Assignment 3 Report

Part 1

For part one I first created a server and client that would just send plain text messages over a socket. This can be seen in the files `server - part 1 plain text.txt` and `client - part 1 plain text.txt` with each received message being printed first in hexadecimal then as a string. I then modified it so that the server first sent the client a random 1024 bit key and using that to perform xor encryption. The out put of this can be seen in `server - part 1 xor text.txt` and `client - part 1 xor text.txt` with each received encoded message being printed first in hexadecimal then as a string then the decoded message being printed in hexadecimal then as a string. For example:

```
e6 25 1e 34 34 99 8b 17 22 78 01 0c fb ...
b2 40 6d 40 14 f4 ee 64 51 19 66 69 f1
.@m@...dQ.fi.
54 65 73 74 20 6d 65 73 73 61 67 65 0a
Test message
```

The first line is the key (truncated), the second line is the received encoded message in hexadecimal and if you take `e6251e3434998b172278010cfb xor b2406d4014f4ee6451196669f1` you get the result `54657374206d6573736167650a`, which is the decoded message on line 4, so it is encoding and decoding correctly. The decoded message correctly translates to `Test message` where as the encoded message translates to `.@m@...dQ.fi.`. As you can see the encoded message does appear to be scrambled with most of the bytes not event being printable characters in ascii.

Part 2

For part 2 I implemented the Diffie-Hellman key exchange. You can see 15 tests of the key exchange in `server - part 2.txt` and `client - part 2.txt` and all the keys match up. The keys generated are 2048 bits and are all generated using the prime number `FFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74020BBEA63B139B22514A08798E3404DDEF9519B3CD3A431B302B0A6DF25F14374FE1356D6D51C245E485B576625E7EC6F44C42E9A637ED6B0BFF5CB6F406B7EDEE386BFB5A899FA5AE9F24117C4B1FE649286651ECE45B3DC2007CB8A163BF0598DA48361C55D39A69163FA8FD24CF5F83655D23DCA3AD961C62F356208552BB9ED529077096966D670C354E4ABC9804F1746C08CA18217C32905E462E36CE3BE39E772C180E86039B2783A2EC07A28FB5C55DF06F4C52C9DE2BCBF6955817183995497CEA956AE515D2261898FA051015728E5A8AACAA68FFFFFFFFFFFFFFFF` with generator `2`. These numbers were obtained from RFC 3526 (the 2048-bit MODP Group) which is an RFC that specifies numbers specifically for Diffie-Hellman.

Part 3

For part 3 I simply changed the server sending the key to the client in part 1 to use the Diffie-Hellman key exchange I made in part 2. The result of this can be seen in `server - part 3.txt` and `client - part

`3.txt` with each received encoded message being printed first in hexadecimal then as a string then the decoded message being printed in hexadecimal then as a string. For example:

```
1b 21 38 55 96 eb cf 47 37 e6 41 12 82 ...
Ready to chat
4f 44 4b 21 b6 86 aa 34 44 87 26 77 88
ODK!...4D.&w.
54 65 73 74 20 6d 65 73 73 61 67 65 0a
Test message
```

The first line is the truncated key and is the same for the server and client, and again `1b21385596ebcf4737e6411282 xor 4f444b21b686aa344487267788` equals `54657374206d6573736167650a` so it is encoding and decoding correctly. Again the encoded message appears scrambled being `ODK!...4D.&w.` as a string.

Security of Design

In order to make sure the key exchange is secure I use a prime of size 2048 bits as this is the recommended size to use nowadays for Diffie-Hellman. Also, a 2048 bit prime will mean a 2048 bit key at the end and the longer the key the more ciphertext you would need to crack the encryption through a frequency analysis. FOr a similar reason I use all of the key evenly, keeping tack of where the last message stopped during encryption, as if I used one part of the key more than another it would reduce the amount of ciphertext needed to start cracking parts of the key.

Reusing primes and generators in no way compromises the security of Diffie-Hellman so the use of the primes from RFC 3526 works fine. If you were to generate your own primes it will just take longer and the best you can do is generate a number that is probably a prime (although the chance of it not being a prime would be extremely low). Another advantage of the primes in RFC 3526 is they are save primes (i.e. a prime of the form `2 * q + 1` where `q` is also a prime) which have additional security benefits for Diffie-Hellman but can easily take over a minute to a 2048 bit one randomly.

For the choice of random exponents by the server and client, I was sure use the SecureRandom class in Java instead of just the Random class, This is because the Random class seeds itself using the system time, something that could be easily guessed, where as SecureRandom uses random sources of data provided by the OS as well as having a few other differences that make it far better equip for generating random numbers for cryptography.