1 Handshake Protocol

Done by Ryan Prashad

The handshake protocol implemented in this assignment is derived from hrefwithprime and uses Diffie-Hellman key exchange in order to obtain a common shared AES and HMAC key in both the Bank and the ATM.

1.1 Client Hello

This first step of the SSL protocol provides the server with a list of applicable ATM cipher schemes that the Bank can then read and interface with. The client random is a series of cryptographically secure random bytes generated at the start of each session in the client to be used in later steps. The client hello is critical to establishing a common public key asymmetric scheme so data can be signed by the authority of the bank and verified.

1.2 Server Hello

Step two is the reply to the client hello by the server. For this step, we establish a common public key scheme with the client (ATM) and send this over. We as the server also generate a series of cryptographically secure random bytes in addition to having received the client random bytes from the client hello also. For our implementation, we do not have an SSL certificate that could be sent to the ATM from a third party. To get around this, we read the Bank public key into the ATM class instance after the Bank sends to the ATM the common scheme. This way, we simulate the ATM getting the Bank public key from that SSL certificate and all is well.

1.3 Diffie-Hellman Exchange

The Diffie-Hellman exchange is a method of key exchange used in this version of the SSL protocol in order for us to establish secure keys that can be used for symmetric encryption(AES) and mac(HMAC) post-handshake.

1.4 Diffie-Hellman Basics

Diffie-Hellman key exchange in this specific scenario relies on a cryptographically secure random integer generation below the 1536 bit prime supplied from hrefwithprime and uses a generator of 2. Assuming Bank generates integer A this way and ATM generates integer B, we have the Bank public number as 2^A and the ATM public number as 2^B. To get a shared secret from these numbers, we need to raise each to the other's power. This leaves us with the Bank having $(2^A)^B$ and the ATM having $(2^B)^A$ which by the commutative property of multiplication are the same shared number in each client.

1.5 Diffie-Hellman applied to AES/HMAC

To apply this method towards what we need, each Diffie-Hellman transfer for us will actually consist of transferring 2 parametersm, one for AES and one for HMAC. Both keys are of bitlength 256 so we can use this prime from hrefwithprime to generate securely for both of them. After sharing both secrets we can do (secrets % prime) and assign it to secrets in each AES/MAC case to start extracting the key (because $(2^A)^B$ is a huge number) and we can finally start to figure out actual key values. After this modulo, we need to preform another one by 2^256 in order to reframe the value in refrence to our 256-bit keys that we need. Once this is done for both AES/MAC, we have essentially completed the key exchange.

1.6 Forward Secrecy

The beauty of this method is that it provides forward secrecy to protect future sessions from attack based on old data. As an example, if an attacker were to save traffic from one of our ATM/Bank transfers and crack the encryption, then their data would only be useful for that one section. This is due in part to the fact that we have server/client random byte sequences, but also thanks to Diffie-Hellman. Back in the basics section, we discussed integers A and B that are obtained by a random cryptograhpically secure number generator with bounds on the chosen prime. This allows us to have nearly the same security each session while changing the secret Diffie-Hellman parameters used each session. Thus, an attacker cannot try and encrypt a public Diffie-Hellman parameter from a current session with a private Diffie-Hellman parameter from a prevoius session, providing us forward secrecy.

1.7 Server Signature

This step allows the ATM to verify the Diffie-Hellman plain text integers with the public key from the bank. The server pads it's generated public Diffie-Hellman paramenter with the client/server randoms. This plaintext message is sent over the network and is recieved in plaintext by the ATM. The server also sends over the network a signed version of this message using the common encryption scheme agreed upon earlier in the handshake with it's own private key. These series of messages ensure that replay attacks are not viable because of the random byte padding, and also secure the plain text without direct encryption because of the public key signature used.

1.8 Server Verification

After recieving the plaintext and the signed message from the server the ATM can now try to confirm the Bank's validity. The signed message is evaluated using the Bank's public key that the ATM already has from previous steps. If the signature holds valid and no tampering has occured, the ATM can then

proceed. This means that the ATM's public Diffie-Hellman parameter is now sent over the network in plaintext to the server in our case.

1.9 AES/HMAC Key generation

Now that the ATM has the public Bank Diffie-Hellman parameter and the Bank has the ATM public Diffie-Hellman parameter, both parties can use their secret randomly generated integers to create the shared secrets for AES/HMAC. This is then formatted down to a usable 256bit hex pattern for use with AES and HMAC. Both pairs of keys for both parties should now be the same and secure if the Diffie-Hellman exchange was done properly, and as a last final step we can use aes and an HMAC on a 'finished!' string in order to finalize the handshake (each party sends/recieves one). This message concludes the handsake and symmetric encryption has been achieved.