

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
```

```
mQENBFTzj1ABCADIEwchoyqRQmu4AyQAMj2Pn68Sqd91TPdPcItwo9LbTdV1YCFZ  
w3dL1p2RORM+Kpd192CIhdUHdmZFHZ3IWTBgo9+y/Np9UJ6tNGocrgsg4xWz15  
4vX4jJRRddC7QySSh9UxDpRwf9sggqEv1ph136r95zuyjC1ExnoNxLJtx8plicxc  
hv/v4+kfoyzYh+HDj4xP2bt1s07dkasYZ6ca7BHYi9k4xGEwxVVYtNjSPjTsQY5R  
cTayXveGafuxmhSAuzKib/2TfErjEt49Y+p07tPTLx7bhMBVbUvojt/JeUKV6VK  
R82dm0d8seUvhwoHYB0JL+3S7PgFFsLo1NV5ABEAAAG0LkJpbGwgQnvjaGFuYW4g  
KE5vbmuDx3Lmj1Y2hhbmFQOG5hcG1lci5hYY51az6JATKEEWECACMFAlTzi1AC  
GwMHcwkIBwMCAQYVCAIJCgsEFgIDAQIeAQIXgAAKCRDSAFZRGtdPQj13B/9KHeFb  
1IAxqbaFFGRDEvx8ufPnEww4FFqWhcr8RLwyE8/C0lUpb/5AS2yvojmBNFMGZURb  
LGf/u1LVh0a+NHQu5u8Sv+g3bBthEPh4bKaEzBYRS/dyHOx3APFyIayfm78JVRF  
zdeToof6PaXUTRx7iscCTkn8DUD31g/465ZX5aH3HWFFX500JSPSt0/udqjoQuAr  
WA5Jqb//g2GfzzeluzH5Dz3PBbjky8G1IfLm0OXSEigAmpvc/9NjzAgjOW56n3Mu  
sjvkibc+11jw+r0o97CfJMpptmcOvehvQv+KGOLZnpibiwmm3v7E6kRy4gEbDu  
enHPDqhsvcTqdaduQENBFTzj1ABCACZpJgZLK/sge2rMLURUQQ6102urs/Gi1GC  
ofq3WPndt5hEjarwmwN65Pb0Dj0i7vnorhl+fdb/j8b8QtIyp7i03dzvhDahcQ5  
8afvcjQtsty8+K6KZfzQOBgyOS5rHAKHNSPFq45M1nP05aaDv7s9mdMILITv1b  
FcHcLoC60qy+JoaHupJqHBqGc48/5NU4qbt6f81AQ/H4M+6og4oozoohgkQb80Hox  
YbJV4sv4yMULd+FKOg2RdGenNM/awdqYo90qb/w2aHCCyXmhGHEEuok9jbc8cr/  
xrWL0gDwlwpad8Rfqwyvu/VZ3Eg30seL4SeEdmwoo  
cr15XDIs6dpABEABAQJAR8E  
GAECAAkFA1Tzi1ACGgwACgkQ7ABWUrrXT0KZTgf9FUpkh3wv7aC5M2wwdEjt0rDx  
nj9Kxh99hhuTx2EHXuNLH+SwLGHbq502sq3jfP+owEhs8/Ez0j1/fskIqAd1z3mb  
dbqWPjzPTY/+0It+w3epOM7uWjd35PF0rKxxZmEf6srjzd1sk0b9bRy2v9iwn9  
92kuvcfh4vT++PognQLTuqnxFGpD1agrG01xsctJwQXCpfwdtbIdThBgzh4f1Z  
ssA1bCaB1QkzfBPrMzdTIP+AxBg6++K9Sn09N/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>

zdeToof6PaXUTRx7iscCTkn8DUD31g/465ZX5aH3HWFFX500JSPSt0/udqjoQuAr
WA5Jqb//g2GfzzeluzH5Dz3PBbjky8G1IfLm0OXSEigAmpvc/9NjzAgjOW56n3Mu
sjvkibc+11jw+r0o97CfJMpptmcOvehvQv+KGOLZnpibiwmm3v7E6kRy4gEbDu

Determine

Version:	4
User ID:	Bill Buchanan (None) <w.buchanan@napier.ac.uk>
Key Fingerprint(20 Bytes in hex):	d7d10cb24f38079377a25c0bcd158ff6f6aa48c
Key ID (8 bytes in hex):	cda158ff6f6aa48c
Public Key (MPIs in base64):	RSA
CACzpJgZLK/sge2rMLURUQQ6102urs/Gi1GCofq3WPndt5h Tiyp7i03dzvhDahcQ58afVCjQtsty8+K6KZfzQOBgyOS5r FcHcLoC60qy+JoaHupJqHBqGc48/5NU4qbt6f81AQ/H4M+6	

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+wvxqirJbv3xRNf8BAoig6wA1GgEX8M/o1secYRZ24ExAXMcJpvor
4AzHm3I+f4TH/isQ4/yUchteL0onSCVLT04aTZjAuev/paInX7c7mKLLr8vzs04/
7s/jAXcscM4/wtbJx3/bmsu51xs0bscsndlPTCs11scNwo8m5BQ+ya2pRHbA2c
c80tYf6h0pn2MX1NK/SQ73A7KJXINW/m1rh1/6kCex6lMzuoogFTfLaoyn5F2VQ
CeVpgs0S2cbdv7w6wyubERFgz/AN0obAt6rc0jFd1Y7Lhs3yi16imlDPM+yYLFLZH
Hpx8Q5ntB9RJY5XebiyVFjeNc1S7qecJg/9PKmRPAsoc432reiASRGupbTexet5
vqlVfA6/rw2Hde+KT0UAmlQtywwz1fnZLdaiydz86ru0dwmuGiZcu0kIebpegV
AVntafwpGnbtgpsyzsvd==
=tx3F
-----END PGP PUBLIC KEY BLOCK-----
```



Determine	
Version:	4
User ID:	Tim Starling <tstarling@wikimedia.org>
Key Fingerprint(20 Bytes in hex):	bc4195238ffdd816a24d37111075249fcc9caaf
Key ID (8 bytes in hex):	1075249fcc9caaf
Public Key (MPIs in base64):	RSA EAD1mdwc4ON/TgxcUA/Yv5vkVRNct7w2ZHFjkRFey4N8Sw1Qut7m4HR5y0BhgC+ANDkdh/uZE+3GUDZf xba8aqh5UbplR6e4gtUonr4dcVRJ25+8Exe/533415y50IZVkgY99Mm+UienKSRdcJZD2U2/8/YV7Q Xw71t5315EcJ0eLAESCFfL/cFq4YC1vW/ryowid4TaDWxfH9P5qL3JnqcC+vFFaUAxrSc/7C1FFDNwgD Un4kevc8nk6ven3+LE3CIE8qHEwoq9yIhvDRCC+uwf6iv/ZkQyijiu07ulPwPKr0jts51H8JDuGtd+ws eaPqb1Ruq7ia+o18oggJqswoPkEuQwQbh009u8P12xEBJFQ3gj5BV6U/1eD4rvLJT4ZLAnZjCPEWEBF JYZHKxcvWU29U2LMkNm3+Pv7+mtWd1HoceosdswyqebkSomYfvVmW+CMztiwrilmQxkQ2NsV/ySojrzVa



```
ShYQBAC/nfdke+m1k7CxwFW0wxTP2SDXTJkpwnIQ1iR510vj3ApXQ+5HNKwzsgy
g73cppuqv6UotGhMOxyrvCT&PnT4A6BHUTnukbf7c+15MFq6mpTHwlrI4+cr8tei
ptut0jqzedci1053kdw11hqMSyb4bKTGBKHxIzD0v1PxjqqqQwADBqQAn8oS1aJ9
XjL1YPMVx1syhRrgPx1m9+4NH4pbdbimfrv94up0sogwL0zexckhJn1jezhp8q
aPN6cyuudvd1f9E7P/58kkmtdcw1vlizLokt1wjlTcy75qioYdwYKVzrbJW5MTG
VvmcEU+qgljs3vgjsdFfw1LnMzpgAwqu+IrqYEQIABgUCP7XmFgAKCRDBGeGM
TXCTjvw5AKcm18ODP1My1noC8FT5qbj6MSVDvQcg1dacCIseRx4VcunVMuBct8i8
C8t=
=72XL
-----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+m1k7CxwFW0wxTP2SDXTJkpwnIQ1iR510vj3ApXQ+5HNKwzsgyg73cppuqv6UotGhMOxyrvC t8PnT4A6BHUTnukbf7c+15MFq6mpTHwlrI4+cr8teiptut0jqzedci1053kdw11hqMSyb4bKTGBKHxIzD0 v1PxjqqqQwADBqQAn8oS1aJ9xjL1YPMVx1syhRrgPx1m9+4NH4pbdbimfrv94up0sogwL0zexckhJn1je zhp8qaPN6cyuudvd1f9E7P/58kkmtdcw1vlizLokt1wjlTcy75qioYdwYKVzrbJW5MTGvvmcEU+qgljs t3vgjsdFfw1LnMzpgAwqu=

Determine	
Version:	4
User ID:	Sam Reed <reedy@wikimedia.org>
Key Fingerprint(20 Bytes in hex):	1a24253c8ba01e44a8cb470e3bbb95ce2b08bfd2
Key ID (8 bytes in hex):	3bbb95ce2b08bfd2
Public Key (MPIs in base64):	<input type="text" value="ELGAMAL"/>
<pre>CADU1Hwvekd1mjRTkyssqXnp1TkjthJeINPNg15wTXzwQK1twzerrAuK/a1m6FnRPwsG3B1ztN4xOry U1YQ1wnU2jqogwdz609yb4cJ1/j1ggogqazAYerWEW1e+1awBE5Mwdbe18emf7EHPhmmzEdgvUGDBuw OjswoPoZMevom1ucvwyd0Nc1yeBrkTdgneQQYIZHyp2pwsWexs6jXy8rqroZNj4c1z/LueowA5P+kC8G Hauu3jrpj1uvbBh7zrd5edoG1/A/u3AcJ/A2/jkuYjrkztzY7ku1EwxwgokC2Qin+la3qOYGi79y7z2b2xw os7hmmgXpowpEcjirDcf+eTXAAmfCACN0WlkodnAzt0goXIUlENpgoSZM/bbw3D0l/TcpmmmqQwqExb0 2/LdgYQppk0+1vcJ+cJSSqAWEC2Hx3L/SmjbwPLEag9kh6zr1xwzSOTBz2gd12BT0rtqf4camHCit6S</pre>	
<pre>fotworlp1wt8AbzrDahHsgm6U/rRBw18KzRkm6/YUczcvyz3vyf5YFKYqcoj2 YTpw5GadopJFoiAAmN5VbRgH+1PGF6arAdFmVjBSBD8TmhgwHijyujJ88h6ZT/40c +3zm61dfAtx8mxPxbt1f0f1c8cfCTqLd0kk1o5iYZqe2sy35igedPejopqvz6kvux Ne1tkerUJDbxnpkVscu+Tnps40M937wr/jBz7K8MbY5SSURJzeNgd6pstVsyhP Ux77Ajzg1Ihwu51impw1H55ekj8m1BATWDQkzw9c4E77/D3el1rYEM64g7vwkbjIW N70cm8NN5931J3duIQPwde7efoERoxnu1zqFXcbgi5A1zp1vq5vn2vAb0de6mF6A 1q60iA7Jmuob7Qu8vIMC3Cjk4AuUmNqjrxNpvcD81NvtzwhHqy1lfEi2BD9NTzAS J1zEs8pNpnumX =mxX+ -----END PGP PUBLIC KEY BLOCK-----</pre>	
Determine	
Version:	4
User ID:	Mukunda Modell (WMF) <mmodell@wikimedia.org>
Key Fingerprint(20 Bytes in hex):	c83a8e4d3c8feb7c8a3a1998131910e01605d9aa
Key ID (8 bytes in hex):	131910e01605d9aa
Public Key (MPIs in base64):	<input type="text" value="RSA"/>
<pre>EADEwlTiJMMbcbQbw7ggc3we1gFOADZK0h7ifY303k2uhKCLKg3aFuBBw893353zvwyt284bewDhsrq mr+uw8qucwM2Sy7tkJ7z0CsjogdRCbrwriehShLtm8JMTAW1L0Nodxbn155yquh5yjm7An1an9xxkUX9 E2bvsFT8drb8kpghsjoLoezb6FxRpVj7shztavQyvr6AribkXCPi1gsoGU3zpjBFx5+N5ScEx8fwFqhq +Qozdbvbc5/12t8ts1tbqiKLSws95k7shk7DeQ0P8BG1lxqyqHgb3ZV2M1d+txhrBlUvymFWRO9 Z5cjLbz2wU1mx21DlUrRevwq705nbqe4Acknjqgsjvetvgt+kn+t7bqrsJ1wM2/sSKTVUzu1iCwORyWg4IV w416lwvo4yaVMndPsyoE/k9+zdlukpgm7KwigB2lg93PSj5jXN1oopFgh8Dzzls+ubTmoFGkgwy+ztu</pre>	

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

We will use OpenSSL to perform the following:

No	Description	Result
B.1	First we need to generate a key pair with: <code>openssl genrsa -out private.pem 1024</code> 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

		<p>Use the following command to view the keys:</p> <pre>cat private.pem</pre>
B.2	<p>Use following command to view the output file:</p> <pre>cat private.pem</pre>	<p>What can be observed at the start and end of the file: Hyphens and two words BEGIN and END are always present.</p>
B.3	<p>Next we view the RSA key pair:</p> <pre>openssl rsa -in private.pem -text</pre>	<p>Which are the attributes of the key shown: modulus, public Exponent, private Exponent, prime1, prime2, exponent1, exponent2, coefficient</p> <p>Which number format is used to display the information on the attributes: Hexadecimal</p>
B.4	<p>Let's now secure the encrypted key with 3-DES:</p> <pre>openssl rsa -in private.pem -des3 -out key3des.pem</pre>	<p>Why should you have a password on the usage of your private key? This makes the key file by itself useless to an attacker</p>
B.5	<p>Next we will export the public key:</p> <pre>openssl rsa -in private.pem -out public.pem -outform PEM -pubout</pre>	<p>View the output key. What does the header and footer of the file identify? Header and footer tell that the key is a public key.</p>
B.6	<p>Now create a file named "myfile.txt" and put a message into it. Next encrypt it with your public key:</p> <pre>openssl rsautl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin</pre>	
B.7	<p>And then decrypt with your private key:</p> <pre>openssl rsautl -decrypt -inkey private.pem -in file.bin -out decrypted.txt</pre>	<p>What are the contents of decrypted.txt Hello, Lets go on planned world tour.</p>

```
[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-----
MIICXQIBAAKBgQDP5ry6xBdGX7W1ke5GCMStaaRHj+q82fIMEzb8/VxHqyu0Sf7
eNxcwk4dmlVcwUIOX1ZrWRXkXlEfVPVhS/N17qVg2f4Aj/eUwL+cZATMJ/XG44sf
sYzVRdyMNGbUPM4H2+mHC+n9fGWkvPEg52mT10rnTh6tPDXfL0ZPxQrGzQIDAQAB
AoGBAL+vRWmLDXqZTDo23XBKUeQp0cdpNVigDZsk9WiDOVSv7JV0Y0faWorn1/Ax
Y00hfo8T1fEp9SzW08LoqUmpJmK2kMpKP0vE0a0RuPyfz3gQBMaFVkgXG2+vmGTj
6peDsNGTyoc0CqFPJCXNXF90JY1Lv1ExbsAPba50lZIKvDY9AkEA7i6kaJ/r9lnv
m+H9RxaXrSUTV2RVeXNgNw0eioopJhSBEVNJrEYMqYTbee88XgbY4zmge/wa4Eex
uENp/cdUYwJBAN90Md1VjD12knoPdt6vZJJ8cuvIPRcX+a2D95Pmvtrj rQEPv3wD+
q+YGnH2nmhRW6qTTjDj66h3QSMFonqTDZw8CQE0vfYk96RBZjb+wlSy8GeD4bs0
CBY7c1wxnxjAhvELYqJJy0XsAi0qVDSSh9UwnPH1rmWNfxW9SgPTJIu52YECQQDe
A9m/gWMMcnIIurK1gHihnujq39jqyBR31SW4msvGRL8egQcG0wLaUnLL06mgSddr
GxE4T/0q0BZprCPruoX7AkABRb425FnQvC3AeuDfupTFEAozOD2+/7kQmieu8le
U5/TvJ58x3D9Mn0f2RwOpRz9AMZwNjjNcdK2byFoU4ux
-----END RSA PRIVATE KEY-----
[11/28/21]seed@VM:~/.../Exp4$
```

B1 and B2

```
-----END RSA PRIVATE KEY-----
[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:f5:56:6b: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
eNxckw4dmLVcwUIOX1ZrWRXkXLeFvPVhS/Nl7qVg2f4Aj/eUwL+cZATMJ/XG44sf
sYzVRdyMNgBUPM4H2+mHC+n9fGWkvPEg52mT1QrnTh6tPDXfL0ZPxQrGzQIDAQAB
AoGBAL+vRWmLDXqZTDo23XBKUeQp0cdpNVigDZsk9WiDOVSv7JV0Y0faWorn1/Ax
Y00hf08T1fEp9SZW08LoqUmpJmK2kMpKP0vE0a0RuPyfz3gQBMaFVkgXG2+vmGTj
6peDsNGTyoc0CqFPJCNXF90JY1Lv1ExbsAPba50lZIKvDY9AkEA7i6kaJ/r91vn
m+H9RxaXrSUTV2RveXNgNw0eioopJhSBEVNJrEYMqYTbee88XgbY4zmge/wa4Eex
uENp/cdUYwJBAN90MdLVjD12knoPdt6vZJJ8cuvIPRcX+a2D95PmvvtjrQEPv3wD+
·q+YGnH2nmhRW6qTTjDj66h3QSMFonqTDZw8CQE0vfYk96RBZjbf+wLsy8GeD4bs0
CBY7c1wxnjxAhvELYqJJy0XsAi0qVDSSh9UwnPH1rmWNfxW9SgPTJIu52YECQQDe
A9m/gWMMcIIurK1gHihnuqj39jqyBR31SW4msvGRL8egQcG0wLaUnLL06mgSdr
GxE4T/0q0BZprCPr0uX7AkABRb425FnQvC3AeuDfupTFEAoz0D2+/7kQmieu8le
U5/TvJ58x3D9Mn0f2Rw0pRz9AMZwNjjNcdK2byFoU4ux
-----END RSA PRIVATE KEY-----
[11/28/21]seed@VM:~/.../Exp4$
```

```

[11/28/21]seed@VM:~/.../Exp4$ openssl rsa -in private.pem -des3 -out key3des.pem
writing RSA key
[11/28/21]seed@VM:~/.../Exp4$ openssl rsa -in private.pem -out public.pem -outform PEM -pubout
writing RSA key
[11/28/21]seed@VM:~/.../Exp4$ cat public.pem
----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgODP5ry6xVBdGX7W1ke5GCMStaaR
Hj+qB2fIMEzB8/VxHqyu0Sf7eNxcwk4dmlcvwUIOX1ZrWRXXLeFVPVh5/Nl7qVg
2f4Aj/eUwL+cZATMj/XG44sfSYzVRdymNGbUPM4H2+mHC+n9fGWkvPEg52mT1Qrn
Th6tPDXfLOZPxQrGzQIDAQAB
----END PUBLIC KEY-----
[11/28/21]seed@VM:~/.../Exp4$ gedit myfile.txt
^C
[11/28/21]seed@VM:~/.../Exp4$ cat myfile.txt
An initial public offering or stock launch is a public offering in which shares of a company are sold to institutional investors and usually also retail investors. An IPO is typically underwritten by one or more investment banks, who also arrange for the shares to be listed on one or more stock exchanges.
[11/28/21]seed@VM:~/.../Exp4$ openssl rsautl -encrypt -inkey

```

B4 and B5

```

[11/28/21]seed@VM:~/.../Exp4$ openssl rsautl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin
RSA operation error
3070654144:error:0406D06E:rsa routines:RSA_padding_add_PKCS1_type_2:data too large for key size:rsa_pk1.c:153:
[11/28/21]seed@VM:~/.../Exp4$ gedit myfile.txt
^[[A^[[A^C
[11/28/21]seed@VM:~/.../Exp4$ openssl rsautl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin
[11/28/21]seed@VM:~/.../Exp4$ cat file.bin
0000hMPt0U)Z[11/28/21]seed@VM:~/.../Exp4$ A%>@0bmm0000U
[11/28/21]seed@VM:~/.../Exp4$ cat decrypted.txt
cat: decrypted.txt: No such file or directory
[11/28/21]seed@VM:~/.../Exp4$ openssl rsautl -encrypt -inkey public.pem -pubin -in file.bin -out decrypted.txt
RSA operation error
3070535360:error:0406D06E:rsa routines:RSA_padding_add_PKCS1_type_2:data too large for key size:rsa_pk1.c:153:
[11/28/21]seed@VM:~/.../Exp4$ openssl rsautl -decrypt -inkey public.pem -pubin -in file.bin -out decrypted.txt
A private key is needed for this operation
[11/28/21]seed@VM:~/.../Exp4$ openssl rsautl -decrypt -inkey private.pem -pubin -in file.bin -out decrypted.txt
A private key is needed for this operation
[11/28/21]seed@VM:~/.../Exp4$ openssl rsautl -decrypt -inkey private.pem -in file.bin -out decrypted.txt
[11/28/21]seed@VM:~/.../Exp4$ cat decrypted.txt
An initial public offering or stock launch is a public offering in which shares of a company are sold.
[11/28/21]seed@VM:~/.../Exp4$ 

```

B6

```

[12/02/21]seed@VM:~/.ssh$ ssh-keygen -t rsa -C abhishek.chopra@spit.ac.in
Generating public/private rsa key pair.
Enter file in which to save the key (/home/seed/.ssh/id_rsa): keys.txt
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Saving key "keys.txt" failed: Permission denied
[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):
Created directory '/root/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:PN40qNDy+zflcTa0uZcrVPTqk4vmeAKwoZ0Da/JCwXw abhishek.chopra@spit.ac.in
The key's randomart image is:
+---[RSA 2048]---+
| . . . . |
| + E . o . o |
| o .0o+S o o.o |
| . . =+.+ ...o+=|
| . . oo ....o =+|
| o o . +.oo=o. |
| =. ....*o o+o |
+---[SHA256]---+
[12/02/21]seed@VM:~/.ssh$ 

```

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+0hYr2NyNCpVWBv71IPLE abhishek.chopra@spit.ac.in
The key's randomart image is:
+---[RSA 2048]---+
| .. .
| =+..
| ==+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 AAAAB3NzaC1yc2EAAAQABAAQCsGgN37LgQoRzlFdh7p8oq/tYpMntAVI3F53k/ECYkeKGdqLCq624tmN9VCdv3l0oQPZsuvTDqF9p
6AiYY/v/FAJiEWu7IuJeHKUcJj4Yw7tlTdU7xCcDv9nybk00w5iE1/9xN9bVp6BTPDh03kjGrfVmRk/nVOFLUP/j18dm0QpI+xvk3KHoVXETdnNm
SB/l8DJ3qe5bqWqZbGl3rWt+rsAPlDu/lttggtmm6Chf/bRdTvaYJN/Cj7hTRpe7NiPSHPbJtgjF44rqtJvx1T/4nz0HTbn8lidSV1GRw0Iplnl+0qnGT
GKL8sJUTP4eBcH7hYL50lSw17kkWD 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  
AAAAB3NzaC1yc2EAAAQABAAQDLrriuNYTyWuC1IW7H6yea3hMV+rm029m2f6Iddt1ImHroXjNwYyt4E1kkc7Azo  
y899C3gpx0kJk45k/ClbPnrHvkLvtQ0AbzWEQpoKxI+tw06PcqJNmTB8ITRLqIFQ++ZanjHWm20dew/514y1dQ8ccCO  
uzeGhL2Lq9dtfhSxx+1cBLCyoSh/lQcs1HpXtpwU8JMXwJ1409RQOVn3g0usp/P/0R8mz/RWkmisFsyDRLgQK+xtQxbpb0  
dpnz51IOPwN5LnT0si7eHmL3WikTyg+QLZ3D3m44NCeNb+b0JbfaQ2ZB+1v8C3Oxy1xSp2sxZPZMbrZWqGSLPjgDiFIBL  
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	<p>First we need to generate a private key with:</p> <pre>openssl ecparam -name secp256k1 -genkey -out priv.pem</pre> <p>The file will only contain the private key (and should have 256 bits).</p>	<p>Can you view your key? Yes</p> <p>Now use “cat priv.pem” to view your key.</p>

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 How many bits and bytes does your private key have: 256 bits
C.3	Now generate your public key based on your private key with: <code>openssl ec -in priv.pem -text -noout</code>	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

```

[12/05/21]seed@VM:~/.../CSS$ cd Exp4/
[12/05/21]seed@VM:~/.../Exp4$ openssl ecparam -name secp256k1 -genkey -out priv.pem
[12/05/21]seed@VM:~/.../Exp4$ cat priv.pem
-----BEGIN EC PARAMETERS-----
BgUrgQQACg==
-----END EC PARAMETERS-----
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEMryFcQKuHWj07ES9LttUxiTdhYJmwY2+y56d1JquY6+oAcGBSuBBAAK
oUQDqAEwuHntewbHy/2q3lNp6Xxo+Yd9C83zA0yMBISvmmmtA9I7aUBnRlitofz5
FR3wCsdC1HxgXhmCk80UbKb8vipB5w==
-----END EC PRIVATE KEY-----
[12/05/21]seed@VM:~/.../Exp4$ openssl ecparam -in priv.pem -text -param_enc explicit -noout
Field Type: prime-field
Prime:
    00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
    ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:fe:ff:
    ff:fc:2f
A:      0
B:      7 (0x7)
B:      7 (0x7)
Generator (uncompressed):
    04:79:be:66:7e:f9:dc:bb:ac:55:a0:62:95:ce:87:
    0b:07:02:9b:fc:db:2d:ce:28:d9:59:f2:81:5b:16:
    f8:17:98:48:3a:da:77:26:a3:c4:65:5d:a4:fb:fc:
    0e:11:08:a8:fd:17:b4:48:a6:85:54:19:9c:47:d0:
    8f:fb:10:d4:b8
Order:
    00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
    ff:fe:ba:ae:dc:e6:af:48:a0:3b:bf:d2:5e:8c:d0:
    36:41:41
Cofactor: 1 (0x1)
[12/05/21]seed@VM:~/.../Exp4$ █
[12/05/21]seed@VM:~/.../Exp4$ openssl ec -in priv.pem -text -noout
read EC key
Private-Key: (256 bit)
priv:
    00:ca:f2:15:c4:0a:b8:75:a3:d3:b1:12:f4:bb:6d:
    53:18:93:76:1c:89:9b:06:36:fb:2e:7a:77:52:6a:
    b9:8e:be
pub:
    04:c2:e1:e7:b5:ec:1b:1f:2f:f6:ab:79:4d:a7:a5:
    f1:a3:e6:1d:f4:2f:37:cc:03:b2:30:12:12:be:69:
    ad:03:d2:3b:69:40:67:47:58:ad:a1:fc:f9:15:1d:
    f0:0a:c7:42:94:7c:60:5e:19:82:93:c3:94:6c:a6:
    fc:be:2a:41:e7
ASN1 OID: secp256k1
[12/05/21]seed@VM:~/.../Exp4$
```

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.

(ECDH - Elliptic Curve Diffie Hellman) which can be used for create a secure tunnel:



```

Message          Hello. Alice
Determine
Key             test
G

++++Keys+++++
Bob's private key: 45edc9892a1bfeb99be14ac779b6ac3d7408aaaf14cedf946be9cefd967410ef11aa0
Bob's public key:
0402e2499a21c59ca4bd3162d306a7824f74c13731ac0c7726130eb6b84ca8f239a4c0d5830171341edb40ff7d75f7214996089a8669681c17724
e309cc5e40698a6bab3460706073c

Alices's private key: 0237912371e5e8d8d34496001aca0238d76608da3f667a350069911f3dfd9a659f7aeb70
Alices's public key:
04010f0c0f1760c3925de0cd7d572a4ff8de28852f2e19e9f03f80388b40a6d01a40d151706ddb775b832136e60806fa44b6b990101a7dd5ac68
0197fb95fe91ac411d9ba9bebe2

++++Encryption++++
Cipher:
1c8e22ff49ec79af6fdc5b6485cd287c0405cbdedec4d62e0b19b15dc77480053db548b7405f8c203402f496cce0a015569650faa803cff961e8e
cb861e9f80183e16ab3027a8b195ad4830f4659a3bbc6f9901124735d000b0b94769c91bc32da9813a4b2dbc6255620334ada7e40992026a2282f
dcf8f6e672da19b666a22dcf8b52cb1488843c51
Decrypt: Hello. Alice

Bob verified: True

++++ECDH++++
Alice:041af59cdff93317c1d238d0b9d74a68a563bd983345c204773f78fc5e62c569
Bob: 041af59cdff93317c1d238d0b9d74a68a563bd983345c204773f78fc5e62c569

```

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

[Encryption Home][Home]



For a finite field elliptic curve we can search for the first 20 points for a curve of $y^2 = x^3 + ax + b$ and for a defined prime number (p)
[\[Improved version\]](#) Note: this version will not find all the ECC points.

Parameters

a: 0

b: 7

Prime: 89

Determine

Examples

The following are some examples:

- $y^2 = x^3 + 7, p=23$. Try!
- $y^2 = x^3 + 7, p=101$. Try!
- $y^2 = x^3 + 7, p=802283$. Try!

A: 0

B: 7

Prime number: 89

Elliptic curve is: $y^2=x^3+7$

Finding the first 20 points

(14, 9) (15, 0) (16, 3) (17, 5) (22, 8) (24, 6) (40, 4) (60, 2) (70, 1) (71, 7) (103, 9) (104, 0) (105, 3) (106, 5) (111, 8) (113, 6) (129, 4) (149, 2) (159, 1) (160, 7)

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+LWLD7wr213HSmbcyGkQICNpgx9yyeOjwyOByqHDKR//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 $(e.d) \pmod{\text{PHI}} = 1$:

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

Now decrypt your ciphertext with:

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

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

```

Run: rsa
D:\Python39\python.exe E:/Exp4/rsa.py
5 161
53 161
123
9
Process finished with exit code 0

```

P =23, Q=7,e=5

E.2 In the RSA method, we have a value of e, and then determine d from $(d \cdot e) \pmod{\Phi} = 1$. 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

```

```
pk4HJ47gkovucd\n\r/p1T52kJ47ajyV1Na0GZUu8P1vC0HBvB78HKMaggc5FGC0Qo07cu9qM1w1zeIix\r\nnr18o31fhpI5zxKn4FPzuSOP1MYDXHSvKvCm3y
8qDauqiX9vOgyN/ID1vJHGEPes\nmrn3r2H0xazyJ\aiJBCKBaBmRgnFDSJNw1eA3FvP108n1rlyIsj\r\n0+kx9k5arp622dv\nns3naMo01laEw7rKtWhapAs1
79ktb2AYIQBKEVAtd9eqoBse2qsVzGzpKp+QGPJ\r\nngevUK3EcgyEAW1a38nGNGVQBa1C1dzedMMRhd0f1QD1YQZh4ZvCkhyhS3q3ua9QW\nhaqBkQpKB980
02Aa+1Y2k89Rq8z2zaYQsuzinjPF0Pd5ruhqs06C7ChZc8g0z+vNndyTsphy/esd/x+A43kJy0f5k5enV61X1qHDHPCupdKhFhv+iLhw5TUgYE2Szj\n/nF
9vMr6a/R7QJnMFHDWJ2VeFvGqVK1dkJZR0Zt9DZah11KefefnxDo0AhjIw+N+\n0w70oYkEv90Tv6S0dNLgDCga/qdsENvsLce7giL63nWciD+9pdtdQMd
nhb/2LwFx,nE1Ue2s0knKftvsYqK2P7MVUZK7zcZIFxijslo6ZEcgYAvzG5awfDr03ppa0jii+d+\nLgGmfkk+qbS/tuRloFwfZCIsQeLvtGpwKGR7St3oqby
TF8isj8HCGMThMdN1dfpk\r\nnjqoGOESXQ9NV61nNx3ACDMtpD6wMlyfaQJ18qQx2Ms6Vst1hrtXBZjhPrFAT/nbrAqUL05NrWc27Ts18sAoAy4UCgYA0THK
BRia/p0Faw50VdTe1+x4fwlyLs9r5PDH/n6ewLG8S6iRthbxwM51WdhSRegDaaNyHMWx+Qenk6Y1kiD74Y/FxH3pT5gsRbo/na1Vhmc0+6CjMzBraat
DPsNICzuXicPcxM2u50nBzqVQ/rTdIipeP3G8u2Bp0fK/nwf+FwgQkBghu4WkDtqxs793o4mYdQ55dd/Vpjx3+32pTzLBk8nB1effF5pi7baXry/nNpdWDAh
91ksLot6Kth8R36AS978RBi56bB18Jpt14GRbVpvbq2IMEfdm1c7SEZ\nnKty0rMS5RCxCX+6tSNxd1lswsbEGsewKmkfx+/Asg/KRN0Tf/kkEE\n----EN
D RSA PRIVATE KEY-----'
```

DER files are digital certificates in binary format, 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

xk0EXEOYvQECAIpLP8wfLxzc0lMpwgzcuzT1H0icgg0IyuQKsHM4XNPugzu
X0neawrJhf1+f8hdroj5Fv8jB10m/KwFMNT8AEQEAc0UymlsbCA8ym1s
bEBob21LmNvbT7CdQQQAQgAHwUCXE0VQYLQcIAwIEFQgKAgnMWAgECGQEC
GwMChgEAcgkQ0NsXEDYt2ZjktAH/b6+pdFQLi6zg/Y0tHs5PPRv1323cw0ay
vMcPjnWq+vFiNyXzY+UJKR1PxskzDvHMLoyVpuCjle5ChyT5Low/ZM5NBFXd
mL0BaGd1TsT06vvQxu3jmflzKMAR4kLqqIuFRCapruHYLojw1gJzs9p0bf
S0qS8zMEGpN9QZxkG8YeCh3gHx1rvALTABEBAAHCxwQYAQgACQUCXEOYvQ1b
DAAKCRCg2xcQNi3zmAGAF9w/XazfELDG1w3512zw12rKwM7rk97aFrTxz5w
XwA/5gqvOp0iqxk1b9qpX7Rvd6rlku7zoX7F+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
TJzgvf4-fLOSLsNYP64QfNah50744y0MLV/EZT3gsbw09v4XF2SSzj6+EHbk
09gwI31BAIDgSaDsJYf7xPohp8iEWwwrUkC+j1GpdTsGDjpeyMISvV8Ycam
0g7MSRL+dYQauIgtVb3d1oLMPTul59nVAYuIgd8HxyAH2vsegszS0n0kVf
+dweqJxwFM/uX5PVKcuYsroJFBEO1zas4ERFxbbwnsQgNHpjdiPueHx6/4EO
b1kmhod6UT7bamuy7bcmalPBsv8PH31jt8SzRriawxsIDxiawxsQghvbwuu
Y29tPSJ1LBABCAsfbQjCQ5i9BgsJbwgDAgQVCAoCAXYCAQIZAQIBAwIeAQAK
CRCg2xcQNi3zmORMaf9vr6kn9AuLrod9j50dlk89G/FbdzchrK8xw+Odar5
V+i3Jfnj5QkpHU9eyTM08cws7Jw1Ry0V7KKHJPks7D9kx8BmBFxDmL0BaGdY
1TsT06vvQxu3jmflzKMAR4kLqqIuFRCapruHYLojw1gJzs9p0bFS0qS8zME
GpN9QZxkG8YeCh3gHx1rvALTABEBAAH+CQM12Gyk+BqvOgzgX3C80jRLBRM
T4sLCHOUglwaspe+qatOVjeEuxASduSSoBvMrw7mJYQZLttjNKFAT921SwfXY
gavS/bIL1w3QGA0CT5mqijkr0nurKkekKDsgkjvbioplmyHfepPOju1322
Nw4V3J004LBh/sdggbrnw3lhHEK4Qe70ciert8C+S5xfG+F5RWAdi5HR8u
UTyH8x1h0zr0F7K0Wq4UCNrUm6c35H61c1c4zaar4JSN8fZPqvKL1HTVCL9
1pDzxxqXkJ505KXXzbh5w18EGAEIAAkFA1xMl0CGwWAcgkQ0NsXEDYt2Zja
BgH/cP12s3xCwxtVt+zds8Ndqysd06yve2ha7cc+v18AP+YKqFT9IKMZJw/a
qV+0Vxeqyyru86F+xfrEKHdbA1qzMA== =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	<p>Create a key pair with (RSA and 2,048-bit keys):</p> <p>gpg --gen-key</p> <p>Now export your public key using the form of:</p> <p>gpg --export -a "Your name" > mypub.key</p> <p>Now export your private key using the form of:</p> <p>gpg --export-secret-key -a "Your name" > mypriv.key</p>	<p>How is the randomness generated?</p> <p>Randomness is generated using pseudorandom number generation.</p> <p>Outline the contents of your key file:</p> <p>The file contains header and footer with naming conventions begin and end</p>

2	<p>Now send your lab partner your public key in the contents of an email, and ask them to import it onto their key ring (if you are doing this on your own, create another set of keys to simulate another user, or use Bill's public key – which is defined at http://asecuritysite.com/public.txt and send the email to him):</p> <pre>gpg --import theirpublickey.key</pre>	<p>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</p>
3	<p><i>Now list your keys with:</i></p> <pre>gpg --list-keys</pre> <p>Create a text file, and save it. Next encrypt the file with their public key:</p> <pre>gpg -e -a -u "Your Name" -r "Your Lab Partner Name" hello.txt</pre>	<p>What does the -a option do: Create ASCII armored output.</p> <p>What does the -r option do: Encrypt for user id name.</p> <p>What does the -u option do: Use name as the user ID to sign.</p>
		<p>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.</p>
4	<p>Send your encrypted file in an email to your lab partner, and get one back from them.</p> <p>Now create a file (such as myfile.asc) and decrypt the email using the public key received from them with:</p> <pre>gpg -d myfile.asc > myfile.txt</pre>	<p>Can you decrypt the message: Yes</p>

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)key/(Q)uit?
Change (N)ame, (C)omment, (E)mail or (O)key/(Q)uit? 0
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, lu
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	<p>Go to your Kali instance (User: root, Password: toor). Now Create a new volume and use an encrypted file container (use <code>tc_yourname</code>) with a Standard TrueCrypt volume.</p> <p>When you get to the Encryption Options, run the benchmark tests and outline the results:</p> 	<p>CPU (Mean)</p> <p>AES: 2.8 GB/s</p> <p>AES-Twofish: 535 MB/s</p> <p>AES-Two-Serpent: 294 MB/s</p> <p>Serpent -AES : 524 MB/s</p> <p>Serpent: 530 MB/s</p> <p>Serpent-Twofish-AES: 290 MB/s</p> <p>Twofish: 648 MB/s</p> <p>Twofish-Serpent: 315 MB/s</p> <p>Which is the fastest: AES</p> <p>Which is the slowest: Twofish-Serpent- AES</p>

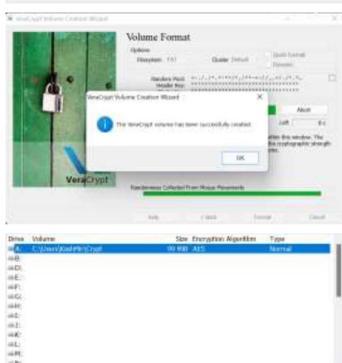
2	Select AES and RIPMD-160 and create a 100MB file. Finally select your password and use FAT for the file system.	What does the random pool generation do, and what does it use to generate the random key?
3		Random pool tracks the user mouse movement And generates a strong cryptographic key.
4	Now mount 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 the files? 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

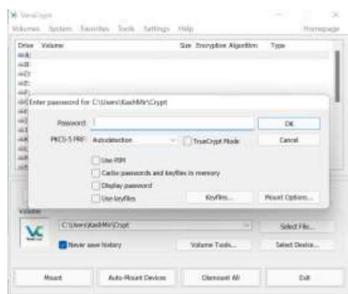
1



2



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 "====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\nnO2QX9eo6p8mommqK6CJ
0ybv4Axun+K1+Dk4arbQLt1Y9CMxvXSP6wB1qd4Y74w\nbj2jyQXZU3zJTN/HxC9rq8swLBEnL2sa9Zfi0Sn05Dj1Ni23L+NgV91XRQIDAQAB\nnAoGAAK9
MARPx4Bx6dvxqKKDYGUWUYGFv39NmxtAfTwws5nb7QqnGRm+OF\nnLWIM5RBOnKnImmt+X3R/Z4sUX9AEsm363UrOS5RQsfEkslijNYJcaC6fVbsn
W9Q\rnnS556fSqph0Sk7NsH7t07QNLFXccQ/1F9wdYXLrXuKhVaqkCQDdodToRi15Ae4G\nn1WE6Y1yOisdzFcoahTZNiDQytgddc1s0SCy9xz078IHxN+D
QHfsE4usLlUstGw\nnYmp4005AKEA+coEZIBLPKxcza8u81kPjDe8zYhaG40HPEItUzEhrRbrlhdaG6\nnVIGS9zm2r9e0Az/JxBHSLScSt438M37QJAW
Lnc3x1CSBjzeLHV7cp1To951e/M\nnj7jfFFLz0ZNnVksUv9T4turX7JodTnolmtT7NgzzrU+r5SiWfs7ZUgiQJBAK11\nnEq2FQ4Grptdeghvfwed8Aqo1H
1rOp8uAkHZGdcqYLJPggHQzbY2G310ezjYsZ8\nnw1weckgYo1q77fvj9ECQCAvAY8d+v1krYhE4f++HBZ7awCqqdMjwU+q73zVAst\nnquaPn6Fvqh8Yq
dKSSreJrY7AQqeLyOT1CFqwYnNZxG5+\n-----END RSA PRIVATE KEY-----'

====Public key====
b'-----BEGIN PUBLIC KEY-----\nMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDi0H18X1zgtZwsTN7/jyluPBQn\nn1mClEwo4AVWiAQQlW8xFG6sn
02QX9eo6p8mommqK6CJ0ybv4Axun+K1+Dk4arbQLt1Y9CMxvXSP6wB1qd4Y74wbj2jyQXZU3zJTN/HxC9rq8swLBEnL2sa9Zfi0Sn0\nn5Dj1Ni23L+Ng
V91XRQIDAQAB\n-----END PUBLIC KEY-----'
====Ciphertext====
b'iH0/2u6LlimOOltJ5canJXaUxj4g06t6GUSV+m1P2PrMcHQtoW10mCdEvboHDYtspzWo++TnL1IgJYC1cXR6MsKcFBgtLX5Bek3KfjaG8aKR080FILvL
DhfRatfJe/Ca/61tetSGrzAeX1l3TzrWnbSshZGjoP8gBIEJn1ELY='

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

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