Jordan Meyer
jmeye006
861015184

SSL Project Design Document

Server

The server is follows a basic setup for SSL connections. First, the parameter for the port number is used, then the SSL-specific information is setup, like the BIO/SSL/CTX variables and the Diffie-Hellman protocol setup is done, as recommended by Zachary. The server is specifically setup to not send any certificate to any connecting client. This is to allow the RSA key verification.

The RSA verification process begins when the client successfully connects to the server, and prompts the server with a random challenge for authorization. The server uses its own private key (rsapriv.pem) to work through the challenge. The server reads in the challenge from the client, and uses the private key to decrypt the challenge using RSA_private_decrypt. The decrypted challenge is hashed using SHA-1, as with the client-side hash. The hashed decryption is then encrypted using the RSA_private_encrypt function, and is the signed hash. The server sends this to the client, and if the client's challenge doesn't match the decrypted hash from the server, the client kills the connection.

When the client wants to send a file to the server, the server receives the file name and size before it receives the file's contents to create or rewrite the local file on the server. The opposite happens when the client receives a file from the server, so the server sends the client the file size and the contents so the client can create or rewrite the defined file.

Client

The server also follows basic SSL connection setup. As soon as it connects to the server, it prompts the server with a random challenge (using SHA-1 hash function on a pseudo-random byte string). The challenge is encrypted using the RSA public key (rsapub.pem) and the library function RSA_public_encrypt.

The client sends this encrypted byte array to the server for authentication. When the server responds, the client reads in the server's encrypted response, which is decrypted using the library function RSA_public_decrypt. This decrypted response is checked against the hash from the SHA-1 function previously mentioned. If the decrypted server response matches the SHA-1 hash the client used for the encryption, then the server is authenticated, and file transfer can begin.

If the client wants to send a file to the server for storage, the client reads the file into a byte string. The client sends the server a message that it wants to store a file, then the size of the file (in bytes), and finally, the byte string of the file's contents.

If the client wants to receive a file from the server, the client opens or creates the desired file, receives the file size from the server, then the file's contents from the server, and writes the buffered contents into the local file, and saves the file.

Notes

The difficulties when working through this server were finding helpful documentation, and debugging some smaller things due to my hiatus from network programming and C coding. Also, determining which functions needed to be used to set up the connection, and which were necessary to set up the Diffie-Hellman protocol system was time consuming.

Writing the skeleton code was annoying, due to all the error checks that were necessary and the Diffie-Hellman setup which was required to avoid using the SSL certificate system that is used as standard. The server took much more time to write, but that is mainly due to the extra setup for this specific authentication system.

Some small, but annoying errors came from using C. Some instantiations of variables are rather specific and cause errors when not done exactly how the compiler wants, even though they can be rewritten later. Also, formatting output was annoying, since printf had to be used in lieu of cout, which is friendlier and more familiar.

It was very rewarding to see the server and client interact quickly through SSL calls and authenticate using the RSA keys and SHA-1. It served as a nice eye-opener as to how powerful these ideas can be when they are implemented.

Upon further testing, I noticed my client/server interaction has some issues when sending/receiving large files (16KB< up through a few MB);

**Above issue may have been fixed by sending 16KB at a time until the file is completely sent, however, sometimes the server halts after sending a large file (3MB in my tests).