**Experiment 4**

Public Key Encryption

**Name:** Shashank Sarma

**UID:** 2019130053

**Class:** TE COMPS

**Objective:** The key objective of this lab is to provide a practical introduction to public keyencryption, and with a focus on RSA and Elliptic Curve methods. This includes the creation of key pairs and in the signing process. As a part of this objective first you perform section c which is given below.

* **Web link (Weekly activities):** https://asecuritysite.com/esecurity/unit04
* **Video demo:** https://youtu.be/6T9bFA2nl3c
* **RSA Encryption**



**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/465ZX5aH3HWFFX500JSPSt0/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

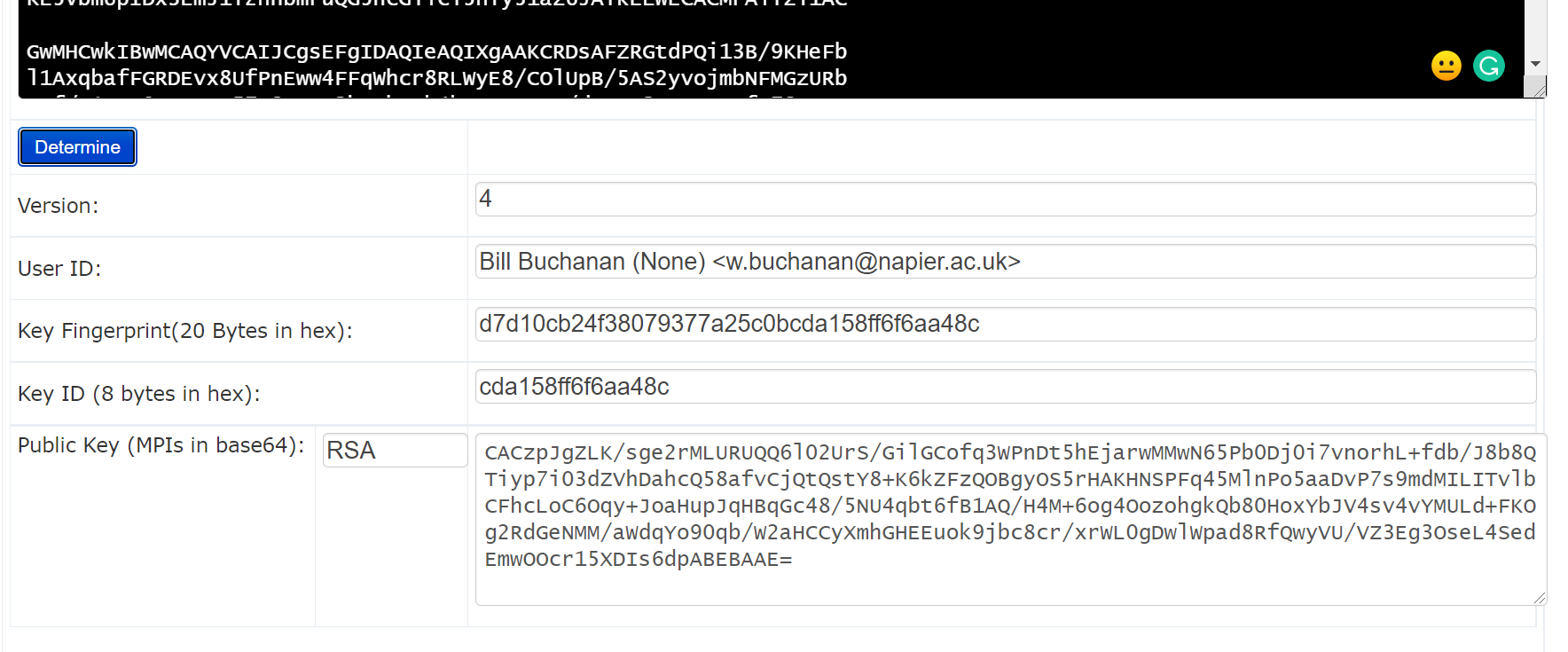
RmyUquF+/zNnSBVgtY1rzwaYi05XfuxG0WHVHPTtRyJ5pF4HSqiuvk6Z/4z3bw==

=ZrP+

-----END PGP PUBLIC KEY BLOCK-----

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

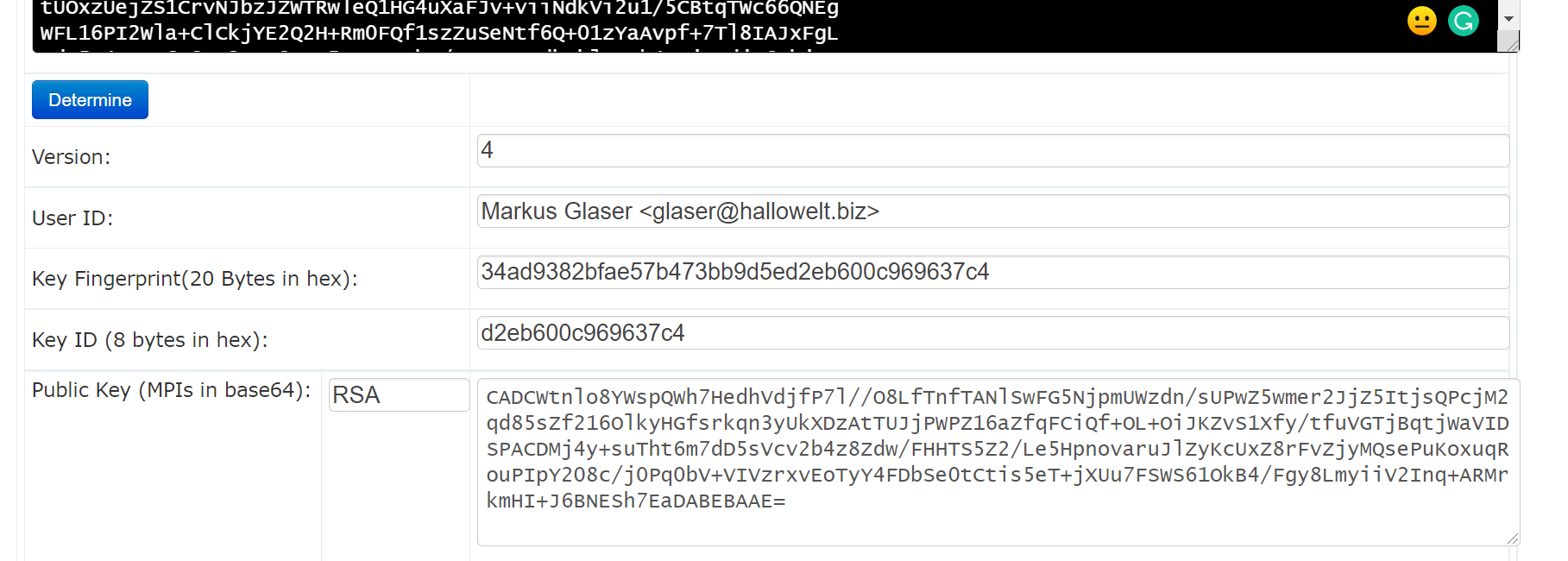
<https://asecuritysite.com/encryption/pgp1>



The owner of the key is **Bill Buchanan**, a Professor in the School of Computing at Edinburgh Napier University.

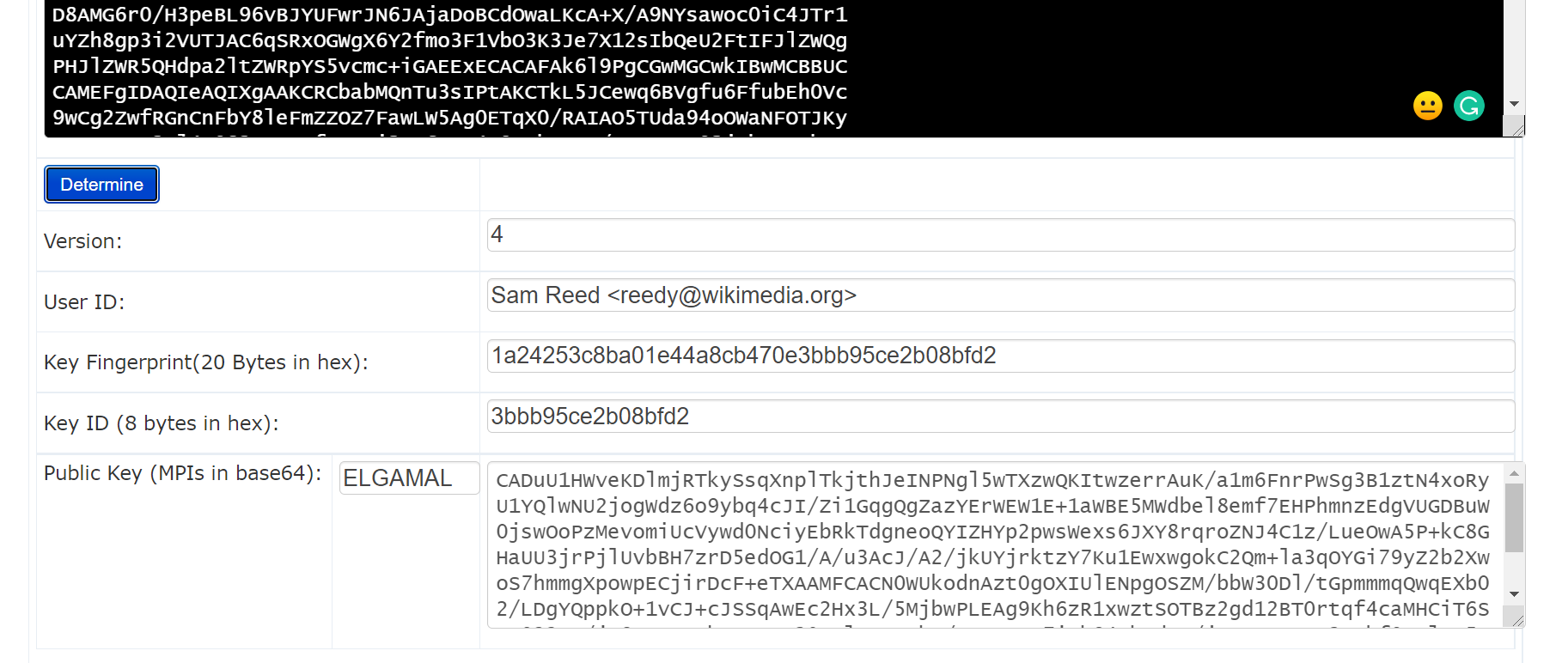
By searching on-line, can you find the public key of three famous people, and view their key details, and can you discover some of the details of their keys (eg User ID, key encryption  method, key size, etc)?

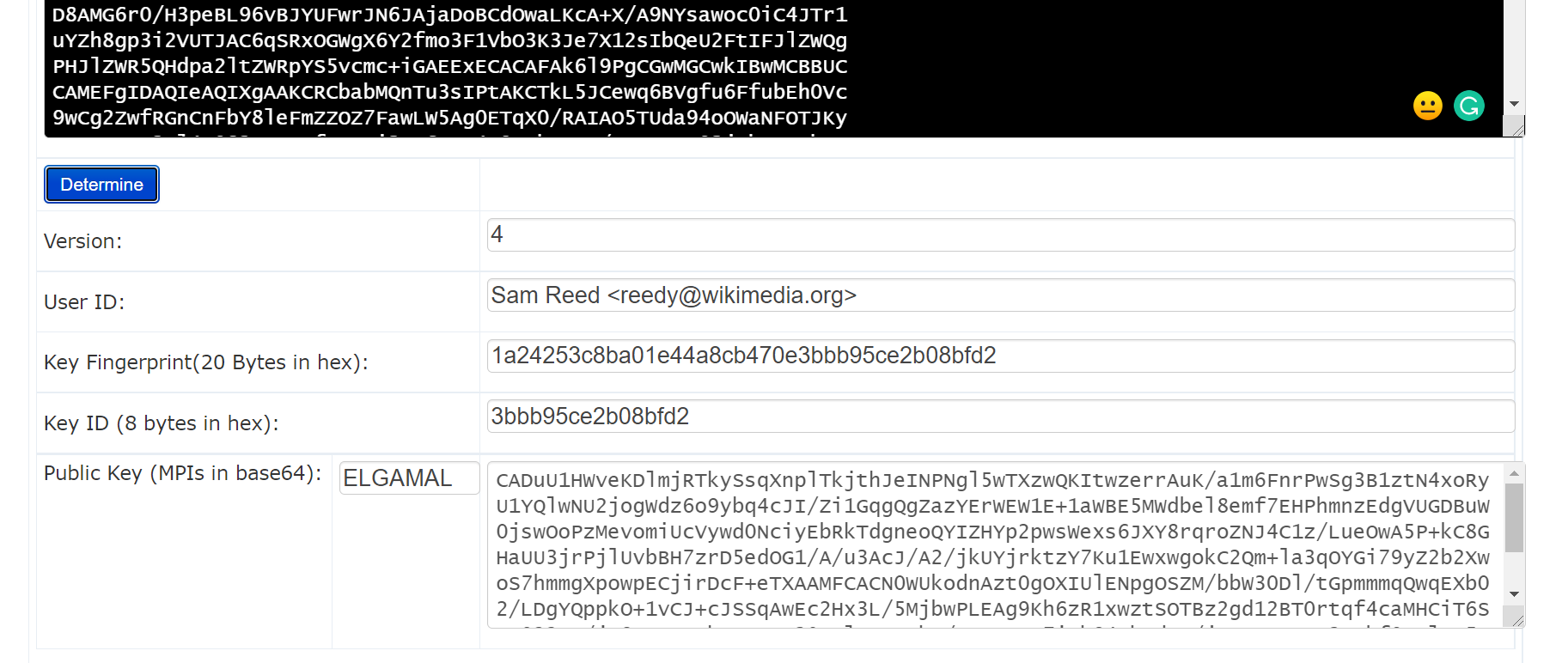
1. **Prof. Markus Glaser.**



**Algorithm:** RSA

1. **Prof Sam Reed**





**Algorithm:** ELGAMAL

1. **Brian Wolff(CEO of Parkers Technology).**



**Algorithm:** RSA

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

ASCII armor is a binary-to-textual encoding converter. ASCII armor is a feature of a type of encryption called pretty good privacy (PGP). ASCII armor involves encasing encrypted messaging in ASCII so that they can be sent in a standard messaging format such as email. The reasoning behind ASCII armor for PGP is that the original PGP format is binary, which is not considered very readable by some of the most common messaging formats. Making the file into American Standard Code for Information Interchange (ASCII) format converts the binary to a printable character representation. Handling file volume can be accomplished through compressing the file.

**A.2** Bob has a private RSA key of:

MIICXAIBAAKBgQCwgjkeoyCXm9v6VBnUi5ihQ2knkdxGDL3GXLIUU43/froeqk7q9mtxT4AnPAaDX3f2r4STZYYiqXGsH

CUBZcI90dvZf6YiEM5OY2jgsmqBjf2Xkp/8HgN/XDw/wD2+zebYGLLYtd2u3GXx9edqJ8kQcU9LaMH+ficFQyfq9UwTjQ

IDAQABAoGAD7L1a6Ess+9b6G70gTANWkKJpshVZDGb63mxKRepaJEX8sRJEqLqOYDNsC+pkKO8IsfHreh4vrp9bsZuECr

B1OHSjwDB0S/fm3KEWbsaaXDUAu0dQg/JBMXAKzeATreoIYJItYgwzrJ++fuquKabAZumvOnWJyBIs2z103kDz2ECQQDn

n3JpHirmgVdf81yBbAJaXBXNIPzOcCth1zwFAs4EvrE35n2HvUQuRhy3ahUKXsKX/bGvWzmC2O6kbLTFEygVAkEAwxXZn

PkaAY2vuoUCN5NbLZgegrAtmU+U2woa5A0fx6uXmShqxo1iDxEC71FbNIgHBg5srsUyDj3OsloLmDVjmQJAIy7qLyOA+s

Cc6BtMavBgLx+bxCwFmsoZHOSX3l79smTRAJ/HY64RREIsLIQ1q/yW7IWBzxQ5WTHgliNZFjKBvQJBAL3t/vCJwRz0Ebs

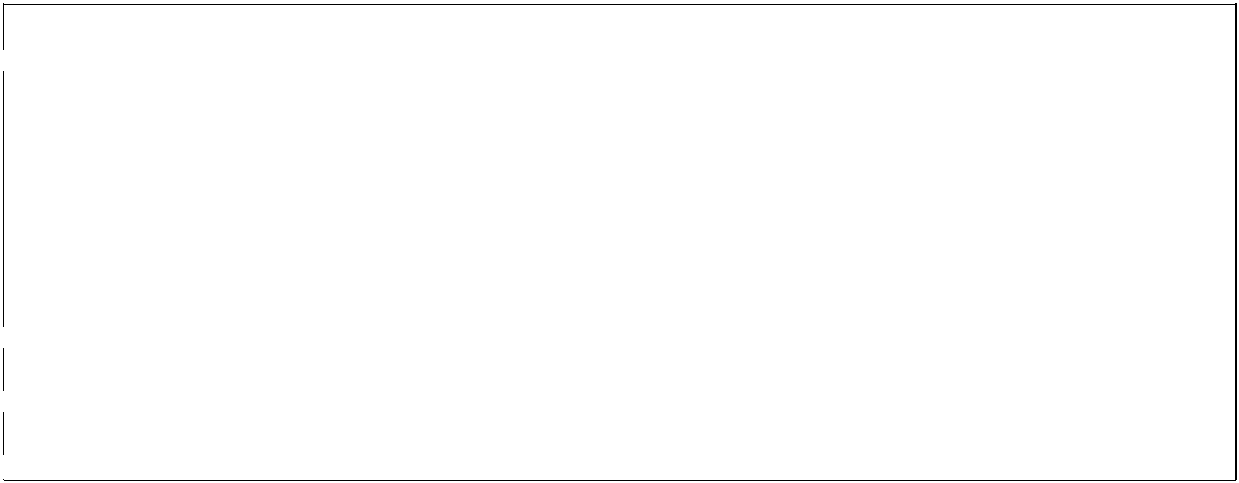
5FaB/8UwhhsrbtXlGdnkOjIGsmV0vHSf6poHqUiay/DV88pvhN11ZG8zHpeUhnaQccJ9ekzkCQDHHG9LYCOqTgsyYms//

cW4sv2nuOE1UezTjUFeqOlsgO+WN96b/M5gnv45/Z3xZxzJ4HOCJ/NRwxNOtEUkw+zY=

And receives a ciphertext message of:

Pob7AQZZSml618nMwTpx3V74N45x/rTimUQeTl0yHq8F0dsekZgOT385Jls1HUzWCx6ZRFPFMJ1RNYR2Yh7AkQtFLVx9l YDfb/Q+SkinBIBX59ER3/fDhrVKxIN4S6h2QmMSRblh4KdVhyY6cOxu+g48Jh7TkQ2Ig93/nCpAnYQ=

Using the following code:



from Crypto.PublicKey import RSA

from Crypto.Util import asn1

from base64 import b64decode

msg="Pob7AQZZSml618nMwTpx3V74N45x/rTimUQeTl0yHq8F0dsekZgOT385Jls1HUzWCx6ZRFPFMJ1RNYR2Yh7AkQtF

LVx9lYDfb/Q+SkinBIBX59ER3/fDhrVKxIN4S6h2QmMSRblh4KdVhyY6cOxu+g48Jh7TkQ2Ig93/nCpAnYQ="

privatekey =

'MIICXAIBAAKBgQCwgjkeoyCXm9v6VBnUi5ihQ2knkdxGDL3GXLIUU43/froeqk7q9mtxT4AnPAaDX3f2r4STZYYiqXGs

HCUBZcI90dvZf6YiEM5OY2jgsmqBjf2Xkp/8HgN/XDw/wD2+zebYGLLYtd2u3GXx9edqJ8kQcU9LaMH+ficFQyfq9UwTj

QIDAQABAoGAD7L1a6Ess+9b6G70gTANWkKJpshVZDGb63mxKRepaJEX8sRJEqLqOYDNsC+pkKO8IsfHreh4vrp9bsZuEC

rB1OHSjwDB0S/fm3KEWbsaaXDUAu0dQg/JBMXAKzeATreoIYJItYgwzrJ++fuquKabAZumvOnWJyBIs2z103kDz2ECQQD

nn3JpHirmgVdf81yBbAJaXBXNIPzOcCth1zwFAs4EvrE35n2HvUQuRhy3ahUKXsKX/bGvWzmC2O6kbLTFEygVAkEAwxXZ

nPkaAY2vuoUCN5NbLZgegrAtmU+U2woa5A0fx6uXmShqxo1iDxEC71FbNIgHBg5srsUyDj3OsloLmDVjmQJAIy7qLyOA+

sCc6BtMavBgLx+bxCwFmsoZHOSX3l79smTRAJ/HY64RREIsLIQ1q/yW7IWBzxQ5WTHgliNZFjKBvQJBAL3t/vCJwRz0Eb

s5FaB/8UwhhsrbtXlGdnkOjIGsmV0vHSf6poHqUiay/DV88pvhN11ZG8zHpeUhnaQccJ9ekzkCQDHHG9LYCOqTgsyYms/

/cW4sv2nuOE1UezTjUFeqOlsgO+WN96b/M5gnv45/Z3xZxzJ4HOCJ/NRwxNOtEUkw+zY='

keyDER = b64decode(privatekey)

keys = RSA.importKey(keyDER)

dmsg = keys.decrypt(b64decode(msg))

print dmsg

a

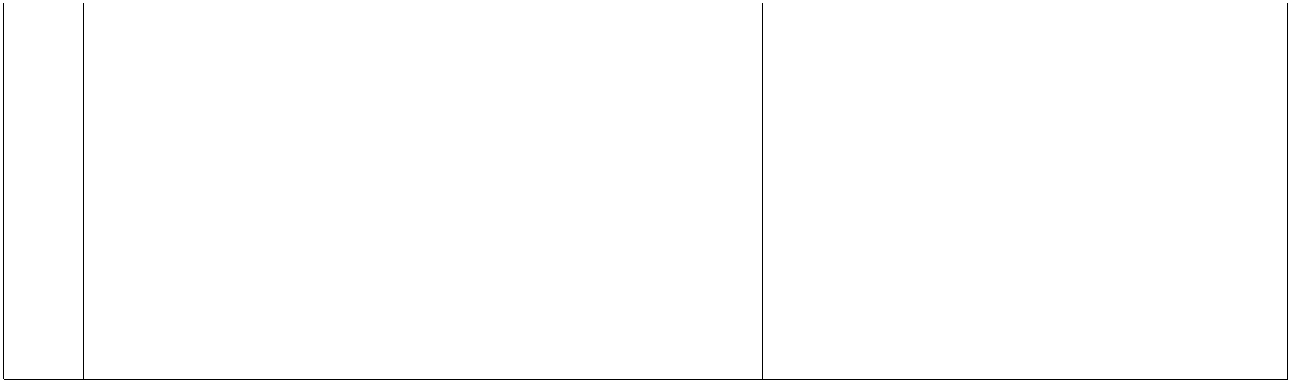
 What is the plaintext message that Bob has been sent?

Raised NotImplemented Error.

* **OpenSSL (RSA)**



We will use OpenSSL to perform the following:

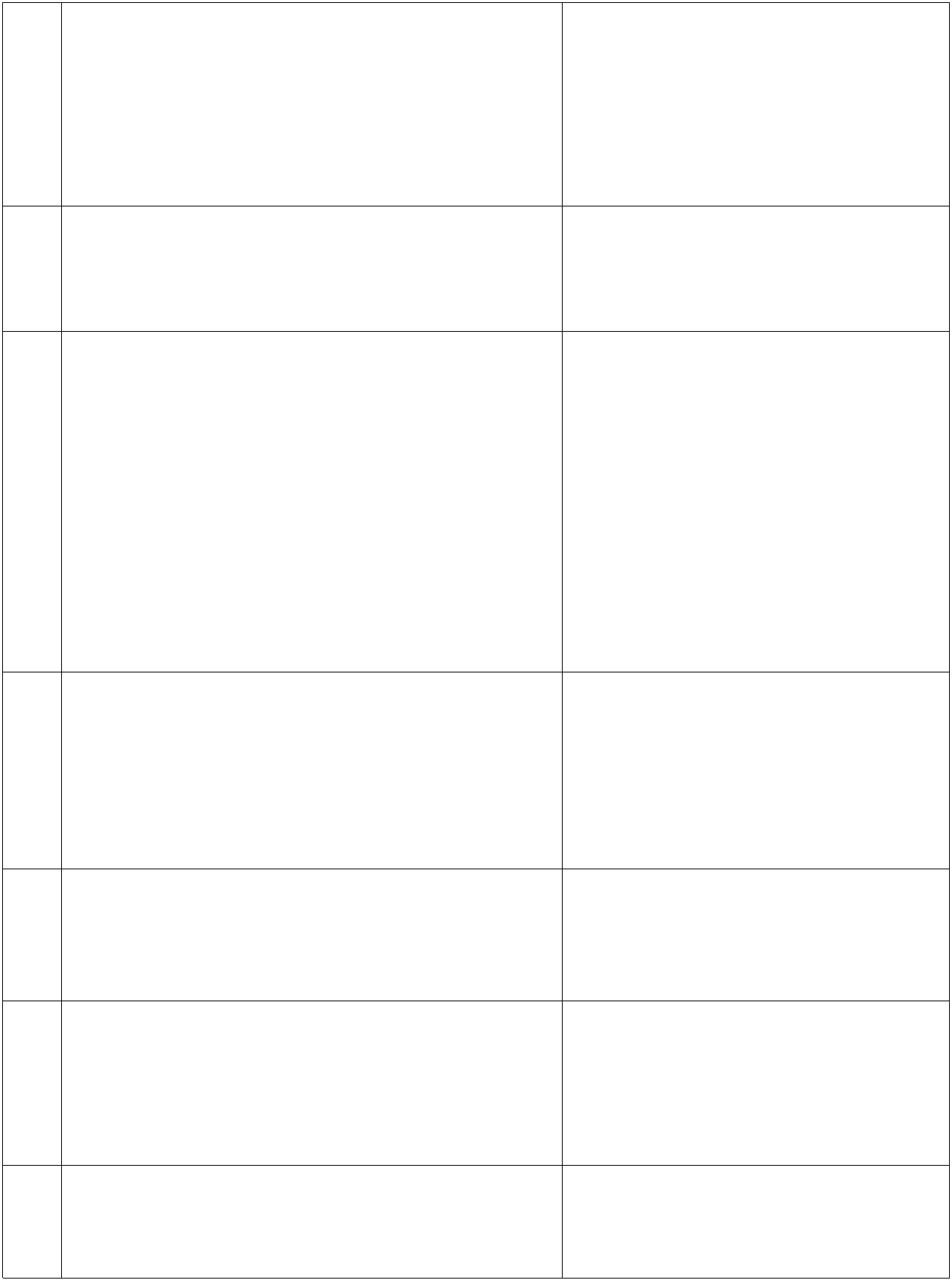


|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **B.1** | First we need to generate a key pair with: | What is the type of public key method |
|  | openssl genrsa -out private.pem 1024 | used: **RSA key generation algorithm** |
|  |  |
|  |  | How long is the default key: **1024 bits** |

This file contains both the public and the private key.

How long did it take to generate a 1,024 bit key? **0.5 seconds**

2



**B.2** Use following command to view the output file:

cat private.pem: Img-1

**B.3** Next we view the RSA key pair:

openssl rsa -in private.pem -text

**B.4** Let’s now secure the encrypted key with 3-DES:

openssl rsa -in private.pem -des3 -out key3des.pem

**B.5** Next we will export the public key:

openssl rsa -in private.pem -out public.pem -outform PEM -pubout

**B.6** Now create a file named “myfile.txt” and put a message into it. Next encrypt it with your public key:

openssl rsautl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin

**B.7** And then decrypt with your private key:

openssl rsautl -decrypt -inkey private.pem -in file.bin -out decrypted.txt

Use the following command to view the keys:

cat private.pem :

What can be observed at the start and end of the file:

**----BEGIN RSA PRIVATE KEY----**

**----END RSA PRIVATE KEY----**

Which are the attributes of the key shown:

* **Modulus**
* **Public Exponent**
* **Private Exponent**
* **Prime1**
* **Prime2**
* **Exponent1**
* **Exponent2**
* **Coefficient**

Which number format is used to display the information on the attributes: **Hexadecimal**

Why should you have a password on the usage of your private key?

**Using a passphrase on the private key adds another layer of security to the key. Otherwise, whoever steals the file from us has access to everything we have access to.**

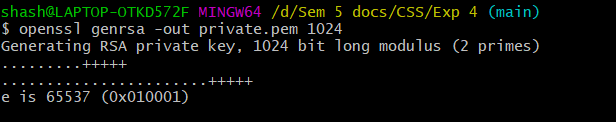
View the output key.

What does the header and footer of the file identify?

**It represents that the key stored in this file is the public key.**

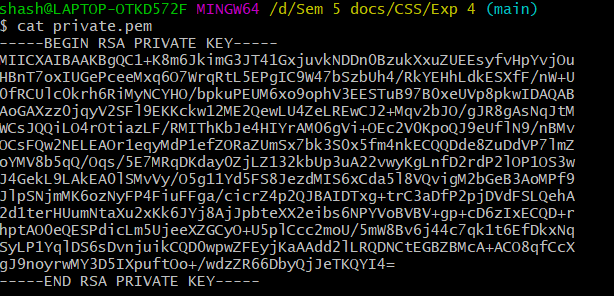
What are the contents of decrypted.txt

B.1)



* **RSA key generation algorithm**
* **1024 bits**
* **0.5 seconds**

B.2)



**-----BEGIN RSA PRIVATE KEY-----**

**MIICXAIBAAKBgQC1+K8m6JkimG3JT41GxjuvkNDDn0BzukXxuZUEEsyfvHpYvjOu**

**HBnT7oxIUGePceeMxq6O7WrqRtL5EPgIC9W47bSzbUh4/RkYEHhLdkESXfF/nW+U**

**0fRCUlc0krh6RiMyNCYHO/bpkuPEUM6xo9ophV3EESTuB97B0xeUVp8pkwIDAQAB**

**AoGAXzz0jqyV2SFl9EKKckw12ME2QewLU4ZeLREwCJ2+Mqv2bJO/gJR8gAsNqJtM**

**WCsJQQiLO4rOtiazLF/RMIThKbJe4HIYrAM06gVi+OEc2V0KpoQJ9eUflN9/nBMv**

**OCsFQw2NELEAOr1eqyMdP1efZORaZUmSx7bk3S0x5fm4nkECQQDde8ZuDdVP7lmZ**

**oYMV8b5qQ/Oqs/5E7MRqDKday0ZjLZ132kbUp3uA22vwyKgLnfD2rdP2lOP1OS3w**

**J4GekL9LAkEA0lSMvVy/O5g11Yd5FS8JezdMIS6xCda5l8VQvigM2bGeB3AoMPf9**

**JlpSNjmMK6ozNyFP4FiuFFga/cicrZ4p2QJBAIDTxg+trC3aDfP2pjDVdFSLQehA**

**2d1terHUumNtaXu2xKk6JYj8AjJpbteXX2eibs6NPYVoBVBV+gp+cD6zIxECQD+r**

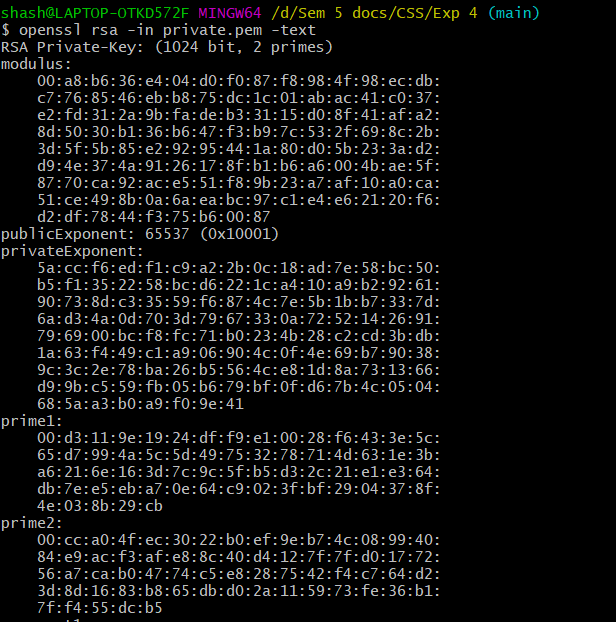
**hptAO0eQESPdicLm5UjeeXZGCyO+U5plCcc2moU/5mW8Bv6j44c7qk1t6EfDkxNq**

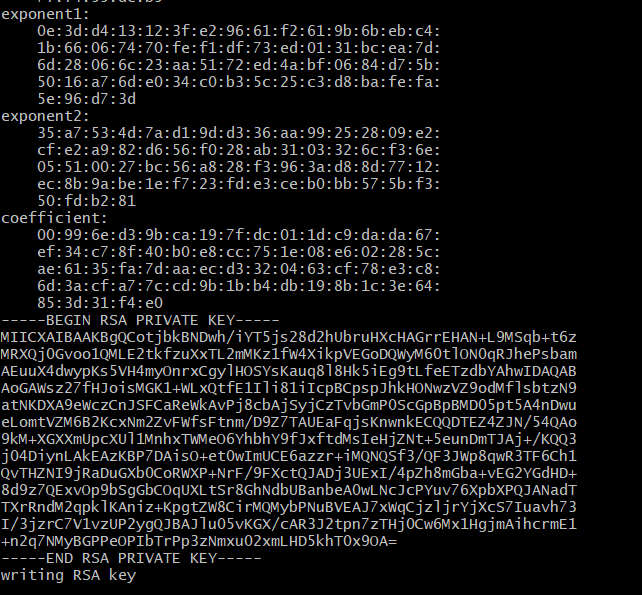
**SyLP1YqlDS6sDvnjuikCQD0wpwZFEyjKaAAdd2lLRQDNCtEGBZBMcA+ACO8qfCcX**

**gJ9noyrwMY3D5IXpuftOo+/wdzZR66DbyQjJeTKQYI4=**

**-----END RSA PRIVATE KEY-----**

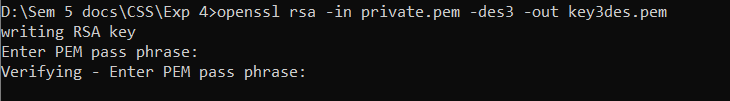
B.3)

\



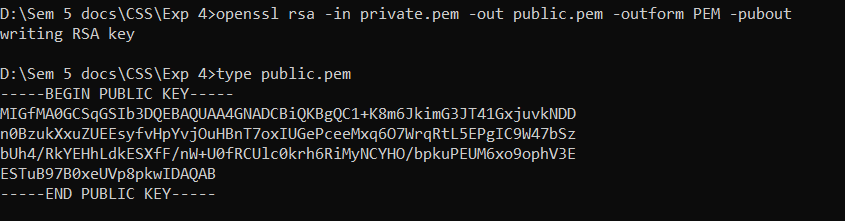
* **Modulus**
* **Public Exponent**
* **Private Exponent**
* **Prime1**
* **Prime2**
* **Exponent1**
* **Exponent2**
* **Coefficient**
* **Hexadecimal**

B.4) secure the encrypted key with 3-DES



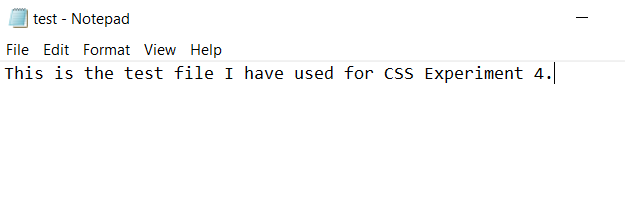
* **Adding password protection makes the key file or anything that we have access to becomes inaccessible to the attacker by adding security layer**

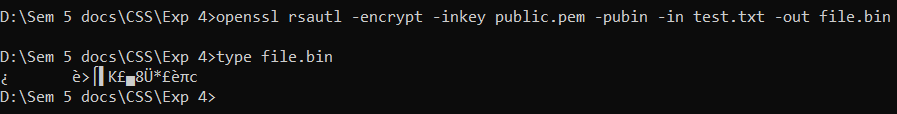
B.5) export the public key:



* **It represents that the key stored in this file is the public key.**

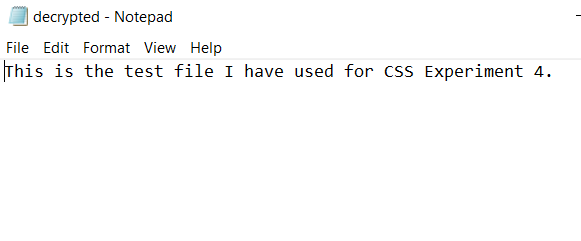
B.6)





B.7)





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

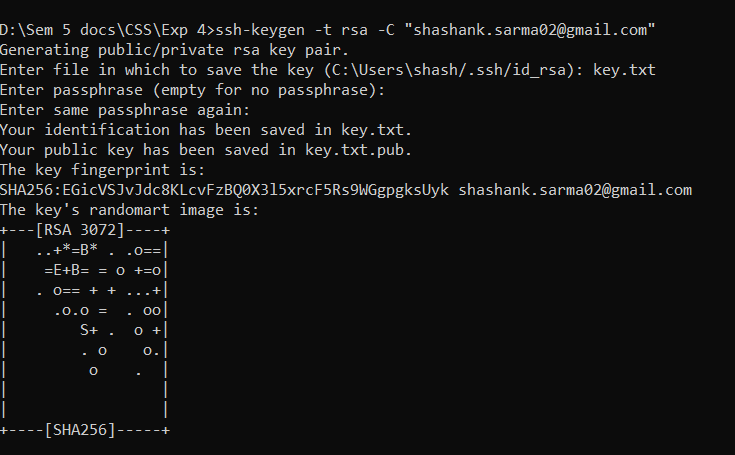
AAAAB3NzaC1yc2EAAAADAQABAAABAQDLrriuNYTyWuC1IW7H6yea3hMV+rm029m2f6IddtlImHrOXjNwYyt4Elkkc7AzO

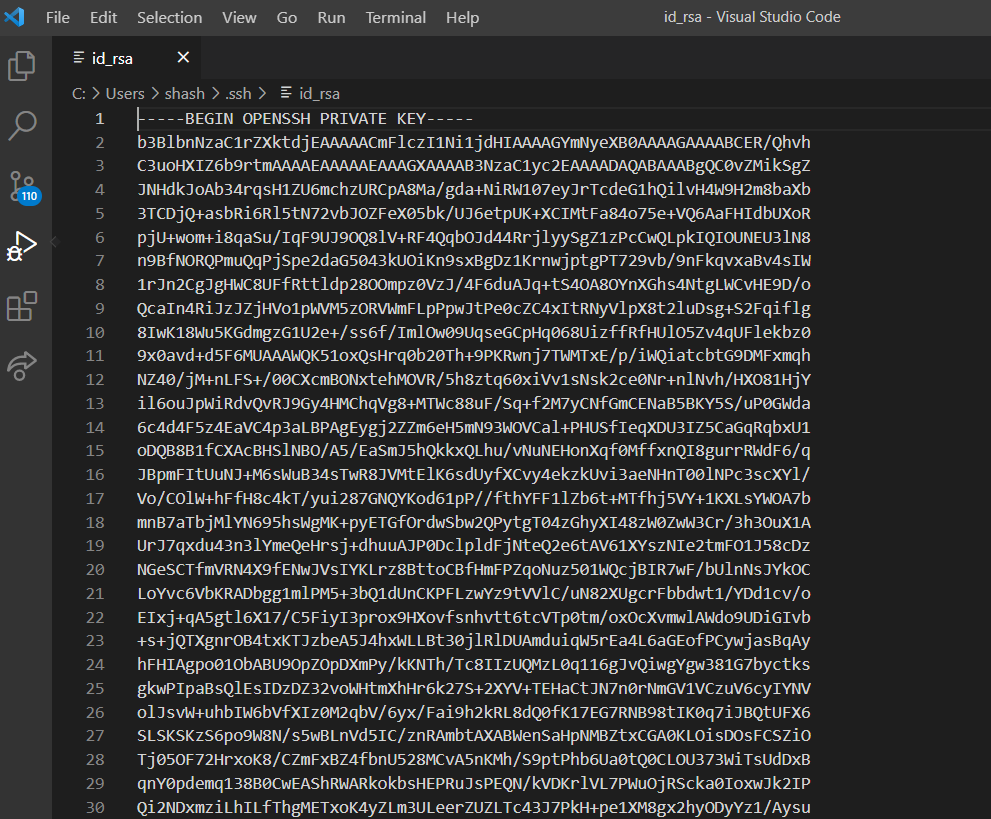
y899C3gpx0kJK45k/CLbPnrHvkLvtQ0AbzWEQpOKxI+tW06PcqJNmTB8ITRLqIFQ++ZanjHWMw2Odew/514y1dQ8dccCO

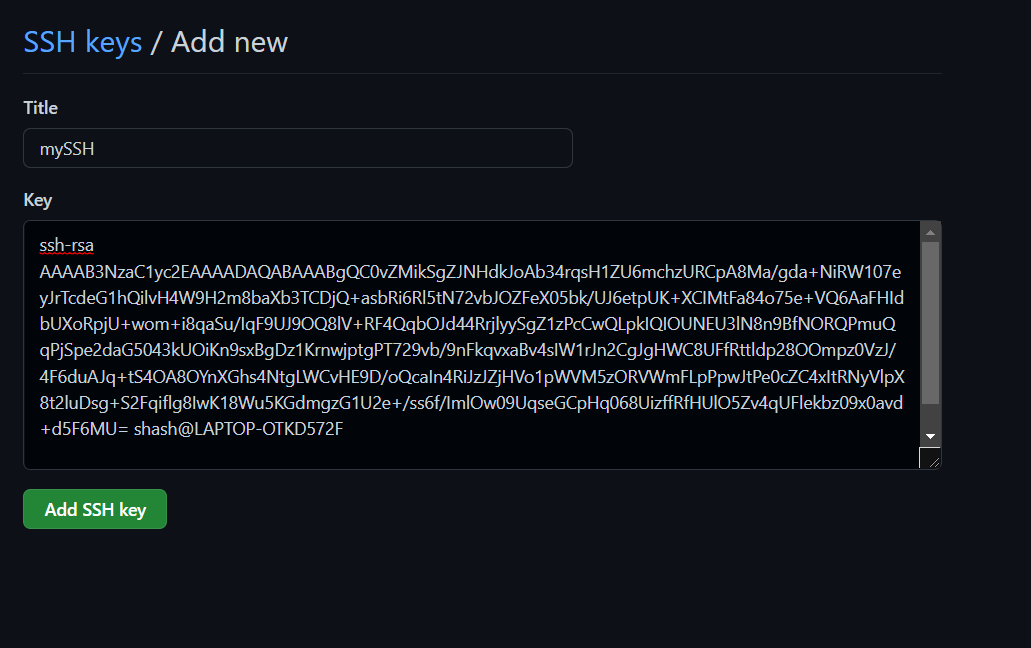
uzeGhL2Lq9dtfhSxx+1cBLcyoSh/lQcs1HpXtpwU8JMxWJl409RQOVn3gOusp/P/0R8mz/RWkmsFsyDRLgQK+xtQxbpbo

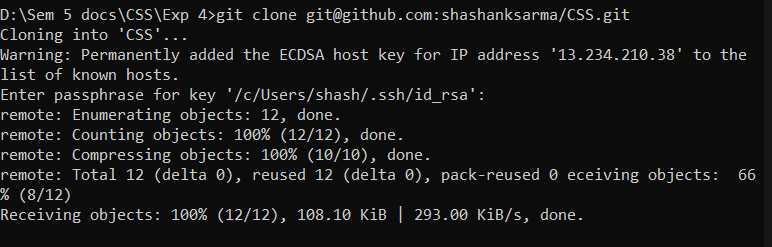
dpnz5lIOPWn5LnT0si7eHmL3WikTyg+QLZ3D3m44NCeNb+bOJbfaQ2ZB+lv8C3OxylxSp2sxzPZMbrZWqGSLPjgDiFIBL

[w.buchanan@napier.ac.uk](mailto:w.buchanan@napier.ac.uk)









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 fileto 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.

* **OpenSSL (ECC)**



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



|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **C.1** | First we need to generate a private key with: | Can you view your key?  **Yes, using the cat command** |

openssl ecparam -name secp256k1 -genkey -out priv.pem

The file will only contain the private key (and should have 256 bits).

Now use “cat priv.pem” to 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): **fc:2f**

A: **0**

B: **7**

Generator (last two bytes): **d4:b8**

Order (last two bytes): **41:41**



**C.3** Now generate your public key based on your private key with:

How many bits and bytes does your private key have: **32 bytes**

openssl ec -in priv.pem -text -noout How many bit and bytes does your

public key have (Note the 04 is not

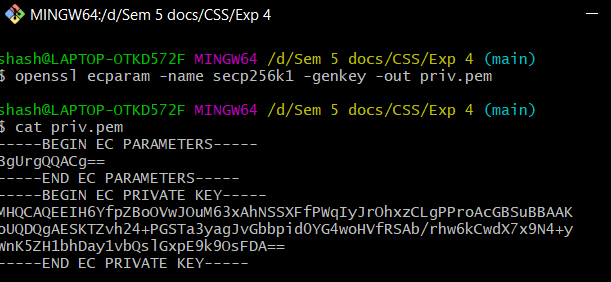
part of the elliptic curve point): **64 bytes**

What is the ECC method that you

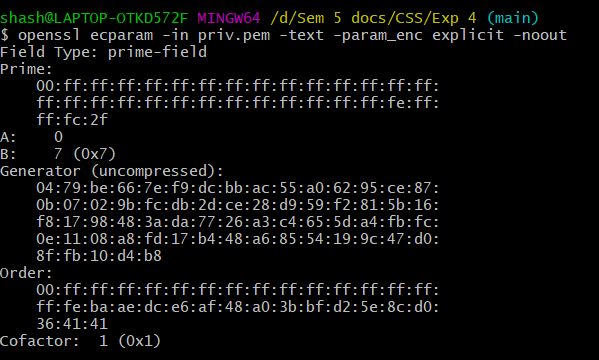
have used?: secp256k1. This is the elliptic curve used by Bitcoin, Ethereum, and many other cryptocurrencies. The equation for the **secp256k1** curve is y² = x³+7

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

C1. generate a private key



C2. We can view the details of the ECC parameters used with:



Prime (last two bytes): **fc:2f**

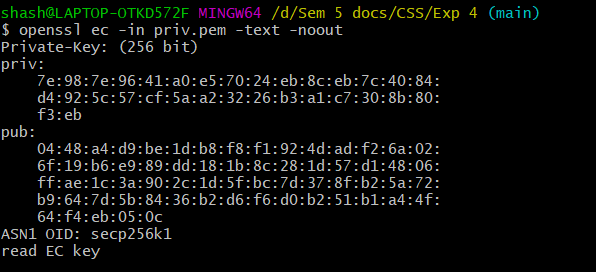
A: **0**

B: **7**

Generator (last two bytes): **d4:b8**

Order (last two bytes): **41:41**

C3. generate your public key based on your private key



How many bits and bytes does your private key have: **32 bytes**

How many bit and bytes does your public key have (Note the 04 is not part of the elliptic curve point): **64 bytes**

What is the ECC method that you have used?

secp256k1. This is the elliptic curve used by Bitcoin, Ethereum, and many other cryptocurrencies. The equation for the **secp256k1** curve is y² = x³+7

* **Elliptic Curve Encryption**



**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/encryption/elc

Code used:



import OpenSSL

import pyelliptic

secretkey="password"

test="Test123"

alice = pyelliptic.ECC()

bob = pyelliptic.ECC()

print "++++Keys++++"

print "Bob's private key: "+bob.get\_privkey().encode('hex')

 print "Bob's public key: "+bob.get\_pubkey().encode('hex')

print

print "Alice's private key: "+alice.get\_privkey().encode('hex')

 print "Alice's public key: "+alice.get\_pubkey().encode('hex')

 ciphertext = alice.encrypt(test, bob.get\_pubkey())

 print "\n++++Encryption++++"

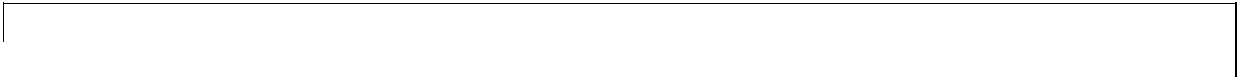
 print "Cipher: "+ciphertext.encode('hex')

 print "Decrypt: "+bob.decrypt(ciphertext)

signature = bob.sign("Alice")

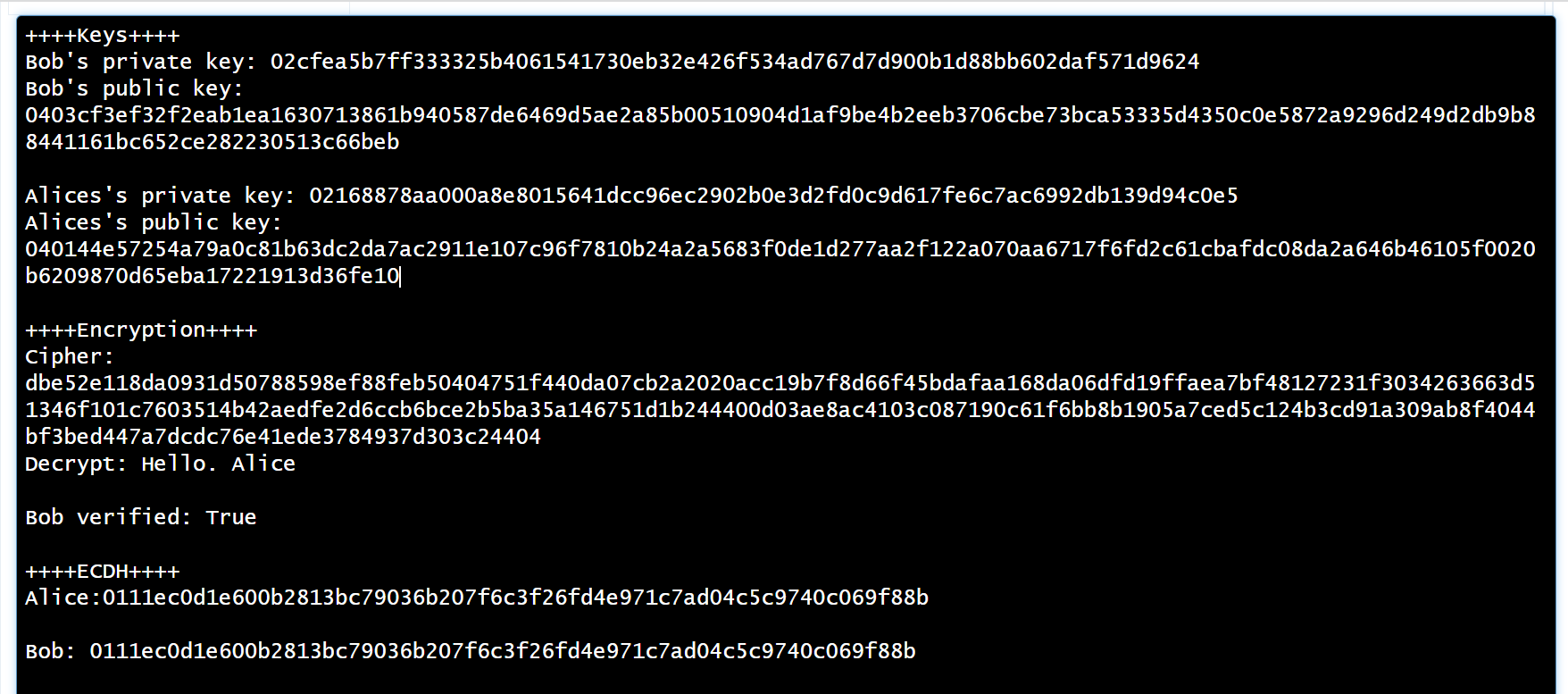
print

print "Bob verified: "+ str(pyelliptic.ECC(pubkey=bob.get\_pubkey()).verify  (signature, "Alice"))



For a message of “Hello. Alice”, what is the ciphertext sent (just include the first four  characters):

**dbe5**



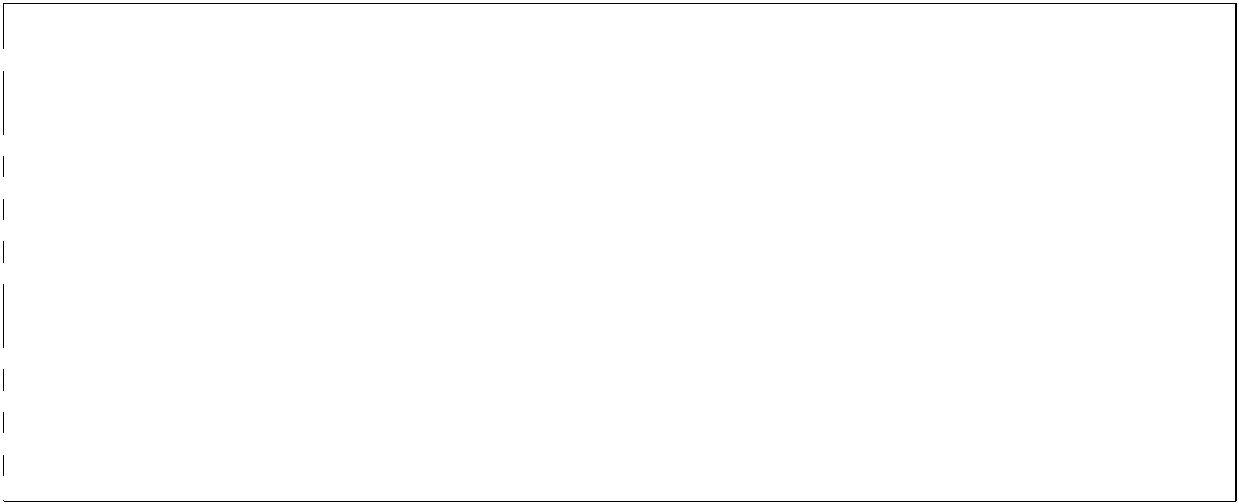
**D.2** Let’s say we create an elliptic curve with *y*2 = *x*3 + 7, and with a prime number of 89, generate the first five (*x*,*y*) points for the finite field elliptic curve. You can use the Python code at the following to generate them:

https://asecuritysite.com/encryption/ecc\_points

First five points: (14, 9) (15, 0) (16, 3) (17, 5) (22,8)



**D.3** Elliptic curve methods are often used to sign messages, and where Bob will sign a message with his private key, and where Alice can prove that he has signed it by using his public key. With ECC, we can use ECDSA, and which was used in the first version of Bitcoin. Enter the following code:



from ecdsa import SigningKey,NIST192p,NIST224p,NIST256p,NIST384p,NIST521p,SECP256k1 import base64

 import sys

msg="Hello"

type = 1

 cur=NIST192p

 sk = SigningKey.generate(curve=cur)

 vk = sk.get\_verifying\_key()

signature = sk.sign(msg)

print "Message:\t",msg

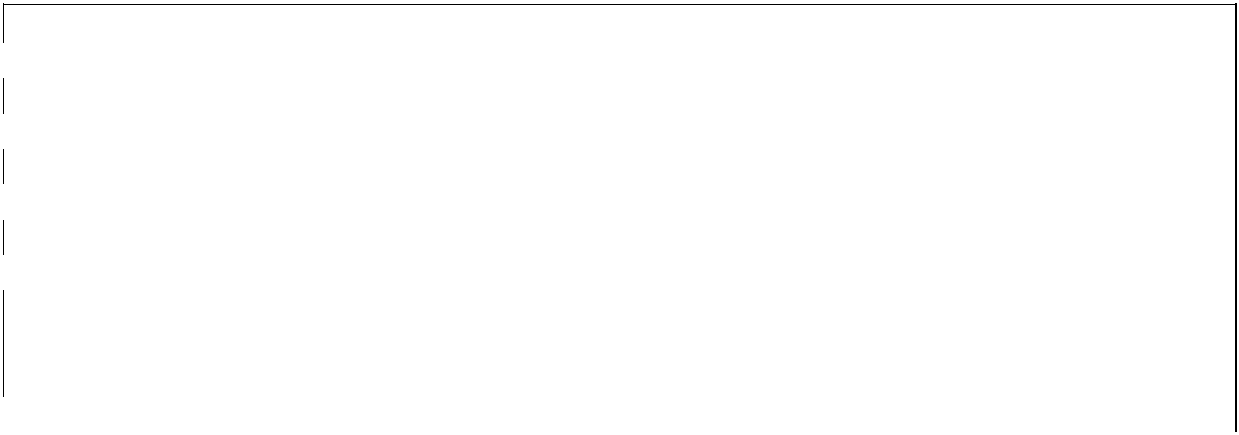
print "Type:\t\t",cur.name

 print "========================="

 print "Signature:\t",base64.b64encode(signature)

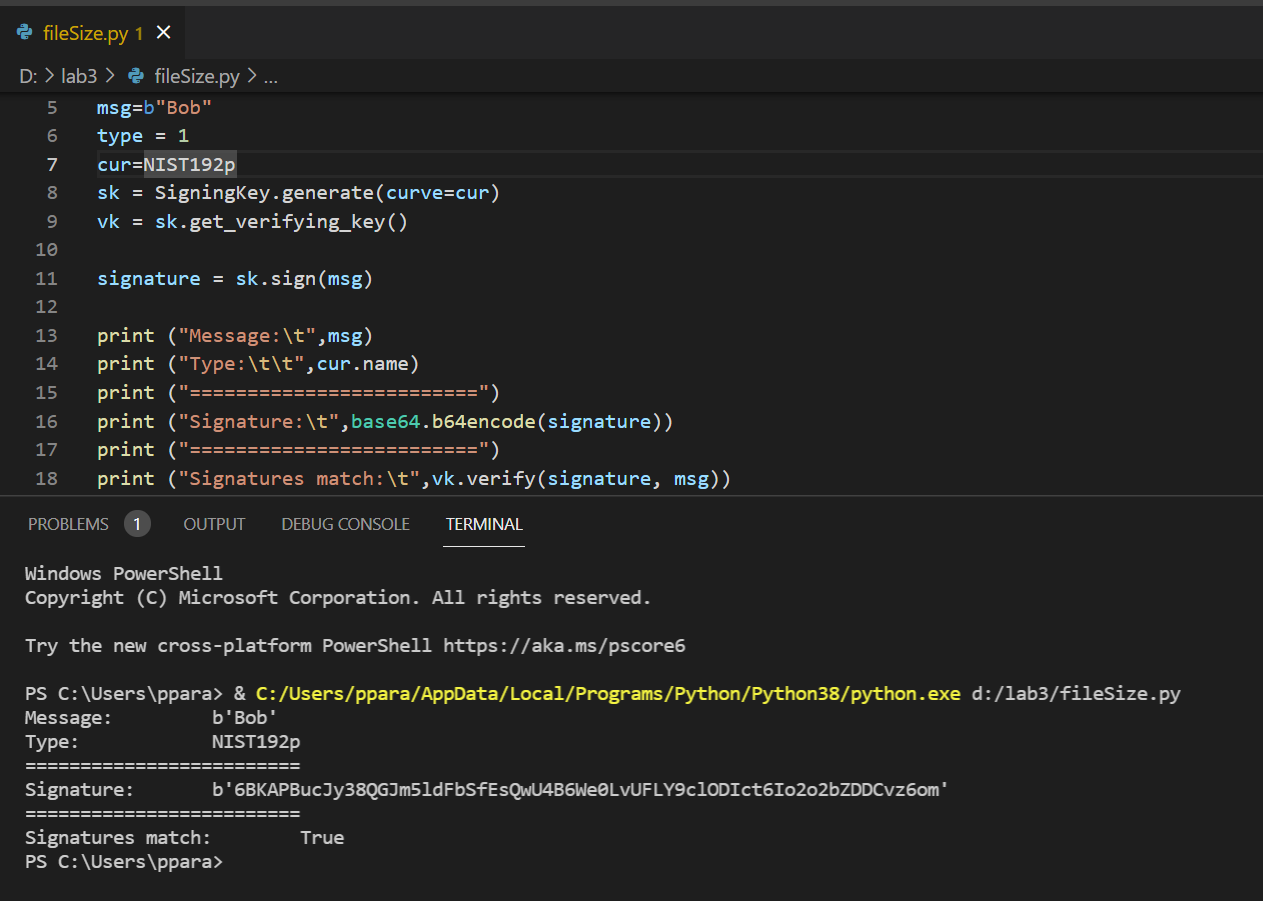
 print "========================="

 print "Signatures match:\t",vk.verify(signature, msg)

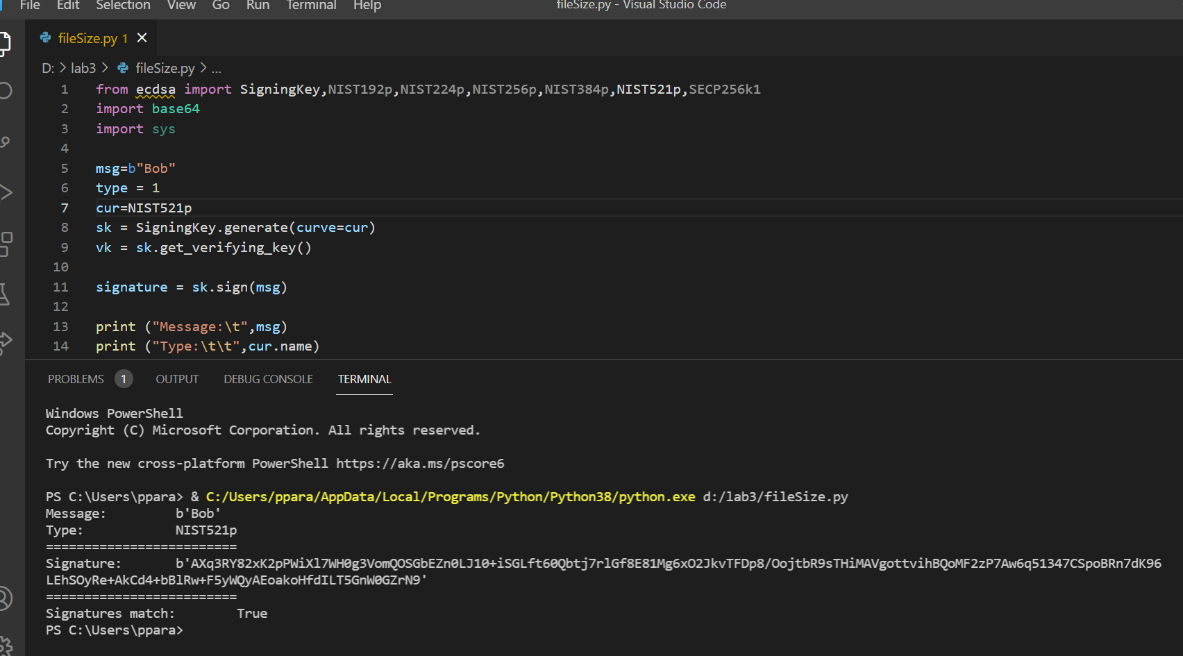


What are the signatures (you only need to note the first four characters) for a message of  “Bob”, for the curves of NIST192p, NIST521p and SECP256k1:

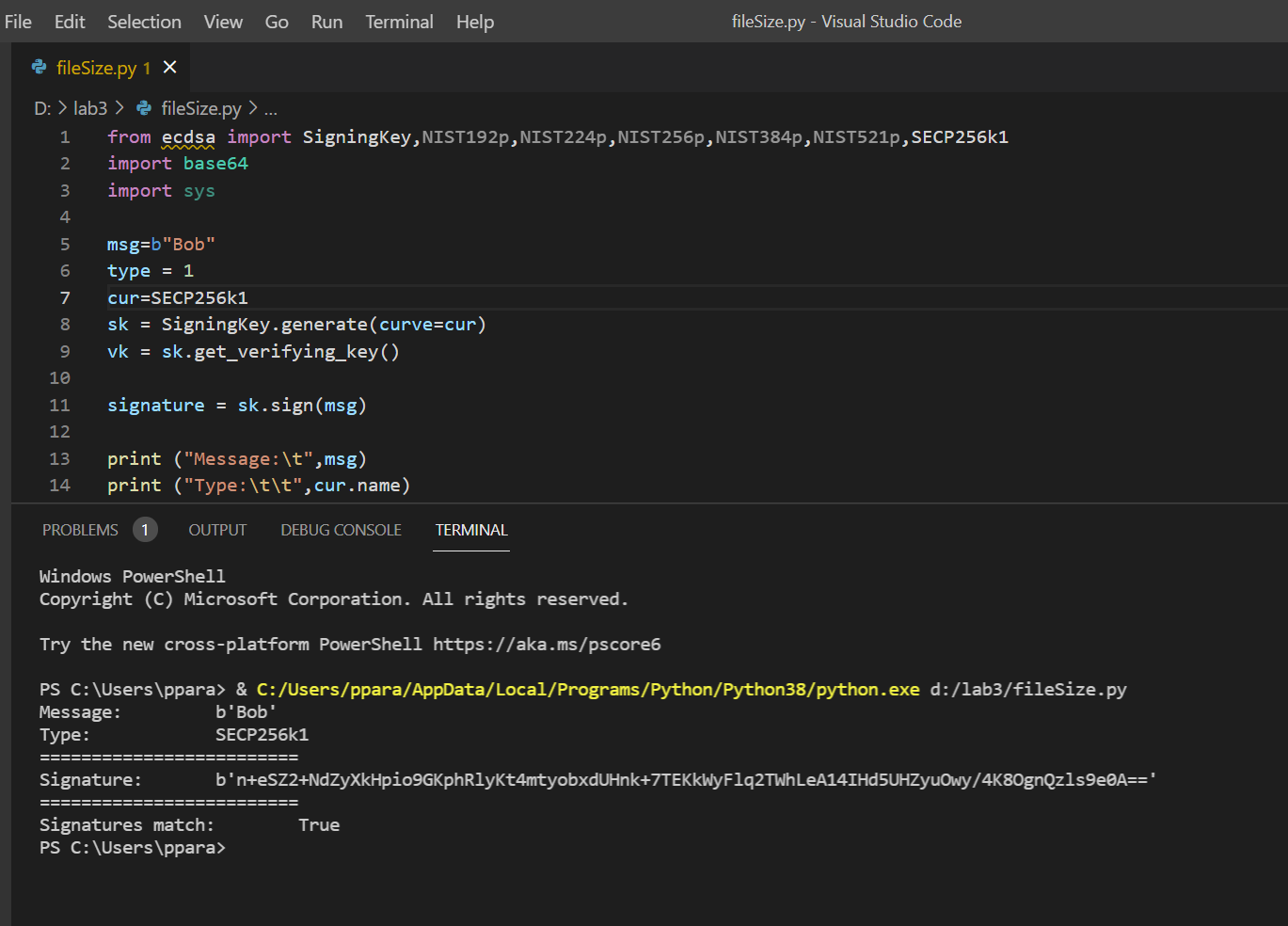
NIST192p: **6BKA**



NIST521p: **AXq3R**



 SECP256k1: **n+eS**



By searching on the Internet, can you find in which application areas that SECP256k1 is used?

**SECP256k1, which has been used by Bitcoin since ECDSA, has some attractive qualities, such as a structure that 'allows for extremely efficient calculation' and'significantly decreases the probability that the curve's inventor incorporated any form of backdoor into the curve'. Because of its appealing qualities, the curve has been integrated into EC-based encryption methods and can be used as an anonymous key agreement mechanism in the elliptic curve Diffie-Hellman. The RLPx transport protocol in Ethereum employs the elliptic curve integrated encryption technique implemented with the SECP256k1 curve.**

* **RSA**



**E.1** 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= 2270113289

q= 6731427277

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



 N= 15281102515454784053

PHI = 15281102506453243488

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*=9168661503871946093

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



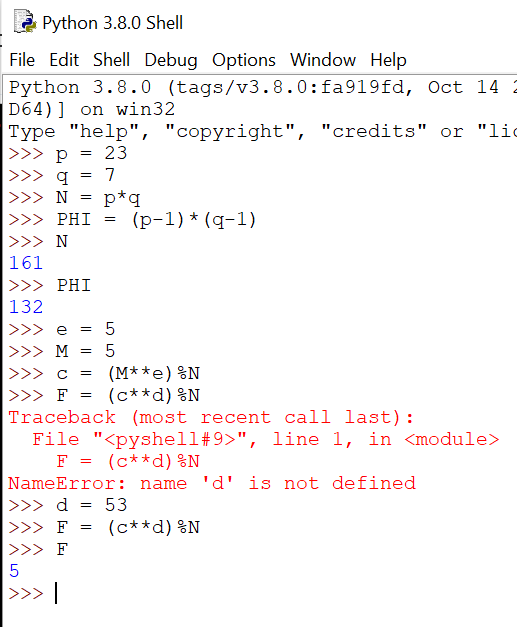
 C = M*e* (mod N) =3125

Now decrypt your ciphertext with:



 M = Cd (mod 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:

p=2270113289

q=6731427277

N=p\*q

PHI=(p-1)\*(q-1)

e=5

for d in range(1,100):

    if ((e\*d % PHI)==1): break

print(e,N)

print(d,N)

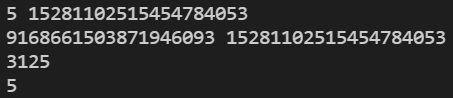
M=5

cipher = (M\*\*e) % N

print(cipher)

message = (cipher\*\*d) % N

print(message)

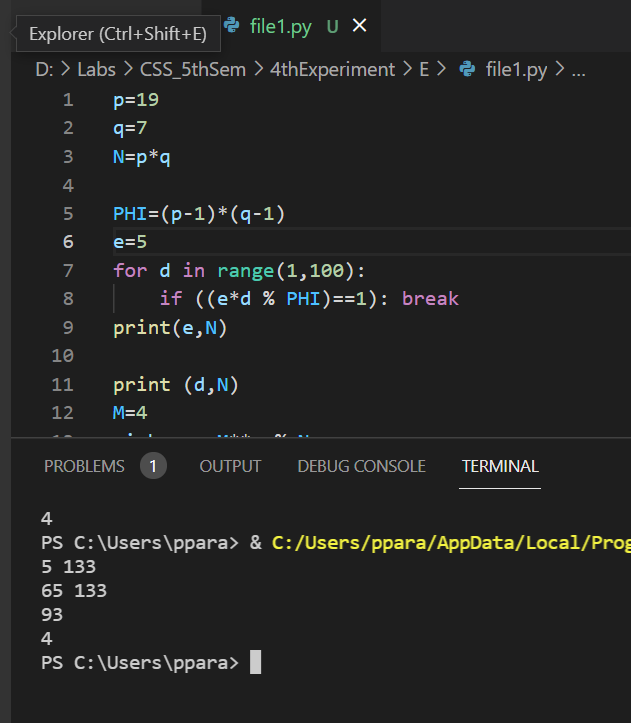


Select three more examples with different values of p and q, and then select e in order to  make sure that the cipher will work:

P = 19

Q = 7

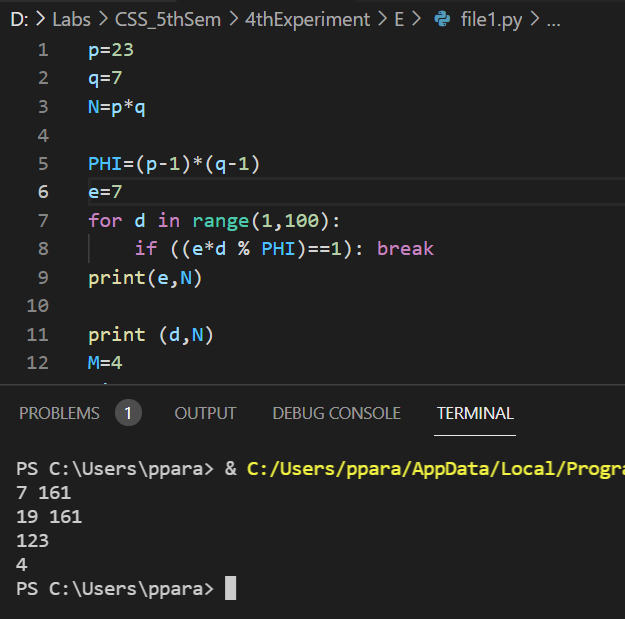
E = 5



P = 23

Q = 7

E = 7

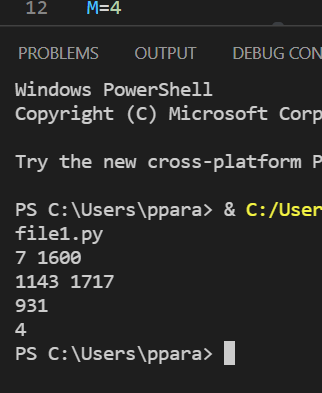


3)

P = 101

Q = 17

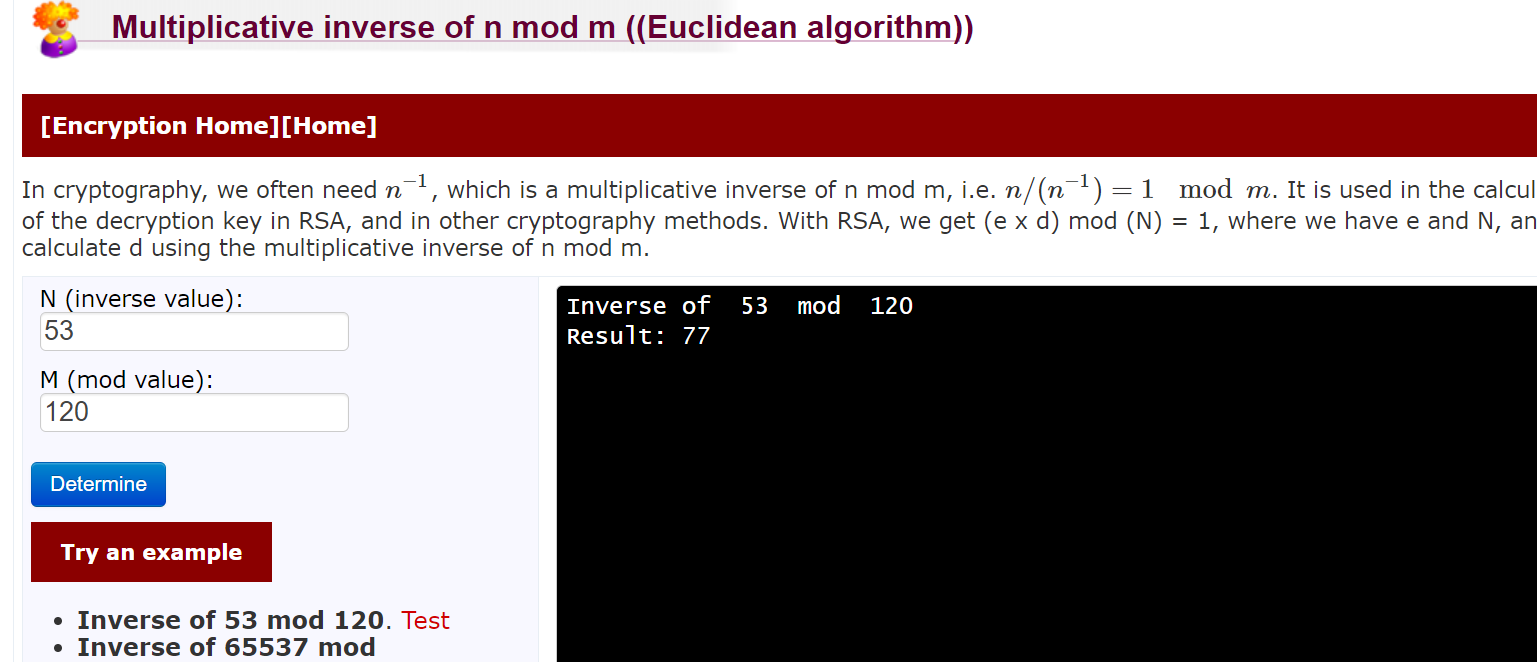
E = 7

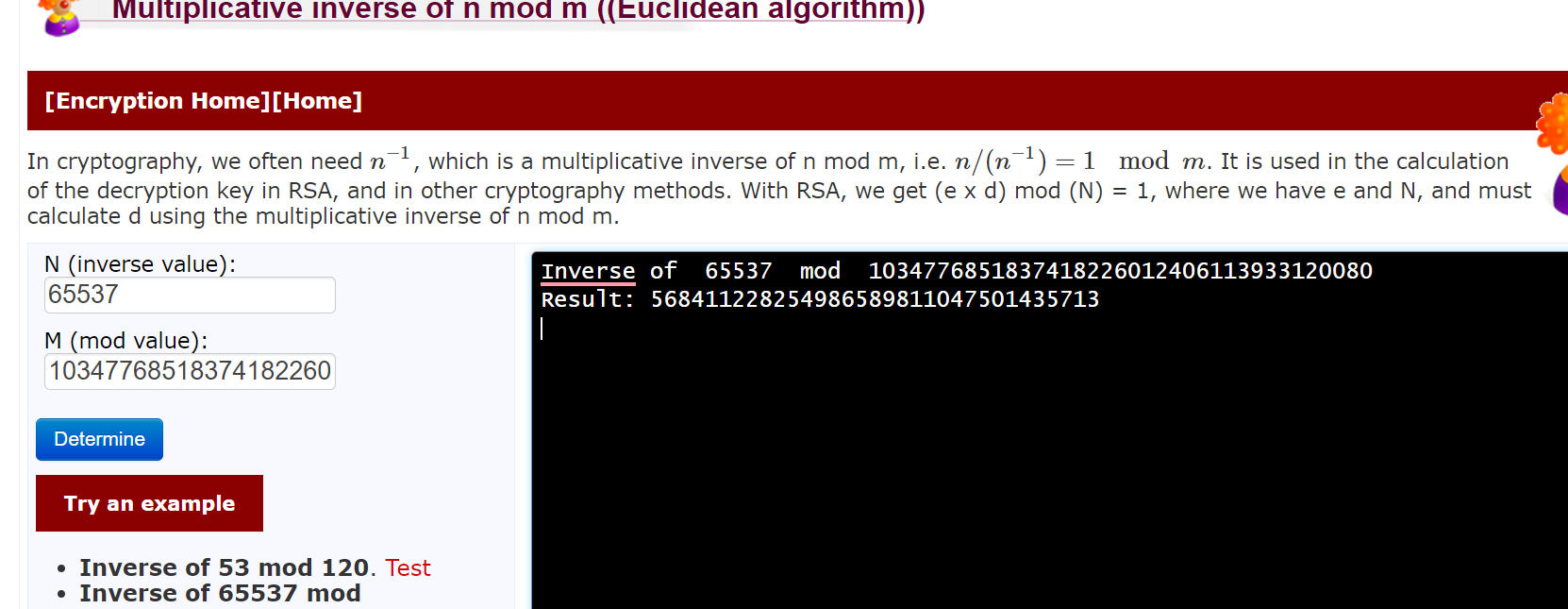


**E.2** In the RSA method, we have a value of e, and then determine d from (d.e) (mod PHI)=1. But how do we use code to determine d? Well we can use the Euclidean algorithm. The code for this is given at:

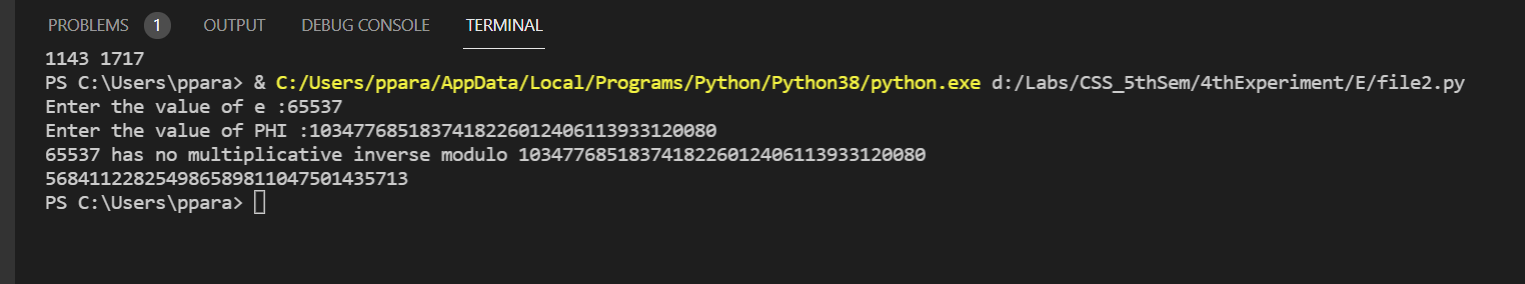
https://asecuritysite.com/encryption/inversemod

Using the code, can you determine the following:





**Inverse of 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)?

def extended\_euclidean\_algorithm(a, b):

    """

    Returns a three-tuple (gcd, x, y) such that

    a \* x + b \* y == gcd, where gcd is the greatest

    common divisor of a and b.

    This function implements the extended Euclidean

    algorithm and runs in O(log b) in the worst case.

    """

    s, old\_s = 0, 1

    t, old\_t = 1, 0

    r, old\_r = b, a

    while r != 0:

        quotient = old\_r // r

        old\_r, r = r, old\_r - quotient \* r

        old\_s, s = s, old\_s - quotient \* s

        old\_t, t = t, old\_t - quotient \* t

    return old\_r, old\_s, old\_t

def inverse\_of(n, p):

    """

    Returns the multiplicative inverse of

    n modulo p.

    This function returns an integer m such that

    (n \* m) % p == 1.

    """

    gcd, x, y = extended\_euclidean\_algorithm(n, p)

    assert (n \* x + p \* y) % p == gcd

    if gcd != 1:

        # Either n is 0, or p is not a prime number.

        raise ValueError(

            '{} has no multiplicative inverse '

            'modulo {}'.format(n, p))

    else:

        return x % p

p=int(input("Enter the value of p :"))

q=int(input("Enter the value of q :"))

e=int(input("Enter the value of e :"))

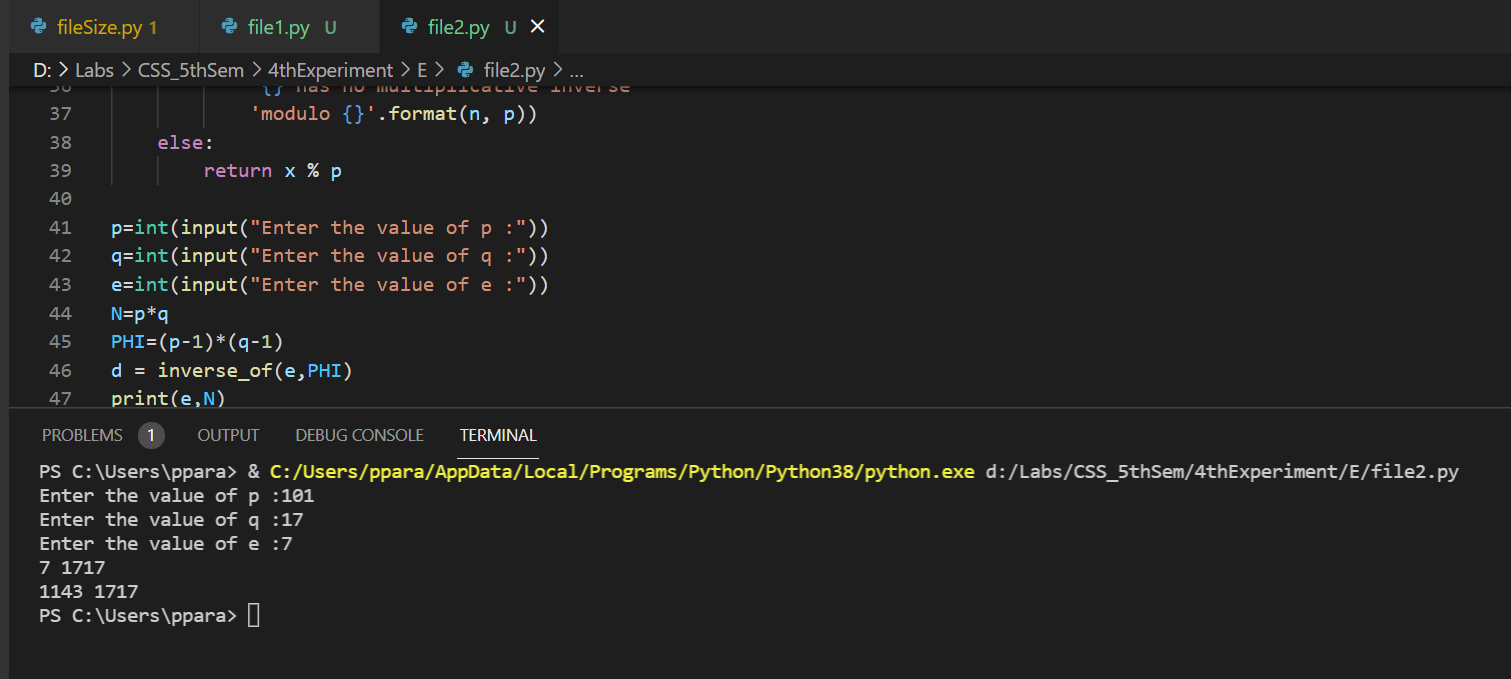
N=p\*q

PHI=(p-1)\*(q-1)

d = inverse\_of(e,PHI)

print(e,N)

print(d,N)



**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)

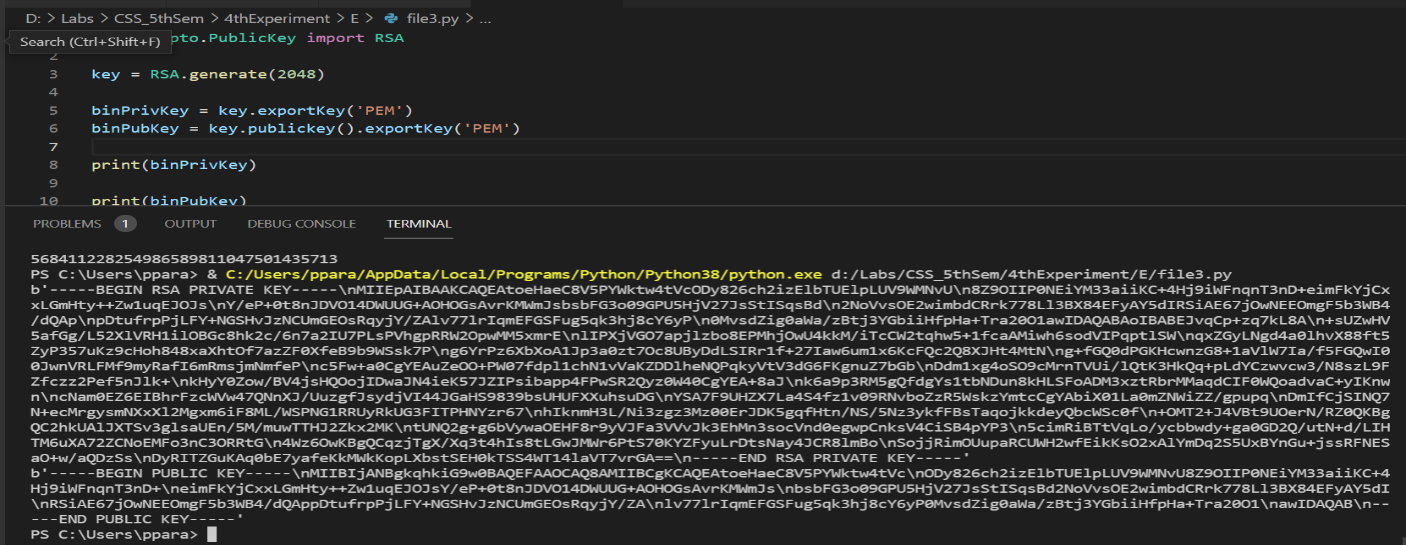
binPrivKey = key.exportKey('PEM')

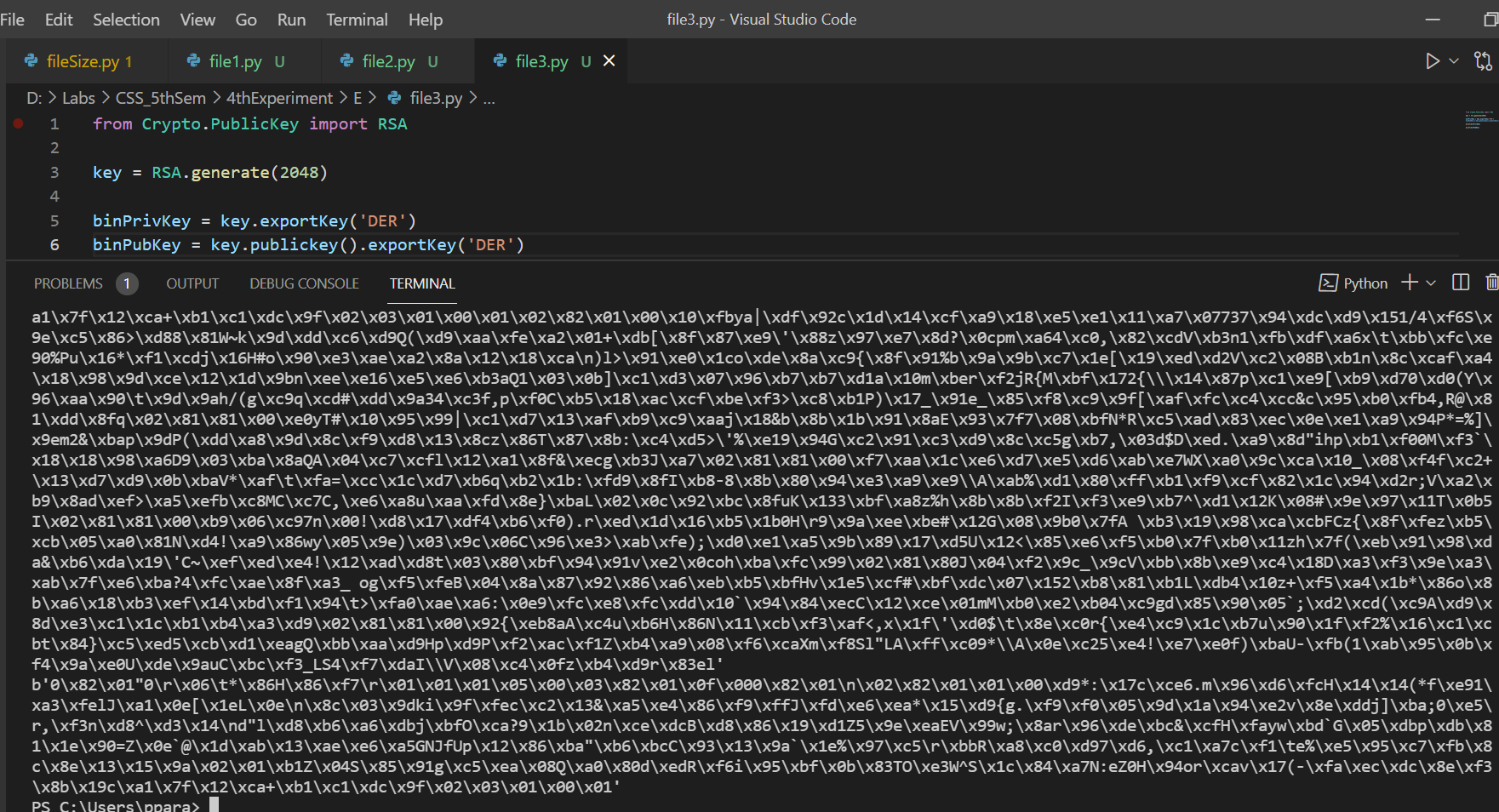
binPubKey = key.publickey().exportKey('PEM')

print binPrivKey

print binPubKey

activate MyEnv





The method of encoding the data that makes up the certificate is known as DER. DER can represent any type of data, however it is most used to describe an encoded certificate or a CMS container, whereas PEM is a means of encoding binary data as a string (ASCII armor). It has a header and a footer line (which indicate the type of data encoded and show the beginning and finish if the data is chained together), and the data in the middle is base 64 data. PEM is an abbreviation for Privacy Enhanced Mail; mail cannot directly contain unencoded binary values such as DER.

* **PGP**



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

-----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

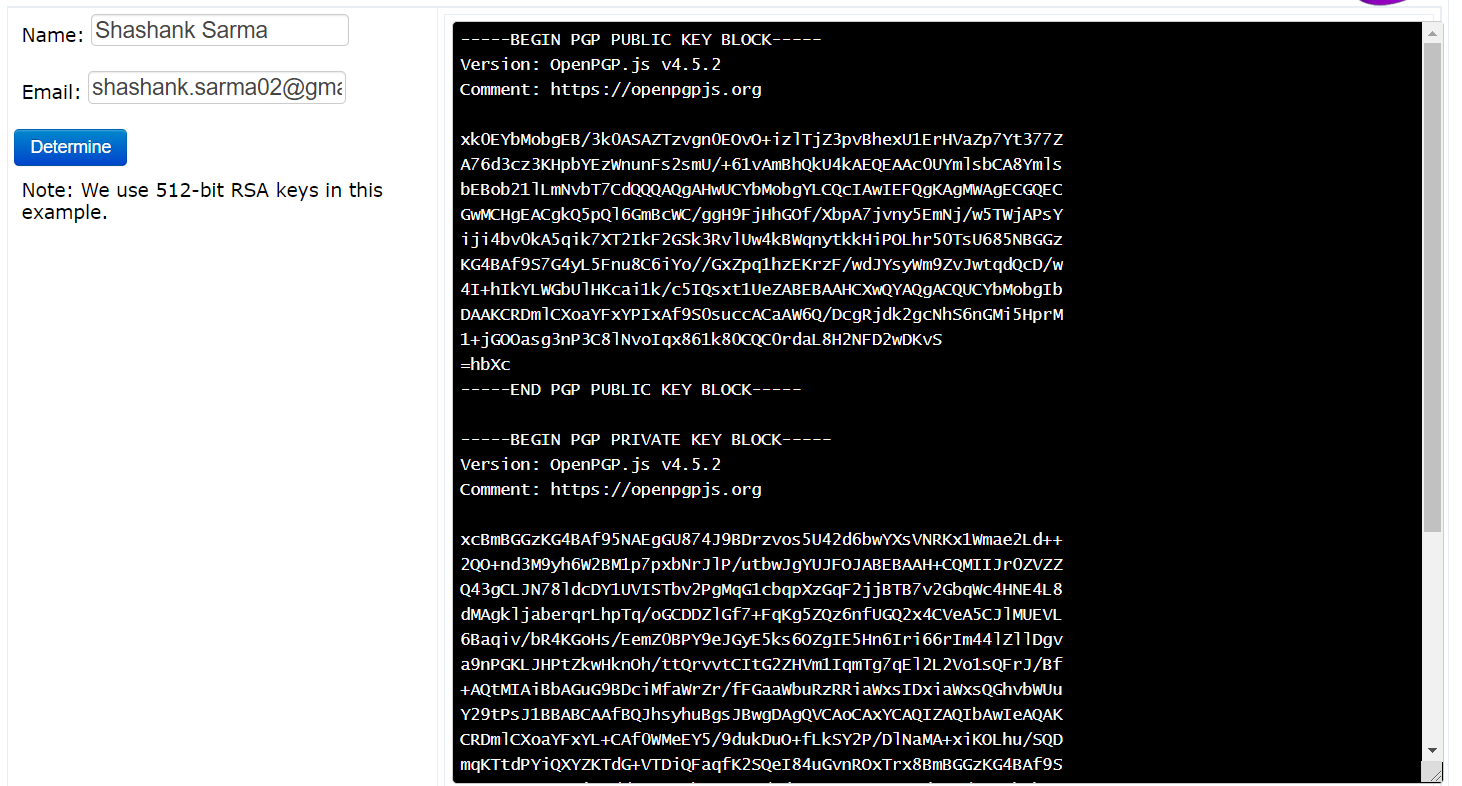
xcBmBFxDmL0BAgCKSz/MHy8c4HKJTKcIM3FM05R9InIIDiMrkCrBzOFzT7oM 1F9DXmmsKyYX4vn/IQ0aIyeRb/IwSNJvysBTDU0/ABEBAAH+CQMIBNTT/OPv TJzgvF+fLOsLsNYP64QfNHav5O744y0MLV/EZT3gsBwO9v4XF2SsZj6+EHbk O9gWi31BAIDgSaDsJYf7xPOhp8iEWWwrUkC+jlGpdTsGDJpeYMIsVVv8Ycam 0g7MSRsL+dYQauIgtVb3dloLMPtuL59nVAYuIgD8HXyaH2vsEgSZSQn0kfvF +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/cP12s3xCwxtVt+Zds8NdqysDO6yve2ha7cc+Vl8AP+YKqFT9IkMZJW/a qV+0VXeqyyru86F+xfrEKHdbAlqzMA== =5NaF

-----END PGP PRIVATE KEY BLOCK-----

**Doesn’t Work**

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

<https://asecuritysite.com/encryption/openpgp>

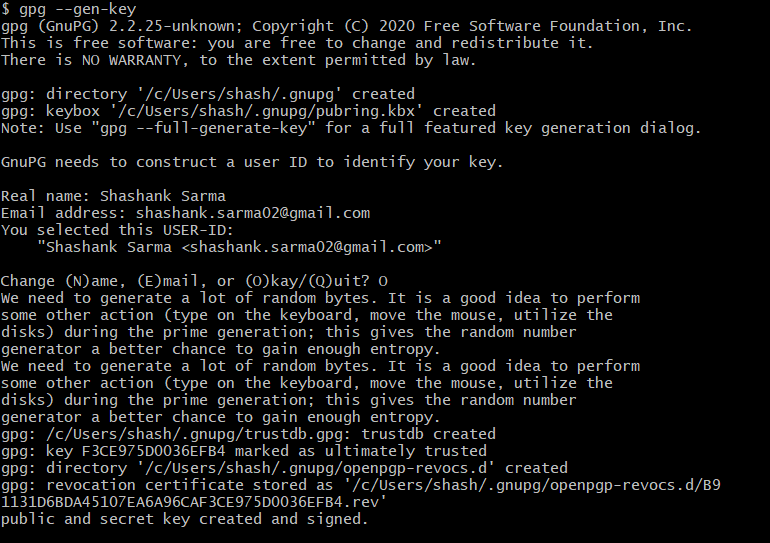


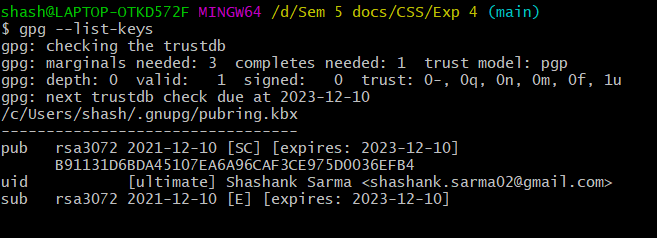
**F.3** An important element in data loss prevention is encrypted emails. In this part of the lab we will use an open source standard: PGP.

|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **1** | Create a key pair with (RSA and 2,048-bit keys):  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 | How is the randomness generated?  **PGP generates a session key, which is a secret key that can only be generated once. This key creates a random number based on your cursor movement and keystrokes. This session key is used to encrypt plaintext with an extremely safe and fast symmetric encryption technique, yielding ciphertext.**  Outline the contents of your key file:  **Both files have a header and footer that indicate whether they are PGP Public or Private key blocks, and the text between them contains the actual key.** |
| **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 publickey.key  Now list your keys with:  gpg --list-keys | Which keys are stored on your key ring and what details do they have:  **After obtaining Bill's key from the internet, importing their key, and then listing the keys, I discovered that the list includes both my personal key and Bill's key. The other information revealed was their public key encryption algorithm (RSA), their uid, which includes the user's name and email address, and finally the pgp key's expiry date.** |
| **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 | What does the –a option do:  **Create ASCII armored output. The default is to create the binary OpenPGP format.**  What does the –r option do:  **Encrypt the name of the user. If neither this option nor '—hidden-recipient' is used, GnuPG prompts for the user-id, unless '—default-recipient' is specified.**  What does the –u option do:  **Sign with your name as the key. It should be noted that this option overrides --default-key'.**  Which file does it produce and outline the format of its contents:  **It generates an.asc file (ascii armoured file) in which the header and footer designate the beginning and conclusion of the PGP communication, respectively, while the actual encrypted message is included between them.** |
| **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**  File Received |
| **5** |  |  |
| **6** |  |  |

**First I had to Download gpg and install it.**

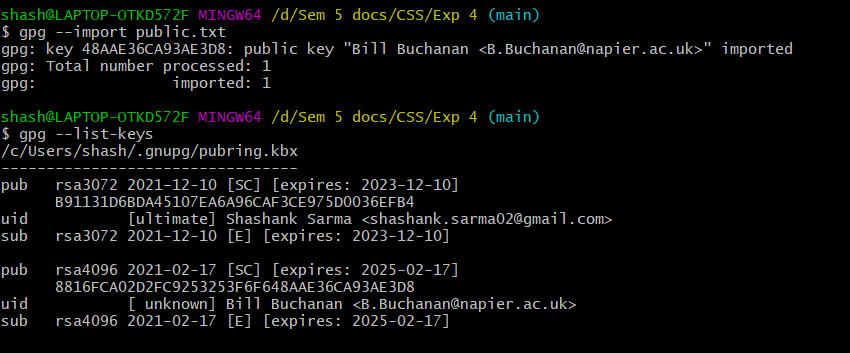
F.3.1)





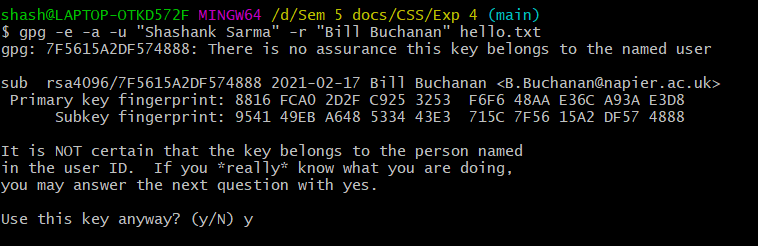
* **Both files have a header and footer that indicate whether they are PGP Public or Private key blocks, and the text between them contains the actual key.**
* **PGP generates a session key, which is a secret key that can only be generated once. This key creates a random number based on your cursor movement and keystrokes. This session key is used to encrypt plaintext with an extremely safe and fast symmetric encryption technique, yielding ciphertext**

F.3.2)



* **After obtaining Bill's key from the internet, importing their key, and then listing the keys, I discovered that the list includes both my personal key and Bill's key. The other information revealed was their public key encryption algorithm (RSA), their uid, which includes the user's name and email address, and finally the pgp key's expiry date.**

F.3.3)



What does the –a option do:

What does the –r option do:

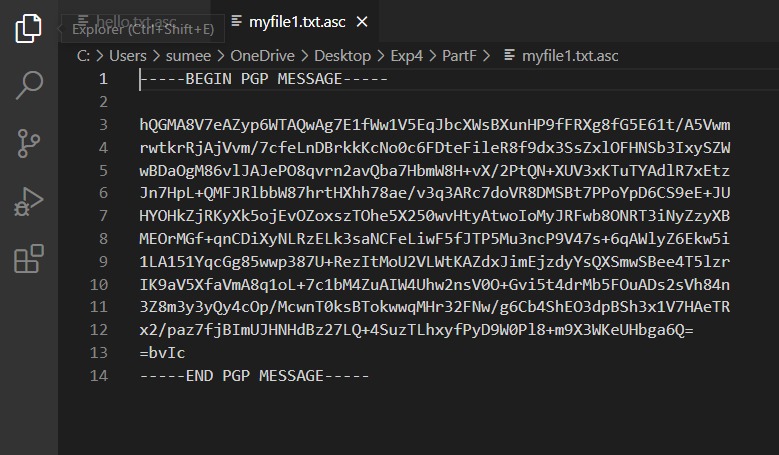
What does the –u option do:

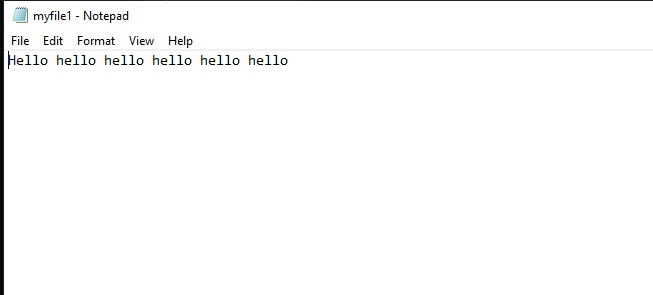
Which file does it produce and outline the format of its contents:

* **Create ASCII armored output. The default is to create the binary OpenPGP format.**
* **Encrypt the name of the user. If neither this option nor '—hidden-recipient' is used, GnuPG prompts for the user-id, unless '—default-recipient' is specified.**
* **Sign with your name as the key. It should be noted that this option overrides --default-key'.**
* **It generates an.asc file (ascii armoured file) in which the header and footer designate the beginning and conclusion of the PGP communication, respectively, while the actual encrypted message is included between them.**

F.3.4)







* **TrueCrypt**



True Crypt Doesn’t work on a WSL Kali on Windows. It requires a Linux Machine.

Since TrueCrypt is deprecated package, I had to use a modern replacement VeraCrypt.

|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **1** | Go to your Kali instance (User: root, Password:toor). Now Create a new volume and use an CPU (Mean) encrypted file container (use tc\_yourname) with a Standard TrueCrypt volume.  When you get to the Encryption Options, run the AES-Two-Seperate benchmark tests and outline the results: | CPU (Mean)  AES: **4.6 GB/s**  AES-Twofish: **939 MB/s**  AES-Two-Serpent: **531 MB/s**  Serpent -AES : **882 MB/s**  Serpent: **974 MB/s**  Serpent-Twofish-AES:  **543 MB/s**  Twofish: **1 GB/s**  Twofish-Serpent: **589 MB/s**  Which is the fastest:  **AES**  Which is the slowest:  **AES-Two-Serpent** |
| **2** | Select AES and RIPEMD-160 and create a 100MB file. Finally select your password and use FAT for the file system. | What does the random pool generation do, and what does it use to generate the random key?  **The random pool constantly captures the user's mouse movements, and the user is also alerted on the screen to move the mouse within the window as randomly as possible, which helps to increase the cryptographic strength of the encryption**  **keys.** |
| **3** | Now mount the file as a drive. | Can you view the drive on the file viewer and from the console? **Yes** |
| **4** | Create some files on your TrueCrypt drive and save them. | **I created a few file named test1.js and test2.js in the drive**    **Once I tried to mount the volume again I was prompted to enter the password that I had set earlier only after entering the correct password was I able to access all my files saved in that volume.** |

* **Reflective statements**



1. **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?**

No, SECP256 was not used neither popular before Bitcoin. It has become popular because unlike most curves SECP256 supports non-random method which allows efficient computation. As a result with good implementation, it could be 30% quicker than other curves. Furthermore, unlike famous NIST curves, secp256k1's constants were chosen in a predictable manner.

Private key size = 256 bit therefore possibilities = 2^256, which is 1.2e +77 keys.

1 TB = 8e+12 keys per second

Therefore it would take roughly 8 seconds to crack the key and so it will be possible to determine the private key.

* **What I should have learnt from this lab?**



The key things learnt:

* The basics of the RSA method.
* The process of generating RSA and Elliptic Curve key pairs.
* How private key can be used to sign the data and public key to verify the data.