Technician's Guide

Welcome! This guide will demonstrate how to use our messaging application, from the perspective of a technician or developer. This document will provide an inside look and explanation on how our program functions.

Packages / Code Attributes

- Server code for communicating between sockets is based on Server Sample written by Dr. Adam J. Lee (adamlee@cs.pitt.edu)
- com.google.gson - Library for Json Serialising created by Google

To run

To run our program, you must have Maven installed as our program relies on certain Maven libraries to function. And a JDK runtime of version 9 or higher. From there, there are three components that need to be run. If you navigate to src/main/java/com/ciphersquad/chat, you will find three files:

- App.java, which represents the Client.
- ResServer.java, which represents the Resource Server.
- AuthServer.java, which represents the Authentication Server.

To run the program, each one of these programs must be run. Whether they are on the same device or different servers is irrelevant, as long as you know the servers' IPs.

To run the program, each one of these programs must be run. Whether they are on the same device or different servers is irrelevant, as long as you know the servers' IPs. The commands to run each component are located in their respective bash files, which can be run as follows:

First, run mvn compile. Then:

For the client, run `bash main.bash` For the AS, run `bash AS.bash` For the RS, run `bash RS.bash`

Demo Notice

Of note, this application is only a demo of a proposed future product. All of the functionality present in this application (including everything listed in this guide and the user's manual) would be present in this hypothetical application, along with additional functionality not present in this demo. This would include, but is not limited to, deleting user messages and blocking users, which is not currently present in this application (but would exist in the actual implementation). Furthermore, all of our cryptographic protocols and operations, designed to ensure client and server security, would exist and properly work in the final product (including necessary cryptographic undertakings for the currently unimplemented functionality).

How does it work?

As mentioned above, there are three primary components of the system: The client, the Authentication Server (AS), and the Resource Server(s) (RS).

The Client

The client is the user-side interface for the application, and the starting point for using the application. Upon running, the client prompts the user for a multitude of options (for more information, please see the user's manual).

The Authentication Server

When the AS first runs, it loads a list of all group_ids and user permissions from a file. Furthermore, the AS loads all file keys (including every version) for every group. These are stores in a HashMap as group_id -> (encoded key, version #) pairs.

There are a multitude of reasons that the client could connect to the AS. If they want to create a group chat, update group permissions, login, register, or delete an account, the user must log in to the AS. Thus, the AS is always ready to accept messages from clients for various reasons.

The AS uses Diffe-Hellman for exchanging keys between the client and the AS, as well as AES with CBC block mode, 256-bit key and PKCS7Padding for sending and reciving communications from client/AS. For verifying the token is unmodified on the RS side, the AS signs the token using RSA with 4096 key size, PSS padding and SHA-256 Hash algorithm. (See Phase-3 Writeup for more specific details on cryptographic functions).

The protocol section below covers how the AS and Client interact more in depth.

Communication between the Authentication Server and the Client generally takes place using the AuthRequest (`com.ciphersquad.chat.AuthenticationServer.AuthRequest`) and AuthResponse (`com.ciphersquad.chat.AuthenticationServer.AuthResponse`) classes. The AuthRequest consists of a username (String), password (String), AuthType (an enum representing one of the of the options the client can take, found in the AuthRequest class), and an optional members list (String ArrayList) and/or a groupID (String), as well as an optional field for the clients public key. The AuthResponse a success boolean, a String message for sending error messages to the client, a String Token for sending tokens, and a byte array for the signature of the token for authentication on the Resource Server, as well as the public key for the server.

When the AuthServer receives an AuthRequest from the client, it creates a new AuthThread (`com.ciphersquad.chat.AuthenticationServer.AuthThread`) to handle the request. Once received, depending on the request, the AuthThread handles the request and returns an AuthResponse object to the Client, which contains information on whether the request succeded or failed (and why). Below lies more information on each of the specific functions.

On startup, the user is given the option to update the IP/Port of the Authentication Server, or continue and connect to AS after IP/Port are set. If the user fails to set the IP/Port it will not allow them to connect to the AS. Localhost is the default port.

For logging in , once the client connects to the AS, the client sends an AuthRequest object, containing a username (String), password (String), and AuthType (LOGIN) to the AS. The AS then compares this username and password combination to the hash map of stored usernames and passwords. If the username is found, the AS will create the user's token containing the user's name and group permissions. Then, the client has the 8 options on what to do after logging in (see user guide on the options). If the user does not exists or the password is incorrect they will be returned to the login screen where they can create a new account.

For creating a groupchat, user 'u' is prompted for usernames of members they want to add to the groupchat via the client's AuthServerScreen. The user will be prompted first for the groupchat number and then prompted repeatedly for usernames until they type done, once "done" is typed, the client connects to the AS to validate permissions of user 'u' and then creates the groupchat and adds the member(s) (checking first if the member exists) all with permission 'Non-admin'(N). The creator of a groupchat is always going to be automatically added, with 'Admin'(A) permissions. If no users are added, no groupchat will be created and an error message will be thrown. The user will then be returned to the main menu. Metadata is updated after every groupchat creation.

For adding or removing a user from a groupchat, client prompts the user for the group number, then prompts for a user to add/remove from the group. The user is then connected to the AS, where they are authenticated using the username and password provided in the AuthRequest. Once authenticated it checks that the group exists, then checks that the user who is requesting the add/remove is the admin for that group. If the group does not exist, the user isn't an admin, or the user isn't authenticated, then the add/remove fails and the user is returned to the main menu. Otherwise the user is added/removed with 'Non-admin'(N) permissions and the user is returned to the main menu. Metadata is updated after every add/remove.

Registering an account is pretty similar to logging in: they are prompted for a username and a password, and as long as the username is not currently in use, everything will be successful. The AS will add this (username, password) pair to the HashMap storing all usernames and passwords, which will eventually be written to file for storage upon AS shutdown.

For deleting an account, the user is prompted for their username and password; if correct, the account along with all their group permissions will be deleted from metadata.

When the client issues a remove group request, after logging into the AS, the AS asks what group they want to delete, and the client sends the group_id to the AS. Next, the AS edits user permissions of ALL users in the group, replacing A/N (for Admin or Non-admin permissions) with "D". Next time a user authenticates with a permission containing a "D", that permission is subsequently removed from their permissions AFTER they receive a token - i.e., they receive a token with the "D", then AS deletes that permission from that user. When a user with a "D" in the token connects to the RS, the RS shuts the socket first, then deletes the group history file.

When the client issues a list group request, if the user is authenticated using the username and password provided, the AS produces a list of every group they are a member of, and stores it in an AuthResponse token. The AS sends this AuthResponse, which the client parses and prints out to the user.

Currently, the token is an object with an associated username and list of permissions and timestamp. Each permission is a String, formatted like "####(A/N)", where the 4 hash tag's represent a group_id, the A represents administrator permissions, and the N represents normal permissions. To dole out these permissions, the AS also keeps track of every group created, and what users have access each group. The AS accomplishes this by keeping track of all groups a user is able to access by storing a HashMap of (userid, ArrayList of groups they belong to) pairs. The timestamp field is updated at creation of the token with the current time. This is used as part of the verification of the token on the RS side. The Token also contains a hash of the public key for which ever resource server the user plans on joing, this is for the purpose of preventing the Resource server from stealing a token and using it on another resource server.

The Resource Server

The RS is divided into three parts, the ResServer application, the RSThread, and the GroupThread.

ResServer

The server application itself only opens a connections at a given port, and actively listens to connection to the port.

- The application will prompt for the IP that the server is hosted on, or the IP that external users connect to.
    - The IP will be used to retrieve the public key of the server
- The application will prompt a port number on startup, with the option for automatic assignments if choose so.
- There is no restriction for opening the port, only that the port number needs to be within the allowed bounds, and is not in use by other threads or applications.

Once a client connects to the server port, a Resource Server Thread (RSThread) is created to handle server-client interactions.

The RSThread is responsible for all client-server interactions on the backend that does not include the messaging.

The server will shutdown upon manual request (Ctrl + C), and will close all associated threads after saving necessary data.

RSThread: Mutual Authentication

NOTE: For diagrams, intuition behind design, etc. Please refer to the phase 3 writeup.

Upon receiving user connection, the RSThread will initiate the mutual authentication protocol to exchange session keys and verify the token from the user. The specific steps are:

- Client generates the a random nonce, encrypt using the RS' public key, then send the encrypted nonce to the RS
- RS receives the nonce, decrypt the nonce, generate a second nonce, sign the new nonce using RSA signing, then send the signed new nonce and its signature to the client
- The two nonces are XORed together to be used as a shared secret to genereate the AES session key
- After the key is generated, all messages from this point on will be encrypted through the session key
- RS will send the encrypted message "session key established" to the client
- Client would follow the same steps to generate the session key, decrypt the confirmation message, and send their encrypted token to the RS
- RS receives the token from the token, verifies the token using the AS' public key and ensures the timestamp is valid (within 5 minutes), and that the public key hash on the server matches the key hash of the RS' public key
    - The encoded key will be hashed using SHA-256

Throughout the protocol, if any part fails (i.e. verification fails, invalid nonce sent by the user). The RS/Client will close all sockets and abort the connection.

RSThread: Message Encryption

For both public key and symmetric key encryptions, the implmentation will be through the BouncyCastle package.

RSA keys will be 4096 bits, with encryption using OAEP and MGF1Padding with SHA-256 as the hashing algorithm, and signing using PSS padding and SHA-256.

AES will have 256 bit keys, with CFB block mode and no padding.

The RS will also be using a SHA-256 HMAC, also with 256 bit keys.

The encrypted messages sent between the server will be byte arrays sent as serialised RSMessage Objects. The objects will contain the encrypted GroupMessage Objects(which contains the actual message), the IV for the encryption, and the HMAC of the message.

- The encrypted group message object is serialised group message object encrypted with AES
- The encrypted message, IV, and HMAC will all be byte arrays

```
{
  "groupMessage" : [1],
  "IV": [2],
  "HMAC": [3],
  "Sender": "user1"
}
```

RSThread: Group Connection

After authenticating, the RSThread will read in names all existing group message histories, stored at predetermined location (default: src/main/resource).

- The message histories are stored as a json file, with a format like the following:

```
{
  "groupID": 1,
  "messageHistory": [
    {
      "theMessage": [-111, 60],
      "IV": [
        0, 16, -114, -29, -3, -51, -38, 72, -52, -26, -122, -42, -86, -24, -20,
        -122
      ],
      "HMAC": [
        106, 55, -11, -123, -44, -19, 92, 59, -34, -81, -68, -84, 25, 46, 9,
        104, 126, 92, -59, 33, 11, -31, 64, -105, -121, -1, 96, -51, -88, 124,
        -27, 82
      ],
      "sender": "user",
      "counter": 1,
      "keyVersion": 1,
      "endToEndEncrypted": true
    },
```

```
    {
      "theMessage": [-74, 126, -45, -109, 35, -92],
      "IV": [
        -40, -69, 21, -43, 121, 64, -47, 56, -4, 24, -118, -70, -83, 18, 59,
        -118
      ],
      "HMAC": [
        87, -25, 56, 40, 46, -80, -76, 47, 68, -45, 117, 6, -123, 7, -23, -105,
        -42, 27, -114, -2, -66, 94, -38, 109, 119, -115, -63, 95, 17, 45, -107,
        -56
      ],
      "sender": "user2",
      "counter": 1,
      "keyVersion": 1,
      "endToEndEncrypted": true
    }
  ]
}
```


- The format is generated from the serialisation of the Group class (`com.ciphersquad.chat.ResourceServer.Group`), consists of the following
  - The ID assigned to the group
  - A linked list of Messages (`com.ciphersquad.chat.ResourceServer.GroupMessage`) that contains the encrypted chat history, information on encryption status, sender, and the version of the file key used to encrypt it.
- The message itself is stored in JSON files with naming format as a four digit group number such as `0404.json`
  - If the groupID is not long enough, it is padded to four digits by adding leading 0s.
  - If the group is a private message group, then there will be a preceeding `P` at the front of the name
    - e.g. `P1145.json`

The RSThread only parses the group names of the file to determine what groups has existed already, for each group that exists, it will create a ServerSocket for the group in case of future connection.

- The port numbers are automatically allocated by the Java ServerSocket class
- The lists are stored inside Hashtables, with the groupID as the key, and the assigned serversocket.
- The list of group chats and PM chats will be stored in serperate Hashtables.
- The list of groups are shared between all RSThreads, so if a group has been assigned a socket, then future RSThread will not be modifying that socket for the current session.

The RSThread will communicate with the client through an Object I/O Stream. And will encode messages as byte matricies (refer back to message encryption for details), and send through these object streams.

- For the format of the token, refer to the Authentication Server section above, and the Token Class files.
  - `com.ciphersquad.chat.AuthenticationServer.Token`

After the group data from the DB is processed, the token will parse the token to verify user access

- The token sent will look like the following

```
tokenMessage[0] -> Encrypted Token
tokenMessage[1] -> IV for encryption
tokenMessage[2] -> Signature for Token
```

- The token is verified using the AS' public key
- If the timestamp within the token is over 5 minutes old, the server will abort connection after sending the user a message on expired token.
- Using `com.ciphersquad.chat.AuthenticationServer.TokenSerializer` for deserialising.
- If there are groups not included in the message history storage, it will be noted by the RS as a newly created group, thus it will create allocate a new socket for it if it had not been created already.
- Again, for the encoded token, refer to the authentication server section, the RS only parses it as intended.

After the token is parsed, the RS sends back a timestamp indicating when the token parsing is finished for the client to verify.

- The timestamp will be expected to be within 6 minutes of token issuance.

Then, the client collects the list of groups that the user has permission to join from the RS and communicates the list as a string to the user's input stream.

- The user can respond with the group number that they want to join, or to select the Private Messaging option, or choose to close the connection
    - The user's response will also be a encrypted string
    - If the user choose the disconnect, closes connection to the socket, and ends the thread.

Private Messaging

If private messaging is chosen, indicated by the client sending `"PM"` to the server, then the RSThread will display the list of possible user to establish a private messaging session with the user.

- If any active user blocked the current user, then they will not appear as an option for the client to choose.
- It will simply be an invalid option even if the client tried to connect with a user that blocked them.
- If the user chooses an invalid option, the RSThread will send a String to the client indicated that an invalid choice was made, and for the client to send a valid response back.
- An `InvalidSelectionException` is thrown on the server thread to indicate that this has happened, this does not affect the client. It is only for logging purposes.
- Users can only try to establish PM sessions with users that are already online
    - Otherwise the user will likely be busy waiting
- The user can send "~back" to return to group selection

Upon a valid choice being received, the RSThread will assign the PM session an ID if it has not been created yet, then assign the session with a server socket with the port number picked by the Java ServerSocket class. The port number is then written as an int number and sent to the client.

- The PM group and its members will also be added to a HashTable

After all this, the RSThread will create a GroupThread to handle messaging operations, then it will enter active waiting until the GroupThread is terminated, which will then teardown the thread, closing the socket for the user.

Group Messaging

The protocol for group number selection is the same here as private messaging.

- All groups that the user does not have access to will not appear as an option for the client to choose.
- It will simply be an invalid option even if the client tried to connect with a group with out permission.
- If the user chooses an invalid option, the RSThread will send a String to the client indicating that an invalid choice was made, and for the client to send a valid response back. The client, upon receiving a String instead of an int, will loop back to the prior step (selecting a group or choosing to PM)
- An `InvalidSelectionException` is thrown on the server thread to indicate that this has happened, this does not affect the client. It is only for logging purposes.

Upon receiving a valid choice from the client (sent as an int), the RSThread will query for the port number stored in the HashTable, then the port number is written as an integer and sent to the client. The client then automatically connects to the RS at the received port number.

After these events, the RSThread will create a GroupThread to handle messaging operations; then, it will enter active waiting until the GroupThread is terminated, which will then teardown the thread, closing the socket for the user.

End to End Encryption

To ensure privacy of the clients, all messages are end to end encrypted.

All messages stored on the resource server will be encrypted by AES, with corresponding HMAC to ensure no messages are tampered. The encryption is done by having the user encrypt the message within of the Group Message object, which contains the plaintext string of what is sent in the grop chat, through a file key distributed by the authentication server. This key is shared between all group members but not the resource server. As the messages are encrypted, the RS simply transmits it between users without the ability to view the contents.

Re-Encryption Requests

Upon an user log in, the user may request re-encryption of the chat history.

The RS will receive the request, replace the chat history with the re-encrypted messages sent by the user, and then all users connected to the group that is being re-encrypted will have their connection closed to have them re-authentication and receive the new file keys from the authentication server.

Group Thread

Of note, private messaging uses the same protocols as group messaging: it is handled as a 2-person group.

During the creation of the group thread, the group thread will receive all necessary information about the client from the RSThread

- i.e. Username, session key, information about the group chat connected to(group number, is pm session, etc.)

Upon creation of the group thread, the thread will attempt to read the message history file for the group.

- If there is group history, then the history will be deserialised into a Group class object
- If there is no previous message history for the group, then a new group object will be created for the group.

The user will be stored into the list of connected users by the GroupThread, then Object I/O stream will be opened to the user. The user's session key will also be stored in a list accessible to all group threads

- The group thread requires the session key of all users to broadcast and decrypt messages.

All group messages sent to the user will be the same format as the RSThread

- byte matricies that contains an encrypted serialised group message object, and the IV for decryption

The server will receive messages that includes one extra field being the sender of the message

- Done so that the user can fetch the correct session key from the shared list to decrypt the message

The first message sent to the user will be the number of messages that is stored in the history, then all messages will be send to the client as encrypted `GroupMessage` object.

Then the GroupThread enters a loop that receives user messages, then broadcast it to all other users connected to the group.

- The messages are also added into the group object's message history list.

If the user enters in the input '~back', the GroupThread will write the group object to persistent storage, close the socket, and remove the user from the list of connected users.

- For format and location, see the start of the RSThread section.

After that the thread will return, after which the RSThread will prompt the user to connect to a new group, if the user choose not to, then the RSThread will also return.

- The user is connected to a the same RSThread for server interactions, so they would not need to be authenticated again.

Server Shutdown

During normal shutdown, the server should have no active threads associtated with it, so it just shuts down after the server side receives a shutdown signal (`Ctrl + C`)

In the case where there are active threads, the server will signal all RSThread to interrupt its associated GroupThreads, which will write the history to persistent storage before both closing the respected sockets and returning. When all thread is terminated, the server application will shutdown.

- If a RSThread does not have an associated GroupThread, it just close its socket and return on its own.
- Crashing by forcefully killing the process is not handled, it will just crash all associated thread and possibly break stuff