(../atom.xml) (https://github.com/yanhan) (http://sg.linkedin.com/pub/yan-han-pang/92/158/91b) (https://twitter.com/yanhan_pang)

☰

How to use GPG to encrypt stuff

27 Sep 2017, by *Pang Yan Han*

Tags: gpg (../tags/gpg.html)

Disclaimer: Opinions expressed on this blog are solely my own and do not express the views or opinions of my employer(s), past or present.

You want to exchange a message / file securely with another party and do not want anyone else to look at the information. This is where you can use GPG to encrypt the message / file you want to send.

The overview of the steps are as follows:

1. Make sure everything is in one file. If there are multiple files, you can put them inside a folder and create a tarball of that folder, or just directly tarball all the files
2. (OPTIONAL) Sign the file using your **private key**. This will generate a signature that can be verified using your public key
3. Encrypt the file using the recipient's **public key**. This will generate an encrypted file that can only be decrypted using the recipient's private key
4. Send the encrypted file and (optionally) the signature to the other person
5. The recipient of the message will decrypt the encrypted file using his/her private key
6. (OPTIONAL) The recipient of the message will verify the signature using your public key

The steps below will not correspond to the points in this overview because the guide is intended to be as complete as possible without bogging you down with unnecessary details.

Step 1: Generate a GPG keypair

For recipient: This step is absolutely required.

For the sender: This step is required if you wish to send a signature to the recipient. Otherwise you can skip this step.

To generate a GPG keypair, you first have to install GPG. This should come automatically with Linux. For Mac OS X users, I highly recommend that you install GPG Suite (https://gpgtools.org/).

To generate a key, run the following command:

```
gpg --gen-key
```

You will be prompted for some information. I will be guiding you through the flow below.

```
gpg (GnuPG) 1.4.16; Copyright (C) 2013 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

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)
Your selection?
```

Type `1` followed by Enter for the `RSA and RSA` option.

```
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048)
```

Type `4096` followed by Enter. We want our key to be as strong as possible.

```
Requested keysize is 4096 bits
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)
```

Type `0` followed by Enter. For convenience, we do not want the key to expire

```
Key does not expire at all
Is this correct? (y/N)
```

Type `y` followed by Enter.

```
You need a user ID to identify your key; the software constructs the user ID
from the Real Name, Comment and Email Address in this form:
    "Heinrich Heine (Der Dichter) <heinrichh@duesseldorf.de>"

Real name:
```

Type your real name followed by Enter.

```
Email address:
```

Type your email address followed by Enter. The email address you use depends on the context. If you are exchanging files securely for work, then type your work email address.

```
Comment:
```

This can be left blank. Press Enter if you have nothing to add.

```
You selected this USER-ID:
    "Your name <your.name@yourdomain.com>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit?
```

Type `O` followed by Enter.

```
You need a Passphrase to protect your secret key.
```

If you are using a GUI, you will see a GUI prompt open up for your passphrase. **Do not forget your passphrase!!!** Otherwise your GPG keypair is worthless. Use something long and easily remembered by you but hard for other people and computers to guess. This blog post by LastPass (https://blog.lastpass.com/2013/04/how-to-create-secure-master-password.html/) is a good guide to generating a good passphrase.

```
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.

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

Once you the above, just go do some other stuff at your computer. It could take a few minutes before this is done. To speed up the process, you can run some intensive commands like `cd ~ && find . -type f` (assuming you have a lot of files in your home folder)

```
gpg: key ________ marked as ultimately trusted
public and secret key created and signed.

gpg: checking the trustdb
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg: depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 1u
pub   4096R/________ 2017-09-26
      Key fingerprint = ____ ____ ____ ____ ____  ____ ____ ____ ____ ____
uid                  Your Name <your.name@yourdomain.com>
sub   4096R/________ 2017-09-26
```

Step 2: Make your GPG public key available to the other party

For the recipient: This step is absolutely required.

For the sender: This step is required if you wish to send a signature to the recipient. Otherwise you can skip this step.

Regardless of whether you are the sender or the recipient, we shall cover 2 methods of making your GPG public key available to the other party.

Method 1: Send your public key as a file to the recipient

Remember the email you used to create your GPG keypair? We shall assume it is `your.name@yourdomain.com`. Run the following command:

```
gpg --armor --output mypubkey.gpg --export your.name@yourdomain.com
```

The `mypubkey.gpg` file should look similar to the following:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1

mfgQighgkm47609/132415jkamfgASHDFGkgm48610xktgy46523jrkfagmb01f4
...
...
... A lot of similar lines omitted ...
...
...
-----END PGP PUBLIC KEY BLOCK-----
```

Now you can send this file to your friend / colleague.

Method 2: Upload your public key to a PGP public key server

The alternative method is to upload your public key a PGP public key server and have your friend / colleague download your public key from there.

We have to find out the public key ID of our GPG key. Do so by running the following command:

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

You should see something similar to the following:

```
/home/youruser/.gnupg/secring.gpg
---------------------------------
sec   4096R/DEADBEEF 2017-09-26
uid                  Your name here <your.name@yourdomain.com>
ssb   4096R/A0156F2D 2017-09-26
```

The key ID of your GPG public key is in the first row on the same line as the `sec` field. In this made up example here, it is `DEADBEEF`.

To export your GPG public key, run the following command, replacing the public key ID accordingly:

```
gpg --send-keys DEADBEEF
```

You should see something like the following:

```
gpg: sending key DEADBEEF to hkp server keys.gnupg.net
```

Take note of the GPG server that the key was uploaded to. We will be needing it later.

Step 3: (For sender) Retrieve the message recipient's public key

This step is for the sender of the message. We shall cover what follows from the 2 methods that we covered in step 2.

Method 1: Friend / colleague sent his / her public key to you

This corresponds to Step 2 Method 1, where your friend / colleague (the recipient of the message) send his / her public key to you in a file. We have to import the public key into our keyring. Suppose this file is called `recipient-pubkey.gpg`. To import it, run:

```
gpg --import recipient-pubkey.gpg
```

You should see output similar to the following:

```
gpg: key ______: public key "Your friend's name <your.friend@yourfriendsdomain.cc
gpg: Total number processed: 1
gpg:               imported: 1
```

Method 2: Friend / colleague uploaded his / her GPG public key to a PGP public key server

Ask your friend / colleague the server that he / she uploaded his / her public key to.

Suppose it is `keys.gnupg.net`. Suppose your friend's email address is `your.friend@yourfriendsdomain.com`. To find his / her key, run the following command (replacing the keyserver and email address accordingly):

```
gpg --keyserver keys.gnupg.net --search-key your.friend@yourfriendsdomain.com
```

If everything goes well, you should see output similar to the following:

```
gpg: data source: http://192.94.109.73:11371
(1)     Your Friend's Name <your.friend@yourfriendsdomain.com>
        4096 bit RSA key 5019A105E6069CD4, created: 2017-09-26
Keys 1-1 of 1 for "your.friend@yourfriendsdomain.com". Enter number(s), N)ext, or
```

Type `1` followed by enter if you are sure that this is your friend's public key and GPG will proceed to import it into your public keyring.

If you are unsure this key belongs to your friend, verify with him / her. Get them to run the following command:

```
gpg --list-keys --keyid-format LONG --fingerprint
```

Verify that the public key ID you see (in our example it is `5019A105E6069CD4`) matches the his / her public key ID. Proceed to import the key if everything is good.

Step 4: (For sender) Encrypt the message

We will now encrypt the message using the sender's public key. Assuming the sender's email that is associated with public key is `your.friend@yourfriendsdomain.com` and the file you want to encrypt is called `myfile.txt`, run the following command:

```
gpg --output myfile.txt.gpg --encrypt --recipient your.friend@yourfriendsdomain.c
```

The encrypted file will be at `myfile.txt.gpg`. If you take a look at it, you will see that it is in a binary format. Now you can send this file to your friend / colleague. Only the recipient will be able to decrypt it using his / her private key.

Step 5: (For sender) Signing the message

NOTE: This step is optional. The reason why you as a sender may want to sign the message is for the recipient to verify that it is indeed you who sent the message and not someone else. This is a form of anti-tampering.

Instead of signing the message (which is also a form of encrypting the message), we shall generate a checksum of the message and sign that instead.

Let us generate a SHA256 sum of the **unencrypted** file (assuming it is named `myfile.txt`) and sign that using our **private key**:

```
shasum -a 256 myfile.txt | awk '{print $1}' >myfile.txt.sha256sum
gpg --output myfile.txt.sha256sum.sig --sign myfile.txt.sha256sum
```

You can then send `myfile.txt.sha256sum.sig` to the recipient.

Step 6: (For receiver) Decrypt the message

Suppose the encrypted message sent by the sender is called `myfile.txt.gpg` and it was encrypted using your public key.

To decrypt this message using your private key, run:

```
gpg --output myfile.txt --decrypt myfile.txt.gpg
```

You will be prompted for the passphrase of your private key. Assuming the sender specified the recipient of the message using the `--recipient` option when encrypting the message, GPG should be able to identify the correct private key to use (assuming you have multiple keypairs).

Now you have the message! It is in the file specified by the `--output` flag. If the sender did not provide a signature and you trust him / her, then you are done. Otherwise, go on to the next step to verify the signature.

Step 7: (For receiver) Verify the signature of the message

Suppose the signature is named `myfile.txt.sha256sum.sig`. To verify that the signature is indeed sent by the sender, run the following command:

```
gpg --verify myfile.txt.sha256sum.sig
```

You should see an output similar to the following:

```
gpg: Signature made Tue 26 Sep 2017 09:10:22 PM SGT
gpg:                using RSA key ID 741A869EBC910BE2
gpg: Good signature from "Sender's name <sender.name@sendersdomain.com>" [unknown
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: 85AF 5410 058C FE1D 76DA  986F 910C B963 468A 0F16
```

Check that the public key ID and fingerprint. matches the sender's public key ID in your keyring. Run `gpg --list-keys --keyid-format LONG --fingerprint` to list the public keys in your GPG keyring alongside their fingerprint.

To get the actual content from the signature, run:

```
gpg --output myfile.txt.sha256sum  --decrypt myfile.txt.sha256sum.sig
```

You should see some output very similar to / the same as that from running the command `gpg --verify myfile.txt.sha256sum.sig`.

You should verify that the sha256 sum inside `myfile.txt.sha256sum` is the same as the sha256 sum of the decrypted file that the sender sent you.

And if you only wish to send and / or receive messages securely, this is the end. If you are curious about why we go through all these trouble just to exchange some files, read the next section.

Why go through all these trouble? / Why use GPG to exchange messages?

I have to admit that I am not the best person to write this section due to my lack of knowledge. That being said, I will be explaining the concepts here to the best of my ability.

GPG uses public key cryptography. This is also known as asymmetric encryption, where a **keypair** consisting of a public and private key is involved, as opposed to symmetric encryption, which makes use of a single key. The public key can be distributed to whoever you want. The private key must be closely guarded and in GPG's case is protected by a strong passphrase.

Suppose Alice wants to send a message to Bob. Alice will encrypt Bob's message using Bob's **public key**. This ensures that no one else other than Bob can read the message, because only Bob can decrypt the message using his **private key**. Anyone with Bob's public key encrypt a message that only Bob can view. This explanation should also explain why Bob must guard his private key - because anyone with his private key can decrypt any message encrypted using his public key.

Now, Alice has guaranteed that whatever message she sends to Bob can only be viewed by Bob. This ensures the **privacy** of the message.

However, anyone with Bob's public key can send a message intended for him. How can Bob ensure that when Alice says she sent him a message, that the message is from Alice and not from someone else?

This is where signatures come in. Alice can generate a file containing the checksum of the original, plaintext message and sign that file using her **private key**. Anyone who has Alice's **public key** (and trusts her public key) can verify the authenticity of that file, because to generate that signature, Alice's **private key** is required. Assuming that Alice's private key has not been compromised, it is pretty much impossible for anyone to forge the signature, or for Alice to deny that she did not generate the signature. This ensures the **authenticity** of the message, that it is indeed sent by Alice herself and not anyone else.

The reason why Alice signs a file containing the checksum of the original message is because signing a file will encrypt it using Alice's public key. Anyone with Alice's public key can decrypt that file. So it is not safe for Alice to sign the message she wants to send to Bob, otherwise anyone with her public key (and not just Bob) can read it. A file containing the checksum can be sent through non trusted

channels without allowing anyone to reverse engineer the contents of the original message, while letting the intended recipient verify the authenticity of the message. This is why I will recommend sending a signature to the recipient even though it is completely optional.

If you are relatively new to public key cryptography, the above could take a while to grasp. It is perfectly normal.

If you have read everything so far, you will notice something. Everything hinges on both parties trusting each other's public keys. Especially Alice trusting Bob's public key. Why do I say so? Suppose there is a malicious 3rd party called Eve that is intercepting all traffic between Alice and Bob. Through some very clever means, Alice got Eve's public key instead of Bob's public key and believes that it is Bob's public key. When Alice encrypts a message intended for Bob using "Bob's" public key, Eve can decrypt the message that was originally meant for Bob. Furthermore, assuming that Eve has Bob's public key (it is a public key after all), Eve can alter the contents of the message, encrypt it with Bob's public key and then send the altered message to Bob. As for signatures, Eve can trick Bob into believing that her public key is Alice's public key. Then Bob will trust whatever signature that Eve sends to him and believes it is a signature from Alice.

While the above scenario may seem hypothetical, it is entirely possible. The crux of the message is: everything is based on trust and there has to be secure key exchange - we have to know whether a public key actually belongs to who it says it belongs to and we cannot trust public keys blindly. Now, these are topics I barely know anything about and shall not go into =)

References

- The GNU Privacy Handbook - Making and verifying signatures (https://www.gnupg.org/gph/en/manual/x135.html)
- The GNU Privacy Handbook - Exchanging keys (https://www.gnupg.org/gph/en/manual/x56.html)
- The GNU Privacy Handbook - Encrypting and decrypting documents (https://www.gnupg.org/gph/en/manual/x110.html)
- What is the difference between encrypting and signing in asymmetric encryption? (https://stackoverflow.com/q/454048)
- How do I check if my OpenPGP key is in the Ubuntu keyserver? (https://askubuntu.com/q/29889)
- GPG Cheat Sheet (http://irtfweb.ifa.hawaii.edu/~lockhart/gpg/)
- How to make GnuPG display full 8-byte/64-bit key ID? (https://superuser.com/q/619145)
- Short OpenPGP key IDs are insecure, how to configure GnuPG to use long key IDs instead? (https://security.stackexchange.com/a/84281)

Zeying Zhu • 7 days ago

Great step by step instruction! Thank you!

Github github.com/yanhan (https://github.com/yanhan/)

Site proudly generated by Hakyll (http://jaspervdj.be/hakyll)