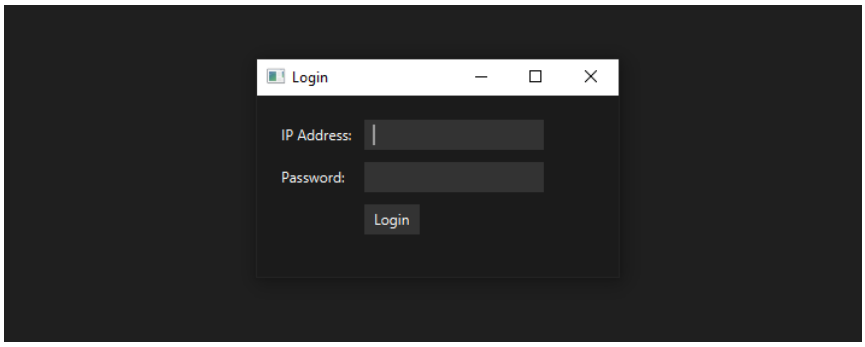
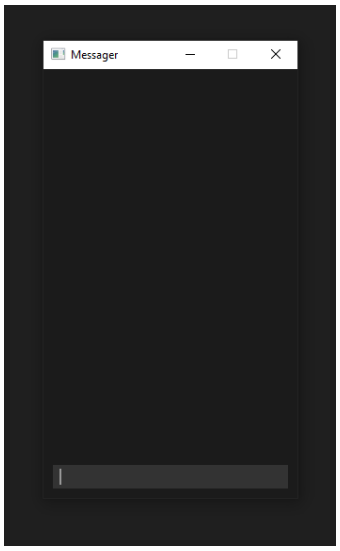
Blaine Jones
OUID#113237959
CS-4173 Computer Security
Dr. Shangqing Zhao

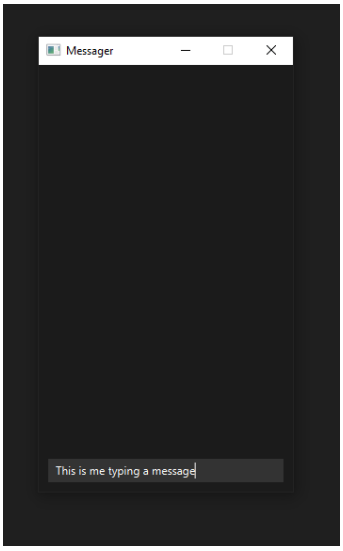
Project Report: Secure Instant Point-to-Point Messaging

For the project, I developed an app called Messager, written in Java, which allows for P2P text communication between two users. The application begins with a Login screen, which asks for the IP address of the person the user wants to communicate with, and the secret password that must be shared between the two of them.



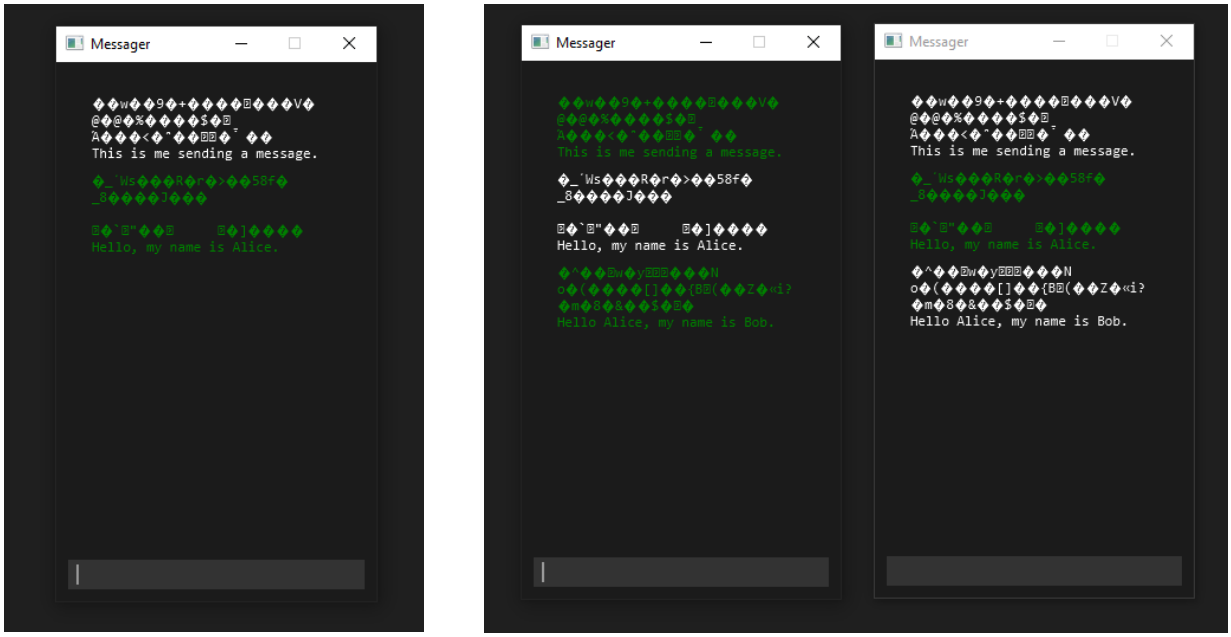
Whoever logs in first will attempt to connect to the given IP address as a client. If this fails, the application will automatically switch to act as host, and whoever logs in second will act as a client connecting to the host. In the application's current state, the host's GUI will not appear until a client connects to the host, so if no client connects, the user is basically staring at nothing. This is an area for improvement for the application – for example a waiting screen could be shown, or the GUI could be displayed with a connection status. For time's sake this has not been implemented for the submitted project. When the GUI launches, this is what the user sees:



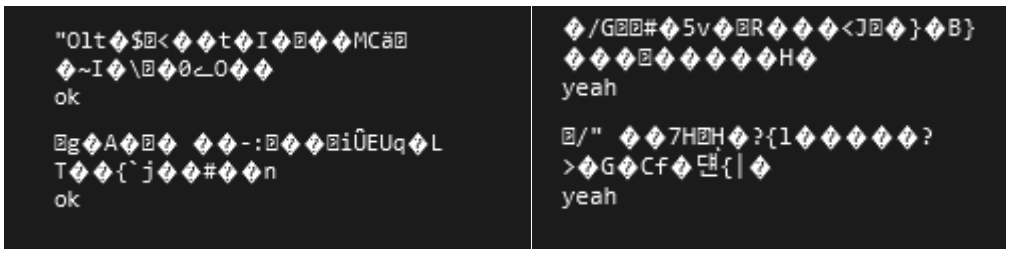


It’s simple, with a text box for typing messages, and a scroll pane for displaying them. Once the enter key is pressed, the message sends, and the ciphertext is displayed in the scrollpane along with the plaintext. The ciphertext in UTF-8 mostly appears as random symbols, many of which display as the question mark symbol since whatever UTF-8 code applies to those bytes isn’t a displayable symbol.

Receiving messages is handled on its own thread, so that the application can check for received messages continuously. When a message is received, both the ciphertext and plaintext are displayed in green:



Additionally, sending the same message twice results in different ciphertexts:



Here’s how it all works. This application consists of six classes: Login, Messager, Message, Hoster, Clienter, and ByteWrapper. The GUI uses JavaFX in Login and Messager. Login contains the main method, and on launch it builds the login window, and passes the login info to Messager. Messager builds the main messaging window and handles the overarching logic of handling connection and messages. Messager works with Message objects from the Message class. The actual encryption/decryption is done inside Message, which stores both the plaintext and ciphertext. Hoster and Clienter are the classes containing sockets and are used for sending and receiving messages. Messager decides whether the user acts as host or client, and then uses Hoster or Clienter appropriately. In

retrospect, these classes should have been merged into one class, with a boolean for client or host in the constructor, because handling the client or host question in Messager instead resulted in multiple segments of essentially duplicate code. ByteWrapper turns byte arrays (byte[]) into objects for transmission.

Below is the code for establishing the connection between users:

```
/*
 * Determines who acts as host and who acts as client.
 * User will attempt to connect to host as client using Clienter,
 * if there is no host, then will proceed as host.
 * Currently the GUI does not launch until the host accepts a client.
 */
try {
    System.out.println(x:"pre connect as client");
    clienter.startConnection(ipAddress, PORT);
    System.out.println(x:"post connect as client");
} catch (Exception e) {
    System.out.println(x:"connect as client failed, into try/catch");
    isHost = true;
} finally {
    if (isHost) {
        System.out.println(x:"pre start as host");
        hoster.start(PORT); // Halts inside here until client connects.
        System.out.println(x:"post start as host");
    }
}
```

Messager takes the password from Login, and creates a 256-bit secret key using password-based-key-derivative function (PBKDF2) with HMAC SHA-1. The key is generated by combining the password with a salt byte array. In my application, the salt is determined using the current date. Thus, even when using the same password, a different key is generated daily. The salt enhances security, firstly, by ensuring the hash generated by SHA-1 is not the same hash generated by the password alone, and secondly a malicious actor obtaining the key itself will only have a good key for at most 24 hours.

Messager generates a Cipher using AES using CBC as its mode of operation, with PKCS7 padding. The initialization vector is generated using a hash of the password. The Cipher has a method for getting an IV, but it seemed to generate inconsistent IVs between users, thus inhibiting communication. Hence a custom IV is generated using the hashed password.

Messager keeps an ArrayList of messages, and each time a new Message is created, Messager grabs the last block of the previous Message, and feeds that to the new Message as the initialization vector. This is how CBC is maintained across messages, and ensures that duplicate plaintexts produce different ciphertexts. In essence this ensures that while messages are separated as objects, the entire conversation is unified as one long CBC ciphertext. The only potential problem with this methodology is in the event that the two users' "timelines" of messages don't match, (perhaps because two messages from each end were sent at nearly the same time, appearing in different order for either user), then initialization vectors for both users would be out of sync and messages would become unreadable.

```
/*
 * Handles message sending here.
 * Text typed in text field will send on press of enter key
 */
textField.setOnKeyPressed(e -> {
    if (e.getCode().equals(KeyCode.ENTER)) {
        try {
            byte[] msgBytes = textField.getText().getBytes(charsetName:"UTF-8"); // String in textField converted to bytes
            if (messages.size() != 0) {
                iv = messages.get(messages.size() - 1).getLastBlock(); // sets IV to the last block of the
                                                                       // previous message
            }
            Message outgoingMessage = new Message(msgBytes, k, ciph, iv, encrypted:false); // creates Message object.
                                                                                           // Encryption happens inside
                                                                                           // constructor.
            messages.add(outgoingMessage);
            textField.clear();
            print(outgoingMessage); // outgoing Message is displayed on screen, both ciphertext and plaintext.
            if (isHost) {
                hoster.send(outgoingMessage.getCipherText()); // sends as host if host
            } else {
                clienter.send(outgoingMessage.getCipherText()); // sends as client if client
            }

        } catch (Exception e1) {
            e1.printStackTrace();
        }
    }
});
```

```
while (hoster.isConnected()) { // receiving loop

    byte[] msgBytes;
    if ((msgBytes = hoster.receive()) != null) {

        if (messages.size() != 0) {
            iv = messages.get(messages.size() - 1).getLastBlock(); // creates IV from last block
                                                                   // of last message
        }
        Message incomingMessage = new Message(msgBytes, k, ciph, iv, encrypted:true); // creates Message
                                                                                      // object.
                                                                                      // Decryption
                                                                                      // happens here.
        messages.add(incomingMessage);
        print(incomingMessage); // prints Message to screen, both ciphertext and plain text.
    }
}
```

As aforementioned, the actual encryption and decryption happens inside the Message class, when a Message is constructed by Messager. The constructor accepts a byte array containing the message, the encryption key, the cipher, the initialization vector, and a boolean that determines whether the given message is encrypted or not. If the given message is encrypted, the constructor stores the encrypted message as the Message ciphertext, and decrypts the message and stores it as the Message plaintext. If the given message isn't encrypted, the constructor stores the message as plaintext, encrypts it, and stores the ciphertext. Thus when Messager constructs a Message, the Message promptly contains the text in both cipher and plain form, and both are accessible using getters.

The Message constructor also performs Message Authentication in an Encrypt-then-Authenticate scheme. After encrypting a message, the ciphertext is hashed and the resulting hash is appended to the end of the byte array. During decryption, the hash is unappended, the ciphertext is hashed, and the two hashes are compared to authenticate the message. Any man in the middle attack in which a malicious

actor attempt to modify the message would result in the hash of the ciphertext being different from the hash appended to the end of the message, thus exposing the attack.

```
/*
 * This branch stores the given encrypted message in cipherText, decrypts the
 * message, and stores the decrypted message in plainText
 */
if (encrypted) {
    cipherText = text;

    try {
        cipher.init(Cipher.DECRYPT_MODE, key, new IvParameterSpec(iv));

        if (!compareHash(sha1(unappendHash(cipherText)), getAppendedHash(cipherText))) {
            throw new Exception(message:"ALERT: This message has been modified.");
        }
        byte[] cipherTextWithHashUnAppended = unappendHash(cipherText);
        plainText = cipher.doFinal(cipherTextWithHashUnAppended); // Decryption happens here.
    } catch (Exception e) {
        System.out.println(e.getMessage());
        e.printStackTrace();
    }
}
/*
 * This branch stores the given unencrypted message in plainText, encrypts it,
 * and stores the encrypted message in cipherText
 */
else {
    plainText = text;

    try {
        cipher.init(Cipher.ENCRYPT_MODE, key, new IvParameterSpec(iv));

        cipherText = cipher.doFinal(plainText); // Encryption happens here.
        cipherText = appendHash(cipherText); // Hash is appended to cipherText, completing
                                             // Encrypt-then-Authenticate scheme

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

After constructing the Message, Messager sends the ciphertext byte array over the socket via an ObjectOutputStream inside the Hoster or Clienter. Hoster or Clienter wraps the byte[] into an Object via ByteWrapper, in order to preserve the byte[]'s integrity. Sending the byte[] without wrapping it results in the byte[] having bytes added, triggering the authentication check and ruining the decyrption. By wrapping it using ByteWrapper, the ByteWrapper object is sent, then received, then the unperturbed byte[] can be accessed via getter, and then passed along for proper decryption. The Message objects themselves are not transmitted, the obvious reason being that Message objects contain both the plaintext and ciphertext and therefore are not secure for transmission.