Age/Minisign Wrapper for Python Documentation

Release 0.1.0

Vinay Sajip

Dec 05, 2021

CONTENTS

Release:	0.1.0
Date:	Dec 05, 2021

The `pagesign` (for 'Python-age-sign') module allows Python programs to make use of the functionality provided by age and minisign. Using this module, Python programs can encrypt and decrypt data, digitally sign documents and verify digital signatures, manage (generate, list and delete) encryption and signing keys.

This module is expected to be used with Python versions >= 3.6. Install this module using `pip install pagesign`. You can then use this module in your own code by doing `import pagesign` or similar.

CHAPTER

ONE

DEPLOYMENT REQUIREMENTS

Apart from a recent-enough version of Python, in order to use this module you need to have access to a compatible versions of *age-keygen*, *age* and *minisign* executables. The system has been tested with *age* v1.0.0 and *minisign* v0.8 on Windows, macOS and Ubuntu.

CHAPTER

TWO

ACKNOWLEDGEMENTS

The `pagesign` module follows a similar approach to python-gnupg (by the same author), and uses Python's `subprocess` module to communicate with the *age* and *minisign* executables, which it uses to spawn subprocesses to do the real work of encryption, decryption, signing and verification.

CHAPTER

THREE

BEFORE YOU START

pagesign works on the basis of a "home directory" which is used to store public and private key data. (Whereas *age* and *minisign* will save created keys in files for you, but nothing beyond that, *pagesign* will allow you to refer to *identities* using simple names). The directory on POSIX systems is *~/.pagesign* and on Windows is *%LOCALAPPDATA%\pagesign*. If this directory doesn't exist, it is created. On POSIX, its permissions are set so only the owner has full access, and everyone else has no access (permission mask of octal 0700).

This directory will contain an identity store (called *keystore* from now on, as it mainly holds keys). On POSIX, its permissions are set so only the owner has full access, and everyone else has no access (permission mask of octal 0600).

CHAPTER

FOUR

GETTING STARTED

You interface to the *age* and *minisign* functionality through the following items in the *pagesign* module:

- The `Identity` class.
- The *encrypt*, *decrypt*, ‘sign` and *verify* functions.

CHAPTER

FIVE

IDENTITY MANAGEMENT

The *Identity* class represents an identity, which can either be a local identity (which has access to secret keys and passphrases in order to decrypt and sign things) or a remote identity (which only has public keys, so it can only be used to encrypt and verify things).

A remote identity consists of:

- A string indicating the creation time of the identity in *YYYY-mm-ddTHH:MM:SSZ* format.
- A public key (from *age*) for encrypting files.
- A public key (from *minisign*) for verifying file signatures.
- A signature ID (from *minisign*) - this is not currently used.

A local identity, in addition to the above, contains:

- A secret key (from *age*) for decrypting files.
- A secret key (from *minisign*) for signing files.
- A passphrase (created automatically by *pagesign* and used for signing).

These are stored in attributes of an *Identity* instance named *created*, *crypt_public*, *sign_public*, *sign_id*, *crypt_secret*, *sign_secret* and *sign_pass*. Creation of a local identities generates four keys - two secret and two public, two for encryption/decryption and two for signing/verification.

5.1 Generating identities

To create a new local identity, you simply call

```
from pagesign import Identity
identity = Identity()
```

Once you've called this, the identity is in memory, but not saved anywhere. To save it, you call its *save()* method with a name - just a string you choose. It could be a simple identifier like *alice* or *bob*, or an email address.

```
identity.save('bob')
```

This saves the identity under the name *bob*. To get it back at a later time, pass it to the *Identity* constructor:

```
bob = Identity('bob')
```

The *save()* method saves the local identity in a keystore which is stored in the *pagesign* home directory mentioned earlier. Passing that name to the constructor just retrieves it from the store. If you pass a name that's not in the keystore, you will get an error.

The keystore is currently just a plaintext file in JSON format. It relies on directory and file permissions for keeping your secret keys secret.

5.2 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; keys should usually 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 can cause key generation to be 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).

5.3 Exporting identities

You can export the public parts of an identity to send to someone. To do this, you call the *export()* method of an instance:

```
exported = identity.export()
```

This returns a dictionary which contains the public attributes of the identity, whose keys are the attribute names mentioned earlier.

5.4 Importing identities

If you receive a dictonary representing an exported identity from someone, you can import it into your local keystore by calling the class method *Identity.imported()*:

```
alice = Identity.imported(sent_by_alice, 'alice')
```

This saves the remote identity in the keystore with the given name. You (*bob*, say) can use this when exchanging information with *alice*.

5.5 Deleting identities

If you want to completely get rid of an identity, you can call the *remove_identities()* function. To remove all identities from the keystore, the *clear_identities()* function is used.

```
from pagesign import remove_identities, clear_identities

remove_identities('bob', 'alice')  # removes just these two
clear_identities()  # removes everything
```

There is no way to undo these operations, so be careful!

5.6 Listing identities

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

```
from pagesign import list_identities

identities = list_identities()
```

This returns an iterable of *(name, info)* tuples in random order. The *name* is the identity name, and the *info* is a dictionary of all the identity attributes for that identity.

5.7 The *Identity* class

The *Identity* class API is here:

class Identity

Attributes

created
This attribute is a string indicating when the identity was created.

crypt_public
This attribute is the public key used for encryption.

sign_public
This attribute is the public key used for signature verification.

sign_id
This attribute is a key ID which is generated by *minisign* but not currently used in *pagesign*.

sign_pass
This attribute is a passphrase automatically generated by *pagesign* and used for signing. It should not be shared with the wrong people, else they could impersonate you when signing stuff.

crypt_secret
This attribute is the private key used for decryption. It should not be shared with the wrong people, else they can decrypt stuff meant only for you.

sign_secret
This attribute is the private key used for signing. It should not be shared with the wrong people, else they could impersonate you when signing stuff.

Methods

__init__(*name=None*)
If *name* is specified, create an instance populated from data in the keystore associated with that name. Otherwise, create a new instance with autogenerated keys for signing and encryption (the key generation takes half a second). To persist such an instance, call its `save()` method with a name of your choice.

export()
Rrturn the public elements of this instance as a dictionary. The dictionary keys match the attribute names listed earlier.

save(*name*)
Save this instance as a dictionary in the keystore against *name*, overwriting any existing data under that name.

classmethod imported(*public_data*)

This is a factory method which generates an *Identity* instance from the dictionary *public_data*. The instance isn't saved in your keystore until you call its *save()* method with a name of your choice.

CHAPTER

SIX

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. A recipient is denoted by a local or remote identity.

6.1 Encryption

To encrypt a message, use the *encrypt* function:

encrypt(*path*, *recipients*, *outpath=None*, *armor=True*)

Encrypt a file at *path* to *outpath*. If *outpath* isn't specified, the value of *path* with *'.age'* appended is used. If *armor* is *True*, the output file is PEM encoded. The *recipients* can be a single identity name or a list or tuple of identity names. The encrypted file will be decryptable by any of the recipient identities.

The function returns *outpath* if successful and raises an exception if not.

Note: Although *age* supports encryption and decryption using passphrases, that is currently not supported here because there is currently no way to pass in a passphrase to *age* using a subprocess pipe.

6.2 Decryption

To decrypt a message, use the *decrypt* function:

decrypt(*path*, *identities*, *outpath=None*)

Decrypt a file at *path* to *outpath*. If *outpath* isn't specified, then if *path* ends with *.age*, it is stripped to compute *outpath* - otherwise it has *'.dec'* appended to determine *outpath*. The *identities* can be a single identity name or a list or tuple of identity names.

The function returns *outpath* if successful and raises an exception if not.

CHAPTER

SEVEN

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, use the *sign()* function:

sign(*path*, *identity*, *outpath=None*)

Sign the file at *path* using *identity* as the signer. Write the signature to *outpath*. If *outpath* isn't specified, it is computed by appending *'.sig'* to *path*.

The function returns *outpath* if successful and raises an exception if not.

7.2 Verification

To verify some data which you've received, use the *verify()* function:

verify(*path*, *identity*, *sigpath=None*)

Verify that the file at *path* was signed by *identity* using the signature in *sigpath*. If *sigpath* isn't specified, it is computed by appending *'.sig'* to *path*.

The function raises an exception if verification fails.

CHAPTER

EIGHT

COMBINING OPERATIONS

Often, you want to combine encryption and signing, or verification before decryption.

8.1 Using signing and encryption together

If you want to use signing and encryption together, use *encrypt_and_sign()*:

encrypt_and_sign(*path*, *recipients*, *signer*, *armor=True*, *outpath=None*, *sigpath=None*)

Encrypt and sign the file at *path* for *recipients* and sign with identity *signer*. Place the encrypted output at *outpath* and the signature in *sigpath*.

If *armor* is *True*, the encrypted output is PEM encoded.

If *outpath* isn't specified, it is computed by appending *'.age'* to *path*. If *sigpath* isn't specified, it is computed by appending *'.sig'* to *outpath*.

The function returns *(outpath*, sigpath)` if successful and raises an exception if not.

8.2 Using verification and decryption together

As a counterpart to *encrypt_and_sign()*, there's also *verify_and_decrypt()*:

verify_and_decrypt(*path*, *recipients*, *signer*, *outpath=None*, *sigpath=None*)

Verify and decrypt the file at *path* for *recipients* and signed with identity *signer*. Place the decrypted output at *outpath* and use the signature in *sigpath*.

If *sigpath* isn't specified, it is computed by appending *'.sig'* to *path*. If *outpath* isn't specified, it is computed as in `decrypt()`.

The function returns *outpath* if successful and raises an exception if not.

CHAPTER

NINE

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 `pagesign` unless you rename the module.

CHAPTER

TEN

TEST HARNESS

The distribution includes a test harness, `test_pagesign.py`, which contains unit tests covering the functionality described above.

CHAPTER

ELEVEN

DOWNLOAD

The latest version is available from the PyPI page.

CHAPTER

TWELVE

STATUS AND FURTHER WORK

The `pagesign` module is quite usable, though in its early stages and with the API still a little fluid. How this module evolves will be determined by feedback from its user community.

If you find bugs and want to raise issues, or want to suggest improvements, please do so here.

All feedback will be gratefully received.

CHAPTER

THIRTEEN

INDEX

- genindex

PYTHON MODULE INDEX

INDEX

Symbols

A

C

D

E

G

I

K

L

M

P

S

V