جامعة التقنية
والعلوم التطبيقية
University of Technology
and Applied Sciences
Salalah صلالة

LAB Manual

CSSY2201 : Introduction to Cryptography

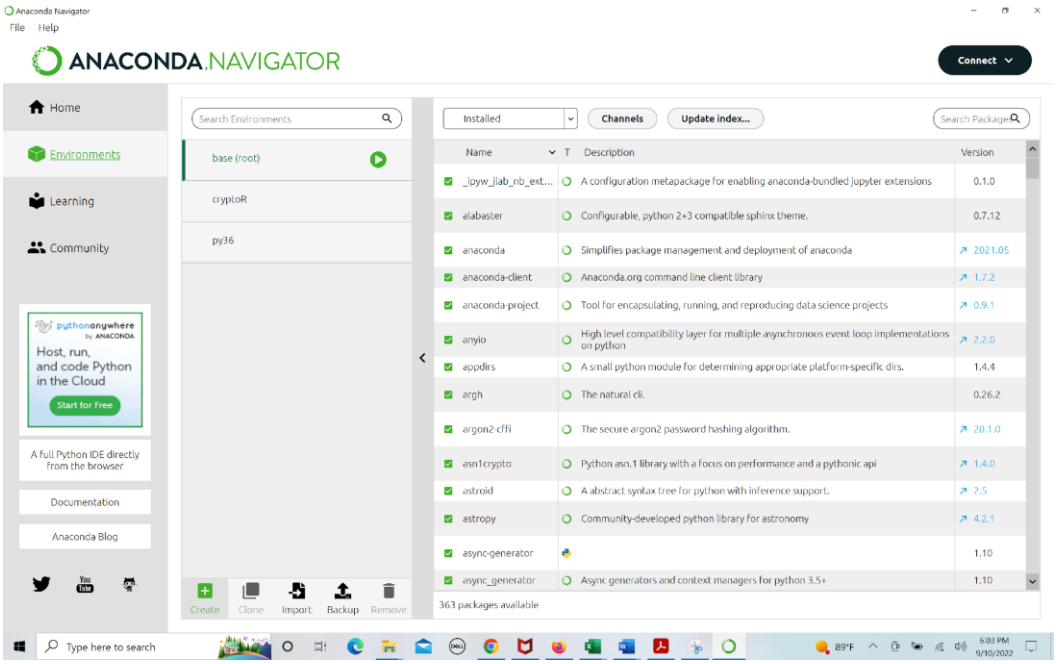
UTAS
Sultanate of Oman

September 2022

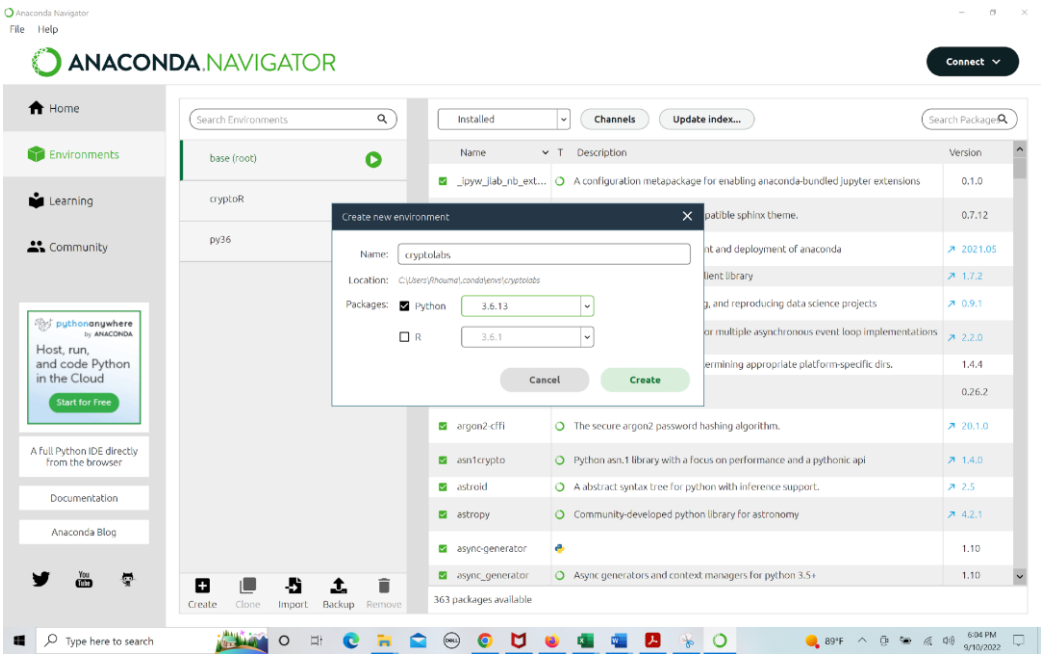
LAB1 : Introduction to Python

1) Install Anaconda with its latest version via : https://www.anaconda.com/products/distribution

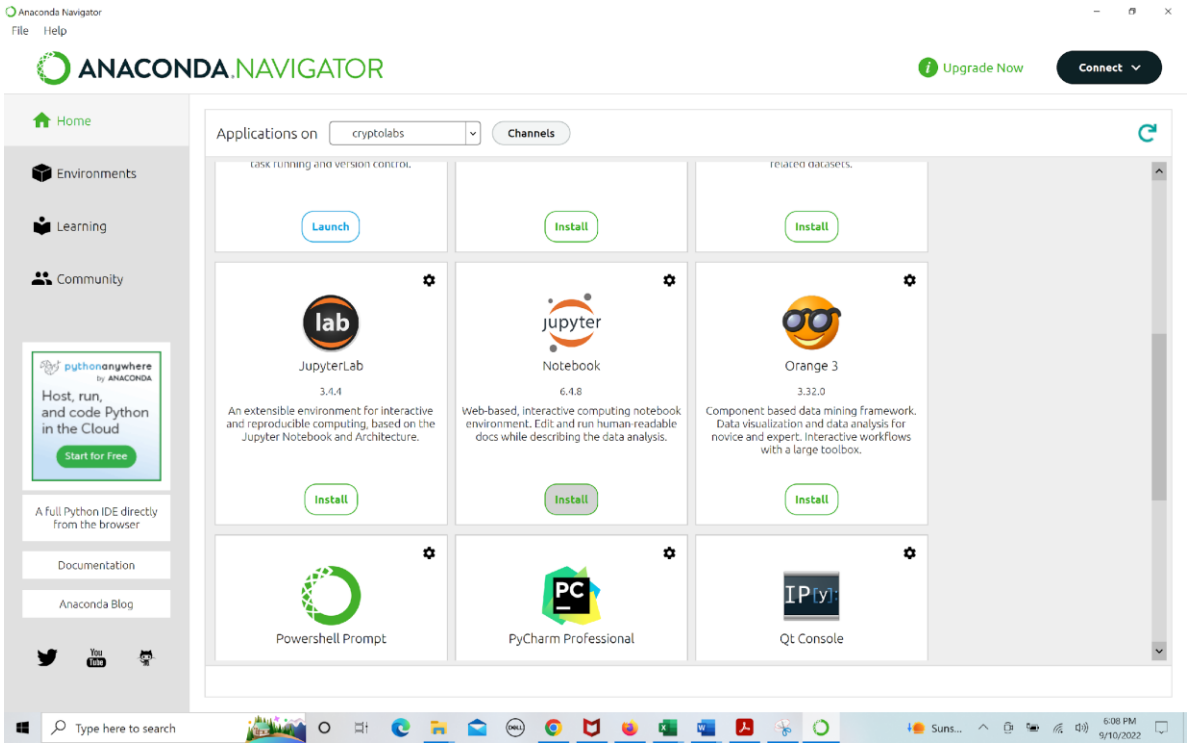
2) Open Anaconda Navigator



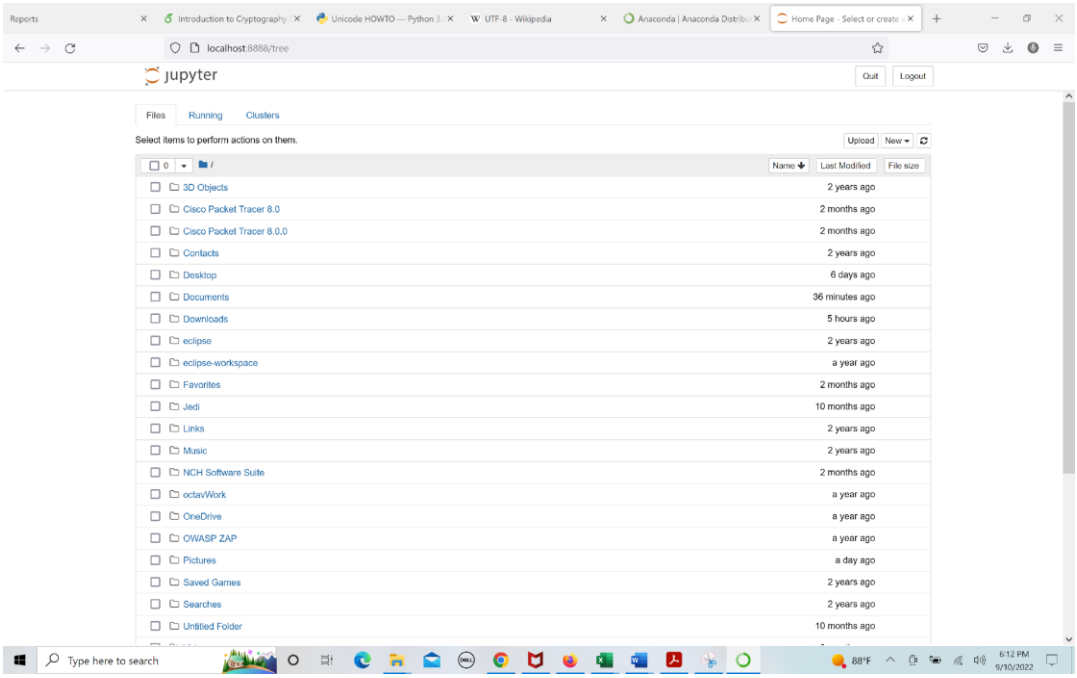
3) Click on Environments and create a new environment. Name it “cryptolabs”.



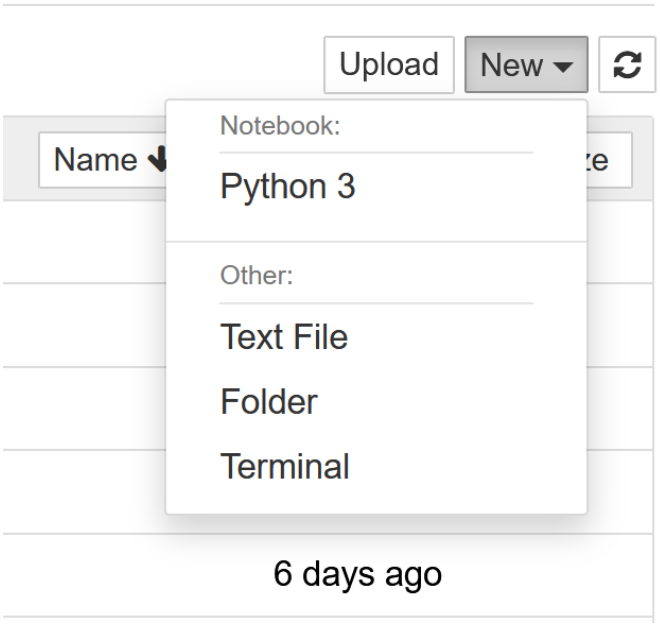
4) After creating the new environment, go back to home in the anaconda navigator, and scroll down the supported applications. Look for Jupyter Notebook and install it (or launch it if it is already installed)

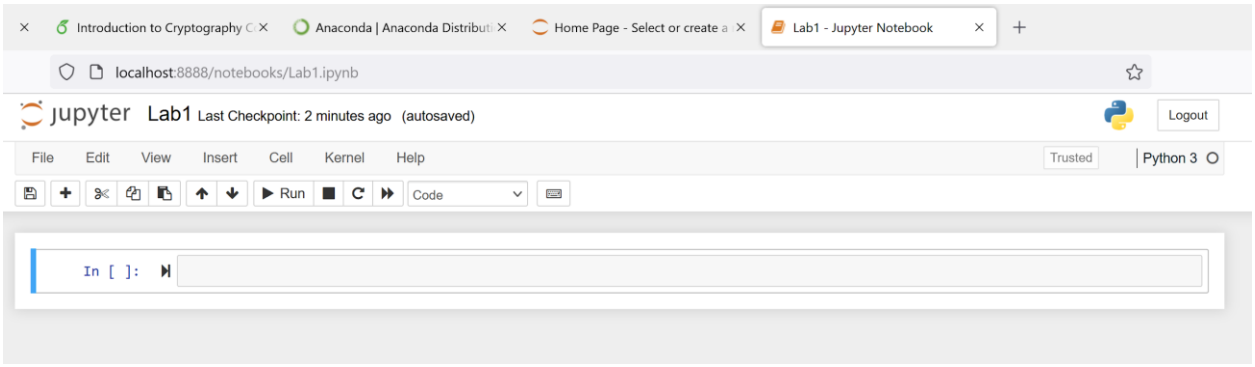


5) Launch Jupyter Notebook now:



6) On the top right of Jupyter notebook, you will find a button “New” by which youCreate a new notebook and name it Lab1:





shift + Enter run the current cell, select below

7) Start exploring python as a calculator by typing the following :

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # division always returns a floating point number
1.6
```

Division (/) always returns a float. To do floor division and get an integer result (discarding any fractional result) you can use the // operator; to calculate the remainder you can use %:

```
>>> 17 / 3  # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3  # floor division discards the fractional part
5
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # result * divisor + remainder
17
```

With Python, it is possible to use the ** operator to calculate powers

```
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

If a variable is not “defined” (assigned a value), trying to use it will give you an error:

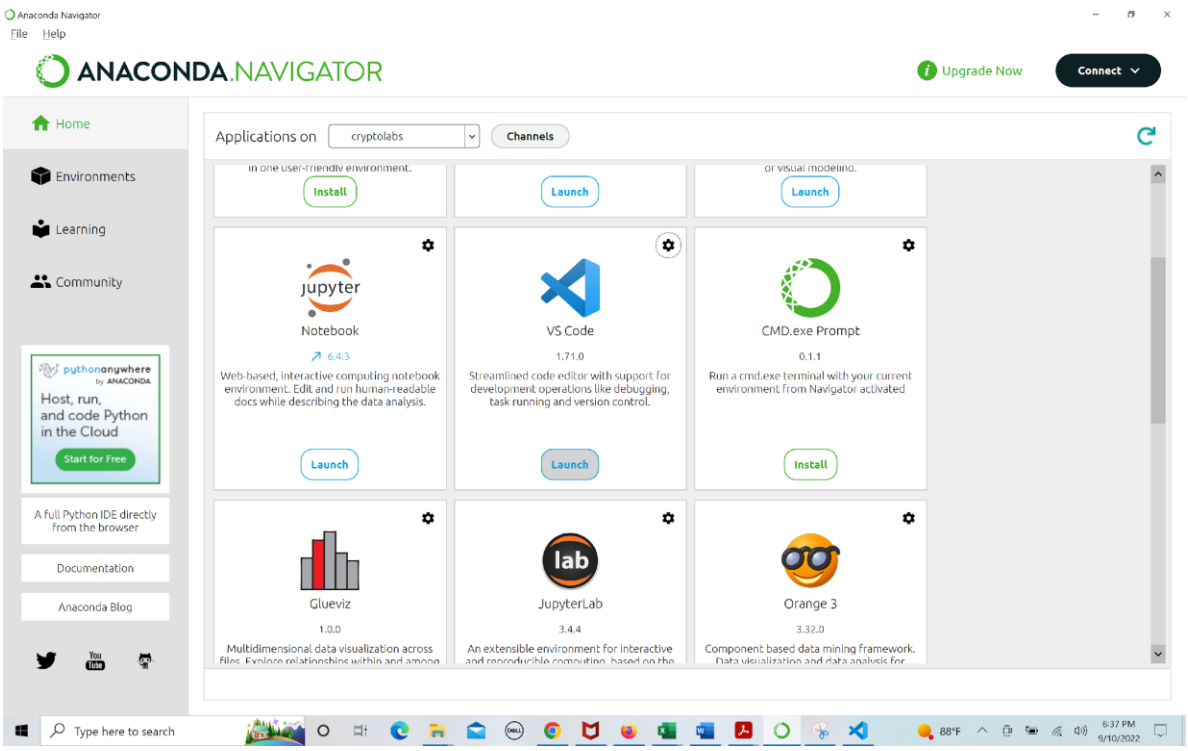
```
>>> n  # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

In interactive mode, the last printed expression is assigned to the variable _. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

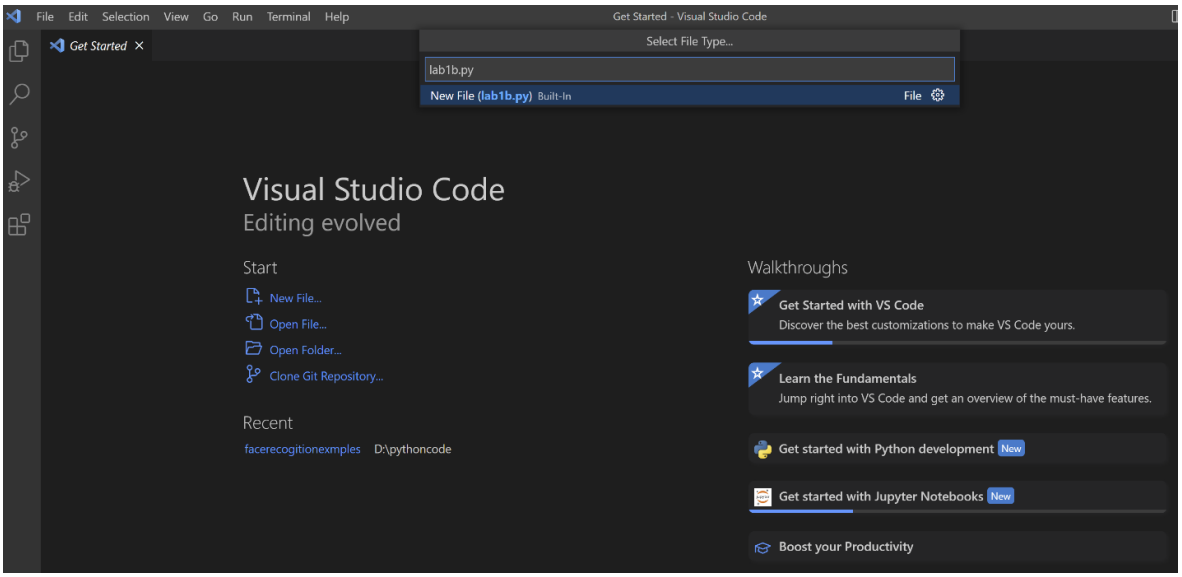
```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

Continue exploring python capabilities based on slides of lecture 1 pp 3-26.
Close Jupyter notebook.

8) You still see the anaconda navigator. And you should be in the same environment. Now browse the apps and install (or launch) VS code.



9) Now you will use an IDE to test Python capabilities (VS code is not a must, you can use Spyder or Pycharm or any suitable IDE) and create a new .py file named lab1b.py



Type the following :

```
for i in range(1,5):
    if i == 1:
        print ('I found one')
    elif i == 2:
        print ('I found two')
    elif i == 3:
        print ('I found three')
    else:
        print ('I found a number higher than three')

print (i)
```

you should get this output :

```
PS C:\Users\Rhouma> & C:/Users/Rhouma/.conda/envs/cryptolabs/python.exe d:/pythoncode/lab1b.py
I found one
I found two
I found three
I found a number higher than three
4
```

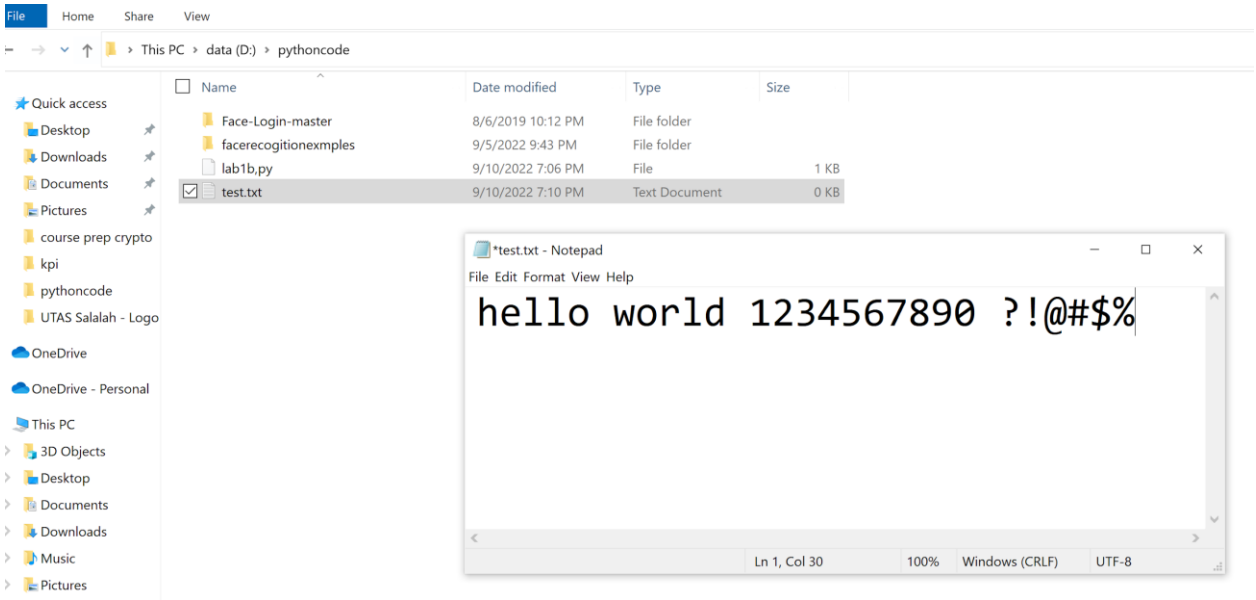
10) Create a function and test its call as follows and call it as follows :

```
def myEnc(plaintext, key):
    return "ciphertext"

s=myEnc('hello','secret key')
```

```
print(s)
```

you should get this output : ciphertext

11) Create a txt file in the same folder where your file “lab1b.py” resides and name it test.txt. write down anything from letters to numbers to special characters as follows and save it:



12) Type the following code to read and display the content of test.txt

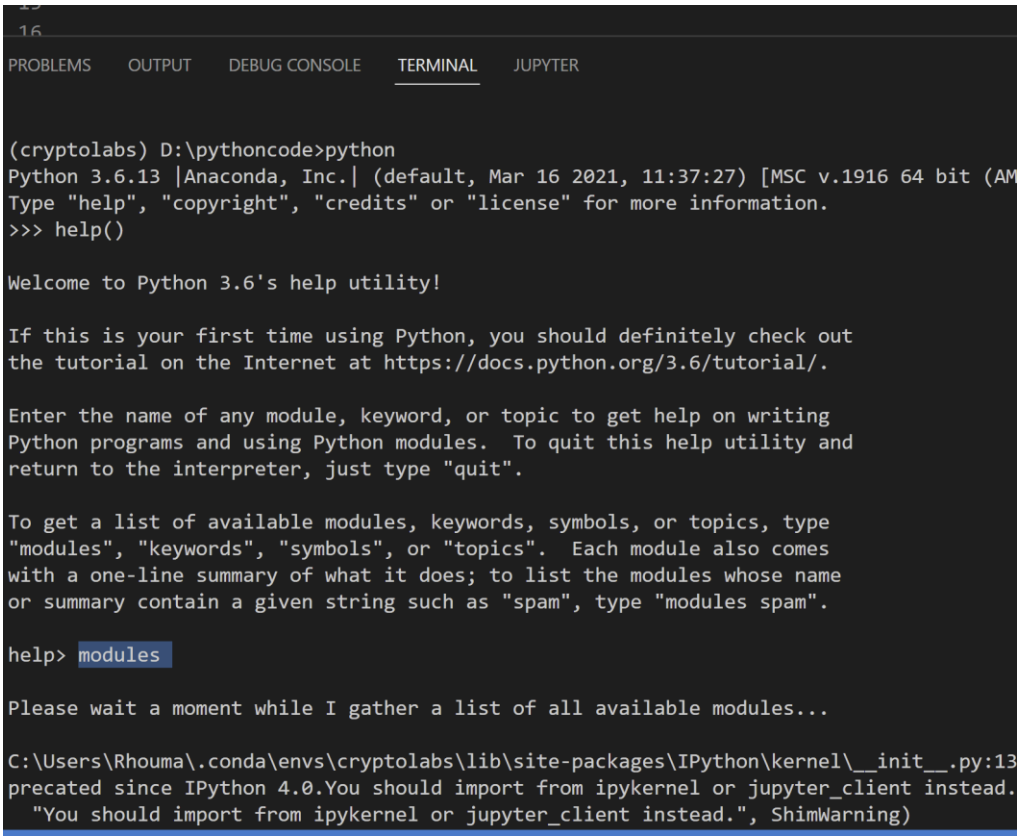
```
f = open("test.txt", "r")
print(f.read())
```

13) Type the following to create a new file test2.txt

```
f = open("test2.txt", "a")
f.write("Now the file has more content!")
f.close()
```

14) Now we will install all the necessary modules for the next labs. Do not close VS code (or the IDE you are using) and go back to Anaconda navigator. In the environments section be sure that “cryptolab” is the active environment and type hashlib to check

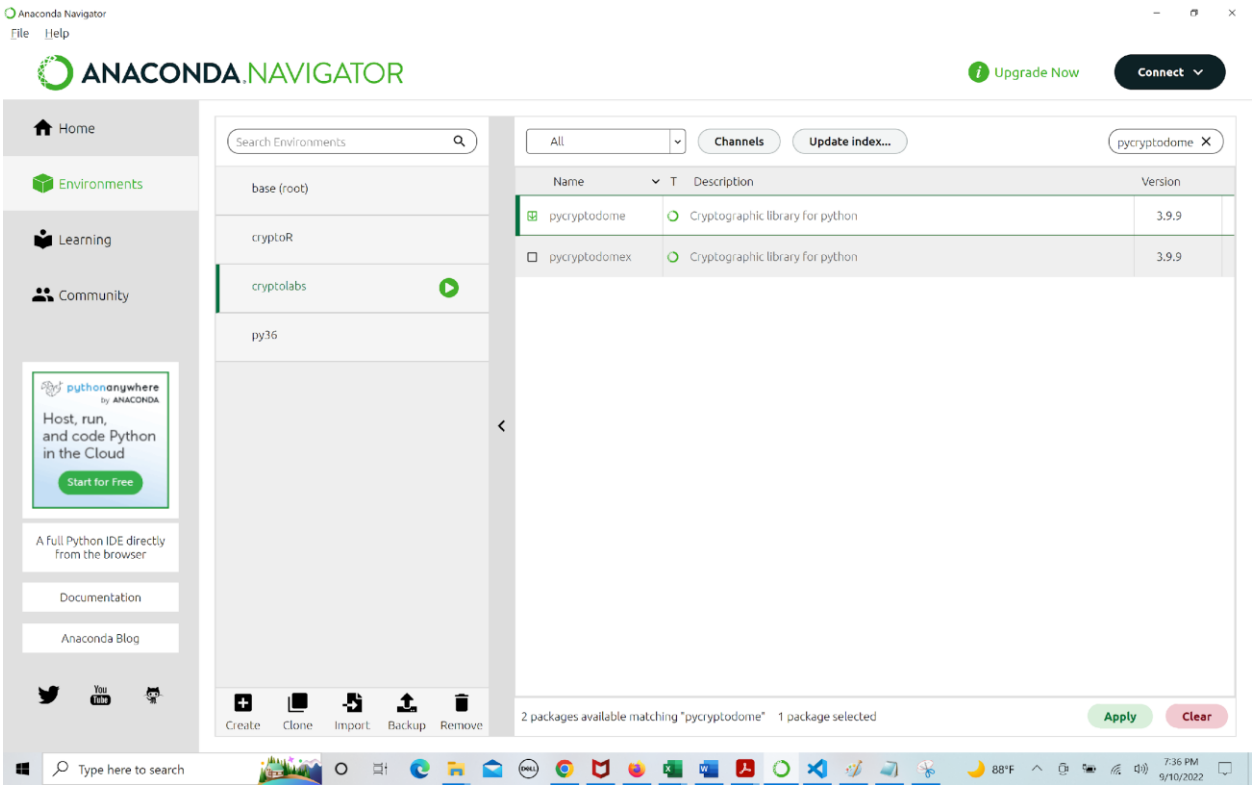
if it is installed. You can also search for it in the terminal by typing python then help(), then modules



```
_signal              gc                  pythoncom           win32gui
_sitebuiltins        genericpath         pywin               win32gui_struct
_socket              getopt              pywin32_bootstrap   win32help
_sqlite3             getpass             pywin32_testutil    win32inet
_sre                 gettext             pywintypes          win32inetcon
_ssl                 glob                queue               win32job
_stat                gzip                quopri              win32lz
_string              hashlib             random              win32net
_strptime            heapq               rasutil             win32netcon
_struct              hmac                re                  win32pdh
_symtable            html                regcheck            win32pdhquery
_testbuffer          http                regutil             win32pdhutil
_testcapi            idlelib             reprlib             win32pipe
_testconsole         imaplib             rlcompleter         win32print
_testimportmultiple  imghdr              rmagic              win32process
_testmultiphase      imp                 runpy               win32profile
_thread              importlib           sched               win32ras
_threading_local     inspect             secrets             win32rcparser
_tkinter             io                  select              win32security
_tracemalloc         ipaddress           selectors           win32service
_warnings            ipykernel           send2trash          win32serviceutil
_weakref             ipykernel_launcher  servicemanager      win32timezone
_weakrefset          ipython_genutils    setuptools          win32trace
_win32sysloader      isapi               shelve              win32traceutil
_winapi              itertools           shlex               win32transaction
_winxptheme          jedi                shutil              win32ts
abc                  jinja2              signal              win32ui
```

In my case it is already installed. If not you can install from the navigator.

15) Install pycryptodome by searching on it in the navigator. Click apply to be installed



Test if hashlib is correctly installed :

```
import hashlib
plaintext_password = b'password'
hashed = hashlib.md5(plaintext_password).digest()
print("binary digest =", hashed)
```

you should get the following output :

binary digest = b"_M\xcc;Z\xa7e\xd6\x1d\x83'\xde\xb8\x82\xcf\x99"

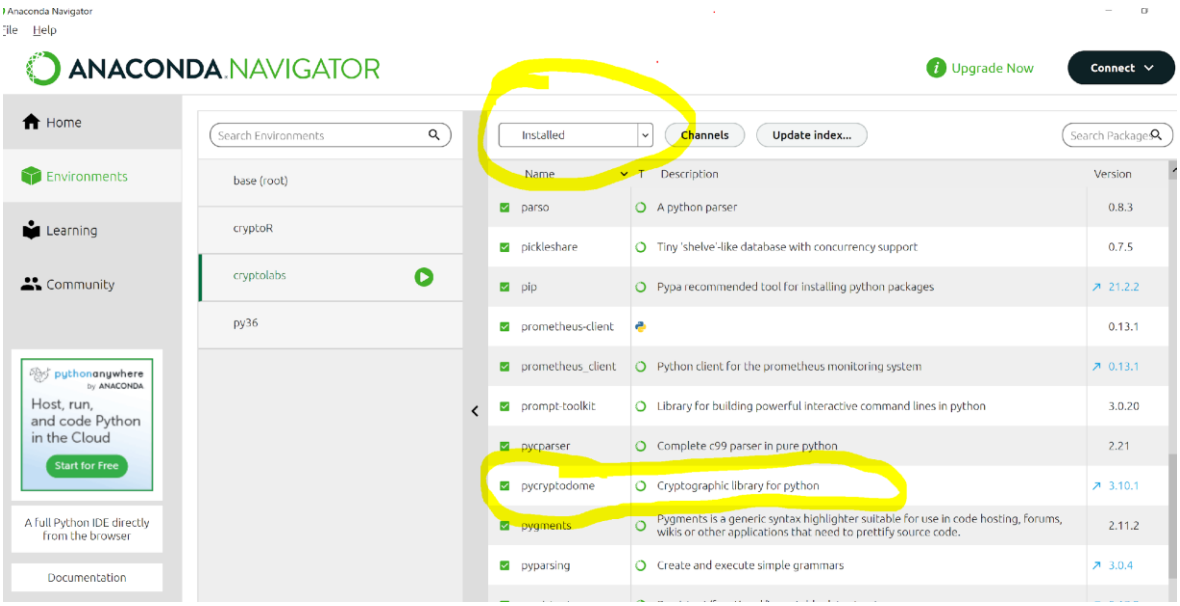
16) You can see that “pycrytodome” is installed in the “cryptolabs” environment. Type the following to test if pycryptodome is correctly installed.

```
from Crypto import Random
from Crypto.PublicKey import RSA

random_generator = Random.new().read
key = RSA.generate(1024, random_generator)
print(key.publickey)
```

you should nearly have the same output format (not exactly the same values because we are generating random numbers and keys) :

<bound method RsaKey.public_key of
RsaKey(n=135336268872937239910919581975340040971549577108918873219963
391887902966813726020856344451659345640399190093259376152158973798207
469894809784837182900886072918202383502887046780411405046373291403192
992903931476255707659134216169445752875485248711985872873058396173366
666686925931136479407503389816527499776508921, e=65537,
d=453481920815676913566961261604661076908498819493089164275202723020331896673546762592935922980794095421855508792880403651534951040792054873258621013550125524396699749093282320125424476433359176883516994893947834590442586116903501519331758805049553503103699890726882704392942381179261504717399878371708226228 9,
p=108928531618347411249259939173327289659763805502989615403967533463122992425974705401053636357377867313487231101496226508892020099556775406761227960349838 87,
q=124243177487340546494770953387518362180706418877519062161596508377543837093590900253739404745620101046378042096145242009761997460400104382304372782314049 83,

u=104125763557798891271838056971215087246976028555729376258389034950
31428523749391695536589049327383548224671205499911138897708579395296
254021284281143792194)>



17) follow the same strategy to install “cryptography”, “bitvector” and “json”

LAB2 : Introduction to Cryptology

1) Complete the following code to create a reverse encryption function, make the encryption and the decryption of a short text :

```
def reverseCipher(plaintext):
    ciphertext = ''
    i = len(plaintext) - 1
    while
        ciphertext =
        i = i - 1
    return ciphertext
#### encryption example (use the reverseCipher)
plaintext = "If you want to keep a secret, you must hide it."
ciphertext =
print( ……..)
#### Decryption test example (use the reverseCipher)
recovered=
print(……)
```

2) You are going to simulate a simple authentication system based on username and a password. The system will save the hash of passwords entered by users who are going to create their credentials by typing their passwords twice. The system will compare between the digests of passwords and will output a successful message is the typed password match. For that we need to use the module “bcrypt” and “getpass”. Remember to use a salt before hashing the password of every user. Informations about each user, his hashed password and the used salt should be saved in a text file.

You are going to create three functions :

- ask_for_username() : to ask the user for his username
- ask_for_password() : to ask the user for a password (using getpass) and generating the salt and hashing the password + salt. This function will ask the user twice for the password, it will then compare between the hashes and will return the hash and its salt if there is a match. If there is no match it will keep asking the user for the correct password.
- store_info(username, hashed_pass, salt) : to write the username + hash + salt in a file.

The main file will of course call these functions in order to simulate the authentication system. We give the general structure of the code :

```
import getpass

import bcrypt

def ask_for_username():

    print("enter the user name would you like to use...")

    username = ……

    return ……
```

```
def ask_for_password():
    v
    inputstr=str.encode(p)

    hashed_password1 = ……..

    print("Please enter the password again...")

            print("Your username, hashed password and salt is stored in testhash.txt file")

            return hashed_password2, salt

     else:

    print("Your password do not match. Please retry...")
```

```
def store_info(username, hashed_pass, salt):

   with open("testhash.txt", "a" ) as password_file:

    password_file.write(username  + " | " + "".join(map(chr,hashed_pass ))   + " | " + "".join(map(chr, salt )) + "\n" + "\n")
```

and

```
username = ……..

hashedPass, salt = ……..

store_info(……., ……., ……)
```

the

main program :

LAB3 : Classical Cryptography

1) Cesar Cipher

Use a plaintext without punctuation or spaces to implement Cesar.
We recall that cesar make a shift of three letters to the right to envrypt every letter in the alphabet. given that x is the current letter of the alphabet, the Caesar cipher function adds three for encryption and subtracts three for decryption.
While this could be a variable shift, let's start with the original shift of 3:

$$Enc(x) = (x + 3) \% 26$$
$$Dec(x) = (x - 3) \% 26$$

The encryption formula adds 3 to the numeric value of the number. If the value exceeds 26, which is the final position of Z, then the modular arithmetic wraps the value back to the beginning of the alphabet. The use of the key simplifies the alphabet indexing.

We give the general structure of the code.

```
key = 'abcdefghijklmnopqrstuvwxyz'
def enc_caesar(n, plaintext):
  result = ''
  for j in plaintext.lower():
    i = ……..
    result += ……..
      return result.lower()
plaintext = 'hello'
ciphertext = enc_caesar(3, plaintext)
print (ciphertext)
```

2) Now write the code for the cesar decryption function and test it using the previous generated ciphertext in question 1.

3) Simulate the brute force attack on cesar cipher by testing all the shifting keys from 0 to 25 in order to decrypt a ciphertext assuming that you don't know the right key. Test your code on a previously generated ciphertext.

4) Enhance the previous three codes by assuming that a text can contain punctuation points and symbols other than letters. (it suffices to not encrypting/decrypting them and letting them as they are).

LAB4 : DES Encryption in Python

NB: this lab needs to install the module “pycryptodome” and “base64”. You need to install them via pip or anaconda navigator or conda command.
In Replit you will find them already installed.

Part A:
- Manipulate DES encryption and decryption in one mode of block-encryption (CFB) on a simple text

Step 1 : Import needed packages. Base64 is used to visualize ciphertext in readable character to the user. DES from Crypto.Cipher contains the needed functions for encryption and decryption with DES in many modes of block encryption. Random from Crypto is used to generate pseudo random number for iv (initialization vector) used in CFB, CBC, OFB or others.

```
import base64
from Crypto.Cipher import DES
from Crypto import Random
```

Step 2 : iv is the initialization vector used to initialize the encryption in CFB. It should be random and of size 8 bytes.

```
iv = Random.get_random_bytes(8)
```

Step 3 : Initialize the encryptor “des1” and the decryptor “des2” having the same key (of course) and the same iv. See it as transmitter and receiver who should share a secret key ‘01234567’ of length 8 bytes and an iv (which is not secret) of length 8 bytes too. Both they use the same mode of block encryption which is in this case CFB.

```
des1 = DES.new('01234567', DES.MODE_CFB, iv)
des2 = DES.new('01234567', DES.MODE_CFB, iv)
```

Step 4 : give a plaintext to be encrypted by DES.

```
text = 'abcdefghijklmnop'
```

Step 5 : Make the encryption of the plaintext by the encryptor. Print the ciphertext to the user in both forms, bytes and base64.

```
cipher_text = des1.encrypt(text)
print(cipher_text)
cipher_textt=base64.b64encode(cipher_text).decode()
print(cipher_textt)
```

Step 6 : Decrypt the ciphertext and recover the plaintext using the decryptor. Visualise it in both forms “bytes” and “UTF-8”. (note that from the origin, the plaintext is in its UTF-8 form, so we need to conver the recovered plaintext after decryption in its original UTF-8 form.

```
decipher_text=des2.decrypt(cipher_text)
print(decipher_text)
decipher_text=decipher_text.decode("utf-8")
print(decipher_text)
```

Step 7 :
- Explore other modes of block encryption like CBC, or OFB
- use other keys
- use other plaintext
- always verify the results of your work.

Part B :

- Manipulate 3DES encryption and decryption in one mode of block-encryption (CBC) on a text file of plaintext. Encrypt it and save it in a ciphertext file. Finally open the ciphertext file and recover the plaintext by decrypting, save the result in a recovered plaintext file. It simulates transmitter and receiver secure communication.

Step 1 :

```
from Crypto.Cipher import DES3
import base64
```

Step 2 : encryption function
- Declare a function for handling and encrypting the plaintext file named “encypt_file” having 4 arguments: the key, the plaintext file, the output file (optional), and the chunksize=16*1024.

The chunksize represent the minimal block size loaded in the buffer when the encryptor read the plaintext file.

If the user does not give an output filename, then a file will be created and take as name (plaintext-filename+enc).

```
12
13 def encrypt_file(key, in_filename, out_filename=None, chunksize=16*1024):
14     """ Encrypts a file using DES3 (CBC mode) with the
15         given key.
16
17         key:
18             The encryption key - a string that must be
19             either 16, or 24 bytes long. Longer keys
20             are more secure.
21
22         in_filename:
23             Name of the input file
24
25         out_filename:
26             If None, '<in_filename>.enc' will be used.
27
28         chunksize:
29             Sets the size of the chunk which the function
30             uses to read and encrypt the file. Larger chunk
31             sizes can be faster for some files and machines.
32             chunksize must be divisible by 16.
33     """
34
35     if not out_filename:
36         out_filename = in_filename + '.enc'
37
```

- generate an initialization vector (iv) of length 8 bytes :

```
38     iv=os.urandom(8)
39     print(iv)
```

- open two loops to handle the input and output files. The input is opened to be read, the output file is opened(created) to write in it.
- Line 52: first thing to write in the output file is the iv. The iv (which was been generated by the transmitter) is not a secret, it should be communicated to the receiver. The best thing to do this is to put the iv in the overhead of the plaintext (in the beginning of the ciphertext file). By doing so, the receiver can extract it first and use it to feed the CBC decryption machine.

```
44
45     with open(in_filename, 'rb') as infile:
46         with open(out_filename, 'wb') as outfile:
47
48             # we have to write the iv in the begining of the file,
49             #so the receiver can extract it easily. the iv is not a secret !
50             outfile.write(iv)
51
```

- Open an infinite loop to be used to read the input file “chunk” by “chunk” until the end of the file (if we reach the end we break the loop). Another case come in when the chunk is not a multiple of 16, so if the remaining data in the input file is less than “a multiple of 16”, we padded by spaces ‘ ‘ in bytes format.

```
53
54              while True:
55                  chunk = infile.read(chunksize)
56                  if len(chunk) == 0:
57                      break
58                  elif len(chunk) % 16 != 0:
59                      chunk += b' ' * (16 - len(chunk) % 16)
60
```

- Still in the previous loop, we begin to encrypt the chunk and write the cipher-chunk in the output file. We print also for the user the cipher-chunk in its base64 format. That is the end of the loop and the function **encrypt_file**.

```
61                  outfile.write(encryptor.encrypt(chunk))
62                  chunkd=encryptor.encrypt(chunk)
63                  chunkd=base64.b64encode(chunkd).decode()
64                  print(chunkd)
65
```

Step 3 : declare the decryption function as follows :
- it should have 4 arguments: the secret key, the input file name (ciphertext file), the output filename (the recovered plaintext file), and the chunksize which should be the same as the transmitter operated.
- open the ciphertext file to be read
- read first the iv of length 8 bytes
- construct the 3DES decryptor having as input: the key, the CBC mode as block decryption and the iv.
- create the output file which will receive the recovered plaintext chunck by chunk
- open an infinite loop to scan all the chunks
- read the inline chunk having as size the predefined chunksize
- the infinite loop will break when we reach the end of file (length of chunk = 0)
- decrypt the extracted chunk and write it to the output file, and also print it to the user.

```
def decrypt_file(key, in_filename, out_filename, chunksize=16*1024):
    """ Decrypts a file using DES3 (CBC mode) with the
        given key. Parameters are similar to encrypt_file,
        with one difference: out_filename, if not supplied
        will be in_filename without its last extension
        (i.e. if in_filename is 'aaa.zip.enc' then
        out_filename will be 'aaa.zip')
    """

    with open(in_filename, 'rb') as infile:

        iv = infile.read(8) #iv must be 8 bytes long. it is read from the file
        #because the trasmitter has written the iv in the front of the file. the iv is not a secret
        decryptor = DES3.new(key, DES3.MODE_CBC, iv)

        with open(out_filename, 'wb') as outfile:
            while True:
                chunk = infile.read(chunksize)
                if len(chunk) == 0:
                    break

                outfile.write(decryptor.decrypt(chunk))
                print(decryptor.decrypt(chunk))
```

Step 4 : Now the main program can use those functions to get the job done.
- set a key of length 16 or 24 bytes long. You can set it static or generate it using some random number generator.
- call the encryption function with arguments: the secret key, a txt file (in my case a file named test.txt) and the chnuksize.
The result of the encyption function is the creation of a file plaintext_file+enc which in my case test.txt.enc
- call the decryption function with arguments : the secret key, the ciphertext file (test.txt.enc), the recovered file (testdec.txt), and the chunksize).

```

#choose a key of 16 or 24 bytes long
key= b'0123456789abcdef'
print(key)



encrypt_file(key, 'test.txt', chunksize=16*1024)
print("Done")
#
#
decrypt_file(key, 'test.txt.enc', 'testdec.txt', chunksize=16*1024)
print("Done")
```

For your reference this is the content of test.txt

test.txt ×

```
1  hello world. this is a test for encrypting a text from a
   txt file 1234567890!?/@#$%
```

This is the content of the ciphertext file test.txt.enc :

test.txt.enc ×

```
1  #M FF ?!?•'b?N^?I:?DC4? FF B RS ?DC3w?}BEL<?NUL?f?ETB?_4?;$?SYN
   ,?DC4?]VT?[ETX?!(7?60?vkCANxG4}?\?Ft??N?HA?CB&?YU GS RS Y BEL
   ¬?  ?!?
```

of course they are not readable symbols. Remember that we have written the ciphertext as bytes.
If we choose to write it encoded as base64, this result will be like this:

KpA0exo34f3mFTBUL5w1111acYAa5EXctHd+PYqXzTYIL50eqdN3AVRg4TS3okVYyGhGREmds
b2c7UEmQ6dsBtu/3IPdDUb4kjVdbFjnW467W80NIDOirOiGsqgXsEZd

And finally this is the recovered file : testdec.txt :

testdec.txt ×

```
1  hello world. this is a test for encrypting a text from a
   txt file 1234567890!?/@#$%
```

Results of course will differ based on your plaintext, iv and secret key.

LAB5 : AES Encryption in Python

PART A : AES using pycryptodome

- Manipulate AES encryption and decryption in one mode of block-encryption (CBC) on a text file of plaintext. Encrypt it and save it in a ciphertext file. Finally open the ciphertext file and recover the plaintext by decrypting, save the result in a recovered plaintext file. It simulates transmitter and receiver secure communication.

Step 1 : import necessary packages

```
 9 import os
10 from Crypto.Cipher import AES
11 import hashlib
12 import base64
13
```

Step 2 : encryption function
- Declare a function for handling and encrypting the plaintext file named “encypt_file” having 4 arguments: the key, the plaintext file, the output file (optional), and the chunksize=16*1024.
The chunksize represent the minimal block size loaded in the buffer when the encryptor read the plaintext file.
If the user dose not give an output filename, then a file will be created and take as name (plaintext-filename+enc).

- generate an initialization vector (iv) of length 16 bytes
- declare the AES encryptor which has 3 arguments: the key, the block mode CBC, and the iv.

- open two loops to handle the input and output files. The input is opened to be read, the output file is opened(created) to write in it.
-Line 6 : First thing to write in the output file is the iv. The iv (which was been generated by the transmitter) is not a secret, it should be communicated to the receiver. The best thing to do this is to put the iv in the overhead of the plaintext (in the beginning of the ciphertext file). By doing so, the receiver can extract it first and use it to feed the CBC decryption machine.

- Open an infinite loop to be used to read the input file “chunk” by “chunk” until the end of the file (if we reach the end we break the loop). Another case come in when the chunk is not a multiple of 16, so if the remaining data in the input file is less than “a multiple of 16”, we padded by spaces ‘ ‘ in bytes format.

- Still in the previous loop, we begin to encrypt the chunk and write the cipher-chunk in the output file. We print also for the user the cipher-chunk in its base64 format. That is the end of the loop and the function **encrypt_file**.

```
def encrypt_file(key, in_filename, out_filename=None, chunksize=16*1024):

    if not out_filename:
        out_filename = in_filename + '.enc'

    iv=os.urandom(16)
    encryptor = AES.new(key, AES.MODE_CBC, iv)
    #filesize = os.path.getsize(in_filename)

    with open(in_filename, 'rb') as infile:
        with open(out_filename, 'wb') as outfile:
            #outfile.write(struct.pack('<Q', filesize))
            outfile.write(iv)

            while True:
                chunk = infile.read(chunksize)
                if len(chunk) == 0:
                    break
                elif len(chunk) % 16 != 0:
                    chunk += b' ' * (16 - len(chunk) % 16)

                outfile.write(encryptor.encrypt(chunk))
                chunkd=encryptor.encrypt(chunk)
                chunkd=base64.b64encode(chunkd).decode()
                print(chunkd)
```

Step 3 : declare the decryption function as follows :
- it should have 4 arguments: the secret key, the input file name (ciphertext file), the output filename (the recovered plaintext file), and the chunksize which should be the same as the transmitter operated.
- open the ciphertext file to be read
- read first the iv of length 16 bytes
- construct the AES decryptor having as input: the key, the CBC mode as block decryption and the iv.
- create the output file which will receive the recovered plaintext chunck by chunk
- open an infinite loop to scan all the chunks
- read the inline chunk having as size the predefined chunksize
- the infinite loop will break when we reach the end of file (length of chunk = 0)

- decrypt the extracted chunk and write it to the output file, and also print it to the user.

```
40 def decrypt_file(key, in_filename, out_filename=None, chunksize=16*1024):
41
42     with open(in_filename, 'rb') as infile:
43         iv = infile.read(16)
44         decryptor = AES.new(key, AES.MODE_CBC, iv)
45
46         with open(out_filename, 'wb') as outfile:
47             while True:
48                 chunk = infile.read(chunksize)
49                 if len(chunk) == 0:
50                     break
51
52                 outfile.write(decryptor.decrypt(chunk))
53                 print(decryptor.decrypt(chunk))
```

Step 4 : Now the main program can use those functions to get the job done.

- set a key of length 256 bits (32 bytes). You can use some random number generator to produce it. For example the sha256 hash function. It is always a good idea to take the key from a digest of a hash because it will be naturally generated randomly.

- call the encryption function with arguments: the secret key, a txt file (in my case a file named test.txt) and the chunksize.

The result of the encyption function is the creation of a file plaintext_file+enc which in my case test.txt.enc

- call the decryption function with arguments : the secret key, the ciphertext file (test.txt.enc), the recovered file (testd.txt), and the chunksize).

```
57 password = b'kitty'
58 key = hashlib.sha256(password).digest()
59 print(key)
60
61 encrypt_file(key, 'test.txt', chunksize=16*1024)
62 print("Done")
63
64
65 decrypt_file(key, 'test.txt.enc', 'testdd.txt', chunksize=16*1024)
66
```

PART B: AES using "cryptography.fernet" module

Cryptograhy.fernet implement AES as the symmetric algorithm by default. It is very simple to use. Test this code to encrypt a simple text.

lab5B.py ×

```
1  from cryptography.fernet import Fernet
2   # Put this somewhere safe!
3  key = Fernet.generate_key()
4  f = Fernet(key)
5  encrypted = f.encrypt(b"A really secret message. Not
   for prying eyes.")
6
7  print(encrypted)
8  print()
9  decrypted=f.decrypt(encrypted)
10 print(decrypted)
11
```

LAB6 : RSA

PartA:

- Manipulate RSA encryption and decryption and handle text files of plaintext and/or ciphertext.

Step 1 : import necessary packages

```
8
9  from Crypto.PublicKey import RSA
10 from Crypto import Random
11 from Crypto.Cipher import PKCS1_OAEP
12 import base64
13
```

Step 2 : Generate the pair of RSA keys

```
14
15 random_generator = Random.new().read
16 key = RSA.generate(1024, random_generator)
17
```

Step 3 : Transmitter : Extract the public key from the key ring and use it to encrypt a simple text. Print the ciphertext to the user. And save the ciphertext in a txt file :

```
18
19 publickey = key.publickey()
20 encryptor = PKCS1_OAEP.new(publickey)
21 encrypted = encryptor.encrypt(b'Hi SK encrypt this message')
22 encrptedbase64=base64.b64encode(encrypted)
23 print('encrypted message:', encrypted)
24 f = open('encryptionn.txt', 'w')
25 f.write(str(encrypted))
26 print(encrptedbase64)
27 f.close()
28
```

Step 4 : Receiver :

- Open the txt file (the one that was used to store the ciphertext).
- Extract the ciphertext
- Construct the RSA decryptor having as key the “private key”
- Decrypt the ciphertext
- print it to the user
- write the recovered plaintext in the same txt file ‘encryptionn.txt’ without deleting the ciphertext.

```
30 f = open('encryptionn.txt', 'r')
31 message = f.read()
32 decryptor = PKCS1_OAEP.new(key)
33 decrypted = decryptor.decrypt(encrypted)
34 print('decrypted', decrypted)
35 f = open('encryptionn.txt', 'w')
36 f.write(str(message))
37 f.write(str(decrypted))
38 f.close()
```

PartB: a simple RSA key generation and encryption/decryption example

Apply step by step RSA algorithm in generating keys and encrypting/decrypting text

1) Import the following modules :

```
1 from Crypto.Util.number import *
2 from Crypto import Random
3 import Crypto
4 import gmpy2
5 import sys
```

2) Use the module pycryptodome to generate random prime numbers p and q as follows (for p) :

```
p = Crypto.Util.number.getPrime(bits, randfunc=Crypto.Random.get_random_bytes)
```

Where bits is the length of the generated number in bits.

3) Compute n and PHI
4) Choose e where gcd(e,PHI) = 1. (recommended e = 65537)
5) Find d where d is the multiplicative inverse of e mod PHI. (use the function gmpy2.invert)
6) Encode the text to utf-8 format, then convert those bytes to long (this is to ensure RSA manipulate numbers)
7) Use the function pow to encrypt the message m with e and n
8) Use the function pow to decrypt the ciphertext with d and n
9) Print all the result.

This code will work for messages (long numbers) inferior to n of course.
Test with longer messages and say what you did find.

PartC: using openssl

The openssl tool provides specific commands for key generation of some public key encryption schemes. The command genpkey is used to generate a public / secret keypair. The way this command is used depends on the selected public key algorithm.

RSA Keys using openssl

$ openssl genpkey -algorithm rsa

outputs a text encoding of a public key and a secret key for the RSA encryption scheme, with the default parameters (e.g., the size of the modulus). For example, the previous call results in the output

-----BEGIN PRIVATE KEY-----
MIICdwIBADANBgkqhkiG9w0BAQEFAASCAmEwggJdAgEAAoGBAKT7B3Vi+hdOPXr
5 lIjR3Ao5U4JjbdmQ4hhX5hh/uJwEEA7GkKrDTGOqLTcZZgeXH4nrq5SwoG6O4Mi1
hT6s/FSdpRkl6LFNcjpXPT4GtzPycMKhsGu+rnLiJmuXtEMM1Cw3gutYVa63ygJx
f7YDcJxI2FcAd02jsKA11MP6n4CTAgMBAAECgYBfsU8xMli3PeWBN9SEy5zivT+H
6J4lzNinoAxRd3uf2udpepkcwyzCvkl9pRi+HFTpza13ED/uAKe3IzqHERVGHFMd
qduffQg5DNDymkqnCcJ5yZLe9qn3tj3/EW3Om3Z2Pvn6R9NLFbN8YnTwjWowFmF
e mR0dYticID6Ilrv54QJBANqGNGQ2qcDTAeaYeWLY5wnvsm/Apgk3u2qogJB8dp+z
rfey7OAq4B5n+r+0FMZPyqHcs4olUvrCjiEEb/eiL7ECQQDBRh2YpzmW2NBNt+Qy
UMrGImRli7GFlp9UKhkRwiOhR1p48mD7yjbSqb7eG0QEVLLN5F02AeALlZDFvTd4
CemDAkEApBc6qDXT6pOITdwY6nztoKx5VSlYhHtxJHo7cEPF385QyDt3XC1V9f8m
b2WOZAvuoPTVbNryIJKPn4NxgIYtQQJAHembJQgkmpsdyhl+4OauK3IhNbIkHHfO
WvlU/fGdEBX6A6QHrJCEYaBR4RA5O65cKg+A+Z3arhBM4r3BOvvVvwJBAJ2A3l0p
TCNwuGbuUVw8KjJ6zHoLro3pTruhgDZvfjvq+ymFmQ2/o6m0ULiJ2XsUqVnpdzYT
WuOKOkYMxNv3GDo=
-----END PRIVATE KEY-----

The text output is given in PEM (Privacy-Enhanced Mail) format, but you can also produce a human readable text format by adding the option -text to the command line:

Private-Key: (1024 bit) modulus:
00:a4:fb:07:75:62:fa:17:4e:3d:7a:f9:94:88:d1:
dc:0a:39:53:82:63:6d:d9:90:e2:18:57:e6:18:7f:
b8:9c:04:10:0e:c6:90:aa:c3:4c:63:aa:2d:37:19:
66:07:97:1f:89:eb:ab:94:b0:a0:6e:8e:e0:c8:b5:
85:3e:ac:fc:54:9d:a5:19:25:e8:b1:4d:72:3a:57:
3d:3e:06:b7:33:f2:70:c2:a1:b0:6b:be:ae:72:e2:

26:6b:97:b4:43:0c:d4:2c:37:82:eb:58:55:ae:b7:
ca:02:71:7f:b6:03:70:9c:48:d8:57:00:77:4d:a3:
b0:a0:35:d4:c3:fa:9f:80:93

publicExponent: 65537 (0x10001)
privateExponent:
5f:b1:4f:31:32:58:b7:3d:e5:81:37:d4:84:cb:9c:
e2:bd:3f:87:e8:9e:25:cc:d8:a7:a0:0c:51:77:7b:
9f:da:e7:69:7a:99:1c:c3:2c:c2:be:49:7d:a5:18:
be:1c:54:e9:cd:ad:77:10:3f:ee:00:a7:b7:23:3a:
87:11:15:46:1c:53:1d:a9:db:9f:7d:08:39:0c:d0:
f2:9a:4a:a7:09:c2:79:c9:92:de:f6:a9:f7:b6:3d:
ff:11:6d:ce:9b:76:76:3e:f9:fa:47:d3:4b:15:b3:
7c:62:74:f0:8d:6a:30:16:61:5e:99:1d:1d:62:d8:
9c:20:3e:88:96:bb:f9:e1

prime1:
00:da:86:34:64:36:a9:c0:d3:01:e6:98:79:62:d8:
e7:09:ef:b2:6f:c0:a6:09:37:bb:6a:a8:80:90:7c:
76:9f:b3:ad:f7:b2:ec:e0:2a:e0:1e:67:fa:bf:b4:
14:c6:4f:ca:a1:dc:b3:8a:25:52:fa:c2:8e:21:04:
6f:f7:a2:2f:b1

prime2:
00:c1:46:1d:98:a7:39:96:d8:d0:4d:b7
:e4:32:50: ca:c6:22:64:65:8b:b1:85:96:9f:54:2a:
19:11:c2: 23:a1:47:5a:78:f2:60:fb:ca:36:d2:a9:
be:de:1b: 44:04:54:b2:cd:e4:5d:36:01:e0:0b:95:
90:c5:bd: 37:78:09:e9:83

exponent1:
00:a4:17:3a:a8:35:d3:ea:93:88:4d:
dc:18:ea:7c: ed:a0:ac:79:55:29:58:84:7b:71:24:
7a:3b:70:43: c5:df:ce:50:c8:3b:77:5c:2d:55:f5:ff:
26:6f:65: 8e:64:0b:ee:a0:f4:d5:6c:da:f2:20:92:8f:
9f:83: 71:80:86:2d:41

exponent2: 1d:e9:9b:25:08:24:9a:9b:1d:ca:19:
7e:e0:e6:ae: 2b:72:21:35:b2:24:1c:77:ce:5a:f9:
54:fd:f1:9d: 10:15:fa:03:a4:07:ac:90:84:61:a0:
51:e1:10:39: 3b:ae:5c:2a:0f:80:f9:9d:da:ae:10:
4c:e2:bd:c1: 3a:fb:d5:bf

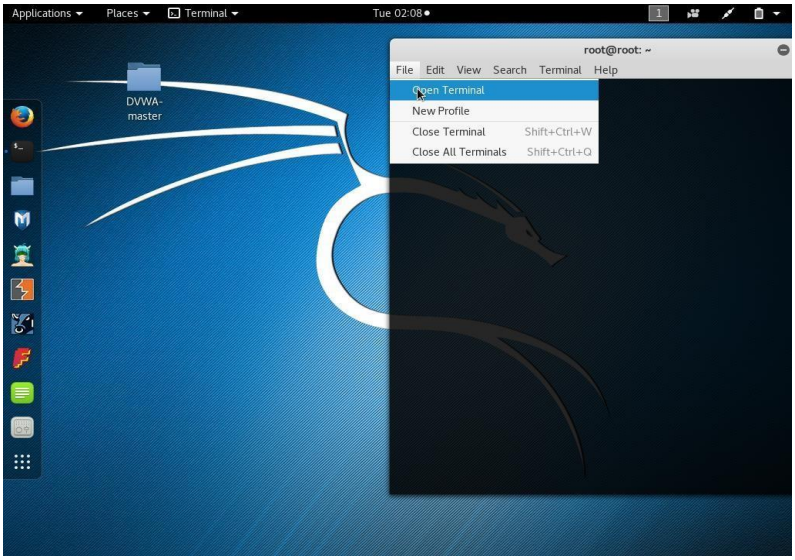
coefficient:
00:9d:80:de:5d:29:4c:23:70:b8:66:ee:51:5c:3c:
2a:32:7a:cc:7a:0b:ae:8d:e9:4e:bb:a1:80:36:6f:
7e:3b:ea:fb:29:85:99:0d:bf:a3:a9:b4:50:b8:89:
d9:7b:14:a9:59:e9:77:36:13:5a:e3:8a:3a:46:0c:
c4:db:f7:18:3a

Let's simulate a secret communication using RSA between Alice and Bob

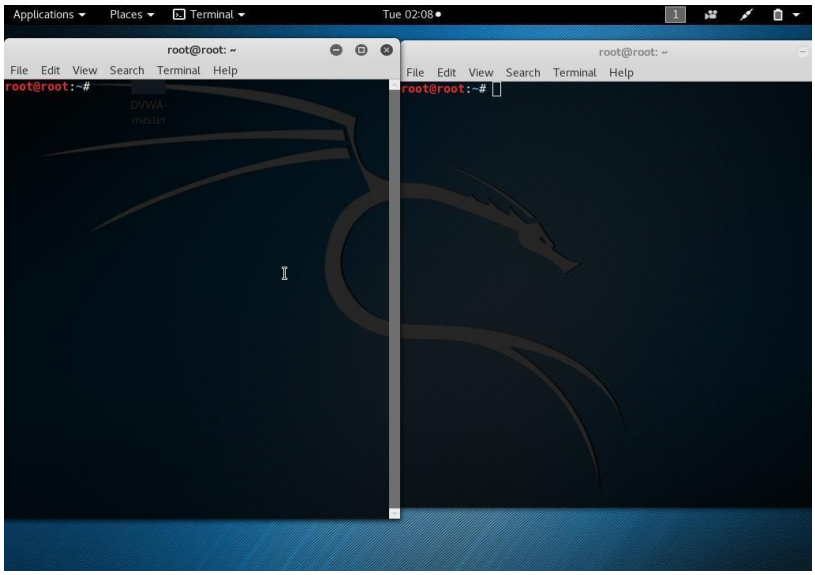
Assume there is a communication between two user Alice and Bob. It will require two terminals to work as two computers to simulate this communication.

Generating public key and private key

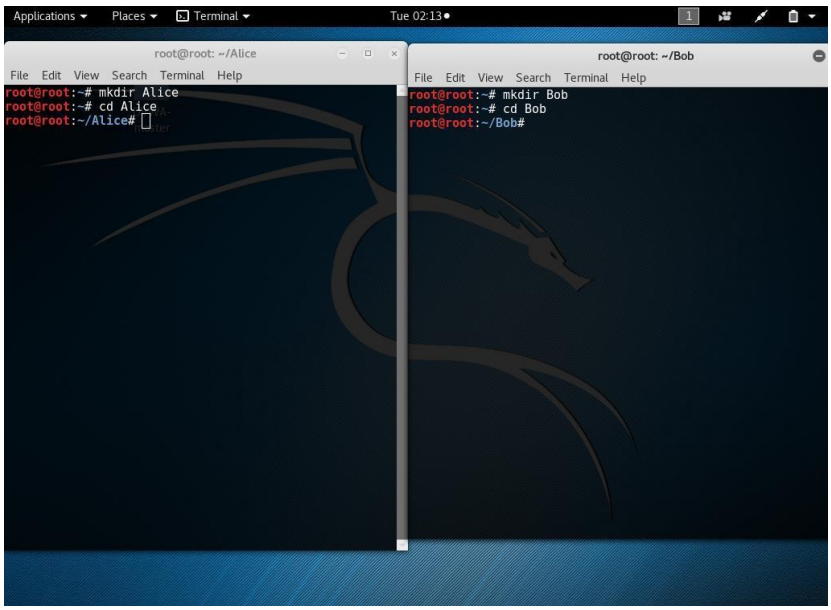
- Use two terminal and make directory for each user



- Open two terminal



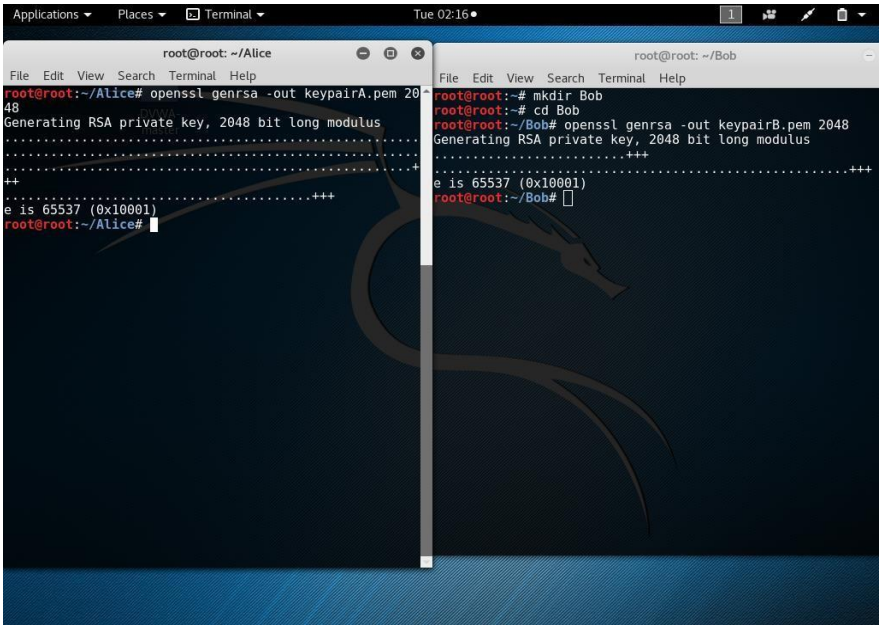
Make directory for both users



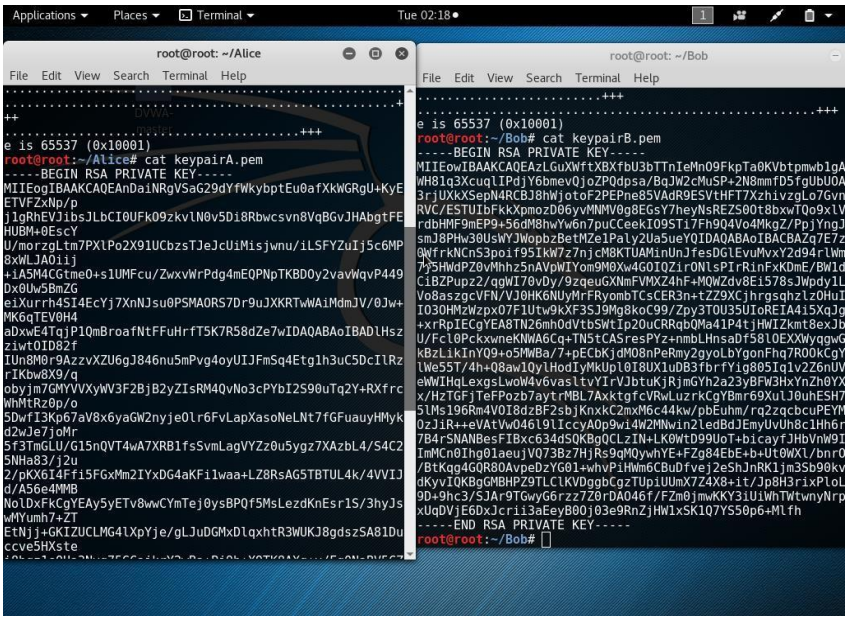
- Generate public and private key for each users (Alice and Bob)

  openssl genrsa –out keypairA.pem 2048

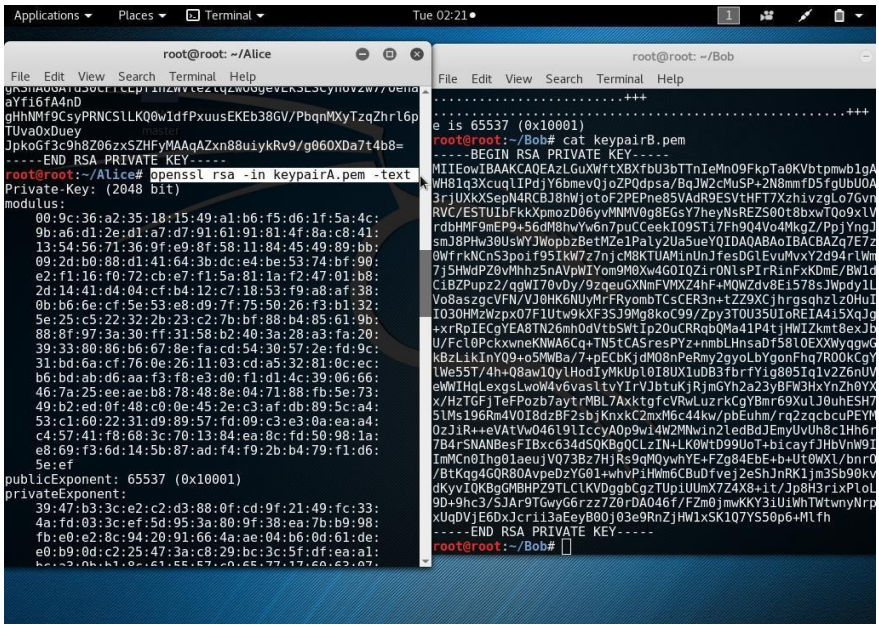
  openssl genrsa –out keypairB.pem 2048



To view the **keypair** file **cat keypairA.pem** the keypair file contain public and private key.

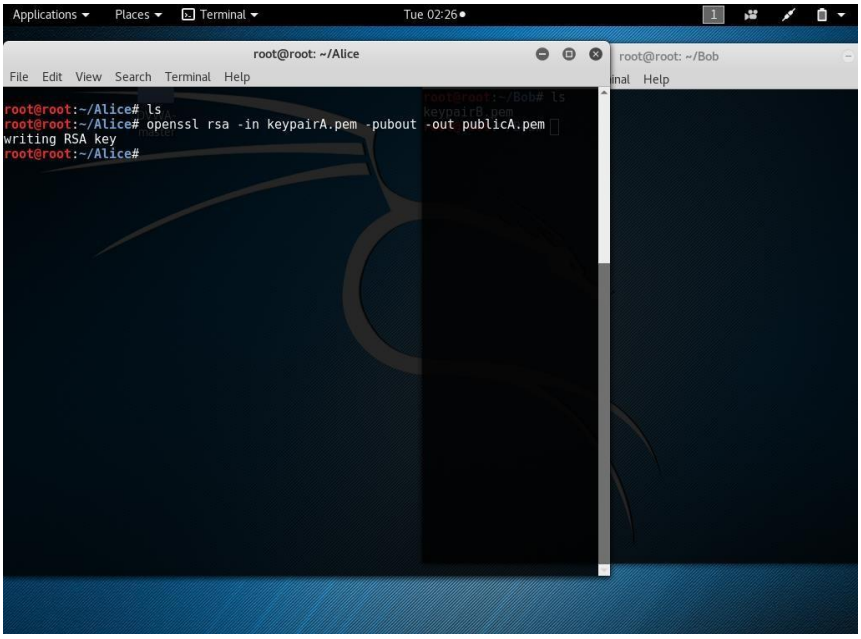


- Or **openssl rsa –in keypairA.pem -text** to view all the numbers in this key.

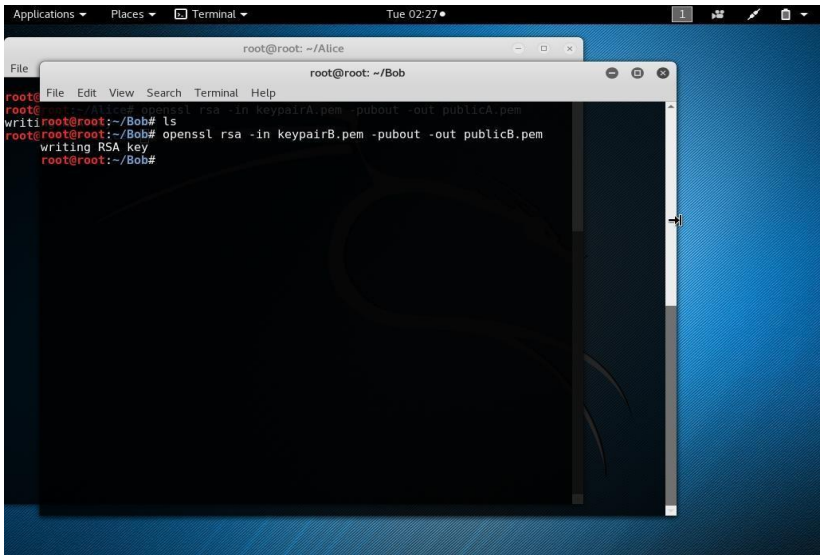


To generate the public key to send it to other user

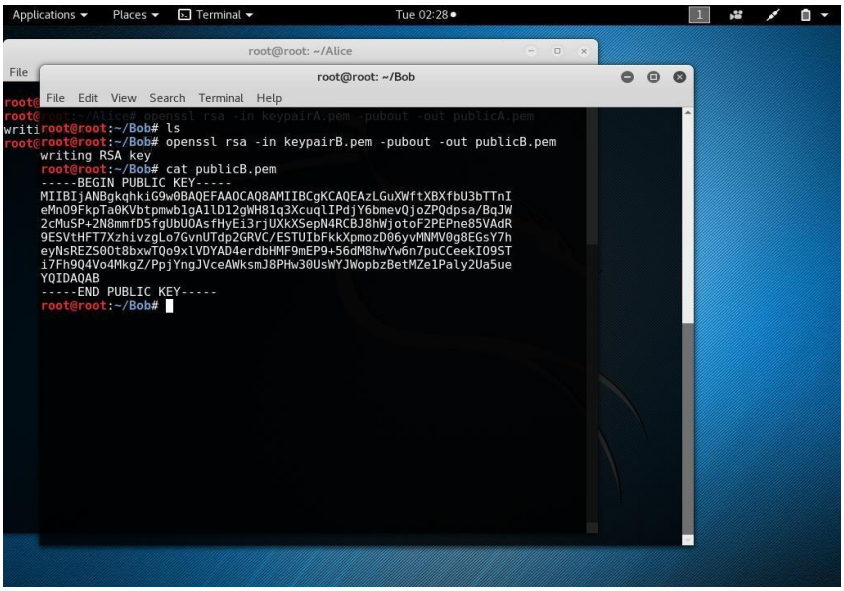
openssl rsa -in keypairA.pem -pubout -out publicA.pem



- Use the same command to generate public key for user Bob.

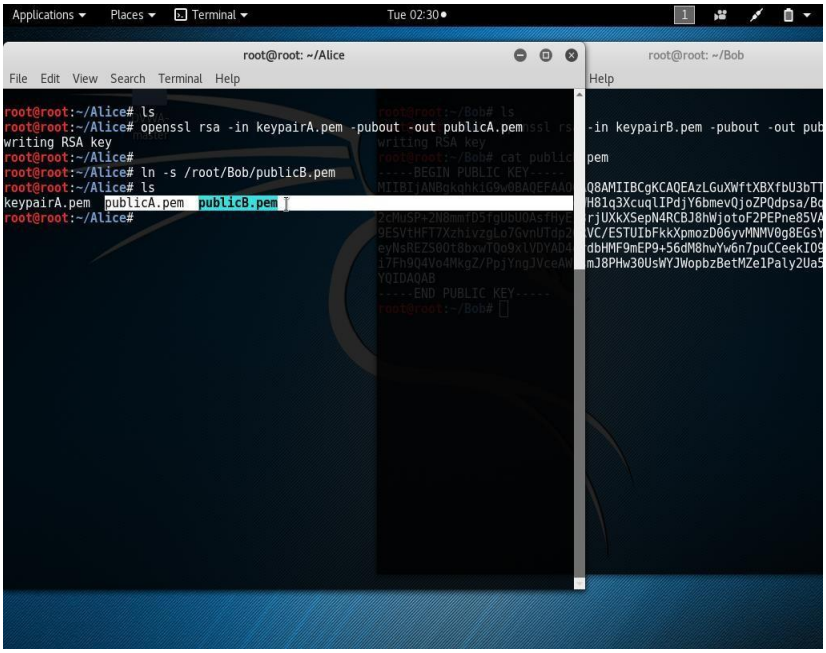


- To view the public key for Alice use **cat publicA.pem**
- To view the public key for Bob use **cat publicB.pem**



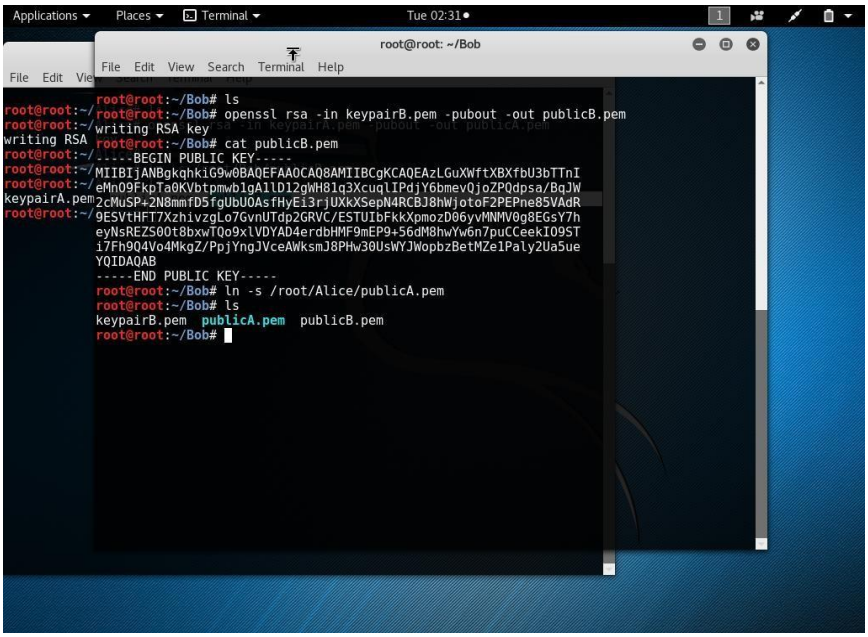
- Now Alice and Bob have to share the public key to encrypt and decrypt the files In Alice terminal copy Bob public key using the following command

  ln -s /root/Bob/publicB.pem



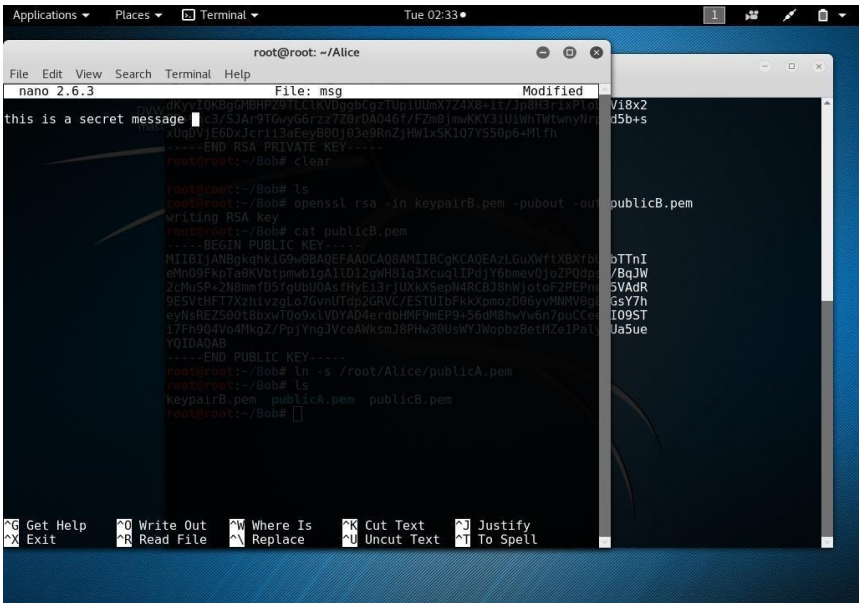
In Bob terminal copy Alice public key using the following command

ln -s /root/Alice/publicA.pem

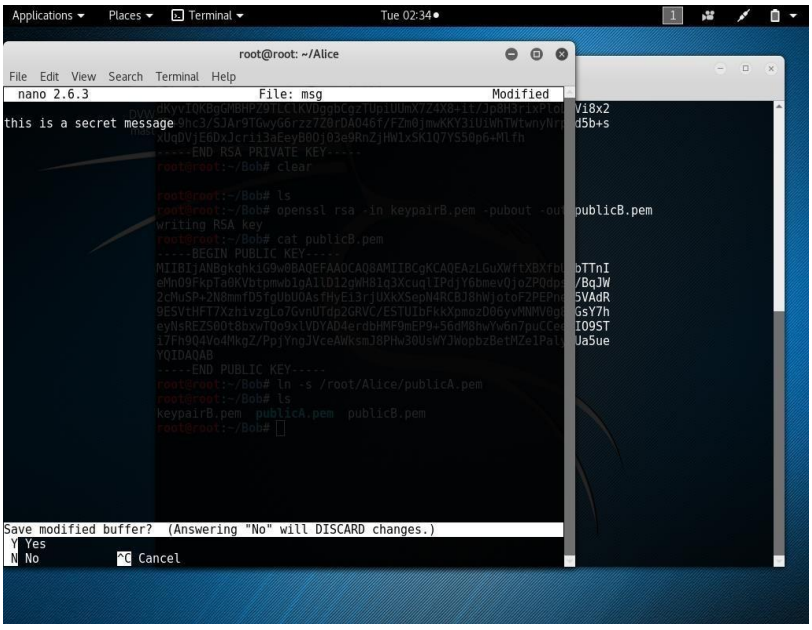


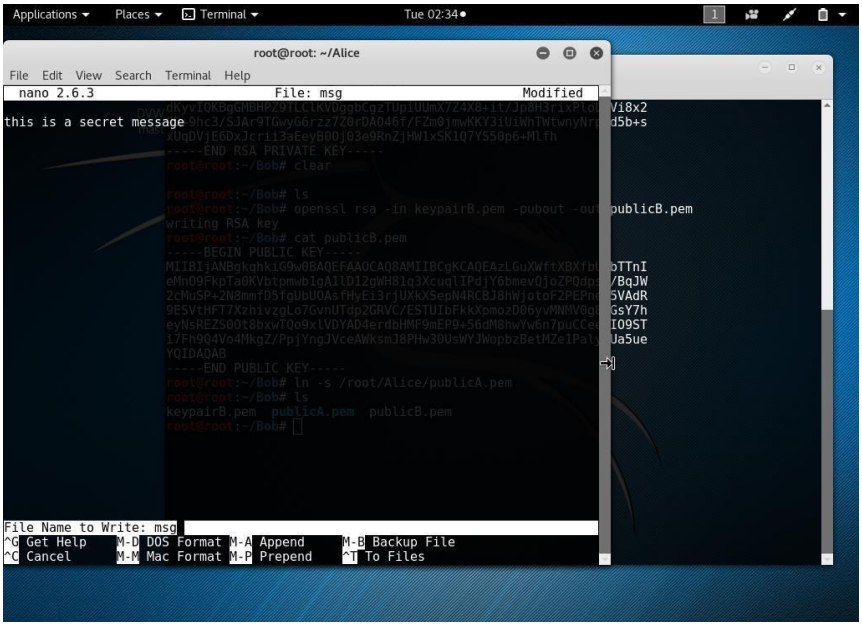
Encryption and decryption using public key and private key

- Create a file with a message that you want to encrypt using any text editor **nano msg**

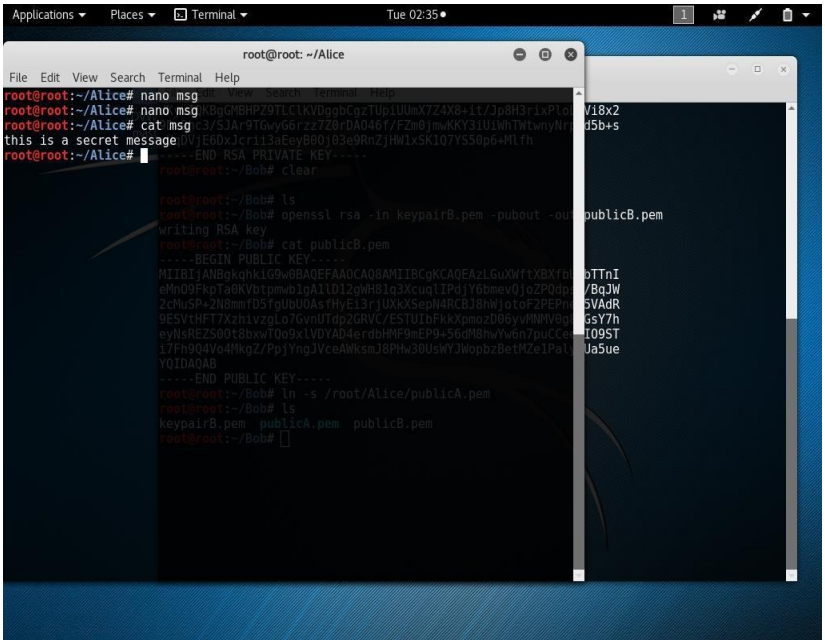


Save the file

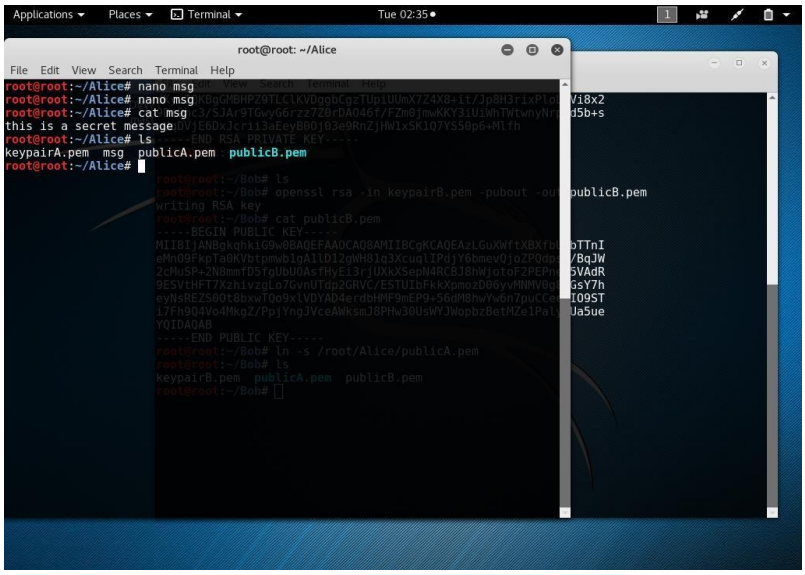




To display message in terminal use command **cat msg**

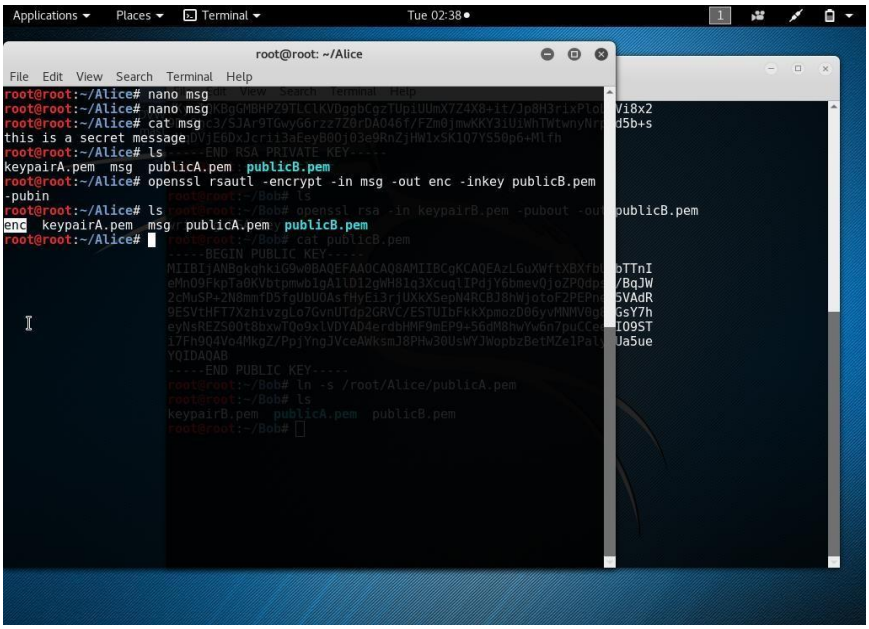


- To display public and private keys use **ls**

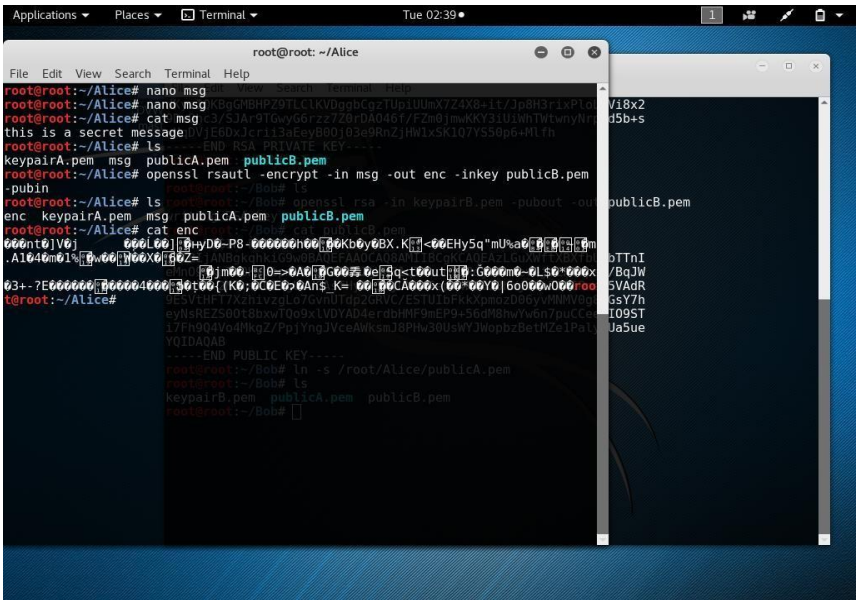


Alice will encrypt the message using the public key of Bob using this command

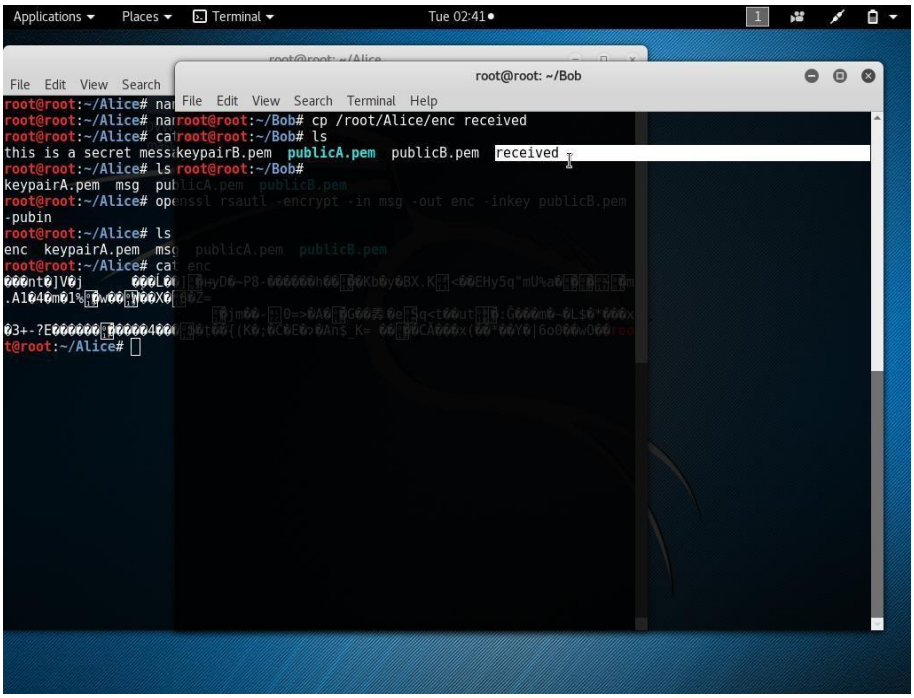
- **openssl rsautl -encrypt -in [textfile] -out enc -inkey publicB.pem pubin**
- The output file after encryption is **enc**



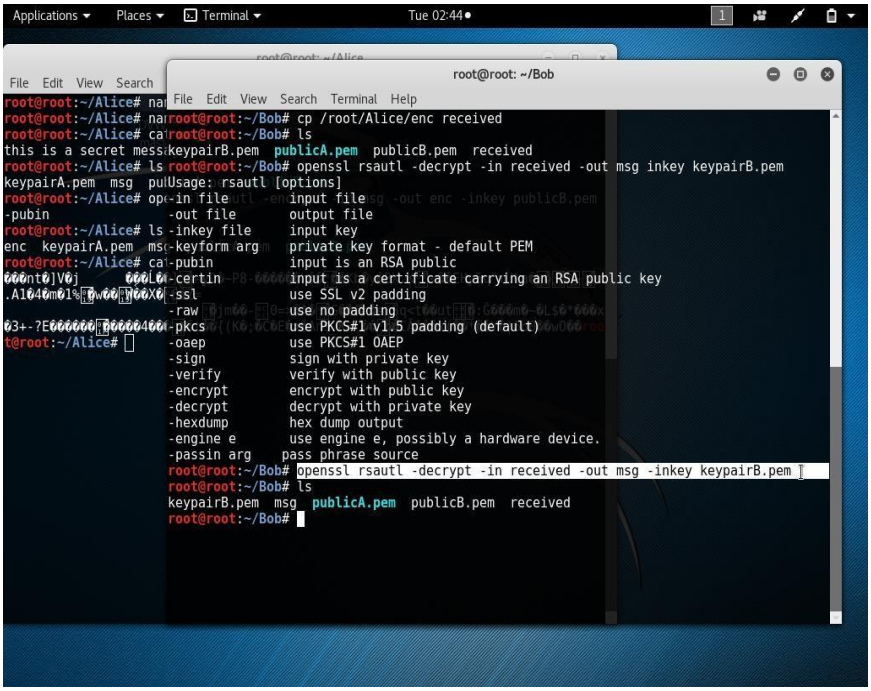
- To view the encrypted messages **cat enc**



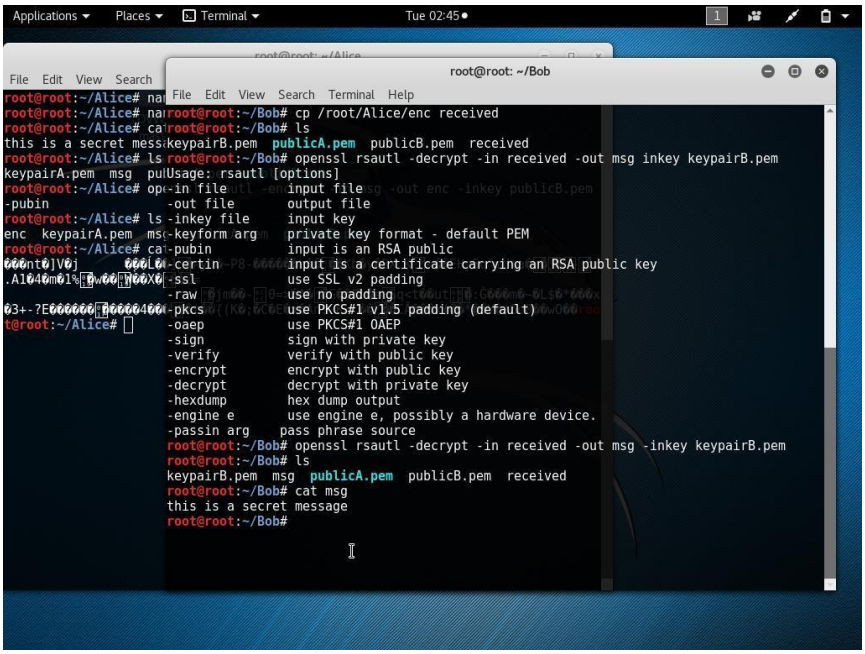
In Bob directory: copy the message from Alice directory **cp /root/Alice/enc received**



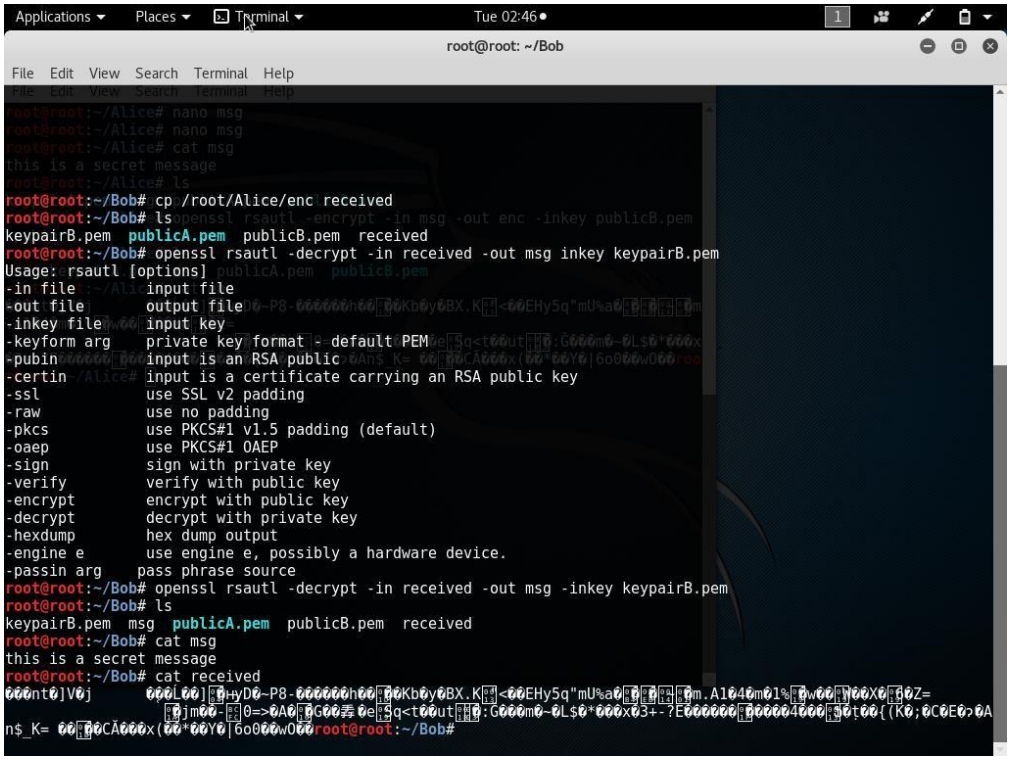
- To decrypt the message using the private key of Bob **openssl rsautl -decrypt -in received -out [textfile] -inkey keypairB.pem**



To view the messages **cat msg**



- To display received message use **cat received**


Applications
Places
Terminal
Tue 02:46
root@root: ~/Bob
File Edit View Search Terminal Help
root@root:~/Bob# cp /root/Alice/enc received
root@root:~/Bob# ls
keypairB.pem publicA.pem publicB.pem received
root@root:~/Bob# openssl rsautl -decrypt -in received -out msg inkey keypairB.pem
Usage: rsautl [options]
-in file input file
-out file output file
-inkey file input key
-keyform arg private key format - default PEM
-pubin input is an RSA public
-certin input is a certificate carrying an RSA public key
-ssl use SSL v2 padding
-raw use no padding
-pkcs use PKCS#1 v1.5 padding (default)
-oaep use PKCS#1 OAEP
-sign sign with private key
-verify verify with public key
-encrypt encrypt with public key
-decrypt decrypt with private key
-hexdump hex dump output
-engine e use engine e, possibly a hardware device.
-passin arg pass phrase source
root@root:~/Bob# openssl rsautl -decrypt -in received -out msg -inkey keypairB.pem
root@root:~/Bob# ls
keypairB.pem msg publicA.pem publicB.pem received
root@root:~/Bob# cat msg
this is a secret message
root@root:~/Bob# cat received
root@root:~/Bob#

LAB 7 : Hash Functions

Part A: Objective : Manipulate hash function using python. Hmac functions produce a "digest" or a "signature" used for message authentication. So we need to import libraries for hash functions named "hashlib" and "base64" for readable visualization digest.

Step 1 : Explore hmac1 which outputs 20 bytes = 160 bits

- import libraries hashlib and base64 :

```
10  import hashlib
11  import base64
```

- consider a secret message or data that you want to produce a digest from it, and print it:

```
18 data = "secret-message"
19 print('data=', data)
```

Step 2 : initialize the hashlib object

```
9   hash1 = hashlib.sha1(data.encode('utf-8'))
0
```

note if we do not encode the data in 'UTF-8' format, the error :

```
19  hash1 = hashlib.sha1(data)
```

```
Traceback (most recent call last):
  File "lab7partA.py", line 19, in <module>
    hash1 = hashlib.sha1(data)
TypeError: Unicode-objects must be encoded before hashing
```

Step 3 : show the output (digest) in three formats :

- show the digest in bytes format :

```
#the fingerprint in bytes
sign1bytes=hash1.digest()
print('sign1 without encoding (means in bytes) = ',sign1bytes)
```

output : b'\x9c-\xc4\xed\xca\xfb\xbd\x9b.\xed\x80\xfe\xb6<\x97%\x91\xc01\x01'

Q. is it readable output for human being ?

……

- show the digest in hexadecimal : base (0123456789abcdef). Every 4 bits a symbol

```
# the fingerprint  in hexadecimal
signt=hash1.hexdigest()
print('sign1 in hexadeciaml=', signt)
```

output : 9c2dc4edcafbbd9b2eed80feb63c972591c03101

Q. is it readable character for human being ? Is it compact ?

……

- show the digest in base64 encoding : base64 (every 6 bits a symbol)

```
# the fingerprint  in base64
signature1 = base64.b64encode(hash1.digest()).decode()
print("sign1 in base64=",signature1)
```

output : nC3E7cr7vZsu7YD+tjyXJZHAMQE=

Q. how can be sure that different outputs and encoding are coherent meaning

Is Signature1 in base64 = signt in hexadecimal ?

Here an online converter to check by yourself : https://base64.guru/converter/encode/hex

Step 4 : verifying the size of the digest (output)

sha1 produce 20 bytes. To verify the size we can use :

```
print('size of sign1 in bits= ',8*hash1.digest_size)
```

output : size of sign1 in bits= 160

Q. is it correct ?

……

Step 5 : now explore other hash functions as for sha256 and sha512 and repeat steps 2-3-4.

```
# here you can explore more hash variants as for sha256 and sha512
#and repeat the same steps
hash2 = hashlib.sha256(data.encode('utf-8'))
hash3 = hashlib.sha512(data.encode('utf-8'))
```

Appendix : Base64 encoding table (6-bits to base64 symbol)

The Base64 index table:

Index	Binary	Char	Index	Binary	Char	Index	Binary	Char	Index	Binary	Char
0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	0
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/
Padding		=									

source : https://en.wikipedia.org/wiki/Base64

Part B : using update()

hash.**update**(*data*)

Update the hash object with the bytes-like object. Repeated calls are equivalent to a single call with the concatenation of all the arguments: m.update(a); m.update(b) is equivalent to m.update(a+b).

Write the following code :

```
>>> import hashlib
>>> m = hashlib.sha256()
>>> m.update(b"Nobody inspects")
>>> m.update(b" the spammish repetition")
>>> m.digest()
b'\x03\x1e\xdd}Ae\x15\x93\xc5\xfe\\\x00o\xa5u+7\xfd\xdf\xf7\xbcN\x84:\xa6\xaf\x0c\x95\x0fK\x
>>> m.digest_size
32
>>> m.block_size
64
```

and compare the result with the following code :

```
>>> hashlib.sha224(b"Nobody inspects the spammish repetition").hexdigest()
'a4337bc45a8fc544c03f52dc550cd6e1e87021bc896588bd79e901e2'
```

Make the necessary modifications in above codes (they will produce the same digests if you use the same hash functions and the same data no ?)

LAB 8 : Keyed Hashing and MAC

<u>Part A:</u> using hmac module

Imagine that Alice's document management system must receive documents from Bob. Alice has to be certain each message has not been modified in transit by Mallory. Alice and Bob agree on a protocol:

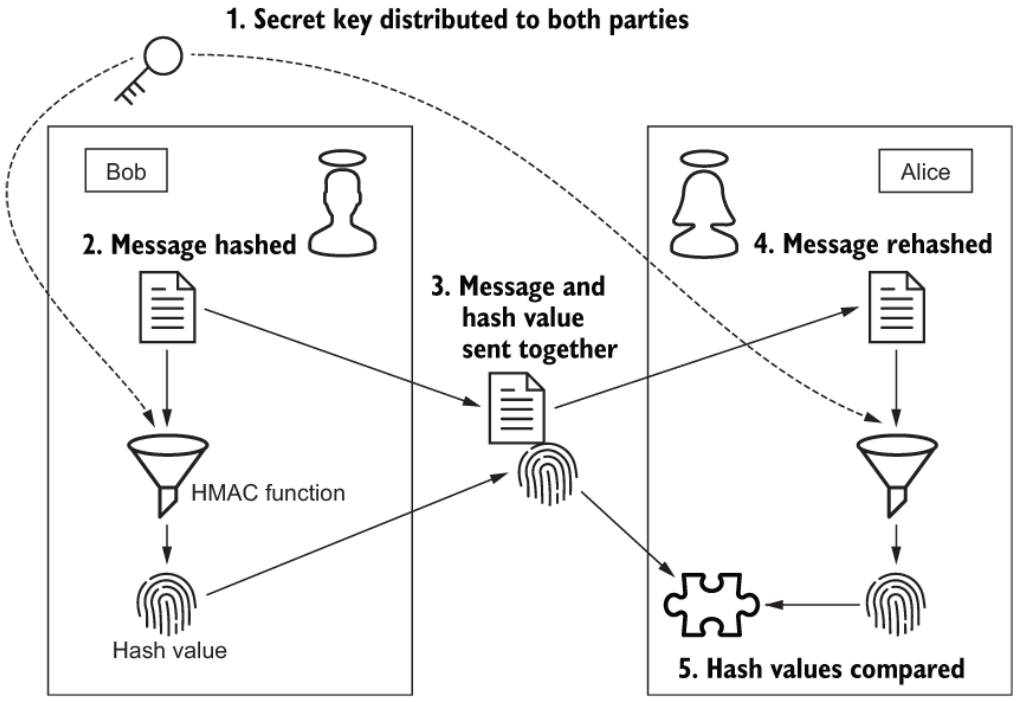
Alice and Bob share a secret key.

1) Bob hashes a document with his copy of the key and an HMAC function.
2) Bob sends the document and the hash value to Alice in a json file format.
3) Alice hashes the document with her copy of the key and an HMAC function.
4) Alice compares her hash value to Bob's hash value.

The following Figure illustrates this protocol. If the received hash value matches the recomputed hash value, Alice can conclude two facts:

The message was sent by someone with the same key, presumably Bob.

- Mallory couldn't have modified the message in transit



Bob's Side of the protocol :

Bob's implementation of his side of the protocol, shown in the following listing, uses HMAC-SHA256 to hash his message before sending it to Alice:

```
import hashlib
import hmac
import json

hmac_sha256 = hmac.new(b'shared_key',
digestmod=hashlib.sha256)
message = b'from Bob to Alice'

hmac_sha256.update(message)

hash_value = hmac_sha256.hexdigest()


authenticated_msg =
{
   'message': list(message),

   'hash_value': hash_value, }

outbound_msg_to_alice = json.dumps(authenticated_msg)
```

Alice's Side of the protocol :

Alice receive the message (json file) and do the following :

- Alice computes her own hash value.
- Alice compares both hash values

Apply the following code for that :

```
import hashlib
import hmac
import json

authenticated_msg = json.loads(inbound_msg_from_bob)
message = bytes(authenticated_msg['message'])

hmac_sha256 = hmac.new(b'shared_key',
digestmod=hashlib.sha256)
hmac_sha256.update(message)

hash_value = hmac_sha256.hexdigest()


if hash_value == authenticated_msg['hash_value']:

    print('trust message')
    ...
```

Part B: Random numbers generation

- Manipulate random number generation using python.

```
from random import *

# Generate a pseudo-random number between 0 and 1.
print(random())

#Generate a random number between 1 and 100
#To generate a whole number (integer) between one and one hundred use:
print(randint(1, 100)) # Pick a random number between 1 and 100.
#This will print a random integer. If you want to store it in a variable you can use:
x = randint(1, 100) # Pick a random number between 1 and 100.
print(x)
#Random number between 1 and 10
#To generate a random floating point number between 1 and 10 you can use the uniform() function
print(uniform(1, 10))
#Picking a random item from a list
items = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
shuffle(items)
print(items)

#To pick a random number from a list:
items = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
x = sample(items, 1) # Pick a random item from the list
print (x[0])
y = sample(items, 4) # Pick 4 random items from the list
print (y)
#We can do the same thing with a list of strings:
items = ['Alissa','Alice','Marco','Melissa','Sandra','Steve']
x = sample(items, 1) # Pick a random item from the list
print (x[0])
y = sample(items, 4) # Pick 4 random items from the list
print (y)
```

LAB 9 :

LAB 10 : Image cryptography and steganography

Part A : Image encryption and decryption with fernet

use a jpg image to simulate encryption/decryption of an image. The encryption process here is not specific to just images and can be used on a variety of data types. The takeaway from this program should be the full encryption at the file level and not the pixel level.

import fernet :

```
from cryptography.fernet import Fernet
```

Write the encryption function (transmitter side) :

```
def encrypt(filename, newfile, key):
    f = Fernet(key)
    with open(filename, "rb") as file:
  # read all file data
        file_data = file.read()
  # encrypt data
    encrypted_data = f.encrypt(file_data)
  # write the encrypted file
    with open(newfile, "wb") as file:
        file.write(encrypted_data)
```

implement the decryption function (receiver side) :

```
def decrypt(filename, newfile, key):
    f = Fernet(key)
    with open(filename, "rb") as file:
# read the encrypted data
        encrypted_data = file.read()
# decrypt data
    decrypted_data = f.decrypt(encrypted_data)
# write the original file
    with open(newfile, "wb") as file:
        file.write(decrypted_data)
```

and use them in the main code :

```
key = Fernet.generate_key()
enc = encrypt("lena.jpg", "e_lena.jpg", key)
dec = decrypt("e_lena.jpg", "d_lena.jpg", key)
```



original **encrypted** **decrypted**

While the encrypted image is now unreadable, it may also throw an error depending

on the software you are using. This isn't a showstopper, though, as you can still email the file anywhere and you will not have to worry about someone decrypting it without the key.

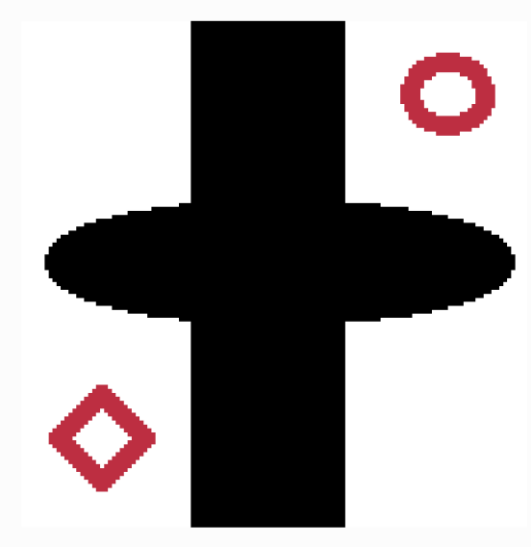
Part B : ECB, CBC modes in image encryption (using pycryptodome)

1) Exploring a Simple ECB Mode Example

For our first example, we will examine how to produce the ECB mode encrypted

image. First, we start out with a bitmap file that has a distinct pattern, as shown

in Figure

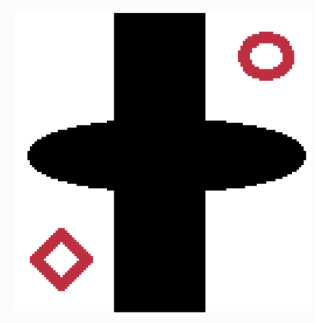


Import Crypto and initialise the encryption machine with a key of size 128 bits = 16 (bytes)*8. Use ECB mode and read the image blocks and encrypt them with AES-ECB. Consider the fact that AES use plaintext blocks of 16 bytes (128 bits) length. So we need to pad the plaintext image blocks to be multiple of 16 bytes.

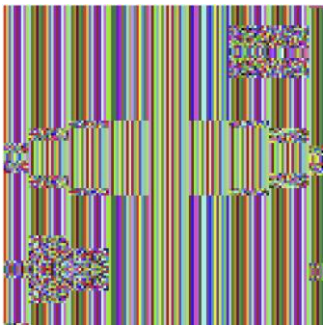
```
from Crypto.Cipher import AES

key = b"aaaabbbbccccdddd"
cipher = AES.new(key, AES.MODE_ECB)

with open("plane.bmp", "rb") as f:
    byteblock = f.read()
print (len(byteblock))
byteblock_trimmed = byteblock[64:-6]
ciphertext = cipher.encrypt(byteblock_trimmed)
ciphertext = byteblock[0:64] + ciphertext + byteblock[-6:]
with open("e_plane.bmp", "wb") as f:
    f.write(ciphertext)
```

And in the decryption side :

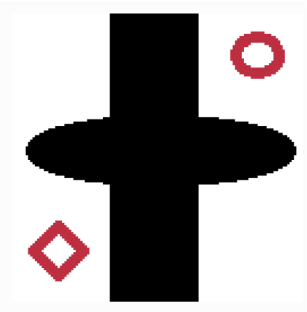
```
# decrypt using the reverse process
with open("e_plane.bmp", "rb") as f:
    byteblock = f.read()

pad = len(byteblock)%16 * -1
byteblock_trimmed = byteblock[64:pad]
plaintext = cipher.decrypt(byteblock_trimmed)
plaintext = byteblock[0:64] + plaintext + byteblock[pad:]
with open("d_plane.bmp", "wb") as f:
    byteblock = f.write(plaintext)
print ("done")
```



original



encrypted



decrypted

2) Exploring a Simple CBC Mode Example

Now that you understand how to encrypt and decrypt using ECB block mode, we will examine the CBC mode. One important difference between the two modes is the use of an initialization vector (IV) and the specification of the block mode, which in this case is AES.MODE _ CBC:

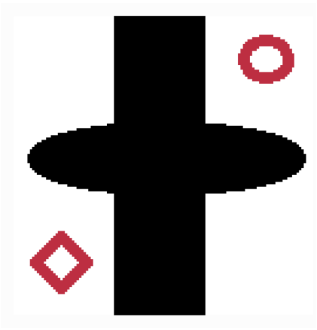
change the above code in ECB to incorporate the following change in the encryption side :

```
iv = b"1111222233334444"
key = b"aaaabbbbccccdddd"
cipher = AES.new(key, AES.MODE_CBC, iv)
```

and the following change in the decryption side :

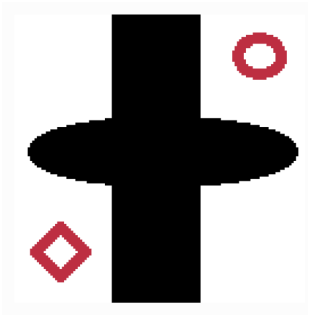
```
24
25  cipherd = AES.new(key, AES.MODE_CBC, iv)
26
```

And this should be the result of encrypting a highly redundant image with CBC mode :



original

encrypted



decrypted

Part C: Steganography

import cryptosteganography module

first we will embed a text message into the image nebula.png

```
from cryptosteganography import CryptoSteganography

key = "1111222233334444!"
crypto_steganography = CryptoSteganography(key)
print()

```

```
################# transmitter
print('The program is looking for an image named nebula.png\n')
origfile = "nebula.png"
print('The image with the hidden message will be called mnebula_m..png\n')
modfile = "mnebula_m.png"
secretMsg = ""
message1 = "Sympathy for the favorite nation, facilitating the illusion of an imaginary common "
message2 = "interest in cases where no real common interest exists, and infusing into one the "
message3 = "enmities of the other, betrays the former into a participation in the uarrels nd "
message4 = "wars of the latter without adequate inducement or justification."
secretMsg = secretMsg.join([message1, message2, message3, message4])
crypto_steganography.hide(origfile, modfile, secretMsg)

```

right receiver with the correct key : he wil extract the concealed message from the image mnebula_m.png and display it to the screen.

```
#### receiver ###############333
secret = crypto_steganography.retrieve(modfile)
print("The secret that is hidden in the file is:\n")
print(secret)
print()

```

wrong receiver with the wrong key : another receiver having the wrong key will try to extract the concealed message buit do not succeed. he will get “none”.

```
################### wrong receiver ########################
print('Now we will try the wronge secret.\n')
key = "AnotherKey"
crypto_steganography = CryptoSteganography(key)
secret = crypto_steganography.retrieve(modfile)
print('The secret message is: {} \n'.format(secret))
```

Now we will embed a mp3 file (binary file) into an image :

for that you need an image + mp3 file

```python
from cryptosteganography import CryptoSteganography


# open sound file
mediafile = 'song.mp3'
message = None
with open(mediafile, "rb") as f:
    # I just embeded a small portion of the song, not all of it
    message = f.read(120000)

print()
print('The program is looking for an image named nebula.png\n')
origfile = "nebula.png"
print('The image with the hidden audio file will be called
steg_audio_nebula.png\n')
modfile = "steg_audio_nebula.png"
key = "1111222233334444!"
crypto_steganography = CryptoSteganography(key)
crypto_steganography.hide(origfile, modfile, message)
print('The extracted data will be called decrypted_song.mp3 \n')
decrypted = 'decrypted_song.mp3'
secret_bin = crypto_steganography.retrieve(modfile)
# Save the data to a new file
with open(decrypted, 'wb') as f:
    f.write(secret_bin)
```

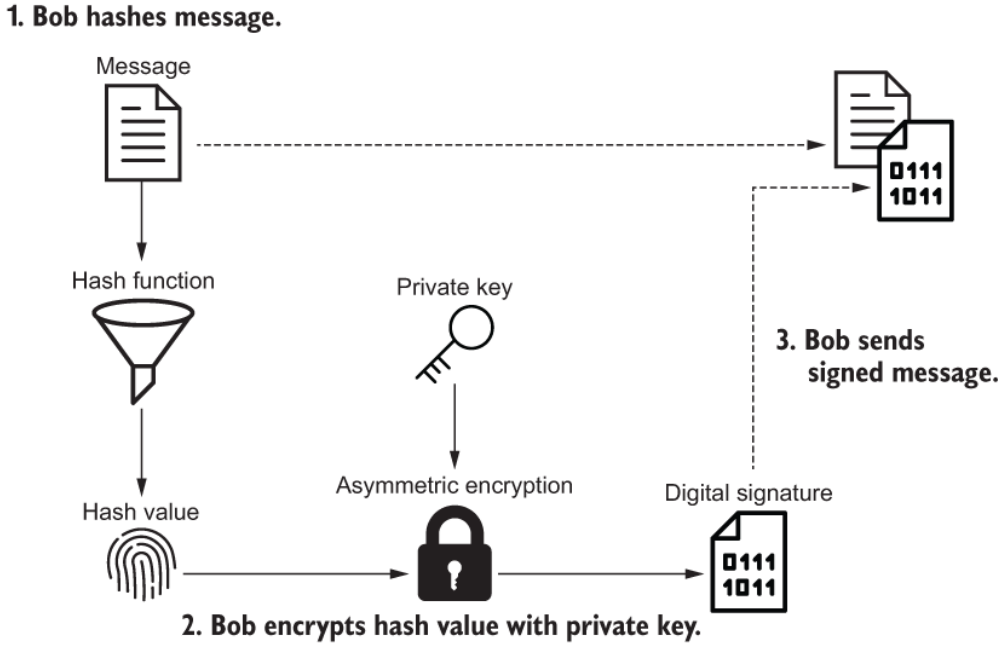
LAB11 : Digital Signature

PartA: Digital Signature generation and verification in python

Digital signatures go one step beyond data authentication and data integrity to ensure nonrepudiation. A digital signature allows anyone, not just the receiver, to answer two questions: Who sent the message? Has the message been modified in transit? A digital signature shares many things in common with a handwritten signature:

- Both signature types are unique to the signer.
- Both signature types can be used to legally bind the signer to a contract.
- Both signature types are difficult to forge.

Digital signatures are traditionally created by combining a hash function with public-key encryption. To digitally sign a message, the sender first hashes the message. The hash value and the sender's private key then become the input to an asymmetric encryption algorithm; the output of this algorithm is the message sender's digital signature. In other words, the plaintext is a hash value, and the ciphertext is a digital signature. The message and the digital signature are then transmitted together. Figure 5.2 depicts how Bob would implement this protocol.



Signing the message :

```
import json
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes

message = b'from Bob to Alice'

padding_config = padding.PSS(
    mgf=padding.MGF1(hashes.SHA256()),
    salt_length=padding.PSS.MAX_LENGTH)

private_key = load_rsa_private_key()
signature = private_key.sign(
    message,
    padding_config,
    hashes.SHA256())

signed_msg = {
    'message': list(message),
    'signature': list(signature),
}
outbound_msg_to_alice = json.dumps(signed_msg)
```

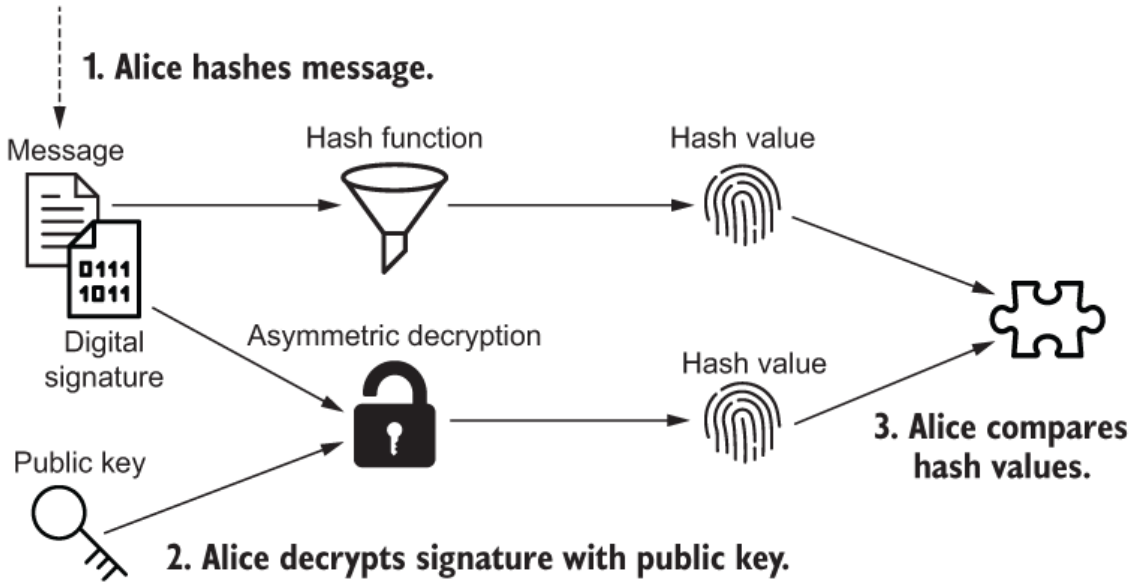
Note :

Alice Verification of the signature :

After Alice receives Bob's message and digital signature, she does three things:

1. She hashes the message.
2. She decrypts the signature with Bob's public key.
3. She compares the hash values.

If Alice's hash value matches the decrypted hash value, she knows the message can be trusted. Figure 5.3 depicts how Alice, the receiver, implements her side of this protocol.




```
import json
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.exceptions import InvalidSignature


def receive(inbound_msg_from_bob):
    signed_msg = json.loads(inbound_msg_from_bob)
    message = bytes(signed_msg['message'])
    signature = bytes(signed_msg['signature'])

    padding_config = padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH)

    private_key = load_rsa_private_key()
    try:
        private_key.public_key().verify(
            signature,
            message,
            padding_config,
            hashes.SHA256())
        print('Trust message')
    except InvalidSignature:
        print('Do not trust message')
```

make the necessary changes in order to write in the same python file the following :

generate the RSA keys, (bob) create a signature, (bob) serialize it in a json file, (alice) loads the json file, (alice) extract the message and signature, verify the signature and print the result (accept or refuse the message).

We give the RSA generation code using cryptography.hazmat:

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import rsa

private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=3072,
    backend=default_backend(), )

public_key = private_key.public_key()
```

This lab can be further improved when you separate Bob and Alice tasks in two differents scripts like in lab 8.

you just need to serialize the RSA keys because you will generate them in Bob's side and you need to distribute the public key to Alice (not generating rsa keys again).

To serialize the keys:

Listing 5.2 RSA key-pair serialization in Python

```
private_bytes = private_key.private_bytes
(
    encoding=serialization.Encoding.PEM,

    format=serialization.PrivateFormat.PKCS8,

    encryption_algorithm=serialization.NoEncryption(), )
```

```
with open('private_key.pem', 'xb') as private_file:
    private_file.write(private_bytes)

public_bytes = public_key.public_bytes
(
    encoding=serialization.Encoding.PEM,
    format=serialization.PublicFormat.SubjectPublicKeyInfo, )

with open('public_key.pem', 'xb') as public_file:
    public_file.write(public_bytes)
```

Bob and Alice need to load the private key (the public key) respectively as follows :

Listing 5.3 RSA key-pair deserialization in Python

```
with open('private_key.pem', 'rb') as private_file:
   loaded_private_key = serialization.load_pem_private_key
(
       private_file.read(),
       password=None,
       backend=default_backend()
   )
```

```
with open('public_key.pem', 'rb') as public_file:

   loaded_public_key = serialization.load_pem_public_key
(
       public_file.read(),

       backend=default_backend()

   )
```

PartB : Load a certificate from a web server using python and parse it using openssl

write the following code :

lab11partB.py

```
1  import ssl
2  address = ('wikipedia.org', 443)
3  certificate = ssl.get_server_certificate(address)
4  print(certificate)
5
```

in the shell terminal type :

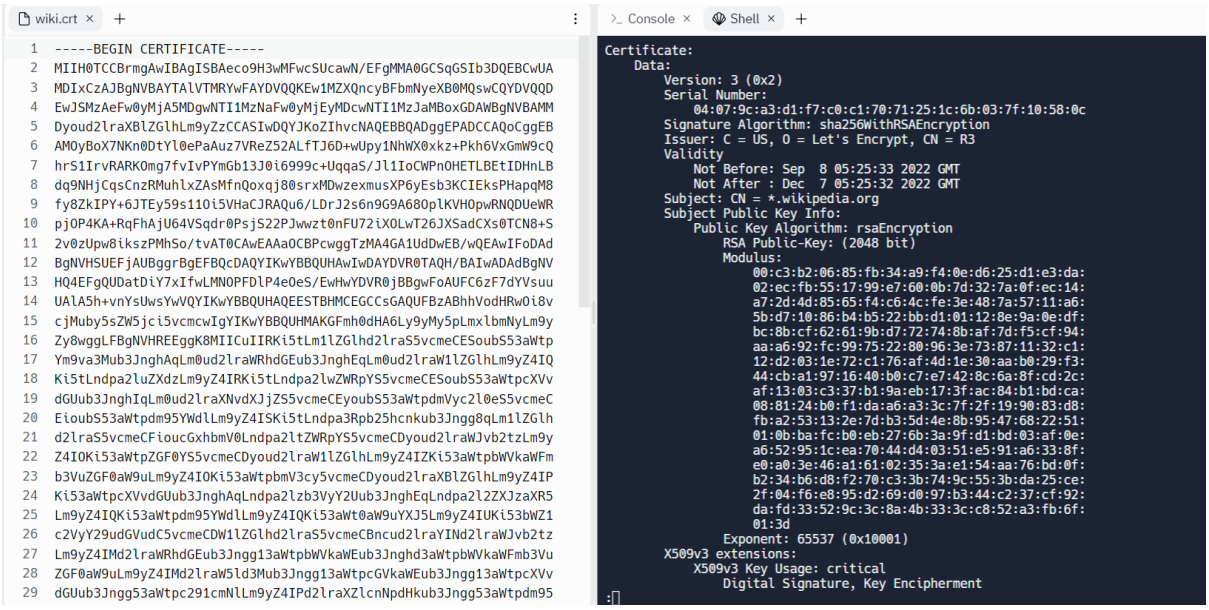
python lab11partB.py > wiki.crt

This will download the certificate and write it to a file named wiki.crt:

now decode the certificate wiki.cert using Openssl

The anatomy of an X.509 certificate is composed of a common set of fields. You can develop a greater appreciation for TLS authentication by thinking about these fields from a browser's perspective. The following openssl command demonstrates how to display these fields in human-readable format:

$ openssl x509 -in wikipedia.crt -text -noout | less



Before a browser can trust the server, it will parse the certificate and probe each field individually. Let's examine some of the more important fields:

Subject : ……

Issuer: ……

Subject's public key (modulus): ……

Exponent :

Certificate validity period: ……

Certificate authority signature : ……..

Signature Algorithm : ……..

Signature : ……

Lab 12 : Key Management of Symmetric cryptography

Part A: Diffie-Hellman

The Diffie-Hellman algorithm was developed to create secure communications over a public network using ECC to generate points on the curve and get the secret key using parameters; for our exploration we will consider four variables that include P, G, A, B:

P: One prime number; publicly available.

G: A primitive root of P. You may remember that a primitive root of a prime is an integer such that the modulus has multiplicative order; publicly available.

A: A user (Alice) picks private values for A and B and use them to generate a key to exchange publicly with a second user (Bob).

B: The second user (Bob), receives the key from Alice and uses it to generate a secret key; this gives both users the same secret key to encrypt.

To get a better understanding, review the following five steps:

1. Alice and Bob get public numbers P = 23, G = 9.

2. Each user selects a private key:

   - Alice selected a private key a = 4
   - Bob selected a private key b = 3

3. Each user computes public values:

   - Alice: x = (9^4 mod 23) = (6561 mod 23) = 6
   - Bob: y = (9^3 mod 23) = (729 mod 23) = 16

4. Alice and Bob exchange public numbers:

   - Alice receives public key y = 16
   - Bob receives public key x = 6

5. Alice and Bob compute symmetric keys:

   - Alice: ka = y^a mod p = 65536 mod 23 = 9
   - Bob: kb = x^b mod p = 216 mod 23 = 9

The completed process generates 9, which is the shared secret. Notice this value was never shared between the two parties.

Part B : PKI infrastructure

generate RSA keys and display them to be used later to encrypt a session key or a plaintext :

lect12partB.py ×

```
#ch8_Generate_RSA_Certs.py
from Crypto.PublicKey import RSA
#Generate a public/private key pair using 4096 bits key length (512 bytes)
new_key = RSA.generate(4096, e=65537)
#The private key in PEM format
private_key = new_key.exportKey("PEM")

#The public key in PEM Format
public_key = new_key.publickey().exportKey("PEM")
print (private_key)
fd = open("private_key.pem", "wb")
fd.write(private_key)
fd.close()
print (public_key)
fd = open("public_key.pem", "wb")
fd.write(public_key)
fd.close()
```


~/practical-crypto-course$ python lect12partB.py
b'-----BEGIN RSA PRIVATE KEY-----\nMIIJKQIBAAKCAgEA2/B4shpNAOEIHqCP1CA5sau1mK7P9+hSEyIlMsHx8su9zPFP\nhABT3gWF
PUD8kNEtx9v7nm4MmzVF4Y+e+ZBA222lMHQG8ja2SmdepJc6TmSQRjAC\nLdGtK9+aVRST/gV5jHOhVUIwbBzAVZFhxggFfL52P9hLHWvlu6Y
LF7NoY+mp9gDu\nkOMb+WszQbyQZMKnAreWMKquWVBIDshbNXdmjYaeNE6KRd4DR732a2qvslaIthRy\n8Bj84+RWP3qsGBI4gSdxDBAL0PcN
8aGfZyUbIZKeSh51Yx+2OcX9P/UWSkCZ9iTf\n5P9ZtAXw6LRqrVpLtT9EgY5qNGHtwZNoNqI1mW6oacJexze9Y9mXTshs7d3IWVVk\nNoUSz
7F68A5e+kAk0NDQwgI5lNdzOkNhxjvjalfwuIA1Tpwlp54YLx909gpuRIRI\nNs9ZxIq/LRgiWDqm60daHK8dKtc0g0th5ntQ9yrn23LrbGfq
agr6HK6lynh+58Ka\noRGOqMwlS0sdbS7tQ8VzV04qJjIUp3rlaICqgesE/mDCFnpIgVbwvd54uDbQQqmB\nJJ8t9u8VnmsMgOt+dRyLL2IWt
1HfDhymgcnB09+cY2rCsuqYbJwdniNUdG4QRy/S\nAe9sZcz4ui/qz3AqLtOQwF1f9bX0sZTFcJ5jKa7iRf3QpYbA5yVHJJVI3H8CAwEA\nAQ
KCAgAC0/S5VYfn8FIfvVtxaXipxYDFlc8sFjCio63BbqejF5Dj3GEesWJircTT\neiJv5OtRxo0AXH83qlexBB0R2Gj9E/aBHlIjNdDiOVyzg
ROe2RbXJYbhpUXfaotG\nzMeaXIz+NqkRuj16e7l9YCvXAANnYeDjW+9F/D0tTH/tWME0MledHNCfIaHFHeZq\n0fcz/yIRRuBc1SpTZlTcMS
H7GK3uOUI6KfMH3Z2X2Tlx6CKaMxf5lLRloeyegYZv\ntS7BSyVZg8hQVLnSK1sNNabMYg3B62tdnfFPH6PthOjctFS1Cl+bFUj2Xid+c7/F\
nPjdLmKhLNvpf2r+YLNTXHa4ZWxHqq7CptEztfx7Va2QSM8krwO8iFPLrPJp6p39P\ndogw+N+ws9nJEdXfad3vf8HOEzfK7ju3vd0GOoYx58
S1o+et00uV8odGgFzEIkE4\nuPyFDdmxHIvfXq2Y5pqanH5NG8I4zp1LwF3J40yErjn3ci1+lyWobZIgX1x5Y4gS\nX73sC4HsTmPqYWVcaEv
SWkNBRQZH6K/8Q0NWTPpJAYLMxEq9vGHE0+UP+3uuhLD5\nqKI1LZ1S9kcn3rA24R/1vlLST8Q367O63+dXD1+sfVsafXvk1Jk+dZDODAVmFK
fW\nBHOXVssaV6Jq8KJb777/LizZCFw+Lddl3N+PcjABBhEbqm0PAQKCAQEA40ZUfsIr\nZKm0L5i/eL4i+1DWou9bYiRk98Wdu3YsDwtsakZ
otzzfZwjOuqFkMw4lNpEaRXv5\nYIn2bzGSnMfMH+EozI5+NwFbhqh5qJFgWmaFojgKgjyV+IjMJXCUQcbdxgAhEH/P\nmCqyEGeIbhmoVZSZ
DFRwYUrCWONB2YTBZ39qsw0tTEFwnem7/Pr4Y1Ld2FOiNjF5\nmis3pZ0bO9tbzUEkYZSxP7IhxxGFbkWgkdJFlw500q8Tfr4G5CK7H35+34e
x4jmD\nBvjPiS8/SM0MwmfXiedCNemntVS2H3rVVRCrrDYXNcRg0PwdL0jspmORLvt6iClp\nGRlFIPGyRJSFHQKCAQEA97zMdxC4kcLqIKFQ
+Zg/tqczr1EnbveNxUNo2k44zhAA\nfUc71qZ0tUWMo+ZOr46Kl0Cja4EAToR5F9P0oKfG/MFjlOxyOWDEDC7pCyNi5x0R\nHfK0PGGFAt8xw
GZNjGA0rzP4qg3DXgFYXDLhk0y1yiGGcKMrYDXmctByAbV+yaqp\n/US2+eLlRyLS9bswB5sAvalgCkgJVjcDrb15mDuHnY7ANLdsueo2QhgK
uftjpRB3\nVqqolQzVpU3t0XUANT3BlNhdy+noLbrLV+pd3XNCQU5kqwBVwJCvsjKYopTmv5+l\n97sYBmcnsqmDbstWo8NHqfRAcMPmBiAtI
niQ1ifBSwKCAQEAyr/2JvLPRUkDfqaU\nwYFVsb+HCunfKylujX9U3GRqn61ANXrEdhsu7Tr1OZc6zTUkS46c0jnr2YyaNM4T\njhvLZPE3tW
y0+hsS7IklpxSx0VO+iV3XOiIUZBidjhTlndxsAfQhixeKOaR1H59N\nHXN0MYyUEGTPsGvB/wwVtZfx9YO8yyjJLUr/zVAzrVay+eB2GkNZ5
WCzq648QVlr\nbzWcqvofG5JLoZgm+/EMiLhCq3PbMwvpef5HtPyPJzw4lLsDMiRas4c89Z5HfomQ\nukB6zn9W0CZncbMVMvX0Vxb8c5ugms
vSXtvmxujynDlBLJqMbU5ZHFlc8dASVHZp\n4zZfnQKCAQEApkeepTwf5hYBt60PALhj62EGm+HIqf6WtZStR/nAvu6WpX+udkB7\njUAFSe1
HWyY9d4/3IC0R26Gm9YtaYoq6UOQsxsR9Vm3QqKzD1qoFtEIaPxOsTerW\nsDrG1fDUlCfutLoZaWqbevlEKX4tUgTm76jY57ldX9JNvzqRiv
wbAZj/R/NbLRZR\n9hE1r6HYdwJe2O5afAwl2CxM18yZVMiAGgy6jaef3pmJmYsxh+tn0l7e+tjJFmHq\n9Gz99tylKxcsW49uRXPWVvtNiND
R99Vo3xSTFQPAiuOSSfUT5ewzhOLPLxvw4DtU\nteJ5SWBJ4tkrvLMonYefWPU31i8+Z79/SQKCAQAw1FWsRHQogOJy/qr+VpBrwoOa\nltTk
5+xpzNHxDHlZPreVSxxKxr98XZRp/LC3OK6DBOfNtLIUqrEflh26kEI/9/cn\nSwCiurKcu7dQsoTJDarcPT70M6WGuQ5trU2v9XGtWJ2pKPu
+2OVxlPXCKiCTp+lk\nJvr3uj/cmRI/wiB1tKDKug43HyvSzYVkMqMZsJ55EEzHeS6ahAGRvzRbO14uM0zW\n3bn8hZtjH+q/VWoPegWU+TJJ
5sGXu1fX9/kk2YrLY2hUaUWHMymxNfJwJj73o/aX\nHi33aG/qacIhS1AfSHqFaMDa3A/weJ5kx9Ufnp7Z1YCgEMbT1hzl4JYrHIKU\n-----
END RSA PRIVATE KEY-----'
b'-----BEGIN PUBLIC KEY-----\nMIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEA2/B4shpNAOEIHqCP1CA5\nsau1mK7P9+hSE
yIlMsHx8su9zPFPhABT3gWFPUD8kNEtx9v7nm4MmzVF4Y+e+ZBA\n222lMHQG8ja2SmdepJc6TmSQRjACLdGtK9+aVRST/gV5jHOhVUIwbBzA
VZFhxggF\nfL52P9hLHWvlu6YLF7NoY+mp9gDukOMb+WszQbyQZMKnAreWMKquWVBIDshbNXdm\njYaeNE6KRd4DR732a2qvslaIthRy8Bj84
+RWP3qsGBI4gSdxDBAL0PcN8aGfZyUb\nIZKeSh51Yx+2OcX9P/UWSkCZ9iTf5P9ZtAXw6LRqrVpLtT9EgY5qNGHtwZNoNqI1\nmW6oacJexz
e9Y9mXTshs7d3IWVVkNoUSz7F68A5e+kAk0NDQwgI5lNdzOkNhxjvj\nalfwuIA1Tpwlp54YLx909gpuRIRINs9ZxIq/LRgiWDqm60daHK8dK
tc0g0th5ntQ\n9yrn23LrbGfqagr6HK6lynh+58KaoRGOqMwlS0sdbS7tQ8VzV04qJjIUp3rlaICq\ngesE/mDCFnpIgVbwvd54uDbQQqmB
8t9u8VnmsMgOt+dRyLL2IWt1HfDhymgcnB\n09+cY2rCsuqYbJwdniNUdG4QRy/SAe9sZcz4ui/qz3AqLtOQwF1f9bX0sZTFcJ5j\nKa7i
QpYbA5yVHJJVI3H8CAwEAAQ==\n-----END PUBLIC KEY-----'
~/practical-crypto-course$

Lab13: Sending Secure Messages Over IP Networks

Create a Server Socket : receives, decrypts and reflects what the client type

The following Python code will allow you to create a server that is listening on port 8080. Once this code executes, it will launch a command window that states Waiting to receive message. . . .

```
#  server side who listen to udp port  = 8080
# Message Receiver
import hashlib
import random
import os
from socket import *
from cryptography.fernet import Fernet

key = Fernet.generate_key()
f = Fernet(key)
print ("The key is :", str(key, 'utf-8'))
host = ""
port = 8080
buf = 1024
addr = (host, port)
UDPSock = socket(AF_INET, SOCK_DGRAM)
UDPSock.bind(addr)
print ("Waiting to receive messages...")

```

Declare a function “decrypt” using the fernet instance as follows :

```
def decrypt(ciphertext):
    try:
        msg = f.decrypt(ciphertext)
    except:
        msg = ciphertext
    return msg
```

Use an infinite loop to listen and receive encrypted message from client and decrypting them using the function “decrypt” :

```
while True:
    (data, addr) = UDPSock.recvfrom(buf)
    h = hashlib.md5(data)
    plaintext = decrypt(data)
    msg = str(plaintext, 'utf-8')
    print ("Received message: " + msg)
    if msg == "exit":
        break
    if msg == 'newkey':
        key = Fernet.generate_key()
        f = Fernet(key)
        print ("The key is :", str(key, 'utf-8'))

```

when the user write ‘exit’ the UDP socket will close and exit the server program :

```
UDPSock.close()
os._exit(0)
```

Create a Client Socket : types, encrypts and sends encrypted message to server

The previous implementation will present you with a key. To get both the client and server communicating with each other, you will need to copy and paste the provided key to the client terminal.

```
#  clien side : type, encrypt and send ciphertext to server
# Message Sender
import os
from socket import *
from cryptography.fernet import Fernet

host = "127.0.0.1" # set to IP address of target computer
port = 8080
addr = (host, port)
UDPSock = socket(AF_INET, SOCK_DGRAM)
key = input("Enter the secret key: ")
f = Fernet(key)

```

Declare a function “encrypt” using the fernet instance as follows :

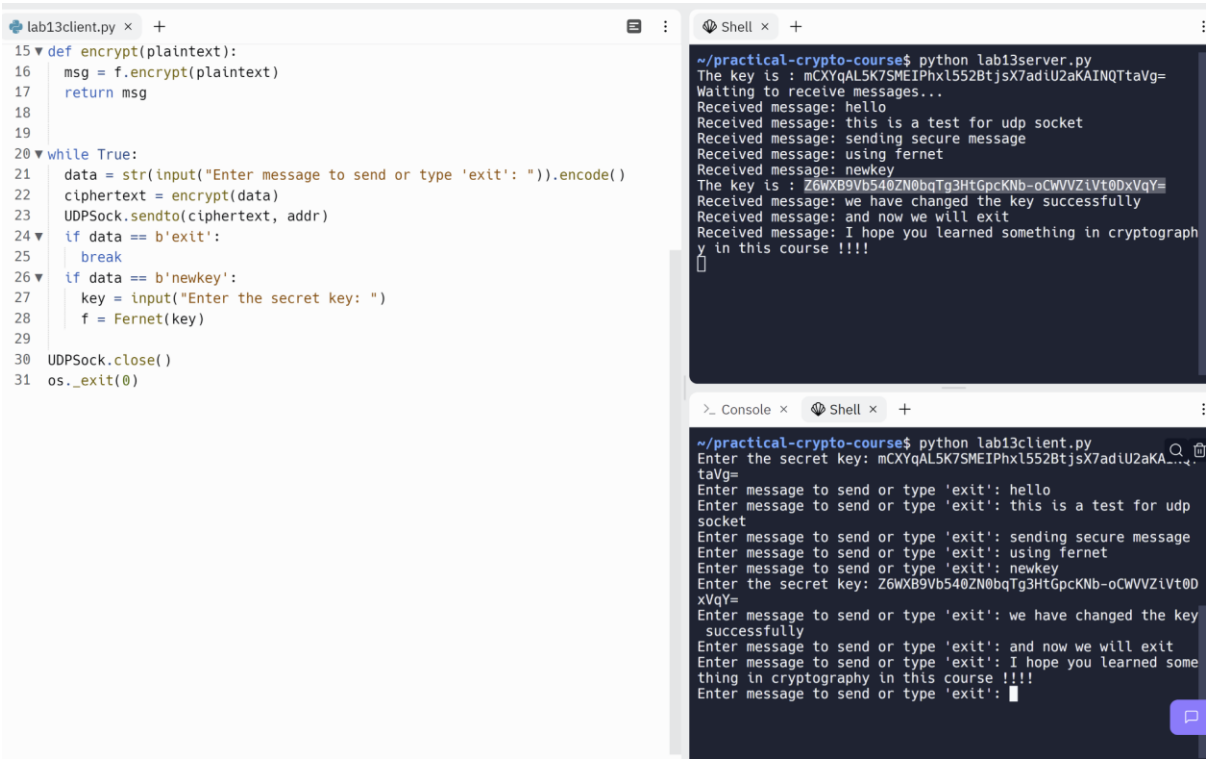
```
15 ▼ def encrypt(plaintext):
16       msg = f.encrypt(plaintext)
17       return msg
18
```

use an infinite loop asking the user to enter a message to be encrypted and sent via the UDP socket.

```
20 ▼ while True:
21       data = str(input("Enter message to send or type 'exit': ")).encode()
22       ciphertext = encrypt(data)
23       UDPSock.sendto(ciphertext, addr)
24 ▼     if data == b'exit':
25           break
26 ▼     if data == b'newkey':
27           key = input("Enter the secret key: ")
28           f = Fernet(key)
29
30   UDPSock.close()
31   os._exit(0)
```

close the UDP socket and exit the client program when the user type “exit”.

note that the user can ask to change the key if he types “newkey”.


lab13client.py
def encrypt(plaintext):
msg = f.encrypt(plaintext)
return msg
while True:
data = str(input("Enter message to send or type 'exit': ")).encode()
ciphertext = encrypt(data)
UDPSock.sendto(ciphertext, addr)
if data == b'exit':
break
if data == b'newkey':
key = input("Enter the secret key: ")
f = Fernet(key)
UDPSock.close()
os._exit(0)
Shell
~/practical-crypto-course$ python lab13server.py
The key is : mCXYqAL5K7SMEIPhxl552BtjsX7adiU2aKAINQTtaVg=
Waiting to receive messages...
Received message: hello
Received message: this is a test for udp socket
Received message: sending secure message
Received message: using fernet
Received message: newkey
The key is : Z6WXB9Vb540ZN0bqTg3HtGpcKNb-oCWVVZiVt0DxVqY=
Received message: we have changed the key successfully
Received message: and now we will exit
Received message: I hope you learned something in cryptography in this course !!!!
Console
Shell
~/practical-crypto-course$ python lab13client.py
Enter the secret key: mCXYqAL5K7SMEIPhxl552BtjsX7adiU2aKAINQTtaVg=
Enter message to send or type 'exit': hello
Enter message to send or type 'exit': this is a test for udp socket
Enter message to send or type 'exit': sending secure message
Enter message to send or type 'exit': using fernet
Enter message to send or type 'exit': newkey
Enter the secret key: Z6WXB9Vb540ZN0bqTg3HtGpcKNb-oCWVVZiVt0DxVqY=
Enter message to send or type 'exit': we have changed the key successfully
Enter message to send or type 'exit': and now we will exit
Enter message to send or type 'exit': I hope you learned something in cryptography in this course !!!!
Enter message to send or type 'exit':