

Liam Donohoe

Cryptography

Prof. Yener

December 3, 2018

SSL Implementation

For this project, we decided on a custom implementation of SSL. With this came many choices and decisions for design and protocol implementations. Firstly and most importantly, we decided on using Python-3 for all implementations. This was primarily for simplicity of implementations, and familiarity of the group. For SSL, there were multiple stages that we had to decide upon. Firstly was the method of authentication, and key distribution. Diffie-Hellman was the obvious choice for this, as we already have an implementation of this from a previous assignment, as well as it being secure and robust. In terms of generating the keys themselves, we branched out into new, more secure algorithms. These include RSA and ECC. Finally, the most important part of the implementation, is the encryption algorithms we support. As a basis, we started with the algorithms we had implemented for previous assignments, SDES and Blum-Goldwasser. SDES is of course very weak, and susceptible to attacks, but instead of trying to reinforce that, we decided instead to add other more robust algorithms, to essentially make SDES redundant, and useless. On top of this we also implemented RC4 for a stronger encryption method.

The basic idea of SSL is essentially very simple. We first establish a connection, then negotiate and authenticate the channel, and are then able to securely communicate over this channel. Establishing the initial connection is done simply using basic socket libraries, connecting over TCP/IP, from a client to the server. Authentication and negotiation is then performed by the SSL Handshake Protocol. Below is a diagram showing an overview of this protocol. In this, we both authenticate the server and client to each other, as well as decide on specifics for their communication. An important note is that this handshake

chooses the strongest encryption algorithm which both parties support. This provides the strongest possible and most secure communication for this given session between the server and client. As can be

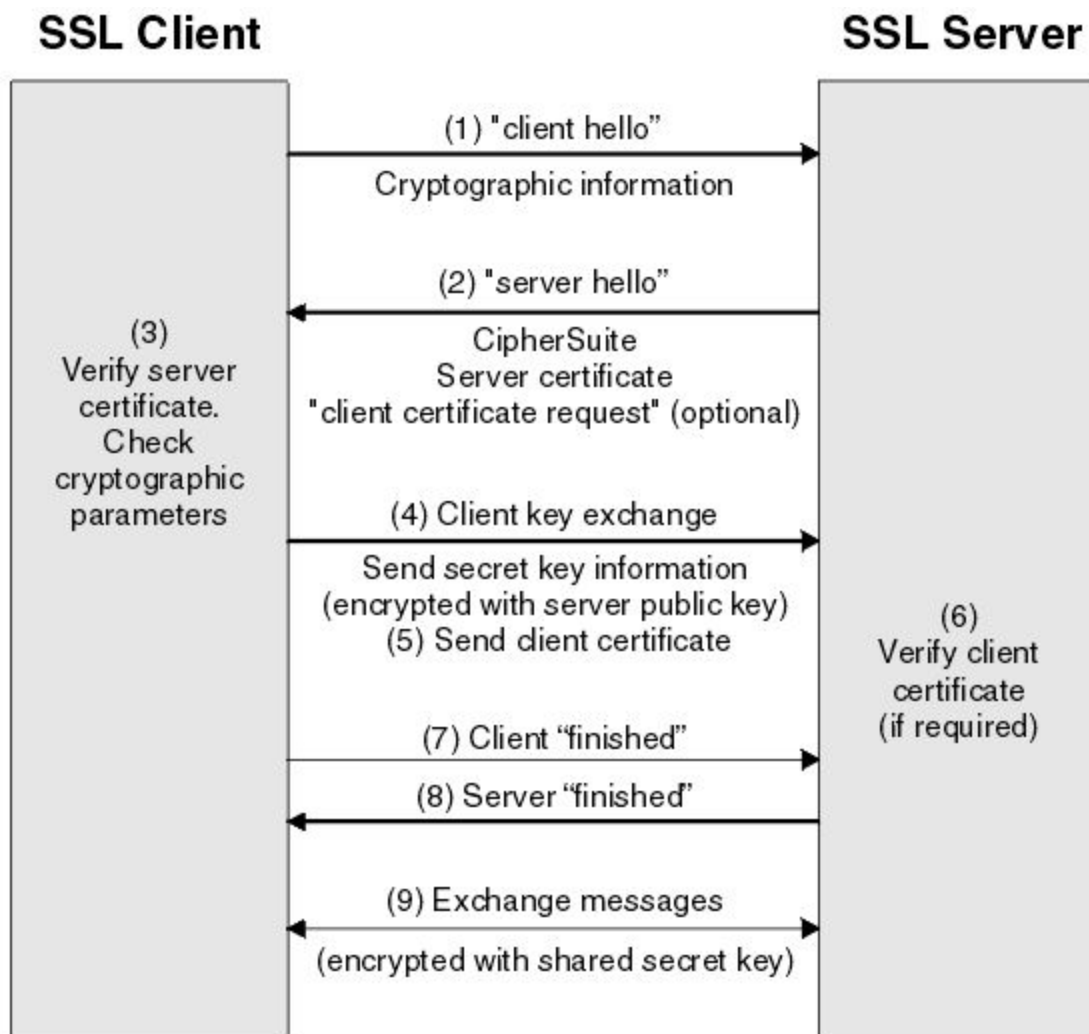


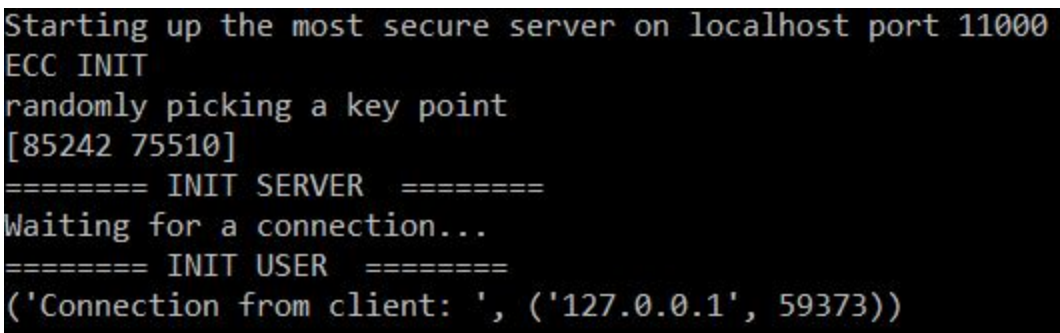
Figure 1: Overview of SSL Handshake Protocol

seen in figure 1 above, the handshake begins with the client sending a "Hello" message, providing the information on what algorithms they support. The server then decides which to use, and sends back its information regarding that encryption. In real world implementations of SSL, this step will also send certificates, to further authenticate the server, and also request the client's certificate. However, due to the complexity and difficulty of implementing this capability, we will not support nor request certificates

from the server or the client. Once an algorithm is chosen, keys are exchanged using Diffie-Hellman, and the chosen/strongest key generation method, and communication can begin.

To expand upon our encryption algorithm support, the strongest choice we have is RC4. This is a stream cipher, and is relatively secure compared to the other algorithms we have implemented. Though it is not perfectly secure, it is more secure than SDES, which is certainly a positive. SDES is by far the weakest algorithm we support in our system, but as there are many other stronger alternatives, we assume that the client and server will never decide on using this algorithm. The other algorithm we support is Blum-Goldwasser, which is also very secure to attacks. For each of these algorithms, any message sent is padded with HMAC before being sent. This ensures that the message will not be jumbled in any way, and will stay secure.

When our server is ran, you will be greeted with the following dialogue. As you can see in figure

A terminal window with a black background and green text. The text shows the server's startup sequence: 'Starting up the most secure server on localhost port 11000', 'ECC INIT', 'randomly picking a key point', '[85242 75510]', '==== INIT SERVER ====', 'Waiting for a connection...', '==== INIT USER ====', and '('Connection from client: ', ('127.0.0.1', 59373))'.

```
Starting up the most secure server on localhost port 11000
ECC INIT
randomly picking a key point
[85242 75510]
==== INIT SERVER ====
Waiting for a connection...
==== INIT USER ====
('Connection from client: ', ('127.0.0.1', 59373))
```

Figure 2 : Server initiation view

2, upon starting up the server, random key seeds are chosen, for use in either ECC or RSA key generation, depending on what is chosen. The server then waits for a client, and then acknowledges the connection when it is received. This then waits for a response from the client about their choice of algorithm for the key generation. Once the client chooses an option for key generation, the server proceeds with the key generation, and uses Diffie-Hellman protocol to exchange the keys. Once the keys are exchanged, an encryption algorithm is chosen, and communication can begin. Below is shown a view from the client's perspective, after having chosen their desired key generation scheme.

```
Welcome to Group 4 Secure Socket Client!
Input your authentication method:
0: ECC
1: DFH
2: RSA
  enter the number of your choice: 1
('Type: {' , 1, '}' )
sending 'DFH'
Recving prime from server
('Recived: ' , u'p806107p')
('Prime: ' , 806107)
('Shared key: ' , 690649L)
SHELL:
```

Figure 3 : Client chosen key view

Once the client and server have exchanged keys, the client essentially has a sort of shell, where they may a command in the form <Encrypton_Type> <Message> <Key>. The type of encryption is represented as a number, from 0-2, where 0 is Blum-Goldwasser, 1 is SDES, and 2 is RC4. The message is encrypted using the given key and method, and sent on to the server.

In terms of weaknesses in our implementation, there is certainly the weakness in SDES, as any messages sent via that can always be intercepted and cracked with little effort from the adversary. The other methods require a bit more effort to break. However, as the server and client are authenticated before any communication can take place, interception of messages will be a more difficult feat. This is certainly not the most robust or secure implementation of SSL, but it shows the ideas, and models the protocol well on a whole.

Looking at what could be improved on this project, initial planning certainly could have been steered in a different direction from the start. Initially we had planned to implement a chat server, where multiple clients would connect to the server, and communicate with each other, using the server as a form of trusted middle-man. As development went along, it became clear that this was not directly the goal of the project, and changes were made. Instead, we settled on a client-server connection basis, and cut out

any client-client interaction. Further, improving this project would largely consist of implementing larger, stronger, and more secure algorithms for encryption. At the point where the connection is secure, the only place that can really be improved with much significance is the encryption itself. If you are sure that Alice is in fact talking to Bob, and there are no adversaries receiving messages instead, then the only thing left to improve is how the messages are encrypted, in the case that someone may be viewing them, and attempting to decrypt in the middle.

Overall, this project greatly improved my understanding of the SSL protocol, as well as strengthened my understanding of Hashes, MACs, and digital signatures. If we were to continue work on this project, it would be an interesting challenge to attempt full SSL and stronger encryption methods. This would mean handling certificates, and modelling the same packet structure for messages as real SSL implementations. In our case, we did not do anything with certificates, and created our own simplified structure for messages. This allowed for more time to be spent on the cryptographic elements of the implementation, and less on the server communication portion, and message handling elements. Another large addition that could be made is transferring all of the interactions of server and client into a GUI, instead of all being handled in the command line itself. Overall, this project has certainly helped pique my interest in Cryptography, and cryptographic ideas and fields. I found implementing these encryption and hashing algorithms to be incredibly satisfying and just generally enjoyable, and am definitely interested in further pursuing this field in the future.