Cavell Teng

Cryptography and Network Security I - White Hat Writeup

The goal of the white hat project is the implementation of a secure communication suite where a several users can connect to a server. To secure this communication channel, SSH/SSL (Secure Shell Host/ Secure Socket Layer) protocol is applied. This protocol utilizes the encryption suite composed of textbook RSA, SDES, and the Blum-Goldwasser encryption algorithm. The protocol starts when the client initiates communications with the server. The server responds with a similar greeting. Following this, the client sends the server their available encryption suite. The server chooses an encryption method available to both sides and informs the client of the decision. At this point, they collectively decide on a shared secret for the encryption algorithm. Once secure communication is obtained, the server can verify the authenticity of the client through a quick password check before open communication can begin.

SSL is a standard security protocol that establishes encrypted links between a web server and a client. The important part of the SSL is the handshake protocol. For our system the server is initialized at the host's ip address and at port 8080. The server can listen for new clients after the current ones disconnect. For our clients they send the server a greetings message, in our case "start" is sent. If the server does not receive this expected message, it will close the connection and wait for another client. Upon obtaining the correct initial message, the server will respond in kind and begin the negotiation.

The key exchange protocol that we used is the Diffie-Hellman key exchange. This implementation is a modified version of a previous homework assignment's code. The idea is that our client and server have an agreed upon modulus p and a base g, which is a primitive root modulo of p. The client chooses a secret integer a and sends the server a value which is g^a mod p. The server simultaneously chooses a secret integer b, and sends the client g^b mod p. The client and server both then exponentiate the value received with their private key to generate the shared secret.

For our MAC we used HMAC. This was one of the given parameters for the project. HMAC requires a secret key to operate. In out case, we used the secret key that was generated from the key exchange. The process of HMAC involves padding a key, concatenating it with the plain text before hashing it to get a checksum. In this case, the hash function is SHA-1. The resulting checksum is concatenated with the padded key and is hashed one more time to obtain the final checksum for the plain text. Ideally, this process resolves some of the weaknesses found in the SHA-1 hash function that we are required to use.

The version of RSA we implemented is the textbook RSA, which is homomorphic. The property of this homomorphism is multiplication. This means the multiplication applied to the cipher text has an equivalent affect on the resulting plain text. RSA is based upon the asymmetry found in factoring the product of two large distinct prime numbers. The way it works is that it generates two prime numbers, p and q. These two numbers must be distinct. We then multiply p and q together to get the first part of the public key, usually denoted by n. The other part of the public key is the least common multiple of (p-1) and (q-1) denoted by *phi*. In this case, *phi* is the

totient function. We then calculate our encryption value e, such that $1 < e < phi$. At this point the server calculate the secret key, given by the variable d. Once this is achieved, n and e are released to the clients. The clients can now securely send messages that only the server can decrypt. While this means that communication is one way, it is more than sufficient for the purposed of this project.

SDES (Simplified Data Encryption Standard) was the weakened system that we implemented for a previous homework assignment. We just modified it to make sure it worked with our security suite. SDES is a fiestel block cipher; it takes a fix-length string of plaintext bits and transforms it through a series of complicated operations into another cipher text bit string of the same length.The way SDES works is that it takes in a 10 bit key, permuted based upon position, and then broken down into 2 5 bit keys. Through a combination of permutations and left shifts, the two keys are used to generate the two sub keys. The sub keys are XOR'd in the main encryption step with the permutated plain text. The final result is the concatenation of the most recent left and right halves of the encryption. Ideally, this fiestel process is normally repeated multiple types and utilizes are larger number of s boxes. For the sake of time and simplicity, this was removed.

The final encryption system that we implemented was the Blum-Goldwasser. It is a semantically secure cryptosystem. It is considered semantically secure because of the assumed intractability of integer factorization, or rather the difficulty of factoring a N value where N = p* q and p,q are large prime numbers. First the Server will generate two large primes p and q such that they are unique numbers, and p,q are congruent to 3 mod 4. The public key N is p*q, and the

private key for this system is the factorization of (p,q). The Server will then send the client N, while keeping the private key a secret. Then the client will encode their message as a string of L sized bits and selects a random element r, where 1 < r < N, and computes xi which is equal to the quadratic residual. The client will then use the Blum Blum Shub pseudo random number generator to generate the key stream where, bi is equal to the least significant bit of xi, i is incremented, and then the new xi value is calculated by (xi-1)^2 mod N. The client then computes the cipher text bits by XORing the plaintext bits with the bits for the keystream, for each section they will XoR m with bi and then each segment comprises the final ciphertext. The client will the send the encrypted message to the server, along with the final XL value.

For the server to decrypt it will receive the message packet containing the ciphertext broken into L-1 segments and IL. using the prime factorization of (p,q) the secret key the server computes r1 = y((p+1)/4)^Lmod p and r2 = y((q+1_/4)^L mod q. The initial seed is then calculated for by the equation: (q(q-1mod p)rp + p(p-1mod q)rq) mod N. Then from the initial seed the server can compute the bit-vector b using the BBS, generator as its encryption algorithm. Final each plaintext segment is recovered by XoRing each part of C with it's b value. It should be noted that due to time constraints and an inability to calculate the required values of *a* and *b,* the values used in encryption and decryption of the of the plain text are hard coded.

Overall, it is necessary to note that there are several issues with the server implemented. Due to some design issues, the communication was encrypted only in one direction. The client can safely send messages to the server, but the server cannot. This necessity was discovered when we attempted to properly implement RSA and Blum Goldwasser with the socket

programming. The designs of these algorithms were to allow for asynchronous encryption. While it is technically possible to simply apply the initial setup to both the server and the client, due to time constraints that was not implemented.