Assignment Cryptography

Introduction

The final assignment consists of a number of parts with which you indicate that you can apply cryptography using Python.

The document refers to recipients/senders (receivers/senders). Depending on the purpose, this means either recipients or senders. It's up to you to choose the right one.

The parts are:

- Generating a private key, public key and your certificate with your public key [crt 1-2].
- Being able to use the standard cryptographic functions, such as hash, symmetric encryption/decryption, asymmetric encryption/decryption and signing/verification [crt 3-7].
- Applying these techniques in a cryptographic problem: CryptoMail [crt 7-16]
- Other requirements [crt 17-20]

We use the python3 cryptography library, see the documentation at: https://cryptography.io

CryptoMail

This assignment simulates S/MIME and PGP. These are two standards for exchanging encrypted and signed mail. Most email programs have support for these techniques. The real implementation of S/MIME and/or PGP is quite complex. This simple version does have all the functionality to exchange messages. With the encode option you create an .xmsg file with the protected message. You can send the .xmsg file via email and the other party can then read it with the decode option of the program. However, it is necessary to transport the private keys or certificate in a secure manner once. This allows the validity of the public key/certificate to be determined.

With a CryptoMail, a message is stored encrypted and signed. As a result, the message cannot be read by others (just authorized persons), but only by a certain number of intended recipients/senders. It also offers the possibility to sign the message by a number of recipients/senders. The intended receiving/sending party can verify
the read message from the recipient/senders.

Note

The senders and/or recipients may be missing, then the message is not signed and/or encrypted. The hash is also in calculated in this case.

Part of the assignment consists of creating the necessary keys and completing/completing a skeleton program. 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 {{
  # Vervang dit door jouw code
# Student Work }}
```

The program does not need changes elsewhere, if you think it is necessary, look at the code again, because changing existing code is **not** necessary.

Python Type Hints

The methods have a signature (form of the method): We use Python Type Hints for this. Here we give after the name of the argument after the type of the variable.

```
def meth(arg1: int, arg2: str) -> bool:
```

This indicated that the first argument must be an integer, the second must be a string, and the result of the method must be a boolean. https://docs.python.org/3/library/typing.html

Description

The task is to create a program cryptomail.py with which you can transfer a text message (mesg) protected with mail. The sender uses this program to create a file in which the text message is stored in a protected manner. This message can be transferred to the recipient. This program can be used by this recipient to read the text message and determine a number of properties of the text message.

The functionality is:

- The message can be encrypted for a number of intended recipients/senders, only they can read the text message..
- The message can be signed by a number of recipients/senders, and it can be checked whether the signatures are correct.

Two methods need to be written for this, encode and decode

encode

The encode method encrypts the text message so that a number of recipients/senders can read the message, the hash of the message is determined and the message is signed by the recipients/senders. The required information can be retrieved from the CryptoMail instance (cm), the results obtained must also be stored there. For example, the encrypted message in cm.code.

The signature of these important methods is:

```
def encode(cmFname: str, mesg: str, senders: list=None, receivers: list=None) -> tuple:
    """ Encode (encrypt and/or sign) the message (`mesg`) for the `receivers` and `senders`.
        The cryptomesg-structure (CryptoMesg) is created and filled with encode-information
        and written to the file `cmFname`.
        Returns a tuple (sendersState, receiversState).
    """
```

decode

The decode method decrypts the message for the recipients/senders. It also indicates which recipient/sender has signed the message and whether the signature is correct.

The signature of these important methods is:

```
def decode(cmFname: str, receivers: list=None, senders: list=None) -> tuple:
    """ Decode (decrypt and/or verify) the coded message `cmFname` for the `receivers` and `sende
        The cryptomeg-structur (CryptoMesg) is read from the file `cmFname` and
        used to decode de containing message.
        Returns a tuple (mesg, sendersState, receiversState, secretState).
    """
```

The assignment

Create asymmetric keys

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.

You must also create a self-signed certificate based on the key .
The subject field should be: `"/CN=${email}@email.nl/C=EN/O=CM/OU=ICT/ST=CB/L=Cambrigde"` . Here is `${email}@email.com` your email adress. The three files start with your student 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`
-

CryptoMesg

This class forms the main data structure of the program. The different fields are used in the different helper methods and they also create new values in this structure. After encode, the structure (class-instance) contains all the necessary information to get the message back with decode, as well as information about the signature.

There is a serialization of this structure (class-instance) as a json structure. The routines CryptoMesg.load and CryptoMesg.dump are used to read and write this json structure.

The CryptoMesg class has the following fields: See also the comments in the skeleton program. Some fields are only needed during the cryptographic process. These values should be set to None to prevent information from being leaked while saving this structure.

`def __init__(self) -> None :`

This method initializes all (class) variables. See the explanation of the instance variables below.

```
    def __init__(self) -> None:
        """ Initialise the used variables """
        self.version = '1.0'    # Version number
        self.modes   = []       # Specifies the used algorithms
        self.snds    = {}       # keys: names of senders, values: relevant data
        self.rcvs    = {}       # keys: names of receivers, values: relevant data
        self.sesIv   = None     # (optional) session Iv     (bytes)
        self.sesKey  = None     # (optional) session key    (bytes)
        self.prvs    = {}       # keys: names of user, values: prvKey-object
        self.pubs    = {}       # keys: names of user, values: pubKey-object
        self.code    = None     # (optional) the encrypted message  (bytes)
        self.mesg    = None     # (optional) the message            (bytes)
        self.dgst    = None     # (optional) the hash the message   (bytes)
```

`CryptoMesg.version`

It is useful to include a version number in the structure. This is currently 1.0 .

`CryptpMesg.modes`

The message can be encrypted and/or signed

- The mode indicates whether the message is encrypted and which cryptographic methods are used. encryptEnv and decryptEnv
- The mode also indicates whether the message has been signed and which cryptographic methods have been used.
- The mode also indicates whether a hash has been calculated and which cryptographic methods have been used

For this assignment, the following modes are used, in which the various parameters for the cryptographic operations are determined

```
    'hashed:sha256'
    'signed:rsa-pss-mgf1-sha384',
    'crypted:aes256-cbf:pkcs7:rsa-oaep-mgf1-sha256',
```

`CryptoMesg.snds`

This is a dict (dictionary) with the key of the sender as the value of relevant information from/for the sender. It is a dict because, in theory, a message can be sent on behalf of multiple parties.

`CryptoMesg.rcvs`

This is a dict with a receiver as key and relevant information from/for the receiver as value. It is a dict because, in theory, a message can be read by several parties.

`CryptoMesg.sesIv`

It must be initialized to the correct value (CryptoMesg.genIv). It is used in encryptEnv and decryptEnv. It's a bytes string, not a hex string..

`CryptoMesg.sesKey`

It must be initialized to the correct value (CryptoMesg.genSesKey). This is used in encryptEnv amd decryptEnv. This is a bytes string, not a hex string.

`CryptoMesg.prvs`

This dict contains the privateKey (type rsa._RSAPrivateKey) for a sender/receiver.

`CryptoMesg.pubs`

This dict contains the publicKey (type rsa._RSAPublicKey) for a sender/receiver.

`CryptoMesg.mesg`

This is the unencrypted message (type bytes)

`CryptoMesg.code`

This is the encrypted message (type bytes)

`CryptoMesg.dgst`

This is the hash of the unencrypted message (type bytes)

CryptoMesg (building) methods

The class also has the following already created methods.

`def load(self, cmFname: str) -> None:`

The method loads the file denoted by cmFname and populates the various class variables. The file must be a representation of the CryptoMesg class.

`def dump(self, cmFname: str) -> None:`

The method dumps the various class variables into the file denoted by cmFname . The file is a representation of the CryptoMesg class

`def addMode(self, mode: str):`

Indicates whether the contents of CryptoMesg are encrypted and/or signed. The cryptographic methods used are also indicated. These methods are described in the sheets

`def hasMode(self, mode: str):`

It can be checked whether the contents of CryptoMesg are encrypted, hashed and/or signed. For example, to check whether the message has been signed:

```
if cm.hasMode('signed'): ...
```

CryptoMesg (to do) methodes

All of these routines have a `# Student Work {{ ... # Student Work }}` section that needs to be completed. Other than that, no adjustment is necessary.

`def loadPrvKey(self, name: str) -> None:`

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]

`def loadPubKey(self, name: str) -> None:`

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] .

`def genSesKey(self, n: int) -> None:`

Generate a proper sessionKey (type bytes) and assign it to self.sesKey The parameter **n** specifies the key length in bytes

`def genSesIv(self, n: int) -> None:`

Generate a proper sessionIntialVector (type bytes) and assign it to self.sesIv . The parameter n specifies the iv length in bytes.

`def encryptSesKey(self, user: str) -> bool:`

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.

def decryptSesKey(self, user: str) -> bool:

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] .

def encryptMesg(self) -> bool:

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

def decryptMesg(self) -> bool:

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.

def signMesg(self, user: str) -> bool :

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]

Note

The cm.mode indicates how to sign/verify.

def verifyMesg(self, user: str) -> bool :

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

def calcHash(self,: str) -> None :

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.

def chckHash(self,: str) -> bool :

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. The proper functioning of these methods is testing in a specialized environment.

Application functionality

The application functionality is realized in two methods. encode and decode . The various tests test the operation of the aspects of this method..

- encode

  The encode method ensures that the message mesg is signed and encrypted by the receiving/sending parties so that only the intended parties can read it.

- decode

  The decode method checks whether the specified recipients/senders can decrypt the message and checks whether the message has been signed by the specified parties (users). It is also indicated whether secrets are wrongly revealed in the message,

```
def encode(cmFname: str, mesg: str, senders: list, receivers: list) -> tuple:
```

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)

`def decode(cmFname: str, rcvs: list, snds: list) -> tuple:`

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` en `senderState` for each specified recipients (`receivers`) en 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 (mesg, receiversState, sendersState, hashState, secretState)
```

Contents CryptoEnc.zip

The assignment zip contains the following files:

```
crypto.pdf
cryptomail.py
user1.prv user1.pub user1.crt
user2.prv user2.pub
user3.prv user3.pub user3.crt
user4.prv user4.pub
user5.prv user5.pub
user6.prv user6.pub
test0.xmbox
test1.xmbox
test2.xmbox
test3.xmbox
```

Start

Follow the next guideline to start the problem

> Try to think of everything you need to do on paper first. Make a to-do list and draw the process. What is the role of the senders and recipients, what must be signed and by whom, what must be encrypted and by whom.

1. Create the `${studnr}.prv` , `${studnr}.pub` and `${studnr}.crt` files with openssl (of 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.
5. If these standard methods work well, implement decode and then encode

To test

It is of course important that you have tested your program before submitting it. In the example output user? for one of the users.

These are the test cases which have to be implemented.

Test 0

In this test there is no encryption or signing or hashing. This test works without any changes `cryptomail_skel.py` .

In **bold font** the command, in normal font the expected output.

```
$ python3 cryptomail.py -f test0a.xmsg decode
Decoded:file:      test0a.xmsg
Decoded:receivers:
Decoded:senders:
Decoded:hash:      no-info
Decoded:secrets:   no-info
Decoded:mesg:      Dit is de message.\n
```

You can see the contents of the xmsg file by:

```
$ python3 cryptomail.py -f test0.xmsg info
{
    "mesg": "Dit is de message.\\n",
    "vers": "1.0"
}
```

Test 1

By not specifying senders or recipients, only the hashing methods are tested. Can this be done in the following way to test.

```
$ python3 cryptomail.py -f test1.xmsg encode
Encoded:file:      test1.xmsg
Encoded:receivers:
Encoded:senders:
Encoded:mesg:      Dit is de message.\n
```

```
$ python3 cryptomail.py -f test1.xmsg decode
Decoded:file:      test1.xmsg
Decoded:receivers:
Decoded:senders:
Decoded:hash:      success
Decoded:secrets:   success
Decoded:mesg:      Dit is de message.\n
```

Test 2

Here 1 receiver (`user1`) is tested

```
$ python3 cryptomail.py -f test2.xmsg -r user1 -m test.txt encode
Encoded:file:      test2.xmsg
Encoded:receivers: user1=success
Encoded:senders:
Encoded:mesg:      Dit is de message.\n

$ python3 cryptomail.py -f test2.xmsg -r user1 decode
Decoded:file:      test2.xmsg
Decoded:receivers: user1=success
Decoded:senders:
Decoded:hash:      success
Decoded:secrets:   success
Decoded:mesg:      Dit is de message.\n
```

The file `test2a.xmsg` is a test file to test `decode`, the secrets are not removed and there is no hash info.

Test 3

Here 1 sender (`user3`) Is tested

```
$ python3 cryptomail.py -f test3.xmsg -s user3 -m test.txt encode
Encoded:file:      test3.xmsg
Encoded:receivers:
Encoded:senders:   user3=success
Encoded:mesg:      Dit is de message.\n

$ python3 cryptomail.py -f test3.xmsg -s user3 decode
Decoded:file:      test3.xmsg
Decoded:receivers:
Decoded:senders:   user3=success
Decoded:hash:      success
Decoded:secrets:   success
Decoded:mesg:      Dit is de message.\n
```

The file `test3a.xmsg` is a test-file to test `decode`, the secrets are not removed and there is no hash info.

Test 4

This tests 1 receiver and 1 sender

```
$ python3 cryptomail.py -f test4.xmsg -m test.txt  -r user1 -s user3 encode
Encoded:file:      test4.xmsg
Encoded:receivers: user1=success
Encoded:senders:   user3=success
Encoded:mesg:      Dit is de message.\n

$ python3 cryptomail.py -f test4.xmsg decode
Decoded:file:      test4.xmsg
Decoded:receivers: user1=success
Decoded:senders:   user3=success
Decoded:hash:      success
Decoded:secrets:   success
Decoded:mesg:      Dit is de message.\n
```

Test 5

Here 2 receivers (user1,user2) are tested.

```
$ python3 cryptomail.py -f test5.xmsg -m test.txt -r user1,user2 encode
Encoded:file:      test5.xmsg
Encoded:receivers: user1=success,user2=success
Encoded:senders:
Encoded:mesg:      Dit is de message.\n

$ python3 cryptomail.py -f test5.xmsg decode
Decoded:file:      test5.xmsg
Decoded:receivers: user1=success,user2=success
Decoded:senders:
Decoded:hash:      success
Decoded:secrets:   success
Decoded:mesg:      Dit is de message.\n
```

Test 6

This tests 2 senders (user3 and user4)

```
$ python3 cryptomail.py -f test6.xmsg -m test6.txt -s user3,user4 encode
Encoded:file:      test6.xmsg
Encoded:receivers:
Encoded:senders:   user3=success,user4=success
Encoded:mesg:      Dit is de message.\n

$ python3 cryptomail.py -f test6.xmsg decode
Decoded:file:      test6.xmsg
Decoded:receivers:
Decoded:senders:   user3=success,user4=success
Decoded:hash:      success
Decoded:secrets:   success
Decoded:mesg:      Dit is de message.\n
```

Test 7

Tests 2 receivers (user1 and user2) and 2 senders (user3 and user4).)

```
$ python3 cryptomail.py -f test7.xmsg -m test.txt -r user1,user2 -s user3,user4 encode
Encoded:file:      test7.xmsg
Encoded:receivers: user1=success,user2=success
Encoded:senders:   user3=success,user4=success
Encoded:mesg:      Dit is de message.\n

$ python3 cryptomail.py -f test7.xmsg decode
Decoded:file:      test7.xmsg
Decoded:receivers: user1=success,user2=success
Decoded:senders:   user3=success,user4=success
Decoded:hash:      success
Decoded:secrets:   success
Decoded:mesg:      Dit is de message.\n
```

Check

Check whether you have written the correct code between # Student work }} and # Student work {{ .These functions must be implemented

- loadPrvKey
- loadPubKey
- genSesKey
- genSeslv
- encryptSesKey
- decryptSesKey
- encryptMesg
- decryptMesg
- signMesg
- verifyMesg
- calcHash
- chckHash
- encode
- decode

To hand in

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
```

You submit

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

So hand in four files!