Public Key Encryption II

IMPORTANT NOTES: Study lecture materials at least 1 hour and attempt Task 4 prior to the lab session. Prepared questions will be discussed in the lab session.

1 Overview

The objectives of this lab are to learn (i) the steps of RSA key generation and encryption/decryption; and (ii) how to use GPG to encrypt/decrypt files with asymmetric algorithms.

2 Lab Environment

SageMath: In this lab, we need to use the SageMath library to perform certain mathematical calculations. Use the SageMath web interface at `https://sagecell.sagemath.org/`.

GPG: GnuPG, also known as GPG, is already installed in our cloud VM. GPG is a complete and free implementation of the OpenPGP standard as defined by RFC4880 (also known as PGP). GnuPG allows to encrypt and sign your data and communication, features a versatile key management system as well as access modules for all kinds of public key directories. GPG is a command line tool with features for easy integration with other applications.

3 Lab Tasks

3.1 Public Key Encryption in GPG

In this task, every group will generate a key pair, share their public key with another lab group and encrypt/decrypt files.

3.1.1 Generating Key-pair

Head to the terminal and write:

```
$ gpg --full-generate-key
```

GPG will then prompts you through the process, giving options for each question.

- The type of key you want: RSA and RSA,
- the key length: 1024,
- the expiration time, default is "key does not expire",
- your real name: Your name,
- your email address: your@address.com
- a comment: "My student mail",
- and a passphrase used to protect you key.

If there is not enough randomness in the randomness pool you will be asked to do other things for instance open other applications, type randomly etc. Once enough randomness is collected, gpg will create the key pair.

```
fkdgpg: /home/muni/.gnupg/trustdb.gpg: trustdb created
gpg: key FF9E5B4F marked as ultimately trusted
public and secret key created and signed.

gpg: checking the trustdb
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg: depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 1u
gpg: next trustdb check due at 2019-04-09
pub   2048R/FF9E5B4F 2018-04-09 [expires: 2019-04-09]
      Key fingerprint = 70B8 3E10 E2D1 E453 4D6E  7C44 6E01 07AF FF9E 5B4F
uid                  Bob Dylan <bob.dylan@monash.edu>
sub   2048R/6C57FBA9 2018-04-09 [expires: 2019-04-09]
```

The keys are generated in a hidden directory .gnupg in your home folder. You can view it by navigating to your home folder and type: `ls -la`. Please notice the permission of file `.gnupg`.

3.1.2 Exchanging Keys

To communicate with others you must exchange public keys. To get hold on your public key you export it from your keyring using the command-line:

```
$ gpg --export your@address.com > your_public.gpg
```

The key is exported in binary format, which is not very well suited as a visual representation. By also giving the option `--armor`, gpg will use a binary-to-ASCII encoding to output a more readable file.

```
$ gpg --export --armor your@address.com
$ gpg --export --armor your@address.com > your_public.gpg
```

Please share `your-public.gpg` file via the Ed forum with other lab group and request them to send theirs. To import other lab group's key and view the keys available in your keyring:

```
$ gpg --import your_friend_public.gpg
$ gpg --list-keys
```

Once you have imported a key it should be validated. This is done by generating a fingerprint from the imported key and verify that to be the same as the one generated by the owner.

```
$ gpg --fingerprint friend@address.com
```

If all checks out, then you can be confident that you have a correct copy of the key, and you certify that by signing the copy with your own key. You should be very careful to always verify the key's fingerprint with the owner before you sign.
If you have multiple local keys to choose one for signing other users' keys you can provide `--local-user` option followed by the ID of the local user. This option must be provided right after the gpg command.

```
gpg --local-user sierra@evilcorp.com --sign-key alice@evilcorp.com
```

```
$ gpg --sign-key friend@address.com
```

3.1.3 Encryption and Decryption

Please create a text file plain.txt, if you want to encrypt the file so that only for friend@address to read, then type:

```
$ gpg --encrypt --output cipher.txt --recipient \
  friend@address.com plain.txt
```

Which will create the encrypted cipher.txt. Please share this file through the Ed forum to other lab group. The file can be decrypted by typing:
Create a large enough file to compare the encryption speed between symmetric and asymmetric encryption algorithm.

```
dd if=/dev/urandom of=./large.bin count=500000 bs=1K
time gpg --encrypt --output clarge.bin --recipient alice.wonderland@monash.edu large.bin

File `clarge.bin' exists. Overwrite? (y/N) y

real        0m28.276s
user        0m22.762s
sys        0m1.869s
```

```
$ gpg --output plain-dec.txt --decrypt friend-cipher.txt
```

You should now transmit and receive messages securely with another lab group.

4 Generating RSA Key Pair Using SageMath

In this task you will use the SageMath library to generate a 1024-bit RSA key pair and use the generated keys to encrypt a message and decrypt the corresponding ciphertext.

1. use p=random_prime(lbound=2**511, n=2**512) to generate a prime number of 512 bits.
2. Repeat the above step to generate another prime number of 512 bits q.
3. Using the values of p and q calculate the velues of n and $\phi(n)$.

   ```
   n=p*q
   phi=(p-1)*(q-1)
   ```

4. Choose the value of e=65537 and check whether $gcd(e, \phi(n)) = 1$ and if true calculate the value of d. (Hint: Use gcd and inverse_mod functions).

   ```
   e=65537
   gcd(e,phi)
   d=inverse_mod(e,phi)
   ```

5. Use ZZ.random_element(2**127, 2**128) to generate a random 128-bit value for the message m.

   ```
   m=ZZ.random_element(2**127, 2**128)
   ```

6. Encrypt the message m using the public key.

```
c=power_mod(m,e,n)
```

7. Decrypt the ciphertext created in previous step using the private key. Does the recovered value match the original message m?

```
mrec=power_mod(c,d,n)
```