The purpose of the project was to construct a basic communication suite where a user could communicate with a server over insecure channels. To ensure that the messages are secure we use a variety encryption suites that we studied in class. For basic communication protocol we ended up using SSL protocol. For the encryption suite we implemented textbook RSA, and SDES, and the Blum-Goldwasser encryption algorithm. The overview of the system is the client reaches out to the server and collectively they decide upon which encryption algorithms they will use, as well as their secret key for that session. Once that is complete the user can now communicate with the server using the encryption scheme of their choice; communicating on unsecure channels in a more secure manner.

SSL, better known as Secure Sockets Layer, 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 we start the server and leaving it running, listening for potential clients on Port 8080. For our clients they send the server a greetings message, In our case the client sends the string “start” to kick off the process. If the first message isn’t start then the port will be closed, and exited. The server will send back a message confirming that the connection has been received, and will then determine the which encryption systems will be used to encrypt the messages between the client and the server.

For our key exchange protocol we used the Diffie-Hellman key exchange. We chose Diffie-Hellman because we had previous experience implementing it on past homework assignments. The premise 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 that is relatively small 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 B = g^b mod p. The client and server both then decrypt the value to create their shared secret, Client does S = B^a mod p, and the server does S = A^b mod p. Now the Client and Server have a shared secret.

For our MAC we used HMAC. We made this design choice because it was one of the parameters that was laid out in the project specifications. HMAC requires a secret key to operate. The way it works is that you pass it a key, a message to be hashed, the block size which is derived from the hash function we used, in the case it’s 64 bits, and the output size, which in this case is 20 bits long. If a key is longer than the required block size they are shorted by hashing them, which brings to be of length equal to output size. If the key is shorter than block size they are padded to the length of block size. The outer padded key is then computed by XoRing the key, and the same thing occurs with the inner padded key. Finally the value is returned where the has of the outer padded key is concatenated to the has of the inner padded key which is then concatenated to the message.

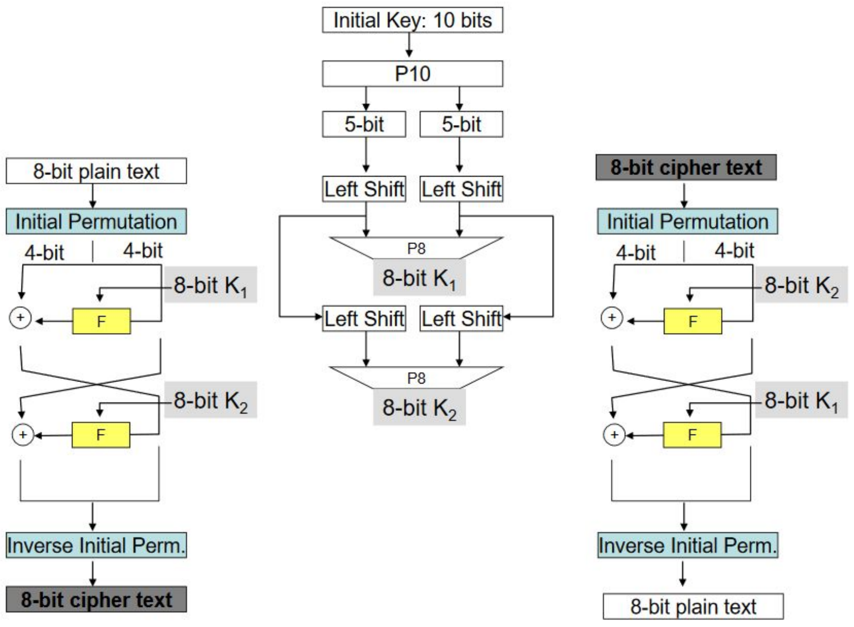
For our hash function we used SHA-1 which was specified in the project outline. SHA-1 works by: first it will create 5 variables H0 to H4. We then convert each character of our plaintext to Ascii, and then into binary. We then concatenate them into one string and add 1 to the end. We then pad it to be of the appropriate length. We then extend the sixteen 32-bit words into eighty 32-bit words. We then perform a circular shift operation. The operation can be noted by S^n (X), on the word X, by n bits, where n is an integer between 0 and 32. X is either left shifted which is done by discarding the left-most n bits of X and padding the remaining values with n zeroes on the right hand side. Otherwise X is right-shifted by discarding the right-most n bits of X and padd the left with n zeroes. Thus S^n(X) represents a circular shift of X by n positions. We then store the has values defined in the first step in 5 variables. Then for 80 iterations, we compute a temp value such that S^5 * (A) + f(i;B,C,D) + E + W(i) + K(i). we then re-assign the variables such that: E = D, D = C, C = S^{30}(B), B=A, A = TEMP. We then store the result of that chunk's hash in the overall has value of all chunks and proceed to execute the next chunk. Finally when all the chunks have been processed, the message digest is represented as the 160-bit string comprised of the 5 hash values.

The version of RSA we implemented is homomorphic in nature, which means allows for computation on ciphertexts, which when decrypted they will match the results of the operations as if they had been performed on the plaintext. RSA is an older encryption system, and has been deprecated. 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. We then calculate our value e, such that 1 < e < phi, and e must be co-prime to phi. At this point the client just needs to calculate its secret key, given by the variable d. d is just calculated using a function that determines the modulo inverse of e and phi. At this point the client will then take their message in plaintext form, convert each letter into its ASCII value, and then encrypt these values using the form m^e mod n. This value is then concatenated to form the final encrypted message. The user will then pass the server the encrypted message, the n value, and the e value. The server is then able to decrypt the message by applying the secret key d to the equation c^d mod n. The client will then loop through and convert each value from ascii into its character value, then create the original message in plaintext form. Sample runs of RSA can be found in figure 2.

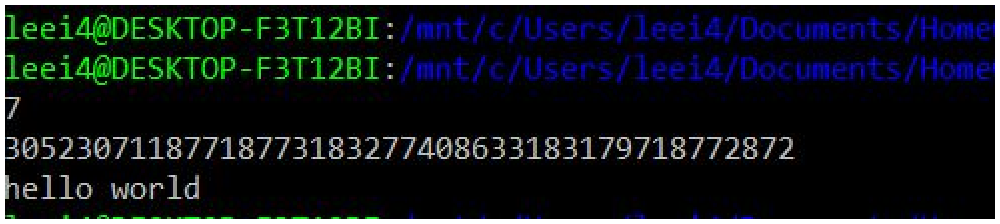
Des was the weakened system that implemented for a previous homework assignment. We just modified it to make sure it worked with our security suite. DES is an example of a block cipher, which means that it takes a fix-length string of plaintext bits and transforms it through a series of complicated operations into another ciphertext bitstring of the same length.The way DES works is that it takes in a 10 bit key, permuted based upon position, and then broken down into 2 5 bit keys, which are then left shifted and then combined and permuted to form our first 8 bit key denoted by K1. The right side of the 5 bits left shifts are then left shifted again and combined to form another 8 bit key noted as K2. It then takes in our plaintext and breaks it into 8 bit chunks and permutes it based upon positions. It then separates the permuted sequence into 2 4 bit chunks, the right chunk is placed into an F box where it is modified using through a process of permutation and XoR using K1. This 4 bit sequence is then XoR'ed with the left 4 bit chunk. The first right chunk is then XoR'ed with the resultant 4 bit chunk that is pushed through an F box with K2. These 2 4 bits are then combined and go through one more permutation, to form the cipher text in 8 bit chunks. To decode this you run it through the same system, except backwards and it will return your 8 bit plain text sequence (fig 1).

The final encryption system that we used was 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 r^2 mod N. 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 X_L value.

For the server to decrypt it will receive the message packet containing the ciphertext broken into L-1 segments and I_L. using the prime factorization of (p,q) the secret key the server computes $r_1 = y^{((p+1)/4)^\wedge L}$mod p and $r_2 = y^{((q+1\_/4)^\wedge L}$ mod q. The initial seed is then calculated for by the equation: $(q(q^{-1}\text{mod } p)r_p + p(p^{-1}\text{mod } q)r_q) \bmod N$. 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.



(Fig 1.)



```
leei4@DESKTOP-F3T12BI:/mnt/c/Users/leei4/Documents/Home
leei4@DESKTOP-F3T12BI:/mnt/c/Users/leei4/Documents/Home
7
30523071187718773183277408633183179718772872
hello world
```

(Fig 2.)