Christopher Reed

Cryptography and Network Security

April 14th, 2024

White Hat Report

Security of communication between ATMs and Banks is a necessity. Banks store lots of sensitive information and maintain a huge responsibility to keep people's money safe. As a group, we decided to use C++ for our implementation. C++ was chosen due to its fast performance, strong compile time guarantees, and its familiarity in my group. Having compile time guarantees made development much easier by catching more errors in the development process rather than the testing process. However, there were still many errors caught in the testing process due to the complexity of the application.

For this project, we decided to make a mock TLS implementation. Our mock implementation has a few very useful guarantees. Our first guarantee is that when establishing communication, we can be certain we are talking to the true bank and not any man in the middle or other server pretending to be the bank. In this case of an IP spoofing attack, communication would be attempted with the false bank, however, it would not make it past the second message. An RSA signature is required on the second message. Another guarantee we have is that no adversary will be able to read any messages transmitted. This is achieved by using symmetric encryption cryptography with its key derived from secure, asymmetric cryptography methods. The last guarantee we have is that no message can be replayed which was achieved by using sequence numbers. This is important since it is fairly easy to replay encrypted messages which can alter the bank state.

To build a mock TLS, we needed to create our own cipher suite. Our chosen cipher suite consists of RSA (Rivest–Shamir–Adleman), HMAC (Hash Message Authentication Code), SHA1 (Secure Hash Algorithm 1), DES (Data Encryption Standard), and DH (Diffie-Hellman). RSA is used to authenticate the bank to the ATM. An RSA key pair was generated ahead of time where the private key was given to the bank and the public key was distributed to all ATMs. When the bank communicates with an ATM before mock TLS is established, it signs the message with its private key. HMAC is used as an extra layer of security on encrypted messages to prevent CCA (Chosen Ciphertext Attacks). The HMAC is verified every time before decrypting with DES. SHA1 was used in the HMAC implementation as a hash function. DES (specifically 3-DES) is used as an asymmetric encryption method for all data after the mock TLS handshake. DH is used during the mock TLS handshake to establish a secure shared secret between the bank and ATM. With all of these together, the guarantees discussed in the previous paragraph are possible.

Many of these ciphers require mathematical operations on large numbers which C++ does not natively support. The largest size supported in C++ is 64 bits while the size we need is more on the order of thousands of bits. We used a library called Boost Multiprecision to allow for support of arbitrarily large integers. Due to C++'s operator overrides, the usage of this library was fairly simple and allowed for the easy implementation of the ciphers.

In order to communicate between the bank and ATM, we used standard Linux sockets using TCP. TCP was chosen since it establishes a connection and guarantees the message will get to the other end. The bank runs a TCP server which accepts incoming ATM connections. The way the code is written, only 1 ATM connection is supported at a time. This code could be extended to support more but for simplicity and security, we only allow 1 ATM to connect with the bank at a time. Once an ATM disconnects, another ATM is allowed to connect again.

Upon initial connection to the bank, our mock TLS handshake begins. Standard TLS has many types of messages during the handshake while mock TLS only has 3. The handshake is described as follows:

1. TCP connection is established between the ATM and bank.
2. The bank sends a ServerHello message which contains the DH partially computed secret and is signed by the bank's private key.
3. The ATM verifies the ServerHello message signature and computes the full DH shared secret.
4. The ATM sends a ClientHello message which contains the DH partially computed secret.
5. The bank can now also compute the full DH shared secret.
6. The ATM and bank can both compute a DES key from the DH shared secret.
7. The bank sends the HMAC key encrypted by using 3-DES.
8. At this point, both parties have the symmetric key and HMAC key which declares the handshake complete.

If at any point, there is some cryptography failure, the handshake and underlying TCP connection is aborted.

Data serialization is an important factor to consider when designing an application which will send messages over the network. For this project, we used a library called Protobuf to easily serialize data and send them over the network. The library allows us to define the structure of the message (found in proto/messages.proto) which it then turns to C++ code that we can use. The messages we use are ServerHelloContent, ServerHello, ClientHello, HMACSend, UnencryptedMessage, EncryptedMessage, and MACMessage. ServerHelloContent, ServerHello, ClientHello, and HMACSend are all used for the mock TLS handshake. The ServerHelloContent contains the DH partially computed secret. ServerHello contains the ServerHelloContent and also a signature of the ServerHelloContent. ClientHello contains only the DH partially computed secret. HMACSend contains the encrypted HMAC key and an IV for 3-DES. The UnencryptedMessage, EncryptionMessage, and MACMessage are all used for regular mock TLS communication after the handshake is complete. UnencryptedMessage contains the underlying message we want to send and a sequence number to help prevent replay attacks. EncryptedMessage contains the encrypted UnencryptedMessage and an IV for 3-DES. MACMessage contains the EncryptedMessage and the MAC for verification. Usage of these messages with Protobuf helped make implementation a lot simpler and improved the security of message parsing. Protobuf has a strict encoding scheme which is checked and if a message does not parse correctly, we abort the entire TCP connection. We did encounter some issues with data parsing when testing between different platforms using Protobuf, however, all of the issues/bugs should be fixed.

The underlying protocol that the bank operates on are ascii string messages. The operations that are supported are LOGIN, DEPOSIT, WITHDRAW, and BALANCE. LOGIN will accept a card number and pin and will attempt to authenticate the user on the bank side. If they are authenticated, then they can run the other commands. Any parameters for depositing or withdrawing are also just ascii strings for simplicity. These ascii strings are what the ATM creates and Bank accepts before DES encryption/decryption.

Our project has two C++ files with entry points which are Bank.cpp and ATM.cpp. These compile down to Bank and ATM binaries which are used to actually run the project. Bank is responsible for setting up its TCP server, maintaining customer data, and handling ATM requests. Bank stores a map corresponding card numbers to their pin. Bank stores another map for corresponding the card number to an amount of money in that account. Bank also stores the RSA private key, HMAC key, DES key, and sequence number. ATM is responsible for connecting to the Bank and handling user interaction. ATM stores the RSA public key, DES key, HMAC key, and sequence number. The binaries each accept one argument for describing which port to use when binding to localhost. Once the bank accepts a TCP connection, it does the mock TLS handshake and then infinitely loops listening for the commands that the ATM requests. The ATM makes the connection and sends the needed message and listens for a response back.

RSA was used for digital signatures to prove the bank is who they say they are. This helps against IP spoofing attacks where someone else may pretend to be the bank and steal the credentials of the user. The implementation of RSA was just textbook RSA using the Boost Multiprecision library. We generated an RSA key pair by using the commented out code in RSA.cpp. We used a 4096 bit RSA key to fit the DH partial secret inside the signature. Typically, RSA signatures will be of the hash of the content they signed, but we chose to just sign the whole DH value since it wasn't that long. In the keys directory of our project, there is a public and private key file which stores the required parameters that level of key would need. public.txt contains the n and e value of textbook RSA. private.txt contains the p, q, and d value of textbook RSA. The bank will read in the private.txt file when starting and the ATM will read in the public.txt file when starting. public.txt is assumed to be a file stored inside the ATM which is untampered. A tampered ATM can do whatever it wants but that is considered outside the scope of this project as there isn't much that can be done to protect against that. If this was scaled to production, a certificate with a CA (certificate authority) signature would have been used instead of the RSA key pair. The RSA key pair works well in this project since it is a single ATM.

DH (Diffie Hellman) was used to produce a shared secret between the bank and ATM. DH is a very simple algorithm to implement but offers a highly secure way of transmitting a secret. The constructor of our DH object will create random local secrets for each side. The class gives a function to produce the partial secret which is needed to send to the opposing side of the connection. The partial secret was computed by doing g^{local_secret} mod p. From an external observer, it is very tough to calculate the local secret value since they were have to solve the discrete log problem. p and g are fixed values chosen from RFC 3526 to create a standardized and secure formula to use. Group 15 was chosen to ensure it produces a big enough number for DES keys. Once the opposing side would receive the partial secret from the other, it could then compute the final shared secret by computing $(partial\_secret)^{local\_secret}$ mod p. Our DH implementation uses 2048 bit local secrets to make any attempt of brute force extremely expensive. Once both sides have computed a shared secret, they use that value to generate a DES key for further communication. Communicating over asymmetric communication like RSA and DH is very secure, however, it is also very computationally expensive. The time taken to complete the mock TLS handshake can take over a second, while the rest of the DES encrypted messages are much quicker. By creating this secure shared secret, sending messages is significantly more efficient.

DES (Data Encryption Standard) was used for symmetric encryption for all messages after the mock TLS handshake. 3-DES was the specific implementation chosen as it adds more security on top of DES and brings the key size to 168 bits. All of the needed implementation details were taken from FIPS PUB 46-3 which describes information necessary to create DES. The implementation required doing 16 rounds with 8 different S boxes which that document provided details on. 3-DES encryption just ends up being an encrypt followed by decrypt followed by encrypt on 64-bit DES. The reverse is applied for decryption. It ended up being very similar to S-DES but scaled up bit-wise. In order to encrypt arbitrary sized data, CBC (Cipher Block Chaining) was used to encrypt multiple blocks in a secure way. While CBC can lead to some information leakage, CCA (Chosen Ciphertext Attack) is not allowed due to the HMAC check before any DES decryption occurs. The DES key is derived from the generated full DH shared secret. It will be generated from the 192 least significant bits of the shared secret which is a secure source. 24 of those bits (8 bits per DES) do not get used in the actual encryption due to how DES key scheduling is performed. This still gives an effective key size of 168 bits which is very expensive to brute force. After 64 bit DES, 3-DES, and CBC were written, they were tested against online cryptography tools to ensure correctness of the algorithms.

SHA1 (Secure Hash Algorithm 1) is used as our hashing algorithm for this project. We use hashing to store the bank credentials and in HMAC. SHA1 is known as an insecure hashing algorithm at this point but we were required to use it for this project. The implementation of SHA1 required adding padding of 0x00s initially. After that, many bit manipulations occurred on 64 byte chunks to create a good hash. Once the implementation was complete, it was tested against online hash tools to ensure correctness in which it worked perfectly.

HMAC (Hash Message Authentication Code) was used to ensure data validity that cannot be tampered by an adversary. We use HMAC to wrap the DES encrypted data to not allow for anyone to attempt CCA on DES. The HMAC key is randomly generated by the bank and sent to the ATM. This happens during the mock TLS handshake and immediately after the DES key is generated. Thus, the HMAC key is sent encrypted, however, that message does not have a MAC which means it could be tampered with for a CCA. In the case of any crypto failure, the entire connection terminates without giving a reason why. HMAC used SHA1 for the hashing algorithm since that is what we had to use for this project and was already implemented.

Once the project was implemented, we tested it with various cases to ensure there were no vulnerabilities. We ended up adding an upper bound limit on the amount of money an account

can hold to prevent a potential memory overflow. No buffer overflows should be possible. All other cases worked as expected and there should not be any flaws in this system.

Many of these algorithms require random numbers to be securely generated. If random numbers are predictable then it might be possible to guess the keys created or at least greatly reduce candidate keys if trying to brute force. Linux provides /dev/urandom which is a file that when read will give secure random bytes. We wrote a utility function which will read bytes from that file and give us a vector of bytes. We also made another utility function that will take those random bytes and create Boost Multiprecision integers of specified size.

In conclusion, all of these algorithms come together to produce a cipher suite capable of securely transmitting data between the bank and ATM. Besides the potential flaw of SHA1, no adversary should be able to break this system without brute forcing a huge amount of keys. Every connection produces new keys as well so any past knowledge is useless in this system. As long as the assumptions that the RSA private key is never leaked, the ATM is never tampered with, and SHA1 is not brute forced, then this should be a very secure system that no one can break. These assumptions are what is expected in the real world today so it is very applicable to reality. Overall, this was a really awesome project to make and see come together.