Cryptography Project

Billy Culver, Josh Barthelmess

12/03/2018

1 Overview

We implemented our SSL in C++. we chose this language since it was a language we were both comfortable working in, using sockets in, and had done the previous homeworks in. To avoid errors with different compilers we didn't use ints but int32_t, int64_t, and there unsigned versions as well. We Implemented RSA, Semantically Secure RSA, DES, and Diffie-Helmann crypto-systems. We set up a server which was assumed to be secure, and allowed clients to connect to the server via a TCP connection. These clients could then talk to each other by passing messages through the server. We assumed that our adversaries could see any message that was being sent or received by the server, could do anything that a regular client could do, but couldn't access information that only the server had access to.

Billy's portion of this project was largely implementing the different encryption algorithms that we used, as well as some other helper functions. These included RSA, semantically-secure RSA, DES, SHA-1, and Diffie-Hellman. Josh's portion was implementing the SSL, the server, the user class, the clients code and the necessary socketing. We both worked on the writeup, and debugging the different parts.

2 BILLY

2.1 Diffie-Hellman

We didn't actually need to do much for Diffie-Hellman since we had already previously written it in a previous homework. We mainly used Diffie-Hellman to exchange keys for DES. It also made many other sections easier since in writing it we had also written a function that generated random large prime numbers. This was helpful for RSA.

2.2 SHA-1

To make SHA-1 I followed sudo-code found online of the program. SHA-1 returns an 160 bit hash for any given string. The first step is to pad the string with null characters(0 bits) to get the length of the message to be 56 mod 64 bytes long(448 mod 512 bits). We then append the length of the original message onto the end as an 8-byte number. This brings the length of the total message up to 64-bytes (512 bits). for each chunk of 64-bytes, you then divide these 64 bytes into 16 4-byte chunks (16 32 bit integers). Next you turn these 16 integers, into 80 integers. This is done by performing XORs of previous integers to get the new chunks.

$$Chunk[i] = Chunk[i-3] \oplus Chunk[i-8] \oplus Chunk[i-14] \oplus Chunk[i-16]$$

for each new chunk you then perform a left-rotate of the bits where you shift the bits left, and move the furthest left bit to the front. We already had a function for this from a previous homework.

Now we start the actual hashing. Five values, a through e, initialized to certain values which we found online. For each of these 80 chunks, we perform the following operations. For each chunk you perform a bit operation on a through e, to get a new value f. The first 20 chunks perform one bit operation, the next 20 a different operation, and similarly for the last two sets of 20. For each set of 20 chunks you also create a variable k, which is set to a specific value which is unique for each set of 20. You then set each of the original 5 values to be a new combination of the 7 values(a-f, and k). after you've done this for each of the 80 chunks you had the new values of a,b,c,d,e to there original values, and repeat the process for each of the 512 bit parts.

2.2.1 SAMPLE RUN

```
$ ./SHA_test.exe
MESSAGE 1: HELLO
5 32-bit Hash Values for Message 1
2732008526
4035617762
2562383102
2419217525
64152047

MESSAGE 2: HELLO
5 32-bit Hash Values for Message 2
2732008526
4035617762
2562383102
2419217525
64152047

MESSAGE 3: HELLO
5 32-bit Hash Values for Message 3
1375932984
2378061196
3636124925
271733878
3285377519
```

This example shows that passing the same message in returns you the same hash value. but passing in even just a slightly different message gets you a very different hash. In order for 2 hashes to be considered the same all 5 of these 32 bit values would have to be the same. SHA-1 is no longer considered secure.

2.3 RSA

To make RSA, I first had to write a function to generate (public, private, n) key sets. The first step of this was to generate two large prime numbers p and q. We had previously written a function to generate large prime numbers, so this wasn't difficult. Multiplying these together got us our n. To find our public and private keys, we calculated

$$\phi = (p-1)(q-1)$$

We then chose random values (public-key) between 0 and phi, and tested whether or not they were co-prime with phi. To do this, we did used Euclid's algorithm to find their greatest common divisor. If their GCD(e,ϕ)=1, then we had a coprime. We then used the extended euclid algorithm to determine what the modular exponential inverse of (public-key) was. This inverse became our private-key. Now that we had the keys, we wrote a function to encrypt a message using RSA, and a function to decrypt encrypted messages. These functions were both just wrapper functions that called our modular exponentiation function that we had written on a previous homework assignment.

2.3.1 SAMPLE RUN

```
$ ./RSA_test.exe
N:           374185807
Public Key:  92148041
Private Key: 359891321

ORIGINAL MESSAGE:  123456789
ENCRYPTED MESSAGE: 342630193
DECRYPTED MESSAGE: 123456789
```

This RSA implementation can only handle messages that are lower then N. Because our implementation of modular exponentiation, N cannot be greater then 32 bits. While RSA as a cryptosystem is still considered secure, our 32 bit limit makes it a pretty weak cryptosystem, that can be broken.

2.4 Semantically-Secure RSA

Semantically-secure RSA was pretty simple once we had SHA-1 and regular RSA working. First we created a random variable, and encrypted that variable using standard RSA. We then hashed that random variable using SHA-1, and performed an XOR of the hash value, and the message we wanted to encrypt. we then sent the encrypted random variable, and the result of the XOR. To decrypt this, they decrypted the random variable, found the hash of that random variable, and then performed an XOR of the resulting has and the encrypted second part of the message. This returns the original message.

2.4.1 SAMPLE RUN

```
$ ./RSA_test.exe
N:           884577523
Public Key:  212653679
Private Key: 107285167

ORIGINAL MESSAGE:  123456789
ENCRYPTED MESSAGE: 322013608 165821796368899838 21
DECRYPTED MESSAGE: 123456789
```

This semantically secure RSA suffers from the same issue as the regular RSA. The message can't be larger then N, and the N can't be greater then 32-bits. This is due to the fact that when we perform modular exponentiation we square the base. the base can be nearly the same size as n. We also don't use then entire SHA-1 Hash value, but only the first 64 bits of it. This makes the hash less secure then the original SHA-1.

2.5 DES

When We had originally written the toy-DES, we had created many helper functions that were very useful in implementing the full DES. The most useful was a function permute which took a block of bits, and

rearranged them according to the order of an array that was also passed into the function. This made the many different permutations implemented in DES as simple as creating an array of the new order of bits and passing it to this function. DES starts with an initial permutation of the 64-bit plain text. This then gets split into two parts, left and right, with right being the lowest 32 bits, and left being the larger 32 bits.

Next we generate 16 48-bit subkeys based on the original 64 bit key. The original key goes through a permutation which lowers it from 64 bits to 56 bits. It then gets split in 2 similar to how the plain text was using bit-shifts and bit-masks. We then perform a single leftrotate or a double leftrotate on the left and right halfs of the key depending on which key number it was. Next we combine the two keys back together, and perform a permutation on this new value that changes it from a 56 bit value to a 48 bit value. This 48 bit value is one of our 16 keys. we repeat this to get the 16 different keys. Now that we have the keys, we can begin the actual feistel cipher portion of the DES. This portion is 16 rounds where in each round the right half of the message(which we got earlier), and the rounds subkey are passed into a function f-box which outputs a 32 bit value. this value is XORed with the left portion. This XOR value becomes the new left half for the next round and the previous left half becomes next rounds right half. This is repeated 16 times. We then recombine them and perform an inverse of the original permutation on this new 64 bit message. This result is the cipher text.

In the f-box function we define an array called s-boxes with dimensions 8x4x16. That is to say that there are 8 4x16 arrays in it that are our 8 s-boxes. the values in these s-boxes are 4 bit values. We pass into the f-box a 32-bit value(the right side in the feistel cipher), and a 48 bit subkey. We then run the right value through a permutation which changes it to a 48 bit value. Next we XOR these 2 48-bit values to get a new value. This resulting 48 bit value is then split into 8 6-bit chunks using bit shifts and bit masks. for each of these chunks, we then split it again into a 2 bit value (outer) which is the value of the two outer bits of the 6 bit value, and a 4 bit value(inner) which is the middle 4 bits. We look up the new 4-bit value in the corresponding location in the S-box array(S-box[chunk number][outer][inner]). Now that we have 8 4-bit values, we combine them to form a 32 bit value. this is the output of the f-box.

To decrypt DES, you do the exact same procedure, except you use the subkeys in reverse order.

2.5.1 SAMPLE RUN

```
$ ./DES_test.exe
ORIGINAL MESSAGE:  1234567890987654321
ENCRYPTED MESSAGE: 16133109034110724291
DECRYPTED MESSAGE: 1234567890987654321
```

This DES implementation can encrypt and decrypt 64 bit values. Anything more then that and it needs to be broken into pieces. anything less and it needs to be padded. DES is no longer considered secure. So

this can in theory be broken.

3 JOSH

There are two main entities for this implementation: the server, and the clients. Both sides have access to all of the different tools and algorithms talked about above, but the client can pick which ones it prefers, as well as which it does not want to use.

3.1 Server

The server uses a multithreaded approach to handling different users who connect to it, with each user being given a dedicated listening thread. It maintains a hashmap of the different users that have connected. these user objects contain that users username, username+password hash,encryption capabilities and preferences, along with socket contact information. It uses this info to facilitate interthread and asynchronous communications. Access to the master user list is locked behind a mutex, to prevent race conditions from occuring. The server's most important job is to mediate messages between clients, and to do this securely. To accomplish a high level of security, the client is expected to encrypt his/her message such that only the recieving client could decrypt it. This way, even if the server is compromised, people monitoring comm's between users would be unable to view it unencrypted.

3.2 Client

Each Client chooses a unique username and password. They are responsible for remembering their login information, as there is no way to retrieve the information once set. Their Username will be saved on the server but their password will not. On start-up the user is expected to perform a handshake with server to establish a set of RSA keys and a symmetric key to be used for DES. If it is a returning user, these new values overwrite their previous values. This is both more secure and easier for the user since the user doesn't have to store there key information locally if they are logged out. After the handshake is complete, users should login, or create login information. Our client will prompt the user to do this, but if custom clients attempt to do anything other than login before having logged in, they will be immediately disconnected. Once Logged in, there are seven different commands that the user has access to.

3.2.1 User Commands

These commands can only be used if the User is logged into the server.

- WHO: This command lets the user know who else is online.
- CONNECT[username]: This establishes a secure channel to another user.
- DISCONNECT[username]: Disconnects from specified user.
- MSG [username] [msg]: This sends a message to the specified person. The user must have already connected to that user.
- LOGOFF: This Logs the user off. there Information will be kept until it expires.
- SET [encrpytion option]: Sets new prefered encryption option default will be DES.
- DISABLE [encryption option]: This Disables specified encryption algorithm so it can no longer be used by this user.
- HELP: Lists these commands.

3.3 Initial Handshake

The Initial handshake is unencrypted and forms the connection between the server and the client. During this handshake they exchange RSA keys so that the server and this client can communicate using RSA, and also they use Diffie-Hellman to create a shared DES key. Our handshake is a bit strict in the beginning in that it requires both RSA options and DES and Diffie-Helman to be used, but once the connection is complete, the connection becomes more flexible as the user can customize what forms of encryption they want to support and use.

3.4 LOGIN

Users will be asked to login. This allows a certain verification to be established once login credentials have been created. First time users will need to set up a username and password. The username can't already be in use in the server. To prevent the exposure of the users personnal password, the client hashes the username and password combination together, and uses that as a proxy password. Attempting to use an already in use username will result in the same behavior as an incorrect password, and you'll be asked to pick another one. Returning users simply enter their username and password, and the server then verifies them as correct, or incorrect. Once the Login has been verified, the client is expected to listen for the server to open a dedicated socket for message passing from other users. This is socket is also used to prevent race conditions and allows messages to come to the user asynchronously.

3.5 SET and DISABLE

These two commands are the staple of our design, and what makes our cryptographic chat unique. Currently there are three supported encryption types on offer: DES encryption, RSA encryption, and a Semantically Secure RSA. By default, DES is used for communications because it is our strongest encryption type, mathematically. However, users are able to customize their choice of encryption method with the SET and DISABLE. The SET command changes the top priority of the users encryption set. This will make it more likely that it will be used to communicate with other users, and the server will automatically use whatever the top preference of the user happens to be. The DISABLE command prevents the user from being communicated to with the given encryption scheme. As an example, if you have disabled DES and RSA, and another user has disabled SEM, then these two users will be unable to connect due to incompatible encryption preferences. You can restart using an encryption method using the SET command. If someone tries to DISABLE all encryption schemes, the last scheme will be unable to be disabled by the user. It should be noted that these preferences need to be reset everytime the user logs into the server.

3.6 CONNECT and DISCONNECT

These commands act as secondary handshakes between users, that are facilitated by the server. CONNECT allows users to establish a cryptographically secure channel between them. The server looks at both users preferences, and decides the best encryption scheme to use. Once it does that, it facilitates both parties creation of keys, which then allows secret messages to be passed between the two users. DISCONNECT does exactly what you would think it does. Once one user decides they are done talking, DISCONNECT disallows communication between the two users until another CONNECT command and handshake is completed. This allows users to leave a conversation that they don't want to be a part of.

4 How to Run

The first step to run this code is to compile both the server file, and the client file using the Makefile. Next run the server file "./server.exe". That will output a port number that the clients will need in order to connect to the server.If you prefer you can add a command line argument for which port number you want to use. The clients can then connect by running "./client.exe IP-Address Port" where PORT is the port that the server is listening on.