A QUICK, EASY YET COMPREHENSIVE GPG TUTORIAL

Posted on July 28, 2022 by Marco Antonio Carcano

GnuPG is a freely available command line encryption suite based on OpenPGP encryption library, and it is probably the most used encryption suite in the world. Every IT professional sooner or later face use cases that require to know how to handle it, but the point is that, despite a trivial usage of this suite may look quite easy, there are some not so obvious nuances that if known can really improve the experience, also saving from making some hidden mistakes that can really painful if anything would go wrong.

This is a quick, easy yet comprehensive GPG tutorial with the aim to provide a quick yet thorough explanation of how to safely use this amazing encryption suite.

PGP, GNUPG, OPEN PGP

These are tightly bound terms that quite often people mislead: this is certainly because of the lots of changes that happened during the long story of the evolution of PGP.

PGP - PRETTY GOOD PRIVACY

PGP (Pretty Good Privacy) is an encryption software developed by **Phil Zimmermann** that uses a serial combination of hashing, data compression, symmetric-key cryptography and of course public-key cryptography.

The world should say a very big thanks to Phil Zimmermann: he is a true philanthropist! He was a long-time anti-nuclear activist, and created PGP encryption to let people exchange confidential messages without having to fear eavesdropping. No license fee was required for its non-commercial use, and the complete source code was included with all copies. He was so keen into freedom of communication to become a target of a criminal investigation by the US Government for "munitions export without a license" in 1993 - at the time within the definition of the US export regulations cryptosystems with keys of more than 40 bits in length were considered munitions: lucky the investigation was closed without filing criminal charges against him or anyone else. He challenged publishing the entire source code of PGP in a hardback book which was broadly distributed and sold.

OPEN PGP

In 1997 Zimmermann became convinced that an open standard for PGP encryption was critical for them and for the cryptographic community as a whole: PGP Inc. proposed to the IETF a new standard granting them permission to use the name OpenPGP to describe it as well as any program that supported the standard. The OpenPGP standard (RFC-4880), managed by the OpenPGP Working Group, is aimed at defining a mail encryption standard.

GPG - GNU PG

GnuPG (GPG) is the free implementation of the PG encryption develped by Werner Koch in 1999. It probably is the most used suite that implements PG.

THE LAB ENVIRONMENT

The most straightforward way to learn GPG is seeing it in action, seeing the interactions between at least five users; for this reason we are going to setup a small lab with the following users:

- foo
- bar
- baz
- qux
- fred

We will use them to learn how to generate the keys and to trust someone else's keys as well as exchange GPG-encrypted or signed data.

INSTALL AND CONFIGURE THE VIM PLUGIN

For the ancient ones (like me) who love the old fashioned Vim, there's a GPG plugin that lets users **seamlessly edit encrypted files within vim**.

Since vim is broadly used, it seems wise to deliver this plugin along with some reasonable defaults to every newly created user; it is enough adding it to the **/etc/skel** directory as follows:

```
sudo mkdir -p  /etc/skel/.vim/pack/bundle/start
sudo dnf install -y git
sudo git clone https://github.com/jamessan/vim-gnupg.git /etc/skel/.vim/pack/bundle/
```

We also add the **/etc/skel/.vimrc** configuration file and set up as to to have Vim create armored files (ASCII files, so that they are easy to cut-and-paste or to send by email).

```
" Armor files
let g:GPGPreferArmor=1
```

Finally, let's configure the gpg-agent related variables, or the plugin can get mixed-up about what values to use:

```
sudo bash -c "cat >> /etc/profile.d/gpg.sh" << "EOF"
GPG_TTY=`tty`
export GPG_TTY
EOF
```

CREATE THE USERS

Let's create the users necessary for this lab as follows:

```
sudo adduser foo
sudo adduser bar
sudo adduser baz
sudo adduser qux
sudo adduser fred
```

then we set a password to each of them.

foo user:

```
sudo passwd foo
```

bar user:

```
sudo passwd bar
```

baz user:

```
sudo passwd baz
```

qux user:

```
sudo passwd qux
```

and of course **fred**:

```
sudo passwd fred
```

The aim of this lab is mock-up real-life: in a real-life scenario most of the times the system engineer connects to a remote host using SSH and once connected type the GPG commands - or switches to other users using sudo before typing them. For this reason I'm either showing some commands after switching the user with sudo, or submitting them using SSH.

Since the actual behavior is the same, to keep our lab conveniently small, instead of connecting to a remote system we SSH connect to the localhost with these users .

Let's SSH connect as any of the above users:

```
ssh foo@localhost whoami
```

Of course the first time we have to **set as trusted the fingerprint the host key** of the localhost:

```
The authenticity of host 'localhost (::1)' can't be established.
ECDSA key fingerprint is SHA256:uTJ2NlprMvpiMbtCIW1Z0vBWacam13gkvHQOPcqpH7c.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

simply type "*yes*".

CONFIGURE USER-SPECIFIC GPG SETTINGS

Obviously each user can customize GPG as by his own needs - for example, to customize it for the "*foo*" user:

```
sudo su - foo
```

create the **~/.gnupg** directory:

```
mkdir -m 700 ~/.gnupg
```

GnuPG reads its configuration from the **~/.gnupg/gpg.conf** file - create it with the following contents:

```
no-greeting
```

```
require-cross-certification

# default-recipient-self

keyid-format long
with-fingerprint

personal-digest-preferences SHA512
cert-digest-algo SHA512
default-preference-list SHA512 SHA384 SHA256 SHA224 AES256 AES192 AES CAST5 ZLIB BZIP2 ZI
P Uncompressed

fixed-list-mode

no-emit-version
#no-comments

verify-options show-uid-validity
list-options show-uid-validity


keyserver hkps://keys.openpgp.org
keyserver-options no-honor-keyserver-url
keyserver-options include-revoked
keyserver-options auto-key-retrieve
```

this sample file contains some options to avoid some information leakage and to prefer strong algorithms.

About KeyServers, mind there is a notice dated 2021-06-21 on sks-keyservers.net stating "Due to even more GDPR takedown requests, the DNS records for the pool will no longer be provided at all".

When done with editing this file, disconnect the "*foo*" user:

```
exit
```

GETTING ACQUAINTED TO GPG

This post focuses on GnuPG 2.1.18 or later - this means that if you are using Red Hat or CentOS 8 or Rocky Linux 8 the GPG suite is already installed with the right version.

Let's verify the installed version on your system:

```
gpg --version
```

the output on my system is:

```
gpg (GnuPG) 2.2.20
libgcrypt 1.8.5
Copyright (C) 2020 Free Software Foundation, Inc.
```

```
License GPLv3+: GNU GPL version 3 or later <https://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Home: /home/vagrant/.gnupg
Supported algorithms:
Pubkey: RSA, ELG, DSA, ECDH, ECDSA, EDDSA
Cipher: IDEA, 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH,
        CAMELLIA128, CAMELLIA192, CAMELLIA256
Hash: SHA1, RIPEMD160, SHA256, SHA384, SHA512, SHA224
Compression: Uncompressed, ZIP, ZLIB, BZIP2
```

as you can see, ECDH is listed among the Pubkey: this means that this version does support Elliptic Curves keys. In order to choose which one to use, we can list the supported ECC curves as follows:

```
gpg --with-colons --list-config curve
```

the output on my system is:

```
cfg:curve:cv25519;ed25519;nistp256;nistp384;nistp521;secp256k1
```

Although it is not strictly necessary for going on with this post, having a good understanding of cryptography helps in understanding the rest of it. I suggest you read Cryptography Quick Guide – Understand Symmetric And Asymmetric Cryptography And How To Use It With Openssl.

GENERATING THE KEY-PAIR(S)

GPG relies both on symmetric and asymmetric cryptography: more precisely it uses

- **public keys** to **encrypt** files and **verify digital signatures**
- **private keys** to **digitally sign files** or **decrypt files**

The best practice in a real-life scenario is to have a **Primary private key** used to generate **subkeys** either for **encrypt**, **sign** or **authentication** specific purposes, but you can just have a single key to do everything as a whole, although this is not recommended.

So the very first thing to do is generate your own keys.

KEY TYPES AND CAPABILITIES

GPG Keys are called:

- **Keys** (The Primary Keys): this kind of key has **Certification** and **Signing** capabilities
  - **sec** identifies the secret key
  - **pub** identifies the public key
- **Subkeys** (Keys bound to the Primary Key)
  - **ssb** identifies the secret key
  - **sub** identifies the public key

Mind that keys can have one or more of the following capabilities:

Capability	Description	Key	Subkey
C	Certification – it is used to certify the identities – subkey cannot have this capability	X	
S	Signing other keys or data	X	X
E	Encrypting data		X
A	Authentication (it requires Signing and Encrypting capability).Example use cases are: SSH as bare ssh-rsa keys (monkeysphere subkey-to-ssh-agent or HSM) SSH as pgp-sign-rsa certificates TLS according to RFC 5081 (supported by GnuTLS) I guess there are other software that can use it		X

GENERATING THE PRIMARY KEY PAIR

The GPG environment of a user gets initialized as soon as he generates (or imports) its first private key.

My personal advice is to use both the **--expert** and **--full-gen-key** command line options since they enable all the features.

Just to show you - don't type it:

```
gpg --expert --full-gen-key
```

the outcome is:

```
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)
(7) DSA (set your own capabilities)
(8) RSA (set your own capabilities)
(9) ECC and ECC
(10) ECC (sign only)
(11) ECC (set your own capabilities)
(13) Existing key
(14) Existing key from card
Your selection?
```

As you see, thanks to the previous command line switches, we can choose among 14 different options; some of them have the **key type twice** - for example "*RSA and RSA*": these options are used to create both the **Primary Key and a subkey within a single task**. The default choice is "*(1) RSA and RSA*".

WHILE CONNECTED USING SSH

Lets generate "*bar*" user's Primary Key simulating the use case of an SSH connected user:

```
ssh -t bar@localhost "gpg --quick-gen-key 'Bar User (Bar User Primary Key) <bar@carc
```

*Mind that we have to set a TTY (**-t** option) when SSH connecting, or we will have problems with the pinentry when typing the password of the key.*

as you see, it is as:

- we picked up the **option "*(8) RSA (set your own capabilities)*"**
- set the **sign** and **cert** capabilities
- set the key to never expire

Right after typing the password to SSH connect as foo user, gnupg asks for the password to be used protect the key we are about to generate:

```
┌──────────────────────────────────────────────────┐
│ Please enter the passphrase to                   │
│ protect your new key                             │
│                                                  │
│ Passphrase: _____________________________________ │
│                                                  │
│                                     <OK> <Cancel> │
└──────────────────────────────────────────────────┘
```

choose a good password and wait until the generation of the key succeeds.

The last part of the output is as follows:

```
Note that this key cannot be used for encryption. You may want to use
the command "--edit-key" to generate a subkey for this purpose.
pub rsa2048 2022-07-21 [SC]
1414EDCC5BBBC87956624F381A44B621829F5714
uid Bar User (Bar User Primary Key) <bar@carcano.local>
```

In real life we would SSH login for true and then type the command to generate the Primary Key, ... but here to go a little bit faster instead of logging in we directly executed the GPG statement to generate the keys.

As by the note in the previous snippet, this key **cannot be used for encryption**: to say you all, the key-pair generated (we can call it primary-key) is **used only for identifying the owner, ...** we'll get on this very soon.

The best practice with GPG is avoiding to operate using the Primary Keys: just create subkeys for encryption and for digital signature, then save the Primary Key and its revocation certificate off-line, and store them in a very secure place.

I'm pretty sure you're asking why ever do you need subkeys. The answer is for your own security: by doing so you can secure and store off-line the Primary Key pair: this spares you from the risk of stealing of the keys or of the device (for example if you are using a laptop)

or from hardware failures. You only have to take absolute care of the store you use for securing your Primary Key pair (it can be a USB stick, HSM or whatever you think is secure enough for your use case). Mind that your own reputation is bound to whatever happens to your Primary Key!

WHILE SWITCHING USER WITH SUDO

This is the time to show you how to deal with sudo, ... let's switch to the "*baz*" user:

```
sudo su - baz
```

we can now create the Primary Key as follows - note that this time we have to supply **--pinentry-mode loopback** or we won't be granted the right to access the to enter the password, causing the command to fail:

```
gpg --pinentry-mode loopback --quick-gen-key 'Baz User (Baz User Primary Key) <baz@c
```

as you see, we are generating a key with the same features of the one we have just created for the "*bar*" user:

- "***(8) RSA (set your own capabilities)***"
- **sign** and **cert** capabilities
- key never expires

exit the sudoed shell to the previous user:

```
exit
```

ELLIPTIC CURVE KEYS

As for the foo user, ... we generate a key with the following features:

- **ECC** with **nistp256** elliptic curve
- **sign** and **cert** capabilities
- key never expires

This time we do not switch user, we run the command "as the foo user" (**-u foo**):

```
sudo -u foo gpg --pinentry-mode loopback --quick-gen-key 'Foo User (Foo User Primary
```

ⓘ *Mind that ECC uses different keys for encryption (ECDH) and sign in (ECDSA or EdDSA).*

Let's generate the Primary Key for the "*qux*" user:

```
sudo -u qux gpg --pinentry-mode loopback --quick-gen-key 'Qux User (Qux User Primary
```

and for the "*fred*" user:

```
sudo -u fred gpg --pinentry-mode loopback --quick-gen-key 'Fred User (Fred User Prim
```

GENERATING THE ENCRYPTION SUBKEY

We are ready to create a subkey for encryption purposes - become the "*foo*" user:

```
sudo su - foo
```

now launch the interactive wizard to create the subkey as follows:

```
gpg --expert --edit-key foo@carcano.local addkey
```

here is a snippet of the interactive procedure:

```
Please select what kind of key you want:
(3) DSA (sign only)
(4) RSA (sign only)
(5) Elgamal (encrypt only)
(6) RSA (encrypt only)
(7) DSA (set your own capabilities)
(8) RSA (set your own capabilities)
(10) ECC (sign only)
(11) ECC (set your own capabilities)
(12) ECC (encrypt only)
(13) Existing key
(14) Existing key from card
Your selection? 12
Please select which elliptic curve you want:
(1) Curve 25519
(3) NIST P-256
(4) NIST P-384
(5) NIST P-521
(9) secp256k1
Your selection? 3
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) 1y
Key expires at Fri Jul 21 09:42:47 2023 UTC
Is this correct? (y/N) y
Really create? (y/N) y
```

as you see we created a key with the following features:

- "*(12) **ECC** (encrypt only)*"
- using elliptic curve **NIST P-256** "*(3) NIST P-256*"

- valid for one year only (**1y**)

the last part of the output of the generation wizard lists the available keys, with the new one among them (**ssb nistp256/F43E205F0A259AC7**)

```
sec nistp256/7B221C7A2ACA4DE9
created: 2022-07-21 expires: never usage: SC
trust: ultimate validity: ultimate
ssb nistp256/F43E205F0A259AC7
created: 2022-07-21 expires: 2023-07-21 usage: E
[ultimate] (1). Foo User (Foo User Primary Key) <foo@carcano.local>
```

type "*save*" to store the new key into the keyring and exit from the GPG interactive command prompt back to the shell:

```
gpg> save
```

Let's create an encryption subkey also for the "*bar*" user - just exit the sudoed shell we are currently in and become the "*bar*" user:

```
exit
sudo su - bar
```

we can avoid to use the interactive procedure, but first we need the fingerprint of the Primary Key:

```
gpg --list-secret-keys --fingerprint
```

the output is as follows:

```
/home/bar/.gnupg/pubring.kbx
----------------------------
sec rsa2048 2022-07-21 [SC]
1414 EDCC 5BBB C879 5662 4F38 1A44 B621 829F 5714
uid [ultimate] Bar User (Bar User Primary Key) <bar@carcano.local>"
```

the interactive procedure can be skipped by using the **--quick-add-key** option along with the answers to the questions of the interactive wizard right at the end of the statement:

```
gpg --pinentry-mode loopback --quick-add-key '1414 EDCC 5BBB C879 5662 4F38 1A44 B62
```

we generated a key with the following features:

- **RSA**
- **encryption** capability
- valid for **one year** only

exit the sudoed shell:

```
exit
```

GENERATING THE SIGNING SUBKEY

Same way we created an encryption subkey, we can create a signing subkey - become the "*foo*" user:

```
sudo su - foo
```

now launch the interactive wizard to create the subkey as follows:

```
gpg --expert --edit-key foo@carcano.local addkey
```

here is a snippet of the interactive procedure:

```
Please select what kind of key you want:
(3) DSA (sign only)
(4) RSA (sign only)
(5) Elgamal (encrypt only)
(6) RSA (encrypt only)
(7) DSA (set your own capabilities)
(8) RSA (set your own capabilities)
(10) ECC (sign only)
(11) ECC (set your own capabilities)
(12) ECC (encrypt only)
(13) Existing key
(14) Existing key from card
Your selection? 10
Please select which elliptic curve you want:
(1) Curve 25519
(3) NIST P-256
(4) NIST P-384
(5) NIST P-521
(9) secp256k1
Your selection? 3
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) 1y
Key expires at Fri Jul 21 09:53:06 2023 UTC
Is this correct? (y/N) y
Really create? (y/N) y
```

as you see we created a key with the following features:

- "*(10) **ECC** (sign only)*"
- using elliptic curve **NIST P-256** "*(3) NIST P-256*"
- valid for one year only (**1y**)

the last part of the output of the generation wizard lists the available keys, with the new one among

them (ssb nistp256/4401D4D107578972)

```
sec nistp256/7B221C7A2ACA4DE9
created: 2022-07-21 expires: never usage: SC
trust: ultimate validity: ultimate
ssb nistp256/F43E205F0A259AC7
created: 2022-07-21 expires: 2023-07-21 usage: E
ssb nistp256/4401D4D107578972
created: 2022-07-21 expires: 2023-07-21 usage: S
[ultimate] (1). Foo User (Foo User Primary Key) <foo@carcano.local>
```

type "*save*" to store the new key into the keyring and exit from the GPG interactive command prompt back to the shell:

```
gpg> save
```

Let's create an signing subkey also for the "*baz*" user - just exit the sudoed shell we are currently in and become the "*baz*" user:

```
exit
sudo su - baz
```

as we saw, we can avoid to use the interactive procedure, but first we need the fingerprint of the Primary Key:

```
gpg --list-secret-keys --fingerprint
```

the output is as follows:

```
/home/baz/.gnupg/pubring.kbx
----------------------------
sec rsa2048 2022-07-21 [SC]
42C1 515B D21D 96B0 6479 D914 7404 BF53 8718 1611
uid [ultimate] Baz User (Baz User Primary Key) <baz@carcano.local>
```

the interactive procedure can be skipped by using the **--quick-add-key** option along with the answers to the questions of the interactive wizard right at the end of the statement:

```
gpg --pinentry-mode loopback --quick-add-key '42C1 515B D21D 96B0 6479 D914 7404 BF5
```

we generated a key with the following features:

- **RSA**
- **sign** capability
- valid for **one year** only

exit the sudoed shell:

```
exit
```

GENERATING THE AUTHENTICATION SUBKEY

Same way we created an encryption and a signing subkeys, we can create an authentication subkey - become the "*foo*" user:

```
sudo su - foo
```

now launch the interactive wizard to create the subkey as follows:

```
gpg --expert --edit-key foo@carcano.local addkey
```

here is a snippet of the interactive procedure:

```
Please select what kind of key you want:
   (3) DSA (sign only)
   (4) RSA (sign only)
   (5) Elgamal (encrypt only)
   (6) RSA (encrypt only)
   (7) DSA (set your own capabilities)
   (8) RSA (set your own capabilities)
  (10) ECC (sign only)
  (11) ECC (set your own capabilities)
  (12) ECC (encrypt only)
  (13) Existing key
  (14) Existing key from card
Your selection? 11

Possible actions for a ECDSA/EdDSA key: Sign Authenticate
Current allowed actions: Sign

   (S) Toggle the sign capability
   (A) Toggle the authenticate capability
   (Q) Finished

Your selection? a

Possible actions for a ECDSA/EdDSA key: Sign Authenticate
Current allowed actions: Sign Authenticate

   (S) Toggle the sign capability
   (A) Toggle the authenticate capability
   (Q) Finished

Your selection? q
Please select which elliptic curve you want:
   (1) Curve 25519
   (3) NIST P-256
   (4) NIST P-384
   (5) NIST P-521
   (9) secp256k1
```

```
Your selection? 3
Please specify how long the key should be valid.
         0 = key does not expire
        = key expires in n days
      w = key expires in n weeks
      m = key expires in n months
      y = key expires in n years
Key is valid for? (0) 1y
Key expires at Fri Jul 21 10:03:00 2023 UTC
Is this correct? (y/N) y
Really create? (y/N) y
```

as you see we created a key with the following features:

- "*(11) ECC (set your own capabilities)*"
- added the "authenticate" capability "(A)"
- using elliptic curve **NIST P-256** "*(3) NIST P-256*"
- valid for one year only (**1y**)

the last part of the output of the generation wizard lists the available keys, with the new one among them (nistp256/F9C6C45EB9650672):

```
sec nistp256/7B221C7A2ACA4DE9
created: 2022-07-21 expires: never usage: SC
trust: ultimate validity: ultimate
ssb nistp256/F43E205F0A259AC7
created: 2022-07-21 expires: 2023-07-21 usage: E
ssb nistp256/4401D4D107578972
created: 2022-07-21 expires: 2023-07-21 usage: S
ssb nistp256/F9C6C45EB9650672
created: 2022-07-21 expires: 2023-07-21 usage: SA
[ultimate] (1). Foo User (Foo User Primary Key) <foo@carcano.local>
```

you can see, now foo user three private subkeys:

- nistp256/F43E205F0A259AC7 for encryption (usage: E)
- nistp256/4401D4D107578972 for signing (usage: S)
- nistp256/F9C6C45EB9650672 authentication (usage: SA)

type "*save*" to store the new key into the keyring and exit from the GPG interactive command prompt back to the shell:

```
gpg> save
```

GENERATING THE REVOCATION CERTIFICATE

The aim of a revocation certificate is providing a way to invalidate the key if it gets stolen.Despite gpg automatically generates the revocation certificate when generating the Primary Key (as by the best practice), it is wise to know how to generate other revocation certificates if necessary:

If you are not working as the "*foo*" user, switch to it using sudo:

```
sudo su - foo
```

now generate the revocation certificate as follows:

```
gpg --gen-revoke --armor --output=foo@carcano.local.revocation.asc foo@carcano.local
```

the output is:

```
sec  nistp256/7B221C7A2ACA4DE9 2022-07-21 Foo User (Foo User Primary Key) <foo@carcano.lo
cal>

Create a revocation certificate for this key? (y/N) y
Please select the reason for the revocation:
  0 = No reason specified
  1 = Key has been compromised
  2 = Key is superseded
  3 = Key is no longer used
  Q = Cancel
(Probably you want to select 1 here)
Your decision? 1
Enter an optional description; end it with an empty line:
>
Reason for revocation: Key has been compromised
(No description given)
Is this okay? (y/N) y
Revocation certificate created.
```

You must keep the certificate in a very secure place, since anybody who gets it can invalidate your key and damage your reputation on the web of trust. The best practice is to store it offline, along with the Primary Key, for example on an USB stick, put into a bank safety deposit box.

OPERATING SECRET KEYS

Now that every user of the lab has some GPG keys, we can see how to operate on keys. The very most of the following examples requires to be run as "*foo*" user, so we switch to the "*foo*" user right now:

```
sudo su - foo
```

LISTING THE AVAILABLE KEYS

We can list the secret (private) keys as follows:

```
gpg --list-secret-keys
```

the output is:

```
/home/foo/.gnupg/pubring.kbx
----------------------------
sec nistp256/7B221C7A2ACA4DE9 2022-07-21 [SC]
Key fingerprint = 56AE 9AFF A54F A111 B08C F0C4 7B22 1C7A 2ACA 4DE9
uid [ultimate] Foo User (Foo User Primary Key) <foo@carcano.local>
ssb nistp256/F43E205F0A259AC7 2022-07-21 [E] [expires: 2023-07-21]
ssb nistp256/4401D4D107578972 2022-07-21 [S] [expires: 2023-07-21]
ssb nistp256/F9C6C45EB9650672 2022-07-21 [SA] [expires: 2023-07-21]
```

The first column specifies the key type:

- **sec**: SECret key
- **ssb**: Secret SuBkey

so the output shows:

- a private key (the line that starts by "*sec*")
- some subkeys (the lines that start by "*ssb*")
- the uid of the secret key (the line that starts by "*uid*")

ADDING ADDITIONAL UID TO A PRIMARY KEY

You may want to add one or more additional uids to a Primary Key, but first you must enter the interactive procedure to modify the Primary Key.

In this example we are modifying foo@carcano.local Primary Key:

```
gpg --pinentry-mode loopback --edit-key '<foo@carcano.local>'
```

just type **adduid** to enter the interactive wizard to add a new uid to the Primary key:

```
gpg> adduid
```

here is a snippet of the wizard:

```
Real name: Thud User
Email address: thud@carcano.local
Comment: Thud User Primary Key
You selected this USER-ID:
"Thud User (Thud User Primary Key)<thud@carcano.local>"
Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O
```

the output is as follows:

```
sec nistp256/7B221C7A2ACA4DE9
created: 2022-07-21 expires: never usage: SC
trust: ultimate validity: ultimate
ssb nistp256/F43E205F0A259AC7
created: 2022-07-21 expires: 2023-07-21 usage: E
ssb nistp256/4401D4D107578972
created: 2022-07-21 expires: 2023-07-21 usage: S
ssb nistp256/F9C6C45EB9650672
```

```
created: 2022-07-21 expires: 2024-07-25 usage: SA
[ultimate] (1) Foo User (Foo User Primary Key) <foo@carcano.local>
[ unknown] (2). Thud User (Thud User Primary Key)<thud@carcano.local>
```

don't bother for the "*[unknown]*" trust on the left of Thud's uid: it's just a matter of reloading.

Remember to save the changes before getting back to the shell:

```
gpg> save
```

by now Thud's key is shown as "*trust ultimate*" when you run the "*gpg --list-secret-keys*" statement.

SELECTING A SPECIFIC UID

In order to run commands for a specific uid, you must first select it: just launch the interactive edit of the key (in this example the Primary Key is foo@carcano.local):

```
gpg --pinentry-mode loopback --edit-key '<foo@carcano.local>'
```

the output is as follows:

```
Secret key is available.

sec  nistp256/7B221C7A2ACA4DE9
     created: 2022-07-21  expires: never       usage: SC
     trust: ultimate      validity: ultimate
ssb  nistp256/F43E205F0A259AC7
     created: 2022-07-21  expires: 2023-07-21  usage: E
ssb  nistp256/4401D4D107578972
     created: 2022-07-21  expires: 2023-07-21  usage: S
ssb  nistp256/F9C6C45EB9650672
     created: 2022-07-21  expires: 2024-07-25  usage: SA
[ultimate] (1). Foo User (Foo User Primary Key) <foo@carcano.local>
[ultimate] (2)  Thud User (Thud User Primary Key)<thud@carcano.local>
```

now select (by index) the uid you are interested to run commands onto.

For example, to select Thud's uid:

```
gpg> uid 2
```

now the output is:

```
sec nistp256/7B221C7A2ACA4DE9
created: 2022-07-21 expires: never usage: SC
trust: ultimate validity: ultimate
ssb nistp256/F43E205F0A259AC7
created: 2022-07-21 expires: 2023-07-21 usage: E
ssb nistp256/4401D4D107578972
created: 2022-07-21 expires: 2023-07-21 usage: S
ssb nistp256/F9C6C45EB9650672
created: 2022-07-21 expires: 2024-07-25 usage: SA
```

```
[ultimate] (1). Foo User (Foo User Primary Key) <foo@carcano.local>
[ultimate] (2)* Thud User (Thud User Primary Key)<thud@carcano.local>
```

note that now Thud's uid is marked with a "star" character - this means that the uid has been selected, and that any following command will be related to this uid. Since we are only learning how to select a uid, we do not run any other command.

Type exit to get back to the shell:

```
gpg> exit
```

DELETING A SPECIFIC UID

You may want to delete a specific uid from a Primary Key, but first you must enter the interactive procedure to modify the Primary Key.

In this example we are modifying foo@carcano.local Primary Key:

```
gpg --pinentry-mode loopback --edit-key '<foo@carcano.local>'
```

the output is as follows:

```
Secret key is available.

sec  nistp256/7B221C7A2ACA4DE9
     created: 2022-07-21  expires: never       usage: SC
     trust: ultimate      validity: ultimate
ssb  nistp256/F43E205F0A259AC7
     created: 2022-07-21  expires: 2023-07-21  usage: E
ssb  nistp256/4401D4D107578972
     created: 2022-07-21  expires: 2023-07-21  usage: S
ssb  nistp256/F9C6C45EB9650672
     created: 2022-07-21  expires: 2024-07-25  usage: SA
[ultimate] (1). Foo User (Foo User Primary Key) <foo@carcano.local>
[ultimate] (2)  Thud User (Thud User Primary Key)<thud@carcano.local>
```

now refer to "Selecting A Specific Uid" to select the uid you want to delete: in this example, pick the key number 2 to delete Thud's uid.

type "*deluid*" to enter the interactive wizard to delete the subkey:

```
gpg> deluid
```

the wizard asks you to confirm:

```
Really remove this user ID? (y/N) y
```

remember to save the changes before getting back to the shell:

```
gpg> save
```

SELECTING SUBKEYS

In order to run commands for a specific subkey, you must first select it: just launch the interactive edit of the key (in this example the Primary Key is foo@carcano.local):

```
gpg --pinentry-mode loopback --edit-key '<foo@carcano.local>'
```

the output is as follows:

```
Secret key is available.

sec  nistp256/7B221C7A2ACA4DE9
     created: 2022-07-21  expires: never       usage: SC
     trust: ultimate      validity: ultimate
ssb  nistp256/F43E205F0A259AC7
     created: 2022-07-21  expires: 2023-07-21  usage: E
ssb  nistp256/4401D4D107578972
     created: 2022-07-21  expires: 2023-07-21  usage: S
ssb  nistp256/F9C6C45EB9650672
     created: 2022-07-21  expires: 2023-07-21  usage: SA
[ultimate] (1). Foo User (Foo User Primary Key) <foo@carcano.local>
```

now select (by index) the subkey (ssb) you are interested to run commands onto.

For example, to select the last subkey:

```
gpg> key 3
```

now the output is:

```
sec  nistp256/7B221C7A2ACA4DE9
     created: 2022-07-21  expires: never       usage: SC
     trust: ultimate      validity: ultimate
ssb  nistp256/F43E205F0A259AC7
     created: 2022-07-21  expires: 2023-07-21  usage: E
ssb  nistp256/4401D4D107578972
     created: 2022-07-21  expires: 2023-07-21  usage: S
ssb* nistp256/F9C6C45EB9650672
     created: 2022-07-21  expires: 2023-07-21  usage: SA
[ultimate] (1). Foo User (Foo User Primary Key) <foo@carcano.local>
```

note that now the last subkey is marked with a "star" character - this means that the subkey has been selected, and that any following command will be related to this subkey. Since we are only learning how to select a subkey, we do not run any other command.

Type exit to get back to the shell:

```
gpg> exit
```

EXTENDING THE EXPIRATION OF A SUBKEY

You may want to extend the expire time of a subkey of a Primary Key, but first you must enter the interactive procedure to modify the Primary Key.

In this example we are modifying foo@carcano.local Primary Key:

```
gpg --pinentry-mode loopback --edit-key '<foo@carcano.local>'
```

the output is as follows:

```
Secret key is available.

sec  nistp256/7B221C7A2ACA4DE9
     created: 2022-07-21  expires: never       usage: SC
     trust: ultimate      validity: ultimate
ssb  nistp256/F43E205F0A259AC7
     created: 2022-07-21  expires: 2023-07-21  usage: E
ssb  nistp256/4401D4D107578972
     created: 2022-07-21  expires: 2023-07-21  usage: S
ssb  nistp256/F9C6C45EB9650672
     created: 2022-07-21  expires: 2023-07-21  usage: SA
[ultimate] (1). Foo User (Foo User Primary Key) <foo@carcano.local>
```

now refer to "Selecting Subkeys" to select the subkey you want to extend the validity - in this example, pick the key number 2 (nistp256/4401D4D107578972)

type "*expire*" to enter the interactive wizard to setup a new expiration for the subkey:

```
gpg> expire
```

this is a snippet of the interactive procedure:

```
Changing expiration time for a subkey.
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) 2y
Key expires at Thu 25 Jul 2024 10:21:58 AM UTC
Is this correct? (y/N) y
```

remember to save the changes before getting back to the shell:

```
gpg> save
```

DELETING A SUBKEY

You may want to delete a specific subkey of a Primary Key, but first you must enter the interactive

procedure to modify the Primary Key.

In this example we are modifying foo@carcano.local Primary Key:

```
gpg --pinentry-mode loopback --edit-key '<foo@carcano.local>'
```

the output is as follows:

```
Secret key is available.

sec  nistp256/7B221C7A2ACA4DE9
     created: 2022-07-21  expires: never       usage: SC
     trust: ultimate      validity: ultimate
ssb  nistp256/F43E205F0A259AC7
     created: 2022-07-21  expires: 2023-07-21  usage: E
ssb  nistp256/4401D4D107578972
     created: 2022-07-21  expires: 2023-07-21  usage: S
ssb  nistp256/F9C6C45EB9650672
     created: 2022-07-21  expires: 2023-07-21  usage: SA
[ultimate] (1). Foo User (Foo User Primary Key) <foo@carcano.local>
```

now refer to "Selecting Subkeys" to select the key you want to delete - in this example, pick the key number 3 (nistp256/F9C6C45EB9650672)

type "*delkey*" to enter the interactive wizard to delete the subkey:

```
gpg> delkey
```

the wizard asks you to confirm:

```
Do you really want to delete this key? (y/N) y
```

remember to save the changes before getting back to the shell:

```
gpg> save
```

EXPORTING THE SECRET KEYS

You must of course take backups of your keys. Since in this lab we created most of the keys for the "foo" user, if you are not working as this user, switch to it using sudo as follows:

```
sudo su - foo
```

since we switched to the "*foo*" user using sudo, we must always specify the **--pinentry-mode loopback** command option.

GnuPG provides two different ways of exporting secret keys, as described below.

EXPORTING A SPECIFIC SECRET KEY ALONG WITH EVERY SUBKEY

This can be achieved by supplying the **--export-secret-keys** command switch as follows:

```
gpg --pinentry-mode loopback --export-secret-keys --armor 7B221C7A2ACA4DE9 > /mnt/fc
```

EXPORTING ONLY SUBKEYS OF A SPECIFIC KEY

Since subkeys are (only a little bit) less sensitive of the Primary Key, you may want to backup only the private keys of the subkeys of a specific Primary Key, and securely store them into another USB stick to put in a secure storage quicker to reach than the one where you put the USB stick with the Primary Key.

This approach shortens the time to restore them from a backup when needed, since it is quite rare that you need to restore the Primary Key - actually you'll need it only when signing other keys and a very few other cases.

This time you must specify the **--export-secret-subkeys** command switch as by the example below:

```
gpg --pinentry-mode loopback --export-secret-subkeys --armor 7B221C7A2ACA4DE9 > foo@
```

SECURELY WORKING WITH THE PRIMARY KEY

The Primary Key is the most sensitive key, since it is the one that **holds your uids** and so that is tightly **bound to your own reputation**. For this reason you must carefully handle it, and always be able to access it.

The best practice is to:

- generate a revocation certificate right after creating the Primary Key
- export the key to a secure offline device, such as an USB stick (best would be an **HSM**) and store it on a secure place such as a bank safety box

STORE THE PRIMARY KEY AND THE REVOCATION CERTIFICATE OFFLINE

Let's save the Primary Key and revocation certificate offline - for the sake of simplicity, in this tutorial we see how to export it on an USB stick.

Mount the device you want to store the Primary Keys - in this example the USB stick is /dev/sde:

```
mount /dev/sde /mnt
```

switch to the user you want export the Primary Key - in this lab we are using the "*foo*" user:

```
sudo su - foo
```

let's now export the secret keys:

```
gpg --pinentry-mode loopback --export-secret-keys --armor 7B221C7A2ACA4DE9 > /mnt/fc
```

remember that after switching user with sudo it is mandatory to specify the **--pinentry-mode**

loopback command option

create the revocation certificate of the Primary Key if you didn't already, then copy it too to the USB stick:

```
cp foo@carcano.local.revocation.asc /mnt
```

exit the sudoed shell:

```
exit
```

unmount the USB stick:

```
sudo umount /mnt
```

switch again to the "*foo*" user:

```
sudo su - foo
```

we are almost ready to shred **the Primary Key**: as every private key, it is stored beneath **~/.gnupg/private-keys-v1.d**, but we need to know the filename first.

This can be easily achieved by listing the secret keys showing the key grip as follows:

```
gpg --with-keygrip --list-secret-keys 7B221C7A2ACA4DE9
```

output is:

```
sec nistp256/7B221C7A2ACA4DE9 2022-07-21 [SC]
Key fingerprint = 56AE 9AFF A54F A111 B08C F0C4 7B22 1C7A 2ACA 4DE9
Keygrip = 8F0E9847884EF3ABCFFD1E0CBEE2F831A1033357
uid [ultimate] Foo User (Foo User Primary Key) <foo@carcano.local>
ssb nistp256/F43E205F0A259AC7 2022-07-21 [E] [expires: 2023-07-21]
Keygrip = D496BB6AC61001B12FAC4813B4D61E7429E7B9F9
ssb nistp256/4401D4D107578972 2022-07-21 [S] [expires: 2023-07-21]
Keygrip = 83A6E2398552A36636A1F804F2CF9DF165A7355C
ssb nistp256/F9C6C45EB9650672 2022-07-21 [SA] [expires: 2023-07-21]
Keygrip = 0A917D86F6744C226BE445AFDED54E681B80096E
```

the key grip of the Primary Key is 8F0E9847884EF3ABCFFD1E0CBEE2F831A1033357.

Let's **shred** and **remove** the file containing the key from the disk:

```
shred -vz -n 7 .gnupg/private-keys-v1.d/8F0E9847884EF3ABCFFD1E0CBEE2F831A1033357.key
rm -f .gnupg/private-keys-v1.d/8F0E9847884EF3ABCFFD1E0CBEE2F831A1033357.key
```

let's check the outcome:

```
gpg --list-secret-keys 7B221C7A2ACA4DE9
```

the output is:

```
sec# nistp256/7B221C7A2ACA4DE9 2022-07-21 [SC]
Key fingerprint = 56AE 9AFF A54F A111 B08C F0C4 7B22 1C7A 2ACA 4DE9
uid [ultimate] Foo User (Foo User Primary Key) <foo@carcano.local>
ssb nistp256/F43E205F0A259AC7 2022-07-21 [E] [expires: 2023-07-21]
ssb nistp256/4401D4D107578972 2022-07-21 [S] [expires: 2023-07-21]
ssb nistp256/F9C6C45EB9650672 2022-07-21 [SA] [expires: 2023-07-21]
```

note the **pound sign (#) right after the sec word**: this means that the secret key is missing, that is exactly what we wanted to achieve.

IMPORTING THE PRIMARY KEY

Some tasks such as extending the subkeys, generate new ones, or signing other keys require the secret key of the Primary Key to be available: this means that sometime we need to reload the private key from the offline backup.

Mount the device containing the backup of the Primary Keys - in this example it is /dev/sde:

```
mount /dev/sde /mnt
```

switch to the user you want import the Primary Key - in this lab we use the "*foo*" user:

```
sudo su - foo
```

now just type:

```
gpg --pinentry-mode loopback --import /mnt/foo@carcano.local.priv.asc
```

the output is:

```
gpg: key 7B221C7A2ACA4DE9: "Foo User (Foo User Primary Key) <foo@carcano.local>" not
gpg: key 7B221C7A2ACA4DE9: secret key imported
gpg: Total number processed: 1
gpg: unchanged: 1
gpg: secret keys read: 1
gpg: secret keys imported: 1
gpg: secret keys unchanged: 1
```

exit the sudoed shell:

```
sudo su - foo
```

unmount the USB stick:

```
sudo umount /mnt
```

GENERATING A NEW PRIMARY KEY

It is certainly worth the effort to spend some words also on the steps to be followed if you generate a new Primary Key to supersede the old one: in order to have this new key to become automatically trusted by each one who have already set your old key as trusted, you must sign the new Primary Key with the old Primary key indeed.

This can be done as follows:

```
gpg -u oldKeyID --sign-key newKeyID
```

for more information on this topic, read "*The Web Of Trust*" later on.

If it isn't already, set the old key as trusted:

```
gpg -u newKeyID --edit-key oldKeyID trust
```

by doing so, every key you signed with the old key is evaluated as trusted.

Lastly, for your own convenience, modify the default key into **~/.gnupg/gpg.conf** file, so to refer commands to this new key by default:

```
default-key newKeyID
```

OPERATING PUBLIC KEYS

EXPORTING THE PUBLIC KEY(S)

Sometimes it is necessary to export a public key - for example to share a key without using a keyserver.

In this example we are going to export fred's public key, so let's switch to "fred" user:

```
sudo su - fred
```

Now we export the public key of fred's specific Primary Key along with the ones of its subkeys:

```
gpg --export --armor fred@carcano.local > /tmp/fred-gpg.pub
```

IMPORTING THE PUBLIC KEY(S)

We can of course import one or more public keys from a file.

In this example we import into foo's public keyring the fred's public keys we previously exported:

```
sudo -u foo gpg --import /tmp/fred-gpg.pub
```

as you see we did it within a single statement running the command using sudo.

LISTING THE AVAILABLE KEYS

Switch to the user you want to list the keys - in this example we are switching to the "*foo*" user:

```
sudo su - foo
```

now list the available public keys:

```
gpg --list-keys
```

the output is as follows:

```
/home/foo/.gnupg/pubring.kbx
----------------------------
pub nistp256/7B221C7A2ACA4DE9 2022-07-21 [SC]
Key fingerprint = 56AE 9AFF A54F A111 B08C F0C4 7B22 1C7A 2ACA 4DE9
uid [ultimate] Foo User (Foo User Primary Key) <foo@carcano.local>
sub nistp256/F43E205F0A259AC7 2022-07-21 [E] [expires: 2023-07-21]
sub nistp256/4401D4D107578972 2022-07-21 [S] [expires: 2023-07-21]
sub nistp256/F9C6C45EB9650672 2022-07-21 [SA] [expires: 2023-07-21]

pub rsa2048/AB3E2E9F213CC658 2022-07-25 [SC]
    Key fingerprint = 0559 862A C951 CFB8 317C 8A7B AB3E 2E9F 213C C658
uid                 [ unknown] Fred User (Fred User Primary Key) <fred@carcano.local>
```

the first column specifies the key type:

- **pub**: PUBlic key
- **sub**: public SUBkey

SHOWING FINGERPRINTS AND KEY-ID

When running some statements - such as when sending keys to keyserver, it is necessary to supply the fingerprint of the key. You can show fingerprints by adding the command switch as follows:

```
gpg --list-keys --with-fingerprint
```

the output is as follows:

```
/home/foo/.gnupg/pubring.kbx
----------------------------
pub   nistp256/7B221C7A2ACA4DE9 2022-07-21 [SC]
```

```
      Key fingerprint = 56AE 9AFF A54F A111 B08C  F0C4 7B22 1C7A 2ACA 4DE9
uid                 [ultimate] Foo User (Foo User Primary Key) <foo@carcano.local>
sub   nistp256/F43E205F0A259AC7 2022-07-21 [E] [expires: 2023-07-21]
      Key fingerprint = 8420 DFC7 4529 DF0A E8A7  2123 F43E 205F 0A25 9AC7
sub   nistp256/4401D4D107578972 2022-07-21 [S] [expires: 2023-07-21]
      Key fingerprint = 6281 0639 A3C3 DB4F 5765  E519 4401 D4D1 0757 8972
sub   nistp256/F9C6C45EB9650672 2022-07-21 [SA] [expires: 2024-07-25]
      Key fingerprint = 2110 42B0 18A8 893B 58BC  06EA F9C6 C45E B965 0672

pub   rsa2048/AB3E2E9F213CC658 2022-07-25 [SC]
      Key fingerprint = 0559 862A C951 CFB8 317C  8A7B AB3E 2E9F 213C C658
uid                 [ unknown] Fred User (Fred User Primary Key) <fred@carcano.local
```

the last 20 characters of the fingerprint are the **key id**.

DELETING A PUBLIC KEY

Switch to the user you want to delete the key - in this example we are switching to the "*foo*" user:

```
sudo su - foo
```

then list the available public keys as follows:

```
gpg --list-keys
```

the output is as follows:

```
/home/foo/.gnupg/pubring.kbx
----------------------------
pub nistp256/7B221C7A2ACA4DE9 2022-07-21 [SC]
Key fingerprint = 56AE 9AFF A54F A111 B08C F0C4 7B22 1C7A 2ACA 4DE9
uid [ultimate] Foo User (Foo User Primary Key) <foo@carcano.local>
sub nistp256/F43E205F0A259AC7 2022-07-21 [E] [expires: 2023-07-21]
sub nistp256/4401D4D107578972 2022-07-21 [S] [expires: 2023-07-21]
sub nistp256/F9C6C45EB9650672 2022-07-21 [SA] [expires: 2023-07-21]

pub rsa2048/AB3E2E9F213CC658 2022-07-25 [SC]
    Key fingerprint = 0559 862A C951 CFB8 317C 8A7B AB3E 2E9F 213C C658
uid               [ unknown] Fred User (Fred User Primary Key) <fred@carcano.local>
```

we can now delete the public key by using the --delete-key command switch.

For example, to delete fred's public key:

```
gpg --delete-key '<fred@carcano.local>'
```

this is a snippet of the wizard:

```
pub  rsa2048/AB3E2E9F213CC658 2022-07-25 Fred User (Fred User Primary Key) <fred@carcano.
local>

Delete this key from the keyring? (y/N) y
```

DELIVERING THE PUBLIC KEY TO GPG KEYSERVERS

In real life you don't need to export public keys to files to exchange them to other parties: you just have to send your public keys to keyservers. These are public available servers that store every GPG public key the GPG community delivers to them. This means that GPG users can easily retrieve the public key of other parties simply asking them to these servers.

Mind there is a notice dated 2021-06-21 on sks-keyservers.net stating "Due to even more GDPR takedown requests, the DNS records for the pool will no longer be provided at all".

It is really easy to send public keys to the key servers - just mind that you must supply the key-id, so first you have to list the keys with the fingerprints so to guess the key id:

```
gpg --list-keys --with-fingerprint
```

the output is as follows:

```
/home/foo/.gnupg/pubring.kbx
----------------------------
pub   nistp256/7B221C7A2ACA4DE9 2022-07-21 [SC]
      Key fingerprint = 56AE 9AFF A54F A111 B08C  F0C4 7B22 1C7A 2ACA 4DE9
uid                 [ultimate] Foo User (Foo User Primary Key) <foo@carcano.local>
sub   nistp256/F43E205F0A259AC7 2022-07-21 [E] [expires: 2023-07-21]
      Key fingerprint = 8420 DFC7 4529 DF0A E8A7  2123 F43E 205F 0A25 9AC7
sub   nistp256/4401D4D107578972 2022-07-21 [S] [expires: 2023-07-21]
      Key fingerprint = 6281 0639 A3C3 DB4F 5765  E519 4401 D4D1 0757 8972
sub   nistp256/F9C6C45EB9650672 2022-07-21 [SA] [expires: 2024-07-25]
      Key fingerprint = 2110 42B0 18A8 893B 58BC  06EA F9C6 C45E B965 0672

pub   rsa2048/AB3E2E9F213CC658 2022-07-25 [SC]
      Key fingerprint = 0559 862A C951 CFB8 317C  8A7B AB3E 2E9F 213C C658
uid                 [ unknown] Fred User (Fred User Primary Key) <fred@carcano.local
```

the last 20 characters of the fingerprint are the **key id**.

Now you can run the actual statement to send the key to the server:

```
gpg --send-keys 7B221C7A2ACA4DE9
gpg: sending key 7B221C7A2ACA4DE9 to hkps://keys.openpgp.org
```

Don't do this while experimenting, but when you create your real keys you should publish

every public key and subkey (that belong to you). By the way, note that if necessary, you can even run your own GPG keyserver and have a private keyserver dedicated to your business.

you can of course override the default keyserver where to send the key by supplying the **--keyserver** option followed by the FQDN of the keyserver you want to use - for example "*--keyserver pgp.mit.edu*".

FETCHING PUBLIC KEYS FROM GPG KEYSERVERS

We can of course fetch the GPG public keys of other entities (either humans or devices) that have been submitted to Key Servers. By the way, if you just need to look up keys, you may find it convenient to use the web ui provided by https://keys.openpgp.org/.

Mind there is a notice dated 2021-06-21 on sks-keyservers.net stating "Due to even more GDPR takedown requests, the DNS records for the pool will no longer be provided at all".

For example, if "fred" user delivered its public keys to the Key Servers, we would be able to retrieve it as follows:

```
gpg --recv-key 8A7BAB3E2E9F213CC658
```

you can of course override the default keyserver where to send the key by supplying the **--keyserver** option followed by the FQDN of the keyserver you want to use - for example "*--keyserver pgp.mit.edu*".

THE WEB OF TRUST

This is how OpenPGP trusts public keys: you can find all the details at http://www.gnupg.org/gph/en/manual.html#AEN385, but by default the rule is that a key is considered validated if it meets both of the following two conditions:

1. It is **signed by enough valid keys**, meaning one of the following:
    1. You have signed it personally
    2. It has been signed by one **fully trusted** key
    3. It has been signed by three **marginally trusted** keys
2. The **path** of signed keys leading from it back to your own key is **five steps or shorter**

As for the trust levels:

Unknown	this is the trust level of newly imported keys – it is the default.
Undefined	this is a way to mark a key as “to be reviewed” to assign the right level of trust later on
Marginal	this kind of trust is typically for people or entities you guess that behave right, but you do not directly know. A key marginally trusted is automatically **shifted to trusted only if it signed by the key of at least three entities whom keys have been trusted by you**

Full	you are sure that the owner of these keys carefully signs the keys of other entities. Be wary that this means that any other key that gets signed by his key get automatically trusted
Ultimate	this is the highest level – you blindly trust who uses this key – you can consider this trust level suitable only for your own keys
Never	it provides a way to never trust an entity, even if it has been (or will be) trusted by other entities you trust. This actually blocks the web of trust on this key

Since the first requisite of the web of trust are signed keys, to see this in action we need some public keys to sign. In a real world scenario, we'd use key servers to receive and send public keys, but since we are in a lab, we rely on exports to files.

Let's begin by exporting the public keys of every user.

```
sudo -u bar gpg --armor --export bar@carcano.local > /tmp/bar-gpg.pub
sudo -u baz gpg --armor --export baz@carcano.local > /tmp/baz-gpg.pub
sudo -u qux gpg --armor --export qux@carcano.local > /tmp/qux-gpg.pub
sudo -u fred gpg --armor --export fred@carcano.local > /tmp/fred-gpg.pub
```

and importing the keys of the users we want to trust: in order to avoid trivial examples, in this lab we are going to trust the keys of bar, baz and qux users as "Marginally Trusted": this way, after having each of them signing fred's key we'll see the web of trust in action, automatically guessing the trust level of the fred's key as "Fully Trusted".

TRUSTING SOMEONE ELSE'S KEY

In our lab we are working as "foo" user, so let's switch to it:

```
sudo su - foo
```

we start by importing the keys of "*bar*", "*baz*" and "*qux*" users:

```
gpg --import /tmp/bar-gpg.pub
gpg --import /tmp/baz-gpg.pub
gpg --import /tmp/qux-gpg.pub
```

now let's trust as "*marginal*" bar's key:

```
gpg --pinentry-mode loopback --edit-key '<bar@carcano.local>' trust
```

the output of the interactive menu is as follows:

```
pub  rsa2048/1A44B621829F5714
     created: 2022-07-21  expires: never       usage: SC
     trust: unknown       validity: full
sub  rsa2048/60C80A0AD439C4A4
```

```
     created: 2022-07-21  expired: 2022-07-22  usage: E
sub  rsa2048/3CF0DB6D5B7B5331
     created: 2022-07-21  expires: 2023-07-21  usage: S
[  full  ] (1). Bar User (Bar User Primary Key) <bar@carcano.local>

Please decide how far you trust this user to correctly verify other users' keys
(by looking at passports, checking fingerprints from different sources, etc.)

  1 = I don't know or won't say
  2 = I do NOT trust
  3 = I trust marginally
  4 = I trust fully
  5 = I trust ultimately
  m = back to the main menu

Your decision? 3
```

remember to save before exiting the GPG interactive menu:

```
gpg> save
```

let's repeat the step for **baz's key** (remember to save before exiting the GPG interactive menu!):

```
gpg --pinentry-mode loopback --edit-key '<baz@carcano.local>' trust
```

and for **qux's key** (remember to save before exiting the GPG interactive menu!):

```
gpg --pinentry-mode loopback --edit-key '<qux@carcano.local>' trust
```

exit the sudoed shell to the previous user:

```
exit
```

SIGNING A KEY

In our mocked up scenario, fred's key will be signed by both "*bar*", "*baz*" and "*qux*" users.

As I told you Certification capability is available only for Primary Key: although we are signing a key, we are doing it for certification purposes – we are certifying that the information stored in the key (Name, Surname and description) are actually true. By the way this is the most crucial point of the trust PGP relies on: do not underestimate this step. Never and ever sign keys of people you don't know, or that you cannot verify the identity with reliable means. In addition to that, pay attention that ***anybody can submit whatever he wants to keyservers****: besides identifying the person holding the key, you should clearly identify the key itself, for example by fingerprint.*

Let's start from "*bar*": we must add fred's public key to bar's public keyring:

```
sudo -u bar gpg --import /tmp/fred-gpg.pub
```

of course, in a real world scenario bar user would retrieve fred's key from keyservers.

Now let's make bar user to sign fred's key (remember to save before exiting the GPG interactive menu!):

```
sudo -u bar gpg --pinentry-mode loopback --edit-key '<fred@carcano.local>' sign
```

Mind that if the private key of the user (bar in this case) has been exported and removed from the keyring, you'll get the following error:

```
gpg: signing failed: No secret key
```

in this case you have to re-import bar's private key into bar's private GPG keyring first.

Type "save" to save the key to the keyring and exit the GPG interactive command prompt:

```
gpg> save
```

As for now **the generated signature is only in bar's keyring**: bar must now export fred's public key, so that other users can see his signature:

```
sudo -u bar gpg --armor --export fred@carcano.local > /tmp/fred-gpg.pub
```

of course in a real world bar user would send fred's public key to key servers.

Now it's baz's turn - let's import the updated fred's key:

```
sudo -u baz gpg --import /tmp/fred-gpg.pub
```

then baz signs fred's key (remember to save before exiting the GPG interactive menu!):

```
sudo -u baz gpg --pinentry-mode loopback --edit-key '<fred@carcano.local>' sign
```

and eventually baz exports a new version of fred's key with the signature he added:

```
sudo -u baz gpg --armor --export fred@carcano.local > /tmp/fred-gpg.pub
```

Finally it has come qux's turn: import fred's public key into his public keyring:

```
sudo -u qux gpg --import /tmp/fred-gpg.pub
```

then qux signs fred's key (remember to save before exiting the GPG interactive menu!):

```
sudo -u qux gpg --pinentry-mode loopback --edit-key '<fred@carcano.local>' sign
```

and eventually he exports it:

```
sudo -u qux gpg --armor --export fred@carcano.local > /tmp/fred-gpg.pub
```

LISTING SIGNATURES ON A KEY

In our lab we work as "*foo*" user, so let's switch to it:

```
sudo su - foo
```

let's import (or of course retrieve from key server) the key we want to check the signatures:

```
gpg --import /tmp/fred-gpg.pub
```

now let's list the signatures:

```
gpg --list-sigs '<fred@carcano.local>'
```

the output is as follows:

```
pub   rsa2048/AB3E2E9F213CC658 2022-07-25 [SC]
      Key fingerprint = 0559 862A C951 CFB8 317C  8A7B AB3E 2E9F 213C C658
uid                 [  full  ] Fred User (Fred User Primary Key) <fred@carcano.local>
sig 3        AB3E2E9F213CC658 2022-07-25  Fred User (Fred User Primary Key) <fred@carcan
o.local>
sig          1A44B621829F5714 2022-07-25  Bar User (Bar User Primary Key) <bar@carcano.lo
cal>
sig          7404BF5387181611 2022-07-25  Baz User (Baz User Primary Key) <baz@carcano.lo
cal>
sig          1317D81E1CE2B6B3 2022-07-25  Qux User (Qux User Primary Key) <qux@carcano.lo
cal>
```

AUTOMATIC ASSIGNMENT OF TRUST LEVEL

As previously told, GPG is able to automatically guess trust level for public keys.

Let's have a look to the available public keys:

```
gpg --list-keys
```

the output is as follows:

```
/home/foo/.gnupg/pubring.kbx
----------------------------
pub   nistp256/7B221C7A2ACA4DE9 2022-07-21 [SC]
      Key fingerprint = 56AE 9AFF A54F A111 B08C  F0C4 7B22 1C7A 2ACA 4DE9
uid                 [ultimate] Foo User (Foo User Primary Key) <foo@carcano.local>
sub   nistp256/F43E205F0A259AC7 2022-07-21 [E] [expires: 2023-07-21]
sub   nistp256/4401D4D107578972 2022-07-21 [S] [expires: 2023-07-21]
```

```
sub   nistp256/F9C6C45EB9650672 2022-07-21 [SA] [expires: 2023-07-21]

pub   rsa2048/AB3E2E9F213CC658 2022-07-25 [SC]
      Key fingerprint = 0559 862A C951 CFB8 317C  8A7B AB3E 2E9F 213C C658
uid                 [ unknown] Fred User (Fred User Primary Key) <fred@carcano.local>

pub   rsa2048/1A44B621829F5714 2022-07-21 [SC]
      Key fingerprint = 1414 EDCC 5BBB C879 5662  4F38 1A44 B621 829F 5714
uid                 [ unknown] Bar User (Bar User Primary Key) <bar@carcano.local>
sub   rsa2048/3CF0DB6D5B7B5331 2022-07-21 [S] [expires: 2023-07-21]

pub   rsa2048/7404BF5387181611 2022-07-21 [SC]
      Key fingerprint = 42C1 515B D21D 96B0 6479  D914 7404 BF53 8718 1611
uid                 [ unknown] Baz User (Baz User Primary Key) <baz@carcano.local>
sub   rsa2048/DE8B6BB533181C7E 2022-07-25 [S] [expires: 2023-07-25]

pub   rsa2048/1317D81E1CE2B6B3 2022-07-25 [SC]
      Key fingerprint = 4086 6F42 0748 D33D 438D  3600 1317 D81E 1CE2 B6B3
uid                 [ unknown] Qux User (Qux User Primary Key) <qux@carcano.local>
```

as you see, despite we previously set "Marginal" trust level to "bar", "baz" and "qux" user's keys, they are all listed with "*unknown trust*".

Let's **check the contents of the trustdb**:

```
gpg --export-ownertrust
```

the output is:

```
g56AE9AFFA54FA111B08CF0C47B221C7A2ACA4DE9:6:
1414EDCC5BBBC87956624F381A44B621829F5714:4:
42C1515BD21D96B06479D9147404BF5387181611:4:
40866F420748D33D438D36001317D81E1CE2B6B3:4:t
```

The meaning of the number in the second column is as follows:

2	I don't know or won't say
3	I do NOT trust
4	I trust marginally
5	I trust fully
6	I trust ultimately

so the "*4*" in the second column means "*Marginal Trust* ", which is exactly what we expected.

The problem is that we are still missing another requirement for the web of trust, ... besides being trusted, bar, baz and qux keys must also be signed by the user, or they will not be considered "safe".

So let's sign bar's key:

```
gpg --pinentry-mode loopback --edit-key '<bar@carcano.local>' sign
```

then baz's key:

```
gpg --pinentry-mode loopback --edit-key '<baz@carcano.local>' sign
```

and finally qux's key:

```
gpg --pinentry-mode loopback --edit-key '<qux@carcano.local>' sign
```

let's check the trust level of the keys now:

```
gpg --list-keys
```

the output is as follows:

```
/home/foo/.gnupg/pubring.kbx
----------------------------
pub   nistp256/7B221C7A2ACA4DE9 2022-07-21 [SC]
      Key fingerprint = 56AE 9AFF A54F A111 B08C  F0C4 7B22 1C7A 2ACA 4DE9
uid                 [ultimate] Foo User (Foo User Primary Key) <foo@carcano.local>
sub   nistp256/F43E205F0A259AC7 2022-07-21 [E] [expires: 2023-07-21]
sub   nistp256/4401D4D107578972 2022-07-21 [S] [expires: 2023-07-21]
sub   nistp256/F9C6C45EB9650672 2022-07-21 [SA] [expires: 2023-07-21]

pub   rsa2048/AB3E2E9F213CC658 2022-07-25 [SC]
      Key fingerprint = 0559 862A C951 CFB8 317C  8A7B AB3E 2E9F 213C C658
uid                 [  full  ] Fred User (Fred User Primary Key) <fred@carcano.local>

pub   rsa2048/7404BF5387181611 2022-07-21 [SC]
      Key fingerprint = 42C1 515B D21D 96B0 6479  D914 7404 BF53 8718 1611
uid                 [  full  ] Baz User (Baz User Primary Key) <baz@carcano.local>
sub   rsa2048/DE8B6BB533181C7E 2022-07-25 [S] [expires: 2023-07-25]

pub   rsa2048/1317D81E1CE2B6B3 2022-07-25 [SC]
      Key fingerprint = 4086 6F42 0748 D33D 438D  3600 1317 D81E 1CE2 B6B3
uid                 [  full  ] Qux User (Qux User Primary Key) <qux@carcano.local>

pub   rsa2048/1A44B621829F5714 2022-07-21 [SC]
      Key fingerprint = 1414 EDCC 5BBB C879 5662  4F38 1A44 B621 829F 5714
uid                 [  full  ] Bar User (Bar User Primary Key) <bar@carcano.local>
sub   rsa2048/3CF0DB6D5B7B5331 2022-07-21 [S] [expires: 2023-07-21]
```

so signing the key

- dynamically raised the "*guessed*" trust level from "*Marginal*" to full for bar, baz an qux
- automatically guessed the "*Full*" trust level for fred's key, since it is signed by three or more marginally trusted users

OPERATING WITH GPG

Now that we have a lab with all the necessary users and keys to play with, we can begin doing

something.

SIGNING DOCUMENTS

The most straightforward GPG application is signing - there are actually three ways to sign a document using GPG:

- GPG signing
- GPG cleartext signing
- GPG Detached signing

The first two methods, since they embed the signature, actually generate a file with both the contents of the document and its signature. The last one instead leaves the document unchanged and generates a signature file: for this reason the last method is suitable for the purpose of signing not only documents, but also software packages too.

GPG SIGNING

This is the first method - in this example we are working as the "*baz*" user:

```
sudo su - baz
```

we just have to provide the **--sign** command switch as follows:

```
gpg --pinentry-mode loopback --out /tmp/README.gpg --sign /usr/share/doc/gnupg2/READ
```

let's inspect the kind of contents of the generated file (README.gpg)

```
file ~/README.gpg
```

as expected, the outcome is a data document that embeds the GPG signature.

```
README.gpg: data
```

exit the sudoed shell and become the "*foo*" user:

```
exit
sudo su - foo
```

we can now verify the signature of the signed document:

```
gpg --verify /tmp/README.gpg
```

the outcome is as follows:

```
gpg: Signature made Wed 27 Jul 2022 01:01:27 PM UTC
gpg: using RSA key 76006E9AEFC950E44EC1D539DE8B6BB533181C7E
```

```
gpg: checking the trustdb
gpg: marginals needed: 3 completes needed: 1 trust model: pgp
gpg: depth: 0 valid: 1 signed: 2 trust: 0-, 0q, 0n, 0m, 0f, 1u
gpg: depth: 1 valid: 2 signed: 1 trust: 0-, 0q, 0n, 2m, 0f, 0u
gpg: Good signature from "Baz User (Baz User Primary Key) <baz@carcano.local>" [full
Primary key fingerprint: 42C1 515B D21D 96B0 6479 D914 7404 BF53 8718 1611
Subkey fingerprint: 7600 6E9A EFC9 50E4 4EC1 D539 DE8B 6BB5 3318 1C7E
```

we can then decipher the GPG signed document and extract the original one by providing both the **--decrypt** and **--output** command line options:

```
gpg --output ~/README --decrypt /tmp/README.gpg
```

the output is as follows:

```
gpg: Signature made Wed 27 Jul 2022 01:01:27 PM UTC
gpg: using RSA key 76006E9AEFC950E44EC1D539DE8B6BB533181C7E
gpg: Good signature from "Baz User (Baz User Primary Key) <baz@carcano.local>" [full
Primary key fingerprint: 42C1 515B D21D 96B0 6479 D914 7404 BF53 8718 1611
Subkey fingerprint: 7600 6E9A EFC9 50E4 4EC1 D539 DE8B 6BB5 3318 1C7E
```

as you see, before extracting the original document gpg anyway checks the signature.

Exit the sudoed shell.

```
exit
```

CLEARTEXT SIGING

This is the second method - in this example we are working as the "*baz*" user:

```
sudo su - baz
```

this time we just provide the **--clearsign** command switch as follows:

```
gpg --pinentry-mode loopback --out /tmp/README.asc --clearsign /usr/share/doc/gnupg2
```

let's see the contents of the "*/tmp/README.asc*" file:

```
head -n 8 /tmp/README.asc;echo "...";tail -n 7 /tmp/README.asc
```

the output is just a snippet of the first 8 lines and last 7 lines of the file:

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA512
```

```
                      The GNU Privacy Guard 2
                     ========================
                            Version 2.2

          Copyright 1997-2019 Werner Koch
...
-----BEGIN PGP SIGNATURE-----

iHUEARMKAB0WIQQhEEKwGKiJO1i8Bur5xsReuWUGcgUCYt+aQwAKCRD5xsReuWUG
cpiwAQCkNil3YdqD4mkeenpmX8fz55HR54jzRXQrwZb54ZIVAgEAsmqrwLp+Xbfh
kl0CNyBJ4macXEXevfjc6uSaNeYcTSU=
=QzPo
-----END PGP SIGNATURE-----
```

as you see the file contains

- a **cleartext header** ("*-----BEGIN PGP SIGNED MESSAGE----- ...*") that claims that the document is PGP signed
- a **cleartext footer**, ("*-----BEGIN PGP SIGNATURE-----*","*-----END PGP SIGNATURE-----*") with the actual signature

so, as suggested by the name of the command line switch we provided, this time we generated a document with a cleartext signature.

Exit the sudoed shell and become the "*foo*" user:

```
exit
sudo su - foo
```

same way as we already did, let's verify the signed document:

```
gpg --verify /tmp/README.asc
```

the outcome is as follows:

```
gpg: Signature made Wed 27 Jul 2022 01:23:11 PM UTC
gpg: using RSA key 76006E9AEFC950E44EC1D539DE8B6BB533181C7E
gpg: Good signature from "Baz User (Baz User Primary Key) <baz@carcano.local>" [full
Primary key fingerprint: 42C1 515B D21D 96B0 6479 D914 7404 BF53 8718 1611
Subkey fingerprint: 7600 6E9A EFC9 50E4 4EC1 D539 DE8B 6BB5 3318 1C7E
```

we can then decipher the GPG signed document and extract the original one by providing both the **--decrypt** and **--output** command line:

```
gpg --output ~/README --decrypt /tmp/README.asc
```

the output is as follows:

```
gpg: Signature made Wed 27 Jul 2022 01:23:11 PM UTC
gpg: using RSA key 76006E9AEFC950E44EC1D539DE8B6BB533181C7E
gpg: Good signature from "Baz User (Baz User Primary Key) <baz@carcano.local>" [full
Primary key fingerprint: 42C1 515B D21D 96B0 6479 D914 7404 BF53 8718 1611
Subkey fingerprint: 7600 6E9A EFC9 50E4 4EC1 D539 DE8B 6BB5 3318 1C7E
```

as you see, before extracting the original document gpg anyway checks the signature.

Exit the sudoed shell.

```
exit
```

DETACHED SIGNATURE

This is the third and last method: since it is broadly used to sign software, I describe it in the next paragraph "*Signing Software*".

SIGNING SOFTWARE

GPG is broadly used to generate signature files of software packages.

In this example we are working as the "*baz*" user:

```
sudo su - baz
```

VERIFYING A DETACHED SIGNATURE

To provide a real life example, we download Gitea - a GIT implementation with a very nice WebUI - along with the file containing the **GPG signature** of the software package.

```
curl -o ~/gitea-1.16.9-darwin-10.12-amd64 https://dl.gitea.io/gitea/1.16.9/gitea-1.1
curl -o ~/gitea-1.16.9-darwin-10.12-amd64.asc https://dl.gitea.io/gitea/1.16.9/gitea
```

The very first thing to do is gather some information on the key that has been used to generate the signature file:

```
gpg --list-packets < ~/gitea-1.16.9-darwin-10.12-amd64.asc
```

the output is as follows:

```
# off=0 ctb=89 tag=2 hlen=3 plen=563
:signature packet: algo 1, keyid 5FC346329753F4B0
version 4, created 1657658277, md5len 0, sigclass 0x00
digest algo 8, begin of digest 81 ad
hashed subpkt 33 len 21 (issuer fpr v4 CC64B1DB67ABBEECAB24B6455FC346329753F4B0)
hashed subpkt 2 len 4 (sig created 2022-07-12)
```

```
subpkt 16 len 8 (issuer key ID 5FC346329753F4B0)
data: [4096 bits]
```

now that we know the fingerprint of the key (**CC64B1DB67ABBEECAB24B6455FC346329753F4B0**), we retrieve the public key used to generate the signature from the keyservers:

```
gpg --keyserver keys.openpgp.org --recv CC64B1DB67ABBEECAB24B6455FC346329753F4B0
```

since we don't have other keys that let us guess a trust for this key (see "*The Web Of Trust*" for more information on this topic), we must trust it manually.

This actually involves to **trust the key** - (set it as fully trusted):

```
gpg --pinentry-mode loopback --edit-key '<teabot@gitea.io>' trust
```

and to **sign the key**:

```
gpg --pinentry-mode loopback --edit-key '<teabot@gitea.io>' sign
```

we are finally ready to **check if the downloaded file matches its signature**:

```
gpg --verify ~/gitea-1.16.9-darwin-10.12-amd64.asc ~/gitea-1.16.9-darwin-10.12-amd64
```

the output is as follows:

```
gpg: Signature made Tue 12 Jul 2022 08:37:57 PM UTC
gpg: using RSA key CC64B1DB67ABBEECAB24B6455FC346329753F4B0
gpg: Good signature from "Teabot <teabot@gitea.io>" [full]
Primary key fingerprint: 7C9E 6815 2594 6888 62D6 2AF6 2D9A E806 EC15 92E2
Subkey fingerprint: CC64 B1DB 67AB BEEC AB24 B645 5FC3 4632 9753 F4B0
```

You may think that we took enough care into the verification of the software, ... ***but we haven't*** *. Beware that since* ***anybody can submit whatever he wants to keyservers, you risk downloading a rogue key. When keyservers are used,*** *a thorough verification must involve also gathering the fingerprint of the key from a secure source and checking the fingerprint of the downloaded key****. You have been warned!***

For the sake of completeness, in the event of a **mismatched signature**, the output would have been:

```
gpg: Signature made Tue 12 Jul 2022 08:37:57 PM UTC
gpg: using RSA key CC64B1DB67ABBEECAB24B6455FC346329753F4B0
gpg: BAD signature from "Teabot <teabot@gitea.io>" [full]
```

GENERATING A DETACHED SIGNATURE

In this example we are still working as the "*baz*" user. If you are not that user, switch to it:

```
sudo su - baz
```

Let's copy the "*echo*" command and pretend that we built a new amazing fancy software:

```
cp /usr/bin/echo /tmp/myfancysoftware
```

now we want to GPG sign our fancy software binary:

```
gpg --pinentry-mode loopback --armor --output /tmp/myfancysoftware.asc --detach-sig
```

let's see the contents of the signature file:

```
cat /tmp/myfancysoftware.asc
```

the output is as follows:

```
-----BEGIN PGP SIGNATURE-----

iQEzBAABCAAdFiEEdgBumu/JUOROwdU53otrtTMYHH4FAmLhQngACgkQ3otrtTMY
HH7Lbwf9HIDRSXUPjxDUEIk9xlbJGtipyNoq3OBnv/Rg5C5LQfKjfsud1SL5BTfK
BmSNJ5sLsmJOSZ6NwupCvMnLMzlXr6vlQvdm7bwW8Nt+xdKOPxpfgrDO4xtfVabS
4dLfhd9inflZg2mtetcBbpIK+pOtK3H9lXI2FtB8cGgMu5DVPXY7KDUrSCdA5b3A
iK62+wmgeGok+bXsoY2WO6bpO5LUA8WE6wy8/pggHKltfrXrDiXu9YOdSH0zm6Rp
sipKhsUR3Z5iJQ9GVzNIEY6wA9Jvi/lHdYbgyj9vliIf6AVdnqQHIoYN7lxY8tNr
/l28QC82649pgxJ755wy1/rkmKXwFg==
=F9k8
-----END PGP SIGNATURE-----
```

now let's pretend that the "*foo*" user want to install myfancysoftware: just exit the sudoed shell and become the "*foo*" user:

```
exit
sudo su - foo
```

since foo user is cautious, he checks the signature first:

```
gpg --verify /tmp/myfancysoftware.asc /tmp/myfancysoftware
```

the output is as follows:

```
gpg: Signature made Wed 27 Jul 2022 01:49:44 PM UTC
gpg: using RSA key 76006E9AEFC950E44EC1D539DE8B6BB533181C7E
gpg: Good signature from "Baz User (Baz User Primary Key) <baz@carcano.local>" [full]
Primary key fingerprint: 42C1 515B D21D 96B0 6479 D914 7404 BF53 8718 1611
Subkey fingerprint: 7600 6E9A EFC9 50E4 4EC1 D539 DE8B 6BB5 3318 1C7E
```

So the signature is good.

Exit the sudoed shell:

```
exit
```

GPG SIGNED RPM PACKAGES

RPM is a sophisticated archive format that does not only pack a set of files and directories within a package file, but can also run scripts and evaluate conditionals. It is made by putting a header structure on top of a CPIO archive. The package itself has four sections: the second one contains the GPG signature to verify the integrity of the package.

We can query rpm to get information about the signature of the already installed RPM package:

```
rpm -q -vv --queryformat '%{siggpg:armor}' basesystem
```

the output is as follows:

```
ufdio: 1 reads, 17154 total bytes in 0.000005 secs
D: loading keyring from pubkeys in /var/lib/rpm/pubkeys/*.key
D: couldn't find any keys in /var/lib/rpm/pubkeys/*.key
D: loading keyring from rpmdb
D: serialize failed, using private dbenv
D: opening db environment /var/lib/rpm cdb:private:0x401
D: opening db index /var/lib/rpm/Packages 0x400 mode=0x0
D: locked db index /var/lib/rpm/Packages
D: opening db index /var/lib/rpm/Name 0x400 mode=0x0
D: read h# 395
Header SHA1 digest: OK
D: added key gpg-pubkey-6d745a60-60287f36 to keyring
D: read h# 397
Header SHA1 digest: OK
D: added key gpg-pubkey-2f86d6a1-5cf7cefb to keyring
D: read h# 514
Header SHA1 digest: OK
D: added key gpg-pubkey-442df0f8-608c8351 to keyring
D: Using legacy gpg-pubkey(s) from rpmdb
D: read h# 16
Header V4 RSA/SHA256 Signature, key ID 6d745a60: OK
Header SHA256 digest: OK
Header SHA1 digest: OK
(none)D: closed db index /var/lib/rpm/Packages
D: closed db index /var/lib/rpm/Name
D: closed db environment /var/lib/rpm
D: Exit status: 0
```

RPM verifies the signature of the packages using its own GPG keyring.

You can add GPG keys to RPM keyrings by saving the key file into the **/etc/pki/rpm-gpg** directory.

For example, we can download the GPG file used to sign PostgreSQL14 RPM packages as follows:

```
curl -o /etc/pki/rpm-gpg/RPM-GPG-KEY-PGDG https://download.postgresql.org/pub/repos/
```

once saved, the **file must also be imported** as follows:

```
sudo rpm --import /etc/pki/rpm-gpg/RPM-GPG-KEY-PGDG
```

we can **list the GPG keys available in the RPM keyring** as follows:

```
rpm -q --queryformat "%{SUMMARY}\n" gpg-pubkey
```

if everything worked properly, the new key must be among the list:

```
gpg(Release Engineering <infrastructure@rockylinux.org>)
gpg(Fedora EPEL (8) <epel@fedoraproject.org>)
gpg(PostgreSQL RPM Building Project <pgsql-pkg-yum@postgresql.org>)
```

Having the key imported in the RPM gpg keyring lets us **enable GPG check** before installing the downloaded package.

This can be achieved by adding the **gpgcheck** and **gpgkey** directives within each repository stanza in the specific "*.repo*" files into **/etc/yum.repos.d** directory.

This is an example snippet of how to set these directives:

```
gpgcheck=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-PGDG
```

You can of course GPG sign your own RPM packages.

The signing feature for the **rpm** command line utility is provided by the **rpm-sign** RPM package - let's install it as follows:

```
sudo dnf install -y rpm-sign
```

in the following example we are working as "*foo*" user, so become the "*foo*" user:

```
sudo su - foo
```

configure the RPM macros of the user.

Please note that since we can have several keys within our GPG keyring, we must **specify which GPG key must be used to sign RPMs**: in this example we are configuring it to use "*foo@carcano.local*" GPG key:

```
cat << \EOF >> ~/.rpmmacros
%_gpg_name <foo@carcano.local>
EOF
```

although this is optional, we can even specify the **GPG command we use to sign** along with its options - here I can specify the "*--pinentry-mode loopback*" since I'm using Red Hat 8 / CentOS 8:

```
cat << \EOF >> ~/.rpmmacros
%__gpg_sign_cmd %{__gpg} gpg --force-v3-sigs --no-armor --pinentry-mode loopback --r
EOF
```

Now everything is properly set up to **sign RPM packages**.

Just to provide an example, the rpm command to sign the foo-0.1-1.el8.x86_64.rpm RPM packages is:

```
rpm --addsign ~/rpmbuild/RPMS/x86_64/foo-0.1-1.el8.x86_64.rpm
```

after typing the password to unlock the secret key for the signature, the RPM packages get signed.

ENCRYPTING AND DECRYPTING

The other straightforward purpose of GPG is file encryption. When encrypting, OpenPGP:

- first compresses the data (if file size is enough to allow compression)
- then applies a **symmetric algorithm** with a **random session key** generated on the fly to encrypt the compressed data

The **session key** itself is then encrypted by asymmetric algorithms using the public key of each one of the specified recipients.

As an example, let's make a copy the /usr/share/doc/gnupg2/README file:

```
cp /usr/share/doc/gnupg2/README ~/README
```

then let's encrypt the copy:

```
gpg -e --armor -r bar@carcano.local -r foo@carcano.local --out /tmp/README.asc ~/REA
```

Remember to always include your own public key among the various public keys used in the encryption statement or you won't be able to decrypt the file if you shred the original one.

Since **gpg does not remove the original file**, it's up to you to properly shred it, for example:

```
shred -vz -n 7 ~/README
rm -f ~/README
```

let's become the "*bar*" user:

```
exit
```

```
sudo su - bar
```

since bar was among the recipients, we must be able to decrypt the file:

```
gpg --pinentry-mode loopback --out ~/README -d /tmp/README.asc
```

the output is as follows:

```
gpg: encrypted with ECDH key, ID F43E205F0A259AC7
gpg: encrypted with 2048-bit RSA key, ID 65010FD4CE7E3E49, created 2022-07-26
"Bar User (Bar User Primary Key) <bar@carcano.local>"
```

Beware the keyID of each recipient is stored within the encrypted file. As you see the keyID is shown in the decryption process indeed. You must be very careful about this disclosed sensitive information: if you are using GPG in high risk situations (there are activists that risk their own life), mind that the evil ones can know who can decrypt it, and use "hard" methods to force him to decrypt it.

Exit the sudoed shell:

```
exit
```

EXPLOITING THE VIM GPG PLUGIN

Since we previously installed it, we can now exploit the vim GPG plugin.

Become the "foo" user:

```
sudo su - foo
```

Let's begin from editing the /tmp/README.asc GPG encrypted file we created a while ago:

```
vim /tmp/README.asc
```

As you can see the plugin seamlessly decrypts the file. You can modify it at wish: when you save it the plugin seamlessly re-encrypts it.

Now let pretend foo is a system administrator that works in the same team with bar: we can set to encrypt for both of them by default simply by adding both of the to the GPGDefaultRecipients list in the ~/.vimrc file:

```
" Set the default recipient list
let g:GPGDefaultRecipients=["foo@carcano.local", "bar@carcano.local"]
```

To try it, let's create a new file - please note that in order to have vim correctly guess we want to create a GPG encrypted file, the new file must be an ".asc" file.

So:

```
vim newfile.asc
```

both foo and bar user are now listed in the recipients list as shown in the following snippet:

```
GPG: ----------------------------------------------------------------------
GPG: Please edit the list of recipients, one recipient per line.
GPG: Unknown recipients have a prepended "!".
GPG: Lines beginning with "GPG:" are removed automatically.
GPG: Data after recipients between and including "(" and ")" is ignored.
GPG: Closing this buffer commits changes.
GPG: ----------------------------------------------------------------------
Foo User (Foo User Primary Key) <foo@carcano.local> (ID: 0x7B221C7A2ACA4DE9 created at We
d 27 Jul 2022 07:19:11 AM UTC
Bar User (Bar User Primary Key) <bar@carcano.local> (ID: 0x1A44B621829F5714 created at Th
u 21 Jul 2022 09:17:15 AM UTC
~
~
~
~
~
~
~
~
GPGRecipients_1
```

If you wish you can of course add other recipients to this list - you must of course already have imported their public key in the GPG public keyring before.

When done, simply type **:q** to exit from the GPG settings of this file and switch to the vim edit mode, then just work as you are usually doing with vim.

If you want to **see the list of recipient**, just switch to command mode and type:

```
:GPGViewRecipients
```

If necessary, you can of course **modify the list of recipients** by switching to command mode by typing:

```
:GPGEditRecipients
```

exit the sudoed shell:

```
exit
```

HIDDEN RECIPIENTS

As we saw the encrypted file can contain **sensitive** information about the key or keys necessary to decrypt the file.

For example, let's become "qux" user:

```
sudo su - qux
```

and try to decrypt the file:

```
gpg --pinentry-mode loopback --out ~/README -d /tmp/README.asc
```

the outcome is the following message:

```
gpg: encrypted with ECDH key, ID F43E205F0A259AC7
gpg: encrypted with RSA key, ID 65010FD4CE7E3E49
gpg: decryption failed: No secret key
```

So, although qux user is not able to decrypt the file, he knows the ID of the keys that are necessary to decrypt it. This can pose serious security risks to the individuals that own those keys, if qux is evil.

For this reason, when dealing with very risky situations, it's better to specify the recipients as hidden:

Let's switch back to the "*foo*" user:

```
exit
sudo su - foo
```

and encrypt the file again, but this time using the **-R** (capital letter) option to list the recipients as hidden:

```
gpg -e --armor -R bar@carcano.local -R foo@carcano.local --out /tmp/README.asc /usr/
```

now let's switch to the "*qux*" user again:

```
exit
sudo su - qux
```

and try to decrypt the file again:

```
gpg -d /tmp/README.asc
```

this time the output is:

```
gpg: selecting card failed: No such device
gpg: selecting card failed: No such device
gpg: encrypted with ECDH key, ID 0000000000000000
gpg: encrypted with RSA key, ID 0000000000000000
gpg: decryption failed: No secret key
```

So now we are not leaking any kind of information.

Just exit the sudoed shell:

```
exit
```

SIGNING AND ENCRYPTING A DOCUMENT

If you wish, you can of course both encrypt and sign a document.

Switch to the "*foo*" user:

```
sudo su - foo
```

in order to both encrypt and sign, you must provide both the **-e** and **--sign** command line switches.

For example, to encrypt and sign the /usr/share/doc/gnupg2/README so that both foo and bar users can use it, type:

```
gpg --pinentry-mode loopback -e --armor -R bar@carcano.local -R foo@carcano.local --
```

FOOTNOTES

Here it ends this tutorial on GPG. We thoroughly saw how to generate and manage Primary Keys and subkeys, what are the best practices and how to use them to sign and encrypt documents. We also learned how GPG can automatically figure out the trust level to assign to public keys using the Web Of Trust.