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

# LAB Manual

## CSSY2201 : Introduction to Cryptography

UTAS
Sultanate of Oman

February 2022

**LAB1 : 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 two entered passwords and will output a successful message if the typed password matches. For that we need to use the module "getpass". Information about each user, his password 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). This function will ask the user twice for the password, it will then compare between them and will return the password if there is a match. If there is no match it will keep asking the user for the correct password.
- store_info(username, password) : to write the username + password 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
def ask_for_username():
    print("enter the user name would you like to use...")
    username = ……
    return ……
```

```
def ask_for_password():
   while True:
    print("What password would you like to create?")
     p=getpass.getpass()
     print(p)
    print("Please enter the password again...")
    q=……
    print(q)
     if p == ……:
      print("Your password is matching...")
      print("Your username and password are stored in testhash.txt file")
      return q
    else:
      print("Your password do not match. Please retry...")
```

```
def store_info(username, password):

#password_file =open_pass_file()

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

    password_file.write(username  + " | " + password + "\n" + "\n")
```

And the main program :

```
username = ……..

password = ……..

store_info(……., ……., ……)
```

**LAB2 : 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).

**LAB3 : DES**

**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.

```
19 cipher_text = des1.encrypt(text)
20 print(cipher_text)
21 cipher_textt=base64.b64encode(cipher_text).decode()
22 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.

```
24 decipher_text=des2.decrypt(cipher_text)
25 print(decipher_text)
26 decipher_text=decipher_text.decode("utf-8")
27 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.

**<u>Part B : 3DES</u>**

- 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** :

```
 9
10 from Crypto.Cipher import DES3
11 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
```

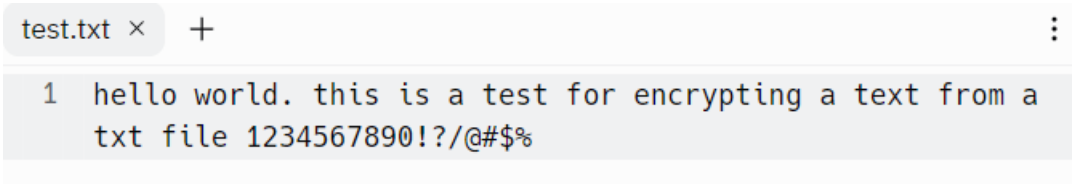
**Step 3** : declare the decryption function as follows :
- it should have 4 arguments: the secret key, the input file name (ciphertext file), the output filename (the recovered plaintext file), and the chunksize which should be the same as the transmitter operated.
- open the ciphertext file to be read
- 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

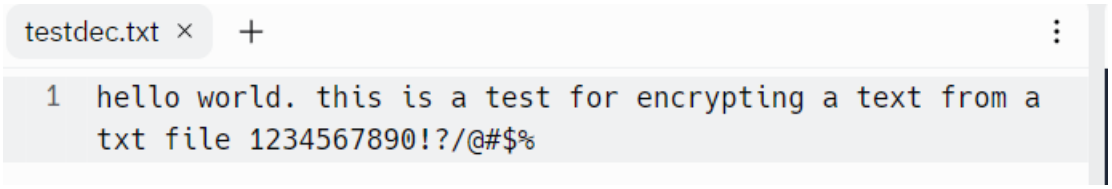



This is the content of the ciphertext file  test.txt.enc :


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+PYqXzTYIL50eqdN3AVRg4TS3okVYyGhGREmdsb
2c7UEmQ6dsBtu/3IPdDUb4kjVdbFjnW467W80NIDOirOiGsqgXsEZd

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




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

**LAB4 : AES**

**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 also print 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 chunk 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=None, chunksize=16*1024):

    with open(in_filename, 'rb') as infile:
        iv = infile.read(16)
        decryptor = AES.new(key, AES.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 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 encryption 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
```

**LAB5 : RSA**

**Part A: 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 :

```
from Crypto.Util.number import *
from Crypto import Random
import Crypto
import gmpy2
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.

**PartB : use "rsa" module**

lab5partB.py

```
import rsa

(publickey,privatekey) = rsa.newkeys(1024) # RSA KEY GENRATION with key sizes- 102420484096
message = input("Enter a message to encrypt with RSA-") # input

#Encryption with Public Key
#ciphertext = rsa.encrypt(message,publickey)
ciphertext = rsa.encrypt(message.encode('ascii'),publickey)
print("Encrypted output is-" ,ciphertext.hex())

#Decryption with private Key
#plaintext = rsa.decrypt(ciphertext,privatekey)
plaintext = rsa.decrypt(ciphertext,privatekey).decode('ascii')
print("Plain text after decryption is- ",plaintext)

```

**Part C: 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

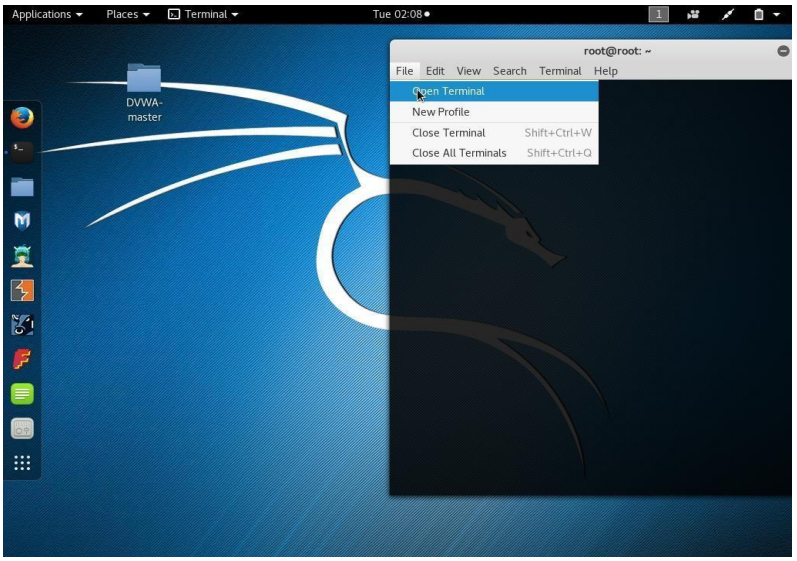
coefficient:
00:9d:80:de:5d:29:4c:23:70:b8:66:ee:51:5c:3c:
2a:32:7a:cc:7a:0b:ae:8d:e9:4e:bb:a1:80:36:6f:
7e:3b:ea:fb:29:85:99:0d:bf:a3:a9:b4:50:b8:89:
d9:7b:14:a9:59:e9:77:36:13:5a:e3:8a:3a:46:0c:
c4:db:f7:18:3a

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

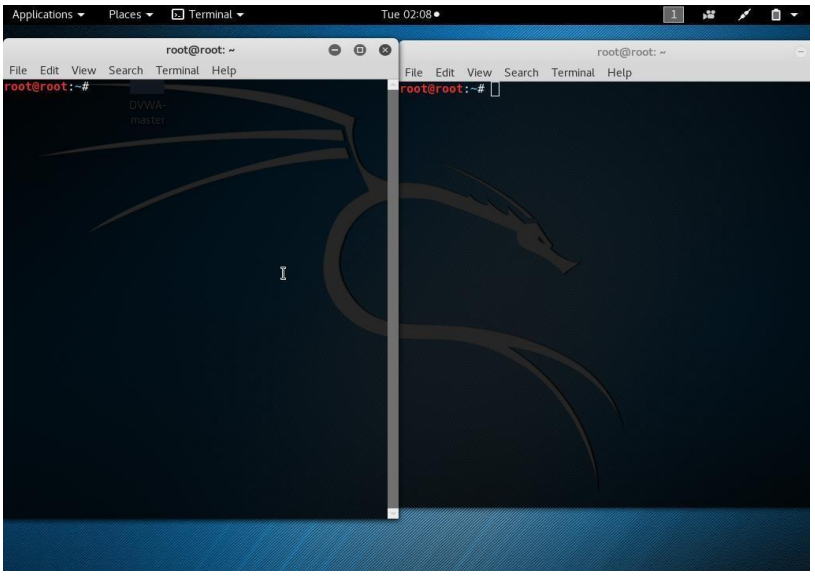
Assume there is a communication between two users Alice and Bob. It will require two terminals to work as two computers to simulate this communication.

**Generating public key and private key**

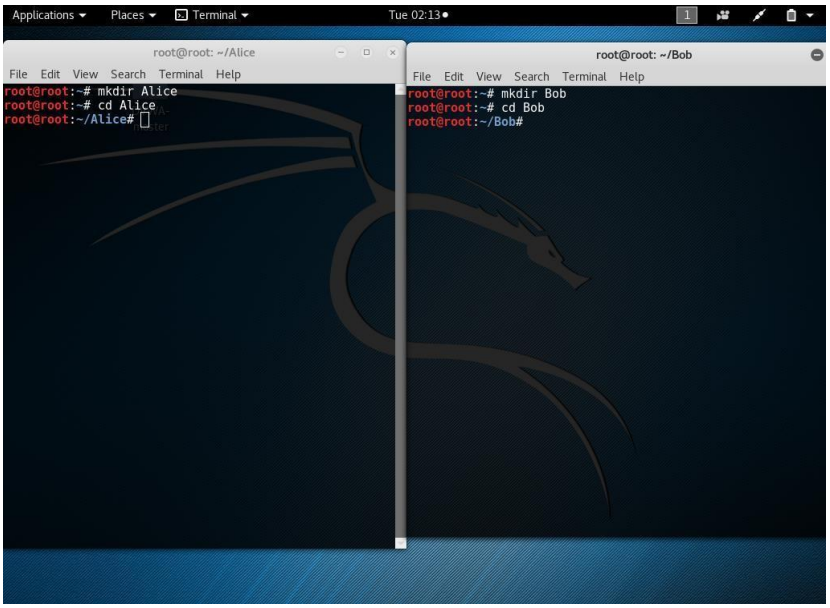
- Use two terminal and make directory for each user




- Open two terminal



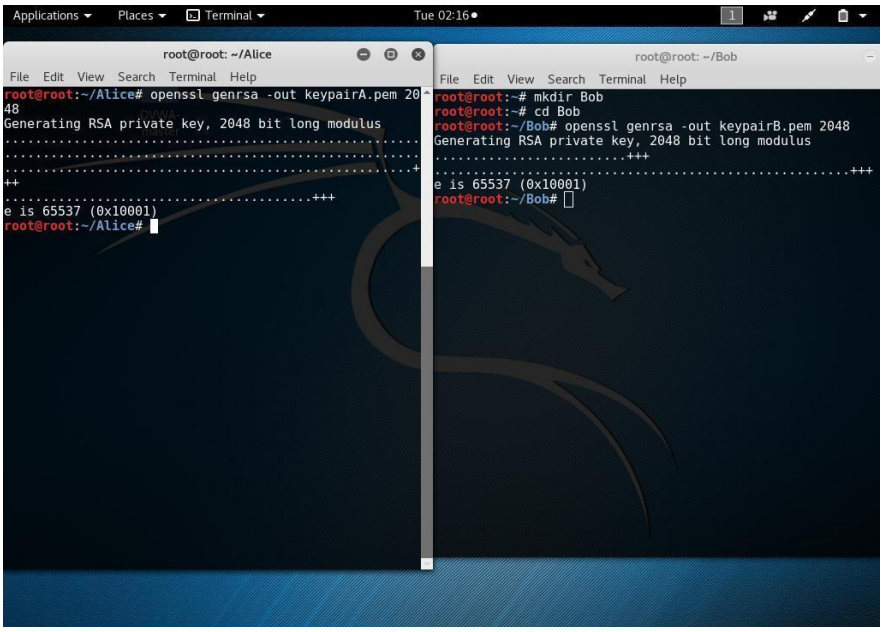

Make directory for both users



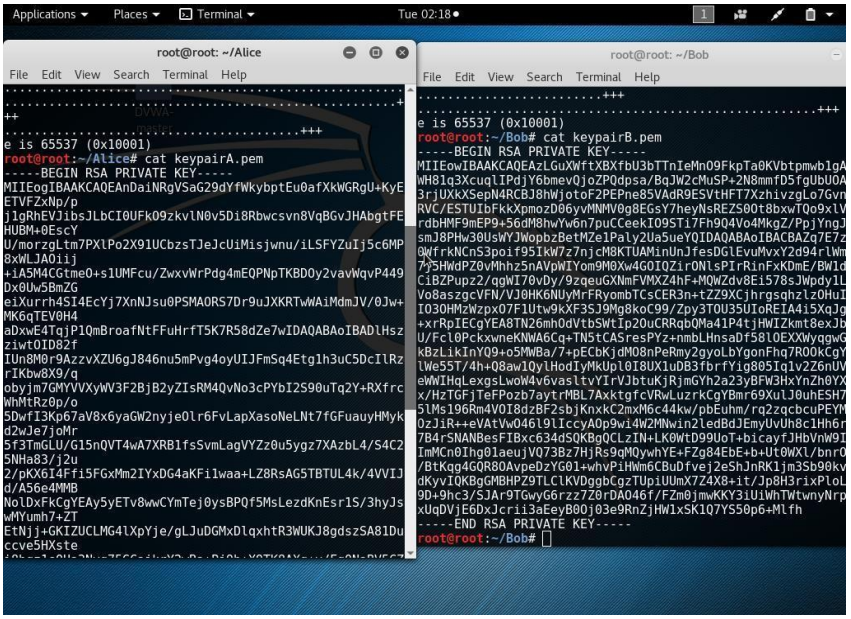

- Generate public and private key for each users (Alice and Bob)

  **openssl genrsa -out keypairA.pem**

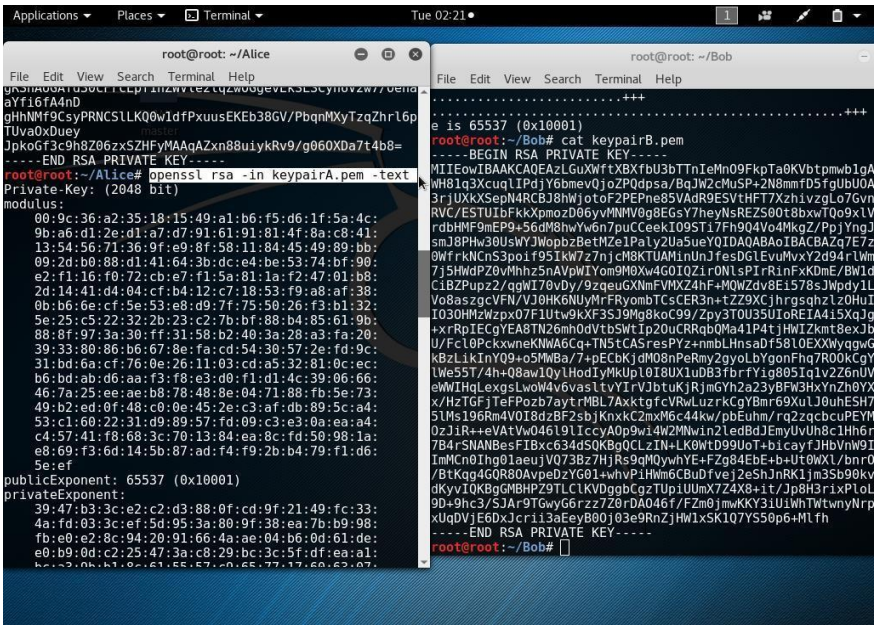
  **openssl genrsa -out keypairB.pem**




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

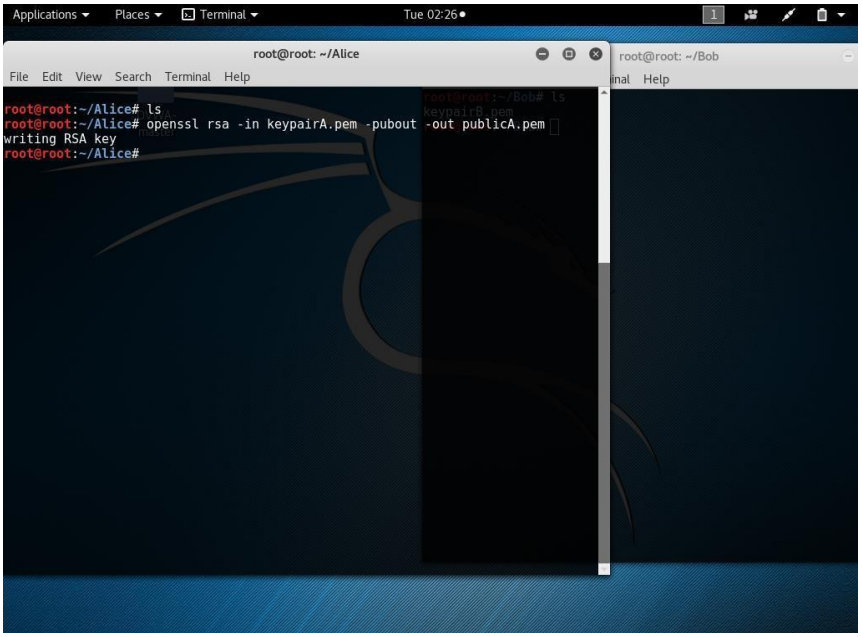



- Or **openssl rsa –in keypairA.pem -text** to view all the numbers in this key.

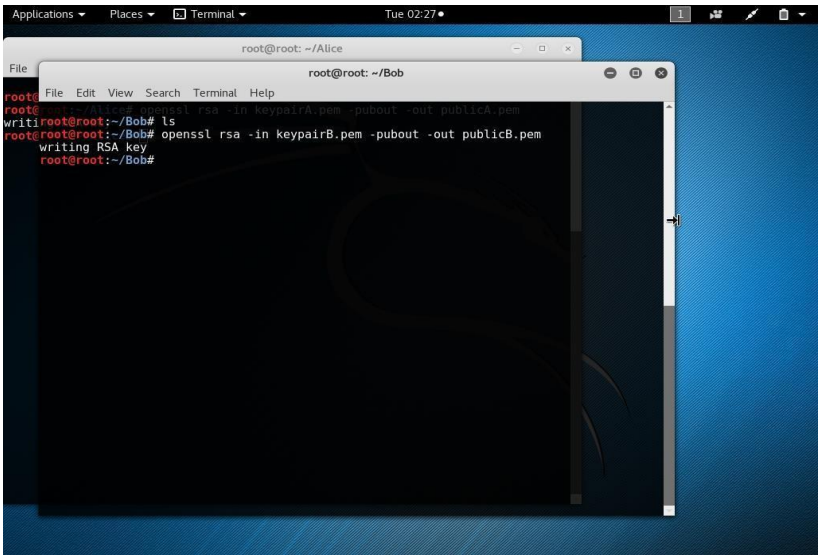



To generate the public key to send it to other user

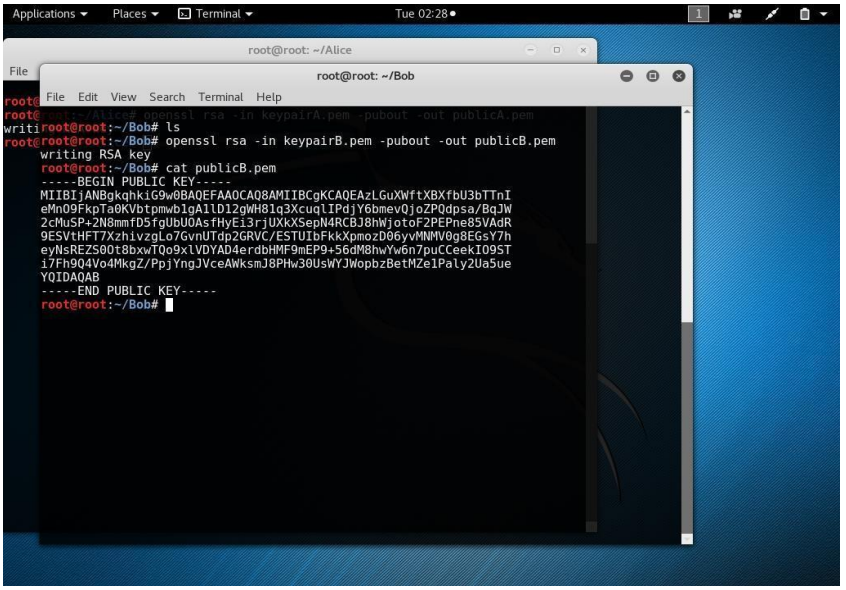
**openssl rsa -in keypairA.pem -pubout -out publicA.pem**




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

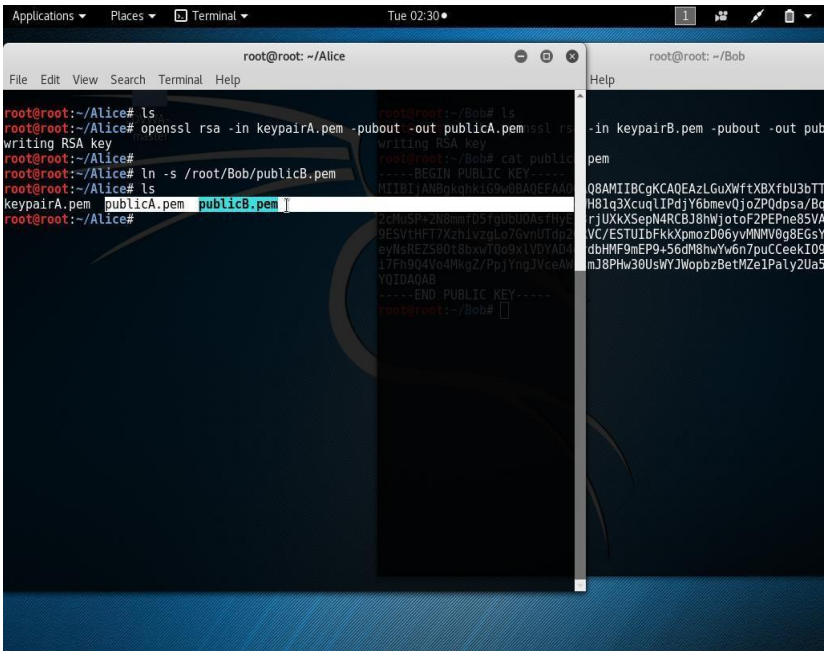



- To view the public key for Alice use **cat publicA.pem**
- To view the public key for Bob use **cat publicB.pem**



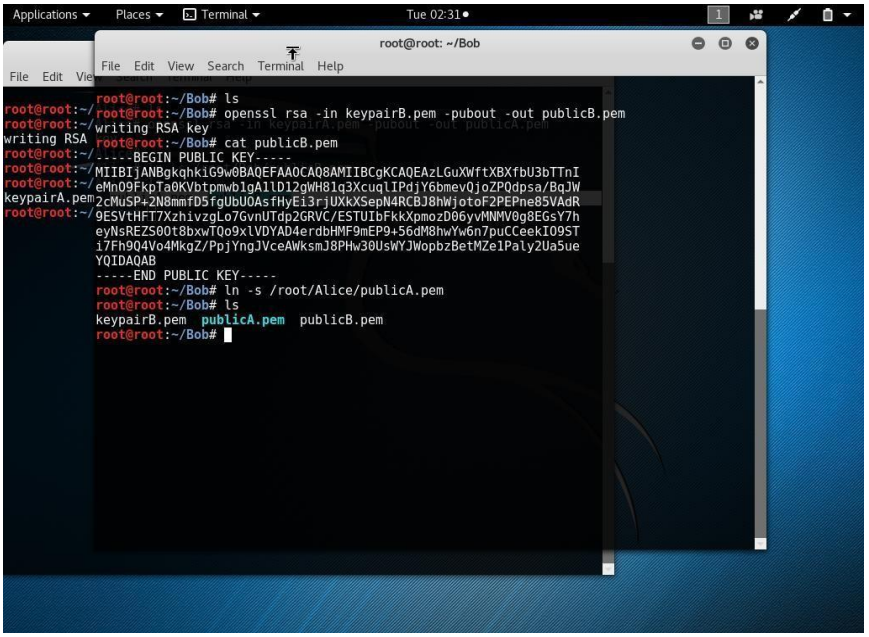

- Now Alice and Bob have to share the public key to encrypt and decrypt the files In Alice terminal copy Bob public key using the following command

  **ln -s /root/Bob/publicB.pem**



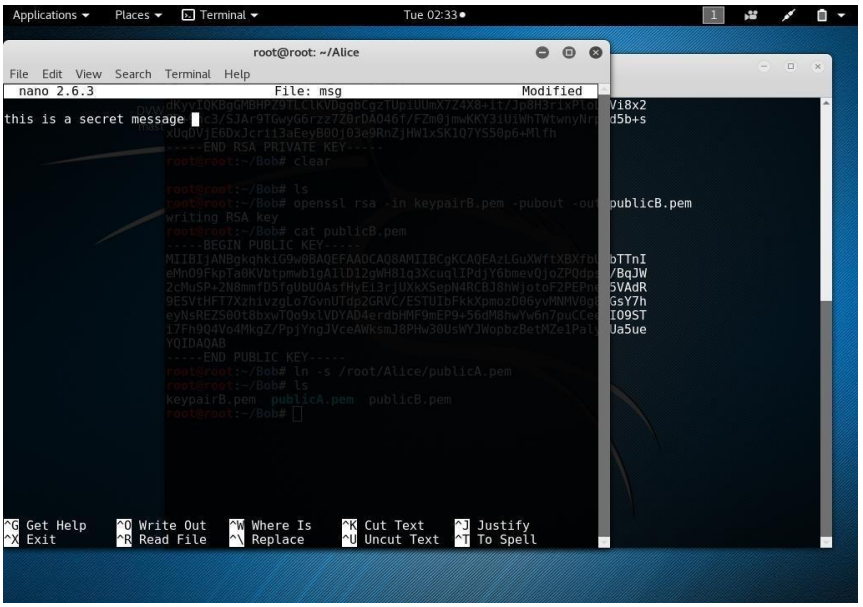

In Bob terminal copy Alice public key using the following command

**ln -s /root/Alice/publicA.pem**

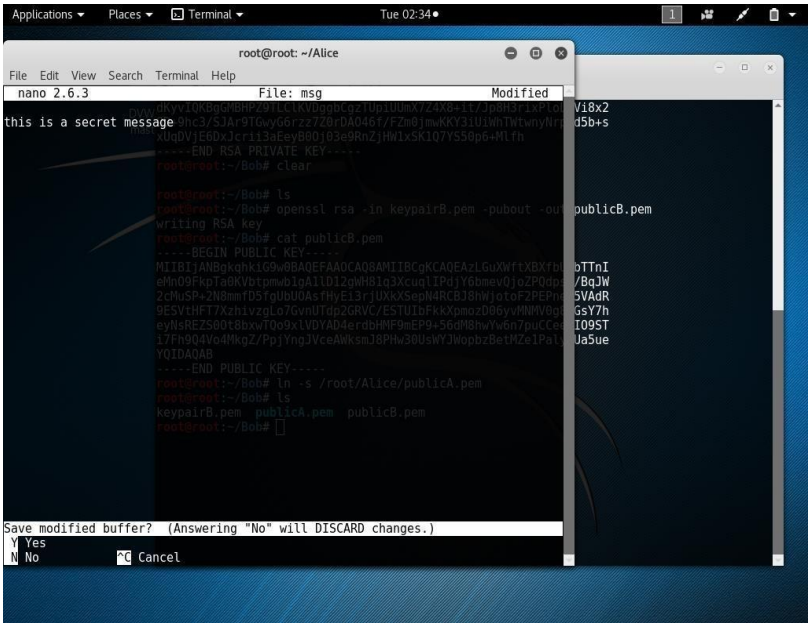


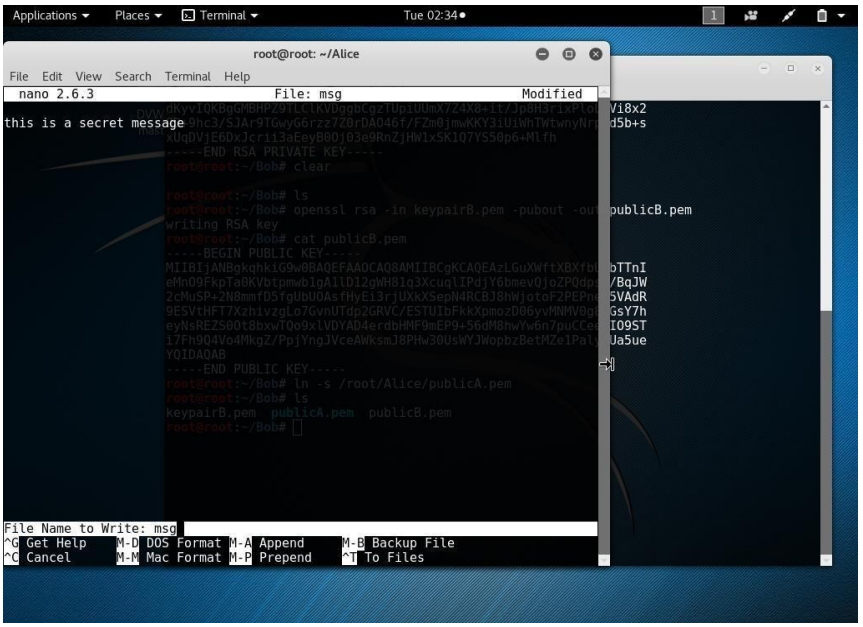

## Encryption and decryption using public key and private key

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

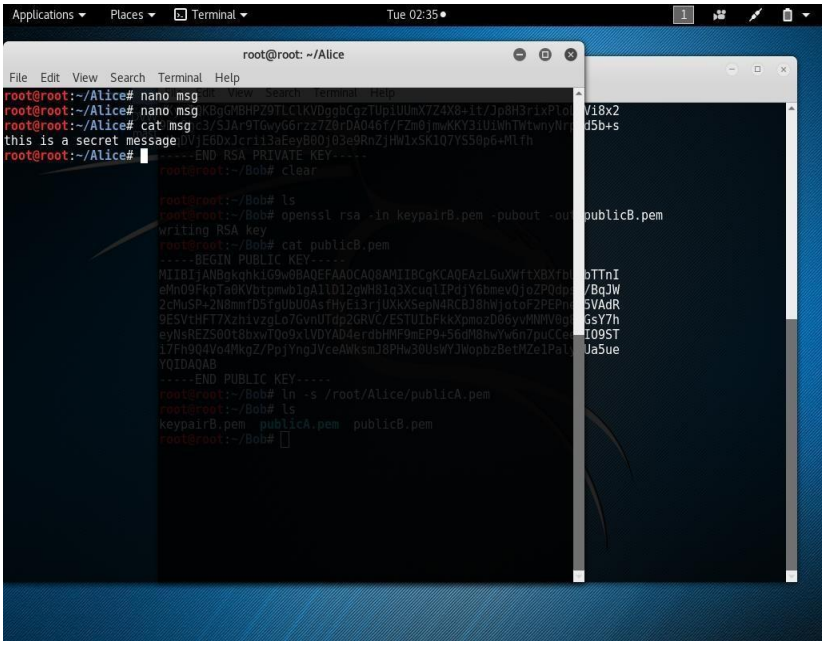



Save the file

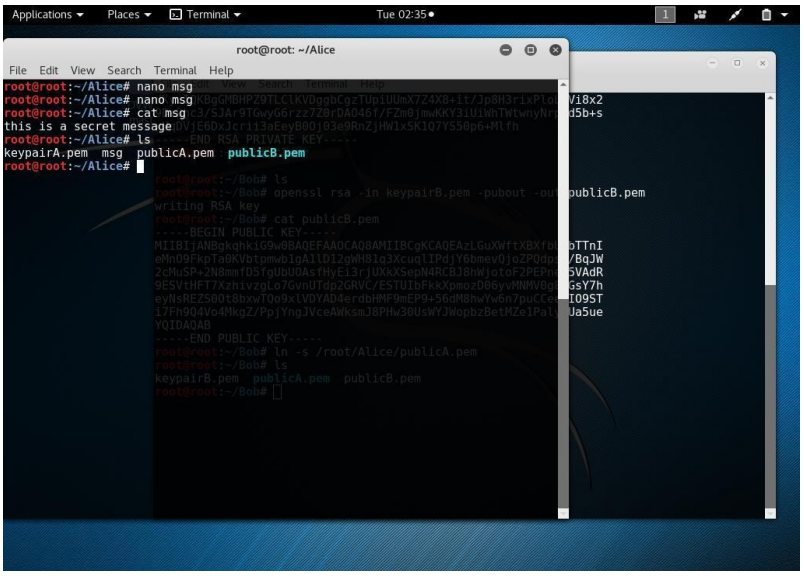





To display message in terminal use command **cat msg**

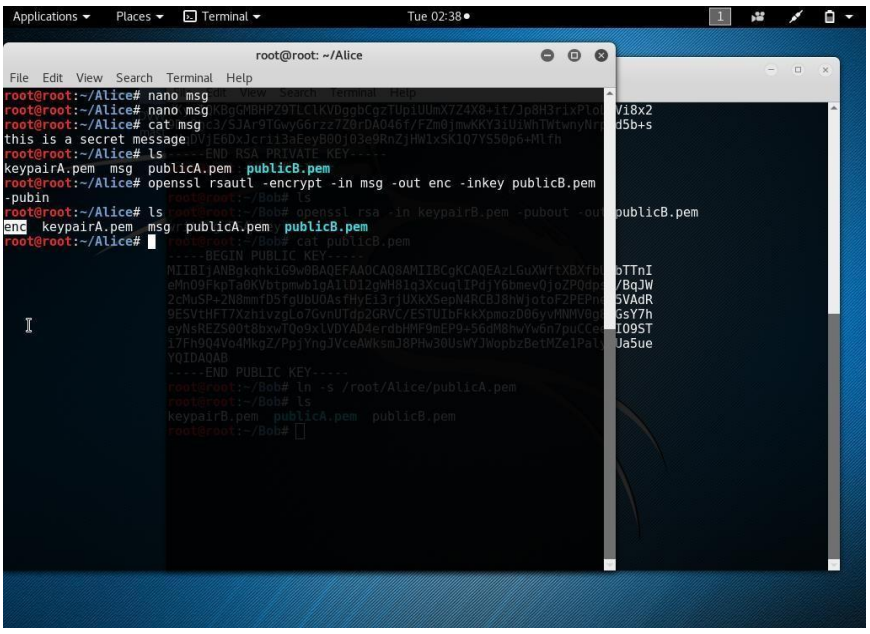



- To display public and private keys use **ls**

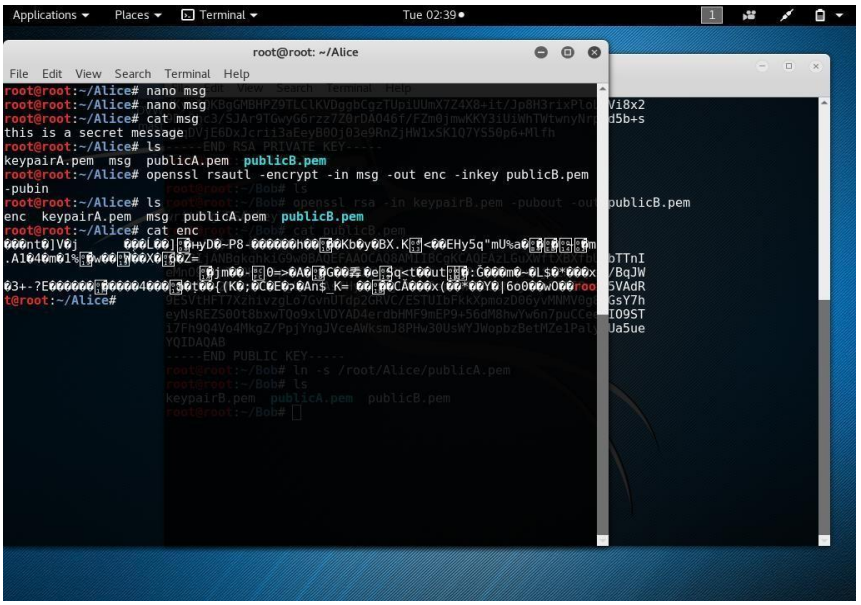



Alice will encrypt the message using the public key of Bob using this command

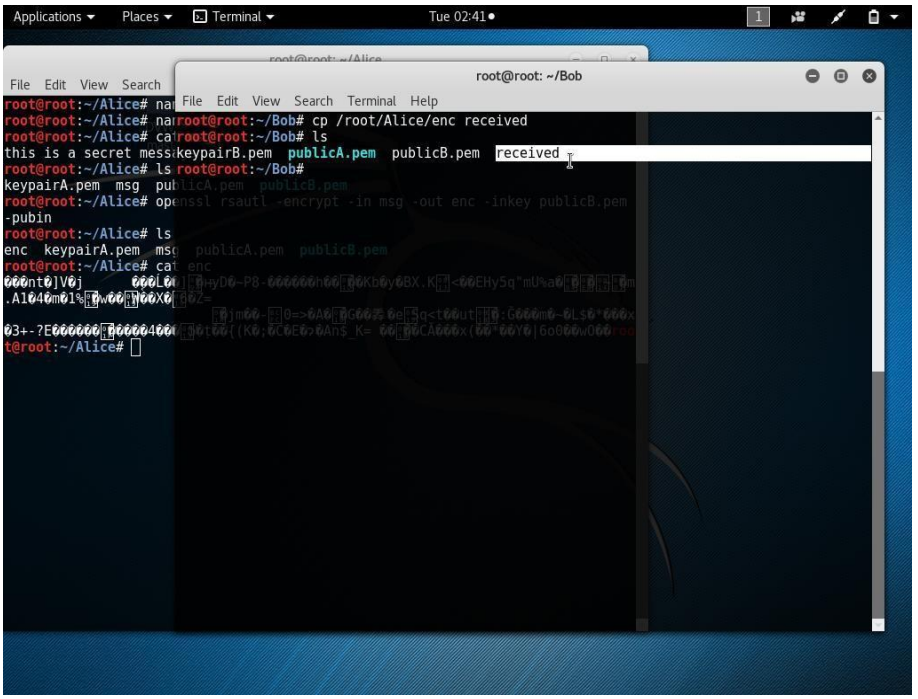
- **openssl rsautl -encrypt -in [textfile] -out enc -inkey publicB.pem pubin**
- The output file after encryption is **enc**



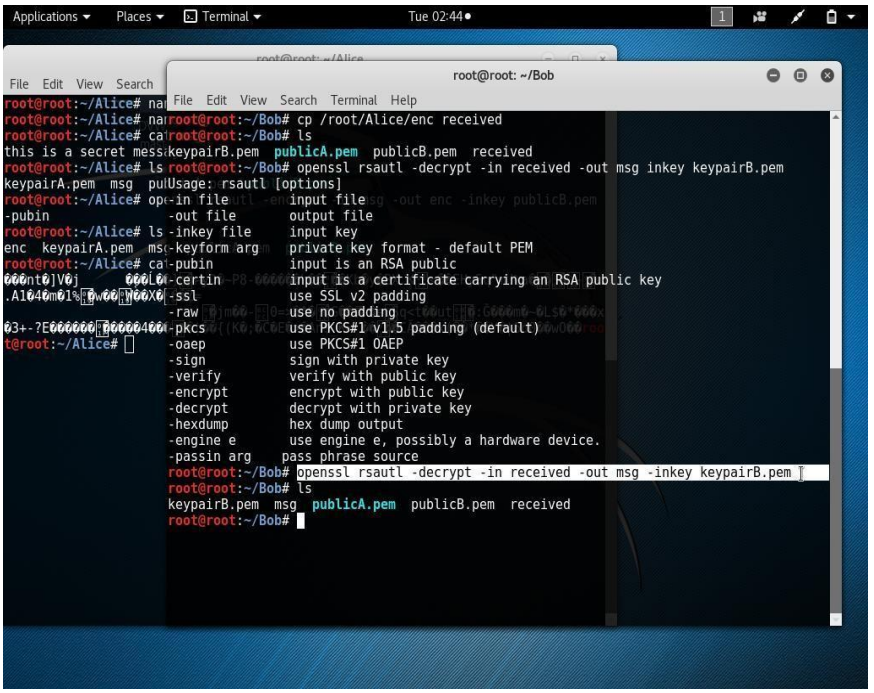

- To view the encrypted messages **cat enc**



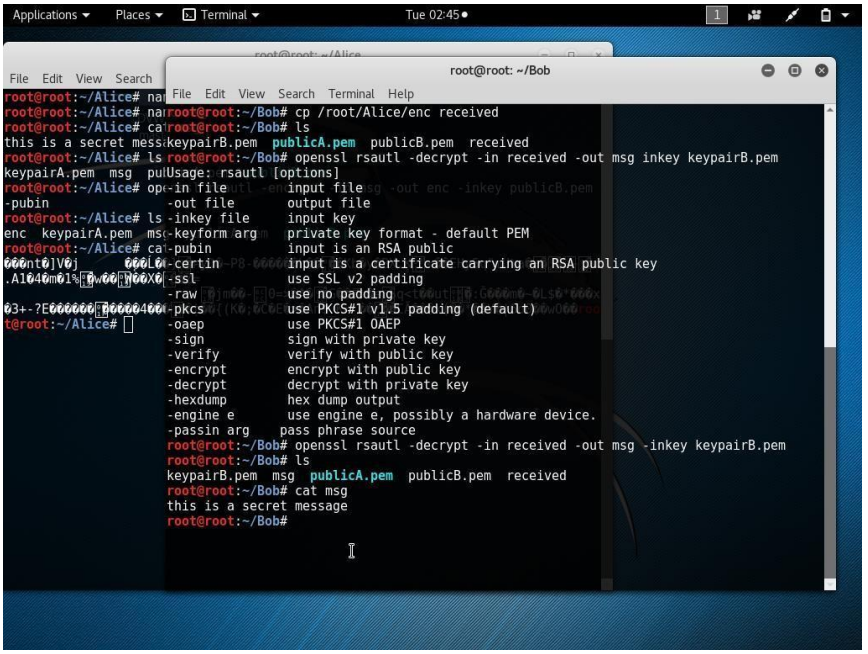

In Bob directory: copy the message from Alice directory **cp /root/Alice/enc received**



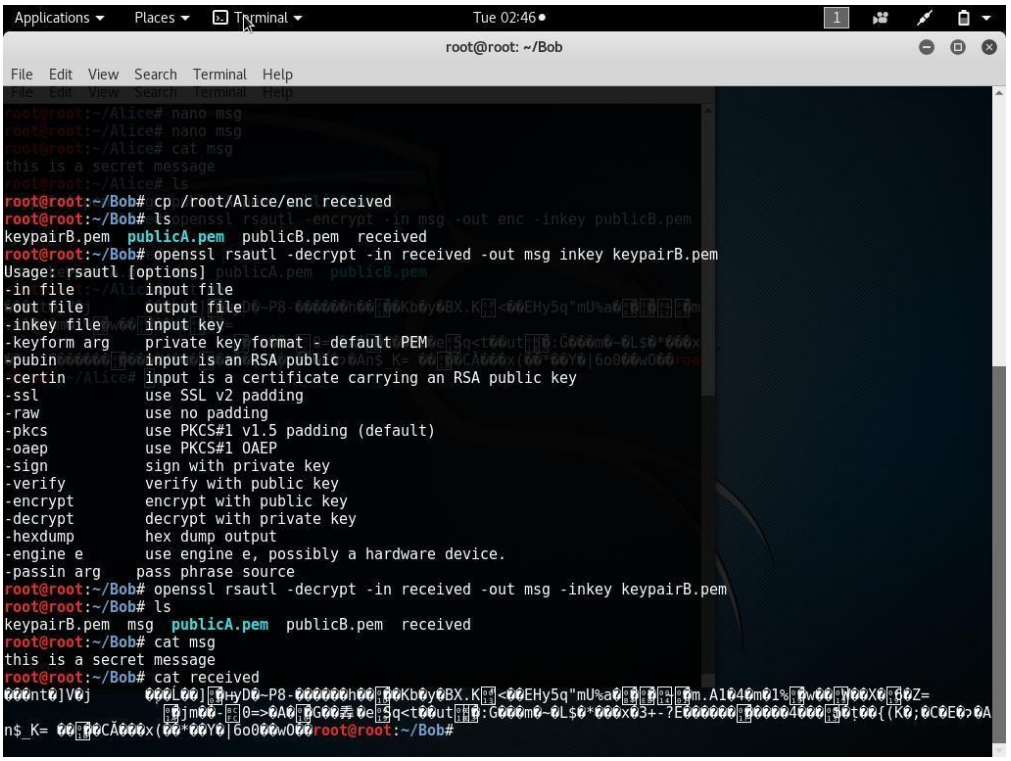

- To decrypt the message using the private key of Bob **openssl rsautl -decrypt -in received -out [textfile] -inkey keypairB.pem**




To view the messages **cat msg**




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




**LAB 6 : Hash Functions, HMAC and PRNG**

**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)
20
```

**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 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:

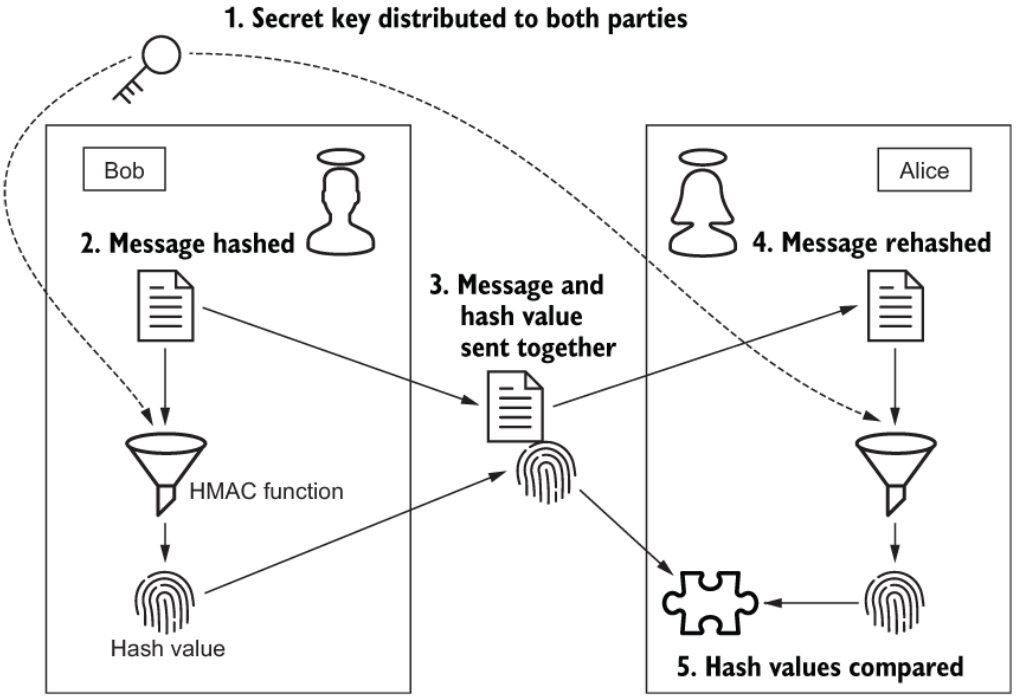
Alice and Bob share a secret key.

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 C: Random numbers generation**

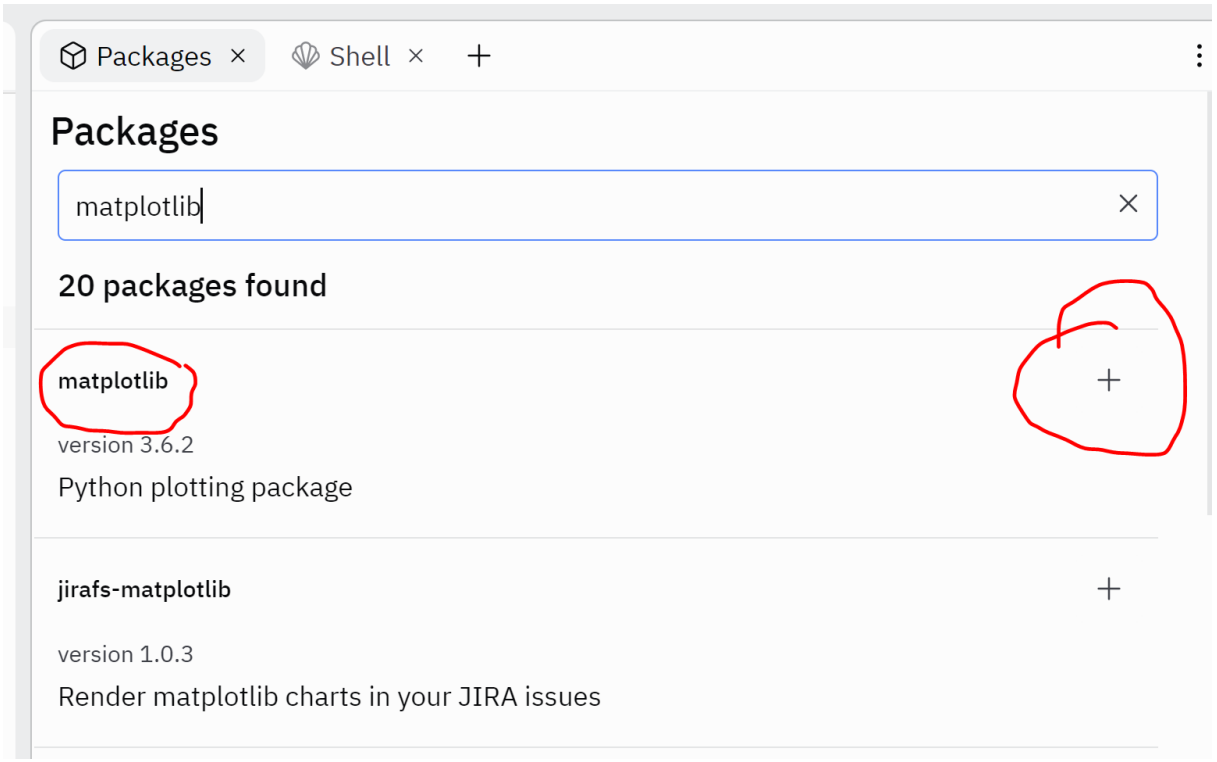
- Manipulate random number generation using python.

```
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 7 : Image representation, image cryptography and steganography**

**PartA: Image representation**

This lab requires you to install the “matplotlib” and “opencv” modules.




You need to install “pillow” module also in the same way if it is not already installed.

You need to consider any PNG true-color image in order to analyze it.

Load the necessary packages (pillow, numpy and matplotlib) :

```
1  from PIL import Image
2  import numpy as np
3  from matplotlib import pyplot as plt
4
```

open a png image and load it into a matrix, then calculate the dimensions of the matrix :

```
5  im = Image.open('nebula.png')
6  im_matrix = np.array(im)
7  print(im_matrix.shape)
8
```

What did you get ? : ……..

Access the pixel value in position [100,150] :

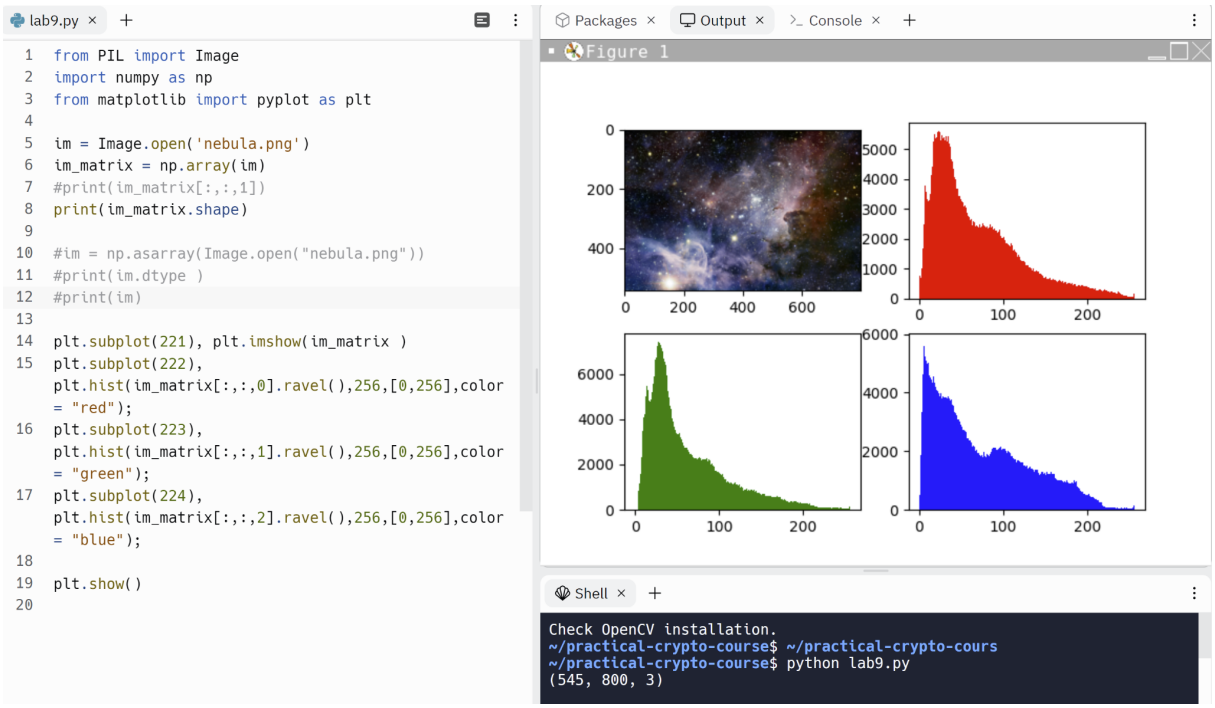
```
14  print(im_matrix[100,150])
15
```

what did you get ? : .....

**Histogram plot:** now you will show the loaded image and plot its corresponding histograms :

```
plt.subplot(221), plt.imshow(im_matrix )
plt.subplot(222), plt.hist(im_matrix[:,:,0].ravel(),256,[0,256],color = "red");
plt.subplot(223), plt.hist(im_matrix[:,:,1].ravel(),256,[0,256],color = "green");
plt.subplot(224), plt.hist(im_matrix[:,:,2].ravel(),256,[0,256],color = "blue");

plt.show()
```

This will create a plot formed by 4 subplots 2*2 where the first one is to display the image, the second to display the red component of the image, the third for the green component and the 4th for the blue component of the image.




**PartB : Random image representation**

To generate a random image we can use randint function in np.random module :

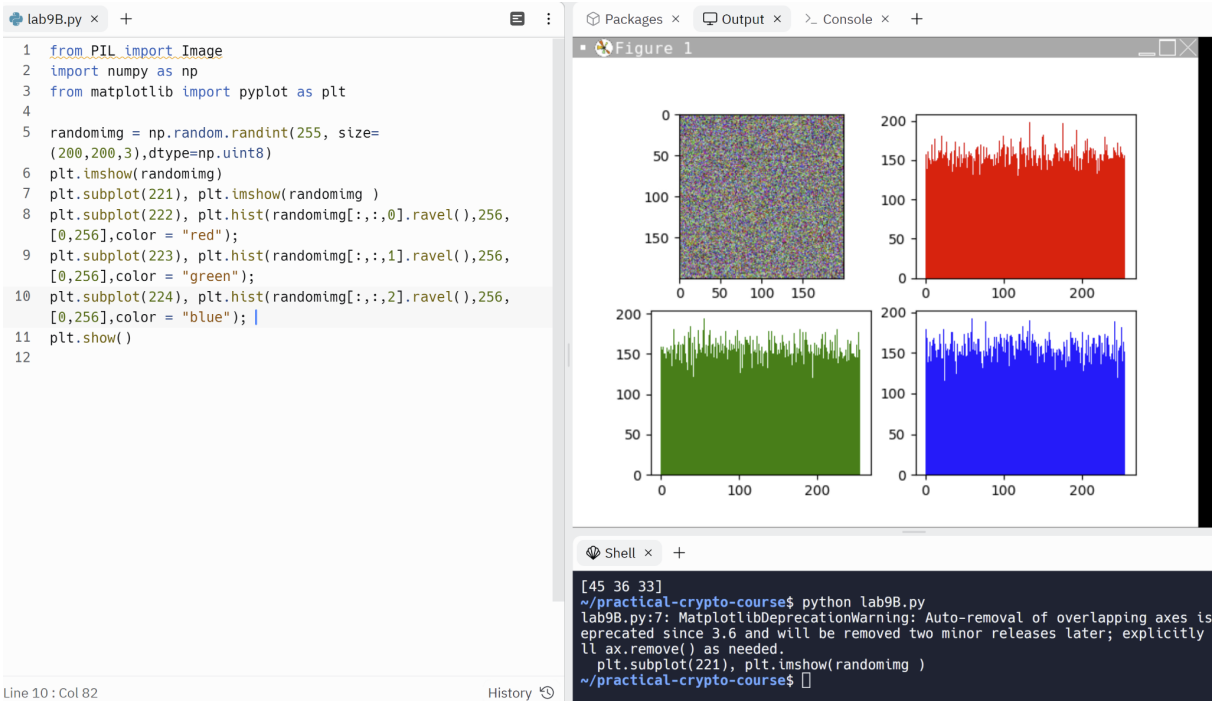
fro example, generate a true-color random image of size (200 * 200) :

randomimg = np.random.randint(255, size=(200,200,3),dtype=np.uint8)

Then use the code in partA to generate the histogram of the random image :

```
randomimg = np.random.randint(255, size=(200,200,3),dtype=np.uint8)
plt.imshow(randomimg)
plt.subplot(221), plt.imshow(randomimg )
plt.subplot(222), plt.hist(randomimg[:,:,0].ravel(),256,[0,256],color = "red");
plt.subplot(223), plt.hist(randomimg[:,:,1].ravel(),256,[0,256],color = "green");
plt.subplot(224), plt.hist(randomimg[:,:,2].ravel(),256,[0,256],color = "blue");
plt.show()
```

You will get something like this :

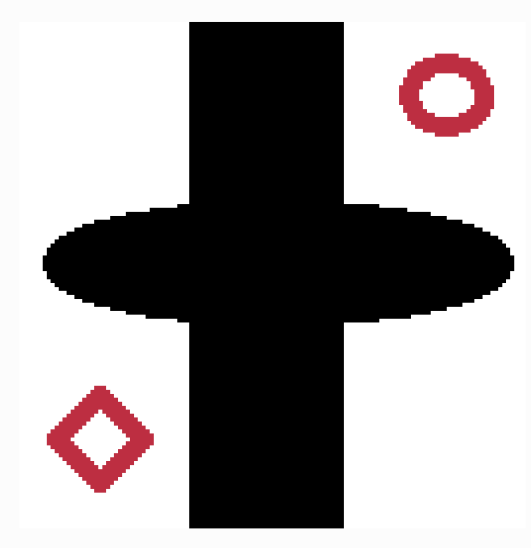



Observe the uniform distribution of the histograms.

## Part C : 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

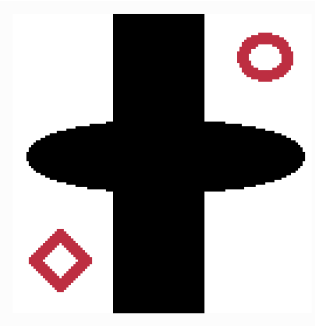


Import Crypto and initialize the encryption machine with a key of size 128 bits = 16 (bytes)*8. Use ECB mode and read the image blocks and encrypt them with AES-ECB. Consider the fact that AES use plaintext blocks of 16 bytes (128 bits) length. So we need to pad the plaintext image blocks to be multiple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

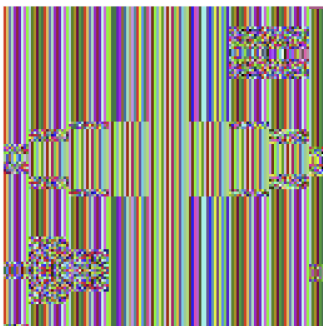
```
from Crypto.Cipher import AES

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 :

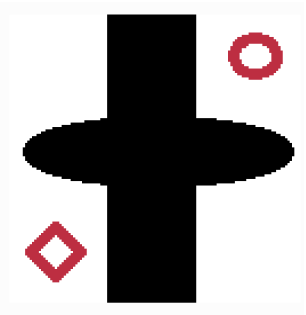
```
# decrypt using the reverse process
with open("e_plane.bmp", "rb") as f:
    byteblock = f.read()

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:

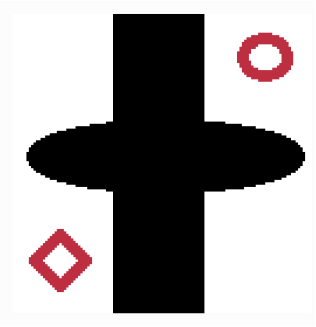
change the above code in ECB to incorporate the following change in the encryption side :

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

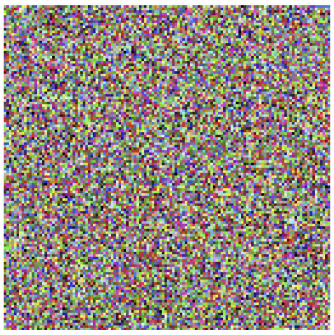
and the following change in the decryption side :

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

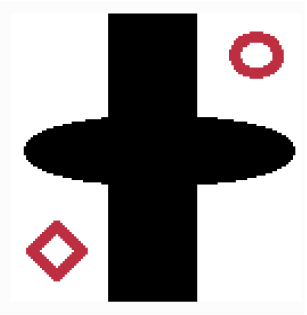
And this should be the result of encrypting a highly redundant image with CBC mode :



original



encrypted



decrypted

**Part D: 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

```
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)
```

## LAB 8 : Digital Signature

**PartA: 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

**$ openssl x509 -in wiki.crt -text -noout | less**

wiki.crt

```
-----BEGIN CERTIFICATE-----
MIIH0TCCBrmgAwIBAgISBAeco9H3wMFwcSUcawN/EFgMMA0GCSqGSIb3DQEBCwUA
MDIxCzAJBgNVBAYTAlVTMRYwFAYDVQQKEw1MZXQncyBFbmNyeXB0MQswCQYDVQQD
EwJSMzAeFw0yMjA5MDgwNTI1MzNaFw0yMjEyMDcwNTI1MzJaMBoxGDAWBgNVBAMM
Dyoud2lraXBlZGlhLm9yZzCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEB
AMOyBoX7NKn0DtYl0ePaAuz7VReZ52ALfTJ6D+wUpy1NhWX0xkz+Pkh6VxGmW9cQ
hrS1IrvRARKOmg7fvIvPYmGb13J0i6999c+UqqaS/Jl1IoCWPnOHETLBEtIDHnLB
dq9NHjCqsCnzRMuhlxZAsMfnQoxqj80srxMDwzexmusXP6yEsb3KCIEksPHapqM8
fy8ZkIPY+6JTEy59s11Oi5VHaCJRAQu6/LDrJ2s6n9G9A68OplKVHOpwRNQDUeWR
pjOP4KA+RqFhAjU64VSqdr0PsjS22PJwwzt0nFU72iXOLwT26JXSadCXs0TCN8+S
2v0zUpw8ikszPMhSo/tvAT0CAwEAAaOCBPcwggTzMA4GA1UdDwEB/wQEAwIFoDAd
BgNVHSUEFjAUBggrBgEFBQcDAQYIKwYBBQUHAwIwDAYDVR0TAQH/BAIwADAdBgNV
HQ4EFgQUDatDiY7xIfwLMNOPFDlP4eOeS/EwHwYDVR0jBBgwFoAUFC6zF7dYVsuu
UAlA5h+vnYsUwsYwVQYIKwYBBQUHAQEESTBHMCEGCCsGAQUFBzABhhVodHRwOi8v
cjMuby5sZW5jci5vcmcwIgYIKwYBBQUHMAKGFmh0dHA6Ly9yMy5pLmxlbmNyLm9y
Zy8wggLFBgNVHREEggK8MIICuIIRKi5tLm1lZGlhd2lraS5vcmeCESoubS53aWtp
Ym9va3Mub3JnghAqLm0ud2lraWRhdGEub3JnghEqLm0ud2lraW1lZGlhLm9yZ4IQ
Ki5tLndpa2luZXdzLm9yZ4IRKi5tLndpa2lwZWRpYS5vcmeCESoubS53aWtpcXVv
dGUub3JnghIqLm0ud2lraXNvdXJjZS5vcmeCEyoubS53aWtpdmVyc2l0eS5vcmeC
EioubS53aWtpdm95YWdlLm9yZ4ISKi5tLndpa3Rpb25hcnkub3Jngg8qLm1lZGlh
d2lraS5vcmeCFioucGxhbmV0Lndpa2ltZWRpYS5vcmeCDyoud2lraWJvb2tzLm9y
Z4IOKi53aWtpZGF0YS5vcmeCDyoud2lraW1lZGlhLm9yZ4IZKi53aWtpbWVkaWFm
b3VuZGF0aW9uLm9yZ4IOKi53aWtpbmV3cy5vcmeCDyoud2lraXBlZGlhLm9yZ4IP
Ki53aWtpcXVvdGUub3JnghAqLndpa2lzb3VyY2Uub3JnghEqLndpa2l2ZXJzaXR5
Lm9yZ4IQKi53aWtpdm95YWdlLm9yZ4IQKi53aWt0aW9uYXJ5Lm9yZ4IUKi53bWZ1
c2VyY29udGVudC5vcmeCDW1lZGlhd2lraS5vcmeCBncud2lraYINd2lraWJvb2tz
Lm9yZ4IMd2lraWRhdGEub3Jngg13aWtpbWVkaWEub3Jnghd3aWtpbWVkaWFmb3Vu
ZGF0aW9uLm9yZ4IMd2lraW5ld3Mub3Jngg13aWtpcGVkaWEub3Jngg13aWtpcXVv
dGUub3Jngg53aWtpc291cmNlLm9yZ4IPd2lraXZlcnNpdHkub3Jngg53aWtpdm95
```

Certificate:

```
    Data:
        Version: 3 (0x2)
        Serial Number:
            04:07:9c:a3:d1:f7:c0:c1:70:71:25:1c:6b:03:7f:10:58:0c
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, O = Let's Encrypt, CN = R3
        Validity
            Not Before: Sep  8 05:25:33 2022 GMT
            Not After : Dec  7 05:25:32 2022 GMT
        Subject: CN = *.wikipedia.org
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (2048 bit)
                Modulus:
                    00:c3:b2:06:85:fb:34:a9:f4:0e:d6:25:d1:e3:da:
                    02:ec:fb:55:17:99:e7:60:0b:7d:32:7a:0f:ec:14:
                    a7:2d:4d:85:65:f4:c6:4c:fe:3e:48:7a:57:11:a6:
                    5b:d7:10:86:b4:b5:22:bb:d1:01:12:8e:9a:0e:df:
                    bc:8b:cf:62:61:9b:d7:72:74:8b:af:7d:f5:cf:94:
                    aa:a6:92:fc:99:75:22:80:96:3e:73:87:11:32:c1:
                    12:d2:03:1e:72:c1:76:af:4d:1e:30:aa:b0:29:f3:
                    44:cb:a1:97:16:40:b0:c7:e7:42:8c:6a:8f:cd:2c:
                    af:13:03:c3:37:b1:9a:eb:17:3f:ac:84:b1:bd:ca:
                    08:81:24:b0:f1:da:a6:a3:3c:7f:2f:19:90:83:d8:
                    fb:a2:53:13:2e:7d:b3:5d:4e:8b:95:47:68:22:51:
                    01:0b:ba:fc:b0:eb:27:6b:3a:9f:d1:bd:03:af:0e:
                    a6:52:95:1c:ea:70:44:d4:03:51:e5:91:a6:33:8f:
                    e0:a0:3e:46:a1:61:02:35:3a:e1:54:aa:76:bd:0f:
                    b2:34:b6:d8:f2:70:c3:3b:74:9c:55:3b:da:25:ce:
                    2f:04:f6:e8:95:d2:69:d0:97:b3:44:c2:37:cf:92:
                    da:fd:33:52:9c:3c:8a:4b:33:3c:c8:52:a3:fb:6f:
                    01:3d
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment
```

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:

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 : ……

**PartB :**

Use "rsa" module to generate RSA keys, Sign a message (sender) and verify it (receiver)

lab8partB.py

```
import rsa

(publickey,privatekey) = rsa.newkeys(1024) # RSA KEY GENRATION with key

message = input("Enter a message to encrypt with RSA-") # input

#Signing using RSA  with private key
signmsg = rsa.sign(message.encode('ascii'),privatekey,'SHA-1')
print("Signed output is ",signmsg.hex())

#verify with Public Key
verifymsg = rsa.verify(message.encode('ascii'),signmsg,publickey)
if(verifymsg):
    print("Signature is verified")
else:
    print("Signature is not verified")

```

**Lab 9 : 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

```python
#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/mDCFnpIgVbwvd54uDbQQqm
8t9u8VnmsMgOt+dRyLL2IWt1HfDhymgcnB\n09+cY2rCsuqYbJwdniNUdG4QRy/SAe9sZcz4ui/qz3AqLtOQwF1f9bX0sZTFcJ5j\nKa7i
QpYbA5yVHJJVI3H8CAwEAAQ==\n-----END PUBLIC KEY-----'
~/practical-crypto-course$
```