پروژه عملی - پیام‌رسان امن

امنیت داده و شبکه

میکاییل قربانی

پانیذ حلواچی

آرمان بابایی

Communication with Server

When communicating with the server:

1. **Including Public Key**: The public key of the user is included alongside the message content. This allows the server to identify the user and establish a secure communication channel.
2. **Message Signing**: The message content is signed with the user's own public key. This ensures the integrity and authenticity of the message. By verifying the signature with the user's public key, the server can confirm that the message was indeed sent by the user and has not been tampered with.
3. **Sequential Number**: A common sequential number is added to each message, representing the order of messages exchanged between the server and the user associated with that specific public key. The sequential number starts from an initial value chosen by the message sender and increases by one in each subsequent message. This allows the server to keep track of the order of messages and detect any missing or repeated messages.
4. **Message Encryption**: After including the public key, signing the message, and adding the sequential number, the entire message (including the public key, signed content, and sequential number) is encrypted with the server's public key. This ensures the confidentiality of the message during transmission and prevents unauthorized access.

When communicating with each client:

1. **Message Signing**: The server signs the content of the message using its private key. This provides authentication and integrity to the messages sent by the server. The clients can verify the signature using

the server's public key, confirming the origin and integrity of the message.

2. **Adding Sequential Number**: Similar to the server's interaction with the user, the server adds the common sequential number to the message. This ensures that the order of messages between the server and each client is maintained consistently. The sequential number helps in identifying the position of the message in the conversation and avoids any ambiguity in message sequencing.

3. **Message Encryption**: After signing the message and adding the sequential number, the entire message is encrypted with the client's public key. This ensures that only the intended client can decrypt and access the message, ensuring confidentiality.

By utilizing the sequential number, the design effectively addresses the security requirement of preventing replay attacks (security requirement number 9). The server can identify and discard any repeated or out-of-sequence messages, thus preventing an attacker from reusing or manipulating previously sent messages.

Due to message signing, messages involving the server in the communication are not subject to non-repudiation issues (security requirement number 5). The server's messages can be verified to have originated from the server itself, ensuring that the server cannot deny sending a particular message.

Since all message contents are signed with the client's public key, an attacker cannot forge or modify the content without detection. This provides authentication and ensures that all messages are genuine and trusted (security requirement number 4).

The sequential nature of the counter used in the messages ensures message consistency (security requirement number 3). The order of messages is maintained, allowing for proper organization and understanding of the conversation between the server and the clients.

Sign Up

The SignUp process in the messenger app involves user registration, including the selection of a username and password, as well as the generation of a public-private key pair. The process is designed to ensure secure user authentication and aligns with the project requirements. Here's an extensive expansion of this part of the report:

1. **User Registration**: When a user wants to sign up for the messenger app, they provide a username, password, and generate a pair of public and private keys. The username serves as a unique identifier for the user, while the password is used for authentication and securing the user's account.
2. **Key Pair Generation**: The user generates a public-private key pair, typically using asymmetric encryption algorithms such as RSA or Elliptic Curve Cryptography (ECC). The private key is kept securely on the user's device and should never be shared with anyone, while the public key is used for various cryptographic operations, including secure communication and message encryption.
3. **Sending User Information**: To complete the sign-up process, the user sends their chosen username, password, and public key to the server. Additionally, a nonce is included in the request. A nonce is a unique and

random number generated by the client to prevent replay attacks and ensure the freshness of the request.

4. **Server Storage**: Upon receiving the user information, the server securely stores the relevant data. The password is not stored directly but rather as a hashed value along with a randomly generated salt. The salt is a random string of characters that adds an additional layer of security by making it computationally expensive for attackers to crack the password hashes through brute-force or dictionary attacks.

5. **Hashing and Salted Password Storage**: The server computes the hash of the password by concatenating the salt with the password and applying a secure hashing algorithm, such as bcrypt or SHA-256. The resulting hash value, along with the salt, is stored in the server's database. The format of the stored hash can be represented as "salt#H(salt|password)".

6. **Password Security**: Storing the password in a hashed format with a unique salt mitigates the risk of password compromise. Even if an attacker gains access to the server's database, they would still need to perform a costly and time-consuming process to crack the hashed passwords, reducing the risk of unauthorized access to user accounts.

By aligning with the requirements of the project, this approach ensures secure user registration and authentication:

- **Secure Password Storage**: The use of salted hashes for password storage protects the confidentiality of user passwords, making it difficult for attackers to retrieve the original password from the stored values.

This aligns with the requirement of secure password storage (security requirement number 7).

By employing secure password storage techniques and following recommended cryptographic practices, the messenger app can securely handle user registration and authentication, safeguarding user accounts and aligning with the project's requirements.

Send Message

To send a message in the messenger app, the following steps are taken:

1. **Get Public Key**: The client retrieves the public key of the intended recipient. This can be obtained from the server or from the recipient's user profile. The public key is necessary to encrypt the message securely and ensure that only the recipient can decrypt and read it.
2. **Message Encryption**: Using the obtained public key, the client encrypts the message content. This encryption process ensures that the message is transformed into an unreadable format that can only be decrypted by the recipient's corresponding private key. By encrypting the message with the recipient's public key, the client guarantees the confidentiality of the message during transmission.

This approach aligns with the requirements of the project as follows:

1. **End-to-End Encryption**: By encrypting the message with the recipient's public key, the design ensures end-to-end encryption. The message can only be decrypted by the recipient's private key, guaranteeing that only the intended recipient can access the message. This aligns with the requirement of providing end-to-end encryption to maintain message confidentiality (security requirement number 1).

2. **Authentication of Recipient**: By obtaining the recipient's public key, the client can verify the authenticity of the recipient. The public key acts as a unique identifier and ensures that the message is encrypted specifically for the intended recipient. This aligns with the requirement of authenticating the recipient to prevent unauthorized access to the message (security requirement number 7).

3. **Prevention of Man-in-the-Middle Attacks**: By encrypting the message with the recipient's public key, the design mitigates the risk of man-in-the-middle attacks. The encryption ensures that even if an attacker intercepts the message during transmission, they will not be able to decrypt it without the recipient's private key. This aligns with the requirement of preventing man-in-the-middle attacks to maintain the security of the communication channel (security requirement number 10).

4. **Safe Encryption Algorithms**: The design should employ secure encryption algorithms to ensure the confidentiality of the message. Commonly used algorithms such as RSA or AES can be employed for encryption, which aligns with the requirement of using safe encryption algorithms to protect the message content (security requirement number 11).

Overall, the process of obtaining the recipient's public key and encrypting the message aligns with the project requirements by providing end-to-end encryption, authenticating the recipient, preventing man-in-the-middle attacks, and using secure encryption algorithms. This ensures the secure transmission of messages in the messenger

app, maintaining the confidentiality of the communication and meeting the necessary security requirements.

Receive Message

To receive a message in the messenger app, the client follows the following steps:

1. **Polling the Server**: The client periodically polls the server to check for any new messages. This involves sending a request to the server to retrieve any pending messages that are addressed to the client.
2. **Message Retrieval**: Upon receiving a response from the server, the client retrieves the encrypted message. The message is typically encrypted with the client's public key, ensuring that only the client with the corresponding private key can decrypt and read the message.
3. **Message Decryption**: Using the client's own private key, the client decrypts the received message. Decryption transforms the encrypted message back into its original readable format, allowing the client to access and view the message content.

This approach aligns with the requirements of the project as follows:

1. **End-to-End Encryption**: By decrypting the message with the client's private key, the design ensures end-to-end encryption. The message is encrypted with the client's public key by the sender, and only the client possessing the corresponding private key can decrypt and read the message. This aligns with the requirement of providing end-to-end encryption to maintain message confidentiality (security requirement number 1).
2. **Authentication of Client**: The use of the client's private key for message decryption also ensures authentication. The private key acts as a unique identifier, allowing the client to prove its identity and

authenticate itself as the intended recipient of the message. This aligns with the requirement of authenticating the client to ensure that only authorized recipients can access the messages (security requirement number 7).

Overall, the process of polling the server, retrieving the encrypted message, and decrypting it with the client's private key aligns with the project requirements by providing end-to-end encryption, authenticating the client, preventing unauthorized access, and ensuring secure key management. This ensures the secure reception and access of messages in the messenger app, maintaining the confidentiality and integrity of the communication while meeting the necessary security requirements.

Groups

To create a group in the messenger app, the following steps are taken:

1. **Admin Requests Group Creation**: The admin user initiates a request to the server to create a new group. This request typically includes the necessary information such as the group name and initial members.
2. **Server Group Creation**: Upon receiving the group creation request, the server creates a new group and associates it with a unique group ID. The server also stores the admin user as the group administrator and includes the initial members in the group's member list
3. **Secret Key Generation and Sharing**: The server generates a secret key for the group and securely shares it with the admin. This secret key is used for encrypting and decrypting the group messages, ensuring that only authorized group members can access the content
4. **Adding Members**: When a new member wants to join the group, the admin takes the responsibility to add them. The admin sends the secret

key to the new member securely, either through a secure channel or by encrypting the key with the new member's public key

5. **Group Messaging**: When sending a message within the group, each member encrypts the message with the group's secret key and sends it to every group member. This ensures that all group members receive the message and can decrypt it using the shared secret key

6. **Key Refreshment**: Upon request or when adding new members, the admin can initiate the sharing of a new secret key. This process involves generating a new key, securely sharing it with the group members, and updating the group's secret key in the server's storage.

This approach aligns with the requirements of the project as follows

1. **End-to-End Encryption**: The use of a secret key for encrypting and decrypting group messages ensures end-to-end encryption within the group. Only authorized group members with access to the shared secret key can decrypt and read the messages. This aligns with the requirement of providing end-to-end encryption to maintain message confidentiality (security requirement number 1)

2. **Authentication of Senders**: By encrypting the group messages with the shared secret key, the design ensures the authentication of senders within the group. Only members who possess the correct secret key can encrypt messages, preventing unauthorized users from sending messages on behalf of others. This aligns with the requirement of authenticating senders to maintain message integrity and prevent unauthorized message forging (security requirement number 2)

3. **Secrecy among Group Members**: The sharing of the secret key only with authorized group members ensures that the group's communication remains confidential and restricted to its members. This

aligns with the requirement of maintaining secrecy among group members to prevent unauthorized access to the group's messages (security requirement number 4).

4. **Key Freshness**: The ability to initiate a key refreshment process, either upon request or when adding new members, allows for the periodic renewal of the secret key. This enhances the security of the group's communication by preventing attackers who may have gained access to previous session keys from decrypting past messages. It aligns with the requirement of key freshness to ensure the confidentiality and security of the group's messages (security requirement number 10)

5. **Group Administration**: The design assigns an admin role to manage group creation, member addition, and key sharing. By granting this responsibility to the admin, the design ensures proper control and governance over the group's activities. It aligns with the requirement of group administration and management to maintain group integrity and prevent unauthorized access (security requirement number 12)

Overall, the process of creating a group, sharing a secret key among members, and encrypting messages using the shared key aligns with the project requirements by providing end-to-end encryption, authentication of senders, secrecy among group members, key freshness, and group administration. This ensures secure group communication, maintaining the confidentiality, integrity, and security of the messages exchanged within the group while meeting the necessary security requirements.

Key Management

In the messenger app, key management is an essential aspect of ensuring the security and confidentiality of sensitive information, such as encryption keys. The following approach is employed for key management:

1. **Password-Based Key Encryption**: To securely store the encryption keys, a password-based encryption scheme is implemented. The key is encrypted using the Fernet algorithm, a symmetric encryption algorithm that provides secure encryption and decryption of data. The Fernet algorithm requires a secret key, which is derived from the user's password using a key derivation function, such as PBKDF2 or bcrypt.

2. **Secure Key Storage**: The encrypted key file is stored securely, ensuring that only the authorized user with knowledge of the password can access the encrypted key. This can be achieved by employing secure storage mechanisms, such as encrypted databases or secure file systems, that provide protection against unauthorized access.
3. **Key Retrieval and Decryption**: When the user needs to use the key, they provide their password, which is used to decrypt the encrypted key file. The decrypted key is then loaded into memory and used for encryption and decryption operations within the messenger app.

This approach aligns with the requirements of the project as follows:

1. **Secure Key Storage**: By encrypting the key file using the Fernet algorithm and protecting it with a password, the design ensures that the key is securely stored and inaccessible to unauthorized users. This aligns with the requirement of secure key storage to prevent

unauthorized access to the encryption keys (security requirement number 8).

2. **Confidentiality of Keys**: The encryption of the key file with the user's password ensures that only the authorized user with knowledge of the password can decrypt and access the keys. This preserves the confidentiality of the keys and prevents unauthorized parties, including the server, from accessing and using the encryption keys. It aligns with the requirement of maintaining the confidentiality of the encryption keys (security requirement number 1).

Overall, the approach of password-based key encryption and secure key storage ensures the confidentiality, integrity, and accessibility of the encryption keys while aligning with the project requirements of secure key management, confidentiality of keys, protection against unauthorized access, and key freshness. By employing strong encryption algorithms and promoting secure password practices, the messenger app can effectively manage and safeguard the encryption keys, maintaining the security and privacy of the communication.

Execution

To execute the code for the server and client components of the messenger app, the following steps can be followed:

Server

1. Install the required dependencies by running the command `pip install -r requirements.txt`. This ensures that all the necessary libraries and packages are installed to run the server code.

2. Run the server setup script by executing the command `python server/server_setup.py`. This script performs any necessary configurations or initializations required for the server.

3. Start the server by running the command `uvicorn server.server:app --host 0.0.0.0 --port 8022`. This command launches the server application using the Uvicorn web server, specifying the host as `0.0.0.0` and the port as `8022`.

By following these steps, the server component of the messenger app will be up and running, ready to handle client connections and facilitate secure communication.

Client

1. Run the client code by executing the command `python client.py {your_username} {your_friend_username}`. Replace `{your_username}` with your chosen username and `{your_friend_username}` with the username of the friend you want to communicate with.

2. Once the client code is executed, you will be prompted to enter the username of the friend you want to send a message to.

3. Enter the message content when prompted.

4. The client will send the message to the server, which will handle the necessary encryption, signing, and delivery to the intended recipient.

Example:

Here's an example usage scenario:

1. Execute the command `python client.py client#2 client#3` in the terminal to run the client code. This initializes a client with the username "client#2" and connects it to the server.

2. When prompted to enter the recipient's username, enter "client#3" to indicate that you want to send a message to client#3.

3. Enter the message content "salam mashti" when prompted.
4. Now, switch to another terminal or window and execute the command `python client.py client#3 client#2` to run another instance of the client code. This initializes a client with the username "client#3" and connects it to the server.
5. Again, enter the recipient's username as "client#2" when prompted.
6. Enter the message content "aleyk" when prompted.
7. Upon sending the message, the client#2 instance will display a notification that client#2 has sent a message to you: "salam mashti".

This example demonstrates the exchange of messages between two clients, client#2 and client#3, via the server. The messages are encrypted, signed, and securely delivered, ensuring the confidentiality, integrity, and authenticity of the communication.