Cryptography Final Project

April 14 2024

Angelica Loshak

White Hat

Overview

The ATM/bank system is designed to handle requests from a client while securing communication and interactions through various cryptographic techniques.
The main goal of the Bank server is to allow for secure communication between an authenticated ATM in order to handle client requests related to banking operations, such as account authentication, deposits, withdrawals, and balance inquiries. These transactions between the server and client are designed to be secure against unauthorized access, eavesdropping, replay attacks, man in the middle, and brute force attacks.
To maintain this level of security, the system makes use of digital signatures, symmetric encryption, hashing, and serialization. The server authenticates the ATM client based on their bank card numbers and PINs stored in a credentials database. Communication between the server and clients is secured using cryptographic protocols, including Diffie-Hellman key exchange and RSA encryption. Sensitive data, such as bank card numbers and PINs, are hashed using SHA-1 before being stored in the credentials database, and the encrypted messages are serialized to prevent replay attacks.

In the following sections, we will discuss the various cryptographic protocols that are used in ATM/bank systems and how they are implemented.

Cryptographic Algorithms

TLS

The Transport Layer Security (TLS) protocol is a secure communication protocol to provide communication between the ATM and the bank over the network. It is built on top of the Secure Sockets Layer (SSL) protocol and adds a layer of security to it. By using TLS communication between the ATM and the bank, we can prevent eavesdropping and tampering with messages that are sent over an unprotected communication channel.

In this project, TLS initiates the connection and establishes a shared key between the ATM and the bank. Both parties establish a handshake using the Diffie-Hellman key exchange algorithm. This shared key is then used in the 3-DES encryption algorithm in order to encrypt the messages exchanged between the bank and the ATM.

The TLS allows the ATM and the bank to authenticate themselves to one another through the Rivest-Shamir-Adleman (RSA) digital signature scheme. During the TLS handshake, the bank provides its RSA public key to the ATM. This key exchange allows the ATM to encrypt data that only the bank, with its corresponding private key, can decrypt. The ATM is able to verify the bank's identity by decrypting the digital signature with its private key. This process is then reversed to verify the ATM's identity to the bank.

SHA-1 Hashing

The SHA-1 hashing algorithm is used to hash the bank card numbers and PINs before storing them in the database on the server side. On the client side, SHA-1 is used to hash the bank card number and PIN before encrypting them and sending them to the bank server. Hashing sensitive data is important so that even if an adversary gains access to the database, the data isn't useful because a hash is a one-way function which can't be reversed to reveal the user's bank card and PIN. It is important to note that SHA-1 is no longer considered to be secure because of its vulnerability to collision attacks. If this ATM/bank system were to be implemented to be used in the real world, hashing data should be switched to using SHA-256.

The implementation of the SHA-1 algorithm has a block length of 64 bytes. The message is separated and operated on in blocks of size 64. The remaining block of the message is padded with "0x00" until it reaches the desired length. Despite the message size, the output hash is fixed to a size of 20 bytes. For example the 12 byte message "CRYPTOGRAPHY" outputs the SHA-1 hash: "0a59264f6d6b69d9c6bb79c5bd0f4199e464e0c0". After padding the message to be a multiple of 64 bytes, the message is divided into 64 byte blocks. Each 64 byte block is then divided into 16 4-byte words which are then extended into 80 4-byte words. We initialize the hash values according to the SHA-1 standard: H0 = 0x67452301, H1 = 0xEFCDAB89, H2 = 0x98BADCFE, H3 = 0x10325476, and H4 = 0xC3D2E1F0. After initializing these values as the initial state of the hash function, we perform a series of 80 rounds of bit manipulations. During each round, we process each word in the input message, applying specific bitwise operations based on the current round index. After the final round of processing, the hashed blocks are concatenated into a 160-bit number, which forms the final hash value.

The SHA-1 algorithm is used by including the header file in the program using the directive '#include "crypto/SHA-1.h"'. Next, an instance of the SHA-1 hashing object is created with the code: 'SHA1 sha1;', and a message is passed to the SHA-1 hashing function using 'sha1(msg);', which computes the SHA-1 hash of the message.

HMAC

The hash-based message authentication code (HMAC) is used to authenticate the ATM to the bank and the bank back to the ATM. SHA-1 algorithm, which was discussed earlier, is used to calculate the HMAC. Although SHA-1 is no longer secure on its own, it is still considered to be secure for HMAC.

The HMAC produces a 20 byte sized hash output. This output is the same size as the SHA-1 output, because HMAC uses SHA-1 to hash the key and the message together. The way that HMAC is implemented in the ATM/bank system is it takes in a key of any length and resizes it to fit a multiple of a 64 byte block size. This key is then padded to create an outer key and an inner key. The outer key is produced by xoring the padded key with 64 bytes of repeated 0x5c hex bytes. Similarly, the inner key is produced by xoring the padded key with 64 bytes of repeated 0x36 hex bytes. The final hash is created using the equation H(outer key + H(inner key + M)), where H is the SHA-1 hash function, and M is the message. When the other party receives the HMAC along with the message, they will be able to use their secret key and create an HMAC with the message, and if it matches, then it is authentic.

DH

The Diffie–Hellman (DH) is an asymmetric key exchange algorithm used by the ATM/bank system in order to create and exchange a shared key between the bank and the ATM party. This shared established key is used in the 3-DES encryption system which will be discussed in the next section.

The way the DH algorithm is implemented in our system is the ATM generates a random number r_a for the local secret, and sends g ^ r_a (mod p) to the bank. The bank generates their own random number r_b and in return sends g ^ r_b (mod p) to the ATM. The ATM calculates k = (g ^ r_a) ^ r_b (mod p) and the bank calculates k = (g ^ r_b) ^ r_a (mod p). Both parties arrive at the same shared key k. Notably, r_a and r_b are secret variables, while g and p are public. The g is a small prime number generator. In this implementation g is chosen to be the value 2. p is a large prime which defines the finite field for which the computations are performed.

3-DES

The 3-DES cipher algorithm is a symmetric key cipher which is used to ensure secure communication between the bank and ATM via encryption. The 3-DES uses a triple iteration of DES encryption to address the security vulnerabilities that come up in using DES alone. However, 3-DES has been identified as insecure. Therefore, for actual implementation of this ATM/bank system, it is recommended to transition to using AES instead of DES for enhanced security.

The DES key is 56 bits long. 3-DES is a method of increasing the key size. Using two keys with DES isn't enough to increase security because it introduces the man-in-the-middle attack. Instead, the 3-DES implementation uses three 56 bit keys. To encrypt the message: $C = E(D(E(M, K1), K2), K3)$ and in order to decrypt the message we reverse the encryption process: $M = D(E(D(C, K3), K2), K1)$.

DSA

The Digital Signature Algorithm (DSA) establishes trust between the parties and ensures message authenticity between the ATM and the bank within our system. By leveraging the RSA algorithm, DSA verifies the sender of messages, confirming they indeed come from the authentic party holding the private key.

Each ATM and bank come with a set of pre-assigned public and private keys. When initiating a connection, a party sends an encrypted message, signed with its RSA private key as a digital signature, to verify its identity.

RSA encryption involves a message M, a public key e, and a private key d, along with a modulus N derived from two large random primes, p and q ($N = p * q$). The encryption equation is represented as $C \equiv M^e \pmod N$, while decryption occurs through $M \equiv C^d \pmod N$. The relationship between the encryption and decryption keys, e and d, is defined by $e * d \equiv 1 \pmod{\varphi(N)}$, where $\varphi(N)$ represents Euler's totient function.

The bank generates an RSA key pair consisting of a public key e and a private key d. These keys are used for encryption and decryption processes, respectively. During the TLS handshake, the bank securely sends its RSA public key to the ATM. This key exchange enables the ATM to encrypt sensitive data using the bank's public key, ensuring that only the bank, with its corresponding private key, can decrypt the information. When the bank sends a message to the ATM, it signs the message using its private key. The ATM can verify the authenticity of the message by decrypting the digital signature using the bank's public key. Similarly, if the ATM needs to prove its identity to the bank, it uses its own RSA key pair. The ATM sends a signed message along with its public key to the bank. The bank verifies the ATM's identity by decrypting the signature with the ATM's public key and ensures the integrity of the message.

The private and public keys for the ATM and the bank are securely stored in the 'keys' folder of the project, accessible through functions such as private_key.getN(), private_key.getD(), public_key.getE(), and public_key.getN(). We assume that in a real world implementation, these keys would be kept more securely, and that these keys would be rotated periodically to prevent any key exposure over the long term.

Looking ahead, one potential risk associated with RSA digital signatures is the development of Shor’s Algorithm implemented on Quantum Computers, which theoretically has the capability to factorize the large primes p and q, and compromise the RSA security. It is recommended to transition to post-quantum cryptographic algorithms like elliptic curve cryptography, especially if the system is expected to remain in use over the next decade.

C++ Libraries

C++

This banking project is implemented in C++ due to the language’s performance, reliability, security, and flexibility. C++ has low-level control over system resources and memory, which is important for handling many transactions between the Bank and potentially multiple ATM systems. C++ also has various tools and libraries which came in handy for this project. For example, libraries like Boost provide multiprecision arithmetic to handle large numbers required in cryptographic operations like RSA encryption and digital signatures. C++'s static typing and compile-time error checking increase the security and robustness of the banking system.

C++ is also compatible across systems. However, this project was designed to run on a Linux operating system.

Boost.Multiprecision

It is important to support large numbers when implementing cryptographic algorithms that rely on large prime numbers for security. To handle these large numbers effectively, we utilize the Boost Multiprecision library. This library provides our project with increased precision and a broader range of integers. By incorporating Boost Multiprecision integers with the cryptographic algorithms, we can store and compute large numbers and store large account balances for users without encountering overflow issues. This approach helps prevent various binary exploitation attacks, such as buffer overflows.

Serialization

Using the Protobuf library, messages go through serialization before they are encrypted using the 3-DES encryption scheme. This process serves as a defense against replay attacks. An example of a replay attack scenario is one where an attacker intercepts a message and replays it later for malicious purposes.

The serialization mechanism combats replay attacks by including a sequence number with each message before encryption. After receiving a message, the recipient verifies that the sequence number aligns with the expected order. If a message arrives with a sequence number which is out of order, then the message is not accepted.

For an attacker to still be able to carry out a replay attack, they would need to decrypt the intercepted message, determine the correct sequence number to append, and then re-encrypt the message before transmission.

Binary Exploitation

Despite the robustness of the cryptographic algorithms used to secure data, it is also important to prevent binary attacks in the system, which can be just as harmful and easier to exploit. We ensure protection against buffer overflow and integer overflow/underflows. By setting a BANK_LIMIT to 10^100, we prevent users from attempting to overflow their account, which could lead to unexpected behaviors.

Furthermore, we enforce restrictions to prevent account balances from going into negatives. If the balance somehow falls below $0, the bank system automatically recalibrates itself by resetting the account balance to $0. Additionally, during deposit or withdrawal transactions, users are required to input numerical values only. If a user attempts to input characters, the bank system will return an "Invalid Amount" error. Any other attempts to exploit the bank system will result in a "Deposit Not Approved" error, and the transaction will not be approved.

To counter brute force attacks on the login page, we limit the number of login attempts to three. After three unsuccessful attempts, the ATM will exit, requiring the user to restart the ATM and establish a new connection with the bank before attempting to log in again.

If a user is logged in and disconnects from the bank at any point, the bank system automatically logs the user out and resets all variables holding the user's credentials. This ensures that unauthorized access is prevented and maintains the security of user accounts.

Upon successful login, users are presented with four options: deposit, withdraw, check balance, or exit. If a user attempts to choose an option not on the list, the program defaults to displaying an error message: "Invalid choice, please try again." This prevents unintended actions and maintains the usability and integrity of the system.

Conclusion

The ATM/bank system achieves the goal of secure communication and interactions between authenticated ATMs and the bank. We guarantee the client’s protection against various attacks through the integration of digital signatures, symmetric encryption, hashing, serialization, and key management.

The Transport Layer Security (TLS) provides a secure communication channel between the ATM and the bank by incorporating the Diffie-Hellman key exchange and RSA encryption. The SHA-1 hashing protects sensitive data such as bank card numbers and PINs, and the HMAC utilizes the SHA-1 to verify message integrity and authenticity. The Diffie–Hellman algorithm creates a shared key between the bank and the ATM, which is used by the 3-DES cipher to encrypt the messages sent between the ATM and the bank.