Multi-party chat application documentation

Dömötör Szilárd *AE9EC3*
Juhász Márton *GASYQY*
Székely Gábor *EDVTAZ*

Changes to original specification

In response to the feedback provided to the specification, we made the following changes: we trust the server to apply timestamps to all messages, and additionally establish a secure channel between each client and the server. If the server isn't compromised, the timestamps will be protected by AES-GCM, and thus every action displayed to the client will be marked with its time of the server originally receiving it. The messages sent will not be possible to be reordered by an attacker. The server broadcasts every message it receives to each participant, including the sender, essentially serving as a confirmation, that the message was received and accepted, thus if there was a sequence number collision and the message was dropped, it can be resent. If the client receives a message without a timestamp, it was not sent by the server, since the server applies the timestamps and can be dropped. Also, to allow for edgecases, where two users send a message at the same moment, we added a userwide counter.

If the server is compromised, it can't decrypt messages in channels, however by colliding with a channel member, it is possible to achieve the order of messages to appear different for different users. This is possible, since the server controls the timestamps, and the malicious user can leave a gap in sequence numbers, that she can later fill with messages.

Data structures

The following json documents are examples for the data structures used for data storage and messaging.

Example message format

```
[
  {
    "type": "addUser",
    "inviter": "userIDa",
    "invitee": "userIDb",
    "channelID": "channelID",
    "channelkey": "AAAA====",
    "signature": "BBBB====",
    "timestamp": 12341234
```

```
  },
  {
    "type": "comms",
    "channelID": "channelID",
    "channelSeq": 123213,
    "userID": "userIDa",
    "userSeq": 123123,
    "msg": {
      "nonce": "AAA===",
      "tag": "AAA====",
      "ct": "AAA===="
    },
    "signature": "BBBB====",
    "timestamp": 12341234
  },
  {
    "type": "initConn",
    "userID": "userIDa",
    "nonce": 123123
  },
  {
    "type": "initKey",
    "key": "AAA===",
    "signature": "BBBB===="
  },
  {
    "type": "replayFinished"
  },
  {
    "type": "error",
    "error": "errorMsg"
  }
]
```

Example storage format

```
{
  "certs": {
    "userid1": {
      "signing": {
        "public": "AAAA===",
        "private": "BBBB==="
      },
      "encryption": {
        "public": "AAAA===",
        "private": "BBBB==="
      }
    },
```

```
    "userid2": {}
  },
  "channels": {
    "channel1": {
      "channelkey": "AAAA====",
      "seqNum": 123,
      "invites": {
        "userid2": {
          "inviter": "userid1",
          "payload": "AAAA====",
          "timestamp": 12341234,
          "seqNum": 234
        },
        "userid3": {}
      },
      "messages": [
        {
          "timestamp": 12341234,
          "sender": "userid1",
          "text": "AAAA===",
          "payload": "AAAA==="
        },
        {
          "timestamp": 12341234,
          "sender": "userid2",
          "text": "AAAA===",
          "payload": "AAAA==="
        }
      ]
    },
    "channel2": {}
  }
}
```

`chat.py`

The `chat.py` contains the entry point for the application. There are three different modes. One for starting the server, one for the client and one for generationg necessary files for normal operation. If the generate mode is chosen, the users and their respective passwords need to be specified. We used the `argparse` module.

client.py

The client.py file contains code controlling the main execution flow of the client. The run function needs to be called to initiate a client.

- The run function loads the storage, connects to the server, creates a new Session object, that is used to start the main menu on a new thread and listen for incoming messages on the original thread.
- The listen function handles incomming messages from the server, and calls the respective handler for the message on the session object.
- The main_menu function lists known channels, handles the execution of creating new channels and joining existing channels.
- The chat function handles input from the user in a channel. Possible actions are sending messages, inviting users and exiting the application.

server.py

ThreadedTCPRequestHandler class is responsible for handling client requests. A new instance of this class is created automatically every time a new client connects. The new instance listens for incoming messages and handles them in accordance to their type. The secure connection is initiated with initConn and initKey messages and if that is successful all subsequent messages are checked as specified and the ones that pass and need to be forwarded are broadcasted to the correct users.

- The run function loads the storage and the server certificates. Starts a TCP server, that automatically spawns a new thread for every incoming connection.
- The setup function initilizes the current user's Session object, and stores it in the session storage (containing the current connections).
- The send function sends the given message to the specified session, or the current session if unspecified. Secure channel is used if a key has been established.
- The send_invites function sends the invites contained in the given channel to the specified user, or the current user if unspecified.
- The send_messages function sends the messages contained in the given channel to the specified user, or the current user if unspecified.
- The send_replay_finished function signals the given user (or the current user if unspecified) that the replaying of messages and invites are finished.
- The persist function in the server.py file *actually* persists the storage object, storing all channels, messages and invites.

config.py

The config.py file contains some global parameters for the application, such as the length of the channel keys and the location for the server's private certificate. It also contains the string for the banner.

frontend.py

The frontend.py file contains code related to the frontend of the application. It's function are designed in a way, that if for example a GUI interface instead of the current CLI interface is used, only this file would need to be modified.

- CACHED_MSG is used to store the users input in a channel, so if a new message arrives and overwrites the in progress text input, it can be restored
- The get_user function Displays the login screen and requests the user's credentials then returns them to the caller

gen.py

The gen.py file contains the code responsible for generating the certificates and constructing the storage file, that is used by all instances of the application, to load the respective certificates for all users. The private keys are contained protected by a passphrase, the password of the user.

session.py

The Session class is responsible for maintining the storage object, mostly with regards to client side functionality (some of which is also needed on the server side). It implements functions for handling incoming messages from the server and updating the storage object in accordance. It also implements getter and setter functionalities for some of the most common accesses into the storage object.

- In the constructor, we load all keys into the storage object, decrypt the user's private keys and initiate all known RSA keys.
- The check_seqnum function checks whether the supplied sequence numbers are valid, and if so, it updates the afore mentioned sequence numbers accordingly.
- The add_user function handles the add_user message.
- The incomm function handles the comms type messages (incomming text message from one of the channels users)
- The persist function does nothing

StringKeys.py

The StringKeys.py file contains a class with static string constants, used in dictionary accesses, to lower the risk of typos.

utils.py

The utils.py file contains the implementation for getch, a function that gets a single character from stdin (and unlike sys.stdin.read(1) it doesn't wait for an endline to flush the buffer).

- The _Getch function Gets a single character from standard input. Does not echo to the screen.

messaging

The messaging module contains code related to handle low-level network communication and encryption.

common.py

The file contains methods used by the client as well as the server.

- The recv_all function receives exactly n bytes
- The recv_message function receives a single message and decrypts it if the key is given.
- The send_msg function sends a single message and encrypts it if the key is given.
- The decrypt_sym function decrypts AES-GCM symmetrical encryption with the given key. Used for server-client secure channel.
- The encrypt_sym function encrypts with AES-GCM symmetrical encryption with the given key. Used for server-client secure channel.
- The pack_msg function packs message, prefixing it with its length in 4 bytes big-endian.
- The verify_sig function verifies signature on a json payload with cert.
- The check_msg_sig function verifies signature on a message, automatically selecting the correct certificate based on the type of the message and its contents. If the extra parameter is given, it is also inserted into the copy of msg before signing (used for implicit nonce signing).
- The create_sig function creates a signature on a json payload with cert.
- The create_msg_sig function creates signature on a message, with the private certificate of the current user. If the extra parameter is given, it is also inserted into the copy of msg before signing (used for implicit nonce signing).
- The select_cert function selects the correct user to sign the message for.

- The pkc_encrypt and pkc_decrypt functions are public key encryption methods.

client.py

The file contains client methods for messaging.

- The init_connection function initializes secure channel, returns with the agreed key.
- The new_channel function sends message to create new channel.
- The invite_user function sends message to invite user.
- The send_msg function send text message to channel.
- The GCM_create_header function serializes GCM header data into bytesarray.
- The encrypt_comm and decrypt_comm functions handles symmetrical encryption used to protect channel messages with AES-GCM.