


# 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

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

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

mQENBFTzi1ABCADIEWch0yqRQmU4AyQAMj2Pn68Ssqo9lTPdPCitwo9Lbtdv1YCFz
w3qLlp2RORMP+kpdi92CIhduYHDMZFHZ3IWTBgo9+y/Np9UJ6tNGocrqs4xwz15
4vx4jJRddC7QySSh9UXDPrwf9sgqEv1pah136r95ZuyjC1EXnoNxdLJtx8PlICxc
hV/v4+kFoyzYh+HDJ4xP2bt1S07dkasYZ6cA7BHYi9k4xgEwxvVYtNjSPjTSQY5R
cTayXveGafuxmhSauZKiB/2TFErjEt49Y+p07tPTLX7bhMBVbUvojt/JeUKV6vK
R82dmOd8seUvhwOHYB0JL+3S7PgFFsLo1NV5ABEBAAG0LkpbGwgQnVjaGFuYw4g
KE5vbmUpIDx3LmJlY2hhbmFuQG5hcGllci5hy51az6JATkEEwECACMFA1Tzi1AC
GWMHCwkIBwMCAQYVCAIJCgSEFgIDAQIEAQIXgAAKCRDsAFZRGtdPQi13B/9KHeFb
l1AxqbafFGRDevx8UfPnEww4FFqwhcr8RLWye8/COlUpB/5AS2yvojmbNFMGzURb
LGf/u1LVH0a+NHQu57u8Sv+g3bBthEPH4bKaEzBYRS/dYH0x3APFyIayfm78JVRf
zdeTOOf6PaXUTRx7iscCTkN8DUD3lg/465ZX5ah3HwFFX500JSPSt0/udqjoQuAr
WA5JqB//g2GfzZe1UZH5Dz3PBbJky8GiIFLm0OXSEIgAmpvc/9NjzAgjOW56n3Mu
sjvkiBc+lljw+rOo97CfJMpmtcovehvQv+KG0LZnpibiwvmm3vt7E6kRy4gEbbDu
enHPDqhsvcqTDqaduQENBFTzi1ABCACzpJgZLK/sge2rMLURUQQ6l02UrS/GilGC
ofq3wPndt5hejarwMMWn65Pb0Dj0i7vnorhL+fdb/J8b8QTiyp7i03dzvhdahcQ5
8afvCjQtQstY8+K6kZFzQOBgyOS5rHAKHNSPFq45MlnPo5aadVP7s9mdMILITv1b
CFhcLoC60qy+JoahupJqHBqGc48/5NU4qbt6fB1AQ/H4M+6og40ozohgkQb80Hox
YbJV4sv4vYMuLd+FKog2RdGenMM/awdqYo90qb/w2aHCCyXmhGHEEuok9jbc8cr/
xrWL0gdWlWpad8RfQwyVU/VZ3Eg30seL4SedEmw00
cr15XDIs6dpABEBAAGJAR8E
GAECAAKFA1Tzi1ACGwWACgkQ7ABWURrXT0KZTgf9Fupkh3wv7ac5M2wwdEjt0rDx
nj9kxH99hhutX2EHXUNLH+SwLGHBq502sq3jFP+owEhs8/Ez0j1/fSKiqAd1z3mB
dbqWPjzPTY/m0It+ww3epOM75uwjD35PF0rKxxZmEf6SrjZD1sK0B9bry2v9iWN9
9ZkuvCFH4vT++PognQLTuqNX0FGpD1agrG01XSCTJWQXCXPfwdtbIdThBgzh4f1Z
ssAibCaB1QkzfbPvrMzdTIP+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?

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-----
\nMIICXgIBAAKBgQD0Ihiws15X/6xiLAVCBzpgvnuvMzHBJk58w0WrdfyEACTY10oG\n+6auNFGqQHYHbfKaz1Ei4prAo
e01S/R6jpx8ZqJUN0WkNn5G9nmjJha9Pag28ftD\nrst+4LktaQrxdNdrusP+qI0NiYbNBH6qvCrK0aGiucextehnuoqg
DcqmRwIDAQAB\nAoGAZCaJu0Mj2ieJxRU+/rRzoFeuXylUNWQC6toCfNY7quxkdDV2T8r038Xc0fpb\nnsdrix3CLYuSnZ
ak3B76Mbo/oxQVBjDQZ7jvQ5K41nvCEZ0trDBEx5Ue6CB54iNmC\n+Qywx+u40ZPURq61YG7D+F1aWRvczdEZgKHPXl/+
s5pIvAKCQQDw4V6px/+dJuZV\n5Eg200Ze0m9Lvaq+G9UX2xTA2AUuH8Z79e+SCus6fMVl+Sf/W3y3uxp8B662bXhz\nny
heH67aDAKEA9rQrvmFj65n/D6eH4JAT40P/+iCQNgLYDW+u1Y+MdmD6A0Yjehw3\nnsuT9JH0rveBET959kP0xCx+iFEj1
81t17QJBAMcp4GZK2eXrx0jhnh/Mq51dku6Z\n/n/NHBG3j1CizGT8oqNaeK2jGLW6D5RxGgZ8TINR+HevGR3JAzhtNftGM
JDtccQQC3\nnIqRexVmZaexnrwu07f9zsI0zG5BzJ8VOpBt7Owah8fdmOsJXNgv55vbsAWdYBbUw\nnPQ+lc+7WPRNKT5sz
/im5AKEAi9Is+fgNy4q68nxP11rBQUV3Bg3S7k7oCJ4+ju4W\nnNXCCvRjQhpnVhlor7y4FC2p3thje9xox6QiwNr/5siy
cCW==\n-----END RSA PRIVATE KEY-----
```

And receives a ciphertext message of:

```
uw6FQth0Pkawc3haoqxbjIA7q2rF+G0Kx3z9ZDPZGU3NmBfzpd9ByU1ZbtbgKC8ATVZzwj15Aeteonbj03EHQC4A5Nu0x
KTwpqpngYRGgmZMgtb1w3wB1NQYovDSRUGt+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-----
\nMIICXgIBAAKBgQD0Ihiws15X/6xiLAVCBzpgvnuvMzHBJk58w0WrdfyEACTY10oG\n+6auNFGqQHYHbfKaz1Ei4prAo
e01S/R6jpx8ZqJUN0WkNn5G9nmjJha9Pag28ftD\nrst+4LktaQrxdNdrusP+qI0NiYbNBH6qvCrK0aGiucextehnuoqg
DcqmRwIDAQAB\nAoGAZCaJu0Mj2ieJxRU+/rRzoFeuXylUNWQC6toCfNY7quxkdDV2T8r038Xc0fpb\nnsdrix3CLYuSnZ
ak3B76Mbo/oxQVBjDQZ7jvQ5K41nvCEZ0trDBEx5Ue6CB54iNmC\n+Qywx+u40ZPURq61YG7D+F1aWRvczdEZgKHPXl/+
s5pIvAKCQQDw4V6px/+dJuZV\n5Eg200Ze0m9Lvaq+G9UX2xTA2AUuH8Z79e+SCus6fMVl+Sf/W3y3uxp8B662bXhz\nny
heH67aDAKEA9rQrvmFj65n/D6eH4JAT40P/+iCQNgLYDW+u1Y+MdmD6A0Yjehw3\nnsuT9JH0rveBET959kP0xCx+iFEj1
81t17QJBAMcp4GZK2eXrx0jhnh/Mq51dku6Z\n/n/NHBG3j1CizGT8oqNaeK2jGLW6D5RxGgZ8TINR+HevGR3JAzhtNftGM
JDtccQQC3\nnIqRexVmZaexnrwu07f9zsI0zG5BzJ8VOpBt7Owah8fdmOsJXNgv55vbsAWdYBbUw\nnPQ+lc+7WPRNKT5sz
/im5AKEAi9Is+fgNy4q68nxP11rBQUV3Bg3S7k7oCJ4+ju4W\nnNXCCvRjQhpnVhlor7y4FC2p3thje9xox6QiwNr/5siy
cCW==\n-----END RSA PRIVATE KEY-----"

ciphertext=base64.b64decode("uw6FQth0Pkawc3haoqxbjIA7q2rF+G0Kx3z9ZDPZGU3NmBfzpd9ByU1ZbtbgKC8A
TVZzwj15Aeteonbj03EHQC4A5Nu0xKTwpqpngYRGgmZMgtb1w3wB1NQYovDSRUGt+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)

We will use OpenSSL to perform the following:

No	Description	Result
B.1	<p>First we need to generate a key pair with:</p> <pre>openssl genrsa -out private.pem 1024</pre> <p>This file contains both the public and the private key.</p>	<p>What is the type of public key method used:</p> <p>How long is the default key:</p> <p>Use the following command to view the keys:</p> <pre>cat private.pem</pre>
B.2	<p>Use following command to view the output file:</p> <pre>cat private.pem</pre>	<p>What can be observed at the start and end of the file:</p>
B.3	<p>Next we view the RSA key pair:</p> <pre>openssl rsa -in private.pem -text</pre>	<p>Which are the attributes of the key shown:</p> <p>What is the number of bits in the public modulus? How many bits do the prime numbers have? What is the value of e?</p>
B.4	<p>Let's now secure the encrypted key with 128-bit AES:</p> <pre>openssl rsa -in private.pem -aes128 -out key3des.pem</pre>	<p>Why should you have a password on the usage of your private key?</p>
B.5	<p>Next we will export the public key:</p> <pre>openssl rsa -in private.pem -out public.pem -outform PEM -pubout</pre>	<p>View the output key. What does the header and footer of the file identify?</p>

<b>B.6</b>	<p>Now create a file named “myfile.txt” and put a message into it. Next encrypt it with your public key:</p> <pre>openssl pkeyutl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin</pre>	
<b>B.7</b>	<p>And then decrypt with your private key:</p> <pre>openssl pkeyutl -decrypt -inkey private.pem -in file.bin -out decrypted.txt</pre>	What are the contents of decrypted.txt?
<b>B.8</b>	<p>What can you observe between these two commands for differing output formats:</p> <pre>openssl pkeyutl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin</pre> <pre>cat file.bin</pre> <p>and:</p> <pre>openssl pkeyutl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin -hexdump</pre> <pre>cat file.bin</pre>	What can you observe in the different of the output files:

## C OpenSSL (ECC)

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
<b>C.1</b>	<p>First we need to generate a private key with:</p> <pre>openssl ecparam -name secp256k1 -genkey -out priv.pem</pre> <p>The file will only contain the private key, as we can generate the public key from this private key.</p> <p>Now use “cat priv.pem” to view your key.</p>	Can you view your key?
<b>C.2</b>	<p>We can view the details of the ECC parameters used with:</p> <pre>openssl ecparam -in priv.pem -text -param_enc explicit -noout</pre>	<p>Outline these values:</p> <p>Prime (last two bytes):</p> <p>A:</p> <p>B:</p> <p>Generator (last two bytes):</p>

		Order (last two bytes):
<b>C.3</b>	<p>Now generate your public key based on your private key with:</p> <pre>openssl ec -in priv.pem -text -noout</pre>	<p>How many bits and bytes does your private key have:</p> <p>How many bit and bytes does your public key have (Note the 04 is not part of the elliptic curve point):</p> <p>What is the ECC method that you have used?</p>
<b>C.4</b>	<p>First we need to generate a private key with:</p> <pre>openssl ecparam -list_curves</pre>	Outline three curves supported:
<b>C.5</b>	<p>Let's select two other curves:</p> <pre>openssl ecparam -name secp128r1 -genkey -out priv.pem openssl ecparam -in priv.pem -text -param_enc explicit -noout</pre> <pre>openssl ecparam -name secp521r1 -genkey -out priv.pem openssl ecparam -in priv.pem -text -param_enc explicit -noout</pre>	How does secp128k1, secp256k1 and secp512r1 differ 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?

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

## D Elliptic Curve Encryption

**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](https://asecuritysite.com/encryption/ecc_points_real)  
(or for simpler code you can use [https://asecuritysite.com/encryption/ecc\\_points3](https://asecuritysite.com/encryption/ecc_points3))

First five points:

## E RSA

**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):

When you identify 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=  
q=

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

N=  
PHI =

Now pick a value of  $e$  which does not share a factor with  $\text{PHI}$  [ $\text{gcd}(\text{PHI}, e) = 1$ ]:

$e =$

Now select a value of  $d$ , so that  $(e \cdot d) \pmod{\text{PHI}} = 1$ :

[Note: You can use this page to find  $d$ : <https://asecuritysite.com/encryption/inversemod>]

$d =$

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

$C = M^e \pmod{N} =$

Now decrypt your ciphertext with:

$M = C^d \pmod{N} =$

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  $(d \cdot e) \pmod{\text{PHI}} = 1$ . 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) =

Inverse of 65537 (mod 1034776851837418226012406113933120080) =

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)?

**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)
```

## F PGP

**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 **pgp mykey.key**):

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

xk0EXEOYVQECAIpLP8wFLxzgco1MpwgzCUzT1H0i cgg0IyuQKsHM4XNPugzu
X0NeaawrJhfi+f8hDRojJ5Fv8jBI0m/KwFMNTT8AEQEAAC0Uym1sbCA8Ym1s
bEBob21lLnNvbT7CdQQQAQgAHwUCXEOYVQYLCQCIawIEFQgKAgMwAgECGQEC
GwMCHgEACgkQoNsXEDYt2ZjkTAH/b6+pdFQLi6zg/Y0tHS5PPRV1323cwoay
vMcPjnwq+VfiNyXzy+UJKR1PXskzDvHMLoyVpUcjle5ChyT5LOW/ZM5NBfxD
mLOBagDY1tSt06vVQxu3jmfLzKMar4kLqQIuFFRCapRUHYLOjw1gJZS9p0bF
S0qS8ZMEGpN9QZxkg8YECH3gHx1rvALTABEBAAHXwQYAQgACQUCE0YVQIb
DAAKCRG2xcQNi3ZmMAGAF9w/XazfELDG1w3512zw12rkWm7rk97aFrtxz5W
Xwa/5gqoVP0iQxklb9qpx7Rvd6rLKu7zoX7F+Sqod1sCwrMw
=cXT5
-----END PGP PUBLIC KEY BLOCK-----

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

xcBmBFxDmLOBagCKSz/Mhy8c4HKJTKcIM3FM05R9InIIDiMrkCrBzOFzT7oM
1F9DXmmskyYX4vn/IQ0aIyerB/IwsNJvysBTDU0/ABEBAAH+CQMIBNTT/OPV
TJzgvF+fL0sLsNYP64QfNHav50744y0MLV/EZT3gsBw09v4XF2SsZj6+EHbk
09gwi31BAIDgSaDsJYf7xPOhp8iEwwwrUkC+j1GpdTsGDJpeYMIsvvv8Ycam
0g7MSRSL+dYQauIgtVb3d1oLMptuL59nVAYuIgd8HXyah2vsEgSZSQn0kfVf
+dweqJxwFM/ux5PVKcuYsroJFBE01zas4ERfxbbwnsQgNHpjdiPueHx6/4EO
b1kmhOd6UT7Bamuby7bcma1PBSv8PH31Jt8SzRRiawxsIDxiawxsQGHvbWuu
Y29tP5J1BBABCAAFBQJCQ5i9BgsJBwgDagQVCAoCAXYCAQIZAQIBawIeAQAK
CRG2xcQNi3ZmORMAF9vr6KN9AuLrOD9jS0dLk89G/XfbdzChrK8xw+odar5
v+I3JfNj5Qkphu9eyTMO8cws7Jw1RyOV7kkHJPks7D9kx8BmBFxDmLOBagDY
1tSt06vVQxu3jmfLzKMar4kLqQIuFFRCapRUHYLOjw1gJZS9p0bFS0qS8ZME
GpN9QZxkg8YECH3gHx1rvALTABEBAAH+CQMI2Gyk+BqVogzgZX3C80JRLBRM
T4sLCHOUG1waspe+qatOVjeEuxA5DuSs0bVMrw7mJYQZLjtjNkFAT921Swfxy
gavS/biL1w3QGA0CT5mqijKr0nurKkekKBDsgjkjvbiOPLMYHfep0ju1322
Nw4V3JQ04LBh/sdgGbrnww3LhHEK4qe70cuiert8C+S5xfG+T5RWADi5HR8u
UTYH8x1h0ZrOf7K0Wq4ucNvrUm6c35H61c1C4Zaar4JSN8fZPqVKLlHTVCL9
```



```

1pDzXxqxKjS05KXXZBh5w18EGAEIAAkFA1xDmL0CGwwACgkQoNsXEDYt2ZjA
BgH/CP12s3xCwxtVt+Zds8NdqysDO6yve2ha7cc+v18AP+YKqFT9IkMZJW/a
qV+0VXeqqyru86F+xfrEKHdbA1qzMA==
=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	Result
1	<p>Create a key pair with (RSA and 2,048-bit keys):</p> <pre>gpg --gen-key</pre> <p>Now export your public key using the form of:</p> <pre>gpg --export -a "Your name" &gt; mypub.key</pre> <p>Now export your private key using the form of:</p> <pre>gpg --export-secret-key -a "Your name" &gt; mypriv.key</pre>	<p>How is the randomness generated?</p> <p>Outline the contents of your key file:</p>
2	<p>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 <a href="http://asecuritysite.com/public.txt">http://asecuritysite.com/public.txt</a> and send the email to him):</p> <pre>gpg --import theirpublickey.key</pre> <p>Now list your keys with:</p> <pre>gpg --list-keys</pre>	<p>Which keys are stored on your key ring and what details do they have:</p>
3	<p>Create a text file, and save it. Next encrypt the file with their public key:</p> <pre>gpg -e -a -u "Your Name" -r "Your Lab Partner Name" hello.txt</pre>	<p>What does the <code>-a</code> option do:</p> <p>What does the <code>-r</code> option do:</p>

		<p>What does the <code>-u</code> option do:</p> <p>Which file does it produce and outline the format of its contents:</p>
4	<p>Send your encrypted file in an email to your lab partner, and get one back from them.</p> <p>Now create a file (such as <code>myfile.asc</code>) and decrypt the email using the public key received from them with:</p> <p><b><code>gpg -d myfile.asc &gt; myfile.txt</code></b></p>	Can you decrypt the message:
5	<p>Next using this public key file, send Bill (<code>w.buchanan@napier.ac.uk</code>) an encrypted question (<a href="http://asecuritysite.com/public.txt">http://asecuritysite.com/public.txt</a>).</p>	Did you receive a reply:
6	<p>Next send your public key to Bill (<code>w.buchanan@napier.ac.uk</code>), and ask for an encrypted message from him.</p>	

## G SSH Key pairs

**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
AAAAB3NzaC1yc2EAAAADAQABAAQDLrriUNYTyWuClIw7H6yea3hMV+rm029m2f6Iddt1ImHrOXjNwYyt4E1kkc7Azo
y899C3gpx0kJK45k/CLbPnrHvKLVtQ0AbzWEQpOKxI+tw06PcqJNmTB8ITRLqIFQ++ZanjHWMw2Odew/514y1dQ8dcccO
uzeGhL2Lq9dtfhSxx+1cBLcyoSh/1Qcs1HpXtpwU8JMxwJ1409RQOVn3gOusp/P/0R8mz/RwkmsFsyDRLgQK+xtQxbpbo
dpnz51IOPwn5LnT0si7eHmL3wikTyg+QLZ3D3m44NCeNb+boJbfaQ2ZB+lv8C3oxy1xSp2sxzPZMbrZWqGSLPjgDiFIBL
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?

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

---

The following is code which performs RSA key generation, and the encryption and decryption of a message ([https://asecuritysite.com/encryption/rsa\\_example](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)
```

Can you decrypt this:

```
fIVuuWFLVANS9MjatXbIbth7/n0dBpDirXki82jzovXS/krxy43cP0J9j1Nz4dqxLgdiqtRe1AcymX06Ju01SrcqDEh31
Qxou1KUVv7jG9GE3pSxHq4dQ1cWdHz95b9go6QYbe/5S/uJgo1R+S9qaDE8tXYysP8FeXIPd0dxxHo=
```

The private key is:

```
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQCfQfirYVXgzT90v6SggeID7q/WK1xavTNGVfo1DUOcrX1/egRG
4iag5tiTbrMYCQ8CSTYn7q0U4AmBXih1bWdQf6MMk60EoDxdwZTiG1MmQ1wZikFE
s7sYSog/poYleCeYw8kVzHNWnt9IuQwekIg6ZHkwp4NE/aw8HxvEwYRqCQIDAQAB
AoGAE6rk1Fmxbt06GHNWZQQ8QsSP2Q2qARgjiGxzY38Dwg6MYiNR8uUL6zQHDBIQ
OQgpW9lPWd24D0tpsRnNOFvtMeafcxmykX+qHGtNekJuTtqSm2eTI6gNbC8iosGT
XJEPm8tc/dfZ2sDoblfi0a1wFOzWo8vKaLnnAdMHoz8mDo8CQQDCMx08Jv1TW1z1
+4UTEnyyYmIezw5ORfMqPtN1LpQ4ptYnHNMVJPwcpRwBYZFh1POPtuvwo6gzv82G
QpgQsd4PAkEA0fA8e8R6jbeUR1HxsqweCnPz3Ahq5Ya5WA6HyJQm19aDVqKDDp2L
```

```
3AcqsvFEKJ/T34r31so2yw6hj2yFBnz0ZwJBAIqanrgJ1CpJYBGJJd6J6FQNIgjp
MUWuaTJyqsvNFd81PF2oFgPWYDKQKV/w/tRkvD2LhVCSjf95wsADkbMASAMCQAHO
wwQOwV2eccbERAJv5yQJMeqKWQ6FTyIx36I/VqqC10bwy2hSnnb9ybGe6BPGgFLE
HMTjSerDEU0Qm5UXhXkCQqCP1ZJq1gksBN/TULHC4RgsXIX+oFylBrkiFamYsuEt
Kn52h41px7FI5TXcqIDPw+uqAu50JnwDR0dLyy6fvIce
-----END RSA PRIVATE KEY-----
```

## J 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.
- To illustrate how the private key is used to sign data, and then using the public key to verify the signature.

Reflective question:

**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?**