BatzenCA Documentation

Release 0.1

Martin R. Albrecht

May 15, 2014

BatzenCA is a set of Python of classes and functions that ought to make managing OpenPGP keys easier for certification authorities.

CHAPTER 1

User-Case

A group of users want to use a mailing list but with OpenPGP encrypted messages. They don't want the server to be able to decrypt their messages either. An easy ad-hoc way of accomplishing this is by every user encrypting to every other user. This can easily accomplished using e.g. Thunderbird/Enigmail's "Per-Recipient Rules".

As the group grows, verifying each other's OpenPGP keys becomes tedious. Our group of users choose not to use the Web of Trust, say because they have a clear definition who belongs on their list and who doesn't. Instead, they nominate a user or a group of users as a Certification Authority (CA), so they are actually doing the X.509 thing with OpenPGP: all users verify the CA's key and grant it full owner trust. The CA then checks new users' identities, verifies their keys, signs and distributes them. When users leave the group the CA revokes its signature. To update the users of our mailing list the CA sends (ir)regular "releases" which contain all keys for those users active on our mailing list. The remaining users import these keys and to update their per-recipient rules to reflect these changes. In a nutshell: a poor person's CA using OpenPGP.

This library makes the job of the CA easier by providing means to prepare such releases.

CHAPTER 2

Library Overview

The purpose of this library is to distribute OpenPGP keys in releases (`batzenca.database.releases.Release`). These releases contain active and inactive keys (`batzenca.database.keys.Key`) one for each user (`batzenca.database.peers.Peer`). Active are those keys which users ought to use, while inactive keys are those where the signature was revoked etc. Releases are meant for specific mailinglists (`batzenca.database.mailinglists.MailingList`). Each mailinglist furthermore has a policy (`batzenca.database.policies.Policy`) which specifies what kind of PGP keys are acceptable - for example, it might specify that keys must expire every 2 years.

CHAPTER 3

Prerequisites

- BatzenCA relies on PyMe 0.9.0 for talking to GnuPG.

  Note that an abandoned branch is available which attempts to switch to the newer PyGPGME is available on Bitbucket. It was abandoned because PyGPGME does not provide an interface to all GPGME functions needed by BatzenCA.
- BatzenCA uses SQLAlchemy to talk to a SQLite database which stores all metadata about keys such as users, releases, mailing lists, policies etc.
- BatzenCA uses GitPython to take snapshots of its database and the internal GnuPG directory.

CHAPTER 4

Installation

The easiest way to install all required Python packages is:

```
pip install -r requirements.txt
```

CHAPTER 5

Alternatives

Alternatives to realising OpenPGP encrypted mailinglists include

- **Schleuder** "Schleuder is a gpg-enabled mailinglist with remailer-capabilities. It is designed to serve as a tool for group communication: subscribers can communicate encrypted (and pseudonymously) among themselves, receive emails from non-subscribers and send emails to non-subscribers via the list. Schleuder takes care of all de- and encryption, stripping of headers, formatting conversions, etc. Further schleuder can send out its own public key upon request and receive administrative commands by email." – http://schleuder2.nadir.org/ Hence, users must trust that the server has not been compromised.
- **SELS** "Secure Email List Services (SELS) is an open source software for creating and developing secure email list services among user communities. SELS provides signature and encryption capabilities while ensuring that the List Server does not have access to email plain text. SELS has been developed with available open-source components and is compatible with many commonly used email clients." – http://sels.ncsa.illinois.edu/ However, the project is discontinued.

CHAPTER 6

Full Documentation

The full documentation of BatzenCA is available at http://batzenca.readthedocs.org.

CHAPTER 7

Table of Contents

7.1 Example

We give a minimal example to get started.

7.1.1 The CA Key

We create a key (`batzenca.database.keys.Key`) for the Certification Authority (CA). First, we generate the GnuPG key:

```
>>> from batzenca import *
>>> session.gnupg.import_secret_key("filename.sec") # not implemented yet, use gpg directly
```

Alternatively, we can generate a fresh secret key for the CA:

```
>>> from batzenca.util import import_secret_key
>>> session.gnupg.generate_secret_key() # not implemented yet, use gpg directly
```

7.1.2 The Policy

Now, we create is a policy (`batzenca.database.policies.Policy`) which specifies constraints on keys on our mailing list. In our case keys must have at least 2048 bits, we only accept RSA and keys must expire within 720 days of creation:

```
>>> import datetime
>>> algorithms = (session.gnupg.GPGME_PK_RSA,)
>>> policy = Policy("main policy", datetime.date(2014,1,1), ca=ca, 2048, 720, algorithms)
```

7.1.3 The Mailing List

We can now create a mailing list object (`batzenca.database.mailinglists.MailingList`). Mailing lists have few message templates attached to it. In particular, a message (template) may be provided which is sent when a new user (`batzenca.database.peers.Peer`) joins the list (`batzenca.database.mailinglists.MailingList.new_member_msg`) which is rendered by `batzenca.database.releases.Release.welcome_messages`:

```
Hello {peer},

you are now subscribed to {mailinglist} <{mailinglist_email}>.

Best regards,
{ca} <{ca_email}>
```

Secondly, we need a message (`batzenca.database.mailinglists.MailingList.key_update_msg`) template for actual releases (`batzenca.database.releases.Release`) of bundles of keys (`batzenca.database.keys.Key`) which is rendered by `batzenca.database.releases.Release.__call__`:

```
Hello {mailinglist},

Users
-----

The following people joined:

{peers_in}

The following people have a new key:

{peers_changed}

The following people have left this list:

{peers_out}

Keys
----

The following keys are new:

{keys_in}

The following keys are no longer to be used:

{keys_out}

The complete list of all keys to be used is:

{keys}

{dead_man_switch}

Best regards,
{ca} <{ca_email}>
```

For the meaning of these fields see (`batzenca.database.mailinglists.MailingList`). Note, that the he expansion of `{peers_in}` and `{peers_out}` is a comma separated list of peers, while `{keys*}` is structured by line breaks.

Thirdly, a message template can be provided which is turned into a message sent to users when their keys are about to expire by `batzenca.database.releases.Release.key_expiry_warning_messages`:

```
Hello {peer},

your key with key id {keyid} is going to expire on {expiry_date}.
```

```
This key is used to encrypt messages for you on {mailinglist} <{mailinglist_email}>.

Please provide a new key to continue receiving messages on this list.

Best regards,
{ca} <{ca_email}>
```

Finally, a dead man switch message may be provided. This message is used to replace the field `{dead_man_switch}` in `batzenca.database.mailinglists.MailingList.key_update_msg` if `still_alive = True` when calling `batzenca.database.releases.Release.__call__`:

```
This CA has not received any requests to disclose and/or modify any data for this
mailinglist. Watch this space for this message to disappear.
```

With these in place, we can construct our `batzenca.database.mailinglists.MailingList` object (split over multiple lines for readability):

```
>>> ml = MailingList(name="batzenca", email="batzenca@thereisnohost.thereisnodomain", policy=policy)
>>> ml.new_member_msg = new_member_msg
>>> ml.key_update_msg = key_update_msg
>>> ml.key_expiry_warning_msg = dead_man_switch_msg
>>> ml.dead_man_switch_msg = dead_man_switch_msg
```

7.1.4 The Peers

Users can have multiple keys over time. This is addressed by creating `batzenca.database.peers.Peer` objects which point to all the keys of a particular user:

```
>>> keys = Key.from_filename("hansolo.asc")
>>> keys[0].sign(ca)
>>> han = Peer("Han Solo", keys)
```

7.1.5 The Releases

Finally, we can create an actual release (`batzenca.database.releases.Release`) which is what we send out to our users:

```
>>> rel = ml.new_release()
>>> rel.add_key(han.key)
```

7.1.6 The Database

Nothing what we've done so far was added to the database, except for the operations with GnuPG. To add our objects (keys, peers, mailing lists, releases) to the database we have to add it:

```
>>> session.add(ml)
```

We only need to add our mailing list as it points to all other objects created so far and `add` automatically cascades. However, this still isn't persisent. We need to commit our changes to the database:

```
>>> session.commit()
```

7.2 GnuPG Interface

7.2.1 Where is my Data?

BatzenCA does not interfere with your normal OpenPGP public-key or secret-key ring. Instead, it uses an independent GnuPG key database or keyring. By default it is located in `$HOME/.batzenca/gnupg`. This default can be changed by setting the environment variable `BATZENCADIR`. If `BATZENCA` is set, then the GnuPG keyring will be located in `$BATZENCADIR/gnupg`. The currently used home dir can be queried as follows:

```
>>> session.gnupg.home_dir
```

Note: Changing `session.gnupg.home_dir` has no effect.

To work with the GnuPG directly you can run:

```
$ gpg --homedir=$HOME/.batzenca/gnupg
```

7.2.2 GnuPG

7.2.3 PGP/MIME

7.3 Database

7.3.1 Where is my Data?

Peers, keys, policies, mailinglists and releases are stored in an SQLite database. By default it is located in `$HOME/.batzenca/batzenca.db`. This default can be changed by setting the environment variable `BATZENCADIR`. If `BATZENCA` is set, then the database will be located at `$BATZENCADIR/batzenca.db`.

7.3.2 Design

- New objects of type `Foo` are created from scratch using `Foo.__init__`. For querying the database class methods like `Foo.from_bar` are provided which query the database for `bar` and return – if found – the object of type `Foo` matching `bar`. Some classes have additional class methods for creating fresh objects from scratch. For example, `batzenca.database.keys.Key` has methods to create new objects from PGP keys stored in a file on disk.
- Newly created objects are were not added to the current session automatically. The caller is responsible for calling:

  ```
  >>> session.add(obj)
  ```

- To save changes to the database the caller must call:

  ```
  >>> session.commit()
  ```

7.3.3 EntryNotFound

7.3.4 Key

7.3.5 Peer

7.3.6 Policy

7.3.7 MailingList

7.3.8 Release

CHAPTER 8

Indices and tables

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