Securing E-mail communications using GPG

Imed Chihi <ichihi@hexabyte.tn>
June 2004

Internet was designed by computer scientists and for this matter it is radically different from telecom networks. The telecom engineers approach is “I'm the network, I know it all. Ask me what you want”, the computer engineers are more into “I'm the smart computer, get me any dumb transporter and I'll communicate”. With the rising security concerns over the last decade, some are left with the impression that the Internet is an insecure network, well, it is, but that shouldn't matter since the computer is supposed to be intelligent enough to use an insecure and unreliable transporter.

Introduction

When communicating over an insecure network, the major concerns are:

- how to make sure no one can intercept and read my data while transiting over the network?
- how do I know wether a message coming from Bruce Lee was actually sent by Bruce Lee and not by an imposter?
- how do I make sure that whatever I send will reach my correspondant exactly the way I wrote it? What if someone intercepts it, changes something and forwards it again!

Answering all of the above appears to be pretty tough, but thanks to the advances in cryptography, a branch of mathematics, we can do all of it in a fairly easy way. In this short essay we'll try to demonstrate an easy way to use data encryption and signing of e-mail communications using a reputed Open Source tool: GNU Privacy Guard or GPG. GPG is an Open Source implementation of the OpenPGP standard that behaves like PGP, it doesn't include any patented algorithms and its use is, therefore, not subject to any legal hassels.

To understand the workings of GPG we need to go through some theoretical aspects of computer cryptography, this one *is* rocket science, but let's remain on the surface not that I don't want to tell you about the details, but because I ignore them.

Data signing and encryption

E-mail signing and encryption algorithms use a pair of keys: a *public key* and a *secret key*, also called a *private key*. You can simply think of these as two large numbers with some correlation between them, the secret key is to be kept in a safe location accessible only to the owner. The correlation between the two keys makes it possible to decrypt with one what was encrypted with the other. The public key is to be distributed to any party we intend to communicate with securely. The quality of the encryption depends on how hard it will be to guess the secret key from the public key.

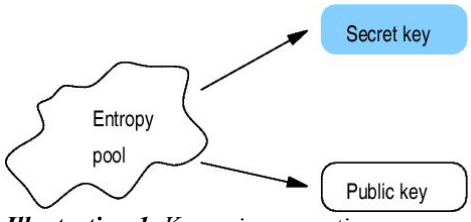

Illustration 1 *Key pair generation*

High quality random numbers in Linux

The quality of the keys, and therefore, the whole encryption, depends greatly on the randomness used for their generation. The higher the randomness, the harder it will be to guess the secret key from the public one.

Linux implements a random number generator that maintains an *entropy pool*. Entropy is, if you remember your thermodynamics course, a measure of disorder and chaos of a system. The randomness fed to the entropy pool is collected from environmental noise like the distribution of timings between key strokes, mouse moves and other IO activities. The returned random numbers are a one-way hash of that entropy pool so that no internal information is revealed to the users.

The random number generator is accessed through two devices `/dev/random` and `/dev/urandom`. The latter is a non blocking version which will return a random number even if the entropy pool doesn't contain enough data.

Some AMD, Via and Intel processors include a hardware Random Number Generator which uses thermal noise generated from inherently random quantum mechanical properties of silicon. For the paranoid, Linux includes a driver (`hw_random`) for some popular chips to generate something inherently random.

Text 1 *Random numbers generators*

By applying a certain function using the secret key to user data we can get it encrypted into an unreadable cipher. The resulting cipher can only be decrypted using the matching secret key. Data encryption is described in illustration 2.

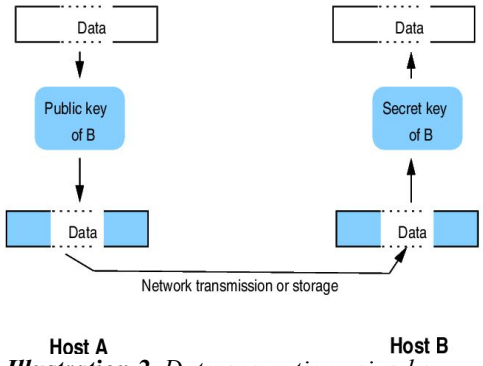

Illustration 2 *Data encryption using key pairs*

Data signing is the encryption of a *hash* calculated from the user data using the sender's secret key. A hash is the result of a hash function ran on a data set, hash functions generate a fixed-size result regardless of the size of the data fed to it. A one-way hash is a hash from which there's, mathematically, no way of generating the original data. For signing purposes, the encrypted hash is sent along with the data and the receiving party can try to decrypt the hash using the sender's public key, recalculate the hash of the received data and compare the two results (cf. illustration 3).

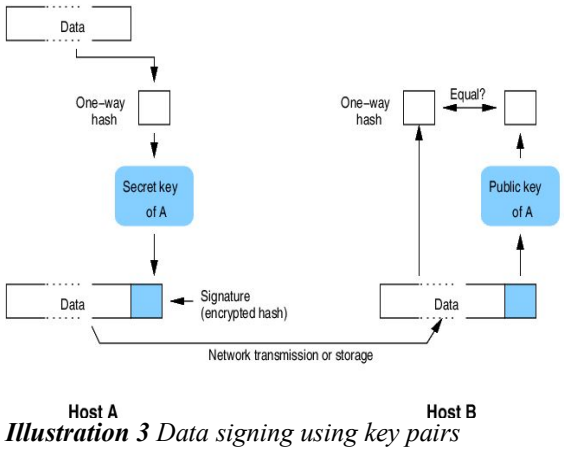

Illustration 3 *Data signing using key pairs*

Kmail and GPG

Now let's see how we can do all this in the real world with real applications. We'll be applying our theory to GPG and Kmail, the KDE mail client, the GPG web site claims that most popular e-mail clients can use GPG.

Step 1 Let's start by generating our keys, this is just a matter of answering some questions. You may do it using the command line tools (`gpg -gen-key`) as outlined in side bar 1. You may also use Kgpg which is a nice GUI to GPG as shown in illustration 3,

Step 2 Enable GPG encryption in Kmail. In Kmail, go to Settings, Configure Kmail, Security, OpenPGP and select “GnuPG”,

Step 3 Set your identity in Kmail (in Settings, Configure Kmail, Identity) and, under the Advanced tab, chose your key from the list,

Step 4 Publish your GPG public key, you may put it on your web site, export it to public key servers using Kgpg or send it as an attachement to your correspondants.

Now you have a working Kmail/GPG set up to sign all your outgoing messages, if you can collect public keys of others then you can start encrypting messages sent to them.

Now let's get back to our target goals we set earlier and see how we can achieve them using this encryption model.

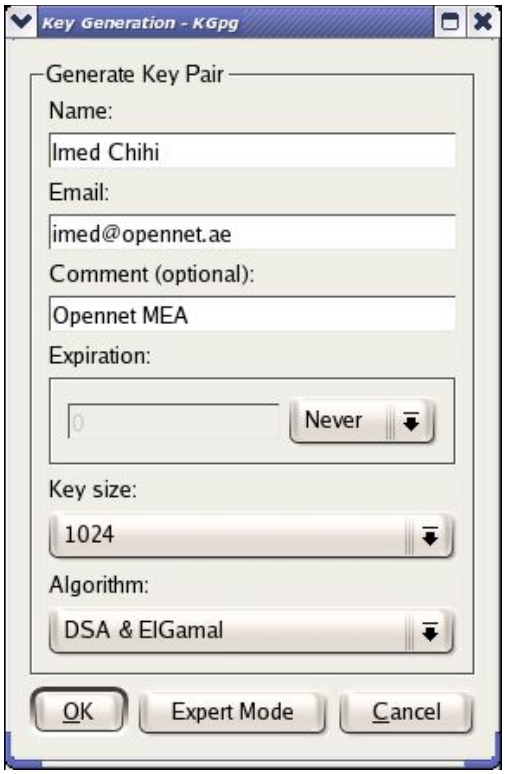

Illustration 4 *Generating a key pair using Kgpg*

- even if someone manages to tap into my network, he won't be able to read the contents of my transmission because all he'll get will be some unreadable, encypted cipher,
- if Jakie Chan sends me a message pretending to be Bruce Lee, then I'll easily notice that, because the public key of Bruce Lee, which I have, won't verify the signed message. Only Bruce Lee has the secret key matching his public key,
- if Bruce Lee sends me a message, and someone intercepts it, changes something in it and forwards it to me, then I'll notice that too since the signature test would fail because the hash calculation would be different,

Illustration 5 *Managing keys with Kgpg*

It's probably very important to notice that in regular e-mail encryption, only the body of the message is encrypted or signed but **none** of the attachements! Attachements need to be encrypted or signed explicitly by the user.

Certificates and certification authorities

All of the above scenarios assume that we know for sure that the public key of Bruce Lee we received, by e-mail say, is really Bruce Lee's. That is what if, while Bruce Lee, is still unaware of encryption, Jakie Chan generates a key pair identifying Bruce Lee and sends it all over the Internet!

It's clear that, in this situation, we need a way to trust the party we are dealing with, but this may not be practical in global networks like the Internet. The way it actually works is by designating some trusworthy parties which will certify the identity of individuals. Usually banks are in a good position the certify the identity of their customers since there must be some trust relationship between them anyway. Besides, the bank would have well identified the customer and met with him personnaly to be dead sure it's Bruce Lee and not an imposter.

These authorities are referred to as Certification Authorities (CA), a CA would sign the public key of the individual being certified using their secret key. The resulting signed public key is called a certificate and client software like web browsers and e-mail clients would have the public keys of reputed CAs and can, therefore, check the authenticity of the public keys.

GPG console session

```
[imed@hannibal imed]# 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: keyring `/home/imed/.gnupg/secring.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: Imed Chihi
Email address: imed@opennet.ae
Comment: Opennet MEA
You selected this USER-ID:
    "Imed Chihi (Opennet MEA) <imed@opennet.ae>"

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

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.
++++++++++.++++++++++.++++++++++++++++..++++++++++.++++++++++++++++.+++++.++++++++++++++++++++
++++++++++++++.+++++.+++++..++++++++++++++++..+++++>++++++++++.......................+++++
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: /home/imed/.gnupg/trustdb.gpg: trustdb created
public and secret key created and signed.
key marked as ultimately trusted.

      pub  1024D/62C90FA1 2004-06-20 Imed Chihi (Opennet MEA) <imed@opennet.ae>
           Key fingerprint = 2F0C 26D8 C6C1 5F06 A057  8858 3A28 5CB0 62C9 0FA1
      sub  1024g/E0CC2EF2 2004-06-20
```

Side bar 1 *Generating a key pair using the command line*