Security Systems Project

Mersinias Michail, 2013030057, Technical University of Crete, 2017

Encryption is one of the most basic concepts of computer security. Confidentiality is a human right and as junior security engineer students, it is our duty to defend it. Therefore, in the following project, we will be trained in the Art of War in the information field. Our task is to create from scratch a secure and reliable system of exchanging encrypted data in a Client-Server environment. Let us begin.

Additional Key Words and Phrases: AES Encryption and Decryption, RSA key generation and exchange, SSL, Client-Server communication, Digital Certificate authentication, etc

1. INTRODUCTION

In this project, we were asked to implement a fully functioning message exchanging system, in a Client-Server environment. The steps were as follows:

> For the first part, we were asked to use the modules created in a exercise 2 (AES encryption and decryption, RSA key generation, RSA encryption and decryption, SHA256 Hashing, Sign and Verify Digital Signatures). We were also asked to create a function to verify digital certificates by comparing the issuer and the subject name, as well as checking the date to see if it has expired.
>
> For the second part, we were asked to create a server and a client. The client connects to the server via a python socket. For this part, only the server is authenticated via a digital certificate. When this is completed, the client will prompt the user for a single line of input and send it to the server. The server receives text from the client, prints it to the screen and sends it back as an echo.
>
> For the third part, we were asked to modify the second part so that authentication is two-way. So after the connection, both the server and the client will exchange and verify digital certificates. Afterwards, both the client and the server generate RSA-2048 key pairs, save them to their respective files and then they exchanged the symmetric key produced at the server through RSA encryption/decryption. Finally, the client and the server exchange AES encrypted messages using the symmetric key they had exchanged with the RSA cryptosystem.

2. PART 1: BUILDING BLOCKS

The menu of this part is located at the ”project_menu” file.

2.1. AES Encryption and Decryption

The AES module was taken from exercise 2 and works as intended. It was used in CBC mode.

Two new functions had to be created (one for padding, one for reverse padding), so we can use the desired PKCS#7 padding, because PKCS#5 padding only works for blocks up to 8 bytes, but our block size is 16 bytes.

Parameters are the file name of the plaintext (for encryption) or file name of the encrypted text (for decryption) and the key. The result is saved to a file named ”aes_enc_file.txt” for encryption or ”aes_dec_file.txt” for decryption.

Source code is located at the ”tools.py” file.

2.2. RSA Key Generation, Encryption and Decryption

The RSA module was taken from exercise 2 and works as intended.

Key Generation is used to produce an RSA key pair based on the desired length specified by user input.

Encrypt and Decrypt is used to securely encrypt information with the public key so that only the private key can decrypt it.

Both of these modules are not contained in part 1 menu options, as the description of the project did not mention it as an option of the menu.

However, it is used in part 3 of the project to securely exchange a symmetric key between the client and the server.

Source code is located at the ”tools.py” file.

2.3. SHA256 Hashing

The SHA256 module was taken from exercise 2 and works as intended.

Parameters are the file name of the plaintext. The result is saved to a file named ””hashed_file.txt””.

Source code is located at the ”tools.py” file.

2.4. Digital Signature Signing

The Sign Digital Signature module was taken from exercise 2 and works as intended.

Parameters are the name of the file to be signed and the name of the file containing the RSA private key. The result is saved to a file named ””signed_file.txt””.

Source code is located at the ”tools.py” file.

2.5. Digital Signature Verification

The Verify Digital Signature module was taken from exercise 2 and works as intended.

Parameters are the name of the signed file, the name of the file containing the digital signature and the name of the file containing the RSA public key. It returns an indication: True or False.

Source code is located at the ”tools.py” file.

2.6. Certificate Verification

In order to verify a certificate, we had to check the issuer name and compare it to the subject name. We also had to check the expiration date and make sure the certificate is still valid. In order to do this, at first we had to get the certificate contents. Then, we extracted the issuer name and the subject name. If they are equal, our first check for certificate verification is completed successfully. Afterwards, we make sure the certificate hasn't expired based on its expiration date. If false is returned, the second check for certificate verification is completely successfully and the certificate is hereby verified.
Parameters are the contents (as a string) of the certificate file. It returns an indication: True or False.

Source code is located at the "tools.py" file.

3. PART 2: ESTABLISHING CLIENT-SERVER CONNECTION

In the second part, we created 2 new python files: a server (server.py) and a client (client.py). Our goal was: Connection, One-way authentication, and message exchanging.

In order to do that, at first we connected the client to the server via a python socket. After the connection is established, the server creates a X509 certificate by utilizing the OpenSSL library and sends its contents to the client for authentication. It is authenticated by using the Verify Certificate module created in Part 1, with the server certificate contents as a parameter, and with the same functionality as described in Part 1.

After successful authentication, by utilizing the .send and .recv SSL commands appropriately, the client prompts the user for a single line of input and sends it to the server. The server receives that text line from the client, prints it to the screen, and also sends it back to the client (echo).

Source code is located at the "server.py" and "client.py" files, also calling functions from the "tools.py" file.

4. PART 3: CLIENT-SERVER ENCRYPTED COMMUNICATION

In the third part, once again, we created 2 new python files: a server (server2.py) and a client (client2.py). This time, our goal was: Connection, Two-way authentication, symmetric key generation and exchange through a RSA cryptosystem (RSA key generation, encryption and decryption) and finally AES encrypted message exchanging using the symmetric key mentioned above.

In order to carry out this task, at first we established the connection between server and client via python socket. Afterwards, we create a server certificate and a client certificate. By utilizing the .send and .recv SSL commands, we send the client certificate contents to the server and the server certificate contents to the client. Both the client and the server authenticate the certificate contents they received by using the Verify Certificate function from Part 1, with the respective contents as a parameter.

Afterwards, the server and the client generate 2 RSA key pairs by utilizing the RSA Key Generation function that had been created in exercise 2 and is located in the ”tools.py” file.

When this is completed, we establish the connection between the server and the client and we write each RSA key pair to a respective file (”client2_rsa_keys.pair” for the client key pair and ”server2_rsa_keys.pair” for the server key pair). In both cases, the first line represents e, the second line represents n, and the third line represents d.

Our next step is to establish the connection between the server and the client via python socket. When this is completed successfully, the server will read the client RSA key pair from the ”client2_rsa_keys.pair” file and the client will read the server RSA key pair from the ”server2_rsa_keys.pair” file, and extract any necessary key parameters (for example: e, n) to save them in local variables for later use in the RSA cryptosystem.

Afterwards, we move on to the two-way authentication by verifying both certificates we had exchanged between the server and the client earlier. If it is successful, the server generates a symmetric key by utilizing the aes_key_generation function located in the ”tools.py” file which simply generates a random AES key.

We proceed by encrypting the key by using the rsa_encrypt function (2048) located in the ”tools.py” file. The result is a byte array, so we have to convert it to a string by using the byte_array_to_string function located in the ”tools.py” file, in order to be able to send it from the server to the client. After converting it to string, we send it by utilizing the .send and .recv SSL commands and making sure the recv buffer is big enough (10*2048) so we can avoid a possible buffer overflow error.

After the client receives the string representing our encrypted symmetric key, it has to convert it to byte array in order to be able to decrypt it. This is done by using the string_to_byte_array function located in the ”tools.py” file. After we convert it to byte array successfully, we decrypt it by using the rsa_decrypt function.

As the previous process is completed, we now have successfully exchanged the symmetric key in a secure way (RSA cryptosystem) and it can be used by both the server and the client. Assuming a fixed v_i for CBC mode, we are now ready to start sending AES encrypted messages.

Therefore, once again the client prompts the user for a single line of input, which is immediately padded in a PKCS#7 way and encrypted with AES (using the symmetric key exchanged before through the RSA cryptosystem) and sends it to the server.
The server receives the encrypted text, decrypts it with AES (using the symmetric key exchanged before through the RSA cryptosystem) and reverses the PKCS#7 padding to get the original message and print it.

Finally, the server will pad the message in a PKCS#7 way, encrypt that message with AES once more (using the same key) and send it to the client (echo). The client will receive the message, decrypt it with AES (using the same key), reverse the PKCS_7 padding and print it.

Source code is located at the ”server2.py” and ”client2.py” files, also calling plenty of functions from the ”tools.py” file.

5. CONCLUSIONS

In this project, we learnt and practised the security concepts regarding client-server connection, certificate management and authentication, and encrypted data exchanging.
-We understood how to implement SSL and establish a Client-Server environment.
-We learnt how to create, exchange, manage and verify X509 certificates.
-We had the chance to create from scratch a complete and secure system of exchanging keys (by using RSA) and encrypted messages (by using AES) in Part 3.

By doing so, we saw how the individual modules we successfully created in exercise 2 can be used together in order to create an encryption system of high standards. This project was practical and useful, giving us the chance to implement and unite many security tools regarding encryption into a fully functioning and secure cryptosystem.

6. REFERENCES

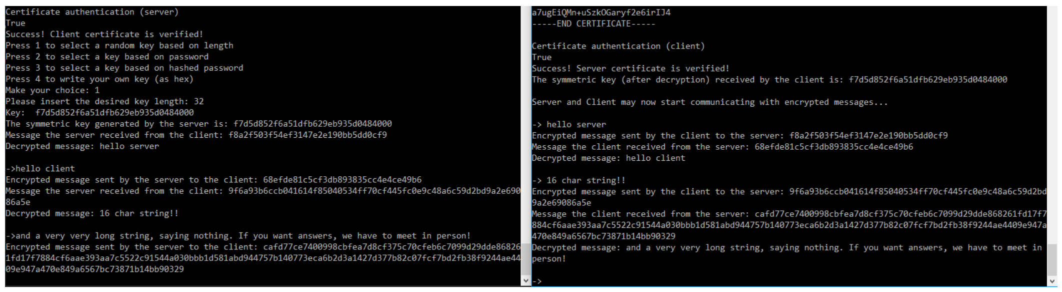
References:

1. http://programmerin.blogspot.gr/2011/08/python-padding-with-pkcs7.html
2. https://cryptography.io/en/latest/x509/tutorial/#creating-a-self-signed-certificate
3. http://www.pyopenssl.org/en/stable/api/crypto.htmlOpenSSL.crypto.X509Name
4. https://jamielinux.com/docs/openssl-certificate-authority/
5. https://www.youtube.com/watch?v=XiVVYfgDolU
6. https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.csfb400/pkcspad.htm
7. https://carlo-hamalainen.net/blog/2013/1/24/python-ssl-socket-echo-test-with-self-signed-certificate

and more..

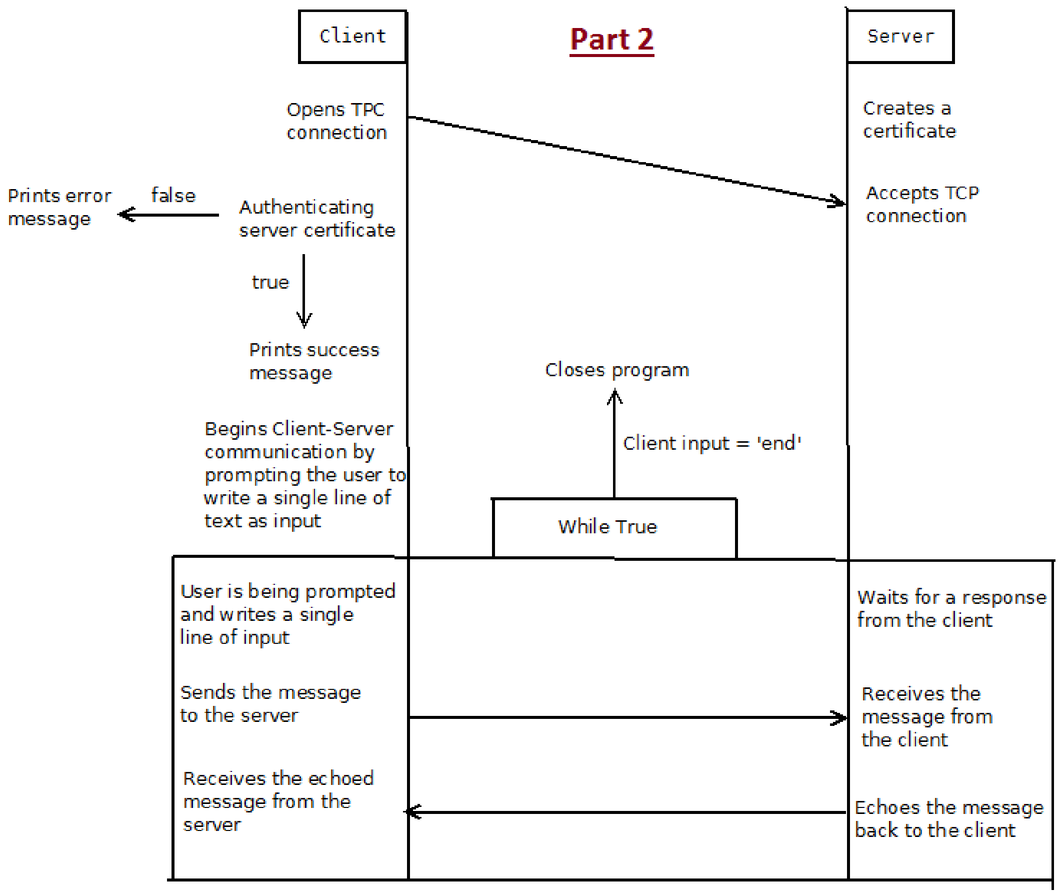
APPENDIX

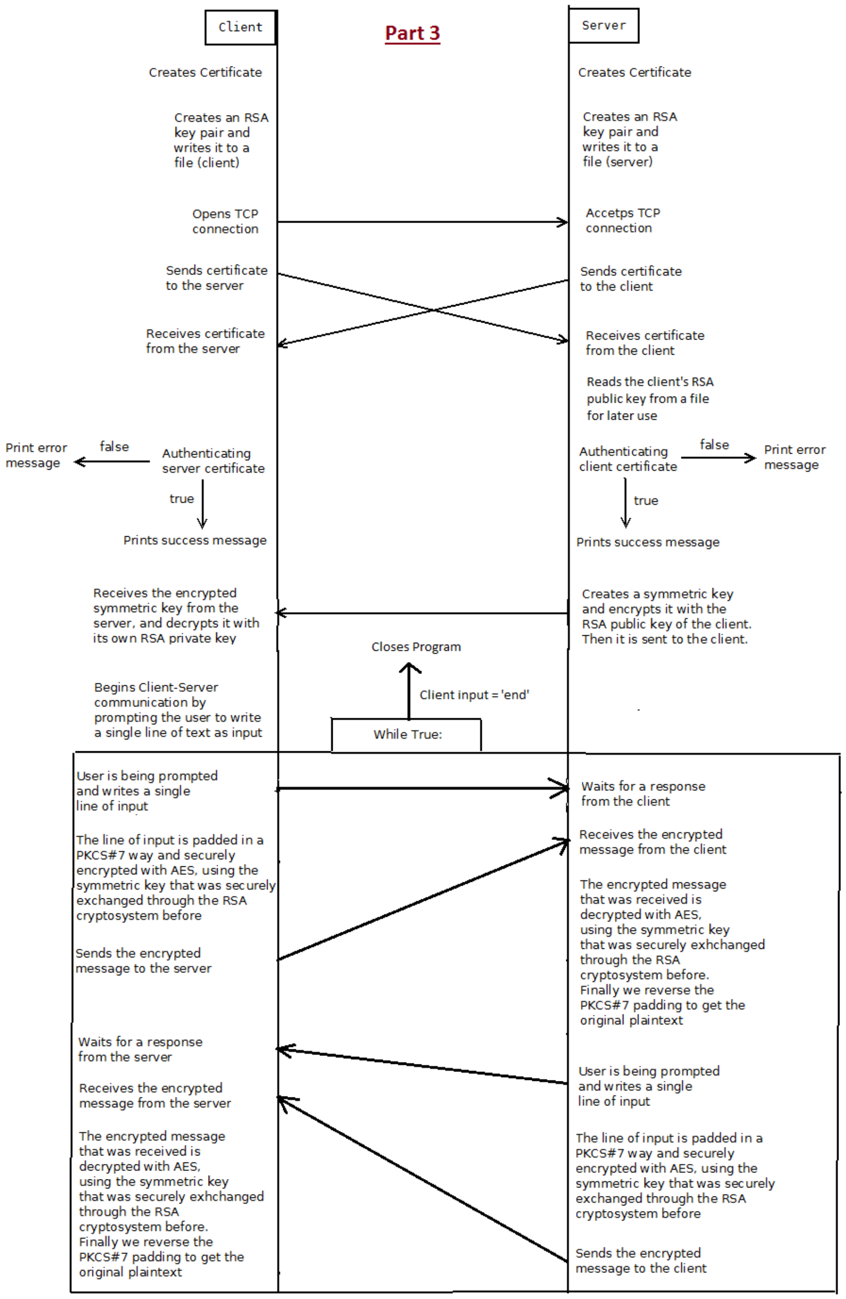
In this last section, I present a block diagram explaining the steps of part 2, a block diagram explaining the steps of part 3, as well as a snapshot showing the correct functionality of part 3.



```
Certificate authentication (server)
True
Success! Client certificate is verified!
Press 1 to select a random key based on length
Press 2 to select a key based on password
Press 3 to select a key based on hashed password
Press 4 to write your own key (as hex)
Make your choice: 1
Please insert the desired key length: 32
Key:  f7d5d852f6a51dfb629eb935d0484000
The symmetric key generated by the server is: f7d5d852f6a51dfb629eb935d0484000
Message the server received from the client: f8a2f503f54ef3147e2e190bb5dd0cf9
Decrypted message: hello server

->hello client
Encrypted message sent by the server to the client: 68efde81c5cf3db893835cc4e4ce49b6
Message the server received from the client: 9f6a93b6ccb041614f85040534ff70cf445fc0e9c48a6c59d2bd9a2e690
86a5e
Decrypted message: 16 char string!!

->and a very very long string, saying nothing. If you want answers, we have to meet in person!
Encrypted message sent by the server to the client: cafd77ce7400998cbfea7d8cf375c70cfeb6c7099d29dde86826
1fd17f7884cf6aae393aa7c5522c91544a030bbb1d581abd944757b140773eca6b2d3a1427d377b82c07fcf7bd2fb38f9244ae44
09e947a470e849a6567bc73871b14bb90329
```

```
a7ugEiQMn+uSzkOGaryf2e6irIJ4
-----END CERTIFICATE-----

Certificate authentication (client)
True
Success! Server certificate is verified!
The symmetric key (after decryption) received by the client is: f7d5d852f6a51dfb629eb935d0484000

Server and Client may now start communicating with encrypted messages...

-> hello server
Encrypted message sent by the client to the server: f8a2f503f54ef3147e2e190bb5dd0cf9
Message the client received from the server: 68efde81c5cf3db893835cc4e4ce49b6
Decrypted message: hello client

-> 16 char string!!
Encrypted message sent by the client to the server: 9f6a93b6ccb041614f85040534ff70cf445fc0e9c48a6c59d2bd
9a2e69086a5e
Message the client received from the server: cafd77ce7400998cbfea7d8cf375c70cfeb6c7099d29dde868261fd17f7
884cf6aae393aa7c5522c91544a030bbb1d581abd944757b140773eca6b2d3a1427d377b82c07fcf7bd2fb38f9244ae4409e947a
470e849a6567bc73871b14bb90329
Decrypted message: and a very very long string, saying nothing. If you want answers, we have to meet in
person!

->
```


Client
Part 2
Server
Opens TPC connection
Creates a certificate
Accepts TCP connection
Prints error message
false
Authenticating server certificate
true
Prints success message
Closes program
Client input = 'end'
Begins Client-Server communication by prompting the user to write a single line of text as input
While True
User is being prompted and writes a single line of input
Waits for a response from the client
Sends the message to the server
Receives the message from the client
Receives the echoed message from the server
Echoes the message back to the client


Client
Part 3
Server
Creates Certificate
Creates Certificate
Creates an RSA key pair and writes it to a file (client)
Creates an RSA key pair and writes it to a file (server)
Opens TCP connection
Accetps TCP connection
Sends certificate to the server
Sends certificate to the client
Receives certificate from the server
Receives certificate from the client
Reads the client's RSA public key from a file for later use
Print error message
false
Authenticating server certificate
true
Prints success message
Authenticating client certificate
false
Print error message
true
Prints success message
Receives the encrypted symmetric key from the server, and decrypts it with its own RSA private key
Creates a symmetric key and encrypts it with the RSA public key of the client. Then it is sent to the client.
Closes Program
Client input = 'end'
While True:
Begins Client-Server communication by prompting the user to write a single line of text as input
User is being prompted and writes a single line of input
Waits for a response from the client
The line of input is padded in a PKCS#7 way and securely encrypted with AES, using the symmetric key that was securely exchanged through the RSA cryptosystem before
Receives the encrypted message from the client
Sends the encrypted message to the server
The encrypted message that was received is decrypted with AES, using the symmetric key that was securely exhchanged through the RSA cryptosystem before. Finally we reverse the PKCS#7 padding to get the original plaintext
Waits for a response from the server
User is being prompted and writes a single line of input
Receives the encrypted message from the server
The encrypted message that was received is decrypted with AES, using the symmetric key that was securely exhchanged through the RSA cryptosystem before. Finally we reverse the PKCS#7 padding to get the original plaintext
The line of input is padded in a PKCS#7 way and securely encrypted with AES, using the symmetric key that was securely exchanged through the RSA cryptosystem before
Sends the encrypted message to the client