slot. To initiate the deletion process, the client must first send the string "delete" to the server. Upon receiving this, the server responds with a signed success message. If the client successfully verifies the server's signature, it can proceed with the deletion.

Next, the client must send their student ID and the username of the account they wish to delete, in the format <ID><username>. If either the ID or the username is incorrect, the server will respond with an appropriate error message. If the information is valid, the server will send a success message and email a removal code (rcode) to the student's Sabancı University email address.

Once the client receives the rcode, it must send the string "rcode" to the server to initiate the verification process (not the 6-digit rcode itself, but a string that says "rcode"). The server responds with another signed success message. After verifying this signature, the client completes the deletion process by sending a concatenation of the rcode, student ID, and username to the server.

If all steps are completed correctly, the server returns a final success message, and the account is deleted.

Some Caveats

In the project, some random numbers will be generated. These random numbers must be cryptographically secure.

The server expects everything to be encoded as a string for step 1. For example, if you want to encode your student ID, encode it as `str(34700).encode()` or `b"34700".` The same applies to the codes.

Every message you receive from the server is signed, and you need to verify these signatures even if it is not stated explicitly.

We will test various failed authentication cases (such as wrong keys, modified messages, etc.) during the demos, so please test your own codes accordingly.

So far, it should be clear that the client needs to load the server's RSA public keys from files provided. For these purposes, the client needs to browse the file system to choose the key file(s).

The listening port number of the server must be entered via the server GUI. Clients connect to the server on the corresponding port and identify themselves with IDs. There might be one or more different clients connected to the server at the same time. Each client knows the IP address and the listening port of the server (to be entered through the GUI).

This step may be tested with one server and multiple clients in the Demos.

Mails may come from the CS411 mail address. Don't worry, you are working on the correct course's project. We just don't have an email address for CS432/532.

Provided RSA Keys

- *server_enc_dec_pub.pem*: This file includes the server's RSA-2048 public key for encryption operations in PEM format.
- *server_sign_verify_pub.pem*: This file includes the server's RSA-2048 public key for signature verification purposes in PEM format.

Project Step 2 (Due: May 15, 2025, Thursday, 23:55)

In the second step of the project, the integrity key distribution mechanism will be implemented on top of the first step.

During the previous phase, following a secure authentication, the client created a master key with the server. This allows clients to send secure and authenticated messages to the server. When a client wants to initiate peer-to-peer messaging with another client, it first

communicates with the server to establish an integrity key (K_I) between itself and the intended client. In this process, the server acts as both a key distribution center and an authentication server.

Integrity Key Distribution

Assume that Client A (initiator) wants to initiate communication with Client B (responder). To do this, Client A (and Client B) must first obtain a K_I from the server. K_I is used to verify the integrity of the session key, K_S, which will later be employed for message encryption purposes. The integrity key K_I distribution protocol is as follows (Note that || stands for byte concatenation. The clients are assumed to parse the concatenated messages accordingly):

1. Client A communicates with the server by sending a message formatted as: $\mathbf{ID_A}$ || **";"** || $\mathbf{username_A}$ || **";"** || $\mathbf{ID_B}$ || **";"** || $\mathbf{username_B}$ || **";"** || $\mathbf{N_1}$. Here, ID_A refers to Client A's ID, while ID_B denotes the ID of Client B, and N_1 is a random nonce value generated by Client A. You can assume the nonces to be 128-bit numbers. Note that, as the length of the usernames can be variable, a semicolon (";") is used as a delimiter in the message byte string.
2. Once the server receives the request from the initiator, it checks whether or not Client B is authenticated to the server. If Client B is not authenticated, the server aborts the protocol. Otherwise, it generates a 128-bit integrity key K_I. The server performs the operations below and sends a message to Client A:

   C_1= AES-CBC-ENC(K_A, [K_I || ID_A || ID_B || N_1])

   C_2 = AES-CBC-ENC(K_B, [K_I || ID_A])

   Message sent to Client A: $\mathbf{C_1}$ || $\mathbf{C_2}$ || **SID**.

   Here, K_A refers to the master key between the server and Client A, K_B is the master key between the server and Client B, and SID is a random 16-byte session ID chosen by the server. The IVs used for the encryption of C_1 and C_2 are denoted as IV_1 and IV_2, respectively. These IV values are generated as follows: IV_1 = SHA256(IV_A||SID)[:16] and IV_1 = SHA256(IV_B||SID)[:16]. Here, IV_A and IV_B are Client A and Client B's static IV values, which were determined during the generation of the master keys. The [:16] notation refers to the upper 128-bits of the hash value. Please note that AES-CBC-ENC refers to AES-128 encryption performed in CBC mode.
3. After receiving this message, Client A decrypts C_1 and learns K_I. It first verifies if the nonces match. If not, it aborts the protocol. If the nonces match, it relays $\mathbf{C_2}$ || **SID** to Client B.
4. Client B decrypts C_2, which is received from Client A. It generates a random nonce N_2, and sends the following message to Client A: **AES-CBC-ENC($\mathbf{K_I}$, [$\mathbf{N_2}$]) || $\mathbf{IV_3}$**.

IV_3 is a random IV value generated for the given encryption and sent along with the encrypted message.

5. Client A decrypts this message and sends the following message back to Client B: **AES-CBC-ENC(K_I, [N_2+1]) || IV_4**. IV_4 is a random IV value used for the given encryption. The last two steps of the protocol are performed to ensure that the message sent in Step 3 is not replayed.
6. After Client B receives the message, it decrypts it and checks whether the received value really equals N_2+1. If so, the K_I is established between Client A and Client B. If not, Client B will silently abort, meaning that it will not continue with the rest of the protocol.

Please note that this protocol follows the same steps as the key distribution protocol presented in the "04-Key Distribution and Authentication" slides, page 8.

Furthermore, remember that for you to test integrity key distribution, there should exist two separate accounts that are authenticated to the server. You can test integrity key distribution between two of your own accounts (i.e., two different accounts you created using your own student ID) or between you and any other account created by another person in this class. Note that both accounts should be authenticated to the server first.

In order to be able to connect to the server, Harpoon1, you have to be connected to Sabanci University's network. However, if you are connected to Sabanci University's network, client-to-client communication may be restricted due to network policies. To work around this limitation, you are expected to simulate client-side communication locally by running two separate client applications on your own machine. Each client should use a different account, and you can establish the integrity key between them. Since this setup is local, both clients are assumed to communicate over the localhost interface (e.g., 127.0.0.1), and the port numbers used are predefined and known to both clients.

Project Step 3 (Due: May 22, 2025, Thursday, 23:55)

In the third step of the project, session key generation and secure messaging mechanisms will be implemented on top of the second step.

To ensure forward secrecy in their peer-to-peer messaging, clients generate a session key (K_S) at the beginning of each communication session using the Diffie-Hellman protocol. The integrity key (K_I) is employed to verify the integrity of the messages exchanged during the Diffie-Hellman protocol. If the integrity checks are successful and the K_S is established, clients will use K_S to generate the necessary communication material. This material will be used to encrypt and validate the integrity of their peer-to-peer messages throughout the

session. Detailed information regarding the session key generation and secure peer-to-peer messaging is provided in the following subsections.

Session Key Generation

Once Client A and Client B have securely established K_I, the next step is to generate a session key, denoted as K_S. To ensure forward secrecy, we assume that the clients communicate in sessions, with each peer-to-peer message in a given session encrypted using key materials that are generated from that session's key. Any of the clients, when prompted, can initiate a new session (and a new session key) to be established. Your client GUI should include a button for new session initiation. If pressed by either Client A or Client B, the current session should be immediately terminated, and the session key generation protocol should be restarted from the beginning.

The clients use a protocol that is similar to the ephemeral Diffie-Hellman (DH) protocol to establish K_S. K_S will be a 256-bit key, and an elliptic curve-based (ECC) DH protocol will be used to generate this key. For DH using ECC (also known as ECDH), you can refer to Python's pycryptodome library's documentation. The session key generation protocol works as follows:

1. Client A and Client B each generate a pair of public and private keys, denoted as (PU_A, PR_A) and (PU_B, PR_B), respectively. Note that, to generate the keys, the clients should agree on the same ECC curve. You should select the curve as 'p256'.
2. Both parties then use the integrity key, K_I, to generate an HMAC of their public key. The hash algorithm used for HMAC is SHA-256. They concatenate the generated HMAC value to their public keys (e.g., PU_A || $HMAC\_256_{K\_I}(PU_A)$), forming a message to be sent to the other client.
3. Upon receiving the message, the recipient client verifies the integrity of the received public key. To do so, first, it generates the HMAC of the received public key. Then, it checks whether the HMAC value it generated matches the HMAC value received from the other client.
4. If the integrity checks are successful and the K_S is securely established. The clients send an "ACK" message to each other, concatenated with an HMAC value calculated over the message using K_I. If the integrity check fails for any client, that client sends a "BAD HMAC" message, which is also concatenated with an HMAC of the message generated using K_I (i.e., ACK||$HMAC\_256_{K\_I}$(ACK) or BAD HMAC||$HMAC\_256_{K\_I}$(BAD HMAC)). You can assume that the ACK and BAD HMAC messages are strings that are encoded as byte objects.

If the clients establish the K_S securely, they need to use this key to generate their key material (i.e., encryption and integrity keys, and IVs), which will be directly used in their

communication. Otherwise, the K_S generation protocol must be repeated from the beginning.

Generating the Communication Keys from K_S

Once the session key K_S has been securely established, both clients must derive the necessary cryptographic materials from K_S to enable secure communication. This includes generating distinct encryption and integrity keys, as well as IVs, for each direction of communication (i.e., A→B and B→A).

Following the standard practices in protocols such as TLS, each party generates separate keys for sending and receiving messages. In order to achieve that, a typical approach is to apply a key derivation function (KDF) or an extendable-output function (XOF) to K_S. For your implementation, you should use the SHAKE128 XOF function, which can produce outputs of any desired length. As SHAKE128 is capable of producing outputs of arbitrary length, a client should invoke a single call to the SHAKE128 function, giving K_S as the input, to produce a continuous block of 96 bytes. Then, the resulting 96 bytes should be split into six 128-bit pieces to generate the communication key materials provided below:

- $\mathbf{EK_{AB}}$: encryption key for messages sent from A to B. This is a 128-bit key for AES-128 CBC encryption.
- $\mathbf{EK_{BA}}$: encryption key for messages sent from B to A. This is a 128-bit key for AES-128 CBC encryption.
- $\mathbf{IK_{AB}}$: integrity key for messages sent from A to B. This is a 128-bit key for HMAC calculation.
- $\mathbf{IK_{BA}}$: integrity key for messages sent from B to A. This is a 128-bit key for HMAC calculation.
- $\mathbf{IV_{AB\_0}}$: initial IV for the messages sent from A to B. This is a 128-bit value used during AES-128 CBC encryption.
- $\mathbf{IV_{BA\_0}}$: initial IV for messages sent from B to A. This is a 128-bit value used during AES-128 CBC encryption.

Note that a single client should not run SHAKE128 multiple times on the same K_S value to generate each of the six key materials. Each call to the function with the same input would repeat the output stream and produce the same values for each of the six key materials, which is not something we want.

Each client has to run the SHAKE128 function over the shared K_S to produce these key materials. This ensures that both clients have the necessary keys and IVs to encrypt and authenticate their outbound messages, and decrypt and verify the integrity of the inbound messages.

Please note that, in order to ensure security using AES-CBC, each message must be encrypted using a fresh and unique IV. For this reason, the clients must derive a unique IV per message. Assume that Client A encrypts and sends a message to Client B. The value of IV for the initial message is IV_{AB_0}. However, for the second message that Client A wants to send to Client B, the IV should be updated as: $IV_{AB\_1} = SHAKE128(IV_{AB\_0})$. Note that both Client A and Client B should perform this update, as one uses this IV for encryption and the other uses it for decryption.

Secure Peer-to-Peer Messaging among Clients

On the client GUI, the messaging feature should be enabled if and only if the clients securely establish a session key (and of course, the communication keys with that too).

When a client wants to send a message (assume that Client A is sending the message, and Client B is receiving), it should encrypt its message using the 128-bit AES algorithm in CBC mode using the EK_{AB} and IV_{AB}. Then it should take the HMAC of this encrypted message using the IK_{AB}. The hash algorithm used in HMAC is SHA-256. The initiating client concatenates the encrypted message and its HMAC together and sends it to the responding client.

When a responding client receives the message from the initiating client, it decrypts the message using EK_{AB} and IV_{AB}, then, checks the validity of the HMAC using IK_{AB}. If everything is all right, the decrypted message is displayed in the GUI; otherwise, the message is discarded, and an error prompt is displayed in the GUI.