Project

Data and Digital Communication

OpenSSL Encryptions

CASTRO, LEO MARK D.C || BS –CPE 3B
PROFESSOR || MA'AM ROSELIA G. DELA CRUZ

Contents

CHAPTER 1 : INTRODUCTION

Main Goal Outline

Cryptography Exercises using OpenSSL

1. Download and install OpenSSL, if you do not have it yet.
   - https://www.openssl.org
   - For each of the following exercises, you are given the freedom on how you would use OpenSSL (i.e., accessed via a shell script, or called from within your own program, etc.).
   - Reference all sources that you use in your answers.

2. SYMMETRIC ENCRYPTION.
   - Study the OpenSSL Library and use it to perform symmetric AES encryption on the 512x512 Color (24-bit) Lena image (http://www.ece.rice.edu/~wakin/images/lena512color.tiff)
   - Use both ECB and CBC mode, for AES-128.
   - Fully document the process (in a document) of how you performed the encryption using OpenSSL, and the results of the encryption.

3. HASHING.
   - Using OpenSSL, hash the same Lena 512x512 image using the following hash functions:
   - SHA-1, SHA-256, SHA-512.
   - Again, fully document the process and the results of the hash.

4. PUBLIC KEY ENCRYPTION.
   - Using OpenSSL, perform an RSA encryption on the Lena 512x512 image, using RSA-2048.
   - Using OpenSSL, generate an ECDSA signature on the same Lena image.
   - If you need to use a hash function, use SHA-256.

- For all other details not outlined in this spec sheet, you have the freedom to choose or decide on the design detail. For example, you can define your own passwords or passphrases as basis for key generation.

TO BE SUBMITTED:

1. All source code, scripts, and documentation (in PDF) are to be housed in a git repository. You may use a public git repository (e.g., create an account on github.com or bitbucket.com) or your own private git repository.

2. Submit an accessible link to your git repository via e-mail to roselia,delacruz@bulsu.edu.ph by the deadline. I should be able to clone your git repo given the link and run (compile) your code/scripts from my machine. You should provide sufficient documentation for me to replicate your environment, e.g., what operating system, programming language, etc.

To-Do List

1. Create a python program that can interact with the OpenSSL program from a console environment
2. Do an AES-128-ECB and AES-128-CBC encryption function
3. Do a SHA1, SHA256, and SHA512 hash function
4. Do an RSA encrypted transaction.
5. Do an ECDSA certification and verification function
6. Test and explain all of this function

Background

OpenSSL is a software library for applications that secure communications over computer networks against eavesdropping or need to identify the party at the other end. It is widely used by Internet servers, including the majority of HTTPS websites.

OpenSSL contains an open-source implementation of the SSL and TLS protocols. The core library, written in the C programming language, implements basic cryptographic functions and provides various utility functions. Wrappers allowing the use of the OpenSSL library in a variety of computer languages are available.

The OpenSSL Software Foundation (OSF) represents the OpenSSL project in most legal capacities including contributor license agreements, managing donations, and so on. OpenSSL Software Services (OSS) also represents the OpenSSL project, for Support Contracts.

OpenSSL is available for most Unix-like operating systems (including Linux, macOS, and BSD) and Microsoft Windows.

Technologies To Be Used

Approach

The approach is to interact with the OpenSSL interface using console commands issued by a general programming language that would handle all the interactions and flow of the overall program.

Stack

- Python 3.9
  - This will operate on a virtual environment to ensure compatibility of packages used:
    - PyFiglet – A UI package
- Portable OpenSSL x64 program
  - This file set was a program downloaded from https://slproweb.com/products/Win32OpenSSL.html. To make the python program portable on Windows x64 computers, after the installation of the program, the files installed at Program Data by this quick installer was copied and put into the python working directory.
  - To access this program, the script will have to refer to its engine by calling it in its relative resident location ("bin\OpenSSL-Win64\bin\openssl.exe")

CHAPTER 2 : BUILDING THE BOILERPLATE

PHASE 1 : Connecting to the console and OpenSSL

Interfacing with the console

We need to have a way to communicate with openssl. OpenSSL mainly interfaces with the console/command prompt. So we need to find a way to first communicate with the command prompt and then communicate with openssl using our console.

In python, we can communicate with the command prompt console by this function.

```
import os
os.system("dir")
```

The example above should call the “dir” function of the command prompt.

Interfacing with the OpenSSL

OpenSSL can be accessed by running its “openssl.exe” file inside its root folder and going to the “bin” folder.

If you double-click “openssl.exe”, this prompt in the console should open. From this point, you can access the commands available.

```
OpenSSL>
```

Accessing OpenSSL from our Script

Now that we can access the console from python script, and we also know how to access the OpenSSL program, we will now connect the two.

The first step is to open OpenSSL program from the command prompt. This can be done by directly openning the file at its directory. In this program, this is the structure of the files.

```
app
    START.py
    OpenSSL.py
  - bin
    - OpenSSL-Win64
        - bin
            openssl.exe
  - KEYS
  - local_python
  - WORK_AREA
____START.bat
____Install.bat
```

The actual script is the “START.py”. So to access “openssl.exe”, we need to go to the bin folder, then open “OpneSSL-Win64”. From the list of folders inside openssl-win64, we only need to access “bin” folder which contains “openssl.exe”

So, to open it, we need to issue this command on command prompt.

```
os.system("bin\\OpenSSL-Win64\\bin\\openssl")
```

Testing

To test everything, we will run the “help” command of OpenSSL to see all the commands available for us.

```python
import os
os.system("bin\\OpenSSL-Win64\\bin\\openssl help")
```

```
Standard commands
asn1parse         ca                ciphers           cms
crl               crl2pkcs7         dgst              dhparam
dsa               dsaparam          ec                ecparam
enc               engine            errstr            gendsa
genpkey           genrsa            help              list
nseq              ocsp              passwd            pkcs12
pkcs7             pkcs8             pkey              pkeyparam
pkeyutl           prime             rand              rehash
req               rsa               rsautl            s_client
s_server          s_time            sess_id           smime
speed             spkac             srp               storeutl
ts                verify            version           x509

Message Digest commands (see the `dgst' command for more details)
blake2b512        blake2s256        gost              md4
md5               mdc2              rmd160            sha1
sha224            sha256            sha3-224          sha3-256
sha3-384          sha3-512          sha384            sha512
sha512-224        sha512-256        shake128          shake256
sm3

Cipher commands (see the `enc' command for more details)
aes-128-cbc       aes-128-ecb       aes-192-cbc       aes-192-ecb
aes-256-cbc       aes-256-ecb       aria-128-cbc      aria-128-cfb
aria-128-cfb1     aria-128-cfb8     aria-128-ctr      aria-128-ecb
aria-128-ofb      aria-192-cbc      aria-192-cfb      aria-192-cfb1
aria-192-cfb8     aria-192-ctr      aria-192-ecb      aria-192-ofb
aria-256-cbc      aria-256-cfb      aria-256-cfb1     aria-256-cfb8
aria-256-ctr      aria-256-ecb      aria-256-ofb      base64
bf                bf-cbc            bf-cfb            bf-ecb
bf-ofb            camellia-128-cbc  camellia-128-ecb  camellia-192-cbc
camellia-192-ecb  camellia-256-cbc  camellia-256-ecb  cast
cast-cbc          cast5-cbc         cast5-cfb         cast5-ecb
cast5-ofb         des               des-cbc           des-cfb
des-ecb           des-ede           des-ede-cbc       des-ede-cfb
des-ede-ofb       des-ede3          des-ede3-cbc      des-ede3-cfb
des-ede3-ofb      des-ofb           des3              desx
idea              idea-cbc          idea-cfb          idea-ecb
idea-ofb          rc2               rc2-40-cbc        rc2-64-cbc
rc2-cbc           rc2-cfb           rc2-ecb           rc2-ofb
rc4               rc4-40            seed              seed-cbc
seed-cfb          seed-ecb          seed-ofb          sm4-cbc
sm4-cfb           sm4-ctr           sm4-ecb           sm4-ofb
```

PHASE 2 : Creating the high-level functions

General Declarations

To make the code easy to read and intuitive in future iteration, we will declare certain variables for global use.

```
WORK_AREA = "WORK_AREA"
KEYS = "KEYS"
IMAGE = f"{WORK_AREA}/lena512color.tiff"
OPENSSL = "bin\\OpenSSL-Win64\\bin\\openssl"
```

Work_Area and Keys contain the folder where images and keys are saved. The image is the location where our sample image is located. Then finally, OpenSSL holds the file location of the openssl application exe. We will use the openssl variable through our the program whenever we will call the openssl program.

Symmetric Encryption

For AES encryptions, it takes a file and then asks a password to be used for file encryption. For file decryption, it will ask for a password that will be used to decrypt the said file

Encryption

```
os.system(f"{OPENSSL} enc -{mode} -e -in {input_file} -out
{output_location} -K {key} {f'-iv {iv}' if (iv and 'cbc' in mode) else ''}")
```

This is the low level code that tells the openssl application to encode an input file with the key we will supply it with. The command then will create an output encrypted file based on our supplied output_location. Finally, there is an optional iv key when we will do supply an additional iv value

Decryption

```
os.system(f"{OPENSSL} enc -{mode} -d -in {input_file} -out {output_location} -K {key} {f'-iv {iv}' if iv else ''}")
```

This code is similar to the encryption method we declared earlier. It will ask an input_file, a password key, an optional iv key and the file location where we will save the decrypted file.

High Level Function

These functions are enveloped in a function on a general class where we can easily and intuitively access it elsewhere

```
class AES:
    def __init__(self):···

    def reshuffle_key(self):···

    def encrypt(self, input_file, output_location, key='', iv='', mode="aes-128-ebc"):···

    def decrypt(self, input_file, output_location, key='', iv='', mode="aes-128-ebc"):···
```

The class also has an internal key which it will use if you don't supply an optional key and iv to the program. By defaut, the program will use "aes-128-ecb" encryption.

Hashing Function

For hashing functions, the openssl takes an input which it will then parse to create a hash value. This is a more straightforward approach as you will only have to supply a hashing mode and file input.

Hashing

```
os.system(f"{OPENSSL} dgst {mode} {input_file} {f'> {output_location}' if output_location else ''}")
```

OpenSSL takes a mode of hashing then an input file. If you don't supply an output location to it, openssl will just display the result on the console window. To save the output, we will save the file into a text file supplied in the output location.

High Level Function

```
class SHA:
    def __init__(self):···

    def hash(self, input_file, output_location="", mode="sha256"):···
```

The hashing function was enveloped in a function inside a general class. The function itself takes an input file location. It also takes an optional output location. Finally, it will also ask for a mode of what hashing technique to use, by default, it will use sha256 algorithm

Public Key Encryption – RSA 2048

RSA encryption are based on exclusive encryption between the receciver and the sender. The receiver has a private key which he can use to decrypt any file encrypted with the public key. The encrypted file itself can only be encrypted by the private key and should be protected at any cost.

Creating Private Key

```
os.system(f"{OPENSSL} genrsa {'-aes256' if encrypted else ''} -out
{private_loc} 2048")
```

This is the code that interacts with the openssl which instructs it to create a private key in rsa 2048 format. It can either by encrypted by an aes256 function or not. In case you encrypt it, it will then ask for a password.

Creating Public Key

```
os.system(f"{OPENSSL} rsa -in {private_loc} -pubout -out {public_loc}")
```

This code calls the rsa function of openssl to create a public key from a supplied private key input.

Encryption Method

Its important to note that RSA is only limited to 117 bytes maximum encryption. It is not meant to be used on sending huge file sizes. It can however be used to encrypt keys used to encrypt files who used other encryption method.

Here we will do an encryption by aes-256-cbc which will take a randomized hex value as password, encrypt it by RSA-2048, send it to receiver, decrypt the RSA-2048 encrypted password and use the decrypted password to decrypt the aes-256-cbc encrypted file.

```
os.system(f'{OPENSSL} enc -p {encrypt_method} -salt -in {input_file} -
out {output_loc} -pass file:./{key}')
```

We encrypt here the file with our encryption method of choice, add salt and a password file. This code should take an input file to be encrypted by the encryption method by choice and produce an output encrypted file.

```
os.system(f'{OPENSSL} rsautl -encrypt –inkey {public_key} -pubin -
in {key} -out {return_value}')
```

In this step, we will RSA-encrypt the password file we used to encrypt the file we will send to the receiver.

```
os.system(f'{OPENSSL} rsautl -decrypt -inkey {private_key} –in
{encrypted_data[0]} -out {decrypted_key}')
```

In this phase, the said file was assumed to be the receivers side. The receiver will decrypt the password file with its own private key.

```
os.system(f'{OPENSSL} enc -d -p {encrypted_data[1]} -salt -in {input_file} -out {output_loc} -pass file:{decrypted_key}')
```

Then, the decrypted password file was used to decrypt the actual file.

High Level Function

We want to envelop this tasks in a neat and easy to understood way.

```
class RSA:
    def __init__(self):...

    def create_keys(self, private_loc="", public_loc="", newprivate=True, encrypted=True):...

    def create_public_key(self, private_loc, public_loc):...

    def create_private_key(self, private_loc, encrypted=True):...

    def encrypt(self, input_file, output_loc, public_key, key="", encrypt_method="-aes-256-cbc"):...

    def decrypt(self, input_file, output_loc, private_key, encrypted_data):...
```

The first expected method is to create a private and public key to be used on later RSA encryption and decryption. The private key can either be encrypted with password or not.

Then in encryption, the function will take an input file, encrypt it with public key with the encryption method of choice (default is aes-256-cbc method). This encryption method will return an array of two: the encrypted key file location and the method of encryption used.

For decryption, it takes an encrypted input file, the desired output location, the private key to be used in decryption and the encrypted data (this is the return value of encrypt function).

Signature and Verification – ECDSA

This method was used to verify the validity of a file sent by the host (private key holder) by testing it agains the member's key (public key).

Thus this require that both the host and members have a private and public key. This process will create a signature of file that the host created and that will be checked by the members.

Creating the Private Key

```
os.system(f'{OPENSSL} ecparam -genkey -name {modeName} -noout -out {private_loc}')
```

The code above will create a unique private key for the host. It is a straightforward process.

Creating the Public Key(s)

```
os.system(f'{OPENSSL} ec -in {private_loc} -pubout -out {public_key}')
```

The code above should create unique public keys that can be used to verify signature and files coming from the host.

File Signing

```
os.system(f'{OPENSSL} dgst -sha256 -sign {private_key} < {input_file} {f"> {output_certificate}" if output_certificate else ""}')
```

This code will create a sha256 hash function signed by a private key (input_file) which will then create an output_signature on your location of choice.

File Signature Verification

```
os.system(f'{OPENSSL} dgst -sha256 -verify {public_key} -signature {check_certificate} < {input_file}')
```

This code will take a signature file and public key which it will then test against a file who was supposed to be the hash source.

High Level Function

```
class ECDSA:
    def __init__(self):...

    def get_modes(self):...

    def ecdsa_createKey(self, private_loc="", public_key="", newprivate=True, modeName="secp384r1"):...

    def ecdsa_create_public_key(self, private_loc, public_key):...

    def ecdsa_create_private_key(self, private_loc, modeName="secp384r1"):...

    def ecdsa_certify(self, input_file, private_key="", output_certificate=""):...

    def ecdsa_verify(self, input_file, public_key="", check_certificate=""):...
```

All of this functions was wrapped in a class. The first logical step in ECDSA signing is to create keys, this can be done by calling ecdsa_create_public_key and ecdsa_create_private_key. The later will ask for an optional mode to use for private key creation, the default value to be used is “secp384r1”.

The private key can be used to create a hash signature of file. The public key then can be used to check the signature file generated by file to check if it’s a legitimate file from the host itself.

PHASE 3 : Serving the Functions in a dinner plate

Our goal here is to create a quick main menu where the user can use the functions without the hassle of supplying the parameters. This is also aimed to be use as the demo interface for when we test the actual functions we created.

```
class MainMenu:
    def __init__(self):···

    def title(self):···

    def _question(self, question, answers=[], question_type="choice"):··

    def ui_main(self):···

    def ui_aes(self):···

    def ui_sha(self):···

    def ui_rsa(self):···

    def ui_ecdsa(self):···
```

The program will have a title followed by a main menu where the user will be asked of what function to be executed. Then , depending on the user's choice, the program will be redirected to the respective function ui.

```
+-+-+-+-+-+-+-+-+-+
|p|y|O|P|E|N|S|S|L|
+-+-+-+-+-+-+-+-+-+


----------

What encryption method to execute

   [0] AES
   [1] SHA
   [2] RSA
   [3] ECDSA


Input your choice: 0

----------

What AES Mode to Use

   [0] aes-128-ecb
   [1] aes-128-cbc
   [2] Quit


Input your choice: 0

----------

What to do

   [0] Encrypt
   [1] Decrypt
   [2] Quit


Input your choice: 0

----------

Process what file

   [0] Sample Image
   [1] External File


Input your choice: 0
Insert key password: 12345
hex string is too short, padding with zero bytes to length
```

CHAPTER 3 : MAKING SENSE OF THE OUTPUTS

Pre-requisites

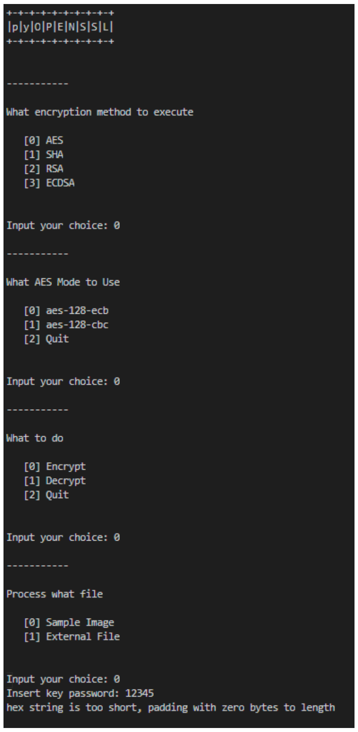
For all of the testing, we will use a single image file (lena512.tiff) for both encryption and decryption, hashing and signing. The report below should provide a general detail of what happened and try to make sense why the said output was achieved.

Part 1 : Symmetric Encryption

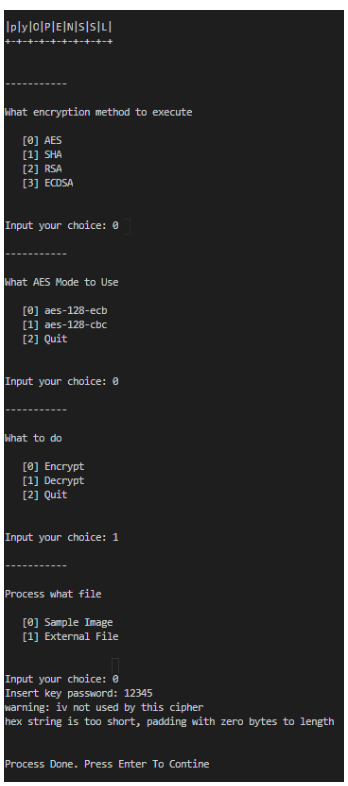
The goal of the symmetric encryption is to encrypt a certain file and decrypt it by using a password only available on the sender and receiver.

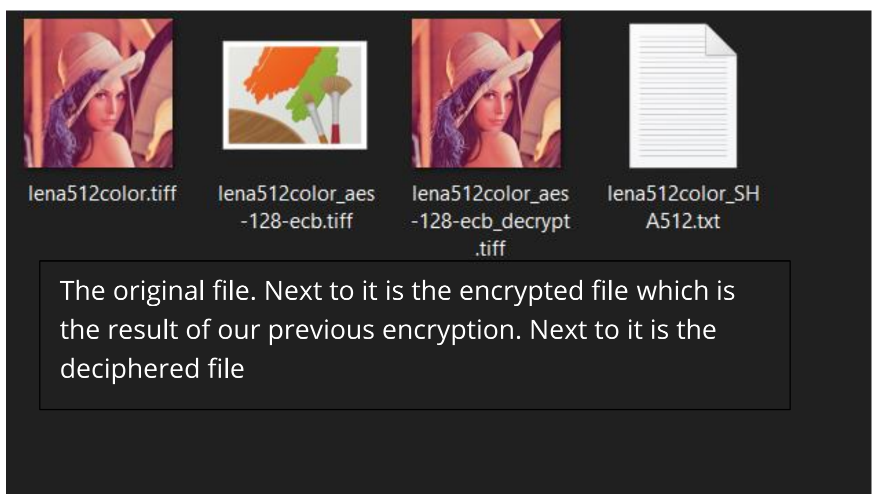
The images below will show you the ui interactions done with the program and the subsequent file changes that happened in its WORK_AREA (which can be accessed by double clicking the “INPUT_OUTPUT Files” shortcut.

Encrypting a file – AES 128 ECB

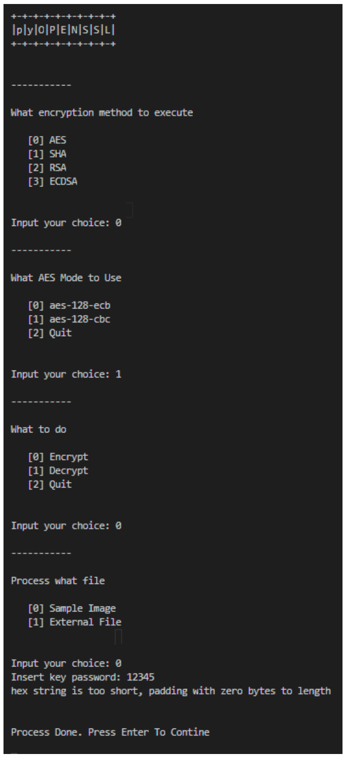


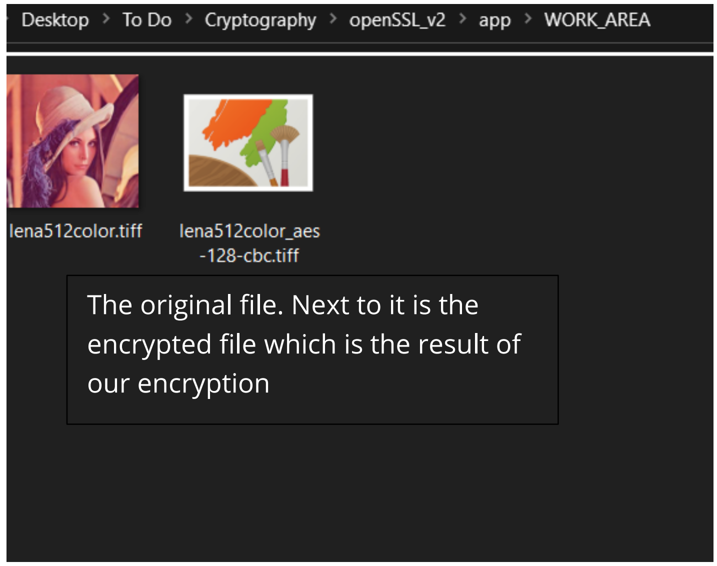
Decryption – AES 128 ECB



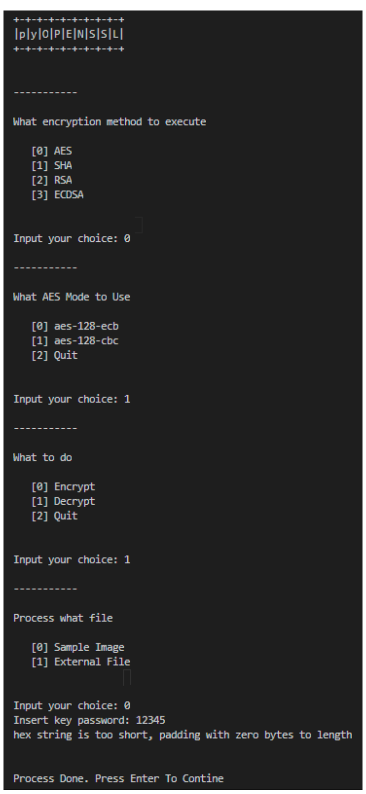


Encryption – AES 128 CBC





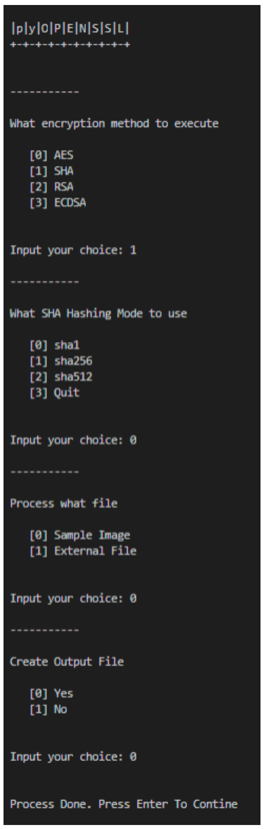
Decryption – AES 128 CBC

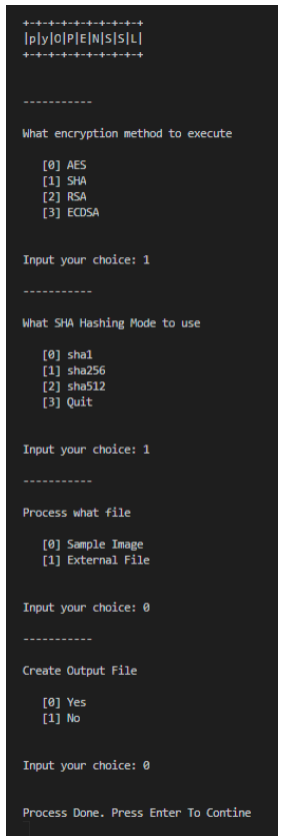


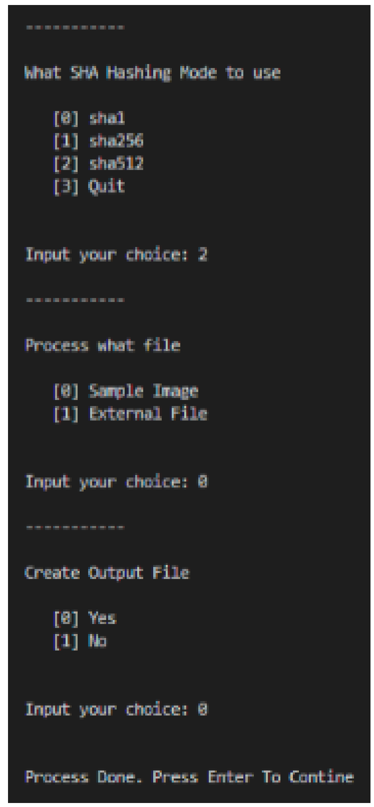
Part 2: Hashing

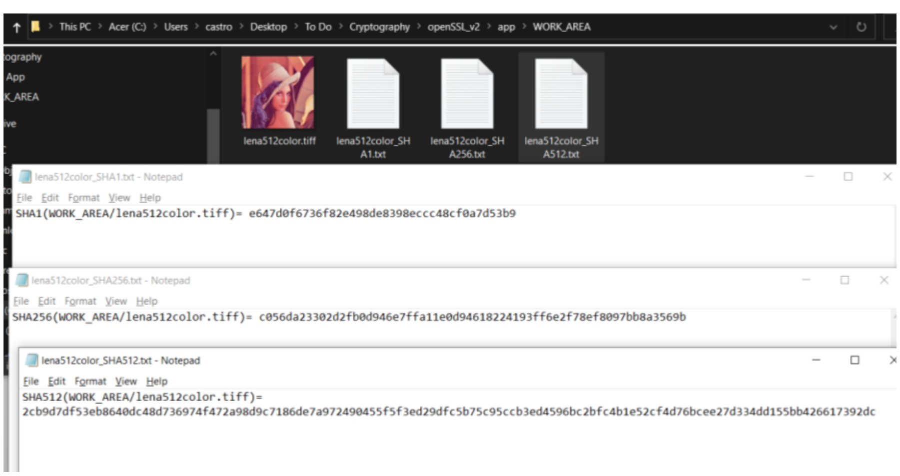
The goal here is to create a hash value for the image sample we have. We will use sha1, sha256 and sha512 here.

SHA 1 / SHA 256 / SHA 512 Hashing







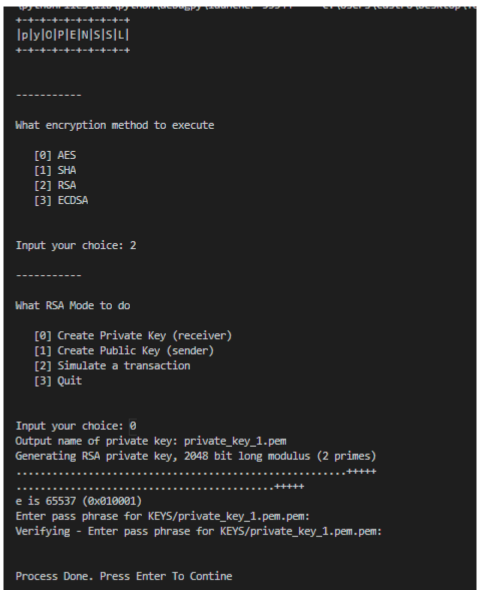


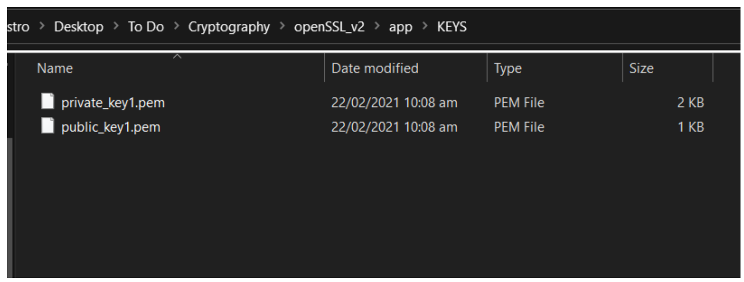
I did the three functions on one run and all of them created theeir own text file containing the hash value

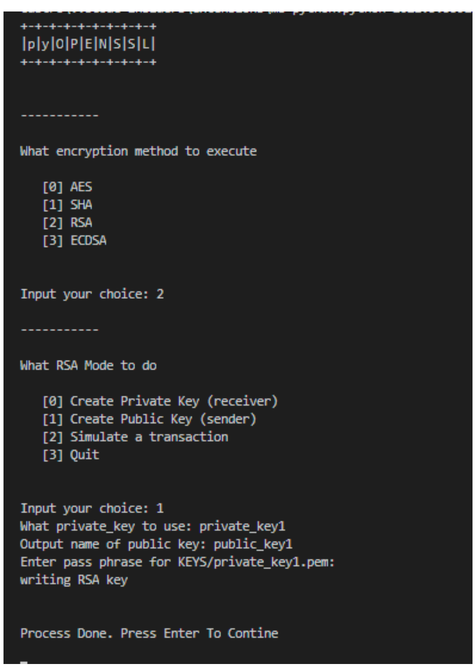
Part 3 : Public Key Encryption

Public Key encryption has two one-way encryption/decryption of data. The receiver will encrypt their data with their public key. The encrypted data then can only be decrypted by the receiver.

Creating Private-Public Key







On the first prompt, a private RSA key was created with AES256 password encryption.

On the second prompt, a public key was created from the private RSA key.

Now, we have a public-private key pair

Simulation of RSA Encrypted - AES Keypass Encrypted – Transfer

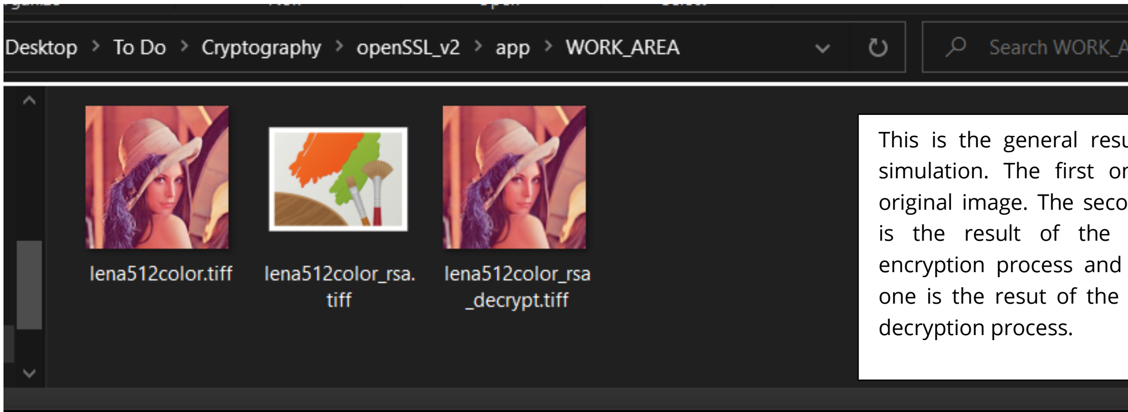
Since RSA Encryption has a maximum supported file size of 117 bytes, full RSA encryption of our image is not possible unless we do some other steps where we can implement RSA encryption on smaller file.

This was done in this program by using the RSA encryption on the AES pass file that contains 32-long random hexadecimal value.

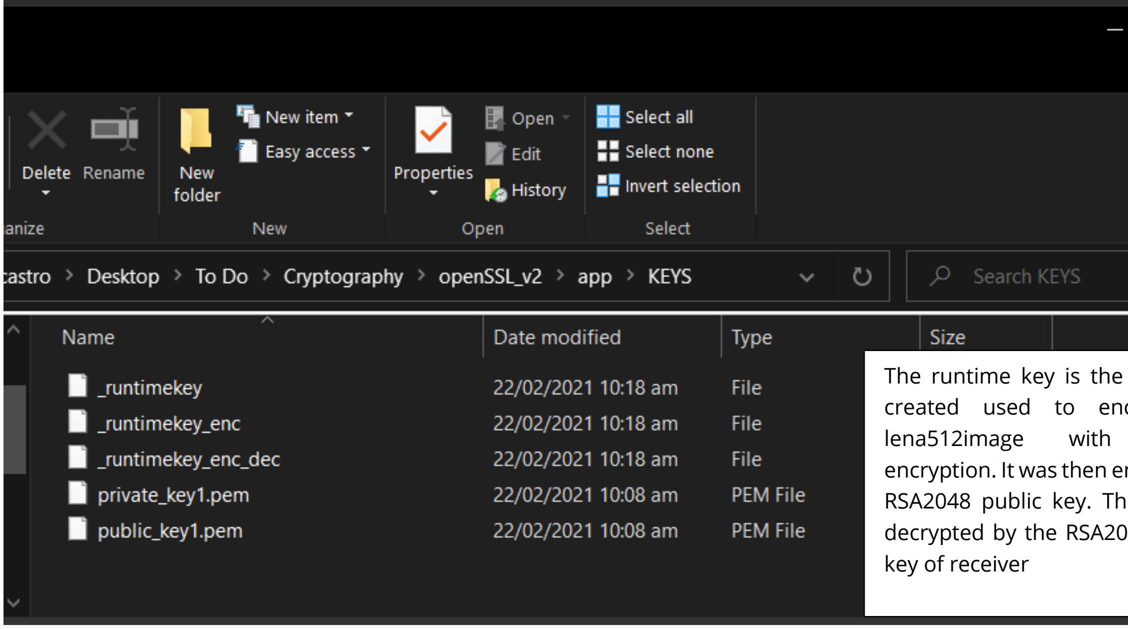
The gist is that the sender will first create a random hex value and save it into a file, use that value file as the pass file when we encrypt our original message with AES-256. Then, the sender will sent the encrypted file and the RSA-encrypted password file into the receiver.

The receiver then can decrypt the RSA-encrypted password file that then will be used to decrypt the AES-256 encrypted image.

```
What encryption method to execute

   [0] AES
   [1] SHA
   [2] RSA
   [3] ECDSA


Input your choice: 2

----------

What RSA Mode to do

   [0] Create Private Key (receiver)
   [1] Create Public Key (sender)
   [2] Simulate a transaction
   [3] Quit


Input your choice: 2


RSA was mostly used for transfering encryption keys because of its small data capability
Here, the goal is to simulate a transaction between a sender with public key and a receiver with private key


What private_key to use: private_key1
What public_key to use: public_key1


Encrypting...


*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
salt=78F02E7AD470BAF6
key=8FA47FAA54AB01808CE13D49F55D7E5BFEC9703FFB33C6D6426A63424F4F4337
iv =2185DB81D36C30EBBE1FFE26675E20D6


Decrypting...


Enter pass phrase for KEYS/private_key1.pem:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
salt=78F02E7AD470BAF6
key=8FA47FAA54AB01808CE13D49F55D7E5BFEC9703FFB33C6D6426A63424F4F4337
iv =2185DB81D36C30EBBE1FFE26675E20D6


Process Done. Press Enter To Contine
```



This is the general result of the simulation. The first one is the original image. The second image is the result of the AES->RSA encryption process and the third one is the resut of the RSA->AES decryption process.



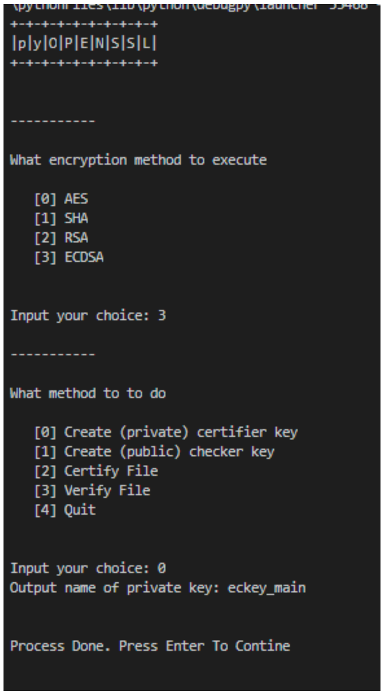
The runtime key is the password created used to encrypt the lena512image with AES256 encryption. It was then enrypted by RSA2048 public key. Then, it was decrypted by the RSA2048 private key of receiver

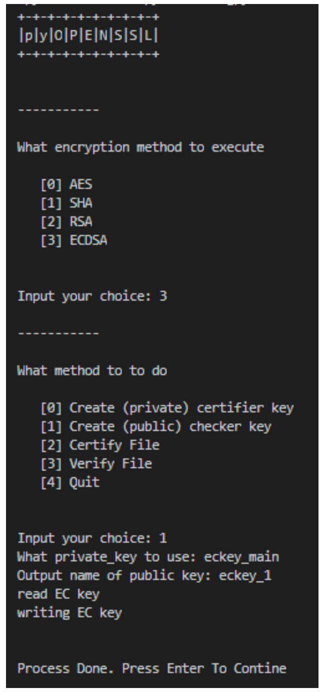
Part 4 : Digital Signing and Verification

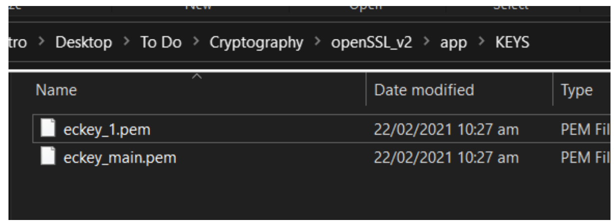
The goal of this function is for a host and its member to have a way to trust the files coming from the host. The host will hash the files he will sent to the members with his own private key. Now, when he sent the file and the signed hash file into the members, the members can know if that actual file is actually the intended file sent by the host or not.

Creating the private-public key pair

The host will create an ECDSA key with secp384r1 mode. Then with this host key, the app will now create a public key ready for distribution.



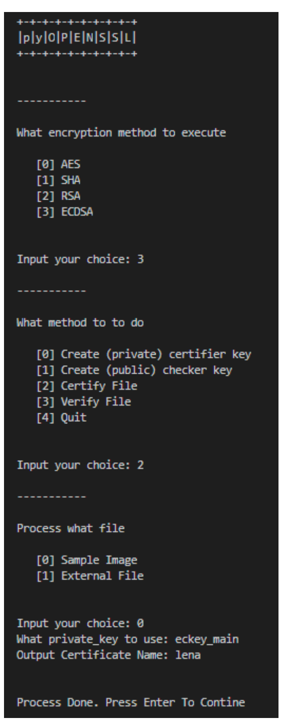


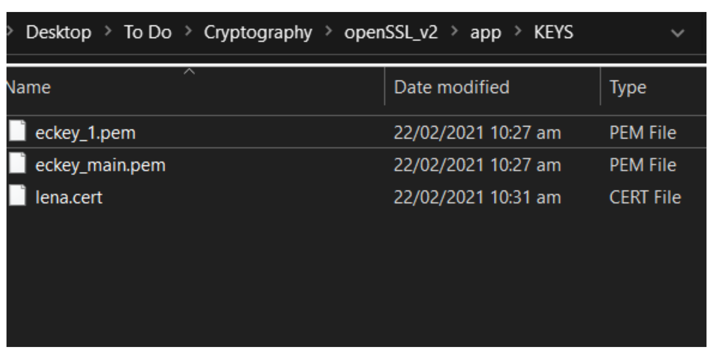


The first picture shows the creation of the private key. The second picture shows the creation of the public key created based on the private key. The third image shows the resulting keys in the KEYS folder.

Signing a File and Verifying it

We first sign a file with our private key.





The left shows the console interface on ECDSA file signing with our created private key. That process created a certificate file on our KEYS folder.

Verification of Signature and File

Now that we have certificate hash to test our file on, lets verify it.

```
+-+-+-+-+-+-+-+-+-+-+
|p|y|O|P|E|N|S|S|L|
+-+-+-+-+-+-+-+-+-+-+


----------

What encryption method to execute

   [0] AES
   [1] SHA
   [2] RSA
   [3] ECDSA


Input your choice: 3

----------

What method to to do

   [0] Create (private) certifier key
   [1] Create (public) checker key
   [2] Certify File
   [3] Verify File
   [4] Quit


Input your choice: 3

----------

Process what file

   [0] Sample Image
   [1] External File


Input your choice: 0
What public_key to use: eckey_1
Certificate File to cross-check: lena
Verified OK


Process Done. Press Enter To Contine
```

Here we validate an ECDSA signed file. It asks for what public key to use, a signature to be used for cross checking and as the result, the program tells us that the verification went ok.

EXTRA: Creating a false verification

To test this, we will create a new private ec_key, then sign the same file again. But instead of creating a new child public key, we will still use the old public key from the previous test.

This process should have a failed verification by the end.

```
----------

What method to to do

   [0] Create (private) certifier key
   [1] Create (public) checker key
   [2] Certify File
   [3] Verify File
   [4] Quit


Input your choice: 0
Output name of private key: eckey_main2


Process Done. Press Enter To Contine
```

```
----------

What method to to do

   [0] Create (private) certifier key
   [1] Create (public) checker key
   [2] Certify File
   [3] Verify File
   [4] Quit


Input your choice: 2

----------

Process what file

   [0] Sample Image
   [1] External File


Input your choice: 0
What private_key to use: eckey_main2
Output Certificate Name: lena


Process Done. Press Enter To Contine
```

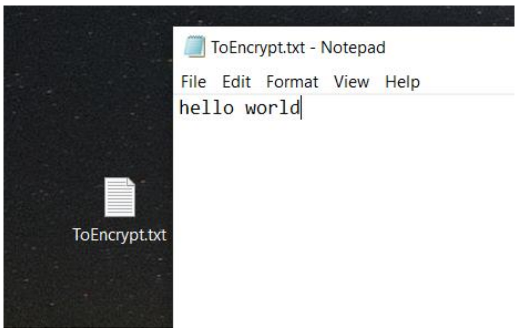
```
----------

What method to to do

   [0] Create (private) certifier key
   [1] Create (public) checker key
   [2] Certify File
   [3] Verify File
   [4] Quit


Input your choice: 3

----------

Process what file

   [0] Sample Image
   [1] External File


Input your choice: 0
What public_key to use: eckey_1
Certificate File to cross-check: lena
Verification Failure


Process Done. Press Enter To Contine
```

Part 5 : Processing external files

Instead of the sample image (lena512color.tiff), you can also process other files for our symmetric encryption, hashing and ecdsa functions.

Prerequisites

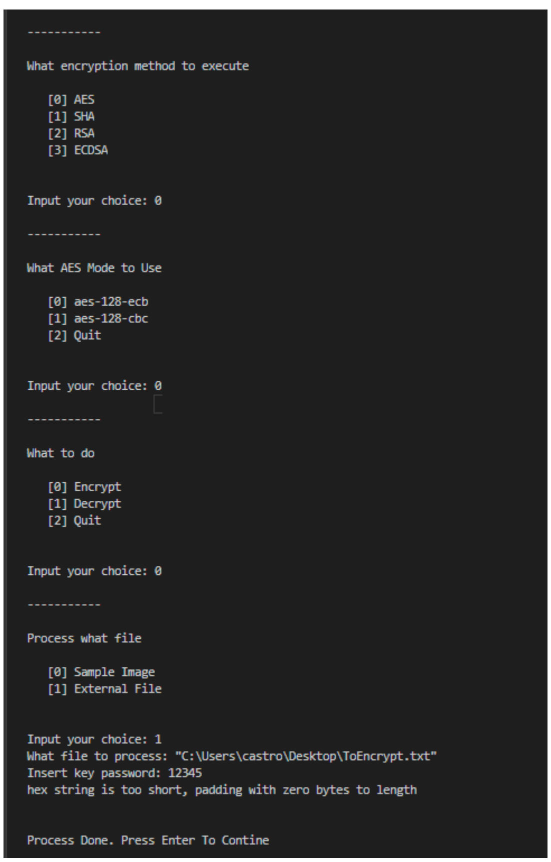
In these processes, we will use a text file outside our ordinary working place.

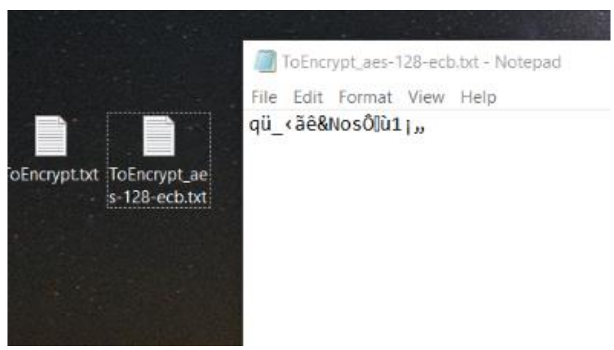


We will take the path of this file by Shift+Right Click and picking “Copy as Path”

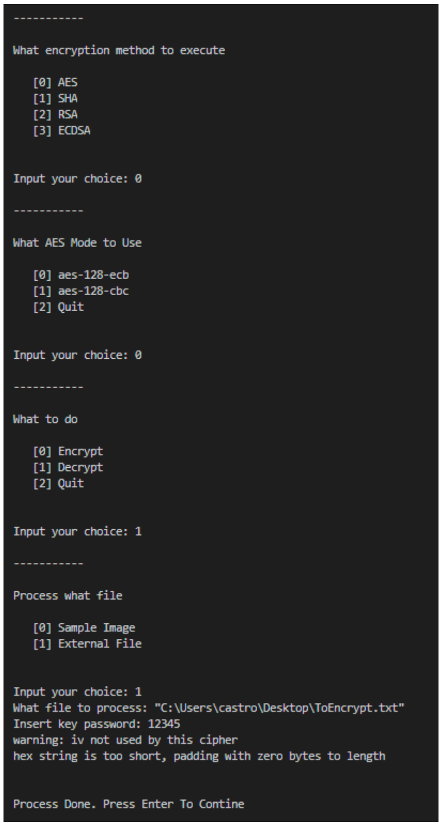
"C:\Users\castro\Desktop\ToEncrypt.txt"

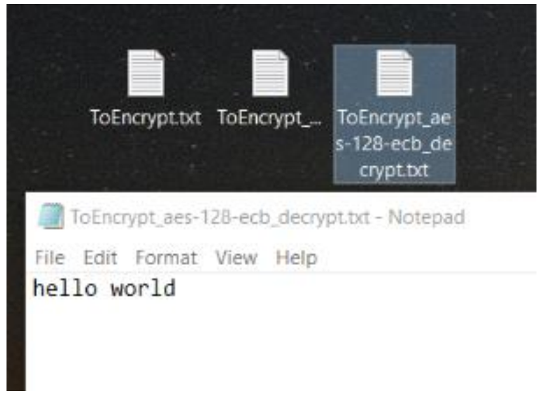
Example





The left picture shows the transaction of encryption. We supplied our external file enveloped by a double quote. The picture above shows the resulting file.





The left picture shows the transaction of decryption. We supplied our external file enveloped by a double quote. Note that we still supplied the original file name, it will automatically search for the encrypted file in that working space (given that you didn't changed its file name).

The external file feature is still not yet a full pledged feature here

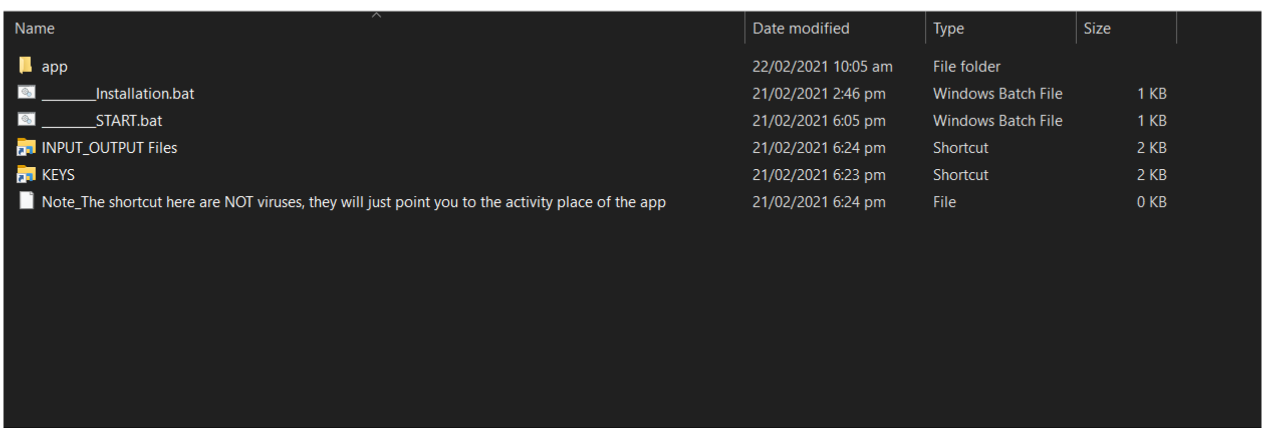
CHAPTER 4 : HOW TO INSTALL AND RUN THE APP

What you need

- A 64 bit Windows PC (preferably Windows 10)
- Latest Python 3.x installed

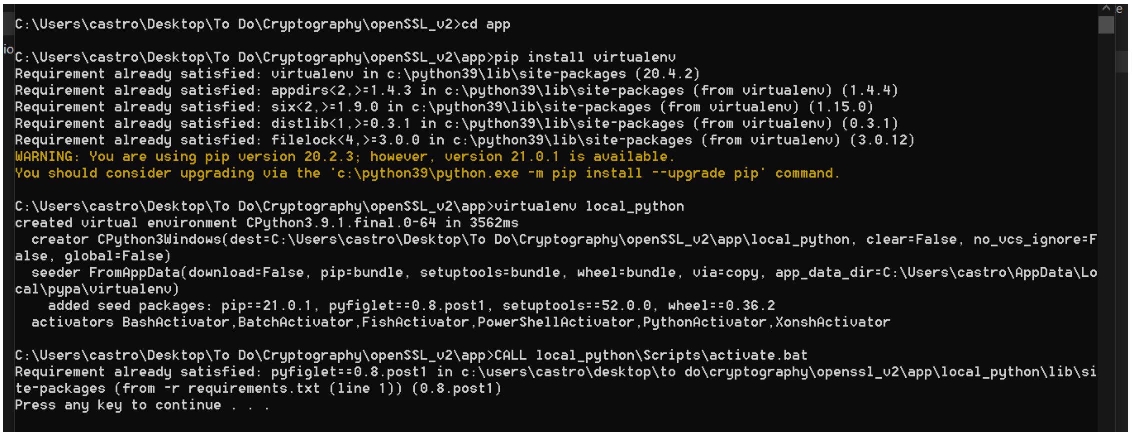
How to install

1. Assumably, you have all the files in the github repository, that's why you're reading this line.
2. Paste the compressed files in a single folder. The target folder after decompression should contain the following files:



Name	Date modified	Type	Size
app	22/02/2021 10:05 am	File folder	
_______Installation.bat	21/02/2021 2:46 pm	Windows Batch File	1 KB
_______START.bat	21/02/2021 6:05 pm	Windows Batch File	1 KB
INPUT_OUTPUT Files	21/02/2021 6:24 pm	Shortcut	2 KB
KEYS	21/02/2021 6:23 pm	Shortcut	2 KB
Note_The shortcut here are NOT viruses, they will just point you to the activity place of the app	21/02/2021 6:24 pm	File	0 KB

3. First, double-click "______Installation.bat", this should create a virtual environment where our python script can run.



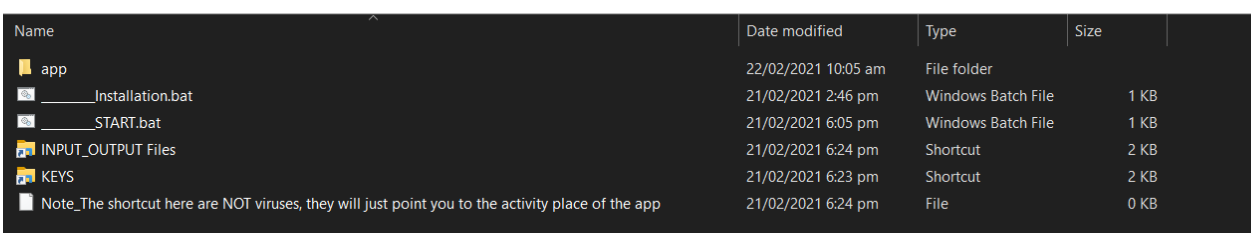
```
C:\Users\castro\Desktop\To Do\Cryptography\openSSL_v2>cd app

C:\Users\castro\Desktop\To Do\Cryptography\openSSL_v2\app>pip install virtualenv
Requirement already satisfied: virtualenv in c:\python39\lib\site-packages (20.4.2)
Requirement already satisfied: appdirs<2,>=1.4.3 in c:\python39\lib\site-packages (from virtualenv) (1.4.4)
Requirement already satisfied: six<2,>=1.9.0 in c:\python39\lib\site-packages (from virtualenv) (1.15.0)
Requirement already satisfied: distlib<1,>=0.3.1 in c:\python39\lib\site-packages (from virtualenv) (0.3.1)
Requirement already satisfied: filelock<4,>=3.0.0 in c:\python39\lib\site-packages (from virtualenv) (3.0.12)
WARNING: You are using pip version 20.2.3; however, version 21.0.1 is available.
You should consider upgrading via the 'c:\python39\python.exe -m pip install --upgrade pip' command.

C:\Users\castro\Desktop\To Do\Cryptography\openSSL_v2\app>virtualenv local_python
created virtual environment CPython3.9.1.final.0-64 in 3562ms
  creator CPython3Windows(dest=C:\Users\castro\Desktop\To Do\Cryptography\openSSL_v2\app\local_python, clear=False, no_vcs_ignore=F
alse, global=False)
  seeder FromAppData(download=False, pip=bundle, setuptools=bundle, wheel=bundle, via=copy, app_data_dir=C:\Users\castro\AppData\Lo
cal\pypa\virtualenv)
    added seed packages: pip==21.0.1, pyfiglet==0.8.post1, setuptools==52.0.0, wheel==0.36.2
  activators BashActivator,BatchActivator,FishActivator,PowerShellActivator,PythonActivator,XonshActivator

C:\Users\castro\Desktop\To Do\Cryptography\openSSL_v2\app>CALL local_python\Scripts\activate.bat
Requirement already satisfied: pyfiglet==0.8.post1 in c:\users\castro\desktop\to do\cryptography\openssl_v2\app\local_python\lib\si
te-packages (from -r requirements.txt (line 1)) (0.8.post1)
Press any key to continue . . .
```

4. Then, after installation, double-click "_____Start.bat". This should open a console interface.

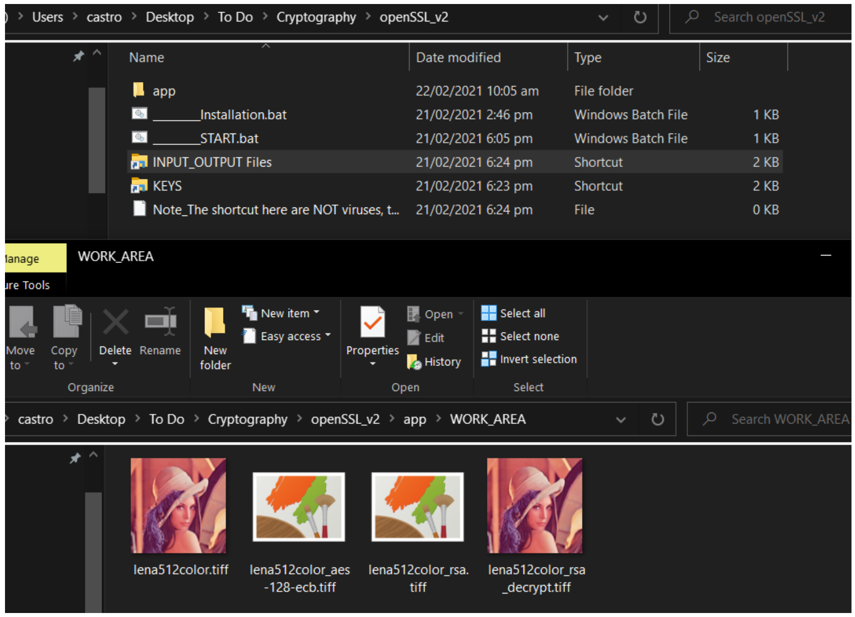
```
-----------

What encryption method to execute

   [0] AES
   [1] SHA
   [2] RSA
   [3] ECDSA

Input your choice:
```

From this point, you can choice to replicate the test cases done on the previous chapter.

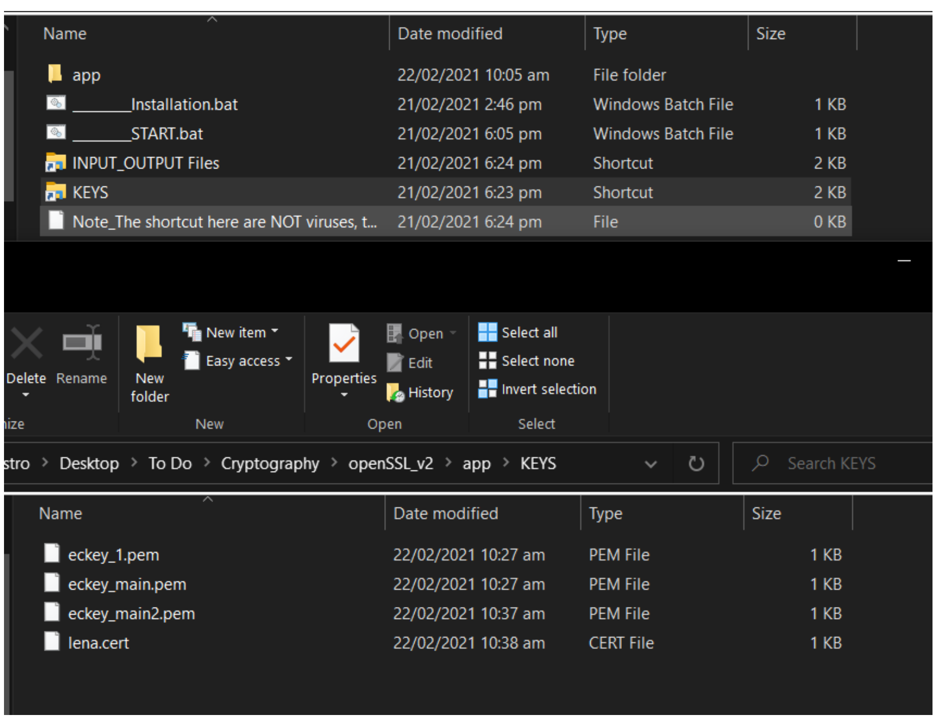
To view the file images being encrypted and decrypted, go to the root directory and double click the "INPUT_OUTPUT Files" shortcut icon. It should open a new window to the images encryption and decryption working directory.



Name	Date modified	Type	Size
app	22/02/2021 10:05 am	File folder	
______Installation.bat	21/02/2021 2:46 pm	Windows Batch File	1 KB
______START.bat	21/02/2021 6:05 pm	Windows Batch File	1 KB
INPUT_OUTPUT Files	21/02/2021 6:24 pm	Shortcut	2 KB
KEYS	21/02/2021 6:23 pm	Shortcut	2 KB
Note_The shortcut here are NOT viruses, they will just point you to the activity place of the app	21/02/2021 6:24 pm	File	0 KB



To view the keys and signatures generated by the application, double click the "KEYS" shortcut icon then. It should open a new window containing all the keys used by the application



CHAPTER 5: FINAL NOTES

Reflection

This is a fun and intersting project, you got to learn how the encryption works especially on the web side of things. However, this project is a bit irritating to work on because of its documentation, it's either lacking or just too complex for my pea brain to comprehend. Had to scour the internet for hours looking for stackoverflow or other programming related websites to see how to do __________. And often, the steps given in this site are 'just right' but sometimes, lacks somes information vital to the whole process like "Where the hell can I create the private and public key pair?". In the end, I managed to make it all work. I understand some part of it better than others but at least, all of them makes sense to me.

I suggest to others to use other newer cryptography application available out there. Those newer tools might be easier to use and has more easy to understand and comprehensive documentation.

But I won't say the learning OpenSSL on barebones is not important, OpenSSL is prevalent on the software industry and was used by many big institutes in their softwares, so learning it surely has a plus side.

References

- https://stackoverflow.com/questions/22856059/openssl-ecdsa-sign-and-verify-file
- https://linuxconfig.org/easy-way-to-encrypt-and-decrypt-large-files-using-openssl-and-linux
- https://www.javacodegeeks.com/2020/04/encrypt-with-openssl-decrypt-with-java-using-openssl-rsa-public-private-keys.html
- https://www.openssl.org/docs/manmaster/man1/
- https://www.openssl.org/docs/manmaster/man1/openssl-genrsa.html
- https://www.openssl.org/docs/manmaster/man1/openssl-enc.html
- https://www.openssl.org/docs/manmaster/man1/openssl-ec.html
- https://www.openssl.org/docs/manmaster/man1/openssl-dgst.html