12 December 2000

# GPG For People Who Hate To Read README Files

P. J. Salzman
*Department of Physics*
*University of California*
*Davis, California 95616 USA*
*p@dirac.org*

## Abstract

If you hate to read README files or just simply hate to read, and want to learn how to use GPG, then this Bud's for you.

# I. Your Keys

In common parlance, a ‘key’ is something like a ‘password’. It probably wouldn’t hurt to mentally translate ‘key’ into ‘password’ as you read this document.

With gpg, you have two keys: A public key that everybody knows and a private key that only you know.

The private key is a passphrase like *ForWhomTheBellRings* or *ZZippetyDoo-Dahh*. You must never tell anyone your private key under any circumstances. It must always be kept secret.

The public key is a rather long bunch of binary data which is really quite ugly to look at. It has an equivalent representation in ASCII. The ASCII representation of your public key is still long and ugly, but not quite as ugly as the binary form. When you represent your key in the ASCII form, the key is called *armored*. If you ever send someone your public key, it will always be armored.

Really, only computers use unarmored public keys. Humans always use the armored form.

## II. What Keys Are Good For

There are two main rules of thumb:

> *Data that is encrypted with a public key can only be decrypted by the matching private key*

Suppose you want to send your friend a file that you don't want other people to read. You would encrypt this message with his public key. Since he's (hopefully) the only one with the matching private key, he's the only one who will be able to read your message.

> *Data that is encrypted with a private key can only be decrypted by the matching public key*

Suppose you want to send a file to someone, and you want them to be confident that the file

1. actually came from you
2. hasn't been tampered with from the time you sent it to the time they received it.

You can use your private key to generate a signature of a file; the signature itself is a small file. Anybody with your public key, the original file and the signature can verify that the file hasn't been modified since you created the signature. This is called *signing a file* and is useful, for instance, in making sure people don't put nasty things like backdoors or viruses in your files.

You might have noticed that this means that not only must you have a public and private key, but you also must have other peoples' public keys as well. This is true; for the people that you want to communicate or swap data with securely, you must have their public keys.

A keyring is simply a large file that stores keys. You have a public keyring which stores your friends' public keys as well as your own public key(s). You also have a private key ring which holds your private key(s).

## III. First Steps

You start by generating a key

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

Use an encryption method of DSA and El-Gamal, a keysize of 2048 and no expiration. Lastly, it'll ask you for a passphrase. This is important because you'll be typing this in A LOT. Make sure it's something you can type in conveniently otherwise you'll be hating life, yet something that can't be guessed.

gpg will now begin generating your key. It needs random bits, so wiggle your mouse until it's done.

Congratulations. You now have a public key. Would you like to see what your public key looks like?

```
gpg --armor --export
```

You can redirect this into a file, of course. Actually, this is the ASCII representation of your public key. You can see what the binary representation of your public key looks like by leaving out the `--armor`. It'll probably hose your xterm, but go ahead; you should see it at least once. If your terminal looks strange afterwards, type `reset`. If that doesn't work, type `stty sane`.

Next make a revocation certificate. You'll use this in case your passphrase gets compromised:

```
gpg --gen-revoke <your username>
```

Store it somewhere safe (not on a computer; print it out). Hopefully you'll never need to use it.

## IV. Publishing Your Key

Now that you have a public key, it's time to tell the rest of the world about it. In the file `$HOME/.gnupg/options`, add the following line:

```
keyserver pgp.ai.mit.edu
```

Now you can send your key to the MIT keyserver by typing :

```
gpg --send-keys <your username>
```

Now you need to give your public key to your friends. Put your public key into a file (this should look familiar):

```
gpg --armor --export <your username> > MyKey.asc
```

Convention dictates that when you put the ASCII version of your public key into a file, you use the extension `.asc`. When you put the binary version of your public key in a file, most people use the `.gpg` extension[1].

Now that you have your public key all nice and tidy in a file, you'll want to send this file to all your friends, and explain to them that you just entered the wonderful world of gpg, and this is your brand new public key.

You might be asking ''How do my friends know that I'm the one sending the email? After all, the email with my public key could've been forged by a Russian spy who wants to fool everyone into thinking that she is me". This is a valid question, and will be one of the things talked about in the section on Trust.

[1] Something like: `gpg --export <your username> > MyKey.gpg`, but this isn't very useful

## V. Collecting Keys

You just sent email to your friends telling them of your public key. They got all excited and have returned your email, giving you their public keys. So far, your public keyring has only 1 key—yours. Now we'll put our friends' public keys into our keyring.

Save their email (suppose you save it under MyFriendsKey.asc). Clip any headers that might exist. The file should look something like[2]:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.0.4 (GNU/Linux)
Comment: For info see http://www.gnupg.org

mQGiBDowSm4RBACjBUxe5H7ewiEY54wjT/uL7xaAuWBSB4cYX759puj1QPS7qY
Vy0lMbs4Xs3hLDSwG0HDyYo3zCHss4fAqcbK2Eddr79vtqjbR0pGv4JBsi2bsr
4TIIHoWf6wx5EZk+PVM4XQlPyYDYQ8WAw5X1kKXPR7YDmG1OzRb42D59pwCgko
kX+NVEjnDzZ72pWwFm3ScIOD/R6juFVBTPN+K4ExiDbiIhDSLDwybN7NM2+OFi
AjowSvIACgkQo7kZRWfqlROO7wCcCWBe41kz/VrkDISsJPh/NIROFHoAnORO4o
ZloQQWDG1P6+el4Zof3z
=RUgH
-----END PGP PUBLIC KEY BLOCK-----
```

I'm up to this chapter....

[2] It'll be longer than this. I just want you to get the flavor of what it looks like.

## VI. Signing A File

Why the whole brouhaha over gpg? When you send a file to someone, you use gpg to reassure the recipient that:

1. the file hasn't been tampered with or modified between the time you sent the file and the time the file was received.
2. you were the sender of the file, and not government agency or Muslim *jihad* participant.

Conversely, when someone sends you a file, gpg will reassure you that the file is coming from the person you think it's coming from and also to reassure you that the file hasn't been modified or tampered with in any way.

Suppose you want to send someone a file, and you want them to be sure that the file came from you and was not tampered with. The first step would be to 'sign the file' before you send it through email. Suppose you were sending a file to Colonel Sanders named 'KFC_recipe'. You would sign the file by:

```
gpg --armor -b < KFC_recipe > KFC_recipe.sign
```

gpg will ask you for your secret passphrase and generate a new file named KFC_recipe.sign. By now you should know what `--armor` does. The signature file isn't very long. Go ahead and make a signature of some random file you may have. See what a signature looks like.

You would send both files to Colonel Sanders. The Colonel now holds both the file and the file's signature. He wants to be absolutely sure that the file hasn't been tampered with. He would verify that you were the sender and that the file wasn't modified by:

```
gpg --verify KFC_recipe.sign - < KFC_recipe
```

If the file had been tampered with, or if the file was sent by someone other than you, gpg would get the Colonel know.

## VII. Trust

In the last section, you may have noticed a flaw in some of the reasoning.

First, Colonel Sanders is sure that the KFC_recipe was not tampered with from the moment you generated the signature to the moment he got the file and signature. What if the file was tampered with *before* you generated the signature? Unfortunately, gpg can't do much for that.

Secondly, what the Colonel really knows is that *somebody* signed the file, and that the signature is good. If you think about it, he really doesn't know that the *somebody* is *you*. The email claims that you sent it, but email is very insecure. What if a drug lord from Bogata signed the file and pretended to be you? The Colonel wouldn't know the difference because the signature on the file is good.