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

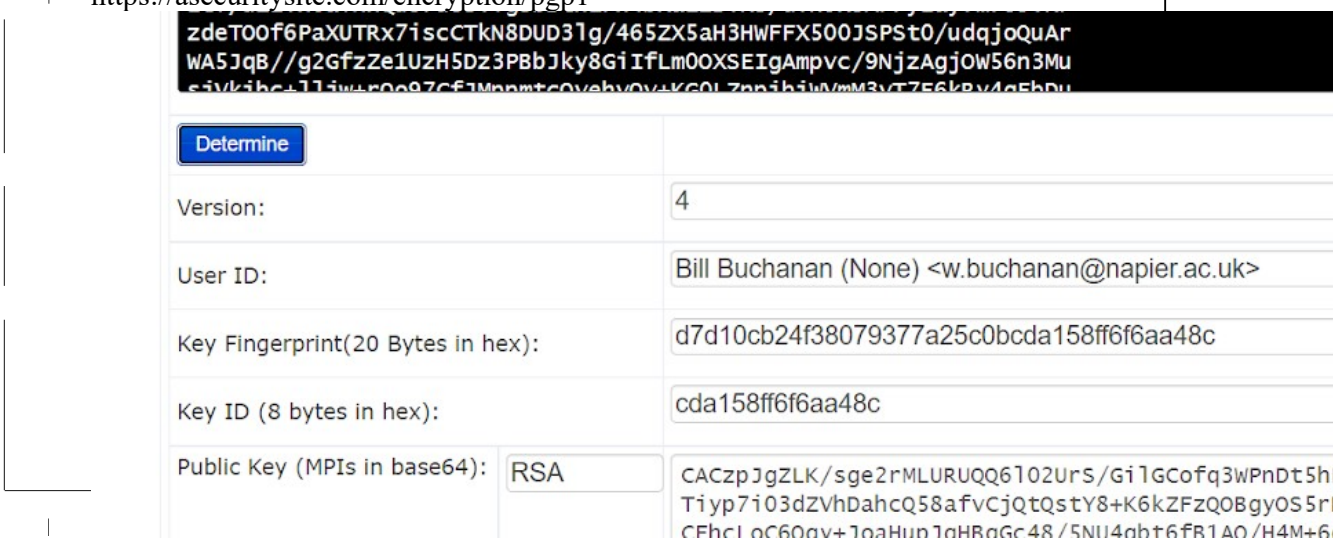
A RSA Encryption

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

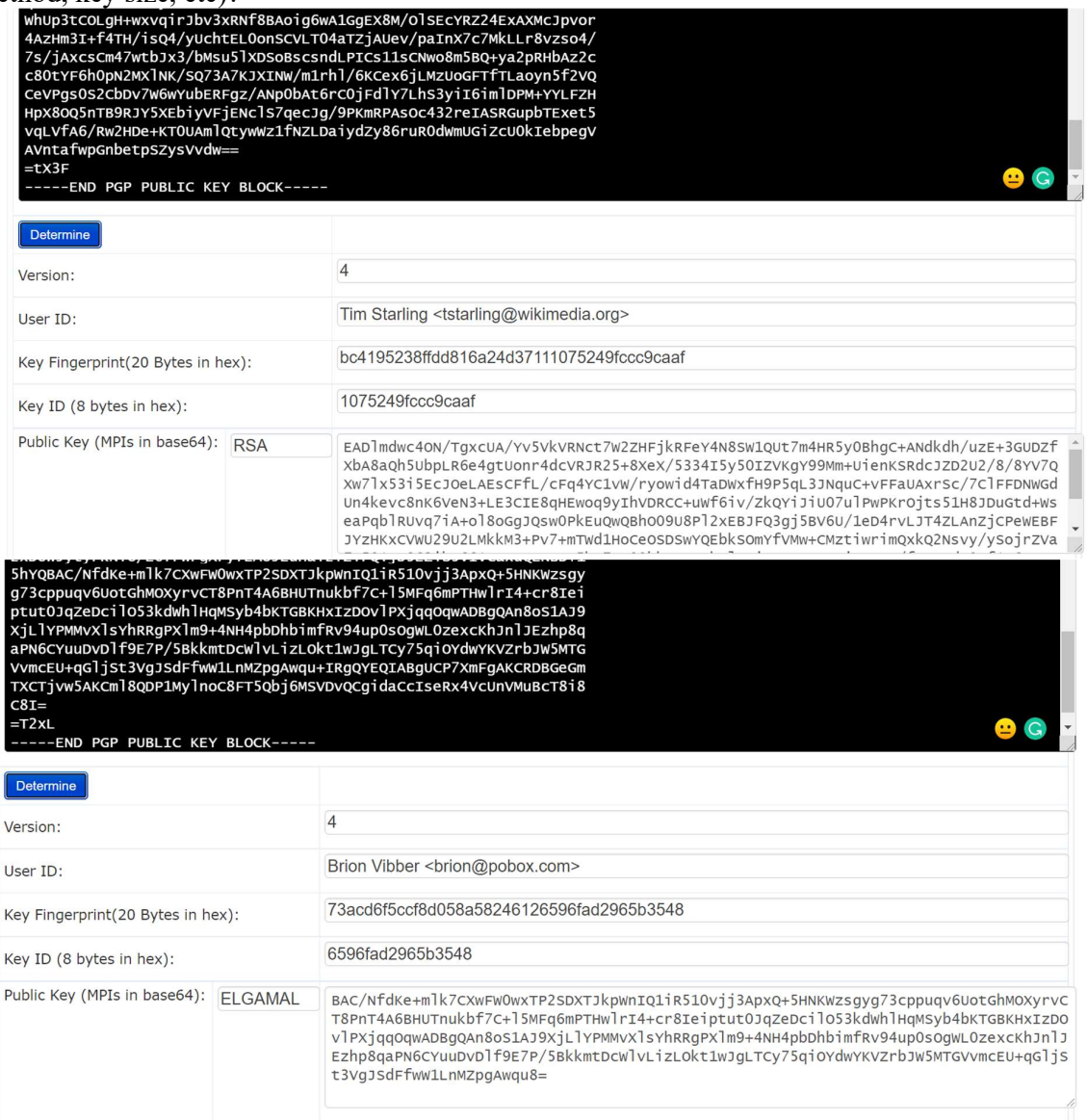
```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2

mQENBFTzi1ABCADIEWchOyqRQmU4AyQAMj2Pn68Sqo9lTPdPcItwo9LbTdv1YCFz
w3qLlp2RORMP+Kpdi92CIhdUYHDmZfHZ3IWTBgo9+y/Np9UJ6tNGocrgsq4xWz15
4vX4jJRddC7QySSh9UxDpRWf9sgqEv1pah136r95ZuyjC1EXnoNxdLJtx8PliCXc
hV/v4+KfOyzYh+HDJ4xP2bt1SO7dkasYZ6cA7BHYi9k4xgEwxVvYtNjSPjTsQY5R
cTayXveGafuxmhSauZKiB/2TFErjEt49Y+pO7tPTLX7bhMBVbUvojtt/JeUKV6vK
R82dmOd8seUvhwOHYBOJL+3S7PgFFsLo1NV5ABEBAAG0LkJpbGwgQnVjaGFuYW4g
KE5vbmUpIDx3LmJ1Y2hhbmFuQG5hcGllci5hYy51az6JATkEEwECACMFAlTzi1AC
GwMHCwkIBwMCAQYVCAIJCgsEFgIDAQIeAQIXgAAKCRDsAFZRGtdPQi13B/9KHeFb
l1AxqbafFGRDEvx8UfPnEww4FFqWhcr8RLWyE8/COlUpB/5AS2yvojmbNFMGzURb
LGf/u1LVHOa+NHQu57u8Sv+g3bBthEPh4bKaEzBYRS/dYHOx3APFyIayfm78JVRF
zdeTOOf6PaXUTRx7iscCTkN8DUD3lg/465ZX5aH3HWFFX5OOJSPStO/udqjoQuAr
WA5JqB//g2GfzZe1UzH5Dz3PBbJky8GiIfLmOOXSEIgAmpvc/9NjzAgjOW56n3Mu
sjVkibc+lljw+rOo97CfJMppmtcOvehvQv+KGOLZnpibiWVmM3vT7E6kRy4gEbDu
enHPDqhsvcqTDqaduQENBFTzi1ABCACzpJgZLK/sge2rMLURUQQ6lO2UrS/GilGC
ofq3WPnDt5hEjarwMMwN65PbODj0i7vnorhL+fdb/J8b8QTiyp7iO3dZvhDahcQ5
8afvCjQtQstY8+K6kZFzQOBgyOS5rHAKHNSPFq45MlnPo5aaDvP7s9mdMILITvlb
CFhcLoC6Oqy+JoaHupJqHBqGc48/5NU4qbt6fB1AQ/H4M+6og4OozohgkQb80Hox
YbJV4sv4vYMULd+FKOg2RdGeNMM/aWdqYo9Oqb/W2aHCCyXmhGHEEuok9jbc8cr/
xrWLOgDwlWpad8RfQwyVU/VZ3Eg3OseL4SedEmwOO
cr15XDIs6dpABEBAAGJAR8E
GAECAAkFAlTzi1ACGwwACgkQ7ABWURrXT0KZTgf9FUpkh3wv7aC5M2wwdEjt0rDx
nj9kxH99hhuTX2EHXuNLH+SwLGHBq5O2sq3jfP+owEhs8/EzOj1/fSKIqAdlz3mB
dbqWPjzPTY/m0It+wv3epOM75uWjD35PFOrKxxZmEf6SrjZD1skOB9bRy2v9iWN9
9ZkuvcfH4vT++PognQLTUqNxOFGpD1agrG0lXSCtJWQXCXPfWdtbIdThBgzH4flZ
ssAIbCaBlQkzfbPvrMzdTIP+AXg6++K9SnO9N/FRPYzjUSEmpRp+ox31WymvczcU
RmyUquF+/zNnSBVgtY1rzwaYi05XfuxGOWHVHPTtRyJ5pF4HSqiuvk6Z/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)?



```
WhUp3tCOLgH+wxvqirJbv3xRNf8BAoig6wA1GgEX8M/OlSEcYRZ24ExAXMcJpvor
4AzHm3I+f4TH/isQ4/yUchtELOonSCVLTO4aTZjAUev/paInX7c7MkLLr8vzso4/
7s/jAxcsCm47wtbJx3/bMsu5lXDSoBscsndLPICs11sCNwo8m5BQ+ya2pRHbAz2c
c8OtYF6hOpN2MxlNK/SQ73A7KJXINW/m1rhl/6KCex6jLMzUoGFTfTLaoyn5f2VQ
CeVPgsOS2CbDv7W6wYubERFgz/ANpObAt6rCOjFdlY7LhS3yiI6imlDPM+YYLFZH
HpX8OQ5nTB9RJY5XEbiyVFjENclS7qecJg/9PKmRPAsOc432reIASRGupbTExet5
vqLVfA6/Rw2HDe+KTOUAmlQtyWWz1fNZLDaiydZy86ruROdWmUGiZcUOkIebpegV
AVntafwpGnbetpSZysVvdw==
=tX3F
-----END PGP PUBLIC KEY BLOCK-----
```

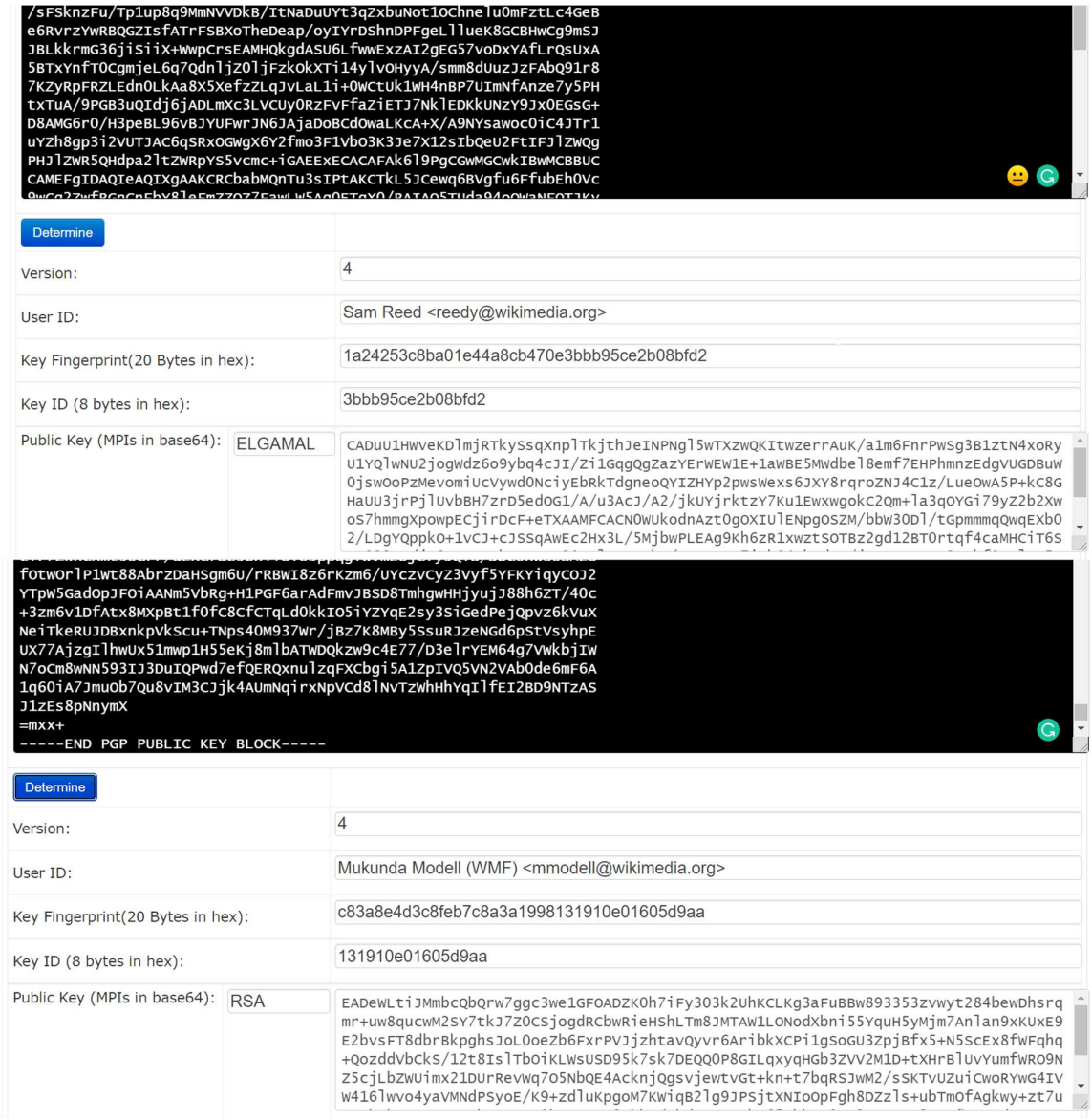
Determine

Version:	4
User ID:	Tim Starling <tstarling@wikimedia.org>
Key Fingerprint(20 Bytes in hex):	bc4195238ffdd816a24d37111075249fccc9caaf
Key ID (8 bytes in hex):	1075249fccc9caaf
Public Key (MPIs in base64): RSA	EADlmdwc4ON/TgxcUA/Yv5VkVRNct7W2ZHFjkRFeY4N8SW1QUt7m4HR5y0BhgC+ANdkdh/uzE+3GUDZf XbA8aQh5UbpLR6e4gtUonr4dcVRJR25+8XeX/5334I5y5OIZVKgY99Mm+UienKSRdcJZD2U2/8/8YV7Q Xw7lx53i5EcJOeLAEsCFfL/cFq4YC1vW/ryowid4TaDWxfH9P5qL3JNquC+vFFaUAxrSc/7ClFFDNWGd Un4kevc8nK6VeN3+LE3CIE8qHEwoq9yIhVDRCC+uWf6iv/ZkQYiJiUO7ulPwPKrOjts51H8JDuGtd+Ws eaPqblRUvq7iA+ol8oGgJQsw0PkEuQwQBhO09U8Pl2xEBJFQ3gj5BV6U/1eD4rvLJT4ZLAnZjCPeWEBF JYzHKxCVWU29U2LMkkM3+Pv7+mTWd1HoCeOSDSwYQEbkSOmYfVMw+CMztiwrimQxkQ2Nsvy/ySojrZVa

```
5hYQBAC/NfdKe+mlk7CXwFWOwxTP2SDXTJkpWnIQ1iR51Ovjj3ApxQ+5HNKWzsgy
g73cppuqv6UotGhMOXyrvCT8PnT4A6BHUTnukbf7C+l5MFq6mPTHwlrI4+cr8Iei
ptutOJqZeDcilo53kdWhlHqMSyb4bKTGBKHxIzDOvlPXjqqOqWADBgQAn8oS1AJ9
XjLlYPMMvXlsYhRRgPXlm9+4NH4pbDhbimfRv94upOsOgWLOzexcKhJnlJEzhp8q
aPN6CYuuDvDlf9E7P/5BkkmtDcWlvLizLOkt1wJgLTCy75qiOYdWYKVZrbJW5MTG
VvmcEU+qGljSt3VgJSdFfWW1LnMZpgAwqu+IRgQYEQIABgUCP7XmFgAKCRDBGeGm
TXCTjvw5AKCml8QDP1MylnoC8FT5Qbj6MSVDvQCgidaCcIseRx4VcUnVMuBcT8i8
C8I=
=T2xL
-----END PGP PUBLIC KEY BLOCK-----
```

Determine

Version:	4
User ID:	Brion Vibber <brion@pobox.com>
Key Fingerprint(20 Bytes in hex):	73acd6f5ccf8d058a58246126596fad2965b3548
Key ID (8 bytes in hex):	6596fad2965b3548
Public Key (MPIs in base64): ELGAMAL	BAC/NfdKe+mlk7CXwFWOwxTP2SDXTJkpWnIQ1iR51Ovjj3ApxQ+5HNKWzsgyg73cppuqv6UotGhMOXyrvC T8PnT4A6BHUTnukbf7C+l5MFq6mPTHwlrI4+cr8IeiptutOJqZeDcilO53kdWhlHqMSyb4bKTGBKHxIzDO vlPXjqqOqWADBgQAn8oS1AJ9XjLlYPMMvXlsYhRRgPXlm9+4NH4pbDhbimfRv94upOsOgWLOzexcKhJnlJ Ezhp8qaPN6CYuuDvDlf9E7P/5BkkmtDcWlvLizLOkt1wJgLTCy75qiOYdWYKVZrbJW5MTGVvmcEU+qGljS t3VgJSdFfWW1LnMZpgAwqu8=



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.

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 How long did it take to generate a 1,024 bit key? Immediately

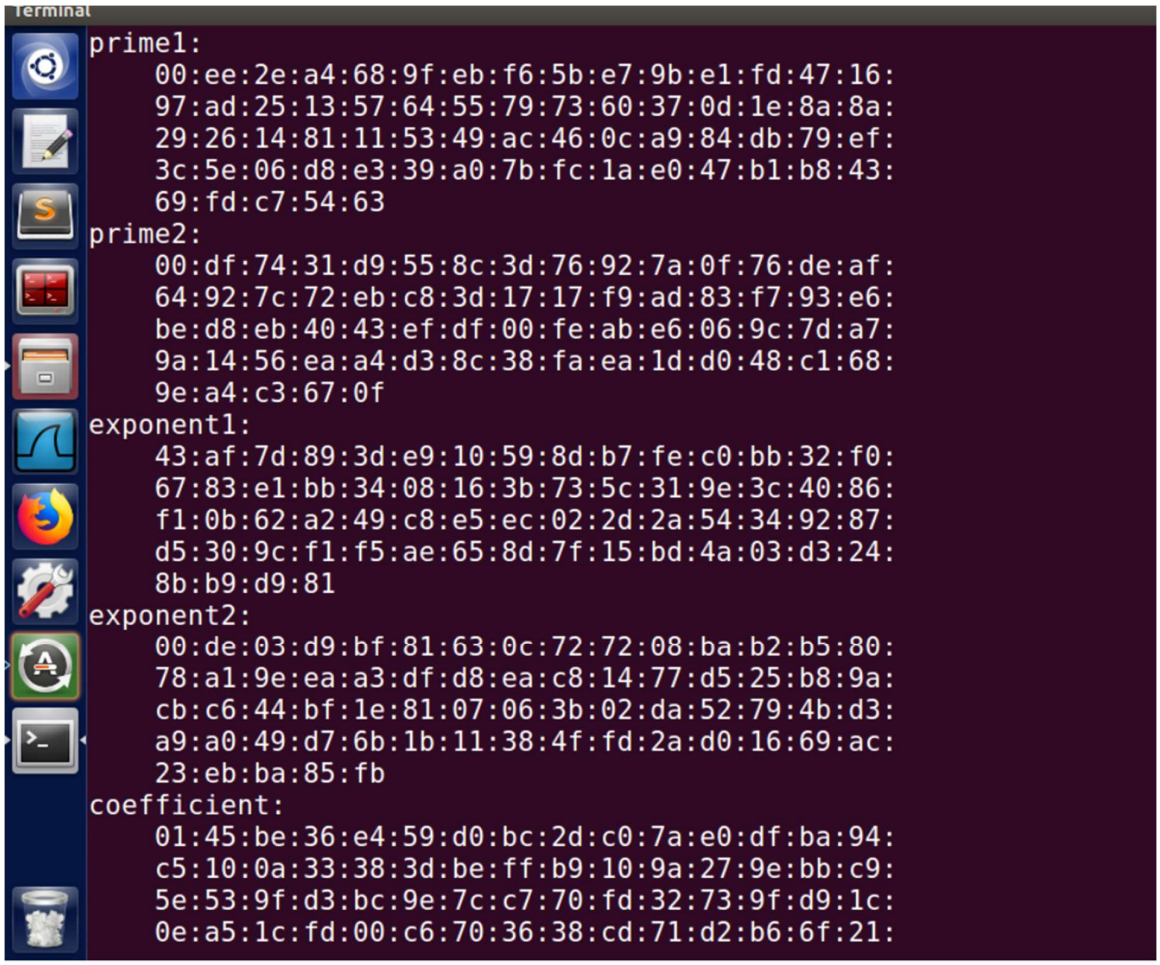
		Use the following command to view the keys: `cat private.pem`
B.2	Use following command to view the output file: `cat private.pem`	What can be observed at the start and end of the file: Hyphens and two words BEGIN and END are always present.
B.3	Next we view the RSA key pair: `openssl rsa -in private.pem -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? This makes the key file by itself useless to an attacker
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? Header and footer tell that the key is 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`	
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 Hello, Lets go on planned world tour.

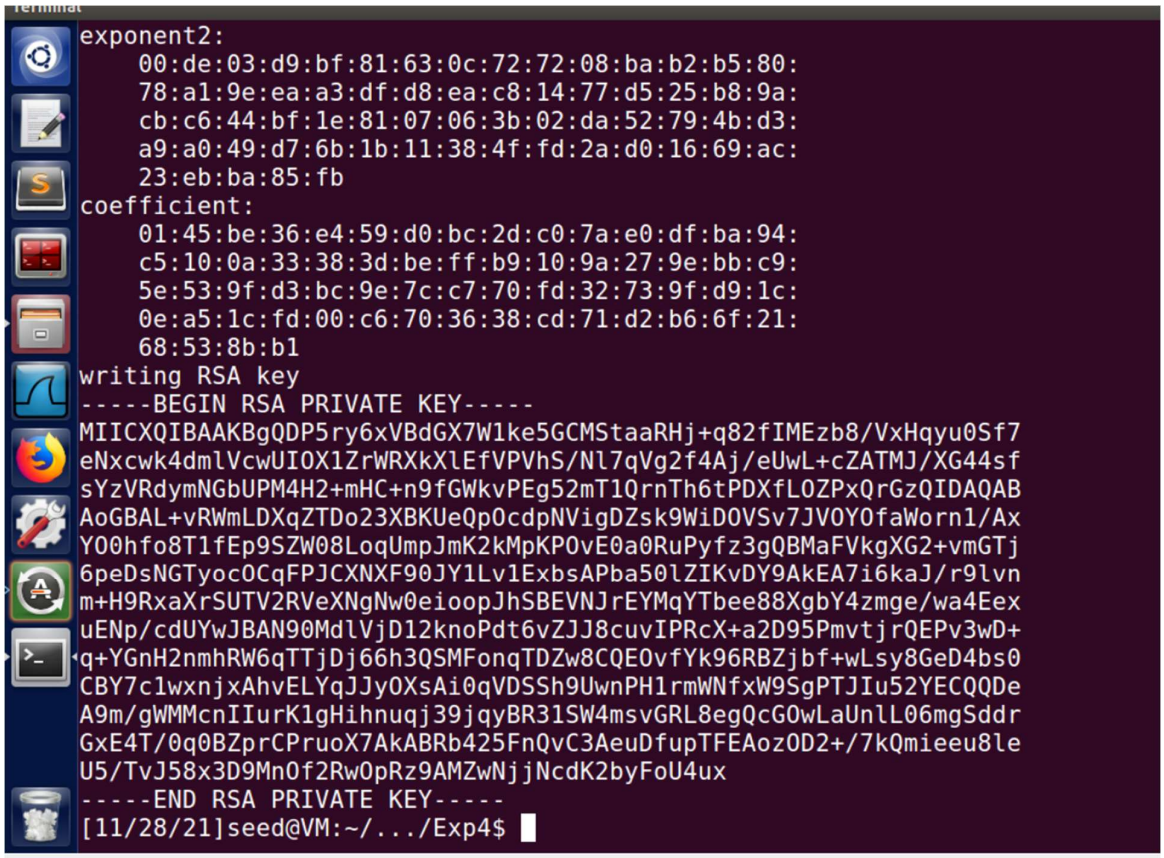
```
[11/28/21]seed@VM:~/.../CSS$ cd Exp4/
[11/28/21]seed@VM:~/.../Exp4$ openssl genrsa -out private.pem 1024
Generating RSA private key, 1024 bit long modulus
.........++++++
.................++++++
e is 65537 (0x10001)
[11/28/21]seed@VM:~/.../Exp4$ cat private.pem
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQDP5ry6xVBdGX7W1ke5GCMStaaRHj+q82fIMEzb8/VxHqyu0Sf7
eNxcwk4dmlVcwUIOX1ZrWRXkXlEfVPVhS/Nl7qVg2f4Aj/eUwL+cZATMJ/XG44sf
sYzVRdymNGbUPM4H2+mHC+n9fGWkvPEg52mT1QrnTh6tPDXfLOZPxQrGzQIDAQAB
AoGBAL+vRWmLDXqZTDo23XBKUeQpOcdpNVigDZsk9WiDOVSv7JVOYOfaWorn1/Ax
YO0hfo8T1fEp9SZW08LoqUmpJmK2kMpKPOvE0a0RuPyfz3gQBMaFVkgXG2+vmGTj
6peDsNGTyocOCqFPJCXNXF90JY1Lv1ExbsAPba50lZIKvDY9AkEA7i6kaJ/r9lvn
m+H9RxaXrSUTV2RVeXNgNw0eioopJhSBEVNJrEYMqYTbee88XgbY4zmge/wa4Eex
uENp/cdUYwJBAN90MdlVjD12knoPdt6vZJJ8cuvIPRcX+a2D95PmvtjrQEPv3wD+
q+YGnH2nmhRW6qTTjDj66h3QSMFonqTDZw8CQEOvfYk96RBZjbf+wLsy8GeD4bs0
CBY7c1wxnjxAhvELYqJJyOXsAi0qVDSSh9UwnPH1rmWNfxW9SgPTJIu52YECQQDe
A9m/gWMMcnIIurK1gHihnuqj39jqyBR31SW4msvGRL8egQcGOwLaUnlL06mgSddr
GxE4T/0q0BZprCPruoX7AkABRb425FnQvC3AeuDfupTFEAozOD2+/7kQmieeu8le
U5/TvJ58x3D9MnOf2RwOpRz9AMZwNjjNcdK2byFoU4ux
-----END RSA PRIVATE KEY-----
[11/28/21]seed@VM:~/.../Exp4$
```

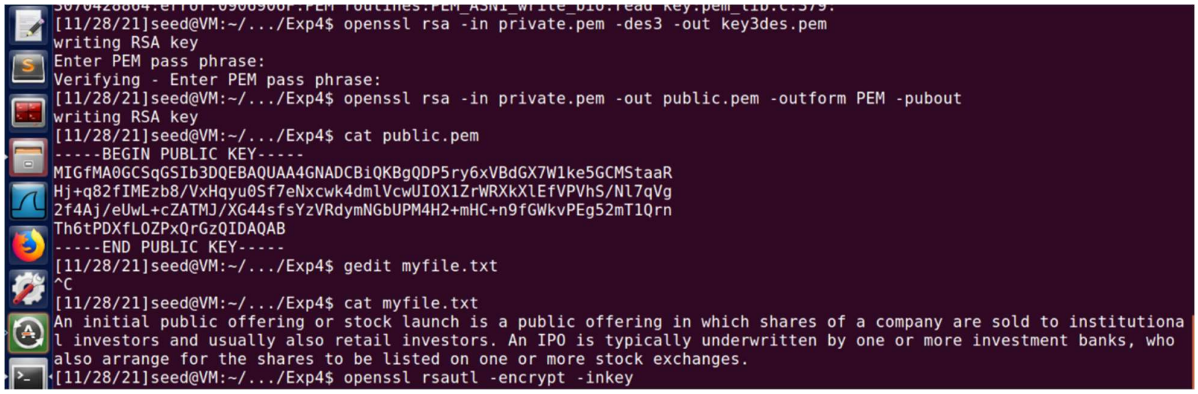
B1 and B2

```
[11/28/21]seed@VM:~/.../Exp4$ openssl rsa -in private.pem -text
Private-Key: (1024 bit)
modulus:
    00:cf:e6:bc:ba:c5:50:5d:19:7e:d6:d6:47:b9:18:
    23:12:b5:a6:91:1e:3f:aa:f3:67:c8:30:4c:db:f3:
    f5:71:1e:ac:ae:d1:27:fb:78:dc:5c:c2:4e:1d:9a:
    55:5c:c1:42:0e:5f:56:6b:59:15:e4:5e:51:1f:54:
    f5:61:4b:f3:65:ee:a5:60:d9:fe:00:8f:f7:94:c0:
    bf:9c:64:04:cc:27:f5:c6:e3:8b:1f:b1:8c:d5:45:
    dc:a6:34:66:d4:3c:ce:07:db:e9:87:0b:e9:fd:7c:
    65:a4:bc:f1:20:e7:69:93:d5:0a:e7:4e:1e:ad:3c:
    35:df:2c:e6:4f:c5:0a:c6:cd
publicExponent: 65537 (0x10001)
privateExponent:
    00:bf:af:45:69:8b:0d:7a:99:4c:3a:36:dd:70:4a:
    51:e4:29:39:c7:69:35:58:a0:0d:9b:24:f5:68:83:
    39:54:af:ec:95:4e:60:e7:da:5a:8a:e7:d7:f0:31:
    60:ed:21:7e:8f:13:d5:f1:29:f5:26:56:d3:c2:e8:
    a9:49:a9:26:62:b6:90:ca:4a:3c:eb:c4:d1:ad:11:
    b8:fc:9f:cf:78:10:04:c6:85:56:48:17:1b:6f:af:
    98:64:e3:ea:97:83:b0:d1:93:ca:87:0e:0a:a1:4f:
    24:25:cd:5c:5f:74:25:8d:4b:bf:51:31:6e:c0:0f:
    6d:ae:74:95:92:0a:bc:36:3d
```

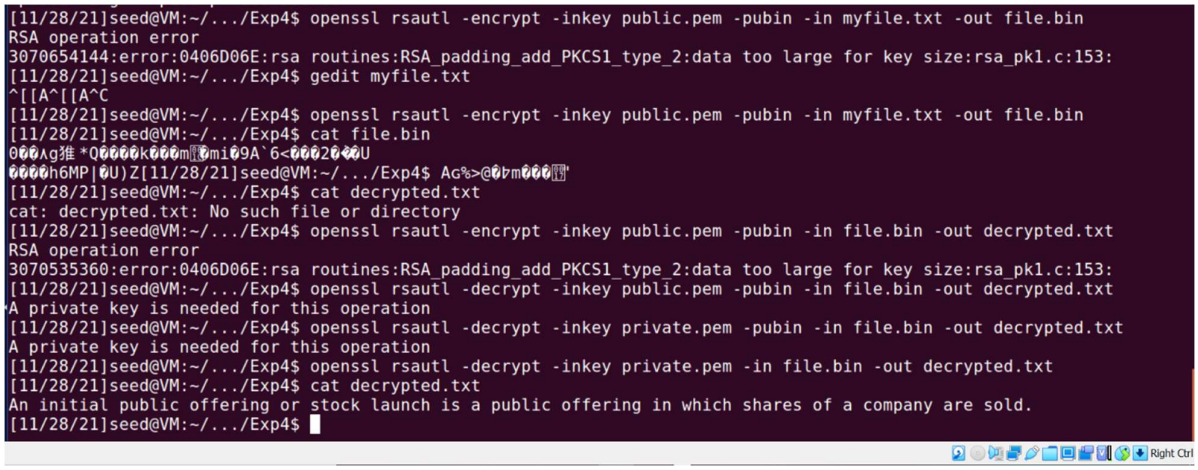
B3


Terminal
prime1:
00:ee:2e:a4:68:9f:eb:f6:5b:e7:9b:e1:fd:47:16:
97:ad:25:13:57:64:55:79:73:60:37:0d:1e:8a:8a:
29:26:14:81:11:53:49:ac:46:0c:a9:84:db:79:ef:
3c:5e:06:d8:e3:39:a0:7b:fc:1a:e0:47:b1:b8:43:
69:fd:c7:54:63
prime2:
00:df:74:31:d9:55:8c:3d:76:92:7a:0f:76:de:af:
64:92:7c:72:eb:c8:3d:17:17:f9:ad:83:f7:93:e6:
be:d8:eb:40:43:ef:df:00:fe:ab:e6:06:9c:7d:a7:
9a:14:56:ea:a4:d3:8c:38:fa:ea:1d:d0:48:c1:68:
9e:a4:c3:67:0f
exponent1:
43:af:7d:89:3d:e9:10:59:8d:b7:fe:c0:bb:32:f0:
67:83:e1:bb:34:08:16:3b:73:5c:31:9e:3c:40:86:
f1:0b:62:a2:49:c8:e5:ec:02:2d:2a:54:34:92:87:
d5:30:9c:f1:f5:ae:65:8d:7f:15:bd:4a:03:d3:24:
8b:b9:d9:81
exponent2:
00:de:03:d9:bf:81:63:0c:72:72:08:ba:b2:b5:80:
78:a1:9e:ea:a3:df:d8:ea:c8:14:77:d5:25:b8:9a:
cb:c6:44:bf:1e:81:07:06:3b:02:da:52:79:4b:d3:
a9:a0:49:d7:6b:1b:11:38:4f:fd:2a:d0:16:69:ac:
23:eb:ba:85:fb
coefficient:
01:45:be:36:e4:59:d0:bc:2d:c0:7a:e0:df:ba:94:
c5:10:0a:33:38:3d:be:ff:b9:10:9a:27:9e:bb:c9:
5e:53:9f:d3:bc:9e:7c:c7:70:fd:32:73:9f:d9:1c:
0e:a5:1c:fd:00:c6:70:36:38:cd:71:d2:b6:6f:21:

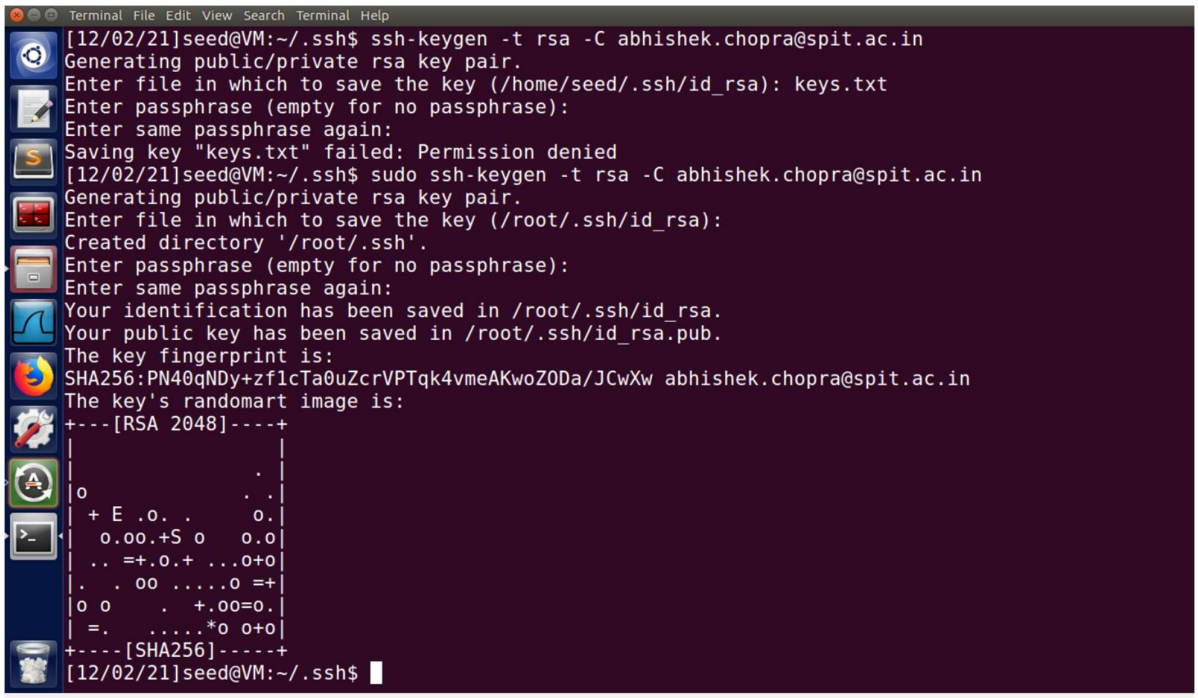

Terminal
exponent2:
00:de:03:d9:bf:81:63:0c:72:72:08:ba:b2:b5:80:
78:a1:9e:ea:a3:df:d8:ea:c8:14:77:d5:25:b8:9a:
cb:c6:44:bf:1e:81:07:06:3b:02:da:52:79:4b:d3:
a9:a0:49:d7:6b:1b:11:38:4f:fd:2a:d0:16:69:ac:
23:eb:ba:85:fb
coefficient:
01:45:be:36:e4:59:d0:bc:2d:c0:7a:e0:df:ba:94:
c5:10:0a:33:38:3d:be:ff:b9:10:9a:27:9e:bb:c9:
5e:53:9f:d3:bc:9e:7c:c7:70:fd:32:73:9f:d9:1c:
0e:a5:1c:fd:00:c6:70:36:38:cd:71:d2:b6:6f:21:
68:53:8b:b1
writing RSA key
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQDP5ry6xVBdGX7W1ke5GCMStaaRHj+q82fIMEzb8/VxHqyu0Sf7
eNxcwk4dmlVcwUIOX1ZrWRXkXlEfVPVhS/Nl7qVg2f4Aj/eUwL+cZATMJ/XG44sf
sYzVRdymNGbUPM4H2+mHC+n9fGWkvPEg52mT1QrnTh6tPDXfLOZPxQrGzQIDAQAB
AoGBAL+vRWmLDXqZTDo23XBKUeQpOcdpNVigDZsk9WiDOVSv7JVOYOfaWorn1/Ax
YO0hfo8T1fEp9SZW08LoqUmpJmK2kMpKPOvE0a0RuPyfz3gQBMaFVkgXG2+vmGTj
6peDsNGTyocOCqFPJCXNXF90JY1Lv1ExbsAPba50lZIKvDY9AkEA7i6kaJ/r9lvn
m+H9RxaXrSUTV2RVeXNgNw0eioopJhSBEVNJrEYMqYTbee88XgbY4zmge/wa4Eex
uENp/cdUYwJBAN90MdlVjD12knoPdt6vZJJ8cuvIPRcX+a2D95PmvtjrQEPv3wD+
q+YGnH2nmhRW6qTTjDj66h3QSMFonqTDZw8CQEOvfYk96RBZjbf+wLsy8GeD4bs0
CBY7c1wxnjxAhvELYqJJyOXsAi0qVDSSh9UwnPH1rmWNfxW9SgPTJIu52YECQQDe
A9m/gWMMcnIIurK1gHihnuqj39jqyBR31SW4msvGRL8egQcGOwLaUnlL06mgSddr
GxE4T/0q0BZprCPruoX7AkABRb425FnQvC3AeuDfupTFEAozOD2+/7kQmieeu8le
U5/TvJ58x3D9MnOf2RwOpRz9AMZwNjjNcdK2byFoU4ux
-----END RSA PRIVATE KEY-----
[11/28/21]seed@VM:~/.../Exp4$



B4 and B5



B6



B7

```
[12/02/21]seed@VM:~$ cd .ssh/
[12/02/21]seed@VM:~/.ssh$ sudo ssh-keygen -t rsa -C abhishek.chopra@spit.ac.in
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa): y
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in y.
Your public key has been saved in y.pub.
The key fingerprint is:
SHA256:/IqxgNBEOAoHKo7kF+wthH+OhYr2NyNCpVWBv71IPLE abhishek.chopra@spit.ac.in
The key's randomart image is:
+---[RSA 2048]----+
|.o. ...          |
|=.+. .           |
|==.+o            |
|Bo+o+o .         |
|oo=*.+= S        |
|.+o.*E . .       |
|oo..o.+ . .      |
|.... * = .       |
|   ..o = .       |
+----[SHA256]-----+
```

```
[12/02/21]seed@VM:~/.ssh$ ls -al
total 32
drwxr-xr-x  2 root root 4096 Dec  2 06:06 .
drwxr-xr-x 32 seed seed 4096 Dec  2 05:53 ..
-rw-r--r--  1 root root  407 Mar 19  2021 authorized_keys
-rw-r--r--  1 root root    1 Mar 17  2021 authorized_keys.save
-rw-r--r--  1 root root 1024 Mar 17  2021 .authorized_keys.swp
-rwxr-xr-x  1 root root 1675 Mar 18  2021 id_rsa
-rw-------  1 root root 1675 Dec  2 06:06 y
-rw-r--r--  1 root root  408 Dec  2 06:06 y.pub
[12/02/21]seed@VM:~/.ssh$ cat y.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCstGgN37LgQoRzlFdh7p8oq/tYpMntAVI3F53k/ECYkeKGdqLCq624tmN9VCdv3l0oQPZsuvtTDqF9p
6AiYy/v/fAJiEWu7IuJeHKUcJj4Yw7w7tLmTdU7XoCcDv9nybkOOw5iE1/9xN9bVp6BTPDhO3kjGrfVmRk/nVQFLUP/jl8dmQQpI+xvk3KHoVXETdnNVm
SB/l8DJ3qe5bqWqZbGl3rWt+rsAPlDu/lttggtmm6CHf/bRdTVaYJN/Cj7hTRpe7NiPSHPbJtgjF44rqtJvx1T/4nzOHTbn8lidSV1GRw0Iplnl+OqnGT
GkL8sJUTP4eBcH7hYl5OlSw17kkWD abhishek.chopra@spit.ac.in
```

On your VM, go into the ~/.ssh folder. Now generate your SSH keys:

```
ssh-keygen -t rsa -C "your email address"
```

The public key should look like this:

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQDLrriuNYTyWuC1IW7H6yea3hMV+rm029m2f6IddtlImHrOXjNwYyt4Elkkc7AzO
y899C3gpx0kJK45k/CLbPnrHvkLvtQ0AbzWEQpOKxI+tW06PcqJNmTB8ITRLqIFQ++ZanjHWMw2Odew/514y1dQ8dccCO
uzeGhL2Lq9dtfhSxx+1cBLcyoSh/lQcs1HpXtpwU8JMxWJl409RQOVn3gOusp/P/0R8mz/RWkmsFsyDRLgQK+xtQxbpbo
dpnz5lIOPWn5LnTOsi7eHmL3WikTyg+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
```

```
[12/05/21]seed@VM:~/.ssh$ sudo git clone https://github.com/AbhishekC20001/CSS-Exp-4.git
Cloning into 'CSS-Exp-4'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
Checking connectivity... done.
[12/05/21]seed@VM:~/.ssh$ sudo git clone git@github.com:AbhishekC20001/CSS-Exp-4.git
fatal: destination path 'CSS-Exp-4' already exists and is not an empty directory.
```

If this doesn't work, try the https connection that is defined on GitHub.

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 (and should have 256 bits).	Can you view your key? Yes Now use "`cat priv.pem`" to view your key.

C.2 We can view the details of the ECC parameters used with:

```
openssl ecparam -in priv.pem -text -
param_enc explicit -noout
```

Outline these values:

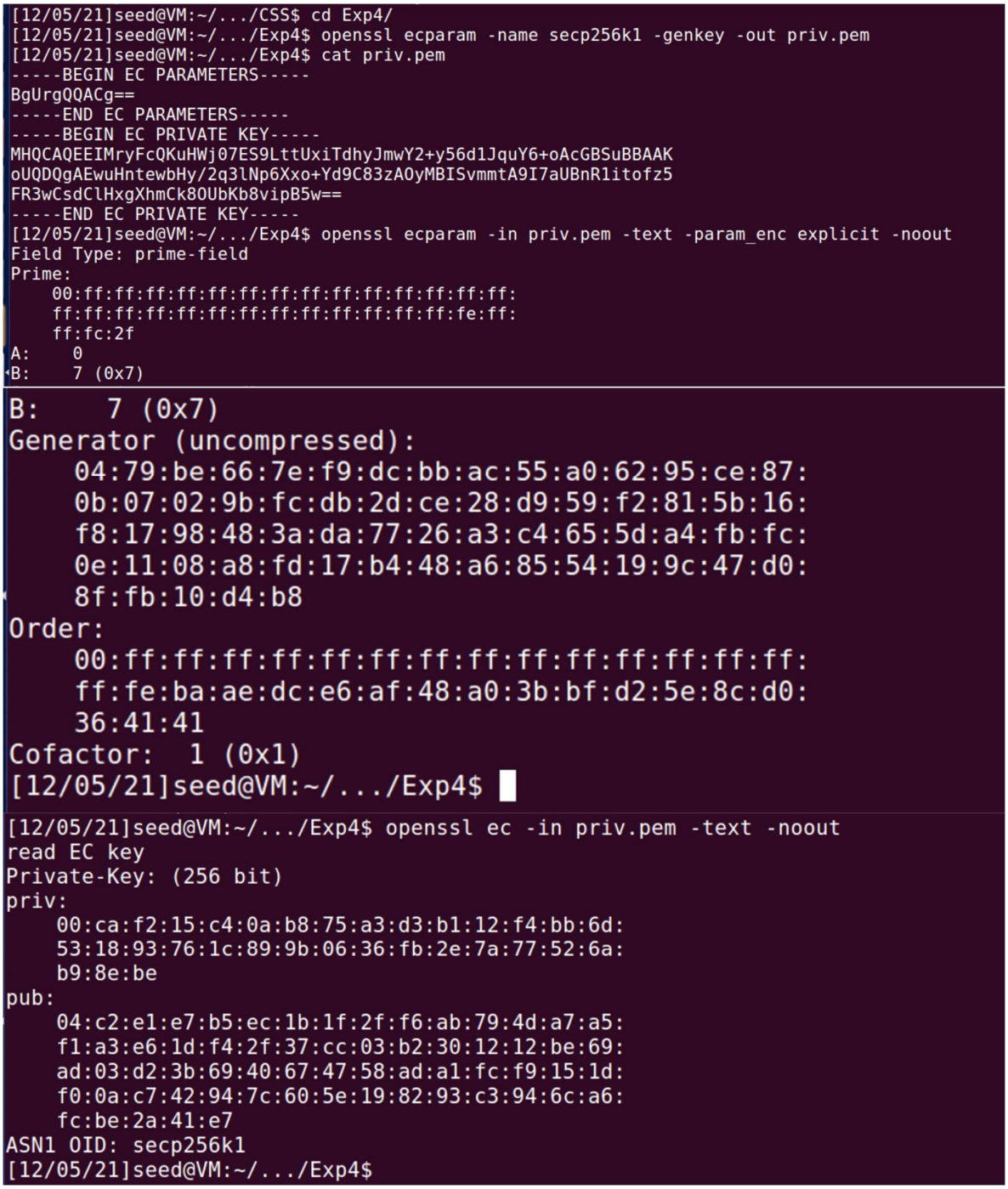
Prime (last two bytes):fc:2f

A:0

B:7

Generator (last two bytes):d4:b8

		Order (last two bytes):41:41
C.3	Now generate your public key based on your private key with: `openssl ec -in priv.pem -text -noout`	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



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

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:

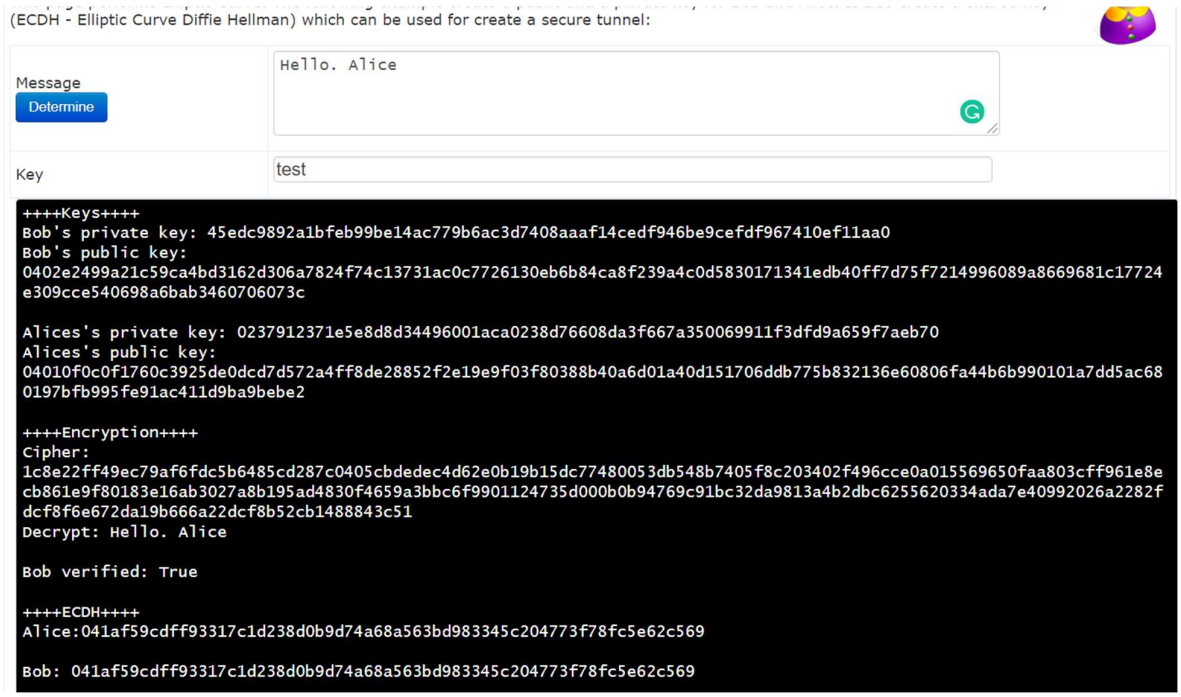
https://asecuritysite.com/encryption/elc

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"))
```

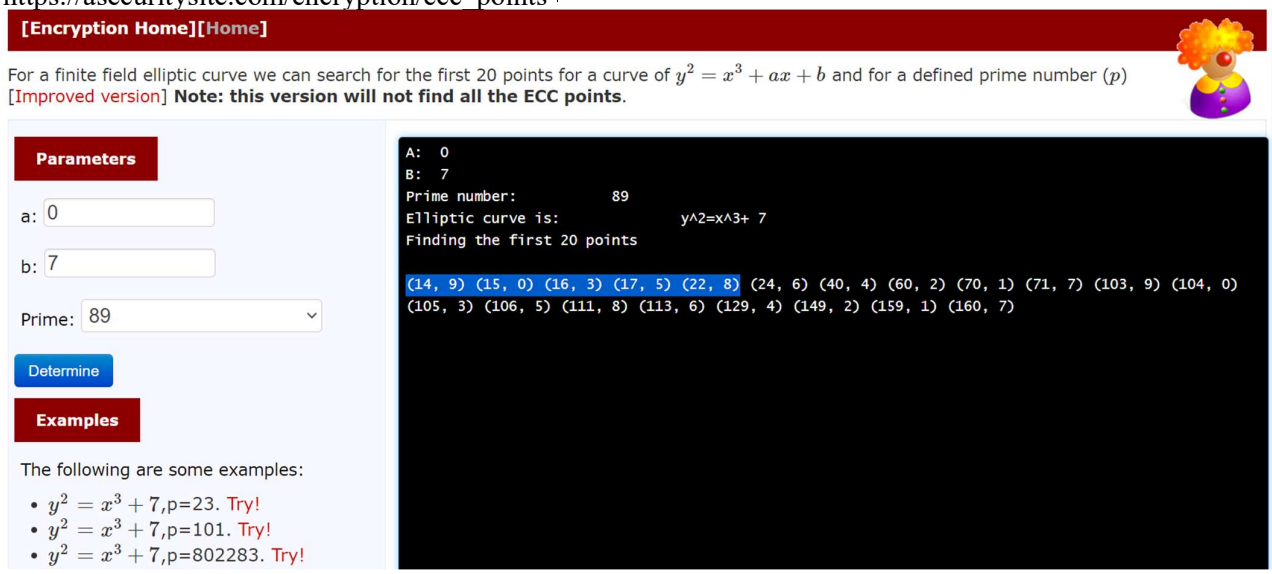
For a message of “Hello. Alice”, what is the ciphertext sent (just include the first four characters): 1c8e

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_points



First five points:

(14, 9) (15, 0) (16, 3) (17, 5) (22, 8)

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

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

msg="Hello"
type = 1
cur=NIST192p

sk = SigningKey.generate(curve=cur)

vk = sk.get_verifying_key()

signature = sk.sign(msg)

print "Message:\t",msg
print "Type:\t\t",cur.name
print "========================="

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

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

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

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

NIST192p:
r44L0nLLZ9BBbiOEv8+LWLD7wr213HSmbcyGkQlCNpgx9yyeOjwyOByqHDKR//hB

NIST521p:

AbjAlbn7o464qPxvLR0vmTLc/cf1XqdrsXDDb9q4RhCqbFqZFlcHfbauV0V5bJFjcULpAPx
MxPq4MPiT6ABLfEkiAD4Gtqe/B7BRsFzCUE2TsqrWu3DnwhRjDJ9sCTcd13xwcIg2iYjNn
VSeihYgDK/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
Curve: $y^2 = x^3 + 7$

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

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

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:

$C = M^e \pmod N = 5\text{^}5 \pmod{133} = 66$

Now decrypt your ciphertext with:

$M = C^d \pmod N = 66\text{^}65 \pmod{133} = 5$

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:

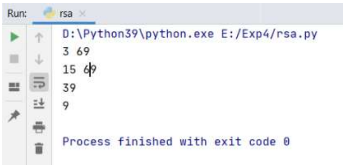
```
p=11
q=3
N=p*q
PHI=(p-1)*(q-1)
e=3
for d in range(1,100):
```

```
        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
```

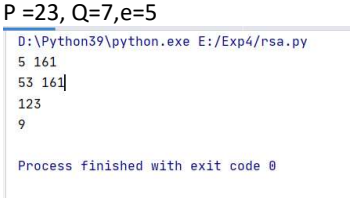
Select three more examples with different values of p and q, and then select e in order to make sure that the cipher will work:

P=19,q=7,e=5
M=5 , d obtained is 65

P = 23, Q=3, e =3



P =23, Q=7,e=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:

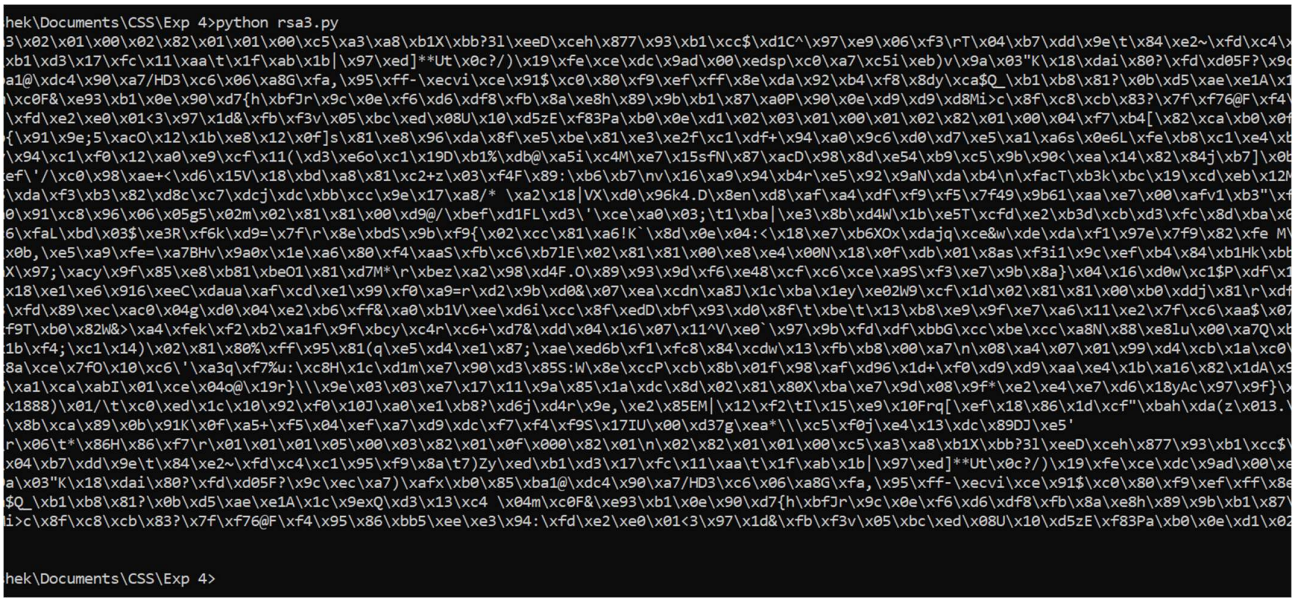
Inverse of 53 (mod 120) = 77

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
```



DER files are digital certificates in binary format, instead of the instead of the ASCII PEM format. A DER file should not have any BEGIN/END statements and will show garbled binary content

PEM -> contains the X. 509 certificate encoded in text (base64 and encrypted) – both have the same content,

DER is a binary format to encode the certificate. 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

xk0EXEOYvQECAIpLP8wfLxzgcolMpwgzcUzTlHOicggOIyuQKsHM4XNPugzU
X0NeaawrJhfi+f8hDRojJ5Fv8jBIOm/KwFMNTT8AEQEAAc0UYmlsbCA8Ymls
bEBob21lLmNvbT7CdQQQAQgAHwUCXEOYvQYLCQcIAwIEFQgKAgMWAgECGQEC
GwMCHgEACgkQoNsXEDYt2ZjkTAH/b6+pDfQLi6zg/Y0tHS5PPRv1323cwoay
vMcPjnWq+VfiNyXzY+UJKR1PXskzDvHMLOyVpUcjle5ChyT5LOw/ZM5NBFxD
mL0BAgDYlTsTO6vVQxu3jmfLzKMAr4kLqqIuFFRCapRuHYLOjw1gJZS9p0bF
SOqS8zMEGpN9QZxkG8YEcH3gHxlrvALtABEBAAHCXwQYAQgACQUCXEOYvQIb
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/MHy8c4HKJTKcIM3FMO5R9InIIDiMrkCrBzOFzT7oM
1F9DXmmsKyYX4vn/IQOaIyeRb/IwSNJvysBTDUO/ABEBAAH+CQMIBNTT/OPv
TJzgvF+fLOsLsNYP64QfNHav5O744yOMLV/EZT3gsBwO9v4XF2SsZj6+EHbk
O9gWi31BAIDgSaDsJYf7xPOhp8iEWWwrUkC+jlGpdTsGDJpeYMIsVVv8Ycam
Og7MSRsL+dYQauIgtVb3dloLMPtuL59nVAYuIgD8HXyaH2vsEgSZSQnOkfvF
+dWeqJxwFM/uX5PVKcuYsroJFBEO1zas4ERfxbbwnsQgNHpjdIpueHx6/4EO
b1kmhOd6UT7BamubY7bcma1PBSv8PH31Jt8SzRRiaWxsIDxiaWxsQGhvbWUu
Y29tPsJ1BBABCAAfBQJcQ5i9BgsJBwgDAgQVCAoCAxYCAQIZAQIbAwIeAQAK
CRCg2xcQNi3ZmORMAf9vr6kN9AuLrOD9jSOdLk89G/XfbdzChrK8xw+Odar5
V+I3JfNj5QkpHU9eyTMO8cws7JWlRyOV7kKHJPks7D9kx8BmBFxDmL0BAgDY
lTsTO6vVQxu3jmfLzKMAr4kLqqIuFFRCapRuHYLOjw1gJZS9p0bFSOqS8zME
GpN9QZxkG8YEcH3gHxlrvALtABEBAAH+CQMI2Gyk+BqVOgzgZX3C8OJRLBRM
T4sLCHOUGlwaspe+qatOVjeEuxA5DuSs0bVMrw7mJYQZLtjNkFAT92lSwfxY
gavS/bILlw3QGAOCT5mqijKrOnurKkekKBDSGjkjVbIoPLMYHfepPOju1322
Nw4V3JQO4LBh/sdgGbRnwW3LhHEK4Qe7Ocuiert8C+S5xfG+T5RWADi5HR8u
UTyH8x1hOZrOF7KOWq4UcNvrUm6c35H6lClC4Zaar4JSN8fZPqVKLlHTVcL9
lpDzXxqxKjSO5KXXZBh5wl8EGAEIAAkFAlxDmL0CGwwACgkQoNsXEDYt2ZjA
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	Result

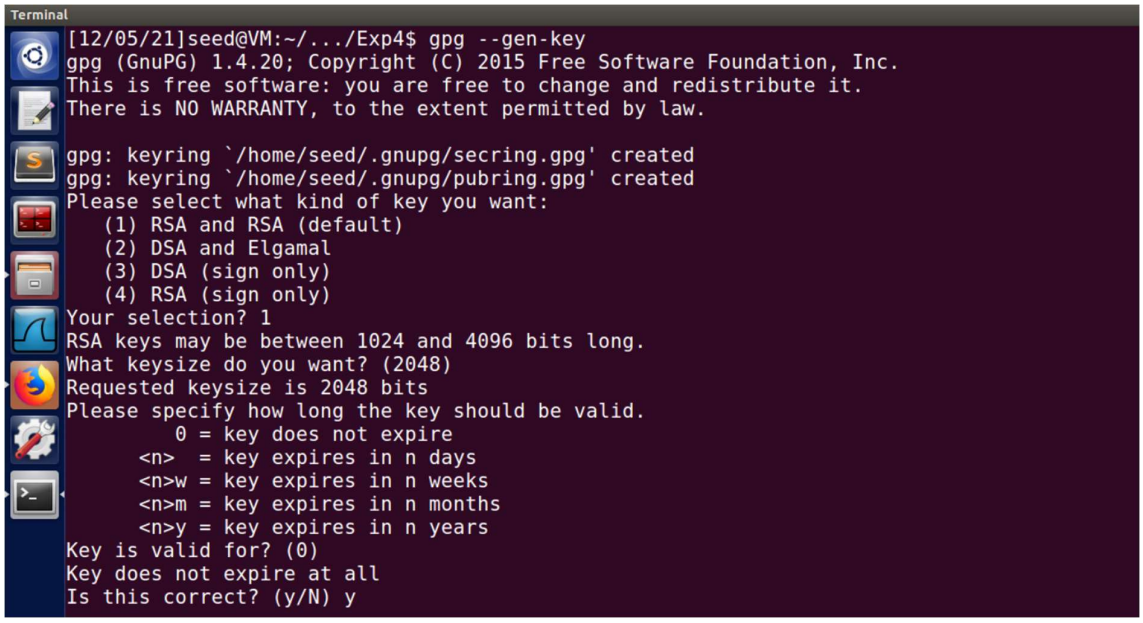
1	Create a key pair with (RSA and 2,048-bit keys):
	`gpg --gen-key`
	Now export your public key using the form of:
	`gpg --export -a "Your name" > mypub.key`
	Now export your private key using the form of:
	`gpg --export-secret-key -a "Your name" > mypriv.key`
		How is the randomness generated?
		Randomness is generated using pseudorandom number generation.
		
		Outline the contents of your key file:
		The file contains header and footer with naming conventions begin and end

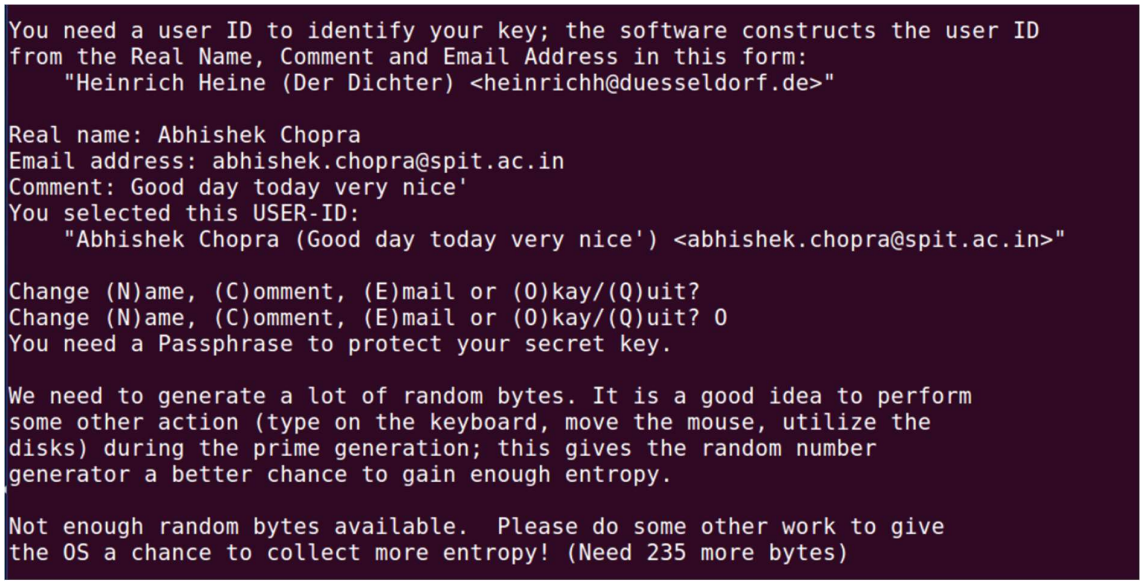
2	Now send your lab partner your public key in the contents of an email, and ask them to import it onto their key ring (if you are doing this on your own, create another set of keys to simulate another user, or use Bill's public key – which is defined at http://asecuritysite.com/public.txt and send the email to him): `gpg --import` *theirpublickey* `.key`	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

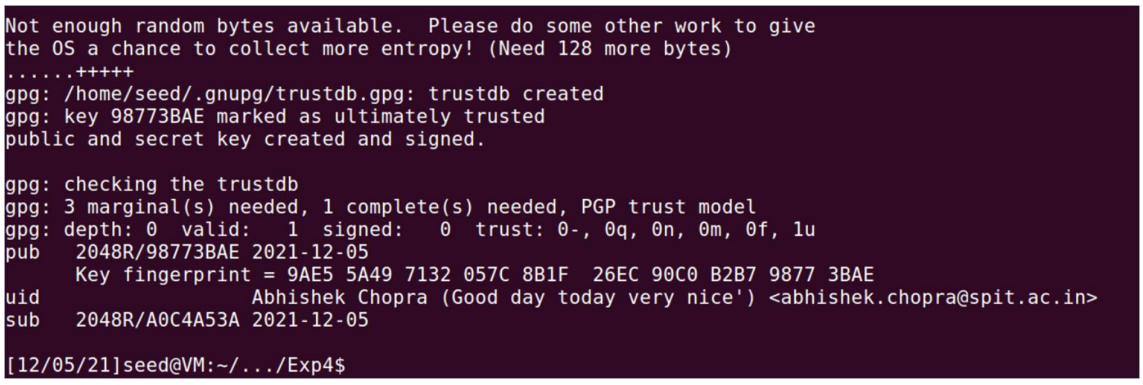
3	*Now list your keys with:*
	
	`gpg --list-keys`
	
	Create a text file, and save it. Next encrypt the file with their public key:
	
	`gpg -e -a -u "Your Name" -r "Your Lab Partner Name" hello.txt`
		What does the –a option do:
		
		Create ASCII armored output.
		
		What does the –r option do:
		Encrypt for user id name.
		
		What does the –u option do:
		Use name as the user ID to sign.
		
		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.

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 > myfi le.txt`	Can you decrypt the message: Yes

G

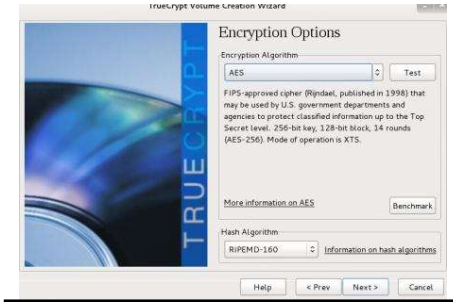

Terminal
[12/05/21]seed@VM:~/.../Exp4$ gpg --gen-key
gpg (GnuPG) 1.4.20; Copyright (C) 2015 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
gpg: keyring `/home/seed/.gnupg/secring.gpg' created
gpg: keyring `/home/seed/.gnupg/pubring.gpg' created
Please select what kind of key you want:
(1) RSA and RSA (default)
(2) DSA and Elgamal
(3) DSA (sign only)
(4) RSA (sign only)
Your selection? 1
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048)
Requested keysize is 2048 bits
Please specify how long the key should be valid.
0 = key does not expire
<n> = key expires in n days
<n>w = key expires in n weeks
<n>m = key expires in n months
<n>y = key expires in n years
Key is valid for? (0)
Key does not expire at all
Is this correct? (y/N) y


You need a user ID to identify your key; the software constructs the user ID
from the Real Name, Comment and Email Address in this form:
"Heinrich Heine (Der Dichter) <heinrichh@duesseldorf.de>"
Real name: Abhishek Chopra
Email address: abhishek.chopra@spit.ac.in
Comment: Good day today very nice'
You selected this USER-ID:
"Abhishek Chopra (Good day today very nice') <abhishek.chopra@spit.ac.in>"
Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit?
Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O
You need a Passphrase to protect your secret key.
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.
Not enough random bytes available. Please do some other work to give
the OS a chance to collect more entropy! (Need 235 more bytes)


Not enough random bytes available. Please do some other work to give
the OS a chance to collect more entropy! (Need 128 more bytes)
.....+++++
gpg: /home/seed/.gnupg/trustdb.gpg: trustdb created
gpg: key 98773BAE marked as ultimately trusted
public and secret key created and signed.
gpg: checking the trustdb
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg: depth: 0 valid: 1 signed: 0 trust: 0-, 0q, 0n, 0m, 0f, 1u
pub 2048R/98773BAE 2021-12-05
Key fingerprint = 9AE5 5A49 7132 057C 8B1F 26EC 90C0 B2B7 9877 3BAE
uid Abhishek Chopra (Good day today very nice') <abhishek.chopra@spit.ac.in>
sub 2048R/A0C4A53A 2021-12-05
[12/05/21]seed@VM:~/.../Exp4$

```
[12/05/21]seed@VM:~/.../Exp4$ gpg --export -a Abhishek Chopra  > mypub.key
[12/05/21]seed@VM:~/.../Exp4$ gpg --export-secret-key -a abhishek Chopra > mypriv.key
[12/05/21]seed@VM:~/.../Exp4$ gpg --import theirpublickey.key
gpg: can't open `theirpublickey.key': No such file or directory
gpg: Total number processed: 0
[12/05/21]seed@VM:~/.../Exp4$ gedit theirpublickey.key
^C
[12/05/21]seed@VM:~/.../Exp4$ gpg --import theirpublickey.key
gpg: key A93AE3D8: public key "Bill Buchanan <B.Buchanan@napier.ac.uk>" imported
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
[12/05/21]seed@VM:~/.../Exp4$ gpg --list-keys
/home/seed/.gnupg/pubring.gpg
------------------------------
pub   2048R/98773BAE 2021-12-05
uid                  Abhishek Chopra (Good day today very nice') <abhishek.chopra@spit.ac.in>
sub   2048R/A0C4A53A 2021-12-05

pub   4096R/A93AE3D8 2021-02-17 [expires: 2025-02-17]
uid                  Bill Buchanan <B.Buchanan@napier.ac.uk>
sub   4096R/DF574888 2021-02-17 [expires: 2025-02-17]

[12/05/21]seed@VM:~/.../Exp4$ gpg -e -a -u Abhishek Chopra -r Bill Buchanan  hello.txt
usage: gpg [options] --encrypt [filename]
[12/05/21]seed@VM:~/.../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: **2.8 GB/s** AES-Twofish: **535 MB/s** AES-Two-Seperent: **294 MB/s** Serpent -AES : **524 MB/s** Serpent: **530 MB/s** Serpent-Twofish-AES: **290 MB/s** Twofish: **648 MB/s** Twofish-Serpent: **315 MB/s** Which is the fastest: **AES** Which is the slowest: **Twofish-Serpent- AES**

2 3	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? Random pool tracks the user mouse movement And generates a strong cryptographic key.
4	Now mount the file as a drive.	Can you view the drive on the file viewer and from the console? Yes
	Create some files your TrueCrypt drive and save them.	Without giving them the password, can they read the file? No, one cannot read them without giving the password. With the password, can they read thefiles? Yes

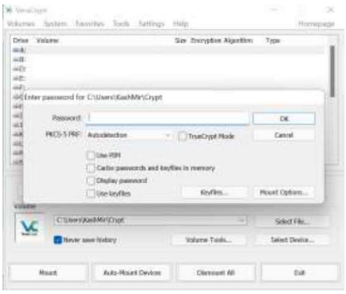
Algorithm	Encryption	Decryption	Mean
AES	2.8 GiB/s	2.7 GiB/s	2.7 GiB/s
Camellia	916 MiB/s	903 MiB/s	910 MiB/s
Twofish	648 MiB/s	613 MiB/s	630 MiB/s
Serpent	530 MiB/s	589 MiB/s	560 MiB/s
AES(Twofish)	535 MiB/s	530 MiB/s	532 MiB/s
Serpent(AES)	524 MiB/s	523 MiB/s	523 MiB/s
Kuznyechik	506 MiB/s	424 MiB/s	465 MiB/s
Kuznyechik(AES)	430 MiB/s	389 MiB/s	410 MiB/s
Camellia(Serpent)	371 MiB/s	363 MiB/s	367 MiB/s
Twofish(Serpent)	315 MiB/s	309 MiB/s	312 MiB/s
Camellia(Kuznyechik)	320 MiB/s	302 MiB/s	311 MiB/s
AES(Twofish(Serpent))	294 MiB/s	282 MiB/s	288 MiB/s
Serpent(Twofish(AES))	290 MiB/s	282 MiB/s	286 MiB/s
Kuznyechik(Twofish)	292 MiB/s	269 MiB/s	281 MiB/s
Kuznyechik(Serpent(Camellia))	179 MiB/s	196 MiB/s	188 MiB/s

1

2

2

3



3

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?**

Bitcoin addresses are actually the 256-bit SHA hash of an ECDSA public key, so any vulnerabilities in those algorithms would constitute a vulnerability in bitcoin itself. Realistically, however, breaking this level of encryption requires a huge amount of processing power. Coincidentally it requires precisely the same kind of processing power that bitcoin mining requires and in almost every scenario it would be massively more profitable to mine than to hack. It requires great computational power but using cracker of 1 Tera keys per second it can be achieved very easily.

I What I should have learnt from this lab?

The key things learnt:

12

- 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.

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 b64decode
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
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)

print
print "====Ciphertext==="
print b64encode(ciphertext)

cipher = PKCS1_OAEP.new(privKeyObj)
message = cipher.decrypt(ciphertext)


print
print "====Decrypted==="
print "Message:",message
```

```
C:\Users\Abhishek\Documents\CSS\Exp 4>python additional.py

====Private key===
b'-----BEGIN RSA PRIVATE KEY-----\nMIICXQIBAAKBgQDi0H18X1zgtZwsTN7/jyluPBQn1mClEwo4AVWiAQQlW8xFG6sn\nO2QX9eo6p8mommqK6CJ
0ybV4Axun+K1+Dk4arbQLTclY9CXMvXSP6WqBlQd4Y74w\nbj2jyQXZU3zJTN/HxC9rq8swLBEnL2sa9Zfi0Sn05DjiNi23L+NgV91xRQIDAQAB\nAoGAAk9
MArPDxB4x6VdvxqKKDYGUVWUYGFv39NmxMztAfTwwws5nbtQqnGKRm+OF\nLWIM5RBOnKnIemnt+X3R/Z4sUX9AESm363UrOS5RQsfEksijGNYJcaC6fVbsn
W9Q\nnnS556fSqph0Sk7NsH7tO7QNLFXCcQ/lF9wdYXLrXuKHVaqkCQQDodDtoRi15Ae4G\nlWE6Y1yOIsdZfcoahTZNiDQytgddc1s0SCy9xaz078IHXN+DP
QHFsE4uSLlUstGw\nYqMp40O5AkEA+coEZIBLPkXcZa8u8lkPjDve8zYhaG4OHPEItUzEHrRbrlghda6G\nvIGS9zm2zr9e0Az/JxBHSLScStH438M37QJAw
Lnc3x1CSBjzeLHV7cp1To951e/M\nj7jfFFLz0ZNnVkUsv9T4turX7JoDTnolmltT7NgxzzrU+r5SiWfs7ZUgiQJBAKll\nEq2FQ4Grptdeghvfwed8Aqo1H
1rOp8uAukMZGDcqYLJPGgHQpZbY2G3IOezjYsz8\nw1wecfkgYo1q77fvj9ECQQCAVaYk8dV+lkrYhE4f+HBZ7awCqqdMjwU+q73ZVAst\nquaPn6Fvqh8Yq
dKSSreJrY7AQqeLyOTlCFqwYnNZxG5+\n-----END RSA PRIVATE KEY-----'

====Public key===
b'-----BEGIN PUBLIC KEY-----\nMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDi0H18X1zgtZwsTN7/jyluPBQn\n1mClEwo4AVWiAQQlW8xFG6sn
O2QX9eo6p8mommqK6CJ0ybV4Axun+K1+Dk4arbQL\nTclY9CXMvXSP6WqBlQd4Y74wbj2jyQXZU3zJTN/HxC9rq8swLBEnL2sa9Zfi0Sn0\n5DjiNi23L+Ng
V91xRQIDAQAB\n-----END PUBLIC KEY-----'
====Ciphertext===
b'iH0/2u6LlimOOltJ5canJXaUxj4g06t6GUSV+mlP2PrMcHQtoWMlOmCdEViboHDYtspzWo++TnLlIgJYClcXR6MsKcFBgtLX5Bek3KfjaG8aKRo80FILvL
DhfRatfJe/Ca/61tetSGrZAeXlL3TZrWNbSshZGJoP8gBIEJnlELY='

====Decrypted===
Message: b'hello...'

C:\Users\Abhishek\Documents\CSS\Exp 4>
```