# 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.

🕮 **Web link (Weekly activities):** https://asecuritysite.com/esecurity/unit04

Demo: https://youtu.be/3n2TMpHqE18

## A RSA Encryption

**A.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=<br>q= |

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

| |
|---|
| N=<br>PHI = |

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

| |
|---|
| *e*= |

Now select a value of d, so that (e.d) (mod PHI) = 1:

| |
|---|
| *d*= |

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

| |
|---|
| C = $M^e$ (mod N) = |

Now decrypt your ciphertext with:

| |
|---|
| M = $C^d$ (mod N) = |

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

**A.2** The following defines a public key:

```
-----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+6og4OozohgkQb8OHox
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

**A.3** Bob has a private RSA key of:

```
MIICXAIBAAKBgQCwgjkeoyCXm9v6VBnUi5ihQ2knkdxGDL3GXLIUU43/froeqk7q9mtxT4AnPA
aDX3f2r4STZYYiqXGsHCUBZcI90dvZf6YiEM5OY2jgsmqBjf2Xkp/8HgN/XDw/wD2+zebYGLLY
td2u3GXx9edqJ8kQcU9LaMH+ficFQyfq9UwTjQIDAQABAoGAD7L1a6Ess+9b6G70gTANWkKJps
hVZDGb63mxKRepaJEX8sRJEqLqOYDNsC+pkKO8IsfHreh4vrp9bsZuECrB1OHSjwDBOS/fm3KE
WbsaaXDUAu0dQg/JBMXAKzeATreoIYJItYgwzrJ++fuquKabAZumvOnWJyBIs2z103kDz2ECQQ
Dnn3JpHirmgVdf81yBbAJaXBXNIPzOcCth1zwFAs4EvrE35n2HvUQuRhy3ahUKXsKX/bGvWzmC
2O6kbLTFEygVAkEAwxXZnPkaAY2vuoUCN5NbLZgegrAtmU+U2woa5A0fx6uXmShqxo1iDxEC71
FbNIgHBg5srsUyDj3OsloLmDVjmQJAIy7qLyOA+sCc6BtMavBgLx+bxCwFmsoZHOSX3l79smTR
AJ/HY64RREIsLIQ1q/yW7IWBzxQ5WTHgliNZFjKBvQJBAL3t/vCJwRz0Ebs5FaB/8UwhhsrbtX
lGdnkOjIGsmV0vHSf6poHqUiay/DV88pvhN11ZG8zHpeUhnaQccJ9ekzkCQDHHG9LYCOqTgsyY
ms//cW4sv2nuOE1UezTjUFeqOlsgO+WN96b/M5gnv45/Z3xZxzJ4HOCJ/NRwxNOtEUkw+zY=
```

And receives a ciphertext message of:

```
Pob7AQZZSml618nMwTpx3V74N45x/rTimUQeTl0yHq8F0dsekZgOT385Jls1HUzWCx6ZRFPFMJ
1RNYR2Yh7AkQtFLVx9lYDfb/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/rTimUQeTl0yHq8F0dsekZgOT385Jls1HUzWCx6ZR
FPFMJ1RNYR2Yh7AkQtFLVx9lYDfb/Q+SkinBIBX59ER3/fDhrVKxIN4S6h2QmMSRblh4KdVhyY
6cOxu+g48Jh7TkQ2Ig93/nCpAnYQ="
privatekey =
'MIICXAIBAAKBgQCwgjkeoyCXm9v6VBnUi5ihQ2knkdxGDL3GXLIUU43/froeqk7q9mtxT4AnP
AaDX3f2r4STZYYiqXGsHCUBZcI90dvZf6YiEM5OY2jgsmqBjf2Xkp/8HgN/XDw/wD2+zebYGLL
Ytd2u3GXx9edqJ8kQcU9LaMH+ficFQyfq9UwTjQIDAQABAoGAD7L1a6Ess+9b6G70gTANWkKJp
shVZDGb63mxKRepaJEX8sRJEqLqOYDNsC+pkKO8IsfHreh4vrp9bsZuECrB1OHSjwDB0S/fm3K
EWbsaaXDUAu0dQg/JBMXAKzeATreoIYJItYgwzrJ++fuquKabAZumvOnWJyBIs2z103kDz2ECQ
QDnn3JpHirmgVdf81yBbAJaXBXNIPzOcCth1zwFAs4EvrE35n2HvUQuRhy3ahUKXsKX/bGvWzm
C2O6kbLTFEygVAkEAwxXZnPkaAY2vuoUCN5NbLZgegrAtmU+U2woa5A0fx6uXmShqxo1iDxEC7
1FbNIgHBg5srsUyDj3OsloLmDVjmQJAIy7qLyOA+sCc6BtMavBgLx+bxCwFmsoZHOSX3l79smT
RAJ/HY64RREIsLIQ1q/yW7IWBzxQ5WTHgliNZFjKBvQJBAL3t/vCJwRz0Ebs5FaB/8Uwhhsrbt
XlGdnkOjIGsmV0vHSf6poHqUiay/DV88pvhN11ZG8zHpeUhnaQccJ9ekzkCQDHHG9LYCOqTgsy
Yms//cW4sv2nuOE1UezTjUFeqOlsgO+WN96b/M5gnv45/Z3xZxzJ4HOCJ/NRwxNOtEUkw+zY='

keyDER = b64decode(privatekey)
keys = RSA.importKey(keyDER)

dmsg = keys.decrypt(b64decode(msg))
print dmsg
```

| What is the plaintext message that Bob has been sent? |
|---|
| |

## B OpenSSL (RSA)

We will using OpenSSL to perform the following:

| No | Description | Result |
|---|---|---|
| **B.1** | First we need to generate a key pair with:<br><br>`openssl genrsa -out private.pem 1024`<br><br>This file contains both the public and the private key. | What is the type of public key method used:<br><br>How long is the default key:<br><br>How long did it take to generate a 1,024 bit key?<br><br>Use the following command to view the keys:<br><br>`type private.pem` (or `cat private.pem` in Linux) |
| **B.2** | Use following command to view the output file:<br><br>`cat private.pem` | What can be observed at the start and end of the file: |

| | | |
|---|---|---|
| **B.3** | Next we view the RSA key pair:<br><br>`openssl rsa -in private.pem -text` | Which are the attributes of the key shown:<br><br><br>Which number format is used to display the information on the attributes: |
| **B.4** | Let's now secure the encrypted key with 3-DES:<br><br>`openssl rsa -in private.pem -des3 -out key3des.pem` | |
| **B.5** | Next we will export the public key:<br><br>`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? |
| **B.6** | Now we will encrypt with our public key:<br><br>`openssl rsautl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin` | |
| **B.7** | And then decrypt with our private key:<br><br>`openssl rsautl -decrypt -inkey private.pem -in file.bin -out decrypted.txt` | What are the contents of decrypted.txt |

## 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:<br><br>`openssl ecparam -name secp256k1 -genkey -out priv.pem` | Can you view your key? |

|  |  |  |
|---|---|---|
|  | The file will only contain the private key (and should have 256 bits).<br><br>Now use "`cat priv.pem`" to view your key. |  |
| **C.2** | We can view the details of the ECC parameters used with:<br><br>`openssl ecparam -in priv.pem -text -param_enc explicit -noout` | Outline these values:<br><br>Prime (last two bytes):<br><br>A:<br><br>B:<br><br>Generator (last two bytes):<br><br>Order (last two bytes): |
| **C.3** | Now generate your public key based on your private key with:<br><br>`openssl ec -in priv.pem -text -noout` | How many bits and bytes does your private key have:<br><br>How many bit and bytes does your public key have (Note the 04 is not part of the elliptic curve point):<br><br>What is the ECC method that you have used? |

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/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):

How is the signature used in this example?

**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:

**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, NIST512p and SECP256k1:

NIST192p:

NIST512p:

SECP256k1:

By searching on the Internet, can you find where SECP256k1 is used?

What do you observe from the different hash signatures from the elliptic curve methods?

## E Inverse of a value mod N

**E.1** 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)** =

**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)?

## F 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?**

**2. If someone takes our elliptic curve public key, how might they determine our public key?**

## G 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.

## Notes

To setup your Python to run Python 2.7:

```
sudo update-alternatives --set python /usr/bin/python2.7
```

To install a Python library use:

```
easy_install libname
```

or:

```
pip install libname
```