Ehren Schindelar
Cryptography 1
Dec. 3, 2018

Implementation of SSL Python Server/Client Pair

For this project, my team and I implemented our own version of SSL hosting a suite of cryptographic algorithms. We decided to use Python-2 as our language of choice, as it is used by most students at RPI. Our suite includes elliptic curve Diffie-Hellman and standard exponent based Diffie-Hellman algorithms for setting up a shared session key, and simplified data encryption standard and rivest cipher 4 algorithms for symmetric encryption.

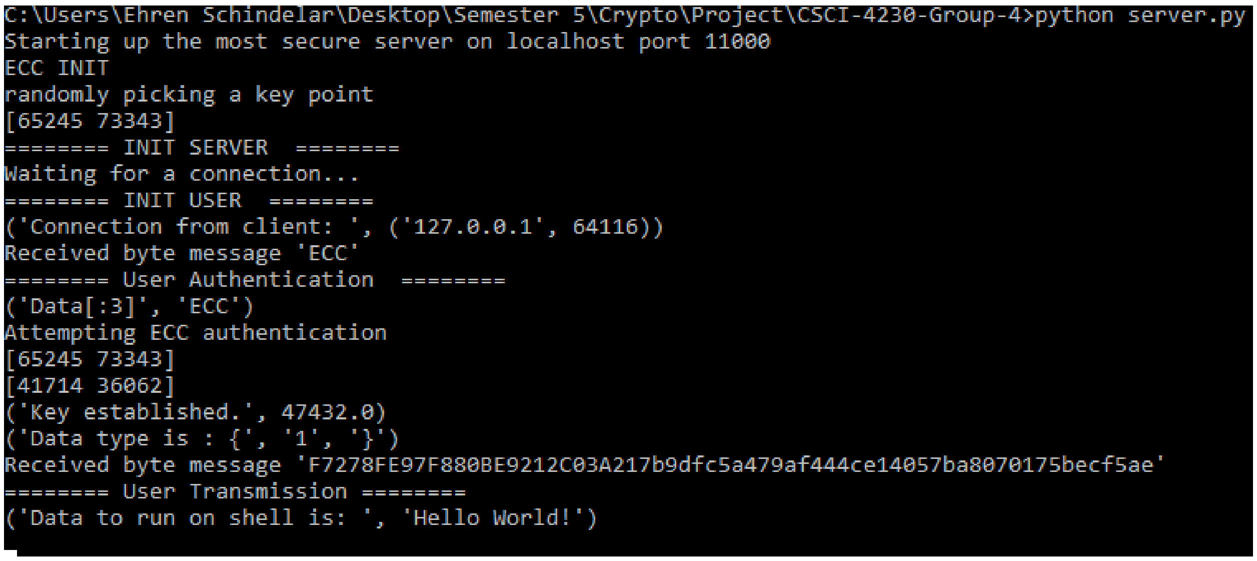
Our program intends to establish a secure connection between a client and a server using socket programming. To accomplish this, both must agree on an encryption scheme to use, and synchronize the required values. In this instance we only support symmetric encryption algorithms, so both parties need to share a common private key. Since the server and client may not already have a way to communicate secretly, some additional cryptography must be used.

Our suite supports the use of both elliptic curve Diffie-Hellman and standard exponent based Diffie-Hellman to generate keys. These both rely on the inability of an adversary to solve the discrete logarithm problem; in this way quantum computing may cause these methods to become insecure in the future. As long as the server is running, and the client has the correct address of the server, the client will be prompted to choose a distribution algorithm. When one is chosen, the server will choose some base value and send it to the client. Both sides will then either exponentiate mod p or multiply over an elliptic curve mod p that base value with some random number. The two will then swap the results, and perform the same operation to the other’s. Because it does not matter in which order these operations take place, this will result in both parties having the same value or point mod p. This will be used as the private key.

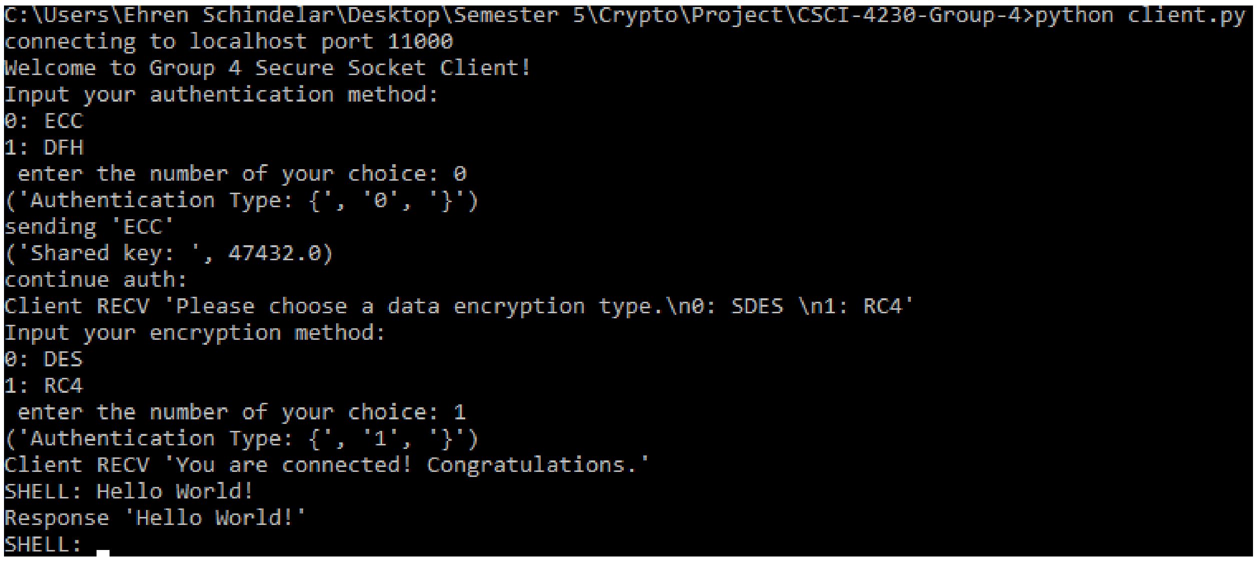
After that has completed, the server will then ask the client to choose an encryption algorithm. The client may choose to use a simplified data encryption standard algorithm, or the rivest cipher 4 algorithm. All messages to and from the server after that point will be encrypted under that choice. The implemented version of DES uses the lower ten bits of the provided key, and encrypts the message one byte at a time. It has two rounds of encryption and uses two four to two bit s-boxes. Due to the small key size, DES is not recommended.

In order to add an additional layer of security, we add a signature to every transmission, using a hash-based message authentication code. This is based on the hash value of the original message, as well as the private key. The adversary hopefully has access to neither, so in order to create a valid signature the best he can do is guess. This message authentication code ensures that the message is from a valid source, which in this case is from either the server or the client. The receiver can check the code by decrypting the message and re-computing the code, to see if they match. The receiver knows what part is the message and what part is the mac, because the mac is always the same length. HMAC provides a code of length 40, which we append to the end of the message.

Below you can find an example output for both the server and the client, using ECC Diffie-Hellman key exchange, and RC4 symmetric key encryption.

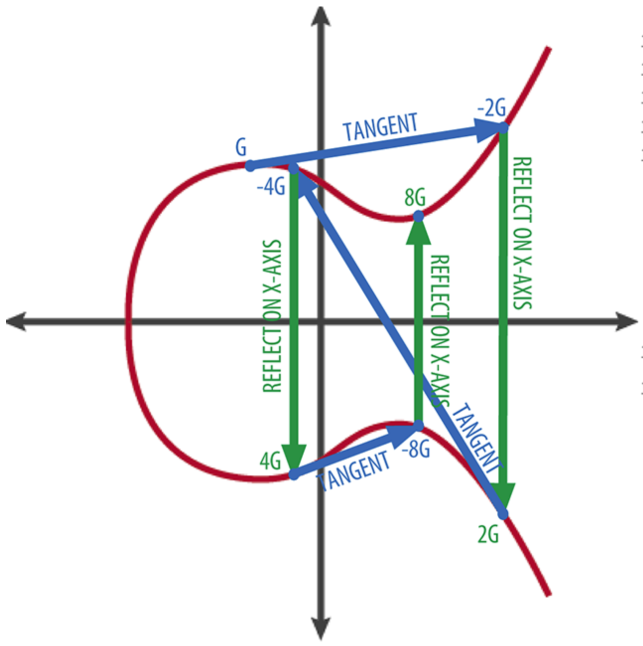


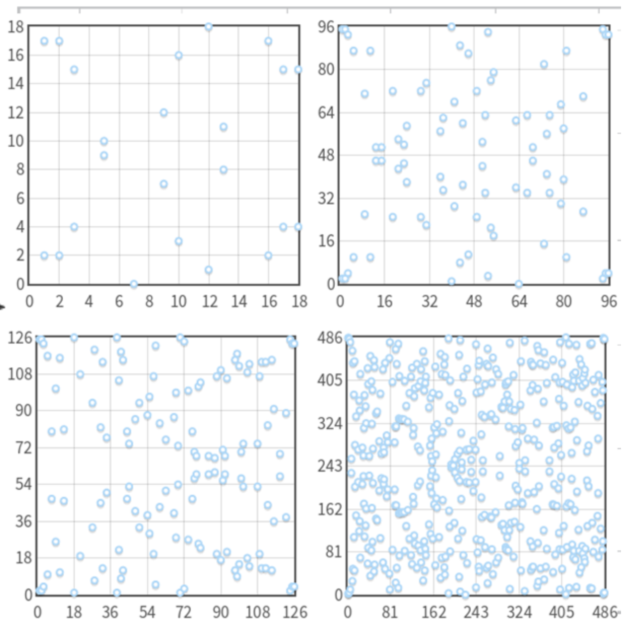
```
C:\Users\Ehren Schindelar\Desktop\Semester 5\Crypto\Project\CSCI-4230-Group-4>python server.py
Starting up the most secure server on localhost port 11000
ECC INIT
randomly picking a key point
[65245 73343]
======== INIT SERVER  ========
Waiting for a connection...
======== INIT USER  ========
('Connection from client: ', ('127.0.0.1', 64116))
Received byte message 'ECC'
======== User Authentication  ========
('Data[:3]', 'ECC')
Attempting ECC authentication
[65245 73343]
[41714 36062]
('Key established.', 47432.0)
('Data type is : {', '1', '}')
Received byte message 'F7278FE97F880BE9212C03A217b9dfc5a479af444ce14057ba8070175becf5ae'
======== User Transmission ========
('Data to run on shell is: ', 'Hello World!')
```



```
C:\Users\Ehren Schindelar\Desktop\Semester 5\Crypto\Project\CSCI-4230-Group-4>python client.py
connecting to localhost port 11000
Welcome to Group 4 Secure Socket Client!
Input your authentication method:
0: ECC
1: DFH
 enter the number of your choice: 0
('Authentication Type: {', '0', '}')
sending 'ECC'
('Shared key: ', 47432.0)
continue auth:
Client RECV 'Please choose a data encryption type.\n0: SDES \n1: RC4'
Input your encryption method:
0: DES
1: RC4
 enter the number of your choice: 1
('Authentication Type: {', '1', '}')
Client RECV 'You are connected! Congratulations.'
SHELL: Hello World!
Response 'Hello World!'
SHELL:
```

As for functionality, our server was originally intended to be a secure shell server, but we instead focused on the cryptography portion and ended up having the server only echo back any data sent to it. If the server encounters any error in decryption or authentication, it will respond with ‘-1’.





My contributions to this project, while helping out in many areas, mainly consisted of my implementation of elliptic curve cryptography. My code includes references to the numpy package, which contains methods for matrix manipulation. I began by implementing it as a public key cryptosystem, and then modified it to work as a Diffie-Hellman key generation protocol. It's main feature is the addition of two points over an elliptic curve. This is equivalent to observing the line between the two points and finding where else it intersects with the elliptic curve. In modular space, the points are compressed into a grid, which may retain the properties of the elliptic curve. In this way, is important to properly assign values of a, b, and p. In order to add a point to itself, one would observe the point's tangent line with the curve. Additionally, there is a point at infinity that results from lines of infinite slope, and any point added to it equals that other point. Multiplication is achieved by repeatedly adding a point to itself. In order to generate a shared key between two users, both users will agree upon some random point at which to start, and then multiply that point by some random number. Neither knows what number the other used, but when they multiply the other's result by their random value, they will end up with the same point. This is due to the commutative property of multiplication.

Aside from cryptographic security, one of the main issues with our code as it is now is that we do not properly segment the socket data stream. This is particularly dangerous during the authentication period, when packets are sent back and forth between the server and client in quick succession. This can cause multiple packets to be concatenated, and leave one end waiting for a second response. One way this could be remedied is by introducing a delimiter, and creating an object that reads from the data stream up until the next instance of the delimiter.