Aidan Flaherty

Cryptography ATM Project

My role in the project was the implementation of the symmetric ciphers, hashing, and authentication methods for use by the system architecture. This report will delve into the cryptographic components first, and then detail how they are employed in the overall program.

DES Algorithm

For our project, we elected to go with Triple-DES, a variant of DES, as our symmetric encryption algorithm. This was chosen over S-DES due to the fact that S-DES is designed for educational purposes alone and is not intended to protect sensitive information from adversarial attacks. DES was also deemed to be insecure, as the 56-bit key size is too small to provide enough protection. By stacking 3 DES algorithms on top of each other, a more reasonable cipher is achieved.

The Triple-DES algorithm accepts a key of 128 bits, and converts this into two keys of 64 bits. Within each DES algorithm, these 64-bit keys are further reduced to 58 bits. The first key is employed in both the first and last layers, while the 2nd key is only used for the middle layer. The reasoning behind this is that using additional key would not provide any additional security due to the meet-in-the-middle attack, and therefore 112-bits of key space is the maximum security that Triple DES can offer.

The algorithm begins by generating the round keys, with order of access dependent on whether encryption or decryption is being performed. The round key generation algorithm begins by splitting the key into 28 bit halves. For each round, the bits on each half are rotated either 1 or 2 times based on a predetermined schedule. To produce the round key, the halves are joined and then permuted to 48 bits.

DES operates on blocks of 64 bits, and therefore all inputs are split into blocks before fed into DES. The 64-bit block is first permuted, then divided into 32-bit halves. For a number of rounds, a function is performed on the right half, XORed with the left half, and then swapped with the left half. This function contains the nonlinear component of the cipher.

The round function first permutes the 32-bit input into 48-bits, and then XORs it with the round key. The data is partitioned into 6 bit intervals for a total of 8 blocks, which are fed into a table of s-boxes. The first and last bit of 6 bits are used to calculate a row from 0 to 3 and the middle bits are used to calculate a column from 0 to 15. The outputs from the s-boxes are all 4-bit values, and proceed to get concatenated and then permuted. This results in an output of 32 bits from the round function.

After iterating through all of the rounds, the cipher joins the halves together and reverses the initial permutation performed on the data. This 64-bit value is returned as the encrypted text.

For additional security, we elected to use CBC to make it more difficult for the adversary to identify patterns in the ciphertext. Since the first value of all messages we send is a timestamp in nanoseconds, this ensures that identical messages sent at different times always look completely different from each other. Additionally, adversaries with a knowledge of the

placement of certain information will not be able to categorize ciphertexts based on whether they have an 8 byte segment in common.

SHA256

We chose to use SHA256 as our hashing algorithm, as it is significantly more secure than SHA1 while belonging to the same family of algorithms.

The algorithm works by ensuring that the text can be parsed in 512-bit blocks. A 1 bit is appended to the message, and 0 bits are appended until the message length is 448 mod 512. The length of the original message is then concatenated to the bit string as a 64-bit integer, making the final length a multiple of 512.

The algorithm initializes 8 32-bit registers to predetermined values, which are also the registers that compose the 256-bit output of SHA256. For each chunk of 512 bits, an array of 32-bit integers is constructed, of which the chunk is loaded into the first 16.

The 17th to the 64th values in the array are all calculated by manipulating previous values in the array. By accessing previous values and XORing various rotations and shifts of the bits, the algorithm produces the next item in the array. Once the array is completely full, the algorithm moves on to the next step.

There are 64 rounds of the same series of operations in order to generate the final states of the registers. The registers are initially set to be the 8 predefined register values; these are XORed with rotated versions of each other, ANDed with each other, and added together with the previous array to produce new register values (which are then rotated as well). The final

register values computed serve as the starting register values for the next block of text that undergoes the chunk algorithm, and the final registers are concatenated to become the output of the SHA256 algorithm.

HMAC

The HMAC algorithm is a simple application of the SHA256 algorithm. We obtain a secret key and the text to produce the MAC for. Since the key is 128 bits in our case, it needed to be padded with zeros until it was 512 bits long.

The 512-bit padded key is then XORed with 0x5c repeated to fill 512 bits and 0x36 repeated to fill 512 bits. The latter XOR value is concatenated with the text and fed into the SHA256 algorithm. This 256-bit value is then concatenated with the first XOR value we computed and fed into the SHA256 algorithm. This value is returned as the HMAC.

Blum-Goldwasser Key Exchange

Our choice for PKC was a semantically-secure method where the encryption of the same message looks radically different due to the influence of random numbers. The algorithm uses a public key n, which is equal to the multiplication of two larges primes p and q generated client-side. For the algorithm to work, these two primes must be congruent to 3 mod 4, so we filter out any primes that do not adhere to this rule. The final part of our private key is generated through an implementation of the Euclidean Extended Algorithm, which gives us some a and b such that $ap + bq = 1$.

The encryption of the method begins by dividing up the message into blocks of size h, where h = **floor(log$_2$(k))** and k = **floor(log$_2$(n))**. This means that the division is completely dependent on the random size of n.

A number x_0 is generated that is equal to some random r^2 mod n. For each iteration of the algorithm, this x is squared mod n. The h least significant bits are then collected and XORed with the plaintext to produce the ciphertext block. After all these blocks are collected, they are sent along with the last value of x after squaring once more.

Decryption uses the values in the private key to reconstruct x_0 from the last value of x, and then runs the exact same algorithm to undo the XORing of the ciphertext and get the plaintext.

During key exchange, our server sends their public key to the client, which then encrypts a session key using that public key. Even if someone intercepts it, only the server can read the session key value using its private key. Both parties then switch to symmetric encryption using the session key that was safely passed, and employ the Triple-DES algorithm referenced earlier.

Server Model

After the BG key exchange takes place, the server waits for messages from the model. These messages take on a certain format:

<8-byte timestamp><4-byte id><32-byte MAC><5-byte message><4-byte amount>

and the server parses decrypted plaintext expecting that the format of the message is constrained to these values and spacing.

Upon receipt of a message from the client, the server decrypts it using the session key and triple-DES. It then concatenates all of the sensitive information and feeds it into the HMAC function using the same session key, and verifies that nothing was tampered with. The timestamp is inspected as well to defend against replay attacks.

When the server responds to requests from the client, it sends back the nanosecond timestamp of the message that it was replying to. This is to ensure that someone cannot capture old server messages and use them to respond to new client requests. The server also keeps a log of timestamps for client requests, and does not permit multiple requests for the same nanosecond time. This ensures that out of many duplicated messages, only one will go through. The server rejects any messages that are greater than 10 seconds old to protect against replay attacks.

The working of the ATM account itself is fairly self-explanatory. Our model assumes that there is only one ATM machine made by our company in the world, and that everyone using the machine gets to use the same account (due to login systems not being part of the assignment). The server knows the public key of the ATM machine, and during key exchange sends its public key encrypted using the public key of the ATM machine. In order to establish a shared key for symmetric encryption, only our ATM machine will be able to actually send a key properly encrypted with the server public key. A fake client that does not have access to the inner workings of the ATM machine will not be able to send the server a correct shared key.

Whenever an error is encountered, the connection hangs. The intention of this is to leak no information about the session key through error messages.

Client Model

The client has a hardcoded private key for the Blum-Goldwasser algorithm. This key is used during key exchange to decipher the server's public key.

When receiving responses to requests, the client ensures that the timestamp of the query the response is targeting is, in fact, the request that it just sent. This prevents bad actors from impersonating the server and sending old captured requests.

Many of the same validations are performed client side as well. The MAC is verified to be generated from the decoded sensitive information.

The client initiates the handshake with the server by sending 'HELLO', which prompts a return 'HELLO' and for the server to send its public key (encrypted using the public key it has hardcoded for its known client).