# 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

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>

User ID: Bill Buchanan <w.buchanan@napier.ac.uk>

Key ID: a7a50d1c92a139b8

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)?

<https://news.ycombinator.com/item?id=6376954>

https://www.theguardian.com/pgp

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

ASCII-Armor is a binary-to-textual encoding converter and is a feature of a type of encryption called pretty-good-privacy. It envolves encasing encrypted messages in ASCII so that they can be sent in a standard messaging format such as an email. Binary data in represented as ASCII characters and hence can be transmitted and received by a variety of messaging format, e.g. email.

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/6xiLAVcBzpgvnuvMzHBJk58wOWrdfyEAcTY10oG\n+6auNFGqQHYHbfKaZlEi4prAoe01S/R6jpx8ZqJUN0WKNn5G9nmjJha9Pag28ftD\nrsT+4LktaQrxdNdrusP+qI0NiYbNBH6qvCrK0aGiucextehnuoqgDcqmRwIDAQAB\nAoGAZCaJu0MJ2ieJxRU+/rRzoFeuXylUNwQC6toCfNY7quxkdDV2T8r038Xc0fpb\nsdrix3CLYuSnZaK3B76MbO/oXQVBjDQZ7jVQ5K41nVCEZOtRDBeX5Ue6CBs4iNmC\n+QyWx+u4OZPURq61YG7D+F1aWRvczdEZgKHPXl/+s5pIvAkCQQDw4V6px/+DJuZV\n5Eg20OZe0m9Lvaq+G9UX2xTA2AUuH8Z79e+SCus6fMVl+Sf/W3y3uXp8B662bXhz\nyheH67aDAkEA9rQrvmFj65n/D6eH4JAT4OP/+icQNgLYDW+u1Y+MdmD6A0YjehW3\nsuT9JH0rvEBET959kP0xCx+iFEjl81tl7QJBAMcp4GZK2eXrxOjhnh/Mq51dKu6Z\n/NHBG3jlCIzGT8oqNaeK2jGLW6D5RxGgZ8TINR+HeVGR3JAzhTNftgMJDtcCQQC3\nIqReXVmZaeXnrwu07f9zsI0zG5BzJ8VOpBt7OWah8fdmOsjXNgv55vbsAWdYBbUw\nPQ+lc+7WPRNKT5sz/iM5AkEAi9Is+fgNy4q68nxPl1rBQUV3Bg3S7k7oCJ4+ju4W\nNXCCvRjQhpNVhlor7y4FC2p3thje9xox6QiwNr/5siyccw==\n-----END RSA PRIVATE KEY-----

And receives a ciphertext message of:

uW6FQth0pKaWc3haoqxbjIA7q2rF+G0Kx3z9ZDPZGU3NmBfzpD9ByU1ZBtbgKC8ATVZzwj15AeteOnbjO3EHQC4A5Nu0xKTWpqpngYRGGmzMGtblW3wBlNQYovDsRUGt+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/6xiLAVcBzpgvnuvMzHBJk58wOWrdfyEAcTY10oG\n+6auNFGqQHYHbfKaZlEi4prAoe01S/R6jpx8ZqJUN0WKNn5G9nmjJha9Pag28ftD\nrsT+4LktaQrxdNdrusP+qI0NiYbNBH6qvCrK0aGiucextehnuoqgDcqmRwIDAQAB\nAoGAZCaJu0MJ2ieJxRU+/rRzoFeuXylUNwQC6toCfNY7quxkdDV2T8r038Xc0fpb\nsdrix3CLYuSnZaK3B76MbO/oXQVBjDQZ7jVQ5K41nVCEZOtRDBeX5Ue6CBs4iNmC\n+QyWx+u4OZPURq61YG7D+F1aWRvczdEZgKHPXl/+s5pIvAkCQQDw4V6px/+DJuZV\n5Eg20OZe0m9Lvaq+G9UX2xTA2AUuH8Z79e+SCus6fMVl+Sf/W3y3uXp8B662bXhz\nyheH67aDAkEA9rQrvmFj65n/D6eH4JAT4OP/+icQNgLYDW+u1Y+MdmD6A0YjehW3\nsuT9JH0rvEBET959kP0xCx+iFEjl81tl7QJBAMcp4GZK2eXrxOjhnh/Mq51dKu6Z\n/NHBG3jlCIzGT8oqNaeK2jGLW6D5RxGgZ8TINR+HeVGR3JAzhTNftgMJDtcCQQC3\nIqReXVmZaeXnrwu07f9zsI0zG5BzJ8VOpBt7OWah8fdmOsjXNgv55vbsAWdYBbUw\nPQ+lc+7WPRNKT5sz/iM5AkEAi9Is+fgNy4q68nxPl1rBQUV3Bg3S7k7oCJ4+ju4W\nNXCCvRjQhpNVhlor7y4FC2p3thje9xox6QiwNr/5siyccw==\n-----END RSA PRIVATE KEY-----"

ciphertext=base64.b64decode("uW6FQth0pKaWc3haoqxbjIA7q2rF+G0Kx3z9ZDPZGU3NmBfzpD9ByU1ZBtbgKC8ATVZzwj15AeteOnbjO3EHQC4A5Nu0xKTWpqpngYRGGmzMGtblW3wBlNQYovDsRUGt+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?

Python is your friend

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** | 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 according to statement, removing the 1024 gives a 2048 private key.  Use the following command to view the keys:  cat private.pem  No, just the private |
| **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:  -----BEGIN PRIVATE KEY-----  -----END PRIVATE KEY-----  Header and Footer |
| **B.3** | Next we view the RSA key pair:  openssl rsa -in private.pem -text | Which are the attributes of the key shown:  Private-Key: (1024 bit, 2 primes)  modulus: 00:d7:57:8a:13:0f:41:d3:b7:ae:a2:6d:fa:f8:3d: a6:72:71:8c:89:9e:80:30:b9:1c:f9:33:8d:0d:c8: 94:a8:59:f7:1c:46:d4:7f:38:e2:33:8f:fb:20:fb: 70:b8:9b:bf:81:de:a2:13:f7:7b:75:e9:37:ad:f9: c9:6b:a4:30:27:e2:bd:3a:3c:4d:65:00:7d:47:7f: e8:b1:23:bf:25:90:6c:46:16:41:2d:95:3c:e5:48: f7:50:f2:48:d1:d2:9d:a3:d2:5e:3b:a4:3f:b8:6c: 95:2d:bf:5b:97:22:3b:de:bb:96:93:37:10:b6:ef: a1:2f:4d:61:8d:65:53:8d:e9  publicExponent: 65537 (0x10001)  privateExponent:  prime1:  prime2:  exponent1:  exponent2:  coefficient:  Which number format is used to display the information on the attributes:  Key attributes are written Hex  The Key is written in Base64  We get **\*\*modulus\*\*** (N), **\*\*publicExponent\*\*** (e), **\*\*privateExponent\*\*** (d), **\*\*prime1\*\*** (p), **\*\*prime2\*\*** (q). The other parameters are stored to speed up the RSA process, such as **\*\*exponent1\*\*** (d mod p-1), **\*\*exponent2\*\*** (d mod q-1) and **\*\*coefficient\*\*** (inv q mod p). |
| **B.4** | Let’s now secure the encrypted key with 3-DES:    openssl rsa -in private.pem -des3 -out key3des.pem | Why should you have a password on the usage of your private key?  Because if someone gets your private key they can decrypt all messages encrypted with your public key and impersonate you (sign things as if they were you) |
| **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?  That it’s 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 rsautl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin | openssl pkeyutl -encrypt -inkey private.pem -in myfile.txt -out file.bin  openssl pkeyutl -decrypt -inkey private.pem -in file.bin |
| **B.7** | And then decrypt with your private key:  openssl rsautl -decrypt -inkey private.pem -in file.bin -out decrypted.txt | What are the contents of decrypted.txt  Exactly what was put in the plaintext file |
| **B.8** | What can you observe between these two commands for differing output formats:  openssl rsautl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin  cat file.bin  and:  openssl rsautl -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 encrypted file is binary and significantly larger than the original which is an ascii text file.  The encrypted file is also smaller than the key file. |

## C 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:  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?  -----BEGIN EC PARAMETERS-----  BgUrgQQACg==  -----END EC PARAMETERS-----  -----BEGIN EC PRIVATE KEY-----  MHQCAQEEICLoBFk/GHhf/7hQkUb/WonODuIpklKtAeaMZz67BDSHoAcGBSuBBAAK  oUQDQgAESY5AspoDs9KLwYrCjbxBbiUDhx8AoVM6fB7TM9nPYZvydTMhJ1kXUHqL  KUM+dPe/0dkIl6qrpit14fvpW9rhoA==  -----END EC PRIVATE 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:  openssl ec -in priv.pem -text -noout | How many bits and bytes does your private key have:  32 bytes or 256 bits  How many bit and bytes does your public key have (Note the 04 is not part of the elliptic curve point):  Ignore the 04: 64 bytes 512 bits  What is the ECC method that you have used? secp256k1  We generate the public key from the private key. If we have a private key (priv) and a generator point (G). The public key is priv G. In this case we are using a curve of secp256k1.  openssl ec -in priv.pem -text -noout  read EC key  Private-Key: (256 bit)  priv:  a1:a2:f3:fb:15:d5:7e:2d:02:8b:f8:7f:4c:f2:2b:  1b:88:49:5a:08:6e:83:0f:88:8c:6a:b3:f7:a0:5b:  8f:4f  pub:  **04**:16:a1:b7:ac:70:1f:f1:fc:51:2b:ca:74:de:42:  f0:7b:06:30:8c:9d:57:b2:d2:01:3c:0b:08:e9:1c:  7d:e3:87:a5:60:9c:8d:6b:ca:6d:43:79:9d:61:fd:  6b:f5:ae:41:1e:39:34:93:0a:98:ea:7d:87:3b:4d:  38:de:10:90:38  ASN1 OID: secp256k1 |
| **C.4** | First we need to generate a private key with:  openssl ecparam -list\_curves | Outline three curves supported:  secp112r1 : SECG/WTLS curve over a 112 bit prime field  secp112r2 : SECG curve over a 112 bit prime field  secp128r1 : SECG curve over a 128 bit prime field  secp128r2 : SECG curve over a 128 bit prime field  secp160k1 : SECG curve over a 160 bit prime field  secp160r1 : SECG curve over a 160 bit prime field  secp160r2 : SECG/WTLS curve over a 160 bit prime field  secp192k1 : SECG curve over a 192 bit prime field  secp224k1 : SECG curve over a 224 bit prime field  secp224r1 : NIST/SECG curve over a 224 bit prime field  secp256k1 : SECG curve over a 256 bit prime field  secp384r1 : NIST/SECG curve over a 384 bit prime field  secp521r1 : NIST/SECG curve over a 521 bit prime field  prime192v1: NIST/X9.62/SECG curve over a 192 bit prime field  prime192v2: X9.62 curve over a 192 bit prime field  prime192v3: X9.62 curve over a 192 bit prime field |
| **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?  The parameters used are the same, except for the curve selected  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  secp128k1 uses 16 bytes or 128 bits for Prime, A and B as well as Order  20 bytes for Seed and  32 bytes for the Generator  Secp521r1 uses 64 bytes for Prime, A and B as well as Order  20 bytes for Seed and  132 bytes for the Generator  For the prime number we have 65 bytes + 1 bit (521 bits).  openssl ecparam -in priv.pem -text -param\_enc explicit -noout  EC-Parameters: (521 bit)  Field Type: prime-field  Prime:  01:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:  ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:  ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:  ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:  ff:ff:ff:ff:ff:ff |

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

## D 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/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.PrivateFormat.PKCS8,encryption\_algorithm=serialization.NoEncryption())

der = private\_key.private\_bytes(encoding=serialization.Encoding.DER,format=serialization.PrivateFormat.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:

Public key (PEM):

-----BEGIN PUBLIC KEY-----

MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAEPWSLAbfAhv2j/RpyTXkUJpqSeTiIXqEA

Nm28wLQA7sur7l92REPMvRA5QhkQBGwA5ewKKMZ4H65/DcFJA8+Usw==

-----END PUBLIC KEY-----

Private key (PEM):

-----BEGIN PRIVATE KEY-----

MIGEAgEAMBAGByqGSM49AgEGBSuBBAAKBG0wawIBAQQgaWMTPTQBmlvbPKthhbtL

503gbVr4tOgeTKk7/BAcBhqhRANCAAQ9ZIsBt8CG/aP9GnJNeRQmmpJ5OIheoQA2

bbzAtADuy6vuX3ZEQ8y9EDlCGRAEbADl7Aooxngfrn8NwUkDz5Sz

-----END PRIVATE KEY-----

Priv 256 bits

Pub 120

**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 (mod 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:

(1, 39) (1, 50) (3, 52) (3, 37) (4, 31)

## 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):

PublicKey(7735103127942066298058636189159963776471724750197536719443111867679022100727229309885293025445219733187004681705357305393037097443524366174028085573454059, 65537)

N is the same value as the encryption key and

d = 7735103127942066298058636189159963776471724750197536719443111867679022100727229309885293025445219733187004681705357305393037097443524366174028085573454059

e = 65537

PrivateKey(7735103127942066298058636189159963776471724750197536719443111867679022100727229309885293025445219733187004681705357305393037097443524366174028085573454059, 65537, 3210320964951465634789430464396463291271051668605108545842083751176730717194177991712864566600722787251226714614678086807234383228411897011779297642575873, 4670065961064050762254727923178593691718914308571740862864547152032530668400815323, 1656315605054037056034747563673863666854909478195137857727490805790230833)

N = 7735103127942066298058636189159963776471724750197536719443111867679022100727229309885293025445219733187004681705357305393037097443524366174028085573454059

e = 65537

p = 4670065961064050762254727923178593691718914308571740862864547152032530668400815323

q = 1656315605054037056034747563673863666854909478195137857727490805790230833

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=41

q=19

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

N= 41 \* 19 = 779

PHI = (19-1)\*(41-1) = 720

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

*e*=7

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*=103

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

C = M*e* (mod N) = 651

Now decrypt your ciphertext with:

M = Cd (mod N) = 777

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:

p: 89, q: 83, N: 7387, PHI:7216, e: 13, d: 6661, (e.d) (mod PHI): 1

M: 777, C: 3964

D: 777

p: 61, q: 103, N: 6283, PHI:6120, e: 107, d: 3203, (e.d) (mod PHI): 1

M: 777, C: 4990

D: 777

p: 463, q: 499, N: 231037, PHI:230076, e: 353, d: 181193, (e.d) (mod PHI): 1

M: 777, C: 175489

D: 777

**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)?

**E.4** 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)

PEM is in Base64 and DER is in hex

## 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 **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/cP12s3xCwxtVt+Zds8NdqysDO6yve2ha7cc+Vl8AP+YKqFT9IkMZJW/a

qV+0VXeqyyru86F+xfrEKHdbAlqzMA==

=5NaF

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

gpg mykey.key

gpg: WARNING: no command supplied. Trying to guess what you mean ...

gpg: DBG: FIXME: merging secret key blocks is not anymore available

gpg: DBG: FIXME: No way to print secret key packets here

pub rsa512 2019-01-19 [SC]

CFF4FA827DB6190A26668555A0DB1710362DD998

uid bill <bill@home.com>

sub rsa512 2019-01-19 [E]

**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** | 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**  u4\_pub.key  -----BEGIN PGP PUBLIC KEY BLOCK-----  mQGNBGPvUOkBDACrTAsm89vNR3f/i1xED3PL3VaqxY+FDBicIgweg0DdscPNW59W  DjrHeFONhCmEYMzn/5ZyDTcifMooqLmIFolbSa/1S5/ERaZW+A27lMMfFSxpPy3U  DAHQcvVgw8NOHF6JMFUAyubnb4of2YBvTfUUvNxFQAPsUgimOVap4sLrxItZ3dnP  aQpZdenVaaObLUfV/RkOJl3q/4JXIKC+2DpSiKU+E+w1JZ9t2cLuEif7heKn8lP9  +ZOqkWyiJadOuPkAK6CcyyrM2+8bpzuKp6XmuCMYYuptFGPPHACgNpxNITxtdTWO  r3a5dlD45lceHzgnB+Y1nOMdHR/oEfbiiiQxvtxwBgEcv32JgqD2xXNNNGBq5aR7  5/71FLyZqBUX3GVf6LUmTpK1OwNrFDXZ9WmRxen03R3g7pwCgcjIGthvZUMVAtjV  9lAP7pB5/jxHDIIj5umbqZ0FKzLA/s1lUK+9GvtMy9xF1/J1ejFsLTbS3qLwT++H  Y2ANm7zjAa8cDrUAEQEAAbRNTmlhbGwgRmlubGF5IEMmVCAoS2V5IFBhaXIgZm9y  IENyeXB0byBhbmQgVHJ1c3QpIDw0MDQzMjMwOUBsaXZlLm5hcGllci5hYy51az6J  AdcEEwEIAEECGwMFCwkIBwICIgIGFQoJCAsCBBYCAwECHgcCF4AWIQTcc+RR/6xh  DuR9e4vEvVPwKPXaXQUCY+9oYAUJACWuFQAKCRDEvVPwKPXaXaEyC/9waIATetXK  f32xR3gNgr45uK/85SAYxMjmgKq6qnU0ljJay7G80/vs7LdGB9EviVX1WVUTL4j2  b6y4s1QSEIRi7EgMwMET4J926Q/NlVFqAJHwP501CpqIiq9EnzndEDzO6WHUcoKa  8YIjJPACkjjnuSWz0ABf6yusHsKaN2HRuYYFV2GeKsXrkCbS2VD+LT8+yaPHTgqn  F5E8QwzFo49mxHYXJ+S7nhShMMsbO/yeOA6dn2KPbmbp8m4CLy5T5JZuv6P6o4dB  aageFvNTp+xGIgB/aSYnIOIwKqQdQ8AYycaEoOwzh2dgFy49ceiSP5/ux1ytpLKB  V462IcVlxX62B51zC3Db0/aCTjvgp0eksPUd4/kYO6YIQZJa1NkS51nyU4Hv8Knk  JecpuZiLZVPOaK5VlE1TvLnWPtuM37VVqiQhMNYY63ypyDDzTInRlnHtFL9NUWex  AwCihDmDGjTwXZ7CCE7yKuu6U31wxOQCVCUAHtDsTzAf5KQ7BShOcFyJAdEEEwEI  ADsWIQTcc+RR/6xhDuR9e4vEvVPwKPXaXQUCY+9Q6QIbAwULCQgHAgIiAgYVCgkI  CwIEFgIDAQIeBwIXgAAKCRDEvVPwKPXaXbNpDACTMuAEqW10HOrVK4zWtu5KXMPs  iLtxnbCeM0ScQiF9ADQ3JJSQo61L2P9EWnDbrda0OslbZ3LAGRUKLCfcOtT4Yu7i  lmYbFwtwz6mKsHFs59CMsTR2Lxe1btL4dx+ig6f3cZiDBVxMAH/D00rqlIlgeep9  e+DT+li/0yqVDzt1L7rCbE1NsTEnZ4pMWgwt/9VabGwEwJ4JkALF1/Ac8k5Ll7ig  I9tNjBFkj/3k6HoEyMoS60LxfXQbmQKO7hn+QuBmuRE9WOl5B+Mh50jbOG3ZNvAE  0Vs8afPxeEGyR/pCxGS2qa/VGxMKoxesK++zmQU6boexpp3hoG60uiK1173uceB8  SbqrVpI1tW5AeI1fjtPpJhSr7+2HJ+XV2QlhjMCq0gcurSKWgaQXUqFUYqT+MdlP  jZsPCl2hcsN8GccEYaV05WXF94m4Bgm0fA2tCRdKg1n9V3CY71tWexeO+rCZD9w6  XSEYibiOPlUspWsWDTy475f0jW5FRWnmHRLeFHm5AY0EY+9Q6QEMALPnRv9odTPF  WBkT8V55wxdKyhOmQ7GQFbq3D/VAhtRSNCI5lKcGxpz93ZzKq7gEVCHyKQ07KYqH  uIo+ZJZaZ5sa46WCzBR5KjIjVV+ByUGnpuUvv7r9nKeu9kGVse/QP5y70rkMph8U  b5xjiOlbj4xkxq1FTVzUYGR4Lq4Xb5MQUaJ92bWhzNV5g19jgG7Xt1IGtZI4thY3  6J70xjRZyuQgZhsh3KQQ3EDSHNDJk6lSdn53lwzGM51cgBSrs2cwss1zOElkIKkB  KOeRBD79McGb1N+MAqPZ7sT3M6XRzfIXJ7LCBfssvmlYP0cjyQiqPb7B3DtEnnvd  VXuVB3OkNTQObpZiOfHxBwPenx4mO3Cl5hUmVeyLF02nj29TMwVlCwfGZnQuIMAf  QWYIKPkjtOoeAHW4pDEDDE6HSUNztv+T6coUXVdLXdlLt/srWbr8rQ6HYrAzXDlQ  BUQj7ok4IC7G0CrbMtFazb/Rv+qC/Drls97ryg9W08IHU1ke4OpBxQARAQABiQG8  BBgBCAAmAhsMFiEE3HPkUf+sYQ7kfXuLxL1T8Cj12l0FAmPvaGkFCQAlrh4ACgkQ  xL1T8Cj12l0e1AwAirTmH4+i6PI/yUdWpkfdleENp0Qdap+e8QRSiL7TvzVrFVi2  xeOo668q2iPB56fJ2Jy1JueBi62F0X5XKP67+m+6yD44q+EENf68nqL+9FkbJhPV  WLPLdkZJi8iuM70iZ8S+KRvDrqoJiZ0y7Aj/mHwAixF+oBCAFa3VXUuBoEV5Hl02  voofNPk8r9/alSLybaAxB1emxWTyYL42xHDeqf403R/FhWEeOSm48/s78KjR5HbQ  Gw+4T+gcywTCj0DXG3+Wdv9gt0eQNHxosaqeFrp62ZSKWNJ+us1PqXd5a9mZmpCR  /qXgQf4D27iZHwXZn706OPF1zmQf9QINt4S9+CoW+GqlcVdtWEXY+sbHFv2tbyco  oRPFVGO0TAIm3QoJzrFx0/iC0T/KwZ7NIHu7SUsKyEmzGidG5ynxEPv2bj7lgIgd  H8fgkEZe0sBqHsKNjyKNtGV5tehGO+Ve17qisDIc1wMNEQx5wZBnvTGgZuXf4JcM  glHkhr+oip2acA9xmDMEY/IdBRYJKwYBBAHaRw8BAQdAZVZCWLLCaG3kATvwhgK1  jeWFYc2h6VMqHqwY8p9StW20JU5pYWxsIEZpbmxheSA8bmlhbGwuZmlubGF5QGdt  YWlsLmNvbT6ImQQTFgoAQRYhBLOVzIBQ113FV2oyPfaXm6wX8ujUBQJj8h0FAhsD  BQkDwmcABQsJCAcCAiICBhUKCQgLAgQWAgMBAh4HAheAAAoJEPaXm6wX8ujU4TcA  /iPB3WFlLkuX1DUXnfqIkdk7frV34+2NMSmMoXi+wYU+APwN75ztJLx35z3bbTrT  mCepFjX2vS97HwdUUKBB526GBrg4BGPyHQUSCisGAQQBl1UBBQEBB0D1Xm2ItlbL  B/saLgyIlAxi44k/7U+s+ftDql+EtlsVAQMBCAeIfgQYFgoAJhYhBLOVzIBQ113F  V2oyPfaXm6wX8ujUBQJj8h0FAhsMBQkDwmcAAAoJEPaXm6wX8ujUla8A/RNNzlt+  2RSKwr/Y0xpMdlelsgcPe63o2uSZ/YUPf7aiAQCrqBhEJj/zd84GYrF3ok6kICrg  VI7QQQdj3PXZScmnDQ==  =fLXa  -----END PGP PUBLIC KEY BLOCK----- | How is the randomness generated?  We need to generate a lot of random bytes. It is a good idea to perform  some other action (type on the keyboard, move the mouse, utilize the  disks) during the prime generation; this gives the random number  generator a better chance to gain enough entropy.gpg  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**  also  gpg -k | 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**  **-----BEGIN PGP MESSAGE-----**  **hF4DHfXx3wW3PtwSAQdAtsIDQaAf2tnOZMAAINA2IVnmRglRy7UQ0ggVyrt3fxkw**  **XS/fu6J2f/eHf5MAF/vKcbPIqNPS+Pi5VVnRmwoaZu7aDL2zt/BX3DL2LBGFwk/D**  **1HQBCQIQqTdLsS4bTrWOsIowEjV9uPPG2Vt1cHqyiOVFj3jTVolwmqiuQFbr+T9r**  **4L7fvjQJkABong8KkEQcJIk2nL+Zh0dJ5+1fGdqjrrTG2BAkqjC/6osGjFNOeHuy**  **pvyM6DAbeJ7z3tJ7THmNexqYapeSzQ==**  **=LSTV**  **-----END PGP MESSAGE-----**  Base64 with BEGIN and END header and footer | What does the –a option do:  Specifies “armor” format  What does the –r option do:  Specifies recipient  What does the –u option do:  Specifies sender  Which file does it produce and outline the format of its contents:  Msg.asc file |
| **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: |
| **5** | Next using this public key file, send Bill ([w.buchanan@napier.ac.uk](mailto:w.buchanan@napier.ac.uk)) an encrypted question (**http://asecuritysite.com/public.txt**). | Did you receive a reply: |
| **6** | Next send your public key to Bill ([w.buchanan@napier.ac.uk](mailto:w.buchanan@napier.ac.uk)), and ask for an encrypted message from him. |  |

## 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 AAAAB3NzaC1yc2EAAAADAQABAAABAQDLrriuNYTyWuC1IW7H6yea3hMV+rm029m2f6IddtlImHrOXjNwYyt4Elkkc7AzOy899C3gpx0kJK45k/CLbPnrHvkLvtQ0AbzWEQpOKxI+tW06PcqJNmTB8ITRLqIFQ++ZanjHWMw2Odew/514y1dQ8dccCOuzeGhL2Lq9dtfhSxx+1cBLcyoSh/lQcs1HpXtpwU8JMxWJl409RQOVn3gOusp/P/0R8mz/RWkmsFsyDRLgQK+xtQxbpbodpnz5lIOPWn5LnT0si7eHmL3WikTyg+QLZ3D3m44NCeNb+bOJbfaQ2ZB+lv8C3OxylxSp2sxzPZMbrZWqGSLPjgDiFIBL [w.buchanan@napier.ac.uk](mailto: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):

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/krxy43cP0J9jlNz4dqxLgdiqtRe1AcymX06JUo1SrcqDEh3lQxoU1KUvV7jG9GE3pSxHq4dQlcWdHz95b9go6QYbe/5S/uJgolR+S9qaDE8tXYysP8FeXIPd0dXxHo=

The private key is:

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

MIICXQIBAAKBgQCfQfirYVXgzT90v6SqgeID7q/WK1XaVTNGVFolDUOcrXl/egRG

4iag5tiTbrMYCQ8CSTYn7q0U4AmBXihlbWDqf6MMk6OEoDxdWZTiG1MmQ1wZikFE

s7sYSog/poYleCeYW8kVzHNWnt9IuQWekIg6ZHkwp4NE/aW8HxvEwYRqCQIDAQAB

AoGAE6rkiFmxbt06GHNwZQQ8QssP2Q2qARgjiGxzY38DWg6MYiNR8uUL6zQHDBIQ

OQgpW9lpwD24D0tpsRnNOFVtMeafcxmykX+qHGtNeKJuTtqSm2eTI6gNbC8iosGT

XJEPM8tc/dfZ2sDobLfi0alWFOzWo8vKaLnnAdMHoZ8mDo8CQQDCMx08JVlTW1zl

+4UTEnyyYmIezw5ORfMqPtN1LpQ4ptYnHNMVJPWcpRwBYZfHlPOPtuVwo6gzv82G

QpgQsd4PAkEA0fA8e8R6JbeUR1HxsqWeCnPz3Ahq5Ya5WA6HyJQml9aDVqKDDp2L

3AcqsvFEKJ/T34r31so2yW6hj2yFBnzOZwJBAIqanrgJ1CpJYBGJJd6J6FQNIgjp

MUWuaTJyqsvNFd8lPF2oFgPWYDKQKV/W/tRkvD2LhVCSjf95WsADkbMAsAMCQAHo

wWQOwV2eccbERAJv5yQJMeqKWQ6FTyIx36I/VqqC1Obwy2hSnnb9ybGe6BPGgFLE

HMTjSeRDEU0Qm5UXhXkCQQCPlZJqlgksBN/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?  
128 bit key would take millions of years to crack, so within our lifetimes any message we could generate is safe from being cracked. The real problem isn’t the key length, it is in the Passphrase used to secure the key. If this isn’t truly random, then that lack of randomization decimates the key’s entropy and provides the most probable vector for attack.**