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. 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

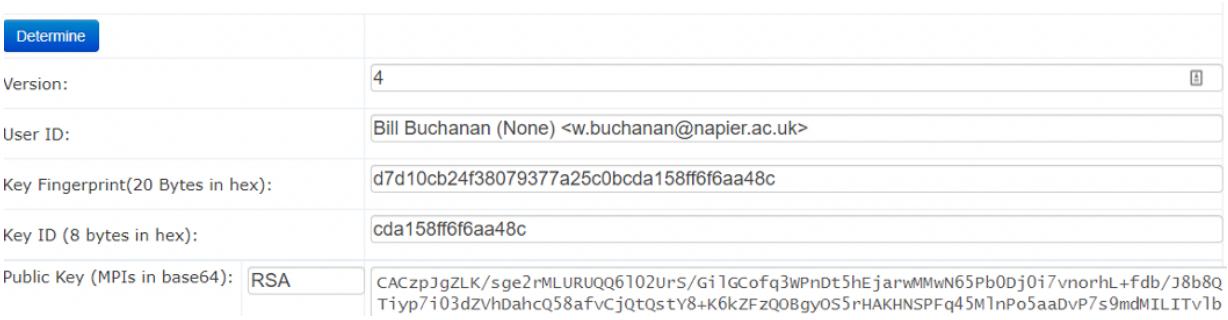
A RSA Encryption

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//g2GfzZe1UzH5Dz3PBbJky8GilfLm0OXSEIgAmpvc/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/fSKlqAdlz3mB
dbqWPjzPTY/m0lt+wv3epOM75uWjD35PF0rKxxZmEf6SrjZD1sk0B9bRy2v9iWN9
9ZkuvcfH4vT++PognQLTUqNx0FGpD1agrG0lXSCtJWQXCXPfWdtbldThBgzH4flZ
ssAIbCaBlQkzfbPvrMzdTIP+AXg6++K9SnO9N/FRPYzjUSEmpRp+ox31WymvczcU
RmyUquF+/zNnSBVgtY1rzwaYi05XfuxG0WHVHPTtRyJ5pF4HSqiuvk6Z/4z3bw==
=ZrP+
-----END PGP PUBLIC KEY BLOCK-----
```

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

https://asecuritysite.com/encryption/pgp1



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

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

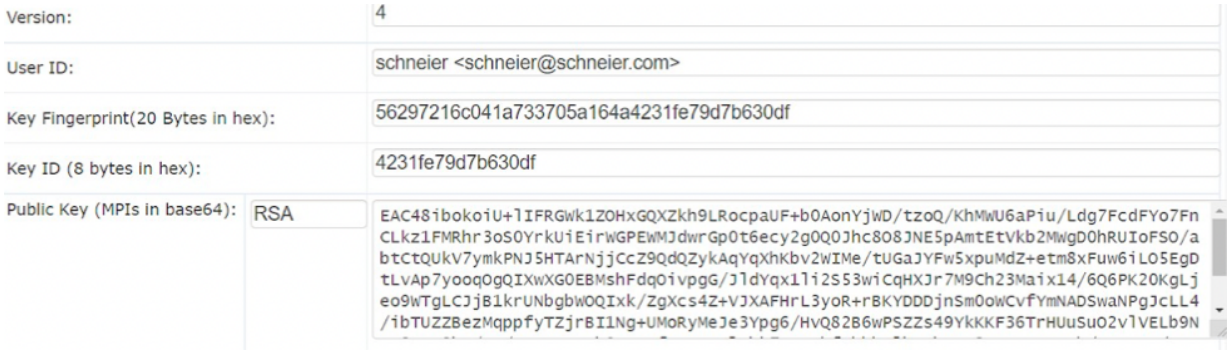
1) Tyler Cipriani

Algorithm: RSA

2) Tim Starling

Algorithm: RSA

3) Schneier



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. 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.

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 bits How long did it take to generate a 1,024 bit key? 0.23 seconds

		Use the following command to view the keys: cat private.pem
B.2	Use following command to view the output file: cat private.pem	What can be observed at the start and end of the file: **----BEGIN RSA PRIVATE KEY----** **And ----END RSA PRIVATE KEY----**
B.3	Next we view the RSA key pair: openssl rsa -in private.pem -text	Which are the attributes of the key shown: modulus, public Exponent, private Exponent, prime1, prime2,exponent1,exponent2,coef ficient Which number format is used to display the information on the attributes: Hexadecimal
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? 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.
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	View the output key. What does the header and footer of the file identify?
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

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+rm029m2f6IddtIImHrOXjNwYyt4Elkkc7AzO
y899C3gpx0kJK45k/CLbPnrHvkLvtQ0AbzWEQpOKxI+tW06PcqJNmTB8ITRLqIFQ++ZanjHWMw2Odew/514y1dQ8dccCO
uzeGhL2Lq9dtfhSxx+1cBLcyoSh/IQcs1HpXtpwU8JMxWJl409RQOVn3gOusp/P/0R8mz/RWkmsFsyDRLgQK+xtQxbpbo
dpnz5lIOPWn5LnT0si7eHmL3WikTyg+QLZ3D3m44NCeNb+bOJbfaQ2ZB+lv8C3OxylxSp2sxzPZMbrZWqGSLPjgDiFIBL
w.buchanan@napier.ac.uk

View the private key. Outline its format? It uses a rsa encryption with the associated email id appended at the end

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.

Image 1 openssl genrsa -out private.pem 1024

```
jashjain@Jashs-MacBook-Air exp4 % openssl genrsa -out private.pem 1024
Generating RSA private key, 1024 bit long modulus
..............................++++++
.................++++++
e is 65537 (0x10001)
jashjain@Jashs-MacBook-Air exp4 %
```

Image 2 cat private.pem

```
jashjain@Jashs-MacBook-Air exp4 % cat private.pem
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQC1W5+RRpHjnA/JAxFROay/UWg55NqgzPLE6uehFcsdISdWNsLw
I8G2Nomo6YHfxG6Vuf4xyaw8fkGsCpoK+UPXkP58zR+es37JOAGy6NmTaltLhMLC
ZPjKUkCx8ssbndv/AcY5Xz4o2yeBEWSWy1fTtMoSxEAWJePMBpwM/WgxIQIDAQAB
AoGAdTibktMCSRP+P/bzooSc26xKqeD6TyUpZY5P2Ra1Ckb7ngYXShBSjOwDXmJP
x8q3j5+AZb3szczttb3Eg4NKDVzaJbSuIQUhv9N4ZKQs6s6hfQUfe5HTV7bcbVXp
UfnK6/qJ+8XkF6F0y9qjTG/2S6NbmpAyh3+dThAOsZ7cI+ECQQDbpeVxPEK9t0Iq
gXMmao5Co5aIklOQe79Ec4NE9PJnD5hOibLtZjXWjHB+0dXV+idCdI9TmzM7hMtG
z8KFb/g/AkEA019w7gT5gT7FV1OG5j/vkhq/ugXJR8FlsTDl2oSNFMfSVkbI+Lvf
1r7brw4YAzmiCrrcw2JcsJ17X9kLY9J+nwJBAJIehMqVuhiXZy77uuzdihIO3b+x
CGJf0FfrOHKROQkSuNHWjO+dRUebWeMkGMsmhSOSgx6taqG3r7tFzzQRzuUCQQCY
ccMexxOztTPacXFfakk8VHJmlXm/eKoBC0UbPnIJxLvcYZzcV5Nua6RBMqb63HwT
xQUnHS/AaTPrwrvGOK/9AkAQHu2UuKig+R+uFQtNEp8o4HcCfib12V4z/RPeznDH
+sOapzaUoI2gBCrA73lWQN+xMAYDDh6Stl4Vlmntht4X
-----END RSA PRIVATE KEY-----
jashjain@Jashs-MacBook-Air exp4 %
```

Image 3 cat private.pem openssl rsa -in private.pem -text

```
publicExponent: 65537 (0x10001)
privateExponent:
    75:38:9b:92:d3:02:49:13:fe:3f:f6:f3:a2:84:9c:
    db:ac:4a:a9:e0:fa:4f:25:29:65:8e:4f:d9:16:b5:
    0a:46:fb:9e:06:17:4a:10:52:8c:ec:03:5e:62:4f:
    c7:ca:b7:8f:9f:80:65:bd:ec:cd:cc:ed:b5:bd:c4:
    83:83:4a:0d:5c:da:25:b4:ae:21:05:21:bf:d3:78:
    64:a4:2c:ea:ce:a1:7d:05:1f:7b:91:d3:57:b6:dc:
    6d:55:e9:51:f9:ca:eb:fa:89:fb:c5:e4:17:a1:74:
    cb:da:a3:4c:6f:f6:4b:a3:5b:9a:90:32:87:7f:9d:
    4e:10:0e:b1:9e:dc:23:e1
prime1:
    00:db:a5:e5:71:3c:42:bd:b7:42:2a:81:73:26:6a:
    8e:42:a3:96:88:92:53:90:7b:bf:44:73:83:44:f4:
    f2:67:0f:98:4e:89:b2:ed:66:35:d6:8c:70:7e:d1:
    d5:d5:fa:27:42:74:8f:53:9b:33:3b:84:cb:46:cf:
    c2:85:6f:f8:3f
prime2:
    00:d3:5f:70:ee:04:f9:81:3e:c5:57:53:86:e6:3f:
    ef:92:1a:bf:ba:05:c9:47:c1:65:b1:30:e5:da:84:
    8d:14:c7:d2:56:46:c8:f8:bb:df:d6:be:db:af:0e:
    18:03:39:a2:0a:ba:dc:c3:62:5c:b0:9d:7b:5f:d9:
    0b:63:d2:7e:9f
exponent1:
    00:92:1e:84:ca:95:ba:18:97:67:2e:fb:ba:ec:dd:
    8a:12:0e:dd:bf:b1:08:62:5f:d0:57:eb:38:72:91:
    39:09:12:b8:d1:d6:8c:ef:9d:45:47:9b:59:e3:24:
    18:cb:26:85:23:92:83:1e:ad:6a:a1:b7:af:bb:45:
    cf:34:11:ce:e5
exponent2:
    00:98:71:c3:1e:c7:13:b3:b5:33:da:71:71:5f:6a:
    49:3c:54:72:66:95:79:bf:78:aa:01:0b:45:1b:3e:
    72:09:c4:bb:dc:61:9c:dc:57:93:6e:6b:a4:41:32:
    a6:fa:dc:7c:13:c5:05:27:1d:2f:c0:69:33:eb:c2:
    bb:c6:38:af:fd
coefficient:
    10:1e:ed:94:b8:a8:a0:f9:1f:ae:15:0b:4d:12:9f:
    28:e0:77:02:7e:26:f5:d9:5e:33:fd:13:de:ce:70:
    c7:fa:c3:9a:a7:36:94:a0:8d:a0:04:2a:c0:ef:79:
    56:40:df:b1:30:06:03:0e:1e:92:b6:5e:15:96:69:
    ed:86:de:17
writing RSA key
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQC1W5+RRpHjnA/JAxFROay/UWg55NqgzPLE6uehFcsdISdWNsLw
I8G2Nomo6YHfxG6Vuf4xyaw8fkGsCpoK+UPXkP58zR+es37JOAGy6NmTaltLhMLC
ZPjKUkCx8ssbndv/AcY5Xz4o2yeBEWSWy1fTtMoSxEAWJePMBpwM/WgxIQIDAQAB
AoGAdTibktMCSRP+P/bzooSc26xKqeD6TyUpZY5P2Ra1Ckb7ngYXShBSjOwDXmJP
x8q3j5+AZb3szczttb3Eg4NKDVzaJbSuIQUhv9N4ZKQs6s6hfQUfe5HTV7bcbVXp
UfnK6/qJ+8XkF6F0y9qjTG/2S6NbmpAyh3+dThAOsZ7cI+ECQQDbpeVxPEK9t0Iq
gXMmao5Co5aIklOQe79Ec4NE9PJnD5hOibLtZjXWjHB+0dXV+idCdI9TmzM7hMtG
z8KFb/g/AkEA019w7gT5gT7FV1OG5j/vkhq/ugXJR8FlsTDl2oSNFMfSVkbI+Lvf
1r7brw4YAzmiCrrcw2JcsJ17X9kLY9J+nwJBAJIehMqVuhiXZy77uuzdihIO3b+x
CGJf0FfrOHKROQkSuNHWjO+dRUebWeMkGMsmhSOSgx6taqG3r7tFzzQRzuUCQQCY
ccMexxOztTPacXFfakk8VHJmlXm/eKoBC0UbPnIJxLvcYZzcV5Nua6RBMqb63HwT
xQUnHS/AaTPrwrvGOK/9AkAQHu2UuKig+R+uFQtNEp8o4HcCfib12V4z/RPeznDH
+sOapzaUoI2gBCrA73lWQN+xMAYDDh6Stl4Vlmntht4X
-----END RSA PRIVATE KEY-----
jashjain@Jashs-MacBook-Air exp4 %
```

Image 4 openssl rsa -in private.pem -des3 -out key3des.pem

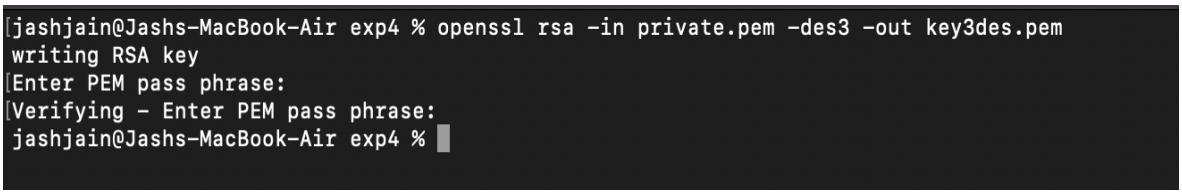


Image 5 openssl rsa -in private.pem -out public.pem -outform PEM -pubout

Image 6 openssl rsautl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin

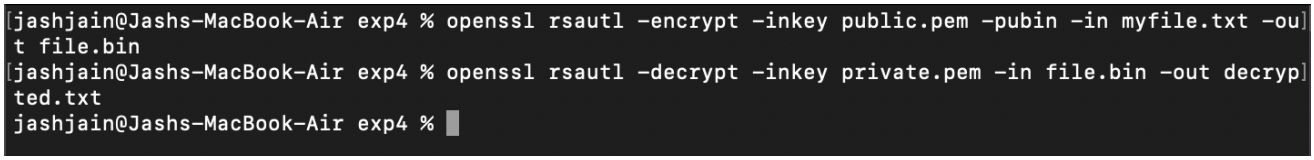


Image 7 Output of the files

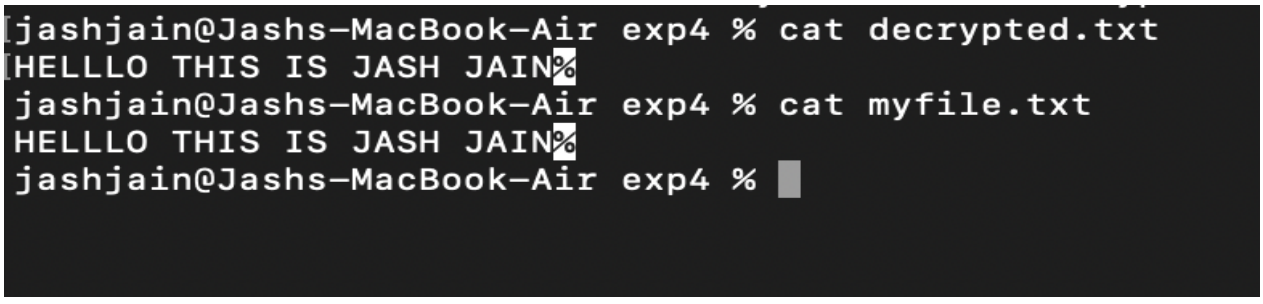


Image 8 ssh keygen

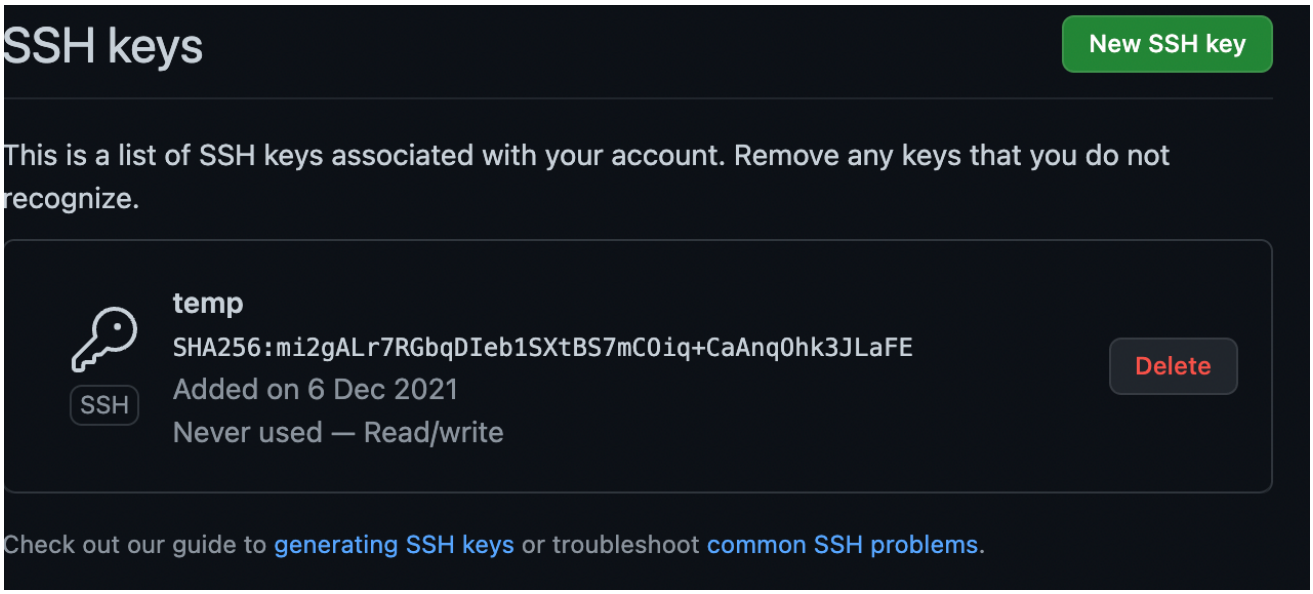
```
jashjain@Jashs-MacBook-Air exp4 % ssh-keygen -t rsa -C "jashjain21@gmail.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/jashjain/.ssh/id_rsa):
Created directory '/Users/jashjain/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/jashjain/.ssh/id_rsa
Your public key has been saved in /Users/jashjain/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:mi2gALr7RGbqDIeb1SXtBS7mCOiq+CaAnqOhk3JLaFE jashjain21@gmail.com
The key's randomart image is:
+---[RSA 3072]----+
|                 |
|                 |
|.  E  .          |
|+ .  o .         |
|*.+ = + S        |
|+@.* * =         |
|X+O o = .        |
|&&o    .         |
|&X=.             |
+----[SHA256]-----+
jashjain@Jashs-MacBook-Air exp4 %
```

Image 9 Ouput of public key

```
jashjain@Jashs-MacBook-Air exp4 % cat /Users/jashjain/.ssh/id_rsa
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAABlwAAAAdzc2gtcn
NhAAAAAwEAAQAAAYEAw6/7K9ZYOPMnWxCPVt6b2W1FaqdOXRB6jDu+g4eoiOBpETN2J04L
3qckhePgnNb51dFubXLqHxaPrjVhZ389NYG7r5y/FMo23J9dPzZbvOjhSINIqgS9G6jE1h
5VmNKsM72PObbaF132qvzZL3BV+OOZ5cC+SGjtgenWCkvnnQRmhWlXsbO9Ml/z5Mo28eVH
RkWZVUuoM/+M8xWKBUuaLV/Y1gImGIc6bCoe1c9/PPSZpjaqE70zRuFMwI6oylnVEm2e4x
T8mrA5jpjY45gN1eHMXdRIdrZ0mnSI3tOv3G5bqDZ50p/B7MVxkMAtma0Ow28n/Q/wmnWe
42hpcrskf1w0Ft5Zq1C3zFbx8YC87Nyxmki+jQqSSAGqqJSsHOVJk6niA/AedjbpEMFfWz
Udo9v8jDbxGtTScaTh+v31BgeXxIiUu3GUgvSDNDoeNAs52VEg2axeqY3k6SPJIoxj2WyQ
KTcFqIdGkn+FPUrgXWaNAqujPpuCUkpuS7mTph+vAAAFkKG4uuehuLrnAAAAB3NzaC1yc2
EAAAGBAMOv+yvWWDjzJ1sQj1bem9ltRWqnTl0Qeow7voOHqIjgaREzdidOC96nJIXj4JzW
+dXRbm1y6h8Wj641YWd/PTWBu6+cvxTKNtyfXT82W7zo4UiDSKoEvRuoxNYeVZjSrDO9jz
m22hdd9qr82S9wVfjjmeXAvkho7YHp1gpL550EZoVpV7GzvTJf8+TKNvHlR0ZFmVVLqDP/
jPMVigVLmi1f2NYCJhiHOmwqHtXPfzz0maY2qhO9M0bhTMCOqMpZ1RJtnuMU/JqwOY6Y2O
OYDdXhzF3USHa2dJp0iN7Tr9xuW6g2edKfwezFcZDALZmtDsNvJ/0P8Jp1nuNoaXK7JH9c
NBbeWatQt8xW8fGAvOzcsZpIvo0KkkgBqqiUrBzlSZOp4gPwHnY26RDBX1s1HaPb/Iw28R
rU0nGk4fr99QYHl8SIlLtxlIL0gzQ6HjQLOdlRINmsXqmN5OkjySKMY9lskCk3BaiHRpJ/
hT1K4F1mjQKroz6bglJKbku5k6YfrwAAAAMBAAEAAAGAF4/R61ZUqw7D91gIGkqVc1mP2S
ap7dS1HLt41fd4mHqEzBhpKpRgI3/2itSET/4meNNNu4nxnKqnxC8TOlX8tbuYMK3r6AHt
FyVtIdZWt2lG7uc30ZjeeD0a7jkaz1rWdyhk3Wsh2Vxr/eyR7f9VqrEk76AEiU1xA4p3Ri
gbiDEicr/bWi8tEaqvACNE+F1E9rUcoEQKzi//9ajrRHX062+1+RarJZrqsuljaxUf2eQN
```

Image 10 cat id_rsa.pub

Image 11 ssh key upload on github



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 (and should have 256 bits). Now use "cat priv.pem" to view your key.	Can you view your key? Yes openssl ecparam -in priv.pem - text - param_enc explicit -noout
C.2	We can view the details of the ECC parameters used with:	

Prime (last two bytes):

A: fc:2f

B : 7

Generator (last two bytes): d4:b8

Outline these values:

		Order (last two bytes):
C.3	Now generate your public key based on	
	your private key with: openssl ec -in priv.pem -text -noout	Order (last two bytes): How many bits and bytes does your private key have: 256 bits How many bit and bytes does your public key have (Note the 04 is not part of the elliptic curve point): 64 bytes and 512 bits What is the ECC method that you have used? secp256k1

Image 11 : Openssl ecparam -name secp256k1 -genkey - out priv.pem

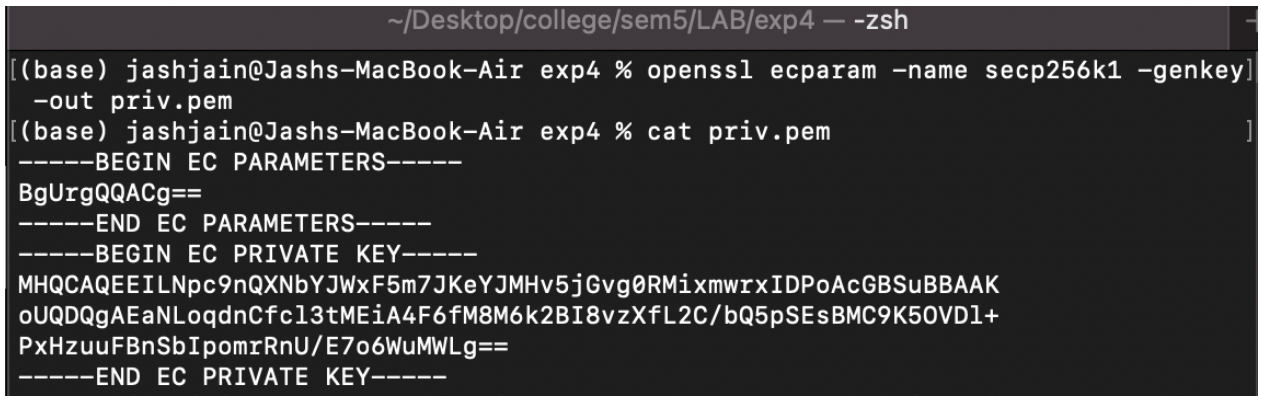


Image 12 : openssl ecparam -in priv.pem -text - param_enc explicit -noout

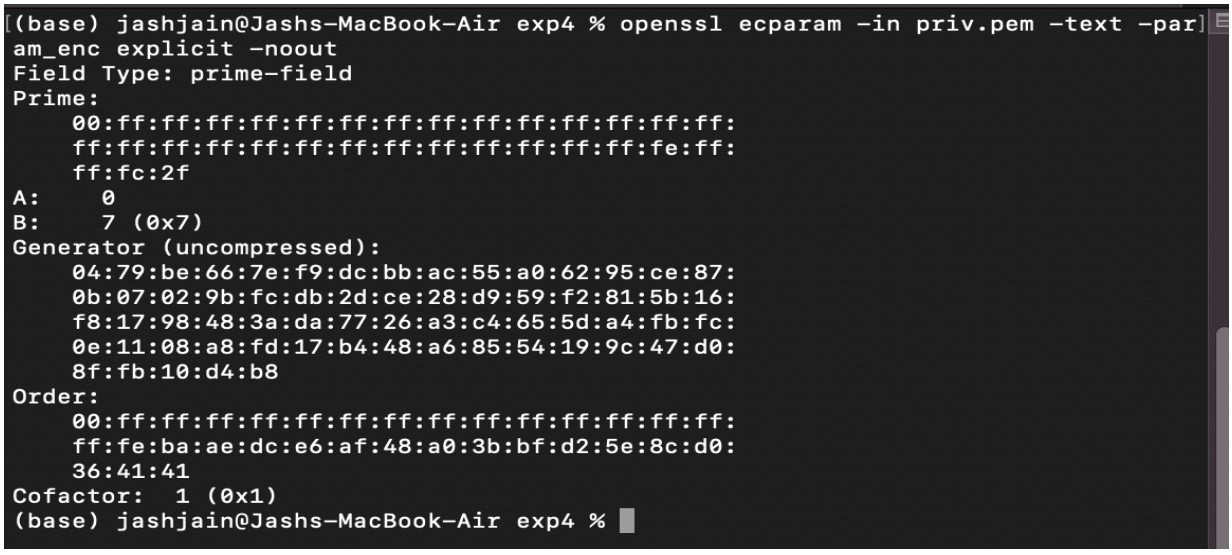


Image 13 : openssl ecparam -in priv.pem -text – nout read EC key

```
(base) jashjain@Jashs-MacBook-Air exp4 % openssl ec -in priv.pem -text -noout
read EC key
Private-Key: (256 bit)
priv:
    b3:69:73:d9:d0:5c:d6:d8:25:6c:45:e6:6e:c9:29:
    e6:09:30:7b:f9:8c:6b:e0:d1:13:22:c6:6c:2b:c4:
    80:cf
pub:
    04:68:d2:e8:a9:d9:c2:7d:c9:77:b4:c1:22:03:81:
    7a:7c:cf:0c:ea:4d:81:23:cb:f3:5d:f2:f6:0b:f6:
    d0:e6:94:84:b0:13:02:f4:ae:4e:54:39:7e:3f:11:
    f3:ba:e1:41:9d:26:c8:a6:89:ab:46:75:3f:13:ba:
    3a:5a:e3:16:2e
ASN1 OID: secp256k1
(base) jashjain@Jashs-MacBook-Air exp4 %
```

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

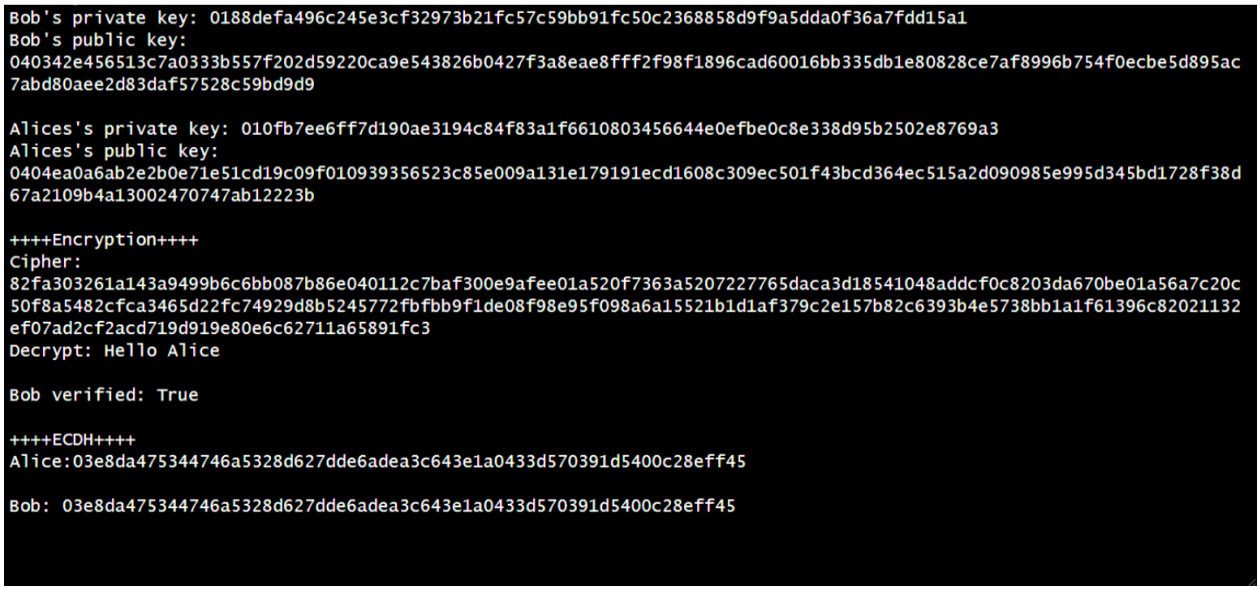
D Elliptic Curve Encryption

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):82fa

How is the signature used in this example? The signature is used to verify Bob's identity which can be done using his public key

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_point

s First five points: (1,39) (1,50) (3,52) (3,37)

(4,31)

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: r44L0nLLZ9BBbiOEv8+LWLD7wr213HSmbcyGkQlCNpgx9yyeOjwyOByqHDKR//hB

NIST521p:
AbjAlbn7o464qPxvLR0vmTLc/cf1XqdrsXDDb9q4RhCqbFqZFlcHfbauV0V5bJFjcULpAPx
MxPq4MPiT6ABLfEkiAD4Gtqe/

B7BRsFzCUE2TsqrWu3DnwhRjDJ9sCTcd13xwcIg2iYjNnVSeihYgDK/
rQzrxG4LGNh8KctNipDeNu1WX

SECP256k1:
H7qyUwtMO8hmAYDq4uBRQ8pdCzoV3d9QRnt8WfwYkknBWNXcps4xnrWCBGDOX+20DvSHTY4hC2DjmDM5k49
xpg==

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

secp256k1 refers to the parameters of the elliptic curve used in Bitcoin's public-key cryptography. So it finds application in Bitcoin. 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.

E **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 = 19
Q = 7

```
p
```

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

```
N= 19* 7 = 133
PHI = 108
```

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= 65
```

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

```
C = M^e (mod N) = 5^5(mod 133) = 66
```

Now decrypt your ciphertext with:

```
M = C^d (mod N) = 66^65(mod 133) = 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=11
q=3
N=p*q
```

```
            if ((e*d % PHI)==1): break
print e,N
print d,N
M=4
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
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)

```

```
5 133
65 133
66
5
>
```

```
p=23
q=7
N=p*q
PHI=(p-1)*(q-1)
e=7
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)

```

```
7 161
19 161
40
5
>
```

```
1  p= 107
2  q=7
3  N=p*q
4  PHI=(p-1)*(q-1)
5  e=7
6  for d in range(1,100):
7      if ((e*d % PHI)==1): break
8  print(e,N)
9  print(d,N)
10 M=5
11 cipher = (M**e) % N
12 print(cipher)
13 message = (cipher**d) % N
14 print(message)
15
```

```
7 749
91 749
229
5
>
```

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:

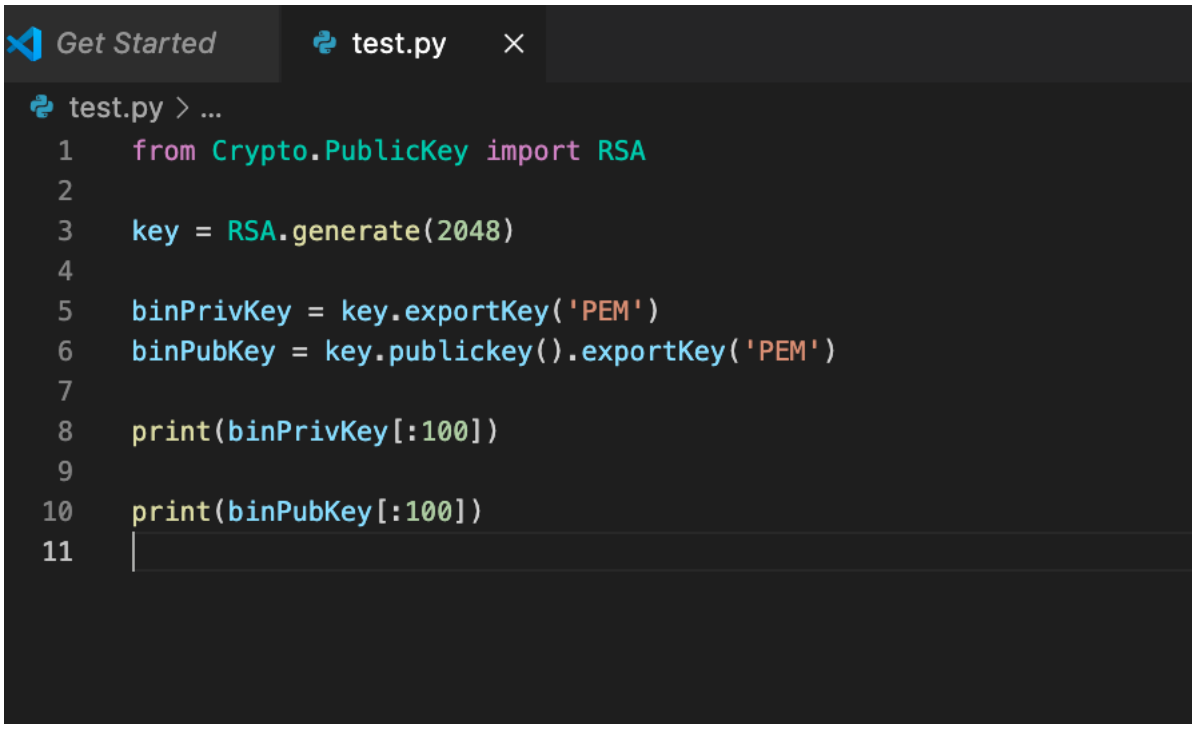
Inverse of 53 (mod 120) = 77

Inverse of 65537 (mod 1034776851837418226012406113933120080) = 568411228254986589811047501435713

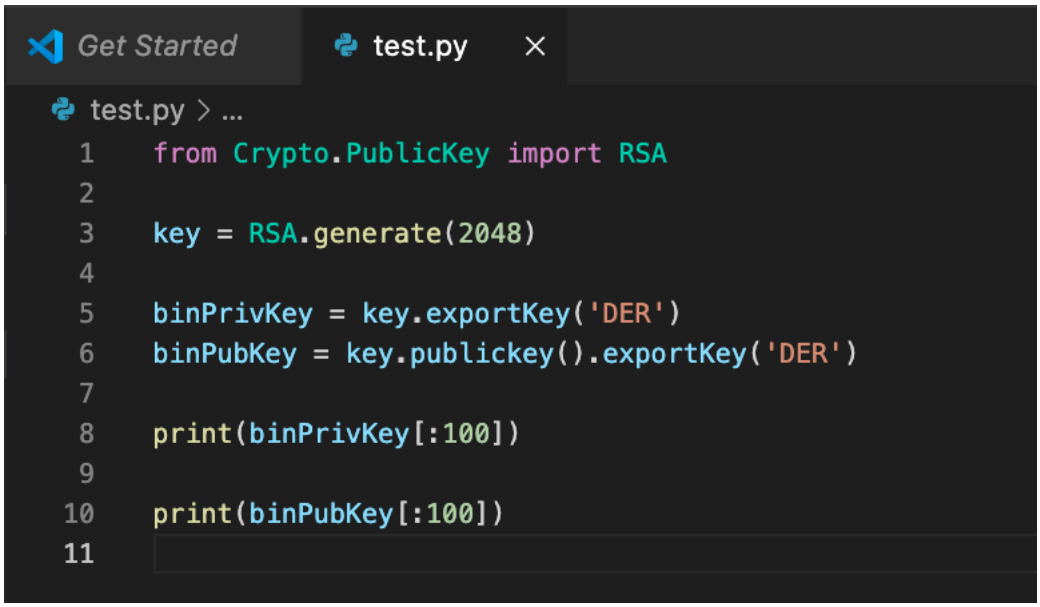
Using this code, can you now create an RSA program where the user enters the values of p, q, and e, and the program determines (e,N) and (d,N)?

E.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
```




```
(prod1.1) jashjain@Jashs-MacBook-Air exp4 % python test.py
b'-----BEGIN RSA PRIVATE KEY-----\nMIIEogIBAAKCAQEAuKbQCDS960+uGtq4pM1BUoKQ3k/4HDkcIAzK2ZMWC5NKvMeV\nBq3'
b'-----BEGIN RSA PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAuKbQCDS960+uGtq4pM1B\nUoKQ'
(prod1.1) jashjain@Jashs-MacBook-Air exp4 %
```



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. PEM is a DER file converted to base64 and adding the headers.

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:

-----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-----

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	
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

Result

How is the randomness generated?
Randomness is generated by using pseudorandom number generation

Outline the contents of your key file:

The file contains header and footer with naming conventions begin and end

2	Now send your lab partner your public key in the contents an email, and ask them to import it onto their key ring (if you are doing this on your own, create another set of keys to simulate another user, or use Bill's public key – which is defined at http://asecuritysite.com/public.txt and send the email to him): **gpg --import** *theirpublickey**.key*** *Now list your keys with:* **gpg --list-keys**	Which keys are stored on your key ring and what details do they have: It displays the UID,email and the public key of a user. It also displays the 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: Creates ASCII armored output What does the –r option do: Encrypt for user id name What does the –u option do: Use name as the user ID to sign
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**	Which file does it produce and outline the format of its contents: It produces a .asc file which has a header and footer indicating a pgp file Can you decrypt the message: Ye

Next send your public key to Bill (w.buchanan@napier.ac.uk), and ask for an encrypted message from him.

Image 14: gpg –gen -key

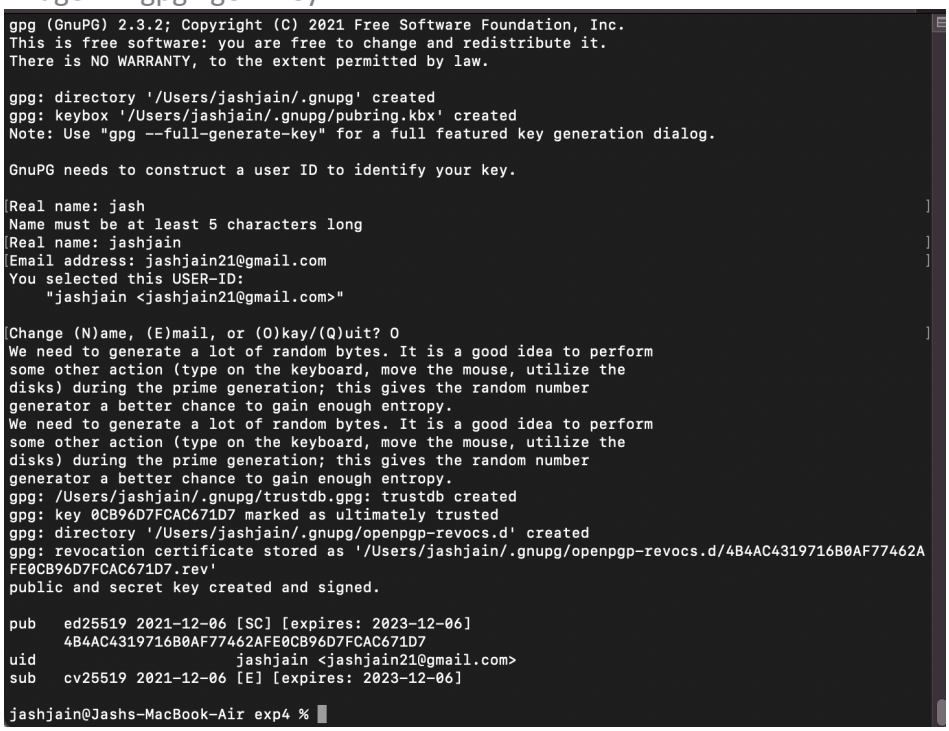


Image 15: gpg --export -a "jash" > mypub.key

```
jashjain@Jashs-MacBook-Air exp4 % gpg --export -a "jash" > mypub.key
jashjain@Jashs-MacBook-Air exp4 % gpg --export-secret-key -a "jash" > mypriv.key
jashjain@Jashs-MacBook-Air exp4 % ls
mypriv.key      mypub.key       priv.pem        test.py
jashjain@Jashs-MacBook-Air exp4 % cat mypriv.key
-----BEGIN PGP PRIVATE KEY BLOCK-----

lIYEYa2aYxYJKwYBBAHaRw8BAQdAfgjepuL399l0TxEmsyCNkf7uVPhwDZp1+NDx
wSQuxxj+BwMC2nxpzFsKAOz7v/Brnt3J7dBvhJ94lrr0iX766+Lo+ro0AwiKHtQA
Mztbqx0Pyr4WC48jiIQXEfXSTn7KZxw9LJcJ5USUwxdZx5Oy0tlCI7QfamFzaGph
aW4gPGphc2hqYWluMjFAZ21haWwuY29tPoiaBBMWCgBCFiEES0rEMZcWsK93Rir+
DLltf8rGcdcFAmGtmmMCGwMFCQPCZwAFCwkIBwIDIgIBBhUKCQgLAgQWAgMBAh4H
AheAAAoJEAy5bX/KxnHXVBIBAPTYZ7Mo5SmeFUZ7YH7g3jtNWWAiBK+KwANslgsU
Kp1zAQDndPi4ajKaKzH6yK/sYNz8u8J0IPkn3TpcATyHAuifDZyLBGGtmmMSCisG
AQQBl1UBBQEBB0CWL1qR1Uk/w6Rjk4CStwuqHjfhmOWtrqYTpJQ+uRrLKgMBCAf+
BwMCx8UXc8r+CQn7ssSWDXeogngZIHO4ZyFYYFnAPUWyTAULKIjulVaYi/k+3YMR
aKuZcWU/r3zDo9RGJLGXwolH1bAViFVhFhBnRkRbpASSBoh+BBgWCgAmFiEES0rE
MZcWsK93Rir+DLltf8rGcdcFAmGtmmMCGwwFCQPCZwAACgkQDLltf8rGcddftAEA
whpvATVIH0iRAcDsSU8UFkb0RTZpT8DJv9/GsODT2g0A/1/5fTOGHGfYU14FYFPH
RFWAgVKNpJwtVfFLAN3j/SYJ
=/p5D
-----END PGP PRIVATE KEY BLOCK-----
jashjain@Jashs-MacBook-Air exp4 % cat mypub.key
-----BEGIN PGP PUBLIC KEY BLOCK-----

mDMEYa2aYxYJKwYBBAHaRw8BAQdAfgjepuL399l0TxEmsyCNkf7uVPhwDZp1+NDx
wSQuxxi0H2phc2hqYWluIDxqYXNoamFpbjIxQGdtYWlsLmNvbT6ImgQTFgoAQhYh
BEtKxDGXFrCvd0Yq/gy5bX/KxnHXBQJhrZpjAhsDBQkDwmcABQsJCAcCAyICAQYV
CgkICwIEFgIDAQIeBwIXgAAKCRAMuW1/ysZx11QSAQD02GezKOUpnhVGe2B+4N47
TVlgIgSvisADbJYLFCqdcwEA53T4uGoymisx+siv7GDc/LvCdCD5J906XAE8hwLo
nw24OARhrZpjEgorBgEEAZdVAQUBAQdAli9akdVJP8OkY5OAkrcLqh434Zjlra6m
E6SUPrkayyoDAQgHiH4EGBYKACYWIQRLSsQxlxawr3dGKv4MuW1/ysZx1wUCYa2a
YwIbDAUJA8JnAAAKCRAMuW1/ysZx11+0AQDCGm8BNUgfSJEBwOxJTxQWRvRFNmlP
wMm/38aw4NPaDQD/X/l9M4YcZ9hTXgVgU8dEVYCBUo2knC1V8UsA3eP9Jgk=
=x2gz
-----END PGP PUBLIC KEY BLOCK-----
jashjain@Jashs-MacBook-Air exp4 %
```

Image 15: generate another key pair

```
jashjain@Jashs-MacBook-Air exp4 % gpg --gen-key
gpg (GnuPG) 2.3.2; Copyright (C) 2021 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Note: Use "gpg --full-generate-key" for a full featured key generation dialog.

GnuPG needs to construct a user ID to identify your key.

Real name: nitin
Email address: nmjain99@gmail.com
You selected this USER-ID:
    "nitin <nmjain99@gmail.com>"

Change (N)ame, (E)mail, or (O)kay/(Q)uit? O
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.
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: key 6B0AECFA172DF6CF marked as ultimately trusted
gpg: revocation certificate stored as '/Users/jashjain/.gnupg/openpgp-revocs.d/E686F9092627C0CD7CE4E5
A06B0AECFA172DF6CF.rev'
public and secret key created and signed.

pub   ed25519 2021-12-06 [SC] [expires: 2023-12-06]
      E686F9092627C0CD7CE4E5A06B0AECFA172DF6CF
uid                      nitin <nmjain99@gmail.com>
sub   cv25519 2021-12-06 [E] [expires: 2023-12-06]
```

```
jashjain@Jashs-MacBook-Air exp4 % gpg --export -a "nitin" > mypub2.key
jashjain@Jashs-MacBook-Air exp4 % gpg --export-secret-key -a "nitin" > mypriv2.key
jashjain@Jashs-MacBook-Air exp4 % gpg --import mypub2.key
gpg: key 6B0AECFA172DF6CF: "nitin <nmjain99@gmail.com>" not changed
gpg: Total number processed: 1
gpg:              unchanged: 1
jashjain@Jashs-MacBook-Air exp4 %
```

Image 16: Listing the keys

```
gpg: revocation certificate stored as '/c/Users/KashMir/.gnupg/openpgp-revocs.d/5FE4FA19310EBFD709B9E0FC49CB20E8A639360E
.rev'
public and secret key created and signed.

pub   rsa2048 2021-11-26 [SC] [expires: 2023-11-26]
      5FE4FA19310EBFD709B9E0FC49CB20E8A639360E
uid                   Kashish <kashishjain32@gmail.com>
sub   rsa2048 2021-11-26 [E] [expires: 2023-11-26]
```

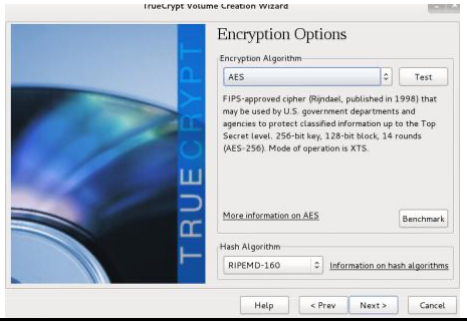
Image 17: Encrypt using partner's public key

```
jashjain@Jashs-MacBook-Air exp4 % gpg -e -a -u "jashjain" -r "nitin" hello.txt
jashjain@Jashs-MacBook-Air exp4 % ls
hello.txt       mypriv.key      mypub.key       priv.pem
hello.txt.asc   mypriv2.key     mypub2.key      test.py
jashjain@Jashs-MacBook-Air exp4 %
```

Image 18: Decrypt and see the ouput

```
jashjain@Jashs-MacBook-Air exp4 % gpg -e -a -u "nitin" -r "jashjain21" hello2.txt
jashjain@Jashs-MacBook-Air exp4 % gpg -d hello2.txt.asc
gpg: encrypted with cv25519 key, ID FBFEAB9B880930E1, created 2021-12-06
      "jashjain <jashjain21@gmail.com>"
Hello this is nitin jain%
jashjain@Jashs-MacBook-Air exp4 %
```

G TrueCrypt

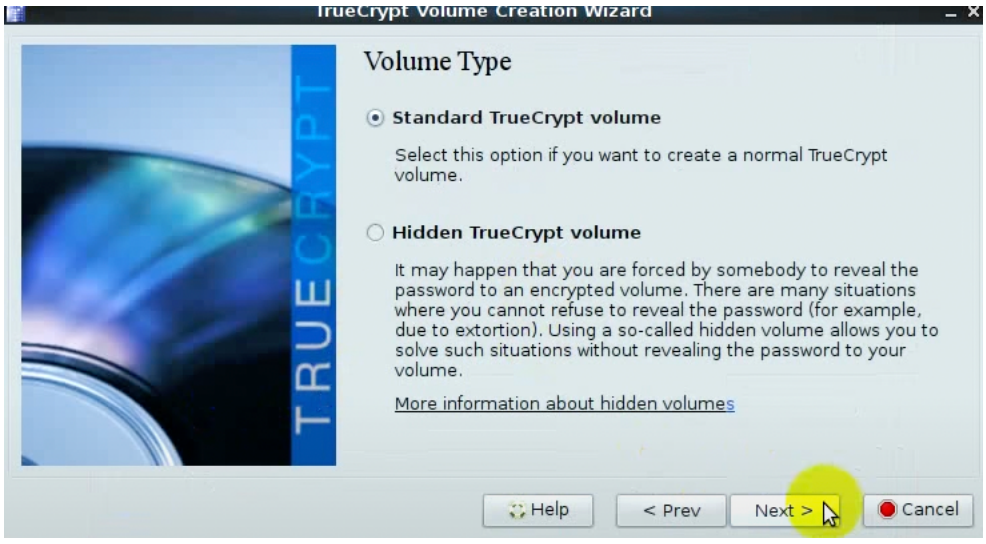
No	Description	Result
1	Go to your **Kali** instance (User: root, Password: toor). Now **Create a new volume** and use an **encrypted file container** (use tc_*yourname*) with a Standard TrueCrypt volume. When you get to the Encryption Options, run the benchmark tests and outline the results: 	CPU (Mean) AES: AES-Twofish: AES-Two-Seperent Serpent -AES Serpent: Serpent-Twofish-AES Twofish: Twofish-Serpent: Which is the fastest: Which is the slowest:

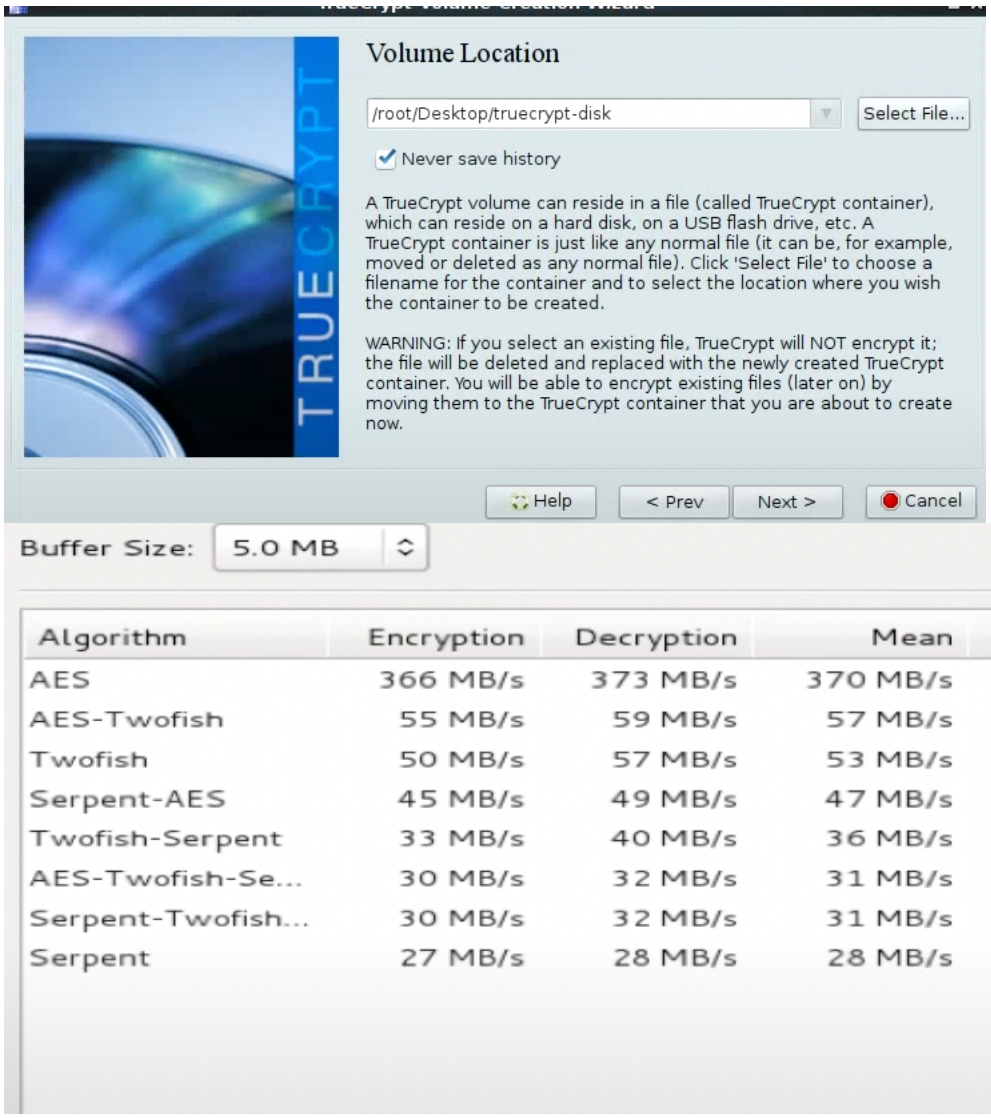
2	Select AES and RIPMD-160 and create a 100MB file. Finally select your password and use FAT for the file system.	What does the random pool generation do, and what does it use to generate the random key?
3	Now mount the file as a drive.	Can you view the drive on the file viewer and from the console? [Yes][No]
4	Create some files your TrueCrypt drive and save them.	Without giving them the password, can they read the file? With the password, can they read the files?

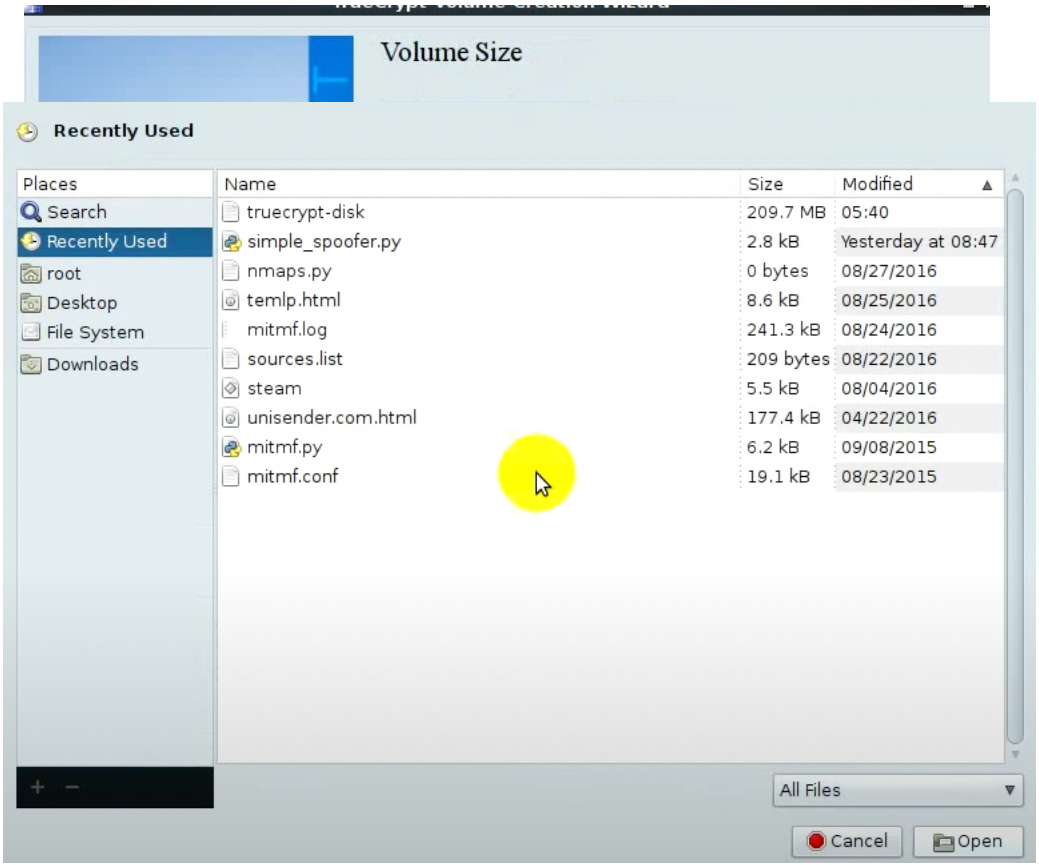
The following files have the passwords of “Ankle123”, “foxtrot”, “napier123”, “password” or “napier”. Determine the properties of the files defined in the table:

File	Size	Encryption type	Key size	Files/folders on disk	Hidden partition (y/n)	Hash method
http://asecuritysite.com/tctest01.zip						
http://asecuritysite.com/tctest02.zip						
http://asecuritysite.com/tctest03.zip						

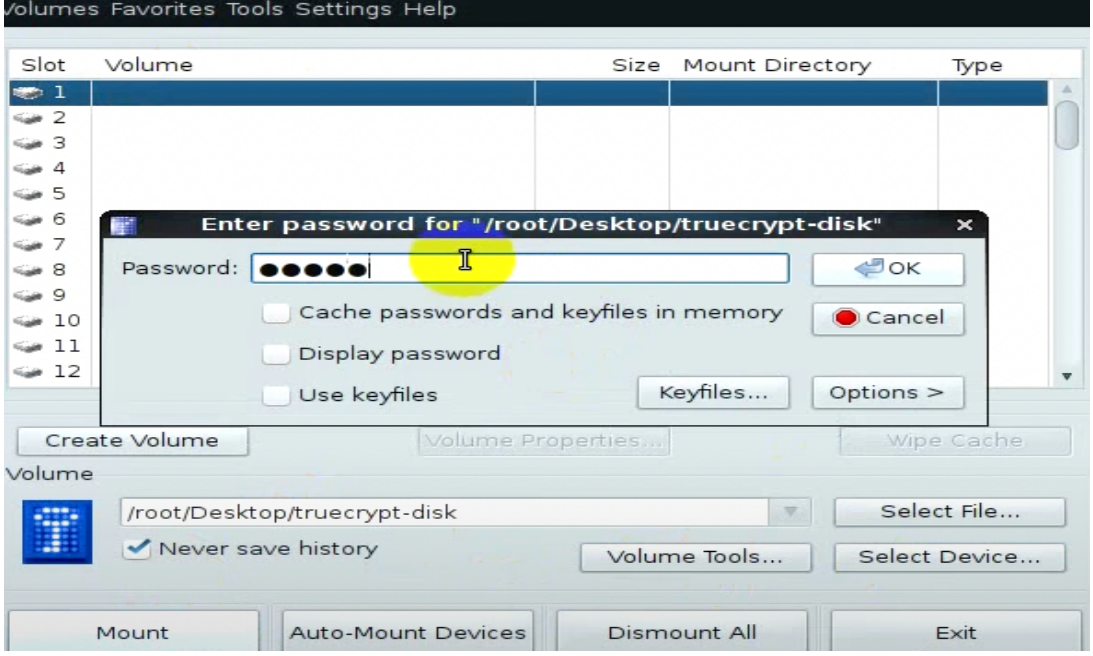
Now with **truecrack** see if you can determine the password on the volumes. Which TrueCrypt volumes can truecrack?

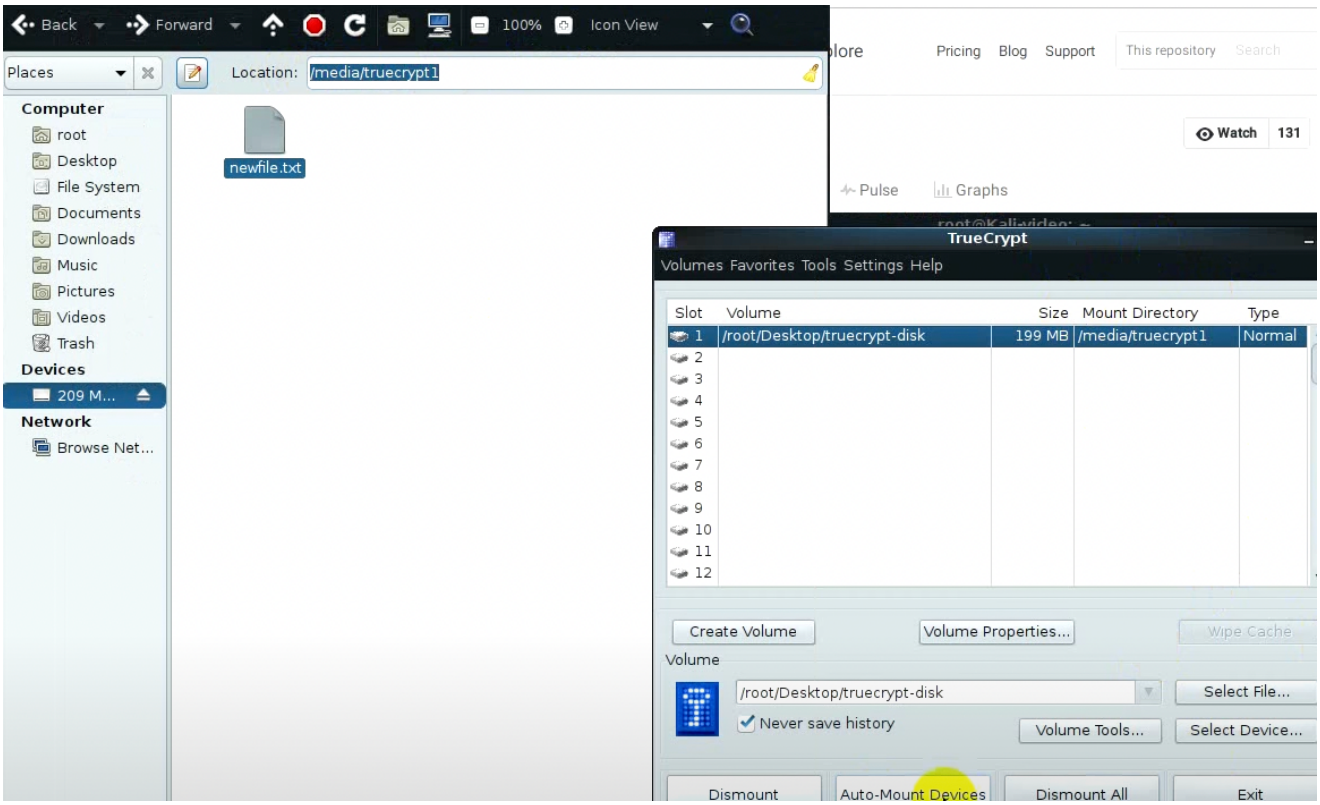



Volume Location
/root/Desktop/truecrypt-disk
Select File...
Never save history
A TrueCrypt volume can reside in a file (called TrueCrypt container), which can reside on a hard disk, on a USB flash drive, etc. A TrueCrypt container is just like any normal file (it can be, for example, moved or deleted as any normal file). Click 'Select File' to choose a filename for the container and to select the location where you wish the container to be created.
WARNING: If you select an existing file, TrueCrypt will NOT encrypt it; the file will be deleted and replaced with the newly created TrueCrypt container. You will be able to encrypt existing files (later on) by moving them to the TrueCrypt container that you are about to create now.
Help
< Prev
Next >
Cancel
Buffer Size: 5.0 MB
Algorithm Encryption Decryption Mean
AES 366 MB/s 373 MB/s 370 MB/s
AES-Twofish 55 MB/s 59 MB/s 57 MB/s
Twofish 50 MB/s 57 MB/s 53 MB/s
Serpent-AES 45 MB/s 49 MB/s 47 MB/s
Twofish-Serpent 33 MB/s 40 MB/s 36 MB/s
AES-Twofish-Se... 30 MB/s 32 MB/s 31 MB/s
Serpent-Twofish... 30 MB/s 32 MB/s 31 MB/s
Serpent 27 MB/s 28 MB/s 28 MB/s


Volume Size
Recently Used
Places
Search
Recently Used
root
Desktop
File System
Downloads
Name Size Modified
truecrypt-disk 209.7 MB 05:40
simple_spoofer.py 2.8 kB Yesterday at 08:47
nmaps.py 0 bytes 08/27/2016
templ.html 8.6 kB 08/25/2016
mitmf.log 241.3 kB 08/24/2016
sources.list 209 bytes 08/22/2016
steam 5.5 kB 08/04/2016
unisender.com.html 177.4 kB 04/22/2016
mitmf.py 6.2 kB 09/08/2015
mitmf.conf 19.1 kB 08/23/2015
All Files
Cancel
Open

H
H
H
H
H





H 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, I don't believe it's big enough. Before Bitcoin became popular, Secp256k1 (which employs a 256-bit private key) was almost never used, but it is now gaining favour due to a variety of advantages. The majority of commonly used curves have a random structure, however secp256k1 was created in a non-random manner, allowing for incredibly fast calculation. As a result, if the implementation is properly optimised, it can sometimes be more than 30% faster than other curves. Furthermore, unlike famous NIST curves, secp256k1's constants were set in a predictable manner, limiting the

chances that the curve's inventor included any kind of backdoor. If the cracker verifies these many keys in one second and the private key size possibilities become 2256, which is approximately 1.2e+77 keys, then breaking the 256 bit private key is not a tough procedure; the key can be cracked in 8 seconds.

I 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.