Innopolis University

SYSTEM AND NETWORKING ENGINEERING

Security of Systems and Networks

LABORATORY REPORT 5

Asymmetric Cryptography

Student	**ID**
Grebennikov Sergey	47611

Lecturer
Dr. Rasheed Hussain, PhD

November 9, 2017

Contents

Introduction

RSA was first described in 1977 by Ron Rivest, Adi Shamir and Leonard Adleman of the Massachusetts Institute of Technology. Public-key cryptography, also known as asymmetric cryptography, uses two different but mathematically linked keys, one public and one private. Many protocols like SSH, OpenPGP, S/MIME, and SSL/TLS rely on RSA for encryption and digital signature functions. It is also used in software programs - browsers are an obvious example, which need to establish a secure connection over an insecure network like the Internet or validate a digital signature. RSA signature verification is one of the most commonly performed operations in IT.

1 RSA

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 encrypted text in your report.
   Answer:

```
$ openssl genpkey -algorithm RSA -out private_key.pem -pkeyopt rsa_keygen_bits
    :2048
.....................................................+++
.....................................+++

$ chmod go-r private_key.pem

$ openssl rsa -pubout -in private_key.pem -out public_key.pem
writing RSA key

$ echo "Grebennikov Sergey" > name.txt

$ openssl rsautl -encrypt -inkey private_key.pem -in name.txt -out enc_name

$ openssl rsa -pubin -text -noout -in public_key.pem
Public-Key: (2048 bit)
Modulus:
    00:db:4f:67:dc:1d:fa:73:33:88:a3:bd:21:fa:74:
    74:ae:42:bc:ef:ee:94:29:dd:cb:41:e7:43:bf:ba:
    d8:e4:66:fe:fe:b2:fa:02:81:9a:25:e7:27:fb:81:
    8f:d8:7f:cf:e1:f0:04:04:1f:f1:0f:55:31:5c:ec:
    17:6e:da:80:4b:18:0f:45:5f:60:1b:a1:47:aa:91:
    31:ff:75:3f:12:77:e6:ab:3d:34:55:df:07:ed:06:
    3a:65:60:61:46:97:79:0f:70:b8:fc:02:76:74:05:
    6a:89:cf:6a:2a:1e:a4:ec:83:ed:3b:df:ae:0c:19:
    4a:ec:82:ff:c3:71:7b:04:e6:db:f9:0c:bc:74:21:
    08:3b:88:34:41:a3:ee:33:70:e4:fb:9a:4b:20:9c:
    6c:d3:05:73:9f:c9:e0:a6:69:06:a7:41:f0:65:ea:
    d0:35:ce:08:ff:8b:a3:fd:15:0f:a5:ed:18:88:8e:
    3a:e1:5d:0e:c5:2b:92:0a:e7:e2:84:af:8f:24:b6:
    46:e9:a0:64:df:22:c4:4b:dd:67:d0:00:be:77:4d:
    5a:fe:3e:64:68:cc:e3:50:c9:62:74:8e:ec:25:4c:
    17:b1:31:5d:c0:80:58:c1:db:de:9c:05:b9:7c:06:
    23:e6:f1:95:5a:d3:d6:b0:30:1a:f1:a7:8b:52:21:
    e6:25
Exponent: 65537 (0x10001)

$ openssl rsa -pubin -in public_key.pem
writing RSA key
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA209n3B36czOIo70h+nR0
rkK87+6UKd3LQedDv7rY5Gb+/rL6AoGaJecn+4GP2H/P4fAEBB/xD1UxXOwXbtqA
SxgPRV9gG6FHqpEx/3U/Enfmqz00Vd8H7QY6ZWBhRpd5D3C4/AJ2dAVqic9qKh6k
7IPtO9+uDBlK7IL/w3F7BObb+Qy8dCEIO4g0QaPuM3Dk+5pLIJxs0wVzn8ngpmkG
p0HwZerQNc4I/4uj/RUPpe0YiI464V0OxSuSCufihK+PJLZG6aBk3yLES91n0AC+
d01a/j5kaMzjUMlidI7sJUwXsTFdwIBYwdvenAW5fAYj5vGVWtPWsDAa8aeLUiHm
JQIDAQAB
-----END PUBLIC KEY-----
```

```
$ base64 enc_name
gdLnkBS+9KMNzFHQKOK4cm7Dvmb4qU1/pkOYUHbKic2va3GfiZoukSVRWUST68TkL4U06i8QcOMA
lKu43mO26q4HdOD+fWLqkHHCh7TS94UvLHAtSsrTRErYOClVFTw6uvU/AdHilVhbP1RwRq0T9CSL
rQlWU+OrEbBwfuB2GXua6bAaTAo2vPL88fFM2PizHN5iz3wBGI8Ya3LB53wEAi39XoswN1BrnNDe
HFIhQUTFrCPnzC06zpvHXoAVgPQ8cVVa+eL5OaIHaEOdcf/R9xU396npm4u57dPnX3auQ7NsUKEn
UpdReMI91P9SOch+bcGRL4KVhXlU5eX4DrQy7g==
```

2. Assuming that you are generating a 1024 bit RSA key and the prime factors have a 512bit length, what is the probability of picking the same prime factor twice? Explain your answer. *Hint: How many primes with length 512bit or less exist*?
Answer:
The prime number theorem states that there are approximately $N = x/\ln(x)$ prime numbers that are less than or equal to x. Here $\ln(x)$ denotes the natural logarithm of x, or in other words the logarithm with base of the number e.

$$x = 2^{512}$$

$$N = \frac{2^{512}}{\ln(2^{512})} \approx 3.778003518 \times 10^{151}$$

Therefore the probability of chosing the same prime factor is $P = ln(x)/x$.

$$P = \frac{\ln(2^{512})}{2^{512}} \approx 2.646900658937254 \times 10^{-152} \approx 2.6 \times 10^{-150}\%$$

3. Explain why using a good RNG is crucial for the security of RSA. Provide one reference to a realworld case where a poor RNG lead to a security vulnerability.
Answer:
If a random number generator can be made predictable, it can be used as backdoor by an attacker to break the encryption since much cryptography depends on a cryptographically secure random number generator for key. In May 2008, security researcher Luciano Bello discovered that the random number generator in Debian's openssl package is predictable. As a result, cryptographic key material may be guessable [3].

4. Here you can find the modulus (public information) of two related 1024bit RSA keys. Your keys are numbered using the list at `http://188.130.155.61/ssn/Instructions.txt`. Your task is to factor them i.e. retrieve p an q. You may use any tools for this. Explain your approach. *Hints: study the RSA algorithm. What private information can two keys share? What practical attacks exist? You may have to write code or use existing code for simple arithmetic operations.*
Answer:
My number: 22
Modulus:
0xbf9ba67048f5cba1106654a615b99c3d8ce11fce98052253c0605c4f256fdc31ad857c3b65c8ceae36ac4058231b9d
a366fa7457de4a10b6aa379158bb08b75b8d15b8681f2cfcc873ad791bca5f641212ab5801d1b8bb28f4b26a4f5ab7c
25f77d2580e042de28c2a66ef3ceed759b790c55ff8dcdae7ab2df8fb547308a1cfL ,

0xc7e608de36ee105131d0d3fbac284890942176bf8f5afe92306bf7738beabb0631e51274cec9145da0de0c1497ac2a
b0065f79fdc1638f34bd679a741d8e71d1b6eec81f1817d19f6f8899d2068ac2bc54b10bf94b91c02170f6d837d8641c
275569531ed5b56c68a0f88986c6d0f822b1e1f6fd3b3c082892d56e30393c77e5L

Since we have the modulus of two **related** 1024bit RSA keys, there are only three different primes p_1, q, and p_2. q has been re-used in two different keys, so the public values are $n_1 = p_1 \times q$ and $n_2 = q \times p_2$.
The security problem comes in if someone comes across both public keys and, looking at the public values n_1 and n_2, decides out of curiosity to calculate $gcd(n_1, n_2)$. This time, the result is not 1, but rather q, because both n_1 and n_2 are evenly divisible by q!
Noticing this leads quickly to cracking both keys, because now it's easy to calculate $p_1 = n_1/b$ and $p_2 = n_2/b$. That reveals both of the secret prime factors of both keys, which is enough to derive a complete private key for each and start decrypting encrypted messages.
The "shared prime" case gives $gcd(qp_1, qp_2) = q$, which leads to discovering both factors, revealing the private keys from the public keys. This means that using a prime in one's RSA key that someone else has already used in their RSA key is a very bad security failing.

Python program to factor modulus:

```
from fractions import gcd

with open("key1") as file1:
  n1 = int(file1.read(), 16)
  print("n1 = " + str(n1))

with open("key2") as file2:
  n2 = int(file2.read(), 16)
  print("n2 = " + str(n2))

q = gcd(n1, n2)
print('q = ' + str(q))
p1 = int(n1 / q)
p2 = int(n2 / q)
print('p1 = {}'.format(p1))
print('p2 = {}'.format(p2))
```

Result:

```
$ python3 crack_rsa.py
n1 = 13455171925326662448654175501689205572707092307463482700804459244176304169260691063734195285542616108589045298329734706042138028723988051324111757106813901590573230379602902546798407783222496265450478428886300615868565627410267403107346844542368267888790978934830187681381531601494618665587272299333469555020312
n2 = 14037355156988239294979694689329393942204821650202323014858235213584549166021597557207994924916398537325738586432332142916426447193217048058476593110397235580276530982585773033952375630857498052248698024107099926927461347456521611645598786650028494519226884921207902302865076550020683712605230558653470398666 1
q = 10480866909519440060818156048596189878528116225795638335970437426135159521684277194026594601663387721209885850786714712280770756073207468656629456841605347
p1 = 12837842557761853816472694765061805664727518299668417720715597310267801689754473997000098943997093299503021770684291376759041936713197182324397747029691173
p2 = 13393314959699138434367662973437535262922692176145410241752921813425887167407060442127470552160313592480069946511662319165448204790195573333756457296134631
```

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 encrypted data in your report.
Answer:

Modifying the script:

```
from Crypto.PublicKey import RSA
import gmpy
n = 13455171925326662448654175501689205572707092307463482700804459244176304169260691063734195285542616108589045298329734706042138028723988051324111757106813901590573230379602902546798407783222496265450478428886300615868565627410267403107346844542368267888790978934830187681381531601494618665587272299333469555020312L
p = 12837842557761853816472694765061805664727518299668417720715597310267801689754473997000098943997093299503021770684291376759041936713197182324397747029691173L
q = 10480866909519440060818156048596189878528116225795638335970437426135159521684277194026594601663387721209885850786714712280770756073207468656629456841605347L
e = 65537L
phi = (p - 1) * (q - 1)
d = long(gmpy.invert(e, phi))
tup = (n ,e, d, p, q )
key = RSA.construct(tup)
with open('private_key','w') as file1:
  file1.write(key.exportKey('PEM') + '\n')
with open('public_key', 'w') as file2:
  file2.write(key.publickey().exportKey() + '\n')
```

Public key:

```
$ cat public_key
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC/m6ZwSPXLoRBmVKYVuZw9jOEf
zpgFIlPAYFxPJW/cMa2FfDtlyM6uNqxAWCMbnaNm+nRX3koQtqo3kVi7CLdbjRW4
aB8s/MhzrXkbyl9kEhKrWAHRuLso9LJqT1q3wl930lgOBC3ijCpm7zzu11m3kMVf
+Nza56st+PtUcwihzwIDAQAB
-----END PUBLIC KEY-----
```

Encrypting Message:

```
$ openssl rsautl -encrypt -inkey private_key -in name -out enc_name

$ base64 enc_name
rvdTA1Iq1d5v2jQIsaLcgkjmvFzshBpjgC42cDmUHmsZnbb82AWHhTvPYqTYVqoBimqvW80qBQk1
m8kG57nY8qn2RYiQya2FVETsk0hDR9pKXlIx0p7Jm1XZOldWrdwkStX3h6zBzY0jHlOdC8ubBoso
ytm30t420/wQZPBgWhc=
```

2 Conclusion

Asymmetric cryptography is a branch of cryptography where a secret key can be divided into two parts, a public key and a private key. The public key can be given to anyone, trusted or not, while the private key must be kept secret (just like the key in symmetric cryptography).

Asymmetric cryptography has two primary use cases: authentication and confidentiality. Using asymmetric cryptography, messages can be signed with a private key, and then anyone with the public key is able to verify that the message was created by someone possessing the corresponding private key. This can be combined with a proof of identity system to know what entity (person or group) actually owns that private key, providing authentication.

Encryption with asymmetric cryptography works in a slightly different way from symmetric encryption. Someone with the public key is able to encrypt a message, providing confidentiality, and then only the person in possession of the private key is able to decrypt it.

References

[1] M. Stamp, Information Security: Principles and Practice, Second Edition, 2011, 606 pages.

[2] Public-key cryptography https://en.wikipedia.org/wiki/Public-key_cryptography.

[3] Debian Security Advisory. DSA-1571-1 openssl – predictable random number generator https://www.debian.org/security/2008/dsa-1571.

[4] Encrypt and decrypt files to public keys via the OpenSSL Command Line https://raymii.org/s/tutorials/Encrypt_and_decrypt_files_to_public_keys_via_the_OpenSSL_Command_Line.html.