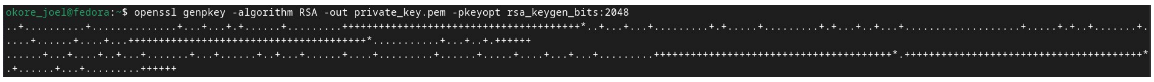
SSN-3rd-Lab-JOEL_OKORE

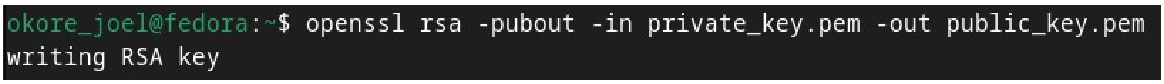
1. Create a 2048-bit RSA key pair using OpenSSL. Write your full name in a text file and encrypt it with your private key. Using OpenSSL extract the public modulus and the exponent from the public key. Publish your public key and the base64 formatted encrypted text in your report. Verify that it is enough for decryption.

   To perform this task, I made use of the openssl linux package to create a 2048-bit RSA key pair. To generate the private and public key pairs, I used the ‘genpkey’ command of the openssl utility, specifying RSA (with a 2048-bit key) as the algorithm of choice by setting the ‘-algorithm’ option to RSA and ‘-pkeyopt’ to ‘rsa_keygen_bits:2048’, and writing the resulting private key into a Privacy-Enhanced Mail file (private_key.pem) as shown in pic. 1.



Picture 1 - Generating RSA key pairs of length 2048 bits

I then went on to extract the public key (public_key.pem) from the just gotten private key (private_key.pem) as shown in pic. 2.



Picture 2 - Extracting the public key

I also created a file (name.txt) containing my full name using the ‘echo’ command as illustrated below (pic. 3)

```
okore_joel@fedora:~$ echo "Joel Chidike Okore" > name.txt
okore_joel@fedora:~$
```

Picture 3 - Writing my full name to name.txt

In order to encrypted the name.txt file, I used the 'pkeyutl' command of the openssl utility, setting the 'inkey' flag to the the private key (private_key.pem) and outputting the result as a binary file (name_encrypted.bin). See picture 4.

```
okore_joel@fedora:~$ openssl pkeyutl -sign -in name.txt -inkey private_key.pem -out name_encrypted.bin
okore_joel@fedora:~$
```

Picture 4 - Encrypting name.txt

The picture below (pic. 5) is the extracted representation of the modulus and exponent of the public key.

```
okore_joel@fedora:~$ openssl rsa -in public_key.pem -pubin -text -noout
Public-Key: (2048 bit)
Modulus:
    00:a9:d7:89:b6:72:13:44:b5:d4:ee:bf:f3:1b:2f:
    72:85:07:39:ad:39:d0:61:21:73:22:f9:ee:fd:e1:
    0e:01:0f:a8:33:89:eb:41:30:13:46:f8:16:96:57:
    49:2d:cd:df:51:5f:89:95:3a:ff:4d:f1:e0:9b:df:
    3d:a1:b9:a7:da:b2:1d:02:d7:19:d6:37:1d:84:73:
    32:c4:1b:d3:f7:c0:a4:e4:26:ce:c6:aa:ed:74:a2:
    73:f6:e9:49:4f:ea:c5:43:89:93:9f:03:2b:8b:67:
    fd:13:f1:03:4a:e9:ae:ec:e9:a7:be:57:94:d2:cd:
    ba:fc:c8:6a:4e:e6:0e:96:ce:f5:ab:c7:f0:93:73:
    82:f4:eb:71:af:aa:c0:5b:a2:79:5c:98:38:e4:47:
    50:2b:64:70:be:73:9c:f1:35:bf:a7:3e:b1:49:a7:
    0b:6f:52:3f:b3:f7:b4:23:a4:b6:c1:bb:8d:74:4e:
    2b:88:8c:5a:dd:fa:ae:ca:03:3e:2a:e6:53:72:6d:
    b4:5a:ba:f3:76:84:0c:b9:4d:32:00:b9:b1:a4:e6:
    94:4e:87:a5:14:38:58:f7:b4:d9:2b:7f:08:36:ce:
    a3:e8:43:88:98:52:81:4e:57:c9:a4:7b:05:ce:b5:
    a0:90:06:8c:90:f5:e7:5d:e0:e3:9f:0c:06:bc:7f:
    f0:79
Exponent: 65537 (0x10001)
```

Picture 5 - Modulus and Exponent of public key

To obtain the base64 format of the encrypted text, I made use of the 'base64' command of the openssl utility and outputted to a readable text file as illustrated in pic. 6.

```
okore_joel@fedora:~$ openssl base64 -in name_encrypted.bin -out name_encrypted_base64.txt
okore_joel@fedora:~$ ls
Desktop    Downloads  Music                      name_encrypted.bin  Pictures         Public          pyew_env  Templates
Documents  get-pip.py name_encrypted_base64.txt  name.txt            private_key.pem  public_key.pem  SSN_1     Videos
```

Picture 6 - Getting base64 format of encrypted text (name_encrypted.bin)

Below is the encrypted text in base64 readable format (pic. 7)

```
okore_joel@fedora:~$ cat name_encrypted_base64.txt
J24BGjFWJx+yGt6TztoRga9i/nT3COF9wm6K5Ey95cRe4Q5sEhHEmu0d639w/EY5
gVtqkGMt79G8CNp8tbCEelNgdaG5X3et+HhjMMROxLgV40WEUOA5cljumvpL/3O0
1rx/pT52zwMwzF5y2RZrDPECsj7MtPgXdehFOdDGD7wMuZtes9qQSDPbqBLzT7qq
5l5r0HC/2Og9HaWA4Gcu19VjKKVwTeongK80Q1zlzgxVO3HcBDbOwWGy2DuhEfaU
wmWZyHCmfnCv9PYiLZ3+YLe2VMYGLWYMVKUu4UwQxISc7YipA7FJ/YMxKFdWLgYI
QXYLgKVwrATy4Cvwl5JBww==
```

Picture 7 -Encrypted text in base64 format

To sign and verify an already encrypted file using my RSA keys, I needed to first apply a cryptographic hash to the encrypted file and then sign the hash with my private key.
Verifying the signature involves using the corresponding public key to confirm the integrity of the encrypted file.

To sign the encrypted file (name_encrypted.bin), I used the 'dgst' (digest) command of the openssl package, with 'sha256' as the hash function of choice, and outputting the result to a binary file (name_encrypted_signature.bin). See picture 8.

```
okore_joel@fedora:~$ openssl dgst -sha256 -sign private_key.pem -out name_encrypted_signature.bin name_encrypted.bin
okore_joel@fedora:~$ cat name_encrypted_signature.bin
okore_joel@fedora:~$ openssl base64 -in name_encrypted_signature.bin -out name_encrypted_signature_base64.txt
okore_joel@fedora:~$ cat name_encrypted_signature_base64.txt
```

```
CE/oSeC32tiLgPBBElc8xWJO7RqEinawIPF6SNCBxrIE/pRCbaZu2aG/ni+gbVVA
fpqgJuwTxu3SWl2NJuCkRTlZH4jH6vgyq3lcXBK3Kgw0GyoQmPubkinKxHQozI+G
vfLFIieiSlsrFrkJtO+pDR430Fq5VnNaz9Uo9XhQTGbSOAdSjwTll3TsBjEmaarg
kvuhF81yuStvMS4j8yJISZNx5a+rVFQPGXW+XN35xLqJ6Rgqpz3HTUfDqRrohkXK
UczVdT6uicFOf+Oeee5CLUFPfPpybV/L5EDGBkALfYD+eqMnZeAvW7uOLnPfNAjV
/u5YlXtLuQ9ejoCpDqgBgA==
```


Picture 8 - Signing encrypted file

FInally, to confirm the integrity of the encrypted file, I verified the signature (name_encrypted_signature.bin) using my public key as shown in picture 9.

```
okore_joel@fedora:~$ openssl dgst -sha256 -verify public_key.pem -signature name_encrypted_signature.bin name_encrypted.bin
Verified OK
okore_joel@fedora:~$
```

Picture 8 - Verification Successful

2. Assuming that you are generating a 1024-bit RSA key and the prime factors have a 512-bit length, what is the probability of picking the same prime factor twice? Explain your answer

To calculate the probability of picking the same 512-bit prime factor twice when trying to generate a 1024-bit RSA key, we first need to know how many 512-bit primes they are.

The prime number theorem states that the number π(x) of primes ≤ x is approximately x/ ln x. Accepting this approximation without further analysis, it can be concluded that the number of 512-bit primes is about:

π(2^512) - π(2^511) ≈ (2^512 / ln (2^512)) - (2^511 / ln (2^511)) ≈ 18.85 * 10^150

But the prime number theorem actually says that lim(x→∞) π(x)/(x/ ln x) = 1. Which simply means that the percentage error in the approximation π(x) ≈ x/ ln x (that is the percentage error if we decide to follow the last calculation) goes to zero as x goes to ∞ (infinity).

J. B. Rosser and L. Schoenfeld in their 1962 journal [1] proved the inequality *$0.6x/ \ln x < \pi(2x) - \pi(x) < 1.4x/ \ln x$*, is true for all $x \geq 20.5$

If x = 2^(m-1), then

π(2^m) - π(2^(m-1)) > 0.3 ∗ 2^m / ((m-1) * ln 2) ≈ 0.43 ∗ 2^m / (m-1), for m ≥ 6.

This proves that the number of 512-bit primes is

π(2^512) − π(2^511) > 0.3 ∗ 2^512 / (511 ln 2) ≈ 11.36 × 10^150,

Which is smaller than the first approximation of 18.85 ∗ 10^150.

The likelihood of selecting the same prime twice is approximately equal to randomly drawing two identical numbers from a pool of 11.36 × 10^150 numbers. This is because RSA generation techniques usually select the two prime factors, p and q, independently from this vast pool of primes.

The probability of this happening is:

P (same prime) ≈ 1 / N where N ≈ 11.36 × 10^150.

Therefore the probability of picking the same prime twice is approximately :

P (same prime) ≈ 1 / (11.36 × 10^150)

In practicality, this is zero, a very small number. Since 11.36 × 10^150 is an almost incomprehensibly great number, there is very little probability that the same prime will be chosen twice.

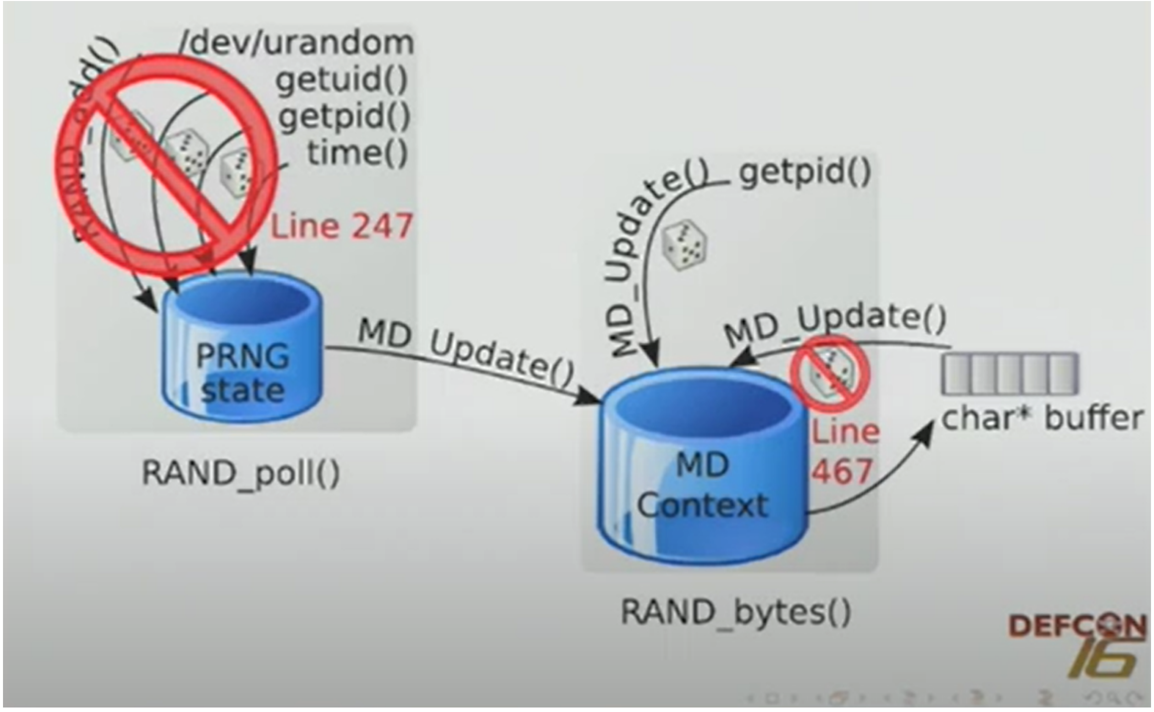
3. Explain why using a good RNG is crucial for the security of RSA. Provide one
   reference to a real-world case where a poor RNG leads to a security vulnerability.

Random number generation (RNG) is very important to RSA cryptography, especially for generating the prime integers p and q, which are then used to find the modulus n=p * q. These primes are selected in a random and nonrepetitive manner only if RNG is considered secure. The difficulty in factoring the modulus n back into its prime factors is the foundation of RSA's security. Security issues such as key collision and decreased entropy arises if the primes are created using an unreliable RNG.

A very popular example of a security vulnerability caused by a poor RNG is the Debian OpenSSL vulnerability with Common Vulnerability and Exposure number of CVE-2008-0166 from the official MITRE National Vulnerability Database.

This happened because an OpenSSL developer commented out the code that seeded OpenSSL's pseudo-random number generator (PRNG) in the Debian version of the OpenSSL library (persisted from version 0.9.8c-1 up to versions before 0.9.8g-9). As a result, the generated random numbers had less entropy, which made the RNG nearly predictable. Because of this, the OpenSSL library produced a very small range of potential keys for Debian-based computers, like Ubuntu. [3]

Commenting out of those sections of code ultimately affected a function within the OpenSSL library source code (RAND_poll()) such that some sources of entropy such as linux own PRNG (/dev/urandom), current user ID (getuid()), process ID (getpid()) and current time (time()), all gotten from the host computer were not read into the entropy pool, as such, they didn't affect the PRNG (Pseudo-Random Number Generator) state, which limited the maximum possible entropy space to 15 bits (the length of the current process ID, the only remaining parameter that introduced some sought of entropy). See picture 9.

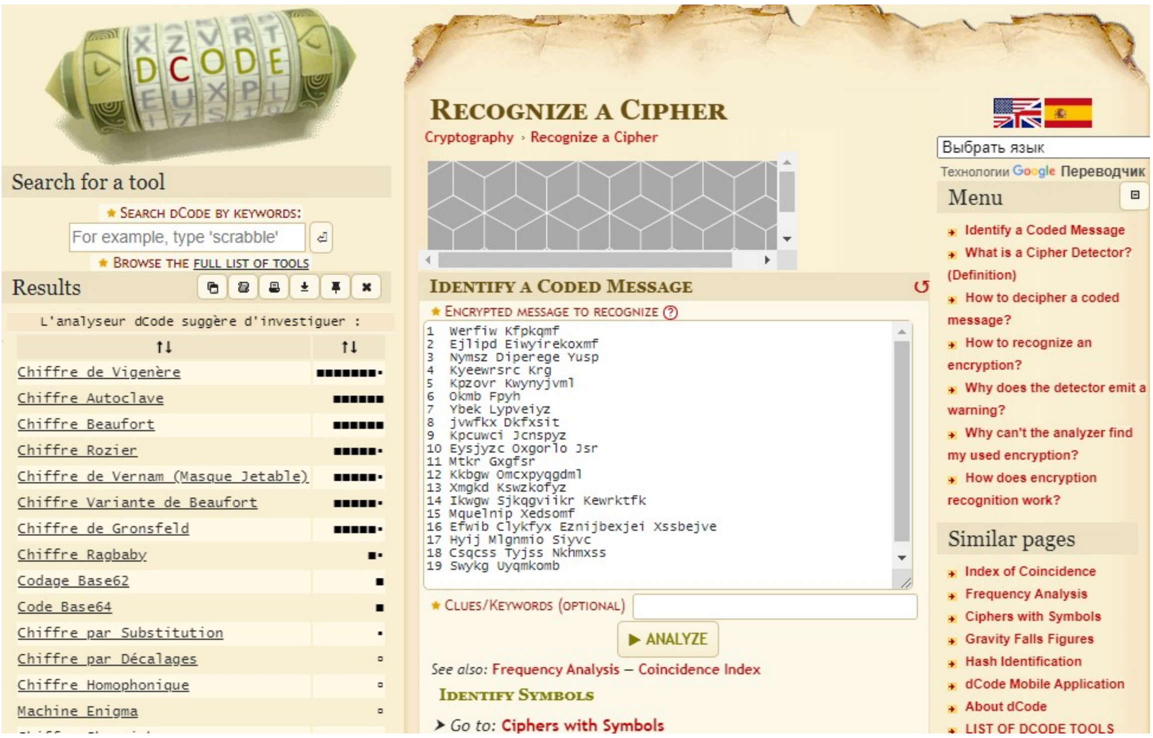


Picture 9 - Visual representation of OpenSSL PRNG showing discarded parameters within the entropy pool

This meant that RSA keys generated on Debian-based systems during this period were highly predictable. It also affected every utility that made use of the flawed OpenSSL package, for example SSH keys, SSL/TLS certificates, VPN keys e.t.c. [4]

4. Here you can find the modulus (public information) of two related 1024-bit RSA
   keys. Your keys are numbered using the list. Your task is to factor them i.e. retrieve p
   and q. You may use any tools for this. Explain your approach.

Lacking knowledge about the cipher used to encrypt the numbered list of keys, I employed the cipher recognition tool on dcode.fr. After analyzing the text, the tool suggested several likely ciphers used for encryption. See picture 10 for the results.

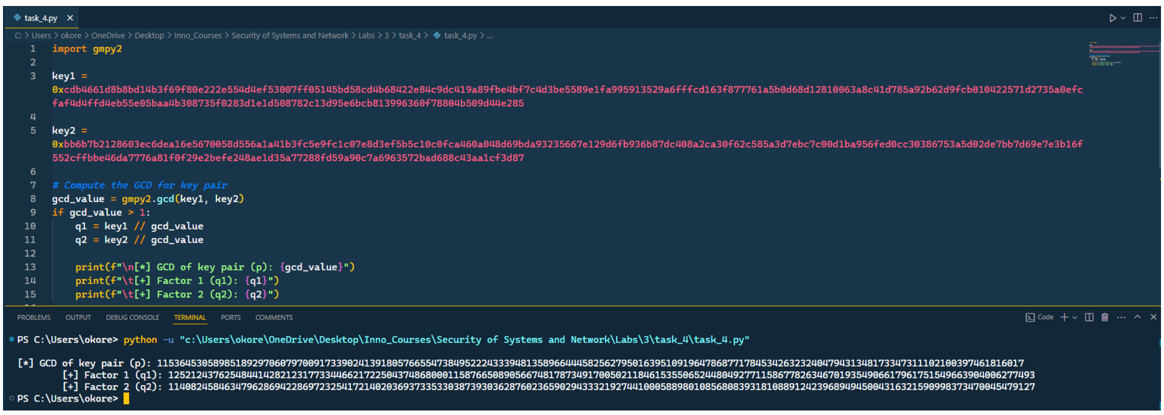


Picture 10 - Using dcode cipher recognition tool on enciphered text

With Vigenère Cipher identified as the most probable encryption method, I proceeded to attempt automatic decryption using the Vigenère cipher tool available on the website, as shown in pic. 11.

Picture 11 - Deciphered text

As my name was 17th on the list, I wrote a Python script to factor p and q from the moduli of the two related keys (number 17). I utilized the gmpy2 Python library to calculate the Greatest Common Divisor (GCD), as illustrated in picture 12.



Picture 11 - Obtaining key pair factors using gmpy2 python library

💡 [*] GCD of key pair (p):

1153645305898518929706079700917339024139180576655
4738495222433394813589664445825627950163951091964
7868771784534263232404794313481733473111021003974
61816017

💡 [+] Factor 1 (q1):

1252124376254844142821231773344662172250437486800
0115876650890566748178734917005021184615355065244
8049277115867782634670193549066179617515496639040
06277493

💡 [+] Factor 2 (q2):

1140824584634796286942286972325417214020369373353
3038739303628760236590294333219274410005889801085
6808393181088912423968949450043163215909983734700
45479127

5. Now that you have the p and q for both keys, recreate the first public and private
key using this script. Encrypt your name with the private key and post the public key
and the base64 formatted encrypted data in your report.

The script below contains the first modulus along with its corresponding factors, p and q. Its purpose is to recreate the first public and private key pair, which will then be used to encrypt data (in this case, my name). See picture 12 for the script.

```
okore_joel@fedora:~/Desktop$ cat rsakey.py
__author__ = 'cdumitru'

from Crypto.PublicKey import RSA
from Crypto import Random
import gmpy2
import base64


#tup (tuple) - A tuple of long integers, with at least 2 and no more than 6 items. The items come in the following order:
#RSA modulus (n).
#Public exponent (e).
#Private exponent (d)
#First factor of n (p).
#Second factor of n (q)

n = 0xcdb4661d8b8bd14b3f69f80e222e554d4ef53007ff05145bd58cd4b68422e84c9dc419a89fbe4bf7c4d3be5589e1fa995913529a6fffcd163f877761a5b0d68d12810063a8c41d785a92b62d9fcb010422571d2735a0efcfaf4d4f
fd4eb55e05baa4b308735f0283d1e1d508782c13d95e6bcb813996360f78804b509d44e285L
p = 11536453058985189297060797009173390241391805766554738495222433394813589664445825627950163951091964786877178453426323240479431348173347311102100397461816017
q = 12521243762548441428212317733446621722504374868000115876650890566748178734917005021184615355065244804927711586778263467019354906617961751549663904006277493
e = 65537L

string_to_be_encrypted="Okore Joel Chidike"

phi = (p - 1) * (q - 1)
d = long(gmpy2.invert(e, phi))

tup = (n ,e, d, p, q )
key = RSA.construct(tup)

random_generator = Random.new().read
data = key.encrypt(string_to_be_encrypted,random_generator)


print "PRIVATE KEY: \n", key.exportKey('PEM')
print
print "PUBLIC KEY: \n", key.publickey().exportKey()
print
print "Encrypted Data in base64: \n", base64.b64encode(data[0])
```

Picture 12 - Encrypting my name with recreated key pairs

Below is the output after running the script. It contains my encrypted name, the private and public key pair, and the base64 format of the encrypted data (Pic. 13).

```
okore_joel@fedora:~/Desktop$ python2 rsakey.py
PRIVATE KEY:
-----BEGIN RSA PRIVATE KEY-----
MIICWwIBAAKBgQDNtGYdi4vRSz9p+A4iLlVNTvUwB/8FFFvVjNS2hCLoTJ3EGaif
vkv3xNO+VYnh+plZE1Kab//NFj+Hd2GlsNaNEoEAY6jEHXhakrYtn8sBBCJXHSc1
oO/Pr01P/U61XgW6pLMIc18Cg9Hh1Qh4LBPZXmvLgTmWNg94gEtQnUTihQIDAQAB
AoGAQZs1rlqhX0Emmn+Y0mIApsV2AbmrJk2V9IgUF5oRIBG/h5m/ZCNnS0ClX+Ec
NsycOAKnp8XsydY8sNHsfNHnu0gY+INthitV4z7taf5Dh9lc2hWuxkVkCBwJcRZ7
awrL5QTy4lP7EmndKVuK86fj4aKYpDLeYMF3i+wiywscrYECQQDcRQIHpI/OFrb5
Zm4Wuxr5iXNciCF0Caboi4lhAk7BWGzeJaT6TYPF3orSKPbkTScPvTulwuwFbtix
ujNMf0LRAkEA7xKQitOPyhPVwVQlZAGwoXq+JUlSSxkXT3hyngHFA558htdfhMKK
wfkU0P82MSw18GVvEGpMnrSYacjZ7iIJdQJAWwd5y0bK11Sz0WXvYR16DF4terQX
fyjt/XSNFbYqbeTpOPU9tOOL7Z4GqMudHMR1vB9sIxjnCWfBzdExmg+NwQJAGW/T
O0IpUWns22YVmF9pqLTDmHoFrIHd7hG1uEQd7zWksEAoJJsCYoMbCOWuoWq/znUs
bQWyY0x5zv+U2HZ8pQJAeakB25elZoL/R2sztS5gNDaV8/sDP5cJ1eug/4qPgZr0
E964Qz11uJXvw59ZwLYOJh4V1+Njf48CL3MaDt1yBA==
-----END RSA PRIVATE KEY-----

PUBLIC KEY:
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDNtGYdi4vRSz9p+A4iLlVNTvUw
B/8FFFvVjNS2hCLoTJ3EGaifvkv3xNO+VYnh+plZE1Kab//NFj+Hd2GlsNaNEoEA
Y6jEHXhakrYtn8sBBCJXHSc1oO/Pr01P/U61XgW6pLMIc18Cg9Hh1Qh4LBPZXmvL
gTmWNg94gEtQnUTihQIDAQAB
-----END PUBLIC KEY-----

Encrypted Data in base64:
ikPVU3+ZpY7CpfMbdznZ63HGdPxKCdbYpkFaY/xu7zo6s3Ctjp0ZUALHEJ9eo4hd4UFBkFMFBnTKs29EJ6eHXHpy/iYUeVeDPfFhWl2WtPLPvx8HL0lmQbLMIPB1RuK9dqNDJZFHFX4DE7Acci/GVD9i+iJwUXZ5OvViQ95pijM=
okore_joel@fedora:~/Desktop$
```

Picture 13 - Output after running the script

REFERENCES

1. J. B. Rosser and L. Schoenfeld. “Approximate formulas for some functions of prime numbers,” Ill. J. Math., vol. 6, pp. 64–94, 1962
2. Abstract: Is there a shortage of primes in cryptography? by Samuel S. Wagstaff, Jr. -https://homes.cerias.purdue.edu/~ssw/shortage.pdf
3. MITRE CVE: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166
4. DEFCON 16: https://www.youtube.com/watch?v=yXr7KBC3G3I