This assignment focuses on cryptography. The assignment itself is fully explained in crypto.pdf. python3 cryptography library is used for this assignment. The file that needs the code is called cryptomail_skel.py. In a number of places the code must be completed in the skeleton file. This is indicated by the following lines in the code of the skeleton file:

Student Work {{

#your code

#Student work }}

The program does not need changes elsewhere. Please leave comments on the code you have written.

Follow the next guideline to start the assignment

1. Create the `${studnr}.prv` , `${studnr}.pub` and `${studnr}.crt` files with openssl (or python)
2. Copy `cryptomail_skel.py` to `cryptomail.py`
3. Run `python3 cryptomail.py -f test0.xmbox decode` , this should work
4. Complete the standard methods as described below
5. If these standard methods work well, implement decode and then encode

Run the testfiles to check if the code is implemented correctly. Details are described within the crypto.pdf under chapter To test. If you have any confusion on how any method or class works, please refer to the crypto.pdf where each is described fully on its functionality

You must generate your own private key, public key and certificate for this assignment. These have the ending .priv, .pub and .crt. The assymentrix keys must be **1024** bit rsa keys. You can generate the keys with openssl. Please give a description on how this is done, so I can implement it myself when you are done.

You must also create a self-signed certificate based on the key number so the files should be named:

`${studNr}.prv` , for example `123456789.prv`

`${studNr}.pub` , for example `123456789.pub`

`${studNr}.crt` , for example `123456789.crt`

These functions must be implemented within the python file:

- **loadPrvKey**
  - This method loads a Private key in pem format from file. The file name is the name followed by .prv . The privateKey (type rsa._RSAPrivateKey) is stored in self.prvs[name]
- **loadPubKey**
  - The code to retrieve the public key from a certificate has already been given. You only need to write the code for a .pub file. This method loads a public key in pem format from a file. The filename is the name followed by .pub. The publicKey (type rsa._RSAPublicKey) is stored in self.pubs[name] .
- **genSesKey**
  - Generate a proper sessionKey (type bytes) and assign it to self.sesKey The parameter **n** specifies the key length in bytes
- **genSesIv**
  - Generate a proper sessionIntialVector (type bytes) and assign it to self.sesIv . The parameter n specifies the iv length in bytes.
- **encryptSesKey**
  - Encrypt the sessionKey (self.sesKey) with the appropriate public or private key for person user . It was previously stored in self.prvs or self.pubs by the loadPrvKey or loadPubKey methods. The protected key (encKey) is stored in self.rcvs[user] or self.snds[user].

    Note

    The sessionKey (sesKey) and the protected sessionKey (encKey) are of type bytes. Note The self.mode
    class variable specifies how to encrypt/decrypt.
- **decryptSesKey**
  - This method decrypts the protected sessionKey (encKey) with the correct key for person user . It was previously stored in self.prvs or self.pubs by the loadPrvKey or loadPubKey methods. The protected sessionKey (encKey) is stored in self.rcvs[user] or self.snds[user] .
- **encryptMesg**
  - This method does the symmetric encryption based on the cryptographic methods specified in it self.modes . The plainBytes are in self.mesg and the encrypted data must be stored in self.code both are of type bytes.

    Note

    The cm.mode indicates how to encrypt/decrypt

- **decryptMesg**
  - This method does the symmetric decryption based on the cryptographic methods specified in self.modes . The encrypted data is in self.code and the decrypted data must be stored in . self.mesg ,both are of type bytes

    Note

    The cm.mode indicates how to encrypt/decrypt.

- **signMesg**
  - This method signs the message self.mesg for user . The key is in self.prvs or self.pubs. It was previously stored in self.prvs or self.pubs by the loadPrvKey or loadPubKey methods. The data to be signed is in self.mesg and the signature (type bytes) is stored in self.rcvs[user] or self.snds[user]
- **verifyMesg**
  - This method verifies the signature (self.rcvs[user] or self.snds[user]) for user with the message self.mesg. Depending on the correctness of the signature True or False is returned. And in other cases None
- **calcHash**
  - This function calculates the hash of the message (self.mesg) and stores it in the CryptoMesg structure (self.dgst)

    Note

    The cm.mode indicates how to hash.
- **chckHash**
  - This function checks whether the hash (self.dsgt) matches the message (self.mesg). If the hash is correct, the return value True otherwise False
  -

    Note

    The cm.mode indicates how to hash.

When all the methods mentioned above have been implemented, you have implemented the basic functionality.

- **Encode**

The encode method will store a CryptoMesg message in the cmFname file with cm.dump

There are four pieces of code to be entered/programmed. To do this, use the routines you implemented earlier..

0) Conversion

The string to be encoded is a unicode string, it must be converted to bytes

1) Encrypt

If there are receivers/senders (think which of the two) the message must be encrypted. This includes creating session key and session iv, encrypting the message (cm.mesg) and protecting the session key.The encrypted message comes in cm.code. You store the protected keys in the dict (cm.rcvs or cm.snds) with the user as the key and the protected key (bytes) as the value.

Use the helper routines: genSesKey , genSesIv , encryptSesKey , encryptMesg and possibly others

2) Hash

Calculate the hash with calcHash This is stored in cm.dgst by the method

3) Sign

If there are recipients/senders (think which of the two) the message (cm.mesg) must be signed. You store the signatures in the dict (cm.rcvs or cm.snds) with the user as the key and the signature (bytes) as the value.

Use the helper routines: signMesg and possibly others.

4) Strip

There is too much (confidential) information in the CryptoMesg class. These class variables must be set to None to prevent them from being written to the file.

Attention

The files `test1.xmbox` , `test2.xmbox` , `test3.xmbox` still contain such secrets.

A) Return

Fill the receiverState and senderState dict for each specified receivers (receivers) and senders (senders) with one of the following
values:

- None: no information available to encrypt/sign True: operation
- True: Operation succeeded
- False: Operation Failed

These two dicts form the return tuple:

return(senderState, receiverState)

- **decode**

The decode method will read a CryptoMail message from the file cmFname with cm.load .

Four pieces of code must be entered. To do this, use the routines you implemented earlier.

1) Secrets

It is possible that the read file contains information that should have been kept secret (ie should not have been sent with it). The secretState variable has no secrets leaked.

2) Decrypt

For the specified recipient/sender check if the message can be decrypted. Make sure that the decrypted message is in cm.mesg. In addition, it must be indicated whether for each recipient/sender the message

Use the helpers: decryptSesKey , decryptMesg and possibly others..

3) CheckHash Check if the hash of cm.mesg is correct. Use the helper routines checkHash and possibly others. The variable hashState indicates whether or not the hash is correct. This value is None if the hash is not verifiable.

4) Verify For the specified senders/recipients, check whether the message is signed correctly. Use cm.mesg as the message to check, remembering where the signature is stored. In addition, it must be indicated whether the message is readable for each recipient/sender..

Use the helper routines: verifyMesg and possibly others.

A) Return

Enter the dict `secretState` , `receiverState` , `hashState` and `senderState` for each specified recipients (`receivers`) and senders (`senders`) with one of the following values:

- None: no information available to encrypt/verify
- True: Operation succeeded
- False: Operation Failed

The conversion from bytes to str is done for you. The message (mesg) and the four states form the return tuple

```
return (senderState, receiverState)
```

Run the following unix commands to create the files ${studNr}S.xmsg and ${studNr}M.xmsg::

```
# use your studnr
$ studnr="123456789"
$ echo "Dit is de geheime tekst van ${studnr}." > ${studnr}.txt
$ python3 cryptomail.py -f ${studnr}S.xmsg -r user1 -s user3 -m ${studnr}.txt encode
$ python3 cryptomail.py -f ${studnr}M.xmsg -r user1,${studnr} -s user3,${studnr} -m
${studnr}.txt
```

It is useful if you check things with:

```
$ python3 cryptomail.py -f ${studnr}S.xmsg decode
$ python3 cryptomail.py -f ${studnr}M.xmsg decode
```

Finally the following files need to be gathered:

1. Certificate (`${studnr}.crt)
2. CryptoMesg-file (`${studnr}S.xmsg`) and (`${studnr}M.xmsg`)
3. Your program (`cryptomail.py`)