CIS551- Project 3 Report

Group Members

Adwoa Osei-Pianim (adwoa) – Responsible for documentation and testing the application.

Brian Breck (bbreck) – Responsible for the transaction protocol and the audit log. Brian spent 16 hours on this project.

Vivien Durey (vdurey) – Responsible for the authentication protocol.

Authentication Protocol

This protocol is composed of 5 steps. In order to implement it, we created the class AuthenticationMessage, which is described in the README file. In the following, the word "bank" actually designates the bank's server.

- First step: Identity of the client and ID of the ATM

The ATM sends the first message containing its ID, the client's account number, as well as a nonce and timestamp. In order to keep the account number secret; we decided to encrypt it using the bank's public key. This way, only the bank is able to decrypt it using its private key.

To keep things simple, we decided not to encrypt the ATM's ID, because we choose to think that this ID would not give any information to any attacker (such as the ATM's location, for example). However, if it were the case that this ID could lead an attacker to the ATM, it would be necessary to encrypt it with the bank's public key.

- Second step: Bank sending challenge

The bank gets the message, decrypts the account ID using the bank private key, and finds the client in its database. If the client is found, the account is loaded in memory (with the corresponding public key). The timestamp is verified and the nonce sent by the ATM is added to the challenge message to help the client verify the authenticity of the message.

Then, in order to verify the client identity, the bank sends a message (instance of AuthenticationMessage) containing a challenge string that must be correctly encrypted by the ATM machine. It will also contain a nonce and timestamp generated by the bank server that will be used by the ATM to help authenticate this

message.

It is important not to sign this message. The key reason here is that the bank did not start the authentication transaction, the ATM did. If the bank had signed this message, then, an attacker could obtain a lot of signed messages from the bank, only by knowing an account number and the bank's public key and doing the first step over and over again.

- Third step: Challenge response

Having received the challenge, the ATM creates a new message containing the encrypted challenge using the bank's public key, a new nonce (atmNonce field) and a timestamp. The message is signed using the client's private key and sent to the bank. Once again, this message cannot be replayed or altered without being detected by the bank.

- Fourth step: Establishment of the shared AES key

The bank receives the challenge response and verifies it: first the nonce (bankNonce) and the timestamp, then the challenge response, and finally the signature using the client's public key.

If everything is fine, the client is authenticated, and the bank creates a new message with the received atmNonce, a new bankNonce and a new timestamp. The bank generates a new AES key, which will be the shared key used for the rest of the transaction, encrypts it with the client's public key and adds it to the message. Finally, the bank signs the message and sends it to the ATM. By sending this signed message with the nonce sent by the ATM, the bank gives proof of its identity.

- Fifth step: End of the authentication, beginning of the transaction.

The ATM receives the final message and verifies the timestamp, the nonce and the signature. If everything is correct, the bank is authenticated, and the ATM can now decrypt the AES key using the client's private key and keep it for the transaction. The authentication is over.

Transaction Protocol

There are several protections in place to ensure each request sent from the ATM to the Bank Server is secure. Each TransactionMessage instance contains a timestamp and the bank nonce sent in the previous bank server message to prevent replay attacks and limit other man in the middle attacks. Additionally, each message is wrapped in a SignedMessage instance using the user's private key to verify the

source of the message. Finally, each payload is encrypted using AES with the session key established in authentication. The AES encryption prevents those listening on the wire from reading the details of the message. Upon receipt of the message by the bank, TransactionMessage object is decrypted, the timestamp is checked to make sure that the message was received in the allotted time (in this case 30 seconds), the bank nonce matches what the Bank Server had recorded, and the signed message is verified.

Each bank response uses similar protection to create secure traffic from the Bank Server to the ATM. Each TransactionMessage instance contains a timestamp and the ATM nonce sent in the previous ATM message, and each message is wrapped in a SignedMessage instance using the bank's private key. The Bank Server encrypts the TransactionMessage using AES with the session key established in authentication. Upon receipt of the message by the ATM, the TransactionMessage object is decrypted, the timestamp is checked to make sure that the message was received in the allotted time, the ATM nonce matches what the ATM had recorded, and the signed message is verified. Even though one class was used for communication between the server and the ATM, we were still able to create asymmetric messages because the ATM message uses the action field, and the Bank Server uses the successful field.

We also made the decision to synchronize all transactions on the Bank Server in order to ensure that the account balances are not being concurrently modified. This could be the case if two separate cards are linked to the same account. We also synchronized on the ID rather than the method itself so that non-conflicting transactions could occur simultaneously.

Log encrypting and auditing

Based on the frequency and magnitude of the logs, a symmetric cryptography mechanism was chosen to reduce the cost of encrypting the content. The key needed to be available after the Server shutdown in order to decrypt the log file, so we made the decision to persist it to a file.

At first we assumed that the key would be written to a file in the secure RAM in the same location as the private key. However, based on the problems we found with writing to the ObjectOutputStream with different streams, we found that we could not append to the log after the server shutdown. Instead, the log file would have to be moved elsewhere, or else overwritten, in order for the new ObjectOutputStream instance to write cleanly to the file. So to protect the key so that it can safely be kept with the log, we decided to use the bank's public key to encrypt the AES log key before writing it to a file. Then when we load the key, we first decrypt use the bank's private key in order to retrieve the AES key.

We decided to log each step of the authentication process using the AuthenticationLogMessage. It contains a timestamp, the ATM being authenticated, and the step of the process. We included the ATM with each step so that if multiple ATMs are authenticating simultaneously, we can keep track of which ATM is being authenticated with each message. When the authentication is finished, a special version of the AuthenticationLogMessage is logged which additionally contains the Account ID and Session Key that can be used to decrypt the SignedMessages described below.

In order for a third party to verify the authenticity of the messages sent by the ATM machine, we decided to log the signed messages sent by the ATM that contained the TransactionMessage object. This allows us to log the signature that can later be verified by the public key stored in the account database. Furthermore, in order to determine the contents of the transaction message, we can use the session key that was saved in the last step of authentication in order to decrypt the contents and read the result. Finally, we log the result of the transaction request with a date, the source ATM, the account ID, the action, and whether it was successful.

That way we are storing proof of authentication, proof that the transaction request is authentic, the actual request made by the user, and the result of the transaction. All of this information will provide a third-party with enough information to verify a transaction.

Reading the log file

Though we store the session key in the above steps, when printing the log we generally just want to see the results of what happened. Therefore a main method was added to the Log class that just iterates over the objects in the log file, decrypting the objects with the AES key stored in the log key file, and prints the results. If we needed to write code that re-authenticated the signed messages, decrypted the transaction messages, and printed the transaction requests, the information is in the log to do so.