PyGPGME Documentation

Release 0.3

PyGPGME contributors

Mar 26, 2018

Contents

Contents:

CHAPTER 1

PyGPGME Cookbook

The following recipes illustrate typical use cases of PyGPGME. For a detailed documentation of the individual classes and methods please refer to the *API documentation*.

1.1 Listing All Keys

Use `Context.keylist()` without arguments to get all keys:

```
from __future__ import print_function
import gpgme

c = gpgme.Context()
for key in c.keylist():
    user = key.uids[0]
    print("Keys for %s (%s):" % (user.name, user.email))
    for subkey in key.subkeys:
        features = []
        if subkey.can_authenticate:
            features.append('auth')
        if subkey.can_certify:
            features.append('cert')
        if subkey.can_encrypt:
            features.append('encrypt')
        if subkey.can_sign:
            features.append('sign')
        print('  %s %s' %(subkey.fpr, ','.join(features)))
```

1.2 Searching for a Specific Key

To search for a key using parts of the key owner's name or e-mail address, pass a query to *gpgme.Context.keylist()*:

```
from __future__ import print_function
import gpgme

c = gpgme.Context()
for key in c.keylist('john'):
    print(key.subkeys[0].fpr)
```

To get a key via its fingerprint, use *gpgme.Context.get_key()* instead (note that you must pass the full fingerprint):

```
from __future__ import print_function
import gpgme

c = gpgme.Context()
fingerprint = 'key fingerprint to search for'
try:
    key = c.get_key(fingerprint)
    print('%s (%s)' % (key.uids[0].name, key.uids[0].email))
except gpgme.GpgmeError:
    print("No key for fingerprint '%s'." % fingerprint)
```

1.3 Encrypting and Decrypting Files

By default, *gpgme.Context.encrypt()* returns the encrypted data in binary form, so make sure to open the ciphertext files in binary mode:

```
import gpgme

c = gpgme.Context()
recipient = c.get_key("fingerprint of recipient's key")

# Encrypt
with open('foo.txt', 'r') as input_file:
    with open('foo.txt.gpg', 'wb') as output_file:
        c.encrypt([recipient], 0, input_file, output_file)

# Decrypt
with open('foo.txt.gpg', 'rb') as input_file:
    with open('foo2.txt', 'w') as output_file:
        c.decrypt(input_file, output_file)
```

If you set *gpgme.Context.armor* to `True` then the ciphertext is encoded in a so-called ASCII-armor string. In that case, the ciphertext file should be opened in text mode.

The example above uses asymmetric encryption, i.e. the data is encrypted using a public key and can only be decrypted using the corresponding private key. If you want to use symmetric encryption instead (where encryption and decryption use the same passphrase) then pass `None` as the first argument to *gpgme.Context.encrypt()*. In that case you will be prompted for the passphrase.

1.4 Encrypting and Decrypting Bytes and Strings

gpgme.Context.encrypt() and *gpgme.Context.decrypt()* operate on streams of data (i.e. file-like objects). If you want to encrypt or decrypt data from bytes variables instead then you need to wrap them in a suitable

buffer (e.g. `io.BytesIO`):

```
import io
import gpgme

c = gpgme.Context()
recipient = c.get_key("fingerprint of recipient's key")

plaintext_bytes = io.BytesIO(b'plain binary data')
encrypted_bytes = io.BytesIO()
c.encrypt([recipient], 0, plaintext_bytes, encrypted_bytes)

encrypted_bytes.seek(0)  # Return file pointer to beginning of file

decrypted_bytes = io.BytesIO()
c.decrypt(encrypted_bytes, decrypted_bytes)

assert decrypted_bytes.getvalue() == plaintext_bytes.getvalue()
```

Note that *gpgme.Context.encrypt()* only accepts binary buffers – passing text buffers like `io.StringIO` raises *gpgme.GpgmeError*. To encrypt string data, you therefore need to encode it to binary first:

```
import io
import gpgme

c = gpgme.Context()
recipient = c.get_key("fingerprint of recipient's key")

plaintext_string = u'plain text data'
plaintext_bytes = io.BytesIO(plaintext_string.encode('utf8'))
encrypted_bytes = io.BytesIO()
c.encrypt([recipient], 0, plaintext_bytes, encrypted_bytes)

encrypted_bytes.seek(0)  # Return file pointer to beginning of file

decrypted_bytes = io.BytesIO()
c.decrypt(encrypted_bytes, decrypted_bytes)
decrypted_string = decrypted_bytes.getvalue().decode('utf8')

assert decrypted_string == plaintext_string
```

Even if *gpgme.Context.armor* is true and the encrypted output is text you still need to use binary buffers. That is not a problem, however, since the armor uses plain ASCII:

```
from __future__ import print_function

import io
import gpgme

c = gpgme.Context()
recipient = c.get_key("fingerprint of recipient's key")
c.armor = True  # Use ASCII-armor output

plaintext_string = u'plain text data'
plaintext_bytes = io.BytesIO(plaintext_string.encode('utf8'))
encrypted_bytes = io.BytesIO()
c.encrypt([recipient], 0, plaintext_bytes, encrypted_bytes)
encrypted_string = encrypted_bytes.getvalue().decode('ascii')
```

(continues on next page)

(continued from previous page)

```
print(encrypted_string)  # Display ASCII armored ciphertext

# Re-initialize encrypted bytes data from ASCII armor
encrypted_bytes = io.BytesIO(encrypted_string.encode('ascii'))

decrypted_bytes = io.BytesIO()
c.decrypt(encrypted_bytes, decrypted_bytes)
decrypted_string = decrypted_bytes.getvalue().decode('utf8')

assert decrypted_string == plaintext_string
```

1.5 Signing

FIXME

1.6 Verifying a Signature

FIXME

1.7 Generating Keys

FIXME

1.8 Using a Passphrase Callback

FIXME

1.9 Using a Different GPG Base Directory

FIXME

CHAPTER 2

PyGPGME API

2.1 Context

class gpgme.**Context**

Configuration and internal state for cryptographic operations.

This is the main class of *gpgme*. The constructor takes no arguments:

```
ctx = gpgme.Context()
```

armor

Property indicating whether output should be ASCII-armored or not. Used by *Context.encrypt()*, *Context.encrypt_sign()*, and *Context.sign()*.

card_edit()

decrypt(*ciphertext*, *plaintext*)

Decrypts the ciphertext and writes out the plaintext.

To decrypt data, you must have one of the recipients' private keys in your keyring (for public key encryption) or the passphrase (for symmetric encryption). If gpg finds the key but needs a passphrase to unlock it, the .passphrase_cb callback will be used to ask for it.

Parameters

- **ciphertext** – A file-like object opened for reading, containing the encrypted data.
- **plaintext** – A file-like object opened for writing, where the decrypted data will be written.

See also *Context.decrypt_verify()* and *Context.encrypt()*.

decrypt_verify(*ciphertext*, *plaintext*)

Decrypt ciphertext and verify signatures.

Like *Context.decrypt()*, but also checks the signatures of the ciphertext.

Returns A list of *Signature* instances (one for each key that was used in the signature). Note that you need to inspect the return value to check whether the signatures are valid – a syntactically correct but invalid signature does not raise an error!

See also *Context.encrypt_sign()*.

delete(*key*, *allow_secret=False*)

edit()

encrypt(*recipients*, *flags*, *plaintext*, *ciphertext*)
Encrypts plaintext so it can only be read by the given recipients.

Parameters

- **recipients** – A list of Key objects. Only people in possession of the corresponding private key (for public key encryption) or passphrase (for symmetric encryption) will be able to decrypt the result.
- **flags** – A bitwise OR combination of ENCRYPT_* constants.
- **plaintext** – A file-like object opened for reading, containing the data to be encrypted.
- **ciphertext** – A file-like object opened for writing, where the encrypted data will be written. If *Context.armor* is false then this file should be opened in binary mode.

See also *Context.encrypt_sign()* and *Context.decrypt()*.

encrypt_sign(*recipients*, *flags*, *plaintext*, *ciphertext*)
Encrypt and sign plaintext.

Works like *Context.encrypt()*, but the ciphertext is also signed using all keys listed in *Context.signers*.

Returns A list of *NewSignature* instances (one for each key in *Context.signers*).

See also *Context.decrypt_verify()*.

export()

genkey(*params*, *public=None*, *secret=None*)
Generate a new key pair.

The functionality of this method depends on the crypto backend set via *Context.protocol*. This documentation only covers PGP/GPG (i.e. *PROTOCOL_OpenPGP*).

The generated key pair is automatically added to the key ring. Use *Context.set_engine_info()* to configure the location of the key ring files.

Parameters

- **params** – A string containing the parameters for key generation. The general syntax is as follows:

```
<GnupgKeyParms format="internal">
    Key-Type: RSA
    Key-Length: 2048
    Name-Real: Jim Joe
    Passphrase: secret passphrase
    Expire-Date: 0
</GnupgKeyParms>
```

For a detailed listing of the available options please refer to the GPG key generation documentation.

- **public** – Must be None.
- **secret** – Must be None.

Returns An instance of *gpgme.GenkeyResult*.

get_key (*fingerprint*, *secret=False*)
Finds a key with the given fingerprint (a string of hex digits) in the user's keyring.

Parameters

- **fingerprint** – Fingerprint of the key to look for
- **secret** – If true, only private keys will be returned.

If no key can be found, raises *GpgmeError*.

Returns A *Key* instance.

import_()

include_certs()

keylist (*query=None*, *secret=False*)
Searches for keys matching the given pattern(s).

Parameters

- **query** – If None or not supplied, the *KeyIter* fetches all available keys. If a string, it fetches keys matching the given pattern (such as a name or email address). If a sequence of strings, it fetches keys matching at least one of the given patterns.
- **secret** – If true, only secret keys will be returned.

Returns A *KeyIter* instance.

keylist_mode
Default key listing behavior.

Controls which keys *Context.keylist()* returns. The value is a bitwise OR combination of one or multiple of the KEYLIST_MODE_* constants. Defaults to *KEYLIST_MODE_LOCAL*.

passphrase_cb()

pinentry_mode()

progress_cb()

protocol
The protocol used for talking to the backend. Accepted values are one of the PROTOCOL_* constants.

set_engine_info (*protocol*, *executable*, *config_dir*)
Configure a crypto backend.

Updates the configuration of the crypto backend for the given protocol. If this function is used then it must be called before any crypto operation is performed on the context.

Parameters

- **protocol** – One of the PROTOCOL_* constants specifying which crypto backend is to be configured. Note that this does not change which crypto backend is actually used, see *Context.protocol* for that.
- **executable** – The path to the executable implementing the protocol. If None then the default will be used.

- **config_dir** – The path of the configuration directory of the crypto backend. If `None` then the default will be used.

set_locale()

sign(*plaintext*, *signed*, *mode=gpgme.SIG_MODE_NORMAL*)
Sign plaintext to certify and timestamp it.

The plaintext is signed using all keys listed in *Context.signers*.

Parameters

- **plaintext** – A file-like object opened for reading, containing the plaintext to be signed.
- **signed** – A file-like object opened for writing, where the signature data will be written. The signature data may contain the plaintext or not, see the `mode` parameter. If *Context.armor* is false and `mode` is not *SIG_MODE_CLEAR* then the file should be opened in binary mode.
- **mode** – One of the `SIG_MODE_*` constants.

Returns A list of *NewSignature* instances (one for each key in *Context.signers*).

signers
List of *Key* instances used for signing with *sign()* and *encrypt_sign()*.

textmode()

verify(*signature*, *signedtext*, *plaintext*)
Verify signature(s) and extract plaintext.

`signature` is a file-like object opened for reading, containing the signature data.

If `signature` is a normal or cleartext signature (i.e. created using *SIG_MODE_NORMAL* or *SIG_MODE_CLEAR*) then `signedtext` must be `None` and `plaintext` a file-like object opened for writing that will contain the extracted plaintext.

If `signature` is a detached signature (i.e. created using *SIG_MODE_DETACHED*) then `signedtext` should contain a file-like object opened for reading containing the signed text and `plaintext` must be `None`.

Returns A list of *Signature* instances (one for each key that was used in `signature`). Note that you need to inspect the return value to check whether the signatures are valid – a syntactically correct but invalid signature does not raise an error!

2.2 GenkeyResult

class gpgme.**GenkeyResult**
Key generation result.

Instances of this class are usually obtained as the return value of *Context.genkey()*.

fpr
String containing the fingerprint of the generated key. If both a primary and a subkey were generated then this is the fingerprint of the primary key. For crypto backends that do not provide key fingerprints this is `None`.

primary
True if a primary key was generated.

sub
True if a sub key was generated.

2.3 Key

class gpgme.**Key**

revoked
True if the key has been revoked.

expired
True if the key has expired.

disabled
True if the key is disabled.

invalid
True if the key is invalid. This might have several reasons. For example, for the S/MIME backend it will be set during key listing if the key could not be validated due to a missing certificates or unmatched policies.

can_encrypt
True if the key (i.e. one of its subkeys) can be used for encryption.

can_sign
True if the key (i.e. one of its subkeys) can be used to create signatures.

can_certify
True if the key (i.e. one of its subkeys) can be used to create key certificates.

secret
True if the key is a secret key. Note that this will always be true even if the corresponding subkey flag may be false (offline/stub keys). This is only set if a listing of secret keys has been requested or if *KEYLIST_MODE_WITH_SECRET* is active.

can_authenticate
True if the key (i.e. one of its subkeys) can be used for authentication.

protocol
The protocol supported by this key. See the PROTOCOL_* constants.

issuer_serial
If *Key.protocol* is *PROTOCOL_CMS* then this is the issuer serial.

issuer_name
If *Key.protocol* is *PROTOCOL_CMS* then this is the issuer name.

chain_id
If *Key.protocol* is *PROTOCOL_CMS* then this is the chain ID, which can be used to built the certificate chain.

owner_trust
If *Key.protocol* is *PROTOCOL_OpenPGP* then this is the owner trust.

subkeys
List of the key's subkeys as instances of *Subkey*. The first subkey in the list is the primary key and usually available.

uids
List of the key's user IDs as instances of *UserId*. The first user ID in the list is the main (or primary) user ID.

keylist_mode
The keylist mode that was active when the key was retrieved. See *Context.keylist_mode*.

2.4 NewSignature

class gpgme.**NewSignature**
Data for newly created signatures.

Instances of this class are usually obtained as the result value of *Context.sign()* or *Context.encrypt_sign()*.

2.5 Signature

class gpgme.**Signature**
Signature verification data.

Instances of this class are usually obtained as the return value of *Context.verify()* or *Context.decrypt_verify()*.

exp_timestamp
Expiration timestamp of the signature, or 0 if the signature does not expire.

fpr
Fingerprint string.

notations
A list of notation data in the form of tuples `(name, value)`.

status
If an error occurred during verification (for example because the signature is not valid) then this attribute contains a corresponding *GpgmeError* instance. Otherwise it is `None`.

summary
A bit array encoded as an integer containing general information about the signature. Combine this value with one of the `SIGSUM_*` constants using bitwise AND.

timestamp
Creation timestamp of the signature.

validity
Validity of the signature. See *Signature.validity_reason*.

validity_reason
If a signature is not valid this may provide a reason why. See *Signature.validity*.

wrong_key_usage
True if the key was not used according to its policy.

2.6 Helper Objects

Stuff that's mostly used internally, but it's good to know it's there.

class gpgme.**KeyIter**
Iterable yielding *Key* instances for keylist results.

gpgme.**gpgme_version**
Version string of libgpgme used to build this module.

class gpgme.**GpgmeError**

class gpgme.**ImportResult**

class gpgme.**KeySig**

class gpgme.**Subkey**

class gpgme.**UserId**

2.7 Constants

2.7.1 Protocol Selection

The following constants can be used as value for *Context.protocol*. They are also returned via *Key.protocol*.

gpgme.**PROTOCOL_OpenPGP**
This specifies the OpenPGP protocol.

gpgme.**PROTOCOL_CMS**
This specifies the Cryptographic Message Syntax.

gpgme.**PROTOCOL_ASSUAN**
[1] Under development. Please ask on gnupg-devel@gnupg.org for help.

gpgme.**PROTOCOL_G13**
[1] Under development. Please ask on gnupg-devel@gnupg.org for help.

gpgme.**PROTOCOL_UISERVER**
[1] Under development. Please ask on gnupg-devel@gnupg.org for help.

gpgme.**PROTOCOL_SPAWN**
[1] Special protocol for use with gpgme_op_spawn.

gpgme.**PROTOCOL_UNKNOWN**
[1] Reserved for future extension. You may use this to indicate that the used protocol is not known to the application. Currently, GPGME does not accept this value in any operation, though, except for gpgme_get_protocol_name.

2.7.2 Key Listing Mode

Bitwise OR combinations of the following constants can be used as values for *Context.keylist_mode*.

gpgme.**KEYLIST_MODE_LOCAL**
Specifies that the local keyring should be searched. This is the default.

gpgme.**KEYLIST_MODE_EXTERN**
Specifies that an external source should be searched. The type of external source is dependant on the crypto engine used and whether it is combined with *KEYLIST_MODE_LOCAL*. For example, it can be a remote keyserver or LDAP certificate server.

gpgme.**KEYLIST_MODE_SIGS**
Specifies that the key signatures should be included in the listed keys.

gpgme.**KEYLIST_MODE_SIG_NOTATIONS**
[1] Specifies that the signature notations on key signatures should be included in the listed keys. This only works if *KEYLIST_MODE_SIGS* is also enabled.

[1] This constant is defined by the gpgme library, but is currently missing in pygpgme.

gpgme.**KEYLIST_MODE_WITH_SECRET**
[1] Returns information about the presence of a corresponding secret key in a public key listing. A public key listing with this mode is slower than a standard listing but can be used instead of a second run to list the secret keys. This is only supported for GnuPG versions >= 2.1.

gpgme.**KEYLIST_MODE_EPHEMERAL**
[1] Specifies that keys flagged as ephemeral are included in the listing.

gpgme.**KEYLIST_MODE_VALIDATE**
[1] Specifies that the backend should do key or certificate validation and not just get the validity information from an internal cache. This might be an expensive operation and is in general not useful. Currently only implemented for the S/MIME backend and ignored for other backends.

2.7.3 Encryption Flags

Bitwise OR combinations of the following constants can be used for the `flags` parameter of *Context.encrypt()* and *Context.encrypt_sign()*.

gpgme.**ENCRYPT_ALWAYS_TRUST**
Specifies that all the recipients in recp should be trusted, even if the keys do not have a high enough validity in the keyring. This flag should be used with care; in general it is not a good idea to use any untrusted keys.

gpgme.**ENCRYPT_NO_ENCRYPT_TO**
[1] Specifies that no default or hidden default recipients as configured in the crypto backend should be included. This can be useful for managing different user profiles.

gpgme.**ENCRYPT_NO_COMPRESS**
[1] Specifies that the plaintext shall not be compressed before it is encrypted. This is in some cases useful if the length of the encrypted message may reveal information about the plaintext.

gpgme.**ENCRYPT_PREPARE**
[1] Used with the UI Server protocol to prepare an encryption.

gpgme.**ENCRYPT_EXPECT_SIGN**
[1] Used with the UI Server protocol to advise the UI server to expect a sign command.

2.7.4 Signing Modes

The following constants can be used for the `mode` parameter of *Context.sign()*.

gpgme.**SIG_MODE_NORMAL**
A normal signature is made, the output includes the plaintext and the signature. *Context.armor* is respected.

gpgme.**SIG_MODE_DETACHED**
A detached signature is created. *Context.armor* is respected.

gpgme.**SIG_MODE_CLEAR**
A cleartext signature is created. *Context.armor* is ignored.

2.7.5 Signature Verification

The following bit masks can be used to extract individual bits from *Signature.summary* using bitwise AND.

gpgme.**SIGSUM_VALID**
The signature is fully valid.

gpgme.**SIGSUM_GREEN**
The signature is good but one might want to display some extra information. Check the other bits.

gpgme.**SIGSUM_RED**
The signature is bad. It might be useful to check other bits and display more information, i.e. a revoked certificate might not render a signature invalid when the message was received prior to the cause for the revocation.

gpgme.**SIGSUM_KEY_REVOKED**
The key or at least one certificate has been revoked.

gpgme.**SIGSUM_KEY_EXPIRED**
The key or one of the certificates has expired.

gpgme.**SIGSUM_SIG_EXPIRED**
The signature has expired.

gpgme.**SIGSUM_KEY_MISSING**
Can't verify due to a missing key or certificate.

gpgme.**SIGSUM_CRL_MISSING**
The certificate revocation list (or an equivalent mechanism) is not available.

gpgme.**SIGSUM_CRL_TOO_OLD**
The available certificate revocation list is too old.

gpgme.**SIGSUM_BAD_POLICY**
A policy requirement was not met.

gpgme.**SIGSUM_SYS_ERROR**
A system error occured.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

R

S

T

U

V

W