Alex Francoletti

3 December, 2018

Cryptography Project White Hat Implementation

For this project, a white hat implementation of Secure Shell Host(SSH)/Secure Sockets Layer(SSL) protocol which supports a cipher suite including a symmetric key algorithm, a public key cryptosystem that is safe against ciphertext only adversaries, a homomorphic public key cryptosystem, and a semantically secure public key cryptosystem at minimum. It also requires a message authentication code, which is required to be done using hash message authentication code(HMAC), and SHA-1 for the hash function. The following ciphers, algorithms, and protocols were implemented in our white hat version of the project: SSH/SSL, Diffie-Hellman, DES, RSA, and HMAC\_SHA1.

SSL stands for Secure Sockets Layer, which is a cryptographic protocol designed to provide secure communications across computer networks. SSL protocol is used to establish a secure channel for our server, and it does so via an established handshake that goes through 4 phases. In the first phase the client sends a hello message to the server containing information regarding the security capabilities, which includes the supported protocol versions, a session ID, a suite of ciphers to use for encryption, and initial random numbers to be used for key generation. The server also sends a message back containing the same information, but based on what the server can support and a different random number to give to the client. After a hello message is sent back from the server, the secure sockets layer handshake moves on to phase 2. During phase 2 of the handshake, the server may send a certificate, a key exchange, and request a certificate from the client, and always sends a hello finished message. From there, it moves on to phase 3 of the handshake where the client will send their certificate if requested, a certificate verification if sent a certificate from the server, and a key exchange. Finally, in phase 4 of the handshake, the client and server will change cipher suite and finish the handshake protocol. After the handshake has finished, a secure channel for communication between the client and server is established.

Diffie-Hellman key exchange was used in the project as a part of the secure socket layer handshake as the key exchange in phase 2 and 3. Diffie-Hellman is a key exchange algorithm between Alice and Bob agreeing to use an equivalent modulus **p** and base **g**. Alice chooses a secret integer **a** and then send Bob **A = ga mod p**. Bob chooses a secret integer **b** and then send Alice **B = gb mod p**. Alice and Bob can then compute a shared secret number by taking the received values raised to their secret integers mod **p**. This key exchange is very vulnerable to a man in the middle attack, but is otherwise completely secret.

DES stands for Data Encryption Standard, and also what we implemented for a symmetric key cipher. Data encryption standard both encrypts and decrypts messages using one secret key, as per the definition of symmetric key ciphers. The difference between encryption and decryption is only the direction in which it starts. For encryption, plaintext is broken into blocks that go through permutations. The first permutation is called the initial permutation, or IP. After the initial permutation, the block is divided into left and right half blocks, where the left half is the leftmost half of the bits and the right half is the remaining bits. Each round of encryption is done by swapping the blocks and then performing an **XOR** operation between the new right block and the previous right block being transformed by a Feistel function. After the last round, the left and right blocks are swapped, so the decryption algorithm uses the same structure. After the last round of encryption, the blocks are combined and then permutated using the inverse permutation. DES is included in our suite of supported ciphers for message encryption.

Another cipher included in the suite of accepted ciphers is homomorphic RSA. To perform RSA, first choose two large primes **p** and **q**. Using these two chosen primes, calculate **n = p \* q**, and then choose an **e** such that the greatest common divisor of **e** and **Y(n)** is 1, where **Y(n) = (p – 1) \* (q – 1)**. Afterwards, find some number **d** such that **d** is the modular inverse of **e**. To calculate the modular inverse, set **d** to be congruent to **e-1 mod Y(n)**. The public key is **e** and the private key is **d**. Using these keys, messages can be encrypted and decrypted using the following functions:

**c = me mod n**  Encrypts message **m** to ciphertext **c**

**m = cd mod n** Decrypts ciphertext **c** to message **m**

This encryption function is homomorphic, which means it has a very convenient set of properties, however it also contains some significant drawbacks. One such drawback is malleability, which means that given a ciphertext **c** that encrypts a message **m,** an adversary can create a second ciphertext **c’** and use it to compute the secret function and make predictable changes to ciphertexts.

Blum-Goldwasser is a semantically-secure encryption algorithm that is based on the difficulty of integer factorization. Blum-Goldwasser generates keys based on Blum integers, which are generated in the same way as in RSA, except that the prime factors **p** and **q** must be congruent to 3 mod 4, and do not equal each other. The public key is generated by multiplying **p** and **q** together. The private key is the factorization of the public key into **p** and **q**. In order to encrypt and decrypt messages between two parties, the person who calculated the public key must give it to the person sending messages. The actual encryption for messages is done by encoding a message as a string of **L** bits. After encoding the messages as a string of **L** bits, selecting a random element **r**, where 1 < **r** < **p\*q**, and using **r** computes **x0 = r2 mod (p\*q)**. Then using a pseudo-random number generator to generate **L** random bits, and then computes a random ciphertext keystream **b**. To get the ciphertext bits from here, loop an **XOR** operation **L** times between the plaintext bits and the new keystream **b**. In order to decrypt the message, take the last pseudo-random number and using prime factorization compute the values **rq** and **rp**, and then use those numbers to compute the initial seed **x0**. From **x0** recompute the bit-vector **b** using the Blum Blum Shub generator like in the encryption algorithm. Compute the plaintext by applying the **XOR** operation on the keystream with the ciphertext. After this the message is recovered and decrypted. Blum-Goldwasser is used in our implementation as a part of the cipher suite for encrypting messages being sent to and from the server.

For the project we were required to implement hash message authentication code as a message authentication code, while also implementing SHA1 to use as the hashing function for HMAC. SHA1 is done by taking in a message, and transforming it block by block by using round functions of bitwise operations to create a hash. Each round function is called 20 times and used to create a 20 bit hexadecimal message digest. Using this hash function, HMAC takes 2 strings, a key and a message, and then uses a trapdoor function to hash the message into a message authentication code. HMAC creates this trapdoor by taking the key and reformatting it to be the correct length, and then returning the result of this hash: H((opad **XOR** key) || H((ipad **XOR** key) || message)). This enables the use of digital signatures for the server and clients connecting to each other.

This project was implemented as a chat server all built in c++ for the algorithms and server code. The cipher suite has a total of three encryption algorithms, DES, Blum-Goldwasser, and textbook RSA. HMAC was used as a digital signature and authentication algorithm. Keys in SSL were exchanged via Diffie-Hellman and the SSL process was implemented to a create a secure channel for communication. There are some weaknesses with this design, such as the limited set of ciphers in the cipher suite, along with a relatively weak pseudo random number generator