Introduction to GnuPG

This page outlines basic setup and use of GnuPG, a much fuller description of the software is available in the official GnuPG manual. This page is only designed to serve as a gentle introduction to the most commonly used commands.

Where examples of commands are given you should only type the section in bold, the other text is included to show you expected or sample output.

Please note that no valid PGP information or e-mail addresss have been used in these examples!

Background

GnuPG is an implementation of PGP (Pretty Good Privacy), which is a form of public key/private key encryption. The strength of the encryption comes from the fact that a file can be encrypted for a given recipient using only the public key, yet both keys are needed in order to decrypt the file. So the idea is to give your public key to your friends and collegues, but keep the private key closely guarded.

With GnuPG the keys are associated with an ID consisting of a name, comment and e-mail address. When specifying recipients you may use either the name or the e-mail adress but due to the complexity of dealing with the spaces in the names we will be using the e-mail address.

The private key is additonally locked with a passphrase which is required to access it. This adds an additonal level of security to prevent someone using your private key if they gain physical access to both your computer and account.

Installation

GnuPG can be downloaded from the GnuPG homepage and you should follow the instructions relating to your own system. This page assumes you are using a Mac OS X or Linux based enviroment and Windows users would have to adjust accordingly (in fact this page would be more appropriate reading).

For installation on a Mac the easiest method is to install Fink and then issue the following command in your terminal:

```
yoda:~ ian$ sudo apt-get install gnupg
```

If you get a message about not being in the sudoers file make sure to add your username into `/etc/sudoers` using the root account. Many programs can be installed in this way with Fink and I highly recommend it for anyone who enjoys the UNIX aspect of OS X.

Verify your installation by typing the following command and checking that a path is returned as follows:

```
yoda:~ ian$ which gpg
/sw/bin/gpg
```

Getting Started, Generating Your Keys

Assuming you have succesfully installed gpg you can go ahead and create a new key pair. This is accomplished by running gpg with the `--gen-key` switch as follows:

```
yoda:~ ian$ gpg --gen-key
gpg (GnuPG) 1.2.4; Copyright (C) 2003 Free Software Foundation, Inc.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions. See the file COPYING for details.

gpg: /Users/ian/.gnupg: directory created
gpg: new configuration file `/Users/ian/.gnupg/gpg.conf' created
gpg: WARNING: options in `/Users/ian/.gnupg/gpg.conf' are not yet active during
gpg: keyring `/Users/ian/.gnupg/secring.gpg' created
gpg: keyring `/Users/ian/.gnupg/pubring.gpg' created
Please select what kind of key you want:
   (1) DSA and ElGamal (default)
   (2) DSA (sign only)
   (4) RSA (sign only)
Your selection? 1
DSA keypair will have 1024 bits.
About to generate a new ELG-E keypair.
              minimum keysize is  768 bits
              default keysize is 1024 bits
    highest suggested keysize is 2048 bits
What keysize do you want? (1024) 1024
Requested keysize is 1024 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) 0
Key does not expire at all
Is this correct (y/n)? y

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

Real name: Ian Atkinson
Email address: ian@atkinson.co.uk
Comment: home
You selected this USER-ID:
    "Ian Atkinson (home) <ian@atkinson.co.uk>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? o
You need a Passphrase to protect your secret key.

Enter passphrase: my passphrase
Enter passphrase: my 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.
++++++++++.++++++++++++++++++++..+++++++++++++
gpg: /Users/ian/.gnupg/trustdb.gpg: trustdb created
public and secret key created and signed.
key marked as ultimately trusted.

pub  1024D/0EB6F689 2006-04-06 Ian Atkinson (home) <ian@atkinson.co.uk>
     Key fingerprint = 908E 6C41 8689 7981 F6B4  CEDF A70C 0200 0EB6 F689
sub  1024g/71445CC9 2006-04-06
```

Throughout this process you can just press enter at any time to accept the default value in brackets. Once again all these options are explained in more detail in the offical manual. For most people the defaults will be fine.

The most important part of your key generation is choosing your passphrase. Your phrase should be something you won't forget (i.e. don't need to write down) and should contain both upper and lower case characters, numbers and punctuation. It should be of a sufficient length but not so long that it's going to take you a lot of attempts to type it in (remember you won't be able to see anything on the screen as you enter it). Something like eight to twelve words is probably about right for most people.

Now that this process is complete you have both a public and a private keyring. A keyring is simply a collection of keys as you might expect!

Your public keyring should just contain your own public key for now, you can view the keyring with the `--list-keys` option:

```
yoda:~ ian$ gpg --list-keys
/Users/ian/.gnupg/pubring.gpg
-----------------------------
pub  1024D/0EB6F689 2006-04-06 Ian Atkinson (home) <ian@atkinson.co.uk>
sub  1024g/71445CC9 2006-04-06
```

If anyone else were to give you their own public keys so that you could send them files, they would appear in here.

You can also check that your private keying contains your own private key with the `--list-secret-keys` option:

```
yoda:~ ian$ gpg --list-secret-keys
/Users/ian/.gnupg/secring.gpg
-----------------------------
sec  1024D/0EB6F689 2006-04-06 Ian Atkinson (home) <ian@atkinson.co.uk>
ssb  1024g/71445CC9 2006-04-06
```

Encrypting a File

Now that we have generated our keys we can go ahead and encrypt a file. For this example we will make a gpgtest directory in our home directory for our test file:

```
yoda:~ ian$ mkdir ~/gpgtest
yoda:~ ian$ cd ~/gpgtest
yoda:~/gpgtest ian$ echo 'this is a secret!!!' >secret.txt
yoda:~/gpgtest ian$ cat secret.txt
this is a secret!!!
```

Here we first create our directory and then change to it, we then create a simple text file containing line ‘this is a secret!!!’ and then test the contents of our file using cat.

For now you can see that the secret file is the only one we have in our directory:

```
yoda:~/gpgtest ian$ ls
secret.txt
```

We will now go ahead and encrypt this file, specifying the recipient as ourselves (remember we can do this because we have our own address in our public keyring).

We will use the `-e` (encrypt) flag of gpg, along with the `-r` (recipient) flag and our recipient and finally the name of the file we wish to encrypt:

```
yoda:~/gpgtest ian$ gpg -e -r ian@atkinson.co.uk secret.txt
gpg: checking the trustdb
gpg: checking at depth 0 signed=0 ot(-/q/n/m/f/u)=0/0/0/0/0/1
yoda:~/gpgtest ian$ ls
secret.txt
secret.txt.gpg
```

As you can see, using the `-e` option gpg has named the encypted file by simply appended .gpg to our clear text file. In many cases this will be suitable as it’s easier to keep track of a file and it’s encrypted equivalent if they have the same root file name. Of course, depending on the level of security you want to obtain, changing the file name and removing the .gpg extension is likely to draw less attention.

If we wanted we could specify the name of the output file by using the `-o` (output) option, for example we could save our encrypted file as encrypted.txt rather than secret.txt.gpg as follows:

```
yoda:~/gpgtest ian$ gpg -e -r ian@atkinson.co.uk -o encrypted.txt secret.txt
```

This file is now encrypted, if you try using the cat command again you should just see a lot of random characters (this may make your terminal go weird though so either take my word for it or be prepared to launch a new one!).

It is worth noting that whilst we have used a plain text file for our example, we can encrypt other types of data of necessary, such as an image.

The encryption process will also hide the file type from the operating system since the headers become encrypted, further helping the privacy of our document. Notice in the following example how, when encypted, the PNG image file is no longer recognised as an image by the system and it simply 'data':

```
yoda:~/gpgtest ian$ file picture.png
picture.png: PNG image data, 1319 x 968, 8-bit/color RGB, non-interlaced
yoda:~/gpgtest ian$ gpg -e -r ian@atkinson.co.uk -o encrypted_picture.png pictur
yoda:~/gpgtest ian$ file encrypted_picture.png
encrypted_picture.png: data
```

Decrypting a File

Now that we have encrypted our file, we can also decrypt it. You may wish to decrypt a file for editing or of couse just to view the contents. We decrypt using the `-d` flag as follows:

```
yoda:~/gpgtest ian$ gpg -d secret.txt.gpg


You need a passphrase to unlock the secret key for
user: "Ian Atkinson (home) <ian@atkinson.co.uk>"
1024-bit ELG-E key, ID 71445CC9, created 2006-04-06 (main key ID 0EB6F689)


Enter passphrase: my passphrase
```

```
gpg: encrypted with 1024-bit ELG-E key, ID 71445CC9, created 2006-04-06
      "Ian Atkinson (home) <ian@atkinson.co.uk>"
this is a secret!!!
```

Gpg will ask us for our passphrase, which you chose when generating your keys. If you type the phrase in successfully then the content of the file will be printed to stdout (the screen).

This isn't usually very useful, and certainly isn't for non text files, so what is more common is to output to a file in the same way as we did when encrypting using the `-o` flag. We can save the decrypted file as notasecret.txt as follows:

```
yoda:~/gpgtest ian$ gpg -d -o notasecret.txt secret.txt.gpg
```

This is all the information you need in order to encrypt and decrypt files for yourself. You can encrypt any of your own sensetive data in this way and keep it safe without learning any further commands.

However, if you wish to encrypt files for other people, or have people encrypt files for you, then you must learn about importing and exporting public keys.

Sharing Keys

The first thing we need to do is export our public key into a text file that we can give to other people to add to their keyrings, this will let them encrypt files before sending them to us. Here we will use the `--export-keys` option along with the `-a` (armor) option.

The armor option will create a plain text file for us, this will be more useful shortly when we come to backing up our keys but for now it will let us see the contents of the keys we're exporting if we wish:

```
yoda:~ ian$ gpg -a -o publickey.txt --export ian@atkinson.co.uk
```

This has now saved our public key into the file publickey.txt. Let us now see how to import this key onto another machine so that that user may encrypt files for us (we could use the same process to add other people's keys to our keyring and encyrpt files for them). First we will see what happens if we try to encrypt the file without first having the correct key on our keyring:

```
ian@chewy:~> echo 'a secret from chewy' >chewysecret.txt
ian@chewy:~> cat chewysecret.txt
a secret from chewy
ian@chewy:~> gpg -e -r ian@atkinson.co.uk chewysecret.txt
gpg: ian@atkinson.co.uk: skipped: public key not found
gpg: chewysecret.txt: encryption failed: public key not found
```

Here we have created our new secret file on another computer, but when we have tried to encrypt the file gpg has told us that we can't as we don't have the key. What we need to do is import the key that we made earlier and try again:

```
ian@chewy:~> gpg --import publickey.txt
gpg: /home/ian/.gnupg/trustdb.gpg: trustdb created
gpg: key 34BEE591: public key "Ian Atkinson <ian@atkinson.co.uk>" imported
gpg: Total number processed: 1
gpg:               imported: 1
```

The next thing we need to do is set up the trust for the key, this tells gpg how much we trust the key owner (see the documentation for more info on the trust model):

```
ian@chewy:~> gpg --edit-key ian@atkinson.co.uk
gpg (GnuPG) 1.4.2; Copyright (C) 2005 Free Software Foundation, Inc.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions. See the file COPYING for details.

pub  1024D/34BEE591  created: 2007-01-01  expires: never       usage: CS
                     trust: unknown       validity: unknown
sub  2048g/F6F6EA8F  created: 2007-01-01  expires: never       usage: E
[ unknown] (1). Ian Atkinson <ian@atkinson.co.uk>

Command> trust
```

```
pub  1024D/34BEE591  created: 2007-01-01  expires: never       usage: CS
                     trust: unknown       validity: unknown
sub  2048g/F6F6EA8F  created: 2007-01-01  expires: never       usage: E
[ unknown] (1). Ian Atkinson <ian@atkinson.co.uk>

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

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

Your decision? 5
Do you really want to set this key to ultimate trust? (y/N) y

pub  1024D/34BEE591  created: 2007-01-01  expires: never       usage: CS
                     trust: ultimate      validity: unknown
sub  2048g/F6F6EA8F  created: 2007-01-01  expires: never       usage: E
[ unknown] (1). Ian Atkinson <ian@atkinson.co.uk>
Please note that the shown key validity is not necessarily correct
unless you restart the program.

Command> quit
```

Now that we have imported the key and set the trust level we can go ahead and encrypt the file without error:

```
ian@chewy:~> gpg -e -r ian@atkinson.co.uk chewysecret.txt
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
```

```
ian@chewy:~> ls chewysecret*
chewysecret.txt  chewysecret.txt.gpg
```

Finally, for completeness, let's go ahead and decrypt this file on the other computer to check everything worked OK (of course, we can't decrypt this file on the machine that we encrypted it on as demonstrated first because we don't have the private key there, only the public key):

```
ian@chewy:~> gpg -d chewysecret.txt.gpg
gpg: encrypted with 2048-bit ELG-E key, ID F6F6EA8F, created 2007-01-01
      "Ian Atkinson <ian@atkinson.co.uk>"
gpg: decryption failed: secret key not available
ian@chewy:~> scp chewysecret.txt.gpg ian@yoda:
Password:
chewysecret.txt.gpg    100%  618     0.6KB/s   00:00
ian@chewy:~> ssh ian@yoda
Password:
Last login: Sat Jan 20 02:13:47 2007 from obi-wan.home
Welcome to Darwin!
yoda:~ ian$ gpg -d chewysecret.txt.gpg

You need a passphrase to unlock the secret key for
user: "Ian Atkinson <ian@atkinson.co.uk>"

Enter passphrase: my passphrase

2048-bit ELG-E key, ID F6F6EA8F, created 2007-01-01 (main key ID 34BEE591)

gpg: encrypted with 2048-bit ELG-E key, ID F6F6EA8F, created 2007-01-01
      "Ian Atkinson <ian@atkinson.co.uk>"
a secret from chewy
```

Backing up and Restoring Your Keys

Imagine if you encrypted a lot of sensetive data and then your computer died and took your keyrings with it, that wouldn't be fun would it! You would have absolutely no feasible way of accessing your data.

For this reason it's a good idea to back up your keys, and to do this you need to know how to export both types of key. We have already seen how to import and export public keys, so all we need to do is learn how to import and export private keys. This is done in much the same way, however you must be incredibly careful with these files as, excepting your passphrase, they are all that is needed to decrypt your data.

So, first we will export both our public key and private key as follows:

```
yoda:~ ian$ gpg --export -a -o mypublickey.txt ian@atkinson.co.uk
yoda:~ ian$ gpg --export-secret-key -a -o myprivatekey.txt ian@atkinson.co.uk
yoda:~ ian$ ls my*
myprivatekey.txt        mypublickey.txt
```

It is now up to you what you do with these two files. At a bare minimum they need both burning to a CD and also printing (this is where the armor option comes in, since they need to be plain text to be printed). In the event of the keys needing to be used they could be imported straight off the CD, or of the media had failed typed in from the printed sheets.

In either case, the media needs to be kept safe and secure.

Let us now imagine that our machine has failed. We get out the CD with these two files on and put them on our new machine, then import the keys as we learnt above (gpg knows whether we are importing a public key or a private key, so we just use `--import` for both):

```
vader:~ ian$ gpg --import myprivatekey.txt
gpg: directory `/Users/ian/.gnupg' created
gpg: new configuration file `/Users/ian/.gnupg/gpg.conf' created
gpg: WARNING: options in `/Users/ian/.gnupg/gpg.conf' are not yet active during
gpg: keyring `/Users/ian/.gnupg/secring.gpg' created
gpg: keyring `/Users/ian/.gnupg/pubring.gpg' created
gpg: key 34BEE591: secret key imported
gpg: /Users/ian/.gnupg/trustdb.gpg: trustdb created
gpg: key 34BEE591: public key "Ian Atkinson <ian@atkinson.co.uk>" imported
```

```
gpg: Total number processed: 1
gpg:               imported: 1
gpg:       secret keys read: 1
gpg:   secret keys imported: 1
vader:~ ian$ gpg --import mypublickey.txt
gpg: key 34BEE591: "Ian Atkinson <ian@atkinson.co.uk>" 1 new signature
gpg: Total number processed: 1
gpg:         new signatures: 1
```

You can see that when we import the first key all of our configuration files and keyrings are created, just the same as they would be if we used `--gen-key` for the first time. All you would need to do now is re-trust your own public key as explained above.

Finally, let's test we can decrypt that file that we made before to ensure we imported the secret key properly:

```
vader:~ ian$ gpg -d chewysecret.txt.gpg

You need a passphrase to unlock the secret key for
user: "Ian Atkinson <ian@atkinson.co.uk>"

Enter passphrase: my passphrase

2048-bit ELG-E key, ID F6F6EA8F, created 2007-01-01 (main key ID 34BEE591)

gpg: encrypted with 2048-bit ELG-E key, ID F6F6EA8F, created 2007-01-01
      "Ian Atkinson <ian@atkinson.co.uk>"
a secret from chewy
```