GnuPG

According to the official website (https://www.gnupg.org/):

> GnuPG is a complete and free implementation of the OpenPGP (http://openpgp.org/about/) standard as defined by RFC4880 (https://tools.ietf.org/html/rfc4880) (also known as PGP). GnuPG allows you to encrypt and sign your data and communication. It features a versatile key management system as well as access modules for all kinds of public key directories. GnuPG, also known as GPG, is a command line tool with features for easy integration with other applications. A wealth of frontend applications and libraries are available. Version 2 of GnuPG also provides support for S/MIME and Secure Shell (ssh).

Related articles

- pacman/Package signing
- Disk encryption
- List of applications/Security#En signing, steganography

Contents

Installation

Install the `gnupg (https://www.archlinux.org/packages/?name=gnupg)` package.

This will also install `pinentry (https://www.archlinux.org/packages/?name=pinentry)`, a collection of simple PIN or passphrase entry dialogs which GnuPG uses for passphrase entry. Which *pinentry* dialog is used is determined by the symbolic link `/usr/bin/pinentry`, which by default points to `/usr/bin/pinentry-gtk-2`.

If you want to use a graphical frontend or program that integrates with GnuPG, see **List of applications/Security#Encryption, signing, steganography**.

Configuration

Directory location

`$GNUPGHOME` is used by GnuPG to point to the directory where its configuration files are stored. By default `$GNUPGHOME` is not set and your `$HOME` is used instead; thus, you will find a `~/.gnupg` directory right after installation.

To change the default location, either run gpg this way `$ gpg --homedir path/to/file` or set `GNUPGHOME` in one of your regular **startup files**.

Configuration files

The default configuration files are `~/.gnupg/gpg.conf` and `~/.gnupg/dirmngr.conf`.

By default, the gnupg directory has its **permissions** set to `700` and the files it contains have their permissions set to `600`. Only the owner of the directory has permission to read, write, and access the files. This is for security purposes and should not be changed. In case this directory or any file inside it does not follow this security measure, you will get warnings about unsafe file and home directory permissions.

Append to these files any long options you want. Do not write the two dashes, but simply the name of the option and required arguments. You will find skeleton files in `/usr/share/gnupg`. These files are copied to `~/.gnupg` the first time gpg is run if they do not exist there. Other examples are found in **#See also**.

Additionally, **pacman** uses a different set of configuration files for package signature verification. See **Pacman/Package signing** for details.

Default options for new users

If you want to setup some default options for new users, put configuration files in `/etc/skel/.gnupg/`. When the new user is added in system, files from here will be copied to its GnuPG home directory. There is also a simple script called *addgnupghome* which you can use to create new GnuPG home directories for existing users:

```
# addgnupghome user1 user2
```

This will add the respective `/home/user1/.gnupg` and `/home/user2/.gnupg` and copy the files from the skeleton directory to it. Users with existing GnuPG home directory are simply skipped.

Usage

Note: Whenever a `user-id` is required in a command, it can be specified with your key ID, fingerprint, a part of your name or email address, etc. GnuPG is flexible on this.

Create key pair

Generate a key pair by typing in a terminal:

```
$ gpg --full-gen-key
```

Tip: Use the `--expert` option for getting alternative ciphers like **ECC**.

The command will prompt for answers to several questions. For general use most people will want:

- the RSA (sign only) and a RSA (encrypt only) key.
- a keysize of the default value (2048). A larger keysize of 4096 "gives us almost nothing, while costing us quite a lot"**[1] (https://www.gnupg.org/faq/gnupg-faq.html#no_default_of_rsa4096)**.
- an expiration date. A period of a year is good enough for the average user. This way even if access is lost to the keyring, it will allow others to know that it is no longer valid. Later, if necessary, the expiration date can be extended without having to re-issue a new key.
- your name and email address. You can add multiple identities to the same key later (*e.g.*, if you have multiple email addresses you want to associate with this key).
- *no* optional comment. Since the semantics of the comment field are **not well-defined (https://lists.gnupg.org/pipermail/gnupg-devel/2015-July/030150.html)**, it has limited value for identification.
- **a secure passphrase**.

Note: The name and email address you enter here will be seen by anybody who imports your key.

List keys

To list keys in your public key ring:

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

To list keys in your secret key ring:

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

Export your public key

gpg's main usage is to ensure confidentiality of exchanged messages via public-key cryptography. With it each user distributes the public key of their keyring, which can be be used by others to encrypt messages to the user. The private key must *always* be kept private, otherwise confidentiality is broken. See **w:Public-key cryptography** for examples about the message exchange.

So, in order for others to send encrypted messages to you, they need your public key.

To generate an ASCII version of a user's public key to file `public.key` (e.g. to distribute it by e-mail):

```
$ gpg --output public.key --armor --export user-id
```

Alternatively, or in addition, you can **#Use a keyserver** to share your key.

Tip: Add `--no-emit-version` to avoid printing the version number, or add the corresponding setting to your configuration file.

Import a public key

In order to encrypt messages to others, as well as verify their signatures, you need their public key. To import a public key with file name `public.key` to your public key ring:

```
$ gpg --import public.key
```

Alternatively, **#Use a keyserver** to find a public key.

Use a keyserver

You can register your key with a public PGP key server, so that others can retrieve your key without having to contact you directly:

```
$ gpg --send-keys user-id
```

Warning: Once a key has been submitted to a keyserver, it cannot be deleted from the server.**[2] (https://pgp.mit.edu/faq.html)**

To find out details of a key on the keyserver, without importing it, do:

```
$ gpg --search-keys user-id
```

To import a key from a key server:

```
$ gpg --recv-keys key-id
```

Warning:

- You should verify the authenticity of the retrieved public key by comparing its fingerprint with one that the owner published on an independent source(s) (e.g., contacting the person directly). See **Wikipedia:Public key fingerprint** for more information.
- Using a short ID may encounter collisions. All keys will be imported that have the short ID. To avoid this, use the full fingerprint or long key ID when receiving a key.**[3] (https://lkml.org/lkml/2016/8/15/445)**

Tip:

- Adding `keyserver-options auto-key-retrieve` to `gpg.conf` will automatically fetch keys from the key server as needed.
- An alternative key server is `pool.sks-keyservers.net` and can be specified with `keyserver` in `dirmngr.conf` ; see also **wikipedia:Key server (cryptographic)#Keyserver examples**.
- If your network blocks ports used for hkp/hkps, you may need to specify port 80, i.e. `pool.sks-keyservers.net:80`
- You can connect to the keyserver over **Tor** using `--use-tor` . See this **GnuPG blog post (https://gnupg.org/blog/20151224-gnupg-in-november-and-december.html)** for more information.
- You can connect to a keyserver using a proxy by setting the `http_proxy` environment variable and setting `honor-http-proxy` in `dirmngr.conf` . Alternatively, set `http-proxy` *`host[:port]`* in `dirmngr.conf` , overriding the `http_proxy` environment variable.
- If you wish to import a key ID to install a specific Arch Linux package, see **pacman/Package signing#Managing the keyring** and **Makepkg#Signature checking**.

Encrypt and decrypt

Asymmetric

You need to **#Import a public key** of a user before encrypting (options `--encrypt` or `-e`) a file or message to that recipient (options `--recipient` or `-r`). Additionally you need to **#Create key pair** if you have not already done so.

To encrypt a file with the name *doc*, use:

```
$ gpg --recipient user-id --encrypt doc
```

To decrypt (option `--decrypt` or `-d`) a file with the name *doc*.gpg encrypted with your public key, use:

```
$ gpg --output doc --decrypt doc.gpg
```

gpg will prompt you for your passphrase and then decrypt and write the data from *doc*.gpg to *doc*. If you omit the `-o` (`--output`) option, *gpg* will write the decrypted data to stdout.

Tip:

- Add `--armor` to encrypt a file using ASCII armor (suitable for copying and pasting a message in text format)
- Use `-R user-id` or `--hidden-recipient user-id` instead of `-r` to not put the recipient key IDs in the encrypted message. This helps to hide the receivers of the message and is a limited countermeasure against traffic analysis.
- Add `--no-emit-version` to avoid printing the version number, or add the corresponding setting to your configuration file.

- You can use gnupg to encrypt your sensitive documents by using your own user-id as recipient or by using the `--default-recipient-self` flag instead; however, you can only do this one file at a time, although you can always tarball various files and then encrypt the tarball. See also **Disk encryption#Available methods** if you want to encrypt directories or a whole file-system.

Symmetric

Symmetric encryption does not require the generation of a key pair and can be used to simply encrypt data with a passphrase. Simply use `--symmetric` or `-c` to perform symmetic encryption:

```
$ gpg -c doc
```

The following example:

- Encrypts *doc* with a symmetric cipher using a passphrase
- Uses the AES-256 cipher algorithm to encrypt the passphrase
- Uses the SHA-512 digest algorithm to mangle the passphrase
- Mangles the passphrase for 65536 iterations

```
$ gpg -c --s2k-cipher-algo AES256 --s2k-digest-algo SHA512 --s2k-count 65536 doc
```

To decrypt a symmetrically encrypted *doc.gpg* using a passphrase and output decrypted contents into the same directory as *doc* do:

```
$ gpg --output doc --decrypt doc.gpg
```

Key maintenance

Backup your private key

To backup your private key do the following:

```
$ gpg --export-secret-keys --armor <user-id> > privkey.asc
```

Note that *gpg* release 2.1 changed default behaviour so that the above command enforces a passphrase protection, even if you deliberately chose not to use one on key creation. This is because otherwise anyone who gains access to the above exported file would be able to encrypt and sign documents as if they were you *without* needing to know your passphrase.

Warning: The passphrase is usually the weakest link in protecting your secret key. Place the private key in a safe place on a different system/device, such as a locked container or encrypted drive. It is the only safety you have to regain control to your keyring in case of, for example, a drive failure, theft or worse.

To import the backup of your private key:

```
$ gpg --import privkey.asc
```

Edit your key

Running the `gpg --edit-key <user-id>` command will present a menu which enables you to do most of your key management related tasks.

Some useful commands in the edit key sub menu:

```
> passwd       # change the passphrase
> clean        # compact any user ID that is no longer usable (e.g revoked or expired)
> revkey       # revoke a key
> addkey       # add a subkey to this key
> expire       # change the key expiration time
```

Type `help` in the edit key sub menu for more commands.

Tip: If you have multiple email accounts you can add each one of them as an identity, using `adduid` command. You can then set your favourite one as `primary`.

Exporting subkey

If you plan to use the same key across multiple devices, you may want to strip out your master key and only keep the bare minimum encryption subkey on less secure systems.

First, find out which subkey you want to export.

```
$ gpg -K
```

Select only that subkey to export.

```
$ gpg -a --export-secret-subkeys [subkey id]! > /tmp/subkey.gpg
```

Warning: If you forget to add the !, all of your subkeys will be exported.

At this point you could stop, but it is most likely a good idea to change the passphrase as well. Import the key into a temporary folder.

```
$ gpg --homedir /tmp/gpg --import /tmp/subkey.gpg
$ gpg --homedir /tmp/gpg --edit-key <user-id>
> passwd
> save
$ gpg --homedir /tmp/gpg -a --export-secret-subkeys [subkey id]! > /tmp/subkey.altpass.gpg
```

Note: You will get a warning that the master key was not available and the password was not changed, but that can safely be ignored as the subkey password was.

At this point, you can now use `/tmp/subkey.altpass.gpg` on your other devices.

Rotating subkeys

Warning: **Never** delete your expired or revoked subkeys unless you have a good reason. Doing so will cause you to lose the ability to decrypt files encrypted with the old subkey. Please **only** delete expired or revoked keys from other users to clean your keyring.

If you have set your subkeys to expire after a set time, you can create new ones. Do this a few weeks in advance to allow others to update their keyring.

Tip: You do not need to create a new key simply because it is expired. You can extend the expiration date.

Create new subkey (repeat for both signing and encrypting key)

```
$ gpg --edit-key <user-id>
> addkey
```

And answer the following questions it asks (see **#Create key pair** for suggested settings).

Save changes

```
> save
```

Update it to a keyserver.

```
$ gpg --keyserver pgp.mit.edu --send-keys <user-id>
```

Tip: Revoking expired subkeys is unnecessary and arguably bad form. If you are constantly revoking keys, it may cause others to lack confidence in you.

Signatures

Signatures certify and timestamp documents. If the document is modified, verification of the signature will fail. Unlike encryption which uses public keys to encrypt a document, signatures are created with the user's private key. The recipient of a signed document then verifies the signature using the sender's public key.

Create a signature

Sign a file

To sign a file use the `--sign` or `-s` flag:

```
$ gpg --output doc.sig --sign doc
```

`doc.sig` contains both the compressed content of the original file `doc` and the signature in a binary format, but the file is not encrypted. However, you can combine signing with **encrypting**.

Clearsign a file or message

To sign a file without compressing it into binary format use:

```
$ gpg --output doc.sig --clearsign doc
```

Here both the content of the original file `doc` and the signature are stored in human-readable form in `doc.sig`.

Make a detached signature

To create a separate signature file to be distributed separately from the document or file itself, use the `--detach-sig` flag:

```
$ gpg --output doc.sig --detach-sig doc
```

Here the signature is stored in `doc.sig`, but the contents of `doc` are not stored in it. This method is often used in distributing software projects to allow users to verify that the program has not been modified by a third party.

Verify a signature

To verify a signature use the `--verify` flag:

```
$ gpg --verify doc.sig
```

where `doc.sig` is the signed file containing the signature you wish to verify.

If you are verifying a detached signature, both the signed data file and the signature file must be present when verifying. For example, to verify Arch Linux's latest iso you would do:

```
$ gpg --verify archlinux-version.iso.sig
```

where `archlinux-version.iso` must be located in the same directory.

You can also specify the signed data file with a second argument:

```
$ gpg --verify archlinux-version.iso.sig /path/to/archlinux-version.iso
```

If a file as been encrypted in addition to being signed, simply **decrypt** the file and its signature will also be verified.

gpg-agent

gpg-agent is mostly used as daemon to request and cache the password for the keychain. This is useful if GnuPG is used from an external program like a mail client. **gnupg (https://www.archlinux.org/packages/?name=gnupg)** comes with **systemd user** sockets which are enabled by default. These sockets are `gpg-agent.socket`, `gpg-agent-extra.socket`, `gpg-agent-browser.socket`, `gpg-agent-ssh.socket`, and `dirmngr.socket`.

- The main `gpg-agent.socket` is used by *gpg* to connect to the *gpg-agent* daemon.

- The intended use for the `gpg-agent-extra.socket` on a local system is to set up a Unix domain socket forwarding from a remote system. This enables to use *gpg* on the remote system without exposing the private keys to the remote system. See **gpg-agent(1) (https://jlk.fjfi.cvut.cz/arch/manpages/man/gpg-agent.1)** for details.
- The `gpg-agent-ssh.socket` can be used by **SSH** to cache **SSH keys** added by the *ssh-add* program. See **#SSH agent** for the necessary configuration.
- The `dirmngr.socket` starts a GnuPG daemon handling connections to keyservers.

Note: If you use non-default GnuPG **#Directory location**, you will need to **edit** all socket files to use the path in the socket directory that `gpgconf --create-socketdir` creates.

Configuration

gpg-agent can be configured via `~/.gnupg/gpg-agent.conf` file. The configuration options are listed in **gpg-agent(1) (https://jlk.fjfi.cvut.cz/arch/manpages/man/gpg-agent.1)**. For example you can change cache ttl for unused keys:

~/.gnupg/gpg-agent.conf

```
default-cache-ttl 3600
```

Tip: To cache your passphrase for the whole session, please run the following command:

```
$ /usr/lib/gnupg/gpg-preset-passphrase --preset XXXXXX
```

where XXXX is the keygrip. You can get its value when running `gpg --with-keygrip -K` . Passphrase will be stored until `gpg-agent` is restarted. If you set up `default-cache-ttl` value, it will take precedence.

Reload the agent

After changing the configuration, reload the agent using *gpg-connect-agent*:

```
$ gpg-connect-agent reloadagent /bye
```

The command should print `OK`.

However in some cases only the restart may not be sufficient, like when `keep-screen` has been added to the agent configuration. In this case you firstly need to kill the ongoing gpg-agent process and then you can restart it as was explained above.

pinentry

Finally, the agent needs to know how to ask the user for the password. This can be set in the gpg-agent configuration file.

The default uses a gtk dialog. There are other options - see `info pinentry`. To change the dialog implementation set `pinentry-program` configuration option:

~/.gnupg/gpg-agent.conf

```
# PIN entry program
# pinentry-program /usr/bin/pinentry-curses
# pinentry-program /usr/bin/pinentry-qt
# pinentry-program /usr/bin/pinentry-kwallet

pinentry-program /usr/bin/pinentry-gtk-2
```

Tip: For using `/usr/bin/pinentry-kwallet` you have to install the **kwalletcli (https://aur.archlinux.org/packages/kwalletcli/)**[AUR] package.

After making this change, reload the gpg-agent.

Unattended passphrase

Starting with GnuPG 2.1.0 the use of gpg-agent and pinentry is required, which may break backwards compatibility for passphrases piped in from STDIN using the `--passphrase-fd 0` commandline option. In order to have the same type of functionality as the older releases two things must be done:

First, edit the gpg-agent configuration to allow *loopback* pinentry mode:

```
~/.gnupg/gpg-agent.conf
```

```
allow-loopback-pinentry
```

Restart the gpg-agent process if it is running to let the change take effect.

Second, either the application needs to be updated to include a commandline parameter to use loopback mode like so:

```
$ gpg --pinentry-mode loopback ...
```

...or if this is not possible, add the option to the configuration:

```
~/.gnupg/gpg.conf
```

```
pinentry-mode loopback
```

Note: The upstream author indicates setting `pinentry-mode loopback` in `gpg.conf` may break other usage, using the commandline option should be preferred if at all possible. **[4] (https://bugs.g10code.com/gnupg/issue1772)**

SSH agent

gpg-agent has OpenSSH agent emulation. If you already use the GnuPG suite, you might consider using its agent to also cache your **SSH keys**. Additionally, some users may prefer the PIN entry dialog GnuPG agent provides as part of its passphrase management.

To start using GnuPG agent for your SSH keys, enable SSH support in the `~/.gnupg/gpg-agent.conf` file:

```
~/.gnupg/gpg-agent.conf
```

```
enable-ssh-support
```

Set SSH_AUTH_SOCK

Then set `SSH_AUTH_SOCK` so that SSH will use *gpg-agent* instead of *ssh-agent*. To make sure each process can find your *gpg-agent* instance regardless of e.g. the type of shell it is child of use **pam_env**.

```
~/.pam_environment
```

```
SSH_AGENT_PID   DEFAULT=
SSH_AUTH_SOCK   DEFAULT="${XDG_RUNTIME_DIR}/gnupg/S.gpg-agent.ssh"
```

Alternatively, depend on Bash. This works for non-standard socket locations as well:

~/.bashrc

```
# Set SSH to use gpg-agent
unset SSH_AGENT_PID
if [ "${gnupg_SSH_AUTH_SOCK_by:-0}" -ne $$ ]; then
  export SSH_AUTH_SOCK="$(gpgconf --list-dirs agent-ssh-socket)"
fi
```

Note:

- If you use non-default GnuPG **#Directory location**, run `gpgconf --create-socketdir` to create a socket directory under `/run/user/$UID/gnupg/`. Otherwise the socket will be placed in the GnuPG home directory.
- The test involving the `gnupg_SSH_AUTH_SOCK_by` variable is for the case where the agent is started as `gpg-agent --daemon /bin/sh`, in which case the shell inherits the `SSH_AUTH_SOCK` variable from the parent, *gpg-agent* **[5] (http://git.gnupg.org/cgi-bin/gitweb.cgi?p=gnupg.git;a=blob;f=agent/gpg-agent.c;hb=7bca3be65e510eda40572327b87922834ebe07eb#l1307)**.

Configure pinentry to use the correct TTY

Also set the GPG_TTY and refresh the TTY in case user has switched into an X session as stated in **gpg-agent(1) (https://jlk.fjfi.cvut.cz/arch/manpages/man/gpg-agent.1)**. For example:

~/.bashrc

```
# Set GPG TTY
export GPG_TTY=$(tty)

# Refresh gpg-agent tty in case user switches into an X session
gpg-connect-agent updatestartuptty /bye >/dev/null
```

Add SSH keys

Once *gpg-agent* is running you can use *ssh-add* to approve keys, following the same steps as for **ssh-agent**. The list of approved keys is stored in the `~/.gnupg/sshcontrol` file.

Once your key is approved, you will get a *pinentry* dialog every time your passphrase is needed. You can control passphrase caching in the `~/.gnupg/gpg-agent.conf` file. The following example would have *gpg-agent* cache your keys for 3 hours:

~/.gnupg/gpg-agent.conf

```
default-cache-ttl-ssh 10800
max-cache-ttl-ssh 10800
```

Smartcards

GnuPG uses *scdaemon* as an interface to your smartcard reader, please refer to the **man page** for details.

GnuPG only setups

Note: To allow scdaemon direct access to USB smarcard readers the optional dependency **libusb-compat (https://www.archlinux.org/packages/?name=libusb-compat)** have to be installed

If you do not plan to use other cards but those based on GnuPG, you should check the `reader-port` parameter in `~/.gnupg/scdaemon.conf`. The value '0' refers to the first available serial port reader and a value of '32768' (default) refers to the first USB reader.

GnuPG with pcscd (PCSC Lite)

Note: `pcsclite (https://www.archlinux.org/packages/?name=pcsclite)` and `ccid (https://www.archlinux.org/packages/?name=ccid)` have to be installed, and the contained **systemd** service `pcscd.service` has to be running, or the socket `pcscd.socket` has to be listening.

pcscd is a daemon which handles access to smartcard (SCard API). If GnuPG's scdaemon fails to connect the smartcard directly (e.g. by using its integrated CCID support), it will fallback and try to find a smartcard using the PCSC Lite driver.

Always use pcscd

If you are using any smartcard with an opensc driver (e.g.: ID cards from some countries) you should pay some attention to GnuPG configuration. Out of the box you might receive a message like this when using `gpg --card-status`

```
gpg: selecting openpgp failed: ec=6.108
```

By default, scdaemon will try to connect directly to the device. This connection will fail if the reader is being used by another process. For example: the pcscd daemon used by OpenSC. To cope with this situation we should use the same underlying driver as opensc so they can work well together. In order to point scdaemon to use pcscd you should remove `reader-port` from `~/.gnupg/scdaemon.conf`, specify the location to `libpcsclite.so` library and disable ccid so we make sure that we use pcscd:

~/.gnupg/scdaemon.conf

```
pcsc-driver /usr/lib/libpcsclite.so
card-timeout 5
disable-ccid
```

Please check `scdaemon(1)` (https://jlk.fjfi.cvut.cz/arch/manpages/man/scdaemon.1) if you do not use OpenSC.

Tips and tricks

Different algorithm

You may want to use stronger algorithms:

~/.gnupg/gpg.conf

```
...

personal-digest-preferences SHA512
cert-digest-algo SHA512
default-preference-list SHA512 SHA384 SHA256 SHA224 AES256 AES192 AES CAST5 ZLIB BZIP2 ZIP Uncompressed
personal-cipher-preferences TWOFISH CAMELLIA256 AES 3DES
```

In the latest version of GnuPG, the default algorithms used are SHA256 and AES, both of which are secure enough for most people. However, if you are using a version of GnuPG older than 2.1, or if you want an even higher level of security, then you should follow the above step.

Encrypt a password

It can be useful to encrypt some password, so it will not be written in clear on a configuration file. A good example is your email password.

First create a file with your password. You **need** to leave **one** empty line after the password, otherwise gpg will return an error message when evaluating the file.

Then run:

```
$ gpg -e -a -r <user-id> your_password_file
```

`-e` is for encrypt, `-a` for armor (ASCII output), `-r` for recipient user ID.

You will be left with a new `your_password_file.asc` file.

Tip: **pass** automates this process.

Revoking a key

Warning:

- Anybody having access to your revocation certificate can revoke your key, rendering it useless.
- Key revocation should only be performed if your key is compromised or lost, or you forget your passphrase.

Revocation certificates are automatically generated for newly generated keys, although one can be generated manually by the user later. These are located at `~/.gnupg/openpgp-revocs.d/`. The filename of the certificate is the fingerprint of the key it will revoke.

To revoke your key, simply import the revocation certificate:

```
$ gpg --import <fingerprint>.rev
```

Now update the keyserver:

```
$ gpg --keyserver subkeys.pgp.net --send-keys <userid>
```

Change trust model

By default GnuPG uses the **Web of Trust** as the trust model. You can change this to **Trust on first use** by adding `--trust-model=tofu` when adding a key or adding this option to your GnuPG configuration file. More details are in **this email to the GnuPG list (https://lists.gnupg.org/pipermail/gnupg-devel/2015-October/030341.html)**.

Hide all recipient id's

By default the recipient's key ID is in the encrypted message. This can be removed at encryption time for a recipient by using `hidden-recipient <user-id>`. To remove it for all recipients add `throw-keyids` to your configuration file. This helps to hide the receivers of the message and is a limited countermeasure against traffic analysis. (Using a little social engineering anyone who is able to decrypt the message can check whether one of the other recipients is the one he suspects.) On the receiving side, it may slow down the decryption process because all available secret keys must be tried (*e.g.* with `--try-secret-key <user-id>`).

Using caff for keysigning parties

To allow users to validate keys on the keyservers and in their keyrings (i.e. make sure they are from whom they claim to be), PGP/GPG uses the **Web of Trust**. Keysigning parties allow users to get together at a physical location to validate keys. The **Zimmermann-Sassaman** key-signing protocol is a way of making

these very effective. **Here (http://www.cryptnet.net/fdp/crypto/keysigning_party/en/keysigning_party.html)** you will find a how-to article.

For an easier process of signing keys and sending signatures to the owners after a keysigning party, you can use the tool *caff*. It can be installed from the AUR with the package **`caff-svn (https://aur.archlinux.org/packages/caff-svn/)`**[AUR].

To send the signatures to their owners you need a working **MTA**. If you do not have already one, install **msmtp**.

Always show long ID's and fingerprints

To always show long key ID's add `keyid-format 0xlong` to your configuration file. To always show full fingerprints of keys, add `with-fingerprint` to your configuration file.

Troubleshooting

Not enough random bytes available

When generating a key, gpg can run into this error:

```
Not enough random bytes available. Please do some other work to give the OS a chance to collect more entropy!
```

To check the available entropy, check the kernel parameters:

```
cat /proc/sys/kernel/random/entropy_avail
```

A healthy Linux system with a lot of entropy available will have return close to the full 4,096 bits of entropy. If the value returned is less than 200, the system is running low on entropy.

To solve it, remember you do not often need to create keys and best just do what the message suggests (e.g. create disk activity, move the mouse, edit the wiki - all will create entropy). If that does not help, check which service is using up the entropy and consider stopping it for the time. If that is no alternative, see **Random number generation#Alternatives**.

su

When using `pinentry`, you must have the proper permissions of the terminal device (e.g. `/dev/tty1`) in use. However, with *su* (or *sudo*), the ownership stays with the original user, not the new one. This means that pinentry will fail, even as root. The fix is to change the permissions of the device at some point before the use of pinentry (i.e. using gpg with an agent). If doing gpg as root, simply change the ownership to root right before using gpg:

```
# chown root /dev/ttyN  # where N is the current tty
```

and then change it back after using gpg the first time. The equivalent is likely to be true with `/dev/pts/`.

Note: The owner of tty *must* match with the user for which pinentry is running. Being part of the group `tty` **is not** enough.

Tip: If you run gpg with `script` it will use a new tty with the correct ownership:

```
# script -q -c "gpg --gen-key" /dev/null
```

Agent complains end of file

The default pinentry program is pinentry-gtk-2, which needs a DBus session bus to run properly. See **General troubleshooting#Session permissions** for details.

Alternatively, you can use `pinentry-qt`. See **#pinentry**.

KGpg configuration permissions

There have been issues with **`kgpg (https://www.archlinux.org/packages/?name=kgpg)`** being able to access the `~/.gnupg/` options. One issue might be a result of a deprecated *options* file, see the **bug (https://bugs.kde.org/show_bug.cgi?id=290221)** report.

GNOME on Wayland overrides SSH agent socket

For Wayland sessions, `gnome-session` sets `SSH_AUTH_SOCK` to the standard gnome-keyring socket, `$XDG_RUNTIME_DIR/keyring/ssh`. This overrides any value set in `~/.pam_environmment` or systemd unit files.

To disable this behavior, set the `GSM_SKIP_AGENT_WORKAROUND` variable:

~/.pam_environment

```
SSH_AGENT_PID   DEFAULT=
SSH_AUTH_SOCK   DEFAULT="${XDG_RUNTIME_DIR}/gnupg/S.gpg-agent.ssh"
GSM_SKIP_SSH_AGENT_WORKAROUND   DEFAULT="true"
```

mutt and gpg

To be asked for your GnuPG password only once per session, see **this forum thread (https://bbs.archlinux.org/viewtopic.php?pid=1490821#p1490821)**.

"Lost" keys, upgrading to gnupg version 2.1

When `gpg --list-keys` fails to show keys that used to be there, and applications complain about missing or invalid keys, some keys may not have been migrated to the new format.

Please read **GnuPG invalid packet workaround (http://jo-ke.name/wp/?p=111)**. Basically, it says that there is a bug with keys in the old `pubring.gpg` and `secring.gpg` files, which have now been superseded by the new `pubring.kbx` file and the `private-keys-v1.d/` subdirectory and files. Your missing keys can be recovered with the following commnads:

```
$ cd
$ cp -r .gnupg gnupgOLD
$ gpg --export-ownertrust > otrust.txt
$ gpg --import .gnupg/pubring.gpg
$ gpg --import-ownertrust otrust.txt
$ gpg --list-keys
```

gpg hanged for all keyservers (when trying to receive keys)

If gpg hanged with a certain keyserver when trying to receive keys, you might need to kill dirmngr in order to get access to other keyservers which are actually working, otherwise it might keeping hanging for all of them.

Smartcard not detected

Your user might not have the permission to access the smartcard which results in a `card error` to be thrown, even though the card is correctly set up and inserted.

One possible solution is to add a new group `scard` including the users who need access to the smartcard.

Then use an **udev** rule, similar to the following:

```
/etc/udev/rules.d/71-gnupg-ccid.rules
```

```
ACTION=="add", SUBSYSTEM=="usb", ENV{ID_VENDOR_ID}=="1050", ENV{ID_MODEL_ID}=="0116|0111", MODE="660", GROUP="scard"
```

One needs to adapt VENDOR and MODEL according to the `lsusb` output, the above example is for a YubikeyNEO.

gpg: WARNING: server 'gpg-agent' is older than us (x < y)

This warning appears if `gnupg` is upgraded and the old gpg-agent is still running. **Restart** the *user*'s `gpg-agent.socket` (i.e., use the `--user` flag when restarting).

gpg: ..., IPC connect call failed

Make sure `gpg-agent` and `dirmngr` are not running with `killall gpg-agent dirmngr` and the `$GNUPGHOME/crls.d/` folder has permission set to `700`.

If your keyring is stored on a vFat filesystem (e.g. a USB drive), `gpg-agent` will fail to create the required sockets (vFat does not support sockets), you can create redirects to a location that handles sockets, e.g. `/dev/shm`:

```
# export GNUPGHOME=/custom/gpg/home
# printf '%%Assuan%%\nsocket=/dev/shm/S.gpg-agent\n' > $GNUPGHOME/S.gpg-agent
# printf '%%Assuan%%\nsocket=/dev/shm/S.gpg-agent.browser\n' > $GNUPGHOME/S.gpg-agent.browser
# printf '%%Assuan%%\nsocket=/dev/shm/S.gpg-agent.extra\n' > $GNUPGHOME/S.gpg-agent.extra
# printf '%%Assuan%%\nsocket=/dev/shm/S.gpg-agent.ssh\n' > $GNUPGHOME/S.gpg-agent.ssh
```

Test that gpg-agent starts successfully with `gpg-agent --daemon`.

Error: [key] could not be locally signed or gpg: No default secret key: No public key

Occurs when attempting to sign keys on a non-standard keyring while a yubikey is plugged in, e.g. as **Pacman** does in `pacman-key --populate archlinux`. The solution is to remove the offending yubikey and start over.

See also

- **GNU Privacy Guard Homepage (https://gnupg.org/)**
- **RFC4880 "OpenPGP Message Format" (https://tools.ietf.org/html/rfc4880)**
- **Creating GPG Keys (Fedora) (https://fedoraproject.org/wiki/Creating_GPG_Keys)**
- **OpenPGP subkeys in Debian (https://wiki.debian.org/Subkeys)**
- **A more comprehensive gpg Tutorial (https://sanctum.geek.nz/arabesque/series/gnu-linux-crypto/)**
- **gpg.conf recommendations and best practices (https://help.riseup.net/en/security/message-security/openpgp/gpg-best-practices)**
- **Torbirdy gpg.conf (https://github.com/ioerror/torbirdy/blob/master/gpg.conf)**
- **/r/GPGpractice - a subreddit to practice using GnuPG. (https://www.reddit.com/r/GPGpractice/)**
- **Protecting code integrity with PGP (https://github.com/lfit/itpol/blob/master/protecting-code-integrity.md)**

Retrieved from "https://wiki.archlinux.org/index.php?title=GnuPG&oldid=510015"