Kernel Maintainer PGP guide

Author: Konstantin Ryabitsev <konstantin@linuxfoundation.org>

This document is aimed at Linux kernel developers, and especially at subsystem maintainers. It contains a subset of information discussed in the more general "Protecting Code Integrity" guide published by the Linux Foundation. Please read that document for more in-depth discussion on some of the topics mentioned in this guide.

The role of PGP in Linux Kernel development

PGP helps ensure the integrity of the code that is produced by the Linux kernel development community and, to a lesser degree, establish trusted communication channels between developers via PGP-signed email exchange.

The Linux kernel source code is available in two main formats:

- Distributed source repositories (git)
- Periodic release snapshots (tarballs)

Both git repositories and tarballs carry PGP signatures of the kernel developers who create official kernel releases. These signatures offer a cryptographic guarantee that downloadable versions made available via kernel.org or any other mirrors are identical to what these developers have on their workstations. To this end:

- git repositories provide PGP signatures on all tags
- tarballs provide detached PGP signatures with all downloads

Trusting the developers, not infrastructure

Ever since the 2011 compromise of core kernel.org systems, the main operating principle of the Kernel Archives project has been to assume that any part of the infrastructure can be compromised at any time. For this reason, the administrators have taken deliberate steps to emphasize that trust must always be placed with developers and never with the code hosting infrastructure, regardless of how good the security practices for the latter may be.

The above guiding principle is the reason why this guide is needed. We want to make sure that by placing trust into developers we do not simply shift the blame for potential future security incidents to someone else. The goal is to provide a set of guidelines developers can use to create a secure working environment and safeguard the PGP keys used to establish the integrity of the Linux kernel itself.

PGP tools

Use GnuPG v2

Your distro should already have GnuPG installed by default, you just need to verify that you are using version 2.x and not the legacy 1.4 release -- many distributions still package both, with the default `gpg` command invoking GnuPG v.1. To check, run:

```
$ gpg --version | head -n1
```

If you see `gpg (GnuPG) 1.4.x`, then you are using GnuPG v.1. Try the `gpg2` command (if you don't have it, you may need to install the gnupg2 package):

```
$ gpg2 --version | head -n1
```

If you see `gpg (GnuPG) 2.x.x`, then you are good to go. This guide will assume you have the version 2.2 of GnuPG (or later). If you are using version 2.0 of GnuPG, then some of the commands in this guide will not work, and you should consider installing the latest 2.2 version of GnuPG. Versions of gnupg-2.1.11 and later should be compatible for the purposes of this guide as well.

If you have both `gpg` and `gpg2` commands, you should make sure you are always using GnuPG v2, not the legacy version. You can enforce this by setting the appropriate alias:

```
$ alias gpg=gpg2
```

You can put that in your `.bashrc` to make sure it's always the case.

Configure gpg-agent options

The GnuPG agent is a helper tool that will start automatically whenever you use the `gpg` command and run in the background with the purpose of caching the private key passphrase. There are two options you should know in order to tweak when the passphrase should be expired from cache:

- `default-cache-ttl` (seconds): If you use the same key again before the time-to-live expires, the countdown will reset for another period. The default is 600 (10 minutes).
- `max-cache-ttl` (seconds): Regardless of how recently you've used the key since initial passphrase entry, if the maximum time-to-live countdown expires, you'll have to enter the passphrase again. The default is 30 minutes.

If you find either of these defaults too short (or too long), you can edit your `~/.gnupg/gpg-agent.conf` file to set your own values:

```
# set to 30 minutes for regular ttl, and 2 hours for max ttl
default-cache-ttl 1800
max-cache-ttl 7200
```

> **Note**
>
> It is no longer necessary to start gpg-agent manually at the beginning of your shell session. You may want to check your rc files to remove anything you had in place for older versions of GnuPG, as it may not be doing the right thing any more.

Set up a refresh cronjob

You will need to regularly refresh your keyring in order to get the latest changes on other people's public keys, which is best done with a daily cronjob:

```
@daily /usr/bin/gpg2 --refresh >/dev/null 2>&1
```

Check the full path to your `gpg` or `gpg2` command and use the `gpg2` command if regular `gpg` for you is the legacy GnuPG v.1.

Protect your master PGP key

This guide assumes that you already have a PGP key that you use for Linux kernel development purposes. If you do not yet have one, please see the "Protecting Code Integrity" document mentioned earlier for guidance on how to create a new one.

You should also make a new key if your current one is weaker than 2048 bits (RSA).

Master key vs. Subkeys

Subkeys are fully independent PGP keypairs that are tied to the "master" key using certifying key signatures (certificates). It is important to understand the following:

1. There are no technical differences between the "master key" and "subkeys."
2. At creation time, we assign functional limitations to each key by giving it specific capabilities.
3. A PGP key can have 4 capabilities:
    - **[S]** key can be used for signing
    - **[E]** key can be used for encryption
    - **[A]** key can be used for authentication
    - **[C]** key can be used for certifying other keys
4. A single key may have multiple capabilities.
5. A subkey is fully independent from the master key. A message encrypted to a subkey cannot be decrypted with the master key. If you lose your private subkey, it cannot be recreated from the master key in any way.

The key carrying the **[C]** (certify) capability is considered the "master" key because it is the only key that can be used to indicate relationship with other keys. Only the **[C]** key can be used to:

- add or revoke other keys (subkeys) with S/E/A capabilities
- add, change or revoke identities (uids) associated with the key
- add or change the expiration date on itself or any subkey
- sign other people's keys for web of trust purposes

By default, GnuPG creates the following when generating new keys:

- A master key carrying both Certify and Sign capabilities (**[SC]**)
- A separate subkey with the Encryption capability (**[E]**)

If you used the default parameters when generating your key, then that is what you will have. You can verify by running `gpg --list-secret-keys`, for example:

```
sec   rsa2048 2018-01-23 [SC] [expires: 2020-01-23]
      000000000000000000000000AAAABBBBCCCCDDDD
uid           [ultimate] Alice Dev <adev@kernel.org>
ssb   rsa2048 2018-01-23 [E] [expires: 2020-01-23]
```

Any key carrying the **[C]** capability is your master key, regardless of any other capabilities it may have assigned to it.

The long line under the `sec` entry is your key fingerprint -- whenever you see `[fpr]` in the examples below, that 40-character string is what it refers to.

Ensure your passphrase is strong

GnuPG uses passphrases to encrypt your private keys before storing them on disk. This way, even if your `.gnupg` directory is leaked or stolen in its entirety, the attackers cannot use your private keys without first obtaining the passphrase to decrypt them.

It is absolutely essential that your private keys are protected by a strong passphrase. To set it or change it, use:

```
$ gpg --change-passphrase [fpr]
```

Create a separate Signing subkey

Our goal is to protect your master key by moving it to offline media, so if you only have a combined **[SC]** key, then you should create a separate signing subkey:

```
$ gpg --quick-addkey [fpr] ed25519 sign
```

Remember to tell the keyservers about this change, so others can pull down your new subkey:

```
$ gpg --send-key [fpr]
```

> **Note**
>
> ECC support in GnuPG
>
> GnuPG 2.1 and later has full support for Elliptic Curve Cryptography, with ability to combine ECC subkeys with traditional RSA master keys. The main upside of ECC cryptography is that it is much faster computationally and creates much smaller signatures when compared byte for byte with 2048+ bit RSA keys. Unless you plan on using a smartcard device that does not support ECC operations, we recommend that you create an ECC signing subkey for your kernel work.
>
> If for some reason you prefer to stay with RSA subkeys, just replace "ed25519" with "rsa2048" in the above command. Additionally, if you plan to use a hardware device that does not support ED25519 ECC keys, like Nitrokey Pro or a Yubikey, then you should use "nistp256" instead or "ed25519."

Back up your master key for disaster recovery

The more signatures you have on your PGP key from other developers, the more reasons you have to create a backup version that lives on something other than digital media, for disaster recovery reasons.

The best way to create a printable hardcopy of your private key is by using the `paperkey` software written for this very purpose. See `man paperkey` for more details on the output format and its benefits over other solutions. Paperkey should already be packaged for most distributions.

Run the following command to create a hardcopy backup of your private key:

```
$ gpg --export-secret-key [fpr] | paperkey -o /tmp/key-backup.txt
```

Print out that file (or pipe the output straight to lpr), then take a pen and write your passphrase on the margin of the paper. **This is strongly recommended** because the key printout is still encrypted with that passphrase, and if you ever change it you will not remember what it used to be when you had created the backup -- *guaranteed*.

Put the resulting printout and the hand-written passphrase into an envelope and store in a secure and well-protected place, preferably away from your home, such as your bank vault.

> **Note**
>
> Your printer is probably no longer a simple dumb device connected to your parallel port, but since the output is still encrypted with your passphrase, printing out even to "cloud-integrated" modern printers should remain a relatively safe operation. One option is to change the passphrase on your master key immediately after you are done with paperkey.

Back up your whole GnuPG directory

> **Warning**
>
> **!!!Do not skip this step!!!**

It is important to have a readily available backup of your PGP keys should you need to recover them. This is different from the disaster-level preparedness we did with `paperkey`. You will also rely on these external copies whenever you need to use your Certify key -- such as when making changes to your own key or signing other people's keys after conferences and summits.

Start by getting a small USB "thumb" drive (preferably two!) that you will use for backup purposes. You will need to encrypt them using LUKS -- refer to your distro's documentation on how to accomplish this.

For the encryption passphrase, you can use the same one as on your master key.

Once the encryption process is over, re-insert the USB drive and make sure it gets properly mounted. Copy your entire `.gnupg` directory over to the encrypted storage:

```
$ cp -a ~/.gnupg /media/disk/foo/gnupg-backup
```

You should now test to make sure everything still works:

```
$ gpg --homedir=/media/disk/foo/gnupg-backup --list-key [fpr]
```

If you don't get any errors, then you should be good to go. Unmount the USB drive, distinctly label it so you don't blow it away next time you need to use a random USB drive, and put in a safe place -- but not too far away, because you'll need to use it every now and again for things like editing identities, adding or revoking subkeys, or signing other people's keys.

Remove the master key from your homedir

The files in our home directory are not as well protected as we like to think. They can be leaked or stolen via many different means:

- by accident when making quick homedir copies to set up a new workstation
- by systems administrator negligence or malice
- via poorly secured backups
- via malware in desktop apps (browsers, pdf viewers, etc)
- via coercion when crossing international borders

Protecting your key with a good passphrase greatly helps reduce the risk of any of the above, but passphrases can be discovered via keyloggers, shoulder-surfing, or any number of other means. For this reason, the recommended setup is to remove your master key from your home directory and store it on offline storage.

> **Warning**
>
> Please see the previous section and make sure you have backed up your GnuPG directory in its entirety. What we are about to do will render your key useless if you do not have a usable backup!

First, identify the keygrip of your master key:

```
$ gpg --with-keygrip --list-key [fpr]
```

The output will be something like this:

```
pub   rsa2048 2018-01-24 [SC] [expires: 2020-01-24]
      0000000000000000000000000AAAABBBBCCCCDDDD
      Keygrip = 1111000000000000000000000000000000000000
uid           [ultimate] Alice Dev <adev@kernel.org>
sub   rsa2048 2018-01-24 [E] [expires: 2020-01-24]
      Keygrip = 2222000000000000000000000000000000000000
sub   ed25519 2018-01-24 [S]
      Keygrip = 3333000000000000000000000000000000000000
```

Find the keygrip entry that is beneath the `pub` line (right under the master key fingerprint). This will correspond directly to a file in your `~/.gnupg` directory:

```
$ cd ~/.gnupg/private-keys-v1.d
$ ls
1111000000000000000000000000000000000000.key
2222000000000000000000000000000000000000.key
3333000000000000000000000000000000000000.key
```

All you have to do is simply remove the .key file that corresponds to the master keygrip:

```
$ cd ~/.gnupg/private-keys-v1.d
$ rm 1111000000000000000000000000000000000000.key
```

Now, if you issue the `--list-secret-keys` command, it will show that the master key is missing (the `#` indicates it is not available):

```
$ gpg --list-secret-keys
sec#  rsa2048 2018-01-24 [SC] [expires: 2020-01-24]
      0000000000000000000000000AAAABBBBCCCCDDDD
uid           [ultimate] Alice Dev <adev@kernel.org>
ssb   rsa2048 2018-01-24 [E] [expires: 2020-01-24]
ssb   ed25519 2018-01-24 [S]
```

You should also remove any `secring.gpg` files in the `~/.gnupg` directory, which are left over from earlier versions of GnuPG.

If you don't have the "private-keys-v1.d" directory

If you do not have a `~/.gnupg/private-keys-v1.d` directory, then your secret keys are still stored in the legacy `secring.gpg` file used by GnuPG v1. Making any changes to your key, such as changing the passphrase or adding a subkey, should automatically convert the old `secring.gpg` format to use `private-keys-v1.d` instead.

Once you get that done, make sure to delete the obsolete `secring.gpg` file, which still contains your private keys.

Move the subkeys to a dedicated crypto device

Move the subkeys to a dedicated crypto device

Even though the master key is now safe from being leaked or stolen, the subkeys are still in your home directory. Anyone who manages to get their hands on those will be able to decrypt your communication or fake your signatures (if they know the passphrase). Furthermore, each time a GnuPG operation is performed, the keys are loaded into system memory and can be stolen from there by sufficiently advanced malware (think Meltdown and Spectre).

The best way to completely protect your keys is to move them to a specialized hardware device that is capable of smartcard operations.

The benefits of smartcards

A smartcard contains a cryptographic chip that is capable of storing private keys and performing crypto operations directly on the card itself. Because the key contents never leave the smartcard, the operating system of the computer into which you plug in the hardware device is not able to retrieve the private keys themselves. This is very different from the encrypted USB storage device we used earlier for backup purposes -- while that USB device is plugged in and mounted, the operating system is able to access the private key contents.

Using external encrypted USB media is not a substitute to having a smartcard-capable device.

Available smartcard devices

Unless all your laptops and workstations have smartcard readers, the easiest is to get a specialized USB device that implements smartcard functionality. There are several options available:

- Nitrokey Start: Open hardware and Free Software, based on FSI Japan's Gnuk. One of the few available commercial devices that support ED25519 ECC keys, but offer fewest security features (such as resistance to tampering or some side-channel attacks).
- Nitrokey Pro 2: Similar to the Nitrokey Start, but more tamper-resistant and offers more security features. Pro 2 supports ECC cryptography (NISTP).
- Yubikey 5: proprietary hardware and software, but cheaper than Nitrokey Pro and comes available in the USB-C form that is more useful with newer laptops. Offers additional security features such as FIDO U2F, among others, and now finally supports ECC keys (NISTP).

LWN has a good review of some of the above models, as well as several others. Your choice will depend on cost, shipping availability in your geographical region, and open/proprietary hardware considerations.

> **Note**
>
> If you are listed in MAINTAINERS or have an account at kernel.org, you qualify for a free Nitrokey Start courtesy of The Linux Foundation.

Configure your smartcard device

Your smartcard device should Just Work (TM) the moment you plug it into any modern Linux workstation. You can verify it by running:

```
$ gpg --card-status
```

If you see full smartcard details, then you are good to go. Unfortunately, troubleshooting all possible reasons why things may not be working for you is way beyond the scope of this guide. If you are having trouble getting the card to work with GnuPG, please seek help via usual support channels.

To configure your smartcard, you will need to use the GnuPG menu system, as there are no convenient command-line switches:

```
$ gpg --card-edit
[...omitted...]
gpg/card> admin
Admin commands are allowed
gpg/card> passwd
```

You should set the user PIN (1), Admin PIN (3), and the Reset Code (4). Please make sure to record and store these in a safe place -- especially the Admin PIN and the Reset Code (which allows you to completely wipe the smartcard). You so rarely need to use the Admin PIN, that you will inevitably forget what it is if you do not record it.

Getting back to the main card menu, you can also set other values (such as name, sex, login data, etc), but it's not necessary and will additionally leak information about your smartcard should you lose it.

> **Note**
>
> Despite having the name "PIN", neither the user PIN nor the admin PIN on the card need to be numbers.

> **Warning**

> Some devices may require that you move the subkeys onto the device before you can change the passphrase. Please check the documentation provided by the device manufacturer.

Move the subkeys to your smartcard

Exit the card menu (using "q") and save all changes. Next, let's move your subkeys onto the smartcard. You will need both your PGP key passphrase and the admin PIN of the card for most operations:

```
$ gpg --edit-key [fpr]

Secret subkeys are available.

pub  rsa2048/AAAABBBBCCCCDDDD
     created: 2018-01-23  expires: 2020-01-23  usage: SC
     trust: ultimate      validity: ultimate
ssb  rsa2048/1111222233334444
     created: 2018-01-23  expires: never       usage: E
ssb  ed25519/5555666677778888
     created: 2017-12-07  expires: never       usage: S
[ultimate] (1). Alice Dev <adev@kernel.org>

gpg>
```

Using `--edit-key` puts us into the menu mode again, and you will notice that the key listing is a little different. From here on, all commands are done from inside this menu mode, as indicated by `gpg>`.

First, let's select the key we'll be putting onto the card -- you do this by typing `key 1` (it's the first one in the listing, the **[E]** subkey):

```
gpg> key 1
```

In the output, you should now see `ssb*` on the **[E]** key. The `*` indicates which key is currently "selected." It works as a *toggle*, meaning that if you type `key 1` again, the `*` will disappear and the key will not be selected any more.

Now, let's move that key onto the smartcard:

```
gpg> keytocard
Please select where to store the key:
   (2) Encryption key
Your selection? 2
```

Since it's our **[E]** key, it makes sense to put it into the Encryption slot. When you submit your selection, you will be prompted first for your PGP key passphrase, and then for the admin PIN. If the command returns without an error, your key has been moved.

Important: Now type `key 1` again to unselect the first key, and `key 2` to select the **[S]** key:

```
gpg> key 1
gpg> key 2
gpg> keytocard
Please select where to store the key:
   (1) Signature key
   (3) Authentication key
Your selection? 1
```

You can use the **[S]** key both for Signature and Authentication, but we want to make sure it's in the Signature slot, so choose (1). Once again, if your command returns without an error, then the operation was successful:

```
gpg> q
Save changes? (y/N) y
```

Saving the changes will delete the keys you moved to the card from your home directory (but it's okay, because we have them in our backups should we need to do this again for a replacement smartcard).

Verifying that the keys were moved

If you perform `--list-secret-keys` now, you will see a subtle difference in the output:

```
$ gpg --list-secret-keys
sec#  rsa2048 2018-01-24 [SC] [expires: 2020-01-24]
      0000000000000000000000000AAAABBBBCCCCDDDD
uid           [ultimate] Alice Dev <adev@kernel.org>
ssb>  rsa2048 2018-01-24 [E] [expires: 2020-01-24]
ssb>  ed25519 2018-01-24 [S]
```

The `>` in the `ssb>` output indicates that the subkey is only available on the smartcard. If you go back into your secret keys directory and look at the contents there, you will notice that the `.key` files there have been replaced with stubs:

```
$ cd ~/.gnupg/private-keys-v1.d
$ strings *.key | grep 'private-key'
```

The output should contain `shadowed-private-key` to indicate that these files are only stubs and the actual content is on the smartcard.

Verifying that the smartcard is functioning

To verify that the smartcard is working as intended, you can create a signature:

```
$ echo "Hello world" | gpg --clearsign > /tmp/test.asc
$ gpg --verify /tmp/test.asc
```

This should ask for your smartcard PIN on your first command, and then show "Good signature" after you run `gpg --verify`.

Congratulations, you have successfully made it extremely difficult to steal your digital developer identity!

Other common GnuPG operations

Here is a quick reference for some common operations you'll need to do with your PGP key.

Mounting your master key offline storage

You will need your master key for any of the operations below, so you will first need to mount your backup offline storage and tell GnuPG to use it:

```
$ export GNUPGHOME=/media/disk/foo/gnupg-backup
$ gpg --list-secret-keys
```

You want to make sure that you see `sec` and not `sec#` in the output (the `#` means the key is not available and you're still using your regular home directory location).

Extending key expiration date

The master key has the default expiration date of 2 years from the date of creation. This is done both for security reasons and to make obsolete keys eventually disappear from keyservers.

To extend the expiration on your key by a year from current date, just run:

```
$ gpg --quick-set-expire [fpr] 1y
```

You can also use a specific date if that is easier to remember (e.g. your birthday, January 1st, or Canada Day):

```
$ gpg --quick-set-expire [fpr] 2020-07-01
```

Remember to send the updated key back to keyservers:

```
$ gpg --send-key [fpr]
```

Updating your work directory after any changes

After you make any changes to your key using the offline storage, you will want to import these changes back into your regular working directory:

```
$ gpg --export | gpg --homedir ~/.gnupg --import
$ unset GNUPGHOME
```

Using gpg-agent over ssh

You can forward your gpg-agent over ssh if you need to sign tags or commits on a remote system. Please refer to the instructions provided on the GnuPG wiki:

- Agent Forwarding over SSH

It works more smoothly if you can modify the sshd server settings on the remote end.

Using PGP with Git

One of the core features of Git is its decentralized nature -- once a repository is cloned to your system, you have full history of the project, including all of its tags, commits and branches. However, with hundreds of cloned repositories floating around, how does anyone verify that their copy of linux.git has not been tampered with by a malicious third party?

Or what happens if a backdoor is discovered in the code and the "Author" line in the commit says it was done by you, while you're pretty sure you had nothing to do with it?

To address both of these issues, Git introduced PGP integration. Signed tags prove the repository integrity by assuring that its contents are exactly the same as on the workstation of the developer who created the tag, while signed commits make it nearly impossible for someone to impersonate you without having access to your PGP keys.

Configure git to use your PGP key

If you only have one secret key in your keyring, then you don't really need to do anything extra, as it becomes your default key. However, if you happen to have multiple secret keys, you can tell git which key should be used (`[fpr]` is the fingerprint of your key):

```
$ git config --global user.signingKey [fpr]
```

IMPORTANT: If you have a distinct `gpg2` command, then you should tell git to always use it instead of the legacy `gpg` from version 1:

```
$ git config --global gpg.program gpg2
$ git config --global gpgv.program gpgv2
```

How to work with signed tags

To create a signed tag, simply pass the `-s` switch to the tag command:

```
$ git tag -s [tagname]
```

Our recommendation is to always sign git tags, as this allows other developers to ensure that the git repository they are pulling from has not been maliciously altered.

How to verify signed tags

To verify a signed tag, simply use the `verify-tag` command:

```
$ git verify-tag [tagname]
```

If you are pulling a tag from another fork of the project repository, git should automatically verify the signature at the tip you're pulling and show you the results during the merge operation:

```
$ git pull [url] tags/sometag
```

The merge message will contain something like this:

```
Merge tag 'sometag' of [url]

[Tag message]

# gpg: Signature made [...]
# gpg: Good signature from [...]
```

If you are verifying someone else's git tag, then you will need to import their PGP key. Please refer to the ":ref:`verify_identities`" section below.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]maintainer-pgp-guide.rst`, line 761); *backlink*

Unknown interpreted text role "ref".

Note

If you get "`gpg: Can't check signature: unknown pubkey algorithm`" error, you need to tell git to use gpgv2 for verification, so it properly processes signatures made by ECC keys. See instructions at the start of this section.

Configure git to always sign annotated tags

Chances are, if you're creating an annotated tag, you'll want to sign it. To force git to always sign annotated tags, you can set a global configuration option:

```
$ git config --global tag.forceSignAnnotated true
```

How to work with signed commits

It is easy to create signed commits, but it is much more difficult to use them in Linux kernel development, since it relies on patches sent to the mailing list, and this workflow does not preserve PGP commit signatures. Furthermore, when rebasing your repository to match upstream, even your own PGP commit signatures will end up discarded. For this reason, most kernel developers don't bother signing their commits and will ignore signed commits in any external repositories that they rely upon in their work.

However, if you have your working git tree publicly available at some git hosting service (kernel.org, infradead.org, ozlabs.org, or others), then the recommendation is that you sign all your git commits even if upstream developers do not directly benefit from this practice.

We recommend this for the following reasons:

1. Should there ever be a need to perform code forensics or track code provenance, even externally maintained trees carrying PGP commit signatures will be valuable for such purposes.
2. If you ever need to re-clone your local repository (for example, after a disk failure), this lets you easily verify the repository integrity before resuming your work.
3. If someone needs to cherry-pick your commits, this allows them to quickly verify their integrity before applying them.

Creating signed commits

To create a signed commit, you just need to pass the `-S` flag to the `git commit` command (it's capital `-S` due to collision with another flag):

```
$ git commit -S
```

Configure git to always sign commits

You can tell git to always sign commits:

```
git config --global commit.gpgSign true
```

> **Note**
>
> Make sure you configure `gpg-agent` before you turn this on.

How to verify kernel developer identities

Signing tags and commits is easy, but how does one go about verifying that the key used to sign something belongs to the actual kernel developer and not to a malicious imposter?

Configure auto-key-retrieval using WKD and DANE

If you are not already someone with an extensive collection of other developers' public keys, then you can jumpstart your keyring by relying on key auto-discovery and auto-retrieval. GnuPG can piggyback on other delegated trust technologies, namely DNSSEC and TLS, to get you going if the prospect of starting your own Web of Trust from scratch is too daunting.

Add the following to your `~/.gnupg/gpg.conf`:

```
auto-key-locate wkd,dane,local
auto-key-retrieve
```

DNS-Based Authentication of Named Entities ("DANE") is a method for publishing public keys in DNS and securing them using DNSSEC signed zones. Web Key Directory ("WKD") is the alternative method that uses https lookups for the same purpose. When using either DANE or WKD for looking up public keys, GnuPG will validate DNSSEC or TLS certificates, respectively, before adding auto-retrieved public keys to your local keyring.

Kernel.org publishes the WKD for all developers who have kernel.org accounts. Once you have the above changes in your `gpg.conf`, you can auto-retrieve the keys for Linus Torvalds and Greg Kroah-Hartman (if you don't already have them):

```
$ gpg --locate-keys torvalds@kernel.org gregkh@kernel.org
```

If you have a kernel.org account, then you should add the kernel.org UID to your key to make WKD more useful to other kernel developers.

Web of Trust (WOT) vs. Trust on First Use (TOFU)

PGP incorporates a trust delegation mechanism known as the "Web of Trust." At its core, this is an attempt to replace the need for centralized Certification Authorities of the HTTPS/TLS world. Instead of various software makers dictating who should be your trusted certifying entity, PGP leaves this responsibility to each user.

Unfortunately, very few people understand how the Web of Trust works. While it remains an important aspect of the OpenPGP specification, recent versions of GnuPG (2.2 and above) have implemented an alternative mechanism called "Trust on First Use" (TOFU). You can think of TOFU as "the SSH-like approach to trust." With SSH, the first time you connect to a remote system, its key fingerprint is recorded and remembered. If the key changes in the future, the SSH client will alert you and refuse to connect, forcing you to make a decision on whether you choose to trust the changed key or not. Similarly, the first time you import someone's PGP key, it is assumed to be valid. If at any point in the future GnuPG comes across another key with the same identity, both the previously imported key and the new key will be marked as invalid and you will need to manually figure out which one to keep.

We recommend that you use the combined TOFU+PGP trust model (which is the new default in GnuPG v2). To set it, add (or modify) the `trust-model` setting in `~/.gnupg/gpg.conf`:

```
trust-model tofu+pgp
```

How to use keyservers (more) safely

If you get a "No public key" error when trying to validate someone's tag, then you should attempt to lookup that key using a keyserver. It is important to keep in mind that there is absolutely no guarantee that the key you retrieve from PGP keyservers belongs to the actual person -- that much is by design. You are supposed to use the Web of Trust to establish key validity.

How to properly maintain the Web of Trust is beyond the scope of this document, simply because doing it properly requires both effort and dedication that tends to be beyond the caring threshold of most human beings. Here are some shortcuts that will help you reduce the risk of importing a malicious key.

First, let's say you've tried to run `git verify-tag` but it returned an error saying the key is not found:

```
$ git verify-tag sunxi-fixes-for-4.15-2
gpg: Signature made Sun 07 Jan 2018 10:51:55 PM EST
gpg:                using RSA key DA73759BF8619E484E5A3B47389A54219C0F2430
gpg:                issuer "wens@...org"
gpg: Can't check signature: No public key
```

Let's query the keyserver for more info about that key fingerprint (the fingerprint probably belongs to a subkey, so we can't use it directly without finding out the ID of the master key it is associated with):

```
$ gpg --search DA73759BF8619E484E5A3B47389A54219C0F2430
gpg: data source: hkp://keys.gnupg.net
(1) Chen-Yu Tsai <wens@...org>
      4096 bit RSA key C94035C21B4F2AEB, created: 2017-03-14, expires: 2019-03-15
Keys 1-1 of 1 for "DA73759BF8619E484E5A3B47389A54219C0F2430".  Enter number(s), N)ext, or Q)uit > q
```

Locate the ID of the master key in the output, in our example `C94035C21B4F2AEB`. Now display the key of Linus Torvalds that you have on your keyring:

```
$ gpg --list-key torvalds@kernel.org
pub   rsa2048 2011-09-20 [SC]
      ABAF11C65A2970B130ABE3C479BE3E4300411886
uid           [ unknown] Linus Torvalds <torvalds@kernel.org>
sub   rsa2048 2011-09-20 [E]
```

Next, find a trust path from Linus Torvalds to the key-id you found via `gpg --search` of the unknown key. For this, you can use several tools including https://github.com/mricon/wotmate, https://git.kernel.org/pub/scm/docs/kernel/pgpkeys.git/tree/graphs, and https://the.earth.li/~noodles/pathfind.html.

If you get a few decent trust paths, then it's a pretty good indication that it is a valid key. You can add it to your keyring from the keyserver now:

```
$ gpg --recv-key C94035C21B4F2AEB
```

This process is not perfect, and you are obviously trusting the administrators of the PGP Pathfinder service to not be malicious (in fact, this goes against :ref:`devs_not_infra`). However, if you do not carefully maintain your own web of trust, then it is a marked improvement over blindly trusting keyservers.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]maintainer-pgp-guide.rst, line 959); *backlink***
>
> Unknown interpreted text role "ref".