Lab 4: Asymmetric (Public) Key

Objective: The key objective of this lab is to provide a practical introduction to public key encryption, and with a focus on RSA and Elliptic Curve methods. This includes the creation of key pairs and in the signing process.

🕮 **Video demo:** https://youtu.be/6T9bFA2nl3c

A RSA Encryption (1p)

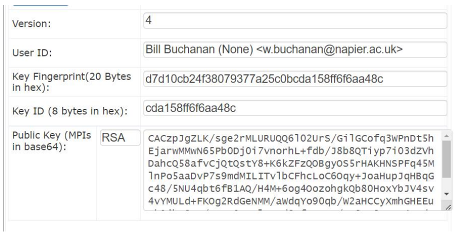
A.1 The following defines a public key that is used with PGP email encryption:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2

mQENBFTzi1ABCADIEWchOyqRQmU4AyQAMj2Pn68Sqo9lTPdPcItwo9LbTdv1YCFz
w3qLlp2RORMP+Kpdi92CIhdUYHDmZfHZ3IWTBgo9+y/Np9UJ6tNGocrgsq4xWz15
4vX4jJRddC7QySSh9UxDpRWf9sgqEv1pah136r95ZuyjC1EXnoNxdLJtx8PliCXc
hV/v4+KfOyzYh+HDJ4xP2bt1S07dkasYZ6cA7BHYi9k4xgEwxVvYtNjSPjTsQY5R
cTayXveGafuxmhSauZKiB/2TFErjEt49Y+p07tPTLX7bhMBVbUvojtt/JeUKV6vK
R82dmOd8seUvhwOHYB0JL+3S7PgFFsLo1NV5ABEBAAG0LkJpbGwgQnVjaGFuYW4g
KE5vbmUpIDx3LmJ1Y2hhbmFuQG5hcGllci5hYy51az6JATkEEwECACMFAlTzi1AC
GwMHCwkIBwMCAQYVCAIJCgsEFgIDAQIeAQIXgAAKCRDsAFZRGtdPQi13B/9KHeFb
l1AxqbafFGRDEvx8UfPnEww4FFqWhcr8RLWyE8/COlUpB/5AS2yvojmbNFMGzURb
LGf/u1LVH0a+NHQu57u8Sv+g3bBthEPh4bKaEzBYRS/dYHOx3APFyIayfm78JVRF
zdeTOOf6PaXUTRx7iscCTkN8DUD3lg/465ZX5aH3HWFFX50OJSPSt0/udqjoQuAr
WA5JqB//g2GfzZe1UzH5Dz3PBbJky8GiIfLm0OXSEIgAmpvc/9NjzAgjOW56n3Mu
sjVkibc+lljw+rOo97CfJMppmtcOvehvQv+KG0LZnpibiWVmM3vT7E6kRy4gEbDu
enHPDqhsvcqTDqaduQENBFTzi1ABCACzpJgZLK/sge2rMLURUQQ6l02UrS/GilGC
ofq3WPnDt5hEjarwMMwN65Pb0Dj0i7vnorhL+fdb/J8b8QTiyp7i03dZVhDahcQ5
8afvCjQtQstY8+K6kZFzQOBgyOS5rHAKHNSPFq45MlnPo5aaDvP7s9mdMILITvlb
CFhcLoC6Oqy+JoaHupJqHBqGc48/5NU4qbt6fB1AQ/H4M+6og4OozohgkQb80Hox
YbJV4sv4vYMULd+FKOg2RdGeNMM/aWdqYo90qb/W2aHCCyXmhGHEEuok9jbc8cr/
xrWL0gDwlWpad8RfQwyVU/VZ3Eg3OseL4SedEmwOO
cr15XDIs6dpABEBAAGJAR8E
GAECAAkFAlTzi1ACGwwACgkQ7ABWURrXT0KZTgf9FUpkh3wv7aC5M2wwdEjt0rDx
nj9kxH99hhuTX2EHXuNLH+SwLGHBq5O2sq3jfP+owEhs8/Ez0j1/fSKIqAdlz3mB
dbqWPjzPTY/m0It+wv3epOM75uWjD35PF0rKxxZmEf6SrjZD1sk0B9bRy2v9iWN9
9ZkuvcfH4vT++PognQLTUqNx0FGpD1agrG0lXSCtJWQXCXPfWdtbIdThBgzH4flZ
ssAIbCaBlQkzfbPvrMzdTIP+AXg6++K9SnO9N/FRPYzjUSEmpRp+ox31WymvczcU
RmyUquF+/zNnSBVgtY1rzwaYi05XfuxG0WHVHPTtRyJ5pF4HSqiuvk6Z/4z3bw==
=ZrP+
-----END PGP PUBLIC KEY BLOCK-----
```

Using the following Web page, determine the owner of the key, and the ID on the key:

https://asecuritysite.com/encryption/pgp1

By searching on-line, can you find the public key of three famous people, and view their key details, and can you discover some of the details of their keys (eg User ID, key encryption method, key size, etc)?



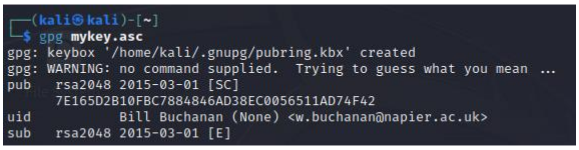
By searching on-line, what is an ASCII Armored Message?

An ASCII Armored Message, often referred to simply as "ASCII armor," is a way to represent binary data, such as cryptographic keys or encrypted messages, using plain text characters from the ASCII character set. This conversion allows binary data to be transmitted or stored as text, making it more accessible for human-readable formats, such as email or text documents.

Save the public key to your Ubuntu instance mykey.asc, and run:

```
gpg mykey.asc
```

What details can you get from the key:



A.2 Bob has a private RSA key of:


```
-----BEGIN RSA PRIVATE KEY-----
\nMIICXgIBAAKBgQDoIhiWs15X/6xiLAVcBzpgvnuvMzHBJk58wOWrdfyEAcTY1OoG\n+6auNFGqQHYHbfKaZlEi4prAo
e01S/R6jpx8ZqJUNOWKNn5G9nmjJha9Pag28ftD\nrsT+4LktaQrxdNdrusP+qIONiYbNBH6qvCrK0aGiucextehnuogg
DcqmRwIDAQAB\nAoGAZCaJu0MJ2ieJxRU+/rRzoFeuXylUNwQC6toCfNY7quxkdDV2T8r038Xc0fpb\nsdrix3CLYuSnZ
aK3B76MbO/oXQVBjDQZ7jVQ5K41nVCEZOtRDBeX5Ue6CBs4iNmC\n+QyWx+u4OZPURq61YG7D+F1aWRvczdEZgKHPXl/+
s5pIvAkCQQDw4V6px/+DJuZV\n5Eg2OOZeOm9Lvaq+G9UX2xTA2AUuH8Z79e+SCus6fMVl+Sf/W3y3uXp8B662bXhz\ny
heH67aDAkEA9rQrvmFj65n/D6eH4JAT4OP/+icQNgLYDW+u1Y+MdmD6A0YjehW3\nsuT9JH0rvEBET959kP0xCx+iFEjl
81tl7QJBAMcp4GZK2eXrxOjhnh/Mq51dKu6Z\n/NHBG3jlCIzGT8oqNaeK2jGLW6D5RxGgZ8TINR+HeVGR3JAzhTNftgM
JDtcCQQC3\nIqReXVmZaeXnrwu07f9zsI0zG5BzJ8VOpBt7OWah8fdmOsjXNgv55vbsAWdYBbUw\nPQ+lc+7WPRNKT5sz
/iM5AkEAi9Is+fgNy4q68nxPl1rBQUV3Bg3S7k7oCJ4+ju4W\nNXCCvRjQhpNVhlor7y4FC2p3thje9xox6QiwNr/5siy
ccw==\n-----END RSA PRIVATE KEY-----
```


And receives a ciphertext message of:


```
uW6FQth0pKaWc3haoqxbjIA7q2rF+GOKx3z9ZDPZGU3NmBfzpD9ByU1ZBtbgKC8ATVZzwj15AeteOnbjO3EHQC4A5Nu0x
KTWpqpngYRGGmzMGtblW3wBlNQYovDsRUGt+cJK7RD0PKn6PMNqK5EQKCD6394K/gasQ9zA6fKn3f0=
```


Using the following code:

```
# https://asecuritysite.com/encryption/rsa_example
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
import base64

binPrivKey = "-----BEGIN RSA PRIVATE KEY-----
\nMIICXgIBAAKBgQDoIhiWs15X/6xiLAVcBzpgvnuvMzHBJk58wOWrdfyEAcTY1OoG\n+6auNFGqQHYHbfKaZlEi4prAo
e01S/R6jpx8ZqJUNOWKNn5G9nmjJha9Pag28ftD\nrsT+4LktaQrxdNdrusP+qIONiYbNBH6qvCrK0aGiucextehnuogg
DcqmRwIDAQAB\nAoGAZCaJu0MJ2ieJxRU+/rRzoFeuXylUNwQC6toCfNY7quxkdDV2T8r038Xc0fpb\nsdrix3CLYuSnZ
aK3B76MbO/oXQVBjDQZ7jVQ5K41nVCEZOtRDBeX5Ue6CBs4iNmC\n+QyWx+u4OZPURq61YG7D+F1aWRvczdEZgKHPXl/+
s5pIvAkCQQDw4V6px/+DJuZV\n5Eg2OOZeOm9Lvaq+G9UX2xTA2AUuH8Z79e+SCus6fMVl+Sf/W3y3uXp8B662bXhz\ny
heH67aDAkEA9rQrvmFj65n/D6eH4JAT4OP/+icQNgLYDW+u1Y+MdmD6A0YjehW3\nsuT9JH0rvEBET959kP0xCx+iFEjl
81tl7QJBAMcp4GZK2eXrxOjhnh/Mq51dKu6Z\n/NHBG3jlCIzGT8oqNaeK2jGLW6D5RxGgZ8TINR+HeVGR3JAzhTNftgM
JDtcCQQC3\nIqReXVmZaeXnrwu07f9zsI0zG5BzJ8VOpBt7OWah8fdmOsjXNgv55vbsAWdYBbUw\nPQ+lc+7WPRNKT5sz
/iM5AkEAi9Is+fgNy4q68nxPl1rBQUV3Bg3S7k7oCJ4+ju4W\nNXCCvRjQhpNVhlor7y4FC2p3thje9xox6QiwNr/5siy
ccw==\n-----END RSA PRIVATE KEY-----"

ciphertext=base64.b64decode("uW6FQth0pKaWc3haoqxbjIA7q2rF+GOKx3z9ZDPZGU3NmBfzpD9ByU1ZBtbgKC8A
TVZzwj15AeteOnbjO3EHQC4A5Nu0xKTWpqpngYRGGmzMGtblW3wBlNQYovDsRUGt+cJK7RD0PKn6PMNqK5EQKCD6394K/
gasQ9zA6fKn3f0=")

privKeyObj = RSA.importKey(binPrivKey)
cipher = PKCS1_OAEP.new(privKeyObj)
message = cipher.decrypt(ciphertext)

print
print ("====Decrypted===")
print ("Message:",message)
```

What is the plaintext message that Bob has been sent?

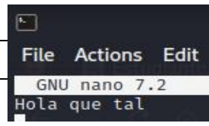
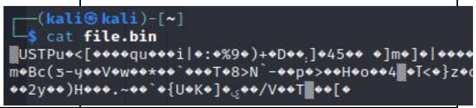
Note: You may have to install Pycryptodome if this example, to do so apply the following command:

```
pip install pycryptodome
```

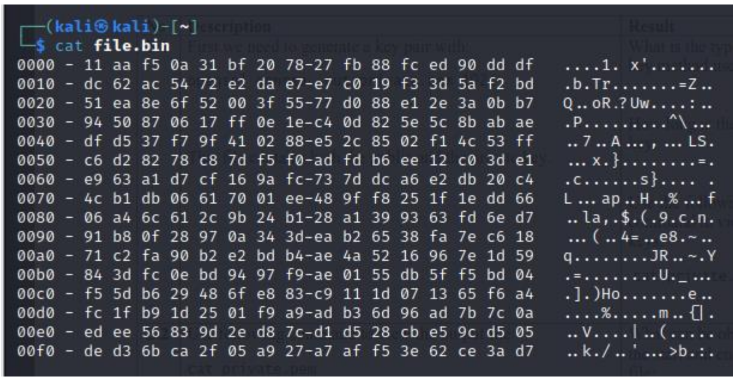
B OpenSSL (RSA) (1p)

We will use OpenSSL to perform the following:

No	Description	Result
B.1	First we need to generate a key pair with: `openssl genrsa -out private.pem 1024` This file contains both the public and the private key.	What is the type of public key method used: ***RSA*** How long is the default key: ***1024*** Use the following command to view the keys: `cat private.pem`
B.2	Use following command to view the output file: `cat private.pem`	What can be observed at the start and end of the file: *At the beginning and end of the file , you can see indicators that show the beginning and end of the private key*
B.3	Next we view the RSA key pair: `openssl rsa -in private.pem -text`	Which are the attributes of the key shown: Modulus, publicExponent, privateExponent, prime1, prime2, exponent1, exponent2, coefficient. What is the number of bits in the public modulus? *4096 bits* How many bits do the prime numbers have? *1992 bits* What is the value of e? *1984 bits*
B.4	Let's now secure the encrypted key with 128-bit AES: `openssl rsa -in private.pem -aes128 -out key3des.pem` You should have a password on the usage of your private key to enhance security by adding an additional layer of protection. This password, often referred to as a passphrase, helps prevent unauthorized access to your private key, which is crucial for securing sensitive data and digital assets.	Why should you have a password on the usage of your private key?
B.5	Next we will export the public key: `openssl rsa -in private.pem -out public.pem -outform PEM -pubout`	View the output key. What does the header and footer of the file identify? *It allows us to identify that the content of the file belongs to a public key.*

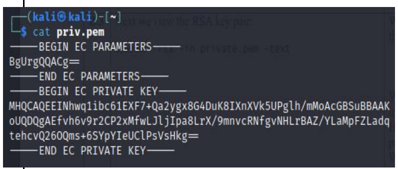
B.6	Now create a file named "myfile.txt" and put a message into it. Next encrypt it with your public key: `openssl pkeyutl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin`	
B.7	And then decrypt with your private key: `openssl pkeyutl -decrypt -inkey private.pem -in file.bin -out decrypted.txt`	What are the contents of decrypted.txt

B.8	What can you observe between these two commands for differing output formats:
	
	`openssl pkeyutl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin`
	
	`cat file.bin`
	
	and:
	
	`openssl pkeyutl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin -hexdump`
	
	`cat file.bin`
		What can you observe in the different of the output files:
		
		The difference is that in the second case the information presentation format is more organized and displayed in hexadecimal values . As the example below shows:



C OpenSSL (ECC) (1p)

Elliptic Curve Cryptography (ECC) is now used extensively within public key signing and key exchange. This includes with Bitcoin, Ethereum, Tor, and IoT applications. In this part of the lab we will use OpenSSL to create an EC key pair. For this we generate a random 256-bit private key (**priv**), and then generate a public key point (which is **priv** multiplied by G). This will use a generator point (G), and which is an (x,y) point on the selected elliptic curve.

No	Description	Result
C.1	First we need to generate a private key with: `openssl ecparam -name secp256k1 -genkey -out priv.pem` The file will only contain the private key, as we can generate the public key from this private key. Now use "`cat priv.pem`" to view your key.	Can you view your key? 
C.2	We can view the details of the ECC parameters used with: `openssl ecparam -in priv.pem -text -param_enc explicit -noout`	Outline these values: Prime (last two bytes): A: *0* B: *7* Generator (last two bytes): *b8* Order (last two bytes): *41*
C.3	Now generate your public key based on your private key with: `openssl ec -in priv.pem -text -noout`	How many bits and bytes does your private key have: *32 bytes and 64 bits* How many bit and bytes does your public key have (Note the 04 is not part of the elliptic curve point): *64 bytes and 128 bits* What is the ECC method that you have used? *secp256k1*

C.4	First we need to generate a private key with: `openssl ecparam -list_curves`	Outline three curves supported: sect113r1, c2pnb163v1, prime239v1

C.5	Let's select two other curves:
	
	`openssl ecparam -name secp128r1 -genkey -out priv.pem`
	`openssl ecparam -in priv.pem -text -param_enc explicit -noout`
	
	`openssl ecparam -name secp521r1 -genkey -out priv.pem`
	`openssl ecparam -in priv.pem -text -param_enc explicit -noout`
		How does secp128k1, secp256k1 and secp512r1 different in the parameters used? Perhaps identify the length of the prime number used, and the size of the base point (G) and the prime number. How does the name of the curve relate to prime number size?

secp128k1:
Have 128 bits prime number length, (G): 256 bits
secp256k1:
Have 256 bits prime number length, (G): 512 bits
secp512r1:
Have 512 bits prime number length, (G): 1024 bits

If you want to see an example of ECC, try here: https://asecuritysite.com/encryption/ecc

D Elliptic Curve Encryption (1p)

D.1 In the following Bob and Alice create elliptic curve key pairs. Bob can encrypt a message for Alice with her public key, and she can decrypt with her private key. Copy and paste the program from here:

https://asecuritysite.com/ecc/hashnew9

Code used:

```
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import serialization
import binascii
import sys

private_key = ec.generate_private_key(ec.SECP256K1())


vals = private_key.private_numbers()
no_bits=vals.private_value.bit_length()
print (f"Private key value: {vals.private_value}. Number of bits {no_bits}")

public_key = private_key.public_key()
vals=public_key.public_numbers()

enc_point=binascii.b2a_hex(vals.encode_point()).decode()

print (f"\nPublic key encoded point: {enc_point} \nx={enc_point[2:(len(enc_point)-2)//2+2]}
\ny={enc_point[(len(enc_point)-2)//2+2:]}")


pem =
private_key.private_bytes(encoding=serialization.Encoding.PEM,format=serialization.PrivateFor
mat.PKCS8,encryption_algorithm=serialization.NoEncryption())

der =
private_key.private_bytes(encoding=serialization.Encoding.DER,format=serialization.PrivateFor
mat.PKCS8,encryption_algorithm=serialization.NoEncryption())



print ("\nPrivate key (PEM):\n",pem.decode())
print ("Private key (DER):\n",binascii.b2a_hex(der))

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

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

print ("\nPublic key (PEM):\n",pem.decode())
print ("Public key (DER):\n",binascii.b2a_hex(der))
```

Verify that the program runs, and observe the difference between the size of the public key and the private key:

```
Private key value:
52962672764586028853413365762758146055572462559267814209516304976678269536309.
Number of bits 255

Public key encoded point:
x=d95a447e696485608f427167da0971a56f21a69258666554622a96e52a15581f
y=f84646b103d0ba527d91df8d713b73f89c8253c106ad71605a088a69ddff036c

Private key (PEM):
 -----BEGIN PRIVATE KEY-----
MIGEAgEAMBAGByqGSM49AgEGBSuBBAAKBG0wawIBAQQgdRfPe8CiUNuE31DTtHOG
/BKhRCVWI7xBFCaOPhk4yDWhRANCAATZWkR+aWSFYI9CcWfaCXGlbyGmklhmZVRi
KpblKhVYH/hGRrED0LpSfZHfjXE7c/icglPBBq1xYFoIimnd/wNs
-----END PRIVATE KEY-----

Private key (DER):
 b'308184020100301006072a8648ce3d020106052b8104000a046d306b02010104207517cf7bc0a250d
b84df50d3b47386fc12a144255623bc4114268e3e1938c835a14403420004d95a447e696485608f42716
7da0971a56f21a69258666554622a96e52a15581ff84646b103d0ba527d91df8d713b73f89c8253c106a
d71605a088a69ddff036c'

Public key (PEM):
 -----BEGIN PUBLIC KEY-----
MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAE2VpEfmlkhWCPQnFn2glxpW8hppJYZmVU
YiqW5SoVWB/4RkaxA9C6Un2R341xO3P4nIJTwQatcWBaCIpp3f8DbA==
-----END PUBLIC KEY-----

Public key (DER):
 b'3056301006072a8648ce3d020106052b8104000a03420004d95a447e696485608f427167da0971a56
f21a69258666554622a96e52a15581ff84646b103d0ba527d91df8d713b73f89c8253c106ad71605a088
a69ddff036c'
```

D.2 Let's say we create an elliptic curve with $y^2 = x^3 + 7$, and with a prime number of 89 ($y^2 = x^3 + 7 \pmod{89}$), generate the first five (x,y) points for the finite field elliptic curve. You can use the Python code at the following to generate them:

https://asecuritysite.com/encryption/ecc_points_real
(or for simpler code you can use https://asecuritysite.com/encryption/ecc_points3)

First five points:

```
Prime number:               89
a,b 0 7
y^2 = x^3 + ax +b (mod p)
(1, 39) (1, 50) (3, 52) (3, 37) (4, 31)
```

E RSA (1p)

E.1 A simple RSA program to encrypt and decrypt with RSA is given next. Prove its operation:

```
import rsa
(bob_pub, bob_priv) = rsa.newkeys(512)

msg='Here is my message'
ciphertext = rsa.encrypt(msg.encode(), bob_pub)
message = rsa.decrypt(ciphertext, bob_priv)
print(message.decode('utf8'))
```

Now add the lines following lines after the creation of the keys:

```
print (bob_pub)
print (bob_priv)
```

Can you identify what each of the elements of the public key (e,N), the private key (d,N), and the two prime number (p and q) are (if the numbers are long, just add the first few numbers of the value):

```
Here is my message
PublicKey(83511008400956575271173256260486195690942715119486873725086767304256169313533963915
16657096315585865927474453150579869341496404367737341717633839736706781, 65537)
PrivateKey(835110084009565752711732562604861956909427151194868737250867673042561693135339639151
6657096315585865927474453150579869341496404367737341717633839736706781, 65537,
3158756713385282427326722079804068396452048407917603968401324251499467750026264208513942922078
0474262226087686831000758688475882458640617132548067094716673,
7174475198155379362550887948513735988272543686667116161224162247792255900967635741,
1164001632097466719197630161523996216472974623364966431816547301261045441)
```

When you identity the two prime numbers (p and q), with Python, can you prove that when they are multiplied together they result in the modulus value (N):

Proven Yes/No

E.2 We will follow a basic RSA process. If you are struggling here, have a look at the following page:

https://asecuritysite.com/encryption/rsa

First, pick two prime numbers:

p= *2*
q= *13*

Now calculate N (p.q) and PHI [(p-1).(q-1)]:

N= *26*
PHI = *12*

Now pick a value of *e* which does not share a factor with PHI [gcd(PHI,e)=1]:

e= 5

Now select a value of d, so that (e.d) (mod PHI) = 1:
[Note: You can use this page to find *d*: https://asecuritysite.com/encryption/inversemod]

d= 1

Now for a message of M=5, calculate the cipher as:

$C = M^e \pmod N$ = *5*

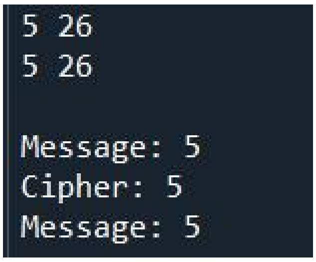
Now decrypt your ciphertext with:

$M = C^d \pmod N$ = *5*

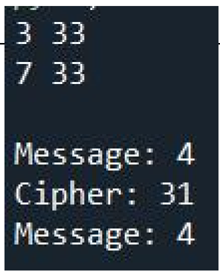
Did you get the value of your message back (M=5)? If not, you have made a mistake, so go back and check.

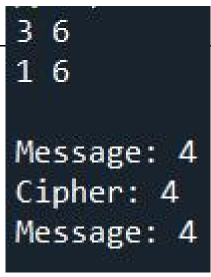
Now run the following code and prove that the decrypted cipher is the same as the message:

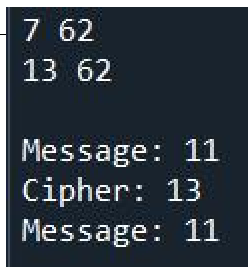
```
import libnum

p=11
q=3
N=p*q
PHI=(p-1)*(q-1)
e=3

d= libnum.invmod(e,PHI)

print (e,N)
print (d,N)
M=4
print ("\nMessage:",M)
cipher = M**e % N
print ("Cipher:",cipher)
message = cipher**d % N
print ("Message:",message)
```



Select three more examples with different values of p and q, and then select e in order to make sure that the cipher will work:







E.3 In the RSA method, we have a value of e, and then determine d from (d.e) (mod PHI)=1. But how do we use code to determine d? Well we can use the Euclidean algorithm. The code for this is given at:

https://asecuritysite.com/encryption/inversemod

Using the code, can you determine the following:

Inverse of 53 (mod 120) = *77*

Inverse of 65537 (mod 1034776851837418226012406113933120080) = *568411228254986589811047501435713*

Using this code, can you now create an RSA program where the user enters the values of p, q, and e, and the program determines (e,N) and (d,N)?

https://github.com/Erick-Tx/Cryptography_Sem_2_2023_Yachay_Lab_04/blob/main/E3.py

```
Ingresa un número primo (p): 2
Ingresa otro número primo (q): 3
Ingresa el exponente público (e): 5
Clave Pública (e, N): (5, 6)
Clave Privada (d, N): (1, 6)
```

E.3 Run the following code and observe the output of the keys. If you now change the key generation key from 'PEM' to 'DER', how does the output change:

```
from Crypto.PublicKey import RSA

key = RSA.generate(2048)

binPrivKey = key.exportKey('PEM')
binPubKey =  key.publickey().exportKey('PEM')

print (binPrivKey)
print (binPubKey)
```


```
b'-----BEGIN RSA PRIVATE KEY-----\nMIIEogIBAAKCAQEArvqUhgx5X4u1gyHA3vw1MApxMxQ/
7rfdrx8eJODZXES4ufZ1\nSl3ukrWYI3r8VrRkeLyg9Q4fXVcSNJnvVmGm9SBb2KJrx58+7jTdOeajmSpkdqfs\nAg
d+8wDKCPbIQbEQ5iLiUAy6FC+T6EUQzqXp5j17wUGGYZicLx6vn9WzzJs4A5rA\nJWUfhMPkgTtV+SduxDryKeHNW4
CTqJiYUjzU7H3666OBSL8iaKoEd7NcNxBwurJT\nTiAvMNvlSaOvG3J4DWRE8cH8/
VKkI3mUS9+ted8fIOeh0/7TcUfFq+4XeuU7ohXw\n0dngrmmKWePG0RHkGXp6/
XVsWFnkP2LN0WMU1QIDAQABAoIBAAdjasaPO8fCRy+K\nixGPTjKhWj7JJdrvVhO5KWVuvhbUiwVFKBY0CakVtF8ZK
BMEKnLjqhs21bG0Bb+F\nWXMJDArxKxemYg0rhvV5c3boq3NB9DyDYiaCyQQ7PgyrWPDXKGrDQdGV+/
xyttjZ\no7hKDqqLYZCbMvCniD7AGbPSyvUCbdUQkyZXuZxm12kssk+kgrYexWrtCD70vnKO\nGaknN6tZrXoBMwvE
WsffAODa7+cKP2p8fDEP5OZLSKXyc3eyCKzdfpk4ZBrPfJ2B\nDCPQywmjNmnYYRZ6Ru1p4MV931/2XYfLJbI+jjWc
IGJ+8aPCvDraeCYnnLsnbNNZ\ndIXhOzkCgYEAu1VasUIdpTFovKg9bwGBQQr4834bD30z28YF/
wZZ0HQKn2WVYUng\nay30Y35WfSlav3BeJKPD2mkgTJITLuE1MBSAESEBQGo1zh9h8/
rBsgIedR9MnN7k\nhlrXarHk0gdhOHk/moYiSGbqk9IvsiYPynrZAVnsoLLFGPxgON0oxa0CgYEA7x3r\nTHApTO/
7H7SIXmnYy6CPHz1YrzTeD7gm8VfU+OaBkuuvV9obMKzFnC01e/QjyMQy\nlaAbcEL2jeUjzw5dt9OvuqclrpBUL/
XUop6CKWfdJxNFaDA01lsTnpdEj1Anhoh0\nI1UbWUOmGHJxHRc1/9jk3nDWZjsEhowXM00bYMkCgYBIZr+rfIMaol
L5fwhnS6yZ\n59S5vsZUoS4WxZzxqJGzx/WHD2rBdB0BL3yhq2Vf8Ei7boFkJcr/
Mdwi294AI1g8\nIeAxPpFORP5QkL553jNYSgtdduBUTK0D5D4XZXOrXODNQ8c6QJclZmlJAlYewykd\nN4eBRzD2Fs
GoW3oQ4CDu7QKBgBQU7y38PXu4q959yvYi7ZIsadi1zjzos72eOacL\nOVeuBDPSl2dJaf8sg2waX+OJHXg/
AgCO8b774LkKBXfePQCzRo7mesOuW2Orwa/m\noAfaMTiW5b1aTxWxkHRBQADJOoiR7S8KdsGIbOqgsd8/k8lo7c/
AqkDrs/WYzloA\nn8wBAoGAOkR7dmSFl5l974qvcKqYXaZ0MvNt0nFKmxdKP5F8G6atVy2fQxAU6LFe\nwSWngOV/
S/lwbgrVV+m/nzteOA0k0ujduz0NJ14tfi2DRD7Z2nTmO5Jbihx9+Qex\nPedgpFmWG9b/DtTePliC2HHPbzw1TSq/
Lsp4jdiYCsaMxULpsOU=\n-----END RSA PRIVATE KEY-----'
b'-----BEGIN PUBLIC KEY-----
\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEArvqUhgx5X4u1gyHA3vw1\nMApxMxQ/
7rfdrx8eJODZXES4ufZ1Sl3ukrWYI3r8VrRkeLyg9Q4fXVcSNJnvVmGm\n9SBb2KJrx58+7jTdOeajmSpkdqfsAgd+
8wDKCPbIQbEQ5iLiUAy6FC+T6EUQzqXp\n5j17wUGGYZicLx6vn9WzzJs4A5rAJWUfhMPkgTtV+SduxDryKeHNW4CT
qJiYUjzU\n7H3666OBSL8iaKoEd7NcNxBwurJTTiAvMNvlSaOvG3J4DWRE8cH8/
VKkI3mUS9+t\ned8fIOeh0/7TcUfFq+4XeuU7ohXw0dngrmmKWePG0RHkGXp6/
XVsWFnkP2LN0WMU\n1QIDAQAB\n-----END PUBLIC KEY-----'
```

```
b'0\x82\x04\xa3\x02\x01\x00\x02\x82\x01\x01\x00\xb6\xa7\x9d\xfc\x8d\xf0\xfc\x0b\xcfh\x94\xf5\xfe\xc5^\xd16\x8d}\xabG\xdfr\xc4\x7f\x81\xdc\x92\xd5\xae\xa6e\xabn~\xca\x16\xc5U-
\xf4\xb1b,>q\x00\x1c\x10\\Ax\xe8\xd6\x9f\x18$\xbae\xf8F)\xb8\x14\x067\xfd!\xa4\x8bv\xa5\x9f#\x12\xae\xfb\xbeQ\xeb\x94}
\x13`\x84\t\x9b\xb6{\xd1\x17\x8cp\x8dL\xee\xc16\x0cw\xb4H\xa2\xba\xd5\r33\xb3\x1b\x9a\x1e\x17U?(\xdf#y\xc4P\xd7Y\x19\x01\xe0\xf7\xa1\x85Va\xeb\x0f\xd4\xda\x85\xf6L\x072@\xc5!
N\x87\xf6\xcb\x02\x0fi&T\xf1!l\x9f\x045\x99\x11\x149ZZ\x0f\xd9\xd4t\xec\x1bKM)\x86\x9f6y\xb3\x0bQ=\xbfY\x829e\xa3\xce\xa0\xf5!\x910\xd8\xaa\xc7"\x98sYR2\x0f\xe4 \xbf%
\x08\x18\xf4\xfa\x06\x83\xef\xea\x1d\x94S\xe7WHlC.x\x8e\xe1\xd7PV\xff\x9b\x89f\'\xc0JZ~\xa2\xa7\xc9\xe1_R9\x8c\x00\x7f\xf2\x0e\x10Y\xff\x93\x89\x02\x03\x01\x00\x01\x02\x82\x01\x0
0\x05\xbe}\xb4l\x8eP\x8e\xda\x85\xf2qN\x02\xcc\xf9\x8dZ\xa3\xc2\x18/
\x89\xbfn7\x1aE\xaf\xdc\x9f5\x01P\'B\x99\xb6\xb8\xf8\n\x02\xee\x10p"C\x9b\x9eT\x18g\x02\x03\x85\xdbO\x84a"\x8e\xaaO4\xaa\x01f\xbf\x81\x1a5\xaf\xfd\xb2/\xab;\x15\x80$
\x05vm~\x92\x88n\xd2-P\xa5\t\xf0\x9c0\xf8-1R\xf1\x0c\xc5`\x00\xba$\x81 \xb6\x18?\x1f\x85"T+\'\xe1H\xba\x08\xc4\xcc\xf3\x81`\x9e\xc8\xf2\xf7v\xed\xdc\xeeE/
\xd3\x02L\x90\xb8\xed\xf4\xeeC\xb7N\x8f\xcb\xcc\xdc1G\xe9\'\xe2\xfe\xba\x11U\xc8J\x04i\xa8Y\xabD\xfeJ\xe8\xa1\xa3\n\x83\xd2-
\x9d>q+Z\xd4e\xcdb\x81\xb7\xe8l\xf8\x1ah\xba\xafb\xe0\xa4*I\x15B\xeb\xa6\xea\x10\xa7}\xda\x06g\xe9\xadS!
\x91l>\xbfT\x12o\x97\x0c5[l\x0e\xd5\xdc\xb7\xf6q\x88\x1c"\x06\xda~8\xcci2\x83\xf6^\x99W{\x93*O\xe8>>\x91\x02\x81\x81\x00\xcd\xce\r\x1fVP\x82\x9b\xc8`QH\x83\x9a\x1eo\xbd\xb7\x1b\x
12uI \x19\'\x9fa\x99\xd2\t\xd8\xb9f\xdf1U\x7f\x92\xe7\x1d\xc8\x00e\xb9\x85\x00\xbe\x88r\xbb=i^\x0f\x95,|\xe1\xe0\x9cU\t\xa9\xd1\xa3&>C\x86(yN<\xfe \xd4\xc0\xa5}S\xc1,
\x1d\xb2x\x13iM\xac\x066\xf3\x933\x03\xbd\xfch$CG/\xbdbmN\x93ae:\xa3\xc4SkMZXh\x99\x8d\xc9\xf4\x89\xb7\xd2\xbc\xe8\xa5\x02\x81\x81\x00\xe34 \xf5z\'\x0b\xc2\xe7z;\xefDY\x9b\x9f|
k\xdf?l\xd4\xa0\xfb\xce,|\x87\xea\x1e\xd3]\x8f\xd0\x92$$\x8b\x8a\x03ZF\x01\xf4\x12N\x9e\xe2\xc3X/|\x15X\x99\xe6xH\xbf\xca\x8f\xe0\xf7\xe39\x18\xe2R+\xe5}
m\xa3\x8c\x81\xe6\x0cO\x9e3\x98w\x82\x83Wg\xe1\xc0\xa7\x12>\xa2V\xfd1\x8d\x80\x0c;_aFUT\xdf(\xe2\xfa\xc8>\'8\xec\xe6\xd8\xef\xdf\xc9H$
\xd72\xd1Y\x14\x98&\x15\x02\x81\x81\x00\xc18\xc5\x858j\x9aU\\Ia\xc7-
\x17\x93\x9f\xe1\x84z<X5q\x97le\xed"\x97\xdaz\xc9\x86\xd5\xe1\xe1\x9d\xc6\xf4\x8a7nx\x137\xd6=\x0c\xaa{4\xeb*N\x83\x05\xa3\xb2\x06I\x80et4\xc8rm^\xc4$W\xcf\xa6\xe0\t>\xf6\'\xe9\x
84\xab\x06\x9aS&\xfdB\x8e\x94\x05\xb9\xad\xf1\xf3\xa1}\xcb\xee\x92\xad?\xb6\x88\x9eRu\xc5UY\xcf6Vj\xe3\x81ZcJ\xc1p\'\xbe\x19V\xa6\xda\x89\xa9\x02\x81\x80C]
\x06T\xa1\xf6\x07\xb7\xae\xf7\xff\xe3.
\t\xbfif\xf3x\xca\x8eR\xf0\xb4\x17\xe9\xfa\xb65\x85\x148\x0eV\xd1\xad\xa3\xa47\xb4L\xc4\xaa\xf9\x99\x9f\xf3\xdd\xd9\xfb@\x80\xf2\xb7ER\x1e\xe1\xba\xf3\xac\xfb\x0f\xa40\x1e\x9f\xa
3\xab^,u c\x9ds\xfe\xbcptm\xc5)N\x15\xd18\x0e\xed\xba,\x15\xe7\xe2\xc4\x19g\x18\xb9U\xcb:\x96<F\xba\x11\xaa7J\xaeHc}\x02_\x04\xa3\xe2\x1f\x02+
\x10\x04\xf0e8\r\x02\x81\x80\x1e0\xc1\xfc\xafC\x9c\xb0\xac) A\xba\xa9\xef\xe7\xe1\xda*\xdd\x83\xb1\xdf\xcaxs\x04\x03@\xc7ol;*\x98\x0e\x85\xa0|\xc6mu\xf1a\x8f]\xfd\x18\x0b.
\xca\xdd\xae\xd6\xb1\xe2!\xdc\xea\x7f\x1eie\xcbZ\x08\xb7\x1bKm\xaa06_\xda\xfb\x90\xbeC+x\x9a\xe8\xc4\xf8_\xf1\xa6\x94\xde\x97\xdb\xb7\xba\xc8\x7f\x05\xbc~(\xb5Vd\xa7\xdc\xac$
\xe0\xe7/\x1d\x1c\xc2\xbb\x9e)G\xe5\x10\xe8%r_\xe8\x958|\xef'
b'0\x82\x01"0\r\x06\t*\x86H\x86\xf7\r\x01\x01\x01\x05\x00\x03\x82\x01\x0f\x000\x82\x01\n\x02\x82\x01\x01\x00\xb6\xa7\x9d\xfc\x8d\xf0\xfc\x0b\xcfh\x94\xf5\xfe\xc5^\xd16\x8d}
\xabG\xdfr\xc4\x7f\x81\xdc\x92\xd5\xae\xa6e\xabn~\xca\x16\xc5U-\xf4\xb1b,>q\x00\x1c\x10\\Ax\xe8\xd6\x9f\x18$\xbae\xf8F)\xb8\x14\x067\xfd!
\xa4\x8bv\xa5\x9f#\x12\xae\xfb\xbeQ\xeb\x94}\x13`\x84\t\x9b\xb6{\xd1\x17\x8cp\x8dL\xee\xc16\x0cw\xb4H\xa2\xba\xd5\r33\xb3\x1b\x9a\x1e\x17U?
(\xdf#y\xc4P\xd7Y\x19\x01\xe0\xf7\xa1\x85Va\xeb\x0f\xd4\xda\x85\xf6L\x072@\xc5!N\x87\xf6\xcb\x02\x0fi&T\xf1!l\x9f\x045\x99\x11\x149ZZ\x0f\xd9\xd4t\xec\x1bKM)
\x86\x9f6y\xb3\x0bQ=\xbfY\x829e\xa3\xce\xa0\xf5!\x910\xd8\xaa\xc7"\x98sYR2\x0f\xe4 \xbf%
\x08\x18\xf4\xfa\x06\x83\xef\xea\x1d\x94S\xe7WHlC.x\x8e\xe1\xd7PV\xff\x9b\x89f\'\xc0JZ~\xa2\xa7\xc9\xe1_R9\x8c\x00\x7f\xf2\x0e\x10Y\xff\x93\x89\x02\x03\x01\x00\x01'
```

F PGP(1p)

F.1 The following is a PGP key pair. Using https://asecuritysite.com/encryption/pgp, can you determine the owner of the keys (or use **gpg mykey.key**):


```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: OpenPGP.js v4.4.5
Comment: https://openpgpjs.org

xk0EXEOYvQECAIpLP8wfLxzgcolMpwgzcUzTlH0icggOIyuQKsHM4XNPugzU
X0NeaawrJhfi+f8hDRojJ5Fv8jBI0m/KwFMNTT8AEQEAAc0UYmlsbCA8Ymls
bEBob21lLmNvbT7CdQQQAQgAHwUCXEOYvQYLCQcIAwIEFQgKAgMWAgECGQEC
GwMCHgEACgkQoNsXEDYt2ZjkTAH/b6+pDfQLi6zg/Y0tHS5PPRv1323cwoay
vMcPjnWq+VfiNyXzY+UJKR1PXskzDvHMLOyVpUcjle5ChyT5LOw/ZM5NBFxD
mL0BAgDYlTsT06vVQxu3jmfLzKMAr4kLqqIuFFRCapRuHYLOjw1gJZS9p0bF
S0qS8zMEGpN9QZxkG8YEcH3gHxlrvALtABEBAAHCXwQYAQgACQUCXEOYvQIb
DAAKCRCg2xcQNi3ZmMAGAf9w/XazfELDG1W35l2zw12rKwM7rK97aFrtxz5W
XwA/5gqoVP0iQxklb9qpX7RVd6rLKu7zoX7F+sQod1sCWrMw
=cXT5
-----END PGP PUBLIC KEY BLOCK-----

-----BEGIN PGP PRIVATE KEY BLOCK-----
Version: OpenPGP.js v4.4.5
Comment: https://openpgpjs.org

xcBmBFxDmL0BAgCKSz/MHy8c4HKJTKcIM3FM05R9InIIDiMrkCrBzOFzT7oM
1F9DXmmsKyYX4vn/IQ0aIyeRb/IwSNJvysBTDU0/ABEBAAH+CQMIBNTT/OPv
TJzgvF+fLOsLsNYP64QfNHav5O744y0MLV/EZT3gsBwO9v4XF2SsZj6+EHbk
O9gWi31BAIDgSaDsJYf7xPOhp8iEWWwrUkC+jlGpdTsGDJpeYMIsVVv8Ycam
0g7MSRsL+dYQauIgtVb3dloLMPtuL59nVAYuIgD8HXyaH2vsEgSZSQn0kfvF
+dWeqJxwFM/uX5PVKcuYsroJFBEO1zas4ERfxbbwnsQgNHpjdIpueHx6/4EO
b1kmhOd6UT7BamubY7bcma1PBSv8PH31Jt8SzRRiaWxsIDxiaWxsQGhvbWUu
Y29tPsJ1BBABCAAfBQJcQ5i9BgsJBwgDAgQVCAoCAxYCAQIZAQIbAwIeAQAK
CRCg2xcQNi3ZmORMAf9vr6kN9AuLrOD9jS0dLk89G/XfbdzChrK8xw+Odar5
V+I3JfNj5QkpHU9eyTMO8cws7JWlRyOV7kKHJPks7D9kx8BmBFxDmL0BAgDY
lTsT06vVQxu3jmfLzKMAr4kLqqIuFFRCapRuHYLOjw1gJZS9p0bFS0qS8zME
GpN9QZxkG8YEcH3gHxlrvALtABEBAAH+CQMI2Gyk+BqVOgzgZX3C80JRLBRM
T4sLCHOUGlwaspe+qatOVjeEuxA5DuSs0bVMrw7mJYQZLtjNkFAT92lSwfxY
gavS/bILlw3QGA0CT5mqijKr0nurKkekKBDSGjkjVbIoPLMYHfepPOju1322
Nw4V3JQO4LBh/sdgGbRnwW3LhHEK4Qe70cuiert8C+S5xfG+T5RWADi5HR8u
UTyH8x1h0ZrOF7K0Wq4UcNvrUm6c35H6lClC4Zaar4JSN8fZPqVKLlHTVcL9
lpDzXxqxKjS05KXXZBh5wl8EGAEIAAkFAlxDmL0CGwwACgkQoNsXEDYt2ZjA
BgH/cPl2s3xCwxtVt+Zds8NdqysDO6yve2ha7cc+Vl8AP+YKqFT9IkMZJW/a
qV+0VXeqyyru86F+xfrEKHdbAlqzMA==
=5NaF
-----END PGP PRIVATE KEY BLOCK-----
```


F.2 Using the Node.js code at the following link, generate a key:

https://asecuritysite.com/encryption/openpgp

Note: to add opengpg, you can install the required library with:

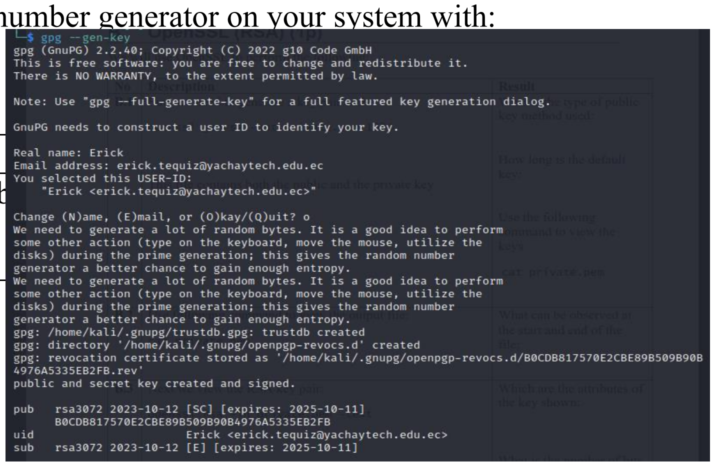
```
npm install openpgp
```

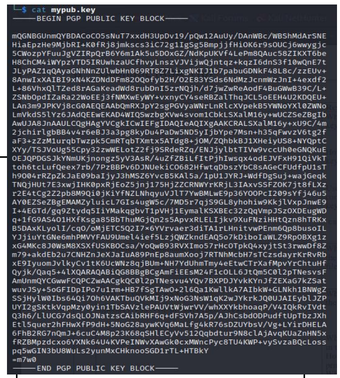
F.3 An important element in data loss prevention is encrypted emails. In this part of the lab we will use an open source standard: PGP.

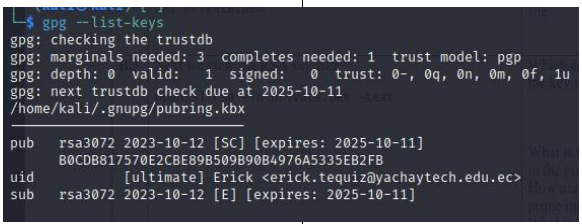
In this challenge, you should install a random number generator on your system with:

```
sudo apt-get install rng-tools
```

No	Description
1	Create a key pair with (RSA and 2,048-b `gpg --gen-key`



	Now export your public key using the form of:
	
	`gpg --export -a "Your name" > mypub.key`
	
	Now export your private key using the form of:
	
	`gpg --export-secret-key -a "Your name" > mypriv.key`
	
	The private key is longer than the public key.
		
		Outline the contents of your key file:

2	Now send your lab partner your public key in the contents of an email, and ask them to import it onto their key ring (if you are doing this on your own, create another set of keys to simulate another user, or use Bill's public key – which is defined at http://asecuritysite.com/public.txt and send the email to him):
	
	`gpg --import` *theirpublickey*`.key`
	
	Now list your keys with:
	
	`gpg --list-keys`
	
	
		Which keys are stored on your key ring and what details do they have:

3	Create a text file, and save it. Next encrypt the file with their public key: `gpg -e -a -u "Your Name" -r "Your Lab Partner Name" hello.txt` -a: Generates output in ASCII-armored format, which means the encrypted data will be presented in a human-readable text format and can be safely transmitted as text. This is useful when you need to share or email the encrypted file. -u "Your Name": Specifies the user ID (or key) to use for encryption. You should replace " Your Name" with your actual GPG user ID or key identifier. Encryption will use your public key to encrypt the file, ensuring that only the corresponding private key can decrypt it. -r "Your Lab Partner Name": Specifies the recipient's user ID or key to use for encryption. " Your Lab Partner Name" should be replaced with the GPG user ID or key identifier of your lab partner. This means that your lab partner will be able to decrypt the file using their corresponding private key. *The resulting ".asc" file has a specific format: it begins with header information detailing encryption and key details, including GPG version, key IDs for the recipient and sender, and metadata. Following the header, the encrypted data from "hello.txt" is present, in a non-human -readable format. Additionally, there may be an end-of-file marker, denoting the conclusion of the encrypted data.*	What does the –a option do: What does the –r option do: What does the –u option do: Which file does it produce and outline the format of its contents:
4	Send your encrypted file in an email to your lab partner, and get one back from them. Now create a file (such as myfile.asc) and decrypt the email using the public key received from them with: `gpg –d myfile.asc > myfile.txt`	Can you decrypt the message: *yes, the message was decrypt*

5	Next using this public key file, send Bill (w.buchanan@napier.ac.uk) an encrypted question (**http://asecuritysite.com/public.txt**).	Did you receive a reply: *Yes, I receive the reply.*
6	Next send your public key to Bill (w.buchanan@napier.ac.uk), and ask for an encrypted message from him.	

G SSH Key pairs (1p)

G.1 On your VM, go into the ~/.ssh folder. Now generate your SSH keys:

```
ssh-keygen -t rsa -C "your email address"
```

The public key should look like this:

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQDLrriuNYTyWuC1IW7H6yea3hMV+rm029m2f6IddtlImHrOXjNwYyt4Elkkc7AzO
y899C3gpx0kJK45k/CLbPnrHvkLvtQ0AbzWEQpOKxI+tW06PcqJNmTB8ITRLqIFQ++ZanjHWMw2Odew/514y1dQ8dccCO
uzeGhL2Lq9dtfhSxx+1cBLcyoSh/lQcs1HpXtpwU8JMxWJl409RQOVn3gOusp/P/0R8mz/RWkmsFsyDRLgQK+xtQxbpbo
dpnz5lIOPWn5LnTOsi7eHmL3WikTyg+QLZ3D3m44NCeNb+bOJbfaQ2ZB+lv8C3OxylxSp2sxzPZMbrZWqGSLPjgDiFIBL
w.buchanan@napier.ac.uk
```

View the private key. What is the **DEK-Info** part, and how would it be used to protect the key, and what information does it contain?

"DEK-Info" is a part of a file that contains an encrypted private key and is used in cryptography to specify the algorithm and parameters used to protect the private key. It consists of two parts: the encryption algorithm and the associated parameters for that algorithm. This information is essential for correctly decrypting the private key, as it provides details on how the encryption was performed, ensuring that only those with the proper information can securely decrypt the private key.

On your Ubuntu instance setup your new keys for ssh:

```
ssh-add ~/.ssh/id_git
```

Now create a Github account and upload your public key to Github (select Settings-> **New SSH key** or **Add SSH key**). Create a new repository on your GitHub site, and add a new file to it. Next go to your Ubuntu instance and see if you can clone of a new directory:

```
git clone ssh://git@github.com/<user>/<repository name>.git
```

If this doesn't work, try the https connection that is defined on GitHub.

H Additional (1p)

The following is code which performs RSA key generation, and the encryption and decryption of a message (https://asecuritysite.com/encryption/rsa_example):

```
from Crypto.PublicKey import RSA
from Crypto.Util import asn1
from base64 import b64encode
from Crypto.Cipher import PKCS1_OAEP
import sys

msg = "hello..."

if (len(sys.argv)>1):
        msg=str(sys.argv[1])

key = RSA.generate(1024)

binPrivKey = key.exportKey('PEM')
binPubKey =  key.publickey().exportKey('PEM')

print ("====Private key===")
print (binPrivKey)
print
print ("====Public key===")
print (binPubKey)

privKeyObj = RSA.importKey(binPrivKey)
pubKeyObj =  RSA.importKey(binPubKey)


cipher = PKCS1_OAEP.new(pubKeyObj)
ciphertext = cipher.encrypt(msg.encode())

print
print ("====Ciphertext===")
print (b64encode(ciphertext))

cipher = PKCS1_OAEP.new(privKeyObj)
message = cipher.decrypt(ciphertext)


print
print ("====Decrypted===")
print ("Message:",message)
```

```
====Private key===
b'-----BEGIN RSA PRIVATE KEY-----\nMIICXAIBAAKBgQDUk8yo6Tf9+1rxZHIeMOd/
suqxDcbr+nsgIb+A0uzen9y9sYsQ\nhefvGsHPq+8UuNrc9fb9wgj4a8NExmS1SuPvMs2uPKOh6kd+ZWXTXvWdib
Kz78X+
\nMmm8YLkCUniaxsRf1F3gzUuo4mzw+A96orqIBO2KUDXbWFb+YL5u918rwwIDAQAB\nAoGAQH8HXUhFQL3+fbbo
GNjlpAz4v/daH0bEDJ+wprlYPMNsCfHQK5c1I4qOux+D\nTlo3rUrDQYKZjiXD/R3keEzckqy/
RrBaUPbMWCKZ869BxQWsxAG7MGgLuY3zThoU\nxqZjFicG4KtwiThANdZyUMiHxwAMZSPfQEE3E34w3bqYzkECQQ
DgExT0qwjECsKM\ncmTcIZtHzeq6pmPWAU64UpNafdvEeRAoaXw3XornPhRgUuOPc95HqbHuCIPvKP4S\nyonZ2t
VbAkEA8t1eHSmo15T2+0YOwmkK3WWQAwaMDd7n/
0Rxk8cIMwqE17cdBi2I\nksbb45efve97V1tdZrQ3Ax4uleRKuqiHuQJBAJJIwtiMRDfAEoygLepsYjR5YTGj\na
PAQgHTjK+Ha4ruckYjgfMClDyeyJplP49IGQeolBc0xb/iVmXFpGp/
INpsCQAcM\nYaTbBqlrC1fzio4d2lDel7VexUXJyFXiOKGfTx+cGIriJ5d8TtBjsRID3oD2qE+6\nuRU3+bi/
cX6sJ3cPWRECQEaaLW8c3FQrkWKKafT9yx2qNBCsuisy3Dq8TZYE8U4/
\nKhwxunse1rOfIlSfHekBxqa5FiZlS0mjyiUPAFpaR7E=\n-----END RSA PRIVATE KEY-----'
====Public key===
b'-----BEGIN PUBLIC KEY-----
\nMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDUk8yo6Tf9+1rxZHIeMOd/
suqx\nDcbr+nsgIb+A0uzen9y9sYsQhefvGsHPq+8UuNrc9fb9wgj4a8NExmS1SuPvMs2u\nPKOh6kd+ZWXTXvWd
ibKz78X+Mmm8YLkCUniaxsRf1F3gzUuo4mzw+A96orqIBO2K\nUDXbWFb+YL5u918rwwIDAQAB\n-----END
PUBLIC KEY-----'
====Ciphertext===
b'mVy+M43ClwSgXJY207oFxhJzAPS8G/
uinF2xM+nPoi9HO4IHPjbMq4s1ikDxt3Y9MLUx8K0BiOP6SiFh5kCFZ8rlVi/
X+50LSBH5acNIvsXc9vzMA0F6ppqMSehnlCC2OvU1rpFn4fqilPe6oltDuorKmNpLcYAcl+PbOsii/PA='
====Decrypted===
Message: b'hello...'
```

Can you decrypt this:


```
fIVuuWFLVANs9MjatXbIbtH7/nOdBpDirXKi82jZovXS/krxy43cPOJ9jlNz4dqxLgdiqtRe1AcymXO6JUo1SrcqDEh3l
QxoU1KUvV7jG9GE3pSxHq4dQlcWdHz95b9go6QYbe/5S/uJgolR+S9qaDE8tXYysP8FeXIPdOdXxHo=
```


The private key is:


```
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQCfQfirYVXgzT90v6SqgeID7q/WK1XaVTNGVFolDUOcrXl/egRG
4iag5tiTbrMYCQ8CSTYn7q0U4AmBXihlbWDqf6MMk6OEoDxdWZTiG1MmQ1wZikFE
s7sYSog/poYleCeYW8kVzHNWnt9IuQWekIg6ZHkwp4NE/aW8HxvEwYRqCQIDAQAB
AoGAE6rkiFmxbtO6GHNwZQQ8QssP2Q2qARgjiGxzY38DWg6MYiNR8uUL6zQHDBIQ
OQgpW9lpwD24DOtpsRnNOFVtMeafcxmykX+qHGtNeKJuTtqSm2eTI6gNbC8iosGT
XJEPM8tc/dfZ2sDobLfi0alWFOzWo8vKaLnnAdMHoZ8mDo8CQQDCMx08JVlTW1zl
+4UTEnyyYmIezw5ORfMqPtN1LpQ4ptYnHNMVJPWcpRwBYZfHlPOPtuVwo6gzv82G
QpgQsd4PAkEA0fA8e8R6JbeUR1HxsqWeCnPz3Ahq5Ya5WA6HyJQml9aDVqKDDp2L
3AcqsvFEKJ/T34r31so2yW6hj2yFBnzOZwJBAIqanrgJ1CpJYBGJJd6J6FQNIgjp
MUWuaTJyqsvNFd8lPF2oFgPWYDKQKV/W/tRkvD2LhVCSjf95WsADkbMAsAMCQAHo
wWQOwV2eccbERAJv5yQJMeqKWQ6FTyIx36I/VqqC1Obwy2hSnnb9ybGe6BPGgFLE
HMTjSeRDEUOQm5UXhXkCQQCPlZJqlgksBN/TULHC4RgsXIx+oFylBrkiFamYsuEt
Kn52h41pX7FI5TXcqIDPw+uqAu5OJnwDROdLYY6fvIce
-----END RSA PRIVATE KEY-----
```

J Reflective question. (1p)

In ECC, we use a 256-bit private key. This is used to generate the key for signing Bitcoin transactions. Do you think that a 256-bit key is largest enough? If we use a cracker what performs 1 Tera keys per second, will someone be able to determine our private key?

A 256-bit private key used in elliptic curve cryptography (ECC) is highly secure and offers robust protection against attacks. ECC's security relies on the difficulty of solving the elliptic curve discrete logarithm problem, and the vast key space of 2^256 makes it computationally infeasible for attackers, even with a "cracker" attempting 1 trillion keys per second. In practice, a 256-bit ECC private key is considered secure for securing Bitcoin transactions and other cryptographic systems. However, it's essential to stay updated on cryptographic advancements and security best practices, as online security is continually evolving.