keybase Documentation

Release

Author

January 24, 2015

Contents

1 Installation **3**

2 Examples **5**
2.1 Get a User's Credentials 5
2.2 Verifying a Signature on String Data 5
2.3 Verifying an Embedded Signature on a File 6
2.4 Verify an Detached Signature on a File 7
2.5 Encrypting a Message for a Keybase User 7
2.6 Encrypting a File for a Keybase User 7

3 The Keybase API **9**
3.1 Keybase Common Methods 9
3.2 The `Keybase` Class – Accessing Public User Data 9
3.3 The `KeybasePublicKey` Class – Public Key Records from the Keybase.io Data Store 11
3.4 The `KeybaseAdmin` Class – Manipulating User's Public Key Data 15
3.5 The Keybase Error Classes 15

4 Indices and tables **17**

What is Keybase? From their website:

Note: Keybase will be a public directory of publicly auditable public keys. All paired, for convenience, with unique usernames.

It provides an easy way to publish public keys, have them validated against known good sources for users like Twitter, email addresses and even web sites, and make all of this stuff discoverable. It's trying to take away the mystery of handing keys around so that cryptography can be more widely used by the masses.

The `keybase` python API allows you to search, download and use the stored keys in the Keybase directory. You can do things like encrypt messages and files for a user or verify a signature on a file from a user. Eventually it will be extended to allow you to administer Keybase user identities and their associated public/private keypairs via the `KeybaseAdmin` class.

If you're not familiar with public/private key encryption check out this tutorial or Laurent Luce's excellent article Python and cryptography with pycrypto.

CHAPTER 1

Installation

Simply run:

```
pip install keybase-api
```

CHAPTER 2

Examples

2.1 Get a User's Credentials

You can retrieve a specific user's credentials from the Keybase data store like so:

```
kbase = Keybase('irc')
primary_key = kbase.get_public_key()
primary_key.kid
u'0101f56ecf27564e5bec1c50250d09efe963cad3138d4dc7f4646c77f6008c1e23cf0a'
```

You can use the `ascii` or `bundle` properties on the `primary_key` object in the above example to get an ASCII version of their primary public key, suitable for feeding in to a signature verification or encryption routine. You can also use the `primary_key` object itself to do verification and encryption.

2.2 Verifying a Signature on String Data

Where the strings are clear-signed text strings that are produced using a `gpg` command like so:

```
gpg --clearsign helloworld.txt --local-user keybase.io/irc
```

These clear-signed text snippets are common in signed email. Where the body of the email is surrounded by the signature like so:

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

Hello, world!
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQEcBAEBAgAGBQJTWHSVAAoJEO7zMmcMHMCAYpEH/j2hJApaHXSj0ddgbrmUdJ2z
vZ5DFDR9syTPHrwtRJLPH7tgdiAtUpyXLozL321JIR7sExzONl7IKdpH1Qn0y1I/
h6mV0Dm+AAJXWtbn08rDW2WWuW4+EBEy12Cfk2r1rF8KT+g3gcc2wLejSACkf7v+
jKo5SnvIwIMze+Msqjcz/+hbKRdEEoD2zihe6ilMfbR1tCt8GALQVa8YEoHpgkcL
MWbXSCgM7Q0gf00kHWa3A8rClW0dzW5kJG+InbymtenaDNwoNlFb6DHUdyF//REx
YjJ6qHf7qFwtXPBiwrZf+VYt5OnjeWW6ybYasfrJiXi1qnd6IM40QCGlR0UXhII=
=oUn0
-----END PGP SIGNATURE-----
```

These types of clear-signed strings can be verified like so:

```
message_good = """
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

Hello, world!
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQEcBAEBAgAGBQJTWHSVAAoJEO7zMmcMHMCAYpEH/j2hJApaHXSj0ddgbrmUdJ2z
vZ5DFDR9syTPHrwtRJLPH7tgdiAtUpyXLozL321JIR7sExzONl7IKdpH1Qn0y1I/
h6mV0Dm+AAJXWtbn08rDW2WWuW4+EBEy12Cfk2r1rF8KT+g3gcc2wLejSACkf7v+
jKo5SnvIwIMze+Msqjcz/+hbKRdEEoD2zihe6ilMfbR1tCt8GALQVa8YEoHpgkcL
MWbXSCgM7Q0gf00kHWa3A8rClW0dzW5kJG+InbymtenaDNwoNlFb6DHUdyF//REx
YjJ6qHf7qFwtXPBiwrZf+VYt5OnjeWW6ybYasfrJiXi1qnd6IM40QCGlR0UXhII=
=oUn0
-----END PGP SIGNATURE-----
"""
message_bad = """
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

Hello, another world!
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQEcBAEBAgAGBQJTWHSVAAoJEO7zMmcMHMCAYpEH/j2hJApaHXSj0ddgbrmUdJ2z
vZ5DFDR9syTPHrwtRJLPH7tgdiAtUpyXLozL321JIR7sExzONl7IKdpH1Qn0y1I/
h6mV0Dm+AAJXWtbn08rDW2WWuW4+EBEy12Cfk2r1rF8KT+g3gcc2wLejSACkf7v+
jKo5SnvIwIMze+Msqjcz/+hbKRdEEoD2zihe6ilMfbR1tCt8GALQVa8YEoHpgkcL
MWbXSCgM7Q0gf00kHWa3A8rClW0dzW5kJG+InbymtenaDNwoNlFb6DHUdyF//REx
YjJ6qHf7qFwtXPBiwrZf+VYt5OnjeWW6ybYasfrJiXi1qnd6IM40QCGlR0UXhII=
=oUn0
-----END PGP SIGNATURE-----
"""
kbase = Keybase('irc')
verified = kbase.verify(message_good)
assert verified
verified = kbase.verify(message_bad)
assert not verified
kbase.verify(message_bad, throw_error=True)
Traceback (most recent call last):
...
KeybasePublicKeyVerifyError: signature bad
```

In the `message_bad` case you can see that either the message was tampered with or the signature was faked. In either case you shouldn't trust it because it couldn't be verified correctly.

2.3 Verifying an Embedded Signature on a File

Where the file was signed with a `gpg` command like so:

```
gpg -u keybase.io/irc --sign helloworld.txt
```

So there is one, binary, file `helloworld.txt.gpg` that contains both the data and the signature on the data to verify:

```
kbase = Keybase('irc')
verified = kbase.verify_file('helloworld.txt.gpg')
assert verified
```

2.4 Verify an Detached Signature on a File

Where the file was signed with a `gpg` command like so:

```
gpg -u keybase.io/irc --detach-sign helloworld.txt
```

So there are two files:

1. The original data file; and
2. The detached `.sig` file that contains the signature for the data.

In this case:

```
kbase = Keybase('irc')
fname = 'helloworld.txt'
signame = 'helloworld.txt.sig'
verified = kbase.verify_file(fname, signame)
assert verified
```

2.5 Encrypting a Message for a Keybase User

Given some `str` formatted data, you can create an ASCII armored, encrypted `str` representation of that data suitable for sending to the user. Only someone with the private key, presumably this Keybase user, will be able to decrypt this data:

```
kbase = Keybase('irc')
instring = 'Hello, world!'
encrypted = kbase.encrypt(instring)
assert encrypted
assert not encrypted.isspace()
assert encrypted != instring
```

This ASCII armored approach to encrypting is useful for embedding secret messages in to standard, plaintext communications like emails, tweets or text messages.

2.6 Encrypting a File for a Keybase User

You can create a binary, encrypted file for a user using their Keybase key. Only the user, with their private key, will be able to decrypt the data. The input file contents does not have to be ASCII in this case:

```
kbase = Keybase('irc')
with open('inputfile.bin', 'rb') as infile:
        with open('inputfile.bin.gpg', 'wb') as outfile:
                data = infile.read()
                encrypted_data = kbase.encrypt(data, armor=False)
                outfile.write(encrypted_data.data)
assert os.path.isfile('inputfile.bin.gpg')
```

The user can now decrypt `inputfile.bin.gpg` with:

```
gpg --decrypt inputfile.bin.gpg
```

They will be prompted for the private key’s password.

CHAPTER 3

The Keybase API

3.1 Keybase Common Methods

The following common, convenience methods exist to make it easier to work with GnuPG and the Keybase API in your code.

`keybase.`**`gpg`**`(`*binary=None*`)`

Returns the full path to the gpg instance on this machine. It prefers `gpg2` but will search for `gpg` if it cannot find `gpg2`.

```
>>> len(gpg()) > 0
True
>>> len(gpg('gpg')) > 0
True
```

I implemented this because the `gnupg.GPG` class was having a hard time dealing with the fact that my Homebrew-installed GPG instance was a symlink in the `/usr/local/bin` directory instead of a real path to a real file.

If you want to use a binary with a specific name, supply the `binary=bName` option when you call `gpg()` and it will use your custom binary name instead.

On windows you shouldn't need to supply an extension to the command like `.exe` or `.cmd` – it will figure it out for you.

Returns `None` if it cannot find a gpg2 or gpg instance in your PATH:

```
>>> gpg('notagpgbinary')
```

3.2 The `Keybase` Class – Accessing Public User Data

The `Keybase` class allows you to find users in the Keybase directory and access their stored public keys. Public keys let you encrypt messages and files for a user; only the person holding the private key from the pair can decrypt a file encrypted with the public key. Public keys also let you verify the signature on data; only the user with the private key can create a signature that can be validated with the specific public key.

class `keybase.`**`Keybase`**`(`*username*`)`

A read-only view of a keybase.io user and their publically available keys. This class allows you to do interesting things with someone's public key data like encrypt a message for them or verify that a message they signed to you was actually signed by them.

The public information is automatically retrieved when you build a new instance of the class.

```
>>> kbase = Keybase('irc')
>>> kbase.username
'irc'
```

If the user cannot be found a `keybase.KeybaseUserNotFound` exception is raised:

```
>>> kbase = Keybase('abcdefghijklmno123notauserhahaha')
Traceback (most recent call last):
...
KeybaseUserNotFound: User abcdefghijklmno123notauserhahaha not found
```

Note: It does not allow you to manipulate the key data in the keybase.io data store in any way.

encrypt(*data*, ***kwargs*)
Equivalent to:

```
kbase = Keybase('irc')
pkey = kbase.get_public_key()
verified = pkey.encrypt(data, **kwargs)
assert verified
```

It's a convenience method on the Keybase object to do data verification with the primary key.

For more information see `keybase.KeybasePublicKey.encrypt`.

get_public_key(*keyname='primary'*)
Returns a key named keyname as a `keybase.KeybasePublicKey` object if it exists in the current Keybase instance. Defaults to a key named `primary` if you opt not to supply a keyname when you call the method.

```
>>> kbase = Keybase('irc')
>>> primary_key = kbase.get_public_key()
>>> primary_key.kid
u'0101f56ecf27564e5bec1c50250d09efe963cad3138d4dc7f4646c77f6008c1e23cf0a'
```

Otherwise it returns None if a key by the name of keyname doesn't exist for this user.

```
>>> kbase.get_public_key('thiskeydoesnotexist')
```

location
The geographical location of the person associated with this Keybase data.

```
>>> k = Keybase('irc')
>>> k.location
u'Bay Area, California'
```

name
The full name of the person associated with this Keybase data.

```
>>> k = Keybase('irc')
>>> k.name
u'Ian Chesal'
```

public_keys
A tuple of all the public keys available for this account. An empty tuple is returned if the instance isn't bound to a user or the user has no keys.

```
>>> kbase = Keybase('irc')
>>> kbase.public_keys
(u'families', u'primary', u'sibkeys', u'subkeys')
```

username

The username of the person associated with this Keybase data.

```
>>> k = Keybase('irc')
>>> k.username
'irc'
```

verify (*data*, *throw_error=False*)

Equivalent to:

```
kbase = Keybase('irc')
pkey = kbase.get_public_key()
verified = pkey.verify(some_message)
assert verified
```

It's a convenience method on the Keybase object to do data verification with the primary key.

For more information see `keybase.KeybasePublicKey.verify`.

verify_file (*fname*, *sigfname=None*, *throw_error=False*)

Equivalent to:

```
kbase = Keybase('irc')
pkey = kbase.get_public_key()
verified = pkey.verify_file(fname, signame)
assert verified
```

It's a convenience method on the Keybase object to do data verification with the primary key.

For more information see `keybase.KeybasePublicKey.verify_file`.

3.3 The `KeybasePublicKey` Class – Public Key Records from the Keybase.io Data Store

class `keybase.`**`KeybasePublicKey`** (***kwargs*)

A class that represents the public key side of a public/private key pair.

It is tied very closely to the keybase.io data that's stored for public keys in user profiles in the data store. As such, it's meant to be initialized with a hash that contains the fields seen in a keybase.io public key record.

Under the hood it uses GnupGP's `gnupg.GPG` class to do the heavy lifting. It creates a keystore that is unique to this instance of the class and loads the public key in to this keystore.

You won't be able to decrypt with this class because it only contains a public key, not a private key. But you can encrypt and and sign:

```
>>> kbase = Keybase('irc')
>>> pkey = kbase.get_public_key()
>>> pkey.key_fingerprint
u'7cc0ce678c37fc27da3ce494f56b7a6f0a32a0b9'
```

If a valid GPG instance cannot be created when you initialize a KeybasePublicKey a KeybasePublicKeyError will be raised.

ascii

Synonym for bundle property.

bundle

The GPG key bundle. This is the ASCII representation of the public key data associated with the Keybase key.

cipher_algos

Returns a tuple of available cypher algorithms that you can use with this key to encrypt data. The available algorithms depend entirely on the GPG version installed on the machine though most, if not all GPG versions, support `AES256`.

```
>>> kbase = Keybase('irc')
>>> pkey = kbase.get_public_key()
>>> 'AES256' in pkey.cipher_algos
True
```

compress_algos

Returns a tuple of available compression algorithms that you can use with this key to compress encrypted data. The available algorithms depend entirely on the GPG version installed on the machine though most, if not all GPG versions, support `ZIP`.

```
>>> kbase = Keybase('irc')
>>> pkey = kbase.get_public_key()
>>> 'ZIP' in pkey.compress_algos
True
```

ctime

The datetime this key was created in the keybase database.

digest_algos

Returns a tuple of available digest algorithms that you can use with this key to hash data. The available algorithms depend entirely on the GPG version installed on the machine though most, if not all GPG versions, support `SHA512`.

```
>>> kbase = Keybase('irc')
>>> pkey = kbase.get_public_key()
>>> 'SHA512' in pkey.digest_algos
True
```

encrypt (*data*, *armor=True*, *cipher_algo=None*, *digest_algo=None*, *compress_algo=None*)

Encrypt the message contained in the string `data` for the owner of this KeybasePublicKey instance.

If `armor=True` the output is ASCII armored; otherwise the output will be a gnupg._parsers.Crypt object.

If encryption fails a KeybasePublicKeyEncryptError is raised.

If it succeeds data object is returned. Assuming `armor=True` the returned data is just plain old ASCII text as a `str()`.

Note: The remaining options are supplied for maximum flexibility with GPG but you can, for the most part, just ignore them and go with the defaults if you want the simpilest (but still secure) path to encrypting data with this API.

If `cipher_algo` is supplied it should be the name of a cipher algorithm to use. The default algorithm is `AES256` and you can get a list of available algorithms from the `keybase.KeybasePublicKey.crypto_algos()` parameter.

If `digest_algo` is supplied it should be the name of a digest algorithm to use. The default is `SHA512` and you can get a list of available algorithms from the `keybase.KeybasePublicKey.digest_algos()` parameter.

If `compress_algo` is supplied it should be the name of a compression algorithm to use. The default is `ZIP` and you can get a list of available algorithms from the `keybase.KeybasePublicKey.compress_algos()` parameter.

For more information on how encryption works please see the `gnupg.encrypt` manual page.

A simple example:

```
kbase = Keybase('irc')
pkey = kbase.get_public_key()
instring = 'Hello, world!'
encrypted = pkey.encrypt(instring)
assert encrypted
assert not encrypted.isspace()
assert encrypted != instring
```

key_fingerprint
The GPG fingerprint for the key.

key_type
The Keybase key type for this key (integer).

kid
The Keybase key ID for this key.

mtime
The datetime this key was last modified in the Keybase database.

ukbid
The UKB ID for the key.

verify(*data*, *throw_error=False*)
Verify the signature on the contents of the string `data`. Returns True if the signature was verified with the key, False if it was not. If you supply `throw_error=True` to the call then it will throw a KeybasePublicKeyVerifyError on verification failure with a status message that tells you more about why verification failed.

Failure status messages are:

- invalid gpg key
- signature bad
- signature error
- decryption failed
- no public key
- key exp
- key rev

For more information what these messages mean please see the `gnupg._parsers.Verify` manual page.

```
>>> message_good = """
... -----BEGIN PGP SIGNED MESSAGE-----
... Hash: SHA1
...
... Hello, world!
... -----BEGIN PGP SIGNATURE-----
... Version: GnuPG v1
...
... iQEcBAEBAgAGBQJTWHSVAAoJEO7zMmcMHMCAYpEH/j2hJApaHXSj0ddgbrmUdJ2z
... vZ5DFDR9syTPHrwtRJLPH7tgdiAtUpyXLozL321JIR7sExzON17IKdpH1Qn0y1I/
... h6mV0Dm+AAJXWtbn08rDW2WWuW4+EBEy12Cfk2r1rF8KT+g3gcc2wLejSACkf7v+
... jKo5SnvIwIMze+Msqjcz/+hbKRdEEoD2zihe6ilMfbR1tCt8GALQVa8YEoHpgkcL
... MWbXSCgM7Q0gf00kHWa3A8rClW0dzW5kJG+InbymtenaDNwoNlFb6DHUdyF//REx
... YjJ6qHf7qFwtXPBiwrZf+VYt5OnjeWW6ybYasfrJiXi1qnd6IM40QCGlR0UXhII=
... =oUn0
... -----END PGP SIGNATURE-----
... """
>>> message_bad = """
... -----BEGIN PGP SIGNED MESSAGE-----
... Hash: SHA1
...
... Hello, another world!
... -----BEGIN PGP SIGNATURE-----
... Version: GnuPG v1
...
... iQEcBAEBAgAGBQJTWHSVAAoJEO7zMmcMHMCAYpEH/j2hJApaHXSj0ddgbrmUdJ2z
... vZ5DFDR9syTPHrwtRJLPH7tgdiAtUpyXLozL321JIR7sExzONl7IKdpH1Qn0y1I/
... h6mV0Dm+AAJXWtbn08rDW2WWuW4+EBEy12Cfk2r1rF8KT+g3gcc2wLejSACkf7v+
... jKo5SnvIwIMze+Msqjcz/+hbKRdEEoD2zihe6ilMfbR1tCt8GALQVa8YEoHpgkcL
... MWbXSCgM7Q0gf00kHWa3A8rClW0dzW5kJG+InbymtenaDNwoNlFb6DHUdyF//REx
... YjJ6qHf7qFwtXPBiwrZf+VYt5OnjeWW6ybYasfrJiXi1qnd6IM40QCGlR0UXhII=
... =oUn0
... -----END PGP SIGNATURE-----
... """
>>> kbase = Keybase('irc')
>>> pkey = kbase.get_public_key()
>>> verified = pkey.verify(message_good)
>>> assert verified
>>> verified = pkey.verify(message_bad)
>>> assert not verified
>>> pkey.verify(message_bad, throw_error=True)
Traceback (most recent call last):
...
KeybasePublicKeyVerifyError: signature bad
```

If you want to verify the signature on a file (either embedded or detached) please see `keybase.KeybasePublicKey.verify_file()` method.

`verify_file` (*fname*, *sigfname=None*, *throw_error=False*)

Verify the signature on a file named `fname`. This is a string file name, not a file object. If only a `fname` is provided the method assumes the signature is embedded in the file itself. An embedded signature is usually produced like so:

```
gpg -u keybase.io/irc --sign helloworld.txt
```

If a `sigfname` argument is prodived it's assumed to be a path to signature file for a detached signature. A detached signature is usually produced like so:

```
gpg -u keybase.io/irc --detach-sign helloworld.txt
```

Returns True if the signature is verifiable with the key, False if it is not verifiable.

If you supply the `throw_error=True` option to the call then it will throw a KeybasePublicKeyVerify-Error on verification failure with a status message that tells you more about why the verification failed.

Failure status messages are:

- invalid gpg key
- signature bad
- signature error
- decryption failed

- no public key
- key exp
- key rev

For more information what these messages mean please see the `gnupg._parsers.Verify` manual page.

An embedded signature example:

```
kbase = Keybase('irc')
pkey = kbase.get_public_key()
verified = pkey.verify_file('helloworld.txt.gpg')
assert verified
```

A detached signature example:

```
kbase = Keybase('irc')
pkey = kbase.get_public_key()
fname = 'helloworld.txt'
signame = 'helloworld.txt.sig'
verified = pkey.verify_file(fname, signame)
assert verified
```

3.4 The `KeybaseAdmin` Class – Manipulating User's Public Key Data

The `KeybaseAdmin` class lets you authenticate as a user to the Keybase.io public data store and manipulate the stored public keys for the user. You can add and revoke keys, create new keys and validate other user's keys.

Note: This class is currently not implemented! Anything you read here is planned, not real, at this point.

3.5 The Keybase Error Classes

class `keybase.`**`KeybaseError`**
General error class for Keybase errors.

class `keybase.`**`KeybaseUnboundInstanceError`**
Thrown when calling a Keybase object method that requires the object be bound to a real user in the keybase store and the instance hasn't had such a binding established yet.

class `keybase.`**`KeybaseUserNotFound`**
Thrown when calling Keybase.lookup(username) and the username cannot be located in the keybase.io public key repository.

class `keybase.`**`KeybaseLookupInvalidError`**
Thrown when calling Keybase.lookup(username) on an instance that has already been bound to a valid user via another lookup() call.

class `keybase.`**`KeybasePublicKeyError`**
Thrown when a KeybasePublicKey cannot be created successfully.

class `keybase.`**`KeybasePublicKeyVerifyError`**
Thrown when a KeybasePublicKey cannot verify the signature on a data object.

class `keybase.`**`KeybasePublicKeyEncryptError`**
Thrown when a KeybasePublicKey cannot perform encryption on some data object.

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*

Index

A

ascii (keybase.KeybasePublicKey attribute), 11

B

bundle (keybase.KeybasePublicKey attribute), 11

C

cipher_algos (keybase.KeybasePublicKey attribute), 12
compress_algos (keybase.KeybasePublicKey attribute), 12
ctime (keybase.KeybasePublicKey attribute), 12

D

digest_algos (keybase.KeybasePublicKey attribute), 12

E

encrypt() (keybase.Keybase method), 10
encrypt() (keybase.KeybasePublicKey method), 12

G

get_public_key() (keybase.Keybase method), 10
gpg() (in module keybase), 9

K

key_fingerprint (keybase.KeybasePublicKey attribute), 13
key_type (keybase.KeybasePublicKey attribute), 13
Keybase (class in keybase), 9
KeybaseError (class in keybase), 15
KeybaseLookupInvalidError (class in keybase), 15
KeybasePublicKey (class in keybase), 11
KeybasePublicKeyEncryptError (class in keybase), 15
KeybasePublicKeyError (class in keybase), 15
KeybasePublicKeyVerifyError (class in keybase), 15
KeybaseUnboundInstanceError (class in keybase), 15
KeybaseUserNotFound (class in keybase), 15
kid (keybase.KeybasePublicKey attribute), 13

L

location (keybase.Keybase attribute), 10

M

mtime (keybase.KeybasePublicKey attribute), 13

N

name (keybase.Keybase attribute), 10

P

public_keys (keybase.Keybase attribute), 10

U

ukbid (keybase.KeybasePublicKey attribute), 13
username (keybase.Keybase attribute), 10

V

verify() (keybase.Keybase method), 11
verify() (keybase.KeybasePublicKey method), 13
verify_file() (keybase.Keybase method), 11
verify_file() (keybase.KeybasePublicKey method), 14