CS 165 Assignment 2 Report

Noel Llanura

861020193

The compiler used for this program is gcc version 4.1.2 and 4.9.2. The OpenSSL version is 0.9.8 for the linux_x86_64 platform.

Server Details:

To compile the server you have several options:

1) make (you may edit the Makefile to provide different arguments)
2) gcc –o server server.c –lssl –lcrypto –ldl
    a. Ex. gcc –o server server.c –lssl –lcrypto -ldl
3) gcc –o server server.c –L(path to openssl directory) –lssl –lcrypto –ldl (if openssl is installed in a non-default location)
    a. Ex. gcc –o server server.c –L./openssl –lssl –lcrypto –ldl

To run the server you have several options:

1) make runs (you may edit the Makefile to provide different arguments)
2) ./server --port=(port number)
    a. ./server --port=20193

Client Details:

To compile the client you have several options:

1) make (you may edit the Makefile to provide different arguments)
2) gcc –o client client.c –lssl –lcrypto –ldl
    a. Ex. gcc –o client client.c –lssl –lcrypto –ldl
3) gcc –o client client.c –L(path to openssl directory) –lssl –lcrypto –ldl (if openssl is installed in a non-default location)
    a. Ex. gcc –o client client.c –L./openssl –lssl –lcrypto –ldl

To run the client you have several options:

1) make runc (you may edit the Makefile to provide different arguments)
2) ./client --serverAddress=(host server is on) --port=(port server is listening on) –(send or receive) ./file (where this can be replaced by any file with the correct path)
    a. ./client --serverAddress=localhost --port=20193 --send ./helloworld.txt

Other options available in the Makefile:

1) make setup
    a. This will set up the directories the client and server use to store files they receive. Use this so that when they receive the file, they actually store it somewhere. Otherwise, they won't receive any files.
2) make clean

a. This will clean up the executables in the current working directory.

Server Design:

The server is designed with minimal work on parsing the arguments. It is a very brute force method of taking in arguments so the user has to be careful in inputting the port number and follow the format strictly.

After the server parses the arguments and gets the port number, it will read in the private key pem file from the directory the server is located in and store it in a char array for later use. The server then initializes the ssl library, an ssl context, generates the diffie helman parameters for use in the connection encryption, creates a bio and assigns the port to it, and then starts accepting connections. Once a connection has been accepted, the server will read from the client an encrypted value, decrypt it using its private key, hash it using the SHA-1 hashing method, and re-encrypt it using its private key. The server then writes the encrypted hash value back to the client and waits for a response. The response it gets should be a flag as to what the client wants to do with the server. If the client wants to send a file to the server, the server will wait for the file name, create the file in a specific directory, and wait for the contents of the file which it then writes to the newly created file. After this process, the server will immediately shut down. If the client wants to receive a file, the server will wait for the file name, search for the file, read in the contents to a char array, and send it over to the client in chunks.

Some specific things to note about the file creation process… The file is transferred over the socket in chunks of size 256 bytes. The limit to the size of the file to be transferred is 4096 bytes. This can be changed if you go through the code and change the chunkSize and limit in both the server and client.

Client Design:

The client is also designed with minimal work on parsing arguments. It is done in a brute force method so the formatting of the arguments has to be done very accurately. Follow the format provided in the Makefile.

After the client parses the arguments, it will read in the public key pem file and store it in a char array. The client will then generate a pseudo random number using openssl's prng and hash it using SHA-1 for later use in authenticating the server.

The client will then encrypt the random number using the public key, initialize the ssl library, create the ssl context, set the cipher suite to use diffie helman, create the ssl, set up the bio, and then try to connect to the server. Once connected, the client will send over the encrypted random number and wait for the server's response. Once it gets the server's response, the client will decrypt the value using the public key and compare it the original hash value the client itself created of its initial random number.

If the hashed values match, this means the server is indeed the server the client wanted to talk to. The client then sends the flag corresponding to what the user entered in as an option. If the client wants to send a file, it will tell the server it is sending a file and send over the name of the file so the server can create it. The client will then send over the contents of the file in chunks of size 256. If the client wants to receive a file, it will tell the server it wants to receive and send over the name of the file it wants to receive. The client will then wait for the contents of the file to be sent over in chunks so that it could be

concatenated onto the file the client will create in its specific directory. Once it's finished receiving or sending, it will shut down the ssl.

Difficulties:

The documentation was very lacking. It didn't explain much about how to use different functions and how they worked. The descriptions were either very brief or non-existent. It was also lacking in explanation about how to involve the Diffie Helmen cipher suite in the ssl. There were also problems compiling because I initially installed it on my own remote server following OpenSSL's instructions for linux operating systems. The installation however, caused some issues in that the ciphers didn't seem available to the program. I'm not sure whether it was because I installed a later version than what was available on bell (the cs servers). I had to clone my files into bell because of this and start working from there. There was also an issue with the flags needed for compiling because I've never used –L and –l for linking different libraries to the compilation before. The other difficulty is with the encryption because of the padding. Encryption doesn't work 100% of the time. I think that this is because the random number generator create a null character sometimes in the middle of the string to encrypt. This might mean that the encryption thinks the string is shorter than it really is and the padding passed in doesn't suit it which causes an error. Therefore, when the program fails to authenticate, just send an interrupt signal to the client and restart the connection starting with the server. If there is a binding issue because of a port that is still being listened on, just change the port in the makefile or the arguments you create yourself.