HY-457
Assignment 1

Assigned: 27 / 2 / 2019
Due: 13 / 3 / 2019

Introduction

In this assignment you are going to develop, step-by-step, a simple symmetric key exchange mechanism, using C, for a secure chat server/client scenario, using the OpenSSL toolkit. The purpose of this assignment is to provide you the opportunity to get familiar with public key (RSA) and symmetric key (AES) cryptography and the very popular general-purpose cryptography toolkit, OpenSSL. Also, you will be able to acquire hands-on experience in implementing simple cryptographic applications. The tool will provide RSA encryption/ decryption in order to exchange AES keys and establish a secure communication channel using the OpenSSL toolkit. Once the secure channel is established, the server and the client can securely communicate using encrypted messages.

Steps and Objectives

1) A simple TCP server/client

You will start the implementation of this exercise by implementing a simple server/client, using C, that communicate using TCP sockets. The server will start first and then wait for a single connection from a client. The client will start second and connect with the server. At this point, the client should be able to send messages to the server. The server will print the messages on the screen and reply back to the client with the same message. The server does not need to support multiple client connections so you do not need to use threads for this implementation.

2) Symmetric Key Cryptography (AES)

Once your server/client task is complete, you need to develop functions that provide symmetric key cryptographic functionality using the OpenSSL EVP API [1]. For this task you will also need to use the **aes_key.txt** file provided with this assignment. More specifically, you have to develop a function that reads the AES key from **aes_key.txt**, a function that encrypts a message and a function that decrypts the message. The encryption and decryption process should be performed using the provided key and the OpenSSL EVP API using AES ECB 128-bit mode. You should be able to successfully decrypt the encrypted messages without corrupting them. You can verify their correctness using a simple test program. We recommend that you verify their correctness without using the server/client infrastructure.

[1] https://wiki.openssl.org/index.php/EVP

3) Asymmetric Key Cryptography (RSA)
Now that you have successfully implemented the previous steps, you have to implement functions that perform Asymmetric Key cryptographic operations. Once again, you have to implement a function that reads the RSA private and public keys. The keys can be found in **.pem** format in the **keys** folder. Then you have to implement a function that encrypts a message using the private key and a function that encrypts a message using the public key. Then, you will implement a function that decrypts a message using a public key and a function that decrypts a message using the private key. At this point, you have to note that a message encrypted with a private key can only be decrypted using the corresponding public key and vice versa, meaning that a message encrypted with a public key can only be decrypted using the corresponding private key. When you finish the implementation of these functions, you should verify their correctness using a simple test program. We recommend that you verify their correctness without using the server/client infrastructure. Once again, you have to use OpenSSL functions and more specifically functions found in openssl/rsa.h, openssl/bio.h and openssl/pem.h

4) Key Exchange Protocol
Now that you have all the functions and the simple server/client implemented, it is time to put it all together in order to enhance the server/client with secure, encrypted communications. In order to achieve this, the server and the client need to implement the following key exchange mechanism, using the functions you implemented.

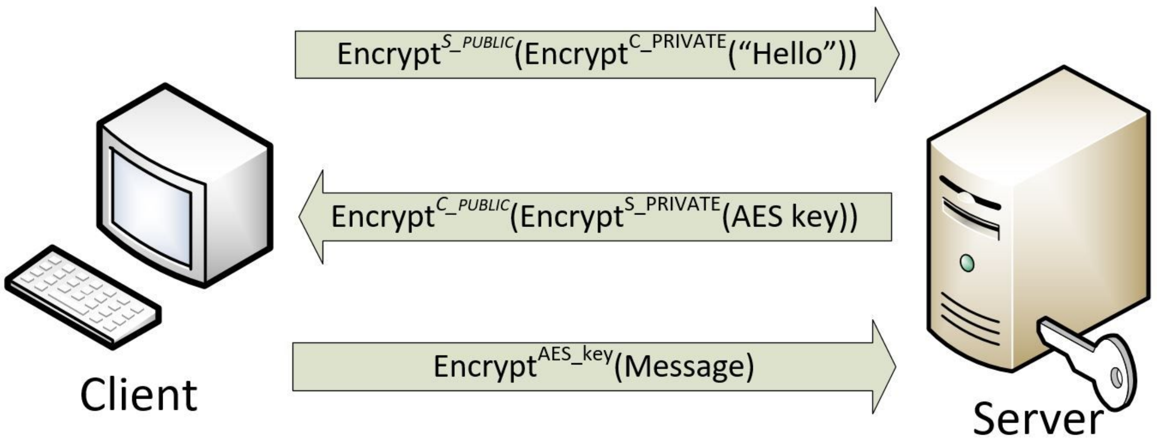
The client should **only** load its private key and the server's public key. The client should **not** load the server's private key or the AES key.The server should **only** load its private key, the client's public key and the AES key. The server should **not** load the clients private key.

In a real world scenario, each entity should **never** expose their private keys or the AES key to anyone. On the other hand, public keys are available to everyone and there are various mechanisms in order for one entity to find the public key of another. However, in order to assist you with the development, we have placed all the keys in a single folder, so each entity (server and client) should **only** load the keys specified above and **ignore** the rest.

The server and the client know the *public key* of each other. When the client wants to contact the server, the following process should take place.

A. The client sends to the server a "hello" message. The message will be first encrypted using the **client's** *private key* and then with the **server's** *public key.*
B. When the message is received from the server, it will first be decrypted using the **server's** *private key* and then with the **client's** *public key*.
C. If the message is decrypted correctly (decrypted message corresponds to "hello") the server will encrypt the AES key (loaded from aes_key.txt) and use the same steps as before to send it to the client i.e. encrypt it first with the **server's** *private key* and then with the **client's** *public key*.
D. Then, the client will decrypt the the message containing the symmetric key by first decrypting it with the **client's** *private key* and then with the **server's** *public key* and store it in a variable in order to use it for the rest of the communication.

At this point both the server and the client will use symmetric cryptography in order to encrypt/decrypt messages exchanged between them. The client will use the message obtained by the command line, encrypt it with the exchanged AES key and send it back to the server. The server is now able to decrypt the message with the AES key and print it on screen.


$\text{Encrypt}^{S\_PUBLIC}(\text{Encrypt}^{C\_PRIVATE}(\text{“Hello”}))$
$\text{Encrypt}^{C\_PUBLIC}(\text{Encrypt}^{S\_PRIVATE}(\text{AES key}))$
$\text{Encrypt}^{AES\_key}(\text{Message})$
Client
Server

The exchanged messages between the Server and the Client

Server and client specifications

In order to assist you in the development of this assignment, we provide a basic skeleton of both the client and the server. We also provide some helper functions, used to print the plaintext as a string and the bytes of the ciphertext and keys in a human readable form.

The server will receive the required arguments from the command line upon execution as such:

-p <port>	Port the server listens to

The client will receive the required arguments from the command line upon execution as such:

-i <ip>	Server’s IP
-p <port>	Port the server listens to
-m <msg>	The message that will be sent, encrypted with the AES key after the key exchange is complete

Bonus

1. Change the AES implementation from AES-ECB 128-bit to AES-CBC 128-bit. This AES mode requires the utilization of Initialization Vectors (IV). Explore and implement the correct way to generate, alter and use the IV in order to perform AES-CBC encryption.
2. Private keys are very important in asymmetric cryptography. Although public keys can be known and shared with everyone, private keys should never be compromised. For this reason, storing private keys should be performed in a secure way. Find a secure way to store and retrieve the private keys.

Important notes

1. You need to submit the server.c, the client.c, a Makefile that compiles the two src codes and a Readme file that explains your implementation or unimplemented parts.
2. The draft we provide for server.c, client.c and cs457_crypto.h are optional and you can perform any modifications you want, or extend the command line arguments. However, it is very advisable to at least use the command line options provided.
3. If you implement more command line options, explain their functionality in the Readme file.
4. This assignment should be implemented using C.
5. Follow the steps described above and do an incremental implementation. This will be very helpful in order to debug the various parts before putting them all together.
6. When assigning port numbers, use high ports (ie. your AM) and avoid using reserved ports, such as 22, 443 etc.
7. You can use the course's mailing list for questions. However, read the previous emails first since your question might have already been answered.
8. Do **not** send private messages with questions to the TAs, since other students might have the same question and everyone deserves the answer.
9. Do not send code snippets or pieces of your implementation to the mailing list when asking a question.