Our group decided to do the implementation of the SSL handshake. SSL is more for communication between the client and the server. We also decided later in the project to cut out a few things, one being the certificate due to the self-signed certificates not being recognized to let our encryption through. We also used RSA to generate the keys along with self-created SHA-1(with help from sites) and HMAC functions for the server. A big part of the project was the server certificate, but we were unable to get it working overall.

Our program first starts by doing the SSL Handshake. We first start the program by first activating the bank server first. That tells the bank server that someone is looking for them. We went with a simple Client Hello as we want to tell that person specifically what we're looking for which then leads to the bank server replying.

The Bank server has now gotten confirmation and someone listening to him. He has to reply to the client with appropriate response, the appropriate responses being the decryption method and the TLS protocol, which in our case is the Simplified DES and TLS protocol being SHA-1, which we made ourselves without having to use a library since one of the TLS protocols that was easier to make was SHA-1. We send this information over to the other side which would be the client so that they will now know what to use for decryption and encrypting future messages.

The Client now has what they want, which would be the TLS protocol and the future Decryption/Encryption method. They just have received all the information that they need and now put it into good use. Client first generates their RSA keys by utilizing the Euler totient function which then creates both a private key and public key. The public key of the Client is sent over to the server which then combines the parts of the key to form the bigger key. This bigger key is then kept while the server send back a shared key for them to decrypt all future messages for Simplified DES. This will lead into all future decryptions and encryptions between both parties.

Since the SSL handshake has happened, both parties now have access to each other systems. The first pf this is to try and deposit money into the bank. They first input their account number and their amount. This starts the encryption process of the message. We first have the message itself for example being “D 123456789 1234”. D is the action that the person is taking, 123456789 is the account number, and 1234 is the amount that is being deposited. We first start by encrypting the entire message along with the Hash-based Message Authentication Code, separated by a ?. This is because we want to authenticate that the message we have sent to them is ours instead of anyone elses. The HMAC can only be made by having both the response and the shared secret code, which the bank server should have. By encrypting the message entirely, we also have to change the message a bit. S_DES leaves behind a whole bunch of lists in binary which would be hard to send so we join them together to make something that looks like a list of Ips but not really. Finally, we encode the message and send it over to the bank server. The Bank server is apply receive this message intact. If there is any corruption in the message, there is a if statement checking if there were any modifications made to the HMAC and response. So we separate the message into its two components, being the request itself and the Hash, which the Bank then uses to authenticate the message. If it works, the bank sorts out the request and gives the according message back. It also does the same encryption as the client as they have a shared secret between them. The Bank encrypts the message along with their hash attached to it using the Simple DES and then sends the message back the Client. The Client then receives the encrypted message with the hash and starts the decryption process over there. It decodes the entire messages, strips it of any stray whitespace, and then starts the decryption using S-DES. They also authenticate the hash on their side and if it works, the response is shown back to the user and the program continues. If it doesn't, then it says the message was corrupted. That is how the implementation works so far.

Now I will get further into the implementation itself. At the beginning, we decided to make two versions of the file. We first went with a bank server and client atm as they were the two entities in the bank scenario. We managed to first create the ports and they listen to each other, allowing them both to accept information from one another. They just needed to listen and that was simply the easy part. Next we had to initiate the SSL handshake. We set up sockets and moved on to sending via sock the first message, being the Client Hello, which would be the encryption and TLS protocol that they would both be doing. I found it odd that for the client wanting to connect to Bank, the bank should set up the security protocols, but if the client wants to be able to connect at all, it must first specify what kind of encryptions/ TLS protocols that it can do. The Bank server is implemented to then take in the information and then sends back to Client a confirmation of the protocols being used to communicate with each other , being server_hello = f"SDES-SHA1". This leads to the server sending back the encryption and TLS protocols confirmation that both parties will be using. I would also have liked the send the certificates, but self-signed certificates were not being allowed to be sent over and they also were not being verified correctly. I would have been able to simply not verify the certificate at all, but that would defeat the purpose of the server certificate, so I decided to simply just remove that code. So the code would send all of this back to the client, which would then receive everything. The client would be able to encrypt messages using the confirmation that they are able to use it from the Bank server and also they can verify from the certificate that this person is actually the bank and not someone else. This leads to the client creating their own keys and sends their public key over to the bank server. We would have liked to encrypt these keys with the public key from the bank server, but we weren’t able to. This eventually led to sending the public key over to the bank server. The Bank server then create the shared secret that will be used between the bank server and the client. This key was made from 40 random bits which is then tranformed into bytes and the bytes are sent toward the Client. The client only need to grab the first 10 bits as the shared key for the simple DES while the bank server simply has the shared key themselves. The Bank also encrypted the shared secret key so that the client can then decode with their key and then have the secret key themselves, after taking only the first ten bits. Now that the SSL handshake is complete albeit a bit insecure since no certificate, the bank and the client can now do their service toward each other.

Now is the implementation of the Bank and Client interactions. We will first start with the deposit itself. There are no security yet as the client is simply making a request to the bank. We first enter in the Bank account number and the number that we want to deposit into said account. This leads to the first message being sent. Since we have a message being sent, we go through all the necessary procedures. We first have to attach the HMAC to the message as well and we chose the delimiter that separate each other with a “?” We start the simple DES encryption. We first generate the subkeys using the shared key that both the bank and client have. The next step is to format the text so that it would be easier to encrypt. Finally, we encrypt the text into something that can’t be decrypted hopefully by other threat actors, although in this day and age, it is most likely being cracked. The encrypted message is then encoded and then sent through the socket. After that, the ATM client is stuck waiting for the next message to appear from the bank.

The Bank now must reply appropriately. The bank has received the message from the client, but isn’t sure if the message really is something that came from the client. So the bank first decode and decrypts the message using the S-DES algorithm they have along with the shared secret that they have as well. They also feed the request and the shared secret key into HMAC formula, which will give back the HMAC that they can use to check with the message’s HMAC. IF the HMACs do not match, then it means the message is either not the same or it has been sent by someone else, most likely malicious. The HMAC is important in checking if the message’s security has been breached or not. So if the HMACs are not correct, there is a message that says “message corrupted, try again”, preventing any attempts in corrupting the HMAC messages. If the HMAC are the same, then it’s okay to follow through with the request. This would be a hard part of the implementation as when you get the HMAC from the ssl file, it would come out as Hex as opposed to the message which brough out in string, so we had to turn the HMAC into compare which would then become string. For the Bank server, we made it really simple. The first letter of the message “D 123456789 1234” is a sign of what kind of quest is being made. This enters the If and elif statements that are below the security. By choosing any of them, you choose a response that is sent back to the client. After the response has been made, it is time to send back to the client. The response is attached to the end a HMAC which the client can confirm who the message is from. Then the message is encrypted the same way that the client has encrypted their message, using Simple DES and their own shared secret, which should be the same as the client and thus allow the client to be able to decrypt their message. Finally, the message is encoded and sent to the client.

The client has been listening all this time for a response from the Bank server and is unable to do anything else while waiting. When the response finally arrived from the bank server, the client begins to decode and decrypt the message. After the decryption process happens, the client checks the HMAC on the message with the HMAC that they have, which they use the shared secret along with the message itself to feed into the HMAC algorithm, which uses SHA-1 to check if the HMAC is working or not. If the HMACs are the same, the client plays the message that has been sent to them. Thus the program ends and if the message has the wrong HMAC, it just says that the response has been corrupted.

The implementation has some simple if statements on the bank side to check if the person is able to withdraw money from the bank. We simply asked for their pin number which would be included in the message that is sent by the client.

Overall, I think that we could have done better with our implementation since we weren’t very good at the beginning part with the encryption. Had we more time, I think that we could have done better in making a more secure SSL handshake with an actual certificate.