Replacing PGP 2.x with GnuPG

This article is based on an earlier PGP 2.x/GnuPG compatability guide (http://www.toehold.com/~kyle/pgp-compat.html) written by Kyle Hasselbacher (`<kyle@toehold.com>`). Mike Ashley (`<jashley@acm.org>`) edited and expanded it. Michael Fischer v. Mollard (`<mfvm@gmx.de>`) transformed the HTML source to DocBook SGML and also expanded it further. Some of the details described here came from the `gnupg-devel` and `gnupg-user` mailing lists. The workaround for both signing with and encrypting to an RSA key were taken from Gero Treuner's compatability script (http://muppet.faveve.uni-stuttgart.de/~gero/gpg-2comp/changes.html). Please direct questions, bug reports, or suggesstions to the maintainer, Mike Ashley.

1. Introduction

This document describes how to communicate with people still using old versions of PGP 2.x GnuPG can be used as a nearly complete replacement for PGP 2.x. You may encrypt and decrypt PGP 2.x messages using imported old keys, but you cannot generate PGP 2.x keys. This document demonstrates how to extend the standard distribution of GnuPG to support PGP 2.x keys as well as what options must be used to ensure inter-operation with PGP 2.x users. It also warns of anomalies with the interoperation of PGP 2.x and GnuPG.

Note: Using the extension modules `idea.c` and `rsa.c` without licensing the patented algorithms they implement may be illegal. I do not recommend you use these modules. If you have PGP 2.x keys, I suggest you revoke them in favor of new keys and encourage correspondents who use PGP 2.x keys to do the same.

2. Extending GnuPG to support PGP 2.x keys

The standard distribution of GnuPG does not support PGP 2.x keys since PGP 2.x uses IDEA as its symmetric cipher and RSA for its public key cipher. These algorithms are patented[1] and may only be used under certain restrictions. It is a GNU policy not to use patented algorithms, since patents on algorithms are a contradiction to the spirit of free software. Employing these algorithms limits your freedom to use GnuPG as you wish.

It may or may not be legal to use RSA and IDEA without licensing these algorithms. RSA is only patented in the United States, so it is legal to build RSA versions outside of the United States. The extension module for GnuPG is such a version and it may be legally used outside the United States, but it is illegal to use it in the United States. In the United States there exists a reference implementation for RSA called RSAREF, available at ftp.funet.fi (tp://ftp.funet.fi/pub/crypt/cryptography/asymmetric/rsa/rsaref2.tar.gz) or at debian.org (ftp://non-us.debian.org/debian-non-US/dists/stable/non-US/source/rsaref_19930105.orig.tar.gz), that may be used legally without a charge in the USA for non commercial use. Due to export restrictions this code cannot be exported, so there are two ways to integrate RSA into GnuPG: one for the USA and Canada, and one for the rest of the world.

The situation for IDEA is simpler. IDEA is patented in Europe and in the USA, and a patent for Japan is pending. The patent holder, Ascom, grants a non-commerical license (http://www.ascom.ch/infosec/idea/licensing.html) for no charge, but the definition of non-commercial use is rather strict You need to buy a license from Ascom if you want to use IDEA commercially.

To use the extension modules first obtain their source code, `idea.c` and `rsa.c` or `rsaref.c` from the directory of code contributed (ftp://ftp.gnupg.org/pub/gcrypt/contrib/) to GnuPG. Once you have the code, it must be compiled. If you use gcc, you would compile it as follows:

```
alice% gcc -Wall -O2 -shared -fPIC -o idea idea.c
[...]
alice% gcc -Wall -O2 -shared -fPIC -o rsa rsa.c
[...] # or
alice% gcc -Wall -O2 -shared -fPIC -o rsa rsaref.c /usr/lib/rsaref.a
```

The last argument `/usr/lib/rsaref.a` must be replaced with the real location of the RSAREF library on your local machine.

Once compiled, GnuPG must be instructed to load it. This may be done using the option `load-extension`, either on the command line or in the options file although typically it is done in the options file. For example, if you have put `idea` and `rsa` in your `~/.gnupg` directory, in the options file you may add

```
load-extension ~/.gnupg/idea
load-extension ~/.gnupg/rsa
```

If you do not specify an explicit path, GnuPG searches the extension modules in the default GnuPG module directory, which is `/usr/local/lib/gnupg`. If you compiled GnuPG with a different install prefix using `--prefix PREFIX` when you configured your GnuPG source, then the module directory is `PREFIX/lib/gnupg`. Copy the two files 'rsa' and 'idea' into the module directory described above. Make sure everyone can read these files. You do not have to make these files executable as these files are not programs but shared modules.

3. Importing PGP 2.x keys

Once the extensions are loaded it is straightforward to import a PGP 2.x key pair using the option `import`. There are two caveats, however.

- You must not export a private key from PGP 2.x as an ASCII-armored file. Because PGP 2.x predates the OpenPGP specification, the armored message header PGP 2.x uses is not compliant with OpenPGP. Because private key export is rare, GnuPG does not check for the case when the ASCII-armored message is a private key.

- GnuPG expects imported public keys to be self-signed by the corresponding private key. This is a prudent precaution, and both GnuPG and newer versions of PGP self-sign public keys when they are first created. This is not done by PGP 2.x, however. To solve this, you can first self-sign the public key before exporting it from PGP 2.x. Alternatively, you can use the option `allow-non-selfsigned-uid` to force GnuPG to take the key anyway. It is recommended that you self-sign the key either before exporting it or after you have imported it, though, since using a non-self-signed key is a security risk.

```
alice% pgp -kx alice public.pgp
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
[...]
Extracting from key ring: '/u/alice/.pgp/pubring.pgp', userid "alice".
Key for user ID: Alice <alice@cyb.org>
1024-bit key, Key ID 24E2C409, created 1999/09/18

Key extracted to file 'public.pgp'.

alice% pgp -kx alice private.pgp .pgp/secring.pgp
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
[...]

Extracting from key ring: '.pgp/secring.pgp', userid "alice".
Key for user ID: Alice <alice@cyb.org>
1024-bit key, Key ID 24E2C409, created 1999/09/18

Key extracted to file 'private.pgp'.

alice% gpg --import public.pgp
gpg: key 24E2C409: public key imported
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)

alice%gpg --import private.pgp
gpg: key 24E2C409: secret key imported
gpg: Total number processed: 1
gpg:       secret keys read: 1
gpg:   secret keys imported: 1
```

4. Using PGP 2.x keys

An imported public key can be used to encrypt documents to a PGP 2.x private key holder and check signatures made using a PGP 2.x private key. *It is important to realize that it is impossible to use a new OpenPGP key to communicate with an PGP 2.x user, so you must import an old style key if you want to communicate with a PGP 2.x user.*

4.1. Encrypting a document to a PGP 2.x user

Encrypting a document uses several command-line options, and the document to be encrypted must be specified as a file.

```
alice% gpg --rfc1991 --cipher-algo idea --compress-algo 1 --encrypt --recipient alice secret
gpg:
```

```
RSA keys are deprecated; please consider creating a new key and use this key in the future
gpg: this cipher algorithm is depreciated; please use a more standard one!
```

Each of the command-line options are necessary.

- The option `rfc1991` is used to force GnuPG to be more compliant with RFC 1991, which is the old PGP specification implemented by PGP 2.x. If it is omitted, the output from GnuPG will be malformed and unusable by PGP 2.x.
- The option `cipher-algo` specifies the symmetric cipher with which the document is to be encrypted. In the special case of encrypting to a PGP 2.x public key, the cipher specified must be IDEA. If it is omitted, the document will usually be encrypted using 3DES, an algorithm unsupported by PGP 2.x.
- PGP 2.x's compression algorithm motivates how the rest of the command is formed. The option `compress-algo` specifies that GnuPG must use the old zlib compression algorithm used by PGP 2.x. Despite this, GnuPG uses partial length headers when encrypting a stream of unknown size, and this is unsupported by PGP 2.x. The document to be encrypted must therefore be in a file so that GnuPG knows the total size of the document to be encrypted before starting. So unfortunately, you cannot use pipes when using PGP 2.x keys.

4.2. Signing a document for a PGP 2.x user

Signing a document is no different than when any other key is used.

```
alice% gpg --local-user 0x24E2C409 --sign document
You need a passphrase to unlock the secret key for
user: "Alice <alice@cyb.com>"
1024-bit RSA key, ID 24E2C409, created 1999-09-18

gpg: RSA keys are deprecated; please consider creating a new key and use this
key in the future
```

In this example, the option `local-user` is used to specify which private key to use for signing. Also, the output file is `document.gpg`. If the signature is to be verified using PGP 2.x, it must be renamed to a filename with a `.pgp` extension.

4.3. Signing and encrypting a document for a PGP 2.x user

GnuPG does not have native support for both signing a document with an RSA key and encrypting it to an RSA key. GnuPG can be used in a workaround, however, that requires a few steps to implement. The process involves creating a detached signature and then using it to build an encrypted file that can be decrypted and verified using PGP 2.x.

There are four steps. The first creates a detached signature

```
alice% gpg --detach-signature --recipient alice --local-user 0x24E2C409 document

You need a passphrase to unlock the secret key for
user: "Alice <alice@cyb.com>"
1024-bit RSA key, ID 24E2C409, created 1999-09-18

gpg: RSA keys are deprecated; please consider creating a new key and use this
```

```
key in the future
```

The second step converts the document to an internal, literal format that is unencrupted.

```
alice% gpg --store -z 0 --output document.lit document
```

The third step combines the detached signature with the literal document. This is what PGP 2.x uses to verify the signature after decryption.

```
alice% cat Notes.sig Notes.lit | gpg --no-options --no-literal --store --compress-algo 1 --output document.z
gpg: NOTE: --no-literal is not for normal use!
```

The fourth and final step is to use GnuPG to encrypt the combined signature and plaintext to yield an signed and encrypted document that can be decrypted and verified using PGP 2.x.

```
alice% gpg --rfc1991 --cipher-algo idea --no-literal --encrypt --recipient alice --output document.pgp document.z
gpg: NOTE: --no-literal is not for normal use!
gpg: RSA keys are deprecated; please consider creating a new key and use this
key in the future
gpg: this cipher algorithm is depreciated; please use a more standard one!
```

The signed and encrypted document can also be ASCII-armored using the usual options.

```
alice% gpg --rfc1991 --cipher-algo idea --no-literal --encrypt --recipient alice --output document.asc --armor documen
gpg: NOTE: --no-literal is not for normal use!
gpg: RSA keys are deprecated; please consider creating a new key and use this
key in the future
gpg: this cipher algorithm is depreciated; please use a more standard one!
```

4.4. Decrypting a PGP 2.x document

An imported private key may be used to decrypt documents encrypted to the key as well as make signatures using the key. Decrypting a message is no more difficult than when any other key is used.

```
alice% gpg secret.pgp

You need a passphrase to unlock the secret key for
user: "Alice <alice@cyb.org>"
1024-bit RSA key, ID 24E2C409, created 1999-09-18

gpg: NOTE: cipher algorithm 1 not found in preferences
gpg: secret.pgp: unknown suffix
Enter new filename [secret]:
```

Again, the file renaming dialog can be avoided by renaming the input file with a `.gpg` extension. The note emitted by GnuPG regarding cipher algorithm 1 not found in the preferences may be safely ignored if seen.

4.5. Verifying a PGP 2.x signature

Verifying a signature made using a PGP 2.x key is straightforward.

```
alice% gpg document.pgp
gpg: document.pgp: unknown suffix
```

```
Enter new filename [document]:
File ‘document’ exists. Overwrite (y/N)? y
gpg: old style (PGP 2.x) signature
gpg: Signature made Sat Sep 18 17:55:30 1999 EST using RSA key ID 24E2C409
gpg: Good signature from "Alice <alice@cyb.org>"
```

The file renaming dialog can be avoided if the document being verified is renamed with a `.gpg` extension before invoking gpg.

5. Working with clear-text signatures

As of GnuPG release 1.0, there are outstanding issues with respect to passing clear-text signed documents between GnuPG and all versions of PGP. The difficulties appear to be due to implementation deviations from the OpenPGP specification. With respect to PGP 2.x, signatures made by PGP 2.x can be verified using GnuPG and signatures made with GnuPG can be verified using PGP 2.x. The document output from the verification will in both cases differ, however, from the original document. These differences are limited to whitespace, so it should not affect human readability of signed documents. If it is important to maintain complete integrity of the document, you should avoid clear-text signatures.

Notes

1. The RSA patent expires in September 2000. The IDEA patent expires in 2011.