GnuPG Wrapper for Python Documentation

Release 0.3.7

Vinay Sajip

December 07, 2014

Contents

Release 0.3.7

Date December 7, 2014

The gnupg module allows Python programs to make use of the functionality provided by the GNU Privacy Guard (abbreviated GPG or GnuPG). Using this module, Python programs can encrypt and decrypt data, digitally sign documents and verify digital signatures, manage (generate, list and delete) encryption keys, using proven Public Key Infrastructure (PKI) encryption technology based on OpenPGP.

This module is expected to be used with Python versions >= 2.4, as it makes use of the subprocess module which appeared in that version of Python. Development and testing has been carried out on Windows (Python 2.4, 2.5, 2.6, 3.1, Jython 2.5.1), Mac OS X (Python 2.5) and Ubuntu (Python 2.4, 2.5, 2.6, 2.7, 3.0, 3.1, Jython 2.5.1). It should work with more recent versions of Python, too. Install this module using `pip install python-gnupg`.

Deployment Requirements

Apart from a recent-enough version of Python, in order to use this module you need to have access to a compatible version of the GnuPG executable. The system has been tested with GnuPG v1.4.9 on Windows and Ubuntu. On a Linux platform, this will typically be installed via your distribution's package manager (e.g. `apt-get` on Debian/Ubuntu). Windows binaries are available here – use one of the `gnupg-w32cli-1.4.x.exe` installers for the simplest deployment options.

Note: On Windows, it is *not* necessary to perform a full installation of GnuPG, using the standard installer, on each computer: it is normally sufficient to distribute only the executable, `gpg.exe`, and a DLL which it depends on, `iconv.dll`. These files do not need to be placed in system directories, nor are registry changes needed. The files need to be placed in a location such that implicit invocation will find them - such as the working directory of the application which uses the `gnupg` module, or on the system path if that is appropriate for your requirements. Alternatively, you can specify the full path to the `gpg` executable. *Note, however, that if you want to use GnuPG 2.0, then this simple deployment approach may not work, because there are more dependent files which you have to ship. For this reason, our recommendation is to stick with GnuPG 1.4.x on Windows, unless you specifically need 2.0 features - in which case, you may have to do a full installation rather than just relying on a couple of files).*

CHAPTER 2

Acknowledgements

This module is based on an earlier version, `GPG.py`, written by Andrew Kuchling. This was further improved by Richard Jones, and then even further by Steve Traugott. The `gnupg` module is derived from Steve Traugott's module, and uses Python's `subprocess` module to communicate with the GnuPG executable, which it uses to spawn a subprocess to do the real work.

I've gratefully incorporated improvements contributed or suggested by:

- Paul Cunnane (detached signature support)
- Daniel Folkinshteyn (`recv_keys`, handling of subkeys and SIGEXPIRED, KEYEXPIRED while verifying, EXPKEYSIG, REVKEYSIG)
- Dmitry Gladkov (handle KEYEXPIRED when importing)
- Abdul Karim (keyring patch)
- Yann Leboulanger (handle ERRSIG and NO_PUBKEY while verifying, get subkeys)
- Kirill Yakovenko (RSA and IDEA support)
- Robert Leftwich (handle INV_SGNR, KEY_NOT_CREATED)
- Michal Niklas (Trust levels for signature verification)
- David Noël (`search_keys`, `send_keys` functionality)
- David Andersen (handle UNEXPECTED during verification)
- Jannis Leidel (output signature to a file)
- Venzen Khaosan (scan_keys functionality)

and Google Code users

- dprovins (ListKeys handle_status)
- ernest0x (improved support for non-ASCII input)
- eyepulp (additional options for encryption/decryption)
- hysterix.is.slackin (symmetric encryption support)
- natureshadow (improved status handling when smart cards in use)

(If I've missed anyone from this list, please let me know.)

CHAPTER 3

Before you Start

GnuPG works on the basis of a “home directory” which is used to store public and private keyring files as well as a trust database. You need to identify in advance which directory on the end-user system will be used as the home directory, as you will need to pass this information to `gnupg`.

CHAPTER 4

Getting Started

You interface to the GnuPG functionality through an instance of the GPG class:

```
>>> gpg = gnupg.GPG(gnupghome='/path/to/home/directory')
```

If the home directory does not exist, it will be created (including any missing parent directories). Thereafter, all the operations available are accessed via methods of this instance. If the gnupghome parameter is omitted, GnuPG will use whatever directory is the default (consult the GnuPG documentation for more information on what this might be).

The GPG() constructor also accepts the following additional optional keyword arguments:

gpgbinary (defaults to "gpg") The path to the gpg executable.

verbose (defaults to False) Print information (e.g. the gpg command lines, and status messages returned by gpg) to the console. You don't generally need to set this option, since the module uses Python's logging package to provide more flexible functionality. The status messages from GPG are quite voluminous, especially during key generation.

use_agent (defaults to False) If specified as True, the --use-agent parameter is passed to GPG, asking it to use any in-memory GPG agent (which remembers your credentials).

keyring (defaults to None) If specified, the value is used as the name of the keyring file. The default keyring is not used. A list of paths to keyring files can also be specified.

options (defaults to None) If specified, the value should be a list of additional command-line options to pass to GPG.

secret_keyring (defaults to None) If specified, the value is used as the name of the secret keyring file. A list of paths to secret keyring files can also be specified.

Changed in version 0.3.4: The keyring argument can now also be a list of keyring filenames.

New in version 0.3.4: The secret_keyring argument was added.

Note: If you specify values in options, make sure you don't specify values which will conflict with other values added by python-gnupg. You should be familiar with GPG command-line arguments and how they affect GPG's operation.

Changed in version 0.3.7: The default encoding was changed to latin-1. In earlier versions, it was either locale.getpreferredencoding() or, failing that, sys.stdin.encoding, and failing that, utf-8.

If the gpgbinary executable cannot be found, a ValueError is raised in GPG.__init__().

The low-level communication between the gpg executable and python-gnupg is in terms of bytes, and python-gnupg tries to convert gpg's stderr stream to text using an encoding. The default value of this is latin-1, but you can override this by setting the encoding name in the GPG instance's encoding attribute after instantiation, like this:

```
>>> gpg = gnupg.GPG(gnupghome='/path/to/home/directory')
>>> gpg.encoding = 'utf-8'
```

Note: If you use the wrong encoding, you may get exceptions. The `'latin-1'` encoding leaves bytes as-is and shouldn't fail with encoding/decoding errors, though it may not decode text correctly (so you may see odd characters in the decoding output). The `gpg` executable will use an output encoding based on your environment settings (e.g. environment variables, code page etc.) but defaults to latin-1.

CHAPTER 5

Key Management

The module provides functionality for generating (creating) keys, listing keys, deleting keys, and importing and exporting keys.

5.1 Generating keys

The first thing you typically want to do when starting with a PKI framework is to generate some keys. You can do this as follows:

```
>>> key = gpg.gen_key(input_data)
```

where input_data is a special command string which tells GnuPG the parameters you want to use when creating the key. To make life easier, a helper method is provided which takes keyword arguments which allow you to specify individual parameters of the key, as in the following example:

```
>>> input_data = gpg.gen_key_input(key_type="RSA", key_length=1024)
```

Sensible defaults are provided for parameters which you don't specify, as shown in the following table:

Parameter	Keyword Argument	Default value	Example values	Meaning of parameter
Key-Type	key_type	"RSA"	"RSA", "DSA"	The type of the primary key to generate. It must be capable of signing.
Key-Length	key_length	1024	1024, 2048	The length of the primary key in bits.
Name-Real	name_real	"Autogenerated Key"	"Fred Bloggs"	The real name of the user identity which is represented by the key.
Name-Comment	name_comment	"Generated by gnupg.py"	"A test user"	A comment to attach to the user id.
Name-Email	name_email	<username>@<hostname>	“fred.bloggs@domain.com“	An email address for the user.

If you don't specify any parameters, the values in the table above will be used with the defaults indicated. There is a whole set of other parameters you can specify; see this GnuPG document for more details. While use of RSA keys is common (they can be used for both signing and encryption), another popular option is to use a DSA primary key (for signing) together with a secondary El-Gamal key (for encryption). For this latter option, you could supply the following additional parameters:

Parameter	Keyword Argument	Example values	Meaning of parameter
Subkey-Type	subkey_type	“RSA”, “ELG-E”	The type of the secondary key to generate.
Subkey-Length	subkey_length	1024, 2048	The length of the secondary key in bits.
Expire-Date	expire_date	“2009-12-31”, “365d”, “3m”, “6w”, “5y”, “seconds=<epoch>”, 0	The expiration date for the primary and any secondary key. You can specify an ISO date, A number of days/weeks/months/years, an epoch value, or 0 for a non-expiring key.
Passphrase	passphrase	“secret”	The passphrase to use. If this parameter is not specified, no passphrase is needed to access the key.

Whatever keyword arguments you pass to `gen_key_input()` will be converted to the parameters expected by GnuPG by replacing underscores with hyphens and title-casing the result. You can of course construct the parameters in your own dictionary `params` and then pass it as follows:

```
>>> input_data = gpg.gen_key_input(**params)
```

5.1.1 Performance Issues

Key generation requires the system to work with a source of random numbers. Systems which are better at generating random numbers than others are said to have higher *entropy*. This is typically obtained from the system hardware; the GnuPG documentation recommends that keys be generated *only* on a local machine (i.e. not one being accessed across a network), and that keyboard, mouse and disk activity be maximised during key generation to increase the entropy of the system.

Unfortunately, there are some scenarios - for example, on virtual machines which don’t have real hardware - where insufficient entropy causes key generation to be *extremely* slow. If you come across this problem, you should investigate means of increasing the system entropy. On virtualised Linux systems, this can often be achieved by installing the `rng-tools` package. This is available at least on RPM-based and APT-based systems (Red Hat/Fedora, Debian, Ubuntu and derivative distributions).

5.2 Exporting keys

To export keys, use the `export_keys()` method:

```
>>> ascii_armored_public_keys = gpg.export_keys(keyids) # same as gpg.export_keys(keyids, False)
>>> ascii_armored_private_keys = gpg.export_keys(keyids, True) # True => private keys
```

For the `keyids` parameter, you can use a sequence of anything which GnuPG itself accepts to identify a key - for example, the keyid or the fingerprint could be used. If you want to pass a single keyid, then you can just pass in a string which identifies the key.

The `export_keys` method has two additional keyword arguments: `armor` (defaulting to `True`) and `minimal` (defaulting to `False`). When `True`, these pass `--armor` and `--export-options export-minimal`, respectively, to `gpg`.

New in version 0.3.7: The `armor` and `minimal` keyword arguments were added.

5.3 Importing and receiving keys

To import keys, get the key data as an ASCII string, say `key_data`. Then:

```
>>> import_result = gpg.import_keys(key_data)
```

This will import all the keys in `key_data`. The number of keys imported will be available in `import_result.count` and the fingerprints of the imported keys will be in `import_result.fingerprints`.

To receive keys from a keyserver, use:

```
>>> import_result = gpg.recv_keys('server-name', 'keyid1', 'keyid2', ...)
```

This will fetch keys with all specified keyids and import them. Note that on Windows, you may require helper programs such as `gpg_hkp.exe`, distributed with GnuPG, to successfully run `recv_keys`. On Jython, security permissions may lead to failure of `recv_keys`.

5.4 Listing keys

Now that we've seen how to generate, import and export keys, let's move on to finding which keys we have in our keyrings. This is fairly straightforward:

```
>>> public_keys = gpg.list_keys() # same as gpg.list_keys(False)
>>> private_keys = gpg.list_keys(True) # True => private keys
```

The returned value from `list_keys()` is a subclass of Python's `list` class. Each entry represents one key and is a Python dictionary which contains useful information about the corresponding key.

5.5 Scanning keys

We can also scan keys in files without importing them into a local keyring, as follows:

```
>>> keys = gpg.scan_keys(key_file_name)
```

The returned value from `scan_keys()` has the same format as for `list_keys()`.

New in version 0.3.7: The `scan_keys` method was added.

5.6 Deleting keys

To delete keys, their key identifiers must be specified. If a public/private keypair has been created, a private key needs to be deleted before the public key can be deleted:

```
>>> key = gpg.gen_key(gpg.gen_key_input())
>>> fp = key.fingerprint
>>> str(gpg.delete_keys(fp)) # same as gpg.delete_keys(fp, False)
'Must delete secret key first'
>>> str(gpg.delete_keys(fp, True))# True => private keys
'ok'
>>> str(gpg.delete_keys(fp))
'ok'
```

```
>>> str(gpg.delete_keys("nosuchkey"))
'No such key'
```

The argument you pass to `delete_keys()` can be either a single key identifier (e.g. keyid or fingerprint) or a sequence of key identifiers.

5.7 Searching for keys

You can search for keys by passing a search query and optionally a keyserver name. If no keyserver is specified, `pgp.mit.edu` is used. A list of dictionaries describing keys that were found is returned (this list could be empty). For example:

```
>>> gpg.search_keys('vinay_sajip@hotmail.com', 'keyserver.ubuntu.com')
[{'keyid': u'92905378', 'uids': [u'Vinay Sajip <vinay_sajip@hotmail.com>'], 'expires': u'', 'length'
```

New in version 0.3.5: The `search_keys` method was added.

5.8 Sending keys

You can send keys to a keyserver by passing its name and some key identifiers. For example:

```
>>> gpg.send_keys('keyserver.ubuntu.com', '6E4D5A2B')
<gnupg.SendResult object at 0xb74d55ac>
```

New in version 0.3.5: The `send_keys` method was added.

CHAPTER 6

Encryption and Decryption

Data intended for some particular recipients is encrypted with the public keys of those recipients. Each recipient can decrypt the encrypted data using the corresponding private key.

6.1 Encryption

To encrypt a message, use the following approach:

```
>>> encrypted_ascii_data = gpg.encrypt(data, recipients)
```

If you want to encrypt data in a file (or file-like object), use:

```
>>> encrypted_ascii_data = gpg.encrypt_file(stream, recipients) # e.g. after stream = open(filename,
```

These methods both return an object such that `str(encrypted_ascii_data)` gives the encrypted data in a non-binary format.

In both cases, `recipients` is a list of key fingerprints for those recipients. For your convenience, if there is a single recipient, you can pass the fingerprint rather than a 1-element array containing the fingerprint. Both methods accept the following optional keyword arguments:

sign (defaults to `None`) Either the Boolean value `True`, or the fingerprint of a key which is used to sign the encrypted data. If `True` is specified, the default key is used for signing. When not specified, the data is not signed.

always_trust (defaults to `False`) Skip key validation and assume that used keys are always fully trusted.

passphrase (defaults to `None`) A passphrase to use when accessing the keyrings.

symmetric (defaults to `False`) If specified, symmetric encryption is used. In this case, specify recipients as `None`. If `True` is specified, then the default cipher algorithm (`CAST5`) is used. Starting with version 0.3.5, you can also specify the cipher-algorithm to use (for example, `'AES256'`). Check your `gpg` command line help to see what symmetric cipher algorithms are supported. Note that the default (`CAST5`) may not be the best available.

Changed in version 0.3.5: A string can be passed for the `symmetric` argument, as well as `True` or `False`. If a string is passed, it should be a symmetric cipher algorithm supported by the `gpg` you are using.

The `encrypt_file` method takes the following additional keyword arguments:

armor (defaults to `True`) Whether to use ASCII armor. If `False`, binary data is produced.

output (defaults to `None`) The name of an output file to write to. If a name is specified, the encrypted output is written directly to the file.

Note: Any public key provided for encryption should be trusted, otherwise encryption fails but without any warning.

This is because gpg just prints a message to the console, but does not provide a specific error indication that the Python wrapper can use.

6.2 Decryption

To decrypt a message, use the following approach:

```
>>> decrypted_data = gpg.decrypt(data)
```

If you want to decrypt data in a file (or file-like object), use:

```
>>> decrypted_data = gpg.decrypt_file(stream) # e.g. after stream = open(filename, "rb")
```

These methods both return an object such that `str(decrypted_data)` gives the decrypted data in a non-binary format.

Both methods accept the following optional keyword arguments:

always_trust (defaults to `False`) Skip key validation and assume that used keys are always fully trusted.

passphrase (defaults to `None`) A passphrase to use when accessing the keyrings.

The `decrypt_file` method takes the following additional keyword argument:

output (defaults to `None`) The name of an output file to write to. If a name is specified, the decrypted output is written directly to the file.

6.3 Using signing and encryption together

If you want to use signing and encryption together, use the following approach:

```
>>> encrypted_data = gpg.encrypt(data, recipients, sign=signer_fingerprint, passphrase=signer_passph
```

The resulting encrypted data contains the signature. When decrypting the data, upon successful decryption, signature verification is also performed (assuming the relevant public keys are available at the recipient end). The results are stored in the object returned from the decrypt call:

```
>>> decrypted_data = gpg.decrypt(data, passphrase=recipient_passphrase)
```

At this point, if a signature is verified, signer information is held in attributes of `decrpyted_data`: `username`, `key_id`, `signature_id`, `fingerprint`, `trust_level` and `trust_text`. If the message wasn't signed, these attributes will all be set to `None`.

The trust levels are (in increasing order) `TRUST_UNDEFINED`, `TRUST_NEVER`, `TRUST_MARGINAL`, `TRUST_FULLY` and `TRUST_ULTIMATE`. If verification succeeded, you can test the trust level against known values as in the following example:

```
decrypted_data = gpg.decrypt(data, passphrase=recipient_passphrase))
if decrypted_data.trust_level is not None and decrypted_data.trust_level >= decrypted_data.TRUST_FULL
    print('Trust level: %s' % decrypted_data.trust_text)
```

New in version 0.3.1: The `trust_level` and `trust_text` attributes were added.

CHAPTER 7

Signing and Verification

Data intended for digital signing is signed with the private key of the signer. Each recipient can verify the signed data using the corresponding public key.

7.1 Signing

To sign a message, do the following:

```
>>> signed_data = gpg.sign(message)
```

or, for data in a file (or file-like object), you can do:

```
>>> signed_data = gpg.sign_file(stream) # e.g. after stream = open(filename, "rb")
```

These methods both return an object such that `str(signed_data)` gives the signed data in a non-binary format. They accept the following optional keyword arguments:

keyid (defaults to `None`) The id for the key which will be used to do the signing. If not specified, the first key in the secret keyring is used.

passphrase (defaults to `None`) A passphrase to use when accessing the keyrings.

clearsign (defaults to `True`) Returns a clear text signature, i.e. one which can be read without any special software.

detach (defaults to `False`) Returns a detached signature. If you specify `True` for this, then the detached signature will not be clear text, i.e. it will be as if you had specified a `False` value for *clearsign*. This is because if both are specified, gpg ignores the request for a detached signature.

binary (defaults to `False`) If `True`, a binary signature (rather than armored ASCII) is created.

output (defaults to `None`) If specified, this is used as the file path where GPG outputs the signature. Convention dictates a `.asc` or `.sig` file extension for this.

Note: If the data being signed is binary, calling `str(signed_data)` may raise exceptions. In that case, use the fact that `signed_data.data` holds the binary signed data. Usually the signature itself is ASCII; it's the message itself which may cause the exceptions to be raised. (Unless a detached signature is requested, the result of signing is the message with the signature appended.)

The hash algorihm used when creating the signature can be found in the `signed_data.hash_algo` attribute.

The `detach` keyword argument was added in version 0.2.5.

The `binary` keyword argument was added in version 0.2.6.

The `output` keyword argument was added in version 0.3.7.

7.2 Verification

To verify some data which you've received, do the following:

```
>>> verified = gpg.verify(data)
```

To verify data in a file (or file-like object), use:

```
>>> verified = gpg.verify_file(stream) # e.g. after stream = open(filename, "rb")
```

You can use the returned value in a Boolean context:

```
>>> if not verified: raise ValueError("Signature could not be verified!")
```

7.2.1 Verifying detached signatures on disk

If you want to verify a detached signature, use the following approach:

```
>>> verified = gpg.verify_file(stream, path_to_data_file)
```

Note that in this case, the stream contains the *signature* to be verified. The data that was signed should be in a separate file whose path is indicated by path_to_data_file.

New in version 0.2.5: The second argument to verify_file (data_filename) was added.

When a signature is verified, signer information is held in attributes of verified: username, key_id, signature_id, fingerprint, trust_level and trust_text. If the message wasn't signed, these attributes will all be set to None.

The trust levels are (in increasing order) TRUST_UNDEFINED, TRUST_NEVER, TRUST_MARGINAL, TRUST_FULLY and TRUST_ULTIMATE. If verification succeeded, you can test the trust level against known values as in the following example:

```
verified = gpg.verify(data)
if verified.trust_level is not None and verified.trust_level >= verified.TRUST_FULLY:
    print('Trust level: %s' % verified.trust_text)
```

New in version 0.3.1: The trust_level and trust_text attributes were added.

Note that even if you have a valid signature, you may want to not rely on that validity, if the key used for signing has expired or was revoked. If this information is available, it will be in the key_status attribute =, and the result will still be False in a Boolean context. If there is no problem detected with the signing key, the key_status attribute will be None.

New in version 0.3.3: The key_status attribute was added.

7.2.2 Verifying detached signatures in memory

You can also verify detached signatures where the data is in memory, using:

```
>>> verified = gpg.verify_data(path_to_signature_file, data)
```

where *data* should be a byte string of the data to be verified against the signature in the file named by *path_to_signature_file*. The returned value is the same as for the other verification methods.

New in version 0.3.6: The verify_data method was added.

CHAPTER 8

Passphrases

Passphrases provided to python-gnupg are not stored persistently, and just passed through to the GnuPG executable through a pipe. The user of python-gnupg is responsible for taking care not to store passphrases where they may become available to malicious code or malicious users, as well as the physical and security aspects of managing their private keys.

CHAPTER 9

Logging

The module makes use of the facilities provided by Python's logging package. A single logger is created with the module's __name__, hence gnupg unless you rename the module. A NullHandler instance is added to this logger, so if you don't use logging in your application which uses this module, you shouldn't see any logging messages. If you do use logging in your application, just configure it in the normal way.

CHAPTER 10

Test Harness

The distribution includes a test harness, `test_gnupg.py`, which contains unit tests (with integrated doctests) covering the functionality described above. You can invoke `test_gnupg.py` with one or more optional command-line arguments. If no arguments are provided, all tests are run. If arguments are provided, they collectively determine which of the tests will be run:

doc Run doctests only (they cover most of the functionality of the module)

crypt Run tests relating to encryption and decryption

sign Run tests relating to signing and verification

key Run tests relating to key management

basic Run basic tests relating to environment setup, or which don't fit into one of the above categories

CHAPTER 11

Download

Here is the current version, 0.3.7, in tarball and Windows formats.

CHAPTER 12

Status and Further Work

The gnupg module, being based on proven earlier versions, is quite usable. However, there are many features of GnuPG which this module does not take advantage of, or provide access to. How this module evolves will be determined by feedback from the user community.

If you find bugs and want to raise issues, please do so via the BitBucket project.

All feedback will be gratefully received; please send it to the discussion group.

CHAPTER 13

Indices and tables

- *genindex*

Python Module Index

Index