COMPE 560 Homework

Skye Russ

Introduction:

The provided module implements a Client and Server pair that communicate using encrypted UDP. The server allows for any client to subscribe to the “chat room” after authentication and send messages between any number of clients. Each of these messages is encrypted using AES and verified using HMAC. There is an initial key exchange between the client and the server where the client provides it’s public RSA key and the server responds with a shared AES encryption key and an HMAC secret.

Implementation:

Both the Client and the Server scripts contain classes that share that name. Both classes implement the provided CursesConsoleApp class. The CursesConsoleApp class handles input and output from the console using curses to provide a streamlined and visually clean environment. Because the python package Curses is somewhat finicky in threads, both the Client and the Server use the Curses.wrapper() function to run their main function. While this would be ideally handled inside the CursesConsoleApp class, it remains in client.py and server.py to help keep the console clean and the curses session intact.

The initialization of the client asks for a given username and then proceeds to make a secure connection with the server. This connection starts with the client sending it’s 2048-byte public RSA token to the server. The server then responds with a shared AES key that is encrypted using the client’s RSA token, and a HMAC shared secret that is encrypted using AES. After this point, the client is free to send messages to the server.

The standard messages that the client sends to the server are in JSON format. These messages contain the following fields:

- “Message Type” field which distinguishes a data message from an acknowledgement
- “Message Data” field which contains the string message that the client wishes to send
- “Sequence Number” which increments based on the length of the Message Data field
- “HMAC Bytes” field which includes the HMAC verification bytes converted into hexadecimal

This dictionary is then converted to a string and encoded using AES before getting sent to the server. The server responds with an acknowledgement that includes the sequence number provided. If this acknowledgement is not received within 0.3 seconds, the client retransmits the message with the same sequence number. This happens three times over the following second before the client declares the message a timeout and gives up sending the message. It then prepares itself for the next message.

On the server side, it receives the client RSA bytes, sends back the AES and HMAC data and then adds the client to its internal list of hosts. This list is then used whenever a message comes in to decode the message, and then propagate the message to any other hosts. To do this, the server first acknowledges the message from the client, and then re-encrypts the message data using each other client’s AES key and

HMAC secret before sending them to each client. In this way, the messages are always encrypted and each client can verify the server's identity using the updated HMAC bytes that change with the message contents. The server asynchronously waits for acknowledgements from each client and does the same 0.3 second retries for 1 second that the client does. If any of these messages fail, the message gets dropped for that client.

Discussion/Conclusion:

The UDP based encrypted client and server are most vulnerable at the initialization phase. The initialization consists of 1 message from the client and then 2 messages from the server that include both the RSA encrypted AES key and the AES encrypted HMAC secret. If these messages were decoded a bad actor could gain full unencrypted access to the system. This could be improved by either or both of these byte streams being passed using a more secure method than UDP. Those methods could include physical, email, or some other secure file transfer protocol.

Furthermore, there is no authentication required to get access to the server. Any client can connect if they have the server address and port and implement the correct initialization phase. This could easily be fixed by having a password mechanism, certificate, or something similar to only allow registered clients into the chat room.

All that said, once the connection is established, there is very little chance that a listener could break the encryption of the clients or server. The messaging is quick and mostly reliable. The UI leaves some to be desired but it gets the job done. Longer messages have to be broken up into smaller chunks because the input box is locked at 64 characters. This is a design feature that requires users to send more frequent smaller messages rather than multiple paragraphs.

Further Study:

Further steps could include implementing a more pleasant console text experience or designing a GUI. These would improve the user experience and potentially provide extra functionality and clarity. Also, a password system could be implemented to add a further layer of security. Multiple chat rooms could be implemented as well. These rooms could have select sets of clients but is not as necessary with a small number of clients. There is no message history available to any new clients which could be useful as well. However, message history could potentially open vulnerabilities in the system and would potentially introduce extra burstiness into the environment because new clients would need more data than is currently required.