Cryptography Project - White Hat Portion

For this group project, our group was tasked with creating a banking system between a server and a client, with the server process representing a bank and the client process representing an ATM. The main goals of the implementation were: be able to pass messages between a bank and an ATM, and to be able to do so securely. To accomplish these goals, we divided the process into three segments: setup, handshake, and relevant banking operations.

Below we will explain the processes we used for each step, how those processes work, why they are there, and what could go wrong if they were absent. We will also briefly discuss why certain technical decisions were *not* made at the end, and of course how, by combining all these processes, we were able to create a secure channel in order to pass banking operations back and forth between the client and server.

To begin, the implementation needed a setup phase: keys needed to be generated, and a basic server and client setup would be needed as the foundation for the banking operations. We used a TCP server hosted locally, as TCP is connection-oriented which we felt would be more secure when trying to communicate with only one ATM. Additionally, we hosted locally as it would be the easiest way for the black-hat team to be able to run the server anytime they want, instead of having to rely on our group hosting the server or re-hosting it in the event of a crash outside of normal hours.

Next in the setup segment, we wanted to use RSA to generate public and private keypairs, as well as the secure symmetric key for banking transactions, before the server or client ran. This was because generating keys during the run of a process

could impose vulnerabilities, such as a bogus client or the server generating fake keys for the client process to use, crippling our security. Instead, we had a config.py file which generated these keys and then situated them into the appropriate serverpublickey, server private key, client publickey, and client private key text files.

To create the asymmetric RSA keys, we needed to find 4 large primes (consisting of 2 p values and 2 q values, one for each keypair belonging to the client and server processes). To do this, we ran a loop to generate a large (768) bit number, then ran a low-level prime test to determine if it could be prime. Numbers that passed the low level prime test were subjected to the miller-rabin algorithm, as it was able to determine likely-primality with a success rate of 75%. We ran this algorithm 15 times on the number, for a total likelihood of $9.17 * 10^{-9}$, or an incredibly low likelihood. In doing this, we could be reasonably sure that the large numbers we picked were prime or pseudoprime.

Next, those numbers were multiplied together to get a large N value, and the totient function ((p-1) * (q-1)) of pq was also calculated in order to find the phi of N, the modulus used to find the private decryption key. The encryption and decryption keys were calculated by finding a number e which was greater than one but less than phi N, in which the greatest common denominator of those two numbers was 1, and then its modular inverse (d). Those numbers were all saved to their appropriate files, and then the setup process exited. The config.py file takes about 1-2 seconds to complete the generation of both sets of keys. Finally, the secret symmetric key to be used later was generated by the secrets library, and is just a 192 bit key to be used in DES.

The config.py file was crucial for the setup segment, as without it, our processes would not be able to perform RSA in order to share the symmetric key. It was similarly crucial that we chose large numbers with a high likelihood of being prime, in addition to the fact that those two numbers would need to be close in length to one another. This was because we wanted to make the factoring of these numbers as difficult as possible seeing as the black-hat team would only have one week in order to successfully deploy their attack, and a brute force attack on RSA would be infeasible with prime numbers of such magnitude. This is because RSA is a one-way function, in that it is very difficult to recover p and q from e and N. With the conclusion of config.py, the setup was complete, and we were ready to begin a secure handshake.

After RSA was complete, we implemented a way for the server and the client to sign each message with their private keys. To do this, the client first encrypted the message they wished to send with their own private key. This used the RSA decryption function with the private key and the plaintext as the input. Once this was done the result was then encrypted with the public key of the server and the message was sent. The server would then decrypt that message with its own private key. The resulting text would then be decrypted again with the public key of the client. This process allowed for the server to be certain that only the proper client was sending it a message as only that client would have access to its private key. This process added an extra layer of security to the authentication step as any receiver of an encrypted message could verify who sent it.

We also implemented RSA signature generation. Using the hashing function SHA-1 to generate an HMAC for any outgoing messages, we implemented a way for

anyone to create and verify signatures. The creation of any of these signatures began with the choice of constants for the ipad and opad values that all HMACs need to utilize in their generation. The importance of these two values lies in the fact that their specific differences determine the computational independence of two subkeys that must be used in the generation of any HMAC. As such, these two values must be chosen in a way that their related Hamming distance is maximized. That is to say, their Hamming distance, which is a measure of the number of bits that differ between the two n-bit binary strings of opad and ipad, must be maximized.

As it stands, there is a default set of two values, respectively repeated bytewise as needed that serve as a standard for the values of opad and ipad. These values, being 36 and the hexadecimal 5c, were chosen as the repeated bytewise portions for any HMAC's values of opad and ipad because of their optimal Hamming distance when repeated. Using the opad and ipad, one must evaluate the bitwise XORs of their respective values with a selected private key to determine their resulting subkeys. A SHA-1 hash is then conducted on the ipad's resulting subkey after the message of interest has been appended onto its end. The result of that SHA-1 hash is then appended upon the end of the opad's previously evaluated subkey. A final SHA-1 hash of that newly created value serves as the HMAC of the message. Afterwards, the resulting HMAC, or signature, of the message could then be verified by the hash of the message and the decryption of the signature with the public key.

Once setup was complete, and the server and client were able to talk to one another, the next step was ensuring that the client and server could share a secret symmetric key. The process proceeded as follows: the server would start up, open a

port, and begin listening on said port for a client. The client program would then start up as well, and connect to this port. Once a connection was established, both parties read in their respective private/public keypairs, and began the handshake.

Each message in the handshake was encrypted twice with RSA: once with the sender's private key, and once with the receiver's public key. This was done for two reasons. Firstly, we wanted to prove to the other party that the message was coming from the intended sender. Secondly, In RSA schemes, it is possible to get the public key from the private key, but it is *hard* (as discussed above) to get the private key from the public key. Therefore, the recipient could be reasonably sure that if a message was encrypted with the private key of the sender, the sender was who they said they were. Additionally, the message was encoded with the public key of the recipient for the same reason. The sender needed to be sure that only the receiver could decrypt their message, and subsequently send back relevant information. Therefore, this RSA encryption on handshake messages was a great, and integral way of providing signatures on our messages, and could induce vulnerabilities such as impostors or men in the middle if it was missing.

As for the actual content sent during the handshake, the process went as follows (remember, these messages are double RSA encrypted): First, security capabilities were calibrated by the client/server by comparing the possible securities each can use, and deciding on the best. As this is just a derivative implementation, these processes only support one set of securities each, and they are the same, so there isn’t much negotiation here. If this were a larger implementation we could have negotiations, but we felt it wasn’t necessary or more secure here.

Next, the client would send a nonce value (random large integer), for confirmation that the sender isn't just mimicking messages that are hard-coded as part of the handshake. With this nonce method, we could confirm that the sender/receiver erre actually understanding sent messages, and adjust them (decrypt/encrypt) in a way that proved to the other party they are who they say they are. Finally, the server uses the symmetric key generated during the config and sends it to the client, using the same RSA method. The client echoes this key back, to tell the server it has received and understood it, and then the server and client exit the handshake.

The handshake is calibrated so that every step, while exchanging information, the server and client are authenticating each other to ensure neither party is pretending to be someone else. They share a symmetric key, and negotiate securities, and then exit. Without this system of authentication, it would be very simple to pose as the client or bank, and receive the symmetric key used for banking transactions.

Finally, banking operations commenced. Using the symmetric key shared earlier, the client would communicate requests to the bank and the bank in turn would return statuses (or warnings) to the ATM. These messages are encrypted with 3DES, a system of DES that is run 3 times with 3 different keys for extra security. The client can request 4 options: the checking of their account balance, the withdrawal of money from their account, a deposit of money into their account, or a quit (out of the system as a whole). Operations would continue until an error was encountered (which will be explained later), or the client chose to quit. Additionally, if the client input non-cooperative messages (such as strings that do not constitute operations, or values of money to deposit/withdraw that are not integers, or values that falsely reflected the maximum that

a user could withdraw from their account given their current balance) they would receive a warning message and nothing would be communicated to the bank. This was in order to provide incoming messages to the bank that can always be understood, so the server would not error and close out/ cease.

However, in the event that the server were to close out/ cease, there would be two distinct reasons for why it could have done so. Firstly, the client could have sent a message to the server whose MAC was unable to be validated, which would indicate that that client may have been trying to send fraudulent messages, such as those that would allow them to perform actions that inaccurately reflect the state of their account, for example, withdrawing more money than they currently have in said account. Secondly, the time-stamping of a message sent from a client to the server could have been found to have been invalid on the basis of when it was originally sent, which would lend vulnerability to the attempt of a “man-in-the-middle” attack. Such an attack would entail the client attempting to send stale messages that they’ve procured whose invalid time-stamps paired with valid signatures would not be flagged as fraudulent and would therefore be wrongfully processed and executed.

For Symmetric Encryption we used a triple DES cipher. We felt that for the scope of this project, it would be unlikely that an opposing black hat team would be able to decode the ciphertext in the amount of time given (which was roughly equivalent to a week). The base DES algorithm uses the standard algorithm, getting the s-boxes and permutations. The source used was: https://csrc.nist.gov/csrc/media/publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf which is the official government archives of the DES cipher. The s-boxes and

permutations were also taken from this standard. The algorithm for triple DES is also shown in Appendix 2 of this document.

The DES algorithm first divides the plaintext into 64 bit blocks. Each block then undergoes 16 rounds of confusion and diffusion. For each round the block is split into two half-rounds and put into an F function that applies the SBOXes to them and permutes each block. The F function first subjects the half block to an expansion permutation. The result of that is then xored with the subkey for that round. Then every 6 bits are placed into 8 SBOXes. Each SBOX outputs a 4-bit string and all the SBOX outputs are combined to form the resulting half-block of 32-bits. The outputs of the SBOXes are then subject to a final permutation before the F function is finished. The blocks are swapped between rounds. The result of this is a 64-bit ciphertext. This process is done for every 64-bit block to produce the corresponding plaintext. If the last block is less than 64-bits then the block is padded with zeros to become a full 64-bit block. This doesn't represent a significant vulnerability due to the Cipher Block Chain (CBC) mode further explained below. The last block will be XORed with the previous block and thus the padding will be obscured.

The DES algorithm uses CBC mode. This also prevents frequency analysis that would have occurred from electronic code block (ECB) mode. We did not use ECB mode due to the many vulnerabilities that arise from it. To encrypt the plaintext each block would be xored with the ciphertext of the previous block. This means that each block would depend on the previous blocks and if the same sequence of characters were input into two different blocks the ciphertext would not indicate this.

The subkeys for each round were XORed with the expanded half-block. The subkeys are generated by an algorithm shown in the DES standard. First the key undergoes an initial permutation. Then the key is split into two halves. Each half is then left-shifted a specific number of times each round. Subsequently, to generate a subkey, the key halves are then combined and another permutation is applied to generate a 48-bit subkey.

Triple DES applies the DES encryption three times to the entire plaintext, using three DES keys. The first time the plaintext is encrypted with DES and the first key. The second time the result of the first encryption is decrypted using the second key. Lastly, the final resulting ciphertext is generated from the output of the second encryption and the third key. Additionally, though it likely need not be stated, any three keys used were functionally distinct.

Furthermore, The triple DES cipher uses a 192 bit key making it infeasible to test all 2^{92} possible keys. While it is true that a DES key only uses 56 bits of its key of 64 bits, the security of the key is still enough for this purpose. 56 * 3 = 168 bits for a key, which is still a very secure key length. In our implementation the default python random number generator is used to generate the key. The random number generator is seeded with the nanosecond time when the generation is called. This does not represent a vulnerability as the blackhat team should not be able to determine the nanosecond time that the server uses as it is running.

Additionally, the bank never outputs any information about the client's balance. The only information it outputs is which command the client has sent, (i.e. check, withdraw, deposit, quit), in order to provide security to the client. We wouldn't want

anyone listening in to know something such as how much money is in the client's account, which is why the remaining balance is encrypted before it is sent back to the client.

Lastly, there are some additional securities distributed throughout in order to provide a smoother experience, and to recover gracefully in case of an attack. The server checks if it has received no data, (i.e. message was lost or deleted in transit), and exits if it discovers this. The server also checks to make sure every message is of the expected length, and if the HMAC of a received message matches the HMAC of the original message appended to the plaintext. This is to ensure that the message was not tampered with in transit. Finally, the server attempts to catch unicode decode/encode errors with try/except blocks, in order to not crash the server if it receives garbage/tampered data.

With these securities and protocols in place, we hope to create a secure banking menu which is not vulnerable to attack and lets ATMs and banks trust who they are working with, using RSA, DES, and MAC's.

Note that in this project, there were lots of options for systems we could have used, but decided not to. Systems we decided not to use are listed below, along with a description of why we thought a process we used was better:

AES:

Although AES is provably more secure than DES or 3DES, we decided to leave it behind in favor of a 3DES process. The reason for this was twofold: one, our group felt much more comfortable implementing a version of DES as we had already gotten

practice with DES in an earlier assignment. We did not want to risk implementing a totally new system incorrectly, which could result in vulnerabilities in our encryption. Additionally, 3DES offers enough security against the black-hat team's short timeframe, and we felt the time we saved in implementing 3DES was better spent in designing stronger authentication algorithms then it would have been in implementing the stronger AES encryption.

Diffie-Hellman:

Although DH is a workable public-key encryption algorithm similar to RSA, we felt authentication was a very important part of this project's security, and RSA provided much better benefits than DH in this field. A major part of RSA is that it assures confidentiality, authentication, and integrity of our handshake messages, qualities we felt were very important in banking processes like this. Additionally, DH leaves the user sensitive to man-in-the-middle attacks, which we were working as hard as possible to prevent. With DH implemented, we would have had to implement additional authentication procedures to ensure client and server could trust each other, and therefore decided that RSA was superior at this job.

MD5:

Additionally, we needed to pick a hashing algorithm for our HMAC function. Although MD5 is simple to implement, Sha-1's complexity is what gives it the edge here as it takes many more operations to perform a brute-force attack against SHA-1 as compared to MD5. For example, if one wants to perform an attack similar to one from

one of our previous homework's and find two messages with the same hash, they would have to perform 2^{80} operations on a message with SHA-1, but only 2^{64} operations on an MD5 message. Time is essential in this project, and a simple numbers advantage for the price of a slightly more complex algorithm which would slow the black-hat team down even more seemed worth it to our group. Therefore, we chose to implement SHA-1 in our project.

SHA-2 (and the extended SHA-X family):

Somewhat similar to our reasoning for not using an implementation of DH, though SHA-2 and its derivatives are hashing functions that are more functionally secure than their SHA-1 predecessor, due to having varying static hashed value lengths of 224, 256, 384 or 512 bits, we felt that using any of them would've been unnecessary due to the value of their increased security not necessarily making up for the increase in their algorithmic complexities.