Cryptography and Network Security I

Summer 2020

Lauren TeBeau, Alex Pinkowski, JJ Tareco

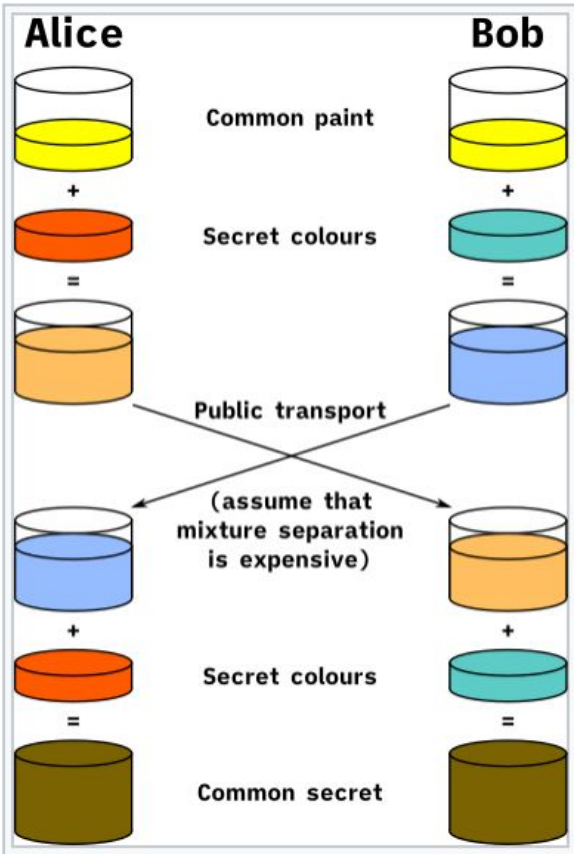
Documentation of White-hat

For our project our group has implemented a version of SSL protocol supporting our own implementation of Blum- Goldwasser, DES, and 3DES and MAC signatures using a SHA1 hash. The project was implemented as a client-server protocol to mimic transactions/ebanking between an ATM (client) and the bank (server). It enables banking operations to deposit, withdraw money, and check balance via ATM by accessing the bank. Our code does the following: negotiate and establish a secure channel using the SSL handshake protocol, and pass back and forth messages to implement the banking operations above. Here I am going to discuss how we did this and the logic behind our security.

Key Negotiation

In order to simulate communication between the client and the server we used sockets programming libraries. Thus, we firstly established a socket connection between two terminals to represent the client and the server. In order to set up socket communication between two terminals we used the TCP protocol. On the server side, we bound the socket to localhost and to the port 1234. On the client side, we connected the socket to the same port 1234. After this, any message sent over the sockets can be read by the other user connected to the same port. This allows the client to send messages to the server, and for the server to reply.

In order to establish a secure communication between these ports we then enacted a Diffie- Hellman key exchange. To do this we established random private keys for the bank and client and each encrypting a seed message to send to each other. Once encrypted these become public keys that each side sends over through the port. When the given public keys are encoded again by each side we obtain a secure private key that is the same for both client and server. This key is used both for encryption and for the creation of the messages' MAC. The diagram below helps to demonstrate this process.



The different colors help to demonstrate how each side starts with a common message to encode, then through manipulation obtain a key that is negotiated and secure from both sides.

This acts as a handshake protocol, the common seed acts as an authenticator alongside the input pin that the user enters, as if the bank enters a different seed than the client or vice versa then there will be a different final key on each side and they will not be able to decrypt each other's messages and if an incorrect pin is inserted the program will not allow communication at all. The sharing of public keys to gain a final private key negotiates the final key in a way that keeps their private values secret and key never expressly communicated, thus securing it from attacks. In our implementation of this protocol, the server (Bank) generates two random primes. It then sends those primes over to the client (ATM). The client generates its own private key and generates a public key using the primes that the server sent over. The server does the same steps, generates a private key to create the public one. After the exchange of public keys, each using their own private key can generate a shared secret key that only the client and the server know. Now that we have authentication and a negotiated key, we move on to passing encoded messages between client and server. The type of encryption that will be used is also negotiated at this time. The client will send the server a list of potential cryptoshemes to use and the server will randomly choose one for the communications and send the name back to the client. We will have the options of Blum-Goldwasser, DES, and 3DES, all implementations of which will be discussed.

Blum-Goldwasser Implementation

We initially chose Blum-Goldwasser encryptions to accomplish secure encryption because it is semantically secure. Blum-Goldwasser is an asymmetric key encryption algorithm that encrypts blocks of messages using a public key that can then be decrypted using the private key. One side generates large primes p and q. $N = pq$ is the public key that will be used. The

private keys must also be generated and this is done by finding a and b such that pa + qb = 1. In our code, we use the extended euclidean algorithm to determine a and b. Having the private key (a, b, p, q), allows us to decrypt the message. For the encryption part of the Blum-Goldwasser, we use the public key n and find the block size of each message by taking the log base two twice. After this, we generate a random quadratic residue. For each block of bits, we XOR this block with the least significant bits of the quadratic residue. After this we generate the next quadratic residue and take the next block size until we have no more blocks to XOR. After this, the last quadratic residue is saved and added to the end of the chain of encrypted blocks, for it is used later for decryption.

Decryption with Blum-Goldwasser is achieved using the public keys (a, b, p, q). After receiving the ciphertext, record the quadratic residue attached to the end of the message. This quadratic residue is used in order to determine the quadratic residue that the user used during encryption. To find the original QR, there are a series of heavy computational equations that require p, q, a, b. Once the QR is found, the process for decrypting is very similar to the process of encrypting. First you determine the block size, and take blocks out of the ciphertext. Then for each block, you XOR the block with the least significant bits of the QR. You then generate the next QR and repeat the process on each block until you get the end result, which is the message.

DES Implementation

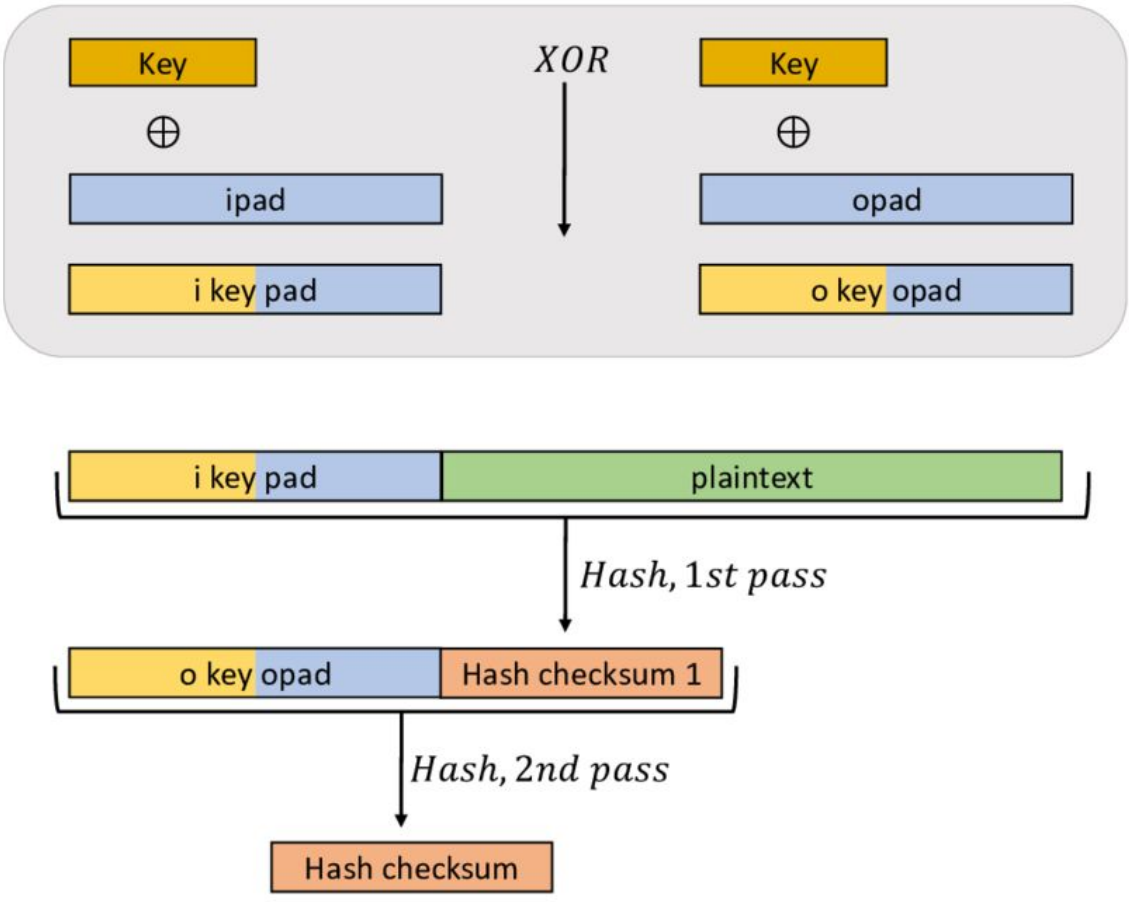
Our cipher suite includes a fully fleshed out 64-bit implementation of DES as well. For this implementation, we decided to use binary strings to represent the key value and the message block, since it is much easier to debug and to understand the operations as we implement the code. We started by hard coding in all of the important values (i.e. the permutation tables,

S-Boxes, etc.) and continued onto the algorithm, doing all of the operations on binary strings. Our DES algorithm splits up the input block into 64-bit chunks of text, performs the encryption algorithm on each block of text, and then adds it to the ciphertext. If we ever encounter an odd number of bits (i.e. the block isn't a multiple of 64 bits long), which will obviously happen often in the real world, we provide padding for the message using the PKCS#5 method. Instead of padding the message using 0, since that could easily decrease the security of the encryption, we repeatedly append the amount of bytes left to reach 64 bits to the end of the block. For example, if the last chunk of the message only had 3 bytes, we would append the block 0x05 5 times to that block before encryption, to ensure that the encryption will run smoothly and securely. We had originally planned on using the simplified DES system that we had already implemented, but instead decided to upgrade it to a full-sized version of DES to increase the security of our client. A 64-bit key doesn't provide enough security to be considered a secure mode of transmission in the real world, but it is a significant upgrade from the S-DES that we implemented for homework 1.

HMAC Implementation

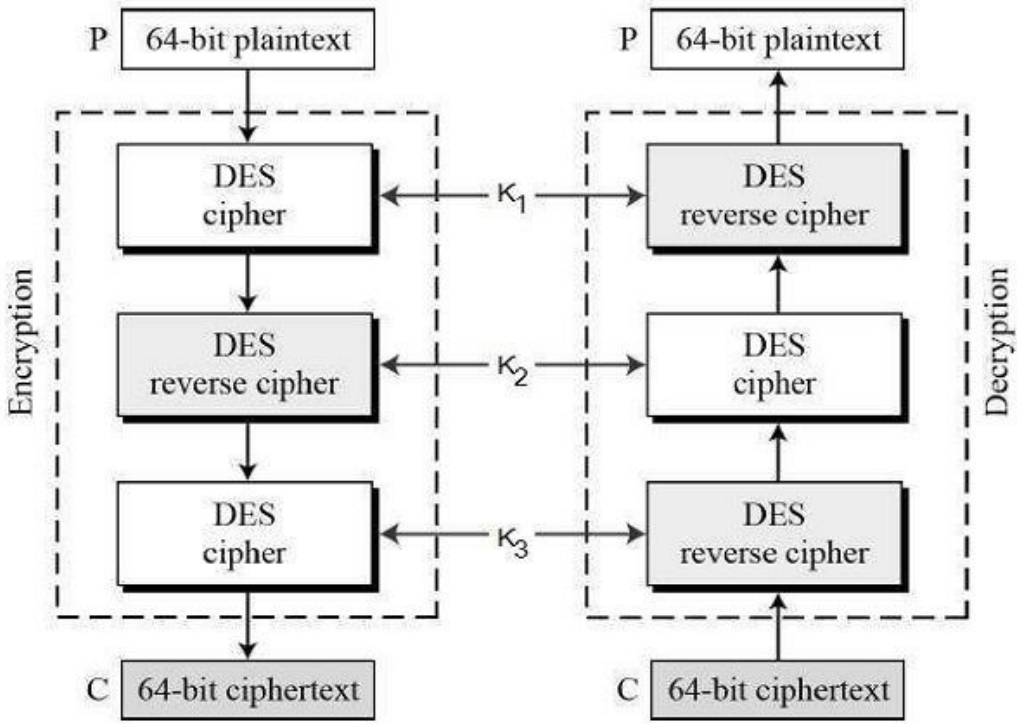
The MAC we implemented uses a SHA-1 hash (just like everyone else) since it is a very efficient and effective hash function. The client and server use their shared private key to generate an HMAC for the message to be sent between them, and then appends it to the end of the message before encryption. The message is then encrypted and sent over the secure channel. Since the client and the server share the same implementation, and create the message in the same way, it is easy to parse the message and find the HMAC that the client generated. Once

receiving the message, the server generates its own HMAC signature and checks to make sure they match. If an incorrect HMAC is provided, the system aborts, since the client would then be an intruder using the MAC function incorrectly. For the implementation of the SHA-1 hash function, we used a full size hash function that produces a 160-bit signature. This is very useful for authentication of the user, since it uses the client and server's shared private key. It would be infeasible for an intruder to generate a false mac verification, since our shared private key is securely generated. The process for our SHA-1 HMAC function is detailed in the diagram below:



3DES Implementation

Once we finished the implementation of single DES, we decided that we could easily and substantially improve security by also adding the triple DES cipher to our suite. Once again, this 3DES algorithm doesn't provide as much security as other, newer algorithms like AES, but it is far more secure than regular DES, and is a very strong upgrade to our system. Our implementation of 3DES creates 3 independently and randomly generated 64-bit keys, using the shared secret key as the seed for the random number generator. This way, the keys will be unique and independent, but the client and server will still always be able to generate the same key set if they share the same secret key. Once they are compressed to 56-bits, the cipher has an effective key length of 168 bits. This should be much more difficult to attack, since the keys are randomly generated using the client and server's shared private key, so it would be very difficult for any intruder to brute force attack our server. In this implementation, just like our DES algorithm, we perform encryption 64-bits at a time.



Communication Between Server and Client

In our implementation of the communication between the server and the client, we implemented a number of ways for maintaining secure communication. Once the client connects to the server, they do the Diffie-Hellman-Key-Exchange, as explained earlier, to determine a secret shared key. After this, there is an exchange of public keys in order to use Blum-Goldwasser. The client then sends over the cryptosheme that will be used for the sessions. The choices are DES, Blum-Goldwasser, and 3DES. The client determines the message that they wish to send over to the bank, then attaches a timestamp to the end of it. The timestamp ensures that no replay attack can be done. The way the timestamp is implemented in our project, is by using the time library in python. The time given is the number of seconds passed since epoch. We use this time and after adding it to the end of the message we encrypt it using one of the

algorithms we wrote (Blum-Goldwasser, DES, 3DES). Then we generate a HMAC using the same message, and using the shared key that was generated during Diffie-Hellman, we attach that to the end of the ciphertext. We send this over to the socket, and the other user, either server or client receives the ciphertext. When decrypting, the user first decrypts the message using the agreed upon cryptosheme, and generates a HMAC using the message. The receiver must then check the timestamp in the message, and check the HMAC received. The HMACs must match exactly, otherwise we know that the message was tampered with, and the timestamp must also be the same. In our implementation, we make sure that the timestamp is at most one second off, to allow for time to encrypt, decrypt, and send the cipher over. This double implementation of timestamp and mac, prevents a number of attacks and increases the security. The reason that this is secure is because if an attacker decided to do a replay attack, the HMACs would match, but the timestamp would be off. Another attack where the attacker changes bits, the HMACs will no longer match and the time might be modified as well

What else is there that makes our project that much more secure for our users? After all communication is over the socket connection is terminated so that in order to establish a new communication new public, private, and final keys are used. Thus if any attacker was to attempt to look at old communications to break the new encryptions they would fail due to the constant changing of keys. Unlike common simulated encryptions like we have used in class or homeworks, the key changes with every thread of communication and is pseudo randomly generated making each transaction independently secure.

ATM and Bank Implementation

For the ATM implementation, the client asks the user for a name, which represents inserting the card, and for the 4 digit pin. We made a dictionary where there are 3 users with their respective pins. We assume that this is private information that should not be used for breaking our system. This is simply to simulate signing into an ATM. For our implementation of the bank, the server acts as the bank. We created a bank object that performs three operations. You can deposit, withdraw and check your balance. The information that the banks has is a file called *private_bank_info.txt* which we assume that only the bank can assess (ie. not part of the security, it is completely private). This file holds the name and the amount that that person has in the bank. Depending on what the client wants, the bank makes the change to the balance and saves the file. This does not involve any cryptography, but we wanted to simulate a bank as best as possible.

Conclusion

In all we have followed the guidelines given to create a secure implementation of SSL to send messages between a client and server with various measures in place to ensure message secrecy. This can be used by an ATM communicating with a bank to fulfill different transactions safely and securely without interference. The encryption methods, changing keys, MAC, and timestamp all make moves to secure our communication against the attacks we discussed in class. With the correct username and pin number, any ‘bank member’ should be able to make transactions at our ATM worry free.