

<b>Course Title:</b>	Network Security
<b>Course Number:</b>	COE817
<b>Semester/Year (e.g.F2016)</b>	W2024

<b>Instructor:</b>	Dr. Truman Yang
--------------------	-----------------

<i>Assignment/Lab Number:</i>	Final Project
<i>Assignment/Lab Title:</i>	Final Project

<i>Submission Date:</i>	April,5,2024
<i>Due Date:</i>	April,12,2024

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Soubra	Karim	500966625	6	KS
Valiakhmetova	Polina	500960865	1	PV

\*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: <http://www.ryerson.ca/senate/current/pol60.pdf>

## Introduction

A banking system requires transferring sensitive information through unsecured communication channels between clients and servers, in this case, the ATM is the client and the server is the bank. Any attacker can obtain sensitive information and cause damage to the users. As a result, a banking system must be secure to protect the sensitive information and transactions going through the ATMs and banks. There are various types of attacks the banking system must be secure from such as replay attacks, and man-in-the-middle attacks. A replay attack occurs when an attacker replays a successful transmission of a previous message to gain unauthorized access. Man-in-the-middle attacks occur when an attacker intercepts messages between the bank and ATM, the attacker can then only eavesdrop or alter the messages. The goal is to build a banking system that is secure and resistant to such attacks. Public keys and symmetric cryptography are used to combat such attacks, as well as Nonces to make the communication between the ATM and bank resistant to replay attacks. A log file is used to keep track of all actions happening between the ATM and the bank. The log file can be used to detect if any malicious activity has occurred.

## Design

The design of the banking system must take into account network security since the main requirement of the banking system is to be secure. Figure 1 shows a high-level architecture diagram of the banking system implemented. The bank server is designed to handle multiple ATM clients connected to it simultaneously. The bank server creates a bank server thread to handle each client connected to the bank.

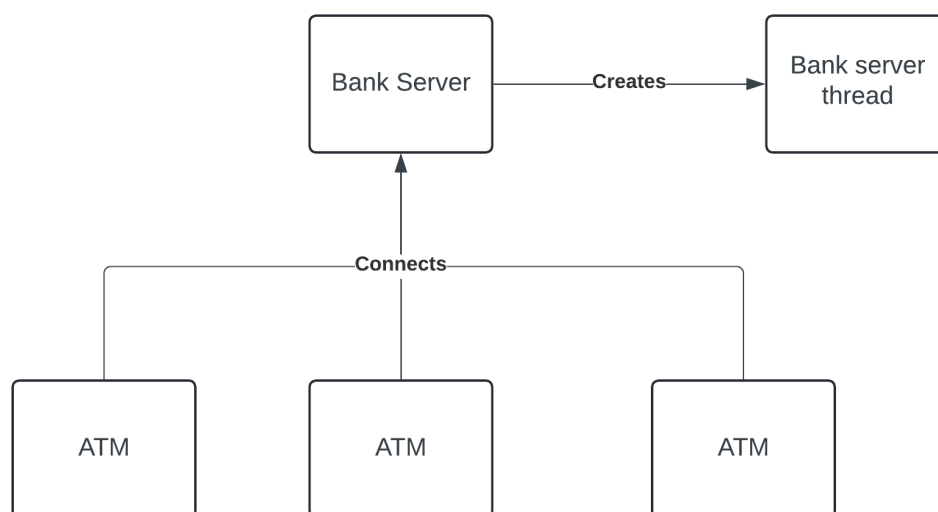


Figure 1. Banking system architecture.

The three components making up the banking system software are the ATM, Bank Server, and Bank Server Thread. Figure 2 shows the class diagram that was designed for the ATM. The ATM generates its own pair of public and private keys in order to communicate with the bank server, which operates as the KDC.

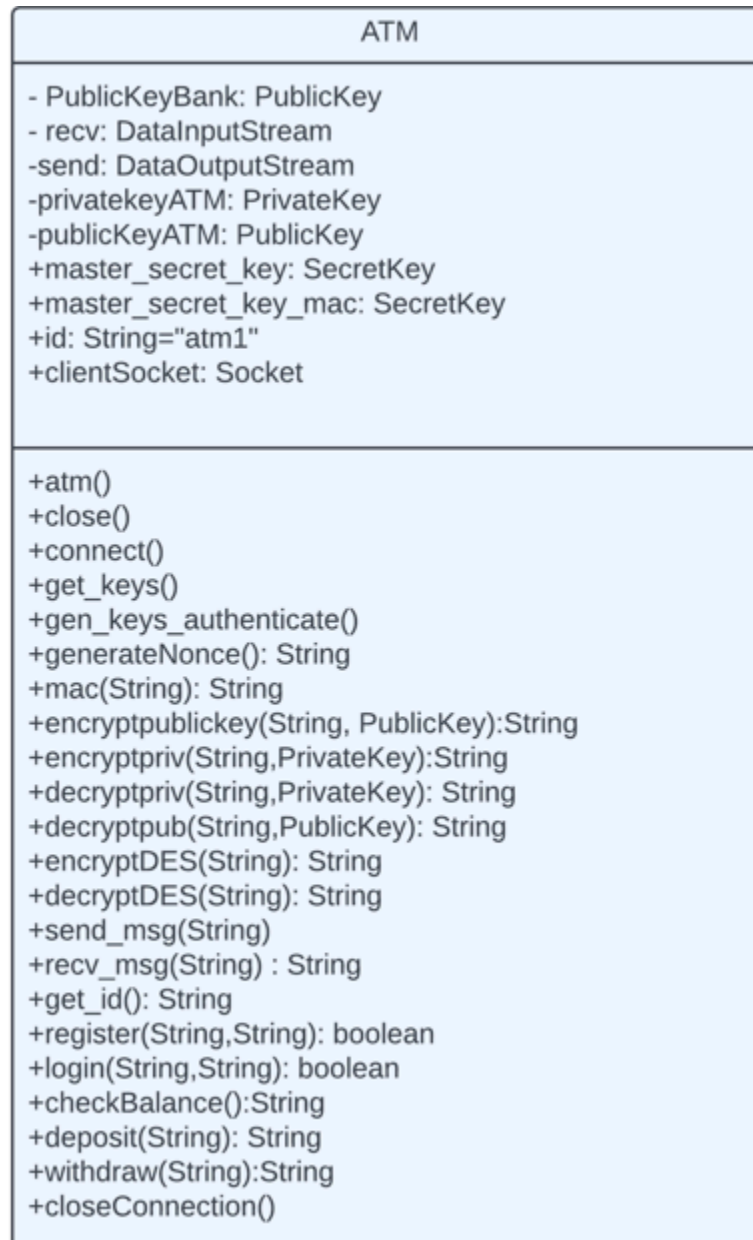


Figure 2. ATM class diagram.

The ATM uses its own private and public keys to begin communication with the bank server. The ATM class also has the bank's public key in order to facilitate initial communication and generate session keys for a user's session. The session keys generated are the `master_secret_key` and `master_secret_key_mac` respectively. The second key mentioned is used for message authentication in order to ensure the message's integrity and to confirm the sender's identity. The ATM also contains support functions in order to connect to the bank. The `close`, `closeConnection`, and `connect` functions connect and disconnect the ATM from the bank server. When a user is done with the ATM, the user closes the ATM and the `closeConnection` function disconnects the user from the bank. Upon successful signup and successful login the ATM then initiates the process of generating session keys for the user. The process of generating session keys is done using the `get_keys` and `gen_keys_authenticate` functions. The `get_keys` function obtains the bank's public key and `gen_keys_authenticate` generates the master secret keys that are used for encrypting session data and message authentication. The `generateNonce` function in the ATM class is there to help keep the messages sent immune to replay attack since the `generateNonce` function generates a new number that is to be sent with the message, therefore if the bank detects the same nonce twice, it can detect a replay attack. The `Mac` function in the atm class diagram is there to sign messages generated by the ATM, that is to be sent to the bank. The `Mac` function is important because it also informs the bank about who the user is since no one other than the atm and bank has access to the master session key. The following functions `encryptpublickey`, `encryptpriv`, `encryptDES`, `decryptpriv`, `decryptpub`, and `decryptDES` are used to encrypt and decrypt messages with different keys. The master secret key uses `encryptDES` and `decryptDES` since these functions handle symmetric encryption and decryption. The master secret key is generated every time the user logs in. The symmetric encryption and decryption use DES due to its security, simplicity, and speed. To increase DES's security, the `mac` function is used while encrypting the user's message, and the `mac` signature is attached to the user's encrypted message. The `Mac` signature is used to maintain the message's integrity and the bank uses it as a check in order to ensure the encrypted message has not been tampered with. The `encryptpublickey`, `encryptpriv`, `decryptpriv`, and `decryptpub` are used when establishing a secret key and authenticating the bank to the user and vice versa. The ATM has the bank's public key and the bank has the ATM's public key. The public and private keys are generated using the RSA algorithm; these add an extra layer of security. After generating the master secret key, the public and private keys are no longer used. This is a hybrid approach that utilizes the benefits of both symmetric and public key encryption. The `send_msg` and `recv_msg` functions are used to send and receive data through the socket to the bank, these functions are used in conjunction with the various encryption and decryption functions the atm has since all messages passing through the sockets are encrypted. The `register`, `login`, `check balance`, `deposit`, and `withdraw` functions are used for the user to interact with their account in the bank. When the user uses the GUI, the GUI then calls the functions in order to execute the requested action. The `register`, `login`, `checkbalance`, `deposit`, and `withdraw` functions use the master secret key for communication with the bank, each function sends the data encrypted using the master secret key alongside the

message authentication signature in order to ensure the messages are transmitted securely. Every message transmitted to the bank contains a nonce to prevent replay attacks, therefore designing a communication system that is robust and secure against attacks by using multiple keys, a mix of public and symmetric encryption, message authentication signatures, and nonces.

Figure 3 shows the bank server's design. The bank server's class diagram shows how the code was designed in order to keep the messages secure and enable the bank to handle multiple ATM clients at the same time.

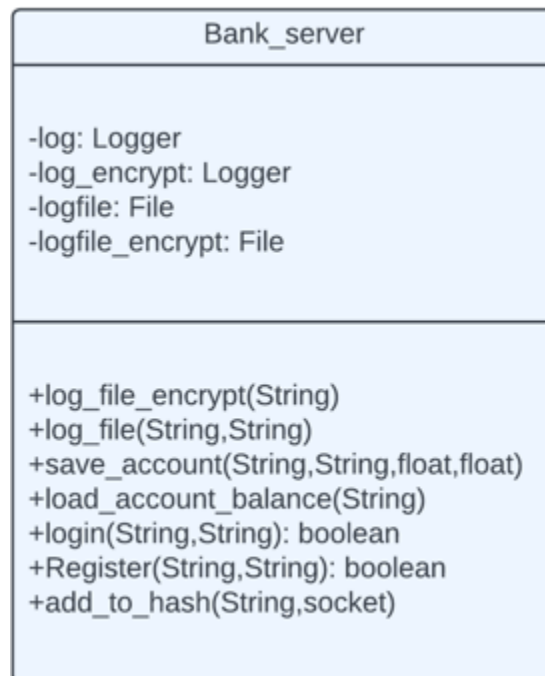


Figure 3. Bank server class diagram.

Part of the requirements of a secure banking system is to design a log file that keeps track of every action the user takes. When a user deposits, withdraws, checks balance, logs in, and registers it is recorded in a log file. The log file keeps track of every user's action by recording it in the log file, with every entry in the log file having the customer's ID, action taken, and a timestamp of when the action has been taken. The log file must be encrypted in order to keep the log file secure. As a result in the design of the bank server, the log\_file\_encrypt function takes in an encrypted string that contains the customer ID, action, and timestamp and records it in a log file. The log\_file function does the same thing as the log\_file\_encrypt function except the data written is not encrypted, this is mainly due to confirming the data being encrypted in the log file is correct. Save\_account saves the user's information and account balance so that a user can log in at any time in the future use any ATM and have their data saved. The load\_account\_balance executes upon a successful login, the load\_account\_balance loads a user's saved account balance such that the user can log out and log back in, and still be able to check their account balance. The login function handles the logic for when a user logs in. The login function checks whether

the user has registered and whether the password given by the user is correct. If the password is incorrect then the user would not be able to access their account. The register function handles the logic for signing up.

Figure 4 shows the design for the bank\_serverthread. The bank\_serverthread is a thread that is created by the bank\_server in order to handle clients. The bank\_server only takes care of the initial client connection and keeps listening for new connections. The bank\_server passes the new socket after the client connection to the bank\_serverthread in order to continue communicating with the client. Every new client connection requires a new bank\_serverthread to execute.

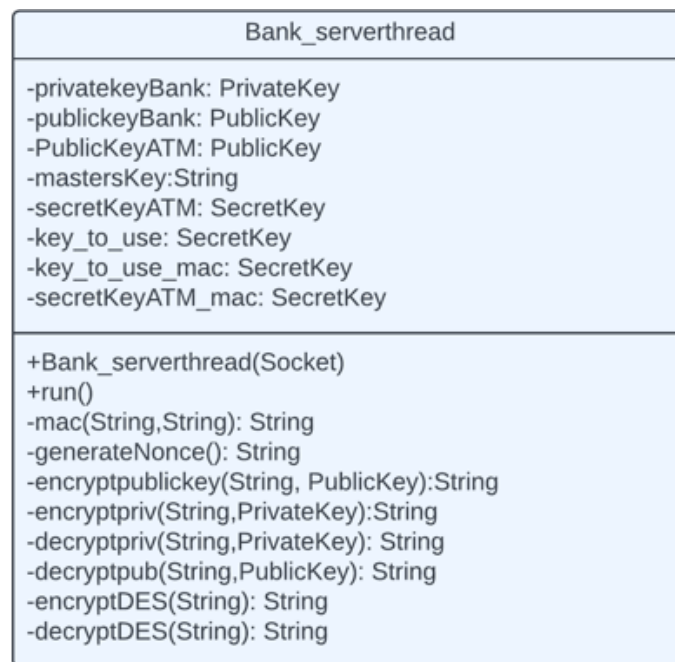


Figure 4. Bank server thread class diagram.

The bank\_serverthread starts executing when its run function is called. The bank\_serverthread keeps executing until it returns. The bank\_serverthread has the ATM's public key and it generates its own public/private key pair, in which the client obtains the bank server's public key. The public keys are used in the initial phases of the connection before the generation of the master secret key. The Mac function is used to generate the digital signature that is attached to every encrypted message being sent. The Mac function can also be used to check the digital signatures of incoming messages in order to ensure the integrity of the messages being received. The encryptpublickey, encryptpriv, decryptpriv, decryptpub, encryptDES, decryptDES functions are used to encrypt and decrypt various messages that are being sent and received. There are multiple encrypt and decrypt functions due to the use of both symmetric and asymmetric encryption methods for key distribution and message sending. The bank\_serverthread also doubles as a KDC, the bank\_serverthread generates a session key for a

user upon each successful login. This keeps the banking system more secure since there are multiple layers of security between the ATM and the bank.

Figure 5 shows the implementation of the various encrypting and decrypting functions. The encrypt and decrypt functions are similar on both the client and server side.

```
private static String encryptpublickey(String plainText, PublicKey publicKey) throws Exception {
    Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
    cipher.init(Cipher.ENCRYPT_MODE, publicKey);
    byte[] secretMessageBytes = plainText.getBytes(StandardCharsets.UTF_8);
    byte[] encryptedMessageBytes = cipher.doFinal(secretMessageBytes);
    return Base64.getEncoder().encodeToString(encryptedMessageBytes);
}

private static String encryptpriv(String plainText, PrivateKey privateKey) throws Exception {
    Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
    cipher.init(Cipher.ENCRYPT_MODE, privateKey);
    byte[] secretMessageBytes = plainText.getBytes(StandardCharsets.UTF_8);
    byte[] encryptedMessageBytes = cipher.doFinal(secretMessageBytes);
    return Base64.getEncoder().encodeToString(encryptedMessageBytes);
}

private static String decryptpriv(String encryptedText, PrivateKey privateKey) throws Exception {
    Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
    cipher.init(Cipher.DECRYPT_MODE, privateKey);
    byte[] decodedMessage = Base64.getDecoder().decode(encryptedText);
    byte[] decryptedBytes = cipher.doFinal(decodedMessage);
    return new String(decryptedBytes, StandardCharsets.UTF_8);
}

private static String decryptpub(String encryptedText, PublicKey publicKey) throws Exception {
    Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
    cipher.init(Cipher.DECRYPT_MODE, publicKey);
    byte[] decodedMessage = Base64.getDecoder().decode(encryptedText);
    byte[] decryptedBytes = cipher.doFinal(decodedMessage);
    return new String(decryptedBytes, StandardCharsets.UTF_8);
}

private static String encryptDES(String plainText, SecretKey secretkey) throws Exception {
    Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
    cipher.init(Cipher.ENCRYPT_MODE, secretkey);
    byte[] encryptedBytes = cipher.doFinal(plainText.getBytes());
    return Base64.getEncoder().encodeToString(encryptedBytes);
}

private static String decryptDES(String encryptedText, SecretKey secretkey) throws Exception {
    Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
    cipher.init(Cipher.DECRYPT_MODE, secretkey);
    byte[] decryptedBytes = cipher.doFinal(Base64.getDecoder().decode(encryptedText));
    return new String(decryptedBytes);
}
```

Figure 5. Encryption and decryption implementation.

The encryption and decryption functions in Figure 5 are crucial to a secure banking system. The implementation of the encryption and decryption function are similar with the main difference being the type of key used for encryption and decryption. Both the encryption and decryption functions require the use of the cipher library, decrypting or encrypting plaintext requires the conversion of the plaintext to a byte array that is then encrypted or decrypted and converted back to either plain text or cipher text depending on whether the user is decrypting or encrypting the text. The cipher doFinal function either encrypts or decrypts the string depending on whether the string given to the cipher is plain text or cipher text. If the string given to the cipher is cipher text then the cipher will attempt to decrypt and if the string given to the cipher is plain text then the cipher will attempt to encrypt. The PKC1Padding and PKC5Padding found in Figure 5 when instantiating the cipher are used to ensure that the message being encrypted or decrypted is of the right size. When initializing the cipher in Figure 5, the key that the cipher will use is given to it. In encryptDES for example the key given to the function is just a SecretKey

used for symmetric encryption, and the key given to the function `encryptpriv` is the bank's private key although the keys assigned to the cipher are different, the general encryption and decryption process remains the same. The general process to encrypt and decrypt plaintext or ciphertext is the following: a cipher instance gets created and initialized with the correct padding and assigned a key, the plaintext or ciphertext is converted into an array of bytes, and the `doFinal` function is called, finally the encrypted or decrypted array of bytes that is returned from the `doFinal` function is converted back to a string and return to the user.

Figure 6 shows the function used to generate a nonce that will be a part of a message being transmitted. The `generateNonce` function is the same for both the ATM and bank.

```
private static String generateNonce() {
    Random rand_num = new Random();
    return "" + rand_num.nextInt(100);
}
```

Figure 6. Generate nonce function.

The `generate nonce` function is a function that generates a random integer. This ensures that replay attacks are not possible since a nonce is a number that is only used once and if the recipient detects that the nonce has been used then the message is dropped.

Figure 7 shows the `mac` function for the bank. The function checks the ID of the recipient in order to use the correct master secret key for the corresponding bank-ATM session.

```
private static String mac(String plainText,String id_atm) throws NoSuchAlgorithmException, InvalidKeyException{
    Mac signer = Mac.getInstance("HmacSHA256");
    if("atm1".equals(id_atm)){
        signer.init(secretKeyATM1_mac);
    }else if("atm2".equals(id_atm)){
        signer.init(secretKeyATM2_mac);
    }else if("atm3".equals(id_atm)){
        signer.init(secretKeyATM3_mac);
    }
    byte[] signed_msg=signer.doFinal(plainText.getBytes());
    return new String(signed_msg, StandardCharsets.UTF_8);
}
```

Figure 7. Mac function bank\_server.

Figure 8 shows the `mac` function for the ATM client, the main difference between the `mac` function in Figure 7 and Figure 8 is that the `mac` function for the ATM only needs to keep track of its own master secret key meanwhile the `bank_server` must keep track of all the master secret keys and make sure to use the correct master secret key.

```

public String mac(String plainText) throws NoSuchAlgorithmException, InvalidKeyException{
    Mac signer = Mac.getInstance("HmacSHA256");
    signer.init(master_secret_key_mac);
    byte[] signed_msg=signer.doFinal(plainText.getBytes());
    return new String(signed_msg, StandardCharsets.UTF_8);
}

```

Figure 8. Mac function ATM client.

The Mac function in Figures 7 and 8 generates the digital signature for message authentication. The digital signature of a message is generated twice, once at the receiver's side and once when the message is being transmitted. If the hash that was generated at the receiver's side is equal to the hash it received in the digital signature then the recipient can confirm that the message is authentic and not tampered with.

Figures 9 and 10 show the key generation for the bank\_server. The bank\_server's role as a KDC means that the bank\_server must also generate and distribute session keys to its clients.

```

//GENERATE PUBLIC AND PRIVATE KEY FOR RSA
KeyPairGenerator KeyGenKDC = KeyPairGenerator.getInstance("RSA");
KeyGenKDC.initialize(2048);
KeyPair pairKDC = KeyGenKDC.generateKeyPair();
PrivateKey privateKeyKDC = pairKDC.getPrivate();
PublicKey publicKeyKDC = pairKDC.getPublic();

ServerSocket serverSocket = new ServerSocket(6789); // Example port number
File publicKeyFileKDC = new File("publicBank.key");
File privateKeyFileKDC = new File("privateBank.key");

X509EncodedKeySpec x509EncodedKeySpec_priv = new X509EncodedKeySpec(privateKeyKDC.getEncoded()); //saving private key
X509EncodedKeySpec x509EncodedKeySpec = new X509EncodedKeySpec(publicKeyKDC.getEncoded());
Files.write(privateKeyFileKDC.toPath(), x509EncodedKeySpec_priv.getEncoded());
Files.write(publicKeyFileKDC.toPath(), x509EncodedKeySpec.getEncoded());

```

Figure 9. Generating bank's public and private keys.

```

//import all key files
Path path = Paths.get("privateBank.key");
byte[] bytes = Files.readAllBytes(path);
PKCS8EncodedKeySpec spec = new PKCS8EncodedKeySpec(bytes);
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
privatekeyBank = keyFactory.generatePrivate(spec);

File publicKeyBank = new File("publicBank.key");
byte[] bytes2 = Files.readAllBytes(publicKeyBank.toPath());
X509EncodedKeySpec spec2 = new X509EncodedKeySpec(bytes2);
KeyFactory keyFactory2 = KeyFactory.getInstance("RSA");
publickeyBank = keyFactory2.generatePublic(spec2);

//create DES keys
String[] mkey_atm1 = mastersKeyATM1.split("\\|");
String[] mkey_atm2 = mastersKeyATM2.split("\\|");
String[] mkey_atm3 = mastersKeyATM3.split("\\|");

//generate session keys
byte[] keyBytesATM1 = mkey_atm1[0].getBytes();
byte[] keyBytesATM2 = mkey_atm2[0].getBytes();
byte[] keyBytesATM3 = mkey_atm3[0].getBytes();
byte[] keyBytesATM1_mac = mkey_atm1[1].getBytes();
byte[] keyBytesATM2_mac = mkey_atm2[1].getBytes();
byte[] keyBytesATM3_mac = mkey_atm3[1].getBytes();
secretKeyATM1 = new SecretKeySpec(keyBytesATM1, "DES");
secretKeyATM2 = new SecretKeySpec(keyBytesATM2, "DES");
secretKeyATM3 = new SecretKeySpec(keyBytesATM3, "DES");
secretKeyATM1_mac = new SecretKeySpec(keyBytesATM1_mac, "DES");
secretKeyATM2_mac = new SecretKeySpec(keyBytesATM2_mac, "DES");
secretKeyATM3_mac = new SecretKeySpec(keyBytesATM3_mac, "DES");

```

Figure 10. Generating master secret keys.

Figure 9 shows the bank generating their public and private keys via RSA. The private keys remain only with the bank, with no one able to get them. The bank's public keys however are distributed to the ATMs in order to facilitate communication between the bank and ATM before a session key is generated. This helps with the authentication process since the ATMs encrypt outgoing messages with the bank's public key therefore only the bank can decrypt the messages on the receiving side, since only the bank has knowledge of the private keys. The messages are sent along with a nonce therefore when the ATM receives a reply with the same nonce the ATM used and it uses the bank's public key to decrypt the reply, then the ATM knows the bank is on the other side and is authenticated. The bank serves as the KDC in the secure banking system as a result, as seen in Figure 10, the bank generates the session keys for both communication and message authentication that are known as the master secret key and master secret key MAC respectively. The bank generates the session keys for all the ATMs that are expected to connect to the bank, the session keys are distributed to the bank once the user enters the password and the bank detects that the user has entered the correct password. The session keys created are symmetric keys, the main benefit of symmetric keys is their speed compared to using public key encryption. Therefore, when a user logs into an ATM the initial phases of authentication and key distribution use public and private keys but for the remainder of the session, the ATM uses session keys as long as the session is valid.

Figure 11 shows the private and public key generation of the ATM. The ATM only generates its own public and private keys but receives the session keys from the bank upon successful login and authentication. The ATM's public key is passed to the bank in order to facilitate initial communication between the two before a session is created.

```
public atm() throws IOException, NoSuchAlgorithmException, InvalidKeySpecException{
    KeyPairGenerator keyGenA = KeyPairGenerator.getInstance("RSA");
    keyGenA.initialize(2048);
    KeyPair pairATM1 = keyGenA.generateKeyPair();
    privateKeyATM1 = pairATM1.getPrivate();
    publicKeyATM1 = pairATM1.getPublic();

    // Save public key A to a file
    File publicKeyFileATM1 = new File("publicATM1.key");
    X509EncodedKeySpec x509EncodedKeySpec = new X509EncodedKeySpec(publicKeyATM1.getEncoded());
    Files.write(publicKeyFileATM1.toPath(), x509EncodedKeySpec.getEncoded());
}
```

Figure 11. ATM public and private key generation.

Figure 12 shows how the ATM gets the bank's public keys. There are four ways to pass keys between two users, the method of passing keys used in Figure 12 is physical delivery since using public and private keys, the security risk from exposing the public key is minimal since the attacker cannot reverse engineer and obtain the private keys. The combination of double encryption using both the sender's private key and the receiver's public key helps combat the man-in-the-middle attack, as a result passing public keys physically is a safe and effective

option, as seen in Figure 12. The method of physically distributing the public key is also used by the bank to obtain the ATM's public key.

```
public void get_keys() throws IOException, NoSuchAlgorithmException, InvalidKeySpecException{
    // Import the public B key
    File publicKeyFileBank = new File("publicBank.key");
    byte[] bytes = Files.readAllBytes(publicKeyFileBank.toPath());
    X509EncodedKeySpec spec = new X509EncodedKeySpec(bytes);
    KeyFactory keyFactory = KeyFactory.getInstance("RSA");
    publicKeyBank = keyFactory.generatePublic(spec);
}
```

Figure 12. Passing the bank's public key.

Figure 13 shows the key distribution protocol that is used in the banking system. The distribution protocol is executed after every account registration and after every successful login.

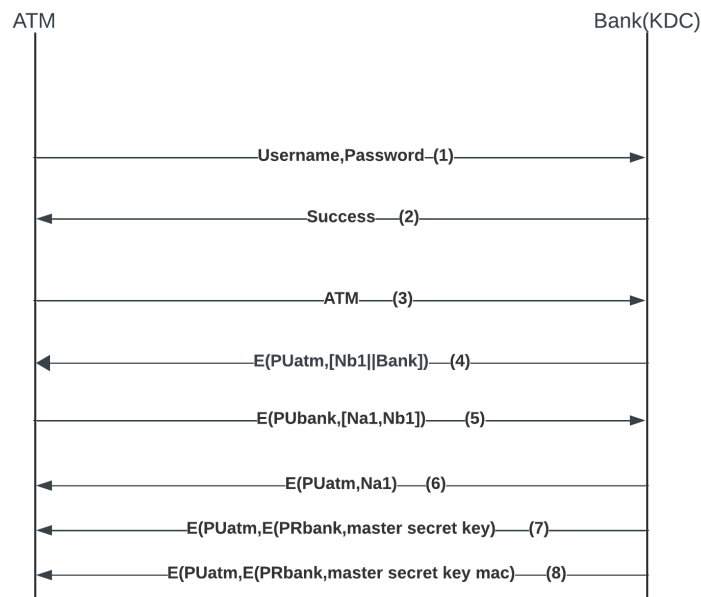


Figure 13. KDC protocol.

The protocol starts with a user entering their username and password, if the username and password are correct then the protocol continues in order to create session keys for the user. After the first message is sent, if the bank decides that the username and password are correct, then a success message is sent to the ATM. Next, the ATM sends its ID to the bank and the authentication phase takes place. After messages 4, 5, and 6 the bank and the ATM would have authenticated each other. This is due to the fact that messages 5 and 6 return the correct nonces to the respective senders. Messages 7 and 8 are the key distribution phase, both messages are first encrypted with the sender's private key and then with the receiver's public key; this is to ensure confidentiality and security of the message since only the ATM can fully decrypt the message. Message 7 distributes the master secret key and message 8 distributes the master secret key used for message authentication which is also known as the MAC key.

Figures 14 and 15 show the implementation of the ATM in the KDC protocol in Figure 13. The messages are sent and received using send UTF functions and receive UTF functions. The messages are encrypted and decrypted using the functions in Figure 5. After the protocol is executed the ATM would have obtained the master secret key and the master secret key mac as seen in Figure 15.

```
public void gen_keys_authenticate() throws IOException, Exception{
    // Recieve E(PUB, [NK1 || IDK])
    String message1 = recv.readUTF();
    String[] messageParts = message1.split("\\\\|\\\\|");
    String message1part1 = messageParts[0];
    String message1part2 = messageParts[1];
    message1part1 = decryptpriv(message1part1, privatekeyATM1);
    message1part2 = decryptpriv(message1part2, privatekeyATM1);
    System.out.println("Recieved message 1(decrypted): " + message1part1 + "||" + message1part2);

    // Send E(PUB, [NB || HK2]) to KDC
    String Na_1 = generateNonce();
    System.out.println("Sending message 2: " + Na_1 + "||" + message1part1);
    String message2part1 = encryptpublickey(Na_1, publicKeyBank);
    String message2part2 = encryptpublickey(message1part1, publicKeyBank);
    String message2 = message2part1 + "||" + message2part2;
    send.writeUTF(message2);
    System.out.println("Sent message 2: " + message2);

    // Recieve E(PUB, NK2)
    String message3 = recv.readUTF();
    message3 = decryptpriv(message3, privatekeyATM1);
    System.out.println("Recieved message 3(decrypted): " + message3);

    // Recieve E(PUB, E(PRK, KB))
    String message4 = recv.readUTF();
    System.out.println("Recieved message master secret key encrypted: " + message4);
    String[] message4Parts = message4.split("\\\\|\\\\|");
    String message4part1 = message4Parts[0];
    String message4part2 = message4Parts[1];
    message4part1 = decryptpriv(message4part1, privatekeyATM1);
    message4part2 = decryptpriv(message4part2, privatekeyATM1);
    String message4final = message4part1+message4part2;
    message4final = decryptpub(message4final, publicKeyBank);
    System.out.println("Decrypted message master secret key: " + message4final);
    byte[] decodedKey = Base64.getDecoder().decode(message4final);
    master_secret_key = new SecretKeySpec(decodedKey, 0, decodedKey.length, "DES");
    System.out.println("Recieved master secret key: " + master_secret_key);
}
```

Figure 14. Implementation of ATM authentication and key distribution protocol.

```
String message4_mac = recv.readUTF();
System.out.println("Recieved message MAC master secret key encrypted: " + message4_mac);
String[] message4Parts_mac = message4_mac.split("\\\\|\\\\|");
String message4part1_mac = message4Parts_mac[0];
String message4part2_mac = message4Parts_mac[1];
message4part1_mac = decryptpriv(message4part1_mac, privatekeyATM1);
message4part2_mac = decryptpriv(message4part2_mac, privatekeyATM1);
String message4final_mac = message4part1_mac+message4part2_mac;
message4final_mac = decryptpub(message4final_mac, publicKeyBank);
System.out.println("Decrypted message MAC master secret key: " + message4final_mac);
byte[] decodedKey_mac = Base64.getDecoder().decode(message4final_mac);
master_secret_key_mac = new SecretKeySpec(decodedKey_mac, 0, decodedKey_mac.length, "DES");
System.out.println("Recieved message MAC master secret key: " + master_secret_key_mac);
}
```

Figure 15. Implementation of ATM authentication and key distribution protocol part 2.

Figures 16 and 17 show the implementation of the bank's KDC protocol, which serves as the trusted KDC in Figure 13. Figures 14 and 15 are where the KDC distributes the session keys. After figures 14 and 15 both the bank and ATM have authenticated each other and the bank deemed it safe to share the session key with the ATM.

```

if("atm".equals(id_atm)){
    File publicKeyFileatm = new File("publicATM.key");
    byte[] bytesatm = Files.readAllBytes(publicKeyFileatm.toPath());
    X509EncodedKeySpec spec_atm = new X509EncodedKeySpec(bytesatm);
    KeyFactory keyFactory_atm = KeyFactory.getInstance("RSA");
    publicKeyATM = keyFactory_atm.generatePublic(spec_atm);
    String Bnonce = generateNonce();
    // Send E(PUB, [BNonce || ID])

    System.out.println("Sending message 1: " + Bnonce + "||" + id_bank);
    String message1part1_atm = encryptPublicKey(Bnonce, publicKeyATM);
    String message1part2_atm = encryptPublicKey(id_bank, publicKeyATM);
    String message1_atm = message1part1_atm + "||" + message1part2_atm;
    send.writeUTF(message1_atm);
    System.out.println("Sent message 1: " + message1_atm);

    // Receive (PUB, [BNonce || ID])
    String message2_atm = recv.readUTF();
    String[] messageParts_atm = message2_atm.split("\\|");
    String message2part1_atm = messageParts_atm[0];
    String message2part2_atm = messageParts_atm[1];
    message2part1_atm = decryptPriv(message2part1_atm, privateKeyBank);
    message2part2_atm = decryptPriv(message2part2_atm, privateKeyBank);
    System.out.println("Received message 2 (decrypted): " + message2part1_atm + "||" + message2part2_atm);

    // Send E(PUB, NK1)
    System.out.println("Sending message 3: " + message2part2_atm);
    String message3_atm = encryptPublicKey(message2part2_atm, publicKeyATM);
    send.writeUTF(message3_atm);
    System.out.println("Sent message 3: " + message3_atm);

    // Send E(PUB, E(PRK, KA))
    String KA_atm = Base64.getEncoder().encodeToString(secretKeyATM.getEncoded());

```

Figure 16. Implementation of bank authentication and key distribution protocol.

```

// Send E(PUB, E(PRK, KA))
String KA_atm = Base64.getEncoder().encodeToString(secretKeyATM.getEncoded());

//String KB = "" + secretKeyB;
System.out.println("Sending message 4: " + secretKeyATM);
String message4_atm = encryptPriv(KA_atm, privateKeyBank);
String firstHalf_atm = message4_atm.substring(0, message4_atm.length() / 2);
String secondHalf_atm = message4_atm.substring(message4_atm.length() / 2);

firstHalf_atm = encryptPublicKey(firstHalf_atm, publicKeyATM);
secondHalf_atm = encryptPublicKey(secondHalf_atm, publicKeyATM);
String message4Final_atm = firstHalf_atm + "||" + secondHalf_atm;

send.writeUTF(message4Final_atm);
System.out.println("Sent message 4: " + message4Final_atm);

String KA_atm_mac = Base64.getEncoder().encodeToString(secretKeyATM_mac.getEncoded());

//String KB = "" + secretKeyB;
System.out.println("Sending message 4: " + secretKeyATM_mac);
String message4_atm_mac = encryptPriv(KA_atm_mac, privateKeyBank);
String firstHalf_atm_mac = message4_atm_mac.substring(0, message4_atm_mac.length() / 2);
String secondHalf_atm_mac = message4_atm_mac.substring(message4_atm_mac.length() / 2);

firstHalf_atm_mac = encryptPublicKey(firstHalf_atm_mac, publicKeyATM);
secondHalf_atm_mac = encryptPublicKey(secondHalf_atm_mac, publicKeyATM);
String message4Final_atm_mac = firstHalf_atm_mac + "||" + secondHalf_atm_mac;

send.writeUTF(message4Final_atm_mac);
System.out.println("Sent message 4: " + message4Final_atm_mac);
Bank_server.log_file(id_atm, "Authentication, verification, and key distribution");
Bank_server.log_file_encrypt(encryptPublicKey(id_atm+"||"+"Authentication, verification, and key distribution", publicKeyBank));
//load file info

key_to_use = secretKeyATM;
key_to_use_mac = secretKeyATM_mac;

```

Figure 17. Implementation of bank authentication and key distribution protocol part 2.

Figures 18 and 19 show the implementation of the check balance functionality for the bank and ATM. The command “c” is sent along with a nonce by the ATM in order to request a check balance from the bank. The message is encrypted with the master secret key and sent alongside a digital signature for message authentication. This is done in order to keep the messages secure and confidential. Figure 19 shows the bank receiving the check balance message from the ATM. The bank first logs the action into its log book and then checks the digital signature with the digital signature calculated by the bank, and checks if the signatures match before sending the requested information back to the ATM. If the check is successful then the bank sends the balance back to the ATM. The bank encrypts and signs the message before sending the balance information back to the ATM. A similar check occurs on the ATM side before displaying the balance information.

```

public String checkBalance() throws Exception {
    //check balance : nonce||action||signature
    String cmd = "c";
    String nonce = generateNonce();
    String msg = nonce + "||" + cmd;
    System.out.println("Sending message");
    System.out.println(msg);
    String sig = mac(msg);
    String encrypt_msg = encryptDES(msg);
    String msg_final = encrypt_msg + "||" + sig;
    System.out.println("Encrypted message and signature");
    System.out.println(msg_final);
    send_msg(msg_final);

    // E(Nonce||balance)||signature
    String balance_encrypted = recv_msg();

    String[] balance_parts = balance_encrypted.split("\\\\|");
    String balance_decrypted = decryptDES(balance_parts[0]);
    String mac_new = mac(balance_decrypted);
    System.out.println("Received message encrypted");
    System.out.println(balance_encrypted);
    System.out.println("Received message decrypted");
    System.out.println(balance_decrypted);
    if(balance_parts[1].equals(mac_new)){
        String[] balance_final = balance_decrypted.split("\\|");
        System.out.println("Nonce is " + balance_final[0]);
        System.out.println("Your balance is " + balance_final[1]);
        return balance_final[1];
    }else{
        System.out.println("ERROR message corrupted");
        return "Error";
    }
}

```

Figure 18. Check balance implementation ATM.

```

if("c".equals(action) && cmd_mac.equals(cmd_parts[1])){
    Bank_server.log_file(username, "check balance");
    Bank_server.log_file_encrypt(encryptpublickey(username+"||"+ "check balance",publickeyBank));
    //reply with
    // E(nonce||balance)||signature

    String Nonce = cmd_parts_nonce_action[0];

    String reply_msg = Nonce+"||"+ String.valueOf(new_balance);
    String sig = mac(reply_msg,id_atm);
    String encrypted_msg = encryptDES(reply_msg,key_to_use);
    send.writeUTF(encrypted_msg+"||"+sig);
}

```

Figure 19. Check balance implementation bank.

Figures 20 and 21 show the implementation of the deposit functionality for the ATM and bank. The ATM gets the user's input for how much money they want to deposit in the bank. The ATM attaches a nonce to the deposit information and deposit command and encrypts it. The ATM then sends the encrypted message along with the digital signature to the bank. Upon receiving the data from the ATM the bank checks the digital signature, and if the check is successful then the bank will update the user's balance and return the new balance to the user. The bank will also log the action in the log file.

```

public String deposit(String amount) throws Exception {

    String cmd = "d";
    String nonce = generateNonce();
    //deposit : E(nonce|action|amount)||signature
    String msg = nonce+"||"+cmd+"||"+amount;
    String sig = mac(msg);
    System.out.println("Message");
    System.out.println(msg);
    String encrypt_msg = encryptDES(msg);
    System.out.println("Encrypted message");
    System.out.println(encrypt_msg);
    send_msg(encrypt_msg+"||"+sig);

    //reply
    //E(Nonce|balance)||signature
    //E(Nonce|Error|balance)||signature
    String balance_encrypted = recv_msg();
    String[] balance_parts = balance_encrypted.split("\\|\\|");
    String balance_decrypted = decryptDES(balance_parts[0]);
    String mac_new = mac(balance_decrypted);
    System.out.println("Received message");
    System.out.println(balance_encrypted);
    System.out.println("Received message decrypted");
    System.out.println(balance_decrypted);
    if(balance_parts[1].equals(mac_new)){
        String[] balance_final = balance_decrypted.split("\\|\\|");
        System.out.println("Nonce is " + balance_final[0]);
        System.out.println("Your balance is " + balance_final[1]);
        return balance_final[1];
    }else{
        System.out.println("ERROR message corrupted");
        return "Error";
    }
}

```

Figure 20. Deposit implementation ATM.

```

} else if("d".equals(action) && cmd_mac.equals(cmd_parts[1])){
    //reply
    // E(Nonce|balance)||signature
    Bank_server.log_file(username, "deposit");
    Bank_server.log_file_encrypt(encryptpublickey(username+"||"+deposit",publickeyBank));
    new_balance = new_balance + Float.parseFloat(cmd_parts_nonce_action[2]);
    String nonce = cmd_parts_nonce_action[0];

    String reply_msg = nonce+"||"+ String.valueOf(new_balance);
    String sig = mac(reply_msg,id_atm);
    String encrypted_msg = encryptDES(reply_msg,key_to_use);
    send.writeUTF(encrypted_msg+"||"+sig);
}

```

Figure 21. Deposit implementation bank.

Figures 22 and 23 show the implementation of the withdrawal functionality for both the ATM and bank. The user can enter the amount of money they would like to withdraw, but the bank ensures that the amount the user wants to withdraw is not greater than the balance. The ATM sends the information about the withdrawal action to the bank before sending the amount of money to withdraw to the bank, first, the ATM must encrypt the withdrawal amount and withdraw command with a nonce and sign the message with a digital signature. The bank checks the signature when receiving the information and action to withdraw, if the check is successful then the money will be withdrawn from their account and the new balance will be returned.

```

// withdraw amount
public String withdraw(String amount) throws Exception {
    String cmd = "w";
    String nonce = generateNonce();
    //deposit : E(nonce|action|amount)||signature
    String msg = nonce+"||"+cmd+"||"+amount;
    System.out.println("Message");
    System.out.println(msg);

    String sig = mac(msg);
    String encrypt_msg = encryptDES(msg);
    System.out.println("Encrypted message");
    System.out.println(encrypt_msg);
    send_msg(encrypt_msg+"||"+sig);

    //reply
    // E (Nonce|balance)||signature
    //or
    //E (Nonce|Error|balance)||signature
    String balance_encrypted = recv_msg();
    String[] balance_parts = balance_encrypted.split("\\|\\|");
    String balance_decrypted = decryptDES(balance_parts[0]);
    String mac_new = mac(balance_decrypted);
    System.out.println("Received message");
    System.out.println(balance_encrypted);
    System.out.println("Received message decrypted");
    System.out.println(balance_decrypted);
    if (balance_parts[1].equals(mac_new)) {
        String[] balance_final = balance_decrypted.split("\\|\\|");
        if ("L".equals(balance_final[1])) {
            System.out.println("Error");
            System.out.println("amount chosen is larger than balance");
            System.out.println("Your balance is " + balance_final[2]);
            return "Error";
        } else {
            System.out.println("Nonce is " + balance_final[0]);
            System.out.println("Your balance is "+balance_final[1]);
            return balance_final[1];
        }
    } else {
        System.out.println("ERROR message corrupted");
        return "Error";
    }
}

```

Figure 22. Withdraw implementation ATM.

```

} else if ("w".equals(action) && cmd_mac.equals(cmd_parts[1])) {
    //reply
    // E (Nonce|balance)||signature
    //or
    //E (Nonce|Error|balance)||signature
    Bank_server.log_file(username, "Withdraw");
    Bank_server.log_file_encrypt(encryptPublicKey(username+"||"+withdraw, publicKeyBank));
    float amt = Float.parseFloat(cmd_parts_nonce_action[2]);
    String Nonce = cmd_parts_nonce_action[0];

    if (amt > new_balance) {
        Bank_server.log_file(username, "Withdraw error");
        Bank_server.log_file_encrypt(encryptPublicKey(username+"||"+withdraw error, publicKeyBank));
        String Error = "L";
        String str_balance = String.valueOf(new_balance);
        String msg = Nonce + "||" + Error + "||" + str_balance;
        String sig = mac(msg, id_atm);
        String encrypted_msg = encryptDES(msg, key_to_use);
        String final_msg = encrypted_msg + "||" + sig;
        send.writeUTF(final_msg);
    } else {
        new_balance = new_balance - amt;
        String reply_msg = Nonce + "||" + String.valueOf(new_balance);
        String sig = mac(reply_msg, id_atm);
        String encrypted_msg = encryptDES(reply_msg, key_to_use);
        send.writeUTF(encrypted_msg + "||" + sig);
    }
}

```

Figure 23. Withdraw implementation bank.

## Results

Below is the GUI for the signup and login process. When a new user joins they have the option to sign in by providing a username and password and once the Register button is pressed they are shown a confirmation message as seen in Figure 25. Next, a user can log in to their account using the same username and password. Similarly, a message will appear as seen in Figure 27 and they will be redirected to the “homepage”.

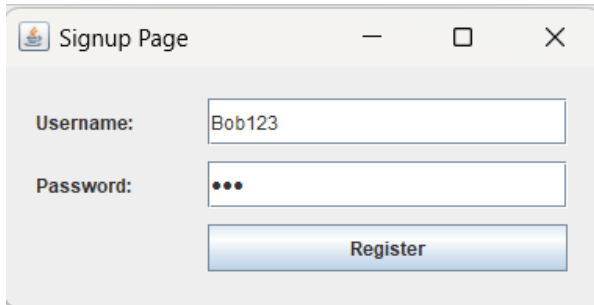


Figure 24: Signup UI

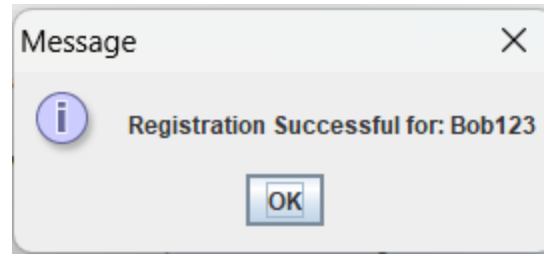


Figure 25: Signup Confirmation

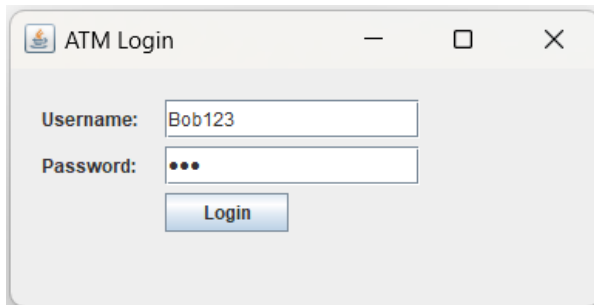


Figure 26: Login UI

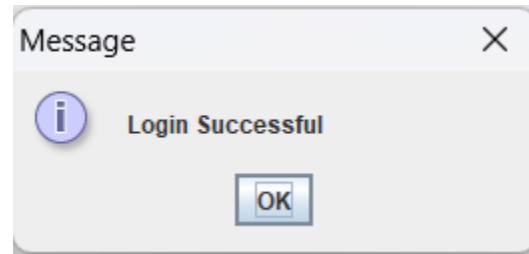


Figure 27: Login Confirmation

When running the server and ATM application during the signup or login, the terminal output provides insight into the authentication and logging process. Firstly, the atm successfully connected to the server after which both parties exchanged encrypted messages such as master secret keys and nonces to ensure a secure connection. As well as the log files are created that store the user's information and the history of transactions.

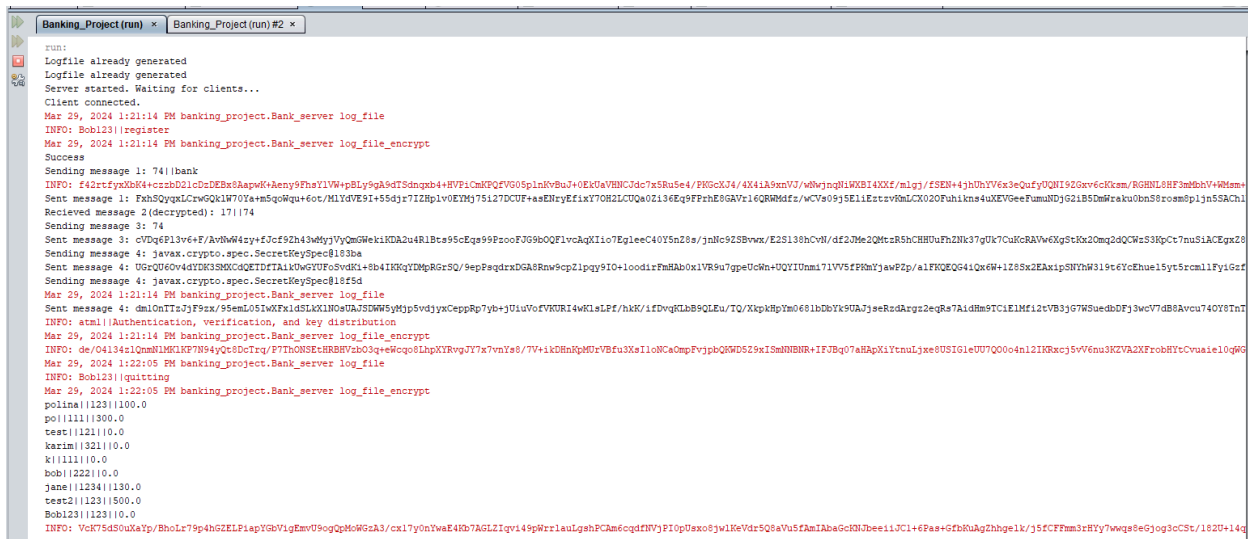


Figure 28: Server Terminal on Signup

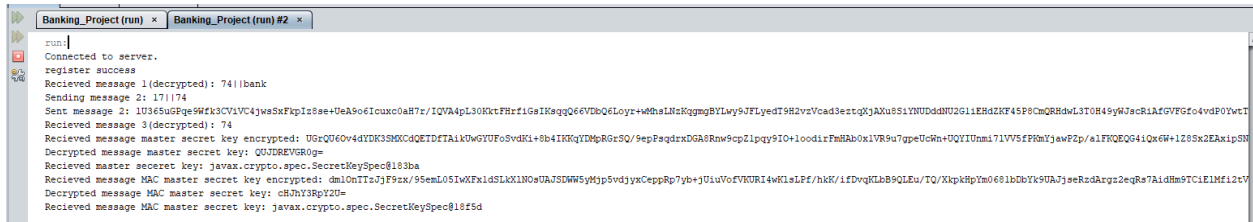


Figure 29: ATM terminal on Signup

After logging in the user is presented with the homepage seen in Figure 30. Here they can choose to deposit or withdraw money and check their balance. Additionally, after every transaction, the user is able to see their current balance in the confirmation message as in Figure 33 and Figure 34.

In Figures 34 to 36, the terminal outputs of the ATM are provided that show how the 3 types of transactions are completed. The ATM sends a message to the server with a nonce, the action (c = check balance, d = deposit, w = withdrawal), and the amount (to withdraw or deposit). Once the server completes the action, it sends a message back to the ATM. The received message contains the original nonce and the updated balance of the user's bank account.

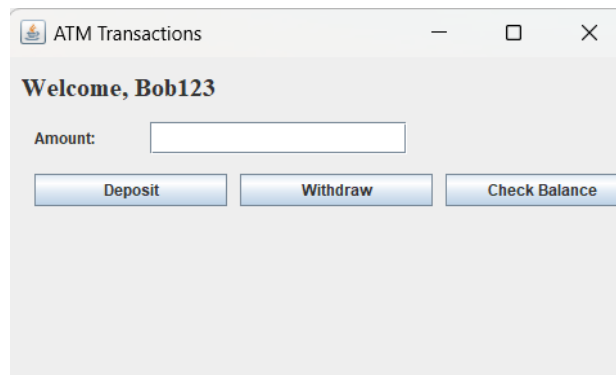


Figure 30: Homepage UI

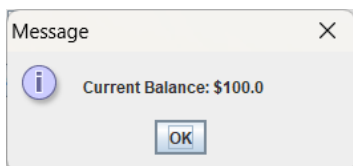


Figure 31: Check Balance

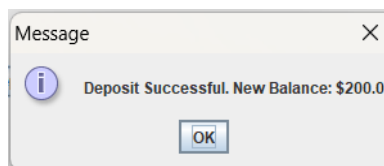


Figure 32: Deposit Money

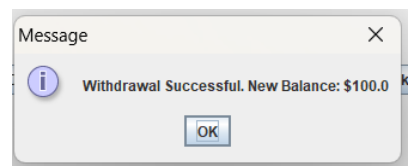


Figure 33: Money Withdrawal

```

Message
72||d||100
Encrypted message
85Wf1InDYM2B59GnBcc63Q==
Received message
kRjMRHTESIYmf53QwseG1w==||AC+0000p 000000+||000000000000
Received message decrypted
72||100.0
Nonce is 72
Your balance is 100.0
Sending message
70||c
Encrypted message and signature
XIt6S+Pv17E=||t60000000000-000000/00000kq60F00
Received message encrypted
vPzEDkQ/b0aMf53QwseG1w==||0000000000000000
0J0
Received message decrypted
70||100.0
Nonce is 70
Your balance is 100.0

```

Figure 34: ATM terminal  
Check Balance

```

Message
46||d||100
Encrypted message
6mhPigXH0H0B59GnBcc63Q==
Received message
ZPvSbZIXIYMMf53QwseG1w==||00000000?000
00@000p0 7D4k%
00Z
Received message decrypted
46||200.0
Nonce is 46
Your balance is 200.0

```

Figure 35: ATM terminal  
Deposit Money

```

Message
35||w||100
Encrypted message
XwdbHY6d/KCB59GnBcc63Q==
Received message
gtVRUMCFV00Mf53QwseG1w==||020xW0b||000000000000>F00m0
Received message decrypted
35||100.0
Nonce is 35
Your balance is 100.0

```

Figure 36: ATM terminal  
Money Withdrawal

When running the server, a GUI pops up which displays the logs of the user's actions. In Figure 37 we can see all the actions that Bob123 took along with the other user.

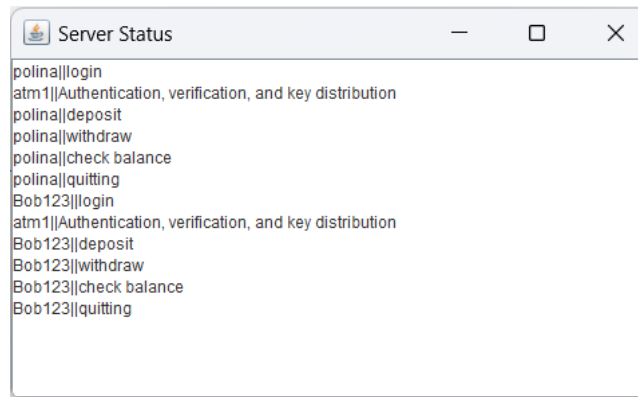


Figure 37: Server GUI

## Conclusion

In conclusion, the area of network security is important for many applications that require the transfer of sensitive information. A banking system is one such application that must utilize different aspects of network security in order to remain secure. The main takeaways from building this application are that multiple steps must be taken in order to ensure the data is secure and there are multiple ways an attacker can attack an application. The leadership experience obtained from doing the project is that the team members must keep a clear line of communication and meet regularly with each other in order to keep track of the project's progress and ensure the project is completed on time. The project was done in a highly collaborative effort as both team members designed every part of the project together and implemented each part together.