**NAME :** Mansi Dwivedi

**BRANCH :** IT

**UID :** 2019140016

**BATCH :** A

**COURSE :** CSS LAB

**EXPERIMENT :** 4

**AIM :** The aim 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.

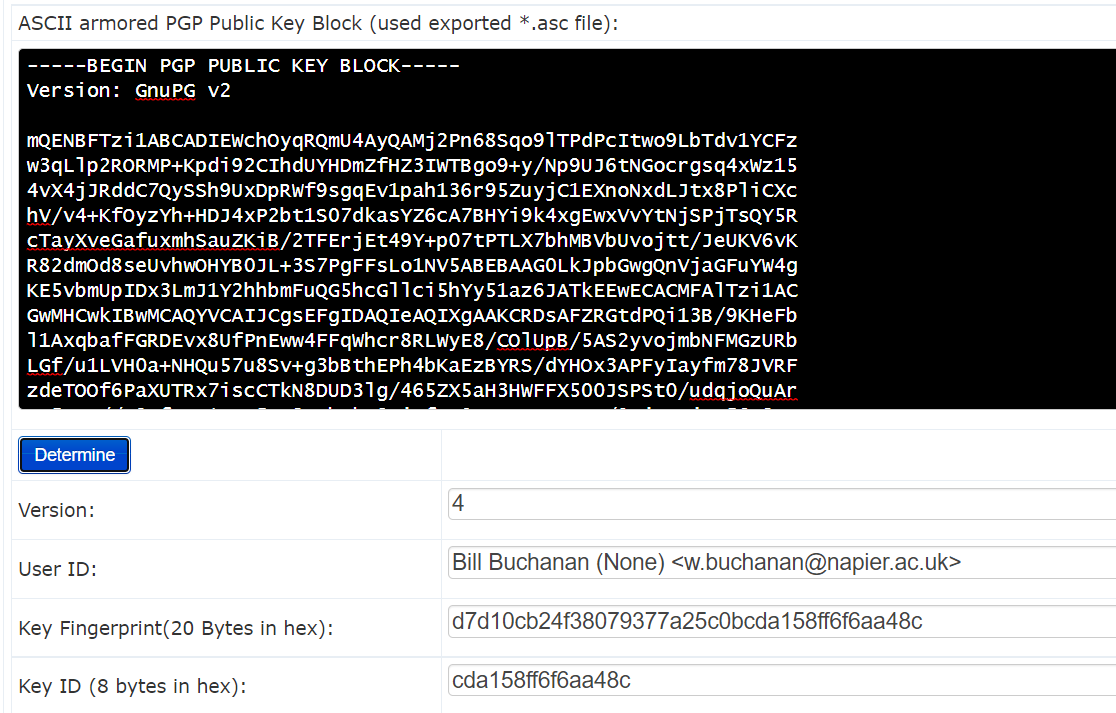
& **Web link (Weekly activities):** [Unit 4: Public Key](https://asecuritysite.com/esecurity/unit04)

& **Video demo:** [Lab 4: Asymmetric Encryption](https://youtu.be/6T9bFA2nl3c)

**PROBLEM STATEMENT :**

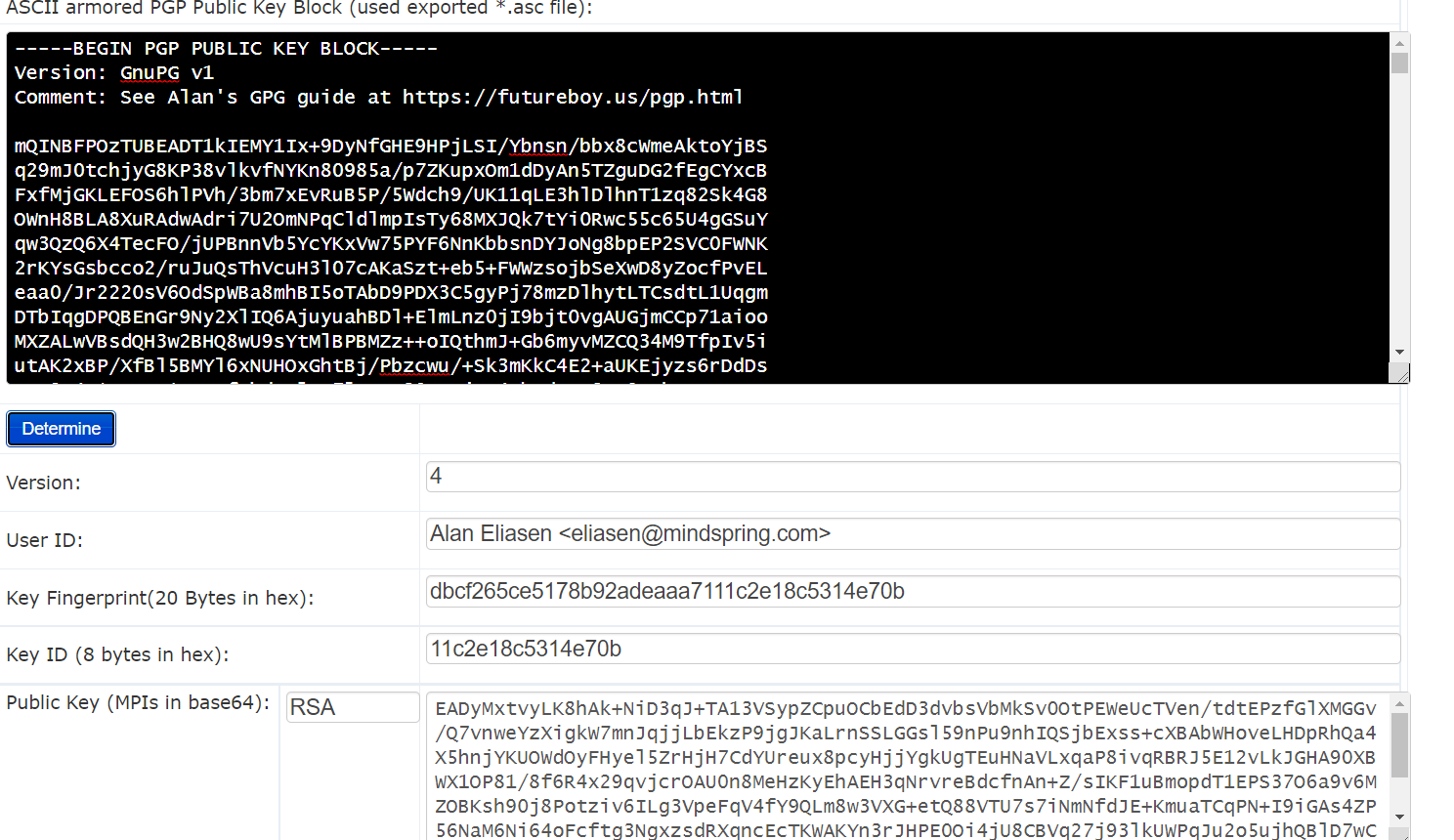
1. **RSA Encryption**

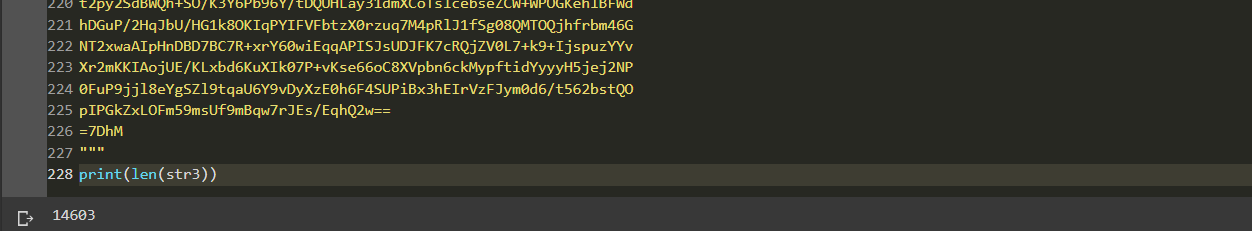
**A.1) OUTPUT :**

****

**Public Key of three famous people :**

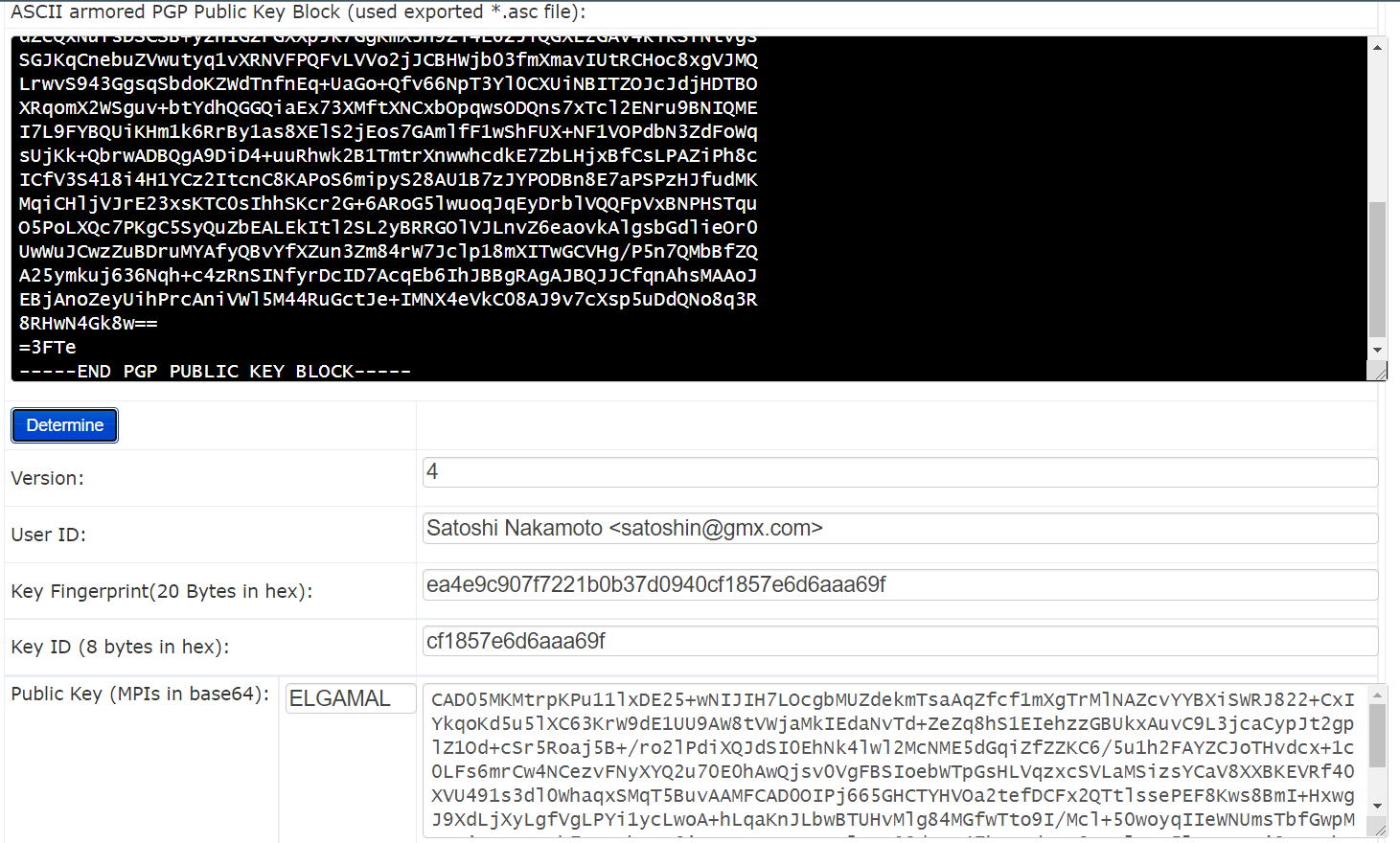
* **ALAN ELIASEN (Developer of Frink language)**

****

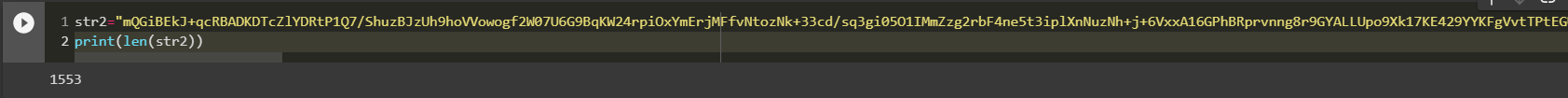
**Key Size : **

**Key Encryption Algorithm : RSA**

* **SATOSHI NAKAMOTO**

****

**Key Size :**

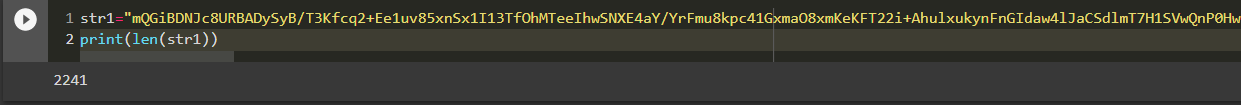
****

**Key Encryption Algorithm : ELGAMAL**

* **PHILIP ZIMMERMANN (Creator of Pretty Good Privacy )**

****

**Key Size :**

****

**Key Encryption Algorithm : ELGAMAL**

**What is ASCII armored Message ?**

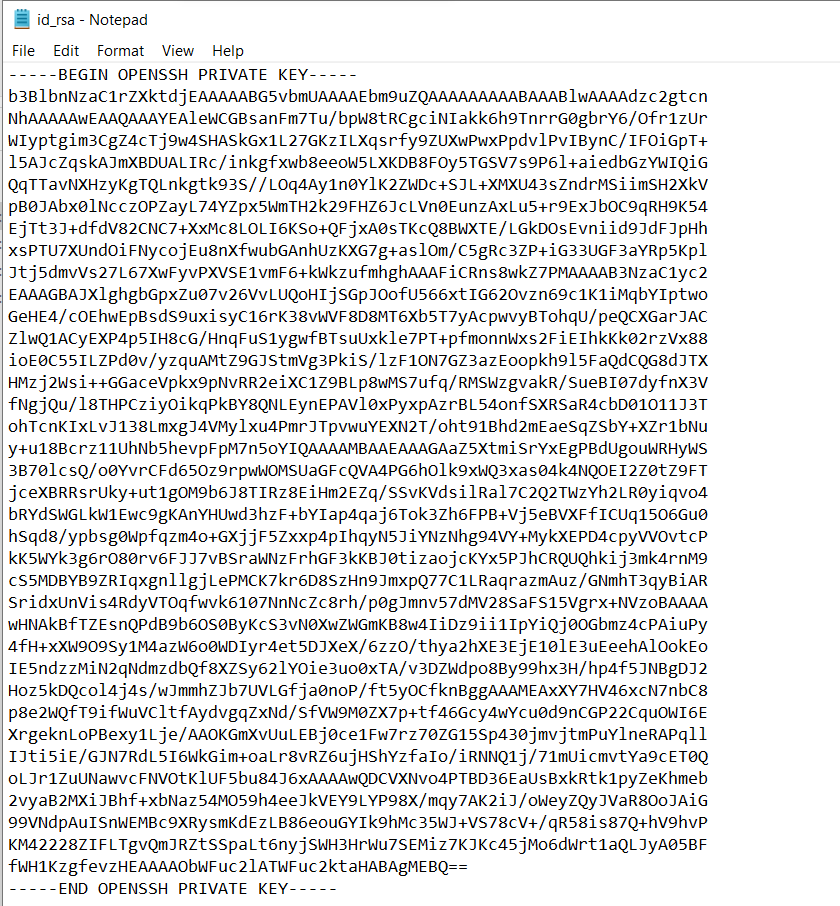
ASCII armour is a binary-to-text conversion tool. It is the feature of a type of encryption known as pretty good privacy (PGP). Encrypted messages are encased in ASCII armour so that they can be transmitted in a normal messaging medium like email.The reasoning behind ASCII armor for PGP is that the original 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.

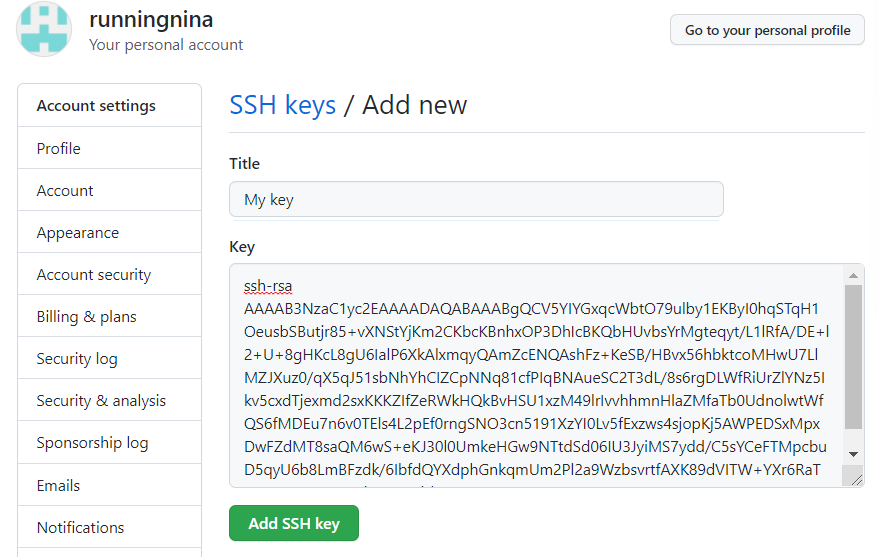
**A.2) OUTPUT :**

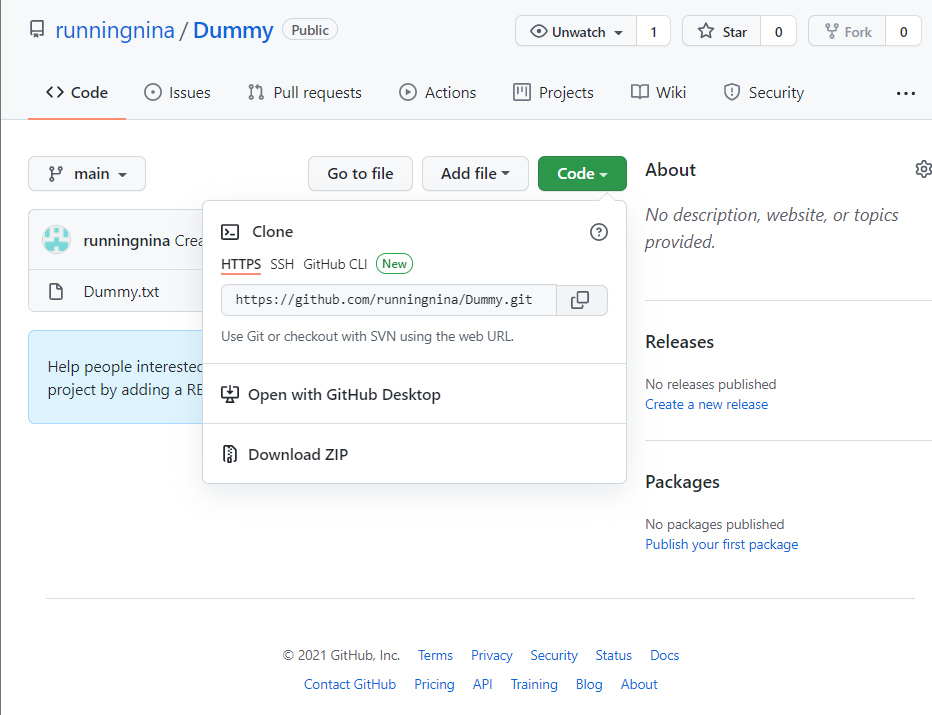
1. **Openssl RSA**

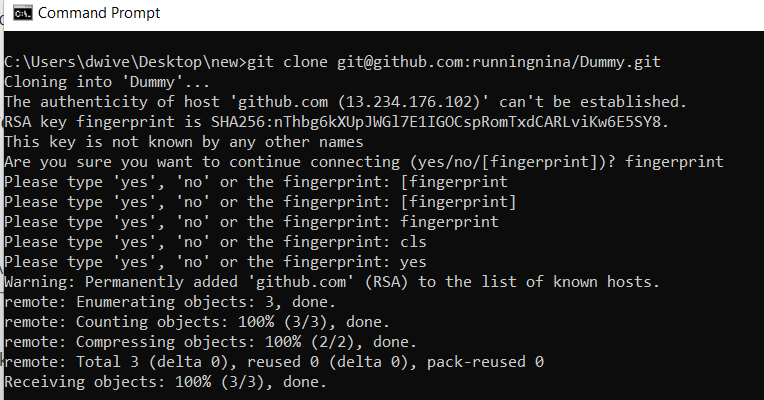
| **No** | **Description** | **Result** |
| --- | --- | --- |
| **B.1** | First we need to generate a key pair with:  openssl genrsa -out private.pem 1024  This file contains both the public and the private key | What is the type of public key method used: **RSA key generation algorithm**  How long is the default key: **1024 bits**  How long did it take to generate a 1,024 bit key? : **About 1 second** |
| **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** * **Coefficient**   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 provides a layer of security on the key. Otherwise anybody who steals the file from us can log into 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 | View the output key.    What does the header and footer of the file identify?  **It represents that the key stored in this file is the 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 ? |

**PART 2 :**

****

****

****

****

Yes I was able to clone the repository.

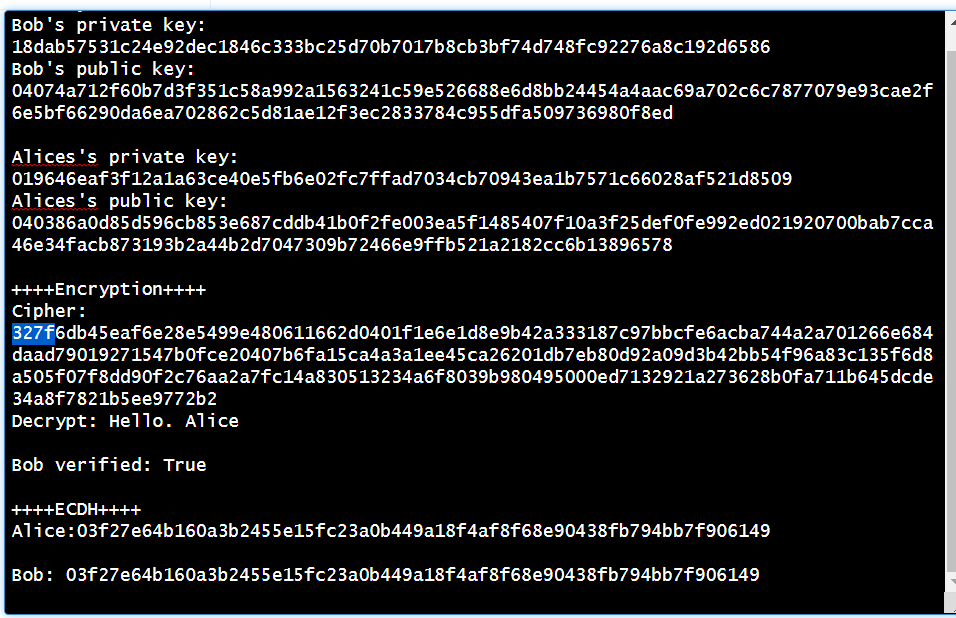
1. **Openssl ECC**

| **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. |  |
| **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** |

1. **Elliptic Curve Encryption**

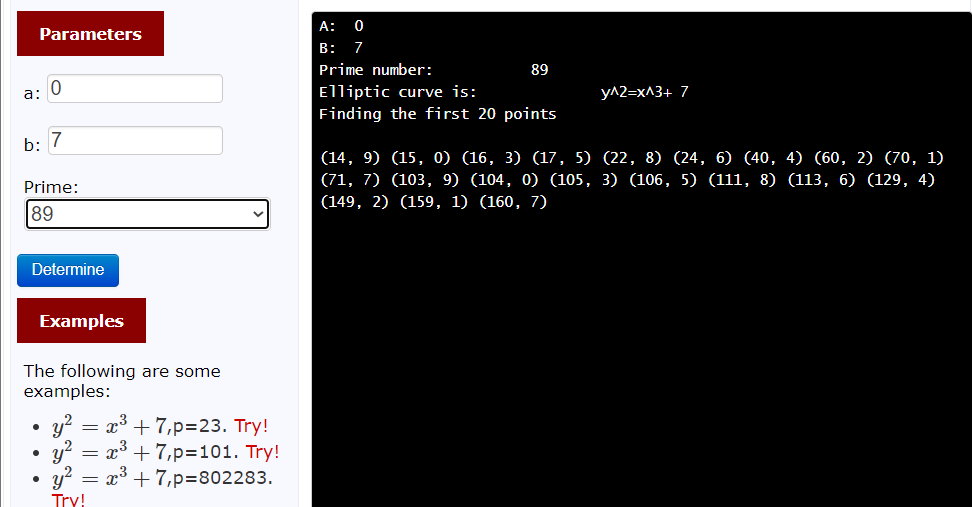
**D.1) OUTPUT :**

**The first four characters are 327f**

****

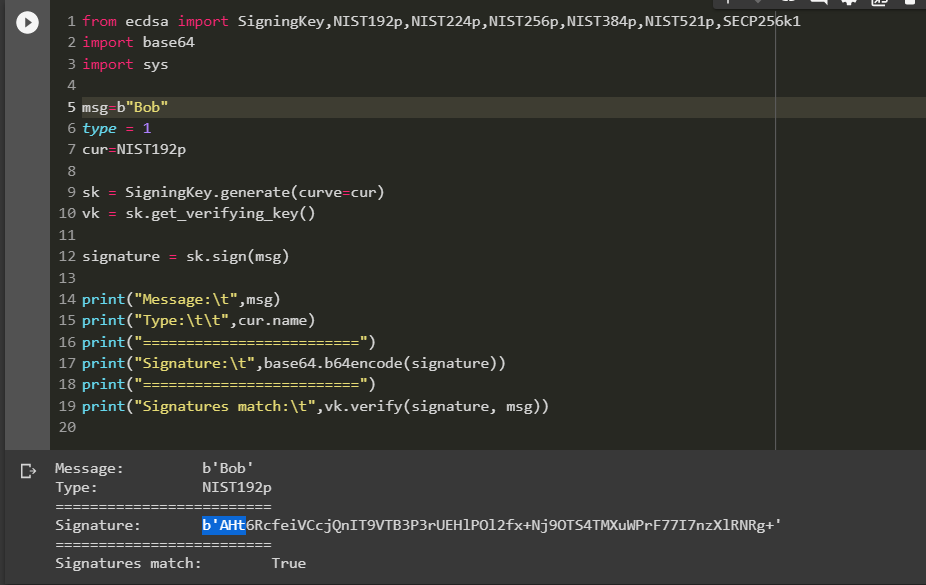
**In this case since Bob is encrypting the message and sending it to Alice, she first needs to verify that the message actually came from Bob so the signature created by Bob along with Bob’s public key is used to verify his authenticity.**

**D.2) OUTPUT :**

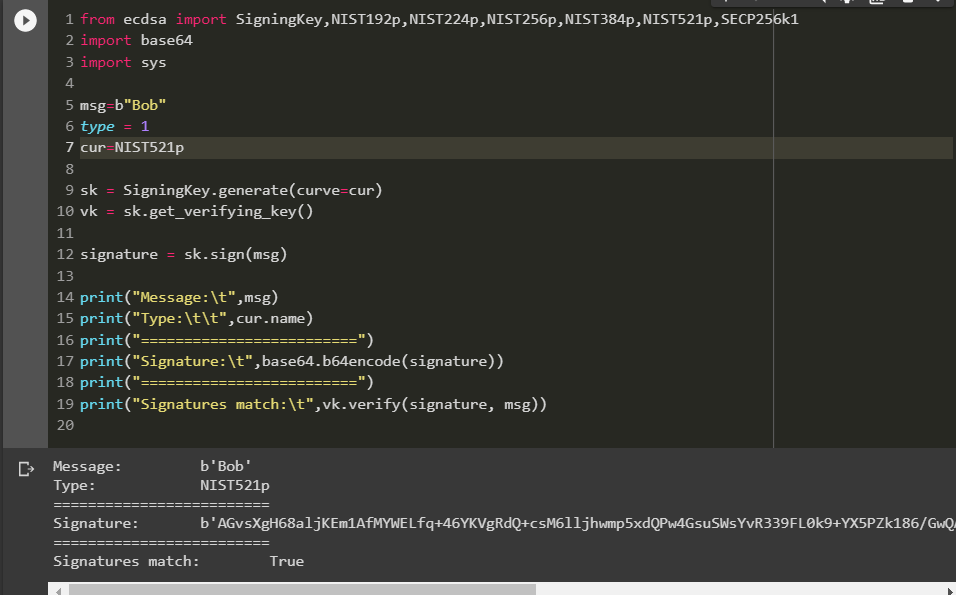
****

**First Five Points are : (14,9) , (15,0) , (16,3) , (17,5) , (22,8)**

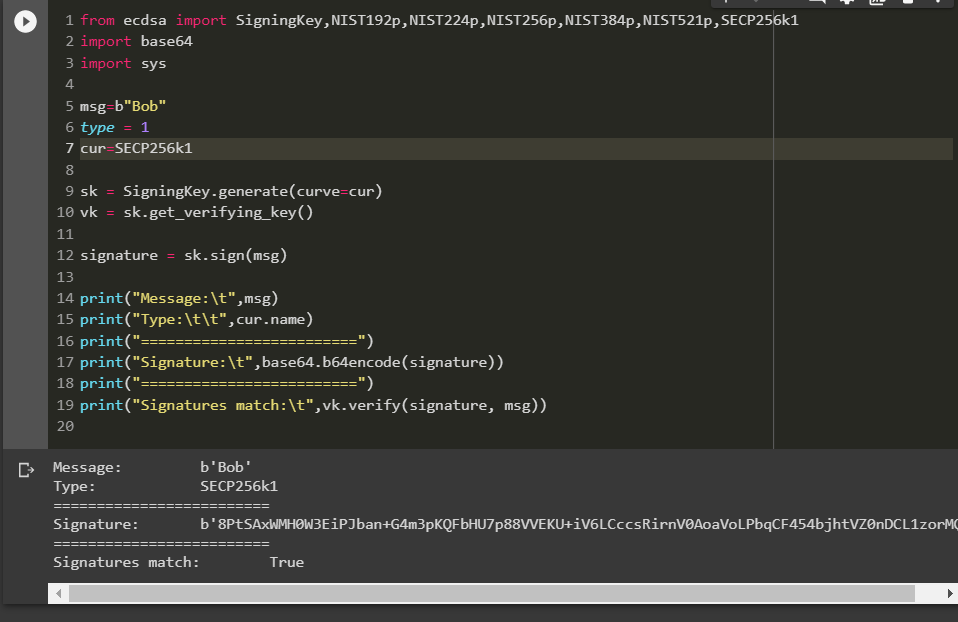
**D.3) OUTPUT :**

**NIST 192p: **

**NIST 521p:**

****

**SECP256k1:**

****

**Application Areas of SECP256k1 :**

SECP256k1 is used by Bitcoin since ECDSA, has some nice properties such as the structure that ‘allows for especially efficient computation’, as well as ‘significantly reduces the possibility that the curve's creator inserted any sort of backdoor into the curve’. Due to its nice properties, the curve has been introduced to the EC-based encryption schemes and can be applied in the elliptic curve Diffie-Hellman as an anonymous key agreement protocol. In Ethereum, the elliptic curve integrated encryption scheme implemented with the SECP256k1 curve is used in the RLPx transport protocol. These applications fit the scenario of our attack where the base-point can be chosen by the attacker. For its increasing application, we use SECP256k1 as the concrete curve parameters to apply the aforementioned attacks.

**Difference between Hashes :**

The difference between the hash signatures that I observed was their size, since NIST192p uses 192 bit size elliptic curve the corresponding hash signature is 64bit and as this elliptic curve size increases like in NIST512 and SECP256 the size correspondingly increases.

1. **RSA**

**E.1) OUTPUT :**

**Picking the prime numbers**

**P = 11**

**Q = 2**

**N = 22**

**PHI = 10**

**e = 3**

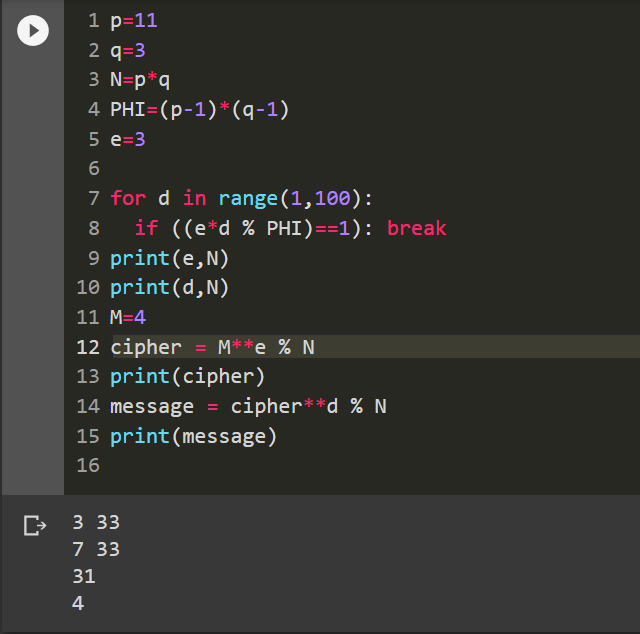
**d = 7**

**Now if M = 5 then cipher is calculated as**

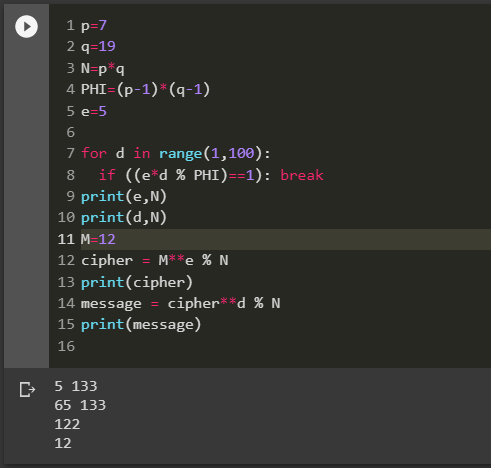
**C = 5^e (mod N) = 15**

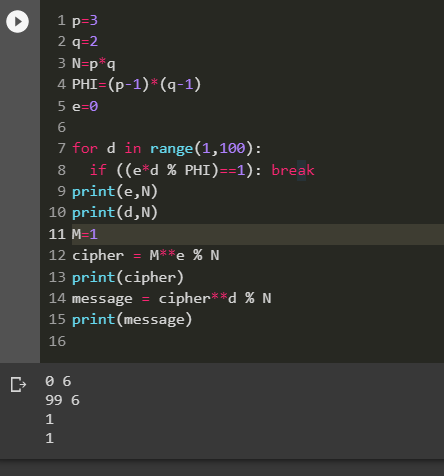
**After decrypting the above ciphertext**

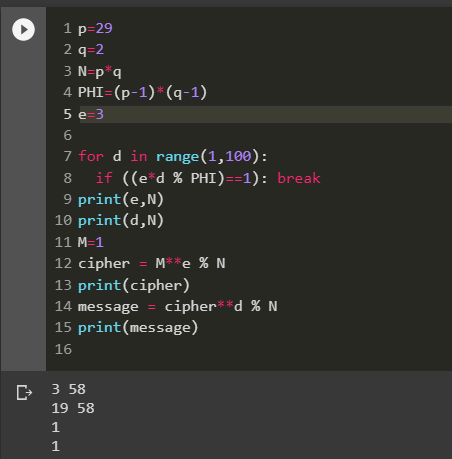
**5 = C^d (mod N) = 15^7 (mod 22)**

****

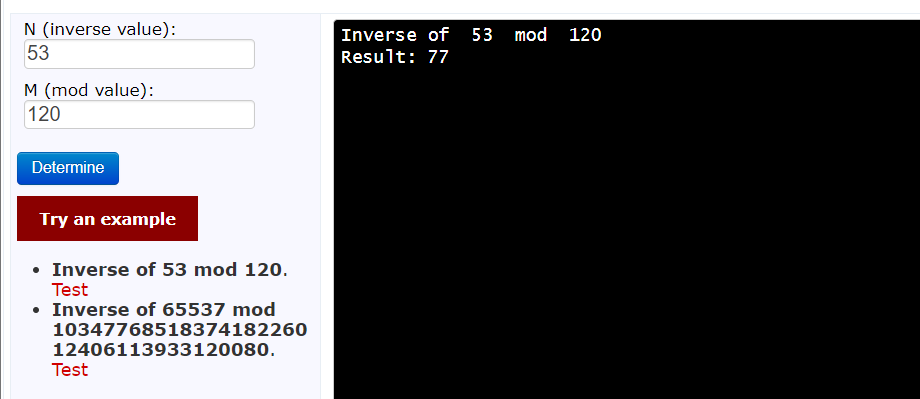
**3 more examples :**

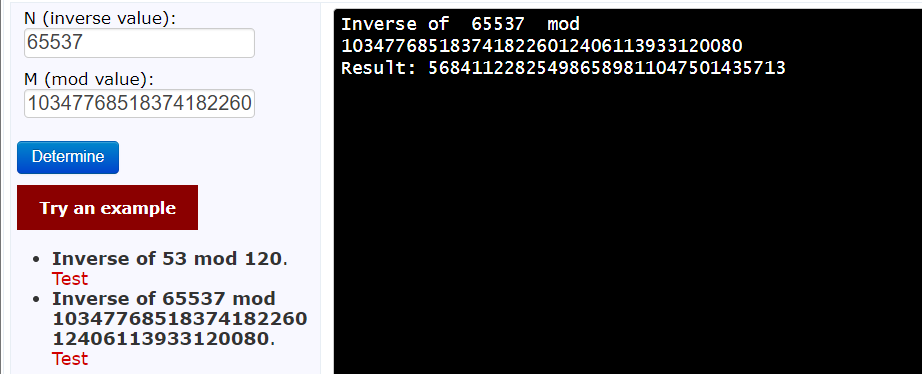
****

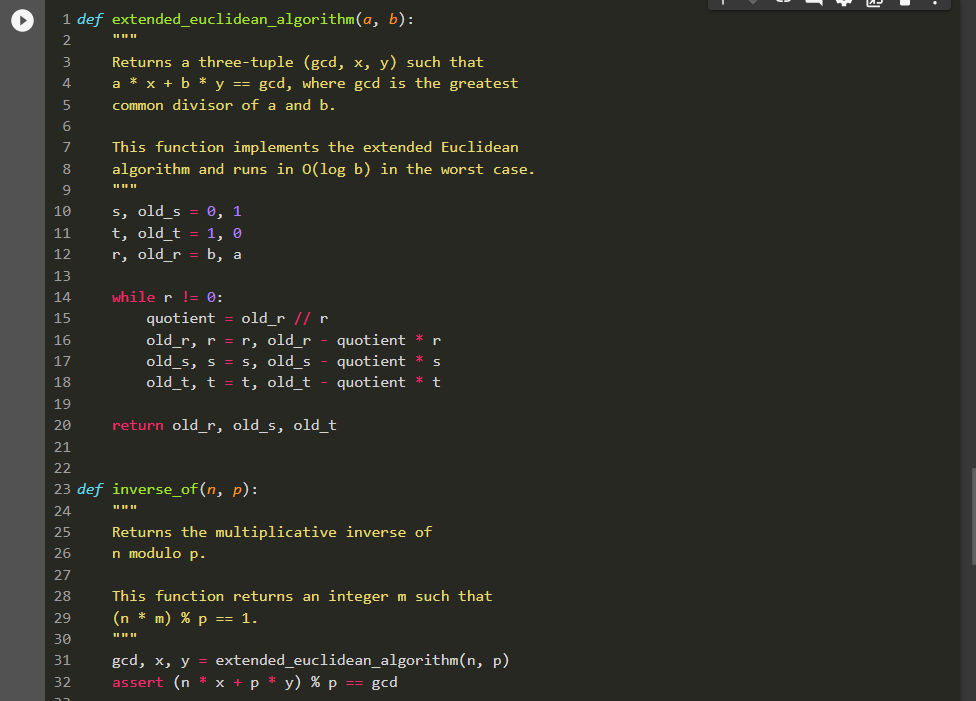
****

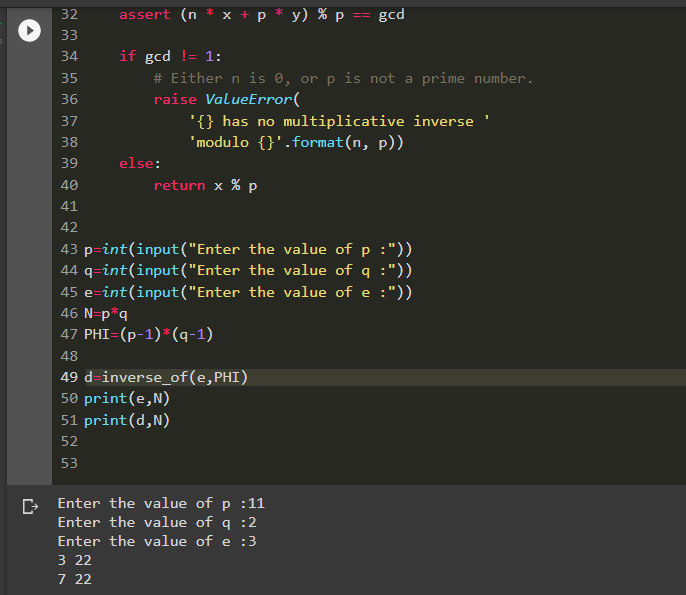
****

**E.2) OUTPUT :**

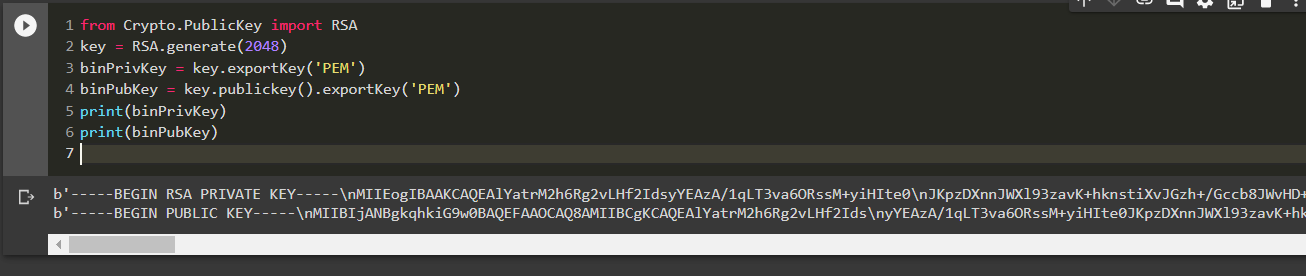
****

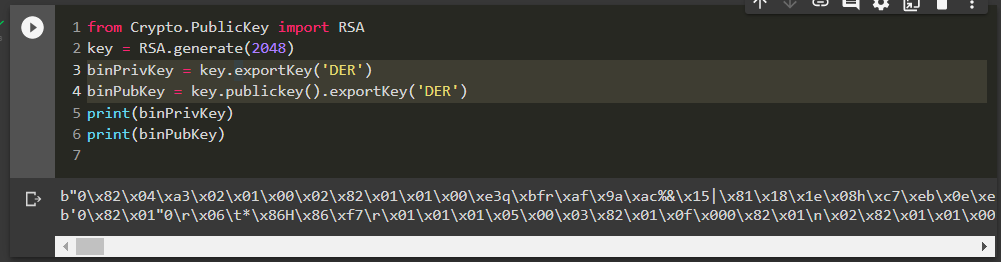
****

****

****

**E.3) OUTPUT :**

****

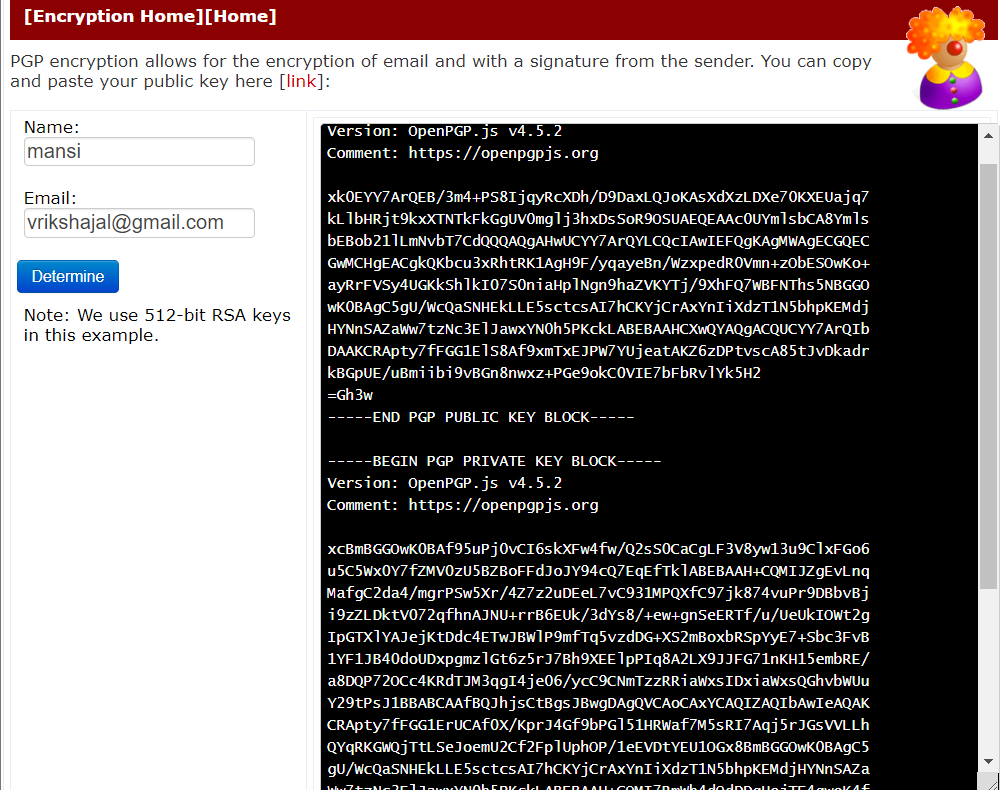
****

**DER is the method of encoding the data that makes up the certificate. DER itself could represent any kind of data, but usually it describes an encoded certificate or a CMS container whereas PEM is a method of encoding binary data as a string (ASCII armor). It contains a header and a footer line (specifying the type of data that is encoded and showing begin/end if the data is chained together) and the data in the middle is the base 64 data. PEM stands for Privacy Enhanced Mail; mail cannot contain un-encoded binary values such as DER directly.**

1. **PGP**

**F.1) OUTPUT :**

**F.2) OUTPUT :**

****

**F.3) OUTPUT :**

| **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?  **PGP generates a session key, which is a secret key and is generated only once. This key generates a random number from the movement of your cursor and the keystrokes you type. This session key is used to encrypt the plaintext with a very secure and fast symmetric encryption algorithm, and the output is ciphertext.**  Outline the contents of your key file:  **Both the files contain a header and footer signifying that they are either PGP Public or Private key blocks and the content between them is the actual key.** |
| **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 their publickey.key  Now list your keys with:  gpg --list-keys | Which keys are stored on your key ring and what details do they have:  **After creating another Dummy User and importing their key and then after listing the keys I found that the list consists of my personal key as well as the key of the user I just created. The other information displayed was their public key encryption algorithm (RSA) their uid that included the user’s name and email address and finally the expiry date of the pgp key.** |
| **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:  **Create ASCII armored output. The default is to create the binary OpenPGP format**  What does the –r option do:  **Encrypt for user id name. If this option or ‘--hidden-recipient’ is not specified, GnuPG asks for the user-id unless ‘--default-recipient’ is given.**  What does the –u option do:  **Use name as the key to sign with. Note that this option overrides ‘--default-key’.**  Which file does it produce and outline the format of its contents:  **It produces a .asc file (ascii armored file) wherein the header and footer denote that beginning and ending of the PGP message and between them is contained the actual encrypted message.** |
| **4** | Send your encrypted file in an email to your lab partner, and get one back from them.  Now create a file (such as myfile.asc) and decrypt the email using the public key received from them with: gpg –d myfile.asc > myfile.txt | Can you decrypt the message: |
| **5** |  |  |
| **6** |  |  |

1. **TrueCrypt**

| **No** | **Description** | **Result** |
| --- | --- | --- |
| **1** | Go to your Kali instance (User: root, Password:toor). Now Create a new volume and use an CPU (Mean) encrypted file container (use tc\_yourname) with a Standard TrueCrypt volume.  When you get to the Encryption Options, run the AES-Two-Seperate benchmark tests and outline the results: | CPU (Mean)  AES: **1.5 GB/s**  AES-Twofish: **149 MB/s**  AES-Two-Seperent  Serpent -AES : **88 MB/s**  Serpent: **93 MB/s**  Serpent-Twofish-AES:  **58 MB/s**  Twofish: **170 MB/s**  Twofish-Serpent: **59 MB/s**  Which is the fastest:  **AES**  Which is the slowest:  **AES-Twofish-Serpent** |
| **2** | Select AES and RIPEMD-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?  **The random pool basically keeps on capturing the mouse movements of the user continuously as it is also alerted on the screen that the user must move the mouse within the window as randomly as possible which helps in increasing the cryptographic strength of the encryption keys.** |
| **3** | Now mount the file as a drive. | Can you view the drive on the file viewer and from the console? **Yes** |
| **4** | Create some files on your TrueCrypt drive and save them. | **After creating some files and saving them I was able to perform any operations on them but once I demounted the volume and tried to access it only a file named ‘Encrypted Folder’ was shown (I had named it so during creation of the volume)**    **But once I tried to mount the volume again I was prompted to enter the password that I had set earlier only after entering the correct password was I able to access all my files saved in that volume.** |

1. **Reflective Statements**

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 large enough? If we use a cracker that performs 1 Tera keys per second, will someone be able to determine our private key?

**Secp256k1(utilizes 256 bit private key) was almost never used before Bitcoin became popular, but it is now gaining in popularity due to its several nice properties. Most commonly-used curves have a random structure, but secp256k1 was constructed in a special non-random way which allows for especially efficient computation. As a result, it is often more than 30% faster than other curves if the implementation is sufficiently optimized. Also, unlike the popular NIST curves, secp256k1's constants were selected in a predictable way, which significantly reduces the possibility that the curve's creator inserted any sort of backdoor into the curve.**

**1 TB = 8e+12 keys so if the cracker checks these many keys in one second and if the private key size possibilities become 2^256 which is approximately 1.2e+77 keys so cracking the 256 bit private key is not a tough task, the key can be cracked within 8 seconds.**