Yash Garala

CSCI 4230

Professor Yener

14 April, 2024

Cryptography ATM Final Project

Our Cryptography implementation can be broken down into three main groups: the SSH/SSL Handshake Protocol and key exchange, ATM banking model, and Encryption Schemes. In order to simulate the ATM, we have implemented different cryptosystems and Client/Server communication in order to securely complete ATM transactions. We start off by using SSH/SSL Handshake Protocol and Blum-Goldwasser for key exchange. This key exchange then enables us to use Triple DES since both the Client and Server have key access. This is doubly strengthened by usage of HMAC, timestamps and SHA256.

In order to land upon the cryptosystems and implementations that we decided to use, there were a lot of pros and cons that we had to consider. For the handshake protocol and key exchange, our original plan was to use some form of Diffie-Hellman or RSA, but realized that there were some vulnerabilities that were present in these. As a result, we decided to use Blum-Goldwasser since it would be more secure and harder to crack for an adversary. Next, we originally considered using AES or S-DES for our symmetric message encryption. The problem with these are security concerns against cryptanalysis which would put our ATM system at risk. As a result, we decided Triple DES would be much more secure as it cannot be brute forced or taken advantage of as easily as the previous two mentioned. Our last choice of security is using timestamps and SHA256. Usage of timestamps helps us to counter replay attacks from adversaries and allows us to validate our request. Going hand in hand with this is using SHA256

because it is much more secure than SHA1 and can be used in our HMAC algorithm for our MAC digital signature scheme.

The overall flow of our ATM is starting the handshake protocol with Blum-Goldwasser Key Exchange, encrypting messages using Triple DES along with usage of timestamps and HMAC, and using these messages to complete said ATM interactions. The main section that I focused on was the SSH/SSL handshake protocol and using Blum-Goldwasser to create a secure key exchange to enable usage of Triple DES encryption for future messages. The rest of this report will delve deeper into how I implemented this portion of our ATM project and why it works and is secure.

The SSH/SSL Handshake Protocol is implemented using the following steps:

1. Starting the Server
2. Starting the Client and starting a request to the Server
3. Sending and Receiving “Hello” messages from the Client to the Server and vice versa
4. The Server Sending a randomly generated public key to the Client
5. The Client sending a random, encrypted shared key to the Client
6. Decrypting the encrypted shared key on the Server side
7. Sending a finally Acknowledgment message from the Server to the Client to complete the SSH/SSL Protocol

The implementation of starting the Server and Clients is standardized through the usage of the Socket library in Python. We start by setting up the server and listening for a connection on the Client side. Upon connection, the Client will send a “hello” message to the server, which

will send a reciprocated “hello” message back to the Client upon reception. Now that the connection has been established, the key exchange can begin.

Instead of using RSA or Diffie-Hellman for the key exchange which can be proven to have some adversarial weaknesses, we opted to use Blum-Goldwasser encryption to create a shared key between the server and the Client. In order to use Blum-Goldwasser for key exchange, the server will generate a random 128 bit key. This key will be used as the public key for encryption on the Client and server side. In addition to this public key, a separate private key is also generated on only the Server side based on the public key. This private key is used solely for decryption on the Server side. Following the SSH/SSL handshake protocol, this public key is now sent from the Server side to the Client side. Now that the Client has been shared the public key, it can generate a new 128 bit shared key that will be used for Triple DES encryption in later steps. This new shared key will be encrypted using the public key from the Server using Blum-Goldwasser encryption and the Client will send the shared key as an encrypted message back to the Server. Upon receiving the encrypted message, the Server is able to decrypt the Blum-Goldwasser encryption using the previously calculated private keys it possesses to obtain a shared key that both the Server and Client now have secure access to.

Using these newly obtained shared keys, the messages from the client and server can now be encrypted using Triple DES to securely send messages back and forth. These messages will be what is used for the ATM interactions such as checking balance, withdrawing, and depositing money. Finally, to conclude the SSH/SSL handshake protocol a simple Acknowledgement message is sent from the Server to the Client once all ATM actions have concluded.

Our plan to use Blum-Goldwasser Probabilistic Encryption is as mentioned earlier: to provide a more secure key exchange than Diffie-Hellman or other encryptions. The strength in

Blum-Goldwasser relies on using quadratic residues in a modular arithmetic field. Key steps for how I implemented this encryption is as follows:

1. Key Generation (keygen function):
    a. Prime Generation: The generate_blum_primes function generates two large prime numbers, p and q, each congruent to 3 modulo 4. This specific congruence condition is crucial as it ensures that the modulus n = p * q used in the encryption and decryption processes has certain mathematical properties beneficial for security, specifically related to the Jacobi symbol which impacts the predictability of quadratic residues.
    b. Public and Private Keys: The public key is the product n, while the private key consists of the primes p and q and the coefficients a and b obtained from the extended Euclidean algorithm ensuring ap + bq = 1. This private key structure is essential for the decryption process.

2. Encryption (encrypt function):
    a. Initial Setup: Encryption starts with an initial quadratic residue x_0 and breaks the message into blocks. The size of each block is determined by h, which is derived from the bit length of n.
    b. Block Processing: Each block is encrypted by squaring xi modulo n, extracting h least significant bits from x_i and then XORing these bits with the message block. This method leverages the one-time pad principle where the unpredictability of x_i's bits (due to quadratic residue properties) ensures strong security for each block.

c. Final State: The last part of the ciphertext is x_{t+1}, which is crucial for the decryption process as it encapsulates the final state of the encryption sequence, aiding in backward calculation during decryption.

3. Decryption (decrypt function):
    a. Recovery of x_0: The function first uses the private key and the last encrypted element x_{t+1} to recover x_0. This involves complex computations using the Chinese Remainder Theorem, which helps reconstruct x_0 from its residues modulo p and q.
    b. Message Reconstruction: Starting from x_0, the function iteratively recomputes each x_i to retrieve the original message blocks. This step exploits the properties of the XOR operation used in encryption, which is reversible given the same input.

Using Blum-Goldwasser provides our ATM implementation with a variety of strengths. Each time the ATM is started, a new random x_0, making it resistant to many forms of attack even if future keys are compromised, providing forward secrecy. Due to the usage of quadratic residues, the difficulty of factoring n and the unpredictability of quadratic residues modulo n provide strong security guarantees and tough computational security. The method is efficient in terms of both computational requirements and message expansion, which is particularly advantageous for systems with resource constraints such as an ATM which requires efficiency. Overall, the Blum-Goldwasser encryption method exemplifies a secure and efficient asymmetric algorithm, ideal for environments where key management and data security are paramount. The modular arithmetic foundation provides both security (through computational hardness assumptions) and flexibility, adapting well to different types of cryptographic applications.