Cryptography – Project 3

Code source available on my GitHub.

For this third project, we were tasked to recreate the Transport Layer Security (TLS) protocol. The requirement were authentication, key exchange, and data encryption.

The standard TLS protocol works as follow:

1. A client initiates a communication with a server.
2. The server responds by sending a digital certificate, containing its personal information which enables authentication, its public key, and a digital signature.
3. The client checks the certificate and uses the given public key to verify the authenticity of the signature.
4. The client and server exchange a secret key use for the upcoming encryption. This can be done by two ways:
    a. The client generates a random number and shares it to the server by encrypting it using the public key. Then both sides use the random number to generate the same key.
    b. Both the client and server generate a private key and public key and share the public one. Then they generate the secret common key by combining their private key and the other's public key.
5. They exchange a dummy encrypted message to confirm they use the same key.

After the initial communication, both sides can encrypt and decrypt messages using the shared generated secret key.

I implemented this using Python, using the Crypto module proposed by PyCryptodome for most of the technical parts. The server's certificate is composed exclusively of the public key. The signature is generated using an SHA-256 hash and the RSASSA-PKCS1-v1.5 encoding method.

The secret key is generated using the second method explained previously, more precisely with the multiplicative group of integers modulo n. The client generates a prime number p of 128-bits, and a random number g lower than it. Then it generates its private key x_c with a random number of 32-bits, and calculates the public key y_c as $y = g^{x_c} \bmod p$. The client shares p, g and y_c to the server, which generates its personal another private and public keys x_s and y_s the same way. It also resolves the secret key k as $k = y_c^{x_s} \bmod p = g^{x_c \cdot x_s} \bmod p$. The server then sends its public key y_s and an encrypted message to the client. The client calculates the secret key and check the message and reply with an equally encrypted message.

The secret key k is used for an AES encryption in CBC mode. The IV required is obtained using the SHA-256 hash of the secret key. The first 128-bits are used for the client to server encryption/decryption, and the last bits for the server to client communication. The modular exponentiation algorithm used for the key generation is originally from Wikipedia.

Four python scripts are available in order to test my implementation. I provided a regular client-server communication, without any encryption, named serverSimple.py and clientSimple.py, as well as their TLS encrypted equivalent as server.py and client.py. All scripts work locally. The server does not require any argument, while the client requires the server port, which is display when launching it. The only interaction possible are on the client side: any inputted string will be sent to the server, which will reply with the reversed string.

I also recorded and provided the message exchanged during an experiment, with both encrypted and plain communication. On both cases, I only created the connection, inputted "Hello World", and closed it. The result was retrieved using Wireshark, and is available as content.txt and contentSimple.txt. Note that the first 44 bytes ($2C_{16}$) on the exchanged data are sockets related.

After checking the memory usage, using this Stack Overflow post, I found that my simple server takes 2.3Kb, while the encrypted one takes 146.9Kb. The clients take 59.2Kb and 234.2Kb respectively. I obtained these results while reproducing the test mentioned before.

Using the result from Wireshark, we can see the initialization of the encryption takes approximately 0.02 seconds with a localhost communication. Additionally, the delta time between the client message and the server response is $2x10^{-3}$ seconds with encryption, and $5x10^{-4}$ seconds without.